9.2 JAX-RPC Overview

JAX-RPC provides a generic mechanism that enables developers to create and access Web services by using XML-based Remote Procedure Calls. While such Web services can communicate by using any transport protocol, the current release of the JAX-RPC Reference Implementation (version 0.7) uses SOAP as the application protocol and HTTP as the underlying transport protocol. Future versions likely will support other transport protocols as they become available.

When Web-service providers publish their Web services to XML registries (e.g., UDDI registries or ebXML registries), they may provide service interfaces or WSDL definitions for these services. Refer to Chapter 7, Web Services Description Language (WSDL) for more information on WSDL. The JAX-RPC specification defines a mapping of Java types (e.g., `int`, `String`, classes that adhere to the JavaBeans pattern) to WSDL definitions. When a client locates a service in an XML registry, the client retrieves the WSDL definition to get the service interface definition. To be able to access the service using Java, the service clients must transform the WSDL definitions to Java types.

Figure 9.1 shows the JAX-RPC architecture. The service side contains a JAX-RPC service runtime environment and a service endpoint. The client side contains a JAX-RPC client runtime environment and a client application. The remote procedure calls use an XML-based protocol, such as SOAP, as the application protocol, and they use HTTP as the transport protocol. The JAX-RPC client and service runtime systems are responsible for sending and processing the remote method call and response, respectively. The JAX-RPC client creates a SOAP message to invoke the remote method and the JAX-RPC service runtime transforms the SOAP message to a Java method call and dispatches the method call to the service endpoint.
Before JAX-RPC, Remote Method Invocation (RMI) was the predominant RPC mechanism for Java. RMI allows Java programs to transfer complete Java objects over networks using Java’s object-serialization mechanism. Since RMI can be used to make remote procedure calls over the Internet, developers may wonder why they might use JAX-RPC, which seems to provide similar functionality.

As with RPC, both RMI and JAX-RPC handle the marshalling/unmarshalling of data across the network. Both RMI and JAX-RPC provide APIs to transmit and receive data. The primary differences between RMI and JAX-RPC are:

1. JAX-RPC inherits SOAP’s and WSDL’s interoperability, which enables Java applications to invoke Web services that execute on non-Java platforms and non-Java applications to invoke Web services that execute on Java platforms. RMI supports only Java-to-Java distributed communication. The service client needs only the WSDL to access the Web service. [Note: RMI/IIOP also provides interoperability with non-Java applications.]

2. RMI can transfer complete Java objects, while JAX-RPC is limited to a set of supported Java types. (See Section 9.3.1 for a discussion of JAX-RPC supported Java types).

JAX-RPC hides the details of SOAP from the developer because the JAX-RPC service/client runtime environments perform the mapping between remote method calls and SOAP messages. The JAX-RPC runtime system also provides APIs for accessing Web services via static stubs (local objects that represent the remote services) and for invoking Web services dynamically through the Dynamic Invocation Interface (DII). We discuss these APIs in detail in Section 9.3.5. [Note: Dynamic Proxies will be supported in the final release of JWSDP.]
9.3 Simple Web Service: Vote Service

In this section, we present a simple JAX-RPC Web service that tallies votes for the users’ favorite programming languages. The four major steps in this example include:

1. Defining a service interface that declares methods that clients can invoke on the remote service.
2. Writing a Java class that implements the interface. [Note: By convention, the service implementation class has the same name as the interface and ends with Impl.]
3. Deploying the service to the Web server. In this example, we use Apache’s Tomcat, which is part of the Java Web Services Developer Pack (JWSDP) and is available free for download from:

   java.sun.com/webservices/download.html

4. Writing the client application that interacts with the service.

Before writing the example code, we will introduce the limitations on JAX-RPC supported Java types.

9.3.1 JAX-RPC Supported Java Types

JAX-RPC supports only a subset of Java types, because the data types transmitted by the remote procedure calls must map to XML data types. When a Web service receives a remote method call from its client, the JAX-RPC runtime service environment first transforms the XML representation of the call inputs to its corresponding Java type (this process also is known as deserialization), then passes the Java object to the service implementation to process the remote call. After the call is processed, the JAX-RPC runtime service environment transforms the return object to its XML representation (this process is also known as serialization), the XML representation of the return object is then sent back to the service client. This serialization/deserialization process happens to both client and service.

