Combining Object-Oriented Design and SOA with Remote Objects over Web Services

Marvin Ferber, Thomas Rauber
Department of Computer Science
University of Bayreuth, Germany
Email: {marvin.ferber,rauber}@uni-bayreuth.de

Sascha Hunold
International Computer Science Institute
Berkeley, California, USA
Email: sascha@icsi.berkeley.edu

***Abstract*—Current approaches of accessing stateful resources via SOAP Web services do not provide a standardized way to use program objects (classes). In this article, we show how the interface of an object-oriented class can be expressed using WSDL. This approach enables a program object to be used in a distributed environment by accessing its Web service interface. The set of Web services associated with a class define a Remote Object over Web Service (ROWS). A ROWS object can be used to facilitate the simultaneous use of the concepts of Service Oriented Architectures (SOA) and Distributed Object Architectures (DOA). In this context, we show how ROWS objects can be utilized in BPEL. Beyond that, we present a ROWS implementation that is suitable to serve as a standalone distributed object middleware. In a case study, we show how classes that were written in Java can be made accessible remotely through the ROWS technology.**

I. Introduction

Distributed object technologies like CORBA [1] or RMI [2] offer sophisticated methods for accessing program modules (e. g., classes in an object-oriented design) over a computer network. However, they were designed for local networks and are hard to apply over the Internet.

Although CORBA provides the Internet Inter-ORB Protocol (IIOP), an adapted version of the CORBA protocol for Inter-ORB communication over the Internet, it is not widely accepted [3]. Instead, Web services are commonly used to share resources and services over the Internet [4]. One reason for the popularity of Web services is its self-describing standardized XML format. But unfortunately, there is no standardized way of accessing objects like object-oriented classes over Web services.

In recent years, the need for standardized communication between business partners over the Internet has increased due to development of complex software products for customers. This often implicates the use of standardized XML-based communication via SOAP Web services [5] in service-oriented environments. Orchestration languages like BPEL [6] have been developed to support the composition of different Web services to derive more complex Web services.

Unfortunately, there is not much support for using objects in a service-oriented environment, e. g., in a service orchestration built upon Web service technology. The simultaneous exploitation of SOA and DOA has not been fully investigated yet. However, the Web service technology is powerful enough to serve as communication middleware in SOA as well as in DOA based applications. The integration of BPEL processes into object-oriented legacy software designs has already been proposed in previous work [7]. We now investigate the more general case of how arbitrary classes, written in object-oriented languages, can be exposed using SOAP Web services and the Web Service Description Language (WSDL) [8].

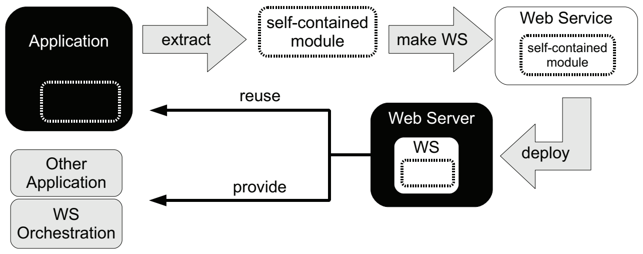
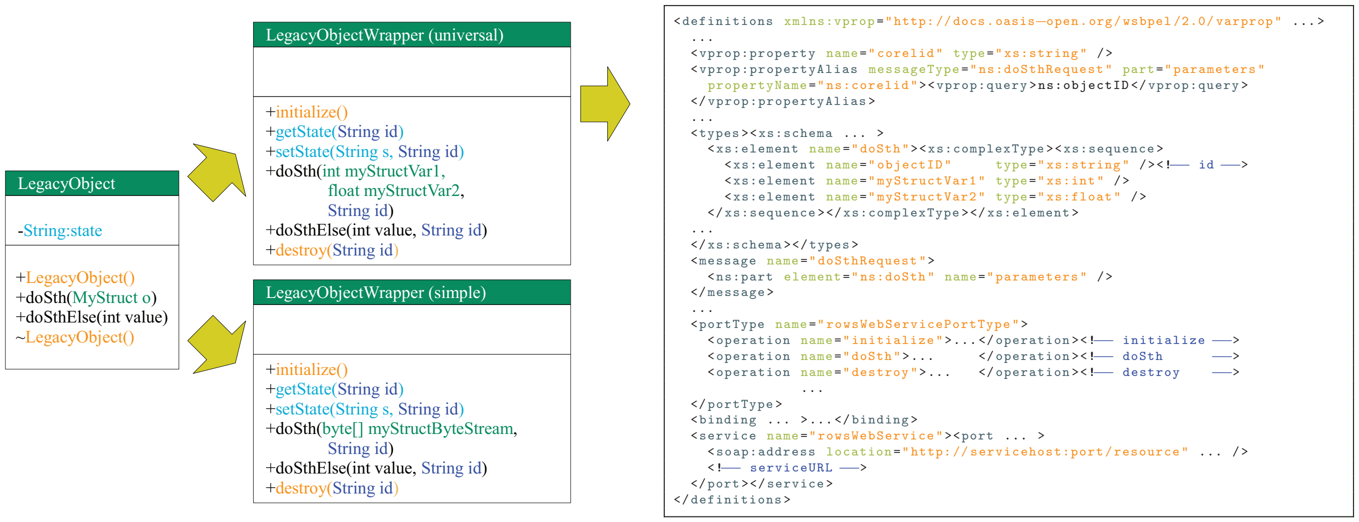


Fig. 1: Objective: Exposing legacy classes as Web services.

