Internet-Draft
draft-salzer-xmlplusrpc-01.txt
Category: Experimental
Expires: January 2005

Mario Salzer

July 2004

XML+RPC - XML marshalled Remote Procedure Calls

(DRAFT)

Status of this Memo

Copyright Notice

Abstract

This document describes a method to make use of remotely available application logic and data processing by sending explicit procedure calls encoded in simple and plattform-independent XML messages over a HTTP connection.

Despite other RPC (Remote Procedure Call) implementations it is not encoded as binary data, and concentrates on the most basic functionality. Therefore it is easier to implement and compatible with various programming languages and data representation systems.

Table of Contents

1. Introduction . . . . . $
1.1 Purpose . . . . . $
1.2 Relationship to "XML-RPC" . . . . . $
1.3 Requirements . . . . . $
1.4 Terminology . . . . . $
2. Remote Procedure Calls
2.1 Overall Operation
2.2 Example Call
3. Message Body Syntax
3.1 XML Compliance
3.1.1 Restricted XML Syntax
3.1.2 Character Encoding
3.1.3 No DTD
3.2 Request Messages
3.3 Response Massages
3.4 Error Responses
3.5 Simple Data Types
3.5.1 Boolean
3.5.2 Integer
3.5.3 Double
3.5.4 Date And Time
3.5.5 String
3.5.6 Binary Base64
3.6 Aggregate Data Types
3.6.1 Array
3.6.2 Struct
3.7 Custom Data Types
4. Transportation Over HTTP
4.1 General Discussion
4.1.1 MIME Media Type "application/rpc+xml"
4.1.2 Charset parameter
4.2 Requests
4.3 Responses
4.3.1 Transport Layer Errors
4.4 HTTP Transport Compression
4.4.1 Transport Feature Handshake Requests
4.5 Server Requirements
5. Implementation Notes
5.1 Error Response Codes
5.2 Standardized System Methods
5.2.1 system.listMethods
5.2.2 system.methodSignature
5.2.3 system.multiCall
5.2.4 system.dataTypes

5.3 Compatibility with XML-RPC
5.3.1 Older Media Type
5.3.2 Use Of A XML-Parser
5.4 Simplifications
5.4.1 HTTP/1.0 To Avoid Chunked Encoding
6. IESG Section
6.1 IANA Media Type Registration
6.2 Security Considerations
7. Acknowledgements
8. Appendix
9. (draft TODO list)

1. Introduction

1.1 Purpose

Remote Procedure Calls allow to share out programm logic and even data storage across multiple networked hosts. The concept of these calls is very similar to that of machine local procedure calls, which often are encoded in machine code and call procedures based on the machines memory access model and processor registers.

Unlike local procedure calls and most earlier RPC implementations XML+RPC does not use binary coding to transform send data, but instead uses a XML format for cross-plattform compatibility. The overall design goal was to make it easy to implement and use. It provides only the most basic data types to be useful in conjunction with nearly all programming languages and computer plattforms, which often differ wideley in how data types are internally represented.

1.2 Relationship to "XML-RPC"

The protocol described within this document is almost identical to of UserLands [MINUS] specification as written by Dave Winer, but that some clearifications were added. Much care has been taken to keep all older implementations of that specification compatible.

"XML-RPC" is a registered trademark of UserLand. (?)

1.3 Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

An implementation is conforming to this specification if it implements all requirements of this document expressed with "MUST", and all forbidden misinterpretations marked with "MUST NOT" were

absent.

This document also makes use of the ABNF schemas as described in [RFC2234] for clear specification of the XML+RPC message body syntax.

A few paragraphes also utilize regular expressions as introduced by the Perl programming language to fully define the syntax of other constructs.

1.4 Terminology

Multiple technical and RFC related terms are used within this document without prior explanation. However here is a list of terms specifically used in this document, which are believed to require definition first.

method
: A remotely callable procedure.

gateway
: An installation of the XML+RPC interface, which provides methods, procedures and funtionality for remote access.

procedure
: A function that is callable by using the XML+RPC interface installed on or as webserver.

server
: Means the actual implementation of the XML+RPC interface, which often however is not a server daemon on its own, but implemented as CGI script and running below an existing server.

tag
: A "tag" is a XML token enclosed in angle brackets. Often this is also code a "node".

vendor
: One of the various organisations, companys and individuals that provide an implementation of XML+RPC. For example Apache.org or UserLand.

whitespace
: Is the class of invisible characters, which includes the space (%x20) and tabulator (%x08) as well as carriage return (%x0D) and the new line character (%x0A).

