IndySoap Demonstration Tutorial

Introduction

This tutorial shows how to use IndySoap for client/server RPC model using a unique number server as an example. As well as providing a demonstration of how to use IndySoap, this tutorial will also show how IndySoap can be used to meet these specifications.

Requirements

You need to have IndySoap version 0.02 or more recent installed in a supported version of Delphi (or Kylix?). All the source for this Tutorial can be found in the file KeyServer.zip

Overview

First, the specifications for the system are defined. Then an implementation of the core server is described. Then an interface is defined that will allow the server to meet the specification. Then a server to generate the key streams is written. Finally, an example client is written

Specifications

For the purposes of this tutorial, we will work with the following specifications:

1. *The KeyServer must serve up unique keys to multiple applications written using different development enivironments*

2. *A name is provided when requesting a key. Each different name will generate a different sequential series of keys*

3. *Users should be able to view the number of keys generated in the current session for all the names used*

4. *Upon provision of an appropriate username and password, administrators are able to reset the next value for a particular key name. This is only allowed if the network traffic is encrypted*

5. *HTTP must be used to communicate with the KeyServer as the Enterprise it supports is widely distributed*

6. *Reliability of the KeyServer is of paramount importance as the Enterprise will stop functioning when it is not running*

7. *Performance of the KeyServer is of considerable interest, particularly on the core network segment for the Enterprise*

Note: The specifications are a little odd, as the point is to work with IndySoap, not develop a real application

Core KeyServer Functionality

For the purposes of this tutorial, we will assume that the Core KeyServer functionality already exists, and consists of a single Unit called “CoreKeyServer.pas”. The interface for this unit is as follows:

```
procedure Start;
procedure Stop;

// Get the Next Key for a name
function GetNextKey ( AName : String ): integer;

// reset the value of the Key to the providede Value
procedure ResetKey ( AUserName, APassword, AName : string; ANewValue : integer);

// get a list of all known keys
procedure ListKeys(AList : TStringList);

// for a given key name, return stats
procedure GeyKeyStats(AName : string; var VStartKey, VNextKey : integer);
```

This is a fairly simple interface. It’s our job simply to expose this on the network using SOAP. The full implementation, which is thread safe, can be found in the accompanying zip file

Defining the Interface

IndySoap uses pascal interfaces as a metaphor for protecting the user from the complexities of SOAP. The intent is to provide an RPC mechanism with compiler type checking. So the first step is to define a pascal interface that exposes this functionality.

First we create a new file, KeyServerInterface.pas. Then we define our base interface, IKeyServer:

```
uses
  IdSoapTypeRegistry;

type
  IKeyServer = Interface (IIdSoapInterface) ['{7C6A85A1-E623-4D6F-BA03-915BD49CE7D4}']
  end;
```

We will call the Interface “IKeyServer”. There’s no great significance to the name, but it does need to be unique. The interface must descend from IIdSoapInterface (an internal rule IndySoap imposes), and it needs a GUID.

Obviously, the start and stop routines that CoreKeyServer.pas publishes are not to be exposed on the network, but the other routines should be. The first 2 functions are easy to publish:

```
type
  IKeyServer = Interface (IIdSoapInterface) ['{7C6A85A1-E623-4D6F-BA03-915BD49CE7D4}']
    function GetNextKey ( AName : String ): integer; stdcall;
    procedure ResetKey ( AUserName, APassword, AName : string; ANewValue : integer); StdCall;
  end;
```

Note the use of StdCall – all interface routines must have the calling convention StdCall.

The next 2 functions aren't quite so easy. If we published them as they are, the procedure ListKeys would be a problem. There is no built in support for transporting TStringList objects using SOAP, and it's not obvious how a non Delphi client would deal with it if we did provide some mechanism for it.

