



Answer Sheet
Hands On – H01

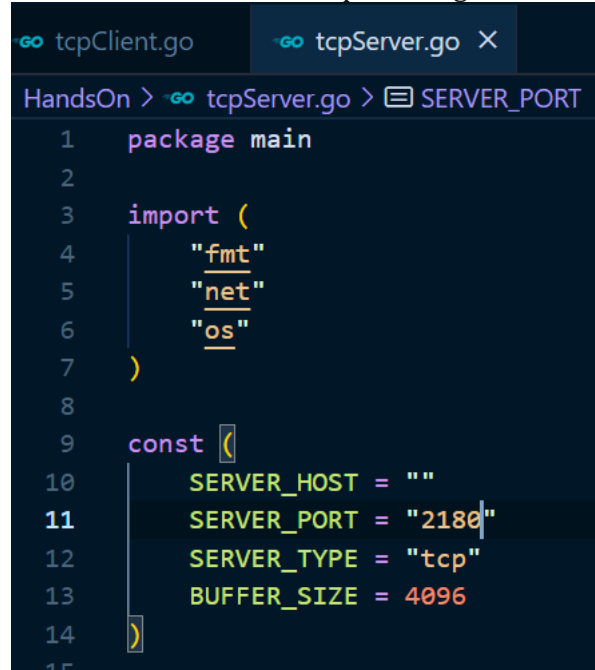
Introduction to Socket Programming

Name : Alvaro Austin
Student ID : 2106752180

[15 Point] Simple TCP Socket

[1] Screenshot of the modified tcpServer.go

Modified code tcpServer.go

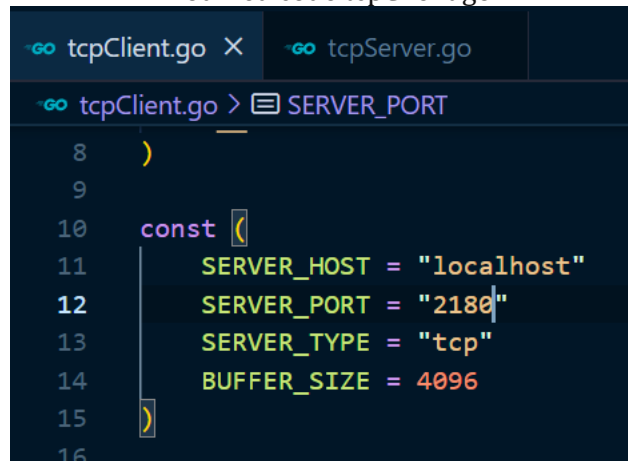


The screenshot shows a code editor with two tabs: 'tcpClient.go' and 'tcpServer.go'. The 'tcpServer.go' tab is active, and the cursor is at the end of line 11. The code defines a package 'main', imports 'fmt', 'net', and 'os', and sets constants for 'SERVER_HOST' (empty string), 'SERVER_PORT' ('2180'), 'SERVER_TYPE' ('tcp'), and 'BUFFER_SIZE' (4096).

```
1 package main
2
3 import (
4     "fmt"
5     "net"
6     "os"
7 )
8
9 const (
10     SERVER_HOST = ""
11     SERVER_PORT = "2180"
12     SERVER_TYPE = "tcp"
13     BUFFER_SIZE = 4096
14 )
15
```

