Assignment 3
Group : 31

Ashish Raj 220101020
Aryan Raj 220101019
Vishnu Shakya 220101113
Aditya Nanda 220101006

1. Autonomous Drone Communication System:

The Autonomous Drone Communication System project is designed to facilitate the communication and data exchange between a server and a fleet of autonomous drones. The system supports three key communication channels:

- **Control Commands (UDP):** For sending commands from the server to the drones.
- **Telemetry Data (TCP):** For receiving telemetry data from the drones to monitor their status.
- **File Transfers (Simulated QUIC using TCP):** For transferring files from drones to the server.

The communication channels employ a simple XOR-based encryption to ensure data security during transmission.

The system comprises two main components: the **Server** and the **Client (Drone)**. The server handles multiple drones simultaneously, with each drone acting as a client.

Server Design

The server is responsible for:

- **Handling Control Commands:** Accepting commands from a user, encrypting them, and sending them to drones via UDP.
- **Receiving Telemetry Data:** Listening for telemetry data from drones on a specific TCP port, decrypting the received data, and displaying it to the user.
- **Receiving Files from Drones:** Accepting file transfers from drones over TCP, saving the files locally, and printing their contents.

Client (Drone) Design

Each drone (client) performs the following tasks:

- **Receiving Control Commands:** Listening for commands from the server on a UDP port, decrypting them, and displaying them to the user.

- **Sending Telemetry Data:** Continuously sending telemetry data to the server over TCP.
- **Transferring Files:** Sending specified files to the server over TCP.

Communication Channels

- **Control Commands (UDP):** The use of UDP allows for quick transmission of control commands, suitable for real-time command and control scenarios.
- **Telemetry Data (TCP):** TCP is used to ensure the reliable transmission of telemetry data, as any loss of such data could result in incorrect monitoring.
- **File Transfers (TCP):** File transfers simulate the QUIC protocol using TCP, ensuring reliable and ordered delivery of file data.

Implementation

The project was implemented in C++ using socket programming. The system is designed to run on Unix-like environments, and the code is multithreaded to handle concurrent communication channels efficiently.

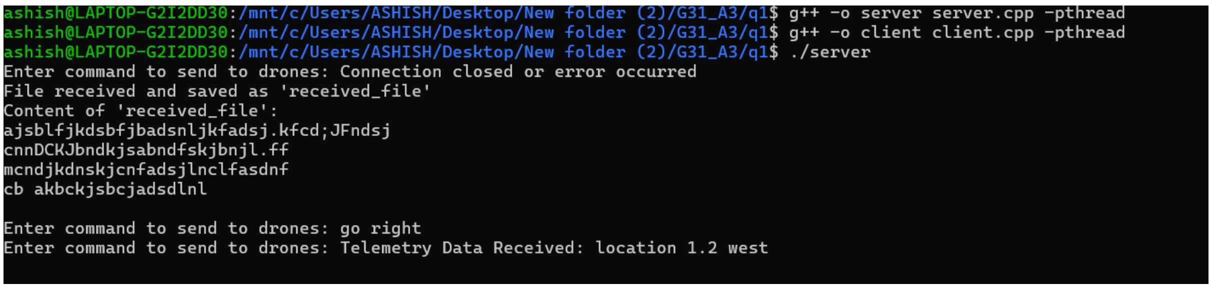
XOR Encryption

To ensure data security, all communications are encrypted using a simple XOR cipher with a fixed key (`'K'`). This method was chosen for its simplicity and to demonstrate basic encryption concepts.

Testing and Validation

The system was tested under various scenarios to validate its functionality:

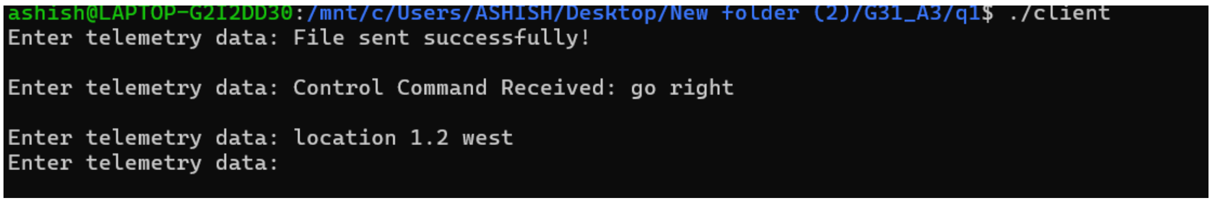
- **Command Transmission:** Verified that control commands sent from the server were correctly received and decrypted by the drones.



```
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q1$ g++ -o server server.cpp -pthread
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q1$ g++ -o client client.cpp -pthread
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q1$ ./server
Enter command to send to drones: Connection closed or error occurred
File received and saved as 'received_file'
Content of 'received_file':
ajsblfjkdsbfjbadsnljkfadsj.kfcd;JFndsj
cnnDCKJbndkjsabndfskjbnjl.ff
mcndjkdnskjcnfadsjlnclfasdnf
cb akbckjsbcjadsdlnl

Enter command to send to drones: go right
Enter command to send to drones: Telemetry Data Received: location 1.2 west
```

- **Telemetry Data:** Tested continuous telemetry data transmission from drones to the server and ensured that data integrity was maintained.

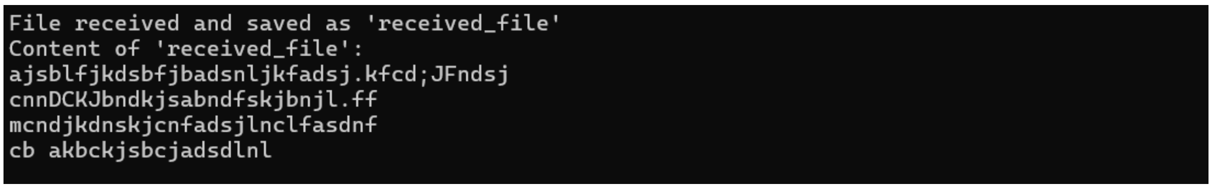


```
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q1$ ./client
Enter telemetry data: File sent successfully!

Enter telemetry data: Control Command Received: go right

Enter telemetry data: location 1.2 west
Enter telemetry data:
```

- **File Transfers:** Conducted file transfers from drones to the server, ensuring that files were received without corruption and correctly saved.



```
File received and saved as 'received_file'
Content of 'received_file':
ajsblfjkdsbfjbadsnljkfadsj.kfcd;JFndsj
cnnDCKJbndkjsabndfskjbnjl.ff
mcndjkdnskjcnfadsjlnclfasdnf
cb akbckjsbcjadsdlnl
```

2. Real-Time Weather Monitoring System:

This project involves the development of a sophisticated real-time weather monitoring system designed to serve a large city. The system architecture comprises multiple weather stations scattered across the city, each sending real-time weather data (e.g., temperature, humidity, air pressure) to a central server. The central server processes this data and provides timely updates for monitoring and analysis. The system is designed to handle high data throughput, adapt to varying network conditions, and efficiently manage data to prevent overloads.

Components:

- **Weather Stations (Clients):** Devices responsible for collecting and transmitting weather data to the central server.
- **Central Server:** The central unit that receives, processes, and displays the weather data from all weather stations.

Communication Protocols:

- **Data Transmission:** Implemented using TCP Reno, a congestion control algorithm to manage the flow of data and adapt to changing network conditions.
- **Network Simulation:** Simulated network constraints, including limited bandwidth and server capacity, to test the adaptability of the system under varying conditions.
- **Data Compression:** Dynamic compression of weather data before transmission to optimize bandwidth usage and ensure faster transmission without sacrificing data integrity.

Efficient Data Handling:

- **Server-Side Management:** The server is designed to handle data from multiple weather stations simultaneously. It uses an optimized data handling mechanism to process incoming data efficiently without being overwhelmed.
- **Load Balancing:** Implemented load balancing techniques to distribute the processing load evenly across the server resources.

Adaptive Data Transmission (TCP Reno):

- **Congestion Control:** The weather stations use the TCP Reno algorithm to adjust their data transmission rate based on current network conditions. TCP Reno helps prevent network congestion by reducing the transmission rate when packet loss is detected and increasing it when the network is stable.
- **Slow Start and Congestion Avoidance:** The transmission starts with a low congestion window (cwnd) and increases exponentially during the slow start phase. Once the threshold is reached, it switches to congestion avoidance mode, where the cwnd is increased linearly.

