

Module 2

Sockets

Prepared by Dr. Manal Mizher

Lecture 2

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

Network programming
Course ID: CYB615



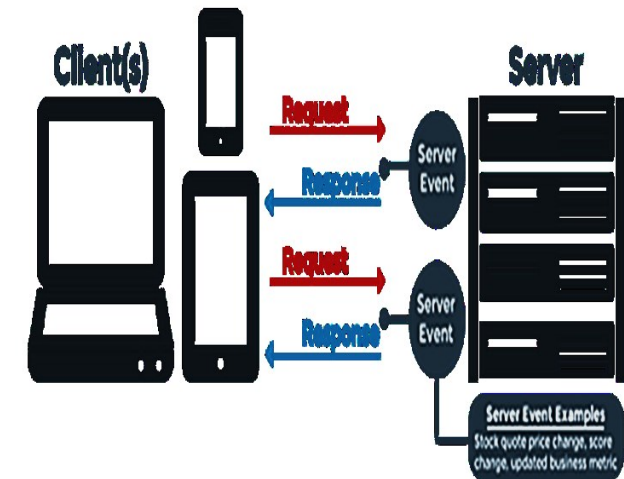
جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

Module 2(Sockets)

- 1.Basics of Sockets - Low-level networking interface
 - Definition, Sockets type, UDP & TCP sockets
- 2.Socket Methods
- 3.IPv4
- 4.Simple Client/Server Programming
- 5.Practical example (Client/Server)
- 6.H.W2

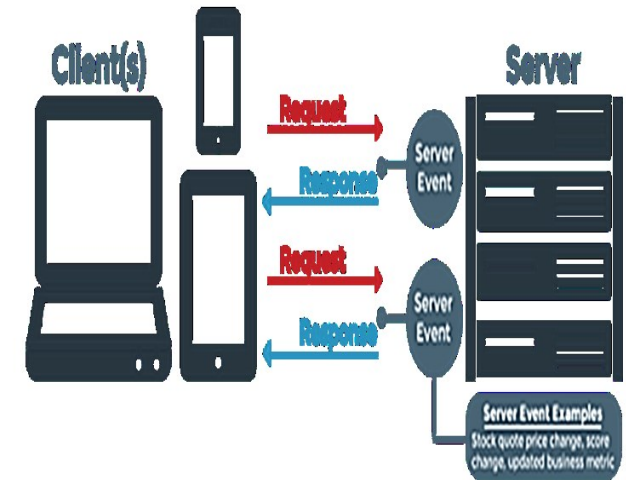
Why Do I Want Networking?

- Goal of Networking: Communication
 - Share data
 - Pass Messages
- Say I want to talk to a friend in Singapore...
 - How can I do this?
 - What applications and services must I use?
 - Where can I access them?
 - How will the data get there?
 - Will it be reliable?



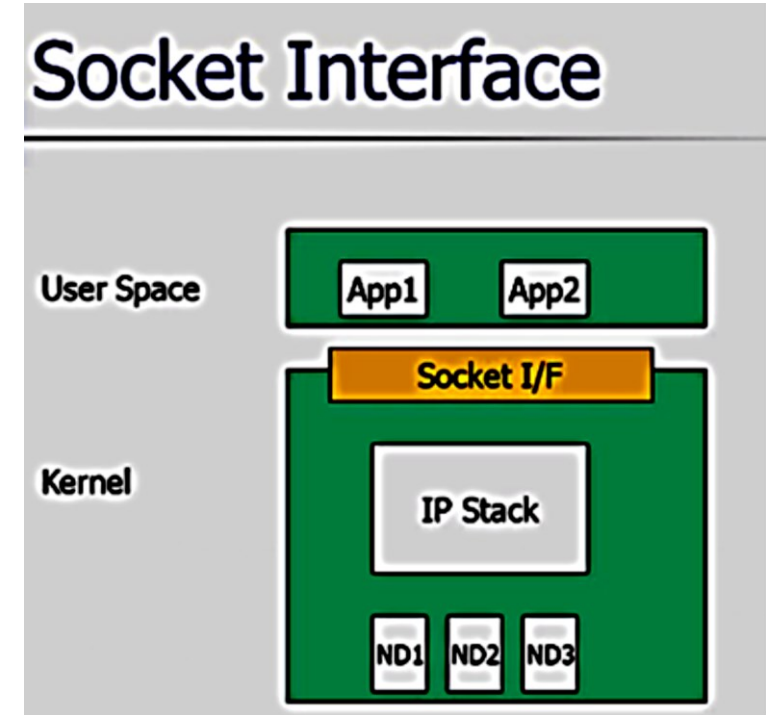
Which is easier?

- An application programmer (writing an IRC server)
 - Doesn't need to send IP packets
 - Doesn't need to send Ethernet frames
 - Doesn't need to worry about reliability
 - Shouldn't have to!
- Sockets do this!
 - TCP streams
 - UDP packetized service (Project 2)
- You'll be doing this! (using sockets)
 - To share data
 - To pass messages



- Socket is a link between two applications that can communicate with one another (either locally on a single machine or remotely between two machines in separate locations) → (within a process, between processes on the same machine or between processes on different machines).
- Basically, sockets act as a communication link between two entities, i.e. a server and a client. A server will give out information being requested by a client. For example, when you (client) visited this (<https://esu.ac.ae/>) page, the browser created a socket and connected you to the server of the university website.
- So, socket is a communication mechanism.

- ✓ First appeared in ARPANET in 1971, and later became an API in the Berkeley Software Distribution (BSD) operating system released in 1983 called Berkeley sockets (4.1 BSD UNIX 1982).
- ✓ An integer associated with a network connection.
- ✓ Implemented as file descriptor.
- ✓ Types of Internet Sockets:
 - Datagram Socket for UDP communication.
 - Stream Sockets for TCP connection.
 - Raw Sockets.



