15

SQL Server Service Broker

SQL SERVER SERVICE BROKER (SSB) is a new part of the database engine that is used to build reliable, asynchronous, message-based distributed database applications. Service Broker allows these applications to concentrate their development efforts on the issues of the problem domain in which they are going to be used. The system-level details of implementing a messaging application are delegated to Service Broker itself and can be managed independently by personnel who know the system issues.

Messaging Applications

Messaging applications are nothing new. Almost all large scalable enterprise applications use some sort of messaging infrastructure. Messaging applications take a different approach to providing a service than application based on functions. When you need a service from a message-based application, you send it a message and go on about your business. If you care, the service will some time later return to you a message concerning the status or completion of your request. Some of the compelling reasons for message-based applications are the following.

- *Deferred processing*—It may not be possible, or necessary, to perform all the work associated with a particular task at one time. For example, a stock trade to sell 100 shares cannot be completely processed when it is initially entered into a trading system; the

person who enters the trade and the one who completes the processing of the trade by actually selling or buying the stock are different people separated in time. Service Broker can manage the trade from the time it is initially entered until it is completed at some later time so that an application can concentrate on implementing each phase of the processing of a trade. Service Broker is completely capable of managing deferred processing over indefinite spans of time, even months or years, and across database restarts.

- *Distributed processing*—The work associated with a task must be completed in a timely manner. However, it is often quite difficult to predict in advance how many tasks there will be and how many resources it will take to complete them. Distributed systems allow processing resources to be applied where there are needed and be incrementally expanded without changing the applications that make use of them. Service Broker allows system administrators to manage the resources in a distributed system so that the application can be developed as though all the resources it needs were always available to it.

These features are compelling because they allow an application to concentrate on its problem domain and leave tedious and difficult to implement system details to Service Broker. A number of details of system implementation for a messaging application will be unrelated to its problem domain and hard to properly implement. Improper implementation of these details results in an unreliable application. These details fall into three major areas.

- *Message order*—It is much easier to write a messaging application that receives messages in the order in which they were sent to it. In practice, messages cannot be depended on to arrive in the order in which they were sent, and may not arrive at all. Service Broker will ensure that messages are received, and received in the order in which they were sent.
- *Message correlation*—Messaging applications often require replies to the messages they send. The replies for these messages may be quite delayed in time and rarely arrive in the same order in which the messages that caused them were sent. Finding the message that caused the reply to be sent is called correlation. Service Broker can be used to manage the correlation of replies with the messages that caused them.

- *Multithreading*—A messaging application will often run on multiple threads in order to more effectively make use of system resources or more easily manage independent items of work. Resources—for example, queues and other states—are shared among all the threads in the application and, if not properly managed, will lead to two threads mutually corrupting the shared resource. This is sometimes called the "synchronizer problem" and is very difficult to prevent. Service Broker can be used to manage shared resources so that a messaging application can be written as though the synchronizer problem did not exist, but still take advantage of running on multiple threads.

SQL Server Service Broker Overview

The SQL Server Service Broker is a technology for building message-based, asynchronous, loosely coupled database applications. Service Broker makes it possible for applications to send and receive ordered, reliable, asynchronous messages. It is built into the SQL Server engine, and applications are developed using extensions to T-SQL. This allows an enterprise to leverage its existing database and/or CLR skills to build message-based applications.

Service Broker manages services that receive, process, and send messages. Multiple services may share an instance of SQL Server, use different instances of SQL Server, or do a combination of both.

Messages are sent to a service. When a message arrives at the service, it is put into a queue associated with the service. Once a message arrives in a queue, it may be processed by a program, called a service program. Any program that has access to the queue can do that processing. However, a standard way of processing these messages is to assign a stored procedure to a queue. When this is done, Service Broker will invoke that stored procedure when a message arrives in the queue.

The service may be configured to use a limited number of instances of the stored procedure so that more than one message at a time may be processed. If the service is very busy, a number of messages may build up in the queue, but eventually they are processed. Figure 15-1 shows a simple message-based system that illustrates how Service Broker invokes instances of a stored procedure so the stored procedure can process messages as they arrive in a queue.

The system in Figure 15-1 receives orders from applications and processes them. It only requires that the service program be implemented

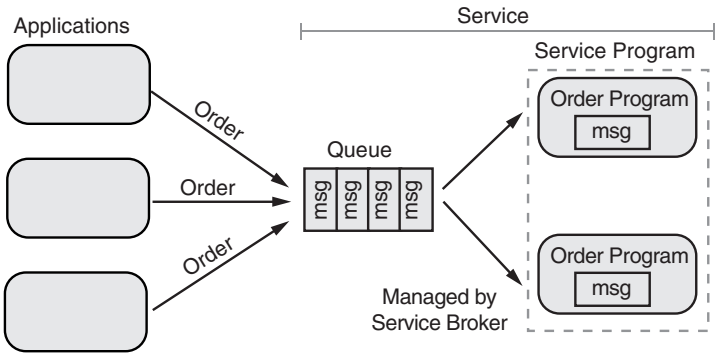

FIGURE 15-1: Simple Message-Based System

by someone who is familiar with the problem space of processing orders; everything else is configuration that can be done after the service program is written and even unit tested. The service in this example is configured to use the order service program to process the messages that arrive in its queue. The applications only need to be able to make a connection to SQL Server to be able to submit orders into the service. If there are messages in the queue, Service Broker will eventually invoke an instance of the service program. In this simple message-based system, the order service program would read the message from the queue and process it.

However, the service program is not required to process any messages in the queue at the time it is invoked; it can choose to do otherwise. In other words, Service Broker will invoke a service program when messages are available for processing, but it is up to the service program to decide whether or not to process messages in the queue.

Of course, the system illustrated in Figure 15-1 could have easily been implemented by having the applications directly call the service program itself, but this example is meant to show the basics of message processing as it is done by Service Broker. It is worth noting, however, that even this simple example limits the number of messages that will be simultaneously processed, which is a key problem in system design. Without Service Broker, this would require the implementer of the order stored procedure to have knowledge beyond the problem space of processing orders, to prevent an unexpected rush of orders from swamping the system by trying to make it process too many messages at once.

Queues are one of the two main features of Service Broker. They allow processing of messages to be deferred. A message stays in a queue until resources are available to process it. Resources may be unavailable because of

the limited number of instances of a service program to process them. Most importantly, queues may be reconfigured while an application is running.

In almost all applications, there are some tasks that must be done immediately and others that can be deferred. For example, during peak load, one queue for a service processing deferrable messages might have the invocation of its service process turned off by the system administrator. And another that must process messages immediately might have the number of instances of its service process increased by the system administrator. This diverts resources to services that must process their messages immediately. When the peak passes, the system administrator can reconfigure again to allow deferred messages to again be processed. This ability to reconfigure Service Broker at runtime greatly aids in maintaining the scalability and performance of a system. Figure 15-2 shows a Service Broker application that has been configured for peak load.

In software, constructing a queue is very easy; in fact, the .NET Framework includes the `System.Collections.Queue` class to do this for you. However, the queue that can be made from this class is transient; it is meant to be used only as long as the program that created it is running.

You can't build a messaging system using the simple queues provided by `System.Collections.Queue` and similar classes, because they just are not reliable enough. Under the covers, in Service Broker a queue is implemented in a table, and messages can be removed or added to a queue using a transaction. This gives the queue in Service Broker the same features we expect of anything else that operates on data in a database—it is atomic, consistent, isolated, and durable. Basically, it is reliable.

In fact, not only does Service Broker use SQL Server to implement queues, it uses SQL Server to store and implement all aspects of a messaging

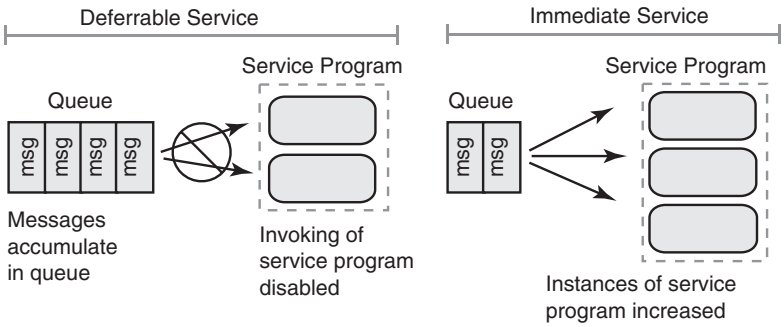

FIGURE 15-2: Service Broker Application Configured for Peak Load

application. This, then, is the second main feature of Service Broker: It provides a complete framework based on SQL Server for implementing reliable messaging applications.

It is hard to overemphasize the importance of this feature—without Service Broker or some similar framework, probably 80% of the code written for a messaging application would be for infrastructure, not the problem space of the application. The resulting framework would, of course, not be SQL Server and would require a completely different set of skills and utilities than those needed to maintain SQL Server. A Service Broker application is just a collection of SQL Server objects that can be maintained using the same skills and tools used to maintain anything else in SQL Server.

A note on terminology: Terms that refer to *type* and *instance* are often overloaded and depend on context, which is often not clear, to distinguish between which overload is being used. The term "message" might sometimes refer to the definition of the format of a message and at other times refer to an actual message. Anytime the term "message" is used in this chapter, it is referring to an actual message that complies with a message type definition; that is, it is an instance of some message type. The term "message type" will always be used to refer to a definition of a message format. This distinction is necessary because Service Broker not only manages messages, it also manages message types.

A service uses Service Broker to send a message to another service. Service Broker does this by putting the message into an output queue and then sending it, possibly at a later time, to the queue for the other service. A number of messages may build up in the output queue, waiting to be sent, but they will eventually be sent and sent in the order in which they were put into the queue.

The advantage of this extra layer of queuing is that the service sending the message never waits for anything. But the extra layer also introduces extra overhead. Service Broker will skip the output queue when both services are on the same instance of SQL Server. In this case, it will put the message directly into the queue from which the receiving service gets its messages. Figure 15-3 shows how Service Broker efficiently sends a message from one service to another.

So far we have seen a very general picture of how Service Broker is used to make messaging applications. Service Broker is a framework and extensions to T-SQL that are used to create and use the components used to build a message-based application. We have already used some of these

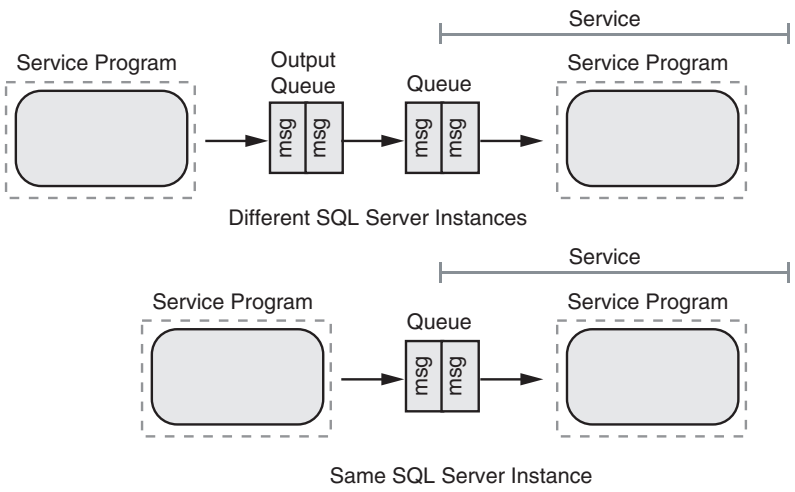

FIGURE 15-3: Sending Messages between Services

components: messages, queues, and services. Concise definitions of the components used in a Service Broker application follow.

