Experimental Robustness Evaluation of JMS Middleware

Nuno Laranjeiro, Marco Vieira, Henrique Madeira
CISUC, Department of Informatics Engineering
University of Coimbra, Portugal
{cnl, mvieira, henrique}@dei.uc.pt

Abstract

The use of Java Message Service (JMS) for enterprise applications communication and integration is increasing very quickly. However, although JMS is frequently used in business-critical environments, applications are typically developed with the assumption that the middleware being used is robust, which is not always the case. Robustness failures in such environments are particularly dangerous, as they may originate vulnerabilities that can be maliciously exploited with severe consequences for the systems subject of attack. This paper proposes an approach for the evaluation of the robustness of JMS middleware. Our approach is presented through a concrete example of evaluating the robustness of three well-known JMS solutions (JBoss MQ 3.2.8.SP1, JBoss MQ 4.2.1.GA, and Active MQ 4.1.1), in which several robustness and critical security related problems have been disclosed (including specification conformance disparities).

1. Introduction

Messaging is an easy method of communication between software components or applications. Typically, a messaging client connects to an agent that offers services for creating, sending, receiving, and reading messages. Messaging enables loose coupling between the participants in a distributed environment as senders and receivers do not have to be available at the same time. Also, they do not need to have any kind of knowledge of each other: the only agreement made is the message format [5].

The use of Java Message Service (JMS) [21] is growing quite fast as it provides a very powerful and easy way for communication and integration in Java enterprise applications. JMS is a Java Application Programming Interface (API) that implements a standard for enterprise messaging. In practice, a JMS provider is a middleware software component that provides and manages messaging services. There are many open source and proprietary providers available today (e.g., Apache ActiveMQ, JBoss Messaging, WebSphere MQ, Oracle AQ, Open MQ).

Software faults (i.e., program defects or bugs) are recognized as the major root cause of computer failures [12], [7]. JMS middleware implementations are no exception, as they are complex software components, developed facing the typical time-to-market constraints that, in many situations, lead to the deployment of software not properly tested and, consequently, with residual software defects.

Problems in the communication among software components or applications are particularly acute in message based environments. In fact, this kind of interface faults is particularly relevant in JMS as the middleware has to provide a robust interface to the client applications, even in the presence of invalid inputs (e.g., invalid JMS messages). These invalid inputs may occur due to bugs in the client applications, corruptions caused by silent communication failures, or even security attacks.

Robustness testing allows the characterization of the behavior of a system or component in presence of erroneous input conditions [1], [13], [9]. Robustness tests stimulate the system under testing in a way that may trigger internal errors. This allows developers to solve or wrap the identified problems. Robustness testing can also be used to differentiate systems according to the number and type of errors uncovered.

This paper proposes an experimental approach for the evaluation of the robustness of JMS middleware. Our approach is based on a set of robustness tests that allows discovering both programming and design problems, including the identification of non-compliances between the specification and the way the JMS middleware actually operates. The robustness of the JMS middleware is characterized according to the following failure modes: Catastrophic, Restart, Abort, Silent, and Hindering (adopted from the CRASH scale [13]). Specification conformance issues are categorized in three classes according to the severity of the detected problems.

It is important to emphasize that, although JMS is increasingly being used in complex business-critical systems, robustness problems in the underlying middleware are typically not known by the client application developers. This may become a relevant source of
defects as the applications are very often developed with the assumption that the JMS middleware used is robust and works as described by the corresponding specification. Another important aspect is that robustness failures in distributed environments as JMS based environments are particularly dangerous. In fact, existing robustness problems may originate critical vulnerabilities that can be maliciously exploited with severe consequences, such as denial-of-service (DoS) and data loss. We believe that the proposed approach can be used by vendors to improve their JMS implementations and by programmers to avoid existing JMS robustness problems.

The structure of the paper is as follows. The next section presents some background and related work. Section 3 presents the robustness testing approach. Section 4 presents the experiments and discusses the results. Finally, Section 5 concludes the paper.

