

CEL 72, DSL, Monsoon 2020

Lab 1: Client Server based program using RPC/RMI

Name: Saumitra Chaskar

Batch A

Student ID: 2017130011

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

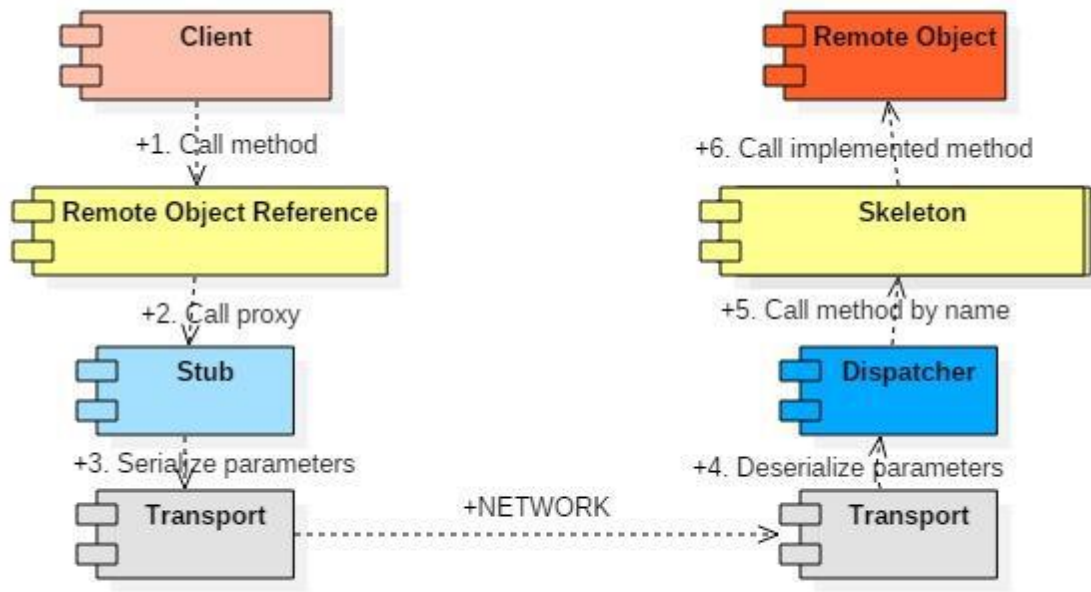
This section defines the message structure that will allow a remote method invocation, or Remote Procedure Call (RPC).

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.

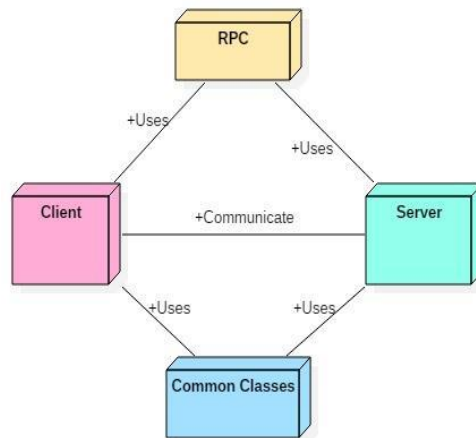
Server receives data: The server receives the data, de-serializes it and sends it to the Dispatcher.
Server calls method: The Dispatcher is responsible for calling the method from the Skeleton that is the interface exposed by the remote object.

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.

Implementation technologies: Choose between JAVA RMI or Python

RPC

Platform Used:

Python XMLRPC (XML based implementation of RPC)

Conceptual architecture:

The conceptual architecture can be visualized as follows:

