Part II: XML-RPC

Chapter 2 - XML-RPC Essentials

Chapter 2. XML-RPC Essentials

XML-RPC provides an XML- and HTTP-based mechanism for making method or function calls across a network. XML-RPC offers a very simple, but frequently useful, set of tools for connecting disparate systems and for publishing machine-readable information. This chapter provides a complete overview of XML-RPC, covering the following topics:

- An introduction to the main concepts and history of XML-RPC
- An exploration of XML-RPC usage scenarios, examining its use in glue code and information publishing
- A technical overview of XML-RPC, including a detailed explanation of XML-RPC data types, requests, and responses
- An example demonstrating the use of XML-RPC to connect programs written in Java and Perl

2.1 XML-RPC Overview

XML-RPC permits programs to make function or procedure calls across a network. XML-RPC uses the HTTP protocol to pass information from a client computer to a server computer, describing the nature of requests and responses with a small XML vocabulary. Clients specify a procedure name and parameters in the XML request, and the server returns either a fault or a response in the XML response. XML-RPC parameters are a simple list of types and content - structs and arrays are the most complex types available. XML-RPC has no notion of objects and no mechanism for including information that uses other XML vocabularies. Despite those limitations, it has proven capable of a wide variety of tasks.

XML-RPC emerged in early 1998; it was published by UserLand Software and initially implemented in their Frontier product. It has remained largely stable since then.[1] The XML-RPC specification is available at http://www.xmlrpc.com/spec, and a list of implementations (55 at this writing, in a wide variety of languages) is available at http://www.xmlrpc.com/directory/1568/.

> [1] *For additional information on the early history of XML-RPC, explaining the roles of UserLand and Microsoft, see http://davenet.userland.com/1999/01/29/microsoftXmlRpc. The "snapshot of the spec we were working on with Microsoft" became XML-RPC, while the rest of the spec went on to become SOAP.*

2.2 Why XML-RPC?

In a programming universe seemingly obsessed with objects, XML-RPC may seem too limited for many applications. While XML-RPC certainly has limitations, its inherent simplicity gives it some significant advantages when developers need to integrate systems of very different types. XML-RPC's selection of data types is relatively small, but provides enough granularity that developers can express information in forms any programming language can use.

XML-RPC is used in two main areas, which overlap at times. Systems integrators and programmers building distributed systems often use XML-RPC as glue code, connecting disparate parts inside a private network. By using XML-RPC, developers can focus on the interfaces between systems, not the protocol used to connect those interfaces. Developers building public services can also use XML-RPC, defining an interface and implementing it in the language of their choice. Once that service is published to the Web, any XML-RPC-capable client can connect to that service, and developers can create their own applications that use that service.

2.2.1 Scenario 1: Glue Code with XML-RPC

As distributed systems have become more and more common (by design or by accident), developers have had to address integration problems more and more frequently. Systems that originally ran their own show have to work with other systems as organizations try to rationalize their information management and reduce duplication. This often means that Unix systems need to speak with Windows, which needs to speak with Linux, which needs to speak with mainframes. A lot of programmers have spent a lot of time building custom protocols and formats to let different systems speak to each other.

Instead of creating custom systems that need extensive testing, documentation, and debugging, developers can use XML-RPC to connect programs running on different systems and environments. Using this approach, developers can use existing APIs and add connections to those APIs as necessary. Some problems can be solved with a single procedure, while others require more complex interactions, but the overall approach is much like developing any other set of interfaces. In glue code situations, the distinction between client and server isn't especially significant - the terms only identify the program making the request and the program responding. The same program may have both client and server implementations, allowing it to use XML-RPC for both incoming and outgoing requests.

2.2.2 Scenario 2: Publishing Services with XML-RPC

XML-RPC can be used to publish information to the world, providing a computer-readable interface to information. The infrastructure for this use of XML-RPC is much like traditional web publishing to humans, with pretty much the same security and architecture issues, but it allows information recipients to be any kind of client that understands the XML-RPC interface. As in web publishing, XML-RPC publishing means that developers have control over the server, but not necessarily the client.

The O'Reilly Network's Meerkat headline syndicator, for example, presents both a human-readable interface (at http://meerkat.oreillynet.com) and an XML-RPC interface (documented at http://www.oreillynet.com/pub/a/rss/2000/11/14/meerkat_xmlrpc.html) to the world. Casual readers can use the forms-based interface to query the headlines, while developers who need to present the headline information in other forms can use XML-RPC. This makes it easy to separate content from presentation while still working in a Web-centric environment.

