LAN-Based File Transfer using Sliding Window Protocol

MINI PROJECT REPORT

Submitted by

JACINTH MANUEL J (9517202309039)

MANOJ KUMAR S (9517202309068)

RAGAVAN R (9517202309092)

in

23AD551 – COMPUTER NETWORKING LABORATORY

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

MEPCO SCHLENK ENGINEERING COLLEGE

SIVAKASI

NOVEMBER 2025

4 QUALITY EDUCATION

8 DECENT WORK AND ECONOMIC GROWTH

9 INDUSTRY, INNOVATION AND INFRASTRUCTURE

12 RESPONSIBLE CONSUMPTION AND PRODUCTION

MEPCO SCHLENK ENGINEERING COLLEGE

(AUTONOMOUS), SIVAKASI

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

BONAFIDE CERTIFICATE

This is to certify that it is the bonafide work of **JACINTH MANUEL J (9517202309039), MANOJ KUMAR S (9517202309068), RAGAVAN R (9517202309092)** for the mini project titled **LAN-BASED FILE TRANSFER USING SLIDING WINDOW PROTOCOL** in 23AD551 – Computer Networking Laboratory during the fifth semester July 2025 – November 2025 under my supervision.

SIGNATURE	**SIGNATURE**
Mrs P.Priyadharshini	**Dr. J. Angela Jennifa Sujana,**
Asst. Professor,	**Professor & Head,**
AI&DS Department,	AI&DS Department
Mepco Schlenk Engg. College, Sivakasi	Mepco Schlenk Engg. College, Sivakasi

ABSTRACT

The objective of this project is to develop a LAN-based file transfer system implementing the sliding window protocol to ensure reliable and efficient data transmission. The system comprises two main components: a graphical user interface (GUI) frontend for ease of use, and a backend peer application responsible for the file transfer process over TCP sockets.The frontend, implemented in Java Swing, offers users an intuitive interface to select files, specify the target IP address within the LAN, and define the sliding window size. It manages peer process initiation and communicates commands to the backend via standard input, enhancing user control and interaction.The backend, written in C for Windows compatibility, handles the core file transfer logic using the sliding window protocol to improve data throughput and reliability over the inherently unreliable TCP transport. The protocol segments files into fixed-size packets, each tagged with sequence numbers. The sender maintains a sliding window defining the range of packets that can be sent but not yet acknowledged. Acknowledgment packets from the receiver enable the sender to slide the window forward, retransmitting any lost or corrupted packets upon timeout. This approach minimizes retransmissions and maximizes network utilization.The receiver listens for incoming packets, reassembles the data in sequence, and writes the output to the local file system, acknowledging receipt of packets to the sender. Special control packets are used to communicate file metadata such as filenames, allowing dynamic file creation.Through synchronization mechanisms and careful handling of packet sequencing and acknowledgments, the system ensures data integrity and robustness against packet loss, duplication, or reordering common in TCP-based networks. Experimental tests demonstrate the system's ability to reliably transfer files over a LAN with configurable window sizes, balancing throughput and latency.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

In the modern era of digital communication, efficient and reliable data transfer across computer networks plays a vital role in information exchange. Within a Local Area Network (LAN), multiple systems are interconnected to share resources, files, and services. However, existing file transfer methods, such as FTP or direct TCP connections, often lack control over packet delivery, retransmission, and acknowledgment management—especially in custom, lightweight applications.

This project focuses on designing and developing a LAN-based File Transfer Application that ensures reliable and bidirectional file transmission between systems connected within the same local network. The core data transfer logic is implemented in the C programming language, which handles network-level operations using the Transmission control Protocol (TCP) combined with the Sliding Window Protocol (Go-Back-N) to manage packet loss, flow control, and retransmission efficiently.

To enhance usability, the project integrates a Graphical User Interface (GUI) developed in Java, allowing users to easily select files, specify target IP addresses, and monitor the transfer process. The GUI communicates with the backend C module (peer.exe) to initiate and control file transfers between peers. Each peer in the system functions as both a sender and receiver, enabling two-way file exchange over the LAN.

The system demonstrates how network programming, protocol design, and user interface development can be combined to create a robust, efficient, and user-friendly file transfer solution. It serves as an educational implementation of reliable data transfer concepts in computer networks while also providing practical utility for local file sharing environments.

1.2 Objectives

The main objective of the LAN Based File Transfer Using Sliding Window Protocol project are as follow:

- Develop a LAN-based communication system using the Transmission control Protocol (TCP) for fast, lightweight data transfer between connected systems.
- Implement the Sliding Window Protocol (Go-Back-N) in C to ensure reliable packet delivery, manage retransmissions, and control flow during transmission.
- Enable bidirectional file transfer, allowing each peer to act as both a sender and receiver.
- Design a Java-based Graphical User Interface (GUI) that allows users to select files, enter IP addresses, and monitor transfer status easily.
- Integrate the GUI with the C backend to execute and control file transfers seamlessly between peers.
- Ensure error detection and recovery through acknowledgments and retransmission mechanisms.
- Demonstrate core concepts of computer networking such as socket programming, flow control, and reliable data transfer in real-world LAN environments.