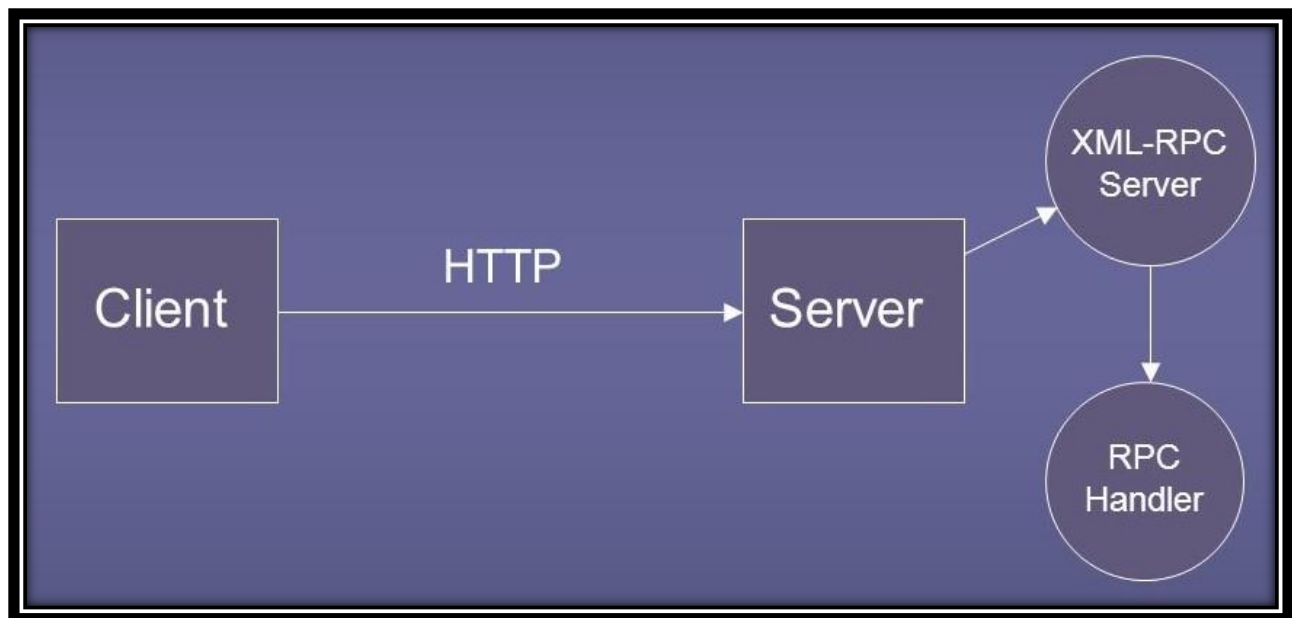


Fig1. Conceptual view of the system.

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 calculate_tax(year, engineCapacity):
    Sum = 1

    if engineCapacity < 1600:
        Sum = 8
    elif 1601 <= engineCapacity <= 2000:
        Sum = 18
    elif 2001 <= engineCapacity <= 2600:
        Sum = 72
    elif 2601 <= engineCapacity <= 3000:
        Sum = 144
    else:
        Sum = 290

    return (engineCapacity*Sum/200)

server = SimpleXMLRPCServer(("192.168.99.1", 8000))
print("Listening on port 8000...")
server.register_function(calculate_tax, "calculate_tax")
server.serve_forever()
```

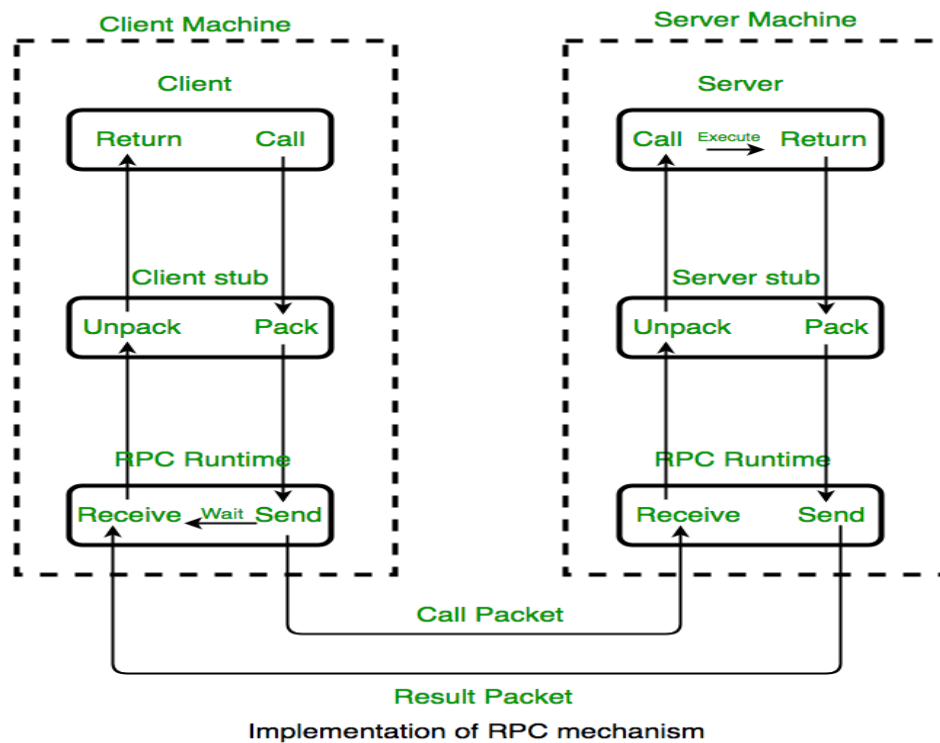
1. Client.py