In this article, we propose Remote Objects over Web Services (ROWS), which provide a generic way of accessing program objects over a WSDL interface as depicted in Fig. 1. ROWS can help to facilitate a SOA/DOA integration because it enables the composition of stateless Web services in standard BPEL and also the use of stateful objects in the same composition. Such stateful program objects can be legacy software modules that provide sophisticated business logic, which should be reused in a modern service orchestration. Furthermore, ROWS adds attributes to the Web service landscape that enable SOAP Web services to be used as a remote object middleware like CORBA. ROWS is built upon a Web service architecture [9]. Thus, it does not interfere with existing Web service related specifications and can therefore be used in addition to them. As a consequence,


LegacyObject
-String:state
+LegacyObject()
+doSth(MyStruct o)
+doSthElse(int value)
~LegacyObject()
LegacyObjectWrapper (universal)
+initialize()
+getState(String id)
+setState(String s, String id)
+doSth(int myStructVar1, float myStructVar2, String id)
+doSthElse(int value, String id)
+destroy(String id)
LegacyObjectWrapper (simple)
+initialize()
+getState(String id)
+setState(String s, String id)
+doSth(byte[] myStructByteStream, String id)
+doSthElse(int value, String id)
+destroy(String id)

```
<definitions xmlns:vprop="http://docs.oasis—open.org/wsbpel/2.0/varprop" ...>
  ...
  <vprop:property name="corelid" type="xs:string" />
  <vprop:propertyAlias messageType="ns:doSthRequest" part="parameters"
    propertyName="ns:corelid"><vprop:query>ns:objectID</vprop:query>
  </vprop:propertyAlias>
  ...
  <types><xs:schema ... >
    <xs:element name="doSth"><xs:complexType><xs:sequence>
      <xs:element name="objectID"     type="xs:string" /><!—— id ——>
      <xs:element name="myStructVar1" type="xs:int" />
      <xs:element name="myStructVar2" type="xs:float" />
    </xs:sequence></xs:complexType></xs:element>
  ...
  </xs:schema></types>
  <message name="doSthRequest">
    <ns:part element="ns:doSth" name="parameters" />
  </message>
  ...
  <portType name="rowsWebServicePortType">
    <operation name="initialize">...</operation><!—— initialize ——>
    <operation name="doSth">...      </operation><!—— doSth      ——>
    <operation name="destroy">...    </operation><!—— destroy    ——>
            ...
  </portType>
  <binding ... >...</binding>
  <service name="rowsWebService"><port ... >
    <soap:address location="http://servicehost:port/resource" ... />
    <!—— serviceURL ——>
  </port></service>
</definitions>
```

Fig. 2: Example of a legacy object, possible wrapper classes for ID handling and parameter transformation and the corresponding object-oriented WSDL interface.

applications using ROWS are easy to deploy into an existing Web service environment.

The remainder of the article is organized as follows. First we introduce our object-oriented WSDL interface ROWS in Section II. In Section III, the necessary steps to expose an existing class as ROWS compatible Web service are described. It is also shown how a ROWS object can be used in a BPEL process. In Section IV we present an implementation of ROWS for Java on top of Apache Tomcat [10] and Apache Axis [11]. In addition, we give an example of how modern software can be adapted to leverage our ROWS implementation. In Section V, related work is discussed and conclusions are drawn in Section VI.

II. Remote Objects in a Web service Context

In order to expose a class in a Web service context, we have to define its interface using a WDSL document. In previous work we have already investigated the case of how to integrate BPEL processes into object-oriented languages. The proposed BPEL Remote Object (BPELRO) [7] can be used to access stateful BPEL processes from an object-oriented language. In the present article, however, we set out to provide a more general interface that besides BPEL can also handle regular classes that were written in an object-oriented language.

First, we want to name the requirements that have to be taken into account in order to expose a class as a Web service. These requirements are:

- an object needs a unique address to access and to identify an instance,
- an object has an internal state (member variables) that can be manipulated by member methods,
- an object can be created and destroyed,
- data can be passed to and obtained from an object by using method parameters or by directly accessing member variables.

