Programming Assignment #3

Introduction to Computer Networks

The Assignment
The assignment will take you through a pair of basic client-server programs. You will be using the APIs for system I/O access again, but this time more specifically on how one writes or reads from a network socket. Take note: just like the screen, keyboard and files, a network socket is also a system I/O.

When you are done following through the examples, you should be ready to program `PA3.go` that works like a file upload client which:

(1) connects to the server that polly implements and runs already on the workstation at port 12000
(2) prompts the user for the upload filename
(3) sends first the file size (just the number in a single line)
(4) sends next the file content (the entire file)
(5) receives a message back from the server
(6) prints what the server says
(7) closes the connection and terminates the program

Now follow through the examples below and practice the socket APIs of Go.

1. Simple Server

This example implements a server that receives a string from the socket, prints it on the screen, and sends back the size of the string. Start a file `server-101.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#>")
    conn, _ := ln.Accept()
    defer ln.Close()
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    message := “”
    if scanner.Scan() {
        message = scanner.Text()
        fmt.Println(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#>` in the program (2nd line in `main()`) with the port number assigned to your team (http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/port_assignment.pdf). Run the code. You'll see the simple server is being launched. It'd look like the program hangs, but it is not. It is simply waiting for a client to connect. We'll see the simple server's action when we connect the simple client (next example) to it. Leave the server waiting and start another terminal to work through the next example. Or, ctrl-c (hold the ctrl key down and then press c) to force-quit the simple server for now.

✓

```
$ go run server-101.go
Launching server...
```

Code walk-through:

- `net.Listen()` is the first socket API to know. `net` is the package in which the `Listen()` function is defined. With the API, the simple server starts a socket and listens for a client request. The first parameter declares the connection type of the socket. In this example, created is a `"tcp"` socket (reliable data transfer). The second parameter indicates the port number the socket is listening on. Any client wishes to connect to the server will send the request to the server's IP address at this port number.
- `net.Listen()` returns two values, the socket handle and the error message. `ln` is the socket handle. To keep the example clean and short, we skip the error check and use `_` in place of a variable that holds the error message. Note that socket is also system I/O. A socket handle works just like a file handle or the standard I/O (`os.Stdin` and `os.Stdout`). The APIs we apply to a file will work just as well on a socket.
- With `ln.Accept()`, we call object `ln` 's class method `Accept()`. The API takes (or waits for) the 1st client request arriving at the listen socket. In turn, it creates another socket `conn`. `conn` is dedicated to data transmission between the client and server, so `ln` can focus on incoming client requests, working pretty much like a call dispatch.
- `ln.Close()` and `conn.Close()` are to close the two socket objects, `ln` and `conn`. `defer` we know will delay the closing to the end of the program.
- There are a lot more APIs for TCP/IP socket operation in the `net` package.
- `bufio.NewScanner()`, `Scan()` and `Text()` work just like what have been explained in the `bufio-read-file.go` example (PA2). The slight difference is we scan and read from a socket this time. `Scan()` reads a line from socket `conn`, without the new line. `Text()` converts the line (byte stream) to text.

- Likewise, `bufio.NewWriter()`, `writer.WriteString()`, and `writer.Flush()` work just like what's explained in `bufio-write-file.go`.
- `fmt.Sprintf()` is yet another API in `fmt`. It Printf() to a string essentially.
- `len()` returns the length of a string.

✓

2. Hello World Client

To pair with `server-101.go`, we now look at a simple client that sends a text message to the server, prints length of the text message, and receives a reply from the server. Start a file `client-101.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() {
    conn, errc := net.Dial("tcp", "127.0.0.1:<your port#>")
    check(errc)
    defer conn.Close()

    writer := bufio.NewWriter(conn)
    len, errw := writer.WriteString("Hello World!\n")
    check(errw)
    fmt.Printf("Send a string of %d bytes\n", len)
    writer.Flush()

    scanner := bufio.NewScanner(conn)
    if scanner.Scan() {
        fmt.Printf("Server replies: %s\n", scanner.Text())
    }
}
```

Replace `<your port#>` in the program (1st line in `main()`). Before trying out `client-`

`101.go` , start `server-101.go` on another terminal first. If you leave the simple server running at the end of the 1st example, there is no need to run `server-101.go` again. Now type the following at the prompt.

```
$ go run client-101.go
```

You should see the following:

```
Send a string of 13 bytes
Server replies: 12 bytes received
$
```

The simple client sends a line `Hello World!`, including the `\n`, to the servers. That's 13 bytes long (1 character is 1 byte). The client prints the length of the line and receives a reply from the server, which acknowledges receipt of a 12-byte stream, as `Scan()` ignores the new line, i.e., the `\n` character. You do want to be a bit careful counting the number of bytes received when `Scan()` is used to take input.

The terminal running the simple server should reflect – printing of the line received, terminating of the program after sending the line's length.

```
$ go run server-101.go
Launching server...
Hello World!
$
```

Code walk-through:

- The only API being new is `net.Dial()`. It is used to connect a client to the server. The first input parameter declares the connection type, `"tcp"` again. The second parameter indicates the IP address and port number the socket connects to. Note the symbol `:` in the middle, It separates the IP address and the port number.
- `net.Dail()` returns two values, the socket handle and the error message.

3. **PA3.go**

Now, start `PA3.go` and make sure it:

(1) connects to the server that polly implements and runs already on the workstation at port 12000
(2) prompts the user for the upload filename
(3) sends first the file size (just the number in a single line)
(4) sends next the file content (the entire file)
(5) receives a message back from the server
(6) prints what the server says, and finally
(7) closes the connection and terminates the program

You might want to go back to PA2 and review the APIs accessing files. You might need to exercise your creativity or detective ability a bit in the meantime to figure out a way to find the file size.

To help you verify your implementation, polly has made the compiled byte code of her `PA3.go` available here: http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA3/PA3. To download it straight to the workstation, use `curl` and make sure you change the permission properly using `chmod`.

```
    $ curl homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA3/PA3
> pollys-PA3
    $ chmod u+x pollys-PA3
    $ ./pollys-PA3
```

Cross compare execution result of your `PA3.go` to the outcome of executing `pollys-PA3`. If they work the same, you will be done and safe.

4. **Go Documentation**

For details and other APIs in the `net` package, visit this page:
https://golang.org/pkg/net/

You will see `Read()` and `Write()` APIs for a socket connection (`TCPConn` or `UDPConn`) working pretty much like `scanner.Scan()` and `writer.WriteString()`. The limitation is however that one can't `Read()` easily in specific units, e.g., one line at a time or one word at a time such as the `scanner.Scan()` does.

5. Submit your PA3

`ssh` to the `140.112.42.221` workstation. At the team account's home directory, create a directory `PA3`. Upload your `PA3.go` to directory `PA3`. Test your `PA3.go` again on the workstation just to make sure it's working as expected.