Elastic Message Queues

Ahmed El Rheddane*, Noël de Palma*, Alain Tchana†, Daniel Hagimont†
*LIG/UJF — Grenoble, France
{ahmed.elrheddane, noel.depalma}@imag.fr
†IRIT/ENSEEIHT — Toulouse, France
{alain.tchana, daniel.hagimont}@enseeiht.fr

***Abstract*—Today's systems are often distributed, and connecting their different components can be challenging. Message-Oriented-Middleware (MOM) is a popular tool to insure simple and reliable communication. With the ever growing loads of today's applications, MOMs needs to be scalable. But as the load changes, static scalability often underuses the resources it requires. This paper presents an elastic message queuing system leveraging cloud's on-demand resource provisioning, which allows the use of just enough resources to handle the current load. We will detail when and how provisioning decisions are made, and show the result of our system's evaluation on Amazon EC2 public cloud. This work is based on Joram, an open-source JMS compliant MOM and is now part of its distribution on OW2 consortium's website.**

***Keywords*-JMS; message queues; elasticity; cloud computing**

I. Introduction

In today's interconnected world, distributed systems are ubiquitous. These systems often run on heterogeneous devices and connecting them in a simple and reliable manner is challenging. Message-Oriented Middleware (MOM) is an established solution to this concern. It relies on exchanging messages as the only means for the distributed components to communicate, synchronize or coordinate. MOMs generally offer two communication paradigms: *one-to-one*, in which each produced message is consumed once and only once, this is done by message queuing; and *one-to-many* or *publish-subscribe* where each message is guaranteed to be received by all subscribed consumers via topics. MOMs have been standardized first in the Java world by the Java Message Service (JMS) API [1] and more recently by Advanced Message Queuing Protocol (AMQP) [2] that goes beyond the API level to specify the transport protocol. This work is based on Joram [3], an open-source JMS compliant MOM written in Java.

As distributed applications rely on MOMs, the latters should be scalable enough to support the formers' loads. One way of scaling MOMs is to allocate once and for all a large enough amount of resources to insure that wort case scenario loads can be handled. However, real life applications have variable and often bursty loads. Thus the statically allocated resources will be most of the time underused, which will reflect badly on the cost of the infrastructure. With the outburst of cloud computing, on-demand resource provisioning can be used to scale in a much *greener* way: if the load increases, extra resources can be allocated almost instantly; if the load decreases, we can get rid of no longer necessary resources.

In this work, we focus on enhancing the scalability of the one-to-one paradigm of our JMS compliant MOM, i.e., message queues. We particularly deal with the common case in which queues and consumers are stressed by the produced messages' load, and try to scale queues along with consumers depending on load variation. This should naturally be done while (i) maintaining the JMS compatibility as well as the cornerstones of MOMs that are (ii) asynchrony, i.e., the decoupling between producers and consumers and (iii) reliability, which basically makes sure that no message is ever lost. We first present a scalable solution which allows to statically scale the number of queues and manages load balancing between them. This solution is then enhanced to achieve elasticity, i.e., automatically scale the number of queues. We discuss when scaling decisions are made and the metrics they are based on. We also detail our provisioning policy, i.e., how scaling is carried out, which includes pre-provisioning virtual machine instances (VMs) and co-provisioning multiple queues on the same VM. Finally, we evaluate our elastic messaging solution and show the effect of each of our provisioning enhancements.

The rest of this paper is organized as follows: Section II presents a static setup of our scalability solution; Section III discusses the different aspects of our elasticity approach; in Section IV we evaluate the proposed solution; the related work is presented in Section V before concluding this work in Section VI.

II. Load-balanced queues

In Message-Oriented Middleware, a queue is used to store the produced messages until a message consumer retrieves them. Since the consumers often process the messages they receive, they cannot always cope with the production speeds imposed by the message producers, and messages soon begin to pend on the message queue. In this section, we propose a scalability mechanism that allows producers to seemlessly send messages to a pool of queues along with their consumers and distributes the messages between them. We first detail the scalability mechanism, then study its

scalability and finally present our flow control based load balancing policy.

A. Scalability mechanism

In order to achieve queues' scalability, we introduced the *alias queue*. An alias queue is a special queue on the producer's side, that automatically forwards the messages it is sent to another, generally distant, queue on the consumers' side, see Figure 1. It is set to write-only mode as the "real" destination, on which the messages are to be consumed, is the queue to whom the messages are forwarded. Thus, once a producer connects to our alias queue, we will be able to internally change the destination while maintaining the producer's connection to the same queue. We can also add or remove destinations, i.e., queues, and notify the alias queue to take our modification into consideration. The alias queue mechanism does not only insure JMS compatibility, it also guarantees a total decoupling between the producer and the consumers as it completely isolates the producer from the consumption system: the producer will always be able to send messages to its alias queue without taking into consideration any changes in the consumption rate or availability of consumer queues. The system's reliability is also increased as the alias queue includes a fail-over mechanism and can resend a given message to another queue if its initial destination is unavailable.

