Implementing a State-Based Application Using Web Objects in XML

Carlos R. Jaimez González and Simon M. Lucas

Department of Computer Science, University of Essex,
Wivenhoe Park, Colchester CO4 3SQ, UK
{crjaim,sml}@essex.ac.uk

Abstract. In this paper we introduce Web Objects in XML (WOX) as a web protocol for distributed objects, which uses HTTP as its transport protocol and XML as its format representation. It allows remote method invocations on web objects, and remote procedure calls on exposed web services. WOX uses URIs to represent object references, inspired by the principles of the representational state transfer (REST) architectural style. Using URIs in this way allows parameters to be passed, and values returned, either by value or by reference. We present a case study, in which an existing chart application is exposed over the Internet using three different technologies: RMI, SOAP, and WOX. WOX proves to be the simplest way to implement this application, requiring less program code to be written or modified than RMI or SOAP. Furthermore, as a consequence of its REST foundations, WOX is particularly transparent, since any objects that persist after a WOX call may be inspected with any XML-aware web browser. It is also possible to invoke methods of persistent objects through a web browser.

1 Introduction

Exposing applications over the Internet has become essential for many areas, due to the potential advantages of accessing data and objects from any place in the world. For this purpose, there are many existing distributed object technologies such as the Remote Method Invocation (RMI) [6] and the Common Object Request Broker Architecture (CORBA) [2], and web service technologies such as XML-RPC [7] and the Simple Object Access Protocol (SOAP) [5]. Both distributed object and web service technologies allow applications to be remotely accessible and allow more complex systems to be composed of components residing on geographically distributed machines. There are, however, some important differences between these two technologies, which can affect their suitability for specific types of applications.

Distributed object technologies base their functionality on two concepts: the *object's reference*, which allows a client application to refer to an existing object and execute operations on it; and the *object's state*, which is maintained between operation calls that can modify it. On the other hand, web service technologies in their current state do not have any of the concepts of distributed object programming and consequently have significant limitations. They do not support

R. Meersman and Z. Tari et al. (Eds.): OTM 2007, Part I, LNCS 4803, pp. 577–594, 2007.

access to remote objects, but instead they provide standalone services through the web by exchanging eXtensible Markup Language (XML) [3] messages, and attempt to solve the interoperability problem that exists with distributed object technologies. Although existing distribute object systems such as CORBA and RMI can work on the web by tunneling their requests through HTTP, this decreases their performance, and can be a technically demanding task.

The suitability of every technology to implement a given application and expose it over the web depends on a series of aspects that need to be considered, such as the importance of interoperability between languages, the encoding of messages, the maintenance of object references (object state), the efficiency in the transfer of data, the support and extensibility of data types, the ease of use and implementation, among others. To explore the issues, we present a case study based on a simple state based data charting application, which was originally implemented in Java. The original application is stateful, and allows data to be incrementally added to a chart object. A user creates a chart object with a title, and can then successively add data-points; this incremental style makes the chart application very easy to use for a client program. For our case study, the application needs to be exposed over the Internet, preserve the state of objects, and have access to remote objects. The use of XML as the encoding format for objects provides a simple text representation of any kind of data, which is machine and human readable, and it also provides a standard format to transfer data, which could lead to language independence. With the existing distributed object and web service technologies, it would be possible to implement such an application, but considerable extra programming effort would be required.

Web Objects in XML (WOX) is a web distributed object protocol that offers features from distributed object programming and web services to expose applications over the Internet. WOX allows the creation of remote objects, the invocation of methods on remote web objects and the invocation of remote procedure calls on exposed web services, among other operations. WOX uses HTTP as its transport protocol, XML as its format representation, and makes objects available through their own Uniform Resource Identifier (URI) [8], inspired by the principles of the Representational State Transfer (REST) [9] architectural style. WOX is simple and light-weight.

This paper is organized as follows. Section 2 introduces the WOX architecture, the set of client operations supported, and the format of the XML messages exchanged. A case study is presented in section 3, in which RMI, SOAP and WOX are evaluated to implement a chart application and deploy it over the web. Finally, conclusions and future work are given in section 4.

2 Web Objects in XML (WOX)

This section presents the WOX architecture, the operations allowed from a client application, the web browser interface to invoke operations on web objects, and the format of the XML messages exchanged between a WOX client and a WOX server. WOX is a working prototype that can be downloaded from

`http://algoval.essex.ac.uk/wox/Downloads.html`. It is straightforward to install and use. It is also accompanied of a set of client example programs.

2.1 WOX Design

We based the design of WOX on standard object-oriented concepts, distributed object programming and resource-oriented (REST) web services.

The WOX architecture is based on interfaces, which separates two important notions in distributed systems: the definition of behaviour given by the interface, and the implementation of that behaviour. A WOX server will have the implementation of the service, and a WOX client will have only an interface of that service in order to create objects, access them, and execute operations.

The nature of WOX required a strong object-oriented language, which fulfilled all of our requirements. We decided to implement it in Java [4] because it is a platform independent language (although we are also implementing a WOX serializer, and WOX client libraries in C#), which makes our system runnable on a variety of platforms; it is free and not restricted to the use of any commercial tool; and it highlights the concept of an interface, which makes it ideal for a distributed system.

