ECE419 M1 Report

Thomas Kingsford
Marcel Mongeon
Zhihao Sheng

28 January 2018

1 Design and Decisions

Architecture Refer to Figure 1 in the Appendices for an architecture diagram.

KVClient The KVClient class handles client side interface. It creats a user interface which contains following operations:

1. connect hostname port: establishing a connection to the host server with the port number.
2. put key value: sending the key-value pair to the server which will store the pair.
3. get key: getting the key-value pair with the key from the server.
4. logLevel: setting the log level. The default level is OFF.
5. disconnect: disconnecting from the current server.
6. quit: quiting the application after disconnecting from the server.

KVStore The KVStore class handles the messages from the KVClient and passes them to the CommMod.

KVServer The KVServer class handles server side interface. It gets the KVMessage from KVClient through the CommMod and does the put or get request, then it will give the response to the KVClient through the CommMode.

CommMod The CommMod class handles client and server communications. The server registers as a listener and receives KVMessages by way of a callback. The server must respond to each received KVMessage with a response message. Clients Connect() then send messages via SendMessage(), which returns a KVMessage response or else times out if the server fails to respond quickly enough.

TLVMessage The TLVMessage is an implementation of the KVMessage interface. it implements a modification of tag-length-value encoding. It (un)marshals a KV message as a sequence of bytes in which:

1. The first byte is the ordinal value of the StatusType enum, referred to as a 'tag'.
2. The second byte is the length of the key $L_K \in [0, 255]$. This protocol imposes an upper limit on key size of 255 bytes.
3. For messages containing a value (the existence of a value is fully determined by the tag), the third byte is the length of the value $L_V \in [0, 255]$. This protocol imposes an upper limit on key size of 255 bytes. This could be trivially extended - for instance, the use of four bytes would give a maximum length of $2^32 - 1 \approx 1$ billion bytes
4. The following L_K bytes are the key.
5. If there is a value, the following L_V bytes are the value.

LRUCache The LRUCache uses a LinkedHashMap implementation which, for every put operation, checks the total cache size and if greater than capacity will evict the entry which was accessed the longest time ago.

LFUCache The LFUCache uses 2 HashMaps in order to store 1) the value to the key and 2) the usage counter. When a put is performed and the cache is becomes large then the enttry with the highest usage counter is removed from the cache.

FIFOCache Similar to the LRUCache, the FIFOCache uses a LinkedHashMap implementation which, for every put operation, checks the total cache size and if greater than capacity will evict the entry which was inserted the longest time ago.

LockManager The lock manager was designed so that locks can be acquired for a particular key, and to provide a timeout specified by the caller in the case a lock is not acquired. For all operations involving put from the KVServer/Cache, getKey(key, timeout) is called from the Cache. The key is then released by calling releaseKey(key), throwing an exception if the calling thread is not the owner. Per key locking is used in order to increase parallelized performance by ensuring that writes/deletes/updates are not blocked when threads are working on different keys.

FilePerKeyKVDB The persistent storage implementation uses a simple storage mechanism by storing all key-value pairs in the same directory, but in seperate files. The file names are the k-v pairs key and the file data is the value for that pair. The use of file-per-key is slower for lookup than storing entire sections of k-v pairs in the same file and representing them as a structure such as a B-tree. The file system stores nodes in the form of arrays, and thus the persistent storage implementation much read each file name when doing a lookup. A major benefit over writing all k-v pairs to file without ordering is that each file is locked seperately through the Lock Manager, thus allowing safe parallelism. Writing 1 file per k-v pair is also advantageous over writing 1 file for all k-v pairs without any strucutre because each k-v pair can be cached without reading all k-v pairs thus alowing for smaller caches than total database size.

2 Performance Evaluation

3 Test Cases

3.1 CacheTests

The cache was tested against : insert, update, delete, get invalid, delete invalid, and checking capactiy. Insert/update/delete were done by performing the operations and checking that the resulting cache either did or did not have the correct resultant entires. Testing invalid get and delete operations was performed on keys that did not exist in the cache or database and so a KeyDoesntExistException was checked to have been thrown. Other methods of writeThrough, eviction, and loadCache were also testing by performing and guaretnteing expected results.

3.2 CommModTests

The CommMod is tested against: message transfer to the server, and appropriate response received from the server.

3.3 ConnectionTest

The ConnectionTest suite tests various socket conditions.

3.4 InteractionTest

The Interactiontest suite tests against: get, put, update, delete, and the handling of error conditions associated.

3.5 KVDBTests

The key value database was tested against critical operations of : insert, update, delete, get/delete. Setup involved creating a temporary storage directory and performing the critical operations. Resultant persistent storage was again queried to ensure the changes had taken place.

