XML-RPC for PHP

version 2.0

Edd Dumbill
Gaetano Giunta
Miles Lott
Justin R. Miller
Andres Salomon

XML-RPC for PHP: version 2.0

by Edd Dumbill, Gaetano Giunta, Miles Lott, Justin R. Miller, and Andres Salomon

Table of Contents

Chapter 1. Introduction

XML-RPC is a format devised by Userland Software for achieving remote procedure call via XML using HTTP as the transport. XML-RPC has its own web site, www.xmlrpc.com

This collection of PHP classes provides a framework for writing XML-RPC clients and servers in PHP.

Main goals of the project are ease of use, flexibility and completeness.

The original author is Edd Dumbill of Useful Information Company. As of the 1.0 stable release, the project has been opened to wider involvement and moved to SourceForge.

A list of XML-RPC implementations for other languages such as Perl and Python can be found on the www.xmlrpc.com site.

1. Acknowledgements

Daniel E. Baumann

James Bercegay

Leon Blackwell

Daniel Convissor

Geoffrey T. Dairiki

Stefan Esser

James Flemer

Tom Knight

Axel Kollmorgen

Peter Kocks

Daniel Krippner

S. Kuip

A. Lambert

Dan Libby

Arnaud Limbourg

Ernest MacDougal Campbell III

Lukasz Mach

Kjartan Mannes

Ben Margolin

Nicolay Mausz

Justin Miller

Jan Pfeifer

Giancarlo Pinerolo

Douglas Squirrel

Idan Sofer

Anatoly Techtonik

Christian Wenz

Jim Winstead

Przemyslaw Wroblewski

Chapter 2. What's new

Note: items the following list have not yet been fully documented, and as such might not be present in any other chapter in the manual. To find a more detailed description of new functions and methods please take a look at the source code of the library, which is quite well commented in javadoc-like form.

1. 2.0 Release candidate 1

- Support for HTTP proxies (new method: `xmlrpc_client::setProxy()`)
- Support HTTP compression of both requests and responses. Clients can specify what kind of compression they accept for responses between deflate/gzip/any, and whether to compress the requests. Servers by default compress responses to clients that explicitly declare support for compression (new methods: `xmlrpc_client::setAcceptedCompression()`, `xmlrpc_client::setRequestCompression()`). Note that the ZLIB php extension needs to be enabled in PHP to support compression.
- Implement HTTP 1.1 connections, but only if CURL is enabled (added an extra parameter to `xmlrpc_client::xmlrpc_client` to set the desired HTTP protocol at creation time and a new supported value for the last parameter of `xmlrpc_client::send`, which now can be safely omitted if it has been specified at creation time)

  With PHP versions greater than 4.3.8 keep-alives are enabled by default for HTTP 1.1 connections. This should yield faster execution times when making multiple calls in sequence to the same xml-rpc server from a single client.
- Introduce support for cookies. Cookies to be sent to the server with a request can be set using `xmlrpc_client::setCookie()`, while cookies received from the server are found in `xmlrpcresp::cookies()`. It is left to the user to check for validity of received cookies and decide whether they apply to successive calls or not.
- Better support for detecting different character set encodings of xml-rpc requests and responses: both client and server objects will correctly detect the charset encoding of received xml, and use an appropriate xml parser.

  Supported encodings are US-ASCII, UTF-8 and ISO-8859-1.
- Added one new xmlrpcmsg constructor syntax, allowing usage of a single string with the complete URL of the target server
- Convert xml-rpc boolean values into native php values instead of 0 and 1
- Force the `php_xmlrpc_encode` function to properly encode numerically indexed php arrays into xml-rpc arrays (numerically indexed php arrays always start with a key of 0 and increment keys by values of 1)
- Prevent the `php_xmlrpc_encode` function from further re-encoding any objects of class `xmlrpcval` that are passed to it. This allows to call the function with arguments consisting of mixed php values / xmlrpcval objects.
- Allow a server to NOT respond to system.* method calls (setting the `$server->allow_system_funcs` property).
- Implement a new xmlrpcval method to determine if a value of type struct has a member of a given name without having to loop trough all members: `xmlrpcval::structMemExists()`
- Expand methods `xmlrpcval::addArray`, `addScalar` and `addStruct` allowing extra php values to be added to xmlrpcval objects already formed.
- Let the `xmlrpc_client::send` method accept an XML string for sending instead of an xmlrpcmsg object, to facilitate debugging and integration with the php native xmlrpc extension

- Extend the `php_xmlrpc_encode` and `php_xmlrpc_decode` functions to allow serialization and rebuilding of PHP objects. To successfully rebuild a serialized object, the object class must be defined in the deserializing end of the transfer. Note that object members of type resource will be deserialized as NULL values.

  Note that his has been implemented adding a "php_class" attribute to xml representation of xmlrpcval of STRUCT type, which, strictly speaking, breaks the xml-rpc spec. Other xmlrpc implementations are supposed to ignore such an attribute (unless they implement a brain-dead custom xml parser...), so it should be safe enabling it in heterogeneous environments. The activation of this feature is done by usage of an option passed as second parameter to both `php_xmlrpc_encode` and `php_xmlrpc_decode`.

- Extend the `php_xmlrpc_encode` function to allow automatic serialization of is08601-conforming php strings as datetime.iso8601 xmlrpcvals, by usage of an optional parameter

- Added an automatic stub code generator for converting xmlrpc methods to php functions and vice-versa.

  This is done via two new functions: `wrap_php_function` and `wrap_xmlrpc_method`, and has many caveats, with php being a typeless language and all...

  With PHP versions lesser than 5.0.3 wrapping of php functions into xmlrpc methods is not supported yet.

- Allow object methods to be used in server dispatch map

- Added a complete debugger solution, in the `debugger` folder

- Added configurable server-side debug messages, controlled by the new method `xmlrpc_server::SetDebug()`. At level 0, no debug messages are sent to the client; level 1 is the same as the old behaviour; at level 2 a lot more info is echoed back to the client, regarding the received call; at level 3 all warnings raised during server processing are trapped (this prevents breaking the xml to be echoed back to the client) and added to the debug info sent back to the client

- New XML parsing code, yields smaller memory footprint and faster execution times, not to mention complete elimination of the dreaded `eval()` construct, so prone to code injection exploits

- Rewritten most of the error messages, making text more explicative

2. 2.0 Release candidate 2

- Added a new property of the client object: `xmlrpc_client->return_type`, indicating whether calls to the send() method will return xmlrpcresp objects whose value() is an xmlrpcval object, a php value (automatically decoded) or the raw xml received from the server.

- Added in the extras dir. two new library file: `jsonrpc.inc` and `jsonrpcs.inc` containing new classes that implement support for the json-rpc protocol (alpha quality code)

- Added a new client method: `setKey($key, $keypass)` to be used in HTTPS connections

- Added a new file containing some benchmarks in the testsuite directory

3. 2.0 Release candidate 3

- Added to server class the property functions_parameters_type, that allows the server to register plain php functions as xmlrpc methods (i.e. functions that do not take an xmlrpcmsg object as unique param)

- let server and client objects serialize calls using a specified character set encoding for the produced xml instead of US-ASCII (ISO-8859-1 and UTF-8 supported)

- let php_xmlrpc_decode accept xmlrpcmsg objects as valid input

- 'class::method' syntax is now accepted in the server dispatch map

- `xmlrpc_clent::SetDebug()` accepts integer values instead of a boolean value, with debugging level 2 adding to the information printed to screen the complete client request

4. 2.0 final

- Added to the client class the possibility touse Digest andNTLM authentication methods (when using the CURL library) for connecting to servers and NTLM for connecting to proxies
- Added to the client class the possibility to specify alternate certificate files/directories for authentiating the peer with when using HTTPS communication
- Reviewed all examples and added a new demo file, containing a proxy to forward xmlrpc requests to other servers (useful eg. for ajax coding)
- The debugger has been upgraded to reflect the new client capabilities
- All known bugs have been squashed, and the lib is more tolerant than ever of commonly-found mistakes

Chapter 3. System Requirements

The library has been designed with goals of scalability and backward compatibility. As such, it supports a wide range of PHP installs. Note that not all features of the lib are available in every configuration.

The *minimum supported* PHP version is 4.2.

A compatibility layer is provided that allows the code to run on PHP 4.0.4pl1 and 4.1. Note that if you are stuck on those platforms, we suggest you to upgrade as soon as possible.

Automatic generation of xml-rpc methods from php functions is only supported with PHP version 5.0.3 and later (note that the lib will generate some warnings with PHP 5 in strict error reporting mode).

