University Of Science And Technology Of Hanoi
ICT Department

LAB REPORT

Title: Practical Work 2 – TCP File Transfer

Subject: Distributed Systems (DS2026)

Student: Nguyen Phuong Anh

StudentID: 22BA13020

Major: CS

Abstract

In Practical Work 1, I implemented a simple file transfer system using TCP sockets in Python. The objective of Practical Work 2 is to upgrade this system to use a Remote Procedure Call (RPC) service. In this report, I describe the design of my RPC service, how I organise the system, the implementation details, and how I tested the file transfer.

1 Introduction

The main goal of this practical work is to transform a raw TCP file transfer system into a higher level *RPC-based* service. Instead of manually sending and receiving bytes through sockets, the client now invokes a remote function on the server, which is responsible for storing the file on the server side.

In my solution I use Python's built-in `xmlrpc` library:

- `SimpleXMLRPCServer` on the server side;
- `ServerProxy` on the client side.

This allows me to focus on the application logic while the RPC framework handles message formatting and transport.

2 Design of the RPC Service

2.1 Overview

The RPC service exposes one main remote procedure:

- `upload_file(filename, data)`: receives the name of the file and its content as a byte array, and saves it on the server.

Figure 1 shows the overall design of the system.
When the user wants to upload a file:

1. The client reads the file from disk.
2. The client calls `upload_file()` on the RPC proxy.
3. The RPC framework encodes the parameters and sends a request to the server.
4. The server decodes the request and executes `upload_file()` locally.
5. The server writes the file to its `uploads/` directory and returns a status message.
6. The client displays the status message to the user.

3 Organization of the System

The project is organised as follows:

- `rpc_server.py` – RPC server that exposes the `upload_file()` method.
- `rpc_client.py` – RPC client that reads a local file and sends it to the server.
- `uploads/` – directory on the server where received files are stored.
- `rpc_evidence.png` – screenshot showing the server and client running successfully (Figure 2).

