Service Broker: Performance and Scalability Techniques

SQL Server Technical Article

Writer: Michael Thomassy

Contributors: Sanjay Mishra

Technical Reviewers: Lindsey Allen, Lubor Kollar, Mark Souza, Thomas Kejser, Burzin Patel

Published: March 2009

Applies to: SQL Server 2008, SQL Server 2005

Summary: SQL Server Service Broker provides support for building asynchronous messaging and queuing applications with the SQL Server Database Engine. This paper describes a large scale customer scenario and the techniques employed in scaling Service Broker to process tens of thousands of messages per second on one server.

Introduction

Microsoft SQL Server Service Broker provides native support to the SQL Server Database Engine for asynchronous, transactional messaging. Finding Service Broker bottlenecks requires a similar approach to tuning any high-end OLTP database systems. This paper will describe the performance and scalability techniques applied to a real-world workload to increase overall system throughput. After you understand the workload and Service Broker system tables that are used, you'll be able to find and remove the bottlenecks to scale Service Broker applications. For more information about Service Broker, see SQL Server Books Online [http://msdn.microsoft.com/en-us/library/bb522893.aspx] .

Service Broker Customer Workload

The Service Broker workload described in this paper is based on an actual customer workload that was simplified for testing purposes. This customer workload scenario routed messages from hundreds of production user databases to a smaller number of processing databases. The primary function of the user databases is to manage live user data through a Web tier of a few thousand servers. The Web tier also *sends* Service Broker messages from these user databases. The messages are routed to a processing database that *receives* and processes the messages.

Sending Messages

Because it is a real customer scenario, there are many details left out of our test workload. Details such as the user database's stored procedures to do the Service Broker sends have logic to look up which Service Broker service was used to send messages. These stored procedures package the payload into a known Service Broker message type associated with a specific contract. Our test workload provides only one set of Service Broker services, queues, message types and contracts with one route between the two databases that send and receive messages.

Our workload sends messages immediately as they arrive. Some customers may opt to disable the receive queue, because they may want to process messages later in batch. Batch processing can improve both send and receive performance, because sends and receives are not done at the same time. One real-world customer example of batch processing is a portfolio risk management application that recalculates portfolio information after the trading day. Messages queue up during the trading day and are processed in batch at night after the target's queue reader is activated.

Receiving Messages

Specifying the Service Broker service on the initiator will map a route to the target database where the messages are received and processed. In production systems, customers will tune the number of receive activation tasks (see SQL Server Books Online topic: ALTER QUEUE – MAX_QUEUE_READERS [

http://msdn.microsoft.com/en-us/library/ms189529.aspx]) for performance reasons. Typically in high-throughput systems, the number of receive activation tasks maps to one to two times the number of CPU cores in the system. This value depends on tuning the workload performed by the receive activation tasks.

Service Broker Test Environment

The Service Broker test workload consisted of three tiers of servers, all on a shared network, with each server being a member of the same Active Directory domain.

In the diagram below from left to right, the Test Clients ran a custom C# application that generated a configurable number of threads calling a stored procedure in SSB Initiator databases. This stored procedure executed the Service Broker SEND commands, where messages were routed to the SSB Target databases on a remote server. The Target server's activation tasks received and processed the Service Broker messages writing the results to a local database on the SSB Target server.

