Joram 5.13 User's Guide

Contents

Contents....2

Figures....6

1.Installation....8

1.1.Requirements....8

1.2.Getting Joram binary distribution....8

1.3.Running a Joram server....9

2.Using samples....10

2.1.Compiling JORAM samples....10

2.2.Running Joram samples....11

2.2.1.The classic sample....11

2.2.2.The chat sample....12

2.2.3.The distributed sample....13

2.2.4.The dotcom demo....14

2.2.5.The perfs samples....16

2.3.Using scripts....16

2.3.1.First step....16

2.3.2.Launching a JORAM platform....17

2.3.3.Launching a JORAM client....17

2.3.4.Running the classic samples using script files....17

2.4.Administration through XML scripts....18

2.4.1.Classic sample administration using XML script....18

3.Administration Guide....20

3.1.Introduction....20

3.2.Administration concepts....20

3.2.1.Overall view....20

3.2.2.User....21

3.2.3.Destinations....22

3.3.Platform configuration....23

3.3.1.Centralized configuration....24

3.3.2.Distributed configuration....25

3.3.3.Stopping a server....26

3.3.4.Dynamic configuration....26
3.3.5.Logging configuration....29
3.4.High level administration....29
3.4.1.Administration "session"....30
3.4.2.Managing a user....30
3.4.3.User connectivity....31
3.4.4.Managing a destination....33
3.4.5.Managing a Queue....35
3.4.6.Managing a Topic....36
3.4.7.Managing the platform....36
3.5.Message interceptors....37
3.5.1.Managing client interceptors....37
3.5.2.Managing server interceptors....38
3.6.JMX administration of Joram....41
3.7.Scripts XML....42
3.7.1.Administrator connection....42
3.7.2.ConnectionFactory....42
3.7.3.User....43
3.7.4.Destination....43
3.7.5.Example....44
3.8.OSGi Commands....45
3.8.1.A3 commands....45
3.8.2.MOM commands....45
3.8.3.JNDI commands....46
3.9.Dead Message Queue....46
3.9.1.Introduction....46
3.9.2.Managing a Dead Message Queue....49
3.9.3.Running the "Dead Message Queue" sample....50
3.10.Hierarchical Topic....51
3.10.1.Hierarchical topic....51
3.10.2.Managing a Hierarchical Topic....52
3.10.3.Running the topic tree sample....53
3.11.Clustered Topic....54
3.11.1.Introduction....54
3.11.2.Managing a clustered topic....55
3.11.3.Running the "Clustered Topic" Sample....57
3.11.4.Using XML Scripts....58
3.12.Clustered Queue....59
3.12.1.Introduction....59

3.12.2.Managing a clustered queue....60
3.12.3.Running the "Clustered Queue" Sample....62
3.12.4.Using XML Scripts....63
3.13.SchedulerQueue....65
3.13.1.Introduction....65
3.13.2.Using a schedulerQueue....65
3.14.Acquisition and distribution....66
3.14.1.Introduction....66
3.14.2.Configuring an acquisition destination....66
3.14.3.Configuring a distribution destination....67
3.14.4.Setting properties....68
3.14.5.Required libraries....68
3.14.6.Mail acquisition / distribution....68
3.14.7.URL acquisition (collector)....70
3.14.8.JMX acquisition (monitoring)....72
3.14.9.JMS acquisition / distribution bridge....73
3.14.10.AMQP acquisition / distribution bridge....79
3.14.11.AMQP acquisition / distribution proxy....82
3.14.12.Acquisition / distribution with PHP scripts....85
3.15.FTPQueue....86
3.15.1.Introduction....86
3.15.2.Managing a FTPQueue....87
3.15.3.Using a FTPQueue destination....88
3.15.4.Running the sample....88
4.Using a collocated server....90
4.1.Introduction....90
4.2.Configure a collocated server....90
4.3.Start a collocated server....90
4.4.Connect to the collocated server....91
4.4.1.Create local connections....91
4.4.2.Connect the administration module....91
4.5.Stop the collocated server....91
4.6.Start the embedding Java application....92
4.7.Getting Joram sources....92
4.7.1.Getting a packaged version of Joram....92
4.7.2.Getting Joram from SVN....92
4.7.3.Directory structure and description....92
4.8.Compiling and shipping Joram....93

4.8.1.Compiling Joram.....93
4.8.2.Generating the javadoc.....96
4.8.3.Generating a distribution.....96
4.8.4.Cleaning.....96

Figures

Figure 1 - Classic samples configuration.....11

Figure 2 - Chat sample configuration.....12

Figure 3 - Distributed sample configuration.....13

Figure 4 - Dotcom sample configuration.....14

Figure 5 - Web Server's interface.....15

Figure 6 - Inventory Server's and Control Server's interfaces.....15

Figure 7 - Customer Server's interfaces.....15

Figure 8 - Delivery Server's interface.....16

Figure 9 - Applications exchanging data through messaging.....20

Figure 10 - Joram platform and clients.....21

Figure 11 - A client connected to a server “through” a standard ConnectionFactory.....22

Figure 12 - A client accessing a server destination”through” a standard Destination.....23

Figure 13 - JConsole view.....41

Figure 14 - Messages on a queue sent to a DMQ.....47

Figure 15 - Dead message queue sample.....50

Figure 16 - A Hierarchical topic.....51

Figure 17 - A distributed Hierarchical topic.....52

Figure 18 - Topic tree sample.....54

Figure 19 - A clustered topic.....55

Figure 20 - Cluster sample configuration.....57

Figure 21 - A cluster of queues balancing heavy deliveries.....60

Figure 22 - The mail sample.....70

Figure 23 - A JORAM client communicating with a XMQ client.....74

Figure 24 - 2 Joram clients communicating through the JMS bridge 79

Figure 25: Distribution to an AMQP server using a distribution queue 79

Figure 26: Acquisition from an AMQP server using an acquisition queue.....80

Figure 27: Distribution to an AMQP client using a distribution queue 82

Figure 28: Distribution to an AMQP client using a distribution queue 82

Figure 29: File transfert through a FTP Queue.....87

Figure 30: File acquisition from a FTP Queue.....87

1. Installation

Joram 5 basically includes:

- A **messaging server** (or MOM), providing the messaging functionalities: basically hosting and routing the messages exchanged by the client applications. It includes either a JMS service and an AMQP service.
- A **JNDI** compliant naming server, distributed (since release 4.1) persistent and reliable.
- **Client classes** allowing applications to access the MOM functionalities. Those interfaces are defined by the **JMS 1.1** specifications.
- **Samples** illustrating the various features provided by Joram.
- JCA 1.5 connector allowing deployment in J2EE platform.

In option it provides:

- A **WebConsole** based on GWT and JMX,
- An eclipse plugin for architecture design,
- A JMS / JMX connector,
- An AMQP compliant provider, etc.

1.1. Requirements

Joram can run on a wide variety of platform, a typical hardware and software platform is:

Hardware requirements

- Year 2000 compliant 32-bit Intel based PC hardware (or equivalent)
- 256 Mb RAM, 5 Gb disk,
- Communication hardware supporting TCP/IP

Software requirements

- Operating system: Linux, Windows 2000 and XP, Windows 7, etc.
- Connectivity: TCP/IP.
- Java environment: JDK 1.5 and later.

1.2. Getting Joram binary distribution

The packages are downloadable from the following location:

- http://forge.ow2.org/project/showfiles.php?group_id=4.

For release **x.y.z**, the following tar file provided:

- **`joram-release-x.y.z.tgz`**, including the client and server libraries, as well as the javadoc and the samples sources.

This package is expanded by UNIX users with the ***`gunzip`*** and ***`tar`*** commands; Windows users can use the ***7-Zip*** utility.

The distribution is expanded in a `joram-x.y.z/` directory. It includes the following directories:

- `doc/`

- samples/
  - bin/…
  - config/…
  - src/joram/…
- ship/
  - lib/
  - licenses/

1.3. Running a Joram server

To run a server you have just to launch the `fr.dyade.aaa.agent.AgentServer` class with 2 parameters, the first one is the unique identifier of the server and the second one is the path of storage directory. For example:

```
java -cp …  fr.dyade.aaa.agent.AgentServer 0 ./s0
```

During its first initialisation the server will be configured according to the a3servers.xml configuration file (see below) then the configuration will be kept in the storage directory.

```
<?xml version="1.0"?>
<config>
  <server id="0" name="S0" hostname="localhost">
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args="root root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer"
             args="16400"/>
  </server>
</config>
```

A simple « a3servers.xml » configuration file

The «a3servers.xml» configuration file above simply defines a centralized Joram JMS server including a JNDI service, the JMS service is listening on TCP port 16010 and the JNDI service on TCP port 16400. The chapter 3.3 which describes more precisely the configuration of a Joram platform.

2. Using samples

This chapter describes the samples provided with JORAM and for each, the architecture of the underlying platform. The samples are provided with the JORAM distributions under the `samples/` directory. It's a good way to verify the correctness of Joram installation.

The **`samples/src/joram`** directory includes the samples codes of JORAM clients. Compiling and launching are done with the *`ant`* command.

Configuration files are located in the **`samples/config`** directory. They might be edited and adapted to your environment. For more information, please refer to the administration part of this document (chapter 3Administration Guide). This directory contains:

- `a3config.dtd`, the DTD for server configuration.
- `a3debug.cfg`, a default logger configuration file used by all samples.
- XML configuration files for centralized and distributed server architecture: `centralized_a3servers.xml`, `distributed_a3servers.xml`, etc.
- OSGi configuration file (Felix implementation) used by the various implementation:
  - config.properties : A simple configuration running Joram's JMS and JNDI services.
  - config_amqp.properties : A configuration running the Joram's AMQP service.
  - config_bridge.properties : The configuration used with the JMS bridge samples.
  - config_console.properties : A configuration with bundles needed to start the Web console.
  - config_stomp.properties : A configuration allowing the use of Stomp protocol.
- `jndi.properties`, a default configuration file for JNDI's clients.

The **`samples/bin`** directory provides Unix and Windows script files for launching JORAM servers and clients if you don't want to use ant targets.

All examples creates a **`samples/run`** where logging files and the persistence root (if any) of each server are created. Current configuration files are copied in this directory. When starting a platform with a new configuration, or when a clean platform is expected, this directory should be removed.

2.1. Compiling JORAM samples

The Joram samples need to be compiled. Under the `samples/src/joram` directory, simply type:

```
ant clean compile
```

This creates a `samples/classes/joram/` directory holding the compiled classes. For removing this directory, type:

```
ant clean
```

2.2. Running Joram samples

2.2.1. The classic sample

The JMS API provides a separate domain for each messaging approach, *point-to-point* or *publish/subscribe*:

- The *point-to-point* domain is built around the concept of queues, senders and receivers.
- The *publish/subscribe* domain is built around the concept of topic, publisher and subscriber
- Additionally JMS provides an unified domain with common interfaces that enable the use of queue and topic. This domain is defines the concept of producers and consumers.

This sample demonstrates the different messaging domains of JMS, *point-to-point* with a sender, a receiver and a queue browser, *publish/subscribe* with a subscriber and a publisher, and unified with messages producers and consumers.

The classic sample uses a very simple configuration (centralized) made of one server hosting a queue and a topic. The server is administratively configured for accepting connections requests from the *anonymous* user.

The platform is run in non persistent mode (The “Transaction” property is set to “fr.dyade.aaa.util.NullTransaction” in a3servers.xml configuration file).

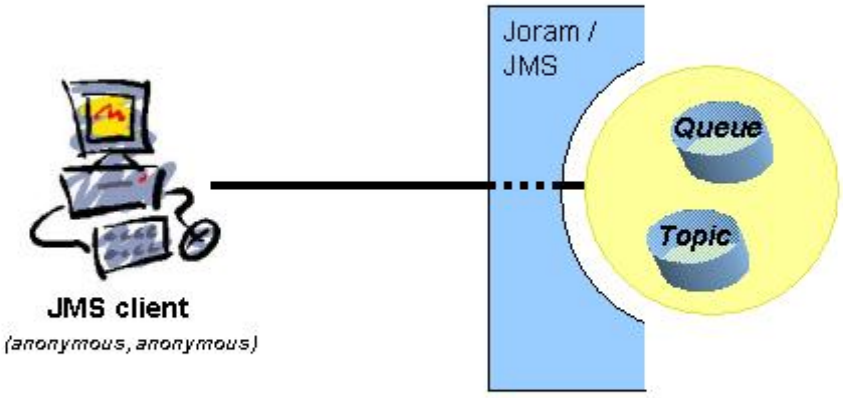

Figure 1 - Classic samples configuration

Running the demo with Ant:

- For starting the platform:

  ```
  ant reset single_server
  ```

  - As defined in the configuration file (run/a3servers.xml) it launches a Joram server without persistency. It creates a ConnectionManager, a TCP/IP entry point and a JndiServer (port 16400); the ConnectionManager defines a default administrator (username “root”, password “root”). The reset target is used to removes all out-of-date data in the run directory.
- For running the admin code:

  ```
  ant classic_admin
  ```

  - This client connects to the Joram’s server, then creates 2 JMS destinations (a queue and a topic) and an anonymous user. It defines 3 different ConnectionFactory, one for each messaging domain.
  - Each administered objects is then bound in JNDI.
  - The corresponding code is in the ClassicAdmin.java file of the classic directory.

- Optionally you can use the `classic_adminxml` Ant target, it does the same job using XML scripts rather than the programmatic API (the corresponding script is in the joramAdmin.xml file of the classic directory).

- Normally JMS is used through the unified messaging domain:
  - It uses the common *`ConnectionFactory`* "cf", and the *`Destination`* "queue" or "topic" retrieved from JNDI.
  - For running the consumer sample, type "ant consumer_queue" or "ant consumer_topic". It continuously reads messages sent to the queue or the topic.
  - For running the producer sample, type "ant producer_queue" or "ant producer_topic". It sends 10 messages to the queue, and 10 messages to the topic.
- Using the *point-to-point* messaging domain:
  - It uses the *`QueueConnectionFactory`* "qcf", and the *`Queue`* "queue" retrieved from JNDI.
  - For running the sender sample, type "*`ant sender`*"; each time, it sends 10 messages to the defined queue.
  - For running the browser sample, type "*`ant browser`*"; it allows to look messages on queue without removing them.
  - For running the receiver sample, type "*`ant receiver`*"; each time, it consumes 10 messages from the queue. If there is not enough messages, it stops until new messages are produced.
- Using the *publish/subscribe* messaging domain:
  - It uses the *`TopicConnectionFactory`* "tcf", and the *`Topic`* "topic" retrieved from JNDI.
  - For running the subscriber sample, type "*`ant subscriber`*". It subscribes to the defined topic, and then receives all messages later published on this topic.
  - For running the publisher sample, type "*`ant publisher`*". It publishes 10 messages on the topic.

2.2.2. The chat sample

The chat sample uses a very simple configuration (centralized) made of one server hosting a single queue. The server is administratively configured for accepting connections requests from the *anonymous* user.

The platform is run in non persistent mode (property "Transaction" is set to "fr.dyade.aaa.util.NullTransaction" in a3servers.xml configuration file).

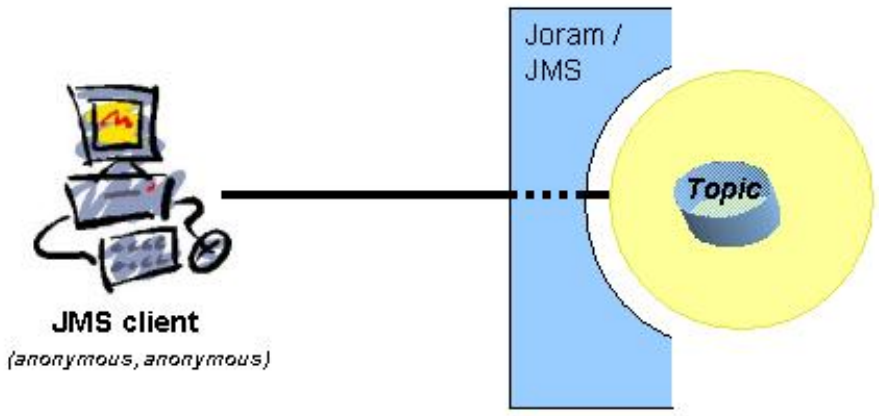

Figure 2 - Chat sample configuration

Running the demo:

- For starting the platform:
  `ant reset single_server`
- For running the admin code:

```
ant chat_admin
```

  - This client connects to the Joram's server, and then creates a topic and an anonymous user. It defines a TopicConnectionFactory. Each administered objects is then bind in JNDI.
- To start a chat client, type "`ant chat1`". It launches a chat client with user1 speudo, then each message typed at console is sent to the topic, and each message published on the topic is written to the console.
- To start a second chat client, type "`ant chat2`". It simply launches a chat client with user2 speudo.

2.2.3. The distributed sample

The distributed sample illustrates Joram under a distributed architecture. Its configuration involves three servers. The clients producing messages (sender and publisher) connect to server 0. The clients consuming the messages (receiver and subscriber) connect to server 2. The destinations they interact with are deployed on server 1. The platform is run in persistent mode. The provided configuration locates all three servers on "localhost" host.

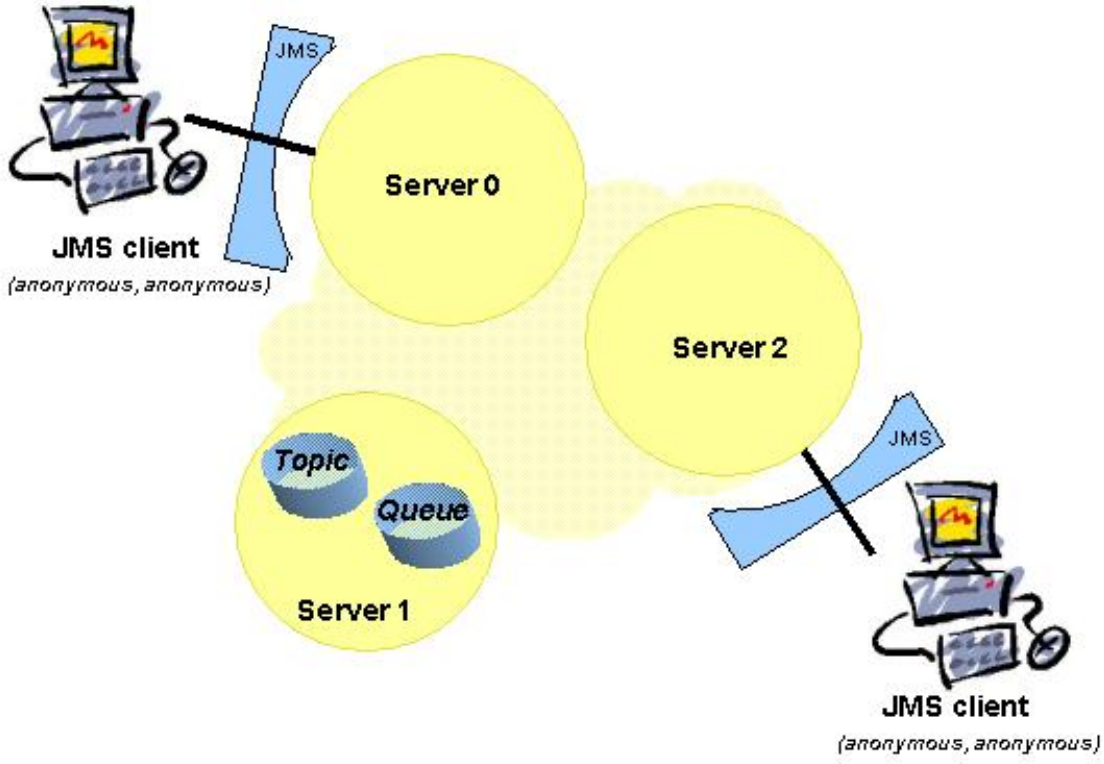

Figure 3 - Distributed sample configuration

Running the demo:

- Starting the configuration, type "`ant reset servers`". It cleans the run directory the launches the 3 servers. You can start separately each servers by typing:
  ```
  ant reset
  ant server0
  ant server1
  ant server2
  ```
- Running the admin code:
  ```
  ant archi_admin
  ```
- Running the producers:
  ```
  ant archi_sender
  ant archi_pub
  ```
- Running the consumers:

```
ant archi_receiver
ant archi_sub
```

2.2.4. The dotcom demo

The dotcom demo simulates what could be a commercial transaction involving many participants:

- Web server: server on which a customer order items.
- Customer server: centralizes the processing of the orders.
- Inventory server: checks if the items ordered are available.
- Billing server: centralizes the processing of the bank references.
- Control server: checks the bank references of the customers.
- Delivery server: receives the order ready for delivery.

The next picture shows the actors of this simulation and the destinations through which they interact. The provided architecture is centralized. The platform runs in persistent mode.

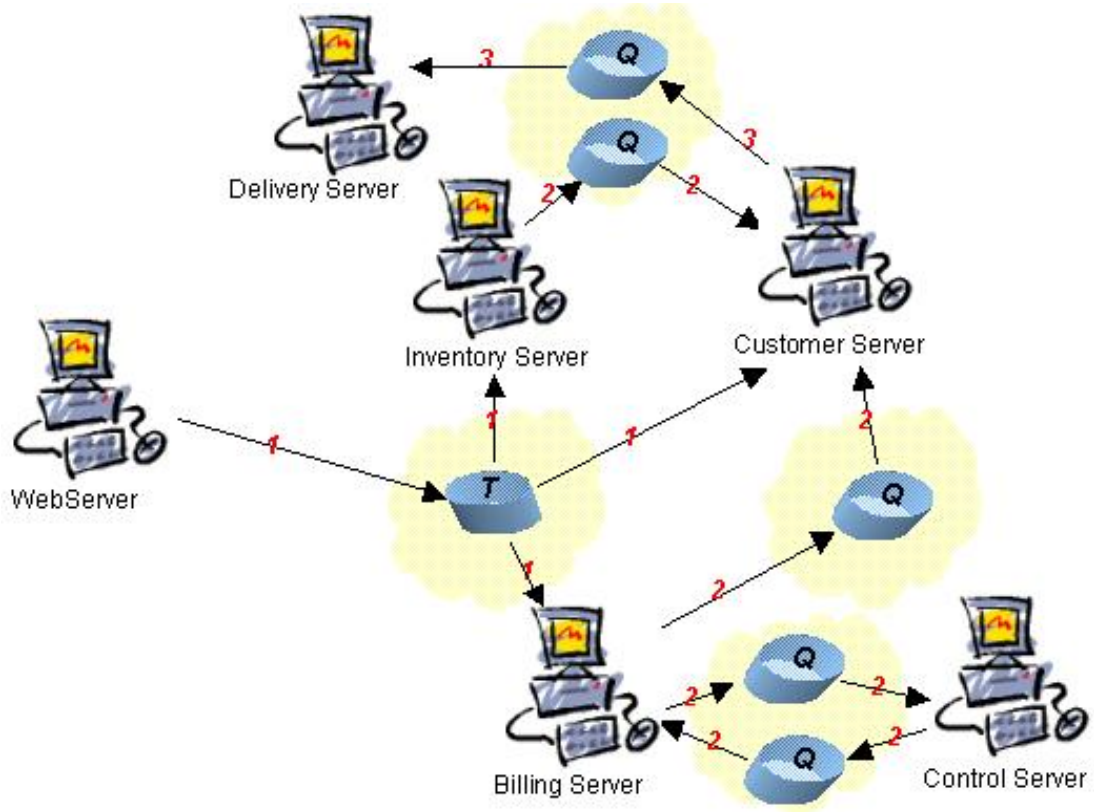

Figure 4 - Dotcom sample configuration

Scenario:

1. A customer buys an item on a web site. The Web Server publishes the order on a topic to which the Customer Server, the Inventory Server and the Billing Server have subscribed.
2. The Inventory Server checks if the item ordered is available and sends his answer back to the Customer Server. The Billing Server forwards the order to the Control Server who will check the bank references of the customer and send his answer back to the Billing Server. Then, the Billing Server forwards that answer to the Customer Server.
3. If the order has been validated by both Inventory Server and Billing Server, the Customer Server forwards it to the Delivery Server for delivery.

Running the demo:

- Starting the configuration:

```
ant reset single_server
```

- Running the admin:
  ```
  ant dotcom_admin
  ```
- Running the servers:
  ```
  ant webServers
  ```
- Running the client:
  ```
  ant webClient
  ```

The dotcom sample's GUI:

A GUI allows to interact with the demo. Each time a server receives a message, its window appears.

The WebServer's interface simulates the choice the user has to make between items: shoes, socks, trousers, shirt and hat. He must select one and then Send the order or set an Other order. The Send button must be pressed after the last order. It commits all previous orders. For cancellation, the Cancel button rollsback the orders. The Quit button sends the quit command to the other participants, closes the connections of the Web Server and terminates the program. Quit doesn't kill the middleware, thus it is possible to simply restart the application without having to relaunch the Agent server and the Admin.

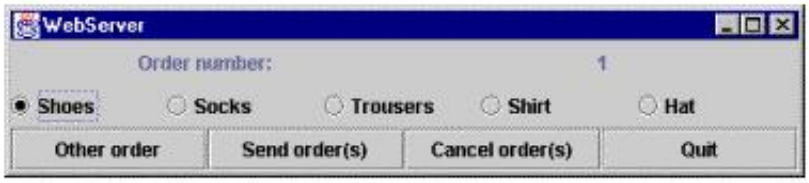

Figure 5 - Web Server's interface

The Inventory and Control Servers windows allow to simulate the work of those servers by validating or not the order they received.

Figure 6 - Inventory Server's and Control Server's interfaces

According to the results of previous controls, the Customer Server either will be able to ask for delivery, or won't.

