University of Toronto
Faculty of Applied Science and Engineering

ECE419
Milestone 1 Report

Team ID	B7
Date	January 28th, 2018
Project Title	ECE419 - Milestone 1
Prepared By (Names and Student #s of Team Members	Aya Elsayed [1000733469] Angel Serah [1000491773] Harshita Huria [1000980398]

In this document, we provide implementation and design decision details for the first milestone of ECE419 project. For milestone 1, we are required to design a single server that is able to communicate with multiple clients and concurrently service their requests. The server has a storage system that consists of a cache and a disk storage management system for persistently storing data. The document covers design decisions relating to client-server communication, the storage clients and servers, how we tested the system and an overall performance evaluation.

1. **Communication logic:**

A communication protocol was developed by using specific request and reply message formats. Messages are sent as <MessageType|Key|Value>. For GET requests sent from the client to the server, the format is <GET|key>. For PUT requests, it is <PUT|key|value>. Each of the GET and PUT request messages are marshalled such that the "|" character is a delimiter between the command, key, and value. Since keys and values may contain any ASCII character, we decided that using the character "|" as a delimiter as opposed to a more common character such as the comma "," makes the most sense. We do not allow keys to have "|" but values may contain the delimiter character. Server replies are also marshalled into a similar format. For GET replies the format is <GET_STATUS>|<value> and for PUT requests the format is <PUT_STATUS>.

2. **Storage server**

Based on the input, a server has a cache of the given size and replacement policy. Each server has a single cache and a single disk storage system. All data is accessible by all clients.
There are three types of replacement policies for caches. A First In First Out (FIFO) cache maintains a queue and evicts the piece of data that was first written to the cache when it reaches its size capacity. To implement the queue, we decided that the Java Queue data structure is not the most suitable. As our system accommodates deletion of data, and the cache must not have stale data in it to be consistent with the disk, we require a data structure that supports flexible random deletion, and using a Queue to implement that will be costly (order N), as we will have to create a second queue, pop the data off the first queue, one at a time, and add to the second queue except for the key value pair to be deleted. Instead, for the sake of efficiency, we decided to use a LinkedList<String> data structure to implement the FIFO cache as it provides a linkedlist.deleteFirst( ) function which is analogous to the queue.remove( ) function, linkedlist.add() where the end of the list is analogous to the back of the queue, but most importantly, it provides random deletion using linkedlist.remove(object). Even though this is still an O(n) function, this saves overcomplicating the implementation of delete node. To further save time, the linked list was only used to keep track of which key value pair is next in line to be evicted, but a separate hashmap data structure (kept in sync with the FIFO) was used for an O(1) lookup of key value pairs. This is done in order to make key and value queries faster.

The Least Recently Used (LRU) cache maintains a priority queue similar to the FIFO's except that accessing or updating a key value pair puts it back at the end of the queue. Like the FIFO cache, the queue was implemented using a LinkedList<String> and a hashmap was used for O(1) lookup and mapping of key value pairs.

The Least Frequently Used (LFU) cache maintains a frequency counter for each piece of data in it and increments the counter every time the key value pair is accessed or updated. It evicts the key value pair that was accessed the least, by using the counter with the least value. Two hashmaps were used to

implement the LFU cache. One maps the key value pairs and the other maps each key to its frequency counter.

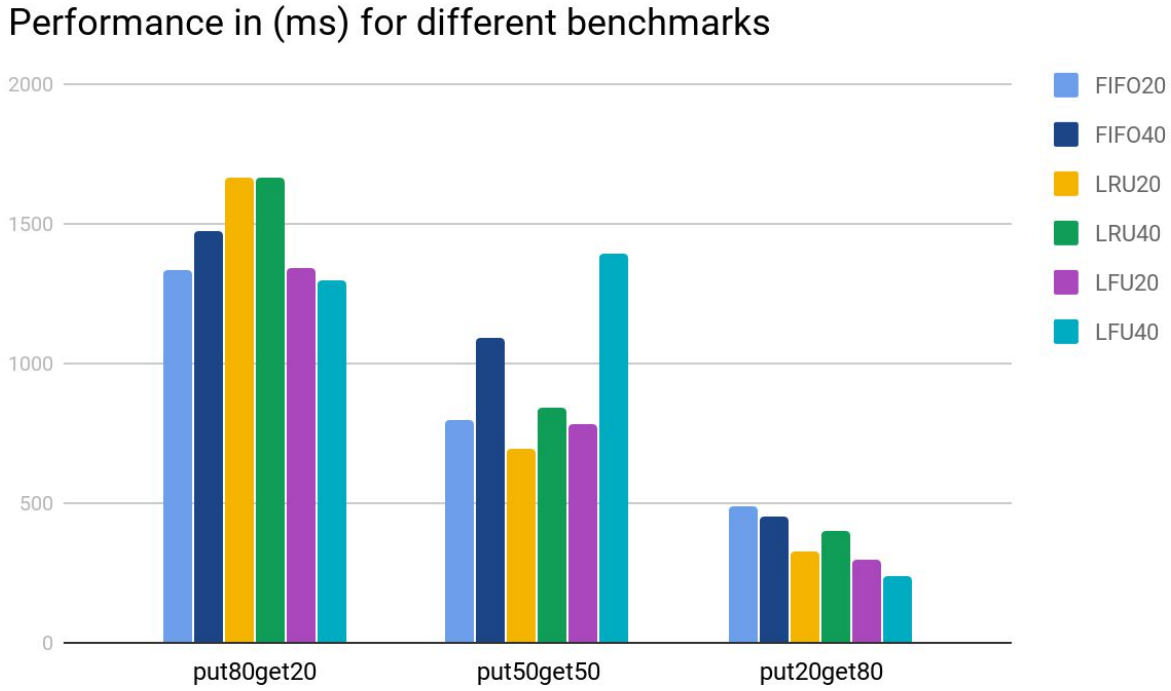
Access to the server's cache (reading and writing) is synchronized using the "synchronized" keyword to avoid race conditions between different client threads. Specifically, they were added to methods in KVCache which accesses the cache to ensure that other threads trying to access the cache at the same time are put to "sleep" until the active thread is done reading or writing. While currently, only a single reader is allowed, in the next milestone, we plan to allow multiple readers to read at the same time as there is no race condition hazard during reading.

