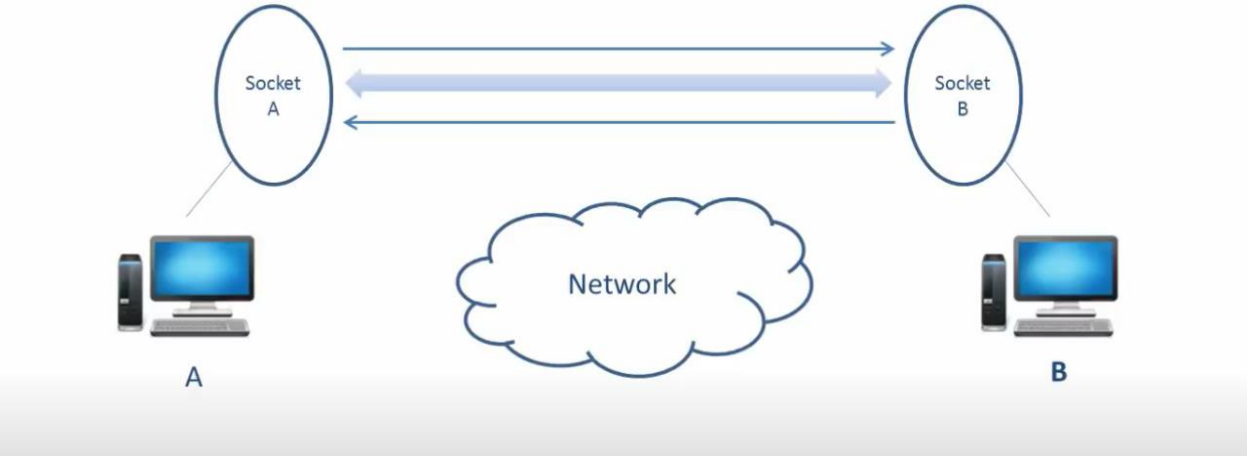
Chat Server using Socket Programming

Sockets

Sockets are the backbone of networking. They make the transfer of information possible between two different programs or devices. For example, when we open up a browser, we as clients are creating a connection to the server for the transfer of information. A single network will have two sockets, one for each communicating device or program. **These sockets are a combination of an IP address and a Port**. A single device

can have ’n’ number of sockets based on the port number that is being used.



Task Overview

👉 Create your own Chat Servers and establish a network to transfer data using Socket Programing by creating both Server and Client machine as Sender and Receiver both. Do this program using UDP data transfer protocol.

👉 Use multi-threading concept to get and receive data parallelly from both the Server Sides. Observe the challenges that you face to achieve this using UDP.

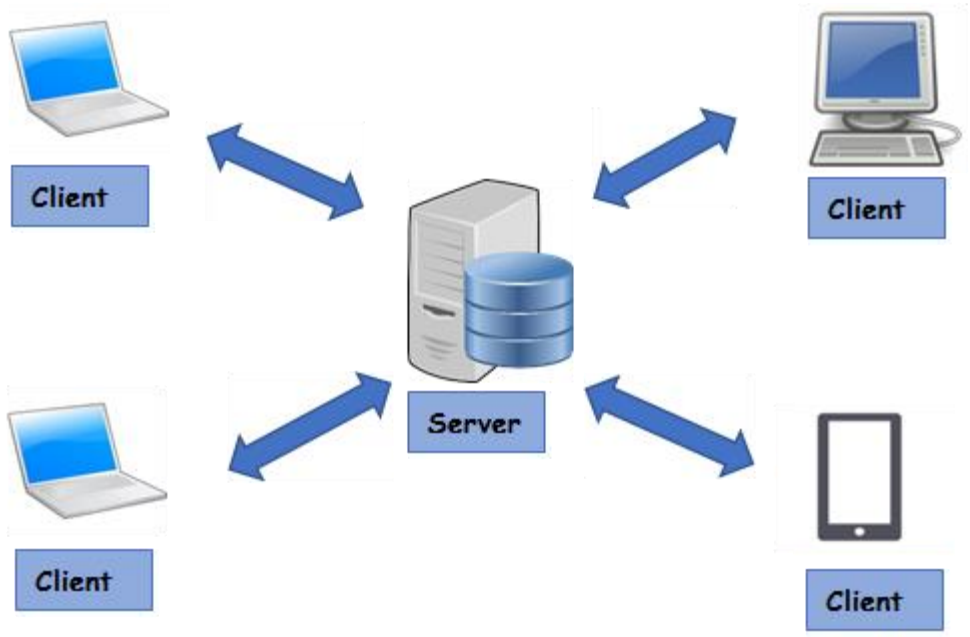
To achieve Socket Programming in Python, we will need to import the **socket** module. This module consists of built-in methods that are required for creating sockets and help them associate with each other.