1.3 Scope of project

This project focuses on designing and implementing a reliable file transfer system over a Local Area Network (LAN) utilizing the sliding window protocol to enhance data transmission efficiency and integrity. The scope includes:

- Developing a user-friendly graphical interface that allows users to select files, specify target IP addresses, and configure the sliding window size for customizable transfer control.
- Implementing the sliding window protocol in the backend to manage packet sequencing, acknowledgment, retransmission on timeout, and flow control, ensuring reliable and ordered file delivery over TCP.
- Supporting file transfer within a LAN environment, targeting medium-scale networks where direct IP communication is feasible.
- Enabling transfer of files of varying sizes by segmenting data into fixed-size packets and reassembling them correctly at the receiver’s end.

- Providing basic error handling, including packet loss detection and retransmission, to maintain data integrity.
- Allowing configuration of parameters such as window size and listen port to adapt to different network conditions and performance requirements.

1.4 Protocols Used

The LAN Based File Transfer using Sliding Window Protocol project primarily uses two key protocols — TCP (Transmission Control Protocol) and Sliding Window Protocol — which work together to enable reliable communication and file transfer between the sender and the receiver.

1. Transmission Control Protocol (TCP)

Type: Transport layer protocol (Connection-oriented)

Purpose:TCP provides reliable, ordered, and error-checked delivery of a stream of between applications running on hosts communicating over an IP network.

Function in the Project:

TCP inherently manage connection setup, segmentation, flow control, error recovery, and retransmission.

TCP ensures that data packets arrive intact and in order, which is critical for file transfers to prevent corruption or data loss.

2.Sliding Window Protocol

Type: Flow control and error control protocol

Purpose: The sliding window protocol regulates the flow of data packets between sender and receiver, allowing multiple packets to be sent before requiring an acknowledgment.

Function in the Project:

In your project, the sliding window protocol is implemented at the application layer over TCP to provide reliable data transfer. It manages:

Each data packet is numbered to keep track of the order. Limits the number of unacknowledged packets to the window size, preventing sender overload. The receiver sends acknowledgments for received packets, allowing the sender to slide the window forward.

Packets not acknowledged within a timeout period are retransmitted to handle loss.

CHAPTER 2
IMPLEMENTATION

2.1 System design

The system is designed as a client-server-inspired peer-to-peer file transfer application operating over a Local Area Network (LAN) with an emphasis on reliable data transmission using the sliding window protocol. It consists of two main components: a graphical user interface (GUI) frontend and a backend peer process. The frontend, developed using Java Swing, provides an interactive environment for users to select files, specify the target IP address, and configure the sliding window size for flow control. The backend, implemented in C, handles the actual data transmission over TCP sockets. It segments files into packets, transmits them sequentially while maintaining a sliding window of unacknowledged packets, and listens for acknowledgments from the receiver to manage retransmissions and ensure data integrity.

The file transfer architecture uses sliding window protocol for transfer file and TCP for communication.The following steps summarize the data flow in the system:

1. User Input via Frontend GUI:
   The user selects a file to transfer using the file chooser dialog.The user inputs the target IP address and optionally specifies the window size.When the user clicks "Send File," the frontend formulates a command containing the target IP, file path, and window size.
2. Command Transmission to Backend Peer Process:
   The frontend sends the command string to the backend peer process's standard input stream.
   The backend process reads this command asynchronously to initiate the file transfer.
3. File Reading and Packetization in Backend:
   The backend opens the specified file and reads it sequentially.
   The file is divided into fixed-size packets, each tagged with a sequence number and metadata (size, end-of-file flag).
   A control packet containing the filename is prepared and sent first to the receiver.
4. Packet Sending with Sliding Window Control:
   The sender maintains a sliding window that controls how many packets can be sent without acknowledgment.

Packets within the window are transmitted over UDP sockets to the receiver's IP and port.
Each sent packet's send time is recorded for timeout tracking.

5. Packet Reception and File Reconstruction:
   The receiver continuously listens on its UDP socket for incoming packets.
   Upon receiving a control packet, it extracts the filename and opens/creates the file for writing.
   Data packets are checked for sequence order; in-sequence packets are written to the file, and out-of-sequence packets are discarded or buffered depending on implementation.
6. Acknowledgment Packet Generation:
   The receiver sends acknowledgment packets (ACKs) back to the sender, indicating the highest contiguous sequence number received.
   ACKs are sent over UDP to the sender's listening port.
7. Acknowledgment Reception and Window Sliding:
   The sender listens for incoming ACK packets.
   Upon receiving an ACK, the sender slides the window forward, allowing new packets to be sent.
   This process repeats until all packets are acknowledged.
8. Timeout and Retransmission Handling:
   The sender monitors the send times of unacknowledged packets.
   If an acknowledgment is not received within a specified timeout period, the sender retransmits all packets in the current window.
   Retransmissions continue until packets are acknowledged or a retry limit is reached.
9. Transfer Completion and Cleanup:
   When the sender receives acknowledgment for all packets, it considers the file transfer complete.
   The receiver closes the file after receiving the final packet with the end-of-file flag.
   Both sender and receiver release resources and prepare for the next transfer.

