HXP

Healthcare Xchange Protocol

Draft proposal 0.1

March 14, 2004

Table of Contents

1.0 Introduction 3

2.0 XML Message 4

3.0 RPC – Remote Procedure Call 5

4.0 XML-RPC Specifications 6
- 4.1 Overview
- 4.2 Request Example
- 4.3 Header Requirements
- 4.4 Payload Format
- 4.5 Response Example
- 4.6 Response Format
- 4.7 Fault
- 4.8 Goals and Strategies

5.0 Interface Layers 10

6.0 XML-RPC Drivers 11

7.0 HXP Server 16

8.0 HXP Client 20

9.0 HXP Guidelines 24

10.0HXP-PCD Procedure Call Dictionary 26
- 10.0.1. Person calls
- 10.0.2. Encounter calls
- 10.0.3. Electronic Medical Record (EMR) calls
- 10.0.4. Department calls
- 10.0.5. Ward calls
- 10.0.6. Room calls
- 10.0.7. System calls
- 10.0.8. Anonymous calls

11.0 Look to the future - HXP Abstraction Layer 31
12.0 About the authors 32

1.0 Introduction

HXP is the standard data exchange protocol being used by healthcare applications to communicate transparently with each other regardless of their platforms.

HXP makes data exchange among healthcare applications:

- simple to implement
- easy to understand
- platform independent
- free or very low cost
- free from vendor lock
- reliable
- secure with authentication
- encrypted with SSL protocol
- flexible
- transparent
- free from geographic restrictions
- open sourced
- developed collaboratively
- peer reviewed
- intercontinental real-time transfer possible

HXP consists of XML message format and the Procedure Call Dictionary (PCD). The XML message is based strictly on the open standard specifications of the XML-RPC protocol.

It is a protocol for making and receiving procedure calls over the internet and thus allow reception and transmission of data among remote healthcare applications.

What this means is that different healthcare information systems can use HXP to "ask each other questions" or "talk to each other".

How is it used?

Using HXP is just like making a function call in your program, only the computer that executes the function could be thousands of miles away.

In normal programming situations, an interface library is usually used that handles the actual message formatting. The programmer does not need to know exactly how the message is formatted. With the use of the library, he uses his usual programming conventions to send and receive messages.

2.0 XML Message

On the data transfer level, HXP uses XML as the message format. The xml data on the wire looks like this:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
         </param>
      </params>
   </methodCall>
```

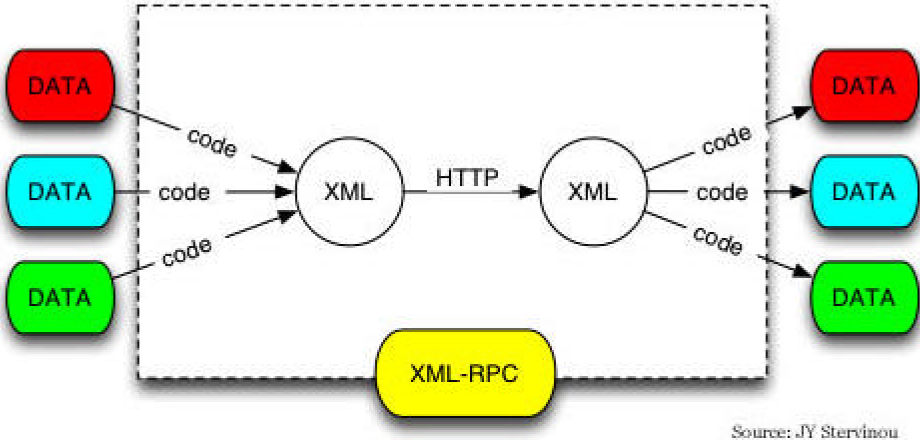
The HXP procedure call “Person” extracting the person registration data with PID “10008900” looks like this on the wire:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>Person</methodName>
   <params>
      <param>
         <value>
            <struct>
              <member>
               <name>usr</name>
                 <value><string>demo_user</string></value>
               </member>
               <member>
                 <name>pw</name>
                  <value><string>demo_pass</string></value>
               </member>
            </struct>
            </value>
         </param>
      <param>
         <value><int>10008900</int></value>
         </param>
      </params>
   </methodCall>
```

3.0 RPC - Remote Procedure Call

On the programming level, HXP uses XML-RPC (Remote Procedure Calling protocol) that works over the Internet, Intranet and local host.

An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML.



Procedure parameters can be scalars, numbers, strings, dates, etc.; and can also be complex record and list structures.

4.0 XML-RPC Specifications

Tue, Jun 15, 1999; by Dave Winer.

Updated 6/30/03 DW

Updated 10/16/99 DW

Updated 1/21/99 DW

This specification documents the XML-RPC protocol implemented in UserLand Frontier 5.1.

For a non-technical explanation, see XML-RPC for Newbies.

This page provides all the information that an implementor needs.

4.1 Overview

XML-RPC is a Remote Procedure Calling protocol that works over the Internet.

An XML-RPC message is an HTTP-POST request. The body of the request is in XML. A procedure executes on the server and the value it returns is also formatted in XML.

Procedure parameters can be scalars, numbers, strings, dates, etc.; and can also be complex record and list structures.

4.2 Request example

Here's an example of an XML-RPC request:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181


<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getStateName</methodName>
   <params>
      <param>
            <value><i4>139</i4></value>
         </param>
      </params>
   </methodCall>
```

4.3 Header requirements

The format of the URI in the first line of the header is not specified. For example, it could be empty, a single slash, if the server is only handling XML-RPC calls. However, if the server is handling a mix of incoming HTTP requests, we allow the URI to help route the request to the code that handles XML-RPC requests. (In the example, the URI is /RPC2, telling the server to route the request to the "RPC2" responder.)

A User-Agent and Host must be specified.

The Content-Type is text/xml.

The Content-Length must be specified and must be correct.

4.4 Payload format

The payload is in XML, a single <methodCall> structure.

The <methodCall> must contain a <methodName> sub-item, a string, containing the name of the method to be called. The string may only contain identifier characters, upper and lower-case A-Z, the numeric characters, 0-9, underscore, dot, colon and slash. It's entirely up to the server to decide how to interpret the characters in a methodName.

For example, the methodName could be the name of a file containing a script that executes on an incoming request. It could be the name of a cell in a database table. Or it could be a path to a file contained within a hierarchy of folders and files.

If the procedure call has parameters, the <methodCall> must contain a <params> sub-item. The <params> sub-item can contain any number of <param>s, each of which has a <value>.

Scalar <value>s

<value>s can be scalars, type is indicated by nesting the value inside one of the tags listed in this table:

Tag	Type	Example
<i4> or <int>	four-byte signed integer	-12
<boolean>	0 (false) or 1 (true)	1
<string>	string	hello world
<double>	double-precision signed floating point number	-12.214
<dateTime.iso8601>	date/time	19980717T14:08:55
<base64>	base64-encoded binary	eW91IGNhbid0IHJlYWQgdGhpcyE=

If no type is indicated, the type is string.

<struct>s

A value can also be of type <struct>.

A <struct> contains <member>s and each <member> contains a <name> and a <value>.

Here's an example of a two-element <struct>:

```
<struct>
   <member>
      <name>lowerBound</name>
      <value><i4>18</i4></value>
      </member>
   <member>
      <name>upperBound</name>
      <value><i4>139</i4></value>
```

```
      </member>
   </struct>
```

<struct>s can be recursive, any <value> may contain a <struct> or any other type, including an <array>, described below.

<array>s

A value can also be of type <array>.

An <array> contains a single <data> element, which can contain any number of <value>s.

Here's an example of a four-element array:

```
<array>
   <data>
      <value><i4>12</i4></value>
      <value><string>Egypt</string></value>
      <value><boolean>0</boolean></value>
      <value><i4>-31</i4></value>
      </data>
   </array>
```

<array> elements do not have names.

You can mix types as the example above illustrates.

<arrays>s can be recursive, any value may contain an <array> or any other type, including a <struct>, described above.

4.5 Response example

Here's an example of a response to an XML-RPC request:

HTTP/1.1 200 OK

```
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
         <value><string>South Dakota</string></value>
         </param>
      </params>
   </methodResponse>
```

4.6 Response format

Unless there's a lower-level error, always return 200 OK.

The Content-Type is text/xml. Content-Length must be present and correct.

The body of the response is a single XML structure, a <methodResponse>, which can contain a single <params> which contains a single <param> which contains a single <value>.

The <methodResponse> could also contain a <fault> which contains a <value> which is a <struct> containing two elements, one named <faultCode>, an <int> and one named <faultString>, a <string>.

A <methodResponse> can not contain both a <fault> and a <params>.

4.7 Fault example

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>faultCode</name>
               <value><int>4</int></value>
               </member>
            <member>
               <name>faultString</name>
               <value><string>Too many parameters.</string></value>
               </member>
            </struct>
         </value>
      </fault>
   </methodResponse>
```

4.7.1 Sample error handling in php

```
if(!$hxpclient->query('Person',$header,10008900)){

      $errcode = $hxpclient->getErrorCode();

      $errmsg = $hxpclient->getErrorMessage();

      echo 'An error occurred - '.$errcode.' '.$errmsg;

}
```

4.8 Strategies/Goals

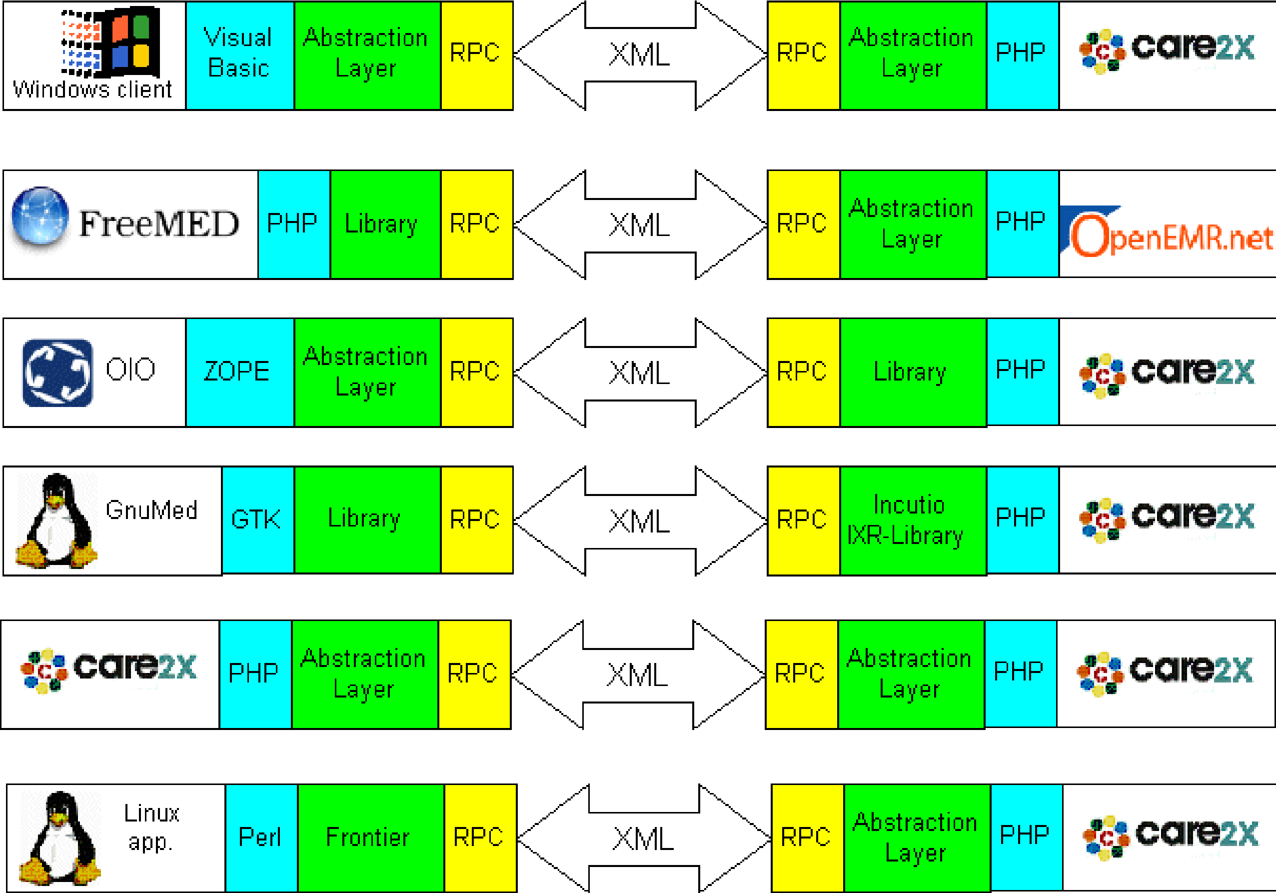
Firewalls. The goal of this protocol is to lay a compatible foundation across different environments, no new power is provided beyond the capabilities of the CGI interface. Firewall software can watch for POSTs whose Content-Type is text/xml.

