# Python: Distributed applications with XML-RPC

# PUPPET OBJECTS

**XML-RPC is a portable XML-based process for Remote Procedure Calls. In conjunction with Python it can be used for quick access to distributed applications. Andreas Jung explains how**

These days many applications exist in a distributed environment. This means applications are running on several machines that communicate with each other and exchange data. There are two basic setups: on the one hand there are message-oriented processes for inter-process communications, like Named Pipes; and on the other hand are Remote Procedure Call processes (RPC). RPC processes are used in Unix, for example, to implement various daemons and services such as Portmapper or NFS.

The first standardised processes for providing services across the boundaries of individual machines were created in the 90s. The most important of these processes are CORBA (Common Object Request Broker Architecture) and its proprietary Microsoft counterpart DCOM. Both standards are very complex and therefore more suitable for large enterprise solutions. For the Java environment there is the RMI mechanism (Remote Method Invocation), which allows remote procedure calls between Java programs. Unfortunately RMI is not portable and cannot therefore be used from other programming languages.

XML-RPC bridges this gap by offering the following two advantages: one, it is portable, i.e. it is independent of any particular programming language or operating system and can therefore be applied universally; two, it is simple to implement because XML-RPC is based on the established standards XML and HTTP.

## Listing 1: Client-Server communication with RPC

Client request to calculate the sum of 17 and 15:

```
<methodCall>
  <methodName>example.sum</methodName>
  <params>
    <param><value><int>17</int></value></param>
    <param><value><int>15</int></value></param>
  </params>
</methodCall>
```

Server response:

```
<methodResponse>
  <params>
    <param><value><int>42</int></value></param>
  </params>
</methodResponse>
```

## Table 1: Data types in XML-RPC

| Type | Description |
|---|---|
| **int** | whole number with sign, 32 bits in length |
| **string** | character string (typically with Unicode support, since XML explicitly demands Unicode support) |
| **boolean** | truth values, true or false |
| **double** | double-precision floating-point number |
| **datetime.iso8601** | date and time |
| **base64** | base64-encoded raw data |
| **array** | one-dimensional array, in which the individual array values can be of any type |
| **struct** | A set of key value pairs; keys must be character strings, values can be of any type |

## How XML-RPC works

Communication between a client and an XML-RPC server always takes place via HTTP. The advantage of this is that it allows the use of existing components. Using HTTP also simplifies communication across firewalls and proxies. The client's request to the server and the server's response are encoded in XML. Listing 1 contains an example of this, showing a simple arithmetic calculation using integer data types. Table 1 shows all data types that can be passed between client and server.

## XML-RPC for Python

At the moment there are two XML-RPC implementations for Python. One is the xmlrpclib package by Frederik Lundh, currently maintained by Pythonware. This package supports XML-RPC for server as well as client applications. The client-side components of the package are going to be integrated into Python and should make their first appearance in version 2.2. All examples in the following text relate to this package.

The other implementation is the py-xmlrpc project, which is more recent. Its authors, Chris Jensen and Shilad Sen, have re-implemented time-critical parts in C with very good performance results. Unfortunately the documentation is still on the sparse side.

## XML-RPC clients with Python

The xmlrpclib server object makes it easy to address XML-RPC servers from Python:

```
import xmlrpclib
```

```
server = xmlrpclib.SERVER
(“http://localhost:9000”)
print server.example.sum(17,15)
```

The constructor links the passed HTTP-URLs to the server object. In our example the XML-RPC server is running on a local machine on port 9000. The actual RPC call of the sum() method in the example class is similar to a local function call – with the restriction that keyword parameters such as sum(a=17,b=15) are not allowed.

Some XML-RPCs support something called Introspection API, which clients can use to obtain information about a server's method calls. All methods of this API can be addressed via the system object of the server instance (see Table 2).

## XML-RPC server in Python

Creating an XML-RPC server with Python also involves little effort. The module xmlrpcserver provides all the important functions required. Listing 2 shows the body of such a server.

The call() method of xmlrpcHandler is called by the underlying socket server for each incoming call and receives the name of a method to be called and the arguments for the function call. The call getattr() checks whether the xmlrpcHandler class contains a method of that name, and returns a reference to this method if successful. The method is then started with the appropriate arguments by s_method() and returns a result. Some methods, like in the example, are linked as xmlrpcHandler class methods.

## Authentication for XML-RPC

The XML-RPC standard does not define any authentication processes. Instead, this is left to the transport protocol HTTP. The most common method is basic authentication, in which the user name and password are transferred in the authorisation section of the HTTP header. Cookie-based authentication works in a similar way, but the information is stored in a cookie and then transferred. Unlike basic authentication this method is not standardised. If you want to be able to use basic authentication via XML-RPC you will need to extend the internal transport class, as shown in Listing 3.

The new transport class BasicAuthTransport extends the HTTP header with the appropriate authorisation at each request. This is done by redefining the request() function of the basic class. Applications can use the new transport class by passing an instance to the constructor of the XML-RPC server object (Listing 4).

