LAB - 6 COMPUTER NETWORK

Report: Multi-Client Chat Application Using Socket Programming in C

Aditya Agarwal
Roll No - 14

---

1. Arithmetic Operations with Multi-Client Support

SERVER:

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

#define PORT 65432
#define BUFFER_SIZE 1024

void *handle_client(void *client_socket_ptr) {
    int client_socket = *(int *)client_socket_ptr;
    free(client_socket_ptr);

    char buffer[BUFFER_SIZE];
    int num1, num2;
    char op;
    char result[BUFFER_SIZE];

    // Read data from client
    recv(client_socket, buffer, BUFFER_SIZE, 0);

    // Parse the data
    sscanf(buffer, "%d %c %d", &num1, &op, &num2);

    // Perform arithmetic operation
    switch (op) {
        case '+':
            snprintf(result, BUFFER_SIZE, "%d", num1 + num2);
            break;
        case '-':
            snprintf(result, BUFFER_SIZE, "%d", num1 - num2);
            break;
        case '*':
            snprintf(result, BUFFER_SIZE, "%d", num1 * num2);
            break;
        case '/':
            if (num2 == 0) {
                snprintf(result, BUFFER_SIZE, "Error: Division by zero");
            } else {
                snprintf(result, BUFFER_SIZE, "%d", num1 / num2);
            }
            break;
```

CLIENT:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define PORT 9999
#define BUFFER_SIZE 1024

void *receive_messages(void *socket_desc) {
    int sock = *(int *)socket_desc;
    char buffer[BUFFER_SIZE];
    int read_size;

    while ((read_size = recv(sock, buffer, sizeof(buffer) - 1, 0)) > 0) {
        buffer[read_size] = '\0';
        printf("Received: %s\n", buffer);
    }

    close(sock);
    return NULL;
}

int main() {
    int sock;
    struct sockaddr_in server_addr;
    pthread_t recv_thread;
    char message[BUFFER_SIZE];

    // Create socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("Socket creation error");
        exit(EXIT_FAILURE);
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(PORT);
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

    // Connect to server
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Connection failed");
        exit(EXIT_FAILURE);
```

```
36
37      server_addr.sin_family = AF_INET;
38      server_addr.sin_port = htons(PORT);
39      server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
40
41      // Connect to server
42      if (connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
43          perror("Connection failed");
44          exit(EXIT_FAILURE);
45      }
46
47      // Start a thread to receive messages
48      pthread_create(&recv_thread, NULL, receive_messages, (void *)&sock);
49      pthread_detach(recv_thread);
50
51      // Main loop to send messages
52      while (1) {
53          printf("Enter message: ");
54          fgets(message, sizeof(message), stdin);
55          message[strcspn(message, "\n")] = '\0'; // Remove newline character
56
57          send(sock, message, strlen(message), 0);
58      }
59
60      close(sock);
61      return 0;
62  }
63
```

Client and Server Communication

In this implementation, the server accepts multiple clients, where each client sends an arithmetic operation for the server to process. The server uses threads to handle multiple clients concurrently. Here's a breakdown of the solution:

Key Features:

- **Threaded Server**: The server uses `pthread` to handle multiple clients simultaneously, ensuring parallel execution without blocking other clients.
- **Arithmetic Parsing**: The server receives a string input from the client, parses it using `sscanf()`, and performs the arithmetic operation based on the operator (`+`, `-`, `*`, `/`).
- **Error Handling**: Division by zero and invalid operators are handled with appropriate error messages sent back to the client.
- **Concurrency**: Each client interaction is processed in a separate thread using the `pthread_create()` function, ensuring that one client's request does not block others.

