**Livrable Selware SP4**

Lot 2, June 26, 2008, 18:1

# Selfware Self-management of JMS-based applications

S. Bouchenak (INRIA)
F. Boyer (INRIA)
F. Métral (INRIA)
A. Freyssinet (ScalAgent)
J. Philippe (INRIA)
S. Sicard (INRIA)
C. Taton (INRIA)

# Contents

# 1 Introduction

Autonomic computing, which aims at the construction of self-managing and self-adapting computer systems, has emerged as an important goal for many actors in the domain of large scale distributed environments and applications. This approach more precisely aims at providing systems and applications with self-management capabilities, including self-configuration (automatic configuration according to a specified policy), self-optimization (continuous performance monitoring), self-healing (detecting defects and failures, and taking corrective actions), and self-protection (taking preventive measures and defending against malicious attacks). Following this approach, the Selfware project aims at providing an infrastructure for developing autonomic management software. An important aspect of this infrastructure is the adoption of an architecture-based control approach as described in the SP1-L1 document, meaning that the control loops that regulate the system have the ability to introspect the current software architecture of the managed system, as well as they have the ability to modify (i.e. reconfigure) this architecture.

The objective of this document is to describe self-optimization and self-repair of JMS-based, or more generally MOM applications in the context of the Selfware platform. MOM applications rely on messages as the single structure for communication, coordination and synchronization, thus allowing asynchronous execution of components. Reliable communication is guaranteed by message queueing techniques that can be configured independently from the programming of software components. The Java community has standardized an interface for messaging (JMS).

This document more precisely analyses the performance of a MOM (JMS) and proposes a self-optimization algorithm to improve the performance of the MOM infrastructure. This mechanism is based on a queue clustering solution : a *clustered queue* is a set of queues each running on different servers and sharing clients.

*Note: as the work on self-repair is a bit late, its description is postponed to a future version of this delivrable.*

This paper targets the optimization of these clustering mechanisms. This optimization takes place in two parts: (i) the optimization of the clustered queue load-balancing and (ii) the dynamic provisioning of a queue in the clustered queue. The first part allows the overall improvement of the clustered queue performance while the second part optimizes the resource usage inside the clustered queue. Thus the idea is to create an autonomic system that:

- fairly distributes client connections among the queues belonging to the clustered queue,
- dynamically adds and removes queues in the clustered queue depending on the load. This would allow us to use the adequate number of queues at any time.

As the work on self-repair is a bit late, its description is postponed to a future version of this delivrable.

This document is organized as follow: Sections 2 and 3 present the context of this work. Section 4 details the different cases that may occur with a clustered queue. Sections 5 and 6 present the control rules and the control loop. Section 7 shows performance evaluation. Finally section 8 draws a conclusion and outlines future work.

## 2 Background: Java Message Service (JMS)

JMS is part of Sun's J2EE platform. It provide a programming interface (API) to interconnect different applications through a messaging middleware. The JMS architecture identifies the following elements:

- **JMS provider**: an implementation of the JMS interface for a Message Oriented Middleware (MOM). Providers are implemented as either a Java JMS implementation or an adapter to a non-Java MOM.
- **JMS client**: a Java-based application or object that produces and/or consumes messages.
- **JMS producer**: a JMS client that creates and sends messages.
- **JMS consumer**: a JMS client that receives messages.
- **JMS message**: an object that contains the data being transferred between JMS clients.
- **JMS queue**: a staging area that contains messages that have been sent and are waiting to be read. As the name queue suggests, the messages are delivered in the order they are sent. A message is removed from the queue once it has been read.
- **JMS topic**: a distribution mechanism for publishing messages that are delivered to multiple subscribers.
- **JMS connection**: A connection represents a communication link between the application and the messaging server. Depending on the connection type, connections allow users to create sessions for sending and receiving messages from a queue or topic.
- **JMS session**: Represents a single-threaded context for sending and receiving messages. A session is single-threaded so that messages are serialized, meaning that messages are received one-by-one in the order sent.

For our experiments we chose JORAM (Java Open Reliable Asynchronous Messaging). It is open source software released under the LGPL license which incorporates a 100% pure Java implementation of JMS. JORAM adds interesting extra features to the JMS API such as the clustered queue mechanisms. The following section describes the mechanism of queue clustering.

## 3 Clustered Queues

The clustered queue feature provides a load balancing mechanism. A clustered queue is a cluster of queues (a given number of queue destinations knowing each other) that are able to exchange messages depending on their load.

Each queue of a cluster periodically reevaluates its load factor and sends the result to the other queues of the cluster. When a queue hosts more messages than it is authorized

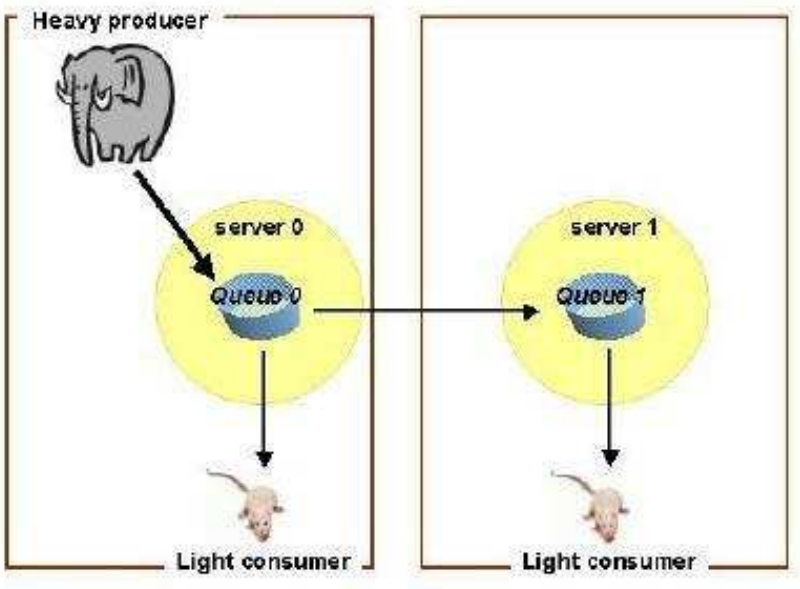


Figure 1: A queue cluster

to do, and according to the load factors of the cluster, it distributes the extra messages to the other queues. When a queue is requested to deliver messages but is empty, it requests messages from the other queues of the cluster. This mechanism guarantees that no queue is hyper-active while some others are lazy, and tends to distribute the work load among the servers involved in the cluster. The figure above shows an example of a cluster made of two queues. An heavy producer accesses its local queue (queue 0) and sends messages. The queue is also accessed by a consumer but requesting few messages. It quickly becomes loaded and decides to forward messages to the other queue (queue 1) of its cluster, which is not under heavy load. Thus, the consumer on queue 1 also gets messages, and messages on queue 0 are consumed in a quicker way.

## 4 Clustered queue load-balancing

We present in this section the key parameters that influence the behavior and the performance of a clustered queue. In the first part, we show the impact of the distribution of clients connections on the performance; in the second part, we provide some details about resource provisioning.

## 4.1 Configuration of clients connections

### 4.1.1 Standard queue

A standard single queue $Q_i$ is connected to $N_i$ message producers that induce a message production rate $p_i$, and to $M_i$ message consumers that induce a message consumption rate $c_i$. The queue length $l_i$ denotes the number of messages waiting to be read in the queue; $l_i$ is always positive and obeys to the law :

$$\Delta l_i = p_i - c_i$$

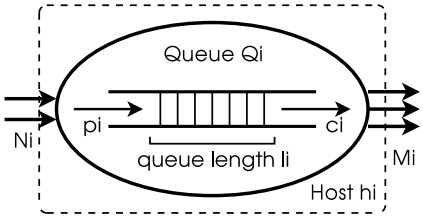


Figure 2: Standard JMS queue $Q_i$

Depending on the ratio between message production and message consumption, three cases are possible:

- $\Delta l_i = 0$: message production and message consumption annihilate themselves and queue length $l_i$ is constant. Queue $Q_i$ is said to be *stable*.
- $\Delta l_i > 0$: there is more message production than message consumption. Queue $Q_i$ will grow and eventually saturate as the queue length $l_i$ gets too big. Queue $Q_i$ is then *unstable* and is said to be *flooded*. Once the queue saturates, the message production rate of producers will be limited. The queue then stabilizes with $\Delta l_i = 0$.
- $\Delta l_i < 0$: there is more message consumption than message production in the queue. Queue length $l_i$ decreases down to 0; the queue is *unstable* and said to be *draining*. Once queue $Q_i$ is empty, message consumers will have to wait and become lazy, $Q_i$ will stabilize with $\Delta l_i = 0$.

The message production and consumption rates are in direct relationships with the number of message producers and consumers:

$$p_i = f(N_i)$$
$$c_i = g(M_i)$$

Thus the stability of a standard single queue is controlled by the ratio between the number of message producers and the number of message consumers.

### 4.1.2 Clustered queue

Clustered queues are standard queues that share a common pool of message producers and consumers, and that can exchange message to balance the load. Each queue runs on a separate server. All the queues of a clustered queue are supposed to be directly connected to each other. This allows message exchanges between the queues of a cluster in order to empty flooded queues and to fill draining queues.

