MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

JMS-based test application for HornetQ fail-over testing

DIPLOMA THESIS

Bc. Martin Zruban

Brno, 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Martin Zruban

Advisor: Mgr. Marek Grác

Acknowledgement

I would like to thank Mgr. Miroslav Novak and Ing. Pavel Slavicek for help during the creation of this thesis.

Abstract

This thesis describes the JMS specification in context of high availability. It analyzes possible messaging scenarios and causes fail-over action in many of them. The tool developed in this thesis is able to automate this testing by performing start, shutdown and kill of JMS servers configured for high availability. Afterwads, it performs a series of tests which include a simulated fail-over event.

Keywords

JMS, HornetQ, testing, fail-over, Arquillian, Hudson, application server

Contents

1 Introduction

The ever-increasing need for reliable IT solutions has driven the development of enterprise computing platforms for the past several decades, which lead to the formulation of standards designed to accelerate development and delivery of robust projects. During this formative timespan, the Java Enterprise Edition (Java EE)[1] set of specifications established itself as one of the main options for prospective enterprise projects. This set includes the Java Message Service (JMS) as a specification for client-server communication.

Formal JMS specification defines the expected behavior of JMS implementations in many standard communication use cases. It does not, however, cover strict behavior in case of server failure. Modern JMS implementations offer high-availability of services by migrating client connections to another server. This action is known as fail-over. Due to lack of formal fail-over specification, every vendor is free to implement fail-over as it sees fit.

This thesis analyzes the JMS specification and suggests a series of tests which cause simulated failure or manual shutdown of a JMS service. The tests executed by the developed Failover testing tool are intended to map and compare the behavior of various JMS solutions. It can be used to compare their fail-over behavior.

The tool is developed to operate with custom configurations of different JMS products. Although many solutions are able to run as standalone, configuration of JMS servers is described here in context of application servers which have them pre-installed. Initial setup of several JMS servers is followed by running standard JMS scenarios. This includes establishing client-server communication, sending and receiving messages and validating their data. Transactions and manual acknowledgment of messages are also covered in their own test cases. A fail-over event might occur at any point in these scenarios and cause unexpected behavior, e.g. loss or duplication of messages. Scenarios with multiple fail-over events are described here as well.

Execution is developed as automated for different JMS solutions, assuring consistency of fail-over behavior among different version of a JMS solution. Developed tests are designed to be included in a continuous integration system called Hudson[2]. Eventually, it can be

used in quality assurance to discover software bugs and in as helper tool in designing a JMS client which could ensure consistent fail-over behavior with various JMS implementations.

2 High availability and Enterprise Java

As more and more enterprises include e-commerce[3] in their business model, their products and services become available to the general public. In order to keep their competitive edge, these companies have to deal with current computational and networking performance limitations. Global and local enterprises have to offer sufficient quality of service in order to protect their brand identity and prevent loss of revenue when their operations are unavailable. The duration of this is known as downtime[4]. The need to minimize it inevitably lead to high availability (HA) approach in design phase of solution development.

HA guarantees that certain level of operational availability will be met for some period of time. If arbitrary component fails or reduces its performance below contractual standards, another one can resume its duties because functionally equivalent services are present on multiple nodes of a distributed system. The basic attributes of individual nodes in HA are to fail fast, signal shutdown to other nodes and allow failure detection by providing a heartbeat signal[5].

2.1 Types of redundancy

HA in arbitrary client-server communication models in general is only possible by introducing redundancy with one or more additional server nodes. These may or may not be active themselves. Failure of arbitrary node to provide sufficient quality of service should result in migration of client connections from active node to a backup in a timely manner. Different systems can have different behavior of backups before primary nodes are deactivated, whether by manual shutdown or failure. Using Shooman's terminology[6] to place backup nodes into several categories:

1. Parallel - backups participate actively in HA cluster. They have a single monitored active node. In the event of shutdown, they take over its workload.

2. Standby - are passive until deactivation event. Redundancy is achieved via master-slave architecture where a backup is acti-

vated by failure of the master node. This offers the option of multiple backups in standby mode, waiting for an active node to fail.

Although systems using either type of backup nodes can be described as clusters, it is important to distinguish between clustering for scalability and clustering for HA[7]. Clients connecting to a HA cluster can use backup nodes only after the original node failed, whereas clustering for scalability allows load-balancing of connections among multiple active nodes.

When original server is functional once more, clients might want to switch back. This behavior is referred to as fail-back. A node can be brought down administratively, for example due to maintenance. Clients can even then optionally perform fail-over, known as fail-over on shutdown.

2.2 Java Enterprise Edition

Development of a custom HA solution for every project would be similar to reinventing the wheel. To implement HA from scratch is both costly and ineffective and the resulting implementation would be inevitably riddled with bugs typical for young projects[8]. As with any complex system, it is optimal to choose already developed components where possible. Therefore, the best way for any service to offer HA is to use existing frameworks. In case of bigger projects, this is usually done via middleware[9].

Project development also needs to take into account the human factor. Potential of using middleware frameworks in arbitrary project is limited when the developers do not have enough experience in using it. The learning curve of its API and programming language should be minimal. Therefore, the frameworks used the most are necessarily those easiest to master. The time needed to achieve at least partial competence can be reduced if the middleware uses a language with which some team members are already familiar. In this respect, Java is one of the best choices[10].

The most stable middleware platforms are usually the ones based on formalized standards. There is a mechanism for developing standard technical specifications for Java called the Java Community Pro-

cess (JCP). Its main function is to review formal documents called Java Specification Request (JSR) identified by a number[11]. They describe details of proposed technologies to add to the Java platform, with dozens of specifications already approved. Some of these describe various versions of an enterprise platform for the Java language called Java Platform, Enterprise Edition (Java EE).

This specification contains multiple supporting technologies, each with specification of their own. It also defines reliable and asynchronous messaging through specification called Java Message Service (JMS)[12]. It describes compatible client interfaces through an API and expected behavior of server-side part of JMS. Although the most recent version can be considered 2.0[13] and was recently approved by the JCP, most modern implementations offer compliance with version 1.1 of JMS (JSR 914)[14].

2.3 JMS communication model

Communication in JMS happens by passing lightweight units of data encapsulated as *Message*[15] objects between two groups of clients. One of these creates and sends messages - its elements are called producers. The other are called consumers and their primary purpose is to receive and read messages. JMS uses a server-side intermediate called provider which holds messages and distributes them to clients. It is responsible for ensuring once and only once delivery of messages to consumers, ensured by message acknowledgment performed on the application layer of the OSI model[16].

Client communicate by exchanging messages with a provider-hosted destination. This functions as a logical channel of communication between producers and consumers. A message can be set as persistent to indicate that it should survive server shutdown. In such case, it is stored by the provider once received and available in case of fail-over.

JMS clients needs two objects to start communication. One of these is *ConnectionFactory* and is responsible for creating connections from client to provider. The other one is *Destination* and specifies the address of a message. According to JMS convention, they are created in client code by performing Java Naming and Directory

interface (JNDI) lookup[17]. This means that the two object have to be set up administratively in the provider's configuration before JMS clients start using them.

3 Persistence in JMS providers

Many different JMS vendors present their solutions as fastest and most scalable. Performance aside, many of them also claim to be reliable. Individual providers approach their persistence in different ways.

This work analyzes several of these solutions. They are selected as representatives of various HA approaches, differing in whether or not their backups also act as active nodes, have their own persistence and how they store data. All of selected solutions are developed as open-source projects under various licenses.

3.1 Types of persistence

In JMS, individual providers need to store different data based on their level of fail-over support. Typically, this includes persistent messages, states of destinations, connections etc. All this information has to be stored in some form of persistence. Providers can be grouped into three main categories by the type of their persistence engine:

- database - connect to a relational database management system (RDBMS) via JDBC interface[18]. While architecture of tables is provider specific, administrators can existing DB solutions for maintenance tasks.

- journaling - custom files and directories are stored on filesystem and accessible from all backup nodes. This structure is referred to as journal.

- mixed - some providers allow connection to either database or journal. It is up to the user to decide.

