Improving the performances of JMS-based applications

Christophe Taton* and Nöel De Palma

Institut National Polytechnique de Grenoble
Grenoble, France
E-mail: christophe.taton@inrialpes.fr
E-mail: noel.depalma@inrialpes.fr
*Corresponding author

Sara Bouchenak

Université Grenoble I
Grenoble, France
E-mail: sara.bouchenak@inrialpes.fr

Daniel Hagimont

Institut National Polytechnique de Toulouse
Toulouse, France
E-mail: Daniel.Hagimont@enseeiht.fr

Abstract: In the Java world, a standardised interface exists for Message-Oriented Middleware (MOM): Java Messaging Service or JMS. 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 policies 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 propose a solution based on autonomic computing to (1) dynamically adapt the load distribution among the servers (load-balancing aspect) and (2) dynamically adapt the replication level (provisioning aspect). We present an evaluation that shows the impact of these rules on the performances and behaviour of the dynamic provisioning of replicated queues.

Keywords: message-oriented middleware; MOM; Java messaging service; JMS; autonomic management; self-optimisation; performance optimisation.

Reference to this paper should be made as follows: Taton, C., De Palma, N., Bouchenak, S. and Hagimont, D. (2009) 'Improving the performances of JMS-based applications', *Int. J. Autonomic Computing*, Vol. 1, No. 1, pp.81–102.

Biographical notes: Christophe Taton has been a PhD student at the Sardes Project, INRIA, France since 2005. His research topic mainly focuses on self-optimising techniques for autonomic systems, and more precisely on dynamic provisioning as one self-optimising technique to address performance and energy issues.

Noël De Palma obtained his PhD in 2001 in the area of component deployment and reconfiguration. He has been an Associate Professor since 2002 at the INPG University in Grenoble, France, where he teaches distributed systems. He is also a permanent member of the Sardes Project (INRIA Rhône-Alpes). He has worked on autonomic computing since 2003 and he is an architect of the Jade autonomic system.

Sara Bouchenak is an Associate Professor of Computer Science at the University of Grenoble I, France, which she joined in 2004. She conducts research on distributed systems as a member of the Sardes research group at INRIA. She received a Master's of Science in Computer Science from the University of Grenoble I in 1998, and a PhD in Computer Science from the Institut National Polytechnique de Grenoble, France, in 2001. She was a postdoctoral researcher at EPFL, Switzerland, in 2003. She is a member of EuroSys, the European Chapter of ACM-SIGOPS.

Daniel Hagimont is a Professor at Polytechnic National Institute of Toulouse, France and a member of the IRIT laboratory, where he leads a group working on operating systems, distributed systems and middleware. He received a PhD from Polytechnic National Institute of Grenoble, France in 1993. After a postdoctorate at the University of British Columbia, Vancouver, Canada in 1994, he joined INRIA Grenoble in 1995. He took his position of Professor in Toulouse in 2005.

1 Introduction

Message-Oriented Middleware (MOM) is a well-recognised technology that enables loosely coupled software interactions. MOM-based applications cooperate using asynchronous and reliable communications. Messages are the only way for software to synchronise and exchange data. These communication properties are supported by message queues, which are staging areas containing messages that have been sent and are waiting to be read.

With the emergence of the internet, MOMs are used intensively in the context of server-side applications. It is well known that internet applications have to deal with unpredictable loads. This leverages the interest of highly scalable and highly available MOMs.

This work illustrates the key parameters that influence the performances of a message queue and describes a solution to increase these performance autonomously while minimising the amount of required resources. We show that a classic queue clustering and load balancing strategy can provide a linear speedup. However, this strategy is sometimes rather inefficient and may waste precious resources. The impact of clustering is strongly influenced by the allocation of client connections to message queues.

Our solution to tackle this problem is based on a novel replication strategy for message queues controlled by a specific load balancing mechanism and combined with a dynamic resource management system. On the one hand, our load balancing mechanism improves the distribution of client connections in the message queues according to the state of each replicated queue (thus, avoiding a lazy or an overloaded queue). On the other hand, the resource manager dynamically adjusts the number of machines and queues based on the current system load. Furthermore, a typical management policy

deploys a fixed number of queues on a fixed set of machines. This solution may waste resources if the number of queues is overestimated compared to the system load, but may also lead to performance problems if underestimated. The unpredictable shape of internet load underlines the need for the dynamic and autonomic adjustments of queue replicas.

This paper targets the improvement of a load-balancing mechanism for replicated queues, as well as the dynamic provisioning of these queues. The benefit is an increase of message queue performances while using the minimal set of required resources. To fulfil these goals, we provide an autonomic system that fairly routes client connections among the queues and dynamically creates or destroys queues to face the load variation.

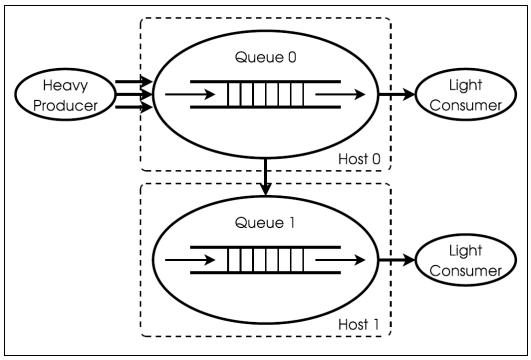
This paper is organised as follows: Sections 2 presents the notion of a replicated queue. Sections 3 and 4 present a model and analyse different cases that may occur with single queues and with replicated queues. Sections 5 and 6 present the control rules and control loop. Section 7 shows a performance evaluation. Finally, Section 8 presents the related work and Section 9 draws our conclusion and outlines future work.