2.3 XML-RPC Technical Overview

XML-RPC consists of three relatively small parts:

XML-RPC data model

A set of types for use in passing parameters, return values, and faults (error messages)

XML-RPC request structures

An HTTP POST request containing method and parameter information

XML-RPC response structures

An HTTP response that contains return values or fault information

The data structures are used by both the request and response structures. The combination of the three parts defines a complete Remote Procedure Call.

It's entirely possible to use XML-RPC without getting into the markup details presented later in this chapter. Even if you plan to stay above the details, however, you probably should read the following sections to understand the nature of the information you'll be passing across the network.

2.3.1 XML-RPC Data Model

The XML-RPC specification defines six basic data types and two compound data types that represent combinations of types. While this is a much more restricted set of types than many programming languages provide, it's enough to represent many kinds of information, and it seems to have hit the lowest common denominator for many kinds of program-to-program communications.

All of the basic types are represented by simple XML elements whose content provides the value. For example, to define a `string` whose value is "Hello World!", you'd write:

```
<string>Hello World!</string>
```

For more information on how Base 64 encoding works, see section 6.8 of RFC 2045, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", available at http://www.ietf.org/rfc/rfc2045.txt. Base 64 is not considered an efficient encoding format, but it does simplify the enclosure of binary information within XML documents. For best results, use it sparingly.

The basic types for XML-RPC are listed in Table 2-1.

Table 2-1. Basic data types in XML-RPC

Type	Value	Examples
`int` or `i4`	32-bit integers between -2,147,483,648 and 2,147,483,647.	`<int>27</int>` `<i4>27</i4>`
`double`	64-bit floating-point numbers	`<double>27.31415</double>` `<double>-1.1465</double>`
`Boolean`	true (1) or false (0)	`<boolean>1</boolean>` `<boolean>0</boolean>`
`string`	ASCII text, though many implementations support Unicode	`<string>Hello</string>` `<string>bonkers! @</string>`
`dateTime.iso8601`	Dates in ISO8601 format: *CCYYMMDD*T*HH*:*MM*:*SS*	`<dateTime.iso8601>20021125T02:20:04 </dateTime.iso8601>` `<dateTime.iso8601>20020104T17:27:30 </dateTime.iso8601>`
`base64`	Binary information encoded as Base 64, as defined in RFC 2045	`<base64>SGVsbG8sIFdvcmxkIQ== </base64>`

These basic types are always enclosed in `value` elements. Strings (and only strings) may be enclosed in a `value` element but omit the `string` element. These basic types may be combined into two more complex types, arrays and structs. Arrays represent sequential information, while structs represent name-value pairs, much like hashtables, associative arrays, or properties.

Arrays are indicated by the `array` element, which contains a `data` element holding the list of values. Like other data types, the `array` element must be enclosed in a `value` element. For example, the following `array` contains four strings:

```
<value>
  <array>
    <data>
      <value><string>This </string></value>
      <value><string>is </string></value>
      <value><string>an </string></value>
      <value><string>array.</string></value>
    </data>
  </array>
</value>
```

The following `array` contains four integers:

```
<value>
  <array>
    <data>
      <value><int>7</int></value>
      <value><int>1247</int></value>
      <value><int>-91</int></value>
      <value><int>42</int></value>
    </data>
  </array>
</value>
```

Arrays can also contain mixtures of different types, as shown here:

```
<value>
  <array>
    <data>
      <value><boolean>1</boolean></value>
      <value><string>Chaotic collection, eh?</string></value>
      <value><int>-91</int></value>
      <value><double>42.14159265</double></value>
    </data>
  </array>
</value>
```

Creating multidimensional arrays is simple - just add an array inside of an array:

```
<value>
  <array>
    <data>
      <value>
        <array>
          <data>
            <value><int>10</int></value>
            <value><int>20</int></value>
            <value><int>30</int></value>
          </data>
        </array>
      </value>
      <value>
        <array>
          <data>
            <value><int>15</int></value>
            <value><int>25</int></value>
            <value><int>35</int></value>
          </data>
        </array>
      </value>
    </data>
  </array>
</value>
```

It's a lot of markup, but for the most part, XML-RPC developers won't have to deal with this markup directly.

XML-RPC won't do anything to guarantee that arrays have a consistent number or type of values. You'll need to make sure that you write code that consistently generates the right number and type of output values if consistency is necessary for your application.

