Benchmark Comparison of Messaging Throughput in Enterprise Messaging Systems using the Java Message Service API

Tim Fox, Messaging Lead, JBoss by Red Hat

Clebert Suconic, Core Messaging Engineer, JBoss by Red Hat

Abstract

A simple set of messaging throughput benchmarks were defined and run against systems implementing the Java Message Service (JMS) API. The benchmarks were chosen to cover the most common messaging use cases, including both lightweight publish/subscribe messaging and persistent point-to-point messaging. Default configuration for each system was used unless the vendor specifically recommended particular tunings for performance in their documentation, or the vendor's default configuration settings did not provide JMS specification compliance. In the majority of the use cases, HornetQ out-performed all other systems, out-performing the next best performing system by a factor of up to 2.5

Introduction

Enterprise Messaging Systems

The term *enterprise messaging system* is used throughout this paper. We define this as follows:

Enterprise messaging systems are general purpose messaging workhorses with a large feature set which usually possess high end features such as clustering and high availability. Often they support their own protocols, and the overwhelming majority support the Java Message Service (JMS) standard API.

Apart from the enterprise messaging systems, there are other messaging systems that focus on low-latency messaging – common in the financial services industry, where time sensitivity, e.g. for financial instrument prices may be very high.

Typically, though not exclusively, these products have a narrow and specialised feature set and often lack high end features, allowing them to concentrate on providing the lowest latency deterministic messaging. Compared to the wider messaging arena, this low-latency market is a niche area, with those implementations that lack enterprise capabilities prevented from being suitable for general purpose messaging applications. Therefore, we do not include them in this analysis.

This paper only concerns itself with the messaging throughput of enterprise messaging systems that implement the JMS API.

Experimental set-up

Physical set-up

Three physical nodes were used for each benchmark. Each node had the following specification:

Model: IBM 3650

CPU: Quad core Intel Xeon 2.5Ghz

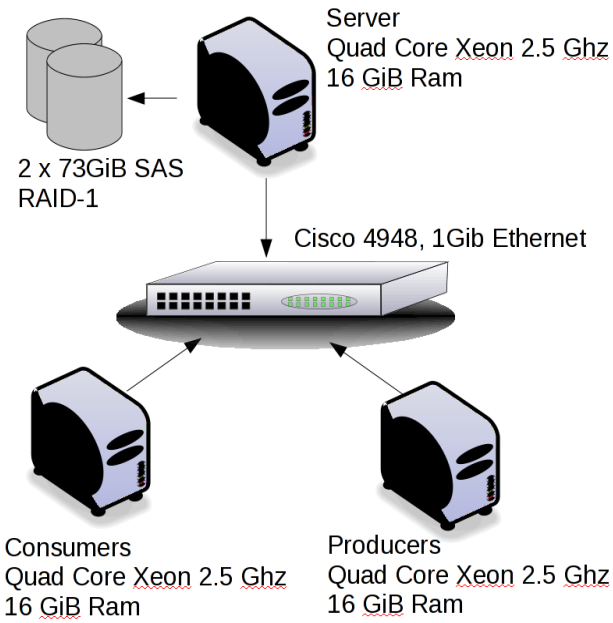
RAM: 16GiB (typically only a fraction of this RAM was used by the processes)

Disk: 2 x 73GiB local SAS drives configured as RAID-0 (striping)

Each node was connected via 1 Gib/s ethernet via a Cisco 4948 switch. The network was completely isolated.

We disabled disk write cache on the disks so we could be sure that after a file sync is executed or a completion is received for asynchronous IO, the data has really been persisted to physical storage, not just written into the disk write cache.

Illustration 1: System set up



Messaging System Versions

The following versions of the messaging systems were used in the measurements. Where possible these were the latest versions of the systems available at the time the measurements were taken.

HornetQ 2.1.1 final

ActiveMQ 5.3.2 GA

SwiftMQ 7.6

OpenMQ 4.4

Operating System and JVM

All nodes were running Red Hat Enterprise Linux 5.3 and Sun JDK 1.6.0-19 (64 bit).

Experimental Method

The benchmark

The Test Harness

The Sonic test harness (http://communities.progress.com/pcom/docs/DOC-29828) was used for the benchmarks.

This is a simple and easy to use benchmarking tool that can be configured for many JMS uses cases and allows JMS parameters such as message size, durability of messages, number of consumers, number of producers to be configured.

The Sonic test harness has previously been used on other published JMS benchmarks, so is already familiar to the messaging user and developer communities.

We made a small number of modifications and rebuilt the test harness from source to remove any dependencies on SonicMQ specific libraries.

We also made a small number of changes to address some issues:

1) The test harness used Java integers to count messages during the run. Due to the very high throughputs obtained in some runs this was insufficient and caused integer overflow. The code was therefore changed to use Java longs to prevent any overflow.
2) SwiftMQ disallowed the '/' character in a JMS client id string. The test harness uses the '/' character so would not work with SwiftMQ. A small change was made so the test harness used the '-' character instead.

Messaging Throughput

All benchmarks consist of a single messaging server, a number of message producers and a number of message consumers.

The messaging server is installed on it's own physical node, all the message producers are on a single second node, and all message consumers are on a single third node.

