ECE419 Milestone 3

Wuqi Li 1000292033
Yuhui Pan 1001239395
Wuzhen Shan 1000411173

File System Design Document

Design Overview

The goal of this milestone is to extend External Configuration Service and KVServer on top of Milestone 2. The storage service will be a controllable, scalable and robust entity with the backup from replicas setup by ECS.

General Responsibility Of Each Component

KVServer has been taking more responsibilities since Milestone 2, including transferring data to replicas and serving as a replica that receives data from other servers. Adding and removing servers still functions the same way as previous milestone by sending and receiving data according to the new metadata change. All these communications are triggered by ECS but involving very few actions by ECS. ECS is still functioning as a central brain taking control of the entire storage service. And this time with the additional functionality that detect failure of servers as well as setting up the replicas. KVClient is serving as the interface the end-users, with crashed server detection that is hidden from the user. All these implementation details will be fully explained in the following section.

Information carried from last two milestones is as below (you can refer it by reading our previous report):

Communication Logic I & II (CommunicationModule)
Translation Layer I & II (KVStore & ClientConnection)
Cache system (KVCache)
File System (KVDB)
External Configuration Service (ECS)
New Data Message (MetaData)
Note: For all the non-trivial details that has been achieved by Milestone 1 and 2, we just assume them to be known and may briefly mention them as necessary.

External Configuration Service II (ECS)

As Milestone 2, ECS is basically made up of three parts: ECSClient, ECS, ECSWatcher.
We have completely refactor the code structure from Milestone 2 to make it more readable and reusable. Also we redesign the mechanisms that work between ECS and KVServers. Basically, ECSWatcher is storing the zookeeper instance to assist the communication between ECS and server, and the message from ECS has been assured to be safe for the entire system to function properly, and the guarantee is

achieved by confirming acknowledgements from all involved servers per operation (add nodes, remove nodes, etc.).

MetaData II

Treeset<IECNode> is what we used to store different kinds of server repository as Milestone 2. Metadata is where we store latest running servers that is deployed by ECS. In addition to that, we add a lot of functionalities to this class in order to wrap it as a utility class, achieving better management of latest server state, and it is also available to be passed between KVServers and ECS by Gson serialization we used through the entire Milestone.

Communication Logic III (Zookeeper between ECS and KVServer)

We have changed many parts of this module in order to fit the specific needs of this milestone as well as improving the efficiency and speed of Distributed Storage Service. The details will be covered as we go through how to we implement the following three sections:

1.Replication Mechanism Implementation

Since we load the ecs.config file entirely at the starting point of ECSClient, we are able to gain a complete hash graph of all servers before starting any server and it will remain relatively stable over the entire life cycle. We basically distribute the replica as instructed by the handout, the two servers right after. We use ECSWatcher to notify all servers of the incoming metadata along with the operation type all these servers has to comply to. Once the server gets the latest metadata, the server will immediately transfer the data to its replicas. The detail is that the server will call the utility functions inside MetaData to check its replicas and signal the target node of transferring action by creating new zookeeper node under that target node. All data are transferred by this zookeeper node and operations are guaranteed to be safe by acknowledging each successful transmission by target the node.

2. Failure Detection and Recovery

We have specifically design one additional class named ECSDetector to detect the failure. It is maintained as a Hashmap of ECSDetectors under ECS. The ECSDetector is basically simulating the behavior of an actual ECSClient by connecting to the KVServer. We have exactly the same number of ECSDetectors as that of running KVServer instances. We connect each ECSDetector to each running KVServer, and once a server has crashed, it will signal the ECSDetector since there will be an exception thrown for a currently connected client if it gets disconnected by the other

side, which in our case, is the ECSDetector that we are monitoring on. We have also registered ECS and the monitored node under every single ECSDetector such that when a failure is detected, ECSDetector is alway able to inform ECS of the event and passes back to crashed node. All the information can be retrieved from the node and ECS will remove it from the list and set up new node into the storage service from the available servers pool in order to maintain the server number. All the necessary data transmission will also be carried out as detection of this event and follows the same workflow of removenodes. But this time, this crashed server will not be put back into the reusable pool and ECS will never use it in the future.