If you wish to use SSL or HTTP 1.1 to communicate with remote servers, you need the "curl" extension compiled into your PHP installation. This is available in PHP 4.0.2 and greater, although 4.0.6 has a bug preventing SSL working, and versions prior to 4.3.8 do not support streamlining multiple requests using HTTP Keep-Alive.

The "xmlrpc" native extension is not required to be compiled into your PHP installation, but if it is, there will be no interference with the operation of this library.

Chapter 4. Files in the distribution

lib/xmlrpc.inc	the XML-RPC classes. `include()` this in your PHP files to use the classes.
lib/xmlrpcs.inc	the XML-RPC server class. `include()` this in addition to xmlrpc.inc to get server functionality
lib/compat/array_key_exists.php, lib/compat/is_a.php, lib/compat/is_scalar.php, lib/compat/var_export.php, lib/compat/vesrions_compare.php	compatibility functions: these files implement the compatibility layer needed to run the library with PHP versions 4.0 and 4.1
demo/server/proxy.php	a sample server implementing xmlrpc proxy functionality.
demo/server/server.php	a sample server hosting various demo functions, as well as a full suite of functions used for interoperability testing. It is used by testsuite.php (see below) for unit testing the library, and is not to be copied literally into your production servers
demo/client/client.php, demo/client/agesort.php, demo/client/which.php	client code to exercise some of the functions in server.php, including the `interopEchoTests.whichToolkit` method.
demo/client/wrap.php	client code to illustrate 'wrapping' of remote methods into php functions.
demo/client/introspect.php	client code to illustrate usage of introspection capabilities offered by server.php.
demo/client/mail.php	client code to illustrate usage of an xmlrpc-to-email gateway using Dave Winer's XML-RPC server at userland.com.
demo/client/zopetest.php	example client code that queries an xmlrpc server built in Zope.
demo/vardemo.php	examples of how to construct xmlrpcval types
demo/demo1.txt, demo/demo2.txt, demo/demo3.txt	XML-RPC responses captured in a file for testing purposes (you can use these to test the `xmlrpcmsg->parseResponse()` method).
demo/server/discuss.php, demo/client/comment.php	Software used in the PHP chapter of The Jellyfish Book to provide a comment server and allow the attachment of comments to stories from Meerkat's data store.
test/testsuite.php, test/parse_args.php	A unit test suite for this software package. If you do development on this software, please consider submitting tests for this suite.
test/benchmark.php	A (very limited) benchmarking suite for this software package. If you do development on this software, please consider submitting benchmarks for this suite.
test/phpunit.php, test/PHPUnit/*.php	An (incomplete) version PEAR's unit test framework for PHP. The complete package can be found at http://pear.php.net/package/PHPUnit
extras/test.pl, extras/test.py	Perl and Python programs to exercise server.php to test that some of the methods work.
extras/workspace.testPhpServer.fttb	Frontier scripts to exercise the demo server. Thanks to Dave Winer for permission to include these. See Dave's announcement of these.

extras/rsakey.pem	A test certificate key for the SSL support, which can be used to generate dummy certificates. It has the passphrase "test."

Chapter 5. Known bugs and limitations

This started out as a bare framework. Many "nice" bits haven't been put in yet. Specifically, very little type validation or coercion has been put in. PHP being a loosely-typed language, this is going to have to be done explicitly (in other words: you can call a lot of library functions passing them arguments of the wrong type and receive an error message only much further down the code, where it will be difficult to understand).

dateTime.iso8601 is supported opaquely. It can't be done natively as the XML-RPC specification explicitly forbids passing of timezone specifiers in ISO8601 format dates. You can, however, use the iso8601_encode() and iso8601_decode() functions to do the encoding and decoding for you.

Very little HTTP response checking is performed (e.g. HTTP redirects are not followed and the Content-Length HTTP header, mandated by the xml-rpc spec, is not validated); cookie support still involves quite a bit of coding on the part of the user.

If a specific character set encoding other than US-ASCII, ISO-8859-1 or UTF-8 is received in the HTTP header or XML prologue of xml-rpc request or response messages then it will be ignored for the moment, and the content will be parsed as if it had been encoded using the charset defined by xmlrpc_defencoding

Very large floating point numbers are serialized using exponential notation, even though the spec explicitly forbids this behaviour. This will not be a problem if this library is used on both ends of the communication, but might cause problems with other implementations.

Support for receiving from servers version 1 cookies (i.e. conforming to RFC 2965) is quite incomplete, and might cause unforeseen errors.

A PHP warning will be generated in many places when using xmlrpc.inc and xmlrpcs.inc with PHP 5 in strict error reporting mode. The simplest workaround to this problem is to lower the *`error_reporting`* level in php.ini.

Chapter 6. Support

1. Online Support

XML-RPC for PHP is offered "as-is" without any warranty or commitment to support. However, informal advice and help is available via the XML-RPC for PHP website and mailing list and from XML-RPC.com.

- The *XML-RPC for PHP* development is hosted on phpxmlrpc.sourceforge.net. Bugs, feature requests and patches can be posted to the project's website.
- The *PHP XML-RPC interest mailing list* is run by the author. More details can be found here.
- For more general XML-RPC questions, there is a Yahoo! Groups XML-RPC mailing list.
- The XML-RPC.com discussion group is a useful place to get help with using XML-RPC. This group is also gatewayed into the Yahoo! Groups mailing list.

2. The Jellyfish Book

Together with Simon St.Laurent and Joe Johnston, Edd Dumbill wrote a book on XML-RPC for O'Reilly and Associates on XML-RPC. It features a rather fetching jellyfish on the cover.

Complete details of the book are available from O'Reilly's web site.

Edd is responsible for the chapter on PHP, which includes a worked example of creating a forum server, and hooking it up the O'Reilly's Meerkat service in order to allow commenting on news stories from around the Web.

If you've benefited from the effort that has been put into writing this software, then please consider buying the book!

Chapter 7. Class documentation

1. xmlrpcval

This is where a lot of the hard work gets done. This class enables the creation and encapsulation of values for XML-RPC.

Ensure you've read the XML-RPC spec at http://www.xmlrpc.com/stories/storyReader$7 before reading on as it will make things clearer.

The `xmlrpcval` class can store arbitrarily complicated values using the following types: `i4 int boolean string double dateTime.iso8601 base64 array struct`. You should refer to the spec for more information on what each of these types mean.

1.1. Notes on types

1.1.1. int

The type `i4` is accepted as a synonym for `int` when creating xmlrpcval objects. The xml parsing code will always convert `i4` to `int`: `int` is regarded by this implementation as the canonical name for this type.

1.1.2. base64

Base 64 encoding is performed transparently to the caller when using this type. Decoding is also transparent. Therefore you ought to consider it as a "binary" data type, for use when you want to pass data that is not 7-bit clean.

1.1.3. boolean

The php values `true` and `1` map to `true`. All other values (including the empty string) are converted to `false`.

1.1.4. string

Characters <, >, ', ", &, are encoded using their entity reference as < > ' " and & All other characters outside of the ASCII range are encoded using their character reference representation (e.g. &#200 for é). The XML-RPC spec recommends only encoding `<` `&` but this implementation goes further, for reasons explained by the XML 1.0 recommendation. In particular, using character reference representation has the advantage of producing XML that is valid independently of the charset encoding assumed.

1.2. Creation

The constructor is the normal way to create an `xmlrpcval`. The constructor can take these forms:

```
$myVal = new xmlrpcval();
;
$myVal = new xmlrpcval($stringVal);
$stringVal;
$myVal = new xmlrpcval($scalarVal, );
$scalarVal;
<initializer>"int" | "boolean" | "string" | "double" | "dateTime.iso8601" |
"base64"</initializer>
;
$myVal = new xmlrpcval($arrayVal, );
$arrayVal;
"array" | "struct";
```

The first constructor creates an empty value, which must be altered using the methods `addScalar`, `addArray` or `addStruct` before it can be used.

The second constructor creates a simple string value.

