ECE419 MILESTONE 1
DESIGN DOCUMENT

James Shen (999673875)
Shengjie Zeng (999751718)
Victor Chun Fung Ko (999741301)

Design Decisions

Communication logic

The message object has been redesigned to contain several fields:

- Command (enum: GET/PUT/UPDATE/DELETE)
- Key (string)
- Value (string)
- Status (enum: GET_SUCCESS/GET_ERROR/PUT_SUCCESS/PUT_ERROR...)

The serialization and deserialization uses JSON formatting whereby each of the fields is placed into the JSON object and serialized into a byte array before being sent across the network. The recipient can use the constructor:

KVMessage(byte[] messageBytes)

in order to deserialize the message. This constructor parses the JSON and reads the values back into the message fields.

The client creates a message by specifying the command, as well as the necessary key and value fields. The server, on receipt of the message, determines what appropriate action to perform given the command and replies with the appropriate key, value, and status code. At this point, our client receives the message and prints the status code and results to the command prompt for the user to see.

Caching and Cache Policies

The cache is created by storing cache objects in different list data structure depending on the caching policy. Each cache object contains its key, the value corresponding to the key, and a timesUsed variable that is incremented every time the object is accessed. Three cache policies were implemented which handled when a cache objected had to be evicted from the list data structure. This is how each policy was implemented:
Least Recently Used:
An arraylist stores the cache objects and whenever an object is accessed/modified, it is placed at the end of the list. The least recently used object will be at the head of the list.
First In First Out:
An arraylist is created, and stores the cache objects. The ordering does not change during modifications or accesses of the objects in order to preserve the chronological order. Eviction occurs by removing the object at the head of the list.

Least Frequently Used:
A priority queue is implemented which preserves the order by comparing the timesUsed variable and puts the lowest valued at the front of the queue. When an object is accessed/modified, its timesUsed variable is incremented and its position in the queue is reevaluated.

Persistent Storage

The persistent storage is implemented by storing key-value pairs onto disk, with each key being allocated it's own file containing the value it corresponds to. This design allows for concurrent accesses to disk given that the client requests operate on different keys. To provide this level of concurrent access, an in-memory concurrenthashmap is used to store a RW lock per key (key = filename on disk). All incoming client requests check for the existence of the RW lock corresponding to their key and acquire the lock to make changes to disk and cache if needed, releasing the lock when done.

There is a particularity that is worth discussing in more detail about the in-memory concurrenthashmap. Given its concurrent nature, it is possible to imagine the following situation occurring.

1. A GET command for key 1.
2. A DELETE command for key 1.
3. A PUT command for key 1.

where the numbering indicates the true order of events. Since in milestone 1 the order of the commands are subject to the OS scheduler the following can happen:

1. The GET thread takes the lock from the concurrenthashmap.
2. Before the GET thread can acquire the lock the DELETE thread takes the lock from the concurrenthashmap and acquires it and deletes the file, updates the cache, and deletes the key from the concurrenthashmap.
3. The PUT thread creates a new RW lock for key 1 and inserts a new key-lock pair into the concurrenthashmap.
4. The GET thread now has a stale lock different from the one in the concurrenthashmap.

One can also imagine scenarios of PUT and DELETE threads holding stale locks. These issues were addressed as follows:

- GET threads with stale locks return GET_ERROR. There were three ways to deal with a GET thread holding a stale lock: return the previous value, return the new value, or return a GET_ERROR. Since the order of requests are heavily influenced by the OS scheduler, there is no definitive way to differentiate between previous and new values, hence it was decided to return a GET_ERROR.

- DELETE threads with stale locks return DELETE_SUCCESS. Since the DELETE thread has a stale lock that must mean a second DELETE or PUT thread preempted it and has already modified the on-disk key and value. Hence, it is justifiable (again since the order of requests are heavily influenced by the OS scheduler) to say the DELETE was a success without needing to access disk or cache since a DELETE or PUT happened immediately after it and overwrote the DELETE.
- PUT threads with stale locks return PUT_SUCCESS. Same logic as with DELETE threads.