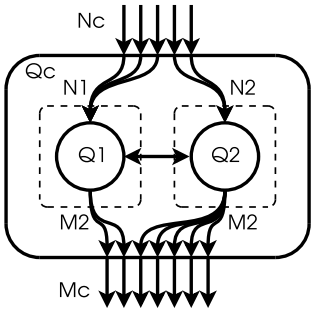


Figure 3: Clustered queue $Q_c$

The clustered queue $Q_c$ is connected to $N_c$ message producers and to $M_c$ message consumers. $Q_c$ is composed of standard queues $Q_i (i \in [1..k])$. Each queue $Q_i$ is in charge of a subset of $N_i$ message producers and of a subset of $M_i$ message consumers:

$$\begin{cases} N_c = \sum_i N_i \\ M_c = \sum_i M_i \end{cases}$$

The distribution of the clients between the queues $Q_i$ is described as follows: $x_i$ (resp. $y_i$) is the fraction of message producers (resp. consumers) that are directed to $Q_i$.

$$\begin{cases} N_i = x_i \cdot N_c \\ M_i = y_i \cdot M_c \end{cases}, \begin{cases} \sum_i x_i = 1 \\ \sum_i y_i = 1 \end{cases}$$

The standard queue $Q_i$ to which a consumer or producer is directed to cannot be changed after the client connection to the clustered queue. This way, the only action that may affect the client distribution among the queues is the selection of an adequate queue when the client connection is opened.

The clustered queue $Q_c$ is characterized by its aggregate message production rate $p_c$ and its aggregate message consumption rate $c_c$. The clustered queue $Q_c$ also has a virtual clustered queue length $l_c$ that aggregates the length of all contained standard queues:

$$l_c = \sum_i l_i = p_c - c_c, \begin{cases} p_c = \sum_i p_i \\ c_c = \sum_i c_i \end{cases}$$

The clustered queue length $l_c$ obeys to the same law as a standard queue:

- $Q_c$ is globally stable when $\Delta l_c = 0$. This configuration ensures that the clustered queue is globally stable. However $Q_c$ may observe local unstabilities if one of its queues is draining or is flooded.

- If $\Delta l_c > 0$, the clustered queue will grow and eventually saturate; then message producers will have to wait.
- If $\Delta l_c < 0$, the clustered queue will shrink until it is empty; then message consumers will also have to wait.

We now suppose that the clustered queue is globally stable, and we list various scenarios that illustrate the impact of client distribution on performance.

**Optimal client distribution** of the clustered queue $Q_c$ is achieved when clients are fairly distributed among the $k$ queues $Q_i$. Assuming that all queues and hosts have equivalent processing capabilities and that all producers (resp. consumers) have equivalent message production (resp. consumption) rates (and that all produced messages are equivalent : message cost is uniformly distributed), this means that:

$$\left\{ \begin{array}{l} x_i = 1/k \\ y_i = 1/k \end{array} \right. , \left\{ \begin{array}{l} N_i = \frac{N_c}{k}, \\ M_i = \frac{M_c}{k} \end{array} \right.$$

In these conditions, all queues $Q_i$ are stable and the queue cluster is balanced. As a consequence, there are no internal queue-to-queue message exchanges, and performance is optimal. Queue clustering then provides a quasi-linear speedup.

**The worst clients distribution** appears when one queue only has message producers or only has message consumers. In the example depicted on Figure 3, this is realized when:

$$\left\{ \begin{array}{l} x_1 = 1 \\ y_1 = 0 \end{array} \right. , \left\{ \begin{array}{l} x_2 = 0 \\ y_2 = 1 \end{array} \right. , \left\{ \begin{array}{l} N_1 = N_c \\ M_1 = 0 \end{array} \right. , \left\{ \begin{array}{l} N_2 = 0 \\ M_2 = M_c \end{array} \right.$$

Indeed, this configuration implies that the whole message production is directed to queue $Q_1$. $Q_1$ then forwards all messages to $Q_2$ that in turn delivers messages to the message consumers.

**Local instability** is observed when some queues $Q_i$ of $Q_c$ are unbalanced. This is characterized by a mismatch between the fraction of producers and the fraction of consumers directed to $Q_i$:

$$x_i \neq y_i$$

In the example showed in Figure 3, $Q_c$ is composed of two standard queues $Q_1$ and $Q_2$. A scenario of local instability can be envisioned with the following clients distribution:

$$\left\{ \begin{array}{l} x_1 = 2/3 \\ y_1 = 1/3 \end{array} \right. , \left\{ \begin{array}{l} x_2 = 1/3 \\ y_2 = 2/3 \end{array} \right.$$

This distribution implies that $Q_1$ is flooding and will have to enqueue messages, while $Q_2$ is draining and will see its consumer clients wait. However the queue cluster $Q_c$ ensures the global stability of the system thanks to internal message exchanges from $Q_1$ to $Q_2$.

**A stable and unfair distribution** can be observed when the clustered queue is globally and locally stable, but the load is unfairly balanced within the queues. This happens when the client distribution is non-uniform.

In the example presented in Figure 3, this can be realized by directing more clients to $Q_1$ than $Q_2$:

$$\begin{cases} x_1 = 2/3 \\ y_1 = 2/3 \end{cases}, \begin{cases} x_2 = 1/3 \\ y_2 = 1/3 \end{cases}$$

In this scenario, queue $Q_1$ processes two third of the load, while queue $Q_2$ only processes one third. Suc situation can lead to bad performance since $Q_1$ may saturates while $Q_2$ is lazy.

It is worthwhile to indicate that these scenarios may all happen since clients join and leave the system in an uncontrolled way. Indeed, the global stability of a (clustered) queue is under responsability of the application developper. For instance, the queue can be flooded for a period; we then assume that it will get inverted and draining after, thus providing global stability over time.

### 4.2 Provisioning

The previous scenario of stable and non-optimal distribution raises the question of the capacity of a queue.

The capacity $C_i$ of standard queue $Q_i$ is expressed as an optimal number of clients. The queue load $L_i$ is then expressed as the ratio between its current number of clients and its capacity:

$$L_i = \frac{N_i + M_i}{C_i}$$

- $L_i < 1$: queue $Q_i$ is underloaded and thus lazy; the message throughput delivered by the queue can be improved and ressources are wasted.
- $L_i > 1$: queue $Q_i$ is overloaded and may saturate; this induces a decreased message throughput and eventually leads to thrashing.
- $L_i = 1$: queue $Q_i$ is fairly loaded and delivers its optimal message throughput.

These parameters and indicators are transposed to queue clusters. The clustered queue $Q_c$ is characterized by its aggregated capacity $C_c$ and its global load $L_c$:

$$C_c = \sum_i C_i \ , \ L_c = \frac{N_c + M_c}{C_c} = \frac{\sum_i L_i \cdot C_i}{\sum_i C_i}$$

The load of a clustered queue obeys to the same law as the load of a standard queue.

However a clustered queue allows us to control $k$, the number of inside standard queues, and thus to control its aggregated capacity $C_c = \sum_{i=1}^{k} C_i$. This control is indeed operated with a re-evaluation of the clustered queue provisioning.

- When $L_c < 1$, the clustered queue is underloaded: if the clients distribution is optimal, then all the standard queues inside the cluster will be underloaded; however, as the client distribution may be non-optimal, some of the single queues may be

overloaded, even if the cluster is globally lazy. If the load is too low, then some queues may be removed from the cluster.

- When $L_c > 1$, the clustered queue is overloaded: even if the distribution of clients over the queues is optimal, there will exist at least one standard queue that will be overloaded. One way to handle this case is to re-provision the clustered queue by inserting one or more queues into the cluster.

# 5 A self-optimizing clustered queue

In this section, we present the design of an autonomic ability which targets the optimization of a clustered queue. The optimization takes place in two steps : (i) the optimal load-balancing of a clustered queue, and (ii) the dynamic provisioning of queues in a clustered queue.

The first part allows the overall improvement of the clustered queue performance while the second part optimizes the queue resource usage inside the clustered queue. Thus the idea is then to create an autonomic system that :

- fairly distribute client connections to the pool of server hosts in the clustered queue,
- dynamically adds and removes queues in a clustered queue depending on the load. That would allow us to use the adequate number of queues at any time.

The implementation of these optimizations relies on the model of clustered queue performance which has been presented in the previous sections.

## 5.1 Control rules

The global clients distribution $D$ of the clustered queue $Q_c$ is captured by the fractions of message producers $x_i$ and consumers $y_i$. The optimal clients distribution $D_{opt}$ is realized when all queues are stable ($\forall i \; x_i = y_i$) and when the load is fairly balanced over all queues ($\forall i, j \; x_i = x_j, \; y_i = y_j$). This implies that the optimal distribution is reached when $x_i = y_i = 1/k$.

$$D = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_k & y_k \end{bmatrix} \; , \; D_{opt} = \begin{bmatrix} 1/k & 1/k \\ \vdots & \vdots \\ 1/k & 1/k \end{bmatrix}$$

**Local instabilities** are characterized by a mismatch between the fraction of message producers $x_i$ and consumers $y_i$ on a standard queue. The purpose of this rule is the stability of all standard queues so as to minimize internal queue-to-queue message transfert.

