Getting Started with XML-RPC in Perl, Part 1

Using XML-RPC for Web services

Joe Johnston (jjohn@cs.umb.edu)
Senior Software Engineer, O'Reilly and Associates
March 2001

Creating an XML-RPC Web service with Perl is almost as easy as CGI scripting. This article will bring you up to speed on what XML-RPC is and how to use Perl's Frontier::RPC library to create simple clients and servers.

Contents:

Application Gateways
XML and RPCs
Remote summing: an example
The sum() client
The sum() server
Simply powerful
Resources
About the author
Rate this article

Application Gateways

Remember the thrill of watching your first HTML form work? Perhaps you simply e-mailed the contents of the form to yourself or displayed another HTML page with whatever information the user entered. Whatever you did, you created what an information architect would call a **two tiered** or **client/server** system. With just a little additional work, the input gathered from a Web form can be stored in a database. In this way, multiple clients can interact with a single database using only their browser. The information stored in the database can be formatted into an appropriate HTML display on demand by CGI scripts. A typical Web application that uses this sort of architecture is a Weblog like SlashDot (see Resources). The code that generates the HTML page is called the **front end** and the part that contains the database and business logic is called the **back end**.

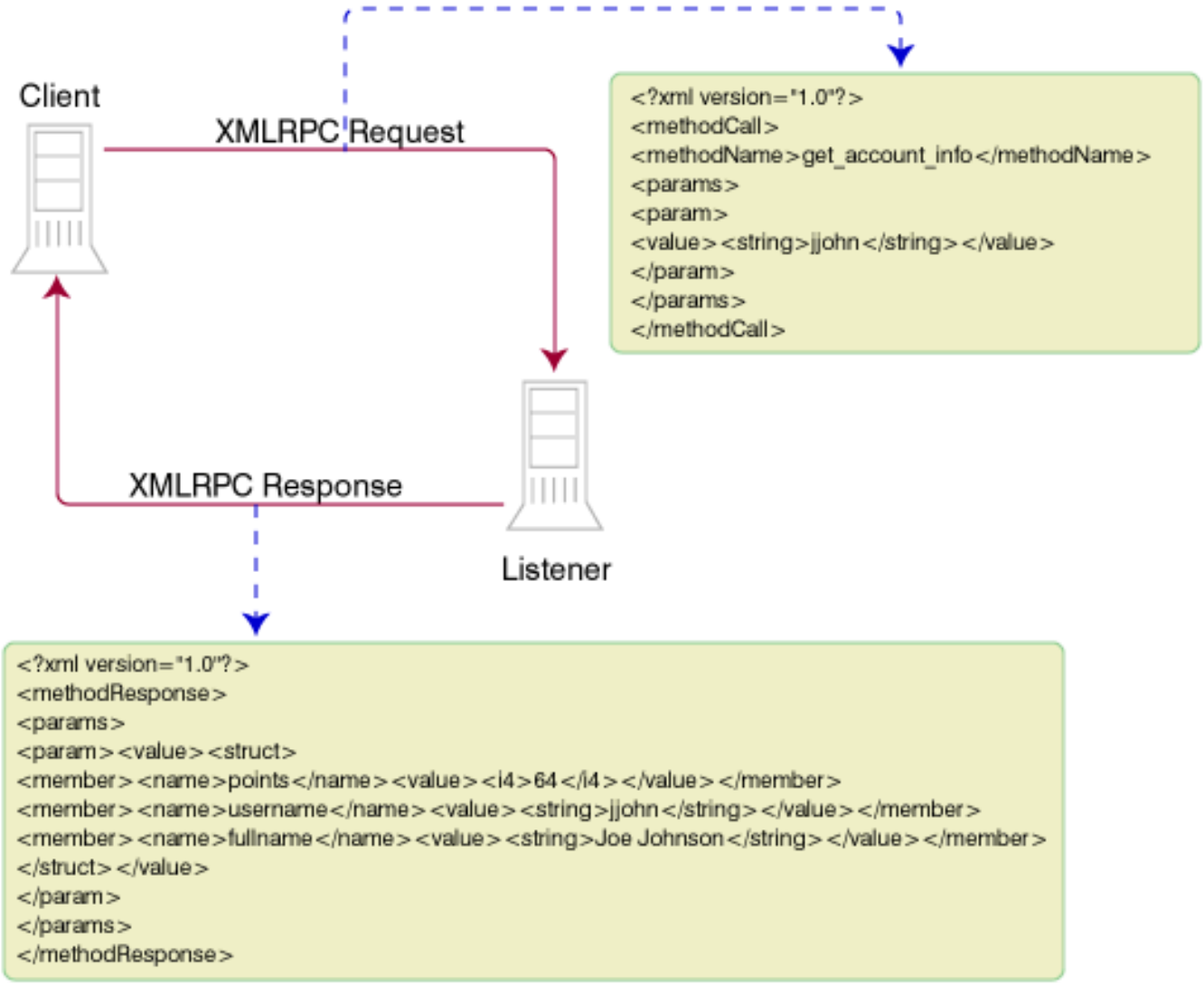
This system works very well until either your database or your Web server can no longer handle the traffic. If the bottleneck lies with your Web server, you may decide to simply add more Web machines to your network. If you connect to your database with native Application Programming Interface (API) calls in your front end, it becomes difficult to change the back end implementation. Switching database vendors or trying to cluster the database servers would mean changing all your front end code. The solution is to separate the presentation logic of the front end from the business logic of the back end, but they still need to be connected. The software that provides the conduit between the front end and the back end is called **middleware**. And one very simple, open architecture middleware protocol that works well in Web applications is XML-RPC.

