Terna Engineering College

Computer Engineering Department
Program: Sem V

Course: Computer Network Lab

PART A

(PART A: TO BE REFERRED BY STUDENTS)

Experiment No. 09

A.1 Objective:

Establishing Client Server communication using TCP socket programming.

A.2 Prerequisite:

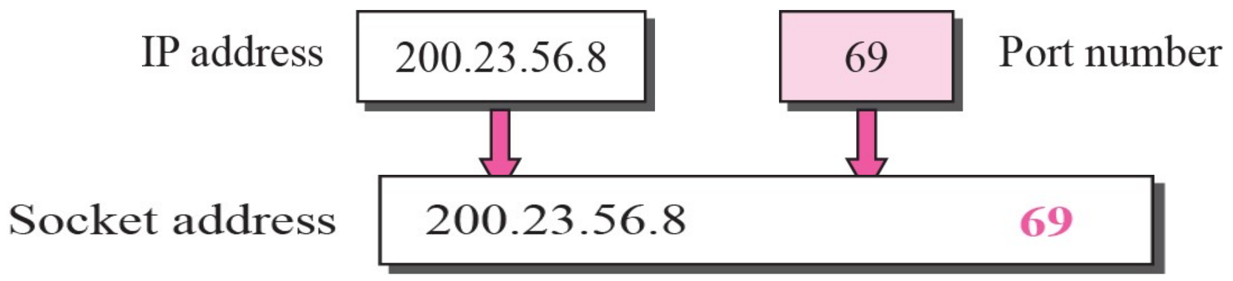
- Knowledge about LAN, MAN and WAN and NW Elements.
- Linux NEW Commands
- HW and IP Address concepts.
- Concept of Port, Socket, Localhost, Client and Server,
- Any programing language such as C, C++, Java or Python
- NW libraries.

A.3 Outcome:

After successful completion of this experiment students will be able to

- Ability to establish connection
- Ability to communicate among the PCS
- Ability to write NW related system program
- Ability to program the socket.

A.4 Theory/Tutorial:



What are Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary −

Sr.No.	Term & Description
1	**Domain** The family of protocols that is used as the transport mechanism. These values are constants such as AF_INET, PF_INET, PF_UNIX, PF_X25, and so on.
2	**Type** The type of communications between the two endpoints, typically SOCK_STREAM for connection-oriented protocols and SOCK_DGRAM for connectionless protocols.
3	**protocol** Typically zero, this may be used to identify a variant of a protocol within a domain and type.
4	**Hostname** The identifier of a network interface − • A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation • A string "<broadcast>", which specifies an INADDR_BROADCAST address. • A zero-length string, which specifies INADDR_ANY, or • An Integer, interpreted as a binary address in host byte order.
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.

The *socket* Module:

To create a socket, you must use the *socket.socket()* function available in *socket* module, which has the general syntax −

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

Here is the description of the parameters −

- **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.

Once you have *socket* object, then you can use required functions to create your client or server program. Following is the list of functions required −

Server Socket Methods

Sr.No.	Method & Description
1	**s.bind()** This method binds address (hostname, port number pair) to socket.
2	**s.listen()** This method sets up and start TCP listener.
3	**s.accept()** This passively accept TCP client connection, waiting until connection arrives (blocking).

Client Socket Methods

Sr.No.	Method & Description
1	**s.connect()** This method actively initiates TCP server connection.

General Socket Methods

Sr.No.	Method & Description
1	**s.recv()** This method receives TCP message
2	**s.send()** This method transmits TCP message
3	**s.recvfrom()** This method receives UDP message
4	**s.sendto()** This method transmits UDP message
5	**s.close()** This method closes socket
6	**socket.gethostname()** Returns the hostname.

A Simple Server

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call **bind(hostname, port)** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

```
#!/usr/bin/python           # This is server.py file

import socket               # Import socket module

s = socket.socket()         # Create a socket object
```

```
host = socket.gethostname() # Get local machine name
port = 12345              # Reserve a port for your service.
s.bind((host, port))        # Bind to the port

s.listen(5)               # Now wait for client connection.
while True:
  c, addr = s.accept()     # Establish connection with client.
  print 'Got connection from', addr
  c.send('Thank you for connecting')
  c.close()                # Close the connection
```

A Simple Client

Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function.

