The Remote Access Generator Engine

Remote Access via Automatic Code Generation (for a Java Web Services Environment)

eingereicht von
Robert Neumayer

MAGISTERARBEIT

zur Erlangung des akademischen Grades:
Magister rerum socialium oeconomicarumque
Magister der Sozial- und Wirtschaftswissenschaften
(Mag. rer. soc. oec)

Fakultät für Informatik
Universität Wien

Studienrichtung: Wirtschaftsinformatik

Begutachter:
Univ. Prof. Dr.techn. Wolfgang Klas
Betreuender Assistent:
Dipl.-Inf. Dr.techn. Gerd Utz Westermann

Wien, im Juni 2005

Zusammenfassung

Remote-Zugriff auf bestehende Applikationen wird in Zeiten von Internet und mobilen Endgeräten für die Softwareentwicklung immer wichtiger. Zumeist geht es hierbei um bestehende objektorientierte Softwaresysteme, wobei hier zusätzlich zu den herkömmlichen Schnittstellen nun auch Schnittstellen für den Remote-Zugriff bereitgestellt werden sollen. Diese Arbeit stellt eine Plattform für die automatische Generierung von Remote-Interfaces für bestehende Softwareprojekte vor. Basierend auf serverseitigem Object-Caching wird eine Methode vorgestellt, um Objekte am Server zu belassen und fernzusteuern. Weiters wird die Implementierung eines solchen Systems basierend auf Java vorgestellt. Die Remote Access Generator Engine (RAGE) realisiert die automatische Generierung von SOAP- (hier gleichbedeutend mit Web service) und RMI- Schnittstellen für bestehende Java-Applikationen, wobei in dieser Arbeit besonders auf bestehende Interoperabilitätsprobleme einzugehen versucht wird. Außerdem werden die Möglichkeiten zur Nutzung von Choreographiesprachen erklärt, um bestehende Web service Schnittstellen genauer zu beschreiben. Die vorgestellte Plattform ist modular aufgebaut, um eine Erweiterung um zusätzliche Remote-Access-Mechanismen wie CORBA und RMI zu ermöglichen.

Abstract

Many standards for remote access to applications have emerged in the last years, as remote access has become an important aspect of almost any software system. Nowadays the main problem in this context is to provide simple interfaces to complex (object-oriented) APIs. This work proposes a modular platform for automatic generation of remote interfaces for existing software packages based on server-side encapsulation of objects and a remote control mechanism for those encapsulated objects. Furthermore the Remote Access Generator Engine (RAGE), a sample implementation for such a tool based on Java and aimed at generation of remote interfaces for Java applications will be introduced. RAGE includes the generation of SOAP (Web service) and RMI interfaces. This work will concentrate on the automatic generation of Web service interfaces for existing Java applications, particularly on involved interoperability issues. Moreover a simple approach to Web services choreography configuration of existing Web service interfaces will be introduced. Additionally RAGE is designed as a modular architecture and hence ensures easy extensibility towards different methods for remote access like RMI or CORBA.

Contents

Chapter 1

Introduction

Distributed systems have become increasingly important throughout the last years. Nowadays application developers are far more often concerned with data or resource sharing and interaction in distributed environments. Those needs are particularly driven by the Internet's ubiquitousness and the rising popularity of mobile systems as well as the increasing availability of widespread broadband access. Therefore, the development of distributed systems or applications that are remotely accessible has become an important issue in software development.

1.1 Remote Access

Almost any software system must be accessible from the outside, realized by different paradigms and implementations. This section introduces some of the main ideas and concepts to explain the situation software developers are facing today.

1.1.1 Requirements and Implications

As far as the object-oriented world is concerned, remote access means access to remote objects and their methods (or procedures). The most important concept here is the remote procedure call (RPC). RPC is initiated by the client who sends a message to the server in order to execute a certain procedure and returning a result. Many different approaches to remotely accessing objects have been developed, including:

- CORBA, (Common Object Request Broker Architecture) [35]
- RMI, (Remote Object Invocation) [23]
- DCOM, (Distributed Component Object Model)[32]

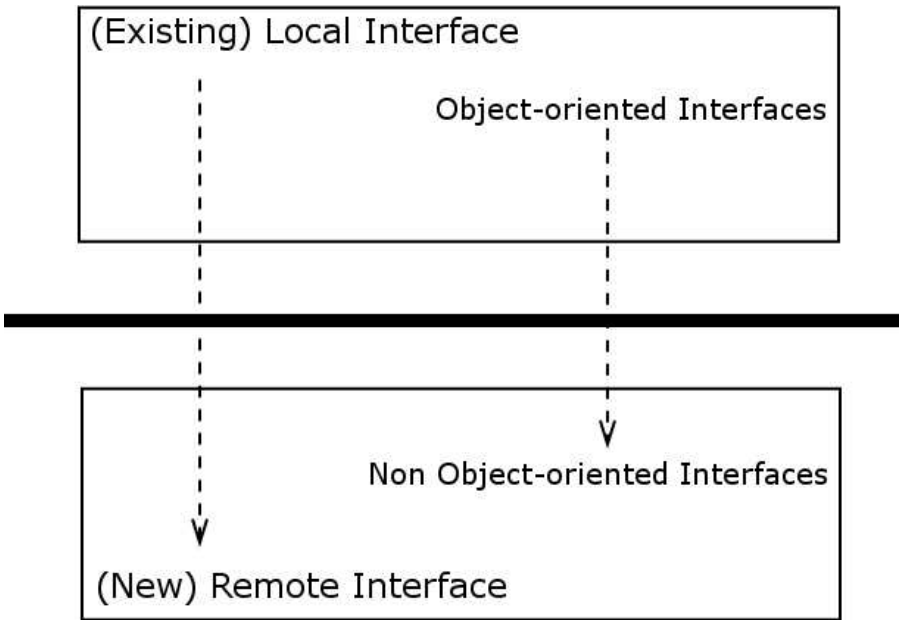

Figure 1.1: Migration to a Remote Interface

Each of those represents a different approach to remote access in object-oriented environments, but are not compatible to each other. However, application programmers are increasingly faced with the need to support their software with remote access, using one or even more of those techniques, particularly if clients implemented in different programming languages should be able to access their application. Whereas designing a new software system that has to support one of those techniques is manageable, the process becomes more complicated when more of those approaches have to be supported. The main problem is to provide remote access to existing application interfaces, especially when object-oriented applications are involved. Whereas local access is not much of a problem, it is often more complicated to realize adequate remote access.

The main issue in coding remote interfaces is the migration from object-oriented to simple interfaces, because the usage of complex objects is rather difficult across the Internet and almost impossible across different programming languages. This involves a change from complex to simple parameters and return types. As shown in Figure 1.1 all complex elements have to be substituted by simple ones when a complex interface is replaced by a simple one. The need for simple interfaces is even more compelling if cross-platform or cross-language interfaces are required, which are getting increasingly popular. Particularly Web services are often used to build large and scalable applications consisting of services implemented in different programming languages.

This makes it even more necessary to provide interoperable interfaces to the participating applications.

When remote interfaces are needed for existing complex APIs, a developer has two main options:

- Via complex parameters
  - Ignoring cross-platform and cross-language needs
- Via wrappers
  - Considering cross-platform and cross-language needs
  - At the cost of a greater programming expenditure

The solution using complex parameters is not feasible because of the importance of cross-platform and cross-language programming. On the other hand, the wrapper solution is not an ideal one either, it provides interfaces with simple parameters and return types and therefore is suitable for a cross-platform and cross-language scenario. The disadvantages of this approach becomes particularly obvious in the development stage of an application when interfaces change repeatedly and they have to be synchronized with the actual application. Writing wrapper classes means to manually program simple interfaces to all needed parts of an existing API. The more often the interfaces of the existing application change, the more programming effort is needed to synchronize the resulting simple interface with the complex API.

1.1.2 Remote Interfaces for Existing Projects

Figure 1.2 illustrates the relation of different types of interfaces with the actual application. A typical interface class used to provide remote access includes functionality to access all of the needed parts of a project. Therefore such an interface would have to to be implemented by the programmer of the application and has to correlate with its use cases and the need for automatic generation of those interfaces is given.

1.2 Contribution

For the above reasons this work proposes a framework for the automatic generation of remote interfaces that only rely on simple datatypes and hence allows access via multiple platforms and programming languages. Furthermore the proposed solution (RAGE, the Remote Interface Generator [16], a tool for the automatic generation of low-level Java interfaces for existing (Java-) packages, principally supports several techniques like RMI and CORBA where this work concentrates on generating Web service interfaces (see [15] for RMI and CORBA support). The main goals that were pursued through the development process are:

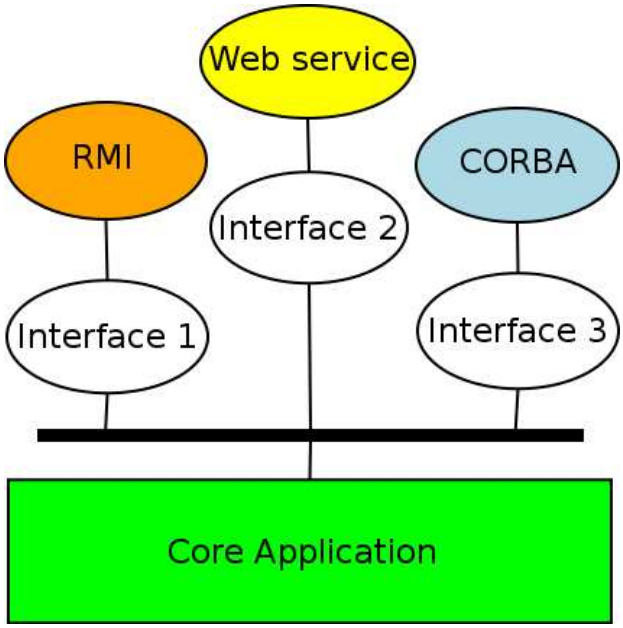

Figure 1.2: Building Blocks of an Application and its Interfaces

- Realizing access to even very complex APIs via any programming language
- Generation of more complex Web services using a Web services choreography language

The main benefits of the presented sample implementation is the fast and easy generation of platform- and language-independent remote interfaces to existing Java packages. Application development could be a field of application, because of the frequent changes to interfaces during this stage (when the manual synchronization between wrappers and interfaces seems to be too costly). RAGE furthermore supports the generation of more complex interfaces making use of a Web services choreography language, because of the fine granularity of the generated low level interface classes from the first transformation step.

1.3 Thesis Structure

Chapter 2 explains the relevance of Web services as well as the main principle of their paradigm. The interoperability issues arising in a Java Web service scenario and its common solutions are pointed out in Chapter 3. Chapter 4 introduces an overall concept of referencing of remote objects and in how far that approach supports the development of Web services. Chapter 5 explains

the main concepts of the implementation RAGE and evaluates its problems. After that Chapter 6 covers Web service choreography and its suitability to an extension of the proposed model. Chapter 7 describes the implementation of this extension. Finally Chapter 8 proposes further improvements to RAGE and its concepts.

Chapter 2

The Web Services Architecture

The fundamental concepts of the Web service architecture, particularly its advantages and disadvantages compared to other approaches are the main topics of this chapter. The contents of this chapter are important to understand the remainder of this work, however, it is an overview, the reader may consult [8, 3, 9] or other W3C pages for a comprehensive description of the Web services architecture.

2.1 Web Services and their Impact on Remote Access

The latest technique to realize remote access is Web services. As they are getting more and more important, the rapid development of Web service interfaces is the main motivation of this work. The idea of a loosely coupled, service oriented architecture offers many new opportunities to software engineers as all software systems can be understood as services, i.e. applications taking part in greater collaborations. Therefore the user (or developer) does not mind:

- Where the service is running,
- How it is implemented,
- And who is responsible for it.

The loosely coupled architecture also known as grid architecture or simply the grid [19] is basically meant as a distributed infrastructure that allows ad hoc resource sharing and therefore the building of large and scalable applications using many different and distributed services. XML [44] is used exclusively for the exchange of data between the included services.

> Definition: A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

This definition of a Web service is taken from the Web service architecture note [3] of the W3 Consortium [8]. Its Protocol Working Group [9] introduces the terms SOAP (Simple Object Access Protocol) [10, 11, 12] and WSDL (Web Services Description Language) [33] which shall be explained in the remainder of this chapter.

> SOAP Version 1.2 (SOAP) is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols.

Hence every sort of communications inside the Web services world relies on SOAP, in particular its XML-representation of different datatypes.

2.1.1 SOAP for Remote Procedure Calls (RPCs)

SOAP defines message structures for remote procedure invocations and responses in XML format. The information needed for such a remote call is:

- Address of the target SOAP node (endpoint).
- The procedure or method name.
- The values of any arguments passed to the remote procedure.
- Values for properties used for the binding (i.e. HTTP GET or POST).

A sample SOAP RPC request is shown in Listing 2.1. It describes a sample SOAP RPC request for the remote procedure *chargeReservation* of the well-known travel Web service context. Note that the encoding style is not bound to soap-encoding as used in this example. The method *chargeReservation* is called for the arguments *reservation* and *creditCard* which is a *struct*, modeled after similar concepts in common programming languages.

