Joram 4.3
User's Guide

Last modified 16/11/2005 by ScalAgent D.T.

Contents

Figures

1. Installation

Joram 4.3 includes:

- A **messaging server** (or MOM), providing the messaging functionalities: basically hosting and routing the messages exchanged by the client applications.
- 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.
- A **graphical administration and monitoring tool**, allowing to modify and visualise the state of a JORAM platform (made of one or many interconnected servers).
- A specific set of classes usable by client applications running on a **J2ME** environment. This set of classes is JMS 1.1 like, and is called **kJoram**.
- **Samples** illustrating the various features provided by Joram.
- JCA 1.5 connector allowing deployment in J2EE 1.4 platform.

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, etc.
- Connectivity: TCP/IP.
- Java environment: JDK 1.4.

1.2. Getting Joram binary distribution

The packages are downloadable from the following location:

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

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

- **`joram-x.y.z.tgz`**, including the client and server libraries, as well as the graphical tool library, the J2ME client library, the javadoc and the samples sources.

A package is expanded by UNIX users with the ***`gunzip`*** and ***`tar`*** commands; Windows developers can use the ***Winzip*** utility.

1.2.1. Directory structure and description

Joram binary distribution

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

- `apidoc/`
- `samples/`
  - `bin/…`
  - `config/…`
  - `src/`
    - `joram/…`
    - `kjoram/…`
- `ship/lib/`
- `ship/licenses/`

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 3). This directory contains:

- `a3config.dtd`, the DTD for server configuration;
- `a3debug.cfg`, a default logger configuration file;
- `centralized_a3servers.xml`, a configuration file for a centralized server architecture;
- `distributed_a3servers.xml`, a configuration file for a distributed servers architecture;
- `soap_a3servers.xml`, the configuration file for the SOAP samples server architecture;
- `jndi.properties`, a default configuration file for JNDI's clients.
- `soap_jndi.properties`, the configuration file for JNDI's SOAP 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) 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.

The **`samples/src/kjoram`** directory includes the samples codes of kJORAM clients. Compiling is done with the *`ant`* command, utility files are provided to Pocket PC users.

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 samples

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 it 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 (property "Transaction" is set to "fr.dyade.aaa.util.NullTransaction" in a3servers.xml configuration file).

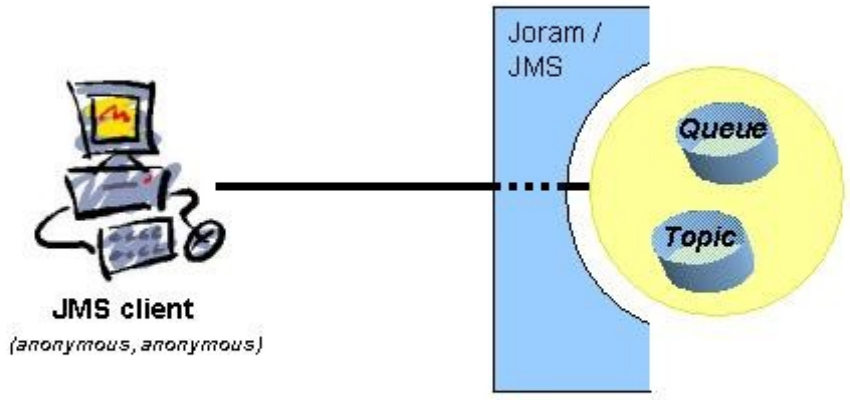


Figure 1 - Classic samples configuration

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.

- 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.
- Using the unified messaging domain:
  - It uses the common `ConnectionFactory` "cf", and the `Destination` "queue" and "topic" retrieved from JNDI.
  - For running the consumer sample, type "ant consumer". It continuously reads messages sent to the queue or the topic.
  - For running the producer sample, type "ant producer". It sends 10 messages to the queue, and 10 messages to 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).

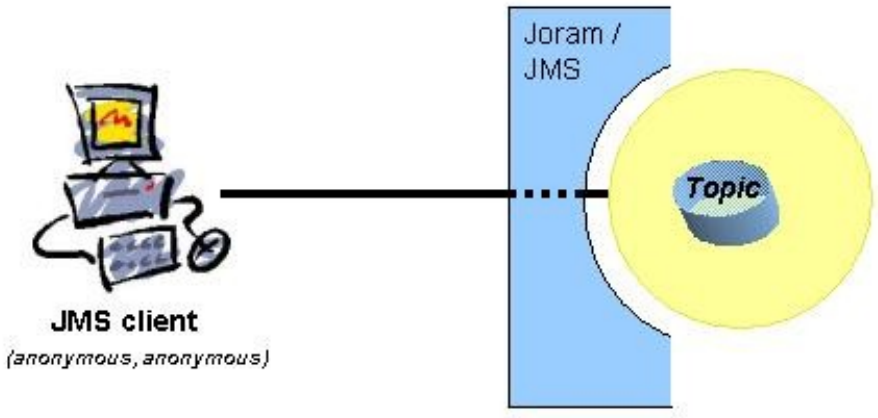


Figure 2 - Chat sample configuration

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.

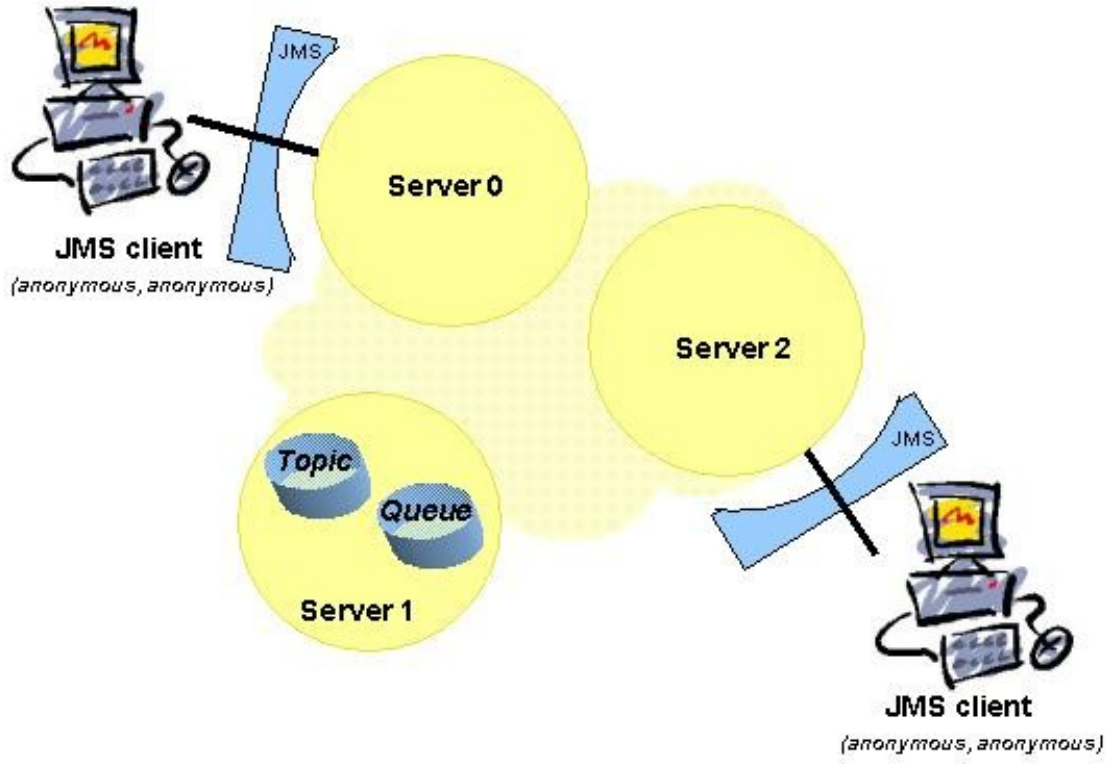


Figure 3 - Distributed sample configuration

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.

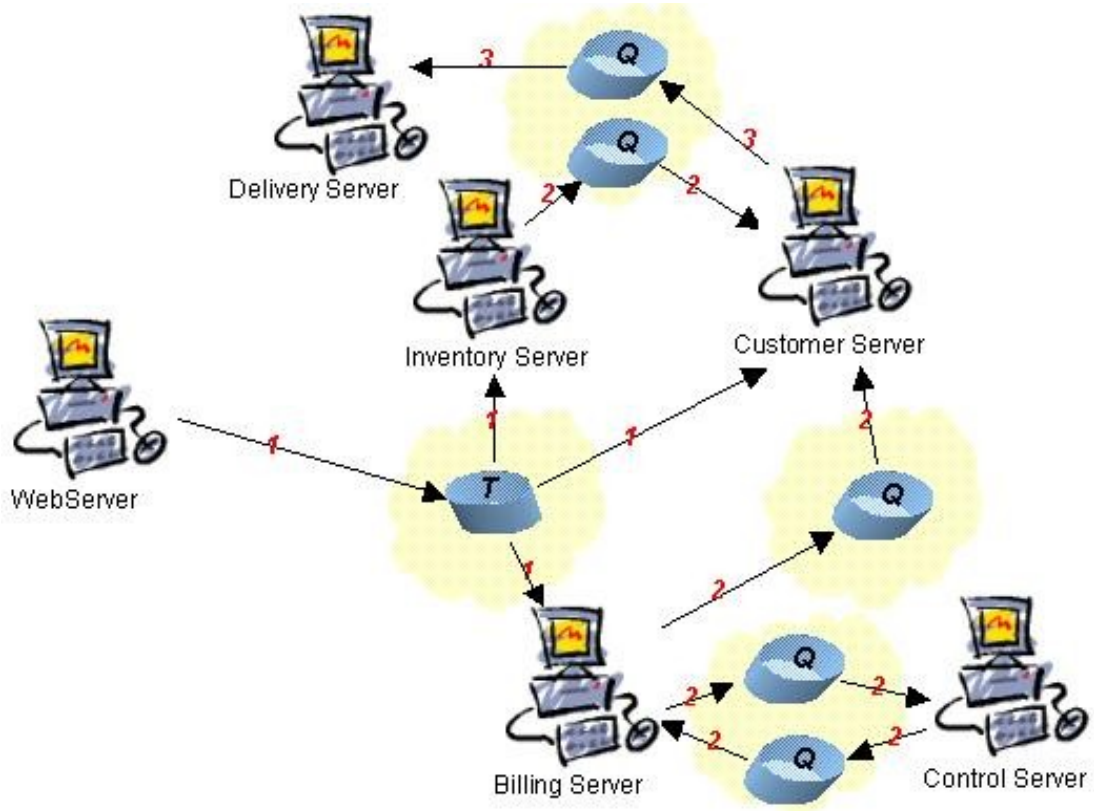


Figure 4 - Dotcom sample configuration

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.

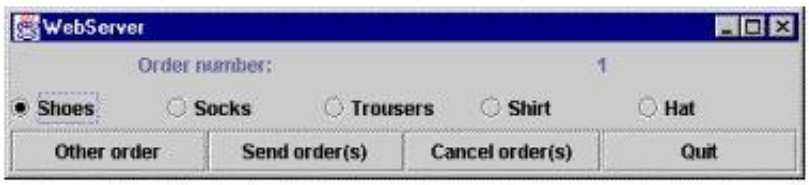


Figure 5 - Web Server's interface

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 the JORAM administration and monitoring tool

Launching the graphical admin and monitoring tool is simply done by executing the *admin.[sh/bat]* script. The tool may be launched before or after starting the JORAM platform.

This supposes that the administration tool has been compiled (by the CVS and sources package users), and that the joramgui.jar library is available in the ship/lib/ directory. Also, the admin GUI tool requires to use a 1.4 jdk.

The JORAM platform may be administered either directly through the tool, or still by launching the appropriate administration client. Nothing prevent you from running an admin code and watching in the tool the configuration of the platform.

2.3.4. 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).

2.3.5. 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 3.6).

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 functionnalities, 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.

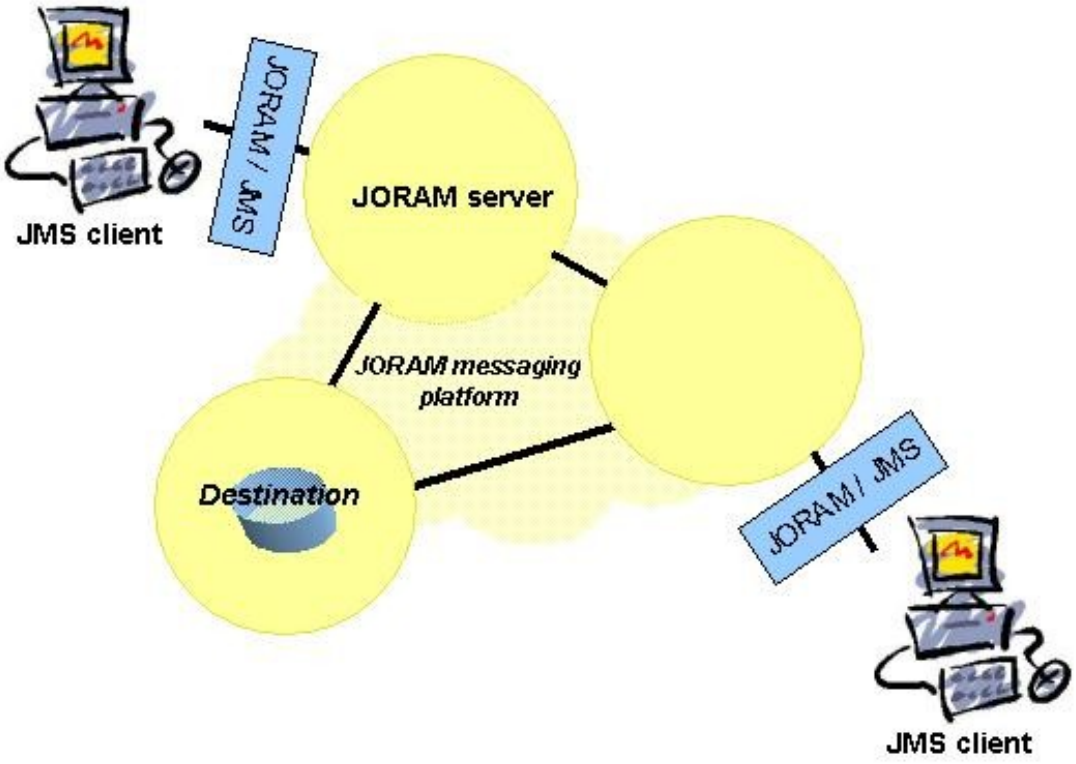


Figure 10 - Joram platform and clients

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 SOAP or "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).

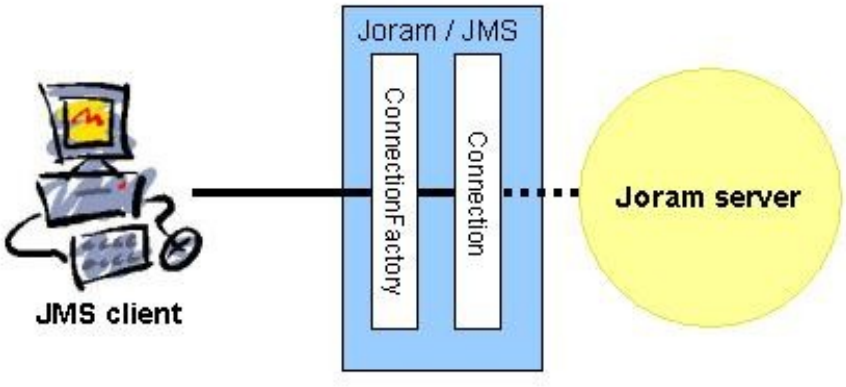


Figure 11 - A client connected to a server "through" a standard ConnectionFactory

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).

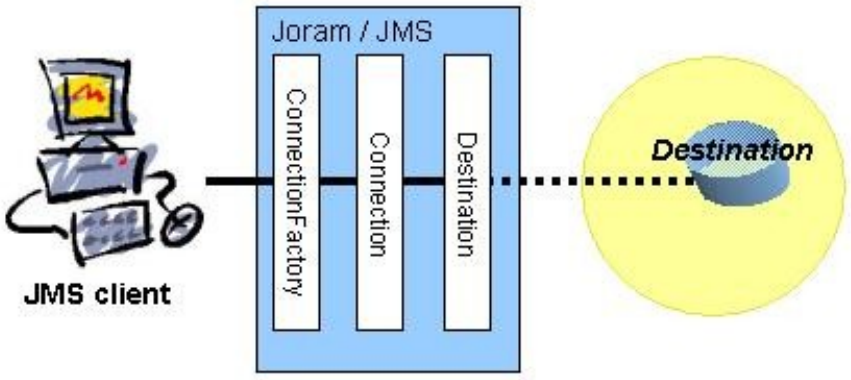


Figure 12 - A client accessing a server destination"through" a standard Destination

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).

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 5.1 below), 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 SOAP or “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://www.objectweb.org/monolog/) 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 graphical interface provided since release 3.7 and documented separately (Administration and Monitoring Tool).

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.soap
- 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

- `User.create(String name, String password, int server)` is a static method setting a user with a given identification on a given server, and creating the corresponding `User` instance.
- `User.create(String name, String password)` is similar to the previous method, except that it creates the user on the server the administrator is connected to (local server).

```
User user = User.create("name", "pass", 0);
```

- 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.