The **socket.connect(hosname, port)** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

```
#!/usr/bin/python           # This is client.py file

import socket               # Import socket module

s = socket.socket()         # Create a socket object
host = socket.gethostname() # Get local machine name
port = 12345                # Reserve a port for your service.

s.connect((host, port))
print s.recv(1024)
s.close()                     # Close the socket when done
```

Now run this server.py in background and then run above client.py to see the result.

```
# Following would start a server in background.
$ python server.py &

# Once server is started run client as follows:
$ python client.py
```

This would produce following result −

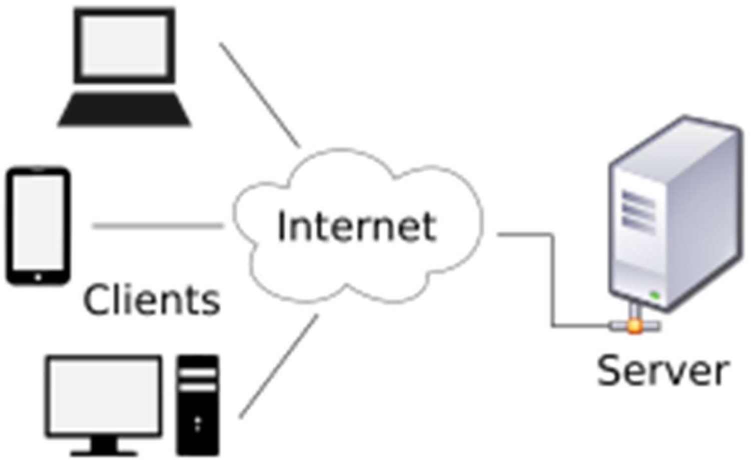
```
Got connection from ('127.0.0.1', 48437)
Thank you for connecting
```

Python Internet modules

A list of some important modules in Python Network/Internet programming.

Protocol	Common function	Port No	Python module
HTTP	Web pages	80	httplib, urllib, xmlrpclib
NNTP	Usenet news	119	nntplib
FTP	File transfers	20	ftplib, urllib
SMTP	Sending email	25	smtplib
POP3	Fetching email	110	poplib
IMAP4	Fetching email	143	imaplib
Telnet	Command lines	23	telnetlib
Gopher	Document transfers	70	gopherlib, urllib

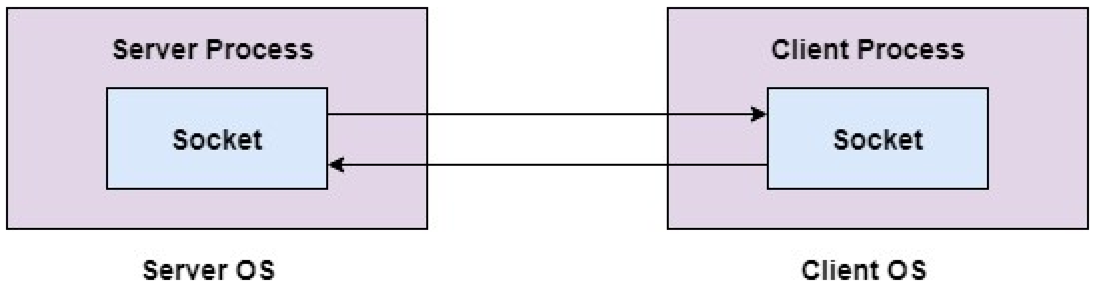
Please check all the libraries mentioned above to work with FTP, SMTP, POP, and IMAP protocols.



Client/Server communication involves two components, namely a client and a server. They are usually multiple clients in communication with a single server. The clients send requests to the server and the server responds to the client requests.

There are three main methods to client/server communication. These are given as follows −

A diagram that illustrates sockets is as follows −

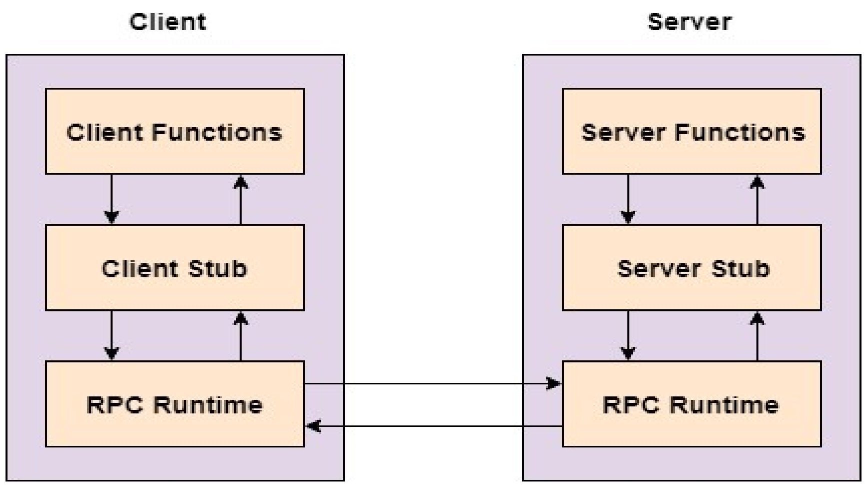


Remote Procedure Calls:

These are interprocess communication techniques that are used for client-server based applications. A remote procedure call is also known as a subroutine call or a function call.

A client has a request that the RPC translates and sends to the server. This request may be a procedure or a function call to a remote server. When the server receives the request, it sends the required response back to the client.

