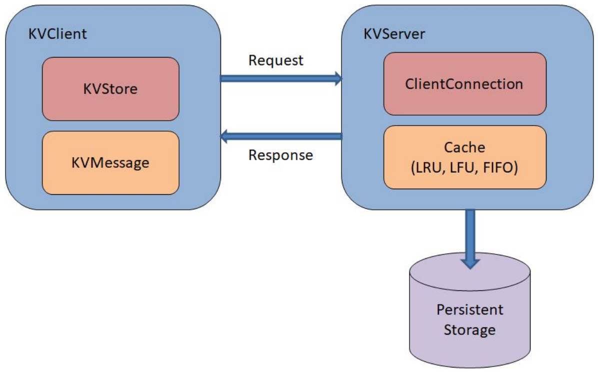
ECE419 Design Document Milestone 1

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

Design Overview



Code Format and Setup

<u>Client Design:</u>

- KVClient.java: Sets up the command line interface used by the client to interact with the storage server, do checks on input from user
- KVStore.java: Provides a library of functions to be used by KVClient, in order to get and put requests, connect and disconnect, and various other tasks
- ClientSocketListener.java: Function declarations for socket logic, imported from M0
- TextMessage.java: Serialization and marshalling logic used to send messages from client to server, imported from M0
- KVMessage.java: Message format used by client to decipher the string received from the server and compartmentalize into status, key, and value

<u>Server Design:</u>

- KVServer.java: Server running as a thread that accepts client connection
- ClientConnection.java: Handles send/receive function to communicate with the client. Also parses client command and calls appropriate function
- Fifo_Cache.java:
- IKVServer.java: Interface for the functions in KVServer
- Lfu_Cache.java:
- Lru_Cache.java:
- persistentDb.java: Handles all functions required to operate on the file stored in disk such as get, put key value pairs.
- TextMessage.java: Serialization and marshalling logic used to send/receive messages from server to client, imported from M0
- TServer.java: Simple class that extends Thread so that the server can be run on an individual thread.

Message Format

The message sent from between the client and server is sent as a stream of bytes (in ASCII characters). To indicate the end of a message, a newline character (i.e., '\n', 0x0D) is used. The server parses the message until it sees the newline character. As per the lab handout, the max message size is 128kBytes.

Client Message Protocol:

The protocol used to send commands from the client to the server are listed below with a brief description of their function. The server returns a status with the value or an error. The format of the command is <get> <key> or <put> <key> <value>.

CLI Command	Function
put <key> <value>	Inserts the key and value pair in the server storage, specifically onto the disk and cache. If the key already exists, then the value of the key is updated. If the value is null or an empty string, the key value pair is deleted from the server storage.
get <key>	Retrieves the specified key from the server storage. If the key does not exist in the cache or disk, an error message is returned to notify the user. If the key is found, the value is returned.

Server Message Protocol:

Upon receiving the request from the client, the server processes it and returns a success or failure message. The table below outlines the command and the message returned by the server.

CLI Command	Server Success Response	Server Failure Response
put <key> <value>	PUT_SUCCESS < key, value >	PUT_ERROR < key , value >
put<key> <“”/”null”/null>	DELETE_SUCCESS< key >	DELETE_ERROR < key >
put <existing_key><value>	UPDATE_SUCCESS < key, value>	PUT_ERROR < key , value >
get <key>	SUCCESS < key, value >	GET_ERROR < key , value >

Persistent Storage

The server uses a file to store key value pairs in the disk. All values inserted are added to the persistent storage file named “persistentDb.txt”. They are stored in the following format: “key : value\n”. Upon a get request, the cache is first checked for the key, if the key does not exist in the cache, then the persistent storage must have the key, value pair (if inserted before). The table outlines the high level algorithm for the operations on the persistent storage:

Operation	High Level Algorithm

put <key> <value>	Add the key, value pair to “persistentDb.txt”. The order of key, value pairs inserted is maintained so that the most recent is appended to the end.
put<key> <“”/”null”/null>	The key, value pair is deleted from the persistent storage and cache if it exists.
put <existing_key><value>	The key, value pair is first deleted from the file and the new key, value pair is appended to the file
get <key>	Finds the key in the cache first based on the policy specified by the user. If not found in cache, it searches for it in the file and returns the value. If the key does not exist in the file, it throws a NameNotFoundException exception
inStorage(key)	Return true if key found in persistent storage file.
clearStorage()	Overwrites the previous file with a new file that is empty

Performance Evaluation

To test for performance, we created 10,000 Total Requests for each type of cache. We tested 3 different sizes of cache: 10, 100, and 500. The results of the total latency, throughput for put and get requests are shown in the table.

