Varun Narravula
ID: 923-287-037

Prof. Qun Wang
CSC645 Computer Networks

Project 2 Report

Overview

The goal of this project was to create a server and client to demonstrate usage of the TCP and UDP transport protocols.

I opted to use the Go programming language to create the server and client, since Go has strong built-in support for TCP and UDP and is often used in production to make networking services.

The code for the server and client is contained in my GitHub repository for this assignment. The code is also zipped and included with this report on Canvas for reference. A README is included with instructions on how to run the program.

Design

I organized the code in a single package called `project2` that builds a single executable called `project2`. This package is a command-line program that takes subcommands and flags to control the behavior of the program.

There are two subcommands:

- `server`: runs the server program for TCP and UDP
- `client`: runs the client program for TCP and UDP

Each subcommand takes the following flags:

- `--port`: the port number to use for the server or client (default: 8080)
- `--address`: the address to use for the server or client (default: localhost)
- `--tcp` || `--udp`: the protocol to use for the server or client

The `client` subcommand also takes a positional argument called `message` that is the sentence to send to the server in order to reverse it.

Usage

In order to demonstrate the program's functionality, the server must be running continuously in the background.

Run the server using `go run . server --<protocol>`, where `<protocol>` is either `tcp` or `udp`. The server will listen on the specified port and address if specified, or on port 8080 and `localhost` by default. The server process can be killed with `Ctrl+C`.

I'm not sure how well this will work on Windows, so I recommend using Linux or macOS for testing this project, due to the fact that Windows doesn't have support for POSIX signals such as `SIGTERM` or `SIGINT`.

Then, separately run the client using `go run . client --<protocol> <message>`. The client will connect to the server and send the specified message to the server. The server will then respond with the reversed message, and the client will print the response and exit.

Task 1: TCP Server/Client

To create the TCP server and client, I used the `net` package in Go, which has many built-in functions/methods for creating TCP and UDP sockets, as well as managing the connections on them.

Creating a TCP server requires dialing the address with `net.ListenTCP()` after resolving the address with `net.ResolveTCPAddr()`. The server can then acccept connections with the `net.TCPListener.Accept()` method, and this can be done forever until the server receives a shutdown signal.

When a connection request is accepted, we can then handle this inside of a new "goroutine", which allows us to continue running other tasks (such as accepting new connections again) concurrently. In order to actually reverse the message that is sent though, we can read the contents of the connection's data with `net.Conn.Read()` method, and then reverse this using a function like below:

```
func ReverseWords(buf []byte) []byte {
  tmp := strings.Split(string(buf), " ")
  slices.Reverse(tmp)
  return []byte(strings.Join(tmp, " "))
}
```

Then, once we have the reversed message, we can send it back to the client by writing it to the connection again with `net.Conn.Write()`. The connection can then be closed, and the goroutine can exit. We are now done handling this connection request. Yay!

The client is much more simple, as it only needs to connect to the server and obtain the response. First, we need to dial the destination address by resolving it and connecting to it with `net.DialTCP()` based on the passed-in parameters. After the connection is established, we can send the message to the server with `net.Conn.Write()`, and then read the response with `net.Conn.Read()`; basically the reverse of what we did with the server. The response can then be printed.

