PART 1

Introducing Windows Communication Foundation

This part of the book introduces web service standards and fundamental components of Service-Oriented Architecture. We will also discuss how these principles are illustrated in Windows Communication Foundation (WCF). Once you have an understanding of some of these concepts, including the business and technological factors, you can appreciate the simplicity and flexibility of WCF. The first chapter will cover the services standards. Then we will introduce WCF in Chapter 2. This is followed by a discussion of the WCF programming model in Chapter 3.

CHAPTER 1

Introducing Service-Oriented Architecture

In today's world, implementing distributed systems that provide business value in a reliable fashion presents many challenges. We take many features for granted when developing nondistributed systems that can become issues when working with disparate distributed systems. Although some of these challenges are obvious (such as a loss of connectivity leading to data being lost or corrupted), for other aspects such as tightly coupled systems the dependencies between the various components of a system make it cost prohibitive to make changes as needed to meet the demands of the business.

Business processes quite often are supported by systems that are running on different platforms and technologies both within and outside the organization. Service-Oriented Architecture (SOA) is a mechanism that enables organizations to facilitate communication between the systems running on multiple platforms. This chapter introduces the fundamental concepts of SOA. The objective of this chapter is to discuss the following:

- What does SOA mean? How do you use messages, which act as the cornerstone for SOA implementations, to facilitate SOA?
- What makes SOA the preferred approach to design complex heterogeneous IT systems? Are web services the same as SOA?
- What are the four tenets of SOA?
- What are the implementation building blocks of SOA?
- How do you utilize all these building blocks to send messages between loosely coupled services?

Note To explain and demonstrate the various areas of Windows Communication Foundation (WCF), in this book we will show how to build an application that is modeled after a fictitious financial trading institution called QuickReturns Ltd. We will build a reference application that will articulate some of the challenges in today's enterprises and show how WCF can help solve some of the challenges. Each chapter will add functionality to this application, and you can download the code from the book's website. This case study will begin in Chapter 3.

What Is Service-Oriented Architecture?

It is not practical to build monolithic systems in current multinational enterprises. These systems often take many years to implement and usually address a narrow set of objectives. Today a business needs to be agile and adapt processes quickly, and SOA is a design principle that can help address this business need. SOA is a collection of *well-defined services*, where each individual service can be modified independently of other services to help respond to the ever-evolving market conditions of a business. Unlike traditional point-to-point architectures, an SOA implementation comprises one or more loosely coupled and interoperable set of application services. Although some of these aspects might be similar to a component-based development (which is based on strict interfaces), the key difference is SOA provides a message-based approach based on open standards. As a result of being based on open standards and using messages that are generic and not representative of any specific platform and programming language, you can achieve a high degree of loose coupling and interoperability across platforms and technologies. Each of these services is autonomous and provides one or more sets of business functions; in addition, since the underlying implementation details are hidden from the consumer, any change to the implementation will not affect the service as long as the contract does not change. This allows systems based on SOA to respond in a quicker and more cost-effective manner for the business.

For a business it is usually cheaper to "consume" an off-the-shelf application service that constitutes the solution instead of writing all the functionality. If a specific module needs to be updated for some reason, the company also benefits from the changes being confined to the specific service.

When coupled with industry-standard frameworks, service-based solutions provide the highly flexible "building blocks" that business systems require to compete in this age. Services encapsulate business processes into independently deliverable software modules. A service alone is just a building block; it is not a business solution but instead is an autonomous business system that is able to accept requests and whose interoperability is governed by various industry standards. These building blocks also provide the basis for increased improvements in quality and reliability and in the decrease of long-term costs for software development and maintenance.

In addition, even though there is a lot of talk about SOA today, point-to-point architectures are not disappearing. Many companies have invested a lot of resources in implementing proprietary solutions that mostly fulfill their business needs. SOA makes it easier to integrate point-to-point systems more easily because one system does not need to know the detailed mechanics of the other system. For those new to SOA, it is a little difficult to grasp this concept initially. This is primarily because SOA implementations target back-end systems. As a result, from a user's perspective, there are few user interface (UI) changes. However, you can also utilize SOA to provide front-end UI implementations. You can achieve this by combining service output XML with XSL to produce target HTML.

The SOA paradigm departs significantly from the OO model, where you are encouraged to encapsulate data. Therefore, an object will hold and protect the data to facilitate a business need. The enterprise will consist of multiple objects that are specialized to handle "specific scenarios" with the data protected within the objects. SOA instructs you to utilize loosely coupled services. The service describes the *contract* to the consuming entities. It does not tightly couple data to the service interface. It is also difficult to implement a single interface across all platforms and languages because of the nature of distributed systems. To fulfill the goals of

SOA, it is essential to implement the interfaces in a generic fashion. As a result, you need to express application-specific semantics in messages. The following are a few constraints for the messages that you need to consider when designing an SOA:

Descriptive: Messages need to be descriptive instead of prescriptive.

Limited structure: For different providers to understand the request, they need to understand the format, structure, and data types being used. This ensures maximum reach to all entities involved and limits the structure of the message. It also encourages you to use simple types, which are platform neutral.

Extensibility: Messages need to be extensible; only this provides the flexibility that allows SOA implementations to be quicker, faster, and cheaper than OO implementations.

Discoverability: Consumers and providers of messages need them to be discoverable so they know what is out there and how to consume the available services.

Disadvantages of Integrating Multiple Applications on Disparate Networks

We'll now discuss some challenges you'll face when you try to integrate multiple applications today. The following are some of the fundamental challenges when integrating multiple applications that reside on disparate physical networks:

Transports: Networks are not reliable and can be slow.

Data formats: The two applications in question are running on different platforms and using different programming languages, which makes interfacing with the various data types an interesting challenge.

Change: You know the applications need to change to keep up with the ever-evolving business requirements. This means any integration solution would need to ensure it could keep up with this change and minimize dependencies between the systems.