2 Replicated queues

A replicated queue is made of a set of identical queues that know each other. When a queue inside a replicated queue is overloaded, the queue is authorised to distribute some messages it cannot process to the other queues in the cluster. In the same vein, if a queue becomes empty and lazy, it is authorised to request messages from the other queues. Thus, all queues in a replicated queue may equilibrate the level of pending messages with each other, depending on the respective number of client requests. To summarise, a replicated queue ensures that no queue is underloaded while some of the others are overloaded and tend to equilibrate the number of pending messages waiting in the replicated queue.

Figure 1 illustrates a replicated queue that is composed of two queues. A producer sends a large amount of messages through its local queue (q_0). A consumer also connects to the queue (q_0), but only consumes a small amount of messages. The queue (q_0) becomes quickly overloaded and pushes messages to another queue in the cluster (q_1). This latter queue is actually used by a light message consumer which requests messages that queue (q_1) could indeed not provide by itself. Thus, the consumer on (q_1) can then retrieve messages at a good rate, while the messages on (q_0) are consumed much faster.

Figure 1 A replicated queue



3 Performances of single queues and replicated queues

We illustrate in this section the behaviour and performances of single queues and replicated queues. We show the impact of client connections on the performances.

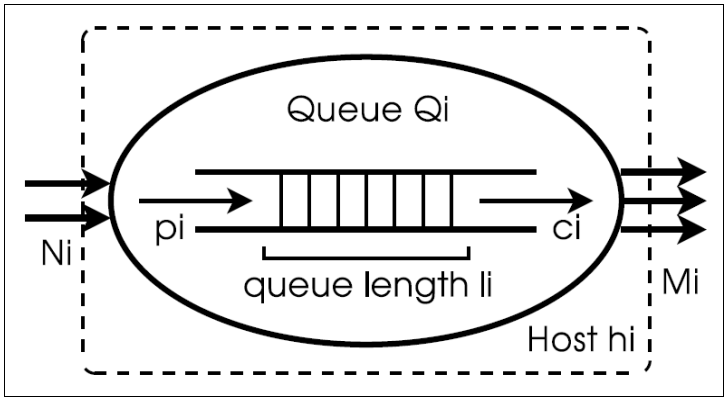
3.1 Single queue

N_i message producers and M_i message consumers are connected to a single queue Q_i. Messages are injected in the queue at rate p_i and retrieved at rate c_i. The number of pending messages to be consumed in the queue is described by the queue length $l_i \geq 0$, which is characterised by:

$$\Delta l_i = p_i - c_i.$$

A well-balanced rate between the producers and consumers is required for the *stability* of the queue. A queue is stable if the queue length l_i remains constant. Thus, $\Delta l_i = 0$ when the producers and consumers are balanced and work with similar throughputs. A queue Q_i can also be *unstable*. It is said to be *flooded* if the rate of message production is higher than the rate of message consumption. If $\Delta l_i > 0$, then the queue length l_i increases and the queue eventually saturates. Thus, the message production rate of the producers will be reduced. However, if $\Delta l_i < 0$, then the queue is *unstable* and is said to be *draining*. The message consumption rate is higher than the message production rate. This means that the queue length l_i is heading to 0. In this case, the message consumers' throughputs fall and the consumers may have to wait for messages to consume. The message production and consumption throughput are related to the number of message producers and consumers. Thus, the ratio between the number of message producers and the number of message consumers is a key parameter that heavily influences the performances of a single queue.

Figure 2 A single queue Q_i



3.2 Replicated queue

The aim of a replicated queue is to maximise its throughput while maintaining its stability. Replicated queues always try to empty flooded queues and fill draining queues. A replicated queue Q_c is then composed of a set of single queues $Q_i (i \in [1..k])$ used to

balance the requests induced by the message producers and message consumers. All the queues in a replicated queue run on different servers, work together and can exchange messages as a way to share their load. We consider the realistic assumption where MOMs do not allow the modification of a client's connection to a particular single queue in the replicated queue once it has been set. Thus, the control logic can only choose the best queue for a client when the connection is opened.

N_c message producers and M_c message consumers have a connection to the replicated queue Q_c. Each queue Q_i is responsible for serving a part of message producers (N_i) and a part of message consumers (M_i):

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

We suppose that x_i and y_i are the fraction of the message producers and consumers that are connected to the queue Q_i. Then, the distribution of the client's connections between the queues Q_i are characterised by:

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

Three indicators are useful for sensing the health of a replicated queue Q_c. These indicators are an aggregation of single-queue indicators: (1) p_c is the global message production level, (2) c_c is the global message consumption level and (3) l_c is the virtual replicated queue length that aggregates the length of all the single queues composing the replicated queue Q_c:

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

If $\Delta l_c = 0$, then the replicated queue Q_c is globally stable while we may observe local instabilities if one of its queues is draining or flooded. The length of a replicated queue rises and the replicated queue is saturated by messages if $\Delta l_c > 0$. On the other hand, the message consumers wait if the replicated queue is empty. This case occurs if $\Delta l_c < 0$, which means that the length of the replicated queue is decreasing. We see that the behaviour of a replicated queue follows the same laws as single queues.

In the following, we detail some interesting performance behaviours based on the assumption that the replicated queue Q_c is globally stable. We make the assumptions that:

- all the machines are identical in terms of processing power
- the message production rate is the same for all the producers and the message consumption rate is the same for all the consumers
- all messages have similar characteristics (size, *etc.*).

