ECE419 Design Document M2

Bohao Wang 1002993442
Jingwei Zhang 1002896690
Yanyi Zhang 1003327517

Hash Ring

The Hash ring class is implemented in our HashRing.java class leveraging the ECSNode class. With the designed notable data structures, given a server name, its previous server can be easily identified to suffice the consistent hashing functionalities (ex, add a server, remove a server, put data, get data, etc). This JSON-serializable HashRing structure is designed to be the metadata we use to communicate among servers, clients, and the ECS.

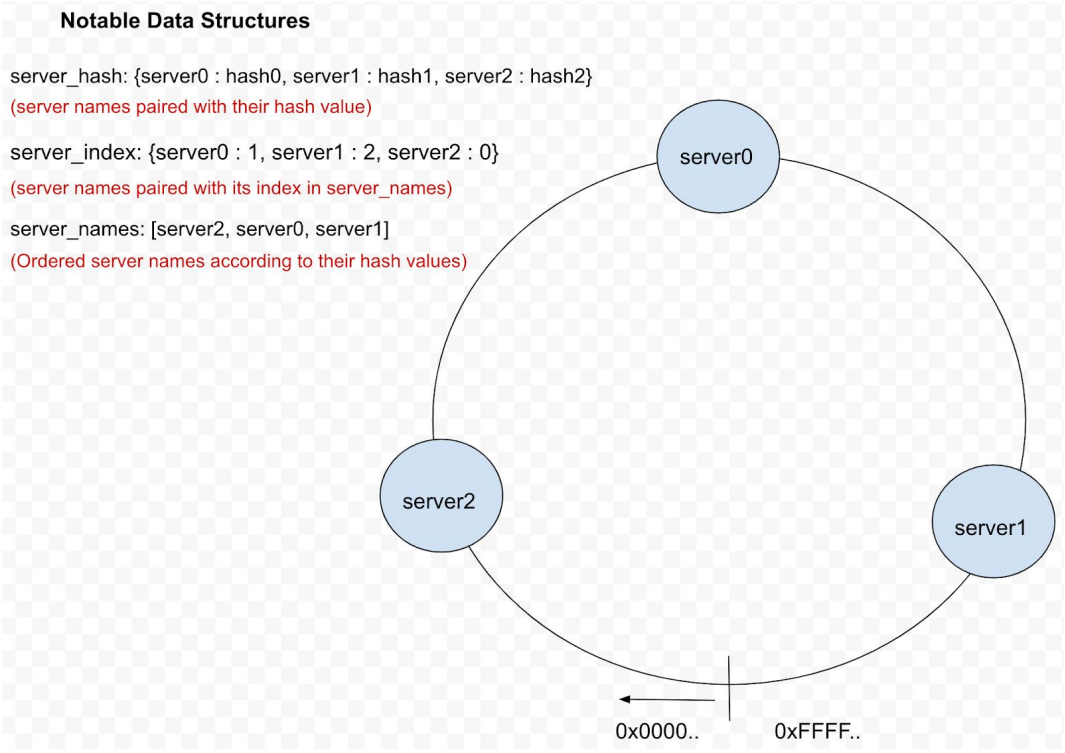

Figure1. The HashRing structure

ZooKeeper

Node Name	Stored Value
ActiveServers	The node serves as a directory, with all its children being the currently active servers.
WaitingServers	The node serves as a directory, with all its children being the currently waiting servers. (In the stage of initialization)

{ServerName}	This is the child node of either ActiveServers or WaitingServers representing the KVServer node. The node will be created/deleted with the ECS command including addNode, removeNode, etc. The stored data in this node is the message that ECS wants to pass to servers, including start(), stop() moveData(), etc.
Metadata	The node stores the HashRing information. The ECS will continuously update the value stored here, while each server will watch for the update on this node if the stored value has been changed.