Simulated Network Constraints:

- **Bandwidth Limitation:** The system was tested under simulated conditions where the available bandwidth was limited, forcing the weather stations to adapt their transmission rates to avoid overwhelming the server.
- **Server Capacity Simulation:** Simulated server processing limits were introduced to evaluate how well the system handles high data loads under constrained conditions.

Dynamic Data Compression:

- **Compression Algorithm:** A dynamic compression algorithm was implemented, which adapts to the nature of the data being transmitted. The compression rate adjusts automatically based on data characteristics, balancing between reduced data size and transmission speed.
- **Decompression on Receipt:** The server decompresses the incoming data in real-time, ensuring that the data is ready for processing and display without delay.

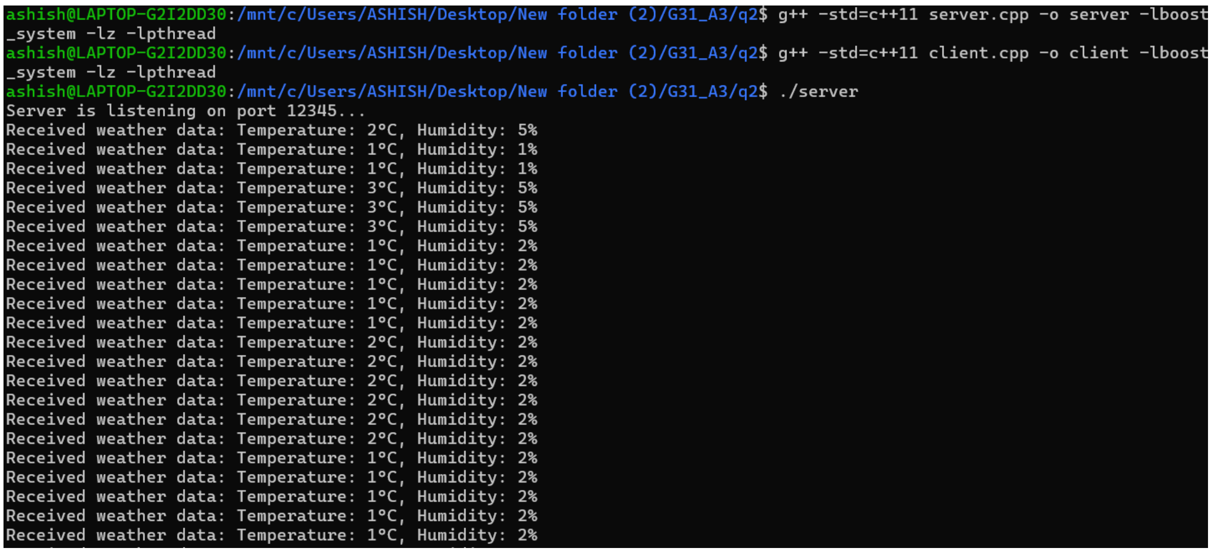
High Data Throughput:

- **Stress Testing:** The system was stress-tested with multiple weather stations transmitting data simultaneously. The server was able to handle the high throughput without data loss or significant delays.

