Computer Networks Lab 03

Group 35

Srinjoy Som	220123074
Shubham Kumar Jha	220123081
Yash Singhal	220123072

All the codes mentioned are programmed for the Windows Operating System, so compile and run them on Windows only.

Question 1.

We must implement a central server and multiple drones.

Here, we have implemented a server with a fixed IP and PORT number, while the client can have different IPs, given that they all can send data to the central server as its IP and PORT is available.

To implement **Mode 1 (Control Commands)** which is a high-performance mode where the server sends control commands to drones with minimal latency, we use a UDP connection as it gives minimal latency and works in the face of congestion.

To implement **Mode 2 (Telemetry Data)** which is a reliable mode where drones send telemetry data to the server and we need to ensure data is delivered accurately and, in order, we use a TCP connection as it provides accurate and in-order data delivery.

To implement **Mode 3 (File Transfers)** in which we need to transfer large files from drones to the server efficiently, we use the QUIC protocol.

Server Side Code:

We have implemented a server capable of sending commands, receiving telemetry data and file transfers from clients.

triplet class: Stores the 3D location(x, y, and z) of a drone.

dronestate struct: Holds the drone's speed, position (as a triplet), and state (indicating whether the drone is READY).

Y: A counter for the number of drones sending telemetry data via UDP.

store: A vector that stores the addresses of the drones (using **sockaddr_in**).

myArray[200]: A character array initialised with a repeating pattern "ABC123", which is used in XOR encryption/decryption.

The function **xorData** is used to XOR-decrypt data by iterating over it and modifying each byte using the corresponding byte in **myArray**.

```
class triplet {
public:
    int x;
    int y;
    int z;
};

struct dronestate {
    int speed;
    triplet pos;
    int state;
};

int Y;
vector<sockaddr_in> store;
char myArray[200];

// XOR function
void xorData(char* data, int len) {
    for (int i = 0; i < len; i++) {
        data[i] ^= myArray[i % 200];
    }
}
```

Now, the program starts execution with main(), which initialises the XOR encryption pattern (myArray) with "ABC123". It then initialises Winsock and creates both a TCP and UDP socket. Binds both sockets to IP address **127.0.0.1** and port **55555**. Then we start three threads:

1. One for accepting telemetry data over TCP.
2. One for receiving UDP messages.
3. One for periodically sending commands over UDP.

Joins the threads and cleans up the sockets and Winsock before exiting.

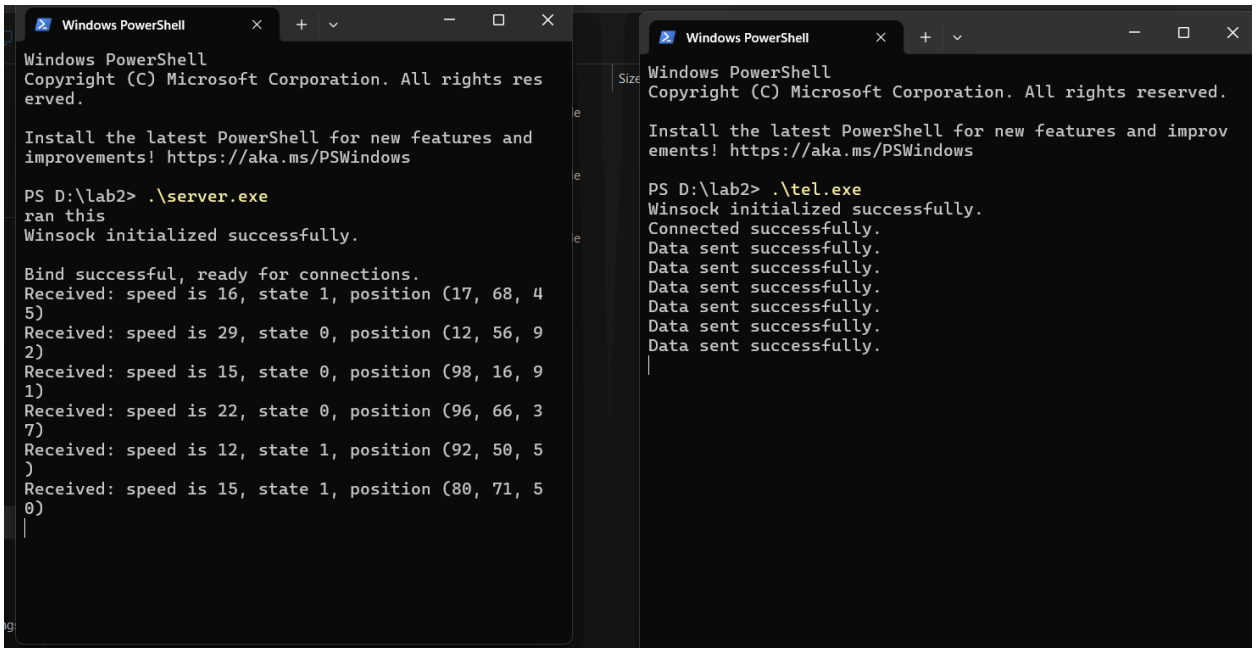
The program uses multi-threading to handle different tasks concurrently:

- **TCP thread** (**tcpThread**) handles incoming telemetry data over TCP.
- **UDP thread** (**udpThread**) handles incoming UDP messages from drones.
- **Command thread** (**commandThread**) sends commands periodically (every 5 seconds) to all drones over UDP.

```
int main() {
    // Fill the myArray with some data
    const char* pattern = "ABC123";
    int patternLength = 6; // Length of "ABC123"
    for (int i = 0; i < 200; i++) {
        myArray[i] = pattern[i % patternLength];
    }

    cout << "ran this " << endl;
    WSADATA wsadata;
    int wsaerr;
    WORD versionreq = MAKEWORD(2, 2);
    wsaerr = WSAStartup(versionreq, &wsadata);
    if (wsaerr != 0) {
        cout << "Failed to initialize Winsock: " << wsaerr << endl;
    } else {
        cout << "Winsock initialized successfully.\n" << endl;
    }

    SOCKET server = INVALID_SOCKET;
    server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    SOCKET server_udp = INVALID_SOCKET;
    server_udp = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (server == INVALID_SOCKET || server_udp == INVALID_SOCKET) {
        cout << "SOCKET COULD NOT BE FORMED: " << WSAGetLastError() << endl;
        WSACleanup();
        return 0;
    } else {
        sockaddr_in serv;
        serv.sin_family = AF_INET;
        serv.sin_addr.s_addr = inet_addr("127.0.0.1");
        serv.sin_port = htons(55555);
```

```
        if (bind(server, (SOCKADDR*)&serv, sizeof(serv)) == SOCKET_ERROR) {
            cout << "Bind failed: " << WSAGetLastError() << endl;
            WSACleanup();
            return 0;
        }
        if (bind(server_udp, (SOCKADDR*)&serv, sizeof(serv)) == SOCKET_ERROR) {
            cout << "Bind failed: " << WSAGetLastError() << endl;
            WSACleanup();
            return 0;
        }
        cout << "Bind successful, ready for connections." << endl;
        if (listen(server, 5) == SOCKET_ERROR) {
            cout << "Listen failed: " << WSAGetLastError() << endl;
            return -1;
        }
    }

    thread tcpThread(accept_telemetry, server);
    thread udpThread(udp_thread, server_udp);
    thread commandThread(command, server_udp);
    tcpThread.join();
    udpThread.join();
    commandThread.join();

    closesocket(server);
    closesocket(server_udp);
    WSACleanup();
    return 0;
}
```

The function **accept_telemetry** accepts TCP connection and calls **tcp_connect_manager**.



