Fiorano

Enabling change at the speed of thought

www.fiorano.com

An Implementation Analysis of JMS Servers

Flow Control, Latency, Scalability, and Persistence

AMERICA'S
Fiorano Software, Inc.
718 University Avenue Suite 212, Los Gatos,
CA 95032 USA
Tel: +1 408 354 3210
Fax: +1 408 354 0846
Toll-Free: +1 800 663 3621
Email: info@fiorano.com

EMEA
Fiorano Software Ltd.
3000 Hillswood Drive Hillswood Business Park Chertsey Surrey
KT16 0RS UK
Tel: +44 (0) 1932 895005
Fax: +44 (0) 1932 325413
Email: info_uk@fiorano.com

APAC
Fiorano Software Pte. Ltd.
Level 42, Suntec Tower Three 8
Temasek Boulevard 038988
Singapore
Tel: +65 68292234
Fax: +65 68292235
Email: info_asiapac@fiorano.com

Executive Summary

This document provides an in-depth technical analysis of FioranoMQ, the leading Java messaging server on the market, and SonicMQ, another relatively popular JMS Server. We compare the flow control, scalability, latency-reduction and persistence mechanisms used with the two server implementations, and illustrate particular conditions in which one of the servers Progress SonicMQ is rendered useless in important real-world circumstances.

FioranoMQ

As the first vendor to implement the JMS specification and over 300 customers including Morgan Stanley, JP Morgan, AT&T, FedEx, and Motorola, Fiorano Software is the leader in providing high performance standards-based messaging infrastructure. FioranoMQ implements all of the key requirements for a messaging server, including clustering and load balancing, key security features (ACL/ACE, SSL), integration with application servers, LDAP servers, and NT/Unix Realms in addition to support for C/C++ clients XML Messages types and SOAP. Visit the Fiorano web site for a comprehensive description of the features and benefits of FioranoMQ.

SonicMQ

SonicMQ, from Sonic Software, provides a hub-spoke implementation of JMS pub/sub as well as point-point domains written entirely in Java. SonicMQ also provides for JMS extensions like XML messages and server clustering.

Flow Control Analysis – FioranoMQ and SonicMQ

The ability of a messaging server to deliver messages at a constant rage (regardless of publisher speeds and the number of connections to the server) depends to a large extent on flow-control algorithms employed by them. In a typical messaging environment, message producers are usually faster than the consumers of the message. To ensure that capacity limits (memory, threads, etc.) within the server are not exceeded, the sending clients must be throttled to prevent the loss of messages.

A particularly important aspect of any flow control algorithm in a publish/subscribe messaging server is to ensure that if a particular subscriber slows down, other subscribers on the same Topic are not adversely affected. That is, a single slow subscriber should not slow down the entire system. This is a particularly difficult feature to implement in an industrial strength messaging server.

FioranoMQ Flow Control

FioranoMQ flow-control provides for effective flow-control using an "exponential-backoff" algorithm to throttle publishers when internal queues are full, coupled with message buffering algorithms and varying internal queue sizes. Message buffering increases the throughput of the system by ensuring that messages are delivered to consumers from buffered queues. FioranoMQ gives Administrators the added control to increase internal buffer limits, the size of which determines the exact extent of Publisher throttling. Publishers are throttled on a topic-specific basis, when one or more internal buffers belonging to those topics exceeds a certain threshold;

however, the algorithm ensures that on or more slow subscribers on any Topic do not affect the speed of message delivery to other subscribers.

SonicMQ Flow Control and associated problems

SonicMQ also buffers messages internally and throttles publishers when internal buffers overflow. Unfortunately, the SonicMQ flow-control algorithm only seems to work under "lab condition" benchmark runs; it suffers from the following severe problems under real-world scenarios:

SonicMQ Slows Down all Publishers to the Speed of the Slowest Consumer

A simple test, wherein one creates two or more subscribers on a given topic and a single publisher on that topic, illustrates that if one of the subscribers is slowed down (for instance, by placing a sleep (5000) call within callback function), the message receive rates of all subscribers slow down to the speed of the slowest subscriber. This also reduces the speed of the publisher and the overall throughput of the system. A single subscriber that blocks for a long time (say 100 ms) can therefore severely slow down the entire messaging system, making the use of SonicMQ software in real-time applications extremely risky.

As an example, consider the following "real-world" case, where all subscribers are consuming messages with different consumption rates. Assume we have one publisher, say P, which continuously publishes messages into the MQ server in BURST mode, an good example of which is a GUI thread that publishes 100 messages of size 1KB each time a button is clicked. In such an instance, the overall publish rate might be 10 msg/sec, but the "burst mode" rate will be substantially higher.

Assume that we also have two subscribers set up as follows: one subscriber (S1) consuming messages at a rate of 100 msgs per second, and another subscriber (S2) consuming messages at a rate of 10 msgs per second.

