CSLR 52
Networks Lab
Assignment 1

106119029
Dipesh Kafle

Questions

---

1) Write a server program for TCP using Python to do the following:

    a. Server returns the binary value of the text sent by the client. Example: for a text string "comnetsii", the client should receive "01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001"

    b. The server should be running at the localhost interface

    c. You are free to choose any port

2) The Binary Decoding Server: Write another server running on eth1 interface to do the following:

    a. Server returns the string value of a binary input. Example: the binary string "01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001" sent from the client should return "comnetsii"

3) Write two client programs one for each of the server.

    a. What happens if the client aborts (e.g., when the input is CTRL/D)?

    b. Can you run two clients against the same server? Why or why not? Hint: Multithreaded socket server in Python

    c. What happens when you try to connect client1 to server2 by passing a wrong network address (e.g., connecting to localhost instead of eth1 IP)?

Question 1

Code

server1.py

```python
import socket
import optparse
import struct
import threading

MAX_QUEUE = 10

'''
This is the part where I am doing
command line argument parsing, so that
the server is more configurable

- The default port is 3000
- The default interface is lo(localhost)
- All these can be overriden through command line args
- To override port number, pass --port <port_no>
- To override interface, pass --interface <interface_name>
'''
parser = optparse.OptionParser()
parser.add_option('-p', '--port', dest='port', default=3000,
                  type="int", help='Port to run the server on')
parser.add_option('-I', '--interface', dest='interface',
                  default='lo', help='Network interface to run the server on')
parser.add_option('-M', '--multi-threaded', action='store_true',
                  dest='multi_threaded', default=False, help='Tells weather multi threading should be done to handle multiple requests')


(options, args) = parser.parse_args()
INTERFACE = options.interface
PORT = options.port
MULTI_THREADED = options.multi_threaded
print("Interface: {}, Port: {}, Multi-threaded?: {}".format(INTERFACE, PORT, MULTI_THREADED))

```

This part of the code creates a command line argument parser, which will parse the arguments for

- Port: to set the port, default is 3000
- Interface: to set the interface, default is lo(localhost)
- Multithreaded: to make the server multithreaded, default behaviour is single single threaded

```python
def pad_to_8chars(c: str) -> str:
    '''
    Given a string of length less than
    or equal to 8, this will pad it to
    8 characters putting 0s in front of it
    '''
    assert(len(c) <= 8)
    return '0'*(8-len(c))+c


def send_int(x: int, conn):
    conn.send(struct.pack("i", socket.htons(x)))


def recv_int(conn) -> int:
    return int.from_bytes(conn.recv(4), byteorder='big')


def data_to_send(msg: str):
    return ''.join((map(lambda x: pad_to_8chars(
        bin(ord(x))[2:])+' ', msg)))

```

These are all helper functions

- **pad_to_8chars**: This will make pad the given string to 8 characters if it is less than 8 characters. It will pad it with 0s in front.
- **send_int**: Sends and integer to the client in binary 4 bytes form in correct endian form.
- **recv_int** : Receives an integer from client and fixes the endian form.
- **data_to_send** : This function takes in a string, and for every character in the string, it will convert it to ascii value and that ascii value to its corresponding binary string. After that, the binary string will be padded with 0s in front to make it 8 characters using the pad_to_8chars function. All these binary strings will be put in one string and returned

```
def handle_client(conn, addr):
    print("Connected to ",  addr)
    while True:
        data_size = recv_int(conn)
        if(data_size == 0):
            break
        msg = conn.recv(data_size).decode('utf-8')
        print("Received: {}".format(msg))
        to_be_sent = data_to_send(msg)
        send_int(len(to_be_sent), conn)
        conn.send(to_be_sent.encode('utf-8'))
        print("Sent: {}".format(to_be_sent))


def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
        sock.bind(('0.0.0.0',  PORT))
        sock.listen(MAX_QUEUE)
        while True:
            if not MULTI_THREADED:
                conn, addr = sock.accept()
                handle_client(conn, addr)
                break
            else:
                conn, addr = sock.accept()
                thrd = threading.Thread(
                    target=handle_client, args=[conn, addr])
                thrd.start()
                continue


main()
```

- **handle_client** : This function will first print the address it is connected to(i.e the client address). After that in a while loop, it does the following:
    - Tries to receive an integer from the client
    - If data_size == 0, we break,meaning the connection is to be closed or was already closed by client.
    - Now it will try to recv data from the client. The amount of data it is ready to take is equal to the data_size.

