# 3.1 Introduction

The most significant aspect of Web services is that every software and hardware company in the world has positioned itself around these technologies for interoperability. No single technological advancement will have as great an impact on the way systems are developed as Web services.

Web services allow systems to communicate with each other using standard Internet technologies. Systems that have to communicate with other systems use communication protocols and the data formats that both systems understand. Developers use technologies such as CORBA, RMI, or DCOM most often. The principal problem with these communication technologies is that not every platform supports them. Developers must create gateways to convert an unsupported protocol and data format into one that the target platform understands.

The emergence of the Internet has forced vendors to support standards such as HTTP and XML. Over the past few years, vendors and their customers quickly realized that programs that communicate with each other could also use the technologies that run the Internet. Web services use Internet technology for system interoperability. The advantage that Web services have over previous interoperability attempts, such as CORBA, is that they build on the existing infrastructure of the Internet and are supported by virtually every technology vendor in existence. As a result of the ubiquitousness of the technologies they use, Web services are platform-independent. This means that whether the Web service is built using .NET or J2EE, the client uses the service in the exact same way.

# 3.2 Web Services Technology

The Web services stack shown in Figure 3.1, categorizes the technology of Web services into a layered model. The stack starts at the bottom with the basic technologies that allow data to transfer from one machine to another. Each layer builds on the lower layers and adds higher-level abstractions. The upper layers of the stack do not necessarily depend on the lower layers and in some ways are orthogonal concerns. They are shown in this format simply to demonstrate a higher level of abstraction.

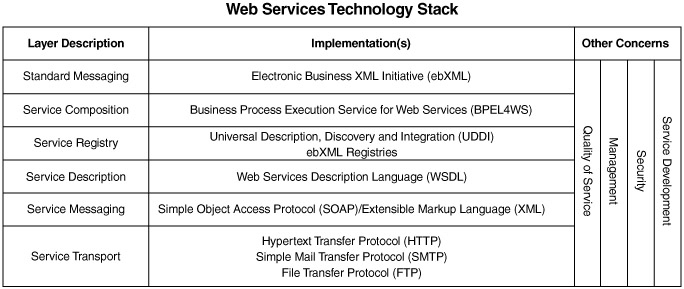


Fig 3.1 Web Service Technology Stack

### 3.2.1 Service Transport

The main function of the service transport layer is to transfer data from one machine to another. This is the protocol for the transmission of data for Web services. Web services use multiple transports to transfer data from service to service—including, but not limited to, HTTP, SMTP, and FTP.

The most popular protocol by far for use in Web services is HTTP. The Internet and the World Wide Web use HTTP to transmit data. HTTP is not blocked by most firewalls and thus is the standard for interoperable systems. HTTP 1.0 is a connectionless and stateless protocol. This means that each request and response a client sends and receives from a server is independent and not in the overall context of a conversation.

### 3.2.2 Service Messaging

The messaging layer of the technology stack describes the data formats used to transmit data from one service to another over the transport. XML is the base format used for Web services. XML is a text-based protocol whose data is represented as characters in a character set. The XML data is structured as a tree with elements, and the entire tree structure is called a document. XML has no data description separate from the data itself, unlike fixed or delimited data formats. The messages are self-describing. The data has specially formatted tags around it that give the data a name as well as a position in the document's tree structure.

Simple Object Access Protocol (SOAP) is a specification that tells a service consumer and a service provider how to format and read a specially formatted XML message for use in a service. A SOAP message has three sections: the envelope, the header, and the body.

The envelope is the top element of the XML message. It indicates that the message is a SOAP message, and it has instructions for processing the message.

The SOAP header contains application context information and directives. The header information is usually read and processed by the server, not the service. For instance, if an application needs entries for authentication or transaction management, the header will contain data for these features. SOAP messages can be passed along through multiple intermediaries between the service consumer and the service provider. Each intermediary reads the header information and uses it for routing, logging, and other system functions.

The SOAP body contains the application data. Web services support multiple message exchange patterns. A SOAP message may be sent in a document-oriented format or an RPC style format. In a document-oriented message exchange, the service consumer and provider exchange XML documents. In the RPC-style message exchange, data is passed as arguments. SOAP messages are by nature one-way transmissions from a sender to a receiver but are usually combined to implement a request/response model.

### 

### 3.2.3 Service Description

The service description specifies three aspects of the service:

* Operations the service has made available
* Messages the service will accept
* The protocol to which the consumer must bind to access the service

Web services uses the Web Services Description Language (WSDL) to specify a service contract. The service contract is a description of a set of endpoints that operate on messages and the specification for how an XML document should be formatted when it is sent to the endpoints. An endpoint is a network address that accepts a message formatted to the specification defined in the WSDL. WSDL uses the term *port* to describe a service endpoint for a message. WSDL describes the contract for a service as a collection of ports the service has made available.