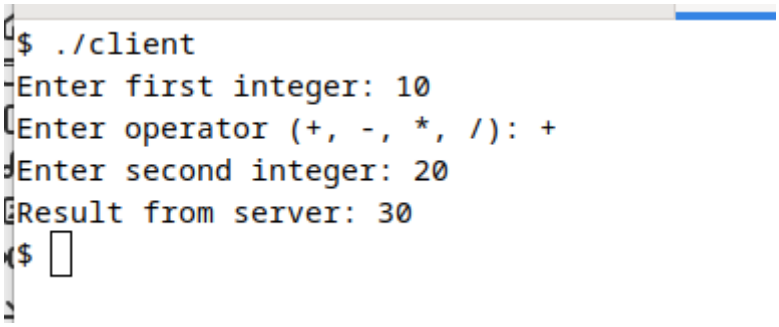
Flow:

1. **Client Input**: The client sends an arithmetic operation like `5 + 3`.
2. **Server Processing**: The server processes the input and returns the result.
3. **Client Output**: The client displays the result, e.g., `8`.

Key Code Snippets:

- Server processing arithmetic operations in `handle_client()`.
- Thread management using `pthread_create()` for concurrency.

OUTPUT:



---

2. Sentence Deduplication and Multi-Client Communication

SERVER:

```c
        }
    }

    // Copy the result back to the original sentence buffer
    strncpy(sentence, temp_sentence, BUFFER_SIZE - 1);
    sentence[BUFFER_SIZE - 1] = '\0'; // Null-terminate the string
}

void *handle_client(void *client_socket_ptr) {
    int client_socket = *(int *)client_socket_ptr;
    free(client_socket_ptr);

    char buffer[BUFFER_SIZE];

    // Receive the sentence from the client
    ssize_t bytes_received = recv(client_socket, buffer, BUFFER_SIZE - 1, 0);
    if (bytes_received < 0) {
        perror("Receive failed");
        close(client_socket);
        return NULL;
    }
    buffer[bytes_received] = '\0'; // Null-terminate the string

    // Remove duplicate words from the sentence
    remove_duplicates(buffer);

    // Send the processed sentence back to the client
    if (send(client_socket, buffer, strlen(buffer), 0) < 0) {
        perror("Send failed");
    }

    close(client_socket);
    return NULL;
}

int main() {
    int server_socket, client_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    pthread_t thread_id;

    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
        perror("Socket creation failed");
```

```
92
93      memset(&server_addr, 0, sizeof(server_addr));
94      server_addr.sin_family = AF_INET;
95      server_addr.sin_addr.s_addr = INADDR_ANY;
96      server_addr.sin_port = htons(PORT);
97
98      if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
99          perror("Bind failed");
100         close(server_socket);
101         exit(EXIT_FAILURE);
102     }
103
104     if (listen(server_socket, 5) < 0) {
105         perror("Listen failed");
106         close(server_socket);
107         exit(EXIT_FAILURE);
108     }
109
110     printf("Server is listening on port %d...\n", PORT);
111
112     while (1) {
113         client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_addr_len);
114         if (client_socket < 0) {
115             perror("Accept failed");
116             continue;
117         }
118
119         printf("Accepted connection from client\n");
120
121         int *client_socket_ptr = malloc(sizeof(int));
122         *client_socket_ptr = client_socket;
123
124         if (pthread_create(&thread_id, NULL, handle_client, client_socket_ptr) != 0) {
125             perror("Thread creation failed");
126             close(client_socket);
127         }
128
129         pthread_detach(thread_id);
130     }
131
132     close(server_socket);
133     return 0;
134 }
135
```

CLIENT:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 65432
#define BUFFER_SIZE 1024

int main() {
    int client_socket;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

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

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_addr.sin_port = htons(PORT);

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

    // Get user input
    printf("Enter a sentence: ");
    fgets(buffer, BUFFER_SIZE, stdin);
    buffer[strcspn(buffer, "\n")] = '\0'; // Remove newline character

    // Send the sentence to the server
    if (send(client_socket, buffer, strlen(buffer), 0) < 0) {
        perror("Send failed");
        close(client_socket);
        exit(EXIT_FAILURE);
    }