Figure 7 - Customer Server's interfaces

Finally, if sent by the Customer Server, the order reaches the Delivery Server.

Figure 8 - Delivery Server's interface

2.2.5. The perfs samples

The perfs samples have been developed for checking Joram's performances. What is actually measured is the messages mean travel time (travel from the producer to the consumer). The configuration used is centralized, made of one queue and one topic. For testing PTP and Pub/Sub modes, the available clients are a Sender, a Publisher, a Receiver and a Subscriber.

These clients, as provided, are non transactional, subscriber is non durable. Of course these parameters may be changed for testing various configurations. Tests might be run on a persistent platform or a non persistent one.

The receiver and subscriber samples produce a PerfsFile file containing the mean messages travel time (computed for groups of 10 messages in the PTP case, 50 messages in the Pub/Sub case).

Starting the platform:

- Persistent platform:
  `ant reset server0`
- Or non persistent platform:
  `ant reset single_server`
- Running the admin code:
  `ant perfs_admin`

Testing the PTP mode:

- Running the receiver:
  `ant perfs_receiver`
- Running the sender:
  `ant perfs_sender`

Testing the Pub/Sub mode:

- Running the subscriber:
  `ant perfs_sub`
- Running the publisher:
  `ant perfs_pub`

2.3. Using scripts

In the previous sections, it has been explained how to launch the provided samples through *Ant* targets. It is also possible to use the script files located in the `samples/bin` directory. This section explains how to use those scripts.

2.3.1. First step

The first step consists in fixing the JAVA_HOME and the JORAM_HOME environment variables. The JAVA_HOME property value must point to your Java installation directory. The JORAM_HOME value must point to your JORAM directory (the directory actually containing the `samples/` sub-directories).

2.3.2. Launching a JORAM platform

Launching a JORAM platform with the scripts has the same effects as using the *Ant* targets.

Depending on the script, it will set the appropriate configuration: copy the rigth `a3servers.xml` and `jndi.properties` from `config/` directory in the created `run/` directory, etc.

Those scripts are:

- *`single_server.[sh/bat]`* : copies the `config/centralized_a3servers.xml` file as `a3servers.xml` and `config/jndi.properties` in `run` directory. If not already done, creates the `run/` directory. Then launches the non persistent server 0.
- *`server.[sh/bat] x`* : copies `distributed_a3servers.xml` as `a3servers.xml` and `jndi.properties` as `jndi.properties` if not already done, creates the `run/` directory if it does not exist, and launches the persistent server x.
- *`clean.[sh/bat]`* : deletes the `a3servers.xml` and `jndi.properties` files, deletes the `run/` directory.

When starting a new persistent server, the *clean* script must be executed in order to remove any existing persistence root which may alter the way the server starts. When re-starting a stopped or crashed persistent server, the *clean* script should not be called in order to keep the needed persistence root.

2.3.3. Launching a JORAM client

The *jmsclient* script may be used for launching a client. It takes as argument the class of the client to execute. For example, for launching the classic sender class:

```
jmsClient classic.ClassicSender
```

Of course, this supposes that the samples have been compiled (and that the JORAM platform has been administered for the classic samples, either by running the `ClassicAdmin` client, or by using the administration graphical tool).

2.3.4. Running the classic samples using script files

The example below use '.sh' scripts on a Linux platform; if you use a Windows™ platform you may use the corresponding '.bat' scripts. All theses scripts need the definition of `JAVA_HOME` and `JORAM_HOME` environment variable:

- Set `JAVA_HOME` to the directory where JDK is installed.
- Set `JORAM_HOME` to the directory that you installed Joram (the directory containing the `ship` and `samples` directories.

First cleans the persistence directory and configuration settings, then launches the server.

```
$> cd $JORAM_HOME/bin
$> ./clean.sh
== Cleaning the persistence directories and configuration settings ==

$> ./single_server.sh
== Launching a non persistent server#0 ==
AgentServer#0 started: OK
```

You can create all needed administered objects through the `ClassicAdmin` class.

```
$> ./jmsclient.sh classic.ClassicAdmin
== Launching the classic.ClassicAdmin client ==
```

```
Classic administration...
Admin closed.
```

Then you can send or receive messages using the Sender/Receiver or Publisher/Subscriber classes; for example:

```
$> ./jmsclient.sh classic.Sender
== Launching the classic.Sender client ==

Sends messages on the queue...
10 messages sent.

$> ./jmsclient.sh classic.Receiver
== Launching the classic.Receiver client ==

Requests to receive messages...
Msg received: Test number 0
Msg received: Test number 1
Msg received: Test number 2
Msg received: Test number 3
Msg received: Test number 4
Msg received: Test number 5
Msg received: Test number 6
Msg received: Test number 7
Msg received: Test number 8
Msg received: Test number 9

10 messages received.
```

You can launch the administration GUI JAMT using the admin.sh (respectively admin.bat) script:

```
$> ./admin.sh
== Launching the graphical administration tool ==
```

2.4. Administration through XML scripts

There is three way to deploy Joram's administered objects: the administration API, the graphical administration tool (JAMT) and now the XML scripting capability.

This feature use the AdminModule to execute the corresponding XML script. The script allows describing the administration connection, creating and binding administered objects (see chapter 3.7 Scripts XML).

2.4.1. Classic sample administration using XML script

The ant target `classic_adminxml` uses the AdminModule main static method to execute the administration script, this script is equivalent to the `ClassicAdmin` code.

```
$> ant classic_adminxml
Buildfile: build.xml

init:
```

```
classic_adminxml:
     [copy] Copying 1 file to C:\cygwin\home\freyssin\owjoram\joram\samples\run

BUILD SUCCESSFUL
Total time: 3 seconds
```

In the script (see file samples/src/joram/classic/joramAdmin.xml) we described:

- The connection to Joram's configuration: a default TCP connection with hostname, port, username and password.
- The connection factory and the JNDI binding:
  - an unified TCPConnectionFactory named “cf”,
  - a QueueTCPConnectionFactory named “qcf”
  - and TopicTCPConnectionFactory named “tcf”.
- The anonymous user.
- The destinations with their JNDI binding and security settings: a queue and a topic with freereader and freewriter settings.

3. Administration Guide

3.1. Introduction

JORAM provides a messaging platform allowing distributed applications to exchange data through message communication (Figure 9).

Figure 9 - Applications exchanging data through messaging

The messaging system takes care of distributing the data produced by an application to another application. Applications do not need to know each other, or to be present at the same time.

In order to provide a standardized way to access its messaging functionalities, JORAM implements the set of classes and methods defined by the JMS API. JMS “client” applications may then, without any modification, use JORAM messaging platform.

This document presents how to configure and start the underlying messaging platform, and how to administer it so that it is usable by standard JMS clients.

3.2. Administration concepts

3.2.1. Overall view

A Joram messaging platform is constituted by one or many servers, interconnected, possibly running on remote nodes (Figure 10).

- A **Joram server** is a Java process providing the messaging functionalities, and hosting messaging destinations.
- A **Joram JMS client** is a Java process using the messaging functionalities through the JMS interfaces. In order to do so it connects to a Joram server.

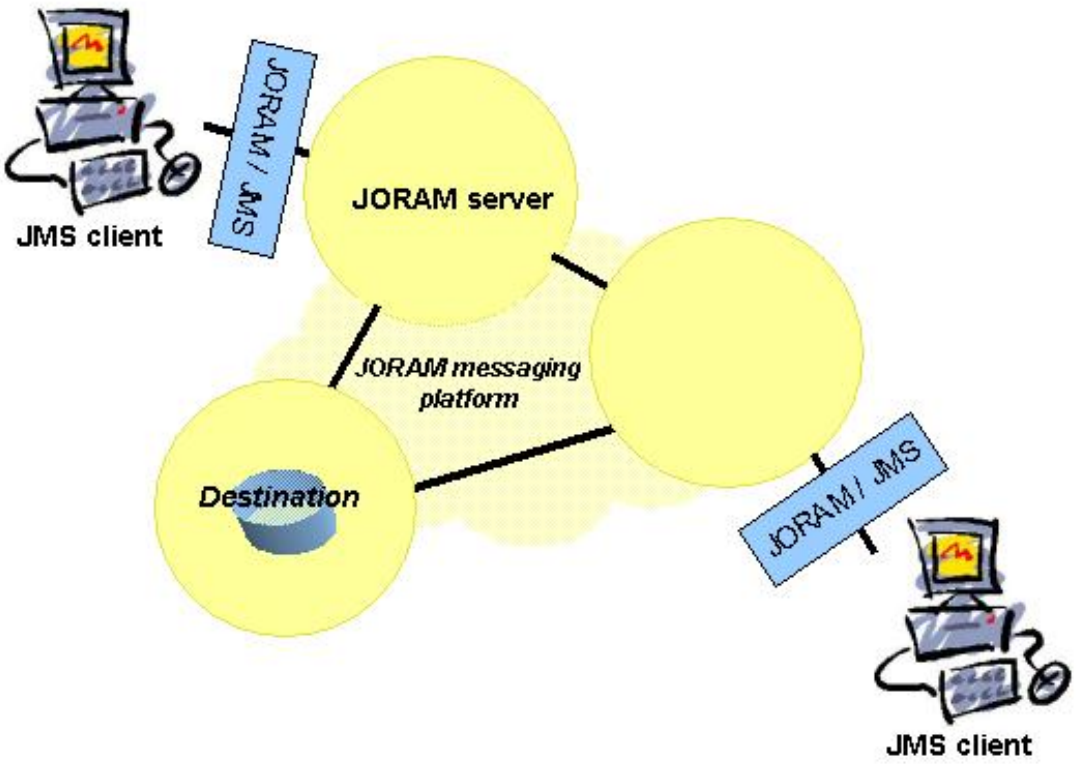

Figure 10 - Joram platform and clients

The goal of administration is to start and configure the messaging platform so that it provides all the features needed by the "client" applications. It is also to administer this platform so that standard JMS clients can access it and use it for their messaging operations.

The basic administration tasks are creating and deleting physical destinations on the messaging platform, setting or removing user's access to this platform.

To have the platform usable by standard JMS clients, the administration phase also consists in creating the `javax.jms.ConnectionFactory` and `javax.jms.Destination` administered objects (see JMS specification, §4.2), and to bind those instances to a JNDI compliant naming server.

3.2.2. User

A user access to a JORAM platform is fully described by:

- server parameters (such as host name and port number), identifying to which server of the platform the user will connect;
- a protocol, used for the client – server communication (usually TCP, might be "local", for collocated client and server);
- a user identification (name and password).

The actual "physical" connection is wrapped by a `javax.jms.Connection` instance. A JMS `Connection` is created by calling the `createConnection` method on a `javax.jms.ConnectionFactory` instance. It is this `ConnectionFactory` instance which wraps the server and communication protocol parameters. This standard object allows to isolate clients from the proprietary parameters needed for opening a connection with a messaging platform (Figure 11).

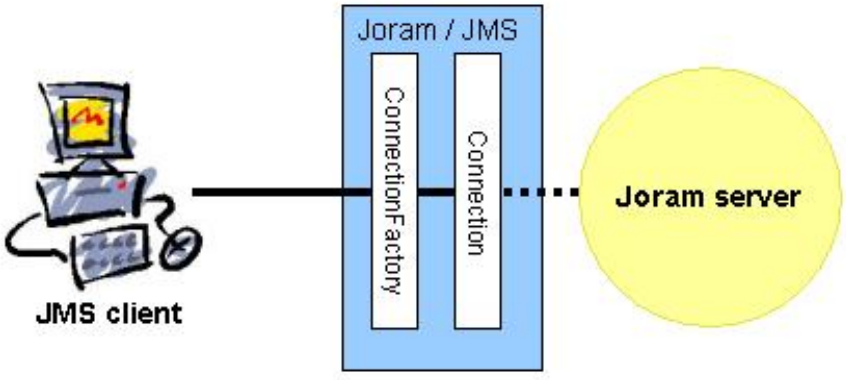

Figure 11 - A client connected to a server "through" a standard ConnectionFactory

A connection is opened by calling the `ConnectionFactory.createConnection` method. You can either use the method specifying an explicit user identity (login name and password) or assume the default identity (login "anonymous", password "anonymous"). The default identity may be adjusted client side by setting the JoramDfltLogin and JoramDfltPassword properties.

If the user identification (either *anonymous – anonymous*, or *name – password*) is unknown server side, the `createConnection` methods won't succeed and will throw a `JMSSecurityException`.

Allowing a client access to the platform requires then:

1. to create the appropriate `ConnectionFactory` instance wrapping the parameters of a server of the plaform, and of the communication protocol;
2. to bind this instance in a name space such as JNDI server so that users may later retrieve it;
3. to set the client as a user on this server.

3.2.3. Destinations

Client applications exchange messages not directly but through destinations. A destination is, server side, an instance of an object receiving messages from producers and answering to consuming requests from consumers. As shown on Figure 12, a destination may be deployed on any server of a configuration, whatever the servers the clients are connected to.

Server-side physical destinations are "represented" client side by `javax.jms.Destination` instances. A `Destination` instance wraps the parameters of the corresponding physical destination, and allows clients to be isolated from the proprietary parameters of a physical server side destination (Figure 12).

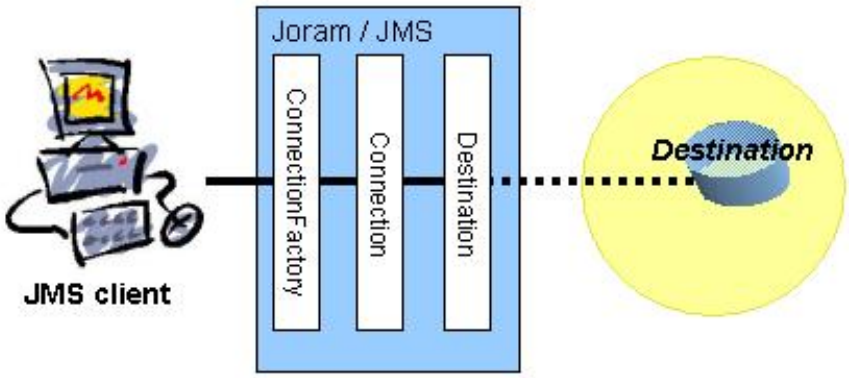

Figure 12 - A client accessing a server destination"through" a standard Destination

A destination might either be a "queue" or a "topic". Messaging semantics is what makes the difference (check any documentation about message-oriented-middleware or the JMS spec §5 and §6):

- Queue: each messages is read only by a single client.
- Topic: All clients that have previously subscribed to this topic are notified of the corresponding message.

Beyond this main characteristic, each destination may have a particular semantic; Joram supply many specific destinations : hierarchical or clustered destinations, bridge, etc.

The creation of a destination is then a three steps process:

1. first, creating the physical destination on a given server of the platform,
2. second, creating the corresponding `javax.jms.Destination` instance wrapping the parameters of the server side destination,
3. third, binding the `Destination` instance in a name space such as a JNDI server, so that clients may then retrieve it.

Once retrieved, a destination allows clients to perform operations according to their access rights. A client set as a READER will be able to request messages from the destination (either as a subscriber to a topic, or as a receiver or browser on a queue). A client set as a WRITER will be able to send messages to the destination.

Dead Message Queue (DMQ)

The Dead Message Queue (DMQ) is a particular queue used to store the dead messages. A dead message is a message that can not be delivered for various reasons (see chapter 3.9). The DMQ can be configured at different levels: server, destination, etc.

3.3. Platform configuration

Configuring a JORAM messaging platform consists in defining the number of servers that will constitute it, where they will run, and in defining services each will provide. The minimal configuration is a single server configuration. A platform configuration is described by an XML configuration file.

A dynamic configuration feature is available since Joram 4.2 , it allows to modify a Joram platform at run-time by adding and removing servers.

Server services

The services a server may host are:

- A **connection manager service**, managing the connection requests from "external" clients. This service may also authorize the connection of an administrator client, authenticated by a name and a password. It is required on any server accepting at least a client connection. At the platform level at least one server must accept an administrator connection, meaning that at least one server must host a connection manager service authorizing an administrator connection.
- A **TCP proxy service**, allowing TCP clients to connect to the server. This service takes as argument a port number, defining on which port the TCP connection requests should be made.
- A **JNDI service**, listening to a given port, providing a naming server to clients for binding and retrieving administered objects. It is required on one of the platform servers if clients and administrators intend to use JORAM's naming server. If this service is provided by none of the platform's servers, that means that clients and administrators do not intend to use JNDI, or that they will use an other JNDI implementation than the one provided by JORAM.

3.3.1. Centralized configuration

The example below sets a configuration made of one server running on host *localhost*. This server, identified by the number *0*, is named *s0*. It provides a connection manager service allowing an administrator identified by *root – root* to connect, and a TCP proxy service listening on port *16010*. A JNDI service is also provided, listening to JNDI requests on port *16400*.

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NullTransaction"/>
  <server id="0" name="S0" hostname="localhost">
    <service
     class="org.objectweb.joram.mom.proxies.ConnectionManager"
     args="root root"/>
    <service
     class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
     args="16010"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer"
     args="16400"/>
  </server>
  </config>
```

The above platform is non persistent, meaning that if it crashes and is then re-started, pre-crash data is lost. To have a platform able to retrieve its pre-crash state when re-starting, it should run in persistent mode. If message persistence is required, this is the mode to use (see below).

In order to allow a standard JNDI access to administrators and clients, a `jndi.properties` file is provided. It must be accessible to the administrators and clients through their classpath.

For the above configuration, this file looks as follows:

```
java.naming.factory.initial fr.dyade.aaa.jndi2.client.NamingContextFactory
java.naming.factory.host localhost
java.naming.factory.port 16400
```

It allows retrieving the naming context through:

```
javax.naming.Context jndiCtx = new javax.naming.InitialContext();
```

Running a platform

The configuration file is named `a3servers.xml`, and it must be accessible through the classpath. Then, the server is launched by typing:

```
java fr.dyade.aaa.agent.AgentServer 0 ./s0
```

Configuring a persistent server

In order to configure a persistent server you have to change the `Transaction` property in `a3servers.xml` configuration file. For example you may use fr.dyade.aaa.util.NTransaction class.

When such a persistent server is stopped or crashes, there are two options when re-starting it:

- Either it is expected to resume the operations it was involved in before the crash, in which case the persistence directory `s0` should not be deleted; it may happen that a `Lock` file in this directory remains and should be removed.
- Or it is a bright new server that is expected to start, in which case the persistence directory `s0` should be totally removed.

3.3.2. Distributed configuration

A distributed configuration made of three persistent server (as on figure 2) looks as follows:

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NTransaction"/>
  <domain name="D1"/>
  <server id="0" name="S0" hostname="localhost">
    <network domain="D1" port="16301"/>
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args="root root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer"
             args="16400"/>
  </server>
  <server id="1" name="S1" hostname="host1">
  <network domain="D1" port="16301"/>
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
  </server>
  <server id="2" name="S2" hostname="host2">
    <network domain="D1" port="16301"/>
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"/>
  </server>
</config>
```

This configuration is made of 3 persistent servers, each running on a given node (*host0*, *host1* and *host2*). All are part of the same domain (multiple domains might be needed for very large configurations). The server 0 of the configuration provides the same services as server 0 of the previous centralized configuration. Server 1 allows TCP connection on its local 16010 port, no administrator access, and no JNDI server. Server 2 allows client connections (thanks to the connection manager service) but the TCP protocol is not supported (the protocol might then be "local").

The `jndi.properties` file needed by administrators and clients should look as follows:

```
java.naming.factory.initial fr.dyade.aaa.jndi2.client.NamingContextFactory
java.naming.factory.host host0
```

```
java.naming.factory.port 16400
```

Running a platform

Each host on which a server of the configuration will run must have a copy of the `a3servers.xml` file, and this copy must be accessible through the classpath.

Then, the servers of the configuration are launched one by one:

- On node 0:
  java fr.dyade.aaa.agent.AgentServer 0 ./s0
- On node 1:
  java fr.dyade.aaa.agent.AgentServer 1 ./s1
- On node 2:
  java fr.dyade.aaa.agent.AgentServer 2 ./s2

Warning: Be careful, removing the persistence directory of one server in a distributed configuration may cause damages.

3.3.3. Stopping a server

A method is provided for stopping a given server of the administered JORAM platform. If the server to stop is the server to which the administrator is connected, the admin session is automatically terminated and closed.

Stopping server 0:

```
AdminModule.stopServer(0);
```

3.3.4. Dynamic configuration

The dynamic configuration feature is available from the Joram version 4.2. It allows to modify a Joram platform at run-time by adding and removing servers. As the servers can be gathered into several domains you can also add and remove domains.

Adding a new server

You can dynamically configure your Joram platform by adding new Joram servers. This is a two steps operation:

1. define the new server in the platform configuration using the Joram administration API
2. start the new server

Let's take an example in order to illustrate how it works. This simple scenario starts from a very simple Joram platform configuration that contains only one server called `S0`. This configuration is defined in Joram user guide (chapter 3.3.1).

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NullTransaction"/>
  <server id="0" name="S0" hostname="localhost">
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager" args="root
root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
args="16010"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer" args="16400"/>
  </server>
</config>
```

Server definition

The definition of a new server is programmatically done using the class `AdminModule` from Joram's administration API (package `org.objectweb.joram.jms.admin`).

```
import org.objectweb.joram.client.jms.admin.AdminModule;
```

First you need to connect the `AdminModule` to the Joram server `S0`:

```
AdminModule.connect("localhost", 16010, "root", "root", 60);
```

In order to define a new server you must specify in which domain the server is added. As the initial configuration doesn't define any domain, you have to add a first domain to the platform configuration.

A domain is defined by three parameters:

1. its name (unique inside a platform)
2. the name of an existing server that will be the first server belonging to this domain. When this server already belongs to a domain, it becomes the router between this domain and the new domain.
3. the port used by the first server to communicate with the other servers from this domain (none at the beginning)

The following code adds the domain `D0` that contains the server `S0`. The port used by `S0` to communicate inside `D0` is `17770`.

```
AdminModule.addDomain("D0", "S0", 17770);
```

Once the domain `D0` is added you can add a new server `S1` into this domain. A server is defined by five parameters:

1. the identifier of the server (unique inside a platform)
2. the address or name of the host where the server is running
3. the name of the domain where the server is added
4. the port used by the server to communicate with other servers inside the domain
5. its name (may be not unique)

```
AdminModule.addServer("localhost", 1, "D0", 17771, "S1");
```

Now the server `S1` has been added you need to get the overall configuration of the platform in order to start `S1`.

```
String platformConfig = AdminModule.getConfiguration();
```

The configuration is returned as a `String` which content is:

```
<?xml version="1.0"?>
<!DOCTYPE config SYSTEM "a3config.dtd">

<config>

  <domain name="D0" network="fr.dyade.aaa.agent.SimpleNetwork"/>

  <server hostname="localhost" id="1" name="S1">
    <network domain="D0" port="17771"/>
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager" args="root
root"/>
  </server>
```

```
  <server hostname="localhost" id="0" name="S0">
    <network domain="D0" port="17770"/>
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager" args="root
root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
args="16010"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer" args="16400"/>
  </server>

</config>
```

As you can see, the initial platform configuration has been extended with the definition of a new domain `D0` and a new server `S1`.

Store this configuration into a file `a3servers_updated.xml`. This file is necessary to start the new server `S1`.

```
File platformConfigFile = new File("a3servers_updated.xml");
FileOutputStream fos = new FileOutputStream(platformConfigFile);
PrintWriter pw = new PrintWriter(fos);
pw.println(platformConfig);
pw.flush();
pw.close();
fos.close();
```

Server start

The server `S1` is started in the same way as described in Joram user guide (see 3.3.2, running a platform):

1. copy the file a3servers_updated.xml in the directory where you want to start `S1` and rename it to `a3servers.xml`. You also need to put the DTD file `a3config.dtd` in the same directory.
1. customize the configuration of `S1` by modifying the file `a3servers.xml`. For example, you can add services (e.g. distributed `JndiServer`).
2. start the server with the following commands:

```
cd <S1_Running_Dir>
java fr.dyade.aaa.AgentServer 1 ./s1
```

Removing a server

This is a two steps operation:

1. stop the server
2. remove the server from the platform configuration using the Joram administration API

Notice that you can also remove it first from the configuration and then stop it.

Server stop

To stop a server you need to specify the identifier of the server. Notice that this operation is not synchronous, i.e. the server is asynchronously stopped. The server may still be running a while after the method `stopServer` returned.

```
AdminModule.stopServer(1);
```

Server removal

To remove a server from the platform configuration, you need to give the identifier of the server. This operation destroys all the pending messages sent to the removed server through the whole platform.

```
AdminModule.removeServer(1);
```

You can also remove a domain even if it is not empty. In this last case, the servers inside this domain are also removed. So you have to stop them.

