Persistent, Reliable JMS Messaging Integrated Into Voyager's Distributed Application Platform

By Ron Hough

Abstract

Voyager Messaging is an implementation of the Sun JMS 1.0.2b specification, based on Voyager, Recursion Software's distributed computing platform. The following provides an introduction to messaging middleware and explores two of its common applications. The following also illustrates the function and value of three key features of Voyager Messaging: shared persistence architecture, client persistence, and message batching.

Overview of Messaging

Messaging middleware allows applications and systems to exchange data and events of various types in an asynchronous fashion. The applications exchange messages in much the same way as people exchange email. Messaging software is valuable because it allows dissimilar systems on different points in the network to communicate with each other, without necessitating code changes to those systems. For example, a company may have an application which tracks product stock in the warehouse. They may also have an online order-processing system obtained from a different vendor. Out of the box, these two systems may be unable to communicate with each other, and any exchange of data would be done manually or with custom-developed software. Using messaging middleware, however, it would be possible for the order-processing system to send a message to the warehouse requesting product shipment whenever an order is received.

Messaging Applications: Enterprise Application Integration (EAI)

The most widespread deployment of messaging systems is for the purpose of "enterprise application integration." This enables communication between different pre-existing applications within an enterprise.

- Technology changes very rapidly; and while large investments in information technology infrastructure are made by corporations, parts of that infrastructure often become outdated due to an inability to interact with newer systems. Voyager Messaging allows the existing software investment to be preserved, while making it possible to exchange data with other, often newer, products.
- Companies can rarely obtain all the software they need from a single vendor. As a result, applications from different (and sometimes competing) sources must be made to work together in order for a company to achieve its business goals. Voyager Messaging can be made to interface with virtually any software vendor's product, thus facilitating efficient communication among all of a company's systems.
- Mergers and acquisitions generally mean that parallel legacy systems used by the parties involved (such as payroll or account management applications) will have to be consolidated. Adapters, which are Voyager Messaging clients, can be created to enable information exchange between parallel systems.

Messaging Applications: Business-to-Business (B2B)

With the emergence of the modern Internet, it has become feasible for companies to cooperate in ways that were not previously possible. Using messaging middleware, businesses can now interact with each other without having to tightly couple their respective systems. This allows organizations to explore new relationships and improves the workflow of existing relationships. For example, a retail outlet that stocks a product from a particular vendor may have a warehousing system that receives a message from the vendor whenever the price of the product changes. If it drops below a certain level, an order could be automatically generated and sent to the supplier.

The JMS Specification

Sun's Java Message Service (JMS) defines a standard interface through which Java programs can send and receive enterprise messages. A number of industry leaders cooperated to create the JMS spec. By providing messaging implementations that comply with the JMS specification, vendors can ensure that their products will be usable by any application that is compatible with the JMS API. Voyager Messaging implements the JMS 1.0.2b specification finalized by Sun in June, 2001.

Voyager Messaging Architecture

The Voyager Messaging JMS implementation consists of five primary components:

- The **message** is created using the JMS API and is one of several defined types, such as text, object, byte, etc.
- The **sender** (also called a **producer)** is a JMS client that creates and sends messages.
- The **receiver** (also called a **consumer)** is a JMS client that has registered itself to accept messages.
- The **broker** is the entity responsible for ensuring the delivery of messages from senders to receivers. In JMS, the broker is either a Topic or a Queue, depending on the messaging model. A server process must be started to manage Topics and Queues.
- A **database** is an optional component used to provide persistent messages and durable subscriptions. These are "quality of service" features that ensure the eventual delivery of messages even when clients are not connected to the network.

These components working together allow applications to use senders to transmit messages through brokers to receivers in other applications. The JMS specification defines two possible models (also called domains) for message exchange: **publish-subscribe** and **point-to-point.**

In the **publish-subscribe model,** one producer sends messages to multiple consumers by way of a virtual channel called a Topic. Message consumers "subscribe" to the Topic, indicating that they wish to receive messages. Producers send messages to the Topic, which are delivered to all subscribed consumers. This model could be demonstrated by a system that provides stock price information to multiple clients. Clients subscribe to a particular stock and receive notifications whenever the price of that stock changes.