- *Service program*—A service program is used to process messages. A service program may be a stored procedure written in T-SQL or a CLR-compliant language. A service program may also be a program written in any language that has access to SQL Server. As part of the processing of a message, the service program may send messages to other services.
- *Queue*—A queue is a component that has a name and can hold messages in the order in which they were received while the messages await processing. A queue may have a particular service program associated with it, but it is not required to.
- *Message type*—A message type is a definition of the format of a message. It is stored in SQL Server and has a name. One service communicates with another service by sending an instance of a message type. A service can only send a message for which a message type has been defined.
- *Contract*—A contract is a set of names of message types. It is stored in SQL Server and itself has a name. The contract defines nothing about the order in which the message types must occur, but each message type must be marked as being sent by an `INITIATOR`, a `TARGET`, or `ANY`, which determines how it may be used.

- *Service*—A service is a specification that is stored in SQL Sever and has a name. It must specify a queue that will be used to hold messages sent to it. Optionally, it may also list a set of contracts that specify the types of message that may be sent to it.
- *Conversation*—A conversation is a component that is used to correlate and order messages a service receives. It is created using the `BEGIN DIALOG CONVERSATION` T-SQL command and is the principal component an application uses to make use of Service Broker. Any program that has access to SQL Server, including a service program, can create a conversation.

Services communicate with each other using a conversation. When a service wants to communicate with another service, it creates a conversation.

The conversation includes a service contract, which will be used during the conversation. A conversation is between two services that are named in the `BEGIN DIALOG CONVERSATION` command, using the message types defined in the contract associated with the conversation.

These are some other Service Broker components used by conversations.

- *Conversation group*—A conversation group is created by Service Broker and assigned a unique identifier. A conversation group represents a user-defined set of conversations.
- *Routes*—Routes are used when conversations are created between different instances of SQL Server. They serve as a level of indirection so that the actual instance of SQL Server being used can be changed without changing any of the service programs when a target is moved.
- *Remote service bindings*—The remote service bindings associate a remote service with the user in the local database. They are used to handle authorization for the remote service and encryption of the messages exchanged with the remote service.

A trivial Service Broker application that makes use of some of these components is shown in Figure 15-4.

In this trivial Service Broker application, there is a validate service and a fulfill service. The validate service is used to validate an order. If the order is valid, the validate service sends the order to the fulfill service to have it shipped.

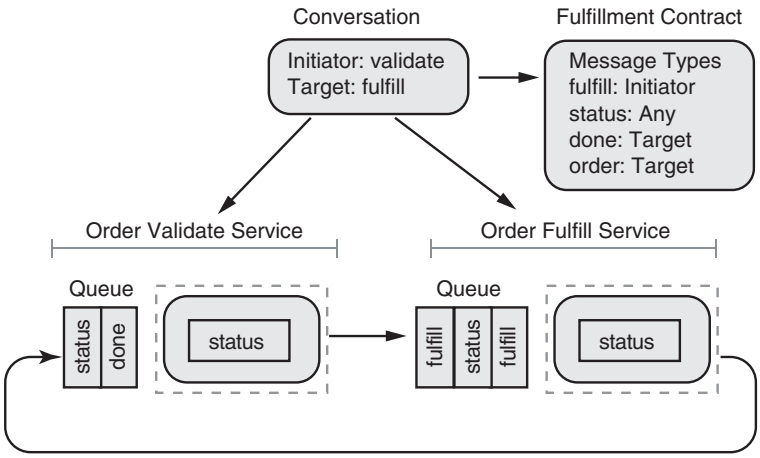

FIGURE 15-4: Trivial Service Broker Application

The fulfillment contract specifies four types of messages: fulfill, status, order, and done. The fulfill message may only be sent by the initiator. The validate service is the initiator because the dialog specifies it as the initiator. The order and done messages may only be sent by the target. The fulfill service is the target because it was specified as a target in the dialog.

The status message may be sent by either the validate or the fulfill service. In this trivial application, the status message would be sent whenever something of significance occurs in the processing of an order, to let the other service know about it. This relieves the other service of having to poll for status.

As shown in the trivial Service Broker application, the fulfill service is currently processing a message. When it is completed, it will send a done message to the validate service. It has a number of other messages in its queue waiting for processing.

The contract ensures that only messages specified can be sent only in the direction specified. An attempt by either service to send the wrong message will result in an error for that service. When the validate service receives a done message, it will use the correlation capabilities of Service Broker to figure out which fulfill message has been processed by the fulfill service.

This example shows the "big picture" of how Service Broker works and glosses over all the details. Conversations are central to understanding Service Broker. Let's take a closer look at the many facets of conversations now. We will not be looking at the details of the APIs yet; that comes

later. Here we just want to present a model of how Service Broker uses conversations.

A conversation is created using BEGIN DIALOG CONVERSATION. It must list a FROM service and a TO service by name; a conversation always takes place between two services. The FROM service is, in effect, the reply-to address the TO service will use when it needs to reply to a message. Messages for the FROM service will go into the queue specified by the FROM service, and similarly for the TO service.

A conversation must also specify the contract from the set of contracts listed as being supported by the TO service. This limits the message types used in the conversation to those listed in the contract. It further limits the FROM service, in this conversation, to receiving in its queue only those message types marked as being sent by an INITIATOR or ANY. And likewise it limits the TO service to receiving in its queue only those message types marked as being sent by a TARGET or ANY.

Although you can think of "sending a message to a service," there is no way to do this directly. In fact, probably the biggest hurdle in understanding Service Broker is in understanding how it manages conversations. Messages are not sent to a service, they are sent ON a particular conversation. This is how messages are correlated and ordered; obviously, all messages sent ON the same conversation are correlated.

A service program reads a message from a queue by using the T-SQL command RECEIVE. A queue physically is a table, and in effect this selects a row from the table and then deletes it. The message includes a conversation handle, which identifies the conversation on which the message was sent.

If the service program needs to send a reply to that message, it sends the reply message on the conversation handle that was in the message it read out of the queue. This way, the service program does not need to specify the specific service as the recipient of the message. Sending messages on a conversation handle rather than to a specific service greatly simplifies creating distributed applications.

Hopefully, by this point you will be thinking, "OK, I can easily reply to any message I receive; how does the first message that starts things going get into the queue?" When a program, which could be a stored procedure or any program that has access to SQL Server, creates a conversation using BEGIN DIALOG CONVERSATION, it gets back a conversation handle. It can then send a message on that conversation handle, and the message will be sent to the queue of the TO service. From there what happens depends on the service program that reads that message out of that queue.

It is easy to see how sending messages on a conversation instead of to a service allows Service Broker to guarantee message order and correlation. However, in real-life messaging applications this is not enough. Only a trivial messaging application could depend on a single conversation between two services. A typical business process would involve many services, which means there would be many conversations. All these conversations must be coordinated for two reasons. One is to guarantee the processing order of messages across a group of conversations. The other is to allow state to be maintained for a group of conversations and used by any service program that processes messages from that group of conversations.

Service Broker calls a group of related conversations a *conversation group*. Every conversation belongs to a single conversation group, and every conversation group has a conversation group ID. Every message contains both a conversation handle and the conversation group ID of the conversation group to which the conversation belongs. An application decides if a conversation should be part of a new conversation group or added to an existing one when it creates the conversation using `BEGIN DIALOG CONVERSATION`.

In order to make the new conversation part of a particular conversation group, the application needs either a conversation handle or a conversation group ID to pass into `BEGIN DIALOG CONVERSATION`. If a conversation handle is passed in, the new conversation will be added to the conversation group of that conversation.

If a conversation group ID is passed in, it will be added to that conversation group. You might get this conversation group ID from a message you are processing, or you might just create a new conversation group with no conversations in it. You can do this by using the `NEWID()` function and passing the `UNIQUEIDENTIFIER` that it returns into `BEGIN DIALOG CONVERSATION`. This allows you to have the conversation group ID before you do `BEGIN DIALOG CONVERSATION`. Later when we look at shared state, you will see that this technique lets you set up shared state before any conversation begins.

If neither a conversation handle nor a conversation group ID is passed into `BEGIN DIALOG CONVERSATION`, a new conversation group is created for that conversation.

A conversation group has a lock associated with it. This lock is used to guarantee message order and manage state across all conversations in the conversation group.

The conversation group can be locked in two ways. Whenever a message is read from or sent to a queue, the conversation group associated

with the message will be locked. Messages are read from a queue using the RECEIVE command and sent to a queue using the SEND command.

The second way to lock a conversation group is to use the GET CONVERSATION GROUP command. The GET CONVERSATION GROUP command is issued for a particular queue. The command locks the conversation group associated with the first message in the queue from a conversation group that is not locked. Note the logic here: The GET CONVERSATION GROUP command will skip over messages from conversation groups that are locked until it finds one from a conversation group that is not locked.

The lock associated with a conversation group has a lifetime. It remains locked until the transaction under which it was locked completes. In typical usage, a transaction will have been started before RECEIVE or GET CONVERSATION GROUP is called.

When a conversation group is locked, all threads except for the one that locked the conversation group are blocked when they try to use RECEIVE to get a message that is from the locked conversation group. A RECEIVE that attempts to get a message from a different conversation group that is not locked will not be blocked.

The RECEIVE command can be selective about which messages is will read from a queue. It can choose to receive all messages in a queue, only the messages associated with a particular conversation group, or the messages for a particular dialog.

In typical usage, a transaction is started, GET CONVERSATION GROUP is used, and then RECEIVE is used, followed by either a COMMIT TRANSACTION or a ROLLBACK TRANSACTION. If a ROLLBACK TRANSACTION is used, all messages that were read from the queue are placed back into it, and all the messages that were sent are removed from the queues that received them. Until the transaction is committed, of course, nothing that was sent is visible to the outside world.

Though it is not required, RECEIVE is typically used after a GET CONVERSATION GROUP command. Using GET CONVERSATION GROUP first will find a message from an unlocked conversation group, lock the conversation group, and then return the conversation group ID. The RECEIVE can then be used to get messages only for the conversation group that GET CONVERSATION GROUP locked, and thus is guaranteed not to block. If RECEIVE is used to indiscriminately read messages from a queue, it may become blocked if one of the messages it is trying to read from the queue is from a locked conversation group. Locking on a RECEIVE command can lead to decreased scalability and performance.

This may seem to be a bit of a complicated way to manage a transaction. Since in fact the queue is implemented as a table with messages in it, why not just start a transaction, select out a row—that is, a message—and then delete that row, all under the transaction? Functionally, this would work but would be extremely inefficient. The problem is that the queue in most cases will be holding messages that come from many different conversation groups. Locking the entire queue, which is what a `SELECT, DELETE` under a transaction would be doing, prevents the queue from being read or written by anyone else. This means that all processing of messages in the queue would be stopped, not just the processing of messages for a single instance. In addition, it would prevent all new messages from being added to the queue. `RECEIVE` in effect does the `SELECT` and `DELETE` under a transaction in a more efficient way.

