PART 1

The Service Broker Programming Model

CHAPTER 1

Fundamentals of Message-Based Processing

Love is the message and the message is love.

This lyric from Arthur Baker's song "The Message is Love" has been with me over the past few years, as I've worked and consulted with many clients around the world on .NET and SQL Server. Many of my clients have implemented large projects that incorporate message technologies in various forms, including Microsoft Message Queuing (MSMQ), XML web services, Microsoft BizTalk Server, and, of course, Microsoft SQL Server 2005 Service Broker. But what does Arthur Baker's song have to do with IT technology and even with technologies that provide frameworks for message-based applications? The answer is easy: the core concept behind message systems is a message, so the message is the central point and the most important part of such a system.

In this first chapter, I'll introduce you to message-based processing, show you why messaging is suitable for scalable solutions, and discuss what problems can occur in distributed messaging architectures. Furthermore, I'll also show how Service Broker solves the messaging problems described throughout this chapter. I'll answer the following questions:

- *Why do you need messaging in your application?* You've probably written successful synchronous applications, so why do you need asynchronous messaging? I'll explain the advantages and benefits of this dramatic change in your software architectures.
- *What problems can a messaging infrastructure solve?* When you use a new technology, such as Service Broker, you might not see the underlying issues that it solves. Infrastructure services for message-based programming offer many different options that encapsulate and solve problems behind the scenes.
- *How do you achieve effective message processing?* You can take several approaches, each of which has its own advantages and disadvantages.
- *Which application architectures are suitable for message-based software systems?* With message-based architectures, you aren't restricted to only client/server or three-tier architectures; more options are available to you.
- *Why use Service Broker as your messaging technology when so many alternatives are available on the Windows platform*? For example, you have your choice between MSMQ, COM+ Queued Components, BizTalk Server, XML web services, and .NET Framework 3.0's Windows Communication Foundation (WCF).

As you can see, you can't just say to yourself or to your project manager, "Hey, there's Service Broker. Let's use it in our next project." So let's have a look at your options and start to answer the fundamental question of why software architectures need messaging.

Message Concepts

Message-based programming allows you to implement more flexible and scalable software architectures that other programming paradigms, such as classic client/server applications, cannot provide. If you want to implement scalable applications that are able to handle thousands of concurrent users, you must take a message-based approach. Otherwise, you'll struggle with performance issues and massive scalability problems. But before diving into the details, let's have a look at the central point of a messaging application—the message itself.

Message Anatomy

When you take a look at a message from a 1,000-foot level, you'll see that it's composed of three parts:

- Envelope
- Header
- Body

The message envelope is the wrapping in which the message header and the message body are transferred. The message header contains routing information, such as existing intermediaries on the route, the final destination, the sender, the priority, and so on. One important thing you should know is that the header is normally extensible—just think of a user-defined Simple Object Access Protocol (SOAP) header in a web services scenario. The body of the message stores the payload, which is transferred from the sender to the receiver. The body can be in any format, such as binary data, XML data, or even textual data (like in an email message). Figure 1-1 illustrates the structure of a message.

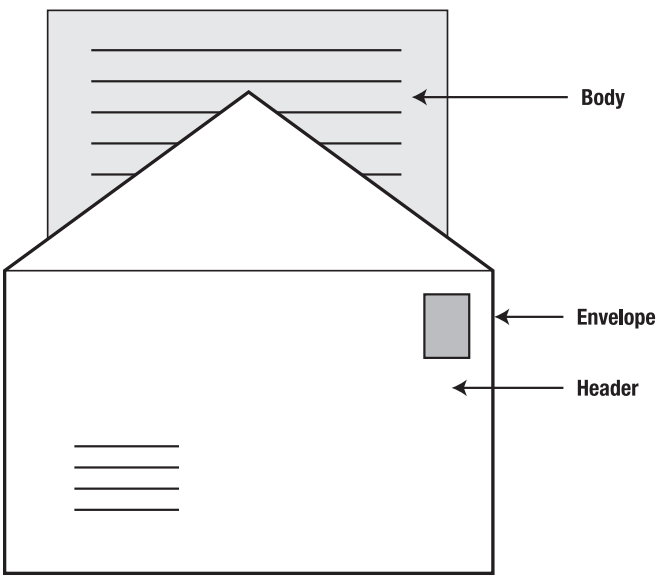

Figure 1-1. *The structure of a message*

However, messages aren't used only in software systems. Messages are used all day long throughout daily life. The messages in your life provide you the flexibility to do several things in parallel and asynchronously. You can map these concepts directly into message-based software solutions.

