Architecting Service Broker Applications

41 out of 49 rated this helpful

Roger Wolter

Microsoft Corporation

October 2006

Applies to:  
   Microsoft SQL Server 2005 Service Broker

**Summary:** Over the past few years, many papers, books, Web casts, and presentations have discussed how Microsoft SQL Server 2005 Service Broker (SSB) works and how to build applications with it. This article moves up a level from the "how" to discuss "why" Service Broker should be used, and what decisions have to be made to design and build a Service Broker application successfully. (18 printed pages)

For more on architecting Service Broker applications and Roger Wolter, listen to the following [ARCast](http://www.arcast.net/) episodes:

* [ARCast: SQL Server Application Platform (Part 1 of 2)](http://www.skyscrapr.net/blogs/arcasts/default.aspx?ID=294)
* [ARCast: SQL Server Application Platform (Part 2 of 2)](http://www.skyscrapr.net/blogs/arcasts/default.aspx?ID=297)

Contents

[Service Broker Architecture](http://msdn.microsoft.com/en-us/library/aa964144.aspx" \l "archsrvbrkapp_topic1)  
[When Should You Build with Service Broker?](http://msdn.microsoft.com/en-us/library/aa964144.aspx" \l "archsrvbrkapp_topic2)  
[Designing the Service Broker Application](http://msdn.microsoft.com/en-us/library/aa964144.aspx" \l "archsrvbrkapp_topic3)  
[Infrastructure Aspects of a Service Broker Application](http://msdn.microsoft.com/en-us/library/aa964144.aspx" \l "archsrvbrkapp_topic4)  
[Conclusions](http://msdn.microsoft.com/en-us/library/aa964144.aspx" \l "archsrvbrkapp_topic5)

Service Broker Architecture

When talking about software architecture, it is often useful to think in terms of what architects who design buildings do. There are a lot of parallels between designing a building and designing an application.

The Art and Science of Design

When you design a building, the land it's going to be built on is usually a given. You might start with a building design and then go looking for a place to build it, but this is unusual. The land imposes constraints on your design. The size of the land determines the maximum dimensions of the foundation. The soil conditions (or zoning laws) might limit the maximum height. The slope might determine what kind of building you build. The surrounding buildings will influence the design and materials used. In the case of an SSB application, the land would be Microsoft Windows and Microsoft SQL Server. While SSB is a database feature that can be accessed in a variety of ways from many different platforms (basically, anything that can connect to SQL Server), you can't build a Service Broker without SQL Server, and you can't run SQL Server on anything but Windows.

The biggest constraints on building design are the customer requirements. Will it be a house or an office building? Does it need two bathrooms or 200? Is the customer a bank or a steel mill? No architect would start construction before understanding these critical requirements. Some decisions can be deferred—the paint color, the furniture—but all the decisions necessary for each phase of construction must be made and signed-off on before construction starts. If a customer decides they really need to build a three-bedroom split-entry house after 50 floors of steel have been completed on the original request for an office building, it's going to cost a lot of money. There's a perception that software is flexible, so that major changes in requirements can be accommodated at any point. While the cost of changing software requirements might not be as high as changing the design of a half-built building, there *are* costs, and changes hurt, so getting the requirements right is vital.

The next step in building design is selecting the materials and tools to use. In some cases, there's quite a bit of freedom in these choices, but if you're building a 30-story office tower, framing it with pine two-by-fours is not an option. The customer's requirements might also limit your options. If the customer loves brick, you might not be able to use adobe—even if it's the best material for the job. The ultimate constraint is often the customer's budget. Some decisions, the customer will make, and others they will leave to the architect's best judgment. They might care passionately about whether the roof is wood or tile, but leave the choice of copper or PVC pipe up to the architect. There are other times when the customer's requirements are either unwise or impossible, so it's the architect's responsibility to change the customer's mind.

For example, no matter how much the customer likes lead pipes, you can't let them have them. The same kinds of choices and trade-offs apply to designing a Service Broker application. You first have to decide whether Service Broker should be used at all. While it *is* the best thing since sliced bread, Service Broker is not the answer to everything. Service Broker has a large number of features and options. You have to use a combination of the customer's requirements, Service Broker capabilities, design constraints, and your own judgment to determine which features you should use and how you should use them.

After the constraints, requirements, and materials available have been determined, the architect can design a building that satisfies the requirements, fits the constraints (including the time and resources available), and satisfies the architect's and customer's aesthetic sense and professional pride. A professional architect won't design an ugly eyesore, even if that's exactly what the customer wants, because it will reflect badly on the reputation and perceived competence of the architect. Similarly, a software architect should never design an application that won't work, just because the customer demands it. If what the customer wants won't work, it's your job to tell them that, and if that means they find someone else to design the application, at least your name won't be associated with a software disaster.

After obtaining a design, the responsibilities of the architect don't end. The architect must closely monitor construction, be ready to make design changes as circumstances change, and be responsible for ensuring that the building satisfies the design criteria. The carpenters can't go moving walls around to suit themselves, even if that does solve a real problem. Similarly, a software architect can’t throw a design over the wall to the developers. The software architect must shepherd the project through implementation, test, and implementation.

When Should You Build with Service Broker?

