EC3022D Computer Networks Winter 2023-24

Socket Programming Assignment

SUBMITTED BY

ABHINAV RAJ V

B210759EC

EC01

Webserver using socket programming

Qs: Build a Web Server in C/Python using Socket programming. The web server should be able to respond to simple HTTP commands like GET, POST etc. When a GET request is sent, the server should respond with a page containing your name and roll number.

Submit a PDF report with the full code.

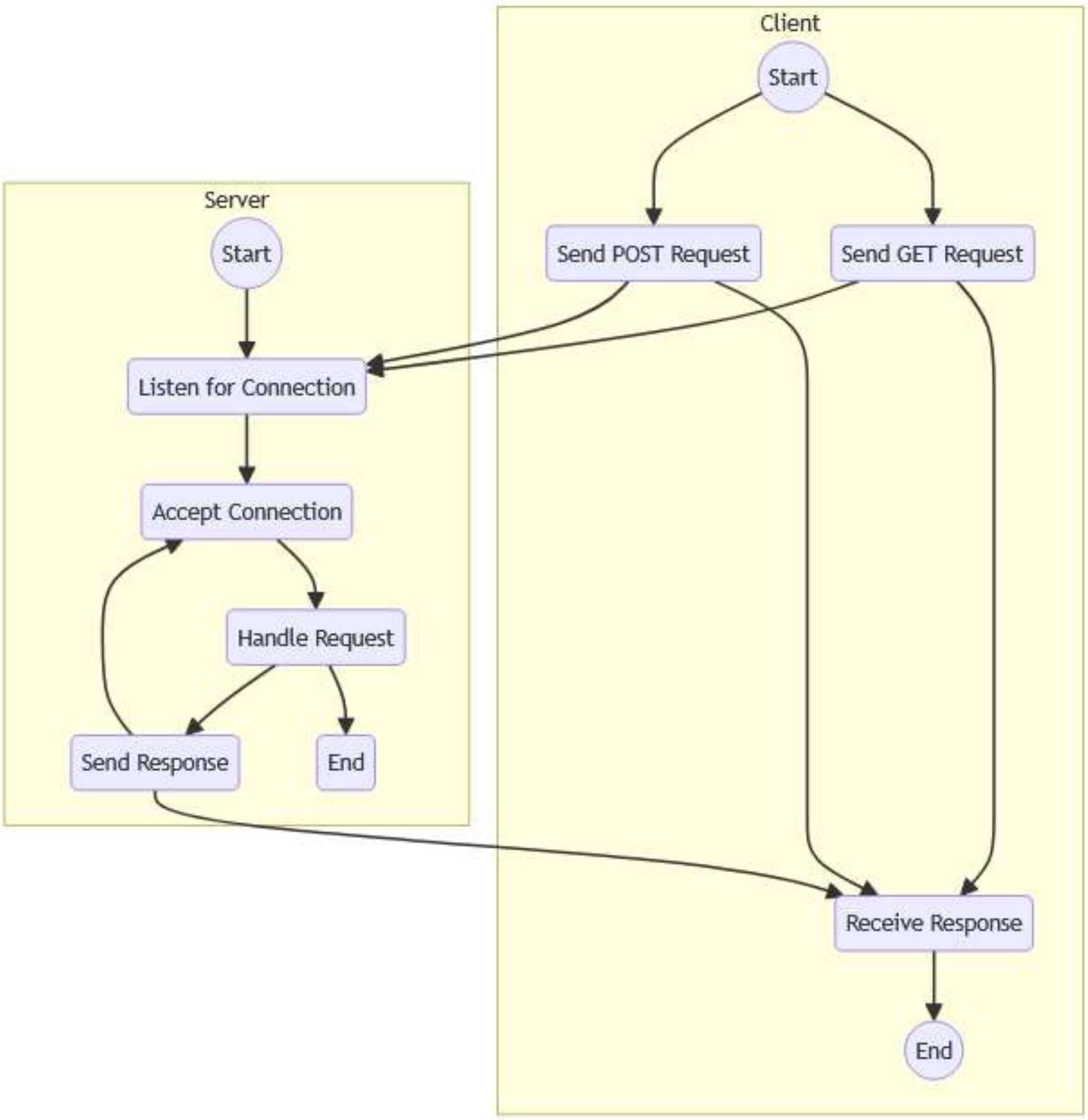
Server Code

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

#define PORT 8080

void handle_request(int client_socket) {
    char response[1024] = {0};
    char request[1024] = {0};

