

**CSLR 52**

**Networks Lab**

**Assignment 1**

**106119029**

**Dipesh Kafle**

<b>Questions</b>	<b>2</b>
<b>Question 1</b>	<b>2</b>
Code	2
<b>Question 2</b>	<b>6</b>
Code	6
<b>Question 3</b>	<b>9</b>
Code	9
Question 3.a	13
Question 3.b	13
Question 3.c	14
<b>Output</b>	<b>16</b>
Output(Single Threaded)	16
Output(Multi Threaded)	18

# 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 “commetsii”, the client should receive “01100011 01101111 01101101 01101110 01100101 01101000 01100111 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 01101000 01100111 01101001 01101001” sent from the client should return “commetsii”
- 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

```

1 import socket
2 import optparse
3 import struct
4 import threading
5
6 ...
7 This is the part where I am doing
8 command line argument parsing, so that
9 the server is more configurable
10
11 - The default port is 3000
12 - The default interface is lo(localhost)
13 - All these can be overridden through command line args
14 - To override port number, pass --port <port_no>
15 - To override interface, pass --interface <interface_name>
16 ...
17 parser = optparse.OptionParser()
18 parser.add_option('-p', '--port', dest='port', default=3000,
19                   type="int", help='Port to run the server on')
20 parser.add_option('-I', '--interface', dest='interface',
21                   default='lo', help='Network interface to run the server on')
22 parser.add_option('-M', '--multi-threaded', action='store_true',
23                   dest='multi_threaded', default=False, help='Tells weather multi threading should be done to handle multiple requests')
24
25
26 (options, args) = parser.parse_args()
27 INTERFACE = options.interface
28 PORT = options.port
29 MULTI_THREADED = options.multi_threaded
30 print("Interface: {}, Port: {}, Multi-threaded?: {}".format(INTERFACE, PORT, MULTI_THREADED))
31

```

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 threaded

```

34 def pad_to_8chars(c: str) → str:
35     """
36         Given a string of length less than
37         or equal to 8, this will pad it to
38         8 characters putting 0s in front of it
39     """
40     assert(len(c) ≤ 8)
41     return '0'*(8-len(c))+c
42
43
44 def send_int(x: int, conn):
45     conn.send(struct.pack("i", socket.htons(x)))
46
47
48 def recv_int(conn) → int:
49     return int.from_bytes(conn.recv(4), byteorder='big')
50
51
52 def data_to_send(msg: str):
53     return ''.join((map(lambda x: pad_to_8chars(
54         bin(ord(x))[2:]+' ', msg))))
55
56
57

```

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 an 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

```
32 def handle_client(conn, addr):
31     print("Connected to ", addr)
30     while True:
29         data_size = recv_int(conn)
28         if(data_size == 0):
27             break
26         msg = conn.recv(data_size).decode('utf-8')
25         print("Received: {}".format(msg))
24         to_be_sent = data_to_send(msg)
23         send_int(len(to_be_sent), conn)
22         conn.send(to_be_sent.encode('utf-8'))
21         print("Sent: {}".format(to_be_sent))
20
19
18 def main():
17     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
16         sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
15         sock.bind(( '0.0.0.0', PORT))
14         sock.listen(MAX_QUEUE)
13         while True:
12             if not MULTI_THREADED:
11                 conn, addr = sock.accept()
10                 handle_client(conn, addr)
9                 break
8             else:
7                 conn, addr = sock.accept()
6                 thrd = threading.Thread(
5                     target=handle_client, args=[conn, addr])
4                 thrd.start()
3                 continue
2
1
90 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

```
1 import socket
2 import optparse
3 import struct
4 import threading
5
6 MAX_QUEUE = 10
7 ...
8 This is the part where I am doing
9 command line argument parsing, so that
10 the server is more configurable
11
12 - The default port is 3000
13 - The default interface is eth1
14 - All these can be overridden through command line args
15 - To override port number, pass --port <port_no>
16 - To override interface, pass --interface <interface_name>
17 ...
18 parser = optparse.OptionParser()
19 parser.add_option('-p', '--port', dest='port', default=3000,
20                   type='int', help='Port to run the server on')
21 parser.add_option('-I', '--interface', dest='interface',
22                   default='eth1', help='Network interface to run the server on')
23 parser.add_option('-M', '--multi-threaded', action='store_true',
24                   dest='multi_threaded', default=False, help='Tells weather multi threading should be done to handle multiple requests')
25 (options, args) = parser.parse_args()
26 INTERFACE = options.interface
27 PORT = options.port
28 MULTI_THREADED = options.multi_threaded
29 print("Interface: {}, Port: {}, Multi-threaded?: {}".format(INTERFACE, PORT, MULTI_THREADED))
30
31
```

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 threaded

