Chapter {10}

JAX-RPC

Java API for XML-based RPC (JAX-RPC) is intended to be a Java API to expose remote procedure calls that use XML to business applications that occur primarily, though not exclusively, on the periphery of organizations. The need for such synchronous API increases as corporations begin to communicate with other business partners using disparate hardware and software systems.

Remote procedure call (RPC) has been around for a while, with many implementations. It essentially enables clients to work with remote procedures, or *routines,* that reside on different machines just as if the procedures were executed locally. In its simplest form, a client calls a procedure, with the name of the procedure and the arguments; the server does something useful and sends the results back to the client.

Allowing different machines or processes in different address spaces to communicate with each other isn't really a new concept. The Java RMI and CORBA models are good examples of RPC that allows objects to marshal arguments, invoke a procedure or method on an object residing on a different machine, unmarshal the results, and use them.

An almost infinite number of data formats is possible for the arguments and results. As more and more Java applications expose themselves to interoperate and move toward a Web-service-based paradigm, XML is the new choice for this data format. JAX-RPC facilitates the invocation of remote procedures, using XML as the data format and SOAP as the data protocol.

SOAP defines the XML-based protocol for exchange of information in a distributed environment, specifying the envelope structure, encoding rules, and a convention for *representing* remote procedure calls and responses. JAX-RPC provides a Java API for developers to invoke remote procedure calls, by abstracting and hiding the low-level SOAP semantics associated with RPC from applications. ▷

▷ In this chapter, *runtime* refers to a JAX-RPC–compliant implementation, such as the reference implementation packed with Java WSDP.

JAX-RPC Service Model

As Figure 10.1 shows, the service model for JAX-RPC is similar to other RPC models, such as RMI-IIOP and CORBA. The model has several components.

The layers shown in Figure 10.1 correspond to the Open System Interconnection (OSI) networking model, which has these characteristics:

- The physical layer conveys the bitstream through the network.
- The data link layer encodes and decodes data packets into bits.
- The network layer provides switching, routing, packet sequencing, addressing, and forwarding between virtual circuits, to transmit data from node to node.
- The transport layer provides transparent transfer of data between hosts and is responsible for end-to-end error recovery and flow control. Clearly, the HTTP binding for SOAP lacks some of this, whereas other bindings, such as POP-SMTP, IMAP, and JMS do not.

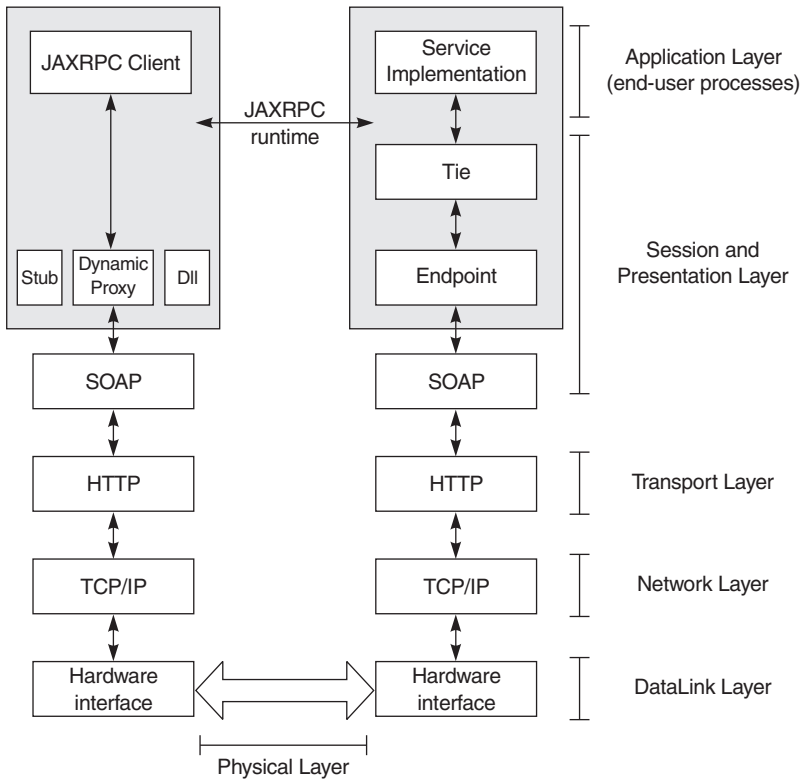

Figure 10.1 The JAX-RPC model

- The session layer establishes, coordinates, and terminates connections, exchanges, and dialogs between the applications.
- The presentation layer, also known as the syntax layer, provides independence from differences in data representation by translating from application to network format, and vice versa. The presentation layer works to transform data into the form the application layer can accept.
- The application layer is the actual application and end-user processes, where business functionality is addressed.

Although JAX-RPC relies on complex protocols, the API hides this complexity from the application developer. On the server side, the developer specifies the remotely accessible procedures by defining methods in a Java *service definition* interface and writing one or more Java classes that implement those methods. JAX-RPC exposes these objects as a *service endpoint* and generates the relevant ties. The client never directly communicates with the *service implementation*. The client uses a stub or other mechanisms to communicate with the endpoint (covered later in this chapter), and the endpoint uses the tie. The client then invokes the service, passing in relevant parameters, and the service returns the results to the client.

Before we dive into the internals of this model, we will take a look at the data types and see how the marshalling and unmarshalling occurs. We will then see how to use that in developing JAX-RPC services.

> Data Types and Serialization

Let us revisit some object-oriented concepts. An object at any time has state. This state, represented by its member variables at that time, is the object's snapshot. The definition of the object is the class file or compiled representation. An object with no member variables—that is, no state—is essentially just a utility that does something useful every time its methods are invoked. It may create other objects and change their states, but the *scope* of such secondary objects is limited to the method.

To do a remote procedure call, something representing *state* must be sent over the wire, and something representing *state* must be returned. Sending objects over the network is not trivial, since the network is not aware of objects; it supports only bit transmission.

The mechanism used to change the objects into a format that can be transmitted over the network is called *marshalling,* and reconstructing the objects from

this format is called *unmarshalling*. Marshalling over the wire requires object state to be extracted and sent in a well-defined format. Unmarshalling requires that the format be known, for reconstruction to take place. To marshal and unmarshal successfully, both sides in the exchange must use the same protocol to *encode* and *decode* object structure and data. For example, RMI Java uses Java serialization to marshal and unmarshal objects over Java Remote Method Protocol (JRMP). CORBA uses IIOP, DCOM uses ORPC, and Gemstone uses SRP.

In summary, four things are required between communication parties in different address spaces:

1. An agreement on the data format
2. An agreement on the mechanism for transforming and reconstructing object state into this format
3. An agreement on the protocol for communication between objects
4. An agreement on the transport protocol

XML helps in achieving item 1, XML schemas and SOAP with 2 and 3, and HTTP (and others in the IP family of protocols) with 4. ▷

So how is this relevant to JAX-RPC? JAX-RPC defines

- The data type mapping of Java-XML and XML-Java for making the remote service invocation possible
- Java-WSDL and WSDL-Java for making the service description possible

This is significant, because JAX-RPC provides a *standard* for vendors to implement and makes developer code vendor-neutral, much the way any of the other Java specifications do. Just as developers write a J2EE application and expect it to behave the same across J2EE-compliant application servers from multiple vendors, JAX-RPC applications will behave the same across JAX-RPC runtimes.

This does not mean that a JAX-RPC client can call only a JAX-RPC service and a JAX-RPC service can be used only by a JAX-RPC client. An application

▷ Java (platform-independent language) + XML (platform-independent data format) + SOAP (platform-independent object communication protocol) + IP family of protocols (platform/network-independent transport) = core of what is driving Web services and their adoption today

could still use a JAX-RPC client to invoke a .NET service and a .NET client to invoke a JAX-RPC service, as we will demonstrate later. As Figure 10.2 shows, because the data format, object communication protocol, and transport protocol are platform- and vendor-implementation independent, the application can be accessed by any client on any platform, as long as it uses these standards. The data type mapping and serialization rules defined by JAX-RPC are useful when the JAX-RPC runtime is being used on the Java platform at the client or server end. ▷

From an RPC perspective, if the client and service are written in Java, the runtime needs to know the following information:

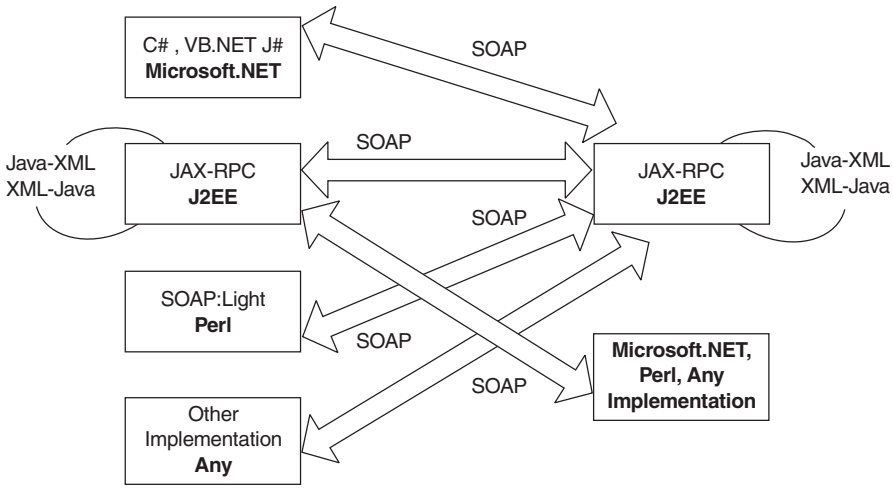

Figure 10.2 JAX-RPC client-server interaction

▷ RPC implementations using SOAP from different vendors:

- Apache SOAP 2.2
- Apache Axis (Alpha-1)
- HP Web Services Platform
- IBM Web Services Toolkit, WSIF
- IONA XML Bus 1.2
- Microsoft SOAP Toolkit 2.0
- Microsoft .NET
- Others: PocketSOAP, SOAP::Lite, Systinet WASP, SOAP-RMI, GLUE, Cape Clear

1. The endpoint of the service–that is, where to invoke the service
2. The method name and signature–that is, what to invoke
3. How to marshal Java data types and objects into an XML format that can be transmitted over the wire to invoke the method
4. How to unmarshal the incoming XML into Java data types and objects to process the results of that operation, if any

Java-to-XML Marshalling

While JAX-RPC does not define the actual marshalling mechanism, it does define the input and output types that result from that marshalling. Vendors write the marshalling code as part of their implementations.

In JAX-RPC, marshalling is different from the standard Java serialization mechanism, where all nontransient fields in the class are automatically serialized. JAX-RPC defines a standard set of Java types as method arguments and return types, meaning that a JAX-RPC–compliant system will provide the ready-to-use serializers and deserializers for these types:

1. All Java primitives, with the exception of a `char` (`int`, `float`, `long`, `short`, `double`, `byte`, `Boolean`). A `char` is treated as a `String`, since XML schemas have no `char` type primitive (see Appendix A).
2. An object that is an instance of
   - `java.lang.String`
   - `java.util.Date`
   - `java.util.Calendar`
   - `java.math.BigInteger`
   - `java.math.BigDecimal`
3. An object that is an instance of a class that conforms to the following restrictions:
   - It should conform to the JavaBean specification, so that its variables can be easily accessed.
   - It should not be a `Remote` object (i.e., should not implement `java.rmi.Remote`).
   - It should have a default no arguments constructor.

   These are important, because any other class is passed between the client and the server. For example, if a `java.util.Map` of `com.flutebank.Account` objects

must be passed from the client to the server, a pluggable serializer and deserializer pair must be written. This is explored later in this chapter.

4. An array (with the caveat that it must contain `bytes` or a supported type)
5. A `java.lang.Exception` class ▷

The above rules differ from the standard Java serialization requirements. Table 10.1 lists the details for the same class to be used across an RMI/RMI-IIOP application as well as a JAX-RPC application. This is relevant where an existing EJB, RMI, or RMI-IIOP object must expose itself directly as a JAX-RPC service and the code must be reused across these interfaces.

Once the parameter types have been defined, rules and a standard mechanism to map these data types from Java to XML must also be defined. JAX-RPC does this, as Table 10.2 shows. ▷▷

XML-to-Java Unmarshalling

To invoke a procedure on an object with incoming XML data, an implementation must map XML data types into Java data types. This is identical to Table 10.2, with some differences explained in Tables 10.3a and 3b. XML type is declared as nillable in the schema, then it maps to its corresponding Java wrapper (primitives in Java cannot be null).

SOAP Bindings and Encoding

In Chapter 4, we looked at SOAP encoding in detail. Encoding refers to how data is serialized and sent over the wire. The parties exchanging messages have to agree on one rule to ensure that both correctly interpret the message sent from the

---

▷ JAX-RPC does not support a pass by reference mode for parameters and does not support passing of remote objects, because both the SOAP 1.1 and SOAP 1.2 treat objects by reference as out of scope. JAX-RPC mandates support for pass by copy semantics for parameters and return values, similar to the way nonremote objects are passed in RMI.

▷▷ JSR–31 defines the XML Data Binding Specification with JAXB, for converting an XML schema into Java classes. In the future, JAX-RPC will include the data type mapping defined by JAXB.

Table 10.1 Portability across JAX-RPC and RMI

JAX-RPC	Java serialization	A portable value object
Should not extend `Remote`.	Can extend `Remote`. It is treated as an remote object and passed by reference.	Should not implement `Remote`.
`Serializable` not required.	`Serializable` required.	Should implement `Serializable`.
`transient` fields are not serialized.	`transient` fields are not serialized.	`transient` fields are not serialized.
`static` fields are serialized.	`static` fields are not serialized.	Should not contain `static` fields.
All `public` variables are serialized.	`public` transient variables are not serialized.	Should not contain any `public transient` variables.
Only private, protected, package-level fields that have get/set methods are serialized.	Get/set methods are not required. Private, protected, package-level fields are still serialized.	Should have get/set methods for all private, protected, package-level fields.
Bean properties are serialized.	Bean properties are serialized.	Can have bean properties with get/set methods.

other side. They can either agree beforehand, using a predefined *encoding scheme*, or use an XML schema directly in the data to define the data types. The message with the former notation is said to be an *encoded* message; the latter is said to be a *literal* message.

SOAP encoding refers to the rules defined by the SOAP specification that the parties can follow to interpret the contents of the `Body` element. SOAP defines an encoding scheme, also referred to as Section 5 encoding (since it is specified in section 5 of the SOAP specifications). It outlines a schema *(http://schemas.xmlsoap.org/soap/encoding/)* containing certain basic data types that participants in the conversation can use to describe the elements in the body of the SOAP message. This encoding is not mandatory, and there is no default.

In Chapter 4, we also looked at the use of the `encodingStyle` attribute and the use of simple and compound types. To recall, the `encodingStyle` attribute in the SOAP message can be used to indicate the encoding in use. For example, the message below indicates that SOAP encodings are in use:

Table 10.2 Java-to-XML Data Type Mapping

Java type	XML type
`Boolean`	`xsd:boolean`
`Byte`	`xsd:byte`
`Short`	`xsd:short`
`Int`	`xsd:int`
`Long`	`xsd:long`
`Float`	`xsd:float`
`Double`	`xsd:double`
`byte[]`	`xsd:base64Binary`
`Byte[]`	`xsd:base64Binary`
`java.lang.String`	`xsd:string`
`java.math.BigInteger`	`xsd:integer`
`java.math.BigDecimal`	`xsd:decimal`
`java.util.Calendar`	`xsd:dateTime`
`java.util.Date`	`xsd:dateTime`
`javax.xml.namespace.Qname`	`xsd:QName`
JavaBean class whose properties are any supported Java data type or another valid JavaBean	XML schema `sequence` of elements
`Array` of any of above	`SOAP array`

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:enc="http://
schemas.xmlsoap.org/soap/encoding/" xmlns:ns0="http://www.flutebank.com/xml"
env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<env:Body>
<ns0:getLastPayment>
    <vendor xsi:type="enc:string">my cable tv provider</vendor>
</ns0:getLastPayment>
</env:Body>
</env:Envelope>
```

Table 10.3a XML-to-Java Data Type Mapping for Basic Types

XML type	Java type	When declared as nillable
xsd:boolean	Boolean	java.lang.Boolean
xsd:byte	Byte	java.lang.Byte
xsd:short	Short	java.lang.Short
xsd:int	Int	java.lang.Integer
xsd:long	Long	java.lang.Long
xsd:float	Float	java.lang.Float
xsd:double	double	java.lang.Double
xsd:base64Binary	byte[]	
xsd:hexBinary	byte[]	
xsd:string	java.lang.String	
xsd:integer	java.math.BigInteger	
xsd:decimal	java.math.BigDecimal	
xsd:dateTime	java.util.Calendar	
xsd:Qname	javax.xml.namespace.Qname	

The message shows that the `vendor` element is of type `enc:string` defined in the encoding schema, represented by the `encodingStyle` attribute. The receiver of this message, processing the `vendor` element, knows it is a SOAP-encoded string, which it can then translate into the language in which the service is implemented.

The SOAP specification does not define any language bindings for the data types described by its encoding schema. Instead, the types are generic enough to model some of the typical data types found in Java and most other programming languages. JAX-RPC defines how the simple types in the SOAP encoding are mapped to Java, as per Table 10.4.

Note that the data types in Table 10.4 are the same as the nillable types in Table 10.3a, because they map to the same underlying basic XML schema types. Of particular interest is the `Array` type, which maps to Java arrays. In the previous

Table 10.3b XML-to-Java Data Type Mapping

XML construct	Java construct
`ComplexType`	JavaBeans class with the same name. Its properties are mapped from the element's `name` and `type`. Complex types derived by extension are mapped into classes with similar hierarchies.

Example