XML and RPCs

Remote Procedure Calls (RPC) are not a new concept. A client/server system, RPCs have traditionally been procedures called in a program on one machine that go over the network to some RPC server that actually implements the called procedure. The RPC server bundles up the results of the procedure and sends those results back to the caller. The calling program then continues executing. While this system requires a lot of overhead and latency, it also allows less powerful machines to access high powered resources. It also allows applications to harness the computational muscle of a network of machines. A familiar example of this type of distributed computing is the SETI@Home project (see Resources).

Dave Winer, of Frontier and Userland fame (see Resources), helped extend the concept of RPC with XML and HTTP. XML-RPC works by encoding the RPC requests into XML and sending them over a standard HTTP connection to a server or **listener** piece. The listener decodes the XML, executes the requested procedure, and then packages up the results in XML and sends them back over the wire to the client. The client decodes the XML, converts the results into standard language datatypes, and continues executing. Figure 1 is a diagram showing an actual XML-RPC conversation between a client (requesting the get_account_info RPC) and a listener who is returning the results of that procedure.

Figure 1: Sample XML-RPC conversation



The exciting part about XML-RPC is that it can cross programming language and operating system platforms, allowing clients and servers written in different languages to work together. Perl clients can talk to Java servers; Python listeners can service PHP requests; you can even write XML-RPC programs in bad, old C. XML-RPC is extremely easy to work with because the details of the XML translations are hidden from the user, unless, of course, you are implementing your own XML-RPC library.

There are two important aspects of this protocol that you should keep in mind when building your middleware. XML-RPC is built on HTTP and, like ordinary Web traffic, its stateless conversations are of the request and response variety. There is no built-in support for transactions or encryption. The other important detail to remember is that XML-RPC has a finite set of datatypes. Client procedure arguments and listener return values are mapped in a non-extendable XML subset. In practice, though, XML-RPC's datatypes are often flexible enough to do complex tasks.

Table 1 is lists all the XML-RPC datatypes. Four of these are used far more than the others. For single value datatypes, <sting> and <int>, which respectively denote string and integer data, are the workhorses for most XML-RPC programs.

There are also two kinds of collection datatypes. Simple sequences of arbitrary datatypes are represented with the <array> tag. Records, structures, and associative arrays are represented with the <struct> tag. XML-RPC structures are formed with key-value pairs, which should feel natural to Perl, Python, and PHP coders.

