Protocol testing with RemObjects SDK

Michaël Van Canneyt

April 2009

Abstract

The RemObjects SDK offers a wide variety of channels (transport mechanisms) and message types (messaging protocols). An application can make use of more than one message type or transport channel. This offers the opportunity to compare the speeds of the various protocols and channels.

1 Introduction

In a previous article, the RemObjects SDK remoting toolkit was presented: It offers all tools needed to communicate easily between client and servers, most notably webservices.

Webservices are commonly (not to say almost exclusively) implemented using HTTP and SOAP as transport and communication protocols. For publicly available services, this is certainly a good thing: If all applications use the same language, then they can more easily interact and communicate, and RemObjects SDK makes it very easy to create both servers and clients for this.

However, Remobjects offers a lot of other possibilities. The aim of this article is to show that the standard solution (SOAP messages over a HTTP transport layer) is not always the best solution: in fact, when communication speed is a factor, it is much better to choose one of the alternatives. Choosing an alternative does not mean that the service will be private: it is perfectly possible that a server offers 2 (or even more) ways of communication. This will be shown as well.

Along the way, some limitations and ways of customizing the transport will be pointed out.

2 The service interface

A remobjects server starts - like any webservice - by defining the service API. For the speed test application, 3 calls will be defined. Each call represents a typical usage scenario for webservices.

To define the API, the easiest is to start a Remobjects server application, using 'File - New' and in the category 'Remobjects SDK', choosing 'VCL Standalone server'. Delphi will create a new server project (and optionally a client project), which we'll name 'Speed-Demo'. Choosing then the 'Edit Service Library' menu item from the 'Remobjects SDK' menu will fire up the Service Builder, where the API can be defined.

An interface with 3 calls will be defined, which, when translated to Object Pascal, looks like this:

```
ISpeedService = interface
  function Sum(const A: Integer; const B: Integer): Integer;
```

Figure 1: Defining the API in the Remobjects Service Library
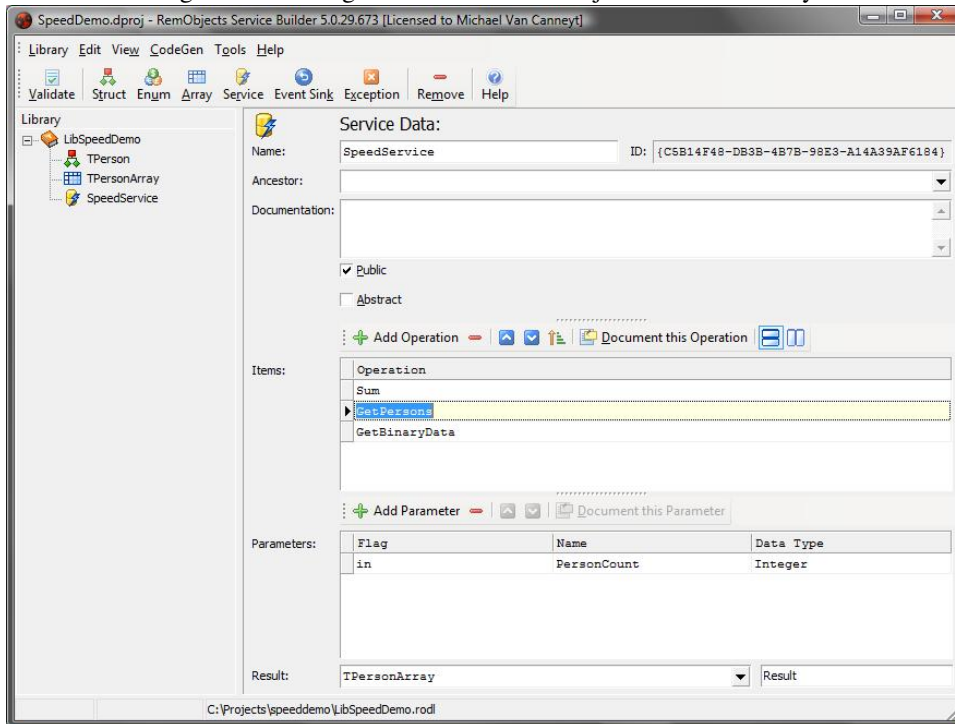

```
  function GetPersons(PersonCount: Integer): TPersonArray;
  function GetBinaryData(MaxSize: Integer): Binary;
end;
```

The first call `Sum` is a short call, intended to demonstrate short communications between client and server. The second call `GetPersons` is intended to demonstrate a common use for Webservices: retrieving information from a server, in this case, an array of `TPerson` records:

```
TPerson = class(TROComplexType)
published
  property ID : Integer;
  property Firstname : String;
  property Lastname : String
end;
```

The `PersonCount` parameter can be used to specify the number of persons in the array, so various packet sizes can be tested.

