

# WES 237A: Introduction to Embedded System Design (Winter 2024)

## Lab 4: Network Communication

Due: 2/19/2024 11:59pm

Ricardo Lizarraga, [rlizarraga0@gmail.com](mailto:rlizarraga0@gmail.com)  
619.252.4157

PID: A69028483  
[https://github.com/RiLizarraga/WES237A\\_Lab4](https://github.com/RiLizarraga/WES237A_Lab4)

In order to report and reflect on your WES 237A labs, please complete this Post-Lab report by the end of the weekend by submitting the following 2 parts:

- Upload your lab 4 report composed by a single PDF that includes your in-lab answers to the bolded questions in the Google Doc Lab and your Jupyter Notebook code. You could either scan your written copy, or simply type your answer in this Google Doc. **However, please make sure your responses are readable.**
- Answer two short essay-like questions on your Lab experience.

All responses should be submitted to Canvas. Please also be sure to push your code to your git repo as well.

### Locating IP Addresses of Devices in your Network

1. Open a serial connection to your PYNQ board (see Lab3 if you forgot)

 COM5 - PuTTY

```
xilinx@pynq:~$
```

2. Connect the PYNQ board to the network switch over ethernet.
3. Run '\$ ifconfig'. This is the *Interface Configuration* command and will tell you the different interfaces on your PYNQ board.

```

root@pynq:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::200:5ff:fe6b:33a prefixlen 64 scopeid 0x20<link>
    ether 00:00:05:6b:03:3a txqueuelen 1000 (Ethernet)
    RX packets 202 bytes 24662 (24.6 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 365 bytes 105114 (105.1 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 36 base 0xb000

eth0:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.2.99 netmask 255.255.255.0 broadcast 192.168.2.255
    ether 00:00:05:6b:03:3a txqueuelen 1000 (Ethernet)
    device interrupt 36 base 0xb000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 3200 bytes 234918 (234.9 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 3200 bytes 234918 (234.9 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

- a. **How many ipv4 addresses are assigned to the board? What is the ipv4 address assigned to the 'eth0' or ethernet interface? What is the netmask of this address?**
  - i. Note: 'eth0: 1' or 'usb0' is a virtual interface through the USB cable. This assigns your board an IP address over USB. This is a static IP address so you can always reach your board from this IP address over USB.

Physical H/W interface	Ip address & type
Eth0	inet6 fe80::200:5ff:fe6b:33a
	192.168.2.99 {IP4} netmask 255.255.255.0
	127.0.0.1 {local loop}

4. Use a lab computer or your personal computer with either of the following setups **{SKIPPED per TA instruction}**
  - a. Connected to the *WES237A\_Private* wifi network (passwd: X!!!nxWes237A)
  - b. Connected directly to the network switch through ethernet cable
5. Open a command prompt and run '\$ **ipconfig**' on windows and '\$ ifconfig' on MAC/linux (it may take a second to connect so wait a minute and then run the command)
  - a. **How many ipv4 addresses are assigned to this machine?**

Total 3:  
192.168.2.11  
192.168.56.1  
100.81.32.26

- b. **What ipv4 address has the same netmask as the PYNQ board?**

192.168.2.11 {my PC} has the same netmask as my pynq board 255.255.255.0

```

C:\Users\rliz0>ipconfig

Windows IP Configuration

Ethernet adapter Ethernet 4:

    Connection-specific DNS Suffix  . : 
    Link-local IPv6 Address . . . . . : fe80::283b:3bcc:d888:fff1%11
    IPv4 Address. . . . . : 192.168.2.11
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 

Ethernet adapter Ethernet 5:

    Connection-specific DNS Suffix  . : 
    Link-local IPv6 Address . . . . . : fe80::77e9:45f4:b0aa:9930%12
    IPv4 Address. . . . . : 192.168.56.1
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 

Wireless LAN adapter Local Area Connection* 3:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . : 

Wireless LAN adapter Local Area Connection* 4:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . : 

Wireless LAN adapter Wi-Fi:

    Connection-specific DNS Suffix  . : ucsd.edu
    IPv6 Address. . . . . : 2607:f720:f00:4042::1:eafe
    Link-local IPv6 Address . . . . . : fe80::3bf6:696d:7c5c:dbcf%14
    IPv4 Address. . . . . : 100.81.32.26
    Subnet Mask . . . . . : 255.255.240.0
    Default Gateway . . . . . : 100.81.32.1