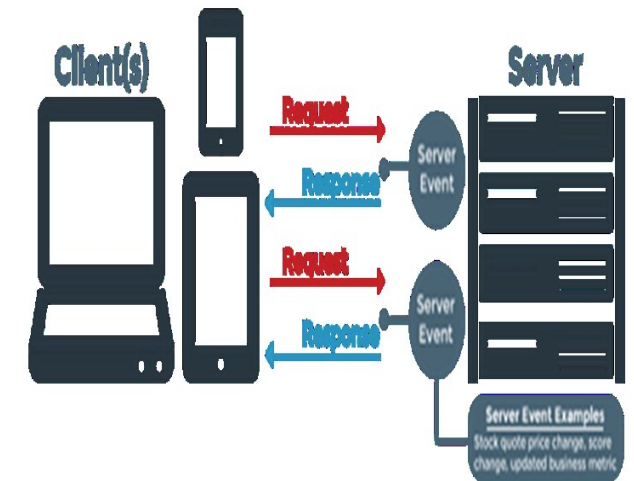
An application programming interface (API):

is a set of protocols, routines, functions and/or commands that programmers use to facilitate interaction between distinct software services.

APIs allow one software service to access data from another software service without the developer's needing to know how the other service works.

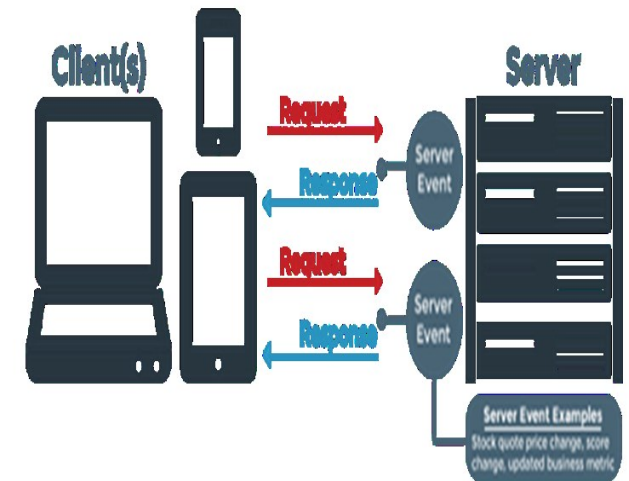
What's in a Socket

- Some information needed...
 - Where is the remote machine?
 - IP Address
 - Hostname (resolved to IP)
 - Which service do I want?
 - Port
- After that...
 - You get a file! A plain old file!
 - As simple as other Unix I/O
 - Don't forget to close it when you're done!