2. Remote Procedure Calls

2.1 Overall Operation

A Remote Procedure Call takes place between two machines. One of them usually provides functionality the other likes to access. We further speak of the server for the machine providing methods and the client that sends processing requests to it.

A RPC is initiated by a client if its programm logic dictates or requires to invoke a procedure on a remote machine. It then
X constructs a request from the available data, often by using a
X specialised XML+RPC library. The required data is made up of the full canonical name of the remote procedure and all the parameter data it requires. The parameter data thereby must first be converted from the machine representation into XML document text strings.

The request is then packaged into a HTTP request, which is send to a remote server to process it. If the server accepts XML+RPC requests at the specified address (URL) it decodes it the XML document text stream into a structure useful on the server machine and within the implemented programming language. Then it of course tries to match the requested method name against a list of available ones and additionally checks parameter types against the expected ones so strong typed languages wouldn't be breaked.

If the called procedure finishes the server encodes the request or an error message again into XML+RPC and sends this back to the client. The client then decodes the message again and transforms the result data back into its machine representation and continues to use that data within the stopped programm logic.

Of course Remote Procedure Calls could also be implemented asynchronous, but it is often more convinient that the client stops programm execution while the contacted server works on the XML+RPC request.

2.2 Example Call

As already explained XML+RPC works by encoding requests into XML and transferring them over HTTP. This short example tries to show how a such a call often works.

We assume that the client is running a programm written in the C programming language and wishes to execute remote procedure code on another machine while it processes following code:

```
  char *var;
  var = xmlrpc("http://example.com/rpc.cgi",
               "server.sprintf", "%s - %i", "Hello World!", 2);
```

The examplary XML+RPC library expressed as xmlrpc() call here could then encode the call to the remote procedure "server.sprintf" with exactly three parameters as follows:

```
POST /rpc.cgi HTTP/1.0
Host: example.com
Accept: application/rpc+xml, text/xml
Accept-Encoding: gzip, deflate
Content-Type: application/rpc+xml
Content-Length: xxxx

<?xml version="1.0"?>
<methodCall>
  <methodName>server.sprintf</methodName>
  <params>
     <param>
       <value><string>%s - %i</string></value>
     </param>
     <param>
       <value><string>Hello World!</string></value>
     </param>
     <param>
       <value><int>2</int></value>
     </param>
  </params>
<methodCall>
```

Server side this call would likeley be accepted as it conforms to the XML+RPC syntax, and the destination function "system.printf" would be executed in a scripting language. After letting the procedure return a result, the XML+RPC server would then likely construct an answer like this:

```
HTTP/1.0 200 OkiDoki
Server: Apache/2.0.86 OpenSSL/0.99.9.9j-9
Accept: application/rpc+xml
Accept-Encoding: deflate, gzip
Content-Type: application/rpc+xml
Content-Length: xxxx

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>Hello World! - 2</string></value>
    </param>
  </params>
</methodResponse>
```

The client side XML+RPC library would place that return value into where it was requested.

3. Message Body Syntax

3.1 XML compliance

A compliant implementation MUST obey the [XML] specification when constructing and parsing procedure calls marshalled as described by this document.

Because there exist various implementations that don't use a fully bloated and standards compliant XML parser and there are certain (security) restrictions for XML+RPC message processing on the server side, this specification makes a few more recommendations on which XML features to avoid, if possible.

3.1.1 Restricted XML Syntax

Namespaces are not allowed, and no XML element attributes should be used as neither the [MINUS] description or this document allow them for any of the data type tags. Implementations should reject all messages that contain unknown or unsupported tags or attributes, which after all could completely distort a requests data.

CDATA sections should be used with care (only in <string> tags) and only XML Entities & and < are required to be encoded, but ">" and """ and "'" have to be decoded as well.