    - After receiving the data, it will call **data_to_send** function with the received data to encode the string sent by client.
    - It then proceeds to send the size of encoded message to the client first, so that client can know the size of message to expect. This is done so that we can have systematic communication.
    - After sending the size, we send encoded message to the client and print that message in server for our own convenience.
    - It will try to repeat the same thing again and again until the connection is to be stopped or interrupted.

- **main** : The flow of the main function is as follows:
    - We open the socket connection
    - We will use setsockopt to connect it to eth1 interface as mentioned in the question. Doing this will set the interface of socket to be eth1.
    - We bind the socket to appropriate port and ip (Binding to 0.0.0.0 will automatically figure out the IP of the interface and bind to it).
    - We set the socket queue length , using sock.listen
    - After that depending on whether the server is supposed to be multi threaded or not , we handle
        - If multithreaded, we spawn a thread for every single client that connects to it and handle it
        - Else we'll just handle that one client in main thread and we exit when that connection terminates

Question 2

Code

server2.py

```
  import socket
  import optparse
  import struct
  import threading


  MAX_QUEUE = 10

  '''
  This is the part where I am doing
  command line argument parsing, so that
  the server is more configurable

  - The default port is 3000
  - The default interface is eth1
  - All these can be overriden through command line args
  - To override port number, pass --port <port_no>
  - To override interface, pass --interface <interface_name>
  '''
  parser = optparse.OptionParser()
  parser.add_option('-p', '--port', dest='port', default=3000,
                    type="int", help='Port to run the server on')
  parser.add_option('-I', '--interface', dest='interface',
                    default='eth1', help='Network interface to run the server on')
  parser.add_option('-M', '--multi-threaded', action='store_true',
                    dest='multi_threaded', default=False, help='Tells weather multi threading should be done to handle multiple requests')
  (options, args) = parser.parse_args()
  INTERFACE = options.interface
  PORT = options.port
  MULTI_THREADED = options.multi_threaded
  print("Interface: {}, Port: {}, Multi-threaded?: {}".format(INTERFACE, PORT, MULTI_THREADED))

```

This part of the code creates a command line argument parser, which will parse the arguments for

- Port: to set the port, default is 3000
- Interface: to set the interface, default is eth1
- Multithreaded: to make the server multithreaded, default behaviour is single single threaded

```
22 def send_int(x: int, conn):
21     conn.send(struct.pack("i", socket.htons(x)))
20
19
18 def recv_int(conn) → int:
17     return int.from_bytes(conn.recv(4), byteorder='big')
16
15
14 def data_to_send(msg: str):
13     return ''.join(map(lambda x: chr(int(x, 2)), msg.split()))
12
```

These are all helper functions

- **send_int**: Sends and integer to the client in binary 4 bytes form in correct endian form.
- **recv_int** : Receives an integer from client and fixes the endian form.
- **data_to_send** : This function takes in a string. First of all, it will split the string. For each string after the split, we'll conver the string to integer,because its a binary string(as per the question). Then convert the integer to ascii character. We'll put all the characters in one string and return it. Basically will decode the binary string and return it.

```
def handle_client(conn, addr):
    print("Connected to ",  addr)
    while True:
        data_size = recv_int(conn)
        if(data_size == 0):
            break
        msg = conn.recv(data_size).decode('utf-8')
        print("Received: {}".format(msg))
        to_be_sent = data_to_send(msg)
        send_int(len(to_be_sent), conn)
        conn.send(to_be_sent.encode('utf-8'))
        print("Sent: {}".format(to_be_sent))


def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
        sock.bind(('0.0.0.0',  PORT))
        sock.listen(MAX_QUEUE)
        while True:
            if not MULTI_THREADED:
                conn, addr = sock.accept()
                handle_client(conn, addr)
                break
            else:
                conn, addr = sock.accept()
                thrd = threading.Thread(
                    target=handle_client, args=[conn, addr])
                thrd.start()
                continue


main()
```

