Chapter 15
SQL Server Service Broker

SQL Server Service Broker 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 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 are knowledgeable in 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 latter return to you a message concerning the status or completion of your request. Some of the compelling reasons for message based applications are:

- **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.
- **Distributing 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. There are a number of details of system implementation for a messaging application that will be unrelated to its problem domain and are 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 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 insure 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.
- **Multi Threading -** 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 state, 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 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 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 xxx1 below shows a simple, message based system that illustrates how Service Broker invokes instances of a stored procedure so stored procedure can process messages as they arrive in a queue.

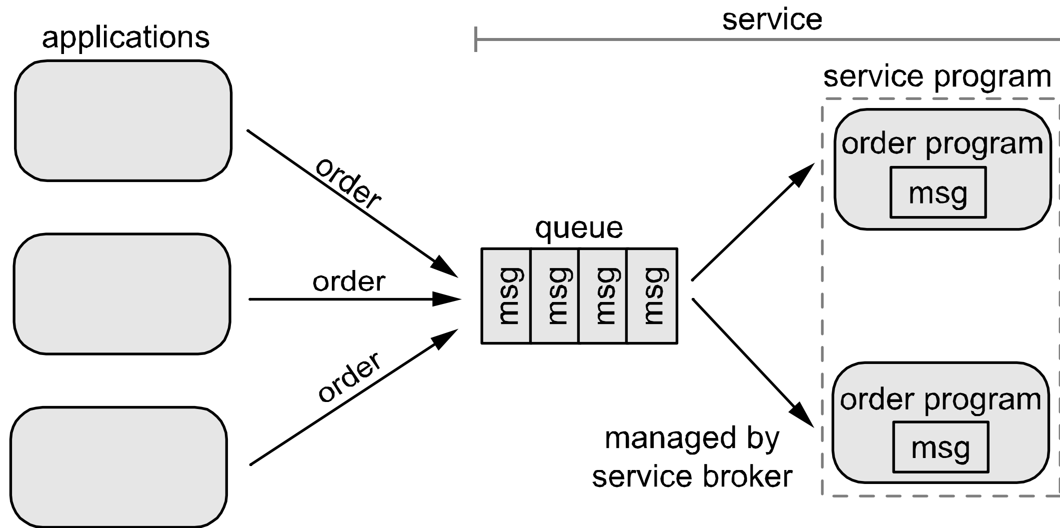

Figure xxx1
Simple Message Based System

The system in Figure xxx1 receives orders from applications and processes them. It only requires that the service program be implemented 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 xxx1 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 service program to process them. Most importantly queues may reconfigured while an application is running.

In almost all applications there are some tasks which must be done immediately and others which 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 xxx2 shows a Service Broker application that has been configured for peak load.

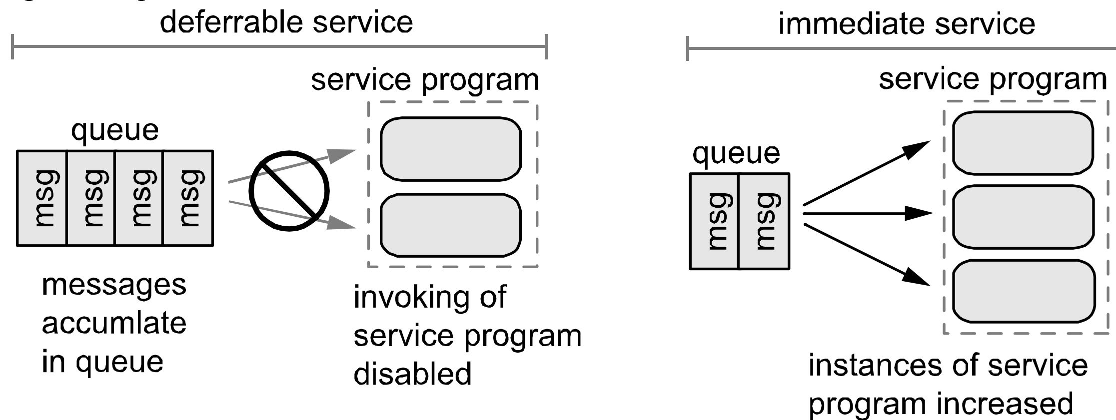

Figure xxx2
Service Broker Application Configured For Peak Load

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 application. This then is the second main feature of Service Broker; it provides a complete framework based on SQL Server for implementing a 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. In the case of term "message" it might refers to the definition of the format of a message and at other times refers 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 manages not only 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 xxx3 below shows how Service Broker efficiently sends a message from one service to another.

