William Clift
Operating Systems
25 March 2020

**Programming Assignment 1 - Quotes Client/Server**

*Concepts*

In this assignment, we were tasked with creating a `quotes.txt` file filled with quotes by different authors and sectioned into different categories. Throughout compeleting the assignment, I was faced with decisions like how to send messages to and from the client and server in a way that they would both understand, and whether to read the file in on the startup of the server or at runtime of the client. The challenges in this assignment included handling the datastructures that stored the quotes while processing, and ensuring that messages were properly sent from client to server and server to client with indication when the message is over.

*Implementation*

For my implementation, I started with that given in the book in Figures 3.27, 3.28 and expanded them to the specific requirements. It was clear that the extra-credit portion by including the option for categories drastically increased the complexity of both the file processing and the communication between the client and the server, so many of my first attempts to do this efficiently were changed.

Early on, I made the decision to have the category of each quote as the first character of the line in the `quotes.txt` file, and have the server ignore this character when printing it out. On startup of the server, I have it read the file and store each quote in an ArrayList called `quotesFromFile`. If a certain number was requested by the client of no specific type, it would put all of the quotes in a new ArrayList called `quotes` and then randomly select a removal of the specified number of quotes or until it got to the end of the list. This way, we would not print out repeats. If a category was selected, we would note which category it is by flagging it with the corresponding character key. On startup, an ArrayList was created made up of the character key for each type followed by an integer indicating which index of `quotesFromFile` started each new category. That way, we can iterate through that category of quotes starting directly at the key index. These quotes were again moved to a new ArrayList where they were removed randomly until reached the desired number, or at the end of the list. For this reason, I also needed to ensure that all quotes of that type were put together in the file.

In order to test efficiently, I made the choice to have the server send a signal indicating the end of the quotes requested that closes the connection from server to client. There is also an option to close the server from the client side by running `java QuoteClient x`.

*Analysis*

**What are the advantages and disadvantages of each of the ways of accessing the quote file?**

For a relatively small file or assuming that there would potentially be a time differential between when the server starts running and when a client forms a connection makes processing the file on startup of the server attractive. This not only allows errors in finding the file to be caught before a client connects, but by my design, it transfers all data in need of processing to member data of the class. The downside is that it takes up a lot of space in the server and this might not be optimal.

However, given a very large file, the server could get stuck processing the file and even run out of space transfering the file data to member data. If the file is very large, it would be much better to process the data in the file once the request comes in from the client and select quotes by reading lines of the file. This limits the ability for all quotes to be chosen with equal probability because you are potentially selecting quotes before knowing how many there are of any given type, but it would make it faster for the server, and save a lot of space.

Another potential solution would be to change the design by writing the quotes back to the client as they are selected, instead of storing them in an Array and then transfering them one by one. This would save a sizable amount of space for the server.

**Give a thorough discussion in terms of runtime, space, and operating system resources.**

For the startup of the server, the runtime to process the quotes before it is ready to serve clients is directly proportional to the number of quotes in the file, as is the space. The quotes being stored in an ArrayList makes the overhead for this data structure greater, but it increases the efficiency of the algorithm to process them because it does not need to first determine the length of the file.

The runtime of the `getQuotes()` functions are determined in the case of a number of random quotes by the size of each category. There could be a category of quotes that is very large, but by storing the character key that indicates the index beginning each category, we only process the quotes of consideration in this function. So, if we select a smaller category relative to the others, those other sizes will not influence our ability to select the category of quotes that we want.

Communicating with any individual client will keep it busy until the quotes for that client are finished being processed. This is why we made the decision to end the connection once the quote are transferred to the client. Essentially, by design, the server can only server one client at a time. If it were multi-threaded, we would be able to serve multiple clients, but the speedup would only be seen if a large number of clients were coming in at the same time, and if each only requested a relatively short list of quotes because the CPU would be able to manage many short processes better than many large. But even large processes that exceed the number of cores on the machine would slow down the server considerably.

**Given your discussion which is the best one to implement in the context of a small OS class environment?**

For this class environment, we will likely have a small number of clients and relatively small processes each, so I stand by my choice to start the processing of the file at startup of the server and randomly select them by index in a data structure. With these assumptions, we should not have to worry about either the server on startup, or any individual process taking up too many resources. But, if we are not prepared for a large request from one client, we could get stuck. For this reason we should still have a way to manage the clients based on the size of their requests so that no one in particular can hog the CPU.

**Which would be better in a large internet environment?**

If we had this server on the internet, we would potentially have a huge amount of quotes in the file with many clients coming in at once. This calls for processing at runtime to select quotes from the file. We would spend less space on the quotes being stored in any sort of data structure, and simply transfer a selected quote to the client as we go along. If we could assume that although the number of clients is large, that the size of each process is relatively small, then we might want to store each "package" of quotes that we receive to give to each client in a datastructure before sending it back, to do it all at once. However, if we are multi-threaded, then we might run out of allocated space with many clients asking for small amounts of space all at once. We would also need to ensure that the space for each client's process is properly managed so that we can avoid a larger process getting stuck or any one process running for too long.