3.6 LockManagerTest

Lock manager tests involved getting a lock on a signle object (key) and guarenteeing that no other thread could aquire the same lock until the lock was released, and also testing that the second thread trying to get a lock would timeout when unsuccessful. Another test was checking the seuqence of many threads trying to aquire the same lock as a parent thread, and ensuring that all locks succeeded in order given that their timeout did not expire.

3.7 SocketTest

The SocketTest suite simply tests that basic socket functionality is working. It is used to quickly pick up networking errors when tests are deployed to our Continuous Integration (CI) server.

3.8 StoreServerTests

The StoreServerTests test the interaction between the KVStore and KVServer. Similar to the IntegrationTests suite, it extends the tests provided by the course.

3.9 TLVMessageTest

Tests the correctness of marshalling, unmarshalling, and various error conditions being correctly handled.

4 Appendices

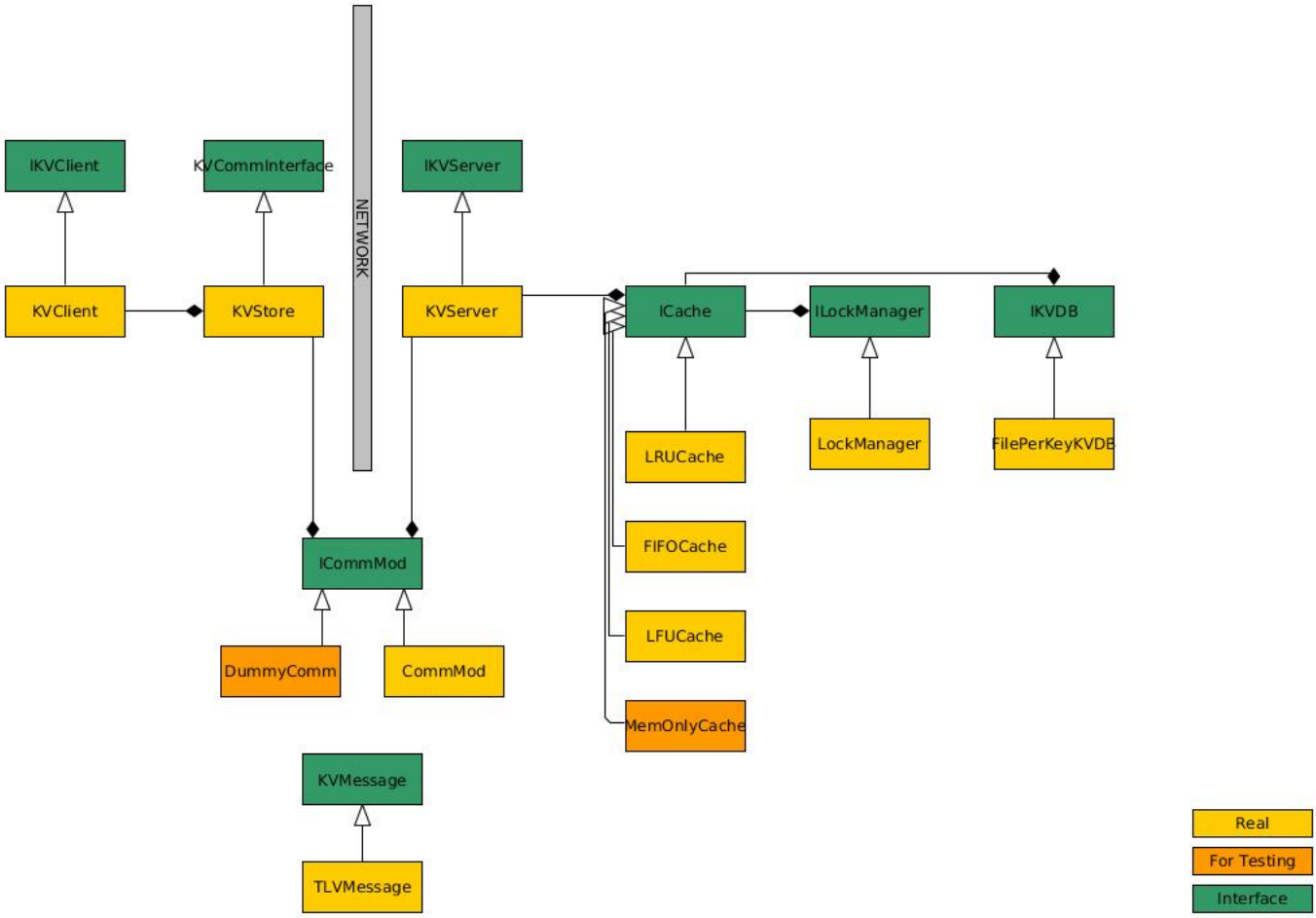

Figure 1: Architecture Diagram