After a backup is activated, it needs access to persistent data in order to function. Whether or not the backup operates on its own data is decided by its persistence mode:

- shared store - uses existing data storage of original node.

- replication - backup node has its own persistence which are a copy of master's persistence. Data are periodically synchronized, typically through network traffic.

Generally, shared data is the preferred HA solution for a number of reasons[19]. Replication requires a sophisticated mechanism for synchronization of persistent information between two nodes, since many sections of persistence may be changed concurrently by multiple threads. JMS vendors might deal with this by not synchronizing the whole persistence, which impair reliability. Also, even when the information transmitted between two nodes in a HA cluster has to traverse only a short distance, this still presents a potential failure point where the slave did not manage to synchronize the full persistence of master node before failure.

3.2 JBoss Messaging

JBoss Messaging (JBM) is a JMS solution developed by JBoss, a division of Red Hat, Inc. It uses multiple active nodes functioning in parallel as a HA cluster so that when a node fails, another already active node takes over its responsibilities. All of these are connected to the same shared database with JMS data divided into multiple tables. In the event of a node failure, another active cluster node takes responsibility for the messages. Chain of responsibility among nodes is decided during cluster formation[20].

JBM uses RDBMS for storage of persistent data. Default database is HyperSQL Database Engine, which is not usable for HA. Other persistence options suitable for production are possible via configuration templates for many popular databases and support for new ones can be added by adjusting the data definition language[21] code in persistence configuration[22]. JBM creates several tables in a shared database if they are not already present (JBM_POSTOFFICE, JBM_MSG, JBM_REF etc.) and these are later used by all nodes in the cluster.

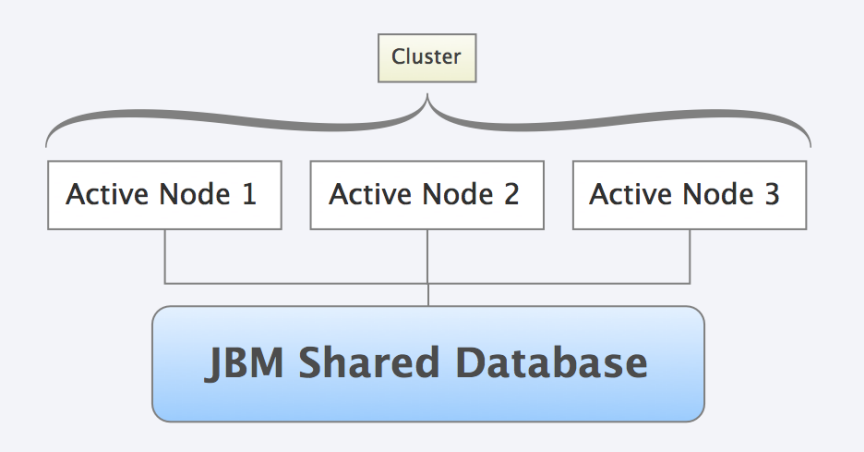

Figure 3.1: *JBoss Messaging nodes connecting to a single database.*

3.3 HornetQ

Version 2.0 of JBM introduced many differences from previous 1.x versions. To prevent confusion, it was renamed to HornetQ and development continued from version 2.0.0. Fail-over behavior relies on standby backup node. In fact, multiple backups are possible by creating live-backup groups, with many passive nodes watching a single active one. Fail-over on shutdown and fail-back are available as well. Unlike JBM, HornetQ offers the option to run as a standalone service without application server.

Usual approach is for backup nodes to access persistence via a shared journal as shown in Figure 3.2. When using shared store, typically it would be mounted on a Storage Area Network (SAN). HornetQ user manual does not recommend using Network Attached Storage (NAS) for performance reasons. Journal is separated into several sections which can be divided into different locations to optimize performance.

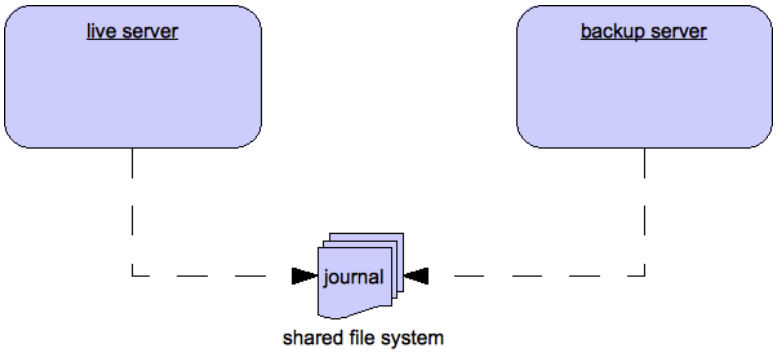

Figure 3.2: *HornetQ live-backup groups sharing a single journal[23].*

3.4 ActiveMQ

ActiveMQ is a JMS solution from Apache. Project documentation describes a single instance of JMS provider as "broker". HA is achieved by master-slave standby architecture[24].

Persistent data owned by the master are stored in either shared journal or database. Multiple slave nodes are waiting to obtain an exclusive lock. When master broker dies, a slave takes the lock and becomes the new master. Clients cannot automatically migrate connections to master once it is live again. RDBMS support is included for most common SQL solution databases[25]. Since version 5.4, journaling uses KahaDB persistence, a performance optimized file-based database[26].

4 JMS specification overview

Typical JMS client scenario starts by using JNDI service to lookup creating an *InitialContext*[27] object using custom properties. This is used to lookup a *ConnectionFactory* which in turn creates *Connection* instances for communication with the provider. Both *InitialContext* and *Connection* can be initialized with optional security settings. When using authorized connection, a client may connect to JMS provider under a role and send/receive messages from secured destinations. These roles can limit functionality of client, e.g. a client may receive messages from a destination, but not send. A client can set *ExceptionListener* on connection for handling special events such as broken connection, node failure, shutdown etc.

Once a connection to provider is opened, it can be used to create multiple *Session* objects. A *Session* creates *Message* objects of various types and its the primary context for transactions. A session is not designed for concurrent use by multiple threads.

Sessions can be instantiated with six possible combinations of constructor arguments. However, acknowledgment of transacted sessions is handled by it's *commit()* and *rollback()* methods so that only four possible transaction combinations are viable[28]. As a matter of fact, JMS implementations of all providers internally recognize four types of sessions[29]:

- AUTO_ACKNOWLEDGE - provider handles acknowledgment of each message automatically, without explicit client action.
- DUPS_OK_ACKNOWLEDGE - lazy automatic acknowledgment. Although faster, this mode allows duplicate messages in event of provider failure.
- CLIENT_ACKNOWLEDGE - messages are acknowledged explicitly by client's invocation of *Message.acknowledge()*, which acknowledges all previous messages received by the session. This session mode is only reasonable for consumers. If a message could not be processed, clients use *Session.recover()* to restart session with last unacknowledged message.
- SESSION_TRANSACTED - delivery of multiple messages is

handled by an explicit commit or rollback. All produced messages in this session are grouped into an atomic unit of output and all consumed messages into an atomic input unit. Commit sends the outgoing unit and acknowledges the incoming unit, while rollback destroys output and recovers consumed messages.

JMS vendors may offer the option for clients to participate in distributed transactions. Although this feature is not required by the JMS specification, it is available with most modern providers. The JMS specification states that if supported, a Java Transaction API (JTA)[30] aware provider must do so using a *XAResource* object[31]. This object interacts with an *XASession* and provides sophisticated methods for working with multiple transactions, including two-phase commit and recovery. This can be used when message processing work interleaves with database transactions.

4.1 Messages

Messages are created from a Session object. A message is composed of three parts:

- Header - set of fields defined by JMS including destination, delivery mode, message ID, priority, expiration etc. They contain identification and routing values used by both client and provider. Several header fields have direct impact on delivery and ordering of messages:
  - *JMSDeliveryMode* - can be set to either PERSISTENT or NON_PERSISTENT. A JMS provider has to ensure that persistent messages are preserved against server failure.
  - *JMSExpiration* - is a sum of time-to-live value and current GMT value. Once current GMT reaches this value, these messages should not be delivered.
  - *JMSPriority* - assigns values from 0 (lowest) to 9 (highest). A provider is not required to strictly implement priority ordering.