3.1.2 Character Encoding

When following the [MINUS] description the default charset= of HTTP transfered XML-RPC calls has been Latin-1. This specification however advises to use a MIME Type from the IETF "application" tree, in which case no default charset shall be assumed. Thus only the encoding= parameter of the initial <?xml?> processing instruction has to be evaluated.

The default charset for [XML] then is UTF-8, and implementors are recommended to use that preferrably and always include the encoding= parameter in the initial XML processing instruction, even if the [XML] standard allows plain ASCII, ISO-8859-1 and all flavours of UTF-16 as well.

3.1.3 ??? (or move below)

Implementors may be tempted to define XML entities like &false; or &true; for use in <boolean> data type tags, but as XML+RPC was created to be processed by applications only, this is considered unnecessary bloat syntax and again a slowdown and should therefore be avoided in general.

3.1.4 No DTD

XML+RPC messages SHOULD NOT contain or reference a DTD, especially not reference one that was located remotely.
(<s>link</s> to the warning article about remotely located DTDs, mangling of XML document meaning caused by entity overrides...)

3.2 Request Messages

A XML+RPC request message is supposed to invoke a remote procedure, and therefore names the remote method to be activated with all the parameters it expects. The structure of a request message is always as follows:

```
XML-processing-instr = '<?xml version="1.0" encoding="UTF-8"?>'

Request-Message =  XML-processing-instr
                   "<methodCall>"
                   method-name
                   parameters
                   "</methodCall>"

method-name = "<methodName>" name "</methodName>"

parameters = "<params>" *(single-parameter) "</params>"

single-parameter = "<param>" (value) "</param>"
```

Hereby only the <methodName> tag is not permitted to carry any count of whitespace. (value) is defined later in the Data Types paragraph.

3.3 Response Massages

Unless a XML+RPC call resulted in an error, a server response MUST have following format:

```
Response-Message = XML-processing-instruction
                   "<methodResponse>"
                   parameters
                   "</methodResponse>"
```

XX The <params> sub node MUST always be present. It always contains
XX exactly one <param> node.

3.4 Error Responses

In case of an error a server will sent a message that differes from the above specified format:

```
Error-Response = XML-processing-instruction
                   "<methodResponse>"
```

```
                    fault
                    "</methodResponse>"

  fault = "<fault>"
          failure-struct
          "</fault>"
```

Where the failure-struct is a <value> node containing a <struct> node as described later. The struct then contains two elements, the first being named "faultCode" and associtated with an integer, and the second is a string and associated by "faultString".

The string thereby corresponds to the returned error number, but both are application dependant, except for the error response numbers described in the section 5.?.? of this document.

3.5 Simple Data Types

Data and values encoded in XML+RPC messages are always of a certain type. The plattform and programming language used by the server and the client, and therefore all system depended types must first be converted into a string expression suitable for almost human-readable form for transportation inside the XML stream. Data types are always mapped to a XML+RPC type and encoded into and from a string representation.

Only a few data types are defined by this document, in order to provide high interoperability between systems and programming languages, that often could handle a far larger amount and much more complex data types. Therefore programmers must take care to convert the used data structures in the basic types defined by XML+RPC.

Every value is encoded as string (from its processor- or language-dependant representation) and written as string into a data type tag additionally enclosed in a <value> tag:

```
  value = "<value>"
          data-type-tag
          "<value>"

  data-type-tag = (data-type-string | data-type-integer
                   | data-type-boolean | data-type-double
                   | data-type-base64 | data-type-datetime
                   | data-type-array | data-type-struct )
```

The "<value>" tag

3.5.1 Boolean

Boolean values are in many programming languages just represented

by a value of 0 mapped to false and some integer different from 0 as being true. But in order to interoperate reliably and to provide type checking on either side of the participating implementations, XML+RPC uses an explicit boolean type.

```
data-type-boolean = "<boolean>"
                    (0 | 1)
                    "</boolean>"
```

Where 0 corresponds to false and 1 represents the true value. There is no whitespace permitted between the opening and closing tag.