Structs contain unordered content, identified by name. Names are strings, though you don't have to enclose them in `string` elements. Each `struct` element contains a list of `member` elements. `Member` elements each contain one `name` element and one `value` element. The order of members is not considered important. While the specification doesn't require names to be unique, you'll probably want to make sure they are unique for consistency.

A simple struct might look like:

```
<value>
  <struct>
    <member>
      <name>givenName</name>
      <value><string>Joseph</string></value>
    </member>
    <member>
      <name>familyName</name>
      <value><string>DiNardo</string></value>
    </member>
    <member>
      <name>age</name>
      <value><int>27</int></value>
    </member>
  </struct>
</value>
```

Structs can also contain other structs, or even arrays. For example, this struct contains a string, a struct, and an array:

```
<value>
  <struct>
    <member>
      <name>name</name>
      <value><string>a</string></value>
    </member>
    <member>
      <name>attributes</name>
      <value><struct>
         <member>
           <name>href</name>
           <value><string>http://example.com</string></value>
         </member>
         <member>
           <name>target</name>
           <value><string>_top</string></value>
         </member>
    </struct></value>
    </member>
    <member>
      <name>contents</name>
      <value><array>
        <data>
          <value><string>This </string></value>
          <value><string>is </string></value>
          <value><string>an example.</string></value>
        </data>
      </array></value>
    </member>
  </struct>
</value>
```

Arrays can also contain structs. You can, in some cases, use these complex types to represent object structures, but at some point you may find it easier to use SOAP for that kind of complex transfer.

2.3.2 XML-RPC Request Structure

XML-RPC requests are a combination of XML content and HTTP headers. The XML content uses the data typing structure to pass parameters and contains additional information identifying which procedure is being called, while the HTTP headers provide a wrapper for passing the request over the Web.

Each request contains a single XML document, whose root element is a `methodCall` element. Each `methodCall` element contains a `methodName` element and a `params` element. The `methodName` element identifies the name of the procedure to be called, while the `params` element contains a list of parameters and their values. Each `params` element includes a list of `param` elements which in turn contain `value` elements.

For example, to pass a request to a method called `circleArea` , which takes a `Double` parameter (for the radius), the XML-RPC request would look like:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>circleArea</methodName>
  <params>
    <param>
      <value><double>2.41</double></value>
    </param>
  </params>
</methodCall>
```

To pass a set of arrays to a `sortArray` procedure, the request might look like:

```
<?xml version="1.0"?>
<methodCall>
  <methodName>sortArray</methodName>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value><int>10</int></value>
            <value><int>20</int></value>
            <value><int>30</int></value>
          </data>
        </array>
      </value>
    </param>
    <param>
      <value>
        <array>
          <data>
            <value><string>A</string></value>
            <value><string>C</string></value>
            <value><string>B</string></value>
          </data>
        </array>
      </value>
    </param>
  </params>
</methodCall>
```

The HTTP headers for these requests will reflect the senders and the content. The basic template looks like:

```
POST /target HTTP 1.0
User-Agent: Identifier
Host: host.making.request
Content-Type: text/xml
Content-Length: length of request in bytes
```

The information in italics may change from client to client or from request to request. For example, if the `circleArea` method were available from an XML-RPC server listening at */xmlrpc*, the request might look like:

```
POST /xmlrpc HTTP 1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169
```

Assembled, the entire request would look like:

```
POST /xmlrpc HTTP 1.0
User-Agent: myXMLRPCClient/1.0
Host: 192.168.124.2
Content-Type: text/xml
Content-Length: 169

<?xml version="1.0"?>
<methodCall>
  <methodName>circleArea</methodName>
  <params>
    <param>
      <value><double>2.41</double></value>
    </param>
  </params>
</methodCall>
```

It's an ordinary HTTP request, with a carefully constructed payload.

2.3.3 XML-RPC Response Structure

Responses are much like requests, with a few extra twists. If the response is successful - the procedure was found, executed correctly, and returned results - then the XML-RPC response will look much like a request, except that the `methodCall` element is replaced by a `methodResponse` element and there is no `methodName` element:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>18.24668429131</double></value>
    </param>
  </params>
</methodResponse>
```

The `User-Agent` header will typically reflect the XML-RPC library used to assemble the request, not the particular program making the call. This is a bit of a change from the browser world, where "browser sniffing" using that header expects to identify the particular program - say, Opera 6.0 for Linux - making the request.

