XML−RPC HOWTO

Eric Kidd

Source Builders

eric.kidd@pobox.com

And Other Contributors

0.8.0, 2001−04−12

Revision History

Revision 0.8.0	2001−04−12
Updated section on common interfaces. Added pointer to XML−RPC.Net information.	
Revision 0.7.0	2001−04−03
Added section on C++ proxy classes.	
Revision 0.6.0	2001−02−02
Added sections on Ruby, K and common interfaces.	
Revision 0.5.0	2001−01−23
Initial version.	

Describes how to use XML−RPC to implement clients and servers in a variety of languages. Provides example code in Perl, Python, C, C++, Java, PHP and other languages. Includes sections on Zope and KDE 2.0. Applies to all operating systems with XML−RPC support.

Table of Contents

Table of Contents

1. Legal Notice

2. What is XML−RPC?

XML−RPC is a simple, portable way to make remote procedure calls over HTTP. It can be used with Perl, Java, Python, C, C++, PHP and many other programming languages. Implementations are available for Unix, Windows and the Macintosh.

Here's a short XML−RPC client written in Perl. (We use Ken MacLeod's Frontier::Client module.)

```
use Frontier::Client;
$server = Frontier::Client->new(url => 'http://betty.userland.com/RPC2');
$name = $server->call('examples.getStateName', 41);
print "$name\n";
```

When run, this program will connect to the remote server, get the state name, and print it. (State #41 should be South Dakota in this example.)

Here's the same program in Python. (This time, we use Fredrik Lundh's xmlrpclib.)

```
python> import xmlrpclib
python> server = xmlrpclib.Server("http://betty.userland.com/RPC2")
python> server.examples.getStateName(41)
'South Dakota'
```

In the following chapters, you'll learn how to write XML−RPC clients and servers in a variety of programming languages.

2.1. How it Works

XML−RPC is described fully in Dave Winer's official specification. If you're curious, go ahead and take a look it's a quick and straight−forward read.

On the wire, XML−RPC values are encoded as XML:

```
<methodCall>
  <methodName>sample.sumAndDifference</methodName>
  <params>
    <param><value><int>5</int></value></param>
    <param><value><int>3</int></value></param>
  </params>
</methodCall>
```

This is verbose, but compresses readily. It's also faster than you might expect according to measurements by Rick Blair, a round−trip XML−RPC call takes 3 milliseconds using Hannes Wallnöfer's Java implementation.

2.2. Supported Data Types

XML−RPC supports the following data types:

int

A signed, 32−bit integer.

string

An ASCII string, which may contain NULL bytes. (Actually, several XML−RPC implementations support Unicode, thanks to the underlying features of XML.)

boolean

Either true or false.

double

A double−precision floating point number. (Accuracy may be limited in some implementations.)

dateTime.iso8601

A date and time. Unfortunately, since XML−RPC forbids the use of timezones, this is very nearly useless.

base64

Raw binary data of any length; encoded using Base64 on the wire. Very useful. (Some implementations don't like to receive zero bytes of data, though.)

array

An one−dimensional array of values. Individual values may be of any type.

struct

A collection of key−value pairs. The keys are strings; the values may be of any type.

2.3. The History of XML−RPC

XML−RPC was inspired by two earlier protocols. The first is an anonymous RPC protocol designed by Dave Winer and announced in an old DaveNet essay. (This is why XML−RPC servers are often installed under `/RPC2`.) The other, more important inspiration was an early draft of the SOAP protocol.

A longer history of XML−RPC has been generously provided by Dave Winer. This also explains the relationship between XML−RPC and SOAP.

3. XML−RPC vs. Other Protocols

XML−RPC is not the only way to make remote procedure calls. Other popular protocols include CORBA, DCOM and SOAP. Each of these protocols has advantages and disadvantages.

The opinions in the section are obviously biased; please take them with a grain of salt.

3.1. XML−RPC vs. CORBA

CORBA is a popular protocol for writing distributed, object−oriented applications. It's typically used in multi−tier enterprise applications. Recently, it's also been adopted by the Gnome project for interapplication communication.

CORBA is well−supported by many vendors and several free software projects. CORBA works well with Java and C++, and is available for many other languages. CORBA also provides an excellent *interface definition language* (IDL), allowing you to define readable, object−oriented APIs.

