Computer Networks
Skill Check

Prof. Dr. rer. nat. Nils Aschenbruck

Leonhard Brüggemann, Bennet Janzen, Leon Richardt, Alexander Tessmer

General information:

- Read the **complete** task sheet **carefully** before you start working on it!
- By starting to work on the task, you declare that you are able to do so both physically and mentally.
- You must only use your assigned computer and work in the directory *Task* on the desktop. This is vital for submission (see below)!
- In case of an attempt to deceive, the assignment of all parties involved will be graded immediately as failed. There is no prior warning. Mobile phones that are switched on and other communication devices (such as smart watches) are also considered an attempt at deception! Changing the font size is also considered an attempt at deception.
- You are not allowed to eat in the digital exam labs, drinking is allowed.
- If you have to go to the bathroom, notify the supervisor. They will make sure that no two people go at the same time.
- Submission:
  - Make sure that your final version is saved in the directory *Task*.
  - Notify the supervisor to present your solution and **do not** turn off your computer.
  - The solution is automatically copied from your computer.
- System Information:
  - The tasks assume you are working on **Ubuntu 22.04**, using **Python 3.10**. We cannot guarantee support for other system environments.
  - Files that are part of this task are provided in the directory *Task* on the Desktop.
- Provided Documentation:
  - Linux man pages
  - Python documentation (might take a few seconds to open)
  - OpenSSL Cookbook
- Recommended IDEs: VS Code, vim, GNU nano, gedit

Please note: In order to pass, your solution must fulfill **all requirements** specified in the task. It is not sufficient to only complete a subset.

Task 1-C: Distributed Calculator

In this task, you will implement a Python server acting as a "distributed calculator". The server allows clients to connect via TCP and send commands (`ADD`, `MULTIPLY`, `RESET`) which the server must execute. When it receives the `GET` command, the server returns the "running total" of commands performed up to that point. To help you develop your server implementation, we provide a client implementation which you may use.

1 Requirements

The server must keep a running total t of the commands performed. On startup, set $t \leftarrow 0$, and listen for incoming TCP connections on port `4711`. For this task, it is sufficient if your implementation can only handle a single client connection at a time.[1] Regardless, your implementation must be able to handle multiple connections *after* one another, i.e., the server should accept new connections after a client disconnects. Note that you *must not* reset t when a client disconnects. t should only be reset on reception of the `RESET` command, or when the server shuts down. The server must only shut down when requested, e.g., via Ctrl + C in the terminal session.

Let x be a double-precision (8 bytes) floating-point number. Your server must implement the following commands:

`ADD` x	Performs $t \leftarrow t + x$
`MULTIPLY` x	Performs $t \leftarrow t \cdot x$
`RESET`	Performs $t \leftarrow 0$
`GET`	Returns the current value of t

The commands are transmitted from the client in a custom message format, which is described in detail in Section 3. Example 1 illustrates the kind of client behavior your server must be able to support.

You have passed the task when your server can be used as a reliable endpoint for the REPL of the client (cf. Section 2).

Hint

When developing your implementation, you might find it useful to call

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

before binding the socket to an IP address. This allows the socket to be rebound to the same address even when it was not shut down properly. See `man 7 socket` for more information.

[1] In later pools, you will extend the implementation to support handling multiple clients in parallel.

Example 1

In a correct implementation, one possible client session could look like this:

```
student@host:~/task-c$ ./client
Trying to connect to 127.0.0.1:4711 ... Connected!
> GET
[GET] 0.0
> ADD 42
> MULTIPLY 42
> GET
[GET] 1764.0
> MULTIPLY 0.5
> ADD -881
> GET
[GET] 1.0
> RESET
> GET
[GET] 0.0
> Ctrl + D
Goodbye!
```

In particular, note that negative and non-integer numbers are supported arguments as well.

2 Client Description

To start the client, run

```
student@host:~/task-c$ ./client [port]
```

with an optional `port` parameter. The client attempts to open a TCP connection to the specified port on the local host. When the `port` parameter is omitted, the client attempts to connect to port `4711` per default.

After a connection has been established, the client enters a *read-eval-print loop* (REPL). The user may enter a command via the terminal, which is translated into a message according to Figure 1 and sent to the server. In case of the `ADD`, `MULTIPLY`, and `RESET` commands, no response is returned by the server, so the REPL returns to reading input from the user. In case of the `GET` command, the server returns a response (cf. Figure 2) which the client parses and prints to the terminal before waiting for further input from the user. To exit the application gracefully and close the connection to the server, input Ctrl + D.

3 Protocol Format

Each of the commands defined in Section 1 is associated with a message. The formats of these messages are shown in Figure 1. Therein, the first field is an identifier for the type of command, represented with a single UTF-8-encoded character (`A`, `M`, `R`, or `G`). For the commands with a parameter, the second field is the parameter value x, encoded as an 8-byte floating-point number.

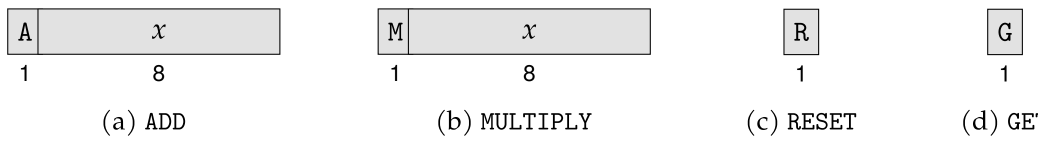

Figure 1: Format of command messages sent by the client. The numbers below the fields indicate the field length in number of bytes.

Since the `GET` command instructs the server to return the current value of t, a response message format is defined in Figure 2. Again, t is encoded as an 8-byte floating-point number.

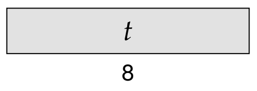

Figure 2: Format of `GET` response messages sent by the server. The number below the field indicates the field length in number of bytes.

4 Follow-Up

Prepare to answer the following questions:

1. Would you recommend the use of UDP over TCP for this application? Why or why not?
2. Which difficulties do you anticipate when the server is extended to support multiple concurrent client connections? Do you know how to solve them?

Note

For transparency: You will not have to answer these questions during the Skill Check. Nonetheless, we encourage you to think about how you would answer them. See them as an opportunity to improve your understanding of the relevant course material. Questions similar to these might be part of the final exam.

Good luck!