Multithreaded Client-Server Model in Python

. Introduction to Client-Server Architecture

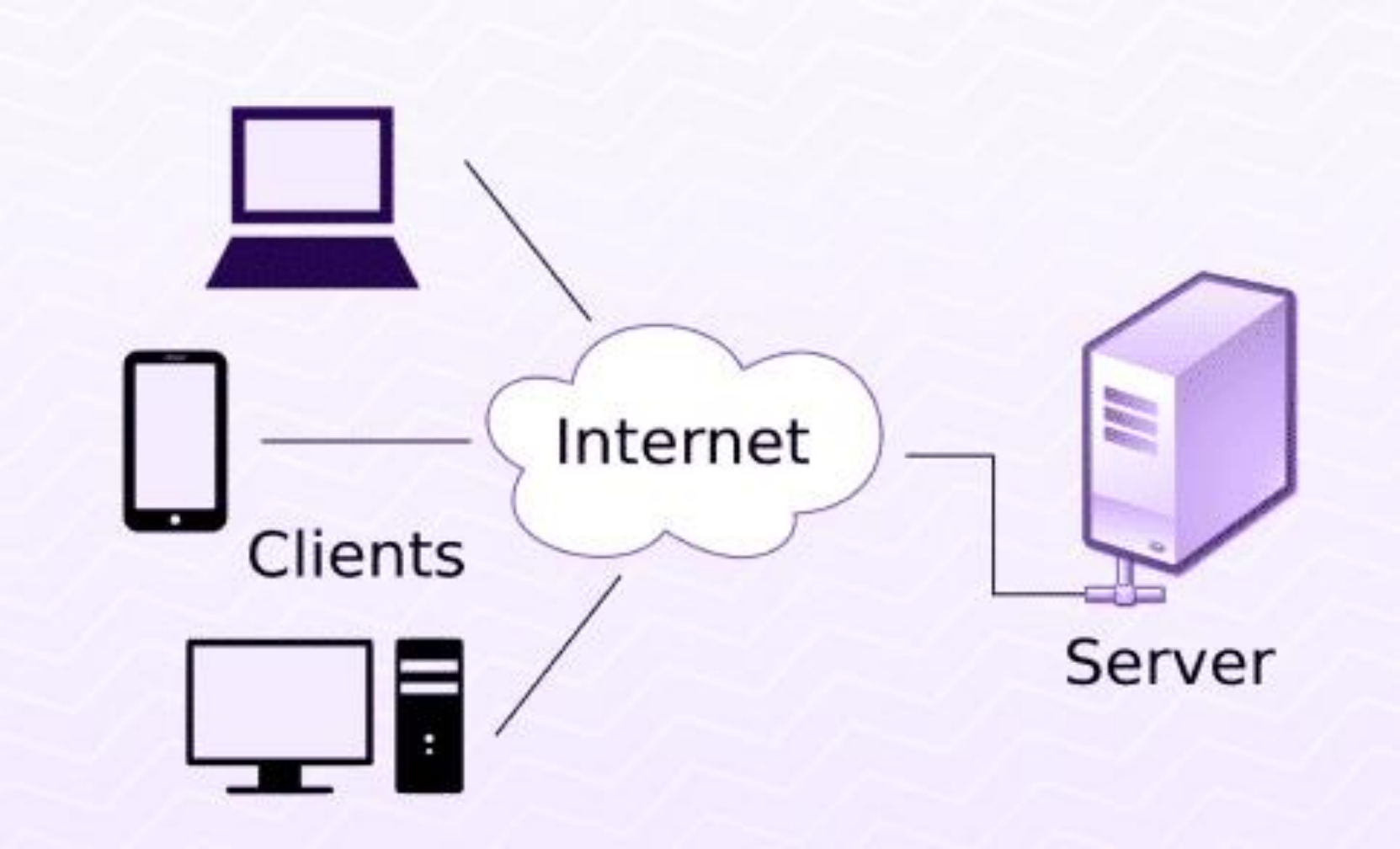
What is Client-Server Architecture?

Client-Server Model: A distributed application structure that divides tasks between service providers (servers) and service requesters (clients).

Server: Hosts, delivers, and manages resources or services (e.g., web servers, database servers).

Client: Requests services or resources from the server (e.g., web browsers, mobile apps).

Basic Client-Server Diagram:



Introduction to Multithreading

What is Multithreading?

Multithreading: The ability of a CPU (or a single program) to manage its use by more than one user at a time, or to manage multiple requests by the same user without having to have multiple copies of the programming running in the computer.