($R_1$) $x_i > y_i$: $Q_i$ is flooding with more message production than consumption and should then seek more consumers and/or fewer producers.

($R_2$) $x_i < y_i$: $Q_i$ is draining with more message consumption than production and should then seek more producers and/or fewer consumers.

**Load balancing rules** control the load applied to a single standard queue. The goal is then to enforce a fair load balancing over all queues.

($R_3$) $L_i > 1$: $Q_i$ is overloaded and should avoid accepting new clients as it may degrade its performance.

($R_4$) $L_i < 1$: $Q_i$ is underloaded and should request more clients so as to optimize resource usage.

**Global provisioning rules** control the load applied to the whole clustered queue. These rules target the optimal size of the clustered queue while the load applied to the system evolves.

($R_5$) $L_c > 1$: the queue cluster is overloaded and requires an increased capacity to handle all its clients in an optimal way.

($R_6$) $L_c < 1$: the queue cluster is underloaded and could accept a decrease in capacity.

## 5.2 Algorithm

This section presents an algorithm for the self-optimization of queue clustering systems. As a first step we do not allow the modification of the underlying middleware. This constraint restricts the control mechanisms that we can use to implement the autonomic behaviour.

### 5.2.1 System events and controls

Without modification, the underlying JMS middleware does not provide facilities such as session migration that would allow us to migrate clients from one queue to another. However clustered queue systems allow the control of the queue that will handle a new message producer (resp. consumer). This control translated in the model terms means that some $x_i$ (resp. $y_i$) will be increased, and we have the choice for $i$.

On the contrary, a message producer (resp. consumer) that leaves the system induces an unavoidable and uncontrolled decrease in some $x_i$ (resp. $y_i$).

Thus a clustered queue system generates 4 types of events that we can use to control and optimize the system:

| | |
|---|---|
| join(Producer) | join(Consumer) |
| leave(Producer, $Q_i$) | leave(Consumer, $Q_i$) |

The control rules must then be implemented as handlers to these events. The algorithms that control the distribution of clients and the queue cluster provisioning are depicted in Algorithms 1 and 2.

The ElectQueue(ClientType) function chooses the queue that is most far away from the targeted client distribution. The elected queue $Q_i$ then maximizes the gap to the optimal. When considering a new client that is a message producer (resp. consumer), the gap is evaluated with $1/k - x_i$ (resp. with $1/k - y_i$). Thus $Q_i$ satisfies:

$$\begin{cases} x_i = \min_j x_j \text{ (when ClientType = Producer)} \\ y_i = \min_j y_j \text{ (when ClientType = Consumer)} \end{cases}$$

**Algorithm 1** Client joining algorithm

**on** join(ClientType ∈ {Producer, Consumer}, $Q_c$)
**if** ($L_c \geq 1$) **then**
  *// Queue cluster will be overloaded*
  *// An additional queue is required*
  $Q_{k+1} \leftarrow$ NewQueue()
  AddQueue($Q_c$, $Q_{k+1}$)
**end if**
$Q_i$ = ElectQueue($Q_c$, ClientType)
**return** CreateSession(ClientType,$Q_i$)

**Algorithm 2** Client leaving algorithm

**on** leave(ClientType ∈ {Producer, Consumer}, $Q_i \in Q_c$)
**if** (IsMarked($Q_i$, "to be removed") and IsEmpty($Q_i$) **then**
  RemoveQueue($Q_c$, $Q_i$)
  DestroyQueue($Q_i$)
**end if**
**if** ($L_c < 1$) **then**
  $Q_i$ = ElectRemovableQueue($Q_c$)
  **if** $Q_i \neq$ *null* **then**
    Mark($Q_i$, "to be removed")
  **end if**
**end if**

The ElectRemovableQueue($Q_c$) chooses one queue that can be removed from the queue cluster. A queue cannot be removed on demand since it may still have clients connected to it: a queue can only be removed when its last client decides to leave. Thus the removal of a queue $Q_i$ will need two steps: (1) $Q_i$ is marked "to be removed" and no more clients will be addressed to it; (2) when $Q_i$'s last client leaves, $Q_i$ can then be removed from the cluster. Moreover, even if $Q_c$ is underloaded, queue $Q_i$ should not be removed if its removal let $Q_c$ be overloaded. Thus the condition to allow $Q_i$'s removal is:

$$C_i \leq C_c - (N_c + M_c)$$

The following section gives implementation details about these algorithms.

# 6 Implementation Details

The following section briefly explains the implementation principles of our self-optimization algorithm on top of JORAM. Details on how JORAM has been wrapped within the SELF-WARE platform are given at the end of this document, in the Annex part.

## 6.1 Requirements

To implement a self-managed queue cluster using the autonomic computing design principles require the following management capabilities:

- to know the current number of message producers and consumers,
- to know where the servers are deployed, 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 queue cluster,
- to allocate a new server to create a new queue,
- to add and remove a queue in a server.

## 6.2 The control loop

To simplify, we will consider that clients create only one session by connection. By doing this we assimilate the creation of sessions and the creation of connections. Assuming this, the first prototype is achieved by wrapping the standard JMS ConnectionFactory by a "LBConnectionFactory" (where LB stands for Load Balancing).

### 6.2.1 LBConnectionFactory

As the client gets the connection factory through JNDI, it gets the LBConnectionFactory instead. This is the main non-functionnal hook in the system that allows to control the distribution of producers and consumers among servers. This component offers the following methods:

**createConnection(...)** takes the type of the client as a parameter (Producer or Consumer). To create the connection with the right server, it requests a component called "ClusterManager" which provisions ("resizes") 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(...)** effectively closes the connection to the server and notifies the ClusterManager so it can decrease the number of queues in the cluster if necessary.

### 6.2.2 ClusterManager

This component stores the state of the global system, i.e. the number of servers currently used, the number of clients connected to each server, their type. The state changes as client requests are received from the LBConnectionFactory. The different requests are:

- a consumer wants a connection;
- a producer wants a connection;
- a consumer wants to close a connection on server $Q_i$;
- a producer wants to close a connection on server $Q_i$.

In the first two cases, the ClusterManager elects a server taking into account the capacities in terms of clients. If the cluster is evaluated to be full of producers or consumers, the LBClusterManager uses the procedures **NewQueue()** and **AddQueue()** to launch a JORAM server on a free host and to create a queue linked to the cluster on that server. Of course, the cluster manager will update its internal image of the global system according to this.

# 7 Evaluation

A series of experiments was run to assess the performance of JORAM. Rather than finding an absolute maximum, these experiments were aimed at finding the relevant factors impacting the performance of JORAM queues. The focus was on assessing the usefulness of using queue clusters instead of single queues.

**Environment** The experiments presented below were run on a cluster of Mac Mini computers with the following specifications:

- *Mac OS X 10.4.7, Intel Core Duo 1.66 GHz, 2 GB SDRAM DDR2 (667 MHz frontal bus)*
- *Java J2SDK1.4.2_13, JORAM 4.3.21*
- *Ethernet Gigabit network*

In each experiment, the measurements were taken with JMX probes located on a computer outside the cluster. Each JORAM queue ran a JMX server which was accessed by one of the JMX probes. The monitored attributes on the queue were *NbMsgsDeliverSinceCreation* which is the number of messages read by consumers on the queue since its creation and *MessageCounter* which is the number of messages presently waiting in the queue. The JMX probes were reading these attributes every second.

In the following experiments, each JORAM queue was located on a distinct node. The queues were running in a persistent configuration. The producers and consumers were transactional with a commit between each message. The Java Virtual Machine hosting each queue was able to use 1536 MBytes of memory. The Garbage Collector was disabled to prevent random hits on performance. The size of the JMS messages used was 1 KBytes. The network was not considered to be meaningful factor in these experiments.

To obtain meaningful results, each experiment was run three times. The charts were constructed using the average of the three tests. The average throughput was calculated excluding the first five and last five seconds as a way to only account for the stable part of the process.

**The number of waiting messages factor** This experiment aims at showing the impact of the number of messages waiting in the queue on the performance. In a first step, producers write 1500 messages in a single queue, while in a second step, consumers read these messages from the queue until it is empty. Figure 4 shows this experiment. We observe that the number of messages waiting in the queue has a strong direct impact on the performance: the message processing rate of the queue decreases as the queue length grows.

Moreover we observe that the performance of the queue is noticeably higher for message production than for message consumption. Indeed, the next experiments figure out the optimal ratio between message producers and message consumers to assign to a single queue in order to ensure its stability. In these experiments, a single message producer injects 15000 messages into the queue, and one or more message consumers read the messages. Figure 5 presents the results when the queue is assigned a single message producer and a single message consumer. In this configuration, the queue is strongly unstable with about two times more message production than consumption. This leads to a growing queue length, hence reduced performance. Figure 6 presents the results when the queue is loaded with one message producer and two message consumers. In this scenario, the queue is stable with equivalent message production and consumption rates. The queue length remains low, and thus the performance are stable. An experiment with one message producer and third message consumers shows a very similar queue behaviour. From these experiments, we deduce that the optimal clients ratio is one message producer for two message consumers.