JAX-RPC supports Java primitive types and their corresponding wrapper classes. Figure 9.2 shows the mappings of Java primitive types and their wrapper classes to XML elements.

<table>
<thead>
<tr>
<th>Java primitive types and their wrapper class</th>
<th>XML elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean (Boolean)</td>
<td>xsd:boolean (soapenc:boolean)</td>
</tr>
<tr>
<td>byte (Byte)</td>
<td>xsd:byte (soapenc:byte)</td>
</tr>
<tr>
<td>double (Double)</td>
<td>xsd:double (soapenc:double)</td>
</tr>
<tr>
<td>float (Float)</td>
<td>xsd:float (soapenc:float)</td>
</tr>
<tr>
<td>int (Integer)</td>
<td>xsd:int (soapenc:int)</td>
</tr>
<tr>
<td>long (Long)</td>
<td>xsd:long (soapenc:long)</td>
</tr>
</tbody>
</table>

Fig. 9.2 Mappings of Java primitive types and their wrapper classes to XML data types.
JAX-RPC supports a subset of standard Java classes as well. Figure 9.3 shows the mappings of a subset of standard Java classes to XML elements.

In addition to the aforementioned supported types, JAX-RPC supports objects of Java classes that satisfy following conditions:
2. Must have a `public` no-argument constructor.
3. Public fields must be JAX-RPC supported Java types.
4. Must follow the JavaBeans `set` and `get` method design pattern.
5. Bean properties must be JAX-RPC supported Java types.

Finally, Java arrays also can be used in JAX-RPC, provided that the type of the array is one of the JAX-RPC supported types. JAX-RPC also supports multi-dimensional arrays.

### 9.3.2 Defining Vote Service Interface

The first step in the creation of a Web service with JAX-RPC is to define the remote interface that describes the remote methods through which the service client interacts with the service using JAX-RPC. There are some restrictions on the service interface definition:

2. Each public method must include `java.rmi.RemoteException` in its `throws` clause. The `throws` clause may include service specific exceptions.
3. No constant declarations are allowed.
4. All method parameters and return types must be JAX-RPC supported Java types.
To create a remote interface, define an interface that extends interface \texttt{java.rmi.Remote}. Interface \texttt{Remote} is a \textit{tagging interface}—it does not declare any methods, and therefore places no burden on the implementing class. An object of a class that implements interface \texttt{Remote} directly or indirectly is a \textit{remote object} and can be accessed—with appropriate security permissions—from any Java virtual machine that has a connection to the computer on which the remote object executes. Interface \texttt{Vote} (Fig. 9.4)—which extends interface \texttt{Remote} (line 9)—is the remote interface for our first JAX-RPC based Web service example. Line 12 declares method \texttt{addVote}, which clients can invoke to add votes for the users’ favorite programming languages. Note that although the \texttt{Vote} remote interface defines only one method, remote interfaces can declare multiple methods. A Web service must implement all methods declared in its remote interface.

\begin{verbatim}
// Fig. 9.4: Vote.java
// VoteService interface declares a method to add a vote and
// return vote information.
package com.deitel.jws.jaxrpc.service.vote;

// Java core packages
import java.rmi.*;

// Vote defines the service interface.
public interface Vote extends Remote {
  // obtain vote information from server
  public String addVote(String language) throws RemoteException;
}
\end{verbatim}

\textbf{9.3.3 Defining Vote Service Implementation}

After defining the remote interface, we define the service implementation. Class \texttt{VoteImpl} (Fig. 9.5) is the Web service endpoint that implements the \texttt{Vote} interface. The service client interacts with an object of class \texttt{VoteImpl} by invoking method \texttt{addVote} of interface \texttt{Vote}. Method \texttt{addVote} enables the client to add a vote to the database and obtain vote information.

Class \texttt{VoteImpl} implements remote interface \texttt{Vote} (line 10). Lines 15–76 implement method \texttt{addVote} of interface \texttt{Vote}. We use a Cloudscape database in this example to store the total number of votes for each programming language in the database. Line 21 loads the Cloudscape database driver.

Lines 24–25 of class \texttt{VoteImpl} declare and initialize \texttt{Connection} reference \texttt{connection} (package \texttt{java.sql}). The program initializes \texttt{connection} with the result of a call to \texttt{static} method \texttt{getConnection} of class \texttt{DriverManager}, which attempts to connect to the database specified by its URL argument. The URL \texttt{jdbc:cloudscape:rmi:languagesurvey} specifies the \textit{protocol} for communication (\texttt{jdbc}), the \textit{subprotocol} for communication (\texttt{cloudscape:rmi}) and the name of the database (\texttt{languagesurvey}).