What is a Server?

A server is either a program, a computer, or a device that is devoted to managing network resources. Servers can either be on the same device or computer or locally connected to other devices and computers or even remote. There are various types of servers such as database servers, network servers, print servers, etc.

What is a Client?

A client is either a computer or software that receives information or services from the server. In a client-server module, clients requests services from servers. The best example is a web browser such as Google Chrome, Firefox, etc.



Common Methods

socket(): This method is used to create the socket and takes two arguments first is a family or domain like **AF_INET** (IPv4) or **INET6** (IPv6) and the second defines the type of sockets like **SOCK_STREAM** (TCP) or **SOCK_DGRAM** (UDP).

bind(): This method is used to bind your socket with a specific host and port which will be passed as an argument to this function and that means your socket will be sitting at a specific location where the client socket can send its data.

recvfrom(): This method can be used with a UDP server to receive data from a UDP client or it can be used with a UDP client to receive data from a UDP server. It accepts a positional parameter called ***bufsize*** which is the number of bytes to be read from the **UDP socket.** It returns a byte object read from a UDP socket and the address of the client socket as a tuple.

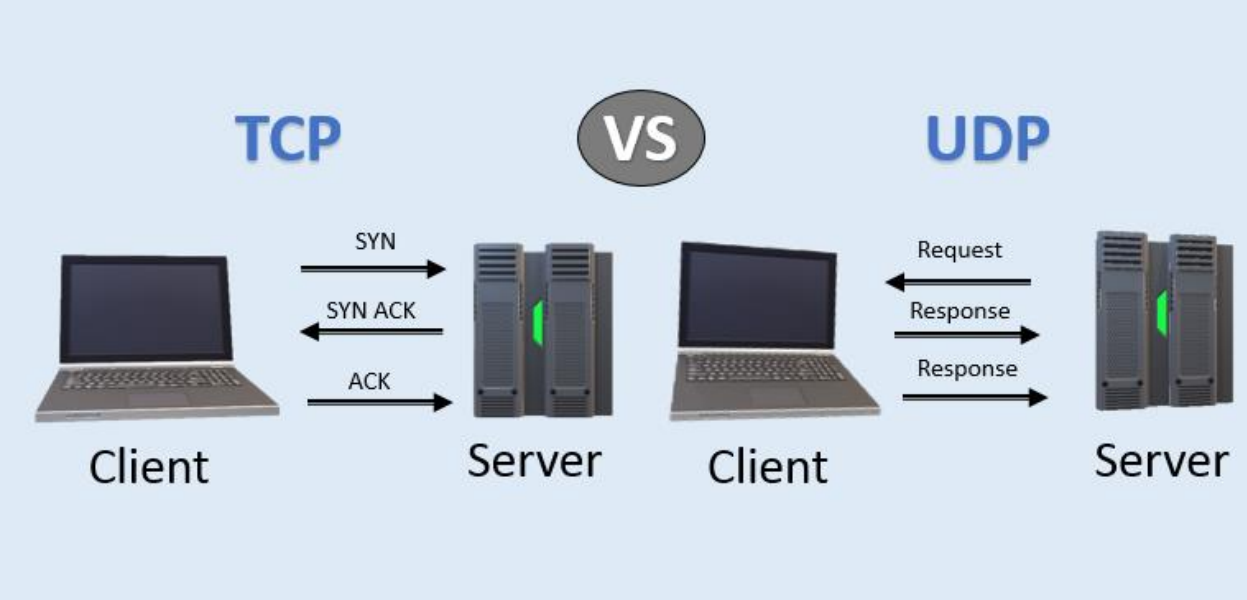
sendto(): It is a method of Python's socket class that is used to send datagrams to a UDP socket. The communication could be from either side. It could be from client to server or from the server to a client. The data to be sent must be in bytes format. If the data is in string format, the **str. encode()** method can be used to convert the strings to bytes. We must also pass a tuple consisting of IP address and port number.

