

Computer Networks

SEECS, NUST

Lab 3: Network Programming

Name: Muhammad Rizwan Khalid

Registration # 180459

Lab Title: Network Programming

1.0 Objectives:

After this lab, the students should be able to

- Explain the concepts of client server communication
- Setup client/server communication
- Use the sockets interface of Python programming language
- Implement simple Client / Server applications using UDP and TCP

2.0 Instructions:

- *Read carefully before starting the lab.*
- *These exercises are to be done individually.*
- *To obtain credit for this lab, you are supposed to complete the lab tasks and provide the source codes and the screen shot of your output in this document (please use red font color) and upload the completed document to your course's LMS site.*
- *Avoid plagiarism by copying from the Internet or from your peers. You may refer to source/ text but you must paraphrase the original work.*

3.0 Background:

3.1 Application programming interface:

An application programming interface (API) is a specification intended to be used as an interface by software components to communicate with each other. An API may include specifications for routines, data structures, object classes, and variables.

3.2 Network Application Programming Interface:

The place to start when implementing a network application is the 'interface exported by network'. Generally all operating systems provide an interface to its networking sub system. This interface is called as the 'Network Application Programming Interface' (Network API) or socket interface.

Computer Networks

SEECs, NUST

Lab 3: Network Programming

3.3 Network Sockets:

A **network socket** is an endpoint of an inter-process communication flow across a computer network. Today, most communication between computers is based on the Internet Protocol; therefore most network sockets are **Internet sockets**.

The socket is a special file in UNIX. The socket interface defines various operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket and so on. Any application uses a socket primitive to establish a connection between client and server.

A **socket address** is the combination of an IP address and a port number, much like one end of a telephone connection is the combination of a phone number and a particular extension. Based on this address, Internet sockets deliver incoming data packets to the appropriate application process or thread.

3.4 Socket API:

A socket API is an interface, usually provided by the operating system, that allows application programs to control and use network sockets. Internet socket APIs are usually based on the Berkeley sockets standard.

3.4.1 Berkeley Socket:

The following are some general information about Berkeley Sockets

1. Developed in the early 1980s at the University of California at Berkeley. Other major alternative was TLI (Transport Layer Interface). There are communications tools that are built on Berkeley sockets.
2. It is an API.
3. Its implementation usually requires kernel code.
4. Can use the UNIX read, write, close, select, etc. system calls.
5. Supports broadcast. This is where the same message may be delivered to multiple systems on a network without additional overhead.

Computer Networks

SEECs, NUST

Lab 3: Network Programming

6. Available on many UNIX system and somewhat available in WIN32.
7. Build for client/server development. That is having one system provide a service to other systems.

3.5 Client/Server Communication

At a basic level, network-based systems consist of a server, client, and a media for communication as shown in Figure 1. A computer running a program that makes a request for services is called client machine. A computer running a program that offers requested services from one or more clients is called server machine. The media for communication can be wired or wireless network.

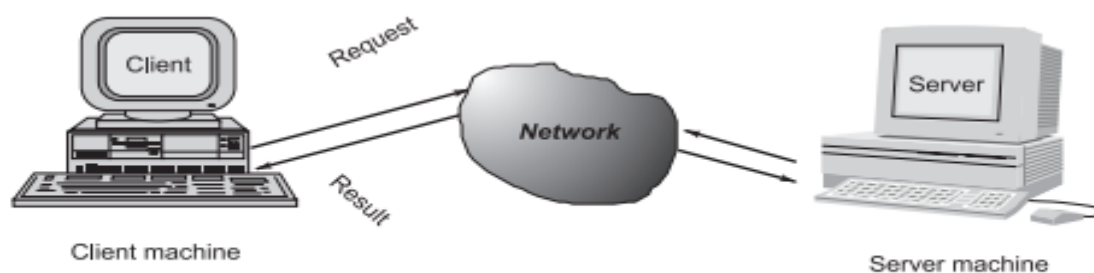


Figure 1. Client server communication

Generally, programs running on client machines make requests to a program (often called as server program) running on a server machine. They involve networking services provided by the transport layer, which is part of the Internet software stack, often called TCP/IP (Transport Control Protocol/Internet Protocol) stack, the transport layer comprises two types of protocols, TCP (Transport Control Protocol) and UDP (User Datagram Protocol)