Messages are sent at as high a rate as the system will sustain.

The benchmarks measure mean *messaging throughput* as seen by the messaging consumers during the duration of the test run. If there are multiple message consumers then any figures shown are the *total* messages consumed by the consumers during the run. E.g. in the case of 10 consumers who each consume 1000 messages during a run, the throughput figure reported would be 10 x 1000 = 10000 messages per second.

Duration of run

Each benchmark run was performed for a duration of 1000 seconds (16 minutes and 40 seconds). We believe this run time was sufficiently long to minimise indeterminacy in the results due to factors such as garbage collection, Just-In-Time compilation etc.

Benchmark Scenarios

The benchmark sent and consumed messages as fast as possible while measuring the total throughput of messages under different circumstances where we alternate these variables:

- Number of Producers

- Number of Consumers
- Size of the message
- Type of Destination: Queue / Topic
- Durability of Messages Persistent / Non Persistent
- Transacted / Non Transacted
- Elements per transaction

The benchmarks scenarios were chosen to cover the most common messaging use cases, including both lightweight publish/subscribe messaging and persistent point-to-point messaging.

The set of benchmarks are not intended to be an exhaustive set that probes deeply into all corners of each systems performance characteristics. They are intended to provide an easy to digest summary of messaging throughput for the most common use cases.

For lightweight publish/subscribe messaging we used a small 12 byte message size. 12 bytes is sufficient to convey useful information (e.g. a stock symbol id + price) but sufficiently small to measure performance characteristics for small messages.

We also measured publish/subscribe messaging for 1kiB messages.

For point-to-point persistent messaging we chose a message size of 1kiB

It should be noted, the message size quoted here represents the size of the JMS message *body*. In general a JMS message will be much larger than this due to the set of headers and other information conveyed in the message by the particular provider. The actual total size of the message is implementation specific.

No optimisations such as disabling message IDs or timestamps were configured.

In the case of non transacted consumers, AUTO_ACKNOWLEDGE mode was used throughout.

The scenarios measured are as follows:

- Scenario A – Non persistent 12 byte messages, 1 producer, 1 consumer, Topic
- Scenario B - Non persistent 12 byte messages, 1 producer, 15 consumers, Topic
- Scenario C - Non persistent 12 byte messages, 15 producers, 15 consumers, Topic
- Scenario D - Non persistent 1kiB messages, 1 producer, 1 consumer, Topic
- Scenario E - Non persistent 1kiB messages, 1 producer, 15 consumers, Topic
- Scenario F - Non persistent 1kiB messages, 15 producers, 15 consumers, Topic
- Scenario G - Persistent 1kiB messages, non transacted, 40 producers, 40 consumers, 40 queues
- Scenario H - Persistent 1kiB messages, transacted, 10 messages per transaction, 40 producers, 40 consumers, 40 queues

Publish / Subscribe – Small messages (12 bytes)

Scenario A

Single publisher, single subscriber on a topic with small messages. Which represents the simplest publish/subscribe scenario possible.

- Number of publishers = 1
- Number of subscribers = 1
- Size of the message body = 12 bytes
- Topic
- Non persistent messages
- Non transacted

