Programming Assignment #7

Introduction to Computer Networks

Horay!

You've cleared stage 1. It is quite a long way already if you haven't noticed:

- Unix commands, SFTP, and VPN in PA1
- access files on the disk in PA2
- write a file upload client using TCP socket in PA3
- write a file upload server using TCP socket in PA4
- extend PA4 to allow repeated file uploads in PA5
- extend PA5 to allow concurrent file uploads in PA6

Some of you might need a bit more time to adapt. It's OK. You'll get a hang of it sooner or later. Some of you might find the programming style of Go amusing. That's cool. Polly recalled her first biking lesson. She thought there's no way she'd be able to ride a vehicle with less than 3 anchor points. She was right for --- like 2.5 minutes. Before she knew it, her uncle who's promised to hold on to the bike no matter what, was smiling and waving yards behind. It's Polly's turn to smile from behind now :).

Stage 2

There will be 3 assignments at the 2nd stage, i.e., PA7 to PA9. In the end, you will come to implement a simplified Web server that allows secure downloading of html objects:

- HTTP Request Interpreter in PA7
- HTTP Response Generator in PA8
- Secure Web server in PA9

For each assignment, there are very different ways to come to the same result. You'll be introduced ways to implement the Web server using the socket APIs (from native `net` packages), as well as the Web APIs (from high-level `net/http` packages). Those who find system-level programming empowering and fancy a career in network and system software engineer, definitely try the native socket APIs out (use as few high-

level packages as possible). Those who find application-level programming more amiable and seek a career in app development, please do try the Web APIs out (use as many high-level packages as possible).

In either case, you'll be googling, and trying and failing more at this stage, as the flexibility is much higher. No worries. You'll also be allowed more time to explore in the process. You see the frequency is lowered to one assignment per 2 weeks from this point on. "Dope. I'll still start a day before the assignment is due," some of you might be thinking. If you have trouble convincing yourself to start early. Think this way – the work needs to be done anyway, sooner or later. The purpose of college-level teaching is not just to propagate knowledge, but also to allow exploration, of the Golang APIs and your selves.

The Assignment

`PA7.go` should work like a half-baked Web (text/html) file server that receives, interprets the HTTP request messages, and prints on screen the file size of the object requested. More specifically, your `PA7.go`:

(1) Listens at <your port#> until there's an HTTP request
(2) reads from the socket
(3) finds the path and name of the text/html file requested
(4) In case the file exists, prints on server screen the file size
(5) In case the file doesn't exist, prints on server screen "File not found".
(6) closes the connections and goes back to (1)

To prepare you for the task, follow through the 2 examples below.

1. HTTP Request as a `Request`

Let's start with the easier example, in which reading and interpretation of the HTTP Request message is almost effortless once you get to know the `Request` `struct` in `net/http` package. Start a file `simple-Request.go` and type up the following code.

```
package main

import "fmt"
import "bufio"
import "net"
import "net/http"


func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    fmt.Println("Launching server...")
    ln, _ := net.Listen("tcp", ":<your port#>")
    defer ln.Close()
    conn, _ := ln.Accept()
    defer conn.Close()

    reader := bufio.NewReader(conn)
    req, err := http.ReadRequest(reader)
    check(err)

    fmt.Printf("Method: %s\n", req.Method)
    fmt.Printf("Host: %s\n", req.Host)
    fmt.Printf("User-Agent: %s\n", req.UserAgent())
}
```

Replace `<your port#>` with the port number assigned to your team. Start the `simple-Request.go` first.

```
$ go run simple-Request.go
Launching server...
```

Then, start a Chrome or Firefox browser and request for http://127.0.0.1:<your port#>/ Alternatively, use the Unix command-line call – `curl` to request an arbitrary Web object on 127.0.0.1:<your port#>.

```
$ curl 127.0.0.1:<your port#>/
Curl: (52) Empty reply from server
$
```

If you are using a Web browser for testing, the browser is likely saying something is wrong, e.g., not connecting to the server or not getting messages back from the server.

The terminal running the `simple-Request.go` server should print the following.

```
$ go run simple-Request.go
Launching server...
Method: GET
Host: 127.0.0.1:<your port#>
User-Agent: curl/7.64.1
```

Code walk-through:

- `net/http` is the package where the http-related data `struct` and APIs are defined. Contained in the package is also a Web file server itself. This is very unique among the programming languages there are today. You can probably see a bit of the emphasis of Golang being a language built to enable fast and efficient development of Internet applications. The code (including examples and tests) is open at: https://golang.org/src/net/http/
- `bufio.NewReader(conn)` converts the socket connection `conn` into an I/O buffer such that APIs such as `ReadString()` can be applied conveniently to read textual input.
- `http.ReadRequest()` is a special `bufio` `reader` API defined in `net/http` (not in `bufio`). It reads from the socket, process the textual input, and store the information as a `Request` object, whose `struct` is defined in `net/http/request.go`, line 108-325.
- `http.ReadRequest()` takes the `reader` as input and returns two values. The

first value `req` is the `Request` object containing the interpreted HTTP request. The second value `err` carries the error code.