The first case is an example of optimal client distribution which occurs for a queue Q_c when clients are fairly distributed among the k queues Q_i. In this context, we have:

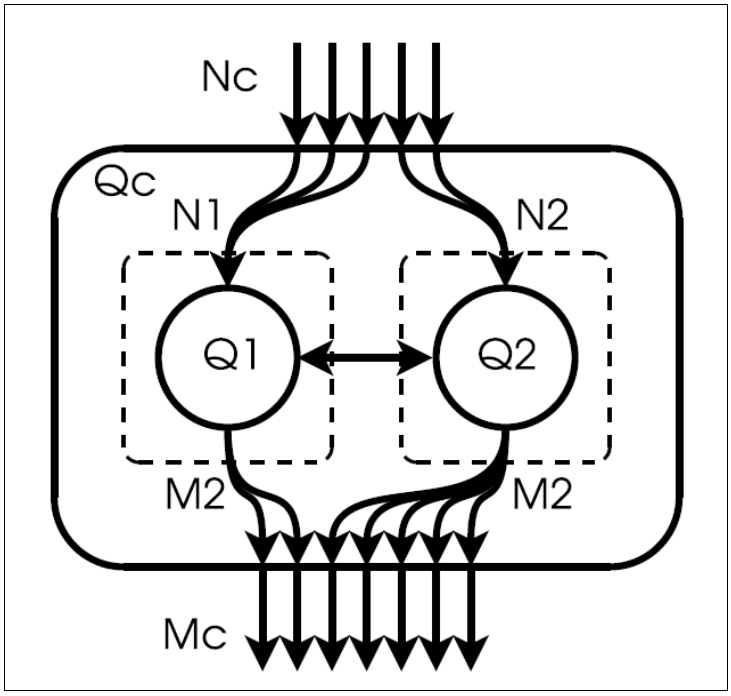
$$\begin{cases} x_i = 1/k \\ y_i = 1/k \end{cases}, \begin{cases} N_i = \frac{N_c}{k} \\ M_i = \frac{M_c}{k} \end{cases}.$$

Under these assumptions, the replicated queue behaviour shows a quasi-linear speedup. This is the best performance scenario that we can expect. The reason is that there are no internal queue-to-queue message exchanges because the replicated queue is well-balanced (there is no unstable queue inside the replicated queue).

Another scenario consists of the worst client distribution. This is the case if a single queue has all its connections related to the message producers or message consumers. In Figure 3, this worst case implies that the queue Q_1 is requested for all message production, whereas Q_2 is only requested by consumers. The produced messages must be transmitted by Q_1 to Q_2 that, in turn, delivers messages to the message consumers. This worst case scenario is characterised as follows:

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

Figure 3 A clustered queue Q_c



The third scenario is related to local instabilities. This case can be observed when some queues Q_i of Q_c are unbalanced. For instance, a local instability can be observed in the scenario depicted in Figure 3, where Q_c is composed of two standard queues Q_1 and Q_2. Let us suppose we have the following connection repartition which implies that Q_1 is flooded and enqueues messages, while Q_2 is draining and has its consumer clients waiting. Despite the local instability, Q_c enforces its global stability by transferring some messages internally from Q_1 to Q_2:

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

In general, this case occurs when the ratio between the number of producers and the number of consumers connected to a single queue Q_i is not balanced:

$$x_i \neq y_i.$$

The last scenario occurs if the repartition of connections is non-uniform. For instance, in Figure 3, this occurs if more clients are connected to Q_1 than to Q_2.

Assuming that queue Q_1 serves two-thirds of the load while queue Q_2 serves one-third, Q_1 may be overloaded while Q_2 is idle. The load is unfairly balanced within the queues, whereas the replicated queue is globally and locally stable. This scenario induces reduced performances:

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

Notice that these scenarios may all happen since the number of clients connected to the system evolves in an uncontrolled manner. For instance, the queue can be flooded for a period; we then assume that it will get inverted and drain afterwards, thus providing global stability over time.

4 Replicated queue management

In this section, we provide some details about replicated queue management. Dynamic provisioning depends on the load of the replicated queue. The ratio between its current number of clients and its capacity represents the load of a single queue (L_i). The capacity C_i of a single queue Q_i is represented by the optimal number of clients connected to the queue (*i.e.*, the number of clients that maximises the queue performances):

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

where:

$L_i < 1$: queue Q_i is not loaded enough, resources are being wasted and the queue may accept additional connections to reach an optimal throughput

$L_i > 1$: queue Q_i saturates, the queue is crippled by a heavy load and the message throughput is non-optimal and eventually leads to thrashing

$L_i = 1$: queue Q_i is fairly loaded and delivers its optimal message throughput.

Queue replication management is derived by the queue parameters that were presented previously. The behaviour of a replicated queue Q_c is dictated by its aggregated capacity C_c and its global load L_c. The load of a replicated queue follows the same law as single queues. However, the aggregated capacity $C_c = \sum_{i=1}^{k} C_i$ of a replicated queue can be managed by controlling the number k of the single queues that compose a replicated queue:

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

where:

If $L_c < 1$ = the replicated queue is lazy: each single queue inside the cluster will be underloaded if the client's connection distribution is optimal. In the case of a non-optimal connection distribution, despite the replicated queue being lazy in a whole, a single queue may be heavily loaded. Some single queues may be garbaged from the replicated queue if the load is below a given threshold.

When $L_c > 1$ = the replicated queue is overloaded: a number of single queues in the replicated queue are overloaded, even if the allocation of a client's connection corresponds to a fair sharing profile. In this case, one or more single queues will be added in the replicated queue.

5 A self-optimising replicated queue

In this section, we present the design of an autonomic ability which targets the optimisation of a replicated queue. The optimisation takes place in two steps: (1) the optimal load-balancing of a replicated queue and (2) the dynamic provisioning of queues in a replicated queue.

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

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

The implementation of these optimisations relies on the model of replicated queue performance, which has been presented in the previous sections.

5.1 Control rules

The global clients' distribution D of the replicated queue Q_c is captured by the fractions of the message producers x_i and consumers y_i. The optimal clients' distribution D_{opt} is realised when all queues are stable ($\forall i\ x_i = y_i$) and 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 characterised by a mismatch between the fraction of the message producers x_i and consumers y_i on a standard queue. The purpose of this rule is the stability of all standard queues to minimise internal queue-to-queue message transfers:

(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 to optimise resource usage.*

Global provisioning rules control the load applied to the whole replicated queue. These rules target the optimal size of the replicated 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-optimisation of replicated queues. 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 autonomic behaviour.

