Improving the Performances of JMS-Based Applications

Abstract:

In the Java world, a standardized interface exists for message-based middleware (MOMs) : Java Messaging Service or JMS. And like other middleware, some JMS implementations use clustering techniques to provide some level of performance and fault-tolerance. In this paper, we analyse the efficiency of various clustering policy in a real-life cluster and the key parameters impacting the performances of MOMs. We show that the resource-efficiency of the clustering methods can be very poor due to local instabilities and/or global load variations. To solve these issues, we describe the rules that control these parameters for optimal performances and we propose a solution based on autonomic computing to (i) dynamically adapt the load distribution among the servers (load-balancing aspect) and (ii) dynamically adapt the replication level (provisioning aspect). We present an evaluation that shows the impact of these rules on the performances and the behavior of dynamically provisioned clustered queue.

Keywords: MOM, JMS, Autonomic management, Self-optimization.

1 Introduction

With the emergence of the internet, multiple applications require to be integrated with each other. One common glue technology for distributed, loosely coupled, heterogeneous software systems is Message-Oriented Middleware (MOM). MOMs are based 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). The use of MOMs in the context of internet has evidenced a need for highly scalable and highly available MOM. This paper analyses the performance of a MOM 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.

We will show that in some cases this mechanism can effectively provide a linear speedup but in other cases this mechanism is completely inefficient. We analyse that the efficiency of this mechanism depends on the distribution of client connections to MOM queues. We describe a solution that will improve the efficiency of this mechanism by optimizing the distribution of client connections in the cluster queue. Furthermore, an important aspect of this clustering policy is the selection of the level of clustering, i.e. the number of queues in the clustered queue. A commonly used solution is to select a fixed number of queues in the clustered queue. However, this static solution has some drawbacks. Let N be the (fixed) number of replicas. If N is too large, resources are wasted; if N is too small, performance may be compromised. In any case, the choice is problematic if the expected load of a queue is difficult to predict. Human administrators can monitor the load of the queuing system using adequate tools. However if a queue is underloaded or overloaded, an administrator cannot react as quickly as required.

This paper targets the optimization of these clustering mechanisms. This optimization will take 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.

This paper 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 presents related work and section 9 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 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

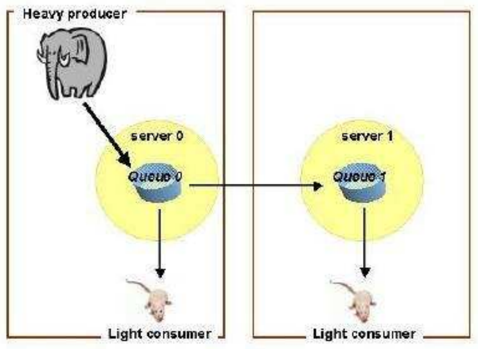

Figure 1: A queue cluster

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 :

$$\Delta l_i = p_i - c_i$$

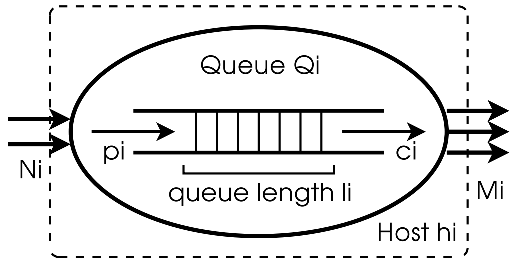

Figure 2: Standard JMS queue Q_i

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:

$$\begin{array}{l} p_i = f(N_i) \\ c_i = g(M_i) \end{array}$$

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.

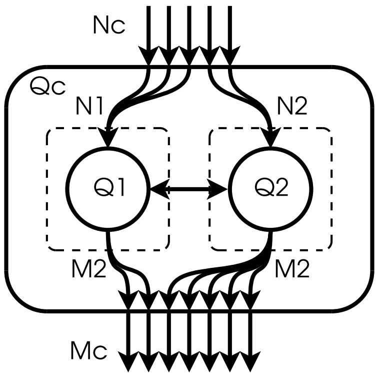

Figure 3: Clustered queue Q_c

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:

$$\begin{cases} x_i = 1/k \\ y_i = 1/k \end{cases}, \begin{cases} N_i = \frac{N_c}{k}, \\ M_i = \frac{M_c}{k} \end{cases}$$

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:

$$\begin{cases} x_1 = 1 \\ y_1 = 0 \end{cases}, \begin{cases} x_2 = 0 \\ y_2 = 1 \end{cases}, \begin{cases} N_1 = N_c \\ M_1 = 0 \end{cases}, \begin{cases} N_2 = 0 \\ M_2 = M_c \end{cases}$$

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:

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

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.