Listing 2.1: SOAP RPC Request

```
<?xml version='1.0' ?>
<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  <env:Header>
```

```
    <t:transaction
    xmlns:t="http://thirdparty.example.org/transaction"
    env:encodingStyle="http://example.com/encoding"
    env:mustUnderstand="true" >5</t:transaction>
 </env:Header>
 <env:Body>
  <m:chargeReservation
     env:encodingStyle=
     "http://www.w3.org/2003/05/soap-encoding"
     xmlns:m="http://travelcompany.example.org/">
    <m:reservation
       xmlns:m=
       "http://travelcompany.example.org/reservation">
     <m:code>FT35ZBQ</m:code>
    </m:reservation>
    <o:creditCard
       xmlns:o="http://mycompany.example.com/financial">
     <n:name
       xmlns:n="http://mycompany.example.com/employees">
        Firstname  Lastname
     </n:name>
     <o:number>123456789099999</o:number>
     <o:expiration>2005-02</o:expiration>
    </o:creditCard>
  </m:chargeReservation>
 </env:Body>
</env:Envelope>
```

This SOAP request would call the specified procedure using the given parameters. A request to this response is constructed in an analogous way, carrying the return type of this method. In the context of this work the term *Web service* relates to any interface to a software system that relies on the SOAP protocol and is described by WSDL, the Web Service Description Language, introduced in the next section.

2.1.2 WSDL, the Web Services Description Language

SOAP covers all communication issues related to Web services, in contrast to WSDL, that is used for the static description of their interfaces.

> Web Services Description Language (WSDL) provides a model and an XML format for describing Web services. WSDL enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as 'how' and 'where' that functionality is offered.

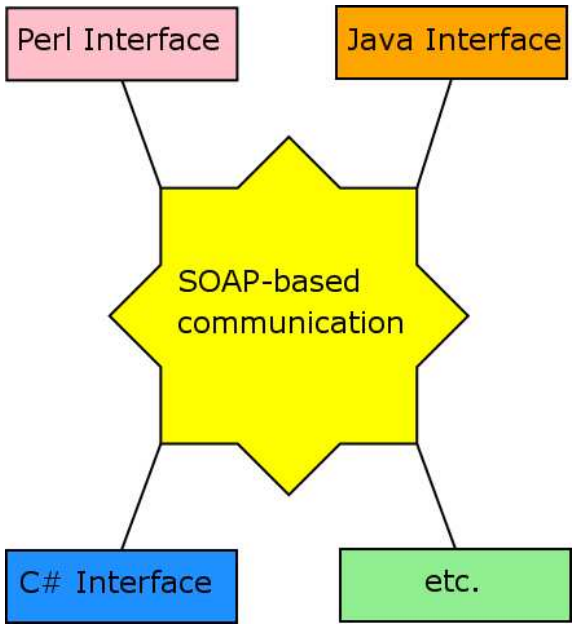

Figure 2.1: SOAP as Wrapper for Different Programming Languages

The WSDL description of a specific service is its interface to client programmers as it specifies all needed operations and (SOAP) message types needed for communication. Anyone who wants to use a specific Web service has to look at its WSDL interface and use it as access point for the client application. This client application can be implemented in any programming language as long it provides SOAP bindings, Web services are therefore not bound to a specific programming language, but to the SOAP protocol.

This allows the usage of Web services across different platforms and programming languages, as all communications happen via transformations to SOAP messages. As presented in Figure 2.1 SOAP is meant to be a wrapper across the different programming languages. The only limitation here is the different maturity of SOAP implementations across those languages. When developing cross-platform interfaces, one is usually restricted by the language with the least mature SOAP implementation included in the development specifications[1]. When talking about Java Web services, Axis [20], the SOAP engine developed by the Apache Software Foundation, has become a standard. Axis already provides many features, but as it is a SOAP implementation for the Java language [2], its full functionality can only be used by Java clients. For Java-only communications the use of Java RMI is recommended, as it's designed for Java

[1] e.g. a project is planned to have interfaces for C sharp and Java, the limitation here is the subset of datatypes supported by both platforms, or by the SOAP bindings of both platforms.

[2] A C++ Version is in development as well, but not regarded in this work

only and is more mature. RMI also has a far better performance than Web services [27][22]. The main disadvantage of RMI in this context of availability and connectivity over different LANs is the necessity of open ports on the server machine. Alternatives are Web services as they do not require open ports, because SOAP is built on top of HTTP[3]. The most common protocol used in connection with SOAP is HTTP, as it relies on the widely available infrastructure of Web servers. Almost any organization hosts Web sites and therefore controls an HTTP server listening on port 80. Those existing servers can be used to accept SOAP calls. Although this is the main advantage of Web services this might get a problem in the future. As SOAP implementations get widely deployed, their weaknesses will certainly be found. Port 25, used for the SMTP protocol for sending e-mails, experienced a similar problem with the rise of viruses as mail attachments [1]. The drawback of Web services in this context is that they (in opposite to RMI) do not provide support for:

- Object references (not natively at least),
- Existing RMI security concepts,
- Distributed garbage collection,
- ...

The benefit of using Web services over RMI is that they are not an extension of an existing programming language (like RMI is an extension to Java). So when developing real cross-platform Web services, the subset of functionality supported by any involved programming languages, must be the main focus. Although those programming languages are mainly .NET's C sharp [13, 14] and Java, the discussed solutions should work for Perl [43] or PHP as well, as both of them provide bindings for simple datatypes. The next chapter describes the interoperability aspects of Web services in general as well as by example and discusses existing problems and possible solutions (and their advantages and drawbacks).

[3]Theoretically, SOAP was designed to fit other protocols like FTP or SMTP too, but HTTP has become the primary target protocol of SOAP.

Chapter 3

Motivation

This chapter discusses the interoperability problems involved in Web service engineering as well as existing solutions and point out the need for an automatic mechanism for interface generation.

3.1 Interoperability Issues

As mentioned before, the usage of Web services in a Java only scenario is easier because of existing serialization and deserialization methods. This section will introduce in how far a Java interface containing complex data types is compatible the script language Perl.

3.1.1 The Problem of Using Complex Types

Within Java it is possible to use complex data types over SOAP. The underlying concept is object serialization, consisting of:

- Serialization on the client side
- Deserialization on the server side

SOAP is used as a container for complex datatypes when they are exchanged between client and server, Figure 3.1 shows the main concept. If the client wants to send a complex type as parameter, it must be serialized into a SOAP message on the client side, and deserialized into an object on the server side. This section will introduce an example that illustrates how serialization and deserialization of complex types works using Axis and Java [5]. Listing 3.1 presents the Java interface class that will be deployed as a Web service via Axis. The weather service provides access to an object of type *Forecast* which

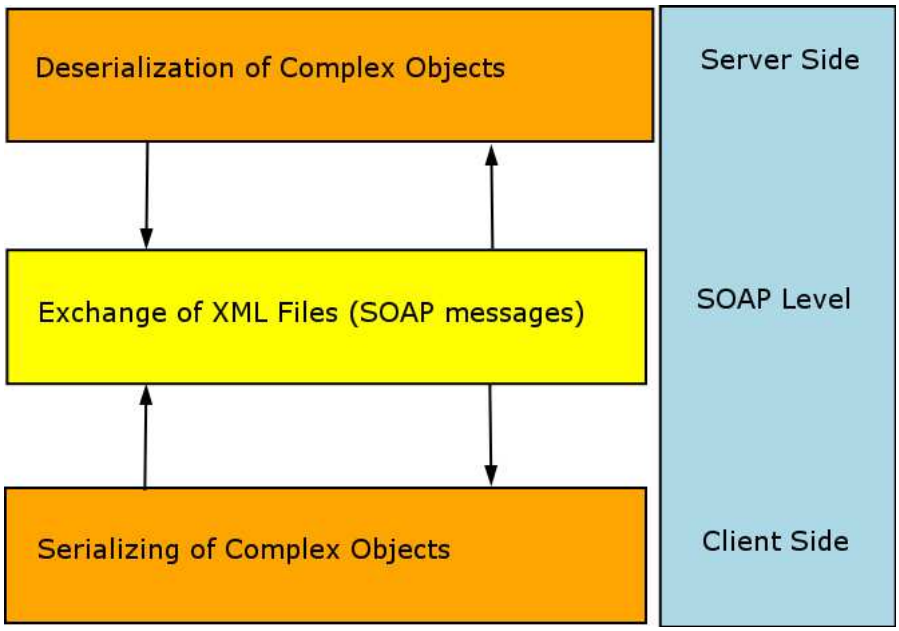

Figure 3.1: SOAP as Container for Complex Datatypes

is a class that contains weather information for a specific region (indicated by its zipcode parameter) and date is the complex data type to be serialized [1].

Listing 3.1: Code for Class Weather (Web service)

```
/*
 * weather.java
 */

package com.pushtotest;

public class weather {

  /** Cretes a new instance */
  public weather() {
  }

  public Forecast getWeather(String zip)
  {
  Forecast fb = new Forecast();
```

[1] This is meant as simple example and therefore the service only instantiates one object of type *Forecast* with static values and does not involve database searches or similar processings.

```
    fb.setZip(ziprq);
    fb.setCity("Campbell");
    fb.setState("CA");
    fb.setDate("April 11, 2003");
    fb.setForecast("20 % chance of rain.");

    return fb;
    }
}
```

Listing 3.2 shows the Java code for the class *Forecast*. The important part is *Forecast's* field that includes several String and byte data variables. Those data types shall be transformed into a SOAP message on the server side and retransferred into the Java object on the client side.

Listing 3.2: The Class Forecast which Will Be Transferred via SOAP

```
package com.pushtotest;

/**
 * Forecast Java Bean for the weather Web Service
 */

public class Forecast {

    String zip = null;
    String city = null;
    String state = null;
    String date = null;
    String forecast = null;
    byte hi = 0;
    byte low = 0;
    byte precip = 0;

    /** Creates a new instance of WeatherBean */
    public Forecast() {
    }

    public void setZip( String thezip )
    {
    zip = thezip;
    }

    ...

}
```

A part of the WSDL file describing the *weather* service is shown in Listing 3.3. It lists the description of the complex data type *forecast* from Listing 3.2.

Listing 3.3: SOAP Representation of the Complex Data Type Forecast

```
<complexType name="Forecast">
  <sequence>
    <element name="city"
             nillable="true"
             type="xsd:string"/>
    <element
            name="date"
             nillable="true"
             type="xsd:string"/>
    <element
             name="forecast"
             nillable="true"
             type="xsd:string"/>
    <element
             name="hi"
             type="xsd:byte"/>
    <element
             name="low"
             type="xsd:byte"/>
    <element
             name="precip"
             type="xsd:byte"/>
    <element
             name="state"
             nillable="true"
             type="xsd:string"/>
    <element
             name="zip"
             nillable="true"
             type="xsd:string"/>
  </sequence>
</complexType>
```

The next step is to serialize this data type, whereas the serialization functionality does not work out of the box. Every service deployed via Axis is configured by a Web Service Deployment Descriptor. Usually there is not a lot of configuration needed. In this case typemapping must be configured in the wsdd deployment files, shown in Listing 3.4.

Listing 3.4: Typemapping Configuration for the Forecast Data Type

```
<deployment
  xmlns="http://xml.apache.org/axis/wsdd/"
```

```
  xmlns:java="xml.apache.org/axis/wsdd/providers/java">

<!-- Services from Weather service -->

  <service name="weather"
      provider="java:RPC"
      style="rpc"
      use="encoded">

    ...

  <operation
      name="getWeather"
      qname="operNS:getWeather"
      xmlns:operNS="urn:weather"
      returnQName="getWeatherReturn"
      returnType="rtns:Forecast"
      xmlns:rtns="http://weather" >
    <parameter
      name="in0"
      type="tns:string"
      xmlns:tns="http://www.w3.org/2001/XMLSchema"/>
  </operation>
  <parameter name="allowedMethods" value="getWeather"/>
  <parameter name="scope" value="Session"/>

  <typeMapping
  xmlns:ns="http://weather"
  qname="ns:Forecast"
  type="java:com.pushtotest.Forecast"
  serializer=
  "org.apache.axis.encoding.ser.BeanSerializerFactory"
  deserializer=
  "org.apache.axis.encoding.ser.BeanDeserializerFactory"
  encodingStyle=
  "http://schemas.xmlsoap.org/soap/encoding/"/>
  </service>
  <service
      name="Forecast"
      provider="java:RPC">
  <parameter
      name="allowedMethods"
      value="*"/>
  <parameter
        name="className"
        value="com.pushtotest.Forecast"/>
```

```
  <beanMapping
  qname="myNS:Forecast"
  xmlns:myNS="urn:ForecastService"
languageSpecificType="java:com.pushtotest.ForecastBean"/>
</service>
```

The *typeMapping* and *beanMapping* elements specify how to process requests regarding the *Forecast* type. The benefit from the more complicated configuration is the possibility to transfer complex objects via SOAP. Listing 3.5 shows how this service can be invoked via a Java client that uses the SOAP layer to retrieve a complex datatype. The *Forecast* object is retrieved from the Web service and can be used like a local object.

