U1 TECHNOLOGIES

"Oils ain't oils."

'Valvoline and the Art of Evaluating JMS Messaging Engines'

A U1 Technologies White Paper

April 2009

Executive Summary

I remember as a kid seeing that Valvoline television commercial for the first time, where a nice looking, gray-haired auto repairman looks confidently into the camera and tells me with absolute authority that “oils ain’t oils”. Valvoline, you see, was different than those other lubricants on the market. Castrol, STP, Pennzoil, Quaker State…no comparison.

Even as a 10-year old, I was suspicious about this bold marketing claim. All of this oil comes out of the same ground, right? And different companies put it into their own cans, and they put their own labels on those cans – so how different could the oil really be? The truth is, for most purposes, there wasn’t much differentiation amongst oils back in the 70s, and there isn’t much today either. Oils *are* (in fact) oils.

Let’s now fast forward into the 21st century, and into the world of high tech software, and more specifically, into the world of computer-to-computer communications, right down to JMS, the Java Message Service (how’s that for a segue?).

I often hear people say that “JMS messaging is a commodity”, that “the world doesn’t need another messaging engine”, and yes, I even hear that “messaging is messaging”.

JMS 1.1, which was last updated in April 2002, is a specification designed by a group of engineers and business people to help ensure a standard way of sending and receiving data between business applications. Because it is a standard, one could – and too often does – make the argument that JMS is therefore an undifferentiated commodity.

However, I’m sitting here today (slightly graying, and looking confidently into a virtual camera) asserting that “*messaging ain’t messaging*”, and “*JMS ain’t JMS*”. But rather than just stating it confidently and fading to black, like the commercial, we’ll use this white paper to describe how implementations can be different, and to give you a framework for evaluating the differences between JMS implementations. In addition, we’ll also give you some tools for gathering empirical data to support such “bold marketing claims”.

The premise is quite simple. A standards-based specification, such as JMS, is only a blueprint – the difference lies in how that blueprint gets translated into a working system. And in how well the implementation has been battle-tested in the real world.

For JMS implementations, the differences can be seen primarily across three tangible, measurable metrics:

- **Performance** – how fast can messages get delivered from publishers (the *producer* applications) to the subscribers (the entire set of *consumers*)?
- **Scalability** – how effectively can the JMS implementation scale across two dimensions:
    - horizontally (eg., deploy globally)
    - vertically (eg., support more users, more messages, more applications, more data, etc.)
- **Reliability** – how certain can we be that all interested parties receive the critical information that is being sent?

Each of these facets is important, and each has a real impact on the quality, effectiveness, and in some cases the viability of the applications using the messaging system (consider the viability of a stock trading application where requests to buy or sell *sometimes* make it to their destination, and where *pretty good* performance is good enough).

In addition, each also has a real impact on the economics of the implementation (e.g., how many servers are required – which translates into hardware, licenses, management, administration, energy usage, licenses, etc?).

Despite their importance, these three metrics are also at odds with each other. When one optimizes their JMS server to increase reliability, the trade-off invariably comes at the expense of performance and scalability. Conversely, optimizing for performance and scalability typically means less reliable delivery of messages to interested parties.

The question we asked ourselves over a decade ago was, is it possible to deliver all three simultaneously? The short answer is – yes absolutely – because we have done it. Our messaging engine powers global stock trading applications for over 30 name-brand financial institutions. And the reason our technology is being used, versus those of the "big boys" (Tibco, IBM, Sonic, etc.) is that our solution, AmbrosiaMQ, delivers higher performance and scalability without sacrificing reliable delivery. If your order to buy or sell arrives a few milliseconds later than your competitor's, it costs you real money – every time you trade.

This white paper discusses some of the unique ways in which U1 has implemented its JMS solution, as well as some extensions to the JMS specification that are at the core of addressing the competing facets of performance, scalability and reliability.