In the past, developers used several approaches to try to integrate applications in an enterprise. These approaches included file transfers, shared databases, remote procedure calls (RPC), and messaging. Although each of these approaches might make sense in some context, messages usually are more beneficial. We'll discuss some of the advantages of using messages in the following section.

Advantages of Using Messaging

The following are the advantages of using messages, like you do in SOA:

Cross-platform integration: Messages can be the "universal translator" between various platforms and languages, allowing each platform to work with their respective native data types.

Asynchronous communications: Messages usually allow for a "fire-and-forget" style of communication. This also allows for variable timing because both the sender and receiver can be running flat out and not be constrained by waiting on each other.

Reliable communication: Messages inherently use a “store-and-forward” style for delivery, which allows them to be more reliable than RPC.

Mediation: Messages can act as the mediator when using the Mediator pattern wherein an application that is disconnected needs to comment only to the messaging system and not to all the other applications.

Thread management: Since messages allow for asynchronous communication, this means one application does not have to block for the other to finish. Since this frees up many threads, the application can get to do other work overall, making it more efficient and flexible when managing its own threads.

Remote communication: Messages replace the need for the serialization and deserialization that occurs when one application makes a remote call to another application. Usually since these can be running on different process or even machines, the calls need to be marshaled across the network. The process of serializing an object to transfer over a network is called *marshaling*. Similarly, the process of deserializing an object on the other end is called *unmarshaling*.

End-to-end security: Unlike in the world of RPC, messages can transfer the “complete security context” to the consumer using a combination of headers and tokens. This greatly increases the ability to provide more granular control including authentication and authorization.

Messages are the “cornerstones” of SOA. Messages enable you to create loosely coupled systems that can span multiple operating systems. SOA relies on messages not only to facilitate the business need but also to provide the “context” around the message (that is, the security context, the routing information of the message, whether you need to guarantee the delivery of the message, and so on). Now you’ll dive into more SOA details.

Understanding Service-Oriented Architecture

SOA and web services are the buzzwords that promise to solve all integration issues in the enterprise space. Although any kind of implementation can be an SOA implementation, unfortunately many implementations using web services are marketed as SOA implementations, when in reality they are not.

SOA can be simply defined as an architectural concept or style that uses a set of “services” to achieve the desired functionality. A *service* is an autonomous (business) system that accepts one or more requests and returns one or more responses via a set of published and well-defined interfaces. Unlike traditional tightly coupled architectures, SOA implements a set of loosely coupled services that collectively achieve the desired results.

Note It is important to understand that although SOA might seem abstract, it is a significant shift from the earlier days of procedural and object-oriented languages to a more loosely coupled set of autonomous tasks. SOA is more than a collection of services. It’s a methodology encompassing policies, procedures, and best practices that allow the services to be provided and consumed effectively. SOA is not a “product” that can be bought off the shelf; however, many vendors have products that can form the basis of an SOA implementation.

It is important that the services don't get reduced to a set of interfaces because they are the key communication between the provider and the consumer. A *provider* is the entity providing the service, and the *consumer* is the entity consuming the service. In a traditional client-server world, the provider will be a server, and the consumer will be a client. When factoring in services, try to model the flow and process based on recognized business events and existing business processes. You also need to answer a few questions to ensure a clean design for services:

- What services do you need?
- What services are available for you to consume?
- What services will operate together?
- What substitute services are available?
- What dependencies exist between services and other versions of services?

Service orientation as described earlier is about services and messages. Figure 1-1 shows an example of how various service providers, consumers, and a repository coexist to form an SOA implementation. *Service providers* are components that execute some business logic based on predetermined inputs and outputs and expose this functionality through an SOA. A *consumer*, on the other hand, is a set of components interested in using one or more of the services offered by the providers. A *repository* contains a description of the services, where the providers register their services and consumers find what services are provided.

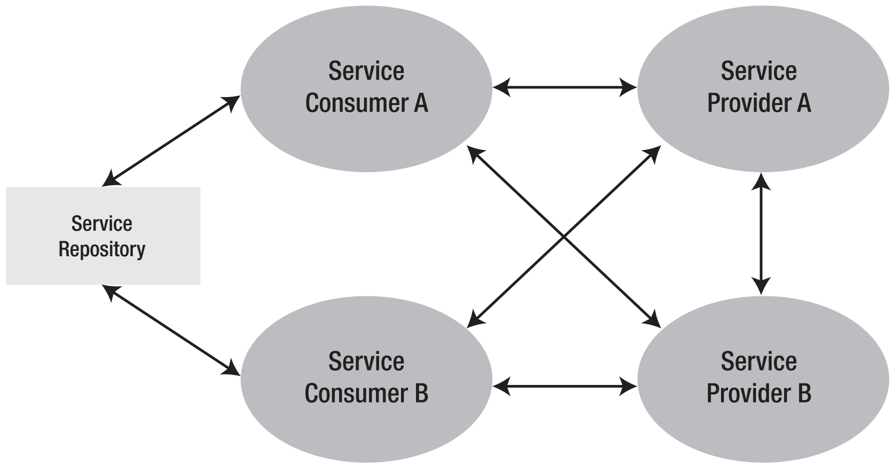

Figure 1-1. *How SOA components interact with each other*

What Is a Service?

The term *services* has been used to describe everything from web services (discussed in detail in the section "Web Services As a Key Enabling Technology for an SOA Implementation" later in the chapter) to business processes and everything in between. You should use services to represent the functions of the business and explicitly define the boundaries of what the business does, which essentially would define what the service can or cannot do. The key is that it is not a technology-driven approach but, rather, is a business-driven approach.

■ **Note** *Loose coupling* means any two entities involved reduce the assumptions they make about each other when they try to exchange information. As the level of assumptions made between two entities goes up (such as the kind of communication protocol used), so does the efficiency between the two entities; for example, the communication between the entities is very efficient. However, at the same time, the two entities are less tolerant to changes or, say, interruptions in the communication, because they are tightly bound or coupled to each other. Local method invocation is an excellent example of tight coupling because there are many assumptions made between the called routine and the calling routine, such as they both need to be in the same process, use the same language, pass the same number of parameters in the agreed data formats, and so on.