- `net/http/request.go` is well documented (whether you like reading others' code or not). One can see in line 115 that the `Method` field stores the method of the HTTP request. `req.Method` therefore refers to the method specified in the HTTP request message.
- In a way, we could borrow what's already done in `http.ReadRequest()` for much of what `PA7.go` demands – interpreting the HTTP request messages.
- In line 240, `Host` is defined as a `string`, which stores the host information in the HTTP request message header. `req.Host` should give us the string in the Host header line.
- With a bit more exploration, one sees also `UserAgent()` in line 406, an API returns the User-Agent value in the HTTP message header. Therefore, `req.UserAgent()` should give us the string in the User-Agent header line. Interestingly, `req.Header.Get("key")` should give you whatever the value the key corresponds to if you read into the definition of `UserAgent()`. Definition and manipulation of the `Header` struct is in net/http/header.go. You'll see in there that `Get()` is a `Header` object API.
- Now explore away. Look for ways to access the tokens or values required to complete the assignment. Note that some tokens in the command line and some values in the header lines might be a bit harder to access.
- It might take some time to find a way to access the tokens/values you are looking for. The advantage is however once you get to know the `Request struct`, you'll be able to access the various tokens/values with very brief code and without the trouble of needing to process the HTTP request messages.

2. HTTP Request as a `string`

The alternative is to read the HTTP request message as a string, process/interpret the string yourself, and extract whatever token/value that are needed to complete the assignment. Start a file `string-Request.go` and type up the following code.

```
package main

import "fmt"
import "bufio"
import "net"
import “strings”

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    fmt.Println("Launching server...")
    ln, _ := net.Listen("tcp", ":<your port#>")
    defer ln.Close()
    conn, _ := ln.Accept()
    defer conn.Close()
    reader := bufio.NewReader(conn)
    for {
        req, err := reader.ReadString('\n')
        check(err)
        if req == "\r\n" {
            break
        }
        tokens := strings.Split(req, " ")
        for i := range tokens {
            fmt.Println(tokens[i])
        }
    }
}
```

Similarly, replace `<your port#>` with the port number assigned to your team. Start `string-Request.go` first.

```
$ go run string-Request.go
Launching server...
```

Then, `curl` an arbitrary object on 127.0.0.1:<your port#>.

```
$ curl 127.0.0.1:<your port#>/
Curl: (52) Empty reply from server
$
```

The terminal running the `string-Request.go` server should print the following or something similar.

```
$ go run string-Request.go
Launching server...
GET
/
HTTP/1.1

Host:
127.0.0.1:11999

User-Agent:
curl/7.64.1

Accept:
*/*
```

Code walk-through:

- The `strings` package is not yet formally introduced. A good thing is though some of you guys have started using it for the purpose of earlier assignments.
- `strings.Split()` is an API defined in the `strings` package that splits a long string by a delimiter, which is also of the type `string`. In the example, we are

splitting a line (`req`), by a space (`" "`).

- `strings.Split()` returns an array of substrings (`tokens`) as the result of the split.
- The `tokens` array could be an arbitrary size. The `for` loop there will go through all elements in the index range of the array and `fmt.Println()`.
- Taking this approach, the HTTP request message is entirely yours to interpret and process. Extracting the tokens needed for the assignment will take a bit time but straightforward.

3. PA7.go

Again, make sure your `PA7.go` is listening on the port number you are assigned to. To test your `PA7.go`, use a Web browser or `curl` to request an html file from the `PA7.go` server. Please feel free to generate an arbitrary text/html file for the testing purpose. Or `curl` a test html file to your PA7 directory (where `PA7.go` will be running in):

```
$ curl homepage.ntu.edu.tw:80/~pollyhuang/teach/intro-cn-pa/server-test.html > server-test.html
```

Now `curl` the `server-test.html` file from the client terminal and see if your `PA7.go` server is printing the file size correctly.

To help you verify your implementation, polly has made the compiled byte code of her `PA7.go` available here: http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA7/PA7. Again, polly's `PA7` is configured to run on port# 11999.

4. Submit your PA7

`ssh` to the `140.112.42.161` workstation. At the team account's home directory, create a directory `PA7`. Upload your `PA7.go` to directory `PA7`. Test your `PA7.go` again on the workstation just to make sure it's working as expected.