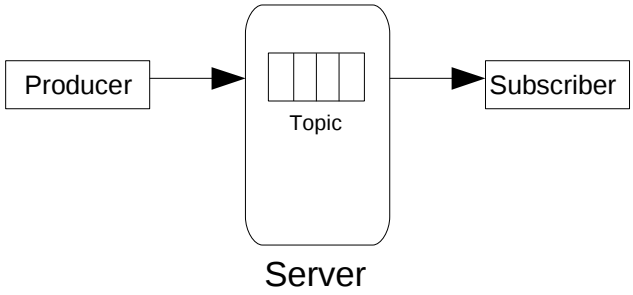


Illustration 2: Scenario A

Chart

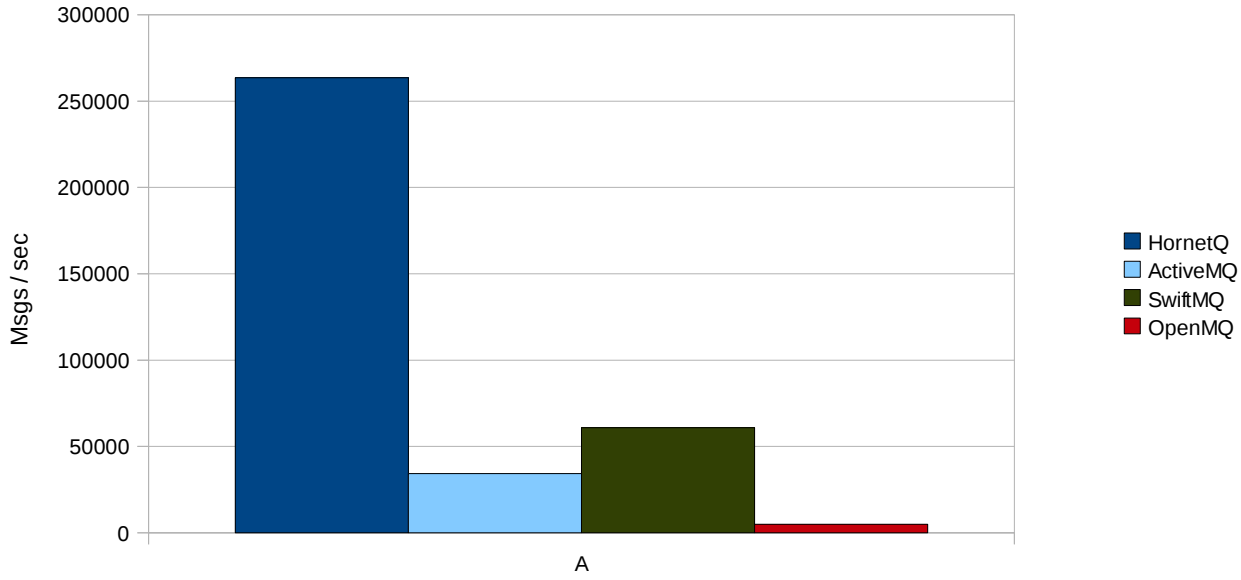


Illustration 3: Benchmark A, Non persistent 12 byte messages, 1 producer, 1 consumer, Topic

Scenario B

Single publisher, 15 subscribers on a topic with small message. Each message produced is consumer by all 15 subscribers. This will measure the capability of the server on handling multiple references of thousands of messages per second.

- Number of publishers = 1
- Number of subscribers = 15
- Size of the message = 12 bytes
- Topic
- Non persistent messages
- Non transacted

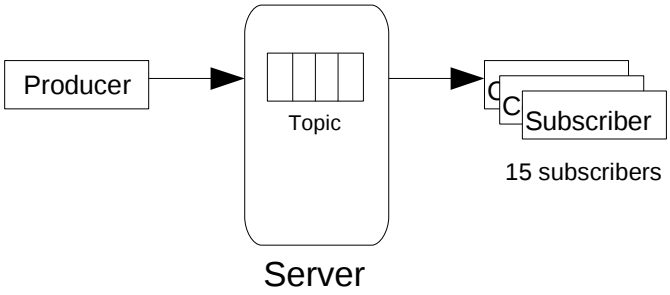


Illustration 4: Scenario B

Chart

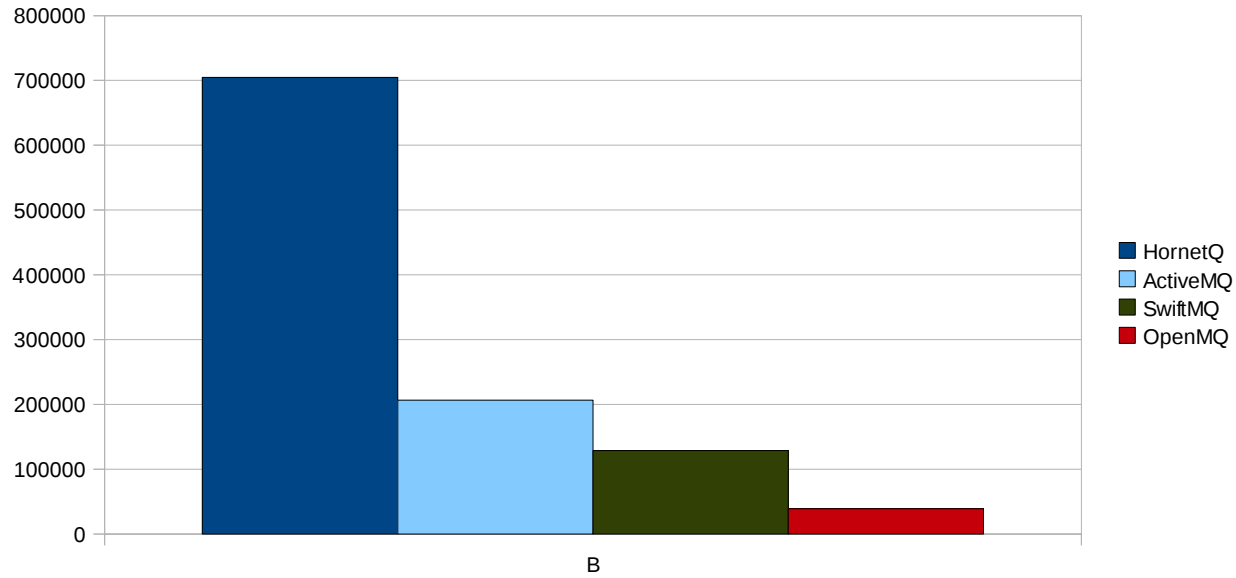


Illustration 5: Benchmark B, Non persistent 12 byte messages, 1 producer, 15 consumers, topic

Scenario C

15 publishers, 15 subscribers on a topic with small messages. Each message produced is consumer by all 15 subscribers.