> *Authors caveat – this document assumes a basic working knowledge of JMS and messaging systems; we start from this base and work forward. For general information on JMS, please refer to http://en.wikipedia.org/wiki/Java_Message_Service*

JMS – Performance, Scalability & Reliability?

This white paper is comprised of three parts, each of which focuses on the three competing facets of performance, scalability and reliability – hereafter referred to as "the three facets".

- Section 1 – Base-Level Capabilities – describes JMS 1.1-specific capabilities that have been implemented in a unique way within AmbrosiaMQ to address the three facets.
- Section 2 – JMS Extensions – describes additional APIs that have been exposed by AmbrosiaMQ to address limitations within JMS 1.1, specifically to address the three facets.
- Section 3 – Testing & Measurement – describes an open source toolkit provided by U1 Technologies that can be used to test JMS implementations from any vendor to measure the differences across each of the three facets; the toolkit was designed so that it could easily be extended by the community to include additional areas of testing and measurement.

Section 1 – Base-Level Capabilities

The JMS 1.1 specification outlines and describes a set of features and capabilities that one must support to adhere to the standard. JMS 1.1, like most standards specifications, is not prescriptive in *how* to implement these features and capabilities. And it is, in fact, the *how* that is so critical in determining the quality of the implementation, and thus, the differentiation between solutions from different vendors.

This section describes some of the unique ways in which U1 Technologies implemented core APIs of the JMS 1.1 specification.

(a) Flow Control

Flow control is a feature designed to address the "weakest link" problem within messaging systems. If one or more of the subscribing applications cannot keep pace with the flow of messages being published, it can potentially bog down its broker and thereby cause performance issues for all publishers and subscribers.

AmbrosiaMQ has implemented an automatic flow control system that is used to ensure that a single slow consumer cannot adversely affect the flow of messages to all consumers. If a subscriber cannot process messages fast enough, the AmbrosiaMQ broker can respond with one of five different remedies:

1. The broker is able to maintain a separate set of message queues for each subscriber.
2. Each publishing session can set up with two different "swim lanes" for messages – a slow lane and an express lane. The flow of messages in the slow lane will not affect the flow of messages in the express lane. Depending on the application, the publisher may or may not send duplicate messages in both lanes. Thus, a slow subscriber to a publisher in session 1 cannot affect a publisher in session 2. In this example, both publishers can be in the same application.
3. A publisher can decide if it wants to slow down its transmission, or if it wants the subscriber to be disconnected.
4. Flow control notifications are sent to the original publisher and not to any intermediaries. Thus, any broker in between the publisher and subscriber will continue to function as fast as possible.
5. An application that receives a message and subsequently publishes that message can link the newly published message to the originally received message. In this manner, in case of a slow subscriber, the original source of the message gets the flow control warning and the intermediaries can continue to function as fast as possible.
    - *Consider an example in which a publisher P1 sends a message to a subscriber S1. S1 can reformat the message and publish it to subscriber S2. S2 can reformat and publish it to S3 and so on until Subscriber Sn, which publishes its message to the target T. Now, if T is slow, who should get the flow control warning? The immediate upstream publisher of T is Sn, but Sn is really not the source of the data flow, S1 is. Said another way, AmbrosiaMQ has the intelligence to recognize these situations and deal with the issue at its sources, namely the original publisher.*

It is important to note that the publisher maintains complete control over all of these elements, on a session-by-session basis.

(b) Fast Serialization

In many applications the schema of messages is well known, and this knowledge can be used to increase data throughput. It enables us to very rapidly and efficiently condense the message down to the bare essentials, so that less information is traveling between points A and B. We call this *fast serialization*.

AmbrosiaMQ takes advantage of the schema knowledge in the following ways:

- Supports many Java objects, including wrappers for primitive types
- Can leverage a schema repository (dictionary) so that only the object index is written into the message
- Uses configuration servers to automatically distribute the schemas to all receiving applications
- Dictionaries are versioned and can be exported and imported using XML files
- Client applications have visibility to object lengths in the message
- We provide the ability to modify messages without de-serializing

Fast serialization saves both space and time, and significantly increases both performance and scalability.