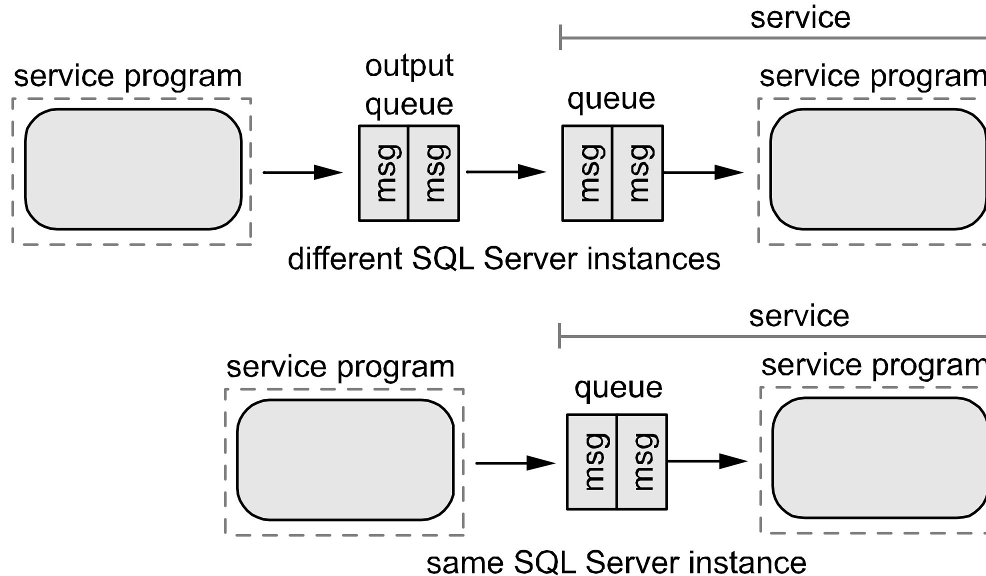

Figure xxx3
Sending Messages Between Services

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 components; messages, queues and services. Concise definitions of the components used in a Service Broker application listed below.

- **Service Program -** A service program is a program that is used to process messages. A service program may be 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 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 and 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 and 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”, “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 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 wishes 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 which are named in the BEGIN DIALOG command, using the message types defined in the contract associated with the conversation.

Some other Service Broker components used by conversations are:

- **Service Instance -** A service instance is created by Service Broker, and assigned a unique identifier. A service instance represents a group 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 conversations.
- **Remote Service Bindings** – The remote service bindings associate a remote service with the user in the local database. It is 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 xxxxx3.

