CEL 72, DSL, Monsoon 2020

Lab 1: Client Server based program using RPC/RMI

Batch A

Student ID: 2017130007

Name: Vinal Bagaria

**Objective**

Design, implement and test a client-server distributed system that uses RPC/RMI to compute taxes and for cars.

**Problem statement**

Suppose we are requested to create a distributed application for computing a car pollution tax using a computational expensive algorithm that cannot be run on any client machine. Thus, the algorithm is run on a remote physical machine having more resources (the server). The customers (remote clients) want to use the algorithm to compute the tax for their cars by sending data to the server and receiving the computation results to be displayed.

The client application sends the data regarding the car 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 using the following formula:

𝑡𝑎𝑥=(𝑒𝑛𝑔𝑖𝑛𝑒𝐶𝑎𝑝𝑎𝑐𝑖𝑡𝑦/200)∗𝑠𝑢𝑚

where *sum* depends on the engine size from following table: Relation between engine size and specific sum

|  |  |
| --- | --- |
| **Engine Size** | **Sum** |
| <1600 | 8 |
| 1601-2000 | 18 |
| 2001-2600 | 72 |
| 2601-3000 | 144 |
| >3001 | 290 |

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

**Application analysis and design**

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 server will expose a method that must be executed remotely by the client, leading to a remote procedure call technique.

The solution can be decomposed into the following subsystems:

* Communication protocol – remote method invocation between client and server
* The server application
  + Algorithm
  + Remote invocation
  + Communication layer over the network
* The client application
  + Communication layer over the network
  + Remote invocation

**Communication mechanism**

This section defines the message structure that will allow a remote method invocation, or Remote Procedure Call (RPC)*.*

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 Object-Oriented Programming (OOP) environment, we have a Remote Method Invocation (RMI) 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, 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.

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. Considering the above aspects, the system communication flow is shown in following figure:



The steps involved in calling a remote method are described below:

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.

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 to be sent over the network.

Data sent over the network: The data is packed and sent over the network. The following information is serialized: remote object name (address space), remote object method and method parameters.

Server receives data: The server receives the data, de-serializes it and sends it to the Dispatcher.

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

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.

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.

**Application structure and implementation**

The solution is implemented in 4 different modules: Client application, Server application, RPC package that contains the classes for remote communication and the Common Classes for both client and server application. The relation between the modules is presented in following figure



Each module has the following components:

* 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
* Server Application – contains two packages:
  + *Communication* - contains the server-side communication
  + *Services* – contains the implementation of the remote object
* 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 with five 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 from 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 must identify the object at the endpoint.

**Deliverables:**

**Platform Used:**

Python XMLRPC (XML based implementation of RPC)

**Conceptual architecture:**

The conceptual architecture can be visualized as follows:

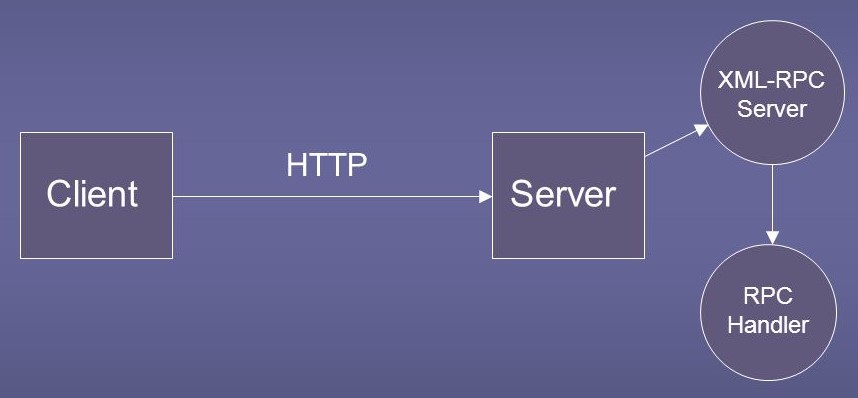


Fig1. Conceptual view of the system.

**1. Client:** The machine/user requesting service from some other remote machine.

**2. HTTP:** The protocol used in XML based Remote Procedure Calls for communication.

**3. Server:** The remote machine which has the computation logic (service) as well as computation capacity (in case of resource intensive functions and services).

**4.** **XML RPC Server:** The actual service responsible for interpreting the XML requests from the client and which acts as a listener.

**5.** **RPC Handler:** The component that consists of the serving class (service), that actually performs the computation.