It is important to note that RMI [6] also uses interfaces to allow access to remote objects, but in a very different way, which can be somewhat tedious in practice. We provide in our case study an explanation of the set of steps and changes required in order to implement the chart application using RMI. In this respect, a WOX client only needs a simple interface of the service.

A WOX client makes method invocations on a proxy that is created dynamically for the interface. A dynamic proxy is basically a class that implements a list of interfaces provided at runtime, such that method invocations through one of the interfaces on an instance of the class will be dispatched to another object [16]. The proxy translates the method invocation to XML and sends it to the WOX server. This mechanism uses the proxy design pattern [1].

One of the main ideas behind WOX is to expose remote objects through their own URI [8] and allow clients to have access to them. URIs are used to identify objects inspired by the principles of the REST architectural style. The notion of URI has been widely and successfully used by the web, and it is also a standard way to identify resources. Proponents of REST [10,11,12] argue that objects should be identified through their own URI, because this is how the web works.

Remote references in WOX are URIs, so that a client can refer to a remote object by specifying its URI. The XML encoding of a WOX object can be viewed by typing the URI into the address bar of a standard XML-aware web browser.

The concept of remote reference in WOX is widely used because a client can request a remote reference to a specific object, pass remote references as parameters, and also receive results from method invocations as remote references. A basic example of the mechanism used by WOX in a method invocation is shown in Figure 1, and the detailed steps carried out are described in the following list.

1. The WOX client program invokes a method on a remote reference (the way in which the client invokes a method on a remote reference is exactly the same as that on a local object, as far as the client program is concerned).
2. The WOX dynamic proxy takes the request, serializes it to XML, and sends it through the network to the WOX server.
3. The WOX server takes the request and de-serializes it to a WOX object.
4. The WOX server loads the object and executes the method on it.
5. The result of the method invocation is returned to the WOX server.
6. The WOX server serializes the result to XML and either the real result or a reference to it is sent back to the client. The result is saved in the server in case a reference is sent back.
7. The WOX dynamic proxy receives the result and de-serializes it to the appropriate object (real object or remote reference).
8. The WOX dynamic proxy returns the result to the WOX client program.

From the WOX client program's point of view it just makes the method invocation and gets the result back in a transparent way. WOX can refer to remote objects that reside in the same WOX server as that of the object (a relative URI can be used), or to remote objects located anywhere on the Web.

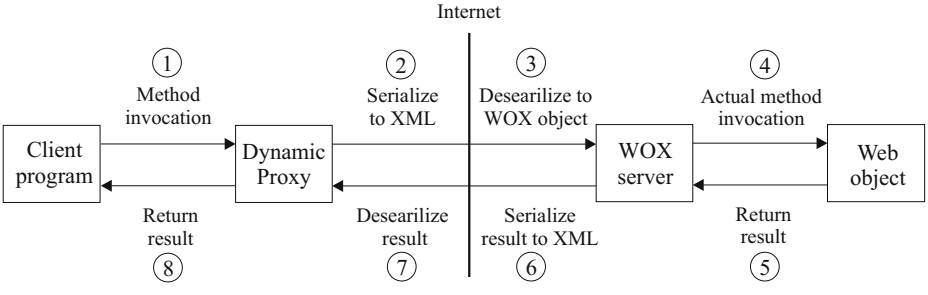

Fig. 1. A remote method invocation in WOX

2.2 WOX Client Operations

This subsection introduces the fundamental operations supported in WOX, and those not covered here are described in [14]. A set of client code examples can be found at `http://algoval.essex.ac.uk/wox/Examples.html`.

Creation of a new object. When a WOX client program requests the creation of an object, the WOX server creates the object and stores it. The WOX server returns to the client, based on the policy chosen, either the actual object or a remote reference to it. The remote reference is specified by a URI that points to the actual object. Once the WOX client program holds the actual object or the remote reference, it can invoke instance methods on it.

```
Chart chart = (Chart) WOXProxy.newObject
                      (serverURL, className, args, policy);
```

The code above creates a `chart` object of type `Chart`, which is a user-defined interface to hold either a real object or a remote reference to it. `WOXProxy` is a class provided by the WOX client libraries to allow the execution of WOX operations like `newObject` that takes four parameters: `serverUrl`, which represents the URL where the WOX server can be contacted to create the object (e.g. `http://csres109:8080/WOXServer/WOXServer.jsp`); `className` is the package qualified class name of the object to be created; `args` are the set of arguments used to construct the object; and `policy` is an integer number that represents whether a real object or a remote reference must be returned. The policy parameter also specifies the default mode in which WOX operates for future method invocations on the object (whether it returns real objects or remote references).