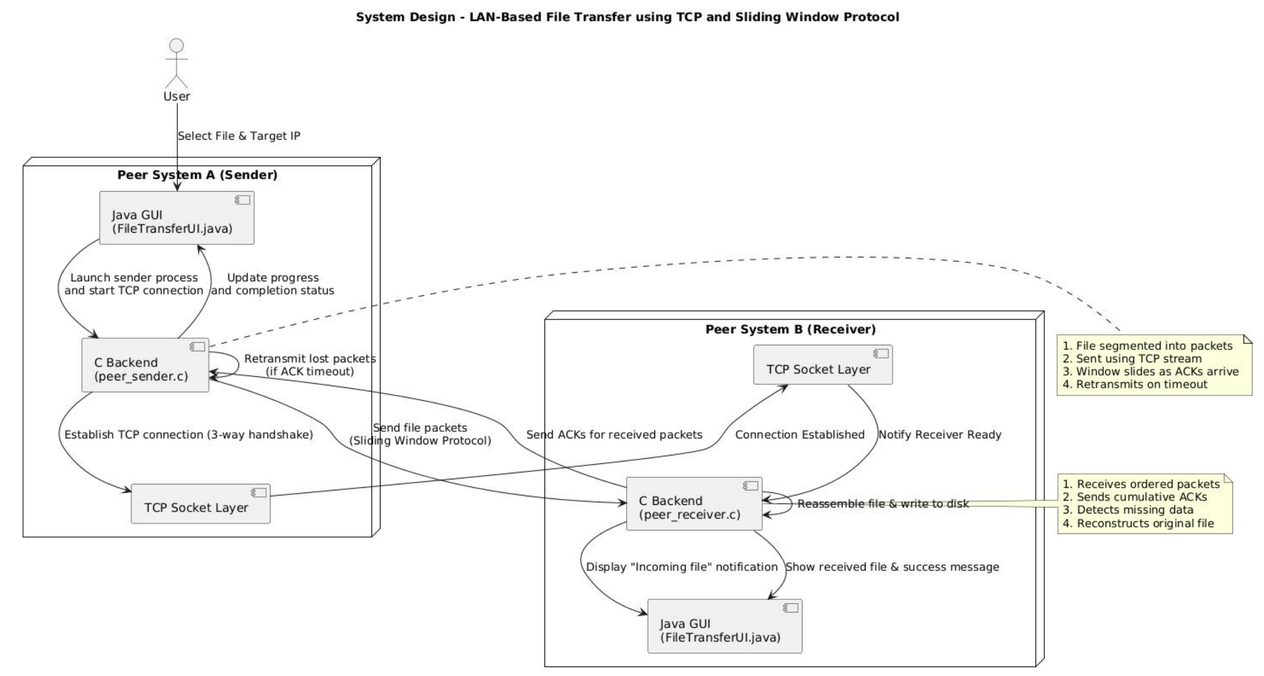

System Design - LAN-Based File Transfer using TCP and Sliding Window Protocol
User
Select File & Target IP
Peer System A (Sender)
Java GUI (FileTransferUI.java)
Launch sender process and start TCP connection
Update progress and completion status
C Backend (peer_sender.c)
Retransmit lost packets (if ACK timeout)
Establish TCP connection (3-way handshake)
TCP Socket Layer
Send file packets (Sliding Window Protocol)
Send ACKs for received packets
Connection Established
Notify Receiver Ready
Peer System B (Receiver)
TCP Socket Layer
C Backend (peer_receiver.c)
Reassemble file & write to disk
Display "Incoming file" notification
Show received file & success message
Java GUI (FileTransferUI.java)
1. File segmented into packets
2. Sent using TCP stream
3. Window slides as ACKs arrive
4. Retransmits on timeout
1. Receives ordered packets
2. Sends cumulative ACKs
3. Detects missing data
4. Reconstructs original file

Fig 2.1 System Design

2.2 Program coding

peer.c(main code):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <stdint.h>
#include <time.h>
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>

