Automated Analysis of Java Message Service Providers

Dean Kuo and Doug Palmer

Software Architectures and Component Technologies Group
CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra, Australia
{dean.kuo, doug.palmer}@cmis.csiro.au

Abstract. The Java Message Service (JMS) is a specification that provides a consistent Java API for accessing message-oriented middleware services. This paper presents a test harness that automates the testing of JMS implementations (providers) for correctness and performance. Since the JMS specification is expressed in informal language, a formal model for JMS behaviour is developed, based on the I/O automata used in other group communication systems. The test harness has been successfully used to test a number of JMS implementations. This paper contains a descriptive presentation of the formal model, the full details are found in a technical report.

Keywords: Automated component testing, performance analysis, formal model, JMS, I/O Automata.

1 Introduction

Message-Oriented Middleware [13] (MOM) is a popular building block for the construction of distributed systems especially loosely coupled systems. When using MOM, decoupled software components communicate by asynchronously exchanging messages. The message management layer can be separated from the application code and MOM systems typically provide a platform for reliably delivering messages between these independent software components. Messaging systems can be contrasted with synchronous remote procedure call systems, such as CORBA; the asynchronous nature of MOM makes it a suitable candidate for designs where a high level of autonomy is a desirable design goal. See [2] for a survey of middleware models.

In general, the semantics of MOM are quite simple: the construction of a message, passing it to the middleware with a destination and receiving a message from the middleware. One source of additional complexity arises from services supplying guaranteed levels of reliability and management. For example, it may be necessary to have messages survive across failures or be delivered in the same order relative to messages from other senders. Another source of complexity is whether the messaging system uses point to point, broadcast or multicast¹ delivery. The problems of reliable message delivery have been extensively studied in *Group Communication Systems* (GCSs). GCS can be categorised according to the properties that they have in terms of message delivery, reliability and membership management [16,3].

¹ Normally called publish/subscribe or pub/sub in MOM circles.

MOM can be provided as a simple API, since it can be modelled as a single service to which requests for delivery or notification are made. The *Java Message Service* (JMS) is a Java interface specification to MOMs [4,12]. Individual JMS providers supply a underlying platform that supports the semantics of JMS. Examples of JMS providers are MQSeries [5] and SoniqMQ [14].

When building a JMS provider, the question of whether the provider accurately follows the JMS specification arises. JMS provides a large number of features: persistent delivery, durable subscription, transactional delivery, to name a few. Testing a JMS provider is complicated by JMS being part of a distributed, asynchronous system, making questions of whether a message has been received, and whether it *should* have been received an essential part of the testing process. Ideally, the JMS specification would contain a formal model that exactly specifies which messages are required to be delivered and to which receivers. Testing a JMS provider would then be a matter of comparing the behaviour of the provider against this formal model. However, the JMS specification is designed only as a standard API to existing MOM and is not a model for messaging. As a result, the JMS specification is, at places, somewhat vague on the expected behaviour of a JMS provider, with delivery latency and subscription latency affecting the delivery of messages.

The properties of GCSs can be used to build a formal model of JMS behaviour. Many of the properties of GCS described by Vitenberg et al [16] can be directly applied to JMS. Some of the formalism of GCSs, however, rely on more knowledge than JMS is able to provide.

GCS group membership is expressed in terms of *views*: sets of processes to which a message should be delivered. The sender of a message knows which view a message has been sent in and, therefore, which receivers should be getting the message. Publish/subscribe, however, is usually an anonymous communication model with a publisher publishing on a *topic* with no knowledge of the potential subscribers [10]. A consequence of this anonymous model is that JMS provides no means of determining view membership.² As a result, the view formalism cannot be applied to JMS without a considerable white-box knowledge of the underlying provider implementation.

Similarly, the liveness properties discussed in Vitenberg et al cannot be directly derived from the JMS specification, as failure detection is not part of the JMS specification. In addition, JMS is a purely asynchronous system; no upper bound is placed on message delay[3]. In general, however, performance measures, such as the maximum sustainable throughput of messages and the maximum message delay can be used to impose an upper bound; messages that are delayed beyond this time are assumed to be lost.