**Source Files:**

**1. server.py**

from xmlrpc.server import SimpleXMLRPCServer

def tax\_calculation(year, engineCapacity):

if engineCapacity <= 1600:

sum = 8

elif engineCapacity <= 2000:

sum = 18

elif engineCapacity <= 2600:

sum = 72

elif engineCapacity <= 3000:

sum = 144

else:

sum = 290

return (engineCapacity/200)\*sum

server = SimpleXMLRPCServer(("localhost", 9000))

print("Listening on port 9000...")

server.register\_function(tax\_calculation, "tax\_calculation")

server.serve\_forever()

**2. Client.py**

import xmlrpc.client

with xmlrpc.client.ServerProxy("http://localhost:9000/") as proxy:

print("Tax for 4000 is : %s" % str(proxy.tax\_calculation(2020,4000)))

print("Tax for 1500 is : %s" % str(proxy.tax\_calculation(2012,1500)))

**RPC Call Steps:**

Communication between the server and the client.

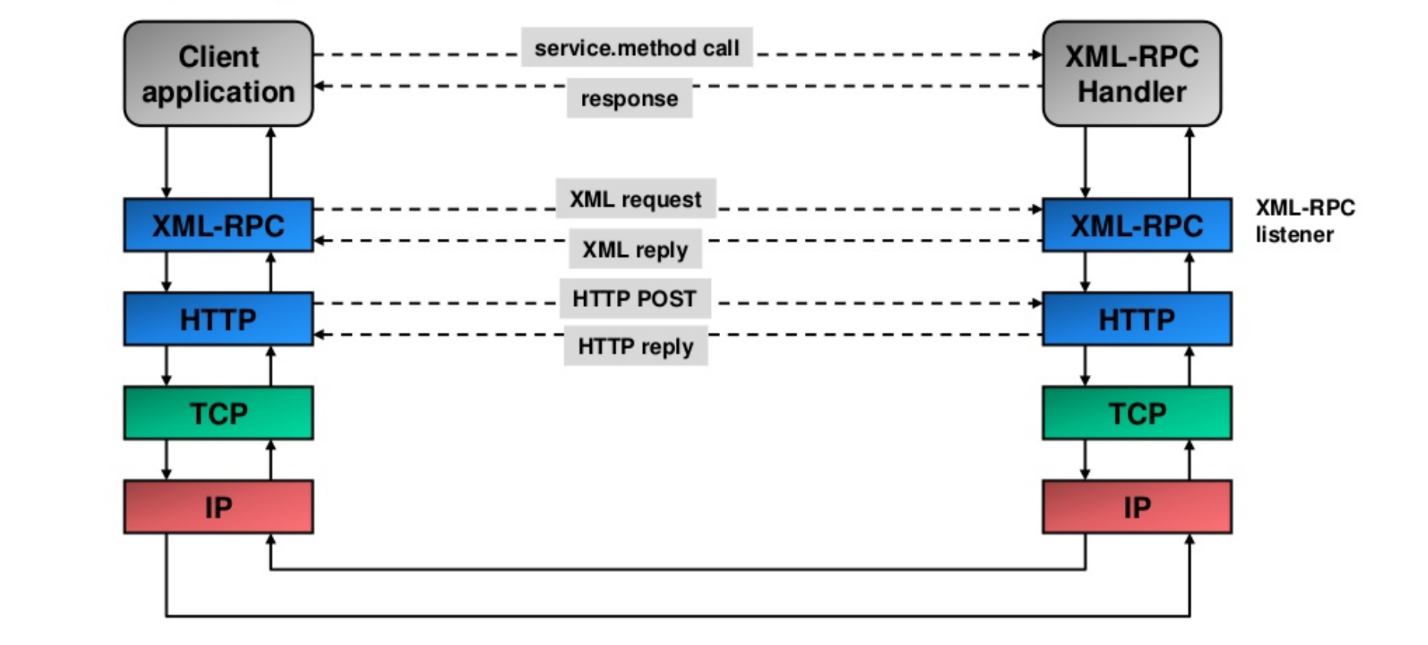


Fig2. Conceptual Transfer of data at different levels

**XML RPC Request:**

1. XML-RPC requests are a combination of XML content and HTTP headers.

2. The XML content uses the data typing structure to pass parameters and contains additional information identifying which procedure is being called, while the HTTP headers provide a wrapper for passing the request over the Web.