Updating a user identity

- `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

- `User.delete()`: unsets the user.

```
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 connection;
- or a SOAP connection (documented in Using SOAP specific documentation);
- or "collocated" (documented in Using a colocated server specific documentation).

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: 5).

Creating a destination: Queue or Topic

- `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)`

[1] Not a JNDI's 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 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 (see chapter 5.1)

- `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.

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

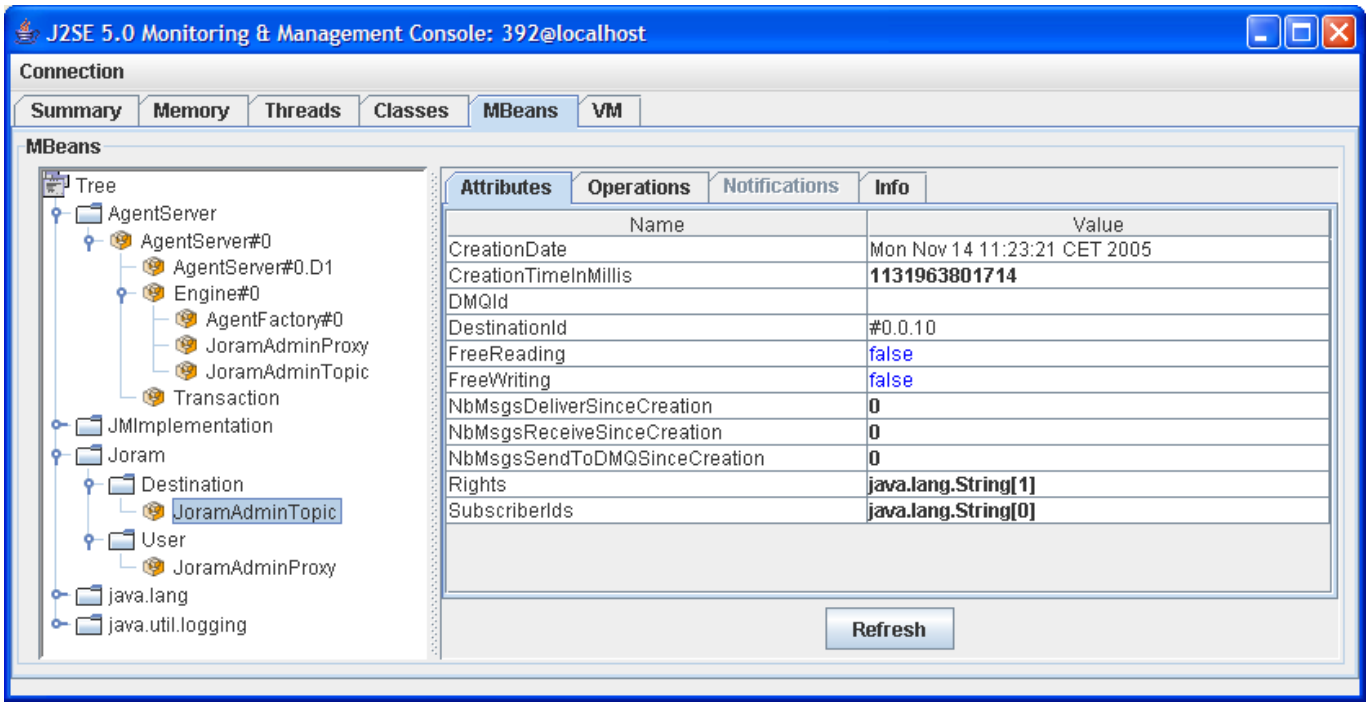
- `AdminModule.getUsers(int serverId)`: returns a `List` containing `User` instances representing all the users set on a given server.
    - An `AdminException` is thrown if the target server does not belong to the platform. A `ConnectException` is thrown if the admin connection to the server is closed 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. JMX administration of Joram

You can configure your Joram server to export somes 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:



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.

3.6. ScriptsXML

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 CVS: “joram/src/org/objectweb/joram/client/jms/admin/joramAdmin.dtd”.

3.6.1. Administrator connection

First, it needs to define the administration connection through a connect tag; you can define various attributes:

- `host`: the DNS name or IP address of the machine hosting the server (default localhost).
- `port`: the listen port of the Joram's TCP service (default 16010).
- `name`: the user identity of administrator (default root).
- `password`: the password for the administrator user (default root).
- `cnxTimer`: time-out value in seconds for the connection (default 60).
- `reliableClass`: implementation class for the protocol between the client and the server. By default it use the TCP protocol:

`org.objectweb.joram.client.jms.tcp.ReliableTcpClient`.

3.6.2. User and connectivity

ConnectionFactory

Each ConnectionFactory is defined by an element with an attribute class specifying the classname of the implementation. It is completed by a protocol element (local, tcp or soap for example) and usually a jndi binding:

- `local`: empty element defining a colocated connection.
- `tcp`: define the tcp settings for a TCPConnectionFactory:
  - `host` and `port`: host address and listen port for server, by default `localhost:16010`.
  - `reliableClass`: the class implementation.
    - `host` and `port`: host address and listen port for server, by default `localhost:8080`.
- `soap`: define the protocol settings for a SoapConnectionFactory.

User

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

- `name`, `password`: login and password for user.
- `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.

3.6.3. Destination

Th 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 JNDI 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`:
- `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 JNDI 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`.

DMQueue

- `name`: the Joram's internal name for the Dead Message Queue.
- `serverId`: unique identifier of location server. If not set the DMQ is created on the server the administrator is connected.

Cluster

A cluster is made up of a set of Queue or Topic elements. Each destination is created separately then linked.

3.6.4. Destination security and naming

Destination element can be completed by security or naming settings:

freeReader

Grants the read right to all on this destination.

freeWriter

Grants the write right to all 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`.

jndi

Registers the destination in JNDI context. The symbolic name is given in the attribute `name`.

property

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

3.6.5. 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>
  <AdminModule>
    <connect hostName="localhost" port="16010" name="root" password="root"/>
  </AdminModule>

  <ConnectionFactory
    className="org.objectweb.joram.client.jms.tcp.TcpConnectionFactory">
    <tcp host="localhost" port="16010"/>
    <jndi name="cf"/>
  </ConnectionFactory>

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

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

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

4. Administration and Monitoring Tool

4.1.1. Overview

The JORAM Administration & Monitoring Tool (jAMT) is a graphical user interface which can be used to manage JMS connection factories, destinations and users pertaining to a single JORAM server or to a cluster of servers. The jAMT allows an administrator to monitor, create, modify and remove various server elements and their corresponding JNDI entries using an intuitive point and click interface.

4.1.2. Guide

Startup

Starting up the jAMT is very easy using the scripts provided in the samples/bin directory.

Tip: Instead of having to open a Command Prompt window every time you want to start the jAMT, you can create a Windows shortcut to the admin.bat file on your desktop so that you may start the tool by clicking on the icon without having to open a window.

Administration server connection

To use the jAMT, you need to first establish a connection to the administration port of a JORAM server. This is done by selecting the Admin > Connect menu option. A dialog box will appear where you need to enter the administration server's host name and administration port (defaults are provided) and the administrator user name and password. Once you click on the Connect button, the jAMT will try to establish a connection to the server. At that point, the server needs to be up or else the jAMT will wait and retry until the server is up.

Once a connection is successfully established with the server, the Configuration tab of the navigation panel (left-hand side of the main window) will contain a tree view of the server and of its destinations and users. As long as the jAMT is connected to a JORAM server, the Admin Connection status bar at the top of the information panel (right-hand side of the main window) will show the host name and port of the server in green. When the jAMT is not connected to any server, the status bar displays a "Not connected" message in red.

When you are connected to a JORAM server, you can easily refresh the configuration tree view under the Admin tab of the navigation panel by selecting the Admin > Refresh menu option. This will reload all of the configuration information from the server. This option can be useful to see changes that may have taken place through another administration client.

To disconnect the jAMT from a JORAM server, select the Admin > Disconnect menu option. The connection will be closed and the configuration tree under the Admin tab of the navigation panel will be emptied. The status bar will display a "Not connected" message in red.

JNDI Server Connection

In addition to connecting to the JORAM administration port, the jAMT needs to be connected to a JNDI for certain administration operations such as the creation and deletion of destinations and of connection factories. The JNDI connection is required because these JORAM elements need to be identified through JNDI to be manipulated without ambiguity and to be usable by JORAM clients.

To establish a connection to the JNDI directory, you need to select the JNDI > Connect menu option. A dialog box will appear where you need to enter the JNDI server's host name, port number (defaults are provided) and optionally a context name (not required in most cases). Once you click on the Connect button, the jAMT will try to establish a connection to the JNDI server. Once the connection is established, the content (bound names) of the JNDI directory will be displayed in a tree view under the JNDI tab of the navigation panel.

When you are connected to a JNDI directory, you can easily refresh the JNDI tree view under the JNDI tab of the navigation panel by selecting the JNDI > Refresh menu option. This will reload all of the binding information from the directory. This option can be useful to see changes that may have taken place in the directory since the connection was established.

To disconnect the jAMT from a JNDI directory, select the JNDI > Disconnect menu option. The connection will be closed and the JNDI tree under the JNDI tab of the navigation panel will be emptied.

Server Administration

The jAMT can be used to modify a server's configuration parameters and to stop a running server.

Server Configuration

A server's configuration parameters can be modified by clicking on the server under the Admin tab in the navigation panel. The server's information and current configuration parameters' values will be displayed in the information panel (right-hand side). The parameters can be edited directly in the information panel and the changes can be submitted by clicking on the Apply Changes button at the bottom of the information panel.

Server Shutdown

A JORAM server can be stopped through the jAMT by selecting the server under the Admin tab in the navigation panel, by right-clicking on the server's entry under the Admin tab and then by selecting the Stop Server option in the popup menu. You will be asked to confirm your action. Once you confirm, the server will be sent a shutdown message.

Warning: It is important to note that once a server is stopped, it cannot be restarted through the jAMT. If the server providing administration services is shutdown, you will not be able to continue to use the jAMT tool to manage other dependant servers in the same group.

Connection Factory Administration

The jAMT can be used to create and delete JMS connection factories for a JORAM server in the JNDI directory. JMS connection factories are objects that are stored in a JNDI directory and used by JMS clients to establish a connection to a JMS server. A connection factory contains mostly the host name and port of the server to which it provides connections and the type of connections (Queue or Topic, transacted or not, etc.) that it handles.

Connection Factory Creation

To create a new connection factory, the jAMT needs to be connected to a JNDI server. You need to select the JNDI > Create Connection Factory menu option, which will display a dialog box where information can be entered about the connection factory to be created. The host name and port number of the JORAM server that this connection factory will point to are required, as well as the JNDI name of the new connection factory. The type of connection factory to create also needs to be selected by using the radio buttons. Once you click on the Create button, the connection factory will be created and bound in the JNDI directory. It becomes immediately available for use by JORAM clients.

Connection Factory Deletion

To delete an existing connection factory, the jAMT needs to be connected to a JNDI server. You need to click on the name of the connection factory that you want to to delete under the JNDI tab of the navigation panel. Once the connection factory's information is displayed in the information panel (right-hand side), right-click on the factory's name in the navigation panel and select the Delete option in the popup menu that appears. You will be asked to confirm your action. Once you confirm, the connection factory you selected will be unbound and removed from the JNDI directory.

Destination Administration

The jAMT can be used to monitor, create, configure and delete destinations on JORAM servers.

Destination Creation

To create a new destination on a JORAM server (Queue or Topic), you need to also be connected to the JNDI server since the new destination will have to be bound to a name in the JNDI directory. To create the new destination, right-click on the server on which you want to create it under the Admin tab of the navigation panel and select the Create Destination option in the popup menu. A dialog box will appear where you need to enter the name to which the new destination will be bound in the JNDI directory and to select the type of destination to create by using the radio buttons. Once you click on the Create button, the destination will be created and bound in the JNDI directory. It will be displayed in the Admin tree of the navigation panel, under the server on which it was created.

Destination Monitoring and Configuration

To check the status of a destination, to change its configuration parameters or to update its access control lists, you need to select the destination under the Admin tab of the navigation panel. The destination's current information will be displayed in the information panel (right-hand side). This information includes:

Destination identifier: the JORAM unique identifier for the destination

JNDI name: the name to which this destination is bound in the JNDI directory (only available if connected to the JNDI directory)

Destination type: whether the destination is a Queue, a Topic or a Dead Message Queue

Pending messages: the number of messages currently in the queue (not available for Topics)

Pending requests: the number of requests currently waiting on the queue (not available for Topics)

Threshold: the destination's specific threshold

Dead Message Queue: the DMQ associated with the destination

Allow free reading: allows any user to receive messages from the destination when selected

Allow free writing: allows any user to send messages to the destination when selected

Reading access control list: this list controls who has access to the destination to receive messages

Writing access control list: this list controls who has access to the destination to send messages

The destination's configuration parameters can be edited directly in the information panel and the changes can be submitted by clicking on the Apply Changes button at the bottom of the information panel.

Destination Deletion

To delete an existing destination, you need to be connected to the JORAM server as well as to the JNDI directory where that destination is bound. You have to select the destination's name under the JNDI tab of the navigation panel, to right-click on the name and then to select the Delete option in the popup menu. You will be asked to confirm your action. Once you confirm, the destination will be unbound from the JNDI directory and it will be deleted from the server where it is located.

User Administration

The jAMT can be used to create and delete users on JORAM servers and to change their passwords.

User Creation

To create a new user on a JORAM server, right-click on the server on which you want to create it under the Admin tab of the navigation panel and select the Create User option in the popup menu. A dialog box will appear where you need to enter the user name and password for the user that you wish to create. Once you click on the Create button, the user will be created on the server and will have immediate access to that server. It will be displayed in the Admin tree of the navigation panel, under the server on which it was created.

User Deletion

To delete an existing user, you have to select the user's name under the Admin tab of the navigation panel, to right-click on the name and then to select the Delete option in the popup menu. You will be asked to confirm your action. Once you confirm, the user will be deleted from the server where it existed and will no longer have access to that server.

Password Change

The change the password of an existing user, you have to select the user's name under the Admin tab of the navigation panel, to right-click on the name and then to select the Change Password option in the popup menu. You will be asked to enter a new password for the user. Once you click on the Apply button, the user's password will be changed on the server where it exists and the previous password will cease to work immediately.

Known Issues

When the jAMT is started, in certain cases, the window is displayed but it doesn't contain any components (it only shows an empty panel). This problem does not affect how the jAMT functions and is strictly related to the interface not displaying. There is an easy work-around when it happens, which is to minimize and then restore the window. When it is restored, all of the window's content will be displayed properly.

4.1.3. Using GUI with classic sample

The chapter 2 describes the use of samples provided with Joram, here we're using the classic sample (2.2.1) to demonstrate the use of the JAMT Tool.

Compile the sample

```
$> cd JORAM_DIR/samples/src/joram
$> ant clean compile
```

Launch the platform

```
$> ant reset single_server &
Buildfile: build.xml

…

server:
     [java] AgentServer#0 started: OK
```

Launch the tool

```
$> ant admin_gui
```

To use the Jamt tool, you need to first establish a connection to the administration port of a JORAM server. This is done by selecting the Admin>Connect menu option. A dialog box (Figure 13) will appear where you need to enter the administration server's host name and administration port (defaults are provided) and the administrator user name and password ("root", "root" by default. Once you click on the Connect button, the jAMT will try to establish a connection to the server.

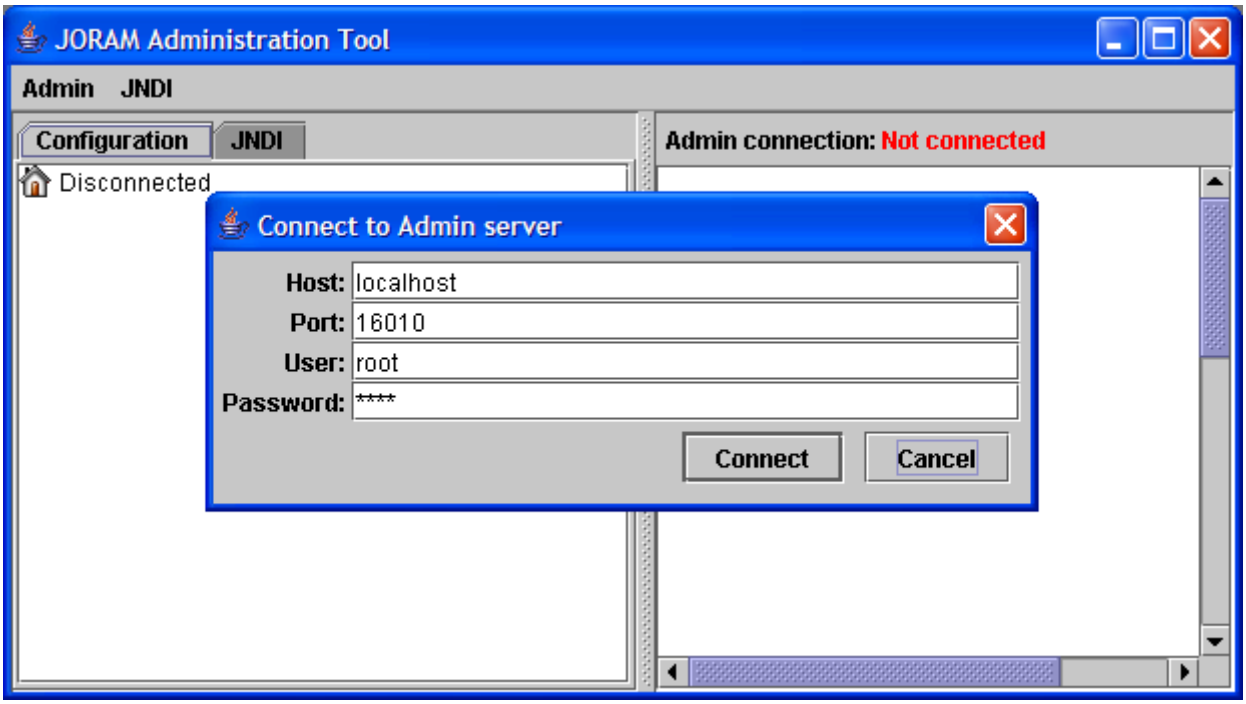


Figure 13 - Connection to the server

Once a connection is successfully established with the server, the Configuration tab will contain a tree view of the server and of its destinations and users. At this point, there is a unique user "root" and no destination (Figure 14).

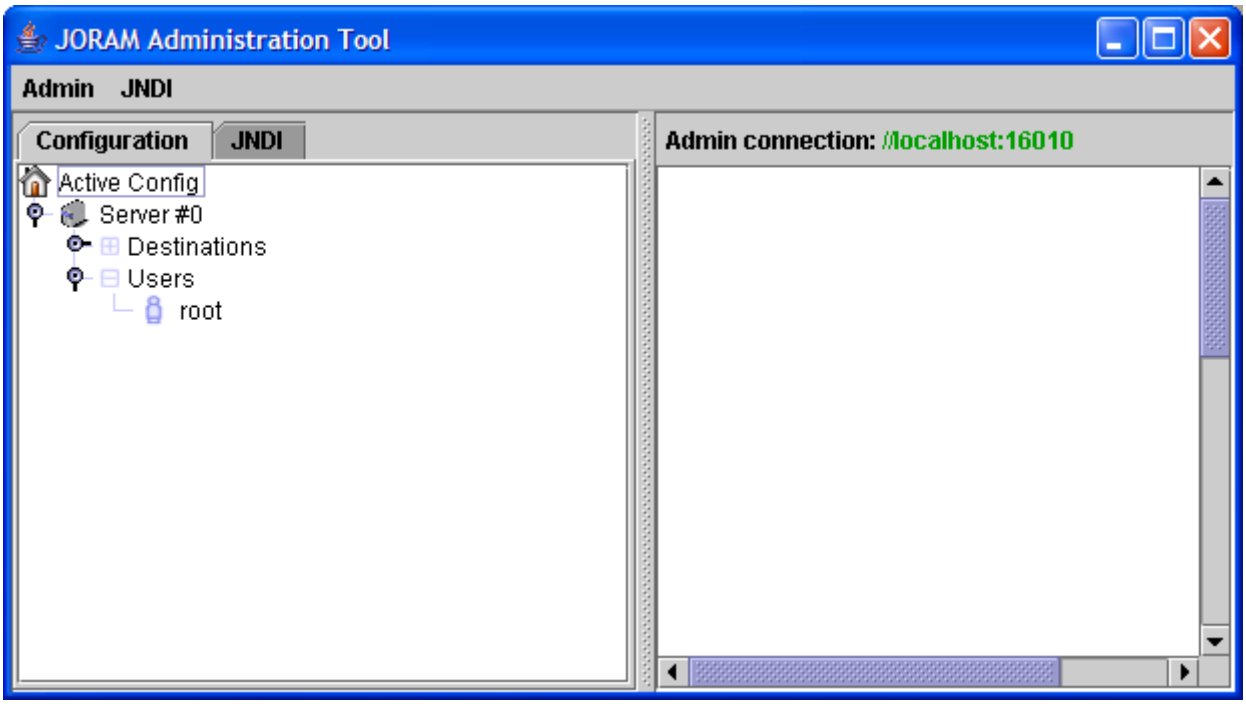


Figure 14 - JAMT connected

In addition to connecting to the JORAM administration port, the tool needs to be connected to a JNDI for certain administration operations. The JNDI connection is required because JORAM elements need to be identified through JNDI to be manipulated without ambiguity and to be usable by JORAM clients.

To establish a connection to the JNDI directory, you need to select the JNDI>Connect menu option. A dialog box will appear where you need to enter the JNDI server's host name, port number (defaults are provided). Once the connection is established, the content of the JNDI directory will be displayed in a tree view under the JNDI tab of the navigation panel.

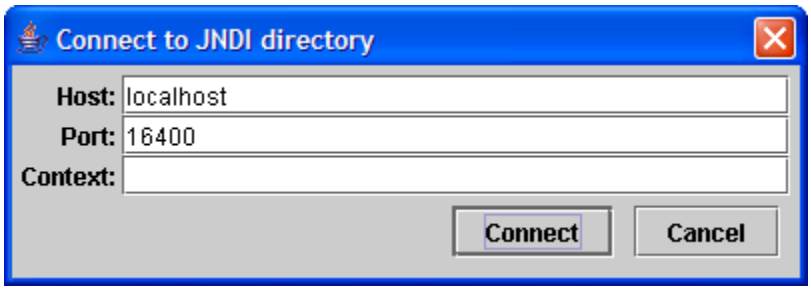


Figure 15 - JNDI connection Dialog box

Run the administration code

Now, we can launch the administration program of the classic sample[2] ; this programm will create one user (“anonymous”) and 2 destinations: a queue “queue” and a topic “topic”.

[2] All theses administration operations can also be made in the JAMT Tool.

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

init:

classic_admin:

     [java] Classic administration...
     [java] Admin closed.

BUILD SUCCESSFUL
Total time: 2 seconds
```