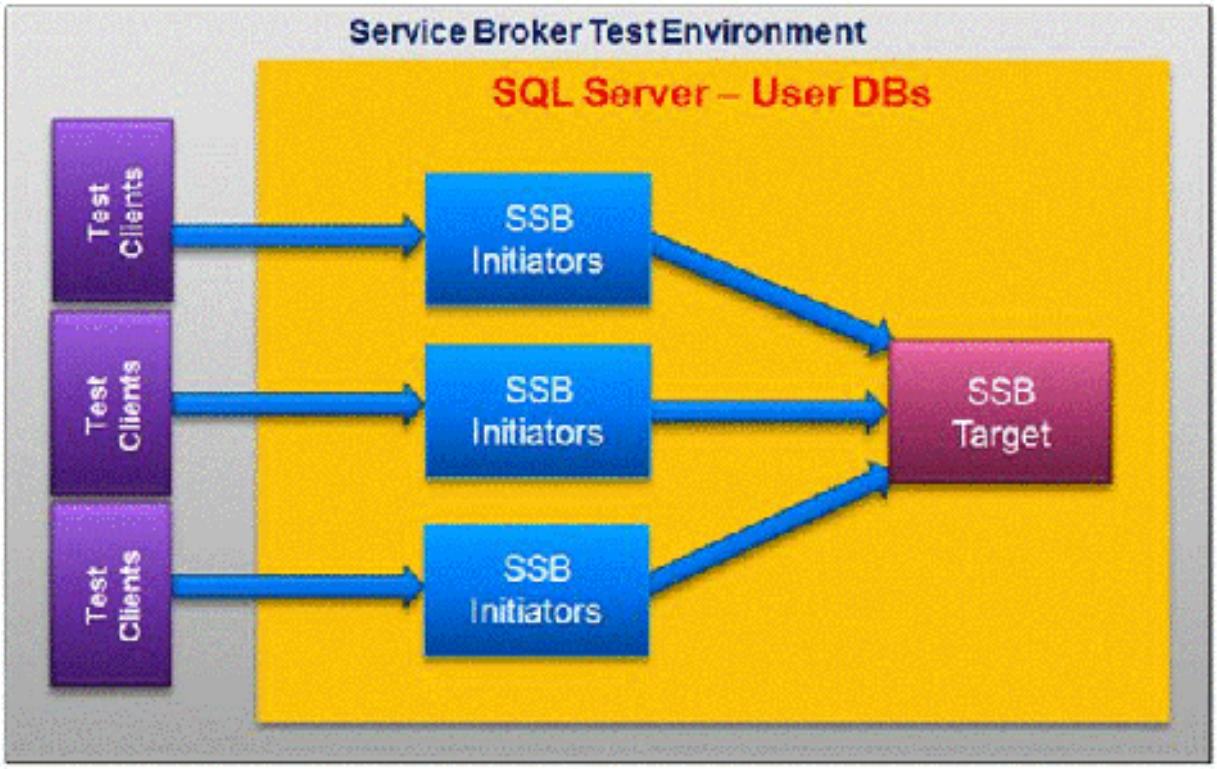

Figure 1: Service Broker test topology

Service Broker Under the Covers

Looking under the covers of Service Broker, there are a few key system tables and system views that are used to track the current state of Service Broker conversations and messages. This is how Service Broker processes messages. This section describes how Service Broker uses these system tables and views, how to monitor them, and how they can affect throughput. The table and view definitions are described in Appendix B later in this paper.

System Tables

The section describes the Service Broker system tables, views, and their associated indexes that are relevant to scaling the workload described here. The updates, inserts, and deletes affect how Service Broker is tuned.

sys.sysdesend – *system dialog endpoint send*: This table tracks each conversation when messages are sent from the initiator database. There is one clustered index on the conversation handle.

sys.sysdercv – *system dialog endpoint receive*: This table tracks conversations used on the target database. There is one clustered index on the conversation handle.

sys.sysxmitqueue – *system transmission queue*: This table shows all messages currently in the transmission queue that have been sent but haven't been transmitted to the target yet. This system table should be monitored regularly to make sure the queue is being processed and messages are not growing faster than being processed. If this table grows too large, your database files may grow!

Tech-Tip: Querying System Tables Through the DAC

The system tables can be queried through sys.objects or by joining on the sys.objects and sys.partitions tables to see the row count, with the right permissions. They can be queried directly using the dedicated admin connection (DAC). For more information, see How to: Use the Dedicated Administrator Connection with SQL Server Management Studio [http://msdn.microsoft.com/en-us/library/ms178068.aspx] and Using a Dedicated Administrator Connection [http://msdn.microsoft.com/en-us/library/ms189595.aspx] .

Transact-SQL

```
-- SSB Transmission row count
SELECT p.rows [XmitQ Depth] FROM sys.objects as o
JOIN sys.partitions AS p ON p.object_id = o.object_id
WHERE o.name = 'sysxmitqueue'

-- Admin connection to get SSB Transmission row count
SELECT COUNT(*) [XmitQ Depth] FROM sys.sysxmitqueue
```

Test Strategy

Use the Courier New font for any code blocks that you want to include.

Database Configuration

The detailed hardware specifications of the test clients and Service Broker servers are described in Appendix A later in this paper. Each of the three Service Broker Send or Initiator servers has one database, and the Receive or Initiator server has one database. The basic hardware and software configuration of these instances and databases is identical.

The database layout for Initiator and Target databases:

- 2 files: 1 data file and 1 Tx log file
- 2 LUNs: 1 LUN for data and 1 LUN for log

Each of these database servers has **tempdb** configured as follows:

- 9 files: 1 file/core = 8 data files and 1 Tx log file
- 1 LUN: 1 LUN for all **tempdb** files

Service Broker Configuration

The Initiator and Target servers have the same Service Broker configuration for the message, contract, queue, and service. The Initiator databases define a route and remote service binding to the Target database.

The Service Broker configuration script for each Initiator is below. Note that the queue specifies the primary filegroup where any can be specified. The route specifies the Target's Service Broker GUID (see **sys.databases** to get the service_broker_guid).

Transact-SQL

```
CREATE MESSAGE TYPE [//MySSB.com/Lab/M_Test] AUTHORIZATION [dbo]
VALIDATION = NONE
GO
CREATE CONTRACT [//MySSB.com/Lab/C_Test] AUTHORIZATION [dbo] ([//MySSB.com/Lab/M_Test]
SENT BY ANY)
GO
CREATE QUEUE [dbo].[MySSBLabTestQueue] WITH STATUS = ON,
RETENTION = OFF ON [PRIMARY]
GO
CREATE SERVICE [//MySSB.com/Lab/S_Test]  AUTHORIZATION [dbo]
ON QUEUE [dbo].[MySSBLabTestQueue] ([//MySSB.com/Lab/C_Test])
GO
CREATE ROUTE [Test_MYTARGETSERVER_INST2_1]
AUTHORIZATION [User_MYINITIATORSERVER_INST1_SSBSender]
WITH  SERVICE_NAME  = N'//MySSB.com/Lab/S_Test',
BROKER_INSTANCE  = N'E7044BCF-CFFF-4526-810B-5A33EB763D0B',
ADDRESS  = N'TCP:// MYINITIATORSERVER:5114'
GO
```

```
CREATE REMOTE SERVICE BINDING [Test_MYTARGETSERVER_INST2_1]
AUTHORIZATION [dbo]  TO SERVICE N'//MySSB.com/Lab/S_Test'
WITH USER = [Proxy:User_MYINITIATORSERVER_INST1_SSBSender],
ANONYMOUS = OFF
```

Service Broker Endpoint

One Service Broker endpoint is setup between any two SQL Server instances over the specified TCP port. The following code creates a Service Broker endpoint.