Table 1: XML-RPC datatypes

XML-RPC tag	Description
<string>	a sequence of characters
<int>	signed or unsigned 32-bit integer values
<boolean>	true(1) or false(0)
<double>	signed double precision floating point numbers
<dateTime.iso8601>	date and time (but no timezone)
<base64>	a base64 encoded string
<array>	a container for a sequence of datatypes
<struct>	a container for key-value pairs

Remote summing: an example

Let's look at a concrete example of XML-RPC in action. Many introductory computer language books create "hello world" programs which do little more than compile and print a string. An introductory XML-RPC program needs a meatier framework. Because much of your XML-RPC work will be with clients, let's build a client to talk to an XML-RPC listener that defines a sum() procedure, which, unsurprisingly, returns the sum of the two integers that were passed to it. Regardless of the language in which you write an XML-RPC client, you always need to know:

- The URL and TCP port of the listener.
- The name of the remote procedure.
- The kind and number of arguments that procedure expects.
- The kind and number of return values.

This is the sort of information defines an API to the XML-RPC listener. Unfortunately, the XML-RPC specification doesn't provide any standard discovery method for listeners to transmit this information. I recommend using a simple Web page that looks like Table 2.

Table 2: XML-RPC API for sum()

URL:	http://marian.daisypark.net/RPC2	Port:	1080
Procedure Definitions			
Procedure Name	**Input**	**Output**	**Description**
sum	<int>, <int>	<int>	The sum of the two supplied integers will be returned as an <int>

Given this information, we can build the client. Since later in this article there is an extended example of a PHP client, let's see Perl's Frontier::RPC module in action. Don't let the name fool you; this really is Perl's XML-RPC library, which for historical reasons has this unexpected name. This library depends on XML::Parser, which in turn depends on the expat XML parser. Both Perl modules can be found on your nearest CPAN (see Resources) mirror, but you will need to visit SourceForge for the expat source (see Resources). The good news is that expat compiles cleanly on most Unix systems, so installation shouldn't be too difficult. Check out the SourceForge expat page to see how well supported your operating system is.

The sum() client

Once you've got Frontier::RPC installed, you may be surprised at how little code it takes to make an XML-RPC call. Listing 1 is a script that makes the RPC with two hard coded integer arguments and prints the results.

Listing 1: A Perl XML-RPC client testing sum()

```
 1  #!/usr/bin/perl
 2  # Testing sum()
 3
 4  use strict;
 5  use warnings;
 6  use Frontier::Client;
 7
 8  my $url  = "http://marian.daisypark.net:1080/RPC2";
 9  my @args = (2,5);
10
11  my $client = Frontier::Client->new( url   => $url,
12                                      debug => 0,
13                                    );
14
15  print "$args[0] + $args[1] = ", $client->call('sum', @args), "\n";
```

This script starts with the ever-present "shbang" line pointing to my installation's Perl interpreter. Lines 4 and 5 turn on warnings. Most readers will recognize use strict, but use warnings may be somewhat unfamilar, because it was recently added to the core Perl distribution. It checks for the same sort of errors that the -w flag does, but it allows for better control over the that are errors are reported. Line 6 brings in our XML-RPC library. Line 8 creates a variable to hold the listener URL that also includes the port number. Line 9 is a simple list of arguments that will be passed to our remote procedure.

The real paydirt is occurs on Line 11 where a new XML-RPC client object is created. This object is initialized with the URL, although no network connection is established yet. This class also provides a very handy debug feature that, when turned on, will print the XML request and response that happens inside the call() method.

Speaking of the call() method, line 15 shows how seemlessly the RPC fits into normal Perl code. The first argument to call() is the name of the remote procedure followed by a list of its arguments. We will see later how to pass around more complex variables like arrays and structures. The return value from the remote procedure is converted into a standard Perl scalar by call and printed.