    // Receive the processed sentence from the server
```

Client and Server with Sentence Deduplication

In this task, the server accepts a sentence from a client and removes duplicate words, returning the processed sentence back. This also supports multiple clients through threading.

Key Features:

- **Sentence Processing**: The function `remove_duplicates()` tokenizes the sentence into words, checks for duplicates, and reconstructs a sentence without any duplicates.
- **Threaded Server**: Just like the previous implementation, the server creates a separate thread for each client, ensuring that multiple clients can be served at the same time.
- **Tokenization and Deduplication**: The server tokenizes the sentence and checks for duplicates using string comparison (`strcmp()`).

Flow:

1. **Client Input**: The client sends a sentence like "Hello world world".
2. **Server Processing**: The server removes duplicates and sends back "Hello world".
3. **Client Output**: The client displays the deduplicated sentence.

Key Code Snippets:

- Sentence deduplication using `strtok()` for tokenizing the sentence.
- Sending back the modified sentence to the client using `send()`.

OUTPUT:

```
$ ./client
Enter a sentence: hello world hello world again
Processed sentence: hello world again
$
```

```
$ bash
cn1@selab-30:~/Desktop/220905106/LAB6/Q2$ ls
 client     IMG1              'IMG2 CLIENT.png'   IMG3.png   server.c
 client.c  'IMG1 CLIENT.png'   IMG2.png           server
cn1@selab-30:~/Desktop/220905106/LAB6/Q2$ gcc -o server server.c
cn1@selab-30:~/Desktop/220905106/LAB6/Q2$ gcc -o client client.c
cn1@selab-30:~/Desktop/220905106/LAB6/Q2$ ./server
Server is listening on port 65432...
Accepted connection from client
```

ADDITIONAL QUESTION

3. Time and Process ID Retrieval Using Fork and Multi-Process Handling

SERVER:

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

#define PORT 65432
#define BUFFER_SIZE 1024

void handle_client(int client_socket) {
    char buffer[BUFFER_SIZE];
    time_t now;
    struct tm *tm_info;
    char time_buffer[26];

    // Get current time and format it
    time(&now);
    tm_info = localtime(&now);
    strftime(time_buffer, 26, "%Y-%m-%d %H:%M:%S", tm_info);

    // Get process ID
    pid_t pid = getpid();

    // Prepare the message
    snprintf(buffer, BUFFER_SIZE, "Current time: %s\nProcess ID: %d\n", time_buffer, pid);

    // Send the message to the client
    send(client_socket, buffer, strlen(buffer), 0);

    // Close the client socket
    close(client_socket);
}

int main() {
    int server_socket, client_socket;
    struct sockaddr_in server_addr, client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    pid_t child_pid;

    // Create the server socket
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket < 0) {
```

```
58          perror("Bind failed");
59          close(server_socket);
60          exit(EXIT_FAILURE);
61      }
62
63      // Listen for incoming connections
64      if (listen(server_socket, 5) < 0) {
65          perror("Listen failed");
66          close(server_socket);
67          exit(EXIT_FAILURE);
68      }
69
70      printf("Server is listening on port %d...\n", PORT);
71
72      while (1) {
73          // Accept a new client connection
74          client_socket = accept(server_socket, (struct sockaddr *)&client_addr, &client_addr_len);
75          if (client_socket < 0) {
76              perror("Accept failed");
77              continue;
78          }
79
80          printf("Accepted connection from client\n");
81
82          // Fork a new process to handle the client
83          child_pid = fork();
84          if (child_pid < 0) {
85              perror("Fork failed");
86              close(client_socket);
87              continue;
88          }
89
90          if (child_pid == 0) {
91              // Child process
92              close(server_socket); // Close the server socket in the child process
93              handle_client(client_socket);
94              exit(0); // Exit the child process after handling the client
95          } else {
96              // Parent process
97              close(client_socket); // Close the client socket in the parent process
98          }
99      }
100
101     close(server_socket);
```

CLIENT:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>

#define PORT 65432
#define BUFFER_SIZE 1024

int main() {
    int client_socket;
    struct sockaddr_in server_addr;
    char buffer[BUFFER_SIZE];

    // Create the client socket
    client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket < 0) {
        perror("Socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Prepare the server address structure
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    server_addr.sin_port = htons(PORT);

    // Connect to the server
    if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("Connection failed");
        close(client_socket);
        exit(EXIT_FAILURE);
    }