The third constructor is used to create a scalar value. The second parameter must be a name of an XML-RPC type. Examples:

```
$myInt = new xmlrpcvalue(1267, "int");
$myString = new xmlrpcvalue("Hello, World!", "string");
$myBool = new xmlrpcvalue(1, "boolean");
$myString2 = new xmlrpcvalue(1.24, "string");
```

The fourth constructor form can be used to compose complex XML-RPC values. The first argument is either a simple array in the case of an XML-RPC `array` or an associative array in the case of a `struct`. The elements of the array *must be `xmlrpcval` objects themselves*. Examples:

```
$myArray = new xmlrpcval(
  array(
    new xmlrpcval("Tom"),
    new xmlrpcval("Dick"),
    new xmlrpcval("Harry")
  ),
  "array");

// recursive struct
$myStruct = new xmlrpcval(
  array(
    "name" => new xmlrpcval("Tom", "string"),
    "age" => new xmlrpcval(34, "int"),
    "address" => new xmlrpcval(
      array(
        "street" => new xmlprcval("Fifht Ave", "string"),
        "city" => new xmlrpcval("NY", "string")
      ),
      "struct")
  ),
  "struct");
```

See the file `vardemo.php` in this distribution for more examples.

1.3. Methods

1.3.1. addScalar

```
$ok = $val->addScalar($stringVal);
$stringVal;
$ok = $val->addScalar($scalarVal, );
$scalarVal;
"int" | "boolean" | "string" | "double" | "dateTime.iso8601" | "base64";
```

If *`$val`* is an empty `xmlrpcval` this method makes it a scalar value, and sets that value.

If *`$val`* is already a scalar value, then no more scalars can be added and `0` is returned.

If *`$val`* is an xmlrpcval of type array, *`$scalarval`* is added as its last element.

If all went OK, `1` is returned.

1.3.2. addArray

```
$ok = $val->addArray($arrayVal);
$arrayVal;
```

The argument is a simple (numerically indexed) array. The elements of the array *must be `xmlrpcval` objects themselves*.

Turns an empty `xmlrpcval` into an `array` with contents as specified by *`$arrayVal`*.

If *`$val`* is an xmlrpcval of type array, the elements of *`$arrayVal`* are appended to the existing ones.

See the fourth constructor form for more information.

1.3.3. addStruct

```
$ok = $val->addStruct($assocArrayVal);
$assocArrayVal;
```

The argument is an associative array. The elements of the array *must be `xmlrpcval` objects themselves*.

Turns an empty `xmlrpcval` into a `struct` with contents as specified by *`$assocArrayVal`*.

If *`$val`* is an xmlrpcval of type struct, the elements of *`$arrayVal`* are merged with the existing ones.

See the fourth constructor form for more information.

1.3.4. kindOf

```
$kind = $val->kindOf();
;
```

Returns a string containing "struct", "array" or "scalar" describing the base type of the value. If it returns "undef" it means that the value hasn't been initialised.

1.3.5. serialize

```
$outString = $val->serialize();
;
```

Returns a string containing the XML-RPC representation of this value.

1.3.6. scalarval

```
$scalarVal = $val->scalarval();
;
```

If `$val->kindOf() == "scalar"`, this method returns the actual PHP-language value of the scalar (base 64 decoding is automatically handled here).

1.3.7. scalartyp

```
$typeName = $val->scalartyp();
;
```

If `$val->kindOf() == "scalar"`, this method returns a string denoting the type of the scalar. As mentioned before, `i4` is always coerced to `int`.

1.3.8. arraymem

```
$xmlrpcVal = $val->arraymem($n);
$n;
```

If `$val->kindOf() == "array"`, returns the *$n*th element in the array represented by the value *`$val`*. The value returned is an `xmlrpcval` object.

```
// iterating over values of an array object
for ($i = 0; $i < $val->arraysize(); $i++)
{
  $v = $val->arraymem($i);
  echo "Element $i of the array is of type ".$v->kindOf();
}
```

1.3.9. arraysize

```
$len = $val->arraysize();
;
```

If *`$val`* is an `array`, returns the number of elements in that array.

1.3.10. structmem

```
$xmlrpcVal = $val->structmem($memberName);
$memberName;
```

If `$val->kindOf() == "struct"`, returns the element called *`$memberName`* from the struct represented by the value *`$val`*. The value returned is an `xmlrpcval` object.

1.3.11. structeach

```
list($key, $value) = $val->structeach();
;
```

Returns the next (key, value) pair from the struct, when *`$val`* is a struct. *`$value`* is an xmlrpcval itself. See also structreset().

```
// iterating over all values of a struct object
$val->structreset();
while (list($key, $v) = $val->structeach())
{
  echo "Element $key of the struct is of type ".$v->kindOf();
}
```

1.3.12. structreset

```
$val->structreset();
;
```

Resets the internal pointer for `structeach()` to the beginning of the struct, where *`$val`* is a struct.

1.3.13. structmemexists

```
$exists = $val->structmemexsists($memberName);
$memberName;
```

Returns `TRUE` or `FALSE` depending on whether a member of the given name exists in the struct.

2. xmlrpcmsg

This class provides a representation for a request to an XML-RPC server. A client sends an `xmlrpcmsg` to a server, and receives back an `xmlrpcresp` (see xmlrpc_client->send).

2.1. Creation

The constructor takes the following forms:

```
$msg = new xmlrpcmsg($methodName, $parameterArray);
$methodName;
$parameterArray;
```

Where *`methodName`* is a string indicating the name of the method you wish to invoke, and *`parameterArray`* is a simple php `Array` of `xmlrpcval` objects. Here's an example message to the *US state name* server:

```
$msg = new xmlrpcmsg("examples.getStateName", array(new xmlrpcval(23, "int")));
```

This example requests the name of state number 23. For more information on `xmlrpcval` objects, see xmlrpcval.

Note that the *`parameterArray`* parameter is optional and can be omitted for methods that take no input parameters or if you plan to add parameters one by one.

2.2. Methods

2.2.1. addParam

```
$ok = $msg->addParam($xmlrpcVal);
$xmlrpcVal;
```

Adds the `xmlrpcval` *`xmlrpcVal`* to the parameter list for this method call. Returns TRUE or FALSE on error.

2.2.2. getNumParams

```
$n = $msg->getNumParams();
;
```

Returns the number of parameters attached to this message.

2.2.3. getParam

```
$xmlrpcVal = $msg->getParam($n);
$n;
```

Gets the *n*th parameter in the message (with the index zero-based). Use this method in server implementations to retrieve the values sent by the client.

2.2.4. method

```
$methName = $msg->method();
;
$msg->method($methName);
$methName;
```

Gets or sets the method contained in the XML-RPC message.

2.2.5. parseResponse

```
$response = $msg->parseResponse($xmlString);
```

```
$xmlString;
```

Given an incoming XML-RPC server response contained in the string *`$xmlString`*, this method constructs an `xmlrpcresp` response object and returns it, setting error codes as appropriate (see xmlrpc_client->send).

This method processes any HTTP/MIME headers it finds.

2.2.6. parseResponseFile

```
$response = $msg->parseResponseFile($fileHandle);
$fileHandle;
```

Given an incoming XML-RPC server response on the open file handle *`fileHandle`*, this method reads the data and passes it to `parseResponse`.

This method is useful to construct responses from pre-prepared files (see files `demo1.txt, demo2.txt, demo3.txt` in this distribution). It processes any HTTP headers it finds.

2.2.7. serialize

```
$outString = $msg->serialize();
;
```

Returns the an XML string representing the XML-RPC message.

3. xmlrpc_client

This is the basic class used to represent a client of an XML-RPC server.

3.1. Creation

The constructor accepts one of two possible syntaxes:

```
$client = new xmlrpc_client($server_url);
$server_url;
$client = new xmlrpc_client($server_path, $server_hostname, $server_port,
$transport);
$server_path;
$server_hostname;
$server_port;
$transport;
```

Here are a couple of usage examples of the first form:

```
$client = new xmlrpc_client("http://phpxmlrpc.sourceforge.net/server.php);
$another_client = new xmlrpc_client("https://james:bond@secret.service.com:4443/xmlrpcserver?agent=007");
```

The second syntax does not allow to express a username and password to be used for basic HTTP authorization as in the second example above, but instead it allows to choose whether xmlrpc calls will be made using the HTTP 1.0 or 1.1 protocol.

Here's another example client set up to query Userland's XML-RPC server at *betty.userland.com*:

```
$client = new xmlrpc_client("/RPC2", "betty.userland.com", 80);
```

The *`server_port`* parameter is optional, and if omitted will default to 80 when using HTTP and 443 when using HTTPS (see the xmlrpc_client->send method below).

The *transport* parameter is optional, and if omitted will default to 'http'. Allowed values are either 'http', 'https' or 'http11'. Its value can be overridden with every call to the send method. See the send method below for more details about the meaning of the different values.

3.2. Methods

This class supports the following methods.

