ECE419 Milestone 1 Design Document

Members:
- Tzu-An Chen 1000538411
- Dexter Rivera 1002433690
- Vlad Ilgovskiy 1001803195

Major Design Decisions

The majority of the project consisted of implementing the required features specified by the Milestone 1 document. However, some key design decisions were made which impacted the performance of the client-server persistent storage, which are listed below.

Message Protocol
To establish a message protocol between client and server, our team decided to use JSON for messages since it is a well-established and easy-to-use message format for modern server-client communication. To implement the JSON format message, we decided to utilized google's GSON library for serializing (from JSON object to string) and deserializing (from string to JSON object) messages. The class that represents the JSON format object can be found in shared/JsonMessage.java.

Persistent Storage
Our team decided to use log-based files for our persistent storage solution. The advantage for using log-based format is the file would be append-only, thus removing the need for constant overwriting of old values when updating an existing key. In addition, since the file containing the key value pairs is append-only, the entire file does not need to be read into memory everytime an update or delete occurs. The unit data for a Key-Value entry in the file is: [Key | Value | keyLength | valueLength]. To write or updating a key-value entry would just be appending the unit data at the end of file. To find the most updated value for a specific key, we start seeking from the end of the file. By reading in keyLength and valueLength (each 4 bytes), we could know where the offset where the previous entry ends and where current entry starts. (offset - 4 * 2 - key length - value length). We read the key and see if it matches the key we are searching for. If not, then we return to the offset where previous entry ends and start the process again.

Caching Policies
For the FIFO Caching policy, a simple queue data structure was implemented along with a HashMap. To get, delete or update a key, the HashMap is used to find which node in the queue the key is located, ensuring O(1) time. For the LRU Caching policy, the same data structure as the FIFO cache is used. On gets or updates, the HashMap enables us to find which node in the queue the key is located and move the node to the rear of the queue in O(1), preserving the least recently used ordering. For the LFU cache, a 2D queue structure with a HashMap is used. There is a single "frequency queue" where each "frequency node" in the queue contains the frequency of each key and a "FIFO queue" containing "item nodes" containing the key-value pairs corresponding to that frequency. In the LFU cache, if there is a tie in terms of the frequency of use for a key, the least recently used is evicted. The "frequency queue" is ordered in increasing order of frequency, with the least frequent at the front of the queue. To evict an element, the first node of the frequency queue is accessed, and from there the first node of its FIFO queue is accessed, which is then deleted. On gets or updates, a HashMap is used to find the "item node"

corresponding to the key in O(1) time, and from there the item node is deleted from that "FIFO queue" and placed in the "FIFO queue" of the "frequency node" to the right.

The data structures used for each caching policy were designed to guarantee O(1) get, put, update and delete operations to the cache. However, more memory is required to achieve this O(1) time due to the extra overhead of the HashMap and Queue data structures. Data structures with less memory overhead than the ones used required O(N) get or put time, there it was deemed a reasonable tradeoff to use these data structures.

Each caching policy implemented "write through" instead of "write back", meaning that on writes, the key-value pair is immediately written to disk instead of being first written to the cache then written to disk upon eviction. This is to ensure the integrity of the data in case the server crashes before the data is written to disk.

Performance Evaluation

To test the performance of the server for each caching policy, 3000 get/put operations with varying ratios were performed by the server for each policy and different cache sizes. Depending on the ratio, a number of key-value pairs were generated. For example, with a 50/50 get/put ratio, 1500 key-value pairs were generated. From there, they were randomly shuffled. The order of the 1500 put and 1500 get operations were also randomly shuffled. If it was time for a get operation, the key was randomly selected from the list of keys already placed in the storage so far. The following tables show the read and write latency for the three strategies for a 50/50 get/put operation on varying cache sizes. Latencies are in ms and calculated by dividing the total write/read time and dividing by 1500.

FIFO

Cache Size	Read Latency	Write Latency
25	0.08867	0.37667
50	0.03533	0.126
100	0.035333	0.129
250	0.0166666	0.1073333

LRU

Cache Size	Read Latency	Write Latency
25	0.089	0.456
50	0.046	0.147333
100	0.0347	0.10666667
250	0.01	0.07066667

LFU

Cache size	Read Latency	Write Latency
25	0.05533	0.145333
50	0.04333	0.207333
100	0.03933	0.16933
250	0.016666	0.1266667

The following tables show the read and write latency for the three strategies for a 100 cache size on varying get/put operations. Latencies are calculated in a similar manner as above.

FIFO

Ratio	Read Latency	Write Latency
20 get/80 put	0.026667	0.08125
50/50	0.0346667	0.128667
80/20	0.018333	0.111853

LRU

20 get/80 put	0.036667	0.0816667
50/50	0.0346667	0.1066667
80/20	0.01875	0.1001669

LFU

20 get/80 put	0.058333	0.192917
50/50	0.0393333	0.1693333
80/20	0.0225	0.1202

As can be seen from the tables, the large the cache size, the better performance on reads, as expected. Furthermore, the higher the number of gets vs. puts, the better performance on reads as well, which is also expected. However, it is difficult to see which caching policy is superior. This is most likely due to the fact that the get and put operations were random. Each policy may work better on a certain sequence of gets and puts. One extra test was done where the cache size was 100, the ratio of get/put was 50/50. Then 1500 kv pairs were first written, then 1500 get operations were performed, but only a set of 300 KV-pairs which were randomly selected. This policy is expected to favour the LRU and LFU caches over

the FIFO, which is what was seen as the read latency for the FIFO cache in this scenario was 0.023ms vs. 0.01 for both LRU and LFU.

Appendix: Additional Unit Tests

Name	Description	Pass/Fail
KVStorage Tests	Test basic functionality of KVStorage including writing KV entry to file, reading KV entry from file and checking if key exists in file	Pass
DLL Tests	A doubly linked list data structure was written from scratch to be used as a queue for each of the different caching policies. Unit tests consisted of inserting at the front, rear, deleting from rear, deleting a node in the middle and ensuring the size increased and decreased upon insertion/deletion.	Pass
Cache Test	Unit tests were written to test insertion, deletion, updates and gets as well as checking the inCache function for all caching policies.	Pass
FIFO/LRU/LFU Test	Tested to ensure each type of cache correctly implemented their eviction policies. This was done by putting and getting a sequence of key value pairs specifically designed to test each policy.	Pass