Practical Work 2: RPC File Transfer

Phạm Tưởng Ngạn
Distributed Systems

3rd December

1 Goal

The objective of this practical work is to migrate a file transfer system from low-level TCP sockets to a Remote Procedure Call (RPC) architecture. This abstraction simplifies the development process by allowing the client to call functions on the server directly, hiding the complexities of packet framing and socket management.

2 Protocol Design

Unlike the previous TCP implementation where we defined a custom header format (length/-filename/size), the RPC implementation relies on a defined **Service Interface**.

I implemented a single Remote Procedure:

- **Method:** `upload_chunk`
- **Parameters:**
  1. `filename` (String): The name of the file being transferred.
  2. `data` (Binary): A chunk of the file content.
- **Return:** `Boolean` (Confirmation of receipt).

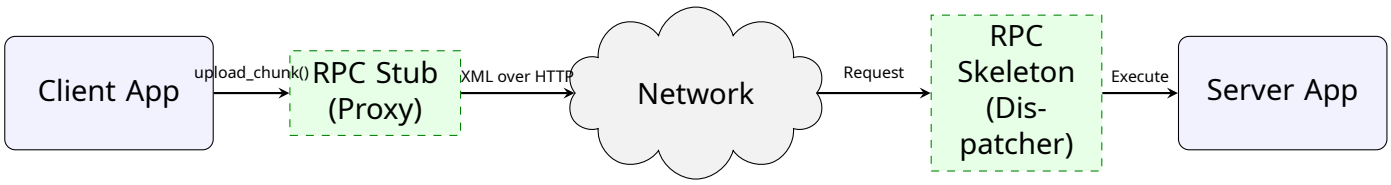

Figure 1: RPC Architecture Abstraction

3 System Organization

The system uses Python's standard `xmlrpc` library. This was chosen for its "batteries-included" nature, requiring no external pip packages.

3.1 Flow of Operations

1. **Server Startup:** The server binds to a port and registers the `upload_chunk` function.
2. **Connection:** The client creates a `ServerProxy` pointing to the server's URL.
3. **Transfer Loop:** The client reads the file in 4KB blocks. For each block, it wraps the raw bytes in an `xmlrpc.client.Binary` object and invokes the remote method.
4. **Writing:** The server receives the call, extracts the bytes, and appends them to the file on the disk.

4 Implementation Code

4.1 Server Implementation

The server uses 'SimpleXMLRPCServer'. The critical part is handling the 'binary_data' object, which is not a simple string but an XML-RPC wrapper around bytes.

```
from xmlrpc.server import SimpleXMLRPCServer

def run_server(port):
    server = SimpleXMLRPCServer(('0.0.0.0', port), allow_none=True)

    def upload_chunk(filename, binary_data):
        with open(f"recv_{filename}", 'ab') as f:
            f.write(binary_data.data)
        return True

    server.register_function(upload_chunk, "upload_chunk")
    server.serve_forever()
```

Listing 1: Server Logic

4.2 Client Implementation

The client connects via a proxy. A key implementation detail is the use of `xmlrpc.client.Binary(chunk)`. Without this wrapper, the XML serializer attempts to treat the file content as a string, which fails for binary files (like images) containing non-printable characters.

```
import xmlrpc.client

def run_client(host, port, filepath):
    filename = os.path.basename(filepath)
    proxy = xmlrpc.client.ServerProxy(f'http://{host}:{port}')

    with open(filepath, 'rb') as f:
        while True:
            chunk = f.read(4096)
            if not chunk: break

            wrapped = xmlrpc.client.Binary(chunk)
            proxy.upload_chunk(filename, wrapped)
```

Listing 2: Client Logic