3. Mechanism for clients to gracefully handle the failure of the storage server

Just like a when a client encounter a metadata change, once a client detect the connection session has failed, it will pull the first available server from the metadata and connect to it to see any change. And this is step may succeed or fail depends on the current storage service setup. If it fails it will get the latest metadata and through there it will find the right server to connect. If it succeed, it will remain as it is until the next time when it request something is not in the range of that server. Finally, it will get updated and sync to the latest metadata. There is only scenario that our client would fail, which is all the available servers in the metadata has crashed, and we are unable to handle this kind of failure.

Testing

The test structure is as follows:

ConnectionTest.class
InteractionTest.class
ECSClientTest.class
ECSClientInteractionTest.class
KVCacheTest.class
KVServerTest.class
MetaDataTest.class
ECSNodeTest.class

The screenshot of all unit tests is in the appendix.

Since the very specific task of our milestone, the test strategy we use is by setting up a ECSClient for most TestCase such that we are able to encapsulate the whole

environment into one single test and make it more modular. We have pass all the test cases. And the following is the full details of what each test case does. The result is attached as appendix at the end of the report.

ConnectionTest.class and InteractionTest.class
These two are what is provided by Milestone 1 to test the basic functionality of the storage service.
ECSClientTest.class
Test all most testable interface provided by IECSClient.
ECSClientInteractionTest.class
1. testLoad(). A very comprehensive amount of put and get request in order to test the stability of the storage service maintained by ECS
2. testDataTransfer(). Test data transmission between servers. ECS has three servers at the starup, and then add another three servers when the storage service is up. Remove three of them and three left. When doing any further operation from the client side, all should succeed since the data has been transferred from the removed server to the still-up servers.
3. testKVServerCrashed(). This test if the storage service is robust when a crash happens. We remotely kill a one of the server to see if ECS has spawn a new server.
KVCacheTest.class and KVServerTest.class
These are test cases to test the Cache, File System and KVServer interface.
MetaDataTest.class
Metadata Class serves as a repository to manage the metadata. It has a lot of utility function that we set up in order to assist metadata operation. Such as removing remove servers from the metadata, getting all the servers between two servers, calculating hash range, getting the next/previous server, etc. You can refer to IMetaData to see a full list of implemented utility functions.
ECSNodeTest
Test all interfaces provided by IECSNode.

Performance Evaluation

The performance test is embedded into the test cases, and the structure is as follow:

PerformanceTest.class

We use Enron Email Dataset to do the performance evaluation. We simply set the file path to be the key and content of the file to be the value. All performance test follow the same test pattern, each client is responsible for all files under one single folder in maildir, and there are 150 folders in maildir, we can reach a maximum of 150 clients.

Our performance test only uses up to 100 folder of them, i.e. 100 clients. All performance test has the same setting of:

KVServers ranging from: 5 - 100, step size 5
CacheSize ranging from 50 - 1000, step size 50
KVClients ranging from 5 - 50, step size 5
Also, we run test on different cachestrategy.

From the result, we can see the testing time decrease huge with increasing server numbers. Also, having cache will speed up the test. LFU having the best speed in our testing.
Besides, increasing the Cache size will also increase the testing time. Increasing clients number will speed up the testing in some cases.

The result is as follows:

