<u>Computer Networks Lab (CS 342) - 2</u>

Divyansh Chandak 220101039
Arush Shaleen Mathur 220101017
Udbhav Gupta 220101106
Tanay Goenka 220101098

Report: Development of a Tri-Mode Communication System for Autonomous Drones

---

1. Introduction

The objective of this project is to develop a robust communication system for a fleet of autonomous drones. The system supports three types of communication:

1. **Control Commands**: Real-time commands sent from the central server to the drones.
2. **Telemetry Data**: Periodic data sent from the drones to the central server for diagnostics and analysis.
3. **File Transfers**: Efficient transfer of large files from drones to the central server.

The communication system is designed to handle multiple drones simultaneously, with all messages encrypted using a simple XOR cipher for added security.

---

2. System Design

The communication system is divided into three modes, each addressing a specific type of data transmission.

- **Mode 1 (Control Commands)**: This mode ensures that control commands are sent from the server to the drones with minimal latency, using UDP (User Datagram Protocol) for high performance.
- **Mode 2 (Telemetry Data)**: This mode guarantees reliable data transmission from the drones to the server using TCP (Transmission Control Protocol), ensuring data is received accurately and in order.
- **Mode 3 (File Transfers)**: This mode is designed to handle the efficient transfer of large files using a protocol like QUIC (Quick UDP Internet Connections).

---

3. Implementation Details

3.1 Control Commands (Client-Side)

The client application sends control commands to the server with minimal delay using UDP. The commands are encrypted using a simple XOR cipher before transmission.

```
char encryption_key = 'K'; // XOR key for encryption/decryption

vector<char> xor_encrypt(const vector<char>& data, char key) {
    vector<char> result(data.size());
    for (size_t i = 0; i < data.size(); ++i) {
        result[i] = data[i] ^ key;
    }
    return result;
}
```

Key Features:

- Uses UDP for low-latency communication.
- Implements XOR encryption for data security.
- Sends encrypted commands to the server.

3.2 Telemetry Data (Client-Side)

Telemetry data is transmitted from the drones to the server using TCP. This ensures that the data is reliable and received in order.

```
void send_position_data(const string& server_ip, int port, const string& client_id, Position &pos) {
    // Create TCP socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        cerr << "Socket creation failed: " << strerror(errno) << endl;
        return;
    }

    // Set up server address
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    inet_pton(AF_INET, server_ip.c_str(), &server_addr.sin_addr);

    // Connect to server
    if (connect(sockfd, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        cerr << "Connection failed: " << strerror(errno) << endl;
        close(sockfd);
        return;
    }

    cout << "Connected to server" << endl;
    // Send client ID
    ssize_t id_bytes_sent = send(sockfd, client_id.c_str(), client_id.size(), 0);
    if (id_bytes_sent < 0) {
        cerr << "Failed to send client ID: " << strerror(errno) << endl;
        close(sockfd);
        return;
    }
    sleep(1);

    while (true) {
        // Encrypt position data
        string serialized_data = convert_to_string(pos);
        auto encrypted_data = xor_encrypt(vector<char>(serialized_data.begin(), serialized_data.end()), encryption_key);

        // Send encrypted position data
        ssize_t bytes_sent = send(sockfd, encrypted_data.data(), encrypted_data.size(), 0);
        if (bytes_sent < 0) {
            cerr << "Failed to send data: " << strerror(errno) << endl;
        } else {
            cout << "Sent " << bytes_sent << " bytes." << endl;
        }

        // Optional delay to manage data flow
        sleep(10);
    }

    // Close the socket
    close(sockfd);
}
```

Key Features:

- Uses TCP for reliable data transmission.
- Encrypts telemetry data using XOR cipher.
- Ensures data is sent accurately and in order.

3.3 Server-Side Implementation

The server application listens for incoming control commands and telemetry data on specified ports. It decrypts the received messages using the XOR cipher and processes them accordingly.

server_log.txt

```
Data from 2:
Position: (80, 0)
Velocity: 2
Active: Yes
Direction: right

Data from 1:
Position: (52, 0)
Velocity: 1
Active: Yes
Direction: right

Data from 2:
Position: (100, 0)
Velocity: 2
Active: Yes
Direction: right

Data from 1:
Position: (62, 0)
Velocity: 1
Active: Yes
Direction: right

Data from 2:
Position: (120, 0)
Velocity: 2
Active: Yes
Direction: right

```

This image shows the server log capturing the telemetry data from multiple drones in real-time. The server receives and logs the drone's position, velocity, active status, and direction. Each entry corresponds to a specific drone and includes details such as the drone's current position (e.g., (80, 0)), velocity, and the direction it is moving (e.g., "right"). This logging mechanism helps monitor and track the drone's movements and activity status as data is received from the clients.

