Ali Azhar 1000616620
Waref Haque 1000265615
Jeffrey (Yufei) Hua 1000067205

M1 Project Report

Our Design and Design Decisions Made

Client:

The client application is implemented based on echo client with the following modification.

Command	Design Decisions
put <key> <value>	Key does not allow white space. If user inputs "null" as value, the server will be requested to find the key and deleted it. Since the server interprets **put <key>** as the command to delete. The client application will filter out the value of "null" and send the rest to the server.
get <key>	Key does not allow white space.

Server:

The server builds upon Echo Server provided in Milestone 0. It is able to handle multiple clients in concurrently. The message sent from the client is parsed in the receiveMessage() function in the ClientConnection file. The message passed into the server must be of the type:

put <key> <value>

get <key>

Where the value can have multiple strings associated with it or null if the user wishes to delete a value.The server uses KVServerListener to communicate the parse client message to the KVServer which in turns invokes appropriate functions to save the data to a file. The server returns a string to the client which has the appropriate message such as:

PUT_SUCCESS <key,value> for a successful put request

PUT_ERROR<key,value> for a faulty put request

The server requires that when the KVServer is initialized for usage or testing the KVClient, it should be initialized with the .start().This was done for ease of multithreading and allowing the server to be tested independently of any input when initialized without .start() in the JUnit test cases.

Key-Value Database System:

The database system was initially built to store the key-value pairs in a file called "storage.txt". They are stored in the file line by line in the format:

key1 value1

key2 value2

We utilized a BufferedReader object to read the file line by line, while we used the PrintWriter object to write a key-value pair onto a line of the file. A put request where the

key-value pair was not in the file basically involved using the PrintWriter object to append a line containing the key-value pair to the end of the “storage.txt” file.

When it comes to a put request that involves a key-value pair that is in the file, it is always considered a put update. In order to modify the value for a particular key, we initially create a temporary file that will contain all the original key-value pairs and the updated one. We use the BufferedReader object to read each line in the original file and utilize the space between the key and value to access the key (in order to write this line with the new value). When we go through each line, we add the key-value pairs to the temporary file, including the modified key-value pair. We then delete the original file and rename the temporary file to “storage.txt”.

When it comes to a delete, we do the exact same thing as we did in put update, but once we get to the line with the key that corresponds to the one in the delete, we do not put the line in the temporary file (essentially deleting it).

For a get request we simply search for the key in the file using the BufferedReader object, and once we hit the line with the corresponding key, we utilize the space and split the string to access all other strings that are not the first one (they make up the value). The value is concatenated and returned.

The cache system is two List<String> data structures. One holds all keys while the other holds all values forming the pairs based on same indexes in the List data structures. Using two data structures was less complex than making a Pair Class and implementing compare and other methods needed by the powerful List object. The FIFO replacement policy was implemented such that the element at index 0 was “First In”. Every other element was appended to the end of the list. If the lists were full, we would remove the key-value pair at index 0 as they were the first pair added to the list. Once removed, the second pair at index 1 now shifts to index 0 and becomes the “First In”.

For the LRU replacement policy, the key-value pair at the first index is usually the least recently used. Once a key-value pair is reused through a “put update” or “get”, if it is in the cache, it is removed from its’ current position and added to the end of the list because it is the most recently used. If it is not in the cache, the first key-value pair is removed as it is the least recently used (index acts like use identifier). A simple “put” command leads to a key-value pair being added to the end of the list if the cache has space (that becomes the most recently used).

For the LFU replacement policy, the frequency of use counter is stored at the end of the value once a key-value pair is stored in the two List data structures. This way we prevented the use of Tuples again, and through the use of string methods, and the fact that we know the last string is the counter, we can easily extract the counter and get the value. It was not placed with the key in its’ cache as the key is used to check whether the cache contains a key and thus a certain key-value pair. A simple “put” where the corresponding key-value pair was not in the cache will get added to the end of the list with a frequency counter value of 1 appended to the

value in its' cache. If the cache is full, we traverse through the List data structure holding all the values and extract one by one the frequency counter for each key-value pair. If the count is lower than the current lowest, we store the index of that value in order to remove the key-value pair if it has the lowest count. Then we add the new key-value pair with a frequency count of 1. A "put update" and a "get" will result in a incrementation of the frequency counter for that particular key-value pair if it is in the cache (otherwise for get, it's added to the end of the list). This is done by acquiring the index of that key-value pair and then extracting the counter from the value, incrementing it, and then adding it back again to the value and then to the List data structure.