(c) Intelligent Routing

AmbrosiaMQ employs an intelligent routing mechanism in which only a single copy of the message is sent between message brokers. The message fan out occurs at the broker closest to the subscribers. Some of the characteristics of the intelligent routing include:

- Each broker filters the messages, based on their topic, so that only relevant information is sent to each client
    - No client side filtering is required
- Only one copy of the message is sent between brokers
    - Fan-out occurs at the broker closest to the subscriber
- Route limits
    - Local
    - Inter-broker (single cluster)
    - Regional
    - Global

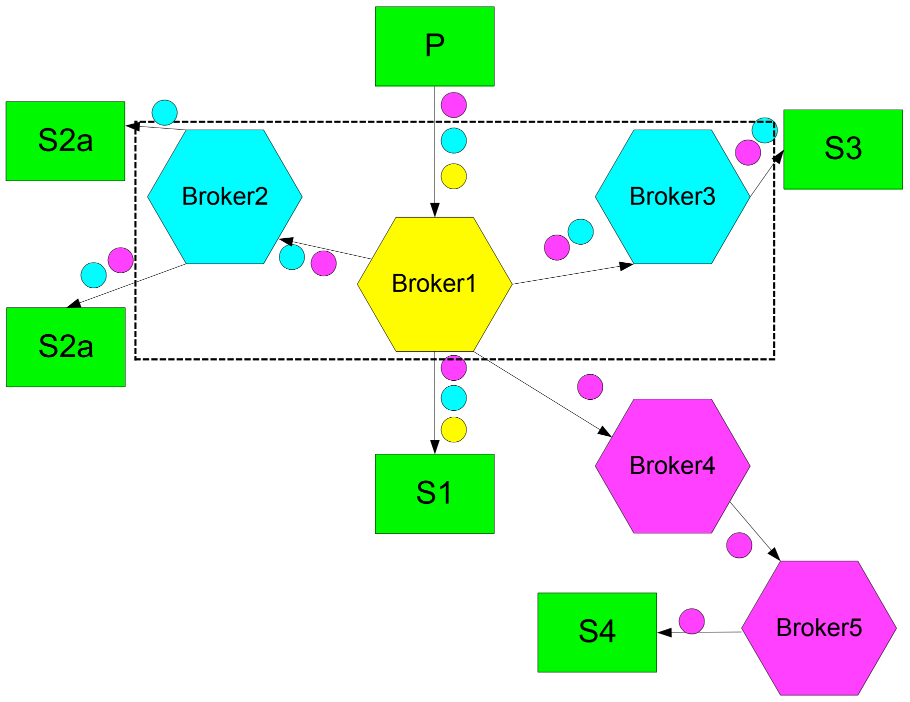

Figure 2 – Message routing through an AmbrosiaMQ broker network

In addition to enhancing reliability and scalability, intelligent routing also provides a more secure environment by delivering messages only to subscribers that are allowed to receive it. This capability is analogous to a firewall that segments a network to "internal" versus "external".

In addition to intelligent routing, AmbrosiaMQ supports the notion of route limits, which are designed to segment traffic and only transmit what is needed between different segments. A concrete example is that NYSE prices need not be transmitted to London subscribers, even though a London subscriber subscribes to the wildcard "stockprices.#".

(d) Broker Clusters

A single point of failure in any messaging system is the server, or broker, that is managing the flow of messages between publishers and subscribers. If a client (producer or consumer) is inextricably tied to a single broker, then failure of that broker would render the client inoperable.

AmbrosiaMQ provides extremely high availability and reliability through the use of multiple, inter-connected broker clusters, which enables potentially infinite levels of redundancy to help mitigate server failure. AmbrosiaMQ also offers multiple deployment methods – a single broker; multiple brokers in a collective; or bridge collectives organized in hierarchies.