A diagram that illustrates remote procedure calls is given as follows −



Pipes:

These are interprocess communication methods that contain two end points. Data is entered from one end of the pipe by a process and consumed from the other end by the other process.

The two different types of pipes are ordinary pipes and named pipes. Ordinary pipes only allow one way communication. For two way communication, two pipes are required. Ordinary pipes have a parent child relationship between the processes as the pipes can only be accessed by processes that created or inherited them.

Named pipes are more powerful than ordinary pipes and allow two way communication. These pipes exist even after the processes using them have terminated. They need to be explicitly deleted when not required anymore.

A diagram that demonstrates pipes are given as follows −

Parent Process
Child Process

References:

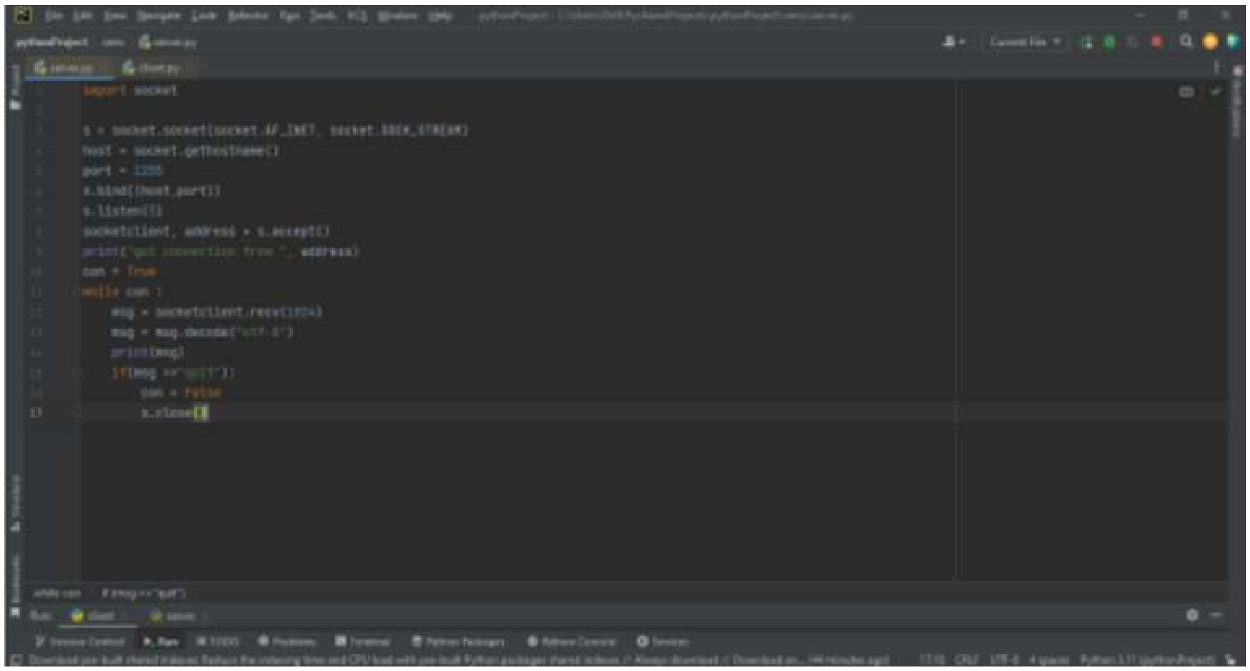
1. https://www.tutorialspoint.com/operating-systems-client-server-communication
2. https://www.tutorialspoint.com/python/python_networking.html
3. https://youtu.be/u4kr7EFxAKk