5.2.1 System events and controls

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

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

Thus, a replicated queue generates four types of events that we can use to control and optimise the system:

1 join(Producer)

2 leave(Producer, Q_i)

3 join(Consumer)

4 leave(Consumer, Q_i).

We implement algorithms controlling the distribution of clients and the provisioning of the replicated queue as handlers to these events. The handlers implementing the control rules are depicted in Algorithms 1 and 2.

Algorithm 1 Client joining algorithm

```
on join(ClientType ∈ {Producer, Consumer}, Q_c)
if (L_c ≥ 1) and not Islnhibited() then
   // The replicated queue is overloaded
   // An additional queue is required
   Q_{k+1} ← NewQueue()
   AddQueue(Q_c, Q_{k+1})
   // Inhibits provisioning to allow stabilisation
   StartInhibition(InhibitionDelay)
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 ∈ Q_c)
if (IsMarked(Q_i, "to be removed") and IsEmpty(Q_i) then
   RemoveQueue(Q_c, Q_i)
   DestroyQueue(Q_i)
   // Inhibits provisioning to allow stabilisation
   StartInhibition(InhibitionDelay)
end if
if (L_c < 1) and not Islnhibited() then
   Q_i = ElectRemovableQueue(Q_c)
   if Q_i ≠ null then
      Mark(Q_i, "to be removed")
   end if
end if
```

The ElectQueue(ClientType) function chooses the queue that is farthest from the targeted client distribution. Thus the elected queue Q_i provides the best step towards the optimal configuration. When considering a new client that is a message producer (respectively consumer), the gap is evaluated with $1/k - x_i$ (respectively 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 and (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 lets Q_c be overloaded. Thus, the condition to allow Q_i's removal is:

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

In fact, we complement the replicated queue load L_c defined in the replicated queue model with online load estimations based on the consumption of system resources to reflect the replicated queue load more accurately. Thus, as provisioning operations induce local measurement instabilities, we introduce an inhibition delay with StartInhibition(Delay), which forbids new provisioning operations in order to allow the system to stabilise. Once the inhibition delay expires, provisioning operations are allowed again.

The following section gives the implementation details about these algorithms.

6 Implementation details

6.1 Technical context

We choose JMS as technical background for our work. It provides an Application Programming Interface (API) used to manipulate the following artefacts:

- ConnectionFactory – an object that a client[1] looks up for connection factory in a naming service[2] and uses to create a connection to the JMS provider. Depending on the type of message, the users will use either a queue connection factory or topic connection factory (point-to-point or publish subscribe mechanism).
- Connection – once a connection factory is obtained, a connection to a JMS provider can be created. A connection represents a communication link between the application and messaging server.
- Destination – an object that encapsulates the identity of a message destination, which is where messages are delivered and consumed. It is either a queue or a topic.
- MessageConsumer – an object created by a session. It receives messages sent to a destination. The consumer can receive messages synchronously (blocking) or asynchronously (nonblocking) for both queue and topic-type messaging.
- MessageProducer – an object created by a session that sends messages to a destination. The user can create a sender to a specific destination or a generic sender that specifies the destination at the time the message is sent.
- Message – an object that is sent between the consumers and producers. JMS specifies five types of messages (text message, map message, bytes message, stream message and object message).

- Session – represents a single-threaded context for sending and receiving messages. A session is single-threaded so that messages are serialised, meaning that messages are received one by one in the order sent.

For our experiments, we choose Java Open Reliable Asynchronous Messaging (JORAM). It is an 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 replicated queue mechanisms.

6.2 Anatomy of self-optimisable replicated queues

Engineering an autonomic system requires implementing one or more control loops that regulate the managed system. Control loops are based on an analysis/decision logic to trigger a system's reconfiguration together with a model of the system. This logic relies on two connections to the managed system: sensors to watch the state of the system and actuators to reconfigure it. In our context, the sensors are used to know the current number of message producers and consumers in each queue and detect the global load of a replicated queue (*i.e.*, if the queue is overloaded or underloaded). The system's model is used to know where the servers are deployed, where the queues are deployed and what their configurations are. The analysis/decision logic is responsible for routing new client connections to the most appropriate queues leading to the optimal client distribution and dynamically provision a queue in the system, if necessary. The actuators are used to allocate/deallocate servers and create or destroy queues in servers.

6.3 The self-management logic

As a first step, we consider that clients create only one session by connection.[3] We have extended the ConnectionFactory to provide a Clustered Connection Factory (CCF). This CCF will control the client's connection to the best queue replicate, depending on the load of the system. It will also regulate the dynamic provisioning of the queues in the system.

6.3.1 Cluster connection factory

The CCF is registered in the Java Naming and Directory Interface (JNDI) instead of a regular connection factory. This allows the regulation of the connection of producers and consumers among servers. The CCF logic is then run when a client requests or releases a connection:

- *createConnection(...)* – to create the connection with the best server, it requests a 'ResourcesManager', which elects a server according to the current state of the system (the servers and the load of each queue in terms of the producers and consumers) and actually provisions the system with the server, if necessary. This method takes the type of the client as a parameter (Producer or Consumer).
- *closeConnection(...)* – closes the connection to the server and notifies the ResourceManager so it may size down the replicated queue, if necessary.

6.3.2 ResourceManager

The Resource/ReplicationManager maintains the system's model (*i.e.*, the number of servers currently used, the number of clients connected to each server, their type). This model is updated each time a client (producer or consumer) opens or closes a connection from the CCF and when the system is reconfigured. When a connection is requested, the ResourceManager elects a server and takes into account the capacities in terms of the clients and the current load of the replicated queue. If the replicated queue is overloaded, the Resource/ReplicationManager uses the procedures *NewQueue()* and *AddQueue()* to launch a JORAM server on a free host and create a queue linked to the replicated queue 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.

