MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

Mini Project Report
of
Computer Networks Lab (CSE 3162)

Implementation of TCP-like functionalities over UDP

SUBMITTED
BY

Arman Malik 210905200, ROLL NO-37, SECTION-A

Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal.

November 2023

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Manipal
17/11/2023

CERTIFICATE

This is to certify that the project titled**. Implementation of TCP-like functionalities over UDP**
is a record of the bonafide work done by **Arman Malik (Reg. No. 210905200) Ashutosh Mishra (Reg. No. 210905320), Rahul Ranjan Mishra (Reg. No. 210905405)** submitted in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology (B.Tech.) in COMPUTER SCIENCE & ENGINEERING of Manipal Institute of Technology, Manipal, Karnataka, (A Constituent Institute of Manipal Academy of Higher Education), during the academic year 2022-2023.

Name and Signature of Examiners:

Manmohana Krishna Sir, Assistant Professor - Selection Grade, CSE Dept.

TABLE OF CONTENTS

INTRODUCTION

We are making a reliable version of UDP by implementing TCP like functionalities over a UDP connection.

To achieve that, we have implemented the following over a stream of UDP packets; implementation should be same as in the TCP :

a. Acknowledgement: Please implement both single packet and cummulative ack.

b. Sequence number: same as in TCP

c. Retransmissions

d. Detecting duplicates and discarding them.

e. Congestion control

f. Flow Control

We have also developed other necessary programs that will let us test these functionalities. This may require the following (and some more):

a. A UDP client and server

b. a mechanism to generate a large number of UDP packets that act as both packet and acknowledgements depending on the need.

c. a random way of deleting some packets so that you can simulate/show loss or delayed arrival.

Problem Statement:

Develop a reliable data transfer system using the User Datagram Protocol (UDP) to transfer large volumes of data between a sender and receiver. The primary challenge is to ensure data integrity and reliability in an inherently unreliable communication protocol like UDP, implementing a Stop-and-Wait protocol with a sliding window mechanism.

Objectives:

Reliable Data Transfer: Design a system that ensures reliable data transfer between a sender and receiver over an unreliable network using UDP.

Error Handling: Implement mechanisms to handle packet loss, duplication, and out-of-order delivery to guarantee data integrity.

Sliding Window Protocol: Implement a sliding window protocol to optimize data transfer efficiency and utilize network resources effectively.

Timeout Mechanism: Develop a timeout mechanism to retransmit lost or delayed packets, ensuring reliable delivery even in the presence of network congestion or errors.

Dynamic Window Sizing: Investigate and implement techniques for dynamically adjusting the window size based on network conditions for optimal throughput.

Performance Evaluation: Measure and analyse the performance of the system concerning throughput, latency, and efficiency under varying network conditions and packet loss scenarios.

Robustness and Scalability: Evaluate the system's robustness and scalability by testing it with varying data sizes and network conditions to ensure it meets real-world requirements.

Documentation and Report: Document the system architecture, algorithms, implementation details, challenges faced, and insights gained during development in a comprehensive project report.

Future Enhancements: Explore potential improvements, such as implementing selective repeat, congestion control mechanisms, or integrating with other protocols for enhanced reliability.

Code Optimization and Cleanliness: Review and optimize the codebase for better readability, maintainability, and adherence to coding standards.

Methodology:

Understanding UDP and its Limitations:

Conducted a comprehensive study of the User Datagram Protocol (UDP) to comprehend its advantages and limitations in comparison to other transport layer protocols like TCP.

Problem Analysis and Requirement Gathering:

Identified the need for a reliable data transfer system over UDP and defined the requirements for ensuring data integrity and reliability in an unreliable network.

Protocol Design and Algorithm Selection:

Selected the Stop-and-Wait protocol with a sliding window mechanism as the basis for the reliable data transfer system.

Designed the protocol flow, packet structure, acknowledgment handling, and error recovery mechanisms to achieve reliability.

Code Implementation:

Developed the sender and receiver applications in C programming language, implementing the designed protocol and algorithms.

Utilized socket programming APIs to establish UDP connections, transmit and receive data packets, and manage acknowledgments.

Packet Handling and Error Simulation:

Implemented packet creation, segmentation, and reassembly at the sender and receiver ends.

Simulated packet loss using a configurable flag to test the system's behaviour in the presence of network errors.

Timeout and Retransmission Mechanism:

Integrated a timeout mechanism to detect packet loss and retransmit the packets that were not acknowledged within a specified time.

