By Lucas Marshall

Table of Contents

Introduction

What is XML−RPC?
XML−RPC was developed by UserLand Software (http://www.userland.com) in April of 1998 as a technology to enable applications to talk to each other, no matter what language they are written in or where they are running. Since then, many implementations have been made beyond the original one for UserLand's Frontier product, and many major companies have adopted the technology for their products, including Microsoft's .NET and Apple's Mac OS X.

XML−RPC is a great enabler for distributed systems and system interoperability, as it allows any XML−RPC enabled applications to call the methods of other XML−RPC enabled applications, regardless of what language either application is written in, or on what machine either application is running on. This allows a perl function to make a call to a Python method, or a Java Servlet to call a PHP function, for example.

The upshot of XML−RPC is that a world of completely heterogeneous applications can talk to each other with very little work from their programmers.

What is covered in this article
This article will cover installing PHP with XML−RPC support, examples of XML−RPC requests and responses, the basic functions used for XML−RPC requests, creating an XML−RPC server, creating an XML−RPC client, and suggests some resources for more information on XML−RPC.

Compiling PHP with XML−RPC Support

As of PHP 4.1.0, the XML−RPC extension is included in the source package with the status of experimental. This extension is simply a copy of the XML−RPC EPI extension that can be found at http://xmlrpc−epi.sourceforge.net/main.php?t=php_about

This extension is by far the best way of implementing XML−RPC in PHP. Being an extension to PHP and not a class or set of methods as the other solutions are, it is written in C and is therefore much faster than any of the other solutions.

The XML−RPC extension is not compiled with PHP by default. In order to enable it, you will need to download the PHP 4.1.0 or greater source from http://www.php.net/downloads.php and run the configure script including the switch:

```
--with-xmlrpc
```

and compile and install as usual.

Doing this will make available several new XML−RPC functions for use in your PHP scripts.

Now, the built in XML−RPC functions are certainly not a joy to use, unless all you want to do is encode PHP variables into XML−RPC requests. The built−in functions will not make the actual requests themselves. I'll show you how to deal with this later in the article.

Dissection of a XML−RPC Call

An XML−RPC call is sent from a client to a server via HTTP POST. This allows XML−RPC applications to work fairly easily behind firewalls, as port 80 is almost always open.

Here's a sample XML−RPC call:

```
POST xmlrpcexample.php HTTP/1.0
User−Agent: xmlrpc−epi−php/0.2 (PHP)
Host: localhost:80
Content−Type: text/xml
Content−length: 191
<?xml version='1.0' encoding="iso−8859−1" ?>
<methodCall>
<methodName>greeting</methodName>
<params>
<param>
<value>
<string>Lucas</string>
</value>
</param>
</params>
</methodCall>
```

The first part of the request is simply a standard HTML post request. It's the data sent in the body of the request that interests us.

The `<methodCall>` tags simply encapsulate the call, with all the parameters enclosed in `<params>` and each parameter enclosed in `<param>`.

Parameters are specified enclosed in tags that tell the application what type of parameter they are. In this case, "Lucas" is a string, so it is enclosed in `<string>`, but there are a lot more parameter types:

<i4> or <int>	four−byte signed integer	10
<boolean>	0 (false) or 1 (true)	1
<string>	ASCII string	Hello, DevShed!
<double>	double−precision signed floating point number	−21.2544
<dateTime.iso8601>	date/time	20011219T12:05:26
<base64>	base64−encoded binary	eW91IGNhbid0IHJlYWQgdGhpcyE=

In addition to the basic types, there are also `<struct>`s and `<array>`s, `<struct>`s containing any number of `<member>`s that consist of `<name>`s and `<value>`s (specified in the same manner as `<param>`s):

```
<struct>
<member>
<name>name</name>
<value>
<string>Lucas Marshall</string>
</value>
</member>
<member>
<name>age</name>
<value>
<i4>20</i4>
</value>
</member>
</struct>
```

and arrays merely containing any number of <value>s:

```
<array>
<value><i4>10</i4></value>
<value><i4>20</i4></value>
<value><i4>30</i4></value>
</array>
```

