Practical Work 1 – TCP File Transfer

1 Introduction

The goal of this practical work is to implement a simple one-to-one file transfer application over TCP/IP using a client–server model. The system is implemented in Python and runs on Linux. Key requirements for the system include:

- **Single client-server pair:** Exactly one server process and one client process are used.
- **TCP sockets for reliability:** Communication is built on TCP for reliable byte-stream delivery.
- **File transfer capability:** The client can send a file (of arbitrary type and size) to the server over the network.

In this report, we describe (i) the design of the application-level protocol, (ii) the overall system organization, (iii) the main implementation details in Python (with code snippets), and (iv) the testing methodology and results for the file transfer system.

2 Protocol Design

2.1 Objectives

The custom application-level protocol is designed with several objectives in mind:

- **Reliability:** Ensure that arbitrary files are transferred correctly and completely using TCP's reliable stream.
- **Clarity and Simplicity:** Keep the protocol format simple and easy to parse.
- **Definite Boundaries:** By sending metadata (like filename and size) first, the server knows exactly how to interpret the incoming stream and how much data to expect.
- **Extensibility:** The protocol can be extended in the future (for example, to handle multiple files, include integrity checks or authentication) without breaking its basic structure.

2.2 Protocol Steps

The file transfer protocol defines the sequence of messages between client and server on top of TCP. The interaction proceeds as follows:

1. **Connection Establishment:** The client establishes a TCP connection to the server (which is listening on a known port).
2. **Metadata – Filename Length:** The client first sends the length of the filename as a 4-byte unsigned integer (in network byte order, big-endian). This informs the server about how many bytes to read for the upcoming filename.
3. **Metadata – Filename:** The client then sends the filename itself as a UTF-8 encoded byte string. The server reads this exact number of bytes to obtain the filename.

4. **Metadata – File Size:** Next, the client sends the file size as an 8-byte unsigned integer (representing the number of bytes in the file). This tells the server the total size of the file content that will follow.
5. **File Data Transfer:** The client sends the actual file content as a stream of bytes, typically in fixed-size chunks (e.g., 4 KB blocks) until all bytes of the file are transmitted.
6. **Connection Teardown:** After sending the file data, the client closes the connection. The server, once it has received the expected amount of data (as indicated by the file size), closes its socket. At this point the transfer is complete.

2.3 Protocol Message Layout

The layout of the message sent from client to server is illustrated below. First comes a **metadata header** consisting of the filename length and file size, followed by the filename string, and then the binary file content. The table shows the format:

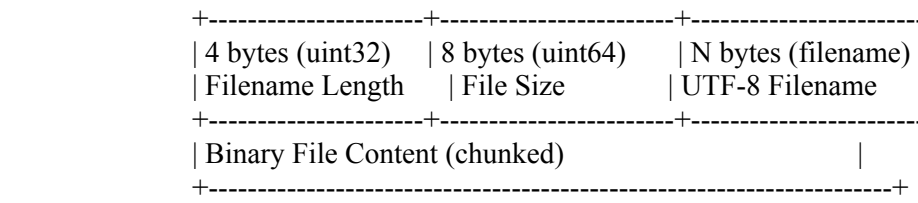

Figure 1: Protocol Message Structure.

In summary, the client first sends a fixed-size header (12 bytes) containing the length of the filename and the file size. It then sends the filename (whose length is given in the header), and finally streams the file content. This structure ensures the server can allocate buffers and know when the transfer of the file is complete.

3 System Organization

3.1 Components

The system follows a classic client–server architecture. It consists of two main components:

- **Client:** The client is responsible for reading a local file from disk, sending the file's metadata (filename and size), and then streaming the file's contents over the TCP connection.
- **Server:** The server listens for an incoming TCP connection. When a client connects, the server receives the metadata, then receives the file data and writes the content to a new file on disk (reconstructing the original file).

Only a single client is handled at a time (the implementation is single-threaded), which is sufficient for this practical demonstration.

3.2 Overall Architecture

The overall organization of the client and server and their interactions is depicted in the diagram below. The client (file sender) and server (file receiver) communicate over a TCP connection. The arrows indicate the flow of data from the client to the server:

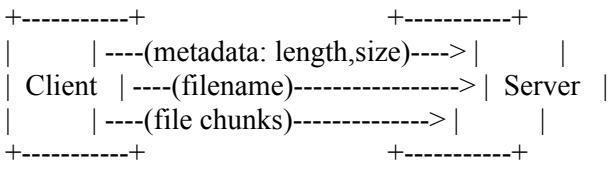

Figure 2: Client–Server Architecture for the file transfer system.

In this design, the **Server** performs the following steps:

1. Create a TCP socket, set socket options (e.g., allow address reuse), and bind to a known port. Then listen for a connection.
2. Accept one incoming client connection.
3. Receive the metadata (filename length, filename, file size) as per the protocol, then receive the file data in chunks and write it to a file on disk.
4. Close the connection (after ensuring the entire file is received and written).