## Conclusion

Applying an XML-RPC interface to Python applications does not involve much effort. The essential part of the XML-RPC infrastructure remains

### Listing 2: Server body

```
01 import SocketServer
02 import xmlrpcserver
03 import xmlrpclib
04
05 class xmlrpcHandler(xmlrpcserver.RequestHandler):
06
07 def call(self, method, args):
08
09              try:
10                  s_method = getattr(self, method)
11              except:
12                  raise AttributeError, \
13                     “Server does not have XML-RPC “ \
14                     “procedure %s” % method
15              return s_method(method, args)
16
17          def sum(self,a,b):
18              print ‘Arguments:’,a,b
19              return a+b
20
21 if __name__ == ‘__main__’:
22          server = SocketServer.TCPServer((‘’, 8000), xmlrpcHandler)
23          server.serve_forever()
```

### Listing 3: Authentication with XML-RPC

```
01 import string, xmlrpclib, httplib
02 from base64 import encodestring
03
04 class BasicAuthTransport(xmlrpclib.Transport):
05      def __init__(self, username=None, password=None):
06          self.username=username
07          self.password=password
08
09      def request(self, host, handler, request_body):
10              h = httplib.HTTP(host)
11          h.putrequest(“POST”, handler)
12
13          # required by HTTP/1.1
14          h.putheader(“Host”, host)
15
16          # required by XML-RPC
17          h.putheader(“User-Agent”, self.user_agent)
18          h.putheader(“Content-Type”, “text/xml”)
19          h.putheader(“Content-Length”, str(len(request_body)))
20
21          # basic auth
22          if self.username is not None and self.password is not None:
23              h.putheader(“AUTHORIZATION”, “Basic %s” % string.replace(
24                       encodestring(“%s:%s” % (self.username,
self.password)),
25                       “\012”, “”))
26          h.endheaders()
27
28          if request_body:
29              h.send(request_body)
30
31          errcode, errmsg, headers = h.getreply()
32
33          if errcode != 200:
34              raise xmlrpclib.ProtocolError(
35                  host + handler,
36                  errcode, errmsg,
37                  headers
38                  )
39          return self.parse_response(h.getfile())
```

## Table 2: Introspection API methods

| Method | Description |
|---|---|
| **server.system.listMethods()** | returns a list of all methods of the XML-RPC server server.system.method |
| **Signature(methodname)** | returns a list of signatures for a method name, for example server.system.methodSignature('sum') returns [('int','int')] |
| **server.system.method Help(methodname)** | returns method documentation; in Python this is typically the documentation string for the function |

## Listing 4: Calling the new transport class

```
import xmlrpclib
server = xmlrpclib.SERVER("http://localhost:9000", \
           BasicAuthTransport('jim','mypassword'))
print server.example.sum(17,15)
```

hidden from the developer. This has contributed greatly to XML-RPC's popularity, which has practically become the de-facto standard.

### New features in Python 2.2

Guido van Rossum and his team are currently working on Python 2.2, due for release at the end of the year. The second alpha release already offers some new features.

### New division operator: //

The present division operator always returns an integer value. This is inadequate for genuine floating-point arithmetic. From version 3.0 the standard operator / will return results as floating-point values, while the new operator // will be responsible for integer division. You can already use this new functionality by linking from __future__ import division into your programs.

### Unification of built-in types and classes

Until now it has been impossible to derive your own classes from built-in types (lists or dictionaries for example). This limitation ends with 2.2. User-defined dictionary classes can now be derived as follows:

```
class MyDictionary(dictionary):
def __getitem__(self,key): ...
```

### Iterators

Iterators are closely connected to for loops. Sequence types (character strings, lists, tuples) used to be the only types through which iteration was possible within a loop. From 2.2 all objects can be used for iteration with for if they have implemented the new iterator interface. For example:

```
01 class count:
02
03     def __init__(self):
04         self.data = range(0,100)
05         self.n    = 0
06
07     def __iter__(self):
08         return self
09
10
11     def next(self):
12         try:
13             num = self.data[self.n]
14         except:
15             raise StopIteration
16
17         self.n+=2
18         return num
19
20 obj = count()
21 iter_obj = iter(obj)
22
23 for item in iter_obj:
24     print item
```

In order to be able to do this, classes must implement the method __iter__(). An iterator object is created with the new function iter(). This calls __iter__() for the object and returns a reference to an iterator (normally the object itself). The for loop calls the next() function of the iterator until it raises a StopIteration exception.

### Generators

The concept behind generators is closely related to that of iterators. They are basically functions which return a generator object when called and which provide data via the next() method. For instance:

```
01 from __future__ import generators
02
03 def numerator(N):
04     for n in range(N):
05         yield n
06
07 gen = numerator(100)
08
09 while 1:
10     print gen.next()
```

The new command yield returns a value with each next() call. However, the local variables of the generator function are frozen and processing continues in the same place at the following next().

## Info

| | |
|---|---|
| XML-RPC | *http://www.xmlrpc.org/* |
| xmlrpclib | *http://www.pythonware.com/products/xmlrpc/* |
| py-xmlrpc | *http://sourceforge.net/projects/py-xmlrpc/* |

## The author

Python expert Andreas Jung currently lives near Washington D.C. and works for Zope Corporation (formerly Digital Creations) as a software engineer in the Zope core team.