2. Background and related work

Java Message Service [21] is a Java API that defines the interfaces and semantics needed for the creation of applications that interact within a messaging environment. JMS unifies messaging concepts in a unique and simple, yet sophisticated, programming model that is implemented by multiple providers.

In a JMS based environment, a middleware component provides services to create, send, receive, and read messages. Client applications produce and consume those messages by interacting with the services supplied by the middleware. Two messaging models are supported: the point-to-point model where, in simple terms, each message is delivered to only one consumer and the publish/subscribe model where a single message can be distributed to multiple subscribers [5].

Robustness benchmarking has become a powerful way to characterize the behavior of systems in presence of erroneous inputs [1]. There are many robustness testing studies in areas such as POSIX and Windows operating systems [13], [4] and Web Services [10]. Nevertheless, JMS messaging environments have been largely overlooked.

Ballista [13] uses a set of tests that combine acceptable and exceptional values on calls to kernel functions of operating systems. The parameter values used in each invocation are randomly extracted from a set of predefined tests and for each parameter a set of values of a certain data type is associated. Each operating system is classified in terms of its robustness and according to a predefined scale (the CRASH scale) that distinguishes several failure modes.

MAFALDA (Microkernel Assessment by Fault injection AnaLysis and Design Aid) [9] is a tool that enables the characterization of the behavior of microkernels in the presence of faults. Fault injection is performed at two levels: in the parameters of system calls and in the memory segments holding the target microkernel. However, only the former is relevant when the goal is robustness testing.

Work on robustness testing of web services infrastructures has started recently. In [10] the authors propose an approach to assess the behavior of web services in the presence of tampered SOAP messages. The proposed approach consists on a set of robustness tests based on invalid web services call parameters. The web services are classified according to the failures observed during the execution of the tests.

Regarding JMS, there are several studies that focus on performance or Quality of Service (QoS) assessment. In [19] the performance of two popular JMS implementations (Sun Java System Message Queue and IBM WebSphere MQ) is tested. Performance testing has also been applied in publish/subscribe domains in the presence of filters [8]. In this work the authors investigate the scalability properties of several JMS implementations. Concerning benchmarking initiatives, the Standard Performance Evaluation Corporation (SPEC) recently released the SPECjms2007 benchmark [18], whose goal is to evaluate the performance of enterprise message oriented middleware servers based on JMS. Note that, in spite of the several JMS performance and QoS studies, no strategy has been proposed so far to assess the robustness properties of JMS middleware. This is just the goal of this paper.

3. Assessing the robustness of JMS Middleware

In a JMS environment a connection factory creates connections that can be used to produce sessions. Each session is able to create message producers or consumers, which in turn have the ability to send/receive messages to/from a destination.

Robustness testing usually involves parameter tampering at some level. Our approach is obviously dependent on the JMS environment specificities and consists of modifying messages immediately before they are sent from a producer to a destination that may be linked to one consumer (point-to-point messaging) or to several consumers (publish/subscribe messaging). Message parameter modification is based on a basic set of mutation rules defined on previous works [13], [9], [10]. This set was extended and adapted to the JMS context as described further ahead.
3.1. Fault model

There are three main parts in a JMS message: header, properties, and body. The fault model proposed considers not only the parameters included in these three parts, but also the multiple ways of setting them (e.g., setting properties using the several methods available: setBooleanProperty, setStringProperty, setObjectProperty, etc).

The JMS specification defines several message header fields (see [21] for a complete list and description). Some of those fields are intended to be set by the client application. For instance, a client may need to set the JMSReplyTo field to specify the object to which a reply to the message should be sent. However, there are several other fields that are set automatically by the JMS implementation just after the client sends the message (by executing the ‘send’ or ‘publish’ method) and before that message is sent to the destination. The JMS implementation must handle and correctly set all fields that are of its own responsibility, even if the client did set them. For example, if a client sets the JMSMessageID field the JMS send or publish method will overwrite it.

It is important to emphasize that header fields are clearly parameters that robustness testing must cover, as they define application level behaviors that must be tested for correctness under faulty conditions.

