User Manual for XML-RPC

Table Of Contents

- Preface: About This Manual
- Introduction to XML-RPC for C/C++
  - What is XML-RPC?
  - How Does XML-RPC For C/C++ Help?
  - More Information On XML-RPC For C/C++
- The Xmlrpc-c Function Libraries
  - C Libraries
  - C++ Libraries
- Utility Programs
  - **xmlrpc**
  - **xmlrpc_dumpserver**
- Alternatives
  - Other Programming Languages
  - Apache Module
- Appendices
  - Introductory Examples
  - About This Document

Preface: About This Manual

This is the manual for users of XML-RPC for C/C++.

The manual documents all current and past releases of XML-RPC for C/C++ and even planned future releases. Each part of the manual tells you to what releases it applies. Since the main difference between releases is that newer ones have more features, this mainly means that the description of a feature tells you in what release it was added to the package.

Compared to the more common system of distributing manuals keyed to particular releases, this makes the manual harder to use for some users (to wit, a user who has no intention of using any but one particular release), but it allows for a higher quality manual for past releases, with the same publication effort. It also allows additional uses: writing portable code and discovering when you could benefit by moving to a newer release.

For more information about the document, see About This Document.

Introduction

XML-RPC for C/C++ is a software package of programming libraries to help a C or C++ program use XML-RPC. In particular, a C/C++ programmer can easily write a program to be an XML-RPC client or server.

XML-RPC for C/C++ is also known as Xmlrpc-c.

What is XML-RPC?

XML-RPC is a standard network protocol that computers can use to talk to each other in a remote procedure call fashion. Remote procedure call essentially means that a program on one computer runs a program on another computer. But a simpler way of looking at this kind of network protocol is just that you have clients and servers. A client makes individual isolated requests of a server. A server sits around waiting for a request to arrive from some client, does what the request asks, and sends a response. It then goes back to waiting for the next request.

Here are some examples of remote procedure call (RPC) style communications:

- There is a server that can measure atmospheric temperature. A client anywhere in the world can ask the server at any time what the temperature is. The "what temperature is it?" request and the "the temperature is..." response constitute an RPC transaction.
- There is a server that can turn a light on or off. A client can tell the server to turn the light on. A request to turn the light on and the acknowledgement that the light has been turned on constitute an RPC transaction.
- There is a server that knows the phone numbers of a million people. A client can supply a name and get back the phone number of the named person.
- A network of 1000 computers is used to search millions of web pages. A dispatcher computer is a client and the rest are servers. The dispatcher sends each of the servers one URL at a time and some search terms. The server responds with yes or no as to whether the page with that URL contains the search terms. The dispatcher keeps a list of the URLs that match. Each of the requests to search a particular URL is an RPC transaction.

Here are some kinds of communication that are *not* RPC:

- A long-lived connection such as an SSH login session.
- A high volume transfer such as an FTP download.
- A one-way transmission such as a UDP packet.
- A dialogue such as an SMTP (mail) transaction.

The original RPC protocol is the ONC RPC protocol -- the one that NFS (the network fileystem protocol) uses. It's often called "Sun RPC" because Sun invented and promulgated it. The ONC RPC protocol is layered over UDP or sometimes TCP and uses a machine-friendly bits and bytes format, just like the TCP/IP layers under it.

XML-RPC differs from ONC RPC in that it encodes the information in the requests and responses in XML. That means they are human friendly -- XML is human-readable text, so a human can readily see what's going on from a network trace and quickly write code to create and decipher XML-RPC streams. Of course, the tradeoff is that XML-RPC uses way more network and computation resources than ONC RPC. Because XML-RPC is meant to be used for relatively small and infrequent transactions, this is thought not to matter.

In XML-RPC, the aforementioned XML is transported via HTTP (the protocol whose principal purpose is to implement web serving -- web browsing is a form of RPC, after all). HTTP is normally carried over TCP, which is carried over IP.

There are lots of servers in the world that use the XML-RPC protocol and lots of programs and programming libraries from which people can build XML-RPC-based servers and clients.

There are also other HTTP-based RPC protocols. SOAP and CORBA are the most famous. REST-RPC is a more recent entry.

For more information on XML-RPC, see The XML-RPC web site and The XML-RPC Howto. The latter includes information (with examples) on implementing XML-RPC clients and servers in languages other than C and C++, which is all that is covered in this document.