Discoverability. We wanted a clean, extensible format that's very simple. It should be possible for an HTML coder to be able to look at a file containing an XML-RPC procedure call, understand what it's doing, and be able to modify it and have it work on the first or second try.

Easy to implement. We also wanted it to be an easy to implement protocol that could quickly be adapted to run in other environments or on other operating systems

5.0 Interface layers

To make the whole system very simple to implement, for example it can use an xmlrpc abstraction layer that makes the xml format transparent to the program language of the application (e.g. php).



Other libraries for other platforms are also available and can be easily found on the internet.

6.0 XML-RPC Libraries

There many libraries available for different platforms.

6.1 PHP

6.2 C

6.3 C++

6.4 Python

6.5 Perl (Frontier)

6.6 Java

6.7 Microsoft .NET

6.8 Ruby

6.9 K

6.2 PHP

6.2.1.Source

A PHP library can be downloaded from http://scripts.incutio.com/xmlrpc/

6.2.2. Sample PHP code of an xmlrpc client

```
// Specifying a client by host, path and port
$client = new IXR_Client('scripts.incutio.com', '/xmlrpc/simpleserver.php', 80);

if (!$client->query('test.add', 3, 4)) {
   die('Something went wrong - '.$client->getErrorCode().' : '.$client->getErrorMessage());
}

echo $client->getResponse();
```

6.3 C

6.3.1.Source

A C library can be downloaded from http://xmlrpc-c.sourceforge.net

6.3.2. Sample C code of an xmlrpc client

```
/* A simple synchronous XML-RPC client written in C. */

#include <stdio.h>

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

#define NAME "XML-RPC C Test Client"
#define VERSION "0.1"

void die_if_fault_occurred (xmlrpc_env *env)
{
   if (env->fault_occurred) {
       fprintf(stderr, "XML-RPC Fault: %s (%d)\n",
               env->fault_string, env->fault_code);
       exit(1);
   }
}

int main (int argc, char** argv)
{
   xmlrpc_env env;
   xmlrpc_value *result;
   char *state_name;

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

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

   /* Call the famous server at UserLand. */
   result = xmlrpc_client_call(&env, "http://betty.userland.com/RPC2",
   "examples.getStateName",
   "(i)", (xmlrpc_int32) 41);
   die_if_fault_occurred(&env);

   /* Get our state name and print it out. */
   xmlrpc_parse_value(&env, result, "s", &state_name);
   die_if_fault_occurred(&env);
   printf("%s\n", state_name);

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

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

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

   return 0;
}
```

6.4 C++

6.4.1.Source

A C++ library can be downloaded from http://xmlrpc-c.sourceforge.net

6.4.2. Sample C++ code of an xmlrpc client

```
// List recently-released Linux applications. (Written in C++.)
// For more details about O'Reilly's excellent Meerkat news service, see:
// http://www.oreillynet.com/pub/a/rss/2000/11/14/meerkat_xmlrpc.html */

#include <iostream.h>
#include <strstream.h>

#include <XmlRpcCpp.h>

#define NAME           "XML-RPC C++ Meerkat Query Demo"
#define VERSION        "0.1"
#define MEERKAT_URL    "http://www.oreillynet.com/meerkat/xml-rpc/server.php"
#define SOFTWARE_LINUX (6)

static void list_apps (int hours) {

   // Build our time_period parameter.
   ostrstream time_period_stream;
   time_period_stream << hours << "HOUR";
   string time_period = time_period_stream.str();

   // Assemble our meerkat query recipe.
   XmlRpcValue recipe = XmlRpcValue::makeStruct();
   recipe.structSetValue("category", XmlRpcValue::makeInt(SOFTWARE_LINUX));
   recipe.structSetValue("time_period", XmlRpcValue::makeString(time_period));
   recipe.structSetValue("descriptions", XmlRpcValue::makeInt(76));

   // Build our parameter array.
   XmlRpcValue param_array = XmlRpcValue::makeArray();
   param_array.arrayAppendItem(recipe);

   // Create a client pointing to Meerkat.
   XmlRpcClient meerkat (MEERKAT_URL);

   // Perform the query.
   XmlRpcValue apps = meerkat.call("meerkat.getItems", param_array);

   // Print our results.
   int first = 1;
   size_t app_count = apps.arraySize();
   for (int i = 0; i < app_count; i++) {
XmlRpcValue app = apps.arrayGetItem(i);

// Get some information about our application.
string title       = app.structGetValue("title").getString();
string link        = app.structGetValue("link").getString();
string description = app.structGetValue("description").getString();

// Print a separator line if necessary.
if (first)
    first = 0;
else
    cout << endl;

// Print this application entry.
if (description.size() > 0) {
    cout << title << endl << description << endl << link << endl;
} else {
    cout << title << endl << description << endl << link << endl;
}
   }
}

// Print out a usage message.
static void usage (void)
{
   cerr << "Usage: meekat-app-list [hours]" << endl;
   cerr << "Data from <http://www.oreillynet.com/meerkat/>." << endl;
   exit(1);
}

int main (int argc, char **argv) {
   int status = 0;
   int hours = 25;

   // Parse our command-line arguments.
   if (argc == 1) {
// Use default value for hours.
   } else if (argc == 2) {
hours = atoi(argv[1]);
   }
   if (hours == 0)
usage();
   if (hours > 49) {
       cerr << "It's not nice to ask for > 49 hours at once." << endl;
       exit(1);
```

```
    }

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

    // Call our implementation, and watch out for faults.
    try {
list_apps(hours);
    } catch (XmlRpcFault& fault) {
cerr << argv[0] << ": XML-RPC fault #" << fault.getFaultCode()
     << ": " << fault.getFaultString() << endl;
status = 1;
    }

    // Shut down our client library.
    XmlRpcClient::Terminate();

    return status;
}
```

6.5 Python

6.5.1.Source

The library is integrated in Python starting at version 2.2

6.5.2. Sample Python code of an xmlrpc query

```
from xmlrpclib import Server

betty = Server("http://betty.userland.com")

print betty.examples.getStateName(41)
```