- Number of publishers = 15
- Number of subscribers = 15
- Size of the message = 12 bytes
- Topic
- Non persistent messages
- Non transacted

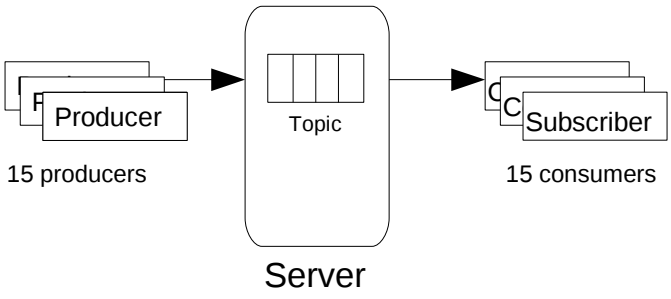


Illustration 6: Scenario C

Chart

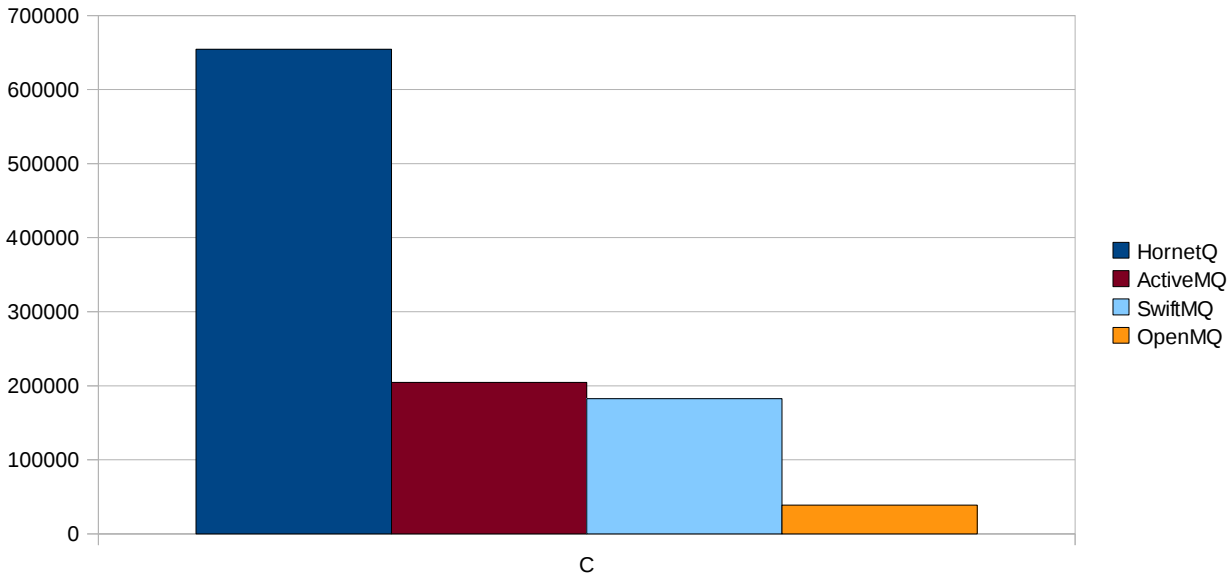


Illustration 7: Benchmark C, Non persistent 12 byte messages, 15 producers, 15 consumers

Publish / Subscribe – 1kiB messages

Scenario D

Same as Scenario A, with 1kiB messages

- Number of publishers = 1
- Number of subscribers = 1
- Size of the message = 1kiB bytes
- Topic
- Non persistent messages
- Non transacted

