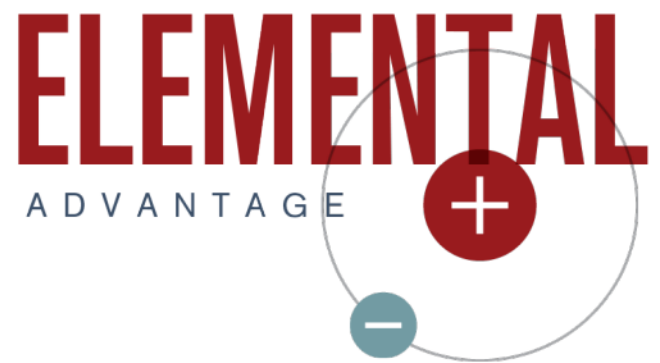




Sonic[®]



Aurea[®] SonicMQ[®]
Performance Tuning Guide

Notices

For details, see the following topics:

- [Notices](#)
- [Third-party Acknowledgments](#)

Notices

Copyright © 1999 – 2015. Aurea Software, Inc. (“Aurea”). All Rights Reserved. These materials and all Aurea products are copyrighted and all rights are reserved by Aurea.

This document is proprietary and confidential to Aurea and is available only under a valid non-disclosure agreement. No part of this document may be disclosed in any manner to a third party without the prior written consent of Aurea. The information in these materials is for informational purposes only and Aurea assumes no responsibility for any errors that may appear therein. Aurea reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of Aurea to notify any person of such revisions or changes.

You are hereby placed on notice that the software, its related technology and services may be covered by one or more United States (“US”) and non-US patents. A listing that associates patented and patent-pending products included in the software, software updates, their related technology and services with one or more patent numbers is available for you and the general public’s access at www.aurea.com/legal/ (the “Patent Notice”) without charge. The association of products-to-patent numbers at the Patent Notice may not be an exclusive listing of associations, and other unlisted patents or pending patents may also be associated with the products. Likewise, the patents or pending patents may also be associated with unlisted products. You agree to regularly review the products-to-patent number(s) association at the Patent Notice to check for updates.

Aurea, Aurea Software, Actional, DataXtend, Dynamic Routing Architecture, Savvion, Savvion Business Manager, Sonic, Sonic ESB, and SonicMQ are registered trademarks of Aurea Software, Inc., in the U.S. and/or other countries. DataXtend Semantic Integrator, Savvion BizLogic, Savvion BizPulse, Savvion BizRules, Savvion BizSolo, Savvion BPM Portal, Savvion BPM Studio, Savvion Business Expert, Savvion ProcessEdge, and Sonic Workbench are trademarks or service marks of Aurea Software, Inc., in the U.S. and other countries. Additional Aurea trademarks or registered trademarks are available at: www.aurea.com/legal/.

The following third party trademarks may appear in one or more Aurea® Sonic® user guides:

Amazon is a registered trademark of Amazon Technologies, Inc.

Eclipse is a registered trademark of the Eclipse Foundation, Inc.

HP-UX is a registered trademark of Hewlett-Packard Development Company, L.P.

IBM, AIX, DB2, Informix, and WebSphere are registered trademarks of International Business Machines Corporation.

JBoss is a registered trademark of Red Hat, Inc. in the U.S. and other countries.

Linux is a registered trademark of Linus Torvalds.

Microsoft, Active Directory, Windows, and Visual Studio are registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Netscape is a registered trademark of AOL Inc.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

Progress and OpenEdge are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries.

Red Hat and Red Hat Enterprise Linux are registered trademarks, and CentOS is a trademark of Red Hat, Inc. in the U.S. and other countries.

SUSE is a registered trademark of SUSE, LLC.

Sybase is a registered trademark of Sybase, Inc. in the United States and/or other countries.

Ubuntu is a registered trademark of Canonical Limited in the United States and/or other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

All other marks contained herein are for informational purposes only and may be trademarks of their respective owners.

Third-party Acknowledgments

Please see the 'notices.txt' file for additional information on third-party components and copies of the applicable third-party licenses.

Table of Contents

Preface.....	9
About this Documentation.....	9
Typographical Conventions	10
Aurea® Sonic® Documentation.....	11
Aurea® SonicMQ® Documentation.....	11
Other Documentation in the Aurea® SonicMQ® Product Family.....	12
Worldwide Technical Support	13
 Chapter 1: Introduction.....	 15
SonicMQ® Messaging.....	16
Distributed Architecture.....	16
Asynchronous Messages.....	16
Buffering.....	17
Messages, Blocks, and Bytes.....	18
Buffer Sizes and Threshold Behavior.....	20
Message Buffers and JMS Priorities.....	21
 Chapter 2: Using Tuning Parameters and Properties.....	 23
Accessing Broker Properties with the Graphical Tool.....	23
Method Name Hints for Administrative Programming.....	26
Settings Listed in This Manual.....	27
Exposed Properties	27
Advanced Properties.....	28
 Chapter 3: Subscriptions and Durable Subscriptions.....	 29
Overview.....	29
Subscriptions.....	30
OutgoingMsgBufferSize Property.....	31
Max Connection Send Buffer Size Property.....	32
TcpNodelay Property.....	32
Durable Subscriptions.....	33
Disconnected Durable Subscriptions.....	33
Connected Durable Subscriptions.....	33
Flow to Disk Publishing.....	38
Non-Durable Subscribers.....	39
Flow To Disk Settings.....	39

Chapter 4: Queues.....43

Overview.....	44
QueueMaxSize Property.....	45
QueueSaveThreshold Property.....	46
QueueDeliveryThreads Property.....	47
System Queues Used in Dynamic Routing.....	47
DISCONNECTED_PENDING_SAVE_THRESHOLD Property.....	48
FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD Property.....	48
Temporary Queues.....	49
MaxTemporaryQueueSize Property.....	49
Expiring Messages in Queues	50
EnableQueueCleanup Property.....	50
QueueCleanupInterval Property.....	50

Chapter 5: Storage.....53

Overview.....	54
Guaranteed Message Stores.....	54
Storage Tuning Properties.....	55
MaxTopicDbSize Property.....	55
Recovery Log.....	57
RecoveryLogPath Property.....	58
RecoveryLogMaxFileSize Property.....	58
RecoveryLogQueueSize Property.....	59
RecoveryLogBlockSize Property.....	60
RECOVERY_LOG_FLUSH_DELAY.....	61
Data Storage.....	61
PsDbQueueSize Property.....	62
PtpDbQueueSize Property.....	62
Using Fast Disks.....	63
public void sync() throws SyncFailedException	64
Multiple Storage Devices.....	65
RecoveryLogPath Property.....	65
DbConnect Property.....	65
Avoid Disk Write Caches.....	66

Chapter 6: Transactions.....67

Transaction Threads and Buffer Size.....	68
TxnBufferSize Property.....	69
TXN_FLUSH_THREADS Property.....	69
TxnPrimaryThreads Property.....	70
TxnSecondaryThreads Property.....	70
Message Batching (TxnBatchSize Properties).....	71

Setting Message Batching in the ConnectionFactory.....	71
Setting Message Batching in the Session.....	72
Chapter 7: TcpNodelay: Nagle Algorithm.....	75
Trade-offs and Risks.....	76
Enabling and Disabling the Nagle Algorithm.....	76
Chapter 8: Message Size.....	79
Overview.....	80
Estimating Message Sizes.....	80
Tuning Table.....	81
Chapter 9: HTTP Dispatch Threads.....	83
Thread Pool for HTTP Direct Outbound.....	83
Effects on Memory Usage.....	85
Chapter 10: Tuning JMS Clients.....	87
Asynchronous Message Delivery Set on Connections.....	88
Low Latency Messaging.....	88
Managing Flow Control with Topic Messages.....	89
Using Client Persistence With Wait Time.....	89
Disabling Flow Control for Publishers.....	89
Send Timeout for Message Producer.....	89
Controlling Flow-to-disk.....	90
Using Discardable Delivery Mode.....	90
Using Shared Subscriptions.....	90
Acknowledgement Mode.....	90
Using Acknowledge and Forward.....	91
Using Only Needed Security and Privacy.....	91
Tuning Client QoS Cache Settings.....	91
Shared vs. Dedicated Connections.....	92
Using Queue Prefetch.....	92
Setting Brokers to Clean Up Connections Not Detected on the Socket.....	92
Chapter 11: Tuning Replicated Broker Connections	95
Backing Up Only Selected Brokers.....	95
Tuning Replication Connection Properties.....	96
Adjusting Intervals and Timeouts.....	96
Replicating Persistent.....	96

Chapter 12: Tuning the JVM Properties.....97

Choosing a Java Virtual Machine for the SonicMQ® Broker.....	98
Setting the Java Heap Size.....	98
Metrics that Measure Heap Size.....	98
Using the Maximum Available Memory for a Messaging Broker.....	99
Adjusting Heap for the Directory Service in a Large Deployments.....	99
Determining Messaging Broker Memory Requirements.....	100
Calculating Broker Memory Use.....	100
Tuning JVM Parameters.....	101
Tuning Garbage Collection To Reduce Latency Spikes.....	102
Guidelines for Tuning Garbage Collection.....	103
Using GC PrintTenuringDistribution and MaxTenuringThreshold Parameters.....	104

Chapter 13: Optimizing Broker CAA and Cluster Failover.....107

Overview.....	107
Tuning Brokers to Optimize Failover.....	108
Connection-failure Detection.....	108
Connect-failure Detection.....	109
Failover Behavior.....	109
Adjusting the Failover Properties.....	110
Tuning TCP to Optimize Failover.....	112
TCP Settings for Fast Failover on Windows® system.....	113
TCP Settings for Fast Failover on Linux® system.....	113

Preface

For details, see the following topics:

- [About this Documentation](#)
- [Typographical Conventions](#)
- [Aurea® Sonic® Documentation](#)
- [Worldwide Technical Support](#)

About this Documentation

This guide is part of the documentation set for Aurea® SonicMQ® 2015 SP1.

This guide provides tips and techniques for setting parameters and properties in SonicMQ® brokers and clients that can let you realize improved throughput with no degradation in service levels. It describes messages, blocks, and bytes showing how the SonicMQ® buffers use flow control, logs, and throttling so that you can use the Sonic Management Console and the Configuration API to tune for optimal performance. It tells you about sizing and how thresholds trigger actions. In addition, this manual describes techniques for tuning client applications to avoid unnecessary overhead and includes advice on tuning JVM settings.

Typographical Conventions

This section describes the text-formatting conventions used in this guide and a description of notes, warnings, and important messages. This guide uses the following typographical conventions:

- **Bold typeface in this font** indicates keyboard key names (such as **Tab** or **Enter**) and the names of windows, menu commands, buttons, and other Sonic® user-interface elements. For example, “From the **File** menu, choose **Open**.”
- **Bold typeface in this font** emphasizes new terms when they are introduced.
- **Monospace typeface** indicates text that might appear on a computer screen other than the names of Sonic® user-interface elements, including:
 - Code examples and code text that the user must enter
 - System output such as responses and error messages
 - Filenames, pathnames, and software component names, such as method names
- **Bold monospace typeface** emphasizes text that would otherwise appear in **monospacetypeface** to emphasize some computer input or output in context.
- **Monospace typeface in italics** or **Bold monospace typeface in italics** (depending on context) indicates variables or placeholders for values you supply or that might vary from one case to another.

This manual uses the following syntax notation conventions:

- Brackets ([]) in syntax statements indicate parameters that are optional.
- Braces ({ }) indicate that one (and only one) of the enclosed items is required. A vertical bar (|) separates the alternative selections.
- Ellipses (...) indicate that you can choose one or more of the preceding items.

This guide highlights special kinds of information by shading the information area, and indicating the type of alert in the left margin.

Note: A Note flag indicates information that complements the main text flow. Such information is especially helpful for understanding the concept or procedure being discussed.

Important: An Important flag indicates information that must be acted upon within the given context to successfully complete the procedure or task.

warning: A Warning flag indicates information that can cause loss of data or other damage if ignored.

Aurea[®] Sonic[®] Documentation

Aurea[®] Sonic[®] platform installations always have a welcome page that provides links to Aurea[®] Sonic[®] documentation, release notes, communities, and support. See the *Product Update Bulletin* for "what's new" and "what's changed" since prior releases.

The Aurea[®] Sonic[®] documentation set includes the following books and API references.

Aurea[®] SonicMQ[®] Documentation

Aurea[®] SonicMQ[®] software installations provide the following documentation:

- *Aurea[®] Sonic[®] Installation and Upgrade Guide* — The essential guide for installing, upgrading, and updating Aurea[®] SonicMQ[®] software on distributed systems, using the graphical, console or silent installers, and scripted responses. Describes on-site tasks such as defining additional components that use the resources of an installation, defining a backup broker, creating activation daemons and encrypting local files. Also describes the use of characters and provides local troubleshooting tips.
- *Aurea[®] SonicMQ[®] Getting Started Guide* — Provides an introduction to the scope and concepts of Aurea[®] SonicMQ[®] messaging. Describes the features and benefits of Aurea[®] SonicMQ[®] messaging in terms of its adherence to the JavaSoft JMS specification and its rich extensions. Provides step by step instructions for sample programs that demonstrate JMS behaviors and usage scenarios. Concludes with a glossary of terms used throughout the Aurea[®] SonicMQ[®] documentation set.
- *Aurea[®] SonicMQ[®] Configuration and Management Guide* — Describes the configuration toolset for objects in a domain. Also shows how to use the JNDI store for administered objects, how integration with Aurea[®] Actional[®] platform is implemented, and how to use JSR 160 compliant consoles. Shows how to manage and monitor deployed components including metrics and notifications.
- *Aurea[®] SonicMQ[®] Deployment Guide* — Describes how to architect components in broker clusters, the Aurea[®] Sonic[®] Continuous Availability Architecture[™] and Dynamic Routing Architecture[®]. Shows how to use the protocols and security options that make your deployment a resilient, efficient, controlled structure. Covers all the facets of HTTP Direct, a Aurea[®] Sonic[®] technique that enables Aurea[®] SonicMQ[®] brokers to send and receive pure HTTP messages.
- *Aurea[®] SonicMQ[®] Administrative Programming Guide* — Shows how to create applications that perform management, configuration, runtime and authentication functions.
- *Aurea[®] SonicMQ[®] Application Programming Guide* — Takes you through the Java sample applications to describe the design patterns they offer for your applications. Details each facet of the client functionality: connections, sessions, transactions, producers and consumers, destinations, messaging models, message types and message elements. Complete information is included on hierarchical namespaces, recoverable file channels and distributed transactions.
- *Aurea[®] SonicMQ[®] Performance Tuning Guide* — Illustrates the buffers and caches that control message flow and capacities to help you understand how combinations of parameters can improve both throughput and service levels. Shows how to tune TCP under Windows[®] and Linux[®] for the Aurea[®] Sonic[®] Continuous Availability Architecture[™].
- *Aurea[®] SonicMQ[®] API Reference* — Online JavaDoc compilation of the exposed Aurea[®] SonicMQ[®] Java messaging client APIs.

- *Management Application API Reference* — Online JavaDoc compilation of the exposed Aurea® SonicMQ® management configuration and runtime APIs.
- *Metrics and Notifications API Reference* — Online JavaDoc of the exposed Aurea® SonicMQ® management monitoring APIs.
- *Aurea® Sonic® Event Monitor User's Guide* — Packaged with the Aurea® SonicMQ® installer, this guide describes the Aurea Soniclogging framework to track, record or redirect metrics and notifications that monitor and manage applications.

Other Documentation in the Aurea® SonicMQ® Product Family

The Aurea® Sonic® download site provides access to additional client and JCA adapter products and documentation:

- *Aurea® SonicMQ® .NET Client Guide* — Packaged with the Aurea® SonicMQ® .NET client download, this guide takes you through the C# sample applications and describes the design patterns they offer for your applications. Details each facet of the client functionality: connections, sessions, transactions, producers and consumers, destinations, messaging models, message types and message elements. Includes complete information on hierarchical namespaces and distributed transactions. The package also includes online API reference for the Aurea® Sonic® .NET client libraries, and samples for C++ and VB.NET.
- *Aurea® SonicMQ® C Client Guide* — Packaged with the Aurea® SonicMQ® C/C++/COM client download, this guide presents the C sample applications and shows how to enhance the samples, focusing on connections, sessions, messages, producers and consumers in both the point-to-point and publish/subscribe messaging models. Provides tips and techniques for C programmers and gives detailed information about using XA resources for distributed transactions. The package also includes online API reference for the Aurea® SonicMQ® C client.
- *Aurea® SonicMQ® C++ Client Guide* — Packaged with the Aurea® SonicMQ® C/C++/COM client download, this guide presents the C++ sample applications and shows how to enhance the samples, focusing on connections, sessions, messages, producers and consumers in both the point-to-point and publish/subscribe messaging models. Provides tips and techniques for C++ programmers and gives detailed information about using XA resources for distributed transactions. The package also includes online API reference for the Aurea® SonicMQ® C++ client.
- *Aurea® SonicMQ® COM Client Guide* — Packaged with the Aurea® SonicMQ® C/C++/COM client download for Windows®, this guide presents the COM sample applications under ASP, and Visual C++. Shows how to enhance the samples, focusing on connections, sessions, messages, producers and consumers in both the point-to-point and publish/subscribe messaging models. Provides tips and techniques for COM programmers. The package also includes online API reference for the Aurea® SonicMQ® COM client.
- *Aurea® SonicMQ® Resource Adapter for JCA - User's Guide for WebSphere®* — Packaged with this JCA adapter in a separate download, this guide describes the Aurea® Sonic® Resource Adapter for JCA and using it with a WebSphere® application server.

- *Aurea® SonicMQ® Resource Adapter for JCA - User's Guide for Weblogic* — Packaged with this JCA adapter in a separate download, this guide describes the Aurea® Sonic® Resource Adapter for JCA and using it with a Weblogic application server.
- *Aurea® SonicMQ® Resource Adapter for JCA - User's Guide for JBoss®* — Packaged with this JCA adapter in a separate download, this guide describes the Aurea® Sonic® Resource Adapter for JCA and using it with a JBoss® application server.

Worldwide Technical Support

Aurea® Software's support staff can provide assistance from the resources on their web site at <http://www.aurea.com/sonic>. There you can access technical support for licensed Aurea® Sonic® products to help you resolve technical problems that you encounter when installing or using Aurea® Sonic® products.

When contacting Technical Support, please provide the following information:

- The release version number and serial number of Aurea® SonicMQ® that you are using. This information is listed on the license addendum. It is also at the top of the Aurea® SonicMQ® Broker console window and might appear as follows:

```
Aurea® SonicMQ® Continuous Availability Edition [Serial Number nnnnnnnn]  
Release nnn Build Number nnn Protocol nnn
```

- The release version number and serial number of Aurea® Sonic ESB® that you are using. This information is listed on the license addendum. It is also near the top of the console window for an Aurea® Sonic ESB® Container. For example:

```
Sonic ESB Continuous Availability Edition [Serial Number nnnnnnnn]  
Release nnn Build Number nnn Protocol nnn
```

- The platform on which you are running Aurea® Sonic® products, and any other relevant environment information.
- The Java Virtual Machine (JVM) your installation uses.
- Your name and, if applicable, your company name.
- E-mail address, telephone, and fax numbers for contacting you.

Introduction

Before starting a tuning process for SonicMQ it is important to be clear on the goals of the effort. Tuning is often a process of trade-offs. Changing one parameter might help performance as measured by one metric, but hurt on others. The following are concerns related to performance tuning:

- Messaging Throughput — Maximize the number of messages handled by SonicMQ in a given period.
- Application Throughput — Maximize the rate that a given application can either produce or consume messages. Certain parameters that maximize total system throughput might hinder individual applications.
- Message Latency — Minimize the amount of time that a message is held in transit. Techniques to minimize transit time of a message from producer to consumer rely on reducing the total system buffers, which can affect maximum throughput of both applications and the messaging system.
- Application Isolation — Maximize the isolation of services distributed over the messaging domain. Effectively, this means increasing buffers to prevent burst producers from overwhelming messages consumers, and to buffer message producers from the effects of temporarily slow or unavailable consuming applications.

A benchmarking environment must allow for realistic numbers of simulated client connections and message volumes. See the Aurea <http://www.aurea.com/sonic> web site for information about test harnesses and applications you can use for benchmarking.

This document focuses on broker tuning. There are also application design options that impact performance in your messaging infrastructure, such as message delivery mode, acknowledgement modes, transacted behaviors, connection factory settings and, Java Virtual Machine parameters used by the client applications.

For details, see the following topics:

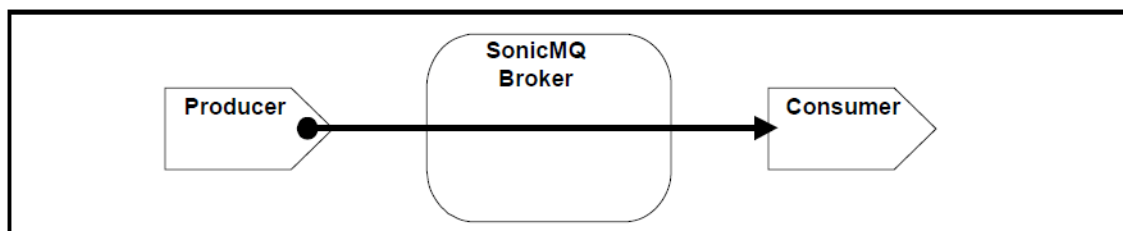
- [SonicMQ® Messaging](#)
- [Messages, Blocks, and Bytes](#)
- [Buffer Sizes and Threshold Behavior](#)
- [Message Buffers and JMS Priorities](#)

SonicMQ® Messaging

Some basic concepts about SonicMQ® messaging establishes a conceptual foundation for understanding performance tuning. SonicMQ® is a provider of the Java Messaging Service (JMS). The JMS specification is just a specification of a programming API. It deals with messaging clients, both producers and consumers, and how they connect to some **messaging provider**.

Message producers in applications send messages to either Topics or Queues, and message consumers in applications receive these messages and consume them. The typical diagram is shown in the following figure.

Figure 1: Messaging Occurs Between JMS Clients Through the Provider SonicMQ



While the above figure reflects the conceptual model of producers sending messages to consumers, it does not cover the following three important characteristics of messaging:

Distributed Architecture

Messaging connects distinct Java processes running on distributed computers. Producers, consumers, and the SonicMQ broker each run in their own Java Virtual Machine (JVM) and communicate over a network.

Asynchronous Messages

While sending or publishing a message may seem like a single action to a message producer, the hand off of that message to the provider actually goes through a series of steps that take place before the `send()` or `publish()` call returns.

In the case of the `send()` or `publish()`, the following must occur:

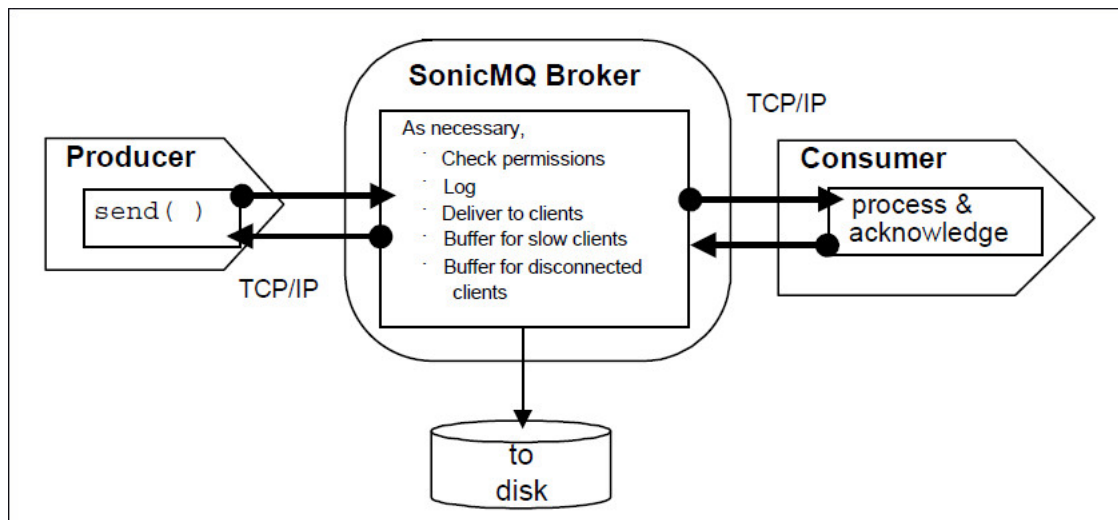
- A message must be sent over the network to the SonicMQ broker.
- The broker must receive it.
- The broker must guarantee the delivery of the message (subject to the quality of service specified). Often this involves persisting the message to disk.

