Bilkent University

Department of Computer Engineering

CS442 - Distributed Systems and Algorithms, Spring 2017

RPC System Middleware Implementation with Stub and Skeleton Generators

Final Report

Group 5

Group Members

Ali Kaviş

Mustafa Yıldız

Bert Yurttaş

Proposed to

Buğra Gedik

May 21, 2017

Table of Contents

1. Introduction

RPC (Remote Procedure Call) is a protocol that helps execute a subroutine in a separate computer with its own address space. An RPC middleware handles the communication without the programmer being bothered with the specifics of the message-passing mechanism. RPC provides the subroutine execution in distant machines/servers with location transparency [1]. Moreover, there is a modern counterpart of RPC, called RMI(Remote Method Invocation) which is specific for object-oriented programming. In this project, we have implemented an RPC middleware on top of sockets that enable communication across processes in remote systems.

The term remote procedure call was coined by Bruce Nelson, who also personally worked on one of the initial implementations of the protocol in 1981 [1]. Stubs and skeletons are the key components of the system. When a remote call is made by the client, client stub packs and serializes parameters for the remote subroutine call (referred to as marshalling) and passes them over to the server as messages by means of sockets and system calls. Server skeleton receives the message, unpacks it, calls the server subroutine and sends the result back to client. Client stub unpacks the returned message and passes the results to caller function [2].

In this project, we proposed to implement an RPC middleware that generates client stub and server skeleton for a given service and its functions. On top of that, we have a simple JSON-based interface definition language, and our users are fervently encouraged to use that specific format for defining interface for their service functions. Our RPC middleware is capable of handling a wide range of faults and errors, that originate both from client and server sides. These errors include incorrect parameter types/numbers, socket connection errors, user-related errors(incorrect function calls) and data loss during transmission.

2. What is a Remote Procedure Call

Remote Procedure Call(RPC) is an important and powerful concept in distributed systems, that allows access to resources broadcasted by distant systems. By using RPC Middleware, application programs do not need the interface detail of the network. Simply, caller procedure is not aware that callee procedure is executing remotely. This communication independence separates application from the physical and logical layer of the message communication. The idea behind the RPC is that clients access the service as if they are making a local call and the respective resources is provided to the user by the middleware that maintains a consistent communication across distant machines. Any communication and message-passing is masked from the user by structures called ‘stubs’. Stubs act like proxies and they forwards function call requests to appropriate distant system that offers the resource. Similarly, servers to do not provide a service by directly processing the remote request. Server side has a stub called skeleton, which has similar functionalities as client stubs. Stubs and skeletons communicate with each other during a remote procedure call.

We can summarize what happens during an RPC call in the following way:

- Client makes a local function call as if they invoke a locally available/defined subroutine
- The relevant client stub locates the server to which the call should be forwarded. At this point stub could be programmed to memorize the location of the remote server or the location could be accessed through a name server to which the server subscribes itself to
- Parameters of the remote subroutine are serialized according to the network protocol used. This is called *parameter marshalling*.
- Once the message is ready, stub connects to the server through the sockets and sends the request with serialized parameters.

- On the other side, server skeleton accepts the request and unmarshalls the parameter set.
- Skeleton invokes the actual call to the function and records the results.
- Returned variables/parameters are marshalled again to be sent back to the client through the socket.
- Client stub receives the serialized results from server skeleton, unmarshalls them and returns the results to the client program that made the initial RPC call.