Both `RECEIVE` and `GET CONVERSATION GROUP` are each specially designed so that, in effect, you can wrap message processing for a single conversation group in a transaction without affecting the processing of messages in other conversation groups.

So, putting conversation group locks all together, in typical usage a service program would first do a `BEGIN TRANSACTION`, then a `GET CONVERSATION GROUP`, then a `RECEIVE`. It would obtain a message that came from some conversation group. It could then process the message, knowing that no other service process could be processing another message from the same conversation group until it either does a `COMMIT TRANSACTION` or `ROLLBACK TRANSACTION`, but still allow messages in other conversation groups to be freely processed.

To see why being able to lock a conversation group is important, let's look at an example of what might happen if we couldn't lock a conversation group. We will look at a simple service that inserts a work order into a database. A work order usually has a header that includes the location where the work is to be done and some line items that indicate the tasks to be completed. In the database there is a table for headers and another for line items that uses referential integrity to link back to the header table. This service is like the one shown in Figure 15-1 at the beginning of this chapter in that it is designed to process many messages at once.

An application starts sending messages to the queue for the work order service. First it sends a message that contains the work order header, followed by a number of messages containing line items for the work order. The header is put into the queue first, and processing is started on it first. However, as soon as processing starts on the header, another instance of

the service process starts working on one of the line items. As luck would have it, the instance of the service process working on the line item finishes first and tries to insert the line item into the line item table, which fails because it violates referential integrity.

It turns out in the end, the work order would be properly inserted into the database because queues are transactional, and when the insert failed, the message would be put back into the queue and processed again later after the header had been inserted, but at the cost of a lot of overhead.

Now let's look at what happens with conversation group locks. The messages for the header and all the line items are in conversations that are in the same conversation group. The application always puts the message with the header into a queue first, followed by a message for each line item. One of the instances of the service process uses `GET CONVERSATION GROUP`, which, as it happens, locks the conversation group for a work order. It then uses `RECEIVE` to get all the messages in the queue associated with the conversation group it has just locked. This could include the header and a number of line items. It processes these in order by making `INSERT`s into the appropriate tables and then returns.

It may be that a second instance of the service process, running at the same time, also does a `GET CONVERSATION GROUP`. It may also lock a conversation group for a work order, but it will not be the same work order as the first instance locked. It will proceed to process this second work order in the same way the first instance is processing the first work order.

After the first instance of the conversation group finishes, it releases the lock on the conversation group. Then yet another instance of the service process does a `GET CONVERSATION GROUP`. It may end up locking the conversation group associated with the first work order and process subsequent line items that were put into queue after the first instance of the service processed completed.

There are many variations on this theme. For example, the service process might issue a second `RECEIVE` after it is done processing the first set of messages but before it completes the transaction, to see if any more messages have arrived for the conversation group it has locked.

There are two important things to note about this example. One is that every possible instance of a service process is running at the same time, each working on a different work order and doing so without ever blocking the others. The second is that a queue can continue receiving more messages for a conversation group even while that conversation group is locked. Neither of these things would be possible if a service process used the `BEGIN TRANSACTION`, `SELECT`, `DELETE` sequence to remove messages

from a queue. Conversation groups and their locks are crucial to the efficient operation of Service Broker.

Another important use of a conversation group is to have a way to maintain state across the conversations in a conversation group. There is always state to be maintained in a messaging application. A trivial example of this is a messaging application that is used to process a purchase order and has to keep track of the purchase order number. It can do this in three ways. One is to keep track of the purchase order number in memory, like a local variable. The second is to put the purchase order number in every message. And the last is to put the purchase order number in a table in SQL Server. Of course, if the state involved was a just purchase order number, almost any solution would work, but in real applications there is a lot more state than that.

The first option is not scalable and is very hard to manage. The more service programs there are, the more memory required to hold onto their purchase order numbers. This means that the memory requirements would be growing at a rate greater than the number of purchase orders being processed.

The second solution is reasonably easy to manage and does use SQL Server for storage, but it also is not scalable. The problem is that the storage required for purchase order numbers goes up as the number of messages increases. This in effect multiplies the amount of storage required in SQL Server by the number of messages involved, not just the number of purchase orders involved.

Both of the first two solutions also suffer from the problem of data being duplicated in many places. Of course, in practice this would be an unreliable way to maintain state.

What you really want to do is to put all the state for a conversation group into some tables in SQL Server. That way, there is only one copy of the state and just one place to maintain it. For this to work, however, you will need two things. Both are easy to get. First of all you need something to key the state you will be storing in SQL Server. The conversation group ID is unique and is a `UNIQUEIDENTIFIER`, so it is ideal to use for a key.

The second thing you need is a lock, and you have that too. As long as your service program is accessing a queue under a transaction, you can be sure other service programs are not touching the shared state. In fact, this is another use of the `GET CONVERSATION GROUP` command. Using `RECEIVE` locks the conversation group, but it also reads the queue. Sometimes you need to access or manipulate the state you are sharing within a conversation group before you read the queue. `GET CONVERSATION GROUP` gets the

conversation group associated with the next message in the queue and returns the conversation group ID, but it does not read the queue. In either case, you can then use the conversation group ID to look up the state and then decide whether or not the queue should be read.

Service Broker Application Guidelines

It is important to keep in mind what Service Broker does best when you are using it to develop an application. Service Broker works best when an application has a number of independent tasks to perform. If your application cannot be broken into a set of independent tasks, it is not a candidate for implementation with Service Broker.

The work order example in the previous section illustrates this. Service Broker can be configured to distribute the independent tasks over all the resources available when the load is light, and when the load is heavy, it can be configured to focus the resources on the critical tasks and defer others until the load lightens.

So the first thing you must do to use Service Broker is to break your application into independent tasks. Once you have done this, you must categorize each task as being critical or deferrable.

Critical means the task must be completed almost immediately when the application is invoked. One of the critical tasks might be to set up state so the overall progress of the application can be tracked.

A deferrable task is one that, of course, does not have to be completed immediately. In the work order example, you might decide that getting the header into the database right away is critical so that users would have some picture of where work was going to be done. However, the actual work order line items might be deferred because they would not be needed until work crews were assigned, which is done overnight.

If your application doesn't have any deferrable tasks, it is probably not a candidate for implementation in Service Broker.

Next you will have to define your services. There is no hard-and-fast rule for this, but you might start with a service, and its associated queue and stored procedure, for each kind of task in your application. Once your services are defined, you can implement the stored procedures, which in turn will create the conversations needed.

You will probably have one task that starts things off. Again, there is no hard-and-fast rule, but this should be a critical task that creates a conversation group and allocates the state that will be needed to track the progress of the application.

Service Broker Example

The sections that preceded this presented a conceptual overview of Service Broker and how it is used. What follows is an example of an application that uses Service Broker. It models a stock brokerage house, which offers to buy and sell stock to the public, and a stock trading house, which executes the actual trades on a stock exchange. A simple example designed to illustrate the use of Service Broker DDL and DML extensions in SQL Server 2005 follows.

Message Type

It is of vital importance that the sender and the receiver in a messaging application understand what messages will be sent. In Service Broker the description of the messages is defined in a `message type` object. The `message type` object defines the name of the message and the type of data the message contains. For each database that participates in a conversation, an identical `message type` is created.

Listing 15-1 shows the syntax for creating a message type.

LISTING 15-1: Syntax for Creating a Message Type

```
CREATE MESSAGE TYPE message_type_name
          [ AUTHORIZATION owner_name ]
    [ VALIDATION = {  NONE | EMPTY | WELL_FORMED_XML |
        VALID_XML WITH SCHEMA COLLECTION schema_collection_name } ]
```

The arguments and their definitions are as follows.

- `message_type_name`—The name of the message to create. It can be any valid SQL string. By convention, it has the form of `//hostname/pathname/name`. An example for a `message type` that deals with order entries could be `//www.develop.com/orders/orderentry`. Using this name in SQL Server would require square brackets around it—for example, `[//www.develop.com/orders/orderentry]`. Although using a URL format is not required, it's generally easier to ensure uniqueness if you use a URL.
- `AUTHORIZATION owner_name`—Defines which user or role owns the `message type`.
- `VALIDATION`—What kind of XML validation should be performed on received messages.

- `NONE`—The message isn't validated at all. This is done for non-XML messages or XML messages coming from a trusted source.
- `WELL_FORMED_XML`—The message is parsed to ensure the XML data is well formed.
- `EMPTY`—The message body is empty.
- `VALID_XML WITH SCHEMA COLLECTION schema_collection_name`—Specifies the XML schema collection to validate the message against. The schema collection must be registered in SQL Server before it can be used in the creation of the `message type`. (Chapter 8 covers how to register a schema collection in SQL server.) If `WITH schema_collection_name` is not used, the message will not be validated; however, the XML will still have to be well formed.

A broker must request that a brokerage make a trade, and that brokerage must acknowledge that request. Two messages will be required to do this:

- The message that is sent from the broker/trader to the brokerage containing the original order
- The acknowledgment message from the brokerage to the broker/trader

Listing 15-2 shows the two message types created in order to accomplish the order entry. Both use XML encoding, which means that any a message with valid XML will be processed.

LISTING 15-2: Example of Creating a Message Type with XML Encoding

```
-- first the message for the trade entry
CREATE MESSAGE TYPE
[//www.develop.com/DMBrokerage/TradeEntry]
VALIDATION = WELL_FORMED_XML

-- then the acknowledgment message
CREATE MESSAGE TYPE
[//www.develop.com/DMBrokerage/TradeAck]
VALIDATION = WELL_FORMED_XML
```

In Listing 15-2 the various endpoints that receive the messages with the `TradeEntry` and `TradeAck message type` don't care about the XML as such. They try to process it as long as it is well-formed XML. This may not

be an ideal situation for an enterprise application. In an enterprise application, you may want to make sure that the endpoints always get valid messages. For that purpose, the `message type` can be created indicating that the message should be validated against an XML schema. This is shown in Listing 15-3. The code listing shows both the code to register the schemas and how to refer to the schemas from the creation of the `message type`.