A particularity that is related to having a file per key on disk is that the storage server must ensure that the key is a legal filename. Since it is safe to assume ASCII values for the keys, the storage server converts the key into a hexadecimal string representation and prepends a special identifier and uses this new string as the filename (and also as the key in the concurrenthashtable).

Performance Testing

We measured the throughput of our system by creating worker threads that would all access methods of a shared storage handler. We vary the amount of worker threads created from 1 to 500 in order to see how the storage handler reacts to concurrent requests. Each thread runs a constant number of operations and we time how long it takes for all the operations to be completed. We varied the PUT/GET ratio to 20/80, 50/50, and 80/20. For each ratio, we gathered data for all three cache replacement policies, and on cache sizes of 128, 256, and 512. Please See Appendix A for the graphed data.

Unit Testing

Interaction Test

This test initializes a Client on a specific port and tests the end-to-end interaction from client command to parsed server response. Several scenarios are enumerated in our test cases:

1. Put
2. Put when disconnected
3. Update
4. Delete
5. Get on unset value

Connection Test

Tests various modes of operation as the client attempts to connect to the server. These test cases include:

1. Connection success
2. Unknown host
3. Illegal port

Cache/ Persistent Storage Tests

These tests govern the functionality of our StorageManager class, which encapsulates our cache management class and persistent storage methods. We initialize a StorageManager with an LRU cache policy and proceed to verify the following test cases:

1. Get existing key-value
2. Get non-existent key-value pair
3. Put a key with null value (delete)
4. Put on existing key-value pair (update)
5. Put a key with null value on non-existent key-value pair (delete should fail)

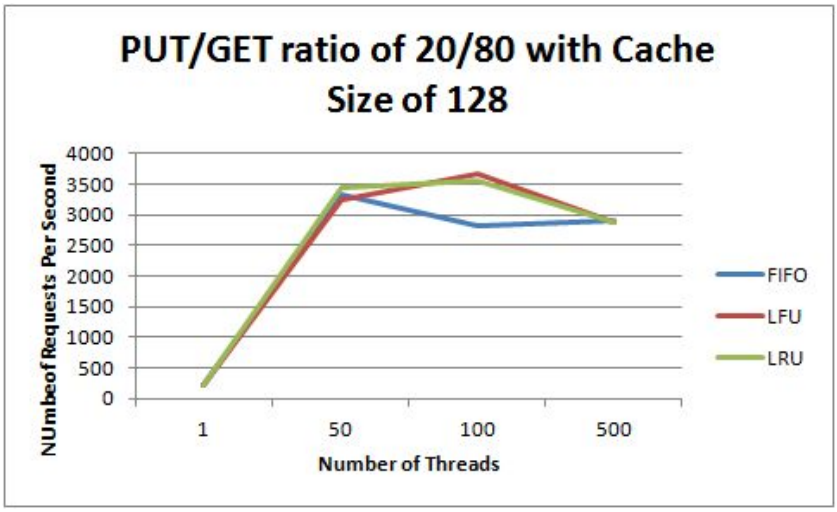
This is a more integrated tests which verifies the functionality of the flow from server API call to storage update, and the resulting value back to the server.

