Table 5.1 Summary of WSDL message exchange patterns

Type	*Definition*
One-way	The operation can receive a message but will not return a response
Request/response	The operation can receive a request and will return a response
Notification	The operation can send a message but will not wait for a response
Solicit/response	The operation can send a request and will wait for a response

Table 5.1 summarizes and compares the four WSDL messaging patterns described in the previous section.

It should be noted that any combination of incoming and outgoing operations can be included in a single WSDL interface. As a result, the four types of operations presented above provide support for both push and pull interaction models at the interface level. The inclusion of outgoing operations in WSDL is motivated by the need to support loosely coupled peer-to-peer interactions between services.

5.3 Using WSDL to generate client stubs

In this chapter we have introduced several elements of WSDL to help readers understand the WSDL foundation for Web services. Understanding this technology is fundamental to comprehending the Web services model. However, most Web services developers will not have to deal with this infrastructure directly. There are a number of Web services development toolkits to assist with this undertaking. There are currently many tools, which automate the process of mapping WSDL to programming languages, such as Java, for both the service requestor and the service provider. One of the most popular tools for achieving this is WSDL2Java provided by Axis. Axis is an open source toolkit that is developed as part of Apache (**xml.apache.org**). Axis allows developers to write Java code and deploy that code as a Web service. In the following, we shall concentrate on how code generation tools allow automatic generation of WSDL definitions and creation of Web services.

Developers can implement Web services logic within their applications by incorporating available Web services and certainly without having to build new applications from scratch. The mechanism that makes this possible is the proxy class. Proxy classes enable developers to reference remote Web services and use their functionality within a local application, as if the data the services return was generated locally. The application developer communicates with any remote objects by sending messages to these local objects, which are commonly known as proxy objects. The proxy classes (or stub classes) are client-side images of the remote (provider) object classes that implement the Web services. The server-side counterparts are commonly known as skeletons in the distributed computing systems parlance, see section 2.4.1. Proxies implement the same interfaces as the remote class counterparts and forward the invoked methods on their local instances to corresponding remote instances (skeletons). The proxy object is simply a local object with methods that are merely a pass-through to the Web service it is representing. There exists

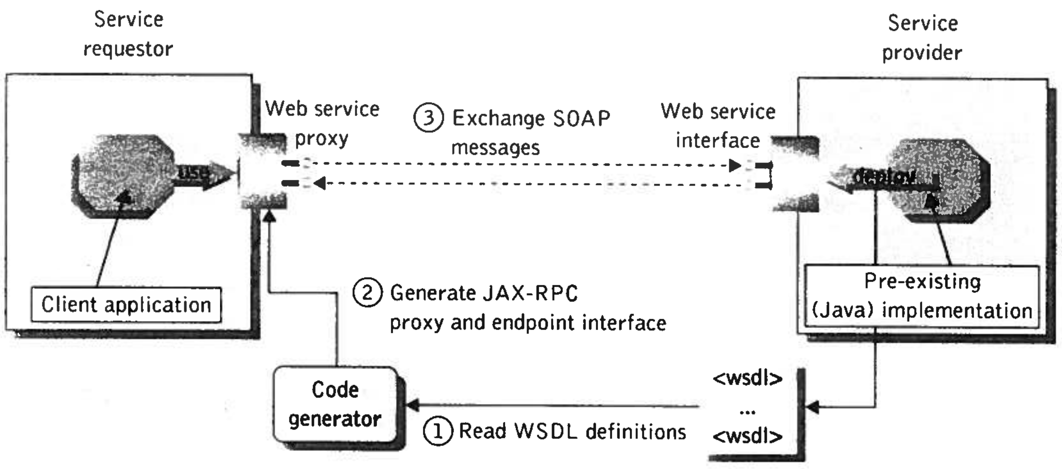

Figure 5.10 Generating proxies from WSDL code generators

exactly one proxy for each remote object for which a local object (application) holds a remote object reference. A proxy implements the methods in the remote interface of the service provider object it represents. This ensures that method invocations are suitable for the remote object in question. The role of the proxy class is to act as the local representative for the remote object and basically is, to the client, the remote reference. However, instead of executing an invocation, the proxy forwards it in a message to a remote object. The proxy hides the details of the remote object reference, the marshaling of arguments to the remote object methods using object serialization, and sends the marshaled invocation across the wire to the service provider's site. It also handles the unmarshaling of results that are received from the remote object implementing the service.