- **handle_client** : This function will first print the address it is connected to(i.e the client address). After that in a while loop, it does the following:
    - Tries to receive an integer from the client
    - If data_size == 0, we break,meaning the connection is to be closed or was already closed by client.
    - Now it will try to recv data from the client. The amount of data it is ready to take is equal to the data_size.
    - After receiving the data, it will call **data_to_send** function with the received data to decode the string sent by client.
    - It then proceeds to send the size of decoded message to the client first, so that client can know the size of message to expect. This is done so that we can have systematic communication.
    - After sending the size, we send decoded message to the client and print that message in server for our own convenience.
    - It will try to repeat the same thing again and again until the connection is to be stopped or interrupted.

- **main** : The flow of the main function is as follows:
    - We open the socket connection
    - We will use setsockopt to connect it to eth1 interface as mentioned in the question. Doing this will set the interface of socket to be eth1.

- We bind the socket to appropriate port and ip (Binding to 0.0.0.0 will automatically figure out the IP of the interface and bind to it).
- We set the socket queue length , using sock.listen
- After that depending on whether the server is supposed to be multi threaded or not , we handle
    - If multithreaded, we spawn a thread for every single client that connects to it and handle it
    - Else we'll just handle that one client in main thread and we exit when that connection terminates

Question 3

Code

client1.py

```
1   import socket
 1  import optparse
 2  import struct
 3
 4
 5  '''
 6  This is the part where I am doing
 7  command line argument parsing, so that
 8  the server is more configurable
 9
10  - The default port is 3000
11  - The default interface is lo(localhost)
12  - All these can be overriden through command line args
13  - To override port number, pass --port <port_no>
14  - To override interface, pass --interface <interface_name>
15  '''
16  parser = optparse.OptionParser()
17  parser.add_option('-p', '--port', dest='port', default=3000,
18                    type="int", help='Port to run the server on')
19  parser.add_option('-I', '--interface', dest='interface',
20                    default='lo', help='Network interface to run the server on')
21  parser.add_option('-a', '--use-ip-addr', dest='ip_addr', default='',
22                    help='Connect through IP instead of using the interface')
23
24
```

This part of the code creates a command line argument parser, which will parse the arguments for

- Port: to set the port, default is 3000
- Interface: to set the interface, default is lo
- IP Address: We can specify the program to use IP Address instead of using the network interface directly. By default it'll use interface unless this argument is provided.

```
def send_int(x: int, conn):
    conn.send(struct.pack("i", socket.htons(x)))


def recv_int(conn) -> int:
    return int.from_bytes(conn.recv(4), byteorder='big')


def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        if(IP_ADDR == ''):
            sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
            sock.connect(('0.0.0.0',  PORT))
        else:
            sock.connect((IP_ADDR, PORT))

        print(sock.getpeername())
        while True:
            msg = input()
            if(msg == ""):
                sock.close()
                break
            msg = msg.strip()
            send_int(len(msg), sock)
            sock.send(msg.encode('utf-8'))
            print("Sent: ", msg)
            msg_sz = recv_int(sock)
            recvd_msg = sock.recv(msg_sz).decode('utf-8')
            print("Received: ", recvd_msg)


main()
```

- **send_int**: Sends and integer to the server in binary 4 bytes form in correct endian form.
- **recv_int** : Receives an integer from server and fixes the endian form.
- **main** : It performs the following steps:
    - Opens a socket(TCP) file descriptor
    - If we have specified IP Address, it connects to that IP address and port
    - Else it will use set sockopt to use the provided interface(default localhost in this case), similar to how it was done in server files. It will now connect to the server.

client2.py

```
import socket
import optparse
import struct


'''
This is the part where I am doing
command line argument parsing, so that
the server is more configurable

- The default port is 3000
- The default interface is eth1
- All these can be overriden through command line args
- To override port number, pass --port <port_no>
- To override interface, pass --interface <interface_name>
'''
parser = optparse.OptionParser()
parser.add_option('-p', '--port', dest='port', default=3000,
                  type="int", help='Port to run the server on')
parser.add_option('-I', '--interface', dest='interface',
                  default='eth1', help='Network interface to run the server on')
parser.add_option('-a', '--use-ip-addr', dest='ip_addr', default='',
                  help='Connect through IP instead of using the interface')

(options, args) = parser.parse_args()
INTERFACE = options.interface
PORT = options.port
IP_ADDR = options.ip_addr
print("Interface: {}, Port: {}".format(INTERFACE, PORT))

