RPC and SOAP with FPC and Lazarus

Michaël Van Canneyt

October 29, 2006

Abstract

Free Pascal has support for RPC (Remote Procedure Call) using XML since a long time, but recently a new toolkit has been developed which allows to do RPC in a variety of ways. It has an extensible interface which allows to extend it in several ways.

1 Introduction

The Web Services Toolkit (WST) is a toolkit to do RPC with Free Pascal. Despite it's name, it is not limited to WebServices (using SOAP) but can be used to implement a RPC mechanism using many methods.

The WST can be compared to RemObjects for Delphi. Obviously, it does not nearly offer the same possibilities as RemObjects, but it serves the same purpose, offers a similar architecture and can be extended easily. It is still under development, but can already be used to implement rather complex services.

WST is being developed by Inoussa Ouedraogo, and can be downloaded from

`http://inoussa12.googlepages.com/`

or

`http://wiki.lazarus.freepascal.org/Web_Service_Toolkit`

To make the remoting work, a framework that implements network access is needed. For this, one of the following must be used:

Indy The Indy internet components that come with Delphi, will also work with FPC. The Indy packages can be installed in the Lazarus IDE.

`http://www.indyproject.org/Sockets/fpc/`

ICS The Internet Component Suite can be downloaded from

`http://www.overbyte.be/`

Synapse Synapse can be downloaded from

`http://www.ararat.cz/synapse/`

Each of these are a well-tested set of TCP/IP components, and WST can use any of them to implement transport over HTTP. Servers can be implemented using the Indy HTTP server, and the ICS TCP server.

So, what does WST offer ? Quite a lot already:

- Message formatting using SOAP or a binary protocol. However, this is configurable, any kind of message format can be implemented and registered.
- Several transport layers. Currently, messages can be transported via HTTP (using Indy, Synapse or ICS) or via plain TCP/IP or in-process: if the server side is included in the binary. New protocols can be added at will: they must implement a certain interface, and that is all.
- A helper application that converts an interface definition to a proxy object - used on the client to execute calls on the server - and a skeleton implementation for the service. It also generates metadata needed for WDSL generation and correct serializing.

The use of all this is demonstrated using several demo programs, which are included in the download.

The WST architecture becomes clear from the above enumeration: WST splits the RPC process out in several layers:

Interface definition described by an Object Pascal interface definition.

Marshalling (called the formatters in WST) is the way in which a call (and the response) is encoded in a message, which is sent from client to server. Currently a SOAP and Binary message can be created.

Transport This is the way in which the message (and response) is transmitted from client to server. This can be HTTP, direct TCP communication, or direct interface invocation.

The 3 layers are configurable independent of each other.

Below we'll demonstrate the use of WST in some simple examples.

2 Defining the service interface

There are 2 ways to use WST:

1. Implement a server and client and communicate with this server through TCP/IP.
2. Implement a client to communicate with a third-party webservice which is implemented using SOAP.

In both cases, the starting point is the same: a service interface definition must be written. This is a simple Interface definition as it exists in Object Pascal, with methods and properties. This definition is then used in the further development cycle.

If the goal is to create a client/server communication protocol, then implementing this interface is not a problem: it is something which one would do anyway. However, to use third-party webservices, one also needs the interface, and it is not likely that this party will publish a Object Pascal interface definition.

WebServices implemented using SOAP are defined through a Web Service Description Language (WSDL) document: a XML document which describes the Web service: what calls are available, the parameters to these calls and their types: all is described in the WSDL document. Most application servers publish this document if they wish to make a service publicly available.

To make a client for an existing SOAP server, it would be nice to have a WSDL-to-interface converter. Unfortunately, this does not yet exist: the interface must be coded manually.

Fortunately, coding an interface is not so hard. As an example, the random quote web service will be taken. The WSDL document for this service is available at

```
http://www.boyzoid.com/comp/randomQuote.cfc?wsdl
```

From this rather intimidating document, the following interface can be distilled:

```
IRandomQuote = Interface
  ['{2EB2E9D7-917A-11D5-8BC8-00409522C25A}']
  Function getQuote(HTMLFormat : Boolean) : String;
  Function getAllQuotes : String;
end;
```

Which is rather simple. The interface definition can be stored in the interface part of a `randomquote` unit.

