

A Simple Sender and Receiver Application

Muhammad Talha Khan (46281174)
CS-230 Distributed Computer Systems

Code & Explanation

The code for this assignment was written in Python3. The code consists of 2 files only named **sender.py** and **receiver.py**. A high level view of the implementation can be seen below in the Figure below. For further clarification I have also added code snippets. Most of the code is explained with comments used inside of the code snippets.

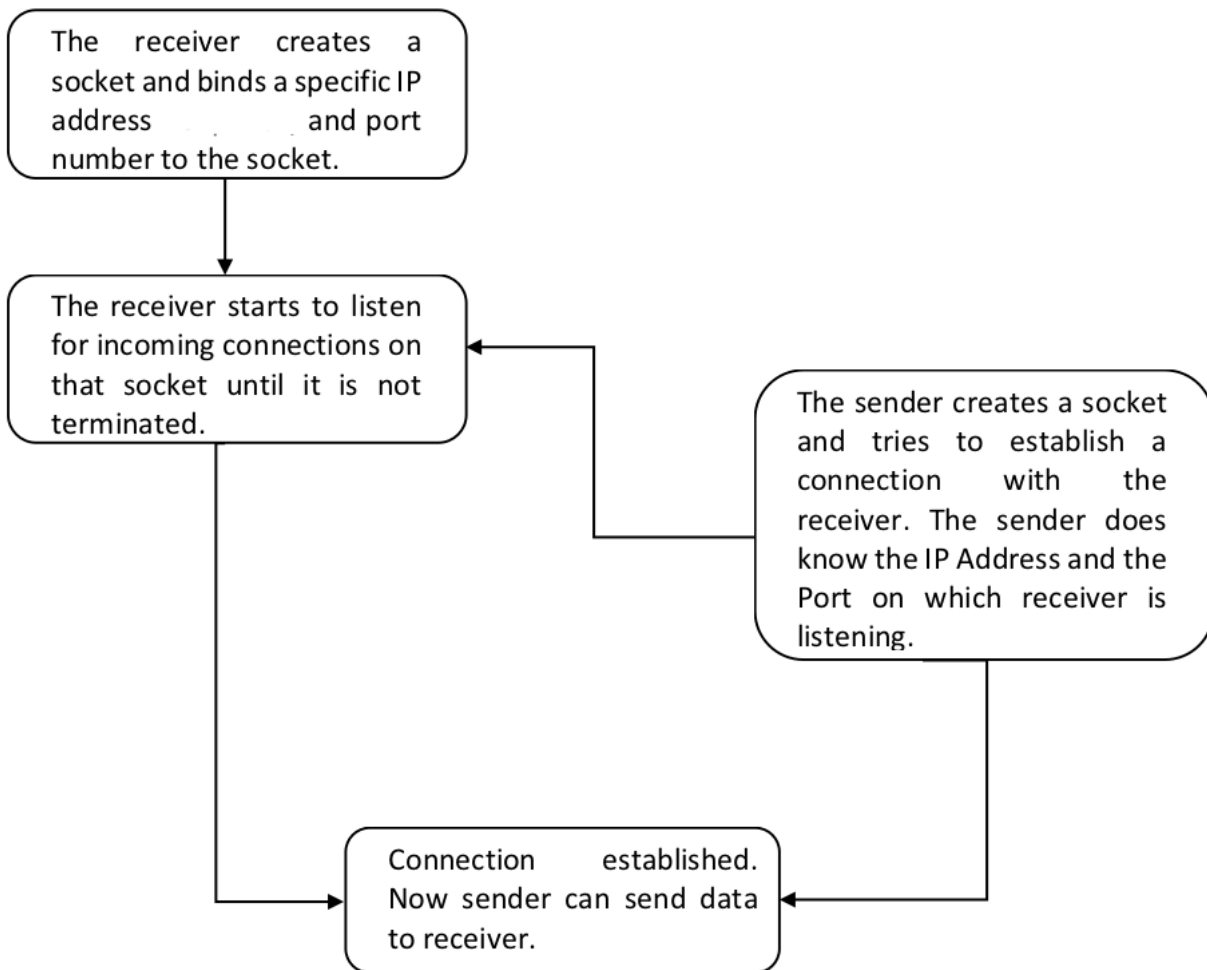


Figure 1: A High Level Overview of the Program

The Sender Program Code



```
1  import socket
2  import json
3
4
5  # This functions sends a message to the receiver
6  # The parameters of this function are conn: which
7  # is a socket object and msg which is a string that
8  # is sent to the sender.
9
10 def send_msg(conn, msg):
11     serialized = json.dumps(msg).encode('utf-8')
12     length = str(len(serialized)) + '\n'
13     length = length.encode('utf-8')
14     conn.send(length)
15     conn.sendall(serialized)
16
17
18 # Try to define a socket
19 try:
20     s = socket.socket()
21     print("Socket Created")
22 except socket.error as err:
23     print("Socket Creation Failed with Error :: ", err)
24
25
26 # Define the port Number on which
27 # the receiver listens
28 RECEIVER_PORT = 1234
29
30 # Define the IP address of the receiver
31 RECEIVER_IP = "192.168.1.252"
32
33 # Connect the sender to the receiver
34 s.connect((RECEVER_IP, RECEIVER_PORT))
35
36 # Send a message to the sender
37 send_msg(s, "Hello World!")
38 print("Message Sent!")
39 s.close()
```

Figure 2: The Sender Code

The Receiver Program Code

```
receiver.py
1 import socket
2 import json
3
4 # This function returns a string
5 # which contains the messages sent
6 # by the sender. The protocol is
7 # that a message ends with a new line
8 # character "\n"
9 def get_message(conn):
10     length_str = b''
11     char = conn.recv(1)
12     while char != b'\n':
13         length_str += char
14         char = conn.recv(1)
15     total = int(length_str)
16     off = 0
17     msg = b''
18     while off < total:
19         temp = conn.recv(total - off)
20         off = off + len(temp)
21         msg = msg + temp
22     return json.loads(msg.decode('utf-8'))
23
24
25 # Try to define a socket
26 try:
27     s = socket.socket()
28     print("Socket Created")
29 except socket.error as err:
30     print("Socket Creation Failed with Error :: ", err)
31