The third and last call `GetBinaryData` is intended for another common use: sending data in binary form. The parameter `MaxSize` tells the server how large the package must be. This call can be used to test how well a protocol deals with raw data. The server will return simply random data in the package.

In the Remobjects Service builder, this API should look as in figure 1 on page 2

3 Implementing the API

When importing the service library into the 'SpeedDemo' server project, Delphi will generate 3 files:

1. an empty implementation (SpeedService_Impl) unit,

2. an interface unit (libspeeddemo_intf),

3. and an invoker unit (libspeeddemo_invk).

The first unit must be filled with the actual API code, the other units should not be edited.

The implementation of the first of the methods of the `SpeedService` service is straightforward:

```
function TSpeedService.Sum(const A: Integer;
                           const B: Integer): Integer;
begin
  Result:=A+B;
end;
```

The `GetPersons` call is slightly longer:

```
function TSpeedService.GetPersons
               (const PersonCount: Integer): TPersonArray;
Var
  P : TPerson;
  I : integer;

begin
  Result:=TPersonArray.Create;
  for I := 0 to PersonCount-1 do
    begin
    P:=Result.Add;
    P.ID:=I;
    P.FirstName:=FirstNames[i mod FirstNames.Count];
    P.LastName:=LastNames[i mod LastNames.Count];
    end;
end;
```

First, the result array is created. Then a loop is entered, which creates `TPerson` instances. The ID is filled with the loop variable, and the `FirstName` and `LastName` properties are filled with elements from 2 stringlists: `FirstNames` and `LastNames`. These arrays are filled from 2 files in the initialization code of the unit:

```
Procedure LoadpersonData;

Var D : String;

begin
  D:=ExtractFilePath(ParamStr(0));
  Firstnames:=TStringList.Create;
  Firstnames.loadFromFile(D+'FirstNames.txt');
  LastNames:=TStringList.Create;
  Lastnames.loadFromFile(D+'LastNames.txt');
end;
```

The stringlists are disposed of when the program is finished.

Finally, the `GetBinary` method is implemented:

```
Function TSpeedService.GetBinaryData
     (const MaxSize: Integer): Binary;
Var
  M : Array of Integer;
  I, Count : integer;

begin
  Count:=(MaxSize div SizeOf(Integer));
  SetLength(M,Count);
  for I:=0 to Count-1 do
    M[i]:=Random(Count);
  Result:=Binary.Create;
  Result.WriteBuffer(M[1],Count*SizeOf(Integer));
  SetLength(M,0);
end;
```

As can be seen, a dynamic array `M` is created and filled with random data. When the array is filled, the function result is created and filled with the data from the array. Since the Remobjects Binary type is nothing but a `TStream` descendent, the `WriteBuffer` method can be used to fill it in one fast call. With this, the demo service is implemented. Remains to create the server listening part.

4 The server communication endpoints

By default, the Delphi 'Remobjects Project' wizard has put a `TROIndyHTTPServer` server component on the server main form and has associated the `TROBinMessage` with it. This `HTTPServer` component can be connected to 3 additional message components through it's `Dispatchers` property:

ROSoapMessage for binary encoded messages. It is associated with the /soap path of the incoming request URL.

ROXMLRPCMessage for XML-RPC encoded messages. It is associated with the /xml-rpc path.

ROBinCompMessage for binary encoded, compressed messages. It is associated with the /bincomp path of the incoming request URL.

The server can be made to listen also for windows messages, or on a simple TCP/IP port, and on a named pipe. Each instance of these communication channels can handle only 1 kind of message. To make the server accept all 4 kinds of messages through these channels as well, 4 instances of each type will be dropped on the main form, and configured in a special way:

TROWinMessageServer This class listens for windows messages. The `ServerID` property - by which the client can find it - of the 4 instances is set to `SpeedDemo` plus the name of the message type it is associated with: `SpeedDemoBin`, `SpeedDemoBinComp`, `SpeedDemoSOAP` and `SpeedDemoXMLRPC`.

TRONamedPipeServer This class listens for connections on a named pipe. The name of the named pipe is again determined by the `ServerID` property. For the 4 instances,

the property is set in the same manner as for the `TROWinMessageServer` instances.