Transact-SQL

```
CREATE ENDPOINT [Endpoint_SRPerf_1]
      STATE=STARTED
      AS TCP (LISTENER_PORT = 5114, LISTENER_IP = ALL)
      FOR SERVICE_BROKER (MESSAGE_FORWARDING = DISABLED
, AUTHENTICATION = CERTIFICATE [PE6950-01_SSBReceiver_EP]
, ENCRYPTION = DISABLED)
```

The Service Broker endpoint users certificate authentication. The steps to create and import the certificates are:

1) Create a master certificate on both SQL Server instances in **master** (CREATE CERTIFICATE permission required).

2) Create custom certificates on both SQL Server instances in **master**.

3) Export custom certificates and Import them into each SQL Server instance's **master** database (export public key only, no private key).

4) Create endpoints specifying AUTHENTICATION = CERTIFICATE (CREATE ENDPOINT permission required, or membership in the **sysadmin** fixed server role).

Dialog Security

When messages are sent, encryption can be enabled for all Service Broker sends. The script below creates a dialog by specifying the service name and Service Broker GUID of the Target. The Service Broker GUID can be hard-coded or queried dynamically through a remote linked server query to the Target server (see comments in script the below).

Transact-SQL

```
declare @service_broker_guid uniqueidentifier
-- @service_broker_guid retrieved
-- from linked server for SSBReceiver database
declare @dh uniqueidentifier
begin dialog @dh from service [//MySSB.com/Lab/S_Test]
 to service '//MySSB.com/Lab/S_Test', @service_broker_guid
 on contract [//MySSB.com/Lab/C_Test]
 with encryption=ON;
```

Receive Activation Procedure

On the Target database for the test workload, a maximum of 16 receive tasks were specified for the queue. This is two receive tasks per core for our hardware configuration. The receive activation stored procedure is a simple I/O workload. When receiving messages, the stored procedure receives up to 10,000 messages for each call to WAITFOR – RECEIVE. All messages received were inserted into a table variable for processing. You can open a cursor on the table variable and iterate through each message to be processed. Each Service Broker message in the table variable was written to a local user table. The total message count and date/time processed were also written to a separate user table to track the total number of messages processed by for each call to WAITFOR – RECEIVE (1 to 10,000 messages).

The maximum number of receive activation tasks and the receive stored procedure are specified when the queue is created or altered.

Transact-SQL

```
CREATE QUEUE [dbo].[MySSBLabTestQueue] WITH STATUS = ON,
ACTIVATION (STATUS = ON,
PROCEDURE_NAME = [dbo].[SRPerf_Queue_LabTest],
MAX_QUEUE_READERS = 16, EXECUTE AS OWNER)
ON [PRIMARY]
```

The Service Broker queue monitor checks to spawn a new receive activation task every five seconds. The five seconds is not a configurable parameter. These Perfmon counters can be used to monitor Service Broker activation:

· **SQLServer:Broker Activation (per instance)**

o Task Limit Reached/sec – Service Broker is attempting to start a new activation task, but the maximum number of activation tasks that can run concurrently has been reached, as configured by the queue's MAX_QUEUE_READERS parameter.

o Tasks Running – The current number of activation tasks running.

o Tasks Started/sec – The number of tasks started per second.

Tech-Tip: Monitoring Activation Tasks

To monitor the currently running Service Broker activation tasks, use the dynamic management view (DMV) **sys.dm_broker_activated_tasks**, which can be joined to **sys.dm_exec_requests by session_id**.

Test Matrix

The test matrix described below has 528 permutations of unique tests. The tests show performance characteristics across 1, 2 and 3 Service Broker Initiator servers that route messages to the Target database server. For each Initiator server, there was a one-to-one mapping of test client servers. Each of the test client servers had 1 to 50 test client threads. The total thread count was between 1 and 150 threads, starting with 1 thread on 1 Initiator server stepping to 50 and then stepping the Initiator servers to 3. The total dialogs used maps to the number of threads multiplied by the total number of client threads. From 1 to 22,500 total dialogs with 3 Initiator servers X 50 client threads/server X 150 dialogs/client thread = 22,500 total dialogs. Each Initiator server had a maximum of 7,500 dialogs, and the Target server totaled 22,500 dialogs. The test payload affected performance as expected in terms of disk I/O and network utilization, where sizes from 100 bytes to 8,192 bytes were used. The performance was comparable between SQL Server 2005 and SQL Server 2008.

Initiator servers	Test client threads	Ratio of used dialogs	Payload (bytes)	SQL Server Version
1, 2, 3	1, 2, 3, 4, 5, 10, 15, 20, 30, 40, 50	1 and 150	100, 1K, 4K, 8K	2005, 2008

Table 1: Test Matrix

Performance and Scalability Techniques

During implementation of Service Broker applications, the goal is to increase throughput by optimizing for concurrency and reducing contention on global resources. Increased throughput can be gained by parallelizing both the sending and receiving of messages. Many production customers using Service Broker have implemented the techniques described in this section. The basic optimization techniques described below are a key first step to our overall solution.

Basic Optimization Techniques

Many Conversations

A conversation is required for sending a message in Service Broker. Using a single conversation for all messages impacts sends, due to the contention in the **sys.sysdesend** table. The row for the single conversation must be updated for every send. To optimize sending messages, multiple conversations should be created by the application.

Sending Multiple Messages per Conversation

One of the least optimal methods for sending messages is to send one message per conversation, where a new conversation is created for each message. Throughput is impacted even if multiple conversations are used with only one message sent per conversation. The impact is due to constant inserts and updates to the **sys.sysdesend** table. To optimize Service Broker concurrency, multiple conversations should be created and multiple messages sent per conversation.

Receiving Multiple Messages per Conversation

A conversation is required for sending a message in Service Broker. Using a single conversation for all messages impacts sends, due to the contention in the **sys.sysdesend** table. The row for the single

conversation must be updated for every send. To optimize sending messages, multiple conversations should be created by the application.