3.6 Port Numbers

At any given time, multiple processes can be using any given transport layer protocol: UDP or TCP. The transport layer uses 16-bit integer *port numbers* to differentiate between these processes. When a client wants to contact a server, the client must identify the server with which it wants to communicate. The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer. Some ports have been reserved to support common/well known services:

- ftp 21/tcp
- telnet 23/tcp
- smtp 25/tcp

Computer Networks

SEECs, NUST

Lab 3: Network Programming

http 80/tcp,udp

User-level process/services generally use port number value ≥ 1024 .

4.0 Client-Server Socket Programming

We introduce UDP and TCP socket programming by way of a simple UDP application and a simple TCP application both implemented in Python.

We'll use the following simple 'Echo' client-server application to demonstrate socket programming for both UDP and TCP:

1. The client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts the characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

4.1 Socket programming using UDP

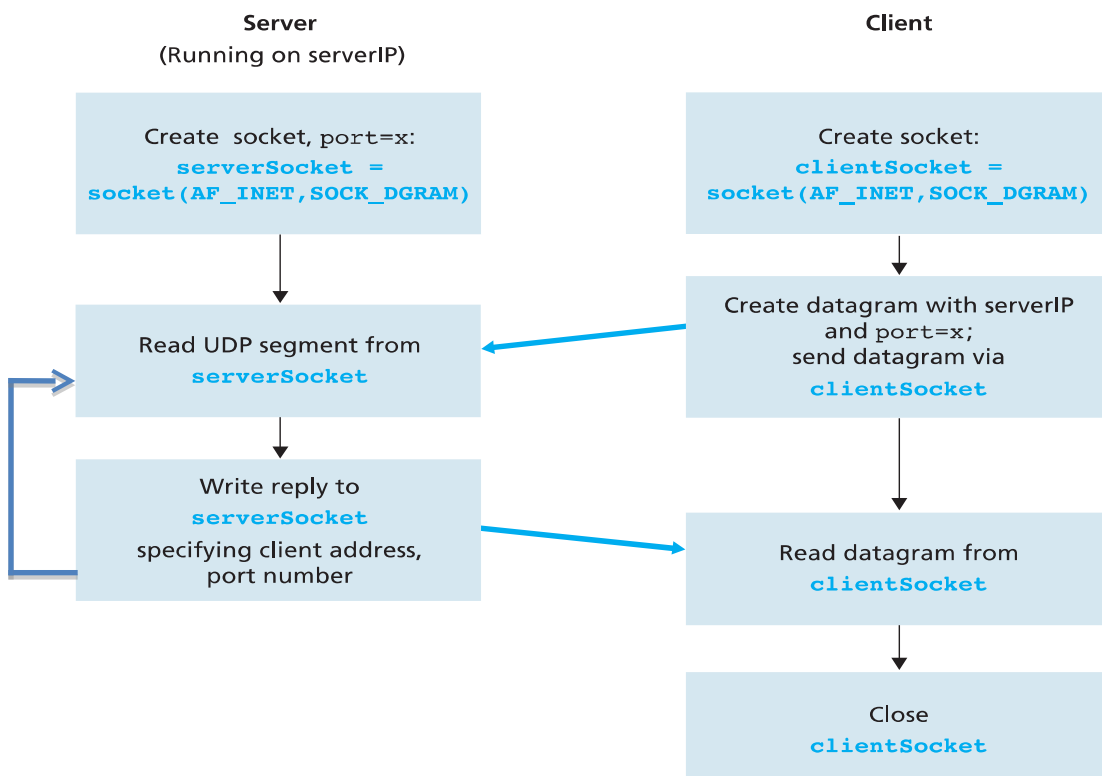


Figure 1: Socket programming using UDP

Figure 1 highlights the main socket-related activity of the client and server that communicate over the UDP transport service.

