Programming Assignment #5

Introduction to Computer Networks

The Assignment

You might be thinking already. Isn't a server always on and ready for client requests any time? The `PA4.go` however accepts only a file upload and then quits. You are right. `PA4.go` is far off a professional server. To address the issue, you need to extend `PA4.go` to `PA5.go` where the file upload service will continue forever. More specifically, the server:

(1) listens at <your port#> until there's an upload request
(2) reads from the socket first the file size (just the number in a single line)
(3) reads from the socket one line at a time
(4) prepend the line count to each line and store the new line into a new file: `whatever.txt`
(5) repeats (3) and (4) until all lines in the file is processed
(6) sends a message back that tells the client the original file and the new file size
(7) closes the connection and goes back to (1)

Give it a bit of thought. You'll see this assignment is light. There is no need of extra APIs. The example below is just for your amusement.

1. Professional Simple Server

This example implements the simple server's service in an infinite loop (receiving a string from the socket, printing it on the screen, and sending back the size of the string). Start a file `server-loop.go` and type up the following code.

```
package main

import "fmt"
import "bufio"
import "net"


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

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

    for {
        conn, _ := ln.Accept()
        defer conn.Close()

        reader := bufio.NewReader(conn)
        message, errr := reader.ReadString('\n')
        check(errr)
        fmt.Printf("%s", message)

        writer := bufio.NewWriter(conn)
        newline := fmt.Sprintf("%d bytes received\n",
len(message))
        _, errw := writer.WriteString(newline)
        check(errw)
        writer.Flush()
    }
}
```

Replace `<your port#>` with the port number assigned to your team. Start the server code first.

```
$ go run server-loop.go
Launching server...
```

Then, run the `client-102.go` code (provided in PA4).

```
$ go run client-102.go
Send a string of 13 bytes
Server replies: 13 bytes received
$
```

The terminal running the server code should print the following and wait for the next client request (instead of returning to the prompt).

```
$ go run server-loop.go
Launching server...
Hello World!
```

Run the `client-102.go` code again, you should see the server terminal now shows:

```
$ go run server-loop.go
Launching server...
Hello World!
Hello World!
```

`server-loop.go` will continue to wait for the next client request until (hopefully) forever.

2. PA5.go

Again, make sure your `PA5.go` is listening on the port number you are assigned to. To test your `PA5.go`, use the modified `PA3.go` (such as the one used in PA4) to dial to the IP address of the machine your `PA5.go` is running on and the port number you are assigned.

To help you verify your implementation, polly has made the compiled byte code of her `PA5.go` available here: http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA5/PA5. Please refer to PA2's section on how to `curl` and `chmod` the byte code before execution. To not run into you guys' port numbers, polly's `PA5` runs on port# 11999.

3. Submit your PA5

`ssh` to the `140.112.42.161` workstation. At the team account's home directory, create a directory `PA5`. Upload your `PA5.go` to directory `PA5`. Test your `PA5.go` again on the workstation just to make sure it's working as expected.