3.5.2 Integer

Integer values in XML+RPC can contain 32 bit signed values, that is values in the range from (1 - 2^31) to (0 + 2^31) or more precisely from -2147483647 to 2147483648. The string of decimal numbers that make up the string representation of a machines integer value representation may be preceeded by a minus character, but also can have a plus sign in front.

```
integer-value = 0*1 ("+" | "-")  integer-string

data-type-integer = "<int>" integer-value "</int>"
                    | "<i4>" integer-value "</i4>"
```

Here the data-type tag can be "<int>", but "<i4>" is allowed for backwards compatiblity with [MINUS].

Whitespace is again not permitted between the opening and the closing data type tags.

3.5.3 Double

Floating point values can be transmitted using the "<double>" data type tag. The sender must first encode the value from its machine representation into a string only consisting of at least one digit before and at least one digit after a full stop charachter (%x25). A minus or plus sign may however preceed the string representation of the floating point value.

```
double-string-representation =~ /^[+-]?\d+[.]\d+$/

data-type-double = "<double>"
                   double-string-representation
                   "</double>"
```

XX Size restrictions for compatibility??

Whitespace is again not permitted to occour within this data type

tag.

3.5.4 Date And Time

Combined data and time values can be transported over XML+RPC using the "<dateTime.iso8601>" date type, with the time value encoded according to [ISO8601]. Basically it is a 14 octets string built out of the 4 year number digits followed by two digits for the month and two for the day, then a literal "T" character and the hour, minutes and seconds with two digits each but separated by colons.

```
iso8601-time-string =~ /^(\d{4})(\d\d)(\d\d)T(\d\d):(\d\d):(\d\d)$/

data-type-datetime = "<dateTime.iso8601>"
                     iso8601-time-string
                     "</dateTime.iso8601>"
```

The text node of the <dateTime.iso8601> tag may again not contain any whitespace. Differently to the original [MINUS] specification, the date and time values of XML+RPC always MUST be specified according to the [UTC].

3.5.5 String

A string can be encoded in a XML+RPC message by enclosing it in the "<string>" tag. As outlined in section 3.1.2 a string may contain any combination of octets, but should be preferrably in the UTF-8 encoding. Also a string MAY contain binary data, and is not limited in length. Only the XML entities "<", ">" and "&" are to be decoded when receiving a "<string>" data value.

Unlike all previous data type tags, the string tags text node MAY contain whitespace, which then however MUST NOT be ignored by the recipant, but instead then belongs to the enclosed character string data stream.

```
string-data-type = "<string>"
                   string-stream
                   "</string>"
```

The older XML-RPC specification also allowed to leave out the "<string>" data type tag and to write the string text node directly into the "<value>" tag. Messages encoded according to this document MUST NOT use this syntax, and never enclose a string value without a corresponding <string> type tag around. However for compatibility reasons implementors MAY choose to accept such uncleanly packaged values.

There is no size restriction for <string> content, and this data type SHOULD generally preferred over <base64> unless you are intentionally

transporting real binary content.

3.5.6 Binary Base64

XX The <base64> data type was introduced in XML-RPC before the whole
XX stream was later defined to be binary-safe. Therefore the <base64>
XX data type, originally introduced for carrying binary data within
the messages as almost outdated today and of little value for XML+RPC transportation over HTTP. However if a transport other than HTTP is to be used, then the <base64> data type MAY be preferred over the <string> type.

```
base64-data-type = "<base64>"
                   base64-string-stream
                   "</base64>"
```

The base64-string-stream is a string encoded with the base64 method, and thus only contains characters in the range from %x30-%x39 and %x41-%x5A and %x61-%x7A but also %x3D and OPTIONAL whitespace characters.

3.6 Aggregate Data Types

Besides the most basic data types most programming languages also provide more complex compound variable types. XML+RPC does however not allow each possible type to be used, but instead concentrates on the minimum of interoperable set of types. The two most important complex data types choosen to be provided by XML+RPC and XML-RPC are the array and the struct constructs.

Like the basic types, the more complex constructs like array and struct are always enclosed in a <value> tag, but itself contain more than one subnode.

3.6.1 Array

An array is an ordered list of data values, with each of them having the same data type then again. Arrays are provided by most programming languages, even if they sometimes are called very differently.

```
data-type-array = "<array>"
                  "<data>"
                  *(value)
                  "</data>"
                  "</array>"
```

Please note that the "<array>" tag itself always has the redundant "<data>" sub node, which always MUST be present, regardless if there

are any values in the array. The number of values is not limited, and an implicit numbering is assumed.