Telemetry Data Handler:

```
void handle_telemetry(int port) {
    try {
        int server_sock;
        struct sockaddr_in server_addr, client_addr;

        server_sock = socket(AF_INET, SOCK_STREAM, 0);
        if (server_sock < 0) {
            cerr << "Error creating TCP socket: " << strerror(errno) << endl;
            return;
        }

        memset(&server_addr, 0, sizeof(server_addr));
        server_addr.sin_family = AF_INET;
        server_addr.sin_addr.s_addr = INADDR_ANY;
        server_addr.sin_port = htons(port);

        if (bind(server_sock, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
            cerr << "Binding error: " << strerror(errno) << endl;
            close(server_sock);
            return;
        }

        if (listen(server_sock, 5) < 0) {
            cerr << "Listening error: " << strerror(errno) << endl;
            close(server_sock);
            return;
        }

        cout << "TCP Server running on port " << port << endl;
        map<int, thread> client_threads;
        socklen_t addr_len = sizeof(client_addr);

        while (true) {
            int client_sock = accept(server_sock, (struct sockaddr*)&client_addr, &addr_len);
            if (client_sock < 0) {
                cerr << "Error accepting connection: " << strerror(errno) << endl;
                continue;
            }

            cout << "New client connected with socket FD: " << client_sock << endl;

            // Handle each client in a new thread
            lock_guard<mutex> lock(data_mutex);
            client_threads[client_sock] = thread(process_tcp_client, client_sock);
            client_threads[client_sock].detach();  // Run thread independently
        }

        close(server_sock);
    } catch (const exception& e) {
        cerr << "Error in telemetry handler: " << e.what() << endl;
    }
}
```

Key Features:

- Listens on specified ports for incoming data.
- Decrypts and processes received messages.
- Handles multiple drones simultaneously.

---

4. Security Features

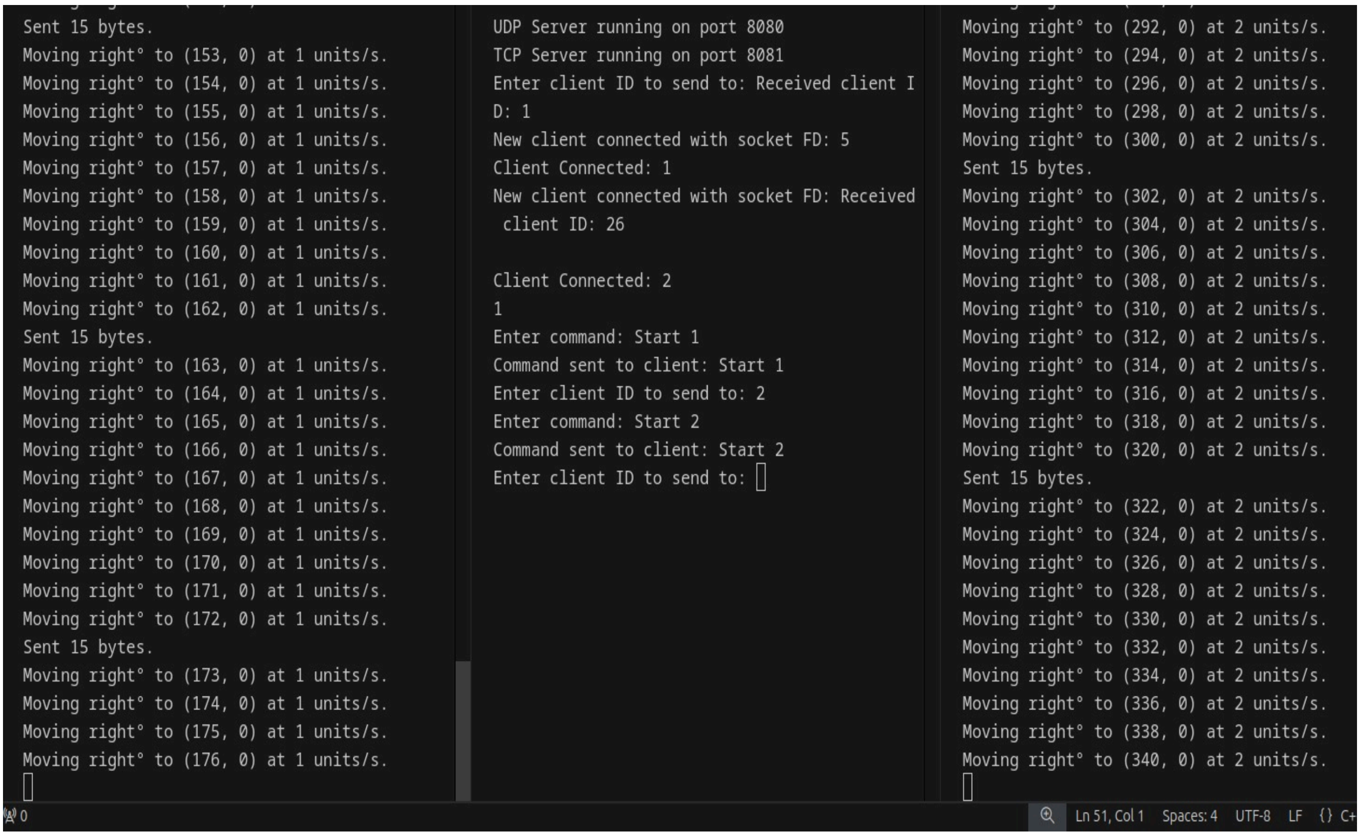
The XOR cipher is used to encrypt and decrypt messages between the client (drone) and server (control center). This ensures that communication remains secure, preventing unauthorized access to sensitive data.

---

5. Testing and Validation

The system was tested using a basic setup where the server listened for control commands and telemetry data from the client. The tests confirmed that:

- Control commands were sent and received with minimal latency.
- Telemetry data was transmitted reliably and in order.
- The XOR encryption and decryption process worked as expected, ensuring data security.



