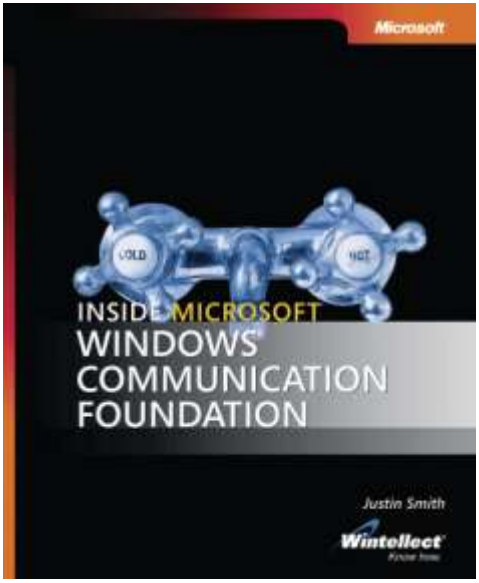

Inside Microsoft® Windows® Communication Foundation

Justin Smith

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/9610.aspx

9780735623064
Publication Date: May 2007

Microsoft Press

Table of Contents

Chapter 2

Service Orientation

In this chapter:

The Internet is awash with talk of service orientation (SO), and most of that discussion addresses service orientation in the abstract. We are going to take a slightly different approach in this chapter. In the next few pages, we'll look at service orientation from a requirements perspective. More specifically, we're going to look at a generic messaging application and expose what is required to make it tick. Through this process, we'll unearth some of the concepts that are essential to comprehending service orientation. The last sections of this chapter are devoted to a more formal definition of service orientation and a discussion of why service orientation makes sense in today's world of distributed computing.

If you ask 10 "SO-savvy" people to define service orientation, you'll probably get 10 different answers. If you ask them again in a couple of years, you'll probably get a different set of answers. This phenomenon is not new. When object orientation (OO) and component-driven development arrived in the mainstream, many developers were confused as to how they should adapt or reconceive their procedural designs given these new architectural models. Understanding OO and component architectures required a fundamental shift in thinking about application designs. The process was at times painful, but the payoffs are more robust designs, greater code reuse, advanced application functionality, easier debugging, and shorter time to market. In my opinion, moving to SO designs from component-driven designs will require a fundamental shift in thinking of the same magnitude as the move from procedural architectures to OO. The good news is that SO designs offer tremendous benefits in the form of richer communication patterns, loosely coupled applications, improved application

functionality, and fulfilling the promise of true application interoperability. Because the term *interoperability* is heavily overloaded, some specificity is needed to avoid confusion. In this context, interoperability refers to the ability for a system to change hardware, operating system, or platform without affecting the other participants in the distributed scenario.

Service orientation, despite the current confusion associated with its definition, is not a new concept. It has been around since the reign of the mainframe and has been more recently adopted as a paradigm in middleware. Recent initiatives toward interoperability and richer communication patterns have reignited interest in service orientation and are moving SO into the mainstream. It's reasonable to assume that the definition of service orientation will evolve as it becomes more widely implemented.

A Quick Definition of Service Orientation

In a nutshell, service orientation is an architectural style in which distributed application components are loosely coupled through the use of messages and contracts. Service-oriented applications describe the messages they interact with through contracts. These contracts must be expressed in a language and format easily understood by other applications, thereby reducing the number of dependencies on component implementation.

Notice that I am not mentioning vendors or technologies when describing service orientation. SO is a concept that transcends vendor and technology boundaries, much in the way that object orientation also transcends these boundaries. OO can be a confusing concept, both initially and when taken to extremes, and I expect the same to be true of SO. For this reason, I will first illustrate SO with a series of examples, and I'll avoid defining abstract concepts with other abstract concepts.

Getting the Message

Messages are the fundamental unit of communication in service-oriented applications. For this reason, service-oriented applications are often called *messaging applications*. At some point, every SO application will send or receive a message. It is helpful to think of a service-oriented message as similar to a letter you receive in the mail. In the postal system, a letter is an abstract entity: it can contain almost any type of information, can exist in many different shapes and sizes, and can relate to almost anything. Likewise, a service-oriented message is an abstract entity: it can contain almost any data, can be encoded in many different ways, and can relate to virtually anything, even other messages. Some properties of a postal letter are widely accepted to be true. For example, a letter is always sent by someone, sent to someone, and might be delivered by someone (more on that "might be" in a moment). Likewise, a service-oriented message is sent by a computer, sent to a computer, and might be delivered by computer. To satisfy the theory wonks, I must say that in the purest sense, entities that interact with service-oriented messages do not have to be computers. Theoretically, they could be

carrier pigeons, Labradors, or maybe even ligers. Regardless, the entities that interact with service-oriented messages are called *messaging participants*, and in this book, a messaging participant will be a process on a computer.

Messaging Participants

Let's imagine that I need to send a thank-you letter to my friend Rusty for giving me tickets to a football game last week. Let's also assume that I will send the letter to Rusty's office. In real life, it's probably easier and cheaper to send an e-mail message to Rusty, but that makes for a more complicated example, and sometimes a written letter is simply more appropriate. What sort of steps would I follow to send Rusty the thank-you letter?

As we all know, the order of these steps is open to several variations, but at some point before I send the letter, I have to write the letter. As I am writing the letter, I'll probably want to reference the football game, as it would be unusual to send a thank-you letter expressing thanks for nothing in particular. Next I would put the letter in an envelope. Then I would write the delivery address on the envelope and place the necessary postage on the envelope. The last step is to drop the letter in any mailbox and let the postal service deliver the letter to Rusty. I am assuming that Rusty will know the letter is from me and that he will know that I appreciated the football tickets.

When we describe messaging participants, it's often helpful to label them according to the role they play in the message delivery. In general, there are three types of messaging participants: the *initial sender*, the *ultimate receiver*, and the *intermediaries*. In our thank-you letter scenario, I am the initial sender, Rusty is the ultimate receiver, and the mail system and Rusty's office staff are intermediaries.

Let's imagine a more real-world business scenario—the order processing system at Contoso Boomerang Corporation. Basically, customers place boomerang orders on the Web site, and the Web site generates an order message and sends it to other internal systems for processing and fulfillment, as shown in Figure 2-1.

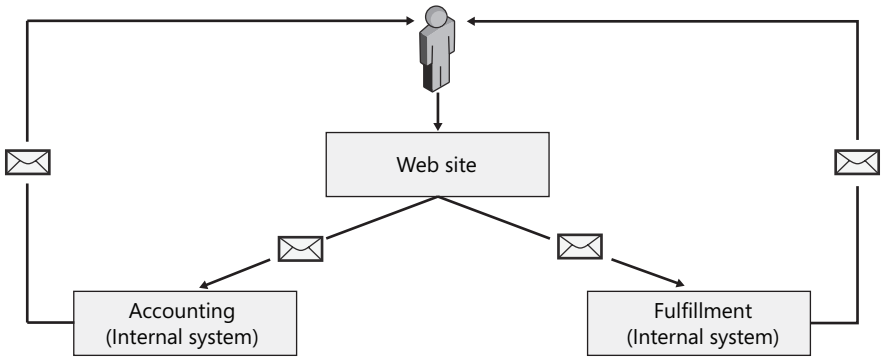

Figure 2-1 Message flow at Contoso Boomerang Corporation

Several facts are implied in this scenario:

- The Web site and the other internal systems have previously agreed upon the format of the message.
- The Web site can create the message in the previously agreed upon format.
- The Web site knows how to send the message to other internal systems.
- The internal systems can use data in the received message to fill the order, send a confirmation message, and ship the order.

Contoso's order processing system has at least two messaging participants. The Web site is the initial sender, and the internal systems are the ultimate receivers. It might be the case that we also have a load-balancing messaging router that routes Web site orders to the proper internal system. As shown in Figure 2-2, we can consider this router an intermediary.

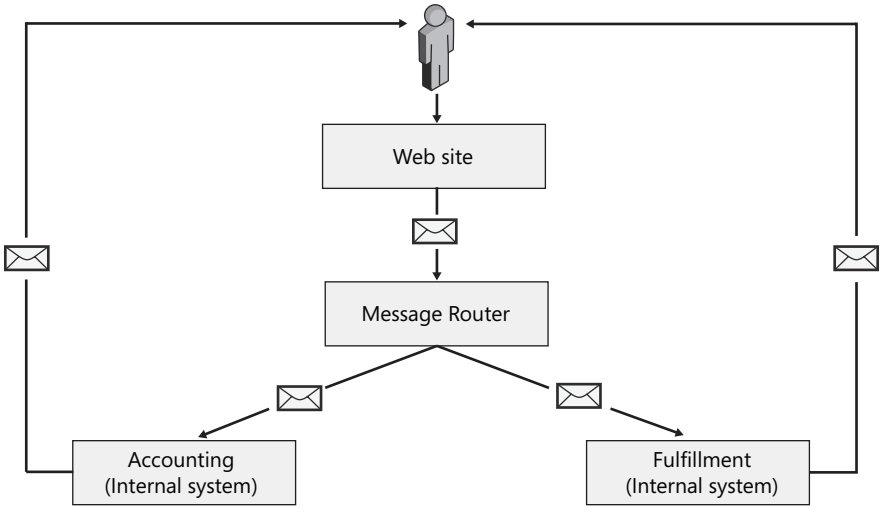

Figure 2-2 Message flow at Contoso Boomerang Corporation with a messaging router

The Initial Sender

Identifying the initial sender can be harder than it looks. In our thank-you letter example, I might appear to be the initial sender. It is plausible, however, to look at my letter as a response to Rusty's action of sending me the tickets. If we follow this train of thought, Rusty is the initial sender, and I am sending him a thank-you letter as a response to his generosity. Along those same lines, it is also possible that I sent Rusty a letter two months ago asking him for the tickets. In this case, I am the initial sender. Rusty was responding to me when he sent the tickets, and my thank-you message is a response to Rusty's response. It is also possible that one of our common friends suggested to Rusty that he should send me the tickets. In this case, our common friend is the initial sender.

Our order processing system can display the same ambiguity. At first glance, the Web site might appear to be the initial sender. It might not look that way, however, from the perspective

of the internal systems. From that point of view, the initial sender might appear to be either the Web site or another internal system (remember the message router). We could go on an on, but the reality is that the initial sender is *relative*. By relative, I mean that the initial sender of a message might change based on the context assigned to the message. In both of our examples, we can draw an arbitrary boundary around two or more participants and change the initial sender of the message.