```
<complexType name="PaymentConfirmation">
    <sequence>
          <element name="confirmationNum" type="int"/>
          <element name="payee" type="string"/>
          <element name="amt" type="double"/>
    </sequence>
</complexType>

public class PaymentConfirmation {
    private int confirmationNum;
    private String payee;
    private double amt;
    public PaymentConfirmation() {
    }

    public PaymentConfirmation(int confirmationNum,
                               java.lang.String payee, double amt) {
        this.confirmationNum = confirmationNum;
        this.payee = payee;
        this.amt = amt;
    }

// getXXX/setXXX methods for each member property
}
```

(continued)

Table 10.3b XML-to-Java Data Type Mapping (Cont'd)

XML construct	Java construct
`Enumerations`	A Java class with the same name as the enumeration. The class must contain 1. The enumerated values as members of the enumeration type 2. A `getValue` method that returns the current value 3. Two static methods for each label

Example

```
<simpleType name="PaymentDetail" >
    <restriction base="xsd:string" >
        <enumeration value="checking" />
        <enumeration value="saving" />
        <enumeration value="brokerage" />
    </restriction>
</simpleType>

public class PaymentDetail implements Serializable {
    private String value;
    public static final String _checkingString = "checking";
    public static final String _savingString = "saving";
    public static final String _brokerageString = "brokerage";

    public static final String _checking = new String
                                    (_checkingString);
    public static final String _saving = new String(_savingString);
    public static final String _brokerage = new String
                                    (_brokerageString);

    public static final PaymentDetail checking =
                                    new PaymentDetail(_checking);
    public static final PaymentDetail saving = new PaymentDetail
                                    (_saving);
    public static final PaymentDetail brokerage =
                                    new PaymentDetail(_brokerage);
```

Table 10.3b XML-to-Java Data Type Mapping (Cont'd)

```
        protected PaymentDetail(String value) {
            this.value = value;
        }

        public String getValue() {
            return value;
        }

        public static PaymentDetail fromValue(String value)
            throws IllegalStateException {
            if (checking.value.equals(value)) {
                return checking;
            } else if (saving.value.equals(value)) {
                return saving;
            } else if (brokerage.value.equals(value)) {
                return brokerage;
            }
            throw new IllegalArgumentException();
        }

        public static PaymentDetail fromString(String value)
            throws IllegalStateException {
            if (value.equals(_checkingString)) {
                return checking;
            } else if (value.equals(_savingString)) {
                return saving;
            } else if (value.equals(_brokerageString)) {
                return brokerage;
            }
            throw new IllegalArgumentException();
            }
        }
    // other methods not shown
    }
```

Table 10.4 Mapping of SOAP Simple Types to Java

SOAP-encoded simple type	Java type
`String`	`java.lang.String`
`Boolean`	`java.lang.Boolean`
`Float`	`java.lang.Float`
`Double`	`java.lang.Double`
`Decimal`	`java.math.BigDecimal`
`Int`	`java.lang.Integer`
`Short`	`java.lang.Short`
`Byte`	`java.lang.Byte`
`Base64`	`byte[]`
`Array`	`Java array`

extract, if the SOAP body contained something like the following, it would be mapped to an array of `String[]` objects in the Java side by JAX-RPC, and vice versa:

```
<ns0:getLastPayment>
<vendor enc:arrayType="xsd:string[4]" xsi:type="enc:Array">
   <item xsi:type="xsd:string">AT&T</item>
   <item xsi:type="xsd:string">Sprint PCS</item>
   <item xsi:type="xsd:string">Flute Electric Co</item>
</vendor>
</ns0:getLastPayment>
```

While mapping arrays, the type of the array is determined from the schema type. The size is determined at runtime rather than at declaration time. JAX-RPC also supports multidimensional arrays, where the types are supported JAX-RPC types. Listing 10.1 shows how a multidimensional array of application-defined `PaymentDetail` objects may be mapped on the wire using SOAP encoding:

Listing 10.1 A multidimensional array of application-defined `PaymentDetail` objects mapped on the wire using SOAP encoding

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
```

```
xmlns:ns0="http://www.flutebank.com/xml"
// other namespaces >
<env:Body>
// other XML
<ns0:ArrayOfArrayOfPaymentDetail id="ID1" xsi:type="enc:Array"
enc:arrayType="ns0:PaymentDetail[2,2]">
           <item href="#ID2"/>
           <item href="#ID3"/>
           <item href="#ID4"/>
           <item href="#ID5"/>
</ns0:ArrayOfArrayOfPaymentDetail>
     <ns0:PaymentDetail id="ID2" xsi:type="ns0:PaymentDetail">
           <date xsi:type="xsd:dateTime">2002-08-26T21:17:37.678Z</date>
           <account xsi:type="xsd:string">Credit</account>
           <payeeName xsi:type="xsd:string">Digital Credit Union</payeeName>
           <amt xsi:type="xsd:double">2000.0</amt>
     </ns0:PaymentDetail>

     <ns0:PaymentDetail id="ID3" xsi:type="ns0:PaymentDetail">
           <date xsi:type="xsd:dateTime">2002-08-26T21:17:37.678Z</date>
           <account xsi:type="xsd:string">Credit</account>
           <payeeName xsi:type="xsd:string">Auto Loan Company</payeeName>
           <amt xsi:type="xsd:double">299.0</amt>
     </ns0:PaymentDetail>

     <ns0:PaymentDetail id="ID4" xsi:type="ns0:PaymentDetail">
           <date xsi:type="xsd:dateTime">2002-08-26T21:17:37.678Z</date>
           <account xsi:type="xsd:string">Credit</account>
           <payeeName xsi:type="xsd:string">AT&amp;amp;T Wireless</payeeName>
           <amt xsi:type="xsd:double">20.0</amt>
     </ns0:PaymentDetail>

     <ns0:PaymentDetail id="ID5" xsi:type="ns0:PaymentDetail">
           <date xsi:type="xsd:dateTime">2002-08-26T21:17:37.678Z</date>
           <account xsi:type="xsd:string">Credit</account>
           <payeeName xsi:type="xsd:string">AT&amp;amp;T Long distance</payeeName>
           <amt xsi:type="xsd:double">12.0</amt>
     </ns0:PaymentDetail>
</env:Body>
</env:Envelope>
```

In Chapter 4, we introduced the concept of RPC/document *style*. Recall that a SOAP message on the wire can be represented in either in *RPC* style or *document* style, which affects the body of the message. In RPC style, the client invokes a method on the server, by sending in the body of the SOAP message all information necessary for that method's execution. It receives a response in the same fashion. All SOAP messages shown till now in this chapter have been in RPC style.

In document style, the client and server communicate using XML documents. The client sends an XML document such as a purchase order; the server does something with it and returns an XML document such as an invoice as a result (Figure 10.3).

Thus, based on the style (RPC or document) and use (encoded or literal), four combinations result:

- RPC/encoded
- RPC/literal
- Document/encoded
- Document/literal

In Chapter 5, we looked at how WSDL could represent these combinations using the `style` and `use` attributes of the `binding` element. Let us elaborate on that with a detailed example. The WSDL extract below shows a sample schema and message extracts. The schema has two elements, `Amount` and `details`, and a custom type, `PaymentDetail`. The `message` element shown has four parts: `part1`, which is of type `PaymentDetail`; `part2`, which is of type `int`; `part3`, which is a sim-

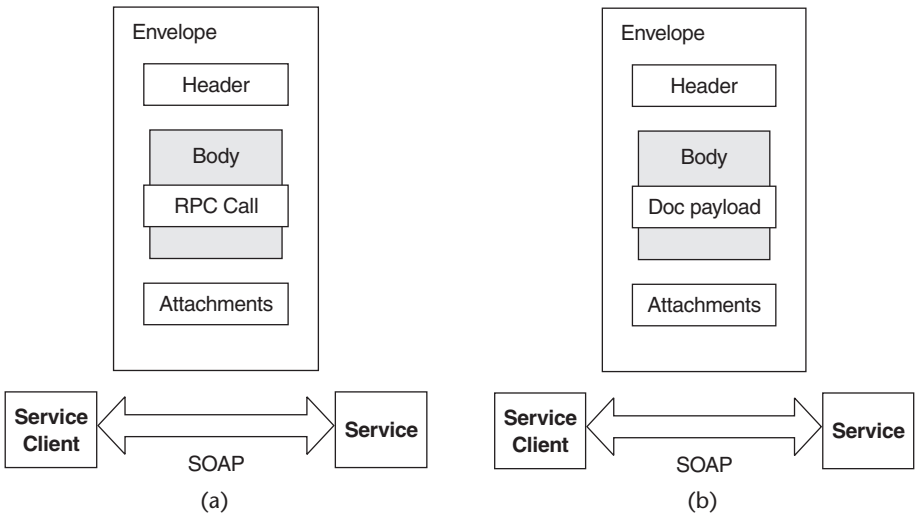

Figure 10.3 (a) RPC style. (b) Document style.

ple element Amount; and part4, which is an element of complex type PaymentDetail. Table 10.5 compares the combinations. ▷

```
<definitions
  targetNamespace="(namespace for service WSDL)"
  xml:typens="(namespace for service schema)">

<types>
<schema targetNamespace="(namespace for the schema)">
  <element name="Amount" type="xsd:int"/>
  <element name="details" type="typens:PaymentDetail"/>
<!--user defined data type-->
  <complexType name="PaymentDetail">
     <sequence>
         <element name="balance" type="xsd:int"/>
         <element name="payeeName" type="xsd:string"/>
     </sequence>
  </complexType>
</schema>
</types>

<message...>
  <part name='part1' type="typens: PaymentDetail"/>
  <part name='part2' type="xsd:int"/>
  <part name='part3' element="typens:Amount"/>
  <part name='part4' element="typens:details"/>
</message>
...
</definitions>
```

▷ In document/literal style, the contents between <SOAP-ENC:Body> and </SOAP-ENC:Body> are sent as an XML string to the application, which is responsible for parsing the XML.

JAX-RPC requires RPC/encoded and document/literal support. These are optional for the other two combinations.

For more about the style use combinations of RPC/literal, document/literal, PRC/encoded, and document/encoded, see Chapter 5.

Table 10.5 Examples of RPC/Encoded, RPC/Literal, Document/Literal, and Document/Encoded Combinations

Style

RPC/encoded

WSDL

```
<operation name="schedulePayment" style="rpc" ... >
<input>
<soap:body parts="part1 part2"
          use="encoded"
          encoding= "http://schemas.xmlsoap.org/soap/encoding/"
 namespace="(namespace for message)"/>
</input>
</operation>
```

The WSDL extract shows an RPC operation named `schedulePayment` being invoked in RPC/encoded format, with SOAP encodings and two input parameters.

SOAP message

```
<soapenv:body xmlns:mns="(namespace for message)">
<mns:schedulePayment>
     <part1 HREF="#1" TARGET="_self"/>
     <part2>5688</part2>
</mns:schedulePayment>

<mns:PaymentDetails id="#1">
     <balance>8933</balance>
     <payeeName>johnmalkovich</payeeName>
</mns:PaymentDetails>
</soapenv:body>
```

Style

RPC/literal

```
<operation name="schedulePayment"
           style="rpc" ... >
  <input>
       <soap:body use="literal"
             parts="part1 part2 part3 part4"
```

Table 10.5 Examples of RPC/Encoded, RPC/Literal, Document/Literal, and Document/Encoded Combinations (Cont'd)

```
 namespace="(namespace for message)"/>
  </input>
// ...
</operation>
```

The WSDL extract shows an RPC operation named `schedulePayment` being invoked in RPC/literal format, with four input parameters. It shows how data types can be passed and how a schema element is sent across the wire directly, without any encoding.

SOAP message

```
<soapenv:body xmlns:mns="(namespace for message)"
        xmlns:typens="(namespace for service schema)".. >
<mns:schedulePayment>

<mns:part1>

<typens:balance>8933</typens:balance>
    <typens:payeeName>johnmalkovich
                </typens:payeeName>
</mns:part1>

<mns:part2>5688</mns:part2>

<mns:part3>

<typens:Amount>5688</typens:Amount>
</mns:part3>

<mns:part4>
      <typens:PaymentDetail>
      <typens:balance>8933</typens:balance>
      <typens:payeeName>johnmalkovich
                    </typens:payeeName>
</typens:PaymentDetail>
</mns:part4>

</mns:schedulePayment>
</soapenv:body>
```

(Continued)

Table 10.5 Examples of RPC/Encoded, RPC/Literal, Document/Literal, and Document/Encoded Combinations (Cont'd)

Style

Document/literal
`<operation name="schedulePayment"`
`  **style="document"** ... >`
`<input>`
`<soap:body parts="part1 part3 part4"`
`  **use="literal"**>`
`</input>`
`</operation>`
The WSDL extract shows a document-style message being sent in literal format with four input parameters. It shows how data types can be passed and how a schema element is sent across the wire directly, without any encoding.
SOAP message
`<soapenv:body`
`  xmlns:typens="(namespace for service schema)" ... >`
 	
`<typens:balance>8933</typens:balance>`
`<typens:payeeName>johnmalkovich`
`  </typens:payeeName>`
 	
`<typens:Amount>5688</typens:Amount>`
 	
`<typens:details>`
`  <typens:balance>5688</typens:balance>`
`  <typens:payeeName>johnmalkovich`
 	
`</typens:payeeName>`
`</typens:details>`
`</soapenv:body>`

Table 10.5 Examples of RPC/Encoded, RPC/Literal, Document/Literal, and Document/Encoded Combinations (Cont'd)

Style

Document/encoded
`<operation name="schedulePayment"`
`**style="document"** ... >`
`<input>`
`<soap:body parts="part1 part2"`
`**use="encoded"**`
`encoding=`
`"http://schemas.xmlsoap.org/soap/encoding/"`
`namespace="(namespace for message)"/>`
`</input>`
`</operation>`
The WSDL extract shows a document style message being sent with SOAP encoding and four input parameters. It shows how data types can be passed and how a schema element is sent across the wire directly.

SOAP message
`<soapenv:body ...`
`xmlns:mns="(namespace for message)">`
`<mns:PaymentDetails>`
`<balance>8933</balance>`
`<payeeName>johnmalkovich</payeeName>`
`</mns:PaymentDetails>`
 	
`<soapenc:int>5688</soapenc:int>`
`</soapenv:body>`

When to Use RPC/Encoded and Document/Literal

To best choose a particular style, an architect would need to understand the implementation or desired use. In general, however, the effort and complexity involved in document style service is greater than for RPC style–for example, in negotiating the schema design with the business partners and validating the

document against the schema. When architecting applications, these are some of the decision points for deciding between RPC or document style:

1. **State maintenance.** If multiple service invocations are involved in a single business transaction, with state maintained between the service invocations, the service must maintain state. Maintaining state is nontrivial and is usually not as simple as exposing a stateful EJB as a Web service over multiple RPC invocations. One alternative is to use document style, pass the contents of an entire transaction in the document, and allow the service implementation to ensure the sequence and state maintenance in the transaction. Information about that state can be returned in a token or in the resulting document to the client if needed.

   If the service consumer is only requesting information or persisting information in a specific format (e.g., industry-standard XML schema), a document style message makes more sense, because it is not constrained by the RPC-oriented encoding.

2. **Integration with external parties and decoupled interfaces.** Service consumers outside the enterprise typically have little control over the use and consequences of changes to the service interface. RPC interfaces are expected not to change, because any change would break the contract between the service and its consumers. In scenarios where a large number of applications have produced stub code from the service's WSDL document (we will see how to do this later in this chapter), changing the WSDL would cause all the applications that rely on a specific method signature to break. If you anticipate frequent changes, you can use a document/literal style, because the impact on the WSDL can be minimized. This is useful for the *late binding pattern* discussed in Chapter 5.

3. **Validate business documents.** A Web service can use the capabilities of a validating parser and schemas to describe and validate high-level business documents. This is opposed to RPC, where the XML describes the method and parameters encoded for that method call, which cannot be used to enforce high-level business rules.

   To enforce these rules for the document with RPC, a message must include an XML document as a string parameter or attachment and must hide the validation in the implementation of the method being called. If an attachment is not used, the service will have to deal with custom marshalling and unmarshalling code for a possibly complex XML structure. This often leads to valid calls with invalid parameters that are not detected till the entire struc-

ture has been processed. In short, if the service is accepting or returning a complex XML structure, a document style is better suited. The XML can be validated against the schema prior to calling the service, and no custom marshalling code is required.

4. **Performance and memory limitations.** Marshalling and unmarsalling parameters to XML in memory can be an intensive process. The SOAP model inherently requires DOM-based processing of the envelope, which can lead to large DOM trees in memory if the XML representation is complex. However, document style services can choose SAX handling of the including XML document, to perform quicker and less memory intensive parsing. This is critical for services that handle many simultaneous requests.
5. **For fine-grained communication.** With RPC calls, only a limited amount of data can be passed around in a single invocation, and it is not possible to include multiple RPC calls in a single SOAP envelope. If the application requires a significant amount of data to be passed around, RPC style with an attachment (e.g.. an XML document) is better suited.
6. **Request-response processing.** SOAP messages are, by nature, one-way transmissions from a sender to a receiver, but they are usually combined to implement a request/response model. SOAP piggybacking on top of a request-response–oriented transport, such as HTTP and JAX-RPC, is well suited to applications that require synchronous request-response processing. Such applications typically involve retrieval of results based on some remote procedure execution, for which RPC/encoded messages are well suited.
7. **Encoding scheme.** The default encoding scheme specified by SOAP is usually sufficient. If you determine a need to use custom encoding (e.g., the SOAP encoding doesn't meet your needs because you have complex types not addressed by JAX-RPC and SOAP), we recommend that you investigate the document style route instead, since schemas are by far a richer metalanguage than SOAP encoding. This allows for a more complex arrangement of information.

Document style combined with literal encoding allows validation. Changing that to *RPC/literal* takes that benefit away, because the surrounding RPC element does not appear in the schemas. A possible example where *RPC/literal* can be used instead of *document/literal* is when multiple RPC operations return XML documents using the same schema. *Document/encoded* takes away the benefits of *RPC/encoded* but does not add anything in return.

> JAX-RPC Development

We have just covered how data can be transferred over the wire, along with the rules and associated mechanics governing that. In this section, we will look at how services can be developed and realized using JAX-RPC and the steps involved in doing so.