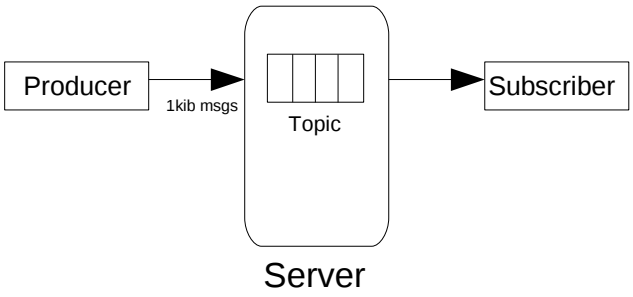


Illustration 8: Scenario D

Chart

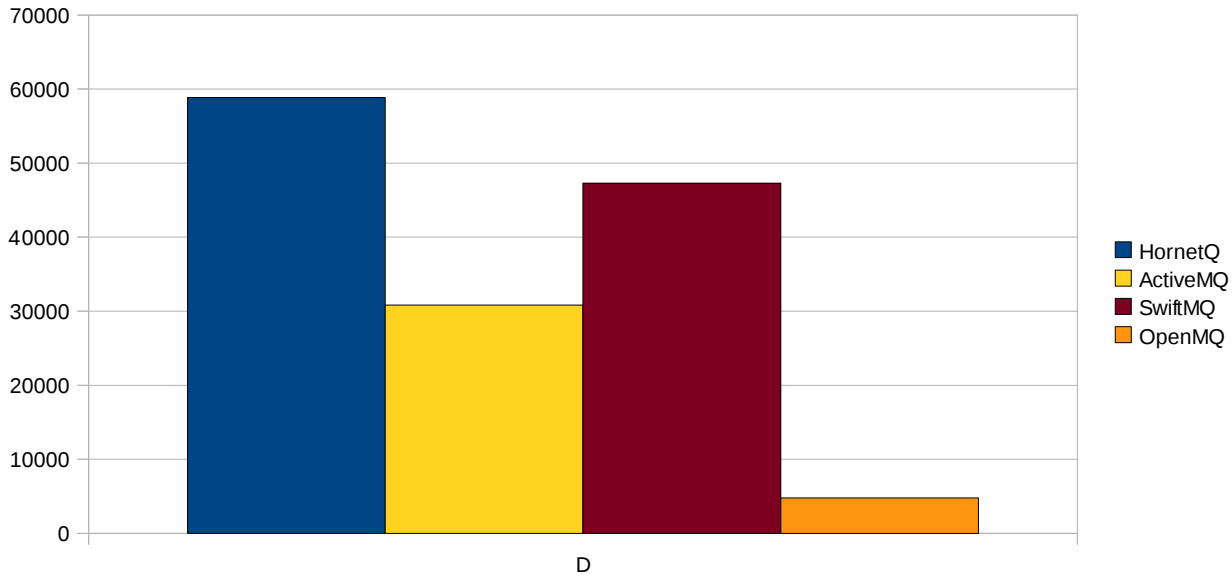


Illustration 9: Benchmark D, Non persistent 1kiB messages, 1 producer, 1 consumer

Scenario E

Same as Scenario B, with 1kiB message

- Number of publishers = 1
- Number of subscribers = 15
- Size of the message = 1kiB bytes
- Topic
- Non persistent messages
- Non transacted

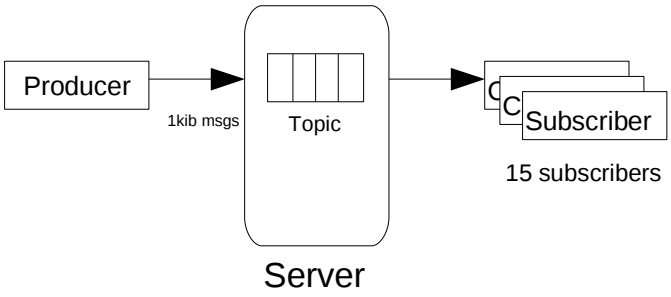


Illustration 10: Scenario E

Chart

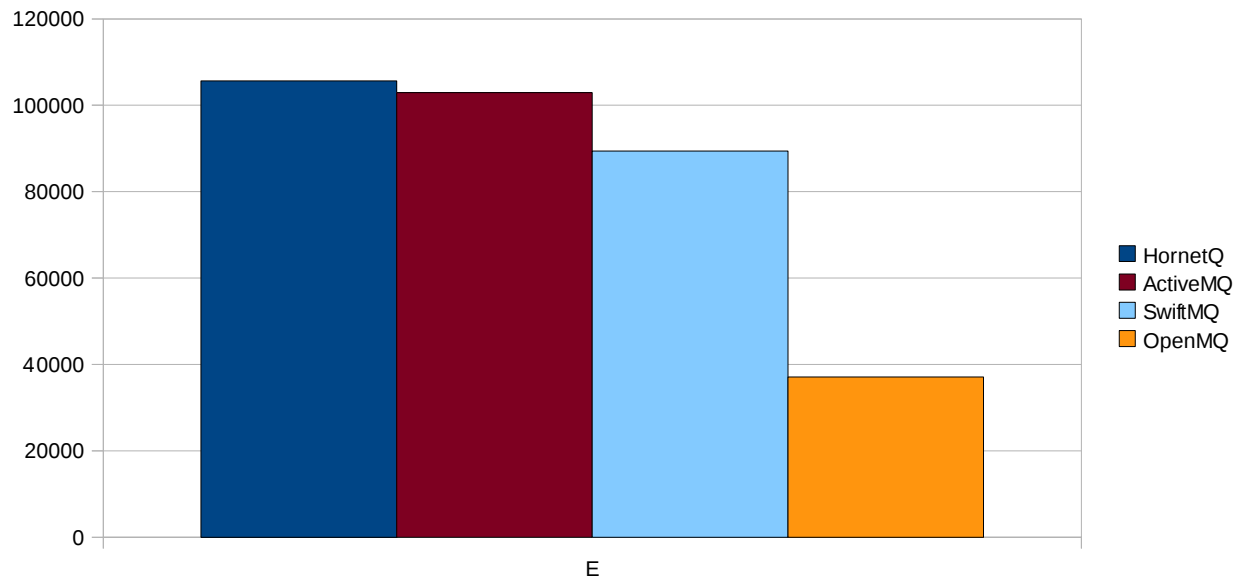


Illustration 11: Benchmark E, Non persistent 1kiB messages, 1 producer, 15 consumers

Observations

- The network was being saturated on this test for HornetQ, ActiveMQ and SwiftMQ, so we have similar figures around the 100 k messages / sec figure for these systems. It would be interesting to see how the systems compared with a faster network, e.g. 10 Gib/s in a future version of this benchmarking.