- The producer must be notified that the broker has guaranteed the message.
- A similar hand off occurs at the Consumer, which receives a message, processes it, and then sends an explicit or automatic acknowledgement—again, subject to JMS quality of service).

Buffering

In addition to the internal asynchronous message/acknowledgement over the network, JMS specifically separates the actions of the producer from the actions of the consumers. SonicMQ acts as a buffer that can deal with transient burst loads from producers, and handle cases where consumers are unavailable momentarily or long intervals of time. This buffering of messages occurs either in memory or is moved to disk (depending on quality of service and available resources). The following figure shows the asynchronous communication over a network, for distributed Producers, Consumers, and brokers.

Figure 2: Asynchronous Message Producers and Consumers

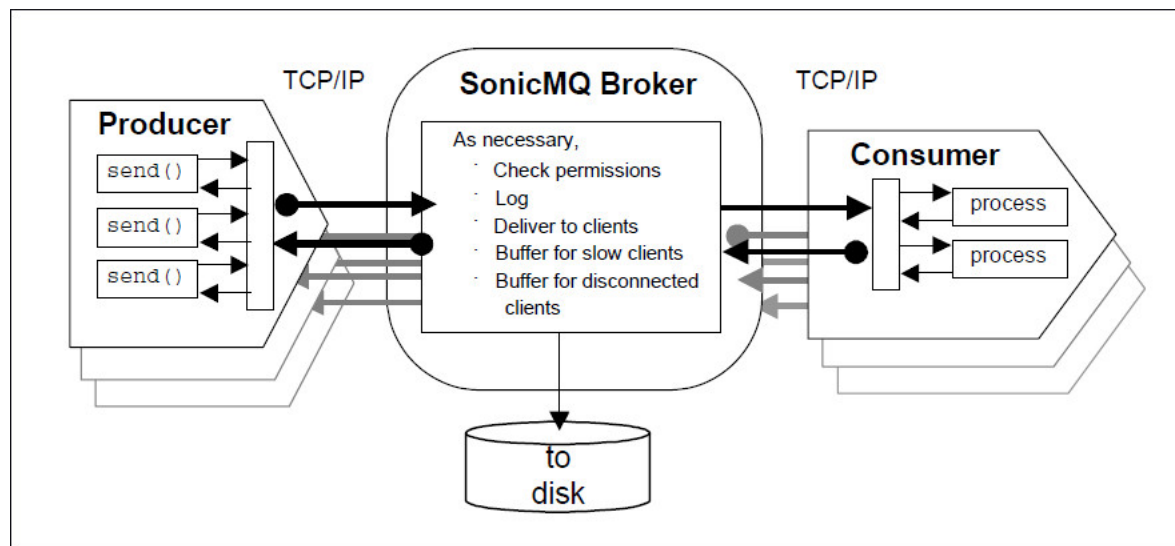


The diagram is more realistic when there are multiple producers and consumers with the potential of many-to-many relationship. Even within a single JMS client, a single network connection might have to serve multiple producing or consuming sessions.

In the following figure, you can see that there are many places where there are multiple processes dealing with messages that have to transfer the contents using limited computer resources. For example:

- **Producers** — Where many sending or publishing operations have to contend for a single network connection.
- **SonicMQ Broker** — Where messages arriving from many clients have to be delivered to many subscribers.
- **Disk** — Where **PERSISTENT** messages from many producers have to be preserved on disk.

Figure 3: Multiple Asynchronous Message Producers and Consumers



In addition to the problem of many processes contending for limited resources, there is also the issue that the underlying disk and network resources are oriented towards **bytes**, while the JMS objects and guarantees deal with **messages**.

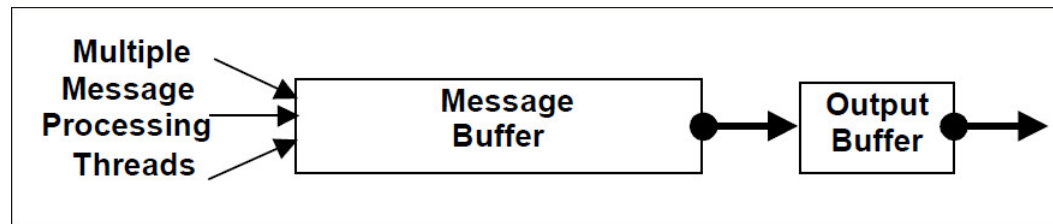
Messages, Blocks, and Bytes

SonicMQ uses a common design pattern to help deal with the joint problems of:

- Multiple requests on limited machine/network resources
- System resources that are oriented to bytes and block sizes, not messages

The following figure shows an architecture used repeatedly in SonicMQ. Many processing threads are handling messages concurrently. At some point, these threads need to use a limited system resource (such a disk storage or network sockets). Instead of waiting for the resource, SonicMQ allows you to configure a Message Buffer to hold messages until the resource becomes available.

Figure 4: Sharing and Managing Limited Resources in SonicMQ Messaging

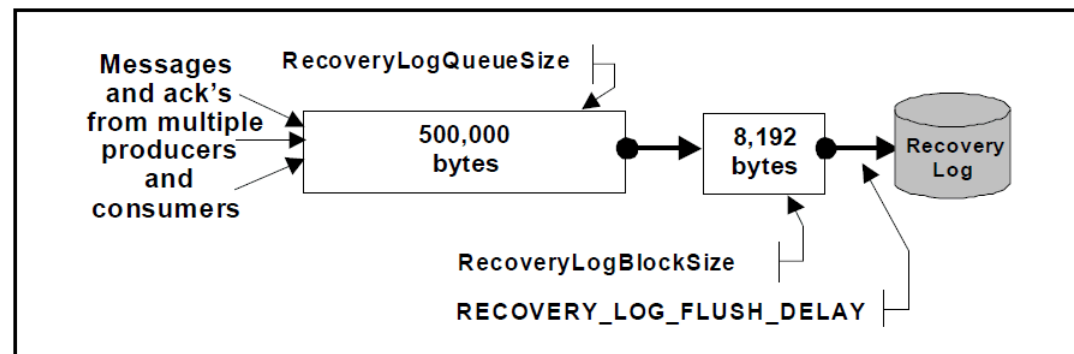


Message Buffers are typically configured to hold transient bursts of messages, and are designed to allow processing threads to perform other tasks in parallel instead of waiting for limited resources. Message buffers are typically processed sequentially, subject to JMS message priority concerns (where applicable).

Internally, SonicMQ takes messages off the Message Buffer and writes them as a stream of bytes to a system resource, such as a disk or network socket. However, for system resources, there is often a performance optimum where the number of bytes written matches the physical characteristics of the resource. For disk I/O, the optimal setting might correspond to a block size. For a network, the optimal setting might be related to packet size, or the overhead of sending a message.

A common pattern used by SonicMQ to match the messages to system resources is the addition of an Output Buffer, which is also tunable. The mechanism used to tune varies according to the resource. An example of this design pattern can be found in how SonicMQ brokers access their log.

Figure 5: Buffers When Writing to the Recovery Log

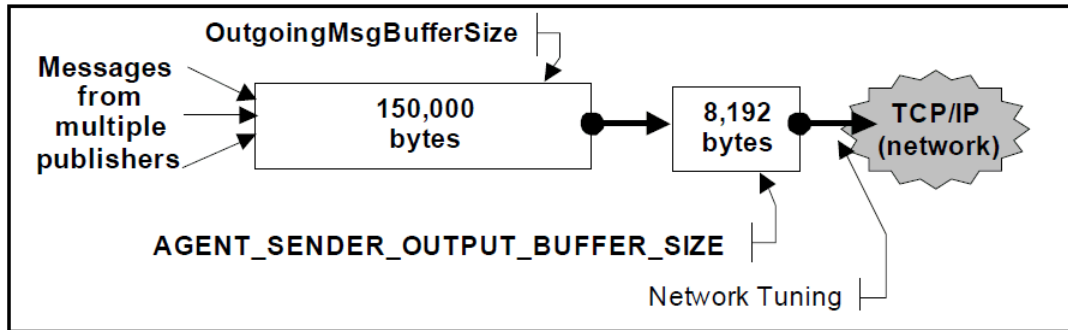


Writing to the Recovery Log has a large buffer to handle messaging bursts, and a small buffer to match disk writes to physical block size.

This example is discussed later in this manual, now it serves as an example to illustrate the point. Writing to the recovery log is a sequential process. A 500K message buffer allows for serializing operations against the log without forcing all tasks to wait. The actual writes to the disk can be tuned using parameters that are relevant to the characteristics of the disk—both its block size and speed (**RecoveryLogBlockSize** and **RECOVERY_LOG_FLUSH_DELAY**).

The following figure illustrates another use of the Message Buffer/Output Buffer design pattern. This example shows the handling of multiple publishers sending messages to a single subscriber. A large message buffer is used to cushion the system against publish rates exceeding the capabilities of the subscriber, and the network. A smaller output buffer is used to hold bytes before they go over the network. An additional tuning parameter, `TcpNoDelay`, can also be used to optimize the hand off to the network.

Figure 6: Buffers when Writing to the Log



Note: More complete details of this example are discussed in Chapter 3, Subscriptions and Durable Subscriptions on page 21.

As you read through this manual, you will see this pattern of Message Buffer/Output Buffer repeated in many areas. The points to remember about this architecture are:

- Message Buffers should be sized to handle burst loads where message production temporarily exceeds the ability of the system to process them.
- The typical size of messages used in your application affects Message Buffers much more than Output Buffers.
- Output Buffers should be configured based on the underlying I/O system and its physical characteristics.

Buffer Sizes and Threshold Behavior

There is another major difference between **Output Buffers** and **Message Buffers** used in SonicMQ® system. **Output Buffers** typically correspond to a real allocation of memory. This is not true for **Message Buffers**, where the sizing parameters represent threshold values that trigger some change in behavior of the SonicMQ® broker.

A message might be handled, simultaneously, by many components of a SonicMQ® broker. However, there is typically only one copy of the message held in memory. The message buffer typically only stores a *reference* to this message. As message references are added to the message buffer, a counter of size is incremented. Similarly, as messages are processed, the counter is decremented.

The following are the different strategies followed by SonicMQ® when a message buffer exceeds its threshold size:

- **Blocking** — A common strategy is to simply force threads to wait until space becomes available. This always works, but limits the concurrency that is possible in processing messages.
- **Throttling Producers** — SonicMQ® system's flow control is used in some cases where producers are asynchronously asked to slow message production when message buffers exceed some threshold size.
- **Make Room** — It might be possible to check existing messages referenced by the message buffer and see if room can be found. In the case where an application is using time-to-live on messages, or the SonicMQ® system DISCARDABLE delivery option, some space in the buffer could be reclaimed, reducing the size under its threshold value.

The key point about tuning Message Buffer Size parameters is to remember that it is a reference to a threshold size that triggers some change in the way SonicMQ® system processes messages.

Message Buffers and JMS Priorities

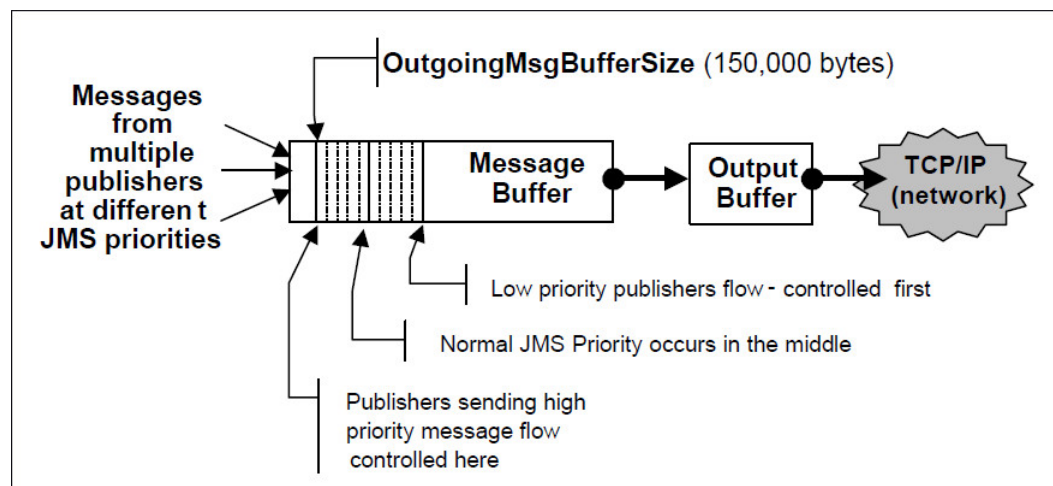
Because JMS allows messages to be delivered with different priorities, many of the Message Buffers used by SonicMQ actually use a sliding threshold to trigger internal behavior.

The threshold that triggers throttling, or blocking, of message producers is typically triggered at a lower value for low-priority messages. This allows SonicMQ to “reserve” message buffer space to handle a later arrival of a higher-priority message.

In typical JMS usage scenarios, messages are all delivered on a common topic with the same priority. In this situation, there is only one observable impact of the sliding priority threshold.

The Message Buffer Size parameter is really the threshold size for the highest priority messages (**JMSPriority=9**). For normal priority messages (**JMSPriority=4**), SonicMQ will invoke threshold behaviors slightly sooner.

Figure 7: Message Buffers



Priority-based Message Buffers support a sliding threshold for changing behavior depending on JMS Priority.

Consider messages sent to a subscribing client as one example. One key turning parameter is the size of the outgoing message buffer used for messages sent from a broker to a subscriber (**OutgoingMsgBufferSize**). When this value is at its default value of 150,000 bytes, a producer sending normal priority messages will start being throttled when the message buffer reaches about 125K.

This sliding priority threshold allows SonicMQ to deal sensibly with publishers that are producing messages at different priorities. A fast publisher using normal priority (**JMSPriority=4**) will not be able to overwhelm the Message Buffer and prevent a high-priority publisher from sending messages.

The other aspect to priority-based Message Buffers is that SonicMQ will service high-priority messages first. In this way, the most important messages will be delivered to the clients before less critical messages.

Using Tuning Parameters and Properties

This guide shows how to use the Sonic graphical tool for administrative configuration, the Sonic Management Console, to review and adjust parameters that set functionality related to performance tuning. The screen captures in this chapter illustrate the primary entry areas for setting these broker parameters.

This chapter demonstrates the relationship between the individual entry areas in the dialog boxes and the methods that can be used to programmatically tune the settings in applications.

For details, see the following topics:

- [Accessing Broker Properties with the Graphical Tool](#)
- [Method Name Hints for Administrative Programming](#)
- [Settings Listed in This Manual](#)

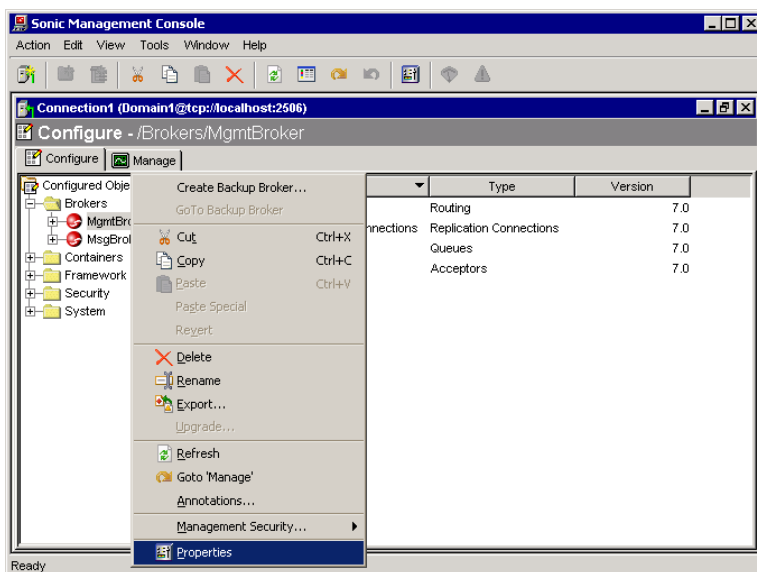
Accessing Broker Properties with the Graphical Tool

The Sonic graphical administrative tool, the Sonic Management Console, provides configuration and runtime management of all aspects of SonicMQ® containers, components, and framework settings.

This manual describes properties that are, for the most part, accessible in the **Properties** dialog box of a broker instance.

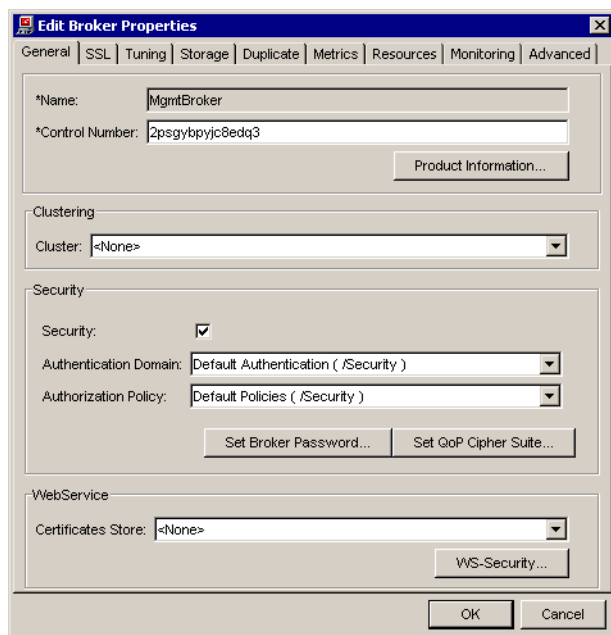
To access this dialog box, start the Sonic Management Console, connect it to the domain where the broker is configured, then open the folders on the **Configure** tab to expose the broker you want. Right click on the broker to choose its **Properties** command, as shown in following figure where the names are the default folder and broker, **/Brokers/Broker1**.

Figure 8: Accessing a Broker's Properties



The **Edit Broker Properties** dialog box opens to its **General** tab, as shown in the following figure.

Figure 9: Broker Properties General tab



The **Tuning** tab includes parameters that are discussed throughout this manual, as shown in the following figure.

Figure 10: Broker Properties Tuning tab

The screenshot shows the 'Edit Broker Properties' dialog box with the 'Tuning' tab selected. The dialog has several tabs: General, SSL, Tuning, Storage, Duplicate, Metrics, Resources, Monitoring, and Advanced. The 'Tuning' tab contains the following sections and parameters:

- Buffer Sizes:**
 - Outgoing Buffer Size: 150000 bytes
 - Guaranteed Buffer Size: 150000 bytes
 - Wait Buffer Size: 75000 bytes
 - Pub/Sub DB Buffer Size: 2097152 bytes
 - PTP DB Buffer Size: 2097152 bytes
- Recovery Log:**
 - Location: ./log
 - Size: 209715200 bytes
 - Block Size: 8192 bytes
 - Buffer Size: 5000000 bytes
 - XONCE Recovery: ☒
- Transactions:**
 - Primary Threads: 10
 - Secondary Threads: 1
 - Idle Timeout: 0 minutes
 - Buffer Size: 32768 bytes
- Flow To Disk:** ☐

At the bottom of the dialog are 'OK' and 'Cancel' buttons.

The **Advanced** tab has several parameters that are discussed in this manual, as shown in the following figure.

Figure 11: Broker Properties Advanced tab

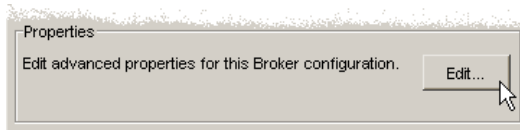
The screenshot shows the 'Edit Broker Properties' dialog box with the 'Advanced' tab selected. The dialog has the same tabs as Figure 10. The 'Advanced' tab contains the following sections and parameters:

- Notifications / Alerts:**
 - DMQ Notify Factor: 85.00 % full
 - Recovery Log Notify Factor: 50.00 % full
- Connections:**
 - TCP Nodelay: ☒
 - Domain Suffix Search Order:
 - Enable Load Balancing: ☒
 - Load Balancing Weight: 1
 - Client Reconnect Timeout: 600 seconds
 - Max Connections: ☒ Unlimited ☐
- Proxy:**
 - URL:
 - Username:
 - Password:
- Properties:**
 - Edit advanced properties for this Broker configuration.

At the bottom of the dialog are 'OK' and 'Cancel' buttons.

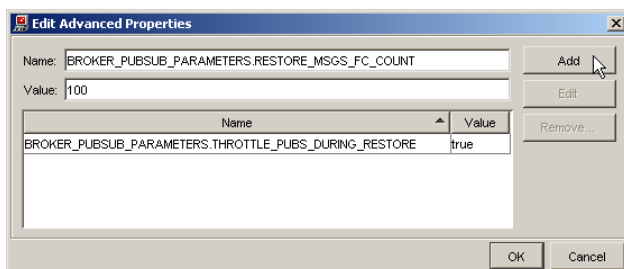
Some of the tuning parameters described in this manual are accessed as fields on this tab. Other advanced properties are entered as name-value pairs. Click the **Edit** button, as shown in the following figure, to open the edit dialog box for these properties.

Figure 12: Accessing the Advanced Properties dialog box



Use the **Edit Advanced Properties** dialog box to set designated properties, as shown in the following figure.

Figure 13: Edit Advanced Properties for a Broker



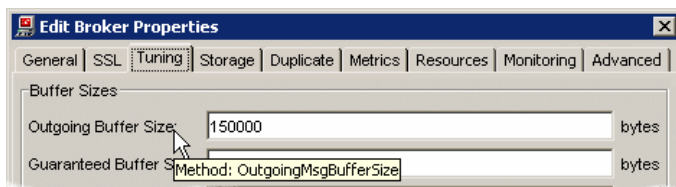
For more information on how to use the Sonic Management Console, see the *Aurea® SonicMQ® Configuration and Management Guide*.

Note: This manual shows Advanced Properties in **UPPER_CASE**. Other properties are directly accessible through the labeled fields in the Management Console and are shown with **LeadingCapitals**. For example, **RECOVER_LOG_FLUSH_DELAY** is an Advanced Property while **OutgoingMsgBufferSize** is directly accessible.

Method Name Hints for Administrative Programming

The names of the various configuration properties discussed in this manual correspond to the method hints provided in the pop-up ToolTip shown in the Sonic Management Console. When you hover over a label in the dialog box, a ToolTip showing the underlying Configuration API method appears, as shown in the following figure.

Figure 14: ToolTip Showing the Selected Broker Parameter's Property Name



The Method hint shown in the above figure is the **OutgoingMsgBufferSize** variable. You want to directly access (access) or change (mutate) the value of that variable. This notion of accessors and mutators is commonly known and implemented as **getters and setters**. As such, the actual methods associated with accessing or changing these properties are:

```
// Get the OutgoingMsgBufferSizepublic int
getOutgoingMsgBufferSize();
// Set the OutgoingMsgBufferSizepublic void
setOutgoingMsgBufferSize(int value);
```

For more information on how to use SonicMQ administrative programming and its samples, see the *Aurea® SonicMQ® Administrative Programming Guide*.

Settings Listed in This Manual

In this manual, whenever a property is described, a table lists the API classes and methods that set the property. The table lists the setter for the value, but in all cases, there is a corresponding getter.

Exposed Properties

Properties that are directly accessible through the labeled fields in the Sonic Management Console and are shown with **LeadingCapitals** and have corresponding methods, as shown in the following table. For example, the **FlowToDisk** option—a technique that lets you choose to offload Pub/Sub messages to disk until slow subscribers are ready to process more messages—is directly accessible on the **Tuning** tab:

Flow To Disk: ☐

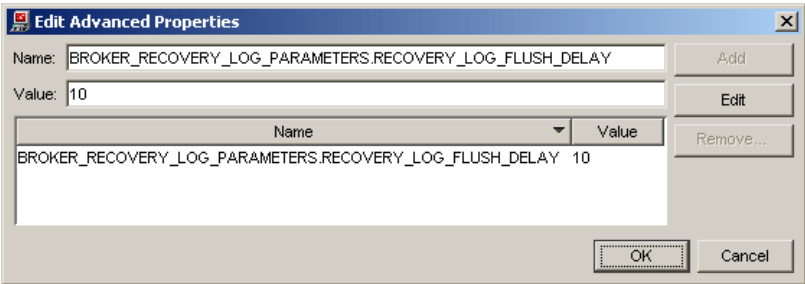
Table 1: Example of Access to a Exposed Properties

Technique	Details
Management Console	Broker Properties > Tuning Tab
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IPubSubParametersType Method: setFlowToDisk(boolean);

Advanced Properties

Advanced properties are not displayed through ToolTips as there is no corresponding input area. The names are presented in UPPER_CASE for input into the name-value entry area in the graphical tool, as shown in the following figure.