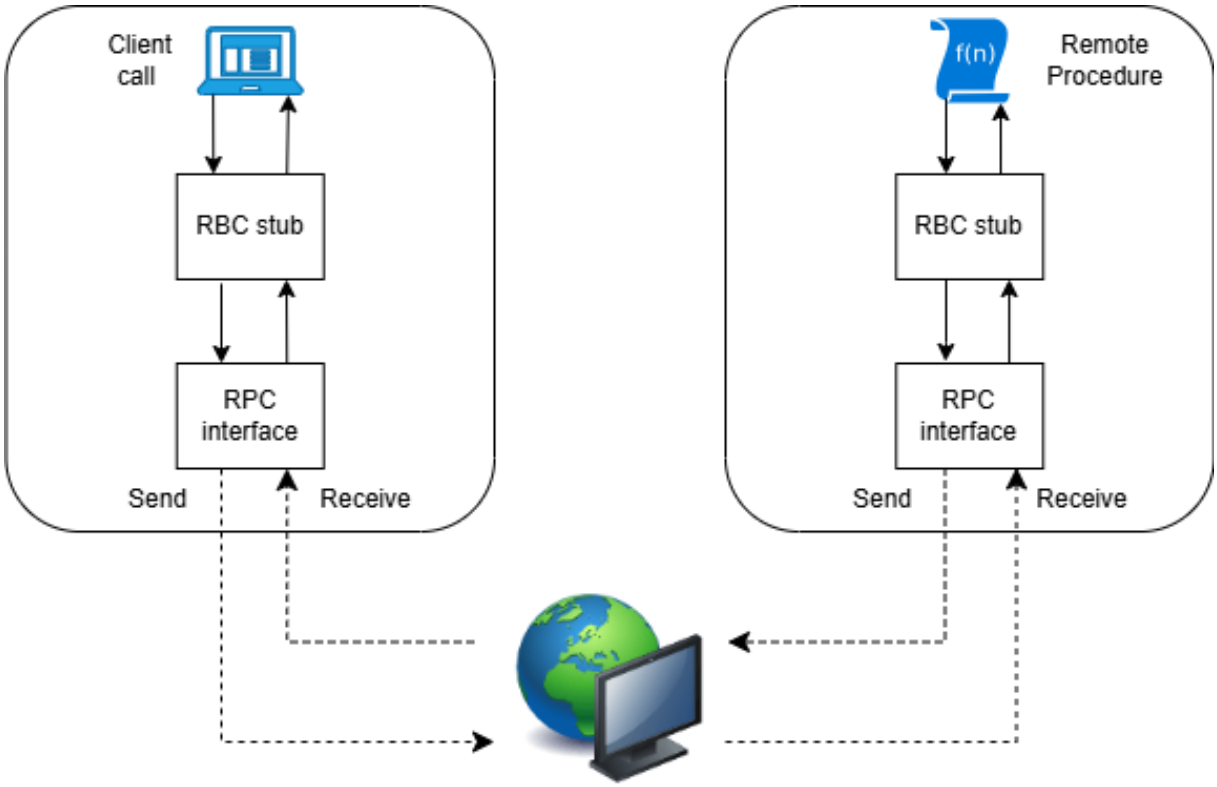

Figure 1: Conceptual design of the RPC file transfer service.

4 Implementation of the File Transfer

4.1 Server Code Snippet

Listing 1 shows the most important part of the RPC server implementation.

Listing 1: RPC server implementation.

```
#!/usr/bin/env python3
import sys
import os
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
from xmlrpc.client import Binary

UPLOAD_DIR = "uploads"

class RequestHandler(SimpleXMLRPCRequestHandler):
    # Allow RPC calls only on /RPC2 (default path)
    rpc_paths = ("/RPC2",)

def upload_file(filename, data):
    """
    RPC method: receive file from client and store it in UPLOAD_DIR.
    filename: str
    data: xmlrpc.client.Binary
    """
    if not isinstance(data, Binary):
        return "ERROR: data must be xmlrpc.client.Binary"

    os.makedirs(UPLOAD_DIR, exist_ok=True)
```

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

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

    return f"OK: saved {safe_name} ({len(file_bytes)} bytes)"

def main():
    if len(sys.argv) != 3:
        print(f"Usage: python {sys.argv[0]} <host> <port>")
        sys.exit(1)

    host = sys.argv[1]
    port = int(sys.argv[2])

    with SimpleXMLRPCServer((host, port),
                            requestHandler=RequestHandler) as server:
        server.register_introspection_functions()
        server.register_function(upload_file, "upload_file")

        print(f"[+] RPC server listening on {host}:{port}")
        print(f"[+] Files will be stored in: {os.path.abspath(UPLOAD_DIR
            )}")
        server.serve_forever()

if __name__ == "__main__":
    main()
```

4.2 Client Code Snippet

Listing 2 shows the corresponding client implementation.

Listing 2: RPC client implementation.

```
#!/usr/bin/env python3
import sys
import os
from xmlrpc.client import ServerProxy, Binary

def main():
    if len(sys.argv) != 4:
        print(f"Usage: python {sys.argv[0]} <server_host> <server_port>
            <file_path>")
        sys.exit(1)

    server_host = sys.argv[1]
    server_port = int(sys.argv[2])
    file_path   = sys.argv[3]

    if not os.path.isfile(file_path):
        print(f"[!] File not found: {file_path}")
        sys.exit(1)

    filename = os.path.basename(file_path)
```

```
    with open(file_path, "rb") as f:
        data = f.read()

    url = f"http://{server_host}:{server_port}/RPC2"
    proxy = ServerProxy(url)

    print(f"[+] Sending file {filename} ({len(data)} bytes)")
    result = proxy.upload_file(filename, Binary(data))
    print(f"[+] Server replied: {result}")

if __name__ == "__main__":
    main()
```

5 Testing and Evidence

To test the system, I performed the following steps:

1. Started the RPC server on port `8000`:

   ```
   python3 rpc_server.py 0.0.0.0 8000
   ```

2. On another terminal, executed the client to send `somefile.txt`:

   ```
   python3 rpc_client.py 127.0.0.1 8000 somefile.txt
   ```

3. The server printed a `POST /RPC2 HTTP/1.1` request and a confirmation message.
4. The client printed the reply `"OK: saved somefile.txt (11 bytes)"` and the file appeared in the `uploads/` directory.

Figure 2 shows a screenshot of both terminals running successfully.

Figure 2: Evidence of successful RPC file transfer (server and client).

6 Conclusion

In this practical work I successfully upgraded my previous TCP file transfer program into an RPC-based file transfer system using Python's `xmlrpc` library. The new design is easier to use, because the client can simply call a remote procedure instead of dealing with low level socket operations. This exercise helped me understand how RPC abstracts network communication and how distributed systems can be structured around remote method calls.