MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DISTRIBUTED SYSTEMS

Assignment 2

Remote Procedure Call (RPC)

A 2.1: The Basics

Ioan Salomie
Marcel Antal
Teodor Petrican

Tudor Cioara
Claudia Daniela Pop

Ionut Anghel
Dorin Moldovan
Ciprian Stan

2018

Contents

1. Introduction

Remote procedure call (RPC) is an important evolutionary step in the continuous development of distributed systems message passing mechanisms.

The idea of RPC is quite simple: develop a **transparent mechanism** that allows the call of a procedure or method to be executed on a different machine from the caller. This mechanism has several advantages:

- the code is located only on one machine and several clients can execute it
- if the code is complex it is enough to have only one machine with powerful hardware resources to execute (the server)
- the programmer on the client side is unaware of the fact that the execution of the RPC is not on the local machine

2. Problem Requirements

Suppose we are requested to create a distributed application with the following requirements:

- A computational expensive algorithm to compute the tax for a car based on its engine capacity.
- The algorithm is run on a remote physical machine because it requires more resources.
- The customers (Remote Clients) want to use the algorithm in order to compute the tax for their cars:
- The data computed by the algorithm is sent back to the clients to be displayed.

3. Problem Analysis

From the problem requirements we notice an important aspect: the algorithm used to compute the tax for the cars **is computational intensive**, thus being unsuited for the clients to run it **locally** on their physical machines.

Consequently, the **chosen solution** will be a **distributed application** having **client-server architecture**. The server, having more physical resources, will run the computational intensive algorithm.

The question remains: What kind of **communication protocol** will be used?

The first choice would be to use the **HTTP protocol defined at A1**. The tax computation could be implemented as a GET method.

However, we can find a problem with this approach: **what happens if we have several methods on the server that need to be called remotely and they cannot be associated to the HTTP methods**?

Of course that we can use **another parameter** in the message to choose the desired method from the server, but this leads to the RPC architecture.

The problem can be decomposed into the following subsystems:

- Communication protocol - RPC
- The server application:
  - Algorithm
  - Remote invocation
  - Communication layer over the network
- The client application:
  - Communication layer over the network
  - Remote invocation

4. Designing a Solution

4.1. General Architecture

We need to create a distributed application over the network. We choose **client-server** software architecture and a request-reply communication paradigm.

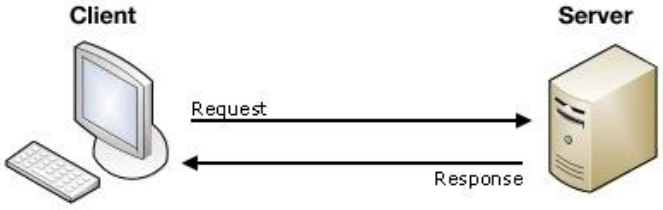

Figure 1. Client-Server software architecture

For the transport layer of the application we use **sockets** [1] that assure a two-way communication between the client and the server, thus allowing us to implement a synchronous request-reply communication method.

4.2. Domain Modeling

The application must compute the tax of a car using a remote procedure call. The client application contains the data regarding the car that will be sent to the server. The car contains the following fields:

- int year – fabrication year
- int engineCapacity – engine size in cmc

Based on this data, the server will compute the tax value after the formula:

$$tax = \left(\frac{engineCapacity}{200}\right) * sum$$

Where sum depends on the engine size:

Engine Size	Sum (in RON)
<1600	8
1601-2000	18
2001-2600	72
2601-3000	144
>3001	290

NOTE:
The formula is a simple formula for this tutorial purpose only. Usually, the method from the server is a computational intensive calculus that requires more physical resources than are available on the client.

4.3. Communication Mechanism

The remaining issue is to define the message structure that will allow a **remote method invocation, or remote procedure call (RPC).**

What happens when a procedure or method is invoked? During the invocation of a method, the parameters are stored on the stack and the control is passed to the code section located at the address mapped to the procedure name. What is important to notice is that a procedure is defined by its name (that maps to an address in the memory where the actual code is located) and its parameters.