Unfortunately, CORBA is very complex. It has a steep learning curve, requires significant effort to implement, and requires fairly sophisticated clients. It's better−suited to enterprise and desktop applications than it is to distributed web applications.

3.2. XML−RPC vs. DCOM

DCOM is Microsoft's answer to CORBA. It's great if you're already using COM components, and you don't need to talk to non−Microsoft systems. Otherwise, it won't help you very much.

3.3. XML−RPC vs. SOAP

SOAP is very similar to XML−RPC. It, too, works by marshaling procedure calls over HTTP as XML documents. Unfortunately, SOAP appears to be suffering from specification creep.

SOAP was originally created as a collaboration between UserLand, DevelopMentor and Microsoft. The initial public release was basically XML−RPC with namespaces and longer element names. Since then, however, SOAP has been turned over a W3C working group.

Unfortunately, the working group has been adding a laundry−list of strange features to SOAP. As of the current writing, SOAP supports XML Schemas, enumerations, strange hybrids of structs and arrays, and custom types. At the same time, several aspects of SOAP are implementation defined.

Basically, if you like XML−RPC, but wish the protocol had more features, check out SOAP. :−)

4. Common XML−RPC Interfaces

Several XML−RPC servers provide built−in methods. These aren't part of XML−RPC itself, but they make handy add−ons.

4.1. Introspection: Discovering Server APIs

Edd Dumbill proposed the following set of methods:

```
array system.listMethods ()
string system.methodHelp (string methodName)
array system.methodSignature (string methodName)
```

If a server supports these methods, you can ask it to print out some documentation:

```
import xmlrpclib
server = xmlrpclib.Server("http://xmlrpc-c.sourceforge.net/api/sample.php")
for method in server.system.listMethods():
    print method
    print server.system.methodHelp(method)
    print
```

These methods are currently supported by servers written in PHP, C and Microsoft .NET. Partial introspection support is included in recent updates to UserLand Frontier. Introspection support for Perl, Python and Java is available at the XML−RPC Hacks page. Please feel free to add introspection support to other XML−RPC servers!

Various client−side tools (documentation generators, wrapper generators, and so on) can be found at the XML−RPC Hacks page as well.

4.2. Boxcarring: Sending Multiple Requests at Once

If you're writing an XML−RPC client which makes lots of small function calls, you may discover that your round−trip time is fairly high, thanks to Internet backbone latency. Some servers allow you to batch up multiple requests using the following function:

```
array system.multicall (array calls)
```

You can find more information in the system.multicall RFC.

This method is currently supported by servers written in C and UserLand Frontier. Servers written in Python and Perl can use the code at the XML−RPC Hacks page.

5. A Sample API: `sumAndDifference`

To demonstrate XML−RPC, we implement the following API in as many languages as possible.

```
struct sample.sumAndDifference (int x, int y)
```

This function takes two integers as arguments, and returns an XML−RPC `<struct>` containing two elements:

`sum`

The sum of the two integers.

`difference`

The difference between the two integers.

It's not very useful, but it makes a nice example. :−)

This function (and others) are available using the URL `http://xmlrpc−c.sourceforge.net/api/sample.php`. (This URL won't do anything in a browser; it requires an XML−RPC client.)

6. Using XML−RPC with Perl

Ken MacLeod has implemented XML−RPC for Perl. You can find his Frontier::RPC module at his website or through CPAN.

To install Frontier::RPC, download the package and compile it in the standard fashion:

```
bash$ gunzip −c Frontier−RPC−0.07b1.tar.gz | tar xvf −
bash$ cd Frontier−RPC−0.07b1
bash$ perl Makefile.PL
bash$ make
bash$ make test
bash$ su −c 'make install'
```

(The process will be slightly different on Windows systems, or if you don't have root access. Consult your Perl documentation for details.)