- Properties - client can set properties for messages. These can be used to filter incoming messages.
- Payload - message body, may contain arbitrary data. Although message without body is also viable, client usually communicate using one of five message types:
  - TextMessage - message is set text represented by a *String*.
  - BytesMessage - contains a stream of uninterpreted bytes as a byte array. Data are written and read sequentially through an interface similar to *java.io.DataInputStream* and *java.io.DataOutputStream*.
  - MapMessage - consists of multiple key-value pairs. Values may be various Java primitive types, String or a byte array. Key must be a *String*.
  - ObjectMessage - used to send a single object which implements the *java.lang.Serializable* interface.
  - StreamMessage - contains multiple primitive data types. Although they are written and read sequentially (much like *BytesMessage*), this message stores type information so that reading incorrect type will result in an exception[32].

A client can also communicate using bodiless messages. In this case, only information carried by header fields and properties is available.

4.2 Destinations

Usually, producers and consumers are created targeting specific destination[33]. Depending on intended number of recipients and messaging model, JMS defines two messaging domains:

- Queue - works as point-to-point messaging. Any messages in queue is consumed by single consumer. If multiple consumers receive messages from a queue, they are redistributed among them according to provider's discretion.
- Topic - acts according to the publish/subscribe principle. Messages are distributed to all connected consumers and any durable subscriptions, as discussed later.

Transmission of messages themselves is performed by producers and consumers, which are also created by the same Session. If the selected destination is a queue, the producers and consumers are referred to as sender and receiver, respectively. For topics, they are called publisher and subscriber.

JMS destinations are typically created by an administrator via provider's configuration interface. However, standard JMS API allows creation of temporary queues and topics. These are available only for the duration of the connection used to create them.

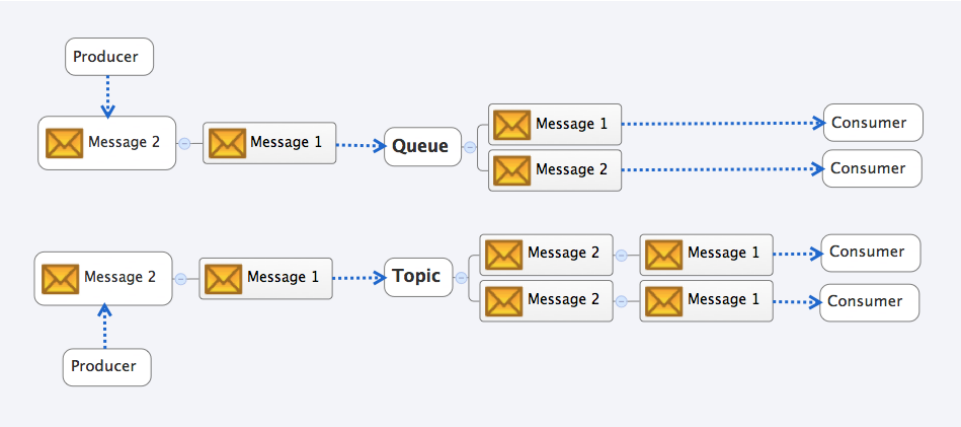

Figure 4.1: *Illustration of queue and topic messaging domains.*

4.3 Producers and consumers

Once a message is created, the next step is to create producer to transmit it to a specific destination. A consumer is created on the receiving end, either before or after start of transmission.

Consumers can be limited to select only certain messages from a destination. This filtering is handled by the provider. It is done by setting a message selector, which performs evaluation operations on header and properties fields using conditional expressions. Their syntax is based on a subset of SQL92[15]. Additionally, topic consumers can set an optional *noLocal* attribute. If true, they will not receive messages sent from the same connection.

Standard behavior of topic consumers is to receive messages only after they are created and before they are closed. If no message can be missed, they must be created using a durable subscription. When such a consumer is closed, provider stores messages for it until it becomes active again. A durable subscription is identified by its name, topic and the *clientID* parameter of connection[34].

Message reception is handled by class *MessageConsumer*. In synchronous mode, a message is by calling the *receive()* method, optionally invoked with a timeout indicating how long it should wait for a message. Asynchronous approach requires setting a *MessageListener* with *onMessage()* method responsible for message processing. A consumer may use only one of these methods, since they are mutually exclusive. Either way, the connection must be instructed to begin message reception by invoking the *start()* method.

A special type of message consumer is *QueueBrowser*. It allows to read messages waiting on a queue without removing them using a *java.util.Enumeration* object. It can also be created using a custom selector, similarly to standard consumers. Viewed messages do not have to conform to a static snapshot of the queue content due to expiration, priorities and arrival of new messages.

4.4 Integration with application servers

The JMS specification defines facilities to be used by an application server, which allow concurrent processing of messages. Connections can create a *ConnectionConsumer*, specified with a selector and a *ServerSessionPool*, from which it takes *ServerSession* objects. Basic idea is to allow application server to optimize message consumption by caching connections when using a JCA resource adapter.

While it is possible to use pure JMS approach, most applications take advantage of this concurrency by using message-driven beans (MDB)[35]. They run within an Enterprise JavaBean (EJB) container and are event-driven, stateless and asynchronous. An MDB has to be marked with the *@MessageDriven* annotation. Although primarily meant for consuming messages from the same server instance on which they were deployed, MDBs can also receive messages from remote destinations[36], although this requires additional configura-

tion. Acknowledgment of messages is handled either by container-managed transactions, or by the bean itself when using bean-managed transaction demarcation.

5 Modelling fail-over scenarios

In the most simple JMS messaging scenario, a client starts a connection, sends messages, receives them and ends. At any point, the provider may experience failure and client connection needs to fail over to backup node. Afterwards, it may experience problems when executing other operations, even if connection itself is reestablished successfully. Messages may be lost, transactions cannot be committed etc. Typically, objects throw an *Exception* when client tries to use them. For remote clients, these objects are: *ConnectionFactory*, *Destination*, *Connection*, *Session*, *MessageProducer*, *MessageConsumer*, *QueueBrowser*, *Message* and all their descendants. The functionality of a JMS object may be impaired in three ways:

- complete failure - unable to perform basic functionality, trying to do so throws an Exception.
- partial functionality - some but not all functions work as expected. Problems occur either in test subject or in JMS objects it creates.
- success - fail-over did not impose any limitations, objects can be used without additional action, as if nothing happened.

This testing tool is designed to run tests to ensure that client and provider survive fail-over without problems. State of both may be tested by causing fail-over in various steps of a messaging scenario and afterwards receiving data, checking that no message was lost or duplicated. Testing full functionality of arbitrary JMS object created by client is inherently complex and takes a long time to complete, if all its possible uses are supposed to be verified. Therefore, their execution is only reasonable after verification of objects ability to perform basic operations. The tool developed in this work performs two groups of tests:

- basic tests - verification of basic functionality by using object in simplest possible messaging scenario. They adhere to the fail-fast principle and answer a basic question: after fail-over happens, can the test object be used at all?

- validation tests - multiple complex scenarios are used to validate that test subject's conformity to JMS specification. Multiple JMS operations are performed on these objects to check that it and its descendants perform all operations correctly.

Basic tests run a "simplest messaging scenario" implemented as sending and receiving a simple *TextMessage*.

5.1 Simple messaging scenario

Once *ConnectionFactory* and *Destination* are specified, hierarchical nature of JMS objects allows the modelling of possible scenarios as a tree. *ConnectionFactory* can be considered root, every following vertex an arbitrary action and edges potential fail-over events. A single branch represents one messaging scenario and has at most one failure event.

Diagram below represents a basic JMS scenario. *ConnectionFactory* is used to create *Connection*, which creates a *Session* and so on until some messages are sent. At this point, a fail-over event may occur and when client tries to receive the messages, various problems may be revealed.

However, they may happen before the whole scenario is finished. If fail-over occurs after *ConnectionFactory* is initialized, it may not be able to open a new *Connection*. In such case, *ConnectionFactory* is considered to be the test subject and basic test fails. Using this line of reasoning, any of the steps in a JMS scenario may be followed by a potential fail-over event. For this simple diagram it means seven potential scenarios, each with its own unique test subject. To use Figure 5.1 as reference, all edges marked by X could be replaced with a fail-over fork.