3. Each request contains a single XML document, whose root element is a ***methodCall*** element. Each *methodCall* element contains a ***methodName*** element and a *params* element.

4. The *methodName* element identifies the name of the procedure to be called, while the *params* element contains a list of parameters and their values.

5. Each *params* element includes a list of param elements which in turn contain *value* elements.

Example XML-RPC Request:

**1. XML Request:**

<?xml version="1.0"?>

<methodCall>

<methodName>tax\_calculation</methodName>

<params>

<param>

<value><int>2020</int></value>

</param>

<param>

<value><int>4000</int></value>

</param>

</params>

</methodCall>

**2. HTTP Header:**

POST /target HTTP 1.0

User-Agent: Identifier

Host: host.making.request

Content-Type: text/xml

Content-Length: length of request in bytes

\*It's an ordinary HTTP request, with a carefully constructed payload.

**2. XML-RPC Response**

1. If the response is successful - the procedure was found, executed correctly, and returned results - then the XML-RPC response will look much like a request, except that the ***methodCall*** element is replaced by a ***methodResponse*** element and there is no ***methodName*** element

**1. XML Response**

<?xml version="1.0"?>

<methodResponse>

<params>

<param>

<value><double>5800.0</double></value>

</param>

</params>

</methodResponse>

**2. HTTP Header**

HTTP/1.1 200 OK

Date: Thu, 20 Aug 2020 09:10:04 GMT

Server: Apache.1.3.12 (Windows)

Connection: close

Content-Type: text/xml

Content-Length: 124

Like requests, responses are packaged in HTTP and have HTTP headers. All XML-RPC responses use the 200 OK response code, even if a fault is contained in the message.

