XML-RPC Client library

Designed for Symbian platform

Adrian Dydecki
Version: 1.0 January 2010

TABLE OF CONTENTS:

1 Introduction

This paper describes how to make procedure calls over the Internet using XmlRpcClient library on Symbian OS.

Information included in this document refers to XML-RPC protocol. XML-RPC is lightweight protocol for a remote procedure calls. It uses XML to encode its calls and HTTP as a transport mechanism. More information about XML-RPC can be found in document “XML-RPC protocol. Requesting remote services”.

1.1 Introduction to XmlRpcClient.dll

XmlRpcClient.dll is C++ implementation of the XML-RPC protocol for the Symbian platform. It implements client functionality and provides an easy to use API.

The user of this library will use it to build an in-memory representation of a XML-RPC request, and serialize (encode) that request into XML. Then send the encoded request to the server via XmlRpcClient’s API. The server will de-serialize the request, call the appropriate registered method and generate a response. The response will be serialized into XML and sent back to the client. The client will de-serialize it into memory, and inform the user about the results via API.

1.2 Features

Main features of the library are:

- an XML-RPC client for accessing XML-RPC services,
- it provides an easy to use API for Symbian developers,
- ideal for small devices like smart phones,
- supports serializing of application's native C and C++ data structures,
- string pools used to reduce string comparison,
- SAX parser used for parsing XML.

2 XmlRpcClient.dll implementation

The library consists of the following main files:

File name	Description
xmlrpcclient.h	This file contains definition of a class CXmlRpcClient that represents a connection to an XML-RPC server.
mxmlrpcclientobserver.h	Callback interface to implement in application to be reported by CXmlRpcClient object about response, status, fault etc.
xmlrpchandler.h	This file contains definition of a class CXmlRpcHanlder that parse XML-RPC messages returned from server and interface MXmlRpcHandlerObserver which CXmlRpcClient is implementing to be notified when parse is finished.
xmlrpcrequest.h	This file contains definition of a class that represents XML-RPC request.
xmlrpcresponse.h	This file contains definition of a class that represents XML-RPC response.
xmlrpcvalue.h	This file contains definition of a class that represents XML-RPC method arguments and results.
xmlrpcutils.h	This file contains definition of a utility class for converting between Unicode strings and UTF8+xml entities.
xmlrpcclientstringtable.st	This file contains a static string table specific to the XML-RPC protocol being used by this library.

Table 1. List of files included in library

Two files not included in the above table, but included in the library, *xmlrpcclientstringtable.h* and *xmlrpcclientstringtable.cpp* are auto generate from *xmlrpcclientstringtable.st* by the Symbian tool called *stringtable.pl*. A string pool is a Symbian mechanism for storing strings in such a way that makes the comparison of strings a very fast operation.

2.1 Class structure

All classes are found on `XmlRpc` namespace, with `CXmlRpcClient` being a main class. Figure 1 shows relationships about all the classes.