Figure 15: Example of Entering an Advanced Property in the Graphical Tool



Similarly, the property can be set programmatically in the specified package and interface using a strongly typed attribute setter (or getter) that takes the name and the value as its parameters, as shown in the example in the following table.

Table 2: Example of Access to an Advanced Property

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Advanced name: BROKER_RECOVERY_LOG_PARAMETERS.RECOVERY_LOG_FLUSH_DELAY
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IRecoveryLogType Method: setIntegerAttribute("RECOVERY_LOG_FLUSH_DELAY", int size);

Subscriptions and Durable Subscriptions

This chapter contains the following sections:

- [Overview](#) on page 29
- [Subscriptions](#) on page 30
- [Durable Subscriptions](#) on page 33
- [Flow to Disk Publishing](#) on page 38

For details, see the following topics:

- [Overview](#)
- [Subscriptions](#)
- [Durable Subscriptions](#)
- [Flow to Disk Publishing](#)

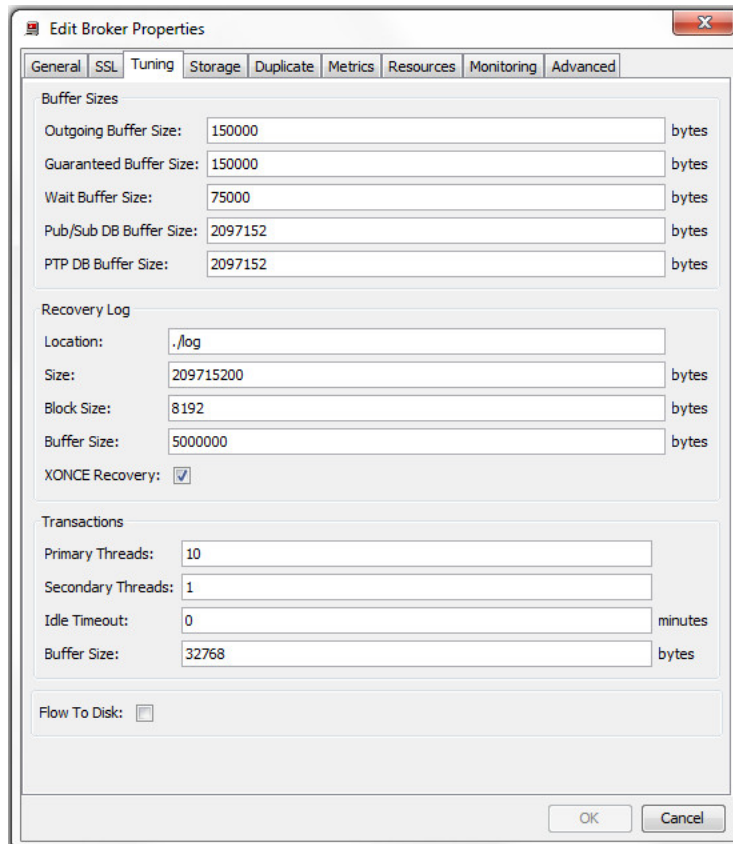
Overview

In the Publish and Subscribe messaging model, there are ways to improve performance in subscriptions and durable subscriptions. Flow-to-Disk publishing provides an additional way to enhance flow control.

Subscriptions

This section deals with the tuning of JMS Consumers and first looks at the subscribers in the Pub/Sub messaging model. The major tuning parameters related to subscriptions are found on the **Tuning** tab of the **Broker > Properties** in the Sonic Management Console.

Figure 16: Subscription-related Tuning Properties



The screenshot shows the 'Edit Broker Properties' dialog box with the 'Tuning' tab selected. The dialog contains several sections for configuring JMS broker properties:

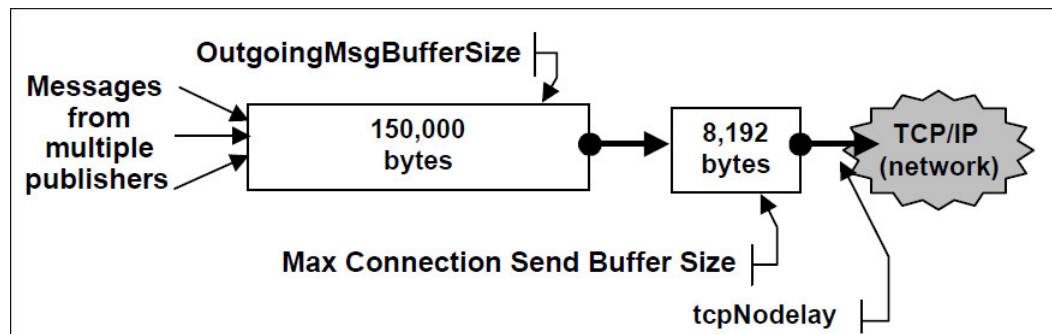
- Buffer Sizes:**
 - Outgoing Buffer Size: 150000 bytes
 - Guaranteed Buffer Size: 150000 bytes
 - Wait Buffer Size: 75000 bytes
 - Pub/Sub DB Buffer Size: 2097152 bytes
 - PTP DB Buffer Size: 2097152 bytes
- Recovery Log:**
 - Location: ./log
 - Size: 209715200 bytes
 - Block Size: 8192 bytes
 - Buffer Size: 5000000 bytes
 - XONCE Recovery: ☒
- Transactions:**
 - Primary Threads: 10
 - Secondary Threads: 1
 - Idle Timeout: 0 minutes
 - Buffer Size: 32768 bytes
- Flow To Disk:** ☐

At the bottom of the dialog are 'OK' and 'Cancel' buttons.

Many publishers could be sending to a single subscriber simultaneously. One or more of these publishers might be sending a constant stream of messages. These messages must be marshaled into a stream of bytes and delivered to the subscribing client on the network.

To manage this interaction, SonicMQ uses a combination of Message and Output Buffers. Messages arrive from many publishers in a relatively large buffer, parameterized by the **OutgoingMsgBufferSize**. They are marshaled into a buffered byte stream (using the value of **Max Connection Send Buffer Size** set on each acceptor). Finally, the information is sent to the subscriber over the network. Setting **TcpNodelay** can also affect this transfer.

Figure 17: Subscriber Properties



The Message Buffer used for a subscriber can also be affected by the use of message publishing options:

- **Time-to-Live** — Affects expiration of messages
- **DISCARDABLE** — Extension to JMS Delivery Modes that allows SonicMQ to discard messages instead of throttling publishers

When an attempt is made to add a message to a subscriber's outgoing message buffer that would cause the buffer to exceed its threshold size, SonicMQ sees if messages in the buffer have expired or are set to **DISCARDABLE** delivery mode. If any such messages are dropped and the pending message is added successfully, no publishers are flow-controlled.

If you do not use **DISCARDABLE** or expiring messages, then this check will not occur. However, if you do use either of these options, consider the following: the more messages in the outgoing message buffer, the more time intensive this check becomes. A good idea is to set a small **Max Connection Send Buffer Size** if you have small messages.

The key parameters determining the behavior of message delivery to subscribers are:

- **OutgoingMsgBufferSize** property
- **Max Connection Send Buffer Size** property on acceptors
- **TcpNodelay** property

Each of these is discussed in the following sections of this guide.

OutgoingMsgBufferSize Property

Subscribing clients have a broker-side output buffer used to hold outgoing messages that need to be sent from the broker to the subscribing client. This parameter specifies the threshold size, in bytes, that will cause subscribing clients to issue flow control notifications to publishers.

Setting this parameter to a larger value will allow more messages to be buffered for each client before publishers are throttled by flow-control, but this action is at the risk of increasing message latency.

The default value of **OutgoingMsgBufferSize** is **150000** bytes.

Table 3: Access to `OutgoingMsgBufferSize` Settings

Technique	Details
Management Console	Broker Properties
Management API	Package: <code>com.sonicmq.mgmtapi.config</code> Interface: <code>IBrokerBean.IConnectionBufferType</code> Method: <code>setOutgoingMsgBufferSize(int value);</code>

Note: For Durable Subscribers, the threshold value `PsGuarMsgBufferSize` will typically take affect before `OutgoingMsgBufferSize` has an impact.

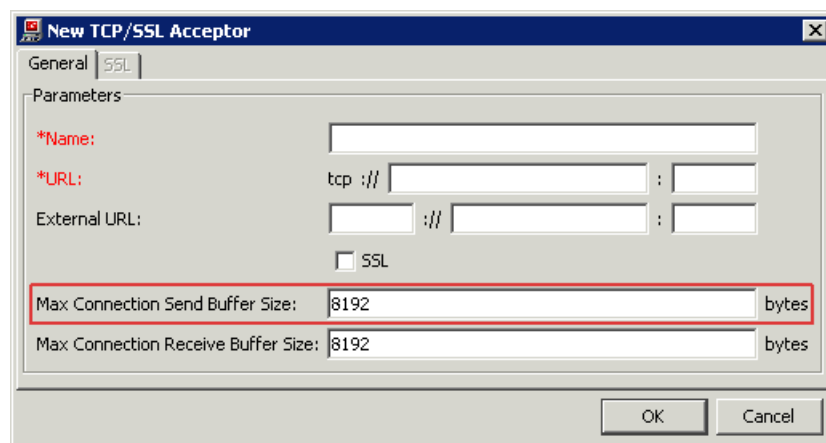
Max Connection Send Buffer Size Property

This buffer is used to pack a series of small messages into a single I/O operation.

SonicMQ will not send messages over the network until either one of the following conditions is met: (1) either this buffer is filled; or (2) the Outgoing Message Buffer contains no additional messages. Increasing this value will minimize SonicMQ's creation of small network packets. However, this increases latency when small messages are used.

The default value of **Max Connection Send Buffer Size** on each TCP and SSL acceptor on a broker is 8192 bytes.

Figure 18: Setting Max Connection Send Buffer Size on a TCP/SSL Acceptor



TcpNodelay Property

Setting the **TcpNodelay** property value to true disables the Nagle algorithm. The Nagle algorithm allows buffering of small data before sending the data as a fully constructed IP packet. You should change this default if you are concerned about overwhelming the network with many small packets. See Chapter 7, *TcpNodelay: Nagle Algorithm* on page 87 for more information. The default value of **TcpNodelay** is **true**.

Table 4: Access to TcpNodelay Setting

Technique	Details
Management Console	Broker Properties > Advanced tab
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.ITuningParametersType</code> Method: <code>setTcpNodelay(boolean value);</code>

Durable Subscriptions

Durable subscribers have two distinct modes of operation: connected and disconnected. There are also special situations when a durable subscriber reconnects. At this point, messages need to be retrieved from long-term storage and delivered.

In addition, when a durable subscriber disconnects without unsubscribing, there might be messages that were delivered to the active client but were not acknowledged. These messages need to be preserved for redelivery when the client reconnects.

Disconnected Durable Subscriptions

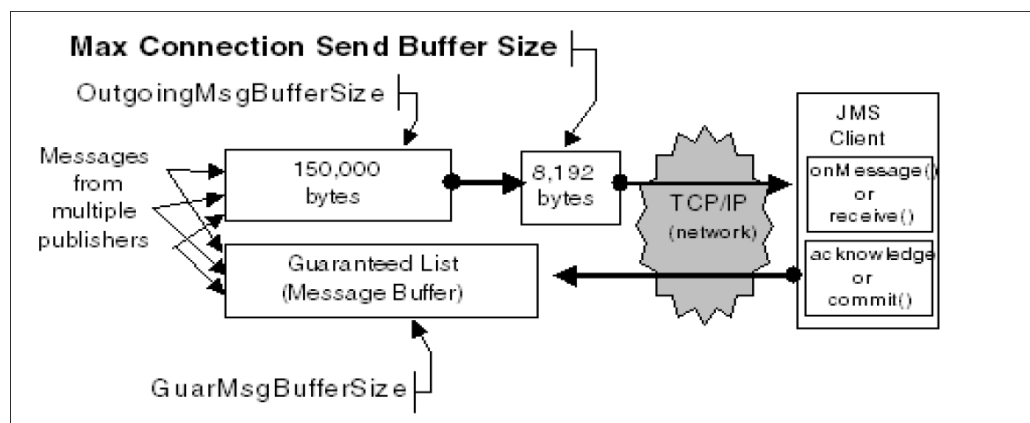
The case of disconnected durable subscriptions is one where messages are stored in long-term storage. This process is configured through the **PsDbQueueSize**.

Connected Durable Subscriptions

When a durable subscriber is connected, it has all the parameters that apply to a nondurable subscriber (**Max Connection Send Buffer Size**, **OutgoingMsgBufferSize**, and such). That is, publishing processes add references to messages to a Message Buffer, where they are held until the message is sent over the network.

Durable subscribers differ from nondurable ones because the SonicMQ broker must retain a reference to the messages delivered to the durable clients until the clients have acknowledged receipt. This acknowledgement occurs using normal JMS acknowledgement modes, or by committing transacted sessions (both local JMS transactions and **XATopicConnections**).

Figure 19: Max Connection Send Buffer Size Properties



PsGuarMsgBufferSize Property

This specifies the threshold size, in bytes, of broker-side buffers used to maintain a list of message references waiting to be acknowledged by the topic subscriber. **PsGuarMsgBufferSize** is a parameter that does not apply to nondurable subscribers.

When a publisher sends a message that causes this threshold size to be exceeded, then that publisher receives a flow control notification, and is asked to throttle its publishing rate.

In SonicMQ, the list of guaranteed messages will include all **PERSISTENT** or **NON_PERSISTENT** messages that have arrived from publishers. It will not include messages that are **DISCARDABLE**.

When all messages are **PERSISTENT** or **NON_PERSISTENT**, then the Guaranteed List will never hold fewer message references than the Outgoing Message Buffer because messages arrive at both buffers simultaneously, but always stay longer in the Guaranteed list. **PsGuarMsgBufferSize** should typically be at least as large as **OutgoingMsgBufferSize**.

PsGuarMsgBufferSize should be sized based on the messaging rate, message size, and the expected processing time of your JMS client. Particular note should be given to the use of **transacted sessions**. The size of this buffer must be at least as large as all the messages expected in a single transaction. When the threshold size is too small, then publishers will be flow controlled and the transaction will never receive all the messages that it expects.

Because flow control for PTP is triggered by a different mechanism (the size of the queue), **PsGuarMsgBufferSize** has no impact when the client is a QueueReceiver.

The default value of **PsGuarMsgBufferSize** is **150000** bytes.

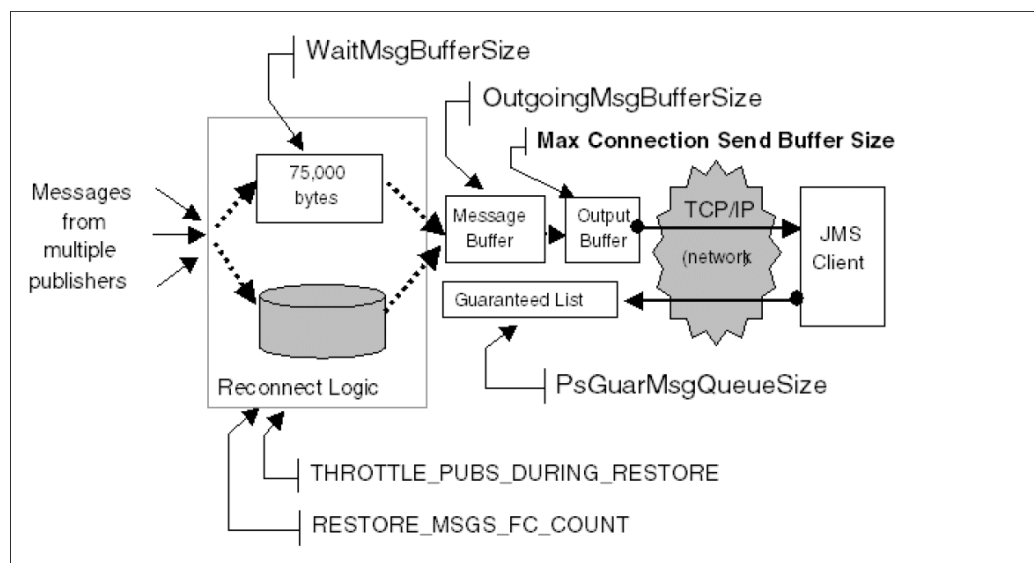
Table 5: Access to PsGuarMsgBufferSize Setting

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IConnectionBuffersType</code> Method: <code>setPsGuarMsgBufferSize(int value);</code>

Reconnecting Durable Subscriptions

A key aspect of Durable Subscriptions is the ability to preserve messages published on a topic for subscribing clients that are not currently connected. When the JMS client is connected, the subscription behaves like a normal Subscriber, and its behavior is characterized by the parameters listed above (for example, **Max Connection Send Buffer Size**, **OutgoingMsgBufferSize**, and **PsGuarMsgBufferSize**). When disconnected, the behavior is tuned using the parameters for Data Storage.

Figure 20: Reconnecting Durable Subscriptions



There is a special period where a JMS client reconnects to a durable subscription, and the behavior must transition from the disconnected, long-term storage behavior, to the normal connected behavior. Initially, there is a period when new messages from publishers continue to be stored while the oldest messages are read and delivered to the client.

Obviously, if the publish rate is too great, it can overwhelm the delivery of older messages. The following parameters are of interest for tuning performance of the initial reconnect logic:

- **THROTTLE_PUBS_DURING_RESTORE** property
- **RESTORE_MSGS_COUNT** property
- **RESTORE_MSGS_FC_COUNT** property

The wait message buffer lets you manage the buffer while restarting:

- **WaitMsgBufferSize** property

THROTTLE_PUBS_DURING_RESTORE Property

When a durable subscriber reconnects, there is a period when messages are being retrieved from the longer-term store. However, messages might still be arriving from active publishers. Setting this parameter to true allows the broker to control the flow of publishers, if they are writing new messages to the persistent storage mechanism faster than the restore is reading them.

The default value of **THROTTLE_PUBS_DURING_RESTORE** is **true**.

Table 6: Access to THROTTLE_PUBS_DURING_RESTORE Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit > Name : BROKER_PUBSUB_PARAMETERS.THROTTLE_PUBS_DURING_RESTORE
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IPubSubParametersType</code> Method: <code>setBooleanAttribute("THROTTLE_PUBS_DURING_RESTORE", boolean value);</code>

RESTORE_MSGS_COUNT Property

Sets the batch size in db queries that are used in restoring messages from the db for reconnected subscribers. You may want to set this parameter to a higher value when there are a small number of durable subscribers concurrently retrieving messages. Setting an excessively large value may cause operations that use the db to experience delays.

The default value of **RESTORE_MSGS_COUNT** is **100** messages.

Table 7: Access to RESTORE_MSGS_COUNT Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : BROKER_PUBSUB_PARAMETERS.RESTORE_MSGS_COUNT
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IPubSubParametersType</code> Method: <code>setIntegerAttribute("RESTORE_MSGS_COUNT", int size)</code>

RESTORE_MSGS_FC_COUNT Property

During reconnect, do not allow the publishers to get more than this number of messages ahead of the reconnecting subscriber.

This parameter allows for slight publishing bursts while a durable subscriber is reconnecting. As long as the subscriber never falls behind by the **RESTORE_MSGS_FC_COUNT**, then the publishers will not be throttled.

The default value of **RESTORE_MSGS_FC_COUNT** is **100** messages.

Table 8: Access to RESTORE_MSGS_FC_COUNT Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name: BROKER_PUBSUB_PARAMETERS.RESTORE_MSGS_FC_COUNT
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IPubSubParametersType Method: setIntegerAttribute("RESTORE_MSGS_FC_COUNT", int size)

At the tail end of the durable subscription reconnection process is a period when the last message is recovered from the long-term storage. To preserve the JMS message-order guarantee, the broker must deliver all these messages to the JMS client prior to delivering any new messages.

SonicMQ adds an additional message buffer to hold these new messages while the Message Buffer for the client is servicing the last stored messages.

WaitMsgBufferSize Property

This parameter specifies the size of the message buffer that holds messages while a durable subscription is being restarted. Effectively, this buffer behaves like the normal Outgoing Message Buffer and stores messages.

Typically, you should increase this parameter when large messages are in use. It is common to size this parameter to about half the size of the **OutgoingMsgBufferSize**. The default value of **WaitMsgBufferSize** is **75000** bytes.

Table 9: Access to WaitMsgBufferSize Setting

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IConnectionBuffersType Method: setWaitMsgBufferSize(int value);

ACKNOWLEDGE_MONITOR_INTERVAL Property

The advanced broker property **ACKNOWLEDGE_MONITOR_INTERVAL** enables monitoring of durable subscribers for the purpose of detecting a possible resource deadlock that may result when a durable subscriber attempts to receive messages whose total size is greater than **GUAR_QUEUE_SIZE**. The possible deadlock is detected by monitoring the each **CLIENT_ACKNOWLEDGE**, **SINGLE_MESSAGE_ACKNOWLEDGE**, or transacted durable subscriber at regular intervals.

An **AcknowledgementPause** event (containing broker name, routing node name, user name, connect ID, JMS ClientID, and durable subscription name) is generated if, over a complete interval, the following conditions have been found:

- No new messages have been published to the subscriber.
- The subscriber has not explicitly acknowledged any messages. In other words, the subscriber has not called `Message.acknowledge()` or `Session.commit()`.
- The subscriber has one or more messages pending acknowledgement.
- The subscriber is throttling one or more publishers at any priority, or a restore cycle pending for the subscriber has been blocked waiting for buffer space, or messages for this subscriber are being offloaded to disk.

The **AcknowledgementPause** event will be generated repeatedly at the configured interval until a new message is published, or the subscriber acknowledges a message, or until the consumer is closed or disconnects. The above conditions suggest—yet do not guarantee—that the subscriber is deadlocked. Note that the event might sometimes be generated for durable subscribers that are not deadlocked, when these conditions apply during the normal pattern of messaging. The monitoring interval is configured by setting the advanced property **ACKNOWLEDGE_MONITOR_INTERVAL** to the preferred number of seconds. The default value of **ACKNOWLEDGE_MONITOR_INTERVAL** is 0 seconds (monitoring disabled).

Table 10: Access to ACKNOWLEDGE_MONITOR_INTERVAL Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : <code>BROKER_CONNECTION_BUFFERS.ACKNOWLEDGE_MONITOR_INTERVAL</code>
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IConnectionBufferType</code> Method: <code>setIntegerAttribute ("ACKNOWLEDGE_MONITOR_INTERVAL", int size)</code>

Flow to Disk Publishing

A configuration option in SonicMQ lets you specify that, when pub/sub messages cannot be delivered immediately to a connected but slow subscriber, the messages should be offloaded to the persistent storage mechanism used by the SonicMQ broker—the same persistent storage mechanism used to store messages for disconnected durable subscriptions—and stored there until the subscriber is ready to process more messages. At that time, the messages are read from the persistent storage mechanism and delivered to the subscriber.

When a message is published to a topic that has multiple connected subscribers, the message is stored only for the subscribers that are not ready to process more messages. Performance to subscribers that can keep with message flow is maintained except for the overhead of writing the messages for the slow subscriber to the persistent storage mechanism. Publishers continue producing messages without being blocked by the broker.

When the subscriber is ready to process more messages, some messages from the in-memory buffer will be sent to the subscriber. When the buffer has enough space for the next message, it is read from the persistent storage mechanism and added to the buffer.

While messages are read from the persistent storage mechanism and delivered to the subscriber, the new messages continue be written to the persistent storage mechanism until the subscriber catches up. When there are no messages in the persistent storage mechanism for the subscriber, new messages are delivered live.

If the persistent storage mechanism gets filled, publishers are flow controlled. The publishers are not allowed to resume publishing to topics until the cleanup threads (see Clean Thresholds on page 59) actually free up the space in the persistent storage mechanism. After cleanup, the remaining persistent storage size must fall below the threshold before publishing is allowed to resume.