WSDL is well suited for code generators that can read WSDL definitions and generate a programming interface for accessing a Web service. For instance, a JAX-RPC provider may use WSDL 1.1 to generate Java RMI interfaces and network stubs, which can be used to exchange messages with a Web service interface. Figure 5.10 shows how a WSDL toolkit, such as JAX-RPC [Monson-Haefel 2004], can generate a Java RMI (an endpoint) interface and networking proxy that implements that interface. From the WSDL definition of a Web service, such as `PurchaseOrderService`, the Web service development tool generates a (Java) client proxy, which uses service requests from a local application (also shown to be coded in Java in Figure 5.10) to interact with a remote Web service implementing the WSDL interface.

WSDL code generator tools allow automatic creation of Web services, automatic generation of WSDL files, and invocation of Web services. These toolkits speed the creation of Web services by generating the service implementation template code from the WSDL specifications, leaving only the application-specific implementation details to the developer. They also simplify the development of client applications by generating service proxy code from the WSDL specification. Several code generators can generate interfaces and network stubs from WSDL documents.

Once the proxy class is built, the client simply calls the Web method from it, and the proxy, in turn, performs the actual request of the Web service. This request may, obviously, have its endpoint anywhere in the network. When we reference the Web service in

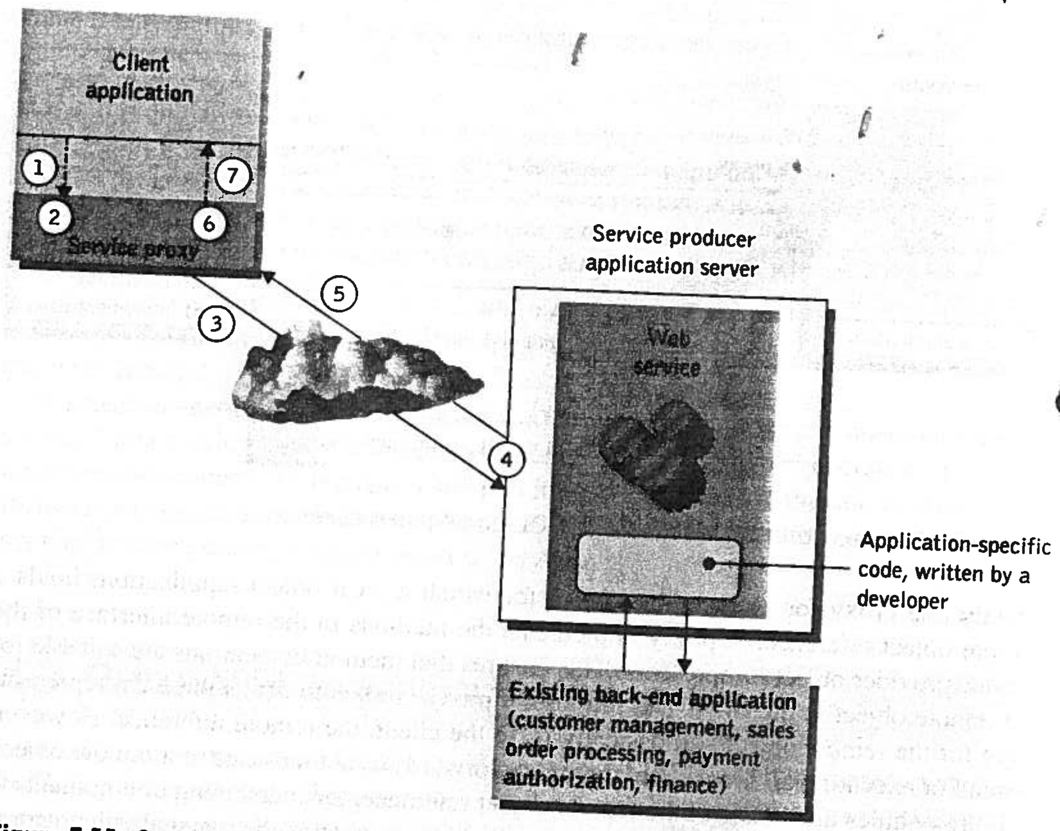
Client application
1
7
2
6
Service proxy
5
3
Service producer application server
Web service
4
Application-specific code, written by a developer
Existing back-end application (customer management, sales order processing, payment authorization, finance)

Figure 5.11 Communicating between proxy classes and Web services

the client application, it appears to be part of the consumer application itself, just like a normal internal function call. Figure 5.11 illustrates the process of communicating between a proxy and a Web service. This is shown to include the following steps:

1. The client-side application performs a call in the proxy class, passing any appropriate arguments to it, unaware that the proxy is actually invoking a remote Web service.
2. The proxy receives the call and formulates the request to the service, using the parameters the client application provides.
3. The call is transported from the proxy to the Web service across the network.
4. The Web service uses the parameters provided by the proxy to execute its Web service callable operations and expresses the result in XML.
5. The resulting data from the Web service is returned to the proxy at the client.
6. The proxy parses the XML returned from the Web service to retrieve the individual data values that are generated. These values may be simple or complex data types as already explained in this section.
7. The application receives the expected values in a normalized form from the proxy operation, completely unaware that they resulted from a Web service call.