Armed with this interface, the coding of a client (or a server) can be started. The WST contains a little program `ws_helper` which scans a unit for interface definitions. From this interface definition, several sources can be generated:

1. A proxy object. The proxy object is an object which implements the service interface. It contains the code to marshall the parameters into a valid request (a SOAP request or a binary request), and to send the request over the transport layer to the webservice. It then analyses the response and returns any parameters or raises an exception if the server reported an error condition.
2. An implementation skeleton. This creates a class declaration for a class that implements the service interface. This class descends from `TBaseServiceImplementation`; The class is then registered in the service repositore; the code for this registration is generated as well.
3. A service binding. This is a unit which binds the implementation of the service to the message formatting and transport layer. It is, in fact, the server-side pendant of the proxy object.

Except for rare cases, the proxy and binder code can be used as-is, i.e. it's not necessary to edit them. They should be simply regenerated when the service interface changes.

Which of these 3 units is generated, depends on the command-line options passed to the `ws_helper` command.

3 Coding a client

The proxy object generated from the interface defition of the IRandomQuote interface looks as follows:

```
TRandomQuote_Proxy=class(TBaseProxy,IRandomQuote)
Protected
  class function GetServiceType() : PTypeInfo;override;
  function getQuote(HTMLFormat : Boolean):String;
  function getAllQuotes():String;
End;
```

The implementation of the .GetQuote method is generated as follows:

```
function TRandomQuote_Proxy.getQuote
                            (HTMLFormat : Boolean):String;

Var
  Ser : IFormatterClient;
  S : string;

Begin
  Ser:=GetSerializer();
  Try
    Ser.BeginCall('getQuote',GetTarget,Self as ICallContext);
      Ser.Put('HTMLFormat', TypeInfo(Boolean), HTMLFormat);
    Ser.EndCall;
    MakeCall;
    Ser.BeginCallRead((Self as ICallContext));
    S:='getQuoteReturn';
    Ser.Get(TypeInfo(String), S, result);
  Finally
    Ser.Clear;
  End;
End;
```

The code is rather straightforward. The `GetSerializer` returns an IFormatterClient instance, which will pack the message for the .GetQuote call. What the exact serializer instance is which will be returned is dependent on the creation parameters for the proxy object, as will be shown below. After the serializer has done it's work, the `MakeCall` method is called, which will send the message to the server, and receives the response from the server. The serializer then reads the response message, and stores it.

Any method defined in the interface will be encoded in a similar manner.

How can this proxy object be used in a client application ? The answer is very simple. The following client application will retrieve and display a quote:

```
program test_randomquote;
{$mode objfpc}{$H+}
uses
  Classes, SysUtils,
  base_service_intf, service_intf,
  soap_formatter,
  synapse_http_protocol,
  randomquote,  randomquote_proxy;

Const
  sADRESS = 'http:Address=http://www.boyzoid.com/comp/randomQuote.cfc';
  sTARGET = 'urn:randomQuote';
  sSERVICE_PROTOCOL = 'SOAP';

Var
  Q : IRandomQuote;
  SRes : string;

begin
  SYNAPSE_RegisterHTTP_Transport();
  Q:=TRandomQuote_Proxy.Create(sTARGET,
```

```
                                  sSERVICE_PROTOCOL,sADRESS);
  Try
    SRes := tmpObj.getQuote(True);
    WriteLn('Received quote: ',SRes);
  Except
    On E : Exception Do
      WriteLn('Caught exception : ',E.Message);
  End;
end.
```

Despite the fact that this is a very short program, a lot can be said about this source code. The `uses` clause of this program deserves some explanation. It contains the following units

base_service, service_intf these units contain the basic definitions of the WST. They define the abstract interfaces for transport layers, message formatters, service registry and so on.

soap_formatter this unit contains the SOAP message formatter for WST. Including this unit will automatically register the SOAP message format in the WST system. The standard `binary_formatter` registers a binary message format. At least one message format should be included in the program.

synapse_http_protocol This implements the HTTP transport mechanism in WST. This implementation of HTTP transport uses the Synapse package. The `SYNAPSE_RegisterHTTP_Transport` call actually registers the HTTP transport in the SWT system.