7.1 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, JORAM4.3.21*
- *Ethernet Gigabit network.*

In each experiment, the measurements were taken with Java Management Extensions (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 the consumers on the queue since its creation and *MessageCounter*, which is the number of messages presently waiting in the queue. The JMX probes read these attributes every second.

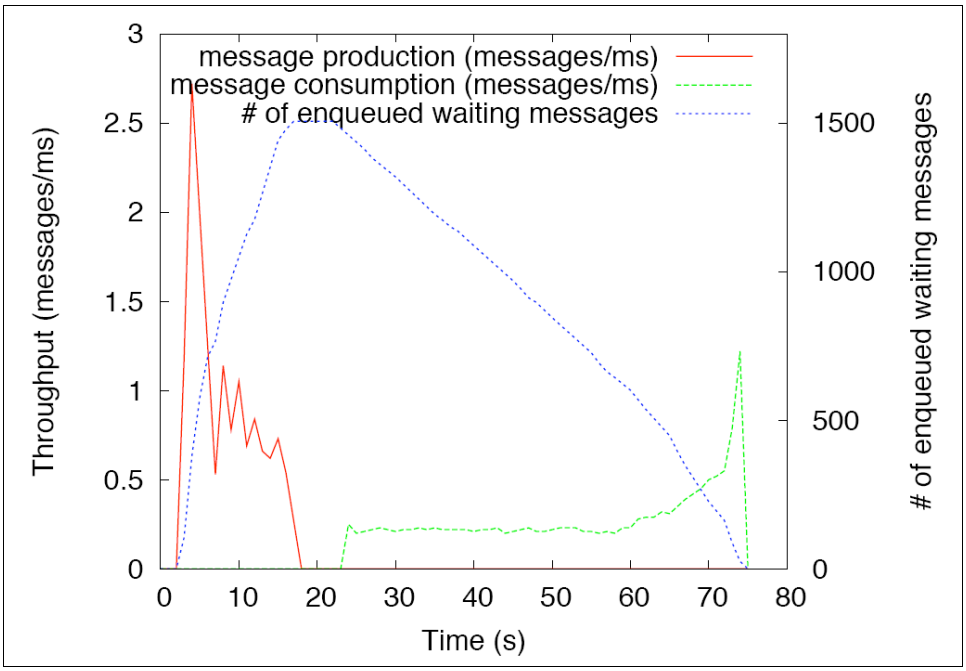
In the following experiments, each JORAM queue was located on a distinct node. The queues were running in a persistent configuration. The producers and consumers were transactional with a commit between each message. The Java Virtual Machine hosting each queue was able to use 1536 Mb of memory. The Garbage Collector was disabled to prevent random hits on performance. The size of the JMS messages used was 1 Kb. The network was not considered to be a 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 5 and last 5 s as a way to account only for the stable part of the process.

7.2 The number of waiting messages factor

This experiment aims to show the impact of the number of messages waiting in the queue on the performance. In a first step, the producers write 1500 messages in a single queue, while in a second step, the 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.

Figure 4 The impact of the waiting messages on the performance (see online version for colours)



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 the 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 15 000 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 three 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.

Figure 5 The behaviour of a single queue with one message producer and one message consumer (see online version for colours)

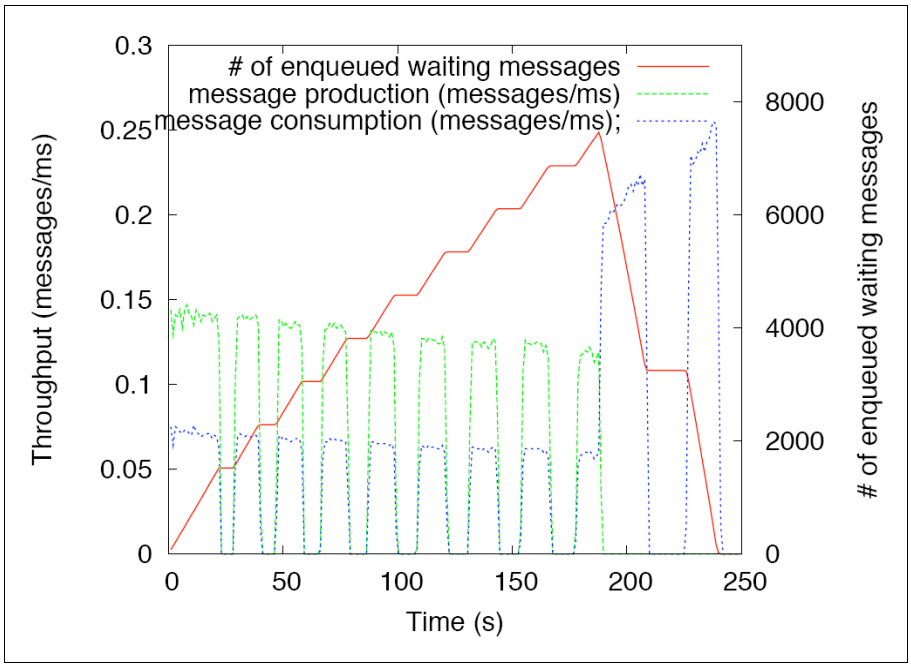
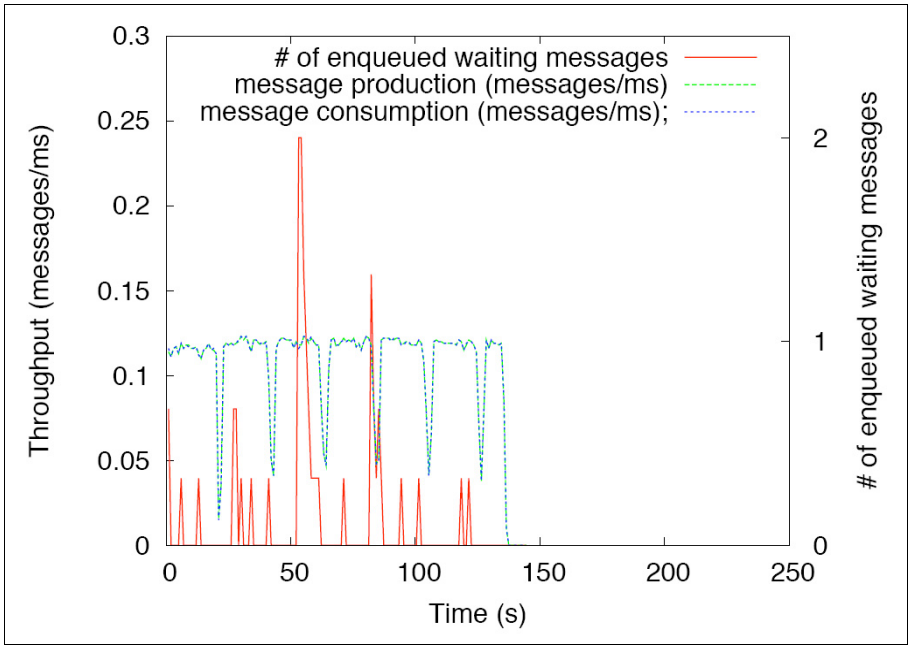