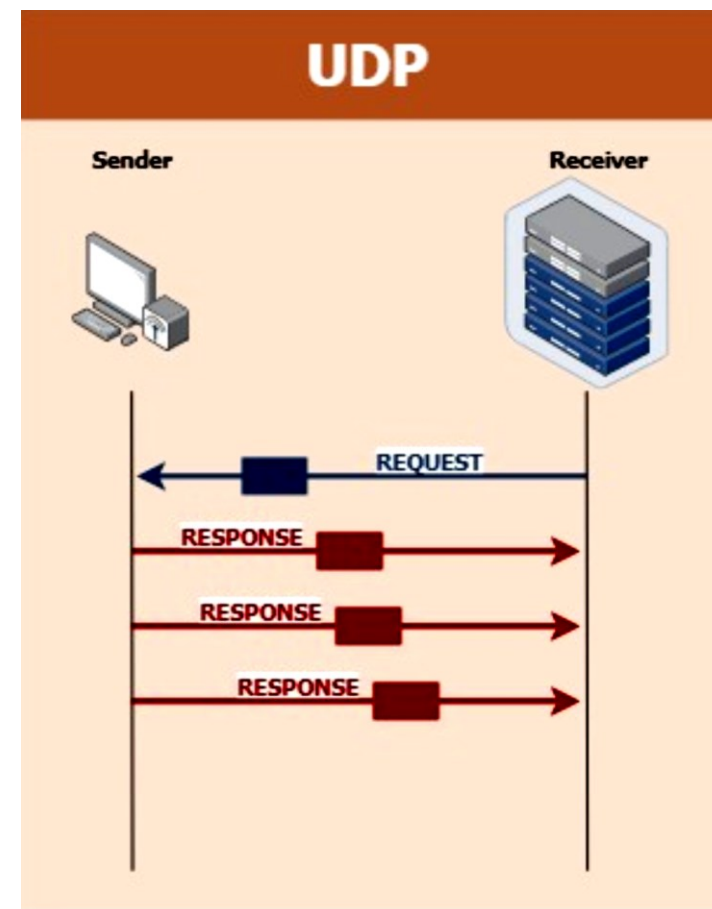
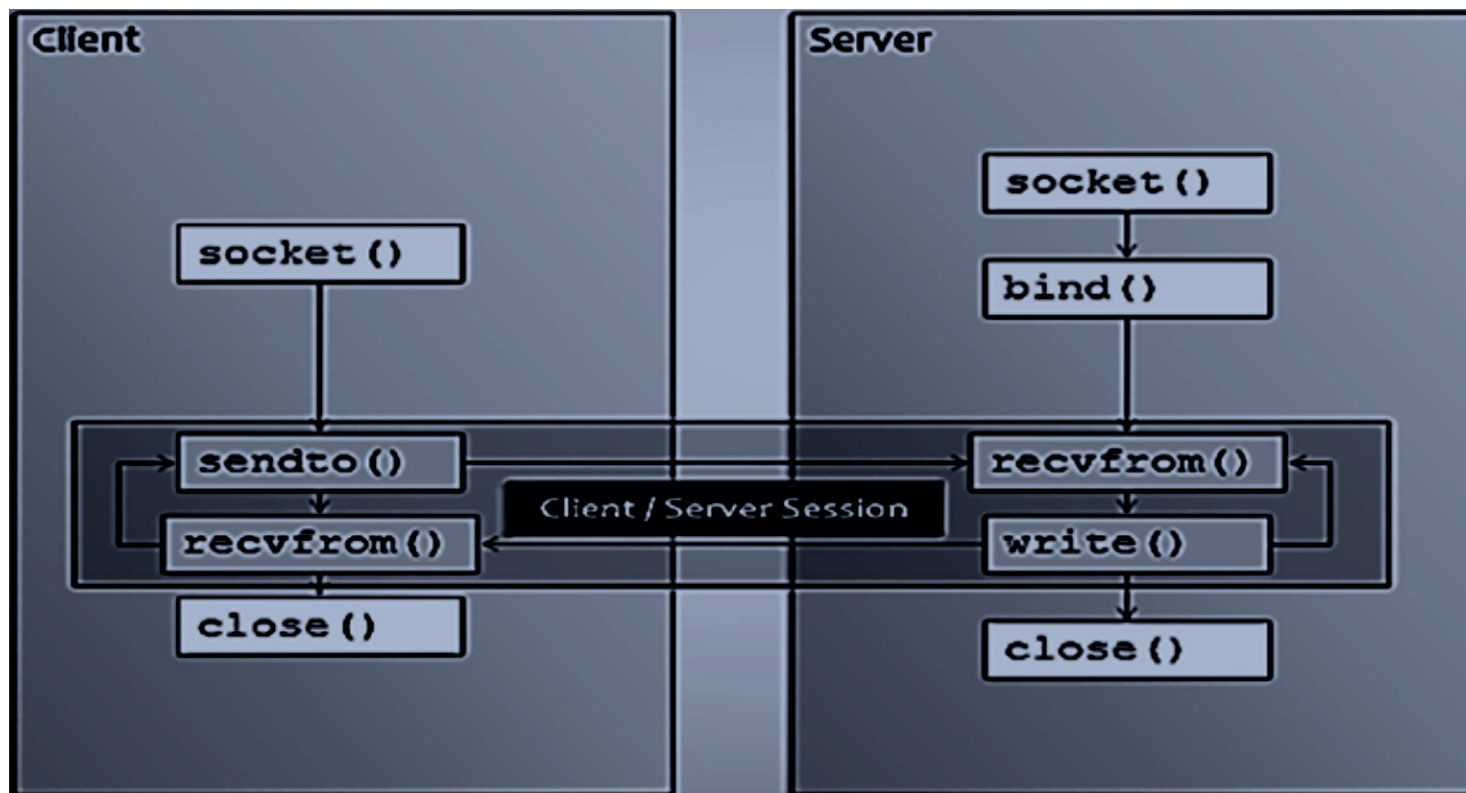
How do I do it?

- Request a socket descriptor
 - Both the client and the server need to
 - Bunch of kernel allocations...
- And the server...
 - Binds to a port
 - “I am offering a service on port x. Hear me roar”
 - Listens to the socket
 - “Hey! Say something!”
 - Accepts the incoming connection
 - “Good, you spoke up!”
- And the client...
 - Connects
 - “I’m interested!”