6.1. A Perl Client

The following program shows how to call an XML−RPC server from Perl:

```
use Frontier::Client;

# Make an object to represent the XML-RPC server.
$server_url = 'http://xmlrpc-c.sourceforge.net/api/sample.php';
$server = Frontier::Client->new(url => $server_url);

# Call the remote server and get our result.
$result = $server->call('sample.sumAndDifference', 5, 3);
$sum = $result->{'sum'};
$difference = $result->{'difference'};

print "Sum: $sum, Difference: $difference\n";
```

6.2. A Stand−Alone Perl Server

The following program shows how to write an XML−RPC server in Perl:

```
use Frontier::Daemon;

sub sumAndDifference {
    my ($x, $y) = @_;
    return {'sum' => $x + $y, 'difference' => $x - $y};
}

# Call me as http://localhost:8080/RPC2
$methods = {'sample.sumAndDifference' => \&sumAndDifference};
Frontier::Daemon->new(LocalPort => 8080, methods => $methods)
    or die "Couldn't start HTTP server: $!";
```

6.3. A CGI−Based Perl Server

`Frontier::RPC2` doesn't provide built−in support for CGI−based servers. It *does*, however, provide most of the pieces you'll need.

Save the following code as `sumAndDifference.cgi` in your web server's `cgi−bin` directory. (On Unix systems, you'll need to make it executable by typing `chmod +x sumAndDifference.cgi`.)

```
#!/usr/bin/perl -w

use strict;
use Frontier::RPC2;

sub sumAndDifference {
    my ($x, $y) = @_;
    return {'sum' => $x + $y, 'difference' => $x - $y};
}

process_cgi_call({'sample.sumAndDifference' => \&sumAndDifference});


#==========================================================================
#  CGI Support
#==========================================================================
#  Simple CGI support for Frontier::RPC2. You can copy this into your CGI
#  scripts verbatim, or you can package it into a library.
#  (Based on xmlrpc_cgi.c by Eric Kidd <http://xmlrpc-c.sourceforge.net/>.)

# Process a CGI call.
sub process_cgi_call ($) {
    my ($methods) = @_;

    # Get our CGI request information.
    my $method = $ENV{'REQUEST_METHOD'};
    my $type = $ENV{'CONTENT_TYPE'};
    my $length = $ENV{'CONTENT_LENGTH'};

    # Perform some sanity checks.
    http_error(405, "Method Not Allowed") unless $method eq "POST";
    http_error(400, "Bad Request") unless $type eq "text/xml";
    http_error(411, "Length Required") unless $length > 0;

    # Fetch our body.
    my $body;
    my $count = read STDIN, $body, $length;
    http_error(400, "Bad Request") unless $count == $length;

    # Serve our request.
    my $coder = Frontier::RPC2->new;
    send_xml($coder->serve($body, $methods));
}

# Send an HTTP error and exit.
sub http_error ($$) {
    my ($code, $message) = @_;
    print <<"EOD";
Status: $code $message
Content-type: text/html

```

```
<title>$code $message</title>
<h1>$code $message</h1>
<p>Unexpected error processing XML-RPC request.</p>
EOD
    exit 0;
}

# Send an XML document (but don't exit).
sub send_xml ($) {
    my ($xml_string) = @_;
    my $length = length($xml_string);
    print <<"EOD";
Status: 200 OK
Content-type: text/xml
Content-length: $length

EOD
    # We want precise control over whitespace here.
    print $xml_string;
}
```

You can copy the utility routines into your own CGI scripts.

7. Using XML−RPC with Python

Fredrik Lundh has provided an excellent XML−RPC library for Python.

To install, download the latest version. You can either stick the `*.py` files in the same directory as your Python code, or you can install them in your system's Python directory.

RedHat 6.2 users can type the following:

```
bash$ mkdir xmlrpclib-0.9.8
bash$ cd xmlrpclib-0.9.8
bash$ unzip ../xmlrpc-0.9.8-990621.zip
bash$ python
python> import xmlrpclib
python> import xmlrpcserver
python> Control-D
bash$ su -c 'cp *.py *.pyc /usr/lib/python1.5/'
```

We `import` two of the `*.py` files to trick Python into compiling them. Users of other platforms should consult their Python documentation.

For more Python examples, see the article XML−RPC: It Works Both Ways on the O'Reilly Network.