Flow to disk has an effect on which subscriber gets the next message in shared subscriptions. When the broker decides which subscriber—whether disconnected durable, connected durable, or connected nondurable—in a subscriber group gets the next message, the broker favors subscribers connected to the local system that are not flow controlled or the subscriber causing flow to disk because it is the slowest subscriber to the topic.

Non-Durable Subscribers

When a non-durable subscriber disconnects and it was the furthest behind, the persistent storage mechanism drops messages pending delivery to that subscriber up to the current message of the next slowest subscriber. If the only subscriber to the topic where flow to disk is in effect disconnects and that subscriber is non-durable, all remaining messages for connected subscribers to that topic are dropped.

Flow To Disk Settings

The FlowToDisk option lets you specify that whenever a subscriber's in-memory buffer is full, the broker should offload messages for the subscriber to the persistent storage mechanism. A publishing application will be flow controlled when the in-memory buffer for at least one subscriber becomes full. There are distinct performance advantages to enabling Flow to Disk functionality so enabling this setting should always be considered in performance tuning.

The parameters that are tuned on the broker and cluster in support of the Flow To Disk Publishing feature are:

- The unclustered broker property [FlowToDisk Option \(Broker\)](#) on page 40
- The broker cluster property [FlowToDisk Option \(Cluster\)](#) on page 41
- The advanced broker setting [MAX_FTD_MEMORY_SIZE Property](#) on page 41

See also [Overriding the Broker's FLOW_TO_DISK Setting in Applications](#) on page 42.

Note: Cluster setting overrides broker setting — When a broker is a member of a cluster, the cluster-wide setting applies and the broker-level setting is ignored. When a broker is a member of a cluster, setting/unsetting Flow To Disk at the broker level has no effect—Flow To Disk must be set at the cluster level.

FlowToDisk Option (Broker)

The default value of **FlowToDisk** on a broker is **false**.

Figure 21: Flow To Disk Option (Broker)

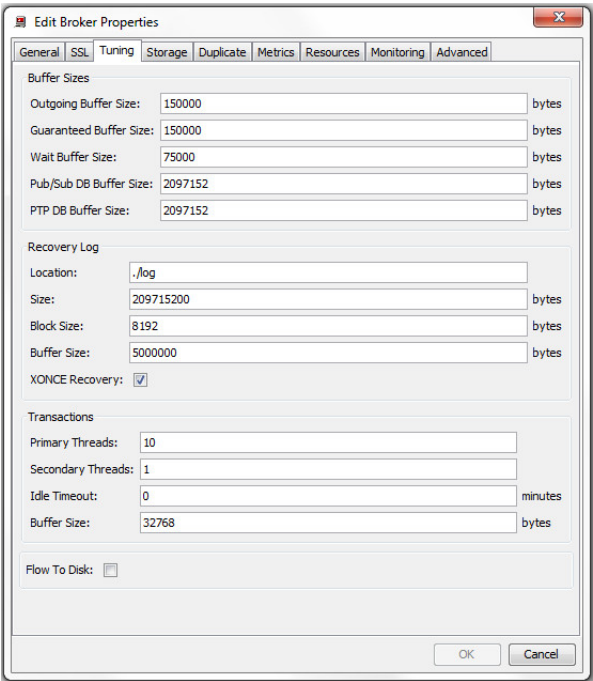


Table 11: Access to FlowToDisk Option (Cluster)

Technique	Details
Management Console	Broker Properties > General tab
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IPubSubParametersType Method: setFlowToDisk(boolean value);

FlowToDisk Option (Cluster)

The FlowToDisk setting on the cluster forces all broker members of the cluster to use a common FlowToDisk setting, and to override the setting on each individual broker member. Changing this parameter immediately applies the change to all brokers in the cluster. The default value of **FlowToDisk** on a cluster is **false**.

Figure 22: Flow To Disk Option (Cluster)

The screenshot shows a 'New Cluster' dialog box with the 'General' tab selected. The fields are as follows:

- *Name:** New Cluster
- *Routing Node Name:** New Cluster
- Enable Dynamic Host Binding:** ☐
- Security:**
 - Authentication Domain:** <None>
 - Authorization Policy:** <None>
- Flow Control Notifications:**
 - Client Default Monitor Interval:** 60 seconds
 - Cluster/Routing Monitor Interval:** 15 seconds
- Flow To Disk:** ☐

Buttons at the bottom: OK, Cancel.

Table 12: Access to FlowToDisk Option (Cluster)

Technique	Details
Management Console	Cluster Properties > General tab
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IClusterBean.IPubSubParametersType Method: setFlowToDisk (boolean value);

MAX_FTD_MEMORY_SIZE Property

To avoid having memory overflows from buffers for FlowToDisk messages, the total memory allocation on the broker for that purpose can be constrained to the value you set for **MAX_FTD_MEMORY_SIZE**. When that value is exceeded, flow-control goes into effect.

The default value of **MAX_FTD_MEMORY_SIZE** is 32 megabytes.

Table 13: Access to MAX_FTD_MEMORY_SIZE Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : BROKER_PUBSUB_PARAMETERS.MAX_FTD_MEMORY_SIZE
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IPubSubParametersType Method: setIntegerAttribute("MAX_FTD_MEMORY_SIZE", int size)

Overriding the Broker's FLOW_TO_DISK Setting in Applications

A JMS application can override the **FlowToDisk** setting on a broker or cluster by using:

- The `setFlowToDisk()` method in `progress.message.jclient.ConnectionFactory`
- The `setFlowToDisk()` method in `progress.message.jclient.Session`,
- Setting it on a JMS Administered **ConnectionFactory** object setting through the Sonic Management Console's JMS Administered Objects tool's **Messaging** tab.

The options for the Flow To Disk setting are the integer value in the methods or the pull down option in the Administered Object GUI:

- **0 (USE_BROKER_SETTING)** — The default setting, specifies that the broker setting of **FLOW_TO_DISK** will be used for the subscriber.
- **1 (ON)** — Sets **FLOW_TO_DISK** on for the subscriber despite the broker setting.
- **2 (OFF)** — Sets **FLOW_TO_DISK** off for the subscriber despite the broker setting.

The default behavior from previous releases is maintained by the default broker setting of **false**: a publishing application will be flow controlled when the in-memory buffer for at least one subscriber becomes full.

Queues

This chapter contains the following sections:

- Overview on page 54
- QueueMaxSize Property on page 45
- QueueSaveThreshold Property on page 45
- QueueDeliveryThreads Property on page 47
- System Queues Used in Dynamic Routing on page 48
- Temporary Queues on page 50
- Expiring Messages in Queues on page 52

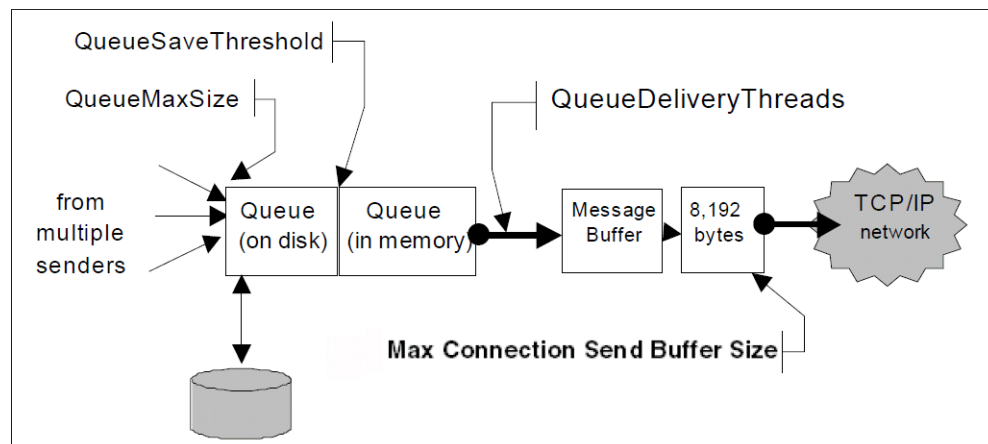
For details, see the following topics:

- [Overview](#)
- [QueueMaxSize Property](#)
- [QueueSaveThreshold Property](#)
- [QueueDeliveryThreads Property](#)
- [System Queues Used in Dynamic Routing](#)
- [Temporary Queues](#)
- [Expiring Messages in Queues](#)

Overview

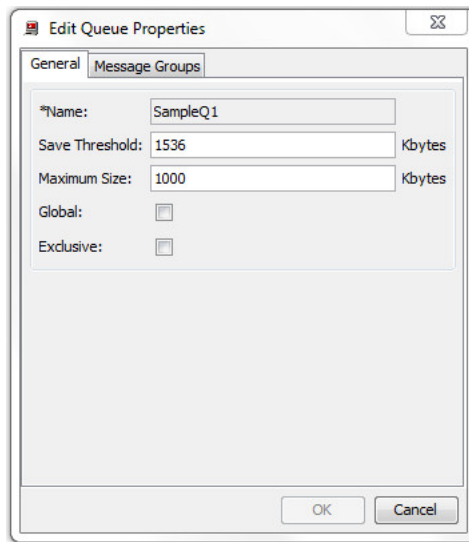
Performance and capacity in Point-to-Point messaging is fundamentally defined by the size of queues, and what portion of queues are held in memory (in contrast to the portion of queues saved in long-term storage).

Figure 23: Queue Settings



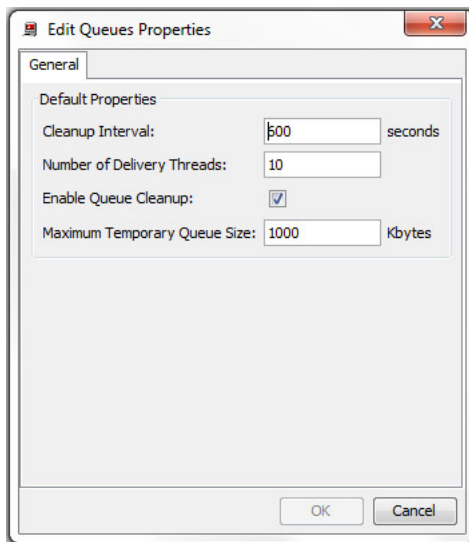
The properties of a queue can be set in a Management Console dialog box accessed by right clicking on a queue name under a broker in the **Configure** view.

Figure 24: Setting Queue Properties on a Queue Instance



The properties for all queues on a broker can also be set in the Management Console by right clicking on the **Queues** folder under a broker in the **Configure** view.

Figure 25: Setting Queues Properties at a Broker's Queues Branch



The parameters that determine what fraction of the queue is kept in memory, and when long-term storage is used are relative settings discussed in this chapter.

QueueMaxSize Property

The total size of messages stored for a queue is the **QueueMaxSize**. When a Queue Sender tries to deliver a message to a queue that is at its maximum size, the sender will be flow controlled and the send of the message will be blocked until space is available.

Table 14: Access to QueueMaxSize Settings

Technique	Details
Management Console	Expand broker components, then right-click on a queue instance and choose Properties .
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IQueuesBean.IQueueAttributes</code> Method: <code>setQueueMaxSize(int value);</code>

In the figure shown in **Overview** topics, the **QueueMaxSize** is shown at its default value, **1000** KB (1 megabyte). The value is a non-negative integer. You cannot make the queue size unlimited.

QueueSaveThreshold Property

The **QueueSaveThreshold** parameter lets you set the point in the queue where new messages are saved to the long-term store.

By setting the save threshold larger than the maximum size, you can ensure that queue messages are never saved to the persistent storage mechanism. This technique avoids the overhead of persistent storage mechanism operations and is appropriate for fast-moving queues.

The default **QueueSaveThreshold** value is **1536** kilobytes.

Table 15: Access to QueueSaveThreshold Settings

Technique	Details
Management Console	Expand broker components, then right click on a queue instance and choose Properties .
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IQueuesBean.IQueueAttributes</code> Method: <code>setQueueSaveThreshold(int value);</code>

All **PERSISTENT** messages will be stored in the recovery log, regardless of the size of the queue. However, once the **QueueSaveThreshold** value is exceeded, all messages (even **NON_PERSISTENT**) will be copied to disk for later retrieval.

Note: In the event of a broker shutdown, only the **PERSISTENT** messages will be restored when the broker restarts. **NON_PERSISTENT** messages from the previous session will not be available after the shutdown/restart.

Eventually, messages saved in a Queue will be delivered to JMS clients. When a **QueueReceiver** is available to receive messages, it requests them from the SonicMQ broker. The broker then tries to satisfy all the outstanding requests for messages pending on a particular queue. This process can be tuned using the next setting, **QueueDeliveryThreads**.

QueueDeliveryThreads Property

QueueDeliveryThreads specifies the number of dispatch threads that are created (and started) for dequeuing messages from queues. Increasing the number of dispatch threads can improve performance when using queues. However, as the thread count grows, internal resource contention will limit any further improvement.

The default **QueueDeliveryThreads** value is **10**.

Table 16: Access to QueueDeliveryThreads Settings

Technique	Details
Management Console	Expand broker components, then right click on Queues and choose Properties .
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IQueuesBean.IQueueParametersType</code> Method: <code>setQueueDeliveryThreads(int value);</code>

The **QueueDeliveryThreads** handles delivery of messages to particular JMS clients. The process by which messages are delivered to the client is (internally) similar to the process described earlier for JMS Subscribers. That is, messages are placed in an Outgoing Message Buffer. From there, the messages are sent over the network to the client (passing through the Output Buffer.) The delivery through the Output Buffer is tuned using the parameters:

- **AGENT_SENDER_OUTPUT_BUFFER_SIZE** (See Chapter 3, Subscriptions and Durable Subscriptions on page 21 for more information.), and
- **TcpNodelay** (See Chapter 7, TcpNodelay: Nagle Algorithm on page 87 for more information.)

System Queues Used in Dynamic Routing

For routing between nodes, the SonicMQ broker has a special system queue, **SonicMQ.routingQueue**, that is used to hold messages that are to be routed to remote nodes. It is the combined size of all messages destined for all remote nodes that triggers flow-control for JMS clients sending messages to remote nodes.

The maximum size of **SonicMQ.routingQueue** can be set in the Sonic Management Console in the same way that the size is set on other queues.

The **QueueMaxSize** for the Routing Queue should be sized to handle the combined needs of all routing connections. However, because each remote node is serviced by a separate routing connection, SonicMQ persists messages to long-term storage only for remote nodes where the routing connection is unusable. The routing connection to a particular node might be unusable because the node is disconnected, or because the connection is flow controlled.

Consider the concept that routing has small, internal queues that hold messages for individual remote nodes. Each one of these pending queues is serviced by a single routing connection. The point where in-memory queuing switches over to long-term storage is not handled by the settings for the **SonicMQ.routingQueue**. Instead, each pending queue uses its own **Save Threshold**, based on the state of the connection—whether it is disconnected or flow controlled. The following parameters can be used to tune this process.

DISCONNECTED_PENDING_SAVE_THRESHOLD Property

The **QueueSaveThreshold** used for all internal routing queues where the destination node is not connected. This might be because the brokers in the destination node are not running, or simply because of a network failure.

The situations that cause a disconnected routing connection are typically those that last for a long time. As such, it makes sense to save most of the messages to long-term storage.

This value sets the cumulative size of messages pending for a routing destination that should be stored in memory. The value is per node: the number of disconnected routing connections that can occur will affect its impact on total memory usage.

The **DISCONNECTED_PENDING_SAVE_THRESHOLD** default value is **1000** kilobytes—one megabyte.

Table 17: Access to DISCONNECTED_PENDING_SAVE_THRESHOLD Settings

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : <code>BROKER_ROUTING_PARAMETERS.DISCONNECTED_PENDING_SAVE_THRESHOLD</code>
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IRoutingParametersType</code> Method: <code>setIntegerAttribute("DISCONNECTED_PENDING_SAVE_THRESHOLD", int size);</code>

FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD Property

The **QueueSaveThreshold** used for all internal routing queues where the destination node is connected, but flow-controlled. When the connection to the broker fails, then the save threshold **FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD** is used.

The situations that cause a flow-controlled routing connection are typically those that will be resolved in a short period of time. As such, it makes sense to save most of the messages in memory. A large value is normal in this case.

The **QueueSaveThreshold** sets the cumulative size of messages pending for a routing destination that should be stored in memory. The value is set on each node.

The default **FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD** value is **15000** kilobytes—fifteen megabytes.

Table 18: Access to FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD Settings

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : BROKER_ROUTING_PARAMETERS.FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IRoutingParametersType Method: setIntegerAttribute("FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD", int size);

Temporary Queues

Temporary Queues are created by JMS clients on an ad hoc basis, and only last for the duration of the **QueueConnection** that created them. These queues are never persisted in long-term storage, and as a result do not have a need for a **QueueSaveThreshold**. Their maximum size is set using the **MaxTemporaryQueueSize** parameter.

MaxTemporaryQueueSize Property

The **MaxTemporaryQueueSize** sets the maximum size of a temporary queue set up by a JMS client application. This size should be set to a value big enough to hold all the messages expected in a temporary queue at one time.

The **MaxTemporaryQueueSize** represents a maximum size. When fewer messages are in the temporary queue, only the actual space used is allocated.

The contents of a temporary queue are lost when the JMS client either explicitly deletes the temporary queue, or when the JMS connection that created it is closed.

The default value is **1000** kilobytes—one megabyte.

Table 19: Access to MAX_TEMPORARY_QUEUE_SIZE Settings

Technique	Details
Management Console	Expand broker components, then right-click on Queues and choose Properties .
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IQueuesBean.IQueueParametersType Method: setMaxTemporaryQueueSize(int value);

Expiring Messages in Queues

There is a queue cleanup thread that can be started to check queues for expired messages. Because this checking takes time, reducing the frequency of the checks (or eliminating them altogether) can help improve your SonicMQ performance. However, you should be aware that this improvement might come at the cost of increased throttling due to flow control if the broker's buffers reach their capacity limit and/or the queues reach their maximum sizes.

You can set the following parameters to adjust either the frequency of the queue cleanups, or disable it entirely. Even if you eliminate the queue cleanup check, expired messages will still be discarded when they reach the front of the queue, or when a queue is browsed. However, if the queue is not being processed (perhaps, because there are no active queue receivers) then expired messages can take space that would otherwise accommodate new messages.

EnableQueueCleanup Property

The **EnableQueueCleanup** parameter enables queue cleanup on local and global queues. Setting this parameter to **false** disables dynamic queue cleanup.

The default value is **true**.

Table 20: Access to EnableQueueCleanup Settings

Technique	Details
Management Console	Expand broker components, then right-click on Queues and choose Properties .
Management API	Package: <code>com.sonicsw.mq.mgmtapi.configInterface:</code> <code>IQueuesBean.IQueueParametersTypeMethod:</code> <code>setEnableQueueCleanup(boolean value);</code>

QueueCleanupInterval Property

The **QueueCleanupInterval** parameter determines the amount of time between cleanups (in seconds) on local and global queues. The setting is only meaningful when **EnableQueueCleanup** is set to **true**, whether explicitly set to **true** or allowed to default.

The default value of **QueueCleanupInterval** is **600** seconds—10 minutes.

Table 21: Access to QueueCleanupInterval Settings

Technique	Details
Management Console	Expand broker components, then right-click on Queues and choose Properties .
Management API	Package: <code>com.sonicsw.mq.mgmtapi.configInterface:</code> <code>IQueuesBean.IQueueParametersTypeMethod:</code> <code>setQueueCleanupInterval(int value);</code>

Note: The queue properties, **EnableQueueCleanup** and **QueueCleanupInterval**, do not apply to the Dead Message Queue (which never deletes expired messages).

Storage

This chapter contains the following sections:

- Overview on page 64
- Guaranteed Message Stores on page 56
- Storage Tuning Properties on page 57
- Recovery Log on page 61
- Data Storage on page 68
- Using Fast Disks on page 72
- Multiple Storage Devices on page 74
- Avoid Disk Write Caches on page 76

For details, see the following topics:

- [Overview](#)
- [Guaranteed Message Stores](#)
- [Storage Tuning Properties](#)
- [Recovery Log](#)
- [Data Storage](#)
- [Using Fast Disks](#)
- [Multiple Storage Devices](#)
- [Avoid Disk Write Caches](#)

Overview

JMS requires that messages (at high quality-of-service) be delivered even in the event of a provider failure. While messages can be held in memory for active messaging, a copy of these guaranteed messages must be persisted to some storage medium.

There is often a misconception that guaranteed messages are those that are sent with the **PERSISTENT** delivery mode. While the two concepts are related, one does not always imply the other.

Guaranteed Message Stores

In Pub/Sub, both **NON_PERSISTENT** and **PERSISTENT** messages are guaranteed if they have durable subscribers. This means they will be saved to the long-term storage when the subscriber is disconnected. The message will also be saved to the recovery log, but only if it is both **PERSISTENT** and has at least one durable subscriber.

In Point-to-Point, with Queues, **PERSISTENT** messages are always saved in the recovery log. Queues also have an option to save memory where messages in full queues can be persisted to a long-term storage.

From a performance and tuning perspective, it is important to note the following about the recovery log and long-term storage:

- **Recovery Log** — Every guaranteed message is potentially written to this log. Writing is sequential. Messages are not read from the recovery log unless there is a failure of the broker.
- **Long-term Storage** — Messages are stored and indexed as records in a persistent storage mechanism. Not all messages are written to the persistent storage mechanism, and there are cases where it will never be used, even if **PERSISTENT** messages are sent. Messages written to long-term storage will be read when consuming clients become available.

Storage Tuning Properties

You can choose to use either the persistent storage mechanism embedded in SonicMQ or specify and use an external store.

MaxTopicDbSize Property

You can set the property **MaxTopicDbSize** to control the volume of published messages the broker offloads to the persistent storage mechanism for disconnected durable subscribers and connected non-durable subscribers that have messages stored because flow to disk has been applied. Because publishers are flow-controlled when this value is reached, you should monitor metrics on **TopicDbSize** to observe the actual traffic flow. There is a balance point between ensuring that publishers get blocked only for occasional delays and ensuring that all subscribers are not bearing the burden of a very slow subscriber.

Figure 26: MaxTopicDbSize Property

Maximum Topic DB Size: Mbytes

The default value for the **MAX_TOPIC_DB_SIZE** configuration parameter is **0** megabytes (no limit on data store space used) which assures compatibility with prior SonicMQ releases.

Table 22: Access to MaxTopicDbSize Setting

Technique	Details
Management Console	Broker Properties > Storage tab
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IPubSubParametersType Method: setMaxTopicDbSize(integer value); //megabytes

TOPIC_DB_SIZE_RESTART_THRESHOLD Property

When you set the **MaxTopicDbSize** to positive integer value, you establish a set size for the topics persistent storage mechanism. When that assigned value is reached, the publishing applications (all publishers on the broker) are flow-controlled when the subscriber's in-memory buffer is full or when flow to disk has offloaded messages into the persistent storage mechanism. The publishers will be unblocked when the size of the persistent storage mechanism space falls below the value of the **MaxTopicDbSize** setting minus the value of the **TOPIC_DB_SIZE_RESTART_THRESHOLD** parameter (default value of 1 megabyte).

Table 23: Access to TOPIC_DB_SIZE_RESTART_THRESHOLD Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : BROKER_PUBSUB_PARAMETERS.TOPIC_DB_SIZE_RESTART_THRESHOLD
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IPubSubParametersType Method: setIntegerAttribute("TOPIC_DB_SIZE_RESTART_THRESHOLD",int size)

The value that you set for the **TOPIC_DB_SIZE_RESTART_THRESHOLD** can be tuned by considering the **MaxTopicDbSize** you set and the average size of messages flowing through the data store. For example, if you have large messages and set **MaxTopicDbSize** to **50000**—50 GB—the restart threshold should be probably be considerably higher than 1 MB.

However, if **maxTopicDbSize** has been configured, **TOPIC_DB_SIZE_RESTART_THRESHOLD** must be less than the specified **maxTopicDbSize**. If it is not, an appropriate value for **TOPIC_DB_SIZE_RESTART_THRESHOLD** is determined and used instead. The value of 1 (MB) for **TOPIC_DB_SIZE_RESTART_THRESHOLD** should be adequate in most cases.

Clean Thresholds

The persistent storage mechanism space for Pub/Sub messages is triggered to clean up the space after set number of messages since the last cleanup have been acknowledged or set number of messages have expired.