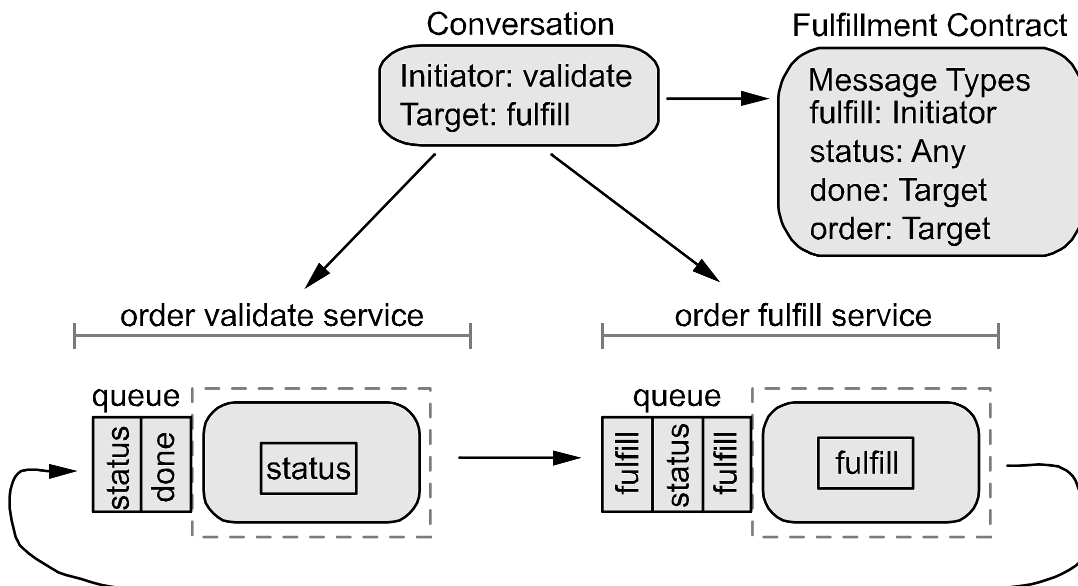

Figure xxxxx3
Typical Service Broker Application

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 sends it to the fulfill service to have it shipped.

The Fulfillment Contract specifies three types of messages; fulfill, status, 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 done message 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 fulfillment service. In this typical 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 insures 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 looks at the many facets of conversations now. We will not be looking at the details of the API’s 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. 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 messages 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 “INITATOR” 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 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 of 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 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 "Service Instance". Every conversation belongs to a single service instance and every service instance has a service instance id. Every message contains both a conversation handle and the service instance id of the service instance to which the conversation belongs. An application decides if a conversation should be part of a new service instance or added to an existing one when it creates the conversation using BEGIN DIALOG.

In order to make the new conversation part of a particular service instance the application needs either a conversation handle or a service instance id to pass into BEGIN DIALOG. If a conversation handle is passed in the new conversation will be added to the service instance of that conversation.

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

If neither a conversation handle or service instance id is passed into BEGIN DIALOG then a new service instance is created for that conversation.

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

The service instance can be locked in three ways. Whenever a message is read from or sent to a queue, the service instance 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 third way to lock a service instance is to use the GET SERVICE INSTANCE command. The GET SERVICE INSTANCE command is issued for a particular queue. The GET SERVICE INSTANCE command locks the service instance associated first message in the queue that from a service instance that is not locked. Note the logic here, the GET SERVICE INSTANCE command will skip over messages from service instances that are locked until it finds one from a service instance that is not locked.

The lock associated with a service instance has a lifetime. It remains locked until the transaction under which it was locked completes. A service instance lock does not have a count; that is calling either RECEIVE or GET SERVICE INSTANCE multiple times under the same transaction will not affect the lifetime of the lock. In typical usage a transaction will have been started before RECEIVE or GET SERVICE INSTANCE is called.

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

The RECEIVE command can be selective about which messages is will read from a queue. For example it can choose to receive all messages in a queue, only the messages associated with a particular service instance or use many other criteria to select messages.

In typical usage a transaction is started, GET SERVICE INSTANCE is used and then RECEIVE is used, followed by either a COMMMIT or ROLLABACK TRAN. If ROLLBACK TRAN is used then all messages that were read from the queue are placed back into it.

Though it is not required, RECEIVE is typically used after a GET SERVICE INSTANCE command. Using GET SERVICE INSTANCE first will find a message from an unlocked service instance, lock the service instance, and then return the service instance id. The RECEIVE can than be used to get messages only for the service instance that GET SERVICE INSTANCE 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 service instance. 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,*i.e.* a message, 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 service instances. 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.

Both RECEIVE and GET SERVICE INSTANCE are each specially designed so that, in effect, you can wrap message processing for a single service instance in a transaction without affecting the processing of messages in other service instances.

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

To see why being able to lock a service instance is important let’s look at an example of what might happen if we couldn’t lock a service instance. 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 xxx1 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.