3.2.1. send

This method takes the forms:

```
$response = $client->send($xmlrpc_message, $timeout, $transport);
$xmlrpc_message;
$timeout;
$transport;
$responses = $client->send($xmlrpc_messages, $timeout, $transport);
$xmlrpc_messages;
$timeout;
$transport;
```

Where *xmlrpc_message* is an instance of xmlrpcmsg (see xmlrpcmsg), and *response* is an instance of xmlrpcresp (see xmlrpcresp).

If xmlrpc_messages is an array of message instances, responses will be an array of response instances. The client will try to make use of a single system.multicall xml-rpc method call to forward to the server all the messages, unless $client->no_multicall has been previously set to TRUE (see the multicall method below), in which case many consecutive xmlrpc requests will be sent.

The *timeout* is optional, and will be set to 0 (wait for platform-specific predefined timeout) if omitted. This timeout value is passed to fsockopen(). It is also used for detecting server timeouts during communication (i.e. if the server does not send anything to the client for *timeout* seconds, the connection will be closed).

The *transport* parameter is optional, and if omitted will default to 'http'. The only other valid values are 'https', which will use an SSL HTTP connection to connect to the remote server, and 'http11'. Note that your PHP must have the "curl" extension compiled in order to use both these features. Note that when using SSL you should normally set your port number to 443, unless the SSL server you are contacting runs at any other port.

Warning

PHP 4.0.2 or greater is required for SSL functionality. PHP 4.0.6 has a bug which prevents SSL working.

In addition to low-level errors, the XML-RPC server you were querying may return an error in the xmlrpcresp object. See xmlrpcresp for details of how to handle these errors.

3.2.2. multicall

This method takes the form:

```
$responses = $client->multicall($messages, $timeout, $transport, $fallback);
$messages;
$timeout;
$transport;
$fallback;
```

This method is used to boxcar many method calls in a single xml-rpc request. It will try first to make use of the system.multicall xml-rpc method call, and fall back to executing many separate requests if the server returns any error.

msgs is an array of xmlrpcmsg objects (see xmlrpcmsg), and *response* is an array of xmlrpcresp objects (see xmlrpcresp).

The *timeout* and *transport* parameters are optional, and behave as in the send method above.

The *fallback* parameter is optional, and defaults to TRUE. When set to FALSE it will prevent the client to try using many single method calls in case of failure of the first multicall request. It should be set only when the server is known to support the multicall extension.

3.2.3. setAcceptedCompression

```
$client->setAcceptedCompression($compressionmethod);
$compressionmethod;
```

This method defines whether the client will accept compressed xml payload forming the bodies of the xmlrpc responses received from servers. Note that enabling reception of compressed responses merely adds some standard http headers to xmlrpc requests. It is up to the xmlrpc server to return compressed responses when receiving such requests. Allowed values for *compressionmethod* are: 'gzip', 'deflate', 'any' or null (with any meaning either gzip or deflate).

This requires the "zlib" extension to be enabled in your php install. If it is, by default xmlrpc_client instances will enable reception of compressed content.

3.2.4. SetCaCertificate

```
$client->setCaCertificate($certificate, $is_dir);
$certificate;
$is_dir;
```

This method sets an optional certificate to be used used in SSL-enabled communication to validate a remote server with (when the *server_method* is set to 'https' in the client's construction or in the send method and SetSSLVerifypeer has been set to TRUE).

The *certificate* parameter must be the filename of a PEM formatted certificate, or a directory containing multiple certificate files. The *is_dir* parameter defaults to FALSE, set it to TRUE to specify that *ceritificate* indicates a directory instead of a single file.

This requires the "curl" extension to be compiled into your installation of PHP. For more details see the man page for the curl_setopt function.

3.2.5. setCertificate

```
$client->setCaCertificate($certificatefile);
$certificatefile;
```

This method sets the optional certificate and passphrase used in SSL-enabled communication with a remote server (when the *server_method* is set to 'https' in the client's construction or in the send method).

The *certificate* parameter must be the filename of a PEM formatted certificate. The *passphrase* parameter must contain the password required to use the certificate.

This requires the "curl" extension to be compiled into your installation of PHP. For more details see the man page for the curl_setopt function.

3.2.6. setCookie

```
$client->setCookie($name, $value, $path, $domain, $port);
$name;
$value;
$path;
```

```
$domain;
$port;
```

This method sets a cookie that will be sent to the xmlrpc server along with every further request (useful e.g. for keeping session info outside of the xml-rpc payload).

$value is optional, and defaults to null.

$path, $domain and $port are optional, and will be omitted from the cookie header if unspecified. Note that setting any of these values will turn the cookie into a 'version 1' cookie, that might not be fully supported by the server (see RFC2965 for more details).

3.2.7. setCredentials

```
$client->setCredentials($username, $password, $authtype);
$username;
$password;
$authtype;
```

This method sets the username and password for authorizing the client to a server. With the default (HTTP) transport, this information is used for HTTP Basic authorization. Note that username and password can also be set using the class constructor. With HTTP 1.1 and HTTPS transport, NTLM and Digest authentication protocols are also supported. To enable them use the constants `CURLAUTH_DIGEST` and `CURLAUTH_NTLM` as values for the authtype parameter.

3.2.8. setDebug

```
$client->setDebug($debugLvl);
$debugLvl;
```

debugLvl is either `0, 1` or 2 depending on whether you require the client to print debugging information to the browser. The default is not to output this information (0).

The debugging information at level 1includes the raw data returned from the XML-RPC server it was querying (including bot HTTP headers and the full XML payload), and the PHP value the client attempts to create to represent the value returned by the server. At level2, the complete payload of the xmlrpc request is also printed, before being sent t the server.

This option can be very useful when debugging servers as it allows you to see exactly what the client sends and the server returns.

3.2.9. setKey

```
$client->setKey($key, $keypass);
$key;
$keypass;
```

This method sets the optional certificate key and passphrase used in SSL-enabled communication with a remote server (when the *transport* is set to 'https' in the client's construction or in the send method).

This requires the "curl" extension to be compiled into your installation of PHP. For more details see the man page for the `curl_setopt` function.

3.2.10. setProxy

```
$client->setProxy($proxyhost, $proxyport, $proxyusername, $proxypassword,
$authtype);
$proxyhost;
$proxyport;
$proxyusername;
```

```
$proxypassword;
$authtype;
```

This method enables calling servers via an HTTP proxy. The *proxyusername*, *proxypassword* and *authtype* parameters are optional. *Authtype* defaults to CURLAUTH_BASIC (Basic authentication protocol); the only other valid value is the constant CURLAUTH_NTLM, and has effect only when the client uses the HTTP 1.1 protocol.

NB: CURL versions before 7.11.10 cannot use a proxy to communicate with https servers.

3.2.11. setRequestCompression

```
$client->setRequestCompression($compressionmethod);
$compressionmethod;
```

This method defines whether the xml payload forming the request body will be sent to the server in compressed format, as per the HTTP specification. This is particularly useful for large request parameters and over slow network connections. Allowed values for *compressionmethod* are: 'gzip', 'deflate', 'any' or null (with any meaning either gzip or deflate). Note that there is no automatic fallback mechanism in place for errors due to servers not supporting receiving compressed request bodies, so make sure that the particular server you are querying does accept compressed requests before turning it on.

This requires the "zlib" extension to be enabled in your php install.

3.2.12. setSSLVerifyHost

```
$client->setSSLVerifyHost($i);
$i;
```

This method defines whether connections made to XML-RPC backends via HTTPS should verify the remote host's SSL certificate's common name (CN). By default, only the existence of a CN is checked. *$i* should be an integer value; 0 to not check the CN at all, 1 to merely check for its existence, and 2 to check that the CN on the certificate matches the hostname that is being connected to.

3.2.13. setSSLVerifyPeer

```
$client->setSSLVerifyPeer($i);
$i;
```

This method defines whether connections made to XML-RPC backends via HTTPS should verify the remote host's SSL certificate, and cause the connection to fail if the cert verification fails. *$i* should be a boolean value. Default value: TRUE. To specify custom SSL certificates to validate the server with, use the setCaCertificate method.

3.3. Variables

NB: direct manipulation of these variables is only recommended for advanced users.

3.3.1. no_multicall

This member variable determines whether the multicall() method will try to take advantage of the system.multicall xmlrpc method to dispatch to the server an array of requests in a single http roundtrip or simply execute many consecutive http calls. Defaults to FALSE, but it will be enabled automatically on the first failure of execution of system.multicall.

