Microsoft SQL Server 9.0 Technical Articles

An Introduction to SQL Server Service Broker

Roger Wolter

February, 2005
Updated June 2005

Applies to:
Microsoft SQL Server 2005
Service Broker

Summary: This paper introduces Service Broker, a new feature in Microsoft SQL Server 2005. With Service Broker, internal or external processes can send and receive guaranteed, asynchronous messaging by using extensions to Transact-SQL. (7 printed pages)

Contents

Introduction

With Service Broker, a feature in Microsoft SQL Server 2005, internal or external processes can send and receive guaranteed, asynchronous messages by using extensions to Transact-SQL Data Manipulation Language (DML). Messages can be sent to a queue in the same database as the sender, to another database in the same SQL Server instance, or to another SQL Server instance either on the same server or on a remote server.

To better understand Service Broker, familiarity with the key concepts of queues, dialogs, conversation groups, and activation is helpful. These are discussed briefly in this section.

Queues

Service Broker uses *queues* to provide loose coupling between the message sender and the message receiver. The sender can use the SEND command to put a message in a queue and then continue on with the application, relying on Service Broker to ensure that the message reaches its destination.

Queues permit a lot of scheduling flexibility. For example, the sender can send out multiple messages for multiple receivers to process in parallel. The receivers might not process the messages until long after they were sent, but because incoming messages are queued, the receivers can process them at their own rate and the sender doesn't have to wait for the receivers to finish before continuing.

Dialogs

Service Broker implements *dialogs*, which are bidirectional streams of messages between two endpoints. All messages in a dialog are ordered, and dialog messages are always delivered in the order they are sent. The order is maintained across transactions, across input threads, across output threads, and across crashes and restarts. Some message systems ensure message order for the messages sent or received in a single transaction but not across multiple transactions, making Service Broker dialogs unique in this regard.

Each message includes a conversation handle that uniquely identifies the dialog that is associated with it. For example, an order entry application might have dialogs open simultaneously with the shipping application, the inventory application, and the billing application. Because messages from each application have a unique conversation handle, it's easy to tell which application sent each message.

Conversation Groups

Service Broker provides a way of grouping all the dialogs that are used for a particular task. This method uses *conversation groups*. In our previous order entry example, all the dialogs associated with processing a particular order would be grouped into a single conversation group. The conversation group is implemented as a *conversation group identifier*, which is included with all messages in all dialogs contained in the conversation group. When a message is received from any of the dialogs in a conversation group, the conversation group is locked with a lock that is held by the receiving transaction. For the duration of the transaction, only the thread that holds the lock can receive messages from any of the dialogs in the conversation group. This makes our order entry application much easier to write because even though we use many threads for scalability, any particular order is only processed on one thread at a time. This means we don't have to make our application resilient to problems that are caused by the simultaneous processing of a single order on multiple threads.

One of the most common uses for the conversation group identifier is to label the state that is associated with a particular process. If a process involves many messages over time, it probably doesn't make sense to keep an instance of the application running through the whole process. For example, the order entry application will scale better if, between messages, any global state that is associated with processing an order is stored in the database and retrieved when the next message associated with that order is received. The conversation group identifier can be used as the primary key in the state tables to enable quick retrieval of the state associated with each message.

Activation

You use the activation feature of Service Broker to specify a stored procedure that will handle messages destined for a particular service. When messages arrive for a service, Service Broker checks whether there is a stored procedure running that can process the messages. If there isn't a running message-processing stored procedure, Service Broker starts one. The stored procedure then processes messages until the queue is empty, after which it terminates. Moreover, if Service Broker determines that messages are arriving faster than the stored procedure can process them, it starts additional instances of the stored procedure until enough are running to keep up with the incoming messages (or until the configured maximum number is reached). This ensures that the right number of resources for processing incoming messages are always available.

Why Use Asynchronous, Queued Messaging?

Queues enable the flexible scheduling of work, which can translate to big improvements in both performance and scalability. To see how, go back to the order entry example. Some parts of an order must be processed before the order can be considered complete, such as the order header, available to promise, and order lines. But other parts realistically don't have to be processed before the order is committed; for example, billing, shipping, inventory, and so on. If the "delayable" piece of the order can be processed in a guaranteed but asynchronous manner, the core part of the order can be processed faster.

Asynchronous messaging can also provide opportunities for increased parallelism. For example, if you need to check the customer's credit and check availability for ordered items, starting both processes simultaneously can improve overall response time.