Not let's look at what happens with service instance locks. The messages for the header and all the line items are in conversations that are in the same service instance. 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 SERVICE INSTANCE which, as it happens, locks the service instance for a work order. It then uses RECEIVE to get all of the messages in the queue associated with the service instance it has just locked. This could include the header and a number of line items. It processes these in order but making INSERTs into the appropriate tables then returns.

It may be that second instance of the service process, running at the same time, also does a GET SERVICE INSTANCE. It may also lock a service instance 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 service instance finishes it releases the lock on the service instance. Then yet another instance of the service process does a GET SERVICE INSTANCE. It may end up locking the service instance 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 service instance 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 queue can continue receiving more messages for a service instance even while that service instance is locked. Neither of these things would be possible if service process used the BEGIN TRAN, SELECT, DELETE sequence to remove messages from a queue. Service instances and their locks are crucial to the efficient operation of Service Broker.

Another important use of a service instance is to have a way to maintain state across the conversations in a service instance. 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. There are three ways it can do this. 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 scaleable 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 is also is not scalable. The problem is that the storage required for purchase order number goes up as the number of messages in increase. This in effect a 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 service instance 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, there are two things that you will need. Both are easy to get. First of all

you need something to key the state you will be storing in SQL Server. The service instance id is unique and is a GUID 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 SERVICE INSTANCE command. Using RECEIVE locks the service instance but it also reads the queue. Sometimes you need to access or manipulate the state you are sharing within a service instance before you read the queue. GET SERVICE INSTANCE gets the service instance associated with the next message in the queue and returns the service instance id but it does not read the queue. In either case you can then use the service instance 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 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 use Service Broker is to break your application up 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 not hard and fast rule, but this should be a critical task that creates a service instance and allocate 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 stock exchange. A simple example designed to illustrate the use of Service Broker DDL and DML extensions in Yukon follows.

Message Type

It is of vital importance that sender and receiver in a messaging application understand what messages will be sent. In Service Broker the description of the messages are defined in a `message type` object. The `message type` object defines the name of the message as well as 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
   ENCODING { EMPTY | VARBINARY | XML
       [ WITH schema_target_namespace ] }
   [ AUTHORIZATION owner_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]`.
- **`ENCODING`** – How the message is encoded. This is required, and must be `XML`, `VARBINARY` or `EMPTY` as per below.
- **`XML`** – The message is encoded as XML and can be any well formed XML document or fragment. If a message is defined as XML encoded, it will be parsed when it is received to verify that it contains well-formed XML.
- **`VARBINARY`** – The message is encoded as a byte stream.
- **`EMPTY`** – The message body is empty.
- **`WITH schema_target_namespace`** - This specifies the XML schema to validate the message against. The schema must be registered in SQL Server before it can be used in the creation of the `message type`. Chapter xx covers how to register a schema in SQL server. If WITH **`schema_target_namespace`** is not used then the message will not be validated.
- **`AUTHORIZATION owner_name`** – Defines which user or role owns the `message type`.

Based on the tasks outlined in Figure 15-1 we can see that there are two types of messages that are involved when the client enters a stock trade order:

- The message that is sent from the broker/trader to the brokerage containing the original order
- The acknowledgement 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]
ENCODING XML

-- then the acknowledgement message
create message type
[//www.develop.com/DMBrokerage/TradeAck]
ENCODING 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 is 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 as well as 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 for the tradeEntry message
CREATE XMLSCHEMA 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 messagetype based on the schema
create message type
```

```
  [//www.develop.com/DMBrokerage/TradeEntry]
ENCODING XML
WITH 'http://www.develop.com/DMBrokerage/schemas/tradeEntry'

--create the schema for the tradeAck message
CREATE XMLSCHEMA N'<?xml version="1.0" ?>
   <xsd:schema xmlns:xsd="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 messagetype for the trade ack
-- based on the schema
create message type
[//www.develop.com/DMBrokerage/TradeAck]
ENCODING XML
WITH 'http://www.develop.com/DMBrokerage/schemas/tradeAck'
```

These `message types` need to be created in both the broker database as well as 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 pre-defined message types in Service Broker. The three that are most common are:

- **`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 error 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.

Changing **`Message Types`**

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

Listing 15-4

Syntax to `ALTER` a `Message Type`