Listing 3.5: Java Client for The Complex Weather Service

```
// setting the endpoint
String endpoint =
"http://localhost:8080/axis/services/weather";

// setting up endpoint and call
Service  service = new Service();
Call  call = (Call) service.createCall();
call.setTargetEndpointAddress
     (new java.net.URL(endpoint));

// select the desired operation
call.setOperation("getWeather");

// register the serializer and deserializer
QName qname = new QName("http://weather", "Forecast");
call.registerTypeMapping
(Forecast.class, qname,
new BeanSerializerFactory(Forecast.class, qname),
new BeanDeserializerFactory(Forecast.class, qname));
call.setReturnQName(qname);

// do the actual call and receive the complex object
Forecast f = (Forecast)
 call.invoke(new Object[] {(Object) "2402"};

System.out.println("State:\t\t\t" + f.getState());
System.out.println("Zip:\t\t\t" + f.getZip());
System.out.println("City:\t\t\t" + f.getCity());
System.out.println("Forecast:\t\t" + f.getForecast());
System.out.println("Date:\t\t\t" + f.getDate());
```

The advantages are the easier use of remote objects and its implications on creating remote access to existing applications. The logical drawback is that

this easy usage of serializers and deserializers is bound to the specific programming language (Java in this case). Listing 3.6 presents a client for the same service written in Perl, using the SOAP Lite package [30]. The complex datatype can be received but not processed, therefore the service is unusable for a Perl client. The last line of the code block sets a call to the *getState* method similar to the Java call in Listing 3.5. Unlike the Java equivalent, Perl can not use the retrieved object, as no deserialization is specified (nor could one be specified).

Listing 3.6: Perl Client for The Complex Weather Service

```
my $complexweatherservice = SOAP::Lite
  -> service
    ('http://localhost:8080/axis/services/weather?wsdl');

my $forecastobject =
   $complexweatherservice -> getWeather();

print $forecastobject -> getState();
```

Hence, when developing real cross platform Web services, there is a need for less complex interfaces that use primitive datatypes instead of complex objects. The next sections discuss why typemapping is not a feasible solution and other interoperability issues.

3.1.2 The Web Services Interoperability Organization

The Web Services Interoperability Organization (WS-I) is an open, industry organization chartered to promote Web services interoperability across platforms, operating systems, and programming languages. The organization works across the industry and standards organizations to respond to customer needs by providing guidance, best practices, and resources for developing Web services solutions [36]. Their main target is to create standards or profiles to support the interoperable development of Web services, where interoperable means suitable for and capable of being implemented on:

- Multiple operating systems and
- Multiple programming languages

Those goals are similar to the goals of RAGE, where RAGE is concentrating on interoperability between programming languages and therefore datatypes which are an issue of the WS-I too.

The WS-I Basic Profile

The WS-I published the Basic Profile [28], consisting of a set of constraints and guidelines that should help to implement interoperable Web services. The

Basic Profile has a scope defined by a group of specifications at particular version levels including XML, SOAP, WSDL, XML Schema, SSL, X.509 public key infrastructure et cetera. The profile consists of several requirements (i.e. restarints and clarifications in addition to those standards), a Web service has to meet in order to be compatible with the profile.

- the use of HTTP binding for SOAP is required
- the HTTP POST method must be used
- the use of HTTP binding for SOAP is required

The profile includes rules regarding:

- Messages
  - that are sent to and received by a service
  - Mainly incorporates the following specifications:
    - SOAP 1.1
    - HTTP 1.1
- Descriptions
  - of types, messages and interfaces
  - Mainly incorporates the following specifications:
    - XML 1.0
    - Namespaces in XML 1.0
    - XML Schema part 1 (structures) and part 2 (datatypes)
    - WSDL 1.1
- Discovery [2]
  - Information according to Web service discovery and integration (e.g. UDDI [41]

The key restraints in addition to those standards are [18]:

- e.g. correct use of WSDL's import construct
- precludes the use of SOAP encoding
- requires the use of HTTP binding for SOAP
- requires the use of HTTP 500 status response for SOAP Fault messages
- requires the use of HTTP POST method

[2] This part is left out in this work as it aims at the development of remote interfaces not their discovery.

- requires the use of WSDL1.1 to describe the interface of a Web service
- requires the use of rpc/literal or document/literal forms of the WSDL SOAP binding
- precludes the use of solicit-response and notification style operations
- requires the use of WSDL SOAP binding extension with HTTP as the required transport
- requires the use of WSDL1.1 descriptions for UDDI tModel elements representing a Web service

The specification is mainly targeted at developers working on different SOAP-processors, WSDL-Parsers or code generators. The most interesting aspect of the basic profile in the RAGE context is the preclusion of the SOAP encoding. The SOAP encoding is precluded, because it has been the source of many interoperability problems. Therefore the WSDL SOAP binding must be either *RPC/literal* or *Document/literal*.

Complex Java Types and the Basic Profile

The service containing the complex return type, presented in the last section, is not conforming to the basic WS-I profile. The encoding for the service was set to *encoded*, whereas a *literal* setting is required (as shown in Listing 3.4). The WS-I basic profile does not allow the use of *encoded* as value for the encoding attribute (which was set to *encoded* to allow typemapping). There exist different types of encodings which specifiy the transferring from an object to an XML message which is going to be exchanged. The most common ones are RPC-style encoding and literal encoding. The latter encodes all parameters in a single XML message, whereas the RPC-style encoding automatically serializes all parameters into XML [6]. The point is that the WS-I organization considers the encoding type as an important factor in allowing interoperability, hence the usage of complex objects is not allowed according to their de facto standard.

The WS-I in addition has released a set of testing tools to determine whether a Web service is conformant with the guidelines of the basic profile [37].

As the use of complex types has been identified as a source for interoperability problems, the next step is to analyze how they can be replaced by simple datatypes.

3.2 The Common Solution - Usage of Wrapper Classes

Whenever simple interfaces are needed for complex APIs, wrapper classes have to be implemented. Such a class takes as well as returns only simple datatypes but uses those parameters to accesses the object-oriented API locally. An example that covers this case is given in Listing 3.7.

Listing 3.7: Wrapper Class for the Complex Weather Service

```
/*
 * Created on Oct 18, 2004
 * as part of complexService
 * weatherWrapper.java
 */

package com.pushtotest;

public class WeatherWrapper {

  // use the complex object as field
  private Forecast forecastObject;

  public WeatherWrapper(){
    this.forecastObject = new Forecast();
  }

  // constructor of the wrapping class,
  // passes values to the ForeCast object
  public void createForecast(String zip){
    this.forecastObject.setZip(zip);
    this.forecastObject.setCity("Campbell");
    this.forecastObject.setState("CA");
    this.forecastObject.setDate
                ("April 11, 2003");
    this.forecastObject.setForecast
                ("20 % chance of rain.");
  }

  public String getCity(){
    return this.forecastObject.getCity();
  }

  public String getState(){
    System.out.println(this.forecastObject.getState());
    return this.forecastObject.getState();
  }

  public String getDate(){
    return this.forecastObject.getDate();
  }

  public String getForecast(){
    return this.forecastObject.getForecast();
```

```
  }

  public void setCity(String city){
    this.forecastObject.setCity(city);
  }

  public void setState(String state){
    this.forecastObject.setState(state);
  }

  public void setDate(String date){
    this.forecastObject.setDate(date);
  }

  public void setForecast(String forecast){
    this.forecastObject.setForecast(forecast);
  }
}
```

This listing can be deployed as a Web service interface and has no limitations regarding languages or datatypes but also shows how much effort is needed to wrap such a simple API (the class *Forecast*. Listing 3.8 shows a Perl script that accesses the manually coded wrapper class.

Listing 3.8: Perl Client for the Manually Coded Wrapper Class

```
#! /usr/bin/perl -w

use SOAP::Lite;

# setting up the endpoint
my $weatherservice = SOAP::Lite
-> service(
'http://localhost:8080
        /axis/services/WeatherWrapper?wsdl');

# call the create method (pass the zip code to set
# the standard values for the forecast class variable)
$weatherservice->createForecast("12345");

# getting the needed values
my $state = $weatherservice  -> getState();
my $zip = $weatherservice  -> getZip();
my $city = $weatherservice  -> getCity();
my $forecast = $weatherservice  -> getForecast();
my $date = $weatherservice  -> getDate();
```

```
# display the retrieved values:
print "\nRetrieved values:\n";
print "State:\t\t".$state."\n";
print "Zip:\t\t".$zip."\n";
print "City:\t\t".$city."\n";
print "forecast:\t".$forecast."\n";
print "date:\t\t".$date."\n";
```

Perl can handle all of the used datatypes (albeit only Strings are used in this case) without any problems and can fully control the remote system.

3.2.1 Arising Problems

As a software project changes from version to version, the remote access interfaces also have to be changed. This means that there is a need of updating those interfaces with nearly every new release of the software. Therefore a solution that generates wrappers automatically and in a uniform way is proposed (in opposite to manual wrapper generation).

3.3 Objectives Revisited

This work proposes a solution for the problems mentioned in this chapter, namely a concept for a software tool that creates a simple interface for any existing Java application and is accessible by any platform or programming language. Requirements for such an interface are:

- It must contain simple datatypes only
  - In order to bypass the SOAP-related problems mentioned above
- The tool shall be able to control sophisticated APIs remotely
  - To be relevant for any application
- Its underlying concept could be ported to any programming language
  - So that it is not a Java-only concept
- The resulting interface should be accessible by any programming language (especially low level or scripting languages), i.e. provide maximum interoperability, and be extendable to other existing remote access mechanisms
- This should be possible automatically, to avoid the efforts of manual wrapper coding

The overall concept is presented in Chapter 4. Chapter 5 introduces RAGE, the Remote Access Generator Engine, a sample implementation of such a system. It supports Web services (via Axis), CORBA and RMI as remote mechanisms. Furthermore RAGE is realized as modular architecture and is easily extendable to support other remote mechanisms, whereas this work concentrates on RAGE's abilities related to Web services.

Chapter 4

Server-Side Encapsulated Objects

The fact that there is no guaranteed support for exchanging objects across different programming languages, and for the Web service scenario more important, across different SOAP implementations, creates the need for restricting the data types exchanged between service and client to very simple ones or even primitive data types. The solution proposed in this chapter could possibly be used for any object-oriented programming language, the overall concept itself is not bound to a particular architecture. This chapter introduces an overall concept of object caching on the server-side that should fit those needs (particularly for a Java environment).

4.1 The Object Cache Approach

Java objects need to be accessed via the net, but can not be transferred due to limitations of SOAP implementations and differences between Java and other programming languages. The access to those classes and their methods must be guaranteed using other techniques than transferring the objects as a whole. The main concept presented in this work is server-side encapsulating the objects and provide remote access to its methods (similar to manually coded wrapper classes). The client uses a mechanism to control objects remotely, access works only via object IDs. Those IDs and parameters to the object's methods, that mostly are primitive datatypes, are the only data submitted via the net. If the client wants to create an object of type *X*, it calls a method *createX* with or without parameters and gets an ID for this created object, used as a reference. Thus the possibility of instantiating any desired class and access any methods without the need of transferring the whole object is given. The basic idea is illustrated in Figure 4.1. Use of objects is analogous to local Java programming. The difference is that the clients do not work with the ob-

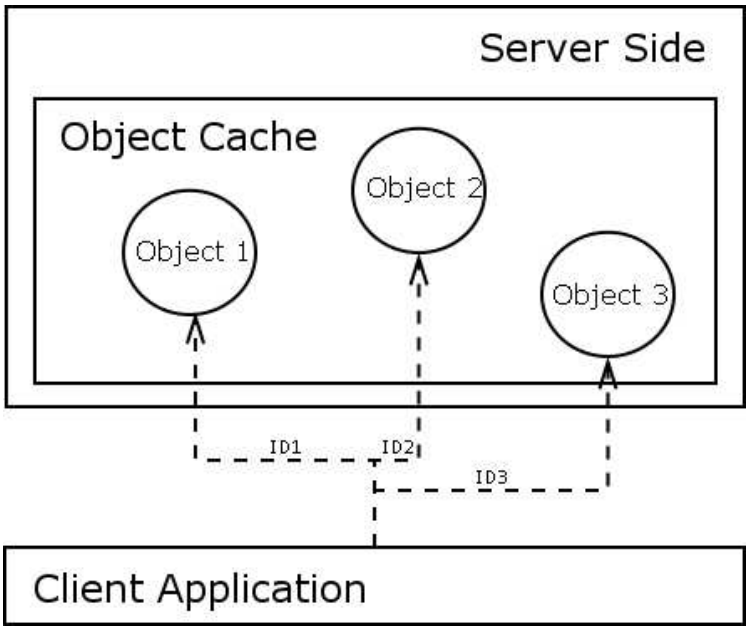

Figure 4.1: Caching Objects on the Server Side

jects themselves, but via references only, realized as an exchange of IDs. This idea can be implemented using a simple HashTable. The process implies the following:

- Create an object (e.g. via its constructor)
- Retrieve an unused ID
- Store that pair in the HashTable and return the ID to the user

Therefore the result of any object instantiation is not the complex object but only an ID referencing the object stored on the server side.

4.2 Providing Uniform Access

Once an object is instantiated and stored on the server side, all its methods can be accessed following this simple scheme:

- Access a specific object via its ID that has been retrieved at its instantiation
- Call any methods of this object with its usual parameters (plus an extra ID parameter representing the pointer to the object)

Theoretically any parameter is possible since one could create objects of any class and store it in the object cache and therefore pass its IDs as parameters. So it would be possible to store an object of built-in datatypes like *Vector* or *HashTable* in the object cache, although it is recommended to avoid the excessive use of external classes (to the actual application) because of its more complicated usage see Section 7.6. Such an object cache implementation is a generic approach to interface generation, although its simple idea has limitations that will be discussed at the end of this chapter.