DB_MSG_CLEAN_THRESHOLD

DB_MSG_CLEAN_THRESHOLD={5000|messages}

In Pub/Sub domains, when a message is acknowledged by a durable subscriber, the broker records the acknowledgement to prevent redelivery of the message but does not immediately delete the message from the store. Instead, the broker keeps a total count of acknowledged messages and when it reaches the limit set by the **DB_MSG_CLEAN_THRESHOLD** parameter, the broker starts a cleanup thread to purge acknowledged and expired messages. The SonicMQ broker starts a cleanup thread whenever the broker restarts. The default setting is 5000 messages.

Table 24: Access to DB_MSG_CLEAN_THRESHOLD Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : BROKER_DATABASE_PARAMETERS.DB_MSG_CLEAN_THRESHOLD
Management API	Package: com.sonicsw.mq.mgmtapi.config Interface: IBrokerBean.IDatabaseParametersType Method: setIntegerAttribute("DB_MSG_CLEAN_THRESHOLD", int size)

EXP_MSG_CLEAN_THRESHOLD

```
EXP_MSG_CLEAN_THRESHOLD={5000|messages}
```

In Pub/Sub domains, the SonicMQ broker keeps track of the total number of topic messages stored in the data store that have an expiration value set. When the count reaches the limit set by the **EXP_MSG_CLEAN_THRESHOLD** parameter, the broker starts a cleanup thread to purge acknowledged and expired messages. The default setting is 5,000 messages.

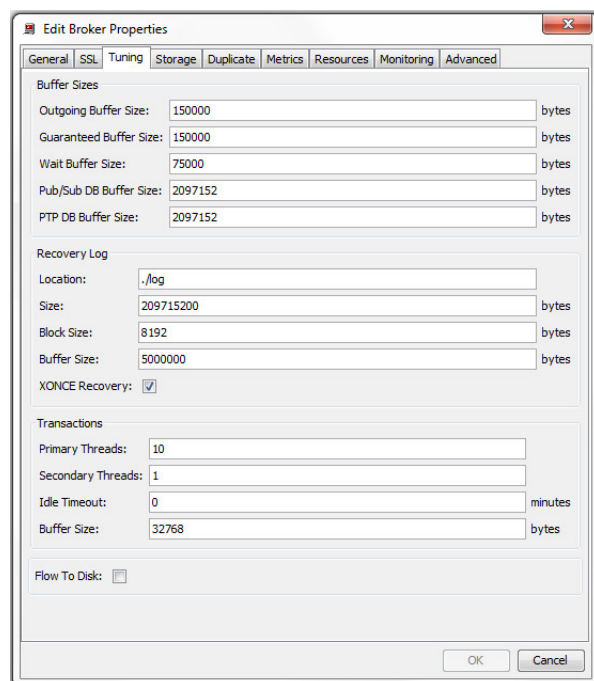
Table 25: Access to EXP_MSG_CLEAN_THRESHOLD Setting

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name : <code>BROKER_DATABASE_PARAMETERS.EXP_MSG_CLEAN_THRESHOLD</code>
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IDatabaseParametersType</code> Method: <code>setIntegerAttribute("EXP_MSG_CLEAN_THRESHOLD", int size)</code>

Recovery Log

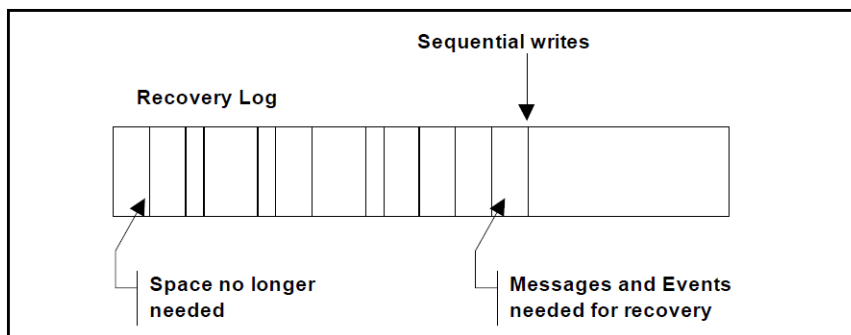
The recovery log is used to record events and messages that must be preserved in the event of a broker failure. Not every message or event needs to be logged.

Figure 27: Tuning Properties for the Recovery Log and Data Store Buffers



Events and messages are written sequentially to the recovery log. As new messages arrive, the log grows. As time passes, previously recorded events and messages become obsolete and are no longer needed.

Figure 28: Sequential Writes in the Recovery Log



To prevent the log from growing indefinitely, SonicMQ uses two log files that are accessed sequentially. When one file grows close to its maximum size, data still needed for recovery is transferred to the second file.

The following two parameters define the recovery log. These apply to all messaging models.

RecoveryLogPath Property

The **RecoveryLogPath** is the location of the log directory and the broker recovery logs. The location pathname can be relative (for example, `./log`).

Table 26: Access to RecoveryLogPath Settings

Technique	Details
Management Console	Broker Properties
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IRecoveryLogType</code> Method: <code>setRecoveryLogPath(String value);</code>

RecoveryLogMaxFileSize Property

Specifies the maximum size in bytes of each of the two broker recovery logs. The default value is **104857600**—100 MB— for the files that are created when the logs are created. The evaluation edition conserves space by setting this value to 25 MB.

Two log files are created. Each file can grow to the maximum size, so you must have double this space available. When the maximum log file size is too small for the messaging load, the broker will shut down. At this point, you can increase the maximum size and restart the broker. A large **RecoveryLogMaxFileSize** will reduce the frequency of having to switch log files. However, in the event of a crashed broker, a large recovery log will likely increase the recovery time.

Reducing the **RecoveryLogMaxFileSize** does not truncate existing files. You must reinitialize the SonicMQ data store to reduce the size of recovery logs.

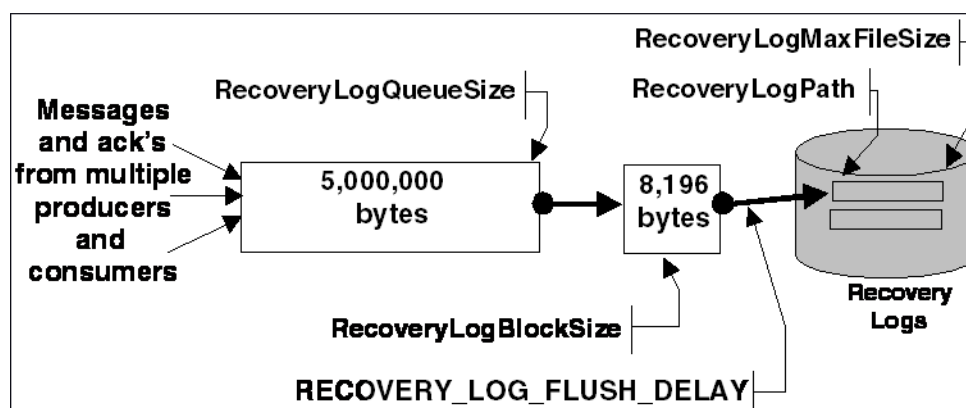
The space used for active messaging in the log depends on messaging load—both the size of messages and the time it takes for all consumers to acknowledge guaranteed messages.

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: <code>com.sonicmq.mgmtapi.config</code> Interface: <code>IBrokerBean.IRecoveryLogType</code> Method: <code>setRecoveryLogMaxFileSize(long value);</code>

Note: Significantly increasing `QueueSaveThreshold` in Point-to-Point, or **PsGuarBufferSize** for Pub/Sub might increase the number of guaranteed messages as well as the time that they need to be retained in the log prior to full acknowledgement.

Many aspects of messaging are recorded in the recovery log. To prevent operations that write to the log from becoming a bottleneck, SonicMQ uses an internal Message Buffer.

Figure 29: Recovery Log Settings



RecoveryLogQueueSize Property

Sets the message buffer used to absorb bursts in message delivery rate. The buffer is used to hold messages and internal events that need to be logged until they can be flushed to disk. While internal events do not usually vary from one application to another, message size will vary.

The parameter value is the maximum size in bytes of the broker-side buffer of log events waiting to be written to the broker recovery log. After the total size of waiting log events exceeds this limit, threads needing to add events will block until the events currently on the queue have been written to disk.

This setting provides a buffer that allows the broker to simultaneously write messages to the log, and to start delivery to recipients. When this value is smaller than your message size, then it limits the amount of parallel processing that the broker can achieve servicing multiple producing clients.

From a client perspective, the send or publish of messages that need logging (such as **PERSISTENT** to Queues or Durable Subscriptions) will not complete until the actual flush to disk occurs. However, other activities done in parallel will stop if this buffer is too small.

Because some internal management events need to be logged if this buffer is exceeded, then internal threads that are adding log events to the buffer must wait for space to become available. This would prevent some management actions (such as allowing new connections) to occur.

The default value of the **RecoveryLogQueueSize** is **5000000** bytes—5 megabytes.

Table 27: Access to RecoveryLogQueueSize Settings

Technique	Details
Management Console	Broker Properties
Management API	Package: com.sonicsw.mq.mgmtapi.configInterface: IBrokerBean.IRecoveryLogTypeMethod: setRecoveryLogQueueSize(int size);

Tuning for Large Sized Messages and Batched Transactions

When using batched transactions or large message sizes, consider tuning the broker's recovery log buffer size to accommodate the expected persistent message load generated by all of the message producers in the system. The log buffer should be tuned to hold all of the persistent messages that can be simultaneously received at the broker. With transaction batching enabled, this is the sum of each producer's batch size. Optimal performance will be achieved if the recovery log buffer can hold all of the batches. Similarly, for non batched producers using persistent delivery modes, optimal performance will be achieved if the recovery log buffer can hold all of the messages if the broker were to receive them at the same time.

There is no need to reduce this value when small- or medium-sized messages are typical.

RecoveryLogBlockSize Property

Optimizes the writing of guaranteed messages and log events to the recovery log file from memory. Disk controllers and drivers have their own limits when they write data to disk (and these limits do not necessarily match the block size of the file system).

There is not much difference in performance between writing 1KB of log events and 8KB if the controller/driver typically writes chunks of data in blocks up to 8KB at a time. For most systems, a value of **8192** (8KB) works quite well.

If, however, a controller/driver on a system writes in chunks up to 4KB, for example, resetting this property to **4096** (4KB) should improve performance. The optimal value for this property depends on the system and device.

The performance of logging can degrade if this property is changed and the logs are not re-created. Re-creating the storage for a broker creates a new log. This is done as a runtime management operation on the broker from within the Sonic Management Console (Select the broker, and choose the **Action > Initialize Storage > Recreate** command.).

A good time to change this property is when other circumstances require you to re-create the broker's storage.

The default value of **RecoveryLogBlockSize** in **8192** bytes.

Table 28: Access to RecoveryLogBlockSize Settings

Technique	Details
Management Console	Broker Properties > Storage tab
Management API	Package: com.sonicsw.mq.mgmtapi.configInterface: IBrokerBean.IRecoveryLogTypeMethod: setRecoveryLogBlockSize(int size);

RECOVERY_LOG_FLUSH_DELAY

It is inefficient to write to the log more often than necessary. In particular, when you have a disk that cannot support a high sync rate, it is better to wait until the **RecoveryLogBlockSize** is full, before trying to write and sync to disk.

This parameter allows SonicMQ to delay writing to the log in order to wait for more messages to arrive from other publishers.

You should consider updating this parameter when you have both a large number of publishers and a typical message size smaller than the **RecoveryLogBlockSize**.

The default value of **RECOVERY_LOG_FLUSH_DELAY** is 5 milliseconds.

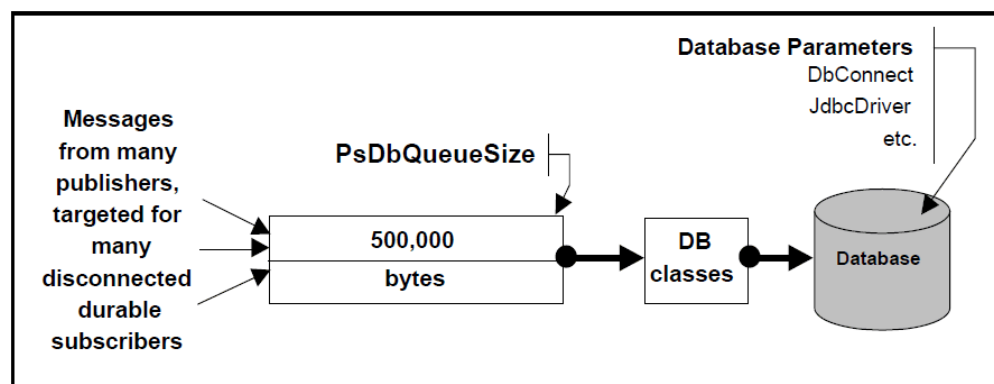
Table 29: Access to RecoveryLogFlushDelay Settings

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Advanced name: <code>BROKER_RECOVERY_LOG_PARAMETERS.RECOVERY_LOG_FLUSH_DELAY</code>
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IRecoveryLogType</code> Method: <code>setIntegerAttribute ("RECOVERY_LOG_FLUSH_DELAY", int size);</code>

Data Storage

In Pub/Sub, messages are stored in a persistent storage mechanism for disconnected durable subscribers and connected non-durable subscribers that have messages stored because flow to disk has been applied. Because many message publishers might all want to access the persistent storage mechanism concurrently, there is a potential for a bottleneck. SonicMQ provides a message buffer to mitigate the impact of this bottleneck.

Figure 30: PsDbQueueSize Property



A message buffer (parameterized with **PsDbQueueSize**) is used to hold messages until they can be saved to the persistent storage mechanism. Messages are pulled off this buffer, converted to indexed records, and stored in a persistent storage mechanism.

PsDbQueueSize Property

Sets the buffer sizes for holding Pub/Sub messages waiting to be written to the data store for disconnected durable subscribers and connected non-durable subscribers that have messages stored because flow to disk has been applied.

Two buffers are internally created, each half this size. While one is being written to the persistent storage mechanism, messages accumulate on the other. When the buffer fills, then threads that are writing to disconnected durables must wait.

From a client perspective, publishing a message that is going to a disconnected durable might tie up the connection being used by that publisher. Other publishers (in that connection) will have to wait.

To avoid blocking when using large messages, you will want to set the value to a minimum of 2-4 times the message size. (Even multiples are recommended because only half the buffer is available at any one time). There is no need to reduce this when small or medium sized messages are used.

The default value of **PsDbQueueSize** in **500000** bytes.

Table 30: Access to PsDbQueueSize Settings

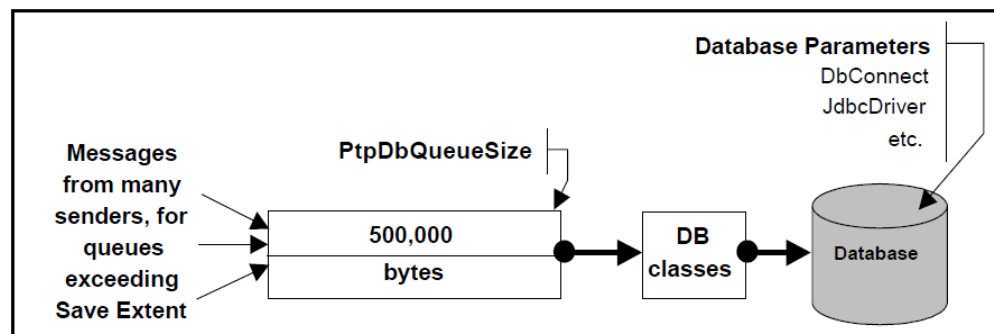
Technique	Details
Management Console	Broker Properties
Management API	Package: <code>com.sonicsw.mq.mgmtapi.configInterface:</code> <code>IBrokerBean.IPubSubParametersTypeMethod:</code> <code>setPsDbQueueSize(int size);</code>

A similar bottleneck can occur in Point-to-Point messaging with Queues. When a queue has exceeded its **QueueSaveThreshold**, then the persistent storage mechanism is used to store messages. (**QueueSaveThreshold** represents the amount of the queue held in memory, or persisted in the recovery log). When you have a number of queues in the situation where they are using long-term data storage, then there is a potential bottleneck when adding messages to the queue.

PtpDbQueueSize Property

Sets the buffer sizes for holding PTP messages waiting to be written to the persistent storage mechanism for Queues whose **QueueSaveThreshold** has been exceeded.

Figure 31: PtpDbQueueSize Property



Two buffers are internally created, each half this size. While one is being written to the persistent store, messages accumulate on the other. When the buffer fills, then threads adding messages to the queue must wait.

From a client perspective, sending a message that is going to a queue not kept in memory might tie up the connection being used by that sender.

Practically, the only time queues utilize the persistent storage mechanism is when there is a shortage of Queue Receivers for an extended period of time. As such, this parameter does not play a significant role in normal messaging.

To avoid blocking when using large messages, you will want to set the value to a minimum of 2-4 times the message size. (Even multiples are recommended because only half the buffer is available at any one time).

The default value of **PtpDbQueueSize** is **500000** bytes.

Table 31: Access to PtpDbQueueSize Settings

Technique	Details
Management Console	Broker Properties
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IQueueParametersType</code> Method: <code>setPtpDbQueueSize(int size);</code>

Using Fast Disks

Within SonicMQ, there are two places that serve as guaranteed message stores:

- Recovery Log
- Long-term Storage

Of the two, the one that typically has the greatest effect on performance of **PERSISTENT** messaging is the speed of the disk used for the Recovery Log. (set with **RecoveryLogPath**).

Unless you are using very large messages, the measure of speed is really how often the IO subsystem can sync data to disk (as opposed to the byte transfer rate).

The persistent storage mechanism is used in normal messaging only for long-term storage. The following are cases when minimal data is read/written to the persistent storage mechanism:

- No disconnected durable subscriptions.
- No messages flowed to disk to handle slow connected subscribers.
- Queues that never exceed their **QueueSaveThreshold**. The speed of the data store disk is particularly important when many operations have to be performed quickly. This possibly occurs when durable subscribers reconnect to a subscription. When there are a large number of messages waiting in the disconnected subscription, these must all be retrieved before new, live messages can be delivered to the subscriber.

The term “fast disk” is somewhat ambiguous. There are, in reality, two measures of disk speed commonly used: raw byte-transfer rate, and disk sync rate.

A disk (particularly those used in laptops) often gets its speed by buffering I/O operations. While this does help the raw byte-transfer rate, it cannot be used by systems that need to guarantee persistence, such as a persistent storage mechanism or SonicMQ. Because this is a common concern for most transactional systems, you can find a good explanation of the issues in the Java documentation. The Javadoc for `java.io.FileDescriptor` has the following description of the `sync()` method.

public void sync() throws SyncFailedException

Force all system buffers to synchronize with the underlying device. This method returns after all modified data and attributes of this **FileDescriptor** have been written to the relevant devices.

In particular, if this **FileDescriptor** refers to a physical storage medium, such as a file in a file system, `sync` will not return until all in-memory modified copies of buffers associated with this **FileDescriptor** have been written to the physical medium. **sync** is meant to be used by code that requires physical storage (such as a file) to be in a known state. For example, a class that provided a simple transaction facility might use `sync` to ensure that all changes to a file caused by a given transaction were recorded on a storage medium. `sync` only affects buffers downstream of this **FileDescriptor**.

The behavior of `sync()` has an important implication to messaging—especially in small scale applications, or simple performance tests. Due to the guarantees in the JMS specification, any application that has a single publisher sending guaranteed messages to a single consumer can never have a message rate that exceeds the sync rate of the disk.

You can create a simple `sync` test, such as:

```
import java.io.*;
public class SyncTest
{
    public static void main (String[] pArgs) throws IOException
    {
        RandomAccessFile f =
            new RandomAccessFile (pArgs [0], "rwd");
        long start = System.currentTimeMillis();
        byte [] bytes = new byte [8192];
        int iters = 10000;
        for (int i = 0; i < iters; i++)
        {
            f.write (bytes);
        }
        f.close ();
        long elapsed = System.currentTimeMillis () - start;
        System.out.println ("elapsed = " + elapsed);
        double opssec =(double) iters / ((double) elapsed / 1000.0);
        System.out.println (" " + opssec + "syncs/second");
    }
}
```

Your tests might show results similar to the following.

- Laptops: <100 syncs/second
- Desktops: 100-500 syncs/second
- Servers (RAID drives): >1,000 syncs/second

Multiple Storage Devices

There is some advantage to having the recovery log (**RecoveryLogPath**) be the only item writing to on the fastest disk, especially if your application does not significantly use long-term storage. This is because the log is accessed sequentially. You can minimize seek times for the disk head if you do not share the disk. (This also means not sharing the disk with the install, **.jar** files, or JDK.) The parameters that can be used to change the location of the Recovery Log and default long-term storage are as follows.

RecoveryLogPath Property

Sets the directory path for the two recovery log files.

Table 32: Access to RecoveryLogPath Settings

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: <code>com.sonicsw.mq.mgmtapi.configInterface:</code> <code>IBrokerBean.IRecoveryLogTypeMethod:</code> <code>setRecoveryLogPath(String value);</code>

DbConnect Property

Sets the JDBC connection parameters for long-term storage to a JDBC database. When using the default embedded persistent storage mechanism, this parameter sets the location, including disk, used for storage.

Table 33: Access to DbConnect Settings

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: <code>com.sonicsw.mq.mgmtapi.configInterface:</code> <code>IBrokerBean.IDatabaseParametersTypeMethod:</code> <code>setDbConnect(String value);</code>

After you change these values, you will need to recreate the broker's storage. To do this, choose the **Manage** tab in the Sonic Management Console, select the broker, and choose **Action > Initialize Storage > Recreate**. The log or storage will be re-created.

Note: Both the data store and the recovery log must be accessible and consistent for recovery to occur in the event of hardware failure. Having the two on separate physical disks does mean that you have doubled the chance that a hardware disk failure could occur that would prevent a broker from restarting.

Avoid Disk Write Caches

Disk file access from the broker can have a major influence on overall performance. Increased drive speeds directly translate to higher message throughput when your system processes guaranteed messages. Many disk drive controllers support write caches that allow disk writes to be delayed, increasing write speeds for the operating system. However, while a write cache increases performance, it also increases the possibility that messages will be lost when a broker machine fails. For this reason, you should not use disks configured to use a write cache.

warning: Reliability cannot be assured when write caches are active during system outages. Mapped drives typically cache disk writes. **DO NOT** use disks configured to use a write cache. If possible, turn off the cache option in the BIOS or system settings.

Transactions

When transacted sessions are used, the normal behavior of the broker changes. Messages that arrive at the broker are received normally, but all the steps associated with delivery are ignored. Instead, the messages are held in a buffer. It is only when the transacted session is committed that the messages in this buffer are processed.

SonicMQ is compliant with the JMS 1.1 specification, so transacted sessions can produce and consume on both messaging models—Point-to-point queues and Publish/Subscribe topics—in a single transaction.

This chapter contains the following sections:

- [Transaction Threads and Buffer Size](#) on page 68
- [Message Batching \(TxnBatchSize Properties\)](#) on page 71

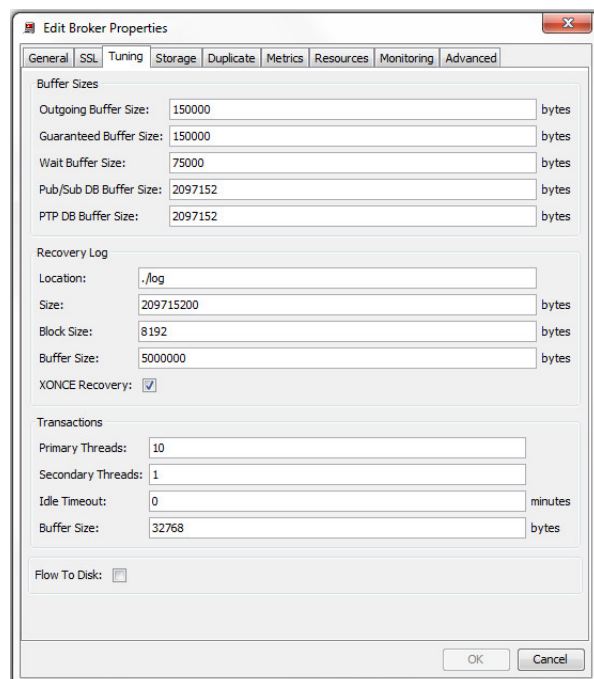
For details, see the following topics:

- [Transaction Threads and Buffer Size](#)
- [Message Batching \(TxnBatchSize Properties\)](#)

Transaction Threads and Buffer Size

The major tuning parameters related to transactions are found on the **Tuning** tab of the **Broker Properties** dialog box in the Sonic Management Console.