TROSuperTcpServer This class listens for connections on a TCP/IP port. Since there is no `ServerID` property, the only way to distinguish the 4 instances is by giving them 4 successive port numbers: 8095, 8096, 8097 and 8098 for the SOAP, Bin, XML-RPC and BinComp messages, respectively.

Oviously, the client will need to adapt the connection settings of it's communication channel depending on which message is used.

On the server main form we'll drop 4 checkboxes, each checkbox can be used to activate or deactivate a channel. This is easily done in the `OnClick` method handler, which looks like this for the checkbox controlling the named pipe server:

```
procedure TServerForm.CBNamedPipeClick(Sender: TObject);

Var B : Boolean;

begin
  B:=CBNamedPipe.Checked;
  BinNamedPipeServer.Active:=B;
  SOAPNamedPipeServer.Active:=B;
  XMLRPCNamedPipeServer.Active:=B;
end;
```

Similar exists for the 3 other types of server component. With this, the server is ready to be run. When the server is run for the first time, and the servers are activated, the Windows firewall may kick in and ask whether the ports should be opened for this program. Obviously, the program should be allowed to open these ports.

5 Creating the client

Depending on the options set when the server was created, the 'New Remobjects Project' wizard has already created a client project, ready with HTTP channel and Binary message component. This can now be filled with testcode. First of all, additional message channels must be dropped on the client form, one for each type of channel:

```
WinMsgChannel: TROWinMessageChannel;
NamedPipeChannel: TRONamedPipeChannel;
SuperTcpChannel: TROSuperTcpChannel;
DLLChannel : TDLLChannel
```

Note the additional DLL channel: The server can also be compiled as a DLL, and the communication can also be done through a call which is exported by the DLL.

The HTTP channel will be tested twice: once with the `KeepConnection` property set to `True`: this value will cause it to keep the HTTP connection open between the successive calls to the server. A second time it will be used with the `KeepConnection` property set to `False`: in this case the connection will be closed after each call, so a new connection is made per call.

Similarly, 3 additional message components must be dropped, so there are in total 4 message components:

```
Bin: TROBinMessage;
BinComp: TROBinMessage;
SOAP: TROSOAPMessage;
XmlRpc: TROXmlRpcMessage;
```

Note that the names of these components match the suffices used in the `ServerID` proper-
ties of the WinMessage and Named Pipe servers, or the paths in the HTTP Server instance's
URLs. 2 `TROBinMessage` instances will be used: once for compressed data, once un-
compressed.

To create test code, a whole range of checkbox components is used to enable or disable
particular tests:

- A checkbox per channel: checking a checkbox means the channel will be included
  in the test. The checkboxes are named `CBXYZ`, with XYZ replaced by the respective
  channel types.

- Similarly, a checkbox is created per message type, with a similar naming scheme.

- Finally, a checkbox is created per call (Sum, GetPersons, GetBinary).

Some spinedits are also added, to set the following test parameters:

- The number of calls that should be made (named `SECount`).

- The number of persons that should be fetched in the `GetPersons` call (`SEPersons`).

- The size of the data returned by the `GetBinary` call (`SEBinSize`).

And finally, an edit box is added where the hostname or IP address can be entered of the
machine where the server process is running. Leaving it empty means that the server is on
the local machine. With the addition of a start button and a memo in which log messages
can be displayed, the client application is ready to be run. It should look as in figure 2 on
page 7.

The test cycle is activated by the Start button and is straightforward:

```
procedure TClientForm.RunAllTests;

begin
  If CBTestDLL.Checked then
    RunTestChannel(DLLChannel);
  If CBTestWinMsg.Checked then
    RunTestChannel(WinMsgChannel);
  If CBTestHTTP.Checked then
    begin
    HTTPChannel.KeepConnection:=False;
    RunTestChannel(HTTPChannel);
    end;
  If CBTestHTTPkeepAlive.Checked then
    begin
    HTTPChannel.KeepConnection:=True;
    RunTestChannel(HTTPChannel);
    end;
  If CBTestTCPSuper.Checked then
    RunTestChannel(SuperTcpChannel);
  if CBTestNamedPipe.Checked then
```