LISTING 15-3: Example of Creating Message Types Whose Messages Will Be Validated against XML Schemas

```
--create the schema collection for the tradeEntry message
CREATE XML SCHEMA COLLECTION TradeEntrySchema AS
N'<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace=
       "http://www.develop.com/DMBrokerage/schemas/tradeEntry">

      <xsd:complexType name="tradeEntry">
       <xsd:sequence>
         <xsd:element name="RICCODE" type="xsd:string"/>
         <xsd:element name="CustomerID" type="xsd:int"/>
         <xsd:element name="OrderID" type="xsd:int"/>
         <xsd:element name="Date" type="xsd:date"/>
         <xsd:element name="BuySell" type="xsd:date"/>
         <xsd:element name="Volume" type="xsd:int"/>
         <xsd:element name="Price" type="xsd:decimal"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>'

--create the message type based on the schema
CREATE MESSAGE TYPE
  [//www.develop.com/DMBrokerage/TradeEntry]
VALIDATION = VALID_XML WITH SCHEMA COLLECTION TradeEntrySchema

--create the schema collection for the tradeAck message
CREATE XML SCHEMA COLLECTION
   N'<?xml version="1.0" ?>
   "http://www.w3.org/2001/XMLSchema"
     targetNamespace=
         "http://www.develop.com/DMBrokerage/schemas/tradeAck">

     <xsd:complexType name="tradeAck">
       <xsd:sequence>
         <xsd:element name="OrderID" type="xsd:int"/>
         <xsd:element name="AckId" type="xsd:int"/>
       </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>'
```

```
-- create the message type for the trade ack
-- based on the schema collection
CREATE MESSAGE TYPE
[//www.develop.com/DMBrokerage/TradeAck]
VALIDATION = VALID_XML WITH SCHEMA COLLECTION TradeAckSchema
```

These `message types` need to be created in both the broker/trader database and the brokerage database. In a real-world application, there would be most likely be more `message types` to accomplish more tasks.

Apart from the message type the developer defines, there are some predefined message types in Service Broker. These three are the most common.

- `http://schemas.microsoft.com/SQL/ServiceBroker/DialogTimer`—A dialog can have an explicit timer assigned. This message is received when the timer expires.
- `http://schemas.microsoft.com/SQL/ServiceBroker/Error`—Service Broker creates error messages based on this message type to report errors to the application. This message type can also be used by the application to report errors or violation of business rules.
- `http://schemas.microsoft.com/SQL/ServiceBroker/EndDialog`—When a dialog ends, the broker sends the `EndDialog` message to the remote endpoint.

These message types are implicitly part of every contract, so any target can receive instances of them.

Changing Message Types

A `message type` can be altered or dropped using the normal T-SQL DDL syntax `ALTER` and `DROP`. The syntax to change a message type is shown in Listing 15-4.

LISTING 15-4: Syntax to Alter a Message Type

```
ALTER MESSAGE TYPE message_type_name
  [ VALIDATION = {  NONE | EMPTY | WELL_FORMED_XML |
        VALID_XML WITH SCHEMA COLLECTION schema_collection_name } ]
```

The `ALTER` syntax allows us to change the `VALIDATION` and/or the `AUTHORIZATION`. To drop a message type, you use this syntax: `DROP MESSAGE TYPE message_type_name [ ,...n ]`. Notice that the syntax permits dropping one or more `message types`. The only caveat with dropping a

message type is that to be dropped, it cannot be referenced from a contract. If that is the case, an error is raised saying that the message type cannot be dropped because it is referenced by one or more contracts.

Now that we have mentioned contracts, it is time to see what a contract is and how it is created.

Contracts

Service Broker services need to know what messages to expect, the outline of the messages, and what messages they can send. As we saw earlier, a message type defines the message, and we use a contract to define what messages each service (endpoint) can send and receive. Contracts are created and persisted in each database that participates in a conversation.

As we will cover later, the endpoints can be defined to be either the initiator or the target of a conversation. Subsequently, message types can be defined by the contract to be sent either by the initiator, the target, or both. Listing 15-5 shows the syntax to create a contract.

LISTING 15-5: Syntax for Creating a Contract

```
CREATE CONTRACT contract_name
[ AUTHORIZATION owner_name ]
( message_type_name
SENT BY { INITIATOR | TARGET | ANY } [ ,...n] )
```

The arguments and their definitions are as follows.

- contract_name—The name of the contract to create. It can be any valid SQL string. As with a message type name, you would enter it in the form of //hostname/pathname/name.
- message_type_name—The message type (or message types) that this contract uses.
- SENT BY—Defines which endpoint can send the defined message type. The possible arguments are INITIATOR, TARGET, or ANY.
- INITIATOR—The initiator of the conversation can send the defined message type.
- TARGET—The target of the conversation can send the defined message type.
- ANY—The specified message type can be sent by either the INITIATOR or the TARGET.

- AUTHORIZATION owner_name—Defines which user or role owns the contract. If this isn't specified, the contract is owned by the user who created the contract.

For the stock trading application, we need to create contracts in both the broker/trader database and the brokerage database. The TradeEntry message type initiates in the broker/trader database, and the TradeAck message type initiates in the brokerage database. The code in Listing 15-6 shows how to create the contract in the broker/trader database.

LISTING 15-6: Create a Contract

```
--create the contract against the message types
CREATE CONTRACT
   [//www.develop.com/DMBrokerage/EnterTrade]
   ( [//www.develop.com/DMBrokerage/TradeEntry]
      SENT BY INITIATOR,
   [//www.develop.com/DMBrokerage/TradeAck]
      SENT BY TARGET
   )
```

When a contract is created, at least one message type needs to be marked as sent by the INITIATOR.

Contracts, like message types, can be dropped. The syntax to drop a contract is this: DROP CONTRACT contract_name [,...n]. Notice that several contracts can be dropped through one DROP statement.

At this stage, according to the outline of how to design a Service Broker application, we should create the outline of the service program. We would like to wait with that and instead create the queues the application uses.

Queues

The queue is used to store the messages the endpoints send. When the service at one end sends a message to the service at the other end, the message is placed in a queue at the receiving end. Later, when the application receives the message and commits the transaction, the broker deletes the messages from the queue. The service broker manages the queues and presents a database table-like view of the queues.

The syntax to create a queue is shown in Listing 15-7.

LISTING 15-7: Syntax to Create a Queue

```
CREATE QUEUE queue_name
[ WITH  [ STATUS = { ON | OFF } ]
   [ RETENTION = { ON | OFF } , ]
   [ ACTIVATION (
      [ STATUS = { ON | OFF } , ]
      PROCEDURE_NAME = stored_procedure_name ,
      MAX_QUEUE_READERS = max_readers ,
      EXECUTE AS { SELF |  'user_name' }) ] ]
```

There are quite a few options when you are creating a queue, and a short explanation of the various arguments follows. The queue is the only Service Broker object that can be named with a three-part name.

- `queue_name`—Is the name of the queue that is created. This must be a SQL Server identifier. Because the queue is never referred to outside the database it is created in, the URL-like syntax used for other Service Broker names isn't necessary for queue names.
- `STATUS`—Decides whether the queue is created in a disabled state or not. The choices are `ON` (active) and `OFF` (disabled). When a queue is disabled, it cannot receive messages, nor can messages be removed from the queue. If this clause isn't specified, the queue is created in the `ON` state.
- `RETENTION`—Specifies the retention setting for the service. If `RETENTION = ON`, all messages sent or received on conversations using this service are retained in the queue until the conversations have ended successfully. The `RETENTION` argument is useful if you do compensating transactions. With `RETENTION = ON`, you can do a compensation transaction if something goes wrong during the conversation (remember that the conversation can have a very long lifespan), since the messages are kept in the queue until the conversation ends.
- `ACTIVATION`—Specifies information about the stored procedure that will be activated to handle messages that arrive on this queue. If `STATUS` is set to `OFF`, the queue does not activate the stored procedure; the default is `ON`. We cover different aspects of activation of the service programs later in this chapter.
- `PROCEDURE_NAME`—Is the stored procedure to execute. This procedure is also the service program for the application. The procedure has to be in the same database as the queue or be fully qualified.

- MAX_QUEUE_READERS—When a message arrives on the queue, the procedure will be activated. As more messages build up in the queue, more instances of the procedure will be activated, up to MAX_QUEUE_READERS.
- EXECUTE_AS—Specifies what SQL Server login the activated procedure runs under. If this optional clause is set to SELF, the procedure runs under the user who created the queue.

In the stock trading application example, we need one queue in the brokerage database to handle the order entry messages coming from the broker/trader database. We also need a queue in the broker/trader database to handle the acknowledgment messages from the brokerage database. In a real-world application, we would probably have more queues to handle different messages. Feel free to create as many queues as you deem necessary for your application.

The code to create the queues for the stock trading application is shown in Listing 15-8. Notice that if you want to automatically activate a stored procedure (the service program) when a message arrives on the queue, you need to supply the PROCEDURE_NAME clause with the name of a valid procedure. The easiest approach is to create a procedure that is just an empty shell, as in the following code snippet, and catalog it in SQL Server.

```
--create in the broker/trader database
CREATE PROCEDURE tradeAckProc
AS

RETURN 0
GO

--create in the DMBrokerage database
CREATE PROCEDURE tradeEntryProc
AS

RETURN 0
```

LISTING 15-8: Creation of Queues

```
--create the queue in the broker/trader database
--this queue handles the acknowledgments from
--the brokerage
USE Trader1
GO

CREATE QUEUE tradeAckQueue
WITH STATUS = ON,
ACTIVATION (
```

```
  PROCEDURE_NAME = tradeAckProc,
  MAX_QUEUE_READERS = 5,
  EXECUTE AS SELF)

--creation of the queue in DMBrokerage
--this queue handles the new orders coming from
--the broker
USE DMBrokerage
GO

CREATE QUEUE tradeEntryQueue
WITH STATUS = ON,
ACTIVATION (
  PROCEDURE_NAME = tradeEntryProc,
  MAX_QUEUE_READERS = 5,
  EXECUTE AS SELF)
```

The queues created in Listing 15-8 act as receive queues for replies and error messages.

When you create a queue, you create an object of the type `Service Queue`. This object maps to a SQL Server internal table with the same name as the queue. To view what queues exists in a database, you can do a `SELECT` against the `sys.service_queues` catalog view. You can view the content of a queue through a simple `SELECT` statement: `SELECT * FROM queue_name`. Issuing a `SELECT` against one of the created queues in Listing 15-8 results in an empty resultset, but at least you can see the columns. Table 15-1 shows the content of a queue.

In the process of developing a Service Broker application, we now have the "basic plumbing," which consists of the following:

- Message types
- Contracts
- Queues

These may exist in different databases on different servers. We now need to create the information about where the messages are sent. This is handled by the services. Table 15-1 shows some columns that hold information about services, and in the following section we cover how to create services.

TABLE 15-1: Columns in a Queue

Column Name	Data Type	Description
`status`	`tinyint`	Status of the message (0=Ready, 1=Dequeued, 2=Disabled).
`priority`	`tinyint`	Reserved for future use.
`queuing_order`	`bigint`	Message order number within the queue.
`conversation_group_id`	`uniqueidentifier`	Conversation group identifier for the message.
`conversation_handle`	`uniqueidentifier`	Conversation identifier for the message.
`message_id`	`uniqueidentifier`	Message identifier.
`message_sequence_number`	`bigint`	Sequence number of the message within the conversation.
`service_name`	`nvarchar(512)`	Name of the service that this message targets.
`service_id`	`int`	The `object_id` of the service that the message targets.
`service_contract_name`	`nvarchar(256)`	Name of the contract that the message follows.
`service_contract_id`	`int`	The `object_id` of the contract that the message follows.
`message_type_name`	`nvarchar(256)`	Name of the message type that describes the message.
`message_type_id`	`int`	The `object_id` of the message type that describes the message.
`validation`	`nchar`	What validation is done before the message is put on the queue; one of (N = None, X = XML, E = Empty).
`message_body`	`varbinary(MAX)`	Content of the message. Note that this is a binary column; you will have to cast it to a readable type to see what it contains: `CAST(message_body as XML)`.