3.3.2. request_charset_encoding

This is the charset encoding that will be used for serializing request sent by the client.

If defaults to NULL, which means using US-ASCII and encoding all characters outside of the ASCII range using their xml character entity representation (this has the benefit that line end characters will not be mangled in the transfer, a CR-LF will be preserved as well as a singe LF).

Valid values are 'US-ASCII', 'UTF-8' and 'ISO-8859-1'

3.3.3. return_type

This member variable determines whether the value returned inside an xmlrpcresp object as results of calls to the send() and multicall() methods will be an xmlrpcval object, a plain php value or a raw xml string. Allowed values are 'xmlrpcvals' (the default), 'phpvals' and 'xml'. To allow the user to differentiate between a correct and a faulty response, fault responses will be returned as xmlrpcresp objects in any case. Note that the 'phpvals' setting will yield faster execution times, but some of the information from the original response will be lost. It will be e.g. impossible to tell whether a particular php string value was sent by the server as an xmlrpc string or base64 value.

Example usage:

```
$client = new xmlrpc_client("phpxmlrpc.sourceforge.net/server");
$client->return_type = 'phpvals';
$message = new xmlrpcmsg("examples.getStateName", array(new xmlrpcval(23, "int")));
$resp = $client->send($message);
if (is_a($resp, 'xmlrpcresp')) echo 'KO. Error: '.$resp->faultCode() else echo 'OK: got '.$resp;
```

For more details about usage of the 'xml' value, see Appendix A.

4. xmlrpcresp

This class is used to contain responses to XML-RPC requests. A server method handler will construct an `xmlrpcresp` and pass it as a return value. This same value will be returned by the result of an invocation of the `send` method of the `xmlrpc_client` class.

4.1. Creation

```
$resp = new xmlrpcresp($xmlrpcval);
$xmlrpcval;
$resp = new xmlrpcresp(0, $errcode, $err_string);
0;
$errcode;
$err_string;
```

The first syntax is used when execution has happened without difficulty: *`$xmlrpcval`* is an `xmlrpcval` value with the result of the method execution contained in it. Alternatively it can be a string containing the xml serialization of the single xml-rpc value result of method execution.

The second type of constructor is used in case of failure. *`errcode`* and *`err_string`* are used to provide indication of what has gone wrong. See xmlrpc_server for more information on passing error codes.

4.2. Methods

4.2.1. faultCode

```
$fn = $resp->faultCode();
;
```

Returns the integer fault code return from the XML-RPC response. A zero value indicates success, any other value indicates a failure response.

4.2.2. faultString

```
$fs = $resp->faultString();
;
```

Returns the human readable explanation of the fault indicated by `$resp->faultCode()`.

4.2.3. value

```
$xmlrpcVal = $resp->value();
;
```

Returns an `xmlrpcval` object containing the return value sent by the server. If the response's `faultCode` is non-zero then the value returned by this method should not be used (it may not even be an object).

Note: if the xmlrpcresp instance in question has been created by an `xmlrpc_client` object whose `return_type` was set to 'phpvals', then a plain php value will be returned instead of an `xmlrpcval` object. If the `return_type` was set to xml, an xml string will be returned (see the return_type var above for more details).

4.2.4. serialize

```
$outString = $resp->serialize();
;
```

Returns an XML string representation of the response.

5. xmlrpc_server

The implementation of this class has been kept as simple to use as possible. The constructor for the server basically does all the work. Here's a minimal example:

```
function foo ($xmlrpcmsg) {
  ...
  return new xmlrpcresp($somexmlrpcval);
}

class bar {
  function foobar($xmlrpcmsg) {
    ...
    return new xmlrpcresp($somexmlrpcval);
  }
}

$s = new xmlrpc_server(
  array(
    "examples.myFunc1" => array("function" => "foo"),
    "examples.myFunc2" => array("function" => "bar::foobar"),
  ));
```

This performs everything you need to do with a server. The single constructor argument is an associative array from method names to function names. The incoming request is parsed and dispatched to the relevant php function, which is responsible for returning a `xmlrpcresp` object, that will be serialized back to the caller. ù

5.1. Method handler functions

Both php functions and static class methods can be registered as xmlrpc method handlers.

The synopsis of a method handler function is:

```
xmlrpcresp $resp = function (xmlrpcmsg $msg)
```

No text should be echoed 'to screen' by the handler function, or it will break the xml response sent back to the client. This applies also to error and warning messages that PHP prints to screen unless the appropriate parameters have been set in the php.in file. Another way to prevent echoing of errors inside the response and facilitate debugging is to use the server SetDebug method with debug level 3.

Note that if you implement a method with a name prefixed by `system.` the handler function will be invoked by the server with two parameters, the first being the server itself and the second being the `xmlrpcmsg` object.

The same php function can be registered as handler of multiple xmlrpc methods.

Here is a more detailed example of what the handler function `foo` may do:

```
function foo ($xmlrpcmsg) {
  global $xmlrpcerruser; // import user errcode value

  $meth = $xmlrpcmsg->method(); // retrieve method name
  $par = $xmlrpcmsg->getParam(0); // decode value of first parameter
  $val = $par->scalarval();

  ...

  if ($err) {
    // this is an error condition
    return new xmlrpcresp(0, $xmlrpcerruser+1, // user error 1
      "There's a problem, Captain");
  } else {
    // this is a successful value being returned
    return new xmlrpcresp(new xmlrpcval("All's fine!", "string"));
  }
}
```

See server.php in this distribution for more examples of how to do this.

Since release 2.0RC3 there is a new, even simpler way of registering php functions with the server. See section 5.7 below

5.2. The dispatch map

The first argument to the `xmlrpc_server` constructor is an array, called the *dispatch map*. In this array is the information the server needs to service the XML-RPC methods you define.

The dispatch map takes the form of an associative array of associative arrays: the outer array has one entry for each method, the key being the method name. The corresponding value is another associative array, which can have the following members:

- `function` - this entry is mandatory. It must be either a name of a function in the global scope which services the XML-RPC method, or an array containing an instance of an object and a static method name (for static class methods the 'class::method' syntax is also supported).
- `signature` - this entry is an array containing the possible signatures (see Signatures) for the method. If this entry is present then the server will check that the correct number and type of parameters have been sent for this method before dispatching it.
- `docstring` - this entry is a string containing documentation for the method. The documentation may contain HTML markup.

Look at the `server.php` example in the distribution to see what a dispatch map looks like.

5.3. Method signatures

A signature is a description of a method's return type and its parameter types. A method may have more than one signature.

Within a server's dispatch map, each method has an array of possible signatures. Each signature is an array of types. The first entry is the return type. For instance, the method

```
string examples.getStateName(int)
```

has the signature

```
array($xmlrpcString, $xmlrpcInt)
```

and, assuming that it is the only possible signature for the method, it might be used like this in server creation:

```
$findstate_sig = array(array($xmlrpcString, $xmlrpcInt));

$findstate_doc = 'When passed an integer between 1 and 51 returns the
name of a US state, where the integer is the index of that state name
in an alphabetic order.';

$s = new xmlrpc_server( array(
  "examples.getStateName" => array(
    "function" => "findstate",
    "signature" => $findstate_sig,
    "docstring" => $findstate_doc
  )));
```

Note that method signatures do not allow to check nested parameters, e.g. the number and type of the members of a struct param cannot be validated.

If a method that you want to expose has a definite number of parameters, but each of those parameters could reasonably be of multiple types, the array of acceptable signatures will easily grow into a combinatorial explosion. To avoid such a situation, the lib defines the global var `$xmlrpcValue`, which can be used in method signatures as a placeholder for 'any xmlrpc type':

```
$echoback_sig = array(array($xmlrpcValue, $xmlrpcValue));

$findstate_doc = 'Echoes back to the client the received value, regardless of its type';

$s = new xmlrpc_server( array(
  "echoBack" => array(
    "function" => "echoback",
    "signature" => $echoback_sig, // this sig guarantees that the method handler will be called with one
    "docstring" => $echoback_doc
  )));
```

Methods `system.listMethods`, `system.methodHelp`, `system.methodSignature` and `system.multicall` are already defined by the server, and should not reimplemented (see Reserved Methods below).