The values of an array may itself be any simple or aggregate data type, but all MUST be of the same type for compatibility reasons with strong typed programming languages.

3.6.2 Struct

The <struct> data type corresponds to the "struct" in C and to the "hash arrays" in most interpreted and scripting languages. A struct has multiple entries, each assigned a name and a value. The order of the struct members is insignificant, and at least one member MUST be present.

```
data-type-struct = "<struct>"
                     1*(struct-member)
                   "</struct>"

struct-member = "<member>"
                  "<name>"  token  "</name>"
                  (value)
                "</member>"
```

The "<name>" subnode MUST preceed the also always present "<value>" sub node in each "<member>" tag. The struct member name assigned withing the "<name>" tag again MUST NOT contain any whitespace, and should be moreover only constructed of letters, numbers and the underscore to meet naming requirements of most programming languages.

3.7 Custom Data Types

Multiple incompliant [MINUS] implementations already arised, because of the wish to transfer data of programming language specific types that aren't generic enough to be supported by an inter-plattform compatible specification like this. Implementors are strictly advised not to introduce incompatibilites by inventing new data type tags. If at all necessary, the following workaround SHOULD be used to represent more complex and system specific data types.

```
user-data-type = "<struct>"
                    "<member>"
                       "<name>type</name>"
                       "<value>"  (string-data-type)  "</value>"
                    "</member>"
                    "<member>"
                       "<name>value</name>"
                       (value)
                    "</member>"
                 "</struct>"
```

A struct with exactly two elements ("type" and "value") should be used as placeholder for any non-standard data type. The "value" member can itself be of any other data type, whatever would be most appropriate for it. Using this scheme the XML+RPC message doesn't get invalid if custom types get introduced. It can however easily be deciphered into language and system specific representations if the reciever knows about the negotiated "type" identifier string and how to unpack the "value" member into its system representation.

4. Transportation Over HTTP

4.1 General Discussion

This document only specifies and discusses, how XML+RPC messages are to be transmitted over HTTP. Still it is possible to use different transport channel, but this is out of the scope of this file.

When a client makes a RPC to a server, it encodes the method name and call parameters into a message as described in the previous sections, and transfers that message to the server using a POST method request as defined by the HTTP specification [RFC2616]. It must correctly set multiple HTTP request headers and also the response send by the server, which needs to obey to likewise strict rules.

The meta data required for compliant transmission over HTTP is discussed in the following paragraphs.

4.1.1 MIME Media Type "application/rpc+xml"

All messages conforming to this specification SHOULD be sent using a MIME type of "application/rpc+xml". For compatibility with older [MINUS] compliant servers, an implementation MAY however obey this rule and use "text/xml" instead for compatibility reasons. This type is to be specified within the "Content-Type" header field for both, requests and responses.

This media type is to be registered with the IANA later in this document. The choosen semantics specifically conforms to [RFC3023] and indicates compliant software that the content is a XML derrived format.

4.1.2 Charset Parameter

A MIME type charset= parameter should only be specified when using "text/xml" in compatibility mode. For "application/rpc+xml" no implicit charset exists and the XML processing instruction soley defines the used character encoding.

4.2 Requests

A request MUST be submitted using the POST request method. The destination URL to pass the request to is not specified by this document and is installation dependend, while of course servers could releay messages merely based on the previously defined very specific media type as well.

In accordance to [RFC2616] a POST request must not only have a MIME Content-Type header field, but also a Content-Length field, which specifies the length of data transferred in the body of the request. A typical request could therefore look like (each line separated by %x0D%x0A and the body example shortened):

```
POST /cgi-bin/plus.cgi HTTP/1.0
User-Agent: plus-client/2.5.2
Accept: application/rpc+xml, text/xml; q=0.01
Accept-Encoding: deflate, gzip
Content-Type: application/rpc+xml
Content-Length: 4321

<?xml version="1.0" encoding="UTF-8"?>
<metho...
```

A client may include all header fields as defined in [RFC2616] and MUST always indicate the correct media type. And moreover a client is advised to provide a useful Accept: and Accept-Encoding: field for negotiation with the server.