For persistent storage of data, that is, data is saved even after the server is closed, we created a disk storage management system to save data to the disk. The disk management system consists of synchronized reads and writes to a txt file that contains a key value pair per line, comma separated. Initially, we thought that a quicker, simpler, and faster implementation will be to create a file per key value pair. This will eliminate the overhead of string parsing and file streams and buffers. However, since future milestones will have multiple servers, we decided that this implementation may not be the cleanest, most scalable solution. Hence, we decided to use a single text file per server, where the text file is named after the server's port number. Deleting is handled by reading the file and writing it back, one line at a time, excluding the key value pair to be deleted. Synchronization is implemented using locks, to ensure that no race conditions between two threads are possible. Like the cache, only a single writer or reader is allowed to access the disk at a time. In future milestones, we will allow multiple readers to read simultaneously.

3. **JUnit Tests:** Please refer to Appendix A for a table of tests added.
4. **Performance evaluation**

The following table reports the latency (in ms) for each of the following cache configurations and benchmarks

Cache configuration	**80% put, 20% get**	**50% put, 50% get**	**20% put, 80% get**
FIFO, 20 entries	1335ms	795ms	487ms
FIFO 40 entries	1475ms	1092ms	451ms
LRU, 20 entries	1662ms	695ms	329ms
LRU, 40 entries	1662ms	839ms	397ms
LFU, 20 entries	1340ms	784ms	297ms
LFU, 40 entries	1298ms	1394ms	241ms



The overall trend in the graph is consistent with the expected results. More gets that puts are faster. In our benchmark, most of the gets get a cache hit and their corresponding values are quickly retrieved. Most puts require cache eviction/replacement and disk access which is slow, so it significantly increases the run time.

Appendix A

To test all the different functionalities of our system, we added four categories of tests, in addition to the existing ones: Cache tests, invalid input tests, multiple client tests and performance tests (the former not automated). The table below summarizes what each automated test does.

	Test Name	**Functionality Tested**
1	CacheTests.testFIFOReplacement	Ensure the FIFO replacement policy works as expected. Ensures the cache contents are as expected at different stages of the test.
2	CacheTests.testLRUReplacement	Ensure the LRU replacement policy works as expected. Ensures the cache contents are as expected at different stages of the test.
3	CacheTests.testLFUReplacement	Ensure the LFU replacement policy works as expected. Ensures the cache contents are as expected at different stages of the test.
4	CacheTests.testEvictRetrieve	Ensure that data evicted from cache (onto the disk) are put back in the cache successfully when the user

		tries to read the evicted value
5	CacheTests.testClearCache	Tests that the clear cache function works as expected
6	AdditionalTest.testInvalidKeySpace	Ensure that we error out gracefully if the key contains a space
7	AdditionalTest.testInvalidKeyDelim	Ensure that we error out gracefully if the key contains the delimiter character “|”
8	AdditionalTest.testValidValueDelim	Ensure that we correctly interpret delimiters inside a value
9	AdditionalTest.testNonExistentKey	Ensure that we error out gracefully if a client sends a GET request for a key that isn’t stored on cache or disk
10	MultipleClients.TestDataAccess	Ensure that data put by one client is accessible by other clients
11	MultipleClients.TestDataUpdate	Ensure that data updates by one client is visible by other clients
12	MultipleClients.TestDataDelete	Ensure that data deleted by one client is no longer accessible by any client