Queuing can also enable systems to distribute processing more evenly, reducing the peak capacity required by a server. For example, a typical incoming order rate might look something like this:

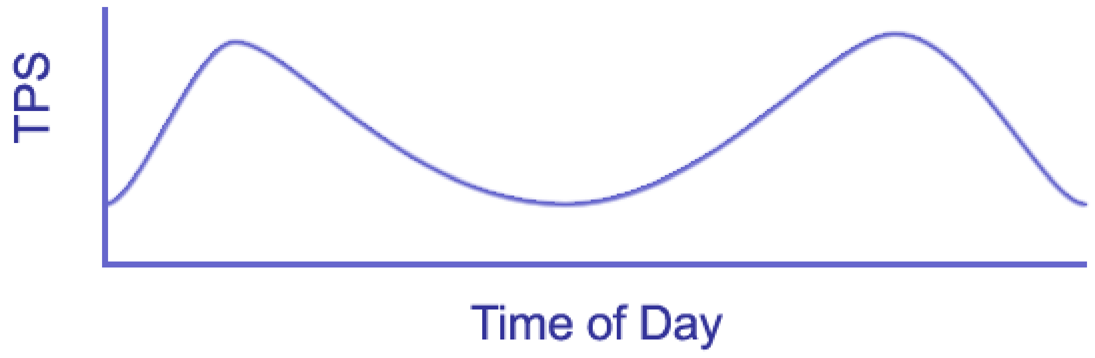

Figure 1

There is a peak at the beginning of the day and another one at the end. If each order is entered into the shipping system as it is created, the shipping system's load would look like this:

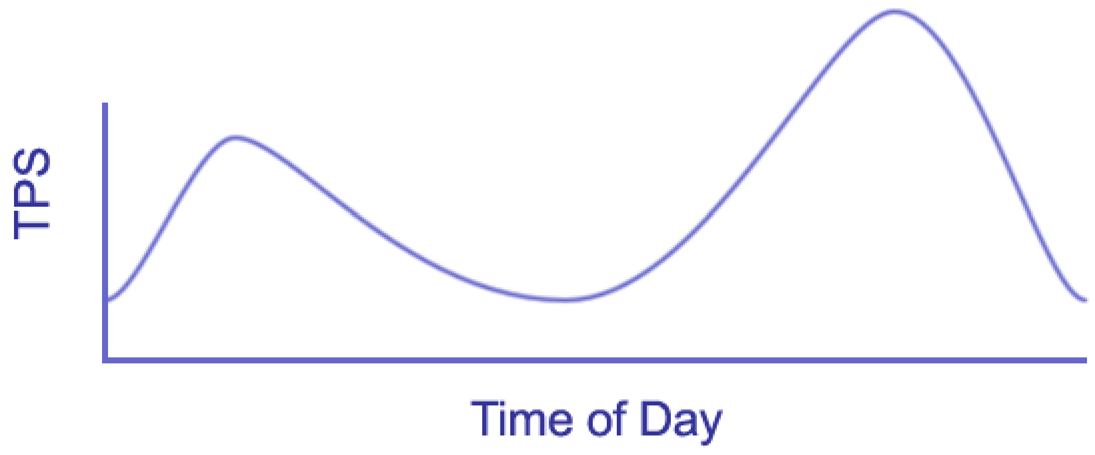

Figure 2

The afternoon peak is bigger because that's when the shipping paperwork is done for outgoing shipments. If the shipping system is connected to the order entry system with a queue, the peaks can be leveled by shifting some of the work to the slack incoming order times:

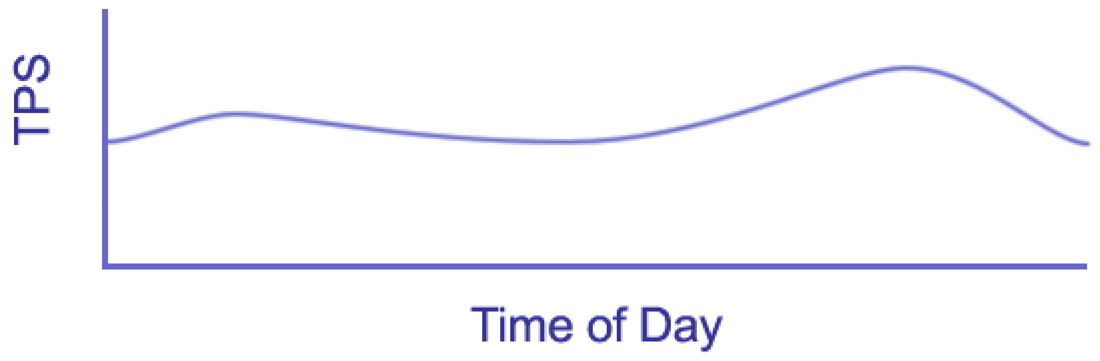

Figure 3

Why Use Transactional Messaging?

Service Broker supports transactional messaging, which means that messages are sent and received as transactions. If a transaction fails, the message sends and receives are rolled back, and don't take effect until the transaction has processed the messages successfully and committed.

Transactional messaging makes programming a messaging application much more straightforward because you can freely scatter sends and receives wherever they make sense and nothing happens until the transaction commits. If the transaction gets rolled back, none of the sends take place, and the messages that were received go back in the queue so they will be received and processed again.

How Service Broker Solves the Hard Problems in Messaging

Messaging applications have been around for a long time, and there are compelling reasons to build them. So why aren't there more of them? The answer is that messaging applications are hard to get right. SQL Server Service Broker, however, solves some of the most difficult messaging application issues: message ordering, coordination, multithreading, and receiver management.