Messaging in Daily Life

Before I cover messaging from a technical perspective, let's first discuss the various ways we all use messaging in daily life:

- Sending and receiving traditional mail
- Sending email
- Calling someone on the phone

When you send a letter, you perform an asynchronous process using a message. Such a message always contains three elements: an envelope, a header, and a body. Figure 1-1 illustrates this concept. The header contains some routing information (the address where the letter must be sent, or the *target*) and some information about the sender (the address of the *initiator*). The header also contains a stamp with the appropriate value. Using this information, the post office can route your letter to its final destination through some intermediaries. Intermediaries could be other post offices along the route to the receiver. For example, when I order books from an online store, the books are routed to a distribution center in Vienna. The local post office then takes the parcel and delivers it directly to my house. The distribution center is a kind of intermediary.

Let's concentrate now on the payload of such a message. The sender defines the payload (body). For example, you can write a handwritten letter, you can type it, and you can also enclose some pictures from your last trip to Seattle. The payload can also be defined through a formalized agreement. When the payload is in XML format, then the agreement can be an XML schema.

Sending an email is similar to sending a letter. An email message is transferred with the Simple Mail Transfer Protocol (SMTP). This protocol defines what goes into the header, and the sender defines the contents of the payload. Just as with a letter, you can write some text, embed some graphics, and even attach some documents. All this information is serialized into the payload, and the message is transferred through the SMTP protocol to the receiver's email address.

Calling someone on the phone entails another type of messaging. The header of this message contains the phone number that you dial, and the payload is the spoken words you deliver through the phone line. In the same manner as you can send an email when the receiver is currently not available, you can also leave a voice-mail message. So, a phone call also works when the receiver is "offline."

When you compare a phone call to an email, you'll find some differences. First, a phone call is not always reliable. You can't assume that the receiver understands your words exactly, because the transmission can sometimes be distorted. Second, the phone connection may be cut off when you're talking to someone. In such cases, a phone call is like the User Datagram Protocol (UDP), because message delivery isn't guaranteed in either case.

Why Messaging?

When you want to use a new technology in a project, you must be able to argue its merits. Nobody can just go to a project leader and insist that messaging technology be used simply because it's available. Let's see how messaging solutions can have a positive impact on your application architecture.

Asynchronous Message Processing

Asynchronous message processing means that the sender can continue working while waiting for the receiver to process and eventually reply to the message. Asynchronous message processing is useful when long-duration business processes should be decoupled from a client application. The client application only captures the necessary input from the user, transforms the input into a message, and sends it to the destination, which does the real work asynchronously. The benefits of this design are that client applications are more responsive and users can immediately do other things within the application. This design may include several destinations to achieve a load-balancing scenario, so that the workload can be distributed and messages can be processed in parallel.

Sending a letter is a form of asynchronous message processing, because you don't have to wait at the post office until your letter arrives at its final destination. Likewise with email: you just click the Send button, and the email is transferred. In the meantime, you can continue with your other work.

When you implement message-based solutions, you work with an asynchronous processing paradigm. When you send a message, the message is sent asynchronously to the receiver, where it may be processed accordingly. I say "may be" because asynchronous messaging does not define when the message is processed (the next section talks more about deferred message processing). One of the benefits of this asynchronous approach is that users can continue with their work while a message is being processed on an application server. This leads to more scalable applications, because clients only have to put messages into a queue. Most other things are done at some later time without further user interaction.

Deferred Message Processing

There's an important difference between asynchronous and deferred message processing: asynchronous message processing defines that the work is done later, and deferred message processing defines when the work is actually done on the application server. The message can be processed immediately, in a few hours, or in a few days—it depends on the message system configuration. Deferred message processing in an asynchronous scenario has the following advantages:

- The receiver controls when messages are processed.
- Load balancing occurs between several servers and during peak times.
- Fault tolerance exists on the receiving side.

Let's assume you run a messaging system based on Service Broker in a company where a high message load occurs during normal work hours. With deferred message processing, you can configure Service Broker so that messages are only put into the queue during the day and are processed overnight when more resources are available. This means that user requests can be processed more quickly, and applications are more responsive under high loads. This is similar to doing automatic database backups at night—another case where demand is less, and more resources are available at night.

With load balancing, you can define on which server a message should be processed. This is useful when you have a high load of messages that must be processed quickly. You can deploy several instances of a Service Broker service and have clients connect to one of them. If you must take an instance offline, perhaps to perform some maintenance work on the server, your messaging system will continue to work without any problems or changes in configuration, because new requests from clients will be distributed among the other available Service Broker instances.

