How to write XML-rpc clients

Alec Clews

November 2017

Contents

1 Introduction

The XML-RPC network protocol is a popular, small, lightweight, network protocol for clients to make function calls to a server and receive the results.

The specification has been around since 1999. Recently developed servers will probably offer remote procedure calls based in more modern technology, for example gRPC. However it's very possible you will still need to write or support an XML-RPC client to access an existing server. Here are at PaperCut we are embracing newer network RPC protocols, but we still support a number of legacy APIs that use XML-RPC.

I hope these notes will be useful enough to get you started if you have never used XML-RPC before. Of particular interest to XML-RPC novices will be using curl to see the raw XML in the function calls, and the troubleshooting tips at the end.

I'd love to get your feedback, especially if you notice something that needs fixing. Please leave a comment below.

The XML-RPC model is very simple – you make a call and you wait to get a single response. There is no asynchronous model, no streaming and no security. Note that some XML-RPC libraries extend this model, but we don't discuss them further.

2 How does XML-RPC work technically?

That question is best answered by reading the specification. But the short answer is

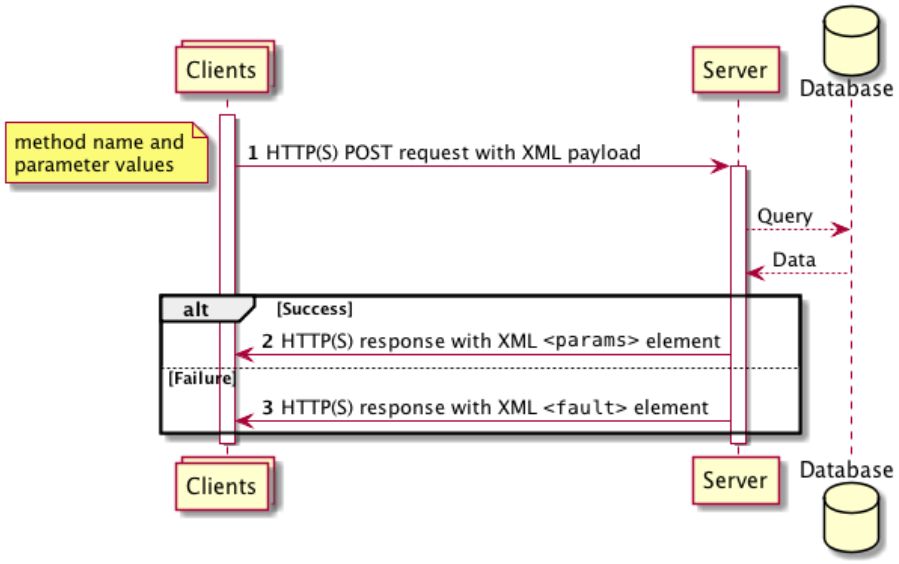

Figure 1: The XML-RPC call sequence

1. The client sends an XML document, using an HTTP(S) POST request, to the server.
2. HTTP status and an XML payload containing return values *OR*
3. HTTP status and a fault, also delivered in an XML document.

The XML schema used is simple – refer to the specification for details.

3 What does this look like?

The easiest way to understand this is to send XML-RPC requests using `curl` on the command line. You can then see the response details, which are often hidden when you program using nice helpful libraries that handle the low level specifics.

If you are using Linux or macOS then you probably already have curl installed, otherwise you will need to install it for yourself.

If you want to follow on with these examples the code is on GitHub

In order to make these examples concrete, I've written a very simple XML-RPC server in Python 3 that supports the following method calls:

- `userExists()` – Does a user exist?
- `getActiveUserUUID()` – Get the UUID for given active user
- `getUserUUIDbyStatus()` – Get the UUID for a given user, any status
- `getUserAllDetails()` – Get all the user details for a given user
- `getAllUsersByStatus()` – List all the users, filtered by their active status

It's all a bit simplistic, but hopefully enough for us to understand how to write clients.

So you can start up the `server/server.py` program and it will serve requests on http://localhost:8080/users.

