



iBus[™]

Softwired

MessageServer

*Server Performance and
Memory Manual*

Release 4.5

www.softwired-inc.com

Softwired AG
Technoparkstrasse 1
CH-8005 Zurich, Switzerland
+41-1-445-2370

info@softwired-inc.com

<http://www.softwired-inc.com/products/products.html>

© Copyright 2001 Softwired AG. All rights reserved.

Document Version : 4.5.O.252 7-NOV-2001/mwi

CHAPTER 1 *Server Architecture* 1-1

Introduction	1-2
Overview	1-2
Point-to-Point Pipeline	1-2
Publish-Subscribe Pipeline	1-3
Module Description	1-3
Socket Reader	1-3
Transaction Module	1-3
Transaction Manager	1-3
Storage Module	1-4
TopicStore	1-4
PersQueueStore	1-4
VolQueueStore	1-4
Distributing Module	1-5
Socket Writer	1-5
Summary	1-5

CHAPTER 2 *Flow Control* 2-1

Introduction	2-2
Why Flow Control?	2-2
Sources of Flow Control	2-2
Consumer-Flow Control	2-3
PersQueueStore/VolQueueStore-Flow Control.	2-4
TopicLiveBuffer-Flow Control	2-4
TopicStorage-Flow Control	2-4
Cleanup of Stores	2-5
Flow Control Live - the Admin Client	2-5
Configuring Flow Control	2-5
Consumer-Flow Control	2-5
PersQueueStore/VolQueueStore/TopicStore-Flow Control	2-5
TopicLiveBuffer-Flow Control	2-7

CHAPTER 3 *Advanced Memory Management* 3-1

Introduction	3-2
Server & AMM Design	3-2
Socket Reader	3-3
Transaction Manager	3-3
Storage Module	3-4

Thread/Buffer-Module	3-4
Queue/-LiveBuffer	3-5
Topic/-LiveBuffer	3-5
VolQueueStore	3-5
IndexCache	3-5
Queue-Threadpool	3-5
Topic-Threadpool	3-6
DurableTopicSubscriber-ThreadPool	3-6
DurableTopicSubscriberManager	3-6
MemWatch	3-6
Configuring AMM	3-7
Socket Reader	3-7
Transaction Manager	3-7
Storage Module	3-8
Thread/Buffer-Module	3-8
Queue/-LiveBuffer	3-9
Topic/-LiveBuffer	3-9
VolQueueStore	3-10
IndexCache	3-10
Queue-Threadpool	3-10
Topic-Threadpool	3-10
DurableTopicSubscriber-ThreadPool	3-10
DurableTopicSubscriberManager	3-10
Default Memory Allocation	3-11

CHAPTER 4 *Performance Tuning* 4-1

Introduction	4-2
What do you want to tune?	4-2
DeliveryMode:	4-2
Transactional:	4-2
Acknowledge-Mode:	4-3
Message Size:	4-3
Message Selectors:	4-3
Number of Connections:	4-4
Throughput versus Latency:	4-4
Low Memory versus Speed:	4-4
Where to tune	4-4
Throughput versus Delay	4-4
CPU Time Allocation	4-5

Memory Allocation 4-6
Default Performance Configurations 4-6
Comprehensive Tuning List 4-6

CHAPTER 5 *Advanced Memory Configuration* 5-1

Introduction 5-2
Parameters 5-3
Input values. 5-3
Notes about user input values 5-3
Output configuration parameters. 5-5



Preface

About This Manual	viii
How This Book is Organized	viii
Terminology and Typographical Styles	ix
	x
Contact information	x

About This Manual

This Manual describes the internal workings of the iBus//MessageServer. Besides giving information about the server architecture it shows how you can improve and dedicate it to a specific JMS application.

This manual covers three main topics: Flow Control, Memory Management and Performance Tuning. Understanding how these three areas work and what they mean helps the JMS application developer to modify and make use of the highly extensible and flexible architecture of the iBus//MessageServer.

How This Book is Organized

This document is organized in the following manner:

Server Architecture describes the main modules and how they interact.

Flow Control describes the design of flow control in the iBus//MessageServer.

Advanced Memory Management describes the architecture and design of AMM in the iBus//MessageServer.

Performance Tuning covers performance areas existing in the iBus//MessageServer and describes how they can be tuned.

Advanced Memory Configuration describes the Softwired Advanced Memory Management Configuration tool.

Terminology and Typographical Styles

Text style

Text Emphasis	Use of Emphasis
blue	For links to WWW, email contact addresses, or references to other documents
red	For cross references within this document
<i>italics</i>	For discretionary words and file names.
<code>courier font</code>	For code
“double quotes”	For citations.

Directions

Direction	Direction Meaning
Caution	Directions, which if not followed could cause damage or implementation problems/errors.
Note	Directions intended to highlight unusual points as an aid to the user.

Person Specification

Direction	Direction Meaning
User	Includes the body with authority over the equipment and those persons who actually handle the equipment.
Qualified Person	Those legally permitted to work on this equipment.
Authorized Person	Those specifically authorized by local management.

Technical Support

Technical support is available from Softwired AG by the following methods:

- Telephone +41 1 445 2370
- Email info@softwired-inc.com
- Website: www.softwired-inc.com

When making contact with Softwired please have the following information available:

Software type	
Version number	
Serial number	
Operating system	
Contact name and details	

Contact information

Worldwide Product & Support Center

Softwired AG
Technoparkstrasse 1
CH-8005 Zurich
Switzerland

Tel + 41 1 445 2370

Introduction	1-2
Overview	1-2
Module Description	1-3
Summary	1-5

Introduction

The first step towards understanding how the three main aspects of the iBus//MessageServer work is to have an understanding of the architecture. The following sections describe the main modules and how they interact.

Overview

The iBus//MessageServer consists of specialized modules each designed for a specific task. Figure 1-1 gives the top-level view of the server.

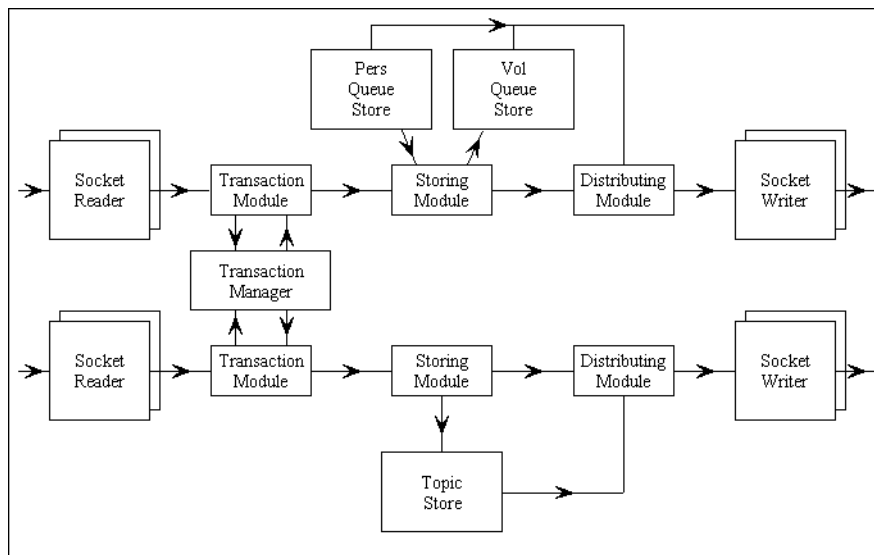


Figure 1-1 Top level view

The server core is built using a highly flexible pipeline architecture. Each processing step is represented by a pipeline module. Both JMS messaging domains (Point-to-Point and Publish-Subscribe) have a separate pipeline. The figure shows the point-to-point (p2p) pipeline in the upper half and the publish-subscribe (pubsub) in the lower half.