#pragma comment(lib, "ws2_32.lib")
```

```
#define DEFAULT_PORT 5004
#define PACKET_DATA 1000
#define HEADER_SIZE (sizeof(int32_t)*3)
#define MAX_WINDOW 64
#define TIMEOUT_MS 2000

typedef struct {
    int32_t seq;
    int32_t size;
    int32_t eof;
    char data[PACKET_DATA];
} Packet;

typedef struct {
    int32_t ack_seq;
} AckPacket;

int listen_port = DEFAULT_PORT;
HANDLE print_mutex;

long long now_ms() {
    SYSTEMTIME st;
    GetSystemTime(&st);
    FILETIME ft;
    SystemTimeToFileTime(&st, &ft);
    ULARGE_INTEGER uli;
    uli.LowPart = ft.dwLowDateTime;
    uli.HighPart = ft.dwHighDateTime;
    return (uli.QuadPart / 10000ULL);
}
```

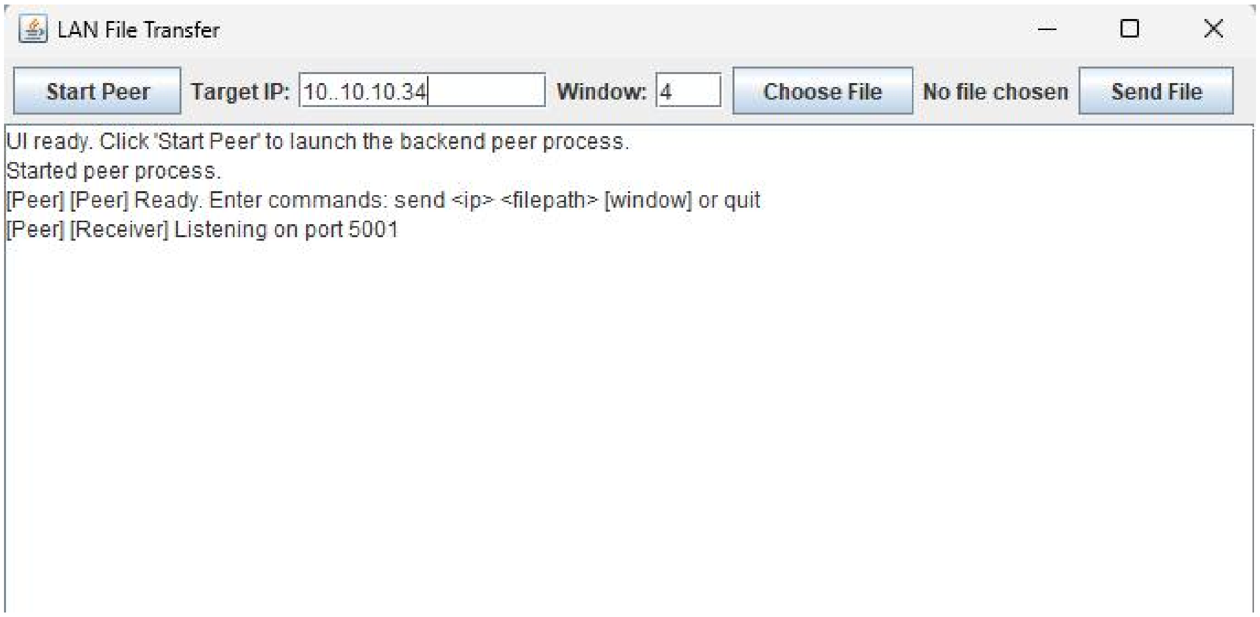
```
void safe_printf(const char *fmt, ...) {
   WaitForSingleObject(print_mutex, INFINITE);
   va_list ap;
   va_start(ap, fmt);
   vprintf(fmt, ap);
   printf("\n");
   fflush(stdout);
   va_end(ap);
   ReleaseMutex(print_mutex);
}

DWORD WINAPI receiver_thread_func(LPVOID arg) {
   SOCKET sockfd;
   struct sockaddr_in servaddr, cliaddr;
   int len = sizeof(cliaddr);
   Packet pkt;
   char current_filename[512] = {0};
   FILE *fp = NULL;
   int32_t expected_seq = 0;
   int n;

   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET) {
      safe_printf("[Receiver] socket() failed: %ld", WSAGetLastError());
      return 1;
   }

   memset(&servaddr, 0, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr = INADDR_ANY;
   servaddr.sin_port = htons(listen_port);
```