Lines 29–33 invoke \texttt{Connection} method \texttt{prepareStatement} to create an SQL \texttt{PreparedStatement} for updating the number of votes for the client’s selected pro-
gramming language. Line 36 sets the parameter of `sqlUpdate` to the client specified language. After setting the parameter for the `PreparedStatement`, the program calls method `executeUpdate` of interface `PreparedStatement` to execute the `UPDATE` operation.

```java
// VoteImpl.java
// VoteImpl implements the Vote remote interface to provide
// a VoteService remote object.
package com.deitel.jws.jaxrpc.voteservice;

// Java core packages
import java.rmi.*;
import java.sql.*;

// Java XML packages
import javax.xml.rpc.server.ServiceLifecycle;
import javax.xml.rpc.JAXRPCException;

public class VoteImpl implements ServiceLifecycle, Vote {

    private Connection connection;
    private PreparedStatement sqlUpdate, sqlSelect;

    // set up database connection and prepare SQL statement
    public void init( Object context ) throws JAXRPCException {
        try {
            // load Cloudscape driver
            Class.forName( "COM.cloudscape.core.RmiJdbcDriver" );

            // connect to database
            connection = DriverManager.getConnection(
              "jdbc:cloudscape:rmi:languagesurvey" );

            // PreparedStatement to add one to vote total for a
            // specific animal
            sqlUpdate =
                connection.prepareStatement(
                    "UPDATE surveyresults SET vote = vote + 1 " +
                    "WHERE name = ?"
                );

            // PreparedStatement to obtain surveyresults table's data
            sqlSelect =
                connection.prepareStatement(
                    "SELECT name, vote " +
                    "FROM surveyresults ORDER BY vote DESC"
                );
        }
    }
}
```

**Fig. 9.5** Class `VoteImpl` implements `Vote` interface. (Part 1 of 3.)
// for any exception throw an JAXRPCException to indicate that the servlet is not currently available

```
catch ( Exception exception ) {
    exception.printStackTrace();
    throw new JAXRPCException( exception.getMessage() );
}
```

// end of method init

// implementation for interface Vote method addVote

```
public String addVote( String name ) throws RemoteException {
    // get votes count from database and update it
    try {
        // set parameter in sqlUpdate
        sqlUpdate.setString( 1 , name );
        // execute sqlUpdate statement
        sqlUpdate.executeUpdate();
        // execute sqlSelect statement
        ResultSet results = sqlSelect.executeQuery();
        StringBuffer voteInfo = new StringBuffer();
        // iterate ResultSet and prepare return string
        while ( results.next() ) {
            // append results to String voteInfo
            voteInfo.append( " " + results.getString( 1 ) );
            voteInfo.append( " " + results.getInt( 2 ) );
        }
        return voteInfo.toString();
    }
    // handle database exceptions by returning error to client
    catch ( Exception exception ) {
        exception.printStackTrace();
        return exception.getMessage();
    }
}
```

// end of method addVote

// close SQL statements and database when servlet terminates

```
public void destroy() {
    // attempt to close statements and database connection
    try {
        sqlUpdate.close();
        sqlSelect.close();
    }
```

Fig. 9.5 Class VoteImpl implements Vote interface. (Part 2 of 3.)
Lines 42–45 invoke Connection method `prepareStatement` to create an SQL `PreparedStatement` `sqlSelect` that selects all programming languages with the corresponding number of votes. The program calls method `executeQuery` of interface `PreparedStatement` to execute the `SELECT` operation. The execution results are stored in `ResultSet` `results`. Lines 52–59 process the `ResultSet` and store the results in `String` `voteInfo`. Line 66 returns vote information to the client.

### 9.3.4 Deploying Vote Service

Once we have defined the service interface and implementation, the next step is to deploy the Web service. The JAX-RPC reference implementation provides a tool—`xrpcc`—to generate stubs, ties (server-side objects that represent the services) and other service and client-side artifacts (such as a WSDL document).

