http://www.sqlservercentral.com/articles/Development/anintroductiontotheservicebroker/1957/
Printed 2007/12/14 05:17AM

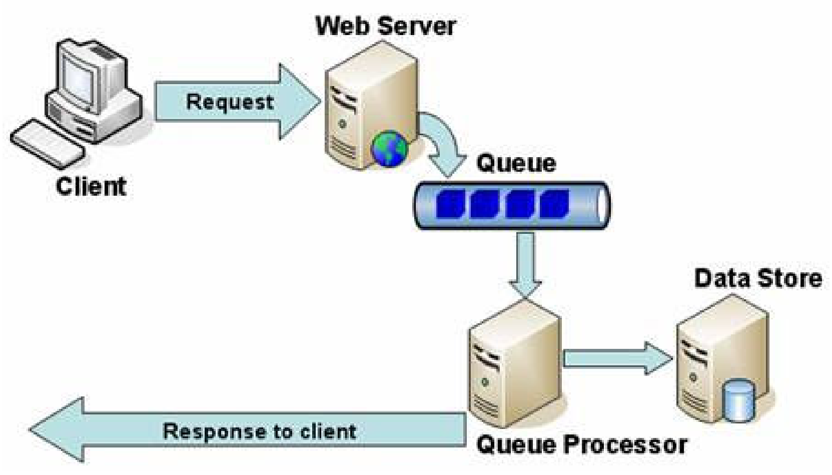
An Introduction to the Service Broker

By Srinivas Sampath, 2005/06/29

Introduction to Service Broker

Service Broker is the new kid in the block in SQL Server 2005. Before we actually delve into the features of Service Broker and write our first program, let us first imagine a scenario where we would want to fit Service Broker and then work off that.

It is often a common requirement for applications to execute processes in an asynchronous fashion. We see it all around us. For example, if you go to any online book store and order for a book, you do not wait for the order to be completed and shipped. Rather, your request is queued and then processed at a later point in time and you are free to do your other shopping. How are these applications built? The following graphic shows a simple schematic of such an application.



As shown in the figure, a client application submits a request to a web server. The web server notes the request into a queue and control is then returned to the client. The queue is read at a later point in time by a queue processor and as each message is read off a queue, the queue processor processes the message by executing whatever is the associated business logic and then persists the result of the processing in a data store. Information is then sent back to the client in some form (for example email) informing about the status of the request.

This sort of asynchronous processing is quite common in many enterprise applications and there are standard technologies that can provide this functionality. However, when building message based applications, there are a number of challenges that we need to be aware of. The following are some notable issues to be handled:

1. **Message Ordering** When a queue handles multiple messages, there needs to be some sort of order among the messages so that related messages are processed in order. For example, if order header and order line items are all posted into a queue, care must be taken that the order line items are processed after the order header and all related order items for a particular order header are processed in sequence.
2. **Transactions** This is an important requirement. When processing a message off a queue, it is quite possible that an error can be encountered. In this case, the message must not be lost; rather it must be posted back into the queue for subsequent processing.
3. **Recoverability** During the processing of messages from a queue, it is quite possible that the system can undergo an outage and in this case, the messages in the queue need to be persisted in some form to ensure

that processing resumes when the system is restored.

4. **Scalability** As messages are posted into the queue, the application must be able to process them at a rate that does not introduce a performance problem for the application. For example, if messages in a queue are posted faster than the queue processor can handle them, the system can experience a performance issue.