Services

A service is an endpoint for specific functionality in the Service Broker application. Based on the service name, Service Broker routes messages between databases and puts the messages on the queue for that particular service and message type. The specific functionality that the service is an endpoint for is defined by the contract. By specifying the contract, the service indicates it serves as a target for that particular functionality.

Having said this, we can see that a service:

- Defines which queue to receive messages on.
- Defines what contracts to support.

Therefore, when a service is created, we need to map it to an existing queue and, optionally, to one or more contracts. The syntax to create a service is shown in Listing 15-9.

LISTING 15-9: Syntax for Creating a Service

```
CREATE SERVICE service_name
[ AUTHORIZATION owner_name ]
ON QUEUE queue_name
[ ( contract_name [ ,...n ] ) ]
```

The arguments for `CREATE SERVICE` are as follows.

- `service_name`—Is the name of the service to create.
- `queue_name`—Specifies the queue that receives messages for the service.
- `contract_name`—Specifies a contract that this service exposes. Notice that when a contract is specified, it does not mean that the particular contract is exclusive to the service. Other services can also use the same contracts, and the service can send messages on contracts that are not specified here.
- `owner_name`—Sets the owner of the service to the name of a database user or role.

The syntax specifies both a queue name and a contract name. By defining those arguments, we make sure that any message(s) based on the defined contract(s) are delivered to that particular queue.

In the stock trading application, we now have two message types, one contract, and one queue in each participating database. Listing 15-10 shows the code to tie this together. The code creates a service in each database that maps to a queue and a contract.

LISTING 15-10: Code to Create Services

```
--create the service in the trader db
USE Trader1
GO

CREATE SERVICE enterTrade
ON QUEUE tradeAckQueue
([//www.develop.com/DMBrokerage/EnterTrade])

--create a service in the brokerage db
USE DMBrokerage
GO

CREATE SERVICE
    [//www.develop.com/DMBrokerage/TradeEntryService]
      ON QUEUE tradeEntryQueue
([//www.develop.com/DMBrokerage/EnterTrade])
```

Figure 15-5 illustrates the interaction between services, messages, and queues.

When you look at Figure 15-5, you can see how messages are sent between the services and queues. In the following section, we'll look at how to initiate the message exchange.

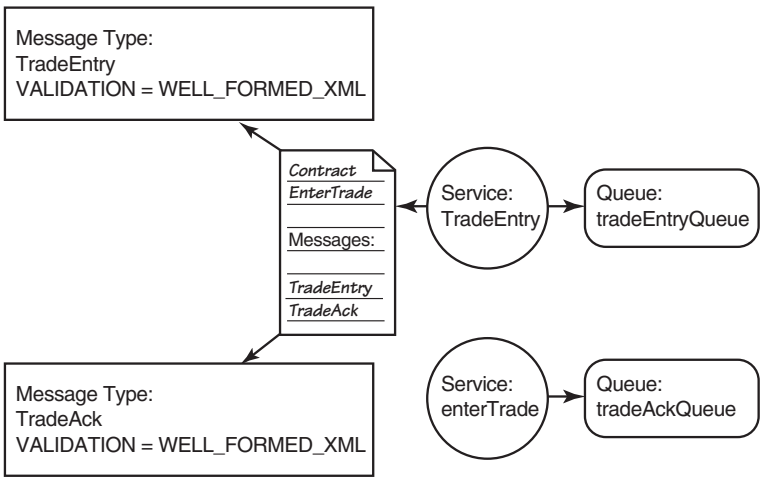

FIGURE 15-5: Interaction between Services, Messages, and Queues

Dialogs

Service Broker applications communicate through *conversations.* A conversation involves endpoints communicating with each other. Theoretically, the conversation can be one-to-one, one-to-many, or even many-to-many. However, at this time Service Broker only supports one-to-one conversation. This type of conversation is called a *dialog.* A dialog is communication between exactly two endpoints. It is a logical connection between service programs, which run on service brokers. The dialog ensures that any messages associated with a dialog are delivered exactly once and in the order in which they were sent throughout the lifetime of the dialog. This is an extremely important point because the lifetime of the dialog can span several transactions. Other messaging applications guarantee in-order delivery within a transaction but not spanning multiple transactions.

The dialog ensures in-order delivery by sequence numbering of the messages. The sending endpoint assigns a sequence number to the message. This sequence number is used by the receiving endpoint to order the messages correctly. If a message is received out of order, Service Broker holds on to the message until the missing messages have arrived. At that time, the out-of-order message is put on the queue.

In a messaging application, it may be of importance to correlate messages from endpoints if there are multiple dialogs. In other words, you want to tell which received messages correspond to which sent messages. The dialog handles correlation automatically for you. The correlation of messages is handled by a unique identifier of the conversation: the `conversation_handle`.

Another important part of dialogs is the message acknowledgment. Dialogs incorporate automatic message acknowledgment for all messages with a sequence number. When a message is sent, the sending broker keeps the message on the transmission queue until an acknowledgment has been received from the remote broker.

Before an application starts sending messages, it needs to establish a conversation. It creates a dialog. At this time it needs to indicate what endpoints are involved and what contract to use. Therefore, the syntax to start communication is shown in Listing 15-11.

LISTING 15-11: Syntax to Begin Communication

```
BEGIN DIALOG [ CONVERSATION ] dialog_handle_identifier
   FROM SERVICE service_name
   TO SERVICE 'service_name' [ , instance_identifier ]
   ON CONTRACT contract_name
```

```
[ WITH
[  { RELATED_CONVERSATION = conversation_handle
   |  RELATED_CONVERSATION_GROUP = conversation_group_id } ]
[ [ , ] LIFETIME = dialog_lifetime ]
[ [ , ] ENCRYPTION = { ON | OFF }  ] ]
```

The arguments for `BEGIN DIALOG CONVERSATION` are as follows.

- `dialog_handle_identifier`—When a dialog is created, the system assigns it a unique dialog handle. This is the variable to store that handle in.
- `FROM SERVICE service_name`—Specifies the service that is initiating the dialog. This is the service that will receive response and error messages from the target.
- `TO SERVICE 'service_name'`—Specifies the target service with which to initiate the dialog.
- `instance_identifier`—Specifies the database that hosts the target service.
- `ON CONTRACT contract_name`—Specifies the contract that defines messages used in this conversation.
- `RELATED_CONVERSATION = conversation_handle`—This dialog can be associated with the conversation group of an existing dialog through the `conversation_handle` variable.
- `RELATED_CONVERSATION_GROUP = conversation_group_id`—When starting a new conversation, Service Broker also creates a new conversation group (we cover conversation groups later). This argument allows us to associate the dialog with an existing conversation group instead of creating a new conversation group.
- `LIFETIME = dialog_lifetime`—Specifies the lifetime in seconds for the dialog. If this is not specified, the dialog remains open until it is explicitly closed.
- `ENCRYPTION`—Specifies whether messages sent and received on this dialog must be encrypted. The default for `ENCRYPTION` is `ON`. Having `ENCRYPTION` set to `OFF` does not necessarily mean that messages won't be encrypted. That depends on the existence of certificates, which are covered later in this chapter.

It is worth noting a couple of things about the `TO SERVICE` argument. The service name variable must be a string. The variable must be quoted and match the name of the remote service, including case.

You should also notice that the RELATED_CONVERSATION and RELATED_CONVERSATION_GROUP clauses do the same thing. They link the dialog created to an existing conversation group. You would use the RELATED_CONVERSATION_GROUP if you created your own UNIQUEIDENTIFIER for the conversation group ID. By using RELATED_CONVERSATION, you link the dialog to the conversation group ID of the specified conversation_handle.

We can also define what instance to target. This is important because we can have deployed the service to several databases in the same instance of SQL Server, or to databases in other instances/remote servers. The instance_identifier specifies which database we want to target. The instance_identifier is the service_broker_guid for the database. You retrieve the UNIQUEIDENTIFIER through the following syntax.

```
SELECT service_broker_guid
FROM sys.databases
WHERE database_id = db_id('<db_name>')
```

An interesting question is, what happens if we have deployed the service to several databases and we have not specified an instance_identifier? In this case, Service Broker picks randomly which service to target.

Let's now go back to the stock trading application and look at the syntax shown in Listing 15-12 to create a conversation between our services.

LISTING 15-12: Start a Conversation in the Stock Trading Application

```
--declare a variable for the conversation handle
DECLARE @dh uniqueidentifier;
--this starts the dialog from the
--broker/trader database
--and we get a dialog/conversation handle automatically
BEGIN DIALOG CONVERSATION @dh
  FROM SERVICE enterTrade
  TO SERVICE
     '//www.develop.com/DMBrokerage/TradeEntryService'
  ON CONTRACT
  [//www.develop.com/DMBrokerage/EnterTrade];
```

The code in Listing 15-12 is run from the broker/trader database and causes an entry in the conversation endpoints table to be made. This can be investigated by calling SELECT * FROM sys.conversation_endpoints. At this stage, nothing has happened in the brokerage database yet. Nothing will happen until we send a message. This can be verified by running SELECT far_broker_instance FROM sys.conversation_endpoints in

the broker/trader database. `far_broker_instance` is a column that holds the ID for the remote server. The `SELECT` returns `NULL`; in other words, Service Broker has not yet decided which endpoint to talk to. The decision about the endpoint happens when the first message is sent for this particular dialog.

The code in Listing 15-12 does not explicitly set a lifetime for the dialog; the dialog is alive until it has been closed explicitly. The syntax to close a dialog follows.

```
END CONVERSATION conversation_handle
  [ [WITH ERROR = failure_code DESCRIPTION = failure_text]
   |[WITH CLEANUP]
  ]
```

The `conversation_handle` variable is obtained through the `BEGIN DIALOG CONVERSATION` call. You may want to end the dialog if there is an error during the dialog lifetime. To do this, call `END CONVERSATION` with an error number and an error description. When you end a conversation, a message will appear on the receiving service queue of the type `EndDialog`. The message type will be `Error` if you have ended the conversation through the `WITH ERROR` option.

You saw in the syntax for `BEGIN DIALOG CONVERSATION` (Listing 15-11) that you can set an explicit lifetime on the `DIALOG`. When you set the lifetime for the dialog and the timeout happens, the dialog ends and an error message is put on the target queue and the initiator queue. At this stage, you cannot use that particular dialog again. In other words, if you set lifetimes, the initiating service should know the expected lifetime of the dialog and be fairly certain that the dialog actually ends before the timeout.

However, sometimes the initiating service does not know the expected lifetime, or the initiator just wants to know that something takes longer than expected but does not want to end the dialog. The initiator wants to be notified when an expected timeout has been exceeded, but the dialog should not end. In this scenario, you can use a `CONVERSATION TIMER`. The `CONVERSATION TIMER` allows you to start a timer on a particular conversation handle. When the timer expires, you get a `Timeout` message on the local queue for that dialog, but the dialog does not end. When the timeout message arrives, the service will be activated to handle the timeout appropriately. The syntax for starting the timer follows.

```
BEGIN CONVERSATION TIMER (conversation_handle)
TIMEOUT = timeout
```

The `conversation_handle` variable is the unique identifier for the conversation, and the `TIMEOUT` is set in milliseconds.