Each activation task that calls RECEIVE processes messages from only one conversation at a time. If multiple messages are sent on a conversation, the receive activation task might receive multiple messages at once from that conversation. Instead of RECEIVE TOP (1) specify RECEIVE TOP (N) where N is some number greater than 1. Because Service Broker does not support cursors on its system tables, multiple messages can be received into a table variable (tempdb) where they are immediately processed as part of one transaction.

SQL Server implicitly executes each statement as a transaction. However, the transaction scope should be explicitly set in the receive activation procedure by using BEGIN TRAN and COMMIT TRAN to include the RECEIVE call into the table variable and any subsequent processing of the messages. If an error occurs after the call to RECEIVE TOP (N), the transaction scope prevents messages from being dropped or lost, because all of the messages received are processed in this scope.

This is an example of the receive activation stored procedure.

Transact-SQL

```
create procedure [dbo].[SRPerf_Queue_LabTest]
as
begin
declare @receive table (message_type sysname,
   message_body varbinary(max),
   [dialog] uniqueidentifier)
-- Loop calling Receive, break after timeout-no messages
while 1=1
begin
      -- Transaction includes rcv msgs, process, and write to table
      begin tran
      -- Receive up to 10K msgs, wait for 3 secs for msgs
      waitfor (receive top (10000)
                  message_type_name,
                  message_body,
                  conversation_handle
            from MySSBLabTestQueue
            into @receive
            ), timeout 3000;
      -- If no messages, exit SP
      if @@rowcount = 0
      begin
            rollback tran
            break
      end
      -- Copy 2K bytes of payload to table
      insert into dbo.messageDrop (messageVal)
      select cast(message_body as varbinary(2000))
      from @receive
      -- Delete temp table
      delete from @receive
      commit tran
end -- end while
end -- end procedure
```

Tech-Tip: Receive Activation and Try-Catch

It's a best practice to have a TRY-CATCH block in the receive activation stored procedure. If an error occurs in the receive activation stored procedure, never do a rollback. This will avoid poison messages impacting Service Broker. Write the problem message to a user-defined error table, which you can later use to analyze and determine the cause of the error. The method employed above, where many messages are processed at once, may yield erroneous messages. Log these to the user-defined error table while processing the rest of the messages, all in the scope of one transaction that always commits the transaction.

Advanced Optimization Techniques

The 150 Trick

A conversation is required for sending a message in Service Broker. Using a single conversation for all messages impacts sends, due to the contention in the **sys.sysdesend** table. The row for the single conversation must be updated for every send. To optimize sending messages, multiple conversations should be created by the application.

In our tests, we described up to three test client computers each with up to 50 threads. Even with three

Initiators, the Initiator was always the bottleneck, not the Target. Here are the reasons:

· The Initiator does more transactions than the Target per message. Essentially, there is one message per transaction to write the message to the transmission queue. Then there is one transaction to remove the message from the Initiator's transmission queue and update the **sys.sysdesend** table when the message is routed to the Target database server. The Target does fewer transactions because the receive activation stored procedure processes many messages (up to 10,000 for our tests) in one transaction.

· The Initiator waittypes are higher for PAGELATCH_EX/SH and WRITELOG. After a test, the total wait time for these waittypes was much less on Target. This means the Initiator was doing most of the waiting.

50 test client threads = 150 dialogs/client thread X 50 client threads is 7,500 total dialogs. This technique will store every 150th dialog in the "dbo.dialogs" user table. For sending messages, we ignore the other 149 dialogs. 50 "usable dialogs" will require 400 KB of memory (that is, 50 * 8 KB).

· **sys.sysdesend** – row data 47 bytes + row header 7 bytes = 54 bytes, at most 144 rows per page, so we rounded up to 150. That is 150 rows per 8K database page.

· **sys.sysdercv** – much bigger row size than **sys.sysdesend** therefore it's not a concern for becoming a bottleneck.

The script below shows how to create 150 dialogs and save every 150th dialog that will be used.

Transact-SQL

```
declare @i int, @k int, @dh uniqueidentifier
select @i = 0, @k = 0;
while (@i < 7500)
begin
      set @i = @i + 1
      begin dialog @dh from service [//MySSB.com/Lab/S_Test]
       to service '//MySSB.com/Lab/S_Test', @service_broker_guid
       on contract [//MySSB.com/Lab/C_Test]
       with encryption=ON;
      if ((@i % 150) = 0)
      begin
            set @k = @k + 1
            insert into dbo.dialogs values (@k, @dh)
      end
end
```

Warm-up System Tables

When a conversation is created a row is added to the **sys.sysdesend** table. A message must be sent to create a corresponding row for the conversation in the remote server's **sys.sysdesrcv** table on the Target server.

Send one message on every conversation that is created otherwise **sys.sysdercv** could become a bottleneck.

Transact-SQL

```
declare @dh uniqueidentifier, @i bigint
declare @s varchar (50)

select @i = 0;
set @s = replicate ('x', 50)

declare conv cursor for
select [conversation_handle] from sys.conversation_endpoints;
open conv;
fetch next from conv into @dh;

while @@fetch_status = 0
begin
      ;send on conversation @dh message type
     [//MySSB.com/Lab/M_Test] (@s)
      fetch next from conv into @dh;
      set @i = @i + 1
end
close conv;
deallocate conv;
```

Test Results

This section provides the overall throughput of the test system under different load conditions. The results provide guidance for when an application can benefit from the 150 Trick. To benefit, the application would have to generate enough concurrent load.

Service Broker Throughput

The test configuration started with one Target and one Initiator server. There was one test client machine generating the workload with 1 to 50 threads. Using the 150 Trick enabled the Target server to receive and process almost 5,000 messages per second, as show in Figure 2.