If we drop the *initial* in initial sender, we have a much more concrete vision of a messaging participant. If we revisit the thank-you letter example, Rusty probably doesn't care who the initial sender is; he simply needs to know who sent the thank-you letter. In practice, the distinction between the initial sender and just a sender is often not worth determining. For this reason, I will use the term *sender* instead. If you see the term *initial sender* in any World Wide Web Consortium (W3C) documents or specifications, be aware of the subtlety embodied in the definition. Given these parameters, the following is how I describe a sender:

A sender is an entity that initiates communication.

Intermediaries

Several people have handled the thank-you letter as it was being delivered to Rusty. To name a few:

- The postal worker who picked up the letter from the mailbox
- The postal workers at the sorting facility
- The postal worker who delivered the mail to Rusty's office building
- The mailroom workers at Rusty's office building who delivered the letter to his office

Through experience, we have come to understand that we don't know how many people will handle a letter as we send it through the mail. We do expect certain behaviors, however, from those handling our mail. For example, we expect them to not open the mail or materially alter its contents. We also expect that each mail handler will move the letter closer, either in process or in location, to our intended recipient. These message waypoints are called intermediaries. Given these parameters, I define an intermediary as follows:

An intermediary is invisible to the sender and is positioned between the sender and the ultimate recipient.

Identifying intermediaries is also harder than it looks at first glance. In our postal example, isn't a mail carrier simply picking up a message and sending it forward to another mail carrier? Isn't the next mail carrier simply picking up a message delivered from another mail carrier and forwarding the message on? Wouldn't a mail carrier be an initial sender if he or she sends the message forward? It is physically true that each mail carrier handling the letter is sending the letter forward in the process. It is also true that each mail carrier handling the letter receives the letter from either another mail carrier or the sender. Logically, however, the mail carrier might be invisible to the sender and therefore not specifically addressed by the

sender. It is also true that mail carriers do not create the message; they are simply handling and delivering the message.

It is also possible, however, that the message envelope will be altered at some point during handling. Think of a postmark. Postmarks do not materially change the contents of the message, but they do provide some information that describes when and where the letter was received into the postal system. The postal service may also add a "Return to Sender" mark on the envelope if the delivery address is not valid. At a high level, these are the types of operations that can be performed by an intermediary. An intermediary should not, however, change the contents of the message.

Let's reexamine Contoso's order processing system for a more computer-based example of an intermediary. As it turns out, Contoso sells custom boomerangs and standard boomerangs. Orders for standard boomerangs are processed through Contoso's inventory system, while custom boomerangs must be sent to the manufacturing system. The system architects at Contoso might have decided to put this logic in a routing system, further encapsulating business logic away from the Web site. The effect of this design is that the Web site sends messages to message routing servers. This routing system might not materially change the contents of the message, but it does route the order to either system. At a high level, the routing system is acting as an intermediary between the initial sender (the Web site) and the ultimate receiver (the inventory or manufacturing systems).

A Few Words About Business Logic

This additional layer in the architecture can be very useful in capturing a business process. In the past, applications "hard-coded" business processes in their applications. For example, business requirements or regulations might require the Contoso accounting system to receive payment for boomerangs before orders are fulfilled. The traditional distributed system paradigm spreads the logic of this business process between the Web site, the accounting system, and the fulfillment system. This design has a major drawback: when business requirements or regulations evolve, each part of the system requires modification.

In recent years, companies have spent fortunes trying to develop their own internal mechanism for dealing with this problem. Often these efforts involved defining a proprietary XML grammar for expressing business processes and building a custom runtime engine for interpreting these rules. It is my guess that, more often than not, these efforts ended badly.

As mentioned in Chapter 1, "The Moon Is Blue," Microsoft Windows Communication Foundation (WCF) ships with a product called Windows Workflow Foundation (WF). Among other things, WF is designed to capture these sorts of business processes. WF does much of the heavy lifting previously required to build this sort of business process engine. In the next few years, expect workflow to be more a part of business application development.

The Ultimate Receiver

My thank-you letter was intended to go to my friend Rusty. When I sent the letter, I had no idea how many people were going to handle it, but I hoped that each handler would work toward delivering the letter to Rusty. As a result, I define the ultimate receiver as follows:

The ultimate receiver is the intended, addressable target of a message.

A single message can have only one logical ultimate receiver. For example, it is not possible to address a postal letter to more than one address. Physically, however, one address could reference multiple entities. For example, if Rusty's department is responsible for sending the football tickets, I could address the thank-you message to the entire department. My intention in this case is that everyone in the department will receive the message. It is also possible that my message is posted on a bulletin board, sent around to each individual in the department, or announced in a department meeting. In the end, however, the message is intended for one logical entity, the ultimate receiver.

The Anatomy of a Message

Early in life, we learn that a postal stamp belongs in the upper-right corner of an envelope and that the address goes somewhere in the center. If we want, we can also add a return address in the upper-left corner of the envelope. All mail handled by the postal service must adhere to this basic structure. If mail is not metered, a delivery address is not present, or the delivery address is illegible, the postal service considers the mail invalid and will not deliver the letter. If we're lucky, invalid mail will be delivered to the return address (if one is specified). Imagine the chaos that would follow if such a structure did not exist. If senders were allowed to place postage or delivery addresses anywhere on the parcel, the postal service would have to scan the entire parcel for postage and delivery addresses. More than likely, the added infrastructure required to complete these tasks would add more than a couple of cents to the next postage rate hike! In practice, the parcel structure as defined by the postal service improves mail handling efficiency and consistency without sacrificing much usability from the sender's perspective.

In contrast to the postal example, SO messages do not have to follow structural pattern. Like the postal example, however, a predefined message structure does improve the processing efficiency, reliability, and functionality of the system. Remember that messaging applications are not conceptually new. Messages originating from a variety of application vendors have been passed between applications for decades. Without a standardized structure, each vendor is free to develop its own structure, and the result is a disparate set of message structures that do not interoperate well with one another.

If we look at companies like FedEx, UPS, and DHL, we see a similar paradigm. Each of these organizations has defined its own addressing format and packaging. It is atypical for an overnight package in a UPS box with a UPS label to be sent via FedEx. Technically it is possible,

but business pressures and efficiency preclude these companies from interacting with another type of address and parcel format.

It's not a huge leap to examine purchasable enterprise computing systems with the same concept. On the whole, vendors have not wanted their applications to interoperate with other systems. Vendors had a hard enough time getting their systems to communicate within a single product suite, let alone interoperate with other systems. In the past, customers were willing, to some extent, to stick within one particular application vendor's toolset to meet all of their enterprise needs. The choice customers faced was one of "Who can sell me the complete package?" rather than "What products are the best for each of my needs?" Over time, the one-stop-shopping paradigm has resonated less and less with would-be customers. As a result, software vendors have had to come to the table to produce a series of common messaging specifications and standards and make their applications produce messages that adhere to these standards. It has taken many years for these standards to be created and agreed upon, but they are finally here, and we can expect more over time.

There are literally dozens of these messaging standards available, and we will examine many of these specifications as you move through this book. Many of these specifications are based, in one form or another, on SOAP, and each serves a specific purpose. For the intellectually curious, the full SOAP specification is available at *http://www.w3.org/TR/soap12-part1/*. As a result of SOAP's flexibility, modern SO messages are usually SOAP messages.[1] At its core, SOAP is a messaging structure built on XML. SOAP defines three major XML elements that can be used to define any XML message you want to send: the envelope, the body, and the header. Here is an example of the key parts of a raw SOAP message:

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Header>
    ...
     </env:Header>
  <env:Body>
    ...
  </env:Body>
</env:Envelope>
```

Because WCF is an SO platform intended for, among other things, interoperability with other systems, it sends, receives, and processes SOAP messages. As you'll see in Chapter 4, "WCF 101," we can think of WCF as a toolkit for creating, sending, and parsing SOAP messages with a myriad of different behaviors. For now, let's take a closer look at what all SOAP messages have in common.

1 WCF supports SOAP, REST, and POX. Most of the current WCF application programming interface (API), however, is dedicated to the SOAP message structure. This will undoubtedly expand in the future to include other message structures, like JSON.

Envelope

As its name implies, the envelope wraps both the body and the header. All SOAP messages have an envelope as a root element. The envelope element is often used to define the different namespaces (and prefixes) that will be used throughout the message. There is not much else that's terribly exciting about SOAP envelopes.

Header

A SOAP header is optional and, if present, it must be the first element after the envelope start tag. A SOAP header is composed of zero or more SOAP header blocks. SOAP header blocks contain information that can be used by the ultimate receiver or by an intermediary. Typically these header blocks contain data that is orthogonal to the message body's data. To put it another way, security information, correlation, or message context can be placed in a header part. Header blocks are mandatory if certain messaging behaviors are expected. Once again, this idea can be illustrated through the postal system. If I want to send a piece of mail through the postal system and receive a return receipt when the parcel is delivered, I have to fill out a special return receipt label and affix it to the envelope. Adding a return receipt to the parcel does not materially change the contents of the parcel. It can, however, change the behavior of messaging participants: I have to fill out and affix the return receipt request, the postal carrier must ask for a signature, the ultimate receiver must sign the receipt, and the postal carrier must deliver the receipt to me (or at least my mailbox).

SO messages can contain similar information in the header. For example, in our order processing scenario, the Web site might want to receive a confirmation that the order message was received by an entity other than the message router. In this case, the Web site could assign a unique identifier to the message and add a special header to the message requesting an acknowledgment. Upon receipt, the message router forwards the message on to the appropriate system and demands that the system produce an acknowledgment. That acknowledgment could then be returned to the Web site directly or through the message router.

It is also possible that an intermediary might modify an existing SOAP header block or even add a brand new SOAP header block to a message. In practice, however, an intermediary should never change or delete a header block unless it is intended for them. Using this model, it would be fairly easy to create a message that contains auditable records of its path. Each intermediary can add its own SOAP header, so by the time the message arrives at the ultimate receiver, the message contains a list of all intermediaries that have touched the message. As described earlier, this behavior is modeled in the real world in the postal system with postmarks or as described in our message router example.

Body

The body element is mandatory and typically contains the payload of the message. By convention, data found in the body is intended for the ultimate receiver only. This is true regardless of how many firewalls, routers, or other intermediaries process the SOAP message. This is only an informal agreement. Just as there is no guarantee that the postal service will not open our mail, there is no guarantee that an intermediary will not open or change the SOAP body. It is possible, however, to use digital signatures and encryption to digitally ensure the integrity of a message as it passes from initial sender to ultimate receiver.

Message Transports

SOAP messages are transport agnostic. In other words, there is no need to place transport-specific information into a message. This simple feature is one of the key features that make SOAP such a powerful messaging structure. Once again, our postal service example can provide an illustration. If a postal message was sent with a dependency on the transport, it would be equivalent to telling your postal carrier where you want the message to be delivered and not including that information on the envelope of the message. If we follow this train of thought, the message is tightly bound to the postal carrier. This tight coupling is bad for several reasons:

- The message can be delivered only to places the postal carrier can go.
- No other postal worker can interact with the message (unless the previous postal carrier communicates it).
- Batch sorting and delivering of messages is difficult.
- Because there is no return address on the message, the sender cannot be notified if something goes wrong while the message is processed.

From a service-oriented perspective, this is a terrible scenario. A much better plan would be to include all relevant addressing information in the message itself, thereby preventing a strong tie to the transport layer. When messages include this information, a myriad of SOAP behaviors (including the aforementioned behaviors) are possible. For example, we all know that mail is picked up by a postal carrier, delivered to a sorting facility, and then sent on to other sorting facilities and postal carriers via planes, trains, boats, or trucks. In our everyday mail example, we see that the transport can change during the delivery of the message (carrier, sorting facility, plane, and so on), and this improves efficiency. None of that is possible if each message does not contain an address.

Message Encodings

Over time, many of us have been conditioned to think of XML (and therefore SOAP) as structured text. After all, text is human readable, and every computing system can process text. The universal nature of text-based XML resonates with our desire to interoperate with connected systems. Text-encoded XML, while being easy to interpret, is inherently bulky. It is reasonable to expect some performance penalty when using XML. Just as it takes some effort to place a thank-you letter in an envelope, it takes some processing time to interact with XML. In some cases, however, the sheer size of text-encoded XML restricts its use, especially when we want to send an XML message over the wire.

Furthermore, if we restrict ourselves to text-encoded XML, how can we send binary data (like music or video) in an XML document? If you've read up on your standard XML Schema data types, you will know that two binary data types exist: *xs:base64Binary* and *xs:hexBinary*. Essentially, both of these data types represent data as an ordered set of octets. Using these XML data types might have solved the problem of embedding binary data in a document, but they have actually made the performance problem worse. It is a well-known fact that base64-encoded data inflates data size by roughly 30%. The story is worse for *xs:hexBinary*, since it inflates the resultant data by a factor of 2. Both of these factors assume an underlying text encoding of UTF-8. These factors double if UTF-16 is the underlying text encoding.

The XML Infoset

To find the answer to our performance dilemma, let's take a closer look at exactly what makes up an XML document. If we look at the specifications, XML is a precise syntax for writing structured data (as defined at *http://www.w3.org/TR/REC-xml/*). It demands that well-formed XML documents have start and end elements, a root node, and so on. Oddly enough, after the XML specification was released, a need arose to abstractly define XML documents. The XML Infoset (as defined at *http://www.w3.org/TR/xml-infoset/*) provides this abstract definition.

In practice, the XML Infoset defines the relationship between items, without defining any specific syntax. This lack of a specific syntax in the XML Infoset leaves the door open for new, more efficient encodings. If our parser adheres to the XML Infoset, as opposed to the XML syntax, we can interpret a variety of different message encodings, including ones more efficient than text, without materially altering our application.

SOAP and the XML Infoset

Remember that SOAP is built on XML. This raises a question: Are SOAP messages built on the earlier XML syntax or on the XML Infoset? The answer is both. Two SOAP specifications exist: SOAP 1.1 and SOAP 1.2. SOAP 1.1 is built on the older XML syntax, while SOAP 1.2 is built on the XML Infoset. Given this fact, it is reasonable to assume that a SOAP 1.2 message might not be readable by a SOAP 1.1 parser. WCF is built on the XML Infoset, but it has the capability to process both SOAP 1.1 and SOAP 1.2 messages.

WCF can be adapted and customized to work with virtually any message encoding, as long as the message is SOAP 1.1 or 1.2 compliant (it can also work with messages that are not SOAP messages). As you will see in subsequent chapters, WCF has a very pluggable and composable architecture, so custom encoders can be easily added to the WCF message pipeline. As new encodings are developed and implemented, either Microsoft or third parties can create these new encoders and plug them into the appropriate messaging stack. I will describe message encoders in greater detail in Chapter 6, "Channels." For now, let's take a look at the encoders included in WCF. At the time of this writing, WCF ships with three encoders: text, binary, and Message Transmission Optimization Mechanism (MTOM).

The Text Encoder

As you can guess from its name, the output of the text encoder is text-encoded messages. Every system that understands Unicode text will be able to read and process messages that have been passed through this encoder, making it a great choice when interoperating with non-WCF systems. Binary data can be included in text-encoded messages via the *xs:base64Binary* Extensible Schema Definition (XSD) data type. Here is a message that has been encoded by using the WCF text encoder (with some elements removed for clarity):

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>...</s:Header>
  <s:Body>
    <SubmitOrder xmlns="http://wintellect.com/OrderProcess">
      <Order xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
      <OrderByte xmlns="http://wintellect.com/Order">
mktjxwyxKr/9oW/jO48IhUwrZvNOdyuuquZEAIcy08aa+HXkT3dNmvE/
+zI96Q91a9Zb17HtrCIgtBwmbSk4ys2pSEMaIzXV3cwCD3z4ccDWzpWx1/
wUrEtSxJtaJi3HBzBlk6DMWOeghvnl652lKEJcUJ6Uh/LRlZz3x1+aereeOgdLkt4gCnNOEFECL8CtrJtY/taPM4A+k/
4E1JPnBgtCRrGWWpVkOOUqRXahz2XbShrDQnzgDwaHDf/
fHDXfZgpFwOgPF1IG88KQZOOJncSYKIp5I8OPYTeqDOyVhB8QSt9sWw59yzLHvU65UKoYfXA7RvOqZkJGtV6wZAgGcA=
=
        </OrderByte>
      <OrderNumber xmlns="http://wintellect.com/Order">
         12345
        </OrderNumber>
      </Order>
    </SubmitOrder>
  </s:Body>
</s:Envelope>
```

The Binary Encoder

The binary encoder is the most highly performing message encoder and is intended for WCF-to-WCF communication only. Of all the encoders in WCF, the binary encoder produces the smallest messages. Keep in mind that this encoder produces a serialized Infoset, even though it is in a binary format. It is likely that in the future, a standard binary encoding will be universally adopted, as these types of encodings can dramatically improve the efficiency of a messaging application.

The MTOM Encoder

The MTOM encoder creates messages that are encoded according to the rules stated in the MTOM specification. (The MTOM specification is available at *http://www.w3.org/TR/soap12-mtom/*.) Because the MTOM encoding is governed by a specification, other vendors are free to create infrastructures that send and receive MTOM messages. As a result, WCF messages that pass through the MTOM encoder can be sent to non-WCF applications (as long as those applications understand MTOM). In general, MTOM is intended to allow efficient transmission of messages that contain binary data, while also providing a mechanism for applying digital signatures. The MTOM message encoding enables these features through the use of Multipurpose Internet Mail Extensions (MIME) message parts and inline base64 encoding. The content of the MTOM message is defined by the Xml-binary Optimized Packaging recommendation. For more information, see *http://www.w3.org/TR/xop10/*.

At run time, the MTOM encoder creates an inline base64-encoded representation of the binary data for digital signature computation and makes the raw binary data available for packaging alongside the SOAP message. An MTOM encoded message looks as follows:

```
// start of a boundary in the multipart message
--uuid:7477fff7-61e6-4cd9-a8a5-e38f47fb042e+id=1
Content-ID: <http://wintellect.com/0>
Content-Transfer-Encoding: 8bit

// set the content type to xop+xml
Content-Type: application/xop+xml;charset=utf8; type="application/soap+xml"
<s:Envelope xmlns:s=”http://www.w3.org/2003/05/soap-envelope”>
  <s:Header>...</s:Header>
  <s:Body>
    <SubmitOrder xmlns="http://wintellect.com/OrderProcess">
      <order xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <OrderByte xmlns="http://wintellect.com/Order">
           // add a reference to another message part
          <xop:Include href=cid:http://wintellect.com/1/12345
           xmlns:xop="http://www.w3.org/2004/08/xop/include"/>
        </OrderByte>
        <OrderNumber xmlns="http://wintellect.com/Order">
          12345
        </OrderNumber>
      </order>
    </SubmitOrder>
  </s:Body>
</s:Envelope>

// end of the boundary in the first message part
--uuid:7477fff7-61e6-4cd9-a8a5-e38f47fb042e+id=1

// add the binary data as an octect stream
Content-ID: <http://wintellect.com/1/12345>
Content-Transfer-Encoding: binary
Content-Type: application/octet-stream

// raw binary data here
```

Notice that the binary data is kept in its raw format in another part of the message and referenced from the SOAP body. Since the binary data is packaged in a message part that is external to the SOAP message, how can one apply a digital signature to the SOAP message? If we use an XML-based security mechanism, like those stated in XML Encryption and XML Digital Signature, we cannot reference external binary streams. These encryption and signing mechanisms demand that the protected data be wrapped in a SOAP message. At first glance, it appears that there is no way around this problem with multipart messages. In fact, this was the Achilles' heel of Direct Internet Message Encapsulation (DIME) and SOAP with Attachments. MTOM provides an interesting way around this problem.

The MTOM encoding specification states that an MTOM message can contain inline binary data in the form of base64-encoded strings or as binary streams in additional message parts. It also states that a base64-encoded representation of any binary data must be available during processing. In other words, additional binary message parts can be created for message transmission, but inline base64 data must be temporarily available for operations like applying digital signatures. While the message is in this temporary inline base64-encoded state, an XML-based security mechanism can be applied to the SOAP message. After the security mechanism has been applied, the message can then be serialized as a multipart message. When the receiver receives the message, the message can be validated according to the rules set forth by the specific XML security mechanism.