In contrast, the **point-to-point** model is used to deliver messages to one-and-only-one consumer. Instead of subscribing to a Topic, message consumers register with a Queue. When a producer sends a message to the Queue, it will be delivered to only one of the registered consumers. This model could be demonstrated by a stock trading system that distributes incoming trade requests to a group of brokers for processing. The brokers register with the system when they are available to process trades. When a trade comes in, it is sent to only one broker (to avoid possible duplication), although any of the available brokers have the same opportunity of being the recipient.

In both models, the message delivery is asynchronous, meaning that the producer continues execution immediately after sending the message, rather than having to wait for the message to be received. The Voyager Messaging server ensures delivery of the message by managing the Topics and Queues.

The **database** component is optional, and is not directly required by the JMS specification. However, JMS does define an interface for persistent messages and durable subscriptions, which require some form of message storage. This may take the form of proprietary files, a relational database, etc. Voyager Messaging supports the usage of a JDBC-compliant database for message storage.

Persistent message delivery means that the JMS server saves the message in permanent data storage before a message is delivered to a receiver. If the server fails while attempting to deliver the message, it can be recovered from storage and redelivered when the server is brought back online. Due to the additional overhead involved in saving a message, persistence typically incurs a performance penalty. However, Voyager Messaging's *staged persistence architecture* helps minimize this performance impact. Staged persistence is discussed in greater detail below.

In addition to persistence, *durable subscriptions* are another important JMS feature necessary for guaranteed message delivery. A receiver that is subscribed to a Topic may be identified as "durable." This means that if it disconnects from the server, any unexpired incoming messages will be stored by the server and will be redelivered when the receiver reconnects.

A full exposition of JMS and how it is used is not included here, although some familiarity with the specification is assumed. For further information, including a tutorial with examples, consult Sun's Java website (http://java.sun.com).

Voyager Messaging Features

Voyager Messaging is based on the award-winning Voyager ORB (object request broker). Since messaging is, at its heart, a specialized form of distributed computing, the ORB is an ideal framework for a JMS implementation. The Voyager ORB version 4.6 is an established, industry-leading, distributed computing platform. It has been in widespread use since its 1.0 release in 1997, and has been time-tested in thousands of deployments - including many Fortune 500 companies. Utilizing its technology, Voyager Messaging provides a stable, robust, and powerful messaging system with a set of advanced features,
including staged persistence architecture, client persistence, and message batching. These quality-of
service features can all be custom-configured, either through Voyager Messaging's GUI administration tool or XML property files.

Voyager Messaging Features: Staged Persistence Architecture

Voyager Messaging uses a Staged Persistence Architecture to persist messages and acknowledgments. The architecture combines the performance benefits of a proprietary file-based persistence mechanism with the robustness of a JDBC-compatible database.

Most companies have made a significant investment in one or more database technologies for their enterprise applications. The Staged Persistence Architecture's use of JDBC allows an administrator to configure Voyager Messaging to take advantage of that investment. If Voyager Messaging relied on JDBC alone, the overhead of handling many fine-grained transactions (one for each message and/or acknowledgment) would severely impact performance. The Staged Persistence Architecture uses log files and batch transactions to eliminate such overhead.

The Staged Persistence Architecture persists messages and acknowledgments in three stages. In the first stage, messages and acknowledgments are written to log files. In the second stage, the log files are archived. In the third and final stage, archived log files are processed and the database is updated. Voyager Messaging provides default values for all configurable properties of the Staged Persistence Architecture. Administrators may configure the provider to use any JDBC data source.

Stage 1 - Writing to the Log Files
In the first persistence stage, Voyager Messaging writes persistent messages and acknowledgments to proprietary log files. If a provider is heavily loaded, using a single log file may cause performance congestion. To eliminate this, Voyager Messaging may be configured to alternately write data (messages and acknowledgments) to several log files. Rotating between several log files minimizes the time spent waiting on any single log file. By default, Voyager Messaging uses two log files for messages and two log files for acknowledgments.

Stage 2 - Archiving the Log Files
In the second persistence stage, Voyager Messaging archives the log files, which are stored in the same directory as the active log file(s). Log files are archived for the following reasons:

- The size of the log file(s) is greater than the maximum specified size.
- The elapsed archive time is greater than the maximum specified time.