[1] Screenshot of the modified tcpClient.go

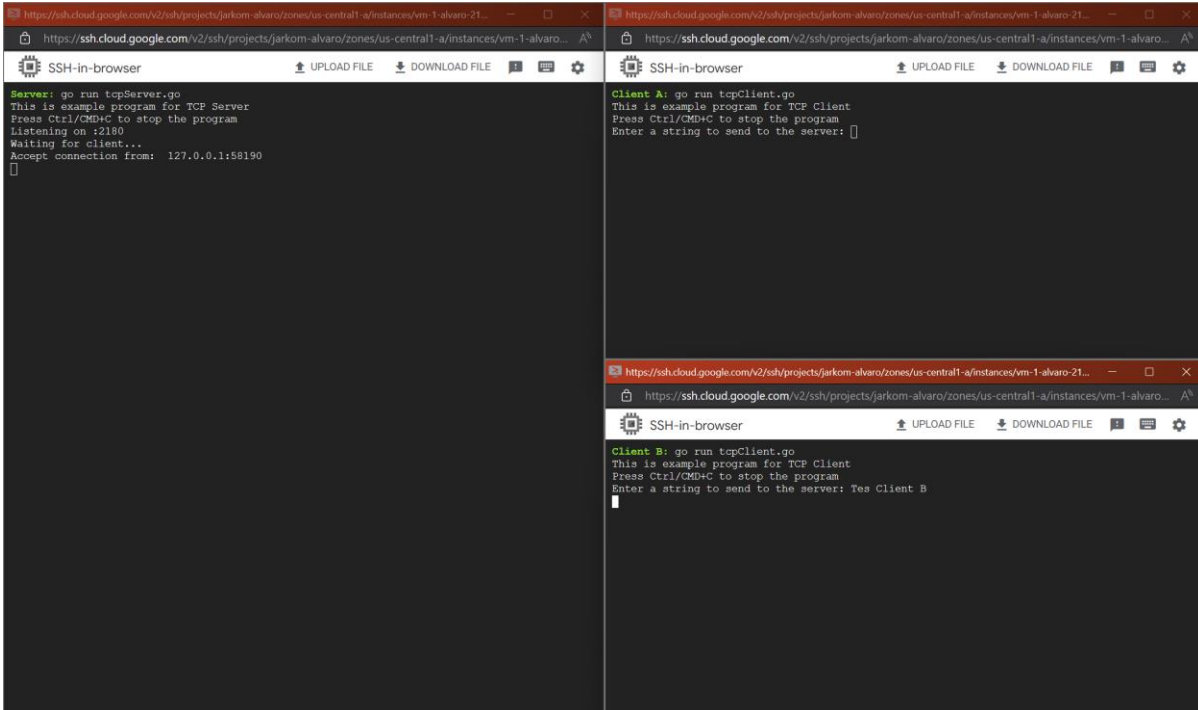
Modified code tcpClient.go



The screenshot shows a code editor with two tabs: 'tcpClient.go' and 'tcpServer.go'. The 'tcpClient.go' tab is active, and the cursor is at the end of line 12. The code defines a package 'main', imports 'fmt', 'net', and 'os', and sets constants for 'SERVER_HOST' ('localhost'), 'SERVER_PORT' ('2180'), 'SERVER_TYPE' ('tcp'), and 'BUFFER_SIZE' (4096).

```
8 )
9
10 const (
11     SERVER_HOST = "localhost"
12     SERVER_PORT = "2180"
13     SERVER_TYPE = "tcp"
14     BUFFER_SIZE = 4096
15 )
16
```

[5] Screenshot for activity number 3



[8] Answer the result for activity number 4

Things that I could find from the observation has 3 main things:

1. Changing the SERVER_PORT will change which PORT it will run on. It will be important to have a fixed PORT because inconsistent PORT will cause client not able to make a connection with our server in TCP.
2. In our server, the TCP server, first thing it does is to accept connection on the client that ask for the connection first, in this case its Client A. When Client B ran, and ask for connection in TCP, it does not process (doesn't mean it's rejected) connection from B.
3. Continuing from step 2, main reason for that to happened is that TCP uses a **three-way handshake** to establish connection between 2 hosts. This handshake involves steps like SYN (Synchronize) and ACK (acknowledge). When the connection is established, TCP closes connection that open, then wait until a connection become available before incoming request could be processed.

Basically what happened is that when connection from client B try to make input, it will keep waiting until connection from client A is finished (by stopping or inputting a result). This happened because TCP works on a single thread so only one connection can be established. There are connection agreement in TCP.

So what happened is that TCP create a new socket for server to communicate with the client. TCP also provides reliable, byte stream transter ("pipe").
Bonus: TCP will immediately close down when the server wasn't ran.