6.6 Perl

6.6.1.Source

The library can be downloaded from http://bitsko.slc.ut.us/~ken/xml-rpc/

6.6.2. Sample Perl code of an xmlrpc query

```
use Frontier::Client;

$server = Frontier::Client->new(url => $url);

$result = $server->call($method, @args);
```

6.7 Java

6.7.1.Source

The library can be downloaded from Apache org here
http://www.apache.org/dyn/closer.cgi/ws/xmlrpc/

6.7.2. Sample Java code of an xmlrpc query

```
// Java example
  XmlRpcClient xmlrpc=new XmlRpcClient
    ("http://xmlrpc.emailservice.com:80/xmlrpcInterface");
  Vector params=new Vector();
  params.addElement("myUserName");
  Integer result=(Integer)xmlrpc.execute
    ("email.getNumberNewMessages", params);
```

6.8 Microsoft .NET

6.8.1.Source

The library and more information can be found at

http://www.xml-rpc.net/faq/xmlrpcnetfaq.html

6.8.2. Sample Microsoft .NET code of a client side proxy class.

```
[XmlRpcUrl("http://www.cookcomputing.com/sumAndDiff.rem")]
class SumAndDiffProxy : XmlRpcClientProtocol
{
  [XmlRpcMethod("samples.sumAndDifference")]
  SumAndDiffValue SumAndDifference(int x, int y)
  {
    return (SumAndDiffValue)Invoke("SumAndDifference",
                                   new Object[]{ x, y });
  }
}
```

6.9 Ruby

6.9.1.Source

The library can be downloaded from here

6.9.2. Sample Ruby code of an xmlrpc query

6.10 K

6.10.1.Source

The library can be downloaded from here

6.10.2. Sample K code of an xmlrpc query

7.0 HXP Server

In order for an application to receive procedure calls coming from a remote application and respond with the needed data, it should have an HXP server created and installed.

7.1 Creating a server

Creating a HXP server is very easy and straightforward by using a ready made library. Below is an explanation on how to create a basic HXP server based on the Incutio IXR library for the php programming language. Other libraries for other languages should have similar methods for creating the server.

7.2 Basic Server Construction

Basic HXP servers based on IXR library are constructed using the IXR_Server class. The methodology is simple - create PHP functions (or class methods if you want to go the OOP route), then call the IXR_Server constructor with an array that "maps" XML-RPC method names to the names of your functions. The IXR_Server class will then call the relevant functions based on information from the client.

Your callback functions can return normal PHP variables (strings, integers, arrays, indexed arrays or even objects) and IXR will convert the return values in to an XML-RPC response and send it back to the client.

Here is a simple HXP server (based on IXR, php):

```
<?php

include('IXR_Library.inc.php');

function PersonNameLast($args) {
    return 'Rodriguez';
}
function PersonNameFirst($args) {
    return 'Jonathan';
}
$server = new IXR_Server(array(
    'Person.Name.Last' => 'PersonNameLast',
    'Person.Name.First' => 'PersonNameFirst'
));


?>
```

The above code will create an XML-RPC server with two available methods which accepts a struct and an integer as arguments- `Person.Name.Last` returns the string 'Rodriguez', and `Person.Name.First` which returns the string 'Jonathan'. Of course, in real implementation, the function's internal routine will access the database, get the name of the patient based on the PID number which was passed as a second argument and return the name.

Note that both of the callback functions accept a single argument - $args. $args is an array of paramaters received from the XML-RPC client. If there is only one argument, $args will be that argument rather than an array with only one element.

7.3 Header arguments

Each function accepts a compulsary heading argument as the first parameter ($args[0]). This parameter is an indexed array (RPC's struct) and must contain by default the following keys and data:

Compulsary keys and data for detection.

Key	Data
usr	Username
pw	Password
emergency	(integer) 1 (If the call is done in "emergency". Can be left unset in normal cases.)

When the header argument is missing, the application will return an error code 1000 and the descriptive string "_ERROR_AUTH_BADHEADER".

By default, the "emergency" key is either not available or set to 0 or empty character.

Optional keys and data:

Key	Data
lang	Language code
sid	session ID
version	HXP version (e.g. 1.0, 1.0.3)

The HXP server should be have the functions to inform an inquiring client of the availability of optional keys by including the following system procedure calls:

```
System.Header.Keys.lang
System.Header.Keys.sid
System.Header.Keys.version
```

If the key is supported by the application, the function should return the value of integer 1.

Other keys and data can be added according to the application's design but these are not considered standard and should be properly documented and published. Once these added keys and data get widespread acceptance or when they were proposed for addition and approved, they will be included as standard header keys and published in the next PCD (Procedure Call Dictionary) version.

7.3 Sending Errors

XML-RPC has a flexible error system, which you can use in your server implementations. An XML-RPC error consists of an error number (an integer) and a descriptive string. The number you use is pretty much up to you as the application developer, but it is important you keep a consistent numbering scheme. You can send errors back to the client using the following code:

```
function ErrorMessage($args) {
    if ($args[0]['pw'] != 'thesecretpassword') {
        return new IXR_Error(1002, '_ERROR_AUTH_USER');
    }
    // Rest of function here
}
```

The above function will return an error if the first argument sent by the client is anything other than 'thesecretpassword' - it can be used to form a very basic (and not particularly secure) authentication system.

7.4 HXP error codes

Error codes of HXP start at 1000. The following codes are draft proposals version 0.1.

Error code	Error string	Meaning
1000	_ERROR_AUTH_BADHEADER	Bad or missing header argument/parameter
1001	_ERROR_AUTH_USER	Wrong user.
1002	_ERROR_AUTH_PW	Wrong password.
1003	_ERROR_AUTH_AREA	No permission for the protected area.
1004	_ERROR_AUTH_LOCKED	The user permission is locked.
1005	_ERROR_SQL_RESULT	The sql query failed.
1006	_ERROR_NORESULT	The procedure returned no result.
1007	_ERROR_PROC_NOSUPPORT	The procedure is not supported
1008	_ERROR_PROC_NOEXISTS	The procedure is not available

7.5 Dates and base64 encoded data

XML-RPC has support for two types that do not have a direct corresponding type in PHP - dates and base64 encoded data. To return data in these formats, you can use the following:

```
function myFunction($args) {
    $data = 'a string to be transmitted as base64 encoded data';
    $time = time();
    return array(
        'data' => new IXR_Base64($data),
        'time' => new IXR_Date($time)
    );
}
```

The above function returns an indexed array (an XML-RPC struct) consisting of a base64 encoded data chunk and an iso encoded date. As you can see, the details of the encoding are kept hidden from the web service programmer.

