A) TCP HTTP Server

Overview:

This system is a basic HTTP server implemented using Python's `socket` module, which serves static HTML files over a TCP connection. It listens for incoming HTTP requests from a client (typically a web browser) and responds with the requested content or an error message if the resource is not found. The server can handle multiple requests sequentially.

Assumptions:

1. **IPv4 Addressing**: The server runs on a local network using an IPv4 address. It assumes a pre-configured function `get_ipv4_address()` is used to fetch the server's IP address dynamically.
2. **TCP Protocol**: The server uses the TCP protocol for reliable communication with clients.
3. **Static File Serving**: The server is designed to serve only static HTML files located in the server's directory, with a `Content-Type` of `text/html`.
4. **Sequential Handling of Requests**: The server handles client requests sequentially (one at a time). Concurrent connections are not handled in this design.
5. **Error Handling**: If a requested file is not found, the server returns a custom `404 Not Found` page.

Design:

1. **Socket Setup**:
    - The server creates a welcoming socket using TCP (`socket(AF_INET, SOCK_STREAM)`).
    - The `serverSocket.listen(1)` method puts the server into a listening state, where it can accept one queued connection request at a time.
2. **Connection Handling**:
    - When a client connects, the server creates a new `connectionSocket` for that specific client.
    - The server retrieves the HTTP GET request sent by the client via the `connectionSocket.recv()` method and decodes it from bytes to a string for processing.
3. **Request Parsing**:
    - The HTTP request is parsed to extract the requested resource path (e.g., `/HelloWorld.html`).
    - The file path is determined by stripping the leading `/` from the resource path.
4. **File Handling**:
    - If the requested file exists in the server directory, it is opened and its content is read into memory.

    - If the file is not found, the server responds with a predefined `404.html` error page.
5. **Response Generation**:
    - The server sends an HTTP response status line (`HTTP/1.1 200 OK` or `HTTP/1.1 404 Not Found`).
    - HTTP headers (such as `Date`, `Server`, `Content-Length`, and `Content-Type`) are constructed and sent to the client.
    - The content of the file (either the requested file or the `404.html` error page is then sent byte-by-byte to the client.
6. **Connection Management**:
    - After the response is sent, the connection to the client is closed to free up resources for future connections.
    - The server remains running and continues to listen for new requests until manually terminated.

Limitations:

- **Single Client at a Time**: The current design can only handle one client connection at a time. For scaling, the server would need to support multithreading or asynchronous I/O.
- **No Security Features**: This server is basic and does not include SSL/TLS for encrypted communication or any authentication mechanisms.

B) MultiThread TCP HTTP Server

To allow the server to handle multiple client requests concurrently, the following modifications were made to the original single-threaded design:

1. **Threading for Concurrent Connections**:
    - The server now spawns a new thread for each client connection, ensuring that multiple clients can be served simultaneously.
    - This was achieved using Python's `Thread` class from the `threading` module.
2. The main server loop (`start_server`) creates a new thread for every incoming client connection using the `Thread` function, where each thread runs the `handle_client()` function. This function is responsible for processing the client's request and sending back the response.
3. **Client Handling Moved to a Separate Function (`handle_client`)**:
    - The code for handling client connections (previously inside the main loop) was moved into a separate function, `handle_client`. This modularizes the code and makes it easier to handle each client independently in its own thread.

4. The function takes the client's socket (`connectionSocket`), address (`clientAddress`), and a `Thread_count` to identify which thread is processing the request.
5. **Thread Identification with `Thread_count`**:
    - To track individual client connections and which thread is handling them, a `Thread_count` variable was introduced. It increments with each new client connection to provide unique identifiers for threads.
6. This allows the server to print detailed messages such as `T1> Connection established...` to identify actions related to each thread.
7. **Increased Connection Queue**:
    - The server now listens for up to 5 queued connections using the `listen` method, allowing the server to accept a larger number of concurrent clients. This ensures multiple clients can be connected while waiting for their requests to be processed.

C) TCP HTTP Client

Overview:

This system is a simple TCP client designed to send HTTP GET requests to a server and receive HTTP responses. The client connects to a specified server and port, requests a file, and displays the server's response, which can be an HTML file or an error message (e.g., 404 Not Found).

Assumptions:

1. **Command-Line Arguments**:
    - The client requires three command-line arguments:
        - `server_host`: The server's hostname or IP address.
        - `server_port`: The port number on which the server is listening.
        - `filename`: The file requested from the server.
2. Example usage: `python TCPclient.py localhost 12000 HelloWorld.html`
3. **TCP Protocol**: The client uses the TCP protocol for reliable communication with the server. The connection is maintained until the full response is received from the server.
4. **HTTP Request**: The client constructs an HTTP GET request to request a resource (e.g., an HTML file) from the server. The request is formatted according to the HTTP/1.1 standard.
5. **Basic Error Handling**: The client handles errors such as connection failure by printing an error message and terminating.

Design:

1. **Command-Line Interface**:
    - The client expects three arguments: the server's hostname, port number, and the name of the file to be retrieved. It ensures these are provided, otherwise it exits with a usage message.
2. **Socket Creation**:
    - A TCP client socket is created using the `socket(AF_INET, SOCK_STREAM)` function to establish a reliable connection to the server.
3. **Connection Establishment**:
    - The client attempts to connect to the server using the `connect()` method. If the connection fails, an error message is printed, and the client exits.
4. **HTTP Request Construction**:
    - The client constructs an HTTP GET request, consisting of:
        - **Request Line**: This specifies the HTTP method (`GET`), the requested file, and the HTTP version (`HTTP/1.1`).
        - **Request Headers**: A set of headers is added, including `Host`, `User-Agent`, and `Accept`. These provide additional information about the request and client capabilities.
5. **Sending the Request**:
    - The constructed HTTP request is sent in two parts: the request line and the headers. Both are encoded from strings to bytes and sent over the socket to the server.
6. **Receiving and Processing the Response**:
    - The client continuously receives data from the server in chunks (using `recv(2048)`) and decodes the bytes into a string. The received data is appended to the `HTTP_response_message` variable until the server stops sending data (i.e., no more data to receive).
    - The entire HTTP response (status line, headers, and body) is then printed to the console.
7. **Connection Closure**:
    - After receiving the response, the client closes the connection with the server using the `close()` method to free resources.

Does this client work with server implemented in Part 1 or Part 2 or both?:

Yes