The function **tcp_connect_manager** handles incoming TCP connections. It receives telemetry data (using **recv**) in the form of the **dronestate** structure. The data is then decrypted using the **xorData** function. After decryption, it prints the drone's speed, state, and position.

```
void tcp_connect_manager(SOCKET acceptsock) {
    struct dronestate curr;
    while (true) { // Keep the connection open to receive multiple messages
        int bytess = recv(acceptsock, (char*)(&curr), sizeof(curr), 0);
        if (bytess <= 0) {
            // Connection closed or error occurred
            cout << "Connection closed or recv failed: " << WSAGetLastError() << endl;
            break;
        } else {
            xorData((char*)&curr, sizeof(curr));  // XOR after receiving
            printf("Received: speed is %d, state %d, position (%d, %d, %d)\n",
                   curr.speed, curr.state, curr.pos.x, curr.pos.y, curr.pos.z);
        }
    }
    closesocket(acceptsock); // Close the socket after done with receiving messages
}

void accept_telemetry(SOCKET server) {
    while (true) {
        SOCKET acceptsock;
        acceptsock = accept(server, NULL, NULL);
        if (acceptsock == INVALID_SOCKET) {
            cout << "accept failed" << endl;
            WSACleanup();
            return;
        }
        tcp_connect_manager(acceptsock);
    }
}
```

This is our implementation for sending large files over UDP connection.
The function **udp_thread** calls **udp_accept_manager** function**.**

The function **udp_accept_manager** handles incoming UDP messages. It receives a message from a drone, decrypts it, and prints it.

The drone's address is stored in **store**, and **Y** (the number of drones) is incremented.

```
void udp_accept_manager(SOCKET server_udp) {
    char recive[200];
    sockaddr_in clientAddr;
    int clength = int(sizeof(clientAddr));
    int bytess = recvfrom(server_udp, recive, 200, 0, (struct sockaddr*)&clientAddr, &clength);
    if (bytess >= 0) {
        xorData(recive, bytess);  // XOR after receiving
        printf("Received: %s\n", recive);
        store.push_back(clientAddr);
        Y++;
    }
}

void udp_thread(SOCKET udp_sock) {
    while (true) {
        udp_accept_manager(udp_sock);
    }
}
```

The function **command** is used to send commands to the drones. It calls **sendcomm** function and then **sleeps** for 5 seconds.

The function **sendcomm** sends a control command ("go right") to all drones stored in the **store** vector. Before sending, the command is XOR-encrypted. The command is sent to each drone's **UDP** address using **sendto**.

```
void sendcomm(SOCKET server_udp) {
    char COMMAND[200] = "go right";
    xorData(COMMAND, strlen(COMMAND));  // XOR before sending
    if (Y == 0) return;
    int bytes;
    for (int i = 1; i <= Y; i++) {
        bytes = sendto(server_udp, COMMAND, strlen(COMMAND), 0, (struct sockaddr*)&store[i - 1], sizeof(store
        [i - 1]));
    }
    if (bytes <= 0) {
        cout << "not working" << endl;
        return;
    }
}

void command(SOCKET udp_sock) {
    while (true) {
        sendcomm(udp_sock);
        this_thread::sleep_for(chrono::seconds(5));
    }
}
```

Client Side Code:

We have implemented a client that is capable of initialising connection with the server, sending telemetry data, receiving commands from the server and sending large files to the server.

myArray[200]: A character array initialised with a repeating pattern "ABC123", which is used in XOR encryption/decryption.

The function **xorData** is used to XOR-decrypt data by iterating over it and modifying each byte using the corresponding byte in **myArray**.

Now, the program starts execution with main(), which initialises the XOR encryption pattern (myArray) with "ABC123". Then, it creates a UDP socket. If the socket creation fails, it outputs an error message, cleans up the Winsock resources using WSACleanup, and terminates.

The client defines a sockaddr_in structure (client_struc) to store the server's address, like the IP version, IP and PORT.

Then, the client prepares a message ("READY") to send to the server. The message is XOR-encrypted using **xorData**. It sends the encrypted message using **sendto**, which sends the data to the specified server. If sending fails, the program outputs an error message and cleans up. If sending succeeds, it prints "Data sent".

After sending the message, the client enters an infinite loop to receive messages from the server continuously. The client receives data from the server using **recvfrom**. Once the message is received, it decrypts it using **xorData**. The decrypted message is then printed to the console.

Once the communication is done, the client closes the socket and cleans up Winsock resources.

