ECE 419 Design Document - Milestone 1

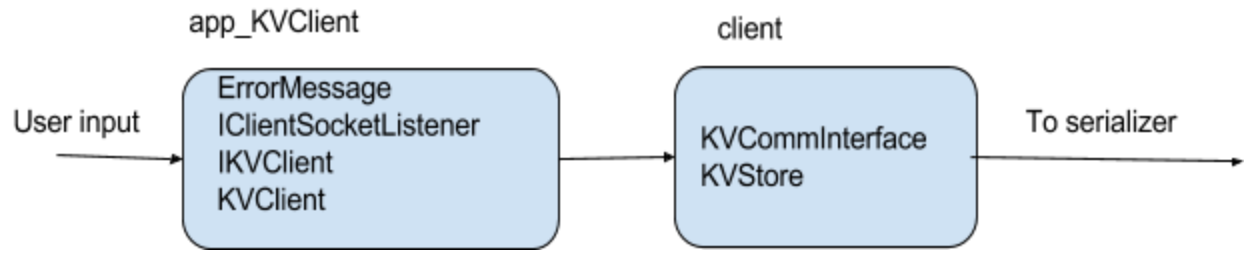
Abdelrahman Osama | 999804476

Hengyue Chen | 1000011667

Zi Chien Chew | 1000757406

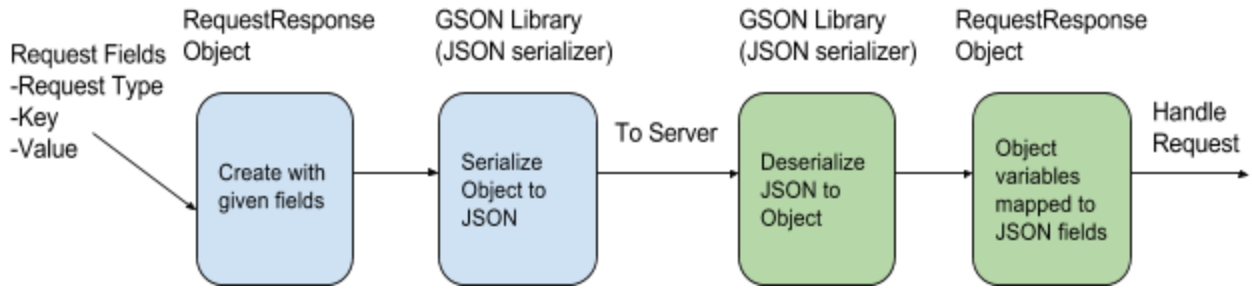
Design

Client



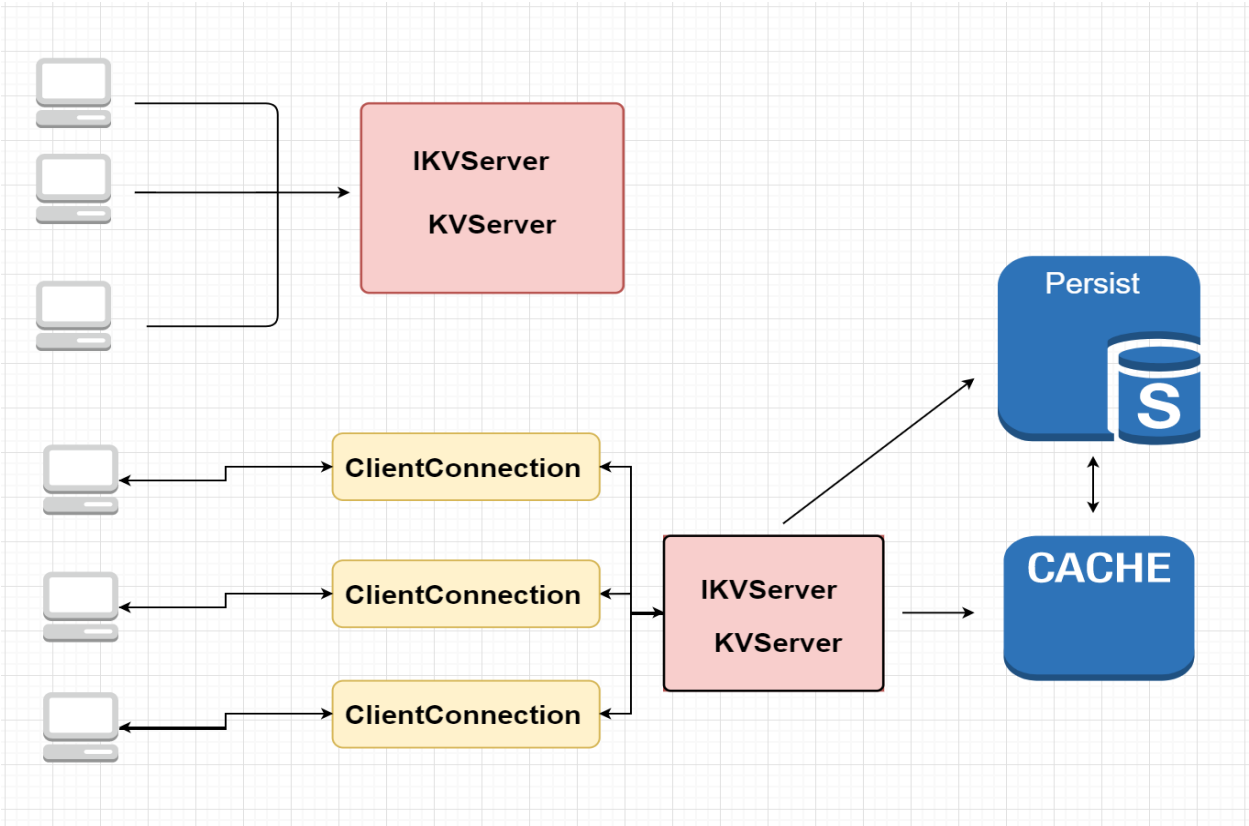
- Client application is single-threaded
- KVClient is the main class for the client application, which implements the IKVClient and IClientSocketListener interfaces
- KVStore is used to create new connections with server application
- IClientSocketListener is used to output messages from client to client terminal (app_KVClient). It may be updated with more features in later milestones
- Error message is used by the client application to output error messages to client's terminal
- KVStore, which implements KVCommInterface, is used to communicate with server using get and put requests through a JSON serializer(see next section for more details)