```
AdminModule.removeDomain("D0");
```

This last operation removes the domain `D0` but not the server `S0` because it is used to make the dynamic configuration.

When you manipulate configurations with multiple domains by removing servers and/or domains, be careful not to split your platform into several parts.

3.3.5. Logging configuration

JORAM uses **Monolog** (see http://monolog.ow2.org/) for logging. Monolog is an API which abstracts log operations from their implementation.

Logging is configured in an `a3debug.cfg` file. It has to be in the classpath of the client and of the server (the server's process as well as the client's might be logged).

The `a3debug.cfg` configuration file defines the *appenders* used to log. By defaults, it logs on the standard output but a file is usable instead.

This file also defines all the categories which are available for logging. These categories are:

- Agent logs (categories starting with fr.dyade.aaa.agent): these categories log what happens in the low level messaging platform.
- MOM logs (categories starting with org.objectweb.joram.mom): these categories log what happens in a JORAM server, more particularly:
- in the server's proxies (org.objectweb.joram.mom.Proxy),
- in the server's destinations (org.objectweb.joram.mom.Destination).
- JORAM logs (org.objectweb.joram.client.jms.Client category): this category logs JMS client operations.
- JNDI logs (fr.dyade.aaa.jndi2): this category logs all JNDI operations, more particularly:
- in JNDI's server side (fr.dyade.aaa.jndi2.server),
- in JNDI's client side (fr.dyade.aaa.jndi2.client).

3.4. High level administration

When the messaging platform has been configured and started, the situation looks as follows:

- one or many interconnected servers run;
- each server may provide services for connecting and administering.

At that point an administrator client needs to connect to the platform and further configure it for allowing JMS clients to access and use it.

This administrator works either through a Java application using proprietary JORAM administration methods (described in this section), or through a web admin interface documented separately.

When the administration process is performed by a Java application, it uses JORAM's proprietary administration methods and objects. Those objects are:

- org.objectweb.joram.client.jms.admin.AdminModule
- org.objectweb.joram.client.jms.admin.AdminHelper
- org.objectweb.joram.client.jms.admin.User
- org.objectweb.joram.client.jms.Queue
- org.objectweb.joram.client.jms.Topic

And the various connection factory objects located in:

- org.objectweb.joram.client.jms.local
- org.objectweb.joram.client.jms.tcp

Exceptions describing failing administration requests are of this class:

- org.objectweb.joram.client.jms.admin.AdminException

3.4.1. Administration “session”

Administration operations (calls to administration methods) may be performed within an administration “session”. Such a session is started when an administration connection is established with the JORAM platform to administer.

The utility class for managing administrator sessions is **`org.objectweb.joram.client.jms.admin.AdminModule`**.

TCP administrator connection

Such a connection is opened as follows:

```
AdminModule.connect(“host1”, 16010, “root”, “root”, 60);
```

This connects an application to a JORAM server running on “host1” and listening to port 16010 through the TCP protocol. It will work if the target server on “host1” provides the following services:

```
  <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
           args="root root"/>
  <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
           args="16010"/>
```

The last parameter of the connecting method (60), is the timer in seconds during which connecting to the server is attempted. This timer will be useful is the server is not yet started when the administration code is launched.

It is also possible to establish a “default” TCP connection to the server running on “localhost” and listening to port 16010 as follows:

```
AdminModule.connect(“root”, “root”, 60);
```

If the connecting request finally fails because the server is not reachable, the methods throw a `ConnectException`. If the administrator identification is incorrect, the methods throw an `AdminException`.

Disconnecting the administrator

The administration session ends by calling:

```
AdminModule.disconnect();
```

Any call to any administration method outside the `AdminModule.connect()` and `AdminModule.disconnect()` boundaries will fail (a `ConnectException` will be thrown).

3.4.2. Managing a user

User identity

Users are manipulated through the helper class `User` from package `org.objectweb.joram.jms.admin`. An instance of this class represents a given user and provides methods for administering it.

Creating a user

The creation of an user is done through the `create` method of the `User` administration class:

- `User.create(String name, String password)` is a static method setting a user with a given identification (name, password) and creating the corresponding `User` instance. The user is defined on the server the administrator is connected to (local server).
- `User.create(String name, String password, int server)` is similar to the previous method, except that it creates the user on the given server.
- `User.create(String name, String password, int server, String identity, Properties prop)` is a static method setting a user with a given identification on a given server, and creating the corresponding `User` instance. The identity class name and user's properties (server side interceptors for example) are specified in parameters.

```
User user = User.create("name", "pass", server, SimpleIdentity, prop);
```

An `AdminException` is thrown if the user creation fails server side or if the server is not part of the platform.

A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Setting a redelivery delay value

If needed you can delay the redelivery of messages when a temporary, external condition prevents your application from properly handling a message. When a message is denied you would prefer that to not redeliver it immediately and to handle it later. The redeliveryDelay property allows to define for all user's subscription the delay in seconds before a redelivery attempt.

Updating a user identity

The user's parameters can be update though the `update` method of the `User` administration class:

- `User.update(String newName, String newPass)`: updates the user identification.

```
user.update("newName", "newPass");
```

An `AdminException` is thrown if the user has been deleted server side, or if its new identification is already taken on its server.

A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Deleting a user

An user can be deleted though the `delete` method of the `User` administration class:

- `User.delete()`: deletes the user server-side.

```
user.delete();
```

The request is not effective if the user has already been deleted server side.

A `ConnectException` is thrown if the admin connection with the server is closed or lost.

3.4.3. User connectivity

A given user accesses the JORAM platform by connecting to a given server (set when actually creating the user, last section). The connection might be of different kinds:

- either a TCP or SSL connection;
- or intra-vm connection (documented in 4Using a collocated server).

The `javax.jms.ConnectionFactory` class is meant to determine to which server and through which protocol a client application will connect when calling the `createConnection` method.

Creating a `ConnectionFactory` instance for the TCP protocol

- `TcpConnectionFactory.create(String host, int port)`: static method creating a `ConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `TcpConnectionFactory.create()`: static method creating a `ConnectionFactory` instance for accessing the server the administrator is connected to.

```
ConnectionFactory cnxFact =  TcpConnectionFactory.create(
        "localhost", 16010);
```

- `QueueTcpConnectionFactory.create(String host, int port)`: static method creating a `QueueConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `QueueTcpConnectionFactory.create()`: static method creating a `QueueConnectionFactory` instance for accessing the server the administrator is connected to.
- `TopicTcpConnectionFactory.create(String host, int port)`: static method creating a `TopicConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `TopicTcpConnectionFactory.create()`: static method creating a `TopicConnectionFactory` instance for accessing the server the administrator is connected to.
- `XATcpConnectionFactory.create(String host, int port)`: static method creating a `XAConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `XATcpConnectionFactory.create()`: static method creating a `XAConnectionFactory` instance for accessing the server the administrator is connected to.
- `XAQueueTcpConnectionFactory.create(String host, int port)`: static method creating a `XAQueueConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `XAQueueTcpConnectionFactory.create()`: static method creating a `XAQueueConnectionFactory` instance for accessing the server the administrator is connected to.
- `XATopicTcpConnectionFactory.create(String host, int port)`: static method creating a `XATopicConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `XATopicTcpConnectionFactory.create()`: static method creating a `XATopicConnectionFactory` instance for accessing the server the administrator is connected to.

Setting the factory parameters

The following parameters may be set on a factory:

- Connecting timer: time (in seconds) during which connecting is attempted in case of failures.
- Transaction pending timer: time (in seconds) during which a transacted JMS session might be inactive before being automatically rolled back.
- Connection pending timer: time (in milliseconds) between two "ping" requests sent by the connection to the server; a connection is kept alive server side during twice the value of this parameter.

Those parameters are accessible through a FactoryParameters object (class `FactoryParameters` in package `org.objectweb.joram.client.jms`), obtainable by calling the `getParameters()` method on the factories.

When a client detects a connection failure, it automatically tries to reconnect every 2 seconds, during the period defined by the connecting timer parameter.

3.4.4. Managing a destination

Destinations are manipulated through the classes **`org.objectweb.joram.jms.Queue`** and **`org.objectweb.joram.jms.Topic`**. An instance of one of these classes represents a given destination and provides methods for administering it. Specialized destination management requires additional classes (see specific documentation: Specialized destinations).

Creating a destination: Queue or Topic

A destination can be created through the `create` method of the `Queue` or `Topic` administration class:

- `Queue.create(int server)`: static method creating a queue on a given server, and creating the corresponding `Queue` instance.
- `Queue.create()` is similar to the previous method, except that it creates the queue on the server the administrator is connected to (local server).
- `Topic.create(int server)`: static method creating a topic on a given server, and creating the corresponding `Topic` instance.
- `Topic.create()` is similar to the previous method, except that it creates the topic on the server the administrator is connected to (local server).

```
Queue queue = Queue.create();
Topic topic = Topic.create();
```

An `AdminException` is thrown if the destination deployment fails server side, or if the server is not part of the platform.

A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Creating a destination with a specified name

When creating a destination, queue or topic, you can specify an internal name[1]; if a destination exists with the specified name it is returned to the user, in the contrary case it is created and registered in the internal naming service.

- `Queue.create(int server, String name)`
  `Queue.create(String name)`: creates a queue on the given or default server with the specified name. If the named queue already exists it is simply returned.
- `Topic.create(int server, String name)`
  `Topic.create(String name)`: creates a topic on the given or default server with the specified name. If the named topic already exists it is simply returned.

Setting Properties

Destination's options are normally set using properties at queue/topic creation time. After creation the properties of the destination can be changed with the appropriate administration command: setProperties, or by using the corresponding MBean queue/topic component.

```
AdminReply reply = destination.setProperties(prop);
```

Many options are available depending of the destination real type, the following options is available for all destinations:

- "period": defines the time in millisecond to run tasks at regular interval: cleaning of out-of-date messages, etc.

[1] Not a JNDI's name.

Setting free access on a destination

- `Destination.setFreeReading()`: grants the READ right to all on the destination.

```
dest.setFreeReading();
```

- `Destination.setFreeWriting()`: grants the WRITE right to all on the destination.

```
dest.setFreeWriting();
```

- An `AdminException` is thrown if the destination has been deleted server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Unsetting free access on a destination

- `Destination.unsetFreeReading()`: removes the READ right to all on the destination.

```
dest.unsetFreeReading();
```

- `Destination.unsetFreeWriting()`: removes the WRITE right to all on the destination.

```
dest.unsetFreeWriting();
```

- An `AdminException` is thrown if the destination has been deleted server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Setting a right for a user on a destination

- `Destination.setReader(User user)`: sets a given user as a reader on the destination.

```
dest.setReader(user);
```

- `Destination.setWriter(User user)`: sets a given user as a writer on the destination.

```
dest.setWriter(user);
```

- An `AdminException` is thrown if the destination or the user does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Unsetting a right for a user on a destination

- `Destination.unsetReader(User user)`: unsets a given user as a reader on the destination.

```
dest.unsetReader(user);
```

- `Destination.unsetWriter(User user)`: unsets a given user as a writer on the destination.

```
dest.unsetWriter(user);
```

- An `AdminException` is thrown if the destination does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Getting the access rights

- `Destination.isFreelyReadable()`: returns `true` if the READ right is granted to all on the destination.
- `Destination.isFreelyWriteable()`: returns `true` if the WRITE right is granted to all on the destination.
- `Destination.getReaders()`: returns a `List` of users granted with the READ right on the destination.

- `Destination.getWriters()`: returns a `List` of users granted with the WRITE right on the destination.
  - An `AdminException` is thrown if the destination does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Handling the DMQ setting

- `Destination.setDMQ(DeadMQueue)`: sets the given Dead Message Queue as the default DMQ for the destination.
- `Destination.getDMQ()`: returns the default Dead Message Queue instance set for the destination, `null` if none.
  - An `AdminException` is thrown if the destination does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Deleting a destination

- `Destination.delete()`: deletes the destination.

```
dest.delete ();
```

The request is not effective if the destination does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

3.4.5. Managing a Queue

Getting the state

- `Queue.getPendingMessages()`: returns the number of messages on the queue waiting to be delivered.
- `Queue.getPendingRequests()`:returns the number of “receive” requests on the queue waiting for matching messages.
  - An `AdminException` is thrown if the queue does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Handling the queue threshold

The threshold value determine the maximum number of times a message can be denied. It allows to avoid an erroneous message to be delivered infinitely; the guilty message is then forwarded to the Dead Message Queue if any (deleted otherwise).

- `Queue.setThreshold(int threshold)`: sets the threshold value for the queue.
- `Queue.getThreshold()`: returns the threshold value set on the queue (-1 for none).
  - An `AdminException` is thrown if the queue does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Setting a redelivery delay value

If needed you can delay the redelivery of messages when a temporary, external condition prevents your application from properly handling a message. When a message is denied you would prefer that to not redeliver it immediately and to handle it later. The redeliveryDelay property allows to define at creation the delay in seconds before a redelivery attempt.

Handling the queue limit

A maximum number of undelivered messages can be set for the queue. Additional messages are forwarded to the Dead Message Queue if any (deleted otherwise).

- `Queue.setNbMaxMsg(int nbMaxMsg)`: sets the maximum number of undelivered messages for the queue (-1 for no limit).
- `Queue.getNbMaxMsg()`: returns the maximum number of undelivered messages value set on the queue (-1 for none).

3.4.6. Managing a Topic

A topic manages subsciptions. Subscription can be retrieved from User objects.

Getting the state

- `Topic.getSubscriptions()`: returns the number of active subscriptions on the topic.
- `Topic.getSubscriberIds()`: returns the list of user's proxy ids registered.
    - An `AdminException` is thrown if the topic does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

3.4.7. Managing the platform

Methods are also provided for getting information about how the platform has been configured. Data is available at the platform, server, destination and user levels.

Getting the servers of the platform

- `AdminModule.getServersIds()`: returns a `List` containing the identifiers of all the servers involved in the monitored JORAM platform.
    - A `ConnectException` is thrown if the admin connection to the server is closed or lost.

Handling default DMQ settings

- `AdminModule.getDefaultDMQ(int serverId)`: returns the `DeadMQueue` instance representing the default DMQ of a given server, null if none.
    - An `AdminException` is thrown if the target server does not belong to the platform. A `ConnectException` is thrown if the admin connection to the server is closed or lost.
- `AdminModule.getDefaultDMQ()` is similar to the previous method, except that it returns the default DMQ of the server the administrator is connected to.
    - A `ConnectException` is thrown if the admin connection to the server is closed or lost.
- `AdminModule.getDefaultThreshold(int serverId)`: returns the default threshold value of a given server.
    - An `AdminException` is thrown if the target server does not belong to the platform. A `ConnectException` is thrown if the admin connection to the server is closed or lost.
- `AdminModule.getDefaultThreshold()` is similar to the previous method, except that it returns the default threshold of the server the administrator is connected to.
    - A `ConnectException` is thrown if the admin connection to the server is closed or lost.

Getting the destinations

- `AdminModule.getDestinations(int serverId)`: returns a `List` containing `Destination` instances representing all the destinations deployed on a given server.
    - An `AdminException` is thrown if the target server does not belong to the platform. A `ConnectException` is thrown if the admin connection to the server is closed or lost.

- `AdminModule.getDestinations()` is similar to the previous method, except that it returns the destinations of the server the administrator is connected to.
  - A `ConnectException` is thrown if the admin connection to the server is closed or lost.

Getting the users

- `AdminModule.getUsers(int serverId)`: returns a `List` containing `User` instances representing all the users set on a given server.
  - An `AdminException` is thrown if the target server does not belong to the platform. A `ConnectException` is thrown if the admin connection to the server is closed or lost.
- `AdminModule.getUsers()` is similar to the previous method, except that it returns the users of the server the administrator is connected to.
  - A `ConnectException` is thrown if the admin connection to the server is closed or lost.

3.5. Message interceptors

To be able to intercept or transform data of messages that applications attempt to send on destinations or get from them, you can use message interceptors. There are two types of interceptors:

- **The JMS client interceptor**: Configured on the ConnectionFactory the JMS client interceptor allows to intercept each message sent or receive through the client connection (see section 3.5.1). The messages are handled in the client's context before the sending or after the receiving.
- **The server interceptor**: Such an interceptor can be used to handle messages during their transit in the MOM (see section 3.5.2). The messages are handled in the server's secure context, only the authorized users can manage this feature.

The architecture is designed for flexibility then you can chain as many interceptors objects as you want. Using a chain you can achieve complex patterns and add functionality by composition.

Sample features you can do with an Interceptor are for example:

- Logging, auditing or validating messages
- Alerts,
- Filtering,
- Content enrichment and tagging,
- and many other cases according to your imagination.

3.5.1. Managing client interceptors

JORAM allows special objects called interceptors to intercept and handle each message entering or exiting the client context through a configured connection.

Writing an interceptor

Writing an interceptor is quite easy, just create a class that implements the *MessageInterceptor*[2] interface. This interface defines a unique method *handle*, The method parameters are the JMS message and the current session. This method does not return values and must not throw

[2] Package **`org.objectweb.joram.client.jms`**

exceptions. By convention, the implementation can modify the original message or the current runtime context.

```
public interface MessageInterceptor {
  public void handle(Message pMessage, Session pSession);
}
```

Configuring a ConnectionFactory with interceptors

The interceptor chain is activated on each message sent or received by the configured session. For a sent message the processing is done before the transfer, for a received message it is done between the recceipt of the message client side and the return to the message consumer.

Normally the interceptors chains are configured at the creation of the ConnectionFactory administered object. It can be achieved either through the administration API or the XML scripting langage. The configuration is done on the FactoryParameters of ConnectionFactory by adding (or removing) individually each interceptor.

Using administration API:

```
  public void addInInterceptor(String pInterceptorClassName);
  public boolean removeInInterceptor(String pInterceptorClassName);

  public void addOutInterceptor(String pInterceptorClassName);
  public boolean removeOutInterceptor(String pInterceptorClassName);
```

In order to configure the interceptors of a destination in a XML script you simply have to add the definition of the interceptor's chains in the ConnectionFactory declaration. For example:

```
<ConnectionFactory name="…" className="…">
  …
  <inInterceptors>
    <interceptor className="interceptor IN classname #1"/>
    <interceptor className="interceptor IN classname #2"/>
  </inInterceptors>

 <outInterceptors>
    <interceptor className="interceptor OUT classname #1"/>
    <interceptor className="interceptor OUT classname #2"/>
 </outInterceptors>
 …
</ConnectionFactory>
```

3.5.2. Managing server interceptors

JORAM allows some special objects called Interceptors to handle messages during their transit in the MOM[3], there are two types of interceptors:

- An user's interceptors (see “Configuring interceptors on user's connections”) receives each message that’s entering or exiting the server.
- A destination's interceptor (see “Configuring interceptors on destination”) receives each messages sent to this destination.

Interceptors can read and also write into the messages. This enables filtering, transformation or content enrichment, for example adding a property into the message. Also Interceptors can stop

[3] It exists also a mechanism allowing the definition of interceptors client-side.

the Interceptor chain by simply returning false to their intercept method invocation, in this case the transmission of the message is stopped.

Writing an interceptor

Writing an interceptor is quite easy, just create a class that implements the *MessageInterceptor*[4] interface.

```
public interface MessageInterceptor {
  boolean handleMessage(Message msg);
}
```

The only method to implement is *handleMessage*, this method returns a boolean result. If it is true the server continue to process the interceptor chain, if it is false the server stops executing the chain and removes the message (the message is sent to the corresponding DMQ if any).

The method parameter is a Message[5] object containing both the header and the payload of the corresponding message.

Each interceptor can observe the state of the message and modify it. From the JORAM server point of view the interceptors are stateless, If you need to keep datas, you must do it yourself.

All interceptors class must be accessible by the class loader of the JORAM server (possibly dynamically load in the case of a server running on OSGi).

Configuring interceptors on destination

The interceptor chain is activated on each message that reaches the destination. This processing is done before storing the message for a queue destination or before forwarding the message for a topic one.

Initial configuration

The interceptor chains can be simply configured at the creation of a the destination using either the administration API or the XML scripting language. It only needs to defines the corresponding property (jms_joram_interceptors) with the comma separated list of the classname of your interceptors.

Using administration API, you simply have to call the appropriate create method of the Queue or Topic administration class with a properties object defining the jms_joram_interceptors property. For example for a Queue:

```
Properties properties = new Properties();
properties.setProperty("jms_joram_interceptors",
                       "interceptors classname list");
Queue.create(..., properties, ...);
```

In order to configure the interceptors of a destination in a XML script you simply have to add the definition of the "jms_joram_interceptors" property in the destination declaration. For example for a Queue:

```
<Queue name="queue">
  <property name="jms_joram_interceptors" value="interceptors class name list"/>
...
</Queue>
```

Administration changes

Once the destination is created you can always change the interceptor's chains, adding a list of interceptors, removing or replacing some of them. The Queue and Topic class of the administration API offer the following methods:

[4] Package **org.objectweb.joram.mom.util**
[5] Package **org.objectweb.joram.shared.messages**

```
public String getInterceptors();
public void addInterceptors(String interceptors);
public void removeInterceptors(String interceptors);
public void replaceInterceptor(String newInterceptor, String oldInterceptor);
```

An `AdminException` is thrown if the destination does not exist on the server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Configuring interceptors on user's connections

There is two distinct chains of interceptors. The first one “interceptors_in” handles each message that’s entering the server (result of a send method on a connection from the selected user). The second one “interceptors_out” handles each message that’s exiting the server (result of a receive method on a connection from the selected user). These two interceptor chains are configurable for each user.

Initial configuration

The interceptor chains can be simply configured at the creation of a user using either the administration API or the XML scripting language. It only needs to defines the corresponding property (jms_joram_interceptors_in or jms_joram_interceptors_out) with the comma separated list of the classname of your interceptors.

Using administration API, you simply have to call the appropriate create method of the User administration class with a properties object defining the jms_joram_interceptors_in and jms_joram_interceptors_out properties:

```
Properties properties = new Properties();
properties.setProperty("jms_joram_interceptors_in",
                       "interceptors classname list");
properties.setProperty("jms_joram_interceptors_out",
                       "interceptors classname list");
User.create(..., properties);
```

In order to configure the interceptors of an user in a XML script you simply have to add the definition of the “jms_joram_interceptors_in” and/or “jms_joram_interceptors_out” properties in the user declaration:

```
<User name="anonymous" password="anonymous">
  <property name="jms_joram_interceptors_in"
            value="interceptors classname list"/>
  <property name="jms_joram_interceptors_out"
            value="interceptors classname list"/>
  ...
</User>
```

Administration changes

Once the user is created you can always change the interceptor's chains, adding a list of interceptors, removing or replacing some of them. The User class of the administration API offers the following methods:

```
public String getInterceptorsIN();
public void addInterceptorsIN(String interceptors);
public void removeInterceptorsIN(String interceptors);
public void replaceInterceptorIN(String newInterceptor, String oldInterceptor);

public String getInterceptorsOUT();
public void addInterceptorsOUT(String interceptors);
```

```
public void removeInterceptorsOUT(String interceptors);
public void replaceInterceptorOUT(String newInterceptor, String oldInterceptor);
```

An `AdminException` is thrown if the user agent does not exist on the server side. A `ConnectException` is thrown if the administration connection with the server is closed or lost.

Running a test

Have a look to the JORAM tests suite, src/joram/interceptors.

Run all interceptors tests: `ant tests.interceptors`

Run a single target: `ant interceptors.testXXX`

3.6. JMX administration of Joram

You can configure your Joram server to export some MXBean, so you can monitor and handle it through a JMX console. This example is designed in a JDK 1.5 environment with the integrated JMX 1.2 implementation.

To launch a Joram server with JMX capabilities enabled, you just have to fix the environment variable MXServer; for example typing `-DMXServer=com.scalagent.jmx.JMXServer` in the command line.

In order to allow a remote access to theses beans, you may either declare the JMXRIHttpService (`com.scalagent.jmx.JMXRIHttpService` class) in `a3servers.xml` configuration file or use a standard adapter. For example, if using JDK1.5, you can declare `-Dcom.sun.management.jmxremote` in the command line and then use the jconsole graphical tool to browse the beans:

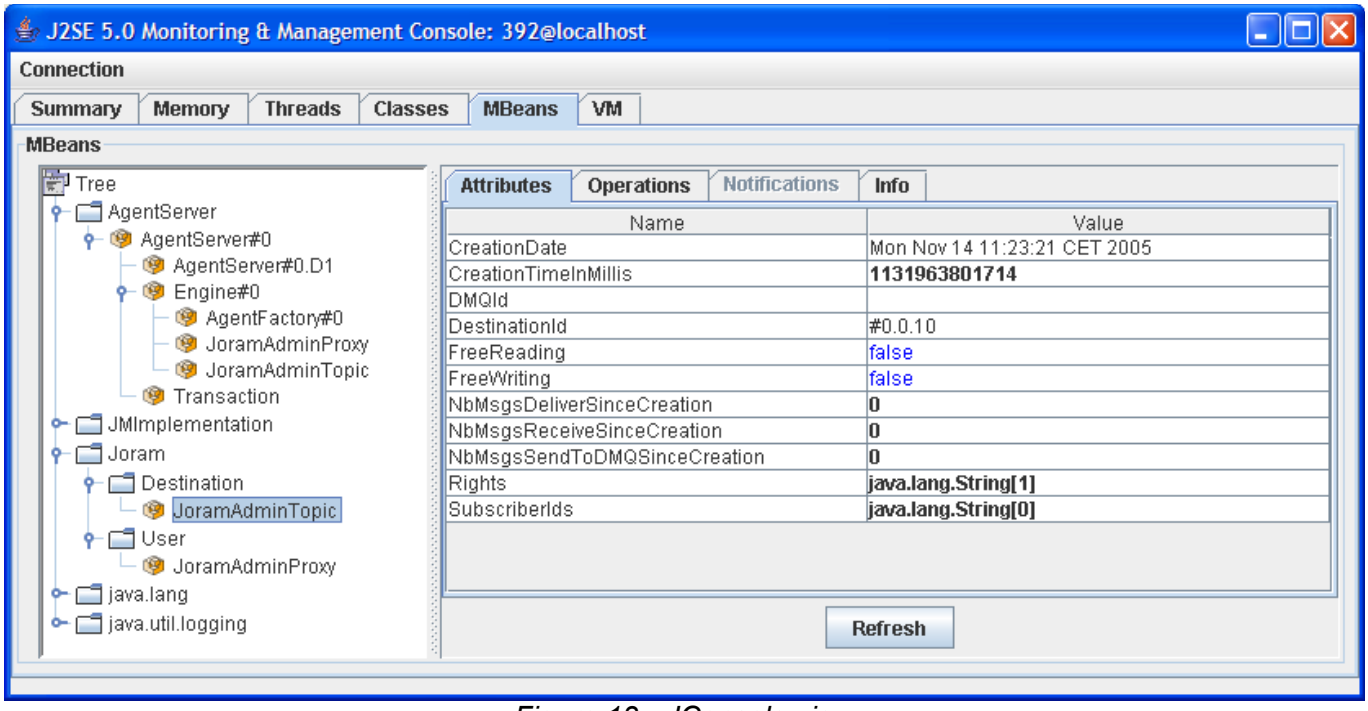

Figure 13 - JConsole view

At starting you there is two nodes added in the MBean's tree:

- The first one, named **AgentServer**, describes the ScalAgent platform: domains and networks, engine and agents.
- The second one, named **Joram**, allows the handling of Joram's users and destinations.

3.7. Scripts XML

This feature allows to execute administration operation using an XML script. It is possible to create and bind in JNDI connection factories, destinations and users. The complete DTD is available in SVN and examples are available with the samples.

3.7.1. Administrator connection

There is three different elements allowing to establish the administration connection:

- `LocalAdminModule` establishes a connection with a colocated JMS provider, it needs to define the login and password attributes.
- `TcpAdminModule` establishes a connection with a remote JMS provider, additionally to login and password it needs to define the hostname (default “localhost”) and listen port (default 16010) of the Joram's TCP service.
- `SSLAdminModule` establishes a connection with a remote JMS provider using SSL, it needs the same attributes than a `TcpAdminModule`.
- <TcpAdminModule host="localhost" port="16010" name="root" password="root">
- <property name="connectingTimer" value="60"/>
- </TcpAdminModule>

Additionally you can add `property` elements to configure the factory's parameters of the underlying connection factory (see example above).

- Naming

If you want to register created objects in a JNDI's repository you have to declare an InitiallContext element defining properties (see example at the end of this section):

- `java.naming.factory.initial`,
- `java.naming.factory.host` and
- `java.naming.factory.host`.

3.7.2. ConnectionFactory

There is three different elements allowing to define a ConnectionFactory:

`LocalConnectionFactory` defines a connection factory to a colocated JMS provider.

- `TcpConnectionFactory` defines a connection factory to a remote JMS provider, it needs to define the hostname (default “localhost”) and listen port (default 16010) of the Joram's TCP service.
- `SSLConnectionFactory` defines a connection factory to a remote JMS provider using SSL, it needs the same attributes than a `TcpConnectionFactory`.

A `ConnectionFactory` element can be named (name attribute) for later use in the script (building of clustered destination's for example). It can be completed by:

- `property` elements to configure the factory's parameters of this connection factory,
- `inInterceptors` or `outInterceptors` to define the list of interceptors.
- Example:

```
<TcpConnectionFactory name="cf" host="localhost" port="16010">
  <property name="queueMessageReadMax" value="10"/>
  <property name="topicAckBufferMax" value="5"/>
  <outInterceptors>
```

```
        <interceptor className="classic.Interceptor"/>
    </outInterceptors>
    <jndi name="cf"/>
  </TcpConnectionFactory>
```

3.7.3. User

A user definition is a simple XML element, you must at least define name and password properties:

- `name`: the name of the user needed for later use in the script (handling of destination's rights for example).
- `login, password`: login and password for user, if the login is not fixed the name is used by default.
- `serverId`: unique identifier of location server. If not set the user is created on the server the administrator is connected.
- `dmq, threshold`: Dead Message Queue settings for the user.

properties

- Additional properties can be defined for user. Each property is an element with two attributes: `name` and `value`.

3.7.4. Destination

The syntax allows to create queue, topic and Dead Message Queue, specialized destinations can be deployed specifying the MOM's implementation class of the destination:

Queue

A queue definition defines some optional properties, it can be completed by properties, naming or security elements:

- `name`: the Joram's internal name for the queue.
- `serverId`: unique identifier of location server. If not set the queue is created on the server the administrator is connected.
- `dmq, threshold`: Dead Message Queue settings for the queue.
- `nbMaxMsg`: Maximum of pending messages in the queue.
- `className`: the real class name of the MOM's destination. By default a simple queue, `org.objectweb.joram.mom.dest.Queue`.

Topic

A topic definition defines some optional properties, it can be completed by properties, naming or security elements:

- `name`: the Joram's internal name for the topic.
- `serverId`: unique identifier of location server. If not set the topic is created on the server the administrator is connected.
- `parent`: the internal name of the hierarchical parent of this topic.
- `className`: the real class name of the MOM's destination. By default a simple topic, `org.objectweb.joram.mom.dest.Topic`.

Destination security

Destination element can be completed by security settings:

- freeReader: Grants the read right to all users on this destination.
- freeWriter: Grants the write right to all users on this destination.
- reader: Sets a user as a potential reader on the destination, the user name is given in the attribute user.
- writer: Sets a user as a potential writer on the destination, the user name is given in the attribute user.

Naming

The `jndi` sub-element allows to register the destination in JNDI context, the symbolic name is given in the `name` attribute.

Properties

Additional properties can be defined for destinations, each property is set by an `property` element with two attributes: `name` and `value`.

3.7.5. Example

In the example below (from the classic sample) we first define an administration connection through TCP to the local host on port 16010. The administrator's login is “root” and the password is “root”.

Remark: as their values are the default ones, these parameter's definitions can be omitted.

A TCPConnectionFactory (localhost:16010) is defined and bind in JNDI (name “cf”).

A user named “anonymous is created (password “anonymous”), then a queue named “queue” and a topic named “topic” are created. All these objects are created on the default server. The Read and write right are granted for all, the queue is bind in JNDI with the name “queue”, and the topic with the name “topic”.

```
<JoramAdmin>
  <TcpAdminModule host="localhost" port="16010" name="root" password="root">
    <property name="connectingTimer" value="60"/>
  </TcpAdminModule>

  <InitialContext>
    <property name="java.naming.factory.initial"
              value="fr.dyade.aaa.jndi2.client.NamingContextFactory"/>
    <property name="java.naming.factory.host" value="localhost"/>
    <property name="java.naming.factory.port" value="16400"/>
  </InitialContext>

  <TcpConnectionFactory name="cf" host="localhost" port="16010">
    <jndi name="cf"/>
  </TcpConnectionFactory>

  <User name="anonymous" password="anonymous"/>
```

```
  <Queue name="queue">
    <freeReader/><freeWriter/>
    <jndi name="queue"/>
  </Queue>

  <Topic name="topic">
    <freeReader/><freeWriter/>
    <jndi name="topic"/>
  </Topic>
</JoramAdmin>
```

3.8. OSGi Commands

Administration commands are now available in Joram to manipulate the server. These commands are packaged in three additional bundles:

- *shell-a3 for commands related to the underlying A3 middleware*,
- *shell-mom* for commands related to the MOM,
- *and shell-jndi for commands related to the embedded implementation of JNDI*.

An adequate OSGi configuration file is shipped in the samples/config directory:

- config_shell.properties

3.8.1. A3 commands

The *shell-a3* bundle give access to commands allowing to administrate the A3 server on which Joram is running.

- `[joram:a3:]close`: This commands stops the Joram and exits the server application.
- `[joram:a3:]engineLoad`: This commands returns the average number of pending notifications over the last minute, it is an evaluation of the server's load.
- `[joram:a3:]garbageRatio`: This commands gives the garbage ratio when the NG transaction mode is set.
- `[joram:a3:]info [options...]`: This commands shows numerous information about the server, the network (in the case of clustering) and/or the transactional persistence manager. Options:
  - `-eng`: Shows info about the engine,
  - `-ngt`: Shows info about the transactional persistence manager,
  - `-net`: Shows info about the network (in the case of clustering).
  - By default, the commands shows info about all available components.
- `[joram:a3:]restartServer`: Stops and restarts Joram
- `[joram:a3:]startServer`: Starts Joram
- `[joram:a3:]stopServer`: Stops Joram

3.8.2. MOM commands

The *shell-mom* bundle offers commands allowing to monitor and administrate the destination, subscriptions and users.

- `[joram:mom:]clear (queue <name> | subscription <username> <sub. Name>`: Clears all pending messages of the queue or suscription.
- `[joram:mom:]create (queue|topic) <name> [options]`: Creates a new queue or topic. Options:
  - `-sid` : ID of the server on which the destination must be deployed.
  - `-ext`: Extension class of the destination.
- `[joram:mom:]create user <name>`: Creates a new user. The command prompts for a password.
- `[joram:mom:]delete (topic|queue|user) <name>`: Deletes a destination or a user.
- `[joram:mom:]deleteMsg (queue <queue name>|subscription <user name> <sub. name>) <msg id>`: Delete a pending message from a queue or a subscription.
- `[joram:mom:]help [<command name>]`: Displays usage information about the specified MOM command. If no command is given, lists all MOM commands.
- `[joram:mom:]info (queue <name> | topic <name> | subscription <user name> <sub. name>)`: Gives info about the destination or subscription.
- `[joram:mom:]list (destination | topic | queue | user | subscription <user name>)`: Lists all destinations, topics, queues, users or user's subscriptions.
- `[joram:mom:]lsMsg (queue <queue name>|subscription <user name> <sub. name>) [[first]:[last]]`: Lists the pending messages in the given interval if any, all otherwise.
- `[joram:mom:]ping`: Tells whether Joram is running.
- `[joram:mom:]queueLoad` <queue name>: Gives the average queue load.
- `[joram:mom:]receiveMsg`: Not yet implemented.
- `[joram:mom:]sendMsg`: Not yet implemented.
- `[joram:mom:]subscriptionLoad <user name> <sub. name>`: Gives the average subscription load.

3.8.3. JNDI commands

The *shell-jndi* bundle currently give access only to the `joram:jndi:list` command. This commands lists all records in the naming context, giving useful info about the registered destinations and connection factories.

Specialized destinations

3.9. Dead Message Queue

3.9.1. Introduction

Any queue can be a dead message queue. In fact, a dead message queue is a classic queue, but used in a particular way: its purpose is to collect dead messages. A dead message is a message located server side and considered as undeliverable for various reasons. The reason can be:

- the target destination does not exist,
- the sender does not have the writing right on the target destination,
- the message expires before it is delivered,
- the message is constantly denied by the consuming client,
- the maximum number of messages in the queue has been reached,

- the message has been deleted on the queue.

An application may also consider a message it got has to be sent to the DMQ. This “manual” sending is allowed to any application.

The Figure 14 shows an example of DMQ usage. A DMQ has been set as the DMQ of a given queue. This queue receives a message from a producer and tries to deliver it to a consumer. This consumer keeps denying the received message. When the number of delivery attempts overtakes a given threshold value, the message is removed from the queue and sent to the DMQ.

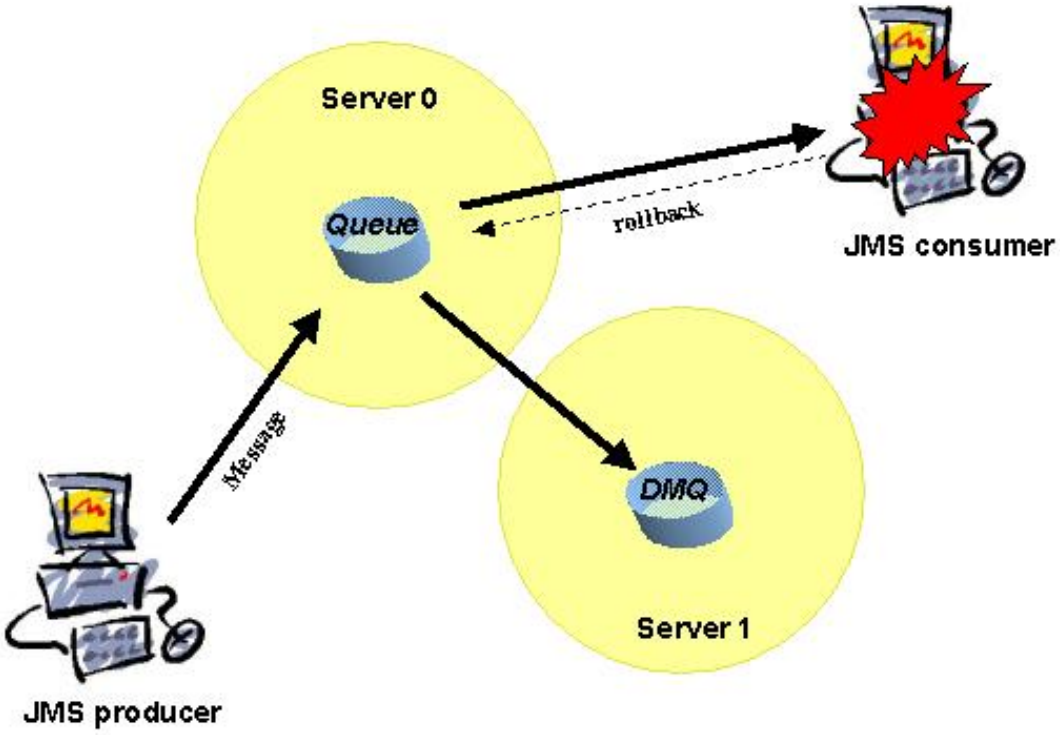

Figure 14 - Messages on a queue sent to a DMQ

Creating and setting a dead message queue

As any destination, a dead message queue may be deployed on any server of the configuration, even if it is intended to log dead messages of destinations located on other servers.

The setting of a dead message queue may take place at various levels. A dead message queue may be set as the dead message queue for:

- the destinations and users on a given server (it is then considered as the default DMQ for this server),
- a given destination,
- a given user.

A threshold value may also be set. If set, this value is the number of times a message may be delivered to a consumer before being considered as undeliverable. Its setting takes place at the same levels as for DMQs:

- as the default value for the queues and subscribers of a given server,
- for a queue,
- for a user.

The settings for a given destination and a given user precede the default settings (see the scenarios). No setting means that message is indefinitely delivered, even to failing consumers.

Scenarios

1. the target destination does not exist: the produced messages are sent to the producer's DMQ if set, or to the default producer server's DMQ if set.
2. the target destination is not writable: the produced messages are sent to the producer's DMQ if set, or to the default producer's server's DMQ if set, or to the destination's DMQ if set, or to the default destination's server's DMQ if set.
3. a message expires on a queue: it is sent to the queue's DMQ if set or to the queue's server's default DMQ if set.
4. a message on a queue reaches the maximum delivery attempts: it is sent to the queue's DMQ if set, or to the queue's server's default DMQ if set; the threshold value is the queue's one if set, or the queue's server's default one if set.
5. a message for a given subscriber expires: it is sent to the subscriber's DMQ if set, or to the subscriber's server's default DMQ if set.
6. a message for a given subscriber reaches the maximum delivery attempts: it is sent to the subscriber's DMQ if set, or to the subscriber's server's default DMQ if set; the threshold value is the subscriber's one if set, or the subscriber's server's default one if set.

Watching a dead message queue

Accessing a dead message queue through a JMS client means that the DMQ has preliminary been bound in a name space like JNDI, as any "normal" destination. Also, watching a dead message queue requires a JMS client granted with a READ access on it.

The client may consume or browse the queue. Remember DMQ is a "normal" queue, the only difference is the origin of the messages. It can even log its own messages as dead messages on other DMQs.

Dead messages carry special properties describing why they were considered as "dead". Those properties are:

- `JMS_JORAM_ERRORCOUNT`, this property is mapped to an integer telling the number of consecutive errors which happened.
- `JMS_JORAM_ERRORCODE_X`, with 1 ≤ X ≤ ERRORCOUNT returns the error code of the error number X.
- `JMS_JORAM_ERRORCAUSE_X`, with 1 ≤ X ≤ ERRORCOUNT is mapped to a string detailing the error. This can be one of the following:
  - `Deleted destination`, if the target destination of the message could not be found,
  - `Destination is not writable`, if the target destination of the message did not accept the sender as a WRITER,
  - `Expired at XXX`, if the message expired before delivery. The `XXX` stands for a long number holding the date when the message expired.
  - `Undeliverable after X tries`, if the number of delivery attempts of the message overtook the threshold. The `X` is replaced by the threshold value.
  - `Message deleted by an admin`, if the message being dead is the result of an admin deletion with `queue.deleteMessage(String msgId)` or `queue.clear()` methods.
  - `Queue full`, if the queue has reached its maximum number of messages.
  - `Unexpected error`, if there was an unexpected error, for example a connection problem while using a mail queue.

The `JMSXDeliveryCount` property is also available for getting the number of delivery attempts of the message, including the delivery to the DMQ consumer. All those properties are available trough the dedicated `Message` methods. A typical check can be as following:

```
// Getting a dead message through a DMQ consumer:
```

```
Message deadM = (Message) deadMconsumer.receive();

int errorCount = deadM.getIntProperty("JMS_JORAM_ERRORCOUNT");
for (int i = 1; i <= errorCount; i++) {
  System.out.println(deadM.getIntProperty("JMS_JORAM_ERRORCODE_" + i));
  System.out.println(deadM.getIntProperty("JMS_JORAM_ERRORCAUSE_" + i));

  // Do specific things if the message has expired.
  if(deadM.getIntProperty("JMS_JORAM_ERRORCODE_" + i) ==
          MessageErrorConstants.EXPIRED) {
    ...
  }
}
```

3.9.2. Managing a Dead Message Queue

Creating a dead message queue

A dead message queue is basically a "normal" queue:

- `Queue.create(int server)`: creates a queue on a given server, and instantiates the corresponding `Queue` object.
- `Queue.create()` is similar to the previous method, except that it creates the dead message queue on the server the administrator is connected to.

```
Queue dmq = Queue.create(0);
```

- An `AdminException` is thrown if the destination deployment fails server side, or if the server is not part of the platform. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Setting a dead message queue

- AdminModule.setDefaultDMQ(int serverId, Queue dmq): sets a given DMQ as the default DMQ for the destinations and users on a given server (set DMQ as null for unsetting it).
- AdminModule.setDefaultDMQ(Queue dmq) is similar to the previous method except that it sets the DMQ on the server the administrator is connected to.

```
AdminModule.setDefaultDMQ(0, dmq);
```

- A `ConnectException` is thrown if the admin connection to the server is closed or lost. An `AdminException` is thrown if the server is not known in the platform, or if the DMQ has been deleted server side.

- Destination.setDMQ(Queue dmq): sets a given DMQ as the DMQ for the destination (set DMQ as null for unsetting it).
- User.setDMQ(Queue dmq): sets a given DMQ as the DMQ for the user (set DMQ as null for unsetting it).

```
topic.setDMQ(dmq);
user.setDMQ(null);
```

- An `AdminException` is thrown if the destination or the user has been deleted server side. A `ConnectException` is thrown if the admin connection to the server is closed or lost.

Setting a threshold value

- AdminModule.setDefaultThreshold(int serverId, int threshold): sets a given value as the default threshold for the queues and users on a given server (set threshold to -1 for unsetting it).

- AdminModule.setDefaultThreshold(int threshold) is similar to the previous method except that it sets the threshold on the server the administrator is connected to.

```
AdminModule.setDefaultThreshold(0, 5);
```

  - A `ConnectException` is thrown if the admin connection to the server is closed or lost. An `AdminException` is thrown if the server is not known in the platform.
- Queue.setThreshold(int threshold): sets a given value as the threshold for the queue.
- User.setThreshold(int threshold): sets a given value as the threshold for the user.

```
queue.setThreshold(5);
user.setThreshold(-1);
```

  - An `AdminException` is thrown if the queue or the user has been deleted server side. A `ConnectException` is thrown if the admin connection to the server is closed or lost.

Setting a redelivery delay value

In some situations (e.g. service temporarily unavailable), if the faulty message is redelivered straightaway without any delay, this would do no good and only increase the load of the server. In this case you would prefer that to not redeliver the faulty message immediately and to try to handle it later.

The joram.jms.redeliveryDelay global property allows to define a default delay in seconds before a redelivery attempt.

3.9.3. Running the “Dead Message Queue” sample

The dead message queue sample simulates various cases where messages are considered as undeliverable. It involves a message producer, a failing consumer, and a DMQ watcher actually consuming the messages on the DMQ.

The next picture shows the DMQ configuration. The configuration used is centralized and the server run in non persistent mode

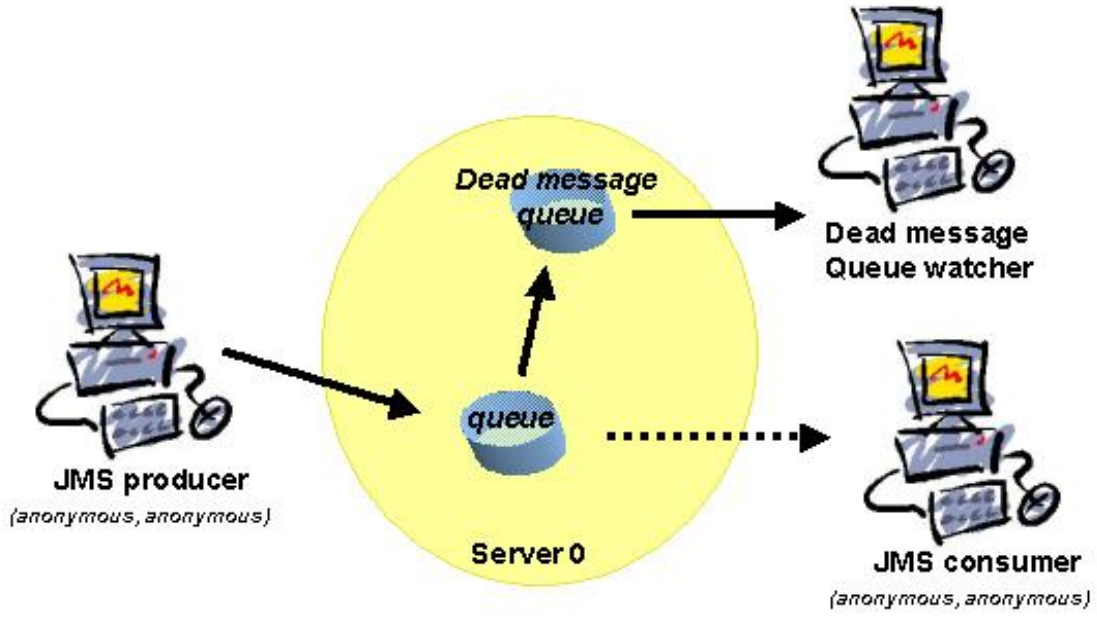

Figure 15 - Dead message queue sample

Running the demo:

- Start the platform:

```
ant reset single_server
```

- Run the admin code:
  ```
  ant dmq_admin
  ```
- Launch the watcher:
  ```
  ant dmq_watcher
  ```
- Launch the producer and the consumer:
  ```
  ant dmq_client
  ```

3.10. Hierarchical Topic

3.10.1. Hierarchical topic

Introduction

The JMS specification allows topics to have a hierarchical structure such as the one shown in Figure 16. The interest of such a structure is to allow a subscriber to specifically choose the type of information it is interested in, by allowing it to subscribe to the corresponding subtopic. To the contrary, a subscriber may want to get all the sub information by subscribing to the topic root or father.

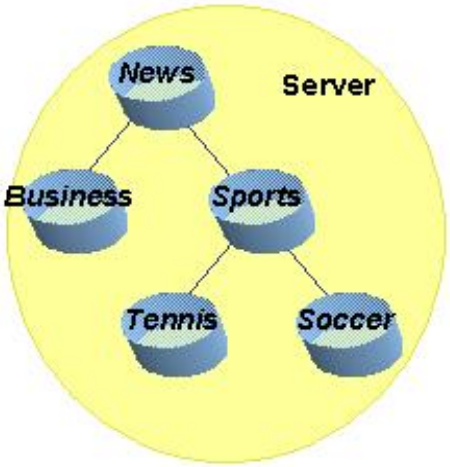

Figure 16 - A Hierarchical topic

Example

The example of Figure 16 shows a hierarchy of news. A subscriber to the Tennis topic would only get the news concerning tennis, whereas a subscriber to the Sports topic would get the news concerning tennis and soccer and sports in general. Also, a subscriber to the Business topic would get business information only, whereas a subscriber to the News topic will get the business related news, the sports related news, and news in general.

Creation

Creating such a hierarchy requires first to create the topics that will constitute it, then to notify each topic of the hierarchy it is part of.

Each topic of the hierarchy may be bound in a name space such as JNDI, so that clients may then retrieve them. Access rights are managed individually, on each topic of the hierarchy.

Distributed deployment

To be noted, a hierarchy may be spread over many servers (Figure 17). A distributed architecture introduces flexibility and availability. If server 1 is down, for example, the News and Business leafs

of the hierarchy would go on working. Subscribers to the news and to the business would get information related to news and business (News subscribers would get nothing related to sports until server 1 is started again).

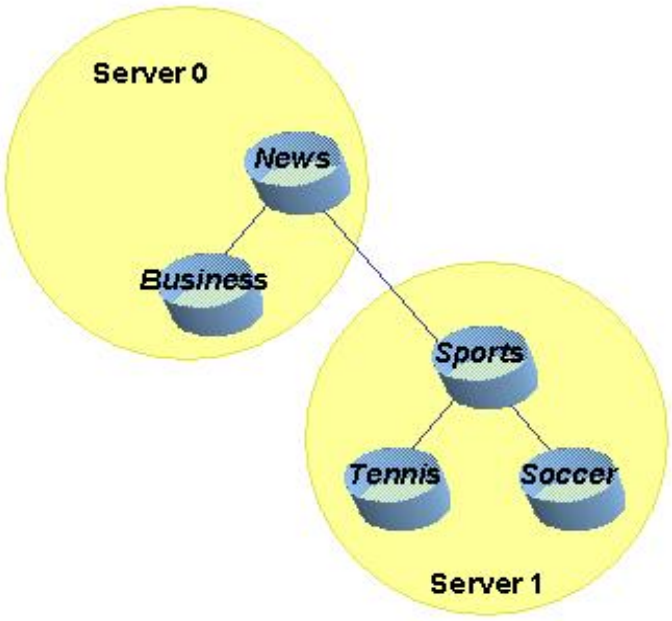

Figure 17 - A distributed Hierarchical topic

3.10.2. Managing a Hierarchical Topic

Creating a hierarchical topic

Creating a hierarchical topic requires first to create all the topics of the hierarchy. If we consider the example shown on figure 6:

```
Topic news = Topic.create(0);
Topic business = Topic.create(0);
Topic sports = Topic.create(0);
Topic tennis = Topic.create(0);
Topic soccer = Topic.create(0);
```

The hierarchy needs then to be constructed. Topics are linked two by two with the following method:

- `Topic.setParent(Topic father)`: sets a given topic as the father of an other topic.

Going back to our example:

```
business.setParent(news);
sports.setParent(news);
tennis.setParent(sports);
soccer.setParent(sports);
```

An `AdminException` is thrown if one of the topics has been deleted server side, or is already part of a cluster, or if the son parameter already has a father. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Modifying a hierarchy

A hierarchy might be modified either by adding a new branch, or by modifying the existing ones, or by removing the existing ones. The following method is provided:

- `Topic.unsetParent()`: unsets the father of the topic.

For example, for unsetting the link between the sports related informations and the general news:

```
sports.unsetParent();
```

Subscribers to the `sports` topic would still get the `tennis` and `soccer` news, but subscribers to the `news` topic would not get anything related to `sports`.

A `ConnectException` is thrown if the admin connection with the server is closed or lost. An `AdminException` is thrown if the topic has been deleted server side or does not have a father.

Also, removing a topic of the hierarchy removes the links this topic was involved in. For example, by calling:

```
sports.delete();
```

the `tennis` and `soccer` topics become independent. News subscribers won't get anything related to tennis or soccer.

Getting info about cluster or hierarchy

- `Topic.getClusterFellows()`: returns a `List` of `Topic` instances representing other topics part of a same cluster.
- `Topic.getHierarchicalFather()`: returns a `Topic` instance representing the topic set as hierarchical father.

An `AdminException` is thrown if the topic does not exist server side. A `ConnectException` is thrown if the admin connection with the server is closed or lost.

3.10.3. Running the topic tree sample

The topic tree sample illustrates the use of a hierarchical topic. A producer produces various informations destinated to the leafs of a hierarchical topic: news, business, sports, tennis. A consumer subscribes to all these leafs. Its subscription to the news will get all the messages. Its subscription to the business information will only get the messages related to business. Its subscription to the sports will get all the sports-related messages, and its subscription to the tennis news will only get the messages about tennis.

The next picture shows the topic tree configuration. The configuration used is centralized and the server run in non persistent mode.

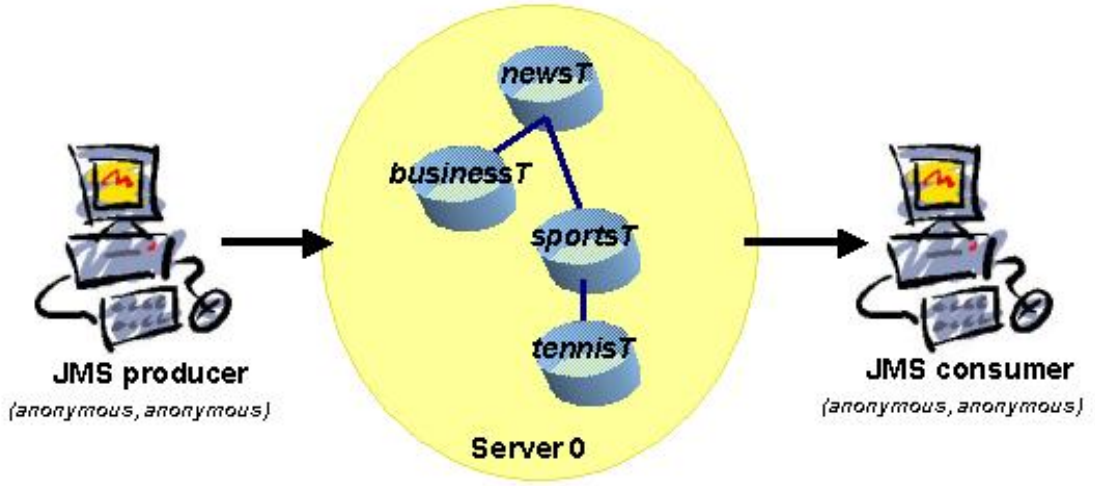

Figure 18 - Topic tree sample

Running the demo:

- For starting the platform:
  `ant reset single_server`
- For running the admin code:
  `ant tree_admin`
- For launching the consumer:
  `ant tree_consumer`
- To start the producer:
  `ant tree_producer`

3.11. Clustered Topic

3.11.1. Introduction

A non hierarchical topic might also be distributed among many servers. Such a topic, to be considered as a single logical topic, it made of topics representatives, one per server. Figure 19 shows such a topic located on three servers.

Such an architecture allows a publisher to publish messages on a representative of the topic. In the example shown in Figure 19, the publisher works with the representative on server 1. If a subscriber subscribed to any other representative (on server 2 in our example), it will get the messages produced by the publisher.

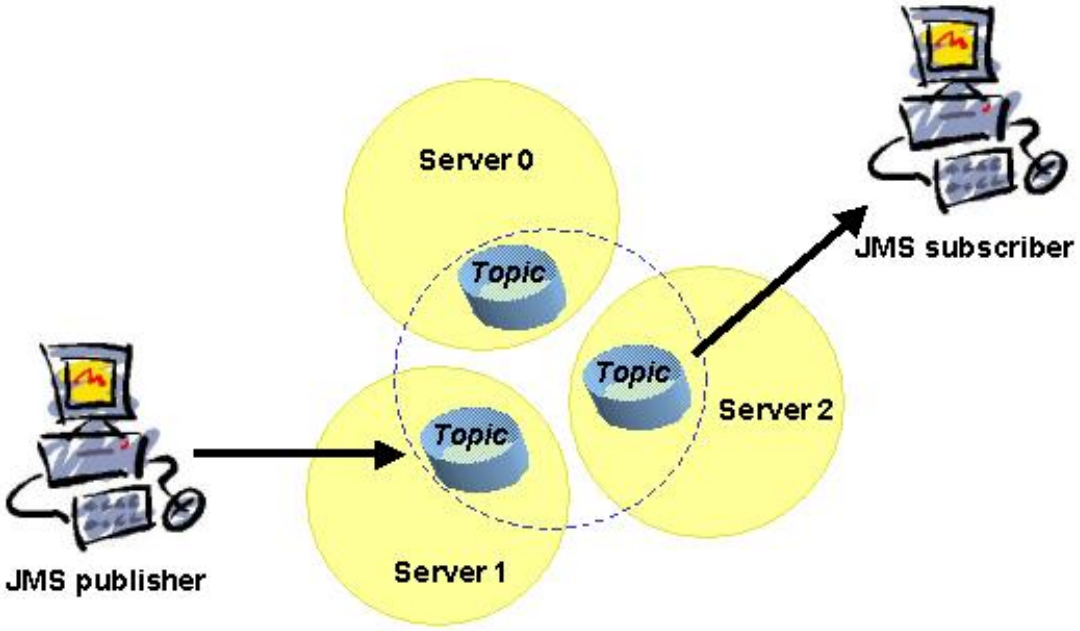

Figure 19 - A clustered topic

Added value

This special feature introduces more flexibility and availability to Publish/Subscribe communication. If server 0 is down, for example, the other representatives of the topic would go on working. The publisher would successfully send its messages to the representative on server 1, and a subscriber to the representative on server 2 would go on getting those messages.

Whereas a regular topic totally depends on the server it is deployed on, a clustered topic still partly works when some (not all) of the servers it is deployed on are down.

Creation and configuration

Creating a clustered topic requires first to create all its representatives. When it is done, each representative must be notified of the cluster it is part of.

Each clustered topic representative must be bound in a name space such as JNDI, so that clients may then retrieve them. Clients do not access the single logical topic, but a given representative of the cluster. Access rights are managed individually, on each topic of the cluster.

3.11.2. Managing a clustered topic

Creating a cluster

Creating a cluster requires first to create all the topics of the cluster. If we consider the example shown on figure 8:

```
Topic topic0 = Topic.create(0);
Topic topic1 = Topic.create(1);
Topic topic2 = Topic.create(2);
```

The cluster needs then to be constructed. Topics are linked two by two with the following method:

- `Topic.addClusteredTopic(Topic addedTopic)`: adds a given topic to a cluster by joining it to a topic already belonging to the cluster, or chosen as the initiator of the cluster.

Going back to our example:

```
topic0.addClusteredTopic(topic1);
topic0.addClusteredTopic(topic2);
```

An `AdminException` is thrown if one of the topics has been deleted server side, or if one of the topics is part of a hierarchy. If the joining topic is already part of a cluster the command is silently ignored.

A `ConnectException` is thrown if the admin connection with the server is closed or lost.

Modifying a cluster

A cluster might be modified either by adding a new topic to it, or by removing a topic from it. The following method is provided:

- `Topic.removeFromCluster()`: notifies a given topic to leave the cluster it is part of.

For example, for removing the representative on server 2 from the cluster:

```
topic2.removeFromCluster();
```

A `ConnectException` is thrown if the admin connection with the server is closed or lost. An `AdminException` is thrown if the topic has been deleted server side, or is not part of any cluster.

This method is similar to removing the topic representative through the `Topic.delete()` method, except that it does not remove the topic. It simply becomes independent.

Using clustered JNDI's object

An object representing the cluster, and which may be bound to a JNDI server, should be instantiated once the cluster is set server side. This object wraps the information allowing a given client application to access the right topic of the cluster according to the server it connects to.

The ClusteredTopic object may be handled either through the administration API or the XML scripting capabilities.

Let assume that there is three existing clustered topics `topic0` (server 0), `topic1` (server 1) and `topic2` (server 2), and the corresponding ConnectionFactory `cf0`, `cf1` and `cf2`. The ClusterConnectionFactory and ClusterTopic objects allow to handle the clustered objects as a whole through a single object; each object is registered with a property specific to its location.

```
ClusterConnectionFactory clusterCF = new ClusterConnectionFactory();
clusterCF.addConnectionFactory("server0", cf0);
clusterCF.addConnectionFactory("server1", cf1);
clusterCF.addConnectionFactory("server2", cf2);
ictx.rebind("clusterCF", clusterCF);

ClusterTopic clusterTopic = new ClusterTopic();
clusterTopic.addDestination("server0", topic0);
clusterTopic.addDestination("server1", topic1);
clusterTopic.addDestination("server2", topic2);
ictx.rebind("clusterTopic", clusterTopic);
```

These objects can be registered in JNDI, then retrieved by a JMS client. When the client creates the JMS connection through the clustered ConnectionFactory, the connection is established depending of the "location" JVM property[6]. Then the client can create the Session and the MessageConsumer; the physical topic used will also depend of the "location" property so the connection and the topic will formed a coherent pair.

```
ConnectionFactory cf = ictx.lookup("clusterCF");
Topic clusterTopic = ictx.lookup("clusterTopic");
```

[6] This property must be fixed according to the client needs; if it is not fixed the location is randomly set for later usage.

```
..
Connection cnx =  cf.createConnection(...);
Session session = cnx.createSession(...);
MessageConsumer consumer = Session.createConsumer(clusterTopic);
```

Setting the access rights

Access rights to the cluster may be set individually, for each topic. They may also be set for the whole cluster, using the same methods. Simply, instead of manipulating `Topic` instances, you have to manipulate the `ClusterTopic` instance.

```
clusterTopic.setFreeReading();
clusterTopic.setFreeWriting();
```

3.11.3. Running the "Clustered Topic" Sample

This sample illustrates the use of Joram's clustered topic. A clustered topic is a group of topics deployed on different servers behaving as a unique "logical" topic. This sample configuration is made of 3 servers, each server hosting a topic part of the cluster. The platform is run in persistent mode. The provided configuration locates all three servers on "localhost" host.

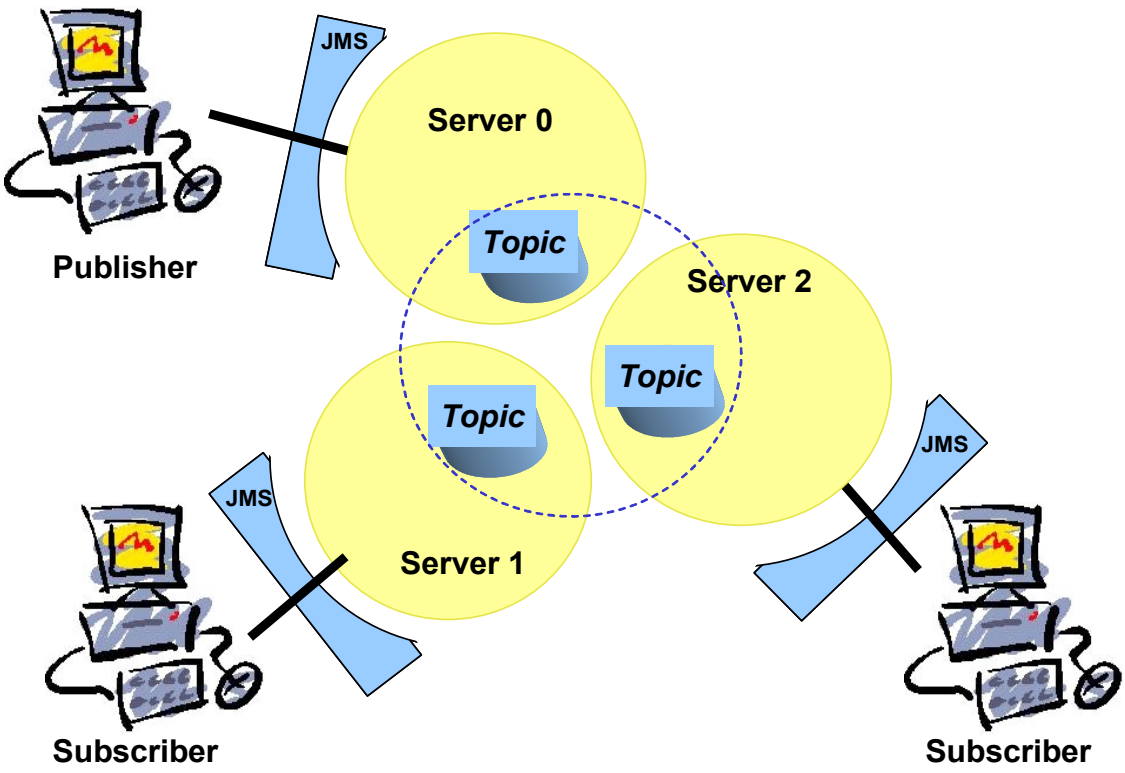

Figure 20 - Cluster sample configuration

This sample code is located in the **`samples/src/joram/cluster/topic`** directory. In order to run the demo described below you must go to the **`samples/src/joram`** directory.

The publisher connects to a server and publishes messages on the local topic of this server. The subscriber connects to a server and subscribes to the local topic of this server. You can either fix dynamically the server used for the connection (giving its identification 0, 1 or 2 when *ant* prompt for it) or use a randomly chosen server (using the '-' key at prompt).

Running the demo

- For compiling the sample code:
  `ant clean compile`
- For starting the configuration:
  `ant reset servers`
- For running the administration code:
  `ant topic_cluster_admin`
  or
  `ant topic_cluster_adminxml`
- For running a subscriber:
  `ant topic_cluster_subscriber`
- For running a publisher
  `ant topic_cluster_publisher`

Scenario:

When you launch a publisher on any server, all subscribers receives the messages sent.

3.11.4. Using XML Scripts

The XML scripting facility allows to create and bind in JNDI clustered ConnectionFactory and Destination.

ClusterConnectionFactory

A ClusterConnectionFactory is defined through a ClusterCF element. It is made up of a set of predefined ConnectionFactory element pointed out by their names. It can be completed by a JNDI declaration.

First we have to define each ConnectionFactory, one for each server of the cluster. The declaration below defines three ConnectionFactory (TcpConnectionFactory by default), cf0 allows the connection to server #0, cf1 to server #1 and cf2 to server #2. Each ConnectionFactory can be bound individually in JNDI.

```
<ConnectionFactory name="cf0">
  <tcp host="localhost" port="16010"/>
  <jndi name="cf0"/>
</ConnectionFactory>

<ConnectionFactory name="cf1">
  <tcp host="localhost" port="16011"/>
  <jndi name="cf1"/>
</ConnectionFactory>

<ConnectionFactory name="cf2">
  <tcp host="localhost" port="16012"/>
  <jndi name="cf2"/>
</ConnectionFactory>
```

Second we can define the ClusterConnectionFactory associating each ConnectionFactory with a location property[7]. The declaration above defines a ClusterConnectionFactory made up of three

[7] This property allows to choose the right association between the ConnectionFactory and the representative of clustered destinations (see paragraph “Using clustered JNDI's object”).

TcpConnectionFactory named cf0, cf1 and cf2. The resulted ConnectionFactory is bound in JNDI with the name clusterCF.

```
<ClusterCF>
  <ClusterElement name="cf0" location="server0"/>
  <ClusterElement name="cf1" location="server1"/>
  <ClusterElement name="cf2" location="server2"/>
  <jndi name="clusterCF"/>
</ClusterCF>
```

ClusterTopic

A clustered topic is made up of a set of Topic elements; each destination needs to be created separately. It can be completed by a JNDI declaration.

```
<Topic name="topic0" serverId="0">
  <jndi name="topic0"/>
</Topic>
<Topic name="topic1" serverId="1">
  <jndi name="topic1"/>
</Topic>
<Topic name="topic2" serverId="2">
  <jndi name="topic2"/>
</Topic>
```

The destinations must be linked in the cluster. The declaration below defines a ClusteredTopic made up of three Topic objects named topic0, topic1 and topic2. The location property allows to associate each Topic object with the corresponding ConnectionFactory of the clusterCF object (see paragraph “Using clustered JNDI's object” above). The resulted queue is bound in JNDI with the name clusterTopic.

```
<ClusterTopic>
  <ClusterElement name="topic0" location="server0"/>
  <ClusterElement name="topic1" location="server1"/>
  <ClusterElement name="topic2" location="server2"/>
  <freeReader/>
  <freeWriter/>
  <jndi name="clusterTopic"/>
</ClusterTopic>
```

3.12. Clustered Queue

3.12.1. Introduction

The clustered queue feature provides a load balancing mechanism. A clustered queue is a cluster of queues (a given number of queue destinations, knowing each other), exchanging messages depending on their load. The figure 9 shows an example of a cluster made of two queues. An heavy producer accesses its local queue (queue 0) and sends messages. The queue is also accessed by a consumer but requesting few messages. It quickly becomes loaded and decides to forward messages to the other queue (queue 1) of its cluster, which is not under heavy load. Thus, the consumer on queue 1 also gets messages, and messages on queue 0 are consumed in a quicker way.

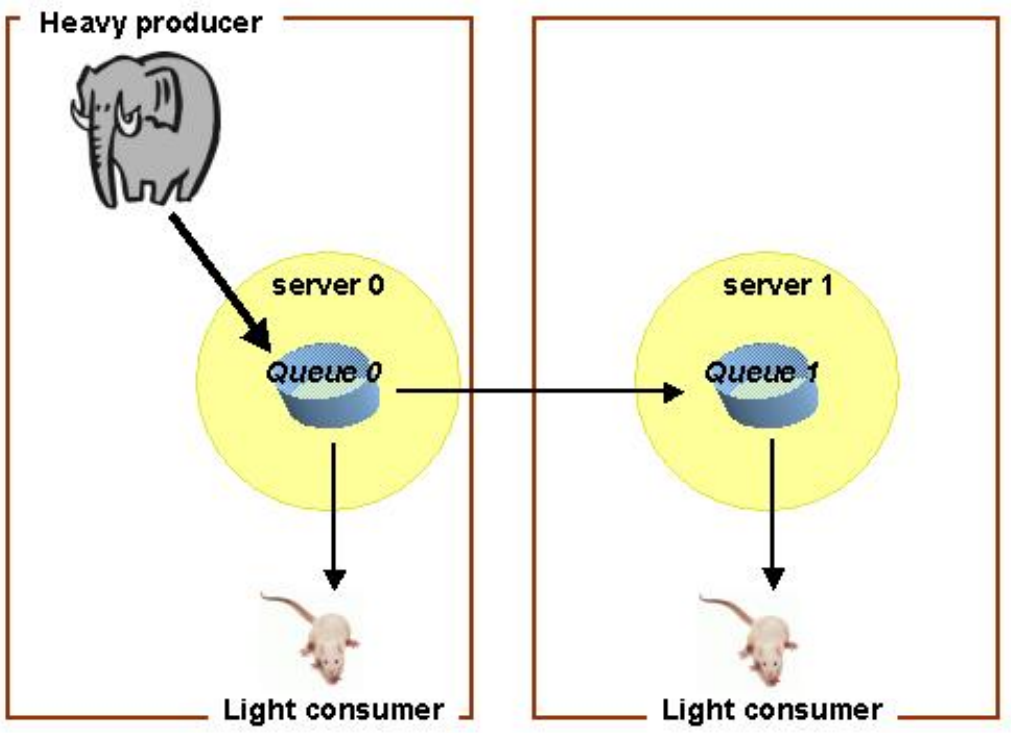

Figure 21 - A cluster of queues balancing heavy deliveries

Basic mechanism

Each queue of a cluster periodically re-evaluates its load factor and sends the result to the other queues of the cluster. When a queue hosts more messages than it is authorized to do, and according to the load factors of the cluster, it distributes the extra messages to the other queues. When a queue is requested to deliver messages but is empty, it requests messages from the other queues of the cluster. This mechanism guarantees that no queue is hyper-active while some others are lazy, and tends to distribute the work load among the servers involved in the cluster.

Creation and configuration

Creating a cluster of queues consists first in setting the cluster's parameters for load-balancing, then in creating the queues one by one, and finally in linking them as part of a same cluster. The needed configuration parameters are:

- a period of time before activating an automatic revaluation of the queues' load factor;
- a number of messages above which a queue is considered as over-loaded;
- a number of pending "receive" requests above which an empty queue requests messages from the other cluster queues;
- a period of time during which a queue which requested something from the cluster queues should not do it again.

Access rights to the cluster may be managed individually, for each queue, or for the whole cluster.

3.12.2. Managing a clustered queue

Setting the clustered queue parameters

Prior to creating the cluster, the following parameters must be set individually for each queue of the cluster[8]:

[8] Each queue can have a different configuration depending of the characteristics of the server, or the number of producer/consumer, etc.

- "period": period in ms between two activations of the load factor evaluation routine for the queue;
- "producThreshold": number of messages above which a queue is considered loaded, a load factor evaluation launched, messages forwarded to other queues of the cluster;
- "consumThreshold": number of pending "receive" requests above which a queue will request messages from the other queues of the cluster;
- "autoEvalThreshold": set to "true" for requesting an automatic revaluation of the queues' thresholds values according to their activity;
- "waitAfterClusterReq": time (in ms) during which a queue which requested something from the cluster is not authorized to do it again.

Properties are set in a `java.util.Properties` instance. For example:

```
java.util.Properties prop = new java.util.Properties();
prop.setProperty("period", "10000");
prop.setProperty("producThreshold", "60");
prop.setProperty("consumThreshold", "2");
prop.setProperty("autoEvalThreshold", "true");
prop.setProperty("waitAfterClusterReq", "500");
```

Creating the clustered queues

Creating a clustered queue consists first in creating the queues that will be part of it. For a cluster of three queues, let's create queue0, queue1 and queue2 and servers 0, 1 and 2 through the `Queue.create` method.

```
String className = "org.objectweb.joram.mom.dest.ClusterQueue";

Queue queue0 = Queue.create(0, "queue0", className, prop);
Queue queue1 = Queue.create(1, "queue1", className, prop);
Queue queue2 = Queue.create(2, "queue2", className, prop);
```

The next step consists in clustering the queues. Queues are linked two by two using Queue class with the following method:

- `addClustered(Queue addedQueue)`: adds a given queue to a cluster by joining it to a queue already belonging to the cluster, or chosen as the initiator of the cluster.

Going back to our example:

```
queue0.addClusteredQueue(queue1);
queue0.addClusteredQueue(queue2);
```

An `IllegalArgumentException` is thrown by this latest method if one of the queues parameters is not a valid Joram queue. An `AdminException` is thrown if one of the queues does not exist server side, or if the joining queue is already part of a cluster. A `ConnectException` is thrown if the connection with the server is lost.

Using clustered JNDI's object

An object representing the cluster, and which may be bound to a JNDI server, should be instantiated once the cluster is set server side. This object wraps the information allowing a given client application to access the right queue of the cluster according to the server it connects to.

The ClusteredQueue object may be handled either through the administration API or the XML scripting capabilities.

Let assume that there is three existing clustered queues `queue0` (server 0), `queue1` (server 1) and `queue2` (server 2), and the corresponding ConnectionFactory `cf0`, `cf1` and `cf2`. The ClusterConnectionFactory and ClusterQueue objects allow to handle the clustered objects as a whole through a single object; each object is registered in the clustered object with a property specific to its location.

```
ClusterConnectionFactory clusterCF = new ClusterConnectionFactory();
clusterCF.addConnectionFactory("server0", cf0);
clusterCF.addConnectionFactory("server1", cf1);
clusterCF.addConnectionFactory("server2", cf2);
ictx.rebind("clusterCF", clusterCF);

ClusterQueue clusterQueue = new ClusterQueue();
clusterQueue.addDestination("server0", queue0);
clusterQueue.addDestination("server1", queue1);
clusterQueue.addDestination("server2", queue2);
```

These objects can be registered in JNDI, then retrieved by a JMS client. When the client creates the JMS connection through the clustered ConnectionFactory, the connection is established depending of the "location" JVM property[9]. Then the client can create the Session and the MessageConsumer; the physical queue used will also depend of the "location" property so the connection and the queue will formed a coherent pair.