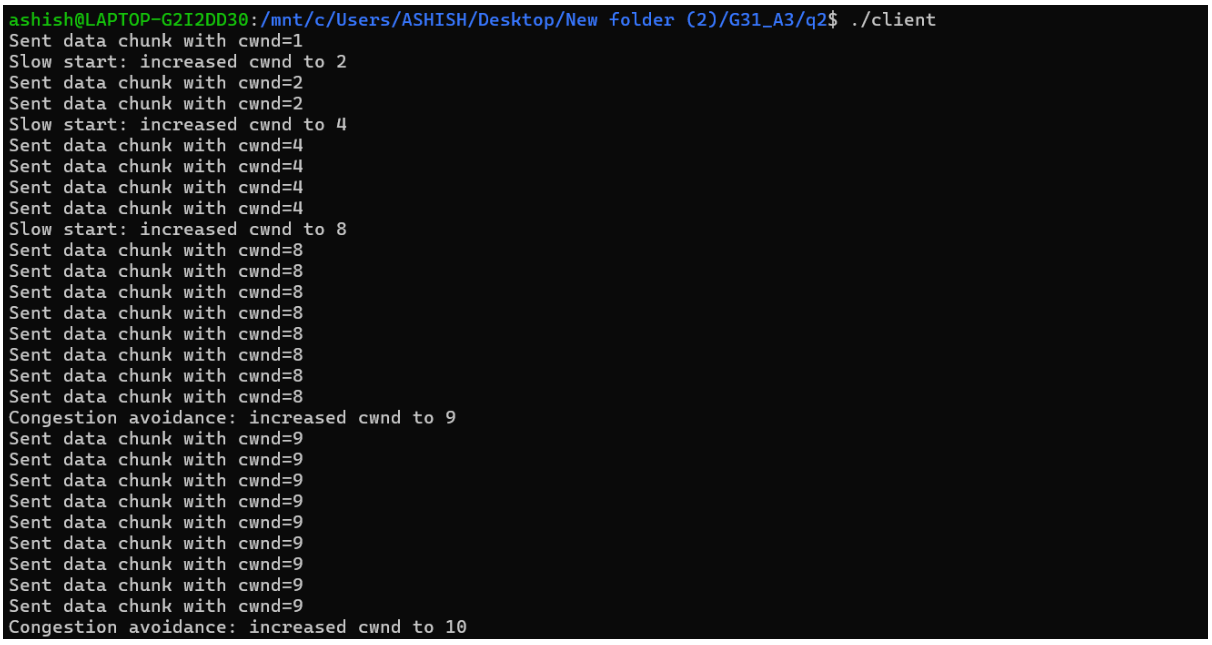
Network Adaptability:

- **Varying Network Conditions:** The system was tested under different simulated network conditions, including varying bandwidth and latency. The TCP Reno algorithm effectively adjusted the data transmission rates to maintain network stability and responsiveness.