**Test Procedure for objective Validation:**

1. The Server (Windows Machine) contains the tax calculation method (**tax\_calculation**).

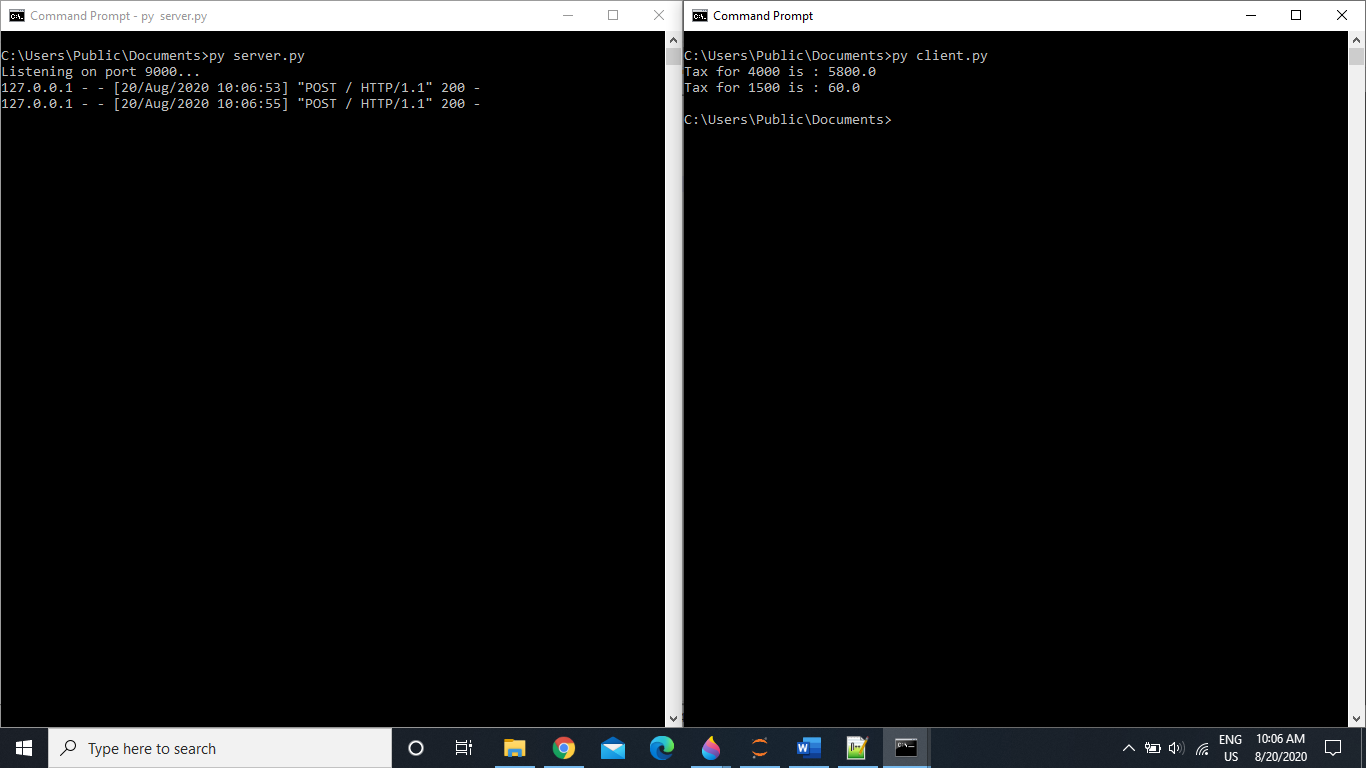


Fig3. RPC for calculation of tax

The Client sends the **Year of Fabrication** and **Engine Capacity** (In XML marshalling) to the server for calculation of taxes with the help of service name **tax\_calculation.**

The server listening on port **9000** received data from the client (In XML unmarshalling). After extracting the data, the server performs the necessary computation on the parameters and returns the result of the computation (In XML) to the client.

**Source Files for Java RMI:**

**TaxCalculation.java**

import java.rmi.Remote;

import java.rmi.RemoteException;

// Creating Remote interface for our application

public interface TaxCalculation extends Remote {

public float calculateTax(int year,int engineCapacity) throws RemoteException;

}

**Implementer.java**

// Implementing the remote interface

public class Implementer implements TaxCalculation {

// Implementing the interface method

public float calculateTax(int year, int engineCapacity){

int sum = 0;

if(engineCapacity <= 1600){

sum = 8;

}

else if( engineCapacity <= 2000){

sum = 18;

}

else if(engineCapacity <= 2600){

sum = 72;

}

else if( engineCapacity <= 3000){

sum = 144;

}

else{

sum = 290;

}

float tax = (engineCapacity/200)\*sum;

return tax;

}

}

**Server.java**

import java.rmi.registry.Registry;

import java.rmi.registry.LocateRegistry;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class Server extends Implementer {

public Server() {}

public static void main(String args[]) {

try {

// Instantiating the implementation class

Implementer obj = new Implementer();

// Exporting the object of implementation class

// (here we are exporting the remote object to the stub)

TaxCalculation stub = (TaxCalculation) UnicastRemoteObject.exportObject(obj, 0);

// Binding the remote object (stub) in the registry

Registry registry = LocateRegistry.getRegistry();

registry.bind("TaxCalculation", stub);

System.err.println("Server ready");

} catch (Exception e) {

System.err.println("Server exception: " + e.toString());

e.printStackTrace();

}

}

}

**Client.java**

import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

import java.util.Scanner;

public class Client {

private Client() {}

public static void main(String[] args) {

try {

// Getting the registry

Scanner scan = new Scanner(System.in);

int year = scan.nextInt();

int capacity = scan.nextInt();

Registry registry = LocateRegistry.getRegistry(null);

// Looking up the registry for the remote object

TaxCalculation stub = (TaxCalculation) registry.lookup("TaxCalculation");

// Calling the remote method using the obtained object

float tax = stub.calculateTax(year,capacity);

System.out.println("Tax for the car with engineCapacity"+ capacity + "is : " + tax);

} catch (Exception e) {

System.err.println("Client exception: " + e.toString());

e.printStackTrace();

}

}

}

**Test Procedure for objective Validation:**

Run the following commands in order for testing.

1. javac \*.java

2. start rmiregistry (A new prompt will open, keep it open till the end)

3. java Server (Run it in separate cmd)

4. java Client (Run it in separate cmd)

**Conclusion:**

An RPC method was used for sending parameters over the network to a remote server responsible for computation and returning the result.

Python XML-RPC library was used for implementing this scenario.

XML-RPC is nothing but transfer of data between two machines using the HTTP Protocol.

**Evaluation**

Grading is performed for this assignment.

|  |  |
| --- | --- |
| **RPC laboratory work grading details Points 5 + Instructor Signature** | **Requirements** |
| 5 | * Client – Server application using Java RMI or Python Remoting with one distributed object and method implemented (tax) (3M) * Documentation (2M) |

|  |  |
| --- | --- |
| **Feedback** | **Yes/No** |
| * Contents in this write up has been useful to perform experiment? * Level of understanding DS has improved? |  |