ECE419 Milestone 1 Design Document

Team 24: (Alphabetical Order) Chenhao Liu, Jue Liu, Ze Ming Xiang

1 Overview

We implemented a high-performance, multi-threaded key-value storage server application featuring data persistence and a command-line interface REPL client. The details of our design are documented in the sections below.

1.1 Communication Logic

The client-server communication is based on a custom communication protocol that allows arbitrary message payload and identification of request, which enables the client to identify correspondence of each request-response pair. Each request must be identified by a non-negative 32-bit integer ID. The request format consists of contiguously packed bytes of the ID, the number of bytes in the serialized request message, and the serialized request message itself, in this order. The response format consists of contiguously packed bytes of the corresponding request ID, the status of the response (e.g. OK) as a 32-bit integer, the number of bytes in the serialized response message, and the serialized response message itself. Additionally, the total size of each request/response is limited at 200kb.

The protocol can be alternatively designed based on special termination sequences (such as using '\n' to signal the end of a request); however, the above header-based design is ultimately chosen since it better supports arbitrary messages without the need to escape the termination sequence if it is to be interpreted literally as part of the message payload.

The actual communication between the client and the server is facilitated by the message object class `KVMessage`, and is sent/received using the protocol described above. Each `KVMessage` consists of a status enum denoting the type of the request/response, and optionally the related key and/or value of the message. The size of each key and value is limited to 20 characters and 120k characters respectively.

To serialize each message into a byte array, we used the serialization system provided by Java's standard library. Under the hood, Java's serialization library uses reflection to write a given Java object's metadata (such as its class information) as well as its properties to a byte stream. The bytes can then be deserialized back into a Java object on the receiving end given that it has access to the corresponding class definition. This serialization framework is chosen over alternatives such as Gson and Protobuf, since it is easy to use (requires almost no manual configuration) and is friendly for Java-to-Java environments.

1.2 Server

The server architecture consists of a single main thread and one thread per client connection. Additionally, a storage layer instance (shared between all client connections) is injected at initialization time. The main thread is responsible for accepting socket connections and opening the child threads. Each client connection thread listens on the socket stream, decodes the incoming data according to the protocol, and deserializes the payload into a `KVMessage` object. The corresponding operation will then be executed by interfacing with the storage layer, and the resulting response will be sent back to the client.

1.2.1 Persistent Storage

The persistent storage layer manages storing and retrieving key-value pairs to and from the file system. Since it is shared among all client connections, an instance-level lock is used to synchronize all operations on the storage layer instance. This design is chosen instead of read-write locks or finer-granularity locks for the following reasons: 1. file system I/O are inherently serial, implying that concurrent access of files on disk is no faster than sequential access; it may even be slower since spinning hard drive performs better when data is read sequentially. 2. multiple read operations are also synchronized since caching mechanisms (described in later sections) may alter internal states (such as LRU) even on read access.

The key-value tuples are stored across a limited set of files. The file responsible for a given key is determined by the hex representation of the first n bytes of the key's MD5 hash value, with n set to 1. For example, the key "a" is stored in the file "0c", and the key "b" is stored in the file "92". Within each file, the key-value tuples are stored in the CSV format, where each row corresponds to a tuple in the form <key>,<value>. Literal commas in the key and value are escaped with a preceding backslash. Once the file is determined, a get operation will iterate through all rows and attempt to find the given key; a put operation will also iterate through all rows, and if the key is found, the remaining rows will be read into memory, the new row value will be written to the file, and the remaining original rows will be written back after the row. If a put operation found no existing rows with the given key, a new row will be appended to the file.

This design is chosen over single-file storage format due to its flexibility: since search operation in each file has linear time complexity, having only a single file is slow in practice when the data volume is large; on the other hand, having one file per tuple can result in heavier file-system overhead, more disk fragmentation, and less efficient space utilization. The above design allows balancing of search-efficiency and file system overhead by adjusting the parameter n.

1.2.2 Cache Layer

To reduce file system I/O overhead, a caching layer is used before querying files on disk. The persistent storage layer contains the caching layer, with the cache strategy (e.g. LRU vs FIFO) and cache capacity determined at initialization time using a command line parameter. Each get operation will query the cache first, and each put operation will also update the entry in cache.

When a key-value tuple is deleted, or when a get operation finds no existing tuple, this fact that no value exists for the given key (anymore) is also remembered by inserting a special `String` object into the cache. When retrieving values from the cache, if the result is equal (by object identity instead of the `equals` method) to the special `String` object, the storage layer will respond to the client that the tuple does not exist, and the file system query is skipped.

1.3 Client

The client architecture consists of a single thread running the client-side user-interface application (`KVClient`), with an encapsulated server interface library `KVStore`. The `KVStore` library opens an additional thread for listening to responses from the server (discussed in more details below).

1.3.1 Client User Interface (CLI)

The client-side user interface is a command-line interface (CLI) that bridges the interaction between the users and the server interface library (`KVStore`), allowing users to create connections with the server and send put/get data requests. The user interface is a standard REPL loop which processes the request upon line break, waits for the response, and prints out the response to the user. The user cannot enter another command before the previous response is received; this design is chosen because of its intuitiveness and simplicity for the user. The `put <key> <val>` and `get <key>` commands use space as token separator, therefore space is not supported in keys or values.

1.3.2 Client Library and Socket Watcher

The user-interface application interfaces with a `KVStore` instance, which manages communication with the server. To asynchronously monitor the connection state and get notified when the socket is closed while the UI is waiting for user input, a watcher thread is created upon connection. The watcher thread will continuously listen to socket input, and forward the data to a shared `BlockingQueue`. When sending a request, the `KVStore` service assigns a unique ID to the request, encodes and sends the request using the protocol (see section 1.1), and performs a blocking-wait on the shared queue for the next server response before returning from the function synchronously. Since the user is only allowed to send one request at a time, any response that has a different ID than the current request ID is skipped, since they can only be from previously timed-out requests.