Message properties are also relevant for robustness testing. All allowed types and conversions are defined by the JMS specification [21].

The specification states that a value written using a given data type can be read as one or more data types. For example, after setting a boolean property it is possible to retrieve it using a getBooleanProperty method or a getStringProperty method (in this case it returns a string representation of the boolean value). Every other retrieval must throw a JMSException as they represent exceptional behavior (e.g., setting a byte property and then retrieving the value as a boolean, float or double should raise a JMSException). In addition it is possible to set an object of an unspecified type by using a setObjectProperty method (the acceptable object types are limited to the primitive objectified types) and retrieve it with a suitable available getter method. In other words, the setObjectProperty method can be used to set, for instance, a Boolean, that can then be retrieved either by using the getObjectProperty method or getBooleanProperty.

Testing the correctness of these conversions requires a later verification of the value set. The goal is to check if the JMS middleware is accurately following the specification. Note that non-compliance between the JMS specification and the way it is implemented may compromise the client application’s business logic. Thus, properties conversion testing is a key aspect that should be included in a robustness test.

Several data types are allowed for the message body: Stream, Map, Text, Object or Bytes. At first sight, it may seem less interesting to mutate the message body to exceptional values. In fact, this is most likely to trigger problems in the client applications instead of triggering faults in the middleware itself. Nevertheless, for sake of completeness we have decided to include in the fault model tests that use valid or null objects for the body.

The robustness tests are based on a set of mutation rules that were defined based on previous works on robustness testing [13], [9], [10]. An important aspect is that we have tried to define rules that focus on the use of limit and exceptional values (which are typically the source of robustness problems), such as:

- Null and empty values (e.g. set a parameter as null or empty).
- Valid values with special characteristics (e.g. non-printable characters in a string).
- Maximum and minimum values in the domain (when a domain is available, either because it is stated in the spec or because the user provided it).
- Values exceeding the maximum or minimum values in the domain (e.g. maximum domain value plus one).
- Values that may cause data type overflow (e.g. add characters to overflow a string’s maximum size).

A complete listing of the mutations rules proposed is presented in Table 1.

3.2. Robustness tests profile

To exercise the JMS provider we need to have a messages workload. This workload is the work to be performed during the testing phase and can be one of the following:

- **User defined**: the benchmark user provides a workload based on the knowledge he has on the environment being tested (e.g., a user may use real applications to have a more realistic scenario). The user may also determine that an elaborate approach is unnecessary and random values are acceptable for the messages generation.

- **Predefined**: the SPECjms2007 benchmark [18] can be used as workload. This is a standard benchmark from SPEC, which guarantees a large degree of representativeness and portability.
Our proposal for JMS middleware robustness testing includes three major phases: Phase 1) valid messages are created and sent; Phase 2) faults are injected in the messages being sent; and Phase 3) involves regular message creation and sending with the goal of exposing errors not revealed in Phase 2.

As shown in Figure 1, Phase 1 is executed once at the beginning of the tests. Phases 2 and 3 are repeated several times, one for each parameter being tested (this includes header fields, properties, and body).

The first phase consists of sending a set of messages (defined by the workload) to a given destination and verifying their delivery at the consumer. The goal is to understand the behavior of the JMS implementation without (artificial) faults. This represents the gold run that allows checking if subsequent robustness tests affect message sending and delivery.

In the second phase a parameter is chosen from the list of parameters not tested yet (this includes header fields, message properties and body). According to the type of the selected parameter, a set of rules are applicable. For instance, if the parameter is numeric, then all the mutations that are applicable to numbers will be applied (see Table 1). Only one rule is selected in each injection period and all messages produced in this period are tampered according to that rule. All fault types applicable to the parameter in question are considered (one fault type is considered in each injection period). Message delivery at the consumer is verified for correctness and consistency.