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 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:

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

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

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

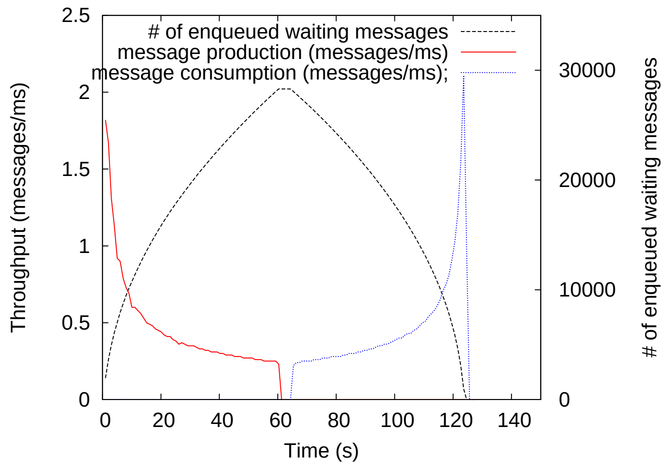

Figure 4: Impact of the Waiting Messages on the Performance

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 pro-

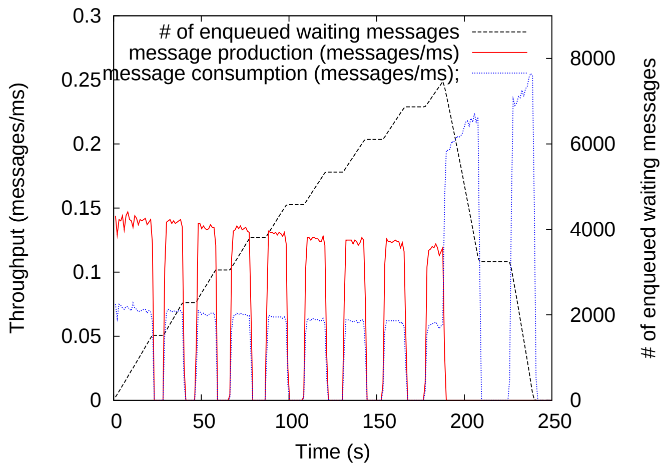

Figure 5: Behaviour of a single queue with one message producer and one message consumer

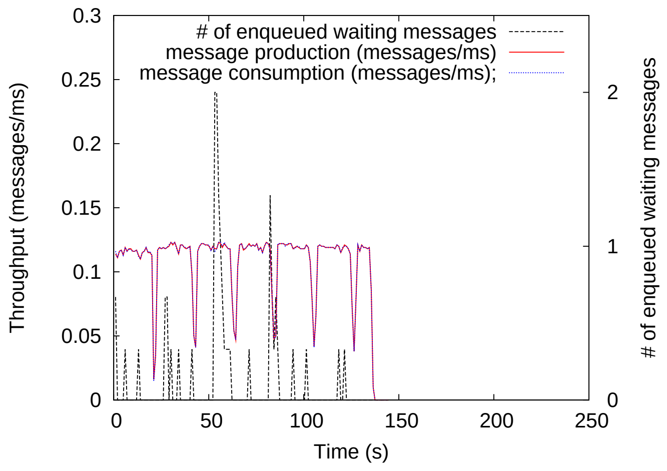

Figure 6: Behaviour of a single queue with one message producer and two message consumers

duction 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 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.