### 3.2.4 Service Registry

Web services support the concept of dynamic discovery of services. A consumer of a service uses a service registry to find the services it is interested in using. Universal Description, Discovery and Integration (UDDI) is a Web service itself that supports a standard set of services that allow a Web service consumer to dynamically discover and locate the description for a Web service. UDDI registries are themselves Web services that expose an API as a set of well-defined SOAP messages. The Web service provider and Web service consumer use SOAP and HTTP to publish and retrieve information about services in the registry. Public UDDI registries contain a Web services contact, business owner, and technical information about the Web service. UDDI supports two types of conversations:

* The service provider uses the UDDI directory to publish information about the Web services it supports.
* The Web service consumer sends SOAP-formatted XML messages over HTTP to the UDDI directory, to retrieve a listing of Web services that match its criteria.
* Java and Web Services
* Web services are platform-independent. This means that a Web service may be developed in a large number of languages to run on many platforms. The Java language and Java 2 Enterprise Edition (J2EE) platform provide features for building and deploying Web services. The benefits of the Java language and the J2EE platform are vendor independence and application portability. Applications built on the J2EE platform may be deployed on J2EE implementations from a large number of vendors. Using Java to develop Web services gives you the benefit of vendor independence in addition to the inherent platform independence of Web services.
* Java has support for Web services through the Java Web Services Developer Pack (JWSDP). JWSDP contains libraries for generating XML and SOAP, processing XML, accessing service registries, and calling RPC-based Web services. JWSDP and the libraries it contains constitute the bulk of this book, so we won't go into detail here.

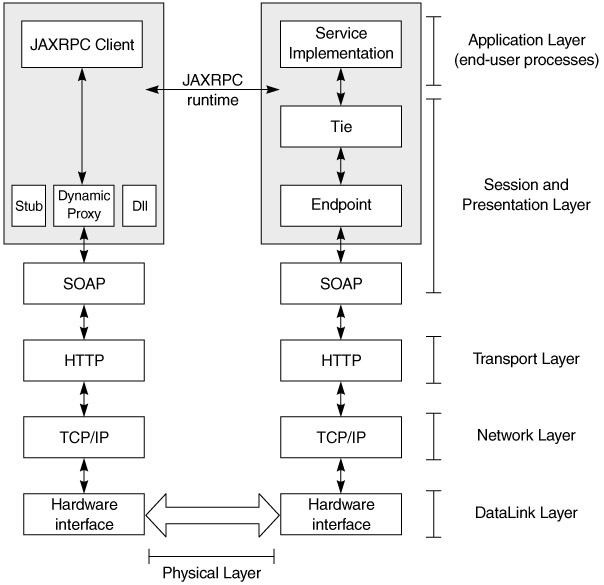
## 3.3 JAX-RPC Service Model

JAX-RPC was designed to provide a simple way for developers to create web services and web service clients using techniques that are not very different from those used in nondistributed Java programming. Programming with JAX-RPC is very similar to using RMI to create a distributed application, in the sense that client code appears to be making ordinary method calls on local objects. In reality, however, the infrastructure handles these calls by converting them to messages that are sent over a network to the server, where they cause a local call to be made on the actual method implementation. The results of this call are used to create a reply message that is sent back the client, where they are extracted and presented as return values from the client application's method call.

Although there are similarities between RMI and JAX-RPC, the major difference arises from the fact that the messages exchanged between JAX-RPC clients and services are encoded using an XML-based protocol and can potentially be carried by a range of transport-level protocols, including HTTP (or its more secure variant HTTPS), SMTP, or even FTP. JAX-RPC allows a client written in the Java programming language to access a service implemented on, for example, the Microsoft .NET platform, whereas RMI clients and servers must both be written in Java (although it is possible to expose an RMI/CORBA hybrid service written in any language that has a binding to CORBA IIOP). In other words, it can communicate with foreign services without needing to be aware of the technology that its peer is actually using.

One of the benefits of using JAX-RPC over a lower-level web services technology such as SAAJ or JAXM (both of which will be covered later in this book) is that it doesn't require you to know much about XML before you can start building a distributed application. This is because, with a few exceptions that fall into the advanced category, the programming interfaces are completely independent of both the underlying messaging infrastructure and the transport protocol that is used to carry the XML messages. The JAX-RPC specification requires every implementation to support at least the use of SOAP over HTTP 1.1, but, as a developer, you can use JAX-RPC without having to be an expert in XML, SOAP, or HTTP. On the other hand, it is possible to use some of the more advanced JAX-RPC features to gain access to the lower levels. Here, you can directly handle SOAP headers or extend the set of data types that the client and server can exchange beyond those supported transparently by JAX-RPC.