In the third phase, we let the system run for a predefined period (at least twice the time necessary to run the full workload) in order to check for abnormal behavior not noticed in Phase 2. Notice that a fault may be silently triggered during Phase 2 and only exposed in Phase 3. Additionally, even if a fault is clearly exposed during the injection period, it may or may not persist to the third step. This may be an indication of a more severe problem.

After Phase 3, the system state is restored so that testing can continue with no accumulated fault effects. Testing continues now for the following parameter (as the system was restored, parameter order should have no effect) by repeating phases 2 and 3 (phase one is performed only once). The process repeats until there are no more parameters to test.

As mentioned before, the set of robustness tests performed are generated by applying the set of mutation rules presented in Table 1 to the header fields, properties, and body of messages. All appropriate rules are applied to each parameter during the execution of the tests. The tests execution is a fully automated process.

### 3.3. Failure modes

The robustness of a JMS middleware can be classified according to the severity of the exposed failures. Our failure classification has been adapted from the CRASH scale [13]. In fact, the new mCRASH scale (m stands for messaging) is specially tailored to the JMS context and includes the following failure modes:

- **Catastrophic:** The JMS provider becomes corrupted, or the server or the operating system crashes or reboots.
- **Abort:** Abnormal termination when sending or
receiving messages. For instance, abnormal behavior occurs when an unexpected exception is thrown by the JMS implementation.

- **Silent**: No error is indicated by the JMS implementation on an operation that cannot be concluded or is concluded in an abnormal way.

- **Hinder**: The returned error code is incorrect.

An important aspect is that, although our approach specifically targets robustness aspects, it can also be used to uncover robustness related compliance problems between the specification and the JMS implementation. This way, we also propose a simple and linear classification for non-conformity, or non-compliance observations. These represent the cases where acceptable inputs (as defined by the specification) trigger a problem that should not occur and that problem is handled in a correct way by the JMS implementation. Nevertheless, the problem should never occur in the first place, since the inputs were acceptable. For instance, the specification states that a JMSException must be thrown if the JMS provider fails to send a message due to some internal error. If one of the injected faults causes this behavior and the parameter value is acceptable (as defined by the specification) this represents non-conformity. This classification distinguishes the following three levels:

- **Level 1**: A severe non-conformity was observed. The problem affects basic messaging services (e.g., the inability to send or receive messages, or a specification part that is ignored).

- **Level 2**: A medium severity non-conformity is observed. This can be any non-conformity that is not severe (as defined by Level 1) and is not related with extra functionality (functionality not specified by the standards that was introduced by the JMS provider). It can also represent any kind of restriction over the specification (e.g., restricting the domain value of a variable to an interval smaller than what is in the specification).

- **Level 3**: This level includes non-conformities observed that are the result of an effort to extend the JMS specification (e.g., permitting additional data types as properties).

This compliance classification targets only JMS environments (it is intimately related to message sending and receiving). Classifying a given non-conformity is a complex task and depends on the person that is performing the analysis. Nevertheless, we believe that a straightforward classification is useful to understand the criticality of compliance problems.

### 3.4. Robustness testing implementation details

The JMS environment poses some interesting questions when implementing robustness testing. First, although an API is defined, including the format of the message to be sent (in the form of a Java interface), there is no lower level specification for the message communication format (each vendor uses its own preferred format). This means that, the same message sent using different JMS implementations can be translated into different byte sequences during transportation as the vendor may consider diverse methods for message transmission (e.g., JBoss MQ [16] uses custom serialization for message transmission). Since robustness testing is based on parameter tampering, this makes extremely difficult the creation of a generic proxy capable of intercepting and modifying messages. The same happens if we want to modify the message directly at the JMS provider repository. In fact, the provider may keep sent (but not yet delivered) messages in a database, a binary file or in any other repository or format. Thus, a vendor independent approach is quite difficult to achieve or is simply not feasible.

Our approach is based in Aspect Oriented Programming (AOP) [6]. We make use of this programming paradigm to intercept the message immediately before it is sent to the destination and after its properties are set by the JMS ‘send’ or ‘publish’ methods. As previously said, some message properties are not controlled by the client, thus mutations have to be performed after the execution of the send or publish methods. This approach has some advantages over the use of proxies for message interception or mutations in the JMS repository: it provides a non intrusive way (from the programmer point-of-view) to intercept messages and it enables message tampering in a generic way that can be applied to any messaging implementation that follows the JMS specification.