```
    int reuse = 1;
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (const char *)&reuse,
sizeof(reuse)) < 0) {
        safe_printf("[Receiver] Warning: setsockopt(SO_REUSEADDR) failed");
    }

    if (bind(sockfd, (struct sockaddr *)&servaddr, sizeof(servaddr)) == SOCKET_ERROR) {
        safe_printf("[Receiver] bind() failed: %ld", WSAGetLastError());
        closesocket(sockfd);
        return 1;
    }

    safe_printf("[Receiver] Listening on port %d", listen_port);

    while (1) {
        n = recvfrom(sockfd, (char *)&pkt, sizeof(pkt), 0, (struct sockaddr *)&cliaddr, &len);
        if (n < 0) {
            safe_printf("[Receiver] recvfrom error: %ld", WSAGetLastError());
            continue;
        }

        int32_t seq = ntohl(pkt.seq);
        int32_t size = ntohl(pkt.size);
        int32_t eof = ntohl(pkt.eof);

        if (seq == -1) { // Control packet: filename
            memset(current_filename, 0, sizeof(current_filename));
            snprintf(current_filename, sizeof(current_filename) - 1, "%s", pkt.data);
            if (fp) { fclose(fp); fp = NULL; }
            fp = fopen(current_filename, "wb");
            if (!fp) {
                safe_printf("[Receiver] Cannot open '%s' for writing", current_filename);
            } else {
                safe_printf("[Receiver] Receiving file: %s", current_filename);
                expected_seq = 0;
            }
            AckPacket ack;
            ack.ack_seq = htonl(expected_seq - 1);
            sendto(sockfd, (char *)&ack, sizeof(ack), 0, (struct sockaddr *)&cliaddr, len);
            continue;
        }

        safe_printf("[Receiver] pkt seq=%d size=%d eof=%d expected=%d", seq, size, eof,
expected_seq);

        if (!fp) {
            safe_printf("[Receiver] No file open yet");
        } else {
            if (seq == expected_seq) {
                if (size > 0) fwrite(pkt.data, 1, size, fp);
                expected_seq++;
            } else {
                safe_printf("[Receiver] out-of-order pkt %d (expected %d)", seq, expected_seq);
            }

            AckPacket ack;
            ack.ack_seq = htonl(expected_seq - 1);
            sendto(sockfd, (char *)&ack, sizeof(ack), 0, (struct sockaddr *)&cliaddr, len);

            if (eof && seq == expected_seq - 1) {
                safe_printf("[Receiver] File '%s' complete.", current_filename);
                fclose(fp); fp = NULL;
                expected_seq = 0;
            }
        }
    }

    closesocket(sockfd);
    return 0;
}

int send_file(const char *target_ip, const char *filepath, int WINDOW) {
    if (WINDOW <= 0) WINDOW = 4;
    if (WINDOW > MAX_WINDOW) WINDOW = MAX_WINDOW;

    FILE *fp = fopen(filepath, "rb");
    if (!fp) {
        safe_printf("[Sender] Failed to open '%s'", filepath);
        return 1;
    }

    SOCKET sockfd;
    struct sockaddr_in servaddr;
    int addrlen = sizeof(servaddr);

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET) {
        safe_printf("[Sender] socket() failed");
        fclose(fp);
        return 1;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(listen_port);

    //  Fixed: replaced inet_pton() / InetPtonA() with inet_addr()
    servaddr.sin_addr.s_addr = inet_addr(target_ip);
    if (servaddr.sin_addr.s_addr == INADDR_NONE) {
        safe_printf("[Sender] invalid target IP: %s", target_ip);
        fclose(fp);
        closesocket(sockfd);
        return 1;
    }

    Packet *packets = NULL;
    size_t total_pkts = 0;
    while (!feof(fp)) {
        Packet p;
        memset(&p, 0, sizeof(p));
        p.seq = htonl((int32_t)total_pkts);
        size_t nread = fread(p.data, 1, PACKET_DATA, fp);
        p.size = htonl((int32_t)nread);
        p.eof = htonl(nread < PACKET_DATA ? 1 : 0);
        packets = realloc(packets, sizeof(Packet) * (total_pkts + 1));
        packets[total_pkts] = p;
        total_pkts++;
        if (nread < PACKET_DATA) break;
    }
    fclose(fp);
    safe_printf("[Sender] File '%s' split into %zu packets. Window=%d", filepath, total_pkts,
WINDOW);

    Packet ctrl;
    memset(&ctrl, 0, sizeof(ctrl));
    ctrl.seq = htonl(-1);
    const char *fname = strrchr(filepath, '\\');
    if (!fname) fname = strrchr(filepath, '/');
    if (fname) fname++; else fname = filepath;
    strncpy(ctrl.data, fname, PACKET_DATA - 1);
    ctrl.size = htonl((int32_t)strlen(ctrl.data));
    ctrl.eof = htonl(0);
    sendto(sockfd, (char *)&ctrl, sizeof(Packet), 0, (struct sockaddr *)&servaddr, addrlen);
    safe_printf("[Sender] Sent control packet (filename=%s)", ctrl.data);

    int base = 0, nextseq = 0, finished = 0;
    long long send_time[MAX_WINDOW];
    memset(send_time, 0, sizeof(send_time));
    AckPacket ack;
    fd_set readfds;
    struct timeval tv;

    int retry_count = 0;
    const int MAX_RETRIES = 10;   // stop after 10 failed timeouts

  while (!finished) {
    while (nextseq < (int)total_pkts && nextseq < base + WINDOW) {
      Packet *p_net = &packets[nextseq];
      sendto(sockfd, (char *)p_net, sizeof(Packet), 0, (struct sockaddr *)&servaddr, addrlen);
      safe_printf("[Sender] Sent pkt seq=%d", ntohl(p_net->seq));
      send_time[nextseq % WINDOW] = now_ms();
      nextseq++;
    }

