

# Practical Report 2 - Design and Implementation of a File Transfer RPC Service

Phung Dam Tien Si / 23BI1484

November 28, 2025

## 1 RPC Service Design

The Remote Procedure Call (RPC) service is designed to provide a simple, synchronous interface for a client to request a file transfer to the server. The actual file transfer, which involves a continuous stream of data, requires a robust, application-layer protocol to ensure message boundaries (i.e., when one file transmission ends and another begins).

### 1.1 Service Interface Definition

The service exposes a single, high-level procedure, `TransferFile`, which abstracts the underlying TCP socket operations:

```
Result TransferFile(string filename, stream fileData)
```

Under the hood, the RPC mechanism uses a three-phase data transmission protocol to send crucial metadata before the bulk file content. This layered design allows the server's `skeleton` to correctly de-serialize the file stream.

### 1.2 Three-Phase Transport Protocol

Since the transfer relies on stream-oriented TCP/IP, the RPC payload must be structured with explicit length headers.

1. **Phase 1: Filename Length (4 Bytes - Unsigned Int)**: This header tells the receiving RPC layer exactly how many bytes to read for the file's name.
2. **Phase 2: Filename (Variable Length - UTF-8)**: The actual name of the file being sent.
3. **Phase 3: File Size (8 Bytes - Unsigned Long Long)**: This crucial header defines the total number of bytes expected for the file content, serving as the stop condition for the receiving loop.
4. **Phase 4: File Data (Variable Length - Raw Binary)**: The file content, sent in chunks.

**Diagram Placeholder: RPC File Transfer Protocol Data Structure**

*Sequence: [4B Filename Length] → [Filename Bytes] → [8B File Size] → [File Content Chunks]*

Figure 1: Data Structure for RPC File Transfer Protocol

## 2 System Organization

The system is organized into a traditional client-server architecture with the RPC layer acting as middleware, enabling seamless cross-process communication. The separation of concerns ensures that the client only interacts with the RPC stub, and the server implementation is shielded from direct network handling.

1. **Client Application:** Invokes the local `TransferFile` stub.
2. **Client RPC Stub:** Acts as a proxy. It marshals (packs) the function parameters and metadata (file size, filename length) into the byte stream according to the protocol defined in Section 1. It is responsible for initiating and sending the entire TCP stream.
3. **RPC Server (Skeleton):** Listens for incoming connections. When a connection is accepted, the skeleton first de-marshals (unpacks) the metadata (Filename Length, File Size) to prepare for the bulk data transfer.
4. **Service Implementation:** This is the server's application logic. Once the metadata is received, the RPC skeleton calls the actual service implementation function, which contains the dedicated file receiving loop.

**Diagram Placeholder: Client-Server System Organization**  
*Flow: Client App ↔ Stub ↔ Network ↔ Skeleton ↔ Service Implementation*

Figure 2: Client-Server System Organization with RPC Layer

## 3 File Transfer Implementation

The most critical aspect of the implementation, housed within the server's service logic, is the loop that handles receiving the large file content reliably. This loop ensures that the connection

remains open until the exact number of expected bytes (`file_size`) is received, providing robustness against partial reads inherent in TCP streaming. The use of `conn.recv(bytes_to_read)` is dynamically sized to prevent over-reading past the file boundary.

## Code Snippet: Server-side File Receiving Loop (Python)

This snippet is executed within the RPC server's service implementation after the file metadata headers have been successfully received and parsed.

```
1 bytes_received = 0
2
3 # 'new_filename' and 'file_size' are extracted from RPC headers
4 with open(new_filename, 'wb') as f:
5     while bytes_received < file_size:
6         # Calculate how many bytes are left to receive
7         remaining_bytes = file_size - bytes_received
8
9         # Determine the maximum size for the current chunk
10        # (min of CHUNK_SIZE or remaining bytes)
11        bytes_to_read = min(CHUNK_SIZE, remaining_bytes)
12
13        # Receive the chunk, ensuring we don't exceed the boundary
14        chunk = conn.recv(bytes_to_read)
15
16        if not chunk:
17            # Connection closed before receiving the full file (error condition)
18            print(f"Connection closed prematurely. Received {bytes_received} of {file_size} bytes.")
19            break
20
21        f.write(chunk)
22        bytes_received += len(chunk)
23
24        # Simple progress indicator for CLI
25        sys.stdout.write(f"\rProgress: {bytes_received}/{file_size}*100:.2f%\r")
26        sys.stdout.flush()
27
28 print("\nFile transfer complete.")
29
30 if bytes_received < file_size:
31     os.remove(new_filename)
32     print("Incomplete file deleted.")
```

Listing 1: Server-side File Receiving Loop (Python)