To incorporate the concept of **forking** in your server-client socket program, let's first understand how the `fork()` system call works and how it will be useful for handling multiple client connections simultaneously.

Overview of Forking and Its Role in Server-Client Communication

In a server-client application, **forking** allows the server to create a new process (child process) for each incoming client connection. This means that the server doesn't block on one client; it can handle multiple clients concurrently. The parent process (the main server) continues to listen for new connections, while the child process handles the communication with the connected client.

How Forking Works:

1. **Forking** is used to create a new process that is a duplicate of the parent process.
   - The parent process continues its execution after calling `fork()`.
   - The child process starts executing the same code but with a `fork()` return value of `0`, which helps differentiate between the two.
2. The parent process gets the **PID of the child** (a positive integer), while the child process gets `0`.
3. Both processes inherit the resources of the parent, including open file descriptors (such as sockets).

Using Fork with Socket Programming:

For a socket server, once a client is accepted, the server creates a **child process** that handles the communication. The parent process continues to listen for new connections.

Modifying Your Code to Use Fork:

We'll modify your **server code** to **fork** a new process each time a client connects. The server will then handle each client in the child process while the parent process listens for new connections.

Forking in the Server Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <time.h>

#define PORT 9999
#define IP "10.0.2.15"

int main()
{
    printf("Welcome to NP-Socket Lab-1 (Server with Fork)\n");

    int fd, confd;                      // File descriptors for the server and client sockets
    char str[INET_ADDRSTRLEN];          // Buffer to store client IP address
    struct sockaddr_in server, client; // Socket address structures
    char sendBuff[1024];                // Buffer for sending data
    time_t ticks;                       // Variable to store current time

    // Create a socket
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0)
    {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Socket created with descriptor [%d]\n", fd);

    // Initialize server structure and buffers
    memset(&server, 0, sizeof(server));
    memset(sendBuff, 0, sizeof(sendBuff));

    // Configure server settings
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr(IP);
    server.sin_port = htons(PORT);

    // Bind the socket to the IP and port
    if (bind(fd, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        perror("Bind failed");
        close(fd);
        exit(EXIT_FAILURE);
    }
    printf("Socket successfully bound to port %d\n", PORT);

    // Listen for incoming connections
    if (listen(fd, 5) < 0)
    {
        perror("Listen failed");
        close(fd);
        exit(EXIT_FAILURE);
    }
    printf("Server is listening for connections...\n");

    // Accept a client connection and fork a new process
    socklen_t clientLen = sizeof(client);
    while (1) {
        confd = accept(fd, (struct sockaddr *)&client, &clientLen);
        if (confd < 0)
        {
            perror("Accept failed");
            continue; // If accept fails, try again (keep server running)
        }

        // Fork to handle multiple clients
        pid_t pid = fork();
        if (pid < 0) {
            perror("Fork failed");
            close(confd);
            continue;
        }

        if (pid == 0) {
            // In child process: Handle the client
            close(fd);  // Close the server socket in child process, no longer needed

            // Get the current time and send it to the client
            ticks = time(NULL);
            snprintf(sendBuff, sizeof(sendBuff), "Current time: %.24s\r\n", ctime(&ticks));
            write(confd, sendBuff, strlen(sendBuff));
            printf("Sent time to client: %s", sendBuff);

            // Close the client socket after communication
            close(confd);
            exit(0); // Terminate the child process after handling the client
        }
        else {
            // In parent process: Close the client socket, continue to listen
            close(confd);
        }
    }

    // Close the server socket
    close(fd);
    return 0;
}
```

Changes and Explanation:

1. **Forking in the Server:**
   - After accepting a client connection, the server calls `fork()`.
   - The **child process** (where `pid == 0`) handles the client by sending the current time.
   - The **parent process** (where `pid > 0`) closes the client connection but continues to listen for new clients.
2. **Handling Multiple Clients:**
   - This forking mechanism allows the server to handle multiple clients simultaneously. Each new connection is handled by a new child process, while the parent process remains free to accept new connections.
3. **Close Server Socket in Child Process:**
   - The child process closes the server socket (`fd`) because it doesn't need to listen for new clients—its job is to communicate with the connected client.
4. **Graceful Termination of Child Process:**
   - The child process exits after handling the client (`exit(0)`), so it doesn't continue running unnecessarily.

Modifying the Client Code (No Forking Needed in Client):

The **client** code doesn't need any modification to support forking because the forking happens in the server. The client will just connect to the server and receive the message.

Here is your **client code** (unchanged):

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

#define PORT 9999
#define IP "10.0.2.15"

int main()
{
    printf("Welcome to NP-Socket Lab-2 Client\n");

    int sockfd;                    // File descriptor for the socket
    struct sockaddr_in serverAddr; // Server address structure
    char sendBuff[1024];           // Buffer for sending data
    char recvBuff[1024];           // Buffer for receiving data

    // Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Socket created with descriptor [%d]\n", sockfd);

    // Initialize server address structure
    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);

    // Convert and set the IP address
    if (inet_pton(AF_INET, IP, &serverAddr.sin_addr) <= 0)
    {
        perror("Invalid address or address not supported");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    // Request a connection to the server
    if (connect(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0)
    {
        perror("Connection to server failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
    printf("Connected to server [%s:%d]\n", IP, PORT);

    // Receive a message from the server
    memset(recvBuff, 0, sizeof(recvBuff));
    if (read(sockfd, recvBuff, sizeof(recvBuff) - 1) < 0)
    {
        perror("Read from server failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }
    printf("Message received from server: %s", recvBuff);

    // Close the socket
    close(sockfd);
    printf("Connection closed.\n");

    return 0;
}
```

Flow of Execution:

1. **Server Initialization:**
    - The server creates a socket and binds it to a specific port and IP address.
    - It then listens for incoming connections from clients.
2. **Forking for Client Connections:**
    - When a client attempts to connect, the server accepts the connection and forks a new process.
    - The **parent process** continues to listen for more clients.
    - The **child process** handles the connection by sending the current time to the client and then closes the connection.
3. **Client:**
    - The client creates a socket and connects to the server.
    - It sends a request to the server and waits for a response.
    - The server sends back the current time, and the client prints it.
    - The client then closes the connection.

Key Points:

- **Parent Process:** Listens for new connections and accepts them.

- **Child Process:** Handles communication with each client, sends data, and then exits.
- **`fork()` behavior:** The server creates a new child process for each incoming client, allowing the server to handle multiple clients concurrently without blocking.

This approach ensures that your server can handle multiple clients without needing to wait for one client to finish before accepting the next. Each client gets its own process (and therefore its own resources),

making the server scalable and efficient.