ECE419 Project: Milestone 1 - Design Document

Patricia Marukot (1002157499), Jing Yi Li (1002346730), Seongju Yun (1002275290)

Message Serialization

The structure of Message class (src/shared/messages/Message.java) is chosen as follows:

```
Public class Message {
Private String key;
Private String value;
Private StatusType status;
}
```

For marshalling and unmarshalling, each Message object was converted into a JSON string or vice versa using *Jackson* library's Object Mapper class. The input and output stream of the socket connections were casted into BufferReader and PrintWriter objects, which automatically does byte-string conversion. This design decision was mainly based on the ease of debugging and implementation.

Concurrent Cache Implementation (DSCache.java)

The cache supports both write-through and write-back writing policies, the default is write-through. Below describes the cache layout as well as the locking scheme.

Table 1: Cache entry properties

Property	Description
Key	Key which identifies the cache entry
Dirty	A cache entry is dirty if its corresponding data in the disk not in sync with the data in the cache. If a cache entry is dirty upon eviction, a write-to-disk is necessary. The opposite is true if a cache entry is clean (i.e. dirty=0).
Last Accessed	Keeps track of the last time the cache entry was accessed (put, update, get). This is to make LRU replacement policy possible.
Frequency	Keeps track of the number of times the cache entry was accessed (put, update, get). This is to make LFU replacement policy possible.
Order	A monotonically increasing integer ID unique for each cache entry. This enforces FIFO ordering of entries in cache. For example, EntryA is older than EntryB if and only if Order(EntryA) < Order(EntryB). This is to make FIFO replacement policy possible.
Lock	Entry-level lock to ensure atomicity of concurrent reader/writer-type accesses to the same cache entry.
Data	Actual data in string format. If dirty bit is set, this data is out-of-sync with what is in the disk.

Table 2: Example Cache Layout

Key	Dirty	Last Accessed	Frequency	Order	Lock	Data	
"ab"	0	12342321	3	1	1	"c2FkZmFkc2Z"	Entry 1
"1"	1	12342322	2	2	1	"MTIzMWxr"	Entry 2
…	…	…	…	…	…	…	…

Synchronization - Locking Scheme:
DSCache consists of 2 levels of lock: Global lock (*gl*), and entry-level lock (*el*). The locking scheme for **get** and **update** are as follows:

lock(*gl*)
entry = get entry from cache
lock(*entry.lock*)
unlock(*gl*)
…
unlock(*entry.lock*)

The intuition for the scheme is to first acquire *gl*, then acquire *el*. Once *el* has been acquired, *gl* can be unlocked to allow other threads to access the cache. No deadlock is possible here because a thread cannot acquire *el* unless it first acquires *gl*, which eliminates circular lock dependencies. Atomicity is guaranteed because entry-wise operations (**get**, **update** - but not **delete**, **insert**) can only be performed once a thread has acquired *el*.

The locking schemes for **delete** and **insert** are as follows:

Insert

lock(*gl*)
…
unlock(*gl*)

Delete

lock(*gl*)
entry = get entry from cache
lock(*entry.lock*)
delete CACHE[*entry*]
unlock(*entry.lock*)
unlock(*gl*)

For **delete**, *el* must also be acquired to make certain no **get**/**update** operations are occurring simultaneously.

Persistent Storage (Disk.java)

Major Decision: Used a single file to store all KV pairs - stored in *kv_store.txt*

File Design:

kv_store.txt ——————— key_1 value1 key_2 value2 …	Each row represents a key-value pair with the following format: `<key> <SPACE> <value>` When searching/inserting to disk, this format facilitates the parsing of the KV pairs from the storage file.

Locking Scheme
The storage file supports a multiple reader single writer synchronization scheme.

- Multiple readers can READ the persistent storage file at once but only one writer is allowed at a time.
- Uses the ReentrantReadWriteLock class that is provided by Java.
- For each read access to the persistent storage file *kv_store.txt* the requesting thread must acquire READ_LOCK. Multiple threads can acquire this lock at a time.