The two-step-approach that shows how an object-oriented WSDL interface can be derived from an ordinary class interface is illustrated in Fig. 2. A class interface can be represented in WSDL by exposing all member methods as operations in the WSDL service section. Access to member variables must be wrapped into getter and setter methods.

Normally, a Web service provides an interface to a stateless resource. It is not possible to access a specific resource behind a standard WSDL Web service interface. To overcome this problem, the Web Service Resource Framework (WSRF) [12] has been introduced. The main idea of the WSRF is to assign a unique identifier (ID) to each resource and add the ID to each Web service call in order to address the underlying resource that should be used to perform the requested operation. ROWS uses a similar mechanism to address underlying objects by adding the ID to the parameter list of all Web service methods. The introduction of the ID parameter can also be seen in Fig. 2. Thus, to reference an object in a Web service context, we need the Web service URL and a unique ID of the object. An object ID is obtained by a central ID factory method, avoiding collisions of object references in the software system.

The methods `initialize()` and `destroy()` are defined to be the standard constructor and destructor of such a Web service object. In contrast to all other ROWS object member methods, the constructor of a ROWS object is always a static method (factory method), because it is invoked without a specific ID parameter. The constructor returns the ID of the newly created ROWS object, which can then be used to utilize the corresponding ROWS object instance. A destructor call with the provided ID will destroy the ROWS object and method invocations can no longer be performed.

All parameters and return values have to be represented in XML. Available basic data types can be taken from the common XML-Schema. Complex data types have to be defined by the developer. Fortunately, there is large tool support for XML data binding of XML-Schema types for many programming languages. Regarding the method parameters and return values, the ROWS framework offers two modes of operation, universal mode and simple mode. An illustration of wrapper objects for both modes is also given in Fig. 2.

In *universal mode*, all parameters in a ROWS interface definition must be XML-Schema types. This is necessary to obtain interoperable messages, that can be exchanged between standard Web service implementations in different programming languages and BPEL. When using the universal mode, client applications that use ROWS objects can be generated easily since an XML data binding for XML-Schema types is available in most programming languages, also in BPEL.

The *simple mode* is not interoperable between modules written in different programming languages. In contrast, this mode offers an easy way of applying ROWS to an existing application because parameters do not have to be expressed in XML. This mode is similar to RMI. Simple parameter types can automatically be converted into XML-Schema types. Complex types like classes are serialized into a byte stream and transferred over SOAP XML messages. This mode can easily be applied with a support for object serialization. However, it is also possible to extend the XML-Schema types by introducing custom XML namespaces. Accordingly, all communication partners have to support this custom XML namespace and have to provide an implementation for the data types of that namespace.

Compared to common object-oriented programming languages and other remote object technologies, ROWS objects are subject to the following restrictions.

- Only object methods can be exposed (no public properties). Access to member variables can only be provided through getter and setter methods.
- Access modifiers are not available. All exposed methods are public by default and can be called from any ROWS client. Thus, private methods of an underlying object are not listed in the corresponding ROWS WSDL document.
- Other method modifiers like `static` are also not supported since such modifiers cannot be assigned to an operation in a standard WSDL document, i. e., the accessing object cannot distinguish if the behavior of the called method is, e. g., static or normal. Therefore, we assume all exposed methods of a ROWS object to be normal member methods. (If a class contains static methods, they first have to be moved to a different class.)
- The ROWS specification does not support the inheritance of object interfaces.

Although most of the restrictions of the ROWS WSDL interface could be solved by introducing a new XML namespace that allows for member method modifiers or interface inheritance, we want to retain conform to the WSDL and BPEL specifications. So, we can benefit from large tool support and Web service technologies that can be used unrestrictedly with our approach. This includes technologies for service discovery (e. g., UDDI), service requirements (e. g., WS-Policy), service reliability (e. g., WS-Reliable Messaging), service security (e. g., WS-Security) and others [13].

III. ROWS as Middleware for SOA/DOA Integration

Integrating legacy software modules or providing services written in object-oriented languages is a hot topic for modern software development. It is often desired to reuse a legacy module in a modern service orchestration or to make an existing software capable of being executed in a distributed environment. Therefore, we show in this section how a legacy software module can be exposed as a ROWS object and also how this ROWS object can be inserted into a BPEL process. The goal is to gain an easier integration of object-oriented legacy modules and modern service-oriented software.