Soap provides a mechanism for transferring a list of strings like this, called an "array". IndySoap models Soap Arrays using Pascal dynamic arrays. If we want to transfer the list as an array, we need to define an interface that returns a dynamic array of string.

```
Type
  TKeyList  = array of String;

  procedure ListKeys(out VList : TKeyList);
```

This is a good first step. Later, when we implement the interface, we would simply fill up the VList with the known key names, and it would be returned as a Soap compliant array. (And an IndySoap client implementation would receive it as an identical array at the other end). We defined this as an out parameter, though it could equally have been a result of a function.

However we can do better than this. With the current declaration, the client will still need to request the stats for each name from the server individually, and this will be slow. It would be better to return all the information at once. In Soap terms, we can do this if we pass an array of complex structs containing the information back to the client.

In IndySoap terms, we model this as a dynamic array of classes, like this:

```
Type
  TKeyInformation = Class
  Private
    FName : string;
    FStartKey : integer;
    FNextKey : integer;
  Published
    Property Name : string read FName write FName;
    Property StartKey : integer read FStartKey write FStartKey;
    Property NextKey : integer read FNextKey write FNextKey;
  End;

  TKeyList  = array of TKeyInformation;

  procedure ListKeys(out VList : TKeyList);
```

When we populate this list and return it to the client, they will get all the information at once. The Class TKeyInformation is used instead of record type, since there is no RTTI information for record types in pascal, and the IndySoap engine needs the RTTI in order to build the SOAP packet from the information provided. This is the reason that the properties are published – to generate the RTTI

Because classes are used in place of records, but they are treated as records in many ways, the normal rules for handling classes may not apply. Consult the IndySoap documentation for further details

This gives us a final interface of

```
uses
  IdSoapTypeRegistry;

type
  TKeyInformation = Class
  Private
    FName : string;
    FStartKey : integer;
    FNextKey : integer;
  Published
    Property Name : string read FName write FName;
    Property StartKey : integer read FStartKey write FStartKey;
    Property NextKey : integer read FNextKey write FNextKey;
  End;

  TKeyList  = array of TKeyInformation;

  IKeyServer = Interface (IIdSoapInterface) ['{7C6A85A1-E623-4D6F-BA03-915BD49CE7D4}']
    function GetNextKey ( AName : String ): integer; stdcall;
    procedure ResetKey ( AUserName, APassword, AName : string; ANewValue : integer); stdcall;
    procedure ListKeys(out VList : TKeyList); stdcall;
  end;
```

Preparing the Interface for IndySoap.

IndySoap has a number of steps that need to be followed before an Interface can be published in SOAP. This is a quick checklist to use when developing an interface that will be published through IndySoap:

- Interface has Unique GUID
- Methods are StdCall
- Types Registered in the IndySoap Registry
- Interface implementation registered
- Method implementations are “published” not public
- ITI prepared

Type and Interface Registration

A basic rule for IndySoap is that all types used in interfaces that are published or consumed by IndySoap must be registered in the IndySoap Type Registry. The common types, such as string, etc, are already defined internally, so we don’t need to define them. Generally, when we define an interface, we will need to register any types that we defined ourselves. We have defined 2 classes, so we need to register them.

```
  IdSoapRegisterType(TypeInfo(TKeyInformation));
```

The first registration is straight forward. The Class TKeyInformation is registered. The second is more involved.

```
  IdSoapRegisterType(TypeInfo(TKeyList), '', TypeInfo(TKeyInformation));
```

Parameter 1: You must provide the TypeInfo for the type.
Parameter 2: Name. This defaults to the actual pascal name. Changing this is only for advanced use
Parameter 3: Only required for dynamic arrays, and then only under Delphi 4 and Delphi 5. This is the type information for the type of value that the array holds. If you are exclusively working in Delphi 6, then you do not need to include this parameter, as it is built into the TypeInfo already, so you can leave it off

Type registrations should generally be done in the unit initialization for the unit that declares the Interface. This way, the interface types are always registered

In addition to the types, it can also be useful to register the interface. (Specifically, this is required if you use the Delphi 6 RTTI to ITI convertor – refer IndySoap documentation for further details).