An XML-RPC response can only contain one parameter, despite the use of the enclosing `params` element. That parameter, may, of course, be an array or a struct, so it is possible to return multiple values. Even if your method isn't designed to return a value (`void` methods in C, C++, or Java, for instance) you still have to return something. A "success value" - perhaps a boolean set to true (1) - is a typical approach to getting around this limitation.

If there was a problem in processing the XML-RPC request, the `methodResponse` element will contain a `fault` element instead of a `params` element. The `fault` element, like the `params` element, has only a single value. Instead of containing a response to the request, however, that value indicates that something went wrong. A fault response might look like:

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value><string>No such method!</string></value>
  </fault>
</methodResponse>
```

The response could also look like:

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
           <name>code</name>
           <value><int>26</int>
        </member>
        <member>
           <name>message</name>
           <value><string>No such method!</string></value>
        </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

XML-RPC doesn't standardize error codes at all. You'll need to check the documentation for particular packages to see how they handle faults.

Like requests, responses are packaged in HTTP and have HTTP headers. All XML-RPC responses use the `200 OK` response code, even if a fault is contained in the message. Headers use a common structure similar to that of requests, and a typical set of headers might look like:

```
HTTP/1.1 200 OK
Date: Sat, 06 Oct 2001 23:20:04 GMT
Server: Apache.1.3.12 (Unix)
Connection: close
Content-Type: text/xml
Content-Length: 124
```

XML-RPC only requires HTTP 1.0 support, but HTTP 1.1 is compatible. The `Server` header indicates the kind of web server used to process requests for the XML-RPC implementation. The header may or may not reflect the XML-RPC server implementation that processed this particular request. The `Content-Type` must be set to `text/xml` ; the `Content-Length` header specifies the length of the response in bytes. A complete response, with both headers and a response payload, would look like:

```
HTTP/1.1 200 OK
Date: Sat, 06 Oct 2001 23:20:04 GMT
Server: Apache.1.3.12 (Unix)
Connection: close
Content-Type: text/xml
Content-Length: 124

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>18.24668429131</double></value>
    </param>
  </params>
</methodResponse>
```

After the response is delivered from the XML-RPC server to the XML-RPC client, the connection is closed. Follow-up requests need to be sent as separate XML-RPC connections.

2.4 Developing with XML-RPC

Using XML-RPC in your applications generally means adding an XML-RPC library and making some of your function calls through that library. Creating functions that will work smoothly with XML-RPC requires writing code that uses only the basic types XML-RPC supports. Otherwise, there is very little fundamental need to change your coding style. Adding XML-RPC support may require writing some wrapper code that connects your code with the library, but this generally isn't very difficult.

As XML-RPC becomes more and more widespread, some environments are building in XML-RPC. UserLand Frontier has done that for years, while the Perl and Python communities are discussing similar integration.

To demonstrate XML-RPC, we're going to create a server that uses Java to process XML-RPC messages, and Java and Perl clients to call procedures on that server. Although this demonstration is simple, it illustrates the connections needed to establish communications between programs using XML-RPC.

The Java side of the conversation uses the Apache XML Project's Apache XML-RPC, available at http://xml.apache.org/xmlrpc/. The Apache package includes a few key pieces that make integrating XML-RPC with Java easier:

- An automated registration process for adding methods to the XML-RPC server
- A built-in server that only speaks XML-RPC, reducing the need to create full-blown servlets
- A client package that makes calling remote methods fairly simple

This demonstration will use a procedure registered with the built-in server of the Apache package and a client for testing the procedure.

For much more information about the Apache XML-RPC package, including data type details and information about creating servlets for XML-RPC processing, see Chapter 3 of *Programming Web Services with XML-RPC* (O'Reilly), by Simon St.Laurent, Edd Dumbill, and Joe Johnston, available online at http://www.oreilly.com/catalog/progxmlrpc/chapter/ch03.html.

The procedure that we'll test returns the area of a circle and is defined in a class called `AreaHandler` , as shown in Example 2-1.

Example 2-1. A simple Java procedure

```
package com.ecerami.xmlrpc;

public class AreaHandler {

     public double circleArea(double radius) {
          double value=(radius*radius*Math.PI);
          return value;
     }
}
```

The `circleArea` method of the `AreaHandler` class takes a `double` value representing the radius, and returns a `double` value representing the area of a circle that has that radius. There's nothing in the `AreaHandler` class that is specific to XML-RPC at all.

Making the `circleArea` method available via XML-RPC requires two steps. The method must be registered with the XML-RPC package, and some kind of server must make the package accessible via HTTP. The `AreaServer` class shown in Example 2-2 performs both these steps.

Example 2-2. Setting up a Java XML-RPC server