- For each write access, the requesting thread must acquire a WRITE_LOCK. If there are threads currently reading the file, the writeLock spins and must wait until all readers have released their locks. Once the WRITE_LOCK is acquired, threads requesting to READ the file will be blocked until the WRITE_LOCK is released.

Performance Evaluation

Note: Performance was evaluated using the System.nanoTime() method to measure the time it takes to send/receive PUT/GET requests to/from the server.

Test Details:

- Sent a total of 1000 requests to the KVServer with varying put/get ratios (20% put vs 80% get, 50% each, and 80% put, 20% get)
- Each of the ratios listed above were tested on 3 different cache sizes (note that the cache sizes listed are the **number of cache lines** - not the number of bytes) for each of the cache eviction policies

The values listed in the table are the total average latencies of all requests.

- The tests were run 3 times each for accuracy (ms)

Table 3: Total average latencies of all requests

		20% puts / 80% gets			50% / 50%			80% puts / 20% gets		
		FIFO	LRU	LFU	FIFO	LRU	LFU	FIFO	LRU	LFU
# entries	16	P: 1236 G:1251	P: 1173 G:1545	P: 791 G:1298	P: 3513 G: 910	P: 1471 G: 608	P:1571 G: 726	P: 7604 G: 410	P: 1775 G: 159	P: 2781 G: 892
	Total	2487ms	1718	2080	4423ms	2079	2297	8014ms	1934	3673
	512	P: 544 G: 463	P: 465 G: 584	P: 448 G: 559	P: 1571 G: 726	P: 1010 G: 283	P: 834 G: 303	P: 1712 G: 149	P: 1320 G: 554	P: 2229 G: 101
	Total	1007	1049	1007	2296	1293	1137	1861	1874	2330
	1024	P: 483 G: 735	P: 399 G: 474	P: 436 G: 494	P: 915 G: 239	P: 894 G: 219	P: 1110 G: 307	P: 1612 G: 266	P: 1432 G: 198	P: 1540 G: 87
	Total	1218	873	930	1154	1113	1417	1878	1630	1627

Table 4: Average time per PUT/GET request in milliseconds

		20% puts / 80% gets			50% / 50%			80% puts / 20% gets		
		FIFO	LRU	LFU	FIFO	LRU	LFU	FIFO	LRU	LFU
# entries	16	P: 6.18 G: 1.56	P: 5.87 G: 1.932	P: 3.955 G: 1.62	P: 7.026 G: 1.82	P: 2.942 G: 1.216	P: 3.142 G: 1.452	P: 9.505 G: 2.05	P: 2.22 G: 0.879	P: 3.476 G: 4.46
	512	P: 2.72 G: 0.579	P: 2.325 G: 0.729	P: 2.24 G: 0.7	P: 2.22 G: 0.755	P: 2.02 G: 0.755	P: 1.667 G: 0.605	P: 2.14 G: 0.746	P: 1.65 G: 2.77	P: 2.78 G: 0.507
	1024	P: 2.465 G: 0.919	P: 2.00 G: 0.592	P: 2.18 G: 0.617	P: 1.831 G: 0.478	P: 1.787 G: 0.438	P: 2.22 G: 0.615	P: 2.015 G: 1.33	P: 1.79 G: 0.992	P: 1.925 G: 0.435

In general, as we increased the cache size, the average **total time** for all put/get requests and time **per** request decreased with the lowest times with cache size = 1024 (i.e. all kv pairs fit into cache).
For almost all cases, FIFO performed the worst with the highest average latencies. LRU performed the best in most of the configurations with LFU following closely behind.