7.6 The Object Oriented Approach

Creating a HXP server with callback functions is all well and good, but IXR can also be used to create a HXP server as PHP classes. This can be achieved by creating a server class that extends IXR_Server and defining methods that you wish to use as XML-RPC procedures in the new class. You can then call the IXR_Server constructor with the now familiar callback array, with one small difference - each of your callback functions should be prefixed with "this:", to inform IXR that you wish to call a class method rather than a normal function.

Here is the above simple HXP server implemented using OOP:

```
<?php

include('IXR_Library.inc.php');

class HXP_Server extends IXR_Server {
   function SimpleServer {
      $this->IXR_Server(array(
         'Person.Name.Last' => 'this:PersonNameLast',
         'Person.Name.First' => 'this:PersonNameFirst'
      ));
   }
      function PersonNameLast($args) {
            return 'Rodriguez';
      }
      function PersonNameFirst($args) {
      return 'Jonathan';
      }
}

$server = new HXP_Server();

?>
```

7.7 Server construction with other languages

Please refer to the documentation of the library that you wish to use.

8.0 HXP Client

In order for an application to call procedures in a remote application and receive the requested data, it should have an HXP client created and installed.

Creating a client
Creating a HXP client is very easy and straightforward by using a ready made library. Below is an explanation on how to create a basic HXP client based on the Incutio IXR library for the php programming language. Other libraries for other languages should have similar methods for creating the client.

8.1 Basic Client Construction

HXP client based on IXR library allows you to make requests to HXP servers using the IXR_Client class. Before you make a connection you will need to know the host, path and port of the HXP server you wish to communicate with, and know which method on the server you wish to access. The HXP methods are listed in the Procedure Call Dictionary (HXP-PCD) which is published for public use.

First you will need to create an IXR_Client object. You must supply the details of the server you wish to connect to when you create the object, either as a URL or as a hostname, path and port:

```
// Specifying a HXP server by URL (port 80 is assumed):
$client = new IXR_Client
('http://www.care2x.net/modules/hxp/server.php');

// Specifying a HXP server by host, path and port
$client = new IXR_Client('www.care2x.net', '/modules/hxp/server.php',
80);
```

The next step is to make your HXP call using the `query()` method. The method accepts a HXP procedure name with a variable number of arguments, which will be passed to the HXP server along with the request. The query() method returns true or false depending on whether or not your request has been successful. Should your request fail the `getErrorCode` and `getErrorMessage` methods will usually provide a more detailed explanation of what went wrong.

```
// Create the header and supply the username and password
$header['usr']='admin';
$header['pw']='admin';

// Create the PID of the person for the second argument
$PID=10003002;

if (!$client->query('Person.Basic', $header, $PID)) {
    die('Something went wrong - '.$client->getErrorCode().' :
'.$client->getErrorMessage());
}
```

Presuming the method was successful, you can access the data returned by the HXP server using the getResponse() method. This will return a single value or an array of values, depending on the data structure returned by the server.

```
echo $client->getResponse();
```

So far, so good. Now let's try for something a bit more complicated. The care2x.net has set up an example HXP server that interfaces with an HIS, an integrated hospital information system. Let's see what it knows about the person with a PID (Person Identifier) of 10000000.

We want to see only some of the basic data of the person and after scanning the Procedure Call Dictionary we found the Person.Basic to be the proper procedure name for the purpose.

```
// Create the client object
$client = new IXR_Client
('http://www.care2x.net/foundry/modules/hxp/server.php');

// Create the header and supply the username and password
$header['usr']='admin';
$header['pw']='admin';

// Create the PID of the person for the second argument
$PID=10000000;

// Run a query
if (!$client->query('Person.Basic', $header, $PID)) {
    die('Something went wrong - '.$client->getErrorCode().' :
'.$client->getErrorMessage());
}else{

    // Get the result
    $response = $client->getResponse();

    // Display the result
    echo '<TABLE BORDER=0>';
    while(list($x,$v) = each($response)){
        echo '<TR><TD>'.$x.'</TD><TD>'.$v.'</TD></TR>';
    }
    echo '</TABLE>';
}
```

You should see something that looks like this:

pid	10000000
title	Lord
name_first	Coon
name_last	Ty
name_2	William
name_3	
name_middle	Heinrich
name_maiden	
name_others	James Gone
date_birth	2003-12-31
sex	m
addr_str	New Hamshire
addr_str_nr	54
addr_zip	5456
addr_citytown_nr	1
photo_filename	

So far, everything looks fine. We were able to get the basic data of the person using our previous knowledge of his PID number. Now, let's say we do not know any PID numbers at all. What should we do? Well, we can try searching for persons. After consulting the Procedure Call Dictionary (PCD), we found the Person.Search procedure name. We will also use the "%" wildcard as the second argument to enable us to see all persons available in the database.

```
// Create the client object
$client = new IXR_Client
('http://www.care2x.net/modules/hxp/server.php');

// Create the header and supply the username and password
$header['usr']='admin';
$header['pw']='admin';

// Use the "%" wildcard as the search keyword
$searchkey='%';

// Run a query for PHP
if (!$client->query('Person.Search', $header, $searchkey)) {
    die('Something went wrong - '.$client->getErrorCode().' :
'.$client->getErrorMessage());
}

// Get the result
$response = $client->getResponse();

// Display the result
echo '<TABLE BORDER=0>';

echo "<TR>
        <TD> PID </TD>
        <TD> Family name </TD>
        <TD> First name </TD>
        <TD> Date of birth </TD>
        <TD> Sex </TD>
        </TR>";

while(list($x,$v) = each($response)){
        echo "<TR>";
        while(list($y, $z) = each($v)){
                echo '<TD>'.$v.'</TD>';
        }
        echo "</TR>";
}

echo '</TABLE>';
```

Now, you should see something that looks like this:

PID	Family name	First name	Date of birth	Sex
10000002	becker	walter	1966-04-05	m
10000004	Caballero	pepe	1978-10-12	m
10000003	Khan	Zamir	1972-03-02	m
10000001	test	test	2002-12-01	m
10000000	Ty	Coon	2004-01-12	m

8.2 Header arguments

Each procedure call requires a compulsary heading argument as the first parameter (shown as the `$header` variable on the above examples). This parameter is an indexed array (RPC's struct) and must contain by default the following keys and data:

Compulsary keys and data.