```
23
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 convert 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.

```

32 def handle_client(conn, addr):
31     print("Connected to ", addr)
30     while True:
29         data_size = recv_int(conn)
28         if(data_size == 0):
27             break
26         msg = conn.recv(data_size).decode('utf-8')
25         print("Received: {}".format(msg))
24         to_be_sent = data_to_send(msg)
23         send_int(len(to_be_sent), conn)
22         conn.send(to_be_sent.encode('utf-8'))
21         print("Sent: {}".format(to_be_sent))
20
19
18 def main():
17     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
16         sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
15         sock.bind(('0.0.0.0', PORT))
14         sock.listen(MAX_QUEUE)
13         while True:
12             if not MULTI_THREADS:
11                 conn, addr = sock.accept()
10                 handle_client(conn, addr)
9                 break
8             else:
7                 conn, addr = sock.accept()
6                 thrd = threading.Thread(
5                     target=handle_client, args=[conn, addr])
4                 thrd.start()
3                 continue
2
1
90 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 overridden 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.

```
31 def send_int(x: int, conn):
30     conn.send(struct.pack("i", socket.htons(x)))
29
28
27 def recv_int(conn) → int:
26     return int.from_bytes(conn.recv(4), byteorder='big')
25
24
23 def main():
22     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
21         if(IP_ADDR == ""):
20             sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
19             sock.connect(('0.0.0.0', PORT))
18         else:
17             sock.connect((IP_ADDR, PORT))
16
15     print(sock.getpeername())
14     while True:
13         msg = input()
12         if(msg == ""):
11             sock.close()
10             break
9         msg = msg.strip()
8         send_int(len(msg), sock)
7         sock.send(msg.encode('utf-8'))
6         print("Sent: ", msg)
5         msg_sz = recv_int(sock)
4         recvd_msg = sock.recv(msg_sz).decode('utf-8')
3         print("Received: ", recvd_msg)
2
1
64 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 setsockopt 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

```
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 eth1
12 - All these can be overridden 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='eth1', 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 (options, args) = parser.parse_args()
25 INTERFACE = options.interface
26 PORT = options.port
27 IP_ADDR = options.ip_addr
28 print("Interface: {}, Port: {}".format(INTERFACE, PORT))
29
```

```
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 overridden 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 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.

```

30 def send_int(x: int, conn):
29     conn.send(struct.pack("i", socket.htons(x)))
28
27
26 def recv_int(conn) -> int:
25     return int.from_bytes(conn.recv(4), byteorder='big')
24
23
22 def main():
21     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
20         if(IP_ADDR == ''):
19             sock.setsockopt(socket.SOL_SOCKET, 25, INTERFACE.encode())
18             sock.connect(('0.0.0.0', PORT))
17         else:
16             sock.connect((IP_ADDR, PORT))
15     print(sock.getpeername())
14     while True:
13         msg = input()
12         if(msg == ""):
11             sock.close()
10             break
9         msg = msg.strip()
8         send_int(len(msg), sock)
7         sock.send(msg.encode('utf-8'))
6         print("Sent: ", msg)
5         msg_sz = recv_int(sock)
4         recvd_msg = sock.recv(msg_sz).decode('utf-8')
3         print("Received: ", recvd_msg)
2
1
62 main()

```

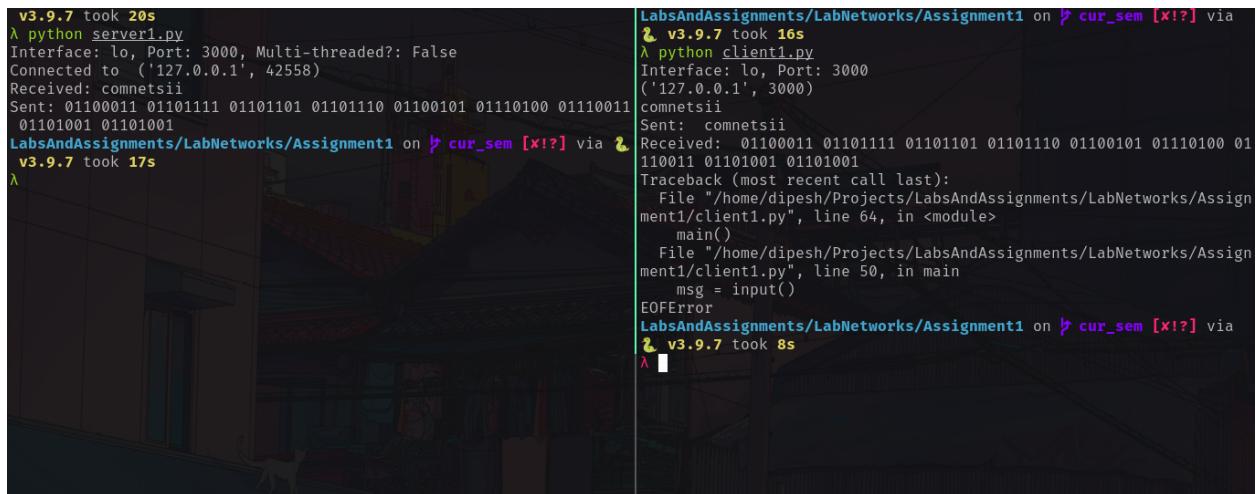
- **send\_int**: Sends an integer to the server in binary 4 bytes form in correct endian form.
- **recv\_int** : Receives an integer from server and fixes the endian form.
- **main** : It performs the following steps:
  - Opens a socket(TCP) file descriptor
  - If we have specified IP Address, it connects to that IP address and port
  - Else it will use setsockopt 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.



