**Introduction**

A Weather Monitoring System using a client-server architecture in Linux and programmed in C is a sophisticated application designed to gather, process, and display weather-related data. This system leverages the robust features of the Linux operating system, such as its networking capabilities, multitasking, and stability, to efficiently handle the communication between the client and server. In this setup, the server is responsible for collecting weather data from various sensors or databases, processing this data, and then transmitting it to the client upon request. The client, on the other hand, sends requests to the server and displays the received weather information to the user. This architecture is not only efficient in terms of resource management but also provides a scalable and flexible solution for real-time weather monitoring across different locations.

**System Requirements**

**Hardware Requirements:**

* **Processor:** Dual-core CPU or higher
* **Memory:** 2 GB RAM minimum
* **Storage:** 500 MB free disk space
* **Network:** Ethernet or Wi-Fi connection for client-server communication
* **Sensors:** Optional external sensors for collecting weather data (e.g., temperature, humidity, pressure)

**Software Requirements:**

* **Operating System:** Linux (any distribution, e.g., Ubuntu, CentOS)
* **Programming Language:** C
* **Compiler:** GCC (GNU Compiler Collection)
* **Libraries:** POSIX threads (for multithreading), sockets API (for networking), optional sensor-specific libraries
* **Version Control:** Git (optional, for source code management)
* **Development Tools:** Text editor (e.g., Vim, VSCode), debugger (e.g., GDB)

**Functionality**

* **Data Collection:** The server collects weather data from sensors or a database.
* **Data Processing:** The server processes raw weather data to extract meaningful information such as temperature, humidity, and pressure.
* **Client Requests:** Clients can request specific weather data from the server.
* **Data Transmission:** The server sends the requested data back to the client over a network.
* **Real-Time Monitoring:** The system supports real-time monitoring, allowing clients to receive up-to-date weather information.
* **Logging:** The system logs weather data and client requests for future reference and analysis.

**Modules**

1. **Server Module:**
   * **Sensor Interface:** Interacts with sensors to gather raw weather data.
   * **Data Processing:** Processes raw data to make it usable.
   * **Client Communication:** Manages client requests and sends responses.
   * **Data Logging:** Logs data for historical analysis.
2. **Client Module:**
   * **User Interface:** Displays weather data to the user.
   * **Server Communication:** Sends requests to the server and receives data.
   * **Data Display:** Presents the data in a user-friendly format (e.g., graphs, text).
3. **Networking Module:**
   * **Socket Programming:** Handles the creation and management of network sockets for client-server communication.
   * **Data Serialization:** Manages the formatting of data before transmission.
4. **Thread Management Module:**
   * **Multithreading:** Handles concurrent client requests using POSIX threads.
   * **Synchronization:** Manages data consistency across threads.

**Implementation**

1. **Server Setup:**
   * **Initialize the server:** Set up the server socket and bind it to a specific port.
   * **Sensor Integration:** If using physical sensors, integrate them into the server using the appropriate libraries.
   * **Thread Management:** Create threads to handle multiple client requests simultaneously.
   * **Data Handling:** Implement data processing functions to convert raw sensor data into meaningful weather information.
   * **Client Interaction:** Write functions to handle incoming client requests, process them, and send back the requested data.
2. **Client Setup:**
   * **Initialize the Client:** Create a client socket and connect it to the server.
   * **Request Data:** Implement functions to send data requests to the server.
   * **Receive Data:** Write functions to receive data from the server and handle it accordingly.
   * **Display Data:** Develop a simple interface to display the weather data received from the server.
3. **Networking:**
   * **Socket Programming:** Implement socket programming in C using the sockets API to enable communication between the client and server.
   * **Error Handling:** Implement robust error handling to manage network failures, data transmission errors, and other potential issues.
4. **Thread Management:**
   * **Create Threads:** Use POSIX threads to manage multiple client requests concurrently.
   * **Synchronize Threads:** Ensure data consistency and manage resource sharing between threads using mutexes or other synchronization techniques.