**xrpcc Tool**

The `xrpcc` tool generates a WSDL document or a remote interface definition, depending on the command-line parameter. If the `xrpcc` tool is given an remote interface definition, it generates stubs, ties, a WSDL document and a server configuration file used during deployment. If the `xrpcc` tool is given a WSDL document, it generates stubs, ties, a server configuration file and the remote interface definition. Starting with a WSDL document is what most users will do to access a Web service published by another vendor. WSDL is the contract between the client and service. The stubs, ties, service and client-side artifacts are dictated by options `–client`, `–server` and `–both` of the `xrpcc` tool. We demonstrate the usage of all these options in the follow up examples. In this example, we use the `xrpcc` tool to generate the WSDL document based on the remote interface definition. In later examples, we use the `xrpcc` tool to generate the remote interface definition and client-side classes to access the Web service bases on the WSDL document.

To generate a WSDL document, the `xrpcc` tool reads an XML configuration file that lists remote interfaces. `VoteConfig.xml` (Fig. 9.6) is the configuration for our Vote service example. The `xrpcc` tool takes the `VoteConfig.xml` and generates the WSDL file and other service-side classes for the Vote service. The root element `configuration` contains one `rmi` elements that correspond to remote interfaces. Element `configuration` may have exact one `rmi` elements. Element `rmi` may have one or more `service` elements. Element `service` may have one or more `interface` elements. In our example, there is only one remote interface (lines 5–17). The `name` attribute of element `rmi` (line 5) indicates the model name, used in the name of the generated WSDL file. The
targetNamespace attribute specifies the target namespace for the generated WSDL document (line 6). The typeNamespace attribute specifies the target namespace within the types section of the WSDL document. Element service (lines 9–16) defines the service name, fully qualified package name and its interface. The name attribute of element service indicates the service name (line 9). The packageName attribute specifies the package name of the generated stubs, ties and other classes (line 10). The value of attribute packageName does not need to match the package name of any of the remote interface. Element interface (lines 12–15) defines the fully qualified name of the service interface via its attribute name and the fully qualified name of the service implementation via its attribute servantName. Each element interface defines a service port in the WSDL file.

When we compile the Vote and VoteImpl Java files, we specify the output directory to VoteOutput to separate the source code and the executable classes. Before running the xrpcc tool, we need to set the PATH environment variable to include \%JWSDP_HOME\%\bin directory, where \%JWSDP_HOME\% is the installation directory of JWSDP. VoteConfig.xml should be in your working directory. To generate the stubs, ties, a WSDL document and a server configuration file for the Vote service, use the command:

```
xrpcc -classpath VoteOutput -both -d VoteOutput VoteConfig.xml
```

Option classpath sets the xrpcc tool’s classpath to directory VoteOutput, which contains the service endpoint interface and implementation. [Note: xrpcc needs to access the service interface and implementation to generate corresponding files, so we add directory VoteOutput to option classpath]. Option both instructs the xrpcc tool to generate both server-side and client-side files. Option d specifies the output directory for the generated files. We also could generate the server-side files using option server and
the client-side files using option client. Other options are available. For more information on other options, please refer to

http://java.sun.com/webservices/docs/eal/tutorial/doc/
JAXRPCxrpcc.html

VoteService.wsdl (Fig. 9.7) is the WSDL document that xrpcc generates for the Vote service. VoteService_Config.properties (Fig. 9.8) is a server-configuration that xrpcc generates for the Vote service, which contains initialization parameters and their values for JAXRPCServlet. The next section discusses JAXRPCServlet in detail.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="VoteService"
targetNamespace="http://www.deitel.com/VoteService.wsdl"
xmlns:tns="http://www.deitel.com/VoteService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
<types/>
<message name="addVote">
  <part name="String_1" type="xsd:string"/>
</message>
<message name="addVoteResponse">
  <part name="result" type="xsd:string"/>
</message>
<portType name="VotePortType">
  <operation name="addVote">
    <input message="tns:addVote"/>
    <output message="tns:addVoteResponse"/>
  </operation>
</portType>

