JORAM 3.7

ADMINISTRATION GUIDE

Author: Frédéric Maistre.
November 6, 2003

TABLE

1. Updates4

2. Introduction 5

3. JORAM's messaging platform 6

3.1. Definitions 6

3.2. Platform configuration 7

3.2.1. Server services 7

3.2.2. Configuration files 7

3.3. Running a platform 9

3.4. Log configuration 10

4. Administering JORAM 11

4.1. Introduction 11

4.2. Basic administration tasks 11

4.2.1. Allowing a user connection 11

4.2.2. Creating a destination 12

4.3. Special features 13

4.3.1. Dead message queue 13

4.3.2. Hierarchical topic 16

4.3.3. Clustered topic 17

5. Administration interfaces 19

5.1. Introduction 19

5.2. Using the JMS interfaces 20

5.2.1. *Connecting to the platform* 20

5.2.2. *Communicating with the platform* 20

5.2.3. *Requests and replies* 22

5.2.4. *Creating administered objects* 27

5.2.5. *Example* 29

5.3. *Using the `AdminItf` interface* 31

5.3.1. *Connecting an administrator* 31

5.3.2. *Stopping a server* 32

5.3.3. *Managing a user* 32

5.3.4. *Managing a destination* 32

5.3.5. *Creating a `ConnectionFactory` instance* 34

5.3.6. *Managing a hierarchical topic* 35

5.3.7. *Managing a clustered topic* 36

5.3.8. *Managing a dead message queue* 37

5.4. *Using the graphical tool* 39

6. JORAM SOAP mode 40

6.1. *Introduction* 40

6.2. *Platform configuration* 41

6.2.1. *The `SoapProxy` service* 41

6.2.2. *Configuration file* 41

6.2.3. *Running the platform* 42

6.3. *Administering* 42

6.3.1. *Introduction* 42

6.3.2. *Setting a SOAP user* 43

6.3.3. *Creating a SOAP `ConnectionFactory` object* 43

6.3.4. *Connecting a SOAP administrator* 45

6.3.5. *Accessing JNDI through SOAP* 46

1. Updates

This new version of the administration document simply corrects some bugs and mistakes of the previous document (JORAM 3.6.3 administration guide).

It also mentions the new graphical administration tool, section 5.4.

2. Introduction

JORAM provides a messaging platform allowing distributed applications to exchange data through message communication (fig. 1).

Figure 1: applications exchanging data through messaging

The messaging system takes care of distributing the data produced by an application to an other 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.

Generally, the link between JORAM servers and clients relies on TCP connections. However, in order to open JORAM's messaging platform to non Java or non TCP clients, the SOAP protocol is supported as a client – platform communication protocol.

Running a JORAM messaging platform takes first to set and start a platform configuration, and then to configure it so that it is accessible and usable by JMS clients. How to achieve this is explained by this tutorial (its last section addresses the setting of a platform providing a SOAP access).

3. JORAM's messaging platform

3.1. Definitions

A JORAM messaging platform is constituted by one or many servers, interconnected, possibly running on remote nodes (fig. 2).

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

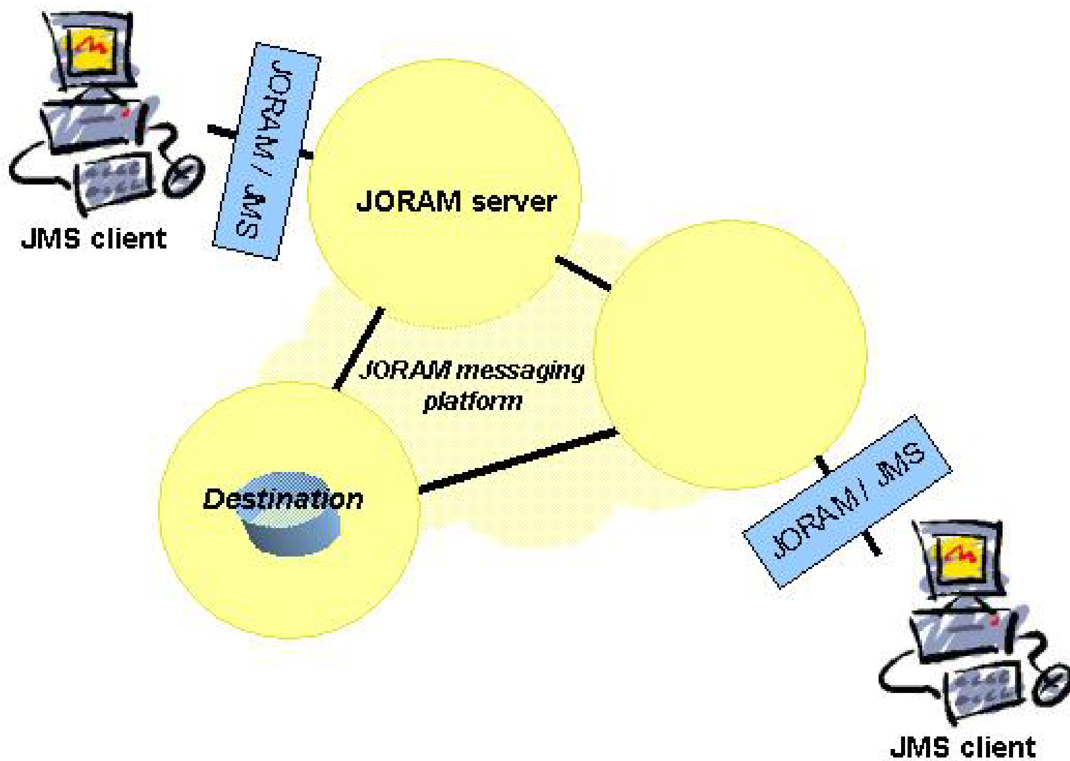

Figure 2: JORAM platform and clients

3.2. Platform configuration

Configuring a 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.

3.2.1. Server services

The services a server may host are :

- A name service, internaly used for booting the administration of a platform. This service is **mandatory** on **server 0**.
- An administration topic service, receiving administration messages and performing administration tasks. This service is **mandatory** on any server that needs to be administered (any server on which a destination will be created, or a user).
- A connection service, listening to a given port, allowing external clients to connect (through JMS connections). 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 service authorizing an administrator connection.
- A JNDI service, listening to a given port, providing a naming server to JORAM 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.

3.2.2. Configuration files

A platform configuration is described by an XML configuration file.

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*, and part of the domain *D1*. It hosts a name service, an administration topic service, and a connection factory service listening on port *16010* and allowing an administrator identified by *root – root* to connect. A JNDI service is also provided, listening to JNDI requests on port *16400*.

```
<?xml version="1.0"?>
<config>
<server id="0" name="S0" hostname="localhost">
  <service class="fr.dyade.aaa.ns.NameService"/>
  <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
  <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
           args="16010 root root"/>
  <service class="fr.dyade.aaa.jndi2.server.JndiServer"
           args="16400"/>
</server>
</config>
```

In order to provide 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 to retrieve the naming context through:

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

Distributed configuration

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

```
<?xml version="1.0"?>
<config>
<domain name="D1"/>
<server id="0" name="S0" hostname="host0">
  <network domain="D1" port="16301"/>
  <service class="fr.dyade.aaa.ns.NameService"/>
  <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
  <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
           args="16010 root root"/>
  <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="fr.dyade.aaa.mom.dest.AdminTopic"/>
  <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
           args="16010"/>
</server>
…
```

```
…
<server id="2" name="S2" hostname="host2">
  <network domain="D1" port="16301"/>
  <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
  <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
           args="16010"/>
</server>
</config>
```

This configuration is made of 3 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. Servers 1 and 2 only provide a administration topic service and a connection service.