This paper describes a methodology for black-box testing of a JMS provider. A test harness exercises the JMS provider and a model of the expected behaviour of the JMS provider is built; this model can then be compared against the actual behaviour of the provider. Black-box testing relies on the externally observable behaviour of a system. Since views are not observable entities in JMS, the model uses initial and final message deliveries to a receiver to mark changes of view. Performance measures are used in place of testing liveness properties. This methodology has been implemented as a distributed

² A JMS provider may have a distributed architecture that makes exact calculation of view either impossible or very expensive. For example, a collection of listeners acting as filters for groups of subscribers.

test harness which manages a series of tests and analyses the results. The test harness has been used to aid the component testing of Fujitsu's JMS product that will be later released onto the market. The harness is also being used to compare performance of different JMS implementations. It should be noted that this paper is not about discussing the results of testing a variety of JMS implementations but to describe a tool that automates the testing.

The test harness provides JMS application developers a vendor independent tool to compare the performance of different JMS providers. The developers can configure the tests to closely resemble the target environment. The result is that developers can gather accurate information on which JMS provider best suits their requirements.

The Ballista project [6] describes the automation of robustness testing of off-the-self software components. That is, it tests for component failures caused by exceptional inputs or stressful environment conditions. They do not test for correct behaviour nor performance testing, which is the focus of the JMS test harness described in this paper.

The paper is structured as follows: Section 2 contains a discussion of the properties of GCSs and their application to JMS. An informal analysis model for JMS, based on the properties discussed in section 2 is presented in section 3. The formal analysis model, upon which section 3 is based, can be found in [7]. Design, implementation and experience gained in developing the test harness is given in section 4. Conclusions and future directions is given in section 5.

2 Background

This section provides a brief overview of JMS and GCS.

2.1 JMS

JMS defines a Java API for message oriented middleware (MOM) and a number of implementations already exist. JMS defines two forms of messaging: point to point and publish and subscribe.

A typical JMS client uses JNDI (Java Naming and Directory Interface) [8] to load a *ConnectionFactory* and a set of destination objects into a JMS client. The connection factory is used to create connections with the MOM which is then used to create a set of sessions. Message producers then use sessions and destinations to send messages to message consumers. Message consumers also use sessions and destinations to receive messages.

A session can be specified as transacted. Each transaction groups a set of message sends and receives into a unit of work. If the session commits then all received messages are acknowledged and all outgoing messages are sent. If the session aborts, all messages received are recovered while all outgoing messages are destroyed — that is, the operations in the transaction never took place.

For non transactional receivers, JMS defines three acknowledgement modes. The options include lazy acknowledgement which reduces the work done by a session but duplicate messages may be delivered; sessions automatically acknowledges each message; and the JMS client manually acknowledges the receipt of message(s).

A JMS client that produces messages create them via sessions. The client specifies whether the message is a persistent message or a non-persistent message. Persistent messages are guaranteed to eventually arrive at its destination(s) even if failures (system or communication) occur. For non-persistent messaging, the messages *should be* delivered since failures may cause it to be lost.

Determining the destination of a message is simple for point to point messaging. The message producer explicitly specifies the intended destination. Defining correct behaviour is thus straightforward.

For publish and subscribe messaging, defining *correct behaviour* requires us to know the destinations of a published message. However, the destination of a message is unknown to the publisher since it has no knowledge of who has subscribed to the topic and there is no way, using the JMS API, to determine who are the current subscribers (i.e. no view membership API in JMS). Discussion and the definition of correct behaviour for publish and subscribe is described in section 3.

JMS defines two forms of subscribers: non-durable and durable. A non-durable subscriber will only receive messages while it is active. For durable subscribers, messages that would have been delivered while it was inactive will be delivered to the subscriber when the subscriber becomes active again — that is, the JMS implementation has to retain all the messages while the subscriber was inactive.

JMS defines a 10 level priority (0 – 9) where 9 is the highest priority and 0 the lowest. The JMS implementation only needs to provide a best effort in delivering higher priority messages before lower priority messages.

Messages from the same message producer are guaranteed to be delivered to consumers in the same order as they were sent if their message priorities and delivery modes are the same. In addition, messages sent in non-persistent mode may skip ahead of messages sent in persistent mode but the reverse is not permitted.