Service Broker is a platform for building loosely coupled, reliable, distributed database applications. It is built into all editions of SQL Server 2005, so that it can be used in any SQL Server 2005 application. If you would like more information about Service Broker, you might try [this article in the MSDN Library](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sqlsvcbroker.asp). If you want a lot more information, I recommend *The Rational Guide to SQL Server 2005 Service Broker*, available [here](http://www.amazon.com/Rational-Server-Service-Broker-Guides/dp/1932577270/sr=1-1/qid=1158009357/ref=pd_bbs_1/103-0049023-7244631?ie=UTF8&s=books).

While I like to think that all applications are potential Service Broker applications, the reality is that only most of them are. The way an architect decides whether to use a building material is by matching its characteristics to the requirements. The following is some guidance on how to do this for Service Broker.

Queues

One of the fundamental features of Service Broker is a queue as a native database object. Most large database applications I have worked with use one or more tables as queues. An application puts something that it doesn't want to deal with right now into a table, and at some point either the original application or another application reads the queue and handles what's in it. A good example of this is a stock-trading application. The trades have to happen with subsecond response time, or money can be lost and SEC rules violated, but all the work to finish the trade—transferring shares, exchanging money, billing clients, paying commissions, and so on—can happen later. This "back office" work can be processed by putting the required information in a queue. The trading application is then free to handle the next trade, and the settlement will happen when the system has some spare cycles. It's critical that once the trade transaction is committed, the settlement information is not lost, because the trade isn't complete until settlement is complete. That's why the queue has to be in a database. If the settlement information were put into a memory queue or a file, a system crash could result in a lost trade. The queue also must be transactional, because the settlement must be either completed or rolled back and started over. A real settlement system would probably use several queues, so that each part of the settlement activity could proceed at its own pace and in parallel.

So, queues in the database are a good thing. But why not just use tables as queues, instead of inventing a new database object? The answer is that it's hard to use tables as queues. Concurrency, lock escalation, deadlocks, poison messages, and so on are all difficult problems to resolve. The Service Broker team spent years coming up with a reliable, high-performance queue, so that you can just call CREATE QUEUE to take advantage of all that work in your application. The logic to put a message on a queue, pull it off, and delete it when you are done with it has been incorporated into new TSQL commands (SEND and RECEIVE), so that you don't have to write that logic either.

SSB queues can be used for just about any asynchronous activity that a database application wants to do. An order-entry application might want to do shipping and billing asynchronously to improve order response times. A trigger that must do a significant amount of processing might use SSB to do the processing asynchronously, so that updates to the original table are not affected. A stored procedure might need to call several other stored procedures in parallel. The list goes on.

The asynchronous-queue pattern is applicable to a tremendous number of applications. Just about any large, scalable application uses queues somewhere. Windows does almost all of its IO through queues. IIS receives HTTP messages on queues. SQL Server TSQL commands are all executed from a queue.

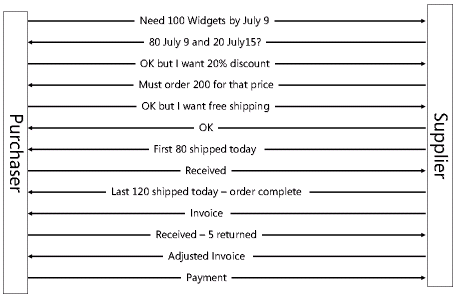
The obvious question here is: If Service Broker queues are so great, why don't these applications use them? The short answer is that Service Broker queues are persistent. Putting persistent messages on an SSB queue—or removing and processing them—involves a write to the SQL Server transaction log. This is a very good thing if you're doing trade settlement or billing, and you want to make sure that the trade or order is not lost if the power goes off. But if the power goes off on Windows or IIS, the incoming connections are dropped, and the work in progress disappears. At this point, the messages in the queue are worthless, because the applications waiting for a response are gone, so that persisting the messages in the queue is a waste of resources and an unnecessary slowdown. It's cool to think about a reliable query mode in which your queries are persisted so that the answer is returned, even if the client or the server crashes in the meantime (and several customers use Service Broker to do that). But the thousands of existing applications aren't built to take advantage of persistent messages.

Therefore, the sweet spot for using SSB queues is database applications that must do things reliably and asynchronously. If the action must happen synchronously, a normal function call or COM or RPC is the right technology. If the action must be started asynchronously, but it's okay for it to disappear if the application dies, some kind of in-memory queue will perform better. Also, Service Broker is a SQL Server feature, so it probably doesn't make sense to use it unless there's a SQL Server database around. There are applications in which reliable, persistent queues are so important that adding a SQL Server database just to do the queuing is justified. In general, however, SSB is a better fit for database applications. It's also worth noting that because SSB is accessed through TSQL commands, any platform, language, and application that can connect to a SQL Server database through SQL Client, OLEDB, ODBC, and so forth can send messages to or receive messages from a Service Broker queue. This makes it easy to integrate applications on many platforms reliably, transactionally, and asynchronously.

Dialogs

One of the more unique features of Service Broker is the *dialog*. A dialog is a reliable, ordered, persistent, bidirectional stream of messages. In most messaging/queuing systems, the messaging primitive is the message; each message is independent and unrelated to other messages at a messaging level. If the application wants to establish relationships between messages—linking a request to a response, for example—the application is responsible for doing the tracking.

In SSB, the dialog is the messaging primitive. Messages that are sent on a dialog are processed in the order in which they were sent—even if they were sent in different transactions from different applications. Dialogs are bidirectional, so request-response relationships are automatically tracked. Dialogs are persistent, so that the dialog remains active even when both ends of the dialog go away, the database is shut down, the database is moved to another server, or what have you. This means that you can use dialogs to implement long-running conversational business transactions that last for months or years. For example, processing a purchase order typically involves a long-running exchange of messages between the purchaser and supplier, as prices are negotiated, delivery dates agreed upon, status communicated, delivery confirmed, and payment exchanged. This whole exchange can be a single Service Broker dialog that can last for months. Figure 1 illustrates a typical long-running dialog conversation.