```
ConnectionFactory cf = ictx.lookup("clusterCF");
Queue queue = ictx.lookup("clusterQueue");
..
Connection cnx =  cf.createConnection(...);
Session session = cnx.createSession(...);
MessageConsumer consumer = Session.createConsumer(queue);
```

Setting the access rights

Access rights to the cluster may be set individually, for each queue. They may also be set for the whole cluster, using the same methods. Simply, instead of manipulating `Queue` instances, simply manipulate the `ClusterQueue` instance. For example:

```
clusterQueue.setFreeReading();
clusterQueue.setFreeWriting();
```

3.12.3. Running the "Clustered Queue" Sample

This sample illustrates the use of Joram's clustered queues. A cluster queue is a group of queues deployed on different servers and handling the load-balancing.

This sample configuration is made of three servers, each server hosting a queue part of the cluster. The platform is run in persistent mode. The provided configuration locates all three servers on "localhost" host.

This sample code is located in the **`samples/src/joram/cluster/queue`** directory. In order to run the demo described below you must go to the **`samples/src/joram`** directory.

Running the demo:

- Starting the configuration:

  `ant reset servers`

- Running the administration code: *`ant queue_cluster_admin`*.
  - This target creates and configures a cluster with 3 queues as described above.
- Running the consumers using the target *`queue_cluster_consumer`*.
  - This target launches a message consumer connected to one of the server, it indefinitely consumes messages on the corresponding queue.

[9] This property must be fixed according to the client needs; if it is not fixed the location is randomly set for later usage.

- Running the producers using the target *queue_cluster_producer*.
  - This target launches a message producer connected to one of the server, it sends 1000 messages to the corresponding queue.
- Using the *queue_cluster_consumer* or *queue_cluster_producer* targets you can either fix dynamically the server used for the connection (giving its identification 0, 1 or 2 when ant prompt for it) or use a randomly chosen server (directly press the return key at prompt).

In order to view the load-balancing mechanism we can run two different scenarios.

Scenario 1:

In this scenario there is three message consumers, one for each queue of the cluster. Messages are all sent to queue0, the load-balancing mechanism dispatches them between the queues, then the consumers.

- Launching three consumers on queue0, queue1 and queue2:

  "ant queue_cluster_consumer" then type '0'.

  "ant queue_cluster_consumer" then type '1'.

  "ant queue_cluster_consumer" then type '2'.

- Launching multiples producers on queue0:

  "ant queue_cluster_producer" then type '0'.

  "ant queue_cluster_producer" then type '0'.

  "*ant queue_cluster_producer*" then type '0'.

Scenario 2:

In this scenario there is only two message consumers listening on queue0 and queue1. Messages are sent on queue1 and queue2, messages produced to queue2 by the second producer are dispatched between the two consumers by the load-balancing mechanism.

- Launching two consumers on queue0 and queue1

  "ant queue_cluster_consumer" then type '0'.

  "ant queue_cluster_consumer" then type '1'.

- Launching two producers on queue1 and queue2

  "ant queue_cluster_producer" then type '1'.

  "*ant queue_cluster_producer*" then type '2'.

3.12.4. Using XML Scripts

The XML scripting facility allows to create and bind in JNDI clustered ConnectionFactory and Destination.

ClusterConnectionFactory

A ClusterConnectionFactory is defined through a ClusterCF element. It is made up of a set of predefined ConnectionFactory element pointed out by their names. It can be completed by a JNDI declaration.

```
<ConnectionFactory name="cf0">
  <tcp host="localhost" port="16010"/>
  <jndi name="cf0"/>
</ConnectionFactory>

<ConnectionFactory name="cf1">
```

```
    <tcp host="localhost" port="16011"/>
    <jndi name="cf1"/>
  </ConnectionFactory>

  <ConnectionFactory name="cf2">
    <tcp host="localhost" port="16012"/>
    <jndi name="cf2"/>
  </ConnectionFactory>
```

The declaration below defines a ClusterConnectionFactory JNDI's object made up of three TcpConnectionFactory named cf0, cf1 and cf2. Each ConnectionFactory is bound in the cluster with a key depending of its location. The resulted ConnectionFactory is bound in JNDI with the name clusterCF.

```
  <ClusterCF>
    <ClusterElement name="cf0" location="server0"/>
    <ClusterElement name="cf1" location="server1"/>
    <ClusterElement name="cf2" location="server2"/>
    <jndi name="clusterCF"/>
  </ClusterCF>
```

ClusterQueue

A clustered destination is made up of a set of Queue or Topic elements; each destination needs to be created separately then linked. It can be completed by a JNDI declaration.

```
  <Queue name="queue0" serverId="0"
         className="org.objectweb.joram.mom.dest.ClusterQueue">
    <property name="period" value="10000"/>
    <property name="producThreshold" value="50"/>
    <property name="consumThreshold" value="2"/>
    <property name="autoEvalThreshold" value="false"/>
    <property name="waitAfterClusterReq" value="1000"/>
    <jndi name="queue0"/>
  </Queue>

  <Queue name="queue1" serverId="1"
         className="org.objectweb.joram.mom.dest.ClusterQueue">
    <property name="period" value="10000"/>
    <property name="producThreshold" value="50"/>
    <property name="consumThreshold" value="2"/>
    <property name="autoEvalThreshold" value="false"/>
    <property name="waitAfterClusterReq" value="1000"/>
    <jndi name="queue1"/>
  </Queue>

  <Queue name="queue2" serverId="2"
         className="org.objectweb.joram.mom.dest.ClusterQueue">
    <property name="period" value="10000"/>
    <property name="producThreshold" value="50"/>
    <property name="consumThreshold" value="2"/>
    <property name="autoEvalThreshold" value="false"/>
```

```
    <property name="waitAfterClusterReq" value="1000"/>
    <jndi name="queue2"/>
  </Queue>
```

The declaration below defines a ClusteredQueue made up of three Queue objects named queue0, queue1 and queue2. Each queue is bound in the cluster with a key according to its location[10]. The location property allows to associate each Topic object with the corresponding ConnectionFactory of the clusterCF object (see paragraph “Using clustered JNDI's object” above). The resulted queue is bound in JNDI with the name clusterQueue.

```
<ClusterQueue>
    <freeReader/>
    <freeWriter/>
    <ClusterElement name="queue0" location="server0"/>
    <ClusterElement name="queue1" location="server1"/>
    <ClusterElement name="queue2" location="sserver2"/>
    <jndi name="clusterQueue"/>
  </ClusterQueue>
```

3.13. SchedulerQueue

3.13.1. Introduction

A scheduler queue is a standard JMS queue extended with a timer behaviour. When a scheduler queue receives a message with a property called 'scheduleDate' (typed as a long) then the message is not available for delivery before the date specified by the property.

The scheduler queue feature is available since the Joram version 4.3.14.

3.13.2. Using a schedulerQueue

Create a scheduler queue

A scheduler queue is created by using the Joram administration API, calling the Queue creation method `create` from the class `org.objectweb.joram.client.jms.Queue`.

```
queue = Queue.create(0, "schedulerQ", Queue.SCHEDULER_QUEUE, null);
```

Schedule a message

Scheduling a message requires to add a property called "scheduleDate" to the message. The value is the message delivery date typed as a long.

The following example shows how to schedule a message for 5 seconds later.

```
long scheduleDate = System.currentTimeMillis() + 5000L;
TextMessage msg = session.createTextMessage("hello");
msg.setLongProperty("scheduleDate", scheduleDate);
producer.send(msg);
```

A scheduled message is visible in the list of message of the queue but it cannot be delivered before its schedule date is reached (it is close to the use of a selector on all the receive requests verifying that the scheduled date is less than the current one).

[10] This key must be the same that the key used for the corresponding ConnectionFactory.

Cancel a scheduled message

You can cancel a previously scheduled message by removing it from the scheduler queue. This removal operation can be performed through the Joram administration API.

```
queue.deleteMessage(msg.getJMSMessageID());
```

3.14. Acquisition and distribution

3.14.1. Introduction

Joram provides the ability to inject messages in the "JMS world" using non-JMS sources like an email account or a JMX server. That's the **acquisition** part. The acquisition destination will periodically search for new items to acquire and make them available as messages.

On the other hand, custom destinations are also available for **distribution**, which means distributing JMS messages outside the JMS world (e-mail, ftp, …). The distribution destination will forward every message handled to the corresponding recipient.

For example, using an e-mail account you can:

- forward Joram's messages to this external email account using distribution
- import emails from this external account and turn them into Joram's messages using acquisition

Acquisition and distribution are available for queues and topics, so we have respectively:

- Acquisition queue:

```
Queue queue = Queue.create(0, "queue", Queue.ACQUISITION_QUEUE, prop);
```

- Acquisition topic:

```
Topic topic = Topic.create(0, "topic", Topic.ACQUISITION_TOPIC, prop);
```

- Distribution queue:

```
Queue queue = Queue.create(0, "queue", Destination.DISTRIBUTION_QUEUE, prop);
```

- Distribution topic:

```
Topic topic = Topic.create(0, "topic", Destination.DISTRIBUTION_TOPIC, prop);
```

The acquisition and distribution destinations are available since the Joram version 5.3.2.

3.14.2. Configuring an acquisition destination

A set of properties is used to configure the **acquisition** destinations:

- `acquisition.className` – Mandatory property which indicates the acquisition class to use, depending on what content is acquired by the destination.
- `acquisition.period` – Tells the period between two acquisitions, default is 0 (no periodic acquisition)
- `persistent`[11] – Tells if produced messages will be persistent, default is true (JMS default).

[11] The `persistent`, `expiration` and `priority` properties are normally handled by the AMQP and JMS bridge depending of the properties of the incoming message. Therefore you do not normally need to set them when you configure such a bridge.

- `expiration`[11] – Tells the life expectancy of produced messages, default is 0 (JMS default time to live).
- `priority`[11] – Tells the JMS priority of produced messages, default is 4 (JMS default).
- `acquisition.max_msg` – Tells the maximum gap between the message number send by the handler and the message number processed by the acquisition queue, default is 20. Over, stop the acquisition.
- `acquisition.min_msg` – Tells gap the below which restarts the acquisition, default is 10.
- `acquisition.max_pnd` – Tells the maximum number of pending messages on the acquisition queue, defalult is 20.
- `acquisition.min_msg` – Tells the minimum number of pending messages on the acquisition queue, default is 10.

As the acquisition period can be null, the destination exists in two modes which will affect the way this acquisition destination will react to messages sent to it:

In **periodic mode** (period > 0), a message with non-null properties will be treated as a new configuration for the destination, and ignored otherwise.

In **request mode**, a message received will launch an acquisition process with the given message properties or use the last known properties if empty.

Destination mode can be changed using the "period" property.

Alternatively, the acquisition destination can work as an **acquisition daemon** which will be notified at any time a new message is coming. In this case, the previous `acquisition.period` property is inoperative. This acquisition method is useful in cases where the acquisition message is not acquired by polling it.

For example, the JMS acquisition bridge works this way because it sets up a MessageListener on a foreign JMS destination and as a consequence it will be notified whenever a new message is received.

The fact that the acquisition destination is a daemon or not depends on the class specified by the `acquisition.className` property. For now, the acquisition daemon destinations available are the JMS bridge and the AMQP bridge.

The two thresholds limiting the number of pending messages in queue and the number of messages in the engine allow to control the flow and to avoid the engine overflow.

You can start or stop the acquisition destination's with an admin command.

```
AdminCommandReply reply = (AdminCommandReply)
    AdminModule.processAdmin(dest.getName(),
                             AdminCommandConstant.CMD_START_HANDLER, properties);

AdminCommandReply reply = (AdminCommandReply)
    AdminModule.processAdmin(dest.getName(),
                             AdminCommandConstant.CMD_STOP_HANDLER, properties);
```

3.14.3. Configuring a distribution destination

A set of properties is used to configure the **distribution** destinations:

- `distribution.className` – Mandatory property which indicates the distribution class to use, according to the kind of distribution wanted.

As a distribution queue will hold messages that can't be distributed properly (if an exception occurred during the initial distribution), two specific properties are useful when using a distribution queue instead of a distribution topic:

- `period` – Tells the time to wait before another distribution attempt. Default is 0, which means there won't be other attempts.
- `distribution.batch` – If set to true, the queue will try to distribute each time every waiting message, regardless of distribution errors. This can lead to the loss of message ordering, but will prevent a blocking message from blocking every following message. When set to false, the distribution process will stop on the first error. Default is false.

3.14.4. Setting properties

The properties of the destination can be set with the appropriate administration command: setProperties.

```
AdminReply reply = destination.setProperties(prop);
```

You can update the properties of the acquisition handler, by sending JMS message containing the properties. The acquisition queue/topic transmit the properties to the acquisition module and the module set properties to the handler.

3.14.5. Required libraries

As you can see in the previous sections, a class name must be specified as a property to specify the way acquisition or distribution is done. So, for the acquisition or distribution destination to work properly, this class must be accessible at runtime. This means that additional bundles must be installed and started when JORAM is running over OSGi. In order to do this, you can customize the file `config.properties` before launching the Joram server to modify the `felix.auto.start` property or the `felix.auto.deploy.dir` property. Additional details can be found in the config.properties file or on the felix website: http://felix.apache.org/site/index.html.

For example, for the mail destinations to work, you need to install the bundle `joram-mom-extensions-mail.jar`. There are multiple examples of such `config.properties` files in the `samples/config` directory.

3.14.6. Mail acquisition / distribution

Introduction

Using an e-mail account, mail destinations allow you to:

- forward Joram's messages to this external email account using distribution
- import emails from this external account and turn them into Joram's messages using acquisition

The mail acquisition class name is com.scalagent.joram.mom.dest.mail.MailAcquisition and the mail distribution class name is com.scalagent.joram.mom.dest.mail.MailDistribution. Both of these classes are located in the `joram-mom-extensions-mail.jar` bundle.

Mail acquisition properties

In addition to common acquisition properties, the following parameters must be set:

- `popServer` – the DNS name or IP address of the POP server;
- `popUser` – the login name for the email account;
- `popPassword` – the password for the email account;
- `expunge` – allows to remove or not email on the server.

```
Properties prop = new Properties();
prop.setProperty("acquisition.className",
      "com.scalagent.joram.mom.dest.mail.MailAcquisition");
prop.setProperty("acquisition.period", "30000");
prop.setProperty("popServer", popServer);
prop.setProperty("popUser", popUser);
prop.setProperty("popPassword", popPassword);
prop.setProperty("expunge", "false");
```

Remark: The current mechanism does not allow the use of protocol other than POP. The transformation is currently hard-coded it should be interesting to configure it.

Mail distribution properties

In addition to common distribution properties, the following parameters must be set:

- `smtpServer` – the DNS name or IP address of the SMTP server;
- `from` – the email address of the sender;
- `to, cc, bcc` – a comma separated list of recipients;
- `subject` – the subject of outgoing message;
- `selector` – additionally a selector can be added to filter the forwarded messages.

```
Properties prop = new Properties();
prop.setProperty("distribution.className",
      "com.scalagent.joram.mom.dest.mail.MailDistribution");
prop.setProperty("smtpServer", smtpServer);
prop.setProperty("from", from);
prop.setProperty("to", to);
prop.setProperty("subject", "JORAM MAIL");
prop.setProperty("selector", "");
```

Remark: The current mechanism does not allow the use of protocol other than SMTP. It could be interesting to allow the overloading of the default sending parameters by message properties.

Running the sample

This sample illustrates the use of Joram's e-mail acquisition and distribution destinations. It uses a mail topic to send email to a predefined account, and a mail queue to receive email from this identical account.

This sample configuration is made of a unique server located on "localhost" host. The platform is run in non-persistent mode.

Before running the sample, you must change two properties file defining the mail configuration. Theses files are pop.properties and smtp.properties, they are located in the joram/samples/config directory.

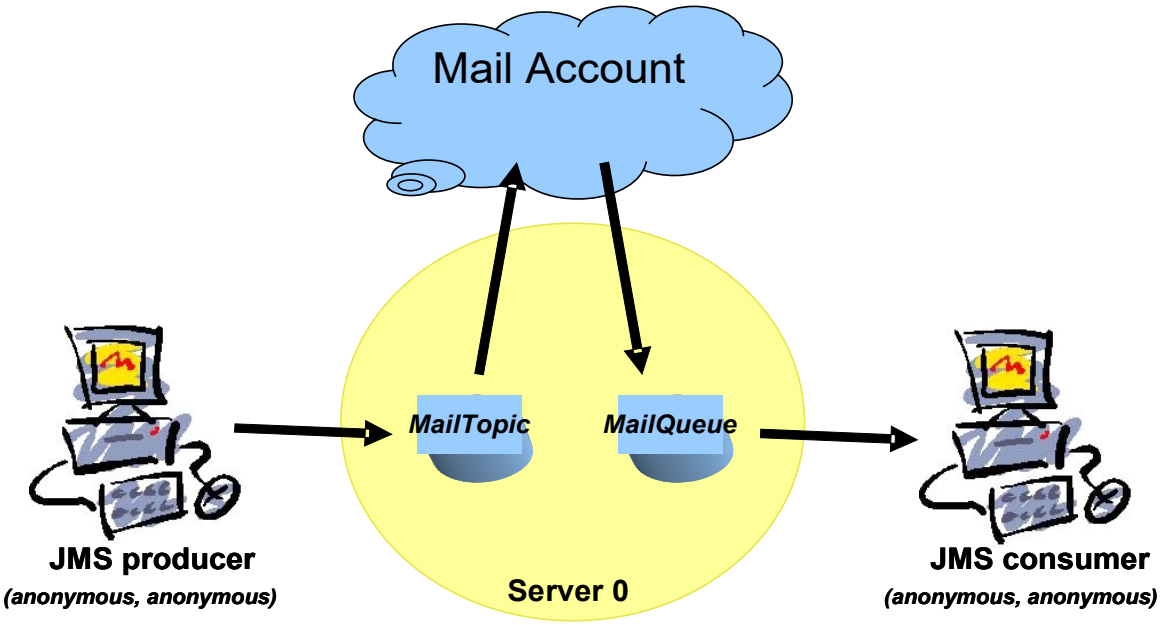

Figure 22 - The mail sample

Running the demo: in the joram/samples/src/joram directory.

- Compiling the samples:

  ```
  ant clean compile
  ```

- Starting the configuration:

  ```
  ant reset extended_server
  ```

- Running the administration code in a new console: this target creates a TcpConnectionFactory, an 'anonymous' user, a mail distribution topic for outgoing mail and a mail acquisition queue for incoming mail.
  ```
  ant mail_admin
  ```
- Running the producer: this target sends 5 messages on the mail topic. Theses messages will be forwarded using the SMTP protocol to the predefined mail account.
  ```
  ant mail_producer
  ```
- Running the consumers: this target launches a message listener on the mail queue. At defined interval the queue will scan the mail account to get new email, then forwards them to the listener.
  ```
  ant mail_consumer
  ```

3.14.7. URL acquisition (collector)

Introduction

URL acquisition queue and topic are special destinations usable to collect a document on a specified URL.

They can be used to periodically :

- import a file from an URL to a Joram's message and store it in this queue.
- import a file from an URL to a Joram's message and forward to each subscribers.

The URL acquisition class name is com.scalagent.joram.mom.dest.collector.URLAcquisition and is located in the `joram-mom-extensions-collector.jar` bundle.

Setting the configuration parameters

In addition to common acquisition parameters (3.14.2), some extra properties are used by the URL acquisition destination:

- `collector.url` – locates the element that will be collected.
- `collector.type` – indicates the type of the generated message. Default is Message.BYTES.

Creating the destination

A monitoring destination is created as any acquisition destination by using the Joram administration API with the previously defined properties.

For example using `org.objectweb.joram.client.jms.Queue` class:

```
Properties prop = new Properties();
prop.setProperty("expiration", "0");
prop.setProperty("persistent", "true");
prop.setProperty("acquisition.period", "10000");
prop.setProperty("acquisition.className",
          "com.scalagent.joram.mom.dest.collector.URLAcquisition");
prop.setProperty("collector.url", url);
prop.setProperty("collector.type", "" + Message.BYTES);

Queue queue = Queue.create(0, "CollectorQueue", Queue.ACQUISITION_QUEUE, prop);
```

This method allows to specify the location server, an internal name, the implementation class and the configuration properties for the new URL acquisition queue/topic.

Running the sample

The sample shows how to collect a file from an URL and store it in the collector queue.

A consumer consumes the message and prints the file.

A producer send message to the collector queue to update some properties.

Running the demo: in the joram/samples/src/joram directory.

- Compile the samples:

  `ant clean compile`

- Start the configuration, this sample needs the joram-mom-extensions-collector.jar bundle so we use the extended_server ant target:

  ant reset extended_server

- Run the administration code: this target creates a ConnectionFactory, an 'anonymous' user, a collector queue and a collector topic. The collector queue is configured to collect the URL only when triggered by a sent message, the initial configuration of the collector topic is to collect the file http://www.gnu.org/licenses/lgpl-2.1.txt every 5 seconds.
  ant collector_admin
- You can alternatively configure this sample using an XML administration script (see the joramAdmin.xml file in the collector directory).
  ant collector_adminxml
- Run the collector: this target connects to the collector queue or topic in order to be notified regularly about the collected file.
  ant consumer_queue **or** *ant consumer_topic*
  Run the trigger: this target sends a message to the collector queue asking the queue to collect the file http://www.gnu.org/licenses/lgpl-3.0.txt.
  ant collector_trigger

3.14.8. JMX acquisition (monitoring)

Introduction

A monitoring acquisition destination is an acquisition destination configured to transform JMX informations into JMS messages.

It works in 2 modes:

- If the `acquisition.period` attribute is set (value greater than 0) the destination periodically scans the selected JMX attributes and generates a message with the value of these attributes.
- In the other case, the user sends to the destination a message with the list of JMX attributes to scan and the destination creates a message with these values, or use the last known JMX attributes.

If the destination is a queue, each message is delivered to a unique consumer, if it is a topic all subscribers receive it.

This topic is based on JMX monitoring so **you must enable JMX monitoring** to use it. See JMX administration of Joram in order to do it.

The JMX acquisition class name is org.objectweb.joram.mom.dest.MonitoringAcquisition and is directly located in the `joram-mom-core.jar` bundle, so you don't need to provide additional bundles for the JMX acquisition destination to work (contrary to other acquisition destinations).

Setting the parameters

Additionally to common acquisition parameters (3.14.2), properties are used to indicate the list of JMX attributes that will be monitored. The property key is the name of the MBean and the value is a comma separated list of attributes to monitor for this MBean. The `'*'` character is allowed to monitor every parameter of the MBean.

- Accessing multiple MBeans is possible using wildcard characters, as defined in the `javax.management.ObjectName` class. See here for JDK6 details.

Example setting monitoring properties:

```
Properties props = new Properties();
props.put("acquisition.className",
          "org.objectweb.joram.mom.dest.MonitoringAcquisition");
props.put("acquisition.period", "5000");
props.put("Joram#0:type=Destination,name=queue",
          "NbMsgsDeliverSinceCreation,NbMsgsReceiveSinceCreation,PendingMessageCou
nt,NbMsgsSentToDMQSinceCreation");
props.put("Joram#0:type=Destination,name=topic",
          "NbMsgsDeliverSinceCreation,NbMsgsReceiveSinceCreation,NbMsgsSentToDMQSi
nceCreation");
```

Example sending a message to the destination:

```
Message msg = sess.createMessage();
msg.setStringProperty("Joram#0:type=Destination,name=*",
          "NbMsgsReceiveSinceCreation,NbMsgsSentToDMQSinceCreation");
```

Creating a monitoring destination

A monitoring destination is created as any acquisition destination by using the Joram administration API with the previously defined properties.

For example using `org.objectweb.joram.client.jms.Queue` class:

```
Queue mQueue = Queue.create(0, "MonitoringQueue",
```

```
                        Queue.ACQUISITION_QUEUE,
                        props);
```

This method allows to specify the location server, an internal name, the implementation class and the configuration properties for the new queue.

Running the sample

The sample shows how to monitor a single queue, it uses the well-known classic sample.

Running the demo: in the joram/samples/src/joram directory.

- Compile the samples:

  ```
  ant clean compile
  ```

- Start the configuration:

  ```
  ant reset single_server
  ```

- Run the administration code of the classic sample. This target creates, configures and registers a ConnectionFactory, an 'anonymous' user, a queue and a topic:
  ```
  ant classic_admin
  ```
- Run the administration code of the monitoring sample. This target creates 2 monitoring destinations, a queue and a topic. The queue is configured to reply to request, the topic is configured to periodically sends JMX information about the queue and topic of the classic sample each 5 seconds.
  ```
  ant monitoring_admin
  ```
- You can alternatively configure this sample using an XML administration script (see the joramAdmin.xml file in the monitoring directory).
  ```
  ant monitoring_adminxml
  ```
- The '*monitoring_monitor*' target subscibes to the monitoring topic in order to be notified regularly about the parameters of the destinations named “queue” and “topic”.
  ```
  ant monitoring_monitor
  ```
- The '' target sends a message to the monitoring queue asking for the parameters of all destinations defined in the server.
  ```
  ant monitoring_diagnose
  ```
- In the other hand you can use the ant targets of the classic sample to produce and consume messages on the 2 destinations (see The classic sample).
  ```
  ant producer_queue
  ant consumer_queue
  ant producer_topic
  ant consumer_topic
  ```

You can launch multiple times the producer and the consumer to see the monitored parameters evolve.

3.14.9. JMS acquisition / distribution bridge

Introduction

The JMS acquisition/distribution bridge allows a JORAM client application to communicate with a JMS destination hosted by a foreign JMS server (let’s call it *XMQ*) in a completely standard way.

The link between JORAM and the *XMQ* heterogeneous platforms is provided by a specifically configured service called JMSConnectionService, and connected to a *XMQ* destination as a standard (JMS 1.1) client application (as illustrated by Figure 23).

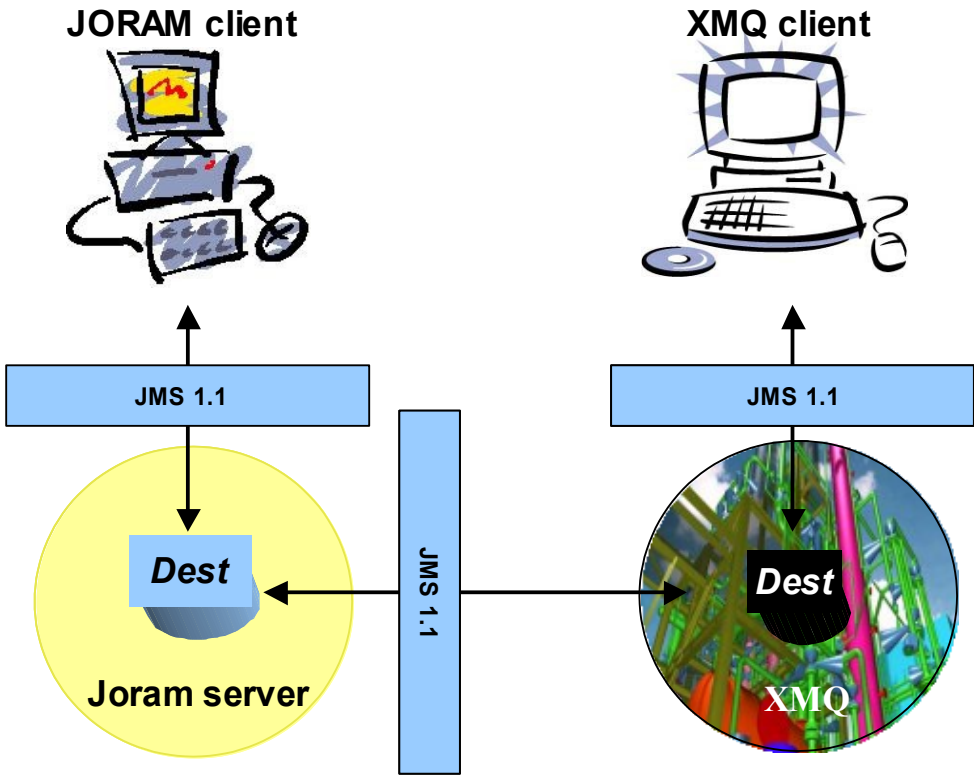

Figure 23 - A JORAM client communicating with a XMQ client

If the JORAM bridge destination is a distribution destination, the JORAM client can send a message to the bridge destination, and the message will be forwarded to the *XMQ* destination.

If the JORAM destination is an acquisition destination, it will automatically subscribe to the XMQ destination, so messages reaching XMQ destination will be forwarded to the bridge destination. Eventually it can be consumed as a regular message on the bridge destination.

From the JORAM client perspective, the target destination is a JORAM destination accessed through the JMS interfaces. It is a fully standard client. The facts that the messages it produces finally reach the *XMQ* destination, and that the messages it consumes originally come from the *XMQ* destination, are totally transparent to the JORAM user.

Acknowledgment policy

Acknowledgements are managed between the JORAM client and the JORAM bridge destination, and between the JORAM bridge destination and the *XMQ* destination, independently.

In fact, the only possible case of message denying between the JORAM bridge destination and the *XMQ* platform occurs when the JMS message delivered by the *XMQ* destination appears not to be readable and thus can not be converted into a JORAM JMS message.

This situation will likely evolve towards the “poison” message scenario, where a JMS client rolls back its session each time it receives the redelivered failing message. In order to avoid this, it is hoped that *XMQ* provides a way to log such messages into a dead message queue. If this feature is available, *XMQ* should be configured to do so.

Once a message delivered by the *XMQ* destination has been successfully converted into a JORAM JMS message, the delivery is acknowledged. *XMQ* does not hold the message any more, it is JORAM which is now responsible for safely distributing it.

Finally, denying a message between the JORAM client and the bridge acquisition destination works the same way as between any JORAM client and JORAM destination. The message is available again for delivery, or logged to a dead message queue if any. In all cases the message stays on the JORAM platform, the *XMQ* destination is not notified of the JORAM client acknowledgements or denials.

Message selection

As for acknowledgements, message selection is handled separately between a JORAM client and the JORAM bridge acquisition destination it interacts with, and between the JORAM bridge destination and the *XMQ* destination. Selectors set-up is done at different times. The selector used to filter the messages on the *XMQ* destination is set at administration time, when configuring the JORAM bridge destination.

The selector used by the JORAM client is set as a standard selector, when creating the `MessageConsumer` instance.

Connection failure handling

From an architectural point of view, the *XMQ* server might be seen as a JORAM server of a JORAM distributed configuration. It might happen that the JMS connection between the JORAM bridge destination and the *XMQ* platform breaks. This case is processed as any connection failure case between two JORAM servers. An automatic reconnection process is launched when the failure is detected (through the setting of a `javax.jms.ExceptionListener` by the JORAM bridge destination). When the JORAM bridge destination finally reconnects, the pending messages or requests are re-routed to the *XMQ* platform.

As a consequence, disconnections between JORAM and *XMQ* are totally transparent to the user, as disconnections between JORAM servers of a distributed JORAM platform.

The reconnection process is as follows:

1. first step: 30 connection trials, one per second;
2. second step: 55 connection trials, one every 5 seconds;
3. last step: infinite connection trials, one every minute.

Cautions with ObjectMessage messages

The bridge must rebuild a new JMS message from the incoming or outgoing message. When using an ObjectMessage the class of the object contained in the message must be accessible otherwise an exception is thrown.

Required libraries

The JMS bridge acquisition class name is org.objectweb.joram.mom.dest.jms.JMSAcquisition and the JMS bridge distribution class name is org.objectweb.joram.mom.dest.jms.JMSDistribution. Both of these classes are located in the `joram-mom-extensions-jmsbridge.jar` bundle.

Additionally, as these classes work as clients of the XMQ server, you need to provide bundles:

- all *XMQ* client jars;
- a `jms.jar` library compatible with the *XMQ* JMS implementation;
- the client jars of the used JNDI server;

This means the following bundles when XMQ is Joram:

- joram-client-jms.jar
- joram-shared.jar (already present in server)
- geronimo-jms_1.1_spec.jar
- jndi-client.jar
- jndi-shared.jar (already present in server)

JMS connection service

The JMS bridge destinations rely on a particular JORAM service which purpose is to maintain a valid connection with the foreign XMQ server (or possibly multiple servers). This service must be declared in the a3servers.xml file, as any other JORAM service (see a3servers.xml configuration file below).

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NTransaction"/>
  <server id="0" name="S0" hostname="localhost">
```

```
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args="root root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
    <service class="org.objectweb.joram.mom.dest.jms.JMSConnectionService"
             args="jndi_url/?name=cnx1&cf=cfName&jndiFactoryClass=com.xxx.yyy&user
=user1&pass=pwd1&clientID=clientID"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer" args="16400"/>
  </server>
</config>
```

The arguments of the `JMSConnectionService` service are used to configure the way we can access the *XMQ* server. The arguments which can be provided are a list of URL separate by a blank space, in order:

- **jndi_url** is the JNDI URL which was used to bound the *XMQ* JMS `ConnectionFactory`.
- **name** represent the connection name, this is used for routing the messages.
- **cf** is the name of the connection factory name of XMQ.
- **jndiFactoryClass** is the factory of the XMQ jndi.
- **user** that should be used by the bridge destination for opening a connection to *XMQ*. If not provided, the connection will be opened with default identification.
- **pass** that should be used by the bridge destination for opening a connection to *XMQ*. If not provided, the connection will be opened with default password.
- **clientID** is provided if *XMQ* requires the setting of such an identifier on its client connection, this is used for the connection API method setClientID.

For example the URL used in the corresponding Joram sample where XMQ is another Joram is:

```
"scn://localhost:16400/?name=cnx&cf=foreignCF&jndiFactoryClass=fr.dyade.aaa.jndi2.
client.NamingContextFactory"
```

JMS bridge properties

In addition to common acquisition parameters (3.14.2), or common distribution parameters (3.14.3), the following properties are required for setting the bridge destination:

- `jms.DestinationName` - JNDI name used to bound the *XMQ* JMS destination.

The following properties are optional:

- `jms.Selector` – selector expression used for filtering messages on the *XMQ* destination (acquisition only).
- `jms.DurableSubscriptionName` – If the *XMQ* destination is a topic, this property sets the name of the durable subscription created. If absent, the subscription will not be durable and messages published when connection with *XMQ* server is failing will be lost (acquisition only).
- `jms.ConnectionUpdatePeriod` – Minimal period in milliseconds between two requests to the JMS connection service. Default value is 5000.

Administration

Bridge destinations can be created either through the administration API or an XML script, examples below comes from the Joram's samples:

- file samples/src/joram/bridge/BridgeAdmin.java for the administration API
- file samples/src/joram/bridge/joramAdmin.xml for the XML scripts.

Using administration API

The code below describes the creation of a JMS distribution queue with a remote queue named "queue".

```
  // Setting the bridge queue properties
  Properties prop = new Properties();
  prop.setProperty("jms.DestinationName", "queue");
  prop.setProperty("distribution.className",
                   "org.objectweb.joram.mom.dest.jms.JMSDistribution");
  // Creating a Queue bridge on server 1:
  Queue bridgeQueue = Queue.create(1, Queue.DISTRIBUTION_QUEUE, prop);
```

The code below describes the creation of a JMS acquisition topic with a remote topic named "topic".

```
  // Setting the bridge topic properties
  prop = new Properties();
  prop.setProperty("jms.DestinationName", "topic");
  prop.setProperty("acquisition.className",
                   "org.objectweb.joram.mom.dest.jms.JMSAcquisition");
  // Creating a Topic bridge on server 1:
  Topic bridgeTopic = Topic.create(1, Topic.ACQUISITION_TOPIC, prop);
```

The code below describes how to add a bridge connection with JMS servers.

```
  AdminModule.connect(....);
  AdminModule.addJMSBridgeConnection(0, "jndi_url/?name=cnx&cf=cfName&jndiFactory
Class=com.xxx.yyy&user=user1&pass=pass1&clientID=clientID");
  AdminModule.disconnect();
```

You can set a list of urls separate by a blank space:

```
  "jndi_url/?name=cnx1&cf=cfName&jndiFactoryClass=com.xxx.yyy&user=user1&pass=pwd1
&clientID=clientID1 jndi_url/?name=cnx2&cf=cfName&jndiFactoryClass=com.xxx.yyy&use
r=user2&pass=pwd2&clientID=clientID2"
```

Joram url sample if XMQ is another Joram:

```
"scn://localhost:16400/?
name=cnx&cf=foreignCF&jndiFactoryClass=fr.dyade.aaa.jndi2.client.NamingContextFact
ory"
```

Using XML scripts

This description below is iso-functional to the code allowing the creation of the JMS distribution queue and JMS acquisition topic (see above).

```
  <Queue name="bridgeQueue" serverId="1"
         className="org.objectweb.joram.mom.dest.DistributionQueue">
    <property name="jms.DestinationName" value="queue"/>
    <property name="distribution.className"
              value="org.objectweb.joram.mom.dest.jms.JMSDistribution"/>
  </Queue>

  <Topic name="bridgeTopic" serverId="1"
         className="org.objectweb.joram.mom.dest.AcquisitionTopic">
    <property name="jms.DestinationName" value="topic"/>
    <property name="acquisition.className"
```

```
                value="org.objectweb.joram.mom.dest.jms.JMSAcquisition"/>
  </Topic>

  <JMSBridgeConnection serverId="0" urls="jndi_url/?name=cnx&cf=cfName&jndiFactor
yClass=com.xxx.yyy&user=user1&pass=pwd1&clientID=clientID"/>
```

Steps

In order to be able to bind the foreign JMS provider administered objects, a naming server is the first thing to start. And in order to be able to successfully deploy a JORAM bridge destination, the foreign JMS administered objects must have been bound. As a consequence, the start-up steps are as follows:

1. start a JNDI server
2. create the foreign JMS administered objects, binding them to the JNDI server
3. configure JORAM by adding a JMSConnectionService in the `a3servers.xml` file
4. start and administer JORAM

Running the sample

A live demo of this sample is available on Joram website documentation: http://joram.ow2.org/doc/tutorials/bridge/bridge.htm.

The demo shows the use of a JMS bridge distribution queue and a JMS bridge acquisition topic (see Figure 24).

Running the demo: in the joram/samples/src/joram directory.

1. Starting a classic JORAM server which will work as the XMQ server. The classic sample (2.2.1) is used to achieve this:
   ant reset single_server
2. Administering the XMQ JMS server, binding its administered objects.
   ant classic_admin
3. Starting the Joram server containing the JMS bridge and the additional JMS client bundles.
   ant bridge_server
4. Administering this Joram bridge server using the administration API:
   ant bridge_admin
   or the XML script:
   ant bridge_adminxml
5. Sending messages to the bridge queue which will forward them to XMQ server:
   ant bridge_producer
6. Consuming the messages on the foreign JMS destination:
   ant consumer_queue

And on the opposite way:

7. Starting a consumer on the bridge topic
   ant bridge_consumer
8. Sending messages to the topic on XMQ server and checking the previously launched consumer acquire them correctly:
   ant producer_topic

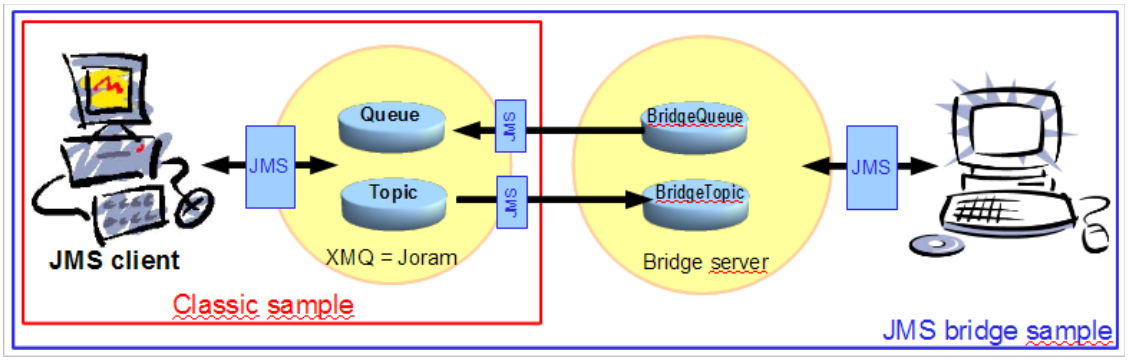

Figure 24 - 2 Joram clients communicating through the JMS bridge

3.14.10. AMQP acquisition / distribution bridge

Introduction

Similarly to the JMS bridge (see previous section), there is acquisition and distribution modules allowing to communicate with an AMQP compliant (v0.9.1) broker.

The AMQP acquisition class name is *org.objectweb.joram.mom.dest.amqp.AmqpAcquisition* and the AMQP distribution class name is *org.objectweb.joram.mom.dest.amqp.AmqpDistribution*. Both of these classes are located in the `joram-mom-extensions-amqp.jar` bundle.

An AMQP queue name must be provided to the JORAM destination to communicate with the AMQP world. This destination must have been previously created on the AMQP server. In the distribution mode the AMQP **default exchange** is used.

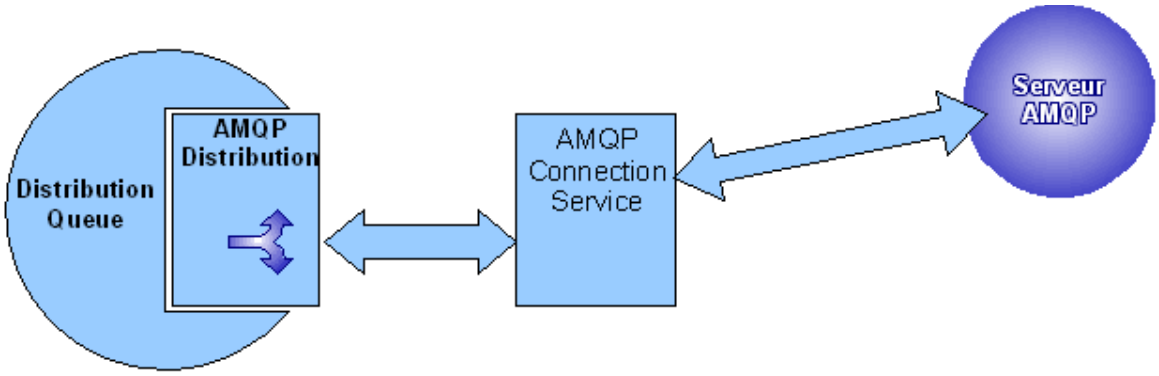

Figure 25: Distribution to an AMQP server using a distribution queue

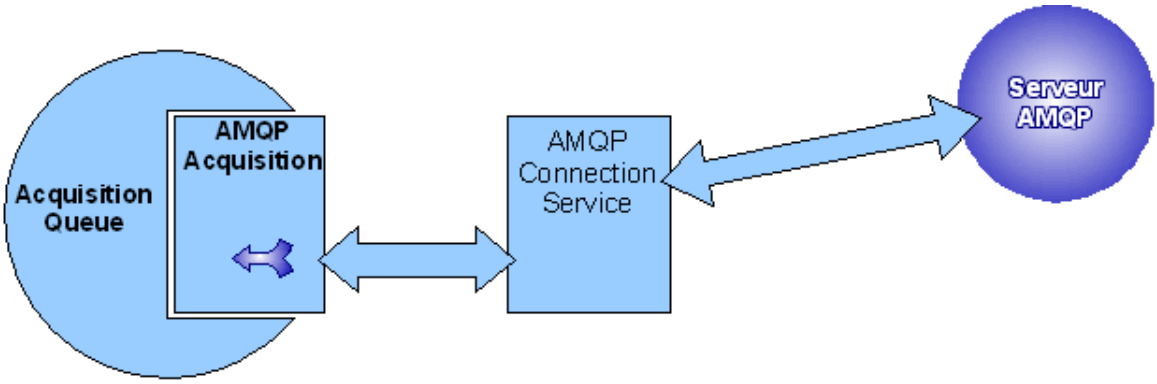

Figure 26: Acquisition from an AMQP server using an acquisition queue

AMQP connection service

The AMQP destinations rely on a particular JORAM service which purpose is to maintain a valid connection with the AMQP server (or possibly multiple servers). This service must be declared in the a3servers.xml file, as any other JORAM service (see a3servers.xml configuration file below).

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NTransaction"/>

  <server id="0" name="S0" hostname="localhost">
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args="root root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
    <service class="org.objectweb.joram.mom.dest.amqp.AmqpConnectionService"
             args="amqp://localhost:5672/?name=server1"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer" args="16400"/>
  </server>
</config>
```

The arguments of the AmqpConnectionService service are used to configure the host and port of the listen socket for the AMQP server. If none is provided, the default host and port are localhost and 5672 (IANA assignment for AMQP).

Administration

Bridge destinations can be created either through the administration API or an XML script.

Using administration API

In addition to common acquisition parameters (3.14.2), or common distribution parameters (3.14.3), the following properties are required to set the AMQP bridge destination:

- `amqp.QueueName` – The name of the AMQP queue with which we want to interact. This queue must have been created previously on the AMQP server.
- `amqp.ConnectionUpdatePeriod` – Minimal period in milliseconds between two requests to the AMQP connection service. Default value is 5000.
- `amqp.Queue.DeclarePassive` – If true declare a queue passively; i.e., check if it exists but do not create it. If false the queue is created if it does not exist. The following parameters are used. Default value is true.
- `amqp.Queue.DeclareExclusive` – If true we are declaring an exclusive queue (restricted to this connection). Default value is false.
- `amqp.Queue.DeclareDurable` – If true we are declaring a durable queue (the queue will survive a server restart). Default value is true.

- `amqp.Queue.DeclareAutoDelete` – If true we are declaring an "autodelete" queue (server will delete it when no longer in use). Default value is false.

Example creating an AMQP acquisition queue:

```
Properties prop = new Properties();
prop.setProperty("acquisition.className",
                 "org.objectweb.joram.mom.dest.amqp.AmqpAcquisition");
prop.setProperty("amqp.QueueName", "amqpQueue");
prop.setProperty("amqp.ConnectionUpdatePeriod", "1000");
prop.setProperty("amqp.Queue.DeclarePassive", "true");
queue = Queue.create(0, "queue", Queue.ACQUISITION_QUEUE, prop);
```

The code below describes how to add a bridge connection with AMQP servers.

```
AdminModule.connect(....);
AdminModule.addAMQPBridgeConnection(0, "amqp://localhost:5672/?name=serv1");
AdminModule.disconnect();
```

You can set a list of amqp url separate by a space:

"amqp://user:pass@host1:5672/?name=serv1 amqp://user:pass@host2:5678/?name=serv2"

Using XML scripts

This description below is iso-functional to the code allowing the creation of the JMS distribution queue and JMS acquisition topic (see above).

```
<Queue name="bridgeQueue" serverId="0"
       className="org.objectweb.joram.mom.dest.DistributionQueue">
  <property name="amqp.QueueName" value="amqpQueue"/>
  <property name="distribution.className"
          value="org.objectweb.joram.mom.dest.amqp.AmqpDistribution"/>
  <freeWriter/>
  <jndi name="bridgeQueue"/>
</Queue>

<Topic name="bridgeTopic" serverId="0"
       className="org.objectweb.joram.mom.dest.AcquisitionTopic">
  <property name="amqp.QueueName" value="amqpQueue"/>
  <property name="acquisition.className"
          value="org.objectweb.joram.mom.dest.amqp.AmqpAcquisition"/>
    <freeReader/>
    <freeWriter/>
    <jndi name="bridgeTopic"/>
</Topic>

<AMQPBridgeConnection serverId="0"
          urls="amqp://user:pass@host1:5672/?name=serv1 amqp://host2:5672/?
name=serv2"/>
```

3.14.11. AMQP acquisition / distribution proxy

Introduction

The AMQP proxy allows a JORAM client application to communicate with an AMQP client through acquisition and distribution modules.

The ProxyAMQP acquisition class name is *com.scalagent.jorammq.mom.dest.proxyamqp.ProxyAmqpAcquisition* and the ProxyAMQP distribution class name is *com.scalagent.jorammq.mom.dest.proxyamqp.ProxyAmqpDistribution*. Both of these classes are located in the `jorammq-mom-extensions-proxyamqp.jar` bundle.

It is not compulsory to indicate destination properties descibes in the following (Setting the parameters). If none is set, the default fanout exchange is used to transmit messages both for acquisition module and distribution module.

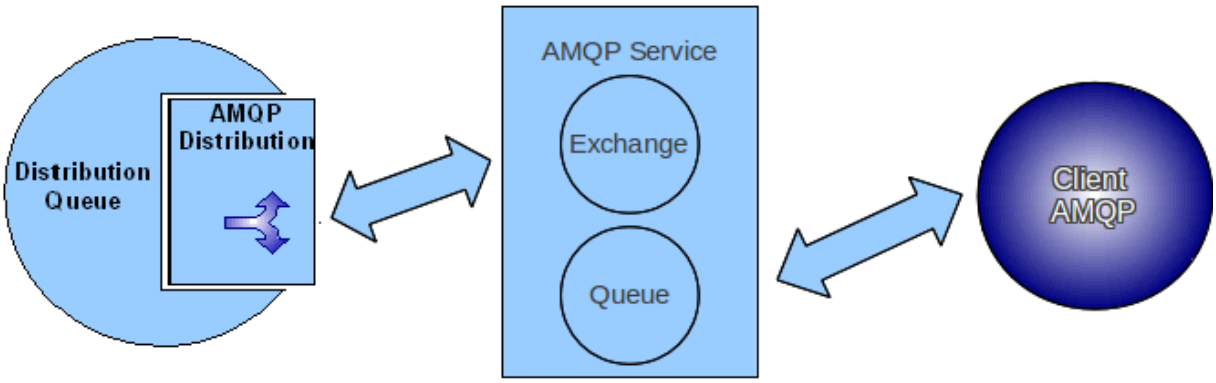

Figure 27: Distribution to an AMQP client using a distribution queue

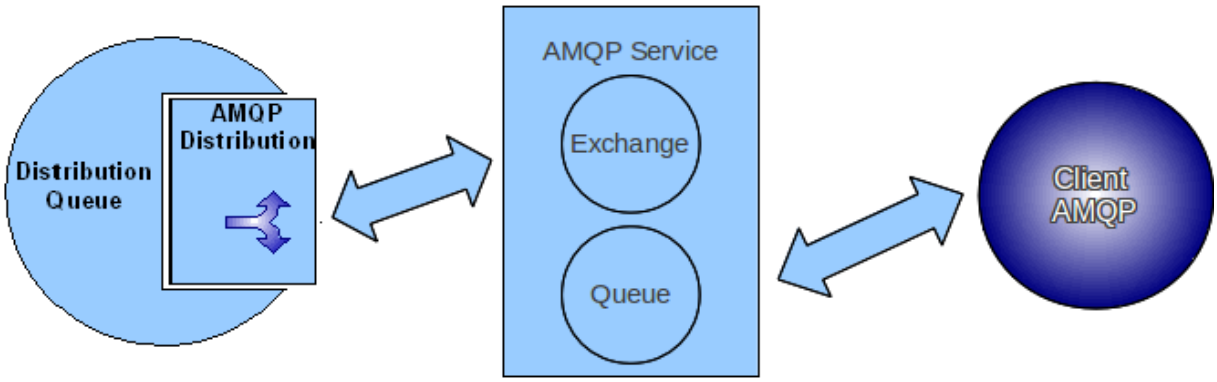

Figure 28: Distribution to an AMQP client using a distribution queue

AMQPService

The ProxyAMQP destinations rely on a particular JORAM service which purpose is to maintain a valid connection with the AMQP destinations. This service must be declared in the a3servers.xml file, as any other JORAM service (see a3server.xml configuration below).

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NTransaction" />
```

```
  <server id="0" name="s0" hostname="localhost">
    <service class="fr.dyade.aaa.agent.AdminProxy" args="7890"/>
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
                  args="root root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
                  args="2560"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer" args="16400"/>
    <service class="org.ow2.joram.mom.amqp.AMQPService" args="5672" />
  </server>
</config>
```

The argument of the AmqpService service is used to configure the port of the listen socket for the AMQP server. If none is provided, the default port is 5672 (IANA assignment for AMQP).