Scenario F

Same as Scenario C, with 1kiB message

- Number of publishers = 15
- Number of subscribers = 15
- Size of the message = 1kiB bytes
- Topic
- Non persistent messages
- Non transacted

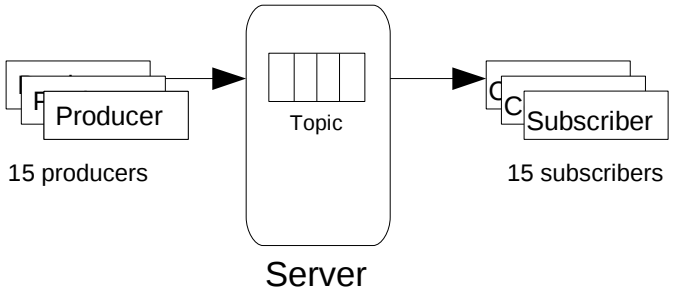


Illustration 12: Scenario F

Chart

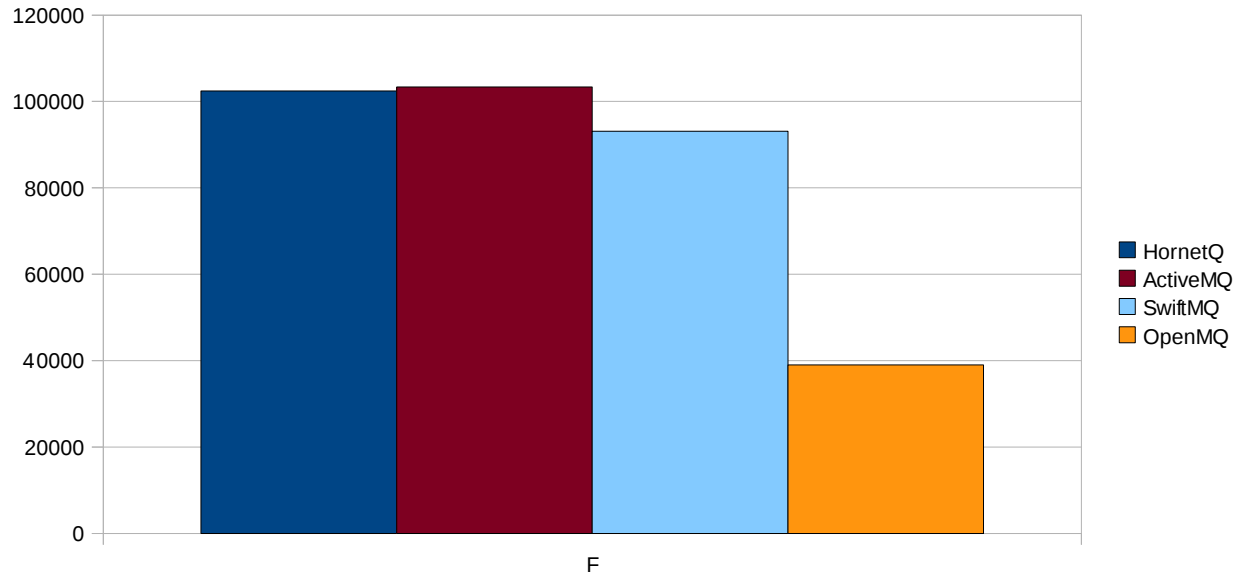


Illustration 13: Benchmark F, Non persistent 1kiB messages, 15 producers, 15 consumers

Observations

- The network was being saturated on this test for HornetQ, ActiveMQ and SwiftMQ, so we have similar figures around the 100 k messages / sec figure for these systems. It would be interesting to see how the systems compared with a faster network, e.g. 10 Gib/s in a future version of this benchmarking.

Persistent Scenarios

Scenario G

40 producers and 40 consumers over 40 different queues sending persistent messages non transactionally. This should generate enough load to stress the server up to its limits including the persistent storage. Each producer / consumer pair are using a different queue. This tests the systems ability to scale horizontally with multiple concurrent writes.

To be JMS specification compliant each JMS persistent message send should not return until the message has been sent to the server and physically persisted to storage. On a pure Java server for example this will require a sync to be performed (e.g. Channel.force())

In their default configurations some providers are known to relax JMS specification compliance, and message durability in order to benefit performance. For the benchmarks we made sure these systems were configured to be specification compliant.

Unfortunately some systems, to the best of our knowledge, do not allow themselves to be configured to always sync to disk or send persistent messages synchronously. So any results obtained with them have been disallowed as their behaviour could be considered “cheating”.

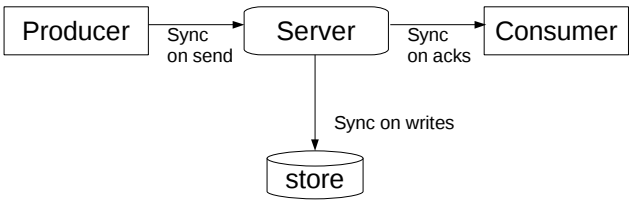


Illustration 14: Syncs on persistence

For well behaved systems, that means an individual producer shouldn't ever exceed the capacity of physical writes of the disk (the slowest part of the equation on the hardware on syncs). However the system can be well optimized to scale up or batch multiple writes in a single sync.

The system we used is capable of doing 250 physical writes / second.