7.1. A Python Client

The following program shows how to call an XML−RPC server from Python:

```
import xmlrpclib

# Create an object to represent our server.
server_url = 'http://xmlrpc-c.sourceforge.net/api/sample.php';
server = xmlrpclib.Server(server_url);

# Call the server and get our result.
result = server.sample.sumAndDifference(5, 3)
print "Sum:", result['sum']
print "Difference:", result['difference']
```

---

8. Using XML−RPC with C and C++

To get a copy of XML−RPC for C/C++, see the xmlrpc−c website.

You can either download everything in RPM format, or you can build it from source.

8.1. A C Client

Save the following code in a file called `getSumAndDifference.c`:

```
#include <stdio.h>
#include <xmlrpc.h>
#include <xmlrpc_client.h>

#define NAME       "XML-RPC getSumAndDifference C Client"
#define VERSION    "0.1"
#define SERVER_URL "http://xmlrpc-c.sourceforge.net/api/sample.php"

void die_if_fault_occurred (xmlrpc_env *env)
{
    /* Check our error-handling environment for an XML-RPC fault. */
    if (env->fault_occurred) {
        fprintf(stderr, "XML-RPC Fault: %s (%d)\n",
                env->fault_string, env->fault_code);
        exit(1);
    }
}

int main (int argc, char** argv)
{
    xmlrpc_env env;
    xmlrpc_value *result;
    xmlrpc_int32 sum, difference;

    /* Start up our XML-RPC client library. */
    xmlrpc_client_init(XMLRPC_CLIENT_NO_FLAGS, NAME, VERSION);
    xmlrpc_env_init(&env);

    /* Call our XML-RPC server. */
    result = xmlrpc_client_call(&env, SERVER_URL,
                                "sample.sumAndDifference", "(ii)",
                                (xmlrpc_int32) 5,
                                (xmlrpc_int32) 3);
    die_if_fault_occurred(&env);

    /* Parse our result value. */
    xmlrpc_parse_value(&env, result, "{s:i,s:i,*}",
                       "sum", &sum,
                       "difference", &difference);
    die_if_fault_occurred(&env);

    /* Print out our sum and difference. */
    printf("Sum: %d, Difference: %d\n", (int) sum, (int) difference);

    /* Dispose of our result value. */
    xmlrpc_DECREF(result);
```

```
    /* Shutdown our XML-RPC client library. */
    xmlrpc_env_clean(&env);
    xmlrpc_client_cleanup();

    return 0;
}
```

To compile it, you can type:

```
bash$ CLIENT_CFLAGS=`xmlrpc-c-config libwww-client --cflags`
bash$ CLIENT_LIBS=`xmlrpc-c-config libwww-client --libs`
bash$ gcc $CLIENT_CFLAGS -o getSumAndDifference getSumAndDifference.c $CLIENT_LIBS
```

You may need to replace **gcc** with the name of your system's C compiler.

8.2. A C++ Client

Save the following code in a file called `getSumAndDifference2.cc`:

```
#include <iostream.h>
#include <XmlRpcCpp.h>

#define NAME       "XML-RPC getSumAndDifference C++ Client"
#define VERSION    "0.1"
#define SERVER_URL "http://xmlrpc-c.sourceforge.net/api/sample.php"

static void get_sum_and_difference () {

    // Build our parameter array.
    XmlRpcValue param_array = XmlRpcValue::makeArray();
    param_array.arrayAppendItem(XmlRpcValue::makeInt(5));
    param_array.arrayAppendItem(XmlRpcValue::makeInt(3));

    // Create an object to resprent the server, and make our call.
    XmlRpcClient server (SERVER_URL);
    XmlRpcValue result = server.call("sample.sumAndDifference", param_array);

    // Extract the sum and difference from our struct.
    XmlRpcValue::int32 sum = result.structGetValue("sum").getInt();
    XmlRpcValue::int32 diff = result.structGetValue("difference").getInt();

    cout << "Sum: " << sum << ", Difference: " << diff << endl;
}

int main (int argc, char **argv) {

    // Start up our client library.
    XmlRpcClient::Initialize(NAME, VERSION);

    // Call our client routine, and watch out for faults.
    try {
        get_sum_and_difference();
    } catch (XmlRpcFault& fault) {
        cerr << argv[0] << ": XML-RPC fault #" << fault.getFaultCode()
             << ": " << fault.getFaultString() << endl;
        XmlRpcClient::Terminate();
        exit(1);
```

```
    }

    // Shut down our client library.
    XmlRpcClient::Terminate();
    return 0;
}
```

To compile it, you can type:

```
bash$ CLIENT_CFLAGS=`xmlrpc-c-config c++ libwww-client --cflags`
bash$ CLIENT_LIBS=`xmlrpc-c-config c++ libwww-client --libs`
bash$ c++ $CLIENT_CFLAGS -o getSumAndDifference2 getSumAndDifference2.cc $CLIENT_LIBS
```

You'll need a reasonably modern C++ compiler for this to work.

