University Of Science And Technology Of Hanoi
ICT Department

REPORT

PEER-TO-PEER File Transfer

Subject: Distributed Systems
Group ID: 1
Members: Trần Tuấn Vũ - 23BI14459
Nguyễn Việt Khoa - 23BI14223
Nguyễn Xuân Đức - 23BI14104
Nguyễn Chí Tôn - 23BI14424
Nguyễn Thế Xuân - 23BI14466

Contents

1 Project Overview

1.1 What It Brings to End Users

This Peer-to-Peer (P2P) file transfer application provides end users with:

- **Direct File Sharing:** Users can transfer files directly between two computers without requiring a centralized server or cloud storage service.
- **Privacy and Control:** Files are transmitted directly over the local network, ensuring that sensitive data never passes through third-party servers.
- **Simplicity:** The application uses a straightforward command-line interface that requires minimal setup and no complex configuration.
- **Cost-Effective:** No subscription fees or storage limits - users can transfer files of any size within their network capacity.
- **Educational Value:** Serves as a learning tool for understanding distributed systems, network programming, and TCP/IP protocols.

1.2 Project Description

This project implements a simple peer-to-peer (P2P) file transfer application using C++ and TCP/IP sockets. Two peers communicate directly over a network without a centralized server. One peer acts as a receiver (listening for incoming connections), while the other acts as a sender (initiating the connection and transferring a file).

The project is designed for academic purposes (midterm group project) and can be executed on:

- A single machine (for testing purposes)
- Two different machines within the same Local Area Network (LAN)

1.3 Key Features

- TCP/IP socket-based communication
- Binary file transfer support
- Chunked data transmission for efficient memory usage
- Error handling and connection management
- Cross-platform compatibility (Linux/Unix systems)

2 Architecture Design

2.1 System Architecture

The system follows a pure peer-to-peer architecture where two peers communicate directly without intermediary servers.

