ECE419 Milestone 3 Design Document

Team 24: Chenhao Liu, Jue Liu, Ze Ming Xiang

(All members have equal contribution)

1 Overview

We implemented a replicated, scalable, high-performance, distributed key-value storage server application based on our Milestone 2 codebase. The new features, improvements, and modifications added to our Milestone 2 implementation are documented in the sections below. This iteration or our application features failure detection and recovery, as well as data replication and eventual consistency.

Additionally, see `README.md` in the source code for instructions on environment setup.

1.1 Failure Detection

Once a KVServer is launched, it will register itself in the Zookeeper by creating a znode under a designated root path. The znode will be created with an ephemeral flag, so that if the storage server crashes, its associated znode will be automatically deleted and the Zookeeper watcher on the ECS will detect the deletion. The ECS will then remove all crashed nodes from the list of active nodes, and disconnect the associated socket connections. After that, the ECS will compute and distribute the new metadata to all active servers, and finally add new storage servers to replace the crashed ones.

Since the ZooKeeper watcher responsible for failure detection is in a separate thread, we guard against race condition by synchronizing all relevant ECS public methods (such as `addNode`) with a reentrant mutex; and when failure is detected in the watcher thread, we spawn a temporary thread that attempts to acquire the mutex and perform failure handling (e.g. updating statuses and adding replacement nodes). The reason to use a temporary thread is to not block the watcher thread from handling future events while the failure handler waits to acquire the lock.

1.2 Replication

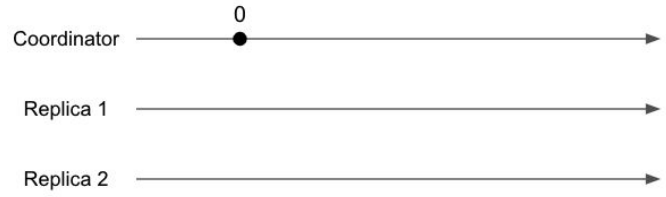
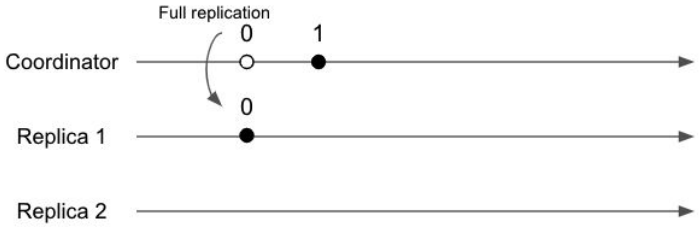
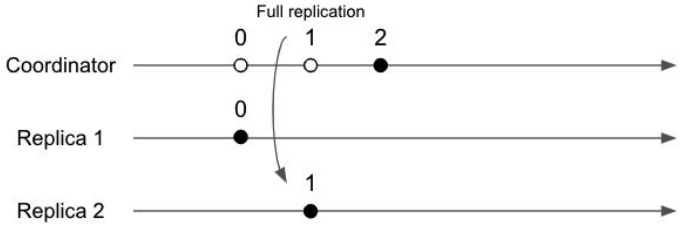
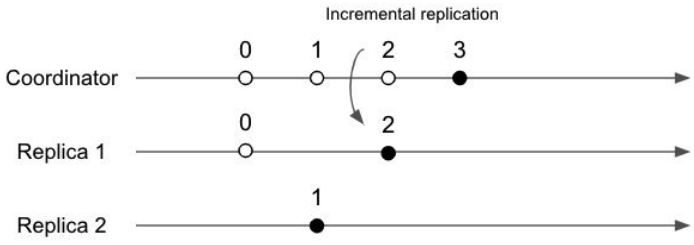
Our application features eventual consistency through automatic replication. The replication algorithm is decentralized, and requires minimal coordination effort from the ECS.