Managed the sender's window size dynamically based on acknowledgments received and timeouts experienced.

Testing and Evaluation:

Conducted rigorous testing under various network conditions, including different data sizes, packet loss scenarios, and network delays.

Evaluated the system's performance in terms of throughput, latency, and reliability.

Performance Analysis and Results:

Analysed the collected data to assess the system's performance metrics, identifying strengths, weaknesses, and areas for improvement.

Generated graphical representations and statistical analyses of the performance measurements for a comprehensive understanding.

By following this methodology, the project aimed to design, implement, and evaluate a reliable data transfer system over UDP, addressing the challenges posed by an unreliable network environment while providing insights into its performance and scope for further improvements.

Description:

The assignment code consists of two files: Sender.c and Receiver.c. These files implement UDP Client and Server respectively, with additional functionalities as desired in the assignment.

The sender assumes the following information:

1. Server/Receiver is at localhost with port 1222, unless specified.
2. Loss Flag is 0
3. Total Data Size is 1,00,000 bytes.
4. Maximum Segment Size (MSS) for each packet is 1,000 bytes.
5. Initially, seq. No = 0.
6. Initially, window size = 1000.

Just for the sake of visualization, the data sent between the sender and receiver is a string of a's. It is divided into packets of size MSS. Sequence no. is then added to this packet with the help of ByteArrayOutputStream and DataOutputStream. In case the loss Flag = 0, the packet is sent to the receiver. Immidiately, we start a timer for that packet and store it in an Array List. Also, the packet in transmission is added to an Array List. The packet can also be dropped with some random probability or when loss Flag = 1.

The packet is then received by the Receiver/Server. The server is always active and keeps on listening for any incoming packets. For every packet received, we store them in an Array List. The receiver decodes the sequence no. of each packet and analyses them. If one of the packets is missing, it sends back the previous ack. However, if all the packets have successfully arrived, it sends the cumulative ack.

This ack is received by the Sender/Client. If the ack received is as desired, we increase the window size and send next packets. Also, the earlier Array Lists are cleared. If the ack is not received and timeout occurs, we reduce the window size to 1 MSS, as required by TCP. And we resend the packets with previous sequence no. Timeout can occur in two ways: either the packet's timer expires, or the socket's timer expires. In both the cases, we restart from 1 MSS. Triple duplicate ack scenario has not been implemented. Also, the ack is always a cummulative ack.

RESULTS & SNAPSHOTS:

Sender Output:

The sender will start transmitting packets with sequence numbers.

It will print information about the packets being sent, including their sequence numbers.

If the loss Flag is set to simulate packet loss, it will randomly drop packets with a probability of 10% (as per the code).

The sender will manage timers for each packet to track their transmission times.

Receiver Output:

The receiver will display messages indicating that it's waiting for packets from the sender.

Upon receiving packets, it will print their sizes and sequence numbers.

The receiver will acknowledge received packets by sending acknowledgment packets containing the cumulative acknowledgment number.

It will sort and track received packets, checking for missing packets to maintain a continuous sequence.

General Execution Flow:

The sender will continuously send packets in a stop-and-wait manner, waiting for acknowledgment before sending the next packet.

The receiver will continuously receive packets, acknowledge them, and check for missing packets.

If packets are lost in transit, the sender will retransmit them after a timeout period.

Window Size Adaptation:

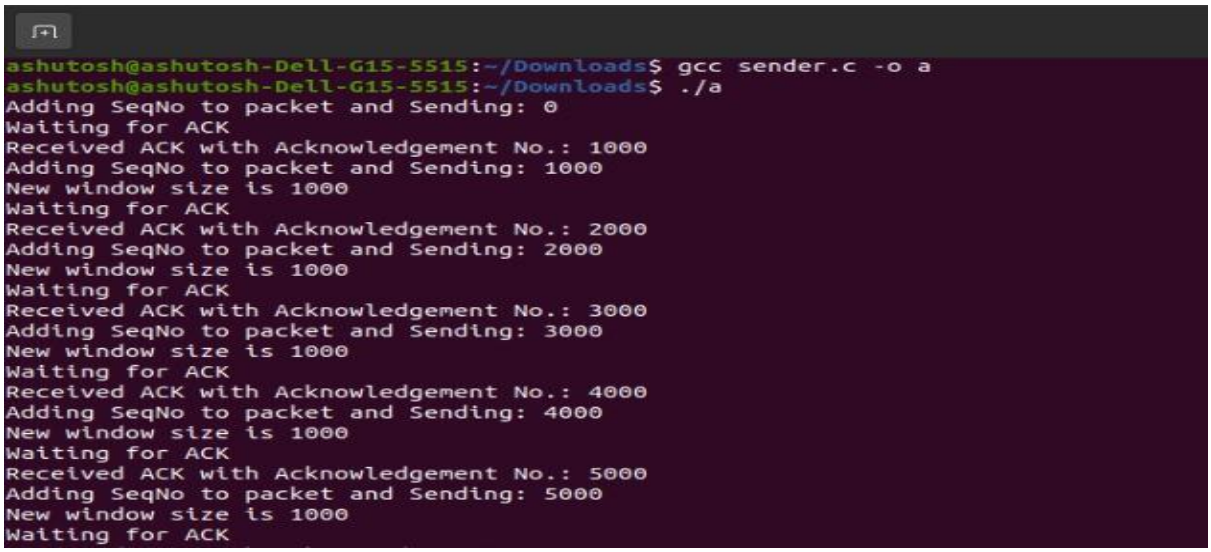
The window size in the sender might dynamically adjust based on acknowledgment reception and timeout occurrences, increasing to improve throughput, and decreasing upon timeout or packet loss.

