Practical Work 2: RPC File Transfer System

Nguyen Tuan Dung
Student ID: 23BI14113

December 1, 2025

1 Introduction

This report details the migration of the file transfer system from a low-level TCP Socket implementation (Practical 1) to a high-level **Remote Procedure Call (RPC)** architecture using the Python **XML-RPC** library. The objective is to demonstrate the principles of RPC transparency, where network complexity is abstracted away, allowing the programmer to focus solely on defining and invoking remote functions.

2 RPC Service Design

The primary goal of the RPC design is to define the necessary functions the Server must expose for the Client to perform a file transfer. We define a `FileTransferService` with three key procedures.

2.1 Defined Remote Procedures

Table 1: RPC Service Interface Definition

Procedure Name	Client Input Parameters	Server Return Value	Purpose
`start_transfer`	`filename`	`success` (boolean)	Initializes file reception on the Server.
`transfer_chunk`	`filename`, `encoded_data`, `is_final`	`success` (boolean)	Writes a data chunk to the file.

The `encoded_data` parameter is necessary because standard XML-RPC protocols cannot directly handle large Python byte objects; thus, the data chunk must be converted to a **Base64-encoded string** (a form of marshaling) before transmission.

2.2 Service Interaction Flow

The RPC protocol dictates a specific sequence of function calls to manage the state of the file transfer process.

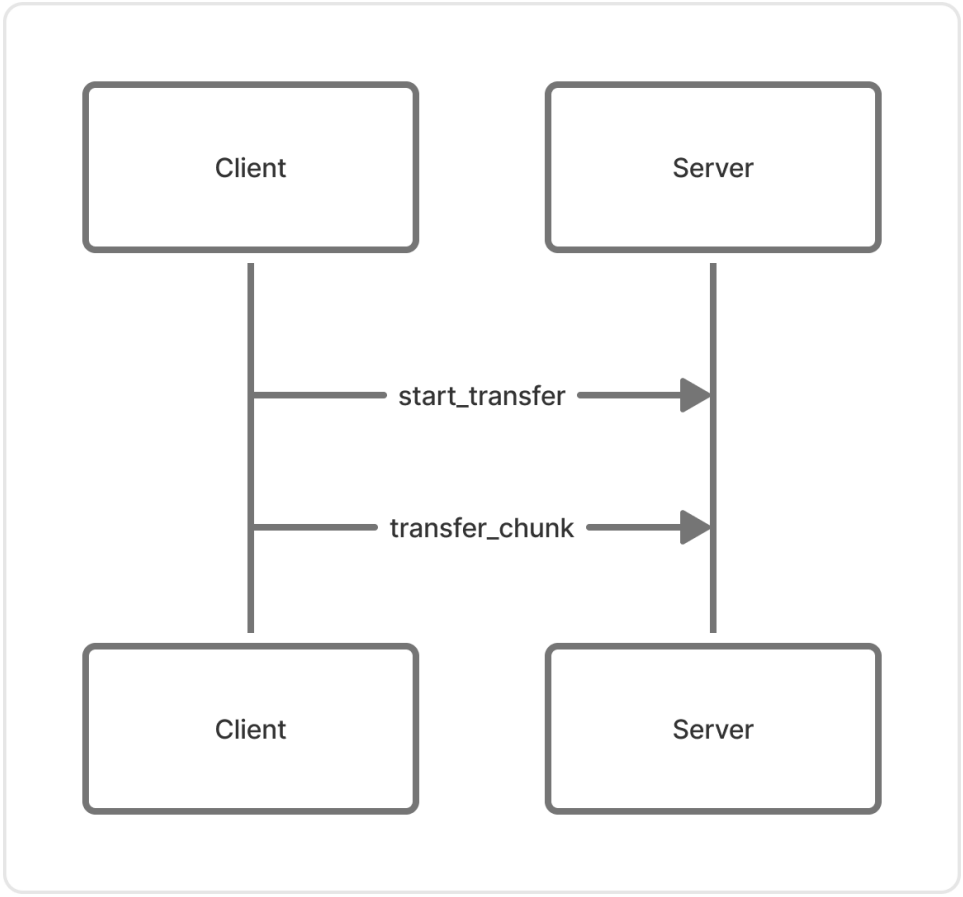

Figure 1: RPC File Transfer Sequence Diagram. The Client first calls `start_transfer` to initialize the Server's file handle before entering a loop of `transfer_chunk` calls.