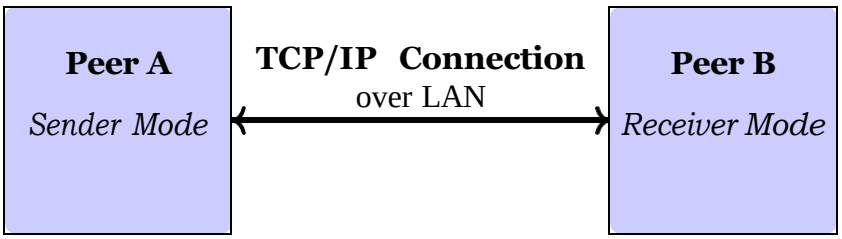


Figure 1: Peer-to-peer system architecture without centralized server

2.2 Component Architecture

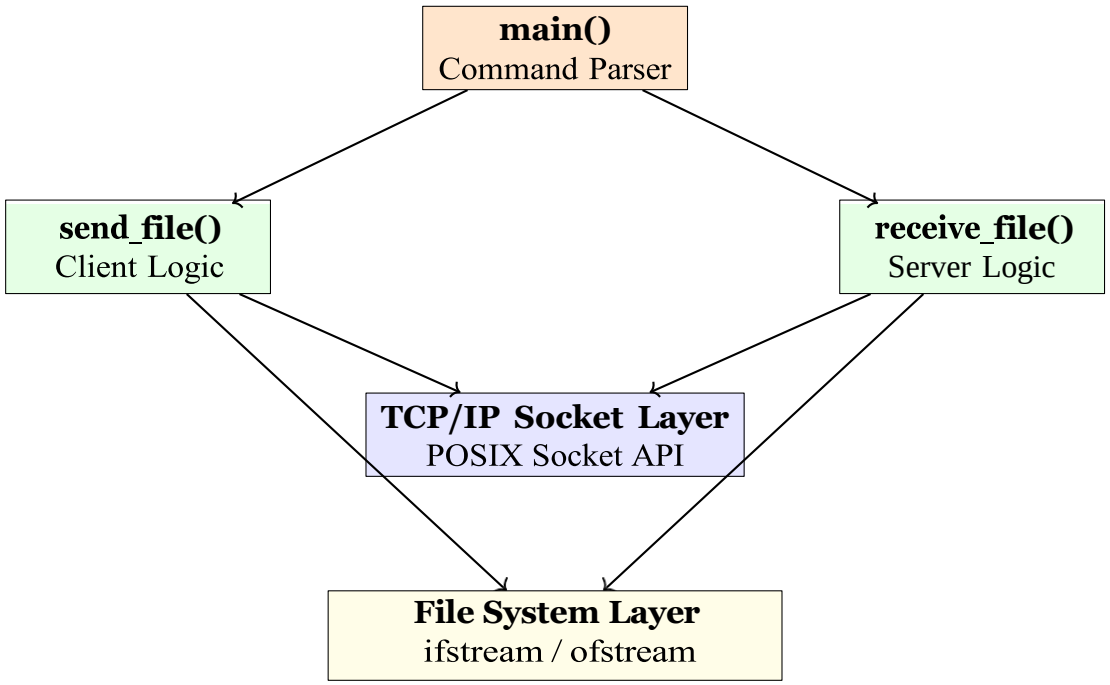


Figure 2: Software component architecture

2.3 Directory Structure

```
project_root/
        peer.cpp              # Main source code
        example.txt           # Sample file to send
        peer                  # Compiled executable
        received_output.txt   # Output file (after receiving)
```

3 Protocol Design

3.1 Communication Protocol Overview

The application implements a custom application-layer protocol on top of TCP/IP to ensure reliable file transfer. The protocol defines a strict sequence of data exchange between sender and receiver.

3.2 Protocol Handshake and Data Transfer Sequence

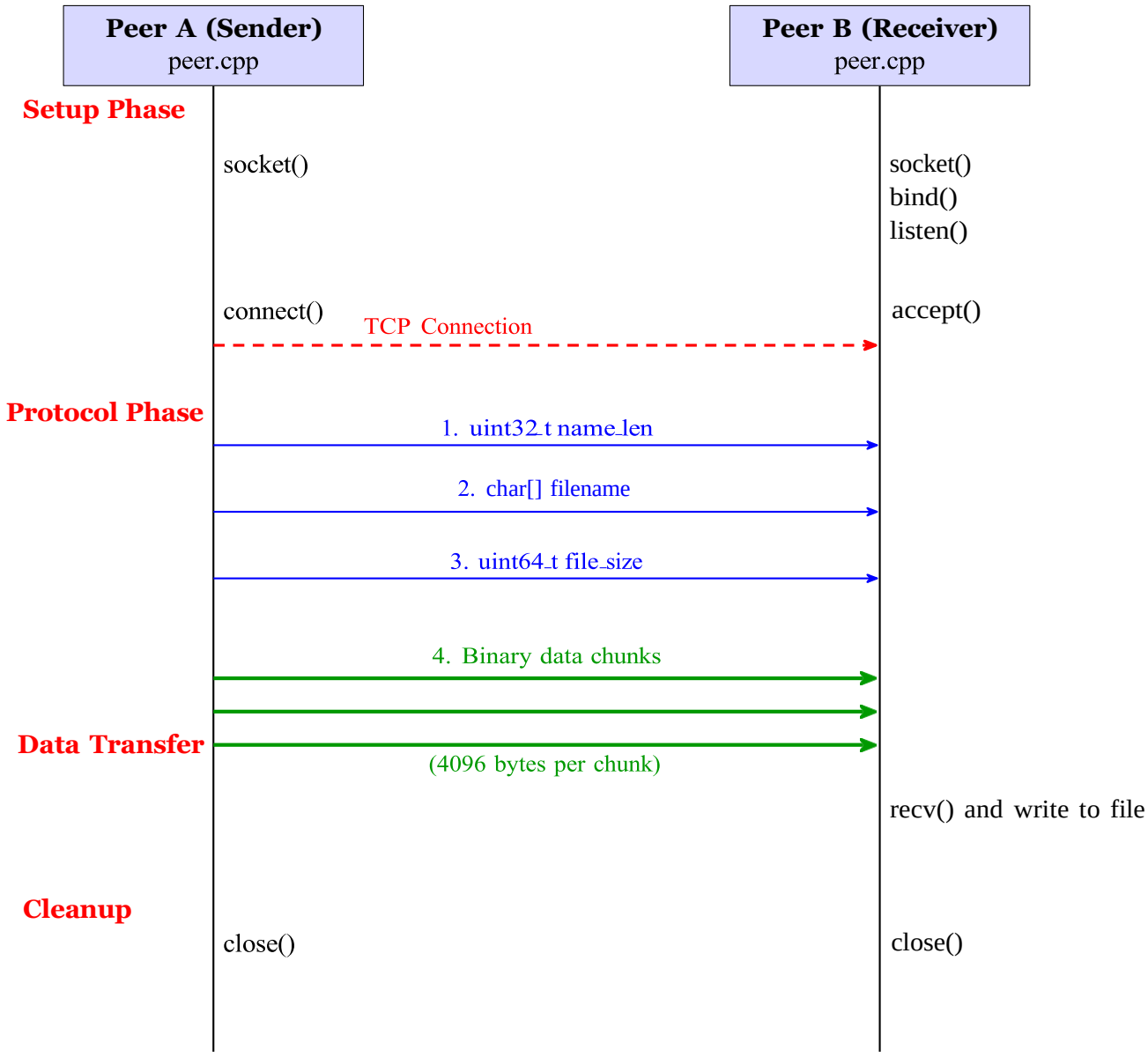


Figure 3: Complete protocol sequence diagram

3.3 Protocol Specification

The protocol consists of four distinct phases:

Phase 1: Connection Setup

1. Receiver creates socket and binds to port
2. Receiver enters listening state
3. Sender creates socket and connects to receiver's IP:port
4. TCP three-way handshake establishes connection

Phase 2: Metadata Exchange

1. Sender transmits filename length (uint32_t, 4 bytes)
2. Sender transmits filename string (variable length)
3. Sender transmits file size (uint64_t, 8 bytes)

Phase 3: File Data Transfer

1. Sender reads file in 4096-byte chunks
2. Each chunk is sent via TCP socket
3. Receiver writes chunks to output file
4. Process repeats until all bytes transferred

Phase 4: Connection Termination

1. Both peers close their sockets
2. TCP connection gracefully terminates

3.4 Data Format Specification

Field	Type	Size	Description
name_len	uint32_t	4 bytes	Length of filename string
filename	char[]	Variable	Actual filename (e.g., "example.txt")
file_size	uint64_t	8 bytes	Total file size in bytes
file_data	binary	Variable	File content in chunks

Table 1: Protocol data format

3.5 Why This Protocol Design?

- **Fixed-size headers:** Using uint32 t and uint64 t ensures consistent byte ordering across different systems
- **Metadata first:** Receiver knows exact file size before data arrives, preventing buffer overflows
- **Chunked transfer:** 4096-byte chunks prevent memory exhaustion for large files
- **TCP reliability:** Underlying TCP protocol handles packet loss, ordering, and error correction

4 Deployment Guideline

4.1 System Requirements

Operating System:

- Linux (Ubuntu 20.04+ recommended)
- Any POSIX-compliant Unix system

Compiler:

- g++ with C++17 support or newer
- Check version: `g++ --version`

Network Requirements:

- Both machines must be on the same Local Area Network (LAN)
- Firewall must allow TCP connections on chosen port

4.2 Compilation Steps

Step 1: Navigate to project directory

```
cd project_root/
```

Step 2: Compile the source code

```
g++ -std=c++17 peer.cpp -o peer
```

Step 3: Verify executable was created

```
ls -lh peer
```

Step 4: Make executable (if needed)

```
chmod +x peer
```

4.3 Network Configuration

4.3.1 Finding Your IP Address

On Linux, use the following command:

```
ip addr
```

Look for an IPv4 address in the format: inet 192.168.x.x/24

4.3.2 IP Address Usage Table

4.4 Port Selection

- Port 5000 is used in examples (arbitrary choice)
- Any unused port above 1024 can be selected
- Port must be the same on both sender and receiver
- Port must not be used by another application

IP Address	Usage
127.0.0.1	Loopback address (same machine only, for testing)
192.168.x.x	Private LAN address (for two different machines)
10.x.x.x	Alternative private network range

Table 2: IP address types and usage

5 Usage Guideline

5.1 Command Syntax

The program supports two modes of operation:

Receiver Mode (Server):

```
./peer receive <port>
```

Sender Mode (Client):

```
./peer send <ip_address> <port> <filename>
```

5.2 Scenario 1: Testing on Single Machine

This scenario is useful for testing the application without network setup.

Step 1: Open two terminal windows

Step 2: Terminal 1 - Start receiver

```
./peer receive 5000
```

Expected output:

```
[Receiver] Waiting for connection...
```

Step 3: Terminal 2 - Send file

```
./peer send 127.0.0.1 5000 example.txt
```

Expected output:

```
[Sender] Connected
[Sender] File sent successfully (XXX bytes)
```

Step 4: Verify received file

```
ls -lh example.txt
cat example.txt
```

5.3 Scenario 2: Transfer Between Two Machines

This is the actual P2P file transfer over LAN.

