Livrable Selfware SP4

Lot 1, May 14, 2007

JMS Management Scenarios

Authors: Roland Balter (ScalAgent D.T.)
Noël de Palma (INRIA Sardes)
André Freyssinet (ScalAgent D.T.)
Daniel Hagimont (IRIT)

Contents

Introduction.

This document aims at describing a first set of autonomous management scenarios for the Joram JMS infrastructure.

Autonomic systems are inspired by the autonomic nervous system of the human body. This nervous system controls important bodily functions (e.g. respiration, heart rate, and blood pressure) without any conscious intervention.

In a self-managing autonomic system, the human operator takes on a new role: he does not control the system directly. Instead, he defines general policies and rules that serve as an input for the self-management process. For this process, IBM has defined the following four functional areas:

- Self-Configuration: Automatic configuration of components;
- Self-Healing: Automatic discovery, and correction of faults;
- Self-Optimization: Automatic monitoring and control of resources to ensure the optimal functioning with respect to the defined requirements;
- Self-Protection: Proactive identification and protection from arbitrary attacks.

This paper is organized as follows. Section 2 presents the messaging context, it describes the JMS specification and the Joram implementation. Section 3 gives the motivations for using autonomous management then describes the two use cases. Section 4 concludes briefly.

1. Overview of JMS and Joram

Message-Oriented Middleware (MOM) is increasingly being seen as a key to improve the enterprise productivity and to facilitate the open services market. Today enterprises are faced with the challenges of time-to-market, data distribution, application integration and business flexibility in the context of loosely-coupled distributed systems encountered in multi-organization environments over the Internet:

- Data integration, E-business, EAI and B2B solutions,
- Mobile users and ubiquitous systems,
- Networked devices management solutions: energy, building/home automation, RFID, etc.

Up to now, the lack of standard has been a strong obstacle to a wide adoption of asynchronous communication systems as a technical basis for implementing interoperability. Over the past few years the JMS™ API (Java Messaging System) has partially filled in this gap and has become a de facto standard for building cooperating software components over Internet based loosely-coupled distributed environments.

Joram is an open-source implementation of the JMS™ messaging specification. Joram is fully compliant with JMS 1.1 and is available on a large range of computer systems, from application servers to internet appliances.

JORAM greatly benefits from the new generation Message Oriented Middleware from *ScalAgent Distributed Technologies*, an agent-based truly distributed architecture. The underlying innovative architecture allows distributed applications to be connected on a large-scale basis through Internet, enables load balancing and guarantees high availability and flexibility.

JORAM is available under the LGPL license, it is a software component available from ObjectWeb (http://joram.objectweb.org).

1.1. The JMS Specification

JMS (*Java Messaging Service*) is the specification of a messaging service for Java applications. More precisely JMS describes the API that allows Java programs to communicate through an asynchronous communication system – i.e. sending and receiving messages. A detailed description of the JMS API is beyond the scope of this report, we present here only the JMS elements that are required to the understanding of architecture issues described later on.

A JMS application consists of the following elements (see Figure 1):

- A **JMS platform** (usually called JMS provider) that provides the JMS run-time environment and a set of control and administrative functions.
- The **JMS clients** are application programs, written in Java, that produce and consume messages according to the messaging protocols defined in the JMS API.
- **JMS messages** are entities that allow information to be conveyed between JMS clients. Different types of messages are supported in JMS: structured text (e.g. an XML file), binary data, java objects, etc.