As Figure 3.2 shows, the service model for JAX-RPC is similar to other RPC models, such as RMI-IIOP and CORBA. The model has several components



## Fig3.2 JAX-RPC Service Model

The layers shown in Figure 3.2 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.
* 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.

# 

# 3.4 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 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 3.3 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.

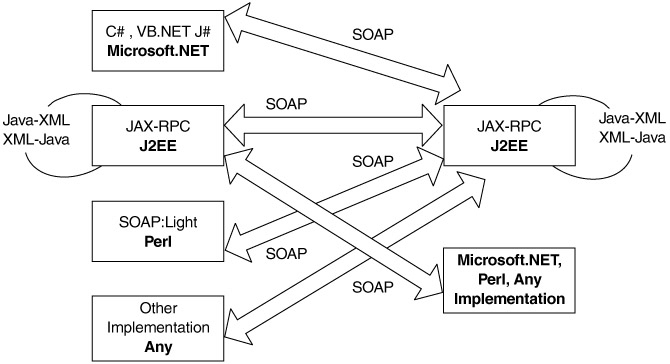


Fig 3.3 JAX-RPC client-server interaction

RPC implementations using SOAP from different vendors:

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

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

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

### 3.4.1 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 .
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.

1. An array (with the caveat that it must contain bytes or a supported type)
2. A java.lang.Exception class.
3. The above rules differ from the standard Java serialization requirements. Fig3.4 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.

| Fig 3.4: 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. |

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 shows. Fig 3.5

| **Java type** | **XML type** |
| --- | --- |
| Boolean | xsd:oolean |
| 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 |

Fig 3.5 Java-to-XML Data Type Mapping

# 3.5 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

### 3.5.1 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.

### 3.5.2 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.

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 lifecycle 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 context 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.

### 3.5.3 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 3.6 shows.

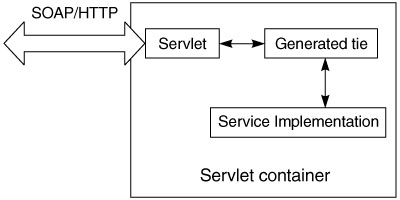


Fig 3.6 Service deployment

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 properties 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.

### 3.5.4 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 3.7 . A developer can start with

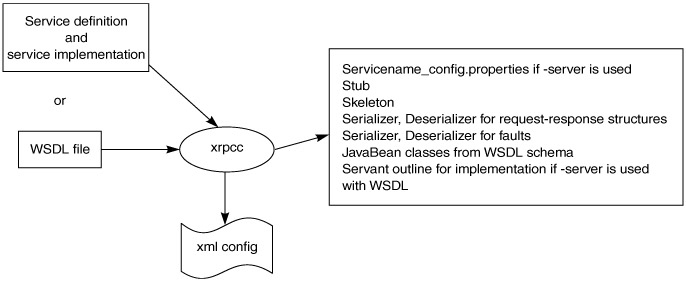


Fig 3.7 xrpcc artifacts

* 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

**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.

#### Java-WSDL Mappings

we will 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 3.8).

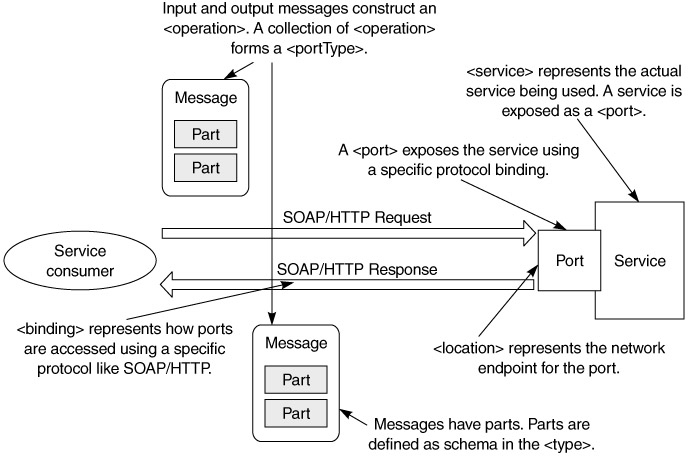


Fig 3.8 WSDL elements and dynamic interaction of a service and its consumer.

A Web service exposes groups of business operations for service consumers to use. Operations are grouped together to form portTypes. To invoke an operation, 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).

### 3.5.5 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.

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

|  |
| --- |
|  |