**Figure 1. Purchase-order dialog conversation**

Conversation Groups

Dialogs exist in *conversation groups*. A conversation group is the unit of locking for a Service Broker queue. Every time a RECEIVE or SEND is executed, the conversation group that contains the dialog used for the RECEIVE or SEND is locked. One of the more difficult problems with asynchronous, queued applications is that if related messages are received by different application threads, the applications state can get corrupted because of simultaneous changes or changes processed out of order. For example, an order line might be processed before its order header, causing the order line to be rejected. In many cases, this can be resolved only by making the application single-threaded, which obviously limits scalability and performance. With Service Broker, the application puts all of the dialogs related to a given business transaction in a single conversation group, so that only one thread will be processing that business transaction at one time. For example, an order-entry application would put all of the dialogs associated with a given order into the same conversation group, so that when hundreds of threads are processing hundreds of order messages simultaneously, the messages for any given order are only processed on one thread at a time. This allows you to write a single-threaded application and let Service Broker manage running it on hundreds of threads simultaneously.

A multireader queue is probably the most efficient load-balancing system available. The queue readers, whether they are on the database server or on remote servers, open a connection to the database and start receiving and processing messages. After each message, is processed the queue-reader application receives another one. In this way, each queue reader receives as much work as it is able to process. If one of the readers slows down for some reason, it just does TSQL RECEIVE commands less often, and other readers are free to pick up the slack. If one of the readers shuts down or crashes, the receive transaction for the message it was processing rolls back, and the message appears on the queue again for another reader to handle. If the queue starts growing because the readers can't keep up, you can start up another one, and it will start processing messages. There's no reconfiguration necessary; just start and stop readers as required. Conversation-group locking makes all of this possible. The sending application doesn't know or care how many queue readers there are or where they are running.

Activation

One of the fundamental issues with asynchronous, queued applications is that the RECEIVE command pulls messages off the queue for processing. This means that the receiving application has to be running when a message arrives on the queue. There are several approaches to this, such as receiving from a Windows service that always runs, or from a startup stored procedure that starts when the database starts. These are good solutions when messages arrive at a constant rate, but in many cases the receiving application is wasting resources when there are no messages on the queue, and getting behind when the message-arrival rate peaks.

Service Broker offers a better alternative called *activation*. To use activation, you associate a queue with a stored procedure that knows how to handle messages in that queue. When a message arrives on the queue, the SSB logic that handles it commits checks to see if there is a copy of the stored procedure running. If there is a copy running, the commit continues; if there isn't, the activation logic starts one. This is better than the triggers that some messaging systems offer, because a new copy is started only when it is needed. Activation assumes that the stored procedure will keep reading messages until the queue is empty, while triggers will start a new reader for every message. If 1,000 messages arrive on a queue per second, activation will start one reader, while triggers would start 1,000. Activation also looks at whether the queue is growing, because messages are arriving faster than the stored procedure is processing them. If the queue is growing, activation will start new copies of the stored procedure until the queue stops growing. When the queue is empty, the stored procedures should exit, because there is no work to do. In this way, activation assures that there are enough resources dedicated to processing messages on the queue, but no more resources than are needed.

Activation has another useful side benefit. You can execute a stored procedure by sending a message to a queue. This stored procedure runs in the background on an execution, transaction, and security context that are different from the stored procedure that sent the message. This is what enables asynchronous triggers and stored procedures that start up multiple other stored procedures in parallel. Because the activated procedure runs in a different security context, it can have more or fewer privileges than the caller. Because the activated procedure runs in a different transaction, deadlocks or failures don't affect the original transaction.

For example, I worked with a customer who inserted an audit record into a log table at the end of every transaction. In too many cases, this insert would cause a deadlock or timeout, and the whole transaction would be rolled back—leading to user frustration. They changed their auditing logic to SEND a message to an SSB queue, and now a problem with the audit table doesn't cause the original transaction to fail. Another customer wrote a simple stored procedure that receives a message from a queue, calls EXEC on the contents of the message, and sends the results back to the originator on the same dialog. They can now run TSQL commands in the background on any system in their data center, by sending a message to it. SSB security makes this more secure than allowing an administrator to log on to the server, and SSB reliable delivery means the commands and responses are never lost.

Reliable Messaging

We've already seen the value of Service Broker in designing asynchronous, queued applications. Dialogs, conversation-group locking, and activation make Service Broker a unique platform for building loosely coupled database services. Once you understand all of the powerful applications that you can write with SSB queues, it won't take long for you to come up with application ideas that require putting messages on a queue in another database. If the other database runs on a different server, SSB reliability assurances require that the message is sent reliably. This means that the remote database acknowledges receipt of the message, and the local Service Broker keeps sending it until an acknowledgement is received. In this way, an application can SEND a message to a remote queue and have the same reliability assurances as if it were sent to a local queue. In fact, the application doesn't know or care whether the message it is sending will be processed locally or remotely. Writing distributed, queued, asynchronous Service Broker applications is no different from writing local applications. This means that you can start with a local application and make it distributed, as processing load or business requirement changes.