Figure 32: Setting Broker Properties That Are Transaction Related



When a transaction is small enough, its entire contents are kept in memory. However, at some point, this would cause memory on the broker to grow unacceptably. You can specify a **TxnBufferSize** that will be used to trigger flushing of the transaction to disk. When a transaction exceeds its **TxnBufferSize**, it is referred to as a file-based transaction.

Transactions become file-based automatically when the size of the transaction increases to a size greater than the **TxnBufferSize**. There is no upper-bound, except available disk space, on transaction size.

Figure 33: Transaction-related Properties

Every transacted JMS session potentially has an associated transaction buffer. It is only used when messages have been sent (or received) within the session. Messages are delivered from this buffer by the **TxnPrimaryThreads** when the transaction commits (or gets rolled back). The `commit()` method will not return successfully until the messages have been—as appropriate—logged, delivered, or stored.

TxnBufferSize Property

The **TxnBufferSize** parameter sets the maximum memory-based buffer size allowed per transaction before the transaction is committed. This should be tuned according to the application. In general, it is preferable to retain the bulk of the transaction in memory up to the point where the commit time overhead of flushing the transaction becomes costly. This should be tuned in conjunction with **TxnPrimaryThreads**.

When there are a small number of **TxnPrimaryThreads**, setting this buffer size to be larger alleviates the operation burden on the transaction threads.

The **TxnBufferSize** is an integer value of a number of bytes and defaults to **32768**.

Table 34: Access to TxnBufferSize Settings

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: com.sonicsw.mq.mgmtapi.configInterface: IBrokerBean.ITxnParametersTypeMethod: setTxnBufferSize(int value);

TXN_FLUSH_THREADS Property

Specifies the number of threads dedicated to flushing transactional messages to disk. The optimal value depends on the disk subsystem in use.

For example, a device such as a single IDE drive only performs a single I/O operation at one time. When two threads try to write to the disk concurrently, one waits for the other to complete—effectively the same as having a single thread. However, a sophisticated dual-channel SCSI/RAID controller using several disks enables the underlying hardware to support multiple concurrent I/O operations. In this situation a **TXN_FLUSH_THREADS** value greater than one is viable and even higher optimal values can be set.

The **TXN_FLUSH_THREADS** are only used for file-based transactions, (that is, those transactions that have exceeded their **TxnBufferSize**).

The default value of **TXN_FLUSH_THREADS** is 1.

Table 35: Access to TXN_FLUSH_THREADS Settings

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name: BROKER_TXN_PARAMETERS.TXN_FLUSH_THREADS
Management API	Package: com.sonicsw.mq.mgmtapi.configInterface: IBrokerBean.ITxnParametersTypeMethod: setIntegerAttribute("TXN_FLUSH_THREADS", int size);

Because there are usually many simultaneously active transactions in a messaging application, transactions are being constantly committed or rolled back. A pool of threads is used to manage requests for actions such as committing, rolling back, or administrative control. You can configure this pool of worker threads using the following parameters.

TxnPrimaryThreads Property

Specifies the number of threads dedicated to handling local transaction commit and rollback requests. Common XA requests, including `commit()`, are also handled by the threads in this pool. (Less common XA requests, such as **XAResource** queries, are handled by the **TxnSecondaryThreads**.)

If you have many clients all of which are dealing with small transactions, you might want to increase this parameter.

The default value of **TxnPrimaryThreads** is **10**.

Table 36: Access to TxnPrimaryThreads Settings

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: <code>com.sonicsw.mq.mgmtapi.configInterface:</code> Interface: <code>IBrokerBean.ITxnParametersType</code> Method: <code>setTxnPrimaryThreads(int value);</code>

TxnSecondaryThreads Property

Specifies the number of threads dedicated to resource-limited operations. These threads are also used for some XA actions, as well as for local transactions that use the duplicate detection feature. These threads deal, typically, with single threaded resources, so having multiple secondary transaction threads rarely has a positive impact on performance.

The default value of **TxnSecondaryThreads** is **1**.

Table 37: Access to TxnSecondaryThreads Settings

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config;</code> Interface: <code>IBrokerBean.ITxnParametersType</code> Method: <code>setTxnSecondaryThreads(int value);</code>

Message Batching (TxnBatchSize Properties)

The SonicMQ producers can batch messages in every transacted session. During a transaction, messages need not be submitted immediately to the SonicMQ broker. Instead, they may be batched on the client for a later submission when the transaction is committed or whenever the batch size or the session's ability to maintain the integrity of the transaction is at risk.

Messages are batched for each destination within a session. When several destinations are used in a single transaction, messages are batched for each destination separately.

Transaction batching provides the following performance advantages:

- Less thread context switching. When messages are sent through the broker in a batch, many messages can be delivered in a single context switch.
- Less protocol overhead

Transaction batching uses allocated memory to store the aggregate length of the payloads of messages in a transacted session. The default value is 51200 bytes. Setting the value to 0 disables transaction batching.

Table 38: Access to TxnBatchSize Settings

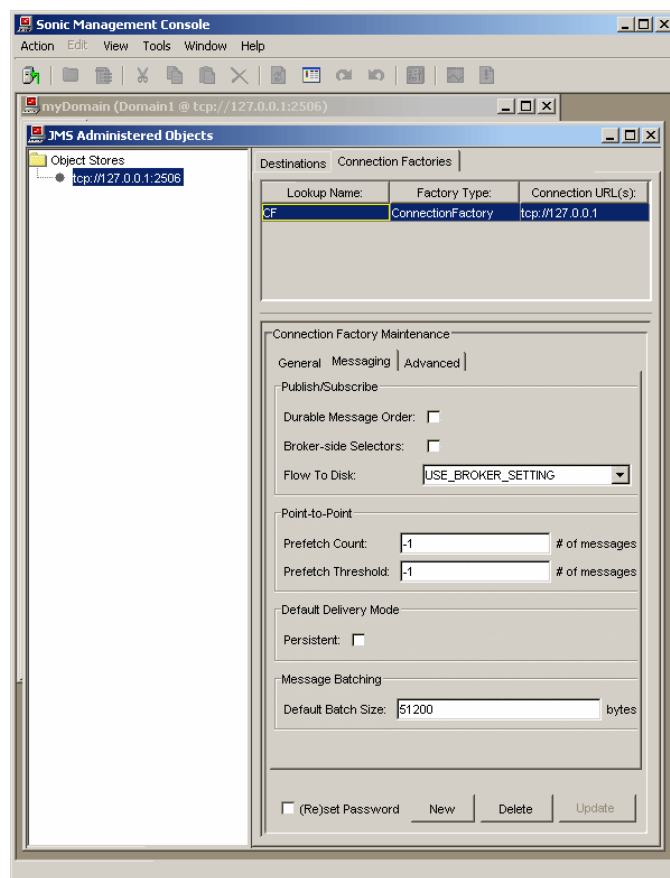
Technique	Details
Management Console	Administered Objects > ConnectionFactories tab
SonicMQ API	Package: <code>progress.message.jclient</code> Interface: <code>Session</code> Method: <code>set TxnBatchSize(int value);</code>
	Package: <code>progress.message.jclient</code> Interface: <code>Session</code> Method: <code>set TxnBatchSize(int value);</code>

Setting Message Batching in the ConnectionFactory

The **ConnectionFactory** setting lets you restate the default size. Its effect is to set the default value for transacted sessions. Setting the value to 0 shuts off the feature.

DefaultTxnBatchSize is a settable parameter in JMS administered **ConnectionFactory**s in the **Message Batching: Default Batch Size** setting, as shown the following figure.

Figure 34: Setting the Message Batch Size for a Connection Factory Object



The default size of a transaction batch is **51200** bytes. You can set the batch size to a value that better represents your common batch size. For example, if a transaction set typically involves six 20 kilobyte messages, you might set the default value to **125000**. If you set the value quite high—say, a few megabytes—the client application must have adequate runtime memory to accommodate the batch.

Setting Message Batching in the Session

When messages are sent or received on a transacted session, they have the following logic in the client application:

- A transacted session is created and message producers/consumers and destinations are created in the session. Its batch size is either explicitly set in the session or determined by the default value set in the **ConnectionFactory**.
- Messages produced are placed in a local batch if the available space in the batch is adequate.
- Messages to be consumed are received as individual messages of zero length. This lets consumer-side flow control to treat the batch as a single batch, rather than as a set of many smaller messages.
- If `commit()` is called, send the batch to the broker; if `rollback()` is called clear the batch.
- When the next message is produced, determine if it can fit in the batch. If it can, add it to the batch; if it cannot, send the batch as transacted with a call for transaction management.

- Set up a fresh batch for that producer in the session and store the current message.
- If an acknowledgment occurs, batch the acknowledgment.
- Produce batches until the client either commits or rollsback. Under commit, the current batch is sent and the transaction manager on the broker perpetuates the commit to the staged batches in the same order that they were passed to the broker. Under rollback, all the messages in the current batch and any batches on the broker are dropped.

If you do choose to reset the transaction batch size in a session, you might throw exceptions when you:

- Attempt to `setTxnBatchSize()` on a non-transacted session
- Attempt to `setTxnBatchSize()` on a closed session

You can change the setting dynamically in an application as long as multiple threads are not attempting to access the session simultaneously. Whenever the `TxnBatchSize` is changed, any current batch is sent to the broker.

TcpNodelay: Nagle Algorithm

The Nagle algorithm allows buffering of small data before sending the data as a fully constructed IP packet. The Nagle algorithm addresses the network congestion issue of sending many small JMS messages by sending fewer large packets of data, each containing many small messages.

In SonicMQ, the Nagle algorithm is, by default, disabled. This feature is customizable, allowing users to enable and disable the algorithm as required by their deployment needs.

A host receiving a stream of TCP data segments can increase efficiency by sending fewer than one ACK (acknowledgment) segment per data segment received; this is known as a delayed ACK. TCP will support a delayed ACK. The delay must be less than 0.5 seconds, and in a stream of full-sized segments there should be an ACK for at least every second segment.

When small messages do not meet the network MTU size, they are held (buffered) until the timer expires and the messages are sent. In a deployment scenario involving small messages, disabling the Nagle algorithm increases performance.

Overall performance improves when the Nagle algorithm is disabled because there are no delays due to the buffering that occurs when the algorithm is enabled. As a result, the latency of the messages is reduced.

This chapter contains the following sections:

- [Trade-offs and Risks](#) on page 76
- [Enabling and Disabling the Nagle Algorithm](#) on page 76

For details, see the following topics:

- [Trade-offs and Risks](#)
- [Enabling and Disabling the Nagle Algorithm](#)

Trade-offs and Risks

When you disable the Nagle algorithm, there is a risk of network congestion collapse. When the Nagle algorithm is disabled and an application writes a stream of data one byte at a time, each byte goes out as a separate packet. In this case there is a risk of clogging the network with tiny packets. As soon as one of the packets is dropped, retransmission can slow messaging down to a crawl with most of the traffic being retransmissions. When many applications are doing this, a momentary overload can occur. As a result, some packets can get dropped and some TCPs can start retransmitting. The retransmit traffic increases the load, and soon all users are sending mostly retransmits. In this case, connections will start breaking, the load will go down, and the net will then return to normal.

However, an application rarely requires sending messages consisting of considerably small sizes. For a relatively fast network there is no impact of congestion, if any, that arises due to disabling the Nagle algorithm.

The problem of network congestion diminishes as the number of tiny messages sent during a 24-hour cycle decreases. In a particularly overloaded network deployment, enabling the Nagle algorithm might benefit processing of the workload (as disabling it might cause network congestion).

However, for a tranquil network deployment that involves small messages, disabling the Nagle algorithm increases the message throughput, and hence performance.

For a Sonic client and server communicating via the network, both having the Nagle algorithm disabled, only the socket connection between those points has the algorithm disabled. Disabling the Nagle algorithm for these two client/server connections does not disable the algorithm for the entire network.

Enabling and Disabling the Nagle Algorithm

The Nagle algorithm is disabled by default in SonicMQ. The following broker.ini parameter is used to enable and disable the Nagle algorithm on the broker.

TcpNodelay Property

Set this parameter to true if you want to disable the Nagle algorithm, and minimize small message latency. A value of false enables the Nagle algorithm.

The default value of **TcpNodelay** is **true**.

Table 39: Access to TcpNodelay Settings

Technique	Details
Management Console	Broker Properties > Tuning tab
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.ITuningParametersType</code> Method: <code>setTcpNodelay(boolean value);</code>

You can also enable or disable the Nagle algorithm for the client. This is done using a system property setting (or `-D<command-line arg>`) in the JVM used for your JMS client.

- Disable the Nagle algorithm: `-DSonicMQ.TCP_NODELAY = true`
- Enable the Nagle algorithm: `-DSonicMQ.TCP_NODELAY = false`

Message Size

This chapter contains the following sections:

- Overview on page 92
- Estimating Message Sizes on page 93
- Tuning Table on page 94

Note: Memory Sizing Worksheets — SonicMQ 2015 documentation includes sizing worksheets to help you estimate memory required for a broker and its destinations (queue and topic). You enter counts and sizes that you anticipate in your deployment, and the worksheet calculates approximate memory requirements. There are two worksheets, one for systems with 32-bit processors (`32bit.xls`), the other for systems with 64-bit processors (`64bit.xls`). Each memory sizing worksheet contains notes and comments to guide you through the gathering of data, and the interpretation and application of results. You can access these files at the remote documentation site accessible from the `welcome.htm` file in your installation root, or from the package you install locally from either the Sonic download site or media.

For details, see the following topics:

- [Overview](#)
- [Estimating Message Sizes](#)
- [Tuning Table](#)

Overview

Message Size is a key factor to consider when tuning a SonicMQ broker. The following guidelines set some guidelines.

- **Large Messages are dominated by disk and network speed** — When messages are large, the performance of SonicMQ is dominated by how fast bytes can be transferred over the network, and to/from the SonicMQ broker's disk. Relatively speaking, there is less messaging overhead, per byte, when messages are large.

With large messages, Message Buffers should be sized to allow for some concurrency (2-3 times message size), but they do not need to hold as many messages as they did when the messages were small.

- **Know your Quality of Service and messaging model** — You do not need to change properties in areas not being used. For example, if you are using Queues, then you do not need to tune **OutgoingMsgBufferSize**. Similarly, if none of your subscribers are durable, then **PsGuarMsgBufferSize** will not be an issue.
- **Large buffers might increase latency** — Making buffers very large will increase the number of messages “in transit” (but only if subscribers slow down). This will increase latency in the system, increase memory usage, and increase “bookkeeping” overhead in the broker (such as message expiration checks).

As long as your message consumers are able to handle the total load, the size of the message buffers should be irrelevant. Larger buffers only provide a margin of safety that allows for burst loads in your messaging applications.

- **Small buffers will not cause failures** — The only danger in keeping message buffers too small is that you limit SonicMQ's options for maximizing concurrent processing. While this will hurt performance, it will never cause messages to be lost, or the broker to fail.

Estimating Message Sizes

When calculating message size in terms of memory usage, you need to account for:

```
(message overhead + properties  
size + message body size)
```

Message overhead is the memory used by the broker for storing information about the message. It varies from about 175 to 500 bytes, depending on whether security is enabled, the QOP encryption setting, the delivery mode, transactions, and such. This value is only valid for the memory estimates on the broker. (The 'on-the-wire' protocol overhead per message is typically smaller.) Compute the property size for each property as:

```
(UTF length of name + size  
of value + 1)
```


When you indicate that message acknowledgement is required, 8 extra bytes are used.

When you make a session transacted, each message uses 4 extra bytes.

Name lengths, such as the destination name or a routing node name, are directly reflected in the message overhead.

The size of a message's body is generally similar to the size of the related Java objects:

- For **MapMessage** types, compute the size of each property as:
(UTF length of name + size of value + 1)
- For **TextMessage** types, the size is computed as using UTF encoding. This writes two bytes of length, followed by the UTF representation of every character in the String, **s**. Each character in the String is represented as one, two, or three bytes, depending on the value of the character, with common ASCII characters using the shortest (1-byte) encoding. The total size would be:
(UTF length of String + 2)
- For **XMLMessage** types, the XML document is converted to its String representation prior to delivery, and can be sized as a **TextMessage**.

For more on UTF encoding in Java, see the Javadoc for the `writeUTF()` method in the `java.io.DataOutput` class.

Tuning Table

The following table attempts to summarize the key points in this manual in a cookbook form. For each tuning parameter, we have tried to list the messaging models where it is most important. Values of **1** or **2** indicate, respectively, that the parameter is of primary or secondary importance.

No value indicates that the messaging model does not use the parameter. The application messaging models listed are:

- **P/S non-durable** — Publish/Subscribe with nondurable subscribers.
- **P/S durable** — Publish/Subscribe with connected and disconnected durable subscriptions.
- **PTP** — Point-to-Point (Queue sender/receiver applications).
- **Txn** — Either XA or local JMS Transacted Sessions are used.
- **DRA** — Dynamic Routing is used (messaging between routing nodes).

The messaging models listed are not mutually exclusive. That is, you might have an application that uses Dynamic Routing (DRA) with Transactions and PTP queues.

For each parameter, some rough ballpark tuning values are given based on typical message sizes. When a parameter is important, and when your typical messages are at this size, the value shown is a good starting point for your tuning process.

In the table the values are:

- **1** when you should look at this property first.
- **2** when the property is of secondary importance.
- **Size** is a multiple of message size.
- **Dflt** means that you should use the default value for this situation.

The following table is the SonicMQ Broker tuning table:

Table 40: SonicMQ Broker tuning

Property	P/S non-durable	P/S durable	PTP	Txn	DRA	Message Size <1K	Message Size ~20K	Message Size ~250K
OutgoingMsg BufferSize	1	2			1	50K	150K	800K
PsGuarMsg BufferSize		1		2	1	75K	150K	800K
WaitMsgBufferSize		1				50K	100K	500K
PsDbQueueSize		2				Dflt	1M	5M
TxnBufferSize				1		32K	Size*N	Size*N
PtpDbQueueSize				2		Dflt	1M	5M
QueueDeliveryThreads				1		10+	10	5
QueueMaxSize			1			1M	5M	5M
QueueMaxSize			1			1M	10M	100M
RecoveryLogMaxFileSize		1	1	2	1	Dflt	100M	1G
RecoveryLogQueueSize		1	1			Dflt	Dflt	1M+
DMQ Maximum Size			2		1	1M	10M	100M
Routing Queue Max. Size					1	Dflt	1M	10M

HTTP Dispatch Threads

SonicMQ broker maintains a set of dispatch threads that are used to deliver messages to HTTP destinations and process HTTP responses.

This chapter contains the following sections:

- [Thread Pool for HTTP Direct Outbound](#) on page 83
- [Effects on Memory Usage](#) on page 85

For details, see the following topics:

- [Thread Pool for HTTP Direct Outbound](#)
- [Effects on Memory Usage](#)

Thread Pool for HTTP Direct Outbound

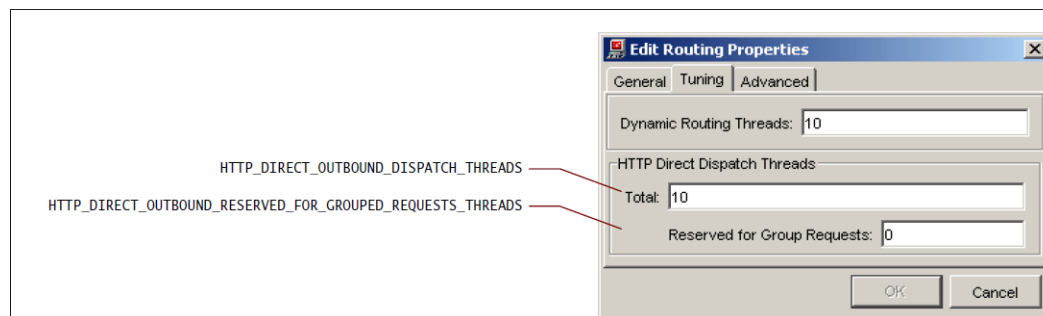
Each SonicMQ broker maintains a thread pool of a configurable size for dispatching messages to HTTP destinations. A separate pending queue is created for each instance defined by message grouping identifiers or common URL fragments. All messages in a pending queue instance are processed in the single-threaded manner so that message order is preserved.

The pending queues are processed by the pool of dispatch threads in a round-robin fashion. However, if a message from a pending queue is being processed by a dispatch thread, other dispatch threads will not process the messages from that pending queue.

You can limit the number of threads that are used to dispatch messages that are submitted to the broker with a specific URL but without using a group ID. This is particularly useful for the default HTTP Direct routing definition named `sonic.http` to send messages without GroupID flagging while other message streams that do use GroupIDs need a subset of the dispatch threads in the thread pool.

GroupIDs are given a bigger pool of threads so that they will be more processed more effectively than general HTTP requests:

Figure 35: HTTP Direct Outbound



HTTP_DIRECT_OUTBOUND_DISPATCH_THREADS property in routing properties on a broker specifies the number of threads to be used by the broker to dispatch messages to the HTTP Direct routing destinations.

Note: The **Dynamic Routing Threads** value specifies how many threads are used by this broker to establish Dynamic Routing connections—not HTTP Direct Outbound connections—to other brokers or to move messages out of Dynamic Routing pending queues.

The **HTTP_DIRECT_OUTBOUND_DISPATCH_THREADS** default value is **10** threads.

Table 41: Access to HTTP_DIRECT_OUTBOUND_DISPATCH_THREADS Settings

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name: <code>BROKER_ROUTING_PARAMETERS.HTTP_DIRECT_OUTBOUND_DISPATCH_THREADS</code>
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IRoutingParametersType</code> Method: <code>setIntegerAttribute("HTTP_DIRECT_OUTBOUND_DISPATCH_THREADS", int size);</code>

The **HTTP_DIRECT_OUTBOUND_RESERVED_FOR_GROUPED_REQUESTS_THREADS** property in routing properties on a broker specifies the minimum number of threads to be reserved by the SonicMQ broker to dispatch JMS messages on routing nodes that do not have Group Messages by URL option deselected or messages that either:

- Have the X-HTTP-GroupID JMS property set to a non-empty String value
- Do not have an X-HTTP-DestinationURL set to a non-empty String value

The value of the reserved threads must be an integer value that is no greater than one less than the total outbound dispatch threads. On the low end, the default value, 0, means that the broker is not required to reserve any HTTP dispatch threads.

Table 42: Access to HTTP_DIRECT_OUTBOUND_RESERVED_FOR_GROUPED_REQUESTS_THREADS Settings

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name: <code>BROKER_ROUTING_PARAMETERS.HTTP_DIRECT_OUTBOUND_RESERVED_FOR_GROUPED_REQUESTS_THREADS</code>
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IRoutingParametersType</code> Method: <code>setIntegerAttribute("HTTP_DIRECT_OUTBOUND_RESERVED_FOR_GROUPED_REQUESTS_THREADS", int size);</code>

Effects on Memory Usage

The amount of memory used by the broker to process HTTP Direct Outbound messages is:

- The size of in-memory buffers used to hold messages being processed by the HTTP dispatch threads. Each HTTP destination uses one buffer of about 75 KB. The number of these buffers is the number of pending queues.
- The size of the save extents for the pending queues. The size of one save extent is specified by the **FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD** broker configuration parameter (the default is **15000**). The number of the pending queues used for HTTP Direct Outbound messages depends on the number of routing definitions and GroupIDs used by the client applications.

Table 43: Access to FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD Settings

Technique	Details
Management Console	Broker Properties > Advanced tab > Edit Name: <code>BROKER_ROUTING_PARAMETERS.FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD</code>
Management API	Package: <code>com.sonicsw.mq.mgmtapi.config</code> Interface: <code>IBrokerBean.IRoutingParametersType</code> Method: <code>setIntegerAttribute("FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD", int size);</code>

Tuning JMS Clients

The broker configuration and tuning parameters are really only useful to fine-tune the performance and throughput of SonicMQ. The biggest impact on performance is really in the hands of the application architect. Choosing the appropriate levels for security, reliability, and messaging properties often do more to enhance performance than any broker tuning accomplishes.