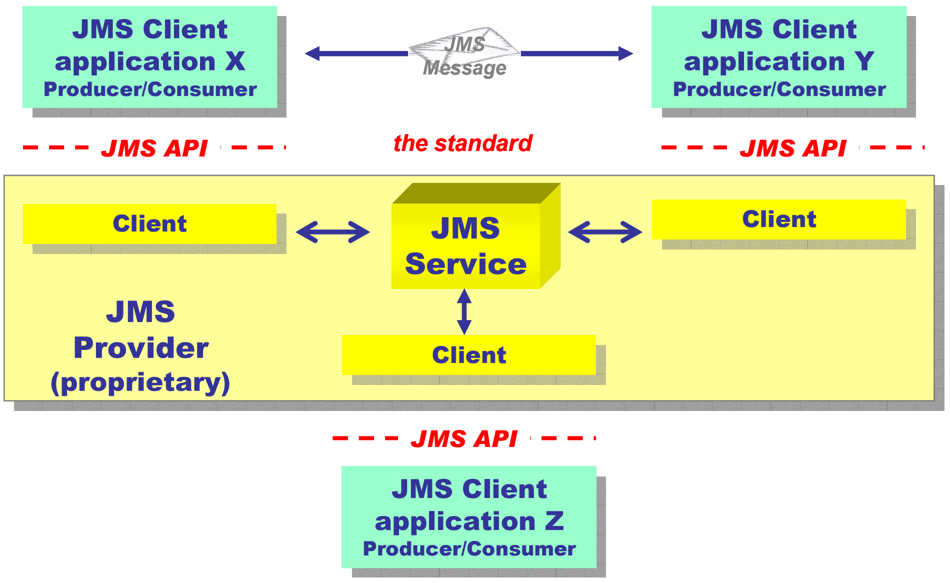
JMS Client application X Producer/Consumer
JMS Message
JMS Client application Y Producer/Consumer
JMS API
the standard
JMS API
Client
JMS Service
Client
JMS Provider (proprietary)
Client
JMS API
JMS Client application Z Producer/Consumer

Figure 1 - JMS Application

Two communication models (called *Messaging Domains* in JMS) are supported:

- **Point to Point model,** based on message queues. A producer client sends a message to a message Queue where it is stored temporarily. A consumer client may then read the message from the Queue to operate on it. A given message is read only by a single consumer client (this explains the "point to point" designation). The message stays in the Queue until it is read or the message time-to-live expires. The message consumption may be synchronous (explicit call by the consumer client) or asynchronous (call of a pre-defined watch function to be executed at the consumer site). Message consumption is then acknowledged either by the system or by the client application.

- **Multipoint model**, based on the **Publish/Subscribe** paradigm. A producer client publishes a message related to a pre-defined **Topic**. All clients that have previously subscribed to this topic are notified of the corresponding message.

The latest version of the JMS specification (i.e. JMS 1.1.) unifies the handling of the two communication models at the client level through the introduction of the **Destination** concept to represent a queue or a topic. This simplifies the API (i.e. the queue and topic functions are syntactically merged) and allows an optimization of communication resources. This unification does not change the semantics of each of the communication models that should be taken into account when programming the JMS client.

The JMS specification is not complete. Some key functions are not described, such as for example the administration of a JMS platform (i.e. deployment, configuration, monitoring, etc.) and are thus subject to proprietary implementations. At the opposite, most of the available products provide additional functions such as load-balancing and high availability features.

1.2. Joram overview

From the above description JORAM is a messaging component that complies with the JMS specification. As any other JMS platform JORAM is structured in two parts: the JORAM server that manages the JMS abstractions (e.g. *queues and topics*) and the JORAM client that is bound with the JMS client application.

As we will see later on in the description of the JORAM architecture, the JORAM server can be implemented as a central service or as a set of cooperating distributed services. It should be noted here that the ability to deploy a JMS platform as a distributed system with variable QoS parameters is a key issue when comparing various JMS platforms.

Communication between a JORAM client and a JORAM server is generally relying on TCP/IP. Usually, clients and servers run on different machines. However they also can be hosted on the same machine or share the same process. In this case communications are optimized.

The sections below detail the key aspects of the JORAM platform architecture.

Design Choices

The main characteristic of JORAM is its distributed configurable architecture. Basically JORAM has adopted a **snowflake** architecture - i.e. a JORAM platform is composed of a set of JORAM servers interconnected by a message bus that offers various communication protocols. Each server can manage a variable number of JMS clients.