Unfortunately, including reliable messaging in Service Broker has led to a lot of confusion. As soon as people see reliable messaging, they think MSMQ or MQ Series. While SSB has a lot of the same capabilities, it is primarily a platform for building distributed database applications. For example, it's trivially easy for a stored procedure to start a stored procedure reliably and asynchronously in a remote database with Service Broker, but doing the same thing by using MSMQ would be very difficult. (I have more thoughts on these issues in [my blog](http://blogs.msdn.com/rogerwolterblog/archive/2006/02/28/540803.aspx) and in the [Architecture Journal Issue 8](http://www.architecturejournal.net/2006/issue8/F1_Reliability/).)

Because Service Broker communicates reliably between database queues, all of the reliability and fault tolerance that is built into SQL Server automatically applies to Service Broker messages. Whatever measures your organization takes to ensure that your database is available—clusters, SANs, transaction logs, backups, database mirroring, or what have you—also work to keep SSB messages available. For example, if you are using Database Mirroring for high availability, when your database fails over to the secondary, all of the messages fail over with it, and the queues remain transactionally consistent with the rest of the data. In addition, Service Broker understands mirroring, so that when the database fails over to the secondary, all of the other Service Brokers with which it is communicating immediately notice the change and start communicating with the secondary database.

Designing the Service Broker Application

After you have done the analysis to decide that Service Broker is the appropriate solution for meeting your customer's requirements, we can assume you are building a reliable, asynchronous, database application. If (a) you're not concerned with reliability, (b) your activities need to be synchronous, and (c) you don't have any data to store, designing a Service Broker solution that meets your requirement will be rather difficult. You can use Service Broker to implement synchronous activities or add a database to your application just to be able to use Service Broker, but in general this is probably only justified if there is a compelling need for SSB reliability, or if there are other parts of the application that already are using SSB.

The section discusses a series of decisions that you should go through when designing a Service Broker application. Not every decision is required for a given application, but it's worth thinking about all of them to ensure that you are not missing something important. These decisions are presented in a given order; in reality, however, design is a very cyclical process, and you will often have to revisit earlier decisions as you proceed to later phases of the design. I put the steps in the order in which I do them, but you might find that a different order works best for you.

Identify SSB Services

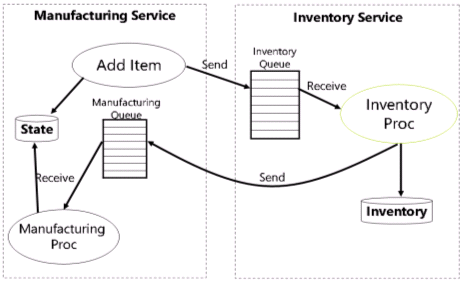
Service Broker enables asynchronous communication between services, so the first thing you have to decide is what those services should be. In many cases, the services are obvious from the problem definition. If you are using Service Broker to log CREATE TABLE events, for example, the services are the CREATE TABLE command and your logging code. The event code is part of SQL Server, so you are left with one service to design.

Most Service Broker dialogs involve three pieces of code:

1. An **initiator** that begins a dialog and sends a message. The initiator code usually is not the service that is specified in the FROM SERVICE parameter of the BEGIN DIALOG command.
2. A **target service** that receives this message, does some work, and sends a response.
3. A **response service** that handles the response message. This is the service specified in the FROM SERVICE parameter of the BEGIN DIALOG command.

It might seem strange at first that the initiator does not receive the response, but that's the nature of an asynchronous application. If the initiator waited around for the response, it would be synchronous (in fact, this is how you implement synchronous requests over an asynchronous messaging system). In an asynchronous system, the initiator kicks off the asynchronous activity and goes on to do something else. When the response comes back, it might be processing a different request or it might even be gone. If the response is handled by a different service, the service is activated when the response message arrives. The same service can handle responses from a number of initiators. A good example is an order-entry application that initiates a dialog to a shipping service to ship the ordered item. As soon as the shipping message is sent, the order-entry program can go on to handle other orders. When the shipping service responds with a ship confirmation (maybe days or weeks later), a service in the order-entry database receives the message, updates the order status to **Shipped**, and sends an e-mail to the customer.

Even if the target does not return a response message, there has to be a minimal service on the initiator side to handle Service Broker control messages, such as errors and end-dialog messages. Because of the asynchronous nature of Service Broker, a successful SEND just means that the message was put on a queue (either the target queue or sys.transmission\_queue). A SEND to a remote service that does not exist will succeed, because the service broker can’t tell the difference between a service that does not exist and a service that is not currently running. That's why the FROM SERVICE is a required parameter in the BEGIN DIALOG command. Figure 2 describes a simple Service Broker service interaction in which a manufacturing service sends an **AddItem** message to an inventory service, to add a new item to the inventory.



**Figure 2. Typical service interaction**

In a more complex application, there are many more functions involved—some synchronous and some asynchronous. Choosing which functions to make Service Broker services is important. The first candidates for SSB services are functions that do not have to complete before the main logic completes. Some examples are order shipping and billing, stock-trade settlement, and hotel and car-rental reservations for a travel itinerary. In all these cases, the original transaction might have completed long before the response is returned, and the response status is returned to the user either through out-of-band communications (e-mail) or through a status that the user can query later.