Dissection of a XML−RPC Response

And here's the response to the call:

```
HTTP/1.1 200 OK
Connection: close
Content−Length: 191
Content−Type: text/xml
Date: Tue, 18 Dec 2001 14:23:52 GMT
Server: xmlrpc−epi−php/0.2 (PHP)
<?xml version='1.0' encoding="iso−8859−1" ?>
<methodResponse>
<params>
<param>
<value>
<string>Hello Lucas. How are you today?</string>
</value>
</param>
</params>
</methodResponse>
```

With your newly found knowledge of XML−RPC calls, it is easy to decipher standard responses. Normal responses consist of `<params>` container with `<param>` elements (and all they are heir to) sent inside `<methodResponse>` tags, but there is another type of response − a fault:

```
HTTP/1.1 200 OK
Connection: close
Content−Length: 356
Content−Type: text/xml
Date: Tue, 18 Dec 2001 13:52:25 GMT
Server: Apache/1.3.20
<?xml version='1.0' encoding="iso−8859−1" ?>
<methodResponse>
<fault>
<value>
<struct>
<member>
<name>faultCode</name>
<value>
<int>4</int>
</value>
</member>
<member>
<name>faultString</name>
<value>
```

```
<string>Too many parameters.</string>
</value>
</member>
</struct>
</value>
</fault>
</methodResponse>
```

As you can see, a fault response consists of a <methodResponse> containing a <fault> which contains a <value> which is a <struct> containing two elements, one named <faultCode>, an <int> and one named <faultString>, a <string>.

Please note that a response cannot contain both a <params> container and a <fault> container.

Creating an XML−RPC Server

In the XML−RPC world, you need a client and a server. The client is simply the application making the request, and the server is simply the service that processes the request and returns a response. In this section, we're looking at how to create an XML−RPC server service.

PHP's XML−RPC extension has several methods that deal with servers, the most important being `xmlrpc_server_create()`, `xmlrpc_server_register_method()` and `xmlrpc_server_call_method()`.

`xmlrpc_create_server()` simply tells PHP that you would like to create a server.

`xmlrpc_server_register_method()` registers PHP methods with the XML−RPC server so they may be called by a client.

`xmlrpc_server_call_method()` is used for passing a request to the XML−RPC server and receiving a response.

Nothing teaches like an example, so here you go (the comments in the example provide more details about the functions).

```
<?php
/*
* First, we define some PHP functions to expose via
* XML−RPC. Any functions that will be called by a
* XML−RPC client need to take three parameters:
* The first parameter passed is the name of the
* XML−RPC method called, the second is an array
* Containing the parameters sent by the client, and
* The third is any data sent in the app_data
* parameter of the xmlrpc_server_call_method()
* function (see below).
*/
function uptime_func($method_name, $params, $app_data)
{
return `uptime`;
}

function greeting_func($method_name, $params, $app_data)
{
$name = $params[0];
return "Hello, $name. How are you today?";
}

/*
* This creates a server and sets a handle for the
* server in the variable $xmlrpc_server
```

```
*/
$xmlrpc_server = xmlrpc_server_create();

/*
* xmlrpc_server_register_method() registers a PHP
* function as an XML−RPC method. It takes three
* parameters:
* The first is the handle of a server created with
* xmlrpc_server_create(), the second is the name to
* register the server under (this is what needs to
* be in the <methodName> of a request for this
* method), and the third is the name of the PHP
* function to register.
*/

xmlrpc_server_register_method($xmlrpc_server, "greeting",
"greeting_func");
xmlrpc_server_register_method($xmlrpc_server, "uptime",
"uptime_func");

/*
* When an XML−RPC request is sent to this script, it
* can be found in the raw post data.
*/
$request_xml = $HTTP_RAW_POST_DATA;

/*
* The xmlrpc_server_call_method() sends a request to
* the server and returns the response XML. In this case,
* it sends the raw post data we got before. It requires
* 3 arguments:
* The first is the handle of a server created with
* xmlrpc_server_create(), the second is a string containing
* an XML−RPC request, and the third is for application data.
* Whatever is passed into the third parameter of this function
* is passed as the third paramater of the PHP function that
the
* request is asking for.
*/
$response = xmlrpc_server_call_method($xmlrpc_server,
$request_xml, '');

// Now we print the response for the client to read.
print $response;

/*
* This method frees the resources for the server specified
* It takes one argument, a handle of a server created with
* xmlrpc_server_create().
```

```
*/
xmlrpc_server_destroy($xmlrpc_server);
?>
```

Creating an XML−RPC Client

Now that we have a server, we need a client to call our methods. As mentioned before, the XML−RPC EPI extension for PHP does not make HTTP requests by itself. For that reason, before it was included in the PHP distribution, the XML−RPC EPI PHP extension came with some include files that had functions that make it a lot easier to work with XML−RPC requests.