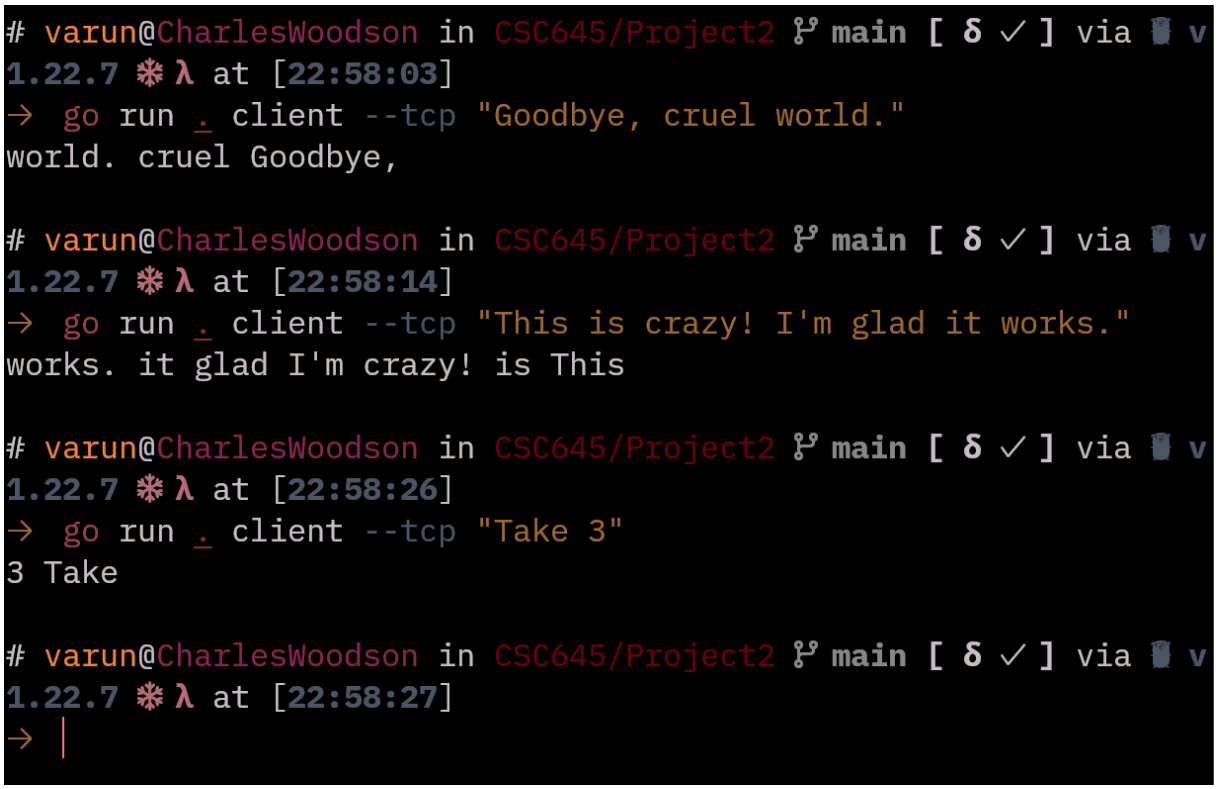
In order to keep implementation details simple, we can assume that the server and client are both using fixed-size buffers for sending/receiving data with a maxium message size of 1024 bytes.

Closing the server itself is a bit more complicated, as we need to wait for all connections to be closed before exiting. Technically, we don't *need* to wait for every connection to be closed, but it's good practice to at least try and wait for many connections to be closed before exiting, instead of abruptly exiting and leaving some connections broken, since clients can act up because of this behavior. To do this, we can close off the server socket with `net.TCPListener.Close()` when the shutdown signal (`Ctrl+C`) is received, and then wait for a second to ensure that most connections are closed.