If a function must be complete before control is returned to the user, it still might make sense to use a Service Broker service if two or more services can execute in parallel. A classic example scenario is a call-center application on which I worked once. Incoming callers were identified through caller ID, and all of a customer's records from all the internal systems were retrieved so that they could be displayed to the service representative when the call was answered. The problem was that this involved remote queries into seven systems, and sometimes this would take so long that the customer would give up before the call was answered. We made this work by starting all seven queries in parallel, then returning the results when they all returned. This decreased the response time from 5 seconds to 1 second. Note that this is an "anti-scalability" approach. Instead of one thread, this used eight database threads, so our improved response time was purchased at the price of lower scalability; but the effect was not huge, and we had to do it to meet the response requirements. Another asynchronous use case might be a mortgage-application Web site where you ask the user for the size of loan that they want, then kick off a bunch of amortization table calculations in the background while you go on to ask the customer for other information. By the time they are ready to look at loan options, you have all the results ready, so that the customer thinks you're calculating them instantaneously. With a little thought, I'm sure that you can come up with dozens of similar scenarios. That's why asynchronous activities are used so often in high-performance applications.

The final point on choosing services is that it's usually a mistake to make SSB services too fine-grained. SSB messages are written to the database, so that if you design a service that makes dozens of calls to other services to execute, you might find that the database overhead is larger than the actual processing time. A Service Broker service should do a reasonably significant piece of work to justify the overhead of the message handling. There are exceptions to this if reliability, remote execution, or security-context isolation make a Service Broker service attractive, even if the service is very small. This is really no different from DCOM, in which a few large DCOM calls are much more efficient than many small DCOM calls to do the same work.

Define Dialogs

After you have defined your services, you need to define the dialogs that they use to communicate. Basically, this consists of deciding what messages are required to communicate and which services need to send them. Dialogs are more complex than the usual request-reply semantics that you're used to in DCOM or Web services, because dialogs can be long-winded conversations that involve many messages in both directions and last for months or years. A dialog should model an entire business transaction. For example, completing a purchase order might involve submitting the order, acknowledging the order, negotiating a price, negotiating ship dates, status information, ship notification, receipt acknowledgement, and billing. This transaction can continue for months and involve the exchange of dozens of messages. With Service Broker, this whole conversation should be modeled as a single dialog. The dialog will correlate and order the messages across time, so that all messages dealing with this purchase order will have the same dialog handle and conversation group ID. If you store the conversation group ID in the purchase-order headers table, your application will easily identify which PO the incoming message is for. Because dialogs are persistent, the ID stays the same, even if the dialog lasts for years.

Dialogs are reasonably cheap, but not free. Beginning or ending a dialog might incur a database write, and there's a database message to ensure that both endpoints know that the dialog is ending. For this reason, when services are engaged in a business transaction, the dialogs that are used to communicate with other services should be kept around until the service is done with them. Some very high-performance SSB applications reuse dialogs for multiple business transactions. This can significantly improve performance, but if not done right can lead to blocking issues. (There's a discussion in [my blog](http://blogs.msdn.com/rogerwolterblog/archive/2006/05/20/602938.aspx) of the issues involved.) Recycling dialogs can improve performance, but the performance comes at the price of increased complexity. If your application is simple or if maximum performance is a key requirement, you should look at recycling dialogs; but in general it's best to limit dialog lifetime to a single business transaction, unless you discover that you need a performance boost.

**Note**I have used the term business transaction several times without clearly defining it. For purposes of this paper, a business transaction is a complete activity at the business level. In the purchase-order example, the business transaction was processing the purchase order, which took many days and involved dozens of database transactions. Another example is booking a trip with a travel site. The business transaction of booking the trip involves the hotel-reservations system, the car-rental system, the airline system, the billing system, a bank or credit card system, and possibly several other systems. There are many database transactions in many databases involved in booking the trip.

Dialogs enforce ordering of the messages in the dialog. This ordering is enforced across transactions from a number of different services. It survives database restarts and failover. This is a very powerful feature that allows the application logic to rely on the order of message delivery. For example, an SSB application does not need to deal with an order line arriving before its corresponding order header, if they are in the same dialog. Another problem that dialogs can solve is mixed types of data in a message. In Web services applications, one of the more difficult problems to solve is binary data embedded in an XML document. For example, an employee message might be an XML document that contains a photograph, fingerprint, or certificate as embedded binary data. With Service Broker, you could send these as separate messages. When the application received the XML documents, it could just receive them on the same dialog and be assured that the messages would arrive in the proper order and on the same thread, because SSB ordering and locking would take care of it.

Ordering is a very powerful feature, but your design must be aware that dialog messages will always be processed in order. One of the more common questions I get involves developers who open a dialog and then start sending a bunch of messages on it. They set up activation to start many queue readers, but they find only one of them processing any messages. To ensure dialog-message order, Service Broker must use conversation-group locks to ensure that only one database transaction can receive messages from a particular dialog at a time. If you want to use the multithreaded capabilities of SSB, you must have at least as many dialogs active as you have threads. (Properly, I should have said as many conversation groups active as threads, because the conversation group is what is locked. If 10 dialogs in three conversation groups are active, only three threads can receive messages at a time.)

The decisions you make about which messages will be sent by each service are used to create the CONTRACT for the dialog. When you begin a dialog, you specify which contract Service Broker will use to govern which messages can be sent on the dialog.