    FD_ZERO(&readfds);
```

```
FD_SET(sockfd, &readfds);
tv.tv_sec = 0;
tv.tv_usec = 100 * 1000;
int rv = select(0, &readfds, NULL, NULL, &tv);
if (rv > 0 && FD_ISSET(sockfd, &readfds)) {
    int n = recvfrom(sockfd, (char *)&ack, sizeof(ack), 0, NULL, NULL);
    if (n > 0) {
        int ack_seq = ntohl(ack.ack_seq);
        safe_printf("[Sender] ACK received: %d", ack_seq);
        if (ack_seq >= base) {
            base = ack_seq + 1;
            retry_count = 0;  // reset retry counter when we get an ACK
        }
        if (base >= (int)total_pkts) finished = 1;
    }
}

if (base < nextseq) {
    long long elapsed = now_ms() - send_time[base % WINDOW];
    if (elapsed >= TIMEOUT_MS) {
        retry_count++;
        safe_printf("[Sender] Timeout #%d on packet %d (base=%d)", retry_count, base, base);
        if (retry_count > MAX_RETRIES) {
            safe_printf("[Sender] Too many retries. Transfer aborted.");
            free(packets);
            closesocket(sockfd);
            return 0;
            //break;  // stop infinite loop
        }

        for (int i = base; i < nextseq; i++) {
                Packet *p_net = &packets[i];
                sendto(sockfd, (char *)p_net, sizeof(Packet), 0, (struct sockaddr *)&servaddr, addrlen);
                safe_printf("[Sender] Re-Sent pkt %d", ntohl(p_net->seq));
                send_time[i % WINDOW] = now_ms();
            }
        }
    }
}

    safe_printf("[Sender] File transfer complete.");
    free(packets);
    closesocket(sockfd);
    return 0;
}

int main(int argc, char *argv[]) {
    WSADATA wsa;
    if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {
        printf("WSAStartup failed!\n");
        return 1;
    }

    if (argc >= 2) {
        listen_port = atoi(argv[1]);
        if (listen_port <= 0) listen_port = DEFAULT_PORT;
    }

    print_mutex = CreateMutex(NULL, FALSE, NULL);
    HANDLE recv_thread = CreateThread(NULL, 0, receiver_thread_func, NULL, 0, NULL);

    safe_printf("[Peer] Ready. Enter commands: send <ip> <filepath> [window] or quit");
```

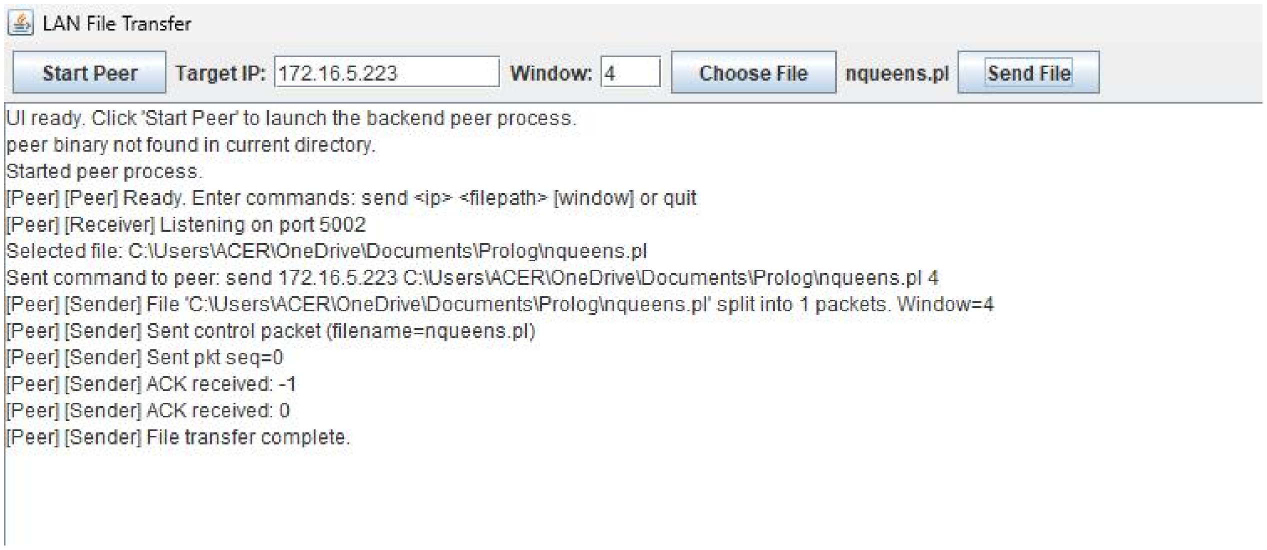
```
char line[1024];
while (fgets(line, sizeof(line), stdin)) {
    char *nl = strchr(line, '\n'); if (nl) *nl = '\0';
    if (strncmp(line, "quit", 4) == 0) break;
    if (strncmp(line, "send ", 5) == 0) {
        char target[64], path[512]; int window = 4;
        int parsed = sscanf(line + 5, "%63s %511s %d", target, path, &window);
        if (parsed < 2) {
            safe_printf("[Peer] Usage: send <ip> <filepath> [window]");
            continue;
        }
        send_file(target, path, window);
    }
}
TerminateThread(recv_thread, 0);
CloseHandle(recv_thread);
CloseHandle(print_mutex);
WSACleanup();
return 0;
}
```

2.3 Outputs and screenshots