2.2 GCS

The publish and subscribe component of JMS is a lightweight form of (GCS) [16]. GCSs typically provide reliable multicast and group membership services. Each message sent by an application will be delivered, if certain conditions are met, to all the processes in the current membership group. This is similar to JMS delivering messages published on a particular topic to all subscribers that have subscribed to the topic. One major difference however is that JMS does not expose to the JMS application who the subscribers to a topic are while in GCS, each process is notified when a *view* changes.

Applications such as state-machine replication and distributed transactions rely on a precise membership service. We view JMS as a lightweight version of a GCS since JMS does not provide any sort of precise membership service and thus do not provide properties such as *sending* and *same view delivery* [16].

GCS and related papers [16,3,10,9] and papers referenced in [16] typically describe algorithms, analysis algorithms, define correctness and describe implementations of GCS. JMS is a standard and just defines an API, and informally defines correctness.

Both JMS and GCS share some basic safety and liveness properties such as delivery integrity (messages received by consumers where actually sent by some message producer), FIFO delivery and reliable message delivery. JMS' reliable message delivery requires that persistent messages be eventually delivered even if communication

or process failures occur while in GCS, it only requires that messages be delivered in the absence of failures. Thus, JMS' notion of reliable message delivery is stronger than GCS'.

3 Analysis

The main objective of the JMS test harness is to analyse the behaviour — correctness and performance — of any JMS implementation. This section describes how the JMS test harness analyses the results from tests. For each test, an execution trace (logged events from a test) is generated by the JMS test harness. The trace is then analysed to verify correctness and to determine performance.

The first task is to formalise the specification as the JMS specification [4] is currently only informally specified in English. There are two types of correctness properties, *safety properties* and *liveness properties* [9]. Safety properties basically say that what has happened is correct or something bad has not happened. Liveness properties basically say that something *good* will eventually happen if certain conditions are met.

The test harness logs each event that occurs. Thus, it is straightforward to analyse an execution trace to verify that safety properties have not been violated. However, this is not the case for liveness properties since one can not determine if an event will *eventually* happen. The approach taken by the test harness is only test for safety, and instead of testing for liveness, perform performance evaluation.

Sections 3.1 and 3.2 respectively describe the JMS safety properties and the performance analyses. The safety properties discussed in this section are presented informally, giving an intuitive feel for the structure of the properties. A full formal definition is given in [7].

3.1 Safety Properties

This section describes the JMS safety properties. They cover message delivery, transactions, message ordering, durable subscriptions, message priority and message expiration.

Point-to-point messaging essentially delivers messages to a destination; a named queue. Receivers can pick up messages from this destination asynchronously, with the messages waiting until a receiver appears. Publish-subscribe messaging delivers messages to the group of subscribers that have subscribed to the topic the message was published on. Senders to a queue and publishers on a topic are collectively termed *message producers*. Receivers from a queue or subscribers to a topic are *message consumers*.

Message Delivery. This section describes the safety properties for both publish and subscribe and point-to-point messaging. *Message delivery* tests that all messages that are required to be delivered to a consumer are actually delivered to the consumer. Defining exactly which messages fall into this category is not straightforward due to message delivery latency times and latency times to register a subscription.

When a message is sent to a queue or published to a topic, there is a *delivery latency* as the message passes through the message system before arriving at its destination(s). The delivery latency may mean that the message, even though sent while the consumer was

active may arrive after the consumer has closed its connection, leading to the message not being received by the ex-consumer.

When a subscriber subscribes to a topic, there is a *subscription latency* while the subscription propagates throughout the system. Since publishers are not likely to send messages to non-subscribing nodes, it is possible for a published message not to be delivered to a subscriber even though the subscriber has recently subscribed.

Queue receivers and durable subscribers have the property that messages not delivered to a consumer wait until a consumer becomes available. The newly active consumer may not be the same consumer as any of the previous consumers. Given this behaviour, it is thus natural to treat a destination or subscription as an end-point for message delivery and reason about groups of consumers. Note, the group of consumers for a non-durable subscription contains exactly one process as the subscription terminates as soon as the process closes the subscription.