    ssize_t read_size = read(client_socket, request, sizeof(request) - 1);
    if (read_size < 0) {
        perror("read");
        return;
    }

    request[read_size] = '\0';
    printf("\nReceived: %s\n", request);

    if (strncmp(request, "GET", 3) == 0) {
        sprintf(response, "HTTP/1.1 200 OK\nContent-Type: 
text/html\n\n<html><body><h1>Name: ABHINAV RAJ V \n</h1><h2>Roll number: 
B210759EC </h2></body></html>");
    }
    else if (strncmp(request, "POST", 4) == 0) {
        sprintf(response, "HTTP/1.1 200 OK\nContent-Type: 
text/html\n\n<html><body><h1> POST request received </h1></body></html>");
    }
    else {
        sprintf(response, "HTTP/1.1 405 Method Not Allowed\nContent-Type: 
text/html\n\n<html><body><h1>Method Not Allowed</h1></body></html>");
    }

    send(client_socket, response, strlen(response), 0);
}

int main() {
    int server_fd, client_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    if (listen(server_fd, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    printf("Server listening on port %d...\n", PORT);

    while (1) {
        if ((client_socket = accept(server_fd, (struct sockaddr *)&address, 
(socklen_t *)&addrlen)) < 0) {
            perror("accept");
            continue;
        }

        handle_request(client_socket);
        close(client_socket);
    }

    return 0;
}
```

Client Code

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

#define PORT 8080
#define BUFFER_SIZE 1024

void send_request(const char *request) {
    int sockfd;
    struct sockaddr_in serv_addr;
    char buffer[BUFFER_SIZE];

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("Socket creation error");
        exit(EXIT_FAILURE);
    }

    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        perror("Invalid address/ Address not supported");
        exit(EXIT_FAILURE);
    }

    if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
{
        perror("Connection failed");
        exit(EXIT_FAILURE);
    }

    if (send(sockfd, request, strlen(request), 0) < 0) {
        perror("Send request failed");
        exit(EXIT_FAILURE);
    }

    int bytes_received;
    while ((bytes_received = recv(sockfd, buffer, BUFFER_SIZE - 1, 0)) > 0) {
        buffer[bytes_received] = '\0';
        printf("%s", buffer);
    }
    if (bytes_received < 0) {
        perror("Receive failed");
        exit(EXIT_FAILURE);
    }

    close(sockfd);
}

int main() {
    printf("Sending GET request:\n");
    send_request("GET / HTTP/1.1\r\nHost: localhost\r\nConnection:
close\r\n\r\n");

    printf("\n\nSending POST request:\n");
    send_request("POST / HTTP/1.1\r\nHost: localhost\r\nConnection:
close\r\nContent-Length: 0\r\n\r\n");

    return 0;
}
```

Block Diagram



Server Block Diagram:

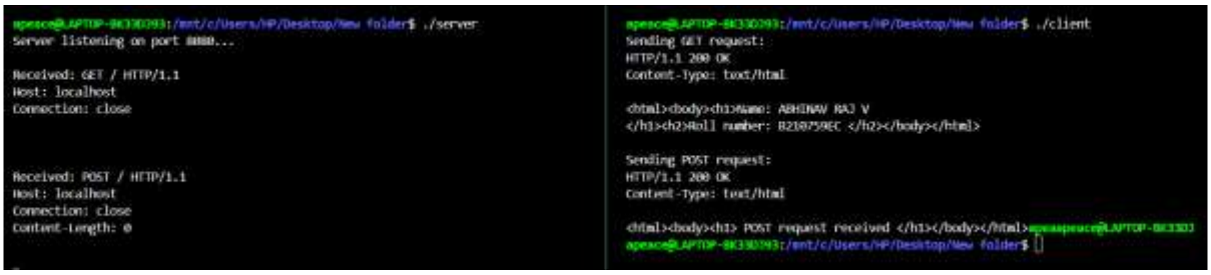
1. **Start:** This is the initial state where the server begins its operation.
2. **Listen for Connection:** The server waits for incoming connections from clients. It sets up a socket and listens on a specified port for incoming connection requests.
3. **Accept Connection:** Once a client initiates a connection, the server accepts it. This step involves establishing a connection socket for further communication with the client.
4. **Handle Request:** After accepting a connection, the server reads the incoming request from the client. It then processes the request, which may involve various tasks such as parsing the request, executing business logic, accessing databases, etc.
5. **Send Response:** Based on the request received and processed, the server formulates a response to send back to the client. This response typically contains the requested data or an acknowledgment of the completed action.
6. **Loop Back to Accept Connection:** After sending the response, the server goes back to accepting connections, waiting for new client requests. This loop continues as long as the server is running and accepting connections.
7. **End:** This represents the termination point of the server's operation. It indicates the end of the block diagram.

Client Block Diagram:

1. **Start:** This is the initial state where the client begins its operation.
2. **Send GET Request:** The client sends a GET request to the server. This request typically involves requesting data from the server.
3. **Send POST Request:** Alternatively, the client can send a POST request to the server. This request is often used for submitting data to the server.
4. **Receive Response:** After sending a request, the client waits to receive a response from the server. The response contains the requested data or an acknowledgment of the completed action.
5. **End:** This represents the termination point of the client's operation. It indicates the end of the block diagram.

OUTPUT

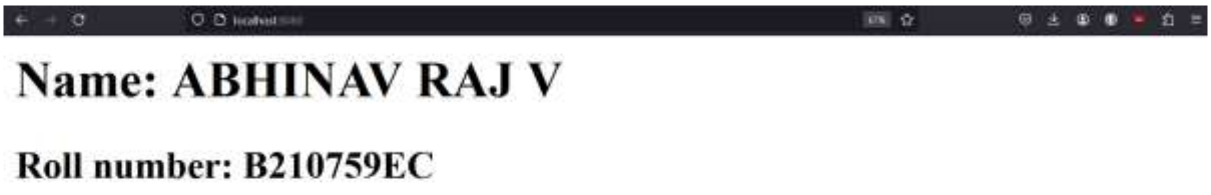
Server Client



```
apeace@LAPTOP-8K33D293:/mnt/c/Users/HP/Desktop/New folder$ ./server
Server listening on port 8080...

Received: GET / HTTP/1.1
Host: localhost
Connection: close



Received: POST / HTTP/1.1
Host: localhost
Connection: close
Content-Length: 0
```

```
apeace@LAPTOP-8K33D293:/mnt/c/Users/HP/Desktop/New folder$ ./client
Sending GET request:
HTTP/1.1 200 OK
Content-Type: text/html

<html><body><h1>Name: ABHINAV RAJ V
</h1><h2>Roll number: B210759EC </h2></body></html>

Sending POST request:
HTTP/1.1 200 OK
Content-Type: text/html

<html><body><h1> POST request received </h1></body></html>apeaapeace@LAPTOP-8K33D3
apeace@LAPTOP-8K33D293:/mnt/c/Users/HP/Desktop/New folder$
```

Webpage



Name: ABHINAV RAJ V

Roll number: B210759EC

Conclusion:

In conclusion, the presented server and client code showcase a basic implementation of web server and client functionality using socket programming in C. The server actively listens for incoming connections, handles both GET and POST requests and returns suitable responses to the clients. Likewise, the client initiates connections to the server, transmits predefined requests, and exhibits the responses received from the server.