Define Conversation Groups

In many cases, a dialog is a rather independent entity, but some applications will use several dialogs to complete a business transaction. We have already talked about an order-entry system in which the order-entry service communicates with a shipping service, inventory service, credit-limit service, CRM service, and billing service to complete an order. Generally, the order-entry service will begin dialogs with each of these services in parallel to optimize processing efficiency. Because the services can return messages at any time in any order, the dialogs for all these services should be put into the same conversation group. When a message on any one of these dialogs is received, the conversation-group lock that is shared by all the dialogs in the group will ensure that no other thread can process messages from any of the dialogs in this group. This will prevent issues like a credit-OK message and an inventory-status message being processed simultaneously on different threads and making conflicting updates to the order state.

I often get questions about the performance aspects of conversation-group locks. Your intuition tells you that locking all these dialogs will cause blocking and slow things down. In reality, these locks can improve performance. If two or more threads are working on the same order object simultaneously, they will have to serialize access to the database rows for the order to prevent conflicting updates, which means that two or three threads will be blocked while waiting for each other to finish. The conversation-group lock will block access to messages for a given order while one of the application threads is processing that order. This means that only one thread is involved, instead of having multiple threads blocking each other. The threads that are freed up by this can then be used to process messages for other orders, which improves the overall performance. So, you can see that conversation-group locks not only make the application logic simpler, they make it more efficient.

Define Message Types

The dialog-definition process determines which messages are required to implement the dialog, so that in this step we must decide what the contents of the message will be. As far as Service Broker is concerned, a message is a 2-GB bucket of binary data. Service Broker will do all the disassembly and assembly required to transport the message to its destination. The contracts that are used to define the contents of a dialog contain message-type definitions. The minimal message-type definition is just a name for the message type that your service can use to determine what kind of message it has received. If the message is an XML document, Service Broker can optionally check the message content to ensure it is well-formed XML or that it is valid in an XML schema.

For each message in a dialog, you must define what the message body will contain. XML is commonly used, because it makes the service more flexible and loosely coupled, but there is some overhead involved with parsing the XML. If the message contents are binary, just send it as a binary message—for example, images, music, programs, and so on. I have seen quite a few customers use a serialized .NET object as a message body. This obviously makes the initiator and target services tightly coupled, because they both have to have the same version of the object; but it is rather efficient and easy to code, so it is commonly done. If the message body is XML, you should define a schema for it as part of the design process. You might or might not want to have Service Broker validate the contents against the schema, but having one gives you the option, and a schema is an unambiguous way of telling the developers what the message has to look like.

Using a schema to validate incoming message can be fairly expensive, because each message is loaded into a parser as it is received. If your service then loads the message into its own parser to process it, each message is parsed twice. I generally recommend that schema validation be turned on for development and unit testing, but then turned off for integration testing and production. The exception to this would be a service that receives messages from a variety of untrusted sources, so that the extra parsing overhead is justified, because bad messages are rejected early.

Design Services

The last step in application design is designing the services that process Service Broker messages. While this is a big job, I won't spend a lot of time on it, because most of the effort goes into the business logic that actually processes the message contents. I'll just point out a few things that you should consider when designing your services.

Service Location

Should the service run as a stored procedure or as an external application? If it's a stored procedure, should it be CLR or TSQL? In many Service Broker applications, the service primarily does database stuff. If the service primarily does database updates and doesn't do a lot of processor-intensive stuff, it should be a stored procedure. If it does a lot of database IO, but also does a significant amount of processing, it should be a CLR stored procedure.

Services that do not do a lot of database work—or do a lot of processor-intensive work or do disk or network IOs—should generally run as external applications that connect to the database to get messages. All an application has to do to process SSB messages is open a database connection. This means that a service that does a lot of processing or network IO can run on a different box, and connect to the database server to get messages and do other TSQL stuff. Most significant business logic can run this way. Another common application is interfacing with Web services. While you can do some of this in SQL Server, the network overhead and XML-processing overhead of Web services make it attractive to do this processing on a commodity server, instead of on your very expensive database server. If the external server goes down, all of the transactions it had open roll back. The messages go back on the queue, so that if there's more than one server processing messages, everything continues without interruption. If the queue starts filling up, you can hook more commodity servers to the network to handle the load. There is a more extensive discussion in [my blog](http://blogs.msdn.com/rogerwolterblog/archive/2006/04/04/568351.aspx) of where service logic should run.

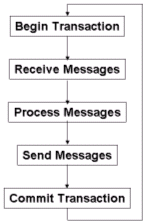
Message-Processing Loop

Almost all Service Broker services are built around the same message-processing loop. This loop is used in a number of examples, so I won't cover it extensively here. Basically, the loop has a RECEIVE command that receives one or more messages, processes the messages, SENDs any output messages, and then starts over. Most of the examples have a RECEIVE TOP (1) at the top of the loop. This makes for simple sample code, but is not necessarily the most efficient thing to do. Without the TOP (1) clause, the RECEIVE command will return all of the messages on the queue from a single conversation group.

Doing one receive command and getting back a bunch of messages is more efficient than receiving them one at a time, so it's worth considering this in your design. The reason almost none of the samples shows this is that they are simple request-reply dialogs in which only one message is sent on a given dialog, so that leaving out the TOP (1) clause wouldn't change the application behavior much. If your application sends many messages in a row on the same dialog (a logging application, for example), receiving many messages per RECEIVE statement will greatly improve efficiency.

The other bad thing that most samples do is commit the transaction at the end of each loop. Again, this will simplify your logic; and, if performance is adequate, this is the best design. But writing the commit to the transaction log is often the ultimate limiting factor on performance, so if you really need the best performance possible, you might want to commit only after a few trips through the processing loop. Doing this will improve performance, but it might increase latency, because no responses will be sent until the transaction commits. In most asynchronous operations, latency isn't too important, so this is a good trade-off. Transaction-rollback handling is tricky in this case, because you have to go back and process the messages one at a time to get around the bad one.