Figure 2: The client and server running
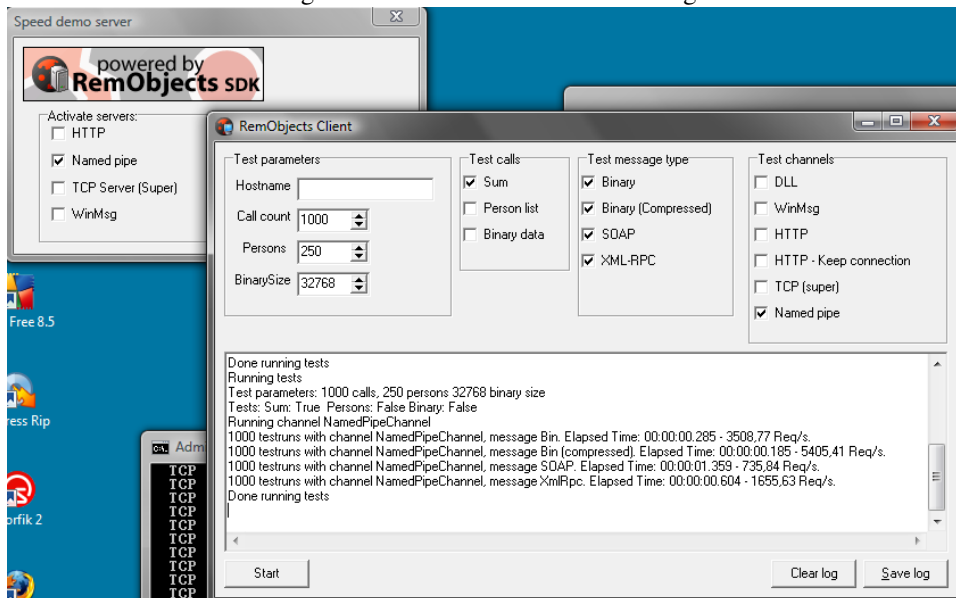

```
    RunTestChannel(NamedPipeChannel);
end
```

The `RunTestChannel` takes the channel to be used, and runs the tests for this channel once per message type:

```
procedure TClientForm.RunTestChannel
    (AChannel: TROTransportChannel);

begin
  RORemoteService.Channel:=AChannel;
  If CBBinMsg.Checked then
    RunMessageType(Bin);
  if CBBinCompressedMsg.Checked then
    RunMessageType(BinComp);
  If CBSOAPMsg.Checked then
    RunMessageType(SOAP);
  If CBXMLRPCMsg.Checked then
    RunMessageType(XMLRPC);
end;
```

Note that the first line sets the channel of the `RORemoteService` component to the channel that must be tested. Then the various message types are tested, depending on which checkboxes are set.

The `RunMessageType` method configures the channel and runs the test:

```
procedure TClientForm.RunMessageType(AMessage: TROMessage);

Var
  Start,Stop,RunTime : TDateTime;
  I,ACount : Integer;
  DoSum,DoPersons,DoBin : Boolean;
```

```
  TS,CN,MN : String;
  AReqS : Double;
  h,m,sec,ms : word;

begin
  ConfigureMessageChannel(RORemoteService.Channel,AMessage);
  RORemoteService.Message:=AMessage;
  DoSum:=CBTestSum.Checked;
  DoPersons:=CBtestperson.Checked;
  DoBin:=CBTestBinary.Checked;
  ACount:=SECount.Value;
  Start:=Now;
  For I:=1 to ACount do
    RunTests(DoSum,DoPersons,DoBin);
  Stop:=Now;
  RunTime:=Stop-Start;
  CN:=RORemoteService.Channel.Name;
  If RORemoteService.Channel=httpchannel then
    If HTTPChannel.KeepConnection then
      CN:=CN+' (keepalive)';
  MN:=RORemoteService.Message.Name;
  If RORemoteService.Message=Bin then
    if Bin.UseCompression then
      MN:=MN+' (compressed)';
  TS:=FormatDateTime('hh:nn:ss.zzz',RunTime);
  DecodeTime(RunTime,h,m,sec,ms);
  AReqs:=(h*3600+m*60+sec+(ms/1000));
  If (AReqS<>0) then
    AReqs:=ACount/AReqs;
  Log(Format(STestResult,[ACount,CN,MN,TS,AReqS]));
end
```

The call to `ConfigureMessageChannel` will configure the channel for use with the selected message: it sets the port number, or sets the `ServerID` property of the channel component, so they match the server endpoint.

After that the tests to be run are identified, and after the start time is saved, the tests are run using the `RunTests` method. When the loop is finished, the end time is noted, and the rest of the method is concerned with formatting the test result message: It displays the number of runs, the elapsed time, and the number of requests per second, which is computed from the elapsed time.