Point-to-Point Pipeline

Incoming messages are read by *socket readers*. There is one socket reader per connection. These modules pass the messages to the

transaction module. Once there, messages are either forwarded to the *transaction manager* (if the session is transacted) or directly to the *storage module*. In the p2p pipeline there are two different *stores* in use: *persistent* messages are stored in the so-called *PersQueueStore*, the *non-persistent* messages in the *VolQueueStore*. The *VolQueueStore* holds a configurable amount of messages in memory while the *PersQueueStore* stores the messages to a database. This allows high-speed handling of non-persistent queue messages. The last step in the pipeline is the *distributing module*: The messages are distributed to the single *queues* and the *QueueReceivers* thereafter.

Publish-Subscribe Pipeline

The pubsub pipeline is very similar to the p2p pipeline. Incoming messages are read by the socket readers and then passed to the transaction module. The storage module stores persistent messages in the *TopicStore*. Note that non-persistent topic messages are not stored and are discarded as soon as they are delivered to connected *TopicSubscribers*. Persistent messages, on the other hand, are stored in a database and are available for later retrieval by *DurableTopicSubscribers* in the so-called *replay*.

Module Description

Socket Reader

Each JMS connection within the server is handled by a socket reader. This module reads messages and forwards them to the pipeline.

Transaction Module

The first pipeline step consists of transaction handling. If the message belongs to a transacted session, it is passed on to the transaction manager. Otherwise the message is forwarded to the storage module.

Transaction Manager

This manager holds all messages belonging to open transactions until the JMS application issues a commit command. On a commit the transaction is passed on to be stored in an atomic fashion. This module can be configured to limit the size of the messages in open transactions to avoid memory overflow; it also holds XA transactions.

Storage Module

The storage modules interact with the *stores*, which themselves access either the configured database or - in the case of the *VolQueueStore* - with in memory store.

TopicStore

The *TopicStore* is responsible for storing, replaying and deleting topic messages. Storage of messages is handled via the storage module.

Replaying refers to the phase during which a *DurableTopicSubscriber* is reading messages from a database. This is necessary if the *DurableTopicSubscriber* was disconnected while persistent messages were sent on its topic. Deletion of messages is done by a cleanup process which takes care of messages that have passed their expiry time. Both replay and cleanup processes can be configured on speed and memory use.

Note that non-persistent topic messages aren't stored in the database. They are only available for connected *TopicSubscribers*. Once all connected and interested (i.e. whom the *MessageSelector* matches the message) *TopicSubscribers* have received a non-persistent message, it is removed from memory.

PersQueueStore

This service is the analog of the topic store for point-to-point messages. It stores queue messages, delivers them to *QueueReceivers* on demand and deletes the messages once the *QueueReceiver* has acknowledged them. The *PersQueueStore* also has a cleanup process which removes messages if they have timed out.

VolQueueStore

Since non-persistent messages do not require persistent storage, queue messages with this delivery mode are held in memory until they are consumed by a *QueueReceiver*. The amount of memory used to store non-persistent queue messages can be configured, i.e. limited. As with all Store modules, if this limit is reached - e.g. if no *QueueReceiver* is connected or very slow - the producer needs to be stopped from sending further messages. This Flow Control feature is further described in Chapter 2.

Distributing Module

In the next pipeline step messages are first dispatched to the destination they are bound for, then the destinations themselves distribute messages to the individual consumers.

Topic messages are delivered to all TopicSubscribers, queue messages are guaranteed to be delivered to exactly one QueueReceiver. Message selectors are evaluated in this step. A message will only be sent to consumers with a message selector that evaluates to true for that message, or consumers that do not have a message selector.

Socket Writer

At the end of the pipeline all messaging actions have been completed (i.e. priorities, delivery modes, message selectors etc.) and the message is sent directly to the consumer.

Summary

The iBus//MessageServer is built on an extensible and flexible pipeline architecture. Each module and service has a variety of configuration parameters which allows the server to be specialized for various JMS application scenarios. The following chapters will cover all areas of available specialization and optimization features.

Introduction	2-2
Why Flow Control?	2-2
Sources of Flow Control	2-2
Flow Control Live - the Admin Client	2-5
Configuring Flow Control	2-5

Introduction

Flow Control is one of the most important aspects of a messaging system. Each transport protocol (see TCP) has flow control capability and highly builds on this feature. This chapter describes the design of flow control in the iBus//MessageServer.

Why Flow Control?

Flow Control limits the bandwidth of connections from the server to the consumer and from the producer to the server. Flow Control is necessary to avoid memory overflow and loss of messages downstream, i.e. if the server sends messages to the consumer too fast, the consumer may encounter a memory overflow. The server needs to be aware of the speed of the consumer and adapt to it. The producer also has to be limited to a certain bandwidth depending on resources in the server and its consumers.

If flow control is not used, resource limits in either the server or the consumers could be exceeded and stability is no longer guaranteed. Therefore in a messaging system, flow control is a necessity.

Sources of Flow Control

A JMS messaging system can be viewed as supporting unidirectional traffic. There are senders and receivers of messages. A generic JMS messaging scenario has two flow control areas:

- From the JMS consumer to a JMS server.
- From the JMS server to a JMS producer.

This means that *a JMS consumer is a source of flow control*: it decides whether it is ready to consume messages or whether the server should stop sending messages. To deal with this, the server monitors the size of the consumer's queue and stops sending messages to the consumer if the queue is full; the client will announce message consumption or acknowledges to the server which then causes the consumer queue to shrink again.

Another obvious *source of flow control is the server* itself. If the server is not able to receive further messages from a (set of) producer(s) it sends a **flow control-stop** command to the

corresponding producer(s). Eventually the server will be ready to receive more messages and sends a *flow control-start* to the same producer(s).

Besides the JMS consumer the server has many internal resources which can cause flow control to be switched on. Figure 2-1 shows the sources of flow control within the iBus//MessageServer.