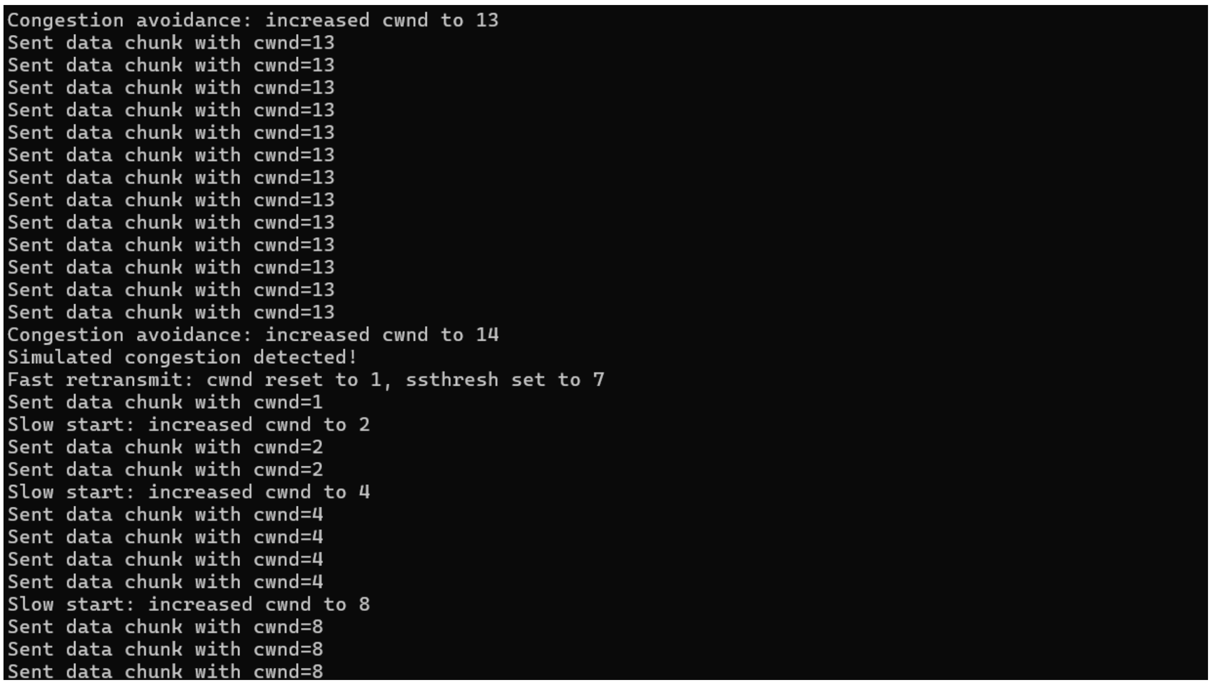
Simulated Output Results:



```
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q2$ g++ -std=c++11 server.cpp -o server -lboost
_system -lz -lpthread
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q2$ g++ -std=c++11 client.cpp -o client -lboost
_system -lz -lpthread
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q2$ ./server
Server is listening on port 12345...
Received weather data: Temperature: 2°C, Humidity: 5%
Received weather data: Temperature: 1°C, Humidity: 1%
Received weather data: Temperature: 1°C, Humidity: 1%
Received weather data: Temperature: 3°C, Humidity: 5%
Received weather data: Temperature: 3°C, Humidity: 5%
Received weather data: Temperature: 3°C, Humidity: 5%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 2°C, Humidity: 2%
Received weather data: Temperature: 2°C, Humidity: 2%
Received weather data: Temperature: 2°C, Humidity: 2%
Received weather data: Temperature: 2°C, Humidity: 2%
Received weather data: Temperature: 2°C, Humidity: 2%
Received weather data: Temperature: 2°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
Received weather data: Temperature: 1°C, Humidity: 2%
```



```
ashish@LAPTOP-G2I2DD30:/mnt/c/Users/ASHISH/Desktop/New folder (2)/G31_A3/q2$ ./client
Sent data chunk with cwnd=1
Slow start: increased cwnd to 2
Sent data chunk with cwnd=2
Sent data chunk with cwnd=2
Slow start: increased cwnd to 4
Sent data chunk with cwnd=4
Sent data chunk with cwnd=4
Sent data chunk with cwnd=4
Sent data chunk with cwnd=4
Slow start: increased cwnd to 8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Congestion avoidance: increased cwnd to 9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Sent data chunk with cwnd=9
Congestion avoidance: increased cwnd to 10
```



```
Congestion avoidance: increased cwnd to 13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Sent data chunk with cwnd=13
Congestion avoidance: increased cwnd to 14
Simulated congestion detected!
Fast retransmit: cwnd reset to 1, ssthresh set to 7
Sent data chunk with cwnd=1
Slow start: increased cwnd to 2
Sent data chunk with cwnd=2
Sent data chunk with cwnd=2
Slow start: increased cwnd to 4
Sent data chunk with cwnd=4
Sent data chunk with cwnd=4
Sent data chunk with cwnd=4
Sent data chunk with cwnd=4
Slow start: increased cwnd to 8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
Sent data chunk with cwnd=8
```

saved in output.txt

The output reflects the behavior of the TCP congestion control algorithm during file transfer, simulating both "Slow Start" and "Congestion Avoidance" phases. The process includes the following key steps:

1. **Slow Start Phase**:
   - The congestion window (cwnd) begins at 1 and doubles after each round-trip time (RTT) until it reaches the slow start threshold (ssthresh). This results in exponential growth of the cwnd.
   - You can see the increase from `cwnd=1` to `cwnd=2`, `cwnd=4`, `cwnd=8`, and so on. Each increment is followed by sending multiple data chunks.
2. **Congestion Avoidance Phase**:
   - Once `cwnd` surpasses `ssthresh`, the algorithm shifts to Congestion Avoidance mode, where `cwnd` increases linearly, by 1 unit, after each RTT.
   - The output reflects this linear growth, where after each RTT, the `cwnd` is increased by 1 (e.g., from `cwnd=9` to `cwnd=10`).
3. **Simulated Congestion Detection**:
   - Congestion is simulated to occur at specific points. When detected, the `cwnd` is reset to 1, and `ssthresh` is halved (e.g., set to 7, 5, or 7.5). This triggers a "Fast Retransmit," and the algorithm re-enters the Slow Start phase, gradually increasing `cwnd` until it reaches `ssthresh`.
4. **Fast Retransmit**:
   - After congestion is detected, the algorithm skips Slow Start's exponential growth and directly enters Congestion Avoidance once `cwnd` reaches the new `ssthresh`.