Computer Networks-Lab 14

Instructor: Hurmat Hidayat

CL30001 – Computer Networks-Lab

SEMESTER Fall 2022

NATIONAL UNIVERSTIY OF COMPUTER AND EMERGING SCIENCES, FAST- PESHAWAR CAMPUS
Department of Computer Science & Software Engineering

Computer Networks - Lab 14

OBJECTIVES

After these Lab students shall be able to understand

- Port Vs Socket
- Socket Programming in Python

PRE-LAB READING ASSIGNMENT

Remember the delivered lecture carefully.

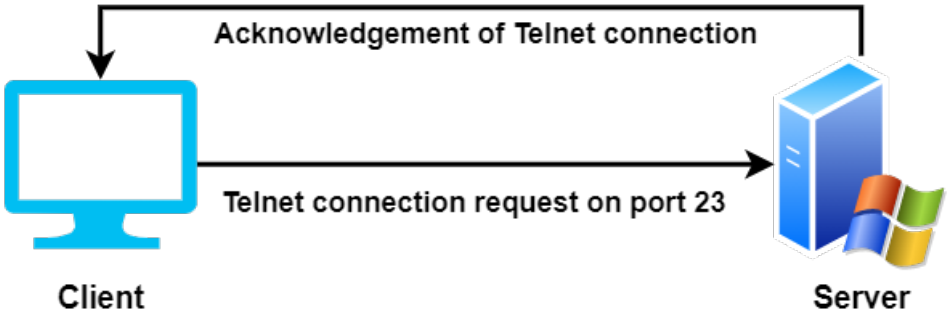
Table of Contents

Difference Between a Port and a Socket

Port is a part of the transport layer and helps in network communication. **A port is a logical identifier assigned to a process in order to identify that process uniquely in a network system.** When two network devices communicate, they do so by sending packets to each other. Each packet received by a receiver device contains a port number that uniquely identifies the process where the packet needs to be sent.

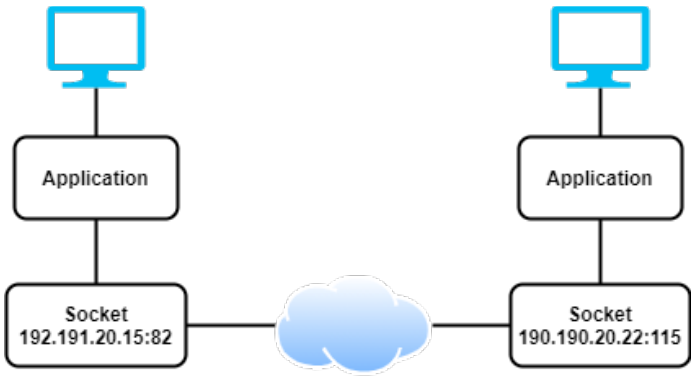
Not all the network protocol uses a port for communication. For example, ICMP doesn't use a port. On the other hand, protocols like TCP, UDP, HTTP utilize a port for communication.

Let's look at an example. A client computer is requesting the server for a virtual connection with the port number. Telnet is a well-known protocol for establishing a remote connection over a TCP/IP and it uses port . Hence, the server acknowledges the request from the client and start a telnet connection:



Two processes that are running on a computer or running on two different systems can communicate via a socket. A **socket works as an inter-process communicator and seen as the endpoint of the process communication.** For communication, the socket uses a file descriptor and is mainly employed in client-server applications.

A socket consists of the IP address of a system and the port number of a program within the system. The IP address corresponds to the system and the port number corresponds to the program where the data needs to be sent:



Sockets can be classified into three categories: stream, datagram, and raw socket. Stream sockets use connection-oriented network point to send and receive data. This type of sockets generally utilizes TCP to permit processes to communicate with each other.
Datagram sockets use connectionless network protocols like UDP to allow process communication. Raw sockets are datagram oriented and allow the processes to use ICMP for communication purpose.