- Number of producers = 40
- Number of consumers = 40
- Size of the message = 1kiB bytes
- 40 different queues
- Persistent messages
- Non transacted

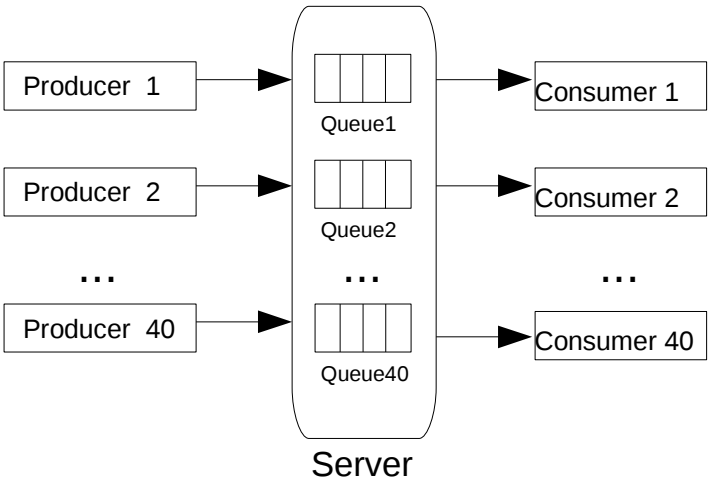


Illustration Scenario G

Chart

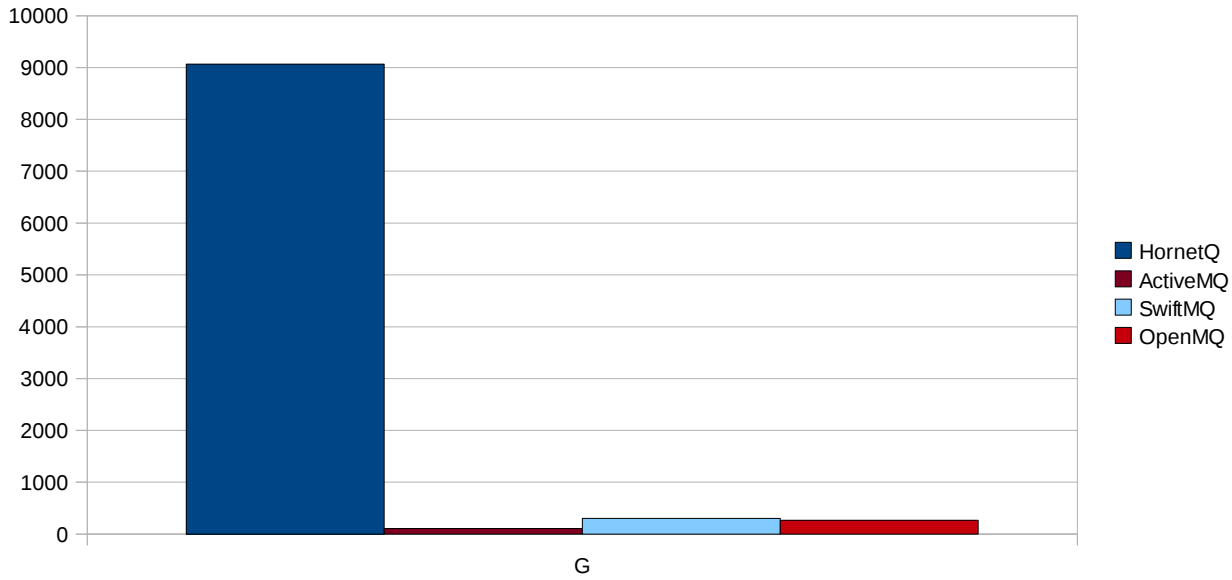


Illustration 15: Benchmark G, Persistent 1kiB messages, non transacted, 40 producers, 40 consumers, 40 queues

Scenario H

Same as scenario G, but using transacted sessions while sending & acknowledging 10 messages per transaction.

- Number of producers = 40
- Number of consumers = 40
- Size of the message = 1kiB bytes
- 40 different queues
- Persistent messages
- Transacted (10 messages / acknowledgements per transaction)