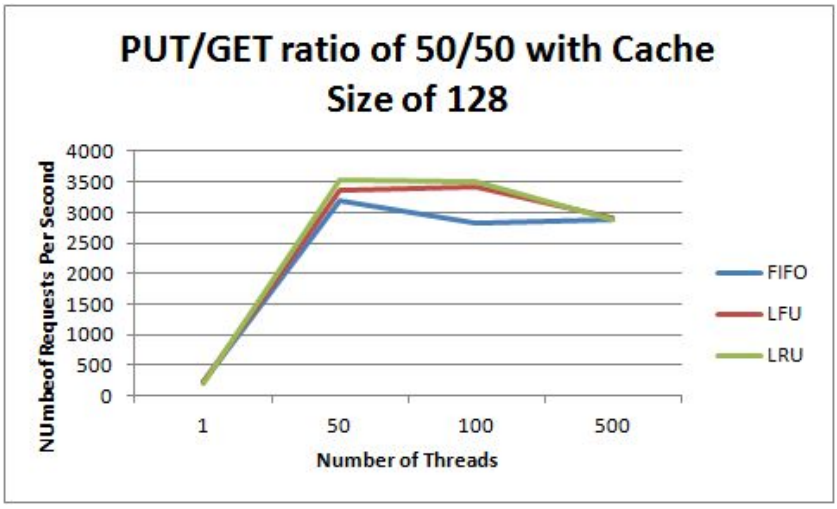
Cache Replacement Policy Tests

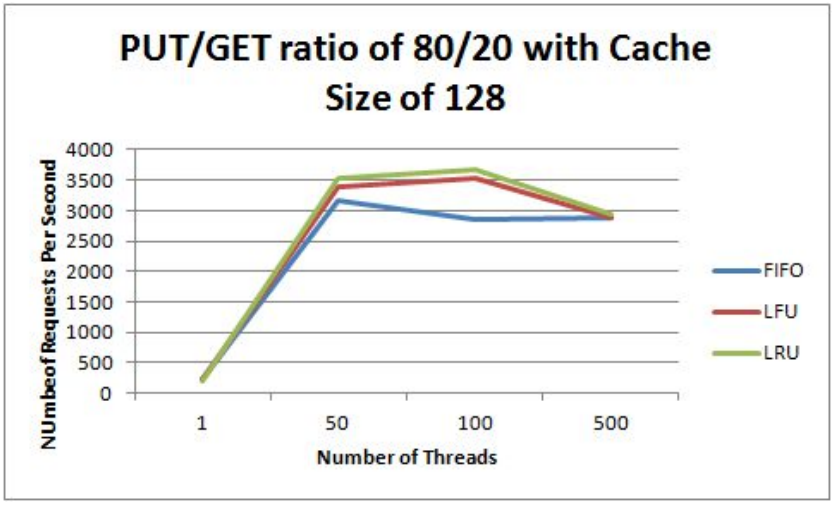
As our cache can operate in 3 different modes of operation : LRU, FIFO, LFU, we need to unit test the functionality of each. The tests instantiate a StorageManager with the given cache replacement policy, with a cache size of 10 elements. The test case then proceeds to put and update a number of key-value pairs, and observes the observed outcome in comparison to the expected outcome.