```
UI ready. Click 'Start Peer' to launch the backend peer process.
Started peer process.
[Peer] [Peer] Ready. Enter commands: send <ip> <filepath> [window] or quit
[Peer] [Receiver] Listening on port 5001
```

Fig 2.2 Home page



```
UI ready. Click 'Start Peer' to launch the backend peer process.
peer binary not found in current directory.
Started peer process.
[Peer] [Peer] Ready. Enter commands: send <ip> <filepath> [window] or quit
[Peer] [Receiver] Listening on port 5002
Selected file: C:\Users\ACER\OneDrive\Documents\Prolog\nqueens.pl
Sent command to peer: send 172.16.5.223 C:\Users\ACER\OneDrive\Documents\Prolog\nqueens.pl 4
[Peer] [Sender] File 'C:\Users\ACER\OneDrive\Documents\Prolog\nqueens.pl' split into 1 packets. Window=4
[Peer] [Sender] Sent control packet (filename=nqueens.pl)
[Peer] [Sender] Sent pkt seq=0
[Peer] [Sender] ACK received: -1
[Peer] [Sender] ACK received: 0
[Peer] [Sender] File transfer complete.
```

Fig 2.3 File transfer

CHAPTER 3
CONCLUSION

The LAN-based file transfer project developed using the sliding window protocol successfully demonstrates the implementation of a reliable data communication system over UDP. By combining a Java-based graphical user interface (GUI) with a C-language backend peer process, the project effectively bridges user interactions with low-level network operations, a seamless and efficient file transfer experience within a local area network environment.

One of the key achievements in this project is the use of the sliding window protocol, which plays a crucial role in enhancing data transmission efficiency and reliability. Unlike simple stop-and-wait protocols, the sliding window approach allows multiple packets to be in transit before requiring acknowledgment, significantly improving throughput and reducing latency. This protocol also helps manage packet loss and reordering by implementing timeout-based retransmissions and acknowledgment tracking, which ensures that all data packets eventually reach the receiver intact and in the correct sequence.

The project involved careful consideration of the challenges inherent in UDP communication, such as its lack of built-in reliability and ordering guarantees. Through the design of custom packet structures, segmentation of large files into manageable chunks, and explicit acknowledgment mechanisms, these challenges were effectively mitigated. The retransmission strategy implemented ensures resilience against packet loss, a common issue in network communication, thereby safeguarding data integrity throughout the transfer process.

User experience was a major focus in this project, addressed by the development of a clear and intuitive GUI that enables users to select files, initiate transfers, and monitor progress with real-time feedback. This interface abstracts the underlying network complexities, making the system accessible even to users without technical expertise. Moreover, detailed logging and status updates help users understand the transfer state, including any errors or retransmissions, thereby improving transparency and trust in the system.

APPENDIX

Frontend code

PeerUI.java

```
// FileTransferUI.java
// Compile: javac FileTransferUI.java
// Run: java FileTransferUI
// Ensure the compiled 'peer' binary is in same folder and executable (chmod +x peer)

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.nio.file.Files;

public class PeerUI extends JFrame {
    private JTextArea logArea;
    private JButton startPeerBtn;
    private JButton chooseFileBtn;
    private JButton sendBtn;
    private JTextField targetIpField;
    private JTextField windowField;
    private JLabel selectedFileLabel;
    private Process peerProcess;
    private File selectedFile;
    private BufferedWriter peerWriter;

    public PeerUI() {
        super("LAN File Transfer");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(700, 500);
        setLayout(new BorderLayout());

        logArea = new JTextArea();
        logArea.setEditable(false);
        JScrollPane sp = new JScrollPane(logArea);
        add(sp, BorderLayout.CENTER);