8.3. A C++ Client with Proxy Classes

If your XML−RPC server supports the Introspection API, you can automatically generate C++ proxy classes for it. To generate a proxy class, type the following command and save the output to a file:

```
bash$ xml-rpc-api2cpp \
> http://xmlrpc-c.sourceforge.net/api/sample.php sample SampleProxy
```

This will generate a proxy class named `SampleProxy` containing wrappers for all the methods beginning with `sample`. You can edit this class in any fashion you'd like.

8.4. A CGI−Based C Server

Save the following code in a file called `sumAndDifference.c`:

```
#include <xmlrpc.h>
#include <xmlrpc_cgi.h>

xmlrpc_value *
sumAndDifference (xmlrpc_env *env, xmlrpc_value *param_array, void *user_data)
{
    xmlrpc_int32 x, y;

    /* Parse our argument array. */
    xmlrpc_parse_value(env, param_array, "(ii)", &x, &y);
    if (env->fault_occurred)
        return NULL;

    /* Return our result. */
    return xmlrpc_build_value(env, "{s:i,s:i}",
                              "sum", x + y,
                              "difference", x - y);
}

int main (int argc, char **argv)
{
    /* Set up our CGI library. */
    xmlrpc_cgi_init(XMLRPC_CGI_NO_FLAGS);
```

```
    /* Install our only method (with a method signature and a help string). */
    xmlrpc_cgi_add_method_w_doc("sample.sumAndDifference",
                                &sumAndDifference, NULL,
                                "S:ii", "Add and subtract two integers.");

    /* Call the appropriate method. */
    xmlrpc_cgi_process_call();

    /* Clean up our CGI library. */
    xmlrpc_cgi_cleanup();
}
```

To compile it, you can type:

```
bash$ CGI_CFLAGS=`xmlrpc-c-config cgi-server --cflags`
bash$ CGI_LIBS=`xmlrpc-c-config cgi-server --libs`
bash$ gcc $CGI_CFLAGS -o sumAndDifference.cgi sumAndDifference.c $CGI_LIBS
```

Once this is done, copy `sumAndDifference.cgi` into your webserver's `cgi-bin` directory.

9. Using XML−RPC with Java

Hannes Wallnöfer has provided an excellent implementation of XML−RPC for Java.

To install it, download the distribution, unzip it, and add the `*.jar` files to your `CLASSPATH`. On a Unix system, you can do this by typing:

```
bash$ unzip xmlrpc-java.zip
bash$ cd xmlrpc-java/lib
bash$ CLASSPATH=`pwd`/openxml-1.2.jar:`pwd`/xmlrpc.jar:$CLASSPATH
```

9.1. A Java Client

Save the following program in a file named `JavaClient.java`.

```
import java.util.Vector;
import java.util.Hashtable;
import helma.xmlrpc.*;

public class JavaClient {

    // The location of our server.
    private final static String server_url =
        "http://xmlrpc-c.sourceforge.net/api/sample.php";

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

            // Create an object to represent our server.
            XmlRpcClient server = new XmlRpcClient(server_url);

            // Build our parameter list.
            Vector params = new Vector();
            params.addElement(new Integer(5));
            params.addElement(new Integer(3));

            // Call the server, and get our result.
            Hashtable result =
                (Hashtable) server.execute("sample.sumAndDifference", params);
            int sum = ((Integer) result.get("sum")).intValue();
            int difference = ((Integer) result.get("difference")).intValue();

            // Print out our result.
            System.out.println("Sum: " + Integer.toString(sum) +
                               ", Difference: " +
                               Integer.toString(difference));

        } catch (XmlRpcException exception) {
            System.err.println("JavaClient: XML-RPC Fault #" +
                               Integer.toString(exception.code) + ": " +
                               exception.toString());
        } catch (Exception exception) {
            System.err.println("JavaClient: " + exception.toString());
        }
    }
}
```