Ethernet adapter Bluetooth Network Connection:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . : 

```

6. Right now, your local machine and your PYNQ board form a network! However, we're more interested in networking two PYNQ boards together rather than your local machine and your PYNQ board. Luckily, every device hooked up to the switch, is assigned an IP address on this network. That means we can communicate with any other board in the class. **Below, compile all the IP addresses of the PYNQ boards in your group.**

Only one PYNQ board (we didn't connect to the class wifi)

7. To access your PYNQ board jupyter notebooks, go to <PYNQ-IP>:9090

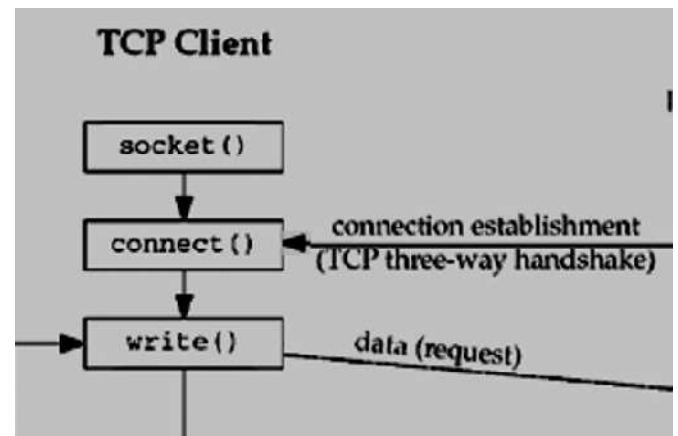
## PYNQ-PYNQ Communication with Python

1. Here we're going to implement basic message sending functionality in python from one PYNQ board to another.
2. Download [`sockets` example.ipynb](#)
3. Go through and complete the code. Answer the following questions.
  - a. What does `socket.SOCK_STREAM` mean (hint: search the documentation link in the notebook)?

The `socket.SOCK_STREAM` constant is used to specify that a socket should use the TCP protocol. TCP is a connection-oriented protocol, which means that a reliable connection is established between the two parties before data is transmitted. This makes TCP ideal for applications such as web browsing and file transfer, where it is important to ensure that all data is received correctly. In contrast, the `socket.SOCK_DGRAM` constant is used to specify that a socket should use the UDP protocol. UDP is a connectionless protocol, which means that data is sent without establishing a connection first. This makes UDP ideal for applications such as streaming video and audio, where it is more important to minimize latency than to ensure that all data is received correctly.

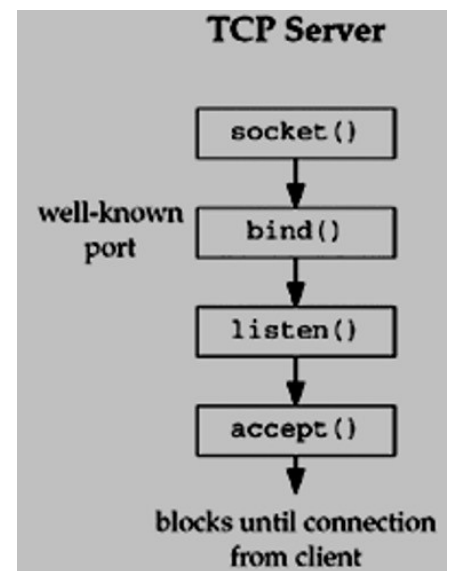
- b. What is the order of operations for starting a client socket and sending a message?

Server(s) is listening for client(s) to connect, after the connections has been accepted by the Server, then the sending & receiving information can start



- c. What is the order of operations for starting a server socket and receiving a message?

The Server Listening has to be ready, before accepting client connection, then after connection is done, messages can be send and received



## Wireshark

1. On your local machine (or lab machine), install [Wireshark](#)
2. Open the Firewall and Network Protection
3. Click 'Allow an app through firewall'
4. Click 'Change Settings'
5. Scroll down to 'Python'
6. Select all 'Python' applications and all 'Public' boxes for each 'Python'

<input checked="" type="checkbox"/> Python	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	No
<input checked="" type="checkbox"/> Python	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	No

7. Open the program 'IDLE (Python 3.7 64-bit)'
8. Click File->New File and paste the following code (**Check for tab v space errors when copying and pasting**)