    // Receive the message from the server
    ssize_t bytes_received = recv(client_socket, buffer, BUFFER_SIZE - 1, 0);
    if (bytes_received < 0) {
        perror("Receive failed");
    } else {
        buffer[bytes_received] = '\0'; // Null-terminate the string
        printf("Received from server:\n%s\n", buffer);
    }

    close(client_socket);
```

Client and Server with Forking

This program involves a server that forks a new process for each client. When a client connects, the server sends back the current time and the process ID of the server handling the request.

Key Features:

- **Forking for Concurrency**: The server forks a new process for each client using `fork()`. Each client request is handled by a new child process, ensuring non-blocking operations.
- **Time and PID Retrieval**: The server sends the current time and process ID back to the client. Time is retrieved using `localtime()` and formatted using `strftime()`.
- **Concurrency**: Unlike threads, this implementation uses `fork()`, where each client is handled by a new child process, isolating client interactions in separate processes.

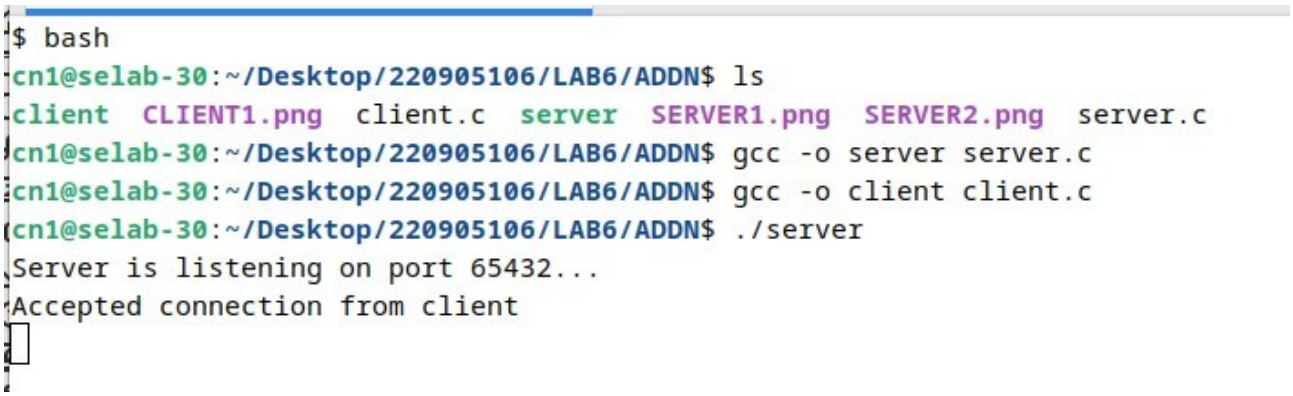
Flow:

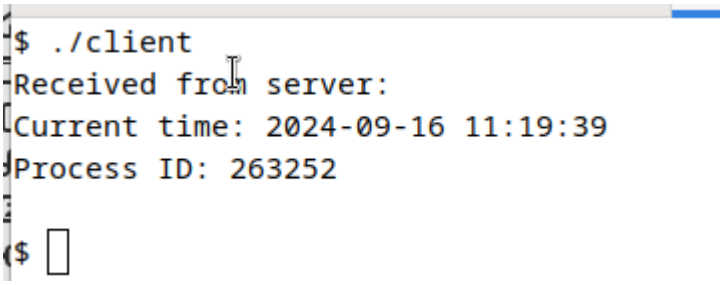
1. **Client Input**: The client connects to the server.
2. **Server Processing**: The server sends back the current time and the PID.
3. **Client Output**: The client displays the received information.

Key Code Snippets:

- Forking server using `fork()` to create a child process for each client.
- Time retrieval using `localtime()` and PID retrieval using `getpid()`.

OUTPUT:





Conclusion

All three implementations demonstrate different techniques for handling multiple clients in a socket programming environment:

- **Threading**: Used in arithmetic operations and sentence deduplication tasks to allow parallel client-server interaction.
- **Forking**: Applied in the time and PID retrieval task to separate client interactions into different processes.

Each solution highlights different ways to manage concurrency and client requests in a multi-client system. These programs provide robust handling of input, error management, and server-client communication in a socket-based environment.