**Single queue limit** In order to assess the interest of having a cluster queue instead of a single queue, we need to measure the highest throughput a single queue can reach with the previously described parameters. We made multiple measurements with a varying number of producers and consumers accessing a single queue. As explained before, for a given number of producers, the ratio to obtain the best throughput was always one producer for two consumers. These measurements are summed up on Figure 7. These results account

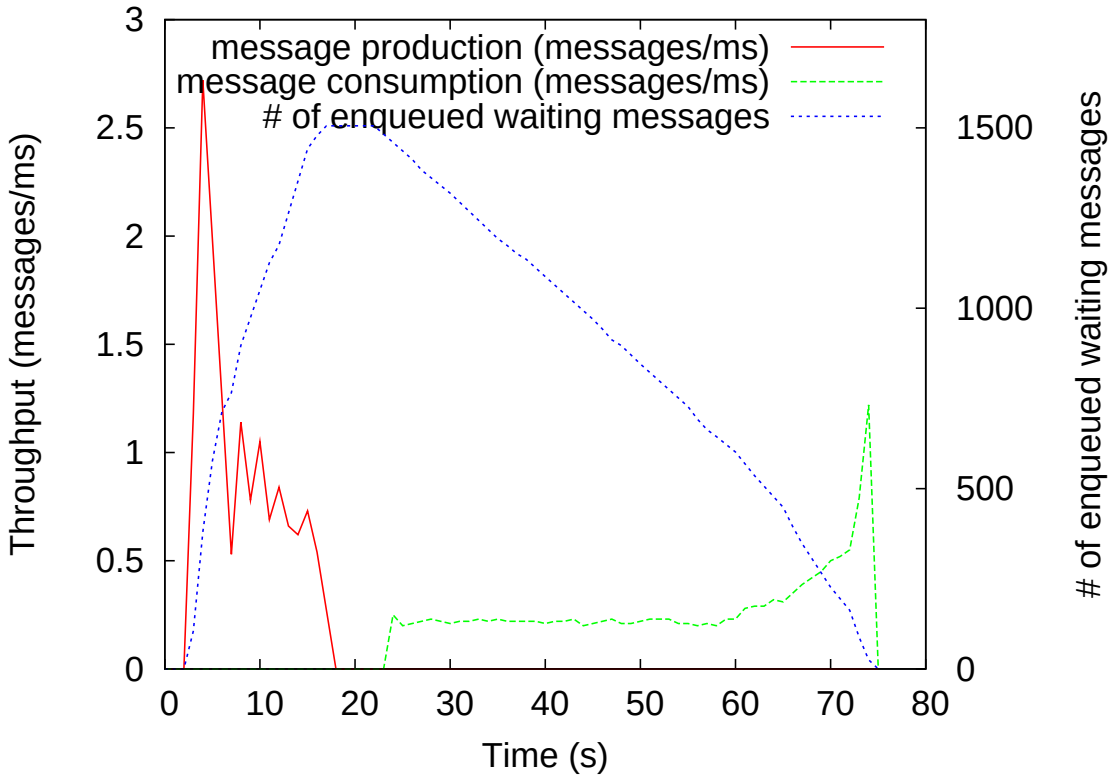


Figure 4: Impact of the Waiting Messages on the Performance

for the strong interest in dynamic provisioning and optimization of the load-balancing of clustered queues in order to always provide the best clustered queue size and clients distribution for best performance.

### 7.1 Load-balancing optimization

The following presents an evaluation of the queue cluster load-balancing optimization that fairly distributes client connections among the queues. For this evaluation, we expose a queue cluster composed of two queues to 4 messages producers and 8 message consumers. A single message producer emits 10000 messages, while a message consumer reads 5000 messages. This configuration ensures that the queue cluster is stable. Figure 8 presents the results of this experiment when the queue cluster is driven with the standard JORAM load-balancing strategy, while figure 9 presents these results when the cluster is driven by our optimized load-balancer. When using the original load-balancing strategy, we observe a noticeable unstability with a higher message production rate than the message consumption rate (see Figure 8). This behaviour is the consequence of a bad distribution of the clients over the internal queues of the cluster, which generates local instabilities that are hardly compensated by the internal queue-to-queue message exchange mechanism. This directly threatens the queue cluster performance which is then suboptimal, with less than 0.3 messages/ms. In comparison, when using our dynamic load-balancing optimization,

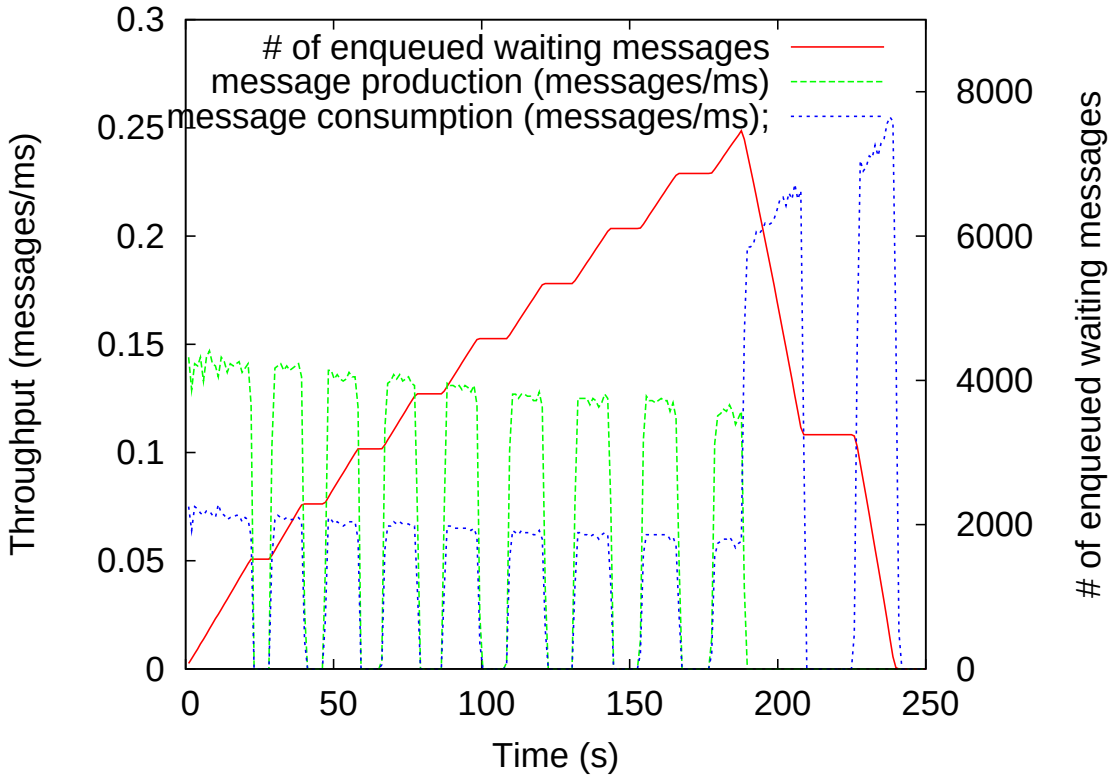


Figure 5: Behaviour of a single queue with one message producer and one message consumer

the queue cluster presents a very stable and balanced behaviour. Indeed, the message production rate and the message consumption rate both reach 0.35 messages/ms.

### 7.2 Dynamic provisioning

We now consider the evaluation of the dynamic provisioning algorithm which dynamically adapts the number of queues inside a queue cluster depending on the load. The workload applied to the queue cluster consists in 5 message producers and 10 message consumers. As in the previous experiment, a message producer generates 10000 messages while a message consumer gets 5000 messages. To generate an increasing workload, the clients are created gradually, one at a time, and new client creations are separated with a delay of 10s. The queue cluster is kept stable by creating clients so as to respect a ratio of two message consumers for one message producer. The queue cluster initially contains one single standard queue.

Figure 10 shows the behaviour of the queue cluster under a static provisioning policy, while figure 11 presents its behaviour under dynamic provisioning. When statically provisioning, the queue cluster contains one single queue during the entire experimentation, no matter how many clients are connected to it. The queue cluster stabilizes quickly after the second step around time 50s, with message production and consumption rates of about 1.9 messages/ms until the end of the experiment. When the queue cluster is dynamically