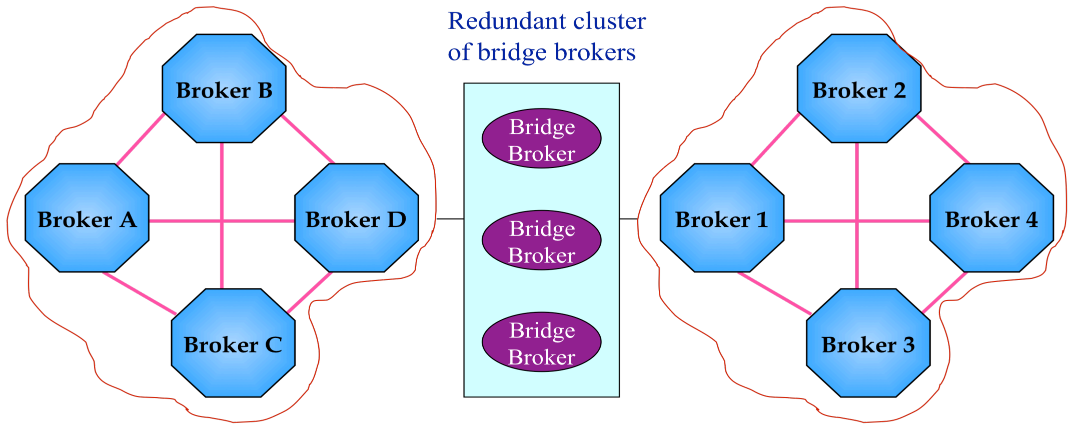

Figure 1 – AmbrosiaMQ Redundant Broker Architecture

Deployments of AmbrosiaMQ inter-broker networks can vary widely in applications. Simple AmbrosiaMQ systems may contain only one or two brokers hosted on a common intranet, whereas a complex system may contain several collectives that cross a wide variety of network topologies including LAN, WAN, DMZ's, and the Internet.

To address this wide range of possible system deployments, AmbrosiaMQ brokers can be configured to assume various service roles that may be required for a particular system. These roles involve distribution of security and configuration data, access restrictions, and various other specialized services.

A key facet of this capability is that the deployment method is completely opaque to the application developer. Applications are written to either publish, subscribe, or both – and the back-end broker topology can be modified any time to address changing network requirements, without any changes to the application.

(e) Access Control

One of the inherently powerful aspects of a publish/subscribe messaging system is the loosely coupled architecture. A new application can come online, subscribe to a topic, and immediately start receiving published messages – without any other application or service needing *a priori* knowledge of its existence or involvement. Similarly, a new application or service can begin publishing to a particular topic, without the consuming applications being aware of any changes.

While this is very powerful, it is also potentially very dangerous. Should any application be able to subscribe to a particular topic? Should any application be able to publish to a particular topic?

AmbrosiaMQ supports critical security elements of authentication and authorization. Users (or applications) must be authenticated before they are able to publish or subscribe, and once authenticated, they must be authorized to publish or subscribe to/from each particular topic.

AmbrosiaMQ supports integration with any JAAS-conformant authentication mechanism through configuration settings of the AmbrosiaMQ broker. The ability to integrate with an existing security infrastructure allows AmbrosiaMQ to support a large number of users without needing to maintain credentials (i.e. passwords) for them. Thus, we can accommodate users of an entire department in one shot by integrating with the authentication system of that department.

Section 2 – JMS 1.1 Extensions

As stated prior, the JMS 1.1 specification was last updated over seven years ago, in April 2002. As is true with most technologies, they evolve over time as requirements expand and usage increases. AmbrosiaMQ has been used extensively, primarily in the financial services vertical, where limits are constantly being challenged, and innovation is the *modus operandi*. In pushing AmbrosiaMQ to keep pace with the evolving requirements, U1 has exposed new capabilities through an extended set of APIs.

This section outlines and describes the features and capabilities that are implemented within AmbrosiaMQ that go beyond the JMS 1.1 specification.