32
33 # Define the port on which the receiver
34 # will listen to receiver requests
35 port = 1234
36
37 # The host refers to the local IP Address of
38 # my computer.
39 host = socket.gethostbyname(socket.gethostname())
40
41
42 # Bind the socket and host on
43 # which the receiver will accept
44 # incoming connections
45 s.bind((host, port))
```

```

48 # Make the receiver listen on the specified
49 # IP Address and port number for connection
50 # requests from the sender
51 s.listen(0)
52 print("Host ", host, " is listening on ", port)
53
54 ▼ try:
55 ▼     while True:
56         # conn is a socket new object used for
57         # communication between the sender and
58         # the receiver. addr is the IP address
59         # of the sender.
60         conn, addr = s.accept()
61         print("Got connection form ", addr)
62         message = get_message(conn)
63         print("Message from Sender :: ", message)
64
65 # A keyboard Interrupt (ctrl + c) ends the socket
66 ▼ except KeyboardInterrupt:
67     print("Socket Closed!")
68     s.close()
69

```

Figure 3: The Receiver Code

The receiver first tries to create a socket. Then binds the IP Address and our specified port number to the socket. After that the receiver goes into an infinite while loop until it receives a connection from the sender and also a message/data. The receiver program can be terminated anytime by pressing (Ctrl + C). This ensures that the socket is closed properly and the port can be reused again. The receiver program has one function defined *get_message(conn)*. The parameter for this function is the socket, using which the sender and receiver communicate, and the output of this function is a string.

The logic behind the sender program is pretty straightforward. The sender creates a socket and establishes a connection with the receiver using the IP Address and the Port Number on which the receiver is running. The sender then simply sends a "Hello World!" message to the receiver using the *send_msg(conn, msg)* function and closes the socket. While all of this is happening **Wireshark** is running in the background to sniff the packets. The images below show the program in working.

```

~/Desktop/HW2
Muhammads-MBP:HW2 talhakhani$ python3 receiver.py
Socket Created
Host 192.168.1.252 is listening on 1234

```

Figure 4: Receiver Side

```

Mohameds-MBP:Desktop mohamed$ python sender.py
Socket Created
Message Sent!
Mohameds-MBP:Desktop mohamed$