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

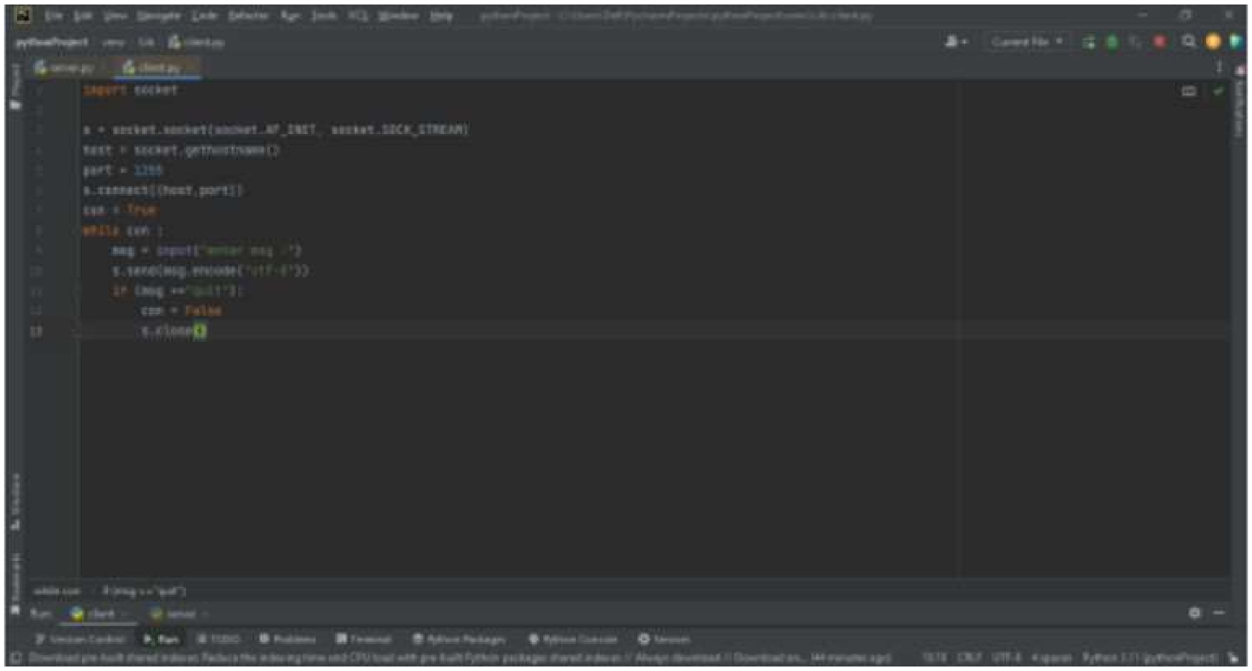
Roll No. A11	Name: Khan Mohammad TAQI Karrar Husain
Class : T.E A	Batch : A1
Date of Experiment:	Date of Submission:
Grade :	

B.1 Document created by the student:
Server.py

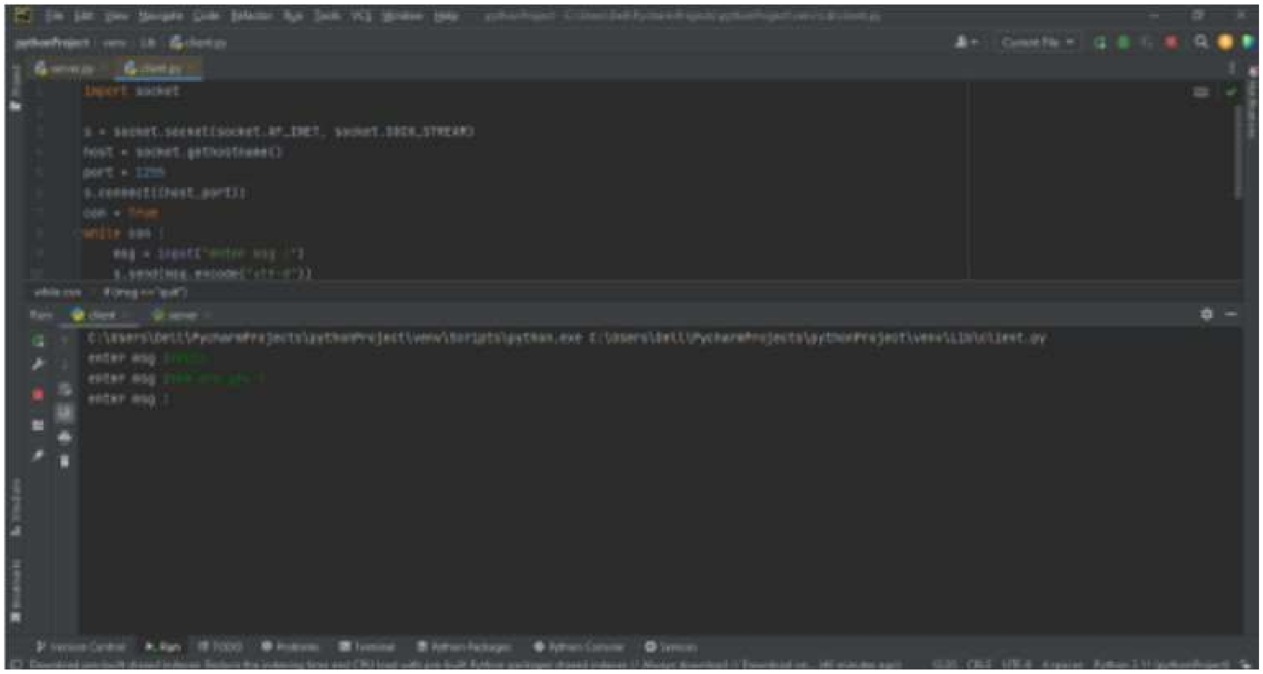


```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 1255
s.bind((host,port))
s.listen(1)
socketclient, address = s.accept()
print("got connection from ", address)
con = True
while con :
    msg = socketclient.recv(1024)
    msg = msg.decode("utf-8")
    print(msg)
    if(msg =="quit"):
        con = False
        s.close()
```