Fault Tolerance

Messaging technologies can also help you provide fault-tolerant systems. You can achieve fault tolerance through the following ways in a message-based system:

- Reliable messaging
- Load balancing
- Dynamic rerouting

Reliable messaging means that the sender can ensure that a message will arrive at the target. This reliability is completely independent of whether the target is currently available. If the target is offline, then the messaging infrastructure on the client will try to resend the message to the target as long as the target is offline or otherwise not reachable through the network. This approach has several advantages: maintenance work can be done more easily, sent messages cannot be lost, offline scenarios are supported, and so on.

A big benefit of reliable messaging is that the target can send a response message back to the sender in a reliable fashion. So, reliable messaging works in both directions on a communication channel. Therefore, reliable messaging is also an important consideration when you design and implement smart client solutions. One characteristic of a smart client is that the application still functions in offline scenarios. Microsoft Office Outlook, for example, is an intelligent smart client because it allows you to work through your emails and even compose replies when you're on the road and not connected to the Internet. When you get back to your office, Outlook synchronizes its local data store with an Exchange Server in the background and sends all unsent emails to their receivers.

You can provide the same flexibility in your own smart client applications when you use a message-based approach. When users want to do some work offline, you can store their requests as messages locally on their notebooks and send the messages to the application server as soon as the notebook goes online and an Internet/intranet connection is established. Reliable messaging makes this possible without any programming.

Load balancing offers the same benefit for fault-tolerant systems that it does for deferred message processing. By distributing message processing power between several servers, you can implement a scale-out scenario (scaling horizontally) for your messaging system. As you'll see in forthcoming chapters, you can do this easily with Service Broker.

Dynamic rerouting means that you can reconfigure a Service Broker service during the processing without making any changes on the client and without any further interruption. As soon as the endpoint of a Service Broker service is reconfigured and points to a new address, the clients send messages transparently to the new location, without any intervention from an administrator. This could be helpful when you need to take a service offline during maintenance work.

Distributed Systems

If you must implement or support a distributed topology for your application, then a message-based approach can help you achieve the necessary functionality. Architectures are referred to as distributed when software components are running on different servers on the network. For example, you can implement an application server where business logic executes in a central location. This provides you the advantages of asynchronous and deferred message processing, and these architectures are also very scalable.

Some messaging technologies support scale-out scenarios without any change in the implementation of the message-based services. Service Broker supports this through a concept called *routes*. A route defines the endpoint on the network where a service is hosted. You'll find more information about routes in Chapter 7.

When you distribute software components across process boundaries, you must also give careful thought to the following questions about security:

- How are clients authenticated and authorized when they call services?
- How are messages transferred between the sender and the receiver? Is encryption necessary?
- Should you use symmetric or asymmetric encryption?
- How do you react to threats from the outside world?

As you can see, there are many aspects to consider when implementing distributed message-based scenarios. In Chapters 7 and 8, you'll get an in-depth look at how to set up distributed scenarios with Service Broker and how you can secure communication between sender and receiver.

Messaging Problems

You've seen why a messaging technology such as Service Broker makes sense for some scenarios. However, using a messaging technology just because it's available isn't necessarily a good idea. In fact, it can have an enormous negative impact on your application architecture. You must decide carefully if a messaging architecture provides benefits for your required scenarios. If you do decide you want to use messaging technology and you don't want the problems of message systems, Service Broker can help because it already includes functionality that solves the messaging problems described in this section. Let's take a detailed look at which messaging problems Service Broker tries to solve.

Performance

Performance is always an issue. When you ask developers what problems they have with their databases and applications, they'll almost always mention performance. Therefore, one of Microsoft's goals with Service Broker was to have it offer better performance than any other messaging technology can currently provide. For example, if you compare Service Broker to MSMQ, you'll see that the biggest difference is in the transaction handling. With Service Broker, you don't need distributed transactions.

Service Broker solves the transaction-handling problem completely differently. In the Service Broker world, queues, like tables, are native database objects, so distributed transactions are not needed when a message is processed and finally removed from the queue. Service Broker can handle all the different tasks in the context of a local SQL Server transaction—and this provides enormous performance benefits over MSMQ and other message systems. You can find out more about this topic in Chapter 6, which discusses transaction management in Service Broker.