```

```
import socket
import optparse
import struct


'''
This is the part where I am doing
command line argument parsing, so that
the server is more configurable

- The default port is 3000
- The default interface is lo(localhost)
- All these can be overriden through command line args
- To override port number, pass --port <port_no>
- To override interface, pass --interface <interface_name>
'''
parser = optparse.OptionParser()
parser.add_option('-p', '--port', dest='port', default=3000,
                  type="int", help='Port to run the server on')
parser.add_option('-I', '--interface', dest='interface',
                  default='lo', help='Network interface to run the server on')
parser.add_option('-a', '--use-ip-addr', dest='ip_addr', default='',
                  help='Connect through IP instead of using the interface')


```

This part of the code creates a command line argument parser, which will parse the arguments for

- Port: to set the port, default is 3000
- Interface: to set the interface, default is eth1
- IP Address: We can specify the program to use IP Address instead of using the network interface directly. By default it'll use interface unless this argument is provided.

```
def send_int(x: int, conn):
    conn.send(struct.pack("i", socket.htons(x)))


def recv_int(conn) -> int:
    return int.from_bytes(conn.recv(4), byteorder='big')


def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        if(IP_ADDR == ''):
            sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
            sock.connect(('0.0.0.0',  PORT))
        else:
            sock.connect((IP_ADDR, PORT))
        print(sock.getpeername())
        while True:
            msg = input()
            if(msg == ""):
                sock.close()
                break
            msg = msg.strip()
            send_int(len(msg), sock)
            sock.send(msg.encode('utf-8'))
            print("Sent: ", msg)
            msg_sz = recv_int(sock)
            recvd_msg = sock.recv(msg_sz).decode('utf-8')
            print("Received: ", recvd_msg)


main()
```

- **send_int**: Sends and integer to the server in binary 4 bytes form in correct endian form.
- **recv_int** : Receives an integer from server and fixes the endian form.
- **main** : It performs the following steps:
    - Opens a socket(TCP) file descriptor
    - If we have specified IP Address, it connects to that IP address and port
    - Else it will use set sockopt to use the provided interface(default localhost in this case), similar to how it was done in server files. It will now connect to the server.

Question 3.a

-> What happens if the client aborts (e.g., when the input is CTRL/D)?

Ans:

When we press Ctrl+D, the client will throw EOFError and exit. The server will see that the client has exited (because the connection will be cut off between them) and hence it is exiting as well .

- On Ctrl D input

Question 3.b

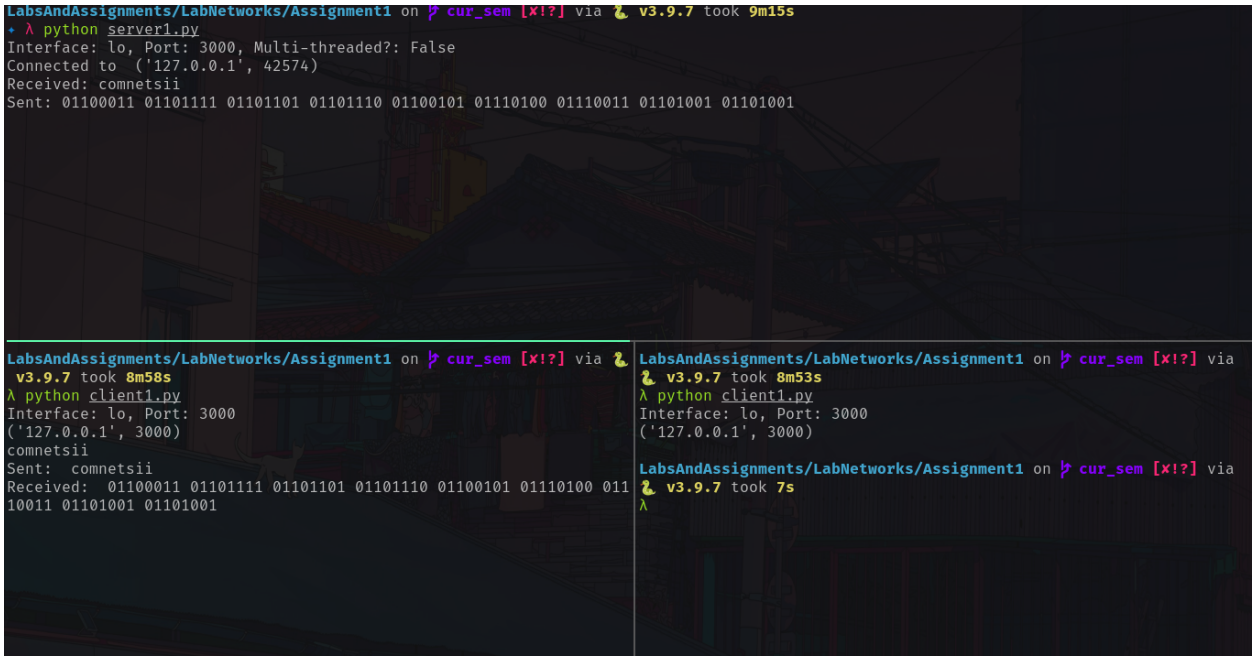
-> Can you run two clients against the same server? Why or why not? Hint: Multithreaded socket server in Python