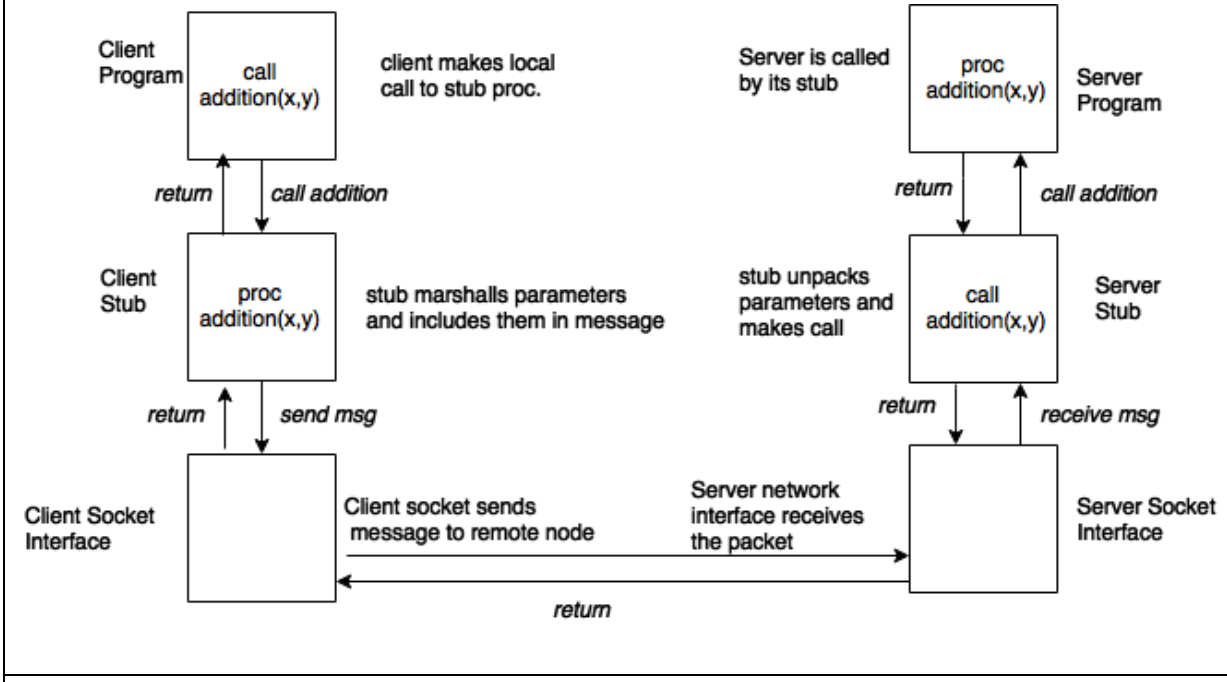

Figure - RPC Mechanism Workflow

3. Description of Distributed System Challenges Addressed

3.1. Different Memory Spaces

Call-by-reference not possible. Client and server do not share any address spaces. Therefore, it is not possible to make a call via pass-by-reference. The proper way of doing so is copying the object in the client and passing this copy to the server skeleton. Since we have used Python as the language of implementation, we pass variables by copy.

3.2. Parameter Marshalling and Passing through Sockets

Marshalling is the parameter packing into a message packet. Unmarshalling is the reverse process. The client stub marshalls the parameters into the call packet; and the server stub unmarshalls the parameters in order to call server's function. After getting things done, on the return, the server stub marshall return parameters into the return packet; the client stub unmarshall return parameters and returns to the client. All parameters are marshalled into strings, and they are transported in JSON format, which makes our application easy-to-debug, and data packets easy-to-transport.

3.3. Concurrent Clients with Threaded Server

An important feature of our RPC middleware is that skeleton initiates a thread per client request. In their respective threads, client requests are processed and results are sent back to them. Threaded server skeleton allows concurrent users to access the same resource, without waiting for each other in some sequential order.

3.4. Binding to Remote Server

Binding is the process of establishing a connection between clients and servers. In a distributed system, distant machines should be able to locate each other whenever they need to communicate. In the vicinity of our application, client stubs should be able to identify remote server addresses before sending the RPC request. We use a Python library that offers a name server service. Once a server goes up, it registers itself to the name server and the skeleton listens to incoming messages. Any client could locate a server using their “service name”.

3.5. Failures and Fault-tolerance

In the absence of faults, the program satisfies its safety and liveness specification. In RPC systems, more failures are expected than simple local invocations. These failures can be categorized into machine and communication failures.

Main difference between local and remote function calls is partial failure, where we encounter a problem at some point of RPC event flow. The following exemplify such errors:

3.5.1. Connection cannot be established

Client cannot locate the server or the located ip:port is not active. The stub raises an exception and returns an empty parameter set with an error message string.

3.5.2. Request message is lost or not properly delivered to server.

In timeout interval, when client gets no reply, it must retransmit requests, in case its request message is not delivered to the other side. (**at least once semantics)**

3.5.3. Reply message is lost or not delivered to the client.

Server is not responsible for retrying reply messages. Client waits for a response for some interval until it times out. It retries to receive a response for some fixed amount of repeats.