Cache Size 10	FIFO			LRU			LFU		
Request Ratio of Put/Get	80%/ 20%	50%/ 50%	20%/ 80%	80%/ 20%	50%/ 50%	20%/ 80%	80%/ 20%	50%/ 50%	20%/ 80%
Latency in seconds	35.921	37.030	12.379	75.280	49.924	14.966	151.660	106.788	14.704
Throughput of put requests (req/sec)	249.43	259.197	605.900	109.568	155.401	371.340	53.522	55.726	410.872
Throughput of get requests (req/sec)	519.62	281.842	881.145	882.054	281.696	835.042	912.569	293.003	813.311

Cache Size 100	FIFO			LRU			LFU		
Request Ratio of Put/Get	80%/ 20%	50%/ 50%	20%/ 80%	80%/ 20%	50%/ 50%	20%/ 80%	80%/ 20%	50% / 50%	20%/ 80%

Latency in seconds	31.035	29.291	14.983	59.404	32.555	19.620	156.414	123.130	15.161
Throughput of put requests (req/sec)	277.085	296.277	308.899	140.024	238.910	194.708	51.912	44.453	326.125
Throughput of get requests (req/sec)	924.458	402.714	940.210	880.275	430.008	855.697	865.881	469.301	886.066

Cache Size 500	FIFO			LRU			LFU		
Request Ratio of Put/Get	80%/ 20%	50%/ 50%	20%/ 80%	80%/ 20%	50%/ 50%	20%/ 80%	80%/ 20%	50% / 50%	20%/ 80%
Latency in seconds	42.495	34.579	17.187	84.744	54.109	16.986	157.208	103.567	81.234
Throughput of put requests (req/sec)	201.806	272.094	261.456	97.889	132.158	315.0485	51.938	54.217	29.142
Throughput of get requests (req/sec)	700.968	308.584	838.767	662.391	307.205	752.019	629.202	440.700	634.619

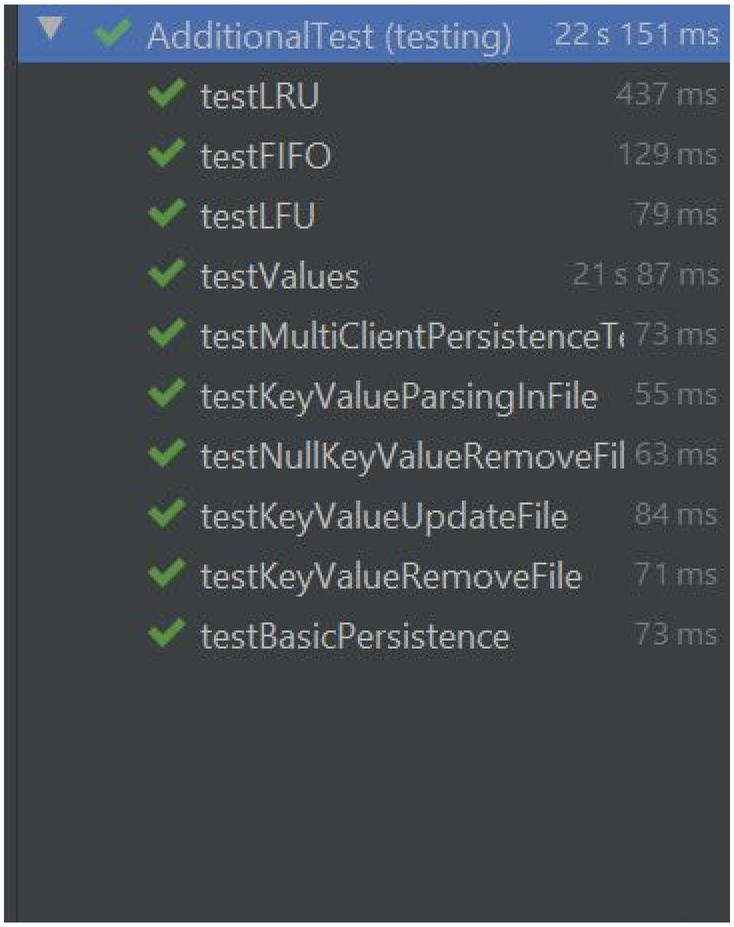
Appendix

Test Report

Cache testing: we created tests for each type of cache: lru, fifo, and lfu. We made put requests and get requests of different frequency and order to test the caches

Persistent storage testing: We tested different cases of persistent storage such as basic storage after put requests to check the file has been written to. We also tested for updates, and deletes to the persistent storage.

Values testing: We have created tests to ensure the values being inserted by user do not exceed the maximum number of characters.



```
▼ ✓ AdditionalTest (testing)        22 s 151 ms
      ✓ testLRU                         437 ms
      ✓ testFIFO                        129 ms
      ✓ testLFU                          79 ms
      ✓ testValues                   21 s 87 ms
      ✓ testMultiClientPersistenceT(     73 ms
      ✓ testKeyValueParsingInFile        55 ms
      ✓ testNullKeyValueRemoveFil        63 ms
      ✓ testKeyValueUpdateFile           84 ms
      ✓ testKeyValueRemoveFile           71 ms
      ✓ testBasicPersistence             73 ms
```