The set of messages that are required to be delivered to a consumer depends on the operational modes of the producer and consumer — e.g. transactional, durable (subscriber) and persistence. Before we can define exactly which messages are required to be delivered, we need a number of preliminary definitions to capture the semantics of transactional producers and consumers, and durable subscribers. The first definition defines exactly which messages have been sent by a producer. In the definition, notice that for transactional producers, a message is not deemed to be sent until the producer commits.

Definition 1. *Sent Messages*

If the producer p is *non-transactional*, then the message msg is defined as sent by p if the producer successfully sends the message — that is, the method `publish(msg)` or `send(msg)` returns with no exception thrown.

If the producer p is *transactional*, then the message msg is defined as sent by p if the publisher successfully sends the message within a transactional context t and t later commits. $\square$

Similar to the above definition, we define exactly which messages have been received by a consumer.

Definition 2. *Received Messages*

If the consumer c is *non-transactional*, then a message is defined as received by c if the consumer successfully receives the message.

If the consumer s is *transactional*, then a message is defined as received by s if the consumer successfully receives the message within a transactional context t and t later commits. $\square$

It is now possible to define the basic delivery integrity property, which states that each message received by a consumer was actually produced by some producer. This property is the same as property 4.1 in [16].

Property 1 *Delivery Integrity*

For each consumer c and each message m in c 's *Received Messages*, m is also in the set *Published Messages* for some producer p . $\square$

Due to delivery latency and subscription latency, it is not possible to define ahead of time exactly which messages will be delivered to a consumer. Instead, given the first message from a producer that arrives at a consumer and the last message from a producer that arrives before the consumer closes, it is possible to construct the set of messages that should have been received in the intervening period. To do this, it is necessary to identify the next message from a producer that should follow a received message.

Definition 3. *Next Message*

Given a message m sent by a producer p and received by a consumer c , the *next message* that should be received by c from p is the message produced immediately after m by p .

Messages are assumed to be delivered to either queues or subscriptions (each with a unique identifier³), representing a *consumer group*. Consumer groups are given the identifier of the queue or the subscription that they represent. A consumer becomes the active consumer of a consumer group when it opens a destination and it stops being the active consumer once it closes the destination. Note, message may still arrive at a consumer group even if there are no active consumers. It is therefore necessary to identify the last close and last message delivered. Note that, for non-durable subscriptions, there will be at most one close operation since the consumer group for a non-durable subscription contains exactly one subscriber.

Definition 4. *Last Close*

The *last close* of a queue or subscription id is the last close operation of consumer (queue receiver or subscription) group on id . □

Definition 5. *Last Message*

The last message from a producer p to a consumer group id is the last message sent from p to id which is received before the last close of id . □

The last message defines the last message received by a consumer group from a producer. It is also necessary to identify the first message received by the consumer. Subscription latency in publish and subscribe style of messaging means that the definition of *first message* will be different for a queue and a subscription.

Definition 6. *First Message*

The first message, m , from a producer p to a consumer group id is either:

- If id is a queue, then m is the first message sent by p .
- If id is a subscription, then m is the first message sent by p that was received by a subscriber to id . □

Once the first and last messages have been identified, correctness requires that all messages between these messages are received by some consumer of the queue or subscription.

Property 2 *Required Messages*

The *required message set* for an end-point (queue or subscription) id and a producer p is defined, recursively, as:

³ Non-durable subscribers are allocated an artificial subscription for the life of the subscriber.

- If m is the first message from p to id then m is in the required message set.
- If m is in the required message set, and m' is the next message after m from p , then either m is the last message from p to id or m' is in the required message set.

The required message set is the union of all required message sets over all p and id . Correctness requires that the union of all messages received by consumers be a super set of the required message set. $\square$

Message Ordering. Messages sent by a message producer with the same message priority and delivery mode and, on the same topic in the case of pub/sub messaging style, must be delivered in the same order as it was sent. Message ordering is a basic property of any GCS and property 6.1 in [16] describes this property. Verifying that message ordering has been preserved can be reduced to verifying the following property:

Property 3 Message Ordering

Take any message msg received by a message consumer and message msg' is the previous message received by the consumer that is from the same producer, on the same topic (in the case of pub/sub messaging) with the same message priority and delivery mode as the message msg .

Ordering is preserved if msg' was published before msg . $\square$

Figure 1 illustrates that if message ordering is not preserved then there exist messages msg and msg' satisfying the conditions in property 3 where msg' was *not* published before msg .