Developing and consuming a JAX-RPC service can be categorized into five steps:

1. Service definition
2. Service implementation
3. Service deployment
4. Service description
5. Service consumption

In walking through these steps we will develop the service example introduced in Chapter 5. The example illustrates a bill payment service developed by Flute Bank as part of its online operations.

1. Service Definition

The term *service definition* is used to refer to the abstraction that defines the publicly surfaced view of the service. The service definition is represented as a Java interface that exposes the service's operations. The service definition is also called a remote interface, because it must extend the `java.rmi.Remote` interface, and because all methods in it must throw a `java.rmi.RemoteException`. The code below shows the BillPay Web service:

```
package com.flutebank.billpayservice;

import java.util.Date;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BillPay extends Remote {
       public PaymentConfirmation schedulePayment(Date date, String nickName, double
                              amount) throws ScheduleFailedException, RemoteException;
```

```
        public PaymentDetail[] listScheduledPayments() throws RemoteException;
        public double getLastPayment(String nickname) throws RemoteException;
}
```

The methods in the interface must have valid JAX-RPC data types (disussed earlier) as arguments and return types. If they are not a supported data type (e.g. `java.util.Map`), then appropriate *serializers* and *deserializers* must be available, so that these types can be marshaled and unmarshalled to and from their corresponding XML representations. The data type can also be a *holder* class. Holders and pluggable serializers are covered later in this chapter.

An implemenatation will usually verify this type information at compile time and warn the developer if it is not correct. A request sent with incorrect type information at runtime will generate a SOAP fault, because it will not be able to unmarshall the XML.

2. Service Implementation

The *service implementation,* also known as a *servant,* is the concrete representation of the abstract service definition; it is a class that provides the implementation or the service definition. The Java class must have a default constructor and must implement the remote interface that defines the service. Listing 10.2 shows the implementation for the BillPay service.

Listing 10.2 Implementation for Flute Bank's BillPay service

```
package com.flutebank.billpayservice;

import java.util.Date;

public class BillPayImpl implements BillPay {

public BillPayImpl(){}

public PaymentConfirmation schedulePayment(Date date, String nickName, double
                                           amount) throws ScheduleFailedException {
        // invoke business logic like EJBs here
               return new PaymentConfirmation(81263767,"Sprint PCS", amount);
           }
```

```
public PaymentDetail[] listScheduledPayments() {
       // lookup the detail objects and other business logic from EJBs here
       PaymentDetail details[]=new PaymentDetail[1];
       PaymentDetail dummy= new PaymentDetail("Digital Credit
                                                    Union","Credit",2000, new Date());
       details[0]=dummy;
       return details;
    }

public double getLastPayment(String nickname) {
   // lookup the detail objects and other business logic from EJBs for this
   // nickname based on the callers user id
            if(nickname.equalsIgnoreCase("my cable tv provider"))
                  return 829;
            else
                  return 272;
    }
}
```

Services are deployed in a JAX-RPC *runtime,* which is a container that implements the JAX-RPC specifications. By default, the runtime will just invoke the methods corresponding to the RPC request in the Java implemenatation. The service implementation can choose to provide hooks to allow the runtime to manage the service's lifecyle and allow the container to invoke callbacks on the service when major lifecycle events occur. The "hook" is defined as a `javax.xml.rpc.server.ServiceLifeCycle` interface that the service can implement. The container will then invoke methods on this service appropriately, via this interface. The interface defines an `init(Object context)` and a `destroy()` method:

```
public interface ServiceLifecycle{
       public void init(Object obj) throws ServiceException;
       public void destroy();
}
```

The behavior of these methods is similar to the `init()` and `destroy()` methods in a servlet. When the implementation is first instantiated, the `init()` method is invoked, and a context object passed to it, the `destroy()` method is called before the implementation needs to be removed (e.g., at shutdown or during a resource crunch). These methods are good places to initialize and release expensive resources, such as database connections and remote references. The con-

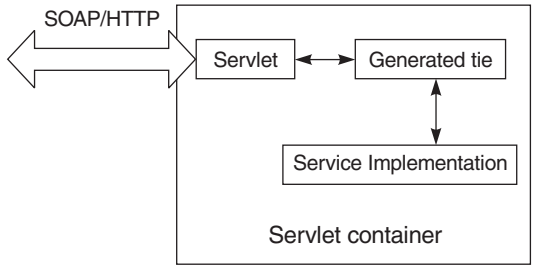

Figure 10.4 Service deployment

text is defined as an `Object`, to allow for different endpoint types to be used, as we will see later (e.g., the context will be different for an HTTP endpoint and a JMS endpoint).

As with a servlet, an implementation should not hold a client-specific state in instance variables, because the runtime can invoke methods from multiple threads. Architects should also avoid synchronizing the methods themselves. There are other ways to maintain client state, as discussed in the next section.

3. Service Deployment

We mentioned earlier that a service is deployed in a JAX-RPC runtime. A *service endpoint* is the perimeter where the SOAP message is received and the response dispatched. It is the physical entity exposed to service consumers that essentially services client requests. An endpoint is provided by the runtime and is not written by developers. An endpoint is bound to the transport protocol. Because a runtime is required to support an HTTP transport, JAX-RPC also defines the behavior of an endpoint for this protocol as a Java servlet, as Figure 10.4 shows. ▷

The servlet receives the SOAP message as the HTTP request, determines the servant to use for servicing that request, and delegates to it or its proxy representation (the tie). Once the service has done its work, the servlet is responsible for packaging the SOAP message and sending it back over HTTP.

The exact implementation of the servlet endpoint is left up to the runtime. The reference implementation contains a single servlet (`com.sun.xml.rpc.server.http.JAXRPCServlet`) that delegates to a tie, based on the `xrpcc`-generated prop-

▷ Even though a JAX-RPC runtime must support HTTP, it can use other transports as well. The JAX-RPC architecture is designed to be transport–independent, even though it describes the way HTTP is used if it is chosen as transport.

erties file (we will see this later in the chapter). Because the endpoint is a servlet, it requires a Servlet 2.2–compliant container. Also, the packaging and deployment to the endpoint of the service has to be the standard J2EE `WAR` file, with its defined structure (WEB-INF/classes and the web.xml file, etc.)

If a service implementation implements the `ServiceLifeCycle` interface, the context object passed in the `init()` is of type `javax.xml.rpc.server.ServletEndpointContext`:

```
public interface ServletEndpointContext{
       public MessageContext getMessageContext();
       public Principal getUserPrincipal();
       public HttpSession getHttpSession();
       public ServletContext getServletContext();
}
```

This context provides methods to access the `MessageContext`, `Principal`, `HttpSession` and `ServletContext` objects associated with the user. The listing below shows an example of how this can be used. These objects are good places for maintaining different kinds of state information:

- The `HttpSession` is a good place to maintain *client*-specific state, using the `getAttribute()` and `setAttribute()` methods.
- The `ServletContext` is a good place to access *application*-specific state, such as configuration parameters, Java Naming and Directory Interface (JNDI) names, and JNDI contexts, using the `getAttribute()` and `setAttribute()` methods.
- The `MessageContext` is a good place to obtain state set by message handlers during preprocessing of the message. Handlers are covered in detail later in the chapter.

```
public class BillPayImpl implements BillPay, ServiceLifecycle {
     private ServletEndpointContext ctx;
 public void init(java.lang.Object context){
        ctx=(ServletEndpointContext)context;
    }
public PaymentDetail[] listScheduledPayments() {
       SOAPMessageContext msgctx= (SOAPMessageContext) (ctx.getMessageContext());
       HttpSession session = ctx.getHttpSession();
          ServletContext servletctx= ctx.getServletContext()
```

```
        // other code
        }
}
```

The usage of the `ServletEndpointContext` is analogous to the `SessionContext` and `EntityContext` in EJBs.

4. Service Description

Once the service is defined, implemented, and ready for deployment as an endpoint, it also must be *described* clearly for service consumers. This is where WDSL comes in. Based on the service definition, the WSDL document describes the service, its operations, arguments, return types, and the schema for the data types used in them.

`xrpcc` *Internals*

The JAX-RPC reference implementation comes with the `xrpcc` (XML-based RPC Compiler) tool, which reads a tool-specific XML configuration file and generates the client- or server-side bindings shown in Figure 10.5. A developer can start with

- A remote interface and use `xrpcc` to generate the stubs, ties, and WSDL
- A WSDL document and generate the stubs to consume the service
- A WSDL document and generate the stubs, ties, and remote interface and implement the service

Listing 10.3 shows the format for the XML configuration file `xrpcc` reads.

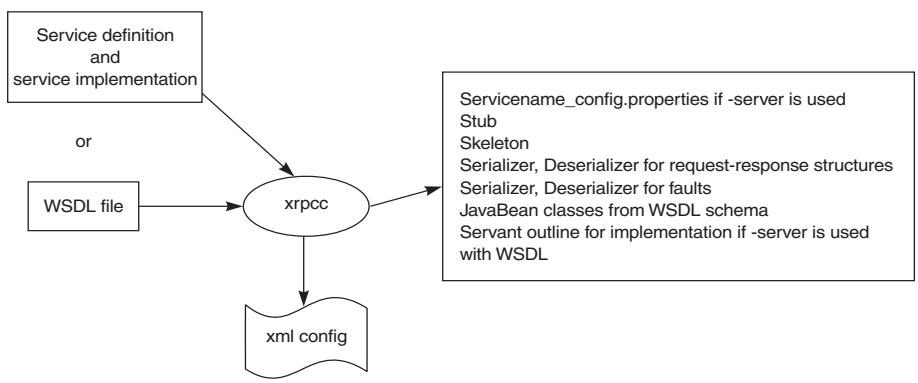

Figure 10.5 `xrpcc` artifacts

Listing 10.3 `xrpcc` configuration in the reference implementation

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
<service name="" packageName="" targetNamespace="" typeNamespace="">
  <interface name="" servantName="" soapAction="" soapActionBase="">
         <handlerChains>
           <chain runAt=" " roles="">
                         <handler className="" headers="">
                          <property name="" value=""/>
                         </handler>
                      </chain>
                    </handlerChains>
</interface>
       <typeMappingRegistry>
                   <import>
                       <schema namespace="" location=""/>
                   </import>
                   <typeMapping encodingStyle="">
                        <entry schemaType=""
                               javaType=""
                               serializerFactory=""
                               deserializerFactory=""/>
                   </typeMapping>
                   <additionalTypes>
                        <class name=""/>
                   </additionalTypes>
                </typeMappingRegistry>
                   <handlerChains>
                        <chain runAt="client" roles="">
                            <handler className="" headers="">
                                 <property name="" value=""/>
                            </handler>
                        </chain>
                        </handlerChains>
                        <namespaceMappingRegistry>
                             <namespaceMapping namespace=""
                                               packageName=""/>
                        </namespaceMappingRegistry>
              </service>
</configuration>
```

Key XML elements of the configuration are discussed below. Some refer to concepts covered later in the chapter (e.g., `handlers` and `typemappings`).

Service Element. This describes the overall service. Only one service can be defined in the XML descriptor, to prevent potential name clashes in the generated code for the different services and the types they use.

- `name`. The name of the service. This is also used as the value for the `service` element in the generated WSDL.
- `package`. The package name for the generated service classes. `xrpcc` generates the stubs with the same package name as the service interface.
- `targetnamespace`. The target namespace for the generated WSDL document.
- `typenamespace`. The namespace for the schema portion of the generated WSDL document.

Interface element. This defines details about the interface the service supports. A service can have multiple interfaces.

- `name`. Fully qualified name of an interface, such as `com.flutebank.billpay.Billpay`.
- `servant`. Fully qualified name of the service interface implementation.
- `soapAction`. Value to be used as the `SOAPAction` for all operations in the corresponding port (optional).
- `soapaActionBase`. Value used as a prefix for the `SOAPAction` strings for the operations in the corresponding port (optional).

Handlerchain element. Defines information about handlers for this service. The `handler` element can be defined inside a service. If so, it is available to all interfaces inside the `interface` element, in which case it is specific only to that interface.

- `runAt`. Defines where the handler is to be executed. Possible values are `client` or `server`.
- `roles`. Lists or defines the roles that the handler will run as. This is the whitespace-separated `List (xsd:anyURI)`value returned by `HandlerChain.getRoles()`.
- `className`. Fully qualified name of the handler class.

- `headers`. The header blocks processed by the handler. This is the whitespace-separated `List(xsd:QName)`-qualified name of a header block's outermost element.
- `property`. Multiple and arbitrary name-value pairs the handler can use internally, such as configuration and initialization parameters. These properties are passed as input to `Handler.init(HandlerInfo config)` through the "`config`" argument. The `HandlerInfo.getHandlerConfig()` method returns a `Map` containing all property name-value pairs specified in the `<property/>` elements.

Typemapping registry element.

- `import`. Specifies a list of schema documents to import and is used to generate the corresponding `<wsdl:import/>` and `<schema:import/>` elements.
- `typeMapping`. Contains one or more entry elements.
- `entry`. Specifies the `encodingStyle`, `schemaType`, Java class, and class for the serializer and deserializer factories.
- `additionalTypes`. Specifies a list of Java classes that do not appear in the remote interface but are still passed to JAX-RPC. For example, if in a method with a signature

  ```
  public java.util.List getPaymentDetails() throws RemoteException;
  ```

  contains `PaymentDetail` objects, and the `PaymentDetail` class is not referenced by any other method in the remote interface, then for `xrpcc` to generate and register a serializer for the `PaymentDetail` type, this element must be specified:

  ```
  <additionalTypes>
       <class name="com.flutebank.PaymentDetail "/>
  </additionalType>
  ```

Namespace mapping registry.

- `namespaceMapping`

The `–both` option of the `xrpcc` tool can be used to generate stubs and ties together. Alternatively, the server and client code can be generated separately, using the `–server` and `–client` options. Note that the `–keep` option must be used to retain the WSDL file.

One of the artifacts `xrpcc` generates when it reads the XML descriptor is an *additional* configuration file. So, what is this new configuration file? Remember,

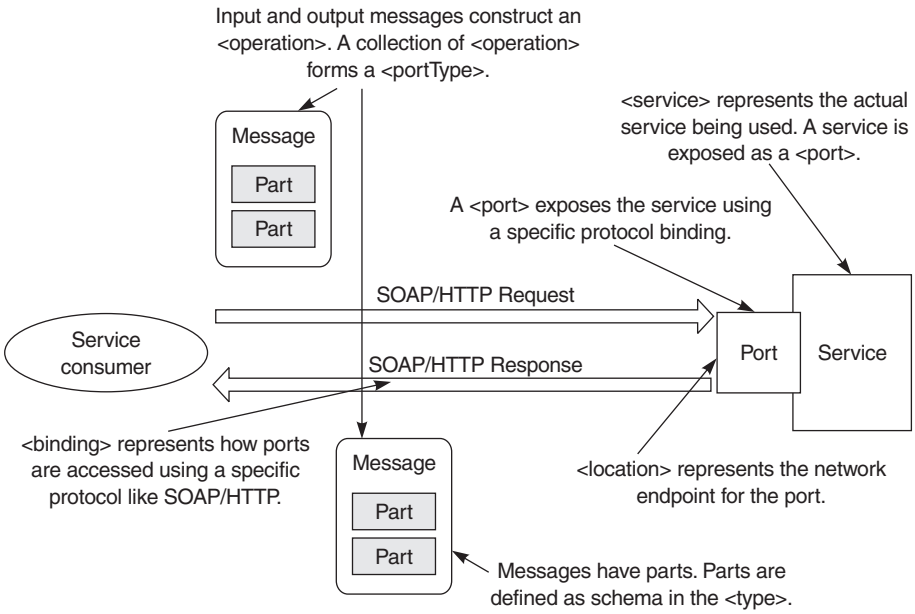

Figure 10.6 WSDL elements and dynamic interaction of a service and its consumer.

the service is being deployed in a servlet container, where the runtime-provided endpoint exists (the reference implementation defined the endpoint as a `com.sun.xml.rpc.server.http.JAX-RPCServlet`). This configuration file is used to hook the endpoint with the service implementation. It is just an implementation detail that, like `xrpcc` itself, is specific to the reference implementation and is not a part of the specifications. Other vendors may use a completely different tool with its own mechanism. ▷

Java-WSDL Mappings

In Chapter 4, we discussed the WSDL structure, the role of vendor tools, and the significance of a standard specification to map WSDL elements to Java (and vice versa). To understand this mapping, let us revisit the role of WSDL elements from that chapter (Figure 10.6).

A Web service exposes groups of business `operations` for service consumers to use. Operations are grouped together to form `portTypes`. To invoke an opera-

▷ The final version of the Hava WSDP and JAX pack include a tool called wscompile. Currently there is no difference in the behavior of wscompile and xrpcc; however in future versions wscompile is likely to evolve, whereas xrpcc may not.

tion, the consumer sends an `input` message containing the input data. It gets an `output` message containing the data that results from the business processing, or a `fault` if a problem occurs. The `input` and `output` messages may have multiple data items in them; each is called a `part`.

The wire protocol used for the invocation and the format of the `input` and `output` messages on the wire for that protocol are specified in a `binding` element. The service exposes itself to consumers through one or more `ports`, each of which specifies a network address where the service is `located` and the `binding` to use with that port. A service may render itself though several ports, where each port has a different binding (e.g., the same service may expose itself via SOAP/HTTP and SOAP/SMTP).

JAX-RPC defines the mapping of Java to WSDL data types, and vice versa. This is the mapping used by `xrpcc` when generating a WSDL file or consuming it. Table 10.6 summarizes this mapping. Listings 10.4 and 10.5 show a complete example of a service definition and its corresponding WSDL generated on the basis of these mappings.

Table 10.6 Data Type Mapping between Java and WSDL

Java type	WSDL mapping
Package	`WSDL document`

Sample extract code

The namespace definition in a WSDL is mapped to a Java package name.

Java type	WSDL mapping
Interface	`wsdl:portType`

Sample extract code