Our robustness testing implementation is generic. The only aspect that is dependent on the JMS middleware being used is the exact signature of the Java method to be intercepted. If this is not disclosed by the vendor in the documentation, it is necessary to search the JMS implementation source code to identify the method that implements the API ‘send’ or ‘publish’ and select the last method signature before the message is actually sent to the destination. This is typically an easy task that represents only a few minutes effort.

### 4. Experimental results and discussion

In this section we present an experimental evaluation of three different JMS providers using the approach proposed before.
As the JBoss Application Server [14] is one of the most widely used application servers on the market, we decided to test JBoss MQ [16] for robustness problems (including two different major versions). We have also tested the Apache Software Foundation project ActiveMQ [2] as it has a large popularity among the open source community.

Most of the JMS implementations are application server independent (i.e., a given JMS implementation can be used in any application server). In real scenarios the JMS provider used is typically the one that is built into the application server being employed (otherwise a serious configuration effort is normally involved). In this way, we tested each JMS implementation in its most used container, which is JBoss AS for JBoss MQ and Apache Geronimo [3] for ActiveMQ.

As shown in Table 2, several robustness problems were exposed during the experimental evaluation. These problems are of extremely high importance as they also represent major security vulnerabilities. Several minor non-conformities were also found in all three providers. All providers revealed Level 2 non-conformities. ActiveMQ added a Level 3 non-compliance issue to these.

### 4.1. JBoss MQ 4.2.1.GA results

JBoss MQ is the latest production ready messaging product available from JBoss. According to the developers, it was submitted to multiple internal tests and to Sun Microsystems compliance tests (it is 100% Sun Compatibility Test Suite (CTS) [11] JMS compliant). It will be soon superseded by JBoss Messaging [15] and no new features are being added: the developers are currently focusing on bug fixing.

Considering this scenario, we were expecting to find none or a very small number of robustness problems. In fact, JBoss MQ passed all the robustness tests related to properties and body, and almost all the tests related to the header fields. However, it failed in the tests where JMSMessageID was set to null. At first, sending a message with this field set to null appeared to cause no harm to the JMS provider, as the message was correctly stored in the internal JBoss JDBC (Java Database Connectivity) database. However when a receiver tried to read the message an unexpected null pointer exception was thrown and any subsequent reads of that message were unable to succeed. Furthermore, regular subsequent messaging operations were also affected. In fact, consumers were unable to retrieve any valid messages sent afterwards.

This failure was classified as Catastrophic as a corruption of the JMS server has occurred. In fact, a restart of the application server was unable to restore the normal behavior of the JMS provider. The only way we found to recover was to manually delete the invalid message information from the repository. This was possible because JBoss MQ uses a JDBC database that is easy to access and modify using SQL (Structured Query Language). If JBoss used any other proprietary repository, we would have had to delete the whole message repository. Nevertheless, both solutions are totally unacceptable and difficult to manage.

As this represents a severe failure, we then analyzed the JBoss MQ source code in order to find the root for this problem. We concluded that, although the JMSMessageID represents a unique identifier for the message, JBoss uses another variable to uniquely identify each message. This allows sending and storing messages with null JMSMessageID. However, when a receiver fetches a message from the repository JBoss MQ tries to load messages into memory in a Map structure that uses a hash of the JMSMessageID as key (and the whole message as value). This of course produces a null pointer exception as hashing a null value is impossible.

It is important to emphasize that, this serious robustness problem also exposes a severe security issue that can be exploited by hackers to cause Denial-of-Service. In fact, it is quite easy for a malicious user to generate a message with a null JMSMessageID and send it to the JMS provider to crash it. This also shows the usefulness of our robustness testing approach to discover possible security vulnerabilities.