```
import socket
import time
import signal
import sys

def run_program():
    sock_l = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock_l.bind(('0.0.0.0', 12345))
    sock_l.listen()
    print('Waiting for connection')
    conn, addr = sock_l.accept()
    print('Connected')
    with conn:
        data = conn.recv(1024)
        print(data.decode())

if __name__ == '__main__':
    original_sigint = signal.getsignal(signal.SIGINT)
    signal.signal(signal.SIGINT, exit)
    run_program()
```

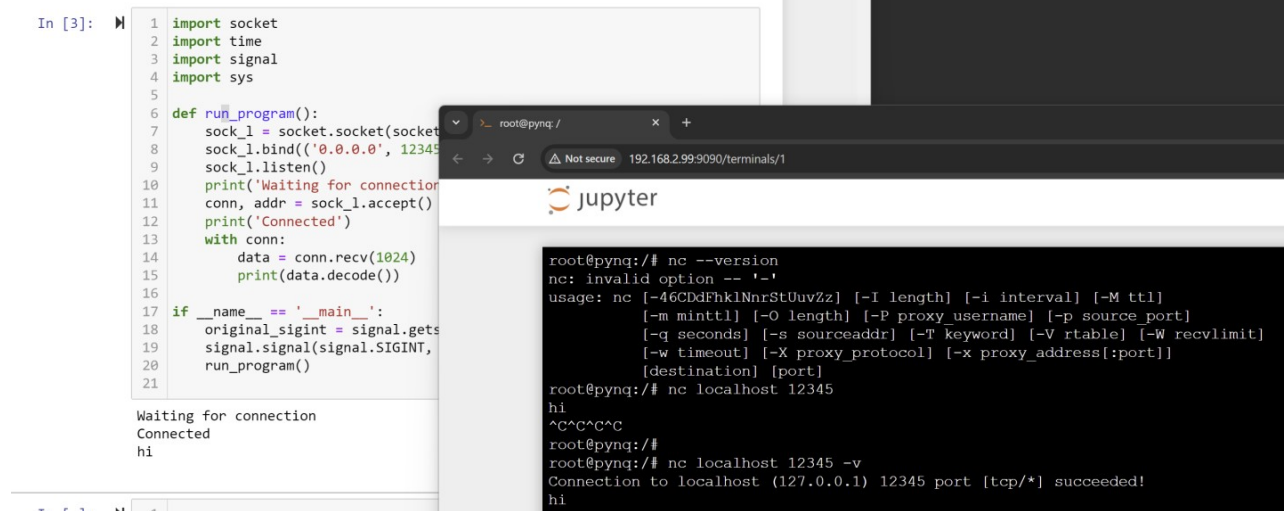
9. Save the file, then select 'Run -> Run Module'. This is a slight variation to your server. It is waiting on port 12345 on the local lab machine.