```

Figure 5: Sender Side

```

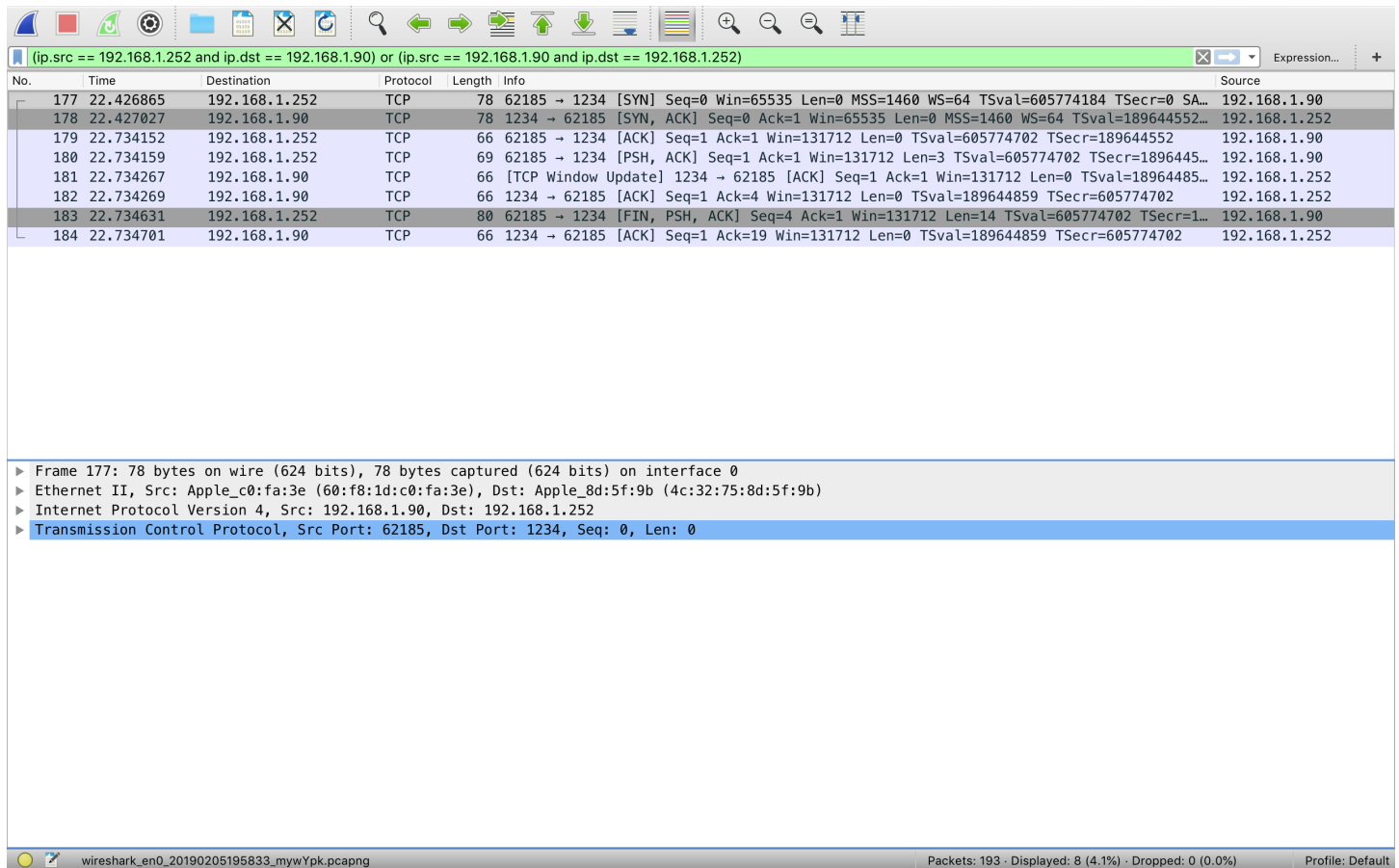
~/Desktop/HW2
Muhammads-MBP:HW2 talhakhani$ python3 receiver.py
Socket Created
Host 192.168.1.252 is listening on 1234
Got connection form ('192.168.1.90', 61954)
Message from Sender :: Hello World!

```

Figure 6: Receiver Side

The Wireshark Captured Packets

Wireshark was used to sniff the packets that were sent to/from the sender receiver. For this experiment I used two different Macbooks. The IP Address of the sender was "192.168.1.90" and for the receiver it was "192.168.1.252". Both of these computer were connected to the same network. I also had to use a filter for Wireshark so that the only the packets which were of interest could be captured. The image below shows the packets that were captured and the filter that was used.