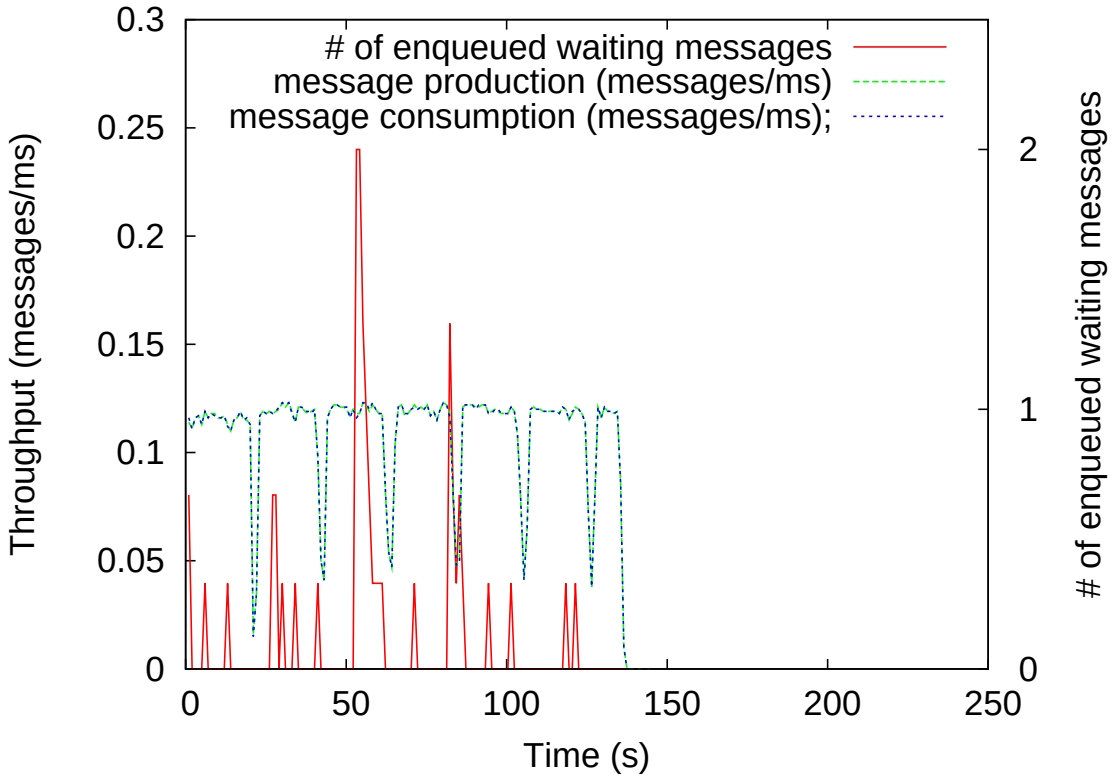


Figure 6: Behaviour of a single queue with one message producer and two message consumers

provisioned, the queue cluster behaves as in the previous experiment as long as the capacity of the single queue is sufficient to absord the workload. Then, arount time 120s, as the workload exceeds the capacity of a single queue, the cluster is provisioned with a second queue, to which new clients are directed. As expected, the performance of the queue cluster doubles, jumping from 1.9 messages/ms to 3.7 messages/ms.

### 7.3 Conclusion for the measurements

These measurements show some interesting points. In a single queue, the critical factor impacting the performance is the number of messages waiting in the queue. Increasing the number of producers and consumers on a single queue leads to an increase in performance which is not linear. Furthermore a ceiling throughput is reached when the number of clients corresponds to the capacity of the queue.

In a cluster queue, the balance of the cluster and the stability of the internal queues are extremely important. Even a slight instability between the queues strongly decreases the overall throughput. The instability seems to lead to an increase in the number of messages waiting in the queues. In contrast of a single queue, adding queues in a stable and well-balanced cluster leads to a linear increase in performance.

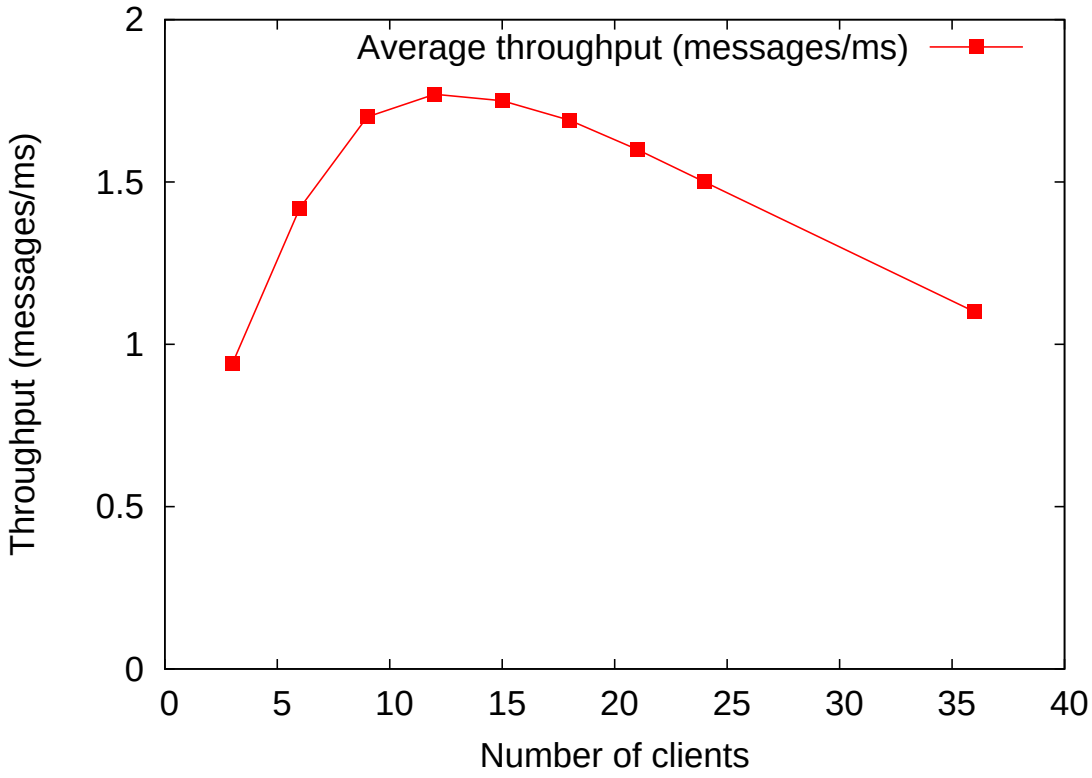


Figure 7: Capacity of a stantard single queue

## 8 Conclusion

Providing a scalable and efficient Message Oriented Middleware is an important topic for today's computing environments. This documentx analyses the performance of a Message Oriented Middleware and proposes, in the context of the Selfware platform, a self-optimization algorithm to improve the efficiency of the MOM infrastructure.

This optimization takes place in two parts: (i) the optimization of the clustered queue load-balancing and (ii) the dynamic provisioning of a queue in the clustered queue. The first part allows the overall improvement of the clustered queue performance while the second part optimizes the resource usage inside the clustered queue.

We describe (i) the key parameters impacting the performance of the MOM and (ii) the rules that control these parameters for optimal performances. This paper also presents an evaluation that shows the impact of these parameters on the performances and the behavior of dynamically provisioned clustered queue.

Currently, the control loop has a very basic actuator to drive a client connection to a specific queue. The advantage of this actuator is its simplicity. However, the control loops cannot reconfigure the client connection during a session. Part of our future work is about

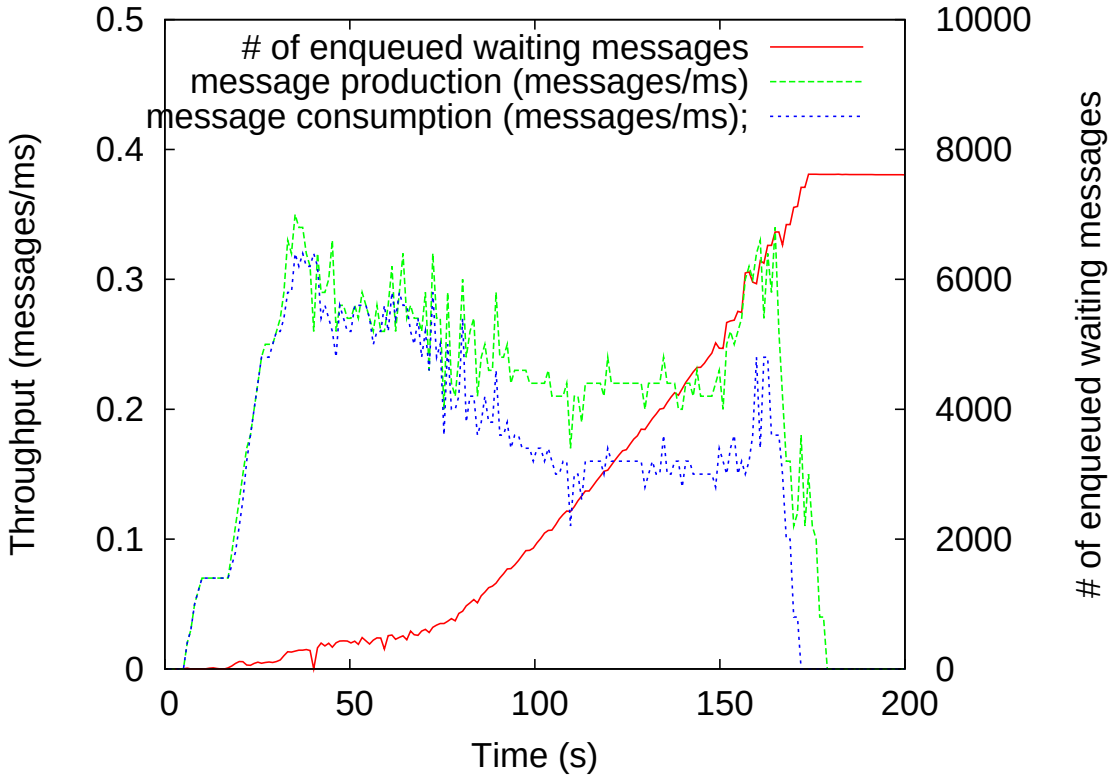


Figure 8: Standard Joram queue cluster load-balancing strategy

providing a more powerful actuator. This actuator will provide the control loop with the ability to migrate a client connection when necessary. This will require a mechanism to move session data on other queue.