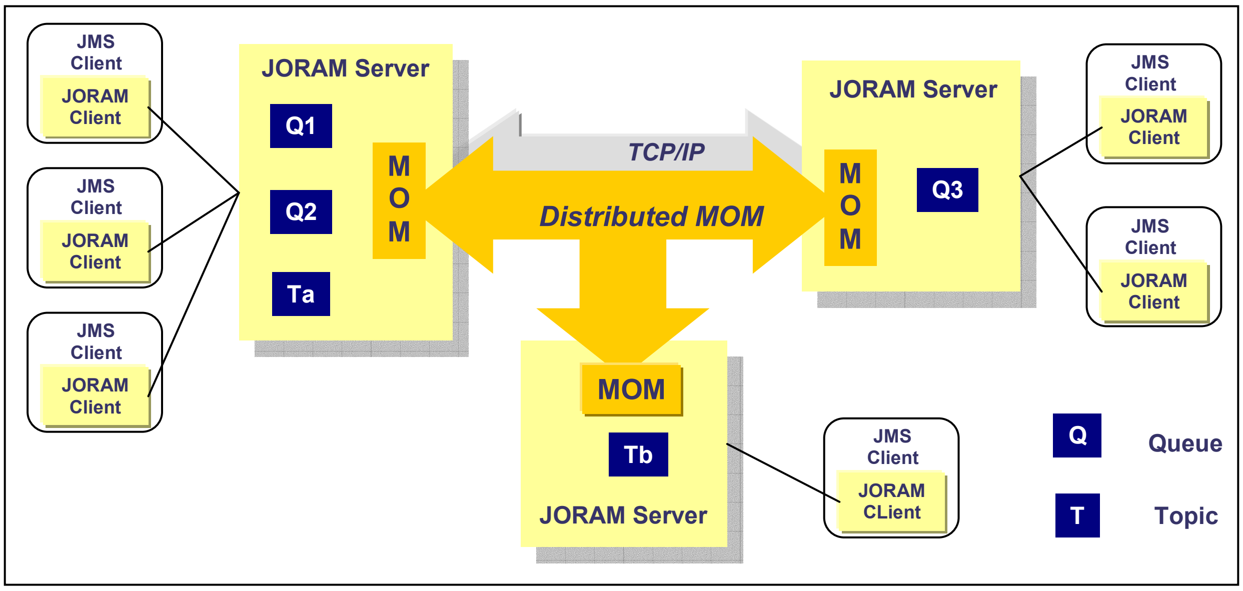

Figure 2 - Joram platform architecture

The geographical location of the servers as well as the distribution of the clients on the various servers are under the responsibility of the architect and are thus enforced by the platform administrator. This choice is a first level of configuration. A second level deals with the location of communication objects (Queues and Topics). Note that these location parameters have a deep impact on performance figures, scalability and availability issues. A third level of configuration consists in defining the QoS parameters (communication protocol, persistency, security, etc.). The choice of a given mechanism is always a trade-off between the expected QoS level and the cost of the solution. The overall architecture of a JORAM platform is depicted in Figure 2.

Logical Architecture

This section describes the principles of a communication path between two JMS clients for both the Point to Point and Publish/Subscribe communication models. Distribution is not depicted in this functional description (i.e. in a given implementation the Destination and Proxy objects may run on different servers).

Connection, ***Session*** and ***Sender*** (respectively ***Receiver***) are temporary JMS objects created by a JORAM client when a connection is established with the server.

On the server each JMS client is represented by a persistent ***Proxy*** object that is created by the server whenever a new client connects. A proxy object implements basically two functions:

Communications management between the server and the client.

Message delivery to (resp. message retrieval from) the destination.

Point-To-Point model

The communication between a producer client and a consumer client is achieved through the following steps (described in **Erreur ! Source du renvoi introuvable.** by the red arrows).

When a Send operation is executed on the producer client site, a JMS message is sent to the corresponding Proxy object within the server. The Proxy object encapsulates the JMS message into a reliable MOM message to be transported by the MOM to the target destination. An acknowledgment is returned to the JMS client. As seen from the producer client the Send operation is over and the client may continue its execution asynchronously while the message is actually delivered.

The MOM message is delivered to the Queue object. This may require a communication between servers if the Proxy and the Queue Objects are not handled by the same server.

When a Receive operation is executed at the consumer client site a control message is sent to its corresponding proxy object in the server which forwards it to the Queue object. The message is then retrieved from the Queue object and sent to the Proxy object which, in turn, extracts the JMS message from its envelope and returns it to the consumer client.

An acknowledgment is returned by the JORAM client to the Queue object in order to free the resources on the server that manages the Queue object. The acknowledgment may be performed explicitly by the JMS client or generated by the JORAM client.