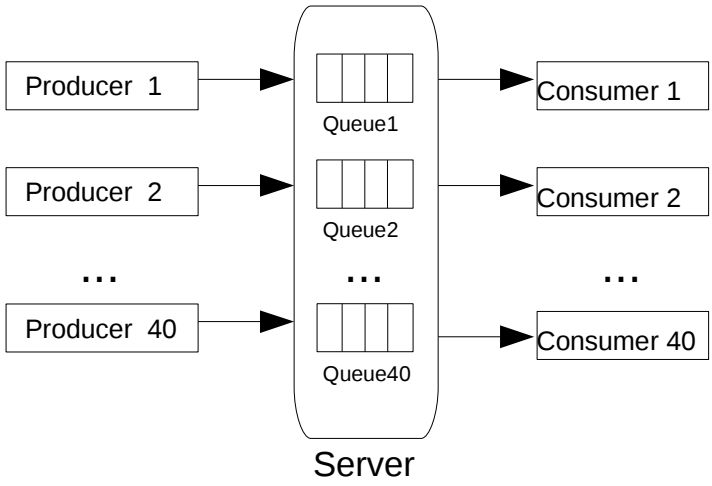


Illustration 16: Scenario H

Chart

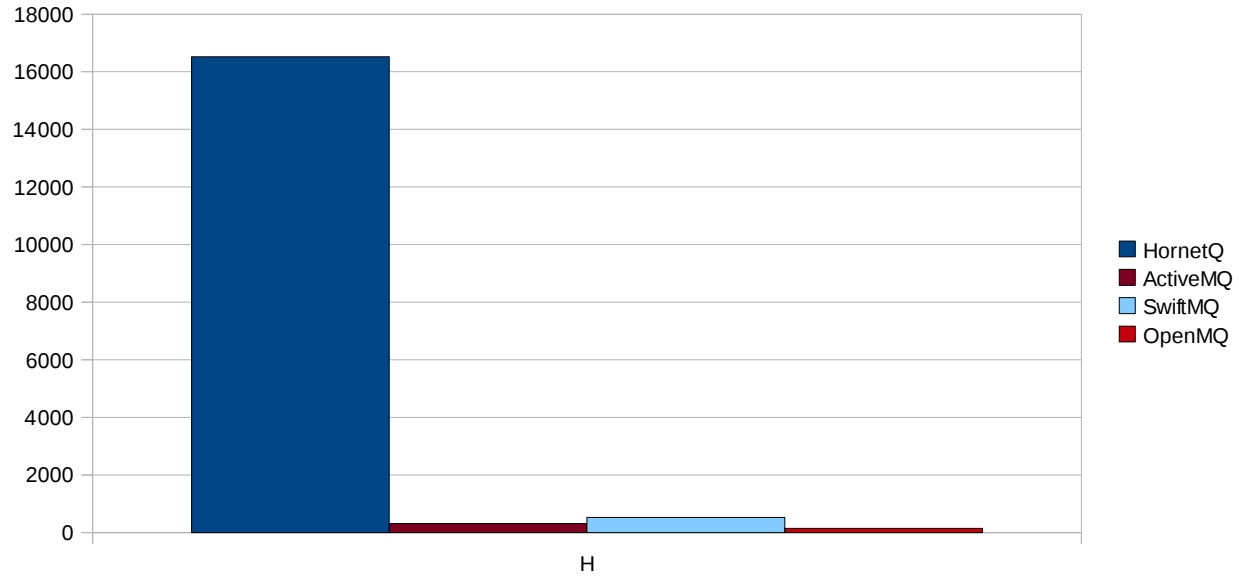


Illustration 17: Benchmark H, Persistent 1kiB messages, transacted, 10 messages per transaction, 40 producers, 40 consumers, 40 queues

Conclusions

For lightweight publish / subscribe messaging with 12 byte messages, there was a very wide range of results with HornetQ as the clear leader.

The network appeared to be saturated for publish / subscribe messaging with larger 1 kiB messages, what gave us similar results around the 100 k messages/sec mark. It would be interesting to see how much higher results would go with a faster 10 Gib/s network in a future work.

For persistent messaging, there was also a wide range of results again with HornetQ as the performance leader.

The results clearly demonstrate that HornetQ is the most performant enterprise messaging solution among the systems tested.

Appendix 1: Configuration changes

Default configuration for each system was used unless the vendor specifically recommended particularly tunings for performance in their documentation, or the vendor's default configuration settings did not provide JMS specification compliance.

ActiveMQ

- Renamed activemq-throughput.xml as activemq.xml
- set enableJournalDiskSyncs="true" on KahaDB

HornetQ

- Set journal-min-files = 100, Used ThroughputConnectionFactory for the publish/subscribe tests

OpenMQ

- Flow control and maxSize = 5000 on queues and topics
- sync on store
  - by adding imq.persist.file.sync.enabled=true at /open-mq/var/instances/imqbroker/props/config.properties

SwiftMQ

- Sync set to true on the storage Swiftlet
- Group commit delays = 1 ms
  - We tried several different values until we could achieve best performance possible on the persistence cases.
  - We tried for instance 10 ms and 100 ms on the group commit delay but that actually made the results worse even though we expected a better throughput.
- Increased number of available threads in thread pools to give better throughput

Appendix 2: Raw data

All the data is represented in messages / second. Throughput calculated at the consumers.

System	A	B	C	D	E	F	G	H
HornetQ 2.1.1	263454	704642	654395	58849	105650	102436	9067	16523
ActiveMQ 5.3.2	34240	206714	204599	30845	102918	103336	107	317
SwiftMQ 7.6	60843	128806	182785	47309	89389	93104	301	529
OpenMQ 4.4	4915	39160	38764	4782	37092	39003	268	147

Appendix 3: Scenario Summary

Metric	Destination Type	Durability	Msg Size	Producers	Consumers	ACK Mode
A	Topic	NP	12 bytes	1	1	Auto
B	Topic	NP	12 bytes	1	15	Auto
C	Topic	NP	12 bytes	15	15	Auto
D	Topic	NP	1 kiB	1	1	Auto
E	Topic	NP	1 kiB	1	15	Auto
F	Topic	NP	1 kiB	15	15	Auto
G	40 Queues	P	1 kiB	40	40	Auto
H	40 Queues	P	1 kiB	40	40	TX (10 msgs)

P = Persistent

NP = Non Persistent