[15 Point] Simple UDP Socket

[1] Screenshot of the modified udpServer.go

Modified code udpServer.go



```
1 package main
2
3 import (
4     "fmt"
5     "net"
6     "os"
7 )
8
9 const (
10     SERVER_HOST = ""
11     SERVER_PORT = "2180"
12     SERVER_TYPE = "udp"
13     BUFFER_SIZE = 4096
14 )
15
16 func main() {
17     fmt.Println("This is example program for UDP Server")
18     fmt.Println("Press Ctrl/CMD+C to stop the program")
19 }
```

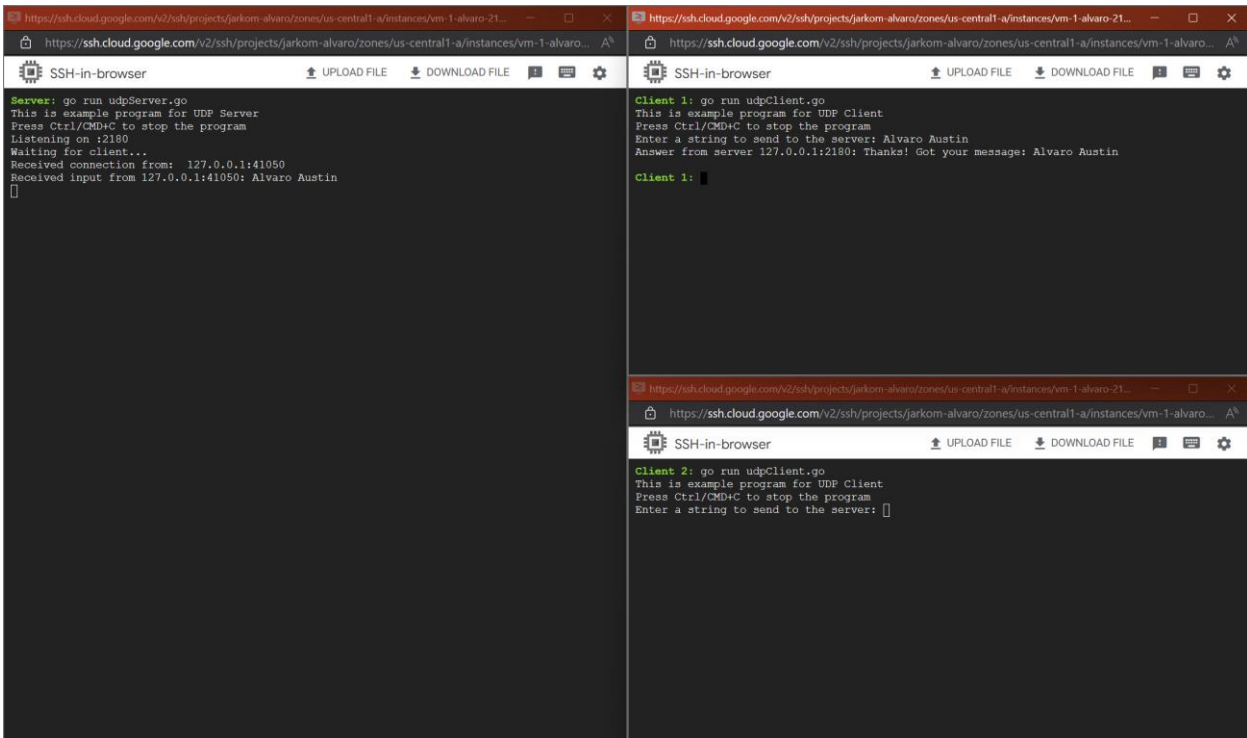
[1] Screenshot of the modified udpClient.go

Modified code udpClient.go

```
udpClient.go X  udpServer.go
HandsOn > udpClient.go > SERVER_PORT

1  package main
2
3  import (
4      "bufio"
5      "fmt"
6      "net"
7      "os"
8  )
9
10 const {
11     SERVER_HOST = ""
12     SERVER_PORT = "2180"
13     SERVER_TYPE = "udp"
14     BUFFER_SIZE = 4096
15 }
16
17 func main() {
18     fmt.Println("This is example program for UDP Client")
19     fmt.Println("Press Ctrl/CMD+C to stop the program")
20 }
```

[5] Screenshot for activity number 3



[8] Answer the result for activity number 4

There are some observation that I find while trying to do the UDP assignment:

1. As I said previously, SERVER_PORT will decide which port the server is gonna run on.
2. Using "localhost", is the same as using "127.0.0.1" because they are equivalent.
3. In UDP socket programming, when one client (Client B) tries to access the server and another client (Client A) tries to access the server as well, it did not need to wait for the server to finish the first client process. In this case, when Client A input the text, it immediately processed by the server without the need for client B to be finished.