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
```

3.3. Running a platform

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

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

- On node 0:

```
java -DTransaction=fr.dyade.aaa.util.NullTransaction
fr.dyade.aaa.agent.AgentServer 0 ./s0
```

- On node 1:

```
java -DTransaction=fr.dyade.aaa.util.NullTransaction
fr.dyade.aaa.agent.AgentServer 1 ./s1
```

- On node 2:

```
java -DTransaction=fr.dyade.aaa.util.NullTransaction
fr.dyade.aaa.agent.AgentServer 2 ./s2
```

When launched using the above commands, the 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.

For starting a persistent server, on node i:

```
java -DTransaction=fr.dyade.aaa.util.ATransaction
fr.dyade.aaa.agent.AgentServer i ./si
```

It is recommended when deploying a distributed architecture to start server 0 first.

3.4. Log 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.

More important, this file 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 a JORAM server underlying agent platform.
- *MOM logs* (categories starting with `fr.dyade.aaa.mom`): these categories log what happens in a JORAM server, more particularly:
  - in the server's proxies (`fr.dyade.aaa.mom.Proxy`),
  - in the server's destinations (`fr.dyade.aaa.mom.Destination`).
- *JORAM logs* (`fr.dyade.aaa.joram.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`).

4. Administering JORAM

4.1. Introduction

When a platform is up and running, it needs to be administered to become accessible and usable by JMS clients. The basic administration tasks are creating/deleting physical destinations server side, and (un)setting users access to the platform. In order to isolate JMS clients from any provider proprietary aspect, the administration phase also consists in creating the `ConnectionFactory` and `Destination` administered objects (see JMS specification, §4.2).

4.2. Basic administration tasks

4.2.1. Allowing a user connection

Clients – platform connections are TCP connections wrapped by JMS `Connection` instances. A JMS `Connection` is created by calling the `createConnection` method of a `ConnectionFactory` instance.

A `ConnectionFactory` instance wraps a set of server parameters, such as the server URL, and the port the server is listening to. Thus, a `ConnectionFactory` instance wrapping the parameters of a given server is mandatory for allowing a client to open a connection with this server. One can easily understand that the `ConnectionFactory` interface allows to isolate clients from the proprietary parameters needed for opening a connection with a platform (fig. 3).

But creating `ConnectionFactory` instances won't be enough for allowing clients connections. A connection is opened either anonymously by calling the `ConnectionFactory.createConnection()` method, or by calling the `ConnectionFactory.createConnection(String name, String password)` method. The user identification (either *anonymous – anonymous*, or *name – password*) must be known server side. Otherwise the `createConnection` methods won't succeed and will throw a `JMSSecurityException`.

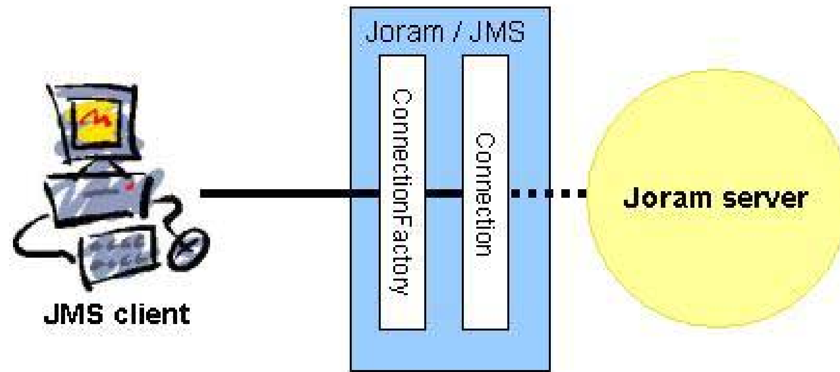

Figure 3: a client connected to a server

Allowing a client access to the platform requires then:

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

4.2.2. Creating a destination

Clients applications exchange messages 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 2, 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 JMS `Destination` instances. A JMS `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 (fig. 4).

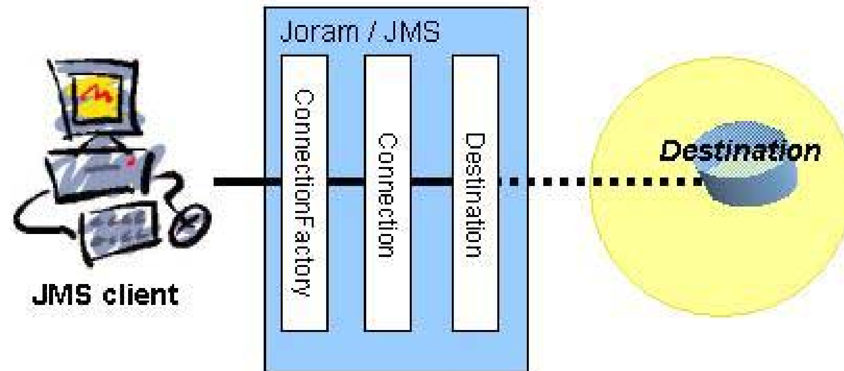

Figure 4: a client accessing a server destination

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 JMS `Destination` instance wrapping the parameters of the server side destination,
3. third, binding the `Destination` instance in a name space such as JNDI, 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.

4.3. Special features

The following special features are features provided by JORAM but not described and/or required by the JMS specifications.

4.3.1. 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 figure 5 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.

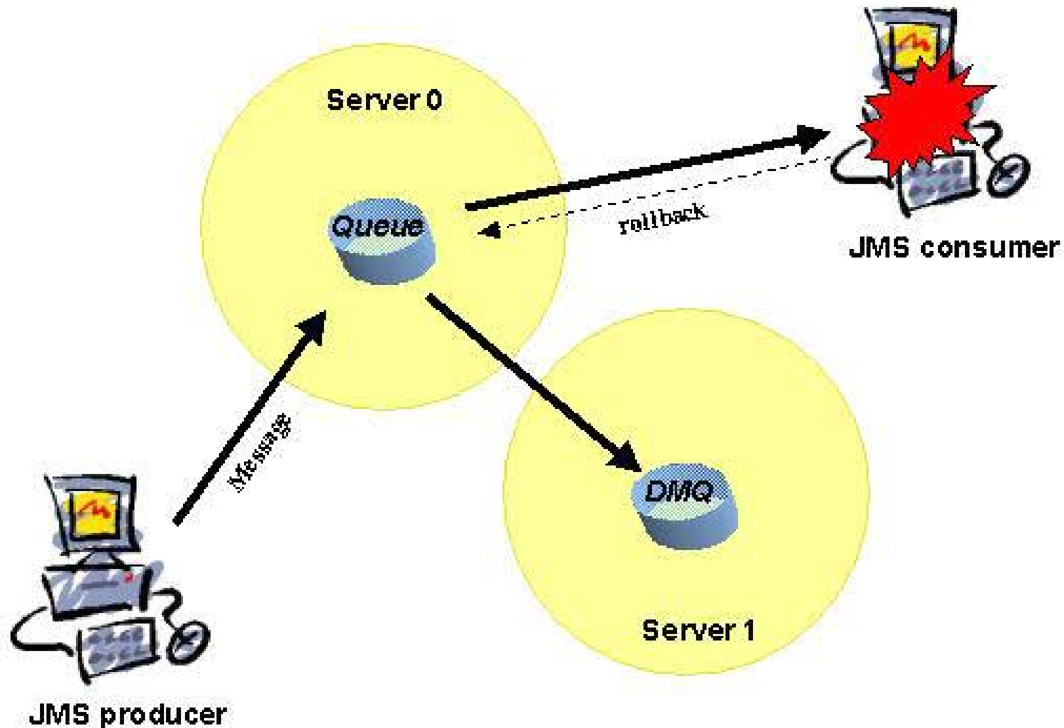

Figure 5: messages on a queue sent to a DMQ

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 are 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"));
```

4.3.2. Hierarchical topic

Introduction

The JMS specification allows topics to have a hierarchical structure such as the one shown in figure 6. 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.

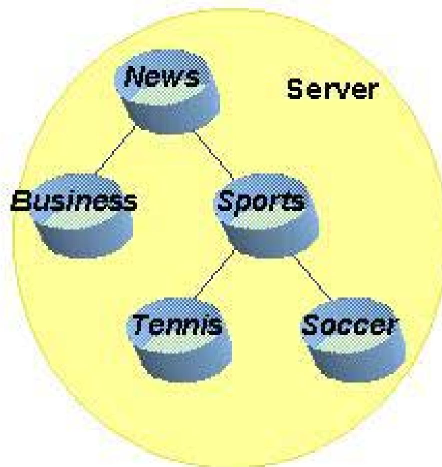

Figure 6: a hierarchical topic

Example

The example of fig. 6 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.

To be noted, a hierarchy may be spread over many servers (figure 7).

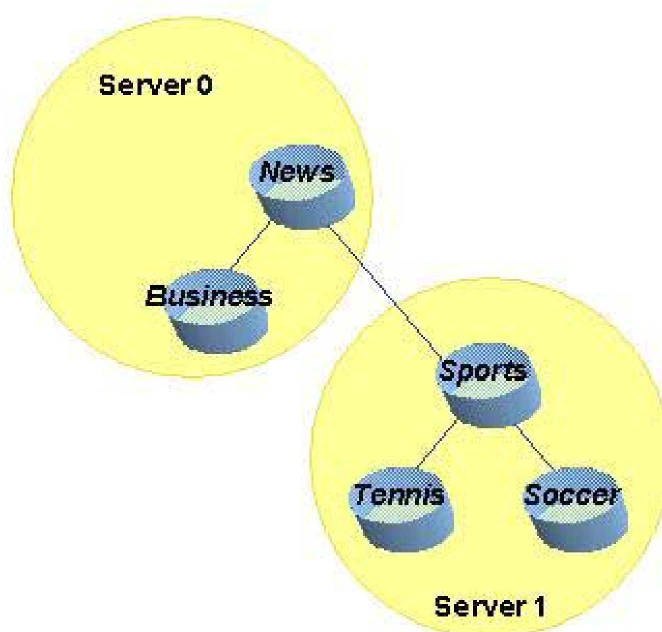

Figure 7: a distributed hierarchical topic

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).

