Assignment: Building Bridges in Computer Networks

We're going to work on and walk through the code that forms the basis of many, many internet-connected programs: the socket.

Learning Objectives

- Recall what client-server architecture is and how clients and servers work together.
- Understand why protocols are important for different languages to talk to each other (protocols are *language-agnostic*).
- Apply what you know by changing code to use a specific port number, learning about hostnames and ports.
- Analyze how to add new commands to a server using function pointers, a common pattern in server development.
- Evaluate how to reuse code by using your sorting algorithms from previous assignments.

GitHub Classroom

Accept the assignment from GitHub Classroom: **LINK**

Step 1: Study the Starter Code

Study the starter code.

Server

The server code is located in the `server/` folder. This is the C implementation of the TCP server. The server listens on a fixed port and can process only one command: `help <arg>`. The server responds with a message `"Helping you with <arg>"`. For any other command, the server responds with `"Invalid command"`.

The server is implemented in the `tcp_server.c` file. The server uses the `socket` and `bind` functions to create a socket and bind it to a port. It then uses the `listen` function to listen for incoming connections. When a client connects, the server uses the `accept` function to accept the connection. Finally, the server uses the `recv` and `send` functions to receive and send data.

The second file, `commands.c`, has two functions: `process` and `help`. The `process` function is a command dispatcher. It takes a command and its arguments and calls the appropriate function. There is an array of function pointers that map commands to functions (defined in the header file `commands.h`). The `help` function is called when the `help` command is received. You will need to add more commands to the server.

You may use ChatGPT to explain the specific functions and how they work.

Compile the server with `make` and run it with `./tcp_server`. The server will start listening on port 10393 (you can change this in the code).

> What is Makefile? It is a file that contains a set of directives used by a make build automation tool to generate a target/goal. It is called Makefile because it controls the program's build process. It is a simple way to organize code compilation, so that you don't have to remember the compilation command and its switches. Just type `make` and it will compile your program.

Clients

The clients are located in the `clients/` folder. There are two clients that can connect to the server. The first client is written in C, and the second one in Python. Each client is compatible with the server. Both work by sending a command to the server and receiving a response.

Compile the C client (there is a single C file, so there's no Makefile, and you can compile it with `gcc`) and run it with `./tcp_client localhost`. The client will connect to the server.

> Both server and client should be run on the server, but use two separate terminals.

Try sending some commands from the client to the server and see how the server responds:

- `help abc` (should respond with "Helping you with abc")
- `help xyz` (should respond with "Helping you with xyz")
- `hello` (should respond with "Invalid command")

Similarly, run the Python client with `python3 tcp_client.py localhost`. The client will connect to the server (make sure that your C client is not running). Run the same commands and see how the server responds.

Step 2: Refactor the Server

In the starter code, the server listens on a fixed port (10393). Refactor the server to take the port number as an argument. The server should be able to listen on any port number.

After refactoring, your server should be run with the following command:

```
./tcp_server <port>
```

Step 3: Refactor the Clients

Refactor **both** clients to add the second argument, which is the port number. The clients should be able to connect to the server on any port number.

After refactoring, your C client should be run with the following command:

```
./tcp_client <hostname> <port>
```

And your Python client should be run with the following command:

```
python3 tcp_client.py <hostname> <port>
```

Step 4: Add More Commands to the Server

In the starter version, the server can only process one command: `help <arg>`. Add more commands to the server. The server should be able to process the following commands:

calc

The server should be able to process the command `calc <arg1><operator><arg2>`. The server should respond with the result of the operation. For example, `calc 2+3` should return `5`.

For division, you should use integer division (e.g., `calc 5/2` should return `2`). It's okay to ignore division by zero and other edge cases.

sort

The server should be able to process the command `sort <arg1> <arg2> ... <argN>`. The server should respond with the sorted list of arguments. For example, `sort 6 2 3 7 4` should return `2 3 4 6 7`.

You can implement any sorting algorithm you want (reuse a sorting algorithm from a previous assignment).

help

The help command should be refactored, so that it can help with the new commands as well. For example, `help calc` should return "Calculates the result of a op b. Usage: calc a op b". For `help sort`, the server should return "Sorts a list of numbers. Usage: sort n1 n2 n3 ...". If there are no arguments, the server should return the list of available commands as follows: "Available commands: help, calc, sort". If the command is not recognized, the help command should return "This command does not exist".

BONUS: An extra command of your choice

You can add an extra command of your choice. For example, you can add a command that returns the current date and time, or a command that reverses a string, or anything else you want.

This is a bonus task and is not required, but it will give you extra practice!

Implementation Details

All commands must be implemented **in the server**. The server should be able to process any of the commands. The clients should **not** be aware of the commands.

Do not change anything in the clients when you add new commands to the server. The clients should work with the new commands without any changes.

Step 5: Demo the Server and Clients

Record a video showing the clients interacting with the server. The video should show the following:

1. Log into the server with ssh in multiple terminals.
2. Run the C server in one terminal.
3. Run the C client in another terminal and test the commands: `help`, `calc`, `sort`, and the extra command (if you implemented one).
4. Run the Python client in another terminal and test the commands: `help`, `calc`, `sort`, and the extra command (if you implemented one).

This is recorded and submitted as a video.

The maximum length of the video is **5 minutes**. If your video is longer than 5 minutes, you will lose points.

Using AI

In this homework, you are dealing with the codebase that you didn't write. It also uses some elements that you might not be familiar with, such as sockets. This is a great opportunity to use AI to help you understand the code and the concepts.

For this homework, you are encouraged to use AI (ChatGPT, or GitHub Copilot Chat) to help you understand the code and implement the new features. You can ask questions about the code, the concepts, and the implementation details. You can also ask for help with the refactoring and the new features.

This does not mean that you should rely solely on AI to complete the homework. You should still study the code and the concepts, and you should still implement the new features yourself. AI is just a tool to **help** you understand and implement the code as you would normally do.

Resources

To understand the C code of the server and the clients, you may want to read https://man7.org/linux/man-pages/man2/socket.2.html.

For the Python code, you can read the official documentation: https://docs.python.org/3/library/socket.html.

Visit the ultimate authority, The Linux Programming Interface's chapters on sockets, for additional help on parameters, function calls, and details: https://learning.oreilly.com/library/view/the-linux-programming/9781593272203/xhtml/ch56.xhtml.

You can also use Chapter 2 of TCP/IP Sockets in C reference book to help understand some of the code, although their implementation is slightly different: https://learning.oreilly.com/library/view/tcp-ip-sockets-in/9780080923215/OEBPS/B9780123745408000043.htm.