4.3 Existing Technologies

Tanenbaum [38] explains different approaches for remote object invocation, including remote invocation for:

- Compile Time Objects
  - per definition: instance of a class
  - bound to programming language (e.g. Java)
- Run Time Objects
  - object adapter as interface between different languages
  - independent of the programming language

He also outlines the drawbacks of compile time objects namely the binding to a specific language. This is why RMI is not a feasible solution for remote access, it is a Java extension and not accessible from other languages. Furthermore he introduces the possibility of wrappers to bind a certain functionality to an object. The main difference between those approaches and the object cache is that there is no binding to a proxy on the client side, the only reference used are the object's IDs. This is analogous to the object cache: objects are bound to the server side and can only be referenced. In out case the adapter is located between Java and SOAP, presenting an interface that is ready for use with any other languages that offer SOAP-bindings.

4.4 Object Caching, the RAGE Way

This section introduces a simple sample implementation of an object caching mechanism in Java. The class *RemoteObjectCache* uses a *Hashtable* to store objects by an ID of type *long*. Furthermore it has methods to add a new object and to access the stored objects. *RemoteObjectCache* is a rather simple class as shown in Listing 4.1. Access to any stored objects is realized via this class in every interface generated by RAGE, i.e. *RemoteObjectCache* is an inner class of any generated interface class.

Listing 4.1: The Remote Object Cache Implementation

```
class RemoteObjectCache {
  private java.util.Hashtable cache;
  private long nextkey;

  public RemoteObjectCache () {
    cache = new java.util.Hashtable();
    nextkey = 1;
  }
  public long add(Object obj) {
    cache.put(new Long(nextkey), obj);
    return this.nextkey++;
  }
  public Object get (long id) {
    Object obj = this.cache.get(new Long(id));
    return obj;
  }
  public void set (long id, Object obj) {
    this.cache.put(new Long(id), obj);
  }
}
```

This is the main caching functionality that all other stages of RAGE rely on. Any enhancement concerning for instance object lifetime could be implemented here.

4.5 The Weather Example, Again

To complete the weather example another interface to the same API is presented. It shows how an interface generated by RAGE using an object cache works and looks like (the details about automatically generated interfaces that rely on the presented *ObjectCache* class will be outlined in Chapter 5. This interface for the weather example one will include primitive datatypes only and will be accessible even by Perl. In addition to the handwritten weather class this one uses a caching mechanism similar to the one explained in the previous sections (the details of the implementation are outlined in the next chapter, this shall only demonstrate the concept of uniformly generated wrappers). Listing 4.2 presents the methods provided by this interface which are automatically generated by the RAGE tool, the details will be explained in the next chapter, this is meant as an outlook.

Listing 4.2: Simple and Interoperable Web Service Interface (Generated by RAGE)

```
package weather;
```

```
public class weatherProxy {
  class RemoteObjectCache {
    ...
  }

  private RemoteObjectCache oc;

  public weatherProxy() {
    oc = new RemoteObjectCache();
  }
  public long createForecast() {
    long id = oc.add(new Forecast());
    return id;
  }
  public String Forecast_getState(long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    String rv = o.getState();
    oc.set(forecastID, o);
    return rv;
  }
  public String Forecast_getDate(long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    String rv = o.getDate();
    oc.set(forecastID, o);
    return rv;
  }
  public void Forecast_setZip
      (long forecastID,String s1) {
    Forecast o = (Forecast)oc.get(forecastID);
    o.setZip(s1);
    oc.set(forecastID, o);
  }
  public void Forecast_setCity
        (long forecastID,String s1) {
    Forecast o = (Forecast)oc.get(forecastID);
    o.setCity(s1);
    oc.set(forecastID, o);
  }
  public void Forecast_setState
        (long forecastID,String s1) {
    Forecast o = (Forecast)oc.get(forecastID);
    o.setState(s1);
    oc.set(forecastID, o);
  }
  public void Forecast_setDate
        (long forecastID,String s1) {
```

```
    Forecast o = (Forecast)oc.get(forecastID);
    o.setDate(s1);
    oc.set(forecastID, o);
  }
  public void Forecast_setForecast
        (long forecastID,String s1) {
    Forecast o = (Forecast)oc.get(forecastID);
    o.setForecast(s1);
    oc.set(forecastID, o);
  }
  public String Forecast_getZip(long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    String rv = o.getZip();
    oc.set(forecastID, o);
    return rv;
  }
  public String Forecast_getCity(long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    String rv = o.getCity();
    oc.set(forecastID, o);
    return rv;
  }
  public String Forecast_getForecast
        (long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    String rv = o.getForecast();
    oc.set(forecastID, o);
    return rv;
  }
  public void Forecast_setLow
        (long forecastID,byte b1) {
    Forecast o = (Forecast)oc.get(forecastID);
    o.setLow(b1);
    oc.set(forecastID, o);
  }
  public byte Forecast_getLow(long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    byte rv = o.getLow();
    oc.set(forecastID, o);
    return rv;
  }
  public void Forecast_setHi
      (long forecastID,byte b1) {
    Forecast o = (Forecast)oc.get(forecastID);
    o.setHi(b1);
    oc.set(forecastID, o);
```

```
  }
  public byte Forecast_getHi(long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    byte rv = o.getHi();
    oc.set(forecastID, o);
    return rv;
  }
  public void Forecast_setPrecip
       (long forecastID,byte b1) {
    Forecast o = (Forecast)oc.get(forecastID);
    o.setPrecip(b1);
    oc.set(forecastID, o);
  }
  public byte Forecast_getPrecip(long forecastID) {
    Forecast o = (Forecast)oc.get(forecastID);
    byte rv = o.getPrecip();
    oc.set(forecastID, o);
    return rv;
  }
  public long createweather() {
    long id = oc.add(new weather());
    return id;
  }
  public long weather_getWeather
      (long weatherID,String s1) {
    weather o = (weather)oc.get(weatherID);
    Forecast rv = o.getWeather(s1);
    oc.set(weatherID, o);
    long id = oc.add(rv);
    return id;
  }
}
```

All parameters and return types are now primitives (like in the handwritten wrapper class), the most important change is the remote control mechanism for objects which works in a similar way for any API, in opposite to handwritten wrapper classes that never stick to a common concept (i.e. programmers provide handwritten wrapper classes that could be very different from each other). To access the same functionality as in the complex client the programmer would have to call the following methods:

- *createweather* to instantiate the service (to get a pointer to a *weather* object),
- *weather_getWeather* to get a pointer on the actual forecast,
- and the different *Forecast_X* methods to access the object's variables

When using only simple data types as in Listing 4.2, the use of literal encoding is not a problem and the service therefore is compatible with the WS-I basic profile. This moreover means that a client doesn't have to be written in Java but could be written in any other programming language that supports the involved primitive datatypes like *String* and *long*, virtually any programming language. The example Perl client that can access this interface easily and is shown in Listing 4.3.

Listing 4.3: Perl Client for the Simple Web Service Interface

```
use SOAP::Lite;

# setting up the endpoint
my $weatherservice = SOAP::Lite
->service
(http://localhost:8080/axis/services/weatherProxy?wsdl)


#  create a weather object (on the server side)
my $weatherid = $weatherservice->createweather();
# get the forecast object from one of weather's services
my $forecastid = $weatherservice
    ->weather_getWeather($weatherid, "2402");

# getting the values of the forecast object
my $state = $weatherservice
    ->Forecast_getState($forecastid);
my $zip = $weatherservice
    ->Forecast_getZip($forecastid);

# display the retrieved values:
print "ID of weather Object: ".$weatherid."\n";
print "ID of Forecast Object: ".$forecastid."\n";
print "\nRetrieved values:\n";
print "State:\t\t".$state."\n";
print "Zip:\t\t".$zip."\n";
```

This interface allows any Perl client to communicate with the service and to fully control its API, albeit the Perl client implementation gets more complicated than the one using object mapping and Java. Further information about the underlying concept of RAGE is given in Chapter 5. So the main goal of this thesis is to present a mechanism that easily allows the automatic generation of remote interfaces for existing applications. It will use a a delegating mechanism for objects using primitive data types. Another goal is to achieve interoperability based on those primitive data types used to remotely control objects and methods on the server side.

4.5.1 Object Lifetime and Session Management

The object cache in its simple form neither provides transaction management nor session handling nor explicit management of object lifetimes. Transaction management must be implemented at the application level, independent from the remote access mechanisms. Open questions in this context are:

- What is the lifetime of objects in the cache?
- How are sessions managed?

The presented implementation saves instances of all created objects. Whenever they are needed the are pulled out of the cache. There is no explicit management of sessions implemented so far which is not a problem as long as Java Web services are concerned, because sessions of the generated interfaces are managed by the SOAP server's session management. The object lifetime is entirely managed by the Java Virtual Machine's automatic garbage collection Axis is running in. Whenever a SOAP session expires, the interface object expires as well. However, at the end of each session (defined by the SOAP server) all previously instantiated objects get destroyed. Axis' session management is limited to a simple object scope parameter (specified at the deployment description for each service) that only supports limited lifetime control of the service's object (the class that is deployed via Axis).

- Request scope, the default, will create a new object each time a SOAP request is sent to the service,
- Session scope creates a new object for each client session,
- Application scope creates one singleton shared object for all requests

Whereas such an approach should not affect transaction management within the application, it probably will instantiate far more objects than actually needed. Axis' application scope setting control seems the best for that matter and is recommended as default setting for interfaces generated by RAGE. This may be a performance drawback, but results in a significantly easier implementation. The disadvantages in performance probably won't be noticed until very complex APIs are used for long lasting communications.

4.5.2 Where Do We Go from Here?

The next step is to use the object caching ideas to generate interfaces automatically. The object cache's uniform approach allows easy generation of simple interfaces based on a projects class structure. The next chapter introduces the Remote Access Generator Engine and its approach to automatic code generation for Java projects, extending the concepts explained in this chapter.

Chapter 5

The Remote Access Generator Engine - Basics

The Remote Access Generator Engine is a software system aimed at the automatic code generation of different kinds of interface classes for an existing Java package. It implements the ideas and concepts presented in the previous sections. RAGE is based on analysis of existing Java classes and packages and is making heavy use of *java.lang.Reflection*, so it is bound to software projects written in Java. Note that interfaces created with RAGE can be accessed with any programming language that provides SOAP bindings.

5.1 RAGE Overview

Figure 5.1 outlines the components of RAGE's architecture. RAGE consists of one base layer that generates a flat proxy class with access to all classes and methods specified in the config file according to the object cache approach presented in the lastchapter. Its advantages and problems are evaluated at the end of this chapter.

Chapter 6 discusses choreography languages and in how far they can solve the problems of layer one. Layer two or the WSCI-layer is set up on top of this proxy and combines its great number of methods to processes, each consisting of one or more layer one methods according to those choreography concepts.

Layer three or the RMI-layer provides RMI-access to either the flat proxy layer one or the choreography enabled proxy of layer two (i.e. either the flat proxy class or the choreography based proxy consisting of fewer but more complex methods), consult [15] for details.

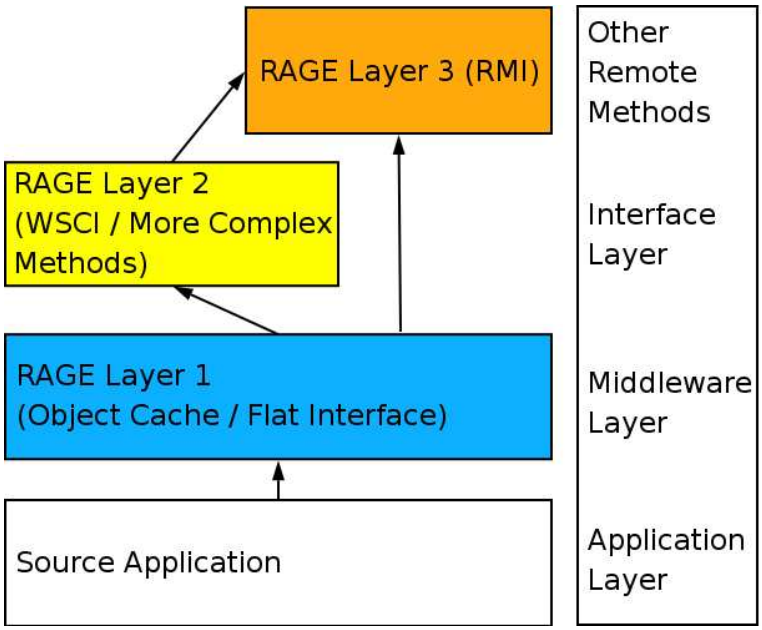

Figure 5.1: Three Layer Architecture of RAGE

5.2 Proxy Generation (RAGE Layer One)

As mentioned before this part is responsible for generating a low-tech flat proxy class that provides access to any methods (of any classes) of a given Java package. Listing 5.2 shows how objects can be instantiated and get stored in the cache. Furthermore it outlines how objects get instantiated and used via references. Each method of a generated interface class has the return type of the method in the source class, if it is a simple datatype, or an ID if it is an object. Method names are following the pattern:

SourceClassName_MethodName(long IDParameter, other parameters)

Every method needs the ID of the corresponding object in the cache as first argument. Then the object is pulled out of the cache, the method gets called and the object (that may have changed) is rewritten into the hashtable at its old position.