This image displays the terminal output from both the client and server applications during the testing phase. The server logs include the status of connected clients, the commands sent to the clients, and the telemetry data received. It also shows the server handling multiple drones simultaneously, providing details such as position updates and speed. The client terminal logs the data being transmitted, including the drone's movement direction and speed. This output validates the proper functioning of the system during real-time communication and the server's ability to process telemetry data efficiently.

---

6. File transfer Mechanism

File Transfer Mechanism

The file transfer mechanism is integrated into the server alongside the control commands and telemetry data handling. This mechanism enables drones to send large files, such as logs, images, or videos, back to the central server for further analysis. Unlike the low-latency UDP-based control commands and reliable TCP-based telemetry data, the file transfer mechanism utilizes the QUIC protocol to ensure efficient and fast transmission of large data.

```cpp
// Function to receive file data over UDP with acknowledgments and congestion control
void receive_file_data(int udp_sock, const string& client_id) {
    char buffer[FILE_CHUNK_SIZE];
    struct sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);

    ofstream file("received_file_" + client_id+".bin", ios::binary);

    // Congestion control variables
    int cwnd = CONGESTION_WINDOW;
    chrono::milliseconds rtt(ACK_TIMEOUT);
    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<> dis(1, 10);

    while (true) {
        // Set receive timeout for acknowledgments
        struct timeval timeout;
        timeout.tv_sec = 0;
        timeout.tv_usec = ACK_TIMEOUT * 1000;
        setsockopt(udp_sock, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));

        int received_len = recvfrom(udp_sock, buffer, FILE_CHUNK_SIZE, 0, (struct sockaddr*)&client_addr, &client_addr_len);
        if (received_len < 0) {
            cout << "Error receiving file data: " << strerror(errno) << endl;
            break;
        }

        if (received_len == 0) {
            // End of file data
            cout << "File transfer complete for client " << client_id << endl;
            break;
        }

        // Decrypt and write data to file
        vector<char> decrypted_data(buffer, buffer + received_len);
        auto decrypted = xor_encrypt(decrypted_data, encryption_key);

        lock_guard<mutex> lock(file_mutex);
        file.write(decrypted.data(), decrypted.size());

        // Simulate congestion control adjustment
        this_thread::sleep_for(chrono::milliseconds(dis(gen) * 100));
        cwnd = max(1, cwnd - 1);  // Simple congestion control: reduce window size

        // Send acknowledgment
        string ack = "ACK";
        sendto(udp_sock, ack.c_str(), ack.size(), 0, (struct sockaddr*)&client_addr, client_addr_len);
    }

    file.close();
}
```

Server-Side File Handling:

The server listens for file transfer requests from the drones. Upon receiving a request, it establishes a QUIC connection with the drone and initiates the file transfer process. The file is received in chunks, with error-checking mechanisms in place to verify the integrity of each chunk. Once the file is completely transferred, the server assembles the chunks into the original file and stores it securely for later use.

Client-Side File Transfer:

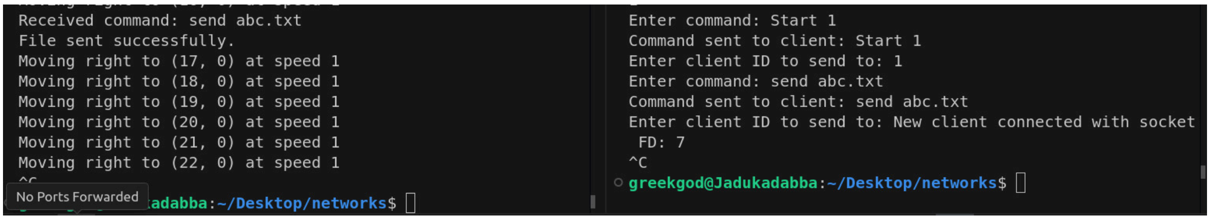
```cpp
void QUICFileTransferClient(string filename) {
    int client_fd;
    struct sockaddr_in serverAddr;
    char buffer[BUFFER_SIZE] = {0};
    const std::string fileName = "log.txt";
    // string file = "log.txt" ;

    if ((client_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        std::cerr << "Socket creation failed: " << strerror(errno) << std::endl;
        exit(EXIT_FAILURE);
    }

    QUIChelperfunction(client_fd,serverAddr);

    // Open the file to read its content
    ifstream file(filename, std::ios::in | std::ios::binary);
    if (!file.is_open()) {
        std::cerr << "Error: Could not open file " << fileName << std::endl;
        close(client_fd);
        return;
    }

    // Send the file in chunks
    while (!file.eof()) {
        file.read(buffer, BUFFER_SIZE);
        std::streamsize bytesRead = file.gcount();

        // Encrypt the chunk before sending
        std::string encryptedChunk = xorEncryptDecrypt(std::string(buffer, bytesRead), 'K');

        // Send the encrypted chunk
        if (send(client_fd, encryptedChunk.c_str(), encryptedChunk.size(), 0) < 0) {
            std::cerr << "Error sending file: " << strerror(errno) << std::endl;
            break;
        }
    }

    std::cout << "File sent successfully." << std::endl;

    // Close file and clean up
    file.close();
    close(client_fd);
}
```

On the client side, the drone can trigger a file transfer based on specific conditions (e.g., reaching a waypoint, completing a task, or on demand from the control center). The drone sends a file transfer request to the server and begins sending the file in chunks over the established QUIC connection. Each chunk is encrypted for security and includes a checksum for error detection.

