CS 4390.502 Computer Networks

Math Server Project Report

Team Members

Akaanksh Raj Kambalimath – AXK180122

Nicolas Chevrie – NJC180030

Assumptions

- A basic mathematical operation is a calculation with two real numbers separated by an operator sign, with the exception of factorial which is one number followed by the factorial sign. The input from the user must have no whitespace, and the numbers entered must be parsable from String to Double. Decimal places are allowed. If the input is too large to be sent by the client, then the request will not be sent.

  The server supports the following operations (with corresponding operator signs): multiplication (*), division (/), addition (+), subtraction (-), exponentiation (^), modulo (%), and factorial (!). The result will be returned as a Double, and must be within the range of a Double to be reported accurately.

  - For example, an operation of 4 * 4 should be entered as “4*4” or “4.0*4.0”. The server would then reply back with “16.0”.

- ‘Multiple clients’ means that the server can serve more than one concurrent client, but the server can potentially handle an unlimited number of clients, bounded by the computational power of the server machine.
- The FIFO policy for servicing client requests applies only after the client’s entire request is read by the server. Partially read requests do not count as first in.
- Assuming that the server will be at IP 127.0.0.1 (local) and port # 6789. This can be changed by editing the value of SERVER_IP and PORT_NUMBER in /client/main.java and PORT_NUMBER in /server/main.java.

Protocol Design

To send messages, the server and client call the marshal method from the Protocol class, which accepts a Map<String, String> (which holds the parameters for the message) and returns a String which the client or server sends over the network.

To receive messages, the server and client read a newline-terminated String from the socket and call the unmarshal method from the Protocol class, which accepts a String and returns a Map<String, String>.

The marshal method converts the Map<String, String> into a String by URL encoding the key and value with the UTF-8 charset, separating the key and its value with an “=”, then separating key-value pairs with an “&”. Lastly, it ends the String with a newline so it knows when to stop reading on the socket. The unmarshal method simply works in reverse to convert a String back into Map<String, String>.

Message Format:

The TCPClient and TCPServer classes have message builder functions that take in parameters which are put in a Map<String, String> and returned as a marshalled String that the client or server can send over the network.

The client's messaging format:

- Only "cmd", "name", and "eq" are valid String keys.
- The String key "cmd" is mandatory. The String key "name" is included if cmd's value is "hello" or "exit". The String key "eq" is included if cmd's value is "math".
- For "cmd", the valid values are "hello", "math", and "exit."
- For "name", any valid UTF-8 string is accepted.
- For "eq", any valid UTF-8 string is accepted, this is validated server-side.
- The client's raw input is required to be less than 1024 bytes in UTF-8. This is validated client-side and this restriction is in place just to ensure the server's buffer of size 2048 does not overflow.

The server's message format:

- "resp" is the only valid String key.
- For the client command "hello", the value for "resp" will be "Hello, <name>". The response is validated client-side and is considered an ACK if it validates correctly.
    - This command establishes the connection between the client and server.
- For the client command "exit", the value for "resp" will be "Bye, <name>". The response is validated client-side and is considered an ACK if it validates correctly.
    - This command terminates the connection between the client and server.

- For client command “math”, the value for “resp” will be either the answer to the equation from the client or an error message.

Logging Format:

Logs are generated using the java.util.logging tool, which will produce a text file in the base folder named “TCPServer.log”. The logs are stored in XML format, but the information can be read from a text editor. Each logging event will store the time of the event in the entry. The log is persistent between executions of the TCPServer file, and new entries will be added to the same log file.

The following events are logged with timestamps:

- Client joins - Logs client’s name, IP address
- Client disconnects - Logs client name and total time connected
- Client command received - Logs client, the command, and the response to the command
- Server startup
- Server shutdown - For each client, logs the total time connected
- Errors - Logs certain types of error and the relevant exception

Programming Environment

Java SE Development Kit 8u211

Java SE Runtime Environment 8u251

Tested in Windows 10 and Kali Linux 5.10.0-amd64

Compilation & Execution

There are no parameters required to run, but the client application will prompt the user for a name before attempting to establish a connection.