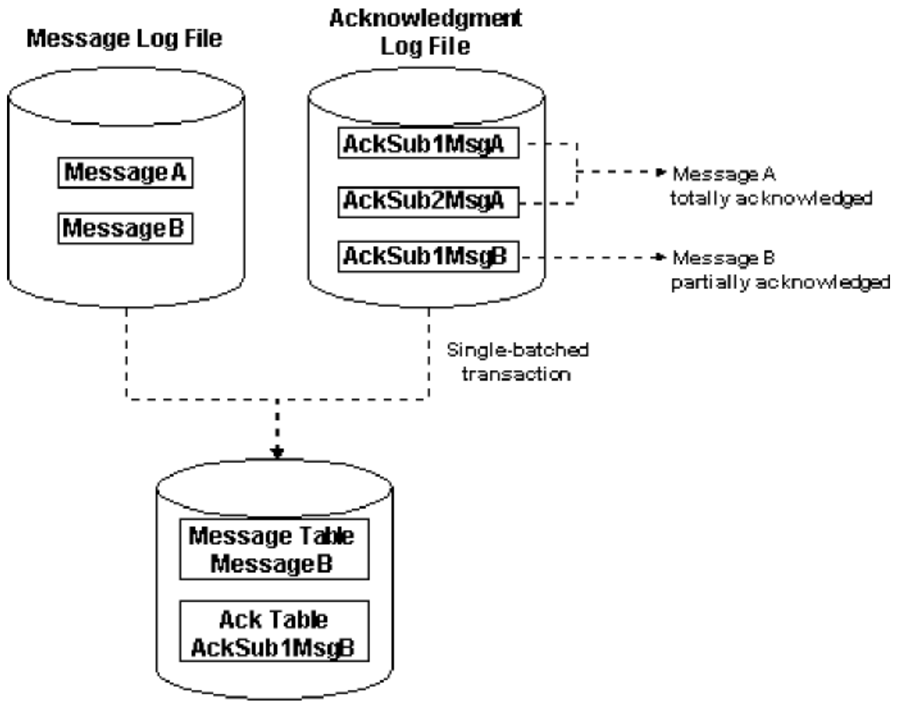
By default, the maximum size of the log file(s) is 100 MB and the maximum time between archiving the log file(s) is one week. Both of these values are configurable.

Stage 3 - Processing the Archived Log Files
In the third and final persistence stage, Voyager Messaging processes the archived log files. Processing archived log files synchronizes the database with the current state of the system. Such synchronization implies that a database backup will reflect the state of the system as of the most recent time the archived log files were processed. Archived log files are processed (and subsequently deleted) for the same reasons that log files are archived (see above). By default, the maximum size of the archived log files is 100 MB and the maximum time between processing the archived log file(s) is one week. Both of these values are configurable.

Example Scenario
To illustrate how the Staged Persistence Architecture minimizes the number of database transactions, as well as the amount of data written to the database, consider the following scenario. An application has two durable subscribers interested in receiving messages published to the *sports* topic. Two messages are published to the sports topic,

`MessageA` and `MessageB`. Both messages are delivered to both subscribers. Both subscribers acknowledge `MessageA`, while only the first subscriber acknowledges `MessageB`. The log files are archived and processed. Since both subscribers acknowledged `MessageA`, it is not necessary to write `MessageA` or any of its acknowledgments to the database. Since only the first subscriber acknowledged `MessageB, MessageB` and its single acknowledgment are written to the database.

Figure 1. Staged Persistence Architecture



Without log files, five database transactions would be required: one for each message and one for each acknowledgment. By using log files however, it is only necessary to write one message and one acknowledgment to the database using a single, batched transaction. In the example scenario, the Staged Persistence Architecture results in an 80% reduction in the number of database transactions (from five to one), a 50% reduction in the number of messages written to the database (from two to one) and a 66% reduction in the number of acknowledgments written to the database (from three to one).