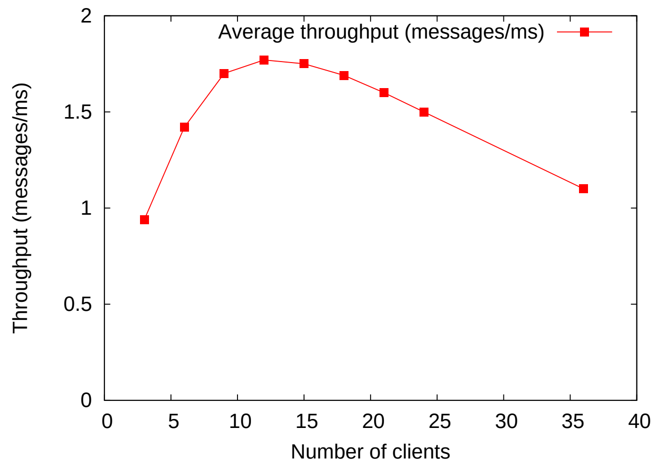

Figure 7: Capacity of a stantard single queue

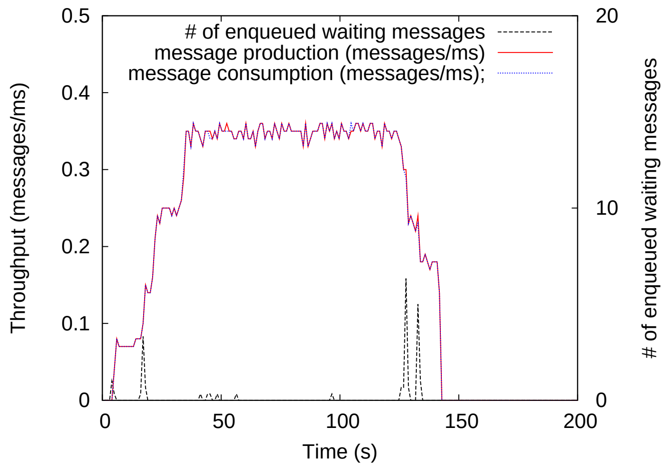

Figure 9: Optimized queue cluster load-balancing

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

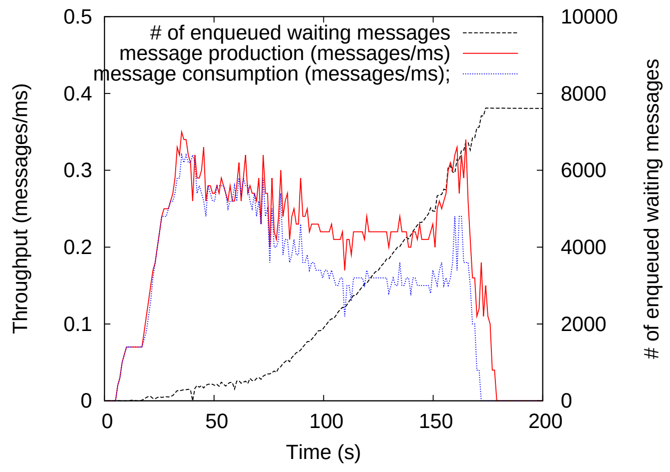

Figure 8: Standard Joram queue cluster load-balancing strategy

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, 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

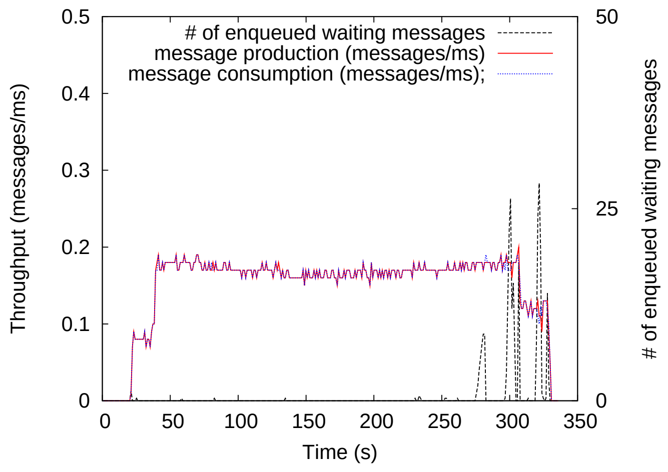

Figure 10: Static provisioning of a clustered queue

shows the behaviour of the queue cluster under a static provisioning policy, while figure 11 presents its behaviour under dy-