Figure 3 shows a typical message-processing loop.



**Figure 3. Service-logic flow**

State Handling

Read any SOA book, and you will find that services should be stateless. While this improves scaleout and performance, in reality a lot of real business transactions involve many messages over a significant time, so that state has to be held somewhere. If state isn't held persistently, it can be lost, which forces the whole business transaction to fail. With Service Broker, dialogs and conversation groups are both persistent, so that they inherently maintain state. Your application can use this fact along with the fact that the messages are in the database anyway to maintain state in a scaleable, high-performance manner. This makes long-running business transactions easier to implement.

While there are many ways to design state handling, Service Broker applications have a special advantage. All messages from any of the dialogs in a conversation group have the same conversation group ID. Because a RECEIVE command returns messages only from a single conversation group, all of the messages returned will have the same conversation group ID and—if you designed your dialogs correctly—will be associated with the same business transaction. The advantage of this is that if you store your application state in tables with the conversation group ID as the key, you can get the key to the state from any message received. This means that you can easily write a TSQL batch that will return both the messages in the queue and the state information required to process them, so that state handling is quick and easy. Also, remember that only one thread can process messages from a particular conversation group at a time, so that if the conversation group ID is the state key, only one thread will be accessing the application state at a time—making conflicting updates not an issue.

Poison Messages

A poison message is a message that can never be processed correctly. A simple example is an order header with an order number that already exists in the database. When you try to insert the header into the database, the insert will fail with a unique constraint violation, the transaction will roll back, and the message will be back on the queue. No matter how many times you try, the insert will fail, so that the service will go into a loop processing the same message over and over. This will cause the order entry to hang and can severely affect database performance. To keep a poison message from bringing your server to its knees, Service Broker will disable the queue if there are five rollbacks in a row. This allows the rest of the server to continue, but the order-entry application is dead, because the queue is disabled.

The way to avoid this is to roll back only the transaction that did the RECEIVE, if there's some hope that trying again will make the transaction succeed next time. If your transaction fails because of a lock timeout, being selected as a deadlock victim, low memory, or a similar reason, rolling back the transaction and trying again make sense. But if the error is permanent, you must handle it in your application. The most common way of handling it is ending the dialog with an error, and logging the error to an error table. The method you choose to handle poison messages depends on your application requirements, but it's important to include poison-message handling in your design.

Message Priority

I'll say up front that Service Broker doesn't have a built-in way to enforce message priority. This leads to a lot of angst among developers who are used to using message priority to ensure that certain messages are processed first. My personal experience is that most people don't really need absolute message priority in their applications. They just need to be able to ensure that high-priority messages don't get queued behind a bunch of low-priority messages. The easiest way to make that happen in Service Broker is with a high-priority and low-priority queues. You can use activation to assign enough queue readers to the high-priority queue to handle the load, and assign a single queue reader to the low-priority queue. This means that low-priority messages are processed in parallel with high-priority messages, but high-priority messages are never blocked by low-priority messages. If you need more control over priority than this approach gives you, there are other approaches discussed in my blog in these two articles, located [here](http://blogs.msdn.com/rogerwolterblog/archive/2006/03/11/549730.aspx) and [here](http://blogs.msdn.com/rogerwolterblog/archive/2006/03/17/554134.aspx).

Compensating Transactions

Service Broker services process received messages in a different transaction from—and, possibly, at a much later time than—the service that sent the message. A business transaction can include dozens of database transactions. For these reasons, you can't just roll back a business transaction if there is an error. Even if you could, undoing the effects of a business transaction generally involves much more than just reverting the database back to a previous state. To cancel an order, for example, you might have to transfer the item from shipping back to inventory (or even pull it off a truck), cancel credit card charges, send out a cancellation notice, and so forth. Your service design might have to include provisions for compensating transactions to undo the effects of an activity that errors or is cancelled. There's a more complete discussion of compensating transactions in [my blog](http://blogs.msdn.com/rogerwolterblog/archive/2006/05/24/606184.aspx), as well as in [this article in the MSDN Library](http://msdn.microsoft.com/architecture/solutions_architecture/data/default.aspx?pull=/library/en-us/dnbda/html/concurev4M.asp).

Infrastructure Aspects of a Service Broker Application

This final section covers some of the infrastructure and deployment aspects of a Service Broker solution. One of the design points of Service Broker is that the application should know as little as possible about how it is to be deployed. For example, the level of security can be determined and changed by a DBA without making changes to the application code. For this reason, it's possible to talk about SSB infrastructure issues in isolation from application-design issues, for the most part.

Because this is an architectural discussion, I won't cover the commands for setting up the infrastructure, but I will point out the infrastructure issues that you should consider when deploying a Service Broker application. Please refer to SQL Server books online or one of the Service Broker books for detailed information to do the configuration and deployment.

Security Issues

Any application that sends messages on a network connection must have a security infrastructure. Service Broker has multiple layers of security with options at each layer, so that you can tune the SSB security to what you need for your network and your data.

One security option is Dialog Security. When a secure dialog is established, Asymmetric (public and private) keys are used to authenticate the dialog connection, SQL Server permissions are verified, a session key is established for the dialog, and all messages are signed and encrypted. This ensures that the messages will be delivered unaltered and unread. Dialog security is established between the two endpoints of a dialog, so that if messages are forwarded through intermediate brokers, they won't be decrypted. This is both more secure and more efficient than SSL encryption on the wire. Dialog security should be used if message must be sent over unsecured networks.

