ECE419 Milestone 2 Design Document

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

(All members have equal contribution)

1 Overview

We implemented a scalable, high-performance, distributed key-value storage server application featuring dynamic scaling. Our implementation is based on our Milestone 1 codebase. The new features, improvements, and modifications added to our Milestone 1 implementation are documented in the sections below.

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

1.1 External Configuration Service (ECS)

To facilitate the distributed implementation of our storage application, an external configuration service (ECS) is used to activate, coordinate, and manage all instances of our KVServer node. The ECS depends on an active instance of Apache ZooKeeper in order or function, which must be started before launching the ECS. Once launched, the ECS provides a command-line user interface similar to that of the KVClient. The interface supports the following operations: adding/removing nodes to/from the system, starting/stopping the serving of active nodes, listing node statuses, and shutting down all nodes (before exiting the ECS itself).

At initialization time, the ECS will read and parse a configuration file for a list of nodes that can be added to the system. The ECS will then connect to the ZooKeeper server, and create a znode at `/kvECS`. To simplify the interfacing between our server/ECS and the ZooKeeper, we implemented a `ZooKeeperService` class that provides utility methods for performing common operations on ZooKeeper, such as creating/deleting znodes, obtaining childrens, and watching znodes for changes. For metadata and node management, the ECS keeps track of the statuses (represented by an enum) of all servers, whether they are active or not. Each server is initially in the `NOT_LAUNCHED` status. The metadata that will be sent to the nodes will be computed as a list of all servers that have the `ACTIVATED` status, along with their hash ring positions. The servers are sorted according to their hash ring positions, which enables the use of binary search to efficiently find the responsible node for a given key.

To add a node to the system, the ECS will randomly select a server from all servers with the `NOT_LAUNCHED` status, and use an SSH call to launch the server process, setting its status to `LAUNCHING`. The SSH call will find and execute the server jar file under the path specified by the environment variable `$ECE419_SERVER_PATH`, which must be configured on the target machine hosting the server. Additionally, the target machine must enable password-less SSH login, such as using an SSH key. After launching the process, the ECS will enter a spinning wait until a znode under `/kvECS` is created with the name of the server that is being launched. This is achieved by using a watcher on the `/kvECS` node that is notified when the children changes, and will set the node's status to `LAUNCHED`. Once the server process starts, it will connect to the same ZooKeeper service and create the corresponding znode using the ephemeral flag, which instructs the file to be deleted after the server terminates. When the ECS detects the znode, it will establish a socket connection with the server process for all future communications (see section 1.2 for more details). At this point, the status of the node is set to `CONNECTED`. Then, the new metadata will be computed, and the ECS will lock write operations on the successor, instruct the successor to transfer relevant data to the new node, send new

metadata to all nodes, unlock write operations, and delete the relevant data from the successor. At this point, the status of the node will be set to `ACTIVATED`.

Other operations, such as node removal and shutdown, are done similarly using socket connections as the means of communication, which is further described in section 1.2.

1.2 Server

The logic of our KVServer remains similar to the Milestone 1 implementation, with a few new functionalities added to support administrative commands from the ECS. Firstly, the server is modified to check if it has been started, if it has write lock, and/or if it is responsible for the given key, and return the appropriate error state to the client if needed. Additionally, we extended the existing message format to support commands from the ECS instead of the client, by adding new status types such as `ECS_COPY_DATA`, `ECS_UPDATE_METADATA` and so on. To support attaching metadata and other arguments along with these special commands, we added `getMetadata()` and `getECSCommandArg()` to the message interface. Since we utilize Java's serialization system, the new message format can easily be sent using the existing serialization protocol. The ECS then communicates with the server by opening a socket connection and using the exact same logic as KVClient to send messages.

When a server node is being asked to send its data to another server, it will scan the file system for all tuples that it needs to send, and then open a socket connection to the target server and send the special `SERVER_PUT` message for each tuple. Recall that our persistent storage will save tuples inside files where the file name is the first byte of the MD5 of all keys that it stores. Therefore, we only check files that can potentially contain tuples in the requested hash range. Furthermore, the special `SERVER_PUT` message is the same as the regular `PUT` message, except that the target server will ignore the write lock and force-write the data.

1.3 Client

The client user interface remains the same as Milestone 1 implementation while a few functionalities have been changed in the KVStore to support the distributed storage service.

We created a metadata class to store the mapping between servers and their properties by a list sorted by the hash ring position of each server. We also store a hashmap inside the KVStore, mapping from ring position to a connection object of the corresponding server. After the client starts, it must first connect to one of the KVServer using the user-entered ip address and port. Once connection is made, the server will send back its metadata. Whenever the client receives an updated version of the metadata (including at connection time), it will close all existing connections with any KVServer (since they will be opened later at request time), and cache this metadata locally. Any subsequent GET or PUT requests will then use the cached metadata to find the ring position of the corresponding ECSNode (using binary search), then try to find the corresponding server connection using the hashmap. If the server connection is found inside the hashmap, the client will use this connection to send messages (and will open the connection if it is currently closed). If not, the client will select a random server connection inside the hashmap to send messages for the purpose of getting updated metadata.

