File System Design Document

Design Overview

The goal for the design is to create a communication system for client and server. Overall, the design has been divided into 3 major parts: communication logic, storage server and storage client.

General Responsibility Of Each Component

The underlying communication logic, i.e. communication module, is shared between server and client for unified message transmissions in the form of bytes. The translation from bytes to json string is handled by KVStore in the client side and Client Connection in the server side respectively. KVServer's job is to listen on the designated port to accept new client request for connection and spawn new thread(ClientConnection) to handle incoming request. KVDB is responsible for handling incoming request for putting and getting key value pairs in the database with the support of cache in memory. KVDB and cache are both parts of the KVServer and is available to each ClientConnection spawned by KVServer to access their services.

Communication Logic (CommunicationModule)

We initially decided to store input into a object and transfer the input data to server using object serialization and deserialization. However, we were advised by Head TA not to do in this way because it will be java dependent if you use java,io,serializable. We then decided to use json as our message format and convert the input string into bytes for transmission between end points. Another reason to use json as our data format is that it is universal and easy to use for different languages. Our communication is built on top of TCP/IP layer and will read or write from/to the inputstream or outputstream according to our specification of the implementation which is to use "0xd" as message stop indicator.

Translation Layer (KVStore & ClientConnection)

We use KVMessage specified by the handout to translate the json string into Java object by using external library Gson (Google's json - pojo converter). The KVMessage is passed back to the application for message display or to the ClientConnection for database/cache manipulation.

Cache system (KVCache)
We implemented 3 different types of cache strategies: LFU, LRU, FIFO in our storage server.

LFU**:** We used 3 Hashmaps to store KV pairs, counts and a list which can store the keys with same count. The run time of put/get of this cache is O(1).

```
private HashMap<String, String> KVs;
private HashMap<String, Integer> counts;
private HashMap<Integer, LinkedHashSet<String>> list;
```

LRU**:** We created a class which extends LinkedHashMap. We used super class functions to get, put and remove.

FIFO**:** Similar to LRU, we used LinkedHashMap and we used superclass functions of LinkedHashMap to get, put and remove.

Note: LinkedHashMap is a data structure in Java which combines LinkedList and HashMap.

File System (KVDB)
We designed a file system that is divided into blocks. In the header of each block we use the first 10 bytes as block metadata. We firstly hash the key and put this key value pair into corresponding hashed position. If the collision happens, then we put the key value pair right after the next available block position that has not been taken by any key value pair regardless if it is the right hashed position. This happens in a cycling manner which means if the following slots are all taken, then loop back to the start of the file to get the available slot. The first byte is the occupied flag, indicating if the block has been taken a key value pair. The second byte is the grey byte, indicating if the block has previously stored a key value pair and now it has been deleted. The reason behind this is that we design our search mechanism by hashing and iterating from the hashed position, since sometimes a key value may be deleted from the file, but due to hash collision some key value pairs have already been put after it (these key values pairs can not be put into a position where it has been taken by some other key value pair, then they are pushed afterwards), thus even a block's occupied flag is off, we should still look into the grey byte to see if some other key value pairs would follow this one. The bytes 3-7 is used for key size and bytes 8-11 for value size. The next 20 bytes is the key and following 120000 bytes is the value. This forms up our complete database structure.

Testing

The primary test framework we used is JUnit. We have implemented more than 10 additional cases to tests such as input of the command, error handling, different cache strategies and file system. We have successfully passed all the test cases. The detailed case report is in Appendix A Fig.A.1. Besides additional test cases, we also added one test case (MultiClientTest) under Connection Test. Also, we have the (CacheTest) which can help test the correctness of different cache algorithm. Besides, we have a (FileSystemTest), it helps test the basic functionalities(put/get) of our file system with none cache condition.

Performance

The cache strategies we implemented significantly improved the efficiency of put/get actions. We used a total number of 1000 get or put actions and we tested each strategies with different caches sizes from 200, 400, 600. 1000 and different put/get ratio from 0.2 to 0.8 and vice versa. The detailed information can be found in APPENDIX B Fig.B.1 to Fig.B.5. Overall, the performance of each strategy was better than not using cache strategies. The performance also decreased with increasing put/get ratio and decreasing cache size.