```
package com.ecerami.xmlrpc;

import java.io.IOException;
import org.apache.xmlrpc.WebServer;
import org.apache.xmlrpc.XmlRpc;

public class AreaServer {

    public static void main(String[] args) {
```

```
      if (args.length < 1) {
         System.out.println("Usage: java AreaServer [port]");
         System.exit(-1);
        }

      try {
        startServer(args);
      } catch (IOException e) {
          System.out.println("Could not start server: " +
              e.getMessage(  ));
      }
    }

    public static void startServer(String[] args) throws IOException {
      // Start the server, using built-in version
      System.out.println("Attempting to start XML-RPC Server...");
      WebServer server = new WebServer(Integer.parseInt(args[0]));

      System.out.println("Started successfully.");

      // Register our handler class as area
      server.addHandler("area", new AreaHandler(  ));
      System.out.println("Registered AreaHandler class to area.");

      System.out.println("Now accepting requests. (Halt program to stop.)");

    }
}
```

The `main` method checks that there is an argument on the command line specifying on which port to run the server. The method then passes that information to `startServer` , which starts the built-in server. Once the server is started (it begins running when created), the program calls the `addHandler` method to register an instance of the `AreaHandler` class under the name `area`. The `org.apache.xmlrpc.XmlRpc` class deals with all of the method signature details, making it possible to start an XML-RPC service in about two lines of critical code. To fire up the server, just execute `com.ecerami.xmlrpc.AreaServer` from the command line, specifying a port.

```
C:\ora\xmlrpc\java>java com.ecerami.xmlrpc.AreaServer 8899
Attempting to start XML-RPC Server...
Started successfully.
Registered AreaHandler class to area.
Now accepting requests. (Halt program to stop.)
```

The `AreaClient` class shown in Example 2-3 tests the `AreaServer` , once started, from the command line. The `AreaClient` class also uses the XML-RPC library and only needs to use a few lines of code (in the `areaCircle` method) to make the actual call.

Example 2-3. A Java client to test the XML-RPC server

```
package com.ecerami.xmlrpc;

import java.io.IOException;
import java.util.Vector;
import org.apache.xmlrpc.XmlRpc;
import org.apache.xmlrpc.XmlRpcClient;
import org.apache.xmlrpc.XmlRpcException;

public class AreaClient {

    public static void main(String args[]) {
        if (args.length < 1) {
```

```
            System.out.println(
                "Usage: java AreaClient [radius]");
            System.exit(-1);
        }
        AreaClient client = new AreaClient(  );
        double radius = Double.parseDouble(args[0]);

        try {
            double area = client.areaCircle(radius);
            // Report the results
            System.out.println("The area of the circle would be: " + area);

        } catch (IOException e) {
            System.out.println("IO Exception: " + e.getMessage(  ));
        } catch (XmlRpcException e) {
            System.out.println("Exception within XML-RPC: " + e.getMessage(  ));
        }
    }

    public double areaCircle (double radius)
      throws IOException, XmlRpcException {

            // Create the client, identifying the server
            XmlRpcClient client =
                new XmlRpcClient("http://localhost:8899/");

            // Create the request parameters using user input
            Vector params = new Vector(  );
            params.addElement(new Double (radius));

            // Issue a request
            Object result = client.execute("area.circleArea", params);

            String resultStr = result.toString(  );
            double area = Double.parseDouble(resultStr);
            return area;
    }
}
```

The `main` method parses the command line and reports results to the user, but the `areaCircle` method handles all of the interaction with the XML-RPC service. Unlike the server, which runs continuously, the client runs once in order to get a particular result. The same request may be reused or modified, but each request is a separate event. For this application, we just need to make one request, using the value from the command line as an argument. The client constructor takes a URL as an argument, identifying which server it should contact with requests.

Making requests also requires additional setup work that wasn't necessary in creating the server. While the server could rely on method signatures to figure out which parameters went to which methods, the client doesn't have any such information. The Apache implementation takes arguments in a `Vector` object, which requires using the Java wrapper classes (like the `Double` object for `double` primitives) around the arguments. Once that `Vector` has been constructed, it is fed to the `execute` method along with the name of the procedure being called. In this case, the name of the method is `area.circleArea` , reflecting that the `AreaHandler` class was registered on the server with the name `area` and that it contains a method called `circleArea`.

When the `execute` method is called, the client makes an XML-RPC request to the server specified in its constructor. The request calls the method identified by the first argument,

`area.circleArea` in this case, and passes the contents of the second argument as parameters. This produces the following HTTP response.