Now let's take a look at the client-server program pair for a UDP implementation of this simple application. We'll begin with the UDP client, which will send a simple application-level message to the server. In order for the server to be able to receive and reply to the client's message, it must be ready and running— that is, it must be running as a process before the client sends its message.

The client program is called UDPClient.py, and the server program is called UDPServer.py. In order to emphasize the key issues, we intentionally provide code that is minimal. "Good code" would certainly have a few more auxiliary lines, in particular for handling error cases. For this application, we have arbitrarily chosen 12000 for the server port number.

UDP:

Don't establish connection, directly sent message.

TCP:

First establish connection then sent message.

UDPClient.py

```
=====
=====

from socket import * # import socket module

serverIP = 'hostname' # replace with IP address of the server

serverPort = 25000 #port where server is listening

clientSocket = socket(AF_INET, SOCK_DGRAM) // IPV4 and UDP used hu rha ha

message = raw_input('Input lowercase sentence:')

clientSocket.sendto(message,(serverIP, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

print modifiedMessage # print the received message

clientSocket.close() # Close the socket
```

Computer Networks

SEECs, NUST

Lab 3: Network Programming

=====

Details of important lines in code follows;

clientSocket = socket(AF_INET, SOCK_DGRAM)

This line creates the client's socket, called clientSocket. The first parameter indicates the address family; in particular, AF_INET indicates that the underlying network is using IPv4. The second parameter indicates that the socket is of type SOCK_DGRAM, which means it is a UDP socket (rather than a TCP socket). Note that we are not specifying the port number of the client socket when we create it; we are instead letting the operating system do this for us.

message = raw_input('Input lowercase sentence:')

raw_input() is a built-in function in Python. When this command is executed, the user at the client is prompted with the words "Input lowercase sentence:" The user then uses the keyboard to input a line, which is put into the variable message. Now that we have a socket and a message, we will want to send the message through the socket to the destination host.

clientSocket.sendto(message,(serverIP, serverPort))

In the above line, the method sendto() attaches the destination address (serverIP, serverPort) to the message and sends the resulting packet into the process's socket, clientSocket. Sending a client-to-server message via a UDP socket is that simple! After sending the packet, the client waits to receive data from the server.

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

With the above line, when a packet arrives from the Internet at the client's socket, the packet's data is put into the variable modifiedMessage and the packet's source address is put into the variable serverAddress. The variable serverAddress contains both the server's IP address and the server's port number. The program UDPClient doesn't actually need this server address information, since it already knows the server address from the outset; but this line of Python provides the server address nevertheless. The method recvfrom also takes the buffer size 2048 as input.

UDPServer.py

=====

=====

from socket import *

Computer Networks

SEECs, NUST

Lab 3: Network Programming

```
serverPort = 25000
```

```
serverIP="10.99.26.161"
```

```
serverSocket = socket(AF_INET, SOCK_DGRAM)
```

```
serverSocket.bind((serverIP, serverPort))
```

```
print "The server is ready to receive"
```

```
while 1:
```

```
    message, clientAddress = serverSocket.recvfrom(2048)
```

```
    modifiedMessage = message.upper()
```

```
    serverSocket.sendto(modifiedMessage, clientAddress)
```

=====

Note that the beginning of UDPServer is similar to UDPClient. The first line of code that is significantly different from UDPClient is:

```
serverSocket.bind((serverIP, serverPort))
```

The above line binds (that is, assigns) the port number 12000 to the server's socket. Thus in UDPServer, the code is explicitly assigning a port number to the socket. In this manner, when anyone sends a packet to port 12000 at the IP address of the server, that packet will be directed to this socket. UDPServer then enters a while loop; the while loop will allow UDPServer to receive and process packets from clients indefinitely. In the while loop, UDPServer waits for a packet to arrive.

```
message, clientAddress = serverSocket.recvfrom(2048)
```

When a packet arrives at the server's socket, the packet's data is put into the variable message and the packet's source address is put into the variable clientAddress. The variable clientAddress contains both the client's IP address and the client's port number. Here, UDPServer *will* make use of this address information, as it provides a return address, similar to the return address with ordinary postal mail. With this source address information, the server now knows to where it should direct its reply.

```
modifiedMessage = message.upper()
```

This line is the heart of our simple application. It takes the line sent by the client and uses the method upper() to capitalize it.