```
public interface BillPay extends java.rmi.Remote {
// methods here
}
```

```
<portType name="BillPay">
   // operations here
</portType>
```

Text continued on p. 355

Table 10.6 Data Type Mapping between Java and WSDL (Cont'd)

Java type	WSDL mapping
Method	`wsdl:operation` 1. The WSDL operation name is the same as the method name. 2. Overloaded methods can map to multiple operations with the same name or unique names that are implementation-specific.

Sample extract code

```
public interface BillPay extends java.rmi.Remote {
   public PaymentDetail[] listScheduledPayments()
       throws RemoteException;
   public PaymentConfirmation schedulePayment(
       Date date, String payee, double amt)
    throws ScheduleFailedException, RemoteException;

public double getLastPayment(String nickname) throws
                                    RemoteException;
}
```

```
<portType name="BillPay">
   <operation name="listScheduledPayments">
     // input output messages for this operation
</operation>
   <operation name="schedulePayment">
     // input output messages for this operation
   </operation>
<operation name="getLastPayment">
     // input output messages for this operation
</portType>
```

Java type	WSDL mapping
Extended interface	`wsdl:portType` with a complete set of inherited operations.

(Continued)

Table 10.6 Data Type Mapping between Java and WSDL (Cont'd)

Sample extract code

```
Public interface LinkedBillPay extends BillPay {
public String getStatus() throws
java.rmi.RemoteException, StatusUnavilableException;
}
<portType name="LinkedBillPay">
   <operation name="listScheduledPayments">
     // input output messages for this operation
</operation>
   <operation name="schedulePayment">
     // input output messages for this operation
   </operation>
   <operation name="getStatus">
     // input output messages for this operation
   </operation>
   </operation>
</portType>
```

Java type	WSDL mapping
Method arguments	`wsdl:input` and corresponding `wsdl:message` elements.

Sample extract code

```
public interface BillPay extends java.rmi.Remote {
   public PaymentDetail[] listScheduledPayments()
       throws RemoteException;
public PaymentConfirmation schedulePayment(Date
date, String payee, double amt)
    throws ScheduleFailedException, RemoteException;
public double getLastPayment(String nickname) throws
RemoteException;
}

  <portType name="BillPay">
    <operation name="getLastPayment"">
      <input message="tns:BillPay_getLastPayment"/>
      // output message
```

Table 10.6 Data Type Mapping between Java and WSDL (Cont'd)

```
</operation>
    <operation name="listScheduledPayments">
      <input message="tns:BillPay_listScheduledPayments"/>
      // output message
</operation>
  <operation name="schedulePayment" >
    <input message="tns:BillPay_schedulePayment"/>
    // output message
</operation>
</portType>

<message name="BillPay_getLastPayment">
    <part name="String_1" type="xsd:string"/></message>
  <message name="BillPay_listScheduledPayments"/>
  <message name="BillPay_schedulePayment">
    <part name="Date_1" type="xsd:dateTime"/>
    <part name="String_2" type="xsd:string"/>
    <part name="double_3" type="xsd:double"/></message>
```

Java type	WSDL mapping
Method returns	`wsdl:output` and corresponding `wsdl:message` elements.

Sample extract code

```
public interface BillPay extends java.rmi.Remote {
   public PaymentDetail[] listScheduledPayments()
       throws RemoteException;
public PaymentConfirmation schedulePayment(Date date, String payee,
                                                    double amt)
    throws ScheduleFailedException, RemoteException;
public double getLastPayment(String nickname) throws
RemoteException;
```

```
<message name="BillPay_getLastPaymentResponse">
   <part name="result" type="xsd:double"/></message>
<message name="BillPay_listScheduledPaymentsResponse">
   <part name="result" type="tns:ArrayOfPaymentDetail"/></message>
```

(Continued)

Table 10.6 Data Type Mapping between Java and WSDL (Cont'd)

```
<message name="BillPay_schedulePaymentResponse">
   <part name="result" type="tns:PaymentConfirmation"/></message>
<portType name="BillPay">
   <operation name="getLastPayment" parameterOrder="String_1">
        // input message here
   <output message="tns:BillPay_getLastPaymentResponse"/>
                                                        </operation>
   <operation name="listScheduledPayments" parameterOrder="">
        // input message here
       <output message="tns:BillPay_listScheduledPaymentsResponse"/>
                                                        </operation>
   <operation name="schedulePayment" parameterOrder="Date_1
                                                  String_2 double_3">
       <output message="tns:BillPay_schedulePaymentResponse"/>
</operation></portType>
```

Java type	WSDL mapping
Checked exceptions	`wsdl:fault` 1. `wsdl:message` name is the same as the exception name. 2. `RemoteExceptions` are mapped to standard SOAP faults. 3. The exception and its hierarchies get mapped to XML types in the schema, using the standard `complexType` extension mechanism.

Sample extract code

```
public interface BillPay extends java.rmi.Remote {
   // other code
public PaymentConfirmation schedulePayment(Date
date, String payee, double amt)
    throws ScheduleFailedException, RemoteException;
 }
```

Table 10.6 Data Type Mapping between Java and WSDL (Cont'd)

```
<operation name="schedulePayment">
      // input  and output elements
   <fault name="ScheduleFailedException">
      <soap:fault encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/" use="encoded" namespace="http://www.flutebank.com/xml"/>
                                                              </fault>
      <soap:operation soapAction=""/>
</operation>
<message name="ScheduleFailedException">
   <part name="ScheduleFailedException" type=
                          "tns:ScheduleFailedException"/></message>
```

Java type	WSDL mapping
`Java identifiers`	XML name.

Sample extract code

Java identifiers are already legal XML names.

```
public class PaymentDetail {
  private String payeeName;
  private String account;
  private double amt;
  private Date date;
  // other code
```

```
<types>
// other code
<complexType name="PaymentDetail">
        <sequence>
          <element name="date" type="dateTime"/>
          <element name="account" type="string"/>
          <element name="payeeName" type="string"/>
          <element name="amt" type="double"/></sequence>
                                                    </complexType>
</types>
```

Listing 10.4 Source file for `BillPay.java`

```
package com.flutebank.billpayservice;

import java.util.*;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface BillPay extends Remote {

public PaymentConfirmation schedulePayment(Date date, String nickName, double amount)
throws ScheduleFailedException, RemoteException;

public PaymentDetail[] listScheduledPayments() throws RemoteException;

public double getLastPayment(String nickname) throws RemoteException;
}
```

Listing 10.5 WSDL `billservice.java` corresponding to Listing 10.4

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="billpayservice" targetNamespace="http://www.flutebank.com/xml"
xmlns:tns="http://www.flutebank.com/xml" xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/
wsdl/soap/">
    <types>
        <schema targetNamespace="http://www.flutebank.com/xml" xmlns:wsdl="http://
schemas.xmlsoap.org/wsdl/" xmlns:tns="http://www.flutebank.com/xml" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xmlns:soap-enc="http://schemas.xmlsoap.org/soap/
encoding/" xmlns="http://www.w3.org/2001/XMLSchema">
            <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
            <complexType name="ArrayOfPaymentDetail">
               <complexContent>
                  <restriction base="soap-enc:Array">
                        <attribute ref="soap-enc:arrayType" wsdl:arrayType=
                                                              "tns:PaymentDetail[]"/>
                  </restriction>
               </complexContent>
            </complexType>
            <complexType name="PaymentDetail">
               <sequence>
                  <element name="date" type="dateTime"/>
```

```
            <element name="account" type="string"/>
            <element name="payeeName" type="string"/>
            <element name="amt" type="double"/>
            </sequence>
         </complexType>
         <complexType name="PaymentConfirmation">
            <sequence>
               <element name="confirmationNum" type="int"/>
               <element name="payee" type="string"/>
               <element name="amt" type="double"/>
               </sequence>
         </complexType>
         <complexType name="ScheduleFailedException">
            <sequence>
               <element name="message" type="string"/>
               <element name="localizedMessage" type="string"/>
            </sequence>
         </complexType>
      </schema>
  </types>
  <message name="BillPay_getLastPayment">
     <part name="String_1" type="xsd:string"/>
  </message>
  <message name="BillPay_getLastPaymentResponse">
     <part name="result" type="xsd:double"/>
  </message>
  <message name="BillPay_listScheduledPayments"/>
  <message name="BillPay_listScheduledPaymentsResponse">
     <part name="result" type="tns:ArrayOfPaymentDetail"/>
  </message>
  <message name="BillPay_schedulePayment">
     <part name="Date_1" type="xsd:dateTime"/>
     <part name="String_2" type="xsd:string"/>
     <part name="double_3" type="xsd:double"/>
  </message>
  <message name="BillPay_schedulePaymentResponse">
     <part name="result" type="tns:PaymentConfirmation"/>
  </message>
  <message name="ScheduleFailedException">
     <part name="ScheduleFailedException" type="tns:ScheduleFailedException"/>
  </message>
```

```
<portType name="BillPay">
   <operation name="getLastPayment" parameterOrder="String_1">
      <input message="tns:BillPay_getLastPayment"/>
      <output message="tns:BillPay_getLastPaymentResponse"/>
   </operation>
   <operation name="listScheduledPayments" parameterOrder="">
      <input message="tns:BillPay_listScheduledPayments"/>
      <output message="tns:BillPay_listScheduledPaymentsResponse"/>
   </operation>
   <operation name="schedulePayment" parameterOrder="Date_1 String_2
                                                              double_3">
      <input message="tns:BillPay_schedulePayment"/>
      <output message="tns:BillPay_schedulePaymentResponse"/>
      <fault name="ScheduleFailedException" message=
                                          "tns:ScheduleFailedException"/>
   </operation>
   </portType>
<binding name="BillPayBinding" type="tns:BillPay">
   <operation name="getLastPayment">
      <input>
          <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     use="encoded" namespace="http://www.flutebank.com/xml"/>
      </input>
      <output>
         <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     use="encoded" namespace="http://www.flutebank.com/xml"/>
     </output>
     <soap:operation soapAction=""/>
</operation>
<operation name="listScheduledPayments">
   <input>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     use="encoded" namespace="http://www.flutebank.com/xml"/>
   </input>
   <output>
         <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     use="encoded" namespace="http://www.flutebank.com/xml"/>
   </output>
      <soap:operation soapAction=""/>
</operation>
```

```
        <operation name="schedulePayment">
           <input>
              <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                         use="encoded" namespace="http://www.flutebank.com/xml"/>
           </input>
           <output>
              <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                         use="encoded" namespace="http://www.flutebank.com/xml"/>
           </output>
           <fault name="ScheduleFailedException">
              <soap:fault encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                         use="encoded" namespace="http://www.flutebank.com/xml"/>
           </fault>
           <soap:operation soapAction=""/>
        </operation>
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
    </binding>
    <service name="Billpayservice">
       <port name="BillPayPort" binding="tns:BillPayBinding">
          <soap:address location="http://127.0.0.1:9090/billpayservice/jaxrpc/BillPay"/>
       </port>
    </service>
</definitions>
```

5. Service Consumption

Until now, we have seen how to define, implement, and deploy a JAX-RPC service. Let us now look at how such a service can be consumed. A *service consumer* represents the abstraction of the entity invoking the facilities of an existing service. Invocation modes for doing so fall into three broad categories:

- **Synchronous request-response.** The client invokes a remote procedure and blocks until a response or an exception is received from the service. The client cannot do any other work while awaiting the response. This is analogous to making a phone call. Either someone responds by picking up the handset on the other end, or a busy tone is received.
- **One-way RPC.** The client invokes a remote procedure but does not block or wait to receive a return and is free to do other work. In fact the client does not

receive any return parameters. This is analogous to sending a fax (fire and forget!). When a fax is sent, a person does not need to pick up the phone on the receiving end for the fax to go through.

- **Nonblocking RPC invocation.** The client invokes a remote procedure and continues processing without waiting for a return. The client may process the return later by polling some service or by using some other notification mechanism. This is analogous to making a phone call and getting an answering machine. The caller leaves a message and continues. The person on the other end gets the message and returns the call by dialing the number left on the machine or a number he or she already knows.

The significant difference between one-way and nonblocking invocation is that in the former, the client will not receive a return value.

As a bare minimum, JAX-RPC implementations must support the first two modes for client invocation and HTTP 1.1 as the transport binding for SOAP. The sematics of nonblocking RPC are quite complicated. For example, the client must inform the service of an endpoint to which the service can repond, and both parties must deal with issues of reliability and availability. If your application requires asynchronous communication, messaging is probably more appropriate. See Chapter 11 for details.

Let us now look at the mechanisms an RPC client can use to consume the service in these invocation modes. The client can be written to invoke the service using one of the following three mechanisms:

- Stub
- Dynamic invocation interface
- Dynamic proxies

In Chapter 5, we described WSDL use cases and early/late binding patterns associated with them. The reader is encouraged to revisit that section before continuing. Recall the usage patterns:

- Static compile-time binding
- Static deploy-time binding
- Static runtime binding

- Dynamic binding
- Dynamic binding with known location

The examples of clients in the following sections show how some of these patterns can be realized.

Clients Using Stubs

Figure 10.1 introduced the concept of stubs. Clients locate the service endpoint by specifying a URI, then simply invoke the methods on a local object, a stub that represents the remote service. JAX-RPC stubs, or *proxies,* as they are sometime referred to, are very different from RMI-IIOP stubs. Keep the following in mind:

- A stub is never required to be downloaded or distributed to clients.
- A client is not a required artifact on the client side. The end result of the invocation is that the required SOAP envelope must be sent on the transport protocol. The client can be written in a completely different programming language, as shown later in the JAX_RPC Interoperability section.
- The stub is implemented in Java and is relevant only for a JAX-RPC client runtime.
- A stub can be dynamically generated by the client side at runtime.
- A stub is specific to the client runtime.
- A stub is specific to a protocol and transport.
- A stub must implement the `javax.xml.rpc.Stub` interface.

The tie represents the server-side skeleton for the implementation. It is used by the endpoint to communicate with the implementation and is generated using tools (such as `xrpcc`) when the implementation is deployed.

Using stubs is also sometime referred to as *static* invocation, because the stub must know the remote interface about the service at *compile* time. It must have the class file representing the remote interface and the implementation available for stub generation to proceed. The client does not need the WSDL file describing the service at runtime. Stubs are specific to a particular runtime and are not portable across vendor implementations.

The code in Listing 10.6 shows the fragment for invoking the `Billpayservice` developed previously.

Listing 10.6 Client using stubs

```
// import generated xrpcc classes + interface class + Helper classes for interface
  import com.flutebank.billpayservice.*;
  import java.util.Date;

public class StubClient {
    public static void main(String[] args) throws Exception {

   String endpoint="http://127.0.0.1:8080/billpayservice/jaxrpc/BillPay";
    String namespace = "http://www.flutebank.com/xml";
    String wsldport = "BillPayPort";
    Billpayservice_Impl serviceproxy= new Billpayservice_Impl();
    BillPay_Stub stub=(BillPay_Stub)(serviceproxy.getBillPayPort());
    stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,endpoint);
    PaymentConfirmation conf= stub.schedulePayment(new Date(),
                                                    "my account at sprint", 190);
      System.out.println("Payment was scheduled " + conf.getConfirmationNum());
      PaymentDetail detail[]=stub.listScheduledPayments();
     for(int i=0;i<detail.length;i++) {
           System.out.println("Payee name "+ detail[i].getPayeeName());
           System.out.println("Account "+ detail[i].getAccount());
           System.out.println("Amount "+ detail[i].getAmt());
           System.out.println("Will be paid on " + detail[i].getDate());
           }
     double lastpaid= stub.getLastPayment("my cable tv provider");
     System.out.println("Last payment was " + lastpaid);
   }
}
```

Before using a stub, a client must first obtain a reference to it. The exact mechanism is specific to the implementation. The reference implementation for the stub is obtained by instantiating the service implementation class. The code below shows the mechanism another vendor might use:

```
InitialContext ctx = new InitialContext();
Billpayservice service =
      (Billpayservice) ctx.lookup("myserver:soap:Billpayservice");
BillPay bill = service. getBillPayPort ();
Stub stub= ((Stub) bill;
```

The stub can be configured by passing it name-value pairs of properties. The `javax.xml.rpc.Stub` interface defines four standard properties to configure the stub, using the `stub. _setProperty(java.lang.String name, java.lang.Object value)` method:

- `javax.xml.rpc.security.auth.username`. Username for authentication.
- `javax.xml.rpc.security.auth.username.password`. Password for authentication.
- `javax.xml.rpc.service.endpoint.address`. Optional string for the endpoint service.
- `javax.xml.rpc.session.maintain`. Use `java.lang.Boolean` to indicate that the server needs to maintain session for the client.

Clients Using DII

The second way a consumer can access a service involves the use of dynamic invocation interface (DII) instead of *static* stubs. DII is a concept that, like most other things in JAX-RPC, should be familiar to CORBA developers. Unlike static invocation, which requires that the client application include a client stub, DII enables a client application to invoke a service whose data types were unknown at the time the client was compiled. This allows a client to discover interfaces dynamically—in other words, at runtime rather than compile time—and invoke methods on objects that implement those interfaces.

JAX-RPC supports DII with the `javax.xml.rpc.Call` interface. A `Call` object can be created on a `javax.xml.rpc.Service` using the `port name` and `service name`. Then, during runtime, the following details are set:

- Operation to invoke
- Port type for the service
- Address of the endpoint
- Name, type, and mode (`in`, `out`, `inout`) of the arguments
- Return type

This information is derived by looking at the WSDL file for the service. For example, the service name is the `service name="Billpayservice">` element, the `portname` is the `port name="BillPayPort"` element, and so on. Listing 10.7 shows a DII client where a `Call` object is configured for the `getLastPayment` method.

The client code wraps the DII request in a Call object. DII can be used directly, by passing these values (`port`, `operation`, `location`, and `part` information) to the `Call`, or indirectly, by passing the WSDL to the `Call`. Listing 10.7 shows how a DII client can be written using the former. (`QName` is a common class used to represent a qualified name in different XML APIs. The qualified name of an XML element consists of its namespace declaration and its local name in the namespace.)