A. Legacy Modules as ROWS Objects

In order to expose a component of a software as a self-contained ROWS object, the structure of the object-oriented software design has to be investigated regarding class dependencies. To identify classes of a software system that can be exposed as a ROWS object, source code patterns can be used to find suitable candidate classes [14]. Tools exists to expose all methods of a class as Web service, e.g., the Apache Axis tools. However, a wrapper class on server side is needed to provide a ROWS compatible class interface and to connect the ROWS interface to the real underlying class object. This wrapper class is referred to as server-side ROWS wrapper class. The interface of the server-side ROWS wrapper class can then be exposed as ROWS object by the mentioned tools, including a ROWS compatible WSDL interface. By using the ROWS approach, the original stateful behavior of the class object is preserved while the object's interface is exposed as Web service. In

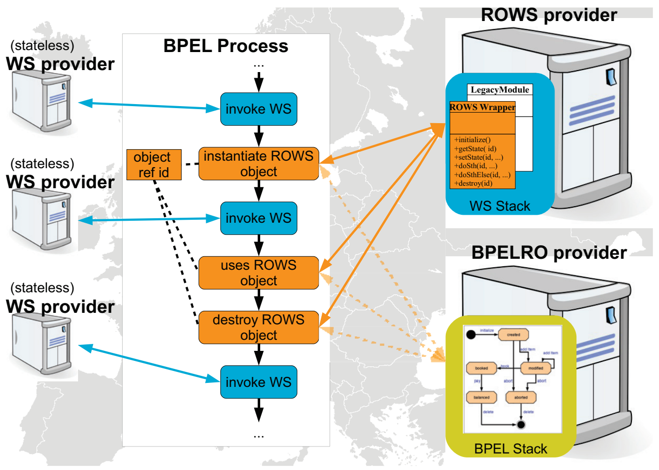


Fig. 3: Fragment of a BPEL process using stateless Web services and ROWS objects. ROWS objects and BPEL Remote Objects can be used similarly.

cases in which the source code of a legacy module is not available, it can still be exposed as ROWS Web service, because the legacy object usually does not need to be modified.

We now show how such a wrapper class can be generated. The wrapper class serves as proxy object that encapsulates all functionality required to handle the object and its interface. This proxy object (wrapper class) is responsible for the following tasks:

- ID handling,
- construction and destruction of the underlying object, and
- handling of parameters and return values according to the ROWS mode of operation (universal/simple).

The constructor of a server-side ROWS wrapper class has to implement two activities. First, a unique object ID has to be obtained from an ID factory. Second, a new instance of the underlying class has to be created on the particular server. Afterwards, the object ID is assigned to the newly created object. Finally, the object ID is returned to the calling object as the constructor's return value. As stated in Section II, a ROWS provider must be able to associate an object ID to the corresponding underlying program object instance. Therefore, a ROWS provider holds an ID database and has a generator to create unique object IDs.

All member methods of a server-side ROWS wrapper class follow the same pattern, which is similar to a proxy. When invoked, the wrapper class method performs a look up of the given ID parameter in the object database and delegates the method call to the associated underlying object. Since the ID parameter is only part of the ROWS communication, it is removed from the list of parameters that are passed to underlying object.

Method parameters and return values of the ROWS object need to be in XML format in order to be transferred via SOAP messages. Depending on the ROWS mode of operation (universal mode or simple mode), conversions of parameters between the underlying class object and the ROWS interface parameters in XML format may be necessary. They can be processed inside the server-side ROWS wrapper class. Such conversions are, e. g., serialization of parameters in simple mode or the transformation of data structures into a custom XML structure for use in universal mode.

The non-blocking methods of the underlying object can easily be associated with synchronous WSDL operations. Blocking methods have to be associated with asynchronous Web service operations to avoid possible network timeouts. To retain compatible with BPEL, asynchronous operations have to be split into a request message that is sent to the server and a response message that is sent from server to the client after the request has been processed. Between the two messages, a persistent connection is not necessary, which avoids a network timeout. However, this approach requires a backward channel from the server to the client, which is not always available due to firewall restrictions. The handling of such an asynchronous behavior can also be realized inside the Wrapper classes on client and server-side and is therefore transparent to the underlying class and the original program.

B. Using ROWS Objects

In this section, we assume that a legacy software module has already been exposed as a ROWS Web service. ROWS objects can be used in distributed object applications as well as in BPEL Web service orchestrations.

The simultaneous use of a ROWS object and an ordinary stateless Web services is illustrated in Fig. 3. The figure shows a fragment of a BPEL process that contains a sequence of Web service invocations. In contrast to the stateless invocations, an explicit object reference is needed in order to perform stateful invocations on the same object instance. The object ID, which is part of the ROWS object reference, can be stored in BPEL variable. Fig. 4 shows such an example for BPEL. The object ID is obtained as return value from the constructor call `initialize()`. Furthermore, the object ID is present in the parameter list of all other ROWS object methods in the example (`doSth()`, `destroy()`). We recall that the usage of ROWS objects in BPEL implies that the universal mode of operation of ROWS is used. In this mode, BPELROs and ROWS share the same interface definition and can therefore be substituted with each other. This is shown in Fig. 3: the BPEL Remote Object (at the bottom) can be used as an alternative to the