Setting the parameters

In addition to common acquisition parameters (3.14.2), the following properties could be used for configuring the acquisition destination: (all are optional)

- `ProxyAmqp.Acquisition.DestinationName` – the name of the amqp destination which we want to connect to. Default value is “amq.fanout”.
- `ProxyAmqp.Acquisition.ConnectionUpdatePeriod` – the minimal period in milliseconds between two requests to the AMQP service. Default value is 5000.
- `ProxyAmqp.Acquisition.Routing` – indicate the routing key used to bind the queue to direct exchanges and topic exchanges. Default value is “”.
- `ProxyAmqp.Acquisition.Headers` – indicate the headers values used to bind the queue to headers exchanges. Defalut value is null.
- `ProxyAmqp.Queue.DeclareExclu`sive – indicate if the acquisition queue is exclusive (restricted to this connection). Default value is false.
- `ProxyAmqp.Queue.DeclareDurable` – indicate if the acquisition queue is durable (the queue will survive a server restart). Default value is true.
- `ProxyAmqp.Queue.DeclareAutodelete` – indicate if the acquisition queue is auto-delete (server will delete it when no longer in use). Default value is false.

Example creating a ProxyAMQP acquisition queue:

```
Properties prop = new Properties();
prop.setProperty(ProxyAmqpAcquisition.DESTINATION_NAME_PROP, "amq.direct");
prop.setProperty(ProxyAmqpAcquisition.ROUTING_PROP, "routing.jms");
prop.setProperty("acquisition.className",
"com.scalagent.jorammq.mom.dest.proxyamqp.ProxyAmqpAcquisition");


Queue queue = Queue.create(0, Queue.ACQUISITION_QUEUE, prop);
```

Similarly to the acquisition destination, there are optional parameters which can be set in addition to common distribution parameters (3.14.3) for configuring the distribution destination.

- `ProxyAmqp.Distribution.DestinationName` – the name of the amqp destination which we want to conect to. Default value is “amq.fanout”.
- `ProxyAmqp.Distribution.DestinationType` – the type of the amqp exchange which we want to connect to. Default valut is “fanout”. Possible values are “direct”, “fanout”, “topic” and “headers”

- `ProxyAmqp.Distribution.ConnectionUpdatePeriod` – the minimal period in milliseconds between two requests to the AMQP service. Default value is 5000.
- `ProxyAmqp.Distribution.Routing` – indicate the routing key used to send messages to direct exchanges and topic exchanges. Default value is null.
- `ProxyAmqp.Distribution.Headers` – indicate the headers values used to send messages to headers exchanges. Defalut value is null.
- `ProxyAmqp.Exchange.DeclareDurable` – indicate if the exchange is durable (the queue will survive a server restart). Default value is true.
- `ProxyAmqp.Message.Immediate` – indicate if the message must be delivered immediately (all routed queue must have active consumer). Default value is false.
- `ProxyAmqp.Message.Mandatory` – indicate if the message must be routed (at least one binding queue match); default value is true.

Example creating a ProxyAMQP distribution queue:

```
Properties prop = new Properties();
prop.setProperty(ProxyAmqpDistribution.DESTINATION_NAME_PROP, "Exchange");
prop.setProperty(ProxyAmqpDistribution.DESTINATION_TYPE_PROP, "direct");
prop.setProperty(ProxyAmqpDistribution.ROUTING_PROP, "routing.amqp");
prop.setProperty("distribution.className",
"com.scalagent.jorammq.mom.dest.proxyamqp.ProxyAmqpDistribution");

Queue exchange = Queue.create(0, Queue.DISTRIBUTION_QUEUE, prop);
```

Running the sample

The sample show how to use ProxyAMQP acquisition and distribution queues.

Running the demo: in the jorammq-samples/src/main/java

- Compile the samples:
  `ant clean compile`
- Start the configuration:
  `ant amqp_server`
- Run the administration code. This target creates, configures and registers a ConnectionFactory, an 'anonymous' user, an acquisition queue and a distribution queue:
  `ant proxyamqp_admin`
- Server is ready, use following target to send and receive messages:
  - for sending messages to distribution queue:
    `ant proxyamqp_jms_producer`
  - for receiving messages from distribution queue:
    `ant proxyamqp_amqp_receiver`
  - for sending messages to acquisition queue:
    `ant proxyamqp_amqp_producer`
  - for receiving messages from acquisition queue:
    `ant proxyamqp_jms_receiver`

3.14.12. Acquisition / distribution with PHP scripts

Introduction

It is possible to exchange messages with Joram application using PHP scripts. Two messaging protocols are available:

- AMQP: using the AMQP proxy
- STOMP: using the JASP tool (Joram Access to STOMP Protocol)

AMQP protocol

Required libraries

To send amqp messages with PHP script, you must install the pecl extension to AMQP, available to download at this adress :http://pecl.php.net/package/amqp. If not already yet, you also need to download rabbitmq-c client library. You can find further information about installing these libraries on the PHP manual: http://www.php.net/manual/en/amqp.installation.php.

At the end of installation, check in the configuration file *php.ini* if AMQP is in the extension list: `extension = amqp.so`

Libraries needed by the proxy AMQP are same as indicated previously (see 3.14.5) more the proxyamqp bundle.

Creating the destination

Destinations used are those of proxyamqp, no additional parameters are neededd. AMQP queues and exchanges can be created by Joram application or PHP scripts.

Running the sample

The sample show how to send messages with PHP and receive them with JMS ans vice versa.

Running the demo: in jorammq-samples/src/main/java

- Compile the samples:
  `ant clean compile`
- Start the configuration:
  `ant amqp_server`
- Run the administration code. This target creates, configures and registers a ConnectionFactory, an 'anonymous' user, an acquisition queue and a distribution queue:

  `ant proxyamqp_admin`
- Server is ready, use following target to send and receive messages:
  - for sending messages from PHP:
    `ant php.amqp_sender`
  - for receiving messages from PHP:
    `ant proxyamqp_jms_receiver`
  - for sending messages froml Joram:
    `ant proxyamqp_jms_producer`
  - for receiving messages from Joram:
    ant php.amqp_receiver

STOMP protocol

Required libraries

To send STOMP messages with PHP scripts, you must install the pecl extension to STOMP available to download at this adress: http://pecl.php.net/package/stomp. At the end of installation, chack in the configuration fil php.ini if STOMP is in the extension list: `extension = stomp.so`

To send and receive STOMP messages with a Joram application, you need the JASP bundle.

Creating the destination

Destinations used to send/receive STOMP messages are default Joram queues. To exchange messages with stomp protocol, a socket on port 61613 is used and the destination is indicated by name or AgentId ("queue" or "'#0.0.1026" in the sample).

Running the sample

Running the demo: in joram/samples/src/joram

- Compile the samples:
  `ant clean compile`
- Start the configuration:
  `ant stomp_server`
- Run the administration code. This target greates, configures and registers a TcpConnectionFactory, an 'anonymous' user, a queue and a topic:
  `ant classic_admin`
- Server is ready, use following target to send and receive messages:
  - for sending messages from PHP:
    `ant php.stomp_sender`
  - for receiving messages from PHP:
    `ant jms.stomp_receiver`
  - for sending messages froml Joram:
    `ant jms.stomp_sender`
  - for receiving messages from Joram:
    `ant php.stomp_receiver`

3.15. FTPQueue

3.15.1. Introduction

Ftp queue is special destinations usable to transfer file by FTP. It wraps the FTP transfer of a file through a message exchange. The sender starts the transfer by sending a JMS message, the receiver is notified of the transfer completion by a JMS message.

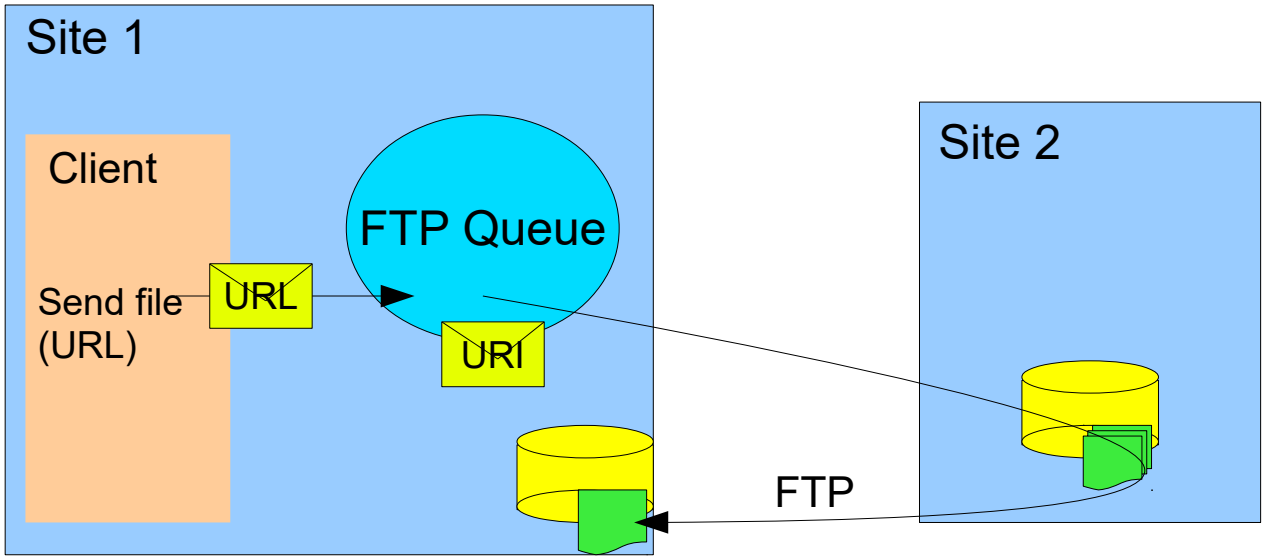

Figure 29: File transfert through a FTP Queue

The sender send the URL information file to the destination FTPQueue. This destination gets the file, stores it in the local directory and generates a message who contain the URI localisation of the transferred file (Figure 29 above).

The Consumer receive a message who contain the URI localisation of the file (Figure 30 below).

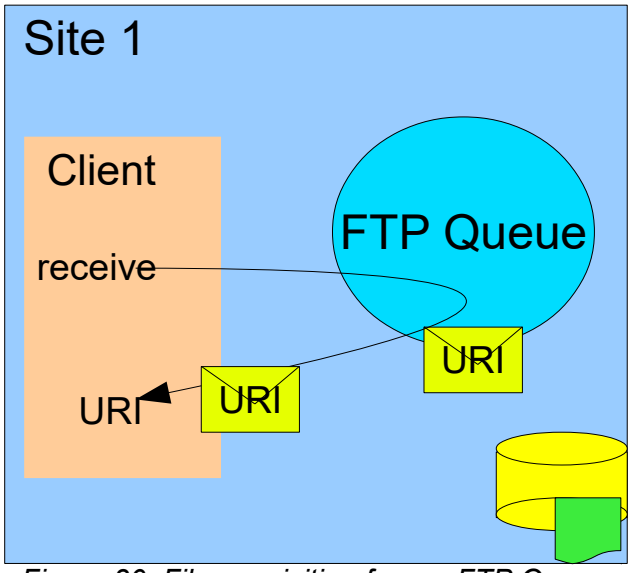

Figure 30: File acquisition from a FTP Queue

3.15.2. Managing a FTPQueue

Setting the FTPQueue parameters

The options are set using properties at FTPQueue creation time.

- user: the user name for the FTP.
- pass: the user password for FTP.
- path: the local directory to store the file transferred. Default is the running directory (path=null).
- ftpImplName: The implementation of the FTP transfer, we provide two implementations:
- the default one based on JDK URL:
  `com.scalagent.joram.mom.dest.ftp.TransferImplRef`

- the second is based on JFTP:
  `com.scalagent.joram.mom.dest.ftp.TransferImplJftp`

The user and pass options are optional, because you can set this information in the sender URL request, like this:

```
msg.setStringProperty("url", "ftp://user:pass@host/file;type=i");
```

Creating the FTPQueue

A FTPQueue is created by using the Joram administration API, calling the Queue creation method `create` from the class `org.objectweb.joram.client.jms.Queue`.

```
prop = new Properties();
prop.setProperty("user", "my_FTP_name"); // optional
prop.setProperty("pass", "my_FTP_pass"); // optional
prop.setProperty("path", "the_local_path"); // optional
Queue queue = Queue.create(0, "ftpQueue",
"com.scalagent.joram.mom.dest.ftp.FtpQueue", prop);
```

This method allows to specify the location server, an internal name, the implementation class and the configuration properties for the new FTPQueue.

3.15.3. Using a FTPQueue destination

Sending a message

Specify the fileName and user / password if the administrator don't set.

```
TextMessage msg = sess.createTextMessage();
msg.setText("transfer " + fileName);
if (user != null && pass != null)
  msg.setStringProperty("url", "ftp://user:pass@host/fileName;type=i");
else
  msg.setStringProperty("url", "ftp://host/fileName;type=i");
msg.setLongProperty("crc", new File(fileName).length());
msg.setBooleanProperty("ack", false);
```

When the FTPQueue destination receive the message, it run a task to get the file specify by the URL and store this file in a local directory. The sending message is drop and a new message containing the URI file location is generate and store in the FTPQueue.

3.15.4. Running the sample

The sample shows how to get the “welcome.msg” file from ftp.kernel.org.

Running the demo: in the joram/samples/src/joram directory.

- Compile the samples:

  `ant clean compile`

- Start the configuration and run the server, this sample needs the joram-mom-extensions-ftp.jar bundle so we use the extended_server ant target:

  `ant reset extended_server`

- Run the administration code: this target creates a ConnectionFactory, an 'anonymous' user and a FTPQueue with user=anonymous and pass=anonymous.
  `ant ftp_admin`

- Run the producer: this target send a message to the FTPQueue with this URL “ftp://ftp.kernel.org/welcome.msg”.
  ```
  ant ftp_producer
  ```
- Run the receiver: this target connects to the FTPQueue and consume the message and print the file URL, crc and ack..
  ```
  ant ftp_receiver
  ```

4. Using a collocated server

4.1. Introduction

A collocated Joram server is a standard Joram server running inside the same process (JVM) as one or more Joram clients. If your Java application needs to start such an embedded server you must configure this server, start it inside your application and connect your JMS clients to it.

A collocated Joram server can be part of a distributed configuration of multiples collocated or not servers, it can eventually be reach by other client through the TCP protocol.

4.2. Configure a collocated server

A collocated server is configured exactly like a non-collocated server, i.e. you don't need to declare any extra services to use a collocated server.

A typical configuration would be:

```
<?xml version="1.0"?>
<config>
  <property name=”Transaction” value=”fr.dyade.aaa.util.NullTransaction”/>
  <server id="0" name="S0" hostname="localhost">
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args=”root root”/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
  </server>
</config>
```

Notice that:

- in the above configuration, the collocated server can also be accessed from remote clients through the TCP protocol. If you don't need the TCP access point you can remove the service `TcpProxyService`.
- you can include a collocated server inside a distributed Joram platform: a collocated server is a server just like any other.

4.3. Start a collocated server

A collocated server must be pro grammatically started inside the same process as your Java client application.

The following code starts the server #0:

```
fr.dyade.aaa.agent.AgentServer.init((short) 0, "./s0", null);
fr.dyade.aaa.agent.AgentServer.start();
```

The method `init` initializes the server with three parameters:

1. its identifier: `0`
2. the directory where its persistent state is stored: `./s0`
3. the monolog logger factory: leave it to `null` if you want the server to configure it itself.

The method `start` actually starts the server.

You can also initialize and start a server by calling the method `AgentServer.main` which aggregates the initialization and the start into a single operation:

```
String args[] = {"0", "./s0"};
fr.dyade.aaa.agent.AgentServer.main(args);
```

4.4. Connect to the collocated server

4.4.1. Create local connections

The class `LocalConnectionFactory` enables you to create local connections to the collocated server:

```
import org.objectweb.joram.client.jms.local.*;

ConnectionFactory cnxFact = LocalConnectionFactory.create();
```

In the same package you can find several factories that you can use to create more specific connections: `<XA><Topic|Queue>LocalConnectionFactory`.

4.4.2. Connect the administration module

The class `AdminModule` provides a method `collocatedConnect` that must be called before doing administration operations through the collocated server.

```
import org.objectweb.joram.client.jms.admin.*;

AdminModule.collocatedConnect("root", "root");
```

4.5. Stop the collocated server

If you need to stop the collocated server without stopping the entire embedding Java application you must call the method `stop` provided by the class `AgentServer`:

```
fr.dyade.aaa.agent.AgentServer.stop();
```

You can then restart the server with the following code:

```
fr.dyade.aaa.agent.AgentServer.start();
```

4.6. Start the embedding Java application

You must ensure that the classpath contains:

- the jar files expected by a Joram server: joram-mom-core.jar, Jcup.jar, etc.
- the directory where the a3servers.xml configuration file is located.

Working with sources distribution

4.7. Getting Joram sources

4.7.1. Getting a packaged version of Joram

The packages are downloadable from the following location:

- http://forge.ow2.org/project/showfiles.php?group_id=4.

For release **x.y.z**, the following tar file is provided:

- **`joram-release-x.y.z-src.zip`**, including the client and server sources.

A package is expanded by UNIX users with the ***`gunzip`*** and ***`tar`*** commands; Windows developers can use the ***Winzip*** utility.

4.7.2. Getting Joram from SVN

JORAM SVN page is located at: http://forge.ow2.org/plugins/scmsvn/index.php?group_id=4. The module to extract is **joram**. A snapshot is regularly generated by bamboo continuous integration tool and can be downloaded at:

http://bamboo.ow2.org/artifact/JORAM-ANT/JOB1/build-latest/Releases/release/target

4.7.3. Directory structure and description

Joram sources distribution

The distribution is expanded in a `joram-x.y.z/` directory. It includes the following directories:

- `a3/`
- `assembly/`
- `conf/`
- `jndi/`
  - `client/`
  - `server/`
  - `shared/`
- `joram/`
  - `client/`
  - `jca/`
  - `mom/`
  - `security/`
  - `shared/`
  - `tools/`

- licenses/
- samples/
  - bin/…
  - config/
  - src/
    - joram/…

a3/ directory

Contains the sources of the agent platform, foundation of Joram server.

conf/ directory

Contains the configuration used to launch Joram on the OSGi platform Felix.

jndi/ directory

Contains the sources of Joram's JNDI (server, client and shared classes).

joram/ directory

Contains the sources of Joram server (mom), client and shared classes. You can also find here the JCA connector, jaas security and a tool to access Joram with the STOMP protocol.

licenses/ directory

Contains the LGPL header displayed on top of each source file, as well as the licences of the external softwares provided in the distribution.

samples/ directory

Contains the Joram samples sources, configuration files, UNIX and Windows scripts for launching JORAM servers and clients (how to use them is explained in chapter 2Using samples).

4.8. Compiling and shipping Joram

JORAM distribution is ready for compiling with Apache ***Maven*** tool. Maven can be downloaded from http://maven.apache.org/. Documentation is available at the same location.

4.8.1. Compiling Joram

The best way to build Joram is:

- Open a console and go to Joram main directory
- *mvn clean install (see result below)*

```
$ mvn clean install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] JORAM
[INFO] JORAM :: a3
[INFO] JORAM :: a3 :: common
```

```
[INFO] JORAM :: a3 :: rt
[INFO] JORAM :: a3 :: osgi
[INFO] JORAM :: jndi
[INFO] JORAM :: jndi :: shared
[INFO] JORAM :: jndi :: server
[INFO] JORAM :: jndi :: client
[INFO] JORAM :: joram
[INFO] JORAM :: joram :: shared
[INFO] JORAM :: joram :: mom
[INFO] JORAM :: joram :: mom :: core
[INFO] JORAM :: joram :: client
[INFO] JORAM :: joram :: client :: jms
[INFO] JORAM :: joram :: client :: JCA 1.5 connector
[INFO] JORAM :: joram :: mom :: extensions
[INFO] JORAM :: joram :: mom :: extensions :: collector
[INFO] JORAM :: joram :: mom :: extensions :: jmsbridge
[INFO] JORAM :: joram :: mom :: extensions :: mail
[INFO] JORAM :: joram :: mom :: extensions :: scheduler
[INFO] JORAM :: joram :: mom :: extensions :: ftp
[INFO] JORAM :: joram :: security
[INFO] JORAM :: joram :: tools
[INFO] JORAM :: joram :: tools :: jasp
[INFO] JORAM :: licenses
[INFO] JORAM :: conf
[INFO] JORAM :: assembly
[INFO] JORAM :: joram :: samples
[INFO]

…

…

[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] JORAM ..................................... SUCCESS [0.141s]
[INFO] JORAM :: a3 ............................... SUCCESS [0.015s]
[INFO] JORAM :: a3 :: common ..................... SUCCESS [1.797s]
[INFO] JORAM :: a3 :: rt ......................... SUCCESS [2.672s]
[INFO] JORAM :: a3 :: osgi ....................... SUCCESS [0.797s]
[INFO] JORAM :: jndi ............................. SUCCESS [0.016s]
[INFO] JORAM :: jndi :: shared ................... SUCCESS [0.765s]
[INFO] JORAM :: jndi :: server ................... SUCCESS [1.422s]
[INFO] JORAM :: jndi :: client ................... SUCCESS [0.860s]
[INFO] JORAM :: joram ............................ SUCCESS [0.000s]
[INFO] JORAM :: joram :: shared .................. SUCCESS [1.859s]
[INFO] JORAM :: joram :: mom ..................... SUCCESS [0.000s]
[INFO] JORAM :: joram :: mom :: core ............. SUCCESS [3.016s]
[INFO] JORAM :: joram :: client .................. SUCCESS [0.000s]
[INFO] JORAM :: joram :: client :: jms ........... SUCCESS [3.031s]
```

```
[INFO] JORAM :: joram :: client :: JCA 1.5 connector ..... SUCCESS [1.313s]
[INFO] JORAM :: joram :: mom :: extensions ............... SUCCESS [0.015s]
[INFO] JORAM :: joram :: mom :: extensions :: collector .. SUCCESS [0.688s]
[INFO] JORAM :: joram :: mom :: extensions :: jmsbridge .. SUCCESS [1.172s]
[INFO] JORAM :: joram :: mom :: extensions :: mail ....... SUCCESS [0.906s]
[INFO] JORAM :: joram :: mom :: extensions :: scheduler .. SUCCESS [0.984s]
[INFO] JORAM :: joram :: mom :: extensions :: ftp ........ SUCCESS [1.422s]
[INFO] JORAM :: joram :: security ........................ SUCCESS [0.016s]
[INFO] JORAM :: joram :: tools ........................... SUCCESS [0.015s]
[INFO] JORAM :: joram :: tools :: jasp ................... SUCCESS [1.000s]
[INFO] JORAM :: licenses ................................. SUCCESS [0.219s]
[INFO] JORAM :: conf ..................................... SUCCESS [0.047s]
[INFO] JORAM :: assembly ................................. SUCCESS [8.500s]
[INFO] JORAM :: joram :: samples ......................... SUCCESS [1.797s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 34.969s
[INFO] Finished at: Mon Nov 08 14:15:22 CET 2010
[INFO] Final Memory: 69M/165M
[INFO] ------------------------------------------------------------------------
```

This creates a `target/` directory for each sub-module where you can find compiled classes and the produced artifact.

This also creates a `ship/` directory where you can find a Joram distribution similar to what you can obtain by downloading a binary version on Joram's website.

You can also compile only one sub-module using the previous method in the right directory.

For example, compiling jndi client:

```
$ cd jndi
$ cd client
$ mvn clean install
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building JORAM :: jndi :: client 5.4.1.54-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]

…

…

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2.187s
[INFO] Finished at: Mon Nov 08 14:22:04 CET 2010
[INFO] Final Memory: 8M/20M
[INFO] ------------------------------------------------------------------------
```

4.8.2. Generating the javadoc

Joram uses a maven profile to easily build the javadoc:

```
mvn install -P doc
```

This creates the target/apidocs/ folder containing the whole javadoc.

4.8.3. Generating a distribution

There is another maven profile available to create a complete distribution of Joram:

```
mvn install -P release
```

The produced files are copied in the `release/target/` directory, they are equivalent to the ones available for download on Joram's web server.

Remark: This command only works when the Joram folder comes from the SVN as the `release/` directory is not included in the Joram sources distribution.

4.8.4. Cleaning

To remove the generated classes and libraries:

```
mvn clean
```

This removes all `target/` directories present in the project and the `ship/` directory containing an expanded Joram distribution.