This chapter contains the following sections:

- [Asynchronous Message Delivery Set on Connections](#) on page 88
- [Low Latency Messaging](#) on page 88
- [Managing Flow Control with Topic Messages](#) on page 89
- [Acknowledgement Mode](#) on page 90
- [Using Only Needed Security and Privacy](#) on page 91
- [Shared vs. Dedicated Connections](#) on page 92
- [Using Queue Prefetch](#) on page 92
- [Setting Brokers to Clean Up Connections Not Detected on the Socket](#) on page 92

For details, see the following topics:

- [Asynchronous Message Delivery Set on Connections](#)
- [Low Latency Messaging](#)
- [Managing Flow Control with Topic Messages](#)
- [Acknowledgement Mode](#)
- [Using Only Needed Security and Privacy](#)
- [Shared vs. Dedicated Connections](#)
- [Using Queue Prefetch](#)
- [Setting Brokers to Clean Up Connections Not Detected on the Socket](#)

Asynchronous Message Delivery Set on Connections

Some message delivery modes set on message producers are explicitly asynchronous—**NON_PERSISTENT** on a security-disabled broker and **NON_PERSISTENT_ASYNC** delivery mode. You can add asynchronous operation to **NON_PERSISTENT_REPLICATED** delivery mode, the delivery mode used by fault-tolerant brokers replicating non-persistent messages from the active peer to its standby by setting the client connection to enable the feature.

Asynchronous message delivery enables clients to buffer messages. You can set the delivery doubt window so that the send call will block when n number of messages have not been reliably delivered to the broker

When a session closes, the close operation waits for all the messages to arrive. You can set a timeout to limit the wait. A rejection listener reports asynchronously delivered messages that could not be delivered to the broker. All asynchronous delivery failures are reported using a **JMSAsyncDeliveryException**.

For more information, see “Asynchronous Message Delivery” in the “SonicMQ® Connections” chapter of the *Aurea® SonicMQ® Application Programming Guide*.

Low Latency Messaging

For some messaging scenarios client and broker connections introduce a short delay before flushing data to an underlying socket. This has the effect of increasing throughput in many cases, but can in some case cost the application in terms of latency. On the client this delay can be disabled by setting the system property:

```
-DSonicMQ.CLIENT_SEND_DELAY=0
```

The delay on the broker can be disabled by setting the advanced broker property through the management console:

```
CONNECTION_TUNING_PARAMETERS.BROKER_SEND_DELAY=0
```

Also see [Tuning Garbage Collection To Reduce Latency Spikes](#) on page 124.

Managing Flow Control with Topic Messages

Due to the implicit load-balancing that is associated with queues, one slow queue receiver will not limit throughput: other queue receivers will simply receive more messages. However, the nature of Publish and Subscribe messaging (zero-to-many subscribed consumers for each message) means that a slow consumer can slow the message flow to all consumers. SonicMQ provides the following additional techniques to manage Pub/Sub flow control:

- [Using Client Persistence With Wait Time](#) on page 89
- [Disabling Flow Control for Publishers](#) on page 89
- [Controlling Flow-to-disk](#) on page 90
- [Using Discardable Delivery Mode](#) on page 90
- [Using Shared Subscriptions](#) on page 90

Using Client Persistence With Wait Time

Publishers that are using client persistence can configure the persistent client to write messages to the local store when a message producer is flow-controlled. When producer flow control is no longer in effect, persisted messages flow to the broker in order while the message producer continues to add messages to the local store. When the local store is cleared, messages flow directly from the producer to the broker.

See “Client Persistence” in the chapter “SonicMQ Connections” of the *SonicMQ® V6.1 Application Programming Guide* for detailed information.

Disabling Flow Control for Publishers

Publisher applications can choose to tolerate backups, buffering any pending messages locally. Alternatively, an application developer can choose to handle the exceptions that are thrown when the publish method does not succeed due to a slow consumer. If you want to prevent the normal flow control throttling and handle exceptions you can invoke the following method (as illustrated for the Java client) on the **TopicSession** that is sending messages:

```
// Cast the session properly to utilize SonicMQ features.  
( (progress.message.jclient.TopicSession) session ).setFlowControlDisabled(true);
```

Managing the exception usually involves discarding messages before they are sent from the publisher, even though some consumers might not be experiencing problems.

Note: You can also use a similar method on QueueSessions if you want QueueSenders to be notified of flow-control situations explicitly.

Send Timeout for Message Producer

Client applications can take control back if a message producer's send() blocks for too long by setting the send timeout value. For more information, see "Send Timeout" in *Aurea® SonicMQ® Application Programming Guide*.

Controlling Flow-to-disk

As described in Flow to Disk Publishing on page 36, a SonicMQ client, when connecting to a SonicMQ broker, can choose to override the broker's FlowToDisk setting at the time of creation of a **ConnectionFactory**, or **Session**.

Using Discardable Delivery Mode

You can set the delivery mode of publish method to **DISCARDABLE**. This delivery mode is an extension of **NON_PERSISTENT_ASYNC** that instructs the message server to deliver all messages to subscribers that are keeping up with the flow of messages but to drop messages from this publisher to a lagging subscriber under any of the following conditions:

- When the message server's internal buffers for that subscriber session are full
- When a neighbor cluster member containing a topic subscription is unavailable and a subscriber is located on the other cluster member
- When an intended durable subscriber is unavailable
- When a routing connection is flow-controlled, or not created

Using Shared Subscriptions

A good approach to dealing with slow subscribers is to use Shared Subscriptions—a pool of subscribers that will each receive a fraction of the messages on a topic.

If you have a slow application, you can simply create a group of similar applications. Different members of the group will handle different messages allowing the throughput on the topic to exceed the individual receive rates.

It is also possible to add one or more disconnected durable subscriptions to the group of slow subscribers. In this case, the active subscribers will receive all the messages up to the point that they would otherwise flow control (in other words, new messages would exceed the threshold size in either **OutgoingMsgBufferSize** or **PsGuarMsgBufferSize**). Instead of throttling the publishers, the disconnected durable member of the shared subscriptions will start persisting messages.

See the *Aurea® SonicMQ® Application Programming Guide* and *SonicMQ C# Client Guide* for more examples showing the use of SonicMQ shared subscriptions.

Acknowledgement Mode

A session's acknowledgement mode sets the session to have all messages produced in its scope acknowledged in one of the following ways:

- Automatically by message receivers — The default is **AUTO_ACKNOWLEDGE**. A “lazier” automatic mode is **DUPS_OK_ACKNOWLEDGE**, the fastest acknowledgement mode you can choose if your application can handle duplicate message delivery.

Note that **AUTO_ACKNOWLEDGE** always has the potential to have one message redelivered (if there is a system failure between the time the application's `onMessage()` code finishes, and when the client sends the acknowledgement to the broker).

DUPS_OK_ACKNOWLEDGE typically extends the redelivery window from one message, to a few messages. It does not say that all messages will be redelivered, only that it is OK to redeliver a few, in failure cases. In addition, in SonicMQ® system, you will not get redelivery without some failure (for example, an abnormal client, broker, or network failure).

- By explicit acknowledgement under the control of the receiver — When you choose explicit acknowledgement, the receiver invokes the acknowledge method to notify the broker that it accepts all the preceding messages in the session (when the session is under CLIENT_ACKNOWLEDGE mode) or only the immediately preceding message (when the session is under SINGLE_MESSAGE_ACKNOWLEDGE mode, a unique Aurea SonicMQ® option).
- Within a Transaction — When a transacted session is used, messages are acknowledged in blocks when the transacted session is committed (either locally within JMS, or through distributed transaction resources such as XA).

Using transacted sessions overrides the explicit acknowledgement modes. Because extra commit/rollback steps are required, using a transacted session will reduce throughput if only a single message is contained in each transaction.

See the *Aurea® SonicMQ® Application Programming Guide* and *SonicMQ® C# Client Guide* for information about session settings for acknowledgement mode.

Using Acknowledge and Forward

SonicMQ provides a method that offers the increased reliability of acknowledge and forward routing without the overhead of a transacted message. Without this feature, failure-proof routing is only possible through use of a JMS Transacted Session.

See the *Aurea® SonicMQ® Application Programming Guide* and *SonicMQ C# Client Guide* for information about using acknowledge and forward.

Using Only Needed Security and Privacy

Security and privacy are important in any communications. However, the range of Quality of Service and Quality of Privacy options in SonicMQ let you choose to make some messages—for example, public catalog updates—unencrypted over unencrypted communication channels.

Other messages might be sent over encrypted channels to security-enabled brokers that have destinations that call for intense ciphers for QoP encryption of messages.

Every increment to security and privacy has a corresponding impact on overall performance. Tune your security and privacy usage to get the best balance of performance and service.

Tuning Client QoP Cache Settings

Performance when using QoP is largely impacted when a connection first starts sending and receiving messages. The first time a client sends to a destination, the client does not know the broker configured QoP value for the destination, so it sends the message with a QoP set to PRIVACY and flags the message indicating that it is querying for the QoP setting on that destination. The broker sends back a QoP update, and the client stores the value in its QoP cache. Then, in subsequent sends, the client uses the preferred QoP setting in its cache. In deployments where there is a mixture of settings, the performance improvement can be significant.

When this client cache gets full, a least-recently-used algorithm drops older cached values and records new ones. Generally, this setting (128) is adequate and efficient. However, an unbounded producer producing to more than 128 topics in a round robin fashion would send every message with QoS of `PRIVACY` and a query for the preferred QoS setting.

Under MultiTopic publishing where the topic list has more than 128 topics, every message is sent out with `PRIVACY` and requires a QoS cache update. The performance impact could be substantial.

You can relieve the performance issue that results from this condition by calling `ConnectionFactory.setQosCacheSize(Integer size)` to establish a cache size that considers the actual number of application destinations. Be aware that there are internal topics and other artifacts that are also buffered in this cache so set a value that is 128 more than the number of topics in the largest list.

Shared vs. Dedicated Connections

When you have a limited number of application programs that are providing the bulk of messages produced or consumed, you can often increase throughput by dividing the application into sub-components that each use a dedicated JMS connection.

Typically, multiple connections working in parallel can support a higher throughput than can a single JMS connection that has a large number of sessions, producers and consumers.

Using Queue Prefetch

SonicMQ supports prefetching messages from a queue to optimize overall throughput. Prefetching allows a client to receive messages from the SonicMQ broker before the client explicitly requests the messages, eliminating the overhead of broker requests on a per-message basis. However, prefetching also changes the operation of the SonicMQ system by allowing messages to accumulate at the client until the number of messages reaches the application-defined count.

You can achieve some performance gain with prefetching, primarily on lightly loaded brokers, where a receiving client tends to govern overall throughput. When the broker is operating at full capacity, other factors (such as queue size and disk I/O) tend to limit message-delivery rates.

Setting Brokers to Clean Up Connections Not Detected on the Socket

When client connections drop unexpectedly, the broker to which the client was connected may not know that it should release the connection. In situations where dropped client connections are not detected by the server socket, the broker needs a way to clean up these connections so that flow control is not being applied on behalf of lost client connections.

To handle this situation, **`CONNECTION_TUNING_PARAMETERS.BROKER_PING_TIMEOUT`** lets the broker set the number of seconds after which the broker closes a client connection if no ping response is received. Set the value to **30** or more seconds (a value of **1** through **29** is not invalid but will be interpreted as 30) or set it to 0 (the default value) to disable the feature.

This property should be defined in relation to the **BROKER_PING_INTERVAL**. The value of the **BROKER_PING_INTERVAL** should be half the value of **BROKER_PING_TIMEOUT**, or less. The broker can then expect that it should receive a ping reply before the socket timeout expires, unless blocking or other timing conditions exist.

Important: The unit of measure for **BROKER_PING_INTERVAL** is milliseconds, and the unit of measure for **BROKER_PING_TIMEOUT** is seconds. Therefore, if you enable the **TIMEOUT** at its minimum value of 30, the appropriate value for the **INTERVAL** should be about **15000**.

When the value of **BROKER_PING_INTERVAL** is greater than the value of the **BROKER_PING_TIMEOUT**, the effective broker ping timeout is the greater of the value of the **BROKER_PING_INTERVAL** and **30**.

The **BROKER_PING_TIMEOUT** parameter is applied to the broker runtime when the broker starts. A change to the value requires a broker restart to effect the change.

The application of this parameter applies to TCP and SSL connections, not connections through HTTP Tunneling. If similar conditions are experienced for HTTP Tunneling connections, the **HTTP_CONNECTION_IDLE_TIMEOUT** parameter provides the resolution.

Note that this feature is different from **CONNECT_PING_TIMEOUT**, a parameter used at the time a client initiates a connection to a broker to resolve whether an attempt to use established connection identification indicates an application error or if the existing connection is no longer in use. The broker pings the existing connection to find out and then—after waiting **CONNECT_PING_TIMEOUT** milliseconds—either closes stale connections, or, if the broker receives a reply, returns an error to the application.

Tuning Replicated Broker Connections

By synchronizing primary brokers and their dedicated backup brokers, the standby broker is always ready to assume the active standalone role when the previously-active fails. When the failed broker resumes operation, the standalone broker takes the active role while the other takes the standby role.

The following topics describe ways to tune performance when using replicated brokers:

- [Backing Up Only Selected Brokers](#) on page 95
- [Tuning Replication Connection Properties](#) on page 96

For details, see the following topics:

- [Backing Up Only Selected Brokers](#)
- [Tuning Replication Connection Properties](#)

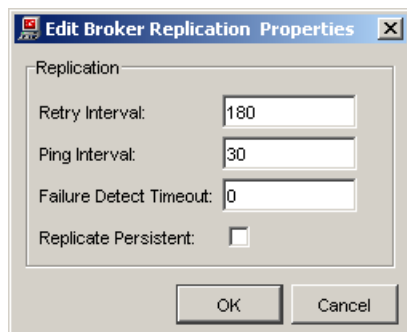
Backing Up Only Selected Brokers

One consideration is to create broker instances that are dedicated only to business-critical messaging. By backing up those brokers, the crucial business data and connections reach very high levels of service. Other brokers, such as those that publish quotes or catalog entries might be better suited to routine data store backups, particularly if their persistent storage mechanism can be backed up while running

Tuning Replication Connection Properties

The general settings for all replication connections on a primary/backup set are defined at by selecting **Replication Connections** on a broker then choosing **Properties**.

Figure 36: Tuning Properties of Replication Connections



The settings in this dialog apply to all the replication connections between the brokers.

Adjusting Intervals and Timeouts

The combination of the intervals and timeouts on replication connections sets the response level of every replication connection on the primary/broker pair where they are defined. The steps proceed as follows:

1. The two peers check every replication connection every **Ping Interval** (default **30**) seconds by sending a ping message on each connection's open socket.
Every half minute is a reasonable time delay. You could make it more frequent if you perceive a need to be more responsive.
2. If a connection does not respond or a notification is received that the socket has closed, each broker tries again right away, and then waits **Retry Interval** (default **180**) seconds before trying again to reestablish the problematic replication connection.
3. When all the replication connections are down, the broker in the standby role waits **Failure Detect Timeout** seconds then fails over to the standalone state. During this time, all replication connections are retried at one second intervals. As the default is set to **0**, failover is immediate upon realization that all replication connections are down. You might want to give the networks a final chance by setting enough time for a last test of the connections, say **10** seconds.

Replicating Persistent

You can specify that the replicated brokers should **Replicate Persistent**. This is a tradeoff between performance and Quality of Service. For example:

- Under the default setting of **false** (option cleared), the standby broker in a replicated broker pair acknowledges a message immediately on receipt and then persists it.
- When set to **true** (option selected), the replication is acknowledged only after the standby broker has persisted the data. This is crucial when rapid consecutive failures of both brokers occurs and the broker that was in the active role loses its storage media.

Tuning the JVM Properties

Your choice of Java Virtual Machine (JVM), Java heap size, and memory settings will have significant impact on your SonicMQ performance. The following sections discuss some issues you should consider when optimizing your JVM.

This chapter contains the following sections:

- [Choosing a Java Virtual Machine for the SonicMQ® Broker](#) on page 98
- [Setting the Java Heap Size](#) on page 98
- [Tuning JVM Parameters](#) on page 101
- [Tuning Garbage Collection To Reduce Latency Spikes](#) on page 102

Note: Memory Sizing Worksheets — Installation of SonicMQ 2015 documentation includes sizing worksheets to help you estimate memory required for a broker and its destinations (queue and topic). You enter counts and sizes that you anticipate in your deployment, and the worksheet calculates approximate memory requirements. The two worksheets, located in the `sonic_install_dir/Docs2015/tuning` folder, are different—one is for systems with 32-bit processors (`32bit.xls`), the other is for systems with 64-bit processors (`64bit.xls`). Each memory sizing worksheet contains notes and comments to guide you through the gathering of data, and the interpretation and application of results.

For details, see the following topics:

- [Choosing a Java Virtual Machine for the SonicMQ® Broker](#)
- [Setting the Java Heap Size](#)
- [Tuning JVM Parameters](#)
- [Tuning Garbage Collection To Reduce Latency Spikes](#)

Choosing a Java Virtual Machine for the SonicMQ® Broker

Both the SonicMQ broker and standard client are written in Java. The Java Virtual Machine that you use to run the SonicMQ broker can have a significant impact on overall messaging performance. Newer JVM versions allow for just-in-time compilation of Java classes, enhanced garbage collection, efficient input and output processing, and other significant capabilities. These advances can improve overall performance by as much as 300%, making the choice of JVM critical to attaining high performance levels.

Setting the Java Heap Size

A significant performance factor is the size of the Java heap, which you can specify with the **-Xmx** parameter as explained in the tables at the end of this chapter. Typically, this parameter is set to 128MB or 256MB. If you plan to send or receive very large messages or have multiple concurrent sessions in your application, you should increase the Java memory for the client machine.

You should determine your maximum heap size based on your available memory and on the size and number of messages and queues you anticipate handling on messaging nodes, and the size of the domain on management nodes. The following sections discuss these considerations.

Metrics that Measure Heap Size

You can set container-wide monitoring to include the metrics on heap size..

The **system.memory.*** container-wide metrics include:

Table 44: Metrics

Metric	Description
CurrentUsage	Heap space (in bytes) used by the container and its hosted components. Supports high thresholds for alert notifications. (See Specifying Alert Thresholds to set the alert thresholds).
MaxUsage	Maximum heap space (in bytes) used by the container and its hosted components since the last metrics reset.

For information on setting and reporting metrics, see the “Managing Containers and Collections” chapter of the *Aurea® SonicMQ® Configuration and Management Guide*.

Using the Maximum Available Memory for a Messaging Broker

To optimize performance, you should set the Java heap size to the maximum possible for your SonicMQ broker. This maximum size should correspond to the available memory on your machine. The more memory you set, the less Java will use the garbage collector. Reduced use of the garbage collector results in better performance.

However, you should be careful not to set the Java heap size too high. If this parameter exceeds the physical memory available to the JVM process, performance can significantly degrade as a result of page swapping in the underlying operating system. The memory available to the JVM might not match the total memory in the broker machine due to the memory requirements of other processes. In this case, lowering the total heap for the JVM will increase performance.

Important: Consult your operating system guide for information about process size limits so that you can specify an appropriate maximum heap size.

Adjusting Heap for the Directory Service in a Large Deployments

The heap space used by the Java runtime needs to be adjusted when the domain has a large number of components.

Adjusting the Heap Size Allocation for the Directory Service’s Container

The management container’s heap space is adjusted in the configuration of the container that hosts the Directory Service by adjusting the **Java VM Options** on the **Environment** tab. A practical entry for setting a large heap is to enter **-Xms32m -Xmx1024m** in the **Java VM Options** entry area. See the “Configuring Containers and Collections” chapter in the *Aurea® SonicMQ® Configuration and Management Guide* for more information.

Adjusting the Heap Size Allocation for SMCs Connecting to the Domain

Any administrator using the SMC to connect to a domain with a large number of components should also set a large heap value on the command line of the SMC launch script `sonic_install_dir/MQ2015/bin/startmc.bat` or `startmc.sh`, as follows: `"%MGMTCONSOLE_JRE%" "%SONIC_BOOT_CLASSPATH%" -cp "%SMC_CLASSPATH%" -Xms32m -Xmx1024m -Dsonicsw.home="%SONIC_HOME%" %SONICMQ_SSL_CLIENT% %SONICMQ_SSL_CLIENT_EXT% com.sonicsw.ma.gui.MgmtConsole.`

Determining Messaging Broker Memory Requirements

You can determine memory requirements by performing the following test:

1. Start the broker and the specific JVM.
2. Run an example application scenario with a fixed number of users for a reasonable amount of time (10 minutes).
3. Take periodic samples of the It is important to note that some JVM's will not necessarily release memory once the peak has been reached.

Multiply the value of the peak memory used for the testing interval by 1.3 to allow 30% padding for the Java garbage collection process. The result is the maximum heap size that should be allocated to the JVM (using the **-Xmx** setting).

Calculating Broker Memory Use

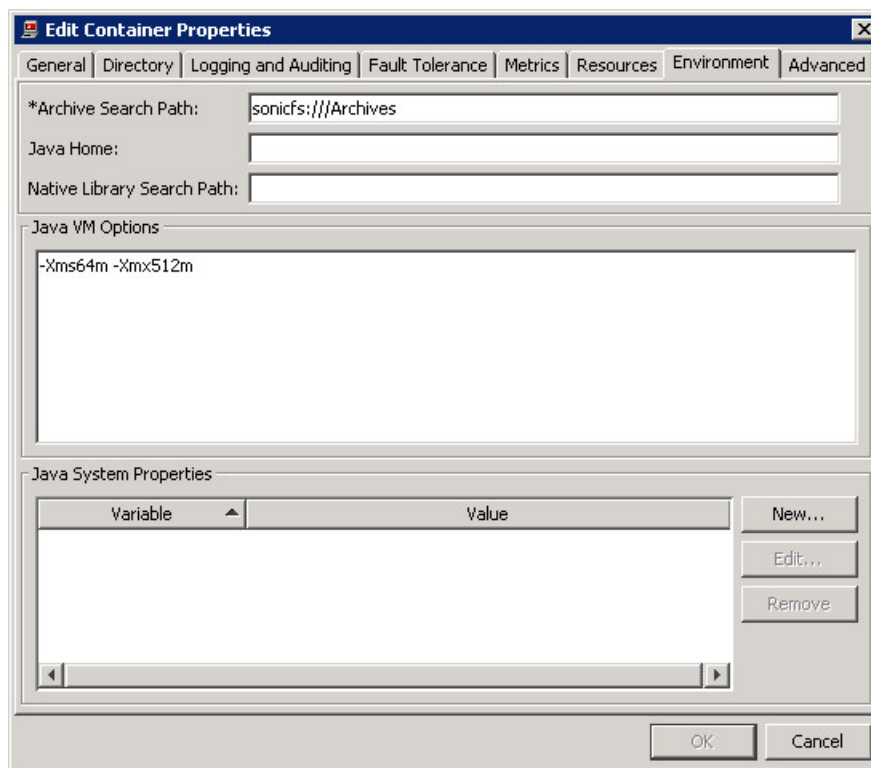
An approximation of required memory can be calculated by determining the sum of the following items:

- The broker allocates 5KB for each connection, session, queue receiver, queue browser, subscriber, and durable subscriber.
- The broker uses a buffered input stream mechanism to read message packets from the wire. This buffer is, by default, 8KB.
- The broker uses a buffered output stream mechanism to write messages to the socket. This buffer is, by default, 8KB.
- For each connection (queue, topic, cluster or routing), the size of each message as it is received prior to processing; in some cases, a copy is required so the upper bound is twice the size of each incoming message.
- Queue elements are allocated for every queue message in JMS queues whether the message is in memory or in the persistent storage mechanism. Each element is approximately 30 bytes + size of Java object header (which is JVM-specific).
- Each transaction requires approximately 2KB + `TXN_BUFFER_SIZE` (default is 8KB).
- For transaction commit: `TXN_THREADS * 2 * largest message in the transaction`.
- Message buffers might be maintained by the persistent storage mechanism.
- The sum of the save extents of all queues—the maximum amount of memory for messages on PTP queues.
- For the routing queue, for each remote node:
 - If the connection is slow or flow controlled, the broker keeps as much as `FLOW_CONTROLLED_PENDING_SAVE_THRESHOLD` (default 15MB) in memory.
 - If the connection has not been established, the broker keeps as much as `DISCONNECTED_PENDING_SAVE_THRESHOLD` (default 1MB) in memory.
- For each interbroker connection or routing connection, the greater of:
 - One message (of whatever size).
 - `GUAR_QUEUE_SIZE` (default 150KB) + `OUTPUT_QUEUE_SIZE` (default 150KB).