4.3.3. Clustered topic

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 representants, one per server. The figure 8 shows such a topic located on three servers.

Such an architecture allows a publisher to publish messages on a representant of the topic. In the example shown figure 8, the publisher works with the representant on server 1. If a

subscriber subscribed to any other representant (on server 2 in our example), it will get the messages produced by the publisher.

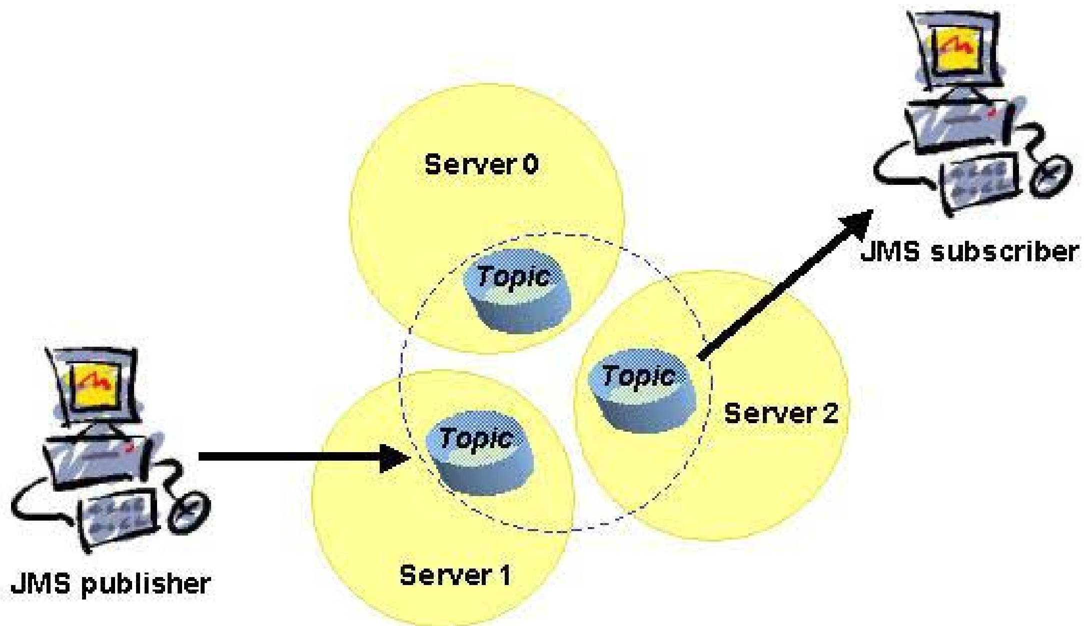

Figure 8: a clustered topic

Added value

This special feature introduces more flexibility and availability to Pub/Sub communication. If server 2 is down, for example, the other representants of the topic would go on working. The publisher would successfully send its messages to the representant on server 1, and a subscriber to the representant 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

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

Each clustered topic representant 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 representant of the cluster.

5. Administration interfaces

5.1. Introduction

Performing the administration tasks described in the previous section is done by a special client called an administrator. This client can be seen as a classical JMS client opening a connection with the JORAM platform, after its identification has been authenticated. Its administrator nature allows it to access a special topic destination dedicated to administration (figure 9).

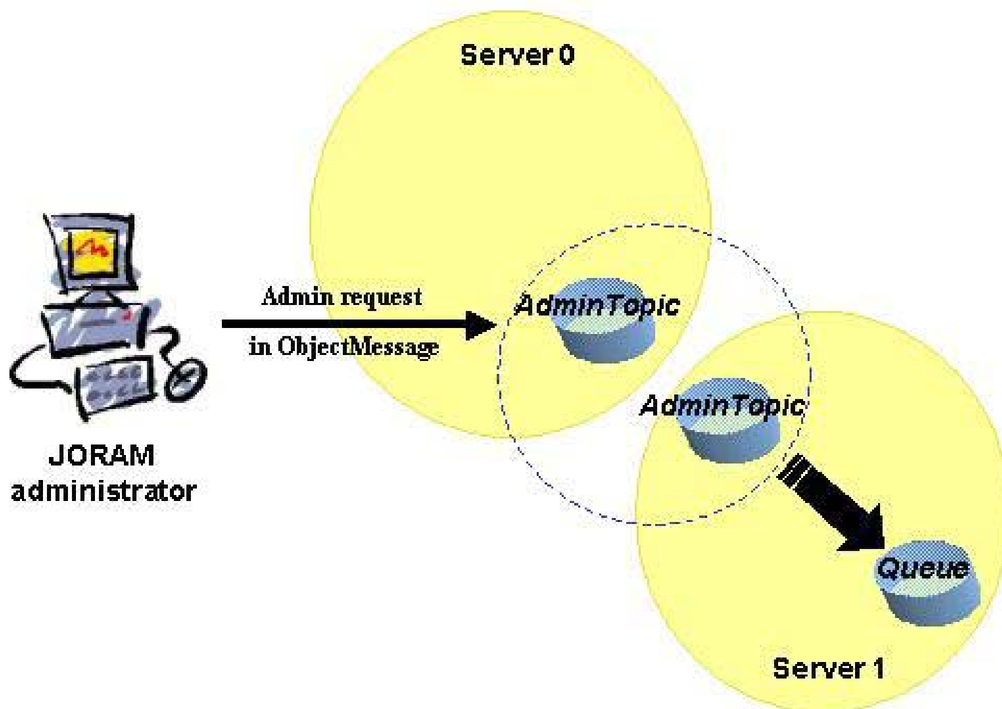

Figure 9: administering through the dedicated topics

As figure 9 shows, each server of a configuration hosts a topic dedicated to administration (let's call it an AdminTopic). The AdminTopics of a platform form a cluster of topics as introduced by section 4.3.3. The administrator sends its requests to its local AdminTopic representant, and the request is processed by the AdminTopic representant on the target server. For example, on figure 9, the administrator requested the creation of a queue on server 1.

Thus, an immediate way to perform administration is to develop a JMS client sending these admin requests (section 5.2). It is also possible to use higher level methods defined in a proprietary administration interface (section 5.3). And finaly, a graphical administration tool is provided since release 3.7 (section 5.4).

5.2. Using the JMS interfaces

An example of an administration client code using the JMS API can be found in the `fr.dyade.aaa.joram.admin` package. The `AdminImpl` class is developed that way.

5.2.1. Connecting to the platform

The administrator client follows the same steps as any JMS client. It first needs a `ConnectionFactory` instance for connecting to a given server (running on *hostName*, and listening on port *portNumber*). As the platform may not have been administered yet, it will construct the instance it needs rather than retrieving it from JNDI.

```
javax.jms.ConnectionFactory cnxFact =
  new fr.dyade.aaa.joram.tcp.TcpConnectionFactory("hostName",
                                                  portNumber);
```

If the server parameters are incorrect, an `UnknownHostException` is thrown. Otherwise, the administrator may then open a connection:

```
javax.jms.Connection cnx;
cnx = cnxFact.createConnection("adminName", "adminPass");
```

This method will throw a `javax.jms.JMSSecurityException` if this identification is invalid. The identification is valid if the server running on *hostName* and listening to port *portNumber* accepts administrator's connection with this identification. Meaning that the XML configuration file must provide the following service to the server on *hostName*:

```
<service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
         args="portNumber adminName adminPass"/>
```

If connecting succeeded, the administrator is able to open a session:

```
javax.jms.Session sess;
sess = cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Finally, the administrator needs to access the representant of the administration topic local to the server it is connected to. This is achieved through the `Session.createTopic` method, by using the reserved name ***#AdminTopic***:

```
javax.jms.Topic adminTopic = sess.createTopic("#AdminTopic");
```

Trying to access the administration topic with the same call from a non administrator client would not work. Accessing the created `Topic` instance would throw a `javax.jms.InvalidDestinationException`.

5.2.2. Communicating with the platform

An administrator sends administration requests to the platform, and may get replies from the platform.

Sending a request

As for any JMS client, a `MessageProducer` instance is needed for sending messages to the administration topic:

```
javax.jms.MessageProducer producer;
producer = sess.createMessageProducer(adminTopic);
```

Administration requests classes are located in the `fr.dyade.aaa.mom.admin` package. They all inherit from the `AdminRequest` abstract class. Sending a request simply consists in building the request, wrapping it into a JMS `ObjectMessage` instance, and sending it:

```
fr.dyade.aaa.mom.admin.AdminRequest request = new …;
javax.jms.ObjectMessage requestMsg;
requestMsg = sess.createObjectMessage(request);
producer.send(requestMsg);
```

Getting a reply

The request will be acknowledged if the administrator set the **`JMSReplyTo`** field of its request message. This field is used by the administration topic for identifying the destination where administration replies should be sent.

For example, a temporary destination may be created and set in the request message:

```
javax.jms.TemporaryQueue repliesQueue;
repliesQueue = sess.createTemporaryQueue();
requestMsg.setJMSReplyTo(repliesQueue);
```

As for any JMS client, a `MessageConsumer` instance is needed for getting messages from the temporary queue:

```
javax.jms.MessageConsumer consumer;
consumer = sess.createMessageConsumer(repliesQueue);
```

Administration replies classes are located in the `fr.dyade.aaa.mom.admin` package. They all inherit from the `AdminReply` class. A reply is wrapped into a JMS `ObjectMessage` instance generated by the AdminTopic:

```
fr.dyade.aaa.mom.admin.AdminReply reply;
javax.jms.ObjectMessage replyMsg;
replyMsg = (javax.jms.ObjectMessage) consumer.receive();
reply = (fr.dyade.aaa.admin.AdminReply) replyMsg.getObject();
```

Linking a reply to a request

It is possible to use the *JMSMessageID* and *JMSCorrelationID* message fields for linking a reply with a request. When sending the request message, its identifier is automatically set. It is accessible through the `getJMSMessageID()` method:

```
producer.send(requestMsg);
String requestId = requestMsg.getJMSMessageID();
```

The reply replying to this request will carry a correlation identifier equal to the request's identifier. This correlation identifier is accessible through the `getCorrelationID()` method:

```
replyMsg = (javax.jms.ObjectMessage) consumer.receive();
String replyId = replyMsg.getJMSCorrelationID();
```

Using a **`<Queue/Topic>Requestor`**

The `javax.jms.QueueRequestor` or `javax.jms.TopicRequestor` classes may be used for simply implementing a synchronous request/reply protocol (a mechanism using temporary destinations is actually involved); however it requires differentiated factory, connection and session objects (PTP or Pub/Sub); for example:

```
javax.jms.TopicSession topicSess = …;
javax.jms.TopicRequestor requestor;
requestor = new javax.jms.TopicRequestor(topicSess, adminTopic);
replyMsg = (javax.jms.ObjectMessage) requestor.request(requestMsg);
```

The protocol implemented in the `fr.dyade.aaa.joram.admin.AdminImpl` class actually uses a `TopicRequestor` instance.

5.2.3. Requests and replies

Requests

The requests available for administering a platform are located in the `fr.dyade.aaa.mom.admin` package and described below.

StopServerRequest

- Purpose: requests to stop a given server.
- Constructor: `StopServerRequest(int serverId)`
  - *serverId*: identifier of the server to stop.

CreateDestinationRequest

- Purpose: requests the creation of a destination, or retrieves an existing destination given its name.
- Constructor: `CreateDestinationRequest(int serverId, String name, String className)`
  - *serverId*: identifier of the server where deploying the destination.
  - *name*: name attributed to the destination, not mandatory (may be null).
  - *className*: name of the destination class; for example, for a queue: "`fr.dyade.aaa.mom.dest.Queue`".
- Failure: the request is not executed if the server does not belong to the platform; it fails if the queue deployement fails server side.

DeleteDestination

- Purpose: requests the deletion of a JORAM destination.
- Constructor: `DeleteDestination(String id)`
  - *id*: identifier of the destination to be deleted. Obtained by calling the `get<Queue/Topic>Name()` method on the destination.
- Failure: the request fails if the destination is not a valid JORAM destination; it is not effective if the destination has already been removed.

SetCluster

- Purpose: links a given topic with an other topic to set a cluster.
- Constructor: `SetCluster(String initId, String topId)`
  - *initId*: identifier of the topic already part of the cluster, or chosen as the initiator of the cluster. Obtained by calling the `getTopicName()` method on the topic.
  - *topId*: identifier of the topic joining the cluster. Obtained by calling the `getTopicName()` method on the topic.

- Failure: the request fails if one of the topics is deleted or is not a valid JORAM topic, if one of the topics is already part of a hierarchy, or if the joining topic is already part of a cluster.

UnsetCluster

- Purpose: requests a topic to leave the cluster it is part of.
- Constructor: `UnsetCluster(String id)`
    - *id*: identifier of the topic requested to leave its cluster. Obtained by calling the `getTopicName()` method on the topic.
- Failure: the request fails if the topic is not a valid JORAM topic, or if it has been deleted, or if it is not part of any cluster.

SetFather

- Purpose: links a given topic with an other topic to set a hierarchy.
- Constructor: `SetFather(String father, String son)`
    - *father*: identifier of the topic chosen to be the father. Obtained by calling the `getTopicName()` method on the topic.
    - *son*: identifier of the topic to be set as the son of the previous topic parameter. Obtained by calling the `getTopicName()` method on the topic.
- Failure: the request fails if one of the topics is deleted or is not a valid JORAM topic, or if one of the topics is already part of a hierarchy or a cluster.

UnsetFather

- Purpose: requests a given topic to unset its hierarchical father.
- Constructor: `UnsetFather(String id)`
    - *id*: identifier of the topic requested to unset its father. Obtained by calling the `getTopicName()` method on the topic.
- Failure: the request fails if the topic is not a valid JORAM topic, or if it is deleted or not part of any hierarchy.

CreateUserRequest

- Purpose: requests the creation of a TCP user.
- Constructor: `CreateUserRequest(String userName, String userPass, int serverId)`
    - *userName*: name of the user.
    - *userPass*: password of the user.
    - *serverId*: identifier of the server where creating the user.
- Failure: the request is not executed if the server does not belong to the platform; it fails if the user name is already taken server side, or if the deployement of its proxy fails.

UpdateUser

- Purpose: updates the identification of a user.
- Constructor: `UpdateUser(String userName, String proxId, String newName, String newPass)`
    - *userName*: name of the user.

  - *proxId*: identifier of its proxy, obtained by calling the `getProxyId()` method on the user.
  - *newName*: the new user name.
  - *newPass*: the new user password.
- Failure: the request fails if the user does not exist server side, of if its new name is already taken.

DeleteUser

- Purpose: requests the deletion of a user.
- Constructor: `DeleteUser(String userName, String proxId)`
  - *userName*: name of the user.
  - *proxId*: identifier of its proxy, obtained by calling the `getProxyId()` method on the user.
- Failure: the request is not effective if the user does not exist server side.

SetReader

- Purpose: requests a destination to provide a reading access to a user.
- Constructor: `SetReader(String userProxId, String destId)`
  - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user. ***null*** for granting the read right to everybody.
  - *destId*: identifier of the destination, obtained by calling the `get<Queue/Topic>Name()` method on the destination.
- Failure: the request fails if the destination is not valid JORAM destination, or if the destination is deleted.

UnsetReader

- Purpose: requests a destination to unset a user reading access.
- Constructor: `UnsetReader(String userProxId, String destId)`
  - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user. ***null*** for removing the free reading right.
  - *destId*: identifier of the destination, obtained by calling the `get<Queue/Topic>Name()` method on the destination.
- Failure: the request fails if the destination is not valid JORAM destination, or if the destination is deleted.

SetWriter

- Purpose: requests a destination to provide a writing access to a user.
- Constructor: `SetWriter(String userProxId, String destId)`
  - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user. ***null*** for granting the write right to everybody.
  - *destId*: identifier of the destination, obtained by calling the `get<Queue/Topic>Name()` method on the destination.
- Failure: the request fails if the destination is not valid JORAM destination, or if the destination is deleted.

UnsetWriter

- Purpose: requests a destination to unset a user writing access.
- Constructor: `UnsetWriter(String userProxId, String destId)`
    - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user. ***null*** for removing the free writing right.
    - *destId*: identifier of the destination, obtained by calling the `get<Queue/Topic>Name()` method on the destination.
- Failure: the request fails if the destination is not valid JORAM destination, or if the destination is deleted.

SetDefaultDMQ

- Purpose: sets a given dead message queue as the default DMQ of a given server.
- Constructor: `SetDefaultDMQ(String dmqId, int serverId)`
    - *dmqId*: identifier of the dead message queue, obtained by calling the `getQueueName()` method on the queue.
    - *serverId*: identifier of the server the DMQ is set for.
- Failure: the request is not effective if the server does not belong to the platform.

UnsetDefaultDMQ

- Purpose: unsets the default dead message queue of a given server.
- Constructor: `UnsetDefaultDMQ(int serverId)`
    - *serverId*: identifier of the server the DMQ is set for.
- Failure: the request is not effective if the server does not belong to the platform.

SetUserDMQ

- Purpose: sets a given dead message queue as the DMQ of a given user.
- Constructor: `SetUserDMQ(String userProxId, String dmqId)`
    - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user.
    - *dmqId*: identifier of the dead message queue, obtained by calling the `getQueueName()` method on the queue.
- Failure: the request fails if the user is not valid JORAM user, or if the user is deleted.

UnsetUserDMQ

- Purpose: unsets the dead message queue of a given user.
- Constructor: `UnsetUserDMQ(String userProxId)`
    - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user.
- Failure: the request fails if the user is not valid JORAM user, or if the user is deleted.

SetDestinationDMQ

- Purpose: sets a given dead message queue as the DMQ of a given destination.
- Constructor: `SetDestinationDMQ(String destId, String dmqId)`
    - *destId*: identifier of the destination, obtained by calling the `get<Queue/Topic>Name()` method on the destination.

  - *dmqId*: identifier of the dead message queue, obtained by calling the `getQueueName()` method on the queue.
- Failure: the request fails if the destination is not valid JORAM destination, or if the destination is deleted.

UnsetDestinationDMQ

- Purpose: unsets the dead message queue of a given destination.
- Constructor: `UnsetDestinationDMQ(String destId)`
  - *destId*: identifier of the destination, obtained by calling the `get<Queue/Topic>Name()` method on the destination.
- Failure: the request fails if the destination is not valid JORAM destination, or if the destination is deleted.

SetDefaultThreshold

- Purpose: sets a given threshold as the default threshold of a given server.
- Constructor: `SetDefaultThreshold(int threshold, int serverId)`
  - *threshold*: threshold value.
  - *serverId*: identifier of the server the threshold is set for.
- Failure: the request is not effective if the server does not belong to the platform.

UnsetDefaultThreshold

- Purpose: unsets the default threshold of a given server.
- Constructor: `UnsetDefaultThreshold(int serverId)`
  - *serverId*: identifier of the server the threshold is set for.
- Failure: the request is not effective if the server does not belong to the platform.

SetUserThreshold

- Purpose: sets a given threshold as the threshold of a given user.
- Constructor: `SetUserThreshold(String userProxId, int threshold)`
  - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user.
  - *threshold*: threshold value.
- Failure: the request fails if the user is not valid JORAM user, or if the user is deleted.

UnsetUserThreshold

- Purpose: unsets the threshold of a given user.
- Constructor: `UnsetUserThreshold(String userProxId)`
  - *userProxId*: identifier of the user proxy, obtained by calling the `getProxyId()` method on the user.
- Failure: the request fails if the user is not valid JORAM user, or if the user is deleted.

SetQueueThreshold

- Purpose: sets a given threshold as the threshold of a given queue.
- Constructor: `SetQueueThreshold(String queueId, int threshold)`

    - *queueId*: identifier of the queue, obtained by calling the `getQueueName()` method on the queue.
    - *threshold*: threshold value.
- Failure: the request fails if the queue is not valid JORAM queue, or if the queue is deleted.

UnsetQueueThreshold

- Purpose: unsets the threshold of a given queue.
- Constructor: `UnsetQueueThreshold(String queueId)`
    - *queueId*: identifier of the queue, obtained by calling the `getQueueName()` method on the queue.
- Failure: the request fails if the queue is not valid JORAM queue, or if the queue is deleted.

Replies

The above requests are replied to by `fr.dyade.aaa.mom.admin.AdminReply` instances, described below.

AdminReply

- Purpose: acknowledges an administration request.
- Methods:
    - `boolean succeeded()`: returns *true* if the request was successful, *false* otherwise.
    - `String getInfo()`: returns information related to the processing of the request.

CreateDestinationReply

- Purpose: acknowledges a destination creation request.
- Methods: the `CreateDestinationReply` class inherits from `AdminReply`. On top of the above methods, the following method is provided:
    - `String getDestId()`: returns the identifier of the destination created server side.

CreateUserReply

- Purpose: acknowledges a user creation request.
- Methods: the `CreateUserReply` class inherits from `AdminReply`. On top of the above methods, the following method is provided:
    - `String getProxId()`: returns the identifier of the user's proxy created server side.

5.2.4. Creating administered objects

The administration phase mainly consists in administering the messaging platform, but also should create administered objects later used by administrators and clients. Those objects are the `ConnectionFactory`, `Destination` and `User` objects.

javax.jms.ConnectionFactory

`ConnectionFactory` classes for the TCP protocol are:

- `fr.dyade.aaa.joram.tcp.TcpConnectionFactory`,
- `fr.dyade.aaa.joram.tcp.QueueTcpConnectionFactory`,
- `fr.dyade.aaa.joram.tcp.TopicTcpConnectionFactory`,
- `fr.dyade.aaa.joram.tcp.XATcpConnectionFactory`,
- `fr.dyade.aaa.joram.tcp.XAQueueTcpConnectionFactory`,
- `fr.dyade.aaa.joram.tcp.XATopicTcpConnectionFactory`.

All are constructed the same way. For example, let's consider the `TcpConnectionFactory` class.

fr.dyade.aaa.joram.tcp.TcpConnectionFactory

- Purpose: providing a TCP access to a given JORAM server.
- Constructor: `TcpConnectionFactory(String host, int port)`
  - *host*: name or IP address of the host on which the server is running.
  - *port*: port number the server is listening on.
- Failure: an `UnknownHostException` is thrown if the host is unknown.
- Example:

```
javax.jms.ConnectionFactory cnxFact =
  new fr.dyade.aaa.joram.tcp.TcpConnectionFactory("localhost",
                                                   16010);
```

Setting a `ConnectionFactory` instance timer parameters

A `ConnectionFactory` instance, apart from the server parameters it holds, also holds a connecting timer parameter and a transaction timer parameter.

The connecting timer parameter is used when trying to open a connection from the `ConnectionFactory`. Its value is the duration in seconds during which connecting is attempted if the server is down or not yet started.

Setting it:

```
// Getting the parameters holder:
fr.dyade.aaa.joram.FactoryParameters params =
  ((fr.dyade.aaa.joram.ConnectionFactory) cnxFact).getParameters();
// Setting the timer:
params.connectingTimer = 60;
```

The transaction timer (only available for non-XA factories) applies to the transacted sessions of the connections opened through the `ConnectionFactory`. Its value is the duration in seconds during which a session transaction may be pending. At the expiration of the timer, the transaction is automatically rolled back and closed.

Setting it:

```
// Getting the parameters holder:
fr.dyade.aaa.joram.FactoryParameters params =
  ((fr.dyade.aaa.joram.ConnectionFactory) cnxFact).getParameters();
// Setting the timer:
params.txPendingTimer = 120;
```

javax.jms.Destination

- Purpose: providing client access to a given JORAM destination.
- Constructor: `fr.dyade.aaa.joram.Destination(String agentId)`
    - *agentId*: identifier of the JORAM destination, obtained by calling the `getDestId()` method on the platform `CreateDestinationReply` reply.
- Example:

```
javax.jms.Queue queue =
  new fr.dyade.aaa.joram.Queue(reply.getDestId());
```

fr.dyade.aaa.joram.admin.User

In order to facilitate the management of users by administrators, a specific `User` class has been implemented.

- Purpose: allowing user administration.
- Constructor: `User(String name, String proxyId)`
    - *name*: name of the user.
    - *proxyId*: identifier of the user proxy, obtained by calling the `getProxId()` method on the platform `CreateUserReply` reply.
- Example:

```
fr.dyade.aaa.joram.admin.User user =
  new fr.dyade.aaa.joram.admin.User("userName", reply.getProxId());
```

Name space

Once created, administered objects should be put in a name space so that they can later be retrieved by JMS clients or administrators. The above objects are all referencable in a name space through the JNDI interfaces:

```
javax.naming.Context jndiCtx = new javax.naming.InitialContext();
jndiCtx.bind("myQueue", queue);
jndiCtx.bind("myUser", user);
jndiCtx.close();
```

Retrieving the objects is straightforward:

```
javax.naming.Context jndiCtx = new javax.naming.InitialContext();
javax.jms.Queue queue;
fr.dyade.aaa.joram.admin.User user;

queue = (javax.jms.Queue) jndiCtx.lookup("myQueue");
user = (fr.dyade.aaa.joram.admin.User) jndiCtx.lookup("myUser");
jndiCtx.close();
```

5.2.5. Example

The following code sample requests the creation of a JORAM queue and instanciates a JMS queue object.