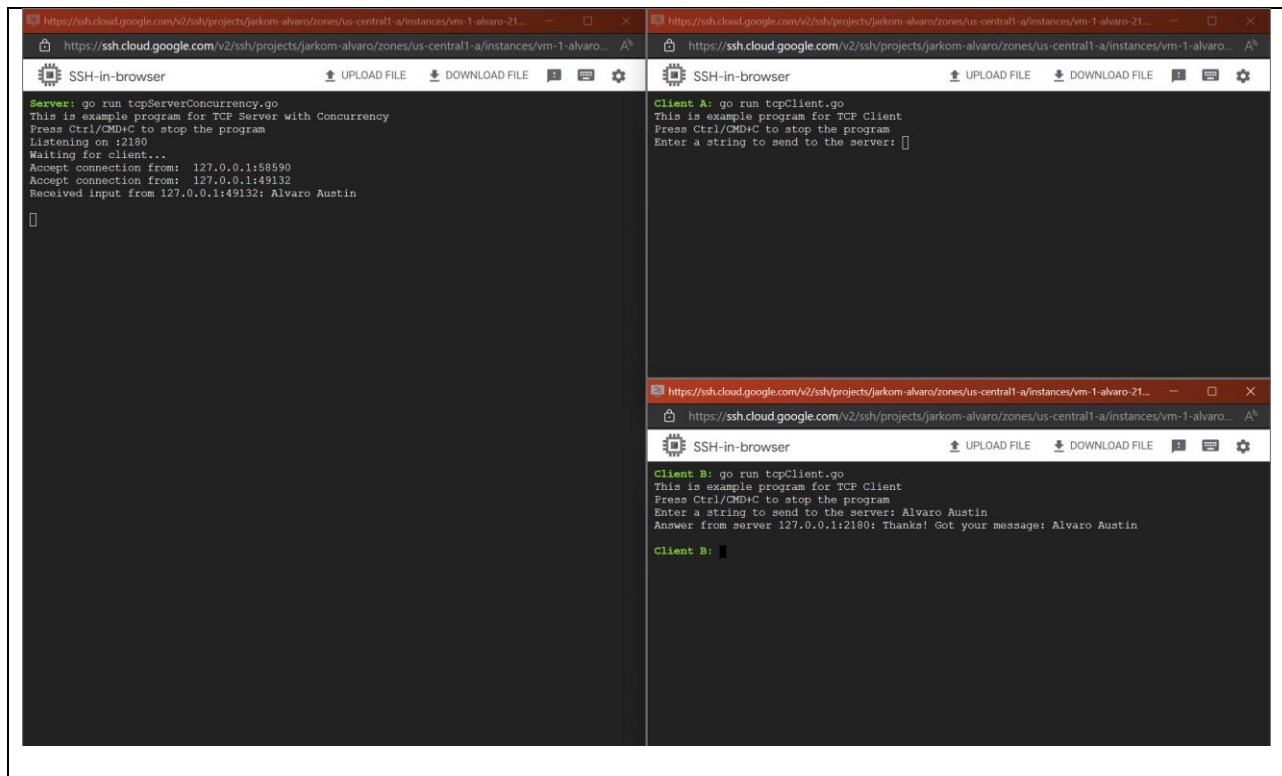
For Client A, UDP also sent back response of being connected after they input the text different from TCP where TCP sent back response of being connected when TCP client ran. This means that UDP doesn't use a three way handshake because they are connectionless protocol, which means, they do not maintain a dedicated connection towards a server. This is why UDP is often referred as "fire-and-forget" because there are no connection agreement.