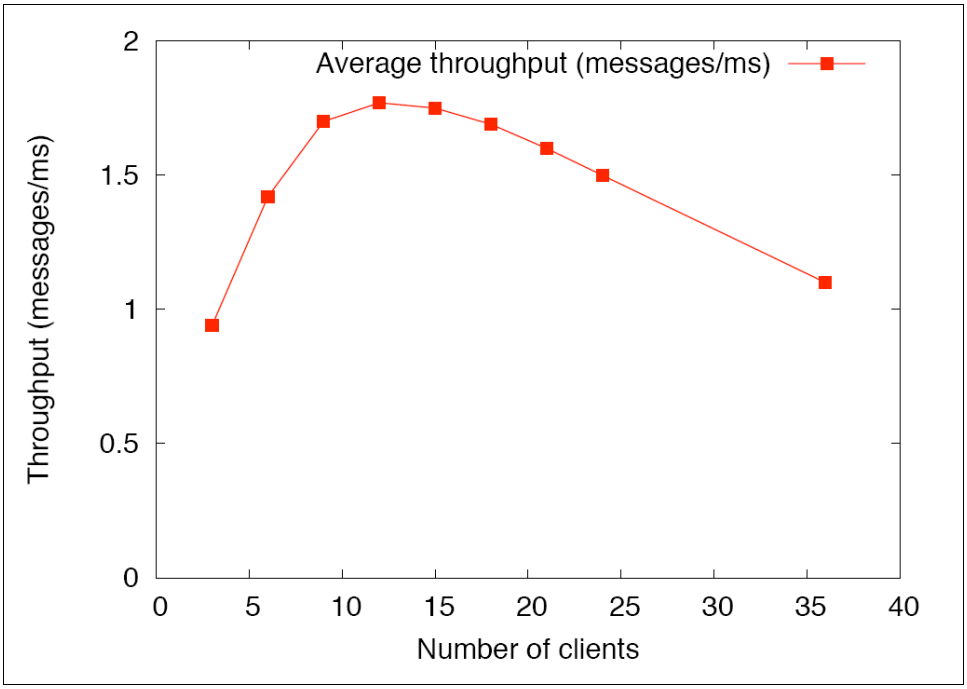
Figure 6 The behaviour of a single queue with one message producer and two message consumers (see online version for colours)



7.3 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 that 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 in Figure 7. These results account for the strong interest in dynamic provisioning and optimisation of the load-balancing of clustered queues in order to always provide the best clustered queue size and client distribution for best performance.

Figure 7 Capacity of a standard single queue (see online version for colours)



7.4 Load-balancing optimisation

The following presents an evaluation of the queue cluster load-balancing optimisation that fairly distributes client connections among the queues. For this evaluation, we expose a queue cluster composed of two queues to four message producers and eight message consumers. A single message producer emits 10 000 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 optimised load balancer. When using the original load-balancing strategy, we observe a noticeable instability with a higher message production rate compared to 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 optimisation, the queue cluster presents a very stable and balanced behaviour. Indeed, the message production rate and message consumption rate both reach 0.35 messages/ms.

Figure 8 The standard JORAM queue cluster load-balancing strategy (see online version for colours)

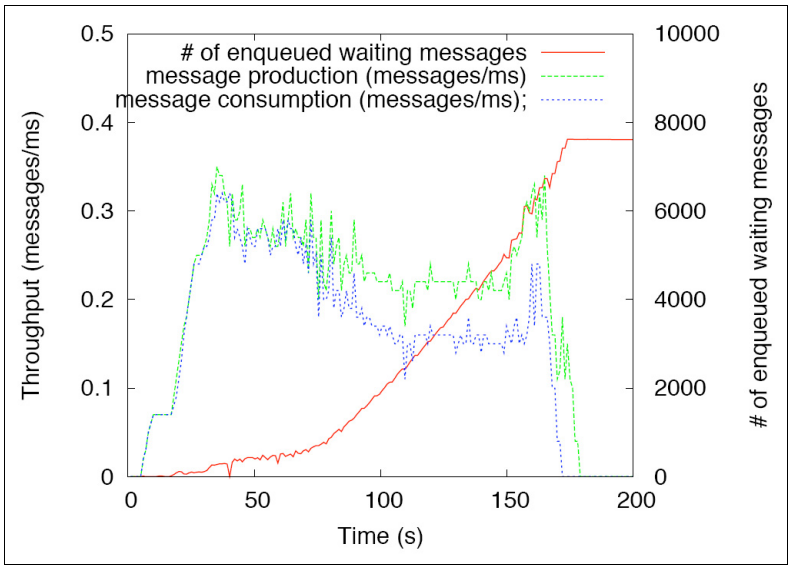
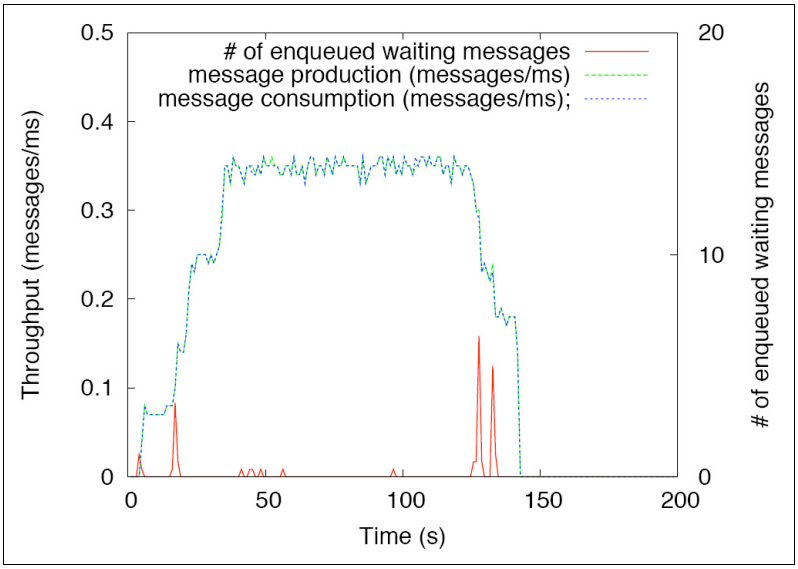