It is also interesting to note that the WCF MTOM encoder reserves the right to serialize the binary chunks of a message as either inline base64-encoded strings or as binary streams in additional message parts. The WCF encoder uses the size of the binary data as a key determining factor. In our previous message, the *OrderBytes* element was about 800 KB. If we reduce the size of the *OrderBytes* element to 128 bytes and check the message format, we see the following:

```
// start of a boundary in the multipart message
--uuid:14ce8c5f-7a95-48d3-a4de-a7042f864fbc+id=1
Content-ID: <http://wintellect.com/0>
Content-Transfer-Encoding: 8bit

// set the content type to xop+xml
Content-Type: application/xop+xml;charset=utf8; type="application/soap+xml"

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>...</s:Header>
  <s:Body>
    <SubmitOrder xmlns="http://wintellect.com/OrderProcess">
      <order xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <OrderByte xmlns="http://wintellect.com/Order">
kF+k2CQd/lCitSYvXnLhuOtaMCk/tZaFZIWeW7keC3YvgstAWoht/wiOiR5+HZPo+TzYoH+qE9vJHnSefqKXg6mw/
9ymoV1i7TEhsCt3BkfytmF9Rmv3hW7wdjsUzoBl9gZ1zR62QVjedbJNiWKvUhgtq8hAGjw+uXlttSohTh6xu7kkAjgoO
3QJntG4qfwMQCQj5iO4JdzJNhSkSYwtvCaTnM2oiO/fBHBUN3trhRB9YXQG/mj7+ZbdWsskg/
Lo2+GrJAwuY7XUROKyY+5hXrAEJ+cXJr6+mKM3yzCDu4B9bFuZv2ADTv6/MbmFSJWnfPwbH1wK0LQi7Ixo95iF
        </OrderByte>
```

```
        <OrderNumber xmlns="http://wintellect.com/Order">
          12345
        </OrderNumber>
      </order>
    </SubmitOrder>
  </s:Body>
</s:Envelope>
--uuid:14ce8c5f-7a95-48d3-a4de-a7042f864fbc+id=1–
```

In this case, the WCF encoder opted to serialize the binary element as an inline base64-encoded string. This optimization is perfectly legal according to the MTOM specification.

Choosing the Right Encoding

Choosing the correct message encoding forces one to consider current and future uses of the message. For the most part, application interoperability and the type of data in the message will dictate your choice of message encodings. Performance, however, can also play a role in determining which encoding is best suited to your system. Table 2-1 ranks encodings based on what type of message is being sent and what sorts of systems can receive the message.

Table 2-1 Message Encodings by Rank and Scenario

Type of Message	Binary	Text	MTOM
Text payload, Interop with other WCF systems only	1	2	3
Text payload, Interop with modern non-WCF systems	N/A	1	2
Text payload, Interop with older non-WCF systems	N/A	1	N/A
Large binary payload, Interop with other WCF systems only	1	3	2
Large binary payload, Interop with modern non-WCF systems	N/A	2	1
Large binary payload, Interop with older non-WCF systems	N/A	1	N/A
Small binary payload, Interop with other WCF systems only	1	2	3
Small binary payload, Interop with modern non-WCF systems	N/A	1	2
Small binary payload, Interop with older non-WCF systems	N/A	1	N/A

It shouldn't be surprising that the binary encoding is the most efficient means to send messages to other WCF systems. What may come as a surprise, however, is the fact that MTOM messages can be less efficient, in an end-to-end sense, than text messages. Interoperability and the size of the binary data being sent are the two factors that should help you decide between MTOM and text encodings in your application. For the most part, one can send MTOM only

messages to systems that implement an MTOM encoder. At the time of this writing, MTOM is a fairly new specification, so only modern systems can effectively process MTOM messages. From a performance perspective, the MTOM encoder makes sense only when the binary data being wrapped in a message is fairly large. MTOM should never be used with messages that do not contain binary data because MTOM's performance will always be worse than the regular text encoding. It is important, however, to run independent tests using messages that accurately represent those in production.

Luckily, as we'll see in Chapter 4, "WCF 101," WCF is designed in such a way that these encoding choices do not require a major change in the application. In fact, it is possible to have one service that can interact with different message encodings. For example, one service can interact with both binary-encoded and text-encoded messages. The benefit in this scenario is that the service can be very highly performing when communicating with other WCF participants and still interoperate with other platforms, like Java.

Addressing the Message

Now that you have seen the entities that can interact with a message, taken a close look at message anatomy, and seen the different message encoders that ship with WCF, let's examine how we can express where we want a message to be sent. After all, messages aren't terribly useful unless we can send them to a receiver. Just as the postal service requires a well-defined addressing structure, service-oriented messages also require a well-defined addressing structure. In this section, we will build our own addressing scheme, see whether it is broadly applicable to messaging applications, and then relate it to the addressing scheme that is typically used with WCF messages.

In-Transport Addressing vs. In-Message Addressing

Service-oriented messages specify the ultimate receiver directly in the message. This is a subtle but important point. If the target of the message is specified in the message itself, a whole set of messaging patterns becomes possible. You will learn more about messaging patterns in Chapter 3, "Message Exchange Patterns, Topologies, and Choreographies".

When we insert an address directly into a message, we pave the way for more efficient message processing. Efficiency can mean many things, and in this sense, I am talking about the ease of implementing more advanced messaging behaviors, as opposed to the speed with which a message can be created. Just as writing an address on an envelope takes time, serializing an address into a message takes time. However, just as writing an address on an envelope improves postal efficiency, serializing an address into a message improves processing efficiency, especially when more advanced messaging behaviors are implemented (like message routers and intermediaries).

Specifying the Ultimate Receiver

So what sorts of items should we place in an address? For starters, an address should identify the ultimate receiver we want to send a message to. Since the ultimate receiver might be hosting multiple services, we should also have a way to uniquely identify the specific service on the ultimate receiver. It's possible that one address element might be able to describe both the ultimate receiver hosting the service and the service itself. Take the following example:

```
http://wintellect.com/OrderService
```

In the age of the Internet, we have come to understand that this address includes both the location of the ultimate receiver and a protocol that we can use to access it. Since most SO messages are SOAP messages, we need some SOAP construct that will convey the same information.

We have learned already that SOAP messages can contain three types of elements: an envelope, one header with multiple header blocks, and a body. The envelope isn't a good choice, since the envelope can occur only once. That leaves the header blocks and the body as the only two remaining candidates. So what about the body? From our earlier discussion, we know that the body is intended for use only by the ultimate receiver. By process of elimination, we see that the only logical place for us to put an address is in the header of a message. So what should this header block look like? How about:

```
<Envelope>
  <Header>
    <To>http://wintellect.com/OrderService</To>
  </Header>
  <Body> …</Body>
</Envelope>
```

At a high level, this simple XML structure accomplishes our goal of identifying the ultimate receiver and service we would like to send a message to.

Specifying the Initial Sender

It might also be useful to add sender information to the message, sort of like a return address on a letter. Adding sender information to the message serves two purposes: to indicate the sender to the ultimate receiver, and to indicate the sender to any intermediaries. We have already seen that a URL can be used to identify the target of a message. So maybe we can in fact use the same construct to identify the sender. Take the following example:

```
<Envelope>
  <Header>
    <To>http://wintellect.com/ReceiveService</To>
    <From>http://wintellect.com/SendService</From>
  </Header>
  <Body> …</Body>
</Envelope>
```

Adding this simple element to the SOAP message indicates where the message came from, and it can be used either by an intermediary or by the ultimate receiver.

Specifying Where to Send an Error

What if there's a problem processing the message? Every modern computing platform has some way to indicate errors or exceptions. These error handling mechanisms make our applications more robust, predictable, and easier to debug. It is natural to want the same mechanism in our messaging applications. Given that we already have a <*To*> and a <*From*> in our message, we could send all of our error notifications to the address specified in the <*From*> element. What if we want error notifications to go to a location specifically reserved for handling errors? In this case, we have to create yet another element:

```
<Envelope>
  <Header>
    <To>http://wintellect.com/OrderService</To>
    <From>http://wintellect.com/SendService</From>
    <Error>http://wintellect.com/ErrorService</Error>
  </Header>
  <Body> …</Body>
</Envelope>
```

Adding the <*Error*> element to the header clearly indicates where the sender would like error messages to be sent. Because this URL is in the header, it can be used by either the ultimate recipient or an intermediary.

Identifying a Message

Our simple addressing scheme requires the sender to add our *To*, *From*, and *Error* information as header blocks in the message and then send the message to the ultimate receiver. As processing occurs at an intermediary or the ultimate receiver, an error might occur. Given that we now have the error element in our message, the intermediary or ultimate receiver should be able to send us an error message. This error message will be an entirely different message from the one originally sent. From the initial sender's perspective, receiving error messages is troubling in and of itself, but it is especially troubling if we don't know the message send that caused the error. It would be great for debugging, troubleshooting, and auditing if there were a way for us to correlate the original message with the error message. To do this, we need two separate elements in our message: a message identifier element, and a message correlation element. Let's look at the message identifier first:

```
<Envelope>
  <Header>
    <MessageID>15d03fa4-1b99-4110-a5e2-5e99887dea23</MessageID>
    <To>http://wintellect.com/OrderService</To>
    <From>http://wintellect.com/SendService</From>
    <Error>http://wintellect.com/ErrorService</Error>
  </Header>
  <Body>...</Body>
</Envelope>
```

In this example, we have called our message identifier element <*MessageID*>. For now, we can think of *MessageID*'s value as a globally unique number. Upon generation, this number does not mean anything to other participants. If the initial sender generates a message as described earlier, all intermediaries and the ultimate receiver know where to send an error message, but they can also use *MessageID* to reference the particular message that caused the error. If the ultimate receiver for error messages and the sender are different, these processes must exchange information between themselves to fully understand the message send that caused the error.