Publish/Subscribe Model

The sending mechanism is similar with that of the point to point communication model. The topic stores the identity of the proxy object for each client that has subscribed. When the MOM message is delivered to the Topic object, it is forwarded directly to the whole set of consumer proxy objects where it is stored. This is a key difference with the prior schema as the Topic object is not a final destination but merely a switch/router towards the set of consumer proxy objects.

The consumer operation is implemented by a simple exchange between the client and its Proxy object. By comparison with the preceding schema it can be noted that the consuming dialog is restricted to exchanges between the client and its proxy object. No dialog occurs with the Topic object itself. This usually leads to shorter interactions and better performance from the consumer point of view.

The next two following sections describe how this logical schema is implemented in a centralized architecture (i.e. a single JORAM server) in a first step, and in a distributed architecture in a further step.

Centralized Architecture

In this configuration, all JMS clients are connected to a single JORAM server that manages all destination and proxy objects. The communication protocol is simplified as all communicating objects are located on the same server. Administration and operation are also greatly simplified.

This centralized solution has two major drawbacks: lack of availability and lack of scalability. The server is a single-point of failure that prevents the whole system to run if a failure occurs on the server side. On the other hand, computing and storage resources being concentrated on a single machine, this may result in serious performance bottlenecks when the load increases (e.g. number of clients, number and size of messages, etc.).

Distributed Architecture

In a distributed architecture approach, several instances of the JORAM server cooperate, each of them being in charge of a set of JMS clients. This architecture is illustrated in Figure 3 for a scenario composed of three physical servers (to simplify the figure, only one JMS client has been represented at each location).

The physical communication between servers is achieved by the underlying distributed MOM that guarantees the delivery of messages at the MOM level despite transient network and server failures.

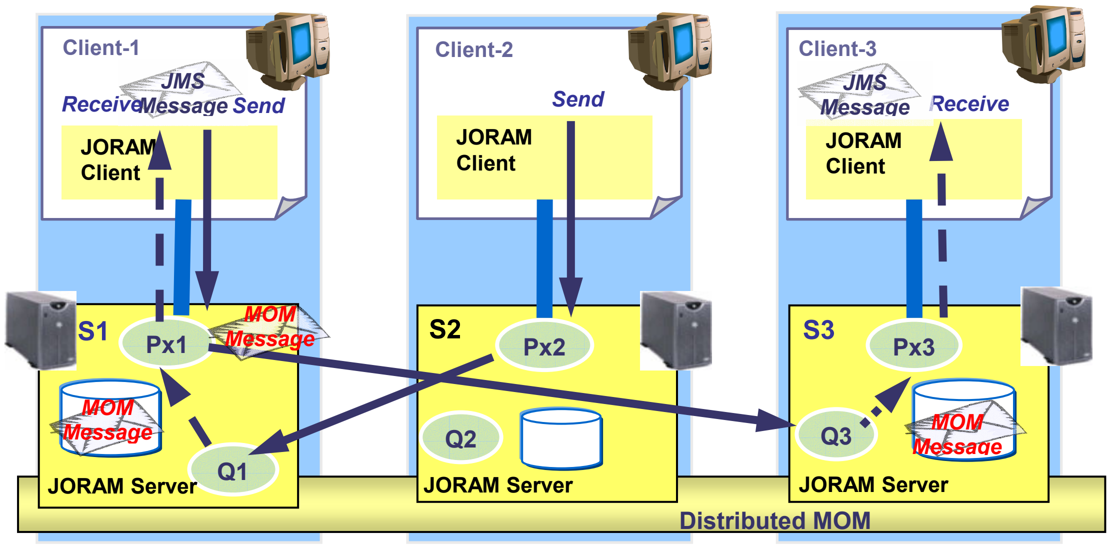

Figure 3 - Distributed architecture

A distributed JORAM platform involves several JORAM servers. To interoperate, these servers must belong to a given administration domain and the following JORAM services have to be available on the servers:

- A connection manager to manage the connections with the clients connected to that server using a set of user proxies.
- A user proxy for each client allowing the interaction with the platform.
- A set of destinations, Queue and/or Topic.