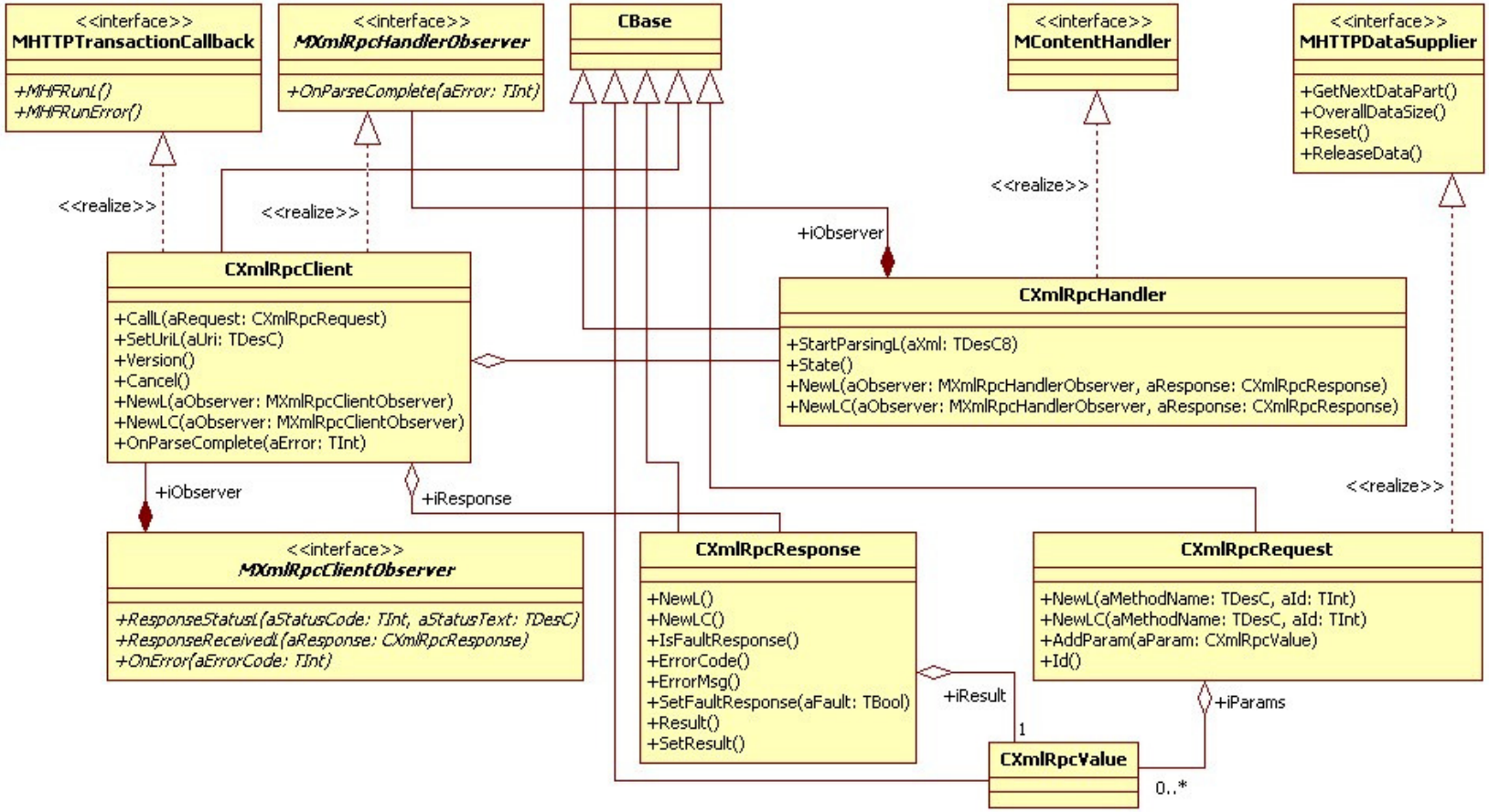

Figure 1. Simplified UML class diagram

`CXmlRpcClient` represents a connection to the server. This class is constructed with one argument, a pointer to object of class that implements interface `MXmlRpcClientObserver`. `CXmlRpcClient` reports responses from the Server, faults, and statuses about connection through the supplied `MXmlRpcClientObserver`.

2.2 HTTP connections

This library uses the Symbian HTTP client API that offers direct support for HTTP. This API enables applications to communicate with HTTP servers on the Internet. Use of the HTTP client API is encapsulated in `CXmlRpcClient` class. HTTP is required for exchanging data with the XML-RPC servers.

As the Symbian HTTP client API asynchronously sets up a connection and prepares data to send, calling of method `CXmlRpcClient::CallL(CXmlRpcRequest& aRequest)` is also asynchronous. This has one good reason – it will not block UI. Figure 2 shows sequence diagram how library is making a call to the XML-RPC server and informing user application about response, status or faults. Below steps describes sequence diagram.

