Protocol Design

A. Request/Response Payload

During the initial planning of the protocol, I knew that I needed:
(a) a way to differentiate between commands
(b) a way to transport the key to and from the server/client
(c) a way to transport data to and from the server/client

Initially, these “packages” were sent as three separate lines, using readLine to determine the delimiter with “\n”. This worked for the basic put(key, data) as well as get(key), however once designing the putter and getter for files, it stopped working because the files themselves included “\n”.

Understanding that the information needed to be sent as one entire package, I was led to use a serializable object as it could be transformed into a byte stream, and transformed back into an object without any distortion. Essentially, I would be sending one payload of metadata rather than multiple sends of the data separately.

Eventually, the serializable object **RUDataPayload** consisted of:

(a) the ***command*** as a String, required by every request/response
The commands themselves would be used by the server to determine which helper function to call. It would also alert the client that a response payload was being received (not necessary for the project however, assists in scalability).

(b) the ***key*** as a String, required by putters, getters, and removers

(c) the ***success*** flag as an int to denote if a command was performed successfully
A successful response would return a 0, failure was a 1, and any command that didn’t require a success flag would receive a -1 (or my equivalent of a null)

(d) the ***dataLength*** flag to denote the size of the byte array that was being sent
Initially this was to avoid having to initialize a byte array to a large size in order to accommodate all lengths of data. Instead, I could use this dataLength to initialize a byte array to the exact length needed, saving memory. If no data was being sent in the command, then this was set to the default of 0.

However, upon implementation, the dataLength wasn’t actually required, because using the serializable payload along with Object I/O Streams, I could grab the data entirely without reading from a stream and having to prepare a pre-sized array. I decided to keep the dataLength flag however just in case.

(e) the ***data*** itself as a byte array
If data transfer wasn’t required, the byte array was set to null.

B. Payload Transfer

The design of the **RUDataPayload** being serializable meant that rather than having to read through a stream byte by byte (tongue twister), the object could be automatically transformed back into a comprehensible Object with all the data intact using **ObjectInputStream** and **ObjectOutputStream**. This saved a lot of time in the long run.

C. Data Storage

The data structure of the stored information obviously had to be done using a Map of some sort. I decided on using a HashMap type as it was what I was most familiar with. Upon further implementation, I realized that in a real-world scenario, a server could have multiple clients connecting to it at the same time. As such, the HashMap (our database) had to be thread safe so I ended up using a ConcurrentHashMap.

I also synchronized the datastore object itself and required any access to grab the same instance of **RUDataStore** so all clients would be sharing the same data.

The ConcurrentHashMap itself stores a byte[] value to a String key.

D. Server Design

The basic server setup was based off of the recitation example v5 where the server was a Runnable and each instance of a connection opened was a separate thread. This supports both multi-client connection, as well as saving me time from having to design the base server from scratch.

Each socket connection had its own global object streams to avoid messing up the serialized data being received and sent. The main “driver” functionality of the server did 4 important steps:
(1) It read the payload object as a byte stream from the client and pieced it back into a RUDataPayload object automatically due to the serialization.
(2) It grabs the 5 information components of the payload
(3) It determines which helper function to call based on the command component
(4) It writes back a payload as a response to the client

Using one universal payload design helped tremendously by standardizing the helper method returns and how commands were sent/received. The helper functions themselves were more or less methods that created a new payload response using the information provided by the request.

E. Client Design

Similarly to the helper functions in the server, the API calls in the client merely build and send a RUDataPayload object and write it to the server. It then reads a payload object as a response and handles it accordingly.

Project Experience

I think my implementation could be better in terms of the server itself. Like I mentioned above, the main code for the server was copied from the recitation so understandably, it's a very basic implementation.

Some improvements if I wanted to create an actual transfer protocol would be to include security handshakes for authenticating client connections, as well as using an actual database to store the information rather than storing in local memory.

Due to my focus on handling multiple client connection (which arose because I didn't fully understand how the backlog and multiple threads of the server worked – since I copied that from recitation v5) I think an improvement would be to also set up some sort of queue system where users can spam a multitude of asynchronous commands and the server would process them in order. Future improvements could also include allowing the server to notify clients when updates to the database were performed.

The scalability of this current implementation is reasonably high as long as the payload design satisfies the required types of transactions. New commands can be added simply by adding a switch case and a command constant. The overall format of the helper function would remain the same, revolving around the creation of a payload response.

The performance of my design should be decent as HashMap has a put and get time complexity of O(1). The only part where my design suffers is in the list command as it requires a O(n) while loop to transform a Set of keys into a single string for byte[] conversion. Ways around this could be adding a String[] field to my payload design (however at this point, I'm too exhausted to change it).

Overall, I think I did a very good job on this project. It should be by far the hardest one since future implementations of server/clients can be easily designed off of the same structure from this project.