Message Ordering

In traditional reliable messaging applications, it's easy to get messages delivered out of order. For example, application A sends messages 1, 2, and 3. Application B receives and acknowledges 1 and 3, but experiences an error with 2, so application A resends it. However, now 2 is received after 3. Traditionally, programmers dealt with this problem by writing the application so that order didn't matter, or by temporarily caching 3 until 2 arrived so the messages could be processed in order. In contrast, Service Broker handles this transparently, so all messages in a dialog are received in the order sent, with no gaps in the message sequence.

A related problem is duplicate delivery. In the previous example, if application B received message 2, but the acknowledgement message back to application A was lost, application A would resend 2 and application B would now receive 2 twice. Again, Service Broker ensures that messages are never delivered twice, even if the power is lost in the middle of a transaction.

Coordination

Messages are generally stand-alone entities, which can make it difficult to determine which conversation a message came from. For example, you may send thousands of messages to an inventory service requesting inventory updates. The inventory service may respond to some of these messages almost immediately and take a very long time to respond to others, making it difficult to decide which response message corresponds to which inventory request.

With Service Broker, by contrast, both the dialog handle and the conversation group identifier are included with every message, making it very easy to determine the order and request that each response goes with. (Some messaging systems have a correlation ID you can set to make this determination, but with dialogs, this is not necessary.)

Multithreading

One of the most difficult issues in a messaging application is making a multithreaded reader work correctly. When multiple threads are receiving from the same queue, it's always possible for messages to get processed out of order, even if they are received in order. For example, if a message containing an order header is received by thread A and a message containing an order line is later received by thread B, it's possible that the order line transaction will attempt to commit first and fail a referential constraint because the order doesn't exist yet. Although the order line message will roll back until the order header exists, it is still a waste of resources.

Service Broker solves multithreading issues by putting a lock on the conversation group when a message is read, so that no other thread can receive associated messages until the transaction commits. Service Broker makes multithreaded readers work simply and reliably.

Receiver Management

In many reliable messaging systems, the application that receives messages from a queue must be started before messages are received. In most cases, the user must decide how many application instances or threads should be running in each queue. If this is a fixed number and the message arrival rate varies, there are either too many or too few queue readers running most of the time.

Service Broker solves receiver management issues by activating queue readers as required when messages arrive. Moreover, if a reader crashes or the system is rebooted, readers are automatically started to read the messages in the queue. Service Broker does many of the same kinds of application management tasks that are typically handled by a middle-tier transaction-processing monitor.

Why Build Messaging into the Database?

Why is Service Broker part of the database engine? Wouldn't it work as well if it were an external application that used the database to store its messages? There are several reasons why the database is the right place for Service Broker to be.

Conversation Group Locking

Service Broker makes multiple queue readers possible by locking the conversation group; however, locking the conversation group with normal database commands is almost impossible to accomplish efficiently. Service Broker accordingly uses a new kind of database lock, and only Service Broker commands understand this lock type.

Remote Transactional Receive Handling

Some messaging systems restrict transactional messaging to receiving applications that are running on the same server as the queue. Service Broker, by contrast, supports remote transactional receives from any server that can connect to the database.

Common Administration

One of the issues with transactional messaging systems is that if the messages are stored in a different place than the data, it's possible for the message store and the database to get out of synch when one or the other is restored from backup. With a single database for both messages and application data in Service Broker, this is very hard to get wrong. When your data, messages, and application logic are all in the database, there is only one thing to back up, one place to set up security, and one thing to administer.

Direct Sends to Receive Queue

Because Service Broker is integrated into the database engine, a message that is addressed to another queue in any database in the same SQL Server instance can be put directly into the receive queue, bypassing the send queue and greatly improving performance.

Common Language Support

The messaging part of an application and the data part of an application use the same language and tools in a Service Broker application. This leverages the developer's familiarity with Microsoft ActiveX Data Objects (ADO) and other database programming techniques for message-based programming. With the CLR (common language runtime) stored procedures available in SQL Server 2005, stored procedures that implement a Service Broker service can be written in a variety of languages and still take advantage of the common administration benefits of Service Broker.

Single-Connection Execution Capability

Service Broker commands can be executed over the same database connection as the rest of the Transact-SQL used by the application. Using a single connection means that a messaging transaction doesn't need to be a distributed transaction, as it would have to be if the messaging system were external to the database.

Enhanced Security

Because Service Broker messages are handled internally by the database, the access permissions of the message sender can be easily checked against database objects. If the message system were an external process, a separate database connection would be required for each user who is sending messages. Having the same identity for the database and the messaging system makes security easier to set up correctly.

Transmission Queue Visibility Across Databases

Because Service Broker runs in the context of the database instance, it can maintain an aggregate view of all messages that are ready to be transmitted from all databases in the instance. This capability enables each database to maintain its own transmission queues for backup and administration while still maintaining fairness in resource usage across all the databases in the instance, something that would be very difficult, if not impossible, for an external messaging manager to do.

Conclusion

The unique features of Service Broker and its deep database integration make it an ideal platform for building a new class of loosely coupled services for database applications. Service Broker not only brings asynchronous, queued messaging to database applications but significantly expands the state of the art for reliable messaging.