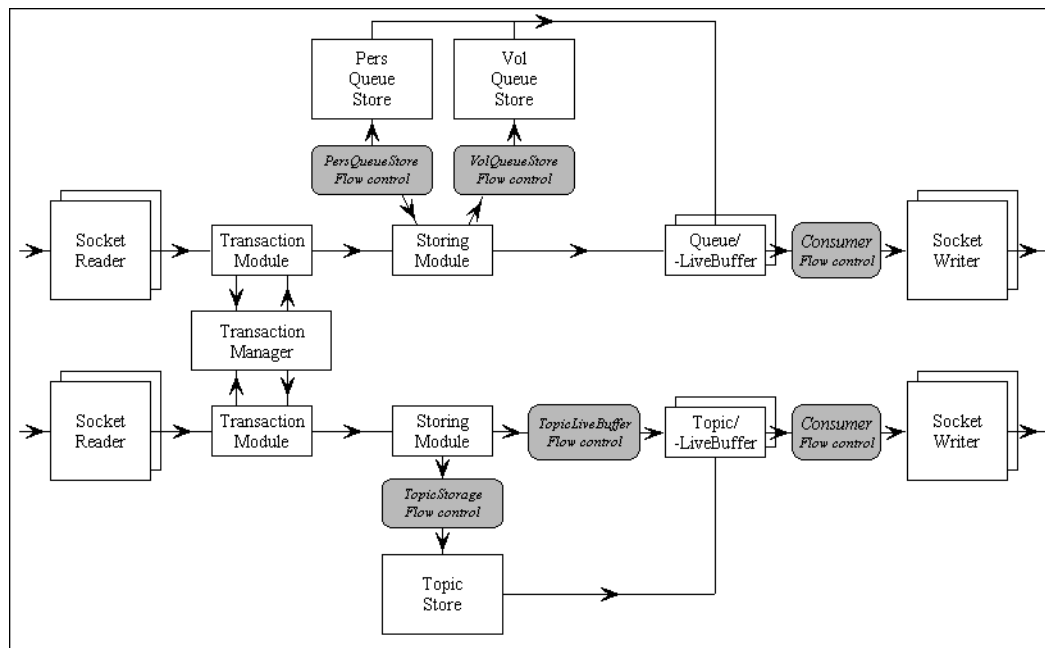


Figure 2-1 Flow Control sources

Here is a short description of the flow control sources within the server.

Consumer-Flow Control

Each consumer has a queue for incoming messages (aka buffer). The server keeps track of the available capacity in each consumer's queue and stops sending messages to the corresponding consumer once the configurable limit on the queue is reached. This limit can be both number of messages and total size of the messages. If the consumer is a TopicSubscriber this will cause a stop in the distribution process for topic messages once a message arrives for this TopicSubscriber.

Therefore such a TopicSubscriber can cause the so-called TopicLiveBuffer to fill up. If the consumer is a QueueReceiver this looks different. The distribution process will no longer send messages to this consumer but will to all other consumers. If there are no other

QueueReceivers or none at all this will cause the PersQueueStore and/or the VolQueueStore to fill-up.

PersQueueStore/VolQueueStore-Flow Control.

Storage within the PersQueueStore and the VolQueueStore can be configured to be limited per queue and/or globally (all queues combined). If the limit of one queue is reached this will cause a flow control-stop on the specific queue: Flow Control sends a flow control-stop to all connected QueueSenders. If the global limit in one of PersQueueStore or VolQueueStore is reached it will cause Flow Control to send a stop-command to all connected QueueSenders.

Generally one should configure these limits in the PersQueueStore very generously: the global limit should only be used to limit the disk space used (i.e. 4GBytes). The VolQueueStore on the other hand should be *configured with caution: the limit corresponds to the amount of heap memory* used for volatile messages.

TopicLiveBuffer-Flow Control

Each topic has a **LiveBuffer** associated which helps isolate TopicPublishers from TopicSubscribers: A TopicPublisher should be able to send messages independent on connected (slow) TopicSubscribers; The LiveBuffer holds messages ready for delivery. If a TopicSubscriber sends a flow control-stop to the server, this can cause the LiveBuffer to grow. Once a limit on this buffer is reached Flow Control will send a flow control-stop command to all connected TopicPublishers on this topic. If a global limit for TopicLiveBuffers is reached, this command will be sent to all TopicPublishers.

Note that the TopicLiveBuffer can cause producers to be stopped if there are slow consumers. The QueueLiveBuffer on the other hand does not stop any producers, as long as the PersQueueStore and VolQueueStore are not full.

TopicStorage-Flow Control

This Flow Control works similarly to the PersQueueStore/VolQueueStore-Flow Control: The TopicStore can be configured to hold a limited amount of messages in the database (per topic and globally). If one of these limits is reached, the TopicPublishers are stopped: if the global limit is reached, all TopicPublishers are stopped, if a single topic's limit is reached only TopicPublishers on that topic are stopped.

Cleanup of Stores

JMS messages carry an expiry date after which a provider is allowed to remove messages from further distribution. The message server has a cleanup process for each of the stores which is responsible for deletion of expired messages. Each cleanup process (for each store) can be configured individually to run after a given interval.

Flow Control Live - the Admin Client

The iBus//MessageServers Admin Console can be used to monitor the state of flow control. Each consumer's flow control state can be seen in the *Connection-Module*. If a consumer's incoming message queue is full its status is shown as **ready: no**. The status for topics and queues flow control is also shown. Click on the topic/queue name in the *Connection-Module* or in the *Messaging-Module*. The status for each of the above mentioned flow controls will be shown there.

Configuring Flow Control

This section describes the individual configuration parameters and how they can be set. Chapters 3 and 4 will have a closer look at typical configuration examples.

Consumer-Flow Control

The consumer's incoming message queue can be configured in the client's config file `config.ibusjms.txt`. The parameters are called:

`consumerQueueEntiresSize`

defines the number of messages allowed

`consumerQueueEntriesBytes`

defines the number of bytes allowed

PersQueueStore/VolQueueStore/TopicStore-Flow Control

The stores have flow control-limits per destination and globally. The parameters can be configured using the administration client in the module `PersQueueFileStore`, `VolQueueStoreImpl`, `TopicFileStore`.

The parameters per destination are (To switch off one of these parameters, use -1):

`msglowwatermark`

defines the lowwatermark for the number of messages

`msghighwatermark`

defines the highwatermark for the number of messages

`bytelowwatermark`

defines the lowwatermark for the message sizes in bytes

`bytehighwatermark`

defines the highwatermark for the message sizes in bytes

The global parameters are:

`globalmsglowwatermark`

defines the global lowwatermark for the number of messages

`globalmsghighwatermark`

defines the global highwatermark for the number of messages

`globalbytelowwatermark`

defines the global lowwatermark for the message sizes in bytes

`globalbytehighwatermark`

defines the global highwatermark for the message sizes in bytes

Note: The parameter `globalbytehighwatermark` is the most important. It defines the global amount of bytes for messages which are allowed to exist in the Databases. For the `PersQueueStore` and the `TopicStore` this limit - the summary of the two - must be smaller than what's available on disk. For the `VolQueueStore` the limit should be low enough to be held in the virtual machine's heap memory.

Note: Each Flow Control parameter has a lowwatermark and a highwatermark. These marks are used to trigger start and stop signals depending on the watermark of each parameter. For example when the number of messages starts increasing and hits the highwatermark, flow control triggers a stop. Once a stop has been triggered, the lowwatermark must be reached before a start is issued again. Therefore these parameters can help reducing the number of start/stop triggers.