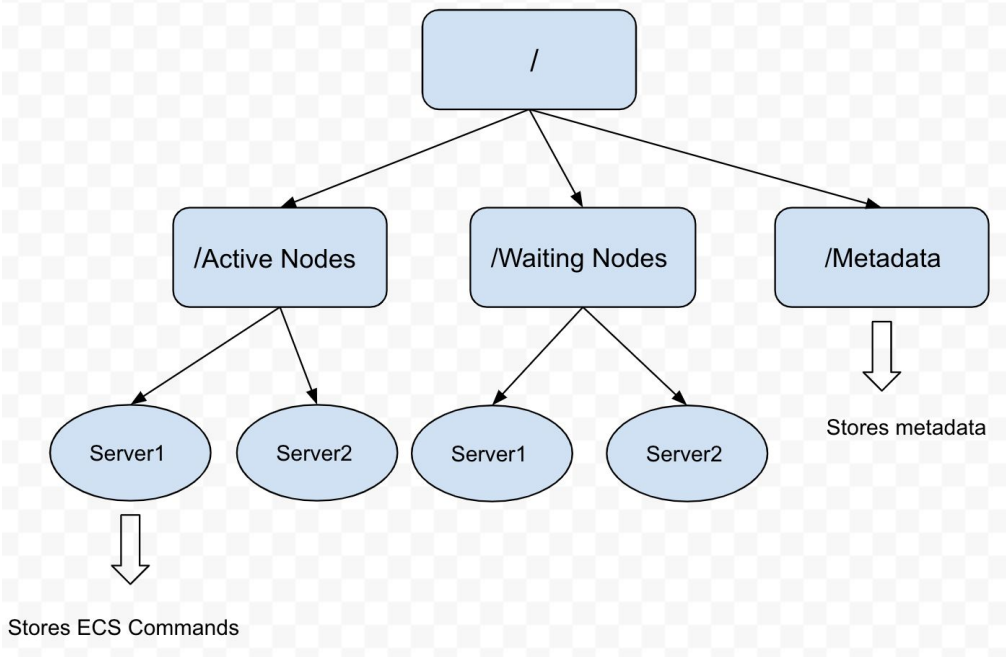

Figure 2. The Zookeeper structure

Server

Upon initialization, the server creates watchers on its related Znode in zookeeper and the Metadata node. When a new message has been stored in its related Znode, the actual server will perform the corresponding operation. After operation, the server will change the data within the Znode to be either 'Done' or 'Failed' to notify the ECS about the operation result. When the data within Metadata has been changed, the watcher set by the server will also notify the server to fetch the updated metadata to its local copy.

Core function: move_data(String range, String server_ip_port)

In the setup, data will be moved from the sender server to the receiver server.

Once the sender server receives the message of 'move_data' action, the sender server will first separate the data to be retained in this server and the data to be sent according to the given range. The retained data will be stored as remain_storage_file, and the data to be sent will be serialized into KVMessage {key: null, value: serialized data, status: MERGE}and transmitted to the receiver server through the receiver's alive socket given the server_ip_port.

On the received side, upon parsing the KVMessage, it will merge the newly received data with the local storage file, and if successful, send a "MERGE_SUCCESS" acknowledge back to the sender receiver.
Then, the sender server will finally delete the original_storage file, and renamed the remain_storage file to the name of the original_storage file as the new replacement. Then, the whole data movement succeeds, and the server will leave a 'Done' message to its related Znode to notify the ECS.

ECS

The ECS stores the information of all the servers in the ECSNode class. The ECSClient is the terminal interface to perform server-configuration commands, including addNode(), removeNode(), etc. Except for addNodes(), which initiates the Znode structure of '/Metadata', '/ActiveNodes', '/WaitingNodes', all other commands will be performed by
ECS through leaving messages to the zookeeper (changing the data to the children Znodes of 'Metadata', '/ActiveNodes' or '/WaitingNodes') to notify the watchers on these Znodes, which are set by the actual servers.

Performance Test

To evaluate our performance, we initiated 1612 requests consisting of 50% get and 50% put requests using the data from Enron Email dataset. We managed to run these requests with different configurations of client numbers, servers and caches. The latencies and throughputs are recorded in the following table

One pattern can be identified is that with more servers being activated, the throughput can be improved significantly, from 4 requests / sec to 155 requests / sec. The latency for the average put and get requests will also decrease with more servers. One possible reason is that with more servers, the system is able to handle more requests at the same time. Another possible reason is that with the usage of consistent hashing, the searching procedures are more efficient as data are "categorized" according to hashing value. Besides, it seems that the increased number of client connections can test the potential of the system, while not affecting the system overall performance.
Another pattern is that cache size seems to play an important role in the system performance. With different cache sizes from 100 to 500, the performance has been greatly improved. However, different cache strategies do not impact the performance explicitly. In comparison, LFU performs the best in terms both throughput and average request latency.

The last pattern is that get request is a lot faster than put request in terms of the average latency, as put request always requires extra time to access disk storage.