In order to deploy the architecture described above we have to describe each server with its own services.

JORAM: a Highly Configurable JMS Platform

Building a JMS platform tailored for a given application context is a difficult task as the resulting architecture is based on complex tradeoffs between numerous evaluation criteria such as: performance, availability, scalability, flexibility and evolution, security, development and operation costs, etc.

Therefore the role of the architect is crucial. Based on a set of application requirements he has to take into account the evaluation criteria mentioned above and to give them a specific weight in order to design the architecture that better fits the requirements. This work is achievable if and only if the messaging system provides the configuration capabilities that enable such a design process. JORAM answers this need through a combination of configuration and tuning capabilities available at various levels:

- Overall organization of JMS servers and clients. As depicted in Figure 2 a JORAM platform is structured according to a “snowflake” architecture. This approach gives to the application architect the freedom to settle servers where they are needed – e.g. to locally serve a set of geographically distributed JMS clients (edge computing), or to answer security constraints, or any other relevant criteria.

- Positioning of JMS objects. The location of JMS communication objects (i.e. queues and topics) is a key issue as it has a strong impact on the overall performance as well as on client availability.
- Server sizing (computing power, storage, communication facilities). The JORAM Web Site provides some benchmarking figures that can help application designers to anticipate the sizing of the various JORAM servers to support the load generated by their local clients.
- System extension to support scalability. This parameter refers to the ability for a JORAM platform to grow dynamically to answer evolving application requirements (e.g. new messaging server, increasing load, etc.). Management facilities are provided to remotely add and remove JORAM servers dynamically through the Administration API of JORAM.
- Communication protocols. Various communication protocols are available between clients and their server as well as between two servers: TCP/IP, HTTP, SSL, SOAP, etc.
- QoS parameters (e.g. persistency, security). Message transfer reliability (i.e. guarantee of message delivery) and security (message confidentiality) have a cost and the application designer can select the right option for his application.
- Level of availability through clustering and replication. JORAM answers the needs of mission critical applications with the ability to design clustered destinations and to deploy highly available JORAM servers. Theses features are briefly introduced in the next section.

Today very few platforms provide a level of flexibility comparable to that of JORAM. This is obviously a major advantage compared to concurrent products, but a factor of complexity. The SelfWare autonomic platform should allow the Joram platform to be automatically and dynamically adapted based on actual application needs.

Clustered queue

The clustered queue feature provides a load balancing mechanism; this mechanism is the basis of the first demonstration scenario. A clustered queue is a cluster of queues (a given number of queue destinations, knowing each other), exchanging messages depending on their load.

Each queue of a cluster periodically re-evaluates its load factor and sends the result to the other queues within 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 asked for messages but is empty, it requests messages from the other queues within the cluster. This mechanism guarantees that no queue is over-active while some others are lazy, and leads to distribute the workload among the servers involved in the cluster.

For example let's consider a cluster made of two queues, named *queue0* and *queue1*. A heavy producer sends messages to its local queue (*queue0*). *Queue0* is also accessed by a consumer that is requesting few messages while *Queue1* is accessed by another consumer. *Queue0* quickly becomes loaded and decides to forward messages to the companion queue (*queue1*) of its cluster, which is not under heavy load. The result of this adaptation policy is that the consumer on *queue1* also gets messages, and *queue0* is no longer overloaded.

Joram High-Availability

Reliability here refers mainly to the guarantee of end-to-end message delivery between a producer and a consumer despite network and server failures. Reliability in JORAM is achieved by a combination of several mechanisms briefly introduced below:

- An acknowledge message between a client and its proxy object allows the communication between a client and the server to be secured.
- The *Store and Forward* function achieved by the client proxy allows the message to be saved before being forwarded to the destination server in a distributed configuration.
- Finally the underlying MOM guarantees the delivery of the message between two servers.

The High Availability feature of Joram is targeted at mission critical applications. This feature is achieved through an active replication mechanism. *Queue* and *Topic* objects are replicated on two JORAM servers running on two distinct machines. One of the servers (the master) executes client requests and propagates the operations to the slave server. If the master server crashes, the client automatically establishes a new connection with the slave server and continues to proceed without any delay. This configuration enables non-stop processing.