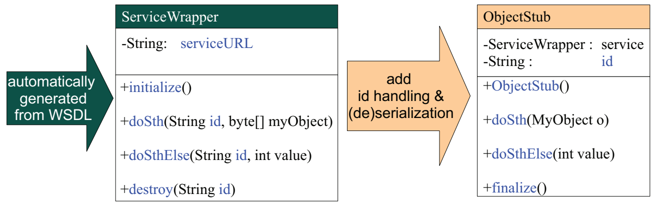
```
<bpel:process ... xmlns:rows="urn:de:edu:bt:rows">
 <bpel:import ... namespace="urn:de:edu:bt:rows"/>
 ...
 <bpel:partnerLinks>
  <bpel:partnerLink name="rowsWebService"
       partnerLinkType="rows:JavaSchedulerWrapperPLT" ... />
  ...
 </bpel:partnerLinks>
 <bpel:variables> ...
  <bpel:variable type="xsd:string" name="rowsObjectID"/> <!-- object id -->
  ...
 </bpel:variables>
 <bpel:sequence> ...
  <bpel:invoke operation="initialize" outputVariable="initializeReturn"
      partnerLink="rowsWebService" ... ></bpel:invoke> <!-- initialize -->

  <bpel:assign>
   <bpel:copy>
    <bpel:from>$initializeReturn.parameters/rows:objectID</bpel:from>
    <bpel:to>$rowsObjectID</bpel:to>
   </bpel:copy>
  </bpel:assign>
  ...
  <bpel:assign>
   <bpel:copy>
    <bpel:from>$rowsObjectID</bpel:from>
    <bpel:to>$doSthRequest.parameters/rows:objectID</bpel:to>
   </bpel:copy>
  </bpel:assign>
  <bpel:invoke inputVariable="doSthRequest" operation="doSth"
        partnerLink="rowsWebService" ... ></bpel:invoke> <!-- doSth -->
  ...
  <bpel:invoke operation="destroy" partnerLink="rowsWebService" ... >
  </bpel:invoke>  <!-- destroy -->
  ...
 </bpel:sequence>
</bpel:process>
```

Fig. 4: Example of a BPEL process that utilizes the ROWS object from Fig. 2.

ROWS object (above). This uniform modeling allows to build software on top of the concepts of SOA and DOA using BPEL and Web services. Moreover, it allows for a flexible selection of the underlying technology.

Tool support is available to automatically derive a (language-specific) class interface from a WSDL document in order to access a stateless Web service from an object-oriented software design (e. g., Apache Axis Tools). However, when using ROWS objects from object-oriented applications, e. g., written in Java, a client side wrapper is necessary to create a stateful class interface from the ROWS WSDL. This wrapper class is referred to as client-side ROWS wrapper class. An illustration how such a wrapper class can be created is given in Fig. 5. The client-side ROWS wrapper class takes care of the ID handling, construction and destruction of a ROWS object on server-side and the parameter processing. In universal mode, the procedure of creating such wrapper classes is the same as for BPELROs and has already been introduced in [7]. However, in simple mode, the client-side ROWS wrapper class needs to take care of object de/serialization and additional tasks that depend on the specific implementation of the ROWS framework.

In general, the creation of a new ROWS object instance is performed in two steps. To access a particular object, the endpoint of the Web service that hosts the ROWS object is obtained. In a worldwide context a suitable endpoint can be obtained from a UDDI server. In local networks it is also possible to use WS-Discovery methods to retrieve an endpoint.



automatically generated from WSDL
ServiceWrapper
-String: serviceURL
+initialize()
+doSth(String id, byte[] myObject)
+doSthElse(String id, int value)
+destroy(String id)
add id handling & (de)serialization
ObjectStub
-ServiceWrapper : service
-String : id
+ObjectStub()
+doSth(MyObject o)
+doSthElse(int value)
+finalize()

Fig. 5: Example of a ROWS wrapper class on client side for the Java implementation using Apache Axis.

Having obtained the endpoint of the desired ROWS Web service, we can instantiate the remote object by calling the object's `initialize()` method. The constructor method returns an object ID of the newly created ROWS object. Now operations can be performed on this ROWS object. Like BPELROs, ROWS objects can be shared among communication partners, because the textual ROWS object reference can be copied and reassigned to another client wrapper class.

IV. ROWS Middleware Implementation

The ROWS framework should provide a general XML-based middleware that supports SOA and DOA concepts and facilitates the collaboration of object-oriented software and services of business partners via Web services. In this section, we compare the ROWS framework to CORBA, which is an already established middleware standard. The advantage of ROWS is that data of messages do not have be converted between different technologies and can be processed directly. In addition, security, resource discovery, and other mechanisms only have to be provided for one technology, which can reduce the maintenance costs of a software system.

Since ROWS uses Web service technology, it can utilize the entire stack of Web service technologies. CORBA and the Web service technology were compared in [3] and [15], concluding that both technologies offer similar features. The most significant characteristics of CORBA and ROWS are highlighted in Table I. The Web service technology offers similar features as CORBA, and our ROWS specification extends SOAP Web services to provide remote objects via Web services. So, ROWS can be used as a replacement for CORBA in many cases.