```
import xmlrpc.client

with xmlrpc.client.ServerProxy("http://192.168.99.1:8000/") as proxy:
    print("Tax for 1800: %s" % str(proxy.calculate_tax(2000,1800)))
    print("Tax for 2700: %s" % str(proxy.calculate_tax(2010,2700)))
```

RPC Call Steps:

Communication between the server and the client.



RPC Call Steps:

1. A client invokes a **client stub procedure**, passing parameters in the usual way. The client stub resides within the client's own address space.
2. The client stub **marshalls(pack)** the parameters into a message. Marshalling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server machine.
4. On the server, the transport layer passes the message to a server stub, which **demarshalls(unpack)** the parameters and calls the desired server routine using the regular procedure call mechanism.
5. When the server procedure completes, it returns to the server stub (**e.g., via a normal procedure call return**), which marshalls the return values into a message. The server stub then hands the message to the transport layer.
6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.
7. The client stub demarshalls the return parameters and execution returns to the caller.

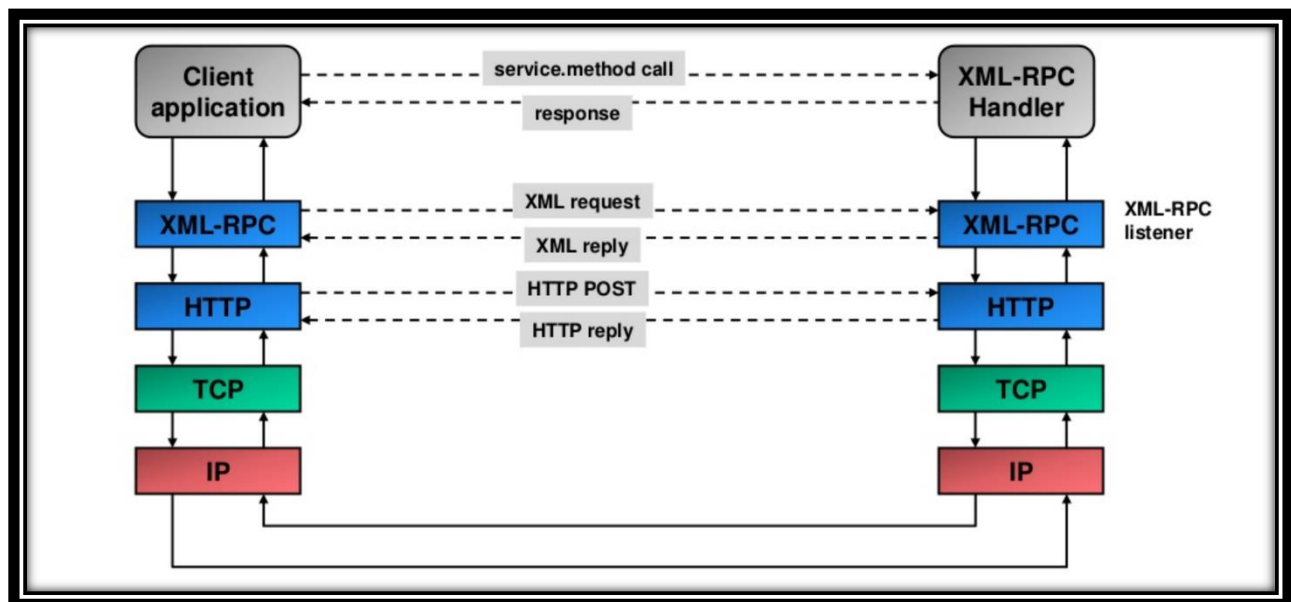


Fig2. Conceptual Transfer of data at different levels

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>calculate_tax</methodName>
  <params>
    <param>
      <value><int>2010</int></value>
    </param>
    <param>
      <value><int>2700</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>1944.0</double></value>
    </param>
```

```
</params>
</methodResponse>
```