In this section about dialogs, we have seen how we (and Service Broker) keep track of the various conversations through the `conversation_handle` identifier. The question is then how to keep track of several related conversations with different identifiers that belong to the same service. Enter the *conversation group.*

Conversation Group

You use the conversation group to group related conversations together. Imagine that when an order is entered in our application, we need to do more things than just notify the brokerage. We may have to check the client's credit, check against some authority that the client actually is allowed to trade, and so on. In this scenario, we would probably start several different dialogs. These dialogs would get different conversation handles, and it might be hard for us to keep track of the different conversations. Fortunately, Service Broker comes to help. When we start a new dialog, it creates a new conversation group identifier automatically. The identifier is a GUID (SQL server data type `UNIQUEIDENTIFIER`). The identifier is appended to the messages we receive. You can create the identifier yourself (in T-SQL you use `NEWID()`). Subsequently, when you do `BEGIN DIALOG CONVERSATION`, you relate the dialog to the identifier using the syntax in Listing 15-11. Relating the dialog to a conversation group identifier is the solution if you want to use an existing identifier for your dialog or associate your dialog with the conversation group of an existing dialog.

To obtain the identifier within a conversation, you do a `SELECT` against the `conversation_group_id` column in the message queue. The identifier can also be retrieved by the `GET CONVERSATION GROUP` call. Calling `GET CONVERSATION GROUP` gets the conversation group identifier for the next message to be retrieved. As we will see later, the conversation group identifier is useful if we want to keep state information. In addition, it puts a lock on the instance. See the SQL Server Books Online for the full syntax for `GET CONVERSATION GROUP`.

The biggest benefit of the conversation group is that of locking the dialogs. You may ask why it is important to lock dialogs. We have already stated that Service Broker guarantees the in-order delivery of messages. That is true, but the issue is that a queue can have multiple readers (this is discussed more in the Activation section later in the chapter). In other words, we have multithreaded queue readers.

The problem with this is that there is parallel processing of messages. If the messages are parent-child data (think orders with order lines), a situation can occur in which even if the messages appear on the queue in order, they can be processed out of order. Think about a scenario where it takes longer to process the parent data than the child data.

In this scenario a parent message arrives first, followed by child messages. Queue reader 1 (qr1) gets instantiated and starts processing the parent data. Shortly thereafter, a child message arrives, and because qr1 is still active, a second queue reader gets activated (qr2). Because the parent data takes so much longer to process than the child data, the child data may try to commit before the parent, and there will be a referential integrity constraint violation. Obviously, the transaction rolls back, and the message is put back on the queue and can be processed later, but this is not optimal.

In Service Broker, when a receive is done, a lock is put on the conversation group, and no other queue readers can receive messages on dialogs in that particular conversation group until the transaction is committed.

A conversation group is also beneficial when you want to keep application state. You can store any state data in the database based on the conversation group identifier and retrieve it when needed, because every message you receive will have the conversation group ID in the result. For those of you who are ASP developers, you can see the conversation group ID as a cookie. Naturally, the lifetime of a conversation group is important if you rely on it for state data. A conversation group is alive as long as it has conversations associated with it.

Service Programs

Figure 15-5 illustrates the process for a new trade in our application.

- A user enters a trade through a user interface.
- A stored procedure in the `Trader` database processes the trade entry and creates a message.
- The stored procedure sends the message, and Service Broker puts the message in the `tradeEntryQueue` in the `DMBrokerage` database.
- The `tradeEntryProc` procedure is activated and processes the message from the queue.
- When the message is processed, the `TradeEntryService` sends a reply to the `enterTrade` service on the `tradeAckQueue`.
- The procedure on the `tradeAckQueue` is activated and processes the message.

These steps are done through service programs, the part of the application that processes messages for the application. A service program is typically a stored procedure, which is activated when a message arrives on a queue. In this case, we say the service program is the *target*. So, if a service program acts as a target, then we also need something that starts a message exchange. Therefore, a service program can be an *initiator,* which sends the first message to a target. A service program can also be the initiator of one dialog and the target of another. A service program could theoretically also be the initiator and the target of the same dialog. This could be used when you're doing some time-critical processing and you may want to queue up some work to do later when you have time.

In our stock trading application, the initiating service program is the stored procedure that is invoked when a user places an order. The target is the stored procedure on the `tradeEntryQueue` in the `DMBrokerage` database. There is an additional target in our application. It is the stored procedure in the `Trader` database that accepts the acknowledgments of the trades on the `tradeAckQueue`.

We mentioned earlier that a stored procedure is *activated* when a message arrives on a queue. In the following section, we will look a little more closely at the activation features in Service Broker.

Activation

In a traditional messaging application, you have basically two options to find out that a message has arrived on a queue.

- You poll against the queue.
- The messaging infrastructure exposes some sort of event that an application listens for.

Service Broker differs in some respect from this—not so much for the polling scenario, because we can poll a queue through either T-SQL or some external program. However, for events, it looks different.

When you rely on events, you normally need to have a program running that is listening for events. In Service Broker you do not need to do this. Service Broker introduces an activation mechanism.

The activation in Service Broker is based on the `CREATE queue` syntax. Remember from Listing 15-7 how the syntax takes some optional `ACTIVATION` arguments. The interesting ones are `PROCEDURE_NAME` and `MAX_QUEUE_READERS`. As we mentioned in the section about queues, the `PROCEDURE_NAME` argument defines which stored procedure to activate

when a message arrives on that particular queue. The `MAX_QUEUE_READERS` argument defines the maximum number of stored procedures that should be activated. The way it works is as follows.

When SQL Server starts, the internals of SQL Server know about the queues in the instance and what queues have activation procedures defined. SQL Server starts to monitor these queues. When the first message arrives on any of the monitored queues, the activation procedure is started to process the message. If messages are put on the queue faster than the procedure can process them, new procedures are started. This continues until the load is handled or the `MAX_QUEUE_READERS` number has been reached. When the `MAX_QUEUE_READERS` number has been reached, the messages are queued up on the queue until a service program is available to process the messages.

To make this work with the best performance possible, you need to bear in mind a couple of things when designing your service programs (stored procedures).

- Make sure the program reads messages from the correct queue. If it doesn't, the program will be killed and the messages will queue up.
- The activation monitor code has no way of knowing when a program has finished executing, apart from noticing that the program has terminated. Therefore, make sure the program exits soon after the queue is empty.
- There may be several message types on any given queue. Make sure the program can handle all message types.

In the last bullet item, we said that the service program should be able to handle all message types. You may ask what all message types are. "All message types" means the following:

- All message types marked `TARGET` or `ANY` in the contracts defined by the service or services that use the queue.
- All message types marked `INITIATOR` or `ANY` in the contracts for the conversations that the service program initiates.
- At least `Error` and `EndDialog`. If the program sets a conversation timer, the program should handle `Timeout` messages, too.

Activation happens when a message is received. Now it is time to look at how messages are exchanged.

Sending and Receiving Messages

When a dialog has been created, your only choices are to send a message, end it, or start a dialog timer. The Listing 15-13 shows the syntax for sending messages. Notice that if SEND is not the first statement in the batch or stored procedure, it has to be preceded with a semicolon (;).

LISTING 15-13: Syntax to Send a Message

```
SEND
   ON CONVERSATION conversation_handle
   MESSAGE TYPE message_type_name
   [ ( message_body_expression ) ]
```

The SEND command takes a conversation_handle variable, which is obtained either from the BEGIN DIALOG CONVERSATION statement or from the conversation_handle column in the message queue. The message_type_name variable must be of a message type that is included in the contract for the conversation. The third argument is the message body. SEND does an explicit cast of the message body to varbinary(MAX), so it will take any valid data type as an argument. If the encoding is EMPTY, the message has no body.

For the stock trading application, we have a stored procedure in the broker/trader database that processes the trade entries. This stored procedure also initiates a dialog and sends a message to the brokerage database. Listing 15-14 shows the part of the stored procedure that creates the dialog (according to Listing 15-12) and sends the message.

LISTING 15-14: BEGIN DIALOG CONVERSATION and Send the Message

```
--code to process the incoming trade omitted

--declare variable for the conversation handle
DECLARE @dh uniqueidentifier;
--variable to hold the message in XML format
DECLARE @msgBody XML

--set the message, in real world you'd create
--it from the in params in the proc
--here we just set it to something
SET @msgBody = '<id>Order1</id>'

--begin the dialog
BEGIN DIALOG CONVERSATION @dh
```

```
  FROM SERVICE enterTrade
  TO SERVICE '//www.develop.com/DMBrokerage/TradeEntryService'
  ON CONTRACT [//www.develop.com/DMBrokerage/EnterTrade];

--send the message we use the conversation handle from
--BEGIN DIALOG CONVERSATION
SEND ON CONVERSATION @dh
  MESSAGE TYPE [//www.develop.com/DMBrokerage/TradeEntry]
  (@msgBody)
```

To test if it works, follow these steps.

1. Create two databases (it can be done in one, but it is more realistic in two). Name them, for example, `Trader1` and `DMBrokerage`.
2. Create the message types and contracts from Listings 15-2 and 15-6 in both databases.
3. Create the `tradeAckQueue` in `Trader1` and the `tradeEntryQueue` in `DMBrokerage` according to Listing 15-8. When you create the queues, do not set any `ACTIVATION` arguments just yet.
4. The last thing to do before you can test is to set up the services as in Listing 15-10.
5. When you finish the preceding steps, you can run the code in Listing 15-14 from the `Trader1` database.

At this stage in the `Trader1` database, there is an entry in both `sys.conversation_groups` and `sys.conversation_endpoints` catalog views. The record in `sys.conversation_endpoints` holds information about the conversation, the conversation group, and the endpoints. There are similar entries in the `sys.conversation_groups` and `sys.conversation_endpoints` catalog views in the `DMBrokerage` database. The actual message is in the `tradeEntryQueue` queue, and you can view it by doing `SELECT * FROM tradeEntryQueue`. Selecting against a queue does not affect the messages in the queue. To remove messages from a queue, you use the `RECEIVE` command.

The full syntax for `RECEIVE` is shown in Listing 15-15. Note that if `RECEIVE` is not the first statement in the batch or stored procedure, it has to be preceded with a semicolon.

LISTING 15-15: RECEIVE Syntax

```
[ WAITFOR ( ]
   RECEIVE [TOP (n)]
      < column_specifier > [ ,...n ]
      FROM queue_name
      [INTO table_variable ]
      [WHERE { conversation_handle = conversation_handle
      |conversation_group_id = conversation_group_id } ]
[ )
[ , TIMEOUT timeout ] ]
```

The syntax looks almost exactly like SELECT, and both SELECT and RECEIVE return a resultset. The difference, as we mentioned earlier, is that RECEIVE removes the message(s) from a queue, whereas SELECT leaves them on the queue.