```
ALTER MESSAGE TYPE message_type_name
[ ENCODING { EMPTY | VARBINARY | XML
  [ WITH schema_target_namespace ] } ]
[ AUTHORIZATION owner_name ]
```

The ALTER syntax allows us to change the ENCODING and/or the AUTHORIZATION. To DROP a message type you use the 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 can not 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.

Having 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 have seen above, 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 I 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 by both. Listing 15-5 shows the syntax to create a contract.

Listing 15-5

Create Contract Syntax

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

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** – This argument 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 send by either the INITIATOR or the TARGET.
- **AUTHORIZATION owner_name** – Defines which user or role owns the contract. If this isn't specified it 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 as well as 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 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 can be altered and dropped. The syntax to alter a contract is: "`ALTER CONTRACT contract_name AUTHORIZATION owner_name`". The only change that can be done through `ALTER CONTRACT` is the owner of the contract.

The syntax to drop a contract is: "`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. I 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 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 } ]
   [ ACTIVATION (
      [ STATUS = { ON | OFF } , ]
      PROCEDURE_NAME = stored_procedure_name ,
      MAX_QUEUE_READERS = max_readers ,
      EXECUTE AS { SELF | USER = 'user_name' }) ] ]
```

There are quite a few options when creating a queue and here follows a short explanation of the various arguments.

- **`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 of the database it is created in, the URL-like syntax used for other service broker names isn't necessary for queue names.

- **`STATUS`** – This clause 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.
- **`ACTIVATION`** –This clause 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, default is `ON`. I cover different aspects of activation of the service programs later in this chapter.
- **`PROCEDURE_NAME`** – This 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 ot MAX_QUEUE_READERS.
- **`EXECUTE_AS`** – This optional clause specifies what SQL Server login the activated procedure to run under. If SELF the procedure runs under the user who created the queue.

In the StockTrade 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 acknowledgement messages from the brokerage database. Figure 15-4 shows this. In a real world application, we would probably have more queues to handle different messages. Feel free to create a many queues as you deem necessary for your application.

[insert Figure 15-4 here]

Figure 15-1
Queues Used in the Stock Application

The code to create the queues for the StockTrade 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 is just to create a procedure that is just an empty shell as in the code snippet below and catalogue it in SQL Server.

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

RETURN 0
GO

--create in the DMBrookerage 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 acknowledgements from
--the brokerage
use Trader1
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
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 down 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.

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.
service_instance_id	uniqueidentifier	Service instance identifier for the message.
conversation_handle	uniqueidentifier	Conversation identifier for the message.
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.
encoding_format	nchar	Encoding of the message. (E=Empty, B=Binary, X=XML)
message_body	varbinary(MAX)	Content of the message.
next_fragment	int	Number of the next fragment to be received for a fragmented message.
fragment_size	int	Size in bytes of each fragment in a fragmented message.
fragment_bitmap	bigint	Bitmap for fragments that have been received. (0=Not received, 1=received)

In the process of developing a Service Broker application we now have the basic plumbing. We have:

- Message Types
- Contracts
- Queues

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

Services

A service is, as defined in Table 15-1, an endpoint for specific functionality in the Service Broker application. Based on the service name, the Service Broker routes messages between databases and puts the messages on the queue for that particular service and message type. The specific functionality 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
- 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 looks like in Listing 15-9.

Listing 15-9

Syntax for Create Service

```
CREATE SERVICE service_name
ON QUEUE queue_name
   [ ( contract_name [ ,...n ] ) ]
   [ AUTHORIZATION owner_name ]
[ RETENTION = { ON | OFF } ]
```

The arguments for CREATE SYNTAX 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.
- **`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.

The syntax specifies both a queue name as well as 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. 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), as the messages are kept in the queue until the conversation ends.