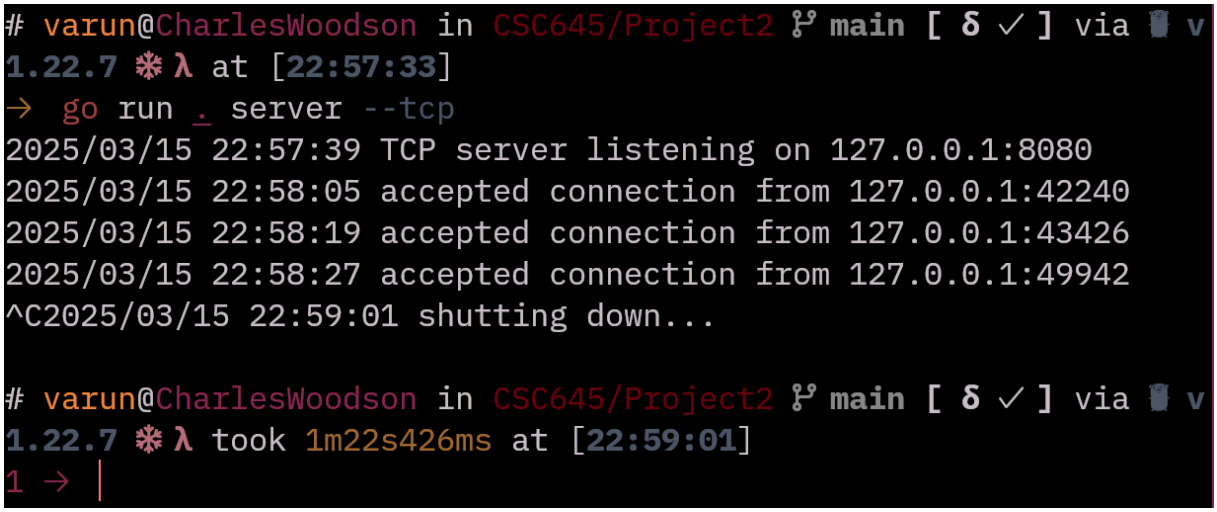
To exit the server program itself, we can handle `net.TCPListener.Accept()` errors slightly differently: if the operation requested was "accept", and this failed, we can safely assume that the server has been closed, and we can break the infinite loop of listening for new connections, because no new connections are being accepted anymore.

Here are some runs of the server and client below, along with server logs for the runs.

TCP Client Run



TCP Server Logs



Task 2: UDP Server/Client

To create the UDP server and client, I also used the `net` package in Go. Implementation for the server and client is rather similar to the TCP server and client, except that instead of using `net.TCPListener.Accept()` and such, we use the UDP equivalents of these functions: `net.UDPListener.Accept()` and such.

The structure of the server and client is also rather similar, so I'll leave some of the details of the implementation out if they are similar enough.

In order to keep implementation details simple, we can assume that the server and client are still both using fixed-size buffers for sending/receiving data with a maximum message size of 1024 bytes.

Using the `net.UDPConn.ReadFromUDP()` method, we can directly obtain the data from a UDP request, as well as the client address to send a response back to.

Then, we can reverse the message using the same function as above, and then send the response back to the client using `net.UDPConn.WriteToUDP()`. Since UDP is a connectionless protocol, we don't need to worry about closing the connection, because there's no connection to close in the first place.

However, since there are no connections to close, we need to handle the `net.UDPConn.ReadFromUDP()` errors slightly differently to shut down the server: if the operation requested was "read", and this failed, we can safely assume that the server has been closed, and we can break the infinite loop of listening for new connections.

UDP Client Run

```
# varun@CharlesWoodson in CSC645/Project2  main [ δ ✓ ] via  v
1.22.7  λ at [23:00:04]
→ go run . client --udp "Goodbye, cruel world."
world. cruel Goodbye,

# varun@CharlesWoodson in CSC645/Project2  main [ δ ✓ ] via  v
1.22.7  λ at [23:00:06]
→ go run . client --udp "This is cool, I like UDP"
UDP like I cool, is This

# varun@CharlesWoodson in CSC645/Project2  main [ δ ✓ ] via  v
1.22.7  λ at [23:00:14]
→ go run . client --udp "Take 3"
3 Take

# varun@CharlesWoodson in CSC645/Project2  main [ δ ✓ ] via  v
1.22.7  λ at [23:00:22]
→
```

UDP Server Logs

```
# varun@CharlesWoodson in CSC645/Project2  main [ δ ✓ ] via  v
1.22.7  λ at [22:59:59]
→ go run . server --udp
2025/03/15 23:00:03 UDP server listening on 127.0.0.1:8080...
2025/03/15 23:00:06 accepted connection from 127.0.0.1:42733
2025/03/15 23:00:14 accepted connection from 127.0.0.1:47045
2025/03/15 23:00:22 accepted connection from 127.0.0.1:48153
^C2025/03/15 23:00:23 shutting down...

# varun@CharlesWoodson in CSC645/Project2  main [ δ ✓ ] via  v
1.22.7  λ took 20s797ms at [23:00:23]
1 →
```