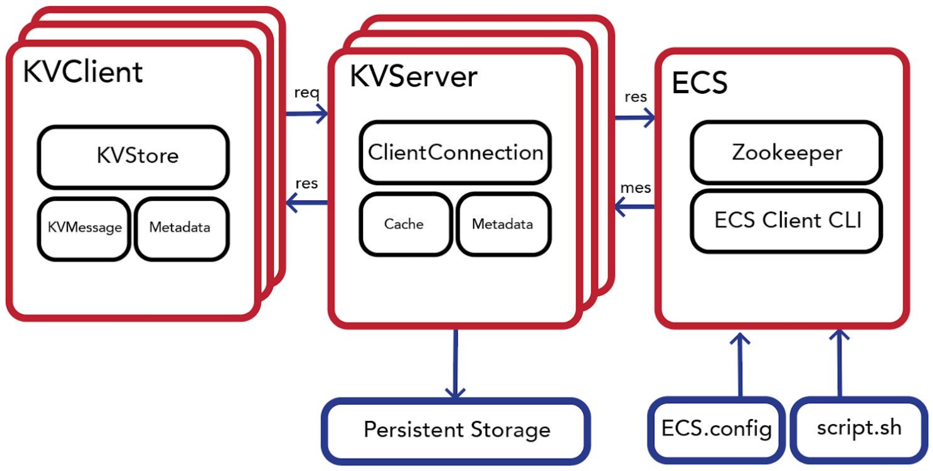
ECE419 Design Document Milestone 2

Sidra Syeda（1002318226）
Ankita Singal (1002478838)
Ahmed Abdulkadir (1002108587)

Design Overview



ECS

ECS is the system that allows servers to be maintained. ECS supports adding, removing, starting, stopping and shutting down of servers. ECS uses zookeeper to keep track of the servers, and communicate with the servers. ECS maintains a consistent hashing hashring which is created using MD5. ECS uses a sorted hash map to store the hash ring. The range of each server is computed using the hash of the current server and previous server. The ranges are computed and stored in the data value of the znode in the form of: "server_ip:server_port previous_server_hash-server_hash\n".

Zookeeper

A Znode is created by ECS in Zookeeper that holds metadata, which stores the hashring configuration. The storage servers read contents from the znode to get the metadata of the servers. The contents of the znode are updated as servers are added or removed and the servers are informed that they require updates.

KVServer

Each instance of KVServer is initiated through the ECS Client. When a server is instantiated, each server in the database will have its metadata updated with the most up to date version of the hash ring, and the servers are put in the stop mode. If there is a client that tries to run a get or a put request then the server replies with the SERVER_STOPPED message. All servers are in stop mode until the ECS Client sends a start message. The ECS Client uses the server metadata to check which server is responsible for a given client request. If a server is not responsible for a request, then the ECS sends an updated metadata to the client, and a SERVER_NOT_RESPONSIBLE message as well.

Any time a node (i.e. a server) is added or removed, then ECS will update the metadata of all the servers. In the case of an added node, the successor node transfers its data to the new server. The following node establishes a connection to the newer node, and moves its cache entries to the new node and clears its own cache.

When nodes are deleted, the data is first transferred to the next node in the ring. The server receiving the new data is put in a lock_write stage which means a client is unable to do a put request on that server until data transfer is complete. In this case the server replies to the client with a SERVER_LOCK_WRITE message.

KVClient

Every time a client is instantiated, the metadata is used to check which server we need to contact for requests. Initially, a client's metadata will only contain the server it is connected to, so when the first request is made, it will always contact that server. If a server is responsible for the request then it responds, but otherwise, the client's metadata is updated by the server that receives the request. The client then reconnects to the correct server using its new metadata and completes the request.

Responsibilities of ECS

Command	**Algorithm**
init <numberOfServers> <cacheSize> <cacheStrategy>	Read ecs.config, create ECSNode objects, create znode, call script to ssh into servers.
start	Send a message to all active servers to start accepting requests from clients.
stop	Send a message to all active servers to stop accepting requests from clients.
addNode <cacheSize> <cacheStrategy>	Add a node from idle servers, update metadata of node and successor node, lockwrite successor node as it updates metadata, send update message to all servers to update their metadata.
removeNode <index1> <index2>	Remove node based on the order it was added in. Remove node, update metadata of successor, transfer data to successor, lockwrite successor as it updates, send update message to all servers to update metadata.
shutdown	Stops all servers.

Performance Evaluation

Each test has 1000 total requests, with put/get load ratios of 50%/50% respectively. The test is performed with cache sizes 50 and 400 each with varying cache strategies (FIFO, LRU, LFU) and varying amounts of clients and servers in the storage service.