2. Scenarios

Today JORAM is deployed in numerous operational environments where it is used in two complementary ways:

- As an independent messaging system between applications running in a Java environment (J2EE to J2ME) on large scale network,
- As an asynchronous communication component integrated within a J2EE application server eventually in a clustered context. Following this schema JORAM is a key building block of the JOnAS application server also available at ObjectWeb - http://jonas.objectweb.org -.
- In addition Joram provides complementary functions for load-balancing and high availability features.

The mechanisms available in the SelfWare platform will allow extending these features. Intended extensions are described in the two scenarios below.

2.1. Self-Optimization

Joram is a highly configurable MOM. For each client application there is an optimal Joram architecture: number of deployed servers, location of servers related to clients location, location of user proxies and destinations.

Some applications are also very dynamic as the number of clients, their location and activity may evolve quickly in a non-predictable way. The proposed scenario defines a self-optimization algorithm to dynamically adapt the MOM infrastructure to the application activity. This mechanism is based on a distributed Joram architecture deployed on a cluster of computers. The application is made up of two types of components: message producers and message consumers.

The aim of this mechanism is to balance the load due to the interaction of the multiple clients on the different servers. The use of distributed destinations (*ClusteredQueue*) is expected to solve this issue transparently. The efficiency of this mechanism depends primarily on the distribution of client connections to the servers that manage an instance of the clustered queue. We describe below a solution that intends to optimize the distribution of client connections to the clustered queue.

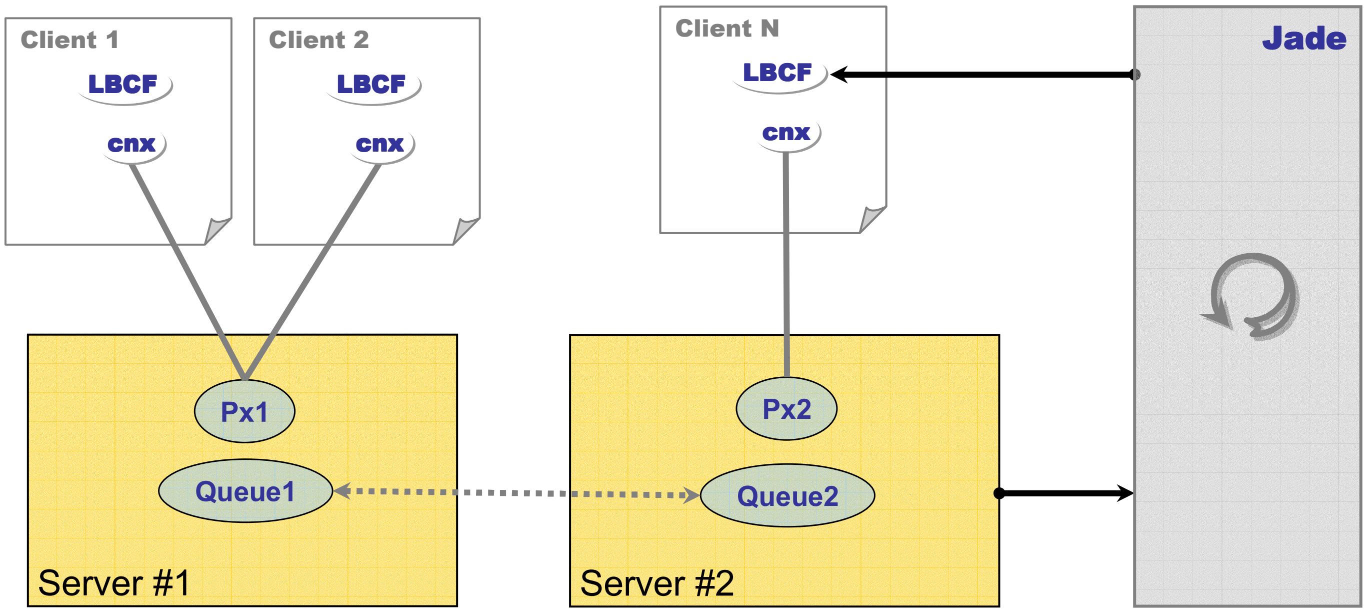

Figure 4

The optimization involves two steps: (i) optimal load-balancing for a clustered queue, and (ii) dynamic provisioning of queues within a clustered queue. The first part allows the overall improvement of the clustered queue performance while the second part optimizes the queue resources usage within the clustered queue. To summarize, the idea is then to create an autonomic system that:

- fairly distributes client connections to the pool of servers within the distributed configuration,
- dynamically adds, configures and removes queues in the system depending on the load. In fact, this would allow to adjust the number of queues at any time.