TRANSACTION HANDLING IN MSMQ

With MSMQ, your data-processing logic is typically implemented in a database. The problem is that MSMQ messages are stored in the file system, but the data-processing logic is in a database. Because of this separation, you must coordinate two different resources during message processing.

You can't remove a message from an MSMQ queue when a database transaction fails, and you can't commit a database transaction when you can't remove a message from the queue. In both cases, you get inconsistent data between the MSMQ queue and the database.

In MSMQ, you use a distributed transaction when you must coordinate more than one resource in the context of a transaction. However, distributed transactions are relatively expensive in terms of execution time because of the coordination overhead. This can be a major problem in itself, so you must always carefully decide if you can afford this extra overhead.

Queue Reader Management

In every message system, a message is retrieved from a queue. This process is called *queue reader management*. There are two possible options: you can have several queues, where incoming messages are distributed among the queues, or you can have one queue that receives all the messages. Let's look at both solutions.

Several queues might seem at first to be the better option, because multiple components listening on the different queues can process the messages. You can compare this approach to a supermarket, where you must wait in one of several check-out lines (queues) to get to a cashier.

But what happens when a new line opens? Many move to the new line, and, of course, they all want to be the first there. It can become even more complicated. What if you're the second person in the newly opened line? It often seems like the person in front of you takes too much time, and it would have been better for you to have stayed in your original line. As you can see, more than one queue isn't always the best option, for supermarkets or for message systems.

■ **Note** The rebalancing of queues always has a cost, and that cost can negatively affect performance.

But can things work efficiently with just one queue? Think of the check-in process at an airport. Here you have one queue with several check-in stations. When a new station opens, does this lead to the same problem as with the supermarket check-outs? No, because the people in line are always distributed among the current available stations without being able to make the choice on their own. So no rebalancing is necessary among queues.

The second approach is the one that Service Broker takes. In Service Broker, there is only one queue, but you can configure the number of queue readers listening on the queue to process incoming messages. In fact, you have multiple concurrent queue readers processing the incoming messages. In a message system, this is a lot more efficient than multiple queues.

Transaction Management

An effective message system must also support transaction management. Transaction management defines how several instances of queue readers are coordinated against one queue. To make this clearer, let's assume a scenario where an order-entry application sends order request messages to a queue. Two kinds of request messages are involved:

- A message with the order header
- Several messages with the order line items

As soon as the first message arrives at the queue, a queue reader is instantiated, which takes the new order header message from the queue and processes it. The queue reader inserts a new row in an order table and commits the transaction. Then, a reader processes the line-item messages and also inserts new rows in the database. Everything seems OK, but what happens when an administrator configures more than one queue reader to listen for new messages? In this case, the order header and the line-item messages can be processed in parallel, with the possibility that an attempt could be made to insert a line item before the order header. This will raise an exception, because it violates referential integrity between an order header and its line items.

One possible way around this is to process messages in the correct order. Service Broker solves this problem through a feature called *conversation group locking*. Conversation group locking groups related messages for reading by multiple readers to prevent data integrity problems caused by processing related messages simultaneously on different threads. During the design of your Service Broker application, you must make sure that related messages are placed into the same *conversation group*.

When you use conversation group locking, Service Broker ensures that the same queue reader processes messages from the same conversation group, eliminating the need to explicitly synchronize message processing among several threads. Chapter 6 takes a more in-depth look at transactions, locking, and conversation groups.

■ **Note** Conversation groups are similar to a dialog, where several messages are exchanged between the sender and the receiver.

Message Sequencing and Correlation

Another issue in message systems is correctly sequencing and maintaining the right correlation among messages. Sequencing means that messages arrive in the same order as they're sent, as Figure 1-2 illustrates.

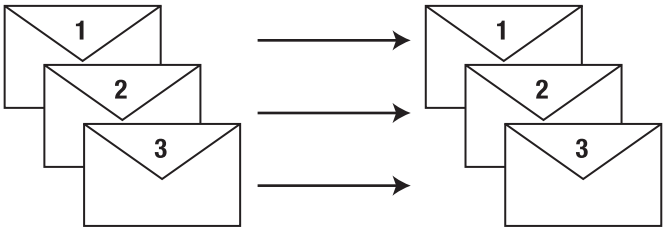

Figure 1-2. *Message sequencing*

Sequencing is always important when a receiver gets several messages, each of which depends on a previous message. In the order-header example, the message with the order header needs to be processed before the messages with line items. Therefore, you can again ensure the referential integrity of your data in the database. To preserve order, Service Broker includes a sequence number in every message it sends and forces the receiving application to accept the messages in the same order as they were sent. Some other messaging systems (such as MSMQ) allow messages to be received in any order, but Service Broker enforces the strict sequencing of messages. Service Broker doesn't include built-in support for priority message ordering, but Chapter 10 shows you how you can overcome this problem.