Computer Networks

SEECs, NUST

Lab 3: Network Programming

serverSocket.sendto(modifiedMessage, clientAddress)

This last line attaches the client's address (IP address and port number) to the capitalized message, and sends the resulting packet into the server's socket. After the server sends the packet, it remains in the while loop, waiting for another UDP packet to arrive (from any client running on any host).

To test the pair of programs, you install and run UDPClient.py in one host and UDPServer.py in another host. Be sure to include the proper hostname or IP address of the server in UDPClient.py. Next, you execute UDPServer.py, the server program, in the server host. This creates a process in the server that idles until some client contacts it. Then you execute UDPClient.py, the client program, in the client. This creates a process in the client. Finally, to use the application at the client, you type a sentence followed by a carriage return.

4.1.1 Lab Task 1: A useful Python UDP client/server application.

Modify the UDPClient program such that the UDPClient is able to calculate the Application level Round Trip Time (RTT) for the communication between the Client and the Server. The Client should also print the time when Request is send and time when the Reply is received in human readable form.

Client Side:

```
import time

from socket import * # import socket module

serverIP = '10.3.93.197' # replace with IP address of the server

serverPort = 25000 # port where server is listening

clientSocket = socket(AF_INET, SOCK_DGRAM) # IPV4 and UDP used hu rha ha

message = raw_input("Enter your name")

time1 = time.clock()

clientSocket.sendto(message, (serverIP, serverPort))

returnMessage, serverAddress = clientSocket.recvfrom(2048)

print returnMessage

time2 = time.clock()

print time2 - time1

clientSocket.close()
```


Computer Networks

SEECs, NUST

Lab 3: Network Programming

```
C:\Windows\system32\cmd.exe
17/09/2018 03:12 PM <DIR> .
17/09/2018 03:12 PM <DIR> ..
07/09/2018 10:23 AM <DIR> .PyCharmCE2017.2
07/09/2018 09:06 AM <DIR> Contacts
24/09/2018 02:55 PM <DIR> Desktop
17/09/2018 02:58 PM <DIR> Documents
24/09/2018 02:36 PM <DIR> Downloads
07/09/2018 09:06 AM <DIR> Favorites
07/09/2018 09:06 AM <DIR> Links
07/09/2018 09:06 AM <DIR> Music
13/09/2018 10:19 AM <DIR> OneDrive
07/09/2018 09:09 AM <DIR> Pictures
07/09/2018 09:06 AM <DIR> Saved Games
07/09/2018 09:07 AM <DIR> Searches
07/09/2018 09:06 AM <DIR> Videos
0 File(s) 0 bytes
15 Dir(s) 56,561,278,976 bytes free

C:\Users\domain1>cd C:\Users\domain1\Desktop

C:\Users\domain1\Desktop>python task1.py
'python' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\domain1\Desktop>task1.py
Enter your nameroshan
ROSHAN
0.0121272934602

C:\Users\domain1\Desktop>
```

Server Side:

```
import time

from socket import* #import socket module

serverIP='10.3.93.197' #replace with IP address of the server

serverPort=25000 #port where server is listening

serverSocket = socket(AF_INET, SOCK_DGRAM)

serverSocket.bind((serverIP, serverPort))

print 'The server is ready to receive'

while 1:

    message, clientAddress = serverSocket.recvfrom(2048)

    modifiedMessage = message.upper()

    serverSocket.sendto(modifiedMessage, clientAddress)
```