3.5.4. Server crashes

Server can crash before, during or after the processing of request. Desirable solution is the **at most once** semantic. This means that if the server crashes, the client will be notified about this event. In our system, such a case will be *partially* signified by a failed response from server. Next time the client makes a call, it immediately discovers the server crash.

3.5.5. Client crashes

Although computation is done at server-side, client may crash before return. In such a case server abort by raising an exception which is caught and handled by the skeleton. Server continues running although there is a crash in any of the client threads (an advantage of threaded server).

4. Our RPC System Design and Major Design Decisions

In this section, we will talk about the specifics of our design choice and how/why we have such a system design. We implemented our middleware in Python, and our RPC system is based on functions defined inside Python classes. At the first sight, it resembles more of an RMI system than a regular RPC system, however, we specifically went for such a design. Classes are analogous to services, and class member functions

are the functions offered by the server. Therefore, we identify each server by its Python class. Next, we explain our system in more details.

4.1. Client Stub Design

Client stub is automatically generated based on interface definition file provided by the user. Each client stub is specific to a certain service and they have the service identification name to which they are bound, generated in their implementation. For each remote function in their associated service, client stub comes with automatically generated *proxy functions*, that have exactly the same and signature. The only additional parameter is an *id* parameter that uniquely identifies each RPC call by the client (we will explain the *id* parameter later in more detail).

The stub intentionally generates proxy function having the same name with their remote counterparts in the distant server because client should be able to access the service functions as if they reside in local machine.

Essentially, any client stub is a Python function that has the same name as the service name. For example, if a user defines a service called Unit_Conversion, then the client stub is generated as a class with the same name. Again, this enables client to access remote service functions with their actual names.

The client stub class stores three important information: name of the remote service, client timeout length and retry count for retrying a request multiple times if it is not properly sent and/or received. User can specify timeout interval and retry count, depending on their needs.

Internally, the stub can be divided into several blocks:

- Parameter type checking
- Locating remote server and retrieving its ip:port
- Marshalling parameters into string(serializable type)
- Initiating socket connection to remote server
- Sending the request
- Waiting for and receiving the response

- Analyzing status of the response
- Type checking return value types

For each of these steps, we carefully check for possible errors. An important characteristic about client stub is that whenever an exception is raised, function returns with an appropriate error explanation. Only exceptions that are tolerated multiple times are the ones related to sending a request and waiting for a response, which we retry for a fixed number of times.

There are two drawbacks of our system from client stubs' perspective: we set a maximum byte length limit for the messages, and we could not marshall user-defined objects that are not a part of standard library of Python. For the first drawback, we have it because both server and the client needs to know some limit about the incoming message, especially in a crowded network traffic. But it discourages passing of big parameters. For the second, in order to marshall and type-check a user defined object, we need to import its script. We have a simple solution for this, which is asking users to include "to-be-imported" modules in the interface definition file with their path.

For simplicity and ease-of-debugging, pass JSON objects in our messages. It is human readable and compatible with many communication protocols.

4.2. Server Skeleton Design

In essence, client stub and server skeleton have very similar responsibilities and structures. Likewise, skeletons are generated per remote service and they are Python classes by themselves. Skeletons have a simple, dummy proxy server class. An instance of this proxy server class stores name of the service, assigned ip and host of the skeleton socket. This dummy instance is embedded into the name server so that client stubs could retrieve the ip and port of the remote service skeleton.

Skeletons class have the same name as the service name defined by the user. This is not a necessity for the smoothness of the distributed system but a design choice we made. Server skeleton instances hold the following information: an instance of the

real server class, a data structure that holds IDs of any previous function calls, and the serversocket from which the requests are listened.

When a server skeleton goes up, it needs to be initialized: it gets assigned a socket with a unique ip:port, it registers itself (or we may say the server) with the name of the service provided by the user, and starts the name server service. In any server skeleton, the client requests are processed in the following way:

- Data is retrieved and checked if client properly transmits the message.
- Length of the serialized parameters are compared with the size that was computed and included in the message. We use this for checking intactness of the parameters.
- Check the function call id, if we received this id before, than the server skeleton stops serving to this client. We add ids that we have seen for the first time
- Requested function of the server is called with parameters
- Return values are marshalled and returned with an appropriate status.