Each KVServer node acts as the **coordinator** for the hash range it is responsible for. Additionally, at most two closest successors of the coordinator on the hash ring (not including the coordinator itself) are designated as the **replicas** for the coordinator's hash range. To replicate data from the coordinator onto the replicas, we implemented the `Replicator` class, which is a companion thread in the KVServer that facilitates replication. Periodically (in our case, every second), the `Replicator` will check the current metadata (to identify if the replicas changed), then select a replica, and perform either a **full replication**, or an **incremental replication**, depending on the state of the replica. A full replication will first send a request to delete all data in the hash range on the target replica, then iterate over all local tuples in the hash range and use `SERVER_PUT` requests to send each tuple. All write requests from clients are blocked (with a `SERVER_WRITE_LOCK` response) during a full replication to trade availability

for consistency. An incremental replication will identify the relative changes in the key value tuples since the last replication on the target replica, and only send the changes instead of all tuples.

To facilitate incremental replication, we extended the server-side `KVStorage` to support delta recording: the ability to report a `Delta` object containing modifications in a given hash range since a certain point in time. The delta object is a dictionary mapping from keys to their latest value (can be null to indicate a delete), such that applying this dictionary mapping has the same effect as applying all write operations since the recording starts. At each put operation on the KVStorage, we also put the same tuple into the delta object (including when the value is null).

To better illustrate the replication algorithm, consider an example with a single coordinator and two replicas:

Assume the coordinator is at logical time = 0. The replicas have just been added to the system, and the coordinator has no local state information about them.	
The coordinator's replicator selects Replica 1; since there's no state information, we perform a full replication. Before replication, we start the delta recording at time 0. After replication, the coordinator then proceeds to time = 1.	
The coordinator's replicator selects Replica 2, and performs a full replication just like Replica 1. Before replication, we save the last delta (changes from time 0 to time 1) into a list, and start the next delta recording at time 1. After replication, the coordinator proceeds to time = 2.	
The coordinator's replicator selects Replica 1. Since we know the logical time of Replica 1's last sync (which was at time 0), we perform an incremental replication, sending the delta object from time 0 to 1, and the one from time 1 to 2. Then, notice that the delta object from time 0 to 1 is no longer needed by any replica, hence we remove it from the list.	
The process continues periodically and indefinitely, alternating between replicas.	

Our algorithm gracefully handles a wide range of topology changes in a robust manner. When a new node is added and becomes one of the two successors of a coordinator, the coordinator's replicator will

detect the new replica node, and will select it in subsequent replication steps. Since the new replica has no associated state (i.e. logical time of the last replication), the replicator will perform a full replication first. When a replicator node is removed or crashed, the replicator will also detect the removal (as well as the next successor that becomes the new replica) simply through the metadata changes, and react accordingly. If the coordinator's hash range changes (e.g. a predecessor is added or removed), the replicator will detect the change and invalidate all associated state, in order to force a full replication to each replica in the next update step. If the connections or replication requests to any of the replicas fail, the associated state of that replica will be invalidated, in order to fall-back to a full replication later if the connection comes back online.

Since our algorithm is decentralized, the ECS is only required to perform two actions: 1. Distribute the most recent metadata, such as when a node failure is detected, in order to inform replicators of the most recent state, and 2. Temporarily stopping and starting replication when transferring data. Recall that, when the ECS adds/removes a node to/from the system, it will coordinate a data transfer (see Milestone 2). During the transfer, to ensure consistency, we instruct the ECS to send a special `STOP_REPLICATION` request to both the nodes sending and receiving the data, and send a `START_REPLICATION` request after the transfer. The replicator implements cooperative cancellation: before each tuple is sent during a full/incremental replication, the replicator will check if a `STOP_REPLICATION` request has been received, and if so, interrupt the replication immediately.

The selection of the next replica to perform replication on is determined in a round-robin fashion, in order to avoid starvation. Additionally, we impose a limit on the total number of entries in all cached delta objects in order to avoid congestion under high load. When this limit is exceeded, the replicator will delete all delta objects, forcing full replications to be performed next. Since full replication will block write requests from clients, this will ensure that replica staleness will not grow unboundedly.

1.3 Client Changes

The main change at the client side is the failure handling method. KVStore uses cached metadata to find the corresponding server connection given a key. When the connection fails, instead of randomly selecting another existing server connection to send messages, we select the next successor. This has the effect of sending the message to the next replica in the event that the coordinator failed.