Implementing a self-managed queue cluster using the autonomic computing design principles requires the following capabilities:

- to know the current number of message producers and consumers, as well as the actual load for each of them,
- to know where the servers are, where the queues are deployed and what is their configuration,
- to route a new client connection to the best queue to reach the optimal,
- to detect the overload or the underload of a clustered queue,
- to allocate a new server to create a new instance of the queue,

- to add and remove a queue within a server.

Without modification, the underlying JMS middleware does not provide facilities such as session migration that would allow client connections to be moved from one queue to another.

We make here the assumption that each client is able to provide its profile (i.e. producer or consumer, number of messages per seconds, etc.). Assuming this, the mechanism is achieved by wrapping the standard JMS ConnectionFactory by a "LBConnectionFactory" (where LB stands for Load Balancing).

As the client gets the connection factory through JNDI, it gets the LBConnectionFactory instead (see Figure 4). This is the main non-functional hook in the system that allows the control of producers and consumers distribution among servers. This component provides the following methods:

- createConnection(...) takes the client's profile as a parameter to create the connection with the best server. It requests a component called "ClusterManager" which manages the cluster and elects a server according to the current state of the system (the servers, the load of each queue in terms of producers and consumers).
- closeConnection(...)closes the connection to the server and notifies the ClusterManager so it can update the state of the cluster.

It should be noted that all interactions between the client and the MOM conform to the JMS behaviour.

In a second step the load-balancing algorithm should be more dynamic with probes deployed on each server. The load balancer will then select the physical server using the real load of each server. To do this we can either use the existing Joram's monitoring[1], or use specific one:

- System probes about CPU, network or disk.
- Joram's extended probes: number of forwarded messages, size of internals message queues…
- The server state can be also checked by sending periodic requests; depending of the period of time needed for the reply the server load can be evaluated.

2.2. Self-Repair

Joram provides a high-availability functioning mode. An HA server is actually a group of servers –replica-, one of which is the master server that coordinates the other slave servers. Only the master server is accessible by the JMS clients, while the replicated servers maintain their state through an active duplication mechanism.

When the master server fails, one of the replica is chosen to replace it, it becomes the master. The HA Joram client owns the list of the replicas. When it detects that the

[1] Number of waiting messages in queue for example.

connection has failed it tries to reconnect. The new connection will be handled by the new master server and the client's exchange can go on.

The availability level of the HA logical server depends on the number of the active replicas. Each time a replica fails, the administrator must start a new one to ensure the availability. This scenario (Figure 5) proposes a self-repair mechanism allowing the detection of a replica failure, and then the deployment and starting of a new replica if the availability level has decreased under a predefined limit.