In an OOP environment, we have a **Remote Method Invocation** technique that allows invoking a method from a remote object. In this case, we must know the object address (or name), the method name and its parameters. Furthermore, in a distributed environment, in order to identify a remote object, besides knowing **the object name** (and implicitly its memory address) we must also know **the address of the server** where it is located.

Basically, this **RPC/RMI** technique introduces an intermediate layer between the method call and its actual execution, mainly because the **method call** happens on the **client** and the **execution** on the **server**.

In order for the client to make the call, it must know the signature of the method (name, parameters and return type). The signature of a method is defined in OOP languages in an **interface**. Thus, we might assume that the methods from the server are defined in an **interface**. This client has also a reference to this **interface**, thus knowing the method signature that will be called. Knowing this, the architecture of the system is the following:

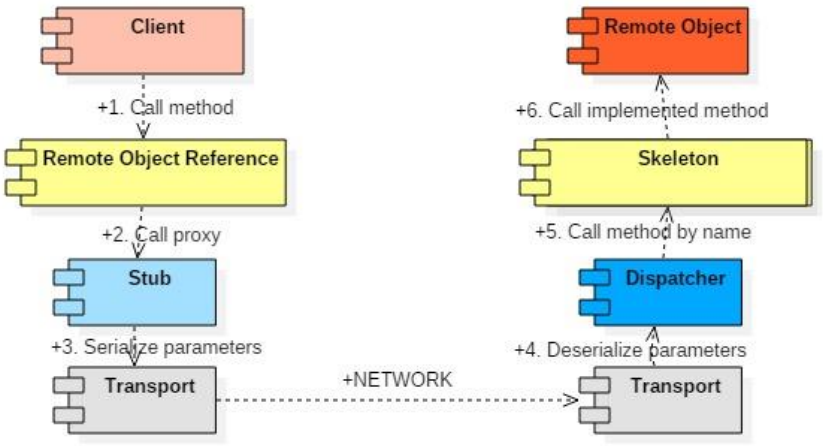

Figure 2. Simple RPC call

The steps involved in calling a remote method are described as follows:

1. Client calls the method

The client application makes a call to a special proxy object that implements the remote interface. The client handles this object as it was a local object implementing the interface. The client calls the desired method.

2. Call forwarded to the proxy

The method call is forwarded to a proxy that has a special implementation of the interface. Instead of implementing the functionality of the methods, this proxy creates a communication mechanism that takes the method's name and parameters and serializes them in order to be sent over the network.

3. Data sent over the network

The data is packed and sent over the network:

- Remote object name (address space)
- Remote object method
- Method parameters

All this information is sent after **serialization.**

4. Server receives data

The server receives the data and de-serializes it and sends them to the **Dispatcher**.

5. Server calls method

The **Dispatcher** is responsible for calling the method from the **Skeleton** that is the **interface** exposed by the **remote object**.

6. Server executes method

The server executes the method with the parameters send from the client. It computes the return value of the method and serializes the result for the client.

7. Result returned to the client

The result is returned to the client, which de-serializes it and returns it to the **Stub** as it has been computed locally.

5. Implementing the Solution

The solution is implemented in 4 different modules:

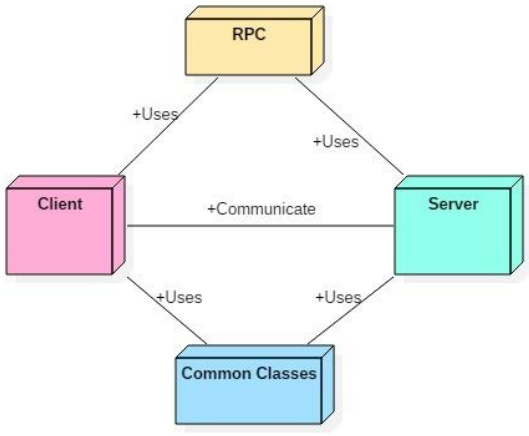

Figure 3. Conceptual architecture

Each module has the following architecture:

- **The Client Application** - contains one package (Communication) with two classes:
  - **ClientStart** - Class which contains the main method. Here, the remote object is invoked after a reference is created.
  - **ServerConnection** – class that contains the sockets connecting the client with the server
- **The Server Application –** contains two packages:
  - **Communication** - contains the server-side communication
  - **Services** – contains the implementation of the remote object
- **The Common Classes –** contain two packages:
  - **Entities** - contains the entities (Car)
  - **ServiceInterface** - contains the definition of the interface exposed by the remote object (Skeleton)
- **RPC** – library that contains the protocol definition. Contains one package (rpc) with 5 classes:
  - **Connection**: Interface specifying the connection of a client to the server. Such a connection must provide a method to send a message to the server, and retrieve the message response.