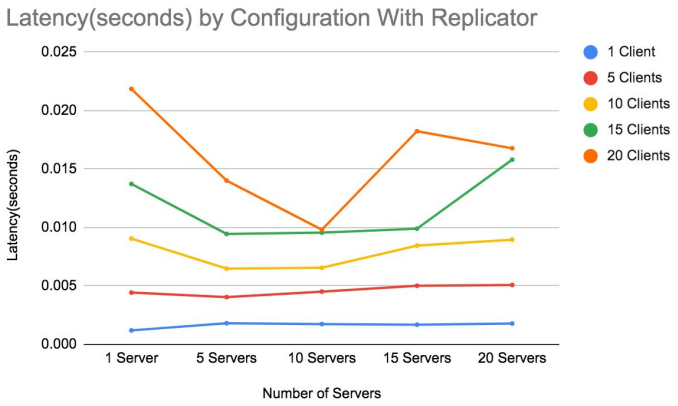
2 Performance Report

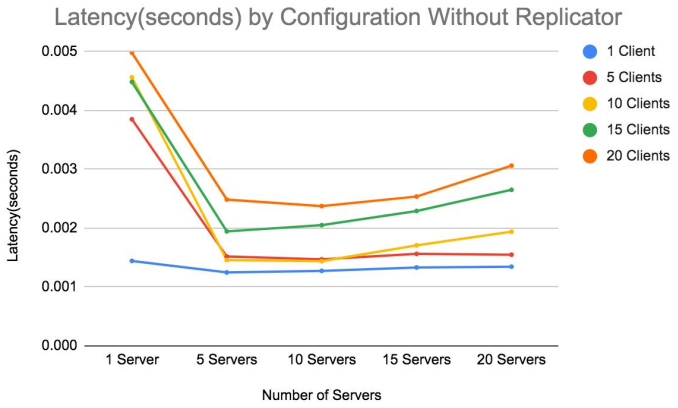
2.1 Method

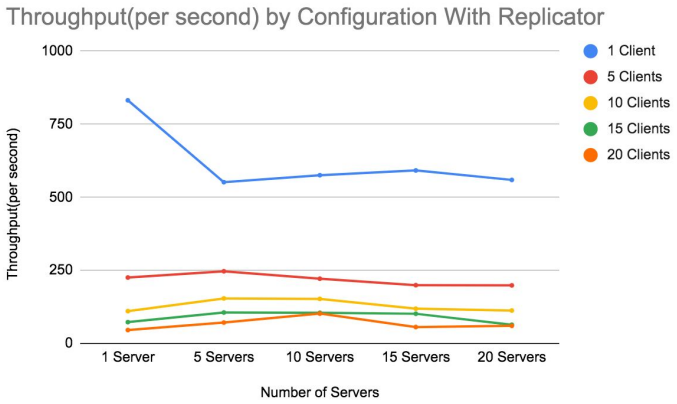
By reading 10,000 files from the Enroll Email dataset, we generated 10,000 key value pairs to populate our storage service. The average latency and throughput of our application is profiled for different numbers of servers and clients. We tested for latency and throughput using 1, 5, 10, 15 and 20 servers. These servers then served the requests from 1, 5, 10, 15 and 20 clients. Our Benchmarker script measures the latency and throughput for each of the 25 combinations, by randomly sending PUT or GET requests with equal probability. Each benchmark lasts for 10 seconds, with the results from the first 1 second ignored as warm-up.

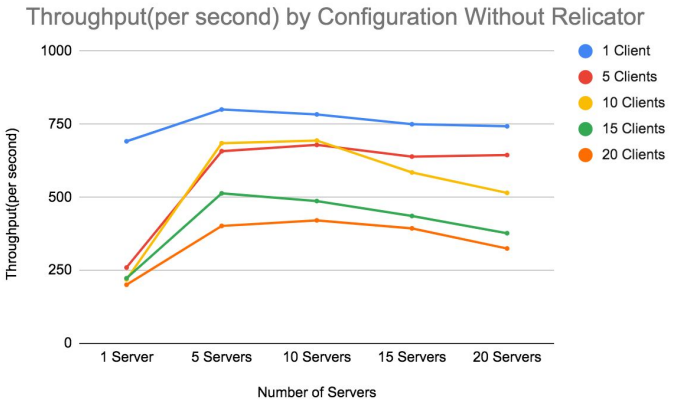
The tests are run on a mid-2015 MacBook Pro, with a 2.2 GHz Intel Core i7 processor and 16GB memory.