Some minor compliance issues were also detected in JBoss MQ 4.2.1.GA. In fact, the overflow mutation (i.e., overwriting the parameter with an overflowed string) applied to JMSMessageID, JMSCorrelationID and JMSMessageType fields caused a JMSException wrapping a UTFDataFormatException. This JMSException is according to the standard, which specifies that this exception must be thrown if the provider fails to send the message due to some internal error. Nevertheless, the specification does not impose the restriction that the value for these header fields has to be handled as a UTF string. This is the cause for the UTFDataFormatException as the chosen method to write the string imposes a limit on its length [17]. Note that, choosing a different writing method would easily solve this issue. The same happens while setting a property name with an overflowed string (it does not happen while setting the value for the property).

### Table 2: Summary of the problems detected.

<table>
<thead>
<tr>
<th>Provider</th>
<th>Robustness issues</th>
<th>Compliance issues</th>
<th>Security issues</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Type</td>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td>JBoss MQ</td>
<td>Catastrophic</td>
<td>Level 2</td>
<td>Level 2</td>
</tr>
<tr>
<td>4.2.1.GA</td>
<td></td>
<td></td>
<td>DoS Attacks</td>
</tr>
<tr>
<td>JBoss MQ</td>
<td>Catastrophic</td>
<td>Level 2</td>
<td>Level 2</td>
</tr>
<tr>
<td>3.2.8.SP1</td>
<td></td>
<td></td>
<td>DoS Attacks</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>Silent</td>
<td>Level 2</td>
<td>Level 3</td>
</tr>
<tr>
<td>4.1.1</td>
<td></td>
<td></td>
<td>Message</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>suppression</td>
</tr>
</tbody>
</table>

It will be soon superseded by JBoss Messaging [15] (and the whole message as value). This of course produces a null pointer exception as hashing a null value is impossible.

As this represents a severe failure, we then analyzed the JBoss MQ source code in order to find the root for this problem. We concluded that, although the JMSMessageID represents a unique identifier for the message, JBoss uses another variable to uniquely identify each message. This allows sending and storing messages with null JMSMessageID. However, when a receiver fetches a message from the repository JBoss MQ tries to load messages into memory in a Map structure that uses a hash of the JMSMessageID as key (and the whole message as value). This of course produces a null pointer exception as hashing a null value is impossible.

It is important to emphasize that, this serious robustness problem also exposes a severe security issue that can be exploited by hackers to cause Denial-of-Service. In fact, it is quite easy for a malicious user to generate a message with a null JMSMessageID and send it to the JMS provider to crash it. This also shows the usefulness of our robustness testing approach to discover possible security vulnerabilities.

Some minor compliance issues were also detected in JBoss MQ 4.2.1.GA. In fact, the overflow mutation (i.e., overwriting the parameter with an overflowed string) applied to JMSMessageID, JMSCorrelationID and JMSMessageType fields caused a JMSException wrapping a UTFDataFormatException. This JMSException is according to the standard, which specifies that this exception must be thrown if the provider fails to send the message due to some internal error. Nevertheless, the specification does not impose the restriction that the value for these header fields has to be handled as a UTF string. This is the cause for the UTFDataFormatException as the chosen method to write the string imposes a limit on its length [17]. Note that, choosing a different writing method would easily solve this issue. The same happens while setting a property name with an overflowed string (it does not happen while setting the value for the property). These are both level 2 compliance issues as they represent
non-severe problems related to what is defined by the JMS specification.

4.2. JBoss MQ 3.2.8.SP1 results

Legacy or older application servers are still used frequently in enterprises as upgrading a system to a newer version may have a high cost. In this sense we found interesting to test the previous JBoss MQ major version. With this experience we also wanted to verify if the serious robustness problem detected in the latest version also existed in the previous version. Again, the same Catastrophic robustness failure occurred in this version. In fact, the results observed (for both robustness and non-conformities) were exactly the same for both JBoss MQ versions tested.

Note that, even though this middleware is normally submitted to a large battery of tests (as stated by the developers), a major robustness problem and security vulnerability has passed silently between versions. This is something that could have been easily avoided by testing the middleware using our testing approach.

