Practical Work 2 - RPC File Transfer (Java RMI)

Tran Hai Nam

December 2025

1 Introduction

Building on the TCP file transfer from Practical Work 1, this lab reformulates the file transfer problem using RPC, specifically Java RMI. The main ideas are:

- raise the abstraction level: instead of manually managing sockets and headers, the programmer calls remote methods (`upload`, `download`) as if they were local;
- practice key RPC concepts: remote interfaces, stubs and skeletons, and a naming or registry service;
- compare the trade-offs between low-level socket programming and higher-level RPC.

2 Objectives

- Design an RPC service that supports 1-to-1 file transfer with two basic operations: upload and download.
- Implement the service using Java RMI with clear and concise code.
- Build client and server programs that interact through the RMI registry.

3 Service Design

3.1 Remote Interface

The service is defined by the remote interface `FileService`:

- `void upload(String name, byte[] data)`: receives the file content as a byte array and stores it on the server under the name `name`.
- `byte[] download(String name)`: returns the content of the file with the given name; throws `FileNotFoundException` if it does not exist.

The interface extends `java.rmi.Remote` and its methods declare `RemoteException`, as required by Java RMI.

3.2 System Components

- **RMI Server**:
  - creates an instance of `FileServiceImpl` with a configurable base directory (default `received_files_rpc`);
  - starts (or creates) the RMI registry on a configurable port (default `1099`);

- registers the service with `registry.rebind("FileService", service)`.

- **RMI Client**:
  - connects to the registry via `LocateRegistry.getRegistry(host, port)`;
  - looks up the service using `registry.lookup("FileService")` to obtain a `FileService` stub;
  - invokes `upload` or `download` as if calling local methods.

3.3 Command-Line Interface

The client supports two main commands:

- Upload:

```
java -cp rpc RpcClient upload <file_path> [remote_name] [host] [port]
```

- Download:

```
java -cp rpc RpcClient download <remote_name> [host] [port] [
   output_path]
```

Default values:

- `host = 127.0.0.1`
- `port = 1099`
- if `remote_name` is omitted for upload, the local filename is used;
- if `output_path` is omitted for download, the remote name is used in the current directory.

4 Implementation

4.1 FileService.java

```
public interface FileService extends Remote {
    void upload(String name, byte[] data) throws RemoteException;
    byte[] download(String name) throws RemoteException,
       FileNotFoundException;
}
```

This interface acts as the contract between client and server and is shared on both sides.

4.2 FileServiceImpl.java

`FileServiceImpl` extends `UnicastRemoteObject` and implements `FileService`:

- stores the base directory `baseDir` as a `Path`;
- in `upload`:
  - normalizes the filename using `Paths.get(name).getFileName()` to drop any directory paths;
  - ensures the base directory exists (`Files.createDirectories`);

  - writes the `byte[]` data to the target file and logs the number of bytes stored;
- in `download`:
  - checks that the file exists and is a regular file;
  - reads all bytes with `Files.readAllBytes` and returns them;
  - throws `FileNotFoundException` if the file does not exist.

4.3 RpcServer.java

`RpcServer` is responsible for starting and exposing the RPC service:

- parses command-line arguments:
  - `output_dir` (default `received_files_rpc`);
  - `registry_port` (default `1099`);
- creates a `FileServiceImpl` using the chosen output directory;
- creates an RMI registry with `LocateRegistry.createRegistry(port)`;
- binds the service with `registry.rebind("FileService", service)`;
- prints status messages so the user knows the service is ready.

4.4 RpcClient.java

`RpcClient` provides the CLI interface for upload and download:

- parses arguments to determine the mode (`upload` or `download`);
- uses a helper method `lookup(host, port)`:
  - calls `LocateRegistry.getRegistry(host, port)` to get the registry;
  - calls `registry.lookup("FileService")` to obtain the `FileService` stub;
- **Upload**:
  - reads the local file into a `byte[]` using `Files.readAllBytes` (suitable for small and medium files);
  - calls `service.upload(remoteName, data)`;
  - logs the local filename, remote name, and number of bytes sent;
- **Download**:
  - calls `service.download(remoteName)` to obtain `byte[]` data;
  - ensures the target directory exists (creates it if necessary);
  - writes `data` to the chosen `output_path`;
  - logs the remote name, local path, and number of bytes received.

5 Build and Run

- Compile (from the project root):

```
javac rpc/*.java
```

- Run the RMI server:

```
java -cp rpc RpcServer [output_dir] [registry_port]
```

- Run the client using the commands from Section 3.3.

6 Evaluation and Comparison with TCP Socket

6.1 Advantages of the RPC Version

- **Communication abstraction**: client and server code focus on `upload` and `download` semantics instead of low-level socket operations and custom headers.
- **Clear interface**: the `FileService` interface is an explicit contract, making the system easier to understand from the API alone.
- **Automatic marshalling**: RMI handles parameter marshalling and unmarshalling and the underlying network transport.

6.2 Limitations

- **Memory usage**: this implementation loads the entire file into a `byte[]` on both client and server; it is not suitable for very large files.
- **Environment dependencies**: requires an RMI registry, open ports, and appropriate security settings.
- **Less low-level control**: fine-grained control over buffers, timeouts, and custom wire formats is harder than with raw sockets.

6.3 Comparison with Practical Work 1

- Practical Work 1 (TCP):
  - fine-grained control over protocol, headers, and buffering;
  - streams file content without loading it entirely into memory.
- Practical Work 2 (RPC):
  - higher-level programming model with remote methods;
  - encourages thinking in terms of a file transfer service rather than packet formats.

The two labs complement each other: Practical Work 1 emphasizes the transport layer (sockets), while Practical Work 2 emphasizes the middleware layer (RPC). Understanding Practical Work 1 helps to see what RMI is doing under the hood.

7 Possible Extensions

- Redesign the interface to support chunked transfers instead of whole-file `byte[]`, for example by splitting the file into multiple smaller RPC calls.
- Add authentication and authorization to restrict who can upload and download files.
- Integrate encryption (or run RMI over a secure channel) to protect file contents.
- Add concurrency controls and detailed logging (which client accessed which file and when).

8 Conclusion

This lab lifts the 1-to-1 file transfer problem from low-level socket programming to an RPC-based design using Java RMI. It illustrates how middleware hides communication details and exposes a simple remote API. Together, Practical Work 1 and 2 provide a complete view of how a distributed file transfer service can evolve from a custom protocol on top of TCP to a higher-level distributed service.