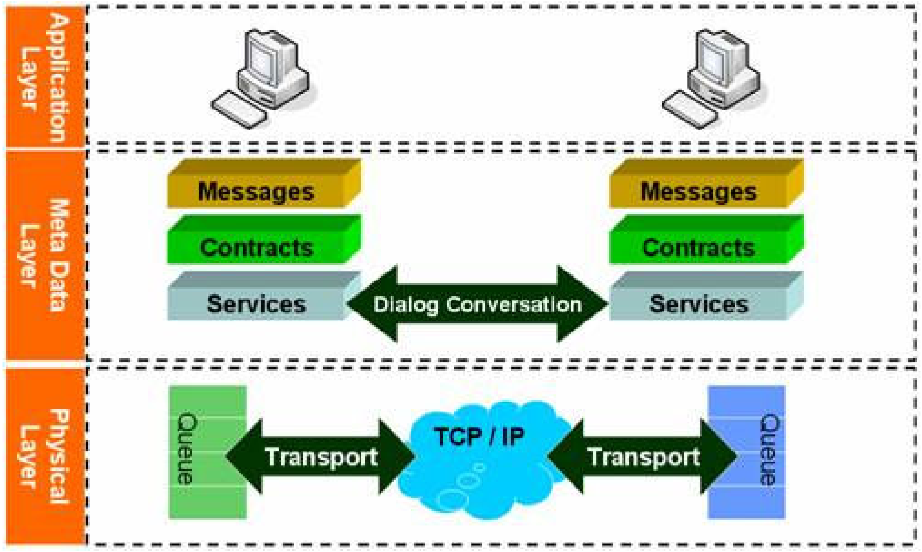
Different message processing systems handle the above challenges in different forms. However, if you need all the above features and not have to code for them, it is better to build the messaging system on an infrastructure that is designed for recoverability, maintainability, durability, performance and scalability and that can only be a **database system**. And what is a database system that ensures all of the above features? **SQL Server**, of course :)

Ok, so here is the formal definition of Service Broker. **Service Broker** is a new feature in SQL Server 2005 that allows you to build **reliable, asynchronous, message-based applications** using extensions to the T-SQL language to process and manage queues. Since queues are managed in the database, you enjoy all the capabilities of a typical database system, which is transactions, reliability, performance, scalability etc.

Service Broker can easily scale to many thousands of messages per second and it can guarantee message ordering among the various messages and even across different reading threads! All message processing is transacted and thus recoverable. Finally, message processing and posting can be distributed across many participating SQL Servers, thus ensuring true scalability and performance. In the subsequent sections of this article, we will see the building blocks of Service Broker and write our first, simple message based application.

Service Broker Building Blocks

The following figure shows the building blocks required to build a service broker application.



At the very top of the stack, the **application layer**, we have two applications that need to exchange messages. This could be our online book store and order processing application. To enable the two applications to talk to each other, we need to define a set of **meta-data** that describes how the applications will talk. The meta-data is broken into the following concepts:

1. **Messages Types** Messages are the atom of service broker applications and define the content of each message. Message types can be either binary, well formed XML or valid XML types.
2. **Contracts** A contract is a collection of messages that applications will ever exchange for their lifetime.
3. **Services** Services are the endpoints into which messages are posted and a service is associated with a contract so that the messages can be validated.
4. **Queue** A queue is the primitive that holds messages and is modeled as a table in SQL Server. Messages are read from a queue by a **service program** which then proceeds to process the messages.

5. **Transport** The transport is the underlying protocol over which messages flow between the service points and this is a proprietary binary protocol that is native to SQL Server 2005.
6. **Dialog** Two services communicate by means of a dialog. The dialog is the primitive that ensures in-order processing across transactions and sending / receiving threads and is an important aspect of service broker.

All the above meta-data are expressed using extensions to the T-SQL language. For example, there is a CREATE MESSAGE TYPE for creating message types and a BEGIN DIALOG CONVERSATION for starting a dialog between two services. We will see more of these new T-SQL commands later on, when we build our first application.

Service Broker Queue Processing

Queues in service broker are multi-reader queues and this is best illustrated with an example.

Consider for example, the queue model involved in an airport. Basically, you have a set of ticket agents who process a queue of passengers. As each ticket agent is done with one (or more) passengers, they are free to process the next person in the queue. If the queue becomes large, more ticket agents are added to ease out the load. Service Broker Queue processing is very similar to this model. For each queue bound to a service, you can specify a **queue reader** for processing messages. If the queue is expecting messages at a faster rate than it can process them, you can specify multiple queue readers. Service Broker will manage these multiple instances of readers and instantiate more if needed until the specified maximum is reached. This way, your application can scale to handle the vast amount of messages that you may expect in your queue.

A very important concept in Service Broker queue processing is that of a **conversation group**. As messages are exchanged between services, related messages can be linked by means of a conversation group. This will ensure that the messages are processed in order. For example, if a queue is receiving order header items and order detail items, you should not process the detail items unless the header items have been processed. This sort of in-order processing can be achieved by the means of a conversation group. Conversation groups also form the locking scope. All messages belonging to a particular conversation group are locked when reading, thus ensuring that they are processed in-order. Note however that this does not prevent other conversation groups to be processed in parallel.