You may have noticed Perl's Do What I Mean (DWIM) attitude surface in the Frontier::RPC library which figured out that 5 and 2 are integers. You can turn on debugging in the XML-RPC object to verify this. For times when you force the datatype, the Frontier::RPC library provides an object oriented interface to this type casting. Listing 2 shows a code snippit how we would force the value 2 to be sent as a string.

Listing 2: Forced Typecasting in Frontier::RPC

```
1  use Frontier::RPC2;
2
3  my $coder = Frontier::RPC2->new;
4  my @args;
5
6  push @args, $coder->string("2");
```

As you can see, we need to bring the Frontier::RPC2 class, which is a base class to Frontier::Client. An instantiated object of this class can coerce values into desired datatypes by creating object containers for the data. The call() method can deal with these objects without additional programming effort for the user.

The sum() server

In many ways, creating a Perl XML-RPC listener is almost easier than creating a client. Listing 3 tells the story.

Listing 3: An XML-RPC listener in Perl

```
 1  #!/usr/bin/perl
 2  # sum() server
 3
 4  use strict;
 5  use warnings;
 6  use Frontier::Daemon;
 7
 8  my $d = Frontier::Daemon->new(
 9                                methods => {
10                                           sum => \&sum,
11                                          },
12                                LocalAddr => 'marian.daisypark.net',
13                                LocalPort => 1080,
14                                );
15
16  sub sum {
17    my ($arg1, $arg2) = @_;
18
19    return $arg1 + $arg2;
20  }
```

We've seen lines 1-5 before. The class is Frontier::Daemon, which is a subclass of HTTP::Daemon. When an object of this class is instantiated, the method doesn't return. Instead, the program enters a while loop waiting for new connections. In a tour-de-force of subclassing, HTTP::Daemon is itself derived from IO::Socket::INET. This allows a Frontier::Daemon object to be configured to listen to any TCP port by setting the LocalPort attribute. The Linux box on which this script is executing answers many IP addresses. Again, by using an attribute of IO::Socket::INET, we can restrict this listener to a particular IP with the LocalAddr attribute.

The heart of the XML-RPC listener is the methods attribute that points to an anonymous hash that maps the API procedure name to a reference of a real Perl subroutine that will perform the required work. Recall that the API in Table 2 indicates that the listener has a procedure called sum(). There is no requirement in the Perl XML-RPC library that there be a *Perl* function called sum(). The sum() subroutine is implemented in a direct way. We might want to perform error checking for production code. Notice that sum() is dealing with native Perl scalars and returning a Perl scalar. The library again shields us from the translation to XML.

Simply powerful

Eric Raymond has called XML-RPC "very much in the Unix spirit" (see Resources). Its simplicity creates a lower barrier of entry than its big brother SOAP. With XML-RPC, you not only create a gateway for remote users to access your application, but you also give them the freedom to do so with whatever language they want to use.

In the next article, we will take a look at a more complex and practical example of XML-RPC. Many Web sites require users to log in with account names and passwords. Often, users are issued a session IDs that can be used later to recall a particular application state (think about shopping a cart application that has to remember what's in the cart). Next time, we will create a Web service that shields front end PHP code from the back end MySQL system in which the account information is stored.

Resources

- Read the XML-RPC spec and find links to XML-RPC libraries at the XML-RPC homepage.
- Dave Winer, who created the Frontier library, runs the Userland community.
- The SETI@Home project is a popular example of distributed computing.
- Eric S. Raymond made an interesting note regarding how XML-RPC follows the traditional Unix programming

methodology.

- Get the expat source code from SourceForge.
- Slashdot.org is a popular Weblog for developers.

About the author

By day, Joe Johnston (jjohn@cs.umb.edu) is a programmer for O'Reilly Labs, a new department at O'Reilly and Associates. Whenever his cat isn't sitting on his keyboard, he writes articles for The Perl Journal, use.perl.org, www.perl.com, and the O'Reilly Network. Along with Michael Lord, he created the humorous UFO folklore site, Aliens, Aliens, Aliens.