After the HTTP transport is registered, an instance of the proxy object is created. The constructor contains several arguments, and is defined as follows:

```
constructor Create(Const ATarget   : String;
                   Const AProtocol : IServiceProtocol);

constructor Create(Const ATarget    : String;
                    Const AProtocol  : string;
                    Const ATransport : string);
```

The first declaration shows what happens when creating a proxy object. 2 arguments are passed to the constructor:

ATarget this is the name which identifies the service. In the above example, this is `'urn:randomQuote'`. This is necessary to identify the metadata for the service (the names and types of arguments etc.).

AProtocol the protocol is actually a combination of marshaller and transport layer, this can be seen from it's declaration:

```
IServiceProtocol = Interface
  function GetSerializer : IFormatterClient;
  function GetCallHandler : ICallMaker;
  function GetTransport : ITransport
  procedure SetTransport(AValue : ITransport);
End;
```

Luckily, no IServiceProtocol must be created in code, the proxy object can do this. For this, the second constructor is made: here 2 arguments `AProtocol` and `ATransport` can be specified. These strings describe the message formatter and transport layer that should be used. The `AProtocol` parameter describes the message formatting: Each message formatter registers itself with a unique name. For the SOAP formatter, this name is simply 'SOAP'.

The `ATransport` parameter describes the transport to be used. It is of the form `transport:transport properties`. The first is the name of the transport mechanism, currently this must be one of the following:

HTTP a HTTP protocol. In the above example, the Synapse implementation of the HTTP protocol will be used.

TCP a TCP protocol. Implemented using ICS.

SAME_PROCESS Here, the service is implemented inside the application itself.

The transport properties is a series of name=value pairs describing the parameters needed by the transport layer to make the connection. The transport layer will decode and apply these parameters. In the above example:

```
sADRESS = 'http:Address=http://www.boyzoid.com/comp/randomQuote.cfc';
```

This means that the HTTP transport layer will be used, and the URL of the service will be passed on to the transport layer as the 'Address' parameter.

The rest of the code is rather simple: the `getQuote` method of the proxy object is invoked, and it's result printed on standard output. Compiling and running this example results in the following display on the console:

```
gru: >./test_randomquote
Received quote:
<em>
"Silence is golden when you can''t think of a good answer."
</em><br><strong>Muhammad Ali</strong>
```

Some linebreaks have been added for formatting purposes.

As can be seen from the example above: It takes no more than 15 lines of actual code (not counting constant declarations and uses clauses) to create a client for webservices. About 5 for the interface definition, and another 10 for the client implementation.

The rest - actually the bulk - of the code is handled by the WST code.

4 Coding a server

Coding a server is equally straightforward: again, the bulk of the code is taken care of by the WST helper application. All that needs to be done is implement the interface, and call the registration routines.

If the helper application is invoked as follows:

```
gru: >ws_helper -i -b randomquote.pas -o.
ws_helper, Web Service Toolkit 0.3 Copyright (c) 2006 by Inoussa OUEDRAOGO
File "randomquote.pas" parsed succesfully.
gru: >
```

then 2 files are generated: `randomquote_imp.pas`, `randomquote_binder.pas`. The first one contains an empty implementation of a class implementing the `IRandomQuote` interface. It looks as follows:

```
TRandomQuote_ServiceImp=class(TBaseServiceImplementation,IRandomQuote)
Protected
  Function getQuote(HTMLFormat : Boolean) : String;
  Function getAllQuotes : String;
End;
```

The `TBaseServiceImplementation` class takes care of registering the service in the service repository (a server can implement multiple services). The registration code is also included in the unit:

```
procedure RegisterRandomQuoteImplementationFactory();
Begin
  GetServiceImplementationRegistry().Register('IRandomQuote',
  TImplementationFactory.Create(TRandomQuote_ServiceImp) as IServiceImplementatio
End;
```

This registration procedure must be called before the service can be used.