Key	Data
usr	Username
pw	Password
emergency	(integer) 1 (If the call is done in "emergency". Can be left unset in normal cases.)

When the header argument is missing, the query will receive an error code 1000 and the error message "_ERROR_AUTH_BADHEADER". By default, the "emergency" key can be either left unset or set to the 0 value or empty character.

Optional keys and data:

Key	Data
lang	Language code
sid	session ID
version	HXP version (e.g. 1.0, 1.0.3)

The availability of the optional keys can be detected either on the fly or during configuration by using the following system procedure calls:

```
System.Header.Keys.lang

System.Header.Keys.sid
System.Header.Keys.version
```

If the key is supported by the remote application, the query should receive the value

of integer 1.

Other keys and data may have been added to the remote application according to the its design but these are not considered standard and should be properly documented and published. It is always important to consult the documentation of the remote HXP server. To be on the safe side, it is good to use only the published standard keys.

8.3 Debugging with IXR library

Should you need to debug your client procedure calls at any time you can do so with the debug flag:

```
$client = new IXR_Client
('http://www.care2x.net/modules/hxp/server.php');
$client->debug = true;

// Create the header and supply the username and password
$header['usr']='admin';
$header['pw']='admin';

if (!$client->query('Person.Basic', $header, 10000000)) {
    die('Something went wrong - '.$client->getErrorCode().' :
'.$client->getErrorMessage());
}
```

With the debug flag turned on, IXR_Client will display the full contents of all outgoing and incoming HXP messages.

9.0 HXP Guidelines

General philosophy

`"Impossible" is not a fact, just an opinion.`

9.1 Basic Design Guidelines

- "Keep it simple, stupid". Do not try to solve all the world's problems.
- This is a simple problem. Avoid making it complex.
- Avoid complex namespaces.
- The procedure names should be self-explanatory.
- Define general error classes only. Specifics maybe filled later on as descriptive error message.
- Allow for flexibility.
- Allow for expandability.

9.2 Data security guidelines

- Use a secure data transport medium or protocol e.g. SSL.
- The HXP server must always check the authentication data supplied by the remote client for every procedure call.
- The HXP server must determine which remote client is allowed to call a certain procedure.
- Every abnormal access attempt must be logged in detail.

9.3 Data privacy guidelines

- The person is the sole owner of his data.
- The person must provide a legal instrument that permits the organization and its server to give out his data.
- The person must determine through a legal instrument which outside agency is allowed to access his data.
- The HXP server must be able to determine which remote client is allowed to access which data.
- Data access in "emergency" cases is still a privilege and privileged remote agencies must be clearly defined.
- Request for "emergency" data access must be verified first before data is released.
- Anonymized data does NOT mean "free-for-all" data.
- The HXP server must determine which remote client is allowed to access anonymized data.

9.4 Conformity Guidelines

Server

- HXP servers must support all published standard HXP procedure calls.
- HXP servers must pass a conformity test.
- Strictly adhere to the data type of the parameters and return of every method.
- The server should be placed under the "hxp" subdirectory.
- The server script should be named "server".

Examples:

---/hxp/server.php ---/hxp/server.cgi ---/hxp/server.pl

---/hxp/server.asp ---/hxp/server.p

9.5 Advanced server conformity guidelines

To benefit from a better performance in return for some additional programming complexity, the advanced server conformity guidelines below can be followed.

- Procedures that belong to a main group like e.g. "Person" should be created as a separate server.
- The server script should be named after the procedures' main group name like e.g. "Person" but in lowercase.

Examples 1: The main group is "Person" like in the procedures Person.Basic, Person.List, Person.Search .

```
---/hxp/person.php
---/hxp/person.cgi
---/hxp/person.pl
---/hxp/person.asp
---/hxp/person.py
```

Examples 2: The main group is "EMR" like in the procedures EMR.History, EMR.Test.Result, EMR.Problem.Care .

```
---/hxp/emr.php
---/hxp/emr.cgi
---/hxp/emr.pl
---/hxp/emr.asp
---/hxp/emr.py
```

Most important! The default server e.g. "server.php" containing all the procedures must always be available.

- The default server must have a system procedure that returns the info whether the system supports the advanced server configuration or not. This procedure is called System.Server.Advanced.Exists and must be callable at the main server (e.g. "server.php").

```
System.Server.Advanced.Exists
```

9.6 Introspection

Introspection will be discussed in the later phases of the first draft's development.

9.7 Client

- Use only the published standard procedure names if communicating with a foreign application.
- Strictly adhere to the data type of the parameters and return of every method.

9.8 Backward Compatibility Guidelines

- A newer HXP-PCD should not discard procedure names from older versions.
- An HXP server should always support all published standard procedure at the time of the server's creation.
- A newer PCD version is not allowed to remove, discard, revise nor modify procedures from older versions.

9.11 Reserved error codes

The integer range from 1000 to 2000 is reserved for the error code reporting of HXP.

10.0 HXP Procedure Call Dictionary

The Procedure Call Dictionary (PC) is the detailed documentation of all the procedure calls. It contains the:

1) description of the procedure call
2) purpose of the procedure call
3) parameter description, type, and values needed
4) response data type , keys and values
5) description of possible error codes and their meaning
6) information on extendability

10.1 PCD versions

Each PCD is released as a version. Each new version should be reverse compatible to earlier versions by supporting all previous procedure calls.

The initial procedure calls are currently authored by Elpidio Latorilla. These procedure calls are still in draft status and need to be extended. Furthermore, these procedure calls need to go through a thorough peer review.

10.2 Adding procedure calls

Each new procedure call should be proposed as candidate for addition. When it is approved, it will be included in the next version release.