9.2. A Stand−Alone Java Server

Save the following program in a file named `JavaServer.java`.

```
import java.util.Hashtable;
import helma.xmlrpc.*;

public class JavaServer {

    public JavaServer () {
        // Our handler is a regular Java object. It can have a
        // constructor and member variables in the ordinary fashion.
        // Public methods will be exposed to XML-RPC clients.
    }

    public Hashtable sumAndDifference (int x, int y) {
        Hashtable result = new Hashtable();
        result.put("sum", new Integer(x + y));
        result.put("difference", new Integer(x - y));
        return result;
    }

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

            // Invoke me as <http://localhost:8080/RPC2>.
            WebServer server = new WebServer(8080);
            server.addHandler("sample", new JavaServer());

        } catch (Exception exception) {
            System.err.println("JavaServer: " + exception.toString());
        }
    }
}
```

10. Using XML−RPC with PHP

Edd Dumbill has implemented XML−RPC for PHP. You can download it from the UsefulInc XML−RPC website.

To install the distribution, decompress it and copy `xmlrpc.inc` and `xmlrpcs.inc` into the same directory as your PHP scripts.

10.1. A PHP Client

The following script shows how to embed XML−RPC calls into a web page.

```
<html>
<head>
<title>XML-RPC PHP Demo</title>
</head>
<body>
<h1>XML-RPC PHP Demo</h1>

<?php
include 'xmlrpc.inc';

// Make an object to represent our server.
$server = new xmlrpc_client('/api/sample.php',
                            'xmlrpc-c.sourceforge.net', 80);

// Send a message to the server.
$message = new xmlrpcmsg('sample.sumAndDifference',
                         array(new xmlrpcval(5, 'int'),
                               new xmlrpcval(3, 'int')));
$result = $server->send($message);

// Process the response.
if (!$result) {
    print "<p>Could not connect to HTTP server.</p>";
} elseif ($result->faultCode()) {
    print "<p>XML-RPC Fault #" . $result->faultCode() . ": " .
        $result->faultString();
} else {
    $struct = $result->value();
    $sumval = $struct->structmem('sum');
    $sum = $sumval->scalarval();
    $differenceval = $struct->structmem('difference');
    $difference = $differenceval->scalarval();
    print "<p>Sum: " . htmlentities($sum) .
        ", Difference: " . htmlentities($difference) . "</p>";
}
?>

</body></html>
```

If your webserver doesn't run PHP scripts, see the PHP website for more information.

10.2. A PHP Server

The following script shows how to implement an XML−RPC server using PHP.

```
<?php
include 'xmlrpc.inc';
include 'xmlrpcs.inc';

function sumAndDifference ($params) {

    // Parse our parameters.
    $xval = $params->getParam(0);
    $x = $xval->scalarval();
    $yval = $params->getParam(1);
    $y = $yval->scalarval();

    // Build our response.
    $struct = array('sum' => new xmlrpcval($x + $y, 'int'),
                    'difference' => new xmlrpcval($x - $y, 'int'));
    return new xmlrpcresp(new xmlrpcval($struct, 'struct'));
}

// Declare our signature and provide some documentation.
// (The PHP server supports remote introspection. Nifty!)
$sumAndDifference_sig = array(array('struct', 'int', 'int'));
$sumAndDifference_doc = 'Add and subtract two numbers';

new xmlrpc_server(array('sample.sumAndDifference' =>
                        array('function' => 'sumAndDifference',
                              'signature' => $sumAndDifference_sig,
                              'docstring' => $sumAndDifference_doc)));
?>
```

You would normally invoke this as something like `http://localhost/path/sumAndDifference.php`.

11. Using XML−RPC with Microsoft .NET

Charles Cook is writing an excellent new chapter for this HOWTO. You can find a draft online. If that page disappears, look for a new version of the XML−RPC HOWTO at the Linux Documentation Project.