Performance Table: 500 possible key, value pairs

Cache Size: 50

Total Number of Requests: 1000	**LRU**					
# of clients (c) / # of servers(s)	5c/1s	5c/15s	5c/50s	25c/1s	25c/25s	25c/50s
Total Latency(sec)	1.72	0.098	0.052	1.47	0.143	0.129
put throughput (request/sec)	647.31	5122.3 6	19131.15	686.53	3708.23	4516.96
get throughput (request/sec)	527.29	28.72M	29.02M	678.34	61050.76	26787.95

Total Number of Requests: 1000	**LFU**					
# of clients (c) / # of servers(s)	5c/1s	5c/15s	5c/50s	25c/1s	25c/25s	25c/50s
Total Latency(sec)	1.43	0.0586	0.0559	1.47	0.09244	0.118
put throughput (request/sec)	707.18	8535.28	8945.94	695.17	6349.91	4977.097
get throughput (request/sec)	687.28	25.46M	30.37M	670.32	36491.43	28290.84

Total Number of Requests: 1000	**FIFO**					
# of clients (c) / # of servers(s)	5c/1s	5c/15s	5c/50s	25c/1s	25c/25s	25c/50s
Total Latency(sec)	1.778	0.0606	0.0545	1.491	0.1369	0.144
put throughput (request/sec)	699.99	8256.42	9179.95	691.92	4091.24	3920.02
get throughput (request/sec)	469.87	24.93M	25.84M	650.67	33965.49	30068.07

Cache Size: 400

Total Number of Requests: 1000	**LRU**					
# of clients (c) / # of servers(s)	5c/1s	5c/15s	5c/50s	25c/1s	25c/25s	25c/50s
Total Latency(sec)	1.41	0.0768	0.0551	1.457	0.119	0.140
put throughput (request/sec)	717.29	12429.31	9085.17	700.73	4486.82	4498.37
get throughput (request/sec)	696.97	13662.09	14.14M	672.43	67571.96	17320.37

Total Number of Requests: 1000	**LFU**

# of clients (c) / # of servers(s)	5c/1s	5c/15s	5c/50s	25c/1s	25c/25s	25c/50s
Total Latency(sec)	1.736	0.044	0.0495	1.494	0.170	0.123
put throughput (request/sec)	546.32	11346.22	10102.19	681.08	3607.37	4677.10
get throughput (request/sec)	609.53	17.27M	18.52M	657.94	15840.28	29330.46

Total Number of Requests: 1000	**FIFO**					
# of clients (c) / # of servers(s)	5c/1s	5c/15s	5c/50s	25c/1s	25c/25s	25c/50s
Total Latency(sec)	1.890	0.7366	0.0565	1.485	0.113	0.129
put throughput (request/sec)	545.446	6789.64	8851.36	694.34	5760.39	4173.25
get throughput (request/sec)	513.544	22.80M	24.89M	653.48	19141.55	55447.67

Scaling Up Performance

	Add latency (s)	**Remove Latency (s)**
5 nodes	5.03	2.52
10 nodes	10.07	5.05
15 nodes	15.12	7.58
20 nodes	20.18	10.13

Performance Conclusions

From milestone 2's performance test, one noticeable pattern that can be seen, is that the throughput of the storage service tends to increase as the number of storage servers increase. Specifically, if the ratio between the number of servers to clients increase, the service tends to perform better as there is less load on each server in the service, as client requests are more evenly distributed among each of the servers. In addition, the average latency of a request seems to decrease as the number of servers increase, this is may be due to the fact that the more servers available, the more client requests can be handled in parallel. As seen from milestone 1 tests, cache size and strategy contributes greatly to performance. The larger the cache size of a server, the better the lower the latency, especially for get requests, as fetching from entries from an in memory cache is much faster than retrieving the entry from disk. Cache strategy still has an impact on the service performance, but it has become much less in milestone 2, LFU still being the best performing strategy. Finally, the get performance seems to be much higher than put performance, as puts must write to disk, while get requests can simply retrieve the entry from the cache if it's there.

Appendix - Unit Tests

The unit tests from milestone 1 have been updated to work with the ecs interface.
Result: 100% pass rate.

Additional Unit tests for M2:

Test Name	**test_ecs_init()**
Description	Initialize 1 server with ecs and connect a client to it
Status	Pass

Test Name	**test_addNodes()**
Description	Initialize 1 server with ecs, then add 2 nodes, and connect clients to each
Status	Pass

Test Name	**test_start()**
Description	Initialize and start 1 server, then connect client and do a put
Status	Pass

Test Name	**test_remove()**
Description	Initialize 1 server then remove it, and try to connect a client
Status	Pass

Test Name	**test_add_remove_node()**
Description	Initialize one server. Add another server. Remove the first server which was initialized. Then, try to connect a client to the removed server.
Status	Pass

Test Name	**basic_metadata()**
Description	Initialize 2 servers. Check if the computed metadata matches a hard-coded expected golden value (found through printing hash values).
Status	Pass

Test Name	**addNode_metadataUpdate()**
Description	Initialize 2 servers. Check if the computed metadata matches a hard-coded expected golden value (found through printing hash values). Add another node. Check if the computed metadata matches the expected golden value. The purpose of this test is to check if a new node's hash range is added to the metadata and if the successor node's hash range is updated as expected.
Status	Pass

Test Name	**removeNode_metadataUpdate()**
Description	Initialize 3 servers. Check if the computed metadata matches a hard-coded expected golden value (found through printing hash values). Remove the second node (index - 1). Check if the computed metadata matches the expected golden value. The purpose of this test is to check if second node was removed from the metadata and the successor node's hash range was updated as expected in the metadata.
Status	Pass

Test Name	**test_stop()**
Description	Initialize 1 server then stop it, and try to do a client request
Status	Pass

Test Name	**removeNode_dataTransfer()**
Description	Initialize 3 server and put keys into each. Then remove 2 servers and try to do a get from the remaining server.
Status	Pass