5.3 From an API to a Proxy Class

This section shall explain how the generation of layer one proxies works in detail. For every class in the source package there will be:

- A method tho create an object of this class
  - One method for each constructor that simply passes the parameters and stores the instantiated object in the object cache
  - And returns an ID to the newly generated object
- One method for every method in the source class
  - That takes all parameters of the source method and an ID parameter as arguments
  - The ID parameter specifies the object the method is called for (the ID returned from another instantiation)

Another very important aspect is the parameter substitution. Every argument of a method is replaced with an ID for the type of the argument. Listing 5.1 shows a possible method in the source package (*sourceMethod* that has a complex type as parameter. The method *generatedMethod* substitutes this complex type with an ID to an object of the required type. This implies that createmethods have to be generated for every class that us used as an argument (i.e. the more external classes will be used either as return types or arguments the more methods will have to be generated).

Listing 5.1: Substitution of Complex Parameters

```
// method in the source package
public void sourceMethod(ComplexType ct){
  ...
}

// method in the generated proxy
public void generatedMethod(long ctID){
  ...
}
```

This mechanism allows RAGE to bypass any problems according to complex types. Every object of any type can be instantiated on the server side and is accessed only by its ID (as well as passed as parameter to other methods).

Listing 5.2: Example Method of Generated layer One Interface Proxy

```
private RemoteObjectCache oc;

public ProxyClass() {
  oc = new RemoteObjectCache();
}
public long createObjectX(Param v1,Param v2){
  long id = oc.add(new ObjectX(v1,v2));
  return id;
}
```

```
public long createObjectX(long vectorID1){
  Vector v1 = (Vector)oc.get(vectorID1);
  long id = oc.add(new ObjectX(v1));
  return id;
}
```

The first method of this example is the constructor of the proxy class which instantiates the object cache. The second method shows how to instantiate an object from the source package, where its parameters are only simple datatypes. The third method *createObjectX(long vectorID1)* shows how a constructor that needs complex types is called (i.e. object parameters pointing to other objects). Before calling this method, another method *createVector()* would have to be called. The parameter is the ID of the instantiated *Vector* object. Before adding the instantiation of *ObjectX* the previously stored *Vector* is taken out of the *HashTable* and passed as argument. So one would have to call *createVector()* to put an object into the *HashTable* and get this *Vectors* ID, and then call *createObjectX(long vectorID1)* and pass the *Vector*'s ID to it, in order to instantiate and get the ID of the *ObjectX*. Finally, Listing 5.3 illustrates how to call a method of a stored object.

Listing 5.3: Example Method of Generated Layer One Interface (2)

```
public boolean ClassName_MethodName(long classNameID) {
  ClassName o = (ClassName)oc.get(classnameID);
  boolean rv = o.MethodName();
  oc.set(sourceclasstypeID, o);
  return rv;
}

public boolean Person_isMarried(long personID) {
  Person o = (Person)oc.get(personID);
  boolean rv = o.isMarried();
  oc.set(personID, o);
  return rv;
}
```

The first method shows the generic pattern for any methods, whereas the second method shows an example. It is assumed that a class *Person* exists in the source package and has a method *isMarried()* with the return type *boolean*. In order to access this method a *Person* object needs to be created before and stored in the object cache (e.g. by a method called *createPerson(String Name, boolean isMarried, ...))*. This object is taken out from the object cache, then the method *isMarried* is called, the *Person* object is written to its old place in the *HashTable* since it could have been changed, and the return value *(true or false)* is passed on.

5.4 Evaluation of the RAGE Approach

This approach is both powerful and simple as it could be extended to any classes and objects and the object cache could hold very complex structures. As the examples of this section (and the previous sections)might imply, the RAGE approach in its simple form has another disadvantage, namely the fine granularity of its generated methods. Listing 4.2 points out how many methods are created for a very simple class.

Therefore the main problems of those RAGE layer one proxies are:

- A very flat structure of the interface class
- A large number of methods in the interface class

Assuming an interface is generated for every single method of a given Java package it would be :

$$ResultingNumberOfMethods = NumberOfClasses * NumberOfMethods$$

This flat structure also is not easy to use and, more important, each method call involves:

- A new server connection to be established
- Serialization to XML
- Deserialization to Java datatypes

A closer look at the Perl client in Listing 4.3 shows possible improvements due to method regrouping on the server side. The client implements calls to the *createweather* and *weather_getWeather* methods, which could easily be transferred to a single method on the server side. Albeit this particular example may seem far-fetched, it shows that a shifting of functionality from the client application to the Web service can clearly reduce the number of connections (and therefore the number of serializations). The benefits of server side regrouping of methods to increase performance are discussed in detail in Section 6.3.

Hence, the main question is if more complex methods can be specified and if their application makes sense at all. Therefore more sophisticated configuration mechanisms are discussed in the next chapter which deals with the feasibility of Web services choreography languages for the specification of more complex methods.

Chapter 6

Web Service Choreography Languages

This chapter gives an introduction to Web service choreography and discusses its feasibility for specifying more complex methods on the server side (compared to RAGE's flat layer one proxy classes) to improve the performance and usability of remote interfaces generated by RAGE.

6.1 What is Web Service Choreography?

WSDL has become a standard for the description of a Web service's interface, neither describing the semantic nor the contextual use of a Web service. Web service choreography describes the dynamic behavior of a Web service in addition to the static interface description of the WSDL and the underlying techniques. Choreography aims at describing a service as part of both internal and possible external business processes. This concept tries to support the easy inclusion of foreign services into business processes. Web service choreography is also a great concern in the workflow community, as Web services can be understood as a new concept of cross-organizational business processes [46]. The main goals of choreography developments are:

- Description of the behavior of a Web service's interface
- (Cross-organizational) business processes built on top of Web services

Figure 6.1 presents the common stack of Web services technologies and how choreography fits in [21]. XML is used for data encoding and typing using XML Schema [45], SOAP is built on top of those technologies for message exchange. The interface and behavior layers are responsible for describing an existing Web service in terms of:

- Their static interface using a description language like WSDL

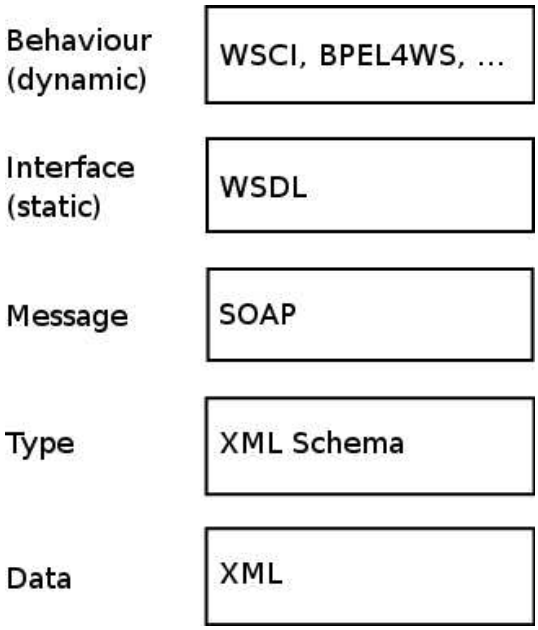

Figure 6.1: Stack of Web Services Including Choreography (Behavior) Layer

- Their dynamic behavior using choreography languages

6.2 An Overview of Existing Technologies

This section shall give an overview of existing technologies as well as point out why RAGE relies on WSCI and explains its key features in greater detail. There have been a lot of specifications and ideas for Web service choreography (as well as XML-based business process modeling in general) from different parties throughout the last years. Some of them (the most common ones) are listed below:

- WSCL, the Web Service Choreography Language [34],
- XLANG, by Microsoft [39]
- WSFL, the Web Services Flow Language by IBM [31]
- BPEL4WS, Business Process Execution Language for Web Services by IBM, a convergence of XLANG and WSFL [40]
- WSCI, the Web Services Choreography Interface by Sun Microsystems which shall be examined in greater detail

- BPSS, the Business Process Specification Scheme by OASIS ebXML [26]
- XPDL, the XML Processing Description Language defined by the Workflow Management Coalition [4]
- BPML, the Business Process Modeling Language by the Business Process Management Initiative (BPMI.org) [25]
- ...

All of those languages aim at similar goals, but none of them has become a standard so far. Not all are designated choreography languages for Web services only, but meant to describe business processes or workflows in general (e.g. BPSS, BPEL, or XPDL). Probably the most important will be the Web Services Choreography Description Language of the W3C's Web Service Choreography Working Group [7]. Most standards in Web services like SOAP and WSDL have been developed by the W3C and so WSCL is ideal as it corresponds closely with those other standards. The W3C also tends to incorporate the most feasible existing standards into its own recommendations, hence many of the standards listed above influenced the WSCL. However, WSCL was not yet released at the time of the evaluation of feasible flow languages, but it would have been the first choice. So far the specification everyone agrees upon has yet to be found.

All presented languages satisfy the main required constructs like synchronization, sequence or exclusive choice [42] and therefore are possible candidates to be used within RAGE. WSFL is the only purely graph-based language, all others are block-structured. BPEL4WS is an exception as it combines features from both WSFL and XLANG, resulting in a hybrid language. Both XLANG and WSFL correspond to the flow language of IBM's MQSeries Workflow and the flow language in Microsoft's BizTalk, respectively. BPEL4WS is supported by Microsoft and IBM and is a well approved approach because of its experience in the workflow area. BPSS as part of the ebXML initiative is a universal approach to business process modeling based on the Unified Modeling Methodology (UMM).

XLANG and WSFL became obsolete because both IBM and Microsoft favor BPEL4WS, WSCL emerged after the evaluation of possible languages. The ebXML approach seems far to complex for RAGE's needs. Only WSCI and BPEL4WS are designed specifically for Web services based choreography and provide rather manageable concepts.

BPEL4WS and WSCI both lie on top of the WSDL interface model of the WSDL. The main difference is that WSCI defines all choreography aspects within the context of individual Web services as shown in Figure 6.2, i.e. every participating Web service has its own WSCI description that references WSDL operations. Besides, WSCI offers a global model referencing two or more WSCI descriptions of Web services. By contrast, BPEL4WS defines those aspects at the process level that involves two or more Web services (which, of course, references the atomic operations of individual Web services

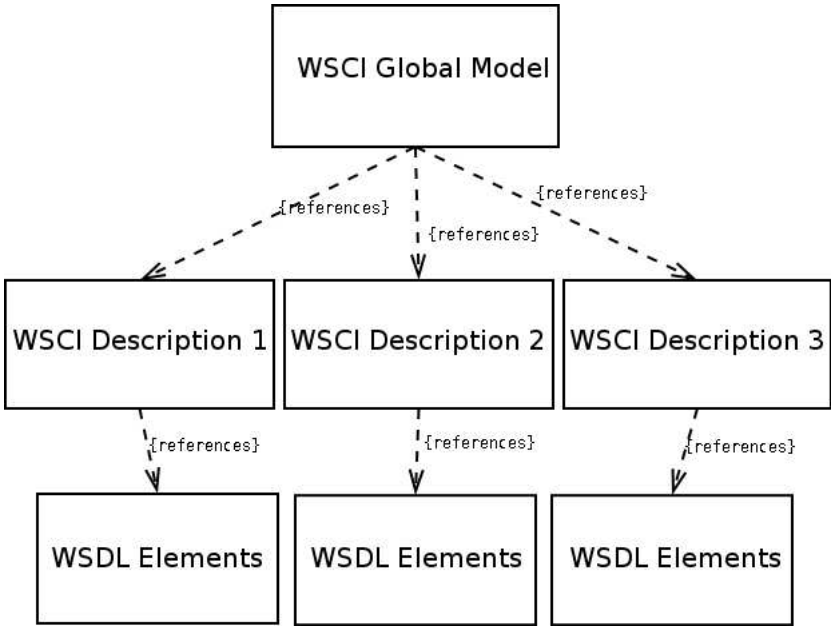

Figure 6.2: The WSCI Model of Referencing WSDL Operations

too). The interesting aspect is that the BPEL4WS model is built of only two layers, whereas WSCI offers three layers (analogous to Figure 6.2). This thesis and the extended RAGE application that will be presented later uses parts of Sun's Web Service Choreography Interface possibilities to describe a static service in terms of business processes, the slight advantage in this context over BPEL4WS is that WSCI's choreography description is bound to the individual Web service rather than greater collaborations or processes. WSCI is also an interesting candidate because of it's simplicity as it is mainly used to examine if the choreography approach is feasible at all.

6.3 Introducing the Web Services Choreography Interface

The Web Services Choreography Interface is meant to describe how Web service operations can be choreographed in the context of a message exchange with the Web service [17]. This standard has initially been developed by Sun, Belinea et al. and finally became adopted by the World Wide Web Consortium in a Note in August 2002. WSCI describes the interdependencies among the Web service's operations so that any client:

- Can understand how to interact with such service in the context of the given process, and
- Can *anticipate* the expected behavior of such service at any point in the process' lifecycle.

Note that a client often is another Web service in this context. WSCI offers a simple, XML-based syntax to describe Web services in terms of business processes. The most important constructs (XML-Elements) are:

- action
  - The basic WSCI construct
  - Describes an atomic part of a process
  - reference to an existing WSDL operation name
- sequence
  - Container for actions
  - refers to a (reusable) list of actions
- process
  - Refers to a business process consisting of several actions or a sequence
- Interface
  - Container for several processes

Listing 6.1 shows a very simple example for a WSCI configuration.