The expected behavior with such a setup is for the MQ server to cache the messages published during the burst and deliver messages to S1 at a rate of 100 msgs/sec and to S2 at a rate 10 msgs/sec without blocking the Publisher for the complete duration of the send cycle.

FioranoMQ performs exactly as expected, while some JMS implementations (like that of SonicMQ) which do not implement any paging mechanism in server, simply block the publisher while subscribers are picking up all the messages. In these cases, flow control is simply – "blocking the publishers till subscribers catch up", which works well for generating performance results, but lead to serious issues while using these products in a real world application. Think of the complexities an application programmer has to deal with if the time taken for a single publish call can take from a few milliseconds to 30 minutes! For example, if you have an instant messaging application that publishes a message each time the "send" button is pressed, then writing the application logic is almost impossible if the

publish call can take up to 30 minutes (depending on the load previously put on the MQ Server to which the application is connected). This can lead to severe problems in An Implementation Comparison of JMS Servers even simple applications including GUI applications because the GUI will not be able to repaint itself for 30 minutes in such an instance (due to the blocking publish call).

A fundamental question to ask for each JMS implementation is: Can we define the maximum time a publish call will take to publish a single message? If such a time cannot be defined, then the implementation will likely have severe flow-control problems. You can download a SonicMQ JMS Application from http://www.fiorano.com and verify the above results in greater detail. Please read the instructions in readme.txt for notes on compiling and running these samples. While running the tests, please bear in mind that the send() call can sometimes never return.

Failure of SonicMQ Distributed Transactions

A SonicMQ Distributed Transaction fails under the following circumstances: if the Publish call is part of a distributed transaction and the call blocks forever, then the DTC (Distributed Transaction Coordinator) would either time-out this transaction or report a state of ambiguity. FioranoMQ avoids the above pitfalls and follows JMS semantics. If the internal buffers overflow, the appropriate publishers are immediately throttled using an exponential backoff algorithm, and non persistent messages are dropped if the internal Queues consistently overflow; persistent messages are never lost and are constantly logged in the offline datastore.

True Push-based Delivery

FioranoMQ uses a true server side push model for delivering messages to subscribers, unlike most other messaging vendors who employ continuous polling (timed wait in some cases) from the runtime to the JMS server. Each FioranoMQ client “registers” its interests in different topics/queues (with corresponding message selectors) in the FioranoMQ server. The FioranoMQ server in turn is responsible for delivering (pushing out) messages to the clients when a message matching their specification is published. The above architecture further adds the benefit of using a single thread per connection as against two threads per JMS connection (as employed by most other JMS vendors).

An advanced flow control mechanism is built in between the FioranoMQ runtime and FioranoMQ server to ensure that runtime buffers are always filled to practical limits, without causing memory problems.

Using Pre-fetching for Queues

FioranoMQ does not perform pre-fetching for queues by default. Perfecting is implemented as a configurable option. Some vendors like Sonic MQ implement pre-fetching by default, resulting in misleading performance results. Some applications that are not aware of pre-fetching under the covers can run into potentially serious issues at the deployment stage,

when some messages might not be delivered to any other receiver because they are queued in a single receiver (which might have crashed/hung due to potential application failure). This results in messages not getting processed even while some queue receivers are waiting for messages all the time.

Scalability Analysis

All Java messaging systems, including FioranoMQ and SonicMQ, which are implemented in 100% pure Java, are best suited for a maximum of up to approximately 2,000 concurrent client connections depending on message size, application type, and hardware. Existing Java messaging vendors may claim they offer high scalability but due to the inherent reasons described below they will not scale beyond a fixed point.

Achieving high scalability requires server resources to remain constant as client connections are added. Current systems however rely on a primitive architecture. For example, SonicMQ server resources (threads) are consumed just waiting for incoming data, adding unnecessary overhead.

Furthermore, as each additional client connects, a new thread needs to be allocated on the server, leading to linearly increasing server load. Under such conditions the server eventually slows down to unacceptable levels or crashes. In typical cases, allocating more than 2000 threads is impractical on a single JVM, although the precise limits vary depending on the hardware and operating system platforms used.

Fiorano's Connection Management architecture overcomes this inherent scalability problem by allowing a pluggable, Scalable Connection Management (SCM) module, which keeps server resources constant regardless of the number of concurrent client connections on the server. Using Fiorano SCM one can, for instance, use a constant thread pool of 300 threads to monitor over 5,000 client connections, and allocate active resources only to those clients that are actually sending and receiving data. Internal testing reveals that a single instance of FioranoMQ Server using SCM, can scale more than 5000 concurrent client connections.

Fiorano SCM is implemented via native code, using efficient C-Select constructs, a time-tested server-scalability development technique familiar to thousands of experienced developers. Until such time as the Java programming system adds support for C-Select type constructs, there is no way of producing a truly scalable 100% Java Server implementation. It should be noted that Fiorano’s default settings invoke a pure Java implementation, which is similar in nature to that of SonicMQ and other popular JMS Servers on the market today.