It was shown in Section III how a program object (class) can be accessed over a network via Web services using ROWS. In this section, we present an implementation of ROWS in simple operation mode, which can be used to share Java objects in a distributed application. In this scenario, the implementation of ROWS serves as a communication middleware like CORBA.

A. ROWS Implementation for Java

Since Java provides automatic serialization and deserialization, Java objects can easily be transferred over

Table I: ROWS capabilities compared to CORBA.

	CORBA	ROWS
Object Definition Language (inheritance support)	IDL (yes)	WSDL XML-Schema (no)
Naming, Discovery	Naming Service, Interface Repository, Trader service	UDDI, WS-Discovery
Location identifier	IOR, URL	URL + ID
Authentication and encryption	Security Service	WS-*Security
Network transport	GIOP, IIOP/HTTP	SOAP/HTTP

a network. We implemented the ROWS middleware for Java on top of Apache Tomcat and Apache Axis2. The goal was to obtain a framework that helps to extract classes from an application, to generate Web service compatible interfaces of these classes, and to move these classes to a remote server. The necessary communication path between the two application parts (client and server) is realized with the ROWS compatible Web service middleware.

In the first step, the application is divided into two independent parts, which is described in more detail in Section IV-B. After that, the server-side object is exposed as ROWS object as described in Section III (server-side ROWS wrapper class). Because we use ROWS in simple mode, it is not necessary to convert method parameters and return values. Instead, complex parameters are serialized into a byte stream, transferred via SOAP messages, and deserialized back into a usable Java object on the other side of the communication. The (de)serialization is performed by the client-side ROWS wrapper class and the server-side ROWS wrapper class. To deserialize the byte stream back into a Java object properly, the type (class) of the serialized object must be know on both sides of the communication. An overview of our ROWS middleware implementation for Java is given in Fig. 6. In this figure the yellow parts represent existing technologies and the red parts represent the ROWS extensions. The current implementation of the ROWS middleware uses an Apache Tomcat application server that holds the ID database and the ID generator. The ID database stores pairs of ID and corresponding Java object. The Apache Axis libraries handle the SOAP and network stack access as well as the XML data binding and marshaling. The transformation steps of the software are performed using the Eclipse IDE [16].

The server-side ROWS wrapper class and client-side ROWS wrapper class are named ID wrapper in Fig. 6. The classes which hide the SOAP communication with the Apache Axis/Axis2 libraries can be created automatically by an Eclipse IDE plugin. The process of obtaining a server side ROWS object has already been described in Section III. The Eclipse IDE also provides a tool for exporting the server-side module as a Tomcat-compatible Web Application Archive (WAR), which can be easily deployed to a target server.

On client side, an ID wrapper class is necessary to provide an object stub that replaces the original class object by providing the same interface. An example of an automatically generated Axis wrapper and a corresponding ID wrapper class is shown in Fig. 5. Since Java provides an implicit garbage collection mechanism, there is no explicit destructor that can be called to destroy a standard Java object. However, by overwriting the `java.lang.Object` method `finalize()` of the client-side ROWS wrapper class, the destruction of the corresponding server side ROWS object is realized. The server side ROWS object is destroyed when the Java object stub on client side is destroyed by the garbage collector.

After the original class has been replaced by the specific client-side ROWS wrapper class and the original class has been exposed as a ROWS object on a remote server, the application can still be executed as before. If an instantiation of the externalized object is requested on client side, a server side object is created on the ROWS provider by the ID wrapper. All method invocations are delegated to the server side object transparently.

B. Case Study: Distributed Sunflow

In the process of software modernization it is often desirable to divide a software into self-contained modules that collaborate with each other over a computer network. In particular, the externalization of a module can be useful for performance or security reasons, or simply to provide it to business partners.

In our case study, we aimed at improving the performance of the Open Source Java desktop application Sunflow [17] by distributing computations to several modern multi-core servers via network. The original Sunflow application is a ray tracer comprised of a multi-threaded renderer, a scene parser, data structures representing the scene, a graphical user interface, and additional logical. Instead of using only one machine, the goal was to exploit the computational power of multiple servers by relocating computational intensive parts over the ROWS middleware. We relocated the renderer module of the software to a high performance compute server, which is illustrated in Fig. 6. For this experiment, ROWS uses the simple mode of operation to expose the renderer interface as ROWS object.

When Sunflow is started it parses a scene file and reads it into an internal data structure. Afterwards, a renderer generates the output image for this data structure.