In the trade application we now have two message types, one contract and one queue in each participating database. In Listing 15-10 I show 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
create SERVICE enterTrade
ON QUEUE tradeAckQueue
([//www.develop.com/DMBrokerage/EnterTrade])

--create a service in the brokerage db
use DMBrokerage
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.

[insert Figure 15-5 here]

Figure 15-2
Interaction Between Services, Messages and Queues

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

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 ***dialog***. A dialog is communication between

exactly two end-points. It is a logical connection between Service Programs, which run on Service Brokers. The communication is bi-directional as Figure 15-6 shows.

[insert Figure 15-6 here]

Figure 15-3
Illustration of Dialogs

The dialog ensures that any messages associated with a dialog are delivered exactly once and in the order 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, the 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 corresponds to which sent messages. The dialog handles correlation automatically for you. The correlation of messages are handled by a unique identifier of the conversation; `conversation_handle`.

Another important part of dialogs is the message acknowledgement. Dialogs incorporates automatic message acknowledgement for all messages with a sequence number. When a message is sent, the sending broker keeps the message on the transmission queue until an acknowledgement 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 looks like 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_SERVICE_INSTANCE = service_instance_id } ]
   [ [ , ] LIFETIME = dialog_lifetime ]
   [ [ , ] ENCRYPTION = { ON | OFF }  ] ]
```

The arguments for `BEGIN DIALOG` is 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`** - The contract that defines messages used in this

conversation.

- **RELATED_CONVERSATION = conversation_handle** –I can associate this dialog with the service instance of an existing dialog through the `conversation_handle` variable.
- **RELATED_SERVICE_INSTANCE = service_instance_id** – When starting a new conversation Service Broker also creates a new service instance (I cover service instances later). This argument allows me to associate the dialog with an existing service instance instead of creating a new service instance.
- **LIFETIME = dialog_lifetime** - Specifies the lifetime in seconds for the dialog. If not specified, the dialog remains open until it is explicitly closed.
- **ENCRYPTION** - Specifies whether messages sent and received on this dialog are encrypted. The default for `ENCRYPTION` is `ON`.

It is worth noting a couple of things regarding the `TO SERVICE` argument above. 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_SERVICE_INSTANCE` clauses do the same thing. They link the dialog created to an existing service instance. You would use the `RELATED_SERVICE_INSTANCE` if you created your own GUID for the Service Instance Id. By using `RELATED_CONVERSATION` you link the dialog to the Service Instance Id of the specified `conversation_handle`.

We can also define what instance to target. This is important 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 `family_id` GUID for the database. You retrieve the GUID through following syntax:

```
select family_id
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 us now go back to the trade application and look at the syntax as in Listing 15-12 to create a conversation between our services.

Listing 15-12

Start a Conversation in the Trade Application

```
--declare a variable for the conversation handle
DECLARE @dh uniqueidentifier;
--this starts the dialog from the
--trader/broker database
--and we get a dialog/conversation handle automatically
BEGIN DIALOG @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 trader/broker 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 trader/broker database. `far_broker_instance` is a column which 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 do not explicitly set a lifetime for the dialog, the dialog is alive until it is being closed explicitly. The syntax to close a dialog is as follows:

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

The conversation_handle variable is the variable obtained through the `BEGIN DIALOG` 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` (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 as well as 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 is:

```
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) keeps 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 ***Service Instance***.

Service Instance

You use the Service Instance 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, etc. 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 Service Instance identifier automatically. The identifier is a GUID (SQL server data type `uniqueidentifier`). The identifier is appended to the messages we send. You can create the identifier yourself (in TSQL you use `NEWID()`). Subsequently when you do `BEGIN DIALOG` you relate the dialog to the identifier using the syntax in Listing 15-11. Relating the dialog to a Service Instance identifier is the solution if you want to use an existing identifier for your dialog or associate your dialog with the service instance of an existing dialog.