Unlike client stubs, which raise an Exception and exit when a problem is encountered, server skeletons always try to reach to client in case of an error. The error could be client related, connection related or server related. Server understands the error and sends back a response with an appropriate status parameter so that the client could take an action accordingly. Status other than “OK” and “STATUS” are considered as problematic by the client.

Our server skeletons are able to server concurrent clients, which is an important matter for liveliness and availability. When a new client makes a request, the server initiates a new thread and the message is handled separately, while the server continues listening in the main thread. Threaded server architecture provides us two main gains: in theory, we can serve clients concurrently without ordering them into a sequential queue, second, if any problem arises in a client thread that we cannot handle with the current fault tolerance practices the failure in a thread does not affect the availability of the server since the main thread remains unaffected.

4.3. Interface Definition Language

We have seen many types of interface definition languages (IDL) for different RPC services available in the literature. We came up with a very simple IDL, based on JSON. Users are expected to define their service, their functions, parameters and their types etc. in this file so that we can automatically generate stub.skeleton code accordingly. Our interface definition file(IDF) has the following structure. In this example, we have simple math service:

```
{
        "service_name" : "add_service",
        "version" : 1,
        "functions" : [
                {
                        "function_name" : "add",
                        "parameters" : [{"first" : "int"}, {"second" : "int"}],
                        "return" : [{"result" : "int"}, {"addition" : "str"}]
                },
                {
                        "function_name" : "subtract",
                        "parameters" : [{"first" : "int"}, {"second" : "int"}],
                        "return" : [{"result" : "int"}, {"subtraction" : "str"}]
                }
        ]
}
```

Figure - Interface Definition File

Here, user defines the name of the service and the function prototypes of its subroutines. Our rpc tool reads and processes this file accordingly to generate code from our predefined code blocks and code templates.

5. Implementation Details

We started of with the idea that it would be appropriate to implement this system in C/C++, since they are comparably low-level languages and the concept of RPC middleware suits them better. We conducted an extensive research and learned about dynamics of RPC programming and automatic code generation in these languages.

There are two possibilities in C\C++ for code generation that we have examined: using metaprogramming in C++ with templates, and writing a lexer/parser for a specific interface definition language to parse and generate code accordingly. Both these method were time consuming and also requires hig-level understanding of these languages. We also believe that working on these methods would make us digress from the scope of distributed systems course. Therefore, we decided to implement the project in a language that allows faster prototyping and easier code generation, which happens to be Python. Instead of putting effort in learning metaprogramming, we focused on error handling, fault tolerance and server design for request management.

We initially implemented a modular stub and skeleton by hand, and analyzed code segments that should be templated so that we can generate stub.skeleton code by embedding necessary information from interface definition file. As we have discussed in 4.1 and 4.2, stub skeleton have several modules/code blocks that could work independently. On top of that, some of these code segments (i.e. parameter type checking, parameter marshalling ...) need to be unique to the given service and function. We had certain string Templates and common code blocks. Common code blocks are shared across all stubs and skeletons, such as receiving a message, sending a message or connecting to name server. Templates are segments that need to be organized according to service names, their functions and parameter number/type. To generate a code, we combine these code blocks with interface definition file descriptions to generate the code.

6. Evaluation of Fault-tolerance and Performance Results

6.1. Evaluation of Fault-tolerance

Fault tolerance is the feature of the system that makes even if there are failures in the system, the program survives as many of these failures as possible to continue execution. A fault-tolerant design lets the program run in stable form, but possibly slow, rather than letting it fail completely. We achieve fault-tolerance by concerning all exceptional cases. We have given an account of possible failures that we can expect our system can face in Chapter 3. We have tested these and the following images show the result:

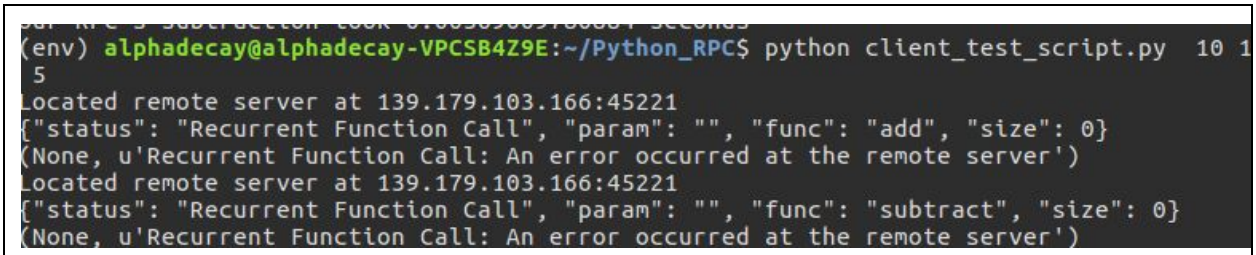

Figure - Requests' have the same id. Server raises "Recurrent Function Call" exception

Such exceptions are implemented to provide all around fault-tolerance.