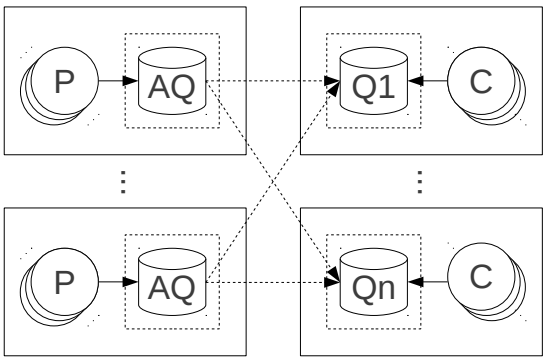

Figure 1: Alias queue principle

We will now compare the scalability of this load-balanced queues setup to that of a single queue.

B. Scalability Study

In this sub-section, we define the parameters that affect the performance of our system, first in the simple case of a single queue, before generalizing the results to the case of load-balanced queues.

1) Single Queue: Let p be the production rate on the queue and c the consumption rate. l being the length of the queue, i.e. the number of waiting messages, we have:

$$\Delta l = p - c$$

Depending on the result, three cases can be identified:

- $\Delta l > 0$: This means that the queue receives more messages than it is asked to deliver. The number of pending messages grows and we say that the queue is *unstable* and *flooded*. This will eventually cause the unavailability of the queue since it is allocated a finite memory.
- $\Delta l < 0$: In this case, the consumption rate is higher than the potential reception rate. The queue is still *unstable* and we say that it is *draining*. This means that the resources linked to this queue are not optimally utilized.
- $\Delta l = 0$: Here, the consumption rate matches the reception rate and the queue is *stable*. This is the ideal case that we aim to achieve.

The stability of a queue is thus defined by the equilibrium between the messages' production and consumption.

2) Load-Balanced Queues: In this case, the alias queues, to which the messages are sent, are wired to n queues, on which the messages are received. Let P be the total production rate on all the alias queues, c_i the consumption rates on each of the consumers' queues, and l_i their respective lengths. The scalability of our distributed system can be discussed on two different levels:

- *Global scalability*: Let L be the total number of waiting messages in all the consumers' queues. We have:

  $$L = \sum_{i=1}^{n} l_i$$

  and:

  $$\Delta L = P - \sum_{i=1}^{n} c_i$$

  The overall stability of our system is given by: $\Delta L = 0$. This shows that, globally, our system can handle the global production load. However, it does not guarantee that on each consumer queue, the forwarded load is properly handled. This will be guaranteed by *local scalability*.
- *Local scalability*: Depending on how we distribute the messages between the different queues, each would receive a ratio r_i of the total messages produced on the alias queues. Thus, for each $i \in \{1..n\}$ we have:

  $$\Delta l_i = r_i.P - c_i$$

  Local scalability is then given by:

  $$\forall i \in \{1..n\}; \Delta l_i = 0$$

Note that local scalability implies global scalability as:

$$\forall i \in \{1..n\}; \Delta l_i = 0 \Rightarrow \Delta L = \Delta \sum_{i=1}^{n} l_i = \sum_{i=1}^{n} \Delta l_i = 0$$

This shows that, ideally, the forwarding rates (r_i) of each queue should adapt to its consumers' consumption rate (c_i). Note that we didn't discuss the alias queue's load as, if our system works properly, it shouldn't have any. As explained earlier, the alias queue automatically forwards all the messages it is received.

C. Flow Control Policy

Our load balancing policy is flow control based. It is a consumption-aware policy that aims at forwarding more messages to the queues with the highest consumption rates. Practically, a controller periodically retrieves the consumption rates on the different load-balanced queues, and computes the new forwarding rates as the ratio between a queue's consumption to the total consumption of our system during the last round. Since this is a reactive policy, a significant change in the consumption rates might result in the overload of some queues. Our policy tries to distribute the overload over all the queues by artificially subtracting the difference between the queue's load and the average load from the number of messages it has consumed: if the queue's load is greater than average, it is forwarded less messages than it can handle so as it can consume some of its pending messages, otherwise, it is forwarded more messages, which would increase its load, and we'd have eventually the same load on all of our queues. This reduces the latency of our system as it minimizes the maximum load per queue, thus reducing the amount of messages that should be consumed before the last pending message can be consumed. If we take up the parameters defined earlier, the forwarding rates based on the values monitored on round k can be expressed as follows, $\forall i \in \{1..n\}$:

$$r_i(k+1) = \frac{1}{C} max(c_i(k) - (l - l_{avg}), 0)$$

where:

$$C = \sum_{i=1}^{n} c_i(k) \; ; \quad l_{avg} = \frac{1}{n} \sum_{i=1}^{n} l_i$$