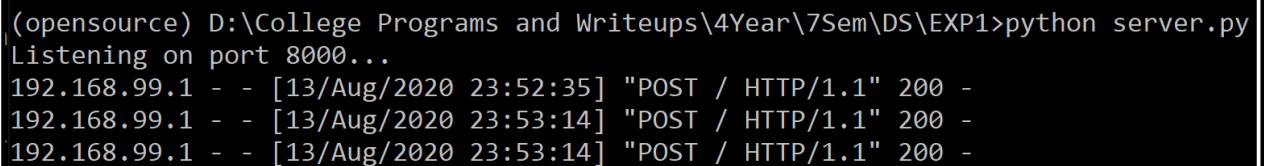
2. HTTP Header

```
HTTP/1.1 200 OK
Date: Sat, 06 Oct 2001 23:20:04 GMT
Server: Apache/1.3.12 (Unix)
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.

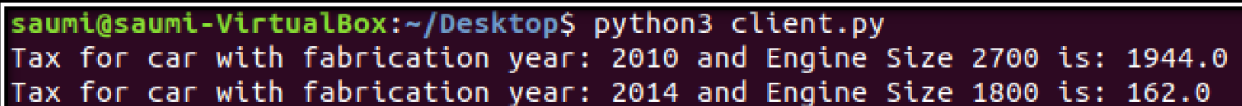
Test Procedure for Objective Validation:

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



```
(opensource) D:\College Programs and Writeups\4Year\7Sem\DS\EXP1>python server.py
Listening on port 8000...
192.168.99.1 - - [13/Aug/2020 23:52:35] "POST / HTTP/1.1" 200 -
192.168.99.1 - - [13/Aug/2020 23:53:14] "POST / HTTP/1.1" 200 -
192.168.99.1 - - [13/Aug/2020 23:53:14] "POST / HTTP/1.1" 200 -
```

Fig3. RPC Server (IP: 192.168.99.1)