But IndySoap may use this registration later for other purposes, so it is recommended to register the interface:

```
  IdSoapRegisterInterface(TypeInfo(IKeyServer));
```

The routine IdSoapRegisterInterface is found in the unit IdSoapIntfRegistry, so this needs to be added to the uses clause too.

Implementation

Now that the interface has been defined, we must create a concrete implementation of it. This is best done in a different unit. So we create a unit KeyServerImplementation The bulk of this unit is fairly straight forward (see the source for further information).

The implementation of ListKeys is worth considering:

```
procedure TKeyServerImpl.ListKeys(out VList: TKeyList);
var
  LList : TStringList;
  i : integer;
  VStart, VNext : integer;
begin
  LList := TStringList.create;
  try
    CoreKeyServer.ListKeys(LList);
    SetLength(VList, LList.count);
    for i := 0 to LList.count - 1 do
      begin
      VList[i] := TKeyInformation.create;
      VList[i].Name := LList [i];
      CoreKeyServer.GeyKeyStats(LList [i], VStart, VNext);
      VList[i].StartKey := VStart;
      VList[i].NextKey := VNext;
      end;
  finally
    FreeAndNil(LList);
  end;
end;
```

The important thing to note is that the objects of the class TKeyInformation are created, and the Soap layer accepts responsibility for freeing them. The full rules for class lifetime management are documented in the Indy Documentation

Once the implementation has been defined, it must be registered with IndySoap, so that the server knows how to find the implementation when the interface is invoked. IndySoap supports 2 forms of registration, by class, and by factory routines. Registration by class is simpler:

```
initialization
  IdSoapRegisterInterfaceClass('IKeyServer', TypeInfo(TKeyServerImpl), TKeyServerImpl);
```

Again, this is best done in the initialization section of the unit that contains the implementation, so that the implementation is always registered

Building the ITI

IndySoap uses a structure called an “ITI” (Interface Type Information) to describe the Interface. ITI was created because interface RTTI is only available under Delphi 6. However other metadata about the interface has been added to the ITI as well.

The ITI is built by an IndySoap parser parsing the Pascal source for the IKeyServer interface. The IndySoap IDE package will automatically build the ITI pre compile if a file name with the same name as the project with the extension .IdSoapCfg exists.

The contents of the .IdSoapCfg file guide the ITI builder when it is building the ITI. For this project we would define the contents in this fashion:

```
[Source]
KeyServerInterface.pas

[Output]
BinOutput=KeyServerInterface.res
```

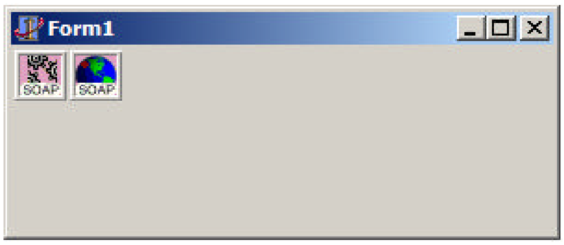
The contents of this file are described in the IndySoap documentation. Once this file is created, we simply need to recompile the KeyServer project, and the .iti file will be created in a res format ready for inclusion into the binary using the {$R xx.res} compiler command

If the Server project in the next section is called KeyServer.dpr, then this file would be called KeyServer.IdSoapCfg

Building the server

This tutorial starts with the 4 files already described, KeyServerCore.pas, KeyServerInterface.pas, KeyServerImplementation.pas, and KeyServer.IdSoapCfg. These can be found in Step1.zip. We will define a simple form based project (though you would usually define some kind of service for an application of this nature).

