USTH Master Spoiler Alert

02.rcp.file.transfer

Nguyen Quynh Trang - 22BA13303

1 Protocol Design

This practical work involved upgrading our previous C socket-based file transfer system to utilize **Remote Procedure Call (RPC)**, specifically the **ONC RPC (Sun RPC)** standard. The main goal was to move from low-level socket programming (`send`/`recv`) to a higher-level function call abstraction, making the distributed system appear more centralized.

1.1 Why RPC?

The fundamental shift was driven by the limitations of raw sockets:

- **Abstraction:** RPC handles the network boilerplate (`connect`, `bind`, port management) automatically, allowing us to focus on application logic.
- **Marshaling:** RPC handles the serialization of parameters (data structures like filename and file blocks) across the network, ensuring correct data representation between machines.
- **Procedure-Oriented:** Instead of sending a raw data stream, we define explicit remote procedures like `send_filename` and `send_data_chunk` to manage the file transfer flow.

1.2 RPC Service Definition (IDL)

The "contract" between the Client and the Server is defined in the **Interface Definition Language (IDL)** file, `ft.x`. This IDL defines the remote functions and the data structures used for transmission.

```
const MAX_BUF_SIZE = 1024;

struct filename_request {
    string filename<MAX_BUF_SIZE>;
};

struct file_data {
    string filename<MAX_BUF_SIZE>;
    opaque content[MAX_BUF_SIZE]; // Binary data array
    int bytes_read;               // Actual bytes sent
};

program FILE_TRANSFER_PROG {
    version FILE_TRANSFER_VERS {
        transfer_status send_filename(filename_request) = 1;
        transfer_status send_data_chunk(file_data) = 2;
    } = 1;
} = 0x20000001;
```

Figure 1: Key Definitions in IDL (`ft.x`)

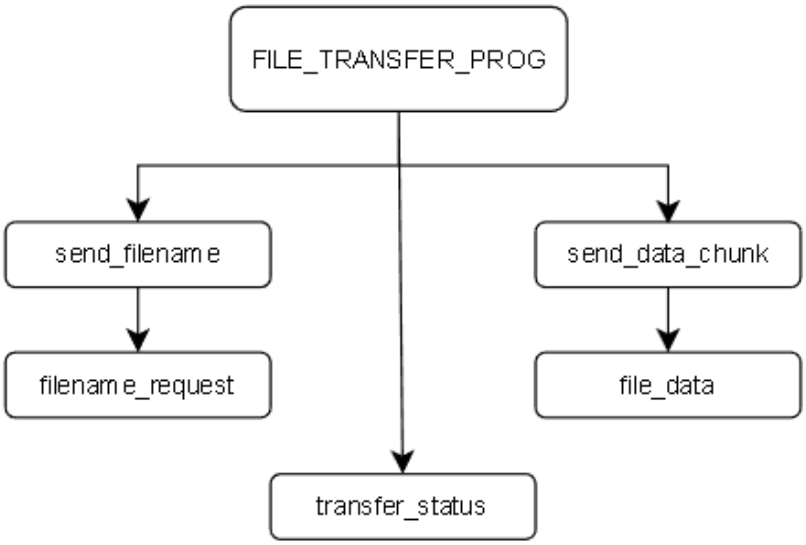

Figure 2: RPC Service Design: High-level Call Flow

2 System Organization

The RPC organization replaces the explicit Client-Server setup with the Stub-Skeleton architecture.

2.1 Component Roles

Component	Role	Key Mechanism	Configuration
RPC Client	Active Initiator	Calls `send_filename_1()`, `send_data_chunk_1()`	Creates Client Stub via `clnt_create`
Client Stub	Local Proxy	Marshals parameters, handles network call	Generated by `rpcgen` (`ft_clnt.c`)
Server Skeleton	Remote Proxy / Listener	Unmarshals parameters, calls implementation	Generated by `rpcgen` (`ft_svc.c`)
RPC Server Impl.	Passive Responder / File Writer	Implements `*_svc` functions, Disk I/O	Devoid of `main()`, `listen()`, `accept()`

2.2 Organization Flow Diagram

The architecture relies heavily on the **RPC Compiler (`rpcgen`)** to generate the bridging code, which is then compiled along with our custom implementation files (`ft_server.c` and `ft_client.c`).