Correspondingly, the **Client** carries out these steps:

1. Create a TCP socket and connect to the server's IP address and port.
2. Open the local file to be sent; determine the filename and measure the file size in bytes.
3. Send the metadata: first send the filename length (4 bytes), then the filename string, then the file size (8 bytes).
4. Send the file data by reading from the file in chunks (e.g., 4096 bytes at a time) until end-of-file.
5. Close the connection after all file bytes have been sent.

This architecture ensures that the server knows what file name to use and how many bytes to expect, and it enables the file content to be transmitted in a streaming fashion efficiently.

4 Implementation

The file transfer system is implemented in Python 3, using the built-in `socket` library for network communication and the `struct` module to pack/unpack binary values in network byte order. Below we provide key snippets of the implementation for both server and client, highlighting how the protocol is realized in code.

Listing 1: *Server-side code to accept a connection and receive a file.*

```
import socket, struct, os

# Server: set up listening socket
server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server_sock.bind(('0.0.0.0', 5000))
server_sock.listen(1)
print("[*] Server listening on port 5000")

# Accept a client
conn, addr = server_sock.accept()
```

```
print(f"[+] Connected from {addr}")

# Receive metadata (filename length and file size)
raw = conn.recv(4)
if len(raw) < 4:
    raise ConnectionError("Connection closed before receiving filename length")
name_len = struct.unpack('!I', raw)[0]                    # filename length in bytes
filename_bytes = conn.recv(name_len)
filename = filename_bytes.decode('utf-8')
size_data = conn.recv(8)
filesize = struct.unpack('!Q', size_data)[0]              # file size in bytes
print(f"[+] Receiving file: {filename} ({filesize} bytes)")

# Receive file content in chunks and write to disk
os.makedirs("received_files", exist_ok=True)
out_path = os.path.join("received_files", filename)
with open(out_path, 'wb') as f:
    bytes_received = 0
    while bytes_received < filesize:
        data = conn.recv(4096)
        if not data:
            break  # connection closed unexpectedly
        f.write(data)
        bytes_received += len(data)
print(f"[+] File saved to {out_path} ({bytes_received} bytes)")

conn.close()
server_sock.close()
```

Listing 2: *Client-side code to send a file to the server.*

```
import socket, struct, os

server_host = "127.0.0.1"
server_port = 5000
file_path   = "test.txt"  # path of the file to send

# Client: set up connection to server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((server_host, server_port))
    print("[*] Connected to server")

    # Prepare metadata
    filename = os.path.basename(file_path)
    filename_bytes = filename.encode('utf-8')
    filesize = os.path.getsize(file_path)
    print(f"[+] Sending file: {filename} ({filesize} bytes)")

    # Send metadata (name length, name, and file size)
    sock.sendall(struct.pack('!I', len(filename_bytes)))  # 4-byte length
```

```
    sock.sendall(filename_bytes)                    # filename string
    sock.sendall(struct.pack('!Q', filesize))       # 8-byte file size

    # Send file content in chunks
    with open(file_path, 'rb') as f:
        while True:
            chunk = f.read(4096)
            if not chunk:
                break  # EOF
            sock.sendall(chunk)
    print("[+] File transmission complete")
```

In the server code (Listing 1), after accepting a connection, the metadata is received step by step: first 4 bytes for the filename length, then the filename itself, and then 8 bytes for the file size. The server then enters a loop to receive the file data in chunks of up to 4096 bytes and writes them to a file (in a directory received_files/). The client code (Listing 2) performs the complementary actions: it reads the file, uses struct.pack to convert integers to network-byte-order bytes (for filename length and file size), and sends these followed by the raw file data. Both client and server use the same chunk size (4096 bytes in this case) to send/receive the file efficiently.

5 Testing and Results

The file transfer system was tested on a local Linux machine using two terminal windows (one for the server and one for the client). Both the client and server were run with Python 3. For example:

- **Server command:** python server.py 5000
- **Client command:** python client.py 127.0.0.1 5000 test.txt

In a test run, a sample text file of 30 bytes (test.txt) was sent by the client and successfully received by the server. The server stored the file (with the same name in a designated folder, e.g. received_files/test.txt), and the content of the received file was verified to be identical to the original. The protocol was further verified using larger files (such as images and PDFs) to ensure it works for binary data and various sizes. In all cases, the files were transmitted correctly.

Because the current implementation is single-threaded (the server handles one connection at a time), multiple clients cannot transfer files simultaneously. However, this limitation is acceptable for this practical work. The results confirmed that the TCP file transfer is reliable and that the simple protocol is sufficient for correct file reconstruction on the server side.

6 Conclusion

This lab exercise resulted in a working TCP-based file transfer system with a custom protocol for metadata and file transmission. By sending metadata (filename and size) before the file content, the server always knows how to interpret the incoming stream and how much data to read, which ensures a robust transfer. The implementation successfully transmitted files of various sizes and types, demonstrating the reliability of TCP for such tasks.

Overall, the project illustrates a clear method for transferring files over a network using sockets. Future enhancements could include support for multiple concurrent clients (e.g., using threading or asynchronous I/O), additional protocol features like acknowledgments or checksums for data integrity, and authentication mechanisms. The current solution, however, meets the requirements of the practical work and provides a solid foundation for more advanced file transfer capabilities.