Message Priority. Since the JMS Specification specifies that supporting message priority only requires a *best effort*, the following property can be relaxed or dropped altogether. There is no corresponding property in [16].

Property 4 Message Priority

The mean message delivery time between a producer and consumer for a lower message

Fig. 1. Message Ordering

priority is greater or equal to the mean message delivery time for a higher message priority.

This correctness property assumes that messages produced for the different priorities are produced at the same rate, on the same topic or to the same queue and with the same message delivery mode. $\square$

Expired Messages. The JMS specification requires that a JMS provider should do its best to accurately expire messages [4]. However, the specification does not define the accuracy required by a JMS provider, beyond stating that time-to-live should not be simply ignored. There are two parts to this property: one relates to the number of expired messages that have been delivered to a consumer; the other relates to the number of non-expired messages that were not delivered to the consumer. The test harness (a JMS client) has no way of determining which messages have been expired while in the JMS provider, and which messages were correctly delivered, but expired by the time the client processed them. Precise tests for message expiry are, therefore, impossible from the perspective of the test harness.

An expectation model allows the expiry property to be tested, without too much knowledge of the internal workings of a JMS provider. To build the expectation model, it is necessary to define which messages *could* have been received by a consumer.

Definition 7. Possibly Received Messages

A message m sent to a queue q is a member of the possibly received messages of q . A message m published on topic t is a member of the possibly received messages of s , if s has subscribed to t . $\square$

The simple expectation model that we have deployed specifies that a possibly received message is expected to be delivered if the mean latency time is less than or equal to the time-to-live time of the message or when the message's time-to-live is 0⁴; Otherwise, the message should not be delivered.

Property 5 Expired Messages

The property is in two parts.

- The number of expired messages that are delivered as a percentage of the number of expected expired messages is less than some pre-specified percentage.
- The number of non-expired messages delivered as a percentage of the number of expected non-expired messages is greater than some pre-defined percentage. $\square$

The configuration we define to test message expiration sets message time-to-live to either 1 (milli-seconds) or 0 (infinity) — thus, all messages with time-to-live of 1 are not expected to be delivered while all messages with time-to-live of 0 are expected to be delivered.⁵

A more realistic expectation model would use a message's time-to-live and the run's delay histogram to determine the probability that a message should be delivered. As the test configuration only use time-to-live of 0 and 1, the simplified expectation model is sufficient for testing purposes.

⁴ A time-to-live of 0 means that the message never expires — i.e. infinite time-to-live

⁵ Granularity considerations within the JMS provider may result in some inaccuracy in expiry. For example, a 1 milli-second time-to-live may be rounded up to 1 second.

3.2 Performance

A trivial JMS implementation — one that never delivers any messages — will satisfy all the safety properties discussed in Section 3.1. Liveness properties show that a JMS implementation does something useful but it is not possible, from analysing an execution trace, to verify liveness properties. Instead of testing for liveness, the JMS test harness measures the performances of the JMS implementations. Trivial JMS implementations can easily be identified, since their throughput will always be zero messages per second.

Performance analysis enables users of JMS to compare the throughput of different JMS implementations. They can then use these results to determine the implementations that meet their performance requirements.

The configuration of a test specifies the rate at which messages should be sent. From the execution trace, the analysis determines the rate at which messages were actually sent and the rate at which messages are delivered. The configuration is highly flexible and allows the users to specify the message body type (StreamMessage, MapMessage, TextMessage, ObjectMessage and BytesMessage) and size of messages to be sent, the message priority, delivery mode, whether the producers and consumers are transactional, the acknowledgement modes of the consumers (if applicable), amongst others. Tests can also be configured such that the senders send messages in bursts or with a profile corresponding to a poisson distribution. The test harness can be employed to determine the performance of the JMS provider under different configurations without the need to write any code.

Figures 2 and 3 show two different behaviours of two JMS providers under pub/sub as the message send rate is increased over a number of tests — note that these figures do not have equal scales. The first figure also shows that the throughput remains constant when the send rate is greater than then maximum throughput that the JMS implementation can handle. The second figure shows the subscriber throughput drops as the system is over-stressed.