Person

Person Person.Basic Person.List Person.Encounter Person.Encounter.List Person.Encounter.Current.ENR Person.Encounter.Search Person.Search Person.Name.First Person.Name.Family Person.Name.Middle Person.Name.Maiden Person.Name.Second Person.Name.Third Person.Name.Others Person.Birth.Date Person.Birth.Details Person.Died.Date Person.Died.Cause Person.Died.Encounter.ENR Person.Registry.Date Person.Appointment.List Person.Appointment Person.Appointment.Add Person.Address Person.Address.FullText Person.Address.Street Person.Address.Street.Nr	Person.Address.CityTown Person.Address.Zip Person.Address.Email Person.Phone.1 Person.Died.Cause.Code Person.Phone.1.Area.Code Person.Phone.2 Person.Phone.2.Area.Code Person.Phone.Cell.1 Person.Phone.Cell.2 Person.Fax.Nr Person.Blood.Group Person.Mother.PID Person.Father.PID Person.Insurance.1.Nr Person.Insurance.1.Co.ID Person.Insurance.2.Nr Person.Insurance.2.Co.ID Person.Contact Person.Contact.PID Person.Contact.Relation Person.Add Person.Update Person.Record.Hide Person.Record.Lock Person.Record.Delete

Encounter

Encounter.List Encounter.Person Encounter.Outpatient.List Encounter.Inpatient.List Encounter.Exists Encounter.Search Encounter.Status Encounter.PID Encounter.Admission.Date Encounter.Admission.Type Encounter.Discharge.Time Encounter.Discharge.Is Encounter.Discharge.Type Encounter.Payment.Type Encounter.Dept.Current.Nr Encounter.Firm.Current.Nr Encounter.Doctor.Attending Encounter.Doctor.Consulting Encounter.Referrer Encounter.Referrer.Diagnosis	Encounter.Referrer.Therapy.Recommend Encounter.Referrer.Notes Encounter.Referrer.Dept Encounter.Referrer.Institution Encounter.Dept.Is.In Encounter.Ward.Current.Nr Encounter.Ward.Is.In Encounter.Followup.Date Encounter.Followup.Responsible Encounter.Post.Notes Encounter.Service.Extra Encounter.Record.Status Encounter.Record.History Encounter.Record.Hide Encounter.Record.Lock Encounter.Record.Delete Encounter.Record.Normal Encounter.Pregnancy

Electronic Medical Record

EMR.History.List EMR.History EMR.History.Add EMR.Prescription.List EMR.Prescription EMR.Prescription.Add EMR.Report.Care EMR.Report.Care.Add EMR.Report.Development EMR.Report.Development.Add EMR.Effectivity.Care EMR.Effectivity.Care.Add EMR.Incident.List EMR.Incident EMR.Incident.Add EMR.Immunization.List EMR.Immunization EMR.Immunization.Add EMR.Diagnosis.Text EMR.Diagnosis.Text.List EMR.Diagnosis.Text.Add EMR.Diagnosis.Code EMR.Diagnosis.Code.List EMR.Diagnosis.Code.Add EMR.Discharge.Summary EMR.Discharge.Summary.Add EMR.Vitals.Weight.List EMR.Vitals.Weight.Add EMR.Vitals.Head.Circumference.List EMR.Vitals.Head.Circumference.Add EMR.Vitals.Head.Circumference.Update EMR.Vitals.Height.List EMR.Vitals.Height.Add EMR.Vitals.Height.Update EMR.Vitals.Temperature.List EMR.Vitals.Temperature.Add EMR.Vitals.Temperature.Update EMR.Vitals.Bloodpressure.List EMR.Vitals.Bloodpressure.Add EMR.Vitals.Bloodpressure.Update EMR.Vitals.Input.List EMR.Vitals.Input.Add	EMR.Vitals.Output.List EMR.Vitals.Output.Add EMR.Report.Daily.List EMR.Report.Daily EMR.Report.Daily.Add EMR.Report.Daily.IV.List EMR.Report.Daily.IV.Add EMR.Report.Daily.Anticoagulant.List EMR.Report.Daily.Anticoagulant.Add EMR.Report.Development EMR.Report.Development.Add EMR.Report.Daily.PT.List EMR.Report.Daily.PT.Add EMR.Test.Result.List EMR.Test.Result EMR.Test.Request.List EMR.Test.Request.Add EMR.Therapy.Text.List EMR.Therapy.Text EMR.Therapy.Text.Add EMR.Birth.Details EMR.Orders.Doctor.List EMR.Orders.Doctor.Add EMR.Operation.Notes EMR.Operation.Notes.Add EMR.Problem.Care EMR.Problem.Care.Add EMR.Allergy.List EMR.Allergy.Add EMR.Inquiry.ToDoctor EMR.Inquiry.ToDoctor.Add EMR.Inquiry.ToNurse EMR.Inquiry.ToNurse.Add EMR.Diet.Daily EMR.Diet.Daily.Add EMR.Image.Dicom.List EMR.Image.Dicom EMR.Image.List EMR.Image.URL EMR.Image.Add

Department

Department
Department.Name
Department.ID
Department.Description
Department.Logo
Department.Logo.Filename
Department.List

Ward

Ward
Ward.Name
Ward.ID
Ward.Description
Ward.List
Ward.Room.List

Room

Room.Nr
Room.Bed.List
Room.Description

System

System.Error.Keys.error_msg
System.Error.Keys.error_nr
System.Error.Keys.List
System.Header.Keys.lang
System.Header.Keys.sid
System.Header.Keys.version
System.Header.Keys.List

System.HXP.PCD.Version
System.HXP.PCD.Version.List
System.HXP.PCD.Anonymized
System.HXP.Type.List
System.HXP.Type.Set

System.Server.Advanced.Exists

Anonymized data calls

Anonymized data calls receive anonymized data. Anonymized data are medical data which are true and accurate but cannot be directly linked to a patient nor his personal data. Most of the anonymized data are extracted for purposes of research and statistics. The data are strictly selected to ensure the anonymity of the patient.

Anonymized data calls are strictly "READ ONLY" calls.

All anonymized data calls begin with "Anon".