Message correlation is another problem. When a client sends several messages to a central service queue, the queue readers process the messages and send back response messages. But how can the client decide which response is for which request? Service Broker solves this problem easily, because each message is associated with a *conversation ID*. This unique identifier identifies the communication channel between the sender and the receiver in a unique fashion. The client application can then associate each response with the correct request.

Maintenance

Maintenance is an important issue for all applications, including message applications, and Microsoft addressed this issue when designing Service Broker. Compared to MSMQ, Service Broker is much easier to administer and maintain because its queues are native database objects. When you back up the database, you also back up both the data created by the messages and the queues holding messages that are not yet processed. This is possible because all Service Broker objects, like queues, are tied to a database.

In MSMQ, messages are always stored in the file system. When you do backups, the database and the messages in the file system can be out of sync (see the "Maintenance in MSMQ" sidebar). Another big advantage of Service Broker is that administration and maintenance are done with tools that database administrators already know. The best and only tool they need is SQL Server Management Studio, but they can use *sqlcmd* if they prefer to. You don't have to introduce a new management tool for your messaging solution.

MAINTENANCE IN MSMQ

With MSMQ, messages are stored in the file system, and data is stored in a database such as SQL Server. You must back up both the messages in the file system and the data in SQL Server. Your backups can easily get out of sync, because while you update one resource, changes may be made to the other resource.

The only solution to this problem is to take your messaging solution offline, make the backup, and then put it back online. However, this doesn't work with a 24/7 system. With Service Broker, synchronizing backups is not a problem, because both messages and data are stored in the same database. You can do one backup, on the whole database, and that's it.

Messaging Architectures

Implementing messaging solutions also leads to new application architectures that you can't achieve with traditional approaches such as client/server or three-tier architectures. Let's have a look at two other possible types of architecture for message-based software systems: Service-Oriented Architecture (SOA) and Service-Oriented Database Architecture (SODA).

SOA

SOA is a collection of several architectural patterns with the goal of combining heterogeneous application silos (such as mainframe applications). A quick search on Google may lead you to believe that SOA is only possible with XML web services, but that's not true. You can use SOA with other technologies, such as Service Broker. SOA defines the following four core principles:

- Explicit service boundaries
- Autonomous services
- Explicit data contracts
- Interoperability

As you'll see throughout this book, you can satisfy these principles better and with more reliability with Service Broker. Explicit service boundaries mean that an SOA service must define a service contract that exposes the operations available to other client applications. This is important when a client uses discovery technologies to find an appropriate service on a network.

An autonomous service is one that a client can use to process a complete business request. Email, for example, is an autonomous service, because a user request can be completed with one service interaction. If you want to send an email with an attachment, you can do it in one step instead of two separate steps. The big challenge when you design your services is to find the right granularity and make them as autonomous as possible.

In SOA, the contract defines the contents of the messages sent in both directions. In the context of Service Broker, you can define the structure of the message body. You have no control over the structure of message headers. XML messages support interoperability, because any computer system can exchange and process them.

SQL Server 2005 allows you to expose Service Broker services to other clients through open standards, such as XML web services. This makes it possible for clients on other platforms, such as Java, to interact with your Service Broker services. You can adhere to all four SOA principles with Service Broker, making it an ideal platform for implementing SOA.

SODA

SQL Server 2005 offers a number of new features, including the following:

- Integration into .NET (SQLCLR)
- Query notifications
- Service Broker
- XML support
- Web services support

Many customers often ask why these features are now integrated directly into the database. There are several good reasons for each feature that I won't go into right now because that's not my purpose. My point is that you can only achieve real benefits from these features when you use them in conjunction. The correct use of these features leads to SODA, the concepts of which are explained in a white paper by David Campbell called "Service Oriented Database Architecture: App Server-Lite?"[1]

In SODA, you implement business functionality as SQLCLR stored procedures in the database, and you use Service Broker as a reliable message bus to make your components available to other clients. To publish services, you use native web services support in combination with the new XML features of SQL Server 2005. When you look at this new architecture, you can see that SQL Server 2005 is an important application server in such scenarios. Chapter 9 discusses implementing SODA applications with Service Broker.