Computer Networks

SEECs, NUST

Lab 3: Network Programming

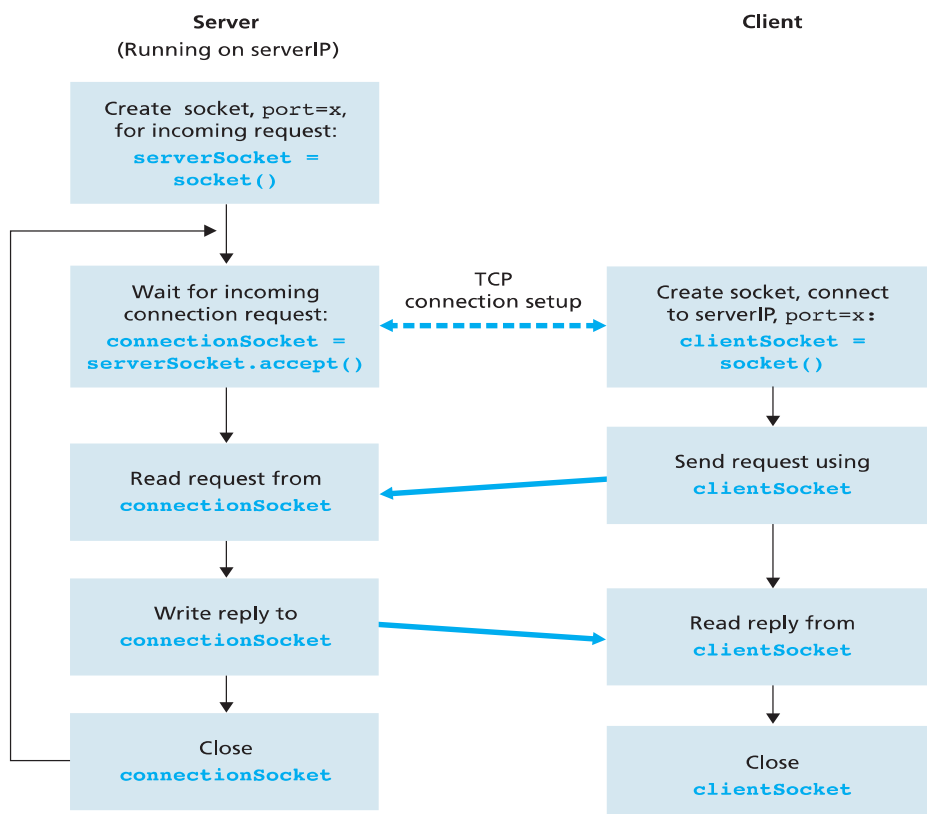
```
socket.error: [Errno 10054] An existing connection was forcibly closed by the remote host

C:\Users\domain1>python C:\Users\domain1\Desktop\UDPServer.py
File "C:\Users\domain1\Desktop\UDPServer.py", line 6
SyntaxError: Non-ASCII character '\xe2' in file C:\Users\domain1\Desktop\UDPServer.py on line 6, but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details

C:\Users\domain1>python C:\Users\domain1\Desktop\UDPServer.py
The server is ready to receive
```

4.2 Socket programming using TCP

We use the same simple client-server application to demonstrate socket programming with TCP: The client sends one line of data to the server, the server capitalizes the line and sends it back to the client. Figure 2 highlights the main socket-related activity of the client and server that communicate over the TCP transport service.



Computer Networks

SEECs, NUST

Lab 3: Network Programming

TCPClient.py

```
from socket import *

serverIP = 'servername'

serverPort = 12000

clientSocket = socket(AF_INET, SOCK_STREAM)

sentence = raw_input('Input lowercase sentence:')

clientSocket.connect((serverIP,serverPort))

clientSocket.send(sentence)

modifiedSentence = clientSocket.recv(1024)

print 'From Server:', modifiedSentence

clientSocket.close()
```

clientSocket.connect((serverName,serverPort))

Before the client can send data to the server (or vice versa) using a TCP socket, a TCP connection must first be established between the client and server. The above line initiates the TCP connection between the client and server. The parameter of the connect() method is the address of the server side of the connection. After this line of code is executed, the three-way handshake is performed and a TCP connection is established between the client and server.

clientSocket.send(sentence)

Note that in this send call, the program does *not* explicitly create a packet and attach the destination address to the packet, as was the case with UDP sockets. Instead the client program simply drops the bytes in the string sentence into the TCP connection.