```
POST / HTTP/1.1
Content-Length: 175
Content-Type: text/xml
User-Agent: Java1.3.0
Host: localhost:8899
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall><methodName>area.circleArea</methodName>
<params>
<param><value><double>3.0</double></value></param>
</params>
</methodCall>
```

The server responds with a `methodResponse` , which the `execute` function reports as an `Object`. Although the XML-RPC response will provide type information about that `Object` , and the underlying content will conform to that type, `Object` is as specific a type as the `execute` function can generally return while still conforming to Java's strong type-checking.

The result of all this work looks pretty simple:

```
C:\ora\xmlrpc\java>java com.ecerami.xmlrpc.AreaClient 3
The area of the circle would be: 28.274333882308138

C:\ora\xmlrpc\java>java com.ecerami.xmlrpc.AreaClient 4
The area of the circle would be: 50.26548245743669
```

Using XML-RPC to connect Java programs to Java programs isn't especially exciting, however. It certainly works - and it can be a great convenience when the only public access to a Java method is through XML-RPC - but much of XML-RPC's potential lies in connecting other environments. To demonstrate that this works with a broader array of environments, we'll create a Perl client that calls the same function.

The Perl client will use the Frontier::RPC module, an implementation of XML-RPC created by Ken MacLeod. (When MacLeod created this library, XML-RPC was primarily a part of UserLand Frontier.) The client component of the Frontier::RPC module is called Frontier::Client.

Frontier::RPC and all of the modules it uses are available from CPAN at http://www.cpan.org.

The logic for the Perl version of the XML-RPC call is much like that of the Java version, except that Perl's flexibility allows us to skip packaging parameters into a vector. The program shown in Example 2-4 accepts a `radius` value from the command line, creates a new XML-RPC connection, and passes the `radius` value as a double to the `area.circleArea` method. Then the program prints the result.

Example 2-4. An XML-RPC client in Perl

```
use Frontier::Client;

$radius=@ARGV[0];

print "for radius: ", $radius, "\n";

my $client=Frontier::Client->new(url=>"http://127.0.0.1:8899");

print " The area of the circle would be: ", $client->call('area.circleArea',
     Frontier::RPC2::Double->new($radius)), "\n";
```

The trickiest part of the procedure call is the casting that needs to be done to ensure that the number is interpreted as a double. Without `Frontier::RPC2::Double->new($radius)`, the Frontier::RPC module will interpret the radius as a string or an integer unless it has a decimal value. Frontier::RPC provides a set of modules that performs this work on Perl values in order to map Perl's loosely typed values to the explicit typing required by XML-RPC. When used on the command line, the Perl procedure call produces results much like those of the Java client:

```
C:\ora\xmlrpc\perl>perl circle.pl 3
for radius: 3
The area of the circle would be: 28.274333882308138

C:\ora\xmlrpc\perl>perl circle.pl 4
for radius: 4
The area of the circle would be: 50.26548245743669
```

For more information on both the Java and Perl implementations of XML-RPC, as well as implementations in Python, PHP, and Active Server Pages, see *Programming Web Services with XML-RPC* (O'Reilly).

2.5 Beyond Simple Calls

XML-RPC is a very simple concept with a limited set of capabilities. Those limitations are in many ways the most attractive feature of XML-RPC, as they substantially reduce the difficulty of implementing the protocol and testing its interoperability. While XML-RPC is simple, the creative application of simple tools can create sophisticated and powerful architectures. In cases where a wide variety of different systems need to communicate, XML-RPC may be the most appropriate lowest common denominator.

Some use cases only require basic functionality, like the library-style functionality described earlier. XML-RPC can support much richer development than these examples show, using combinations of arrays and structs to pass complex sets of information. While calculating the area of a circle may not be very exciting, working with matrices or processing sets of strings may be more immediately worthwhile. XML-RPC itself doesn't provide support for state management, but applications can use parameters to sustain conversations beyond a single request-response cycle, much as web developers use cookies to keep track of extended conversations.

Servers may be able to use XML-RPC to deliver information requested by clients, providing a window on a large collection of information. The O'Reilly Network's Meerkat uses XML-RPC this way, letting clients specify the information they need to receive through XML-RPC procedures. XML-RPC can also be very useful in cases where a client needs to deliver information to a server, both for logging-style operations and operations where the client needs to set properties on a server program. The richness of the interface is up to the developer, but the possibilities are definitely there.