For Windows:

- cd to the base folder of the project in Powershell
- Server
    - To build the server, run: javac ./server/Main.java
    - To run the server, run: java server.Main
- Client
    - To build the client, run: javac ./client/Main.java
    - To run the client, run: java client.Main
- To build Javadocs (outputted to as index.html in ./docs/javadoc):
    - javadoc -private -splitindex -d ./docs/javadoc ./server/TCPServer.java ./server/Main.java ./client/TCPClient.java ./client/Main.java ./lib/Protocol.java

For Linux:

- cd to the base folder of the project
- The Makefile contains targets, so use the following:

- Server
    - To build the server, run: make build-server
    - To run the server, run: make start-server
- Client
    - To build the client, run: make build-client
    - To run the client, run: make start-client
- To build Javadocs (outputted to as index.html in ./docs/javadoc), run:
    - make docs
- To remove the .class files, run make clean

Challenges Faced

We initially approached the design of the server using a multi-threaded approach, with a new thread being created for each client. The client's thread would've received messages, queued them in a global queue shared between all threads, and responded with the answer once it was available. In addition, there would've been a separate evaluator thread to evaluate the clients' messages from the global queue in FIFO order and signaling the corresponding thread after evaluation was complete. However, we realized this approach would not have guaranteed FIFO because the CPU could've switched the order of thread execution at any time, making it possible for messages to be not sent back in the order they were evaluated. We landed on the solution of using a single thread with NIO selectors because this guarantees that when a message has arrived in its entirety, the server will respond right away without the possibility of having the CPU switching threads and handling a different client's command. As such, this implementation

guarantees FIFO execution of client requests. If one client were to enter an expensive operation, this could potentially force other clients to wait, but this is not a serious issue for this implementation since most of the calculations are very short and FIFO execution is prioritized.

For the first interaction of the equation evaluator, we had used floats for storing the arguments of the mathematical calculations. While this worked fine for some operations, it led to the occasional floating-point error, which was unideal for a calculator application. We rewrote the evaluator to use Doubles for storing arguments and evaluating the equations, which improved the accuracy of the system.

Takeaways

Along the way, we needed to research the tools in Java that would allow us to create our applications. We learned about how the Java logging tool automatically formats the logging output and how to manipulate the priority level of each message. To implement a single-threaded server, we had to learn about non-blocking IO and how they could be used in Java. To implement the protocol, we learned about URL encoding and decoding and character sets. We also had good success with how we divided up the work, and learned that we were most effective when we met up at regular deadlines to compare our work and check in on each other during the process.