Once the server is running then we can start to experiment from the command line. First create the request payload in a file called `simpleExample1.xml` (it can be called anything, this is just the name I am using).

```
<?xml version="1.0"?>
<methodCall>
<methodName>getUserAllDetails</methodName>
<params>
<param>
  <value><string>alec</string></value>
</param>
</params>
</methodCall>
```

Now I can run the following command to test the connection

```
curl -v http://localhost:8080/users --data @simpleExample1.xml
```

(note: don't forget the "`@`")

and hopefully get something like this

```
*   Trying ::1...
* TCP_NODELAY set
* Connection failed
* connect to ::1 port 8080 failed: Connection refused
```

```
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST /users HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
> Content-Length: 158
> Content-Type: application/x-www-form-urlencoded
>
} [158 bytes data]
* upload completely sent off: 158 out of 158 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: BaseHTTP/0.6 Python/3.6.3
< Date: Mon, 27 Nov 2017 05:26:33 GMT
< Content-type: text/xml
< Content-length: 359
<
{ [359 bytes data]
<?xml version='1.0'?>
<methodResponse>
<params>
<param>
<value><struct>
<member>
<name>UUID</name>
<value><string>1111113</string></value>
</member>
<member>
<name>activeStatus</name>
<value><boolean>1</boolean></value>
</member>
<member>
<name>user</name>
<value><string>alec</string></value>
</member>
</struct></value>
</param>
</params>
</methodResponse>
* Closing connection 0
```

Notice that this simple example is actually not that simple. The `getUserAllDetails()` returns a struct that contains different types (strings and a boolean).

So now you can start to experiment and see whats happens when you get the URL wrong (the HTTP status changes), when you send ill formed XML and when you try and call method that does not exist. I'm not going to go through all these examples here but for instance what happens when we ask for the UUID of a non existent user?

If we create another payload file with the following content

```
<?xml version="1.0"?>
<methodCall>
<methodName>getUser</methodName>
<params>
<param>
<value>priyanka</value>
</param>
</params>
</methodCall>
```

The response from the server is

```
*   Trying ::1...
* TCP_NODELAY set
* Connection failed
* connect to ::1 port 8080 failed: Connection refused
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST /users HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.54.0
> Accept: */*
> Content-Length: 133
> Content-Type: application/x-www-form-urlencoded
>
} [133 bytes data]
* upload completely sent off: 133 out of 133 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Server: BaseHTTP/0.6 Python/3.6.3
< Date: Mon, 27 Nov 2017 05:26:33 GMT
< Content-type: text/xml
< Content-length: 314
<
{ [314 bytes data]
<?xml version='1.0'?>
<methodResponse>
```

```
<fault>
<value><struct>
<member>
<name>faultCode</name>
<value><int>1</int></value>
</member>
<member>
<name>faultString</name>
<value><string>&lt;class 'Exception'&gt;:method "getUser" is not supported</string></value>
</member>
</struct></value>
</fault>
</methodResponse>
* Closing connection 0
```

Notice that the HTTP response is still 200, but the XML payload now contains a `<fault>` element, instead of a `<params>` element. It will depend on the library functions you use as to how the details of this work in your client code. For instance in Python the caller gets a `Fault` exception, but in Java it's part of the `xmlRpcExcption` (which also handles the HTTP exceptions). I have included examples of error handling in some of the samples on GitHub.

I recommend you experiment further with this technique both as learning *and* a debugging tool.

I have also included a sample XML payload that shows what happens when you call a method with the wrong paramters.

4 Using a real programming language.

Working at the XML level is very educational, but writing shell scripts is not a very practical way to write a high performance, robust, client. So what tools do you need?

1. Your favourite programming language: The good news is that you have lots of choices because XML-RPC is language agnostic. The rest if this post will use Python3 to illustrate the concepts, but I have provided some equivalent examples in Java and Go.
2. An XML-RPC specific library that handles all of the hard work around method names, arguments and responses from the server. There are sometimes multiple options for a specific environment so you may need to do some investigation to see what works best for you.

Here is a list of the libraries that we have used here at PaperCut.

Language	Library

Language	Library
Java	org.apache.xmlrpc
Go	gorilla-xmlrpc
.NET	Cook Computing XML-RPC.NET
Python3	xmlrpc.client
Python2	xmlrpclib
Perl	RPC::XML::Client

You can find other suggestions on the XML-RPC website.

I'll use the same Python server and create a Python client using the `xmlrpc.client` library.