The `RunTests` method by comparison is very simple, it runs the tests and displays a message if something goes wrong:

```
procedure TClientForm.RunTests(DoSum,DoPersons,DoBin : Boolean);

Var
  P : TPersonArray;
  B : TStream;
  I : ISpeedService;

begin
  I:=(RORemoteService as ISpeedService);
  if DoSum then
```

Table 1: Sum Test on a local computer

Channel	SOAP	XMLRPC	Bin Comp.	Bin
HTTP	23,85	23,74	25,10	25,53
HTTP (keep con.)	405,84	622,28	942,51	865,80
SuperTCP	634	1233,05	3164,56	3030,52
Named Pipe	736,92	1602,56	5847,95	5347,59
DLL	791,77	2066,12	8849,56	7692,31
WinMsg	811,69	2066,12	10416,30	9523,81

```
    if I.Sum(1,2)<>3 then
      Log('Sum call failed');
  If DoPersons then
    begin
    P:=I.GetPersons(SEPersons.Value);
    if (P.Count<>SEpersons.Value) then
      Log('Person count wrong');
    P.Free;
    end;
  If DoBin then
    begin
    B:=I.GetBinaryData(SEBinSize.Value);
    If (B.Size>SEBinSize.Value) then
      Log('Binary size too large');
    B.Free;
    end;
end;
```

Note that the array and binary data returned by the server must be freed explicitly by the client.

6 Timing Results

With all this in place, some extra logging calls and checks for errors can be added (the interested reader can consult the sources that are on the disc accompagnying this issue) and the application is ready to be used.

To test, each of the tests will be run 1000 times, once on a local computer, once over the network. 250 persons will be requested, and a Binary data size of 32Kb will be specified. These parameters can be tuned of course.

The server was run in all tests on a similar machine: a Lenovo T61 laptop with dual core Intel centrino, 2Gb ram, running Windows Vista. No other programs were running.

The timing results of a test run for the Sum algorithm is displayed in table 1, the numbers are the number of requests per second: higher numbers are better.

The sum test is an indicator of when the message size is rather small. The absolute values of the results are less important: they depend on the speed of the CPU and memory, they vary slightly if the tests are repeated several times. More important are the relative values: the ratios of values stay more or less constant across various test runs that were performed.

Here are some observations that can be deduced from the values in the table.

- fastest results are obtained with the windows msg protocol, using a binary message. The only surprise is that it is faster than the DLL channel. A possible cause - but this

Table 2: Person Test on a local computer

Channel	SOAP	XMLRPC	Bin Comp.	Bin
HTTP	18,95	19,26	25,10	25,28
HTTP (keep con.)	44,21	45,50	354,11	501,24
SuperTCP	46,05	47,77	504,80	839,63
Named Pipe	46,79	47,91	525,49	959,10
DLL	47,52	49,89	599,16	1077,21
WinMsg	46,68	49,01	577,37	1067,70

  needs careful investigation - is that because client and server run in 2 processes, the dual core allows parallel execution of algorithms. In the DLL, all is run in a single thread, and hence on a single core.

- The HTTP protocol without keeping the connection performs particularly bad: The most likely cause is the linger time of the socket: the fast repeatedly opening and closing of the connections causes Windows to run out of sockets. Another cause could be the server implementation, which may not be optimal.

- The not compressed message is faster than the compressed message in the DLL and Message protocol. This is logical, since no CPU cycles are wasted on compressing the message, and the copy is a simple memory copy.

- Using the SOAP message is more than 13 times slower than the binary message in the DLL and message channel. This shows that the CPU must work very hard to format and decypher the XML. This difference diminishes as the transport overhead becomes larger: when using Named Pipe, HTTP and SuperTCP, the difference is much less pronounced: a factor 2 or 4.

- The difference in speed between HTTP and SuperTCP shows the overhead that the HTTP protocol imposes: it is a noticeable difference.

- This is also shown in the speed difference between SOAP and XML-RPC message: Both use XML, but XML-RPC is a far less verbose format as SOAP.

Running the `GetPersons` call (displayed in table 2) shows the same general trends, but there are nonetheless some surprises: the SOAP and XMLRPC message protocols are exceptionally slow, independent of the used channel: The difference between these protocols and the binary messages is a factor more than 20. This should not come as a complete surprise: encoding the 250 persons in a XML envelope will take much more time as encoding 2 numbers which happens in the sum call.