Screenshots

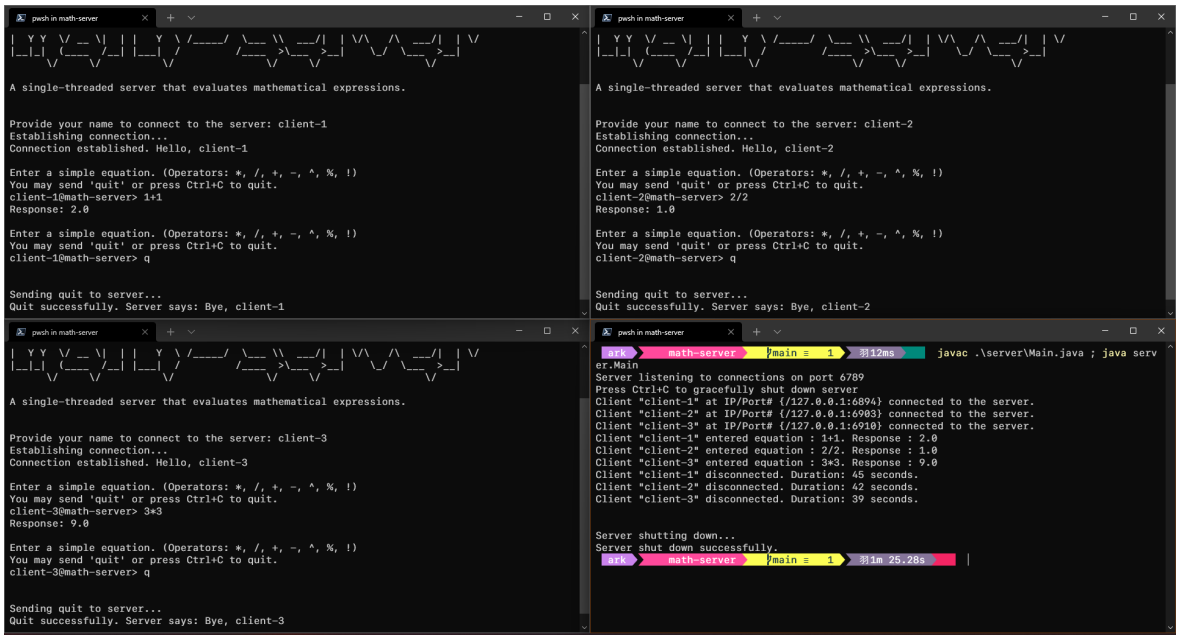

Image 1: Execution of the client and server applications using the Windows 10 command line.

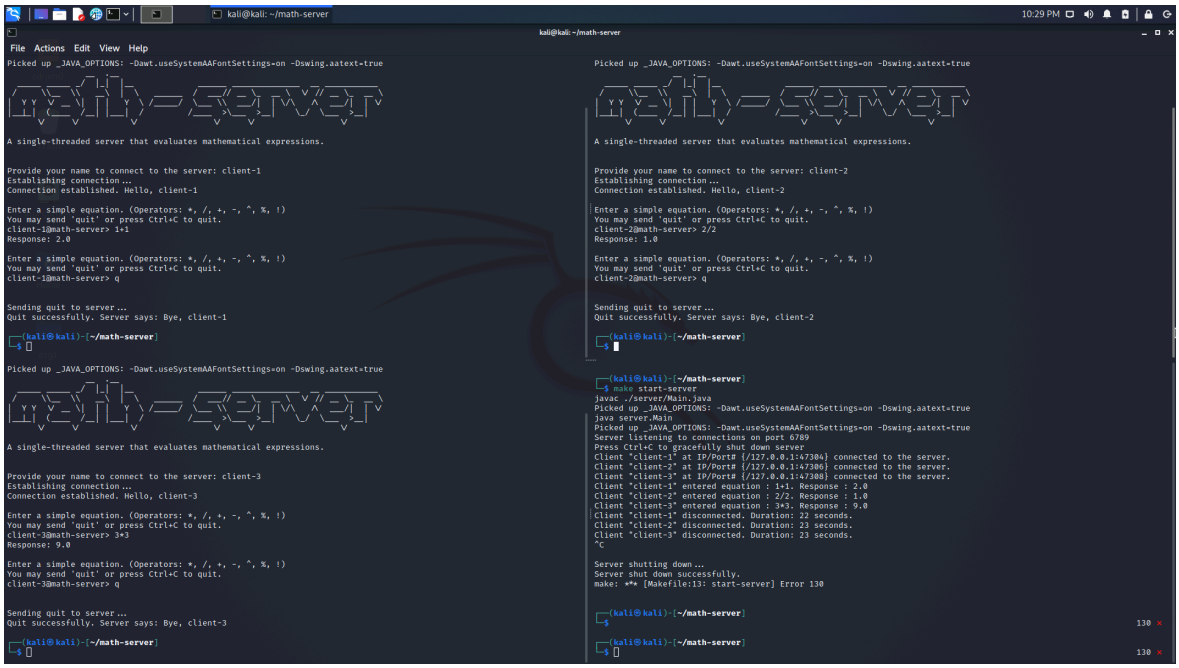

Image 2: Execution of the client and server applications using the Makefile in Linux.