APPENDIX A

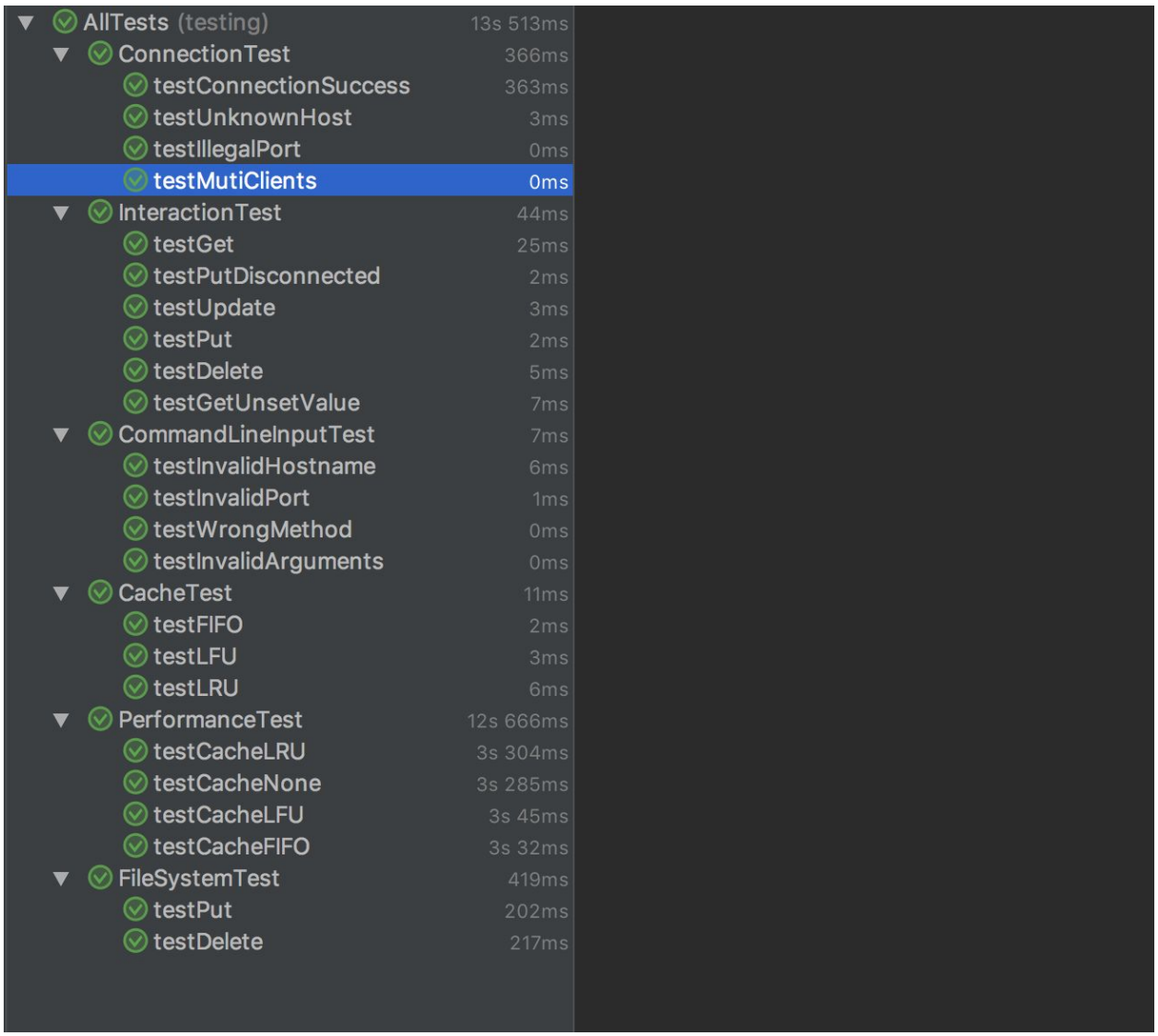

Fig.B.1 Test report and run time

APPENDIX B

For performance and evaluation

```
With Total (put, get) operations of 1000
cache LRU(size200)  put ratio: 0.2 Processing time: 44ms
cache LRU(size200)  put ratio: 0.4 Processing time: 29ms
cache LRU(size200)  put ratio: 0.6 Processing time: 48ms
cache LRU(size200)  put ratio: 0.8 Processing time: 38ms
cache LRU(size400)  put ratio: 0.2 Processing time: 23ms
cache LRU(size400)  put ratio: 0.4 Processing time: 19ms
cache LRU(size400)  put ratio: 0.6 Processing time: 20ms
cache LRU(size400)  put ratio: 0.8 Processing time: 24ms
cache LRU(size600)  put ratio: 0.2 Processing time: 10ms
cache LRU(size600)  put ratio: 0.4 Processing time: 12ms
cache LRU(size600)  put ratio: 0.6 Processing time: 15ms
cache LRU(size600)  put ratio: 0.8 Processing time: 15ms
cache LRU(size800)  put ratio: 0.2 Processing time: 9ms
cache LRU(size800)  put ratio: 0.4 Processing time: 8ms
cache LRU(size800)  put ratio: 0.6 Processing time: 8ms
cache LRU(size800)  put ratio: 0.8 Processing time: 8ms
cache LRU(size1000)  put ratio: 0.2 Processing time: 1ms
cache LRU(size1000)  put ratio: 0.4 Processing time: 3ms
cache LRU(size1000)  put ratio: 0.6 Processing time: 4ms
cache LRU(size1000)  put ratio: 0.8 Processing time: 5ms
```