5.4. Delaying the server response

You may want to construct the server, but for some reason not fulfill the request immediately (security verification, for instance). If you pass the constructor a second argument of `0` this will have the desired effect. You can then use the `service()` method of the server class to service the request. For example:

```
$s = new xmlrpc_server($myDispMap, 0);

// ... some code that does other stuff here

$s->service();
```

5.5. Modifying the server behaviour

A couple of methods / class variables are available to modify the behaviour of the server. The only way to take advantage of their existence is by usage of a delayed server response (see above)

5.5.1. setDebug()

This function controls weather the server is going to echo debugging messages back to the client as comments in response body. Valid values: 0,1,2,3, with 1 being the default. At level 0, no debug info is returned to the client. At level 2, the complete client request is added to the response, as part of the xml comments. At level 3, a new PHP error handler is set when executing user functions exposed as server methods, and all non-fatal errors are trapped and added as comments into the response.

5.5.2. allow_system_funcs

Default_value: TRUE. When set to FALSE, disables support for `System.xxx` functions in the server. It might be useful e.g. if you do not wish the server to respond to requests to `System.ListMethods`.

5.5.3. compress_response

When set to TRUE, enables the server to take advantage of HTTP compression, otherwise disables it. Responses will be transparently compressed, but only when an xmlrpc-client declares its support for compression in the HTTP headers of the request.

Note that the ZLIB php extension must be installed for this to work. If it is, `compress_response` will default to TRUE.

5.5.4. response_charset_encoding

Charset encoding to be used for response (only affects string values).

If it can, the server will convert the generated response from internal_encoding to the intended one.

Valid values are: a supported xml encoding (only UTF-8 and ISO-8859-1 at present, unless mbstring is enabled), null (leave charset unspecified in response and convert output stream to US_ASCII), 'default' (use xmlrpc library default as specified in xmlrpc.inc, convert output stream if needed), or 'auto' (use client-specified charset encoding or same as request if request headers do not specify it (unless request is US-ASCII: then use library default anyway).

5.6. Fault reporting

Fault codes for your servers should start at the value indicated by the global `$xmlrpcerruser` + 1.

Standard errors returned by the server include:

`1` Unknown method	Returned if the server was asked to dispatch a method it didn't know about
`2` Invalid return payload	This error is actually generated by the client, not server, code, but signifies that a server returned something it couldn't understand. A more detailed error report is sometimes added onto the end of the phrase above.
`3` Incorrect parameters	This error is generated when the server has signature(s) defined for a method, and the parameters passed by the client do not match any of signatures.
`4` Can't introspect: method unknown	This error is generated by the builtin `system.*` methods when any kind of introspection is attempted on a method undefined by the server.

`5` Didn't receive 200 OK from remote server	This error is generated by the client when a remote server doesn't return HTTP/1.1 200 OK in response to a request. A more detailed error report is added onto the end of the phrase above.
`6` No data received from server	This error is generated by the client when a remote server returns HTTP/1.1 200 OK in response to a request, but no response body follows the HTTP headers.
`7` No SSL support compiled in	This error is generated by the client when trying to send a request with HTTPS and the CURL extension is not available to PHP.
`8` CURL error	This error is generated by the client when trying to send a request with HTTPS and the HTTPS communication fails.
`9-14` multicall errors	These errors are generated by the server when something fails inside a system.multicall request.
`100-` XML parse errors	Returns 100 plus the XML parser error code for the fault that occurred. The `faultString` returned explains where the parse error was in the incoming XML stream.

5.7. 'New style' servers

In the same spirit of simplification that inspired the `xmlrpc_client::return_type` class variable, a new class variable has been added to the server class: `functions_parameters_type`. When set to 'phpvals', the functions registered in the server dispatch map will be called with plain php values as parameters, instead of a single xmlrpcmsg instance parameter. The return value of those functions is expected to be a plain php value, too. An example is worth a thousand words:

```
function foo($usr_id, $out_lang='en') {
  global $xmlrpcerruser;

  ...

  if ($someErrorCondition)
    return new xmlrpcresp(0, $xmlrpcerruser+1, 'DOH!');
  else
    return array(
      'name' => 'Joe',
      'age' => 27,
      'picture' => new xmlrpcval(file_get_contents($picOfTheGuy), 'base64')
    );
}

$s = new xmlrpc_server(
  array(
    "examples.myFunc" => array(
      "function" => "bar::foobar",
      "signature" => array(
        array($xmlrpcString, $xmlrpcInt),
        array($xmlrpcString, $xmlrpcInt, $xmlrpcString)
      )
    )
  ), false);
$s->functions_parameters_type = 'phpvals';
$s->service();
```

There are a few things to keep in mind when using this simplified syntax:

to return an xmlrpc error, the method handler function must return an instance of xmlrpcresp. There is no other way for the server to know when an error response should be served to the client;

to return a base64 value, the method handler function must encode it on its own, creating an instance of an xmlrpcval object;

the method handler function cannot determine the name of the xmlrpc method it is serving, unlike standard handler functions that can retrieve it from the message object;

when receiving nested parameters, the method handler function has no way to distinguish a php string that was sent as base64 value from one that was sent as a string value;

this has a direct consequence on the support of system.multicall: a method whose signature contains datetime or base64 values will not be available to multicall calls;

last but not least, the direct parsing of xml to php values is much faster than using xmlrpcvals, and allows the library to handle much bigger messages without allocating all available server memory or smashing PHP recursive call stack.

Chapter 8. Global variables

Many global variables are defined in the xmlrpc.inc file. Some of those are meant to be used as constants (and modifying their value might cause unpredictable behaviour), while some others can be modified in your php scripts to alter the behaviour of the xml-rpc client and server.

1. "Constant" variables

1.1. $xmlrpcerruser

```
$xmlrpcerruser = 800;
```

The minimum value for errors reported by user implemented XML-RPC servers. Error numbers lower than that are reserved for library usage.

1.2. $xmlrpcI4, $xmlrpcInt, $xmlrpcDouble, $xmlrpcString, $xmlrpcDateTime, $xmlrpcBase64 , $xmlrpcArray, $xmlrpcStruct, $xmlrpcValue

For convenience the strings representing the XML-RPC types have been encoded as global variables:

```
$xmlrpcI4="i4";
$xmlrpcInt="int";
$xmlrpcBoolean="boolean";
$xmlrpcDouble="double";
$xmlrpcString="string";
$xmlrpcDateTime="dateTime.iso8601";
$xmlrpcBase64="base64";
$xmlrpcArray="array";
$xmlrpcStruct="struct";
$xmlrpcValue="undefined";
```

1.3. $xmlrpcTypes, $xmlEntities, $xmlrpcerr, $xmlrpcstr, $xmlrpcerrxml, $xmlrpc_backslash, $_xh, $xml_iso88591_Entities

Reserved for internal usage.

2. Variables whose value can be modified

2.1. xmlrpc_defencoding

```
$xmlrpc_defencoding = "UTF8";
```

This variable defines the character set encoding that will be used by the xml-rpc client and server to decode the received messages, when a specific charset declaration is not found (in the messages sent non-ascii chars are always encoded using character references, so that the produced xml is valid regardless of the charset encoding assumed).

Allowed values: `"UTF8"`, `"ISO-8859-1"`, `"ASCII"`.

Note that the appropriate RFC actually mandates that XML received over HTTP without indication of charset encoding be treated as US-ASCII, but many servers and clients 'in the wild' violate the standard, and assume the default encoding is UTF-8.

2.2. xmlrpc_internalencoding

```
$xmlrpc_internalencoding = "ISO-8859-1";
```

This variable defines the character set encoding that the library uses to transparently encode into valid XML the xml-rpc values created by the user and to re-encode the received xml-rpc values when it passes them to the PHP application. It only affects xml-rpc values of string type. It is a separate value from xmlrpc_defencoding, allowing e.g. to send/receive xml messages encoded on-the-wire in US-ASCII and process them as UTF-8. It defaults to the character set used internally by PHP (unless you are running an MBString-enabled installation), so you should change it only in special situations, if e.g. the string values exchanged in the xml-rpc messages are directly inserted into / fetched from a database configured to return UTF8 encoded strings to PHP. Example usage:

```
<?php

include('xmlrpc.inc');
$xmlrpc_internalencoding = 'UTF-8'; // this has to be set after the inclusion above
$v = new xmlrpcval('Î°á½¹ÏƒÎ¼Îµ'); // This xmlrpc value will be correctly serialized as the greek word 'k
```

2.3. xmlrpcName

```
$xmlrpcName = "XML-RPC for PHP";
```

The string representation of the name of the XML-RPC for PHP library. It is used by the client for building the User-Agent HTTP header that is sent with every request to the server. You can change its value if you need to customize the User-Agent string.