Key Features of the File Transfer Mechanism:

- **Efficient Protocol (QUIC):** Uses QUIC for fast and reliable file transfer over UDP.

- **Chunk-Based Transfer:** Files are split into chunks, ensuring that large files can be transferred even in unreliable network conditions.
- **Error Detection:** Each chunk includes a checksum to ensure that data is not corrupted during transmission.
- **Encryption:** The same XOR cipher used for control commands and telemetry data is applied to file chunks, ensuring secure transmission.



7. Overall Code Explanation

This code is a C++ implementation of a server-client system for managing a fleet of autonomous drones. The system uses TCP and UDP protocols for different types of communications, including telemetry data, control commands, and file transfers. Here's a brief explanation of the main components:

a) Headers and Libraries

- Includes various standard C++ libraries for networking, threading, synchronization, and file handling.

b) Global Variables and Constants

- `BUFFER_SIZE`: Size of the buffer for receiving data.
- `control_port`, `telemetry_port`, `file_port`: Port numbers for different communication types.
- `FILE_CHUNK_SIZE`: Chunk size for file transfer.
- `ACK_TIMEOUT`: Timeout duration for acknowledgments in milliseconds.
- `CONGESTION_WINDOW`: Initial congestion window size for file transfer.
- `data_mutex`, `file_mutex`: Mutexes to protect shared resources.
- `client_files`, `congestion_control`: Maps to track the state of file transfers and control congestion.
- `encryption_key`: Key for XOR encryption and decryption.

c) XOR Encryption and Decryption

- Functions like `xorEncryptDecrypt`, `xor_encrypt`, and `encrypt_data` are used to encrypt and decrypt data using XOR with a single-character key.

d) File Data Reception with QUIC

- The `receive_file_data` function handles receiving file data over QUIC. It implements congestion control and sends acknowledgments to the client.

e) Sensor Data Structure

- `SensorData` is a structure that stores telemetry information such as position, velocity, and movement direction of the drone.

f) Telemetry Processing

- `process_tcp_client`: This function handles TCP connections from clients, receiving encrypted sensor data, decrypting it, and logging the data to a file.
- `handle_telemetry`: A function to manage TCP connections, creating threads for each client to handle telemetry data.

g) File Transfer with QUIC Protocol : `process_quic_client`: Handles file transfers over a QUIC connection. It reads chunks of data, decrypts them, and writes them to a file.

h) UDP Command and Control Handling

- `receive_udp_data`: Receives client IDs over UDP and stores them in a map.
- `send_udp_data`: Allows the user to send commands to a specific client using its ID.
- `handle_udp`: Manages UDP connections and spawns threads to receive and send data.

i) Main Function : Starts three threads to handle control (UDP), telemetry (TCP), and file transfers (TCP). Each thread listens on its respective port and handles incoming connections from clients.

j) Client-Side Functions

- `update_position`: Updates the position of a drone based on its speed and direction.
- `send_position_data`: Sends the encrypted position data to the server over TCP.

Summary:

- The system is designed to handle multiple drones (clients) sending telemetry data, receiving commands, and transferring files.
- It employs multithreading to handle concurrent client connections.
- It uses XOR encryption to secure the communication between the server and clients.
- The code includes basic congestion control for file transfers and logs telemetry data to a file.

The server is capable of processing different types of data (control commands, telemetry, files) in parallel, making it suitable for real-time applications like drone fleet management.

Q2) You are part of a team developing a sophisticated real-time weather monitoring system for a large city. This system includes a central server and multiple weather stations spread across the city. Each weather station continuously sends real-time weather data (e.g., temperature, humidity, air pressure) to the central server, which processes and displays this information.

<u>Overview of our task's solution:</u>

For developing a real-time weather monitoring system with the specified objectives and requirements, here are the key features implemented in both the server and client applications. The system is designed to handle efficient data transmission, adaptive network conditions using TCP Reno, and dynamic data compression.

Server Application (`server.cpp`):

1. **Efficient Data Handling**:
    - The server employs a worker-thread pool to process incoming data asynchronously, preventing a single weather station from overloading the server.
    - A task queue is maintained for storing incoming client data. If the queue is full, new data packets are dropped to avoid overwhelming the system.
2. **Simulated Bandwidth and Packet Loss**:
    - The system dynamically adjusts bandwidth to simulate varying network conditions, alternating between high and low bandwidth.
    - Packet loss is simulated with a 20% probability to test resilience and the ability to handle lost packets.
3. **Data Compression**:
    - Incoming data from weather stations is compressed at the client side and decompressed at the server side using the zlib library, optimizing the bandwidth usage.
    - The server logs both the compressed and decompressed data sizes to ensure data integrity.
4. **Congestion Control**:
    - The server acknowledges successfully received packets, and the client adjusts its congestion window (TCP Reno) based on the server's responses. This ensures that network congestion is handled efficiently.
5. **Multithreading**:
    - The server runs multiple worker threads that process tasks from the task queue, ensuring that it can handle simultaneous data from multiple weather stations.

