Practical Work 1 - TCP File Transfer

Tran Hai Nam

December 2025

1 Introduction

In distributed systems, file transfer is a basic operation for sharing data between nodes. This lab implements a simple 1-to-1 file transfer system using raw TCP sockets with a command-line interface (CLI). The main goals are to:

- practice the core concepts of sockets, client/server, and TCP connections;
- design a simple application-level protocol on top of TCP;
- organize code and handle file I/O in a streaming manner.

2 Objectives

- Build a TCP server that listens on a configurable `host:port` and serves one client per run.
- Build a CLI client that can upload and download files to and from the server.
- Design a minimal header that still carries enough information (filename, size, status).
- Ensure that files are transferred correctly (size and content) under normal conditions.

3 System Design

3.1 Overview

The system consists of two components:

- **Server**: runs on a machine, listens for TCP connections, and saves files into a configured directory (default `received_files/`).
- **Client**: runs on the user's machine, connects to the server to send or receive files.

High-level flows:

- Upload: `Client -> TCP -> Server -> Store file`
- Download: `Client -> TCP -> Server -> Read file`

3.2 Application-Level Protocol

The protocol is intentionally minimal, using fixed-size primitive fields in the header for easy parsing.

- **Upload request** (client to server):
  - `byte op` = `'U'` (operation code for upload).

  - `int name_len`: length of filename in bytes (UTF-8).
  - `long file_size`: file size in bytes.
  - `name_len` bytes: filename (UTF-8).
  - `file_size` bytes: file content.

- **Upload reply** (server to client):
  - ASCII string: `OK <filename> <size> bytes\n`, used as an acknowledgement and for logging.

- **Download request** (client to server):
  - `byte op` = `'D'` (operation code for download).
  - `int name_len`: length of filename.
  - `name_len` bytes: filename (UTF-8) to download.

- **Download reply** (server to client):
  - `byte status`: status code (`0` = OK, `1` = file not found).
  - If `status == 0`:
    * `long size`: file size.
    * `size` bytes: file content.

Rationale:

- A small header with only primitive types (`byte`, `int`, `long`) makes reading and writing straightforward.
- Filenames are UTF-8 to support non-ASCII names.
- File content is streamed over the socket; there is no need to load the entire file into memory.

4 Implementation

4.1 Technology and Environment

- Language: Java.
- Standard libraries only: `java.net`, `java.io`, `java.nio.file`.
- No external frameworks, to keep the focus on plain TCP sockets.

4.2 Code Structure

- `lab1_file/Server.java`
  - Parses command-line arguments: `host`, `port`, `output_dir` (defaults: `0.0.0.0`, `9000`, `received_files`).
  - Creates a `ServerSocket` and accepts exactly one client connection.
  - Delegates request handling to `handleClient`, which reads `op` and dispatches to upload or download handlers.

- `lab1_file/Client.java`
  - Supports two main modes: `upload` and `download`.
  - Parses CLI arguments to determine local file, remote name, `host`, and `port`.

4.3 Upload Flow

Client side:

1. Check that the local file exists (`Files.isRegularFile`).
2. Choose the remote filename (defaults to the local filename).
3. Open a TCP connection to the server via `new Socket(host, port)`.
4. Write the upload header: `'U'`, `name_len`, `file_size`, `name_bytes`.
5. Stream the file in chunks of 64 KB (`CHUNK_SIZE = 64 * 1024`) from disk to the socket.
6. After sending all bytes, read the server's acknowledgement (if any) and print it.

Server side:

1. Read `op = 'U'`, then `name_len`, `file_size`, and `name_bytes`.
2. Normalize the filename with `Paths.get(rawName).getFileName()` to drop any directory components.
3. Resolve the destination path in the output directory.
4. Loop reading from the socket into a buffer and write to the file until `file_size` bytes are received.
5. Log the result and send back the `OK <filename> <size> bytes` message.

4.4 Download Flow

Client side:

1. Read the remote filename from command-line arguments.
2. Decide the local output path (defaults to the same name in the current directory).
3. Send the download header: `'D'`, `name_len`, `name_bytes`.
4. Read `status` from the server.

- If `status != 0`: print an error message ("file not found") and return.
- If `status == 0`: read `size`, then loop reading file content and writing to disk until `size` bytes are received.

Server side:

1. Check that the requested file exists (`Files.isRegularFile`).
2. Read the file into a buffer and send it, preceded by `status = 0` and the file size.
3. If the file does not exist, send `status = 1`.

5 Build and Run

- Compile:

```
javac lab1_file/Server.java lab1_file/Client.java
```

- Run server:

```
java -cp lab1_file Server 0.0.0.0 9000 received_files
```

- Upload a file:

```
java -cp lab1_file Client upload <file_path> [host] [port] [remote_name
    ]
```

- Download a file:

```
java -cp lab1_file Client download <remote_name> [host] [port] [
    output_path]
```

6 Evaluation and Discussion

6.1 Strengths

- Simple protocol that is easy to implement and debug.
- Streaming transfer with a 64 KB buffer avoids loading the whole file into memory.
- Server-side filename sanitization helps mitigate directory traversal attacks.

6.2 Limitations

- The server serves only one client per run; there is no concurrency.
- No authentication or encryption; data travels in plaintext.
- No checksum or resume mechanism; interrupted transfers must restart from the beginning.

6.3 Possible Extensions

- Extend the server to handle multiple clients concurrently (for example, using a thread per connection or a thread pool).
- Add checksums (for example, MD5 or SHA-256) to verify data integrity after transfer.
- Integrate TLS to protect file content over the network.
- Implement resume support for partially transferred files.

7 Conclusion

This lab builds a simple 1-to-1 TCP-based file transfer system with a custom protocol, demonstrating the full path from header design to client and server implementation. It provides a concrete baseline for later labs that introduce higher-level abstractions such as RPC, where developers focus on service interfaces instead of low-level socket details.