- For the server recovery log queue, the greater of one log event of whatever size, or LOG_QUEUE_SIZE (default 500KB).
- For the point-to-point storage queue, the greater of one message of whatever size, or PTP_DB_QUEUE_SIZE (default 500KB). Similarly the pub/sub data store queue uses up to SAVE_QUEUE_SIZE (default 500KB).
- For the Pub/Sub storage queue, the greater of one message of whatever size, or SAVE_QUEUE_SIZE (default 500KB).

Tuning JVM Parameters

JVM parameters for use by SonicMQ brokers are configured in the Container Properties:



The most commonly set parameters are those controlling the Java heap size.

Table 45: JVM Heap Size Parameters

Option	Description
-Xmsn	Sets the initial Java heap size in bytes. Typically <i>n</i> is postfixed with 'm' to specify a value in megabytes, e.g. -Xms64m
-Xmxn	Sets the maximum Java heap size in bytes. Typically <i>n</i> is postfixed with 'm' to specify a value in megabytes, e.g. -Xmx512m

You can run the following on a system to see the usage for the current Java version:

```
java -X
```

Tuning Garbage Collection To Reduce Latency Spikes

This section discusses tuning generation garbage collection (GC) with Sun JDKs. This section assumes that you are familiar with generational garbage collection, which is outlined in the GC documentation on Sun's web sites listed below. Also see the Low Latency Messaging topic.

Sun JDK 1.5

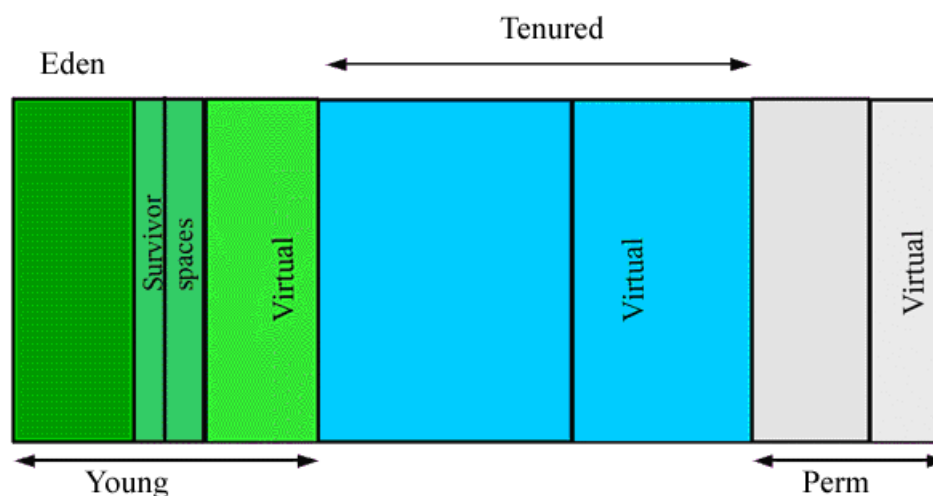
Note the introduction of `-XX:MaxGCPauseMillis=<nnn>` in JDK 1.5 which, when effective, makes GC tuning much simpler:

http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

Note: IBM JDK <http://www-128.ibm.com/developerworks/java/library/j-ibmjv2/index.html> Tuning GC for the IBM JDK is more complex, but choosing one of the following two policies is likely to produce the best results: **-Xgcpolicy:optavgpause** Trades high throughput for shorter GC pauses by performing some of the garbage collection concurrently. The application is paused for shorter periods. **-Xgcpolicy:gencon** Handles short-lived objects differently than objects that are long-lived. Applications that have many short-lived objects can see shorter pause times with this policy while still producing good throughput.

Sun's generational garbage collector groups objects into three generations: **young**, **tenured**, and **perm (permanent)**, as shown in the following figure.

Figure 37: Sun's generational garbage collector



The types of objects used by SonicMQ messaging will show how these groupings relate to SonicMQ objects:

- **Immortal** — Objects that exist for the lifetime of the JVM. An example on the broker is a queue.
- **Long Lived** — Objects that are related to JMS Connections, Sessions, Producers and Consumers are typically long lived and will require garbage collection once closed.
- **Short Lived** — Messages are frequently short lived, especially in applications with stringent latency requirements as they are received and delivered to consumers quickly.

Immortal and Long Lived objects will eventually end up in the tenured generation. Messages, on the other hand, are short lived objects that should not end up in the tenured generation for low latency applications.

Garbage collection pauses are proportional to the number of live objects in a generation. Major collections occur when the tenured generation fills up, and minor collections occur when the young generation fills up. Generally speaking if memory is available, it is advantageous to use a larger heap size so that minor and major collections occur less frequently since large sizes do not lead to long pauses just less frequent pauses.

In SonicMQ, major GC pauses are typically the longest pauses because there are usually many live Immortal and Long Lived objects to traverse in the tenured generation. Therefore, the goal of GC tuning for SonicMQ is to avoid major collections as much as possible. To accomplish this, the young generation should be tuned so that Short Lived objects do not leak into the tenured generation. This is discussed below. This also means that, from a programming standpoint, it is best not to create and close JMS objects because doing so causes the tenured generation to fill up, and lead to more frequent major collections.

Guidelines for Tuning Garbage Collection

Garbage collection (GC) tuning should always be done under the highest expected messaging and client load. Once a sufficient heap size for the JVM has been determined (see the preceding sections), it is best to fix the heap size by setting `-Xms` and `-Xmx` to the same value so that the JVM doesn't try to resize the heap, because heap resizing requires a costly major collection. It is also easiest to work with a fixed young generation size which can be achieved setting the `-XX:MaxNewSize` and `-XX:NewSize` JVM arguments to the same value.

To begin analyzing GC, use the following parameters to start with a young generation size of 8Mb with 1Mb survivor space and enable GC tracing:

```
-Xms32m -Xmx32m -XX:NewSize=8m
-XX:MaxNewSize=8m -XX:SurvivorRatio=6 -XX:+UseParNewGC -verbose:gc
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps
-XX:+PrintTenuringDistribution
```

This tracing might produce the output shown in [the following code sample](#):

Code Sample: GC Tracing Output

```
957.628: [GC 957.628: [ParNew:
6152K->8K(7168K), 0.0007314 secs] 10852K->4708K(31744K), 0.0008680
secs]957.991:
[GC 957.991: [ParNew: 6152K->12K(7168K), 0.0006993 secs]
10852K->4712K(31744K),
0.0008465 secs]958.358:
[GC 958.358: [ParNew: 6156K->12K(7168K), 0.0007370 secs]
10856K->4712K(31744K),
0.0008730 secs]958.719:
[GC 958.719: [ParNew: 6155K->12K(7168K), 0.0007076 secs]
10856K->4716K(31744K),
0.0008423 secs]
```

Given that the goal is to eliminate any memory growth in the tenured generation, these lines representing minor GC collections can be examined to see how the GC settings are performing. In the first line in the code sample, `10852K->4708K(31744K)` indicates that after GC total used memory was reduced to 4708K. At line two of code sample, you can see that memory was only reduced to 4712k. At first glance, this seems to suggest that 4k of memory has been allowed to leak into the tenured generation. This is not the case. This can be determined by looking at the young generation collections. At line 1, the young generation was reduced to 8k as indicated by `ParNew: 6152K->8K(7168K)`. In line 2, the young generation was only reduced to 12k, so the additional 4k of memory was retained in the young generation.

A comparison of lines 3 and 4 in code sample shows a point where some memory did leak into the tenured generation. The young generation in both is reduced to 12k however total memory usage increases from 4712k to 4716k meaning that 4k worth of objects was promoted into the tenured generation.

Note: When tuning GC for client applications, ensure that the cause of leaking objects into the tenured area is not the application itself holding on to objects for too long. This applies in particular to benchmarking applications that could hold on to per message statistics over a sampling interval.

Using GC PrintTenuringDistribution and MaxTenuringThreshold Parameters

You can then diagnose whether this 4k leaking into the tenured generation can be avoided, by adding additional GC parameters:

```
-XX:+PrintTenuringDistribution  
-XX:MaxTenuringThreshold=31
```

These parameters will print out the age distribution of objects in the survivor spaces, and also force objects to remain in the survivor area for 31 minor collections before being promoted to the tenured generation. Note that printing tenuring distribution details adds some cost to the pause times:

Code Sample: GC Tracing Output with Tenuring Distribution

```
77.440: [GC 77.440: [ParNewDesired survivor size 524288  
bytes, new threshold 31 (max 31)- age 1: 648 bytes,  
648 total-  
age 2: 3448 bytes, 4096 total- age 5: 64 bytes,  
4160 total: 6151K->8K(7168K),  
0.0083779 secs] 10931K->4787K(31744K), 0.0085273 secs]...88.384: [GC  
88.384: [ParNewDesired survivor size 524288  
bytes, new threshold 31 (max 31)- age 1: 1280 bytes,  
1280 total-  
age 2: 64 bytes, 1344 total- age 18: 608 bytes,  
1952 total-  
age 31: 3448 bytes, 5400 total: 6151K->12K(7168K), 0.0094099  
secs] 10931K->4791K(31744K), 0.0095484 secs]88.755: [GC 88.755:  
[ParNewDesired survivor size 524288  
bytes, new threshold 31 (max 31)- age 1: 744 bytes,  
744 total-  
age 3: 64 bytes, 808 total- age 19: 608 bytes,  
1416 total:  
6155K->8K(7168K), 0.0086335 secs] 10935K->4795K(31744K), 0.0087704  
secs]
```

The output in [the code sample](#) follows 3448 bytes leaking from the young generation into the tenured generation (intermediate minor collections have been omitted). The first minor collection shows the objects at age 2 and follows them to age 31 after which they are tenured.

At this point two things can be done, either:

- **-XX:MaxTenuringThreshold** can be increased to allow the objects to live longer in the survivor spaces
- The size of the young generation can be increased so that minor GCs occur less frequently giving the objects a longer time to die before being tenured.

The latter approach has the advantage of spacing out minor collection causing less frequent minor pauses. If the minor pauses are too long and leaking of objects into the tenured space cannot be avoided, then it may be appropriate to reduce the tenuring threshold so that leaking objects are tenured earlier.

If leakage into the tenured generation can't be avoided completely, it is often times not an issue. The above example shows that it took over a second for the data to be tenured. At a rate of 4k per second tenured and a tenured generation size of 24Mb a major collection would then occur every 102 minutes. For many applications a major collection pause at this frequency is acceptable. If not, the total heap size can be increased. For example, increasing the heap size to 256Mb would cause a major collection every 17 and half hours.

Refer to the JDK GC tuning guides for further tuning information. It also provides some mechanisms for incremental garbage collection of the tenured generation to reduce the major GC pause times, and newer versions of the JDK provide even more robust tools for garbage collection tuning.

Optimizing Broker CAA and Cluster Failover

This chapter contains the following sections:

- Overview on page 122
- Tuning Brokers to Optimize Failover on page 130
- Tuning TCP to Optimize Failover on page 137

For details, see the following topics:

- [Overview](#)
- [Tuning Brokers to Optimize Failover](#)
- [Tuning TCP to Optimize Failover](#)

Overview

Connections to Sonic brokers can fail for a number of reasons (processes die, machines fail, network hardware fails, wires get cut, etc.). Many failures do not result in immediate TCP layer identification and reporting of the failure, and thus a Sonic broker could be unaware of failure through a delay that goes beyond intended service levels.

SonicMQ's Continuous Availability Architecture (CAA) enables broker pairs to work together, one actively accepting client connections and performing broker functions while the other replicates the message traffic and connection state information in constant preparation to take on the active role when the active broker or all of their replication connections have failed.

Sonic broker clusters provide massive scalability and resilience. Each cluster member's connection to other members can have an alternate network to resume communication when a fault is detected on the primary interbroker connector.

Sonic lets you tune the broker behavior for connection failure situations to optimize failover without any platform-level delays. You can also tune the platform's TCP layer to control any of its delays, although these settings could be harmful to other applications running on that same host. However, Sonic's ping-based failure detection mechanism will detect connection-failures well before the operating system TCP layer on certain platforms configured with default TCP settings, such as on a Linux® system.

The following sections in this chapter show you how to:

- Configure Sonic brokers for fault detection in clusters and in broker replication
- Configure platform-specific mechanisms to tune connection failures detection.

Tuning Brokers to Optimize Failover

There are three aspects to network and broker failover:

- [Connection-failure Detection](#) on page 108 the ability to detect failure of an already established connection. Early connection-failure detection facilitates accurate recording of when failures occur, and timely triggering of application failover facilities.
- [Connect-failure Detection](#) on page 109 the ability to detect failure of an attempt to establish a connection—creating a socket connection, and then handshaking over the socket connection.
- [Failover Behavior](#) on page 109 which comprises the elapsed time for connection-failure detection, connect-failure detection, and establish the connection over another network, and—if all replication connections are lost—failover to the peer broker.

Connection-failure Detection

Brokers send ping requests to their peers after an interval of inactivity on an established connection. Peers are expected to immediately reply to the ping request with a ping response. When the ping timeout is a positive integer value and no ping response or other traffic is received within the timeout, the broker assumes that the connection is broken, so it reports the failure. Ping timeouts are required to be at least twice the duration of ping interval to allow for network latency, JVM garbage collection, and the effects of other processes.

The ping mechanism that supports connection failure detection can be set as low 5 seconds where the ping timeout defaults to 0 (thus, no ping timeout). That limit could be too low in some scenarios where environmental conditions would trigger frequent false negatives due to factors out of Sonic's control.

Configuration of ping intervals and timeouts will be per connection type: replication or cluster. To ensure predictable behavior, configure broker peers with the same ping intervals and timeouts.

You can use a ping timeout connection-failure detection mechanism:

- As a supplement to operating system TCP connection-failure detection mechanisms
- As a replacement for operating system TCP connection-failure detection mechanisms
- As a means to achieve reduced connection failure detection times for platforms that do not support configuration of TCP settings for early connection-failure detection

Connect-failure Detection

Early detection of connect-failure requires mechanisms to preempt socket connect calls and subsequent handshaking. Brokers limit the time they wait for establishment of replication and cluster connections using a configurable connect timeout. The preempting mechanism allows configuration to support connect failure detection within 5 seconds.

Connect-failure detection covers two stages of connection establishment:

- Creating a socket connection (initiating peer)
- Handshaking over a socket connection to establish connection type, authorization, etc. (initiating peer and accepting peer)

Failover Behavior

The failover behavior for cluster members and replicated peers vary slightly.

Cluster Members

Cluster members can be configured with interbroker acceptors for two available networks. One such acceptor is designated as the primary to support a connection model where a connection over a secondary network is not established until connection/connect failure over the primary network has been detected. Upon connection-failure detection over the primary network, a connect attempt failure over the primary network must occur before a connect attempt over a secondary network is attempted. The result is that failover to connection over an available secondary network is the sum of the elapsed time for:

- Connection-failure detection over the primary network
- Connect-failure detection over the primary network
- Establishing the connection over the secondary network.

Replicated Peers

Replicating brokers can be configured with replication connections on several available networks. Each network acceptor is configured with a weight. Replicated brokers attempt to maintain connections over each configured network all the time, but communicate using the connection established over the highest weighted network, the active connection. When connection-failure is detected on the active connection, broker replication traffic immediately switches to the next highest weighted network over which a connection has been established.

The result is that failover to a connection over an available secondary network occurs when connection-failure is detected on the currently active connection.

When all their replications connections have failed, the peer broker in the **STANDBY** state takes the **ACTIVE** role. As a result, replicated broker failover to its peer is the sum of the elapsed time for:

- Connection-failure detection on all replication connections.
- Establishment of the active state on the now-standalone peer.

Adjusting the Failover Properties

The failover properties have default settings. The following table shows how the default settings apply, followed by some examples of settings and the anticipated elapsed time before network failover in cluster. (Replication connection are different, they are already established.)

Table 46: Fault Detection settings for network failover in a cluster

Values	Connect Timeout (seconds)	Ping Interval (seconds)	Ping Timeout (seconds)	Maximum elapsed time before failover (seconds)
Default	30	30	0	60
Minimum	5	2	5	15
Example #1	30	30	60	95
Example #2	45	30	60	115

When a ping timeout is specified, the ping interval cannot be more than half of the ping timeout value.

You can adjust the time allowances from their default values to define an appropriate delay before acting but take care not to set values so low that you trigger unwarranted failovers.

You can set the failover properties in the Sonic Management Console, in administrative applications using the Configuration API, and specifying properties in a Sonic Deployment Manager (SDM) model.

Using the Sonic Management Console to Set Failover Properties

The broker properties that support these features are located in the Sonic Management Console as follows:

- **Clusters** — A cluster configuration's **Connections** tab lists the **Fault Detection** properties for all member of a specific defined cluster. The dialog box with its default values is as shown:

The screenshot shows a 'New Cluster' dialog box with a 'Connections' tab selected. Under the 'Fault Detection' section, there are three text input fields with the following values: 'Connect Timeout' is 30, 'Ping Interval' is 30, and 'Ping Timeout' is 0. Each field is followed by the unit 'seconds'. At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

For more information about setting cluster properties, see the **Clusters** section of “Configuring SonicMQ Brokers” chapter of the *Aurea® SonicMQ® Configuration and Management Guide*.

- **Replication Connections**—The **Connections** tab of **Replication Connections** for a primary broker lists the **Fault Detection** properties for all replication connection definitions that will be shared by the replicating peers. The dialog box with its default values is as shown:

The screenshot shows a dialog box titled "Edit Broker Replication Properties". It has two tabs: "Connections" (selected) and "Replication". Under the "Replication" section, there are two fields: "Retry Interval" set to 180 seconds and "Replicate Persistent" which is an unchecked checkbox. Under the "Fault Detection" section, there are four fields: "Connect Timeout" set to 30 seconds, "Ping Interval" set to 30 seconds, "Ping Timeout" set to 0 seconds, and "Failure Detect Timeout" set to 0 seconds. At the bottom of the dialog are "OK" and "Cancel" buttons.

For more information about setting replication connection properties, see the “Configuring Broker Replication” chapter of the *Aurea® SonicMQ® Configuration and Management Guide*.

Using the Administrative API to Set Failover Properties

Use the following Configuration API methods to set failover properties:

- In the interface `com.sonicsw.mq.mgmtapi.config.gen.IAbstractClusterBean`
 - `setClusterBrokerConnectTimeout(int value)`
 - `setClusterBrokerPingInterval(int value)`
 - `setClusterBrokerPingTimeout(int value)`
- In the interface `com.sonicsw.mq.mgmtapi.config.gen.IAbstractBrokerBean.IAbstractReplicationParametersType`:
 - `setConnectTimeout(int value)`
 - `setPingInterval(int value)`
 - `setPingTimeout(int value)`

Using the Sonic Deployment Manager to Set Failover Properties

In the `Tuning.xml` file for a cluster's parameter set, the highlighted properties set failover properties:

- **ClusterParameters**
 - **Annotation**
 - **ClientDefaultFCMonitorInterval**
 - **FlowControlMonitorInterval**
 - **ConnectTimeout**
 - **PingInterval**
 - **PingTimeout**
- **Replication**
 - **FailureDetectTimeout**
 - **PingInterval**
 - **ReplicatePersistent**
 - **RetryInterval**
 - **Annotation**
 - **ConnectTimeout**
 - **PingTimeout**

Tuning TCP to Optimize Failover

You can tune the TCP layer on the platform where clustered or replicated brokers so that the TCP values are closer to the broker configuration settings. While the intended result is faster failover, the enhanced settings for application level failover are often more precise, and will be more consistent in Sonic deployments across heterogeneous platform topologies.

The TCP settings for fast failover on platforms are sometimes difficult to adjust. These problems are notable on AIX® platforms as they limit TCP configurability. This section shows some settings you might apply on Windows® and Red Hat® Linux® platforms.

Important: The parameters and values discussed in this chapter are derived from IETF RFCs. The general concepts should be applicable, with some adjustments, on all the operating system platforms that Aurea Sonic supports for applications.

TCP Settings for Fast Failover on Windows® system

The Microsoft® Windows® scenario considers two server-quality computers each with two network connections: one for general client connections, and the other for interconnection for replication traffic. Both systems have SonicMQ brokers, licensed for Continuous Availability. The brokers are defined as primary and backup brokers and the replication connection is defined on the local network.

Note: A Microsoft® knowledge base entry describes the Windows® registry settings that affect the detection of a lost network connection: <http://support.microsoft.com/default.aspx?kbid=140325>.

Windows® Registry settings TCP/IP parameters were reviewed and changed to modify the TCP settings on the registry path:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

Table 47: Windows® Registry TCP IP Parameters that Affect Fast Failover

Parameter Name	Type	Range	Default Value	New Value
TcpMaxData Retransmissions	REG_DWORD	0x0-0xFFFFFFFF (retransmission attempts)	0x5 (5 attempts)	1
TcpMaxConnect Retransmissions	REG_DWORD	0x0-0xFFFFFFFF (retransmission attempts)	0x2 (2 attempts)	1

The keys in the above table will be added (or changed if they already exist) to reduce the attempts to connect and retransmit to assure a fast failover at the TCP level.

Applying the Settings

To apply the settings to the Windows® registry, do the following:

1. Create a text named **TCP-SETTINGS.reg**.
2. Enter:

```
Windows® Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters]
"TcpMaxDataRetransmissions"=dword:00000001
"TcpMaxConnectRetransmissions"=dword:00000001
```

3. Save the file.
4. Place the file on the Windows® system, and then run it. The registry is updated with these values.
5. Immediately restart the computer to apply the changes.

TCP Settings for Fast Failover on Linux® system

The scenario under Red Hat® Linux® uses the sysctl utility to change TCP parameters.

Note: A tutorial that details the procedure that follows is available at: <http://www.frozentux.net/documents/ipsysctl-tutorial> See also the Internet Engineering Task Force (IETF) RFC documents: <http://www.ietf.org/rfc/rfc0793.txt?number=793> and <http://www.ietf.org/rfc/rfc1122.txt?number=1122>

The following parameters are all dependent on the value set for Retransmission Timeout (RTO). See IETF RFC 793 for information about RTO.

The TCP parameters tuned in this RHLinux scenario are described in the following sections.

tcp_retries2

Sets the number of permitted re-transmission attempts of TCP packets before killing a TCP connection. The variable takes an integer value and is set to 15 by default. The elapsed time before timing out will be this value times the Retransmission Timeout (RTO) time. The example sets `tcp_retries2` to 3.

tcp_syn_retries

The `tcp_syn_retries` parameter tells the kernel how many times to try to retransmit the initial SYN packet for an active TCP connection attempt. This integer value should not be set higher than 255, because each retransmission consumes considerable time and bandwidth. As each connection retransmission takes about 30 to 40 seconds, and the default setting is 5, a connection times out after a delay of about 180 seconds. The example sets this to 1.

Example of conf settings

```
# Kernel sysctl configuration file for Red Hat® Linux®
# Controls IP packet forwarding
net.ipv4.ip_forward = 0
# Controls source route verification
net.ipv4.conf.default.rp_filter = 1
# Controls the System Request debugging functionality of the kernel
kernel.sysrq = 0
# Controls whether core dumps will append the PID to the core filename.
# Useful for debugging multi-threaded applications.
kernel.core_uses_pid = 1
```

Applying the Settings

To apply the settings to `/etc/sysctl.conf` on target systems, do the following:

1. Create a configuration file as described in the example.
2. Save the changed file on the target computer as **FastFailover.conf**.
3. Run:

```
sysctl -p FastFailover.conf
```

When you run the configurations, you should see a dramatic reduction in the time it takes to detect a lost connection, move through several replication connections, and ultimately discover that the replicated broker's peer is inaccessible, and then failover.

The expectation is that the default settings allow for 15 to 30 minutes with a single network while the specified settings reduce that interval to several seconds.

All the preceding settings at their optimal values should failover to be really fast: a few seconds on one replication connection and only a minute or so when there are dozens of networks assigned to replication connections for a SonicMQ replicated broker pair.