4.3. ActiveMQ 4.1.1 results

Apache ActiveMQ is a widely used open source JMS implementation, for which we were also expecting to discover a few or no robustness problems at all.

Testing the message properties and body revealed no robustness problems. However, a failure was triggered while setting the JMSMessageID to a predefined value. In fact, mutating the JMSMessageID of a single message posed no problem. However, tampering a second message (before delivering the first to the consumer and, consequently, removing it from the repository), caused the first message to be overwritten. Note that the JMS specification states that the JMSMessageID field contains a value that uniquely identifies each message. Overwriting messages represents an unacceptable behavior as it may lead to messages corruption and losses. After analyzing the source code, we concluded that this implementation uses this field as a unique identifier for the messages. Although this is an acceptable and natural approach, exceptional cases like duplicated values should be verified.

This failure was classified as Silent since an operation was incorrectly completed and no error was indicated by the JMS server.

The above robustness failure can also represent a security issue. In fact, security attacks that try to generate duplicate message IDs are quite easy. These messages will silently disappear from the provider’s repository and will never be delivered to consumers.

Similarly to JBoss MQ, some minor compliance issues were detected. JMSMessageID, JMSCorrelationID and JMSType are susceptible to the overflow mutation. In fact, this implementation imposes a limit on the size of these fields (\(2^{15} - 1\) characters) that is different from the one imposed by JBoss MQ (and both are incorrect as there is no limit defined in the standards). This same limitation is found in the size of the properties keys that, when violated, invalidate message delivery. Note that these are level 2 non-conformity issues as they correspond to non-severe restrictions to the standards.

Another compliance issue was detected while testing the message properties. The JMS specification states that [21]:

“The setObjectProperty method accepts values of class Boolean, Byte, Short, Integer, Long, Float, Double, and String. An attempt to use any other class must throw a JMSException.”

This was not the case with this implementation. In fact, it accepts a few other data types (Map, List and Date). This represents a level 3 non-conformity issue as it is clearly related to a specification extension. Note that a generic JMS based application may have problems if it is built upon the assumption that these data types are not allowed.

4.4. Additional comments on the results

Although only catastrophic and silent failure modes were observed, we believe that all the failure modes defined by the mCRASH scale are useful. Our results are obviously dependent on the JMS providers tested and cannot be generalized. Additionally, we believe that there are many JMS implementations where restart and abort failures are triggered when invalid parameters are provided.

Some preliminary tests on the latest Java EE 5 [20] JMS reference implementation (OpenMQ 4.1 [22]) exposed an Abort failure. In fact, setting an Object property using null as value exposes a robustness problem. This is caused by the fact that this implementation checks if the object type is one of those permitted by the specification. If it is not an allowed type (as is the case of null that has no type), then it tries to build a String (for logging purposes) with information of the object itself. This produces a null pointer exception as the null object does not contain any information.

5. Conclusion and future work

This paper proposes an experimental approach for the robustness evaluation of JMS middleware. Robustness testing focuses on abnormal conditions particularly frequent in complex environments. As messaging
middleware is increasingly playing an important part at enterprise application integration and development, the existence of an approach of this kind is of utmost relevance.

Our approach consists on a set of robustness tests applied to JMS messages, in order to expose robustness problems and possible non-conformity issues. A classification is defined to describe the middleware according to the severity of failures and level of non-compliance observed.

We applied our approach successfully to three major JMS implementations exposing some serious robustness problems. Some of the issues detected also presented severe security problems. These results clearly show that robustness testing can be applied to JMS middleware and can be of extreme importance when robustness and security is a major concern.

The above work focuses on the JMS message for introducing faulty elements. Future studies can include the extension of the robustness tests to other important resources in a JMS environment such as the connection, session, producers or consumers. These are all participating resources in JMS based applications, so in this sense it can be useful to study the robustness properties of each of these components (for example, applying mutations to the metadata of a connection object) and verify the response of the JMS provider.

6. References