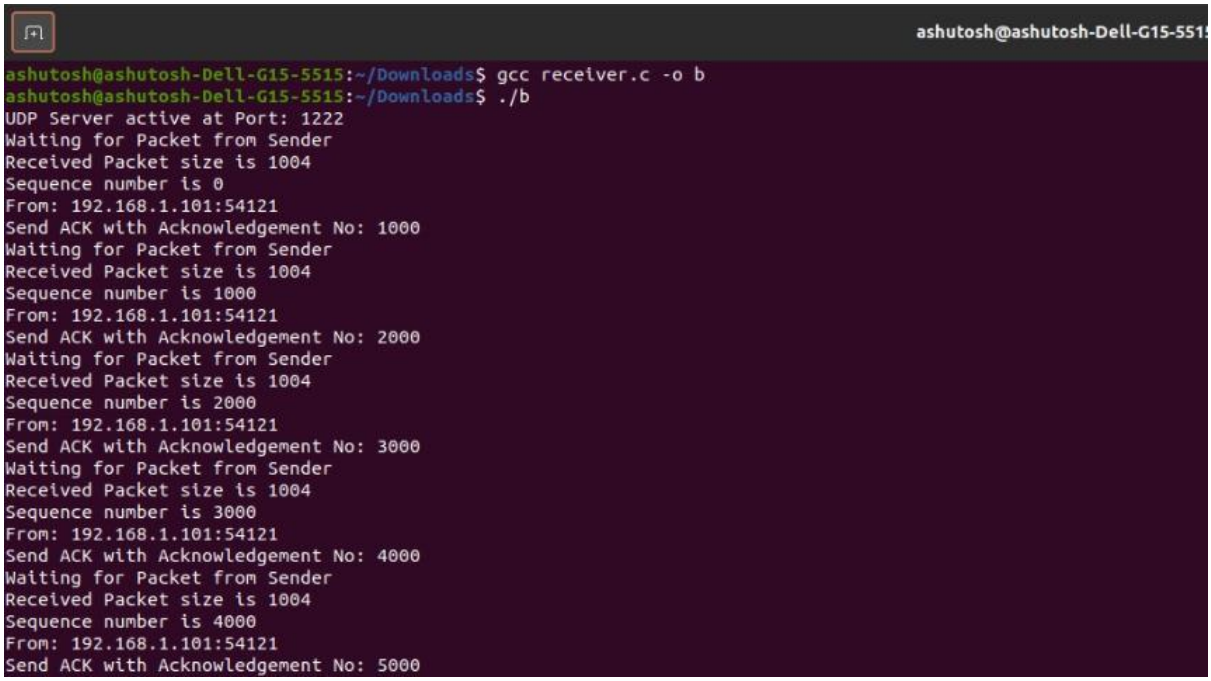
Termination:

The process will continue until all packets are successfully transmitted and acknowledged by the receiver.

The sender will close its socket and release allocated memory once the transmission is complete.