```
saumi@saumi-VirtualBox:~/Desktop$ python3 client.py
Tax for car with fabrication year: 2010 and Engine Size 2700 is: 1944.0
Tax for car with fabrication year: 2014 and Engine Size 1800 is: 162.0
```

Fig4. RPC Client (IP: 192.168.99.101)

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 **calculate_tax**.

The server listening on port **8000** 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.

RMI

Platform Used:

Python Pyro : Pyro means PYthon Remote Objects. It is a library that enables you to build applications in which objects can talk to each other over the network, with minimal programming effort.

Conceptual architecture:

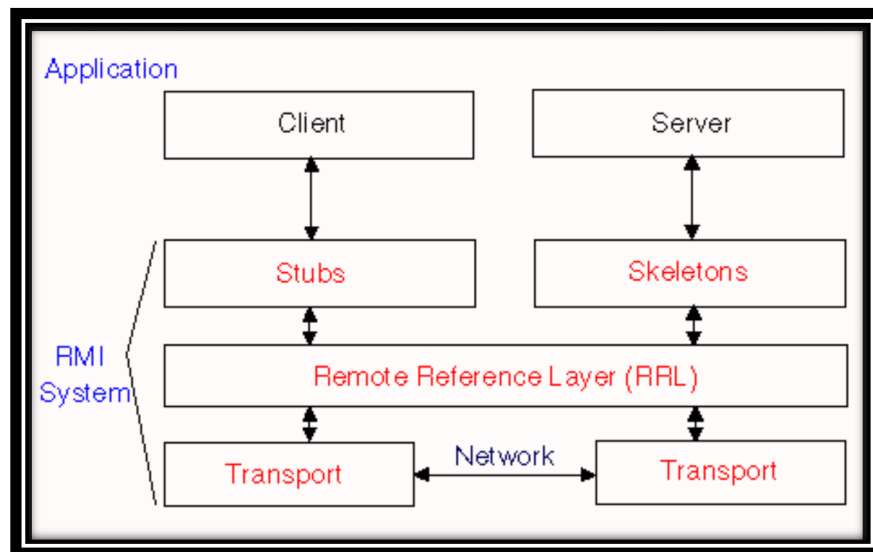


Fig5. RMI Architecture.

Stub – A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.

Skeleton – This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.

RRL(Remote Reference Layer) – It is the layer which manages the references made by the client to the remote object.

Transport Layer – This layer connects the client and the server. It manages the existing connection and also sets up new connections.

Working of an RMI Application

The following points summarize how an RMI application works –

1. When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
2. When the client-side RRL receives the request, it invokes the method of the remote object. It passes the request to the RRL on the server side.
3. The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
4. The result is passed all the way back to the client.

Whenever parameters or objects are to be sent over the network they are bundled into a message and this process is called **Marshalling** while on the server side when the method is to be invoked the parameters are obtained by unbundling the message, this process is called **Unmarshalling**. The same reverse process happens when message is to be passed from the server to the client.

RMI Call Steps: In Pyro

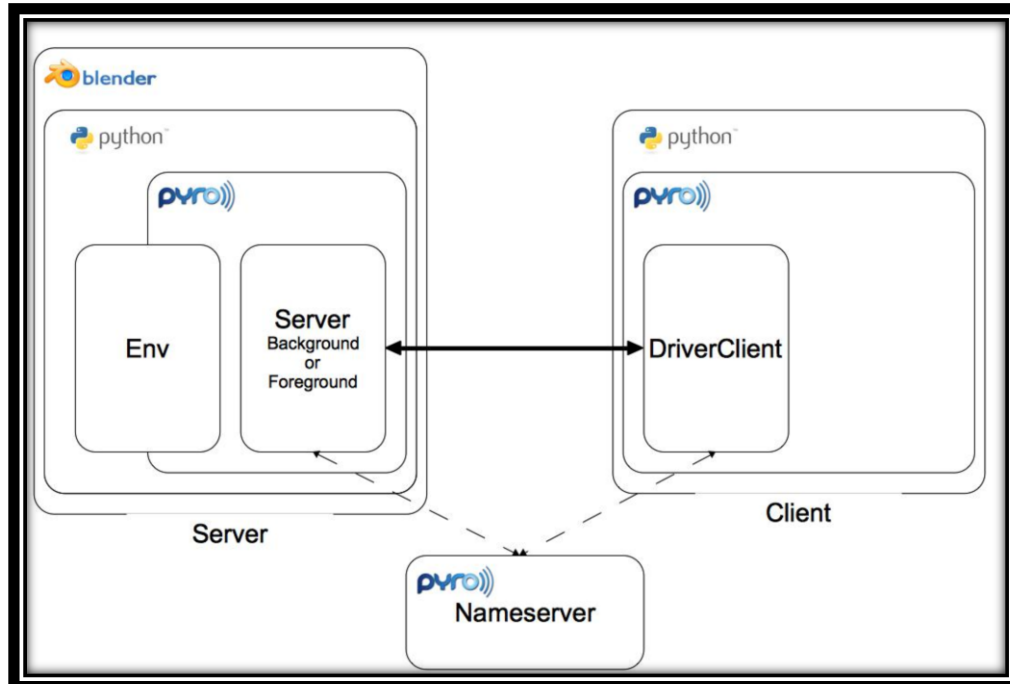


Fig6. Python Pyro RMI Architecture.

The above figure shows how RMI in Pyro is implemented. Blender is an application software on the server side. Any other application software can replace it.

SERVER

1. Objects are published for clients by converting them to Pyro Objects. The Pyro Daemon is created for listening and processing requests as well as publishing the objects.
2. Name server is run on a machine (Usually sever machine).