```
v3.9.7 took 20s
λ python server1.py
Interface: lo, Port: 3000, Multi-threaded?: False
Connected to ('127.0.0.1', 42558)
Received: commetsii
Sent: 01100011 01101111 01101101 01101110 01100101 01110100 01110011
01101001 01101001
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via λ
v3.9.7 took 17s
λ

LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via λ
v3.9.7 took 16s
λ python client1.py
Interface: lo, Port: 3000
('127.0.0.1', 3000)
commetsii
Received: 01100011 01101111 01101101 01101110 01100101 01110100 01
110011 01101001 01101001
Traceback (most recent call last):
  File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/client1.py", line 64, in <module>
    main()
  File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/client1.py", line 50, in main
    msg = input()
EOFError
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via λ
v3.9.7 took 8s
λ
```

- 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.

```
LabsAndAssignments/LabNetworks/Assignment1 on ⚡ cur_sem [x!?] via 🐍 v3.9.7 took 9m15s
+ λ python server1.py
Interface: lo, Port: 3000, Multi-threaded?: False
Connected to ('127.0.0.1', 42574)
Received: commetsii
Sent: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001

LabsAndAssignments/LabNetworks/Assignment1 on ⚡ cur_sem [x!?] via 🐍 v3.9.7 took 8m58s
λ python client1.py
Interface: lo, Port: 3000
('127.0.0.1', 3000)
commetsii
Sent: commetsii
Received: 01100011 01101111 01101101 01101110 01100101 01110100 01100111 01101001 01101001

LabsAndAssignments/LabNetworks/Assignment1 on ⚡ cur_sem [x!?] via 🐍 v3.9.7 took 8m53s
λ python client1.py
Interface: lo, Port: 3000
('127.0.0.1', 3000)

LabsAndAssignments/LabNetworks/Assignment1 on ⚡ cur_sem [x!?] via 🐍 v3.9.7 took 7s
λ
```

- 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.

```

LabsAndAssignments/LabNetworks/Assignment1 on ✘ cur_sem [x!?] via 2 v3.9.7 took 8s
λ python server2.py
Interface: eth1, Port: 3000, Multi-threaded?: False

LabsAndAssignments/LabNetworks/Assignment1 on ✘ cur_sem [x!?] via 2 v3.9.7
λ python client1.py
Interface: lo, Port: 3000
Traceback (most recent call last):
  File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/client1.py", line 65, in <module>
    main()
      File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/client1.py", line 45, in main
        sock.connect(('0.0.0.0', PORT))
ConnectionRefusedError: [Errno 111] Connection refused
LabsAndAssignments/LabNetworks/Assignment1 on ✘ cur_sem [x!?] via 2 v3.9.7
λ

```

### - 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.