Machine A (Receiver):

Step 1: Find IP address

```
ip addr
# Example output: inet 192.168.1.13/24
```

Step 2: Start receiver

```
./peer receive 5000
```

Machine B (Sender):
Step 1: Ensure you're on the same network
Step 2: Send file to Machine A

```
./peer send 192.168.1.13 5000 example.txt
```

Important Notes:

- Receiver must be started BEFORE sender attempts connection
- Both machines must be on the same LAN
- Firewall may need to allow connections on chosen port
- Use Machine A's actual IP address (not 127.0.0.1)

5.4 Output Files

- Received file is saved with its original filename
- No pre-existing output file is required
- File is created automatically by the program
- File integrity is preserved (binary-safe transfer)

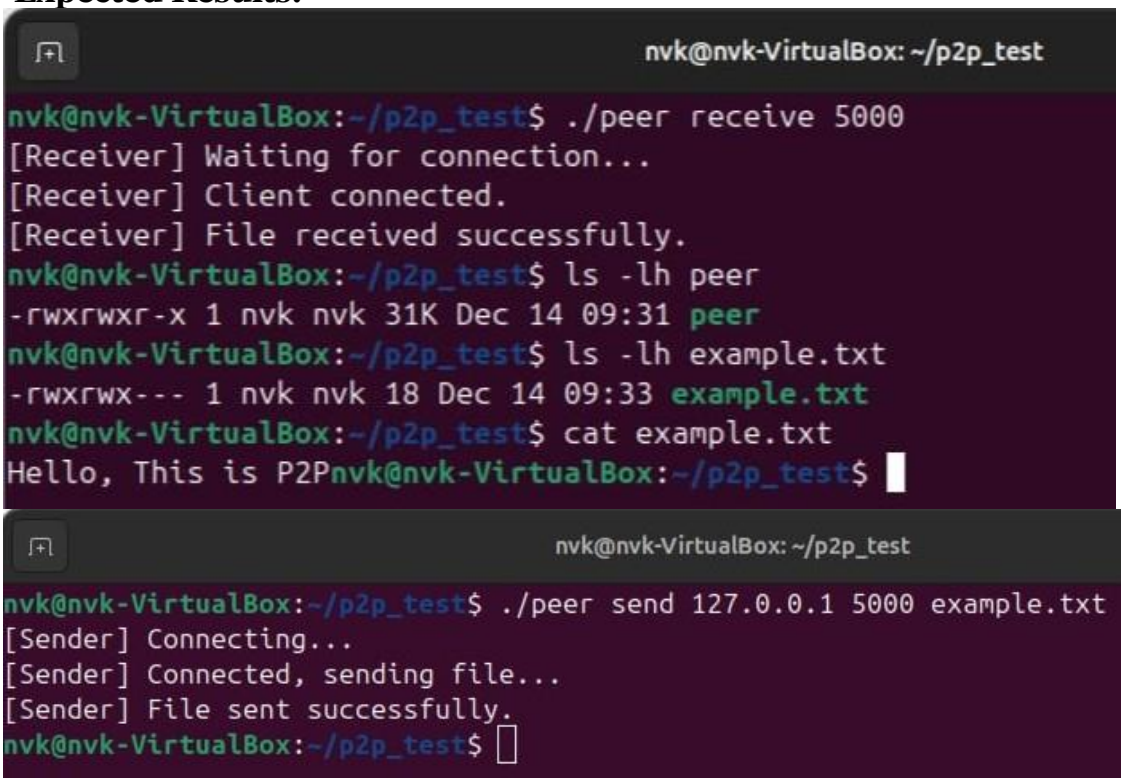
6 Results and Testing

6.1 Test Case 1: Local Machine Transfer

Test Setup:

- Single machine (Ubuntu 22.04)
- Loopback address (127.0.0.1)
- Port 5000
- Test file: example.txt (1.2 KB)

Expected Results:



Verification:

```
# Compare file sizes
ls -l example.txt
# Compare file contents
diff example.txt example.txt
# Should show no differences
```

6.2 Test Case 2: LAN Transfer

Test Setup:

- Two laptops on same Wi-Fi network (e.g: 192.x.x.x)
- Machine A: 192.168.1.13 (Receiver)
- Machine B: 192.168.1.25 (Sender)
- Port 5000
- Test file: example.txt

6.3 Error Handling Tests

Test: Connection Refused

```
Scenario: Sender tries to connect before receiver starts
Result: Connection failed with clear error message
```

Test: File Not Found

```
Scenario: Sender tries to send non-existent file
Result: "Failed to open input file" error message
```

Test: Port Already in Use

```
Scenario: Start receiver on port already occupied
Result: bind() fails with appropriate error
```

7 Technical Implementation Details

7.1 Socket Programming Concepts

7.1.1 Receiver (Server) Socket Operations

1. socket(): Creates endpoint for communication
2. bind(): Associates socket with specific port number
3. listen(): Marks socket as passive, ready to accept connections
4. accept(): Blocks until incoming connection arrives, returns new socket for data transfer
5. recv(): Receives data from connected socket
6. close(): Closes socket and releases resources