3 System Organization

The system organization shifts from focusing on sockets to focusing on the RPC layer, which acts as the intermediary between the application logic and the network.

- **Client (Client Application):**
  1. Reads the file from local disk chunk by chunk.
  2. Performs **Marshaling**: Encodes the binary chunk into Base64 format.
  3. Invokes the remote procedures (`proxy.transfer_chunk`) as if they were local functions.
- **Server (Server Application):**
  1. Registers the remote procedures (making them available).
  2. Performs **Unmarshaling**: Decodes the Base64 string back into binary data.
  3. Executes the business logic: Writes the binary data to the disk.
  4. Manages file state using the global `file_handles` dictionary.

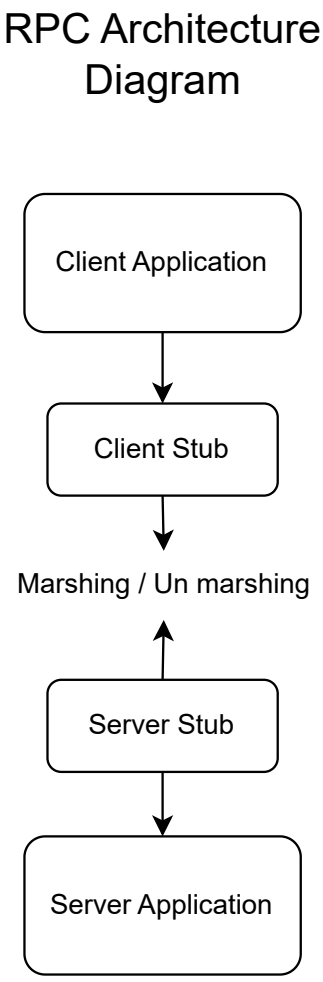

Figure 2: RPC Client-Server Architecture. The RPC layer abstracts the underlying TCP communication.

4 Implementation of File Transfer

The key difference in implementation is the elimination of manual protocol management (e.g., using `struct.pack` for metadata) in favor of calling a function with strongly typed parameters (string, base64-string, boolean).

4.1 Client Side Implementation (RPC Call)

The client no longer deals with network setup; it merely obtains a proxy object and makes calls.

Listing 1: RPC Client: Invoking the Remote Procedure

```
# Create Proxy
server_url = f"http://{HOST}:{PORT}"
proxy = xmlrpc.client.ServerProxy(server_url)

# 2. Inside the transfer loop
while True:
    chunk = f.read(BUFFER_SIZE)
    is_final = not chunk

    # Marshaling: Base64 encoding the binary data
    encoded_chunk = base64.b64encode(chunk).decode('utf-8')

    # Remote Procedure Call (RPC)
    success = proxy.transfer_chunk(filename, encoded_chunk, is_final)

    if is_final or not success:
        break
```

4.2 Server Side Implementation (Function Registration)

The server's network logic is reduced to initializing the RPC handler and registering the application functions.

Listing 2: RPC Server: Registration of Remote Function

```
def transfer_chunk(filename, encoded_data, is_final):
    # Unmarshaling: Decode Base64 string back to binary
    chunk = base64.b64decode(encoded_data)

    # Application Logic: Write to file handle
    file_handles[filename].write(chunk)

    if is_final:
        # Finalize and close the file
        file_handles[filename].close()
        del file_handles[filename]

    return True

# Initialize and register the Server
with SimpleXMLRPCServer((HOST, PORT), logRequests=False) as server:
    server.register_function(transfer_chunk, 'transfer_chunk')
    server.serve_forever()
```

5 Conclusion

The upgrade to an RPC-based file transfer system successfully demonstrated the primary advantage of RPC: **network transparency**. By shifting the responsibility of serialization and transport to the XML-RPC middleware, the application code became cleaner, focusing exclusively on file handling logic (reading, writing, Base64 conversion). This architecture is inherently more scalable and easier to maintain than the manual socket implementation.