Serializer



- Client makes RequestResponse (request in this case) object using fields provided by client
- Client uses GSON library to serialize the object to JSON and sends it through the socket
- Server receives JSON string and deserializes it to RequestResponse (request in this case) Object
- The opposite is true when the server is responding by creating a RequestResponse object (response in this case), then using json to serialize it and sends it back to client to deserialize it.

Server



- Server accepts connection requests from clients by opening independent “ClientConnection”s for each client connection request
- Server performs simple validation of client get/put requests sent to KVServer:
    - All get requests are passed to cache directly
        - Cache will check if it has the requested key. If key is not present in cache or if cache was not initialized properly, key is obtained from Persist (disk). In all cases, excluding when cache is not initialized, cache will be updated using a selected cache strategy. Strategies include LRU, LFU and FIFO.
        - If Cache does not find the requested, it will return null
    - All put requests are passed to Persist directly and Persist updates the cache
- KVServer will reply to client with an appropriate response through ClientConnection

Design Decisions

KVClient

- User input validation is performed by the client (KVClient)
- If client request is invalid (eg. key length is excessive), it will not be sent to server.
- KVStore and KVServer check briefly for validity of commands (Req/Resp)

- If an error occurs in server (ie, save put failed as it was not able to open a file on disk), server will respond with “SERVER_ERROR” not “PUT_ERROR”
- GET_ERROR is only displayed if key does not exist in server. Otherwise, “SERVER_ERROR” is displayed

KVServer

- Server will exit if Persist fails to initialize as it does not function without it. However, it will display a warning if cache is not initialized properly as the server can still function without a cache
- Persist creates 27 files; first file will store all keys starting with ‘a’, second file will save all keys starting with ‘b’, etc. The 27th file stores any other key that does not start with an alphabet letter
- JSON is used to serialize Request/Response object that is sent through sockets

JUnit Tests & Performance Report

The details of our JUnit tests can be found in Appendix A. All tests are passing.

The raw output regarding the performance of our server can be found in Appendix B.

Since different caching strategies are geared towards specific user behaviour(e.g. LFU would work well when there are few keys in a batch that are accessed more often than others), we decided to pick keys at random to better assess the speed of caching operations. Therefore, we used 400 total requests and 100 unique keys selected at random(random number function is excluded from timing) for every ratio batch.

Three caching methods(FIFO, LRU, and LFU) were tested at cache sizes 10, 50, and 100. Looking at the results in Appendix B, we notice a positive correlation in time per average request and the ratio of “put” requests. However, there is marginal difference with an increase of cache size from 10 to 100(the total possible number of unique keys). We feel that the reason for this is due to two things, the first being the small number of data stored on our database and the second being too few connections and requests that the filesystem is always readily available.