Anon.PID.List Anon.PID.Search Anon.PID.Search.ByDOB Anon.PID.Search.ByAddress Anon.PID.Dead.List Anon.PID.Search.ByAge Anon.PID.Male.List Anon.PID.Female.List Anon.Demographic Anon.Encounter.ENR.List Anon.Encounter.ENR.Search.ByDiagnosis Anon.Encounter.ENR.Search.ByTherapy Anon.Encounter.ENR.Search.ByPrescription Anon.Encounter.ENR.Search.ByIncident Anon.Outpatient.ENR.List Anon.Inpatient.ENR.List Anon.Encounter Anon.EMR.History.List Anon.EMR.History Anon.EMR.Prescription.List Anon.EMR.Prescription Anon.EMR.Report.Care Anon.EMR.Report.Development Anon.EMR.Effectivity.Care Anon.EMR.Incident.List Anon.EMR.Incident Anon.EMR.Immunization.List Anon.EMR.Immunization Anon.EMR.Diagnosis.Text Anon.EMR.Diagnosis.Text.List Anon.EMR.Diagnosis.Code	Anon.EMR.Diagnosis.Code.List Anon.EMR.Discharge.Summary Anon.EMR.Vitals.Weight.List Anon.EMR.Vitals.Head.Circumference.List Anon.EMR.Vitals.Height.List Anon.EMR.Vitals.Temperature.List Anon.EMR.Vitals.Bloodpressure.List Anon.EMR.Vitals.Input.List Anon.EMR.Vitals.Output.List Anon.EMR.Report.Daily.List Anon.EMR.Report.Daily Anon.EMR.Report.Daily.IV.List Anon.EMR.Report.Daily.Anticoagulant.List Anon.EMR.Report.Development Anon.EMR.Report.Daily.PT.List Anon.EMR.Test.Result.List Anon.EMR.Test.Result Anon.EMR.Test.Request.List Anon.EMR.Therapy.Text.List Anon.EMR.Therapy.Text Anon.EMR.Birth.Details Anon.EMR.Orders.Doctor.List Anon.EMR.Operation.Notes Anon.EMR.Problem.Care Anon.EMR.Allergy.List Anon.EMR.Inquiry.ToDoctor Anon.EMR.Inquiry.ToNurse Anon.EMR.Diet.Daily

10.3 HXP Error Codes

Error codes of HXP start at 1000. The following codes are draft proposals version 0.1.

Error code	Error string	Meaning
1000	_ERROR_AUTH_BADHEADER	Bad or missing header argument/parameter
1001	_ERROR_AUTH_USER	Wrong user.
1002	_ERROR_AUTH_PW	Wrong password.
1003	_ERROR_AUTH_AREA	No permission for the protected area.
1004	_ERROR_AUTH_LOCKED	The user permission is locked.
1005	_ERROR_SQL_RESULT	The sql query failed.
1006	_ERROR_NORESULT	The procedure returned no result.
1007	_ERROR_PROC_NOSUPPORT	The procedure is not supported
1008	_ERROR_PROC_NOEXISTS	The procedure is not available

10.4 Error handling

It is important that the client create a common routine to handle possible error messages being returned by the responding server.

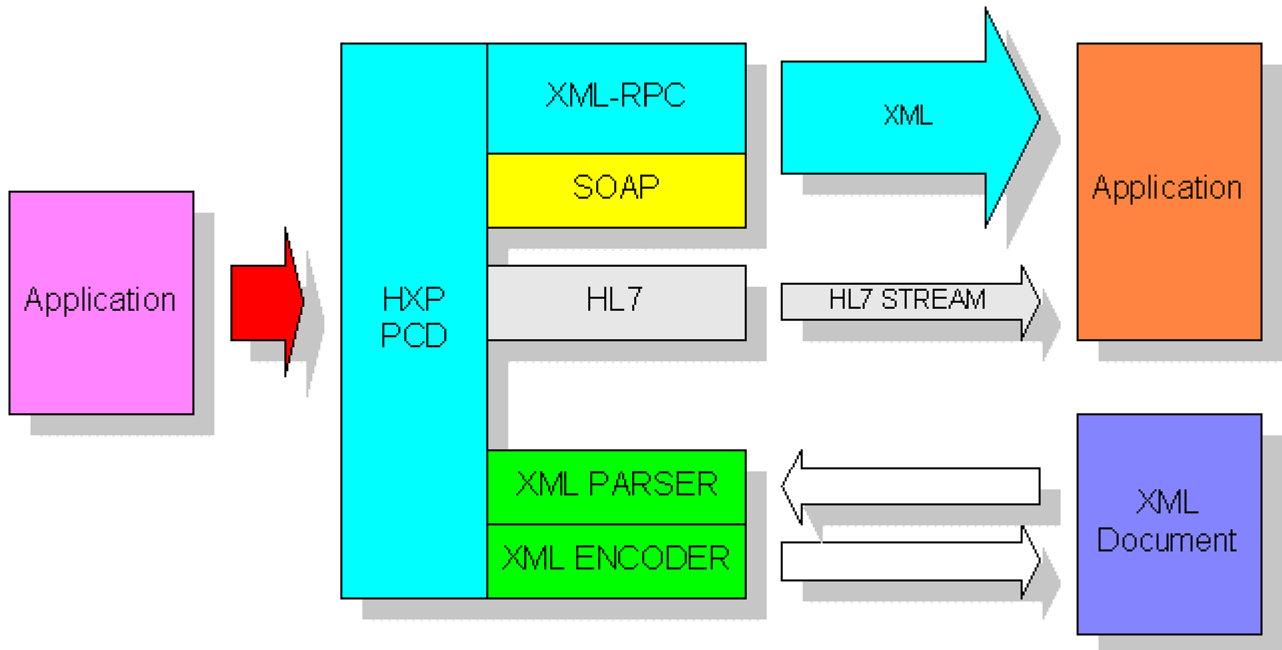
10.5 TRUE and FALSE

To avoid cross platform conflicts regarding boolean values, HXP calls will return:

Value	Meaning
(integer) 1	TRUE
(integer) 0	FALSE
Empty character	undefined

11.0 HXP Abstraction Layer with other "protocols"

This is the vision of the future enhancement of the HXP protocol.



11.1 Future additions to the HXP Procedure Call Dictionary:

HL7 Calls

System.HXP.PCD.HL7

XML Document calls

XML Document calls are used to process xml coded data from or to a document (not the application). All procedure calls have a parameter containing the filename of the xml coded document.

Document.XML.Decode
Document.XML.Encode

Plain Document Calls

Document.Text.Read
Document.Text.Save

12.0 About the Authors:

An author is a person who has contributed any of the following to the development of the HXP protocol:

- A procedure name and/or specification
- Documentation of the protocol

List is alphabetically ordered by first name:

Elpidio Latorilla

Created the first draft proposals for the protocol and the PCD (Procedure Call Dictionary).

Professions:
Programmer, OR Nurse, Electronics & telecommunications technician

He is also the founder and main developer of the Care2x Integrated Healthcare Environment project which is also using this HXP protocol.

Elpidio Latorilla can be contacted here.

Additional authors will be listed soon.