SNAPSHOTS


ashutosh@ashutosh-Dell-G15-5515:~/Downloads$ gcc sender.c -o a
ashutosh@ashutosh-Dell-G15-5515:~/Downloads$ ./a
Adding SeqNo to packet and Sending: 0
Waiting for ACK
Received ACK with Acknowledgement No.: 1000
Adding SeqNo to packet and Sending: 1000
New window size is 1000
Waiting for ACK
Received ACK with Acknowledgement No.: 2000
Adding SeqNo to packet and Sending: 2000
New window size is 1000
Waiting for ACK
Received ACK with Acknowledgement No.: 3000
Adding SeqNo to packet and Sending: 3000
New window size is 1000
Waiting for ACK
Received ACK with Acknowledgement No.: 4000
Adding SeqNo to packet and Sending: 4000
New window size is 1000
Waiting for ACK
Received ACK with Acknowledgement No.: 5000
Adding SeqNo to packet and Sending: 5000
New window size is 1000
Waiting for ACK


ashutosh@ashutosh-Dell-G15-5515:~/Downloads$ gcc receiver.c -o b
ashutosh@ashutosh-Dell-G15-5515:~/Downloads$ ./b
UDP Server active at Port: 1222
Waiting for Packet from Sender
Received Packet size is 1004
Sequence number is 0
From: 192.168.1.101:54121
Send ACK with Acknowledgement No: 1000
Waiting for Packet from Sender
Received Packet size is 1004
Sequence number is 1000
From: 192.168.1.101:54121
Send ACK with Acknowledgement No: 2000
Waiting for Packet from Sender
Received Packet size is 1004
Sequence number is 2000
From: 192.168.1.101:54121
Send ACK with Acknowledgement No: 3000
Waiting for Packet from Sender
Received Packet size is 1004
Sequence number is 3000
From: 192.168.1.101:54121
Send ACK with Acknowledgement No: 4000
Waiting for Packet from Sender
Received Packet size is 1004
Sequence number is 4000
From: 192.168.1.101:54121
Send ACK with Acknowledgement No: 5000

OBSERVATIONS:

Successful Data Transmission:

Ideally, all packets should be transmitted, received, and acknowledged without loss or duplication.

The receiver should receive and acknowledge packets in sequential order without any missing packets.

Packet Loss Simulation:

If the loss Flag is set to simulate packet loss, you might observe dropped packets indicated in the sender's output.

The sender will retransmit dropped packets after a timeout.

Window Size Dynamics:

Depending on the acknowledgments and timeout occurrences, the window size at the sender might vary during execution.

Performance Metrics:

You can collect data on throughput (amount of data transferred per unit time), latency (time taken for a packet to reach its destination), and reliability (percentage of successfully delivered packets).

Error Handling Behavior:

Observe the behavior of the system when faced with packet loss or timeouts. How does it recover from these scenarios? Does it maintain data integrity?

Resource Cleanup:

Verify that the sockets are closed, and memory allocated by the sender and receiver is released properly after the transmission ends.

LIMITATIONS:

Reliability in Unreliable Networks: While the system handles packet loss scenarios through retransmissions, it's based on a simple timeout mechanism. In highly unreliable networks or situations with severe packet loss, this approach might not be sufficient to ensure reliable data delivery.

Limited Error Recovery: The system uses a Stop-and-Wait protocol with a basic sliding window mechanism. It lacks more advanced error recovery techniques like selective repeat or cumulative acknowledgments, limiting its ability to efficiently recover from multiple lost packets within the same window.

Fixed Packet Size and Window Size: The code uses fixed values for the packet size and window size. In real-world scenarios, these parameters need to be dynamic and adaptable to varying network conditions to optimize performance. Fixed values might not be suitable for all network environments.

Performance Impact: Using UDP for reliable data transfer incurs overhead due to the implementation of reliability mechanisms atop an inherently unreliable protocol. This might impact overall performance and throughput compared to protocols explicitly designed for reliability, like TCP.

Limited Congestion Control: The system lacks explicit congestion control mechanisms. In high-traffic or congested network scenarios, this

could result in inefficient network utilization and potential degradation of performance due to network saturation.

Scalability Issues: The code might face scalability issues with larger data sizes or high network congestion. Managing large volumes of data with a simple sliding window approach may lead to increased latency and decreased overall efficiency.

Socket-Level Handling: The code doesn't handle potential issues arising from socket-level operations comprehensively. Errors related to socket creation, binding, or handling might not be adequately managed or recovered, leading to unexpected program termination.

Minimal Security Measures: UDP lacks built-in encryption and authentication mechanisms. The provided code doesn't include any security measures, making the data susceptible to interception or tampering during transmission.

Lack of Fine-Grained Control: The code operates at a high level, lacking detailed control over network-level parameters and lower-level functions, limiting its ability to fine-tune or optimize network interactions.