Also noteworthy is the fact that the compression factor now starts to show effect: the compression is only done for packages larger than 4k: this clearly slows down the communication (almost a factor 2 in DLL/WinMSg protocols), although the effect is less if the overhead of transport becomes higher.

Running the `GetBinary` test yields a surprise: the compressed binary format is slower than the SOAP and XML formatted messages, but the uncompressed binary format is faster. The surprise is not so much the fact that there is a difference between SOAP and the binary message formats - this was to be expected: XML doesn't handle binary data very well - but that the compression takes up a huge part of the time.

As observed, the compressed binary message actually slows down the communication very much, for all channels. This called for deeper investigation. The reason turns out to be the random nature of the data: by changing the `GetBinary` call so it simply zeroes out the data which it returns to the client, causing fast and very good compression. this drastically

Table 3: GetBinary Test on a local computer

Channel	SOAP	XMLRPC	Bin Comp.	Bin	Bin Comp.
HTTP	24,06	24,62	23,33	23,40	218,0
HTTP (keep con.)	198,49	269,32	119,15	735,62	298,0
SuperTCP	237,42	337,38	132,71	1433,00	559,91
Named Pipe	266,00	400,00	130,28	2898,29	613,50
DLL	292,74	454,55	133,52	3086,39	719,42
WinMsg	270,42	424,63	133,83	3496,39	751,31

Table 4: Tests over the network

Test	Channel	SOAP	XMLRPC	Bin Comp.	Bin	Bin comp 0.
Sum	HTTP	119,82	162,62	187,44	180,08	
	HTTP (keep con.)	197,85	271,59	320,51	320,51	
	SuperTCP	284,90	504,09	1186,24	1124,86	
Person	HTTP	23,13	19,97	98,62	104,07	
	HTTP (keep con.)	24,15	25,53	135,52	140,27	
	SuperTCP	26,97	28,16	221,04	287,52	
Binary	HTTP	55,50	63,53	40,93	89,28	158,28
	HTTP (keep con.)	65,41	76,04	45,79	113,46	244,02
	SuperTCP	82,71	100,32	55,93	199,08	421,76

improved the timings of the calls, as can be seen in the column headed 'Bin comp. 0', although it is still significantly slower than the uncompressed results.

When running the tests over the network, the following network is used: a 100Mbit network with several passive hubs and a switch between the client and server machines, with a low network load of less.

The DLL and WinMsg channel tests cannot be run over the network, and the named pipe channel test failed consistently: Windows was unable to locate the server computer despite all efforts to fix this problem.

All tests have been summarized in table 4. It confirms the trends that showed in the local tests: The differences between the message types are less pronounced, but they still exist:

1. In the SuperTCP channel, the difference between the SOAP and Binary message is the most pronounced. This is due to the size of the transmission: SOAP is a bulky protocol, and binary message is likely to fit in a single TCP packet, thus reducing drastically the TCP/IP overhead.

2. The HTTP protocol (without 'keep connection') performs better than when run on a local computer, a fact which calls for investigation.

3. The gain of compressing of the binary message becomes visible. Obviously, this is only true for data which compresses well and fast.

7   Conclusions

It is of course dangerous to draw conclusions from such simple tests. Common sense observations such as the fact that SOAP introduces more overhead as XML-RPC are confirmed by the tests, although the difference is not spectacular. The arguments in favour of compressing binary messages are not really compelling: this would have to be researched more

extensively in order to pull well-founded conclusions. The serious difference is made by the binary message and the use of the super TCP channel.

If one considers 2 typical use cases for remoting in an administrative intranet application, namely:

- Using plain SOAP calls over a HTTP protocol, as is the norm in webservices: a typical call would be the getpersons call.

- Using a plain TCP/IP protocol with Binary messages, as in a more classical N-tier solution using Datasnap, then a typical call would be the GetBinary call.

The speed difference between these 2 use cases is almost a factor 10 (20 if the data compresses well). Even if it was not so much, this factor surely makes the difference between an application that has a sluggish feeling, and an application which responds briskly to a user action: Classical client-server communication still has it's uses in this world dominated by webservices, SOA.

Despite this rough conclusion, there are some areas that need investigation, such as the slow results for the DLL channel and the extremely bad performance of the local HTTP channel. Some tests identifying the kinds of data that compresses well could also give an indication of the gain that can be expected from compression. The additional impact of a .NET server implementation (as opposed to the native Delphi implementation used here) would also be interesting to examine. This will be left for a future contribution.