**Testing**

1. **Unit Testing:**
   * Test individual functions like data processing, socket communication, and thread management.
   * Use test cases to ensure each module performs as expected in isolation.
2. **Integration Testing:**
   * Test the interaction between different modules (e.g., client-server communication, data handling).
   * Ensure that the data flow from sensors to the client is seamless and accurate.
3. **System Testing:**
   * Deploy the system in a real or simulated environment to test the overall functionality.
   * Test the system under different conditions (e.g., high traffic, network failure) to ensure stability.
4. **Performance Testing:**
   * Measure the system’s response time to client requests.
   * Test the server’s ability to handle multiple concurrent connections.
5. **User Acceptance Testing (UAT):**
   * Present the system to end-users and gather feedback.
   * Make adjustments based on user feedback to improve usability and performance.
6. **Debugging:**
   * Use debugging tools like GDB to identify and fix any bugs or issues encountered during testing

Source code

**Server Code (server.c)**

The server will listen for incoming connections, generate weather data, and send it to connected clients.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#include <pthread.h>

#include <time.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

typedef struct {

int temperature;

int humidity;

int wind\_speed;

} WeatherData;

void generate\_weather\_data(WeatherData\* data) {

// Simulate weather data generation

srand(time(NULL));

data->temperature = rand() % 40; // Random temperature between 0 and 39

data->humidity = rand() % 100; // Random humidity between 0 and 99

data->wind\_speed = rand() % 150; // Random wind speed between 0 and 149

}

void\* handle\_client(void\* arg) {

int client\_sock = \*(int\*)arg;

char buffer[BUFFER\_SIZE];

int read\_size;

WeatherData data;

// Generate weather data

generate\_weather\_data(&data);

snprintf(buffer, BUFFER\_SIZE, "Temperature: %d C, Humidity: %d %%, Wind Speed: %d km/h",

data.temperature, data.humidity, data.wind\_speed);

// Send weather data to the client

if (send(client\_sock, buffer, strlen(buffer), 0) < 0) {

perror("Send failed");

close(client\_sock);

free(arg);

return NULL;

}

// Receive confirmation from the client

if ((read\_size = recv(client\_sock, buffer, BUFFER\_SIZE, 0)) > 0) {

buffer[read\_size] = '\0';

printf("Client: %s\n", buffer);

}

if (read\_size == 0) {

printf("Client disconnected\n");

} else if (read\_size == -1) {

perror("recv failed");

}

close(client\_sock);

free(arg);

return NULL;

}

int main() {

int server\_sock, client\_sock, \*new\_sock;

struct sockaddr\_in server, client;

socklen\_t client\_len = sizeof(client);

pthread\_t thread\_id;

// Create socket

server\_sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_sock == -1) {

perror("Could not create socket");

exit(EXIT\_FAILURE);

}

printf("Socket created\n");

// Prepare the sockaddr\_in structure

server.sin\_family = AF\_INET;

server.sin\_addr.s\_addr = INADDR\_ANY; // Bind to any IP address

server.sin\_port = htons(PORT);

// Bind

if (bind(server\_sock, (struct sockaddr\*)&server, sizeof(server)) < 0) {

perror("Bind failed");

close(server\_sock);

exit(EXIT\_FAILURE);

}

printf("Bind done\n");

// Listen

if (listen(server\_sock, 3) < 0) {

perror("Listen failed");

close(server\_sock);

exit(EXIT\_FAILURE);

}

printf("Waiting for incoming connections...\n");

// Accept incoming connections

while ((client\_sock = accept(server\_sock, (struct sockaddr\*)&client, &client\_len))) {

printf("Connection accepted\n");

new\_sock = malloc(1);

\*new\_sock = client\_sock;

if (pthread\_create(&thread\_id, NULL, handle\_client, (void\*)new\_sock) < 0) {

perror("Could not create thread");

free(new\_sock);

close(client\_sock);

} else {

printf("Handler assigned\n");

}

}

if (client\_sock < 0) {

perror("Accept failed");

close(server\_sock);