The advanced implementation of SCM provides much higher scalability, handling more than 5K concurrent client connections in a single instance of the FioranoMQ server without leading to any substantial performance degradation when a low number of clients (say 2) are connected to the server. Advanced thread context switching mechanisms enable optimal performance for both low bandwidth and high bandwidth clients.

Advanced "Three Thread-Pools" Architecture

The Fiorano MQ server employs a three-pool architecture for achieving maximum scalability and performance in real-world applications. A pool of threads (the Reader Thread Pool) is employed to receive messages and control information from FioranoMQ clients. All calls on the socket are non blocking and are guaranteed to finish within a specified time within the FioranoMQ server (releasing the thread immediately while data is being written on the TCP stack). This allows a fixed number of threads to be employed for a potentially large number of connected clients.

Message distribution (based on Message selection and CBR – content based routing) is implemented within the context of a second thread pool (the Demux Thread Pool), responsible for delivering messages to those subscribers who match the specified criterion. Demux Worker Threads maintain delivery order of messages published by each publisher, without introducing a system wide serialization of delivery or message selection. Outbound messages are added to delivery queues for a "controlled push" to their corresponding clients.

Delivery of messages from delivery queues to clients is handled by the third thread pool (the Delivery Thread Pool) which picks up messages from the flow controlled delivery queues and writes them to the corresponding PUSH sockets opened for the subscribers. Delivery queues are flow controlled with the FioranoMQ runtime library, ensuring that the runtime is never overloaded with memory (and has enough streaming to keep it busy all the time.) Fiorano implements a dynamic load-balancing algorithm for Thread Pool implementation, whose limits are configurable using the server configuration file (server.cfg).

Latency Analysis

In many real world applications, especially in real-time systems, there are strict requirements governing the delay between the time a message is published and the time it is received by one or more subscribers. Latency is a critical factor for measuring performance along with overall throughput of the system.

Some vendors publish results for server throughput and latency using independent tests (in order to show their product in best light in those cases). This leads to an entirely different performance matrix at deployment time when the final application performs much worse when compared to the original latency-test results. As such, any tests used for An Implementation Comparison of JMS Servers latency performance analysis should be as close to real deployment requirements as possible. This means that the test should measure both performance as well as latency of messages concurrently, as they are routed through a JMS server.

Tests show that the average latency for FioranoMQ software is of the order of 20 to 25 ms, while for JMS Vendors like IBMMQSeries/ SonicMQ the numbers are typically over an order of magnitude higher. This makes the use of IBM-MQSeries/SonicMQ in real-time environments

(such as stock quote systems in investment banks, or telecommunications switches) virtually impossible. To check the results for yourself, please visit http://www.fiorano.com.

- **Machine config** PIII 733 MHz, 386 RAM, Single CPU, Win2K
- **Test Scenerio** 1 Non Persistent Publisher, 1 Non Durable Subscriber, Non transacted sessions,
  FioranoMQ latency 22 ms
  SonicMQ latency: < order of magnitude higher > – Sonic Software does not allow its results to be published for the benefit of end-users and developers

Persistent Storage and Caching

FioranoMQ uses a file-based data store to log persistent messages. Fiorano key performance edge in the real world stems from its use of a highly optimized file-based data-store for the transient storage of persistent messages as mandated by the JMS standard. The use of a file-based data store allows FioranoMQ to deliver messages between 10 and 15 times faster than SonicMQ, which uses a JDBC compliant databases to deliver messages. Fiorano is unique among JMS vendors in its use of a file based data-store. All persistent messages are efficiently logged onto the file-based store, and durable subscriptions use sophisticated caching techniques to pick up data from separate sections of the store concurrently.

Having understood the obvious advantages of a file-based data store, SonicMQ claims in some analyst reports to use a file-based data store to store “in-flight” messages. In reality, SonicMQ uses Cloudscape, which is a Java-based database management system and not a file-based data store.

Syncing data to the file system

FioranoMQ "syncs" up with the underlying file system after every operation performed on the file based data store. FioranoMQ implements crash protection and recovery using advanced indexing and locking mechanisms within its file-based DB layer. A message once persisted is synced to disk before the publish call returns in an application.

Conclusion

The above mention test results reveal the performance advantage that Fiorano’s file based data-store architecture provides. Don’t be fooled by other vendor’s performance comparisons. We invite you to judge the performance results for yourself by downloading the performance test source code from the Fiorano web site.

About Fiorano Software

Fiorano Software (www.fiorano.com) is a leading provider of enterprise class business process integration and messaging infrastructure technology. Fiorano's network-centric solutions set a new paradigm in ROI, performance, interoperability and scalability. Global leaders including Fortune 500 companies such as Boeing, British Telecom, Credit Agricole Titres, Lockheed Martin, NASA, POSCO, Qwest Communications, Schlumberger and Vodafone among others have used Fiorano technology to deploy their enterprise nervous systems.