Listing 10.7 Client using DII directly, where all parameters are known (WSDL is not passed)

```
import javax.xml.namespace.QName;
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.ParameterMode;
import javax.xml.rpc.ServiceFactory;

public class DIIClient_NoWSDL{
  public static void main(String[] args) throws Exception {

     String endpoint="http://127.0.0.1:9090/billpayservice/jaxrpc/BillPay";
     String namespace = "http://www.flutebank.com/xml";
     String schemanamespace = "http://www.w3.org/2001/XMLSchema";
     String serviceName = "Billpayservice";

     ServiceFactory factory = ServiceFactory.newInstance();
     // the Billpayservice service does not exist
     // (no stub, skeleton, or Service was generated by xrpcc)
     // but createService will return a Service object
     // that can be used to create the dynamic call

    Service service = (Service) factory.createService
                                              (new QName(namespace,serviceName));

     QName portName =      new QName(namespace,"BillPayPort");
     QName operationName = new QName(namespace,"getLastPayment");
     Call call = service.createCall(portName, operationName);
     call.setTargetEndpointAddress(endpoint);
     call.setProperty(Call.ENCODINGSTYLE_URI_PROPERTY,
                      "http://schemas.xmlsoap.org/soap/encoding/");

     QName paramtype = new QName(schemanamespace, "string");
```

```
     QName returntype = new QName(schemanamespace, "double");

     call.addParameter("String_1", paramtype, ParameterMode.IN);
     call.setReturnType(returntype);

     Object[] params = {"my cable tv provider"};
     Object lastpaid= (Double)call.invoke(params);
     System.out.println("Last payment was " + lastpaid);
    }
}
```

What is relevant in Listing 10.7 is that there is no coupling between the service interface and the client (e.g., see `import` statements).

In indirect DII, only the `port` and `operation` names are knowm at compile time. The runtime will determine the `type` information about the `part` and `location`, based on the WSDL. In this case, the parameters and return types do *not* need to be configured using the `addParameter` or `setReturnType` method. Listing 10.8 shows a sample DII client using the WSDL.

Listing 10.8 Client using DII indirectly, where all parameters are not known (WSDL is dynamically inspected)

```
public class DIIClient_WSDL{

    public static void main(String[] args) throws Exception {

  String wsdllocation= http://127.0.0.1:9090/billpayservice/billpayservice.wsdl";

    String namespace = "http://www.flutebank.com/xml";
    String serviceName = "Billpayservice";

    ServiceFactory factory = ServiceFactory.newInstance();
    Service service = (Service) factory.createService
                      (new URL(wsdllocation),new QName(namespace,serviceName));

    QName portName = new QName(namespace,"BillPayPort");
    QName operationName = new QName(namespace,"getLastPayment");
    Call call = service.createCall(portName, operationName);
```

```
    Object[] params = {"my cable tv provider"};
    Object lastpaid= (Double)call.invoke(params);
    System.out.println("Last payment was " + lastpaid);
   }
}
```

Note that neither use of DII generates stubs.

WSDL with DII. When deciding whether to use WSDL or not in the client, keep in mind that though it may be more convenient to use, it requires an extra network call and processing overhead for the runtime to fetch and process the WSDL and perhaps even validate the call against the WSDL.

One of the major differences between static invocation and dynamic invocation is that, while both support synchronous communication, only DII supports one-way communication. From an API perspective, instead of using the `invoke()` method, DII can be used to invoke the `invokeOneWay(java.lang.Object[] inputParams)` method. Attempting to invoke a `call.getOutputParams()` in a one-way invocation will result in a `JAX-RPCException`.

Clients Using Dynamic Proxies

The JAX-RPC specification also specifies a third way for clients to access services: using the concept of dynamic proxy classes available in the standard J2SE Reflection API (the `java.lang.reflect.Proxy` class and the `java.lang.reflect.InvocationHandler` interface). A dynamic proxy class implements a list of interfaces specified at runtime. The client can use this proxy or façade as though it actually implemented these interfaces, although it actually delegates the invocation to the implementation.

Classes allowing any method on any of these interfaces can be called directly on the proxy (after casting it). Thus, a dynamic proxy class is used to create a type-safe proxy object for an interface list without requiring pregeneration of the proxy class, as you would with compile-time tools. Listing 10.9 shows how a client can use dynamic proxies.

Listing 10.9 Client using dynamic proxies

```
// jaxrpc classes
import javax.xml.namespace.QName;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
```

```
// java classes
import java.util.Date;
import java.net.URL;
// Interface class
import com.flutebank.billpayservice.BillPay;

public class DynamicProxyClient {
  public static void main(String[] args) throws Exception{
     String namespace = "http://www.flutebank.com/xml";
     String wsldport = "BillPayPort";
     String wsdlservice = "Billpayservice";
     String wsdllocation =
                          "http://127.0.0.1:8080/billpayservice/billpayservice.wsdl";
     URL wsldurl = new URL(wsdllocation);
     ServiceFactory factory = ServiceFactory.newInstance();
     Service service = factory.createService(wsldurl,
                                                new QName(namespace, wsdlservice));
// make the call to get the stub corresponding to this service and interface
        BillPay stub = (BillPay) service.getPort(new QName(namespace,wsldport),
                                                                    BillPay.class);
// invoke methods on the service
        double lastpaid= stub.getLastPayment("my cable tv provider");
        System.out.println("Last payment was " + lastpaid);
    }
}
```

In Listing 10.9, there is no compile-time stub generation. The `getPort()` method will return the proxy, which is also required to implement the `Stub` interface at runtime–that is, the stub is generated internally at runtime. Again, CORBA developers will see the similarity in the above code with its counterpart:

```
BillPay stub = (BillPay)PortableRemoteObject.narrow(initial.lookup("Billpayservice"),
                                                                    BillPay.class);
```

Clients Using WSDL

Until now, we have seen how to start with a Java service definition and implement it as an XML-RPC Web service. One could also do the reverse:

- *Start* with a WSDL file for an existing service and generate the stubs, to consume the service.
- *Start* with a WSDL file and generate the ties (and stubs if needed) and remote interfaces, and fill in appropriate business logic to implement the service.

Let us look at how to consume the `Billpayservice` Web service using the service's WSDL and the WSDL 1.1–compliant `xrpcc` tool. The client-side bindings are generated from the WSDL using `xrpcc`, with only a configuration file change:

```
<?xml version="1.0" encoding="UTF-8"?>
      <configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
          <wsdl location=http://127.0.0.1:9090/billpayservice/billpayservice.wsdl
                      packageName="generated">
                  </wsdl>
      </configuration>
```

The client-side code is identical to the `StubClient` shown previously, except that

- The client no longer depends on the server interfaces but is coupled to the tool-*generated* classes–which, from the above configuration, reside in a generated package (see `import` statement).
- The `Date` parameter has been changed to `Calendar`, as per the date type mappings from XML to Java specified in Tables 10.3a and 10.3b.
- The endpoint does not need to be configured (unless you want to) and is picked from the `soap:address` element in the WSDL by the tool.

```
import generated.*; // generated classes by xrpcc from WSDL file
import java.util.Calendar;

public class WSDLClient {
    public static void main(String[] args) throws Exception {
        String namespace = "http://www.flutebank.com/xml";
        String wsldport = "BillPayPort";
        Billpayservice_Impl serviceproxy= new Billpayservice_Impl();
        BillPay_Stub stub=(BillPay_Stub)(serviceproxy.getBillPayPort());
        PaymentConfirmation conf=stub.schedulePayment(Calendar.getInstance(),
                                                "my account at sprint", 190);
        System.out.println("Payment was scheduled " +
                                                  conf.getConfirmationNum());
```

```
        PaymentDetail detail[]=stub.listScheduledPayments();
        for(int i=0;i<detail.length;i++) {
          System.out.println("Payee name "+ detail[i].getPayeeName());
          System.out.println("Account "+ detail[i].getAccount());
          System.out.println("Amount "+ detail[i].getAmt());
          System.out.println("Will be paid on " +
                                              detail[i].getDate().getTime());
          }
          double lastpaid= stub.getLastPayment("my cable tv provider");
          System.out.println("Last payment was " + lastpaid);
    }
}
```

What Client Is Right for Me?

Choosing either option shown above to implement the client affects only client-side development. When a server method is invoked, that server has no knowledge of whether a method was invoked via the conventional static stub mechanism, through DII, through proxies, or even by a non-Java client. From the server's perspective, it receives a SOAP request and generates a SOAP response; these are identical for all client types. For example, Listings 10.10a and 10.10b show SOAP request and response messages for the `getLastPayment()` method, which is identical for stubs, DII (with or without WSDL), dynamic proxies, or WSDL.

Listing 10.10a SOAP request

```
POST /billpayservice/jaxrpc/BillPay HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: 506
SOAPAction: ""
User-Agent: Java1.3.1_01
Host: 127.0.0.1:9090
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns0="http://
www.flutebank.com/xml" env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
```

```
<env:Body>
<ns0:getLastPayment>
    <String_1 xsi:type="xsd:string">my cable tv provider</String_1>
</ns0:getLastPayment>
</env:Body>
</env:Envelope>
```

Listing 10.10b SOAP response

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
SOAPAction: ""
Transfer-Encoding: chunked
Date: Mon, 29 Jul 2002 19:28:50 GMT
Server: Apache Coyote HTTP/1.1 Connector [1.0]

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns0="http://
www.flutebank.com/xml" env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <env:Body>
      <ns0:getLastPaymentResponse>
          <result xsi:type="xsd:double">829.0</result>
      </ns0:getLastPaymentResponse>
   </env:Body>
</env:Envelope>
```

In most practical situations, an enterprise will develop a service and publish its WSDL. Service consumers will use the WSDL and a vendor-provided tool (such as `xrpcc`) to generate client-side bindings and invoke the service. This has several advantages. There is no distribution of client code (e.g., remote interfaces), and in most cases, the tool will generate the serializers and deserializers, using the encoding scheme. For example, `xrpcc` generates the serializers and deserializers using the SOAP encoding scheme for `PaymentDetail[]` and `PaymentDetail` and maps the supported XML Schema types to Java.

Using stubs directly has the disadvantage of having to share the Java interface (and interface-dependent classes) with the service consumer. However, in scenarios where services will be developed within the boundaries of the enterprise, static stubs are the preferred client model, along the same lines as above. The performance with stubs is also expected to be better, since all type casting information is built in. All that occurs at runtime is service invocation.

DII is quite attractive, because it allows dynamic creation and invocation of object requests. In most cases, the architect of an application knows the kind of objects the application will need to access, and if not, WSDL should suffice. In some cases, such as object browsers and object brokers, DII is useful, but we don't envision these as frequent.

In practical situations and architecturally, DII is also not completely dynamic. Let us explain this further. Enterprise-level Web services will be coarse-grained and will frequently deal with passing data objects, such as the JavaBean's (e.g., `PaymentDetail` or `PaymentConfirmation`, as in the `StubClient.java` example). Simple data types will not suffice. For a DII client to be able to invoke these services, it *will* need the classes at compile time for the objects being passed around. (For example, if the DII code above invoked the `schedulePayment` method, the result would be a `PaymentConfirmation` object). The question is, where do these classes come from? The alternatives include using the same classes as the service, producing a coupling, or producing the classes from WSDL, using a tool (`xrpcc`). Further, if the data type that needs to be passed around is a custom type and not a JavaBean (e.g., a vector), a serializer and deserializer would need to be written for it. All this offsets the benefits DII offers of being a "dynamic invocation" at runtime.

> Advanced JAX-RPC

Attachments in JAX-RPC

A SOAP message may also contain one or more attachments using the MIME encoding, as Listing 10.11 shows. This is often refered to as a *compound* message. The attachments are referenced in the SOAP message with an HREF, analogous to how HTML anchor tags are used to create links on the same Web page. The special characters in Listing 10.11 are the binary content of the attachment printed as text.

Listing 10.11 A compound message with a MIME attachment

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope... >
<env:Body>
<storeDocumentService>
        <!--Some XML Here-->
        <something xsi:type="ns1:something" href="cid:ID1"/>
```

```
</storeDocumentService>
</env:Body>
</env:Envelope>

--3317565.1028340932732.JavaMail.Administrator.BYTECODE
Content-Type: application/octet-stream
Content-Id: ID1

ÐÏ_ài±_á
```

Sending information in an attachment rather than in the SOAP message body is more efficient, because smaller SOAP message bodies are processed faster. The message contains only a reference to the data and not the data itself, which reduces the translation time in mapping the data to Java objects. JAX-RPC uses the JavaBeans activation framework for dealing with SOAP attachments. When unmarshalling this message to Java, the JAX-RPC runtime can use either of two mapping techniques:

- It can map well-known MIME types to Java objects, as per Table 10.7, and vice versa, using built-in `DataHandlers` and `DataContentHandlers` in the runtime.
- It can map the attachment to a `javax.activation.DataHandler` using the JavaBeans Activation framework, and vice versa.

What this essentially means is that if a method in a service implementation is exposed in a Web service and has a return type that contains either a Java type, as per the mappings shown in Table 10.2, or a `DataHandler`, the runtime will marshal that as an attachment to the outgoing SOAP message. If the argument is of the type in Table 10.7 or is a `DataHandler`, it will be passed the corresponding attachment from the incoming SOAP message. The content of the attachment can then be extracted using a `getContent()` on the `DataHandler`. If the installed `DataContentHandler` does not understand the content, it will return a `java.io.InputStream` object with the raw bytes. ▷

▷ The JavaBeans Activation framework is a standard extension API originally designed for bean components. It adds support for typing arbitrary blocks of data and handling the content accordingly.

Table 10.7 MIME-to-Java Data Type Mapping

MIME	Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
text/plain	java.lang.String
multipart/*	javax.mail.internet.MimeMultipart
text/xml or application/xml	javax.xml.transform.Source

Let us now look at an example of a Flute Bank Web service that stores and archives any incoming documents it receives from partners. The remote interface defines a single method, as shown in the following code.

```
public interface AttachmentService extends Remote{
    public String storeDocumentService(DataHandler dh,String filename)
                                                    throws RemoteException;
}
```

The service implementation (Listing 10.12a) is also straightforward; it just extracts the content from the `DataHandler` and stores it to a file. It returns a date/timestamp to the caller.

Listing 10.12a Service implementation for processing attachments

```
public class AttachmentServiceImpl implements AttachmentService {
   /**
     * This method implements a web service that stores any attachment it receives.
     */
    public String storeDocumentService(DataHandler dh, String filename) {
       try{
              BufferedOutputStream out = new BufferedOutputStream(new
                                                  FileOutputStream (filename));
              BufferedInputStream in = new BufferedInputStream (dh.getInputStream());

              byte[] buffer = new byte[256];
              while (true) {
         int bytesRead = in.read(buffer);
                     if (bytesRead == -1)
                              break;
```

```
                out.write(buffer, 0, bytesRead);
                  }
                in.close();
                out.close();
                }catch(Exception e){
                             System.out.println(e);
                             return e.toString();
                }
                return ("File processes succesfully " + filename + " " + new Date());
    }
}
```

Listing 10.12b shows the `xrpcc` configuration used to generate stubs and ties.

Listing 10.12b `xrpcc` configuration for stub and tie generation

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration
  xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="attachservice"
     targetNamespace="http://www.flutebank.com/xml"
     typeNamespace="http://www.flutebank.com/xml"
     packageName="com.flutebank.attachmentservice">
     <interface name="com.flutebank.attachmentservice.AttachmentService"
        servantName="com.flutebank.attachmentservice.AttachmentServiceImpl"/>
     </service>
</configuration>
```

The relevant extract from the client code is shown below, where the stub is instantiated and the service invoked:

```
   Attachservice_Impl() service =new Attachservice_Impl();
AttachmentService_Stub stub=(AttachmentService_Stub)
                             (service.getAttachmentServicePort());
stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,url);
   DataHandler dh = new DataHandler(new FileDataSource(filename));
   String response = stub.storeDocumentService(dh,filename);
   System.out.println("Response from server " + response);
```

The SOAP request to the server includes an attachment, as shown below. The MIME segments are highlighted:

```
POST /attachmentservice/jaxrpc/AttachmentService HTTP/1.1
Content-Type: multipart/related; type="text/xml"; boundary=
3317565.1028340932732.JavaMail.Administrator.BYTECODE
Content-Length: 26994
SOAPAction: ""
User-Agent: Java1.3.1_01
Host: 127.0.0.1:9090
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

--3317565.1028340932732.JavaMail.Administrator.BYTECODE
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd=
"http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns0=
"http://www.flutebank.com/xml" xmlns:ns1="http://java.sun.com/jax-rpc-ri/internal"
env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"><env:Body>
<ns0:storeDocumentService><DataHandler_1 xsi:type="ns1:datahandler" href="cid:ID1"/>
<String_2 xsi:type="xsd:string">Uploadme.doc</String_2></ns0:storeDocumentService>
</env:Body></env:Envelope>

--3317565.1028340932732.JavaMail.Administrator.BYTECODE
Content-Type: application/octet-stream
Content-Id: ID1