Hence, UDP transmitted data in an unreliable way between the client to the server in a group of bytes ("datagrams")

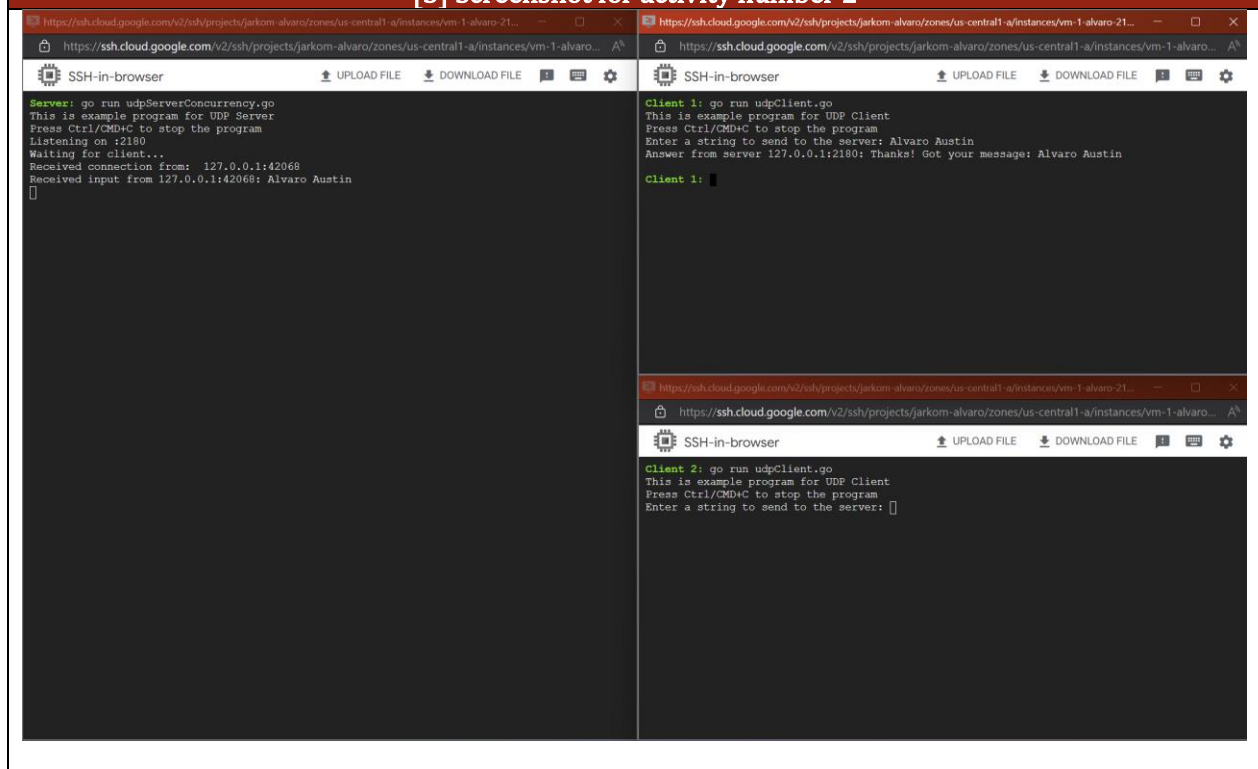
I also found an intriguing thing about UDP while trying to connect to a server. If I ran the udpClient, it did not immediately close down. This is proving that UDP doesn't have to make a connection agreement to the server.

[30 Point] Sockets with Concurrency

[3] Screenshot for activity number 1



[3] Screenshot for activity number 2



[8] Answer the result for activity number 3a

There are difference that I found during doing activity 3 in TCP. I noticed that if we tried to make 2 connection using TCP, unlike the first activity, when we tried to connect to second client (in this case Client B), it immediately received the connection from second client without the need of waiting the first client connection to be finished.

What happened in this case is TCP creates multiple sockets to handle concurrent connections. Each sockets will take request and give response without the need of depending to the other thread. This will lead to multiple request able to be process on TCP. In this case, when a new connection was made, a new socket is also made to handle another request. This new socket will be used to handle communication with the client.