Ans:

This will depend on how we have written the server code. I had written in such a way that we can optionally make the server multi threaded by passing '--multi-threaded' flag while running server1.py or server2.py. If those options are not passed however, it wont have multi-threading.

In the image attached below, we can see that the second instance of client1.py is connecting to the server1.py but the server1.py is not accepting the connection(had it accepted it would've printed it). The same thing would happen for server2.py and client2.py.

If the multi-threaded mode is turned on however, it will handle multiple connections by spawning a thread to handle each connection. It can be seen happening clearly in the Output image for multi-threaded that I have put above in **Output(Multi Threaded)** section above.



- Multiple clients trying to connect to single threaded server1.py

Question 3.c

-> What happens when you try to connect client1 to server2 by passing a wrong network address (e.g., connecting to localhost instead of eth1 IP)?

Ans:

There can be two things that might happen.

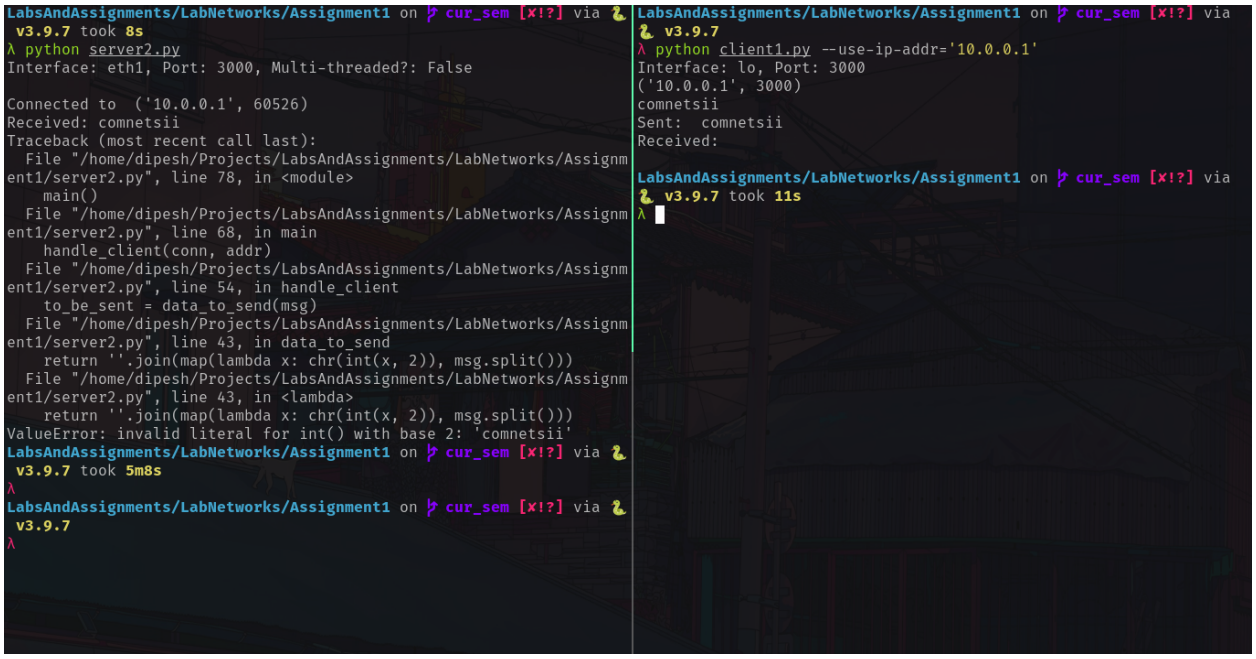
Case 1:

If we have server2.py running and we run client1.py , client1.py will try to connect to localhost but localhost wouldnt be running(only eth1 ip would be connectable because server2.py is running). In this case, the result can be seen in the image attached below.

- Case 1

Case 2:

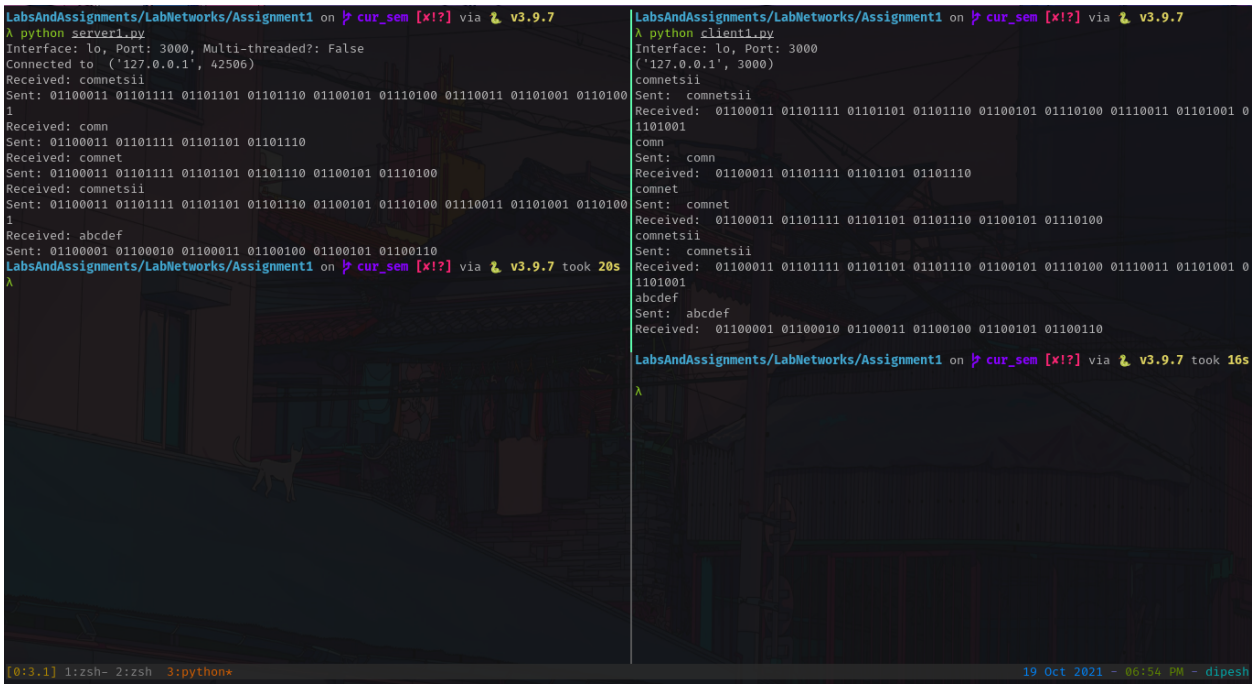
If we pass the eth1 ip to client1.py instead and try to connect to server2.py, the behaviour will be weird because the format in which server2.py expects its message will be different from what its getting. server2.py expects space separated bit strings of length less than or equal to 8 but its getting random ascii strings, so server2.py wont be able to decode those letters and crash as a result.