Client.py

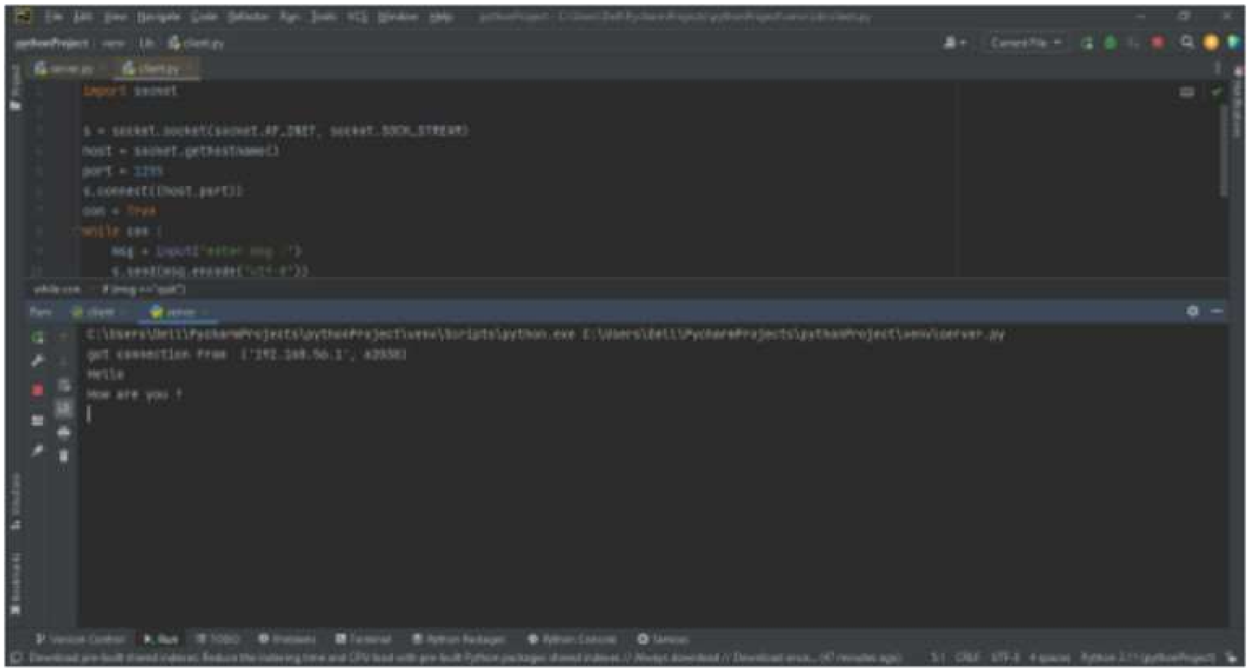


```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 1255
s.connect((host,port))
con = True
while con :
    msg = input("enter msg :")
    s.send(msg.encode("utf-8"))
    if (msg =="quit"):
        con = False
        s.close()
```



```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 1235
s.connect((host,port))
con = True
while con :
    msg = input('enter msg :')
    s.send(msg.encode('utf-8'))
```

```
C:\Users\Dell\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\Dell\PycharmProjects\pythonProject\venv\Lib\client.py
enter msg :Hello
enter msg :How are you ?
enter msg :
```



```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
port = 1235
s.connect((host,port))
con = True
while con :
    msg = input('enter msg :')
    s.send(msg.encode('utf-8'))
```

```
C:\Users\Dell\PycharmProjects\pythonProject\venv\Scripts\python.exe C:\Users\Dell\PycharmProjects\pythonProject\venv\server.py
got connection from  ('192.168.56.1', 63938)
Hello
How are you ?
```

B.3 Observations and learning:

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to. An endpoint is a combination of an IP address and a port number.

B.4 Conclusion:

Socket programs are used to communicate between various processes usually running on different systems. It is mostly used to create a client-server environment. This post provides the various functions used to create the server and client program and an example program.

B.5 Question of Curiosity:

1. What is Client? Give an example.

Answer: In the context of networking and communication, a "client" refers to a program or device that requests services or resources from another program or device known as a "server." The client-server model is a fundamental architecture in computing where clients initiate requests, and servers fulfill those requests.
Examples of clients include:
1. Web Browser: A web browser (e.g., Chrome, Firefox, Safari) is a client that requests web pages from web servers. When you enter a URL and hit Enter, the browser acts as a client to retrieve and display the requested web page from the server.
2. Email Client: An email client, such as Microsoft Outlook or Mozilla Thunderbird, is a program that connects to an email server to send and receive emails. The client interacts with the server to download and store emails.
3. FTP Client: File Transfer Protocol (FTP) clients like FileZilla or WinSCP are used to connect to FTP servers for uploading and downloading files. The client sends commands to the server to manage file transfers.
4. Database Client: Database clients, such as MySQL Workbench or pgAdmin, enable users to interact with database servers. They send queries and receive results, facilitating the management and retrieval of data.
5. Chat Applications: Messaging applications like WhatsApp or Slack act as clients that connect to messaging servers. Users send messages, and the server facilitates the delivery of those messages to the intended recipients.
6. Online Games: In online gaming, the game client is the software that players use to connect to game servers. The client sends and receives data related to the game, such as player movements and actions.
7. Print Client: A print client in a network can send print jobs to a print server. For example, when you print a document from your computer to a network printer, your computer acts as the print client.

