Lab 3: Networked Calculator

TA in charge: Ethan Lee

In this exercise you are going to get some practice working with protocols. In particular, you're going to learn to frame and parse input to implement a very simple calculator.

You are going to write a Go TCP-based server that reads commands from its input, and sends the results back to the client over the connection. A few examples are given later in this document.

Updates

- [Apr 16] - Clarifying note at the bottom of the write-up that regrade requests will be time limited and grade will be finalized after 3 days of grades being out.
- April 18, 8:34 PM - clarified expected behavior when testing with nc (https://piazza.com/class/m852xr25ufs14m/post/287)
- April 21, 5:02 PM - clarified differences in expected behavior with nc and alternatives to testing with nc.

GitHub Link

- You can accept the GitHub invitation for the starter code via https://classroom.github.com/a/lciad17F

SimpleCalc specification

Each SimpleCalc instance has an *Accumulator*, which is simply an integer variable. The accumulator's initial value is 0. This value can be changed via the following commands:

- *ADD*: Adds an integer to the accumulator, and stores the result back in the accumulator
- *SUB*: Subtracts an integer from the accumulator, storing the result back in the accumulator
- *MUL*: Multiplies an integer by the value of the accumulator, storing the result back in the accumulator
- *SET*: Sets the value of the accumulator to the given integer

You only have to support integers (no floating point numbers), and the accumulator's value, and all arguments to the above four commands, should fit into the **`int64`** datatype (you do not need to support values outside this range). You do not need to support overflow or other

corner-case/exceptional conditions (we won't push the accumulator's value close to the edge of the integer type's range).

Each command consists of the command name, followed by a space, followed by the argument, followed by a **CRLF** terminator. Thus, each command is on a line by itself.

A sequence of commands is permitted, and the end of that sequence is signified by a blank line. The accumulator is reset to 0 after each sequence.

Your program should read and parse commands, keeping a running value in the accumulator, until you encounter the end of the sequence (i.e., a blank line). At that point, your program should return to the client the value of the accumulator. Your program should be able to continue reading in sequences from a given client even after it finishes reading everything in the initial input, i.e. the server should not force the client to close the connection. If the client itself disconnects, the server should close its handler for that specific connection (doable with a defer statement). After reading inputs, your server should continue to accept connections from clients, and the server should not exit back to the command line unless it is forcibly shut down (i.e. with "kill" or ctrl + D).

Examples

Example 1

Given input:

```
ADD 3<CRLF>
SUB 4<CRLF>
ADD 10<CRLF>
<CRLF>
```

SimpleCalc should return to the client:

```
9<CRLF>
```

Example 2

Given input:

```
SET 3<CRLF>
ADD 3<CRLF>
ADD 2<CRLF>
```

```
<CRLF>
ADD 10<CRLF>
SUB 3<CRLF>
<CRLF>
```

SimpleCalc should return to the client:

```
8<CRLF>
7<CRLF>
```

Example 3

Given input

```
SET 5<CRLF>
MUL 4<CRLF>
<CRLF>
SET 2<CRLF>
ADD 5<CRLF>
SET 3<CRLF>
<CRLF>
```

SimpleCalc should return to the client:

```
20<CRLF>
3<CRLF>
```

Notes about testing

We will mainly test that the output produced matches an expected result, not that netcat behaves a specific way. The described commands and behavior may slightly differ based on your specific choice of OS and command prompt. Note that you can set up your own client (for instance, based on turing-client.go) to achieve similar effect.

To develop your server, you will need to test it with various inputs. The 'nc' (netcat) tool is very useful for this type of testing. You can type "man nc" at the Linux command line to get documentation on this tool.

In particular, *nc* allows you to open a connection to a server, and to send an input file over that connection, printing out whatever response is returned (or redirecting that output to a file). For example:

```
$ nc localhost 5555 < input.txt > output.txt
```

This will open a TCP client connecting to host *localhost* on port 5555. Once that connection is established, the nc client will send the contents of *input.txt* over the socket, and write out any reply into *output.txt*. Since nc by design "hangs" even after the entire stdin is read (unless you choose to add the -q flag, which we will not test for), waiting for additional responses from the server, you should not "force" the client to quit the connection after all initial inputs have been read. nc should only jump back to the command line when you kill the server, which closes the TCP connection.

Creating input files

The SimpleCalc protocol requires a very specific delimiter consisting of newlines and carriage-returns. When you create testing input files, the end of line's representation will depend on your computer and operating system. By default, Mac and Unix computers (e.g., Linux) end a line with a **LF** line feed character. Windows/DOS traditionally ended lines with a **CR** then a **LF**. As you create your own input files for development and testing, you'll need to make sure that each line ends in the **CRLF** delimiter.

To do this you can use the *printf(1)* command-line tool. This acts just like *printf()* in C/C++, in that you can directly specify non-printable characters to output. You can redirect that output into a file as well. So for example, you can store a test case in a file called test01.txt this way:

```
$ printf "ADD 3\r\nSUB 4\r\nADD 19\r\n\r\n" > test01.txt
$ cat test01.txt
ADD 3
SUB 4
ADD 19

$
```

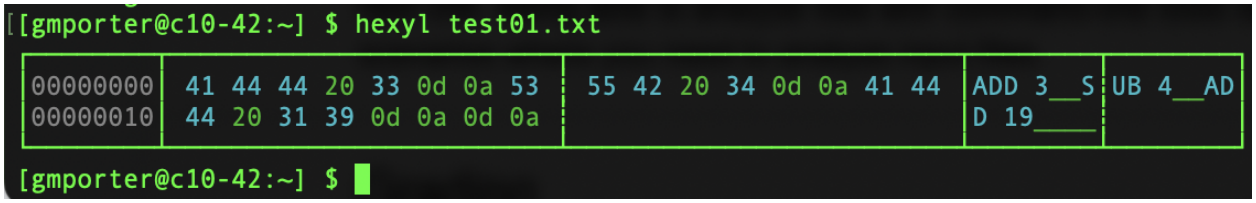
Examining non-printable characters like CR and LF

As I'll mention in class, you can run 'hexdump -C' on a file and it will give you output that indicates the characters:

```
$ hexdump -C test01.txt
```

```
00000000  41 44 44 20 33 0d 0a 53  55 42 20 34 0d 0a 41 44  |ADD 3..SUB 4..AD|
00000010  44 20 31 39 0d 0a 0d 0a                           |D 19....|
00000018
$
```

If you install the package “hexyl” it gives you a version of hexdump that prints in pretty colors:



This shows that the first line is "ADD 3" followed by "\0d" (CR) followed by "\0a" (LF), then the next line. Being able to analyze files with command-line tools is very helpful in real-world scenarios where you need to analyze input files.

Grading

We’ll run your code against a number of sample inputs and assign points accordingly. To test your code, you can build your own little programs and test them against the known-good output.

We're going to start your server program, and send it a number of examples of SimpleCalc "programs", and assign points proportionally to the number of correct responses. For example, if there were 10 programs in the input file, then each would be worth 10% of the final grade.

Report

There is no report for this lab–only the code.

Expected effort level

Our reference solution was about 100 lines of code.

Submission checklist

1. Commit and ‘push’ your code to GitHub (main branch) based on the invitation you received above.
2. Go to gradescope (there is a link from Canvas) and submit your code there. You will need to link your GitHub repository to Gradescope for that to work.

Note: Grades will be finalized three days after grades are returned, and no additional regrade/correction requests will be entertained after that point.