Service orientation is a business-driven "modeling strategy" that defines the business functionality in terms of loosely coupled autonomous business systems (or services) that exchange information based on messages. The term *services* is used in many contexts, but in the context of service orientation, a service is based on four fundamental tenets. We'll discuss these four tenets, originally proposed by the WCF team at Microsoft, in the following sections.

Tenet 1: Boundaries Are Explicit

Crossing boundaries is an expensive operation because it can constitute various elements such as data marshaling, security, physical location, and so on. Some of the design principles to keep in mind vis-à-vis the first tenet are as follows:

Know your boundaries: A well-defined and published public interface is the main entry point into the service, and all interactions occur using that.

Services should be easy to consume: It should be easy for other developers to consume the service. Also, the service interface should allow the ability to evolve over time without breaking existing consumers of the service.

Avoid RPC interfaces: Instead, use explicit messages.

Keep the service surface area small: Provide fewer public interfaces that accept a well-defined message, and respond likewise with a well-defined message. As the number of public interfaces grows, it becomes increasingly difficult to consume and maintain the service.

Don't expose implementation details: These should be kept internal; otherwise, it will lead to tight coupling between the consumer and the service.

Tenet 2: Services Are Autonomous

Services are self-contained and act independently in all aspects such as deploying, versioning, and so on. Any assumptions made to the contrary about the service boundaries will most likely cause the boundaries to change themselves. Services need to be isolated and decoupled to accomplish the goal of making them autonomous.

The design principles to keep in mind for the second tenet are as follows:

- Service versioning and deployment are independent of the system in which they are deployed.
- Contracts, once published, should not be changed.
- Adopt a pessimistic approach, and isolate services from failure.

■ **Note** Business Process Execution Language (BPEL) is a business process language that is based on XML and built using web service standards. You can use BPEL to define and manage a long-running business process. BPEL is an orchestration language and is used for abstracting the "collaboration and sequencing" logic from various web services into a formal process definition that is based on XML, Web Services Description Language (WSDL), and XML Schema. BPEL is also known as BPEL4WS or WSBPEL.

Tenet 3: Services Share the Schema and Contract, Not the Class

Services interaction should be using policies, schemas, and behaviors instead of classes, which have traditionally provided most of this functionality. The service contract should contain the message formats (defined using an XML schema), message exchange patterns (MEPs, which are defined in WSDL), any WS-Policy requirements, and any BPEL that may be required. The biggest challenge you face is the stability of the service, once it has been published. It gets difficult to change it then without impacting any of the consumers.

The design principles to keep in mind for the third tenet are as follows:

- Service contracts constituting data, WSDL, and the policy do not change and remain stable.
- Contracts should be as explicit as possible; this will ensure there is no confusion over the intent and use of the service. Additional contracts should be defined for newer versions of the server in the future.
- If breaking service contracts is inescapable, then version the services because this minimizes the ripple to existing consumers of the service.
- Do not expose internal data representation publicly; the public data scheme should be absolute.

Tenet 4: Service Compatibility Is Based on Policy

At times you will not be able to express all the requirements of service interaction via WSDL alone, which is when you can use policies. Policy expressions essentially separate the structural and semantic compatibility. In other words, they separate "what is communicated" and "how/whom a message is communicated." A policy assertion identifies a behavior of a policy entity and provides domain-specific semantics. When designing a service, you need to ensure that policy assertions are as explicit as possible regarding service expectations and semantic compatibilities.

The four tenets of service orientation provide you with a set of fundamental principles when you are designing services. When defining a service, it is always easier to work with well-defined requirements because that allows for a well-defined scope and purpose of a service. This enables a service to encapsulate distinct functionality with a clear-cut context. Sadly, more often than not, requirements are not well defined, which poses more of a problem. It is difficult to define the service that accurately represents its capabilities because one cannot relate the service operations by some logical context.

When defining services from scratch, it is helpful to categorize them according to the set of existing business service models already established within the organization. Because these models already establish some of the context and purpose in their boundary, it makes it easier to design the new services.

In addition, the naming of the service should also influence the naming of the individual operations within the service. As stated earlier, a well-named service will already establish a clear context and meaning of the service, and the individual operations should be rationalized so as not to be confusing or contradict the service. Also, because the context is established, the operations should also try to avoid confusing naming standards. For example, if you have a service that performs stock operations, then one of the operations in that should be GetQuote instead of GetStockQuote, because the context has already been established. Similarly, if you can reuse the service, then avoid naming the operations after some particular task, rather trying to keep the naming as generic as possible.

Naming conventions might not seem important at first, but as your service inventory in the organization grows, so will the potential to reuse and leverage the existing service to achieve integration within the various groups and systems. The effort required to establish a consistent naming convention within an organization pays off quickly. A consistent set of services that cleanly establish the level of clarity between the services enables easier interoperability and reuse.

Unfortunately, no magic bullet can help you standardize on the right level of granularity that will enable service orientation. But, the key point to remember is the service should achieve the right balance to facilitate both current and upcoming data requirements, in essence meeting the business's need to be more agile and responsive to market conditions.

"COMPONENTS" AND "SERVICES"—ARE THEY THE SAME?

It is natural to be confused about the terms *component* and *services* and what they mean. A *component* is a piece of compiled code that can be assembled with other components to build applications. Components can also be easily reused within the same application or across different applications. This helps reduce the cost of developing and maintaining the application once the components mature within an organization. Components are usually associated with the OOP paradigm.

A *service* is implemented by one or more components and is a higher-level aggregation than a component. Component reuse seems to work well in homogeneous environments; service orientation fills the gap by establishing reuse in heterogeneous environments by aggregating one or more components into a service and making them accessible through messages using open standards. These service definitions are deployed with the service, and they govern the communication from the consumers of the service via various contracts and policies, among other things.