How Does XML-RPC For C/C++ Help with XML-RPC?

The function libraries in XML-RPC For C/C++ (Xmlrpc-c) let you write a program that makes XML-RPC calls (a client) or executes XML-RPC calls (a server program) at any of various levels of understanding of the XML-RPC protocol.

Here are some examples of client and server code.

More Information On XML-RPC For C/C++

This manual is a user's guide, meant to tell you how to write programs using the library. It does not cover such things as how Xmlrpc-c is developed or how to get or install it for use.

The master source of information about Xmlrpc-c is the Xmlrpc-c web site.

The Xmlrpc-c Function Libraries

This is a list of the function libraries Xmlrpc-c provides, with links to the manual for

each.

There are two sets of libraries -- C and C++. You can of course use the C libraries in a C++ program, but you cannot in general mix the two -- you use either the C Xmlrpc-c facilities or the C++ Xmlrpc-c facilities. The C++ libraries depend heavily on the C libraries, but apart from having to install and link to the C libraries, you won't see that from the outside.

The C++ libraries were all new in Xmlrpc-c 1.03 (June 2005).

There is an older C++ facility, which is just a thin wrapper around the C libraries. We do not document it in this manual or recommend it, but it remains part of Xmlrpc-c for backward compatibility. This library consists in the header file **xmlrpc-c/oldcppwrapper.hpp** (fka **XmlRpcCpp.h**) and the library **libxmlrpc_cpp**.

C Libraries

For information common to all the libraries, see General Library Information - C

- libxmlrpc - general XML-RPC facilities, not specific to clients or to servers, such as XML encoding and decoding and XML-RPC value processing.
- libxmlrpc_client - facilities for implementing a client
- libxmlrpc_server - facilities for implementing a server — facilities which are independent of the transport mechanism.
- libxmlrpc_server_abyss - facilities for implementing a server, based on an Abyss HTTP server.
- libxmlrpc_abyss - The Abyss server software.
- libxmlrpc_server_cgi - facilities for implementing a server via CGI scripts under an arbitrary HTTP server (i.e. web server).
- libxmlrpc_util - general programming facilities not specific to XML-RPC, but used by the libraries above, either internally or in their interfaces.

C++ Libraries

For information common to all the libraries, see General Library Information - C++

- libxmlrpc++ - general XML-RPC facilities, not specific to clients or to servers, such as XML encoding and decoding and XML-RPC value processing.
- libxmlrpc_client++ - facilities for implementing a client
- libxmlrpc_server++ - facilities for implementing a server — facilities which are independent of the transport mechanism.
- libxmlrpc_server_abyss++ - facilities for implementing a server, based on an abyss HTTP server.
- libxmlrpc_server_cgi++ - facilities for implementing a server via CGI scripts under an arbitrary HTTP server (i.e. web server).
- libxmlrpc_server_pstream++ - facilities for implementing a pseudo-XML-RPC server that uses a packet stream in place of HTTP and has multi-RPC client/

server connections.

Utility Programs

Xmlrpc-c comes with a few utility programs that you can use to diagnose problems or learn about XML-RPC or Xmlrpc-c. These programs double as examples of how to use the Xmlrpc-c function libraries.

Because the utility programs are not essential to the package, the default install tools don't install them at all. The builder does build them, though, and if you build Xmlrpc-c from source, you will find them all in the **tools/** directory.

You'll also find more complete documentation there.

xmlrpc

xmlrpc is a general purpose XML-RPC client program. It performs one XML-RPC call, which you describe in its command line arguments.

Example:

```
$ xmlrpc http://localhost:8080/RPC2 sample.add i/3 i/5
```

This makes a call to the XML-RPC server at the indicated URL, for the method named "sample.add", with two arguments: the integer 3 and the integer 5.

xmlrpc prints to Standard Output the result of the call, which it gets back from the server.

For details, see the manual.

String arguments look like this:

```
$ xmlrpc http://www.xmlrpc.com/RPC2 method_that_takes_string s/hello
```

If you don't include a type specifier such as "i/" or "s/", **xmlrpc** assumes "s/". So you can use this shortcut, equivalent to the above:

```
$ xmlrpc http://www.xmlrpc.com/RPC2 method_that_takes_string hello
```

xmlrpc_dumpserver

xmlrpc_dumpserver is a server program for analyzing XML-RPC issues. The server it runs executes any RPC by printing to Standard Output the fact that it is executing it and what its parameters are. You can use this to analyze a client program.