1. Create a working directory. This tutorial uses the path c:\temp\keyserver
2. extract the files from KeyServer.IdSoapCfg into the working directory
3. Create a new application (either VCL or CLX)
4. Save the project as KeyServer.dpr and the form as KeyServerForm.pas in the working directory
5. On the form, place 2 components, a TIdSoapServer, and a TidSoapServerHTTP:



6. For the TIdSoapServer, set the following values:

Property	*Value*
ITISource	islResource
ITIResourceName	KeyServerInterface
Active	TRUE

7. For the TIdSoapServerHTTP, set the following values

Property	*Value*
DefaultPort	2445
SoapServer	IdSoapServer1
Active	TRUE

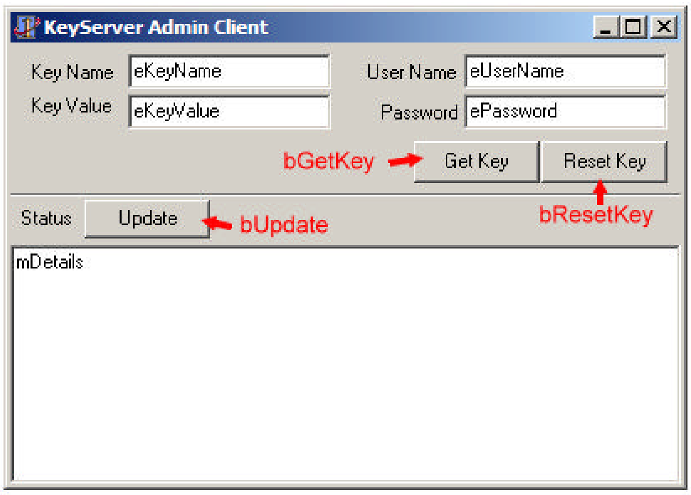
8. Add the unit KeyServerInterface to the project
9. Execute the program
10. In your browser (preferably IE), go to the following URL:http://localhost:2445/wsdl/IKeyServer. You should see the WSDL for the service
11. Add the unit KeyServerImplementation.pas to the project
12. Add the unit KeyServerCore to the KeyServerForm uses clause
13. In the OnCreate event for the form, start the Core
14. In the OnDestroy event for the form, stop the Core

The server is now complete. (This is saved as Step2.zip)

Building a Client

For demonstration purposes, we will build a testing/administration application. You would use these same techniques in a real application.

1. Copy KeyServer.IdSoapCfg to KeyClient.IdSoapCfg. This will ensure that the KeyServerInterface.res file is always up to date
2. Create a new application, eithe VCL or CLX
3. Save the project in the working directory as KeyClient.dpr and KeyClientForm.pas
4. On the form, create controls as shown here:



5. Add a TIdSoapClientHTTP to the form. Give it the following settings:

Property	*Value*
ITISource	islResource
ITIResourceName	KeyServerInterface
SoapURL	http://localhost:2445/soap/
Active	TRUE

6. Add the unit KeyServerInterface to the client uses clause
7. Add the following code to the bGetKey OnClick event:

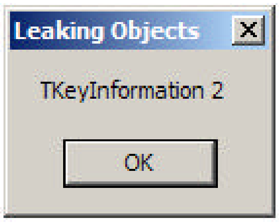
```
procedure TForm1.Button1Click(Sender: TObject);
var
  IKey : IKeyServer;
begin
  IKey := IdSoapClientHTTP1 as IKeyServer;
  eKeyValue.Text := inttostr(IKey.GetNextKey(eKeyName.Text));
end;
```

8. Make sure that the server is running, run the client, and click on the “Get Key” button. This should generate a sequential number each time you click the button

9. Add the following code to the bUpdate OnClick Event:

```
procedure TForm1.Button3Click(Sender: TObject);
var
  KeyList : TKeyList;
  IKey : IKeyServer;
  i : integer;
begin
  IKey := IdSoapClientHTTP1 as IKeyServer;
  IKey.ListKeys(KeyList);
  mDetails.Lines.Clear;
  for i := Low(KeyList) to High(KeyList) do
    begin
    mDetails.lines.add(KeyList[i].Name+
          ' '+inttostr(KeyList[i].StartKey)+
          '/'+inttostr(KeyList[i].NextKey));
    end;
end;
```