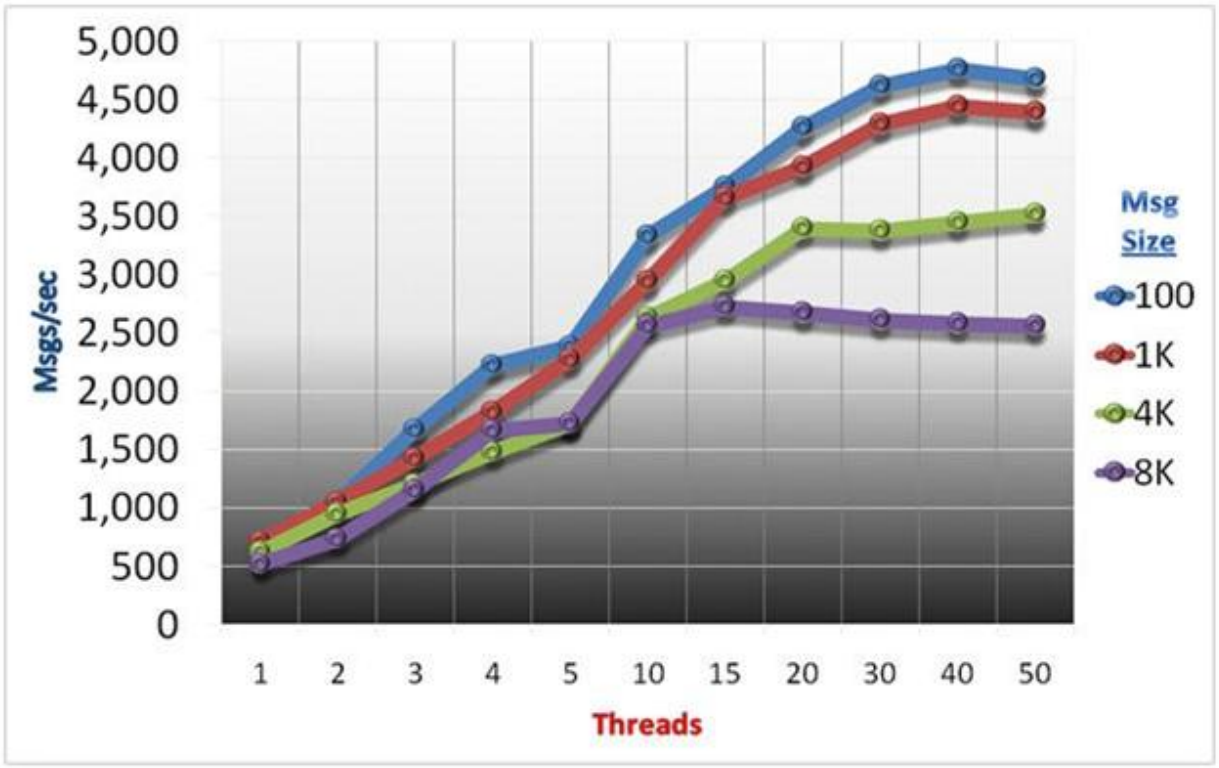

Figure 2: Service Broker messages per second processed by one Target. The number of threads for one Initiator increased throughput. Messages of four different sizes are compared.

Two Initiators and One Target

When a second Initiator server with another test client computer was added, the throughput increased to almost 12,000 messages per second.