2. What is a Server? Give Examples.

Answer: In computer networking, a "server" is a program or device that provides services or resources to other programs or devices, known as clients. Servers are designed to respond to requests from clients, fulfill those requests, and facilitate the sharing of data or resources. Here are examples of servers:
1. Web Server: Apache, Nginx, and Microsoft Internet Information Services (IIS) are examples of web servers. They respond to HTTP requests from web browsers (clients) and deliver web pages, images, and other content.

2. Email Server: Servers such as Microsoft Exchange, Postfix, and Sendmail handle email communication. They store, send, and receive emails on behalf of email clients.
3. File Server: Servers like Windows File Server or Network Attached Storage (NAS) devices provide file storage and sharing services. Clients can access files and folders stored on these servers over a network.
4. Database Server: Database servers such as MySQL Server, Oracle Database Server, or Microsoft SQL Server store and manage databases. Clients, such as database management tools or applications, connect to these servers to retrieve or modify data.
5. FTP Server: File Transfer Protocol (FTP) servers, like vsftpd or ProFTPD, facilitate the transfer of files between clients and the server. Clients use FTP clients to upload or download files.
6. Print Server: Print servers manage print jobs in a network. They receive print requests from clients and control the printing process. For instance, CUPS (Common Unix Printing System) is used as a print server.
7. Application Server: Application servers, such as Java EE servers (e.g., Apache Tomcat, WildFly) or Microsoft's Internet Information Services (IIS), host and run applications. Clients connect to these servers to access specific functionalities or services provided by the applications.
8. DNS Server: Domain Name System (DNS) servers resolve domain names to IP addresses. When a client requests a domain (e.g., www.example.com), a DNS server translates it into the corresponding IP address.
9. Game Server: Game servers, like those used in multiplayer online games, manage the gaming environment. They handle player interactions, store game data, and synchronize the game state across clients. Examples include game servers for Minecraft or Counter-Strike.
10. Chat Server: Chat servers, like those used in messaging applications or online chat platforms, facilitate real-time communication between clients. Examples include servers used by platforms like Slack, Discord, or WhatsApp.

3. What is a Port? How many ports are possible? What are the well-known ports, registered ports and Dynamic or private port numbers?

Answer: In computer networking, a "port" is a virtual endpoint for communication. It is a numeric identifier that, along with the IP address, helps direct data to the correct application or process on a networked device. Ports are essential for enabling multiple services or applications to run simultaneously on a single device.Here are key points about ports:
1. Port Numbers:
- Port numbers range from 0 to 65535.
- Ports up to 1023 are considered "well-known ports" and are typically reserved for standard services like HTTP (80), HTTPS (443), FTP (21), etc.
- Ports from 1024 to 49151 are "registered ports," which can be used by user- or vendor-specific applications.
- Ports from 49152 to 65535 are "dynamic" or "private ports" and are typically used for ephemeral connections (temporary connections between clients and servers).

2. Well-Known Ports:

- Well-known ports are standardized and assigned to specific protocols or services to ensure consistency across systems.
- Examples of well-known ports:
- HTTP (HyperText Transfer Protocol): 80
- HTTPS (HTTP Secure): 443
- FTP (File Transfer Protocol): 21
- SMTP (Simple Mail Transfer Protocol): 25
- DNS (Domain Name System): 53

3. Registered Ports:

- Registered ports are used by applications, services, or protocols that are not standardized but have been registered with the Internet Assigned Numbers Authority (IANA).
- Examples of registered ports:
- MySQL Database: 3306
- PostgreSQL Database: 5432
- Oracle Database: 1521

4. Dynamic or Private Ports:

- Dynamic or private ports (also known as ephemeral ports) are used for temporary connections.
- They are typically assigned by the operating system to client applications when initiating a connection to a server.
- The range for dynamic ports is 49152 to 65535.

4. What is Socket? And why should one program it?

<u>Answer:</u> A "socket" is a software abstraction representing an endpoint for sending or receiving data across a computer network. It provides a programming interface (API) for network communication and enables communication between processes or applications running on different devices. Sockets are fundamental to network programming and facilitate the development of client-server applications. Here are key points about sockets:

1. Communication Endpoint:

- A socket is essentially an endpoint for sending or receiving data across a network.
- It is identified by a combination of an IP address and a port number.