TopicLiveBuffer-Flow Control

The parameters for the `topiclivebuffer` can be configured using the administration client in the module `DestinationManagerImpl`. The parameters are:

`topicmsglowwatermark`

defines the lowwatermark for the number of messages

`topicmsghighwatermark`

defines the highwatermark for the number of messages

`topicbytelowwatermark`

defines the lowwatermark for the message sizes in bytes

`topicbytehighwatermark`

defines the highwatermark for the message sizes in bytes

The global parameters are:

`topicglobalmsglowwatermark`

defines the global lowwatermark for the number of messages

`topicglobalmsghighwatermark`

defines the global highwatermark for the number of messages

`topicglobalbytelowwatermark`

defines the global lowwatermark for the message sizes in bytes

`topicglobalbytehighwatermark`

defines the global highwatermark for the message sizes in bytes

Advanced Memory Management

Introduction	3-2
Server & AMM Design	3-2
MemWatch	3-6
Configuring AMM	3-7
Default Memory Allocation	3-10

Introduction

The previous chapter has shown that there are critical resources in the iBus//MessageServer that need to be managed. Additionally, overall memory usage is a critical resource which requires special attention. iBus//MessageServer introduces a new approach called **Advanced Memory Management (AMM)** which allows the administrator to manage the server's memory depending on the JMS application in use. AMM gives the server the ability to guarantee stability while allowing the administrator to specially tune performance to use the optimum amount of memory for a given application. This chapter describes the architecture and design of AMM in the iBus//MessageServer.

Server & AMM Design

Figure 3-1 shows the modules that have memory critical parameters.

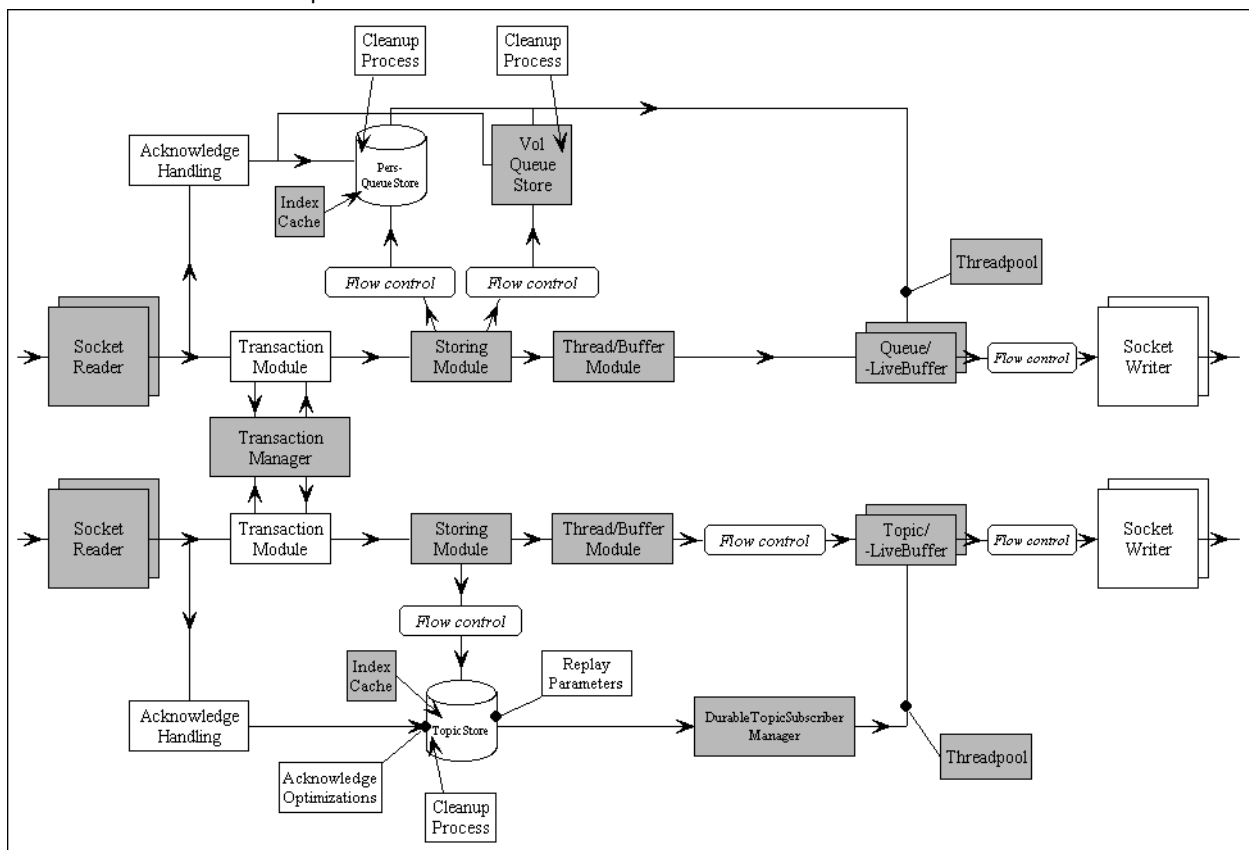


Figure 3-1 Modules with Memory Critical parameters

Each JVM can be configured with a maximum heap size (using the command line option `-mx`). While the default setting in many VMs is too small for server applications, iBus sets this value to 512MBytes per default (in the start scripts). Note that it is generally better for this value to be too large than to be too small, although excessively large values will reduce the maximum number of connections the server can handle.

The server itself can additionally be configured to run with a given maximum memory usage. The default for this memory usage is 128MBytes. Make sure to adjust the `-mx` parameter once you configure the server to use more than 512MBytes. (Also refer to the Server Administrator's Manual page 5-4)

Socket Reader

Obviously incoming messages need to be read into memory. Therefore especially messages need special attention. The `iBus//MessageServer` checks the current memory usage on message arrival to see if the message should be loaded into memory or not. If the memory usage is too high the server can be configured to wait until memory becomes free or to eventually discard the connection that sent the message.

Transaction Manager

Messages belonging to open transactions are held in memory before being written to the stores and passed on to distribution. They represent a potential risk for memory usage. The transaction manager can be configured to allow only a limited amount of memory to be used for such messages. If this limit is reached there are different strategies that can be invoked. Furthermore, an ordering qualifier of the strategy can influence if and when the strategy is applied to standard JMS transactions, and distributed transactions. Both strategy and ordering qualifier are configurable via the Administration Console.

The default strategy is to discard the largest transaction (in terms of memory usage). The default ordering qualifier applies the strategy first to standard JMS transactions and then to distributed transactions. The default strategy provides complete protection from memory overuse because it applies to both XA and standard transactions.

The strategies for freeing transaction memory are:

- discard the largest transaction in terms of transaction memory usage.
- discard the largest transaction in terms of number of messages.

- discard the oldest transactions first.
- discard the youngest transactions first.
- discard all transactions.
- discard none, which effectively switches this feature off.

Ordering qualifiers can be one of:

- First XA - all XA transactions which satisfy the strategy are freed before standard transactions.
- First Standard - all standard transactions which satisfy the strategy are freed before XA transactions.
- Mixed First XA - free XA and standard transactions alternatively which satisfy the strategy, starting with XA transactions.
- Mixed First Standard - free XA and standard transactions alternatively which satisfy the strategy, starting with standard transactions. This is for all practical intents and purposes equivalent to Mixed First XA.
- Only XA - only XA transactions are freed subject to the strategy.
- Only Standard - only standard transactions are freed subject to the strategy.

All distributed transactions which have exceeded their transaction timeout limits and have not been prepared are discarded upon reaching the memory limit, irrespective of the strategy or ordering qualifier. This is an implicit rollback of the distributed transaction, so that committing the transaction later is not possible.