6.1.1. Stub's Fault tolerance:

- When the given parameters and their given type do not match, the stub raises '*Invalid input arguments*' error and returns 'None'.
- If the stub cannot locate the name server, it raises *'Failed to locate name server'* error and returns 'None'.

- If the stub cannot connect to the server over sockets in time, it raises *'Failed to connect to the remote server'* error and returns 'None'. This time interval is specified with timeout value.
- If the number of retries does not exceed the total retry count allowed, and stub cannot send the request, *'Client Remote Server Connection Error…'* error is raised and 'None' is returned. Each retry corresponds to a single request.
- Else if the number of retries does exceed the total retry count allowed, stub raises *'Client retry count reached'* error and returns 'None'.
- After receiving the response, if marshalled parameters from server fail at type check, *'Invalid return value types'* error message is raised and 'None' is returned.
- If the number of parameters received from server do not match the size value specified by the server, *'Return parameter checksum failure'* is raised and 'None' is returned.
- If response has a status other than 'OK' or 'SUCCESS', the stub returns 'None' and raises the exception *': An error occurred at the remote server'*

6.1.2. Skeleton's Fault tolerance:

- If the skeleton receives empty request from the client, IO error '*Client socket disconnected*' is raised.
- If the skeleton cannot decode the stub's request, it notices the stub with *'Unable to decode serialized parameters'* error message.
- If the skeleton receives the request with an id that is already stored in the skeleton's request set, it notices the stub with *'Recurrent Function Call'* error message.
- If the stub's request's has mismatching or wrong function name, the skeleton notices the stub with *'Invalid function name for server'* error message.
- If all processes flow as expected, the skeleton notices the stub with *'SUCCESS'* message.

- While sending response, if a segment of package cannot be send, the skeleton raises *'Client socket disconnected'* exception.
- While the skeleton is running, if any process flows as unexpected, or does not work, the skeleton raises *'Problem while sending data back to client'* error exception.

6.2. Evaluation of Performance

For evaluation of performance metrics, python's time module is utilized. For each function call, we have measured the elapsed time. To demonstrate and compare our performance, we picked XMLRPC[3] library in the same way.