To obtain the identifier within a conversation you do a `SELECT` against the `service_instance_id` column in the message queue. The identifier can furthermore be retrieved by the `GET SERVICE INSTANCE` call. Calling `GET SERVICE INSTANCE` gets the Service Instance identifier for the next message to be retrieved. As we will see later, the Service Instance 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 SERVICE INSTANCE`.

The biggest benefit of the Service Instance is that of locking of the dialogs. You may ask why it is important to lock dialogs. I 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 below). In other words, we have multi-threaded queue readers.

The problem with this is that there is parallel processing of messages. If the messages are data of a parent child nature (think orders with order lines) a situation can occur that 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-reader1 (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 Service Instance and no other queue readers can receive messages on dialogs in that particular Service Instance until the transaction is committed.

Service Instance is also beneficial when you want to keep application state. You can store any state data in the database based on the Service Instance identifier and retrieve it when needed because every message you receive will have the service instance ID in the result. For those of you that are ASP developers, you can see the Service Instance Id as a cookie. Naturally, the lifetime of a Service Instance is important if you rely on it for state data. A Service Instance 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 proc on the tradeAckQueue is activated and processes the message.

These steps are done through Service Programs. Service programs are 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 initiator of one dialog and the target of another. A service program could theoretically also be initiator and target of the same dialog. That is however very unlikely.

In our stock trade 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 acknowledgements of the trades on the `tradeAckQueue`.

I mentioned above that a stored procedure is ***activated*** when a message arrives on a queue. In the following section we will look a little bit closer at the activation features in Service Broker.

Activation

In a traditional messaging application you have basically two options to find out that a message have 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 to 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 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 I 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 should be activated. The way it works is as follows:

When SQL Server starts the internals of SQL Server knows 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_READER` number has been reached. When the `MAX_QUEUE_READER` 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 there are a couple of things to bear in mind 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 queue up.
- The activation monitor code has no way of knowing when a program has finished executing, apart from noticing that the program is 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 above I said that the service program should be able to handle all message types. You may ask what all message types are:

- 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 you normally want to send a message. 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 semi colon (;).

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` 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. For `XML` and `VARBINARY` encoding the message body should be sent as `varbinary(MAX)`. If the encoding is `EMPTY` the message has no body.

For the trade application we have a stored procedure in the trader/broker database which 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 (as per Listing 15-12) and sends the message.

Listing 15-14

Begin Dialog and Send 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 @msgXml XML
--variable for the message that will be eventually
--converted to varbinary
DECLARE @msgString nvarchar(MAX)
--the message body that will be sent
DECLARE @msgBody varbinary(MAX)

--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 @msgXml = '<id>Order1</id>'

--convert xml to varbinary(MAX)
SET @msgBody =
cast (NCHAR(0xFEFF) + cast (@msgXml as NVARCHAR) AS VARBINARY(MAX))


--begin the dialog
BEGIN DIALOG @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
SEND ON CONVERSATION @dh
  MESSAGE TYPE [//www.develop.com/DMBrokerage/TradeEntry]
  (@msgBody)
```

To test if it works:

- create two databases (it can be done in one but it gets more realistic in two). Name them for example Trader1 and DMBrokerage.
- Create the message types and contracts from Listing 15-2 and Listing 15-6 in both databases.
- Create the tradeAckQueue in Trader1 and the tradeEntryQueue in DMBrokerage as per Listing 15-8. When you create the queues, do not set any `ACTIVATION` arguments just yet.
- The last thing to do before you can test is that you need to set up the services like in Listing 15-10.
- When the above is done, you can run the code in Listing15-14 from the Trader1 database.

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

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

Listing 15-15

RECEIVE Syntax

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

The syntax looks almost exactly like `SELECT` and both `SELECT` and `RECEIVE` return a resultset. The difference as I mentioned above is that `RECEIVE` de-queues the message(s) on 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 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 if 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 there will not be any other messages 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`.

Seeing that the `RECEIVE` command allows you to do `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 doing T-SQL you are probably better off to do a `RECEIVE TOP(1)` especially because as you can not 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 to receive a resultset of multiple messages.

Listing 15- 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
```