SOA also assists in promoting reuse in the enterprise. Services can provide a significant benefit because you can achieve reuse at many levels of abstraction compared to the traditional methods (in other words, object orientation provide only objects as the primary reuse mechanism). SOA can offer reuse at multiple levels, including code, service, and/or functionality. This feature enhances flexibility to design enterprise applications.

WCF makes it easier for developers to create services that adhere to the principle of service orientation. For example, on the inside, you can use OO technology to implement one or more components that constitute a service. On the outside, communication with the service is based on messages. In the end, both of these technologies are complementary to each other and collectively provide the overall SOA architecture.

Although there have been a few attempts to solve the distributed application problem in the enterprise, there has yet to be a more demanding need to be consistent for standardizing. The scope of an SOA approach allows you to incorporate far-reaching systems across a number of platforms and languages. One great example of standardization in an enterprise today is web services, which we will discuss in the next section. Web services expose functionality that can be discovered and consumed in a technology-neutral, standardized format.

Web Services As a Key Enabling Technology for a Service-Oriented Architecture

There has been a lot of discussion about SOA and web services in the past few years. It might seem that web services and services are analogous in the context of SOA. On the surface this might seem accurate, but the reality is far from it. A web service is just *one* kind of implementation of a service. Web services are just a *catalyst* for an SOA implementation. In recent years with the relative ease that allows one to create web services, it has become easier to deliver SOA implementations; the SOA concept is not new, and certain companies (such as IBM) have been delivering it for more than a decade.

Almost everyone talks about web services, but interestingly no definition is universally acceptable. One of the definitions that is accepted by some is as follows: “A web service is a programmable application component accessible via standard web protocols.” The key aspects of a web service are as follows:

Standard protocol: Functionality is exposed via interfaces using one of the few standard Internet protocols such as HTTP, SMTP, FTP, and so on. In most cases, this protocol is HTTP.

Service description: Web services need to describe their interfaces in detail so that a client knows how to “consume” the functionality provided by the service. This description is usually provided via an XML document called a *WSDL document*. (WSDL stands for Web Services Description Language.)

Finding services: Users need to know what web services exist and where to find them so the clients can bind to them and use the functionality. One way for users to know what services exist is to connect to a “yellow pages” listing of services. These yellow pages are implemented via Universal Discovery, Description, and Integration (UDDI) repositories. (These can be private or public UDDI nodes.)

Note A web service is not an object model and is not protocol specific. In other words, it's based on a ubiquitous web protocol (HTTP) and data format (XML). A web service is also not dependent on a specific programming language. You can choose to use any language or platform as long as you can consume and create messages for the web service.

Figure 1-2 shows the basic protocol stack for web services. Interaction with the service will usually follow a top-down fashion—that is, service discovery down to messaging—invoking the methods on the service. If you are new to web services and do not understand the various protocols and standards, don't worry, because we will be describing them later in this chapter.

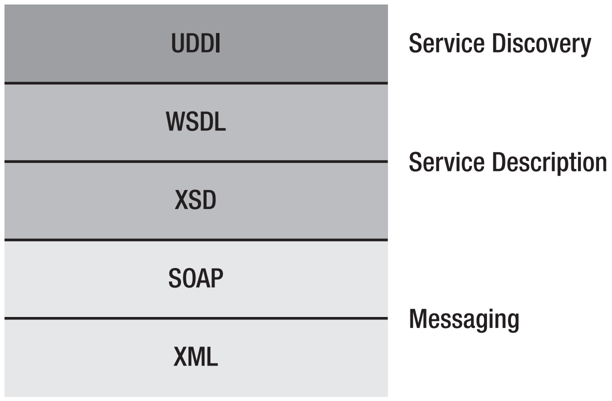

Figure 1-2. *Protocol stack for web services*

To consume a web service, you first find the service and what it offers (you can accomplish this by using UDDI). Once you have found the web service, you need to understand the interface: what the methods are, the parameters accepted, and so on. Traditionally, part of this discovery also entails what the data types are and the schema that the service expects. You can achieve the service description using WSDL and XML Schema Definition (XSD). However, with WCF, the recommendation is to use UDDI purely for publishing Web Services Metadata Exchange (WS-MetadataExchange, or MEX) endpoints and ask the service directly for the WSDL and policies. Lastly, the client needs to invoke the web service from the client via SOAP, which is based on XML.

WCF services also follow the open standards stack. Therefore, this stack not only addresses web services but it also describes the protocol stack for any service. You can argue the terminology *services* originated from *web services*. By definition, services do not need to depend on IIS or web servers to provide hosting environments. WCF enables developers to host services outside IIS. (We'll discuss this topic in detail in Chapter 5.) Therefore, you do not need to restrict the services to originate from a web server. Hence, they do not need to be called *web services*. The protocol stack plays a major role in understanding SOA functionality. Therefore, we'll discuss these protocols one by one. We will start with SOAP.

Note Web services use metadata to describe what other endpoints need to know to interact with them. This includes WS-Policy (which describes the capabilities, requirements, and general characteristics), WSDL, and XML Schema. To bootstrap communication with web services and retrieve these and other types of lmetadata, you use MEX, which is a specification that defines messages to retrieve metadata and policies associated with an endpoint. This is an industry-standard specification agreed on by most of the leading software companies such as Microsoft, IBM, Sun Microsystems, webMethods, SAP, and so on. The interactions defined by this specification are intended for metadata retrieval only and are not used to retrieve types of data such as states, properties, and so on, that might be associated with the service. For the detailed specification, see `http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-metadataexchange.pdf`.

Introducing SOAP

Simply put, SOAP is a lightweight communication protocol for web services based on XML. It is used to exchange structured and typed information between systems. SOAP allows you to invoke methods on remote machines without knowing specific details of the platform or software running on those machines. XML is used to represent the data, while the data is structured according to the SOAP schema. The only thing both the consumer and provider need to agree on is this common schema defined by SOAP. Overall, SOAP keeps things as simple as possible and provides minimum functionality. The characteristics of a SOAP message are as follows:

- It is extensible.
- It works across a number of standardized underlying network protocols.
- It is independent of the underlying language or platform or programming model.