2.4. xmlrpcVersion

```
$xmlrpcVersion = "2.0";
```

The string representation of the version number of the XML-RPC for PHP library in use. It is used by the client for building the User-Agent HTTP header that is sent with every request to the server. You can change its value if you need to customize the User-Agent string.

Chapter 9. Helper functions

XML-RPC for PHP contains some helper functions which you can use to make processing of XML-RPC requests easier.

1. Date functions

The XML-RPC specification has this to say on dates:

> Don't assume a timezone. It should be specified by the server in its documentation what assumptions it makes about timezones.

Unfortunately, this means that date processing isn't straightforward. Although XML-RPC uses ISO 8601 format dates, it doesn't use the timezone specifier.

We strongly recommend that in every case where you pass dates in XML-RPC calls, you use UTC (GMT) as your timezone. Most computer languages include routines for handling GMT times natively, and you won't have to translate between timezones.

For more information about dates, see ISO 8601: The Right Format for Dates, which has a handy link to a PDF of the ISO 8601 specification. Note that XML-RPC uses exactly one of the available representations: CCYYMMDDTHH:MM:SS.

1.1. iso8601_encode

```
$isoString = iso8601_encode($time_t, $utc=0);
$time_t;
$utc=0;
```

Returns an ISO 8601 formatted date generated from the UNIX timestamp *`$time_t`*, as returned by the PHP function `time()`.

The argument *`$utc`* can be omitted, in which case it defaults to `0`. If it is set to `1`, then the function corrects the time passed in for UTC. Example: if you're in the GMT-6:00 timezone and set *`$utc`*, you will receive a date representation six hours ahead of your local time.

The included demo program `vardemo.php` includes a demonstration of this function.

1.2. iso8601_decode

```
$time_t = iso8601_decode($isoString, $utc=0);
$isoString, $utc=0;
```

Returns a UNIX timestamp from an ISO 8601 encoded time and date string passed in. If *`$utc`* is `1` then *`$isoString`* is assumed to be in the UTC timezone, and thus the *`$time_t`* result is also UTC: otherwise, the timezone is assumed to be your local timezone and you receive a local timestamp.

2. Easy use with nested PHP values

Dan Libby was kind enough to contribute two helper functions that make it easier to translate to and from PHP values. This makes it easier to deal with complex structures. At the moment support is limited to int, double, string, array, datetime and struct datatypes; note also that all PHP arrays are encoded as structs, except arrays whose keys are integer numbers staring with 0 and incremented by 1.

These functions reside in `xmlrpc.inc`.

2.1. php_xmlrpc_decode

```
$phpval = php_xmlrpc_decode($xmlrpc_val, $options);
$xmlrpc_val, , $options;
$phpvals = php_xmlrpc_decode($xmlrpcmsg_val, $options);
$xmlrpcmsg_val, , $options;
```

Returns a native PHP value corresponding to the values found in the xmlrpcval *`$xmlrpc_val`*, translated into PHP types. Base-64 and datetime values are automatically decoded to strings.

In the second form, returns an array containing the parameters of the given *`xmlrpcmsg_val`*, decoded to php types.

The *`options`* parameter is optional. If specified, it must consist of an array of options to be enabled in the decoding process. At the moment the only valid option is decode_php_objs. When it is set, php objects that have been converted to xml-rpc structs using the `php_xmlrpc_encode` function and a corresponding encoding option will be converted back into object values instead of arrays (provided that the class definition is available at reconstruction time).

Example:

```
// wrapper to expose an existing php function as xmlrpc method handler
function foo_wrapper($m)
{
  $params = php_xmlrpc_decode($m);
  $retval = call_user_func_array('foo', $params);
  return new xmlrpcresp(new xmlrpcval($retval)); // foo return value will be serialized as string
}

$s = new xmlrpc_server(array(
   "examples.myFunc1" => array(
     "function" => "foo_wrapper",
     "signatures" => ...
  )));
```

2.2. php_xmlrpc_encode

```
$xmlrpc_val = php_xmlrpc_encode($phpval, $options);
$phpval;
$options;
```

Returns an xmlrpcval object populated with the PHP values in *`$phpval`*. Works recursively on arrays and objects, encoding numerically indexed php arrays into array-type xmlrpcval objects and non numerically indexed php arrays into struct-type xmlrpcval objects. Php objects are encoded into struct-type xmlrpcvals, excepted for php values that are already instances of the xmlrpcval class or descendants thereof, which will not be further encoded. Note that there's no support for encoding php values into base-64 values. Encoding of date-times is optionally carried on on php strings with the correct format.

The *`options`* parameter is optional. If specified, it must consist of an array of options to be enabled in the encoding process. At the moment the only valid options are encode_php_objs and auto_dates.

The first will enable the creation of 'particular' xmlrpcval objects out of php objects, that add a "php_class" xml attribute to their serialized representation. This attribute allows the function php_xmlrpc_decode to rebuild the native php objects (provided that the same class definition exists on both sides of the communication)

Example:

```
// the easy way to build a complex xml-rpc struct, showing nested base64 value and datetime values
$val = php_xmlrpc_encode(array(
  'first struct_element: an int' => 666,
  'second: an array' => array ('apple', 'orange', 'banana'),
```

```
  'third: a base64 element' => new xmlrpcval('hello world', 'base64'),
  'fourth: a datetime' => '20060107T01:53:00'
  ), array('auto_dates'));
```

3. Automatic conversion of php functions into xmlrpc methods (and vice versa)

For the extremely lazy coder, two functions have been added that allow to convert a php function into an xmlrpc method, and a remotely exposed xmlrpc method into a local php function. Note that this comes with many caveats.

3.1. wrap_xmlrpc_method

```
$new_php_function_name = wrap_xmlrpc_method(, , , , , );
$client;
$methodname;
$signum;
$timeout;
$protocol;
$funcname;
```

Given an xmlrpc server and a method name, creates a php wrapper function that will call the remote method and return results using native php types for both params and results. The generated php function will return an xmlrpcresp object for failed xmlrpc calls.

The server must support the `system.methodSignature` xmlrpc method call for this function to work.

The *`$client`* param must be a valid xmlrpc_client object, previously created with the address of the target xmlrpc server, and to which the preferred communication options have been set.

The $signum param is optional. Its purpose is to indicate which method signature to use, if the given server method has multiple signatures (defaults to 0).

The *`$timeout`* and *`$protocol`* params are the same as in the `xmlrpc_client::send()` method.

If set, the optional *`$funcname`* indicates which name should be used for the generated function. In case it is left null, the function name will be auto-generated.

In case of an error during generation of the wrapper function, FALSE is returned, otherwise the name of the new function.

Known limitations: server must support `system.methodsignature` for the wanted xmlrpc method; for methods that expose multiple signatures, only one can be picked; for remote calls with nested xmlrpc params, the caller of the generated php function has to encode on its own the params passed to the php function if these are structs or arrays whose (sub)members include values of type base64.

Note: calling the generated php function 'might' be slow: a new xmlrpc client is created on every invocation and an xmlrpc-connection opened+closed. An extra 'debug' param is appended to the parameter list of the generated php function, useful for debugging purposes.

Example usage:

```
$c = new xmlrpc_client('http://phpxmlrpc.sourceforge.net/server.php');

$function = wrap_xmlrpc_method($client, 'examples.getStateName');

if (!$function)
  die('Cannot introspect remote method');
else {
  $stateno = 15;
```

```
  $statename = $function($a);
  if (is_a($statename, 'xmlrpcresp')) // call failed
  {
    echo 'Call failed: '.$statename->faultCode().'. Calling again with debug on';
    $function($a, true);
  }
  else
    echo "OK, state nr. $stateno is $statename";
}
```

3.2. wrap_php_function

```
$dispatch_map_definition = wrap_php_function($funcname, $wrapper_function_name);
$funcname;
$wrapper_function_name;
```

Given a user-defined PHP function, create a PHP 'wrapper' function that can be exposed as xmlrpc method from an xmlrpc_server object and called from remote clients.

The optional *`$wrapper_function_name`* specifies the name that will be used for the auto-generated function.

Since php is a typeless language, to infer types of input and output parameters, it relies on parsing the javadoc-style comment block associated with the given function. Usage of xmlrpc native types (such as datetime.dateTime.iso8601 and base64) in the docblock @param tag is also allowed, if you need the php function to receive/send data in that particular format (note that base64 encoding/decoding is transparently carried out by the lib, while datetime vals are passed around as strings).