```
In [*]: ▶ 1 import socket
          2 import time
          3 import signal
          4 import sys
          5
          6 def run_program():
          7     sock_l = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
          8     sock_l.bind(('0.0.0.0', 12345))
          9     sock_l.listen()
         10     print('Waiting for connection')
         11     conn, addr = sock_l.accept()
         12     print('Connected')
         13     with conn:
         14         data = conn.recv(1024)
         15         print(data.decode())
         16
         17 if __name__ == '__main__':
         18     original_sigint = signal.getsignal(signal.SIGINT)
         19     signal.signal(signal.SIGINT, exit)
         20     run_program()
         21
```

Waiting for connection

10. From your PYNQ board, connect your client to

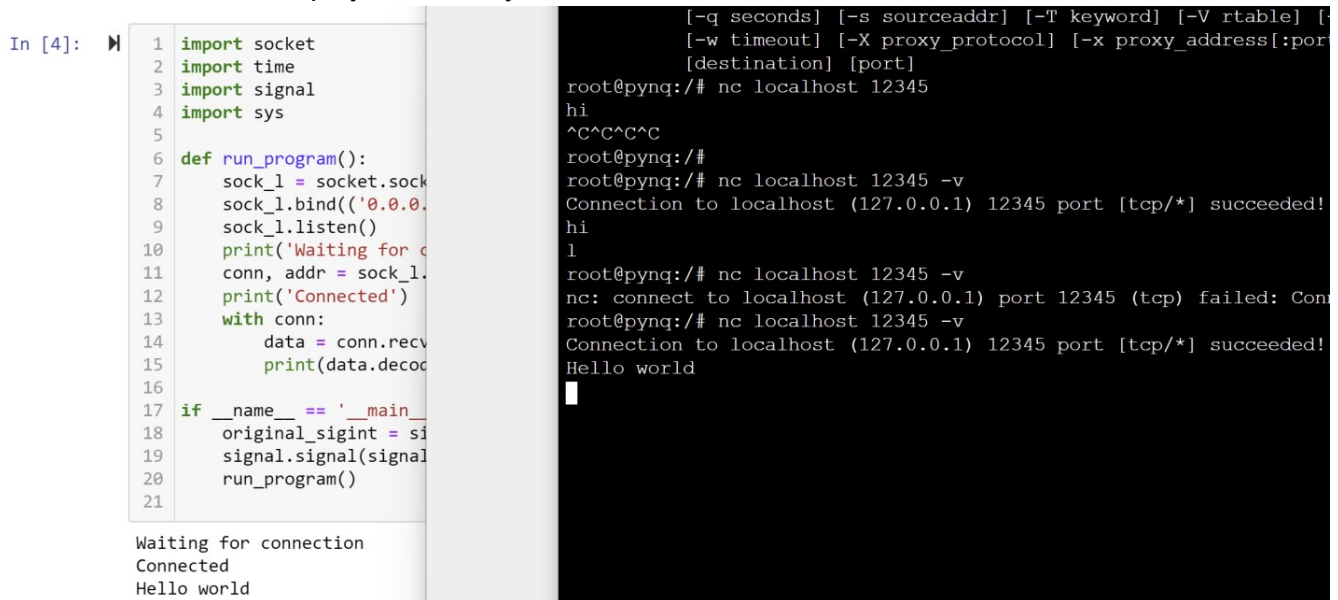
- a. Ip: local lab IP
- b. Port: 12345



```
In [3]: 1 import socket
2 import time
3 import signal
4 import sys
5
6 def run_program():
7     sock_l = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8     sock_l.bind(('0.0.0.0', 12345))
9     sock_l.listen()
10    print('Waiting for connection')
11    conn, addr = sock_l.accept()
12    print('Connected')
13    with conn:
14        data = conn.recv(1024)
15        print(data.decode())
16
17 if __name__ == '__main__':
18     original_sigint = signal.getsignal(signal.SIGINT)
19     signal.signal(signal.SIGINT, run_program)
20
21
Waiting for connection
Connected
hi
```

11. Send the message “Hello world\n”

12. You should see it displayed in the Python terminal