Port	Socket
Port specifies a number that is used by a program in a computer.	A socket is a combination of IP address and port number.
A program running on different computers can use the same port number. Hence port numbers can't be used to identify a computer uniquely.	It identifies a computer as well as a program within the computer uniquely.
Port number is used in the transport layer.	Sockets are involved in the application layer. A socket is an interface between the transport and application layer.
Port uses a socket to drop the data to a correct application.	A server and a client uses a socket to keep an eye on the data request and responses.

The most common type of socket applications are client-server applications, where one side acts as the server and waits for connections from clients. This is the type of application that you'll be creating in this Lab.

Socket API Overview

Python's socket module provides an interface to the Berkeley sockets API. This is the module that you'll use in this tutorial.

The primary socket API functions and methods in this module are:

- socket()
- .bind()
- .listen()
- .accept()
- .connect()
- .connect_ex()
- .send()
- .recv()

- .close()

TCP Sockets

You're going to create a socket object using socket.socket(), specifying the socket type as socket.SOCK_STREAM. When you do that, the default protocol that's used is the Transmission Control Protocol (TCP). This is a good default and probably what you want.

Why should you use TCP? The Transmission Control Protocol (TCP):

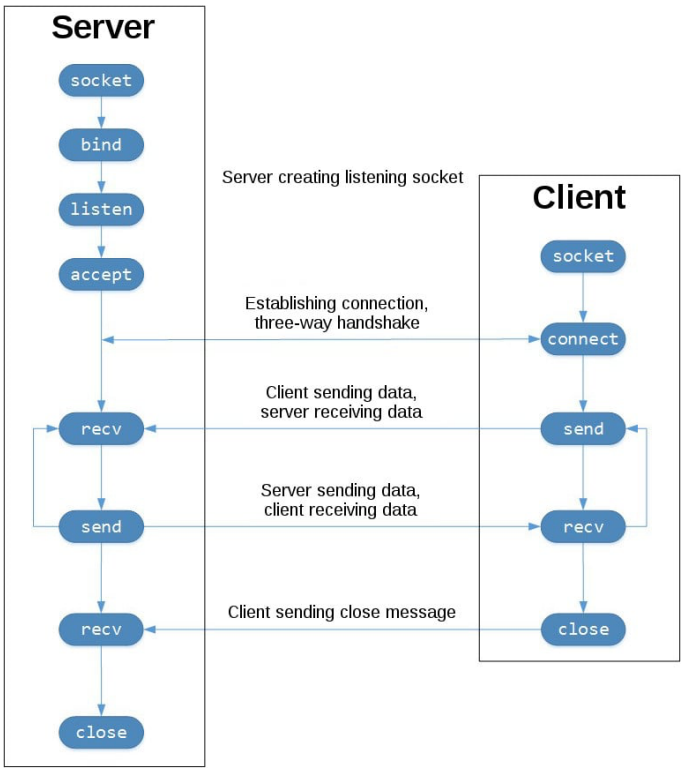
- **Is reliable:** Packets dropped in the network are detected and retransmitted by the sender.
- **Has in-order data delivery:** Data is read by your application in the order it was written by the sender.

In contrast, User Datagram Protocol (UDP) sockets created with socket.SOCK_DGRAM aren't reliable, and data read by the receiver can be out-of-order from the sender's writes.

Why is this important? Networks are a best-effort delivery system. There's no guarantee that your data will reach its destination or that you'll receive what's been sent to you.

Network devices, such as routers and switches, have finite bandwidth available and come with their own inherent system limitations. They have CPUs, memory, buses, and interface packet buffers, just like your clients and servers. TCP relieves you from having to worry about packet loss, out-of-order data arrival, and other pitfalls that invariably happen when you're communicating across a network.

To better understand this, check out the sequence of socket API calls and data flow for TCP:



The left-hand column represents the server. On the right-hand side is the client.
Starting in the top left-hand column, note the API calls that the server makes to set up a "listening" socket:

- socket()
- .bind()
- .listen()
- .accept()

A listening socket does just what its name suggests. It listens for connections from clients. When a client connects, the server calls .accept() to accept, or complete, the connection.
The client calls .connect() to establish a connection to the server and initiate the three-way handshake. The handshake step is important because it ensures that each side of the connection is reachable in the network, in other words that the client can reach the server and vice-versa. It may be that only one host, client, or server can reach the other.