Once the connection is found, the client will send the command, key and value (for PUT request) through the server connection. After receiving the message back from the server, the client will check if the status of the response message is `NOT_RESPONSIBLE`. If the message contained other statuses, the response message is returned directly to KVClient; however, if the message contained `NOT_RESPONSIBLE`, it means that the metadata cached by the client now is stale, and the client will process the new metadata from the response message, closing existing connections and updating the cached metadata and hashmap. After that, the client will retry the request using the same procedure.

We also limited the maximum number of retry attempts. If the client tries to connect and send the requests too many times, we will return a failure message.

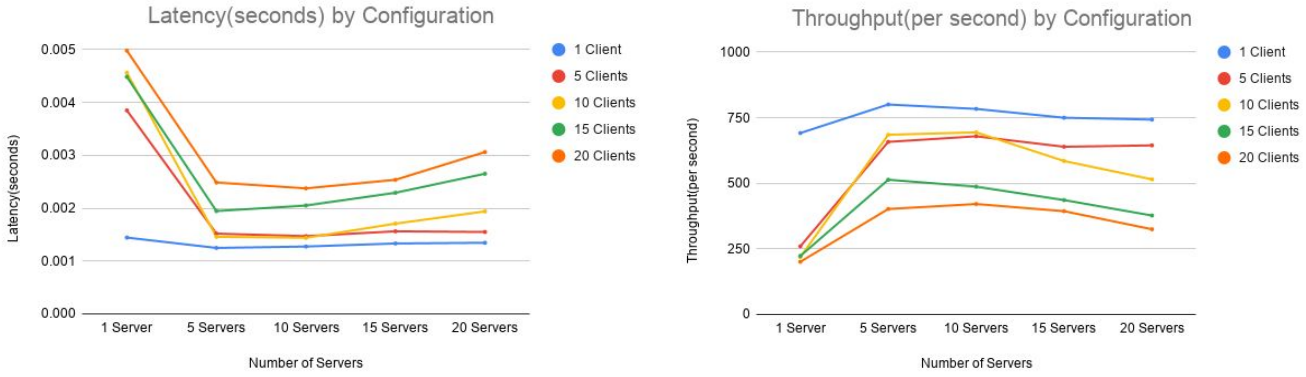
2 Performance Report

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. There are two trends immediately visible from the graphs: 1. As the number of clients increase, the performance of the entire system decreases; 2. As the number of servers increases, the performance of the entire system increases at first then starts to decrease. The first trend can be explained by the fact that more clients implies more requests made per second, which increases the number of requests processed per client. The second trend can be explained by the fact that the system is executed on a single computer, which has a limited number of independent processor cores. As the number of servers increases from 1 to 5, the performance is improved since the system now utilizes more cores for processing client requests. However, as the number of servers continues to increase, the computing power of the processors start to become depleted, and the overhead of coordination (as well as server upkeep costs) starts to take over, resulting in decreased performance.

Appendix A: Test Report

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

Additionally, the automatic performance benchmark code from Milestone 1 is also modified to work with our Milestone 2 codebase, for testing the performance of a single KVServer.

Tests added in AdditionalTest.java for Milestone 2:

Test Name	Test Summary	Test Details
testMetadataGetServer	Test the correctness of finding the responsible server given a ring position	Test the binary search algorithm, which finds the next server in the ring having ring position larger than or equal to the input ring position. Also ensure that the algorithm properly “wraps around” the hash ring.
testMetadataGetRingPosition	Test the correctness of the hashing logic	Test whether the hashing logic gives the correct MD5 hash of a given key
testMetadataGetPredecessor	Test the correctness of predecessor query given a node	Test if the subroutine can correctly obtain the previous 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 predecessor.
testECSReadConfig	Test the correctness of ECS config file parsing	Write a sample config file to a temporary directory, and verify if the parsing logic is correct on this temporary file.
testKVStorageGetAllKeys	Test the correctness of querying the keys that need to be sent to another server	Given a hash range and sample storage state, test if KVStorage can correctly find the keys within the range.
testKVFileStorageReadKeys	Test the correctness of querying the keys that need to be sent to another server on KVFileStorage	Given a hash range and sample storage state, test if KVFileStorage (which operates on a single file) can correctly find the keys within the range.
testKVServerShutdown	Test the correctness of server shutdown	Start a KVServer and ask it to shutdown immediately, check

		that there is no error thrown.
testKVServerNotResponsible	Test the correctness of responsibility check	Start a KVServer with sample metadata, start a client and query a key outside of the server's hash range. Check if the server returns `NOT_RESPONSIBLE`.
testKVServerWriteLock	Test the correctness of write lock check	Start a KVServer with sample metadata, ask it to lock write operations; start a client and query a key. Check if the server returns `SERVER_WRITE_LOCK`.
testKVServerStartStop	Test the correctness of serving status check	Start a KVServer without calling start, then start a client and query a key. Check if the server returns `SERVER_STOPPED`.