ÐÏ_à¡±_á
```

`xrpcc` contains an option—`Xdatahandleronly`—that forces attachments to always map to the **DataHandler**, instead of to the mappings shown in Table 10.7.

MessageHandlers *and* HandlerChains

A SOAP message handler is a Java class that provides a filtering mechanism for preprocessing and postprocessing the SOAP message, by intercepting it and acting on the SOAP request and response. As Figure 10.7 shows, a handler can be used on the client side, server side, or both. Handlers can be used to add features to a service call and are a good means to layer additional functionality over the core message. They are useful because they provide the ability to introduce

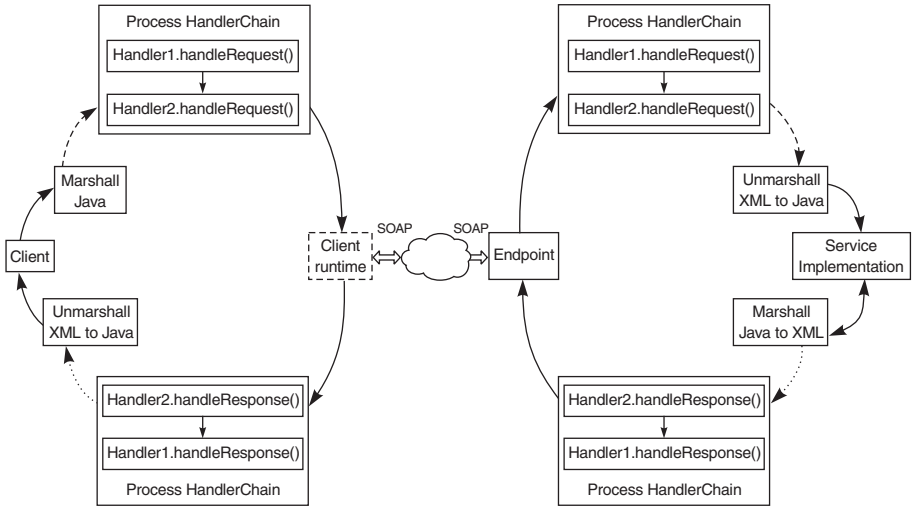

Figure 10.7 Handler architecture

security services, business processing, and error handling. They also permit managing the selection of content creation strategies in both service consumers and service implementations without changing client or server code.

All handler implementations must implement the `javax.xml.rpc.handler.Handler` interface shown below:

```
public interface Handler{
    public abstract void init(HandlerInfo handlerinfo);
    public abstract boolean handleRequest(MessageContext messagecontext);
    public abstract boolean handleResponse(MessageContext messagecontext);
    public abstract boolean handleFault(MessageContext messagecontext);
    public abstract void destroy();
    public abstract QName[] getHeaders();
}
```

The handler is passed an instance of a `MessageContext`, which can be used to access the underlying Soap with Attachments API for Java (SAAJ) `javax.xml.soap.SOAPMessage` that represents the actual message. It can also be used to pass objects between handlers in the chain, to share state information specific to a request. Note that a handler is always stateless itself and should not hold any message-specific state in an instance variable. The lifecycle of a handler instance is quite similar to that of a servlet:

1. The runtime initializes the handler by calling the `init()` method and passing configuration information to the instance via the `HandlerInfo` object. This is a useful place to obtain references to reusable resources.
2. Depending on the stage of request processing, the `handleRequest()`, `handleResponse()`, or `handleFault()` method is invoked.
3. The runtime can call these methods multiple times from different threads that handle different requests and can even pool handler instances for optimization.
4. When the runtime is done or is under resource constraints, it will invoke the `destroy()` method, which is a good place to release the resources obtained in the `init()` method.

Multiple handlers can be combined together an ordered group called a *handler chain*. Chained handlers are invoked in the order in which they are configured. When a handler completes its processing, it passes the result to the next handler in the chain. Chaining and managing communication between handlers in a chain is done by the runtime. Developers write handlers as individual units that do not need to be aware of other handlers and are thus highly reusable.

The order in which handlers are deployed is important. For example, if a client sends an encrypted request in a compressed format, the handlers on the server must first decompress and then decrypt the input. Like individual handlers, chains can be defined on the client, the server, or both. The steps below describe how execution occurs in a chain (see Figure 10.7):

1. The `handleRequest()` methods of the handlers in the chain on the client are all executed, in the order specified. Any of these `handleRequest()` methods might change the SOAP message request.
2. When the `handleRequest()` method of the last handler in the chain has been executed on the client side, the runtime dispatches the request to the server.
3. When the endpoint receives the request, it invokes the `handleRequest()` methods of the handlers in the chain on the server, in the order specified in the chain.
4 When all the handlers are done processing the request, the endpoint delegates the invocation to the service implementation via the tie.
5. When the service has completed its work, the runtime invokes the `handleResponse()` methods of the handlers in the chain on the server, in reverse

order. The last handler to process the request will be the first to process the response. Any of these `handleResponse()` methods might change the SOAP message response.

6. When the client receives the response from the server, the `handleResponse()` methods of the chain on the client are executed in the same reverse manner. Any handler can change the SOAP message.
7. The response is then returned to the client application that invoked the Web service.

In a chain, if any of the `handle` methods in the handler return `true`, the next handler in the chain is invoked.

Request processing can be terminated by returning `false`. As Figure 10.8 shows, developers can throw a `SOAPFaultException` to indicate a SOAP fault or a `JAX-RPCException` and trigger the `handleFault` callbacks in the handler. Table 10.8 describes the main classes and interfaces relevant to handlers.

Handler Advantages

Handlers and handler chains offer a valuable tool to architects. We list below some best practices and usage scenarios for handlers:

- **Introducing security.** A handler can be used to encrypt and decrypt the header or body data, using symmetric or asymmetric ciphering techniques. Clients use a handler to encrypt data before sending the SOAP request. A handler on the server decrypts the data before invoking business components, such as EJBs, and encrypts the outgoing response after business processing occurs.

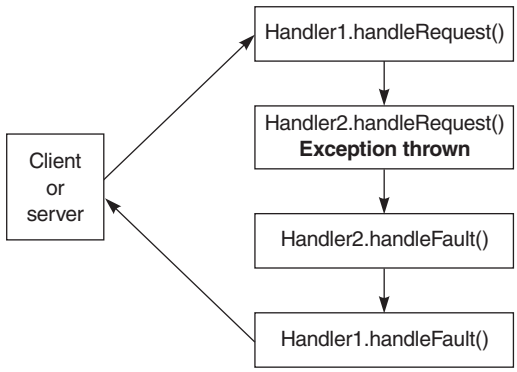

Figure 10.8 Fault handling in handlers

Table 10.8 Handler-Specific API in JAX-RPC

`javax.xml.rpc.handler.Handler`	Must be implemented by a handler class.
`javax.xml.rpc.handler.HandlerInfo`	Contains information about the handler—in particular, the initialization parameters.
`javax.xml.rpc.handler.MessageContext`	Abstracts the message processed by the handler and contains `getProperty(String)` and `setProperty(String,Object)` methods that can be used to share state between handlers in a handler chain. This is analogous to the `pageContext` in JSPs or a `ServletContext` in servlets.
`javax.xml.rpc.handler.soap.SOAPMessageContext`	Extends the `MessageContext` and provides access to the actual SOAP message. It also contains the `getRoles()` method, which returns the SOAP actor roles associated with the `HandlerChain`.
`javax.xml.soap.SOAPMessage`	Object that contains the actual request or response SOAP message, including its header, body, and attachment.
`javax.xml.rpc.handler.HandlerChain`	Implemented by the JAX-RPC implementation to represent a chain. A `HandlerChain` can have SOAP actor roles associated with it.

- **Processing metadata.** A handler can be used to access and manipulate the SOAP header containing metadata or context information about the service invocation or service consumer.
- **Validating data.** Intercepting the request before any processing occurs on the data and validating the request or attachment in the request against a schema, especially for a handling compound messages with XML attachments is best done using handlers.
- **Handling data content.** Handlers can be used to process SOAP attachments–for example, plain text, XML, JPEG images, and octet streams.
- **Optimizing and improving performance.** Handlers can be used to introduce optimizations in service processing by introducing features such as
  - Data caching or result caching for frequently accessed results
  - Prefetching of additional data that may be required during request processing

- Initializing, preparing, and caching resources that may otherwise introduce latencies

- **Implementing intermediaries.** Chapter 4 introduced the concepts of actors, intermediaries, and roles. To recall, a SOAP message may pass though *intermediaries* capable of processing and forwarding the request. The SOAP message may contain *header* information intended only for an intermediary's consumption. The targeted intermediary will process that particular header and ensure that it is not passed along.The SOAP actor attribute is a URI that indicates whom the header intended for. The actor `next` corresponds to a URI of `http://schemas.xmlsoap.org/soap/actor/next` and indicates that the header element is intended for the first SOAP application that processes the message.

  Handlers offer a good mechanism to implement SOAP intermediaries and process headers. When the handler chain executes, the runtime will identify the SOAP *actor roles* for which the chain is configured and ensure that the handlers are passed the header blocks they need. If the processing was unsuccessful or any of the mandatory headers is not present, a corresponding SOAP fault (e.g., a SOAP `MustUnderstand` fault) is generated and propagated back to the client.

Configuring Handlers

Message handlers can be configured in two ways: programmatically, using JAX-RPC API, or declaratively, using a JAX-RPC runtime-provided tool or deployment descriptor. Client-side handlers can be configured either way, but server-side handlers can be be configured only declaratively. The fragment below shows the relevant extract for `xrpcc` in the reference implementation given in Listing 10.3. The `runAt` property can be `client` or `server`, indicating where the handler is to be deployed, and the property fields indicate arbitrary properties (e.g., configuration information) required by the handlers. Multiple handlers can be registered per interface or per service.

```
<handlerChains>
    <chain runAt="client|serverw" roles="">
           <handler className="" headers="">
                  <property name="" value=""/>
           </handler>
    </chain>
</handlerChains>
```

Programmatic registration of handlers on the client can be done in code such as the following:

```
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService (...);
HandlerRegistry registry = service.getHandlerRegistry();
// pass the namespace and portname to get the handler chain object
List chain = registry.getHandlerChain(new QName(...));
Map config =... //configuration poperties
Qname headers[]=... //headers
HandlerInfo info = new HandlerInfo(MyHandler.class, config,headers);
chain.add(info);
```

Let us now look at an example of using handlers. Flute Bank has exposed a service that allows third-party vendors to send sensitive information about customers as a part of a larger business transaction. The code below shows how a handler can be implemented on both the client and server sides to first compress that information and then encrypt it, using password-based symmetric ciphering (PBEWithMD5AndDES).

As Figure 10.9 shows, the client-side handler intercepts the request, compresses the outgoing data, and encrypts it, using a symmetric cipher. (Listing 10.13 uses JCE, the Java Cryptography Extension API bundled with JDK 1.4.) Once this is done, it places the data back in the SOAP message and sends the request on its way to the service.

Listing 10.13 Client-side handler

```
public class SecureZipClientHandler implements Handler {
    private static final byte salt[] = new byte[8];
    private static final int iterations =1;
    private final static String algorithm = "PBEWithMD5AndDES";
    private static SecretKeyFactory skf;
    private static PBEParameterSpec aps;
    private final static char[] password = "1eallysecurepassword".toCharArray();

    public void init(HandlerInfo hi) {
        try {
            // Initialize JCE and the key factory
            Security.addProvider(new com.sun.crypto.provider.SunJCE());
```

```
            skf = SecretKeyFactory.getInstance(algorithm);
            aps = new PBEParameterSpec(salt,iterations);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
/**
The handlerequest method that intercepts the outgoing request from the client
*/
    public boolean handleRequest(MessageContext context) {
        try {
            SOAPMessageContext smc = (SOAPMessageContext)context;
            SOAPMessage msg = smc.getMessage();
            SOAPPart sp = msg.getSOAPPart();
            SOAPEnvelope se = sp.getEnvelope();
    // next step based on the processing model for this handler
            SOAPBody body = se.getBody();
            Iterator it = body.getChildElements();
            SOAPElement opElem = (SOAPElement)it.next();
            it = opElem.getChildElements();
            SOAPElement pin = (SOAPElement)it.next();
            it = pin.getChildElements();
            Text textNode = (Text)it.next();
            textNode.detachNode();
            String encContent = textNode.getValue();

  // Use a utility class to decode the Base64 encoded binary SOAP data
            byte[] contentBytes = Base64.decode(encContent);
            // zip the content
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            GZIPOutputStream zos = new GZIPOutputStream(baos);
            zos.write(contentBytes);
            zos.flush();
            zos.finish();
            zos.close();

   // Encrypt the content
            byte[] zippedbytes = encrypt(baos.toByteArray());

   // Use a utility class to encode the bytes back to the binary SOAP data
```

```
            String zippedContent = Base64.encode(zippedbytes);
            System.out.println("Client handler done with encryption and compression");

   // Add the content to the outgoing message
            pin.addTextNode(zippedContent);
            return true;
    }
    catch (Exception e) {
            System.out.println(e);
            return false;
    }
}

    private static byte[] encrypt(byte[] clear) throws Exception {
        byte[] ciphertext = null;
        PBEKeySpec ks = new PBEKeySpec(password);
        SecretKey key = skf.generateSecret(ks);
        Cipher desCipher = Cipher.getInstance(algorithm);
        desCipher.init(Cipher.ENCRYPT_MODE, key,aps);
        ciphertext = desCipher.doFinal(clear);
        return ciphertext;
    }
/* The handleResponse method does nothing on the response returned from the
 * server. Only outgoing data needs to be encrypted and compressed.
 */
    public boolean handleResponse(MessageContext context) {
        return true;
    }

// Other Handler methods with empty implementations not shown
}
```

The handler on the server side intercepts the request from the endpoint, decrypts the data using the same password as the client, and decompresses the data. It then places the data back on the SOAP request and sends it on the way to the service implementation or tie, as Listing 10.14 shows.

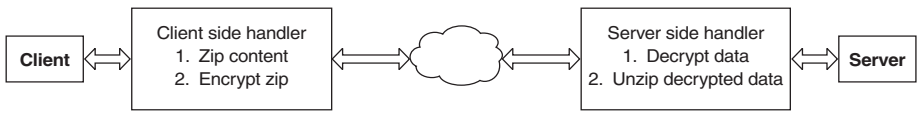

Figure 10.9 Handler example

Listing 10.14 Server-side handler

```
public class SecureZipServerHandler implements Handler {
// member variables are identical to client handler shown previously

    public void init(HandlerInfo hi) {
// Initialize JCE here identically to the client handler shown previously
    }

    public boolean handleRequest(MessageContext context) {
        try {
            SOAPMessageContext smc = (SOAPMessageContext)context;
            SOAPMessage msg = smc.getMessage();
            SOAPPart sp = msg.getSOAPPart();
            SOAPEnvelope se = sp.getEnvelope();

// next step based on the processing model for this handler
            SOAPBody body = se.getBody();
            Iterator it = body.getChildElements();
            SOAPElement op = (SOAPElement)it.next();
            SOAPElement param = (SOAPElement)op.getChildElements().next();
            Text textNode = (Text)param.getChildElements().next();
            String zippedenccontent = textNode.getValue();
            System.out.println(zippedenccontent);
            textNode.detachNode();

// Use a utility class to decode the Base64 encoded binary SOAP data
            byte[] rawbytes = Base64.decode(zippedenccontent);

// First decrypt the data using ciphers
            rawbytes = decrypt(rawbytes);

// unzip the data
            ByteArrayInputStream bais = new ByteArrayInputStream(rawbytes);
            GZIPInputStream zis = new GZIPInputStream(bais);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            int c = -1;
            while ((c = zis.read())!= -1) {
                baos.write(c);
            }
```

```
            baos.flush();
            byte[] contentBytes = baos.toByteArray();
            System.out.println("Server handler done with decryption and
                                                          decompression");
// Use a utility class to encode bytes to Base64 binary SOAP data
            String encContent = Base64.encode(contentBytes);
            param.addTextNode(encContent);
            return true;
        }
        catch (Exception e) {
            System.out.println(e);
            return false;
        }
    }

// method to decrypt bytes
    private static byte[] decrypt(byte[] input) throws Exception {
        byte[] cleartext1 = null;
        PBEKeySpec ks = new PBEKeySpec(password);
        SecretKey key = skf.generateSecret(ks);
        Cipher desCipher = Cipher.getInstance(algorithm);
        desCipher.init(Cipher.DECRYPT_MODE, key,aps);
        cleartext1 = desCipher.doFinal(input);
        return cleartext1;
    }

// The server does not need to process the outgoing response to the client

  public boolean handleResponse(MessageContext context) {
        return true;
    }

// Other Handler methods with empty implementations not shown
}
```

The service implementation in Listing 10.15 is no different from any of the previous examples and requires no additional code. Note that in this case, the service implementation is not aware of any of the changes (compression, encryption, decryption, and decompression) applied to the SOAP message between the time the client initiated the request and the time it was processed.

Listing 10.15 Service implementation

```
public interface Fileservice extends Remote{
    public String acceptContent(byte[] parameter_in) throws RemoteException;
}

public class FileserviceImpl implements Fileservice {

    public String acceptContent(byte[] input) throws RemoteException {
                try {
        BufferedOutputStream fos= new BufferedOutputStream
                                                (new FileOutputStream("Myfile.doc"));
                fos.write(input,0,input.length);
                fos.flush();
                fos.close();
                }catch(Exception e){
                            System.out.println(e);
                }
    return "Data sucessfully processed and timestamped as:"+ new Date();
    }
```

The client code also does not require any modification and remains the same as any of the previous examples:

```
// instantiate the service.
Contentservice_Impl service = new Contentservice_Impl();
Fileservice_Stub stub =(Fileservice_Stub) service.getFileservicePort();
stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,args[0]);

// get the content of file to be sent
byte[] rawbytes = readFile(args[1]);
// send the content
String timestamp= stub.acceptContent(rawbytes);
```

What is different from the previous examples is the configuration file shown in Listing 10.16 for `xrpcc`, where the handlers are declaratively specified.

Listing 10.16 `xrpcc` configuration for handlers

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service name="Contentservice" targetNamespace="http://www.flutebank.com/xml"
            typeNamespace=http://www.flutebank.com/xml
```

```
            packageName="com.flutebank.encryptedposervice">
<interface name="com.flutebank.encryptedposervice.Fileservice"
  servantName="com.flutebank.encryptedposervice.FileserviceImpl"/>
<handlerChains>
  <chain runAt="client">
        <handler className="com.flutebank.encryptedposervice.SecureZipClientHandler">
      </handler>
    </chain>
  <chain runAt="server">
        <handler className="com.flutebank.encryptedposervice.SecureZipServerHandler">
        </handler>
  </chain>
</handlerChains>
  </service>
</configuration>
```

Handler Disadvantages

Though handlers offer a nice way of pre- and postprocessing the SOAP message, certain issues must be kept in mind:

- If the handler code introduces propietary modifications in the outgoing SOAP message, the service may no longer be interoperable with other platforms. For example, just based on WSDL, the caller of the above service will never be able to deduce that the server endpoint expects the data in a particular compressed and encrypted format.
- Introducing handlers that alter the response message at an endpoint may break existing clients written for the service interface.
- Introducing another layer of pre- and postprocessing of the SOAP message may degrade performance by increasing reponse times.