Storage Module

The storage module contains a *Small Processing Buffer* which combines multiple store requests into a single one. This helps to speed up database accesses. The size of this buffer can be configured so that it does not hold more than a certain amount of memory (the default is 512kBytes for topics and for queues).

Thread/Buffer-Module

This module is a buffer for incoming messages: the messages are queued up and are later processed by another thread. This helps speed up server throughput since the thread reading messages from the socket can return before the messages really have left the server. This module should be configured large enough to allow a high decoupling of incoming and outgoing processes. The default is set to 1MByte. The thread which processes messages from this buffer will distribute the messages into the individual *LiveBuffers* for queues and topics, which continue processing data in a separate thread. To keep

a control over the number of threads, all queue, topics and durable subscribers have thread pools assigned.

Queue/-LiveBuffer

There is a LiveBuffer for each individual queue. These LiveBuffers hold messages that have been produced recently. Once these LiveBuffers reach their limit, no new messages are cached until more space is available. Messages which are not cached must be read from the database, thus allowing a reasonable size for these livebuffers increases speed. QueueLiveBuffers can be configured individually and/or globally.

Topic/-LiveBuffer

LiveBuffers for topics play a slightly different role than for queues. As with queues the topic LiveBuffers hold recently produced and not yet delivered messages. But unlike queues, when the limit is reached, Flow Control action stops publishers until more space is available.

VolQueueStore

Unlike the PersQueueStore this module holds queue messages in memory until they are either delivered successfully or have expired. Since the messages are kept in memory there must be a limit for the amount of memory that should be used. Unlike the queue LiveBuffer the limit on this store causes Flow Control to stop the corresponding QueueSenders (or all QueueSenders if the global limit is reached). Messages can be in both queue LiveBuffer and VolQueueStore and thus occupy the same memory.

IndexCache

To speed up access to the persistent store modules (PersQueueStore and TopicStore) these modules have index caches which hold indexing information for a limited amount of messages. Each entry in the index cache occupies 128bytes.

Queue-Threadpool

In theory, each queue can have an associated thread which processes it. This poses a problem, however, if each queue is reading a big message from the database. To avoid memory problems in this case and to limit the number of threads, a threadpool is used.

Topic-Threadpool

As with queues, topics also have an associated threadpool.

DurableTopicSubscriber-ThreadPool

Similar to the queues, durable topic subscribers can potentially read big messages in parallel causing high memory usage. To control this memory usage the number of simultaneously running durable topic subscribers can be limited. The default number of threads for durable topic subscribers is 4.

DurableTopicSubscriberManager

DurableTopicSubscribers that are replaying messages from the database read a number of messages in one atomic action and then forward these messages to the actual DurableTopicSubscriber in the client. The number of messages can be configured in terms of number and size. When many durable topic subscribers are active the amount of memory used for this reading can be limited globally. The default for this global limit is 2MBytes. Still, reading a single message which is bigger than the limit (which is 256kByte per default) is allowed.

MemWatch

Each java virtual machine (JVM) has a memory garbage collector built in. By default this garbage collector scans for memory that is eligible for collecting every 20-30 seconds. If the JVM is very busy though, the garbage collector may or may not have enough time to work. If big messages are also in use, the server may need a lot of memory in a relatively short amount of time.

To deal with these issues the iBus//MessageServer has a built-in MemWatch service which helps keeping memory usage low. It can be configured in different memory usage levels in which different actions can be taken:

- In the lowest level (0) where memory usage is within 50% of the configured heap size, the MemWatch service does not take any specific action.

- In the level 1 where the memory usage is between 50 and 80% of the configured heap size, the MemWatch service does additional garbage collection every 2 seconds.
- In the highest level garbage collection is done every 750ms plus a warning is issued. If the level drops back to zero after such a warn, a note is issued.

Due to the open design you can change the percentage levels and configure additional levels (or even implement your own levels).

Configuring AMM

This section describes the individual configuration parameters available in the advanced memory management. The parameters can be configured using the administration client.

Socket Reader

`MemWatch.messagethreshold`

defines the minimum amount of memory which can be assigned without checking memory usage - increasing this value speeds up small message handling

Transaction Manager

`TransactionMemSizeManager.uncommittedtxmemsizelimit`

defines the maximum amount of memory used for uncommitted transactions if this limit is reached, then the reduce strategy defines how memory can be freed up.

`TransactionMemSizeManager.txmemsizereducestategy`

defines the strategy on how memory is freed up. The default is `MAXBYTES`, which discards all messages in the largest transaction in bytes. Other parameters include `MAXMESSAGES` (discard all messages in the largest transaction in number of messages), `ALL` (discard all messages in all transactions), `NONE` (discard none), `OLDESTTX` (time dependent, starting with the oldest) and `YOUNGESTTX` (time dependent, starting with the most recent).

`TransactionMemSizeManager.txtypeorder`

defines the order qualifier for the application of the strategy above. The default is `MIXEDFIRSTSTANDARD`, freeing standard transactions

and XA transactions alternatively starting with standard transactions. Other values are `MIXEDFIRSTXA`, similar to `MIXEDFIRSTSTANDARD`, but starting with XA transactions. Also `FIRSTXA`, `FIRSTSTANDARD` which free all XA or standard transactions before the other. `ONLYXA`, `ONLYSTANDARD`, frees only XA or standard transactions.

Storage Module

The parameters for the point-to-point pipeline:

`P2PStore.maxdelayedmessagessize`

defines the maximum amount of memory used in the small processing buffer

`P2PStore.maxdelayedmessagesnumber`

defines the maximum number of messages held in the small processing buffer

`P2PStore.maxdelay`

defines the maximum amount of time (in ms) messages (only non-transactional) are kept in the storage module before being stored on disk.

The parameters for the publish-subscribe pipeline are equivalent but defined in the module

`PubSubStore`

Thread/Buffer-Module

The parameters for the point-to-point pipeline:

`P2PPrioQueue.maxnumbytes`

defines the maximum number of bytes held in the thread/buffer-module

`P2PPrioQueue.maxnummessages`

defines the maximum number of messages held in the thread/buffer-module

The parameters for the publish-subscribe pipeline:

`PubSubPrioQueue.maxnumbytes`

defines the maximum number of bytes held in the thread/buffer-module

`PubSubPrioQueue.maxnummessages`

defines the maximum number of messages held in the thread/buffer-module

Queue/-LiveBuffer

`DestinationManager.globalvolqueuebytehighwatermark`

`DestinationManager.globalpersqueuebytehighwatermark`

define the global upper limit for bytes in the queue livebuffers. There is a separate queue livebuffer for volatile (non persistent) and persistent messages which can be configured individually.

`DestinationManager.volqueuebytehighwatermark`

`DestinationManager.persqueuebytehighwatermark`

define the upper limit for bytes in each queue livebuffer. There is a separate queue livebuffer for volatile (non persistent) and persistent messages which can be configured individually.