Fig. 2. Example Throughput Results: JMS Provider I

The performance measures taken give some indication of the overall behaviour and usefulness of the system. The performance measures are also used to evaluate some correctness properties, where approximate measures are taken (properties 4 and 5).

Fig. 3. Example Throughput Results: JMS Provider II

A running test is divided into three periods: *warm-up*, *run* and *warm-down*. The warm-up period is a pre-defined period at the start of each test. After the warm-up period, the harness assumes that the system is in a steady state. The warm-down period represents a period during which producers stop producing messages, allowing consumers to catch up with any tail of un-consumed messages that has built up. The correctness properties apply to all three periods. Performance measurements are taken only for messages produced during the run period.

The performance measures are calculated by examining the test harness logs and matching message identifiers. The following performance measures are taken:

producer throughput. The rate of message production, both in terms of messages per second and message body bytes per second.

consumer throughput. The rate of message consumption, both in terms of messages per second and message body bytes per second.

message delay. Message delay is calculated as time between the start of the message delivery to a consumer and the start of the call to send or publish the message⁶. The maximum, minimum, mean and standard deviation of message delays are calculated.

fairness. Fairness is a measure of the level to which the provider favours a producer or consumer over other producers and consumers — the standard deviation of the message delay is another measure of fairness. Unfairness is defined as the standard deviation of the per-producer or per-consumer mean delay.

4 Design

The JMS test harness is designed to allow automated testing of a JMS provider across a number of independent machines, connected by a network. The architecture of the harness is shown in figure 4.

⁶ An alternative measurement of delay would be the time between the message delivery and the time the message production method returns. Some JMS providers appear to implement message delivery, particularly for non-persistent messages, via a series of synchronous calls. The alternative measurement, in these environments, can produce apparently negative delays.

Fig. 4. Test Harness Architecture

A number of tests are run, in separate Java virtual machines and distributed across several systems. A test creates a variety of producers and consumers and starts sending and receiving messages. As each message is sent and received, these events are logged to disk, along with the unique message identifier and a timestamp.⁷ Individual producers and consumers can be configured with different message production, persistence, durability and other characteristics, as well as connection and disconnection behaviour. The producers and consumers are grouped into *nodes*, which can be configured to share resources such as JMS connections or sessions.

The running of a test is handled by *test daemons*, usually one to a machine. The test daemons are responsible for launching the tests, starting the tests in a coordinated fashion and monitoring the tests for completion (or failure). The test daemons are coordinated by a *daemon prince*, a program responsible for scheduling tests and ensuring that the test daemons stay coordinated. The daemon prince, test daemons and tests use RMI [15] to coordinate, to avoid relying on the same middleware as the middleware being tested.

When a test completes successfully, the test logs are collected and returned to the daemon prince. The daemon prince then inserts the logs into a SQL database, using JDBC [17]. A set of SQL statements are then used to verify correctness and to determine performance. The SQL statements correspond to the JMS properties described in section 3.

The SQL database used is Microsoft Access. Configuration screens, test generators and results reporting are handled by the Access forms and reports facilities.

4.1 Experience

In general, the test harness has performed well. It has proved quite robust, catching crashed tests, cleaning up and continuing on with the next test. Hooks for initialisation scripts allow the JMS provider to be reset between each test, if the JMS provider has failed.

⁷ The test analysis is dependent, particularly when testing performance, on all system clocks being synchronised. The network time protocol [11] (NTP) provides synchronisation to millisecond accuracy — the precision of the timestamps.

During simple correctness testing, where the number of messages produced is fairly small, the above design is adequate. However, when performance or stress testing is performed, the number of messages is larger. With the larger message load, JDBC represents a bottleneck as each message needs to be loaded into the database⁸. In addition Microsoft Access tends to slow with the larger numbers of records, even when tables and queries are optimised. For performance testing, a database is not really necessary, as only simple statistical information needs to be gathered. This information can be computed by the daemon prince and then inserted into the database.

5 Conclusions and Future Work

This paper has described the design and implementation of a JMS test harness that automates the process of correctness testing and performance evaluation. For JMS vendors, the harness automates the process of component testing. For JMS users, the harness automates independent performance evaluation of a number of JMS implementations. The results is then used to decide which is the most suitable JMS implementation that meets the user's requirements.