        JPanel top = new JPanel(new FlowLayout(FlowLayout.LEFT));
        startPeerBtn = new JButton("Start Peer");
        top.add(startPeerBtn);
        top.add(new JLabel("Target IP:"));
        targetIpField = new JTextField(12);
        top.add(targetIpField);
        top.add(new JLabel("Window:"));
        windowField = new JTextField("4", 3);
        top.add(windowField);
        chooseFileBtn = new JButton("Choose File");
        top.add(chooseFileBtn);
        selectedFileLabel = new JLabel("No file chosen");
        top.add(selectedFileLabel);
        sendBtn = new JButton("Send File");
        top.add(sendBtn);

        add(top, BorderLayout.NORTH);

        startPeerBtn.addActionListener(e -> startPeer());
        chooseFileBtn.addActionListener(e -> chooseFile());
        sendBtn.addActionListener(e -> sendFile());

        appendLog("UI ready. Click 'Start Peer' to launch the backend peer process.");
    }
    private void appendLog(String s) {
        SwingUtilities.invokeLater(() -> {
            logArea.append(s + "\n");
            logArea.setCaretPosition(logArea.getDocument().getLength());
        });
    }

    private void startPeer() {
        if (peerProcess != null && peerProcess.isAlive()) {
            appendLog("Peer already running.");
            return;
        }
        try {
            if (!Files.exists(new File("peer.exe").toPath())) {
                appendLog("peer binary not found in current directory.");
                return;
            }
            ProcessBuilder pb = new
ProcessBuilder("C:\\Users\\ACER\\Downloads\\Ragavan\\LANFileTransfer\\backend\\peer.exe")
;
            pb.redirectErrorStream(true);
            peerProcess = pb.start();
            appendLog("Started peer process.");
            peerWriter = new BufferedWriter(new
OutputStreamWriter(peerProcess.getOutputStream()));

            // thread to read peer stdout
            new Thread(() -> {
                try (BufferedReader br = new BufferedReader(new
InputStreamReader(peerProcess.getInputStream()))) {
```

```
                String line;
                while ((line = br.readLine()) != null) {
                    appendLog("[Peer] " + line);
                }
            } catch (IOException ex) {
                appendLog("Error reading peer output: " + ex.getMessage());
            }
        }).start();

    } catch (IOException ex) {
        appendLog("Failed to start peer: " + ex.getMessage());
    }
}

private void chooseFile() {
    JFileChooser fc = new JFileChooser();
    if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
        selectedFile = fc.getSelectedFile();
        selectedFileLabel.setText(selectedFile.getAbsolutePath());
        appendLog("Selected file: " + selectedFile.getAbsolutePath());
    }
}

private void sendFile() {
    if (peerProcess == null || !peerProcess.isAlive()) {
        appendLog("Peer is not running. Click 'Start Peer' first.");
        return;
    }
    if (selectedFile == null) {
        appendLog("Choose a file first.");
        return;
    }
    String targetIp = targetIpField.getText().trim();
    if (targetIp.isEmpty()) {
        appendLog("Enter target IP.");
        return;
    }
    String windowStr = windowField.getText().trim();
    int window = 4;
    try { window = Integer.parseInt(windowStr); } catch (Exception ex) { window = 4; }

    // send command to peer's stdin: send <ip> <filepath> <window>
    String cmd = String.format("send %s %s %d\n", targetIp, selectedFile.getAbsolutePath(),
window);
    try {
        peerWriter.write(cmd);
        peerWriter.flush();
        appendLog("Sent command to peer: " + cmd.trim());
    } catch (IOException ex) {
        appendLog("Failed to write to peer stdin: " + ex.getMessage());
    }
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        PeerUI ui = new PeerUI();
        ui.setVisible(true);
    });
}}
```

REFERENCES

[1] W. R. Stevens, UNIX Network Programming: The Sockets Networking API, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2004.

[2] B. A. Forouzan, Data Communications and Networking, 5th ed. New York, NY, USA: McGraw-Hill Education, 2012.

[3] A. S. Tanenbaum and D. J. Wetherall, Computer Networks, 5th ed. Boston, MA, USA: Pearson Education, 2011.

[4] D. E. Comer, Internetworking with TCP/IP: Principles, Protocols, and Architecture, 6th ed. Boston, MA, USA: Pearson Education, 2013.

[5] J. F. Kurose and K. W. Ross, Computer Networking: A Top-Down Approach, 8th ed. Boston, MA, USA: Pearson, 2020.

[6] DataFlair, "Sliding Window Protocol-Working and Types"[Online]. Available: https://data-flair.training/blogs/sliding-window-protocol//

[7] TutorialsPoint, "TCP/IP Protocol Suite – Overview and Communication," [Online]. Available:Https://www.tutorialspoint.com/data_communication_computer_network/tcp_ip_protocol_suite.htm

[8] Microsoft Developer Network (MSDN), "Windows Sockets 2 (Winsock) Reference," [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/winsock/