Request for a remote reference. A remote reference is requested by a WOX client program to invoke instance methods on the object to which the remote reference refers to, or simply to use it as a parameter in another method invocation. In this type of request, the WOX server looks for the specific object and returns a reference to it. The code example below gets a remote reference to a `Chart` object. The `objectUrl` parameter is the URL where the object is located. A `WOXException` will be thrown if there is no object at the URL specified.

```
Chart chart = (Chart) WOXProxy.getReference(objectUrl);
```

Static Method Invocation. A static method invocation is similar to a web service remote procedure call (like in SOAP[5], XML-RPC[7], or JSON-RPC[19]), in the sense that the WOX client requires no access to a particular object stored in the WOX server. When a static method invocation is requested, the WOX client invokes the `invokeService` method of `WOXProxy` class, in which it is specified the particular method to be executed (`methodName`), the class to which it belongs (`className`), the set of parameters received by the method (`args`), and the mode of operation (`policy`). The WOX server executes the method and returns the result. A `WOXException` will be thrown if the WOX server cannot find the method with the signature specified.

An important difference between web service remote procedure calls and WOX static method invocations is that a WOX client can also specify using the `policy` parameter, whether the result returned is the actual result, or a reference to it. Moreover, SOAP does not handle remote references, XML-RPC has a limited set of data types, and JSON-RPC does not support marshalling of objects with circular references.

An example of a WOX static method invocation is shown below, in which its return type is the user-defined interface `Manager`. The interface `Manager` is used in this example, but it could be used any other interface or concrete class, as long as it is consistent with the return type of the method invocation and the policy chosen in the WOX operation.

```
Manager manager = (Manager) WOXProxy.invokeService (serverUrl,
                            className, methodName, args, policy);
```

Instance Method Invocation. When a WOX client holds a remote reference, it can invoke instance methods on it in the same way as if it was a local object. The WOX mechanism will redirect the call through the network to the WOX server, which will in turn execute the method on the specific object. Once the method has been executed, the WOX server returns the result to the client. It should be noticed that a remote reference is actually a dynamic proxy on which the WOX client invokes methods. In the code below the client gets a remote reference to a `Chart` object, and then invokes its `getImage` instance method.

```
Chart chart = (Chart) WOXProxy.getReference(objectUrl);
byte[] graph = chart.getImage();
```

Destruction of an object. When a client does not require a remote object any more, it should be destroyed and garbage collected. Our current prototype of WOX includes an operation to explicitly destroy an object by providing its URI. However, in a future release, we could also allow clients to specify how long the object's life should be, and when the time for the destruction was reached then the object would be destroyed (i.e. removed from persistent storage). We appreciate that one of the reasons that SOAP does not allow object references is to eradicate any problems with distributed garbage collection, but believe that for many problem domains, object references are essential. We predict that for particular application areas and user communities, sensible usage policies will evolve given a suitably flexible framework.

2.3 Web Browser Interface

This subsection describes another way of executing WOX operations. As part of its REST foundations, WOX objects can be inspected by an XML-aware web browser. Furthermore, methods can be invoked on persistent objects using a web browser interface. This mode of operation allows clients to execute methods on a specific object without needing a Java client program.

Inspecting a web object. Every object that is persisted after a WOX call is stored in XML and can be inspected by a web browser. Below is the XML representation of a Manager object, which was used in one of our previous examples.

```
<object type="company.Manager" id="0">
  <field name="name">
    <object type="String" id="1">Robin Dyson</object> </field>
  <field name="age" type="int" value="35" />
  <field name="department">
    <object type="String" id="3">Finance</object> </field>
</object>
```

Invoking methods on a web object. Since every object can be identified uniquely, it is also possible to invoke methods on persistent objects through a web browser. A method invocation on a Manager object is as follows.

```
http://csres109:8080/WOXServer4/invokeMethod.jsp?
       objectId=Manager645313585&method=getName
```

The method invocation shown above would invoke the `getName` method of the `Manager645313585` object. The result of the method invocation would be returned as XML, which is the default mode of operation for WOX answers, but it can also be returned as html (by specifying `mode=html` in the query string), in which case only the string would be returned. There is a special case in which WOX can also return an image (`mode=image`) when the return type of a method is an array of bytes. This mode of operation will be presented in the case study.

Using a web browser, WOX is also capable of invoking methods with parameters of primitive data types, but not with parameters of other data types, like user-defined classes. This way of operation through the browser is similar to the way in which Apache Axis [13] allows to invoke methods of classes. The main differences are that Axis, which is SOAP based, does not have the concept of an object, thus the methods are invoked as if they were static methods. Another important difference in this mode of operation is that Axis does not support package qualified classes. In this respect, WOX enables the invocation of methods on any web object, and methods of any package qualified class.

Alternatively, WOX provides a user interface with all the possible methods to invoke on a specific web object. This can be accessed by typing the same URL presented before, but omitting the method parameter. WOX will present all the methods available for invocation on that object. Figure 5 in subsection 3.4 shows this web user interface with some of the methods available for a `Chart` object.

When clicking the *Invoke Method* button of the desired method, a query string is built with all the information needed for the method invocation. The request is made to the WOX server, which will send an answer via an XML message with the result of the method invocation.