Fig.B.1 LRU strategy performance[ms]

```
With Total (put, get) operations of 1000
None cache  put ratio: 0.2 Processing time: 24ms
None cache  put ratio: 0.4 Processing time: 26ms
None cache  put ratio: 0.6 Processing time: 24ms
None cache  put ratio: 0.8 Processing time: 23ms
None cache  put ratio: 0.2 Processing time: 22ms
None cache  put ratio: 0.4 Processing time: 24ms
None cache  put ratio: 0.6 Processing time: 24ms
None cache  put ratio: 0.8 Processing time: 27ms
None cache  put ratio: 0.2 Processing time: 25ms
None cache  put ratio: 0.4 Processing time: 31ms
None cache  put ratio: 0.6 Processing time: 26ms
None cache  put ratio: 0.8 Processing time: 25ms
None cache  put ratio: 0.2 Processing time: 24ms
None cache  put ratio: 0.4 Processing time: 21ms
None cache  put ratio: 0.6 Processing time: 25ms
None cache  put ratio: 0.8 Processing time: 28ms
None cache  put ratio: 0.2 Processing time: 22ms
None cache  put ratio: 0.4 Processing time: 22ms
None cache  put ratio: 0.6 Processing time: 26ms
None cache  put ratio: 0.8 Processing time: 29ms
```

Fig.B.2 No cache strategy performance[ms]

```
With Total (put, get) operations of 1000
cache LFU(size200)  put ratio: 0.2 Processing time: 20ms
cache LFU(size200)  put ratio: 0.4 Processing time: 19ms
cache LFU(size200)  put ratio: 0.6 Processing time: 20ms
cache LFU(size200)  put ratio: 0.8 Processing time: 24ms
cache LFU(size400)  put ratio: 0.2 Processing time: 17ms
cache LFU(size400)  put ratio: 0.4 Processing time: 17ms
cache LFU(size400)  put ratio: 0.6 Processing time: 16ms
cache LFU(size400)  put ratio: 0.8 Processing time: 20ms
cache LFU(size600)  put ratio: 0.2 Processing time: 10ms
cache LFU(size600)  put ratio: 0.4 Processing time: 11ms
cache LFU(size600)  put ratio: 0.6 Processing time: 15ms
cache LFU(size600)  put ratio: 0.8 Processing time: 12ms
cache LFU(size800)  put ratio: 0.2 Processing time: 6ms
cache LFU(size800)  put ratio: 0.4 Processing time: 7ms
cache LFU(size800)  put ratio: 0.6 Processing time: 8ms
cache LFU(size800)  put ratio: 0.8 Processing time: 9ms
cache LFU(size1000)  put ratio: 0.2 Processing time: 2ms
cache LFU(size1000)  put ratio: 0.4 Processing time: 3ms
cache LFU(size1000)  put ratio: 0.6 Processing time: 4ms
cache LFU(size1000)  put ratio: 0.8 Processing time: 5ms
```

Fig.B.3 LFU strategy performance[ms]

```
With Total (put, get) operations of 1000
cache FIFO(size200)  put ratio: 0.2 Processing time: 18ms
cache FIFO(size200)  put ratio: 0.4 Processing time: 18ms
cache FIFO(size200)  put ratio: 0.6 Processing time: 20ms
cache FIFO(size200)  put ratio: 0.8 Processing time: 22ms
cache FIFO(size400)  put ratio: 0.2 Processing time: 16ms
cache FIFO(size400)  put ratio: 0.4 Processing time: 15ms
cache FIFO(size400)  put ratio: 0.6 Processing time: 20ms
cache FIFO(size400)  put ratio: 0.8 Processing time: 18ms
cache FIFO(size600)  put ratio: 0.2 Processing time: 9ms
cache FIFO(size600)  put ratio: 0.4 Processing time: 10ms
cache FIFO(size600)  put ratio: 0.6 Processing time: 11ms
cache FIFO(size600)  put ratio: 0.8 Processing time: 12ms
cache FIFO(size800)  put ratio: 0.2 Processing time: 6ms
cache FIFO(size800)  put ratio: 0.4 Processing time: 6ms
cache FIFO(size800)  put ratio: 0.6 Processing time: 8ms
cache FIFO(size800)  put ratio: 0.8 Processing time: 9ms
cache FIFO(size1000)  put ratio: 0.2 Processing time: 2ms
cache FIFO(size1000)  put ratio: 0.4 Processing time: 3ms
cache FIFO(size1000)  put ratio: 0.6 Processing time: 4ms
cache FIFO(size1000)  put ratio: 0.8 Processing time: 7ms
```

Fig.B.4 FIFO strategy performance[ms]