University Of Science And Technology Of Hanoi

ICT Department

LAB REPORT

Title: Practical Work 2 – RPC File Transfer

Subject: Distributed Systems (DS2026)

Student: Le Nam Anh

StudentID: 23BI14029

Major: Cyber Security

1 Introduction

Following the implementation of the TCP-based file transfer system in Practical Work 1, this project focuses on upgrading the communication mechanism to **Remote Procedure Call (RPC)**.

The objective is to abstract the low-level network socket details and allow the client to invoke file-saving operations on the server as if they were local function calls. The system is implemented in Python using the standard `xmlrpc` library. It maintains the core functionality of transferring files while demonstrating the architectural shift from message passing (streaming bytes) to procedure invocation.

2 RPC Service Design

2.1 Service Definition

Unlike the socket implementation where we defined a custom binary protocol (header + payload), the RPC implementation relies on exposing a specific method to the client.

We designed a remote function named `save_file` with the following signature:

- **Function Name:** `save_file`
- **Parameters:**
  - `filename` (String): The name of the file to be saved on the server.
  - `binary_data` (XML-RPC Binary): The actual file content wrapped in a binary object to ensure safe transmission over HTTP.
- **Return Value:** String ("OK" or "ERROR").

2.2 Mechanism

The communication relies on the XML-RPC standard:

1. **Client Stub (Proxy):** The client uses a `ServerProxy` to wrap the `save_file` call into an XML request.
2. **Transport:** The request is sent via HTTP POST to the server.
3. **Server Skeleton:** The server receives the XML, parses the parameters, and executes the actual Python `save_file` function.

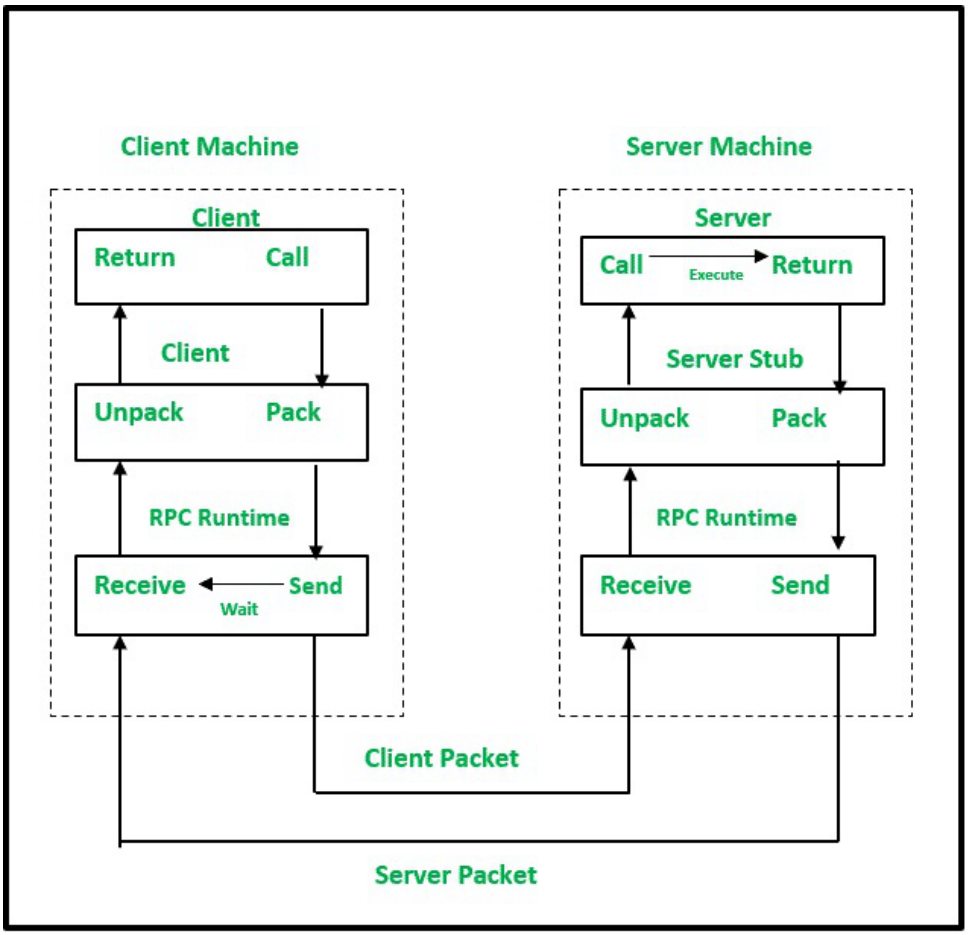

Figure 1: RPC Communication Mechanism Diagram

3 System Organization

3.1 Architecture

The system utilizes a Client-Server architecture, simplified by the RPC middleware:

RPC Server:

- Initialized using `SimpleXMLRPCServer`.
- Registers the `save_file` function.
- Listens on a specific port (e.g., 5000) for incoming HTTP requests.
- Handles file I/O operations and security checks.