Appendix A: Given Test Cases

ConnectionTest.java

Test Case	Result
public void testConnectionSuccess();	PASS
public void testUnknownHost();	PASS
public void testIllegalPort();	PASS

InteractionTest.java

Test Case	Result
public void setUp();	PASS
public void tearDown();	PASS
public void testPut();	PASS
public void testPutDisconnected();	PASS
public void testUpdate();	PASS
public void testDelete();	PASS
public void testGet();	PASS
public void testGetUnsetValue();	PASS

Appendix B: Custom Test Cases

Cache Implementation (DSCache.java)

Test Description	Result

`public void testClearCache();`
Verify cache contents are flushed to disk upon call to **clearCache()**.
 	
Steps:
1. Initialize a cache of size 4, strategy = FIFO
2. Fill up cache using putKV(*key*, *value*)
  a. putKV("1", "one");
  b. putKV("2", "one_two");
  c. putKV("3", "one_two_three");
  d. putKV("4", "one_two_three_four");
3. Call clearCache()
4. Assert that all of the cache contents (2a-d) are present in persistent storage using Disk.getKV(*key*) method
	PASS

`public void testDelete();`
Verify all possible delete inputs are functioning.
 	
Steps:
1. Initialize a cache of arbitrary size with arbitrary replacement strategy
2. Fill up cache using putKV(*key, value*)
  a. putKV("1", "1265309548");
  b. putKV("2", "9665117208");
	PASS

<table>
<tr>
<td>

c. putKV("3", "3979847452");

3. Call putKV again on each inserted entry but with value intended for deletion
   a. putKV("1", "null");
   b. putKV(“2", null);
   c. putKV("3", “”);
4. Assert that all keys have been deleted from both cache and disk

</td>
<td></td>
</tr>
<tr>
<td>

`public void testWriteThrough();`
Verify that the write-through policy is persisting all putKV requests to disk.

Steps:

1. Initialize a cache of arbitrary size with arbitrary replacement strategy
2. Fill up cache with arbitrary values by calling putKV(*key, value*)
3. Assert that those values are in both cache and disk

</td>
<td>PASS</td>
</tr>
<tr>
<td>

`public void testCacheLRU();`
Verify that the LRU replacement policy is working.

Steps:

1. Initialize a cache of size 4, and strategy as LRU
2. Execute the following putKV commands
   a. **putKV("1", "1265309548")**; *// t=0*
   b. **putKV("2", "9665117208")**; *// t=1*
   c. **putKV("3", "3979847452")**; *// t=2*
   d. **putKV("4", "6531077644")**; *// t=3*
3. Execute the following getKV commands
   a. **getKV("1")**; *// t=4*
   b. **getKV("2")**; *// t=5*

At this time, the cache layout should be as follows:
(Assume time starts at t=0)

Key	Last Accessed	Frequency	Order	Data
1	4	2	1	**1265309548**
2	5	2	2	**9665117208**
3	2	1	3	**3979847452**
4	3	1	4	**6531077644**

4. Now the cache is full, execute the following insert (putKV) calls
   a. putKV("5", "6853866846")
   b. putKV("6", "0802567709")
5. Assert that keys “3” and “4” have been evicted and “5” and “6” are now in the cache

</td>
<td>PASS</td>
</tr>
<tr>
<td>

`public void testCacheFIFO();`
Verify that the FIFO replacement policy is functioning correctly.

</td>
<td>PASS</td>
</tr>
</table>

Steps:

1. Initialize a cache of size 4, with FIFO replacement policy
2. Execute the following putKV commands:
    a. **putKV("1", "1265309548");** *// t=0*
    b. **putKV("2", "9665117208");** *// t=1*
    c. **putKV("3", "3979847452");** *// t=2*
    d. **putKV("4", "6531077644");** *// t=3*
3. Execute the following getKV commands:
    a. **getKV("1");** *// t=4*
    b. **getKV("2");** *// t=5*

At this time, the cache layout should be as follows:
(Assume time starts at t=0)

Key	Last Accessed	Frequency	Order	Data
1	4	2	1	**1265309548**
2	5	2	2	**9665117208**
3	2	1	3	**3979847452**
4	3	1	4	**6531077644**

4. Since “1” and “2” were inserted first, they should be the ones to be evicted. To test this, issue the following putKV calls to insert new values “5” and “6”
    a. **putKV("5", "6853866846");** *// t=6*
    b. **putKV("6", "0802567709");** *// t=7*
5. Assert that “1” and “2” were indeed evicted and that “5” and “6” are now in the cache

---

`public void testCacheLFU();`

PASS

Verify that the FIFO replacement policy is functioning correctly.