Asynchronous Invocation with Attachments and Handlers

Earlier in this chapter, we mentioned the use of attachements for creating compound messages. Let us look at a possible realization, shown in Figure 10.10.

A business document, such as a purchase order or invoice in XML format, is sent as an attachment to the SOAP message in the request.The service definition exposes a method similar to

```
public String submitInvoice(Source invoice) throws RemoteException;
```

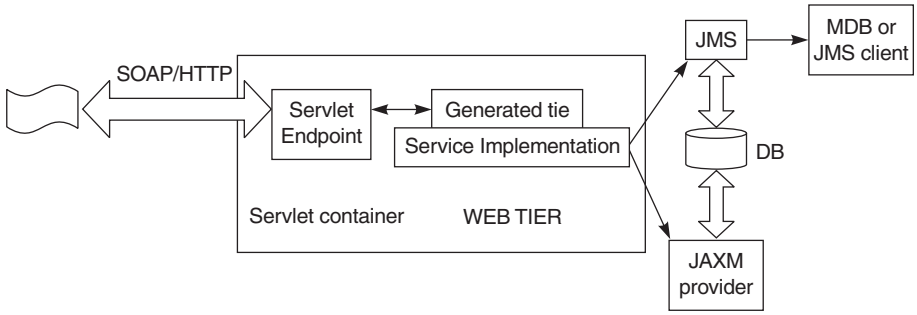

Figure 10.10 Using XML attachments with JAX-RPC for asynchronous invocation

which receives the message and in turn maps the attachment as per the mapping in Table 10.7. An XML attachment (`text/xml` MIME type) automatically maps to the `javax.xml.transform.Source`. The service could then process that XML representation in many ways. For example, it could parsed and transformed it, place it on a JMS queue, or even pass it to a JAXM provider, as Figure 10.10 shows. Additionally, handlers could intercept the message and perform a hard validation against a schema for the document, if necessary.

The difference between this asynchronous model and the one-way RPC is essentially that the client receives a response from the endpoint, because of its asynchronous persistence framework using JMS. If, as a result of HTTP issues, the client could not receive that response, it is the client's prerogative to retry the invocation or query the service. The latter would require introspection into the integration tier (e.g., JMS queue). The service has no way to communicate back to the HTTP client with a callback unless the client realizes a similar service on its side. This realization, though useful in most scenarios, is *pseudo-asynchronous*. Look at asynchronous messaging with JAXM and messaging profiles in Chapter 11.

Holder Classes

CORBA developers would already be familiar with the concept of `in`, `out`, and `inout` parameters and `Holder` classes. Like IDL, operations in WSDL may take `out` or `inout` parameters as well as `in` parameters. To understand the `in`, `out`, and `inout` concepts, consider the following signature:

```
public Something myMethod(Somearg somearg) {
// code
}
```

In Java, the somearg is the argument the method receives, and the Something is what the method returns after doing its work. However if clients pass the somearg as an object and expect the method to change that value, it is an inout parameter. For example, consider the following code:

```
Somearg param= Somearg(...).;
Something val= myMethod(param);
if (param.xxx){
       //
}
```

The above coding practice is discouraged in Java but is used in other languages, such as C and C++. Java passes parameters only by value and has no concept of out or inout parameters; therefore, in JAX-RPC these are mapped Holder classes. In place of the out parameter, a Java method will take an instance of the Holder class of the corresponding type. The result assigned to the out or inout parameter is assigned to the value field of the Holder class. ▷

A service operation signature written in Java will typically return a single value: a primitive or a JavaBean. If there is a need for the service operation to return multiple values, the data type of the return value can be a complex type, such as an object with multiple parts (e.g. a Portfolio object with many Position objects) or an array. The third alternative is to specify that one or more of the parameters of the Web service operation be out or inout parameters.

For example, assume a Web service operation contains one out parameter, and the operation is implemented with a Java method. The method sets the value of the out parameter and sends this value back the client application that invoked it. The client application can then access the value of this out parameter as if it were a return value. The code below illustrates this with a method whose second parameter is an inout parameter:

▷ In WSDL, if a part name appears

- In both the input and output message, it is an inout parameter
- In only the input message, it is an in parameter
- In only the output message, it is an out parameter

out parameters are undefined when the operation is invoked but defined when the operation completes; inout parameters are defined when invoked and when completed.

```
public float payBalance(String userid,javax.xml.rpc.holders.IntHolder balance) {
           System.out.println ("The input value is: " + balance.value);
           // do some work here
           balance.value = 90; // the new value of the out parameter
       }
```

When the client invokes the above method with two parameters, a `String` and an integer, it will be returned two values: a float and an integer. If at invocation the balance parameter value was 1000 when the method completed, the value of the second parameter is now 90 and will also be returned to the client.

```
IntHolder inoutbalance = new IntHolder(1000);
System.out.println("Holder value is " + inoutbalance.value);
float interest= service.payBalance("johnmalkovich",inoutbalance);
System.out.println("Interest charged on credit card is " + interest);
System.out.println("Remaining balance,holder value is " + inoutbalance.value);
```

The above client code invoking the above service implementation will produce the following output:

```
Holder value is 1000
Interest charged on credit card is 9.0
Remaining balance, holder value is 90
```

Holder classes for `out` and `inout` parameters must implement the `javax.xml.rpc.holders.Holder` interface. In the service implementation, use the `value` field to first access the input value of an `inout` parameter and then set the value of `out` and `inout` parameters.

If the `out` or `inout` parameter is a standard data type, JAX-RPC provides a set of holder classes in the `javax.xml.rpc.holders` package, listed in Table 10.9.

If the data type of the parameter is not provided, developers must create their own implementation of the `javax.xml.rpc.holders.Holder` interface to handle `out` and `inout` parameters, based on the following guidelines:

- Name the implementation class `XXXHolder`, where `XXX` is the name of the complex type. For example, if the complex type is called `Portfolio`, the implementation class is called `PortfolioHolder`.
- Create a `public` field called `value`, whose data type is the same as that of the parameter.

Table 10.9 JAX-RPC–Defined Holder Classes

Built-in holder class	Java data type it holds
javax.xml.rpc.holders.BooleanHolder	boolean
javax.xml.rpc.holders.ByteHolder	Byte
javax.xml.rpc.holders.ShortHolder	short
javax.xml.rpc.holders.IntHolder	Int
javax.xml.rpc.holders.LongHolder	Long
javax.xml.rpc.holders.FloatHolder	float
javax.xml.rpc.holders.DoubleHolder	double
javax.xml.rpc.holders.BigDecimalHolder	Java.math.BigDecimal
javax.xml.rpc.holders.BigIntegerHolder	Java.math.BigInteger
javax.xml.rpc.holders.ByteArrayHolder	Byte[]
javax.xml.rpc.holders.CalendarHolder	Java.util.Calendar
javax.xml.rpc.holders.QnameHolder	javax.xml.namespace.QName
javax.xml.rpc.holders.StringHolder	Java.lang.String

- Create a default constructor that initializes the `value` field to a default.
- Create a constructor that sets the `value` field to the passed parameter.

The following example shows the outline of a custom `PortfolioHolder` implementation class:

```
package com.flutebank.brokerage;
public final class PortfolioHolder implements javax.xml.rpc.holders.Holder {
       public Portfolio value;
       public PortfolioHolder() {
       }
           // set the value variable to a default value
       }
       public PortfolioHolder(Portfolio value) {
           // set the value variable to the passed in value
       }
   }
```

Using Custom Data Types

Besides the data types supported by JAX-RPC discussed earlier, it may be necessary to pass data types that do not satisfy the requirements. For example, `BillPay.java` demonstrated earlier could define the `listScheduledPayments()` to return a `java.util.Vector` of `PaymentDetail` objects, instead of the `PaymentDetail[]` it did return. Note that a `Vector` is not a supported data type, as per the mappings in Table 10.2.

JAX-RPC supports the concept of pluggable serializers and deserializers for such custom data types. A serializer marshals a Java object to an XML representation, and a deserializer unmarshals an XML representation to a Java object. As Figure 10.11 shows, serialization and deserialization are symmetrical functions and both use *type mapping* to map the Java and XML data types.

Developers can specify the serializer and deserializer to use for a service on the server using the deployment tool. `xrpcc` has the `typemapping` element for this purpose. This allows the endpoint to unmarshal the XML to the corresponding Java type, and vice versa. For example, a `com.fluebank.Vector` may be serialized as

```
<avector xmlns:tns="http://www.flutebank.com" xsi:type=" tns:Vector">
<item xsi:type="xsd:string">some value here</item>
<item xsi:type="xsd:anyType" xsi:null="true"/>
</avector >
```

If the server know that this namespace and type correspond to a `com.flutebank.Vector`, it can invoke the corresponding deserializer and create and pass the

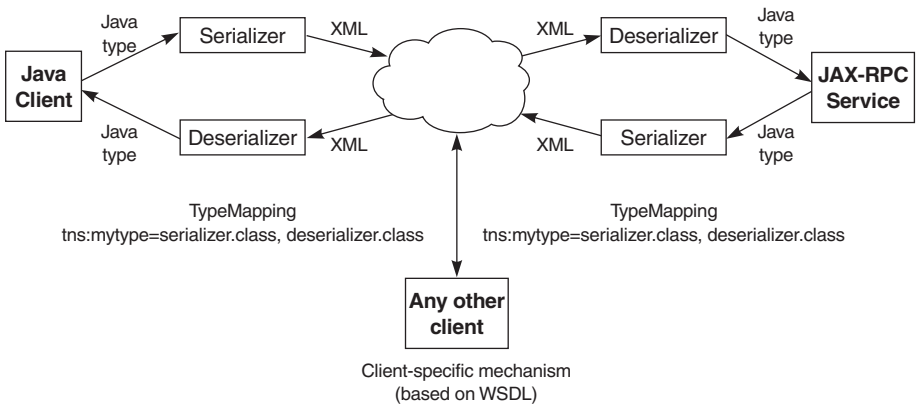

Figure 10.11 Serializers and deserializers

corresponding `com.flutebank.Vector` object to the service implementation. When the service client is written, the developers will need to write a similar serializer and deserializer on the client-side runtime or take a shortcut and use the same classes from the server, if the same vendor runtime is used. If it is not used, the runtime will not know what to do when it comes across this custom data type and will throw a serialization exception.

JAX-RPC Pluggability Mechanism

The JAX-RPC part of the API, the type system relevant to development of pluggable serializers and deserializers, is simple and is shown in Figure 10.12 and Table 10.10.

The base serializer and deserializer interfaces are implemented by a runtime-specific class or extended by a runtime-specific interface. Developers use this to write their serializers and deserializers for that particular runtime. However, a larger issue is at hand. A closer look at the `Serializer`, `DeSerializer`, `SerializationContext`, and `DeSerializationContext` interfaces reveals no methods relate to serialization or deserialization and that these are just marker interfaces. What

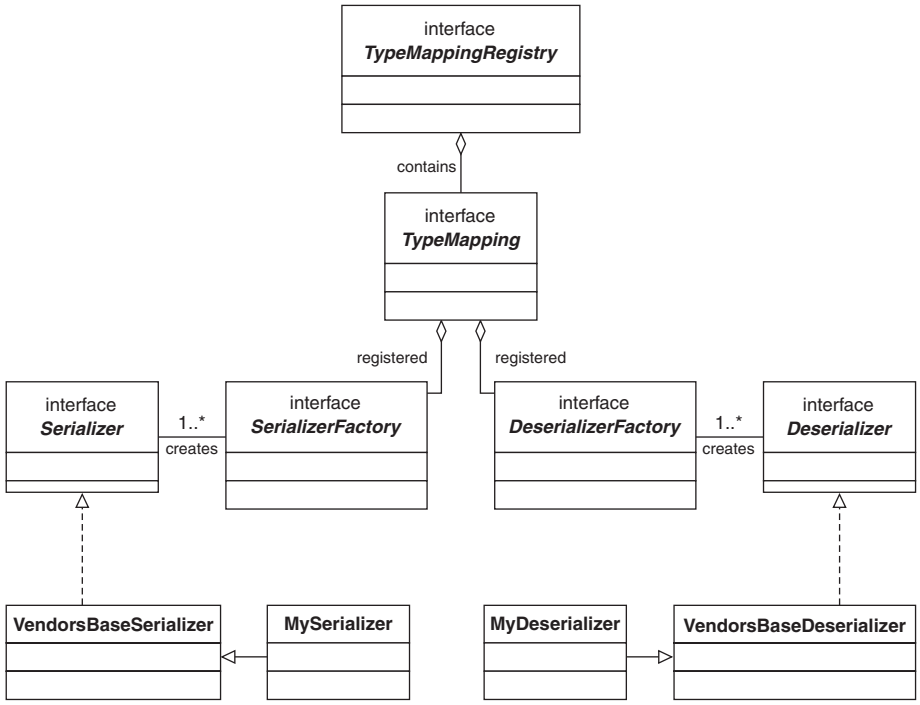

Figure 10.12 The type mapping system

Table 10.10 The Type Mapping System API

`TypeMappingRegistry`	Defines an internal registry that holds a mapping of encoding styles and the corresponding `TypeMapping`.
`TypeMapping`	Maintains a set of tuples of the type {Java type, `SerializerFactory`, `DeserializerFactory`, XML type}.
`Serializer`	The base interface for serializers to implement.
`DeSerializer`	The base interface for deserializers to implement.
`SerializationContext`	Passed to the serializer as context information.
`DeSerializationContext`	Passed to the deserializer as context information.

this means is that serializers and deserializers are not guaranteed to be *portable* across implementations, because there is no contract with the runtime. They are specific to and pluggable only in a particular implementation. For example, if a developer writes a serializer and deserializer for the JAX-RPC RI, these classes are not guaranteed to be usable in another vendor's JAX-RPC implementation.

The API is structured like this for a very good reason. Different runtimes may (and do) use different XML parsing techniques (e.g., DOM parser, SAX parser, streaming pull). Porting a serializer written for SAX parsing (i.e., one that expects a SAX stream) into a runtime that uses a different parsing mechanism cannot be done completely transparently. The next version of the JAX-RPC specification is supposed to address transparent pluggability further.

Most vendors will provide several built-in serializers and deserializers, to help developers as utility classes for their runtimes. The code will never be aware of the need for a serializer/deserializer for that particular custom data type, as long the code is deployed in that vendor's runtime. (If you move it to another, you may need to write the serializer and deserializer yourself.) For example, JAX-RPC 1.0 RI supports a subset of Java collection classes and provides corresponding serializers and deserializers as utilities for developers (Table 10.11).

So if the `listScheduledPayments()` method returned a `java.util.ArrayList`, even though it is not a data type for which a standard Java-XML mapping exists, the runtime will generate the corresponding SOAP message and response, based on internal type mapping and custom serializers and deserializers.

Table 10.11 Deserializers and Serializers Provided as Utilities by the Reference Implementation

java.util.Collection
java.util.List
java.util.Set
java.util.Vector
java.util.Stack
java.util.LinkedList
java.util.ArrayList
java.util.HashSet
Java.util.TreeSet
Java.util.Map
Java.util.HashMap
Java.util.TreeMap
Java.util.Hashtable
java.util.Properties

Configuring Custom Serializers and Deserializers

Like message handlers, pluggable serializers and deserializers can be configured in two ways: programmatically, using JAX-RPC API, or declaratively, using a JAX-RPC runtime-provided tool or deployment descriptor. Client-side serializers and deserializers can be configured in either way, but server-side handlers can be be configured only declaratively. The fragment below shows the relevant extract for `xrpcc` in the reference implementation given in Listing 10.3.

```
<typeMappingRegistry>
    <import>
        <schema namespace="" location=""/>
    </import>
    <typeMapping encodingStyle="">
    <entry schemaType="" javaType="" serializerFactory="" deserializerFactory=""/>
    </typeMapping>
    <additionalTypes>
```

```
        <class name=""/>
    </additionalTypes>
</typeMappingRegistry>
```

Programmatic registration on the client can be done in code similar to the following:

```
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService (...);
TypeMappingRegistry registry = service. getTypeMappingRegistry();

TypeMappingRegistry maprping = registry.createTypeMapping()
// or registry.getDefaultTypeMapping();

    SerializerFactory sfactory= // some runtime specific code

    DeserializerFactory dfactory= // some runtime specific code

// register the custom handlers passing the Java class, the namespace, the serializer
// factory to use and the deserializer factory to use

    mapping.register(myclass, qname, sfactory, dfactory)
    registry.register(encodingStyleURI,mapping)
```

JAX-RPC and Security

Security has multiple aspects; Chapter 15 covers them in detail. From a JAX-RPC perspective, there are two major points:

- **Securing the transport layer.** JAX-RPC does not explicitly require runtimes to support Hypertext Transfer Protocol Secure (HTTPS). However most servlet containers where the HTTP endpoints are deployed potentially support HTTPS (e.g., Tomcat). Just switching on HTTPS on the server will be enough for the service deployment. To enable SSL support on the client side, however, JSSE must be used to change the default HTTP handlers.
- **Securing users.** JAX-RPC requires support for basic HTTP authentication. The username and password on the client side can be passed via the `javax.xml.rpc.security.auth.username` and `javax.xml.rpc.security.auth.username.password` properties discussed in the Clients Using Stubs section earlier

in this chapter. If at runtime the username or password is not found or is incorrect, the server code will send the client an HTTP code 401 along with the basic HTTP authentication header (`WWW-Authenticate`). In the service implementation, the service can access the `java.security.Principal` via the `getUserPrincipal()` in the `ServletEndpointContext`, as shown earlier.