5.4 Non-functional descriptions in WSDL

From what we have seen so far, one can understand that WSDL specifies the syntactic signature for a service but does not specify any non-functional service aspects. However, in section 1.8, we argued that non-functional characteristics are an integral part of any Web service. The Web services platform should be capable of supporting a multitude of different types of applications with different QoS requirements. In fact, programmers and applications need to be able to understand the QoS characteristics of Web services to be able to develop applications that invoke Web services and interact with them. Thus, the non-functional characteristics of a Web service should be described too.

QoS-enabled Web services require a separate language to describe non-functional characteristics of Web services. Currently, the most widely used approach to describing non-functional Web services characteristics is the combination of two specifications, which we shall examine in Chapter 12. These are WS-Policy and WS-PolicyAttachment. The Web services policy framework provides an additional description layer for services and offers a declarative policy language for expressing and programming policies. By employing this policy language, characteristics of the hosting environment can be described including security characteristics (including authentication and authorization) at the provider's endpoint, transactional behavior, the levels of QoS and quality of protection offered by the provider, privacy policies observed by the provider, and application-specific service options, or capabilities and constraints specific to a particular service domain.

When considering QoS-aware Web services, the service interface specifications need to be extended with statements on QoS that can be associated to the whole interface or to individual operations and attributes. It would be helpful if these non-functional service descriptions were to be added to WSDL in a standard manner. WS-PolicyAttachment accomplishes this objective (see section 12.4.3). It offers a flexible way of associating policy expressions with existing and future Web services artifacts. For instance, WS-PolicyAttachment addresses the requirements for associating Web services policy with a policy subject such as a WSDL `<portType>` or `<message>` and can even attach policies to UDDI entities.

5.5 Summary

A service description language is an XML-based language that describes the mechanics of interacting with a particular Web service and is inherently intended to constrain both the service provider and all requestors who make use of that service.

The Web Services Description Language is an XML-based specification schema providing a standard service representation language used to describe the details of the public interface exposed by a Web service. This public interface can include operational information relating to a Web service such as all publicly available operations, the XML message protocols supported by the Web service, data type information for messages, binding information about the specific transport protocol to be used, and address information for locating the Web service. WSDL allows the specification of services in terms of XML documents for transmission under SOAP.

The service implementation part of WSDL describes how a particular service interface is implemented by a given service provider. The service implementation describes where the service is located, or more precisely, to which network address the message must be sent in order to invoke the Web service.

WSDL specifies the syntactic signature for a service but does not specify any non-functional service aspects. The most widely used approach to describing non-functional Web services characteristics is by means of the Web services policy framework which provides an additional description layer for services and offers a declarative policy language for expressing and programming policies. This framework adds non-functional service descriptions to WSDL in a standard manner. In this way QoS characteristics such as performance, security (including authentication and authorization) at the provider's endpoint, transactional behavior, the levels of quality of protection offered by the provider, privacy policies observed by the provider, and so on, can be attached to the Web services description.

Currently, the World Wide Web Consortium is busy standardizing WSDL. Although WSDL 1.1, which we use throughout this book, is the *de facto* standard, WSDL 2.0 is the version of WSDL that the W3C is currently standardizing. WSDL 2.0 is a simpler and more usable language than WSDL 1.1. It provides several improvements over WSDL 1.1, including language clarifications and simplifications, support for interoperation, and making it easier for developers to understand and describe services. The recent WSDL 1.2 definition has introduced some changes, among which `<portType>` elements are renamed to `<interface>` elements [Weerawarana 2005]. The WSDL 1.2 definition also supports a useful new feature in the form of the `<extends>` attribute, which allows multiple `<interface>` declarations to be aggregated together and further extended to produce a completely new `<interface>` element. It is expected that the adoption of WSL 2.0 will take quite some time given the wide support that WSDL 1.1 enjoys in terms of tooling and run-time support environments [Weerawarana 2005].

Review questions

- Why is a service description necessary for representing Web services?
- What is the purpose of the Web Services Description Language? How does WSDL achieve its objective?
- Define and describe the Web services interface.
- Define and describe the Web services implementation.
- How do the Web services interface and implementation relate to each other?
- Describe the parts of the Web services `<portType>` element.
- Describe the parts of the Web services `<wsdl:binding>` element.
- How can you define RPC and document-style Web services in WSDL?