Performance Evaluation

For a cache size of 100 elements (the test is located in the Appendix in Section 1), since we are only performing 50 operations, it is definite that all key-value pairs are probably in the cache. This means that get requests will be faster, while any put requests will be slower especially put updates because they modify the cache. For the 80% put, 20% get test for each replacement policy, we observe that FIFO is the fastest. This is probably because it does not modify the cache on put update. For the 50% put, 50% get test we observe LRU to be much faster. This is because LRU pushes key-value pairs already in the cache to the end of the list in get commands while FIFO does nothing in get commands, thus those reused key-value pairs are probably getting removed. So in FIFO, they have to get the value from the file and add it to the cache. For the 20% put, 80% get test we observe that LFU is the fastest. This is surprising because it is the most complex. This is the case because due to alot more get operations, there is a need for a better system to hold key-value pairs in the cache. The LFU is the most accurate system of the three (as LRU just puts key-value pairs at the end of the list, which is not the best system), just it becomes the fastest as it keeps the most used key-value pairs in the cache (the other policies cause the server to go to the file, which is slow).

For a cache size of 25 elements (the test is located in the Appendix in Section 1), it is obvious that now less of the fifty commands will fit in the cache than when the cache size was 100 (so it always fit). The 80% put, 20% get test shows again that LRU (rather than FIFO) is the best because the get commands first try to get the value from the cache and so the LRU is the best replacement policy as there are not enough get commands for the LFU to be effective enough (it is also more complicated during put commands). For the 50% put, 50% get test, the FIFO is the best because there are only 25 put commands and so they all fit in the cache. Since FIFO is the simplest policy it's also the fastest. For the 20% put, 80% get test we see that FIFO just edges it since again there are only 10 put commands therefore they all fit in the cache, and FIFO is the simplest and fastest of the three policies (does nothing on put update commands to the key-value pairs in the cache, except modify them, while the other policies move/modify them).

Ali Azhar 1000616620
Waref Haque 1000265615
Jeffrey (Yufei) Hua 1000067205

Appendix

Section 1:

Cache Size = 100 elements

[junit] Performance test 40 puts, 10 gets, FIFO, 100
[junit] elapsedTime: 1766 milliseconds
[junit] Performance test 40 puts, 10 gets, LRU, 100
[junit] elapsedTime: 1819 milliseconds
[junit] Performance test 40 puts + 10 gets, LFU, 100
[junit] elapsedTime: 1832 milliseconds

[junit] Performance test 25 puts, 25 gets, FIFO, 100
[junit] elapsedTime: 1214 milliseconds
[junit] Performance test 25 puts, 25 gets, LRU, 100
[junit] elapsedTime: 850 milliseconds
[junit] Performance test 25 puts + 25 gets, LFU, 100
[junit] elapsedTime: 1428 milliseconds

[junit] Performance test 10 puts, 40 gets, FIFO, 100
[junit] elapsedTime: 677 milliseconds
[junit] Performance test 10 puts, 40 gets, LRU, 100
[junit] elapsedTime: 688 milliseconds
[junit] Performance test 10 puts + 40 gets, LFU, 100
[junit] elapsedTime: 361 milliseconds

Section 2:

Cache Size = 25 elements

[junit] Performance test 40 puts, 10 gets, FIFO, 25
[junit] elapsedTime: 1537 milliseconds
[junit] Performance test 40 puts, 10 gets, LRU, 25
[junit] elapsedTime: 1104 milliseconds
[junit] Performance test 40 puts + 10 gets, LFU, 25
[junit] elapsedTime: 1226 milliseconds

[junit] Performance test 25 puts, 25 gets, FIFO, 25
[junit] elapsedTime: 780 milliseconds
[junit] Performance test 25 puts, 25 gets, LRU, 25
[junit] elapsedTime: 884 milliseconds
[junit] Performance test 25 puts + 25 gets, LFU, 25

[junit] elapsedTime: 1047 milliseconds

[junit] Performance test 10 puts, 40 gets, FIFO, 25
[junit] elapsedTime: 395 milliseconds
[junit] Performance test 10 puts, 40 gets, LRU, 25
[junit] elapsedTime: 399 milliseconds
[junit] Performance test 10 puts + 40 gets, LFU, 25
[junit] elapsedTime: 535 milliseconds
[junit] ------------ ---------- ----------

Section 3:

Test	**Description**
testGetCheck()	Tests whether you can retrieve a value from persistent storage after putting it there
testGetError()	Tests for when you are getting a key is not there
testUpdateCheck()	Does two puts and the latter one should say PUT_UPDATE instead of creating a new entry
testDeleteCheck()	Checking if a delete is successful by putting a value and then deleting it
testDeleteError(){	Check if an invalid delete fails by trying to delete a key that does not exist
testCacheCheck(){	Function puts a value in the cache and the DB and then checks specifically in the cache to check if the value was retrieved from the cache