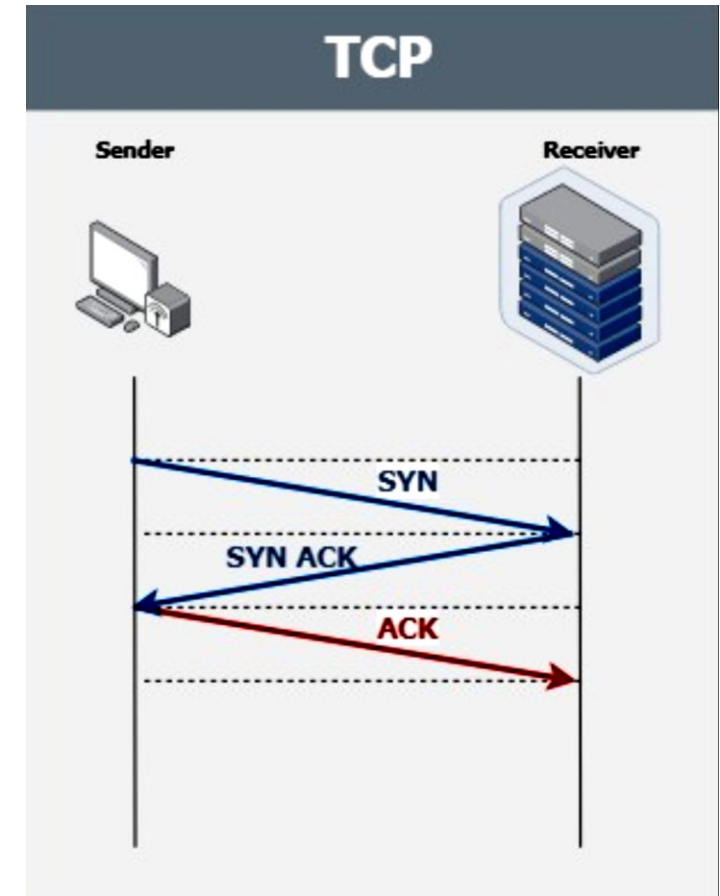
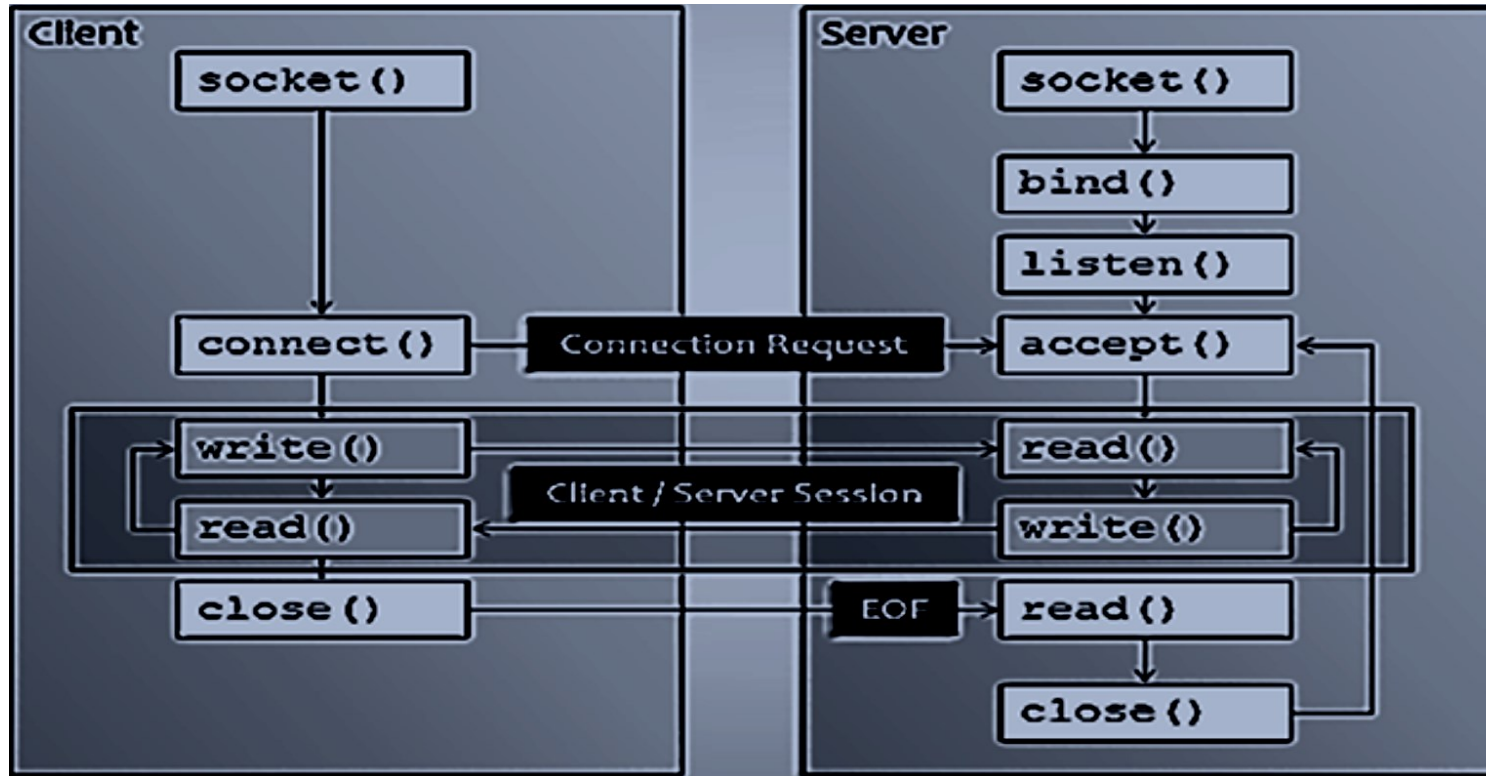
Module 2

UDP socket



Module 2

TCP socket



```
import socket
```

```
host_name = socket.gethostname()  
ip_address = socket.gethostbyname(host_name)  
print('Host name: ', host_name)  
print('IP address: ', ip_address)
```



```
import socket
```

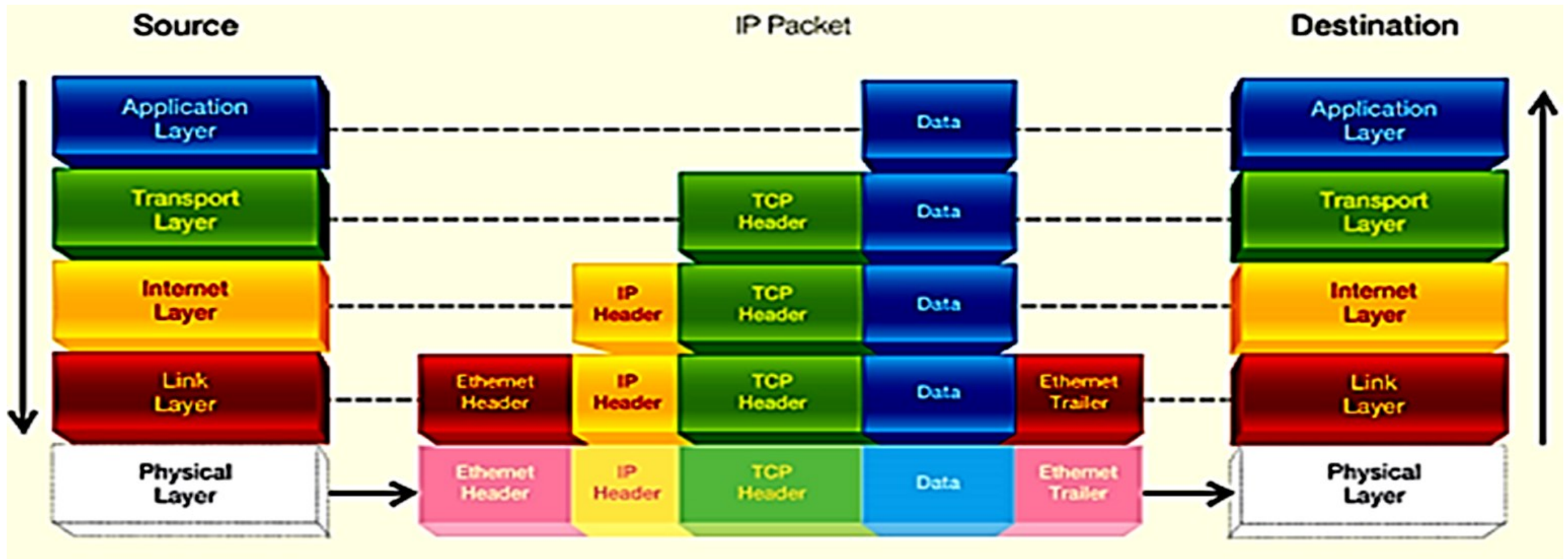
```
remote_host = 'www.python.org'  
print('IP address of :',(remote_host,socket.gethostbyname(remote_host)))  
err_msg= socket.error()  
print('Is there any error?')  
print(remote_host, err_msg)
```