2.4 XML Messages in WOX

An XML message in WOX is a request from a WOX client or the response sent back from a WOX server. A request can be any of the operations described in the previous section, while the response could be a real object, a remote reference, or an exception generated by the WOX server.

Since our system is based on object-oriented programming, the XML messages are generated by serializing objects of different classes according to the request made by a client, or the result or exception generated by the server. Some of these classes are shown in Figure 2, which contain attributes with all the information required to accomplish the request made. For example, the `WOXConstructor` class represents a request of object creation, where the `className` attribute specifies the name of the class of the object to be created, `types` and `args` are arrays that contain the parameters needed to construct an instance of the class,

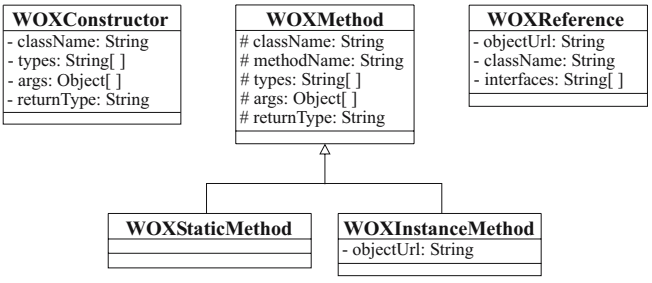

Fig. 2. Some WOX classes

and `returnType` indicates whether a remote reference or a real object must be returned. The `WOXMethod`, `WOXStaticMethod` and `WOXInstanceMethod` classes represent method invocations; and `WOXReference` remote references.

In addition to the classes shown in Figure 2, there are some other classes to represent all of the operations described in the previous section, such as `WOXDestructor`, `WOXUpdate`, `WOXUpload`, etc. Similarly there are classes to represent exceptions thrown by a WOX server, which inherit from `WOXException`.

The XML message shown below represents a static method invocation (an object of the `WOXStaticMethod` class). Although our WOX prototype (WOX serializer, WOX client libraries, and WOX server) is only implemented in Java (we are implementing a WOX serializer, and part of our WOX client libraries in the C# programming language), the XML messages should be appropriate for any other object-oriented programming language.

```
<object type="server.WOXStaticMethod" id="0">
  <field name="className">
    <object type="String" id="1">problems.test.MathClass</object>
  </field>
  <field name="methodName">
    <object type="String" id="2">returnArrayInt</object> </field>
  <field name="types">
    <array type="String" length="1" id="3">
      <object type="String" id="4">int</object> </array> </field>
  <field name="args">
    <array type="Object" length="1" id="5">
      <object type="Integer" id="6">5</object> </array> </field>
  <field name="returnType">
    <object type="String" id="7">Copy</object> </field>
</object>
```

2.5 WOX Server Operation

Every request of a WOX client is received by the WOX server as an XML message, it is de-serialized to a `WOXAction` object, and its `doAction` method is

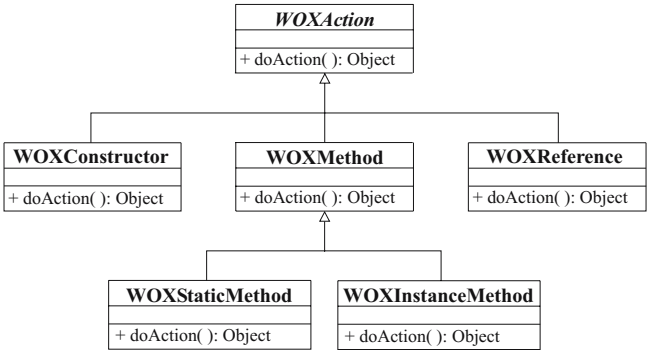

Fig. 3. `WOXAction` hierarchy diagram

executed. Figure 3 illustrates the hierarchy diagram for the `WOXAction` class. `WOXConstructor`, `WOXInstanceMethod`, `WOXStaticMethod` and `WOXReference` extend `WOXAction`. This mechanism allows a `WOXAction` object to invoke the `doAction` method of the appropriate class, which is coded differently, based on the type of request. This design is very flexible in the sense that there can be any other types of new requests without modifying the existing code. New types of requests would be represented as classes that extend the `WOXAction` class.

2.6 Limitations

WOX in its current state has also limitations, that somewhat can be seen as features not included in this release. A list is presented along with an explanation:

- Language independence: Although all the messages are represented in XML, WOX server and WOX client libraries have been only developed using the Java programming language, but we believe that the XML messages generated by WOX are appropriate to be implemented in any other class-based object-oriented programming language, such as C#, C++, Ruby, or Smalltalk. Our initial approach has been to develop our WOX serializer in C#.

The main issue to be considered when implementing WOX in other object-oriented programming languages is the serialization process, which is actually how objects will be represented in XML. This leads to consider multiple inheritance, and other programming language-specific features, which would be represented in the XML message.

- Security and ownership of objects: WOX does not support the concept of ownership of an object nor security policies for accessing web objects. Any client can have access to any objects created previously by another client. There is no restriction on who is executing what method of what object. It is just necessary for a client to have the URL to be able to access the object. For most practical applications, this would be a severe limitation, but at the prototype stage we did not want to be distracted by these considerations. However, since WOX is layered over HTTP, any HTTP-based security mechanism could be used.