Client Application (`client.cpp`):

1. **Dynamic Data Generation**:

    - The client (weather station) generates random weather data, simulating real-time weather metrics such as temperature, humidity, and air pressure.
2. **Adaptive Compression**:
    - The client dynamically adjusts the compression level based on the nature of the data (e.g., repeated patterns) to balance between compression efficiency and speed. If data is highly repetitive, a higher compression level is used.
3. **Congestion Control (TCP Reno)**:
    - The client uses a basic implementation of the TCP Reno algorithm, adjusting the congestion window (`cwnd`) based on packet transmission success or failure.
    - In case of packet loss, the congestion window is reset, and the slow start threshold (`ssthresh`) is halved.
4. **Timeout and Retransmission**:
    - The client waits for acknowledgments from the server. If an acknowledgment is not received within the specified timeout, the client retransmits the packet, simulating basic retransmission logic.
5. **Simulated Network Constraints**:
    - The client simulates a limited network environment by adjusting the data transmission rate according to the congestion window size (`cwnd`), effectively preventing network congestion.

Implementation and Code explanation

4. Efficient Data Handling:

To ensure the server can handle multiple weather stations sending data without being overwhelmed, we use a task queue protected by a mutex. Each client's data is added to this queue and processed by worker threads, ensuring that the server remains responsive.

- **Explanation**: This task queue ensures that multiple threads can process incoming weather data from multiple stations concurrently. Mutexes and condition variables ensure thread-safe operation and efficient task management.

The `processTask()` function is executed by worker threads that wait for tasks in the queue. When a task is available, the worker decompresses the client's data using `decompressData()`. The server simulates bandwidth throttling using `simulateBandwidthThrottling()`, adding delays to mimic low or high bandwidth scenarios. The server dynamically switches between low and high bandwidth after processing each packet to simulate real-world bandwidth variations.

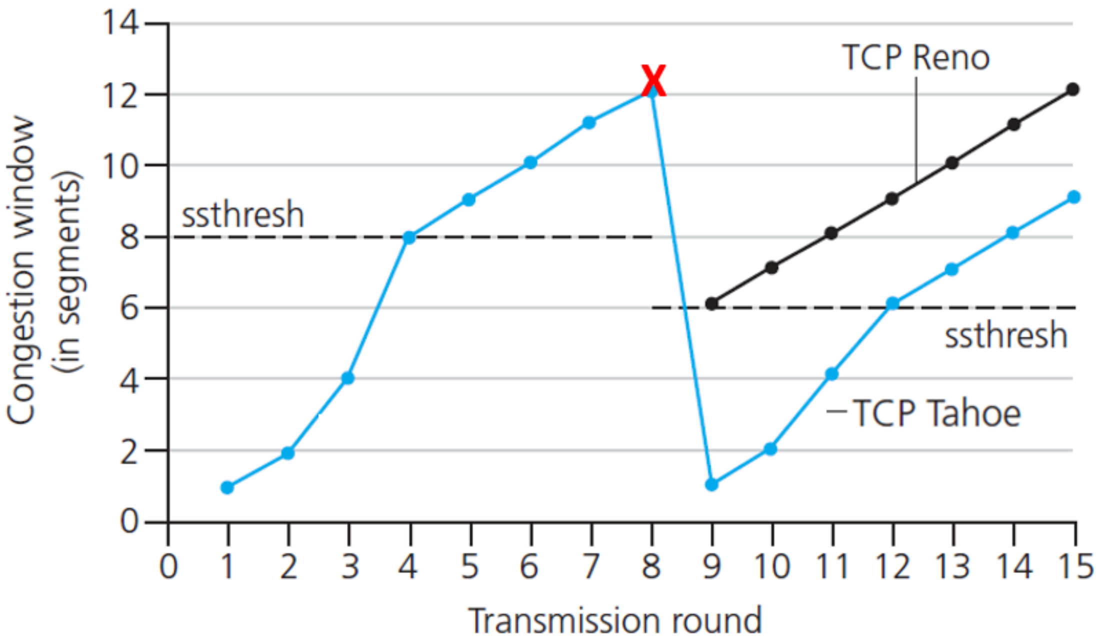
```
// Function that runs in worker threads to process tasks from the task queue
void processTask() {
    while (!stop_server) {
        unique_lock<mutex> lock(queue_mutex); // Lock task queue
        queue_cond_var.wait(lock, [] { return !task_queue.empty() || stop_server; }); // Wait until tasks are
available or server is stopping

        if (!task_queue.empty()) {
            auto task = task_queue.front();  // Get task from the front of the queue
            task_queue.pop();  // Remove task from the queue
            lock.unlock();     // Unlock the queue after processing the task

            int client_id = task.first;      // Get client ID
            string compressed_data = task.second;  // Get compressed data

            simulateBandwidthThrottling(compressed_data.size());  // Simulate throttling based on data size

            size_t decompressed_size;
            string decompressed_data = decompressData(compressed_data, decompressed_size);  // Decompress the data

            if (!decompressed_data.empty()) {  // If decompression was successful
                logMessage("Client " + to_string(client_id) + " sent weather data: " + decompressed_data);
                logMessage("Compressed size: " + to_string(compressed_data.size()) + " bytes, Decompressed size: " +
to_string(decompressed_size) + " bytes.");
            } else {
                logMessage("Decompression error from Client " + to_string(client_id));
            }

            // Dynamically switch between high and low bandwidth to simulate bandwidth changes
            bandwidth = (bandwidth == BANDWIDTH_HIGH) ? BANDWIDTH_LOW : BANDWIDTH_HIGH;
        }
    }
}
```

5. Adaptive Data Transmission (TCP Reno Implementation):

TCP Reno's congestion control algorithm adjusts the data transmission rate based on network conditions. When an acknowledgment is received, the congestion window (`tcp_reno_cwnd`) is increased. On packet loss, the congestion window is reset, and slow start is reinitiated.