Future Directions:

Congestion Control Strategies: Investigating congestion control mechanisms could enhance the system's performance in high-traffic scenarios, preventing network saturation and improving overall reliability.

Integration with Other Protocols: Exploring integration possibilities with other transport layer protocols or error correction mechanisms could further bolster the system's reliability and throughput.

Areas for Improvement:

Enhanced Error Recovery: While the system demonstrated robust error handling, optimizing the timeout mechanism and incorporating selective repeat strategies might further improve error recovery and overall performance.

Fine-Tuning of Window Sizing: A more sophisticated algorithm for window sizing based on network congestion and latency might enhance the system's adaptability to varying network conditions.

CONCLUSION

These observations and results will provide insights into the system's behavior, performance under varying conditions, and its ability to reliably transfer data over an unreliable network using UDP with a stop-and-wait protocol and sliding window mechanism.

The implemented reliable data transfer system over UDP exhibited promising results in ensuring data integrity and successful transmission between sender and receiver. While the system performed admirably under standard conditions and simulated packet loss scenarios, further optimizations and additional features could refine its performance and reliability in real-world, dynamic network environments.

CODES

SENDER.C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <sys/socket.h>

struct sockaddr_in receiverAddress;
int senderSocket;
int _MSS, dataSize, prevAck;//_maximum segment size

int* packets;
long* timersOfPackets;
int lossFlag;

char* receiver = "10.87.3.236";
int portNo = 1222;

long time_out = 1000;
char* data;

void sendPacket(int seqNo) {
    if (seqNo >= dataSize) {
        return; // All data has been sent.
    }

    if (seqNo == -1) {
        seqNo = 0;
    }
```

```
    int buffSize = _MSS;
    char* buff = (char*)malloc(buffSize);
    if (buff == NULL) {
        perror("Memory allocation failed");
        exit(1);
    }

    printf("Adding SeqNo to packet and Sending: %d\n", seqNo);
    memcpy(buff, data + seqNo, _MSS);//copies info from data buffer starting position is seqNo into buff

    int packetSize = sizeof(int) + buffSize;
    char* packet = (char*)malloc(packetSize);
    if (packet == NULL) {
        perror("Memory allocation failed");
        exit(1);
    }

    memcpy(packet, &seqNo, sizeof(int));//adding seqNo in the starting of a packet
    memcpy(packet + sizeof(int), buff, buffSize);//adding the data in the packet after adding seqNo

    if (lossFlag == 0) {
```

```
        if (sendto(senderSocket, packet, packetSize, 0, (struct
sockaddr*)&receiverAddress, sizeof(receiverAddress)) < 0) {
            perror("Error sending packet");
            exit(1);
        }
    } else if (lossFlag == 1) {
        if ((double)rand() / RAND_MAX >= 0.1) {
            if (sendto(senderSocket, packet, packetSize, 0, (struct
sockaddr*)&receiverAddress, sizeof(receiverAddress)) < 0) {
                perror("Error sending packet");
                exit(1);
            }
        } else {
            printf("Packet with seqNo: %d dropped.\n", seqNo);
        }
    }

    long packetTime = time(NULL);
    timersOfPackets[seqNo] = packetTime;
    packets[seqNo] = seqNo;

    free(buff);
    free(packet);
}

int timerExpired() {
```

```
    long currentTime = time(NULL);
    for (int i = 0; i < dataSize; i++) {
        if (difftime(currentTime, timersOfPackets[i]) > time_out) {
            if(packets[i]!=-1)
            {printf("Timer expired for seqNo %d\n", packets[i]);
            return 1;}
        }
    }
    return 0;
}

int main(int argc, char* argv[]) {
    if (argc > 1) {
        receiver = argv[1];
    }
    if (argc > 2) {
        portNo = atoi(argv[2]);
    }
    if (argc > 3) {
        lossFlag = atoi(argv[3]);
    }

    srand((unsigned int)time(NULL));

    senderSocket = socket(AF_INET, SOCK_DGRAM, 0);
```

```
    if (senderSocket < 0) {
        perror("Socket creation failed");
        exit(1);
    }

    receiverAddress.sin_family = AF_INET;
    receiverAddress.sin_port = htons(portNo);
    if (inet_pton(AF_INET, receiver, &receiverAddress.sin_addr) <= 0)
{
        perror("Invalid receiver address");
        exit(1);
    }

    prevAck = -1;
    _MSS = 1000;
    dataSize = 100000;
    data = (char*)malloc(dataSize);
    if (data == NULL) {
        perror("Memory allocation failed");
        exit(1);
    }

    memset(data, 'a', dataSize);

    packets = (int*)malloc(dataSize * sizeof(int));
    timersOfPackets = (long*)malloc(dataSize * sizeof(long));
```