The WAITFOR argument in the RECEIVE syntax indicates that the RECEIVE operation is to wait for a message to arrive on the queue if the queue is empty or the WHERE criteria doesn't return a result. TIMEOUT can only be used together with WAITFOR, and it indicates, in milliseconds, how long to wait for a message to arrive. If WAITFOR is specified and TIMEOUT is –1 or TIMEOUT is not specified, the wait is unlimited. If a timeout occurs, the RECEIVE statement returns an empty result.

Because WAITFOR is optional, you may ask yourself whether you should use it or not. In messaging applications in general, it is considered good practice to use WAITFOR (or equivalent statements). One scenario where you probably would not use WAITFOR is when your service program (where your receive code is) is activated by an incoming message, and you are certain that no other messages will be arriving within the time it takes to process a message plus the time it takes to activate a new instance of the stored procedure. If the volume is high, it is better use a TIMEOUT value (fairly short). The reason is that it probably does not make sense to start a new service program for each new message. On the other hand, if the service program is an external application and not activated by Service Broker, you should use a reasonably long TIMEOUT.

Since the RECEIVE command allows you to do a RECEIVE with a TOP clause, should you consider receiving message by message or multiple messages? In most cases, you need to process the messages on a message-by-message basis. Bearing this in mind, if you are using T-SQL, you are probably better off doing a RECEIVE TOP(1), especially since you cannot create a cursor of the result from a RECEIVE. You would have to retrieve the messages into a table variable and create the cursor over that variable. If

you receive messages into an external service program, you are better off, from a performance perspective, receiving a resultset of multiple messages.

Listing 15-16 shows the code to receive the conversation handle and the message body from the first message in the `tradeEntryQueue` as a resultset.

LISTING 15-16: RECEIVE from the Queue

```
--receive the first message on the queue
RECEIVE  TOP(1) conversation_handle,
      message_body FROM tradeEntryQueue
```

When you run the code in Listing 15-16 from the `DMBrokerage` database, it retrieves the message you sent in Listing 15-14. If you do a `SELECT` against the `tradeEntryQueue` after the `RECEIVE`, no messages are there. Notice that the body of the message is output as `VARBINARY`. If you want to view it as readable text, you have to cast it to another type, like this.

```
RECEIVE CAST(message_body AS XML) FROM MyQueue
```

Listing 15-17 shows a code snippet that uses the `WAITFOR` and `TIMEOUT` arguments. The `TIMEOUT` is set to one minute (60,000 milliseconds).

LISTING 15-17: RECEIVE with WAITFOR and TIMEOUT

```
--declare a variable to hold the conversation handle
DECLARE @dh UNIQUEIDENTIFIER;

--declare a variable for the message body
DECLARE @msg VARBINARY(max)

WAITFOR(
--receive the first message on the queue
RECEIVE  TOP(1) @dh=conversation_handle,
       @msg=message_body FROM tradeEntryQueue),
TIMEOUT 60000

SELECT @dh, @msg
```

You can test the code in Listing 15-17 by first executing the code in the `DMBrokerage` database. The status bar in SQL Server Management Studio will say "Executing Query." Switch over to the broker/trader database and execute the code in Listing 15-14. Switch back to the `DMBrokerage` database again, and you can see that a message has been removed from the queue.

Flow in a Service Program

At this point, all code necessary for the stock trading application is done. It is time to tie it together and look at the work flow in a service program. We have discussed how service programs can have different roles when it comes to conversations: initiators, targets, or both. A pure initiator is not that interesting, because it only begins a conversation and goes away.[1] Look at Listing 15-14 for an example of an initiator program.

The interesting role is the target, and most service programs that act as a target follow a common process model. This model looks very much like Windows message loop programming. Listing 15-18 illustrates this through the code for the stored procedure that is activated in the `DMBrokerage` database when a message arrives on the `tradeEntryQueue`.

LISTING 15-18: Code for Service Program in DMBrokerage

```
CREATE PROCEDURE tradeEntryProc
AS

--declare variables
DECLARE @dh           UNIQUEIDENTIFIER
DECLARE @msg          XML
DECLARE @ack          XML
DECLARE @cg           UNIQUEIDENTIFIER

WHILE (1=1)
BEGIN

-- 1. start a transaction
BEGIN TRAN;
SET @cg = NULL;


-- 2. Lock the conversation group if
-- dealing with state data
WAITFOR (
GET CONVERSATION GROUP @cg
FROM tradeEntryQueue
),
TIMEOUT 10000

--check that we have a message
IF @cg IS NULL
```

[1] Even though a pure initiator service doesn't expect actual data to return, it should be prepared to handle error messages and end-dialog messages. For this purpose, even a pure initiator should have an activation procedure attached to the initiator queue.

```
  BEGIN
    ROLLBACK TRANSACTION
    BREAK
  END;

-- 3. Do what is necessary to deal with state

-- 4. RECEIVE
-- we need the conversation handle from the send
  ;RECEIVE TOP(1) @dh=conversation_handle,
         @msg=message_body
  FROM tradeEntryQueue
  WHERE conversation_group_id = @cg

-- 5. process received data
-- code omitted that deals with the received data
-- here we probably create the message to send
-- back as well

-- in our case we just hardcode something
SET @ack='<id>1</id><ackid>1</ackid>'

-- 6. send the message
;SEND ON CONVERSATION @dh
  MESSAGE TYPE [//www.develop.com/DMBrokerage/TradeAck]
  (@ack)

-- 7. end the conversation
END CONVERSATION @dh

-- 8. Update eventual state data
-- do some stuff to deal with state

-- 9. Commit or roll back
COMMIT TRAN
END
```

The reason we compare this flow with Windows message loops is that when the process is done, it starts all over again. Let's look at the various parts of the model.

- *Begin transaction*—The transactional model is one if the biggest benefits with Service Broker, compared with other messaging systems. In Service Broker the messaging and the database share the same transactional engine, which is very rare in other systems. Everything done against the database, which is associated with a message, should be part of a transaction. If the transaction rolls

back, the database changes are rolled back and the removed messages are put back on the queue. The outgoing messages are not sent, and we can start all over again.

- *Lock conversation group*—Applicable if we deal with application state. The conversation group will under all circumstances be locked when the `RECEIVE` happens.
- *Retrieve state data*—If applicable.
- *Retrieve messages*—For this application, we have decided to use a fairly short `TIMEOUT` and retrieve one message per `RECEIVE`. We need the conversation handle in order to send messages back to the initiator.
- *Process data*—In a real-world application, we would probably receive messages for different message types on the same queue. We need, therefore, to check what message type we receive and process the message accordingly. For this sample, we suggest that you just insert the message body in some table in the database.
- *Send data*—In our application, we need to process the incoming data before we can send. There is nothing that says, however, that a send has follow a receive. It can be anywhere in the model.
- *End conversation*—A conversation should always be ended at one stage. It does not need to be ended when the service program exits, if it makes sense to keep it alive. In our example, this is the last message for this particular task, so it makes sense to end.
- *Update state*—If applicable.
- *Commit transaction*—This is where it happens. Database updates are committed, messages are sent and received messages are taken off the queue.

To see this in action, you need to change the stored procedure in Listing 15-18 so it does something with the processed data. Then it needs to be cataloged in the `DMBrokerage` database. `ACTIVATION` arguments need to be added to the `tradeEntryQueue` with the following code snippet.

```
ALTER QUEUE tradeEntryQueue
WITH STATUS = on,
ACTIVATION (
  PROCEDURE_NAME = tradeEntryProc,
  MAX_QUEUE_READERS = 5,
  EXECUTE AS SELF)
```

Create a stored procedure in the broker/trader database that functions as the service program for messages on the `tradeAckQueue`. You can see an example of this in Listing 15-19. Note that you need to add some code to handle the received messages.

LISTING 15-19: Code for the Service Program in the Broker/Trader Database

```
CREATE PROCEDURE tradeAckProc
AS

--declare variables
DECLARE @dh           UNIQUEIDENTIFIER
DECLARE @msg          VARBINARY(max)
DECLARE @cg           UNIQUEIDENTIFIER
DECLARE @mt           NVARCHAR(max)

WHILE (1=1)
BEGIN

BEGIN TRAN;
SET @cg = NULL;

WAITFOR (
GET CONVERSATION GROUP @cg
FROM tradeAckQueue
),
TIMEOUT 10000

IF @cg IS NULL
BEGIN
  ROLLBACK TRANSACTION
  BREAK
END;


  RECEIVE TOP(1) @dh=conversation_handle,
        @msg=message_body,
        @mt = message_type_name
  FROM tradeAckQueue
  WHERE conversation_group_id = @cg


-- process received data
-- code omitted that deals with the received data

END CONVERSATION @dh

COMMIT TRAN
END
```

Make sure that the queue in the broker/trader database (Listing 15-8) has its activation arguments set to use the stored procedure in Listing 15-19. Run the code in Listing 15-14 and notice how the stored procedures were activated and handled the messages.

At this stage, messages have been exchanged between different databases in the same server instance. What about message exchange between different instances or different machines altogether? In order to achieve message exchange between instances and/or machines, we need to discuss routes and remote service bindings.

Routes

In the Dialogs section earlier, we mentioned how entries are added in the `sys.conversation_endpoints` table when a dialog is started through the `BEGIN DIALOG CONVERSATION` syntax. We also mentioned that at that time the endpoint has not been resolved to a physical location; this happens when the first message is sent.

The way Service Broker resolves endpoints is by using routes, where a route is an entry in a routing table (`sys.routes`) in a specific database and/or in `MSDB` (`msdb.sys.routes`). When Service Broker tries to resolve an endpoint for a message originating in the local database, it first looks in the routing table in the local database and searches for a matching service name and a broker instance identifier (if an identifier is included in `BEGIN DIALOG CONVERSATION`). If no entry is found, Service Broker searches for a matching service in the local instance databases and picks the first that is found. When searching for a matching local service, Service Broker does not look through all the different databases on the local instance. It does instead a lookup against an in-memory mapping table. If the message Service Broker tries to resolve the endpoint for was received from outside the service (a forwarding scenario), Service Broker looks in the routing table in `MSDB`.

So, by this we can see that in order to exchange messages with remote servers, we need to create routes. A route is created with the following syntax.

```
CREATE ROUTE route_name
[ AUTHORIZATION owner_name ]
WITH
   [ SERVICE_NAME = 'service_name' , ]
   [ BROKER_INSTANCE = 'broker_instance' , ]
   [ LIFETIME = route_lifetime , ]
   ADDRESS =  'next_hop_address'
```

```
   [ , MIRROR_ADDRESS = 'next_hop_mirror_address' ]
[ ; ]
```

Some of the more interesting options are as follows.