- Asynchronous processes: All the operations in WOX are synchronous, which means that a result is immediately sent back to the caller. Asynchronous operations would allow clients to submit their jobs or processes and wait for results. Results would be returned to the caller normally via a callback operation.

- Object navigation: All the objects that persist after a WOX call can be inspected through an XML-aware web browser. An additional feature in WOX would allow clients to navigate through object graphs and be able to request specific nodes (objects). The use of an object-oriented db as the persistent storage for objects, such as db4o[17], could facilitate the implementation of this feature.

3 Case Study

This section presents a case study, in which a chart application is exposed through the Internet using three different technologies: RMI as a distributed object technology; SOAP as a web service technology; and finally our WOX prototype as a technology with features from both paradigms. The next subsections describe the chart application, and focus on the set of steps needed to implement it in the three different technologies. A different case study, describing a pattern recognition application, can be found in [15]. The case study presented in this section shows how best to deploy this chart application over the web with existing technologies and WOX. We chose this simple case study to introduce WOX, but we are already working on a more realistic application (the development of a game server) to demonstrate our ongoing work, which covers some of the limitations described in the subsection 2.6.

3.1 The Chart Application

The chart application we want to expose through the Web is used to input the data we collect from our experiments, get a statistical summary of the data, and get an image with a line graph of the data provided. The way in which the chart application works is very simple. We generate a new `Chart` object for every experiment, which will collect all the data for that particular experiment. The following line of code would create a `LineChart` object labeled *WOX Experiment*.

```
Chart chart = new LineChart("WOX Experiment");
```

Every time a new result from an experiment is ready it can be added to the `chart` object. The following code gets a new result from an experiment and adds it to the previously created `Chart` object. Since this is only for demonstration purposes we are using the `getResult` static method of the `Experiment` class (which returns a randomly generated `double` value), but this could be easily a method of any object which actually returns a result.

```
double x = Experiment.getResult();
chart.addValue(x);
```

Values are added to a `chart` object as results come from an experiment. Once the experiments have finished we can invoke the `getImage` method to get an array of bytes representing a line chart image.

```
byte[] image = chart.getImage();
```

In this simple application, the `chart` object would be created once, and experiments could be run over the world and use the `chart` object to add new results to it. This would also allow you to get a graph with the results at any point in time. This application is clearly state-based, because it needs to do the data collection, which will serve to do the statistical summary and generate the graph. The application consists of the `Chart` interface and the `LineChart` class.

The aim of the case study is to evaluate how well the deployment of the chart application is supported in the three different technologies: RMI, SOAP, and WOX. The deployment of this application through the Internet will allow clients ideally to refer remotely to chart objects that were created previously. In that way, clients do not need to hold `chart` data objects in their own computers, and they can also eventually save time by requesting a reference to an existing `chart` object. They can also add new values to the `chart` object and request an updated graph. A more realistic example would also allow chart styles to be set up and referenced. The implementation of this application will need to maintain the state of the objects and some mechanism to handle remote references.

3.2 Implementation Using RMI

In order to implement the chart application in RMI, several changes must be made to the original java source files, and follow a list of steps to make the objects remotely accessible to client applications. Since this is a very simple application the changes required will be only to the `Chart` interface and the `LineChart` class.

Modifying the original classes. The set of modifications to the classes in the chart application are described in the following list.

- Select the interface that clients will be using to access remote objects on the server. In this case the `Chart` interface would be used for this purpose.
- The `Chart` interface must extend the `java.rmi.Remote` interface provided in the Java API. The `java.rmi.RemoteException` exception must be thrown by all its method signatures.
- The `LineChart` class also needs to throw the `java.rmi.RemoteException` exception in every one of its methods and constructors.
- Those classes that will be traveling through the network must implement the `java.io.Serializable` interface. This is the case for the `LineChart` class.
- In order to expose `chart` objects remotely it is necessary either to extend the `java.rmi.server.UnicastRemoteObject` class or to specify that in the constructor of the class, when the object is created. In this case `LineChart` class must extend the `UnicastRemoteObject` class.

An alternative solution to implement this application using RMI could be to provide wrapper classes for every class or interface that requires modifications.

Those new wrapper classes would contain the modifications described in this section, and the original source classes would not be affected.

Deploying and running the application. Once the source files have been changed as described,the following steps must be carried out:

- Generate the stub for the remote interface `Chart`, by executing the `rmic` stub compiler (even though this is no longer required in Java 5 or later version, as dynamic proxies are generated). A client application will need the remote interface and the stub generated in order to access the remote objects.

- On the server side computer it will be required to copy all the classes and interfaces modified and start up the *Object Registry* (this is where client applications will find the remote objects). A server program needs to be written to create and register some objects in the *Object Registry*. In this case, the server program will have to create `LineChart` objects. These objects will actually be the remote objects available to the clients. It must be noticed that this server program creates some objects, which will be available for clients to access. Some extra methods would have to be provided in the `LineChart` class in order to allow clients to create their own objects.