Figure 9 Optimised queue cluster load-balancing (see online version for colours)



7.5 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 of five message producers and ten message consumers. As in the previous experiment, a message producer generates 10 000 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 10 s. The queue cluster is kept stable by creating clients 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 stabilises quickly after the second step at around 50 s, 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 in absorbing the workload. Then, at around 120 s, 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.

Figure 10 The static provisioning of a clustered queue (see online version for colours)

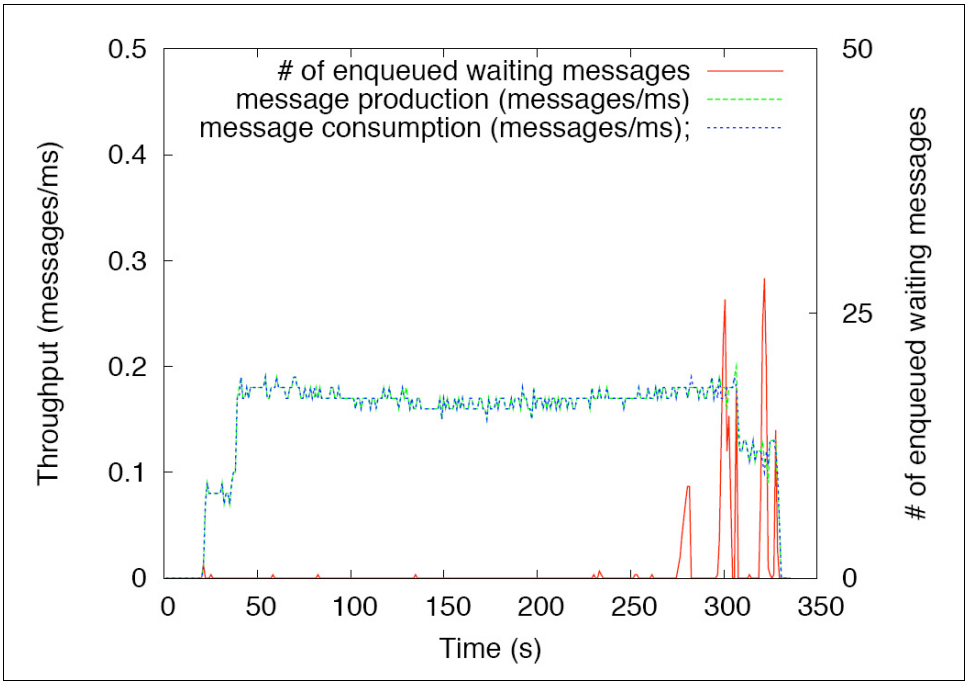
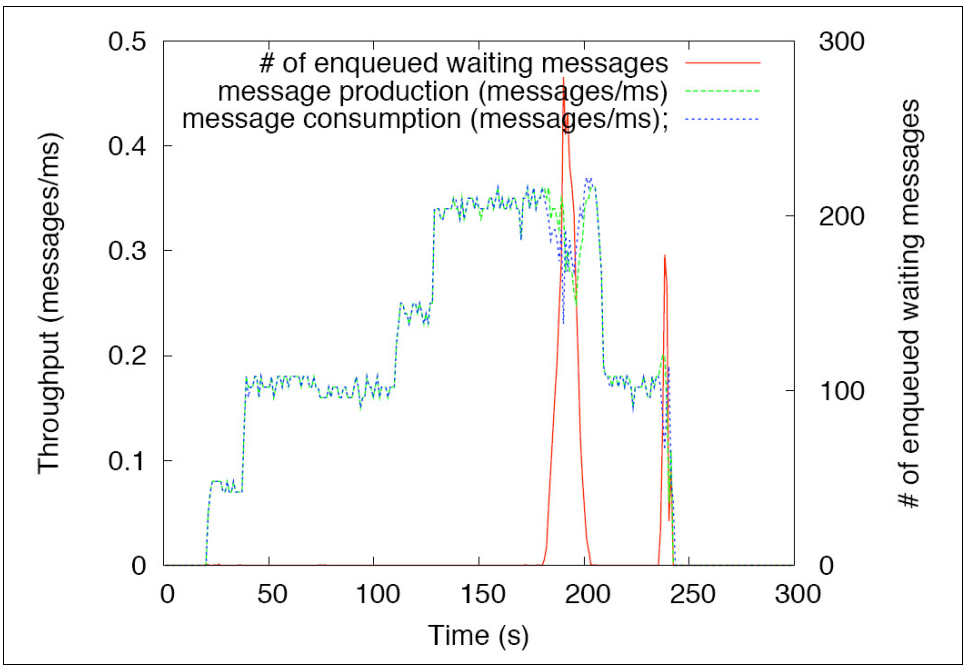


Figure 11 The dynamically provisioned clustered queue (see online version for colours)



7.6 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 with 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 the resource management of internet services falls into different categories.