For the purposes of this article, a dummy implementation for the 2 methods will be made:

```
function TRandomQuote_ServiceImp.getQuote(
                                  HTMLFormat : Boolean):String;
Begin
  if HTMLFormat then
    Result:='<em>Hello, World !</em>'
  else
    Result:='Hello, World !';
End;

function TRandomQuote_ServiceImp.getAllQuotes():String;
Begin
  Result:='No more quotes available';
End;
```

That's it. The quote service is ready to be used. To demonstrate the binary format and the in-server calling mechanism, the testapplication will be slightly rewritten:

```
program do_randomquote;
{$mode objfpc}{$H+}
uses
  Classes, SysUtils,
  base_service_intf, service_intf,
  binary_formatter,server_binary_formatter,
  same_process_protocol,
  randomquote,
  randomquote_imp, randomquote_binder, randomquote_proxy;

Const
  sADRESS = 'SAME_PROCESS:Adress=IRandomQuote';
  sTARGET = 'urn:randomQuote';
  sSERVICE_PROTOCOL = 'binary';
```

```
Var
  Q : IRandomQuote;
  SRes : string;

begin
  Server_service_RegisterBinaryFormat;
  RegisterRandomQuoteImplementationFactory;
  Server_service_RegisterRandomQuoteService;
  SAME_PROCESS_Register_Local_Transport;
  Q:=TRandomQuote_Proxy.Create(sTARGET,sSERVICE_PROTOCOL,sADRESS);
  Try
    SRes := Q.getQuote(True);
    WriteLn('Received quote: ',SRes);
  Except
    On E : Exception Do
      WriteLn('Caught exception : ',E.Message);
  End;
end.
```

The program looks much the same as the earlier version. The following things should be noted:

1. The `soap_formatter` unit has been replaced with the `binary_formatter` and `server_binary_formatter` units. The server message formatter is different from the client message formatter, so both must be included in a program which acts both as client and server. For a server, only the latter unit would need to be included.
2. the `randomquote_binder` `randomquote_imp` units are included: This is the server part of the application.
3. Finally, the `same_process_protocol` unit is included. This comes in stead of the `synapse_http_protocol` unit.

Obviously, the initialization of the various server components must be done:

formatters Client binary formatter is registered automatically by including the `binary_formatter` unit. The server binary formatter needs to be registered explicitly with the `Server_service_RegisterBinar` call.

Interface Implementation the implementation of the `IRandomQuote` is registered using the `RegisterRandomQuoteImplementationFactory` call.

Bindings The binding of the `IRandomQuote` interface is registered with the `Server_service_RegisterRandom` call.

Finally, the in-process calling of the service is enabled with the `SAME_PROCESS_Register_Local_Transport` call.

The rest of the code is the same, except that 2 of the creation parameters to the proxy object are now different:

sADRESS Now specifies the in-process protocol:

```
sADRESS = 'SAME_PROCESS:Adress=IRandomQuote';
```

The address parameter is now simply the name of the interface.

sSERVICE_PROTOCOL Now specifies the binary protocol:

```
sSERVICE_PROTOCOL = 'binary';
```

The target remains the same, obviously, as the same service is used.

And with this, the program is ready to be compiled and run:

```
gru: >ppc386 -Fu../../ do_randomquote.pas
gru: >./do_randomquote
Received quote: <em>Hello, World !</em>
```

Obviously, for a HTTP server or a TCP server, slightly more code is needed to connect the transport mechanism to the service binding. Currently, this still needs to be coded manually. Examples are available in the WST distribution:

1. Code to include the service as a module in Apache.
2. Code to include the service in a Indy HTTP server application.
3. Code to include the service in a TCP server (using ICS)

Closer examination of the code in these examples will show that the extent of the code that is needed to make them running, is very limited indeed. The expectation is that this code will be refactored to some components that can be manipulated in the Lazarus IDE, making coding of the server much easier.

5 Conclusion

While not yet 100% complete, the WST toolkit is definitely worth a look if a RPC mechanism in FPC must be implemented: it makes writing client/server applications really simple. If a SOAP client must be written, it is currently the easiest way of working. Although the missing support for writing an Interface from a WSDL document makes writing the interface a bit tricky in the case of more complex interfaces, this is made up for by the extreme simplicity of the rest of the coding: WST takes care of all the gory details. If support for XML-RPC and analysing of a WSDL documents is implemented, the WST could well become the tool of choice for all remoting needs for Lazarus/Free Pascal programmers.