The only way to get these functions is to download the XML−RPC EPI PHP extension source package from http://xmlrpc−epi.sourceforge.net/project/showfiles.php?group_id=23199, and install the files in the `/sample/utils` directory into a directory in your PHP `include_path`. For the purpose of the next pieces of example code, I will assume that you have copied the entire `utils` directory into a directory in your `include_path`, and renamed it `xmlrpcutils`.

Making a client is fairly simple. The function that does all the work when we call our methods is `xu_rpc_http_concise()`, which is defined in the `utils.php` file of the `xmlrpcutils` directory now in our PHP `include_path`.

First we'll make a client that calls the first method we defined in the server section of the article, the `uptime` method. This method does not require any parameters, so we won't pass any in. My comments about what we are doing and why will appear as comments in the code.

```
<?php
/*
* This allows us to use the functions that make
* making XML−RPC requests easy.
*/

include("xmlrpcutils/utils.php");

/*
* For the next two lines use this guide.
* If your server script can be found at the URL:
* http://myhost.mydomain.com/xmlrpc_server.php
* $host should be "myhost.mydomain.com"
* and
* $uri should be "/xmlrpc_server.php"
*/

// Change these to match your configuration
$host = "myhost.mydomain.com";
$uri = "/xmlrpc_server.php";

/*
* Here's where we make the request. xu_rpc_http_concise()
* takes one parameter − a keyed array that specifies all
* the info needed for the request.
* The most important (and required) elements are:
```

```
* 'method' − specifies the method to call
* 'host' − specifies the host the server script is on
* 'uri' − specifies the uri of the server script on the server
* 'port' − specifies the port the server is listening on
*
* Here we are calling the uptime method of the
* server script who's uri is specified in $uri of the
* server that is at the hostname specified in $host and
* listening on port 80 and storing it's response (converted
* to a PHP data type) in $result.
*/

$result = xu_rpc_http_concise(
array(
'method' => "uptime",
'host' => $host,
'uri' => $uri,
'port' => 80
)
);

print "The uptime on " . $host . " is " . $result;
?>
```

When this is run it should print something like:

> The uptime on myhost.mydomain.com is 9:43pm up 5:12, 2 users, load average: 0.25, 0.24, 0.22

Where the uptime numbers are the same as those you get if you run the 'uptime' command on the server running the server script.

Our second example will call the `greeting` method. This is one that requires a parameter − a name.

```
<?php
include("xmlrpcutils/utils.php");

$host = "myhost.mydomain.com";
$uri = "/xmlrpc_server.php";

// Change this to your name
$name = "yourname";

/*
* The difference between this call and the last is we pass
* in an array as the 'args' element of the array passed as
```

```
* the argument of the xu_rpc_http_concise() function.
*/

$result = xu_rpc_http_concise(
array(
'method' => "greeting",
'args' => array($name),
'host' => $host,
'uri' => $uri,
'port' => 80
)
);

print $result;
?>
```

When run, this should print:

```
Hello, yourname. How are you today?
```

These are some very simple examples − `xu_rpc_http_concise()` can take some other arguments as well, and there are more functions available. I recommend that you read through the files in the `xmlrpcutils` directory to learn these for yourself.

Conclusion

Well, that's it for now. It should be noted that you could make XML−RPC calls to the server we created with other XML−RPC implementations − you aren't limited to using PHP. Also it should be noted that you could make XML−RPC calls to other, non−PHP servers with PHP. That is the beauty of XML−RPC − it isn't restricted to one language or environment. Remember this when thinking of ways to implement it. Until next time − It's been fun!

Other PHP XML−RPC Solutions

Useful, Inc. − http://xmlrpc.usefulinc.com/
eZ xmlrpc − http://developer.ez.no/article/static/53/
fase4 − http://www.fase4.com/xmlrpc/

Recommended Reading

XML−RPC.com − http://www.xmlrpc.com/
PHP Manual − http://www.php.net/manual/en/ref.xmlrpc.php
xmlrpc−epi−php API reference − http://xmlrpc−epi.sourceforge.net/main.php?t=php_api