Listing 6.1: Basic WSCI Syntax

```
<?xml version = "1.0" ?>
<wsdl:definitions name = "Service␣Dynamic␣Interface"
    targetNamespace = "http://serviceEndpoint"
    xmlns:wsdl = "http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema"
    xmlns = "http://www.w3.org/2002/07/wsci10">

    WSDL complex types
    WSDL message definitions
    <wsdl:portType name="portTypeName">
     <wsdl:operation name="Operation1">
       ...
     </wsdl:operation>
     <wsdl:operation name="Operation2">
       ...
     </wsdl:operation>
```

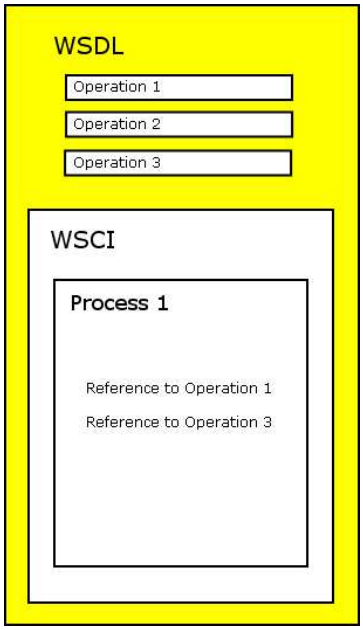

Figure 6.3: Relationship of WSDL and WSCI

```
    ...
  </wsdl:portType>

  <wsci:interface name="WSCIExample"
                  xmlns:wsci="WSCI-NS">
    <wsci:process name="Process1">
      <wsci:sequence>
        <wsci:action
              operation="Operation1">
        </wsci:action>
        <wsci:action
              operation="Operation2">
        </wsci:action>
      </wsci:sequence>
    </wsci:process>
  </wsci:interface>
</wsdl:definitions>
```

This relationship is presented in Figure 6.3, the WSCI elements are embedded into the WSDL file of the described service. WSCI extends the WSDL definitions by one ore more interface elements, the main containers, and one ore more processes in each interface element. This example defines a process

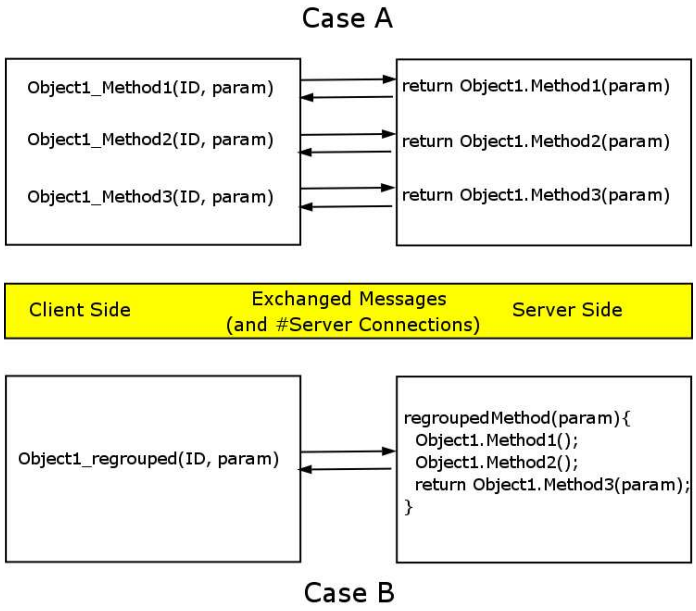

Figure 6.4: Regrouping of Methods on the Server Side and its Impact on the Number of Connections

named *Process1* consisting of two actions, references to WSDL operation elements, i.e. the services methods. The syntax allows to specify the semantic of a service in terms of the order of the called methods and their overall meaning in the business process. The interface from Listing 4.2 on could benefit from such a re-grouping of methods. A connection between *weather_getWeather* and *Forecast_getHi* or any other of *Forecast's* methods would make assessing the service easier. This would simply eliminate the first step of the client, namely to get an ID for a weather object and avoid the need of making another call to the service. Figure 6.4 outlines the possible benefit from such a regrouping on the server side. *A* covers the simple client using many connections and calling many methods. *B* shows the process for the regrouped interface, resulting in only one method, one server connection and one message exchange. This figure exactly covers the intended usage of WSCI, namely to reduce the number of messages exchanged and the number of server connections to be established.

6.3.1 Usage Scenarios for WSCI

WSCI is designed to describe the participation of Web services in long lasting and choreographed message exchanges. A WSCI description should help any developer in using the service and understanding its tasks and which

messages it is able to receive at what time. It offers the possibility to describe a services usage in different views and by different users or clients (the wsci:interface element could be used as adoption to specific clients). In addition, WSCI provides constructs to address message correlation, an aspect that is entirely ignored in this context, but a very important one, as message correlation describes how conversations between services are structured.

Moreover WSCI proposes so called Global Models, that consist of several interface descriptions like the ones presented in this section. Global Models are used to manage larger business collaborations including different parties and services. This aspect of WSCI is also very important since the ultimate goal of Web services is to automate business collaborations both inside and outside businesses. The Global Model aspect of WSCI is not explained in greater detail because collaborations among more than one service are not considered in the RAGE implementation.

6.4 Weaknesses in this Context

The obvious limitations of the usage of WSCI is the limited expressiveness of the WSCI syntax. It is not possible to change any semantics of involved methods or processes. WSCI's task in this scenario is the regrouping and summarizing of atomic methods to aggregated methods. Therefore it fulfills its task for the RAGE scenario. A more complex syntax would on the one hand lead to a greater descriptiveness of any resulting interface methods, but on the other hand get very close to actually coding the methods manually. However, a more complex choreography language would still preserve the information about the interfaces beyond a certain release of a software package. Another important advantage of the simple language is that it is far less prone to changes between releases of a certain API (i.e. there could be changes in the API that would force changes in a more complex language but not in WSCI). The decision to use WSCI was primarily based on its simplicity in comparison to other flow languages and the fact that its standardisation was finished. It was important to chose a language that includes much less effort than manual wrapper implementation. The next chapter therefore describes how the WSCI language is used within the RAGE framework to generate more complex methods than outlined in the previous chapter.

Chapter 7

More Complex Methods - RAGE Advanced

This Chapter describes the extension to cover more complex methods and introduces a use case to demonstrate the usage of RAGE and its configuration possibilities, i.e. it will give a more comprehensive guide to the application of RAGE for interface generation.

7.1 How to Realize More Complex Methods Using WSCI

The method shown in Listing 5.3 fits the needs for a first layer proxy in terms of atomic methods and avoidance of passing of complex datatypes. However, it is not an easy to use interface as it consists of a whole lot of methods. The next step provided by RAGE is to reduce the number of methods but equip them with more functionality. One of those more complex methods could consist of the following atomic parts:

- Get an object of type X
- Call a method of object *X* that delivers an object *Y*
- Retrieve a (primitive) value from one of *Y's* methods

As this method consists of two methods from layer one there is only one message needed to be exchanged.

Listing 7.1: Example Method of Generated Layer Two Interface

```
public boolean NameOfProcess(long XID){
  boolean retValue = false;
  long id0 = X_getY(XID);
```

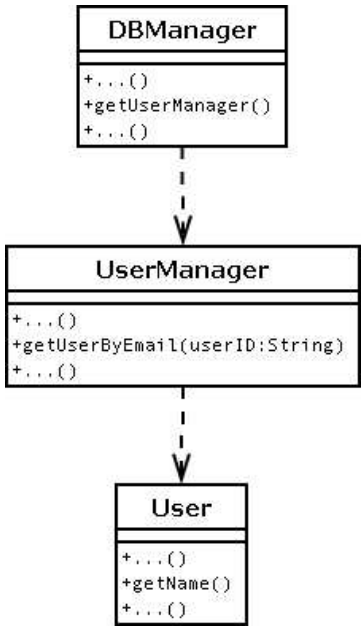

Figure 7.1: (Simplified) MAMA Class Structure

```
  retVal = Y_whatEver(id0);
  return retVal;
}
```

Listing 7.1 shows how such an aggregation of methods could look like. The next sections will cover how and in how far the generation of such interfaces is possible. Those sections will also explain the presented solutions based on an example and in greater detail.

7.2 Introducing the MAMA-Use Case

The MAMA software package will be used to explain many concepts throughout the upcoming sections. To get an idea of how it works, this section will introduce the relevant parts of it. MAMA is written in Java and designed as a framework for multimedia metadata archiving. For our purpose it is irrelevant what it is used for, but some details about the implementation are worth to know. Figure 7.1 shows a subset of the structure of the MAMA software package as UML class diagram. Any non-relevant methods, parameters and return values are ignored for reasons of simplification. The core of MAMA is the class *DBManager* which is the main entry point to the API. Every functionality can be accessed via this class, that is implemented as singleton. The upcoming ex-

amples are based on the user-related parts that mainly consist of the classes *UserManager* and *User*. As shown in Figure 7.1 to find out the name of a user stored in the database one has to:

- Instantiate a *DBManager*
- Call its *getUserManager* method to retrieve an *UserManager* object
- Call the *UserManager's getUserByEmail* method
- And finally call the *getName* method of the previously obtained *User*

with the corresponding parameters. Listing 7.2 shows the Java code needed to get the name of a specific user (note that this is an example that only works on the machine MAMA is installed on, so this listing does not contain code for remote access).

Listing 7.2: How to Retrieve the Name of a Specific User

```
class example{
  public static void main(String[] args){
    DBManager dbm = new DBManager();
    UserManager um = dbm.getUserManager();
    User u = um.getUserByEmail("admin@admin.com");
    u.getName();
  }
}
```

A typical use-case for the RAGE application would be to automatically generate code that provides equivalent functionality as Listing 7.2 from the compiled classes corresponding to the UML diagram of Figure 7.1. The next sections will explain how such an interface can be generated automatically in greater detail.

7.3 The Layer One Proxy Generation

The input for the layer one proxy is a compiled Java package of the software project the proxy shall be generated for. RAGE parses every class in the Java classpath and generates code for all classes that match a specified package name. An XML-config file is used to limit the parsed classes and methods (and, as a consequence, the the number of methods in the generated interface class).

7.3.1 The Configuration Possibilities

The RAGE configuration file includes settings for:

- Which packages to include (which packages interfaces should be created for)

- Which classes and methods to include (from those packages)
- Which methods are treated as constructors (important for singleton implementations in the source package that are realized via static class methods)
- Names of outputclass and outputpackage

Note that all of those classes and packages have to be in the classpath of the Java Virtual Machine. All entries are based on a regular-expressions-like syntax and stored in an XML-file, an example is shown in Listing 7.3.

Listing 7.3: Example config File for RAGE Layer One

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xnoNamespaceSchemaLocationion="configXMLSchema.xsd">

  <output package="testpackage" class="MAMAProxy" />
    <package pattern="edu.univie.mama.*">
      <class pattern="DBManager">
        <method pattern="*" />
      </class>
      <class pattern="UserManager">
        <method pattern="getUserByID" />
        <method pattern="getUserByEmail" />
      </class>
      <class pattern="User">
        <method pattern="*" />
      </class>
    </package>
    <package pattern="java.util.*">
      <class pattern="Vector">
        <method pattern="toString" />
      </class>
    </package>
    <createMethod>getInstance</createMethod>
    <freeText>
        public void testMethod(){
                System.out.println("test");
        }
    </freeText>
</configuration>
```

RAGE, if used with this configuration file, will generate code for:

- Every method of the class *DBManager*

- Two methods of the class *UserManager* (*getUserByEmail* and *getUserByID*
- Every method of the class *User*

that are part of the *edu.univie.mama.** package. Moreover it would generate the method *toString* for the class *Vector* which is part of the *java.util* package. Additionally, a section of free text can be specified, its contents will be copied to the generated output class [1]. As discussed in Section 5.4, the main problem of this approach is the granularity of the resulting methods in the interface file. This example would generate about 80 methods for both the MAMA API and the class *Vector* of the JDK. The next section covers the generation of more complex methods using WSCI as configuration mechanism.

7.4 WSCI Interface Generation (RAGE Layer Two)

This section describes the RAGE layer two functionality. The second layer of RAGE takes any interface generated by layer one and a configuration file that specifies which methods could be summarized and their order. It generates another interface class consisting of more complex methods than the ones from layer one. The goal is to copy all needed simple methods from layer one and to enrich them with more complex ones. In other words: the interface generated from layer two uses the interface from layer one, but can substitute it, because it copies all functionality. If the layer two functionality of RAGE is used, the only interface to be deployed is the layer two interface.

7.4.1 A Closer Look at the WSCI-Functionality of RAGE

The configuration of layer two consists of a .wsci file containing the information about which methods should be combined. In our case the WSCI file is detached from the WSDL information, because the WSCI information is not used to help client programmers understanding the business processes behind the invoked Web service but for internal choreography (see Section 6.3.1. A WSCI config file configuration. It consists of one or more *interface* elements which itself consist of one or more *process* elements. Every process represents a method in the target interface. Each method/process specifies which methods from layer one shall be used. So the *name* attribute of the *process* element denotes the method name in the resulting class. Every *action* element inside *process* denotes a method call to be included in the generated interface.

Listing 7.4: Example Config File for RAGE Layer Two (WSCI-layer)

[1] The free text section must contain valid Java code as it is going to be compiled as part of the resulting interface.