Note SOAP used to stand for Simple Object Access Protocol, but the W3C dropped that name when the focus shifted from object "access" to object "interoperability" via a generalized XML messaging format as part of SOAP 1.2.

SOAP recognizes the following message exchange patterns: one-way, request-response, and so on. Figure 1-3 shows a one-way SOAP message (that is, no response is returned). The SOAP sender will send the message over some communication protocol.

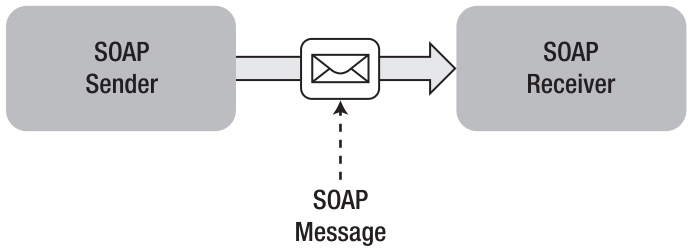

Figure 1-3. *Simple one-way SOAP message*

As Figure 1-3 shows, the SOAP message can be sent over any communication protocol, and the sender and receiver can be written in any programming model or can run on any platform.

Extensible

Extensibility is the key factor for SOAP in addition to the simplicity of design. Extensibility allows various features such as reliability, security, and so on to be "layered" via SOAP extensions. Every vendor defines its own set of extensions providing many feature-rich features on its platform.

Transport

SOAP can use one of the many standard transport protocols (such as TCP, SMTP, FTP, MSMQ, and so on). You need to define standard protocol bindings, which outline the rules for the environment to address interoperability. The SOAP specification provides a flexible framework for defining arbitrary protocol bindings and provides an explicit binding for HTTP because it's so widely used.

■ **Note** Most programmers new to SOAP are confused about the difference between the SOAP specification and the vendor implementations of the SOAP specification. Developers usually use a SOAP toolkit to create and parse SOAP messages instead of handcrafting them. The types of functional calls and supported data types for the parameters vary between each vendor implementation. As a result, a function that works with one toolkit may not work with the other. This is not a limitation of SOAP but rather a limitation of the particular vendor-specific implementation being used.

Programming Model

One of the strengths of SOAP is that it is not tied to RPC but can be used over any programming model. Most developers are surprised to learn that the SOAP model is akin to a traditional messaging model (such as MSMQ) and less to an RPC style, which is how it is used primarily. SOAP allows for a number of MEPs, one of them being the request-response model.

■ **Note** MEPs are essentially a set of design patterns for distributed communications. MEPs define the template for exchanging messages between two entities. Some examples of MEPs include RPC, Representational State Transfer (REST), one-way, request-response, and so on. RPC is essentially a protocol that allows one application to execute another application or module on another computer, without the developer having to write any explicit code to accomplish the invocation. REST, on the other hand, is an architectural style that is different from RPC. This uses a simple XML- and HTTP-based interface, but without the abstraction of protocol such as SOAP. Some purists see this as the subset of the "best" architectures of the Web.

SOAP Message

SOAP's building block is the SOAP *message*, which consists of the following four parts:

- A SOAP *envelope* is an XML document that encapsulates the message to be communicated. This is the only part that is required; the rest is optional.
- The second part of the SOAP message is used to define any custom data types that the application is using.
- The third part of the message describes the RPC pattern to be used.
- The last part of the message defines how SOAP binds to HTTP.

A SOAP envelope is the root element of the message. The *envelope* has two sections: the header and the body. The header has metadata about the message that might be needed for some specific processing, such as a date-time stamp when the message was sent or an authentication token. The *body* contains the details of the message or a SOAP fault. Figure 1-4 shows the structure of a SOAP message. A SOAP message can be one of three types, namely, request messages, response messages, and fault messages.

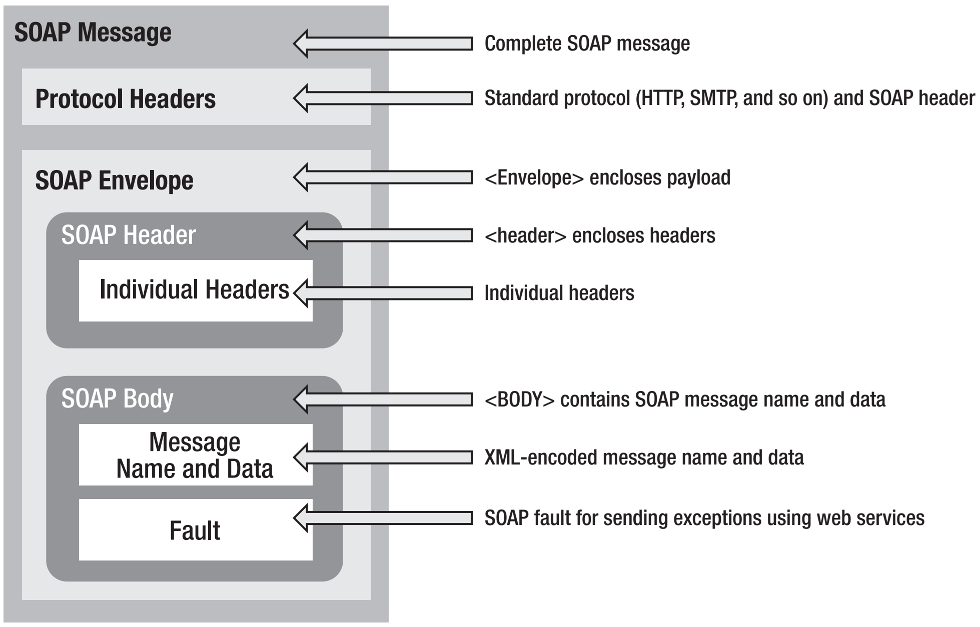
SOAP Message
Protocol Headers
SOAP Envelope
SOAP Header
Individual Headers
SOAP Body
Message Name and Data
Fault
Complete SOAP message
Standard protocol (HTTP, SMTP, and so on) and SOAP header
<Envelope> encloses payload
<header> encloses headers
Individual headers
<BODY> contains SOAP message name and data
XML-encoded message name and data
SOAP fault for sending exceptions using web services

Figure 1-4. *SOAP message structure*

A request-response SOAP message, as Figure 1-5 shows, essentially has two messages: one is the request message sent to the service, and the other is a response message sent back to the client. Although these messages are independent from each other, the request-response pattern provides automatic correlation and synchronization between the messages. This results in "normal" procedure call semantics to the client.