5.2 ConnectionFactory subject

The first test is run without a test subject and with an administered *Destination*. This basic test ensures that both JNDI objects are configured correctly. Next one is basic test with *ConnectionFactory* as subject. This case is represented by the first potential failure point

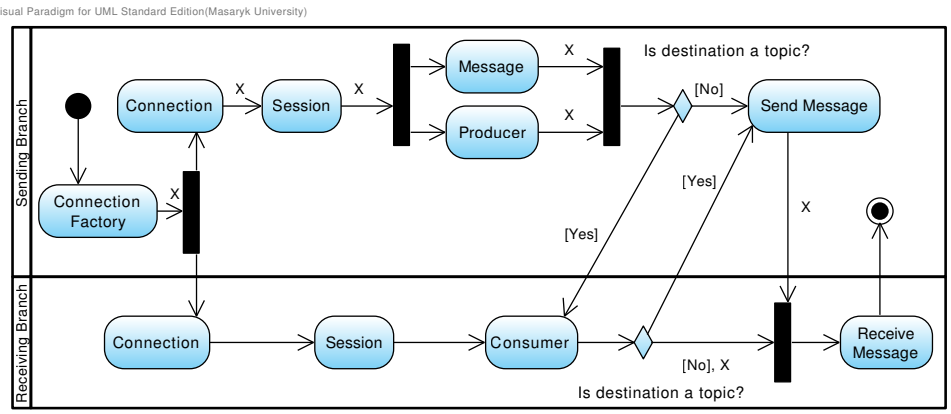

Figure 5.1: *A basic JMS messaging scenario. Nodes directly affecting the destination are in italics, potential failure events are marked with X. Note that when the destination is a topic, a Consumer has to be created before sending a message.*

in Figure 5.1. After the factory is created, fail-over occurs and subsequent *Connection* object connects to backup node. Success of this test proves that HA behavior is enabled on *ConnectionFactory*.

Failover testing tool recognizes two moments when to lookup destination - before lookup of *ConnectionFactory* and immediately before creating producers and consumers. When choosing the first way, *Destination* is also subject of basic *ConnectionFactory* test. If it is decided that this object does not survive fail-over, the second method is used.

Destination object can be tested with this scenario as well, if it is looked up before fail-over. After fail-over occurs, it is used by *Session* when creating producer and afterwards consumer. If it could not be used, the test for *ConnectionFactory* would be able to create a new *Connection* but would fail when trying to send a message.

Besides testing administered destinations, JMS tests can be executed using temporary destinations. They should remain present after fail-over, if the *Connection* used to create them survived. This tool runs individual tests for both types of destinations.

Results of this basic test depends on correct provider configuration and can be considered a basic validation tool. If any of these ad-

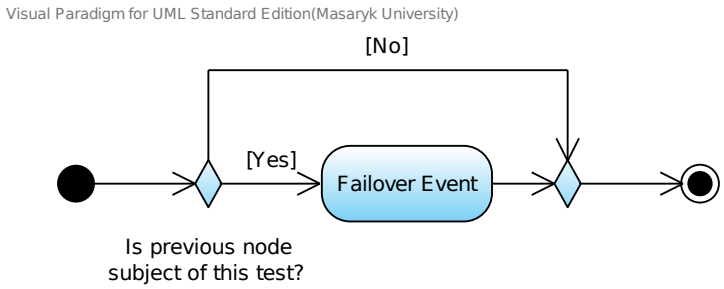

Figure 5.2: *Adding a possible fork to every marked branch.*

ministered objects is not reusable, subsequent tests try to look them up anew.

5.3 JMS subjects

Usually, each JMS object can be created using various attributes, which adds various forks to original graph in Figure 5.1. Given a *ConnectionFactory* and a *Destination*, a connection is created using optional authentication parameters. Next, a session object is created. In JMS, all four types of *Session* objects have slightly different behavior. Validation tests are executed for all four of them.

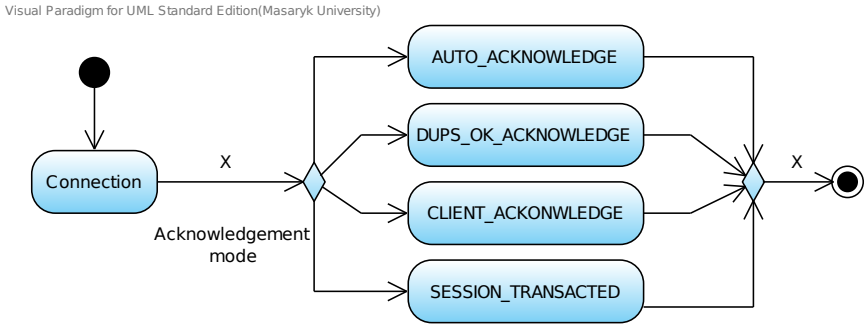

Figure 5.3: *Forks added to potential scenarios depending on what session type is tested.*

Producers and consumers are created from *Session* objects. The

only parameter used to create a producer is *Destination*. It is stored inside in the producer object and used when the *send()* method is invoked, so there is no need to perform additional test for producers constructed with destination set to *null*.

Consumers can be created with many different parameters, starting with selectors, which can be set both for queue and topic consumers. Syntax of selectors is quite complex and offers many variations. Therefore, this tools creates consumers with various selectors and creates consumers for each of them.

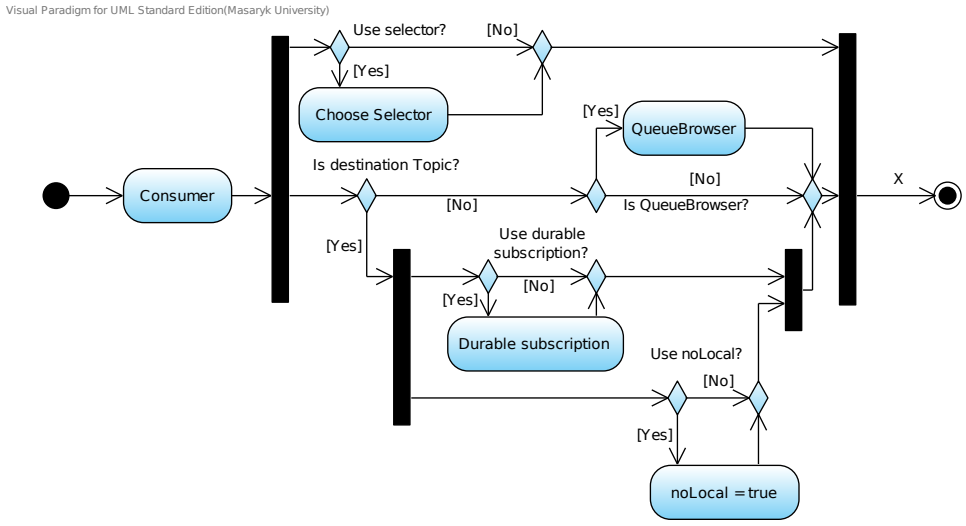

Figure 5.4: *Multiple ways to create a consumer.*

If the destination is a topic, consumers can be configured with two more additional parameters. They can ignore messages sent from the same *Connection* by the *noLocal* attribute and they can be set created as durable to continue receiving messages after they are closed. This expands the *Consumer* node of original diagram some more, as shown in Figure 5.4]. A *QueueBrowser* can be also considered a special type of consumer, even though it does not implement the *MessageConsumer* interface used by consumers.

5.4 Attribute noLocal

When tested destination is a topic and the consumer to be created in this branch uses the *noLocal* attribute in tests, a new dedicated *Connection* needs to be created for sending messages. A consumer needs to use a different connection to receive the messages then the one used to produce them. The original *Connection* object used by the producer is then used to send a single control message for purposes of negative testing, to verify that this consumer indeed does not receive messages from.