```
<wsci:interface name="MAMAMainInterface"
                xmlns:wsci="WSCI-NS">
  <wsci:process name="createDBManager">
    <wsci:sequence>
      <wsci:action
            operation="createDBManager">
      </wsci:action>
    </wsci:sequence>
  </wsci:process>
  <wsci:process name="shutdownDBManager">
    <wsci:sequence>
      <wsci:action
            operation="DBManager_shutdown">
      </wsci:action>
    </wsci:sequence>
  </wsci:process>
  <wsci:process name="getUserManagerComplete">
    <wsci:sequence>
      <wsci:action
            operation="createDBManager">
      </wsci:action>
      <wsci:action
            operation="DBManager_startReadTransaction">
      </wsci:action>
      <wsci:action
            operation="DBManager_getUserManager">
      </wsci:action>
      <wsci:action
            operation="DBManager_commitReadTransaction">
      </wsci:action>
      <wsci:action
            operation="DBManager_shutdown">
      </wsci:action>
    </wsci:sequence>
  </wsci:process>
  <wsci:process name="getUserManagerPartial">
    <wsci:sequence>
      <wsci:action
            operation="DBManager_startReadTransaction">
      </wsci:action>
      <wsci:action
            operation="DBManager_getUserManager">
      </wsci:action>
      <wsci:action
            operation="DBManager_commitReadTransaction">
      </wsci:action>
```

```
    </wsci:sequence>
  </wsci:process>
</wsci:interface>
```

Listing 7.4 defines three methods to be created and five existing methods from layer one to be referenced to. It includes two simple atomic methods that only contain one method from layer one:

- *createDBManager*
- *shutdownDBManager*

These methods are taken over directly from the layer one interface. The next entries handle more complex methods, such methods are the main motivation for layer two. In this case:

- *getUserManagerComplete*
- *getUserManagerPartial*

Those methods stand for similar functionality, though have methodical differences. The method *getUserManagerPartial* needs the ID of an already created *DBManager*. By contrast, *getUserManagerComplete* itself creates a *DBManager* and therefore does not need the ID as parameter. This example shows that the configuration allows to step in at every point of dependencies. Whenever one needed class is left out it is substituted by parameter IDs.

Listing 7.5: Resulting Interface from Layer Two (WSCI-layer)

```
package testpackage;

/**
* Generated on Apr 28, 2004
* by RAGE as part of WSCIProcessGenerator
* Dimitrij Denissenko & Robert Neumayer
* univie, 2004
*/

public class MAMAWSCIProxy {

testpackage.MAMAProxy mc = new testpackage.MAMAProxy ();

  public long createDBManager() {
    long retVal = 0;
    long id0 = mc.createDBManager();
    retVal = id0;
    return retVal;
  }
  public void shutdownDBManager(long idPar0) {
    mc.DBManager_shutdown(idPar0);
```

```
  }
  public long getUserManagerComplete() {
    long retVal = 0;
    long id0 = mc.createDBManager();
    mc.DBManager_startReadTransaction(id0);
    long id1 = mc.DBManager_getUserManager(id0);
    retVal = id1;
    mc.DBManager_commitReadTransaction(id0);
    mc.DBManager_shutdown(id0);
    return retVal;
  }
  public long getUserManagerPartial(long idPar0) {
    long retVal = 0;
    mc.DBManager_startReadTransaction(idPar0);
    long id0 = mc.DBManager_getUserManager(idPar0);
    retVal = id0;
    mc.DBManager_commitReadTransaction(idPar0);
    return retVal;
  }
}
```

Listing 7.5 presents the generated interface for the configuration from Listing 7.4. There is an interesting difference between *createDBManager* and *shutdownDBManager*, which demonstrates the differences between methods with return type void and constructors, that normally return objects. The object that would be returned by the constructor of *DBManager* is presented by an ID (this is actually done in layer one). Whereas the ethod *shutdownDBManager* doesn't return anything as the underlying method is of type void too. The methods *getUserManagerPartial* and *getUserManagerComplete* illustrate the differences in instantiation hierarchies mentioned before. The method *getUserManagerComplete* includes the calls to all needed methods (both *createDBManager* and *getUserManager* that finally returns the desired ID). On the contrary *getUserManagerPartial* doesn't contain an explicit instantiation of a *DBManager*, therefore the user needs to specify the ID of an existing one (that was created previously). The method first gets the existing *DBManager* out of the object cache and then performs the same actions as the *getUseManagerComplete* method. This mechanism on the one hand allows complete method calls like *getUserManagerComplete*, on the other hand makes possible very modular calls that allow the client to imitate object like behavior. The client programmer can handle IDs like objects and pass it to several methods, hence RAGE layer two can make very independent access to any software system possible.

7.4.2 A Detailed Example

This example will help to understand how the code generation in RAGE layer two works in detail.There are two important inputs to layer two:

- A WSCI config file
- A proxy generated by RAGE layer one

The config file is an XML-file nearly following the WSCI syntax (see Section 7.5.2 for details), the proxy from layer one code generation needs to be a compiled Java class (a .class file). The output of a layer two transformation is another interface class consisting of less, but more complex methods.Listing 7.6 shows another WSCI config file specifying only one method, but a more complex one than in the previous examples.

Listing 7.6: Config File for RAGE Layer Two (User example)

```
<wsci:interface name="MAMAUserInterface"
                xmlns:wsci="WSCI-NS">
  <wsci:process name="getUserNameByEmail">
    <wsci:sequence>
      <!-- instantiate the DBManager -->
      <wsci:action operation="createDBManager">
      </wsci:action>
      <!-- handle to Usermanager -->
      <wsci:action
            operation="DBManager_startReadTransaction">
      </wsci:action>
      <wsci:action
            operation="DBManager_getUserManager">
      </wsci:action>
      <!-- getting User by Email -->
      <wsci:action
            operation="UserManager_getUserByEmail">
      </wsci:action>
      <!-- getting the user name -->
      <wsci:action
            operation="User_getName">
      </wsci:action>
      <wsci:action
            operation="DBManager_commitReadTransaction">
      </wsci:action>
      <wsci:action operation="DBManager_shutdown">
      </wsci:action>
    </wsci:sequence>
  </wsci:process>
</wsci:interface>
```

Listing 7.7 shows the corresponding methods of the layer one proxy. According to the config file from Listing 7.6 all of those methods shall be aggregated to one single method.

Listing 7.7: Corresponding Layer One Proxy (User example)

```
public long createDBManager() {
  long id = oc.add(new DBManager());
  return id;
}
public void DBManager_shutdown(long dbmanagerID){
  DBManager o = (DBManager)oc.get(dbmanagerID);
  o.shutdown();
  oc.set(dbmanagerID, o);
}
public void DBManager_startReadTransaction(long dbmanID){
  DBManager o = (DBManager)oc.get(dbmanagerID);
  o.startReadTransaction();
  oc.set(dbmanagerID, o);
}
public void DBManager_commitReadTransactio(long dbmanID){
  DBManager o = (DBManager)oc.get(dbmanagerID);
  o.commitReadTransaction();
  oc.set(dbmanagerID, o);
}
public long DBManager_getUserManager(long dbmanagerID){
  DBManager o = (DBManager)oc.get(dbmanagerID);
  UserManager rv = o.getUserManager();
  oc.set(dbmanagerID, o);
  long id = oc.add(rv);
  return id;
}
public long
       UserManager_getUserByEmail(long umID,String s1){
  UserManager o = (UserManager)oc.get(usermanagerID);
  User rv = o.getUserByEmail(s1);
  oc.set(usermanagerID, o);
  long id = oc.add(rv);
  return id;
}
public java.lang.String User_getName(long userID){
  User o = (User)oc.get(userID);
  String rv = o.getName();
  oc.set(userID, o);
  return rv;
}
```

In order to explain the algorithm, some concepts have to be introduced:

- The *NeededIDType* of a given method is used to describe methods from a layer one proxy. It is the class the method belongs to, e.g. the method *DBManager_startReadTransaction* belongs to the class *DBManager*, hence it needs an ID for an object of type *DBManager* as an argument

- A *ReturnedIDType of a given method* is the class this method returns an object of. The method *DBManager_getUserManager* obviously returns an ID for an object of type *UserManager*.

- An *action* is the smallest part of any process modeled in a WSCI configuration (i.e. the action elements seen in Listing7.6) and directly refers to a method of a layer one proxy

- A *method* or *resulting method* is the method generated in the layer two interface file (created out of the corresponding process element of the wsci config file, see Listing 7.6

The algorithm for this transformation works as follows:

- Determine which IDs will be needed for all method calls (in this example: an ID of each a *DBManager*, *UserManager* and *User* object where none of them is needed as a part of the resulting method, because all three of them are created within the process

- Initialize a variable for the return value (*String retVal = ""* is the correct initialization in this case, in general it depends on the type)

- For every action in the config file ($\equiv$ method from the layer one interface) do the following:

  - determine its parameters
  - determine its return value
  - determine if it is a method that returns an object[2]
  - if so, assign the return value to an ID and store the pair (idvariablename, type) to a *HashMap*

- Processing (starting again), code generation

  - Add a parameter to the resulting method for every type not created within itself (there is no ID parameters in this case, because all of the needed methods are created within the process)
  - Create a long id*X* for every method within the process that returns an object (e.g. *long id0 = createDBManager()* in Listing 7.8)

[2]Any method that is a $createXXX$ method (calling a constructor) or fits the pattern $get*By*$, see Section 7.6.2 for details

- For every method that needs an object as parameter, search the already created ones and use the corresponding id*X* method and use it as id parameter of that method, i.e. take the id that is stored for the type *DBManager* (e.g. *id0* for the *DBManager_startReadTransaction in this example* of Listing 7.8), if none is found use the id parameters of the resulting method (not needed here, because everything is created within the resulting method)
- If the last action of the process that has a return value is reached, it is used in the return statement of the whole method (*retVal = mc.User_getName(id2)* in this case)
- The return type of the resulting method is only void if *all* of its actions return void [3]

The resulting layer two proxy is presented in Listing 7.8.

Listing 7.8: Resulting WSCI Proxy (User example)

```
package testpackage;

/**
* Generated on June 13, 2004
* by RAGE as part of WSCIProcessGenerator
* Dimitrij Denissenko & Robert Neumayer
* univie, 2004
*/

public class MAMAWSCIProxy {

testpackage.MAMAProxy mc = new testpackage.MAMAProxy ();

  public String getUserNameByEmail(String par0){
    java.lang.String retVal = "";
    long id0 = mc.createDBManager();
    mc.DBManager_startReadTransaction(id0);
    long id1 = mc.DBManager_getUserManager(id0);
    long id2 = mc.UserManager_getUserByEmail(id1, par0);
    retVal = mc.User_getName(id2);
    mc.DBManager_commitReadTransaction(id0);
    mc.DBManager_shutdown(id0);
    return retVal;
}
```

The method *getUserNameByEmail* combines seven methods from the layer one proxy. It is far more easier to use because the client programmer only has to call one method instead of seven and the even more important advantage is

[3] i.e. every resulting wsci method returns the ID of the object last used within it or, if a method with a return value is called, returns the last return values of all methods used

that only one message needs to be exchanged between the client and the Web service. This also substantially decreases the lines of code needed for a client program as there is only one instead of seven remote methods to be called by the client application. This example shall have given an idea about in how far the use of a WSCI-like choreography or process definition can shape a Web service interface to a given interface. Of course this simple approach is limited in several ways which will be described in the next section.

7.5 Constraints of the WSCI Approach

RAGE layer two assumes that the client programmer won't call a method for one object before the constructor is called (e.g. one needs to instantiate a *DB-Manager* before calling *startReadTransaction*. This could again be specified by WSCI, as it is the actual field for using WSCI (see Section 6.3.1). Furthermore the use-case presented in Section 7.4.2 is nearly the most complex transformation possible using that simple type of configuration (only specifying processes consisting of actions). Although WSCI offers far more possibilities of describing the dynamic behavior of a Web service, there is a case, that it does not cover directly, which is covered by rhe next section in greater detail.

7.5.1 Limitations of the WSCI Standard and Problems in the Context of RAGE

Pure WSCI only allows to specify references to methods as references to WSDL operationnames. Those operations do not explicitly support the definition of overloaded methods. Listing 7.9 shows how overloaded methods are represented by the WSDL operations construct. The two methods named *createMediaType* only slightly differ in their *name* attribute of the <*wsdl:input wsdl:input*/ > element.

Listing 7.9: (Simplified) WSDL description of a Layer One Interface

```
<wsdl:operation name="createMediaType">
  <wsdlsoap:operation soapAction=""/>
  <wsdl:input name="createMediaTypeRequestest">      *
    <wsdlsoap:body/>
  </wsdl:input>
  <wsdl:output nam2="createMediaTypeResponsense">
    <wsdlsoap:body/>
  </wsdl:output>
</wsdl:operation>
<wsdl:operation name="createMediaType">
  <wsdlsoap:operation soapAction=""/>
  <wsdl:input name="createMediaTypeRequestest1">     *
    <wsdlsoap:body/>
```

```
  </wsdl:input>
  <wsdl:output name="createMediaTypeResponse1">
    <wsdlsoap:body/>
  </wsdl:output>
</wsdl:operation>
```

This is the reference to a different <*wsdl:message wsdl:message*/ > element in the same WSDL-file. Listing 7.10 shows these two referenced message elements. This makes clear that the name attribute of the operation element alone can not be used to differentiate between such methods distinctly.

Listing 7.10: WSDL Message Descriptions Representing Different Parameters

```
<wsdl:message name="createMediaTypeRequest1">
  <wsdl:part name="in0" type="xsd:long"/>
</wsdl:message>

