Danial Hasan, Graham Hoyes, Daniil Orekhov

ECE419 M1 Design Document

Architecture

Interfaces to the KV store on the server side and client side (via the server) are packaged in the `store` and `client` modules, respectively. Both provide the same interfaces for `put` and `get`, and the `store` module also contains interfaces for checking for key existence and deletion.

The TCP server is in the `app_kvServer` package. It runs as a single thread, looping for new TCP connections. Each new connection is handled by the `ClientConnection` class, which is responsible for receiving messages, dispatching to the KV Store, and sending responses. A single `KVStore` instance is instantiated by and stored in the main server thread. To dispatch actions, the `ClientConnection` class interacts with methods on the server thread which themselves dispatch to the `KVStore` class. Multiple reader / single writer mutual exclusion is handled within these server methods with a `ReentrantReadWriteLock`.

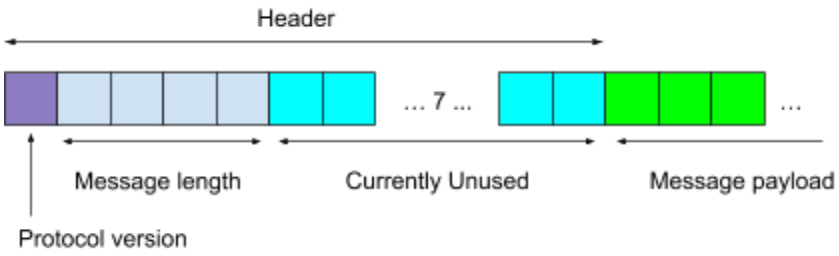
The CLI client is in the `app_kvClient` package. It contains the logic for parsing command line inputs, establishing a connection and communicating with the server via the `KVStoreConnection` class within the `client` module. `KVStoreConnection` extends the same abstract base class as `ClientConnection` used by the server, hence they share the same serialization and communication code. The client processes results or errors returned by the server and displays them to the user.

Communication Protocol

Communication between the client and server is done over TCP sockets. Messages packets begin with a 16 byte header. The first byte of the header contains the communication protocol version, fixed to 1 in this iteration. Upon receiving the message, the client or server first checks the protocol version to ensure it matches the version it expects to be receiving. If it does not, a warning of potential unexpected behaviour is raised, but the communication attempt is not aborted. We thus expect that future protocol versions should remain backwards compatible.

The next 4 bytes of the header contain the length of the message payload, in little endian order. In the communication code, this value is technically treated as a signed integer, so message length is limited to 2^{31}-1 bytes (which, given the limitation in the Milestone 1 specification of maximum 20 byte keys and 120 kilobyte values, does not pose an issue). The remaining 11 bytes of the header are currently unused, and are reserved for future use.

If the entire header cannot be read successfully, the communication attempt is aborted from the receiver side. If it is read successfully, the message is read based on the message length header tag. This structure is shown visually below.



Message payloads are UTF-8 encoded JSON strings of the desired action. Using JSON strings and a message length header tag makes it simple to add future fields to the message payload without any changes to the network communication logic or serializing/deserializing logic.

Each status type has an expected JSON schema. All require a `status` enum field, and a combination of `key`, `value`, or `message` string fields. The presence of any additional fields beyond those required for each status type will have no effect. For the interested reader (and our own future reference), the full table of schemas is included in Appendix B.

Persistent Storage

Key-value pairs are persisted on disk for permanent storage. The `KVStore` interface is implemented by the `KVSimpleStore` class. This class persists data to a file where key-value pairs are encoded as UTF-8 encoded JSON strings, formatted as key: `{"<key>", value: "<value>"}`. Each pair's JSON string is written to a single line within the file, and each line is separated with a newline character. The `KVStore` interface is expanded to include a method for directly deleting a value, similar to a DELETE request in HTTP, for ease of use. Additionally, the `void put(String key, String value)` is modified to return a boolean that indicates if the value was updated or inserted as a new key-value pair.

All `KVSimpleStore` methods that query the database (`get`, `put`, `delete`, and `exists`) call a method, `boolean find(String key)`, that does an O(n) lookup to find the designated key in the database. If the key exists in the database, the `find` method stores the location of the key-value pair and the value itself into class variables and returns `true`. If the key does not exist, `find` returns `false`.

Calls to `get` will simply call `find`, and return the value of the key-value pair if the key exists, otherwise throwing an `KeyInvalidException`. Similarly, calls to `exists` will call `find` and return the output of `find` directly.