```
In [4]: 1 import socket
2 import time
3 import signal
4 import sys
5
6 def run_program():
7     sock_l = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8     sock_l.bind(('0.0.0.0', 12345))
9     sock_l.listen()
10    print('Waiting for connection')
11    conn, addr = sock_l.accept()
12    print('Connected')
13    with conn:
14        data = conn.recv(1024)
15        print(data.decode())
16
17 if __name__ == '__main__':
18     original_sigint = signal.getsignal(signal.SIGINT)
19     signal.signal(signal.SIGINT, run_program)
20
21
Waiting for connection
Connected
Hello world
```

13. Now open Wireshark

14. Double click ‘Wi-Fi’ or ‘Ethernet’ depending on how you connected to the network. You’re now capturing a trace of the network which is only between your machine and the PYNQ board through the router. Look at a few of the traces. Notice which are between your PYNQ board and the local machine (check the IP addresses) and which involve the router. There will only be a difference if you also connected the PYNQ board directly to your local machine.





15. Where it says 'Apply a display filter' at the top, type 'tcp'

Wireshark interface showing a packet capture with a display filter 'tcp' applied. The packet list shows 43 TCP packets. The packet details pane shows the structure of packet 8: Ethernet II, Internet Protocol Version 4, and Transmission Control Protocol.

No.	Time	Source	Destination	Protocol	Length	Info
3	6.806165	192.168.2.99	192.168.2.11	TCP	60	9090 → 63672 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
4	6.806492	192.168.2.11	192.168.2.99	TCP	60	63672 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8195 Len=6
5	6.806939	192.168.2.99	192.168.2.11	TCP	60	9090 → 63672 [ACK] Seq=3 Ack=7 Win=1002 Len=0
8	17.444768	192.168.2.99	192.168.2.11	TCP	60	9090 → 54707 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
9	17.445039	192.168.2.11	192.168.2.99	TCP	60	54707 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8195 Len=6
10	17.445488	192.168.2.99	192.168.2.11	TCP	60	9090 → 54707 [ACK] Seq=3 Ack=7 Win=1002 Len=0
11	18.573713	192.168.2.99	192.168.2.11	TCP	60	9090 → 59420 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
12	18.573947	192.168.2.11	192.168.2.99	TCP	60	59420 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8190 Len=6
13	18.574413	192.168.2.99	192.168.2.11	TCP	60	9090 → 59420 [ACK] Seq=3 Ack=7 Win=1002 Len=0
16	25.542133	192.168.2.99	192.168.2.11	TCP	60	9090 → 54709 [PSH, ACK] Seq=1 Ack=1 Win=1002 Len=2
17	25.542464	192.168.2.11	192.168.2.99	TCP	60	54709 → 9090 [PSH, ACK] Seq=1 Ack=3 Win=8192 Len=6
18	25.542887	192.168.2.99	192.168.2.11	TCP	60	9090 → 54709 [ACK] Seq=3 Ack=7 Win=1002 Len=0
22	36.805707	192.168.2.99	192.168.2.11	TCP	60	9090 → 63672 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
23	36.805938	192.168.2.11	192.168.2.99	TCP	60	63672 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8195 Len=6
24	36.806366	192.168.2.99	192.168.2.11	TCP	60	9090 → 63672 [ACK] Seq=5 Ack=13 Win=1002 Len=0
27	47.445176	192.168.2.99	192.168.2.11	TCP	60	9090 → 54707 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
28	47.445523	192.168.2.11	192.168.2.99	TCP	60	54707 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8195 Len=6
29	47.445877	192.168.2.99	192.168.2.11	TCP	60	9090 → 54707 [ACK] Seq=5 Ack=13 Win=1002 Len=0