Relating Messages to Other Messages

If we assume that either an intermediary or the ultimate receiver has encountered a problem processing a message, it follows that a new message should be sent to the address specified in the error element. If an intermediary or an ultimate receiver sends an entirely new message, the intermediary or the ultimate receiver becomes the sender of the new message. Likewise, the address specified in the original *Error* header block now becomes the ultimate receiver of the new message. We just established that the initial message that caused the error will contain a *MessageID* element. Somehow, the error message needs to contain a reference to this *MessageID* element. The correlation between the original message and the error message can be described by using a *RelatesTo* element:

```
<Envelope>
  <Header>
    <MessageID>66bc85ab-9799-433c-b338-3d718e491dc2</MessageID>
    <RelatesTo>15d03fa4-1b99-4110-a5e2-5e99887dea23</RelatesTo>
    <To>http://www.wintelelct.com/ErrorService</To>
    <From>http://wintellect.com/OrderService</From>
    <Error>http://wintellect.com/ErrorService</Error>
  </Header>
  <Body> …</Body>
</Envelope>
```

The error service at *http://wintellect.com/ErrorService* is the ultimate recipient of this message. When this error service reads the message, information about the message that caused the error is available in the *RelatesTo* element. Although the error service might not do anything with the *RelatesTo* information, it can be used for debugging, troubleshooting, and auditing. Notice also in this example that the *To*, *From*, and *Error* elements have all changed to reflect the new context of the message.

Who Is Listening for a Response?

Let's step away from error messages for a bit and go back to the initial message. As you've seen, we have a way to specify the ultimate receiver, the address of the initial sender, a unique identifier for the message, and where error notifications should be sent. It is possible that we want a way to specify a reply address while still specifying the address of the initial sender. Examples of this behavior abound in the real world. For example, invoices commonly have

a "Send further correspondence here" address that is different from the initial sending address. Our SO messages need a similar construct. We can once again use the notion of an address combined with a new element to describe this information. We will call this new element *ReplyTo* in the following example:

```
<Envelope>
  <Header>
    <MessageID>e563751c-3ed0-40b9-a6da-0cc9d3b34396</MessageID>
    <To>http://wintellect.com/OrderService</To>
    <ReplyTo>http://wintellect.com/OrderReplyService</ReplyTo>
    <From>http://wintellect.com/SendService</From>
    <Error>http://wintellect.com/ErrorService</Error>
  </Header>
  <Body> …</Body>
</Envelope>
```

It might seem repetitive to have both a *From* and a *ReplyTo* element in the same message. It's important to remember, however, that *From* and *ReplyTo* might be describing exactly the same service, but they can also describe two different services. Adding a *ReplyTo* element simply adds more flexibility and functionality to the set of header blocks we are creating.

Specifying an Operation

This next header block will require a little context, especially if you don't have much experience dealing with Web services. Once again, I would like to step into a real-world example first. We all know that postal addresses can contain an ATTN line. Typically, this line is used to route the parcel to a particular person, department, or operation. Take a look at the following postal address:

Contoso Boomerang Corporation ATTN: New Customer Subscriptions 2611 Boomerang Way Atlanta, GA 30309

From experience, we know that this address refers to Contoso Boomerang Corporation. More precisely, we know that the address specifically refers to the New Customer Subscriptions group within Contoso Boomerang Corporation.

If you expect to send mail to a large company, you may not have to specifically address a particular department. You could send mail to Contoso Boomerang Corporation and expect someone to ultimately open the mail, make a decision about who should receive the mail, and route the mail to the inferred recipient. Clearly this process will take longer than if we specifically addressed the message to the correct department or group.

Contoso Boomerang Corporation might have several groups that can receive mail. Each group might have its own set of actions to perform. For example, Contoso might have one group responsible for signing up new customers, another group responsible for customer support, and yet another group for new product development. At an abstract level, addresses can specify different levels of granularity for the destination, and each destination might have its own set of tasks or actions to perform.

So far, we have created elements that define the ultimate receiver, a reply-to receiver, an error notification receiver, a message identifier, a message correlation mechanism, and the initial sender. We have not, however, defined a way to indicate an action or operation for the message. Let's assume, for now, that we can use another header element containing a URL as a way to identify an action or operation. The following example illustrates this assumption with the addition of a new header:

```
<Envelope>
  <Header>
    <MessageID>ca9b172b-9f67-49af-9abd-7fa4b3a63c10</MessageID>
    <To>http://wintellect.com/OrderService</To>
    <Action>urn:ProcessOrder/Action>
    <ReplyTo>http://wintellect.com/OrderReplyService</ReplyTo>
    <From>http://wintellect.com/SendService</From>
    <Error>http://wintellect.com/ErrorService</Error>
  </Header>
  <Body> …</Body>
</Envelope>
```

In this example, the *Action* element states that the *ProcessMsg* operation should be performed on this message. It is possible that *OrderService* defines additional operations. For example, we can send another message to the archive message operation by using the following *Action* element:

```
<Envelope>
  <Header>
    <MessageID>6d73f358-cf18-4e3b-8b28-9871c8a21cda</MessageID>
    <To>http://wintellect.com/OrderService</To>
    <Action>urn:ArchiveMessage</Action>
    <ReplyTo>http://someotherurl.com/OrderReplyService</ReplyTo>
    <From>http://wintellect.com/SendService</From>
    <Error>http://wintellect.com/ErrorService</Error>
  </Header>
  <Body> ... </Body>
</Envelope>
```

The Need for Standard Header Blocks

We have just arbitrarily defined seven elements that help us address messages. By no means can we assume that our element names will be universally adopted. We could, however, build our own infrastructure that understands these elements and use this infrastructure in each of our messaging participants. In other words, we can't send these messages to an application that does not understand what our seven message headers mean. Likewise, our application could not receive messages that contained different addressing headers. For example, another application vendor could have defined message headers like the following:

```
<Envelope>
  <Header>
    <MessageIdentifier>1</MessageIdentifier>
    <SendTo>http://wintellect.com/OrderService</SendTo>
```

```
    <Op>http://wintellect.com/OrderService/ArchiveMessage</Op>
    <Reply>http://someotherurl.com/OrderReplyService</Reply>
    <SentFrom>http://wintellect.com/SendService</SentFrom>
    <OnError>http://wintellect.com/ErrorService</OnError>
  </Header>
  <Body>...</Body>
</Envelope>
```

Applications that contain our infrastructure cannot process this message.

If we were to take a survey of most enterprise applications, we would see that software vendors have followed this exact model in defining their own messages. For several years, SOAP has been the agreed-upon message format, but there was no agreement on the header blocks that could appear in a message, and as a result, applications could not easily interoperate. True SOAP message interoperability requires a set of header blocks that are common across all software vendors. As mentioned in Chapter 1, the WS-* specifications go a long way toward solving this problem by defining a common set of messaging headers.

WS-Addressing

WS-Addressing is one of the WS-* specifications that has been widely embraced by the software vendor community. It provides a framework for one of the most fundamental tasks of any service-oriented application—indicating the target of a message. To this end, all other WS-* specifications (for example, WS-ReliableMessaging, WS-Security, WS-AtomicTransaction, and so on) build on WS-Addressing. The full WS-Addressing specification is available at *http://www.w3.org/TR/ws-addr-core/*.

This specification defines two constructs that are normally found in the transport layer. The purpose of these constructs is to convey addressing information in a transport-neutral manner. These two constructs are *endpoint references* and *message addressing properties*.

Endpoint References

So far, we have used the terms *initial sender*, *intermediary*, and *ultimate receiver* to describe the entities participating in a message exchange. These participants can also be considered *service endpoints*. Simply defined, a service endpoint is a resource that can be the target of a message. Endpoint references, as defined in the WS-Addressing specification, are a way to refer to a service endpoint.

Can't we just use a URL to identify the target of a message? URLs will work in some cases, but not all. URLs are not well suited for expressing certain types of references. For example, many services will create multiple server object instances, and we might want to send a message to a particular instance of the server object. In this case, a simple URL just won't do. Based on our experience with the Internet, we might assume that we could add parameters to the address, thereby associating our message with a specific set of server objects. This introduces a few problems. For example, adding parameters to a URL will tightly bind our message to a

transport, and we might not know the specific parameters until after we have initiated contact with the server (as is the case with Amazon's session IDs).

A Legitimate Debate

It's reasonable to ask the question "Why do we need more than a URL to refer to a service endpoint?" In fact, this is a really good question, and one that is actively debated in distributed architecture communities today. On one side of the discussion is a community that says a service should be referenceable via a URL. Furthermore, it is even possible to reference a specific instance of a service through a URL, as this is commonly done on the Internet today. All you have to do to prove this point is take a look at the URL generated as you purchase something at Amazon.com. You'll notice that after you sign in, your URL changes to contain a unique session ID. That session ID is tracked on the Amazon server and associated with you and your shopping cart. The people on this side of the debate see no reason to ever venture outside of describing a service with the URL, and they use the global adoption of the Internet as evidence of the viability of their position. Representational State Transfer (REST) is an architectural style that embraces this mode of thought. WCF can be used in the REST architectures.

On the other side of the debate is a group that says that HTTP URLs and the PUT/DELETE/GET/POST HTTP commands are not sufficient for all services. If we take another look at the Amazon example, several things are implicit. For example:

- HTTP is always the right transport.
- Security is provided via the transport (HTTPS).
- We need to secure only the message transmission (from client to Web server).
- It is OK to make a request for session-specific parameters.

The people on this side of the debate claim that these limitations are not acceptable for all services and distributed applications. In their opinion, service orientation demands transport independence and security outside the transport. Those who agree with SOAP and the WS-* specifications embrace this side of the debate.

In my view, there is room for both architectural styles, and each has its place. There is no question that the architecture of Amazon.com is wildly successful for publicly available services, but for back-end processing, I do not think that the implicit limitations in a REST architecture will work in all circumstances. The big limitations I see with the REST architectural style are dependence on a single transport, a lack of message-based security, and a lack of transactional support.

Clearly, WCF can be used in SOAP/WS-* implementations, and most of this book is dedicated to describing these concepts. In future releases of WCF, there will be more support for REST architectures.

URI, URL, and URN

The terms URI, URL, and URN (Uniform Resource Indicator, Uniform Resource Locator, and Uniform Resource Name) are used frequently in the WS-* specifications. To comprehend the full impact of what the WS-* specifications reference, we must understand the subtle differences between these three terms. In general, URI, URL, and URN are ways to name and/or locate a resource. If we were to think of the information world as an information space, a URI is a string that one can use to locate or name a point in that space. A URL, as opposed to a URI, is strictly intended to locate a resource. A URN, as opposed to a URL, is strictly intended to name a resource. From a set perspective, the URL and URN sets are members of the greater URI set.