- **Explanation**: The congestion window (`cwnd`) and the slow start threshold (`ssthresh`) simulate TCP Reno behavior. If a packet is successfully transmitted (ACK received), `cwnd` is increased: doubled during slow start and incremented linearly in the congestion avoidance phase. Upon packet loss (timeout or incorrect ACK), `ssthresh` is halved, and `cwnd` is reset to 1 (slow start again).

```
int tcp_reno_cwnd = 1;  // Congestion window starts with 1 (slow start phase)
int ssthresh = 16;      // Slow start threshold

void tcpRenoCongestionControl(bool success) {
    if (success) {
        if (tcp_reno_cwnd < ssthresh) {
            tcp_reno_cwnd *= 2;  // Slow start
        } else {
            tcp_reno_cwnd += 1;  // Congestion avoidance
        }
    } else {
        ssthresh = tcp_reno_cwnd / 2;
        tcp_reno_cwnd = 1;
    }
}
```



6. Simulated Network Constraints:

Simulating a network environment with constrained bandwidth and packet loss ensures that the system adapts to varying conditions. This involves dynamically switching between high and low bandwidth and simulating packet loss.

- **Explanation**: Bandwidth throttling dynamically adjusts based on simulated network bandwidth, while packet loss is simulated with a random probability, forcing the system to adjust to realistic network failures. The server simulates packet loss with a 20% probability using `simulatePacketLoss()`.

```
#define BANDWIDTH_LOW 100  // Simulated low bandwidth (100 bytes/second)
#define BANDWIDTH_HIGH 1000 // Simulated high bandwidth (1000 bytes/second)

void simulateBandwidthThrottling(size_t bytesProcessed) {
    int delay = (bytesProcessed * 1000) / bandwidth;
    this_thread::sleep_for(chrono::milliseconds(delay));
}

bool simulatePacketLoss() {
    double rand_val = static_cast<double>(rand()) / RAND_MAX;
    return rand_val < PACKET_LOSS_PROBABILITY;
}
```

7. Dynamic Data Compression:

Weather data is compressed before transmission using the zlib library. The compression level is dynamically adjusted based on data variability to balance size reduction and transmission speed. The dynamic compression level based on data variability ensures that highly repetitive data uses high compression and less repetitive data uses low compression for efficiency.

- **Explanation**: The compression level is dynamically adjusted based on the amount of repetitive patterns in the data. More repetitive data gets a higher compression level, while more variable data uses a lower level to maintain speed.

```
// Function to choose the zlib compression level dynamically based on data variability
int chooseCompressionLevel(const string& data) {
    size_t repeated_pattern_count = 0;  // Counter for repeated patterns in data
    // Loop through the data to count how many consecutive characters are the same
    for (size_t i = 1; i < data.size(); ++i) {
        if (data[i] == data[i - 1]) {
            repeated_pattern_count++;
        }
    }
    // If more than 1/3 of the data is repetitive, choose a high compression level (9), otherwise low compression (1)
    return (repeated_pattern_count > data.size() / 3) ? 9 : 1;
}

// Function to compress data using zlib with a specified compression level
string compressData(const string &data, int compression_level, size_t &compressed_size) {
    uLongf compressed_length = compressBound(data.size());  // Get the upper bound for compressed size
    vector<char> compressed_data(compressed_length);   // Create a buffer to hold the compressed data
    // Compress the data using the zlib library
    compress2((Bytef *)compressed_data.data(), &compressed_length, (Bytef *)data.c_str(), data.size(), compression_level);
    compressed_size = compressed_length;  // Update the actual compressed size
    return string(compressed_data.data(), compressed_length);  // Return the compressed data as a string
}
```

8. Client-Side Data Transmission and ACK Handling:

Each weather station sends weather data to the central server. After each data packet is sent, the station waits for an acknowledgment from the server and adjusts its transmission based on the TCP Reno algorithm.