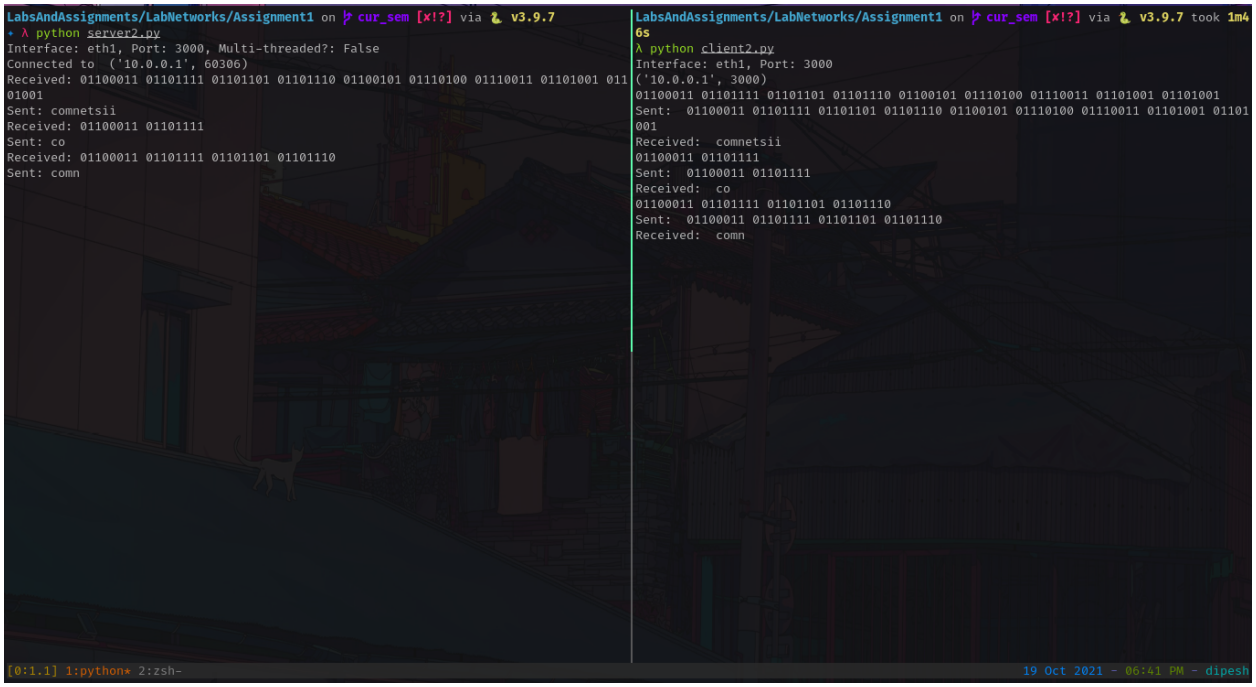
- Case 2

Output

Output(Single Threaded)