You can easily refresh the configuration tree view under the Admin tab of the navigation panel by selecting the Admin>Refresh menu option, then the view obtained is:

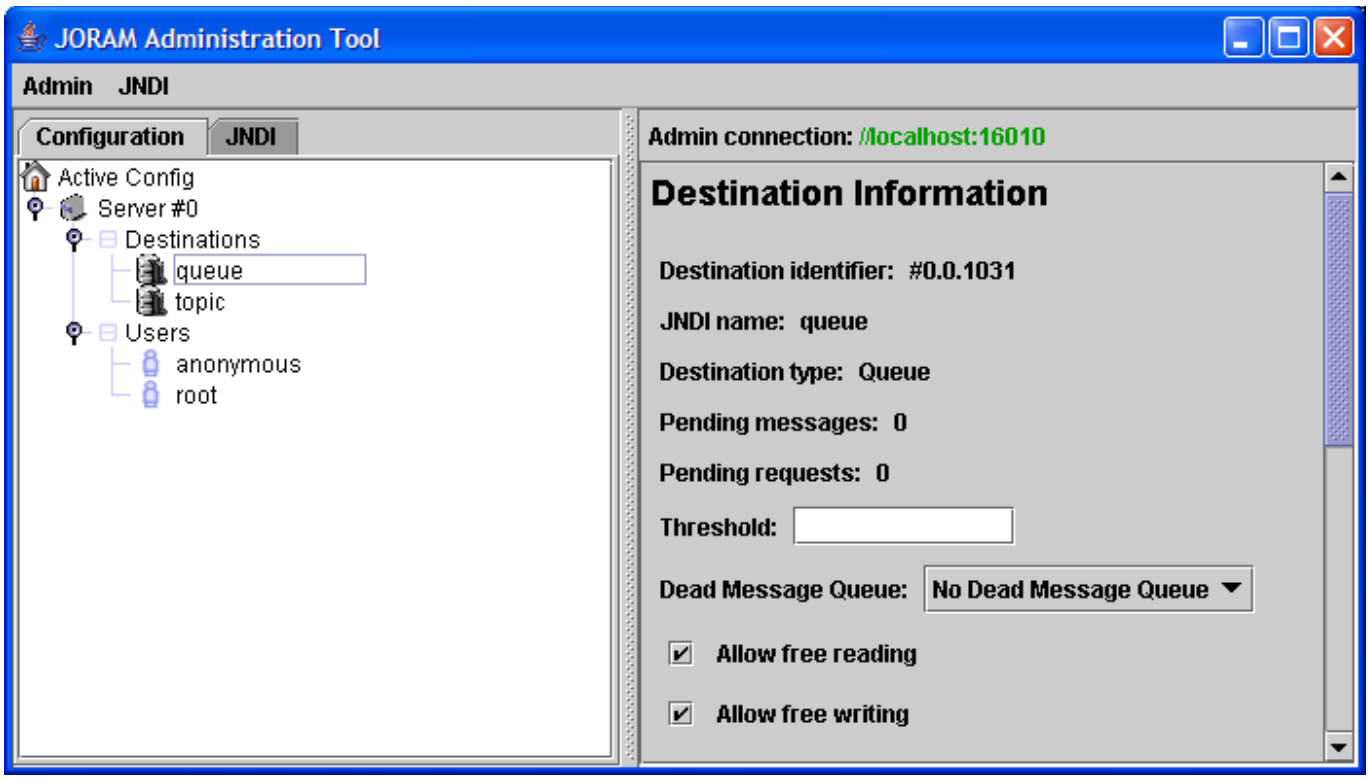


Figure 16 - Server view after administration

If you launch the message producer, by typing "`ant sender`", it sends 10 messages to the queue. If you refresh the view by clicking again the queue, you can look the 10 waiting messages.

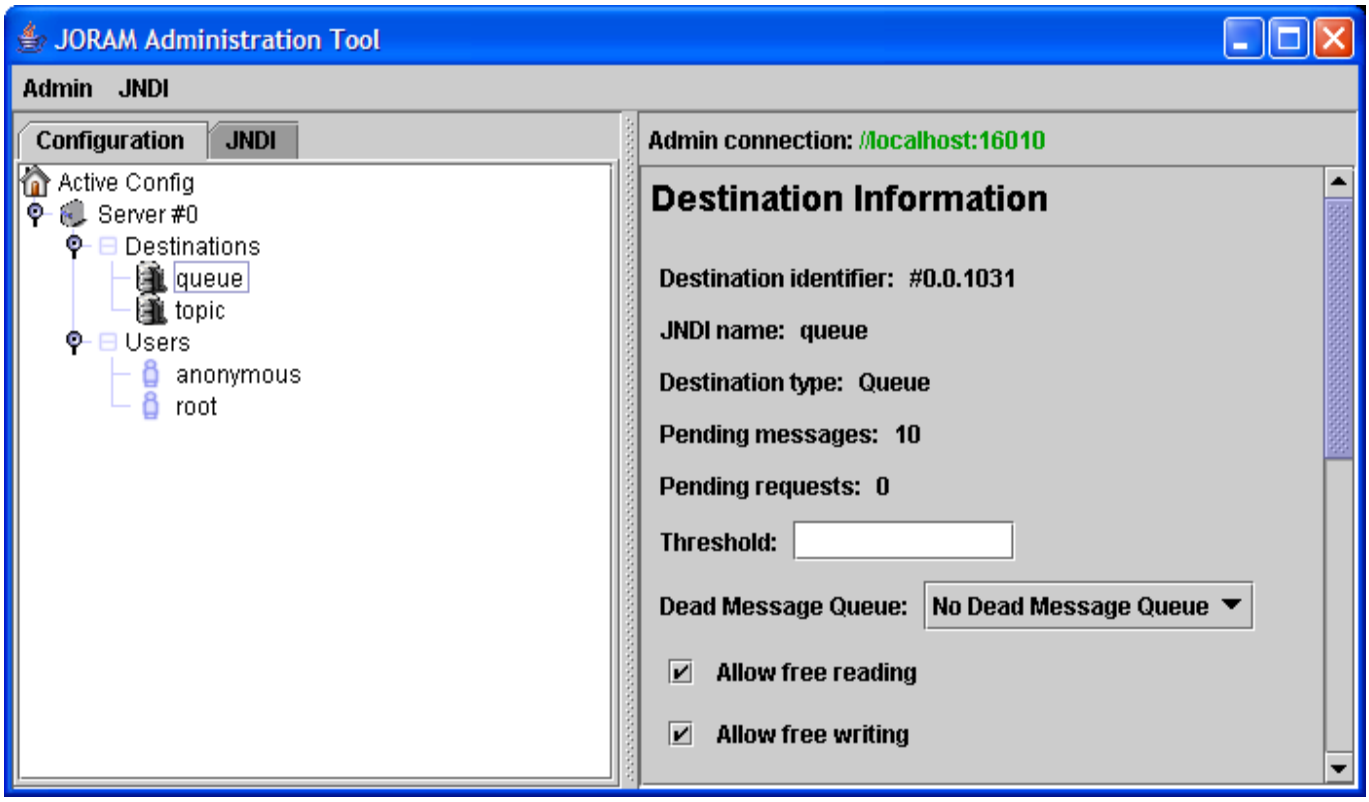


Figure 17 - Server view after messages sending

Then you can launch the message consumer, by typing "`ant receiver`"…

4.1.4. Additional monitoring features

Subscriptions

A user node owns a subnode called `subscriptions` that lists all the subscriptions opened by this user. The node is titled: <subscription name>(<topic id>, <undelivered messages count>)

Each subscription node contains one node per undelivered message.

The loading of subscriptions, messages identifiers and contents is lazy:

- The subscriptions are loaded when the node `subscriptions` is selected.
- The identifiers of the undelivered messages are loaded when the subscription node is selected.
- The message content is loaded when the message node is selected.

You can delete one or more messages from a subscription by selecting the popup menu of the message node. You can also clear the whole subscription (popup menu of the subscription node).

Queues

You can monitor the content of a queue by selecting the subnode `messages` of the node that represents the queue. A node is created for each undelivered message in the queue.

The loading of messages identifiers and contents is lazy:

- The identifiers of the undelivered messages are loaded when the node `messages` is selected.

- The message content is loaded when the message node is selected.

You can delete one or more messages from a subscription by selecting the popup menu of the message node.

Topics

A topic node contains a sub node called `subscribers` that contains the identifiers of the users that subscribed to this topic.

The loading of the users identifiers is lazy:

- The identifiers of the users are loaded when the node `subscribers` is selected.

4.1.5. Dynamic configuration

Configuration representation

The platform configuration is structured as a tree. The root node is the Joram server connected to the tool: `Server#0`. A server node has 3 children:

1. `Destinations`
2. `Users`
3. `Domains`

The nodes `Destinations` and `Users` respectively contain the nodes of the destinations and the users created in the Joram server. The node `Domains` contain the domains declared in the Joram server configuration: none in this configuration.

If you want to extend this configuration with new servers you first have to define a domain where these servers will be added.

Define a new domain

Right click on the node `Server#0` in order to show its popup menu and select the item `Create domain...`. A dialog window asks you to define the new domain:

1. Domain name
2. Port

Give the following values:

1. D0
2. <port used by Server#0 to communicate with the other servers from the domain D0>

A domain node can contain some server nodes, the nodes of the servers that belong to this domain. The domain `D0` is empty.

Click `Apply`: a new node appears called `D0`.

Define a new server

Select the popup menu of the domain node `D0` and select the item `Create server...`. A dialog window asks you to define the new server:

1. Server name
2. Host name
3. Server id
4. Port

Give the following values:

1. S1
2. <name/address of the host where the server will be started>
3. 1
4. <port used by S1 to communicate with the other servers from the domain D0>

Be careful, the server name and its identifier must be unique across the platform.

Click `Apply`: a new node appears called `Server #1`.

Start the new server

Create a directory `run1` where the server `S1` will run. Open the popup menu of the node `Platform` and select the item `Save config...` Copy the platform configuration inside `run1` in a file called `a3servers.xml`. Then start the server:

```
cd run1
java AgentServer 1 S1
```

Once the server `S1` is started you can use it through the administration tool: create a destination, add a user...

5. Specialized destinations

5.1. Dead Message Queue

5.1.1. Dead message queue

Introduction

A dead message queue is a destination where dead messages are sent. A dead message is a message located server side and considered as undeliverable for various reasons. Those reasons are:

- 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.

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

The 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.

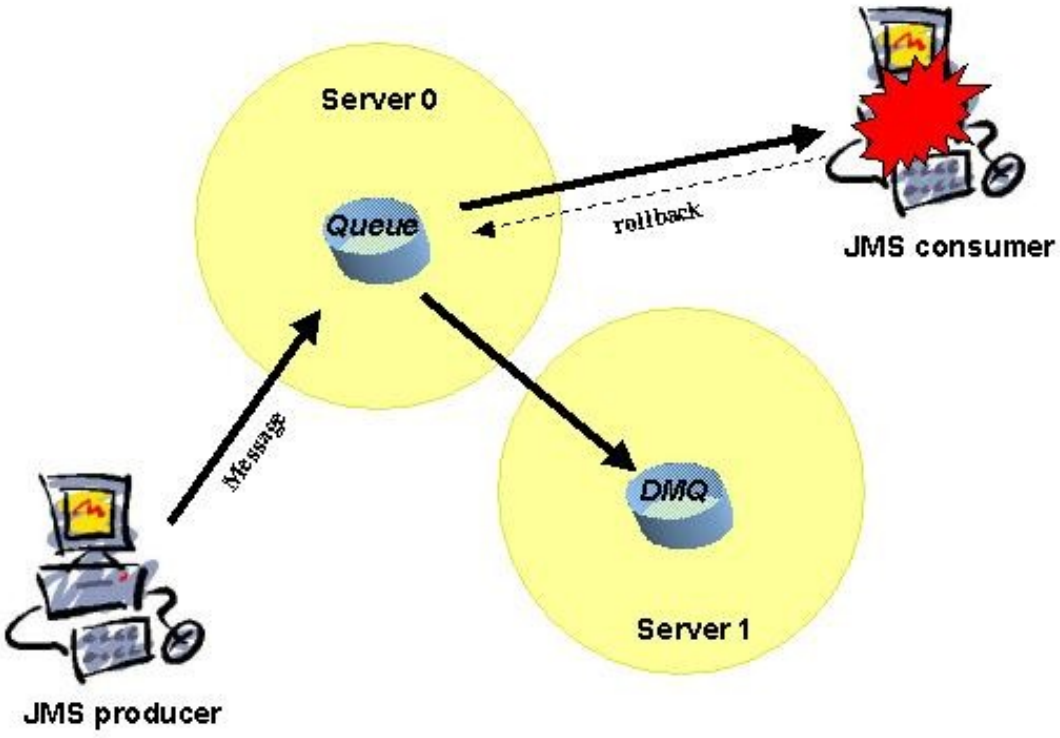


Figure 18 - Messages on a queue sent to a DMQ

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 destinated to 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 destinated to 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. The single difference with a "normal" queue is that a DMQ does not keep the dead messages it delivers for acknowledgement or denying. Also, it does not log its own messages (which are already dead) as dead messages on other DMQs.

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

- `JMS_JORAM_DELETEDDEST`, if the target destination of the message could not be found,
- `JMS_JORAM_NOTWRITABLE`, if the target destination of the message did not accept the sender as a WRITER,
- `JMS_JORAM_EXPIRED`, if the message expired before delivery,
- `JMS_JORAM_UNDELIVERABLE`, if the number of delivery attempts of the message overtook the threshold.

The `JMSXDeliveryCount` property is also available for getting the number of delivery attempts of the message. All those properties are available trough the dedicated `Message` methods, such as in:

```
// Getting a dead message through a DMQ consumer:
Message deadM = (Message) deadMconsumer.receive();

if (deadM.getBooleanProperty("JMS_JORAM_DELETEDDEST"))
  System.out.println("Destination does not exist.");
else if (deadM.getBooleanProperty("JMS_JORAM_NOTWRITABLE"))
  System.out.println("Non writable destination.");
else if (deadM.getBooleanProperty("JMS_JORAM_EXPIRED"))
  System.out.println("Message expired.");
else if (deadM.getBooleanProperty("JMS_JORAM_UNDELIVERABLE"))
  System.out.println("Undeliverable message.");

System.out.println("Number of delivery attempts :"
                   + deadM.getIntProperty("JMSX_DeliveryCount"));
```

5.1.2. Managing a Dead Message Queue

Creating a dead message queue

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

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

- An `AdminException` is thrown if the destination deployement 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, DeadMQueue 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(DeadMQueue 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(DeadMQueue dmq): sets a given DMQ as the DMQ for the destination (set DMQ as null for unsetting it).
- User.setDMQ(DeadMQueue 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 threhsold 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.

5.1.3. 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

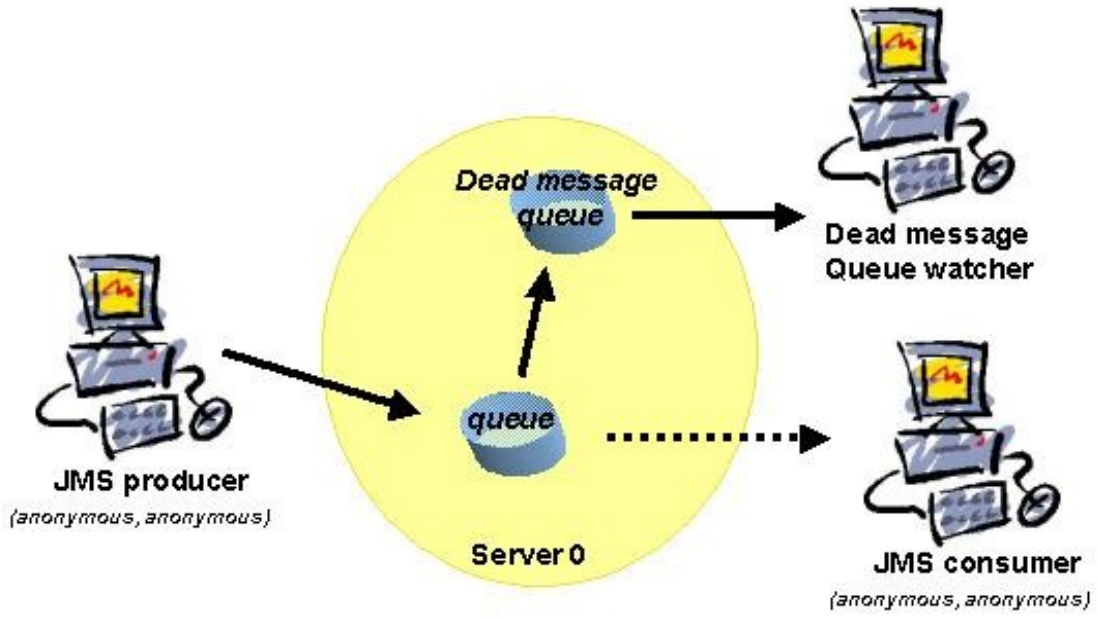


Figure 19 - Dead message queue sample

Running the demo:

- For starting the platform:

```
ant reset single_server
```

- For running the admin code:

```
ant dmq_admin
```

- For launching the watcher:

```
ant dmq_watcher
```

- For launching the producer and the consumer:

```
ant dmq_client
```

5.2. Hierarchical Topic

5.2.1. Hierarchical topic

Introduction

The JMS specification allows topics to have a hierarchical structure such as the one shown in . 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.

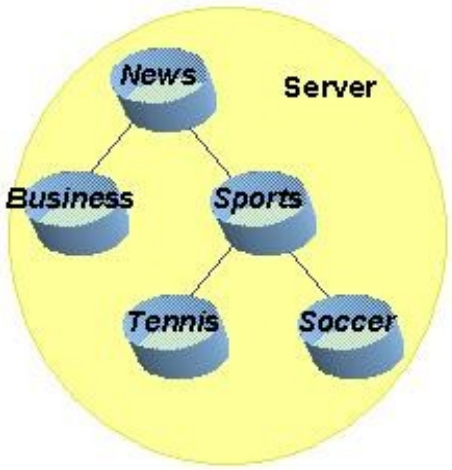


Figure 20 - A Hierarchical topic

Example

The example of Figure 20 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 21). 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).