<binding name="VoteBinding" type="tns:VotePortType">
  <operation name="addVote">
    <input>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       use="encoded"
       namespace="http://www.deitel.com/VoteService.wsdl"/>
    </input>
    <output>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       use="encoded"
       namespace="http://www.deitel.com/VoteService.wsdl"/>
    </output>
  </operation>
</binding>
</definitions>
```

Fig. 9.7 VoteService.wsdl document generated by xrpcc. (Part 1 of 2.)
Deploying the Vote Service with Tomcat

To deploy a Web service to Tomcat (which is included with the JWSDP), we need to write a deployment descriptor. Web.xml (Fig. 9.9) is the deployment descriptor for the Vote service. Element servlet (lines 14–30) describes the JAXRPCServlet servlet that is distributed with the JWSDP EA2 package. Servlet JAXRPCServlet is a JAX-RPC implementation for dispatching the request to the Web service implementation. In our case, the JAXRPCServlet dispatches the client request to the VoteImpl class. When the JAXRPCServlet receives an HTTP request that contains a SOAP message, the servlet retrieves the data that the SOAP message contains, then dispatches the method call to the service implementation class via the tie. Element servlet-class (lines 20–22) specifies the compiled servlet’s fully qualified class name—com.sun.xml.rpc.server.JAXRPCServlet. The JAXRPCServlet obtains information about the server-configuration file, which is passed to the servlet as an initialization parameter. Element init-param (lines 23–28) specifies the name and value of the initialization parameter needed by the JAXRPCServlet. Element param-name (line 24) indicates the initialization parameter name, which is configuration.file. Element param-value (lines 25–27) specifies the initialization-parameter value, /WEB-INF/VoteService_Config.properties (generated by the xrpcc tool), which is the location of the server-configuration file. Element servlet-mapping (lines 33–36) specifies

Fig. 9.7 VoteService.wsdl document generated by xrpcc. (Part 2 of 2.)

Fig. 9.8 VoteService_Config.properties file generated by the xrpcc tool.

Deploying the Vote Service with Tomcat

To deploy a Web service to Tomcat (which is included with the JWSDP), we need to write a deployment descriptor. Web.xml (Fig. 9.9) is the deployment descriptor for the Vote service. Element servlet (lines 14–30) describes the JAXRPCServlet servlet that is distributed with the JWSDP EA2 package. Servlet JAXRPCServlet is a JAX-RPC implementation for dispatching the request to the Web service implementation. In our case, the JAXRPCServlet dispatches the client request to the VoteImpl class. When the JAXRPCServlet receives an HTTP request that contains a SOAP message, the servlet retrieves the data that the SOAP message contains, then dispatches the method call to the service implementation class via the tie. Element servlet-class (lines 20–22) specifies the compiled servlet’s fully qualified class name—com.sun.xml.rpc.server.JAXRPCServlet. The JAXRPCServlet obtains information about the server-configuration file, which is passed to the servlet as an initialization parameter. Element init-param (lines 23–28) specifies the name and value of the initialization parameter needed by the JAXRPCServlet. Element param-name (line 24) indicates the initialization parameter name, which is configuration.file. Element param-value (lines 25–27) specifies the initialization-parameter value, /WEB-INF/VoteService_Config.properties (generated by the xrpcc tool), which is the location of the server-configuration file. Element servlet-mapping (lines 33–36) specifies
The URL pattern enables the server to determine which requests should be sent to the servlet (JAXRPCServlet).

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc./DTD Web Application 2.3//EN" "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>
  <display-name>Java Web Service JAX-RPC VoteService Example</display-name>
  <description>Vote Service Application</description>
  <servlet>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <display-name>JAXRPCEndpoint</display-name>
    <description>Endpoint for Vote Service</description>
    <servlet-class>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-class>
    <init-param>
      <param-name>configuration.file</param-name>
      <param-value>/WEB-INF/VoteService_Config.properties</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <!-- Servlet mappings -->
  <servlet-mapping>
    <servlet-name>JAXRPCEndpoint</servlet-name>
    <url-pattern>/vote/endpoint/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>
</web-app>
```

Figure 9.9  Web.xml for deploying the Vote service.

Figure 9.10 shows the resulting jaxrpc-vote Web-application deployment directory structure. Since the Vote service implementation uses a Cloudscape database, we need to include both cloudclient.jar and RmiJdbc.jar in the lib directory. These two jar files are available from directory %J2EE_HOME%\lib\cloudscape, where J2EE_HOME is the J2EE installation directory. The classes directory contains all
classes in the `VoteOutput` directory, including `Vote.class`, `VoteImpl.class` and other classes generated by `xrpcc`.

Fig. 9.10  Webapp directory structure.

We may verify whether service `Vote` is deployed successfully. To verify the deployment, start Tomcat and point your browser to:

```
http://localhost:8080/jaxrpc-vote/vote/endpoint
```

Figure 9.11 shows the result.

Fig. 9.11  Service deployment result.