CHALLENGES:

In the third part, where the client has to send Files via QUIC, we need additional libraries that are specific to certain version of operating system and very hard to set up. We tried to use MSquic by Microsoft and Quiche by Cloudflare. I tried to set up them with instructions found on the internet but failed to do so.

Question 2.

We must implement a central server(server) and multiple weather stations(client).

Here, we have implemented a server with a fixed IP and PORT number, while the client can have different IPs, given that they all can send data to the central server as its IP and PORT is available.
Also, we handle each client in a separate thread, preventing the server from getting overwhelmed with multiple clients.
So, as per the question, our code has features like:

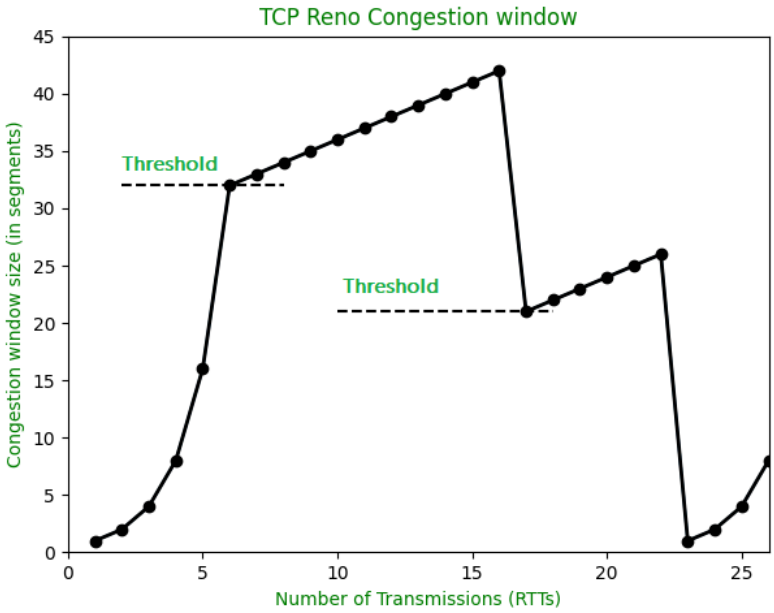
1. **DATA HANDLING:** First of all, we have compressed the data so to decrease packet size and use of the network before sending the data via TCP.
2. **CONGESTION CONTROL:** TCP already has some TCP congestion control algorithms in the transport layer. We are going to replicate some of the TCP Reno protocol in the application layer. To reduce transmission rate when the network starts to become congested, thus preventing significant packet losses. This ensures that the network remains responsive and data is transmitted efficiently.

TCP Reno is a congestion control algorithm that builds on TCP Tahoe, incorporating Slow Start, AIMD (Additive Increase Multiplicative Decrease), Fast Retransmit, and Fast Recovery.

Key Phases:

1. **Slow Start:** The congestion window (cwnd) increases exponentially until it reaches the Slow Start Threshold (ssthresh). After this, AIMD takes over.
2. **AIMD:** When cwnd reaches ssthresh, Additive Increase raises cwnd by 1 per RTT, while Multiplicative Decrease cuts ssthresh to 50% of cwnd on packet loss.
3. **Fast Retransmit:** Triggered by 3 duplicate ACKs, it detects packet loss and resets cwnd to the initial size.
4. **Fast Recovery:** Upon loss detection by duplicate ACKs, cwnd is halved. If loss is detected by a Retransmission Timeout (RTO), cwnd resets to the initial size for recovery from severe congestion.

On packet loss detection through RTO, reset **cwnd** to the initial window size. If the RTO timer expires, it means that the network is badly congested. So, the **cwnd** has to be reduced to the initial value to allow the network to recover from congestion.



3. Network Constraints :

In the server code, we have set the max connection limit to 10 so a maximum of 10 weather channels can be connected to the server simultaneously. Also, we have restricted the data flow by the simulated congestion control over the existing one. Also, we have set a maximum cwnd even if there is no packet loss, the cwnd becomes half if it tries to cross the max_cwnd.

4. Data Compression:

We have used an additional package, "**zlib**" which is used for compressing and decompressing data.