Computer Networks

SEECs, NUST

Lab 3: Network Programming

TCPServer.py

```
from socket import *

serverPort = 12000

serverSocket = socket(AF_INET,SOCK_STREAM)

serverSocket.bind((' ',serverPort))

serverSocket.listen(1)

print 'The server is ready to receive'

while 1:

    connectionSocket, addr = serverSocket.accept() //

    sentence = connectionSocket.recv(1024)

    capitalizedSentence = sentence.upper()

    connectionSocket.send(capitalizedSentence)

    connectionSocket.close()
```

Let's now take a look at the lines that differ significantly from UDPServer and TCPClient.

serverSocket.listen(1)

This line has the server listen for TCP connection requests from the client. The parameter specifies the maximum number of queued connections (at least 1).

connectionSocket, addr = serverSocket.accept()

When a client knocks on this door, the program invokes the `accept()` method for `serverSocket`, which creates a new socket in the server, called `connectionSocket`, dedicated to this particular client. The client and server then complete the handshaking, creating a TCP connection between the client's `clientSocket` and the server's `connectionSocket`. With the TCP connection established, the client and server can now send bytes to each other over the connection. With TCP, all bytes sent from one side not are

Computer Networks

SEECS, NUST

Lab 3: Network Programming

not only guaranteed to arrive at the other side but also guaranteed to arrive in order.

connectionSocket.close()

In this program, after sending the modified sentence to the client, we close the connection socket. But since serverSocket remains open, another client can now knock on the door and send the server a sentence to modify.

4.2.1 Lab Task 2: A useful Python TCP client/server application.

Modify the TCPClient program such that the TCPClient is able to calculate the Application level Round Trip Time (RTT) for the communication between the Client and the Server. The Client should also print the time when connection request is send and time when the Reply (capitalized words) is received in human readable form.

Client Side:

```
import time

from socket import * # import socket module

serverIP = '10.3.93.197' # replace with IP address of the server

serverPort = 25000 # port where server is listening

clientSocket = socket(AF_INET, SOCK_STREAM) # IPV4 and UDP used hu rha ha

message = raw_input("Enter your name ")

clientSocket.connect((serverIP,serverPort))

time1 = time.clock()

clientSocket.send(message)

returnMessage = clientSocket.recv(2048)

print returnMessage

time2 = time.clock()

print time2-time1

clientSocket.close()
```

Computer Networks

SEECs, NUST

Lab 3: Network Programming

```
C:\Users\domain1\Desktop>task1.py
Enter your name abbas
Traceback (most recent call last):
  File "C:\Users\domain1\Desktop\task1.py", line 10, in <module>
    returnMessage = clientSocket.recv(2048)
socket.error: [Errno 10054] An existing connection was forcibly closed by the remote host

C:\Users\domain1\Desktop>task1.py
Enter your name abbas
Traceback (most recent call last):
  File "C:\Users\domain1\Desktop\task1.py", line 7, in <module>
    clientSocket.connect((serverIP,serverPort))
  File "C:\Python27\lib\socket.py", line 228, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 10061] No connection could be made because the target machine is refusing connections

C:\Users\domain1\Desktop>task1.py
Enter your name abbas
ABBAS
0.00115776706671
```

Server Side:

import time

from socket import #import socket module*

serverPort=25000 #port where server is listening

serverSocket = socket(AF_INET,SOCK_STREAM)

serverSocket.bind(('',serverPort))

serverSocket.listen(1)

print 'The server is ready to receive'

while 1:

connectionSocket, addr = serverSocket.accept()

sentence = connectionSocket.recv(1024)

capitalizedSentence = sentence.upper()