31	48.573207	192.168.2.99	192.168.2.11	TCP	60	9090 → 59420 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
32	48.573470	192.168.2.11	192.168.2.99	TCP	60	59420 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8190 Len=6
33	48.573926	192.168.2.99	192.168.2.11	TCP	60	9090 → 59420 [ACK] Seq=5 Ack=13 Win=1002 Len=0
36	55.542599	192.168.2.99	192.168.2.11	TCP	60	9090 → 54709 [PSH, ACK] Seq=3 Ack=7 Win=1002 Len=2
37	55.542816	192.168.2.11	192.168.2.99	TCP	60	54709 → 9090 [PSH, ACK] Seq=7 Ack=5 Win=8192 Len=6
38	55.543219	192.168.2.99	192.168.2.11	TCP	60	9090 → 54709 [ACK] Seq=5 Ack=13 Win=1002 Len=0
41	66.806192	192.168.2.99	192.168.2.11	TCP	60	9090 → 63672 [PSH, ACK] Seq=5 Ack=13 Win=1002 Len=2
42	66.806453	192.168.2.11	192.168.2.99	TCP	60	63672 → 9090 [PSH, ACK] Seq=13 Ack=7 Win=8195 Len=6
43	66.806903	192.168.2.99	192.168.2.11	TCP	60	9090 → 63672 [ACK] Seq=7 Ack=19 Win=1002 Len=0

Frame 8: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0  
 Ethernet II, Src: Xerox\_6b:03:3a (00:00:05:6b:03:3a), Dst: Realtek\_88:00:00:00:00:00  
 Internet Protocol Version 4, Src: 192.168.2.99, Dst: 192.168.2.11  
 Transmission Control Protocol, Src Port: 9090, Dst Port: 54707, Seq: 9090, Len: 2  
 Data (2 bytes)

16. Repeat steps 9-11

17. Check the packet trace for any changes

a. What is the sequence number of the TCP SYN segment that is used to initiate the TCP connection between the PYNQ board and the local machine?

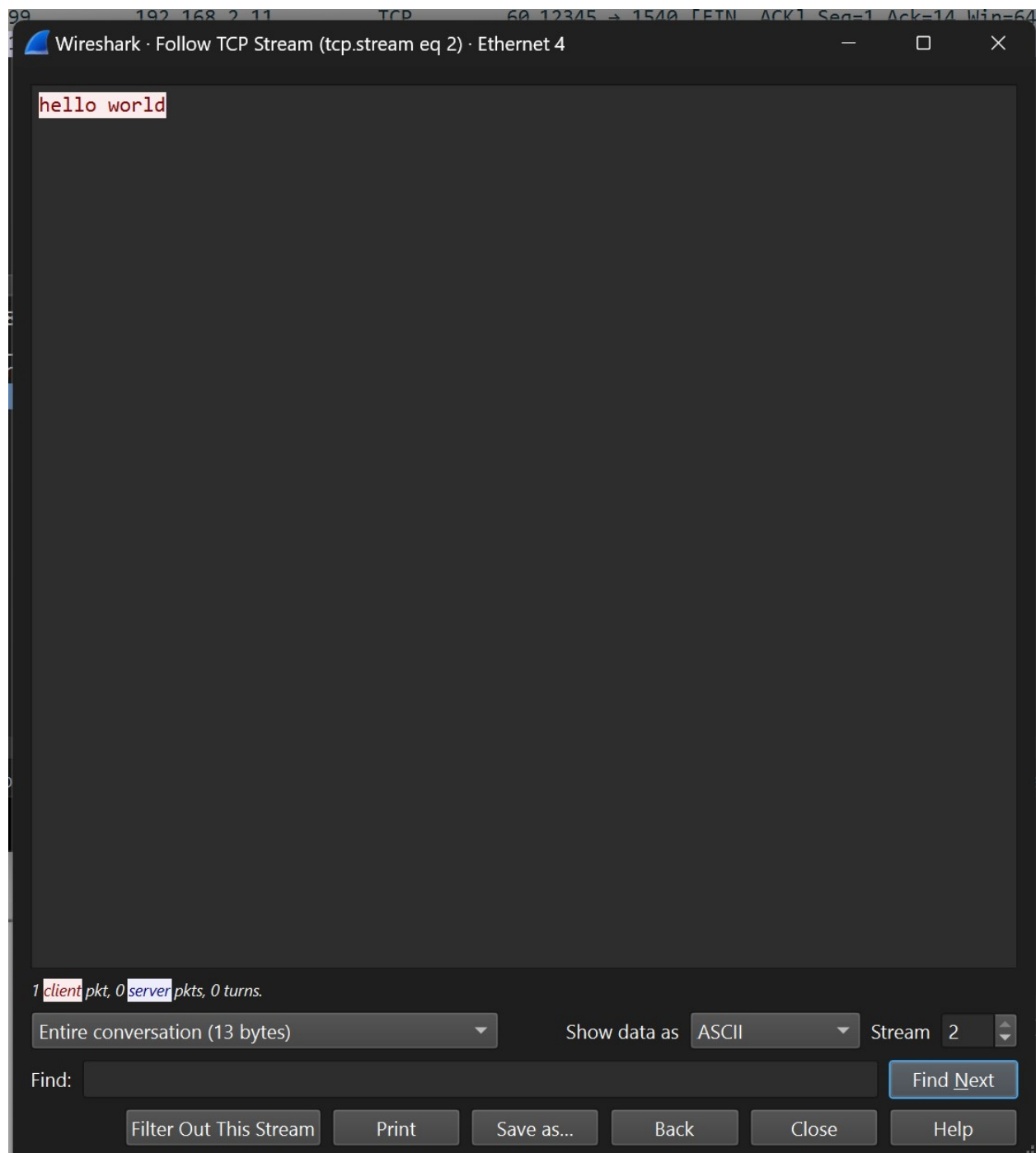
18. Seq = 0

19. 21 21.249819 192.168.2.11 192.168.2.99 TCP 66 1540 → 12345 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK\_PERM

a. What is it in the segment that identifies the segment as a SYN segment?

[SYN]  
[SYN] Seq=0

b. Right click this trace and select 'Follow->TCP Stream'



c. Repeat a few times with different messages. Describe what's happening in the 5-10 steps of the TCP sequence for this communication. You can refresh your TCP flags [here](#).

21	21.249819	192.168.2.11	192.168.2.99	TCP	66	1540 → 12345 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
22	21.250168	192.168.2.99	192.168.2.11	TCP	66	12345 → 1540 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=64
23	21.250249	192.168.2.11	192.168.2.99	TCP	54	1540 → 12345 [ACK] Seq=1 Ack=1 Win=262656 Len=0
24	21.467884	192.168.2.99	192.168.2.11	TCP	684	9090 → 1417 [PSH, ACK] Seq=2409 Ack=917 Win=1002 Len=630
25	21.507714	192.168.2.11	192.168.2.99	TCP	54	1417 → 9090 [ACK] Seq=917 Ack=3039 Win=8190 Len=0
27	26.289364	192.168.2.11	192.168.2.99	TCP	67	1540 → 12345 [PSH, ACK] Seq=1 Ack=1 Win=262656 Len=13
28	26.289708	192.168.2.99	192.168.2.11	TCP	60	12345 → 1540 [ACK] Seq=1 Ack=14 Win=64256 Len=0
29	26.290424	192.168.2.99	192.168.2.11	TCP	60	12345 → 1540 [FIN, ACK] Seq=1 Ack=14 Win=64256 Len=0
30	26.290450	192.168.2.11	192.168.2.99	TCP	54	1540 → 12345 [ACK] Seq=14 Ack=2 Win=262656 Len=0
32	26.315670	192.168.2.99	192.168.2.11	TCP	690	9090 → 1417 [PSH, ACK] Seq=3039 Ack=917 Win=1002 Len=636
33	26.359843	192.168.2.11	192.168.2.99	TCP	54	1417 → 9090 [ACK] Seq=917 Ack=3675 Win=8195 Len=0
34	26.360338	192.168.2.99	192.168.2.11	TCP	1482	9090 → 1417 [PSH, ACK] Seq=3675 Ack=917 Win=1002 Len=1428
35	26.406604	192.168.2.11	192.168.2.99	TCP	54	1417 → 9090 [ACK] Seq=917 Ack=5103 Win=8189 Len=0