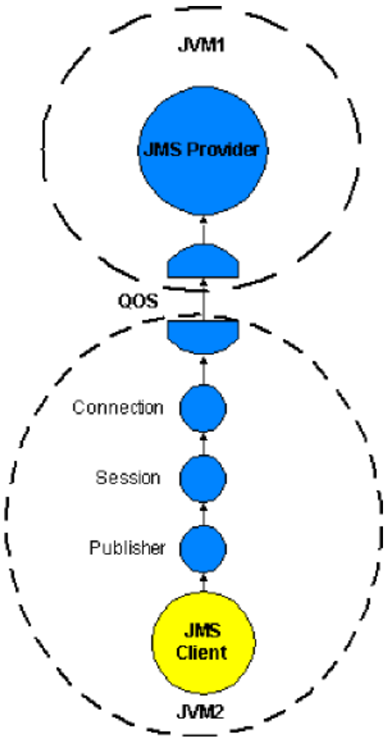
Voyager Messaging Features: Client Persistence

Client persistence allows a client to continue publishing messages even if a fault causes the server to become temporarily unavailable. If the server becomes available while the client is still running, client persistence automatically reestablishes the connection between the client and the server. Any messages published by the client while the server is unavailable are automatically redelivered.

The following scenario describes how client persistence provides fault tolerance for a publishing client:

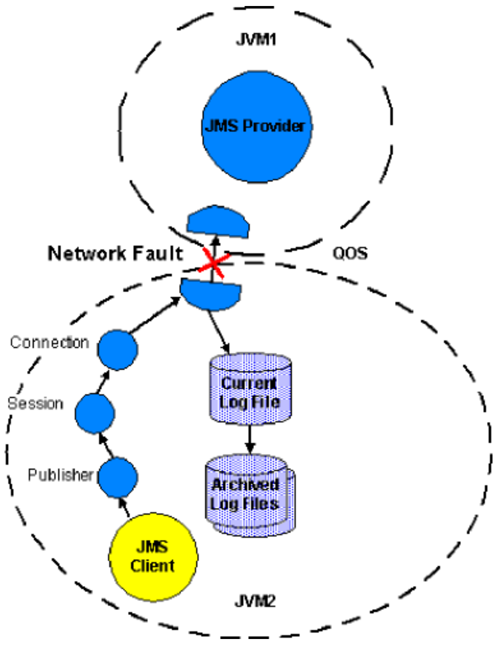
1. A client uses Java Naming and Directory Interface (JNDI) to lookup a connection factory that is configured for client persistence. The client creates a connection, a session and a publisher.
2. The client publishes messages. The messages are asynchronously sent to the provider (see Figure 2).

Figure 2. Publishing before network fault



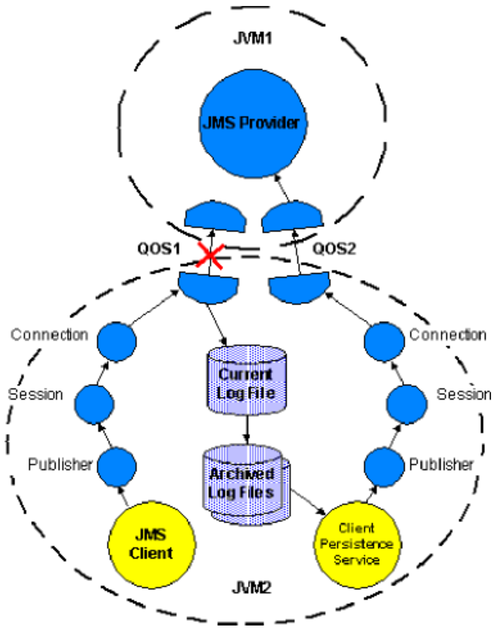
3. The provider becomes unavailable. The client persistence service writes all of the messages to a local log file. Periodically, the client persistence service checks to see if the provider has become available. As long as the provider remains unavailable, the messages are written to the log file. Periodically, the client persistence service archives the log file, Figure 3.

Figure 3. Publishing during network fault



4. The client continues publishing messages. The messages are asynchronously sent to the provider. An exception is thrown and the connection's exception listener is notified.
5. The provider becomes available.
6. The client persistence service detects that the provider is once again available. The service uses a private connection factory (not available via JNDI) to create a connection, session and publisher. The private connection factory is configured specifically for publishing messages while a client is reconnecting to a provider. The client persistence service archives the current log file and begins publishing the archived messages to the provider. After the client persistence service has published all of the archived messages, it looks for new messages in the current log file. In order to preserve message order, the client persistence service is still handling messages sent by the client via its original publishing connection. The original publishing connection cannot be reestablished with the provider until the backlog of messages is delivered. After the client persistence service has published all logged messages to the provider, it reconnects the original publishing connection to the provider. If the client continues publishing messages on the reestablished connection, those messages may be received by a subscriber prior to the last messages sent by the client persistence service (see Figure 4).