RPC Client:

- Connects to the server using the server's URL (`http://<IP>:<PORT>`).
- Reads the local file in binary mode.
- Invokes the remote method `proxy.save_file()`.

3.2 Workflow

1. **Server Startup:** The server starts and waits for RPC calls.
2. **File Preparation:** The client identifies the file and reads its content into memory.
3. **Remote Invocation:** The client calls the remote function. The `xmlrpc` library handles marshaling.
4. **Execution:** The server receives the data, verifies it is a binary object, creates the output directory, and writes the file.
5. **Response:** The server returns a confirmation string to the client.

4 Implementation

The system is implemented using Python 3. Below are the core snippets demonstrating the transition to RPC.

4.1 Server Implementation

The server code focuses on logic rather than socket management. It exposes the `save_file` function.

```
import os
import sys
from xmlrpc.server import SimpleXMLRPCServer
import xmlrpc.client

UPLOAD_DIR = "received_files"

def save_file(filename, binary_data):
    if not isinstance(binary_data, xmlrpc.client.Binary):
        return "ERROR: Data must be xmlrpc.client.Binary"

    if not os.path.exists(UPLOAD_DIR):
        os.makedirs(UPLOAD_DIR)

    safe_filename = os.path.basename(filename)
    dest_path = os.path.join(UPLOAD_DIR, safe_filename)

    with open(dest_path, "wb") as f:
        f.write(binary_data.data)

    print(f"Saved: {safe_filename} ({len(binary_data.data)} bytes)")
    return f"OK: Saved {safe_filename}"

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python server.py <port>")
        sys.exit(1)

    port = int(sys.argv[1])

    print(f"Server listening on 0.0.0.0:{port}")
    server = SimpleXMLRPCServer(('0.0.0.0', port), allow_none=True)
```

```
    server.register_function(save_file, 'save_file')

    try:
        server.serve_forever()
    except KeyboardInterrupt:
        print("\nServer stopped.")
```

Listing 1: Server-side Code (server.py)

4.2 Client Implementation

The client uses `xmlrpc.client.ServerProxy` to interact with the server.

```
import xmlrpc.client
import os
import sys

def send_file(server_ip, server_port, filepath):
    if not os.path.isfile(filepath):
        print(f"Error: File '{filepath}' not found.")
        return

    filename = os.path.basename(filepath)

    server_url = f"http://{server_ip}:{server_port}"
    print(f"Connecting to RPC Server at {server_url}")

    try:
        proxy = xmlrpc.client.ServerProxy(server_url)

        with open(filepath, "rb") as f:
            file_content = f.read()

        print(f"Sending file: {filename} ({len(file_content)} bytes)...")
        response = proxy.save_file(filename, xmlrpc.client.Binary(file_content))
        print(f"Server replied: {response}")

    except ConnectionRefusedError:
        print("Error: Could not connect to the server. Is it running?")
    except Exception as e:
        print(f"RPC Error: {e}")

if __name__ == "__main__":
    if len(sys.argv) < 4:
        print("Usage: python client.py <server_ip> <server_port> <file_path>")
        print("Example: python client.py 127.0.0.1 5000 test.txt")
        sys.exit(1)

    server_ip = sys.argv[1]
    server_port = int(sys.argv[2])
    filepath = sys.argv[3]
    send_file(server_ip, server_port, filepath)
```

Listing 2: Client-side Code (client.py)

5 Testing and Results

The system validation was conducted locally using two separate terminal windows to simulate the interaction between the RPC Server and the Client. The procedure consisted of the following steps:

- **Server Initialization:** First, the server script was executed to bind to port 5000 and await incoming XML-RPC calls.

  ```
  python server.py 5000
  ```

  Observation: The console confirmed the service was active with the message `Server listening on 0.0.0.0:5000`.

- **Client Execution:** In a second terminal, the client application was triggered to upload a specific file named `test.txt` to the local server (127.0.0.1).

  ```
  python client.py 127.0.0.1 5000 test.txt
  ```

- **Server-side Verification:** Upon receiving the remote call, the server logged a standard HTTP transaction: `"POST /RPC2 HTTP/1.1" 200`. This status code confirms a successful handshake. The server output also indicated that the file was correctly written to the disk with a size of **213 bytes**.

- **Client-side Confirmation:** The client terminal displayed the server's return message: `OK: Saved test.txt`, confirming the operation was completed without errors. The integrity of the transferred file was manually verified by checking the `received_files` directory.

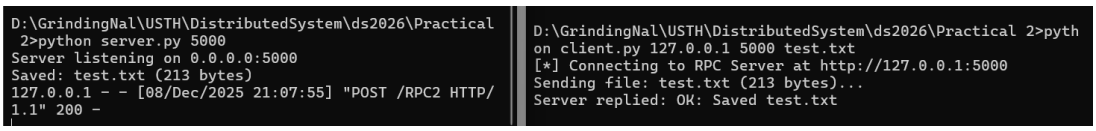

Figure 2: Server and Client Execution Log