- Client applications require the interface `Chart`, in addition to the stub generated by the `rmic` compiler (stubs are not required for Java 5). A sample code for a client application that uses RMI is shown below. The code gets a remote reference to a `chart` object, which has been created by the server program in the *Object Registry*. The client application can work with the remote reference as if it was a local object. It adds a new value to the `chart` object and then it gets an updated image with the line graph. One restriction as stated before is that clients are not able to create their own remote `chart` objects directly, even though extra methods could be provided to do so.

```
Chart chart = (Chart) Naming.lookup("chart01");
double x = Experiment.getResult();
chart.addValue(x);
byte[] image = chart.getImage();
```

Extra work would be needed to expose those objects over the Web, in which RMI tunnels its requests through HTTP. We could also have chosen to implement the entire chart application with Enterprise JavaBeans (EJB) Technology[20] (which is a more powerful technology that communicates through RMI) and an application server, such as JBOSS[21]; but we know that EJBs introduce many more unnecessary steps for this simple type of application. We would have had to deal additionally with EJB and home objects, home and local interfaces, and deployment descriptors.

3.3 Implementation Using SOAP

The implementation of the chart application using SOAP needs many more changes than those in RMI, because the application requires maintenance of the state of the objects between method calls, which SOAP does not support. In

order for SOAP to refer to remote objects a considerable extra programming effort is required, which leads to changes in the source classes or the creation of wrapper classes to encapsulate the instantiation and maintenance of remote objects. We prefer the latter method.

Creating wrapper classes. Following the idea of writing wrapper classes, there must be a mechanism to maintain the objects created by the client, and to refer to them. Figure 4 illustrates a class diagram that shows how to wrap up the original classes in the chart application in order to be able to expose chart objects remotely. The following changes are needed:

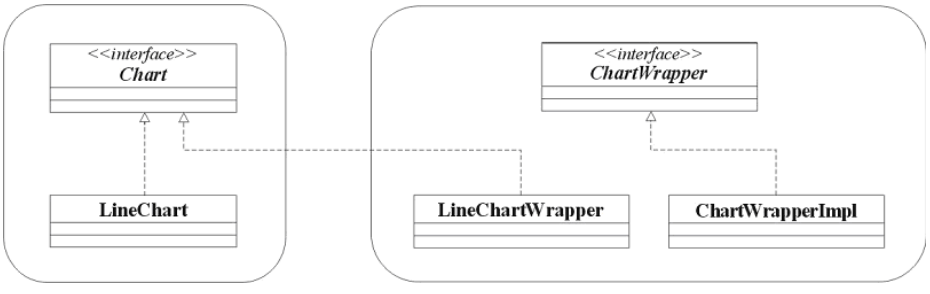

Fig. 4. Wrapper classes for the chart application

- A `ChartWrapper` interface that represents the SOAP interface of the service, which wraps up the `Chart` interface. `ChartWrapper` takes each instance method of the original `Chart` interface, and adds an id parameter to them. An example is illustrated below, where the signature of the `addValue` method in the `ChartWrapper` interface has now an additional parameter.

```
public void addValue(String id, double x);
```

- A `ChartWrapperImpl` class which is an implementation of `ChartWrapper` interface, and maintains a map of chart objects. The map can be maintained in memory or persistent storage. This implementation of the service will allow maintaining the state of chart objects on the server.

- A `LineChartWrapper` class that implements the `Chart` interface and provides a wrapper for `LineChart` class on the client side. It will be the interface of the service to the client. Clients will create chart objects of `LineChartWrapper` class instead of `LineChart` class, as can be seen from the code below. The class `LineChartWrapper` contains the logic to maintain the state of a chart object on the client side by keeping track of the id sent by the server to identify a specific chart object. `LineChartWrapper` class uses a proxy mechanism to send the requests from the client to the appropriate web service on the server.

```
Chart chart = new LineChartWrapper("WOX Experiment");
double x = Experiment.getResult();
```

```
chart.addValue(x);
byte[] image = chart.getImage();
```

- A Proxy class that receives the request from the client, extracts from it the web service name to be executed and the set of input parameters needed to invoke it. Each parameter must be mapped to the appropriate XML data type by using the serializers provided in SOAP, which can only serialize primitive data types, arrays, vectors, and user-defined classes that follow the Java Bean conventions. If there are other classes in the application to be serialized, then it would be also required to provide serializers for them. Those provided in SOAP require the classes to be modified to follow certain conventions in order to work properly. For example if a bean serializer is used, then the class to be serialized must follow the Java Bean conventions. On the other hand, one can write its own custom serializers, though it is a demanding and time-consuming task.

- The classes in the original chart application were modified to provide set and get methods for all their attributes, in order for them to be serialized properly.

Deploying and running the application. The deployment of the SOAP-version of this chart application involves the following steps.

- Copy to the server all those classes of the original application, in addition to `ChartWrapper` and `ChartWrapperImpl`, which define the SOAP service.