```
   javax.jms.TopicConnectionFactory cnxFact;
   javax.jms.TopicConnection cnx;
   javax.jms.TopicSession sess;
   javax.jms.Topic adminTopic;
   javax.jms.TopicRequestor requestor;

   cnxFact =
     new fr.dyade.aaa.joram.tcp.TopicTcpConnectionFactory("hostName",
                                                          portNumber);
   cnx = cnxFact.createTopicConnection("adminName", "adminPass");
   sess = cnx.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
   adminTopic = sess.createTopic("#AdminTopic");
   requestor = new javax.jms.TopicRequestor(sess, adminTopic);

   javax.jms.ObjectMessage requestMsg;
   javax.jms.ObjectMessage replyMsg;
   fr.dyade.aaa.mom.admin.AdminRequest request;
   fr.dyade.aaa.mom.admin.AdminReply reply;

   cnx.start();

   // Building the request:
   requestMsg = sess.createObjectMessage();
   request  =  new  fr.dyade.aaa.mom.admin.CreateDestinationRequest(0,
null, "fr.dyade.aaa.mom.dest.Queue");
   requestMsg.setObject(request);

   // Sending it and expecting the reply:
   replyMsg = (javax.jms.ObjectMessage) requestor.request(requestMsg);

   // Getting the wrapped object
   reply = (fr.dyade.aaa.mom.admin.AdminReply) replyMsg.getObject();
   …
```

```
…
// If the request succeeded, building the JMS queue:
if (reply.succeeded()) {
  fr.dyade.aaa.mom.admin.CreateDestinationReply destRep;
  destRep = (fr.dyade.aaa.mom.admin.CreateDestinationReply) reply;
  String destName = destRep.getDestId();
  javax.jms.Queue queue = new fr.dyade.aaa.joram.Queue(destName);
}
// Otherwise, throwing an exception:
else
  throw new Exception(reply.getInfo());

// Closing the administrator's connection:
cnx.close()
```

5.3. Using the `AdminItf` interface

Examples involving the `AdminItf` interface can be found in the JORAM samples. All administration codes use the interface methods.

Joram is provided with a reference implementation of the `AdminItf` interface. This class is the `fr.dyade.aaa.joram.admin.AdminImpl` class.

5.3.1. Connecting an administrator

An administrator must first instanciate a class implementing the `AdminItf` interface:

```
AdminItf admin = new fr.dyade.aaa.joram.admin.AdminImpl();
```

It must then open a connection with the platform, locally, with identification *root* – *root* and connection timer set to *60* seconds:

```
admin.connect("root", "root", 60);
```

or remotely, to a server running on *host1* and listening to port *16010*, with identification *root* – *root* and connection timer set to *60* seconds:

```
admin.connect("host1", 16010, "root", "root", 60);
```

These methods will work if the following service is provided by the server either running on *localhost*, or by the server running on *host1* and listening to port *16010* (see section 3.2.2):

```
<service class="fr.dyade.aaa.mom.proxies.tcp.TcpConnectionFactory"
         args="16010 root root"/>
```

The last parameter of the connecting methods (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.

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`.

For disconnecting an administrator:

```
admin.disconnect();
```

5.3.2. Stopping a server

A connected administrator can stop any server of the platform. If it stops the server it is connected to, its session is automatically terminated and closed.

Stopping server 0:

```
admin.stopServer(0);
```

5.3.3. Managing a user

Setting a user

- `createUser(String name, String password, int server)`: sets a user with a given identification as a TCP user on a given server, and instanciates a JORAM `User` instance, representing the user client side.
- `createUser(String name, String password)`: sets a user with a given identification as a TCP user on the server the administrator is connected to, and instanciates a JORAM `User` instance, representing the user client side.

```
fr.dyade.aaa.joram.admin.User user;
user = admin.createUser("hisName", "hisPass", 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 connection with the server is lost.

Updating a user

- `updateUser(User user, String newName, String newPass)`: updates a given user identification.

```
admin.updateUser(user, "hisNewName", "hisNewPass");
```

An `AdminException` is thrown if the user does not exist server side, or if its new identification is already taken on its server. A `ConnectException` is thrown if the connection with the server is lost.

Deleting a user

- `deleteUser(User user)`: unsets a given user.

```
admin.deleteUser(user);
```

The request is not effective if the user does not exist server side. A `ConnectException` is thrown if the connection with the server is lost.

5.3.4. Managing a destination

Creating a destination

- `createDestination(int server, String name, String className, Properties prop)`: creates or retrieves a given destination on a given server with a given name, and initializes it with given properties. Returns the String identifier of the JMS `Destination`.

- `createDestination(int server, String className, Properties prop)`: creates a given destination on a given server, and initializes it with given properties. Returns the String identifier of the JMS `Destination`.
- `createDestination(int server, String className)`: creates a given destination on a given server, returns the String identifier of the JMS `Destination`.

```
String destId =
  admin.createDestination(0, "fr.dyade.aaa.mom.dest.Queue");
javax.jms.Queue queue = new fr.dyade.aaa.joram.Queue(destId);
```

- `createQueue(int server)`: creates a queue on a given server, instanciates the corresponding JMS `Queue`.
- `createQueue()`: creates a queue on the server the administrator is connected to, instanciates the corresponding JMS `Queue`.

```
javax.jms.Queue queue = admin.createQueue(0);
```

- `createTopic(int server)`: creates a topic on a given server, instanciates the corresponding JMS `Topic`.
- `createTopic()`: creates a topic on the server the administrator is connected to, instanciates the corresponding JMS `Topic`.

```
javax.jms.Topic topic = admin.createTopic(1);
```

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 connection with the server is lost.

Setting free access on a destination

- `setFreeReading(javax.jms.Destination dest)`: grants the READ right to all on a given destination.

```
admin.setFreeReading(queue);
```

- `setFreeWriting(javax.jms.Destination dest)`: grants the WRITE right to all on a given destination.

```
admin.setFreeWriting(queue);
```

An `IllegalArgumentException` is thrown if the destination parameter is not a valid JORAM destination. An `AdminException` is thrown if the destination has been deleted. A `ConnectException` is thrown if the connection with the server is lost.

Unsetting free access on a destination

- `unsetFreeReading(javax.jms.Destination dest)`: removes the READ right to all on a given destination.

```
admin.unsetFreeReading(queue);
```

- `unsetFreeWriting(javax.jms.Destination dest)`: removes the WRITE right to all on a given destination.

```
admin.unsetFreeWriting(queue);
```

An `IllegalArgumentException` is thrown if the destination parameter is not a valid JORAM destination. An `AdminException` is thrown if the destination has been deleted. A `ConnectException` is thrown if the connection with the server is lost.

Setting a right for a user on a destination

- `setReader(User user, javax.jms.Destination dest)`: sets a given user as a reader on a given destination.

```
admin.setReader(user, queue);
```

- `setWriter(User user, javax.jms.Destination dest)`: sets a given user as a writer on a given destination.

```
admin.setWriter(user, queue);
```

An `IllegalArgumentException` is thrown if the destination parameter is not a valid JORAM destination. An `AdminException` is thrown if the destination or the user does not exist server side. A `ConnectException` is thrown if the connection with the server is lost.

Unsetting a right for a user on a destination

- `unsetReader(User user, javax.jms.Destination dest)`: unsets a given user as a reader on a given destination.

```
admin.unsetReader(user, queue);
```

- `unsetWriter(User user, javax.jms.Destination dest)`: unsets a given user as a writer on a given destination.

```
admin.unsetWriter(user, queue);
```

An `IllegalArgumentException` is thrown if the destination parameter is not a valid JORAM destination. An `AdminException` is thrown if the destination does not exist server side. A `ConnectException` is thrown if the connection with the server is lost.

Deleting a destination

- `deleteDestination(javax.jms.Destination dest)`: deletes a given destination.

```
admin.deleteDestination(queue);
```

An `IllegalArgumentException` is thrown if the destination parameter is not a valid JORAM destination. The request is not effective if the destination does not exist server side. A `ConnectException` is thrown if the connection with the server is lost.

5.3.5. Creating a **ConnectionFactory** instance

A dedicated `ConnectionFactory` instance is mandatory for connecting to a given server.

Creating a **ConnectionFactory** instance

- `createConnectionFactory(String host, int port)`: creates a `ConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `createConnectionFactory()`: creates a `ConnectionFactory` instance for accessing the server the administrator is connected to.

```
javax.jms.ConnectionFactory cnxFact =
  admin.createConnectionFactory("localhost", 16010);
```

- `createQueueConnectionFactory(String host, int port)`: creates a `QueueConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `createQueueConnectionFactory()`: creates a `QueueConnectionFactory` instance for accessing the server the administrator is connected to.

```
javax.jms.QueueConnectionFactory cnxFact =
  admin.createQueueConnectionFactory("localhost", 16010);
```

- `createTopicConnectionFactory(String host, int port)`: creates a `TopicConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `createTopicConnectionFactory()`: creates a `TopicConnectionFactory` instance for accessing the server the administrator is connected to.

```
javax.jms.TopicConnectionFactory cnxFact =
  admin.createTopicConnectionFactory("localhost", 16010);
```

- `createXAConnectionFactory(String host, int port)`: creates an `XAConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `createXAConnectionFactory()`: creates a `XAConnectionFactory` instance for accessing the server the administrator is connected to.

```
javax.jms.XAConnectionFactory cnxFact =
  admin.createXAConnectionFactory("localhost", 16010);
```

- `createXAQueueConnectionFactory(String host, int port)`: creates a `XAQueueonnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `createXAQueueConnectionFactory()`: creates an `XAQueueConnectionFactory` instance for accessing the server the administrator is connected to.

```
javax.jms.XAQueueConnectionFactory cnxFact =
  admin.createXAQueueConnectionFactory("localhost", 16010);
```

- `createXATopicConnectionFactory(String host, int port)`: creates a `XATopicConnectionFactory` instance for accessing a server running on a given host and listening to a given port.
- `createXATopicConnectionFactory()`: creates an `XATopicConnectionFactory` instance for accessing the server the administrator is connected to.

```
javax.jms.XATopicConnectionFactory cnxFact =
  admin.createXATopicConnectionFactory("localhost", 16010);
```

Setting a **`ConnectionFactory`** instance timer parameters

A `ConnectionFactory` instance, apart from the server parameters it holds, also holds a connecting timer parameter and a transaction timer parameter: see section 5.2.4.