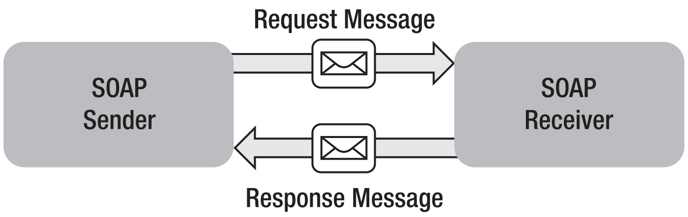

Figure 1-5. *Request-response exchange pattern*

Syntactically a SOAP message is quite simple, but the following are a few rules you need to keep in mind when manually writing SOAP messages:

- Must be encoded in XML
- Must use the SOAP envelope namespace
- Must use the SOAP encoding namespace
- Cannot contain a DTD reference
- Cannot contain XML processing instructions

Note Contract first or code first? When designing a service contract, you have two approaches; you can either use the contract-first approach or use the code-first approach. The contract-first approach ensures interoperability and for most situations is the recommended approach. This ensures that any consumers of the services conform to the published contract. This is especially important if any third parties are involved who need to conform to a predetermined contract. One of the biggest challenges to the contract-first approach is the lack of tool support, which hurts productivity. If in a given situation productivity has a higher precedence than interoperability, then it might make sense to use the code-first approach. Please refer to Chapter 4 for a more detailed discussion.

SOAP Faults

When an exception needs to be returned by the service, this takes place using a fault element in a SOAP response. A fault element needs to be within the body element of the SOAP message and can appear only once. The fault element must contain a `faultcode` element and a `faultstring` element. The fault message contains the exception details such as error code, description, and so on. Table 1-1 lists the fault codes defined by the specification.

Table 1-1. *SOAP Fault Codes*

Error	Description
`VersionMismatch`	The SOAP receiver saw a namespace associated with the SOAP envelope that it did not recognize.
`MustUnderstand`	The receiver of the message did not understand a required header.
`Client`	The message sent by the client was either not formed correctly or contained incorrect information.
`Server`	An error happened at the server while processing the message.

SOAP Message Format

Two types of SOAP messaging formats or styles exist: document and RPC. Document style indicates the message body contains XML where the format must be agreed upon between the sender and the receiver. The RPC style indicates it is a representation of a method call.

You also have two ways to serialize the data to XML: using literal XML schema definitions and using SOAP encoding rules. In the first option, the schema definition defines the XML format without any ambiguity. These rules are defined in the specification and are sometimes referred to as *Section 5 encoding*. It is more suitable for the RPC-style usage of SOAP, because this option specifies how to serialize most data types including arrays, objects, structures, and so on. On the other hand, in the second option, the various SOAP encoding rules (usually defined via a W3C schema) must be parsed at runtime to determine the proper serialization of the SOAP body. Although the W3C permits the use of both document-literal and RPC-literal formats, Microsoft recommends the former over the latter because the message should not dictate the way it's being processed by the service.

SOAP Implementations by Major Software Vendors

Every major software vendor in the industry has a SOAP stack implemented that covers both the open source market and the commercial market. On the open source front, solutions exist from the likes of Apache Axis implemented in Java/C++ to PocketSoap implemented via COM. Implementations exist for almost all platforms such as Soap::Lite in Perl and PHP Soap in, you guessed it, PHP.

On the commercial side of things, every major software vendor has a stack that is tightly integrated with its other offerings. Microsoft, of course, has the .NET Framework, which tightly integrates with Visual Studio and ASP.NET to provide a seamless experience for a developer. IBM's implementation is its WebSphere web services that allow you to integrate with technologies such as JAX-B, EMF/SDO, and XMLBeans. Oracle's web service is implemented as part of the Oracle Application Server (10g) line. Sun Microsystems has implemented the web services stack via the Web Services Developer Pack (WSDP), and BEA has WebLogic Server.

■**Note** <soaprpc/> lists web service implementations by various vendors at `http://www.soaprpc.com/ws_implementations.html`.

Some vendors do not implement the complete Web Services Interoperability (WS-I) profile stack but do only a subset. Security, for example, has had a lot of interest from most enterprise organizations. Both hardware devices (such as XML Firewall from Reactivity) and software products (such as XML Trust Services from VeriSign) exist that provide many security features and act as catalysts for web services. We'll address some of these implementations in detail in Chapter 13.

With so many implementations covering almost all platforms and runtimes, it is not possible to standardize on the various data types used by the components and how they are represented in memory. A consumer and provider running on different platforms will not be able to exchange any information without standardizing, thus nullifying the promise of web services. To help solve this, the W3C recommends XSD, which provides a platform-independent description language and is used to describe the structure and type of information using a uniform type system.

How do you describe a service to the consuming party? What open standards do you have to "describe" a service to its consumer? You achieve this by implementing WSDL. You'll now learn a bit more about WSDL.

Web Services Description Language: Describing Service Endpoints

If no standards existed, it would have been difficult for web services and in turn SOAs to be so widely accepted. WSDL provides the standardized format for specifying interfaces and allows for integration. Before we get into the details of WSDL, you need to understand the notion of endpoints because they are paramount to WSDL.

What Are Endpoints?