- **Explanation**: The client sends the compressed data to the server, attaching a packet ID to ensure the server acknowledges the correct packet. `sendWithTimeout()` handles packet transmission with retries (up to `MAX_RETRIES`) if no ACK is received within the defined timeout (`TIMEOUT_SECONDS`). The `waitForAck()` function waits for the ACK, and `tcpRenoCongestionControl()` adjusts `cwnd` and `ssthresh` based on whether the ACK is received or lost. After sending each packet, the client pauses for 1 second (`sleep_for()`) to simulate real-time data transmission.

```
// Function to wait for a specific ACK packet from the server, with timeout handling
bool waitForAck(int client_socket, int packet_id) {
    fd_set read_fds;  // File descriptor set to track readability of client_socket
    FD_ZERO(&read_fds);  // Clear the set
    FD_SET(client_socket, &read_fds);  // Add client_socket to the set

    struct timeval timeout;
    timeout.tv_sec = TIMEOUT_SECONDS;  // Set the timeout value (in seconds)
    timeout.tv_usec = 0;  // Microseconds set to 0

    int select_result = select(client_socket + 1, &read_fds, nullptr, nullptr, &timeout);  // Use select() to wait for data
    if (select_result == 0) {
        // If select() times out (no data received)
        cout << "[" << currentTimestamp() << "] Timeout occurred waiting for ACK." << endl;
        return false;
    } else if (select_result < 0) {
        // If an error occurred with select()
        cerr << "[" << currentTimestamp() << "] Error in select() during ACK wait." << endl;
        return false;
    }

    // If data is available to read, check if it is the expected ACK
    char ack_buffer[BUFFER_SIZE] = {0};  // Buffer to store the received ACK message
    ssize_t ack_received = recv(client_socket, ack_buffer, sizeof(ack_buffer), 0);  // Receive the ACK from server

    string expected_ack = "ACK " + to_string(packet_id);  // Form the expected ACK message
    if (ack_received > 0 && string(ack_buffer, ack_received) == expected_ack) {
        return false;
    }
}
```

```cpp
// Function to send a packet and handle retransmissions in case of timeout or incorrect ACK
bool sendWithTimeout(int client_socket, const string &data, int packet_id) {
    int retries = 0;  // Initialize retry counter
    while (retries < MAX_RETRIES) {
        // Attach packet ID to data before sending it
        string packet_with_id = to_string(packet_id) + ":" + data;

        // Send the packet to the server
        ssize_t sent_bytes = send(client_socket, packet_with_id.c_str(), packet_with_id.size(), 0);
        if (sent_bytes < 0) {
            // If packet sending fails
            cerr << "[" << currentTimestamp() << "] Error sending packet " << packet_id << endl;
            return false;
        }

        // Wait for ACK or timeout
        if (!waitForAck(client_socket, packet_id)) {
            retries++;  // Increment retry count if no ACK or incorrect ACK received
            cout << "[" << currentTimestamp() << "] Retransmitting packet " << packet_id << " (Retry #" << retries << ")" << endl;
            tcpRenoCongestionControl(false);  // Simulate packet loss by invoking congestion control
        } else {
            // If ACK is received successfully
            cout << "[" << currentTimestamp() << "] ACK received for packet " << packet_id << endl;
            tcpRenoCongestionControl(true);  // Update congestion control for successful transmission
            return true;
        }
    }

    // If the packet fails to send even after retries
    cerr << "[" << currentTimestamp() << "] Packet " << packet_id << " failed to send after " << retries << " retries." << endl;
    return false;
}
```

Moreover, the server processes clients concurrently using worker threads (`processTask()`). For each connected client, a thread is spawned (`handleClient()`), where the server continuously receives packets. If data is received, it's placed into a task queue (`task_queue`) for processing. The server can be signaled to stop using the `stop_server` flag, which gracefully stops worker threads once they finish processing the current tasks.

```cpp
// Main loop to accept and handle incoming client connections
while (!stop_server) {
    new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen);
    if (new_socket >= 0) {
        client_count++;
        int client_id = client_count;
        logMessage("New client connected. Client ID: " + to_string(client_id));
        thread client_thread(handleClient, new_socket, client_id);  // Handle the new client in a separate thread
        client_thread.detach();  // Detach the thread to allow it to run independently
    }
}

// Clean up worker threads after server stops
stop_server = true;
queue_cond_var.notify_all();  // Wake up all worker threads so they can exit
for (auto &t : worker_threads) {
    if (t.joinable()) {
        t.join();
    }
}

close(server_fd);  // Close the server socket
```

How to run

Run the following commands on the terminal to perform the given task:

g++ -o server server.cpp -pthread -lz
g++ -o client client.cpp -lz
./server > server_output.log 2>&1 &
./run_clients.sh

Bash script to run multiple client simultaneously:

run_clients.sh

```bash
#!/bin/bash

# Number of clients to spawn
CLIENT_COUNT=5

# Run multiple clients in parallel
for i in $(seq 1 $CLIENT_COUNT); do
./client > client_output_$i.log 2>&1 &
done
```

OUTPUT

We are logging the output for each of the client and the server in their respective log files. The details which are logged in the files are as shown below:

server_output.log

```
server_output.log
[2024-09-08 18:41:34.686] Server started on port 8090
[2024-09-08 18:41:39.498] New client connected. Client ID: 1
[2024-09-08 18:41:39.498] New client connected. Client ID: 2
[2024-09-08 18:41:39.498] New client connected. Client ID: 3
[2024-09-08 18:41:39.498] New client connected. Client ID: 4
[2024-09-08 18:41:39.498] New client connected. Client ID: 5
[2024-09-08 18:41:39.498] Simulating packet loss for Packet 1 from Client 3
[2024-09-08 18:41:39.498] Received Packet 1 from Client 1
[2024-09-08 18:41:39.498] Sent ACK for Packet 1 to Client 1
[2024-09-08 18:41:39.498] Received Packet 1 from Client 2
[2024-09-08 18:41:39.498] Sent ACK for Packet 1 to Client 2
[2024-09-08 18:41:39.499] Received Packet 1 from Client 4
[2024-09-08 18:41:39.499] Sent ACK for Packet 1 to Client 4
[2024-09-08 18:41:39.499] Received Packet 1 from Client 5
[2024-09-08 18:41:39.499] Sent ACK for Packet 1 to Client 5
[2024-09-08 18:41:39.568] Client 1 sent weather data: Temperature: 36.710000C, Humidity: 48.610000%, Pressure: 1092.180000hPa
[2024-09-08 18:41:39.568] Compressed size: 69 bytes, Decompressed size: 71 bytes.
[2024-09-08 18:41:39.568] Client 4 sent weather data: Temperature: 36.710000C, Humidity: 48.610000%, Pressure: 1092.180000hPa
[2024-09-08 18:41:39.568] Compressed size: 69 bytes, Decompressed size: 71 bytes.
[2024-09-08 18:41:39.568] Client 2 sent weather data: Temperature: 36.710000C, Humidity: 48.610000%, Pressure: 1092.180000hPa
[2024-09-08 18:41:39.568] Compressed size: 69 bytes, Decompressed size: 71 bytes.
[2024-09-08 18:41:39.568] Client 5 sent weather data: Temperature: 36.710000C, Humidity: 48.610000%, Pressure: 1092.180000hPa
[2024-09-08 18:41:39.568] Compressed size: 69 bytes, Decompressed size: 71 bytes.
[2024-09-08 18:41:40.499] Received Packet 2 from Client 1
[2024-09-08 18:41:40.499] Sent ACK for Packet 2 to Client 1
[2024-09-08 18:41:40.499] Simulating packet loss for Packet 2 from Client 5
[2024-09-08 18:41:40.499] Received Packet 2 from Client 2
[2024-09-08 18:41:40.500] Sent ACK for Packet 2 to Client 2
```

The server output shows the different clients being connected to the server and sending acknowledgements for their packets. Their respective data is being printed. Moreover, if a packet gets lost in the simulation, that also is being printed in the log file of the server.

client_output_1.log

```
client_output_1.log
[2024-09-08 18:41:39.498] Connected to server.
[2024-09-08 18:41:39.498] Correct ACK (ACK 1) received.
[2024-09-08 18:41:39.498] ACK received for packet 1
[2024-09-08 18:41:39.498] Slow start phase: cwnd doubled to 2
[2024-09-08 18:41:39.498] Successfully sent packet 1 with compressed size 69 bytes.
[2024-09-08 18:41:40.499] Correct ACK (ACK 2) received.
[2024-09-08 18:41:40.499] ACK received for packet 2
[2024-09-08 18:41:40.499] Slow start phase: cwnd doubled to 4
[2024-09-08 18:41:40.499] Successfully sent packet 2 with compressed size 70 bytes.
[2024-09-08 18:41:43.502] Timeout occurred waiting for ACK.
[2024-09-08 18:41:43.502] Retransmitting packet 3 (Retry #1)
[2024-09-08 18:41:43.502] Packet loss detected. Resetting cwnd to 1 and ssthresh to 2
[2024-09-08 18:41:43.503] Correct ACK (ACK 3) received.
[2024-09-08 18:41:43.503] ACK received for packet 3
[2024-09-08 18:41:43.503] Slow start phase: cwnd doubled to 2
[2024-09-08 18:41:43.503] Successfully sent packet 3 with compressed size 69 bytes.
[2024-09-08 18:41:46.505] Timeout occurred waiting for ACK.
[2024-09-08 18:41:46.505] Retransmitting packet 4 (Retry #1)
[2024-09-08 18:41:46.505] Packet loss detected. Resetting cwnd to 1 and ssthresh to 1
[2024-09-08 18:41:46.506] Correct ACK (ACK 4) received.
[2024-09-08 18:41:46.506] ACK received for packet 4
[2024-09-08 18:41:46.506] Congestion avoidance phase: cwnd increased to 2
[2024-09-08 18:41:46.506] Successfully sent packet 4 with compressed size 70 bytes.
[2024-09-08 18:41:47.507] Correct ACK (ACK 5) received.
[2024-09-08 18:41:47.507] ACK received for packet 5
[2024-09-08 18:41:47.507] Congestion avoidance phase: cwnd increased to 3
[2024-09-08 18:41:47.507] Successfully sent packet 5 with compressed size 69 bytes.
[2024-09-08 18:41:48.508] Correct ACK (ACK 6) received.
[2024-09-08 18:41:48.508] ACK received for packet 6
[2024-09-08 18:41:48.508] Congestion avoidance phase: cwnd increased to 4
[2024-09-08 18:41:48.508] Successfully sent packet 6 with compressed size 67 bytes.
[2024-09-08 18:41:49.508] Correct ACK (ACK 7) received.
[2024-09-08 18:41:49.508] ACK received for packet 7
[2024-09-08 18:41:49.509] Congestion avoidance phase: cwnd increased to 5
[2024-09-08 18:41:49.509] Successfully sent packet 7 with compressed size 68 bytes.
```

The dynamic compression level based on data variability ensures that highly repetitive data uses high compression and less repetitive data uses low compression for efficiency.
The output shows the different types of steps which are being taken care of by the code including retransmission, congestion avoidance, receiving acks, detecting packet loss, slow start phase, timeout and successful transmission of the package.