5.3.6. 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:

```
javax.jms.Topic news = admin.createTopic(0);
javax.jms.Topic business = admin.createTopic(0);
javax.jms.Topic sports = admin.createTopic(0);
javax.jms.Topic tennis = admin.createTopic(0);
javax.jms.Topic soccer = admin.createTopic(0);
```

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

- `setFather(javax.jms.Topic father, javax.jms.Topic son)`: sets a given topic as the father of an other topic.

Going back to our example:

```
admin.setFather(news, business);
admin.setFather(news, sports);
admin.setFather(sports, tennis);
admin.setFather(sports, soccer);
```

An `IllegalArgumentException` is thrown by this latest method if one of the topics parameters is not a valid JORAM topic. An `AdminException` is thrown if one of the topics does not exist server side, or is already part of a cluster, or if the son parameter already has a father. A `ConnectException` is thrown if the connection with the server is 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:

- `unsetFather(javax.jms.Topic topic)`: unsets the father of a given topic.

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

```
admin.unsetFather(sports);
```

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.

An `IllegalArgumentException` is thrown by this latest method if the topic parameter is not a valid JORAM topic. A `ConnectException` is thrown if the connection with the server is lost. An `AdminException` is thrown if the topic parameter does not exist 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:

```
admin.deleteDestination(sports);
```

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

5.3.7. 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:

```
javax.jms.Topic topic0 = admin.createTopic(0);
javax.jms.Topic topic1 = admin.createTopic(1);
javax.jms.Topic topic2 = admin.createTopic(2);
```

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

- `setCluster(javax.jms.Topic clusterTopic, javax.jms.Topic joiningTopic)`: 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:

```
admin.setCluster(topic0, topic1);
admin.setCluster(topic0, topic2);
```

An `IllegalArgumentException` is thrown by this latest method if one of the topics parameters is not a valid JORAM topic. An `AdminException` is thrown if one of the topics does not exist 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 connection with the server is 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:

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

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

```
admin.leaveCluster(topic2);
```

An `IllegalArgumentException` is thrown if the topic parameter is not a valid JORAM topic. A `ConnectException` is thrown if the connection with the server is lost. An `AdminException` is thrown if the topic does not exist server side, or is not part of any cluster.

This method is similar to removing the topic representant through the `admin.deleteDestination` method, except that it does not remove the topic. It simply becomes independent.

5.3.8. Managing a dead message queue

Creating a dead message queue

- `createDeadMQueue(int server)`: creates a DMQ on a given server, and instanciates a JORAM `DeadMQueue` object, representing the DMQ client side.
- `createDeadMQueue()`: creates a DMQ on the server the administrator is connected to, and instanciates a JORAM `DeadMQueue` object, representing the DMQ client side.

```
fr.dyade.aaa.joram.admin.DeadMQueue dmq =
  admin.createDeadMQueue("dmq", 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 connection with the server is lost.

Setting a dead message queue

- `setDefaultDMQ(int serverId, DeadMQueue dmq)`: sets a given DMQ as the default DMQ for the destinations and users on a given server.

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

A ConnectException is thrown if the connection to the server is lost. An AdminException is thrown if the server is not known in the platform, or if the DMQ does not exist server side.

- setDestinationDMQ(javax.jms.Destination dest, DeadMQueue dmq): sets a given DMQ as the DMQ for a given destination.

```
admin.setDestinationDMQ(topic, dmq);
```

An IllegalArgumentException is thrown if the destination parameter is not a valid JORAM destination. An AdminException is thrown if the destination does not exist server side. A ConnectException is thrown if the connection to the server is lost.

- setUserDMQ(User user, DeadMQueue dmq): sets a given DMQ as the DMQ for a given user.

```
admin.setUserDMQ(user, dmq);
```

An AdminException is thrown if the user does not exist server side. A ConnectException is thrown if the connection to the server is lost.

Unsetting a dead message queue

- unsetDefaultDMQ(int serverId): unsets the default DMQ for the destinations and users on a given server.

```
admin.unsetDefaultDMQ(0);
```

An AdminException is thrown if the server is not known in the platform. A ConnectException is thrown if the connection to the server is lost.

- unsetDestinationDMQ(javax.jms.Destination dest): unsets the DMQ of a given destination.

```
admin.unsetDestinationDMQ(topic);
```

An IllegalArgumentException is thrown if the destination parameter is not a valid JORAM destination. An AdminException is thrown if the destination does not exist server side. A ConnectException is thrown if the connection to the server is lost.

- unsetUserDMQ(User user): unsets the DMQ of a given user.

```
admin.unsetUserDMQ(user);
```

An AdminException is thrown if the user does not exist server side. A ConnectException is thrown if the connection to the server is lost.

Setting a threshold value

- setDefaultThreshold(int serverId, int threshold): sets a given value as the default threshold for the queues and users on a given server.

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

A ConnectException is thrown if the connection to the server is lost. An AdminException is thrown if the server is not known in the platform.

- setQueueThreshold(javax.jms.Queue queue, int threshold): sets a given value as the threshold for a given queue.

```
admin.setQueueThreshold(queue, 5);
```

An IllegalArgumentException is thrown if the queue parameter is not a valid JORAM queue. An AdminException is thrown if the queue does not exist server side. A ConnectException is thrown if the connection to the server is lost.

- setUserThreshold(User user, int threshold): sets a given value as the threshold for a given user.

```
admin.setUserThreshold(user, 5);
```

An `AdminException` is thrown if the user does not exist server side. A `ConnectException` is thrown if the connection to the server is lost.

Unsetting a threshold value

- `unsetDefaultThreshold(int serverId)`: unsets the default threshold for the queues and users on a given server.

```
admin.unsetDefaultThreshold(0);
```

An `AdminException` is thrown if the server is not known in the platform. A `ConnectException` is thrown if the connection to the server is lost.

- `unsetQueueThreshold(javax.jms.Queue queue)`: unsets the threshold for a given queue.

```
admin.unsetQueueThreshold(queue);
```

An `IllegalArgumentException` is thrown if the queue parameter is not a valid JORAM queeu. An `AdminException` is thrown if the queue does not exist server side. A `ConnectException` is thrown if the connection to the server is lost.

- `unsetUserThreshold(User user)`: unsets the threshold for a given user.

```
admin.unsetUserThreshold(user);
```

An `AdminException` is thrown if the user does not exist server side. A `ConnectException` is thrown if the connection to the server is lost.

5.4. Using the graphical tool

The 3.7 release provides a new graphical administration tool entirely conceived and developed by Alexander Fedorowicz, contributor to JORAM.

This tool makes use of the `AdminItf` interface and `AdminImpl` class described in the previous section. It allows to administer a platform made of one or many interconnected servers.

The user manual is provided as an independent document reachable from the JORAM documentation page: http://joram.objectweb.org/doc/index.html.

6. JORAM SOAP mode

6.1. Introduction

In order to open JORAM's messaging platform to non TCP or non Java clients, the SOAP procotol may be used as client server communication protocol.

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 10), and acting as a router between Tomcat and the other servers of the platform.

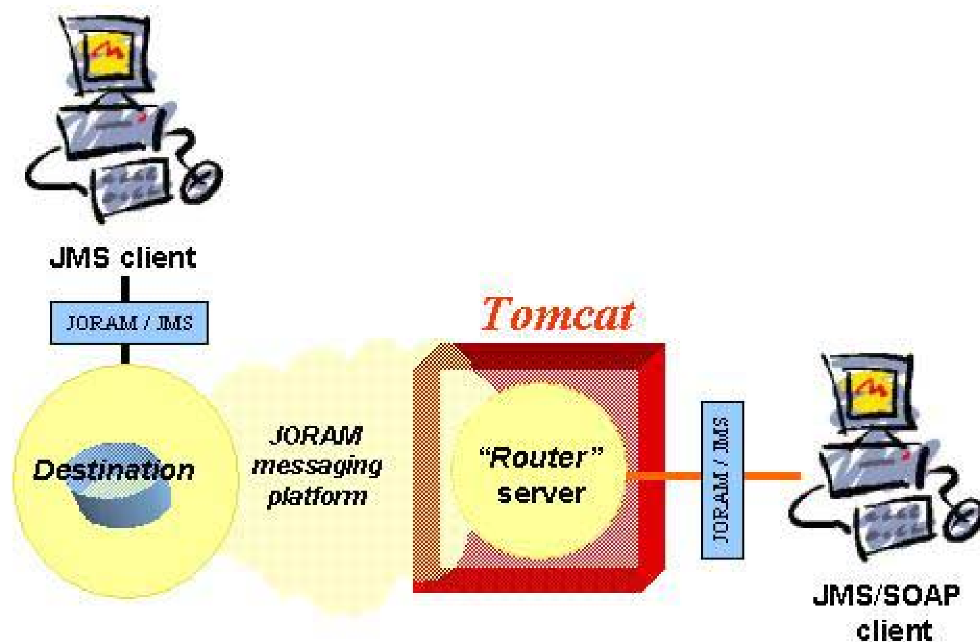

Figure 10: a JORAM platform providing a SOAP access

6.2. Platform configuration

6.2.1. The **SoapProxy** service

The SOAP proxy of the embedded server is declared as a service. This service is mandatory on the embedded server for accepting clients SOAP requests. It may be accessible to administrators also using the SOAP protocol, authenticated by a name and a password. At the platform level, this administrator gate is not mandatory, as the platform (including the embedded server) might be administered by a TCP administrator connecting through a TCP connection service.

6.2.2. Configuration file

Though it is possible to have a configuration made of a single embedded server, we will describe the configuration shown on figure 10. It is a reasonnable situation, with the server embedded in Tomcat's JVM (server **s1**) providing a SOAP proxy service, and the **s0** server providing a TCP connection service and hosting a destination.

The following file describes this configuration. Server 0 and server 1 both host a mandatory administration service, server 0 also hosts the mandatory name service and a JNDI service. Server 0 and server 1 accept administrator's connections with identification *root – root*.

To be noted, an additional *Transaction* property has been declared for server 1. It allows to set the transaction mode (persistent or not) of the embedded server.

For a non persistent server, change *fr.dyade.aaa.util.ATransaction* to *fr.dyade.aaa.util.NullTransaction*.