For details, see the manual.

Here is an example of the output, which is the result of performing an RPC on the server with the following commands.

```
$ xmlrpc_dumpserver -port=8080 &
$ xmlrpc localhost:8080 mymethod i/7 s/secondparameter
```

Output from server:

```
Running XML-RPC server on TCP Port 8080...
Server has received a call of method 'mymethod'

Number of parameters: 2

Parameter 0:

  Integer: 7

Parameter 1:

  String: 'secondparameter'
```

xmlrpc_dumpserver was new in Xmlrpc-c 1.49 (March 2017).

Alternatives

This section describes some alternatives to using XML-RPC For C/C++.

Other Progamming Languages

There are plenty of facilities to help you create XML-RPC clients and servers in a language other than C or C++. Search on Freshmeat.

It is worth mentioning that with some of these other-language facilities, the client or server is way slower than with XML-RPC for C/C++, because of the nature of the language. For example, I have used the Perl **RPC::XML** modules from CPAN and found a client program that takes 50 milliseconds to run when written with XML-RPC For C And C++ takes 2000 milliseconds to run when done with RPC::XML::Client.

In the case of Perl, there is a middle ground. The **RPC::Xmlrpc_c** modules from

CPAN are based on Perl extensions that use the libraries of XML-RPC For C And C++. One reason **RPC::XML** is so slow is that it is built on top of a stack about 6 layers high, each one implemented in interpreted Perl. With **RPC::Xmlrpc_c**, all those layers except the top are implemented as executable machine code, efficiently compiled from C, so you have the same ease of Perl coding, without the slowness.

RPC::Xmlrpc_c is much younger than **RPC::XML**, so doesn't have many features, and in fact does not include any server facilities. But you could add missing features yourself (and, ideally, submit them for inclusion in the **RPC::Xmlrpc_c** package on CPAN, so others can use them).

RPC::Xmlrpc_c was new in December 2006 and needs XML-RPC For C And C++ Release 1.08 or better.

In other interpreted languages, the same hybrid may be possible -- replacing slow interpreted code with executable XML-RPC libraries.

Apache Module

You can make a nice XML-RPC server based on an Apache HTTP server (which may or may not simultaneously be a regular web server) using an Apache module.

There once was an Apache module based on Xmlrpc-c, so you could use the same method code as you do for other Xmlrpc-c-based implementations. It was called **mod_xmlrpc** and is described by a Freshmeat entry, but as of April 2009, the download link is dead.

Another module, also called **mod_xmlrpc**, is distributed via a Sourceforge project, but hasn't been updated since 2001, is undocumented, and looks pretty weak.

An even simpler, though less efficient and more limited way to make an XML-RPC server out of an Apache server is to do it via a CGI script. That script can be written in a variety of languages, but if you write it in C, you can use Xmlrpc-c's libxmlrpc_server_cgi library.

Other RPC Protocols

SOAP and CORBA are common alternatives to XML-RPC. Lots of expensive commercial software helps you use those. There is more to know and more you can do with them.

REST-RPC was invented in 2006 and was meant to be superior to XML-RPC for at least some things. It is easier to use in many ways than XML-RPC. Like XML-RPC, it uses HTTP, and like XML-RPC an RPC's result is XML. But unlike XML-RPC, a call is *not* XML. It is encoded entirely in the query part of a URL. (Example: `http:/test.rest-rpc.org/?_function=GetCart&cart=1563`). In the result, there are no inherent data types; server and client agree on those separately.

JSON-RPC is like XML-RPC using JSON instead of XML, but isn't really an RPC protocol at all. It is a protocol for sending arbitrary messages back and forth between two communicants (whereas in an RPC protocol, the messages must have a request/response relationship). JSON is a way of representing typical program data structures such as integers and arrays in text, and is much simpler than XML-RPC XML elements or indeed any XML. It is far easier for a person to read a JSON-RPC message than to read an XML-RPC message. JSON-RPC was developed after XML-RPC.

Appendices

Introductory Examples

Here, to show you what Xmlrpc-c is, we present example code (almost an entire C program) for a simple XML-RPC client that exploits the Xmlrpc-c libraries, and a corresponding simple XML-RPC server.

There is not enough here for you just to copy and paste it and build the program, because there is more than Xmlrpc-c details to building a program.