A final point to note is that conversation groups helps in implementing **stateful applications**. If you want to maintain some type of state between message processing, the state information can be stored in a table indexed by the conversation group. Since each conversation group is unique, you can maintain state across the different messages that you want to process.

OK, that's enough of theory. Let's see some code.

Our First Service Broker Program

In this section, we will write our first service broker program. The example illustrates a simple one-way messaging application. We will just create a message and send it to a service which will then read it. Very simple, but illustrates the concepts that we discussed above.

Building a service broker program can be broken down into the following simple steps:

1. Identifying the required **message types** and the associated validation.
2. Identifying the various **contracts** that will be needed and determining the message sequence.
3. Creating the required **queues**.
4. Creating the required **services** and binding them to the appropriate queues.
5. Start passing messages around and reading them.

At each of the above steps, we will use various T-SQL extensions and our example will illustrate some aspects of each. For more information on the syntax and other options available, refer to the SQL Server 2005 Books Online.

The following is the T-SQL script batch that we will use. You can execute the script in portions (until each GO statement) to see the various artifacts of a service broker program being created.

```
-- We will use adventure works as the sample database
USE AdventureWorks
GO
-- First, we need to create a message type. Note that our message type is
-- very simple and allowed any type of content
CREATE MESSAGE TYPE HelloMessage
VALIDATION = NONE
GO
-- Once the message type has been created, we need to create a contract
-- that specifies who can send what types of messages
CREATE CONTRACT HelloContract
(HelloMessage SENT BY INITIATOR)
GO
-- The communication is between two endpoints. Thus, we need two queues to
-- hold messages
CREATE QUEUE SenderQueue

CREATE QUEUE ReceiverQueue
GO
-- Create the required services and bind them to be above created queues
CREATE SERVICE Sender
  ON QUEUE SenderQueue

CREATE SERVICE Receiver
  ON QUEUE ReceiverQueue (HelloContract)
GO
-- At this point, we can begin the conversation between the two services by
-- sending messages
DECLARE @conversationHandle UNIQUEIDENTIFIER
DECLARE @message NVARCHAR(100)

BEGIN
  BEGIN TRANSACTION;
  BEGIN DIALOG @conversationHandle
        FROM SERVICE Sender
        TO SERVICE 'Receiver'
        ON CONTRACT HelloContract
  -- Send a message on the conversation
  SET @message = N'Hello, World';
  SEND  ON CONVERSATION @conversationHandle
        MESSAGE TYPE HelloMessage (@message)
  COMMIT TRANSACTION
END
GO
-- Receive a message from the queue
RECEIVE CONVERT(NVARCHAR(max), message_body) AS message
 FROM ReceiverQueue
-- Cleanup
DROP SERVICE Sender
DROP SERVICE Receiver
DROP QUEUE SenderQueue
DROP QUEUE ReceiverQueue
DROP CONTRACT HelloContract
DROP MESSAGE TYPE HelloMessage
GO
```

Note the syntax for sending and receiving messages. The syntax for receiving a message is similar to a SELECT statement. The RECEIVE statement reads a message off the queue and deletes. This is termed as a **destructive read**. Note that you can issue a SELECT statement against a queue to see its contents. SELECT is non-destructive and is akin to peeking into a queue.

Ok, that brings us to the end of this introductory article and I hope you had a sense of the possibilities with this new programming model. Before I close the article, there is one question that constantly pops into people mind: Why do messaging in the database? The following are some valid reasons that I can think of:

1. By representing messages in databases, you can use the familiar syntax of regular database programming to queue based application also. There is no separate API to know and understand.
2. Database based messaging applications enjoy all the capabilities of a database system. For example, Service Broker applications are scalable, recoverable and maintainable.
3. Transaction semantics for messaging are built-in in the database. No special programming semantics are needed.
4. As an aside: The multiple queue reader models can help write multi-threaded applications in SQL Server. Something that was never possible in earlier versions!

Hope you use this as the starting point to explore this great new addition to SQL Server. In future articles, I shall write more about some interesting programming use cases with Service Broker.