6 Message factories

When the testing tool starts transmission, it changes the set of messages awaiting retrieval in destination. After fail-over, some or all of these messages may be lost. This can be caused by failure of backup to connect to the whole persistence of original node or to some its partitions, if it is divided into multiple sections. This is possible regardless of provider's approach towards accessing persistent data, be it shared storage or replication.

A message can be stored in different locations within persistence engine according to various attributes, most common of which is size. Since every provider runs with only a limited amount of memory, they often choose to store large messages separately[37]. Insufficient RAM may be a problem with great number of smaller messages as well. Provider usually stores some messages in a paging location. Selecting which ones is decided by provider's own heuristic evaluation. Validation tests for all are designed as sending of multiple messages created by a *MessageFactory* before causing fail-over event.

6.1 Recognized types of messages

Basic testing of a JMS object should verify that it can be used to send a simple message. This is done with a simple "Hello world" *TextMessage*, without additional header fields or properties. However, validation testing needs to be implemented for messages of all types and sizes. Some of these messages should also have header and properties set to different values in order to test their filtering with selectors.

To correctly implement all possibilities, multiple messages need to be sent to a destination and then validated after fail-over event. Fail-over tool creates one message for each combination of these attributes:

- size - message body is filled with different amount of data. Testing recognizes two sizes of messages:
  - small - body filled with only small amount of data or none at all. This can be anything from a single character to a custom arbitrary number.

  - large - body is filled with random data to meet certain size. Default value is 50MB.

- type - all five data types are created with additional message without a body. A bodiless message can be only small, since there is nothing to fill.

Messages are created by a *MessageFactory*, which equips them with a custom identifier. These are stored in this factory and later used to verify that no message was lost or received out of order.

6.2 Selector tests

Language of JMS selectors offers flexible syntax to inform the provider which messages it wants to receive. A selector statement can reference several header fields and any property of a message. Header fields are restricted[15] to *JMSDeliveryMode, JMSPriority, JMSMessageID, JMSTimestamp, JMSCorrelationID* and *JMSType*. Properties can be set using all Java primitives, primitive object types and String objects. Because the number of every combination of header field and property type is finite, creating a selector expression for them would be possible, albeit impractical.

This tool uses several predefined selector values for creation of various messages. Each of those is supplied with a combination of header/property values for both true and false evaluation with the tested selector. If specified value is a property, it is defined along with its type. A message is then created with combination of each of these values.

When performing validation testing, this tool adds possible selector values for each combination of message type and size. They are transmitted in sequence and referred to as message unit in context of this work.

In a single message unit, there are at least ten large messages and more, if multiple selectors are used. In such cases, it is expected that some of these messages cannot be stored in available memory and are stored in paged locations. Using this many large messages then proves that paginated data survive fail-over as well.

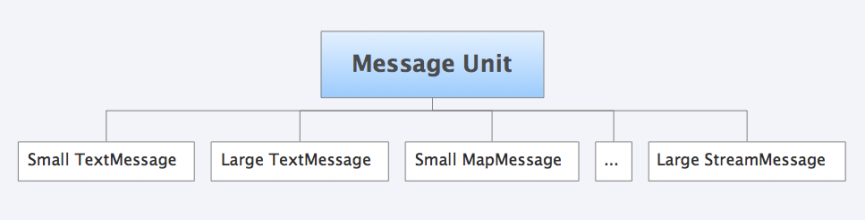

Figure 6.1: *A message unit.*

6.3 Messages with expiration

A message waiting to be consumed may surpass an expected time to live value and expire. Typically, a provider runs a daemon that periodically removes messages pending on a destination. When it finds a message, it can either remove it completely, move it to a special destination or use arbitrary tagging mechanism. Testing tool recognizes two options when this removal can happen - before fail-over event and afterwards.

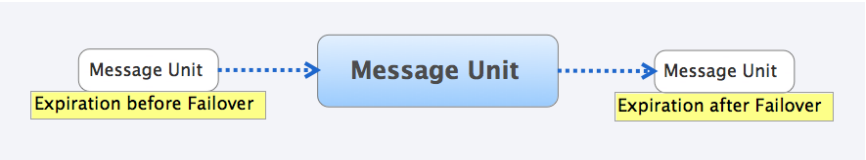

Figure 6.2: *Adding before and after expiration units.*

Expiration is tested by creating two message units with different time-to-live (TTL) values. One of these is set to expire before fail-over event, the other one after it. Expiry daemon usually runs in specified intervals to minimize impact on performance, so the two message units need to be send in sufficiently separated intervals.

7 Send and receive scenarios

In some cases, fail-over happens between two various message sends or receives, or during them. Afterwards, producers and consumers may not be able to send and receive more messages or finish a transaction. These cases differ from automatic acknowledgment modes and require additional test scenarios.

Validation of received messages is performed during the receive phase of accepting messages. A *MessageFactory* object keeps identifiers of all created messages. This is a number that is incremented continually, starting with a random number for each factory. During reception, an internal counter keeps track of already received messages. This counter lets the verification mechanism know which message is expected as next. When a message arrives and its identifier value is less than expected, a duplicate is logged. If it is greater, a message was either lost or received out of order.

7.1 Transacted sessions

When communicating with a transacted session, fail-over can happen before invoking the *commit()* or *rollback()* method. The state of uncommitted transaction may or may not be carried over to backup. Even if it is, provider may not be able to send or receive more messages within this transaction when it is marked as rollback only. Additionally, the ability to start a new transaction on the backup may be compromised. Similar rules apply when receiving a message with client acknowledgment session mode.

A consumer may start reading messages only after connection is started. Once this is done and messages start arriving, most JMS providers introduce a buffer to hold at least one message for synchronous receive. If no message is ready when receive is called without a timeout, null is returned. Failover testing tool uses both reception modes when consuming messages, by implementing the *MessageListener* interface or invoking *receive()* periodically.

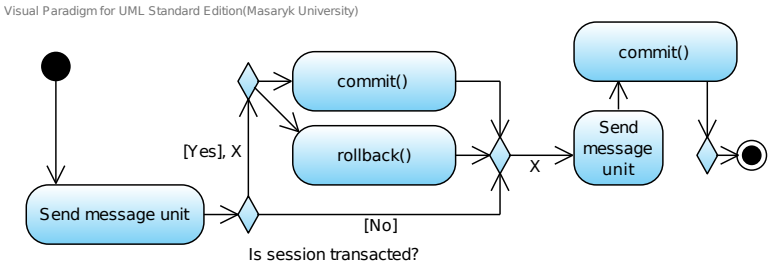

Figure 7.1: *Potential failure event when sending a message with transacted session.*

7.2 Concurrent execution

Once a message is sent or *Connection.start()* is invoked, state of destination is considered to be changed. In some scenarios, fail-over happens before starting traffic on a destination. Subjects of these tests can be created simultaneously in a single execution and use a common fail-over event. This includes test subjects: *ConnectionFactory, Connection, Session, Message, Producer, Consumer*.

When sending messages in an arbitrary branch, they are created using a specified *MessageFactory*. Besides equipping them with custom identifier to check that they were all received in proper order, messages are tagged with the name of the branch used to send them, so that verification of messages is not confused with messages generated by foreign test scenarios.

Some scenarios use consumers created with selectors for receiving messages. To perform concurrent execution using single destination, the receiving nodes of branches would receive messages from foreign branches as well as their own. If the destination is a topic, this is solvable by marking the messages with identifier of branch used to send it. This solution would also work if the consumer is supposed to be *QueueBrowser*.

7.3 Bucket for queue messages

When a message is removed from a queue, no other test branch is able to receive it. In this regard, receiving from a queue can be viewed as destructive operation. This is true for standard consumers as well as consumers with selectors, since some messages may pass the filter of more than one selector. To run queue tests sequentially would greatly increase the time needed for overall execution.

A simpler solution is to mark a message with branch name like with topics, create a common bucket for all messages received by queue receivers. There they can wait to be picked up by validation phases of individual branches later.

