Programming Assignment #6

Introduction to Computer Networks

The Assignment

`PA5.go` handles clients' file upload requests sequentially. That is, it will complete a file upload before starting another one. If the upload time is long, the waiting time for the clients farther back in line can be unbearably long. What's nice about Go is that it is easy to specify a code block for concurrent execution. That concurrent code block is referred to as a **Goroutine**. As a result, processing of later file upload requests can be started before the earlier file upload requests are completed. Although the completion time as a whole (time to finish all file uploads) might be the same, clients are generally happier to see their requested being started early. In other words, with concurrency, we get better user experience, while the system capacity remains unchanged.

You will turn the file uploading part of your `PA5.go` concurrent in `PA6.go`. This is the specification of the server in `PA5.go`.

(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 connections and goes back to (1)

`PA6.go` will still be a file upload server, but it will be a user-friendly file upload server. In that, you will need to:

(1) Make the code block (2)-(7) concurrent.
(2) Add a 5 second delay between (6) and (7), just to make the delay noticeable.

To prepare you for the task, follow through the 3 examples below.

1. Simple Server with `func`

This example works just the same as `server-loop.go`. The only difference is that a block of the code is casted into a `func` outside of `main()`. Start a file `server-func.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 handleConnection (c net.Conn) {
    reader := bufio.NewReader(c)
    message, errr := reader.ReadString('\n')
    check(errr)
    fmt.Printf("%s", message)

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

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

    i := 1
    for {
        conn, _ := ln.Accept()
        defer conn.Close()

        fmt.Printf("%d ", i)
        handleConnection(conn)
        i++
    }
}
```

`server-func.go` works just like `server-loop.go` in PA5. Test of `server-func.go` is therefore similar. Replace `<your port#>` with the port number assigned to your team. Start the server code first.

```
$ go run server-func.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.

```
$ go run server-func.go
```

```
Launching server...
Hello World!
```

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

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

`server-func.go` will continue to wait for the next client request until forever.

Code walk-through:

- None of the APIs is new. The only difference is just that a block of code in the `for` loop is moved into a function `handleConnection`.

2. Simple Server with `time.Sleep()`

To make the scenario a bit more interesting, we add an extra 5-second delay to the execution of the separated code block. Start a file `server-slow.go` and type up the following code. Copy the `main()` part from `server-func.go`.

```
package main

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



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



func handleConnection (c net.Conn) {
    reader := bufio.NewReader(c)
    message, errr := reader.ReadString('\n')
    check(errr)
    fmt.Printf("%s", message)

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

    time.Sleep(5 * time.Second)
}
```

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

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

Then, run the `client-102.go` code.

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

Run the `client-102.go` code again quickly.

```
$ go run client-102.go
Send a string of 13 bytes
```

You should feel a significant wait here, about 5 seconds, before the 2nd line is printed.

```
Server replies: 13 bytes received
$
```

Code walk-through:

- The `time` package is new. Defined in the package are APIs and variables allowing time manipulation.
- `time.Sleep()` is an API that enforces a sleep time, i.e., adding a pause of a specific duration to the process.
- `time.Second` is a constant that counts the time in seconds.
- What's happening here is that the 1st `go run client-102.go` takes the server into the 5 second wait. When the 2nd `go run client-102.go` is fired, the client's `net.Dial()` needs to wait until the 5-second wait from the 1st fire expires and the server returns to the `for` loop to `ln.Accept()` the client's `net.Dial()`.

3. Simple Server with `Go`

Finally to make the simple server run concurrently (a form of parallelism), start a new file `server-conc.go` and prepend `go` to the `handleConnection` function in the `for` loop.

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

    i := 1
    for {
        conn, _ := ln.Accept()
        defer conn.Close()

        fmt.Printf("%d ", i)
        go handleConnection(conn)
        i++
    }
}
```

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

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

Then, run the `client-102.go` code.

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

Run the `client-102.go` code again quickly.

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

You should not feel any wait here, before the 2nd line is printed.

Code walk-through:

- The only API being new is `go`. It is used to allow multiple instances of `handleConnection` to run concurrently.
- When the 1st `go run client-102.go` is fired, the server invokes a separate process to execute instructions in `handleConnection`. In the meantime, the server is back to the `for` loop waiting to `ln.Accept()` another `net.Dial()`. Therefore, when the 2nd `go run client-102.go` is fired, the client's `net.Dial()` connects real quick (no wait) to the server and another instance of `handleConnection` is invoked.
- To some degree, the 5-second wait at the end of `handleConnection` emulates the server processing time (e.g., doing some computation and/or writing data to a large database). The processing will still occur but it will quietly progress in the background without the client knowing.
- This way, the client gets initial response back from the server quickly. And there appears no wait time for the next client to access the server. Note though the overall completion time at the server side will be the same, with concurrency or not.

4. PA6.go

Again, make sure your `PA6.go` is listening on the port number you are assigned to. To test your `PA6.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 `PA6.go` available here: http://homepage.ntu.edu.tw/~pollyhuang/teach/intro-cn-pa/PA6/PA6. polly's `PA6` is configured to run on port# 11999. To see how polly's `PA6` behaves, you'll need to modify your `PA3.go` to dial to the IP address of the

machine you are running polly's `PA6` and the port# 11999. Cross compare your `PA6.go` to the behavior of polly's `PA6`. If they work the same, you are ready to submit.

5. Submit your PA6

`ssh` to the `140.112.42.221` workstation. At the team account's home directory, create a directory `PA6`. Upload your `PA6.go` to directory `PA6`. Test your `PA6.go` again on the workstation just to make sure it's working as expected.