2. Socket Programming:

- Socket programming involves using a programming language's socket API to create, configure, and manage sockets.
- Common socket programming APIs include the Berkeley Sockets API, Winsock (Windows Sockets), and Java's Socket API.

3. Types of Sockets:

- Stream Sockets (e.g., TCP): Provide a reliable, connection-oriented communication stream. Data sent using stream sockets is guaranteed to arrive in the correct order without loss.
- Datagram Sockets (e.g., UDP): Provide connectionless, unreliable communication. Datagram sockets are suitable for scenarios where some data loss is acceptable, such as real-time streaming.

4. Why Program Sockets:

- Network Communication: Sockets enable communication between processes running on different devices over a network. This is crucial for distributed computing, where applications need to exchange data.
- Client-Server Applications: Sockets are the foundation for building client-server applications. For example, a web server listens for incoming connections on a specific port using a socket, and a web browser connects to that port to request and receive web pages.
- Protocols Implementation: Sockets are used to implement various network protocols, such as HTTP, FTP, SMTP, etc. Developers can use sockets to create custom applications that adhere to specific communication protocols.
- Real-Time Communication: Sockets are essential for real-time communication applications like chat applications, online gaming, and video conferencing. They allow for quick and efficient data exchange between clients and servers.
- Data Exchange Between Processes: Sockets provide a standardized way for processes or applications on the same or different machines to exchange data. This is vital for sharing information between components of a distributed system.
- Low-Level Network Interaction: Sockets allow developers to interact with the lower levels of the network stack, giving them fine-grained control over communication parameters and behavior.

5. What are the server side commands/library functions used and mention their purpose?

Answer: In server-side programming, especially in the context of network programming and creating server applications, various commands or library functions are commonly used. The specific commands or functions can vary depending on the programming language and the server framework or library being utilized. Below are some general categories of server-side commands and functions along with their purposes:

1. Socket Creation and Configuration:

- `socket()`: Creates a new socket.
- `bind()`: Associates a socket with a specific IP address and port.
- `listen()`: Puts the socket in listening mode, allowing it to accept incoming connections.
- `accept()`: Accepts an incoming connection and returns a new socket for communication with the client.

2. Data Exchange:

- `send()`: Sends data over a connected socket.
- `recv()`: Receives data from a connected socket.
- `sendto()`: Sends data to a specific address using a connectionless socket (e.g., UDP).
- `recvfrom()`: Receives data from a specific address using a connectionless socket.

3. Connection Management:

- `connect()`:Initiates a connection to a remote server.
- `close()`: Closes a socket and releases associated resources.

4. Address Conversion:

- `getaddrinfo()`: Translates human-readable addresses (such as domain names) into a form suitable for use with network operations.
- `inet_pton()`: Converts a human-readable IP address to a network address structure.

5. Multithreading/Concurrency:

- Thread Management Functions: For creating and managing threads for handling multiple client connections concurrently.
- Synchronization Mechanisms: Such as mutexes and semaphores to manage shared resources among threads.

6. Error Handling:

- `perror()`: Prints a description for the last error that occurred.
- `strerror()`: Returns a pointer to the textual representation of the current errno value.

7. Event Handling (Asynchronous I/O):

- Libraries or functions for handling asynchronous events or non- blocking I/O, like `select()` or libraries providing event-driven architectures.

8. Security:

- SSL/TLS Libraries: If secure communication is required, libraries like OpenSSL or native TLS support in some languages.

9. Server Frameworks:

- Framework-specific functions or classes that handle common server tasks, such as request routing, middleware execution, and response generation.

Here's an example using Python and the `socket` module:

```
python
import socket
# Create a socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Bind the socket to a specific address and port
server_socket.bind(("127.0.0.1", 8080))
# Listen for incoming connections
server_socket.listen()
# Accept a connection
client_socket, client_address = server_socket.accept()
# Receive data from the client
data = client_socket.recv(1024)
# Send a response to the client
client_socket.sendall(b"Hello, client!")
# Close the sockets
client_socket.close()
server_socket.close()
```

6. What are the Client side commands/library functions used and mention their purpose?

Answer: Client-side programming for network communication involves using various commands or library functions to establish connections, send requests, and receive responses.

The specific commands or functions can vary based on the programming language and the libraries or frameworks being used. Here are common client-side commands and functions along with their purposes:

1. Socket Creation and Configuration:

- `socket()`: Creates a new socket for communication.
- `connect()`: Initiates a connection to a server using a specified IP address and port.
- `bind()`:Used in some scenarios where the client needs to bind to a specific local address.