3. A URI for the object is obtained by the sever once it publishes it or registers it with the daemon. This URI is then registered to the name server under a name which will represent that object.

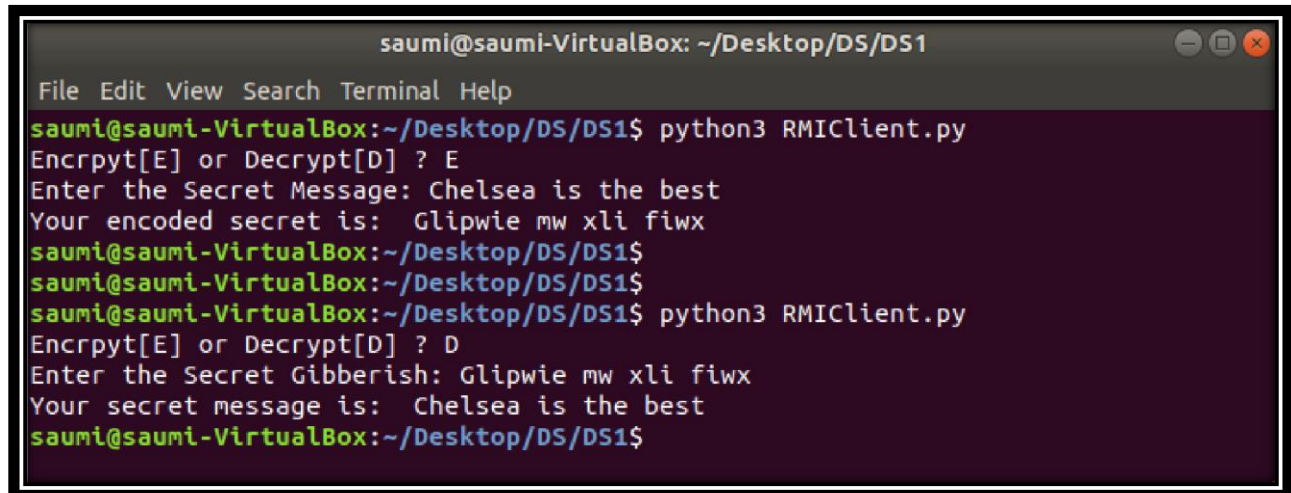
CLIENT

1. The client establishes a connection with any available name server.
2. It creates a proxy object of the actual object on which it wants to invoke a method by providing the domain name corresponding to the URI of the object to the name server.
3. Once the proxy object is obtained the client can invoke any of the available methods available on that particular object.

CALLS

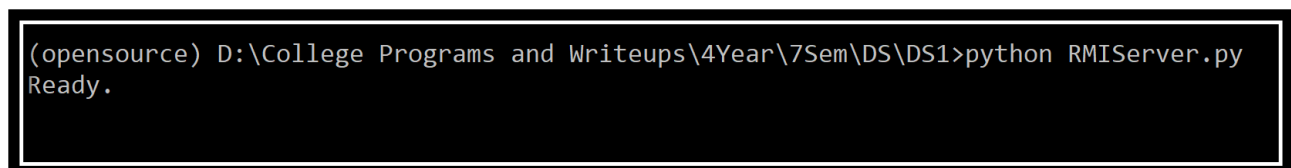
1. Client calls the name server using a domain name for the object.
2. Proxy (Stub) object is created on the client side.
4. Client invokes method on the stub.
5. Parameters are marshalled and are passed to the server. The RRL Layer passes the request to the skeleton.
6. The skeleton executes the code and returns the results of the method and marshals it to the client side.
7. The client unmarshalls and uses the results.

Test Procedure for Objective Validation:

A terminal window titled 'saumi@saumi-VirtualBox: ~/Desktop/DS/DS1' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the execution of 'python3 RMIClient.py'. It prompts for encryption/decryption mode (E for Encrypt, D for Decrypt). In the first run, mode 'E' is chosen, and the message 'Chelsea is the best' is entered, resulting in the encoded secret 'Glipwie mw xli fiwx'. In the second run, mode 'D' is chosen, and the gibberish 'Glipwie mw xli fiwx' is entered, resulting in the decoded secret message 'Chelsea is the best'.

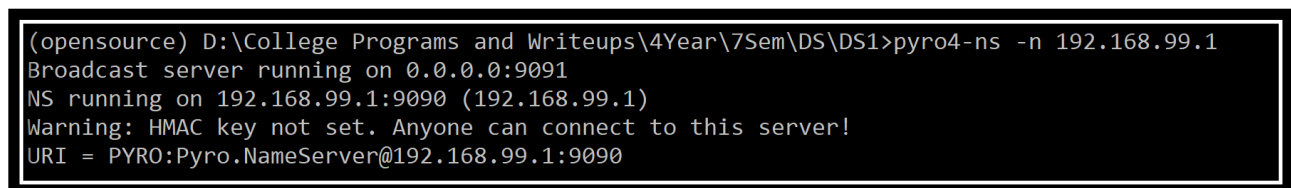
```
saumi@saumi-VirtualBox: ~/Desktop/DS/DS1
File Edit View Search Terminal Help
saumi@saumi-VirtualBox:~/Desktop/DS/DS1$ python3 RMIClient.py
Encrpyt[E] or Decrypt[D] ? E
Enter the Secret Message: Chelsea is the best
Your encoded secret is: Glipwie mw xli fiwx
saumi@saumi-VirtualBox:~/Desktop/DS/DS1$
saumi@saumi-VirtualBox:~/Desktop/DS/DS1$
saumi@saumi-VirtualBox:~/Desktop/DS/DS1$ python3 RMIClient.py
Encrpyt[E] or Decrypt[D] ? D
Enter the Secret Gibberish: Glipwie mw xli fiwx
Your secret message is: Chelsea is the best
saumi@saumi-VirtualBox:~/Desktop/DS/DS1$
```

Fig5. RMI Client (IP: 192.168.99.101)

A terminal window showing the command 'python RMIServer.py' being executed. The output is 'Ready.'.