1: CXmlRpcCient::NewL	Creates new CXmlRpcClient object
2: CXmlRpcCient::SetUriL	Sets the address of the XML-RPC host
3: CXmlRpcRequest::NewL	Creates new XML-RPC request
4: CXmlRpcClient::CallL	Asynchronous call. At this point XmlRpcClient library at the separate thread is doing all the things needed to connect to remote server, send request and parse response. User application is not blocked. When message comes from the server or an error was occurred during a call, user application will be informed via appropriate method of interface MXmlRpcClientObserver.
5: 6: 7:	opens an HttpConnection, sends an XML-RPC request to the server. Read HTTP headers from the server.
8: ResponseStatusL	HTTP header was received from the server. Informing the user about status. If all okay, status code 200 OK should be returned.

9:
10: } Reading HTTP body data from the Server and parsing it.
11:

12: ResponseReceivedL Inform the user application about response from the Server.

13:
14:
15: } Closing application and deleting CXmlRpcClient.
16:

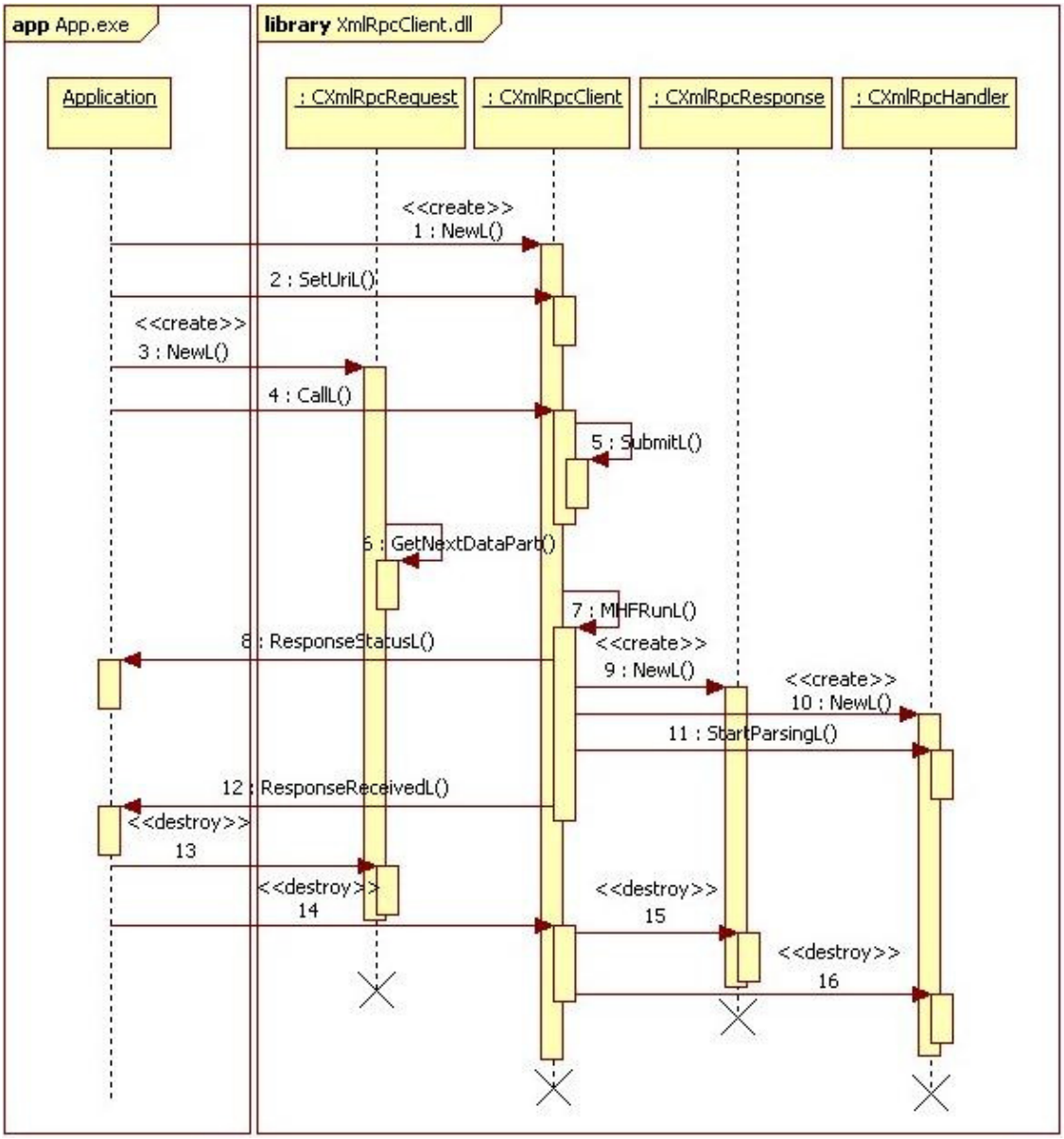

Figure 2. Sequence diagram of requesting remote services.

If new call to the Server is needed, user application starts from step 4 or step 3 if another request then previous will be send.

2.3 Representation of the XML-RPC request

XML-RPC request is represented by object of class `CXmlRpcRequest`. This object is constructed by the user and has all the information about remote method. As the `CXmlRpcClient::CallL(CXmlRpcRequest&)` is asynchronous call, the user application in some way needs to know