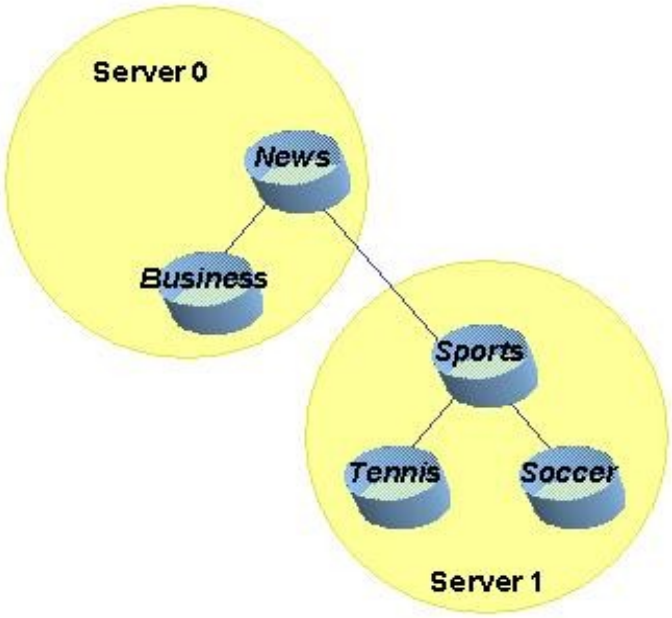


Figure 21 - A distributed Hierarchical topic

5.2.2. 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.

5.2.3. 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 subcription 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.

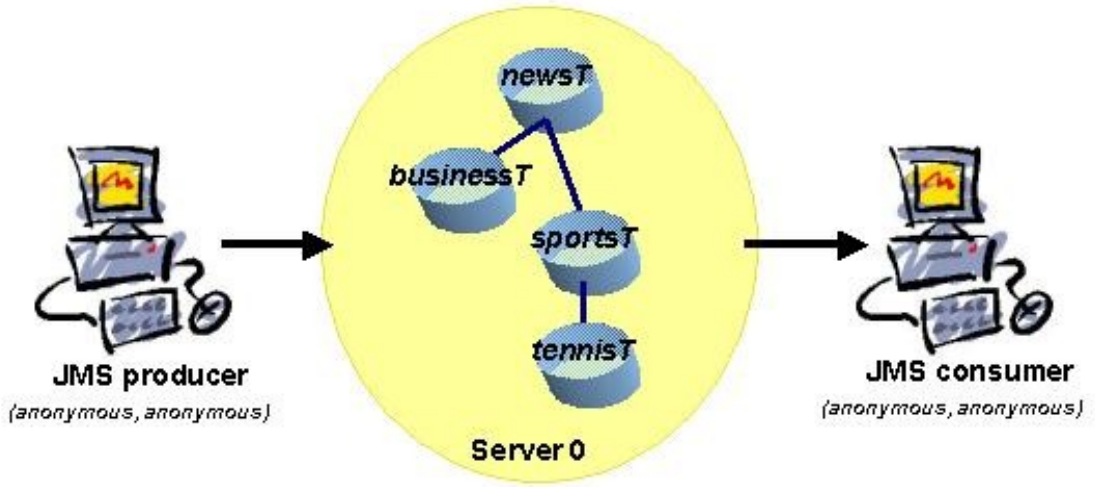


Figure 22 - Topic tree sample

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
  ```

5.3. Clustered Topic

5.3.1. 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 23 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 23, 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.

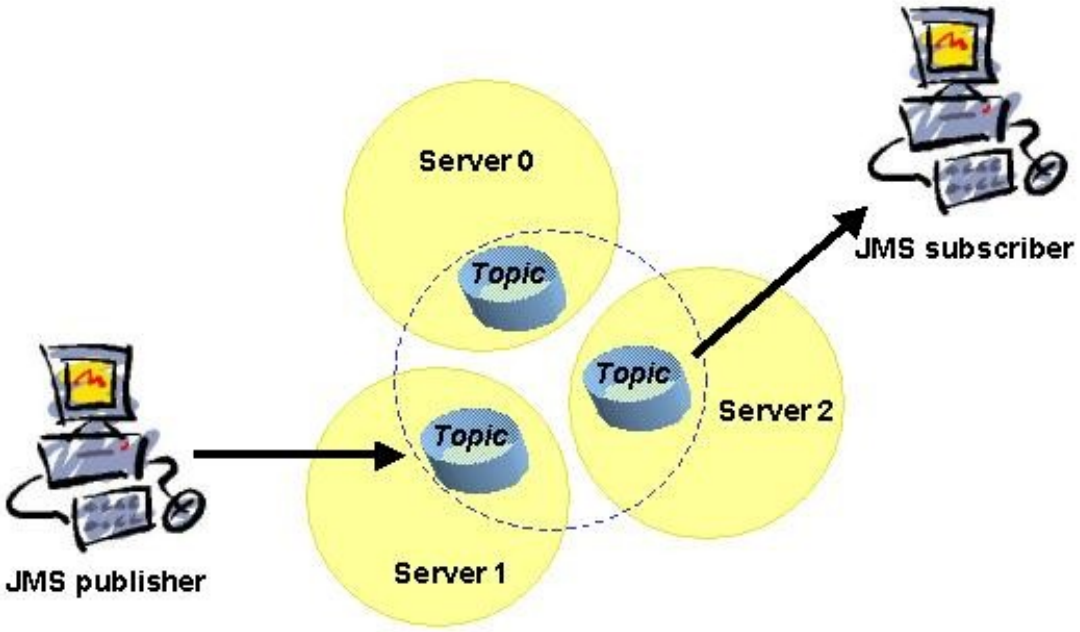


Figure 23 - A clustered topic

Added value

This special feature introduces more flexibility and availability to Pub/Sub communication. If server 2 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 0 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.

5.3.2. 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 the joining topic is already part of a cluster, or if one of the topics is part of a hierarchy. 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.

5.3.3. Running the "Clustered Topic" Sample

This sample illustrates the use of JORAM's clustered topic. JORAM's 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. A publisher connects to server 0 and publishes messages on the local topic representant, while subscribers are connected to servers 1 and 2 and have subscribed to the local representant of the topic. The platform is run in persistent mode. The provided configuration locates all three servers on "localhost" host.

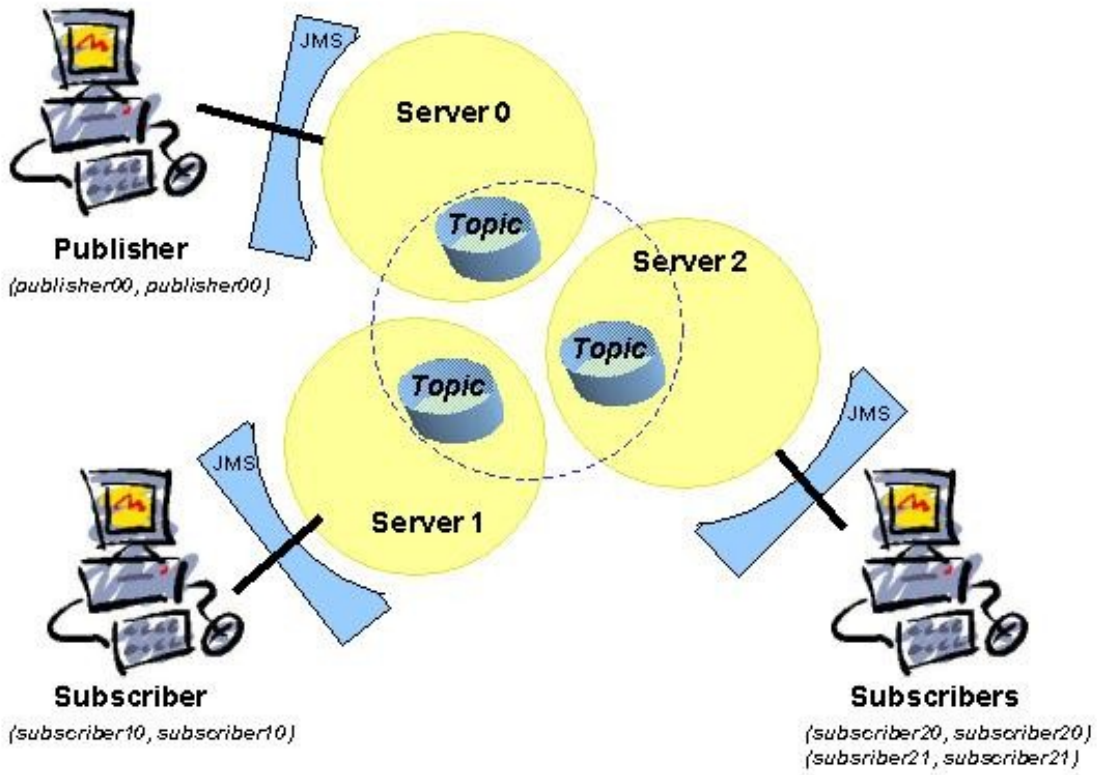


Figure 24 - Cluster sample configuration

Running the demo:

- Starting the configuration:

  ```
  ant reset servers
  ```

- Running the admin code:

  ```
  ant cluster_admin
  ```

- Running the subscribers:

  ```
  ant subscriber10
  ant subscriber20
  ant subscriber21
  ```

- Running the publisher

  ```
  ant cluster_pub
  ```

5.4. Clustered Queue

5.4.1. 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.

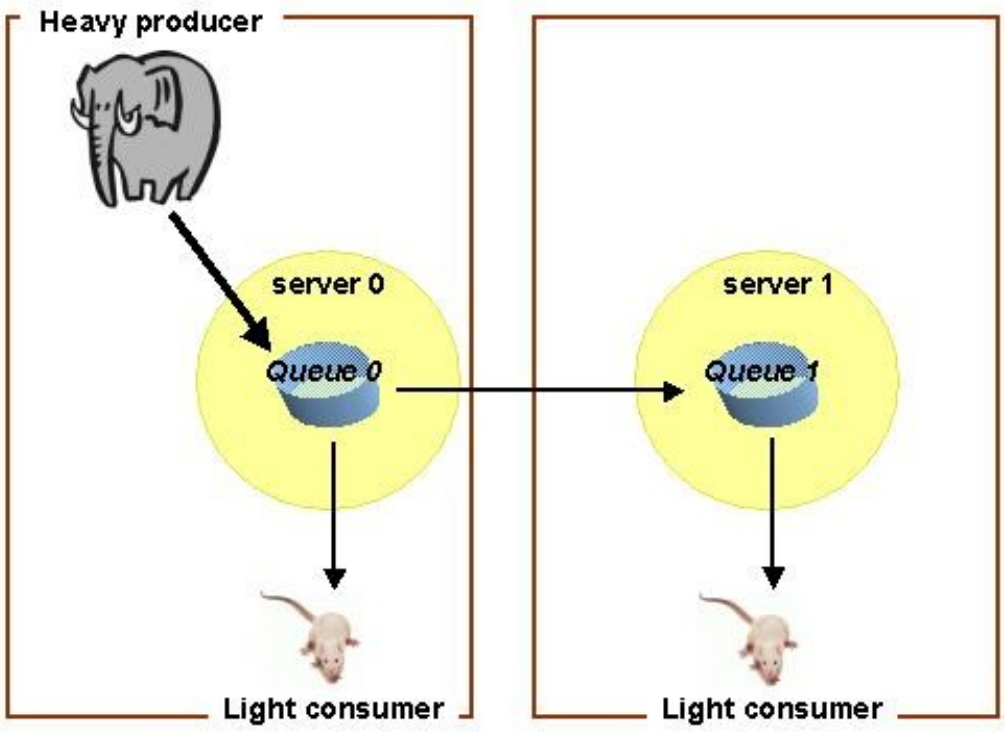


Figure 25 - A cluster of queues balancing heavy deliveries

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, 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.

5.4.2. Managing a clustered queue

Setting the clustered queue parameters

Prior to creating the cluster, the following parameters must be set:

- "period": period (in ms) of activation of the load factor evaluation routine for a 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 queue

Creating a clustered queue consists first in creating the queues that will be part of it. For a cluster of two queues, let's create queue 0 and queue 1 and servers 0 and 1 through the `Destination.doCreate` method.

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

String qid0 = Destination.doCreate(0, null, className, prop);
Queue queue0 = new Queue(qid0);

String qid1 = Destination.doCreate(1, null, className, prop);
Queue queue1 = new Queue(qid1);
```

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.addClustered(queue1);
```

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.

An object representing the cluster, and which may be bound to a JNDI server, should be instanciated 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. This information is contained in a hash table as follows:

```
java.util.Hashtable h = new java.util.Hashtable();
// queue 0 should be accessed by any client connected to server 0
h.put("0", queue0);
// queue 1 should be accessed by any client connected to server 1
h.put("1", queue1);
```

The object to instantiate is a `ClusterQueue`. It is an abstract class but a reference implementation is provided. Its constructor takes the hash table as parameter:

- `ClusterQueue(java.util.Hashtable h)`

```
ClusterQueue clusterQueue = new ClusterQueue(h);
```

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.

5.4.3. 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.

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.
- Running the consumers using the target *consumer0*, *consumer1*, *consumer2*.
  - Each of these target launches a message consumer that indefinitely consumes messages on the corresponding queue.
- Running the producers using the target *producer0*, *producer1*, *producer2*.
  - Each of these target launches a message producer that sends 100 messages to the corresponding queue.

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

Scenario1:

- Launching two consumers on queue0 and queue1

  ant consumer0

  ant consumer1

  ant consumer2
- Launching multiples producers on queue0

  ant producer0 & ant producer0 & ant producer0

In this scenario, messages are all sent to queue0, the load-balancing mechanism dispatchs them between the queues,then the consumers.

Scenario2:

- Launching two consumers on queue0 and queue1

  ```
  ant consumer0
  ant consumer1
  ```

- Launching two producers on queue1 and queue2

  ```
  ant producer1
  ant producer2
  ```

In this scenario, messages produced to queue2 by producer2 are dispatched between consumer0 and consumer1 by the load-balancing mechanism.

6. Using SOAP

The Joram client – server protocol may be based on **SOAP**. This allows clients to rely on HTTP rather on TCP for communicating with a JORAM platform. It also allows applications running on lightweight devices (J2ME environment) to use the JORAM platform through the **kJoram** libraries.

The SOAP protocol (more info on http://www.w3.org/TR/SOAP/) defines a way to remotely access services methods by exchanging XML messages on HTTP connections. Supporting the SOAP protocol means that:

- server side, a proxy developed as a SOAP service provides an access to SOAP clients;
- client side, a specific client `Connection` relies on HTTP and XML/SOAP format for writing and routing requests and replies.

The SOAP implementation used by JORAM is Apache's (http://ws.apache.org/soap/index.html) and works with the servlet container **Tomcat** (http://jakarta.apache.org/tomcat/). Developing a JORAM proxy as a SOAP service led to consider a specific platform configuration with a server running embedded in Tomcat's JVM (Figure 26), and acting as a router between Tomcat and the other servers of the platform.

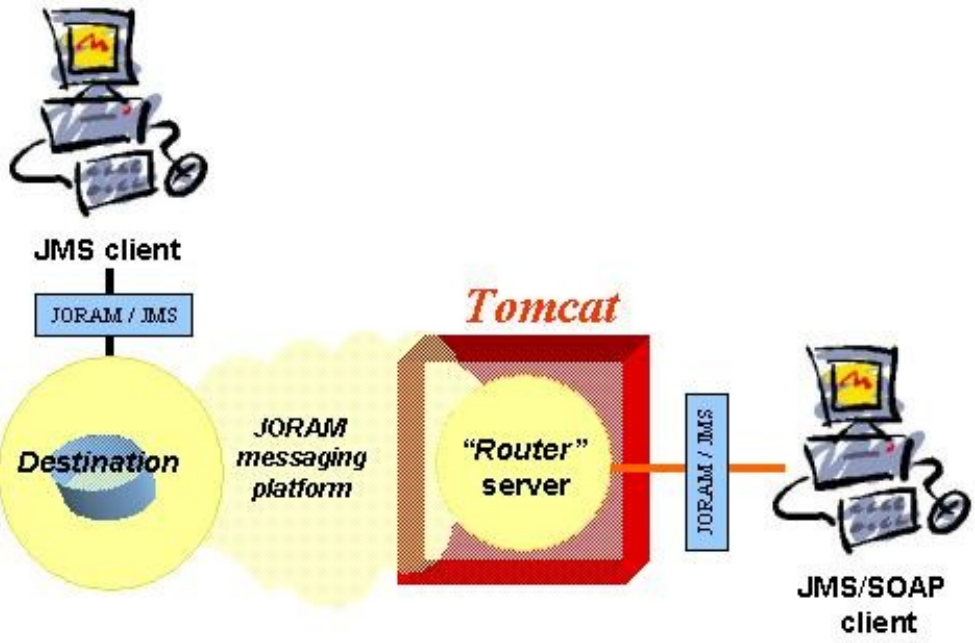


Figure 26 - A Joram platform providing a SOAP access

This document presents how to set a JORAM platform providing an access to SOAP clients, and how to administer it. The configuration of Tomcat for this specific usage is explained in the last section of the doc.

6.1. Platform configuration

6.1.1. Configuration

Though it is possible to have a configuration made of a single embedded server, we will describe the configuration shown on figure 1. It is a reasonable situation, with a server embedded in Tomcat's JVM (server **s1**), and the **s0** server providing a TCP proxy service and hosting a destination.

The following file describes this configuration. Server 0 and server 1 both provide access to external clients through their connection manager service. Server 0 also hosts a TCP proxy service listening on port 16010 and a JNDI service. Server 0 and server 1 accept administrator's connections with identification *root* – *root*.

```
<?xml version="1.0"?>
<config>
  <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="localhost">
    <network domain="D1" port="16302"/>
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args="root root"/>
  </server>
</config>
```

6.1.2. Running the platform

Starting the non embedded server

The non embedded server is started as any normal server (check administration documentation). It should be started in a transaction mode consistent with the embedded server's, that is persistent.

Starting Tomcat

Tomcat's `bin/` directory contains the start scripts. Use one of them for starting Tomcat once it is correctly configured (check the installation chapter), and once the appropriate `a3servers.xml` and `a3debug.cfg` configuration files have been put in Tomcat's `conf/` directory.

Starting the embedded server

The `org.objectweb.joram.client.jms.soap.SoapServiceStarter` utility class is provided for starting the embedded server. Its `main()` method takes for parameters the name of the host hosting Tomcat, Tomcat's HTTP port (generally 8080), the embedded server identifier and its name. It causes the instantiation of JORAM's SOAP service by Tomcat and requests the

starting of the embedded server. The embedded server successfully starts if the `a3servers.xml` platform configuration file has been put in the `conf/` directory of Tomcat.

Server 1 of the above configuration would be launched that way:

```
java      fr.dyade.aaa.joram.soap.SoapServiceStarter localhost 8080 1 S1
```

6.2. Administering

6.2.1. Introduction

Administering a platform providing a SOAP access is similar to administering a non SOAP platform. A user access must be created on the target server, and an appropriate `ConnectionFactory` object must be created so that the client will access the embedded server through the SOAP protocol.

6.2.2. Setting a user

Setting a user is done as explained in the administration document:

```
org.objectweb.joram.client.jms.admin.User user;
user = org.objectweb.joram.client.jms.admin.User.create("name", "pass", 1);
```

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.

6.2.3. SOAP ConnectionFactory object

SOAP connection factory objects are located in the `org.objectweb.joram.client.jms.soap` package.

The following creation methods are static methods:

- `SoapConnectionFactory.create(String host, int port, int timer)`:
  - creates a `ConnectionFactory` instance for accessing a server embedded in a Tomcat JVM running on a given host and listening to a given port, with a given timer value in seconds for pending SOAP connections.
- `SoapConnectionFactory.create(int timer)`:
  - creates a `ConnectionFactory` instance for accessing a server embedded in the Tomcat JVM the administrator is connected to, with a given timer value in seconds for pending SOAP connections.