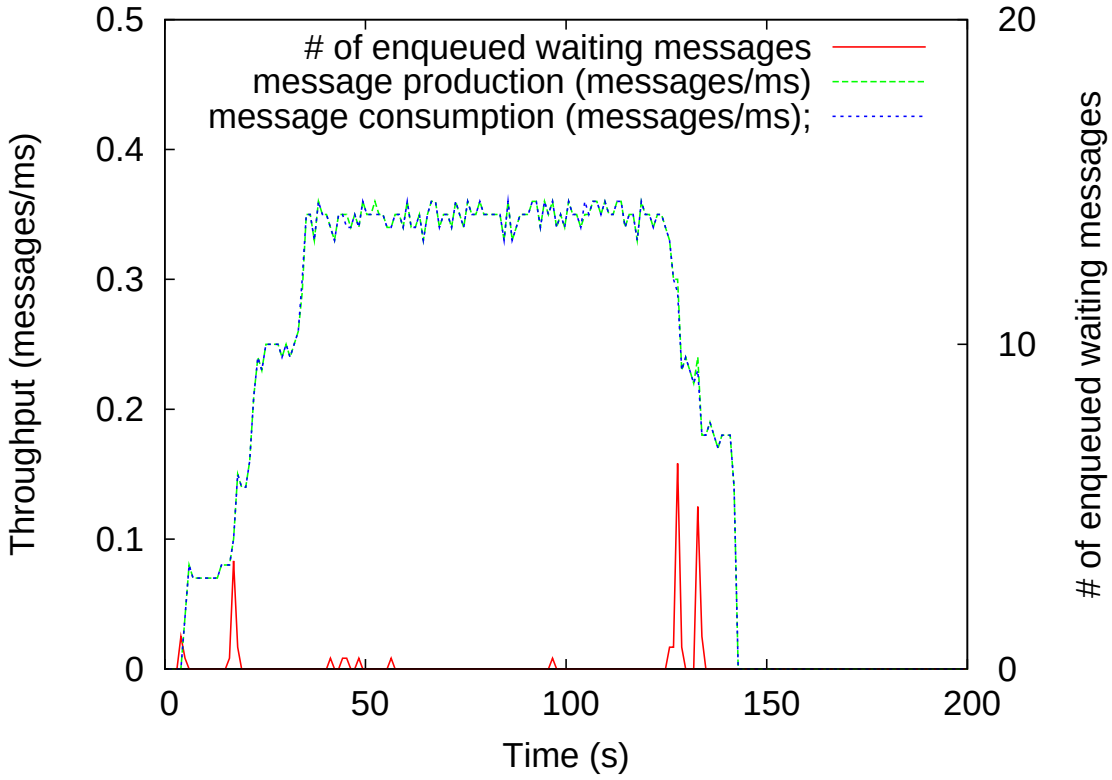


Figure 9: Optimized queue cluster load-balancing

# 9 Annex

## 9.1 Fractal Architecture

To modelize as best as possible the behaviour of the Joram architecture and to stay as close as possible to reality, we have chosen the Fractal architecture of figure 12.

The highest level component is the domain. It is constituted of a set of servers, themselves constituted of users, topics and queues components. These components are wrappers of Joram elements.

Components representing cluster queue and cluster topic are at the same level as the domain. They contain all the queues and topics that are part of the cluster. It is useful to notice that these components are not new components (re-created by cluster component) but that they are shared. Indeed, they are already sub-components of servers.

Moreover, at the same level as the domain is a component wrapping a JNDI registry. This component wrapps the **Joram JNDI** registry used by all servers to bind destinations and cluster queues.

Finally, to represent the graph of a hierarchical topic, we use bindings between the several components part of the hierarchy. A topic component also has a client interface

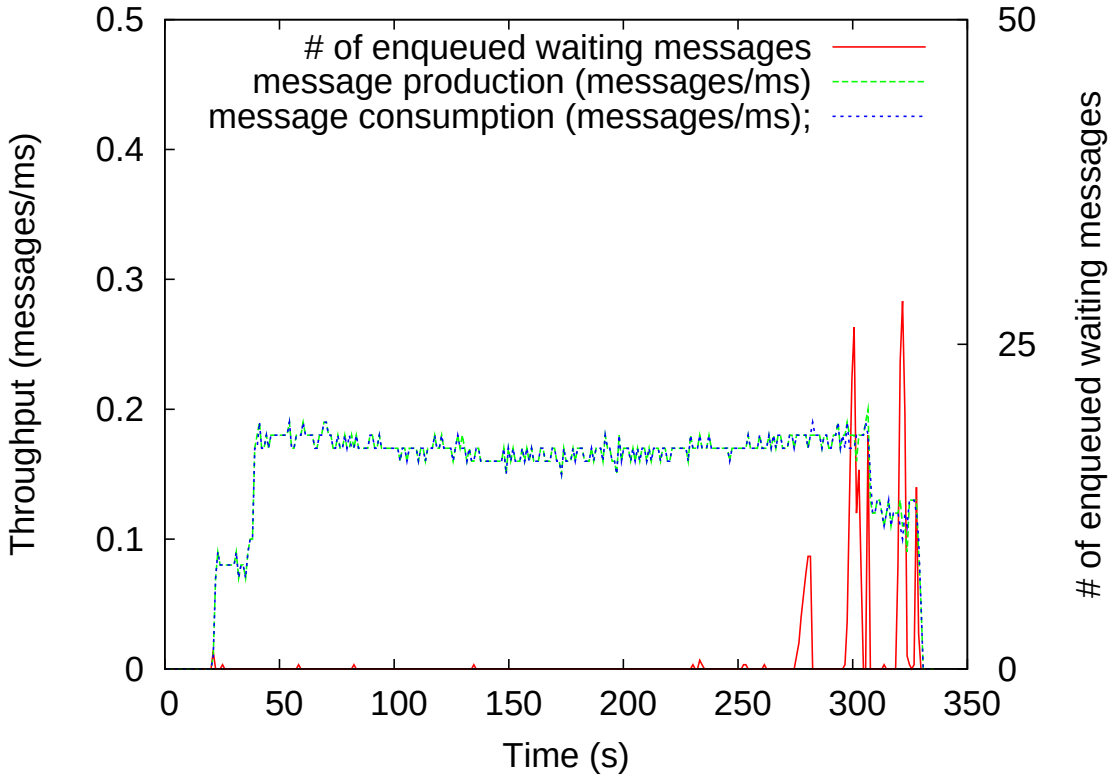


Figure 10: Static provisioning of a clustered queue

that can be bound to an other topic component which will be its father in the hierarchy.

## 9.2 Instrumentation

To deploy Joram applications on Jade, some wrappers and controllers that specialize composite components have been developped.

### 9.2.1 Domains

A Joram domain is represented by a simple Fractal composite component.

### 9.2.2 JNDI Registry

All servers and cluster queues components are bound to this JNDI component because they need a direct access to the registry to bind destinations and connection factories. The JNDI component is directly bound in the Fractal RMI registry with the name *JndiRegistry*. We also suppose that there is only one JNDI registry per Joram architecture deployed with Jade.

The JNDI interface listed below presents methods that provide the JNDI server host and port.

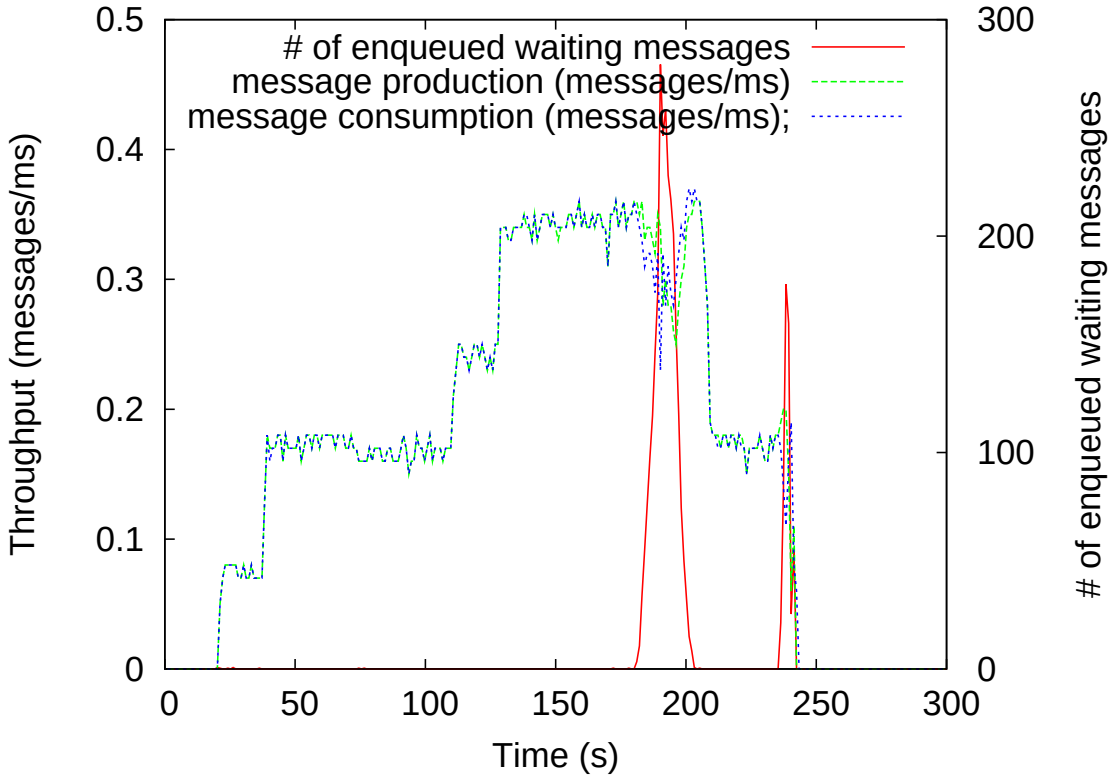


Figure 11: Dynamically provisioned clustered queue

```
public interface JndiInterface {
        public String getHost();
        public String getPort();
}
```