to which request server responded. This is done by providing request ID. Request ID is stored only in client side (this is not send to server) and should be unique along the application. When application receive response from the Server, checks for what request this response is by calling `RequestId()` method on object of class `CXmlRpcResponse`.

2.4 Representation of the XML-RPC response

Responses from the server are in XML and need to be parsed. A `CParser` a part of Symbian XML framework is used for parsing XML. A SAX parser in this way is more appropriate as it doesn't require whole document to be loaded into memory. Parsing is done in class `CXmlRpcHandler`. This class sets up object of class `CXmlRpcResponse` which is created by object of class `CXmlRpcClient`. `CXmlRpcResponse` contain all the information returned from the Server.

Object of class `CXmlRpcResponse` is create when the data starts flowing from the Server back to the client. Figure 2 shows this on sequence 9. Once this object was constructed it is destroyed only when the user destroy object of the main class `CXmlRpcClient`. When new data starts flowing from the Server this object is sets with new values (it is not constructing anymore).

According to XML-RPC specification XML-RPC response can contain only one XML-RPC value (object of class `CXmlRpcValue`). User can obtain pointer to `CXmlRpcValue` by calling method `CXmlRpcResponse::Result()`.

3 Developing XML-RPC Client Applications

When developing GUI application on Symbian a typical structure of application looks like below [1]:

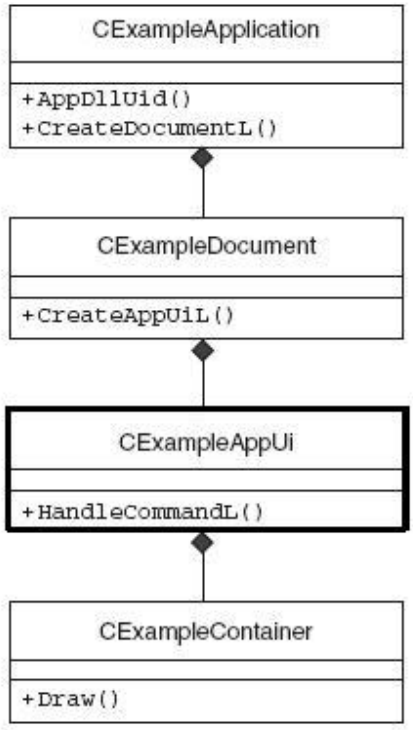

Figure 3. GUI application structure in Symbian

Figure 3 shows a minimum number of classes that need to be created to build GUI application on Symbian. The `CXmlRpcClient` can be placed as a member of class `CExampleAppUi`. The following example shows simple definition of this class:

```
class CExampleAppUi : public CAknAppUi,
                      public XmlRpc::MXmlRpcClientObserver
      {
public:
      // constructor and destructor
      CExampleAppUi();
      virtual ~CExampleAppUi();
      void ConstructL();
```

```
public: // From CCoeAppUi
      void HandleCommandL(TInt aCommand);

public: // From XmlRpc::MXmlRpcClientObserver
      void ResponseStatusL(TInt aStatusCode, const TDesC& aStatusText);
      void ResponseReceivedL(XmlRpc::CXmlRpcResponse& aResponse);
      void OnError(TInt aErrorCode);

public:
      /**
       * Call remote methods on the Server.
       */
      void CallL(TInt aRequestId);
      /**
       * Cancel an outstanding transaction.
       */
      void CancelCall();

private:
      void Distance(XmlRpc::CXmlRpcResponse& aResponse);
      void LatLngPosition(XmlRpc::CXmlRpcResponse& aResponse);

private: // AppUi owns the XmlRpcClient engine
      XmlRpc::CXmlRpcClient* iXmlRpcClient;
      XmlRpc::CXmlRpcRequest* iXmlRpcRequest;
      enum TRequestIds
          {
          ECalculateDistanceRequestId,
          EGetLatLngPositionRequestId
          };
```

Class `CExampleAppUi` implements also interface `MXmlRpcClientObserver` so constructing of `CXmlRpcClient` can be done as follows:

```
void CExampleAppUi::ConstructL()
    {
    iXmlRpcClient = CXmlRpcClient::NewL(*this);

    // server address and port number
    _LIT(KXmlRpcServerHost, "http://localhost:8080");
    iXmlRpcClient->SetUriL(KXmlRpcServerHost);
    }
```

The final implementation of this library is a DLL so user has to change mmp file to link to this library.

```
/*...*/
LIBRARY  XmlRpcClient.lib
/*...*/
```

3.1 Sending request to the server

Sending request to the server is done in method `CallL` of class `CExampleAppUi`. The following example demonstrates how to create request and call remote server:

```
void CExampleAppUi::CallL(TInt aRequestId)
    {
    delete iXmlRpcRequest;
    iXmlRpcRequest = NULL;

    switch (aRequestId)
        {
        case ECalculateDistanceRequestId
            iXmlRpcRequest = CXmlRpcRequest::NewL(
                _L("server.calculateDistance"),
                ECalculateDistanceRequestId);
            break;
        }
    if (iXmlRpcRequest)
        {
        // call remote method using XmlRpcClient Engine DLL
        iXmlRpcClient->CallL(*iXmlRpcRequest);
        }
    }
```

First we delete previous request. Then we create new request with the request Id given in the argument. When new XML-RPC request is created user application invoke `CallL(CXmlRpcRequest&)` on `CXmlRpcClient` object.

3.2 Getting response from the server

Getting response from the Server is done in two methods. One is `ResponseStatusL` and second is `ResponseReceivedL`. In first method as we know from the chapter 2.2 “HTTP connections” library inform user application about the HTTP status we got from server. In the second method, we will get the object of class `CXmlRpcResponse` which represent XML-RPC response. The following example demonstrates how to get response from the remote server:

```
void CExampleAppUi::ResponseStatusL(TInt aStatusCode, const TDesC&
aStatusText)
    {
    iMainContainerView->RemoveXmlRpcCallWaitDialogL();

    // XmlRpc HTTP response should always have status code 200.
    // Other codes are treated as errors.
    if (aStatusCode != 200)
        {
        iMainContainerView->RunGlobalWarningNoteL(&aStatusText);
        }
    }

void CExampleAppUi::ResponseReceivedL(CXmlRpcResponse& aResponse)
    {
    if (aResponse.IsFaultResponse())
        {
        // Handling fault response
        return;
        }
    switch (aResponse.RequestId())
        {
        case ECalculateDistanceRequestId:
            // Do something with results
            Distance(aResponse);
            break;
        }
```