A first category that has focused on studying the resource management of internet services has considered the management of a dynamically extensible set of resources, where the infrastructure can dynamically grow or shrink (Appleby *et al.*, 2001; Norris *et al.*, 2004; Soundararajan and Amza, 2005; Soundararajan *et al.*, 2006; Urgaonkar and Shenoy, 2004; 2005).

Oceano provides an adaptive hosting environment with a dynamic partitioning of the resources among the running applications (Appleby *et al.*, 2001). This dynamism allows the system to react to load peaks by increasing the partition size of the concerned application and shifting unused resources from underloaded 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 it specifically targets the fast handling of load spikes thanks to an approach based on virtual machines which can be promptly activated when required (Norris *et al.*, 2004). 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 Soundararajan and Amza (2005) and Soundararajan *et al.* (2006), the authors proposed a self-optimised 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 the improvement of the latency of provisioning operations. Furthermore, oscillations are explicitly prevented as a result of an allocation mechanism of database replica which takes replicas allocation latency into account.

Cataclysm is a hosting platform for internet services which features dynamic provisioning through a dynamic partitioning of nodes between the running applications and an adaptive size-based admission control mechanism which takes advantage of a request classifier to optimally degrade the service quality in case of overloads (Urgaonkar and Shenoy, 2004; 2005). 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 maximise the revenue during overloads, while the dynamic provisioning algorithm adds extra resources in case of overloads. The provisioning algorithm relies on a coarse-grained modelling 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 abovementioned heuristics-based approaches, another category of work on the resource management of internet services has studied mathematical characterisation and analytical modelling of the systems (Urgaonkar *et al.*, 2005; 2007; Chandra *et al.*, 2003; Zhang *et al.*, 2007; Stewart and Shen, 2005).

For instance, in Urgaonkar *et al.* (2007; 2005), the authors proposed a model for multitier internet applications. This model captures the structure and behaviour of internet applications built as cooperative entities (*i.e.*, entities in series) thanks to a network of queues. Transitions between the queues standing for two connected tiers are probabilistic. Indeed, this allows the model to capture request 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 in achieving accurate capacity planning is demonstrated in a dynamic provisioning scenario in which the parameters of the model are determined by mean-value analysis.

Finally, another category of work (Henjes *et al.*, 2006; Menth and Henjes, 2006; Chen and Greenfield, 2004) studied JMS performances. Regarding JMS performance, Henjes *et al.* (2006) provided an analysis of the throughput performance of JMS using Websphere-MQ. Menth and Henjes (2006) analysed a specific performance problem: the message waiting time for the Fiorano-MQ Server. Chen and Greenfield (2004) described a QoS evaluation of JMS and examined the impact of JMS attributes on performance.

9 Conclusion and future work

Providing a scalable and efficient MOM is an important topic for today's computing environments. This paper analysed the performance of a MOM and proposed a self-optimisation algorithm to improve the efficiency of the MOM infrastructure.

This optimisation takes place in two parts: (1) the optimisation of the clustered queue load-balancing and (2) 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 optimises the resource usage inside the clustered queue.

We described (1) the key parameters impacting the performance of MOM and (2) the rules that control these parameters for optimal performances. This paper also presented an evaluation that shows the impact of these parameters on the performances and behaviour of a 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 the session data to another queue.

References

Appleby, K., Fakhouri, S.A., Fong, L.L., Goldszmidt, G.S., Kalantar, M.H., Krishnakumar, S., Pazel, D.P., Pershing, J.A. and Rochwerger, B. (2001) 'Oceano-SLA based management of a computing utility', *Proceedings of Integrated Network Management*, pp.855–868.

Chandra, A., Gong, W. and Shenoy, P. (2003) 'Dynamic resource allocation for shared data centers using online measurements', *Proceedings of the Eleventh IEEE/ACM International Workshop on Quality of Service (IWQoS 2003)*, Monterey, California, June.

Chen, S. and Greenfield, P. (2004) 'Qos evaluation of jms: an empirical approach', *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS '04) – Track 9*, IEEE Computer Society, Washington DC, USA.

Henjes, R., Menth, M. and Zepfel, C. (2006) 'Throughput performance of java messaging services using websphereMQ', *5th International Workshop on Distributed Event-Based Systems (DEBS)*, Lisboa, Portugal, July.

Menth, M. and Henjes, R. (2006) 'Analysis of the message waiting time for the fioranoMQ JMS server', *26th International Conference on Distributed Computing Systems (ICDCS)*, Lisboa, Portugal, July.

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

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

Soundararajan, G., Amza, C. and Goel, A. (2006) 'Database replication policies for dynamic content applications', *First EuroSys Conference (EuroSys 2006)*, Leuven, Belgium, April.

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

Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M. and Tantawi, A. (2007) 'Analytic modeling of multitier internet applications', *ACM Transaction on the Web*, Vol. 1, No. 1, p.2.

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

Urgaonkar, B. and Shenoy, P. (2004) 'Cataclysm: handling extreme overloads in internet services', Technical report, Department of Computer Science, University of Massachusetts, November.

Urgaonkar, B. and Shenoy, P.J. (2005) 'Cataclysm: policing extreme overloads in internet applications', *Proceedings of the 14th International Conference on World Wide Web, (WWW'05)*, Chiba, Japan, May, pp.740–749.

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

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

Notes

1 A producer or a consumer.

2 JNDI.

3 We assimilate the creation of sessions and the creation of connections.