### 9.2.3 Servers

A Joram server is represented by a Fractal composite component. Its content controller has been overridden to add some specific treatments when we add or remove a queue, a topic or a user. For example, if we choose to add a queue, the *addFcSubComponent* method starts the queue component and then calls the setProperties method of its queue interface (this is in the setProperties method that the queue is bound in the JNDI registry).

This composite has a client interface named *jndi* that is bound to the component wrapping the JNDI registry. So, the composite keeps a reference to the JNDI and if it has to bind an object, can do it easily.

That's why, if we restart the JNDI server on an other host during execution (in a case of repair), we just have to update the binding to make the server able to bind objects. With this solution, the user doesn't need to give the host and port of JNDI server as parameters when he starts a server. The FScript command to create a server introspects the Fractal

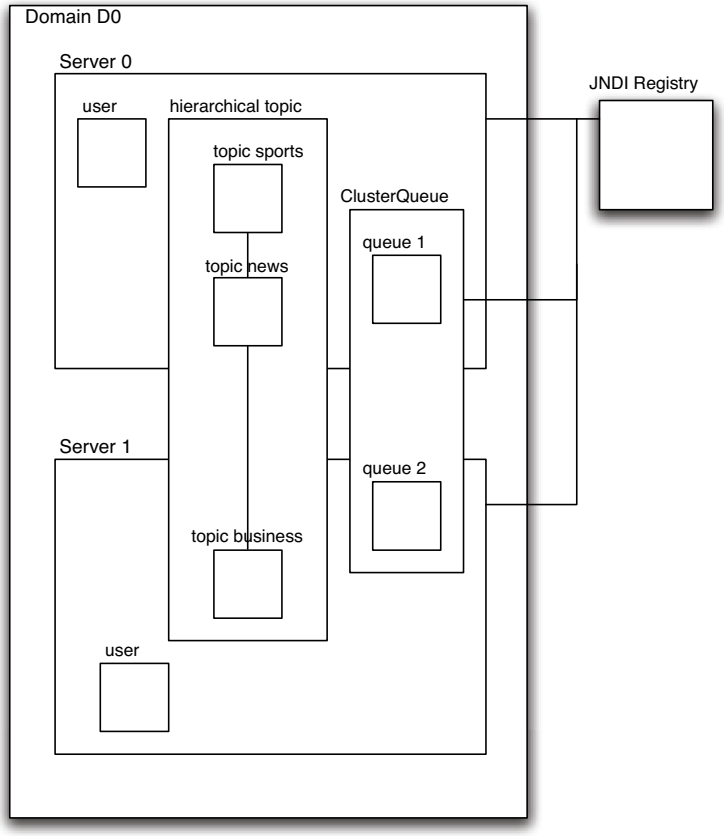


Figure 12: Fractal architecture

RMI registry to find the object named *JndiRegistry* and binds the server composite to it.

Moreover, this composite life cycle has been overridden to link the Fractal and legacy life cycles. So, starting the component at Fractal level leads to starting the Joram server.

In addition, starting a Joram server requires specific actions. It must be, in the classpath an *a3servers.xml* file describing the static architecture of the servers we want to deploy. This file and its particular structure is mandatory to start a server. It's only when this file is well written that a server can know its configuration and can start. As we can't know, at the begining, which architecture will be deployed, we have to update dynamically this file during execution.

If we want to deploy more than one server, it's mandatory to create a domain. It is done by using a connection to the *AdminModule* of the server previously created, and by calling the *addDomain* method. This call updates the Joram architecture configuration with the domain description. Then, the specific life cycle controller updates the *a3servers.xml* file by overwritting it with the *AdminModule* configuration updated with the new domain.

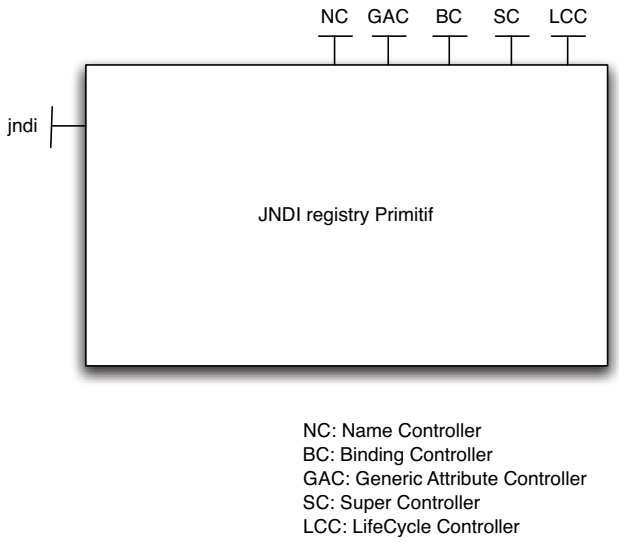


Figure 13: Fractal primitive JNDI registry component

It's nearly the same if we want to add a new server to the domain. We have to connect to the platform *AdminModule* and call the *addServer* method. This call doesn't start the new server, but updates the *AdminModule* configuration with the new server configuration. The new server is thus able to know its own and other servers configuration. Then, the specific life cycle controller updates the *a3servers.xml* file and the new server can start (at Joram level).

But, two cases must be distinguished : if the server is the first of the platform or if it is just a new server in the domain. According to that, the server start is different.

If we create the first server, we have to:

- Create the server Fractal component.
- Create a well-known *a3servers.xml* file with the first server description updated with its own parameters. This file doesn't declare any domain.
- Start the Joram server.
- If the server is part of a domain, update the AdminModule configuration to create a domain.
- Update the a3servers.xml file with the *AdminModule* configuration.

If we just add a server to an existing domain containing other servers, we have to:

- Create the server Fractal component.

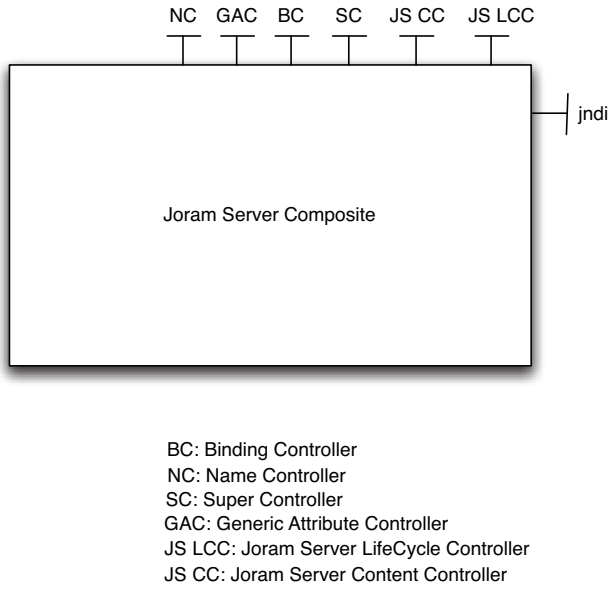


Figure 14: Fractal composite Joram server

- Find a existing server part of the domain that the new server want to join. Connect to its *AdminModule* and call the *addServer* method.
- Update the a3servers.xml file with the *AdminModule* configuration.
- Start the Joram server.

In the server life cycle controller, the distinction between the first server and the others is made by instrospection. When a server component is created, the Fractal architecture is introspected to find if there is already one server component. According to that, the component can specify its start actions.

So, if the server to create is the first server, the life cycle will follow these steps:

- Introspect the Fractal architecture to find other servers.
- Create a weel-known a3servers.xml file with the first server description updated with its own parameters. This file doesn't declare any domain.
- Start the Joram server.
- If a domain name attribute exists, connect to the *AdminModule* and call the *AddDomain* method.
- Update the a3servers.xml file using the *AdminModule*.

and if the server to create is only one server to add to an already existing domain that already has servers, the life cycle will follow these steps:

- Introspect the Fractal architecture to find other servers.
- Connect to the *AdminModule* and call the *addServer* method to update the platform configuration.
- Update the a3servers.xml file using the *AdminModule*.
- Start the Joram server.

Note: According to Joram specification, we have to deploy one unique Joram server per Java Virtual Machine. It is also impossible to start 2 servers in one Java process. That's why we advise to use one Jade node for each server to deploy.