Available Messaging Technologies

Service Broker is not the one and only messaging technology available for the Windows platform. You can use several technologies to implement a message-based system, but Service Broker offers some advantages over all the other messaging technologies described in this section. For example, one important aspect of Service Broker is its distributed programming paradigm. When you develop a Service Broker application dedicated for one SQL Server and you later decide to spread the Service Broker application out to several physical SQL Servers (maybe because of scalability problems), then you just have to configure your application to support a distributed scenario. You don't have to change the internal implementation details of your Service Broker application.

Likewise, with load balancing, if you see in a later phase of your project that you must support load balancing because of several thousands of concurrent users, then you just have to deploy your Service Broker application to an additional SQL Server and make some configuration changes. Service Broker will handle the load-balancing mechanism for you in the background. Chapter 11 talks more about these scenarios.

Despite these advantages of Service Broker, let's now take a look at one of the most important and familiar messaging technologies available today.

MSMQ

MSMQ has been available as part of Windows since the first version of Windows NT. MSMQ was the first messaging technology from Microsoft used to provide messaging capabilities for a wide range of business applications. One of the biggest advantages of MSMQ is that it is licensed and distributed with Windows, so you don't have any additional licensing costs when you use it in your own applications. In addition, it's not bound to any specific database product. If you want to use Oracle with

1. David Campbell, "Service Oriented Database Architecture: App Server-Lite?" Microsoft Research (September 2005), `http://research.microsoft.com/research/pubs/view.aspx?tr_id=983`.

MSMQ, you can do it without any problems. However, as with every product and technology, there are also some drawbacks, including the following:

- Message size is limited to 4 MB.
- MSMQ is not installed by default. Furthermore, you need the Windows installation disk to install MSMQ.
- You need distributed transactions if you want to run the message processing and data-processing logic in one Atomic, Consistent, Isolated, and Durable (ACID) transaction. This requires installation of the Microsoft Distributed Transaction Coordinator (MS DTC).
- Message ordering is not guaranteed.
- Message correlation is not supported out of the box.
- You must implement queue readers manually.
- You must conduct synchronization and locking between several queue readers manually.
- Backup and restoration can be a challenge, because message data and transactional data are stored in different places.

Queued Components

Queued Components are a part of the Component Object Model (COM+) infrastructure. With Queued Components, you have the possibility to enqueue a user request to a COM+ application and execute it asynchronously. Internally, a message is created and sent to a dedicated MSMQ queue. On the server side, a component referred to as a *Listener* is used to dequeue the message from the queue and make the needed method calls on the specified COM+ object. For replay of these method calls, a component referred to as a *Player* is used. Queued Components are attractive for a project that already uses the COM+ infrastructure and requires doing some functions asynchronously and decoupled from client applications.

BizTalk Server

BizTalk Server is a business process management (BPM) server that enables companies to automate, orchestrate, and optimize business processes. It includes powerful, familiar tools to design, develop, deploy, and manage those processes successfully. BizTalk Server also uses messaging technology for enterprise application integration (EAI). One drawback is its licensing costs, which are very high if you need to support larger scenarios where scale-out is an issue.

XML Web Services

XML web services is a messaging technology based on open standards such as SOAP and Web Services Description Language (WSDL), and it's suitable for interoperability scenarios. .NET 1.0 was the first technology from Microsoft that included full support for creating applications based on web services technologies.

Over the past few years, Microsoft has made several improvements in the communication stack and has made it even easier to design, implement, publish, and reuse web services.

WCF

The goal of WCF, which is part of .NET 3.0, is to provide a unique application programming interface (API) across all communication technologies currently available on Windows. This includes the technologies already mentioned, as well as some others, such as .NET Remoting. With a unique

communication API, you can write distributed applications in a communication-independent way. During deployment, an administrator can configure which communication technology the application should use. Microsoft's Service Broker team might also include a WCF channel to Service Broker in an upcoming version of SQL Server, so that you can talk with Service Broker applications directly from WCF-based applications.

Summary

In this first chapter, I provided an overview of the fundamentals of message-based programming. I talked about the benefits of using messaging and how to achieve scalability for your applications. I then discussed several problems that can occur when using messaging technology, and I showed you how Service Broker solves these problems so that you don't need to bother with them and can simply concentrate on the implementation details of your distributed applications.

I then described possible application architectures based on messaging architectures such as SOA and SODA. Finally, I briefly described other messaging technologies available on Windows and presented the pros and cons for each. With this information, you have all the necessary knowledge for understanding the concepts behind Service Broker. In the next chapter, I'll present an introduction to Service Broker itself.