Thread: The smallest sequence of programmed instructions that can be managed independently by a scheduler.

Benefits of Multithreading in Client-Server Models

Concurrency: Handle multiple client requests simultaneously.

Improved Performance: Better resource utilization and responsiveness.

Scalability: Ability to manage a large number of clients efficiently.

Python's threading Module
Overview

Python provides the threading module to create and manage threads.

Useful for I/O-bound tasks where threads can run concurrently.

Key Functions and Classes

- **threading.Thread()**: Creates a new thread.
- **start()**: Begins the thread's activity.
- **join()**: Waits for the thread to finish.
- **is_alive()**: Checks if the thread is still running.
- **Lock()**: Ensures thread safety when accessing shared resources.

Key Threading Functions Explained

start() Function

Purpose: Begins the execution of a thread by invoking the thread's run() method.

```
thread = threading.Thread(target=some_function)
thread.start()
```

Behavior: Once started, the thread runs independently from the main program.

join() Function

Purpose: Blocks the calling thread until the thread whose join() method is called is terminated.

thread.join()

Lock() Function

Purpose: Provides a mechanism to synchronize threads, ensuring that only one thread can access a resource at a time.

Usage:

```
lock = threading.Lock()
lock.acquire()
# critical section
lock.release()
```

Behavior: Prevents race conditions by controlling access to shared resources.

Developing a General Client-Server Model in Python

Steps to Develop a Client-Server System

Designing the Communication Protocol:

- Define how client and server communicate (e.g., TCP, HTTP).
- Choose data format (e.g., plain text, JSON).

Server-Side Development:

- **Socket Initialization**: Create a socket, bind to an IP and port, and listen for connections.
- **Handling Requests**: Accept connections and process client requests.
- **Response**: Send responses back to clients.

Client-Side Development:

- **Socket Creation**: Create a socket and connect to the server.
- **Sending Requests**: Send data or requests to the server.
- **Receiving Responses**: Receive and process data from the server.

General Multithreaded Client-Server Application Template

Server Code (General Purpose)

```
import socket  # For networking and communication
import threading  # For handling multiple clients concurrently
# Function to handle each connected client
def handle_client(client_socket, client_address):
    print(f"Connection established with {client_address}")
        try:
        while True:
            # Receive data from the client
            data = client_socket.recv(1024)
            if not data:                # If no data is received, break the loop (client has disconnected)
                break
            print(f"Received data from {client_address}: {data.decode('utf-8')}")
           # Process the received data (general processing logic can be added here)
            response = process_data(data)
            # Send the processed data back to the client
            client_socket.send(response)
    except Exception as e:
        print(f"Error with client {client_address}: {e}")
    finally:        # Close the connection to the client
        client_socket.close()
```

General Multithreaded Client-Server Application Template

```
    # Function for processing the received data (customize based on your use case)
    def process_data(data):
        # Placeholder for data processing logic
        # This is where you would implement the core logic to process requests
        # For now, we'll just echo the data back as a response
        return data
    # Main server function
    def start_server():
        # Create a socket (IPv4, TCP)
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
         # Bind the socket to an IP address and port
        server_socket.bind(('0.0.0.0', 9999))  # Listening on all interfaces on port 9999
        # Start listening for connections (maximum of 5 queued connections)
        server_socket.listen(5)
        print("Server is listening on port 9999...")
        while True:
            # Accept a new client connection
            client_socket, client_address = server_socket.accept()
            print(f"Accepted connection from {client_address}")
            # Create a new thread to handle the client
            client_handler = threading.Thread(target=handle_client, args=(client_socket,
    client_address))
            client_handler.start()
    # Start the server
```

Explanation of the General Server Code

- **Socket Setup**:
  - A TCP/IP socket is created using **socket.AF_INET** and **socket.SOCK_STREAM**, which specifies an IPv4 address and a TCP connection.
  - **server_socket.bind(('0.0.0.0', 9999))**: The server binds to all available interfaces (0.0.0.0) on port 9999.
- **Listening for Connections**:
  - The server starts listening for incoming connections using **server_socket.listen(5).** The 5 indicates the maximum number of queued connections.
  - It enters an infinite loop, waiting to accept incoming client connections.
- **Accepting and Handling Clients**:
  - **server_socket.accept()**: When a client connects, the server accepts the connection, and the client is assigned its own socket (client_socket).
  - A separate thread is created for each client using threading.Thread, which calls the handle_client function to manage communication with that specific client.
- **Client Communication (handle_client function)**:
  - In this function, the server receives data from the client using client_socket.recv(1024) and processes it using the process_data() function. You can customize the processing logic based on your use case (e.g., handling file uploads, performing calculations, or interacting with databases).
  - After processing, the server sends a response back to the client using client_socket.send(response).
- **Processing Logic (process_data function)**:

  The process_data() function is a placeholder where you can implement your custom logic for handling different types of data. In this example, the data is simply echoed

Client Code (General Purpose)

```
import socket  # For networking and communication
# Main client function
def start_client():
    # Create a socket (IPv4, TCP)
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Connect to the server (specify the IP address and port of the server)
    client_socket.connect(('127.0.0.1', 9999))  # Assuming the server is running on
localhost
    try:
        while True:
            # Get the data to send to the server
            data_to_send = input("Enter data to send to the server (or 'exit' to quit): ")
            if data_to_send.lower() == 'exit':
                break
            client_socket.send(data_to_send.encode('utf-8')) # Send the data to the server
            response = client_socket.recv(1024) # Receive the response from the server
            print(f"Received from server: {response.decode('utf-8')}")
    except Exception as e:
        print(f"Error occurred: {e}")
    finally:        # Close the connection to the server
        client_socket.close()
if __name__ == "__main__": # Start the client
    start_client()
```

Explanation of the General Client Code

- **Socket Setup**:
  - The client creates a TCP/IP socket using socket.AF_INET and socket.SOCK_STREAM.
- **Connecting to the Server**:
  - **client_socket.connect(('127.0.0.1', 9999))**: The client connects to the server at the specified IP address (127.0.0.1, which is localhost) and port (9999).
- **Data Exchange**:
  - The client enters a loop where it prompts the user for data to send to the server.
  - **client_socket.send(data_to_send.encode('utf-8'))**: The user input is sent to the server after being encoded into bytes.
  - **client_socket.recv(1024)**: The client waits for a response from the server and then prints it.
- **Closing the Connection**:
  - The client breaks out of the loop and closes the connection when the user enters 'exit'.

Key Concepts in General Client-Server Applications

- **Key Concepts in General Client-Server Applications**
- **Concurrency**:
  - In this multithreaded server, each client is handled in its own thread, meaning multiple clients can communicate with the server simultaneously.
- **Threading**:
  - The threading module is used to create threads. Each client is managed in its own thread, so the server can handle multiple clients without blocking.
- **Data Processing**:
  - The **process_data()** function in the server can be modified to handle various types of requests or data, such as performing computations, handling file uploads, or interacting with databases.
- **Socket Programming**:
  - Sockets provide the mechanism for communication between the client and the server. TCP sockets ensure reliable, ordered, and error-checked data transmission.
- **Message Encoding and Decoding**:
  - Data sent between the client and server is encoded using UTF-8 before being transmitted and then decoded upon receipt.

Example Use Cases for This Template

- **File Transfer**:
  - You can modify the process_data() function to handle file uploads and downloads, where the client sends files and the server processes them.
- **Remote Calculation**:
  - You can turn this template into a remote calculator, where the client sends mathematical operations, and the server performs the calculations and returns the results.
- **Database Interaction**:
  - The server can be connected to a database, and the client can send requests to fetch, insert, update, or delete data from the database.
- **IoT Applications**:
  - In an IoT system, the client (an IoT device) can send sensor data to the server, which processes it and responds accordingly (e.g., storing it or controlling another device).

Homework

- Rewrite the calculator client-server application in slide 20&21according to the template that you learned above.

Example 1: Multithreaded Client-Server Calculator

```
import socket
import threading
def handle_client(client_socket):
    try:
        data = client_socket.recv(1024).decode()
        operation, a, b = data.split()
        a, b = int(a), int(b)
        if operation == 'add':
            result = a + b
        elif operation == 'subtract':
            result = a - b
        elif operation == 'multiply':
            result = a * b
        elif operation == 'divide':
            result = a / b
        client_socket.send(str(result).encode())
    except Exception as e:
        client_socket.send(b'Error')
    finally:
        client_socket.close()
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('0.0.0.0', 9999))
server.listen(5)
while True:
    client, addr = server.accept()
    thread = threading.Thread(target=handle_client, args=(client,))
    thread.start()
```

Client Code for Calculator

```
import socket

client = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
client.connect(('127.0.0.1', 9999))

operation = input("Enter operation (add/subtract/multiply/divide): ")
a = input("Enter first number: ")
b = input("Enter second number: ")

client.send(f"{operation} {a} {b}".encode())
response = client.recv(4096)
print(f"Result: {response.decode()}")

client.close()
```