In the code in **tcpServerConcurrency.go**, each socket will be handled by a goroutine to communicate with the client.

[8] Answer the result for activity number 3b

In this activity, we can't see anything different than the second activity. UDP will work as usual, if the we ran client 2 for udpClient, after that we ran client 1 and input the text. It will return exactly the same as activity 2 did. It will also give the same behaviour and the same output message as the activity 2.

The reason for this happened because UDP doesn't make any connection between client and server, which is there is no handshaking. Because of that, we don't have know whose going first in UDP, because whoever write to the server first, will be the one that were processed first.

But this doesn't mean that concurrency is useless, concurrency for UDP is still fine because UDP still uses single thread so if there was double input on the same time it could potentially dropped the packets.

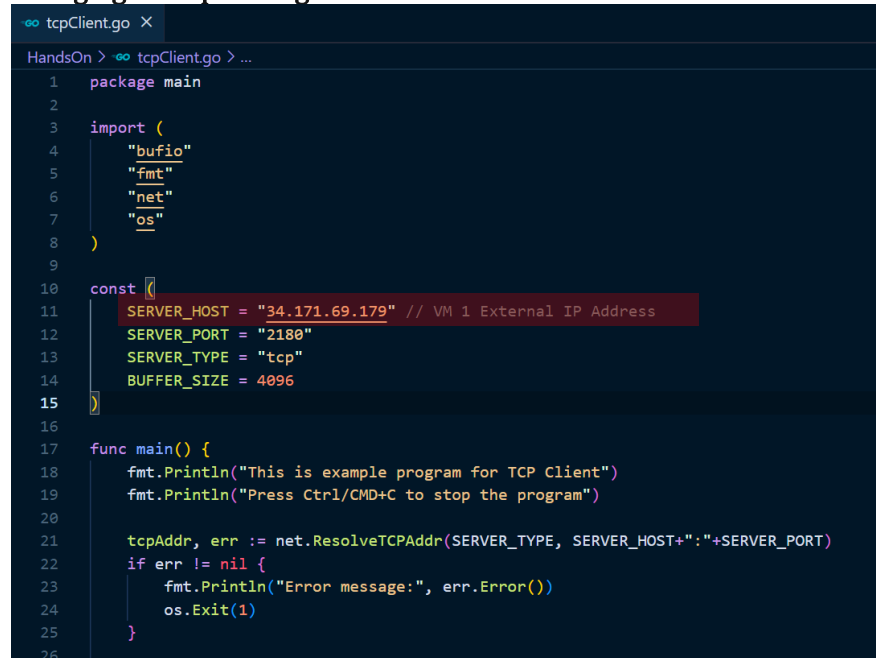
[8] Answer the result for activity number 3c

In this case I will chose to use TCP socket as the type. The reason for this is that TCP is known as reliable connection protocol that ensure the safety of the connection. What I meant by the safety is that, ensures the data that was received is the same order as the one that was sent. TCP also gives benefit such as error checking and correction that is good when transmitting sensitive information such as password. The connection then could be closed once authentication is successful to prevent unnecessary/dangerous request. Based on what I read on Internet, data that are transmitted using TCP socket could be encrypted, basically means that data that are being sent are secured.