```
ConnectionFactory cnxFact =
  SoapConnectionFactory.create("localhost", 8080, 60);
```

- `QueueSoapConnectionFactory.create(String host, int port, int timer)` : creates a `QueueConnectionFactory` instance for accessing a server embedded in a Tomcat JVM running on a given host and listening to a given port, with a given timer value in seconds for pending SOAP connections.
- `QueueSoapConnectionFactory.create(int timer)`:
  - creates a `QueueConnectionFactory` instance for accessing a server embedded in the Tomcat JVM the administrator is connected to, with a given timer value in seconds for pending SOAP connections.

```
javax.jms.QueueConnectionFactory cnxFact =
  admin.createQueueSoapConnectionFactory("localhost", 8080, 60);
```

- `TopicSoapConnectionFactory.create(String host, int port, int timer)`: creates a `TopicConnectionFactory` instance for accessing a server embedded in a Tomcat JVM running on a given host and listening to a given port, with a given timer value in seconds for pending SOAP connections.
- `CreateTopicoapConnectionFactory.create(int timer)`:
  - creates a `TopicConnectionFactory` instance for accessing a server embedded in the Tomcat JVM the administrator is connected to, with a given timer value in seconds for pending SOAP connections.

```
javax.jms.TopicConnectionFactory cnxFact =
  admin.createTopicSoapConnectionFactory("localhost", 8080, 60);
```

Pending SOAP connections

Contrary to a TCP connection which is opened by a connecting client (calling the `ConnectionFactory.createConnection(…)` method), and closed by the closing client (calling the `Connection.close()` method), the HTTP connection the SOAP connection is based on is opened and closed for each client – platform request / reply exchange. Thus, it is impossible from the server point of view to detect a connection failure. If a given SOAP connection is never closed by the `Connection.close()` method, its context is kept forever server side, and this could lead to **memory leaks**.

This is why a **timer** parameter is set when creating a SOAP `ConnectionFactory`. It sets the duration in seconds between two "ping" requests sent by the client JMS connection to the server. If the server does not receive any "ping" request during **timer * 2** seconds, it acts as it does when detecting a TCP connection failure: the connection's resources are cleaned, its non acknowledged messages are rolled back, temporary destinations are deleted, temporary subscriptions are removed, etc.

Setting this value to 0 means that no timer is set. Such a factory's connections never die, this is a dangerous situation.

6.2.4. SOAP administrator

The above administration tasks might be performed by a TCP administrator or a SOAP administrator.

Connecting a TCP administrator to the server 0 of the platform would look like:

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

Connecting a SOAP administrator to the server 1 of the platform looks like:

```
TopicSoapConnectionFactory cnxFact =
  TopicSoapConnectionFactory.create("localhost", 8080, 60);

AdminModule.connect(cnxFact, "root", "root");
```

6.2.5. Accessing JNDI through SOAP

SOAP administrators and clients also need to access JNDI through the SOAP protocol. This does not change the way JNDI is set in the `a3servers.xml` configuration file, but the `jndi.properties` must be modified as follows:

```
java.naming.factory.initial
  fr.dyade.aaa.jndi2.client.SoapExt_NamingContextFactory
java.naming.factory.soapservice.host localhost
java.naming.factory.soapservice.port 8080
java.naming.factory.host localhost
java.naming.factory.port 16400
```

This file says that the JNDI server is hosted on *localhost* and reachable through port *16400*, and that SOAP clients access it through the `fr.dyade.aaa.jndi2.client.SoapExt_NamingContextFactory` class, the servlet container running on *localhost* and listening on port *8080*.

6.3. Configuring Tomcat

The JORAM platform includes a SOAP proxy, accepting connection requests from SOAP clients. This proxy requires the installation and the setting of Apache servlet container, ***Tomcat***.

JORAM has been successfully tested with *tomcat-3-3*, *tomcat-4-0* and *tomcat-4-1* and *Tomcat-5.0*. This section describes how those versions should be configured.

6.3.1. Getting Tomcat

Tomcat can be downloaded from http://jakarta.apache.org/tomcat/. Documentation is available at the same location.

6.3.2. Needed resources

Libraries and configuration files must be added in the *Tomcat* environment. The jar and war files must be taken from the `lib/` directory of your JORAM installation. The configuration files must be taken from the `samples/config` directory of your JORAM installation.

6.3.3. Configuring Tomcat

Tomcat 4.1

- Files to modify in the `bin/` directory:
  - `catalina.bat` or `catalina.sh`: add the `conf/` directory in the **classpath** building.
- Libraries to add in the `common/lib` directory:
  - jakarta-regexp-1.2.jar,
  - JCup.jar,
  - joram-shared.jar,
  - joram-mom.jar,
  - ow_monolog.jar.
- Files to add in the `conf/` directory :
  - a3debug.cfg,
  - `soap_a3servers.xml` renamed `a3servers.xml`.
- File to add in the `webapps/` directory:
  - `soap.war`.

Tomcat 5.0

- Files to modify in the `bin/` directory:

  - `catalina.bat` or `catalina.sh`: add the `conf/` directory in the **classpath** building.
- Libraries to add in the `common/lib` directory:
  - jakarta-regexp-1.2.jar,
  - JCup.jar,
  - joram-shared.jar,
  - joram-mom.jar,
  - ow_monolog.jar.
- Libraries to add in the `shared/lib` directory:
  - mail.jar,
  - activation.jar.
- Files to add in the `conf/` directory :
  - a3debug.cfg,
  - `soap_a3servers.xml` renamed `a3servers.xml`.
- File to add in the `webapps/` directory:
  - `soap.war`.

6.4. Running the Soap Sample

The SOAP samples are a simple message producer and consumer pair. Their goal is to show that JORAM successfully integrates and supports SOAP as an other communication protocol than TCP. Running these samples requires to have installed and configured Apache's ***Tomcat*** servlet container (check above).

The following picture illustrates the platform architecture. Server 1 is embedded in *Tomcat*'s JVM and holds the proxy for the SOAP clients. Server 0 runs in an other JVM and hosts the destinations the clients actually interact with.

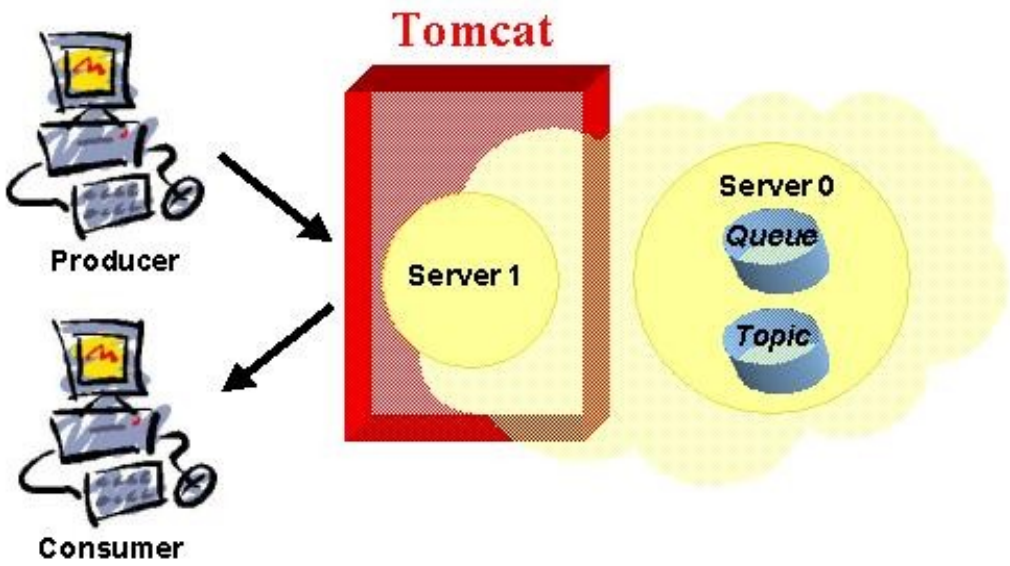


Figure 27 - SOAP sample configuration

Running the demo:

- Starting a correctly configured ***Tomcat*** instance:

  - in `tomcat/bin` remove any remaining persistence directory (`s1/`) and launch *`startup.sh`* or *`startup.bat`*.
- Launching the server 0 of the configuration:
  - Under `samples/src/joram:` *`ant soap_server`*
- Launching the server 1, embedded in Tomcat's JVM:
  - Under `samples/src/joram:` *`ant soap_start`*
- Running the administration:
  - Under `samples/src/joram:` *`ant soap_admin`*
- Launching the consumer:
  - Under `samples/src/joram:` *`ant soap_consumer`*
- Launching the producer:
  - Under `samples/src/joram:` *`ant soap_producer`*

6.5. Running kJoram sample

6.5.1. Environment

Let's consider the JORAM distribution is installed in a `JORAM_HOME/` directory, whereas the kJORAM distribution is in a `kJORAM_HOME/` directory. Both distributions have been compiled if needed, and ***Tomcat*** has been configured.

Applications running in a **J2ME** environment are able to connect and use JORAM messaging platform functionalities. For doing this, they use specific JORAM client libraries and resources, which are available in the JORAM CVS distribution or by downloading the kJORAM packages.

The provided sample allows to run an application on a Pocket PC and make it communicate through a JORAM platform with an other Pocket PC application, or a "normal" Java application. The platform used for this demonstration is exactly similar to the one used for the SOAP sample (see Figure 28 and section 6.4. Running the Soap Sample).

The kJORAM samples directory provides utility files valid for **J9** users on **Pocket PC**s. The configuration information provided in this section is only valid for those users. **Launching the samples on other devices requires some specific configuration users should be aware of.**

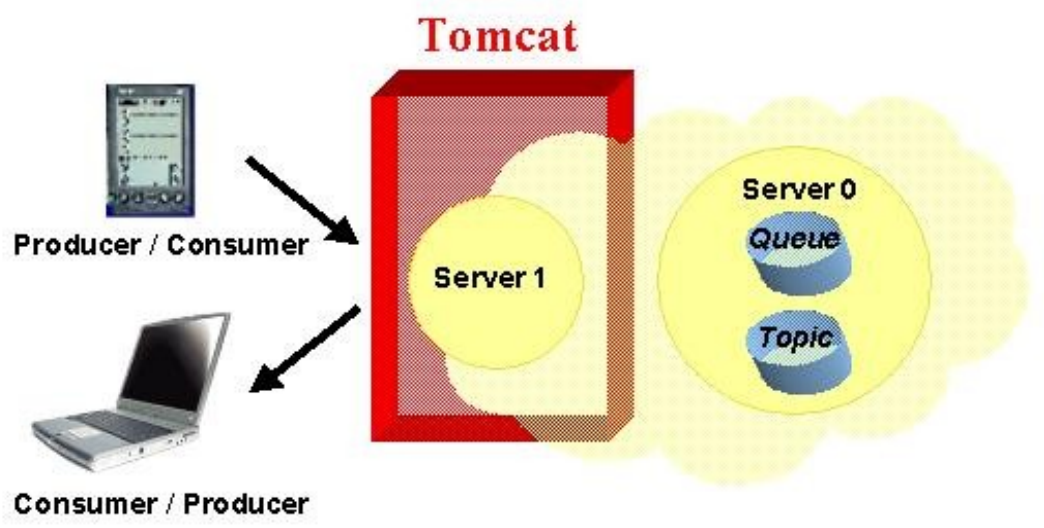


Figure 28 - kSOAP sample configuration

6.5.2. Compiling the samples files

The samples source files in `kJORAM_HOME/samples/src/kjoram/classic` need first to be edited. In the `KConsumer` and `KProducer` classes, the **X.X.X.X** fields must be replaced by the actual IP number of the machine which will host the JORAM server.

Once this is done, the samples in the kJORAM packages and in the CVS distribution need to be compiled. Under the `samples/src/kjoram` directory, simply type:

```
ant clean compile
```

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

```
ant clean
```

As kJORAM sources, the samples are compiled with target 1.1 for J2ME compliance;

6.5.3. Installing the samples on the Pocket PC

The following files and libraries must be transferred on the Pocket PC:

- `kjoram.jar` and `kxml.jar` from `kJORAM_HOME/lib` to "`My Documents/WSDD`";
- the samples classes from `kJORAM_HOME/samples/classes/kjoram` to "`My Documents/WSDD`";
- the "`.jad`" files from `kJORAM_HOME/samples/src/kjoram/classic` to "`My Documents/WSDD`";
- the "`.lnk`" files from `kJORAM_HOME/samples/src/kjoram/classic` to "`Windows/Start Menu`".

6.5.4. Starting and administering the JORAM platform

On the server's host:

- Starting a correctly configured **Tomcat** instance:

        - in `tomcat/bin` remove any remaining persistence directory (`s1/`) and launch
        - *`startup.sh`* or *`startup.bat`*.
    - Launching the server 0 of the configuration:
        - Under `JORAM_HOME/samples/src/joram`: *`ant soap_server`*
    - Launching the server 1, embedded in Tomcat's JVM:
        - Under `JORAM_HOME/samples/src/joram`: *`ant soap_start`*
    - Running the administration:
        - Under `JORAM_HOME/samples/src/joram`: *`ant soap_admin`*

Launching the demo between a Pocket PC and a PC

- First start the consumer, either on the Pocket PC, or on the PC:
    - on the Pocket PC, in the **Start Menu** select **kconsumer**
    - on the PC, under `JORAM_HOME/samples/src/joram`, type: *`ant soap_consumer`*
- Then start the producer, on the device which is not hosting the consumer:
    - on the Pocket PC, in the **Start Menu** select **kproducer**
    - on the PC, under `JORAM_HOME/samples/src/joram`, type: *`ant soap_producer`*

Launching the demo between two Pocket PCs

- First start the consumer, on Pocket PC 1 for example:
    - on the Pocket PC, in the **Start Menu** select **kconsumer**
- Then start the producer, on Pocket PC 2:
    - on the Pocket PC, in the **Start Menu** select **kproducer**

7. Using a colocated server

7.1. 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 colocated or not servers, it can eventually be reach by other client through the TCP protocol.

7.2. 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.

7.3. 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);
```

7.4. Connect to the collocated server

7.4.1. 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`.

7.4.2. 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");
```

7.5. 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();
```

7.6. Start the embedding Java application

You must ensure that the classpath contains:

- the jar files expected by a Joram server: joram-mom.jar, Jcup.jar, etc.
- the directory where the a3servers.xml configuration file is located.

8. High-Availability

8.1. Platform Configuration

A HA Joram server is launched on a cluster of ScalAgent servers. So the configuration of a HA Joram server implies to configure a clustered ScalAgent server.

8.1.1. Clustered ScalAgent server configuration

A clustered ScalAgent server is defined by the element “cluster”. A cluster owns an identifier and a name defined by the attributes “id” and “name” (exactly like a standard server). Two properties must be defined:

- “Engine” must be set to “fr.dyade.aaa.agent.HATransactionEngine” which is the class name of the engine that provides high availability (*set by default in a next version* ?).
- “nbClusterExpected” defines the number of replicas that must be connected to the group communication channel used before this replica starts

```
<?xml version="1.0"?>
<config>
  <domain name="D1"/>
  <cluster id="0" name="s0">
    <property name="Engine"
              value="fr.dyade.aaa.agent.HATransactionEngine"/>
    <property name="nbClusterExpected" value="3"/>
```

For each replica an element “server” must be added. The attribute “id” defines the identifier of the replica inside the cluster. The attribute “hostname” gives the address of the host where the replica is running. A server replica is defined exactly like a standard server. For example this one owns a network and two services. The network is used by the replica to communicate with external agent servers, i.e. servers located outside of the cluster and not replicas.

```
    <server id="0" hostname="localhost">
      <network domain="D1" port="16301"/>
    </server>
```

Two other replicas are defined in the clustered ScalAgent server s0. Notice that they define their network with a different port value because they work on the same host (localhost).

```
    <server id="1" hostname="localhost">
      <network domain="D1" port="16302"/>
    </server>
    <server id="2" hostname="localhost">
      <network domain="D1" port="16303"/>
    </server>
  </cluster>
</config>
```

8.1.2. Group communication

There are two properties to configure the address and the port of the IP multicast used by JGroups: *JGroups.MCastAddr* and *JGroups.McastPort*.

The group communication semantic is not configurable as it requires to understand exactly how the high availability works. However some higher level configuration handles will be given if needed.

8.1.3. Joram server configuration

A Joram server is started by a ScalAgent service called “ConnectionManager” (see Joram documentation).

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

This service must be added to all the replicas. Some more services can be added to each replicas according to the entry points needed by the clients: Tcp, Soap and collocated.

TCP entry point

The service “TcpProxyService” starts a TCP entry point to the Joram server.

```
<service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
         args="2560"/>
```

Soap entry point

Not yet available (in HA mode)

Collocated entry point

The service “HALocalConnection” provides a collocated entry point to the replica which is the master. On the other replicas, this service blocks the collocated connections until the local replica becomes the master.

```
<serviceclass="org.objectweb.joram.client.jms.ha.local.HALocalConnection"/>
```

Configuration example

The following configuration defines a HA Joram server “s0” that provides two entry points: Tcp and collocated. It is replicated three times on the same host (localhost).

```
<config>
 <domain name="D1"/>
 <cluster id="0" name="s0">
  <server id="0" 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="2560"/>
    <service class="org.objectweb.joram.client.jms.ha.local.HALocalConnection"/>
  </server>
  <server id="1" hostname="localhost">
   <network domain="D1" port="16302"/>
   <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
            args="root root"/>
   <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
            args="2561"/>
   <service class="org.objectweb.joram.client.jms.ha.local.HALocalConnection"/>
  </server>
  <server id="2" hostname="localhost">
   <network domain="D1" port="16303"/>
   <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
            args="root root"/>
   <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
            args="2562"/>
   <service class="org.objectweb.joram.client.jms.ha.local.HALocalConnection"/>
  </server>
 </cluster>
</config>
```

8.2. Platform startup

A server replica is started exactly like a standard agent server. The arguments are:

- server id: identifier of the clustered server (not the identifier of the replica)
- storage directory path: path of the directory where the persistency files are created
- replica id: identifier of the replica inside the clustered server

java fr.dyade.aaa.agent.AgentServer <server id> <storage directory path> <replica id>

The replicas can be started in any order. The first master replica elected is the first replica connected to the group communication channel.

8.2.1. Host clock synchronization

The clocks of the hosts from the cluster should be quite synchronized if the “heart beat” feature is used.

8.3. Programming an external HA Joram client

The only available protocol for an external client is TCP. *The protocol Soap is not yet available.*

8.3.1. Joram administration

Create a connection factory

There are several types of HA TCP connection factory depending on the destination reached (generic, queue or topic) and the transactional mode (XA or none):

- <XA|none><Queue|Topic|none>HATcpConnectionFactory

These connection factories belong to the package *org.objectweb.joram.client.jms.ha.tcp*. To instantiate a connection factory, call the static method *create* defined in the class of the connection factory to instantiate. A parameter must be given specifying the URL of the HA Joram server. This URL defines the list of the TCP entry points owned by each replicas of the HA Joram server.