Known limitations: requires PHP 5.0.3 +; only works for user-defined functions, not for PHP internal functions (reflection does not support retrieving number/type of params for those); the wrapped php function will not be able to programmatically return an xmlrpc error response; functions returning php objects will generate "special" xmlrpc responses: when the xmlrpc decoding of those responses is carried out by this same lib, using the appropriate param in php_xmlrpc_decode, the objects will be rebuilt.

In short: php objects can be serialized, too (except for their resource members), using this function. Other libs might choke on the very same xml that will be generated in this case (i.e. it has a nonstandard attribute on struct element tags)

Example usage:

```
/**
* State name from state number decoder. NB: do NOT remove this comment block.
* @param integer $stateno the state number
* @return string the name of the state (or error description)
*/
function findstate($stateno)
{
  global $stateNames;
  if (isset($stateNames[$stateno-1]))
  {
    return $stateNames[$stateno-1];
  }
  else
  {
    return "I don't have a state for the index '" . $stateno . "'";
  }
}

// wrap php function, build xmlrpc server
$methods = array();
$findstate_sig = wrap_php_function('findstate');
if ($findstate_sig)
  $methods['examples.getStateName'] = $findstate_sig;
```

```
$srv = new xmlrpc_server($methods);
```

4. Functions removed from the library

The following two functions have been deprecated in version 1.1 of the library, and removed in version 2, in order to avoid conflicts with the EPI xml-rpc library, which also defines two functions with the same names.

To ease the transition to the new naming scheme and avoid breaking existing implementations, the following scheme has been adopted:

- If EPI-XMLRPC is not active in the current PHP installation, the constant `XMLRPC_EPI_ENABLED` will be set to `'0'`
- If EPI-XMLRPC is active in the current PHP installation, the constant `XMLRPC_EPI_ENABLED` wall be set to `'1'`

The following documentation is kept for historical reference:

4.1. xmlrpc_decode

```
$phpval = xmlrpc_decode($xmlrpc_val);
$xmlrpc_val;
```

Alias for php_xmlrpc_decode.

4.2. xmlrpc_encode

```
$xmlrpc_val = xmlrpc_encode($phpval);
$phpval;
```

Alias for php_xmlrpc_encode.

5. Debugging aids

5.1. xmlrpc_debugmsg

```
xmlrpc_debugmsg($debugstring);
$debugstring;
```

Sends the contents of *`$debugstring`* in XML comments in the server return payload. If a PHP client has debugging turned on, the user will be able to see server debug information.

Use this function in your methods so you can pass back diagnostic information. It is only available from `xmlrpcs.inc`.

Chapter 10. Reserved methods

In order to extend the functionality offered by XML-RPC servers without impacting on the protocol, reserved methods are supported in this release.

All methods starting with `system.` are considered reserved by the server. PHP for XML-RPC itself provides four special methods, detailed in this chapter.

Note that all server objects will automatically respond to clients querying these methods, unless the property allow_system_funcs has been set to `false` before calling the `service()` method. This might pose a security risk if the server is exposed to public access, e.g. on the internet.

1. system.listMethods

This method may be used to enumerate the methods implemented by the XML-RPC server.

The `system.listMethods` method requires no parameters. It returns an array of strings, each of which is the name of a method implemented by the server.

2. system.methodSignature

This method takes one parameter, the name of a method implemented by the XML-RPC server.

It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

Multiple signatures (i.e. overloading) are permitted: this is the reason that an array of signatures are returned by this method.

Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply "string, array". If it expects three integers, its signature is "string, int, int, int".

If no signature is defined for the method, a not-array value is returned. Therefore this is the way to test for a non-signature, if *`$resp`* below is the response object from a method call to `system.methodSignature`:

```
$v = $resp->value();
if ($v->kindOf() != "array") {
  // then the method did not have a signature defined
}
```

See the `introspect.php` demo included in this distribution for an example of using this method.

3. system.methodHelp

This method takes one parameter, the name of a method implemented by the XML-RPC server.

It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned.

The documentation string may contain HTML markup.

4. system.Multicall

This method takes one parameter, an array of 'request' struct types. Each request struct must contain a *methodName* member of type string and a *params* member of type array, and corresponds to the invocation of the corresponding method.

It returns a response of type array, with each value of the array being either an error struct (containing the faultCode and faultString members) or the successful response value of the corresponding single method call.

Chapter 11. Examples

The best examples are to be found in the sample files included with the distribution. Some are included here.

1. XML-RPC client: state name query

Code to get the corresponding state name from a number (1-50) from the demo server available on SourceForge

```
$m = new xmlrpcmsg('examples.getStateName',
  array(new xmlrpcval($HTTP_POST_VARS["stateno"], "int")));
$c = new xmlrpc_client("/server.php", "phpxmlrpc.sourceforge.net", 80);
$r = $c->send($m);
if (!$r->faultCode()) {
    $v = $r->value();
    print "State number " . htmlentities($HTTP_POST_VARS["stateno"]) . " is " .
      htmlentities($v->scalarval()) . "<BR>";
    print "<HR>I got this value back<BR><PRE>" .
      htmlentities($r->serialize()) . "</PRE><HR>\n";
} else {
    print "Fault <BR>";
    print "Code: " . htmlentities($r->faultCode()) . "<BR>" .
          "Reason: '" . htmlentities($r->faultString()) . "'<BR>";
}
```

2. Executing a multicall call

To be documented...

Chapter 12. Frequently Asked Questions

1. How to send XML as payload of a method call

Unfortunately, at the time the XML-RPC spec was designed, support for namespaces in XML was not as ubiquitous as it is now. As a consequence, no support was provided in the protocol for embedding XML elements from other namespaces into an xmlrpc request.

To send an XML chunk as payload of a method call or response, two options are available: either send the complete XML block as a string xmlrpc value, or as a base64 value. Since the '<' character i string values is encoded as '<' in the xml payload of the method call, the XML string will not break the surrounding xmlrpc, unless characters outside of the assumed character set are used. The second method has the added benefits of working independently of the charset encoding used for the xml to be transmitted, and preserving exactly whitespace, whilst incurring in some extra message length and cpu load (for carrying out the base64 encoding/decoding).

2. Is there any limitation on the size of the requests / responses that can be successfully sent?

Yes. But I have no hard figure to give; it most likely will depend on the version of PHP in usage and its configuration.

Keep in mind that this library is not optimized for speed nor for memory usage. Better alternatives exist when there are strict requirements on throughput or resource usage, such as the php native xmlrpc extensions (see the PHP manual for more information).

Keep in mind also that HTTP is probably not the best choice in such a situation, and XML is a deadly enemy. CVS formatted data over socket would be much more efficient.

If you really need to move a massive amount of data around, and you are crazy enough to do it using phpxmlrpc, your best bet is to bypass usage of the xmlrpcval objects, at least in the decoding phase, and have the server (or client) object return to the calling function directly php values (see `xmlrpc_client::return_type` and `xmlrpc_server::functions_parameters_type` for more details).

3. My server (client) returns an error whenever the client (server) returns accented characters

To be documented...

4. My php error log is getting full of "deprecated" errors on different lines of xmlrpc.inc and xmlrpcs.inc

This happens when the PHP in usage is version 5, and the error reporting level is set to include `E_STRICT` errors. Since the main development platform of the library remains (for the time being) PHP 4, there are no plans to fix this asap. The best workaround is to set the error reporting level to `E_ALL ^ E_STRICT`.

Appendix A. Integration with the PHP xmlrpc extension

To be documented more...

In short: for the fastest execution possible, you can enable the php native xmlrpc extension, and use it in conjunction with phpxmlrpc. The following code snippet gives an example of such integration

```
/*** client side ***/
$c = new xmlrpc_client('http://phpxmlrpc.sourceforge.net/server.php');
// tell the client to return raw xml as response value
$c->return_type = 'xml';
// let's the native xmlrpc extension take care of encoding request parameters
$r = $c->send(xmlrpc_encode_request('examples.getStateName', $_POST['stateno']));
if (!$r->faultCode()) {
  // HTTP request OK, but XML returned from server not parsed yet
  $v = xmlrpc_decode($r->value());
  // check if server sent an error response
  if (is_array($v) && array_key_exists('faultCode'))
    echo 'Got error '.$v['faultCode'];
  else
    echo'Got response: '.htmlentities($v);
} else {
  // HTTP transport error
  echo 'Got error '.$r->faultCode();
}
```

Appendix B. 'Enough of xmlrpcvals!': new style library usage

To be documented...

In the meantime, see docs about xmlrpc_client::return_type and xmlrpc_server::functions_parameters_types, as well as php_xmlrpc_encode and php_xmlrpc_decode