These logical properties are physically implemented as XML Infoset element information items. Some properties, like *Reference Properties*, *Reference Parameters*, and *Policy*, can wrap other XML element information items. Here's how these properties can be represented in XML:

```
<wsa:EndpointReference xmlns:wsa="http://schemas.xmlSOAP...">
  <wsa:Address> ... </wsa:Address>
  <wsa:ReferenceProperties> ... </wsa:ReferenceProperties>
  <wsa:ReferenceParameters> ... </wsa:ReferenceParameters>
  <wsa:PortType> ... </wsa:PortType>
  <wsa:ServiceName> ... </wsa:ServiceName>
  <wsp:Policy> ... </wsp:Policy>
</wsa:EndpointReference>
```

Message Information Headers

WS-Addressing also defines a set of standard SOAP headers that can be used to fully address a message. As you might expect, these headers are actually XML Infoset element information items that represent the same functionality we derived in the section "Addressing the Message" earlier in this chapter. The real benefit seen here is a standard set of headers whose function can be commonly agreed upon between application vendors.

The following code snippet contains message information headers and their data types as defined in the WS-Addressing specification. These headers should look quite familiar:

```
<wsa:MessageID> xs:anyURI </wsa:MessageID>
<wsa:RelatesTo RelationshipType="…"?> xs:anyURI </wsa:RelatesTo>
<wsa:To> xs:anyURI </wsa:To>
<wsa:Action> xs:anyURI </wsa:Action>
<wsa:From> endpoint-reference </wsa:From>
<wsa:ReplyTo> endpoint-reference </wsa:ReplyTo>
<wsa:FaultTo> endpoint-reference </wsa:FaultTo>
```

Notice that the *MessageID*, *RelatesTo*, *To*, and *Action* elements are of type *xs:anyURI*. Why is *To* of type *xs:anyURI* instead of an endpoint reference? After all, we just went through great pains describing the reasons a simple URI is not enough to address a message. The answer lies in how additional properties that would normally be in an endpoint reference are serialized into a message header. WS-Addressing defines a default way to represent an endpoint reference that happens to be the target of a message as follows.

If a message is going to be sent to the endpoint reference as described here:

```
<wsa:EndpointReference xmlns:wsa="..." xmlns:wnt="...">
  <wsa:Address>http://wintellect.com/OrderService</wsa:Address>
  <wsa:ReferenceProperties>
    <wnt:OrderID>9876543</wnt:OrderID>
  </wsa:ReferenceProperties>
  <wsa:ReferenceParameters>
    <wnt:ShoppingCart>123456</wnt:ShoppingCart>
  </wsa:ReferenceParameters>
</wsa:EndpointReference>
```

That endpoint reference can be serialized in a message as follows:

```
<S:Envelope xmlns:S="..." xmlns:wsa="..." xmlns:wnt="... ">
  <S:Header>
    ...
    <wsa:To>http://wintellect.com/RcvService</wsa:To>
    <wnt:OrderID>9876543</wnt:OrderID>
    <wnt:ShoppingCart>123456</wnt:ShoppingCart>
     ...
  </S:Header>
  <S:Body>
    ...
  </S:Body>
</S:Envelope>
```

Notice that the *ReferenceProperty* and *ReferenceParameter* elements for *To* were promoted to full-fledged headers, no longer subordinate to the *EndpointReference* element. This happens only for the *To* element, as the *From*, *FaultTo*, and *ReplyTo* elements are endpoint references.

Message Information Header Block Dependencies

As you might expect, certain message information header blocks depend on other message information header blocks. For example, if a *ReplyTo* header block is present, it would stand to reason that a *MessageID* header must also be present. Table 2-2 describes the dependencies of the standard message information headers.

Table 2-2 Message Information Header Dependencies

Header #	Header Name	Min Occurs	Max Occurs	Depends On
1	*wsa:MessageID*	0	1	N/A
2	*wsa:RelatesTo*	0	Unbounded	N/A
3	*wsa:ReplyTo*	0	1	1
4	*wsa:From*	0	1	N/A
5	*wsa:FaultTo*	0	1	1
6	*wsa:To*	1	1	N/A
7	*wsa:Action*	1	1	N/A

The Four Tenets of Service Orientation

So far, we have explored the concept of service orientation, looked at the structure of service-oriented messages, examined the requirements for message addresses, and discussed the industry standard for message addressing. If you understand the motivation for a standard addressing structure in an SO message, then it is not much of a stretch to understand the principles of service orientation. Every service-oriented design adheres to the following four principles (often called the four tenets).

Explicit Boundaries

In service orientation, services can interact with each other by using messages. To put it another way, services can send messages across their service boundary to other services. Services can send and receive messages, and the shapes of the messages that can be sent or received define the service boundaries. These boundaries are well defined, clearly stated, and the only accessible point for the service's functionality. More practically, if Service1 wants to interact with Service2, Service1 must send a message to Service2. In contrast, an object-oriented or component-oriented world would demand that Service1 should create an instance of Service2 (or a proxy referring to Service2). In this case, the boundary between these services is blurred, since Service1 is, for all intents and purposes, in control of Service2.

If Service1 sends a message to Service2, does it matter where Service2 is located? The answer is no, as long as Service1 is allowed to send the message to Service2. One must assume, however, that sending a message across a boundary comes with a cost. This cost must be taken into consideration when building services. Specifically, our services should cross service boundaries as few times as possible. The antithesis of an efficient service design is one that is "chatty."

Service Autonomy (Sort Of)

In my opinion, service-oriented systems should strive to be sort of autonomous, because pure autonomy is impossible. True service autonomy means that a service has no dependencies on anything outside itself. In the physical world, these types of entities are nonexistent, and I doubt we will see many pure autonomous services in the distributed computing world. A truly autonomous service is one that will dynamically build communication channels, dynamically negotiate security policy, dynamically interrogate message schemas, and dynamically exchange messages with other services. A purely autonomous service reeks of an overly late-bound architecture. We have all seen these sorts of systems, whether in the excessive use of *IUnknown* or the compulsive use of reflection. The bottom line is that developers and architects have proven time after time that these types of architectures just do not work (even though they look great on paper). I must temper these comments by admitting that movement in the area of service orientation is picking up at a blinding pace. Just five years ago, service-oriented applications were few and far between, and now they are commonplace. This momentum may take us to a place where purely autonomous services are the way to go, but for now, I think it is reasonable to settle for a diluted view of autonomy.

So what does autonomy mean in a practical sense? From a practical perspective, it means that no service has control of the lifetime, availability, or boundaries of another service. The opposite of this behavior is exhibited with the SQL 2000 database and agent services. Both of these services are hosted as separate Microsoft Windows services, but the agent service has a built-in dependency on the database service. Stopping the database service means that the agent service will be stopped as well. The tight coupling between these two services means that they can never be considered as separate, or versioned independently of each other. This tight coupling reduces the flexibility of each service, and thereby their use in the enterprise.

Contract Sharing

Since service orientation focuses on the messages that are passed between participants, there must be a way to describe those messages and what is required for a successful message exchange. In a broad sense, these descriptions are called *contracts*. Contracts are not a new programming paradigm. On the Windows platform, contracts came into their own with COM and DCOM. A COM component can be accessed only through a published and shared contract. Physically, a COM contract is an interface, expressed in Interface Definition Language (IDL). This contract shields the consumer from knowing implementation details. As long as the contract doesn't break, the consumer can theoretically tolerate COM component software upgrades and updates.

Service-oriented systems conceptually extend the notion of COM IDL contracts. Service-oriented systems express contracts in the widely understood languages of XSD and WSDL. More specifically, schemas are used to describe message structures, and WSDL is used to describe message endpoints. Together, these XML-based contracts express the shape of the messages that can be sent and received, endpoint addresses, network protocols, security

requirements, and so on. The universal nature of XML allows senders and ultimate recipients to run on any platform more easily than with a technology like COM. Among other things, a sender must know the message structure and format of the receiving application, and this is answered by the contract. In essence, a message sender requires a dependency on the contract, rather than the service itself.

Compatibility Based on Policy

Services must be able to describe the circumstances under which other services can interact with it. For example, some services might require that any initial sender possess a valid Active Directory directory service account or an X509 certificate. In this case, the service should express these requirements in an XML-based policy. At the time of this writing, WS-Policy is the standard grammar for expressing these types of requirements. In a fanatically devoted service-oriented world, message senders would interrogate this metadata prior to sending a message, further decoupling a message sender from a message receiver. For the same reasons stated earlier, it is more probable that service policy will be interrogated at design time more than at run time.

Putting It All Together

I hope that by this point in the chapter you have a clear conceptual view of service orientation. For the next few pages, let's look at how this concept can physically take shape in WCF applications. In our example, we will be building a simple order processing service that receives customer orders. To keep things simple, there are two message participants, as shown in Figure 2-3.

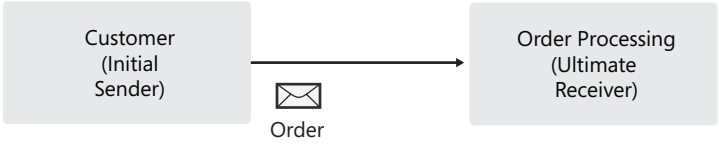

Figure 2-3 A simple message exchange

The purpose of these code samples is to solidify your vision of service orientation and provide an introduction to WCF, not to detail every aspect of WCF or to build a fully functional order processing system. The types and mechanisms introduced in these examples will be detailed throughout this book.