- Write a deployment descriptor to actually deploy the chart service on the server. The deployment descriptor specifies the name of the web service, the java class to be used for the service, the methods that can be invoked by clients, the scope of the web service, and the type mappings, which define the serializers to be used for user-defined classes.

- On the client side it will be required the classes of the original application, in addition to the Proxy class and the `LineChartWrapper` class, which will be the interface of the service to the clients.

- Running the application involves creating a `chart` object, adding some values to it, and getting the image with the line chart.

Despite the application is functional after all the modifications made and the lengthy procedure, it still has some drawbacks in the serialization efficiency. SOAP lacks of an efficient way for serializing arrays of primitive datatypes. Table 1 in subsection 3.4 shows a comparison in time and storage space between WOX and SOAP when they serialize an array of a primitive data type.

3.4 Implementation Using WOX

The implementation of this chart application in WOX is straightforward. For this application there is no need to create stubs for clients, rewrite classes to extend or implement interfaces, change method signatures to throw remote exceptions, write wrapper classes, or any extra programming effort, as long as the services to be exposed have an interface and an implementation.

The `Chart` interface and the `LineChart` class will reside on the server side, with no modifications. The client application will need the `Chart` interface in

order to create new remote chart objects, access them, and invoke methods on them. Note that there could be more `Chart` implementations added to the server, and there would be no need to edit any configuration files, or recompile any classes. The only requirement is that the WOX server can locate the necessary classes to execute the implemented methods. However if methods were added to the interface, the client would need the new interface in order to invoke the new methods. A WOX client would use the fragment of code shown below to use the chart application.

```
Chart chart = (Chart)WOXProxy.newObject(serverUrl,classN,args,pol);
double x = Experiment.getResult();
chart.addValue(x);
byte[] image = chart.getImage();
```

The first statement creates a `Chart` object. The `serverUrl` is the URL of the WOX server; `classN` is the class of the object to be created (`stats.LineChart` in this case); `args` are the set of arguments used to construct the object; and `pol` is an integer value that represents whether the real object or a reference to it must be returned from the WOX server (we specify that a remote reference must be returned, since we want the chart objects on the server side).

Requesting a remote reference of a `chart` object is particularly useful in this application that needs to preserve the state of the object, which is modified by adding values to the chart. In cases such as this the ability of WOX to allow clients to create and manipulate objects on the server becomes essential. While it is possible to do the same with SOAP with considerable extra programming effort, as we have shown in the previous subsection, the difference is that WOX actively supports this stateful style of interaction.

The process of creating a remote object continues when the WOX server receives the request, creates the `chart` object and returns the remote reference to the client. When the WOX client receives the remote reference to the new object, it creates a proxy that implements the `Chart` interface. This proxy will be used to make the subsequent method invocations on the remote object. The third and fourth code statements from the code above are adding a new value to the `chart` and getting an image with a line graph.

Since adding values to a `chart` object and getting an image with a graph are method invocations on a `chart` object, they can also be executed through the web browser user interface that WOX provides. Figure 5 shows the user interface provided by WOX to invoke methods on the `chart` object previously created. The `getImage` method has 3 different modes of operation: `xml` to return the array of `byte` serialized in xml, `html` to get the plain array of `byte`, and `image` to get the actual image shown in the web browser. This mode of interaction is only possible in the web browser interface. It actually uses the *image/jpeg MIME* type [18] to decode the array of `byte` when it is sent to the web browser. The possibility to include other MIME types to a WOX server would allow to decode array of bytes into specific formats that can be displayed by a web browser (e.g. audio and image files, text documents, etc.).

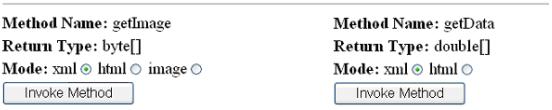

Fig. 5. Web browser interface to invoke methods on a `Chart` object

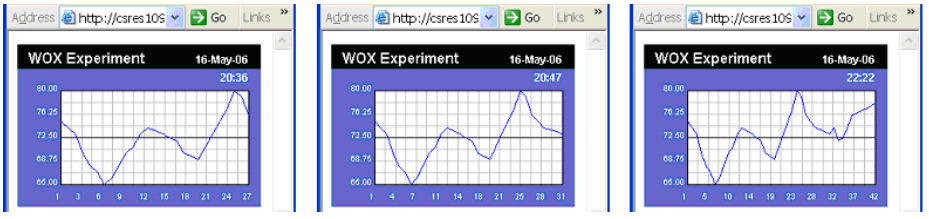

Fig. 6. Method invocations through the web browser

It can also be possible to build a URL with a query string specifying the method to be invoked, the parameters needed, and the mode of operation. Figure 6 shows three different invocations of the `getImage` method using the `image` mode. Clients can access the `chart` object either via a Java client program, through this web browser interface, or simply by building a URL with an appropriate query string.