Appendix

Appendix A: JUnit Tests

ConnectionTest.java	
Function	**Description**
testConnectionSuccess()	Tests a valid host and port number to connect to, should return no exceptions.
testUnkownHost()	Tests connecting to an invalid host, should return UnkownHostException.
testIllegalPort()	Tests connecting to an invalid port, should return IllegalArgumentException.
InteractionTest.java	
testPut()	Tests a valid put request. Should return no exceptions.
testPutDisconnected()	Tests put after client has been disconnected. Should return an exception with our custom message “Not Connected”.
testUpdate()	Tests update feature (putting the same key with different values twice). Should return the value of the update.
testDelete()	Tests deleting(put(<key>, null)) a key after it is put. Should return DELETE_ERROR.
testDelete2()	Tests deleting(put(<key>, “”)) a key after it is put. Should return DELETE_ERROR.
testGet()	Tests get after put of a key. Should return no exceptions and the value we put.
testGetUnsetValue()	Tests getting a value for a key that does not exist, should return GET_ERROR.
AdditionalTest.java	
testKeyFileLookUp()	Checks that persistence is actually working.

testReconnect()	Tests a disconnect and a reconnect to the same server port. Should not throw any exceptions.
testMultipleConnections()	Tests 50 client connections on the server. Should not throw any exceptions.
testPersistGet()	Tests a “get” for a key that was “put” before a disconnect and a reconnect. Should return the GET_SUCCESS and the correct value.
testDoubleUpdate()	Tests updating the same key twice with different values. Should return the value of the second update.
testGetAfterDelete()	Tests getting a key after it is deleted. Should return GET_ERROR.
testDoubleDelete()	Tests deleting(put(<key>, “”)) a key after it is deleted once. Should return DELETE_ERROR.
testDoubleDelete2()	Tests deleting(put(<key>, null)) a key after it is deleted once. Should return DELETE_ERROR.
testDeleteNonExistant()	Tests deleting(put(<key>, “”)) a key that should not exist in the server. Should return DELETE_ERROR.
testDisconnectedPut()	Tests put after client has been disconnected. Should return an exception with our custom message “Not Connected”.
testPersistUpdate()	Tests a “get” for a key that was “put” twice before a disconnect and a reconnect. Should return the GET_SUCCESS and the updated value.
PerformanceTest.java	
testCacheSize10()	Checks the timing for 400 put/get requests on FIFO, LRU and LFU caching server with cache size 10 and put/get ratios of 80/20, 50/50, and 20/80. More info in Appendix B.
testLRU()	Checks the timing for 400 put/get requests on FIFO, LRU and LFU caching server with cache size 50 and put/get ratios of 80/20, 50/50, and 20/80. More info in Appendix B.
testLFU()	Checks the timing for 400 put/get requests on FIFO, LRU and LFU caching server with cache size 100 and put/get ratios of 80/20, 50/50, and 20/80. More info in Appendix B.

Appendix B: Performance Test

Note all ratios below are in put/get format
Total Number of Unique Keys is 100

<xxxxxxxxxxxxxxxxxxxxCache Size 10xxxxxxxxxxxxxxxxxxxx>

<====================FIFO====================>
Average time taken per 80/20 request is: 11.0ms
Average time taken per 50/50 request is: 9.0ms
Average time taken per 20/80 request is: 3.0ms
<====================LRU====================>
Average time taken per 80/20 request is: 12.0ms
Average time taken per 50/50 request is: 7.0ms
Average time taken per 20/80 request is: 3.0ms
<====================LFU====================>
Average time taken per 80/20 request is: 19.0ms
Average time taken per 50/50 request is: 7.0ms
Average time taken per 20/80 request is: 6.0ms

<xxxxxxxxxxxxxxxxxxxxCache Size 50xxxxxxxxxxxxxxxxxxxx>

<====================FIFO====================>
Average time taken per 80/20 request is: 16.0ms
Average time taken per 50/50 request is: 8.0ms
Average time taken per 20/80 request is: 2.0ms
<====================LRU====================>
Average time taken per 80/20 request is: 10.0ms
Average time taken per 50/50 request is: 7.0ms
Average time taken per 20/80 request is: 2.0ms
<====================LFU====================>
Average time taken per 80/20 request is: 10.0ms
Average time taken per 50/50 request is: 6.0ms
Average time taken per 20/80 request is: 3.0ms

<xxxxxxxxxxxxxxxxxxxxCache Size 100xxxxxxxxxxxxxxxxxxxx>

<====================FIFO====================>
Average time taken per 80/20 request is: 11.0ms
Average time taken per 50/50 request is: 9.0ms

Average time taken per 20/80 request is: 2.0ms
<====================LRU====================>
Average time taken per 80/20 request is: 10.0ms
Average time taken per 50/50 request is: 6.0ms
Average time taken per 20/80 request is: 2.0ms
<====================LFU====================>
Average time taken per 80/20 request is: 11.0ms
Average time taken per 50/50 request is: 6.0ms
Average time taken per 20/80 request is: 2.0ms