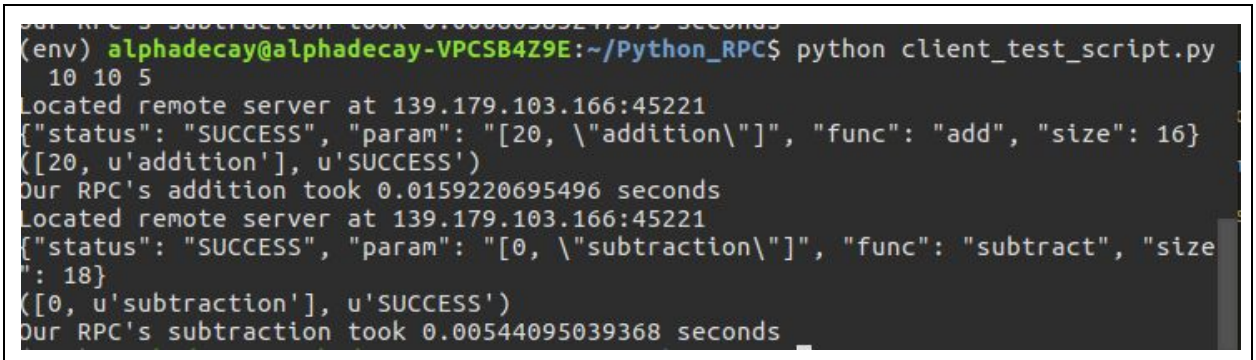

Figure - Elapsed time for our implementation

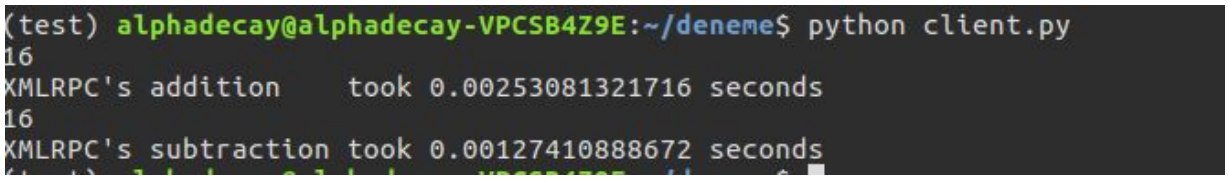

Figure - Elapsed time for XMLRPC implementation

in ms	Elapsed time of addition	Elapsed time of subtraction
Our RPC	15.92	5.44
XMLRPC	2.53	1.27

In both systems, we can observe that second stub process takes less time that first stub process. This is, perhaps, due to the OS's management of sockets. During the first function call, the connection has established. So that, second function call operates on already opened socket. In our RPC, subtraction gets 3 times faster. In XMLRPC, subtraction gets 2 times faster.

Between two systems, our implementation is almost 4-6 times slower. Since our implementation is not optimized, this result is expected. Still, this difference is in terms of milliseconds.

7. Possible Improvements

This is a simple RPC middleware that needs to be improvement from many aspects. A couple of improvements that we have thought of are listed as follows:

- Allowing user-defined objects to be serialized. This comes with amendments to code generator and interface definition file structure
- We do not have a cache on server for failed response messages. Server can hold a data structure of previously failed packets and retry them periodically.
- We open a socket connection each time we make an RPC call to the same server. We can have sessions that can accommodate multiple requests to increase performance
- At the moment, we run our name server in localhost, so it is not accessible from other computers. We can deploy this name server in a fixed location and let truely distant servers and client communicate with each other.

8. User Manual

Use of our tool is very simple, user should only provide an interface definition for their system and provide an implementation for these service functions. The rest is handled by our tool. User should only import the stub and generator in their client or server application to be able to use them.

1. First, user should generate a JSON file that defines the service and its functions. To learn about the IDF structure, go to the directory where “python_rpcgen.py” script is located. In the terminal type: “python python_rpcgen.py --help”. This command gives a simple definition of our IDF.
2. When the user prepares a correct IDF file, the user is ready to generate stub and skeleton for the service. The tool should be called like this: “python python_rpcgen.py <interface definition json file>‘. When the call finishes, stub and skeleton scripts/classes are generated with the name of the service as it appears in the IDF.
3. To use the stub in a client application and the skeleton in a server application, the scripts must be imported. We have two sample codes that show how to use it:
4. Sample Client Application:

```
from __future__ import division
import numbers, json, jsonpickle, Pyro4, socket, select, exceptions, time, sys
import add_service_stub as stub

if __name__ == '__main__':
    if len(sys.argv) < 3 :
        sys.exit(0)

    stub = stub.add_service()
    print stub.add(int(sys.argv[1]), int(sys.argv[2]), time.time())
    print stub.subtract(int(sys.argv[1]), int(sys.argv[2]), time.time())
```

Figure - Sample usage of client stub

5. Sample Server Application:

```
from __future__ import division
import numbers, json, jsonpickle, Pyro4, socket, select, exceptions, sys, threading
import add_service_skeleton as skeleton

class A(object):
    def __init__(self, name):
        self.name = name

    def add(self, a, b):
        return a+b, 'addition'

    def subtract(self, a, b):
        return a-b, 'subtraction'

if __name__ == '__main__':
    a = A('test')
    server_skeleton = skeleton.add_service(a)
    print 'Running the server'
    server_skeleton.run()
```

Figure - Sample usage of server skeleton

9. References

[1] "Remote procedure call", Wikiwand.com , 2017. [Online]. Available: http://www.wikiwand.com/en/Remote_procedure_call. [Accessed: 20- May- 2017].

[2] "Remote Procedure Calls", Cs.rutgers.edu , 2017. [Online]. Available: https://www.cs.rutgers.edu/~pxk/417/notes/08-rpc.html. [Accessed: 20- May- 2017].

[3] "xmlrpclib — XML-RPC", Python.org, 2017. [Online] Available: https://docs.python.org/2/library/xmlrpclib.html. [Accessed: 20- May- 2017]