(a) Virtual Circuits

Virtual Circuits are designed to address a tantalizing problem in the messaging paradigm: reliable delivery (zero-or-more) may not be sufficiently robust, but guaranteed delivery (at-least-once or exactly once) may be too expensive from the perspective of performance and scalability. For these situations, developers can leverage Virtual Circuits. Two processes establish a Virtual Circuit (VC) through which they receive notifications about each other's connectivity status (i.e., up or down).

The essential value of the Virtual Circuit paradigm is that it enables a style of programming that delivers the best of both worlds: consistent, high-performance message delivery, with no risk of failures from undetected non-delivery of messages.

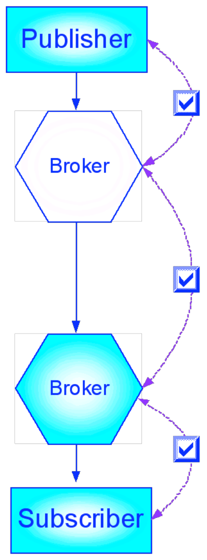

Figure 3 – Communication of status between two AmbrosiaMQ applications via a Virtual Circuit

With VCs, an application can simultaneously leverage the speed of reliable delivery and take corrective actions when its peer cannot take delivery of the message.

(i) Virtual Circuit Client

This interface specifies the client-side methods of a Virtual Circuit. A client uses the methods in this interface to set up its half of the VC. Once the client sets up its half of the VC, it calls the start method. Subsequently, the client's various handler methods are called to indicate the status of the circuit. If/when the client loses the VC, it should automatically try to re-establish the connection. The method `setRetryInterval` controls how long the client will retry.

(ii) Virtual Circuit Server

This interface specifies the methods for the server side of a Virtual Circuit. A server uses the methods in this interface to set up its half of the VC. Once the server sets up its half of the VC, it calls the start method to listen for requests. Request messages are delivered to the handler. It is in the handler that the server can accept the request and establish the circuit with the client.

(b) Connection Properties

A JMS *Connection* is a client's active connection to its JMS provider. An enterprise-class messaging system typically requires control over the behavior of the client or the provider for performance, security, and scalability reasons.

In AmbrosiaMQ, the interface provides the mechanism for an application to get and set various connection properties such as:

1. Client side buffer/queue sizes
2. Flow control parameters – the various parameters that govern the behavior of the provider in case of there are too many messages buffered for a client application.
3. Duplicate connection checker – does the provider allow duplicate connections for the same user/application pair? Implementations of this interface would impose similar type conversion rules as defined in the JMS `MapMessage` interface.
4. Maximum bandwidth of the client's connection to the provider

(c) Delivery Specifier

Enterprise-class messaging systems often consist of a collection of message servers (e.g., message brokers) that are deployed in various network topologies across diverse geographical locations. Messaging applications often desire to specify additional semantics on message delivery, such as route limit (i.e., how far a subscription request or a published message is allowed to travel in a messaging network).

In other cases, when messages arrive, the application may prefer to replace prior messages (e.g., when delivering real-time trading values of a particular stock, or delivering the current location of a visual object in a real-time game).

This interface includes a set of methods that messaging applications can use to instruct a provider regarding the delivery of a message including:

1. Route limits
2. Discardable delivery
3. Most Recent Value (MRV)

(d) Message Producer

Messaging applications often face a situation in which a slow message consumer can adversely affect the flow of messages from a producer to all consumers. It is often very desirable to specify what should happen in the event that a consumer cannot handle messages fast enough. For example, if a consumer's buffer becomes full (due to the inability to process incoming messages in a timely manner), the consumer may send a request to the message producer asking it to suspend publication until the consumer is able to *catch up*. The consumer's request is only a suggestion to the producer; the producer may decide to suspend publication, or may decide to have the consumer's connection be severed.

The `MessageProducer` interface describes how a message producer will behave if it receives a suspend request. The message producer can either suspend itself, or it may force the consumer to be kicked off.