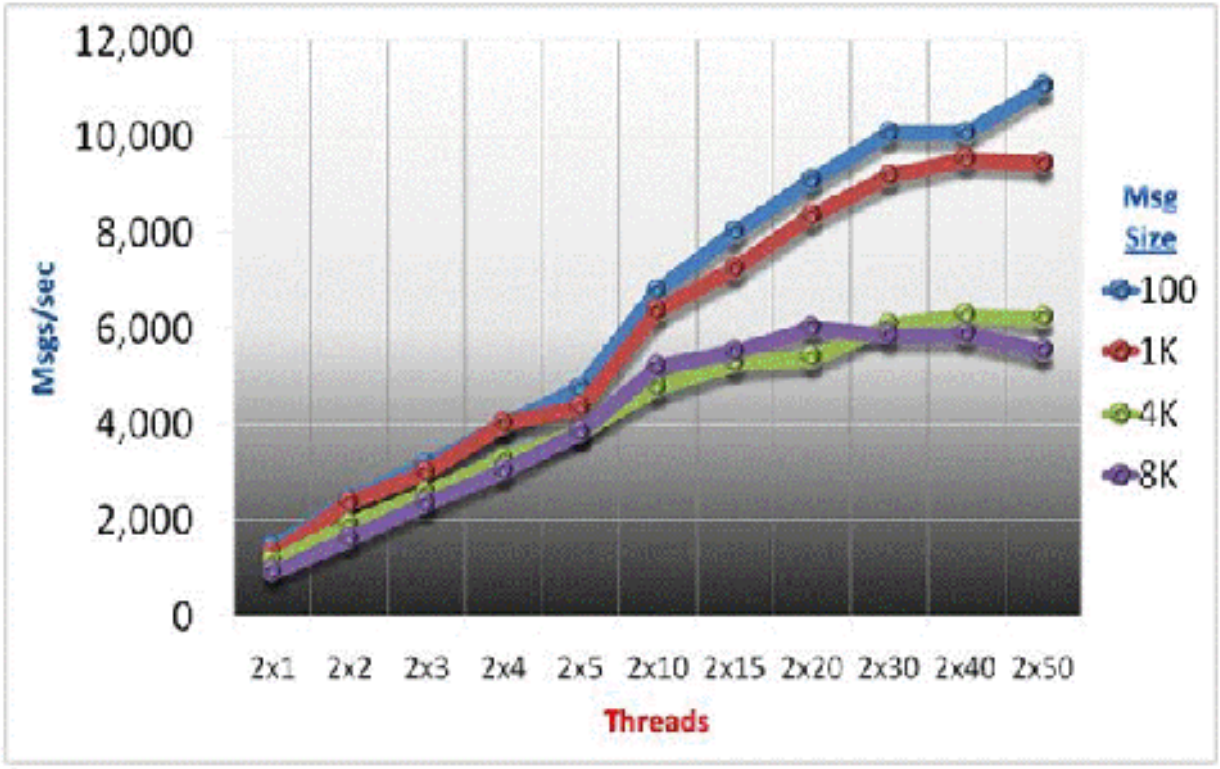

Figure 3: Two Initiator servers and two test client computers where each test client had 1 to 50 threads each.

For example, 2x1 threads means 2 Initiators each with 1 thread executing Service Broker sends; 2 threads total. 2x50 threads means 2 Initiators each with 50 threads executing Service Broker sends; 100 threads total.

Three Initiators and One Target

Figure 4 below shows adding the third Initiator server with another test client computer with almost linear scale increase of the messages per second. Note that the larger message sizes begin to plateau at 45 (3x15) concurrent threads. Peak throughput was over 18,000 messages per second with smaller-size messages.

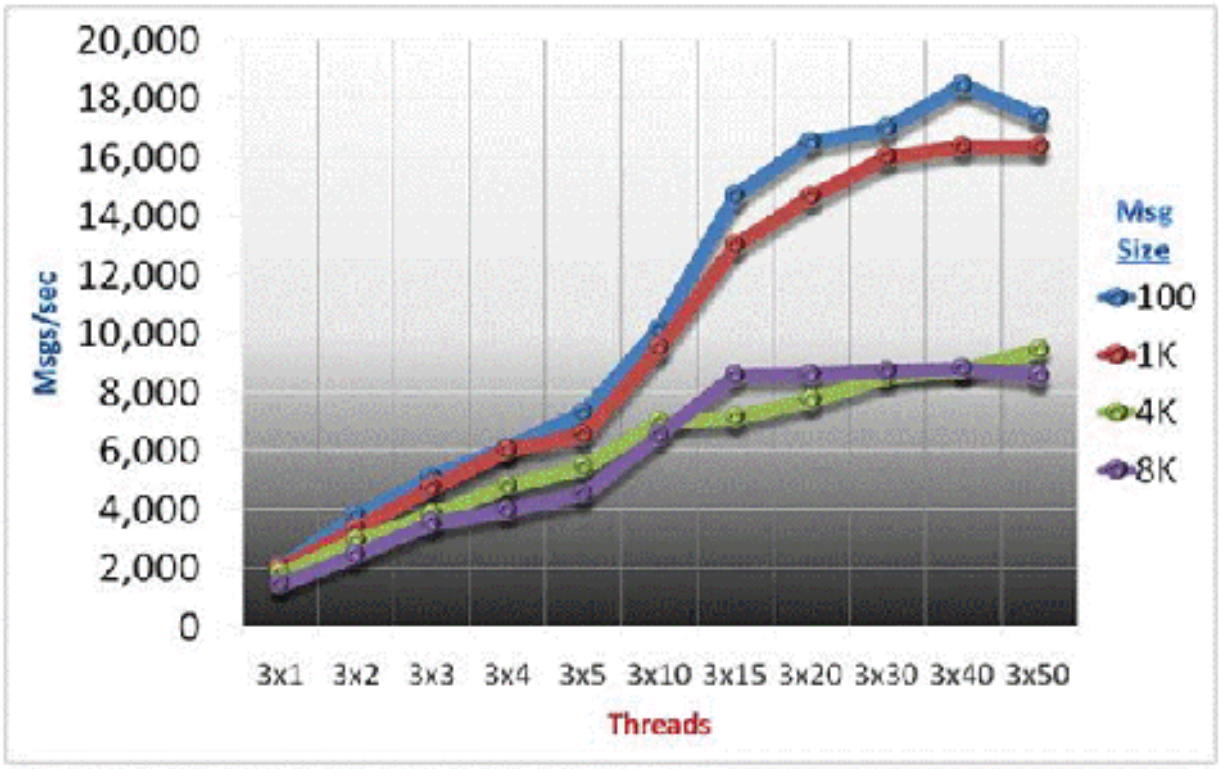

Figure 4: Three Initiator servers and three test client computers where each test client had 1 to 50 threads each. One Target server was used for all tests.

Initiator and Target Server Performance

Each of the three Initiator servers displayed the same performance characteristics. Because only one test client computer was used per Initiator server, the load on each Initiator was the same. The average CPU utilization was about 50%. The top wait stat was PAGELATCH_EX/SH without the 150 Trick. Using the 150 Trick, the top wait stat became WRITELOG.

The Target server with three Initiators had an average CPU utilization of 45%. The top wait stat was WRITELOG.

1 Dialog vs. 150 Dialogs

The graph below shows a comparison of test runs with and without the 150 Trick. There's as much as a 315 percent increase in performance when using more than 15 (3x5) client threads and using every 150th dialog. The 150 Trick proved not be beneficial at 15 threads or less, because there is low contention on the sys.sysdesend table. If more than 15 threads access the sys.sysdesend table in parallel at high-volume, there is page latch contention.