Topic/-LiveBuffer

`DestinationManager.globaltopicbytehighwatermark`

`DestinationManager.topicbytehighwatermark`

defines respectively, the global and individual (per topic) upper limit for bytes in the topic livebuffers

VolQueueStore

`VolQueueStoreImpl.globalbytehighwatermark`

defines the global upper limit for bytes reserved for non-persistent queue messages

IndexCache

`TopicFileStore.indexcachesize`

defines the size of the index cache per topic

`PersQueueFileStore.indexcachesize`

defines the size of the index cache per queue

Queue-Threadpool

`QueueThreadPool.maxthreads`

defines the maximum number of threads used for queue processing

Topic-Threadpool

`TopicThreadPool.maxthreads`

defines the maximum number of threads used for topic processing

DurableTopicSubscriber-ThreadPool

`DurableTopicSubscriberThreadPool.maxthreads`

defines the maximum number of threads used for durable topic subscribers during their replay from the database.

DurableTopicSubscriberManager

`DurableTopicSubscriptionManager.globaldursubbytelimit`

defines the maximum number of bytes available for durable topic subscribers replaying messages from the database. Individual messages are still allowed to be bigger than this size.

Default Memory Allocation

This section contains the default allocation of server memory for the individual modules. Note that these parameters might need to be changed for `iBus//MessageServer` to work best with your application. Tuning of these parameters is described in “Performance Tuning” on page 4-1.

The default memory allocation assumes the following parameters:

- a maximum of 100 connections exist with active producers (other connections which only have consumers don't count)
- transactions are small and do not surpass a total of 2MByte in one time
- the number of messages is the same for queues and topics
- the largest message size is 1MBytes

- not more than 3 durable topic subscribers need a replay from the database in one time. This number can be higher but should remain in the area of 10-20.

For the above scenario a typical allocation looks like this:

```
MemWatch.messageThreshold=65535
```

the memory check threshold for incoming messages in the socket readers is 64kBytes

```
DestinationManager.uncommittedtxmemsizelimit=2000000
```

the maximum amount of memory for uncommitted messages is 2MBytes.

```
P2PStore.maxdelayedmessagessize=131072
```

```
PubSubStore.maxdelayedmessagessize=131072
```

the storage optimization buffers contain a max of 128kBytes

```
PersQueueFileStore/TopicFileStore.indexcachesize=1000
```

the stores (which use the embedded database StreamStore) have indexcachesizes of 1000

```
P2PPrioQueue.maxnumbytes=262144
```

```
PubSubPrioQueue.maxnumbytes=262144
```

a maximum of 0.5MBytes is used as a buffer between the socket reader threads and the distribution thread

```
DestinationManager.globaltopicbytehighwatermark=1024000
```

the TopicLiveBuffer is configured to hold a maximum of 1MByte globally.

```
DestinationManager.
```

```
globalpersqueuebytehighwatermark=1024000
```

the persistent QueueLiveBuffer is configured to hold a maximum of 1MByte globally.

```
DestinationManager.
```

```
globalvolqueuebytehighwatermark=1024000
```

the volatile QueueLiveBuffer is configured to hold a maximum of 1MByte globally.

```
VolQueueStoreImpl.globalbytehighwatermark=16000000
```

the volatile queue store is allowed to hold a maximum of approx. 16MBytes of messages in all queues.

```
QueueThreadPool.maxthreads=4
```

with the maximum message size of 1MBytes and 4 threads in this threadpool, 8MBytes will be allocated at maximum for queue message delivery (messages are copied in the Socket Writer).

```
TopicThreadPool.maxthreads=4
```

this restricts the number of running threads to 4

```
DurableTopicSubscriptionManager.
```

```
globaldursubbytelimit = 2097152
```

the maximum amount of memory reserved for `durabletopicsubscribers` during their replay from the database should not exceed 2MBytes. For individual messages that exceed this size, the limit does not apply. If there are messages this big, they are sent directly to the consumer: therefore they only use memory while they are in process. To limit the number of such processes, the threadpool below needs to be configured low enough.

```
DurableTopicSubscriberThreadPool.maxthreads=4
```

with the maximum message size of 1MBytes and 4 threads in this threadpool, 8MBytes will be allocated at maximum during `durabletopicsubscriber` replays (messages are copied in the Socket Writer).

The above configuration results in a theoretical maximum of 46 MBytes. If the server is configured with this limit and the assumptions made in the scenario above do hold, the memory consumption of the server should not substantially exceed 46MBytes: if large messages are sent on topics - e.g. 1MBytes - the server still needs temporary memory during the distribution process. Therefore memory usages of 64MBytes and above is still possible.

Introduction	4-2
What do you want to tune?	4-2
Where to tune	4-4
Default Performance Configurations	4-6
Comprehensive Tuning List	4-6

Introduction

As a matter of course, performance tuning is an extremely wide area. Performance can be improved by using more memory, faster CPUs, higher bandwidth etc. Additionally, the performance of the server improved by optimally tuning its parameters.

Tuning can be done for a variety of scenarios. One can tune a server for handling small messages fast with a very high throughput or for better handling of large message. Note that one tuning can be negative for another scenario. This chapter first covers performance areas and describes how they can be tuned.

What do you want to tune?

Generally speaking, the best application performance can be reached by design. JMS has various parameters which have performance impacts. Knowing these impacts can help you improve the speed by knowing the fastest JMS parameters to choose possible for your application.

The `iBus//MessageServer` allows you to tune for different scenarios. A list of JMS parameters which you might want to keep in mind follows:

DeliveryMode:

Non-persistent messages are generally much faster than persistent messages since they don't need access to the database. Choose non-persistent messages wherever the data is not application critical.

Transactional:

Messages sent on non transacted sessions can be handled faster by the server than in transacted sessions. While transacted messages are guaranteed to be stored only once a commit has successfully returned, non-transacted messages must be stored on arrival. Non-transacted messages are queued up until one of three categories are met: a certain delay is passed (5ms), the number of messages has reached a certain level (5 msgs) or a certain size has been reached (128kByte). Tuning these parameters can greatly increase speed.

Distributed transactions are slower than normal JMS transactions due to the necessary overhead of JTA's transaction demarcation and two phase commit protocol. The client memory required to support distributed transactions is less than for normal transactions because rollback is handled by the server rather than locally. This allows the client to free memory of consumed messages when using distributed transactions.

Transaction size has an impact on performance and typically producing and consuming multiple messages within a single transaction can be more efficiently handled than having a single transaction per message. This performance tuning is not configurable within the server, but is rather a client design issue. Grouping messages into a single transaction increases throughput up to a point when either memory management or flow control becomes necessary. The optimal size can best be determined through empirical testing. For distributed transactions, it is advisable to keep the delay between transaction demarcation and transaction completion to a minimum.

Acknowledge-Mode:

Auto-acknowledge mode needs one server request per message while with client-acknowledge the number of server interactions can be chosen by hand. **Client-Acknowledge** where not every single message is acknowledge **is generally faster** than auto-acknowledge if the application acknowledges, say, every 5th or 10th message.

Message Size:

The theoretical message size limit of the iBus//MessageServer is 1GByte (due to current JVM limitations). If your application needs large messages (large means > 1MByte) performance could be improved by splitting such messages into many small ones which can then be handled faster.

Message Selectors:

While message selectors are a powerful tool to filter messages within the server there are other design aspects which result in faster speed. Such designs include a generally different use of destinations which helps eliminating message selectors, or the use of wildcard subscriptions.

On the other hand, message selectors can help reducing the number of topics or queues. Before relying extensively on message selectors, it is best to benchmark with a prototype.