### 9.2.4 Cluster Queue

Queues and Topics Clusters are represented by composite components specialized with a specific content controller. The particularity of these components is that they don't re-create the components they contain (when we add a queue to a cluster for example, the queue component is not re-created) but that they share them with the composites representing the servers. So, cluster components just surround topics or queues they contain.

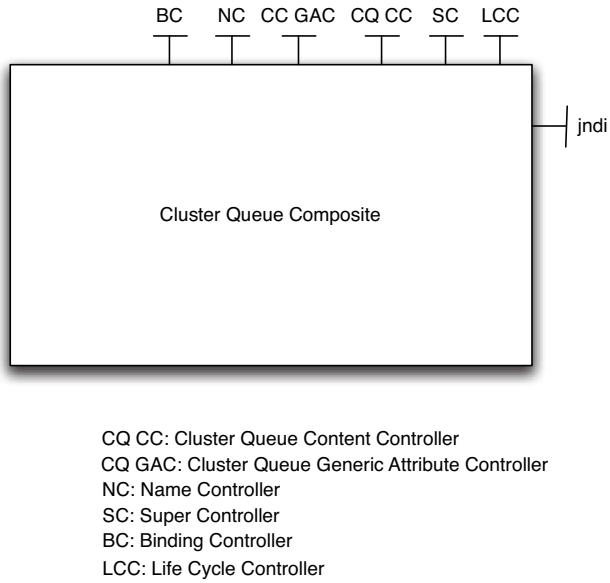


Figure 15: Cluster queue Fractal model

Two FScript scripts allow to manage cluster queues. One to create the cluster queue component and another one to add destinations to the cluster queue. The first script creates a Fractal component representing the cluster and binds it in the Fractal RMI registry. Moreover, it gets the JNDI registry component in the Fractal RMI registry and binds it to the cluster queue component.

In the case of a cluster queue, we have to bind a Joram cluster queue object in the JNDI registry. This particular object is created and binded for the first time, when we add the first queue to the cluster. That's why the cluster queue content controller has been overridden.

Moreover, basic Julia's implementation doesn't authorize to remove a composite sub-component if the composite isn't stopped. As we need to authorize this particular case in our application, we have specialized the content controller by removing this check.

Finally, each time we add (or remove) a queue to (or from) the cluster, the cluster queue object bound in the JNDI registry must be updated. This is done in the *AddFcSub-Component* (or *RemoveFcSubComponent*) method of the overridden content controller.

### 9.2.5 Destinations and Users

Queues, topics and users are modelized as Fractal primitive components. They have a specific functional interface that allows to access their application wrapper.

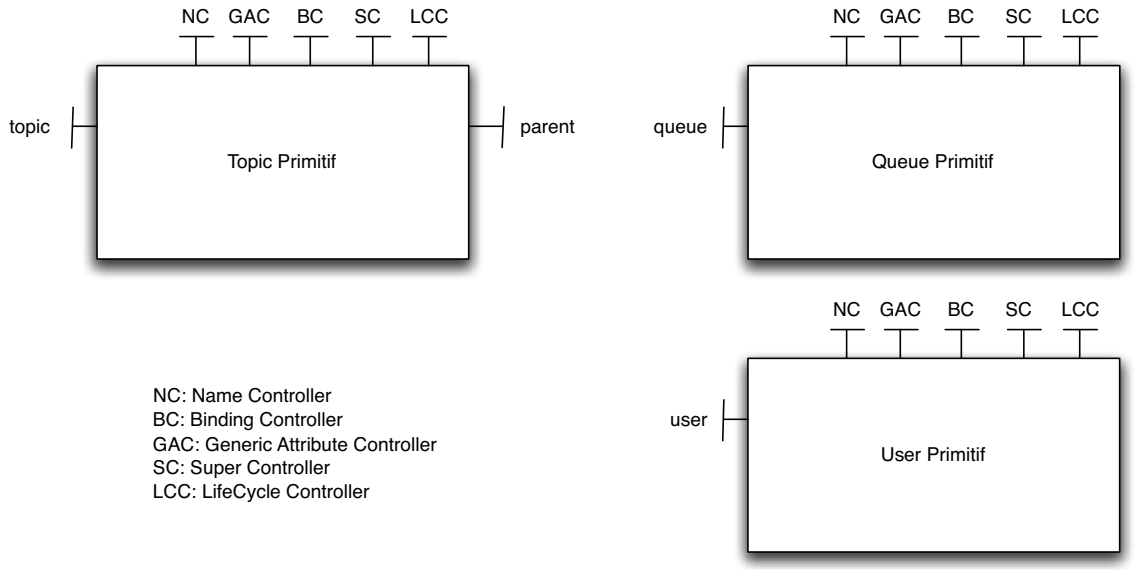


Figure 16: Fractal model for Joram elements

They are contained in the composite representing the Joram server. Moreover, topics and queues can be shared between servers and clusters and so, surrounded by the composite representing a cluster.

```
package org.ow2.jasmine.jade.resources.joram;

import org.objectweb.jasmine.jade.util.JadeException;
import org.objectweb.joram.client.jms.Queue;

public interface QueueInterface {
        public Queue getQueue();
        public void setProperties() throws JadeException;
        public String getMyServerSID();
}
```

```
package org.ow2.jasmine.jade.resources.joram;

import org.objectweb.jasmine.jade.util.JadeException;
import org.objectweb.joram.client.jms.Topic;

public interface TopicInterface {
        public Topic getTopic();
        public void setProperties() throws JadeException;
        public void unsetParent();
        public void setChildInHierarchy(String topicName, TopicInterface ti);
        public void unsetChildInHierarchy(String topicName);
}
```

```
package org.ow2.jasmine.jade.resources.joram;

import org.objectweb.joram.client.jms.admin.User;

public interface UserInterface {
        public User getUser();
}
```

### 9.2.6 Connection Factory and Cluster Connection Factory

In addition to cluster queue, Joram introduces the concept of cluster connection factory. A cluster connection factory is composed of a set of connection factories.

Like a cluster queue, a client who wants to create a connection, just needs to get the cluster connection factory. He is then route on a particular connection factory, thanks to optimization criteria.

Cluster connection factories are composites components and connection factories are primitives one, shared by servers and cluster connection factories components.

## References


[1] Appleby, K., Fakhouri, S.A., Fong, L.L., Goldszmidt, G.S., Kalantar, M.H., Krishnakumar, S., Pazel, D.P., Pershing, J.A., Rochwerger, B.: OcÃ©ano-SLA based management of a computing utility. In: Proceedings of Integrated Network Management. (2001) 855–868

[2] Norris, J., Coleman, K., Fox, A., Candea, G.: OnCall: Defeating spikes with a free-market application cluster. In: 1st International Conference on Autonomic Computing (ICAC'04), New York, NY, USA (May 2004) 198–205

[3] Soundararajan, G., Amza, C.: Autonomic provisioning of backend databases in dynamic content web servers. Technical report, Department of Electrical and Computer Engineering, University of Toronto (2005)

[4] Soundararajan, G., Amza, C., Goel, A.: Database replication policies for dynamic content applications. In: First EuroSys Conference (EuroSys 2006), Leuven, Belgium (April 2006)

[5] Urgaonkar, B., Shenoy, P.: Cataclysm: Handling extreme overloads in internet services. Technical report, Department of Computer Science, University of Massachusetts (November 2004)

[6] Urgaonkar, B., Shenoy, P.J.: Cataclysm: policing extreme overloads in internet applications. In: Proceedings of the 14th international conference on World Wide Web, (WWW'05), Chiba, Japan (May 2005) 740–749

[7] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P.: Dynamic provisioning of multi-tier internet applications. In: Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC'05), Seattle (June 2005)

[8] Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., Tantawi, A.: Analytic modeling of multitier internet applications. ACM Transaction on the Web **1**(1) (2007) 2

[9] Chandra, A., Gong, W., Shenoy, P.: Dynamic resource allocation for shared data centers using online measurements. In: Proceedings of the Eleventh IEEE/ACM International Workshop on Quality of Service (IWQoS 2003), Monterey, CA (June 2003)

[10] Zhang, Q., Cherkasova, L., Smirni, E.: A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In: ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing, Jacksonville, Florida, USA (June 2007) 27

[11] Stewart, C., Shen, K.: Performance modeling and system management for multi-component online services. In: NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation. (2005) 71–84

[12] Urgaonkar, B., Pacifici, G., Shenoy, P.J., Spreitzer, M., Tantawi, A.N.: An analytical model for multi-tier internet services and its applications. In: Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05), Banff, Alberta, Canada (June 2005) 291–302

[13] Henjes, R., Menth, M., , Zepfel, C.: Throughput performance of java messaging services using websphereMQ. In: 5th International Workshop on Distributed Event-Based Systems (DEBS), Lisboa, Portugal (7 2006)

[14] Menth, M., Henjes, R.: Analysis of the message waiting time for the fioranoMQ JMS server. In: 26th International Conference on Distributed Computing Systems (ICDCS), Lisboa, Portugal (7 2006)

[15] Chen, S., Greenfield, P.: Qos evaluation of jms: An empirical approach. In: HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9, Washington, DC, USA, IEEE Computer Society (2004) 90276.2