Please note that all this examples are very simplistic and are designed to illustrate the basic process of making XML-RPC calls and handling the responses.

In production code you will probably want to provide an application wrapper to map between domain structures or objects and the data structures supported by the XML-RPC library you are using.

For an example of this wrapper approach please see the complex Go example.

Most XML-RPC libraries work in a similar fashion

1. Create some form of proxy structure. Note that this is internal to the client only, no connection is started with the server.
2. Make method calls via the proxy.
3. Check the results and handle any exceptions.

So now you can start to explore how to use code to call your XML-RPC server.

So in Python:

```
import xmlrpc.client

host = "localhost"
port = "8080"
urlEndPoint="http://"+host+":"+port+"/users"

proxy = xmlrpc.client.ServerProxy(urlEndPoint)
```

Now we have a proxy object "connected" to the correct URL. But remember nothing has happened on the network yet, we've just set up a data structure.

Let's actually try and make a procedure call across the network:

```
    result = proxy.getUserAllDetails(testUser)
    print("""Called getUserAllDetails() on user {},
UUID is {},
active status is {}""".format(
        testUser, result["UUID"], result["activeStatus"]))
```

Straight away we are working at a much higher level.

- We are not handling raw XML
- Information is stored in native Python data structures

However things can go wrong and we we can use standard Python exception mechanisms to manage any errors.

A complete version of the above example would be

```
try:
    result = proxy.userExists(testUser)
    print("\nCalled userExist() on user {},\nresult {}".format(
        testUser, result))

    result = proxy.getUserAllDetails(testUser)
    print("""Called getUserAllDetails() on user {},
UUID is {},
active status is {}""".format(
        testUser, result["UUID"], result["activeStatus"]))

except xmlrpc.client.Fault as error:
    print("""\nCall to user API failed with xml-rpc fault!
reason {}""".format(
        error.faultString))

except xmlrpc.client.ProtocolError as error:
    print("""\nA protocol error occurred
URL: {}
HTTP/HTTPS headers: {}
Error code: {}
Error message: {}""".format(
        error.url, error.headers, error.errcode, error.errmsg))

except ConnectionError as error:
    print("\nConnection error. Is the server running? {}".format(error))
```

and when we run it we get the following output

```
Called userExist() on user alec,
result True
```

```
Called getUserAllDetails() on user alec,
UUID is 1111113,
active status is True
```

By contrast if we get the user name wrong for instance we get an exception.

```
Called userExist() on user anotherUser,
result False

Call to user API failed with xml-rpc fault!
reason No user anotherUser found
```

I have included the full code to this example (`simpleExample1.py`) and another more complex example (`simpleExampleWithErrors1.py`) to show what happens when things goes wrong.

5 Security

XML-RPC provides no security mechanisms and so it's up to the server developer to provide security for client requests.

At the very minimum all method calls and responses should be sent via HTTPS. However the mechanics of using HTTPS will vary depending on the XML-RPC library you are using. Please refer to the documentation for the appropriate library.

Additional security options include:

1. TLS
2. JWT
3. Shared secret, provided via an additional method parameter. This is the approach used by PaperCut as it is easy for client developers to use.
4. username/password authentication. This can also be used with JWT or shared secret.

6 Troubleshooting

So the Python client `simpleExampleWithErrors1.py` shows examples of a number of problems you can experience. The way that the error is reported back to your client will depend the server and the XML-RPC library you use.

Things to check are:

1. Are you using the correct URL endpoint?
2. Is the server running?

3. Is there a firewall or something else blocking network connections?
4. Does the server code have some additional security requirement you have not satisfied? (e.g. passing additional security parameters)
5. Are you using the correct method name?
6. Do you have the correct number of parameters?
7. Is each parameter of the correct type?

6.1 Passing the correct parameter type

This problem can be hard to spot if your language has defaults about type. For instance 3 is an integer and so will be passed as `<int>3</int>`, but if the server is expecting a float then this will probably fail with a type mismatch.

6.2 Low level debugging

If you have checked all the above then I find the most productive approach is to use curl to send the XML request I *think* my code is sending. This is the approach as I showed at the start of the post and I used this technique to debug some problems with the test server I wrote for this article.

If it succeeds then there is a bug in my client and if the call fails then I've made an incorrect assumption about how the server works.