Andrew Park (ap1614)
CS417: Distributed Systems
Spring 2021

Project 1 Report: RUStore

In this project, I designed the project using basic Java sockets and with OutputStream and InputStreams as the basis of our communication. The project can be separated into 2 parts, the client and the server. For the server, I used a HashTable in order to store all data in a key-value pair, with keys being strings and values being byte arrays. I would accept clients using a ServerSocket and would wait for clients to connect. Using ServerSocket, I am able to accept clients and using the socket associated with that client, create a thread which would handle all communication between the server and the client. This actually allows the server to handle multiple clients at once, though the actual ability for the server to do so is very limited. The thread that handles the client runs on a while-true, which first reads from the socket. The first read is interpreted as a command for the server, which the server then handles using a switch case statement for the different commands that it supports. Generally, commands and messages are formatted in the following way: [command]:[arg1]:[arg2]
The implementation for each of the commands are as follows:

1. **put:** For the put command, the first message is structured as follows:
   put:[key]:[data.length]
   The message is first colon separated , the first being the put command itself. Then the 2 following parts of the message are interpreted as the key that we want to store and the length of the data we want to store. If the server already has the key, we send a simple **serverHasKey** message to the client. Otherwise, we send the message **putContinue** to denote to the client that it may proceed with sending the data. We store this in a **byte[]** buffer and then store it in the HashTable.
2. **get:** For the get command, the first message from the client is structured as follows:
   get:[key]
   The first portion is the get command, with the second portion of the message being the key which the client would like to retrieve. We check if this key is in our HashTable and send a **serverNoKey** message to the client in the event the key is not in our server. If the key does exist, we send **serverHasKey:[length]** where length is the size of the data that we are about to transmit. The client then acknowledges by sending a **getContinue** message, which at this point we will send the data to the client
3. **list**: For the list command, the message is always structured as
   list
   We simply use **keySet()** and convert the given set into an array using **toArray**, which we then convert to a string and send to the client.
4. **remove:** For the remove command, the message is structured as
   remove:[key]
   The first portion is our remove command, with the second being the key we wish to remove. We send a **serverNoKey** or a **removeSuccess** message to the client when the

server does not have the given key or when the server has removed the key, respectively

5. **disconnect:** The disconnect message is structured as just:
   disconnect
   When the server receives this, it will first send to the client **serverDisconnect** letting the client know that it is going to disconnect. It then closes the socket and stops the client handling thread.

When we are transmitting larger amounts of data(say a file or data larger than 8192 bytes), we must treat the read operations slightly differently. By default, the server uses size 8192 buffers, but in the event that we are transmitting larger amounts of data, it will create a buffer using the length that is specified in the commands. Because InputStream reads are not completely blocking, I created a wrapper function called **readSocket**, which would read into the buffer the full length, and is completely blocking. This allows us to transmit large files in the event that the buffer is not large enough or the InputStream read does not read the full length in one invocation. This occurs both on the server side and the client side. The specific reason we use a HashTable on the server is because of the key-value pair, which HashTables are specifically designed for. However, a HashTable is preferred over a HashMap due to its synchronization, which is important since the server is able to handle multiple clients simultaneously.

There are a couple of ways my implementation can be improved. One is that the actual multi client system with threads has certain race conditions that may arise when multiple clients are connected to the server at once. This may occur when a client sends a get while another client sends a remove. Depending on the order that the messages are sent, the client get try to get a key that exists at the time, and then the server may receive a remove key message. The first client would have received a **serverHasKey** but end up not being able to receive the data at all. The proper solution for this would be to use some locking mechanism like semaphores or mutex locks, or to use Java's built in synchronized blocks, though my experience with these are limited which is why I did not pursue this option. I opted instead to use HashTables which are inherently synchronized to try and limit these race conditions as much as possible. Another flaw in my design is the way commands are actually handled. Since the server uses a switch case statement, it is harder to scale our server. Introducing more features into the server requires more work than if we were to use separate functions for each of the different commands instead of using switch case statements. There are some smaller, even more trivial changes to my implementation that I feel would improve the program but are very unnecessary(though I will list some of them at the bottom of this report as a postface).

Specifically for this project, the hardest parts to implement were the reads using InputStream. InputStream.read() is not blocking, which brings out an issue where we are not always sure that all the data that is needed has been written to the buffer. In order to overcome this, I created a wrapper function **readSocket** which would read from the socket until the required amount of data has been successfully read. We can then treat this as a blocking-io call. Outside of this, the other problem was how to actually structure the communication protocols, though a lot of how I write it was inspired by a previous project I had worked on which is very similar to this project, Where's That File from CS416: Systems Programming. If I were to change how communication was handled though, I would include some more reliability and more

acknowledgements in order to ensure that the messages or data sent between server and client is actually processed properly and nothing is lost. While there are some message acknowledgements, there is no actual check to make sure the message received or the data that was sent is not malformed. Including checksums or verification would probably help to make sure that messages are not lost or corrupted.

Postface: My research interests are mostly in computer and systems security, so my main concern with 2 way communication has to do with the security of the communication. In my implementation, there is no way of ensuring that our messages or data hasn't been tampered with. Since our communication is built on top of TCP without any encryption, it is easy for a third party to eavesdrop or launch a man-in-the-middle attack. We can use encryption or transport layer security to try and make our communication secure(RSA, Diffie-Hellman, TLS, etc.).
I also found that sometimes sending and receiving files/data might be very slow due to them being a couple megabytes in size. Using zlib or some similar compression libraries could be useful to make communication faster at the expense of some more computation needed for decompression. These improvements aren't really necessary which is why I included them instead in this nice postface.