Implementing a TCP/IP Client-Server Model Using Socket Programming

1. OBJECTIVE

This paper implements a reliable TCP/IP client-server communication system using sockets to exchange JSON messages, ensuring data integrity while addressing common connection challenges like address reuse and connection refusal.

2. PROBLEM STATEMENT

Modern networked applications frequently encounter interoperability failures when transmitting raw, unstructured data across TCP connections. This occurs because receiving systems cannot consistently interpret sender payloads without standardized serialization protocols, leading to data corruption, inconsistent state management, and costly debugging cycles. Specifically, the absence of explicit JSON serialization within TCP communication channelsdespite TCPs connection-oriented naturecreates a critical gap where structured data exchange remains non-deterministic. This experiment addresses the fundamental problem: How does the implementation of JSON serialization over TCP connections resolve interoperability failures while ensuring end-to-end data integrity?

3. CODE

3.1 SERVER FLOW

The server follows a sequential process:

1. Create a socket file descriptor
2. Bind the socket to a specified IP address and port
3. Listen for incoming client connections
4. Accept a connection and create a new socket for communication
5. Read data from the client and send a response
6. Close the connection

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

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
  int server_fd, new_socket, valread;
  struct sockaddr_in address;
  int addrlen = sizeof(address);
  char buffer[BUFFER_SIZE] = {0};
  const char* hello = "Hello from server";

  // 1. Create socket file descriptor
  if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    perror("socket failed");
    exit(EXIT_FAILURE);
  }

  // Optional: Setsockopt to reuse address and port immediately
  int opt = 1;
  if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt, sizeof(opt))) {
    perror("setsockopt failed");
    exit(EXIT_FAILURE);
  }

  // Prepare the sockaddr_in structure
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = INADDR_ANY; // Listen on all network interfaces
  address.sin_port = htons(PORT);

  // 2. Bind the socket to the port
  if (bind(server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
    perror("bind failed");
    exit(EXIT_FAILURE);
  }

  // 3. Listen for incoming connections
  if (listen(server_fd, 3) < 0) { // Max 3 pending connections
    perror("listen failed");
    exit(EXIT_FAILURE);
  }

```

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

    // 4. Accept a new connection
    if ((new_socket = accept(server_fd, (struct sockaddr*)&address,
        (socklen_t*)&addrlen)) < 0) {
        perror("accept failed");
        exit(EXIT_FAILURE);
    }

    printf("Client connected.\n");

    // 5. Read data from the client
    valread = read(new_socket, buffer, BUFFER_SIZE);
    printf("Client message: %s\n", buffer);

    // 6. Send a message to the client
    send(new_socket, hello, strlen(hello), 0);
    printf("Hello message sent to client.\n");

    // 7. Close the sockets
    close(new_socket);
    close(server_fd);

    return 0;
}
```

3.2 CLIENT FLOW

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

#define PORT 8080
#define BUFFER_SIZE 1024

int main() {
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char* hello = "Hello from client";
    char buffer[BUFFER_SIZE] = {0};

    // 1. Create socket file descriptor
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\nSocket creation error\n");
        return -1;
    }

    // Prepare the sockaddr_in structure for the server
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 address from text to binary form
    if (inet_pton(AF_INET, "10.1.78.47", &serv_addr.sin_addr) <= 0)
        {
      printf("\nInvalid address/ Address not supported\n");
      return -1;
    }

    // 2. Connect to the server
    if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(
        serv_addr)) < 0) {
      perror("connect failed");
      return -1;
    }

    // 3. Send a message to the server
    send(sock, hello, strlen(hello), 0);
    printf("Hello message sent to server.\n");

    // 4. Read data from the server
    valread = read(sock, buffer, BUFFER_SIZE);
    printf("Server message: %s\n", buffer);

    // 5. Close the socket
    close(sock);

    return 0;
}
```

4. CHALLENGES AND SOLUTIONS

Several challenges were encountered during the implementation, along with their respective solutions:

1. **Address Already in Use (EADDRINUSE)**: This error occurs when a server is restarted too quickly, and the port remains occupied. The solution involves using the `grep` and `kill` commands to terminate the process occupying the port.

2. **Server Not Accepting New Connections**: The basic server handles only one client at a time, blocking new connections until the current one is closed. Solutions include increasing the connection queue limit to 15 or implementing concurrency models using multithreading, multiprocessing, or non-blocking I/O with `select()` or `poll()`.

3. **Client Connection Refused (ECONNREFUSED)**: This issue arises due to:

   - The server not running.

- Incorrect IP address or port in the client program.
- Firewall blocking the connection.

5. CONCLUSION

The implemented client-server system demonstrates TCPs connection-oriented nature through a three-way handshake for connection establishment and a four-way handshake for termination. The server binds to a specific IP address (INADDR_ANY) and port (8080), enabling client connections. The sequential processing model handles one client at a time, suitable for basic applications but limited for concurrent scenarios. Socket file descriptors are utilized effectively, with the server creating a new socket for each client. Future improvements could include implementing concurrency to handle multiple clients simultaneously and incorporating JSON encoding for structured data exchange.

lmodern