```

[SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
[SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM WS=64
[ACK] Seq=1 Ack=1 Win=262656 Len=0
[PSH, ACK] Seq=2409 Ack=917 Win=1002 Len=630
[ACK] Seq=917 Ack=3039 Win=8190 Len=0
r - Transaction ID 0xaef18649
[PSH, ACK] Seq=1 Ack=1 Win=262656 Len=13
[ACK] Seq=1 Ack=14 Win=64256 Len=0
[FIN, ACK] Seq=1 Ack=14 Win=64256 Len=0

```

<ul style="list-style-type: none"> <li>• [SYN] Len =0</li> <li>• [SYN, ACK] Len =0</li> <li>• [ACK] Len =0</li> <li>• [PSH, ACK] Len =0</li> <li>• [ACK] Len =0</li> <li>• [PSH, ACK] Len =13</li> <li>• [ACK] Len =0</li> <li>• [FIN, ACK] Len =0</li> </ul>	<ul style="list-style-type: none"> <li>• Initiate connection</li> <li>• Packet(s) confirmation received</li> <li>• Packet(s) confirmation received</li> <li>• Incoming un-buffered data0</li> <li>• Packet(s) confirmation received</li> <li>• Len =13 "Hello world \n"</li> <li>• Packet(s) confirmation received</li> <li>• Both the sender &amp; receiver send the FIN packets to gracefully terminate the connection</li> </ul>
---	---

<https://www.howtouselinux.com/post/tcp-flags>

### TCP Flags List

- **SYN** Packets that are used to initiate a connection.
- **ACK** Packets that are used to confirm that the data packets have been received, also used to confirm the initiation request and tear down requests
- **RST** Signify the connection is down or maybe the service is not accepting the requests
- **FIN** Indicate that the connection is being torn down. Both the sender and receiver send the FIN packets to gracefully terminate the connection
- **PSH** Indicate that the incoming data should be passed on directly to the application instead of getting buffered
- **URG** Indicate that the data that the packet is carrying should be processed immediately by the TCP stack. It can be used to provide out-of-band data transfer, such as signaling that a message is urgent and should be delivered before other data.

Here are the numbers which match with the corresponding TCP flags.

Flag	Decimal Value
URG	32
ACK	16
PSH	8
RST	4
SYN	2
FIN	1