Officially, the W3C defines an endpoint as “an association between a fully specified interface binding and a network address, specified by a URI that may be used to communicate with an instance of a web service.” In other words, an *endpoint* is the entity that a client connects to using a specific protocol and data format when consuming a service. This endpoint resides at a well-known location that is accessible to the client. At an abstract level, this is similar to a port, and in some implementations such as BizTalk, this can literally be exposed as a port for others to consume over. When looking from the perspective of endpoints, a service can also be defined as a collection of endpoints.

WSDL

WSDL (pronounced as “whiz-dull”) forms the basis of web services and is the format that describes web services. WSDL describes the public interface of a web service including metadata such as protocol bindings, message formats, and so on. A client wanting to connect to a web service can read the WSDL to determine what contracts are available on the web service. If you recall from earlier in the chapter, one of the key tenets of service orientation is that the services share contracts and schemas, not classes. As a result, when you are creating a service, you need to ensure the contract for that service is well thought out and is something you would not change.

Any custom types used are embedded in the WSDL using XML Schema. WSDL is similar to Interface Description Language (IDL) for web services. The information from the WSDL document is typically interpreted at design time to generate a proxy object. The client uses the proxy object at runtime to send and receive SOAP messages to and from the service.

■ **Note** IDL is a standardized language used to describe the interface to a component or routine. IDL is especially useful when calling components on another machine via RPC, which may be running on a different platform or build using a different language and might not share the same “call semantics.”

A WSDL document has three parts, namely, definitions, operations, and service bindings, which can be mapped to one of the elements listed in Table 1-2.

Table 1-2. *WSDL Document Structure*

Element	Description
`<portType>`	Operations performed by the web service
`<message>`	Message used by the web service
`<types>`	Data types used by the web service
`<binding>`	Defines a communication endpoint (by means of protocol and address) to access the service
`<Service>`	Aggregates multiple ports (in combination with binding and address into a service)

Definitions

Definitions are expressed in XML and include both data type and message definitions. These definitions are based upon an agreed-on XML vocabulary that in turn should be based on a set of industry-wide vocabulary. If you need to use data type and message definitions between organizations, then an industry-wide vocabulary is recommended.

■ **Note** Definitions are not constraints to XML and can be expressed in formats other than XML. As an example, you can use the Object Management Group (OMG) IDL instead of XML. If you use a different definition format, as with XML, both the senders and receivers would need to agree on the format and the vocabulary. However, as per the official W3C WSDL specification, the preference is to use XSD as the canonical type system. Sticking to this would ensure maximum interoperability and platform neutrality.

Operations

Operations describe the actions for the message supported by the web service and can be one of four types, as listed in Table 1-3. In a WSDL document structure, operations are represented using the `<portType>` element, which is the most important element because it defines the operations that can be performed. In context of the OO paradigm, each operation is a method.

Table 1-3. *Operation Types*

Error	Description
One-way	The service endpoint receives a message.
Request-response	The service endpoint receives a message and sends a correlated message.
Solicit-response	The service endpoint sends a message and receives a correlated message.
Notification	The service endpoint sends a message.

Figure 1-6 shows the structure of a WSDL document, which consists of abstract definitions and concrete descriptions. On the left is the abstract definition where the data type's definition is a container for using some type system such as XSD. *Message definitions*, as the name suggests, are the typed definitions of the data being communicated. An *operation* is a description of the action supported by the service and contains one or more data types and message definitions, as shown in Figure 1-6. Port types are a set of operations supported by more than one endpoint. All of these are brought together by the *binding*, which is the concrete protocol and data format specified for a particular port type. A *port* is a single endpoint defined as the combination of the binding and the network address. Most developers do not handcraft the WSDL but instead use the underlying .NET Framework to generate this for them.

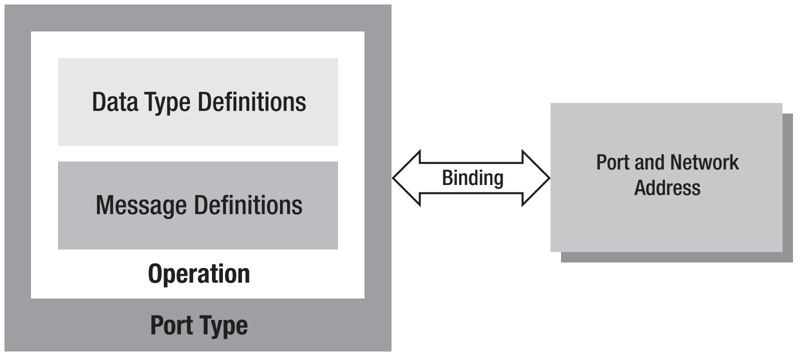

Figure 1-6. *WSDL*

Service Bindings

Service bindings connect port types to a port (that is, the message format and protocol details for each port). A port is defined by associating a network address with a port type. A service can contain multiple ports. This binding is commonly created using SOAP. The binding element has two attributes: a name that can be anything to define the binding and the type, which points to the port for the binding.

Now you are familiar with SOAP and how you describe the services with WSDL. However, how can you discover all the services that are available to you? What open standards are available to dynamically discover services and consume them at runtime? You achieve this by implementing UDDI. You'll now investigate what UDDI is.

Dynamically Discovering Web Services

UDDI is a platform-independent directory protocol for describing services and discovering and integrating business services via the Internet. UDDI is also based on industry-standard protocols such as HTTP, XML, SOAP, and so on, and it describes the details of the services using WSDL and communicates via SOAP. The philosophy behind UDDI is like a traditional "yellow pages" where you can search for a company, search for the services its offers, and even contact the company for more information.

A UDDI entry is nothing but an XML file that details the business and the services it offers. It consists of three parts: the white, yellow, and green pages. The *white pages* contain details of the company and its contact information. The *yellow pages* include industry categories based on standardized taxonomies such as the North America Industry Classification System, and so on. The *green pages* contain the technical details that describe the interface via a WSDL so a consumer has enough information about how to use the service and what is expected. A UDDI directory also includes several ways to search for the services it offers including various filtering options such as geographic location, type of business, specific provider, and so on.