[40 Point] Run the Servers and Clients Programs on Different Machines

[20] Screenshot for activity number 1

Changing the tcpClient.go



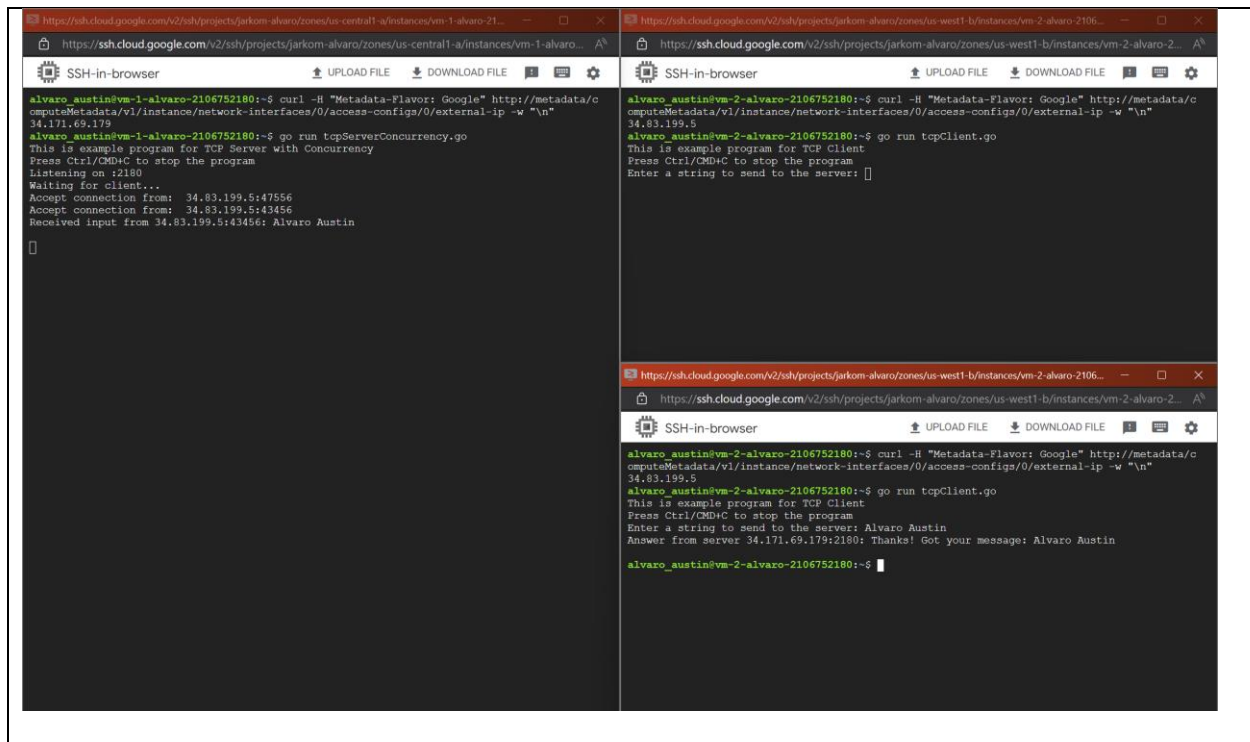
```
1 package main
2
3 import (
4     "bufio"
5     "fmt"
6     "net"
7     "os"
8 )
9
10 const {
11     SERVER_HOST = "34.171.69.179" // VM 1 External IP Address
12     SERVER_PORT = "2180"
13     SERVER_TYPE = "tcp"
14     BUFFER_SIZE = 4096
15 }
16
17 func main() {
18     fmt.Println("This is example program for TCP Client")
19     fmt.Println("Press Ctrl/CMD+C to stop the program")
20
21     tcpAddr, err := net.ResolveTCPAddr(SERVER_TYPE, SERVER_HOST+": "+SERVER_PORT)
22     if err != nil {
23         fmt.Println("Error message:", err.Error())
24         os.Exit(1)
25     }
26 }
```

Changing the udpClient.go

```
udpClient.go X
HandsOn > udpClient.go > SERVER_TYPE

1  package main
2
3  import (
4      "bufio"
5      "fmt"
6      "net"
7      "os"
8  )
9
10 const {
11     SERVER_HOST = "34.171.69.179" // VM 1 External IP Address
12     SERVER_PORT = "2180"
13     SERVER_TYPE = "udp"
14     BUFFER_SIZE = 4096
15 }
16
17 func main() {
18     fmt.Println("This is example program for UDP Client")
19     fmt.Println("Press Ctrl/CMD+C to stop the program")
20 }
```

We don't have to change the configuration in TCP&UDP Server with concurrency because the program will work as intended even though we didn't change anything in TCP & UDP, this happened because the server accept request from the client without the need to authenticate/restricted us. Usually in a web server we could allow incoming request based on the hostname of the client. But in this case, we don't have to change anything because it doesn't have any restrictions on which client could request to the server



[10] Screenshot for activity number 3