Additionally, when the server connection is closed (such as by network error or server shutdown), the watcher thread will immediately notify all listeners (namely, the client UI) on the `KVStore` that the connection is closed. Appropriate messages will then be sent to the user.

2 Performance Report

The average latency and throughput of our application is profiled for different cache strategies and PUT-to-GET ratios.

Each benchmark repeatedly sends random queries to the server, with the operation (PUT or GET) determined randomly using the PUT-to-GET ratio, the key generated as string representation of a random integer in [0, 999], and the value generated as string representation of a random integer in [0, 10^9). Each benchmark lasts for 10 seconds, with the results from the first 1 second ignored as warm-up. The server's cache size is set to 100.

The benchmarks are executed on a mid-2015 MacBook Pro, with a 2.2 GHz Intel Core i7 processor and 16GB memory.

Cache strategy	FIFO			LRU		
PUT-to-GET ratio	20:80	50:50	80:20	20:80	50:50	80:20
Average latency (milliseconds)	0.2078	0.2591	0.2505	0.1984	0.2242	0.2488
Throughput (queries per second)	4318	3466	3589	4527	4011	3616

Appendix A: Test Report

Tests in AdditionalTest.java (Original tests, not included in starter code)

Test Name	Test Summary	Test Details
testSerialization	Test the functionality of KVMessageSerializer.	Test if the KVMessage can be correctly encoded and decoded.
testLRUCache	Test the functionality of the LRU cache.	Test if the LRU cache can correctly support put and get operations. And test the correctness of the eviction policy when the cache exceeds its capacity.
testFIFOCache	Test the functionality of the FIFO cache.	Test if the FIFO cache can correctly support put and get operations. And test the correctness of the eviction policy when the cache exceeds its capacity.
testCSVStringEscape	Special characters are required to be escaped before storing to disk and be unescaped after reading from disk. This tests the correctness of the escape and unescape function.	Test if the function correctly escapes and unescapes special characters '\n', '\\', ',' and '\r'.
testCSVStringSplit	Test if the string can be correctly splitted by comma.	The edge cases in which the key or value contains commas are also tested.
testKVFileStorage	Test the functionality of the file storage interface.	Test if the key value pairs can be correctly read from the disk. Test if the key value pairs can be correctly written to the disk. Some edge cases are also tested, for example, the keys or values in the tests may contain special characters such as "\n", ',' and "\\".

Tests in ApplicationTest.java (Original tests, not included in starter code)
This is an end-to-end test of the client-server application; it uses a custom input stream to simulate command-line user input and captures the standard output for testing.

Test Name	Test Summary	Test Details
testConnect	Connect Client and Server through UI	Send the “connect” request through the UI, test if the shell output is the same as expected.
testPut	Test the PUT command of new tuple	Send the “put <key> <value>” request through the UI, test if the shell output is the same as expected.
testGet	Test the GET command	Send the “get <key>” request through the UI (where the key exists), test if the shell output is the same as expected.
testGetError	Test the GET command where the key does not exist	Send the “get <key>” request through the UI (where the key does *not* exist), test if the shell output is the same as expected.
testUpdate	Test the PUT command where key exists	Send two “put” requests with the same key and different values though UI, test if the shell output is the same as expected.
testDelete	Test the PUT command where key is null (deletion)	Send a “put” request with value null and existing key though UI, test if the shell output is the same as expected.
testDeleteError	Test the PUT command where key is null (deletion) and key does not exist	Send a “put” request with value null and a key that does not exist though UI, test if the shell output is the same as expected.
testDisconnect	Disconnect the server and client through UI	Send the “disconnect” request through the UI, test if the shell output is the same as expected.

Tests in ConnectionTest.java

Test Name	Test Summary	Test Details
testConnectionSuccess	Test if `KVStore` can connect to server	Call the “connect” method on the `KVStore` and verify that no error is thrown.
testUnknownHost	Test when host name is invalid	Call the “connect” method on the `KVStore` where host name is invalid, and verify that `UnknownHostException` is thrown.
testIllegalPort	Test when port is invalid	Call the “connect” method on the `KVStore` where port is invalid, and verify that `IllegalArgumentException` is thrown.

Tests in InteractionTest.java

Test Name	Test Summary	Test Details
testPut	Test adding new data to `KVStore`	Test if the key-value pair can be put and return PUT_SUCCESS on success
testPutDisconnected	Test adding new data to `KVStore` when disconnected	Send a put request when it is disconnected with the server, test if it returns an error.
testUpdate	Test updating existing tuple	Send two put request with the same key, the value should be updated return UPDATE_SUCCESS on success
testDelete	Test deleting existing tuple	Send put request with null as the value, the server would return DELETE_SUCCESS on success
testIllegalDelete	Test deleting non-existent tuple	Send put request with value null and non-existent key, the server would return DELETE_ERROR
testGet	Test retrieval of tuple	Send get request after putting the key-value pair into the server, the server should return

		the value of the key
testGetUnsetValue	Test retrieval of non-existent tuple	Send a request with a non-existent key, GET_ERROR should be returned.
testGetDisconnect	Test retrieval of tuple when disconnected	Send a get request when disconnected with the server, an error should be thrown.
testPutDeleteGet	Test put, delete, get request together	Put the data into the server then delete it. Send the get request with the key, GET_ERROR should be returned.