10. Run the client, and click the Update button. You will see the details as appropriate
11. When the client stops, you will get a message listing some leaking objects, one for each key name used:



TKeyInformation 2

12. The ListKeys routine returns a dynamic array of TKeyInformation. Just like a normal function call, the application accepts responsibility for the array of objects. The dynamic array will be finalized when it goes out of scope, but this will not clean the objects themselves up. The easiest way to do this is by adding this line to the for loop code:

```
      FreeAndNil(KeyList[i]);
```

which is very convenient in this case. An alternative approach is to ask the client to garbage collect all objects. Read the IndySoap documentation to fully understand the implications of this approach
13. Add the following code to the bResetKey button:

```
procedure TForm1.bResetKeyClick(Sender: TObject);
var
  IKey : IKeyServer;
begin
  IKey := IdSoapClientHTTP1 as IkeyServer;
  IKey.ResetKey(eUsername.Text, ePassword.Text, eKeyName.Text,
                StrToInt(eKeyValue.Text));
end;
```

Note that the user name and password are blank by default. Consult KeyServerCore.pas for further details

This step is saved as Step3.zip

Secure Network traffic

Specification #5 said that the key reset operations will only be possible if the network traffic is encrypted. One possible implementation is to encrypt all network traffic but this has both performance and administrative implications. It would be better if the server could inspect the communications in the case of the ResetKey routine. This is not a decision that can be made in the comms layer as it doesn't know which routine is being called. The decision must be made in the code implementing the interface.

This code can access information about the request itself – the raw packets, and communication elements themselves - through the GIdSoapRequestInfo thread variable in the IdSoapRequestInfo. This global variable will contain a descendent of TIdSoapRequestInformation that contains all these elements, what ever is appropriate for a given communication implementation. In addition, the base TIdSoapRequestInformation class contains several properties that summarise the communication environment

GIdSoapRequestInfo will be nil if the interface is invoked by some other method than IndySoap. (Server's running IndySoap can also run the Borland Soap library simultaneously, for example).

All we need to do as add the following lines of code to KeyServerImplementation.pas:

```
Uses
  IdSoapRequestInfo

  if not (GIdSoapRequestInfo.ClientCommsSecurity = ccSecure) then
    begin
    raise Exception.Create('Must be associated with a secure request');
    end;
```

Obviously, this will be raised every time that ResetKey is called, because we have not implemented a SSL layer. This is outside the scope of this tutorial, but it would be done by configuring IdSoapServerHTTP to use SSL

A Fast RPC implementation

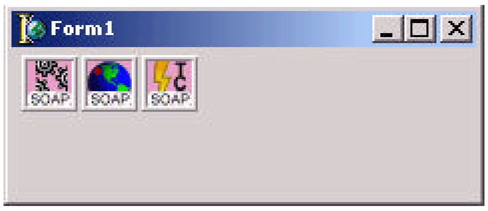
Soap is not an inherently fast RPC implementation. There is overheads associated with the XML encoding and decoding, and with the use of HTTP as a communications layer. For this reason, IndySoap includes alternative implementations of both the encoding layer, and the communications layer.

The alternative encoding layer uses a binary stream format. This format writes the data types to the stream in their native format, and that does not require any parsing. The format is not based on any published specification, and is documented in the IndySoap documentation.

The alternative communications layer uses a direct TCP/IP connection which is kept open. There is no inbuilt support for connection reestablishment – this is left to the application to manage. There is a very light weight protocol built on top of this for exchange of messages. This protocol is also not based on any published specification, and is documented in the IndySoap documentation.