Definition

IPv4 is the fourth version of IP, which establishes the rules for computer networks functioning on the principle of packet exchange.

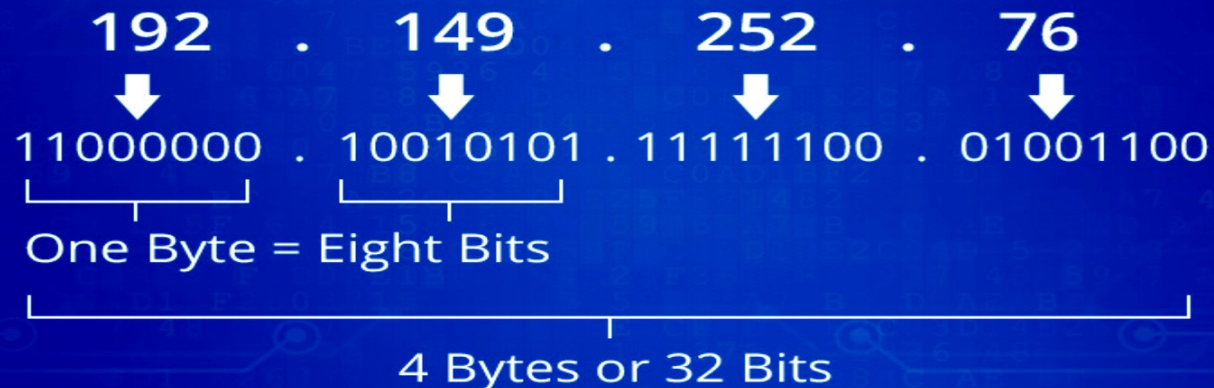
1. It can uniquely identify devices connected to the network through an addressing system. Whenever a device gets access to the Internet (whether it's a switch, PC, or other devices), it is assigned a unique, numerical IP address such as 192.149.252.76. The IPv4 uses a 32-bit address scheme allowing to store 2^{32} addresses (4.19 billion addresses).
2. The increasing end-users connected to the Internet leads to the exhaustion of IPv4 addresses. That's also why the new Internet addressing system, IPv6, is being deployed to fulfill the need for more Internet addresses.



Features of IPv4

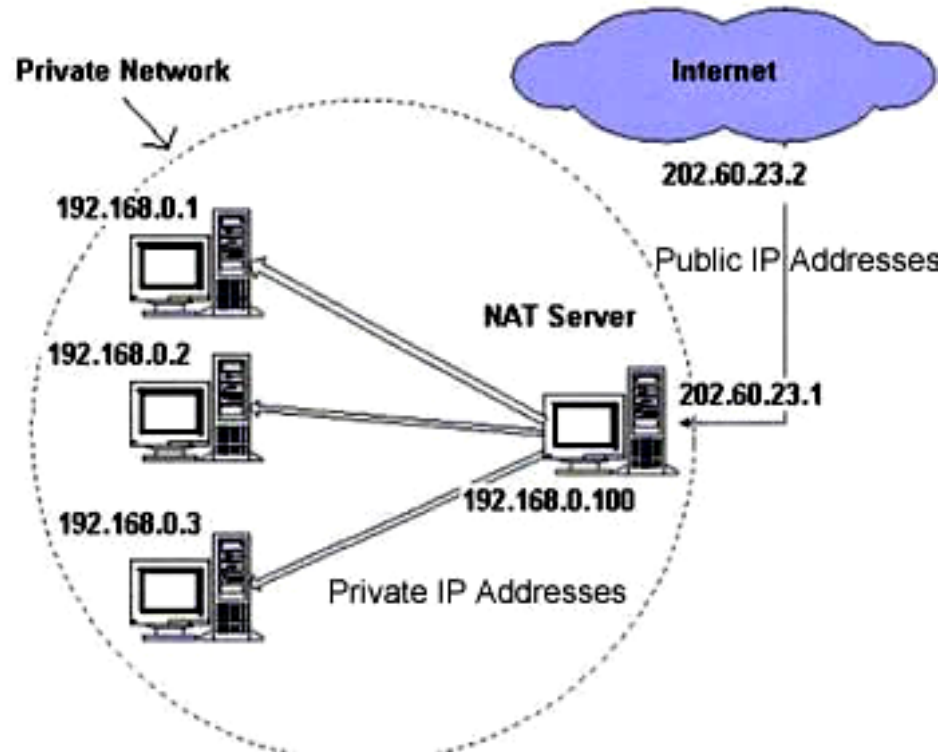
1. Connectionless Protocol
2. Allow creating a simple virtual communication layer over diversified devices
3. It requires less memory and ease of remembering addresses
4. Already supported protocol by millions of devices
5. Offers video libraries and conferences