```
<?xml version="1.0"?>
<config>
<domain name="D1"/>
<server id="0" name="S0" hostname="localhost">
  <network domain="D1" port="16302"/>
  <service class="fr.dyade.aaa.ns.NameService"/>
  <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
  <service class="fr.dyade.aaa.mom.proxies.tcp.ConnectionFactory"
           args="16010 root root"/>
  <service class="fr.dyade.aaa.jndi2.server.JndiServer"
           args="16400"/>
</server>
```

```
  <server id="1" name="S1" hostname="localhost">
    <property name="Transaction"
              value="fr.dyade.aaa.util.ATransaction"/>
    <network domain="D1" port="16301"/>
    <service class="fr.dyade.aaa.mom.dest.AdminTopic"/>
    <service class="fr.dyade.aaa.mom.proxies.soap.SoapProxy"
             args="root root"/>
  </server>
  </config>
```

6.2.3. Running the platform

Starting a non embedded server

A non embedded server is started as any normal server (check §2.3). It must be started in a transaction mode (persistent or not) consistant with the embedded server's.

If **server 0** is not embedded, as for any distributed platform, it is recommended to **start it first**.

If server 0 actually is the embedded server, first start Tomcat and then server 0 as an embedded server as explained below.

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 document), and once the appropriate `a3servers.xml` configuration file has been put in Tomcat's `conf/` directory.

Starting an embedded server

The `fr.dyade.aaa.joram.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 instanciation 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
```

6.3. *Administering*

6.3.1. Introduction

Administering a platform providing a SOAP access is similar to administering a non SOAP platform. As for any classical TCP client, a SOAP client needs to be set as a user, and a `ConnectionFactory` object must be created for accessing the embedded server it is destinated to connect to.

6.3.2. Setting a SOAP user

As for a TCP user, setting a SOAP user might either be done by sending a special request wrapped into an `ObjectMessage` to the administration topic, or by using a new method of the `AdminItf` interface.

The `CreateSoapUserRequest`, to be sent in an `ObjectMessage`

- Purpose: requests the creation of a SOAP user.
- Constructor: `CreateSoapUserRequest(String userName, String userPass, int serverId)`
    - *userName*: name of the user.
    - *userPass*: password of the user.
    - *serverId*: identifier of the embedded server where creating the user.
- Failure: the request is not executed if the server does not belong to the platform; it fails if the user name is already taken server side, or if the target server does not host any SOAP proxy.

It is replied by a `CreateUserReply` instance. SOAP users might be updated or deleted like TCP users.

Setting a SOAP user with the `SoapExt_AdminItf` interface

The `AdminItf` interface presented in section 5.3 has been extended for providing SOAP administration methods.

Examples involving the `SoapExt_AdminItf` interface can be found in the JORAM SOAP sample. The administration code uses the interface methods.

Joram is provided with a reference implementation of the `SoapExt_AdminItf` interface. This class is the `fr.dyade.aaa.joram.admin.SoapExt_AdminImpl` class.

A new method has been added for setting a SOAP user. This method is the `createSoapUser(String name, String password, int server)`. It sets a user with a given identification as a SOAP user on a given server, and instanciates a JORAM `User` instance, representing the user client side.

```
fr.dyade.aaa.joram.admin.User user;
user = admin.createSoapUser("hisName", "hisPass", 1);
```

An `AdminException` is thrown if the user creation fails server side. The request is not effective if the server is not part of the platform. A `ConnectException` is thrown if the connection with the server is lost.

The methods for updating or removing a TCP user also work for a SOAP user.

6.3.3. Creating a SOAP `ConnectionFactory` object

Creating a SOAP `ConnectionFactory` might either be done by direct instanciation, or through a method of the `SoapExt_AdminItf` interface.

Instanciating a SOAP `ConnectionFactory`

`ConnectionFactory` classes for the SOAP protocol are:

- `fr.dyade.aaa.joram.soap.SoapConnectionFactory`,
- `fr.dyade.aaa.joram.soap.QueueSoapConnectionFactory`,
- `fr.dyade.aaa.joram.soap.TopicSoapConnectionFactory`.

All are constructed the same way. For example, let's consider the `SoapConnectionFactory` class:

- Constructor: `SoapConnectionFactory(String host, int port, int timeout)`
  - *host*: name or IP address of the host hosting Tomcat.
  - *port*: Tomcat's HTTP port (generally 8080).
  - *timeout*: duration in seconds during which a SOAP connection might stay inactive server side before being considered as dead.
- Example:

```
java.jms.ConnectionFactory cnxFact =
  new fr.dyade.aaa.joram.soap.SoapConnectionFactory("localhost",
                                                     8080,
                                                     60);
```

The setting of the other timer parameters are done as described in §4.2.4.

Calling a **`SoapExt_AdminItf`** method

The following methods are provided for creating SOAP `ConnectionFactory` objects:

- `createSoapConnectionFactory(String host, int port, int timeout)`: 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 timeout value in seconds for pending SOAP connections.
- `createSoapConnectionFactory(int timeout)`: creates a `ConnectionFactory` instance for accessing a server embedded in the Tomcat JVM the administrator is connected to, with a given timeout value in seconds for pending SOAP connections.

```
javax.jms.ConnectionFactory cnxFact =
  admin.createSoapConnectionFactory("localhost", 8080, 60);
```

- `createQueueSoapConnectionFactory(String host, int port, int timeout)`: 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 timeout value in seconds for pending SOAP connections.
- `createQueueSoapConnectionFactory(int timeout)`: creates a `QueueConnectionFactory` instance for accessing a server embedded in the Tomcat JVM the administrator is connected to, with a given timeout value in seconds for pending SOAP connections.

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

- `createTopicSoapConnectionFactory(String host, int port, int timeout)`: 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 timeout value in seconds for pending SOAP connections.
- `createTopicoapConnectionFactory(int timeout)`: creates a `TopicConnectionFactory` instance for accessing a server embedded in the Tomcat JVM the administrator is connected to, with a given timeout 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 **timeout** parameter is set when creating a SOAP `ConnectionFactory`. It sets the duration in seconds during which a SOAP connection might be pending before being considered as dead. Then, the server 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.

6.3.4. Connecting a SOAP administrator

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

Connecting a TCP administrator to the platform configured in section 6.2.2 would look like:

```
javax.jms.ConnectionFactory cnxFact =
  new fr.dyade.aaa.joram.tcp.TcpConnectionFactory("localhost",
                                                  16010);
javax.jms.Connection cnx;
cnx = cnxFact.createConnection("root", "root");
```

or:

```
SoapExt_AdminItf admin = new SoapExt_AdminImpl();
admin.connect("localhost", 16010, "root", "root", 60);
```

Connecting a SOAP administrator looks like:

```
javax.jms.ConnectionFactory cnxFact =
  new fr.dyade.aaa.joram.soap.SoapConnectionFactory("localhost",
                                                    8080,
                                                    60);
javax.jms.Connection cnx;
cnx = cnxFact.createConnection("root", "root");
```

or:

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

SoapExt_AdminItf admin = new SoapExt_AdminImpl();
admin.connect(cnxFact, "root", "root");
```

6.3.5. 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*.