A fragment of the XML message returned from the WOX server with the array of `byte` is shown below. It is encoded using base64.

```
<array type="byte" length="3023" id="0">
    iVBORw0KGgoAAAANSUhEUgAAAQ4AAAC0CAIAAADq9VVVAAAIg0lEQVR42
    <!-- rest of array omitted -->
</array>
```

WOX serialization process transfers data in XML in a more optimized way than SOAP. By default, SOAP uses an XML element for each element of an array of primitive elements (such as int for example). This means that SOAP-encoded arrays can be over 40 times the size of their binary encoding. The exception to this are `byte` arrays like the one illustrated above, which are encoded efficiently using base-64 (which WOX also uses for byte arrays). Given the speed of modern computers, and the fact that many of us have access to high bandwidth Internet connections, this difference in encoding efficiency might seem unimportant. However, Table 1 emphasizes how significant this difference is, both in time and space usage. For arrays of more than 30,000 int, the SOAP server (Apache Axis) crashed with an out of memory error.

Using WOX to implement the chart application allows other clients to have access to chart objects remotely through their own URI, either by using a Java client program or the web browser user interface. Client applications are also able to create their own `chart` objects, and add new values to existing ones.

Table 1. Time and space usage for passing an array of 20,000 int in WOX and SOAP

Method	Time (ms)	Size (KB)
WOX	80	106
SOAP	3,300	4,200

4 Conclusions and Future Work

In this paper we introduced WOX (Web Objects in XML), which is a web distributed object protocol that allows remote method invocations on web objects, and remote procedure calls on exposed web services. WOX uses HTTP as its transport protocol and XML to encode the messages exchanged between client and server. WOX exposes object references as URIs, inspired by the principles of the Representational State Transfer architectural style. Using URIs in this way allows parameters to be passed, and values returned, either by value or by reference. WOX objects can also be accessed through a web browser interface, from which methods invocations can be executed. We described the WOX architecture, the set of client operations supported, the web browser interface, the format of the XML messages, and the WOX operation modes.

We also presented a case study, in which a state-based chart application is described and exposed over the Internet using three different technologies: RMI, SOAP and WOX. WOX proves to be the most straightforward system for implementing this type of application. Applications like the one presented in this paper, which has some special features such as being accessible remotely over the Internet, maintaining the state of objects, having access to remote objects, storing them in a standard text format (XML), among others, can be implemented using WOX. The possibility to inspect the object through an XML-aware web browser and execute method invocations on web objects via a web browser are also built-in features of WOX. The ease of use to implement this type of applications is another of its advantages over the other technologies discussed.

The limitations or features not included in WOX have also been discussed. They include the language independence given by the XML messages generated by WOX; the security and ownership of objects; the support for asynchronous processes; and the possibility to navigate through web objects. Even in its current state, however, we are already putting WOX to good use, and find it to be a simple, easy to use, and robust protocol.

References

1. Gamma, E., Halm, R., Johnson, R., Vlissides, J.: Design Patterns: elements of reusable object-oriented software. Addison-Wesley, Reading (1995)
2. Common Object Request Broker Architecture (CORBA) Object Management Group (2000), available at `http://www.omg.org`

3. Extensible Markup Language (XML), World Wide Web Consortium, available at http://www.w3.org/TR/REC-xml/
4. Java Technology Sun Microsystems (1994), available at http://java.sun.com
5. Latest SOAP Versions, World Wide Web Consortium (2003), available at http://www.w3.org/TR/soap/
6. Wollrath, A., Waldo, J.: The Java Tutorial, Trail: RMI Sun Microsystems, available at http://java.sun.com/docs/books/tutorial/rmi/
7. Winer, D.: XML-RPC Specification, available at http://www.xmlrpc.com/spec
8. Berners-Lee, T.: Universal Resource Identifiers - Axioms of Web Architecture, World Wide Web Consortium, available at http://www.w3.org/DesignIssues/Axioms.html
9. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures, available at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
10. Costello, R.: Building Web Services the REST Way, xFront, available at http://www.xfront.com/REST-Web-Services.html
11. Prescod, P.: Second Generation of Web Services, available at http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html
12. He, H.: Implementing REST Web Services: Best Practices and Guidelines, available at http://www.xml.com/pub/a/2004/08/11/rest.html
13. Web Services - Axis, available at http://ws.apache.org/axis/ ASF, 2004
14. Jaimez González, C., Lucas, S.: Web Objects in XML: a Web Protocol for Distributed Objects, Technical Report, University of Essex (2005)
15. Jaimez González, C., Lucas, S.: Implementing a Pattern Recognition Application Using RMI, SOAP and WOX, Technical Report, University of Essex (2005)
16. Dynamic proxy classes, Sun Microsystems (1999), available at http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html
17. db4o database (2006), available at http://www.db4objects.com/,db4objects
18. MIME Media Types, Internet Assigned Numbers Authority (1999), available at http://www.iana.org/assignments/media-types/
19. JSON-RPC 1.1 Specification Working Draft (2006), available at http://json-rpc.org/wd/JSON-RPC-1-1-WD-20060807.html
20. Enterprise JavaBeans Technology, Java Platform, Enterprise Edition (Java EE) (2007), available at http://java.sun.com/products/ejb/
21. JBoss Application Server (2007), http://www.jboss.org/products/jbossas