IPv4 Address Format (Dotted-decimal Notation)



Module 2

Public IP vs Private IP



Public IP vs Private IP

Public IP	Private IP
Used over the Public Network ex. WAN	Used with in the private network ex. LAN
Recognized over the Internet	Not recognized over the Internet
Public IP are unique over the Globe.	Private IP are unique with In the network or LAN.
Public IP are paid.	Private IP are free of cost.
Assigned by Internet Service Provider /ISP	Assigned by Network Administrator.
Class A-- 10.0.0.0 to 10.255.255.255	RANGE-- Class A-- 1.0.0.0 to 9.255.255.255 11.0.0.0 to 126.255.255.255
Class B-- 172.16.0.0 to 172.31.255.255	Class B-- 128.0.0.0 to 172.15.255.255 172.32.0.0 to 191.255.255.255
Class C-- 192.168.0.0 to 192.168.255.255	Class C-- 192.0.0.0 to 192.167.255.355 192.169.0.0 to 223.255.255.255

IP Address Terminology

Static means the IP address never changes as long as you stay with the same provider or same server.

Dynamic means the IP address can change from time-to-time.

Public means the IP address can be reached via the Internet from any computer in the world.

Private means the IP address can only be reached by other devices on the same network.

Shared means other people besides you use your IP address for their connection.

Dedicated means no one else uses your IP address for their connection.

Class identifies the range of your IP address and the default subnet mask. Examples of IP classes:

- *A class - 0 to 127 with default mask of 255.0.0.0*
- *B class - 128 to 191 with default mask of 255.255.0.0*
- *C class - 192 to 223 with default mask of 255.255.255.0*
- *D class - 224 to 247 (not currently used)*
- *E class - 248 to 255 (not currently used)*

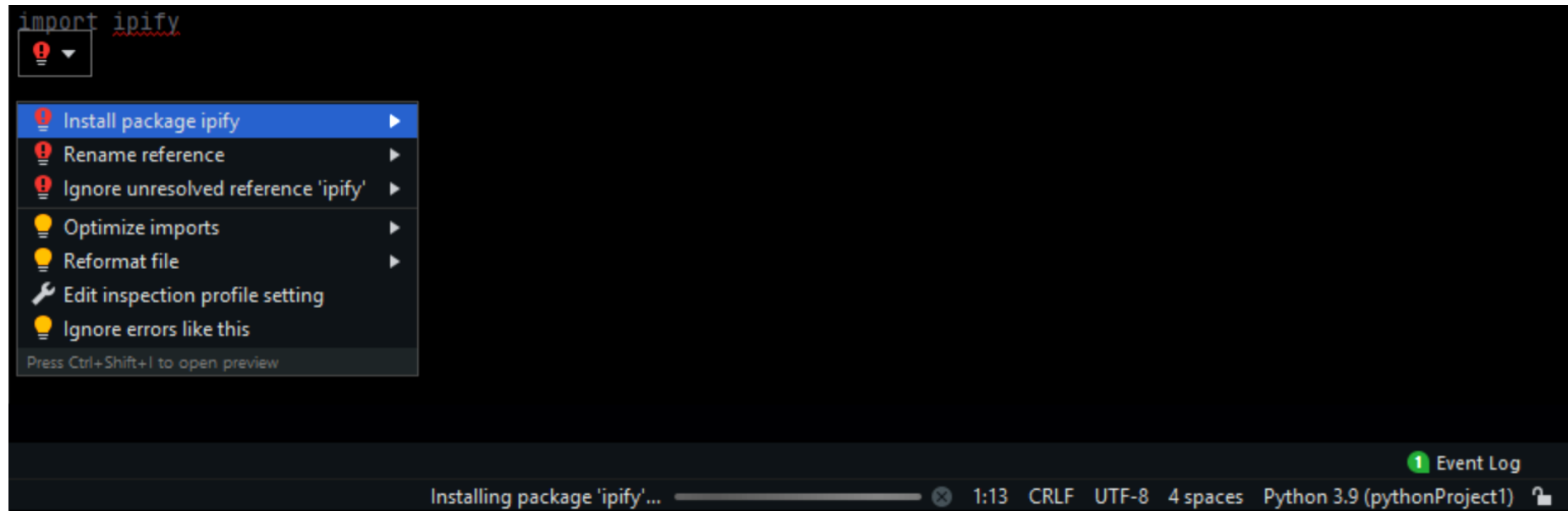


`ipaddress` – IPv4/IPv6 manipulation library

- Provides the capabilities to create, manipulate and operate on IPv4 and IPv6 addresses and networks.
- The functions and classes in this module make it straightforward to handle various tasks related to IP addresses, including checking whether or not two hosts are on the same subnet, iterating over all hosts in a particular subnet, checking whether or not a string represents a valid IP address or network definition, and so on.