The image shows the Wireshark network protocol analyzer interface. The top toolbar contains various icons for file operations, network analysis, and display. Below the toolbar is a green filter bar with the expression: `(ip.src == 192.168.1.252 and ip.dst == 192.168.1.90) or (ip.src == 192.168.1.90 and ip.dst == 192.168.1.252)`. The main packet list displays 8 captured packets, numbered 177 to 184. The details pane for the selected packet (183) shows the following structure:

No.	Time	Destination	Protocol	Length	Info	Source
177	22.426865	192.168.1.252	TCP	78	62185 → 1234 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=64 TSval=605774184 TSecr=0 SA...	192.168.1.90
178	22.427027	192.168.1.90	TCP	78	1234 → 62185 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=64 TSval=189644552...	192.168.1.252
179	22.734152	192.168.1.252	TCP	66	62185 → 1234 [ACK] Seq=1 Ack=1 Win=131712 Len=0 TSval=605774702 TSecr=189644552...	192.168.1.90
180	22.734159	192.168.1.252	TCP	69	62185 → 1234 [PSH, ACK] Seq=1 Ack=1 Win=131712 Len=3 TSval=605774702 TSecr=1896445...	192.168.1.90
181	22.734267	192.168.1.90	TCP	66	[TCP Window Update] 1234 → 62185 [ACK] Seq=1 Ack=1 Win=131712 Len=0 TSval=18964485...	192.168.1.252
182	22.734269	192.168.1.90	TCP	66	1234 → 62185 [ACK] Seq=1 Ack=4 Win=131712 Len=0 TSval=189644859 TSecr=605774702	192.168.1.252
183	22.734631	192.168.1.252	TCP	80	62185 → 1234 [FIN, PSH, ACK] Seq=4 Ack=1 Win=131712 Len=14 TSval=605774702 TSecr=1...	192.168.1.90
184	22.734701	192.168.1.90	TCP	66	1234 → 62185 [ACK] Seq=1 Ack=19 Win=131712 Len=0 TSval=189644859 TSecr=605774702	192.168.1.252

The details pane for packet 183 shows the following information:

- Frame 177: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface 0
- Ethernet II, Src: Apple_c0:fa:3e (60:f8:1d:c0:fa:3e), Dst: Apple_8d:5f:9b (4c:32:75:8d:5f:9b)
- Internet Protocol Version 4, Src: 192.168.1.90, Dst: 192.168.1.252
- Transmission Control Protocol, Src Port: 62185, Dst Port: 1234, Seq: 0, Len: 0

The status bar at the bottom indicates: `wireshark_en0_20190205195833_mywYpk.pcapng`, Packets: 193 · Displayed: 8 (4.1%) · Dropped: 0 (0.0%) · Profile: Default

Figure 7: Wireshark Captured Packets

The first three packets (177, 178, 179) correspond to the TCP handshake between the sender and the receiver. Packet 183 is the packet which contains the data which in our case was "Hello World!". After further inspection of packet 183 I was able to answer the questions.

- Bytes **0-5** encode the MAC Address or Physical Address of the receiver.
- Bytes **6-11** encode the MAC Address of Physical Address of the sender.
- Byte **14** encodes the IP Header length which in my case was 20 Bytes.
- Bytes **16-17** encodes the total length, which was 66 bytes in my case. The total length specifies the length of the IP Packet which includes the IP Header size (byte **14**) plus the user data size. This can also be referred to as the IP Datagram length.
- Bytes **18-19** encode the IP Identification or IP ID. As an IP Packet moves through the Internet it sometimes is fragmented because of bandwidth restrictions on some links. This IP ID helps reassemble it again.

- Bytes **20-21** encode the fragment offset which specifies the offset of a particular fragment relative to the beginning of the original unfragmented IP Datagram. In my case it was 0.
- Byte **23** encodes the Protocol. The value of the Protocol field is used to specify the protocol used to create the data in the Data field. In my case the protocol was TCP.
- Bytes **26-29** encode the source (sender) IP Address.
- Bytes **30-33** encode the destination (receiver) IP Address.
- Bytes **34-35** encode the source (sender) port number.
- Bytes **36-37** encode the destination (receiver) port number.
- The bytes after **65** encode the data contained in the packet which in my case was "Hello World!". In my case the data size was 14 bytes.
- Bytes **0-13** correspond to the Link Layer.
- Bytes **14-33** correspond to the Network Layer.
- Bytes **34-65** correspond to the Transport Layer.
- Bytes **65-79** correspond to the Application Layer