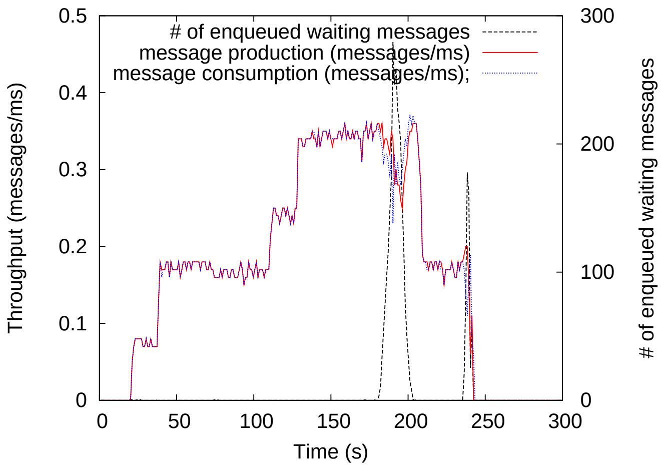

Figure 11: Dynamically provisioned clustered queue

namic 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 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.

8 Related work

The related work for this paper comes from the context of resources management for Internet services. Past work on resource management of internet services falls to different categories.

A first category has focused on studying on resource management of Internet services has considered the management of a dynamically extensible set of resources, where the infrastructure can dynamically grow or shrink [1, 2, 3, 4, 5, 6].

Oceano provides an adaptive hosting environment with a dynamic partitioning of the resources among the running applications [1]. This dynamism allows the system to react to load peaks by increasing the partition size of the concerned application and to shift unused resources from under-loaded applications to the others. The main issue in this work seems to be the node allocation delay. That explains why the platform assumes that some application parts cannot be dynamically and are thus statically allocated and configured (e.g. the database tier). OnCall is similar to Oceano but specifically targets fast handling of load spikes thanks to an approach based on virtual machines which can be promptly activated when required [2]. In case of load spikes extra nodes are allocated to applications willing to pay more, based on a free market of nodes. Contrary to Oceano this project does not assume any statically allocated resources and looks more generic with respect to the managed applications though this aspect has not been demonstrated.

In [3, 4], the authors propose a self-optimized dynamic provisioning algorithm that specifically targets a cluster of databases. Regarding load spikes the system always provisions a set of unused nodes with database instances kept within a given range of freshness with respect to the active database instances. This contributes to improve the latency of provisioning operations. Furthermore oscillations are explicitly prevented as a result of a delay-aware allocation mechanism of database replica.

Cataclysm is a hosting platform for Internet service which features dynamic provisioning through a dynamic partitioning of nodes between the running applications and a adaptive size-based admission control mechanism which takes advantage of a request classifier to optimally degrades the service quality in case of overloads [5, 6]. The provisioning algorithm is based on a basic model of clustered network services. Cataclysm has been specially designed to absorb extreme overloads: the size-based admission controller prevents the system from thrashing as a result of accepting too many requests, additionally taking advantage of a request classifier to maximize the revenue during overloads, while the dynamic provisioning algorithm adds extra resources in case of overloads. The provisioning algorithm relies on a coarse-grained modeling of simple Internet services. The strength of Cataclysm is the cooperation of admission control and dynamic provisioning as components of an integrated resource management system. It assumes simple Internet services structures where the database back-end is statically provisioned.

Besides the above-mentioned heuristics-based approaches, another category of work on resource management of Internet services has studied mathematical characterization and analytical modeling of the systems [7, 8, 9, 10, 11].

For instance, in [8, 12], authors propose a model for multi-tier Internet applications. This model captures the structure and the behavior of Internet applications built as cooperative entities (i.e. entities in series) thanks to a network of queues. Transitions between queues standing for two connected tiers are probabilistic. Indeed this allows the model to capture requests processing paths (including caching mechanisms) through appropriate values for these transition probabilities. Replication and load-balancing, concurrency limits and requests classification and differentiation are taken into account as enhancements over the baseline model. The effectiveness of the model to achieve accurate capacity planning is demonstrated in a dynamic provisioning scenario in which parameters of the model are determined by mean-value analysis.

Finally, another category of work [13][14][15] as studied JMS performances. Regarding JMS performance, [13] provides an analysis of the throughput performance of JMS Using Websphere-MQ. [14] analyses a specific performance problem: The Message Waiting Time for the Fiorano-MQ Server. [15] describes a QoS Evaluation of JMS, it examines the impact of JMS attributes on performance.

9 Conclusion and future work

Providing a scalable and efficient Message Oriented Middleware is an important topic for today's computing environments. This paper analyses the performance of a Message Oriented Middleware and proposes 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 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.

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.: Océano-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