

# Practical Work 2: RPC File Transfer (Manual Implementation)

Nghiem Xuan Son - 23BI14388

December 3, 2025

## 1 RPC Protocol Design

In this practical work, instead of using a high-level library like XML-RPC, we implemented a **\*\*Manual RPC\*\*** mechanism using a custom binary protocol. This allows for lower overhead and gives us a deeper understanding of how RPC stubs work underneath.

We defined a strict binary header structure (Stub) that precedes every payload.

### 1.1 Packet Structure

Every message sent to the server follows this binary format (8 bytes header):

[RPC\_ID (4 bytes int)] + [PAYLOAD\_SIZE (4 bytes int)] + [PAYLOAD DATA...]

We defined 3 RPC Operations:

- **ID 1 (rpc\_sendFilename):** Payload contains the filename string.
- **ID 2 (rpc\_sendChunk):** Payload contains a chunk of binary file data.
- **ID 3 (rpc\_endFile):** Empty payload. Signals the server to close the file.

## 2 System Organization

The system is organized into three components:

1. **Stub (rpc\_stub.py):** Acts as the middleware. It provides helper functions that abstract the packet packing logic. The client calls these functions instead of sending raw bytes directly.
2. **Client (client.py):** Imports the stub. It reads the file and invokes stub functions in sequence (Filename → Chunks → End).
3. **Server (server.py):** Acts as the skeleton. It sits in a loop receiving 8-byte headers, unpacks them to find the 'RPC\_ID', and dispatches the logic to the appropriate handler.

## 3 Implementation Code

### 3.1 The Stub (RPC Marshalling)

We use Python's `struct` module to pack integers into binary bytes, mimicking C structs.

```
1 HEADER_FORMAT = 'ii' # Two integers (ID, Size)
2
3 def rpc_send_filename(sock, filename):
4     data = filename.encode('utf-8')
5     # Pack ID=1 and Size
6     header = struct.pack(HEADER_FORMAT, 1, len(data))
7     sock.sendall(header + data)
8
9 def rpc_send_chunk(sock, chunk):
10    # Pack ID=2 and Size
11    header = struct.pack(HEADER_FORMAT, 2, len(chunk))
12    sock.sendall(header + chunk)
```

Listing 1: `rpc_stub.py` logic

### 3.2 The Server (RPC Dispatcher)

The server unpacks the header and switches based on ID.

```
1 # Receive exactly 8 bytes for header
2 header = recv_exact(sock, 8)
3 rpc_id, size = struct.unpack('ii', header)
4
5 if rpc_id == 1:
6     # Handle Filename
7     filename = recv_exact(sock, size)
8     create_file(filename)
9 elif rpc_id == 2:
10    # Handle File Content
11    data = recv_exact(sock, size)
12    write_to_file(data)
13 elif rpc_id == 3:
14    close_file()
```

Listing 2: Server dispatch logic

## 4 Conclusion

By implementing the RPC stub manually, we successfully abstracted the networking complexity from the client logic. The client code is clean and looks like local function calls, fulfilling the core concept of RPC.