The Xmlrpc-c website has a page of full examples for each of these programs, and we provide links to the relevant files along with the examples below.

But even that leaves some work for you to do, figuring out how to compile it with your tools and for your environment.

You can find complete working versions of these, with a working build system, and lots of other examples in the **examples/** directory in the Xmlrpc-c source tree. You will probably have an easier time getting started with Xmlrpc-c starting from those examples than by cutting and pasting the code shown below.

In these examples, the service to be provided is adding of two numbers. You wouldn't do this with RPC in real life, of course, because a program can add two numbers without the help of a remote server. This is just to demonstrate the concept.

Table Of Contents

- Small C Client Example
- Small C Server Example
- CGI C Server Example
- Small C++ Server Example
- Small C++ Client Example

Small C Client Example

Here is an example of C code that implements an XML-RPC client using the highest level facilities of Xmlrpc-c. This client sends a request to add 5 and 7 together to an XMLRPC-C server that is designed to provide the service of adding numbers.

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

    xmlrpc_env env;
    xmlrpc_value * resultP;
    int sum;
    char * const clientName = "XML-RPC C Test Client";
    char * const clientVersion = "1.0";
    char * const url = "http://localhost:8080/RPC2";
    char * const methodName = "sample.add";

    /* Initialize our error-handling environment. */
    xmlrpc_env_init(&env);

    /* Start up our XML-RPC client library. */
    xmlrpc_client_init2(&env, XMLRPC_CLIENT_NO_FLAGS, clientName, clientVersion, NULL, 0);
    [ handle possible failure of above ]

    /* Make the remote procedure call */
    resultP = xmlrpc_client_call(&env, url, methodName,
                 "(ii)", (xmlrpc_int32) 5, (xmlrpc_int32) 7);
    [ handle possible failure of above ]

    /* Print out the sum the server returned */
    xmlrpc_parse_value(&env, resultP, "i", &sum);
    [ handle possible failure of above ]

    printf("The sum  is %d\n", sum);

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

    /* Clean up our error-handling environment. */
    xmlrpc_env_clean(&env);

    /* Shutdown our XML-RPC client library. */
    xmlrpc_client_cleanup();

    return 0;
}
```

See the complete source file **examples/xmlrpc_sample_add_client.c** in the Xmlrpc-c source tree.

Small C Server Example

Now, here is code that implements an XML-RPC server that provides the number-adding service from the previous section.

```
#include <xmlrpc.h>
#include <xmlrpc_server.h>
#include <xmlrpc_server_abyss.h>

static xmlrpc_value *
sample_add(xmlrpc_env *   const envP,
           xmlrpc_value * const paramArrayP,
           void *         const serverContext) {

    xmlrpc_int32 x, y, z;

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

    /* Add our two numbers. */
    z = x + y;

    /* Return our result. */
    return xmlrpc_build_value(envP, "i", z);
}



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

    xmlrpc_server_abyss_parms serverparm;
    xmlrpc_registry * registryP;
    xmlrpc_env env;

    xmlrpc_env_init(&env);

    registryP = xmlrpc_registry_new(&env);

    xmlrpc_registry_add_method(
        &env, registryP, NULL, "sample.add", &sample_add, NULL);

    serverparm.config_file_name = argv[1];
    serverparm.registryP = registryP;

    printf("Starting XML-RPC server...\n");

    xmlrpc_server_abyss(&env, &serverparm, XMLRPC_APSIZE(registryP));