Before UDDI was standardized, there was no universal way to know what services were offered by partners or off-the-shelf options. You also have no way to get standard integration and dependencies between providers. The problems that UDDI solves are as follows:

- UDDI makes it possible to discover the right service that can be used, instead of reinventing the wheel.
- UDDI solves the customer-driven need to remove barriers to allow for the rapid participation in the global Internet economy.
- UDDI describes services and business processes programmatically in a single, open, and secure environment.

■**Note** UDDI offers more than design-time support. It plays a critical role after the discovery of a service because it allows the client to programmatically query the UDDI infrastructure, which further allows the client applications to be more robust. With a runtime layer between the client and the web service, the clients can be loosely coupled, allowing them to be more flexible to the changes. Microsoft recommends publishing a WS-MEX as part of the entries in the UDDI registry.

Now you are familiar with SOAP, WSDL, and UDDI. However, how do all these technologies work together to send a message from the sender to the receiver in a loosely coupled system? You'll now investigate how to achieve this.

Sending Messages Between Loosely Coupled Systems

To achieve service orientation, you need the ability to send messages from one service to another. In the context of WCF, *service invocation* is a general mechanism for sending messages between an entity that requests a service and another entity that provides the service. It is important to understand that it does not matter where the provider and consumer physically exist; they could be on the same physical machine or spread across the opposite ends of the planet. However, from a service *execution* perspective, it matters, and WCF fills this infrastructure gap.

Service invocation, irrespective of platform and technology, follows a similar pattern. At a high level, the steps involved when a consumer sends a message to a provider are as follows:

1. Find the relevant service that exposes the desired functionality.
2. Find out the type and format of the messages that the service would accept.
3. Understand any specific metadata that might be required as part of the message (for example, for transaction or security).
4. Send the message to the provider with all relevant data and metadata.
5. Process the response message from the service in the appropriate manner (for example, the request might have been successful, or it might have failed because of incorrect data or network failure, and so on).

Because web services are the most popular implementation of service orientation, let's look at an example of how you would use a web service to send messages from one application to another. Invoking a web service "method" is similar to calling a regular method; however, in the first case, the method is executed on a remote machine. As the web service is running on another computer, all the relevant information needed by the web service needs to be passed to the machine hosting the service. This information then is processed, and the result is sent to the client.

The life cycle of an XML web service has eight steps, as shown in Figure 1-7:

1. A client connects to the Internet and finds a directory service to use.
2. The client connects to the directory service in order to run a query.
3. The client runs the relevant query against the directory service to find the web service that offers the desired functionality.
4. The relevant web service vendor is contacted to ensure the service is still valid and is available.
5. The description language of the relevant web service is retrieved and forwarded to the client.
6. The client creates a new instance of an XML web service via the proxy class.
7. The runtime on the client serializes the arguments of the service method into a SOAP message and sends it over the network to the web service.
8. The requested method is executed, which sets the return value including any out parameters.

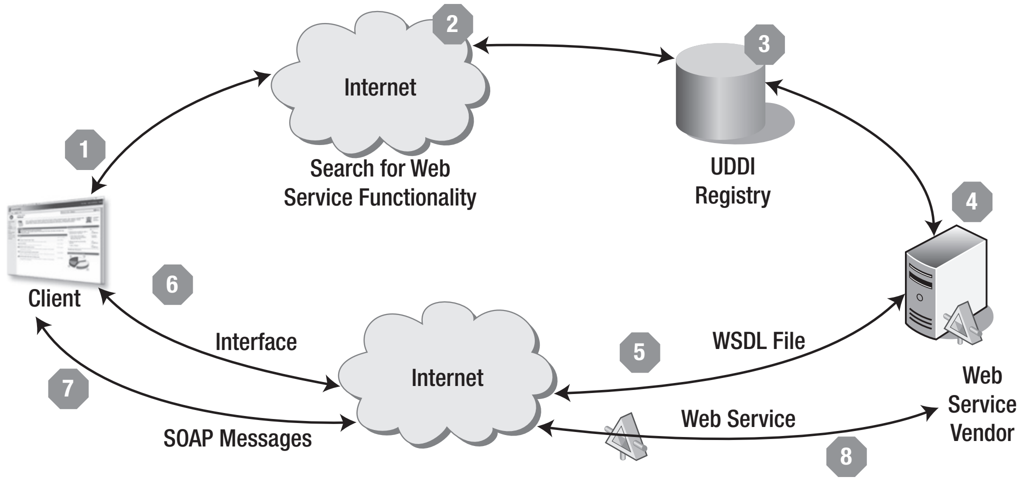

Figure 1-7. *Web service lifetime*

■ **Note** Although we have shown the example of a web service implementation, WCF provides unification across the many distributed computing technologies and provides a common programming model irrespective of the underlying protocol or technology. Also, differences exist in the design time and the runtime of a WCF implementation. At design time, for example, an SOA implementation would usually account for the definition of the service, the different data types it expects in what order, and so on. But many elements are accounted for only at runtime, such as the physical location of the service, network latency and failure, error detection and recovery, and so on. All these attributes are not usually addressed at design time but are picked up by the WCF runtime; the WCF runtime can also change the behaviors of the service during the actual operation. This behavior can also be adjusted based on the demand of the service. For more details, refer to Chapter 2.

Summary

This chapter introduced the concepts of services and SOA. We also described the governing principles for services, in terms of the four tenets. It is important to understand that at the end of the day SOA is not about how to invoke objects remotely or how to write web services; it is all about how to send messages from one application to another in a loosely coupled fashion. Web services are just one of the many, albeit the most popular, ways to send messages between disparate systems. Adopting an SOA approach is important to an enterprise to help it deliver the business agility and IT flexibility needed to be able to succeed in today's marketplace.

The next chapter will introduce you to the new features of WCF, the challenges it helps solve, and the unification of the various distributed technologies. We will also illustrate how WCF addresses SOA concepts to promote WCF as a practical SOA implementation from Microsoft.