Insertion or updating of key-value pairs takes place through the `put` method. A call to `find` determines if the key-value pair exists and its file pointer within the file. If the key does not exist in the database, the `KVSimpleStore` class appends the encoded key-value pair to the end of the file. If the key exists, hence an update to the key-value pair, the `KVSimpleStore` class will

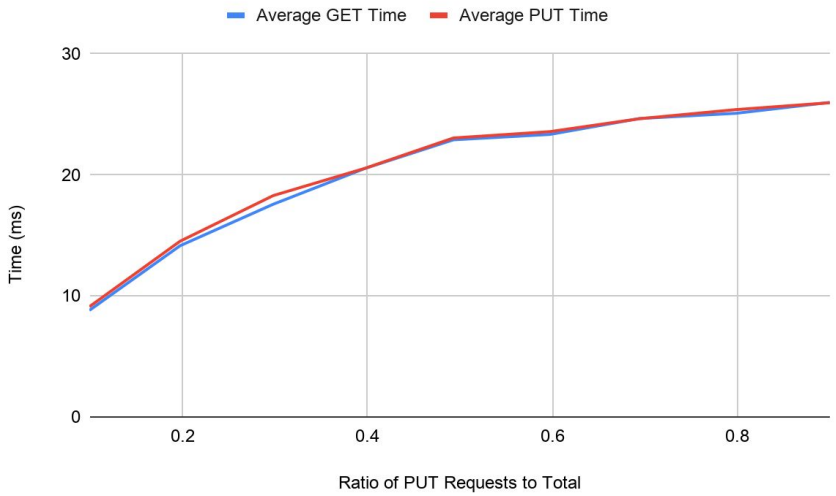
open a temporary file and copy the data proceeding the key-value in question from the storage file to the temporary file. The key-value pair in question is updated and the proceeding key-value pairs are then copied back to the storage file. The copying uses Java's `FileChannel.transferFrom()` and `FileChannel.transferTo()` methods for direct data transfer without the use of an intermediate buffer for potentially large speedups in copying performance.

Similarly, deletion of key-value pairs, which use the `delete` method, will call `find` to determine if the key-value pair exists. If the key exists, the data proceeding the key-value pair to be deleted is copied from the storage file to the temporary file. The storage file is truncated to the position preceding the key-value pair to be deleted and the remaining data is copied back from the temporary file to the storage file.

Calls to `clear` will simply truncate the storage file thereby clearing its contents.

Performance Evaluation

To evaluate performance, we perform 10k total GET and PUT operations on the server, varying the proportion of each. There are 2028 possible keys, ensuring that some keys are repeated (getting values that have been put, or updating previously inserted values). GET and PUT requests are interleaved with each other throughout the performance evaluation, so as time progresses the database saturates with values. No deletion is performed, so these numbers represent a worst case scenario for PUT requests. Beyond 50% PUT operations, the number of updates exceeds the number of inserts.



The graph of average operation time to ratio of put requests to total from 10% to 90% is shown above. As the fraction of put requests increases, the number of items in the database approaches the number of unique keys. Put and get are both linear in the number of keys in the database, hence we see average response time plateau as the number of keys in the database stops increasing.

Appendix

Appendix A: Unit Tests

Unit tests were written in JUnit 4 style (the provided test cases were modified to conform to this when required).

Provided tests (unmodified, other than formatting):

- Connection tests
    - Valid connection
    - Unknown host
    - Invalid port
- KVStore Client Interaction tests
    - Successful PUT insert
    - Successful PUT update
    - PUT after a disconnect
    - Successful delete
    - Successful GET
    - GET with unset key

Additional tests (18 total):

- JSON Message tests: Test the initialization, serialization, and deserialization of JSON messages
    - Initialization with only a status type
    - Deserialization from valid json
    - Throws DeserializationException for deserialization from invalid json
    - Initialization from a json string
    - Sets missing optional fields to null
    - Raises DeserializationException for missing status type
- Communication protocol tests: Test the byte communication protocol used by the TCP connections (JSON serialization tested separately, above)
    - Test message sending: Test that protocol version, message length, header unused space, and message payload bytestream are properly constructed
    - Test message receiving: Test that a message can be properly received from a valid communication bytestream
    - Test that IOException is thrown when input connection is closed
    - Test that IOException is thrown when message payload is shorter than indicated in header tag
- KVStore tests: Testing the persistent storage class for permanent storage of key-values
    - Test invalid keys for put/get/delete: Test that the storage throws an exception for keys that are invalid i.e. the key-value pair does not exist
    - Test put/get: Test that a put request for a new key-value pair is correctly stored and that a subsequent get request returns the correct value

- Test put update: Test that a put request correctly updates an existing key-value pair
- Test delete: Test that key-value pairs are deleted as intended
- Test storage file deletion: Test that a correct exception is thrown if the storage file is moved/deleted
- Test invalid data format: Test that an exception is thrown for invalid data formatting within the key-value persistent storage file

Appendix B: Message JSON Schemas

Status Type	**JSON Schema**
GET	{status: "GET", key: "<key>"}
GET_SUCCESS	{status: "GET_SUCCESS, key: "<key>", value: "<value>"}
GET_ERROR	{status: "GET_ERROR", key: "<key>", message: "<message>"}
PUT	{status: "PUT", key: "<key>", value: "<value>"}
PUT_SUCCESS	{status: "PUT_SUCCESS", key: "<key>", value: "<value>"}
PUT_UPDATE	{status: "PUT_ERROR", key: "<key>", value: "<value>"}
PUT_ERROR	{status: "PUT_ERROR", key: "<key>", value: "<value>", message: "<message>"}
DELETE_SUCCESS	{status: "DELETE_SUCCESS", key: "<key>"}
DELETE_ERROR	{status: "DELETE_ERROR", key: "<key>", message: "<message>"}
BAD_REQUEST	{status: "BAD_REQUEST", message: "<message>"}
SERVER_ERROR	{status: "SERVER_ERROR", message: "<message>"}