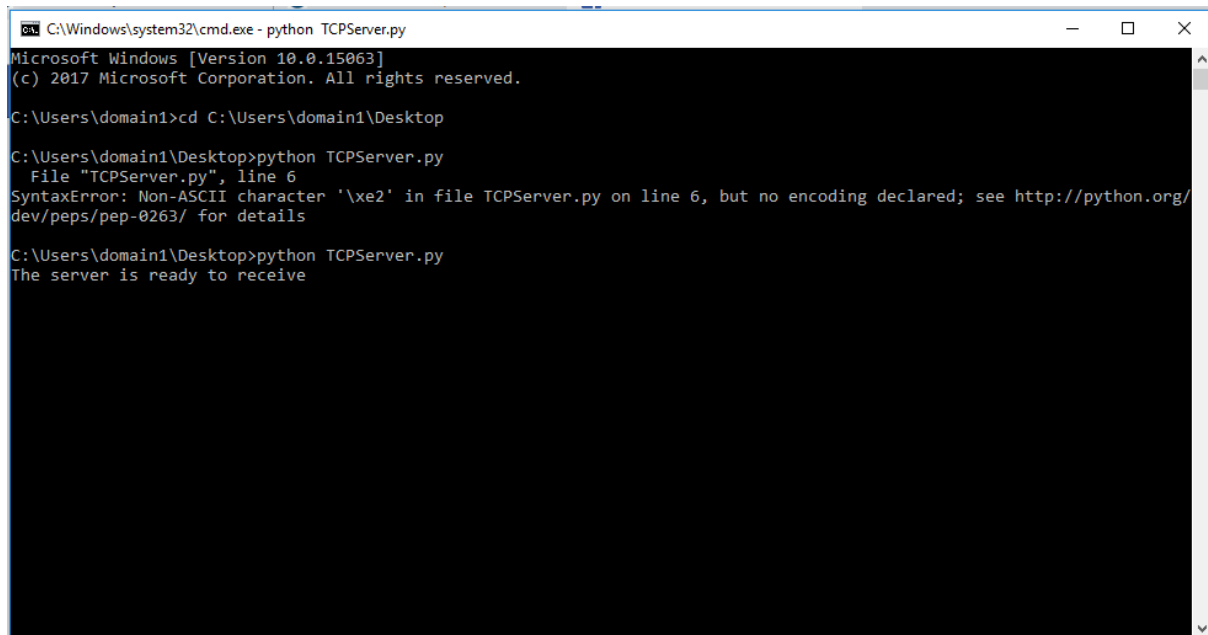
connectionSocket.send(capitalizedSentence)

connectionSocket.close()

Computer Networks

SEECs, NUST

Lab 3: Network Programming



```
C:\Windows\system32\cmd.exe - python TCPServer.py
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\domain1>cd C:\Users\domain1\Desktop

C:\Users\domain1\Desktop>python TCPServer.py
File "TCPServer.py", line 6
SyntaxError: Non-ASCII character '\xe2' in file TCPServer.py on line 6, but no encoding declared; see http://python.org/dev/peps/pep-0263/ for details

C:\Users\domain1\Desktop>python TCPServer.py
The server is ready to receive
```

4.2.2 Lab Task 3: Compare the values of the RTT for both the UDP and TCP. Which one has got higher RTT? Why?

Answer:

RTT is the round trip time, the time for sending request and receiving the response. RTT value of UDP is '0.01212' and the RTT value of TCP is '0.001155' as seen from the above screenshots. Clearly the RTT value of UDP is higher than RTT value of TCP. In case of TCP, it sends the request to the server, server acknowledges the request and then packets are forwarded to the server. In case of UDP, there is no connection establishment and acknowledgment.

4.2.3 Lab Task 4: What happens when your client (both UDP and TCP) tries to send data to a non-existent server?

Answer:

When we try to send data to the non-existent server in the UDP, an error is invoked about the host unavailability. But in the case TCP Client tries to send the data to non-existent server, no-error is invoked and time-out message appears. It is because in case of TCP, first the connection is established and client waits for acknowledgment from server. If there is no host, there will be no response. So, timeout message will appear.