2.2 Results









The results of our experiments are shown in the above figures, with M3 results on the left and M2 results on the right. Compared with the results of experiments from M2, we can see the overall performance has degraded due to the replicator. Replication makes the system more reliable, but reduces the speed because of the extra space and procedures taken by replication. Aside from that, the overall trends of the two implementations are quite similar.

With more clients, the performance of the entire system decreases, since more clients implies more requests made per second. With more servers, the performance of the entire system increases at first then starts to decrease, since the system first utilizes more cores for processing client requests, but is then ultimately limited due to the execution being on a single computer.

We also measured the latency (in seconds) of adding and removing nodes, with different server count. The servers are also pre-populated with 10,000 key value pairs from the Enroll Email dataset.

Server count	1	5	10	15	20
Add node	13.851	8.383	2.838	1.798	1.345
Remove node	4.654	4.085	3.048	1.516	0.642

The cost of scaling the system decreases as the number of servers increases, which is expected since more servers in the system implies less data to be transferred when adding/removing a server.

Appendix A: Test Report

All Milestone 1 and Milestone 2 tests, including ConnectionTest, InteractionTest, ApplicationTest, and AdditionalTest (from M1 and M2), are modified to work with our Milestone 3 codebase. These act as regression tests to ensure that the functionalities in Milestone 1 and 2 are intact.

Tests added in AdditionalTest.java for Milestone 3:

Test Name	Test Summary	Test Details
testReplicatorFullReplication	Test if the Replicator class can successfully perform a full replication	Start two temporary KVServers and a Replicator, put tuples into one of the servers, use the replicator to perform a full replication to the other server, and check GET requests on the other server.
testReplicatorIncrementalReplication	Test if the Replicator class can successfully perform an incremental replication	Start a temporary KVServer and a Replicator, put tuples into a KVStorageDelta object, use the replicator to perform an incremental replication of this delta to the server, and check GET requests on the server.
testKVStorageDeltaCorrectness	Test if the KVStorageDelta class can correctly record write operations	Create a temporary KVStorageDelta object, put write operations into it, retrieve the entry set from the object, and check its correctness.
testKVStorageDeltaEntryCount	Test if the KVStorageDelta class can correctly provide the total recorded entry count	Create a temporary KVStorageDelta object, put write operations into it, retrieve the entry count from the object, and check its correctness.
testKVStorageDeltaDeletion	Test if the KVStorageDelta class can correctly record deletion operations	Create a temporary KVStorageDelta object, put delete operations into it, retrieve the entry set from the object, and check its correctness (which should contain the deletion entry with value = null).
testKVStorageDeltaPutHashRange	Test if the KVStorageDelta class only records write operations in the given hash range	Create a temporary KVStorageDelta object with a restricted hash range, put write operations into it, retrieve the entry set from the object, and check its correctness (should only contain operations within

		this range).
testKVStorageDeltaRecording	Test if the KVStorage class can correctly record and return KVStorageDelta objects	Create a temporary KVStorage object, instruct it to start recording delta objects, then perform a few put operations, and check if the correct delta objects can be obtained.
testKVServerSelfWriteLock	Test the correctness of the special write lock on KVServer, used during full replication	Create a temporary KVServer, call `lockSelfWrite()`, and check if subsequent client requests to the server result in `SERVER_WRITE_LOCK`.
testReplicaRead	Test if the client can read value from replica	Start two temporary KVServers with metadata and establish two connections with the servers. Send GET Request to the replicator server according to the key and check the return message status.
testReplicaWrite	Test if the client is unable to write value to replica	Start two temporary KVServers with metadata and establish two connections with the servers. Send PUT Request to the replicator server according to the key and check if the return message status is `NOT_RESPONSIBLE`.
testMetadataGetSuccessor	Test the correctness of successor query given a node	Test if the subroutine can correctly obtain the next node in the ring given an existing node. Also check the edge case when there's only one node in the system, which should return the node itself as its successor..