The initial request may also be compressed, but many implementations do not process that correctly and simply fail. HTTP compliant Web server would answer at least with an error 415 response.

4.3 Responses

Likewise a server response MUST obey all rules of the transport protocol, and include a correct MIME media type describing the body message. For compatiblity with [MINUS] compliant clients a server MAY send a Content-Type: header of "text/xml" if the clients initial request was of that type and its Accept: header didn't indicate that it supported XML+RPC fully. A typical response would look like:

```
HTTP/1.1 200 OK
Date: Mon, 19 Jan 2004 17:17:17 GMT
Accept: application/rpc+xml, text/xml
Accept-Encoding: gzip, deflate
Connection: close
Server: Apache/2.0.94 OpenSSL/0.9.9i-9
```

```
  X-Server: plus-server/2.5.2
  Content-Type: application/rpc+xml
  Content-Encoding: gzip
  Content-Length: 1234

  _?H@h1K@@CstxD$&rQx
  ...
```

In this example the body data was compressed according to the acceptable compression methods specified by the previous examples` client. Implementors are advised to add support for content-coding according to [RFC2616] to reduce bandwidth use for data intensive remote procedure calls.

5. Implementation Notes

5.3 Error Response Codes

The <struct> returned with a <fault> response always gives an error number together with a human readable error description. Since negotiation [FAULTC] on the most often used error codes, the range from -32768 till -32000 is reserved for generic error codes and should not be used for application error codes.

The currently registered and known error results are as follows:

Number	Error string
-32300	Transport error
-32400	System error
-32500	Application error
-32600	(Server) - invalid message format
-32601	(Server) - requested method does not exists
-32602	(Server) - invalid method parameters
-32603	(Server) - internal XML-RPC error
XX -32604	Too many parameters

```
   +--------+-----------------------------------------------------+
XX | -32605 |  Parameter type mismatch                             |
   +--------+-----------------------------------------------------+
   | -32700 |  (Parsing) - Not well-formed XML                     |
   +--------+-----------------------------------------------------+
   | -32701 |  (Parsing) - Unsupported encoding - XML parsers only  |
   |        |  must support UTF-8, ISO-8859-1, ASCII and UTF-16     |
   +--------+-----------------------------------------------------+
   | -32702 |  (Parsing) - invalid characters, encoding mismatch    |
   +--------+-----------------------------------------------------+

XX still wrong here, TODO: incorporate the almost-agreed-on numbers
XX some pretty silly entries here? a few of those errors are transport
XX (HTTP) errors - a 415 HTTP error would do for malformed XML
```

All other error codes are free to be used to signalise internal method processing errors, while -32500 could be used as generic return code in such cases (not recommended). The human readable error string should contain as much information as possible (up to an method call syntax description).

5.4 Standardized System Methods

The wideley implemented system.*() call package provides interesting stuff... TODO

5.4.1 system.listMethods()

Returns an array of strings, each the name of a callable remote procedure... TODO

5.4.2 system.methodSignature()

Allows to query parameter and return variables number and types for a given method name... TODO

5.4.3 system.multiCall()

Packages multiple calls in an array with specifically designed structs, where each mimics a single XML+RPC method call. The result is an array likewise... TODO

5.4.4 system.dataTypes()

This call is a recommendation to allow for future extensibility. The described procedure call will return an array of strings, which name data types supported by the queried server... TODO

Would return an array with ["boolean", "int", "double", "string", "dateTime.iso8601", "base64", "array", "struct"] for unextented implementations of this specification. Note the absence of the "i4" type, which is deprecated by now.

An implementation could invent new types (like "nil" or "unicode" or even "object") and signalise support by means of this method call. See paragraph 3.7 of this document on how to encode extended data types without breaking intercompatibility.
Note, that this information however can only by queried from the server by a client, so that a server could never assume the client to support an extended data type unless there is a way to negotiate on this.

5.3 Compatibility with XML-RPC

Maintining backwards compatibility with XML-RPC may not be desired, but is a simple task, because there were basically no changes to XML+RPC, except for some deprecations and minor transport level enhancements.