Steps:

1. Initialize a cache of size 4, with LFU replacement policy
2. Execute the following putKV commands:
    a. **putKV("1", "1265309548");** *// t=0*
    b. **putKV("2", "9665117208");** *// t=1*
    c. **putKV("3", "3979847452");** *// t=2*
    d. **putKV("4", "6531077644");** *// t=3*
3. Prep the cache by executing the following getKV commands
    a. **getKV("1");** *// t=4*
    b. **getKV("1");** *// t=5*
    c. **getKV("1");** *// t=6*
    d. **getKV("1");** *// t=7*
    e. **getKV("1");** *// t=8*
    f. **getKV("2");** *// t=9*
    g. **getKV("2");** *// t=10*
    h. **getKV("2");** *// t=11*
    i. **getKV("3");** *// t=12*

j. **getKV**(**"3"**); *// t=13*
k. **getKV**(**"4"**); *// t=14*

At this time, the cache layout should be as follows:
(Assume time starts at t=0)

Key	Last Accessed	Frequency	Order	Data
1	8	6	1	**1265309548**
2	11	4	2	**9665117208**
3	13	3	3	**3979847452**
4	14	2	4	**6531077644**

Notice we've set up the cache so that the Most-Recently Used entry is also the Least Frequently Used: Take "4" for example, in LRU, "4" would not be evicted. but in LFU "4" is prime candidate because it has only been accessed 2 times while other entries have been accessed more than 2.

Conversely, "1", who would be prime candidate to be evicted in LRU is last-in-line to be evicted in LFU since it is the most frequently accessed entry by far.

We expect, therefore, that "4" be evicted first. Since the replaced entry will have accessFrequency of 1, this new entry will be prime candidate to be evicted again since it is LFU. As shown below, we expect both evictions to come from the same cache entry.

4. Execute **putKV**(**"5"**, **"6853866846"**)
5. Assert that “4” was evicted

After putKV(“5”, …) was executed, the cache should look like this:

Key	Last Accessed	Frequency	Order	Data
1	8	6	1	**1265309548**
2	11	4	2	**9665117208**
3	13	3	3	**3979847452**
5	14	1	5	**6853866846**

6. Execute **putKV**(**"6"**, **"0802567709"**)
7. Assert that “5” was evicted (since “5” is actually LFU in this case)

`public void testCacheToDisk();`

PASS

Verify that contents that are evicted from the cache are written to disk and are correctly brought back to cache upon a GET/PUT request.

Steps:

1. Initialize a cache of size 1, any replacement policy Alternate access to whichever entry is not in cache.
2. Generate 2 put events:

a. <Key A, Value A>: will be evicted upon 2nd put request b. <Key B, Value B>: will be in the cache 3. Call a GET request for Key A - entry should be retrieved from disk 4. Verify GET returns the correct value 5. Generate PUT request for A: <Key A, Value A> a. Key A should be brought into cache b. Key B should be evicted 6. Repeat steps 3-5 for Key B → Key A → Key B → … alternating between Key A and Key B as A and B continuously evict each other from the cache for 1000 iterations 7. For each GET request, the initial values (Value A, Value B) should be retrieved from the disk.	
`public void testCacheThreadSafety();` Verify thread safety (no deadlocks) of the cache. Spawn 10 updater threads, 10 reader threads, 10 deleter threads and run them with a small set of predefined keys selected at random for 1000 iterations. This should guarantee a high amount of concurrent accesses for each cache entry. This test simply checks for timeout - if the test takes an unusual amount of time to execute, assume deadlocked. If the test finishes, no deadlocks occurred and we conclude that the cache is lock free.	PASS