Servers 5 Strategy: None Size 50

Server Number: 5 | Client Number: 5
Processing time: 672ms

Server Number: 5 | Client Number: 10
Processing time: 1386ms

Server Number: 5 | Client Number: 15
Processing time: 407ms

Server Number: 5 | Client Number: 20
Processing time: 1415ms

Server Number: 5 | Client Number: 25
Processing time: 2428ms

Server Number: 5 | Client Number: 30
Processing time: 1440ms

Server Number: 5 | Client Number: 35
Processing time: 1459ms

Server Number: 5 | Client Number: 40
Processing time: 1485ms

Server Number: 5 | Client Number: 45
Processing time: 2504ms

Server Number: 5 | Client Number: 50
Processing time: 2519ms

Servers 5 Strategy: FIFO Size 50

Server Number: 5 | Client Number: 5
Processing time: 678ms

Server Number: 5 | Client Number: 10
Processing time: 1371ms

Server Number: 5 | Client Number: 15
Processing time: 404ms

Server Number: 5 | Client Number: 20
Processing time: 1414ms

Server Number: 5 | Client Number: 25
Processing time: 1461ms

Server Number: 5 | Client Number: 30
Processing time: 1408ms

Server Number: 5 | Client Number: 35
Processing time: 2463ms

Server Number: 5 | Client Number: 40
Processing time: 1465ms

Server Number: 5 | Client Number: 45
Processing time: 1436ms

Server Number: 5 | Client Number: 50
Processing time: 2537ms

Servers 5 Strategy: LRU Size 50

Server Number: 5 | Client Number: 5
Processing time: 668ms

Server Number: 5 | Client Number: 10
Processing time: 1361ms

Server Number: 5 | Client Number: 15
Processing time: 408ms

Server Number: 5 | Client Number: 20
Processing time: 1401ms

Server Number: 5 | Client Number: 25
Processing time: 1461ms

Server Number: 5 | Client Number: 30
Processing time: 2442ms

Server Number: 5 | Client Number: 35
Processing time: 1464ms

Server Number: 5 | Client Number: 40
Processing time: 1497ms

Server Number: 5 | Client Number: 45
Processing time: 2496ms

Server Number: 5 | Client Number: 50
Processing time: 1486ms

Servers 5 Strategy: LFU Size 50

Server Number: 5 | Client Number: 5
Processing time: 668ms

Server Number: 5 | Client Number: 10
Processing time: 1380ms

Server Number: 5 | Client Number: 15
Processing time: 1390ms

Server Number: 5 | Client Number: 20
Processing time: 1414ms

Server Number: 5 | Client Number: 25
Processing time: 94ms

Server Number: 5 | Client Number: 30