> JAX-RPC Interoperability

In the context of Web services, interoperability can be summarized as meaning that the functional characteristics of the service should remain immutable across differing application platforms, programming languages, hardware, operating systems, and application data models. By definition, Web services should be interoperable, and the service consumer should not be tied to the service implementation.

However there are bound to be issues when applications use disparate SOAP libraries that generate and manipulate the underlying SOAP message, disparate programming languages, and disparate hardware-software stacks. The following are common causes for interoperability problems between these libraries or toolkits:

- Implementations conform only to a subset of the full SOAP or other XML specifications.
- Implementations depend on optional aspects of the SOAP specifications. For example:
  - Sending type information for encoded parameters is optional; however, if an implementation assumes this will be present in messages it receives, it may not interoperate with others that do not send this information.
  - There is no differentiation between `SOAPAction` values of `""` and `null` in the specifications, but some implementations support both, whereas others do not quote this value at all for non-null `SOAPActions`.
- Implementations interpret ambiguous definitions of the SOAP specification differently. For example:
  - It is not clear how a service should represent an RPC response with a void return and no `out` parameters. It could be an empty SOAP envelope, an empty SOAP response element, or even an HTTP 204 (“No Response”) code.

- A null value can be represented either by not including that XML element or by an element with the `xsi:nil="true"`. ▷

In general, architects should keep the following in mind while designing Web services:

1. **Avoid propeietary extensions.** Avoid building dependencies into the application that use any vendor-specific extension to the specifications JAX-RPC depends on (SOAP, WSDL, XML schemas, and HTTP).
2. **Test interoperability.** Never assume things will work as they should. It is essential to test interoperability of the service implementation across multiple consumers, especially if the consumers are outside the boundaries of the organization. Public interoperability tests are also available from the Web Services Interoperability organization *(www.ws-i.org)* and White Mesa *(www.whitemesa.com).* ▷▷
3. **Analyze disparate data models.** When a service is used to integrate applications that have disparate data models, the models may need to be resolved by creating an intermediate model. For example, flutebank.com integrates with brokerage.com to provide customers the ability to view their accounts simultaneously online when in any of the portals. The data model for an account as represented in flutebank.com may be quite different from an account in

---

▷ The only real mechanism for ensuring interoperability is to verify compliance with standards:

- HTTP 1.1 for the transport protocol
- XML Schema to describe your data
- WSDL 1.1 to describe your Web service
- SOAP 1.1 for the message format

The JAX-RPC API and SAAJ provide a standard interface for Java developers leveraging these same standards.

▷▷ Testing! To promote SOAP-level interoperability and address issues between implementations, the SOAPBuilders community—with members as diverse as IBM, Microsoft, Sun Microsystems, Apache, and even individuals—has come together to develop an interoperability test suite specification and regularly conduct testing of their endpoints against this specification. See *http://soapinterop.java.sun.com/soapbuilders/index.shtml* and *www.xmethods.net/ilab.*

brokerage.com. In this scenario, the architects will need to reconcile the models by creating an XML schema acceptable to both parties.

4. **Analyze disparate data types.** Data types passed as arguments and return types from the service invocation can impact interoperability.
   - **JAX-RPC–defined data types.** The data types and mappings defined in the specifications are available in all JAX-RPC runtimes. Because they are subsets of the XML schema specifications and map directly to the data types in the SOAP encoding, they are completely interoperable.
   - **Custom data types.** If the data type is custom defined (e.g., a `java.util.HashMap` of `com.flutebank.accounts.Account` objects), an XML schema must be created to describe the representation of the data and the custom serializer and deserializer for that data on the server. Such a schema may not be completely interoperable. In addition, the JAX-RPC client would need to write serializers and deserializers to invoke the service. We looked at handling custom data types earlier in this chapter. In summary, custom data types may not be completely interoperable across all service consumers.
5. **Avoid custom data types.** Custom data types that force the use of custom serializers and deserializers can potentially cause interoperability issues with other implementations. For example, a `List` may be represented differently by implementations from vendors A and B. If vendor A's client runtime is used to invoke a service deployed in vendor B's runtime, serialization errors may occur, because each implementation uses its own XML mapping of that data type. If the mapping is not available, the corresponding serailizers and deserializers will need to be written.

   For collection classes in particular, the SOAPBuilders community plans to pursue interoperability testing across vendor runtime implementations.
6. **Customize data, protocols, and encoding schemes.** Architects should be wary of any code that customizes the messages. A good example is the Compress-Secure handlers example in Listing 10.13. The endpoint of that service cannot be invoked by clients that are not aware of the compression and security algorithm used and understood by the service. From a service perspective, there is no standard way to communicate this information (e.g., it cannot be specified in WSDL).
7. **Promote portability of client code between JAX-RPC implementations.** J2EE developers would be familiar with the concept of writing an EJB and deploying it transparently in a J2EE server. The EJB *client* can be written with complete transparency and used in any J2SE environment by simply altering configuration properties. This portability of client code does not translate

identically in the JAX-RPC environment, especially when using custom data types. A JAX-RPC client is not guaranteed to be portable if it uses anything beyond the simple data types. This is tied to the way the serializers and deserializers are written, as discussed earlier. In other words, if architects choose vendor A's implementation of JAX-RPC and write client code that uses serializers and deserializers to invoke the service, they should not expect to simply take the *client* code and use it in vendor B's runtime. Applications should be designed to abstract away the specificity, minimizing the changes needed.

Let us now look at an example of interoperability in action. We will write a Microsoft C#.NET client to demonstrate how a JAX-RPC Web service can be consumed from a Microsoft.NET environment. We will use the `BillPay` service developed and deployed previously in this chapter. As in most practical service consumer scenarios, we will invoke a service based on the WSDL describing it.

During service deployment, `xrpcc` was used to generate the WSDL. This WSDL file can now be passed to the Microsoft.NET `wsdl` compiler, to generate the client-side stubs:

```
wsdl /l:CS /protocol:SOAP http://127.0.0.1:9090/billpayservice/billpayservice.wsdl
```

The above generates the C# source file `Billpayservice.cs`, which contains the structures and serialization rules based on the schema and bindings defined in WSDL.

Next, we build a Windows DLL out of the generated proxy code using the C# compiler, passing it the referenced dlls from the .NET framework:

```
csc /t:library /r:System.Web.Services.dll /r:System.Xml.dll Billpayservice.cs
```

The next step is to write a client and invoke the three methods exposed by the JAX-RPC Web service. Listing 10.17 shows the C# client code for this purpose.

Listing 10.17 C# client for JAX-RPC

```
BillPay service
using System;
namespace BillpayClient{
     /// <summary>
     /// This is a simple C# client to invoke the flutebank.com Web service.
```

```
     /// @Author Sameer Tyagi
     /// </summary>
     class JAX-RPCClient{
           /// <summary>
           /// The main entry point for the application.
           /// </summary>
           [STAThread]
           static void Main(string[] args) {

     // Instantiate the stub/proxy
                Billpayservice serv = new Billpayservice();
     // Set the endpoint URL
           if(args.Length ==1)
                 serv.Url=args[0];
           else
                 serv.Url= "http://127.0.0.1:9090/billpayservice/jaxrpc/BillPay";

// Invoke the schedule payment method
           PaymentConfirmation conf = serv.schedulePayment(DateTime.Today,"my
                                                        account at sprint",190);
           Console.WriteLine("Payment was scheduled "+ conf.confirmationNum);

// Invoke the listSchedulePayment method
           PaymentDetail[] detail= serv.listScheduledPayments();
                 for(int i=0;i< detail.Length;i++){
                       Console.WriteLine("Payee name "+ detail[i].payeeName);
                       Console.WriteLine("Payee name "+ detail[i].account);
                       Console.WriteLine("Payee name "+ detail[i].amt);
                       Console.WriteLine("Payee name "+ detail[i].date);
                 }
// Invoke the getLastPayment method
          Double lastpaid= serv.getLastPayment("my cable tv provider");
          Console.WriteLine("Last payment was "+ lastpaid);
          }
     }
}
```

The C# client code is remarkably similar to the JAX-RPC stub client written earlier, because of the syntactic and semantic similarities between the two

programming languages. The client code can now be compiled and executed. The output will be similar to the following:

```
C:\Dotnetclient>billpayClient
Payment was scheduled 81263767
Payee name Digital Credit Union
Payee name Credit
Payee name 2000
Payee name 8/1/2002 11:27:33 PM
Last payment was 829 ▷
```

We have just developed a Web service in Java, deployed it in a JAX-RPC runtime, and exposed the service with only a WSDL interface. WSDL was used to develop a client in a completely different language and platform, C# and .NET, yet produced identical behavior.

> JAX-RPC and J2EE

Three specifications tightly integrate JAX-RPC with J2EE:

- J2EE 1.4 specifications (JSR-151)
- EJB 2.1 specifications (JSR-153)
- Web services for J2EE (JSR-109)

J2EE 1.4 includes JAX-RPC as a required API, which means that all J2EE 1.4 application servers will support JAX-RPC. The EJB 2.1 specifications–part of J2EE 1.4–also define how an EJB can be exposed as a Web service and how EJBs can consume a Web service. The *Implementing Enterprise Web Services* specification will lay out the deployment and service requirements for portability of client and server code across containers.

▷ The complete C# project and Microsoft .NET runtime distributable can be found on the CD. Java developers can think of the runtime distributable as the JRE. It allows developers to execute the compiled code. To build the source, however, Microsoft .NET Visual Studio is needed.

JAX-RPC and JSR 153

EJB 2.1 allows a *stateless* session bean to be exposed as a Web service, by defining a new interface type in addition to the home, local, and remote interfaces. It is called an *endpoint interface* and is essentially the JAX-RPC service definition. EJB developers provide the service definition and the EJB class. As Figure 10.13 shows, the container generates the implementation of the endpoint interface. much as it generates the implementation of the `EJBObject` during deployment.

The container exposes the EJB through its service endpoint interface and a WSDL document that clients can use. Once it is deployed, clients use it like any other JAX-RPC service–that is, they access this stateless session bean using the JAX-RPC client APIs over an HTTP transport, just like clients covered earlier in the chapter.

EJBs can look up other Web services with Java Naming and Directory Interface (JNDI), using a logical name called a *service reference*. It maps to a `service-ref` element in the deployment descriptor, obtains a stub instance for a Web service endpoint, and invokes a method on that endpoint. The J2EE client or EJB can do this, as Figure 10.14 and Listing 10.18 show.

Listing 10.18 EJB client and deployment descriptor code extract

```
InitialContext ctx = new InitialContext();
BillPayService service = (BillPayService)ctx.lookup
                                    ("java:comp/env/service/billpayservice");
BillPay stub=(BillPay)(serviceproxy.getBillPayPort());
PaymentConfirmation conf= stub.schedulePayment(new Date(),
                                                    "my account at sprint",190);

<enterprise-beans>
<session>
        <service-endpoint> com.flutebank.billpayservice.BillPay</service-endpoint>
        <ejb-class> com.flutebank.billpayservice.BillPayEJB </ejb-class>
      <service-ref>
           <service-ref-name> service/billpayservice</service-ref-name>
           <service-ref-type>com.flutebank.BillPayImpl</service-ref-type>
        </service-ref>
</session>
</enterprise-beans>
```

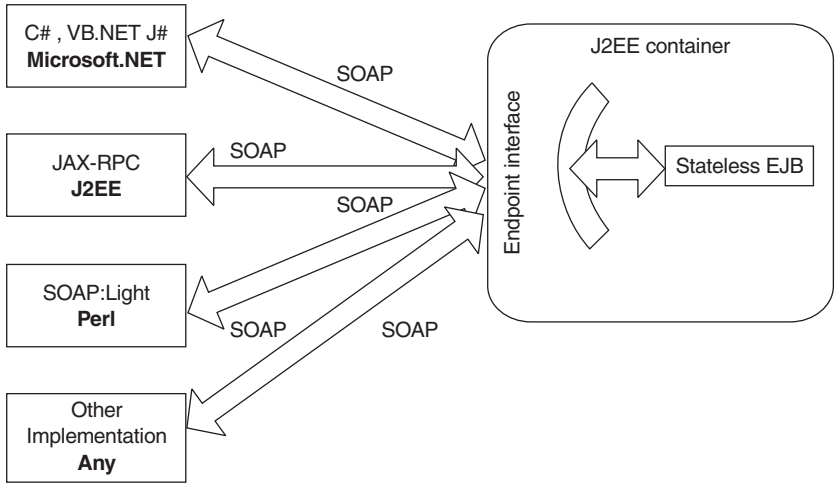

Figure 10.13 EJB endpoint for JAX-RPC

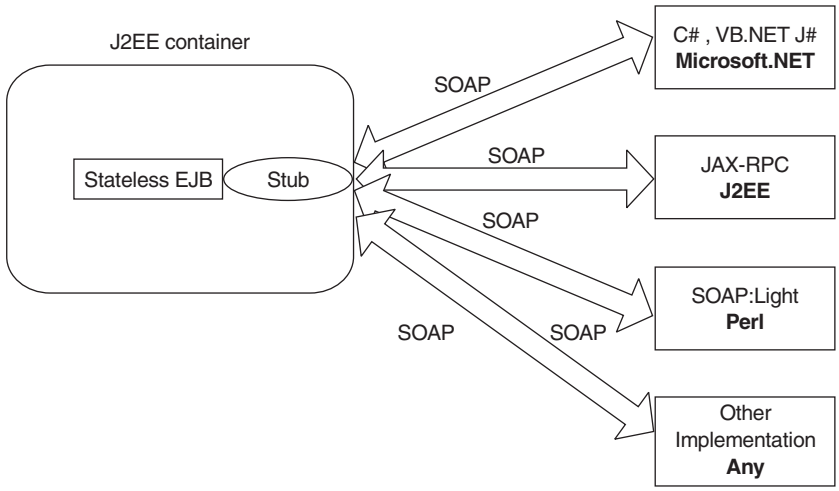

Figure 10.14 EJB invoking other Web services

What Implementation Is Right for Me?

Implementing a JAX-RPC service as an EJB in a J2EE container has four significant advantages:

- Integrated support for transactions
- A comprehensive security model
- Integration with existing business logic
- Scalability through the application server (e.g., clustering and failover)

If the service was implemented as a class and *not* deployed in a J2EE container, it forgoes the advantage of the ACID (atomic, consistent, isolated, and durable) characteristics of the Java Transaction API (JTA) transaction. A non-EJB class in a J2EE container could still leverage a JTA transaction by directly using the `javax.transaction.UserTransaction` object though a JNDI lookup. The stateless EJB with its service endpoint interface, like other EJBs, can propagate and demarcate transactions in a J2EE container and also use bean-managed transactions. It can also leverage the role-based security features the J2EE container provides. (Note, however, that transaction context *propagation* is not required by the current JAX-RPC specifications.)

Using just a servlet endpoint and a Java class(s) (without an EJB) implementation has the following advantages:

- Generally better performance
- Simplicity in deployment
- No need for a full-blown J2EE application server; any Servlet 2.3–compliant Web server or container can be used

JAX-RPC and JSR-109

Implementing Enterprise Web Services (JSR-109) defines a complete mechanism for deploying Web services in a container, using a `webservices.xml` file for a module and a `webservicesclient.xml` file for the clients. The key elements of the former are shown below.

```
<webservices>
   <description>A sample file </description>
   <webservice-description>
      <wsdl-file>billpayservice.wsdl</wsdl-file>
      <port-component>
         <port-component-name>BillPayerComponent</port-component-name>
         <port-qname-namespace>http://www.flutebank.com/xml</port-qname-namespace>
         <port-qname-localname>BillPayService</port-qname-localname>
         <service-def-interface>com.flutebank.billpayservice.BillPayt
                              </service-def-interface>
         <service-impl-bean>
      <!--If the service implementation is an EJB →
            <ejb-link >com.flutebank.billpayservice.BillPayEJB </ejb-link>
```

```
      <!--If the service implementation is a Servlet →
            <servlet-link>com.sun.xml.rpc.server.http.JAXRPCServlet</servlet-link>
         </service-impl-bean>
      </port-component>
   </webservice-description>
</webservices>
```

At the time of writing, all three of these specifications were still in draft form. They may possibly undergo changes as a part of the Java community process.

> Summary

So, why do you need JAX-RPC? And more important, what value do these APIs add to architects? Should developers stop writing RMI, RMI-IIOP, and Java-IDL/CORBA applications and discard code that consumed significant time and money, just because newer technology is available?

The answer is an obvious no. Those APIs are as integral a part of J2EE specifications as JAX-RPC is in J2EE 1.4.

JAX-RPC adds value only if you are sure you want to use SOAP, because it allows developers to write distributed and loosely coupled applications using this technology: distributed because the objects may not be colocated, and loosely coupled because the model inherently enforces a level of independence between the implementation and the calling code. As with all software, architects have to understand the tradeoffs in integration with existing and new applications, performance, bandwidth, and accessibility. However, there are possibly two major use cases where JAX-RPC API can be exploited.

The RMI Analogy

Just as in RMI developers write a remote interface, the implementation, and use the `rmic` compiler to generate the stubs and ties, you could write a service definition, the service implementation, and use a JAX-RPC tool (e.g., `xrpcc`) to generate the relevant JAX-RPC stubs, ties, and the WSDL file. The client could then use the WDSL file and stubs to invoke the service. This is something architects can do to expose existing business logic contained in RMI objects or EJBs as Web services.

The CORBA Analogy

J2SE has the `idlj` compiler that reads an IDL file and generates the Java bindings. Developers can do something similar with the JAX-RPC tools (e.g., `xrpcc`) and consume a WSDL file to generate the client, server, or both sides of the code, to serve as relevant adaptors for the servant code they write. This is something architects can do to implement a service that conforms to a given interface or consume an existing Web service on the JAX-RPC or on a completely different platform, such as Microsoft.NET.

This chapter was meant to give you an insight into JAX-RPC, an API that provides an invaluable *standard* for developers and architects who want to build XML-RPC based Web services.