2. Data Exchange:

- `send()`: Sends data over a connected socket to the server.
- `recv()`: Receives data from the server through the connected socket.
- `sendto()`: Sends data to a specific address using a connectionless socket (e.g., UDP).
- `recvfrom()`: Receives data from a specific address using a connectionless socket.

3. Connection Management:

- `close()`: Closes the socket after communication is complete.

4. Address Conversion:

- `getaddrinfo()`: Translates human-readable addresses (such as domain names) into a form suitable for use with network operations.
- `inet_pton()`: Converts a human-readable IP address to a network address structure.

5. SSL/TLS (Secure Sockets Layer/Transport Layer Security):

- If secure communication is required, client-side libraries may include functions for initializing secure connections, certificate verification, etc.

6. Error Handling:

- `perror()`: Prints a description for the last error that occurred.
- `strerror()`: Returns a pointer to the textual representation of the current errno value.

7. Asynchronous I/O (Optional):

- Libraries or functions for handling asynchronous events or non- blocking I/O, if asynchronous behavior is required.

8. HTTP(S) Libraries (for Web Clients):

- Functions for making HTTP requests and handling responses, such as `fetch()` in JavaScript or libraries like `requests` in Python.

9. FTP Libraries (for FTP Clients):

- Functions for handling File Transfer Protocol (FTP) operations, such as connecting to an FTP server, uploading, and downloading files.

10. WebSockets (for Real-Time Communication):

- Libraries or functions for establishing WebSocket connections and handling real-time bidirectional communication.

Here's an example using Python and the `socket` module for a simple TCP client:

python

```
import socket
# Create a socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# Connect to the server
server_address = ("127.0.0.1", 8080)
```

```
client_socket.connect(server_address)
# Send data to the server
client_socket.sendall(b"Hello, server!")
# Receive a response from the server
data = client_socket.recv(1024)
print("Received:", data.decode("utf-8"))
# Close the socket
client_socket.close()
```

7. What is a Remote procedure call?

Answer: A Remote Procedure Call (RPC) is a protocol that enables a program to cause a procedure (subroutine) to execute in another address space (commonly on another machine) as if it were a local procedure call without the programmer explicitly coding the details for the remote communication. In other words, an RPC allows a program to invoke procedures on a remote server, as if they were local procedures, making it easier to develop distributed applications.
Key features and concepts of Remote Procedure Calls include:

1. Procedure Invocation:

- An RPC allows a program to invoke a procedure (function or method) that resides on a remote machine, as if it were a local procedure call.
- The programmer writes code as if they were making a regular function call, and the underlying RPC system takes care of the communication details.

2. Transparent Communication:

- The goal of RPC is to provide transparent communication, meaning that the programmer doesn't need to worry about the intricacies of network communication or data serialization.
- Parameters are passed to the remote procedure, and results are returned as if the call was local.

3. Stub or Proxy:

- A key component of RPC is the use of a stub or proxy on the client side and a corresponding skeleton on the server side.
- The client-side stub intercepts the local procedure call, packages the parameters, and sends them to the server. The server-side skeleton unpacks the parameters, invokes the actual procedure, and sends back the results.

4. Data Marshalling:

- The process of converting complex data types or objects into a format that can be easily transmitted over a network is called data marshaling.
- Data marshaling is handled by the RPC system to ensure that data can be transmitted and reconstructed correctly on both ends of the communication.

8. What is connection oriented and connectionless communication? Which protocols are used?

Answer: Connection-oriented and connectionless are two communication paradigms used in networking to facilitate the exchange of data between devices. These paradigms are associated with specific network protocols that govern the way data is transmitted. Here's an overview of each:

Connection-Oriented Communication:

1. Characteristics:

- Reliability: Emphasizes the reliable, ordered delivery of data.
- Connection Setup: Requires a setup phase before data exchange, where a connection is established between the sender and receiver.
- Stream-Based: Operates in a stream of data, ensuring that data arrives in the correct order without duplication or loss.

2. Protocols:

- TCP (Transmission Control Protocol): The most common example of a connection-oriented protocol. Used in applications where reliability and order of delivery are crucial, such as web browsing, file transfer (FTP), and email (SMTP).

3. Usage Scenarios:

- Suitable for applications where the integrity and order of data are essential, and the cost of establishing and maintaining a connection is acceptable.

Connectionless Communication:

1. Characteristics:

- Less Overhead: Minimal setup is required before data transmission.
- Unreliable: Does not guarantee the order of delivery or that data will be received at all.
- Datagram-Based: Operates on discrete packets (datagrams) of information.

2. Protocols:

- UDP (User Datagram Protocol): A common example of a connectionless protocol. Used in scenarios where real-time communication is crucial, such as online gaming, streaming media, and DNS.

3. Usage Scenarios:

- Suitable for applications that can tolerate some data loss and don't require a continuous, reliable stream. Real-time applications often prefer connectionless communication due to lower latency.

****