5.3.1 Older Media Type

The MIME media type "text/xml" was falsely used by [MINUS] compliant servers and clients and is likely to quickly disappear with final registration of the more appropriate "application/rpc+xml" MIME type.

Implementations compliant to the rules outlined in this document may however provide and signalise compatibility to unfixed clients and servers by means of handshake requests and Accept: type negotiation.

5.3.2 Use Of A XML-Parser

5.4 Simplification Recommendations -cut-

6. IESG Section

6.1 IANA Media Type Registration

To: ietf-types@iana.org
Subject: Registration of Standard MIME Media type application/rpc+xml

MIME media type name: application
MIME subtype name: rpc+xml
Required parameters: none

Optional parameters: none

Encoding considerations:
   The message format can contain arbitrary binary data, and thus
   MUST be encoded for non-binary transport channels such as SMTP.
   The base64 encoding is suitable for transport protocols other
   than HTTP.

Security considerations:
   See section 7.2 of this document.

Interoperability considerations:
   Registration of this Media Type happens to correct the abuse of
   the "text/xml" MIME Type within XML-RPC transmissions following
   the [MINUS] spec.

Published specification:
   Use of the Media Type is described within this document.

Applications which use this media type:
   ...

Additional information:
   Magic number(s):   none
   File extension(s):   none
   Macintosh File Type Code(s):   none
   Object Identifier(s) or OID(s): none

Person to contact for further information:
   The author of this document.

Intended usage: COMMON

Author/Change controller:
   "Mario Salzer" <mario@erphesfurt.de>

6.2 Security Considerations

Many evil things could happen to everbody who implemented this
protocol... (TODO)

* distinction between read-only and data manipulation calls
* sensitive information (visible transport content)
** ticketing within messages
** auth in lower level protocol
*** http basic auth
*** ssl or tls
* data types and strict typed languages
* tricking scripting languages? (known bugs only)
* compression bombs in deflate or zlib?
* drawbacks of compatibility pressure

* again recommend full xml parsers (charset issues esp)
* ...

7. Acknowledgments

Dave Winer wrote the initial specification and various revisions of the [MINUS] protocol. Adam Megacz wrote the first Internet-Draft to attemp to standardize the XML-RPC protocol, then eventually known as "XMC".

8. Appendix

8.1 a DTD
8.2 SMTP Transport
8.3 Jabber Transport

9. draft TODO list

- mk fully XML compliant
- user-defined types via struct construct (advise against namespaces and incompatible extensions like <null/> tag)
- jabber+smtp bindings in appendix
- [XML] and others are normative refs, aren't they?
- add more SHOULD server requirements for backwards compatibility
- error code registration with the IANA, above 100 user-definable??

Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2234] Crocker, D. (Ed.) and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[RFC3470] Hollenbeck, S., Rose, M. and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, January 2003.

[RFC3023] Murata, M., St. Laurent, S. and D. Kohn, "XML Media Types", RFC 3023, January 2001.

[RFC2048] Freed, N., Klensin, J. and J. Postel, "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures", BCP 13, RFC 2048, November 1996.

[UTF8] Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[XML] Bray, T., Paoli, J., Sperberg-McQueen, C. and E. Maler, "Extensible Markup Language (XML) 1.0 (2nd ed)", W3C REC-xml, October 2000, <http://www.w3.org/TR/REC-xml>.

Informative References

[MINUS] D. Winer, "XML-RPC Specification", June 1999, <http://www.xmlrpc.org/spec>.

[FAULTC] D. Libby and others, "Specification for Fault Code Interoperability", May 2001, http://xmlrpc-epi.sourceforge.net/specs/rfc.fault_codes.php

[ISO8601] "International Standardization Organisations` 8601", ...

[UTC] "Universal Time Code" (AKA "GMT"), ...

Authors' Addresses

Mario Salzer
FH Erfurt, University of Applied Sciences
Fischersand 12, 99084 Erfurt
Europe / Germany / Thuringia

email: mario@erphesfurt.de
phone: +49-361-6433638
icq: 95596825

Trademark Statement

The name "XML-RPC" is a registered trademark of Userland(?). To allow description of a compatible protocol we named this document and the methods and syntax it describes "XML+RPC".

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive

Director.

Full Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.