3.2.1 Handling XML-RPC fault messages

When XML-RPC fault message arrive from the Server to user application, `ResponseReceivedL(CXmlRpcResponse& aRespone)` is called. Then inside this method user have to check if response is fault response. By calling `ErrorCodeL()` and `ErrorMsg()` user can get status code and error message. The following example demonstrates how to check response:

```
void CExampleAppUi::ResponseReceivedL(CXmlRpcResponse& aResponse)
    {
    if (aResponse.IsFaultResponse())
        {
        TBuf<KFaultResponseMaxLength> respBuf;
        respBuf.Format(KErrFaultResponseFormat,
                       aResponse.ErrorCodeL(),
                       &aResponse.ErrorMsgL());
        RunGlobalWarningNoteL(&respBuf);
        }
    }
```

3.3 Dealing with errors

If something went wrong, other then XML-RPC fault message or system errors like out-of memory, than library will call `OnError(TInt aErrorCode)` passing error code. This method is called every time when an error was occurred during the call, a failure occurs if the sever does not respond at all. Also called when error occurred while parsing XML-RPC responses from the server.

This method will not be called when XML-RPC fault messages are received. This type of XML-RPC messages are reported through the `ResponseReceivedL()` method as mentioned on previous chapter.

4 The XmlRpcClient.dll's API

Class	Method	Description
CXmlRpcClient	NewL	Creates the new instance of the CXmlRpcClient class.
	NewLC	Creates the new instance of the CXmlRpcClient class, but also pushes instance to Cleanup Stack.
	Version	Gets the version of the CXmlRpcClient library.
	SetUriL	Sets the uri. The address of the XML-RPC host. The port must be specified as well. If nothing is specified the connection is made to localhost on the default port 8080.
	CallL	It opens an HttpConnection on the URL given by the SetUriL method and sends an XML-RPC request to the server. This method is asynchronous call in a separate thread and reporting responses, faults, and statuses through the supplied MXmlRpcClientObserver.
	Cancel	Cancel an outstanding transaction.
MXmlRpcClientObserver	ResponseStatusL	Called by the CXmlRpcClient when a HTTP headers was received from the server. When all okay, aStausCode contains code 200 and aStatusText contains text "OK".
	ResponseReceivedL	Called by the CXmlRpcClient when a response was received from the server.
	OnError	Called by the CXmlRpcClient when an error was occured during the call. Remote errors are transported as XML-RPC faults and are reported through the ResponseReceivedL().A failure occurs if the sever does not respond at all or returns a standard HTTP error code. Also called when error occured while parsing XML-RPC responses from the server.
CXmlRpcRequest	NewL	Creates new instance of this class with the name of the method to call and helper id only for client side to help recognize request sent to the server.
	NewLC	Like previous method, but also pushes instance to Cleanup Stack.
	AddParam	Adds XML-RPC param (an object of CXmlRpcValue) to XmlRpc request message.

Class	Method	Description
	`Id`	Return id of the request given in the NewL or NewLC methods
`CXmlRpcResponse`	`NewL`	Creates new instance of this class. This object is constructed by CXmlRpcClient. Copy constructor or assignment operator are not supported. This object stays valid unless new object from the server arrive.
	`NewLC`	Like previous method, but also pushes instance to Cleanup Stack.
	`IsFaultResponse`	Check if returned response from server is fault response.
	`ErrorCodeL`	Gets the error code reported by the remote XML-RPC server.
	`ErrorMsgL`	Gets the error message reported by the remote XML-RPC server.
	`Result`	Gets the result returned from server.
	`RequestID`	Gets the request Id for which the response is.

5 Summary

This paper has presented an overview of the XmlRpcClient library and demonstrated how to send XML-RPC request to the Server and how to receive and interpret XML-RPC response returned from the Server.

This library has been designed to do minimum to communicate with remote servers and call remote methods on that servers. This is full implementation of the XML-RPC protocol and can be used as communication layer in any Symbian applications.

Literature

[1]. “S60 Programming. A tutorial guide”, Wiley, 2007

[2]. "Developing series S60 Application”, Addison Wesley, March 01 2004