Processing time: 104ms

Server Number: 5 | Client Number: 35
Processing time: 141ms

Server Number: 5 | Client Number: 40
Processing time: 132ms

Server Number: 5 | Client Number: 45
Processing time: 144ms

Server Number: 5 | Client Number: 50
Processing time: 188ms

Servers 8 Strategy: LFU Size 50

erver Number: 8 | Client Number: 5
Processing time: 685ms

Server Number: 8 | Client Number: 10
Processing time: 63ms

Server Number: 8 | Client Number: 15
Processing time: 96ms

Server Number: 8 | Client Number: 20
Processing time: 100ms

Server Number: 8 | Client Number: 25
Processing time: 118ms

Server Number: 8 | Client Number: 30
Processing time: 145ms

Server Number: 8 | Client Number: 35

Processing time: 191ms

Server Number: 8 | Client Number: 40
Processing time: 164ms

Server Number: 8 | Client Number: 45
Processing time: 196ms

Server Number: 8 | Client Number: 50
Processing time: 227ms

Servers 5 Strategy: FIFO Size 100

Server Number: 5 | Client Number: 5
Processing time: 1561ms

Server Number: 5 | Client Number: 10
Processing time: 398ms

Server Number: 5 | Client Number: 15
Processing time: 1384ms

Server Number: 5 | Client Number: 20
Processing time: 2420ms

Server Number: 5 | Client Number: 25
Processing time: 1405ms

Server Number: 5 | Client Number: 30
Processing time: 1458ms

Server Number: 5 | Client Number: 35
Processing time: 2444ms

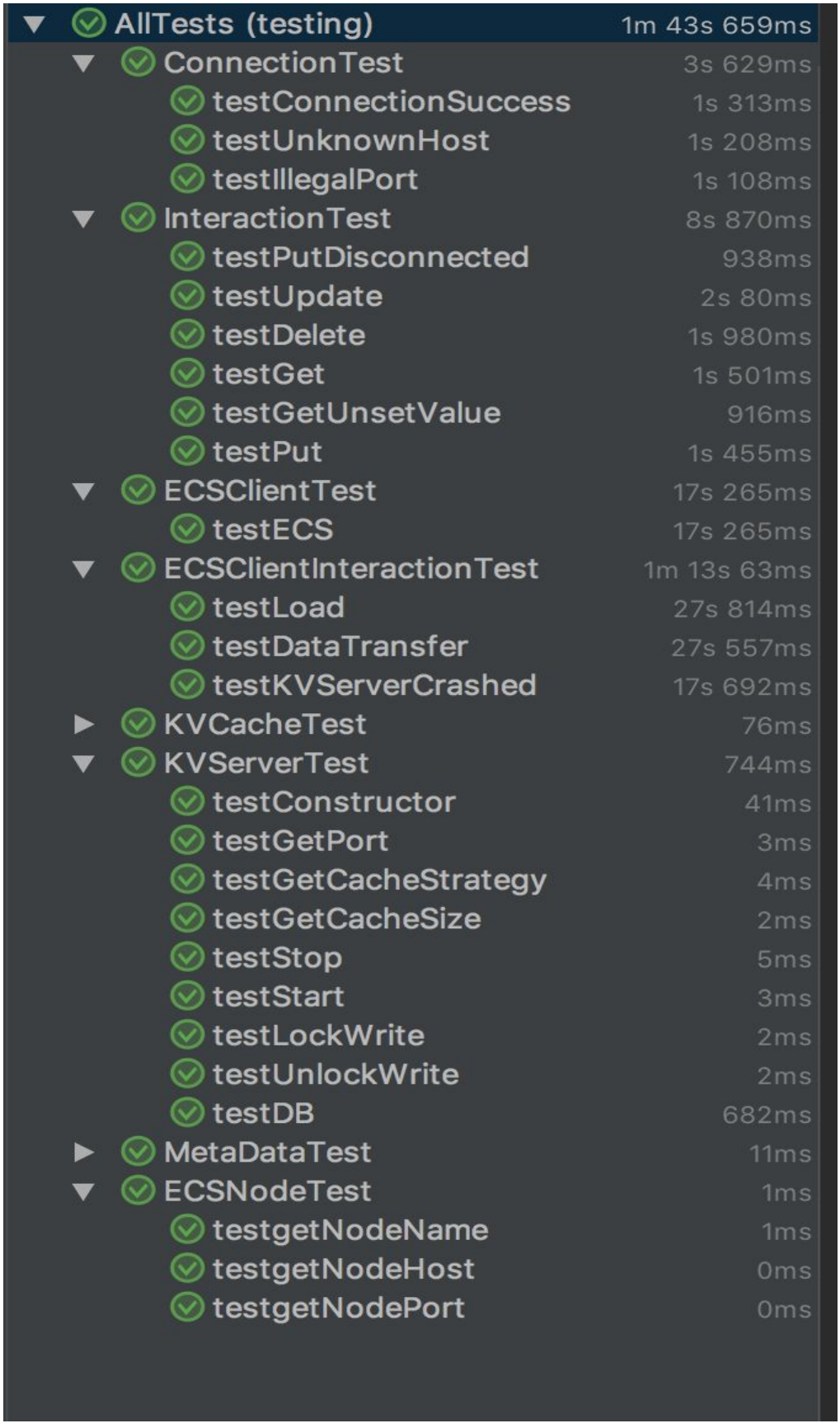
Server Number: 5 | Client Number: 40

Processing time: 1489ms

Server Number: 5 | Client Number: 45
Processing time: 1476ms

Server Number: 5 | Client Number: 50
Processing time: 2454ms

APPENDIX A


AllTests (testing) 1m 43s 659ms
ConnectionTest 3s 629ms
testConnectionSuccess 1s 313ms
testUnknownHost 1s 208ms
testIllegalPort 1s 108ms
InteractionTest 8s 870ms
testPutDisconnected 938ms
testUpdate 2s 80ms
testDelete 1s 980ms
testGet 1s 501ms
testGetUnsetValue 916ms
testPut 1s 455ms
ECSClientTest 17s 265ms
testECS 17s 265ms
ECSClientInteractionTest 1m 13s 63ms
testLoad 27s 814ms
testDataTransfer 27s 557ms
testKVServerCrashed 17s 692ms
KVCacheTest 76ms
KVServerTest 744ms
testConstructor 41ms
testGetPort 3ms
testGetCacheStrategy 4ms
testGetCacheSize 2ms
testStop 5ms
testStart 3ms
testLockWrite 2ms
testUnlockWrite 2ms
testDB 682ms
MetaDataTest 11ms
ECSNodeTest 1ms
testgetNodeName 1ms
testgetNodeHost 0ms
testgetNodePort 0ms