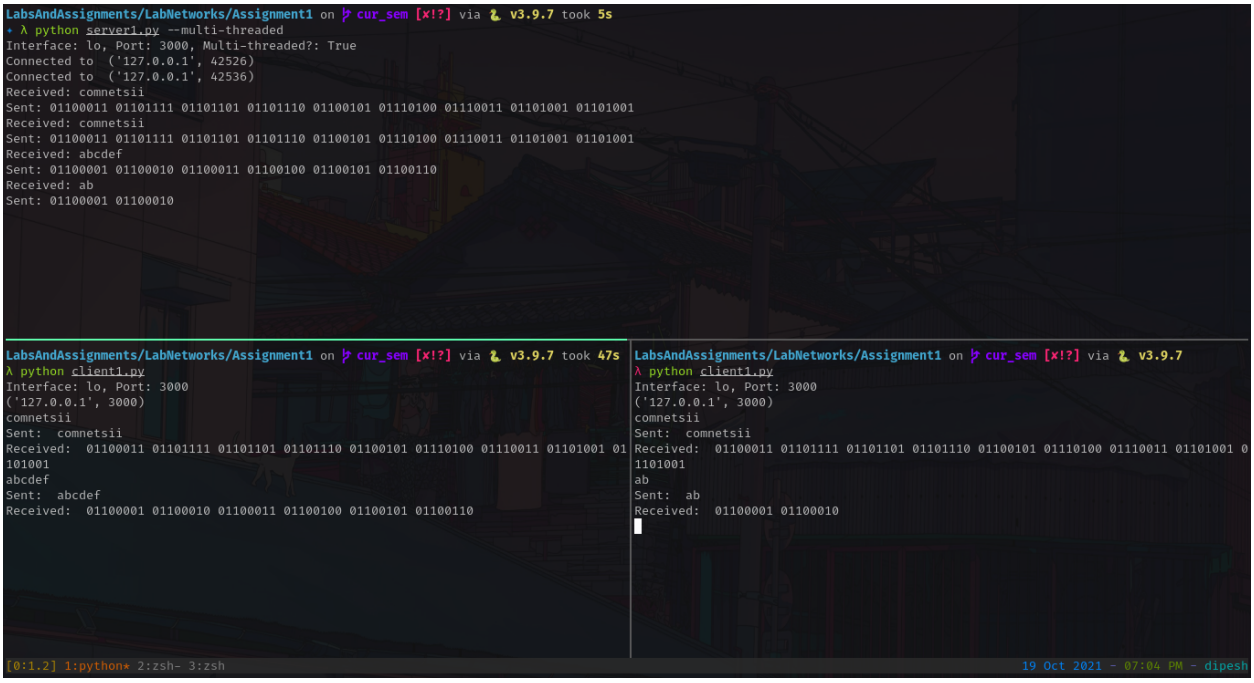


-Running server1.py and client1.py side by side

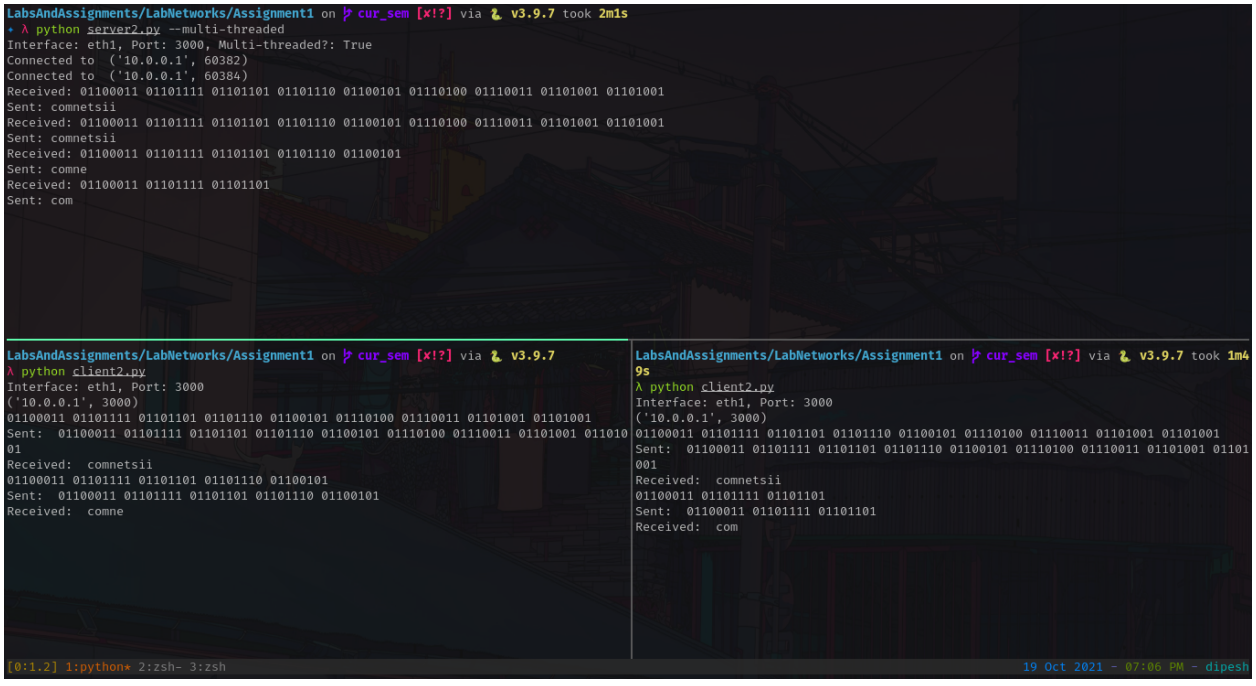


- Running server2.py and client2.py

Output(Multi Threaded)



- Multiple client1.py connecting to server1.py



- Multiple client2.py connecting to server2.py