12. Using XML−RPC with Ruby

(This section of the XML−RPC HOWTO was generously provided by Michael Neumann.)

Ruby is an object−oriented scripting language. It already has a major following in Japan, and it's becoming popular elsewhere.

To use XML−RPC with Ruby, you must first install Yoshida Masato's xmlparser module (a wrapper for James Clark's expat parser). This can be found at the Ruby Application Archive.

Then, you must install xmlrpc4r using the following commands:

```
bash$ tar −xvzf xmlrpc4r−1_2.tar.gz
bash$ cd xmlrpc4r−1_2
bash$ su root −c "ruby install.rb"
```

12.1. A Ruby Client

Here's a simple Ruby client:

```
require "xmlrpc/client"

# Make an object to represent the XML-RPC server.
server = XMLRPC::Client.new( "xmlrpc-c.sourceforge.net", "/api/sample.php")

# Call the remote server and get our result
result = server.call("sample.sumAndDifference", 5, 3)

sum = result["sum"]
difference = result["difference"]

puts "Sum: #{sum}, Difference: #{difference}"
```

12.2. A Ruby Server

Here's a simple Ruby server:

```
require "xmlrpc/server"

s = XMLRPC::CGIServer.new

s.add_hanlder("sample.sumAndDifference") do |a,b|
  { "sum" => a + b, "difference" => a - b }
end

s.serve
```

This could also have been written as follows:

```
require "xmlrpc/server"
```

```
s = XMLRPC::CGIServer.new

class MyHandler
  def sumAndDifference(a, b)
    { "sum" => a + b, "difference" => a - b }
  end
end

s.add_handler("sample", MyHandler.new)
s.serve
```

To run either server in standalone mode, replace the second line of code with the following:

```
s = XMLRPC::Server.new(8080)
```

13. Using XML−RPC with Proprietary Languages

Implementations of XML−RPC are available for the following proprietary programming languages and environments.

13.1. Using XML−RPC with K

(This section of the XML−RPC HOWTO was generously provided by Christian Langreiter.)

K is a language used in finance and database development. To install XML−RPC for K, download it from the Kx website. Uncompress and copy the files into the directory in which you keep your .k programs.

Here's a short client:

```
\l kxr
server:("localhost";"/cgi-bin/sumAndDifference.cgi");
.kxr.c[server;"sumAndDifference";2 3]
```

And here's a server:

```
\l kxrsc
.kxr.sm: ,("sumAndDifference";{.((`sum;+/x);(`difference;-/x))})
```

More examples are included in the distribution.

14. Applications with Built−in XML−RPC Support

Several popular Linux applications include support for XML−RPC. These have already been described elsewhere, so we mostly provide pointers to articles.

14.1. Zope

Articles on using XML−RPC with Zope are available elsewhere on the web:

- XML−RPC Programming with Zope by Jon Udell
- Zope XML−RPC at UserLand.Com

14.2. KDE 2.0

KDE 2.0 includes Kurt Ganroth's kxmlrpc daemon, which allows you to script KDE applications using XML−RPC.

Here's a short sample application in Python. It shows you how to connect to kxmlrpc, manipulate your KDE address book, and query the KDE trader.

If you have any other articles or example code, please see Section 15.3. We'd like to have more information on scripting KDE.

15. About This Document

This document is part of the Linux Documentation Project. Thanks go to Dave Winer and maintainers of all the various XML−RPC libraries.

15.1. New Versions of This Document

New versions of this document are available at the Linux Documentation Project website.

They also provide this manual in alternate formats, including PDF, gzipped HTML and ASCII text.

15.2. Contributors to the XML−RPC HOWTO

The section on Ruby was contributed by Michael Neumann (neumann AT s−direktnet DOT de). The section on K was contributed by Christian Langreiter (c DOT langreiter AT synerge DOT at).

These contributors are all credited at the beginning of their sections.

15.3. Submitting Other Snippets

If you have a sample client or server in another language or environment, we'd love to include it in this manual. To add a new entry, we need the following information:

- The URL of the XML−RPC implementation used.
- Installation instructions.
- A complete, runnable program.
- Compilation instructions, if applicable.

E−mail your example to the xmlrpc−c−devel mailing list or directly to Eric Kidd.

Thank you for your help!