Server Side Code:

We have implemented a multi-threaded server that listens for client connections on port (65432) and handles multiple clients concurrently. The server supports multiple clients simultaneously by creating new threads for each connection and processing data in parallel. We first set up the server using Winsock2 to initialize networking. It creates a TCP socket, binds it to a port, and listens for incoming connections. When a client connects, the server spawns a new thread (**handle_client**) to manage the connection. The server receives data from the client in compressed form.

```
WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    cerr << "WSAStartup failed." <<endl;
    return 1;
}

SOCKET server_socket = socket(AF_INET, SOCK_STREAM, 0);
if (server_socket == INVALID_SOCKET) {
    cerr << "Failed to create socket." << endl;
    WSACleanup();
    return 1;
}

sockaddr_in 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_socket, (sockaddr*)&server_addr, sizeof(server_addr)) == SOCKET_ERROR) {
    cerr << "Bind failed." << endl;
    closesocket(server_socket);
    WSACleanup();
    return 1;
}
```

```
if (listen(server_socket, MAX_CLIENTS) == SOCKET_ERROR) {
    cerr << "Listen failed." << endl;
    closesocket(server_socket);
    WSACleanup();
    return 1;
}

cout << "Server listening on port " << PORT << endl;

thread data_processing_thread(process_data);
data_processing_thread.detach();

while (true) {
    SOCKET client_socket = accept(server_socket, nullptr, nullptr);
    if (client_socket == INVALID_SOCKET) {
        cerr << "Accept failed." << endl;
        continue;
    }

    thread client_thread(handle_client, client_socket);
    client_thread.detach();
}

closesocket(server_socket);
WSACleanup();
return 0;
```

The received data is decompressed using the zlib library (**uncompress**). If decompression is successful, the data is pushed into a shared queue.

A mutex and condition_variable ensure safe access to the shared queue, enabling synchronization between client threads and the data-processing thread.

```
void handle_client(SOCKET client_socket) {
    char buffer[BUFFER_SIZE];
    while (true) {
        int bytes_received = recv(client_socket, buffer, BUFFER_SIZE, 0);
        if (bytes_received <= 0) {
            cerr << "Client disconnected or error occurred." << endl;
            break;
        }

        // Decompress data
        uLongf decompressed_size = BUFFER_SIZE * 2;
        vector<char> decompressed_data(decompressed_size);
        int result = uncompress((Bytef*)decompressed_data.data(), &decompressed_size, (const Bytef*)buffer,
        bytes_received);

        if (result == Z_OK) {
            decompressed_data.resize(decompressed_size);
            lock_guard<mutex> lock(queue_mutex);
            data_queue.push(decompressed_data);
            data_available.notify_one();
        } else {
            cerr << "Failed to decompress data." << endl;
        }
    }

    closesocket(client_socket);
}
```

A separate thread (**process_data**) continuously waits for new data in the queue. When data is available, it processes and prints the decompressed data.

```
// Function to process data in the queue
void process_data() {
    while (true) {
        unique_lock<mutex> lock(queue_mutex);
        data_available.wait(lock, []{ return !data_queue.empty(); });

        auto data = data_queue.front();
        data_queue.pop();
        lock.unlock();

        cout << "Processed data: " << string(data.begin(), data.end()) << endl;
    }
}
```

Client Side Code:

We have implemented a weather station(client) that uses TCP Reno congestion control to send weather data (temperature, humidity, and pressure).

The code initializes Winsock (WSAStartup), creates a TCP socket, and connects to a server at **127.0.0.1** on port **65432**.

```
WSADATA wsaData;
if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
    cerr << "WSAStartup failed." << endl;
    return 1;
}

SOCKET client_socket = socket(AF_INET, SOCK_STREAM, 0);
if (client_socket == INVALID_SOCKET) {
    cerr << "Failed to create socket." << endl;
    WSACleanup();
    return 1;
}

sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
server_addr.sin_port = htons(PORT);

if (connect(client_socket, (sockaddr*)&server_addr, sizeof(server_addr)) == SOCKET_ERROR) {
    cerr << "Connection failed." << endl;
    closesocket(client_socket);
    WSACleanup();
    return 1;
}
```

```
// Implement TCP Reno
tcp_reno(client_socket);

closesocket(client_socket);
WSACleanup();
return 0;
```

Inside the tcp_reno function, we define **cwnd**, **ssthresh** and **acked** counter.