The Contract

Typically, the place to start in a service-oriented application is to create the contract. To keep our example simple, an order will contain a product ID, a quantity, and a status message. Given these three fields, an order could be represented with the following pseudo-schema:

```
<Order>
  <ProdID>xs:integer</ProdID>
  <Qty>xs:integer</Qty>
  <Status>xs:string</Status>
</Order>
```

From our message anatomy and addressing discussions, we know that messages need more addressing structure if they are going to use WS-Addressing. In our order processing service, both the sender and the receiver agree to use SOAP messages that adhere to the WS-Addressing specification to dictate the structure of the message. Given these rules, the following is an example of a properly structured message:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:wsa="http://
schemas.xmlsoap.org/ws/2004/08/addressing">
  <s:Header>
    <wsa:Action s:mustUnderstand="1">urn:SubmitOrder</wsa:Action>
    <wsa:MessageID>4</wsa:MessageID>
    <wsa:ReplyTo>
      <wsa:Address> http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous
      </wsa:Address>
    </wsa:ReplyTo>
    <wsa:To s:mustUnderstand="1">http://localhost:8000/Order</wsa:To>
  </s:Header>
  <s:Body>
    <Order>
      <ProdID>6</ProdID>
      <Qty>6</Qty>
      <Status>order placed</Status>
    </Order>
  </s:Body>
</s:Envelope>
```

After we have created the schemas that describe our messages, our next step is to define the endpoint that will receive those messages. For this, we can turn to WSDL. You might be thinking to yourself: "I am not really in the mood to deal with raw schemas or WSDL." Well, you are not alone. The WCF team has provided a way for us to express a contract (both the schema and the WSDL) in the Microsoft .NET Framework language of our choosing (in this book, it will be C#). Basically, the expression of a contract in C# can be turned into XSD-based and WSDL-based contracts on demand.

When choosing to express our contracts in C#, we can choose to define a class or an interface. An example of a contract defined as an interface in C# is shown here:

```
// file: Contracts.cs
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;

// define the contract for the service
[ServiceContract(Namespace = "http://wintellect.com/ProcessOrder")]
public interface IProcessOrder {
    [OperationContract(Action="urn:SubmitOrder")]
    void SubmitOrder(Message order);
}
```

Notice the *ServiceContractAttribute* and *OperationContractAttribute* annotations. We will talk more about these attributes in Chapter 9, "Contracts." For now, it is enough to know that this interface is distinguished from other .NET Framework interfaces through the addition of these custom attributes. Also notice the signature of the *SubmitOrder* interface method. The only parameter in this method is of type *System.ServiceModel.Message*. This parameter represents any message that can be sent to a service from an initial sender or intermediary. The *Message* type is a very interesting and somewhat complex type that will be discussed thoroughly in Chapter 5, "Messages," but for now, assume that the message sent by the initial sender can be represented by the *System.ServiceModel.Message* type.

Regardless of the way we choose to express our contracts, it should be agreed upon and shared before further work is done on either the sender or the receiver applications. In practice, the receiver defines the required message structure contract, and the sender normally attempts to build and send messages that adhere to this contract.

There is nothing preventing the sender from sending messages that do not adhere to the contract defined by the receiver. For this reason, the receiver's first task should be to validate received messages against the contract. This approach helps ensure that the receiver's data structures do not become corrupted. These points are frequently debated in distributed development communities, so there are other opinions on this matter.

This contract can now be compiled into an assembly. Once the compilation is complete, the assembly can be distributed to the sender and the receiver. This assembly represents the contract between the sender and the receiver. While there will certainly be times when the contract will change, we should consider the contract immutable after it has been shared. We will discuss contract versioning in Chapter 9.

Now that we have our contract in place, let's build the receiver application. The first order of business is to build a class that implements the interface defined in our contract:

```
// File: Receiver.cs

// Implement the interface defined in the contract assembly
public sealed class MyService : IProcessOrder {

  public void SubmitOrder(Message order) {
    // Do work here
  }
}
```

Because this is a simple application, we are content to print text to the console and write the inbound message to a file:

```
// File: Receiver.cs
using System;
using System.Xml;
using System.IO;
using System.ServiceModel;
using System.ServiceModel.Channels;


// Implement the interface defined in the contract assembly
public sealed class MyService : IProcessOrder {

  public void SubmitOrder(Message order) {
    // Create a file name from the MessageID
    String fileName = "Order" + order.Headers.MessageId.ToString() + ".xml";

    // Signal that a message has arrived
    Console.WriteLine("Message ID {0} received",
      order.Headers.MessageId.ToString());

    // create an XmlDictionaryWriter to write to a file
    XmlDictionaryWriter writer = XmlDictionaryWriter.CreateTextWriter(
      new FileStream(fileName, FileMode.Create));

    // write the message to a file
    order.WriteMessage(writer);

    writer.Close();
  }
}
```

Our next task is to allow the *MyService* type to receive inbound messages. To receive a message:

- *MyService* must be loaded into an AppDomain.
- *MyService* (or another type) must be listening for inbound messages.

- An instance of this type must be created at the appropriate time and referenced as long as it is needed (to prevent the garbage collector from releasing the object's memory).
- When a message arrives, it must be dispatched to a *MyService* instance and the *SubmitOrder* method invoked.

These tasks are commonly performed via a *host*. We will talk more about hosts in Chapter 10, but for now, assume that our AppDomain is hosted in a console application and the type responsible for managing the lifetime of and dispatching messages to *MyService* objects is the *System.ServiceModel.ServiceHost* type. Our console application is shown here:

```
// File: ReceiverHost.cs

using System;
using System.Xml;
using System.ServiceModel;

internal static class ReceiverHost {
  public static void Main() {
    // Define the binding for the service
    WSHttpBinding binding = new WSHttpBinding(SecurityMode.None);
    // Use the text encoder
    binding.MessageEncoding = WSMessageEncoding.Text;

    // Define the address for the service
    Uri addressURI = new Uri(@"http://localhost:4000/Order");

    // Instantiate a Service host using the MyService type
    ServiceHost svc = new ServiceHost(typeof(MyService));

    // Add an endpoint to the service with the
    // contract, binding, and address
    svc.AddServiceEndpoint(typeof(IProcessOrder),
                           binding,
                           addressURI);

    // Open the service host to start listening
    svc.Open();

    Console.WriteLine("The receiver is ready");
    Console.ReadLine();

    svc.Close();
  }
}
```

In our console application, we must set some properties of the service before we can host it. As you will see in subsequent chapters, every service contains an *address*, a *binding*, and a *contract*. These mechanisms are often called the ABCs of WCF. For now, assume the following:

- An address describes where the service will be listening for inbound messages.
- A binding describes how the service will be listening for messages.
- A contract describes what sorts of messages the service will receive.

In our example, we are using the *WSHttpBinding* binding to define how the service will listen for inbound messages. We'll talk more about bindings in Chapter 8. Our service also uses the *Uri* type to define the address our service will be listening on. Our service then instantiates a *ServiceHost* object that uses our *MyService* class to provide shape to the *ServiceHost*. *ServiceHost*s do not have default endpoints, so we must add our own by calling the *AddServiceEndpoint* instance method. It is at this point that our console application is ready to start listening at the address *http://localhost:8000/Order* for inbound messages. A call to the *Open* instance method begins the listening loop (among other things).

You might be wondering what happens when a message arrives at *http://localhost:8000/Order*. The answer depends on what sort of message arrives at the endpoint. For that, let's switch gears and build our simple message sending console application. At a high level, our message sender is going to have to know the following:

- Where the service is located (the address)
- How the service expects messages to be sent (the service binding)
- What types of messages the service expects (the contract)

Assuming that these facts are known, the following is a reasonable message sending application:

```
// File: Sender.cs

using System;
using System.Text;
using System.Xml;
using System.ServiceModel;
using System.Runtime.Serialization;
using System.IO;
using System.ServiceModel.Channels;

public static class Sender {

  public static void Main(){
    Console.WriteLine("Press ENTER when the receiver is ready");
    Console.ReadLine();

    // address of the receiving application
    EndpointAddress address =
      new EndpointAddress(@"http://localhost:4000/Order");

    // Define how we will communicate with the service
    // In this case, use the WS-* compliant HTTP binding
    WSHttpBinding binding = new WSHttpBinding(SecurityMode.None);
    binding.MessageEncoding = WSMessageEncoding.Text;

    // Create a channel
    ChannelFactory<IProcessOrder> channel =
      new ChannelFactory<IProcessOrder>(binding, address);
```

```
    // Use the channel factory to create a proxy
    IProcessOrder proxy = channel.CreateChannel();

    // Create some messages
    Message msg = null;
    for (Int32 i = 0; i < 10; i++) {
      // Call our helper method to create the message
      // notice the use of the Action defined in
      // the IProcessOrder contract...
      msg = GenerateMessage(i,i);

      // Give the message a MessageID SOAP header
      UniqueId uniqueId = new UniqueId(i.ToString());
      msg.Headers.MessageId = uniqueId;

      Console.WriteLine("Sending Message # {0}", uniqueId.ToString());

      // Give the message an Action SOAP header
      msg.Headers.Action = "urn:SubmitOrder";
      // Send the message
      proxy.SubmitOrder(msg);
    }
  }

  // method for creating a Message
  private static Message GenerateMessage(Int32 productID, Int32 qty) {

    MemoryStream stream = new MemoryStream();

    XmlDictionaryWriter writer = XmlDictionaryWriter.CreateTextWriter(
      stream, Encoding.UTF8, false);

    writer.WriteStartElement("Order");
    writer.WriteElementString("ProdID", productID.ToString());
    writer.WriteElementString("Qty", qty.ToString());
    writer.WriteEndElement();

    writer.Flush();
    stream.Position = 0;

    XmlDictionaryReader reader = XmlDictionaryReader.CreateTextReader(
      stream, XmlDictionaryReaderQuotas.Max);

    // Create the message with the Action and the body
    return Message.CreateMessage(MessageVersion.Soap12WSAddressing10,
                                 String.Empty,
                                 reader);
  }
}
```

Try not to get too distracted by the *ChannelFactory* type just yet—we will fully explore this type in Chapter 4. For now, notice the code in the *for* loop. The instructions in the loop generate 10 messages and assign each one a pseudo-unique ID and an action.

At this point, we should have two executables (ReceiverHost.exe and Sender.exe) representing an ultimate receiver and an initial sender. If we run both console applications, wait for the receiver to initialize, and press ENTER on the initial sender application, we should see the following on the receiver:

```
The receiver is ready
Message ID 0 received
Message ID 1 received
Message ID 2 received
Message ID 3 received
Message ID 4 received
Message ID 5 received
Message ID 6 received
Message ID 7 received
Message ID 8 received
Message ID 9 received
```

Congratulations! You have just written a service-oriented application with WCF. Remember that the service is writing inbound messages to a file. If we examine one of the files that our service wrote, we see the following:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">urn:SubmitOrder</a:Action>
    <a:MessageID>1</a:MessageID>
    <a:ReplyTo>
      <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
    </a:ReplyTo>
    <a:To s:mustUnderstand="1">http://localhost:4000/Order</a:To>
  </s:Header>
  <s:Body>
    <Order>
      <ProdID>1</ProdID>
      <Qty>1</Qty>
    </Order>
  </s:Body>
</s:Envelope>
```

The headers in this message should look eerily similar to the ones we see in the WS-Addressing specification, and their values should look like the properties we set in our message sending application. In fact, the *System.ServiceModel.Message* type exposes a property named *Headers* that is of type *System.ServiceModel.MessageHeaders*. This *MessageHeaders* type exposes other properties that represent the WS-Addressing message headers. The idea here is that we can use the WCF object-oriented programming model to affect a service-oriented SOAP message.