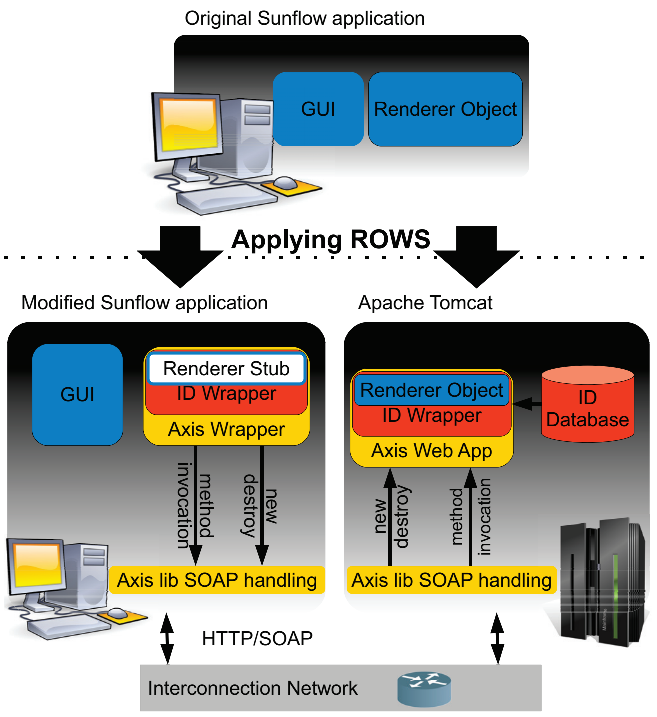


Fig. 6: Example of ROWS in simple mode applied to Sunflow. The renderer class is relocated to a more powerful server and reconnected to the original application using our ROWS implementation.

Fortunately, Sunflow already includes a multi-threaded renderer that can utilize multiple cores on a shared-memory machine. Because the renderer cannot access the scene data structure anymore from remote side, the application was modified to include a copy of the scene data structure in the renderer object as well. Such class dependencies can be discovered by sophisticated tools such as TransFormr [14] or Eclipse IDE.

In this experiment, we used an AMD Opteron 244 (2 cores) workstation with 4 GB of memory and an Intel XEON E7330 (16 cores) server machine with 16 GB of memory. The modified version of Sunflow (renderer running on server machine only) achieved a speedup of up to 11 compared to the original unmodified workstation application (renderer running on the workstation only). This maximum speedup was achieved for both scenes from the original Sunflow examples (`aliens_shiny` and `gumbo_and_teapot`). We used a granularity of 32x32 pixels for the rendering tasks and a 1 Gbit Ethernet connection between the server machine and the workstation.

The experiments have shown that we can obtain a good speedup of the distributed application. Of course, speedup and efficiency depend on the particular scene, the network topology, and the properties of the machines used in this experiment.

In this case study we have shown how ROWS can be used to merge the concept of DOA with Web service technology. In this context, ROWS is comparable to CORBA. However, the resulting performance and the necessary steps to adapt the software to make use of ROWS depend on the specific application. The current implementation of ROWS for Java offers a homogeneous communication middleware for distributed objects since Java runs on nearly all hardware platforms and Web service technology is flexible enough to serve as a reliable communication middleware.

V. Related Work

Researchers and developers often argue whether Web services are a variation of the distributed object concept or not [18], [19]. Also, Web services are said to be a successful technology for implementing SOA concepts [4]. Since Web service technology is just a standardized way of exchanging XML messages between communication partners, it is flexible enough to serve as a communication middleware.

Furthermore, especially in the case of legacy software modernization the utilization of different design concepts is desirable. In [20] the utilization of CORBA and Web services is favored. Other approaches of legacy software modernization propose methods for adapting a legacy application to a SOA by reengineering the source code to use legacy program modules in a SOA environment [21].

Common technologies for distributed computing in Java (CORBA, EJB, and Web services) have been evaluated with respect to performance and modeling issues in [22]. In general, CORBA offers a better average performance than SOAP when accessing and transferring data between remote objects [23]. This is due to the large and complex software stack of Web services. Usually payload data in CORBA consumes less network traffic than the same data in XML format. XML has more overhead due to XML tags and binary data can only be transferred in base64 format, which inflates the binary data by 30%. Thus, since ROWS uses Web service technology for communication it might not be the best choice for very data-intensive use cases.

To identify modules of an object-oriented legacy system that can be relocated in order to transform the legacy application into a distributed application, data mining techniques can be used [24]. Our approach is based on analyzing the source code, which was introduced as part of the TransFormr toolset in [14]. For the described ROWS middleware approach for Java, the pattern based externalization process has been modified in order to apply to the presented Sunflow application. The externalization of existing source code parts of a Java application to remote compute resources has been targeted recently in [25]. Although they do not provide a practical evaluation, they propose a framework based on Web service technology, summarize requirements for the granularity of source code parts that should be externalized, and present a round robin load balancing strategy for homogeneous environments.

