Distributed Systems

Practical Work 1: High-Performance TCP File Transfer

(Optimized with Zero-Copy & Header Batching)

Name: Dang Trung Nguyen
Student ID: 22BA13239
Class: 3.8

University of Science and Technology of Hanoi (USTH)

November 25, 2025

1 Introduction

The objective of this practical work is to implement a file transfer system using TCP sockets. While a standard implementation ensures data delivery, it often suffers from performance bottlenecks due to excessive system calls and memory copying between user space and kernel space.

This report presents an **optimized implementation** that addresses these issues using two advanced techniques:

1. **Header Batching:** Reducing network overhead by packing all file metadata (name length, file size, filename) into a single binary packet.
2. **Zero-Copy Transfer:** Utilizing the OS kernel's `sendfile` mechanism to transfer data directly from disk to network, bypassing the application's memory buffer.

2 Protocol Design

To ensure the server can correctly interpret the incoming byte stream without ambiguity, a strict binary protocol is defined. Unlike naive approaches that send metadata in separate steps, this optimization packs the header into a single structure.

2.1 Header Structure

The client constructs a single binary payload containing all necessary metadata before sending the file content.

Order	Field	Data Type (Struct)	Size
1	Name Length	Unsigned Int (`!I`)	4 bytes
2	File Size	Unsigned Long Long (`!Q`)	8 bytes
3	Filename	String (`s`)	Variable

Table 1: Packed Header Format

2.2 Data Transmission Flow

1. **Step 1 (Metadata):** Client sends the packed header (12 bytes + filename bytes).
2. **Step 2 (Parsing):** Server reads the first 12 bytes to extract the filename length and file size, then reads the filename.
3. **Step 3 (Zero-Copy Stream):** Client triggers `sendfile()`, causing the kernel to stream the file content immediately after the header.

3 Implementation Details

3.1 Client Optimization: Header Batching & Zero-Copy

The most significant optimization is in the client's send function. Instead of reading the file into a user-space buffer (RAM) and then writing it to the socket, we use `socket.sendfile`.

```python
def send_file(filepath):
    file_size = os.path.getsize(filepath)
    filename = os.path.basename(filepath).encode('utf-8')
    name_len = len(filename)

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((HOST, PORT))

        # OPTIMIZATION 1: Batching Headers
        # Format: ! (Network Endian), I (4 bytes), Q (8 bytes), s (string)
        header_fmt = f'!IQ{name_len}s'
        header = struct.pack(header_fmt, name_len, file_size, filename)
        s.sendall(header)

        # OPTIMIZATION 2: Zero-Copy
        # Transfer directly from Disk -> Kernel Buffer -> NIC
        with open(filepath, 'rb') as f:
            sent = s.sendfile(f)

    print(f"Sent {sent} bytes via Zero-Copy mechanism.")
```

Listing 1: Client Implementation using Struct Packing and Sendfile

3.2 Server Implementation: Robust Parsing

The server uses a helper function `receive_all` to ensure it reads exactly the amount of data required for the fixed-length header parts, preventing fragmentation errors.

```python
# 1. Read the Fixed Header (Length + Size = 12 Bytes)
fixed_header = receive_all(conn, 12)
if not fixed_header: return

# Unpack: I (4 bytes) + Q (8 bytes)
name_len, file_size = struct.unpack('!IQ', fixed_header)

# 2. Read the Variable Filename
filename = receive_all(conn, name_len).decode('utf-8')
safe_filename = f"RECV_{os.path.basename(filename)}"

# 3. Stream Data to Disk
with open(safe_filename, 'wb') as f:
    received = 0
    while received < file_size:
        to_read = min(BUFFER_SIZE, file_size - received)
        chunk = conn.recv(to_read)
        if not chunk: break
        f.write(chunk)
        received += len(chunk)
```

Listing 2: Server Header Parsing Logic

4 Performance Analysis

The optimized approach offers distinct advantages over the traditional read/write loop used in basic implementations:

- **Reduced Context Switching:** Traditional methods switch between User Mode and Kernel Mode for every chunk read and written. `sendfile` performs the entire transfer largely within Kernel Mode.
- **CPU Efficiency:** By avoiding the copy of data into the application's memory buffer, the CPU is freed for other tasks, and the memory bus bandwidth is conserved.
- **Network Efficiency:** Packing headers reduces the number of small TCP packets, mitigating the overhead associated with TCP headers and acknowledgement latency.

5 Conclusion

This practical work successfully implemented a high-performance TCP file transfer system. By leveraging `struct` for efficient protocol definition and `sendfile` for zero-copy data transmission, the system achieves maximum throughput with minimal CPU usage, satisfying requirements for modern distributed system components.