(e) Solicit Producer and Solicit Consumer

JMS provides a mechanism for synchronous request/reply. It also anticipates the use of asynchronous request/reply, though it does not specify any APIs for it. Solicit/response is an expansion of JMS request/reply and represents a communication model in which a *Solicitor* asks for a reply from any *Responder*. This model has been used by messaging applications before, but up to now there has been no standard JMS APIs that specify this model.

The methods in these interfaces enable a message producer and consumer to communicate with each other using a Solicit/Response model via a standard set of JMS APIs. The Solicitor can cancel its solicitation based on its own application-specific rules. Examples include:

1. After the first response is received
2. After at least *n* responses are received
3. After a time-out period has elapsed, this interface specifies the Responder

Section 3 – Testing & Measurement

Getting back to our "oils" theme – it is not until you get the car out on the track that you'll find out how well all the components hold up. Translation: white papers may be good vehicles for communicating concepts and information, but they typically stop short of being actionable.

Our goal with this white paper is to make this an actionable exercise. We want to give you, the reader, not only the data to understand the argument intellectually, but also the tools and measurement capabilities to be able to see how this all works in the real-world.

The JMT described below is currently being finalized and will be released in mid-May 2009

(a) JMT – the "Java Messaging Thermometer"

To that end, U1 has developed a tool called JMT – the "Java Messaging Thermometer" – that is designed to test and measure the three facets of performance, scalability and reliability across any JMS implementation. In addition, we also provide an interface to AmbrosiaMQ proprietary APIs, as well as an interface to the Tibco Rendezvous™ APIs.

While we could publish the results of tests we've run using JMT, we believe it's more powerful to enable you to run the tests yourself.

In addition, some messaging vendors explicitly prohibit anyone from publishing benchmark data that includes their product, and we want to be respectful of that. Our belief is that transparency wins in the end – get the results out on the table for everyone to see, and make decisions based on facts.

(b) JMT Design & Extensibility

The JMT design is very straightforward, enabling developers, testers or even network administrators to simulate real-world scenarios that push the limits of any JMS implementation. JMT is implemented in Java, and it can be run on virtually any environment.

We recommend that side-by-side tests be performed in a variety of ways:

- Over a local area network
- Over the Internet
- With a single broker
- With multiple brokers
- Combinations of the above

JMT has a client-side component that is used to produce (or publish) messages. The publishing client has two levers that can be throttled up or down, to test different application scenarios:

1. vary the speed – how often is each new message published?

2. vary the size – how small or large is each message?

Obviously, as the size of the messages and the speed of publishing both increase, the JMS server gets more effectively exercised, and it becomes easier to see when something breaks, or how the three facets degrade under stress.

JMT also has a client-side component that is used to consume (or subscribe to) messages. The JMT subscriber listens for topics of interest that get published, and consumes each of those messages.

The last major component is a set of methods for measurement. We provide methods for measuring several metrics out-of-the-box:

1. Latency (Publish Latency and Channel Latency)
2. Throughput
3. Message Sequence Flow (detection of duplicate/missing/out-of-sequence)
4. Inter-Arrival Time

The metrics can be used to do general comparisons of different messaging products. Additional metrics can be easily developed to help analyze application specific use cases.

The source code for JMT is provided, and we encourage third parties to develop additional metrics for testing. U1 is committed to updating and distributing enhancements, such as additional measurements and adapters that are provided by the community, or that are developed by U1.

Summary

We started this discussion by talking about oils, and the bold claims of ‘that Valvoline guy back in the 70s’.

Let us reassert the key point we are trying to make: JMS implementations are not all the same, and in fact, the differences between them can be quite stark.

We provide a framework for evaluating the most critical aspects – the performance, reliability and scalability attributes – and we’ve gone further by providing an actual toolkit for doing the testing and measurement of each of these attributes.

We invite you to download AmbrosiaMQ and see for yourself.

Please contact us for more information at solutions@u1.com

##