```
(opensource) D:\College Programs and Writeups\4Year\7Sem\DS\DS1>python RMIServer.py
Ready.
```

Fig6. RMI Server (IP: 192.168.99.1)

A terminal window showing the command 'pyro4-ns -n 192.168.99.1' being executed. The output shows the broadcast server running on 0.0.0.0:9091, the name server running on 192.168.99.1:9090, and a warning that the HMAC key is not set.

```
(opensource) D:\College Programs and Writeups\4Year\7Sem\DS\DS1>pyro4-ns -n 192.168.99.1
Broadcast server running on 0.0.0.0:9091
NS running on 192.168.99.1:9090 (192.168.99.1)
Warning: HMAC key not set. Anyone can connect to this server!
URI = PYRO:Pyro.NameServer@192.168.99.1:9090
```

Fig6. RMI Name Server (IP: 192.168.99.1) same as RMI Server.

Name Server Running on Windows

Server Running on Windows

Client Running on Windows

Source Code

1. RMIServer.py

```
import Pyro4
import string

@Pyro4.expose
class SecretEncoder(object):
    def encode_secret(self, secret):
        all_letters= string.ascii_letters
        dict1 = {}
        key = 4

        for i in range(len(all_letters)):
            dict1[all_letters[i]] =
all_letters[(i+key)%len(all_letters)]

        cipher_txt=[]

        for char in secret:
            if char in all_letters:
                temp = dict1[char]
                cipher_txt.append(temp)
            else:
                temp =char
                cipher_txt.append(temp)

        cipher_txt= "".join(cipher_txt)

        return cipher_txt

    def decode_secret(self, gibberish):
        all_letters= string.ascii_letters
```

```

dict2 = {}
key = 4

for i in range(len(all_letters)):
    dict2[all_letters[i]] = all_letters[(i-
key)%(len(all_letters))]

decrypt_txt = []

for char in gibberish:
    if char in all_letters:
        temp = dict2[char]
        decrypt_txt.append(temp)
    else:
        temp = char
        decrypt_txt.append(temp)

decrypt_txt = "".join(decrypt_txt)

return decrypt_txt

```

```

daemon = Pyro4.Daemon(host = "192.168.99.1")
ns = Pyro4.locateNS()
uri = daemon.register(SecretEncoder)
ns.register("example.secret", uri)

print("Ready.")
daemon.requestLoop()

```

1. RMIClient.py

```

import Pyro4

alter = input("Encrpyt[E] or Decrypt[D] ? ")

secret_maker = Pyro4.Proxy("PYRONAME:example.secret")

```



```

if alter.lower() == "e":
    secret = input("Enter the Secret Message: ")
    print("Your encoded secret is:
",secret_maker.encode_secret(secret))
else:
    gibberish = input("Enter the Secret Gibberish: ")
    print("Your secret message is:
",secret_maker.decode_secret(gibberish))

```

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 bundled in XML.

RMI mechanism was implemented in which methods of an object are executed remotely over a network.

Python Pyro Library was used for implementing this scenario.

Python Pyro uses the principles of Java RMI for implementation and uses a name server for registering remote objects which can be accessed by any other clients accessing that particular name server by simply providing a domain name.

Evaluation

Grading is performed for this assignment.

RPC laboratory work grading details Points 5 + Instructor Signature	Requirements
--	---------------------

5	<ul style="list-style-type: none"> • 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?	Yes
• Level of understanding DS has improved?	Yes