TCP vs UDP

TCP/IP helps us to determine how a specific computer should be connected to the internet and how we can transmit data between them. It helps us to create a virtual network when multiple computer networks are connected.

TCP/IP stands for Transmission Control Protocol/ Internet Protocol. It is specifically designed as a model to offer a highly reliable and end-to-end byte stream over unreliable internetwork.

UDP uses a simple transmission method without implied hand-shaking dialogues for ordering, reliability, or data integrity. UDP also assumes that error checking and correction is not important or performed in the application, to avoid the overhead of such processing at the network interface level. It is also compatible with packet broadcasts and multicasting.



Features of TCP

- Delivery Acknowledgements
- Retransmission
- Delays transmission when the network is congested
- Easy Error detection

Features of UDP

- Supports bandwidth-intensive applications that tolerate packet loss
- Less delay
- It sends the bulk quantity of packets.
- Possibility of the Data loss
- Allows small transaction (DNS lookup)

UDP Disadvantages:

- Data corruption is a common occurrence on the Internet, UDP has a primitive form of error detection.
- No compensation for lost packets.
- Packets can arrive out of order.
- No congestion control.

Conclusion: UDP may be lightweight, but not that reliable.

Now let's jump right into the solution!!!

For the practical, I would be using two systems. One is Windows 10 and the other is RHEL-8.

The IP address of my Windows machine (**192.168.137.1**)

```
Wireless LAN adapter Local Area Connection* 1:

   Media State . . . . . . . . . . . : Media disconnected
   Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 2:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::d86b:dde0:4e66:1417%21
   IPv4 Address. . . . . . . . . . . : 192.168.137.1
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . :

Ethernet adapter Bluetooth Network Connection:
```

The IP of my RHEL-8 VM (192.168.99.130)

root@ansible_controller:~

```
[root@ansible_controller ~]# ifconfig enp0s8
enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.99.130  netmask 255.255.255.0  broadcast 192.168.99.255
        inet6 fe80::1072:608d:1ff2:482e  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:82:e3:49  txqueuelen 1000  (Ethernet)
        RX packets 76  bytes 8494 (8.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 81  bytes 15335 (14.9 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

[root@ansible_controller ~]#
```