To practice:

`ipify` : get your public IP address library. Try, installing `ipify` in your computer, then find your public IP address.



```
import ipaddress
```

```
# Extracting the IP version:
```

```
addr1Ver = ipaddress.ip_address('192.0.2.1')  
print('address 1 version: ', addr1Ver.version)
```

```
addr2Ver = ipaddress.ip_address('2001:db8::1')  
print('address 2 version: ', addr2Ver.version)
```

```
# Obtaining the network from an interface
```

```
host1Net = ipaddress.ip_interface('192.0.2.1/24')  
print('host 1 network: ', host1Net.network)
```

```
host2Net = ipaddress.ip_interface('2001:db8::1/96')  
print('host 2 network: ', host2Net.network)
```

```
# Finding out how many individual addresses are in a network:
```

```
net4 = ipaddress.ip_network('192.0.2.0/24')  
print('Individual network4: ', net4.num_addresses)
```

```
net6 = ipaddress.ip_network('2001:db8::0/96')  
print('Individual network6: ', net6.num_addresses)
```

```
# Iterating through the "usable" addresses on a network:
```

```
net4 = ipaddress.ip_network('192.0.2.0/24')  
for x in net4.hosts():  
    print('host', x)
```

```
# Obtaining the netmask (i.e. set bits corresponding to the network prefix)  
# or the hostmask (any bits that are not part of the netmask):
```

```
net4 = ipaddress.ip_network('192.0.2.0/24')  
print('Mask network4: ', net4.netmask)  
print('Mask host4: ', net4.hostmask)
```

```
net6 = ipaddress.ip_network('2001:db8::0/96')  
print('Mask network6: ', net6.netmask)  
print('Mask host6: ', net6.hostmask)
```

There are two levels of network service access in Python. These are:

1. Low-Level Access

You can access the basic socket support in the underlying operating system using python's libraries, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

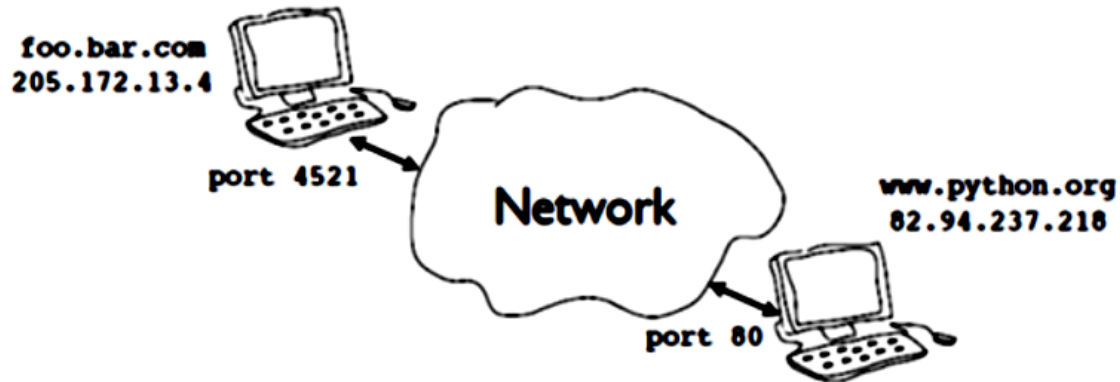
2. High-Level Access

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

- Sockets are created using a set of programming requests called socket API (Application Programming Interface). Python's socket library offers classes for handling common transports as a generic interface.
- Sockets use protocols for determining the connection type for port-to-port communication between client and server machines. The protocols are used for:
 1. Domain Name Servers (DNS)
 2. IP addressing
 3. E-mail
 4. FTP (File Transfer Protocol) etc...

Network Addressing

- Machines have a hostname and IP address
- Programs/services have port numbers



Standard Ports

- Ports for common services are preassigned

21	FTP
22	SSH
23	Telnet
25	SMTP (Mail)
80	HTTP (Web)
110	POP3 (Mail)
119	NNTP (News)
443	HTTPS (web)

- Other port numbers may just be randomly assigned to programs by the operating system

```
import socket
def find_service_name():
    protocolname = 'tcp'
    for port in [21, 22, 23, 25, 80, 110, 119, 443]:
        print ("Port: %s => service name: %s" %(port, socket.getservbyport(port, protocolname)))
    print ("Port: %s => service name: %s" %(53, socket.getservbyport(53,'udp')))

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

1. Domain

Domain is the family of protocols that is used as the transport mechanism. These values are constants such as AF_INET for IPv4, AF_INET6 for IPv6, PF_UNIX, PF_X25, and so on.

2. Type

Type means the kind of communication between two endpoints, typically SOCK_STREAM (TCP) for connection-oriented protocols and SOCK_DGRAM (UDP) for connectionless protocols.

3. Protocol

This may be used to identify a variant of a protocol within a domain and type. Its default value is 0. This is usually left out.

4. Hostname

This works as the identifier of a network interface. A hostname may be a string, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation.

5. Port

Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service.

To implement socket programming in python, we need to use the Socket module. Following is a simple syntax to create a Socket:

```
import socket  
s = socket.socket (socket_family, socket_type, protocol = 0)
```

Different parameters used while making socket

- **socket_family** – This is either AF_UNIX or AF_INET, as explained earlier.
- **socket_type** – This is either SOCK_STREAM or SOCK_DGRAM.
- **protocol** – This is usually left out, defaulting to 0.

Socket_family:

`AF_INET` is used for **IPv4** Internet addressing.

`AF_INET6` is used for **IPv6** Internet addressing.

`AF_UNIX` is the address family for Unix Domain Sockets (**UDS**)

Socket_type:

`SOCK_DGRAM` for user datagram protocol (**UDP**) → connection-less.

`SOCK_STREAM` for transmission control protocol (**TCP**) → connection-oriented protocols.

Module 2

Create Socket in Python

```
import socket
```

```
# Create a UDP socket
```

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

UDP socket

```
import socket
```

```
# Create a TCP/IP socket
```

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

TCP socket



Server Socket Methods

In the client-server architecture, there is one centralized server that provides service and many clients receive service from that centralized server. The clients also do the request to server. A few important server socket methods in this architecture are as follows –

- **socket.bind()** – This method binds the address (hostname, port number) to the socket.
- **socket.listen()** – This method basically listens to the connections made to the socket. It starts TCP listener. Backlog is an argument of this method which specifies the maximum number of queued connections. Its minimum value is 0 and maximum value is 5.
- **socket.accept()** – This will accept TCP client connection. The pair (conn, address) is the return value pair of this method. Here, conn is a new socket object used to send and receive data on the connection and address is the address bound to the socket. Before using this method, the socket.bind() and socket.listen() method must be used.