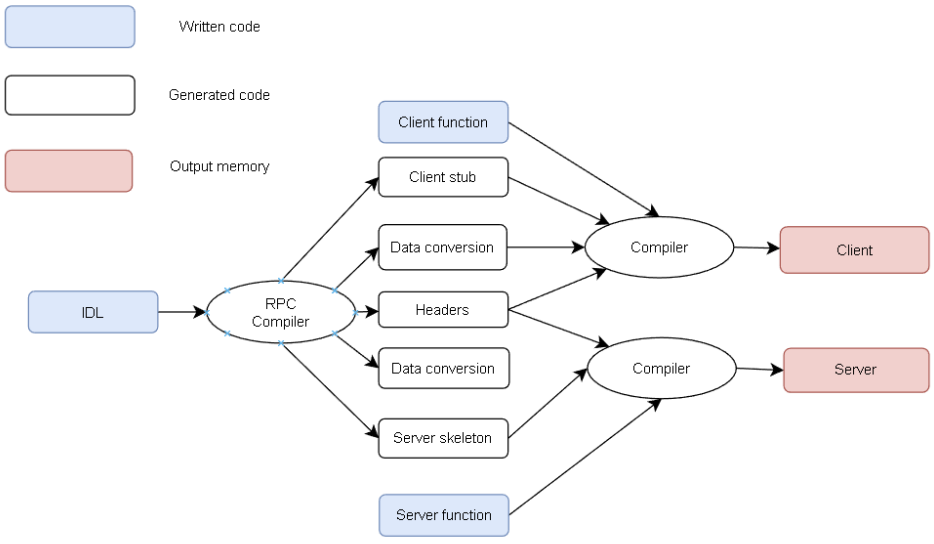

Figure 3: RPC System Architecture and Compilation Flow

3 File Transfer Implementation

The core implementation involves replacing the two-phase socket protocol with two dedicated RPC calls.

3.1 Code Snippet: Client Side (RPC Call Abstraction)

The client logic no longer deals with the socket file descriptor after initialization. It uses the ‘clnt‘ handler (Client Stub) to make remote calls. The ‘send()‘ call is replaced by a loop calling the remote procedure.

```
// CLIENT: Uses RPC calls to abstract network activity

// 1. Initialize Client Stub (Abstraction of socket() and connect())
clnt = clnt_create(server_hostname, FILE_TRANSFER_PROG, FILE_TRANSFER_VERS, "tcp");
if (clnt == NULL) { clnt_pcreateerror(server_hostname); exit(1); }

// 2. RPC Call 1: Send filename (Replaces first send())
name_request.filename = (char *)filename_to_send;
status_name = send_filename_1(&name_request, clnt);

// 3. Loop: Read from disk, Marshal, and RPC Call 2
while ((bytes_read = read(file_fd_read, data_buffer, BUFFER_SIZE)) > 0) {
    // Replaces send(data): Copy data into RPC struct and call remote function
    memcpy(data_chunk.content, data_buffer, bytes_read);
    data_chunk.bytes_read = (int)bytes_read;

    // Remote Procedure Call!
    status_data = send_data_chunk_1(&data_chunk, clnt);

    if (status_data == NULL || status_data->success == 0) { break; }
}

// 4. Send EOF signal via RPC and cleanup
data_chunk.bytes_read = 0;
send_data_chunk_1(&data_chunk, clnt); // Final call to signal EOF
clnt_destroy(clnt); // Replaces close(client_fd)
```

Figure 4: Client Code Snippet: Sender Logic using RPC

3.2 Code Snippet: Server Side (Implementation Logic)

The server code is now cleaner, focusing purely on file handling. The recv() calls and the accept() loop are gone, handled by the Server Skeleton.

```
// SERVER: Implementation of the remote procedure

// Procedure to handle data chunks (called by the Server Skeleton)
transfer_status *
send_data_chunk_1_svc(file_data *argp, struct svc_req *rqstp)
{
    static transfer_status result;
    // ... (logic checks)

    // Writes the Unmarshaled data to disk (Replaces write(file_fd_write, buffer, byte
    if (argp->bytes_read > 0) {
        if (write(current_file_fd, argp->content, argp->bytes_read) < 0) {
            // Error handling
            // ...
        }
    }

    // argp->bytes_read == 0 is the EOF signal from the final RPC call
    if (argp->bytes_read == 0) {
        printf("File received successfully via RPC. Closing file handle.\n");
        close(current_file_fd);
        current_file_fd = -1;
    }

    result.success = 1;
    return &result;
}
```

Figure 5: Server Code Snippet: Receiver Logic (RPC Implementation)