Appendix

FIFO Size 100	5 Client/5 Server	5 Client 20 Server	5 Client 50 Server	20 Client/ 5 Server	20 Client/20 Server	20 Client/50 Serve
Request/sec	2.8	49	123	2.7	112	153
Avg Latency (ms/request)	335.2	20.2	7.1	364.2	9.2	8.4
Avg Get Latency (ms/request)	301.3	14.6	5	372.5	10	9.2
Avg Put Latency (ms/request)	369.6	26.6	9.2	381	11.8	10.1

FIFO Size 500	5 Client/5 Server	5 Client 20 Server	5 Client 50 Server	20 Client/ 5 Server	20 Client/20 Server	20 Client/50 Serve5
Request/sec	3.6	52.2	142	3.9	116	156
Avg Latency (ms/request)	253.3	18.4	7.7	280.2	8.8	8.1
Avg Get Latency (ms/request)	220.2	16.4	5.2	190	7.5	6
Avg Put Latency (ms/request)	286.4	24.7	9.2	270.4	10	10.2

LRU Size 100	5 Client/5 Server	5 Client 20 Server	5 Client 50 Server	20 Client/ 5 Server	20 Client/20 Server	20 Client/50 Serve
Request/sec	3.17	48	118	3.2	111	149
Avg Latency (ms/request)	315	23	8	325.2	9	7.5
Avg Get Latency (ms/request)	290.5	17	5.5	312.6	7.5	6
Avg Put Latency (ms/request)	338.5	29	10.5	337.2	10.5	9

LRU Size 500	5 Client/5 Server	5 Client 20 Server	5 Client 50 Server	20 Client/ 5 Server	20 Client/20 Server	20 Client/50 Serve
Request/sec	4.06	51.3	141	4.2	120	159
Avg Latency (ms/request)	252.2	19.8	6.5	252.8	8.2	7.9
Avg Get Latency (ms/request)	218.3	13.5	5	265.5	6.2	6.4
Avg Put Latency (ms/request)	284.4	26	8	278	10.2	9.4

LFU Size 100	5 Client/5 Server	5 Client 20 Server	5 Client 50 Server	20 Client/ 5 Server	20 Client/20 Server	20 Client/50 Serve
Request/sec	3.18	49	119	2.9	114	150
Avg Latency (ms/request)	320	23	6.8	312.8	10	7.8
Avg Get Latency (ms/request)	292.4	16	5	300.4	9	6

Avg Put Latency (ms/request)	347.6	30	8.6	325.6	11	9.6

LFU Size 500	**5 Client/5 Server**	**5 Client 20 Server**	**5 Client 50 Server**	**20 Client/ 5 Server**	**20 Client/20 Server**	**20 Client/50 Serve**
Request/sec	4.28	51	146	4.6	117	156
Avg Latency (ms/request)	254.1	16	6.75	256.4	8	6.2
Avg Get Latency (ms/request)	220	12	5.2	240.2	7.5	5.6
Avg Put Latency (ms/request)	288.2	20	8.3	272.6	9.5	7.2

Table 1. Performance comparison with different cache sizes, strategies, number of servers and clients

Test Cases:

Utility Functions:

Name	**Description**
testHashRing	Test whether servers and data can be added to correct positions in the hashring.

Name	**Description**
testHashRingRange	Test hash ring “inRange” query: whether a data is in the range of a specific server’s responsibility.

Name	**Description**
testSerialization	Test serialization and deserialization functionalities

Name	**Description**
testZookeeperConnection	Test whether the zookeeper server can be connected correctly.

KV Service:

Name	**Description**

testServerStopped	Test whether a stopped server will block all incoming client requests and inform the client that the server is stopped.

Name	**Description**
testMultiClientMultiServerPutGet	Test put and get functionalities of the KV store service with multiple clients to multiple servers connections.

Name	**Description**
testMultiClientMultiServerUpdateDelete	Test update and delete functionalities of the KV store service with multiple clients to multiple servers connections.

Name	**Description**
testClientDynamicServer	Test put and get functionalities of the KV store service with dynamic multiple servers connections where the number of servers keep changing.

ECS Control:

Name	**Description**
testECSShutdown	Test whether ECS is able to shut down all the running servers correctly.

Name	**Description**
testECSStartStop	Test whether ECS is able to start and stop all the running servers correctly.

Name	**Description**
testECSAddNode	Test whether ECS can add nodes correctly

Name	**Description**
testECSRemoveNode	Test whether ECS can remove nodes correctly