```
import socket
import sys

# Create a TCP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the address given on the command line
# server_name = sys.argv[1] if len(sys.argv) > 1 else '.'

server_address = ('localhost', 10000)
print('starting up on port :', server_address)
sock.bind(server_address)
sock.listen(5)

while True:
    print('waiting for a connection')
    connection, client_address = sock.accept()
    ,
    ,
    ,
```



Client Socket Methods

The client in the client-server architecture requests the server and receives services from the server. For this, there is only one method dedicated for clients –

- **socket.connect(address)** – this method actively intimate server connection or in simple words this method connects the client to the server. The argument address represents the address of the server.

```
import socket
import sys

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect the socket to the port where the server is listening
server_address = ('localhost', 10000)
print('connecting to port :', server_address)
sock.connect(server_address)
```

General Socket Methods

Other than client and server socket methods, there are some general socket methods, which are very useful in socket programming. The general socket methods are as follows –

- **socket.recv(bufsize)** – As name implies, this method receives the TCP message from socket. The argument bufsize stands for buffer size and defines the maximum data this method can receive at any one time.
- **socket.send(bytes)** – This method is used to send data to the socket which is connected to the remote machine. The argument bytes will gives the number of bytes sent to the socket.
- **socket.recvfrom(data, address)** – This method receives data from the socket. Two pair (data, address) value is returned by this method. Data defines the received data and address specifies the address of socket sending the data.
- **socket.sendto(data, address)** – As name implies, this method is used to send data from the socket. Two pair (data, address) value is returned by this method. Data defines the number of bytes sent and address specifies the address of the remote machine.
- **socket.close()** – This method will close the socket.
- **socket.gethostname()** – This method will return the name of the host.
- **socket.sendall(data)** – This method sends all the data to the socket which is connected to a remote machine. It will carelessly transfers the data until an error occurs and if it happens then it uses socket.close() method to close the socket.

The Steps in Establishing a Socket on Server

1. Create a socket with the `socket()` system call
2. Bind the socket to an address using the `bind()` system call. For a server socket, an address consists of a port number on the host machine
3. Listen for connections with `listen()` system call
4. Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server
5. Send and receive data with the `send()`, and finally close the connection with the `close()`

The Steps in Establishing a Socket on Client

1. Create a socket with the `socket()` system call
2. Connect the socket to the address of the server using the `connect()` system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` or `send()` and `recv()` system calls



Create a Simple Client

Before we get started, let's look at the client socket methods available in Python.

- `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- `s.connect()` : Initiates a TCP server connection.

To create a new socket, you first import the socket method of the socket class.

```
1 import socket
```

Next, we'll create a stream (TCP) socket as follows:

```
1 stream_socket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
```

The `AF_INET` argument indicates that you're requesting an Internet Protocol (IP) socket, specifically IPv4. The second argument is the transport protocol type `SOCK_STREAM` for TCP sockets. Additionally, you can also create an IPv6 socket by specifying the socket `AF_INET6` argument.

Module 2

Practical example (ClientTCP/ServerTCP)



Specify the server.

```
1 | server = "localhost"
```

Specify the port we want to communicate with.

```
1 | port = 80
```

Connect the socket to the port where the server is listening.

```
1 | server_address = ((host, port))  
2 | stream_socket.connect(server_address)
```

It's important to note that the host and port must be a tuple.

Send a data request to the server:

```
1 | message = 'message'  
2 | stream_socket.sendall(message)
```

Get the response from the server:

```
1 data = sock.recv(10)
2 print data
```

To close a connected socket, you use the close method:

```
1 stream_socket.close()
```

Build a Simple Server

Now let's take a look at a simple Python server. The following are the socket server methods available in Python.

- `s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- `s.bind()`: Binds address (hostname, port number) to socket.
- `s.listen()`: Sets up and starts TCP listener.
- `s.accept()`: Accepts TCP client connection.

We will follow the following steps:

- Create a socket.
- Bind the socket to a port.
- Start accepting connections on the socket.

Server Output

```
1 $ python server.py
2 waiting for a connection
3 ('127.0.0.1', 47050) connected
4 received "message"
```

Client Output

```
1 $ python client.py
2 connecting
3 message
4 socket closed
```

Run the client and server programs in separate terminal windows, so they can communicate with each other.

Q1: Find the following errors in HW2 program between the client and the server:

- 1- syntax error (1)
- 2- logic error (1)
- 3- runtime error (1)
- 4- operational error (1)

Q2:

How do I do it?

- Request a socket descriptor
 - Both the client and the server need to
 - Bunch of kernel allocations...
- And the server...
 - Binds to a port
 - "I am offering a service on port x. Hear me roar"
 - Listens to the socket
 - "Hey! Say something!"
 - Accepts the incoming connection
 - "Good, you spoke up!"
- And the client...
 - Connects
 - "I'm interested!"



And when were done...

- You must `close()` it!
 - Just like a file (since it appears as one!)
- What next? Loop around...
 - Accept new connections
 - Process requests
 - Close them
 - Rinse, repeat
- What's missing here?