    return 0;
}
```

See the complete source file **examples/xmlrpc_sample_add_server.c** in the Xmlrpc-c source tree.

There's a lot going on under the covers of this example server. What the xmlrpc_server_abyss() statement does is run a whole HTTP server. The function doesn't normally return. The HTTP server runs the **abyss** web server (i.e. HTTP server) program. **abyss** is like the more serious web server program **apache**, but on a much smaller scale. An XML-RPC call is just an HTTP POST request, so while **abyss** was not designed specifically for XML-RPC, it provides much of the function an XML-RPC server needs.

The only way this Abyss web server differs from one you would run to do traditional web serving is that it contains a special handler to call Xmlrpc-c functions to handle an XML-RPC POST request. The server calls that handler for any URI that starts with "/RPC2", which is what XML-RPC URIs conventionally have.

While **abyss** is distributed independently of Xmlrpc-c, Xmlrpc-c contains an old copy of it, somewhat modified. So you don't need to install **abyss** separately.

In this example, you have to provide an example **abyss** configuration file as a program argument. The main thing you need that file for is to specify on which TCP port the server will listen. A single "Port 8080" statement is probably enough. (I say 8080, because in the example client code above, I hardcoded 8080 as the port in the URI the client uses).

There are lots of other ways to use Xmlrpc-c libraries to build XML-RPC clients and servers. The more code you're willing to write, and the more involved in the guts of the protocol you want to get, the more control you can have.

CGI Server Example

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

static xmlrpc_value *
sample_add(xmlrpc_env *   const env,
           xmlrpc_value * const param_array,
           void *         const user_data) {

    xmlrpc_int32 x, y, z;

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

    /* Add our two numbers. */
    z = x + y;
```

```
    /* Return our result. */
    return xmlrpc_int_new(env, z);
}



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

    /* Process our request. */
    xmlrpc_cgi_init(XMLRPC_CGI_NO_FLAGS);
    xmlrpc_cgi_add_method_w_doc("sample.add", &sample_add, NULL,
                                "i:ii", "Add two integers.");
    xmlrpc_cgi_process_call();
    xmlrpc_cgi_cleanup();

    return 0;
}
```

See the complete source file **examples/xmlrpc_sample_add_server_cgi.c** in the Xmlrpc-c source tree.

Small C++ Client Example

Here is an example of C++ code that implements an XML-RPC client using the highest level facilities of Xmlrpc-c. This client sends a request to add 5 and 7 together to an XMLRPC-C server that is designed to provide the service of adding numbers.

The example server above would be a suitable server for this client.

```
...
#include <xmlrpc-c/base.hpp>
#include <xmlrpc-c/client_simple.hpp>

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

    string const serverUrl("http://localhost:8080/RPC2");
    string const methodName("sample.add");

    xmlrpc_c::clientSimple myClient;
    xmlrpc_c::value result;

    myClient.call(serverUrl, methodName, "ii", &result, 5, 7);

    int const sum((xmlrpc_c::value_int(result)));
        // Assume the method returned an integer; throws error if not
```

```
    cout << "Result of RPC (sum of 5 and 7): " << sum << endl;

    return 0;
}
```

See the complete source file **examples/cpp/xmlrpc_sample_add_client.cpp** in the Xmlrpc-c source tree.

To keep it brief, we don't catch any thrown errors, though various parts of this program can throw them.

Small C++ Server Example

Here is C++ code that implements the same kind of XML-RPC server shown in C above.

```
...
#include <xmlrpc-c/base.hpp>
#include <xmlrpc-c/registry.hpp>
#include <xmlrpc-c/server_abyss.hpp>


class sampleAddMethod : public xmlrpc_c::method {
public:
    sampleAddMethod() {}

    void
    execute(xmlrpc_c::paramList const& paramList,
            xmlrpc_c::value *   const  retvalP) {

        int const addend(paramList.getInt(0));
        int const adder(paramList.getInt(1));

        paramList.verifyEnd(2);

        *retvalP = xmlrpc_c::value_int(addend + adder);
    }
};



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

    xmlrpc_c::registry myRegistry;

    xmlrpc_c::methodPtr const sampleAddMethodP(new sampleAddMethod);
```

```
    myRegistry.addMethod("sample.add", sampleAddMethodP);

    xmlrpc_c::serverAbyss myAbyssServer(
        myRegistry,
        8080,              // TCP port on which to listen
        "/tmp/xmlrpc_log"  // Log file
        );

    myAbyssServer.run();
    // xmlrpc_c::serverAbyss.run() never returns
    assert(false);

    return 0;
}
```

See the complete source file in the Xmlrpc-c source tree.

About This Document

This document is part of the XML-RPC For C/C++ project. It is the main user documentation for the project.

The master copy of this document lives at http://xmlrpc-c.sourceforge.io/doc/. The HTML copy there is the original source -- it is hand edited.

This is a living document. It gets updated continuously, both to document changes in Xmlrpc-c and to improve the documentation. It documents all current and past, and, where possible, future releases of Xmlrpc-c. There is no benefit to keeping an old copy of the document to use with an old copy of the code.

Bryan Henderson wrote and published the first draft of this document in November 2004, as an entirely original work. Bryan placed it in the public domain.

Bryan enthusiastically maintains the document. If you have a problem with it, from typos to missing topics, please email Bryan at bryanh@giraffe-data.com.