Ali Azhar 1000616620
Waref Haque 1000265615
Jeffrey (Yufei) Hua 1000067205

void testCheckFifo(){	This checks if fifo works. Put 6 values. The 6th value kicks out the first value from the cache
testPutLargeKey() {	put <key><value>, <key> has a max length of 20 bytes <value> has a max length of 120kBytes check if a put-operation with a large key returns error
void testGetLargeKey() {	get <key>, <key> has a max length of 20 bytes check if a get-operation with a large key returns error
testPerformace()	check latency of different combination of cache size, cacheing strategy and put/get operations Look at chat

The Following is a sample output of the above test cases together with provided test cases:

```
[junit] Testsuite: testing.AllTests
[junit] Tests run: 20, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 20.536 sec
[junit]
[junit] ------------- Standard Output ---------------
[junit] kvClient created
[junit] KEY: foo2
[junit] VALUE: bar2
[junit] KEY: foo
[junit] VALUE: bar
[junit] KEY: updateTestValue
[junit] VALUE: initial
[junit] 2017-01-29 23:23:39,922 ERROR [Thread-2] root: Error! Connection lost!
[junit] 2017-01-29 23:23:39,922 ERROR [Thread-3] root: Error! Connection lost!
[junit] KEY: updateTestValue
[junit] VALUE: updated
[junit] 2017-01-29 23:23:39,972 ERROR [Thread-4] root: Error! Connection lost!
[junit] KEY: deleteTestValue
[junit] VALUE: toDelete
```

Ali Azhar 1000616620
Waref Haque 1000265615
Jeffrey (Yufei) Hua 1000067205

```
[junit] Deleting key deleteTestValue
[junit] Removing deleteTestValue and toDelete
[junit] [foo2, updateTestValue]
[junit] [bar2, updated]
[junit] 2017-01-29 23:23:40,018 ERROR [Thread-5] root: Error! Connection lost!
[junit] KEY: foo
[junit] VALUE: bar
[junit] KEY: foo
[junit] VALUE: bar
[junit] 2017-01-29 23:23:40,068 ERROR [Thread-6] root: Error! Connection lost!
[junit] 2017-01-29 23:23:40,069 ERROR [Thread-7] root: GET_ERROR
[junit] KEY: anunsetvalue
[junit] VALUE: null
[junit] 2017-01-29 23:23:40,070 ERROR [Thread-7] root: Error! Connection lost!
[junit] 2017-01-29 23:23:40,087 ERROR [main] root: GET_ERROR
[junit] Removing one and 52
[junit] []
[junit] []
[junit] 2017-01-29 23:23:40,436 ERROR [main] root: Temp Deletion: true
[junit] key exceeds max length 20 bytes
[junit] key exceeds max length 20 bytes
[junit] Performance test 40 puts, 10 gets, FIFO, 100
[junit] elapsedTime: 1761 milliseconds
[junit] Performance test 40 puts, 10 gets, LRU, 100
[junit] elapsedTime: 1811 milliseconds
[junit] Performance test 40 puts +  10 gets, LFU, 100
[junit] elapsedTime: 1796 milliseconds
[junit] Performance test 25 puts, 25 gets, FIFO, 100
[junit] elapsedTime: 1018 milliseconds
[junit] Performance test 25 puts, 25 gets, LRU, 100
[junit] elapsedTime: 1001 milliseconds
[junit] Performance test 25 puts +  25 gets, LFU, 100
[junit] elapsedTime: 622 milliseconds
[junit] Performance test 10 puts, 40 gets, FIFO, 100
[junit] elapsedTime: 314 milliseconds
[junit] Performance test 10 puts, 40 gets, LRU, 100
[junit] elapsedTime: 271 milliseconds
[junit] Performance test 10 puts +  40 gets, LFU, 100
[junit] elapsedTime: 654 milliseconds
```

Ali Azhar 1000616620
Waref Haque 1000265615
Jeffrey (Yufei) Hua 1000067205

[junit] Performance test 40 puts, 10 gets, FIFO, 25
[junit] elapsedTime: 2048 milliseconds
[junit] Performance test 40 puts, 10 gets, LRU, 25
[junit] elapsedTime: 1794 milliseconds
[junit] Performance test 40 puts + 10 gets, LFU, 25
[junit] elapsedTime: 1199 milliseconds
[junit] Performance test 25 puts, 25 gets, FIFO, 25
[junit] elapsedTime: 815 milliseconds
[junit] Performance test 25 puts, 25 gets, LRU, 25
[junit] elapsedTime: 1205 milliseconds
[junit] Performance test 25 puts + 25 gets, LFU, 25
[junit] elapsedTime: 1104 milliseconds
[junit] Performance test 10 puts, 40 gets, FIFO, 25
[junit] elapsedTime: 591 milliseconds
[junit] Performance test 10 puts, 40 gets, LRU, 25
[junit] elapsedTime: 736 milliseconds
[junit] Performance test 10 puts + 40 gets, LFU, 25
[junit] elapsedTime: 575 milliseconds
[junit] ------------ ---------------- ---------------