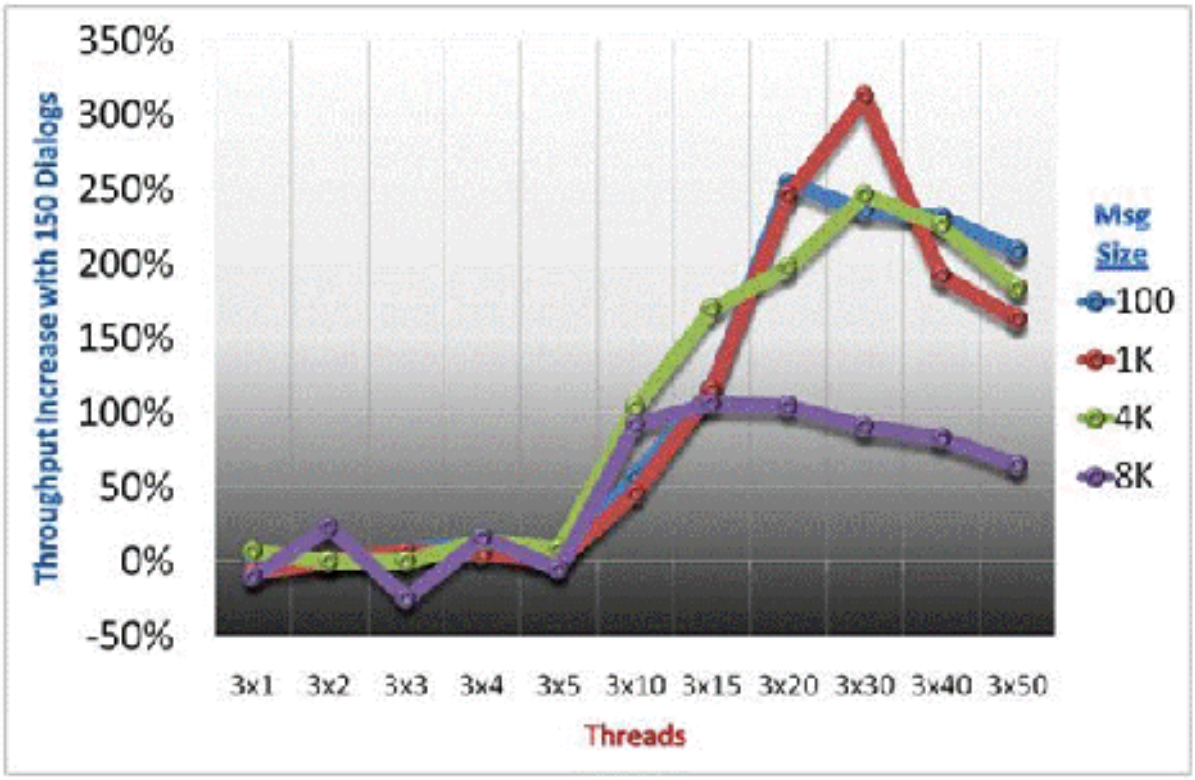

Figure 5: Shows increase in throughput by comparing 1 dialog used versus every 150th dialog.

The throughput increase in using every the 150 Trick starts to be noticeable at 30 (3x10) concurrent threads or greater. The recommendation is only to use the 150 Trick at or beyond 30 concurrent threads that are able to generate a high rate of messages.

Recommendations

Here is a summary of the techniques used to scale Service Broker. Each technique builds on its predecessor to increase overall performance and scalability:

1) Dialog reuse: Send multiple messages per conversation, and use multiple conversations.

2) Process multiple messages: The receive activation process should RECEIVE TOP N, not RECEIVE TOP 1. Each receive call processes messages from only one conversation; therefore receiving N messages assumes multiple messages have been sent per conversation.

3) Activation task lifetime: Service Broker checks whether it needs to spawn a new receive activation task every 5 seconds. The time-out value for the RECEIVE call should be set appropriately (1 second, 3

seconds, or whatever meets your need) to keep the activation task active and in memory, which will avoid the 5-second delay and the overhead of starting a new activation task.

4) Activation tasks per core: For large scale workloads, define the maximum number of queue readers for one to two per core. The actual number depends on your workload and its utilization of system resources, including disk I/O, memory, CPU, and network.

5) The 150 Trick: The key technique described in this paper is to avoid the classic page hot-spot problem on the **sys.sysdesend** system table. Using every 150th dialog when sending and receiving a high volume of messages removes the latch contention on this table.

Conclusion

Service Broker can be optimized by using standard performance-tuning techniques that would be applied to any database. The techniques described in this paper enable Service Broker to scale to almost 19,000 messages/second on a single Target server while writing all of the messages to disk. Understanding the Service Broker design is required to implement these techniques.

What are the concerns associated with the 150 Trick and creating 150 dialogs per usable dialog? Allocating 150 dialogs uses about one database page or 8KB. Therefore, creating 200 usable dialogs or 30,000 (200x150) total dialogs as described by the 150 Trick amounts to 1.6 MB of SQL Server's memory. The concern is not the size but the manageability aspect, which can be complex and should be factored into the overall cost of implementing the 150 Trick.

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

- Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
- Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

Send feedback [mailto://microsoft.com:25/default.aspx?subject=White%20Paper%20Feedback:%20Service%20Broker:%20Performance%20and%20Scalability%20Techniques] .

Appendix A: Hardware Environment

Service Broker can be optimized by using standard performance-tuning techniques that would be applied to any database. The techniques described in this paper enable Service Broker to scale to almost 19,000 messages/second on a single Target server while writing all of the messages to disk. Understanding the Service Broker design is required to implement these techniques.