Disk Implementation (Disk.java)

Test Description	**Result**
`public void testDiskPutCreate();` Verify key value pair insertion into persistent storage. Verifies functionality of both putKV() and getKV() requests. *Steps*: 5. Insert key value pairs to persistent storage using putKV(key, value) Disk method 6. Call the getKV(key) method to retrieve the inserted data. 7. Assert that getKV returns the correct value. 8. Assert that Disk.inStorage() returns TRUE for the key 9. Repeat steps 1-3 for 1000 loop iterations	PASS
`public void testDiskPutUpdate();` Update existing key value pairs in persistent storage. Verifies functionality of both putKV() and getKV() requests. *Steps*: 1. Insert a single key value pair into persistent storage using putKV(key, value) 2. Using the same *key* from Step 1), call putKV(*key*, value) again with a different updated value.	PASS

3. Call getKV(*key*) method to retrieve the updated data 4. Assert that getKV returns the *updated* value 5. Repeat steps 2-4 for 1000 loop iterations	
`public void testDiskGetNone();` Call getKV for non-existent key value pairs. *Steps*: 1. Do not insert any key value pairs into persistent storage (i.e. no calls to putKV().) 2. Call getKV(key) for a random value of *key* 3. Assert that getKV returns NULL as the value does not exist in storage 4. Assert that Disk.inStorage() returns false 5. Repeat Steps 2-4 for 1000 iterations	PASS
`public void testDeleteDisk();` Delete key value pairs with putKV(*key*, NULL) request. Tested the following 2 cases for pair deletion: 1. *key* exists in storage 2. *key* DNE in storage - nothing to be done to disk For each case: 1. Assert that getKV(*key*) returns NULL 2. Assert that inStorage(*key*) returns FALSE	PASS

Communication Layer (server and client)

Test Description	**Result**
`public void testMultiClientInteraction();` Verify that the storage server is persistent and consistent even when it is serving multiple clients *Steps*: 1. Instantiate 3 clients 2. Client 1 calls `put hello world` 3. Client 2 calls `get hello` 4. Client 3 calls `put hello WORLD` 5. Client 1 calls `get hello` *Expected Results:* 1. From Step 3, Client 2 expects value = "world" (GET_SUCCESS) 2. From Step 4, Client 3 expects its request to be PUT_UPDATE 3. From Step 5, Client 1 expects value = "WORLD" (GET_SUCCESS)	PASS
`public void testTooLongKey();` Call putKV and getKV with a key that exceeds the maximum size of 20 bytes.	PASS

Steps: 1. Put a KV pair with key longer than 20 bytes and a normal string for the value 2. Put a KV pair with key longer than 20 bytes and an empty string for the value 3. Get a KV pair with key longer than 20 bytes *Expected Results:* 1. Assert that the reply message has PUT_ERROR status type 2. Assert that the reply message has DELETE_ERROR status type 3. Assert that the reply message has GET_ERROR status type	
`public void testTooLongValue;` Call putKV with a value that exceeds the maximum size of 120 KB. *Steps:* 1. Put a KV pair with a normal key and value that is bigger than 120 KB *Expected Results:* 1. Assert that the reply message has PUT_ERROR status type	PASS
`public void testEmptyString();` Call putKV and getKV with a key that is an empty string *Steps:* 1. Put a KV pair with an empty string as the key and a normal string as a value 2. Get a KV pair with an empty string as the key *Expected Results:* 1. Assert that the reply message has PUT_ERROR status type 2. Assert that the reply message has GET_ERROR status type	PASS
`public void testValueWithSpaces();` Call putKV with a value that has multiple words *Steps:* 1. Put a KV pair with a single-word key as the key and a multi-word value 2. Issue a get request to retrieve the value from Step 1 *Expected Results:* 1. Server response for getKV() contains the value given for the putKV from Step 1	PASS