Practical Work 2: RPC File Transfer System

Dng Đc Anh - 23BI14012

December 5, 2025

Abstract

This report describes the conversion of our TCP file transfer system into an RPC-style service using raw C sockets. Rather than relying on an external RPC framework, we emulate RPC behavior using a custom fixed-size metadata header and a well-defined client–server protocol. The implementation includes a client stub, server skeleton, structured messaging sequence, reliable transfer utilities, and robust error handling.

1 RPC Service Design

We introduce a single RPC method:

`UploadFile(filename, filesize, data)`

To simulate RPC serialization, we use a fixed-size structure sent as the first packet:

Listing 1: Metadata Header

```c
typedef struct {
    char method[32];            // "UploadFile"
    char filename[256];
    long long filesize;
} Metadata;
```

This deterministic header enables the server to parse request information reliably.

1.1 RPC Communication Workflow

1. Client connects to server.
2. Client sends `Metadata`.
3. Server validates RPC method.
4. Server replies with 200 (ready).
5. Client streams file in chunks.
6. Server writes file and verifies byte count.
7. Server replies with 201 (success) or 500 (failure).

2 System Organization

Figure 1 shows a conceptual RPC mapping:

RPC Stub (Client) → Metadata Header → RPC Skeleton (Server)
Data Stream (Chunks) → File Writer

Figure 1: RPC-like system organization for file transfer.

Client Responsibilities

- Extract filename, compute filesize.
- Populate the `Metadata` structure.
- Send metadata to server.
- Stream file reliably with partial-send handling.
- Shutdown write-end and wait for final server status.

Server Responsibilities

- Accept TCP connection.
- Receive `Metadata` with fixed-size reading.
- Validate RPC method name.
- Send acknowledgment code (200).
- Receive file in chunks and write to output directory.
- Send final response (201/500).

3 Client Implementation

3.1 Reliable Header Send + ACK Receive

The client builds the metadata request:

Listing 2: Building RPC Metadata (Client)

```
Metadata metadata;
memset(&metadata, 0, sizeof(Metadata));
strncpy(metadata.method, "UploadFile", sizeof(metadata.method)-1);
strncpy(metadata.filename, filename_ptr, sizeof(metadata.filename)-1);
metadata.filesize = file_size;
send(sock_fd, &metadata, sizeof(Metadata), 0);
```

After sending metadata, it waits for server acknowledgment:

```
int ack_code;
if (recv_all(sock_fd, &ack_code, sizeof(ack_code)) <= 0 || ack_code != 200) {
    printf("[Client] Invalid server ACK.\n");
    return;
}
```

3.2 Reliable File Transmission

We ensure full-send even when `send()` transmits fewer bytes:

Listing 3: Safe Partial-Send Transmission

```
while ((bytes_read = fread(buffer, 1, CHUNK_SIZE, file)) > 0) {
    ssize_t sent = 0;
    while (sent < bytes_read) {
        ssize_t n = send(sock_fd, buffer + sent, bytes_read - sent, 0);
        sent += n;
    }
}
```

After sending all data:

```
shutdown(sock_fd, SHUT_WR);
```

3.3 Final Status from Server

```
int response_code;
recv_all(sock_fd, &response_code, sizeof(response_code));
```

4 Server Implementation

The server continuously accepts connections and processes RPC requests.

4.1 Receiving Metadata

Listing 4: Receiving Header (Server)

```
Metadata metadata;
ssize_t bytes_read = recv_all(conn_fd, &metadata, sizeof(Metadata));

metadata.method[31] = '\0';
metadata.filename[255] = '\0';

if (strcmp(metadata.method, "UploadFile") != 0) {
    printf("Invalid RPC Method.\n");
    return;
}
```

4.2 Acknowledging RPC Call

```
int ack_code = 200;
send(conn_fd, &ack_code, sizeof(ack_code), 0);
```

4.3 Receiving File Data

The server writes into `received_files/` automatically:

Listing 5: File Writing Loop (Server)

```
long long received_size = 0;

while (received_size < metadata.filesize) {
    size_t to_receive = (metadata.filesize - received_size > CHUNK_SIZE)
                        ? CHUNK_SIZE
```

```
                            : metadata.filesize - received_size;

    bytes_read = recv(conn_fd, buffer, to_receive, 0);
    write(fd, buffer, bytes_read);
    received_size += bytes_read;
}
```

4.4 Final Response

```
int response_code = (received_size == metadata.filesize) ? 201 : 500;
send(conn_fd, &response_code, sizeof(response_code), 0);
```

5 Conclusion

The project successfully simulates RPC behavior using raw sockets, fixed-size metadata, and structured client–server communication. Despite not using any RPC framework, the system preserves key RPC principles such as method invocation, request headers, structured replies, and modular client/server logic.