Number of Connections:

There are practical limits to the number of simultaneous connections which can be supported by the `iBus//MessageServer`. The limits are in the area of 1000-2000 on medium sized machines and 4000-5000 on high-end machines. The maximum number of simultaneous client connections is limited by the number of threads that can be supported in a Java VM. The theoretical limit is 4000 assuming the default stack size of 1MByte per thread. By decreasing the stack size using the `-ss` parameter of the `java` command, this theoretical maximum could be increased to around 5000, but not more. If your application includes a large number of subscribers you might **consider the usage of `iBus//MessageBus`** in combination with the `iBus//MessageServer`. The `iBus//MessageBus` uses **IP multicast** and can handle a large number of connections in the Publish/Subscribe domain. Messages sent via IP multicast are received by all clients simultaneously, with minimal latency.

The following points show parameters for which the `iBus//MessageServer` itself can be tuned:

Throughput versus Latency:

There is a tradeoff between throughput and latency of messages. The default server configuration is tuned for high throughput at the cost of greater latency. If your application requires a small latency of messages consider optimizing this point.

Low Memory versus Speed:

Generally speaking if you have more memory you can have more speed. You can tune how the server allocates available memory between internal components to optimize overall performance.

Where to tune

Throughput versus Delay

There are various buffers in the server which potentially increase delay but also increase throughput. Consider the following modules:

- Storage Module: Small Processing Buffer: used to increase storage speed, this module can be switched off to reduce delay.

- Thread/Buffer Module: Decoupling of incoming and outgoing messages, this module can be switched off to reduce delay.

CPU Time Allocation

The server has many processes which can be configured to run at a very high CPU load, therefore finishing faster versus a lower CPU load and not “disturbing” other traffic running through the server.

- database cleanup process: This process can be configured to run every hour and cleanup the stores. You can change this to run more frequently if your messages have very short time-to-live and are persistent. On the other hand you can increase this value if you have a lot of disk space.

- durable topic subscribers: A durable topic subscriber needs to read data from the stores. As reading single messages doesn't perform well, the message server reads sets of messages into memory. The size of this set can be configured in the module `TopicFileStore/TopicOracleStore`:
`replaymsgamount` and `replaybyteamount`

Memory Allocation

Memory allocation is a broad area with respect to performance tuning. Consider these facts:

- Threadpools: Threadpools help to limit the number of threads and the memory used by these threads. If you have only small messages and a lot of memory you can increase the size of the threadpools.
- Thread/Buffer-Module: This module is used to increase throughput as threads reading from sockets can resume reading sooner when this module is used. To reduce memory usage, this module can be removed.

Default Performance Configurations

The `iBus//MessageServer` is preconfigured to allow high throughput with a small to moderate delay. Also check with the chapter "Advanced Memory Configuration" on page 5-1 where the Configuration Wizard is shown to help with performance tuning.

Comprehensive Tuning List

Following is a comprehensive list of tuning parameters and their impacts on server performance and memory usage. The list is organized according to the modules where the parameters are defined. Note that all configuration parameters can be easily modified using the administration client GUI. Also note that flow control parameters mentioned in "Flow Control" on page 2-1 have a substantial impact on performance as well. Those parameters are not listed here again.

Property name	Description	Implications
Module(s) P2PStore/PubSubStore		
maxdelayedmessagesnumber	maximum number of non transacted messages held in a queue before they are forwarded to the store.	Increase this number to 10-50 to increase non-transacted throughput
maxdelayedmessagessize	maximum amount of bytes in all messages held in that queue (see above)	mainly used to limit memory usage (besides implications above)
maxdelay	maximum time a message spends in that queue (see above)	this defines maximum delay of non-transacted messages in the pipeline
Module(s) P2PProQueue/PubSubPrioQueue		
maxnummessages	maximum number of messages held in this pipeline element, once this queue is filled up, producers are stopped	increase this number to better handle burst of messages
maxnumbytes	maximum amount of bytes in all messages held in this pipeline element (see above)	reduce mainly to limit memory usage (besides implications above)
Module(s) TopicFileStore, PersQueueFileStore, VolQueueStore		
cleanupinterval	interval time between two cleanup cycles in the database	critical parameter, depends on total number of messages in the database
cleanupgaptime	time to sleep between two cleanup transactions during a cleanup process	increase to reduce server load during a cleanup process, decrease to speedup the cleanup process
cleanuptransactionsiz	maximum size of transaction served during a cleanup process, decrease this value to reduce server load during cleanup	increase to speedup the cleanup process, decrease to reduce server load during a cleanup process
Module(s) TopicFileStore, PersQueueFileStore		
cleanuptransactiontime	maximum time spent during a cleanup process	same as above plus gives better control of server load during cleanup process
dbcacherefreshlimit	specifies how many bytes are stored in the database before the database caches are refreshed	good default values are 20-50MBytes, lower values reduces speed

Property name	Description	Implications
Module(s) TopicFileStore		
fastacknowledgesize	sets the optimization value for the number of acknowledges received by clients to be stored in a separate file before indexes are updated	50-100 seems to be optimal, increase to speedup durable subscribers during replay
durableoptimizationinterval	specifies the number of acknowledged messages after which optimization for a specific durable subscriber is done	values between 100-500 seem to be optimal, increase to speedup durable subscribers during replay
replaymsgamount	specifies the maximum number of messages read in at one time during replaying messages for a durable subscriber	50-100 seems to be optimal, increase may speedup durable subscribers during replay
replaybyteamount	specifies the amount of bytes read at one time (see above)	mainly used to limit memory usage (besides implications above)
replaysleep	for increasing parallelism with multiple durable subscribers, a delay can be specified between two transactions (see above)	10-50 seems to be optimal, decrease may speedup few durable subscribers during replay
chkpslottime	maximum time a message-acknowledge command is held in the acknowledge combining queue before being stored	increase speeds up handling of many durable subscribers, reduce (not 0!) to speedup handling of very few durable subscribers
chkpmaxqueuesize	maximum number of message-acknowledge commands held in the acknowledge combining queue mentioned above	same as above
expiryonreplay	defines whether messages found being expired during a replay for a durable subscriber are deleted right away	this can reduce replay speed but improve cleanup time by doing cleanup on the fly.
Module(s) PersQueueFileStore		
fastdeletionsize	sets the optimization value for the number of acknowledges received by clients to be stored in a separate file before indexes are updated	30-100 seems to be optimal, increase to speedup persistent queues
expiryonread	defines whether messages found being expired during a read by a QueueReceiver are deleted right away	this can reduce read speed, but improves cleanup time by doing cleanup on the fly.