7.1.2 Sender (Client) Socket Operations

1. socket(): Creates endpoint for communication
2. inet_pton(): Converts IP address string to binary format
3. connect(): Establishes connection to receiver's IP and port
4. send(): Transmits data through connected socket
5. close(): Closes socket and releases resources

7.2 Memory Efficiency: Chunked Transfer

Instead of loading entire file into memory, the application uses a buffer-based approach:

```
char buffer[4096];    // Small fixed-size buffer

// Sender reads and sends in chunks
while (infile.read(buffer, sizeof(buffer))) {
    send(sock, buffer, infile.gcount(), 0);
}

// Receiver receives and writes in chunks
while (received < file_size) {
    ssize_t bytes = recv(client_fd, buffer,
                         min(sizeof(buffer), file_size - received
                           ), 0);
    outfile.write(buffer, bytes);
    received += bytes;
}
```

Benefits:

- Constant memory usage regardless of file size

- Can transfer files larger than available RAM
- Reduces memory fragmentation
- Improves performance through streaming

7.3 TCP/IP Advantages for File Transfer

Feature	**Benefit**
Reliable delivery	Guarantees all data arrives in correct order
Error checking	Automatic detection and correction of corrupted packets
Flow control	Prevents sender from overwhelming receiver
Congestion control	Adapts to network conditions automatically
Connection-oriented	Clear session establishment and termination

Table 3: TCP/IP benefits for file transfer

8 Common Issues and Troubleshooting

8.1 Connection Failed

Symptoms:

```
[Sender] Connection refused
```

Possible Causes and Solutions:

1. **Receiver not started**
   - Solution: Start receiver first, then sender
2. **Wrong IP address**
   - Solution: Verify IP with ip addr command
   - Use 127.0.0.1 only for same-machine testing
3. **Firewall blocking connection**
   - Solution: Temporarily disable firewall or add rule
   - Ubuntu: sudo ufw allow 5000/tcp
4. **Different networks**
   - Solution: Ensure both machines connected to same LAN

8.2 Permission Issues

Symptoms:

```
bash: ./peer: Permission denied
```

Solution:

```
chmod +x peer
```

8.3 Port Already in Use

Symptoms:

```
bind: Address already in use
```

Solutions:

1. Choose different port number (e.g., 5001, 5002)
2. Find and kill process using the port:

```
lsof -i :5000
kill -9 <PID>
```

8.4 File Not Found

Symptoms:

```
Failed to open input file
```

Solutions:

- Verify file exists: ls -l example.txt
- Check current directory: pwd
- Use absolute path: ./peer send ... /full/path/to/file.txt

8.5 Incomplete Transfer

Symptoms:

- Received file smaller than original
- Program hangs during transfer

Possible Causes:

1. Network disconnection during transfer
2. Insufficient disk space on receiver
3. Process killed before completion

Solutions:

- Restart both sender and receiver
- Check disk space: df -h
- Monitor network stability

9 Technical Summary

Aspect	Implementation
Protocol	TCP/IP
Architecture	Peer-to-Peer (P2P)
Language	C++ (C++17)
API	POSIX Sockets
Transfer Method	Byte stream (chunked)
Chunk Size	4096 bytes
File Type Support	Binary and text files
Network Scope	Local Area Network (LAN)
Platform	Linux/Unix systems
Concurrency	Single connection at a time

Table 4: Technical specifications summary

9.1 Code Statistics

- Total lines of code: 127 lines
- Functions: 3 (main, send_file, receive_file)
- External dependencies: Standard C++ library + POSIX sockets

9.2 Performance Characteristics

- Memory footprint: Constant (independent of file size)
- CPU usage: Minimal
- Transfer speed: Limited by network bandwidth
- Scalability: Suitable for files up to several GB

10 Contribution of Team Members

Member	Job part	Contribution
Nguyễn Việt Khoa	Code	30%
Nguyễn Xuân Đức		25%
Nguyễn Thế Xuân	write report	20%
Nguyễn Chí Tôn	PowerPoint	15%
Trần Tuấn Vũ	Present	10%

Table 5: Team member contributions with percentages

11 Conclusion

This project successfully demonstrates a working peer-to-peer file transfer application using TCP/IP sockets in C++. The implementation showcases fundamental concepts of distributed systems, including:

- Direct peer-to-peer communication without centralized servers
- Custom application-layer protocol design
- Reliable data transfer using TCP
- Efficient memory management through chunked transfers
- Cross-platform socket programming using POSIX API

The application serves both educational and practical purposes, providing students with hands-on experience in network programming while delivering a functional file transfer tool for local network use.