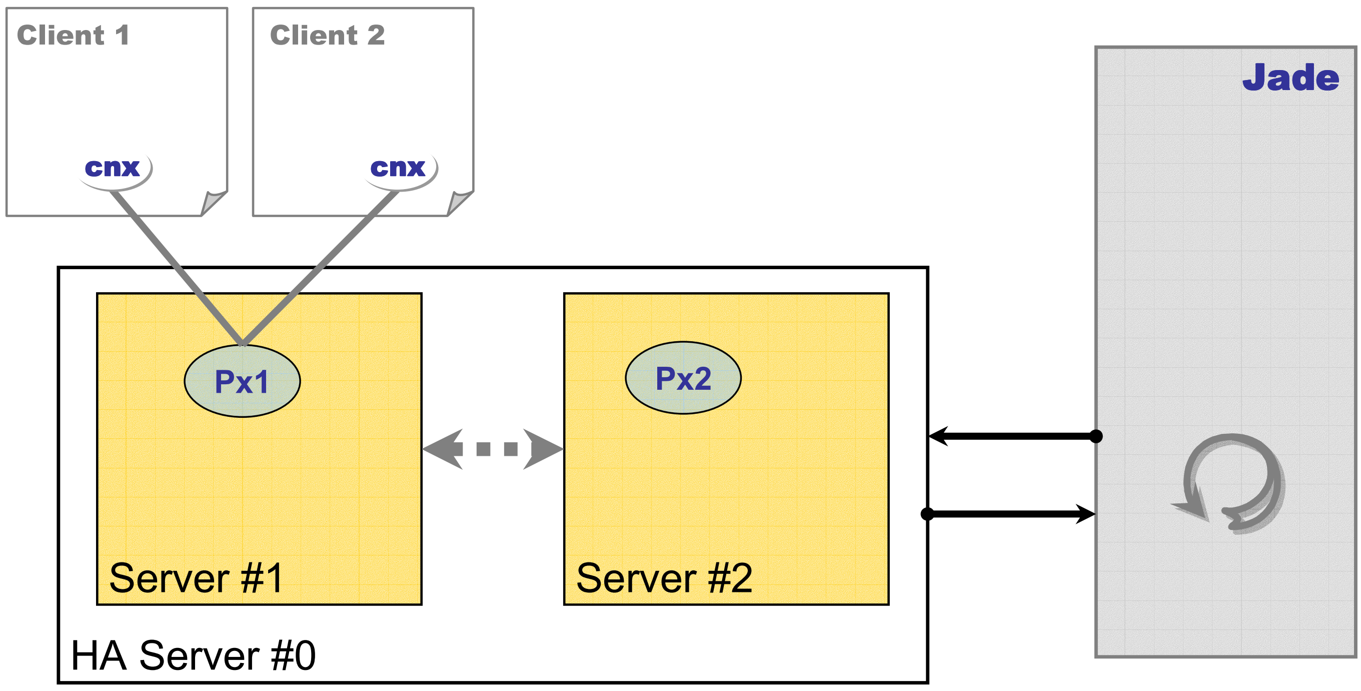

Figure 5

Implementing a self-repair HA Joram using the autonomic computing design principles requires the following capabilities:

- to detect the failure of a replica,
- to know the current number of replicas,
- to add and remove a replica within a HA configuration,
- to allocate a new server and to start a new replica.

This auto-repair feature can be implemented easily as the underlying Joram middleware provides facilities such as state replication that would allow a new replica to be created from a running one.

3. Conclusion

These two scenarios demonstrate the capabilities of the SelfWare platform to enhance the Joram messaging middleware. They need some additional functions in two domains:

- Monitoring: generic functions as ping or heartbeat, or Joram's specific functions in term of number of clients, number of waiting messages, etc.
- Configuration and deployment: creation of a server in the configuration, creation (resp. removal) of a destination, etc.

Their implementation should lead to a best comprehension of the capabilities of autonomous management and should allow the definition of other interesting scenarios. We describe below advanced scenario for Joram in three areas of the autonomous computing.

Self-configuration

Some Joram configuration includes thousands of nodes over a large and complex network. Building such Joram platform is difficult, time-consuming and error-prone even for experts. The resulting architecture is a trade-off between numerous criteria and constraints.

Using SelfWare, Joram will configure itself automatically in accordance with high level rules. Each new server will incorporate the configuration depending of its role and location; the network links between servers will be setup depending of the underlying network architecture. The overall platform will take in account its capabilities and other servers will adapt to its presence.

Self-optimization

As any complex middleware, Joram have hundreds of tunable parameters that must be set correctly for an optimal behaviour. Setting a parameter on a specific node can have unanticipated effects on the entire system; moreover the usage of the system evolves during the time.

Using SelfWare, a Joram's configuration will continually seek ways to improve its functioning. It will monitor constantly the performance and QOS indicators, experiment with parameters and tune them depending of the monitoring results.

Self-Healing

Determining the root cause of malfunctioning in operational Joram's configuration can be difficult. Identifying the issue can take time and the problem can disappears without any real diagnosis.

Using SelfWare, Joram will automatically detect problems and apply actions depending of the known policies:

- repair of bypass the issue,
- request additional information from the logging system,
- alert a human operator if needed.

For example, on a server failure, the system can either start a new one, or redeployed the server's functions on other existing servers.