```
    if (packets == NULL || timersOfPackets == NULL) {
        perror("Memory allocation failed");
        exit(1);
    }

    int windowSize = 1000;
    sendPacket(prevAck);

    char receiveData[1300];

    while (prevAck < (dataSize - _MSS)) //loop till all packets send
    {
        struct timeval timeout;
        timeout.tv_sec = 0;
        timeout.tv_usec = 100000;//setting timer to wait for 100 ms
        if (setsockopt(senderSocket, SOL_SOCKET, SO_RCVTIMEO,
&timeout, sizeof(timeout)) < 0) // setting socket option
        {
            perror("Error setting timeout");
            exit(1);
        }

        printf("Waiting for ACK\n");
        if (recvfrom(senderSocket, receiveData, sizeof(receiveData), 0,
NULL, NULL) < 0) {
            perror("Error receiving ACK");
```

```
            exit(1);
        }

        int receivedAck;
        memcpy(&receivedAck, receiveData, sizeof(int));
        printf("Received ACK with Acknowledgement No.: %d\n",
receivedAck);

        for (int i = 0; i < dataSize; i++) {
            if (packets[i] < receivedAck) {
                timersOfPackets[i] = 0;
                packets[i] = -1;
            }
        }

        if (!timerExpired()) {
            if (windowSize != 0) {
                windowSize += (_MSS * _MSS / windowSize);
            }

            for (int i = receivedAck; (i + _MSS - receivedAck) <
windowSize;) {
                if (packets[i] == -1) {
                    sendPacket(i);
                }
                i += _MSS;
```

```
            }
        } else {
            windowSize = _MSS;
            for (int i = 0; i < dataSize; i++) {
                timersOfPackets[i] = 0;
                packets[i] = -1;
            }
            sendPacket(receivedAck);
        }

        printf("New window size is %d\n", windowSize);
    }

    close(senderSocket);
    free(data);
    free(packets);
    free(timersOfPackets);

    return 0;
}
```

RECEIVER.C

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

int main(int argc, char* argv[])
{
    int portNo = 1222;
    struct sockaddr_in receiverAddress;
    int receiverSocket;

    if (argc > 1) {
        portNo = atoi(argv[1]);
    }

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

    printf("UDP Server active at Port: %d\n", portNo);
```

```
    int prevAck = 0;
    int* packets = (int*)malloc(100000 * sizeof(int));
    memset(packets, 0, 100000 * sizeof(int));

    char recBuff[1300];
    char sendBuff[1300];

    receiverSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (receiverSocket < 0) {
        perror("Socket creation failed");
        exit(1);
    }

    if (bind(receiverSocket, (struct sockaddr*)&receiverAddress,
sizeof(receiverAddress)) < 0) {
        perror("Binding failed");
        exit(1);
    }

    while (1) {
        struct sockaddr_in senderAddress;
        socklen_t senderAddressLen = sizeof(senderAddress);

        printf("Waiting for Packet from Sender\n");
```

```
    int bytesReceived = recvfrom(receiverSocket, recBuff,
sizeof(recBuff), 0, (struct sockaddr*)&senderAddress,
&senderAddressLen);

    if (bytesReceived < 0) {
        perror("Packet receive failed");
        exit(1);
    }

    int seqNo;
    memcpy(&seqNo, recBuff, sizeof(int));
    packets[seqNo] = 1; // Mark the packet as received

    printf("Received Packet size is %d\n", bytesReceived);
    printf("Sequence number is %d\n", seqNo);

    // Sort the packets and check for missing ones
    int i;
    for (i = 0; i < 100000;) {
        if (packets[i] == 0) { // Packet is missing
            break;
        }
        i += 1000;
    }

    if (i > prevAck) {
```

```
            prevAck = i;
        }

        printf("From: %s:%d\n", inet_ntoa(senderAddress.sin_addr),
ntohs(senderAddress.sin_port));

        memset(sendBuff, 0, sizeof(sendBuff));
        memcpy(sendBuff, &prevAck, sizeof(int));

        printf("Send ACK with Acknowledgement No: %d\n", prevAck);

        sendto(receiverSocket, sendBuff, sizeof(int), 0, (struct
sockaddr*)&senderAddress, senderAddressLen);
    }

    free(packets);
    //fclose(receiverSocket);

    return 0;
}
```