- **Dispatcher**: Dispatches the call received from the client. It interprets the given Message, gets the correct object from the registry, calls the required method of that object and then bundles and returns a response Message.
- **Message**: Represents the object of communication between the client and the server. It contains all the necessary fields for communication. For example, when the client sends the message to the server, the message contains:
  - the endpoint (in the Registry, which is associated to the remote object)
  - the name of the method to be called
  - the arguments of the method, in order

  When the server replies, it adds the result (return value of the method, or a status message, or an exception) in the arguments array, on the first position.
- **Naming**: Provides a static method to look up for a remote object on the server.
- **Registry**: Provides a mapping of endpoint-object. It is used by the server to specify which object can be remotely used by a client. The client has to identify the object by the endpoint.

In the next sections we will present the functionality of the client and server applications by means of sequence diagrams:

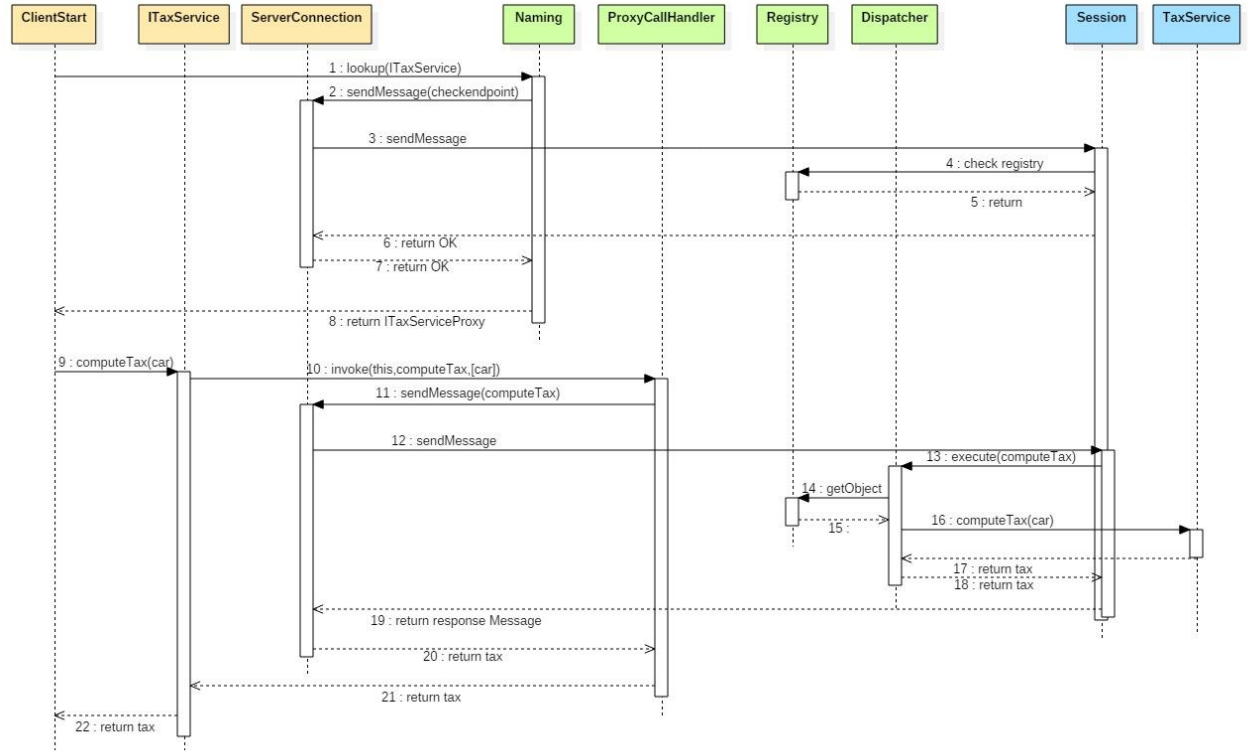

Figure 4. Sequence diagram of an RPC call

A RPC call has the following steps:

1. The Server registers the new object in the Registry
2. **Client** looks up for remote object by name
3. The RPC **Naming** creates a special message that will check if there exists such an object on the server *(package rpc, class Naming, method lookup, lines 39-43)*
4. The RPC **Naming** sends the message to the server using a **ServerConnection** *(package rpc, class Naming, method lookup, line 43)*
5. The **Server** receives the message, and checks in the rpc **Registry** if there exists such an object (**Server**, package communication, class **Session**, method checkEndpoint)
6. If an object exists, OK is returned, otherwise an exception is thrown.
7. If an object exists, OK is returned, otherwise an exception is thrown.
8. If an object exists, OK is returned, otherwise an exception is thrown.
9. The **Naming** creates dynamically a reference of the **Remote** object using only the interface of the remote object, using the java.lang.reflect.Proxy class.

```
(T) Proxy.newProxyInstance(clazz.getClassLoader(),new Class[]{clazz},new ProxyCallHandler(connection));
```

10. The client calls the remote method
11. When the method is called the call is redirected automatically to the invoke method from the static **ProxyCallHandler** class, from **Namig** class in rpc.
12. This method uses reflection to create a message containing the name and parameters of the RPC (in this case the **computeTax** method with parameter **Car**)
13. The **ServerConnection** sends the message to the server
14. The **Session** receives the message and uses the **Dispatcher** to execute the call
15. The **Dispatcher** uses the **registry** to return a reference of the remote object (line 51)
16. The Dispatcher returns a local reference of the remote object.
17. The method is called using reflection and the method **computeTax** is executed in the local **TaxService** object
18. The tax is returned, and the **Dispatcher** creates a response **Message** with the results
19. The result is returned to the **Session**
20. The message is returned to the client
21. The message is returned to the **ProxyHandler**. The return value is stored in the arguments array of the message, on the first position. It is extracted as an object and is converted automatically to double by the **Proxy**.
22. The tax is returned to the client.

6. Running the Solution

1. Setup GIT and download the project from https://bitbucket.org/utcn_dsrl/ds.handson.assignment2.git
    - Create an empty local folder in the workspace on your computer
    - Right-click in the folder and select **Git Bash**
    - Write commands:
        - git init

  - git remote add origin https://bitbucket.org/utcn_dsrl/ds.handson.assignment2.git
  - git pull origin master
2. Import the project into Eclipse: FILE-> Import->Maven-> Existing Maven Projects-> Browse for project in the folder created at step 1
3. Run the project:
   - Run the ServerStart class from the server module, package communication
   - Run the ClientStart class from the client module, package communication
4. The application computes the tax for two cars whose information is hardcoded in the main method from the client, and displays it in the console. For the second car, because the characteristics are invalid (engine capacity is negative), an exception is thrown.

```
System.out.println("Tax value: " + taxService.computeTax(new Car(2009, 2000)));

System.out.println(taxService.computeTax(new Car(2009, -1)));
```

7. Reinforcement Learning

1. Run and study the project and identify the following features:
   - Message encoding / decoding – method name, parameters
   - Creating remote reference of the object
   - Calling method on the server
2. Add a new field to the car containing the purchasing price (double price). Add a new distributed object with a method that computes the selling price of the car based on the formula:

$$price_{selling} = \begin{cases} price_{purchasing} - \frac{price_{purchasing}}{7} * (2018 - year) & if\ 2018 - year < 7 \\ 0 & otherwise \end{cases}$$

8. Bibliography

1. Java Sockets: https://docs.oracle.com/javase/tutorial/networking/sockets/
2. RPC: https://docs.oracle.com/javase/tutorial/rmi/