Database Servers

Four DELL PowerEdge 6950 servers. Each server consists of:

· 4 dual-processors @2.8 MHz, AMD

· 64 GB RAM

· 2 HBAs, 2 Gbs fiber channel

· 1 NIC, Broadcom NetXtreme II GigE

· The Windows Server 2003 Enterprise x64 Edition operating system with Service Pack 2 (SP2)

· SQL Server 2008 Enterprise CTP6 and SQL Server 2005 Enterprise Edition with SP2

Storage

A 3 PAR SAN was used with the following configuration on each server:

· 16GB Data Cache, 4GB Control Cache. Data Cache is dynamically allocated based on I/O patterns.

· 240 disks, each 147 GB, 10K RPM

· 12 Ports, 2 directly attached per server

- All LUNs are striped across all disks
- 3PAR InServ S400 Overview: http://www.3par.com [http://www.3par.com/default.aspx]

SQL Server Database Storage Layout

Purpose	Drive	RAID	# LUNs	Total GB
Log	L:	1+0	1	50
Data	M:	1+0	1	300
Log	N:	1+0	1	50
Data	O:	1+0	1	300
Tempdb	T:	1+0	1	30

Table 2: Service Broker Send Server (Initiator DB)

Purpose	Drive	RAID	# LUNs	Total GB
Log	L:	1+0	1	50
Data	M:	1+0	1	300
Tempdb	T:	1+0	1	30

Table 3: Service Broker Receive Server (Target DB)

Test Client Computers

Three Dell PowerEdge 1750 servers:

- 2 processors @3.1 GHz, Single core
- 4 GB RAM
- 1 NIC, Broadcom NetXtreme Gigabit Ethernet
- Windows Server 2003 Enterprise Edition with SP2, x86

Appendix B: Service Broker System Tables and System Views

System Tables

sys.sysdesend

handle uniqueidentifier 16

diagid uniqueidentifier 16

initiatortinyint 1

sendseq bigint 8

sendxact binary 6

Clustered, unique (handle)

sys.sysdercv

diagid uniqueidentifier 16

initiatortinyint 1

handle	uniqueidentifier	16
rcvseq	bigint	8
rcvfrag	int	4
status	int	4
state	char	2
lifetime	datetime	8
contract	int	4
svcid	int	4
convgroup	uniqueidentifier	16
sysseq	bigint	8
enddlgseq	bigint	8
firstoorder	bigint	8
lastoorder	bigint	8
lastoorderfr	int	4
dlgtimer	datetime	8
dlgopened	datetime	8
princid	int	4
outseskey	varbinary	56
outseskeyid	uniqueidentifier	16
farprincid	int	4
inseskey	varbinary	56
inseskeyid	uniqueidentifier	16
farsvc	nvarchar	512
farbrkrinst	nvarchar	256
priority	tinyint	1

Clustered, unique (diagid, initiator)

sys.sysxmitqueue

dlgid	uniqueidentifier	16
finitiator	bit	1
tosvc	nvarchar	512
tobrkrinst	nvarchar	256
fromsvc	nvarchar	512
frombrkrinst	nvarchar	256
svccontr	nvarchar	512
msgseqnum	bigint	8
msgtype	nvarchar	512

unackmfn	int	4
status	int	4
enqtime	datetime	8
rsndtime	datetime	8
dlgerr	int	4
msgid	uniqueidentifier	16
hdrpartlen	smallint	2
hdrseclen	smallint	2
msgenc	tinyint	1
msgbodylen	int	4
msgbody	varbinary	-1

Clustered, unique (dlgid, finitiator, msgseqnum)

System Views

sys.conversation_endpoints

Joined on sys.sysdesend and sys.sysdercv.

sys.trasmission_queue

Joined on sys.sysxmitqueue, sys.sysdesend and sys.sysdercv.

Appendix C: References

· SQL Server Service Broker Team Blog

http://blogs.msdn.com/sql_service_broker [http://blogs.msdn.com/sql_service_broker.aspx]

· SQLCAT Tech Note on SSB DB Identity Uniqueness

http://sqlcat.com/technicalnotes/archive/2008/08/12/sql-server-service-broker-maintaining-identity-uniqueness-across-database-copies.aspx [http://sqlcat.com/technicalnotes/archive/2008/08/12/sql-server-service-broker-maintaining-identity-uniqueness-across-database-copies.aspx]

· Fire-and-Forget

http://rusanu.com/2006/04/06/fire-and-forget-good-for-the-military-but-not-for-service-broker-conversations/ [http://rusanu.com/2006/04/06/fire-and-forget-good-for-the-military-but-not-for-service-broker-conversations/]

· Reusing Conversation

http://rusanu.com/2007/04/25/reusing-conversations/

· Recycling Conversation

http://rusanu.com/2007/05/03/recycling-conversations/

· Resending Messages

http://rusanu.com/2007/12/03/resending-messages [http://rusanu.com/2007/12/03/resending-messages]

· Dynamic Routing

http://rusanu.com/2007/07/31/dynamic-routing-service/ [http://rusanu.com/2007/07/31/dynamic-routing-service/]

· Service List Manager

http://www.codeplex.com/slm/ [http://www.codeplex.com/slm/]

· Troubleshooting dialogs

http://rusanu.com/2005/12/20/troubleshooting-dialogs [
http://rusanu.com/2005/12/20/troubleshooting-dialogs]

http://rusanu.com/2007/11/28/troubleshooting-dialogs-the-sequel/ [
http://rusanu.com/2007/11/28/troubleshooting-dialogs-the-sequel/]

· Error Handling

http://rusanu.com/2007/10/31/error-handling-in-service-broker-procedures/ [
http://rusanu.com/2007/10/31/error-handling-in-service-broker-procedures/]

· "The Rational Guide to SQL Server 2005 Service Broker" by Roger Wolter

Publisher: Rational Press (June 30, 2006)

ISBN-10: 1932577270

ISBN-13: 978-1932577273