VI. Conclusions

In this article we proposed ROWS, a framework that helps to access object-oriented classes using a Web service interface. The main advantage of ROWS is an easy integration of stateful objects with Web service technologies. In addition, our approach is compatible with BPEL. Thus, it helps developers to combine SOA and DOA based software models in the business domain.

In the domain of distributed computing our approach is comparable to CORBA. It has been shown how ROWS can distribute computations over a network. In a case study, the ray tracing software Sunflow was modified (ROWS enabled) so that it could be executed on a distributed platform with ROWS as communication middleware.

Another advantage of ROWS is that it can possibly lead to a more homogeneous software landscape as it only requires Web services and not additional remote object technologies.

References

[1] “Common Object Request Broker Architecture.” [Online]. Available: http://www.corba.org/

[2] “Remote Method Invocation.” [Online]. Available: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[3] S. Baker, “Web Services and CORBA,” in *Proc. of On the Move to Meaningful Internet Systems 2002*, ser. Lecture Notes In Computer Science, vol. 2519. Springer, 2002, pp. 618 – 632.

[4] E. Newcomer and G. Lomow, *Understanding SOA with Web Services*. Addison-Wesley Professional, 2004.

[5] “Simple Object Access Protocol (SOAP) 1.1,” 2000. [Online]. Available: http://www.w3.org/TR/soap/

[6] “WS-BPEL 2.0 Specification,” 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[7] M. Ferber, S. Hunold, and T. Rauber, “BPEL Remote Objects: Integrating BPEL Processes into Object-Oriented Applications,” in *Proc. of the 7th IEEE Int. Conference on Services Computing (SCC 2010)*, 2010, pp. 33–40.

[8] “Web Services Description Language,” 2001. [Online]. Available: http://www.w3.org/TR/wsdl/

[9] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, “Web Services Architecture,” 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/

[10] “Apache Tomcat Application Server.” [Online]. Available: http://tomcat.apache.org/

[11] “Apache Axis2.” [Online]. Available: http://ws.apache.org/axis2/

[12] “Web Service Resource Framework (WSRF) v1.2,” 2006. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf

[13] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 2005.

[14] S. Hunold, M. Korch, B. Krellner, T. Rauber, T. Reichel, and G. Rünger, “Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing,” in *Proc. of the 32nd IEEE Int. Computer Software and Applications Conf. (COMPSAC 2008)*, 2008, pp. 303–310.

[15] A. Gokhale, B. Kumar, and A. Sahuguet, “Reinventing the Wheel? CORBA vs. Web Services,” in *Proc. of the 11th Int. World Wide Web Conference (WWW 2002)*, 2002.

[16] “Eclipse Integrated Development Environment for Java version 3.6,” 2009. [Online]. Available: http://www.eclipse.org/

[17] “SunFlow Render System version 0.07.2,” 2007. [Online]. Available: http://sunflow.sourceforge.net/

[18] K. Birman, “Like it or not, web services are distributed objects,” *Communications of the ACM*, vol. 47, Issue 12, pp. 60 – 62, 2004.

[19] W. Vogels, “Web Services Are Not Distributed Objects,” *IEEE Internet Computing*, vol. 7, Issue 6, pp. 59 – 66, 2003.

[20] S. Baker and S. Dobson, “Comparing Service-Oriented and Distributed Object Architectures,” in *Proc. of the Int. Symp. on Distributed Objects and Applications*, ser. Lecture Notes in Computer Science, vol. 3760. Springer, 2005, pp. 631–645.

[21] S. Chung, J. B. C. An, and S. Davalos, “Service-Oriented Software Reengineering: SoSR,” in *Proc. of the 40th Hawaii Int. Conf. on System Sciences*. IEEE Computer Society, 2007, p. 172.

[22] D. Vassilopoulos, T. Pilioura, and A. Tsalgatidou, “Distributed technologies CORBA, Enterprise JavaBeans, Web services: a comparative presentation,” in *Proc. of the 14th Euromicro Int. Conf. on Parallel, Distributed, and Network-Based Processing (PDP 2006)*, 2006, p. 5.

[23] R. Elfwing, U. Paulsson, and L. Lundberg, “Performance of SOAP in Web Service Environment Compared to CORBA,” in *Proc. of the 9th Asia-Pacific Software Engineering Conf.* IEEE Computer Society, 2002, p. 84.

[24] M. A. Serrano, D. L. Carver, and C. M. de Oca, “Mapping Object-Oriented Systems to Distributed Systems Using Data Mining Techniques,” in *Proc. of the 13th Int. Conf. on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, vol. 1821/2000. Springer, 2000, pp. 79–84.

[25] T. Noda, H. Mine, N. Fujimoto, and K. Hagihara, “A Parallel Computing Framework for Nonexperts of Computers: Easy Installation, Programming and Execution of Master–Worker Applications Using Spare Computing Power of PCs,” *Frontiers of Computational Science*, p. 305–308, 2007.