Then, we generate random values for temperature, humidity, and pressure, which are generated and formatted into a string (**weather_data**).

```
// Function to simulate TCP Reno congestion control
void tcp_reno(SOCKET client_socket) {

   int cwnd = 1;            // Congestion window starts at 1
   int ssthresh = 64;       // Slow start threshold
   int acked = 0;           // Simulated acknowledgment counter

   while (true) {
       int temp=-10 +rand()% 45;
       int humid = 10 +rand()%90;
       int pressure =800 + rand()%200;
       string weather_data = "Temperature: "+to_string(temp)+"C, Humidity:"+ to_string(humid)+"%, Pressure: "
       +to_string(pressure)+ " hPa";
       // Compress data before sending
       vector<char> compressed_data = compress_data(weather_data);
       if (compressed_data.empty()) {
           continue;
       }
```

This string is then compressed using the zlib library (**compress_data** function).

```cpp
// Function to compress data before sending
vector<char> compress_data(const string& data) {
    uLongf compressed_size = compressBound(data.size());
    vector<char> compressed_data(compressed_size);

    int result = compress((Bytef*)compressed_data.data(), &compressed_size, (const Bytef*)data.c_str(), data.size());
    if (result == Z_OK) {
        compressed_data.resize(compressed_size);
    } else {
        cerr << "Compression failed." << endl;
        compressed_data.clear();
    }

    return compressed_data;
}
```

Now, we implement the TCP Reno Algorithm,

Slow Start: The congestion window (**cwnd**) doubles (exponential growth) until it reaches the slow start threshold (**ssthresh**). After which, the **AIMD** phase starts.

AIMD: Additive Increase increases cwnd by 1, and Multiplicative Decrease reduces ssthresh to 50% of **cwnd**. Data is sent in chunks based on the current congestion window (**cwnd**), simulating how TCP controls data flow.

Simulated Packet Loss: For sending short lengths of data from one port of localhost to another, the probability of packet dropping is rare, so to demonstrate how it would affect the **ssthresh** and **cwnd** every 20 ACKs, packet loss is simulated, causing the congestion window and threshold to be halved. A minor delay (**Sleep**) is added between transmissions to simulate network delays.

4 things occur when a packet loss is detected in **TCP Reno:**

1. **cwnd** is reduced to half, i.e. **cwnd = cwnd/2**
2. **ssthresh** is reduced to half of cwnd, i.e. **ssthresh = cwnd/2**
3. The sender becomes silent for half a window, called a half window of silence.
4. The sender enters the Fast Recovery phase.

```
while (true) {
    // Send the compressed data in chunks based on the current congestion window
    int data_size = compressed_data.size();
    int bytes_sent = 0;

    while (bytes_sent < data_size) {
        int chunk_size = min(cwnd, data_size - bytes_sent);
        int result = send(client_socket, compressed_data.data() + bytes_sent, chunk_size, 0);

        if (result == SOCKET_ERROR) {
            cerr << "Failed to send data." << endl;
            return;
        }

        bytes_sent += result;

        // Simulate receiving an ACK (for simplicity, assuming ACKs are always received)
        acked++;

        // Slow Start phase
        if (cwnd < ssthresh) {
            cwnd = min(cwnd * 2, MAX_CWND);  // Exponential growth
        }
        // Congestion Avoidance phase
        else {
            cwnd = min(cwnd + 1, MAX_CWND);  // Linear growth
        }
```

```
        // Simulate waiting time before sending the next chunk of data
        Sleep(100); // Simulate network delay
    }

    // Simulate packet loss after some time
    if (acked % 20 == 0) {
        ssthresh =cwnd / 2;  // Cut ssthresh in half
        cwnd = cwnd / 2;
        cout << "Packet loss detected, reducing cwnd to " << cwnd << endl;
    }

    // Simulate delay between sending weather data
    Sleep(1000);
}
```