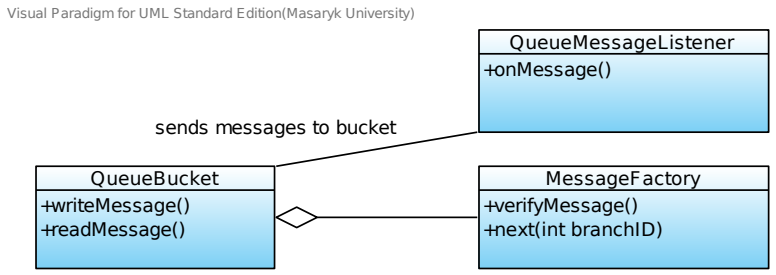

Figure 7.2: *Messages received from a queue are put in a common repository, from which they are selected in verification phase.*

When fail-over happens during receive phase but before manual acknowledgment, some of these messages could be present twice in both topic and queue tests. The JMS specification states that in such case, the *JMSRedelivered* flag should be set, although this behavior is not guaranteed. This tool does not verify the state of this flag. It does, however, expect messages received before fail-over without acknowledgment to be received twice.

Implementation of bucket has to take into account that it will be accessed from multiple threads. Passing messages directly into this bucket via *synchronized* methods would be relatively slow for large messages. For this reason, messages first go through an adapter which first clears their body before they are passed to the common bucket.

7.4 Fail-over during send and receive calls

A fail-over event can also occur during send and receive operations. To cover this case, a special kind of message is needed. Its transmission has to take a certain period of time during which the provider is killed. This requires two mechanisms:

- separate thread - creation of message and *send()* are executed in a new *Thread* object.
- bloated message - a message of any type whose body was filled with random data to meet an arbitrary size.

Calculation of bloated message size is done before start of actual test. Messages of ever increasing size are transmitted until the time needed to transmit them reaches a desired value. Although length of transmission is configurable, it should be enough to cause fail-over. Default target values are 1 second for server kill and 5 seconds for clean shutdown.

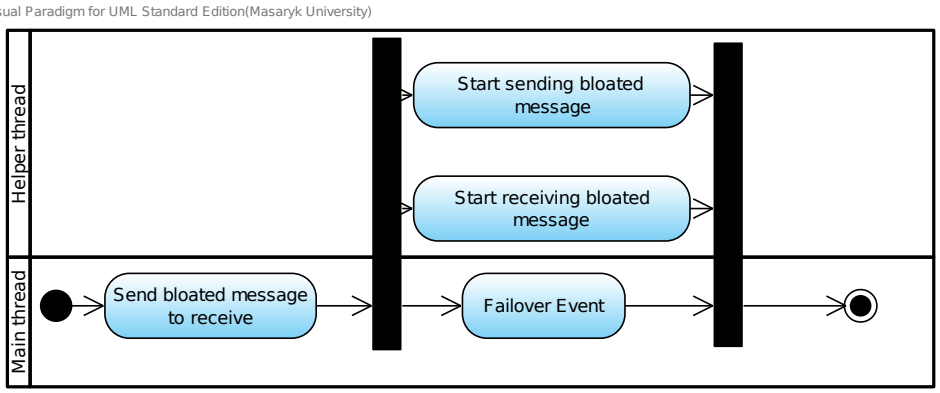

Figure 7.3: *Send and receive of a bloated message. Fail-over happens during transmission.*

8 Advanced testing scenarios

Before execution of any messaging scenario, client-server communication needs to be established. A client can connect to a remote JMS provider from a standalone application or from within an application server. All testing scenarios developed with this tool can be run from within a web application as well.

Another important fail-over features are the time it takes to complete and possibility to repeat the fail-over action. This can be influenced by provider configuration, among other factors. Fail-over testing tool measured the time needed for a fail-over as well as simulates multiple fail-over and fail-back actions in a row.

8.1 Connecting from application server

When a client connects to a provider from an application server, it can either connect to a remote provider or a local one. To run this fail-over tool while connected to provider hosted on the same server would require crashing the JMS portion of the server while leaving the rest intact. Much easier way to implement this is by connecting to a remote provider and targeting it.

Modern JMS solutions use different *ConnectionFactory* objects for in-VM and remote communication. When destination is not located on the same AS as running application, similar rules apply as with remote clients.

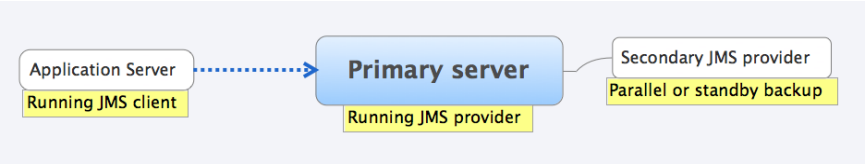

Figure 8.1: *Connecting from client in an application server to remote provider.*

Using JMS from inside the application server lets the client use optimization techniques offered by JMS application server facilities.

Server-side testing recognizes another test subject created using a *Connection* object, namely *ConnectionConsumer*. Testing tool uses also durable version of this object if the targeted destination is a topic. It uses the same rules for choosing message selectors as other consumers. As *ServerSessionPool*, it uses as default a custom implementation which creates a new *ServerSession* with every call. By default, *ConnectionConsumer* is created to have the maximum of ten messages assigned to a server session at a time.

8.2 Timing thread

To measure how long a fail-over takes, a simple message with timestamp is send in regular intervals. The timestamp is set using the default *JMSTimestamp* header field. On receiving end, a consumer is created to receive the timing messages. These are marked with a special flag and the consumer is created using a selector. When a message is received, the difference between message's timestamp and current GMT time is calculated and later dumped in an output file.

This whole communication happens in a separate thread. If the timestamp message is received by another consumer, it is dumped into the common bucket like other queue messages. When the producer is created, the tool verifies that timestamps are enabled on sending producer by calling the method *getDisableMessageTimestamp()*. This timer cannot run with tests verifying fail-over during send and receive operation. Therefore, this timer sends messages to a different destination than the one currently tested.

8.3 Linking multiple fail-over events

Failover testing tool implements several ways to bring down a node in a HA cluster. These actions can be performed on different nodes, depending on their role in testing. Together, they define a failure event. Tested cluster nodes belong to at least one of these groups:

- active - test uses it as primary node to send messages.
- primary backup - a connection fails over to this node. When

HA is achieved by connecting to another active node in a cluster (such as JBM), this node is meant.

Causing multiple failure events may cause errors that are not so apparent after only one fail-over occurs. This tool also performs tests which cause multiple server failures in a row. These follow the same rules as those with a single fail-over performed on active node.

9 Design of testing tool

To make suggestions on application's design, let's reiterate the requirements and features of Failover testing tool. It's purpose is to simulate JMS scenarios of real applications and test arbitrary solution's ability to provide continuous service after failure of one or multiple nodes. Persistent messages should survive this failure. Any testing of cluster fail-over ability begins with start of a HA cluster. It is possible to do this manually but it would require fair amount of scripting and could not be provider-independent. It would greatly simplify matters if this tool could automate the start of the HA cluster. A simple solution is a project called Arquillian designed for Java EE testing.

9.1 Arquillian

This project is developed by JBoss to test web applications inside servlet containers and entire application servers[38]. It offers many useful features, main of which are test-driven deployment of archives and controlled start and shutdown of containers.

Declaration of containers is stored in resource file named *arquillian.xml*. Multiple containers can be aggregated into named groups, allowing existence of identically named containers in different groups. Arbitrary tests can then be run with identical container start-up sequence, regardless of which provider is being tested. Testing group is selected by the *arquillian.launch* environment property.

It is possible to start arbitrary container before testing automatically for each test. However, most scenarios in this tool require specific start-shutdown sequence. This can be done by selecting single Arquillian container group and invoking methods on a *ContainerController* object. Its initialization is handled by Arquillian.

Arquillian is designed to be used in conjunction with one of two popular testing frameworks, JUnit and TestNG. While both offer rich variety of features, this application is implemented with the former one. Readers should bear in mind that this choice of testing framework is motivated purely by developer's preference. To run JUnit 4 tests with Arquillian, they need to be marked with *@RunWith* anno-

```
@RunWith(Arquillian.class)
public class SomeTest{

@ArquillianResource
ContainerController container;

...
container.start("target"); ...
```

Figure 9.1: *Using manual container management from tests.*

tation as shown in Figure 9.1.