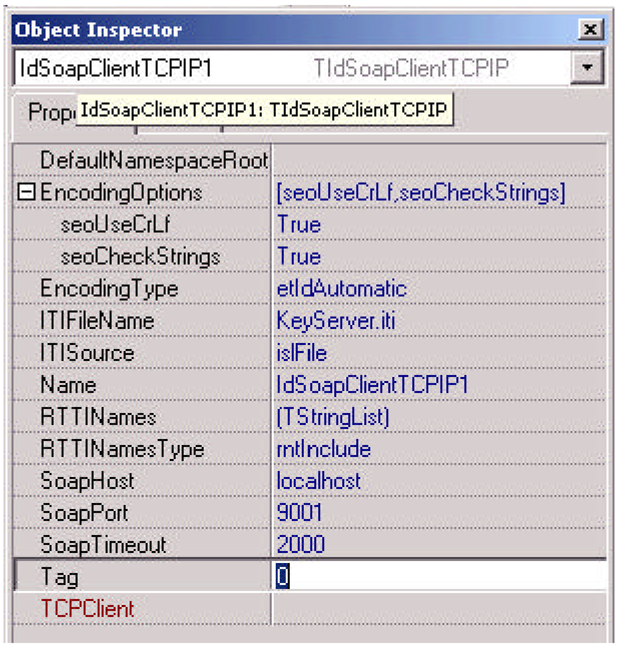
You must have IndySoap on both client and server to use these alternatives.

To set the server up for this, simply add a TIdSOAPServerTCPIP object to the form



and set the SoapServer to IdSoapServer1, and the DefaultPort to something useful (9001 in this example), and set Active to true. The server will now communicate using either communication protocol. Any IndySoap TIdSoapServer will respond to any of the supported encoding methods, and by default will respond with the same encoding method as the packet it received.

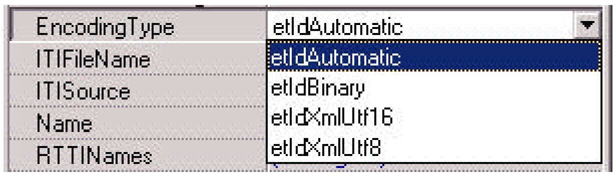
To make the client use the fast alternatives, add a TIdSoapClientTCPIP to the form, and set the properties as follows:


Object Inspector
IdSoapClientTCPIP1 TIdSoapClientTCPIP
Prop
IdSoapClientTCPIP1: TIdSoapClientTCPIP

DefaultNamespaceRoot	
⊟EncodingOptions	[seoUseCrLf,seoCheckStrings]
seoUseCrLf	True
seoCheckStrings	True
EncodingType	etIdAutomatic
ITIFileName	KeyServer.iti
ITISource	islFile
Name	IdSoapClientTCPIP1
RTTINames	(TStringList)
RTTINamesType	rntInclude
SoapHost	localhost
SoapPort	9001
SoapTimeout	2000
Tag	0
TCPClient	

The ITI settings are the same as for the HTTP client, and the Communication options are determined by the location of the server.

Both transport layers can use either encoding type. By default, the EncodingType is etIdAutomatic. In the Object inspector, the valid options are:



When the setting is etIdAutomatic, the communication protocol chosen defines the encoding type used. The HTTP client uses UTF-8 by default, and the TCP/IP layer uses Binary (which is the custom IndySoap encoding). So if we leave this at Automatic, it will use the fast encoding scheme. To use the new client, simply use this instead of the other:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  IKey : IKeyServer;
begin
  IKey := IdSoapClientTCPIP1 as IKeyServer;
  eKeyValue.Text := inttostr(IKey.GetNextKey(eKeyName.Text));
end;
```

Note: When this demonstration was written it was anticipated that more encoding options and delivery protocols would be implemented, so other options may exist that are not documented here.

This is saved as Step4.zip

Summary

This document has covered the creation of a basic client and server using IndySoap for an RPC layer. Further information can be found in the IndySoap documentation