In the middle is the round-trip section, where data is exchanged between the client and server using calls to .send() and .recv().

At the bottom, the client and server close their respective sockets.

Python Socket Example

We have said earlier that a socket client requests for some resources to the socket server and the server responds to that request. So we will design both server and client model so that each can communicate with them. The steps can be considered like this.

1. Python socket server program executes at first and wait for any request
2. Python socket client program will initiate the conversation at first.
3. Then server program will response accordingly to client requests.
4. Client program will terminate if user enters “bye” message. Server program will also terminate when client program terminates, this is optional and we can keep server program running indefinitely or terminate with some specific command in client request.

Python Socket Server

We will save python socket server program as socket_server.py. To use python socket connection, we need to import **socket** module. Then, sequentially we need to perform some task to establish connection between server and client. We can obtain host address by using socket.gethostname() function. It is recommended to user port address above 1024 because port number lesser than 1024 are reserved for standard internet protocol. See the below python socket server example code, the comments will help you to understand the code.

```
import socket

def server_program():

    # get the hostname

    host = socket.gethostname()

    port = 5000  # initiate port no above 1024

    server_socket = socket.socket()  # get instance

    # look closely. The bind() function takes tuple as argument

    server_socket.bind((host, port))  # bind host address and port
together

    # configure how many client the server can listen simultaneously

    server_socket.listen(2)

    conn, address = server_socket.accept()  # accept new connection

    print("Connection from: " + str(address))

    while True:
```

```
        # receive data stream. it won't accept data packet greater than
1024 bytes

        data = conn.recv(1024).decode()

        if not data:

            # if data is not received break

            break

        print("from connected user: " + str(data))

        data = input(' -> ')

        conn.send(data.encode())  # send data to the client



    conn.close()  # close the connection



if __name__ == '__main__':

    server_program()
```

So our python socket server is running on port 5000 and it will wait for client request. If you want server to not quit when client connection is closed, just remove the if condition and break statement. Python while loop is used to run the server program indefinitely and keep waiting for client request.

Python Socket Client

We will save python socket client program as socket_client.py. This program is similar to the server program, except binding. The main difference between server and client program is, in server program, it needs to bind host address and port address together. See the below python socket client example code, the comment will help you to understand the code.

```
import socket
```

```
def client_program():
    host = socket.gethostname()  # as both code is running on same pc
    port = 5000  # socket server port number

    client_socket = socket.socket()  # instantiate
    client_socket.connect((host, port))  # connect to the server

    message = input(" -> ")  # take input

    while message.lower().strip() != 'bye':
        client_socket.send(message.encode())  # send message
        data = client_socket.recv(1024).decode()  # receive response

        print('Received from server: ' + data)  # show in terminal

        message = input(" -> ")  # again take input

    client_socket.close()  # close the connection


if __name__ == '__main__':
    client_program()
```

Python Socket Programming Output

To see the output, first run the socket server program. Then run the socket client program. After that, write something from client program. Then again write reply from server program. At last, write **bye** from client program to terminate both program. Below short video will show how it worked on my test run of socket server and client example programs.

```
pankaj$ python3.6 socket_server.py
Connection from: ('127.0.0.1', 57822)
from connected user: Hi
 -> Hello
from connected user: How are you?
 -> Good
from connected user: Awesome!
 -> Ok then, bye!
pankaj$
```

```
pankaj$ python3.6 socket_client.py
 -> Hi
Received from server: Hello
 -> How are you?
Received from server: Good
 -> Awesome!
Received from server: Ok then, bye!
 -> Bye
pankaj$
```

Lab 14 Task (Review):

- **Consider the FAST NU Peshawar Campus. Identify the requirements and design a Network topology in packet tracer.**
  - **Write down the requirements in a document**
  - **Design the topology in packet tracer**

Note: 2 Bonus points are reserved for subnetting.