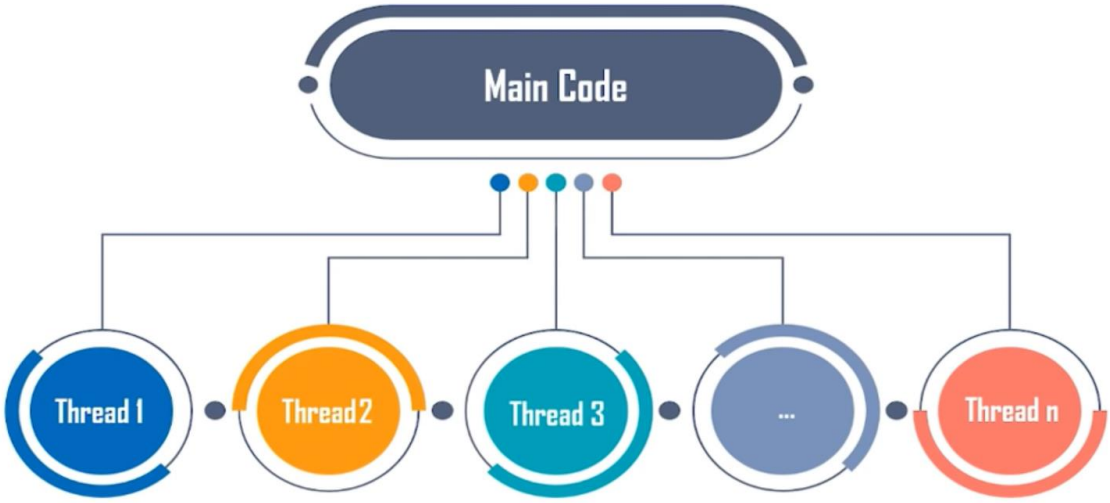
Code

```
import socket

from threading import *

def receive(ip,port):

      myp = socket.SOCK_DGRAM
      afn = socket.AF_INET
      s = socket.socket(afn,myp)
      s.bind((ip,port))
      while True:

            x = s.recvfrom(1024)
            print("\t\t\tReceived Message: ",x[0].decode())

def send(ip,port):

      myp = socket.SOCK_DGRAM
      afn = socket.AF_INET
      s = socket.socket(afn,myp)
      while True:

            msg = input()
            s.sendto(msg.encode(), (ip,port))

WinIP = input("Enter your IP: ")

WinPort = int(input("Enter your Port: "))

LinIP = input("Enter target IP: ")

LinPort = int(input("Enter target Port: "))

print("-----------Welcome to Python Chat------------")

receiveThread = Thread( target = receive , args = (WinIP,WinPort))
senderThread = Thread( target = send , args = (LinIP,LinPort))

receiveThread.start()

senderThread.start()
```

MultiThreading

A **thread** is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS. In simple words, it is a sequence of such

instructions within a program that can be executed independently of other code. **Multithreading** is defined as the ability of a processor to execute multiple threads concurrently.



In our chat application, every machine acts as both the client and the server. Hence, the functions ***send*** and ***receive*** must be executed concurrently. **Multithreading** must be used to achieve this.

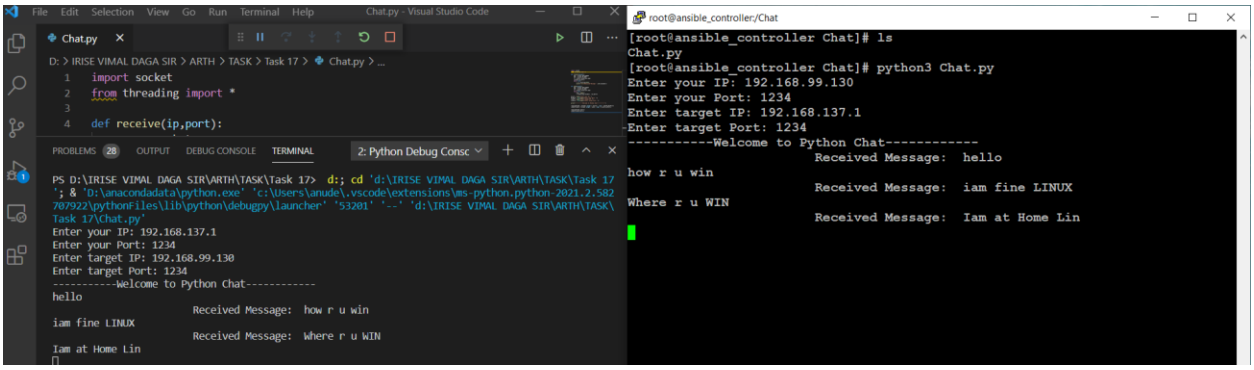
To create a new thread, we create an object of the **Thread** class. It takes the following arguments:

- **target**: the function to be executed by the thread
- **args**: the arguments to be passed to the target function

Eg: **t1 = Thread(target = receive , args = (myIP,myPort))**

Once a thread object is created, its activity must be started by calling the thread's `start()` method. This invokes the `run()` method in a separate thread of control.

```
t1.start()
```



Conclusion: UDP may be lightweight, but not that reliable.

Thanks for Reading!!!

GitHub Link: https://github.com/Anuddeeph/Chat-app.git