Server-side tests are implemented using a *WebArchive* object generated by the ShrinkWrap framework[39]. This archive is created dynamically and deployed on target server. It contains all classes needed to run the tests. Thus, through integration of Arquillian and JUnit frameworks, archives can be created after compilation of the test classes and before execution.

9.2 Dependency management

Final application is dependent on several third-party libraries. These include libraries for Arquillian, JUnit and different JMS client libraries. Furthermore, different application servers require different Arquillian adapters for lifecycle management and deployment. Two of these cannot be simultaneously present on classpath. Both issues can be addressed by using a solution for dependency management of Java projects called Maven. In addition to handling libraries, it allows definition of profiles[40], each of which can define independent or incompatible dependencies and environment variables.

Test runs in Maven are performed by the Surefire plugin. Each type of application server has different profile with its own group qualifier stored in *arquillian.launch* environment variable. By default, they are run as forked process in a separate JVM, which creates a minor nuisance during the debugging process[41].

9.3 Initialization

The first step of running any JMS test is creating *InitialContext* defined in the *javax.naming* package to lookup *Destination* and *ConnectionFactory* objects. An instance of *InitialContext* requires several properties in order to successfully perform object lookup. Failover testing tool requires the following to be setup properly:

- PROVIDER_URL - JNDI provider URL with protocol, address and port.
- INITIAL_CONTEXT_FACTORY - factory used to generate initial context, represented by fully qualified class name.
- URL_PKG_PREFIXES - list of package prefixes to use when loading in URL context factories.

These key/value pairs differ for each application server, resulting in one settings combination per each of them. They can be loaded from *provider_name.properties* files and passed to constructor of *InitialContext*. Some providers and application servers require security policy to be enabled[42]. For these, a default user is created on the server. JMS providers should take into consideration common naming conventions for connection factories and destinations. By default, testing tool relies on naming conventions specified by Oracle's guide for resource references[43]. Overriding default values for tests is enabled by specifying *jms-destination.properties* on classpath. Example of this configuration file is bundled with application. This file includes also different names of *ConnectionFactory* objects for server-side and remote tests.

9.4 Clean and forced shutdown of nodes

Fail-over event can be caused by many different different reasons, most common being hardware or software failure. At the possible cost of over-simplification, Fail-over testing tool recognizes two types of fail-over events:

- Expected - clean shutdown. Tool performs administered shutdown of application server. This scenario often occurs when server maintenance is needed.

- Unexpected - simulated by crashing application server JVM by killing the OS process of cluster node, which is as close as possible to crashing of whole machine.

Besides providing tools to start servlet container automatically, Arquillian framework can manage the end of container's lifecycle as well. Clean shutdown can be done via Arquillian API using the *ContainerController* class. It is initialized by injection when Arquillian test starts and offers *stop()* method for clean shutdown. However, default Arquillian *kill()* method is implemented as internal call to above-mentioned *stop()*.

Some projects already experimented with killing of application server via Arquillian[44]. Their solution is specific to JBoss AS 7 and doesn't cover multiple instances running from the same directory with different profiles. Since development and use of testing tool is it both allowed and expected to run multiple instances of application server on single system. This calls for more sophisticated process killing mechanism.

Let IP_ADDRESS be the IP address on which is bound the targeted application server and PORT be the port value of its JNDI service. The IP:port combination is made available for tests anyway, for creation of *InitialContext*. It is possible to get the process identifier (PID) of process listening on given TCP port via command *lsof*. A simple Unix-based solution would then resemble following:

```
"kill -9 `lsof -i tcp:$PORT -s tcp:LISTEN \\
| grep $IP_ADDRESS | awk '{print $2}'`"
```

Invocation of this command from Java can be done via *Runtime.exec()*. The kill command is stored in *String* variable *killAS*:

```
Runtime.exec().exec(new String[] {"sh",
"-c", killAS});
```

9.5 Hudson jobs

Part of assignment is also the creation of Hudson jobs as well. Hudson is a CI tool written in Java able to run inside many modern servlet containers[45]. It offers integration with modern SCM tools such as Git and is able to run Ant and Maven based projects. Hudson is currently one of the most popular CI solutions along with Jenkins[46].

Hudson defines jobs that materialize a project from a Git repository. Once a copy of the project is present, Maven targets can be executed with additional configuration. Tests are run by executing the goal test. Vital part of this project is the selection of proper Maven profile to be used, since every provider is required to run with different libraries and *arquillian.launch* parameters.

10 Tested configurations

Selected providers were tested using a single machine. At start-up, they need to be bound to different IP addresses to prevent collisions. All application servers were configured as one main server used by the testing tool and one redundant server.

Modern application servers support multiple configurations using a single installation. Configuration can be a single XML file or a folder with configuration divided into multiple files. This tool does not rely on multiple profiles running from a single application server, so when two application servers are started, they run from two different installations.

10.1 JBM and JBoss AS 5

Tests for JBM are executed on JBoss AS 5.1.0.GA, which is distributed with version 1.4.3 of JBM. As persistence was used a MySQL database. Tested configuration of JBM relies on using the Java Management Extensions (JMX) technology. Main configuration file is *messaging-service.xml*, holding the main managed bean (MBean) called *ServerPeer*. Its identification is defined by attribute *ServerPeerID*. When configured into a cluster, its value needs to be unique for every node. This tool passes this argument to server at start-up as a command line argument, along with binding IP address

Additional configuration is split into several files. Connection factories are defined in *connection-factory-service.xml*, with one MBean for every *ConnectionFactory*. Fail-over support of generated connections is enabled by setting the variable *SupportsFailover* to true. The *destinations-service.xml* file contains one MBean for every administered destination. Clustering of a topics and queues is controlled by the boolean *Clustered* attribute.

10.2 HornetQ with JBoss AS 7

HornetQ was tested with JBoss Application Server version 7.1.1.Final. It comes pre-installed with HornetQ in version 2.2.13.Final. Tested

configuration runs one server as live on active node and one backup HornetQ node on a second server.

Configuration of application server is in the form of a single XML file divided into multiple modules. One of these is a HornetQ messaging module as an XML element. Inside are defined all the connection factories and JMS destinations. *ConnectionFactory* objects need to be configured with *<ha>true</ha>* to enable fail-over on created connections. One or more HornetQ modules may be present. HornetQ uses a common journal, which has the form of several directories for storing persistent data. Each persistence folder can point to a different destination to optimize performance:

- *bindings-directory* - stores queues, topics, connections and their JNDI bindings.
- *journal-directory* - directory for standard messages.
- *large-messages-directory* - messages larger than a configured value, default is 100KB.
- *paging-directory* - when messages sent to an address can no longer be stored in memory, they are paged into this location.

HornetQ in combination with JBoss AS 7 allows creation of collocated topology, where one or more backup nodes are running inside one application server. This application server can then contain backups for different live nodes. Since it less probable that multiple live nodes would fail at the same time, colocated topology can be an effective solution when a system uses more than one live node.

10.3 ActiveMQ and Apache Geronimo

Apache develops their own Java EE implementation named Geronimo. It is pre-bundled with one of two different servlet containers: Jetty and Tomcat. Arquillian can automatically start both. Tests use version 3.0 with Tomcat 7 servlet container and ActiveMQ as JMS provider. Both type of persistence options were tested. As RDBMS was used MySQL database system and journal configuration with kahaDB file-based database.

Configuration of Geronimo is based on so-called GBeans as opposed to MBeans used in JBM. Definition of *ConnectionFactory* is done by specifying a *"failover://"* protocol which points to primary broker. Since version 5.4, if *updateClusterClients* property is enabled, primary broker is notified when a new slave connects and clients are notified about new option of fail-over.

11 Conclusion

Developed testing tool runs uses a client to run individual tests for objects created during a typical JMS scenario. After validating their basic functionality, more complex tests are executed. These verify that when objects experience fail-over, they are still able execute more than basic JMS operations. The developed Failover testing tool can also check the behavior of clients when fail-over happens before a transaction is complete.