Why SO Makes Sense

Developers and architects often ask me, "Why do I need service orientation?" My response is simple: scalability, maintainability, interoperability, and flexibility. In the past, distributed component technologies like DCOM tightly bound distributed components together. At the bare minimum, these distributed components had to share a common type system and often a common runtime. Given these dependencies, upgrades and software updates can become complex, time-consuming, and expensive endeavors. Service-oriented applications, in contrast, do not engender the same sorts of dependencies and therefore exhibit behaviors that better address enterprise computing needs.

Versioning

Application requirements change over time. It has been this way since the dawn of computing, and there are no signs of this behavior slowing down in the future. Developers, architects, and project managers have gone to great lengths to apply processes to software development in hopes of regulating and controlling the amount and pace of change an application endures. Over the lifetime of an application, however, some of the assumptions made during the development process will certainly turn out to be invalid. In some cases, the resultant application changes will cause a cascading series of changes in other parts of the application. Autonomous, explicitly bounded, contract-based service-oriented applications provide several layers of encapsulation that buffer the effects of versioning one part of a system. In a service-oriented application, the only agreement between the message sender and the receiver is the contract. Both the sender and the receiver are free to change their implementations as they wish, as long as the contract remains intact. While this was also true of component architectures, the universal nature of service-oriented contracts further decouples the sender and receiver from implementation, thereby making the upgrade and version cycle shorter. Service orientation does not, however, remove the need for a good versioning process.

Load Balancing

Every application has bottlenecks, and sometimes these bottlenecks can prevent an application from scaling to evolving throughput demands. Figure 2-4 shows an order processing Web site built with components.

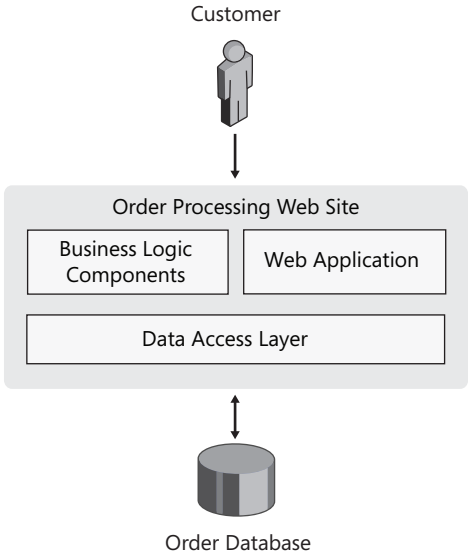

Figure 2-4 A traditional component-oriented application

In this scenario, data retrieval might be the bottleneck. If that is the case, one way to scale the component-driven Web site is shown in Figure 2-5.

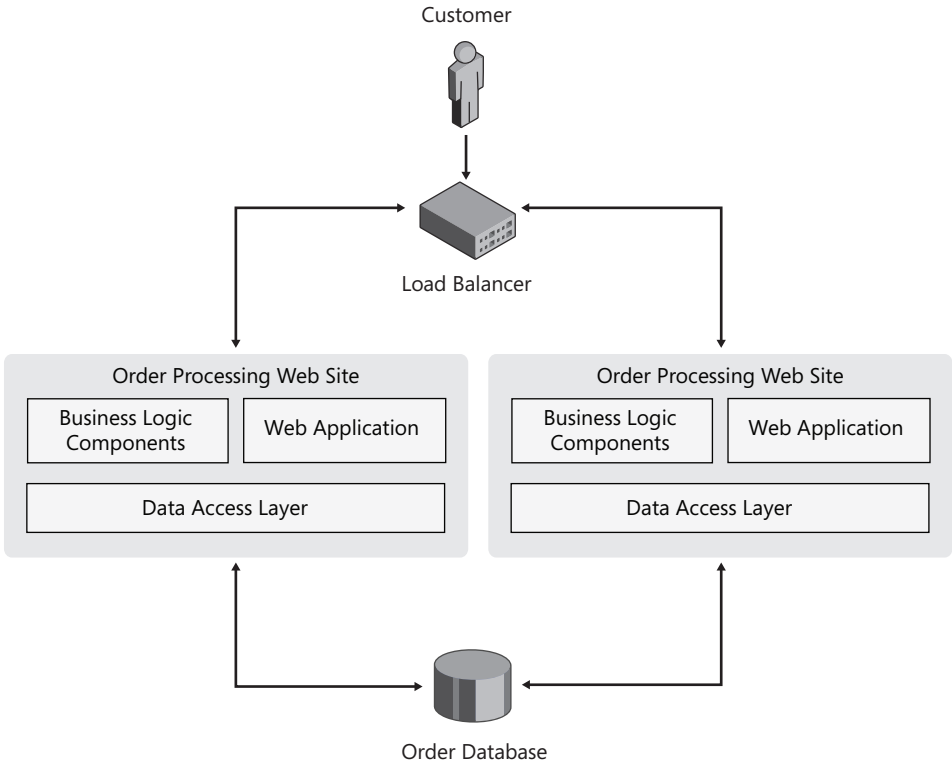

Figure 2-5 Scaling a component-oriented application

Essentially, we re-create the entire Web application on another server and use a load balancer to redirect requests to the least busy Web server. This type of scalability has proven effective in the past, but it is inefficient and costly, and creates configuration problems, especially during versioning time.

A service-oriented way to scale the order processing system in the Figure 2-5 example is shown in Figure 2-6.

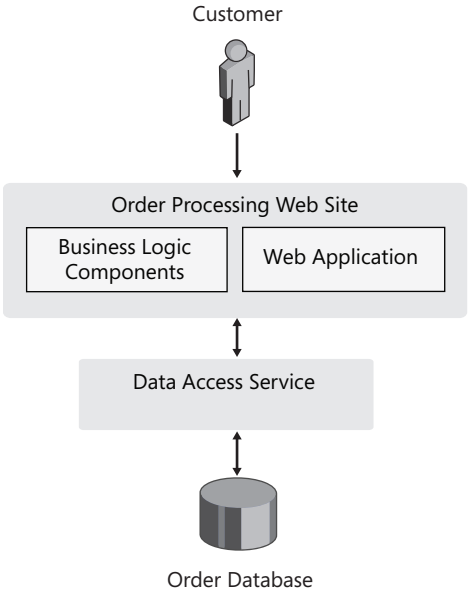

Figure 2-6 Using services

Service-oriented applications can more easily scale the parts of the application that need to be scaled. This reduces total cost of ownership and simplifies configuration management.

Platform Changes over Time

Platforms change, sometimes dramatically, over time. This is true within any platform vendor, as patches and service packs, and ultimately new versions of a platform, are constantly being released. With distributed components, there is often a dependency on a platform component runtime. For example, how does an application architect know that a DCOM component will behave the same on servers running Microsoft Windows Server 2000, Windows Professional 2000, Windows XP, or Windows Server 2003? Since a DCOM component relies on the component runtimes on each of these systems, many testing scenarios appear seemingly out of thin air. When you start to think about testing within each possible configuration, service pack, and hotfix, your nose might bleed from anxiety.

Many of these problems disappear when applications become service oriented. This is largely due to the fact that messaging contracts are expressed in a platform-neutral XML grammar.

This contract language decouples the sender from the receiver. The sender's responsibility is to generate and send a message that adheres to the contract, while the receiver's responsibility is to receive and process these messages. No platform-specific information must be serialized with the message, so endpoints are free to version their platform as they want. Furthermore, testing is much simpler, since each endpoint has to test only to the explicit service boundary.

Content-Based Routing

The nature of service-oriented messages lends itself to routing scenarios that have been very difficult in the past. We can build some business rules around our order processing example for an illustration:

- Orders can be for new items or repairs to existing items.
- Orders for new items should ultimately be sent to the manufacturing system.
- Orders for repairs should be sent to the repair system.
- Both orders, however, must be sent to the accounting and scheduling systems before they are sent to their ultimate destination.

Service-oriented messaging applications are well suited for fulfilling these types of requirements. Essentially, routable information can be placed in SOAP message headers and used by any endpoint to determine a message path.

End-to-End Security

Many distributed systems secure communication at the transport level in a point-to-point manner. While the transmission event might be secure, the data transmitted might not be secure after the transmission. Log files and other auditing mechanisms often contain information that is secured when transmitted, and as a result, they are frequent targets of many security attacks. It is possible, however, using standard XML security mechanisms, to provide end-to-end security with service-oriented messages. Even if the message is persisted into a log file and later compromised, if the message was secured using one of the standard XML security mechanisms, the data in the message can be kept confidential.

Interoperability

When an initial sender sends a message to an ultimate receiver, the initial sender does not need to have a dependency on which platform the ultimate receiver is running. As you've seen with the binary message encoder, this is not *always* the case. Some message formats can introduce platform dependencies, but this is a matter of choice. In the purest sense, service-oriented applications are platform agnostic. This platform independence is a direct result of the universal nature of messaging contracts expressed in XML grammar. It is truly possible (not just theoretically) to send a message to an endpoint and have no idea what platform that

endpoint is using. This resonates with businesspeople and managers because systems do not need to be completely replaced with a homogenous set of applications on a single platform.

Summary

This chapter illustrates the motivation for service orientation, and some of the basics of a service-oriented system. Service orientation requires a focus on the messages that an application sends, receives, or processes. Service-oriented systems can take functionality previously reserved for a transport, and place it in the structure of a message (addresses, security information, relational information, etc.). Focusing on the message provides a way to remove dependencies on platforms, hardware, and runtimes. In my view, the version resiliency of a service-oriented application is the biggest win for most IT organizations, because choreographing system-wide upgrades is one of the more expensive parts of maintenance. In the next chapter, we see some of the different ways we introduce the concepts necessary to build advanced messaging applications.