For example to create a generic (neither Queue nor Topic, unified JMS1.1) non-transactional (no XA) connection factory you need to call:

```
ConnectionFactory cf;

cf = org.objectweb.joram.client.jms.ha.tcp.HATcpConnectionFactory.create(
      "hajoram://localhost:2560,localhost:2561");
```

Configure the connection factory

It is also necessary to set the *connectionTimer* of the connection according to the switching time required by the Joram server cluster, i.e. the delay needed to elect a new master replica and to activate it. If the *connectionTimer* is too short, then the external client may not be able to connect to the new master replica. So the failure of a Joram server replica may break the HA connection between the external client and the HA Joram server.

```
((org.objectweb.joram.client.jms.ConnectionFactory)cf).
         GetParameters().connectingTimer = 30;
```

8.3.2. JMS programming

The JMS programming is completely standard. A HA server behaves exactly the same as a standard server. However there is a slight difference about the naming of the destination. Just like with a standard Joram, you can choose to use JNDI or the internal Joram naming. But if you use JNDI then you have to configure a replicated JNDI server. This operation is available but not documented here as it is simpler to use the Joram internal naming.

8.4. Programming a collocated Joram client

8.4.1. Joram administration

Create a connection factory

There are several types of HA local connection factory depending on the destination reached (generic, queue or topic) and the transactional mode (XA or none):

- <XA|none><Queue|Topic|none>HALocalConnectionFactory

These connection factories belong to the package *org.objectweb.joram.client.jms.ha.local*. To instantiate a connection factory, call the static method *create* defined in the class of the connection factory to instantiate.

For example to create a generic (not Queue either Topic) non-transactional (no XA) connection factory you need to call:

```
ConnectionFactory cf;

cf = org.objectweb.joram.client.jms.ha.local.HALocalConnectionFactory.create();
```

8.4.2. Collocated client process

A collocated client must be instantiated inside the same JVM processes as the Joram server replicas. It also must use the same class loader (or a child) as the server's class loader. A simple way to do this is to write a main class that starts an agent server replica and then creates one or more HA local connections.

In the following code example, an unified non-transactional (not XA) connection is opened. It is important to notice that the connection creation blocks until the server replica becomes master.

```
public static void main(String[] args) throws Exception {
 ConnectionFactory cf;

 AgentServer.init(args);
 AgentServer.start();
 cf= org.objectweb.joram.client.jms.ha.local.HALocalConnectionFactory().create();

 // The next line blocks until the server replica becomes master
 Connection cnx = cf.createConnection("root", "root");
```

8.4.3. JMS programming

The JMS programming is completely standard. A HA server behaves exactly the same as a standard server. However there is a slight difference about the naming of the destination. Just like with a standard Joram, you can choose to use JNDI or the internal Joram naming. But if you use JNDI then you have to configure a replicated JNDI server. This operation is available but not documented here as it is simpler to use the Joram internal naming.

9. JNDI

9.1. Overview

The goals of the ScalAgent JNDI are to distribute and replicate the naming contexts among various servers in order to provide load balancing and fail over.

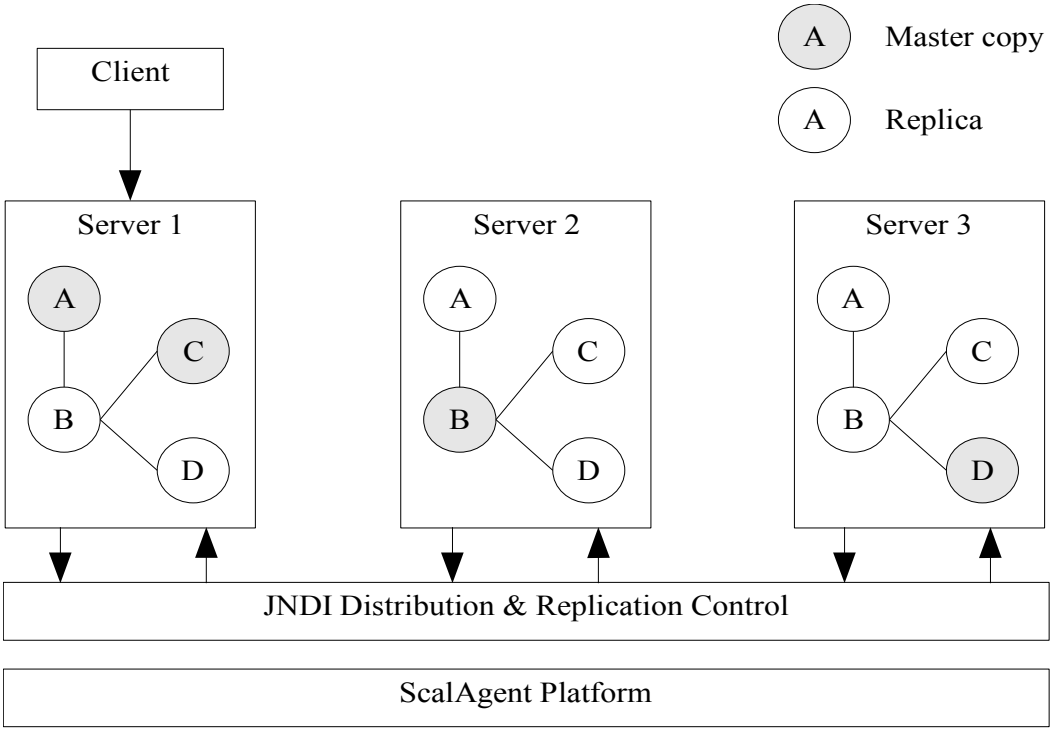


Figure 29 - JNDI replication

The chosen replication follows the master-copy scheme. The master copies are distributed among the servers. The following figure gives an overview of the ScalAgent JNDI architecture. The client sends requests to a server that may communicate with one or more servers of the JNDI configuration. This communication is handled by a control layer responsible for distribution and replication built on top of the ScalAgent platform.

9.2. Replication

9.2.1. Master ownership strategy

The naming contexts are replicated with a master copy replication scheme. Each naming context has one unique master server. Only the master can update the primary copy of the naming context. All other replicas are read-only. Other servers wanting to update the context (bind, rebind, create/destroySubcontext) request the master to do the update.

9.2.2. Lazy propagation strategy

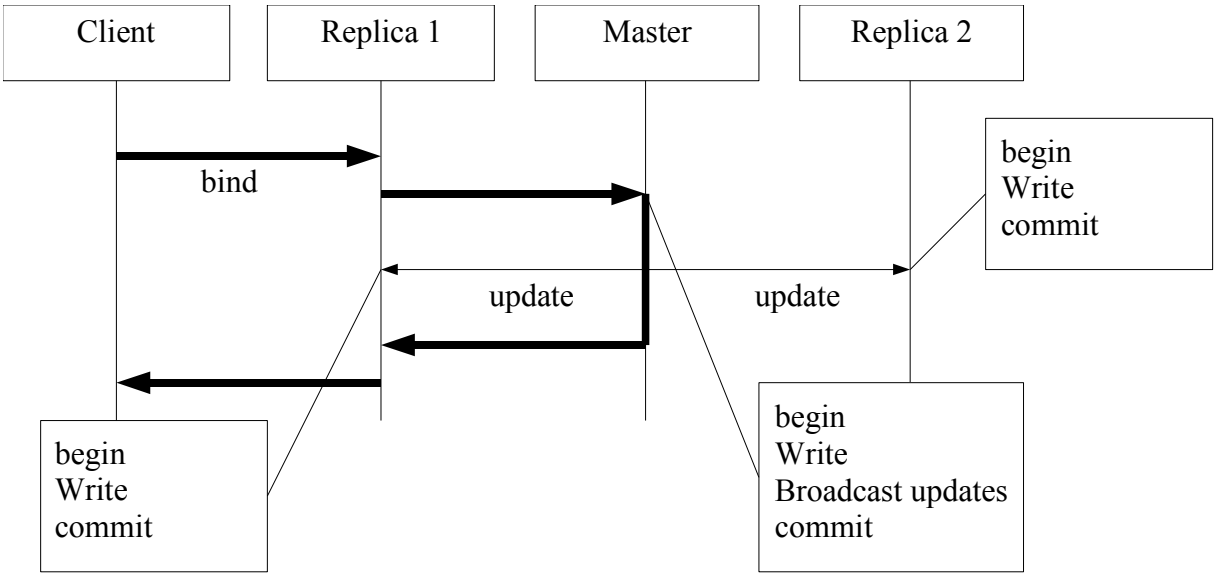


Figure 30 - Lazy propagation

All updates emanate from the master copy of a naming context. The updates are propagated in a lazy way. The master copy is updated in a local transaction that updates the copy and broadcasts the updating notifications to the replicas. Those updates are asynchronously propagated and performed in a separate transaction for each server.

9.2.3. Replicas synchronization

Once a client has done several write requests, he may see different naming data depending on the requested server because updates are asynchronous. The resulting errors are: name not found and stale (out of date) data.

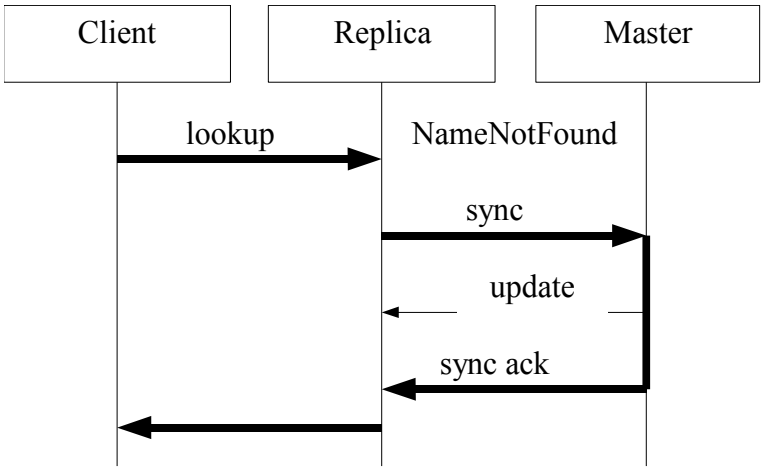


Figure 31 - Replicas synchronization

The name not found error is handled by synchronizing the replica with the master copy of the context. Once synchronized, the request is retried. If the name is still not found, then the client gets the error.

The stale data error cannot be detected. It is inherent to the lazy propagation strategy.

However an explicit synchronization operation for a specific naming context (as described above) could be provided in a JNDI extension interface, enabling the client to get the latest naming data. This extension is still not available.

9.3. Distribution of the naming servers

The naming servers are completely interconnected through the ScalAgent platform. A server owns two entry points: a TCP entry point through which client requests are transmitted and an Agent entry point used to send and receive notifications from the other servers (request forwards, replica updates …).

The following figure displays a JNDI distributed configuration that includes 4 servers.

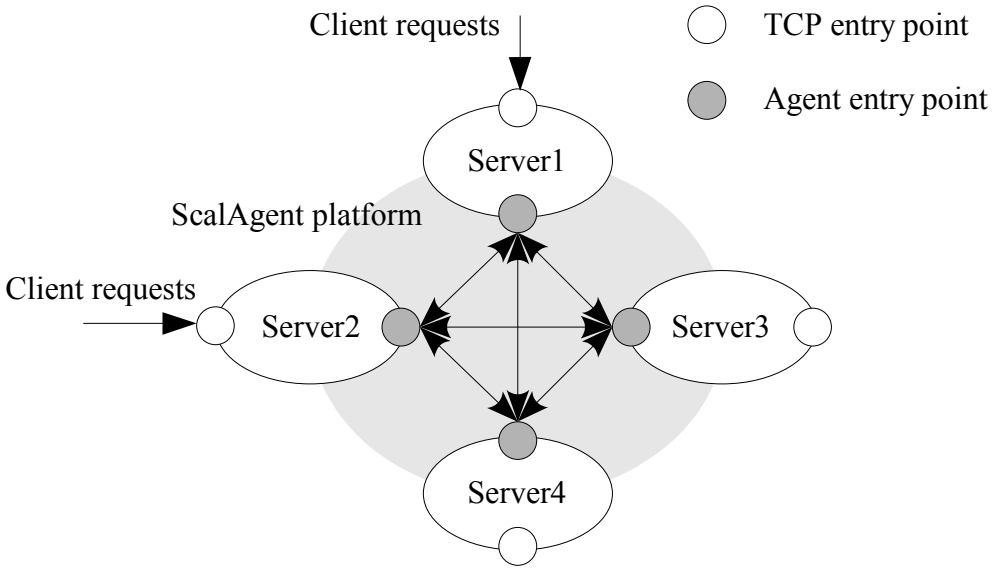


Figure 32 - Distributed configuration

A new naming server is added into a JNDI configuration by assigning it one or more references of other servers already included into the configuration. When a naming server discovers another server, it sends to it an initialization notification containing its master copies and a list of the known servers. In this way a server discovers all the servers of a JNDI configuration.

The following figure displays a sequence diagram of the entrance of a new server into a JNDI configuration. The new server S3 initially knows S1, so it sends to it an initialization notification containing its master copies contexts[S3]. S1 replies with its contexts[S1] and the reference of the server S2. S3 stores the contexts[S1] as replicas, registers the reference of S2 and sends to it an initialization notification.

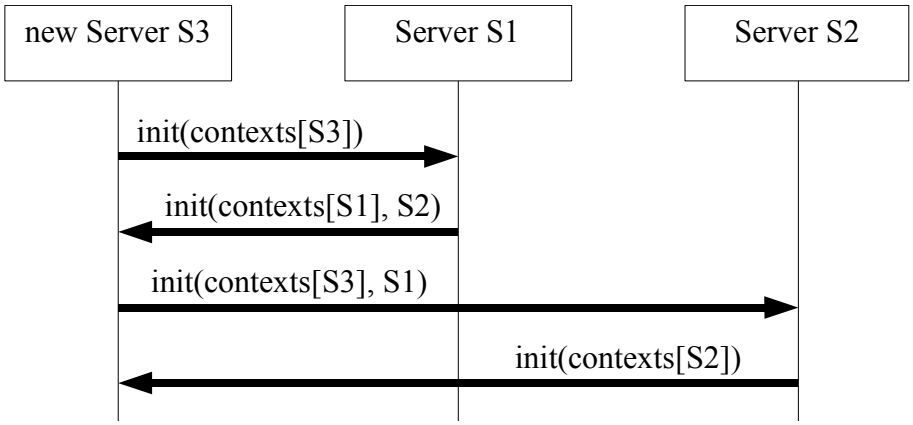


Figure 33 - Adding a new server

A JNDI configuration is wholly replicated. All the servers contain the same (differed) naming data. Some replication groups could be defined in order to reduce the number of updates.

The servers are implemented with the ScalAgent programming model which is asynchronous and reliable. This model is particularly well adapted for the loose coupling of a lazy replication strategy. Moreover this model simply controls the isolation by serializing the operations performed by a server.

9.4. Distribution of the naming contexts

9.4.1. Context creation

When a server (called “local”) creates a subcontext of a naming context which master copy is owned by a remote server, it must ask the remote server to create the subcontext. But this subcontext is owned by the server that initiates the creation. So the master copy of this new context belongs to the local server.

9.4.2. Context name resolution

A naming context is accessed by its name which is a path leading from the root naming context to the final name of the context.

Names are not resolved from their path but directly using an index that returns the identifier of the context from its name. In this way a server can handle JNDI requests which path cannot be resolved because of missing intermediate contexts (still not initialized), but which final context locally exists.

This feature is useful if a naming server is started whereas there is a network partition that prevents it from getting the root context. But it can receive some contexts from local surrounding servers and immediately begin to reply to its clients.

9.5. Configuration

A ScalAgent JNDI configuration gathers several JNDI servers. Each server is defined as a ScalAgent service. The class name is *DistributedJndiServer* from the package *fr.dyade.aaa.jndi2.distributed*. The argument line has the following syntax:

```
<listening port> <root owner id> [<server id> …]
```

- listening port: the port used by the JNDI server to listen to clients requests

- root owner id: the identifier of the JNDI server that owns the root naming context
- server id: the identifier of a JNDI server
    - The listening port and the root owner identifier are mandatory. The list of server identifiers is optional.

A server needs to know the identifier of the root owner in order to compare this identifier with its own identifier. If it is the same, it means that the server is the root owner and must initially create the root naming context. If it is not the same, it means that the server is not the root owner and must register itself with the root owner in order to receive the copies of the naming data owned by the root owner.

The list of JNDI server identifiers is optional. It is useful if the root owner server is not available. In this case the naming data owned by those servers may be accessed (read and write) even if the root server is not available.

The following configuration includes three JNDI servers:

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

<config>
  <domain name="D0" network="fr.dyade.aaa.agent.SingleCnxNetwork"/>
  <server id="0" name="s0" hostname="localhost">
    <network domain="D0" port="27300"/>
    <service class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
args="16400"/>
  </server>
  <server id="1" name="s1" hostname="localhost">
    <network domain="D0" port="27301"/>
    <service class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
             args="16401 0"/>
  </server>
  <server id="2" name="s2" hostname="localhost">
    <network domain="D0" port="27302"/>
    <service class="fr.dyade.aaa.jndi2.distributed.DistributedJndiServer"
             args="16402 0 1"/>
  </server>
</config>
```

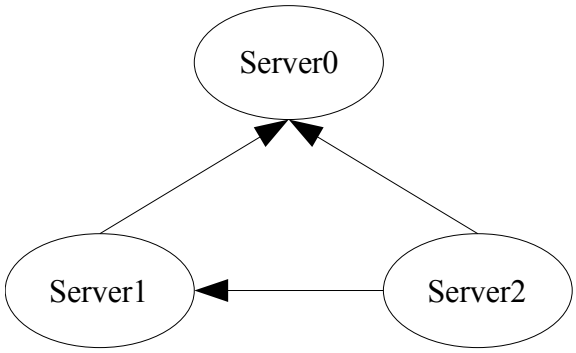


Figure 34 - A 3 servers configuration

The server 0 is the root owner (note that it could have another identifier), the server 1 only knows the root owner server and the server 2 knows both 0 and 1. In this way, if the server 0 is not available when the server 2 starts, it can get the naming data owned by the server 1 anyway.

Let's say that the servers 0 and 1 are first started. The client 0 asks the server 0 to create a subcontext A in the root context. Then the client 1 creates the context B in A. Notice that if the client 1 creates B before the server 1 received the naming data from the server 0, the request is blocked. When the server 1 receives the copy of A, it retries all the requests waiting for this context (e.g. the creation of B).

If the server 2 is started when the server 0 is unavailable, it can't receive the naming context A. So the client 2 can't do any requests (read and write) in the context A. But the server 2 knows the server 1 so its receives the naming data owned by server 1 (context B). So the client 2 can do a request in B, e.g. create a sub context C.

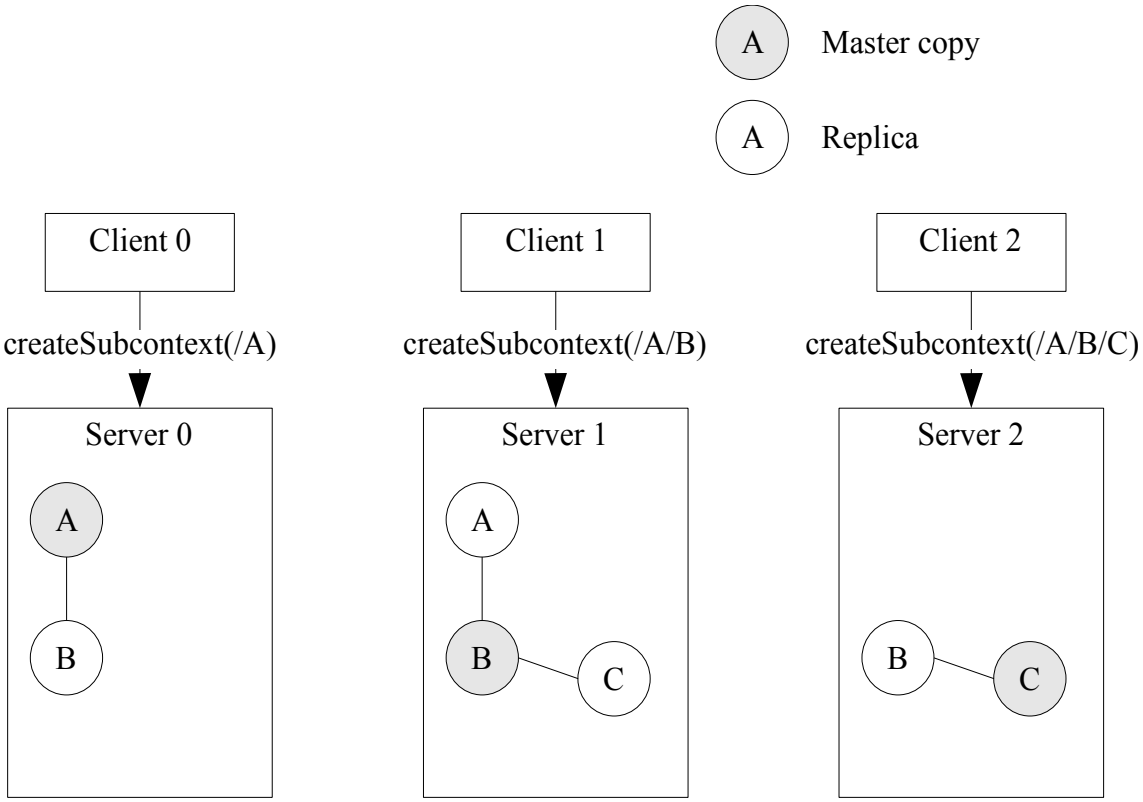


Figure 35 - The 3 servers configuration replicas

The following figure shows the final states of the JNDI servers once the previous scenario has been executed. The server 2 will get the naming data from server 0 as soon as it is available. At this time the server 0 will also get the naming data owned by server 2 (context C).

10.Using JMS Bridge

10.1. Bridging Joram 4.1 and XMQ

The JORAM 4.0 release has introduced a bridge feature. This 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 the JORAM and *XMQ* heterogeneous platforms is provided by a specifically configured JORAM destination (a queue or a topic), connected to a *XMQ* destination as a standard (JMS 1.0.2b or JMS 1.1) client application (as illustrated by Figure 36).

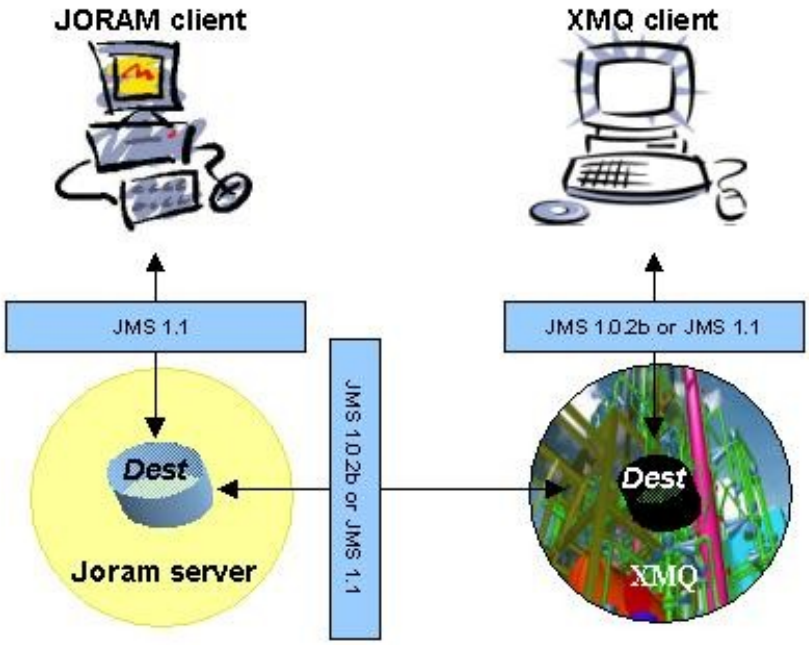


Figure 36 - A JORAM client communicating with a XMQ client

When the JORAM client sends a message on the JORAM bridge destination, the message is actually forwarded to the *XMQ* destination. If the JORAM client requests a message from the JORAM bridge destination, the request is forwarded to the *XMQ* destination. The *XMQ* destination answers by returning a JMS message, which is stored in the JORAM bridge destination and eventually consumed by the JORAM client.

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.

This document presents how the JORAM bridge feature works, how to use it, and finally shows sample codes for bridging JORAM with IBM's WebSphere MQ platform.

10.2. Concepts and implementation

As said in the introduction, the JORAM JMS bridge feature is provided by specialized JORAM destinations. The communication between a JORAM client and a JORAM bridge destination is standard. It is the processing of the client requests within the bridge destination that is specialized.

A bridge destination has 3 main roles:

- as a JORAM destination, it receives messages sent by JORAM JMS producers, and requests sent by JORAM JMS consumers;
- as a *XMQ* JMS producer, it forwards the received messages to the *XMQ* destination; as a *XMQ* JMS consumer, it requests messages from the *XMQ* destination;
- as a *XMQ* JMS consumer, it consumes messages coming from the *XMQ* destination; as a JORAM destination, it delivers those messages to the JORAM JMS consumers.

The JMS resources involved in the JORAM client – *XMQ* client communication are illustrated by the next picture.

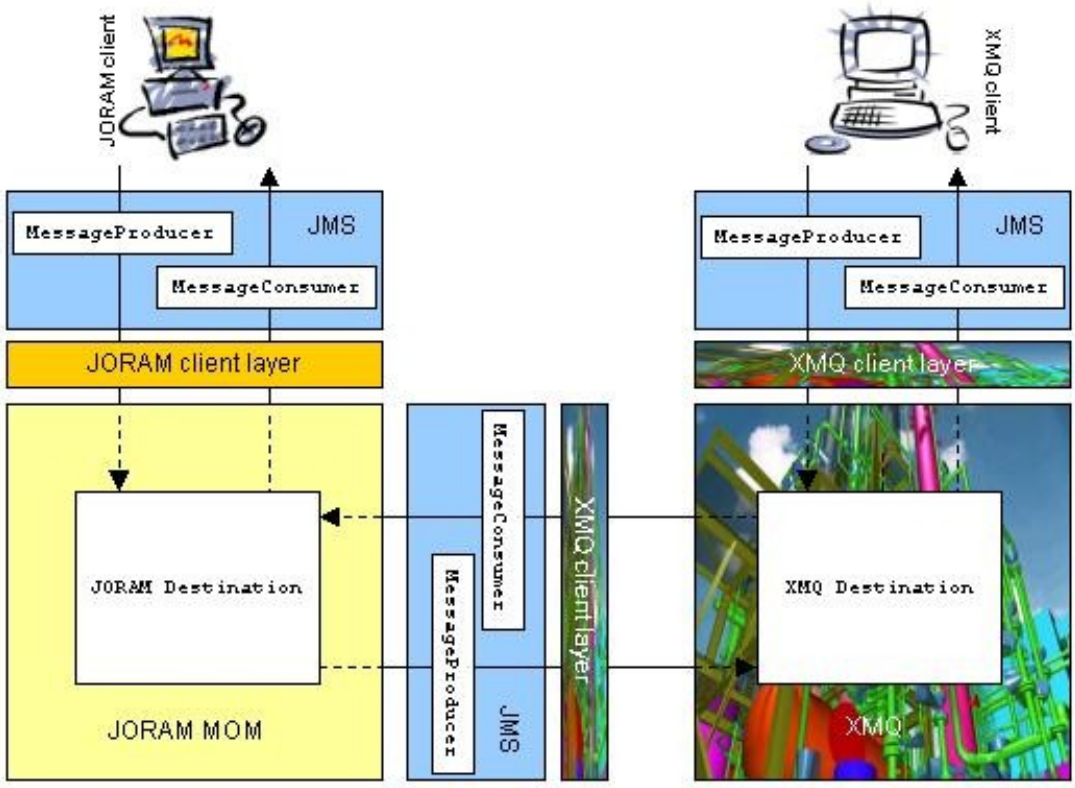


Figure 37 - Bridge communication diagram

10.2.1. Message exchange

Communication modes

The JORAM bridge destination may either be a queue or a topic. And in both cases it may either communicate with a foreign queue or a foreign topic. A JORAM client application may then interact with a *XMQ* destination normally incompatible with the communication mode. For example, a JORAM client can interact with a *XMQ* queue in a Publish/Subscribe way.

Message sending

A message sent by a JORAM client application first reaches the bridge destination. It is then forwarded to the *XMQ* destination. Once successfully delivered to the *XMQ* platform, it is removed from the bridge destination. While the message was hosted by the JORAM bridge destination (between its arrival and its delivery to the foreign platform), it has never been available for delivery to a JORAM client. Even if the bridge destination had pending consumers requests, they wouldn't have been answered with this message. Messages arriving on a bridge destination and coming from a JORAM client application are only deliverable to a foreign JMS destination.

Message consumption

A consuming request sent by a JORAM client application first reaches the bridge destination. It is then forwarded to the *XMQ* destination. When the *XMQ* target destination is a topic, the JORAM bridge destination, when receiving the first consuming request, durably subscribes to the *XMQ* topic. This for not loosing messages if the JMS connection between the bridge destination and the *XMQ* platform fails (connection failures handling is addressed section 2.4). Messages delivered by the XMQ platform are consumed by the JORAM bridge destination. At that point they are converted into JORAM JMS messages (if the conversion fails, the delivery is rolled back, see section 2.2). Finally, those messages are consumed by the JORAM client applications.

Queue browsing

The JMS API allows a client application to browse the messages available for delivery on a queue. When requesting to browse a JORAM queue bridge, the "browse" request is forwarded to the *XMQ* destination if it is a queue. It eventually returns an enumeration of messages, converted into JORAM JMS messages. The enumeration is finally forwarded to the original requester. If the bridge destination is connected to a *XMQ topic*, the request is directly answered by an empty enumeration.

10.2.2. 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.

Figure 38 shows a first scenario of message rollback. In fact, it is the only possible case of message denying between the JORAM bridge destination and the *XMQ* platform. It 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.

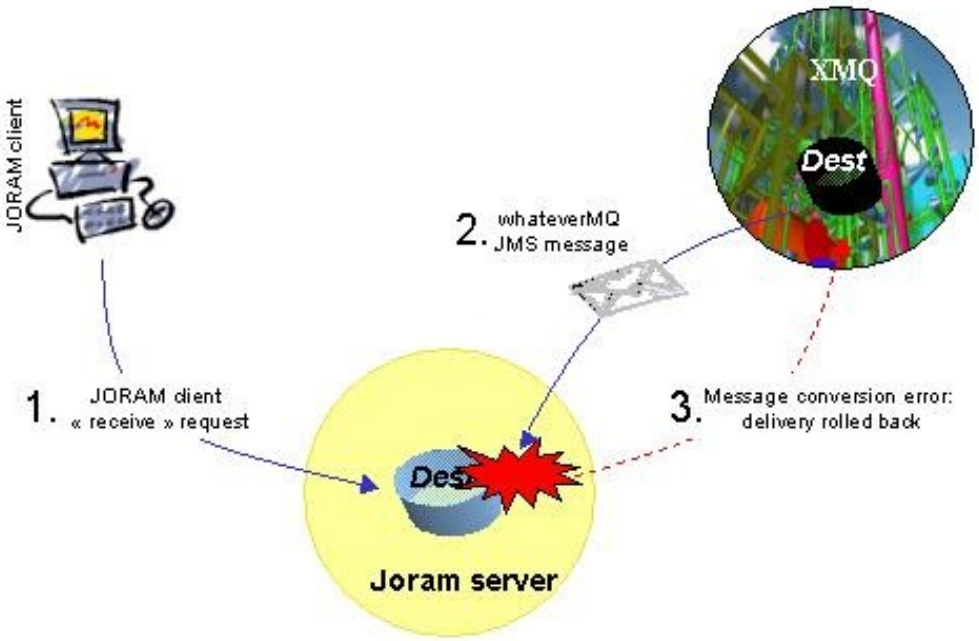


Figure 38 - Delivery rolled back by the JORAM bridge destination

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 yes, *XMQ* should be configured for doing 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.

Figure 39 shows the second possible scenario involving a rollback or message denying. It occurs when the JORAM client application, which finally consumes the message, either fails before acknowledging the message or explicitly rolls back its session.

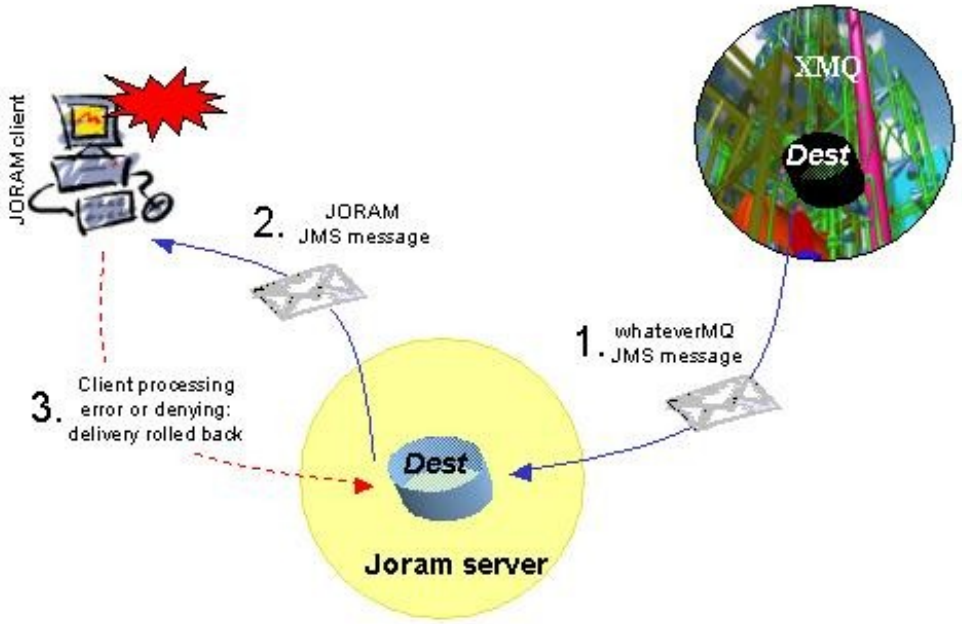


Figure 39 - Delivery rolled back by the JORAM client application

The consequence of this message denying is the same 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.

10.2.3. Message selection

As for acknowledgements, message selection is handled separately between a JORAM client and the JORAM bridge 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 for filtering the messages on the *XMQ* destination is set at administration time, when configuring the JORAM bridge destination. It can not be changed once the bridge destination has been created. The selector used by the JORAM client is set as a standard selector, when creating the `MessageConsumer` instance.

Figure 40 shows how those selectors operate. The bridge's selector selects messages with a given property value above 2. As a consequence, only the messages which property value is above 2 will actually be transferred from *XMQ* to JORAM.

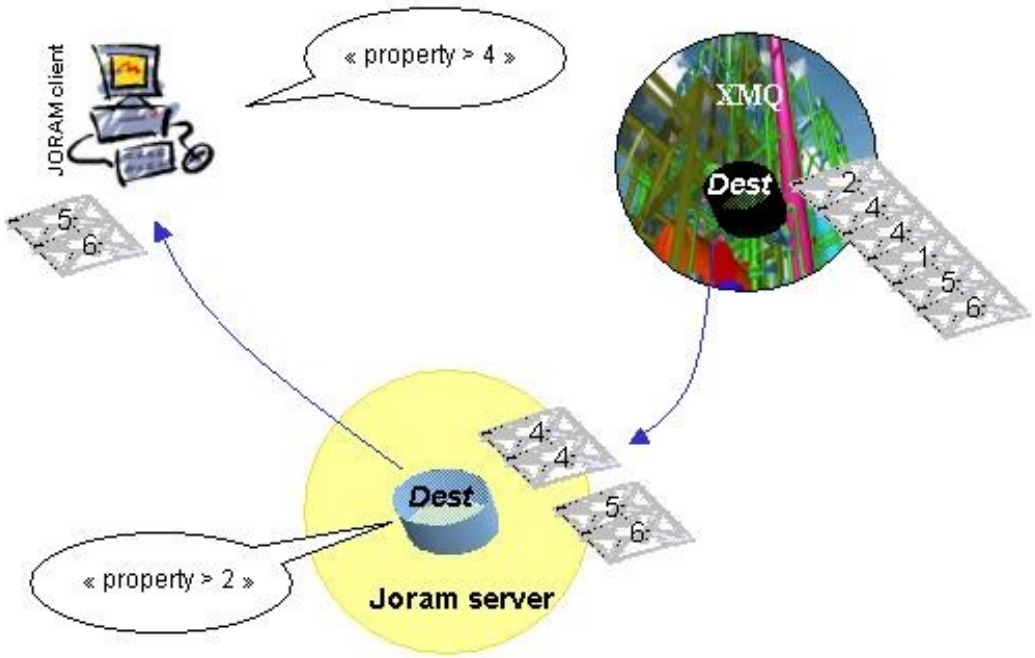


Figure 40 - How selectors operate

The JORAM client application selects messages with the same property value above 4. It then consumes the corresponding messages available on the bridge destination. Other clients may consume the other messages by setting different selectors.

10.2.4. 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.

10.3. User manual

A JORAM bridge client is a fully standard JMS client which has no visibility of the foreign *XMQ* platform. Of course achieving this requires to properly configure and administer the JORAM and *XMQ* platforms.

10.3.1. Configuring the foreign platform

Prerequisites

The prerequisites on the foreign JMS server to bridge with JORAM are as follows:

- full implementation of the JMS 1.0.2b or JMS 1.1 API;
- particularly, the `javax.jms.ExceptionListener` must be supported.

Configuration and administration

The foreign *XMQ* platform should be appropriately configured for hosting a destination, and authorizing the connection of an external client.

JMS administered objects (`javax.jms.ConnectionFactory` and `javax.jms.Destination`) that will be used by the JORAM bridge to connect to the *XMQ* platform and access the right destination should be created and bound to a JNDI compliant naming server.

10.3.2. Configuring JORAM

Configuration

JORAM's server classpath must provide access to the following libraries:

- all *XMQ* client jars;
- a `jms.jar` library compatible with the *XMQ* JMS implementation;
- the client jars of the used JNDI server;
- a `jndi.jar` library compatible with the naming server used for binding the *XMQ* administered objects.

Administration

Creating a JORAM bridge destination is done through the `org.objectweb.joram.client.jms.Queue.create` and `org.objectweb.joram.client.jms.Topic.create` static methods.

The parameters of these methods are:

- `int serverId`: identifier of the server on which deploying the destination;
- `String className`: name of the bridge's class:
  - "org.objectweb.joram.mom.dest.BridgeQueue" for a queue,
  - "org.objectweb.joram.mom.dest.BridgeTopic" for a topic;
- `java.util.Properties prop`: properties configuring the bridge destination.

The following properties are required for setting the bridge destination:

- "connectionFactoryName": JNDI name used to bound the *XMQ* JMS `ConnectionFactory` object;
- "destinationName": JNDI name used to bound the *XMQ* JMS `Destination` object;
- "jmsMode": says what is the type of the *XMQ* `ConnectionFactory` instance bound to JNDI:
  - "ptp" for a `QueueConnectionFactory`,
  - "pubsub" for a `TopicConnectionFactory`,
  - "unified" for a `ConnectionFactory`.

The following properties are optional:

- "userName": user identification that should be used by the bridge destination for opening a connection to *XMQ*; if not provided, the connection will be opened with no identification.
- "password": user password that should be used by the bridge destination for opening a connection to *XMQ*; if not provided, the connection will be opened with no identification.
- "jndiFactory": name of the JNDI initial context factory class (for example "fr.dyade.aaa.jndi2.client.NamingContextFactory"); if not provided, it is expected that environment variables are set or that a `jndi.properties` file is accessible through the server's classpath.
- "jndiUrl": URL of the JNDI server (for example "joram://localhost:16400"); if not provided, it is expected that environment variables are set or that a `jndi.properties` file is accessible through the server's classpath.
- "clientId": provided if *XMQ* requires the setting of such an identifier on its client connection.
- "selector": selector expression used for filtering messages on the *XMQ* destination.

10.3.3. 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. starting a JNDI server;
2. creating the foreign JMS administered objects, binding them to the JNDI server;
3. starting and administering JORAM.

If the JNDI server used is hosted by the JORAM server, then the JORAM server is the first thing to start (in order to start the JNDI service). Then the XMQ JMS administered objects should be created and bound. Finally, the JORAM server should be administered, and particularly the bridge destination should be created.

10.3.4. Failures

If one of the mandatory configuration properties is missing, the bridge destination creation will fail (the static `create` method will throw an `AdminException`).

If the provided properties do not allow to successfully set a link with the foreign JMS server, the bridge won't be usable. But as the initialisation process is asynchronous, the administrator won't get any information about this failure.

Simply, when sending or requesting messages to/from this bridge, ERROR messages will be logged. The possible error cases are:

- JNDI server not reachable (due to invalid "jndiFactory" or "jndiUrl" values);
- the administered objects could not be retrieved from the JNDI server (invalid "cnxFactoryName" or "destinationName" values);
- missing *XMQ* client libraries in JORAM server's classpath, preventing the successful re-construction of the retrieved administered objects;
- communication mode ("jmsMode" value) not compatible with the retrieved administered objects (`TopicConnectionFactory` instance for a "ptp" mode for example);
- invalid retrieved administered objects types (for example a `QueueConnectionFactory` instance and a `Topic` destination);
- incorrect JMS user identification ("userName and "password" values).

10.4. WebSphere-MQ example

This example links JORAM with IBM's WebSphere MQ platform (formerly MQ Series). It shows a JORAM topic bridge linked to a WebSphere MQ queue.

10.4.1. Configuring and starting JORAM

Configuring JORAM

We will consider a simple JORAM platform made of one standard server (hosting a JNDI service). It is described by the following `a3servers.xml` file:

```
<?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="fr.dyade.aaa.jndi2.server.JndiServer"
     args="16400"/>
  </server>
</config>
```

The following WebSphere MQ libraries must be added to JORAM's server `lib/` directory:

- com.ibm.mqjms.jar
- com.ibm.mqbind.jar
- com.ibm.mq.jar
- connector.jar
- fscontext.jar

The JORAM distribution already provides the needed `jms.jar`, `jndi.jar` and `jndi.properties`. No need to add those in the `lib/` directory.

Starting JORAM

Starting JORAM is required first because the JORAM server provides the JNDI service needed for binding WebSphereMQ's administered objects.

Launching the JORAM server is done as usual:

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

10.4.2. WebSphere MQ setup

The WebSphere MQ administration tool should be used for creating a queue. Let's call this queue "foreignQueue".

JMS administered objects should then be created. They will eventually be bound to JORAM's JNDI. For that, WebSphere MQ `JMSAdmin.config` file must be edited as follows:

```
INITIAL_CONTEXT_FACTORY=fr.dyade.aaa.jndi2.client.NamingContextFactory
PROVIDER_URL=joram://localhost:16400
```

The JMSAdmin command should then be launched and the JMS objects created and bound to JNDI (with names “foreignCF” and “foreignDest”):

```
InitCtx> DEFINE QCF(foreignCF)
InitCtx> DEFINE Q(foreignDest) QUEUE(foreignQueue)
```

10.4.3. Administering JORAM

Administering JORAM consists in creating and configuring the required bridge destination, and authorizing a client access to it. The following code is proposed:

```
public class AdminSample
{
  public static void main(String[] args) throws Exception
  {
    AdminModule.connect("root", "root", 60);

    Properties prop = new Properties();
    // Setting the communication mode to PTP for interacting
    // with WebSphere MQ
    prop.setProperty("jmsMode", "ptp");

    // Setting the JNDI names of the WebSphere MQ
    // administered objects
    prop.setProperty("connectionFactoryName", "foreignCF");
    prop.setProperty("destinationName", "foreignDest");

    // Creating the bridge topic on JORAM server 0
    String className = "org.objectweb.joram.mom.dest.BridgeTopic";
    Topic bridgeTopic = Topic.create(0, className, prop)

    // Providing free access to it
    bridgeTopic.setFreeReading();
    bridgeTopic.setFreeWriting();
```

```
    // Creating an anonymous user access on JORAM server 0
    javax.jms.ConnectionFactory cf =
      TcpConnectionFactory.create("localhost", 16010);
    User user = User.create("anonymous", "anonymous", 0);

    // Binding the JORAM administered objects
    javax.naming.Context jndiCtx =
      new javax.naming.InitialContext();
    jndiCtx.bind("bridgeDest", bridgeDest);
    jndiCtx.bind("cf", cf);

    jndiCtx.close();

    AdminModule.disconnect();
  }
}
```

10.4.4. The JORAM subscriber

The development of the JORAM subscriber is fully standard. The JORAM application is apparently in relationship with a pure JORAM topic destination, but in fact it consumes messages coming from the WebSphere MQ queue destination, in a JMS 1.1 way, through the Publish/Subscribe communication mode.

The following code is proposed:

```
public class ConsumerSample
{
  public static void main(String[] args) throws Exception
  {
    // Retrieving the JORAM administered objects
    javax.naming.Context jndiCtx =
      new javax.naming.InitialContext();
    Destination dest = (Destination) jndiCtx.lookup("bridgeDest");
    ConnectionFactory cnxFact =
     (ConnectionFactory) jndiCtx.lookup("cf");
    jndiCtx.close();

    Connection cnx = cnxFact.createConnection();
    Session sess =
      cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageConsumer cons = sess.createConsumer(dest);
    cons.setMessageListener(new MsgListener());

    cnx.start();
    System.in.read();
    cnx.close();
  }
}
```

10.4.5. The WebSphere MQ sender

The WebSphere MQ sender is a JMS 1.0.2b standard sender, sending messages to the WebSphere MQ queue through a `QueueSender` resource.

The following code is proposed:

```
public class SenderSample
{
  public static void main(String[] args) throws Exception
  {
    // Retrieving the WebSphere MQ administered objects
    javax.naming.Context jndiCtx =
      new javax.naming.InitialContext();
    Queue queue = (Queue) jndiCtx.lookup("foreignDest");
    QueueConnectionFactory cnxFact =
     (QueueConnectionFactory) jndiCtx.lookup("foreignCF");
    jndiCtx.close();

    QueueConnection cnx = cnxFact.createQueueConnection();
    QueueSession sess = cnx.createQueueSession(true, 0);
    QueueSender sender = sess.createSender(queue);
    TextMessage msg = sess.createTextMessage();

    // Sending 10 messages to the queue
    for (int i = 1; i < 11; i++) {
      msg.setText("WebSphere MQ message " + i);
      sender.send(msg);
    }
    // Commiting the session
    sess.commit();

    cnx.close();
  }
}
```

10.4.6. As a conclusion...

The WebSphere MQ demo distribution is JMS 1.0.2b compliant and does not provide the Publish/Subscribe communication mode. This example illustrates JORAM as a JMS 1.1 Publish/Subscribe broker to WebSphere MQ.

10.5. Running the bridge sample

This sample provides an administration code configuring a single JORAM server with a bridge topic, linked to a foreign JMS queue. A JORAM producer and a JORAM consumer actually interact with this foreign JMS destination through the JORAM bridge.

The sample requires then a foreign JMS server to be running and to have been administered so that the JORAM bridge is usable. How to achieve this is explained in the bridge documentation.

The next picture shows the bridge configuration. The configuration used is centralized and the server run in non persistent mode. The bridge is a topic with which application interact through the JMS 1.1 unified interfaces. This topic is linked with a foreign queue with which it interacts through the JMS 1.0.2b and JMS 1.1 PTP specific interfaces.

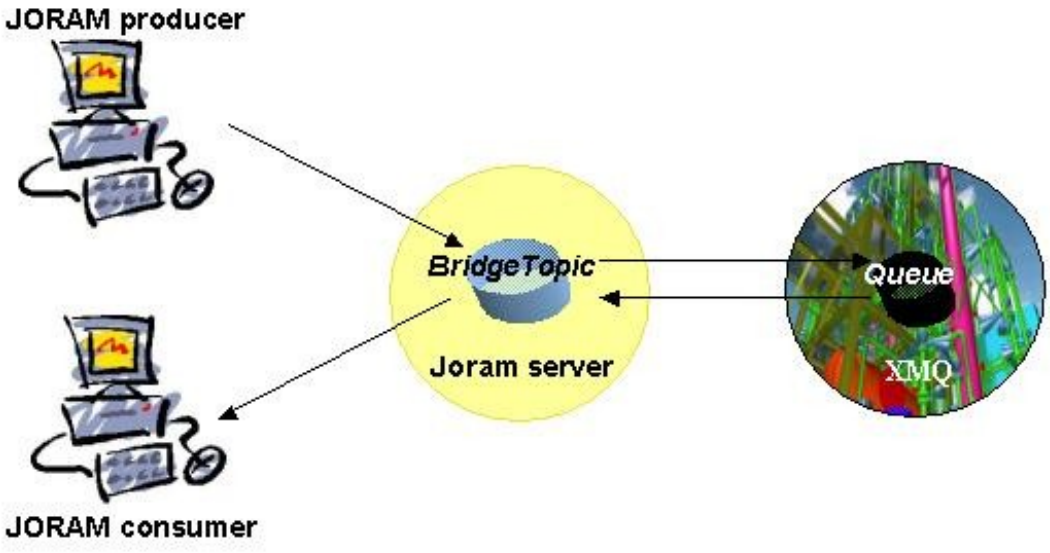


Figure 41 - Bridge sample

Running the demo:

1. Starting JORAM:

   ```
   ant reset single_server
   ```

2. Administering the foreign JMS server, binding its administered objects.
3. Administering JORAM:

   ```
   ant bridge_admin
   ```

4. Sending messages to the foreign JMS destination:

   ```
   ant bridge_producer
   ```

5. Consuming messages on the foreign JMS destination:

   ```
   ant bridge_consumer
   ```

11.Working with sources distribution

11.1. Getting Joram sources

11.1.1. Getting a packaged version of Joram

The packages are downloadable from the following location:

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

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

- **`joram-x.y.z-src.tgz`**, including the client and server sources, as well as the graphical tool sources, the J2ME client sources and the samples sources.

A package is expanded by UNIX users with the ***`gunzip`*** and ***`tar`*** commands; Windows developers can use the ***Winzip*** utility.

11.1.2. Getting Joram from CVS

JORAM CVS page is located at: http://forge.objectweb.org/cvs/?group_id=4. The module to extract is **joram**. A nightly snapshot is generated each day at 00:00 AM CEST and can be downloaded at http://forge.objectweb.org/cvssnapshots/joram-cvs-latest.tar.gz.

11.1.3. Directory structure and description

Joram sources distribution

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

- `lib/`
- `licenses/`
- `samples/`
  - `bin/…`
  - `config/`
  - `src/`
    - `joram/…`
    - `kjoram/…`
- `src/`
  - `com/scalagent/…`

- fr/dyade/aaa/…
- org/objectweb/joram/…

lib/ directory

Contains the libraries needed for compiling the distribution, and running the samples.

- activation.jar, JavaBeans activation framework (neede by soap);
- commons-logging-api.jar, Apache's logger API (used by Jgroups);
- connector-1_5.jar, the JCA 1.5 API;
- jakarta-regexp-1.2.jar, Jakarta's regular expression parser;
- JCup.jar, javaCup Java parser generator;
- jgroups-all.jar, JGroups library (used for H.A. Implementation)
- jms.jar, the JMS API;
- jmxri.jar, JMX reference implementation (used for JDK prior 1.5);
- jmxtools.jar, JMX tools (used for JDK prior 1.5);
- jndi.jar, the JNDI API;
- jta.jar, the JTA API;
- kxml.jar, Enhydra's kXML implementation (needed for kJoram);
- mail.jar, the JavaMail API;
- midpapi.jar, Sun's MIDP API (needed for kJoram).
- ow_monolog.jar, ObjectWeb's logger API and its wrappers;
- soap.jar, Apache's SOAP implementation;
- soap.war, Apache's SOAP web archive file;
- versions, reference version for libraries.

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 2).

src/com/scalagent/ directory

Contains the sources of specific ScalAgent component: scheduler queue, kJORAM client, etc.

src/fr/dyade/aaa/ directory

Contains the sources of the agent platform.

src/org/objectweb/joram directory

Contains the sources of JORAM server and client.

11.2. Compiling and shipping Joram

JORAM distribution is ready for compiling with Apache ***Ant*** utility. *Ant* can be downloaded from http://jakarta.apache.org/ant/. Documentation is available at the same location.

In the src/ directory the build.xml files provides the main targets for convenience. The available *Ant* targets can be listed by typing:

```
$ ant -projecthelp
Buildfile: build.xml
Main targets:

 clean        cleans the generated shipments
 javadoc      generates the Joram javadoc
 ship.joram   builds and ships Joram

Default target: ship.joram
```

The default target creates a `ship/` directory with a typical Joram binary distribution.

JORAM distribution is ready for compiling with Apache ***Ant*** utility. *Ant* can be downloaded from http://jakarta.apache.org/ant/. Documentation is available at the same location. In the `src/` directory, 3 build files are provided:

- `build.xml` specifies the main targets for convenience, it has been described in 11.2.
- `joram.xml` specifies the targets for compiling and shipping the MOM classes, the Joram client classes and the graphical admin tool classes;

The available *Ant* targets for ***joram.xml*** can be listed for each build files by typing:

```
$ ant -buildfile joram.xml -projecthelp
Buildfile: joram.xml
Main targets:

 build.joram          --> Builds Joram server and client jars.
 build.kjoram         --> Builds kjoram jars.
 compile              --> Compiles sources.
 javadoc              --> Generates the Joram's client and server JavaDoc.
 javadoc.all          --> Generates the complete JavaDoc.
 release.jar          --> Builds a Joram binary distribution.
 release.src          --> Builds a Joram source distribution.
 releases             --> Builds a Joram complete distribution.
 ship                 --> Creates a Joram shipment.
 ship.adapter         --> Ships a JORAM JCA adapter
 ship.jonasadapter    --> Ships a JORAM JCA adapter for JOnAS
 ship.remoteadapter   --> Ships a JORAM JCA adapter for non collocated use

Default target: ship
```

The main targets are:

- `ship` builds from `src/` and `lib/` directories a directory `ship/` with all the jars needed to launch Joram's client and server.
- `javadoc` builds a `doc/` directory with Joram's client and server javadoc.
- `release;jar`, `release.src` and `release` respectively build a binary distribution (joram-x.y.z.tgz), a source distribution (joram-x.y.z-src.tgz) and a complete distribution (all files available for download on ObjectWeb). Theses files are produced in `releases/` directory.

11.2.1. Compiling Joram

The best way to build Joram is :

- *ant -f joram.xml ship (see below)*

```
$ ant -f joram.xml ship
Buildfile: joram.xml
init:
setver:
init.joram:
  [echo] Joram 4.3.11

prepare:
  [mkdir] Created dir: C:\owjoram\joram\classes

compile:
  [javac] Compiling 611 source files to C:\owjoram\joram\classes
  [javac] Note: * uses or overrides a deprecated API.
  [javac] Note: Recompile with -Xlint:deprecation for details.

build.joram:
  [mkdir] Created dir: C:\owjoram\joram\build\META-INF
  [jar] Building jar: C:\owjoram\joram\build\joram-shared.jar
  [jar] Building jar: C:\owjoram\joram\build\joram-mom.jar
  [jar] Building jar: C:\owjoram\joram\build\joram-client.jar
  [jar] Building jar: C:\owjoram\joram\build\joram-connector.jar
  [jar] Building jar: C:\owjoram\joram\build\joram-raconfig.jar
  [jar] Building jar: C:\owjoram\joram\build\joram-config.jar
  [jar] Building jar: C:\owjoram\joram\build\joram-gui.jar

build.kjoram:
  [javac] Compiling 129 source files to C:\owjoram\joram\
classes
  [jar] Building jar: C:\owjoram\joram\build\joram-kclient.jar

ship:
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib
  [copy] Copying 1 file to C:\owjoram\joram\ship\lib

BUILD SUCCESSFUL
Total time: 35 seconds
```

The release number (in our case Joram 4.3.11) is printed at starting; this number is fixed in the `src/build.properties` file, you can overload it with a -Dversion=x.y.z in the command line.

This creates the `classes/`, `build/` and `doc/` directories when compiling is requested for the first time or after a clean. The `classes/` directory holds the compiled classes, the `build/` directory contains jar files related to joram, then the ship/ directory contains a binary distribution of Joram with all needed jars :

- joram-client.jar, needed by Joram's client, it contains all classes implementing the JMS API.
- joram-mom.jar, needed by Joram's server, it contains all classes to launch a MOM instance.
- joram-shared.jar, contains classes shared between Joram's client and server. Essentially the classes for communication object between the client and the server.
- joram-kclient.jar, contains all Joram's classes needed for J2ME client.

The classes specific to the administration tool are in a separate jar's file :

- joram-gui.jar

And there is 3 jars related to the JCA adapter :

- joram-config.jar
- joram-connector.jar
- joram-raconfig.jar

It contains also all 'external' jars needed to execute Joram's client or server.

You can also, either choose to build separately each component or directly the release files as available for download (see below).

11.2.2. Compiling kJoram

Building kJoram requires to use a 1.3 or 1.4 JDK, and to generates 1.1 bytecode. Then, type:

```
ant -buildfile kjoram.xml build.kjoram
```

It creates the `classes/`, `build/` and `doc/` directories when compiling is requested for the first time or after a clean. The `classes/` directory holds the compiled classes, the `joram-kclient.jar` library is created in the `build/` directory.

11.2.3. Compiling the administration tool

Compiling the administration tool requires to use jdk 1.4. It is done by typing:

```
ant -buildfile joram.xml ship.gui
```

This creates the `classes/` and `javadoc/` directories when compiling is requested for the first time or after a clean. The generated `joramgui.jar` library is created in the `lib/` directory, the `classes/` directory holds the compiled classes.

11.2.4. Generating the javadoc

Simply type:

```
ant -buildfile joram.xml javadoc
```

This creates the `classes/`, `build/` and `apidoc/` directories when generating the javadoc is requested for the first time or after a clean. The `classes/` and `build/` directories stays empty, `javadoc/` holds the doc.

The `javadoc.all` target allows to produce the documentation for all classes, not only Joram's client and server (`org.objectweb.joram` packages), but also the agent platform's (`fr.dyade.aaa` packages) or the additional features (`com.scalagent` packages).

11.2.5.

You can build source, binary or complete distribution of Joram using `release.src`, `release.jar` and `release target`. The related files are copied in the releases directory, they are equivalent to the one available for download on Joram's web server.

11.2.6. Cleaning

To remove the generated classes and libraries:

ant clean (equivalent to ant -buildfile joram.xml clean and ant -buildfile kjoram.xml clean)

This removes the `classes/` and `javadoc/` directories if they exist, the `joram.jar`, `kjoram.jar`, `joramgui.jar` and `mom.jar` libraries from the `lib/` directory if they exist.