<wsdl:message name="createMediaTypeRequest">
  <wsdl:part name="in0" type="xsd:string"/>
  <wsdl:part name="in1" type="xsd:string"/>
</wsdl:message>
```

This ambiguity leads to problems when running RAGE, so the next section discusses a possible solution.

7.5.2 Extension of WSCI

The RAGE workaround is an extension to the WSCI syntax which is shown in Listing 7.11 and aims to fix this problem.

Listing 7.11: Extension of WSCI Syntax

```
<wsci:process name="createMediaType"
              instantiation="message">
  <wsci:sequence>
    <!-- starting the update transaction -->
    <wsci:action
          operation="DBManager_startUpdateTransaction">
    </wsci:action>
    <!-- instantiating the mediatype -->
    <wsci:action operation="createMediaType">
      <extension:methodIdentifier>
        <extension:parameter>
          java.lang.String
        </extension:parameter>
        <extension:parameter>
          java.lang.String
        </extension:parameter>
      </extension:methodIdentifier>
```

```
    </wsci:action>
    <wsci:action
          operation="DBManager_commitUpdateTransaction">
    </wsci:action>
  </wsci:sequence>
</wsci:process>
```

This extension consists of two elements that are needed to solve this problem, the first one, <*extension:methodIdentifier*/ > is used for any methods with ambiguous definitions. Within this element all parameters of the given methods, two strings in this example, are specified in a <*ext:parameter*/> element. A WSCI file with this extensions can handle any ambiguities concerning overloaded methods.

7.6 RAGE-Compatible Software

This section shall give some guidelines that help the programmer of a software system to get it best-working with RAGE. It describes possible problems that could occur in the different layers.

7.6.1 RAGE Layer One

In theory RAGE layer one should handle even very badly designed classes and is designed very robust. If there is a lot of *Vectors* and *HashTables* to be used as parameters and return values, layer one creates a lot of delegating methods [4], but it works. Remote control of classes that are part of the JDK is more complicated than the remote control of self-designed classes. The less classes from JDK are used the better and easier the generated interface will be to use.

7.6.2 RAGE Layer Two

As a crucial part of the layer two functionality is to find out which methods will give back IDs and which not, the programmer of the source implementation has to follow certain rules concerning the naming of methods, they should either be named:

- getClassname or
- getClassnameByWhatever

The layer two implementation parses a method and needs to find out if it returns another object, of course only by means of IDs, or just a normal simple datatype. All methods which have names starting with *create** are identified as

[4] Rather the config file has to be changed to generate a lot of delegating methods

returning an ID, as those methods cover the constructors of the source classes, because this suffix is created by RAGE layer one. This determination is more difficult for other methods, where it is done by parsing the method name only, as there would be no other chance to differentiate between the normal datatype long and an ID of type long. This could be improved by checking if the name of the object a given method returns can be instantiated [5]. This would on the one hand lessen the probability to mishandle the return type of a normal method, on the other hand it would slow down the performance of the application. Note that such an extension would not change the main problem of differentiating between long as datatype and long as ID type. RAGE layer two handles every method that follows the above mentioned naming conventions as a method returning a reference to an object.

7.6.3 RAGE Layer Three

RAGE Layer Three can be applied to any layer one proxy and to any layer two proxy. The layer one compatibility issues from the fact that RMI outreaches Web services in terms of datatypes (and those interfaces generated by RAGE are customized for Web service use). It can also be applied to any layer two interfaces, because they are limited by the naming conventions discussed in tha last section anyway. For further details about RAGE's layer three consult [15].

[5] In terms of the inclusion of a *class.forName()* call for the retrieved classname. E.g. if a the method *getObject* is supposed to return an object of class *Object*, check if *class.forName(Object)* works to be sure that Object references an existing class.

Chapter 8

Discussion and Outlook

The concepts described in the last chapters can be used to create quite satisfying remote interfaces for Java applications. The main usage scenario for RAGE is in the development process of a software system. RAGE can automatically generate interfaces that can be used to test a still growing API in terms of remote feasibility and rapid prototype development. The main purpose of RAGE will probably be to check if a given software package is feasible for remote access at all. The development of the remote interface could possibly start after the system had worked with RAGE. Moreover RAGE could provide extra remoteinterfaces that are meant as extension of the actual ones, just to offer an existing API to all SOAP-compatible languages.

The main advantages of RAGE can be summarized as:

- Fits to any Java package
- Applicable to any needed packages (see Vector example on page (46)
- Simple configuration and therefore
- Rapid generation of interfaces for testing purposes
- Extensibility because of its modular architecture
- Modular architecture by means of different, detached layers
- Interoperability, i.e. very low tech interfaces resulting in hardly any limitation to client programming languages

8.1 Possible improvements - Future Work

The following improvements could lead RAGE out of its development tool status closer to the use in a real production like scenario, but probably won't guarantee this. This proposals could be used when improving RAGE and are meant as ideas of new modules and extensions.

- Security concepts, at its current state RAGE does not provide for any user authentication or other security functionality. There is no guaranteed order of events like authentication before accessing the remote API
- Error and exception handling, as exception handling is a general problem of remote applications. The easiest method is to introduce a convention to take some standard return values as indicators of failure, like:
  - Return value 0 for any methods that use numerical datatypes (e.g. an object has the id 0 when it could not have been created)
  - Return value "" for the datatype String (e.g. a name of a user can not be retrieved, note that this leads to problems if the specific user does not have a name assigned anyway)
  - ...

  Although this is a rather primitive kind of exception handling and is, of course, inferior to the Java exception handling due to the limitation of datatypes, it could substantially improve the application.
- Extend functionality to other programming languages, this would mean to nearly completely redesign the RAGE software, but the approach of caching objects on the server side could in theory applied to any other programming language. Possible problems could arise in the area of parsing compiled binaries which would rather mean to recompile them and parse the source code (even if the porting to programming languages that to produce compiled binaries,may lead to problems, using Microsoft's C sharp [14] would probably be rather simple because of its precompiled intermediate language code similar to Java).
- Add a layer for CORBA functionality, similar to layer three for RMI to satisfy more of the common remote mechanisms
- Implement a better type checking mechanism for the layer two.
- Consider the use of more complex choreography description languages, see Chapter6
  - Especially the usage of the W3C Choreography Language may be of great interest as the W3C is leading in specifying Web services standards and technologies
- Redesign RAGE to meet the WSI requirements (known issue is duplicate operation names)
- Add functionality to collect return values (fill up an arrray with return values of several methods)

List of Figures

Listings

Bibliography

[1] Conan C. Albrecht. How clean is the future of soap? *Commun. ACM*, 47(2):66–68, 2004.

[2] Mikio Aoyama, Sanjiva Weerawarana, Hiroshi Maruyama, Clemens Szyperski, Kevin Sullivan, and Doug Lea. Web services engineering: promises and challenges. In *Proceedings of the 24th international conference on Software engineering*, pages 647–648. ACM Press, 2002.

[3] David Booth, Hugo Haas, and Francis McCabe. *Web Services Architecture*. World Wide Web Consortium, w3c note edition, February 2004. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`, visited August 2004.

[4] Workflow Management Coalition. *XML Processing Description Language - Process Definition Interchange 1.0*, October 2002. `http://www.wfmc.org/standards/docs/TC-1025\_10\_xpdl\_102502.pdf`, visited November 2004.

[5] Frank Cohen. Complex datatypes in soap-based web services. Internet, May 2003. `http://www-106.ibm.com/developerworks/webservices/library/ws-complex.ht%ml`, visited August 2004.

[6] Frank Cohen. Discover soap encoding's impact on web service performance. Internet, March 2003. `http://www-106.ibm.com/developerworks/webservices/library/ws-soapenc/`, visited August 2004.

[7] World Wide Web Consortium. Web services choreography working group. `http://www.w3.org/2002/ws/chor/`, visited August 2004.

[8] World Wide Web Consortium. The world wide web consortium official website, www.w3.org. `http://www.w3.org`, visited June 2004.

[9] World Wide Web Consortium. The world wide web consortium protocol working group. `http://www.w3.org/2000/xp/Group/`, visited October 2004.

[10] World Wide Web Consortium. Soap version 1.2 part 0: Primer, March 2004. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/, visited October 2004.

[11] World Wide Web Consortium. Soap version 1.2 part 1: Messaging framework, March 2004. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/, visited October 2004.

[12] World Wide Web Consortium. Soap version 1.2 part 2: Adjuncts, March 2004. http://www.w3.org/TR/2003/REC-soap12-part2-20030624/, visited June 2004.

[13] Microsoft Corporation. *Microsoft .NET*, 2001. http://www.microsoft.com/net, visited August 2004.

[14] Microsoft Corporation. *C# Language Specification 1.2*, 2003. http://msdn.microsoft.com/vcsharp/team/language/default.aspx, visited August 2004.

[15] Dimitrij Denissenko. Entwicklung verteilter anwendungssysteme. Master's thesis, University of Vienna, 2004.

[16] Dimitrij Denissenko and Robert Neumayer. Rage, the remote access generator engine. http://www.unet.univie.ac.at/\~a0003208/rage, last update December 2004.

[17] Assaf Arkin et al. The web services choreography interface 1.0. Technical report, BEA Systems, Intalio, SAP, Sun Microsystems, 2002. http://www.w3.org/wsci, visited June 2004.

[18] Chris Ferris. First look at the ws-i basic profile. Internet, October 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-basicprof.%html, visited October 2004.

[19] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration, 2002. http://citeseer.ist.psu.edu/foster02physiology.html, visited June 2004.

[20] Apache Software Foundation. Axis, implementation of soap protocol. http://ws.apache.org/axis/, visited June 2004.

[21] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM Press, 2004.

[22] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, and Dennis Gannon. Requirements for and evaluation of rmi protocols for scientific computing. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 61. IEEE Computer Society, 2000.

[23] Sun Microsystems Inc. Java remote method invocation (java rmi). `http://java.sun.com/products/jdk/rmi/`, visited August 2004.

[24] Sun Microsystems Inc. Java technology. `http://java.sun.com/`, visited August 2004.

[25] Business Process Management Initiative. *BPML, the Business Process Modeling Language*, March 2001. `http://www.bpmi.org/BPML.htm`, visited November 2004.

[26] Kurt Kanaskie et al. Jim Clark, Cory Casanave. *Business Process Specification Schema, 1.01*. ebXML, OASIS, May 2001. `http://www.ebxml.org/specs/ebBPSS.pdf`, visited August 2004.

[27] Matjaz B. Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, and Ivan Vezocnik. Java rmi, rmi tunneling and web services comparison and performance analysis. *SIGPLAN Not.*, 39(5):58–65, 2004.

[28] Martin Gudgin et al. Keith Ballinger, David Ehnebuske. Basic profile version 1.0. `http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html`, visited August 2004.

[29] S. Kelly and S. Ramamoorthi. Requirements for ipsec remote access scenarios, 2003.

[30] Paul Kulchenko and Byrne Reese. Soap::lite - perl client and server side soap implementation. `http://www.soaplite.com/`, visited August 2004.

[31] Prof. Dr. Frank Leymann. *Web Servics Flow Language, (WSFL 1.0)*. IBM, May 2001. `http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`, visited August 2004.

[32] Microsoft Corporation. Dcom technical overview. Technical report, Microsoft Corporation, 1996.

[33] Roberto Chinnici (Sun Microsystems) and Martin Gudgin (Microsoft). *Web Services Description Language (WSDL) 2.0*. World Wide Web Consortium, w3c working draft edition, March 2004. `http://www.w3.org/TR/2004/WD-wsdl20-20040326`, visited June 2004.

[34] Greg Ritzinger Nickolaos Kavantzas, David Burdett. *Web Services Choreography Description Language Version 1.0*. World Wide Web Consortium, w3c working draft edition, April 2004. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/`, visited August 2004.

[35] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 2.5 edition, September 2001. `http://www.omg.org/technology/documents/corba\_spec\_catalog\\.htm\#COR%BA\_IIOP`, visited August 2004.

[36] Web Services Interoperability Organization. Web services interoperability organization. http://www.ws-i.org/, visited August 2004.

[37] Web Services Interoperability Organization. Ws-i testing tools (java version) - final. http://www.ws-i.org/Testing/Tools/2004/01/WSI_Test_Java_01.00.01_bi%n.zip, visited August 2004.

[38] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, 2001.

[39] Satish Thatte. *XLANG, Web Services for Business Process Design*. Microsoft, initial public draft release edition, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, visited August 2004.

[40] Hitesh Dholakia et al. Tony Andrews, Francisco Curbera. *Business Process Execution Language for Web Services Version 1.1*. IBM, Microsoft, May 2003. http://www-106.ibm.com/developerworks/library/ws-bpel/, visited August 2004.

[41] UDDI. *Universal Description, Discovery, and Integration of Business for the Web*, October 2001. http://www.uddi.org.

[42] Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Web service composition languages: Old wine in new bottles? In *Proceedings of the 29th Conference on EUROMICRO*, page 298. IEEE Computer Society, 2003.

[43] Larry Wall. Practical extension and report language, perl 5.8.4, 1987. http://www.perl.org, visited August 2004.

[44] World Wide Web Consortium. *Extensible Markup Language, (XML)*. http://www.w3.org/XML/, visited August 2004.

[45] World Wide Web Consortium. *XML Schema*. http://www.w3.org/XML/Schema, visited August 2004.

[46] Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. Developing web services choreography standards - the case of rest vs. soap. *Decision Support Systems*, 37(2004), 2004.