We have also shown that, even given the restrictions placed on the harness by the JMS API, it is possible to construct a rigorous set of test criteria for most of the requirements of the JMS specification. The test criteria are basically the formalisation of parts of the JMS specification. In the implementation, these are then translated to SQL statements to test for correctness. Similarly, a set of SQL statements are used to generate performance evaluation reports.

The JMS test harness, at this time, can not initiate a system or program crash and then recover from the failure. This is required to fully test persistent delivery mode. Future work includes adding this feature into the test harness.

There are several other possible areas for future work. The criteria for testing message priority and time-to-live (criteria 4 and 5 in section 3) are not particularly rigorous. The strictness of message priority analysis can be enhanced by building a model that indicates whether two messages are candidates for priority considerations. A more accurate expectation model for message expiry can also be built, allowing more accurate testing. Improving both criteria requires more sophisticated delay expectation models. More sophisticated models can be built either by constructing a histogram of message delays throughout the run period or by using a normal distribution for expected message delay, using the mean and standard deviation for the run.

By the time a JMS provider is released, it is only to be expected that the provider conforms to the JMS specification. However, different providers show markedly different performance characteristics.⁹ When a client chooses a JMS provider, a considerable amount of effort needs to be expended in acquiring and configuring JMS providers and running tests that simulate the load that the client's application is expected to produce. The test harness described in this paper allows the automated testing of multiple JMS platforms. The platforms can be configured by a service vendor and a web interface

⁸ Analysis is done at the end of each test and before a new test starts and thus does not affect the performance of JMS

⁹ A preliminary comparison of MQ Series[5], WebLogic[1] and SoniqMQ[14] JMS providers shows performance differences of a factor of 10 in some cases.

used to describe the type of scenario envisaged. The client can then see the performance results of several systems, allowing them to focus on a few likely candidates for more rigorous evaluation.

References

1. BEA Systems. Programming weblogic 6.0 jms. <http://e-docs.bea.com/wls/docs60/jms/>.
2. P. A. Bernstein. Middleware: A model for distributed system services. *CACM*, 39(2):86–98, 1996.
3. V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullendar, editor, *Distributed Systems*, pages 97–145. Addison-Wesley, 2nd edition, 1994.
4. M. Hapner, R. Burridge, and R. Sharma. *Java Message Service (JMS) 1.0.2*. Sun Microsystems, Java Software, Nov. 1999.
5. IBM. Mqseries family. <http://www.ibm.com/software/ts/mqseries/>.
6. N. Kropp, P. Koopman, and D. Siewiorek. Automated robustness testing of off-the-shelf software components. *Fault Tolerant Computing Symposium*, 1998.
7. D. Kuo and D. Palmer. Automated analysis of java message service providers. Technical Report 01/123, CSIRO Mathematical and Information Sciences, GPO Box 664 Canberra ACT Australia, 2001. <http://www.cmis.csiro.au/adsat/reports>.
8. R. Lee and S. Seligman. *JNDI API Tutorial and Reference*. Addison-Wesley, 2000.
9. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
10. A. G. Mathur, R. W. Hall, F. Jahanian, A. Prakash, and C. Rasmussen. The publish/subscribe paradigm for scalable group collaboration systems. Technical Report CSE-TR-270-95, Department of Electrical Engineering and Computer Science, University of Michigan, Nov. 1995.
11. D. Mills. Network time protocol (ntp). <http://www.eecis.udel.edu/~ntp/>.
12. R. Monson-Haefel and D. A. Chappell. *Java Message Service*. O'Reilly, 2001.
13. T. Ouellette. For many companies, MOM knows best. *Computerworld*, 30(24), 1996.
14. Progress Software. Soniqmq. <http://www.progress.com/sonicmq/>.
15. Sun Microsystems, Java Software. *Java Remote Method Invocation Specification*, 1999. <ftp://ftp.java.sun.com/docs/j2se1.3/rmi-spec-1.3.pdf>.
16. R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report CS0964, Computer Science Department, the Technion – Israel Institute of Technology, Sept. 1999.
17. S. White, M. Fisher, R. Cattell, G. Hamilton, and M. Hapner. *JDBC(TM) API Tutorial and Reference*. Addison-Wesley, 2nd edition, 1999.