Figure 4. Publishing after network fault



7. If the provider does not become available while the client process is running, the logged messages will be delivered by the next client that creates a connection with a topic connection factory configured for client persistence and uses the same log directory. Typically, messages are delivered the next time the original client is started.

The following scenario describes how client persistence provides fault tolerance for a subscribing client:

1. A client uses JNDI to look up a topic connection factory configured for client persistence. The client creates a connection, a session and a subscriber. The client begins receiving messages.
2. The provider becomes unavailable.
3. The client attempts to acknowledge receipt of a message. If the client application has set a connection exception listener, the listener is notified that the provider is unavailable. The client persistence service

queries the provider's availability. The client persistence service does not write acknowledgments to a local log file.

4. The provider becomes available.
5. The client persistence service detects the provider is available. The client persistence service reconnects the original subscription. The client continues receiving messages.

Voyager Messaging Features: Message Batching

Each JMS application has unique messaging requirements. Some applications publish small messages and others publish large ones. Some applications publish many messages and others publish only a few.
Not only does each application have unique messaging requirements, each client within an application has unique messaging requirements. Message batching allows an administrator to define multiple topic connection factories, each offering a different message flow quality-of-service. A client uses the topic connection factory that provides a message batching quality-of-service which best satisfies its messaging requirements.

Message batching is primarily used to optimize the performance of an application. Batching messages causes messages to be delivered in groups, which improves performance by minimizing the number of network calls. Messages sent by a publisher are batched in a message queue running on the client. Messages delivered to a subscriber are batched in a message queue running on the provider.

A publisher can send messages either synchronously or asynchronously. Messages sent synchronously are sent one-at-a-time on the client's thread, while messages sent asynchronously are batched and sent as a group on a JMS-managed thread. A configurable batch interval specifies the time between batch sends. Since the message-ordering guarantee defined by JMS only applies to messages published with the same delivery mode, it is possible to independently specify the message batching quality of service for persistent and nonpersistent messages.

There are several issues associated with sending messages asynchronously. If a message is sent to an unknown destination, for example, the client receives immediate feedback that the message was not sent. If the message is sent asynchronously, an exception is not immediately thrown and control successfully returns to the client. Subsequently, an exception is thrown after the batch of messages is sent, which is reported to the connection's exception listener. If the client application has no set connection exception listener, it will be unaware that the message was not successfully published. If the message is sent synchronously, the provider throws an exception to the client.

Another example is message producer sending a persistent message to a valid destination. If the message is sent asynchronously, the client is not guaranteed that when control returns that the provider will have persisted the message. However, if the message is sent synchronously, the client is guaranteed that when control returns that the provider will have persisted the message.

Messages delivered to a subscriber are always batched. A configurable batch interval specifies the time between batch receives. It is not possible to independently specify the message batching quality of service for sending persistent or non-persistent messages.

Extended Messaging through Voyager ORB

In addition to the messaging capabilities defined in the JMS specification, the ORB on which Voyager Messaging is based supports a number of other powerful features. Messages can be sent as direct invocations on object methods, both synchronously or asynchronously. Messaging via the ORB can take advantage of the Voyager Security module, which supports advanced encryption and policy management. It also is compatible with Voyager Transactions, which supports transactions in a distributed environment across heterogeneous resources and allows participants to be distributed across multiple hosts and/or virtual machines. For additional information on

Voyager ORB, see the Voyager ORB white paper or the documentation packaged with the ORB on www.recursionsw.com.

Conclusion

The introduction of the JMS standard opens many doors for both vendors and users of messaging systems. Voyager Messaging is a feature-rich, cost-effective JMS implementation built on a proven distributed computing platform. Features such as staged persistence architecture, client persistence, and message batching enable the deployment of an efficient, reliable messaging solution.

About the author

Ron Hough is a senior software engineer at Recursion Software, Inc. He may be contacted by email at engineer@recursionsw.com.

Recursion Software, Inc.
2591 North Dallas Parkway, Suite 200
Frisco, Texas 75034
1.800.727.8674 or 972.731.8800
www.recursionsw.com