If you feel that the message traffic for a particular dialog is so sensitive that it should be encrypted over any network, you should let the ENCRYPTION parameter in the BEGIN DIALOG command default to **On**. If the data can be sent unencrypted over some networks, you should set the ENCRYPTION parameter to **Off**. Setting this to **Off** doesn't mean that the data won't be encrypted. It means that if the DBA sets up dialog security, it will be encrypted; otherwise, it will not be. If the ENCRYPTION parameter is set to **On** (the default), no messages will be sent outside the local instance unless dialog security is configured.

Service Broker also has security at the TCP/IP connection level. All Service Broker connections require authentication, authorization, and digital signatures to ensure that the connection is authorized and that messages can't be changed on the network. The TCP/IP connection can optionally be encrypted. If the messages are already encrypted by dialog security, they will not be double-encrypted, so dialog security and transport security are complementary.

The application-deployment plan must define which dialogs and connections should be encrypted. If you use dialog security, you must include a plan for handling certificate expiration.

Availability Issues

Service Broker is often used in highly available systems, so that availability must be taken into account when designing the SSB infrastructure. Because Service Broker is part of the database engine, you make Service Broker highly available by making the database highly available. Much has been written about SQL Server availability, so I won't try to cover it here. The one unique availability feature that Service Broker offers is that it is tightly integrated with database mirroring. If a Service Broker opens a connection to a database that is mirrored, it will also open a connection to the secondary database on the mirror, so that when the primary fails over to the secondary, Service Broker will detect the change and automatically start sending messages to the new primary. Because SSB messages are transactional and stored in the database, any in-process messages and any unprocessed messages will still be on the queue after the failover, so that everything continues with no loss of data or uncompleted work.

Routing Issues

Service Broker dialogs are opened between *services*: a FROM service and a TO service specified in the BEGIN DIALOG command. A Service Broker service is basically a name for a dialog endpoint. The dialog endpoint is an SSB queue in a SQL Server database. This indirection from the service to the queue means that you can write your application to communicate between logic services and make the decision on where the services are actually located at deployment time. In fact, services can be moved to another location while dialogs are active, without losing any messages.

The mapping between the logical service name and the transport address where the messages are sent is done with a Service Broker *route*. A route is just a row in the sys.routes table in a database. When Service Broker must send a message, it looks for a route to the destination service, and passes the address down to the transport layer, which then sends it to the database where the destination service lives.

Both the initiator and target of a dialog need a route to the opposite endpoint. One of the more common SSB scenarios involves many initiators sending messages to a single target—for example, point-of-sale terminals sending transactions to the home office, or stock-trader workstations sending trades to a back-office system. Maintaining hundreds of routes on the target server to all of the initiators can be messy. To avoid this issue, SSB provides a TRANSPORT route. The initiator-service name contains the initiator's network address, and this name is used as the FROM service in the BEGIN DIALOG command. When the target wants to send a message back to the initiator, it uses the service name as the network address. In this way, the initiator provides its own return address, so that you do not have to maintain return addresses at the target system.

Service Broker also supports forwarding. Messages coming into a SQL Server instance are routed by routes in the MSDB database. If the route for a service specifies "local" as the network address, the message is routed to a service in the instance. If the network address in the MSDB route for a service specifies a TCP/IP address, the message is forwarded to the specified address. Forwarded messages are not written to the database; they are held in memory until they can be forwarded, so that SSB forwarding is very efficient. One common use of SSB forwarding is to set up a concentrator machine that accepts messages from a large number of connections, then forwards them over a single connection to the target. This moves most of the TCP/IP connection handling to the forwarder, which reduces overhead on the target machine. If you use SQL Express on the forwarder machine, this can be a very economical way to reduce overhead on the target.

Management and Monitoring

No major application is complete without provisions for monitoring and managing the application in production. This is especially important in Service Broker applications, because the reliable, asynchronous nature of SSB means that it's often difficult to tell if the application is running correctly or not. For example, if one of the services stops processing messages, Service Broker just queues up the messages for the service until it starts working again. If the operations staff does not notice that the service is not running, messages can queue up for hours.

Because Service Broker is part of SQL Server, the tools and techniques that are used to monitor SQL Server work for Service Broker. There are several perfmon counters, to measure performances and processing rates. There are quite a few trace events that can be used to trace message delivery to resolve problems. The SQL Server MOM pack also includes several Service Broker statistics. Every queue has a SQL view on it, so that you can easily find out how many messages are in a queue and where they came from.

Your Service Broker application design should include a plan for monitoring the health and performance of the application. The service should also include logic for reporting and diagnosing problems. For example, including an "echo" message in every contract that just returns a message to the service that sent it can be a useful tool for determining that the service is alive and processing messages.

Conclusions

The unique features of SQL Server 2005 Service Broker enable a whole new class of reliable, asynchronous applications. Service Broker can bring new levels of efficiency and fault tolerance to database applications.

The power of asynchronous queued operations can be used to design high-performance database applications, but the asynchronous design patterns can be difficult to master for an architect who is schooled in traditional, synchronous RPC applications. This article pointed out some of the design decisions that anyone designing a Service Broker application should consider. Designing a Service Broker application can be very different from designing a traditional application, but the results are worth the effort.