- `SERVICE_NAME`—The name of the service this route points to. The name is case sensitive and collation independent. Notice that `SERVICE_NAME` is optional.
- `BROKER_INSTANCE`—The optional `service_broker_guid` (which we mentioned in the Dialogs section earlier) of the database that hosts the service.
- `LIFETIME`—The number of seconds the route is kept in the routing table. `LIFETIME` is optional, and the route never expires if this option is not defined.
- `ADDRESS`—The network address for the route. It is defined in the format of `TCP://address:port_number`, where `address` can be either DNS name, NetBIOS name or IP address. The port number is the port on which the Service Broker listens. By default, it is 4022. This can be changed by the `CREATE/ALTER ENDPOINT` syntax. At the time of writing this chapter, that syntax hadn't been fully specified, so we use an alternate syntax: `sp_configure 'broker TCP listen port', port_number`.[2] Notice that the service needs to be restarted after changing the port. We mentioned earlier that Service Broker searches through the routing tables for services, and if no service is found, it looks in local databases. In the case of a service residing in a local database, there would be a slight performance gain from explicitly pointing to the service in a routing table. For this purpose, Service Broker accepts an `ADDRESS` of `LOCAL` with no port number. In fact, whenever a broker application is created, Service Broker inserts a default `LOCAL` address in the routing table for that database.
- `MIRROR_ADDRESS`—If `ADDRESS` points to a principal database mirror service, `MIRROR_ADDRESS` should point to the mirror server for automatic failover to work.

To apply this to our example from earlier in this section, we first need to imagine that our application runs on different machines on the same

[2] In order to run this `sp_configure` statement, you may need to enable advanced `sp_configure` options first by calling `sp_configure 'show advanced option', 1` followed by `RECONFIGURE`.

network: `Trader` for the broker/trader database and `DMBroker` for the `DMBrokerage` database. The following code shows how to create the necessary routes (from `Trader` to `DMBroker` and vice versa).

```
--create the route in the trader db on the Trader machine
USE Trader1
CREATE ROUTE TradeEntry
WITH service_name =
             '[//www.develop.com/DMBrokerage/TradeEntryService] ',
ADDRESS = 'TCP://DMBroker:4022'

--now on the DMBroker machine in the DMBrokerage db we need a
--route to the service on the Trader machine
USE DMBrokerage
GO

CREATE ROUTE EnterTrade
WITH service_name = 'enterTrade',
ADDRESS = 'TCP://Trader:4022'
```

Theoretically, we should now be able to exchange messages between services on these two machines (if the necessary message types, contracts, queues, and services are set up). However, at this stage, if we tried, no messages would be delivered. This is because Service Broker by default only allows messages to be exchanged between services on the same database server. In the following section, we cover some of Service Broker's security features and how to enable sending messages between services on different servers.

Security

Service Broker is designed to run enterprise applications in very secure environments. Service Broker obviously uses regular SQL Server security to assign permissions to users in order for the users to create and use the various Service Broker objects. However, Service Broker also has special security requirements because of the loosely coupled and asynchronous nature of its applications, where the involved services may be located in different trust domains and so on. In addition, because Service Broker supports routing of messages, it is not guaranteed that the initiator and the target are directly connected to each other. Because of this, Service Broker can not always use NTLM or Kerberos security but is based on public key certificates.[3]

[3] For an explanation of certificate usage in SQL Server, look under the Certificate topic in SQL Server Books Online.

The special security requirements we mentioned earlier apply to the following parts of the Service Broker architecture.

- Dialogs—Make sure the services that exchange messages are who they say they are.
- Messages—Make sure no one tampers with or reads the message in between the endpoints.
- Transport—Enable exchanging messages between instances, and optionally authenticate the different SQL Server instances on the transport level.

Transport Security

The last bullet item covers transport, and we mentioned earlier that Service Broker by default only allows the exchange of messages between services on the same database instance. The way Service Broker enforces this is by having the transport protocol disabled. It is controlled by this setting in the registry:

```
HKLM\Software\Microsoft\Microsoft SQL
Server\MSSQL.<instance_no>\SSB\TransportEnabled
```

By default this value is 0 (not enabled), and it needs to be set to 1 to enable the transport layer. After the change, SQL Server needs to be restarted.

When the transport is enabled, it is a question whether additional security is needed on the TCP/IP level. Transport security controls what instances can communicate with each other and has no impact on dialog and messages, which we cover later. If the instances run on the same network or between trusted networks, it may not be necessary. But if the communication goes over the Internet, it is definitely necessary. Service Broker decides whether to use transport security or not by using the `CREATE` or `ALTER ENDPOINT` DDL. There are three possible values, as follows:

- 1—Authentication not supported
- 2— Authentication supported
- 3—Authentication required (default)

In our example, because we run on a trusted network, we can change our setting to "not supported." If the application needs authentication, Service Broker supports the following two models:

- SSPI
- Certificate authentication

So the authentication depends on the setting on the endpoint and whether the instances contain certificates in the master database.

SSPI

For SSPI authentication, it is required that both instances be part of the same domain. It is also required that the master database of each instance have a login account for the remote SQL Server's startup service account.

Certificate Authentication

For certificate-based authentication, a certificate holding the credentials for each local instance is required. The certificate is created for the owner of the master database of each instance. Listing 15-20 shows the code that the `dbo` for the `Trader` instance runs to create the certificate. The `dbo` for the `DMBroker` instance runs similar code to create the `DMBroker` certificate.

LISTING 15-20: Create the Certificate

```
--create the local cert for the Trader instance
USE master
GO

CREATE CERTIFICATE TraderCert
AUTHORIZATION sa
FROM file = 'c:\Certs\Trader.cer'
WITH PRIVATE_KEY (FILE = 'c:\Certs\Trader.pvk',
DECRYPTION_PASSWORD = 'Tra5de@r')
GO
```

By setting the `AUTHORIZATION` to `sa`, we make sure that it is owned by `master.dbo`. The `PRIVATE_KEY FILE` argument points to the private key file you create together with the certificate (or that the certificate is created against). In a production environment, the certificate would be issued by a trusted issuer, such as VeriSign. For testing, the certificate can be created through the `makecert` tool in the .NET Framework, as the following code shows:

```
C:\>makecert -n "cn=Yukon1" -r -m 12 -sv c:\Certs\Yukon1.pvk
c:\Certs\Yukon1.cer
```

The argument `c:\Certs\Trader.pvk` sets the path and the name of the private key file created, and the last argument, `c:\certs\Trader.cer`, is the path and file name of the actual certificate.

When the instances in the application try to connect, they somehow need to authenticate each other based on credentials. The local credentials are the certificate we just created. The remote credentials are being authenticated against a certificate based on the public key from the remote instance, which means that the instances need to exchange public keys.

In our example, as Listing 15-21 shows, the `Trader` instance creates a specific login and creates a certificate with this login as owner, based on the `DMBroker` public key.

LISTING 15-21: Certificate for a Remote Public Key

```
--create the remote login which will hold the remote pk cert
USE master
CREATE LOGIN DMBrokersa
WITH PASSWORD = '5#C1yk4*'

--create the cert for the remote instance
CREATE CERTIFICATE RemoteDMBrokerCert
AUTHORIZATION DMBrokersa
FROM file = 'c:\Certs\DMBroker.cer'
GO
```

The `FROM FILE` argument is the path and name of the public key certificate that the `dbo` of the `Trader` instance has received somehow from the `dbo` of the `DMBroker` instance. The `dbo` of the `DMBroker` instance does the same with the public key certificate it has received from the `Trader` instance.

Now each master database in the two instances holds its own certificate plus the public key certificate from the other instance. When the two instances try to authenticate each other, the certificates will be used just because the certificates exist in the master database. Note that the authentication mode setting needs to be set to at least support authentication (value 2).

Dialogs and Messages

For dialogs and messages, Service Broker has the following types of security:

- Full security
- Anonymous security
- No security

The difference between these three types is the level of trust between the initiator and the target of the dialog and what user the connection on each side runs as.

In full security, both sides trust each other, and the two sides have designated user accounts in the respective databases. This is accomplished though certificates and `REMOTE SERVICE BINDINGs`, both of which we cover later.

For anonymous security, it is required that the initiating service trust the target but not vice versa. For this to work, the initiating service needs a `REMOTE SERVICE BINDING` in the database. The target database needs to enable the `guest` user and give the `guest` user `SEND` rights on the service.

With no security, no certificates and no `REMOTE SERVICE BINDINGs` exist, and the dialog is created with `ENCRYPTION = OFF`. In this case both databases need to enable the `guest` user and grant it `SEND` rights on the service.

By default, messages are encrypted and dialog security is used when you are using SSB between instances. For Service Broker dialogs and messages inside the same server instance, messages are never encrypted and dialog security is not used.

The type of security used is based on a combination of the following three things:

- How the dialog is created through the `BEGIN DIALOG CONVERSATION` syntax
- The existence of `REMOTE SERVICE BINDINGs`
- The existence of certificates in the initiating and target databases

Remember from the discussion about dialogs earlier in the chapter how the syntax of `BEGIN DIALOG CONVERSATION` allowed a last argument, which was `ENCRYPTION = ON |OFF`. This argument indicates to Service Broker whether to encrypt the messages or not. We say "indicates," because Service Broker may encrypt the message even if `ENCRYPTION` is set to `OFF`. This is dependent on the existence of `REMOTE SERVICE BINDINGs` and certificates.

Certificates

To accomplish full security, Service Broker uses certificate authentication. The certificate authentication for dialog security is mostly set up the same way as for transport security mentioned previously, and the syntax for creating a certificate is the same, except that the certificate is created against a

user instead of a login. To accomplish full security, take the following steps for certificates.

1. Create a full certificate against a login in the respective database.
2. Create a user.
3. Create the public key certificate from the remote database against the user.
4. Grant the user CONNECT rights in the local database and SEND rights on the service.

The need for the user is because in full security, operations in the remote database run as that particular user. For this purpose, REMOTE SERVICE BINDINGs are used.

Remote Service Bindings

We mentioned earlier that operations in the remote database run in the context of the user for the remote login. The credentials that are exchanged are based on the certificates, as in transport security. Therefore, when Service Broker starts a dialog, it must determine which particular certificate to use for the public key for that service. To tell Service Broker which certificate to use, the developer creates a REMOTE SERVICE BINDING. The REMOTE SERVICE BINDING needs only to be created in the initiating database.

The syntax to create a REMOTE SERVICE BINDING follows.

```
CREATE REMOTE SERVICE BINDING binding_name
   [ AUTHORIZATION owner_name ]
   TO SERVICE 'service_name'
   [ ON CONTRACT  contract_name ]
   WITH  USER = user_name
   [ , ANONYMOUS = { ON | OFF } ]
[ ; ]
```

Notice that a contract can be defined in the case where different contracts on the same service should be used with different certificates. The last argument, ANONYMOUS, decides whether the credentials of the local user are transferred to the remote service or not. If it is ON, the credentials are not transferred, and the initiating service connects as guest. In this case, guest needs SEND rights on the remote service and CONNECT rights to the database. The default for ANONYMOUS is OFF.

Where Are We?

SQL Server Service Broker is a platform that deeply integrates messaging into the database architecture. By doing this, Service Broker delivers on both performance and reliability. At the same time, Service Broker solves some of the more common problems in message-based applications.

The Service Broker platform natively provides much of the plumbing required to build a distributed application, significantly reducing the application development time. Developers can now use the queuing and messaging functionality that Service Broker provides to develop powerful database applications.

Service Broker is not only for external applications, but those that are used extensively inside the SQL Server engine as well.

Service Broker is an example of how Microsoft moves SQL Server from being a pure database server to an application platform. In the next chapter, we will see another example of this, when we look at SQL Server Notification Services.