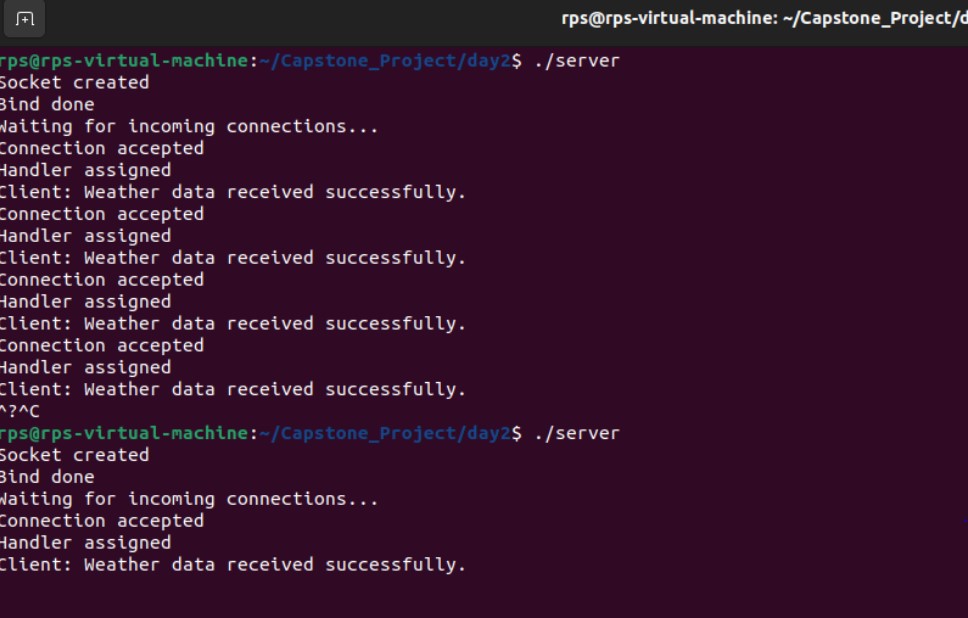
exit(EXIT\_FAILURE);

}

close(server\_sock);

return 0;

}



**Client Code (client.c)**

The client connects to the server, requests weather data, and displays the received information.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <arpa/inet.h>

#define PORT 8080

#define BUFFER\_SIZE 1024

int main() {

int sock;

struct sockaddr\_in server;

char message[BUFFER\_SIZE], server\_reply[BUFFER\_SIZE];

// Create socket

sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (sock == -1) {

perror("Could not create socket");

exit(EXIT\_FAILURE);

}

printf("Socket created\n");

server.sin\_family = AF\_INET;

server.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); // Connect to localhost

server.sin\_port = htons(PORT);

// Connect to remote server

if (connect(sock, (struct sockaddr\*)&server, sizeof(server)) < 0) {

perror("Connect failed");

close(sock);

exit(EXIT\_FAILURE);

}

printf("Connected to server\n");

// Receive weather data from the server

int recv\_len = recv(sock, server\_reply, BUFFER\_SIZE, 0);

if (recv\_len < 0) {

perror("recv failed");

close(sock);

exit(EXIT\_FAILURE);

}

server\_reply[recv\_len] = '\0'; // Null-terminate the received string

printf("Weather Data: %s\n", server\_reply);

// Send confirmation to server

snprintf(message, BUFFER\_SIZE, "Weather data received successfully.");

if (send(sock, message, strlen(message), 0) < 0) {

perror("Send failed");

}

close(sock);

return 0;

}

**Compilation and Execution**

1. **Compile the Server and Client Code:**

bash

Copy code

gcc -o server server.c -lpthread

gcc -o client client.c

1. **Run the Server:**

bash

Copy code

./server



**Conclusion**

The implementation of a Weather Monitoring System using a client-server architecture in Linux with C programming offers a powerful and efficient solution for real-time weather data management. By utilizing Linux's inherent strengths in networking and concurrency, the system ensures reliable and quick communication between the client and server, making it suitable for continuous and large-scale weather monitoring tasks. The system's modularity and scalability also allow for easy upgrades and integration of additional features, such as more sensors or advanced data analytics, making it a robust tool for meteorological applications.