Run the code in Listing 15-16 from the DMBrokerage database and it retrieves the message you sent in Listing 15-14. If you do a `SELECT` against the `tradeEntryQueue` after the `RECEIVE,` there are no messages there. Notice that the body of the message is output as varbinary. In a `SELECT` statement you can convert the column to another data type. That is not possible when doing `RECEIVE`. You have two options to view the message body column in a more human readable form from a `RECEIVE`:

- Declare a table variable. `RECEIVE` the columns you are interested in into the table variable. Do a `SELECT` from the variable and in the `SELECT` do the conversion.
- Declare variables for the columns you are interested in. During `RECEIVE` assign the columns you are interested in, to those variables. Convert the necessary variable and do a `PRINT` (or `SELECT`)

Listing 15-17 shows a code snippet that uses the latter method. It also applies `WAITFOR` and `TIMEOUT` arguments. The `TIMEOUT` is set to one minute (60000 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 Workbench will say “Executing Query”. Switch over to the trader/broker 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 de-queued.

Work Flow in a Service Program

At this point all code necessary for the trade application is done. It is time to tie it together and look at the work flow in a service program. I 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.

[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.

Listing 15-18

Code for Service Program in DMBrokerage

```
create procedure tradeEntryProc
as

--declare variables
DECLARE @dh          uniqueidentifier
DECLARE @msg         varbinary(MAX)
DECLARE @ack         XML
DECLARE @ackString   nvarchar(MAX)
DECLARE @si          uniqueidentifier

WHILE (1=1)
BEGIN

-- 1. start a transaction
BEGIN TRAN;
SET @si = null;


-- 2. Lock the service instance if
-- dealing with state data
WAITFOR (
GET SERVICE INSTANCE @si
FROM tradeEntryQueue
),
TIMEOUT 10000

--check that we have a message
if @si is null
  BEGIN
    ROLLBACK TRANSACTION
    BREAK
  END

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

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

-- 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>'
set @ackString = NCHAR(0xFEFF) + convert(nvarchar(MAX), @ack);

-- 6. send the message
SEND ON CONVERSATION @dh
  MESSAGE TYPE [//www.develop.com/DMBrokerage/TradeAck]
  (CONVERT(varbinary(MAX), @ackString))

-- 7. end the conversation
```

```
END CONVERSATION @dh

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

-- 9. Commit or rollback
COMMIT TRAN
END
```

The reason I compare this flow with Windows message loops is because when the process is done it starts all over again. Let us look at the various parts of the model.

Begin Transaction – The transactional model is one if the biggest benefits with Service Broker compared to 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 de-queued messages are put back on the queue. The outgoing messages are not sent, and we can start all over again.

Lock Service Instance – Applicable if we deal with application state. The Service Instance 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 pre RECEIVE. We need the conversation handle in order to send messages back to the initiator.

Process Data - In a real world application we probably receive messages for different message types on the same queue. We need therefore to check what message type we receive and process accordingly. For this sample, I suggest 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 to be after 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 for send 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 catalogued 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 trader/brokerage database that functions as the Service Program for message on the tradeAckQueue. You can see an example of this in Listing 15-19. Notice that you need to add some code to handle the received messages.

Listing 15-19

Code for Service Program in the Trader/Broker Database

```
create procedure tradeAckProc
as

--declare variables
DECLARE @dh           uniqueidentifier
DECLARE @msg          varbinary(MAX)
DECLARE @si           uniqueidentifier
DECLARE @mt           nvarchar(MAX)

WHILE (1=1)
BEGIN

BEGIN TRAN;
SET @si = null;

WAITFOR (
GET SERVICE INSTANCE @si
FROM tradeAckQueue
),
TIMEOUT 10000

if @si is null
BEGIN
  ROLLBACK TRANSACTION
  BREAK
END


  ;RECEIVE TOP(1) @dh=conversation_handle,
         @msg=message_body,
         @mt = message_type_name
  FROM tradeAckQueue
  WHERE service_instance_id = @si


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

END CONVERSATION @dh

COMMIT TRAN
END
```

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