Property name	Description	Implications
Module(s) DestinationManager		
persqueuebytehighwatermark	defines the size of the QueueLiveBuffer per queue	increasing this value can speedup queue processing
globalpersqueuebytehighwatermark	defines the global size of all QueueLiveBuffers combined	increasing this value can speedup queue processing
qlbcleanupinterval	interval time between two cleanups of expired and "old" messages (defined by qlbcleanuptimetolive) in the QueueLiveBuffers	qlb cleanup helps keeping memory free for queues with traffic while removing messages from queues with low or no traffic at all.
qlbcleanuptimetolive	maximum amount of time a message is held in the QueueLiveBuffer before being cleaned up (messages will then have to be read from the database)	this time should be higher than the average expected time that a message remains in the server but should remain small to help timeout old (unused) messages from this buffer
tlbcleanupinterval	interval time between two cleanups of expired message in the TopicLiveBuffers	reduce this value if low expiration times are used
topicmsghighwatermark (topicmsglowwatermark) topicbytehighwatermark (topicbytelowwatermark)	defines the size (with respect to number of messages and total size of all messages) of the topic live buffer	increase this value to both speed up topic processing and reduce coupling of producers and consumers (a larger topic live buffer ensures producers are less dependent on the consumer speed.)
Module(s) QueueThreadPool, TopicThreadPool, DurableTopicSubscriberThreadPool		
maxthreads	defines the maximum number of threads per threadpool	modify this value to speedup many destinations/durable subscribers
maxprocessoriterations	defines the maximum number of iterations done by one task in the threadpool at a time	increase this value to speedup processing on high-load
Module(s) XATransactionController		
defaulttimeout	defines the default timeout for distributed transactions if a transaction manager / client does not set explicitly	transactions not completed before the timeout may be implicitly rolled back when memory management limits are reached
consumerQueueEntriesSize consumerQueueByteSize	Number related to the total size of messages held in the consumer queue before flowcontrol.	Increase this number to reduce flowcontrol issues related to performance bottlenecks.

Advanced Memory Configuration

Introduction	5-2
Parameters	5-3

Introduction

This chapter describes the Configuration Wizard's advanced memory configuration. The wizard is started at installation time, but can be started at any time using the configwizard(.bat or .sh) scripts. In the Memory Management screen, the user can enter a few values which are used by the wizard to configure a number of output values relevant for memory management. The screen is shown below:

The screenshot shows the 'Config Wizard' window with the 'Memory Management' section. The window title is 'Config Wizard' and the header includes the 'Softwired' logo. The main title is 'Memory Management'. The form contains the following fields and values:

Field	Value	Description
Server heap size in MB	256	Server heap size in MB
Manual memory configuration	<input checked="" type="checkbox"/>	Manual memory configuration
Estimated number of JMS Topics	10	Estimated number of JMS Topics
Estimated number of JMS Queues	10	Estimated number of JMS Queues
Estimated maximum message size published on any topic, in bytes	20000000	Estimated maximum message size published on any topic, in bytes
Estimated maximum message size published on any queue, in bytes	20000000	Estimated maximum message size published on any queue, in bytes
Average message rate msg/s	50	Average message rate msg/s
Estimated number of client connections (producers and consumers)	20	Estimated number of client connections (producers and consumers)
maximum number of threads	500	maximum number of threads

At the bottom of the window are buttons for 'Cancel', 'Help', '< Previous', and 'Next >'.

The supplied input values help the Configuration Wizard categorize the environment in which the server will run. This includes

characteristics such as: memory available to the server, the use of topics or queues, the message rate and message sizes. Based on the user description of the environment, the Configuration Wizard will alter the configuration and save it to the server *config.xml* file. The resulting configuration is a best heuristic effort and can often be improved with specific parameter changes. This can be done with the Administration Console.

Parameters

Input values.

AMMConfig parameter	Description
Number of JMS topics	The number of JMS topics to which clients will publish or subscribe.
Number of JMS queues	The number of JMS queues to which clients will send or receive.
Maximum topic JMS message size	Size in bytes of the largest topic message that will flow through the server.
Maximum queue JMS message size	Size in bytes of the largest queue message that will flow through the server.
Message rate	The rate, in messages per second, at which messages will flow through the server.
Number of client connections	The number of client connections to the server.
Java process -mx value	The amount of memory available to the java message server process. The memory is distributed among subsystems of the server.
Java process threads	The number of threads that the java message server process will use. The threads are distributed among thread pools for Queues, Topics and Durable Topic Subscribers.

Notes about user input values

Number of topics.

This parameter helps determine the ratio of topics to queues in the JMS environment in which the server is deployed.

Number of queues.

This parameter helps determine the ratio of topics to queues in the JMS environment in which the server is deployed.

Maximum topic message size.

It is possible that each thread in a topic thread pool will receive one message of maximum size, this can lead to $(\text{Number of threads in thread pool} \times \text{Maximum message size}) > \text{Available Memory}$. The Configuration Wizard sets durable topic subscriber thread pool size inversely proportional to this maximum message size to limit the frequency with which large messages lead to a memory overflow.

Maximum queue message size.

It is possible that each thread in a queue thread pool will receive one message of maximum size, this can lead to $(\text{Number of threads in thread pool} \times \text{Maximum message size}) > \text{Available Memory}$. The Configuration Wizard sets queue thread pool size inversely proportional to this maximum message size to limit the frequency with which large messages lead to a memory overflow.

Message Rate.

The bottleneck for high message throughput is the process of persisting the messages. File store index sizes are increased for higher message rates.

Number of client connections.

Each client connection will require a thread. Clients publish and subscribe to topics and queues, a client that connects to multiple queues or multiple topics, or that requests messages from both topics and queues is still only one client connection. A larger number of clients will result in smaller thread pools. The number of java threads must be higher than the number of connections plus 20.

Java threads.

The Configuration Wizard allocated approximately 20 threads for java activities such as garbage collection, finalizer, reference handler and periodic tasks, the rest are allocated to tasks within the server, for instance topic connection and queue connection thread pools. The number that the user enters here is expected to be the maximum number of threads that the operating system can provide to the java process of the message server, some factors affecting this are, number and speed of CPUs, JVM implementation, and OS type. The maximum number of threads is influenced by the settings for the heap size (-mx) and the stack size (-ss). In general decreasing the heap and stack values increases the maximum possible number of threads.

Java memory.

The Configuration Wizard shares out available JVM memory to different subsystems of the server, the amounts which can be allocated, depend on the total available.

Memory allocations can be broadly categorized as:

- Incoming messages area.
- Replay area.
- Live buffers area.
- Queue thread pool area.
- Topic and Durable Topic Subscribers thread pool area.

Not all memory will be allocated since calculations are based on approximately half of the -mx parameter value.

Output configuration parameters.

The set of output parameters from the Configuration Wizard which are relevant for memory management is listed in the table below, each entry shows the config.xml file property group name, the property name and the default value that is assigned to it.

Property Group Name	Property Entry Name	default value
PubSubStore	maxdelayedmessagessize	131072
PersQueueFileStore	indexcachesize	1000
TopicFileStore	indexcachesize	1000
PubSubPrioQueue	maxnumbytes	262144
P2PPrioQueue	maxnumbytes	262144
MemWatch	messagethreshold	65535
ConnectionManager	uncommittedtxmemsizelimit	2000000
VolQueueStore	globalbytehighwatermark	16000000
	globalbytelowwatermark	15000000
DurableTopicSubscriptionManager	globaldursubbytelimit	2097512
DestinationManager	globaltopicbytehighwatermark	1024000
	globaltopicbytelowwatermark	1000000
	globalpersqueuebytehighwatermark	1024000
	globalpersqueuebytelowwatermark	1000000
QueueThreadPool	maxthreads	4

Property Group Name	Property Entry Name	default value
TopicThreadPool	maxthreads	4
DurableTopicSubscriberThread-Pool	maxthreads	4