Since creation of *InitialContext* is not covered defined in the JMS specification, this tool does not test that an *InitialContext* object can be used to create JMS connection factories after fail-over. Commit and rollback in transacted sessions and sending of message acknowledgments in general are not atomic actions themselves, although this tool does view them so. It does not simulate situation where a commit is sent to from client to provider and executed there but fail-over happens before acknowledgment about the success is sent to the client. The case where transacted session communicates with multiple destinations at once is also not covered since its behavior resembles that of a session communicating with a single destination.

In case of multiple live servers configured into a cluster, most providers offer *ConnectionFactory* with HA configuration. Subsequent connections are created using client-side load-balancing with different policies, typically round-robin, random or custom. While this feature can be useful in real production, it is not part of the JMS specification itself. This tool uses only connections created with non-HA *ConnectionFactory* objects.

It is possible to make testing physically distributed on client side by running JMS clients in separate virtual machines and comparing sent and received messages using arbitrary network protocol. These JVMs could be even started on separate machines. For the sake of simplicity, remote JMS clients in this tool are connecting to providers from a single virtual machine.

Even though most JMS products offer capabilities beyond the scope of version 1.1 specification (e.g. streaming of messages, instantiation of connection factories without JNDI etc.), this tool only tests features defined in the JMS specification.

Future development of this tool could include new scenarios performing additional operations on tested JMS providers. This could include simulation of heavy traffic, fail-over induced through various network failures, more complex scenarios with multiple producers and consumers, clustered destinations etc. Architecture of this tool allows modelling of such cases by adding new nodes to scenarios. Eventually, this tool could load JMS scenario scripts from external configuration files.

Bibliography

[1] Oracle. Java EE at a Glance, May 2012. http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[2] Oracle. Hudson Continuous Integration, May 2012. http://hudson-ci.org/.

[3] Adrian Tkacz, Ewaryst; Kapczynski. *Internet - Technical Development and Applications*. Springer, 2011.

[4] evolven.com. Cost of downtime, outages and System failure in IT, September 2012. http://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html.

[5] Jim Gray. Why Do Computers Stop and What Can Be Done About It?, 2012. http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf.

[6] Martin L. Shooman. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis,and Design*. John Wiley, 2002.

[7] Atlassian. Clustering for Scalability vs Clustering for High Availability (HA), 2013. https://confluence.atlassian.com/pages/viewpage.action?pageId=154633658.

[8] Judith A. Clapp. *Software quality control, error analysis, and testing*. William Andrew, 1995.

[9] Edward Curry. *Middleware for Communications*. John Wiley, 2004.

[10] DedaSys LLC. Programming Language Popularity, September 2012. http://www.langpop.com/.

[11] The Java Community Process(SM) Program - JSRs: Java Specification Requests - JSR Overview. The Java Community Process Program, January 2013. http://jcp.org/en/jsr/overview.

[12] Sun Microsystems. Java Message Service, April 2002. http://download.oracle.com/otn-pub/jcp/7195-jms-1.1-fr-spec-oth-JSpec/jms-1_1-fr-spec.pdf.

[13] Java Community Process. JSR 343: Java Message Service 2.0, 2013 May. http://jcp.org/en/jsr/detail?id=343.

[14] Java Community Process. JSR 914: Java Message Service 1.1, 2003 December. http://www.jcp.org/en/jsr/detail?id=914.

[15] Oracle. Message (Java EE 6). http://docs.oracle.com/javaee/6/api/javax/jms/Message.html.

[16] ISO. Information technology - open Systems Interconnection - Basic Reference Model: The Basic Model. http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip.

[17] Oracle. The JNDI Tutorial, December 2012. http://docs.oracle.com/javase/jndi/tutorial/.

[18] Oracle. The JNDI Tutorial, 2013. http://docs.oracle.com/javase/tutorial/jdbc/.

[19] Evan L. Marcus. Which is better for HA, shared storage or disk data replication?, June 2003. http://searchstorage.techtarget.com/answer/Which-is-better-for-HA-shared-storage-or-disk-data-replication/.

[20] JBoss Community. JBoss Messaging 1.4.x Clustering Overview, 2008. https://community.jboss.org/wiki/JBMCluster.

[21] T. William Olle. The Codasyl Approach to Data Base Management, 1978.

[22] JBoss. JBoss Messaging - Configuring the Persistence Manager, 2011. http://docs.jboss.org/jbossmessaging/docs/userguide-1.4.3.GA/html_single/index.html#conf.persistencemanager.

[23] JBoss. HornetQ - High Availability and Failover, 2012. http://docs.jboss.org/hornetq/2.2.14.Final/user-manual/en/html/ha.html.

[24] Apache. Apache ActiveMQ – MasterSlave, 2012. http://activemq.apache.org/masterslave.html.

[25] Apache. Apache ActiveMQ – JDBC Support, 2012. http://activemq.apache.org/jdbc-support.html.

[26] Apache. Apache ActiveMQ – KahaDB, 2012. http://activemq.apache.org/kahadb.html.

[27] Oracle. Interface InitialContext, 2012. http://docs.oracle.com/javase/6/docs/api/javax/naming/InitialContext.html.

[28] Oracle. Session - transacted Session, 2012. http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#SESSION_TRANSACTED.

[29] Oracle. Session - getacknowledgemode, 2012. http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#getAcknowledgeMode().

[30] Java Community Process. JSR 907 - Java Transaction API (JTA), 2002. http://jcp.org/en/jsr/detail?id=907.

[31] Oracle. XAResource docs, 2012. http://docs.oracle.com/javaee/6/api/javax/transaction/xa/XAResource.html.

[32] Oracle. MessageFormatException docs, 2012. http://docs.oracle.com/javaee/6/api/javax/jms/MessageFormatException.html.

[33] Oracle. Xaresource, 2012. http://docs.oracle.com/javaee/6/api/javax/jms/Session.html#createProducer(javax.jms.Destination).

[34] Oracle. Creating durable subscriptions, 2012. http://docs.oracle.com/javaee/6/tutorial/doc/bncfu.html#bncgd.

[35] Oracle. Message-driven bean - tutorial, 2012. http://docs.oracle.com/javaee/6/tutorial/doc/gipko.html.

[36] Oracle. Mdb deployment options, 2012. http://docs.oracle.com/cd/E13222_01/wls/docs91/ejb/message_beans.html#1160756.

[37] JBoss. Large Messages, 2012. http://docs.jboss.org/hornetq/2.2.14.Final/user-manual/en/html/large-messages.html.

[38] JBoss. Arquillian - Write Real Tests, 2012. http://arquillian.org/.

[39] ShrinkWrap. WebArchive, 2012. http://docs.jboss.org/shrinkwrap/1.0.0-cr-3/org/jboss/shrinkwrap/api/spec/WebArchive.html.

[40] Apache. Maven - Introduction to build profiles, 2012. http://maven.apache.org/guides/introduction/introduction-to-profiles.html.

[41] Apache. Maven Surefire Plugin - Debugging Tests, 2012. http://maven.apache.org/surefire/maven-surefire-plugin/examples/debugging.html.

[42] JBoss. Wiki - AS 7.1.0 Beta1 - Security Enabled By Default, 2012. https://community.jboss.org/wiki/AS710Beta1-SecurityEnabledByDefault.

[43] Oracle. Guidelines, Patterns, and code for end-to-end Java applications, 2012. http://www.oracle.com/technetwork/java/namingconventions-139351.html.

[44] Martin Gencur. edg-tests. https://github.com/mgencur/edg-tests.

[45] Hudson-ci/Using Hudson/Containers, April 2013. http://wiki.eclipse.org/Hudson-ci/Containers.

[46] Jenkins. Who's driving this thing? Jenkins CI, 2013. http://jenkins-ci.org/content/whos-driving-thing.

[47] Martin Zruban. HornetQ Failover Testing Tool - GitHub repository. 2013. https://github.com/naviator/hornetq-failover-testing.

A Content of CD

The CD contains following directories:

- *thesis* - In this directory is the source code for this thesis in LaTeX and digital version as a PDF file.
- *testingtool* - This directory contains the source code of Failover testing tool. Project is also hosted on GitHub[47].
- *servers* - Configuration files for application servers to be executed with this testing tool.