```

LabsAndAssignments/LabNetworks/Assignment1 on ✘ cur_sem [x!?] via 2 v3.9.7 took 8s
λ python server2.py
Interface: eth1, Port: 3000, Multi-threaded?: False

Connected to ('10.0.0.1', 60526)
Received: commetsii
Traceback (most recent call last):
  File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/server2.py", line 78, in <module>
    main()
    File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/server2.py", line 68, in main
      handle_client(conn, addr)
      File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/server2.py", line 54, in handle_client
        to_be_sent = data_to_send(msg)
        File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/server2.py", line 43, in data_to_send
          return ''.join(map(lambda x: chr(int(x, 2)), msg.split()))
          File "/home/dipesh/Projects/LabsAndAssignments/LabNetworks/Assignment1/server2.py", line 43, in <lambda>
            return ''.join(map(lambda x: chr(int(x, 2)), msg.split()))
ValueError: invalid literal for int() with base 2: 'commetsii'
LabsAndAssignments/LabNetworks/Assignment1 on ✘ cur_sem [x!?] via 2 v3.9.7 took 5ms
λ
LabsAndAssignments/LabNetworks/Assignment1 on ✘ cur_sem [x!?] via 2 v3.9.7
λ python client1.py --use-ip-addr='10.0.0.1'
Interface: lo, Port: 3000
('10.0.0.1', 3000)
commetsii
Sent: commetsii
Received:

LabsAndAssignments/LabNetworks/Assignment1 on ✘ cur_sem [x!?] via 2 v3.9.7 took 11s
λ

```

### - Case 2

# Output

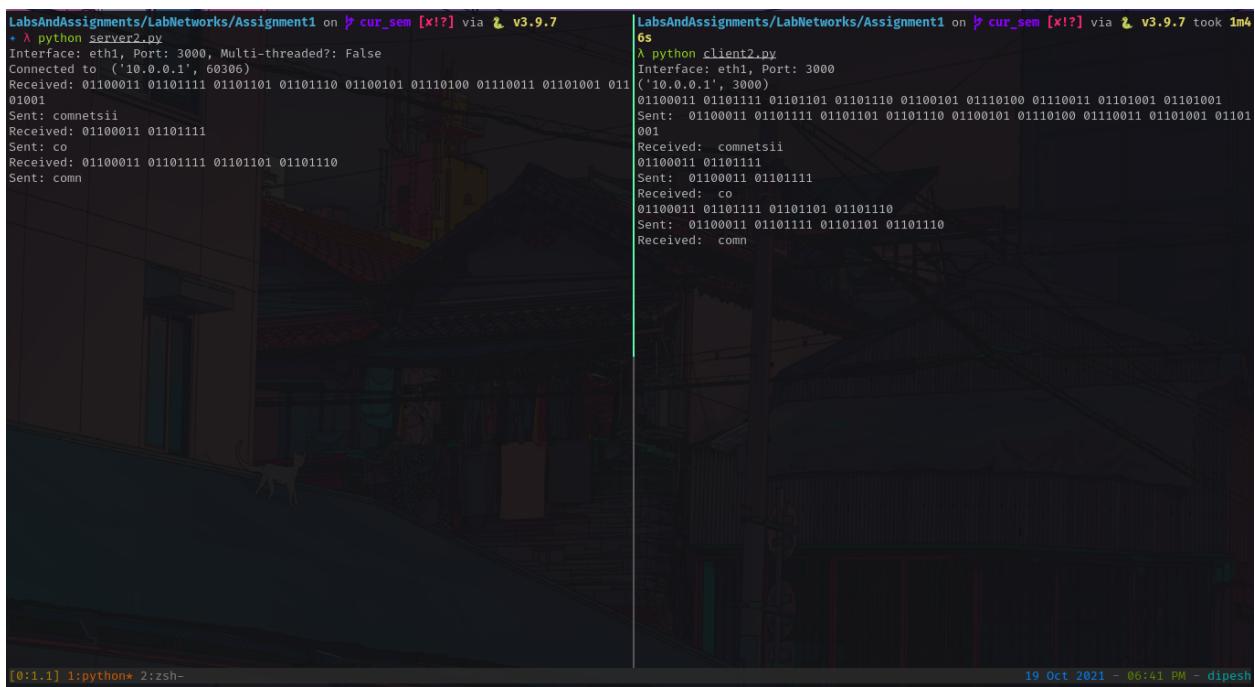
## Output(Single Threaded)

```
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via ℹ v3.9.7
λ python server1.py
Interface: lo, Port: 3000, Multi-threaded?: False
Connected to ('127.0.0.1', 42506)
Received: commetsii
Sent: 01100011 0101111 01101101 01101110 01100101 01110100 01110011 01101001 0110100
1
Received: comm
Sent: 01100011 01101111 01101101 01101110
Received: comnet
Sent: 01100011 01101111 01101101 01101110 01100101 01110100
Received: commetsii
Sent: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 0110100
1
Received: abcdef
Sent: 01100001 01100010 01100011 01100100 01100101 01100110
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via ℹ v3.9.7 took 20s
λ

LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via ℹ v3.9.7
λ python client1.py
Interface: lo, Port: 3000
('127.0.0.1', 3000)
commetsii
Sent: commetsii
Received: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001
1101001
comm
Sent: comm
Received: 01100011 01101111 01101101 01101110
comnet
Sent: comnet
Received: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001
1101001
abcdef
Sent: abcdef
Received: 01100001 01100010 01100011 01100100 01100101 01100110
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via ℹ v3.9.7 took 16s
λ

[0:3.1] 1:zsh- 2:zsh 3:python*
```

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



```
LabsAndAssignments/LabNetworks/Assignment1 on ✪ cur_sem [x!?] via ✲ v3.9.7
+ λ python server2.py
Interface: eth1, Port: 3000, Multi-threaded?: False
Connected to ('10.0.0.1', 60306)
Received: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 011001
01001
Sent: commetsii
Received: 01100011 01101111
Sent: co
Received: 01100011 01101111 01101101 01101110
Sent: comm
[0:1.1] 1:python* 2:zsh-
LabsAndAssignments/LabNetworks/Assignment1 on ✪ cur_sem [x!?] via ✲ v3.9.7 took 1m4
6s
λ python client2.py
Interface: eth1, Port: 3000
('10.0.0.1', 3000)
01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001
001
Received: commetsii
01100011 01101111
Sent: 01100011 01101111
Received: co
01100011 01101111 01101101 01101110
Sent: 01100011 01101111 01101101 01101110
Received: comm
```

- Running server2.py and client2.py

## Output(Multi Threaded)

```
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via 🐍 v3.9.7 took 5s
+ λ python server1.py --multi-threaded
Interface: lo, Port: 3000, Multi-threaded?: True
Connected to ('127.0.0.1', 42526)
Connected to ('127.0.0.1', 42536)
Received: comnetsii
Sent: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001
Received: comnetsii
Sent: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001
Received: abcdef
Sent: 01100001 01100010 01100011 01100100 01100101 01100110
Received: ab
Sent: 01100001 01100010

LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via 🐍 v3.9.7 took 47s
λ python client1.py
Interface: lo, Port: 3000
('127.0.0.1', 3000)
comnetsii
Sent: comnetsii
Received: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001
abcdef
Sent: abcdef
Received: 01100001 01100010 01100011 01100100 01100101 01100110
[0:1.2] 1:python* 2:zsh- 3:zsh
```

```
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via 🐍 v3.9.7
λ python client1.py
Interface: lo, Port: 3000
('127.0.0.1', 3000)
comnetsii
Sent: comnetsii
Received: 01100011 01101111 01101101 01101110 01100101 01110100 01110011 01101001 01101001
ab
Sent: ab
Received: 01100001 01100010
[0:1.2] 1:python* 2:zsh- 3:zsh
```

19 Oct 2021 - 07:04 PM - dipesh

- Multiple client1.py connecting to server1.py

```
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via 🐍 v3.9.7 took 2m1s
+ λ python server2.py --multi-threaded
Interface: eth1, Port: 3000, Multi-threaded?: True
Connected to ('10.0.0.1', 60382)
Connected to ('10.0.0.1', 60384)
Received: 01100011 01101111 01101101 01101110 01100101 01101000 01100011 01101001 01101001
Sent: comnetsii
Received: 01100011 01101111 01101101 01101110 01100101 01101000 01100011 01101001 01101001
Sent: comnetsii
Received: 01100011 01101111 01101101 01101110 01100101 01101000 01100011 01101001 01101001
Sent: com
Received: 01100011 01101111 01101101 01101110 01100101
Sent: come
Received: 01100011 01101111 01101101
Sent: com
[0:1.2] 1:python* 2:zsh- 3:zsh
```

```
LabsAndAssignments/LabNetworks/Assignment1 on ▶ cur_sem [x!?] via 🐍 v3.9.7 took 1m4
9s
λ python client2.py
Interface: eth1, Port: 3000
('10.0.0.1', 3000)
01100011 01101111 01101101 01101110 01100101 01101000 01100011 01101001 01101001
Sent: 01100011 01101111 01101101 01101110 01100101 01101000 01100011 01101001 01101001
01
Received: comnetsii
01100011 01101111 01101101 01101110 01100101
Sent: 01100011 01101111 01101101 01101110 01100101
Received: come
[0:1.2] 1:python* 2:zsh- 3:zsh
```

19 Oct 2021 - 07:06 PM - dipesh

- Multiple client2.py connecting to server2.py