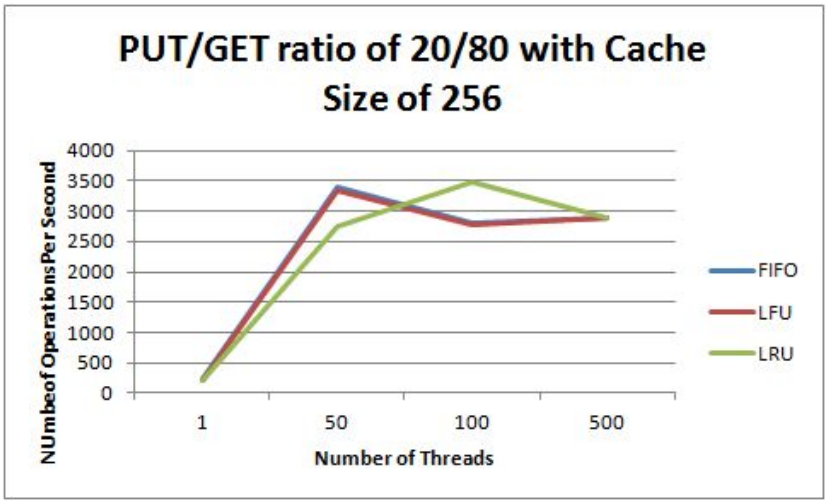
Appendix

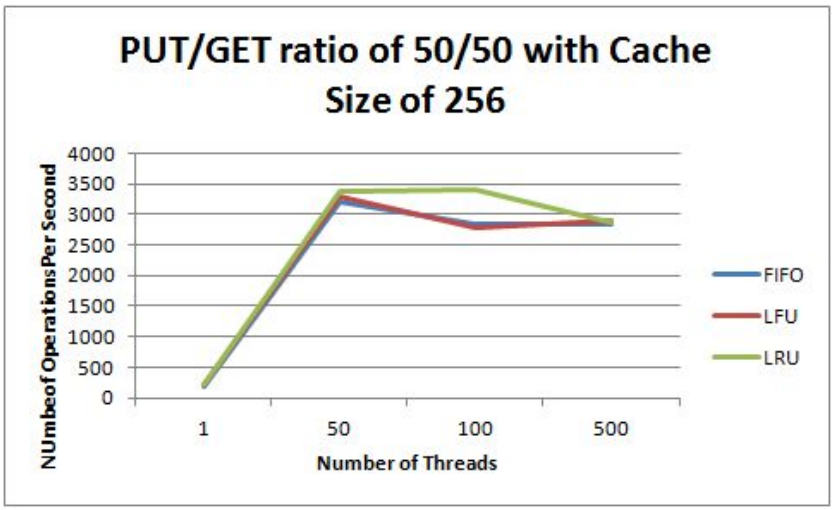
Appendix A

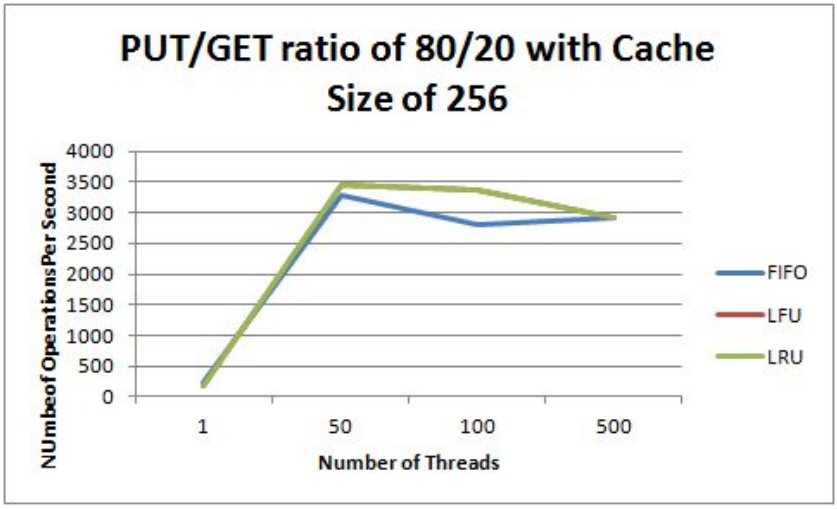


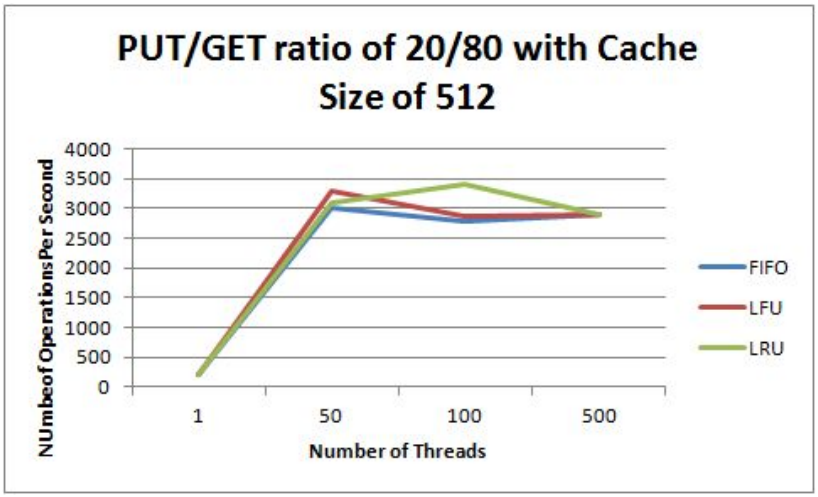


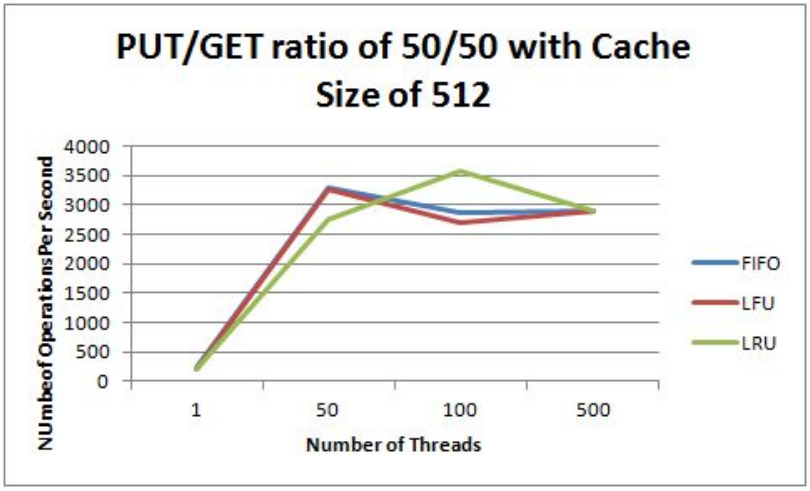

PUT/GET ratio of 80/20 with Cache Size of 128
NUmbeof Requests Per Second
4000
3500
3000
2500
2000
1500
1000
500
0
1
50
100
500
Number of Threads
FIFO
LFU
LRU

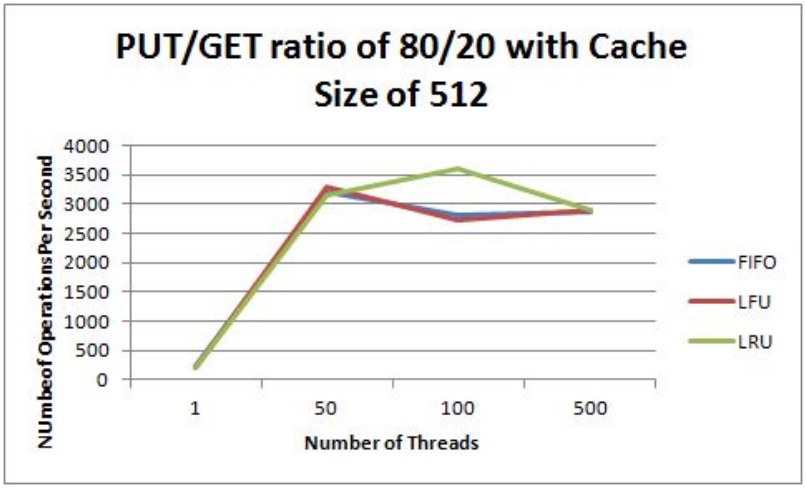

PUT/GET ratio of 20/80 with Cache Size of 256
NUmbeof OperationsPer Second
4000
3500
3000
2500
2000
1500
1000
500
0
1
50
100
500
Number of Threads
FIFO
LFU
LRU


PUT/GET ratio of 50/50 with Cache Size of 256
NUmbeof OperationsPer Second
4000
3500
3000
2500
2000
1500
1000
500
0
1
50
100
500
Number of Threads
FIFO
LFU
LRU


PUT/GET ratio of 80/20 with Cache Size of 256
NUmbeof OperationsPer Second
4000
3500
3000
2500
2000
1500
1000
500
0
1
50
100
500
Number of Threads
FIFO
LFU
LRU


PUT/GET ratio of 20/80 with Cache Size of 512
NUmbeof OperationsPer Second
4000
3500
3000
2500
2000
1500
1000
500
0
1
50
100
500
Number of Threads
FIFO
LFU
LRU


PUT/GET ratio of 50/50 with Cache Size of 512
NUmbeof OperationsPer Second
4000
3500
3000
2500
2000
1500
1000
500
0
1
50
100
500
Number of Threads
FIFO
LFU
LRU


PUT/GET ratio of 80/20 with Cache Size of 512
NUmbeof OperationsPer Second
4000
3500
3000
2500
2000
1500
1000
500
0
1
50
100
500
Number of Threads
FIFO
LFU
LRU