This load balancing policy has two key assets: (i) it adapts to the changing consumption rates of the different load-balanced queues, and (ii) it distributes the overload over all the queues. Provided that our static system can handle the production load (global scalability), our flow control based policy guarantees, eventually, the local stability of each of the load-balanced queues. However, if the global load of our system increases beyond its maximum consumption capacity, all what our load balancing policy can do is cause the loads on the load-balanced queues to grow uniformly. Thus the need for a dynamic provisioning of resources to automatically cope with a global load change.

III. Elastic messaging

In this section, we present the different elements of our elastic messaging solution. We first describe when does our elasticity algorithm provision new queues (scaling out) and remove unnecessary ones (scaling in), then we detail our provisioning approach.

A. Scaling decision

In order to guarantee the scalability of our messaging system with regard to a change in the global load, we have implemented an elasticity controller that periodically: (i) monitors the different loads on the different queues of our system, (ii) potentially adds or removes queues based on the monitored values. Note that we base our decision solely on the queues' loads, since the consumption capacity on each queue may vary, particularly in the dynamic context of a cloud infrastructure.

1) Scaling up: This is achieved when we monitor that the average load of the system's queues is beyond an acceptable limit $maxLoad$. This guarantees that our system's scalability is beyond the scope of the flow control mechanism as the latter can only bring the queues' loads to this average load, whereas what is needed is to reduce the average load itself, which can only be done by provisioning more resources.

2) Scaling down: This should be triggered when we see that the system is using too many resources than it actually needs. We can *suspect* our system to be underloaded when all its queues are underloaded, i.e., almost empty, practically when the average load is below a threshold $minLoad$). This means that we have enough resources for the messages to be consumed as soon as they are produced, possibly *just enough* resources, in which case we shouldn't proceed with the removal of any of the queues. Thus, the scaling down decision can not be made at once.

To make sure that our system is effectively underloaded, when the elasticity controller suspects the system's overload, it elects a queue to be removed, and starts decreasing the amount of messages it is forwarded gradually. If, doing so, the average load goes above the specified limit, the scaling down plan is canceled and the elected queue receives messages normally, as specified by our flow control policy. Otherwise, if the elected queue is no longer forwarded any messages without the average load exceeding $minLoad$, then we can safely assume that this queue is no longer needed and it is effectively removed from our system.

Figure 2 outlines our elasticity algorithm.

Now that we have presented when scaling should be done, the next section details how it is actually achieved.

B. Provisioning

One a scaling decision is made, fast execution is of great importance to the proper working of our solution. Or, provisioning a virtual machine instance (VM) is relatively slow, for instance, it takes about a minute to provision a

```
while(TRUE) {
    sleep(period);
    monitorQueues();

    /* Scaling down */
    if (avgLoad > minLoad) {
        // Cancel scaling down plan
        toRemove = NULL;
    }

    if (avgLoad < minLoad && !toRemove) {
        // Start a new scaling down plan
        toRemove =
            queues.electQueueToRemove();
    }

    if (toRemove) {
        // Continue scaling down plan
        toRemove.reduceRate()
        if (toRemove.rate == 0) {
            queues.remove(toRemove);
            toRemove = NULL;
        }
    }

    /* Scaling down */
    if (avgLoad > maxLoad) {
        queues.addNewQueue();
    }

    /* General case */
    queues.applyFlowControlPolicy();
}
```

Figure 2: Elasticity algorithm outlines

small Ubuntu instance on Amazon EC2 [4]. To deal with our solution relies on co-provisioning queues on a same instance and pre-provisioning a pool of VMs.

1) Co-Provisioning: Since we are using a cloud computing infrastructure, where the resource unit is a virtual machine instance, an intuitive approach would be to add each queue on a separate VM instance. However, Joram's evaluation shows that due to the internal functioning of Joram, and depending on the size of the VMs, two or more queues can coexist on the same VM instance and still have comparable performance as with a configuration where each runs on a separate VM instance. Figure 3, shows the maximum throughput that can be achieved on a small EC2 VM on queues in different setups with regard to persistency of the messages, connection type and co-locality, with a message size of 100B. We can see that in a persistent setup, which is the most reliable, we can fairly co-provision up to

	Persistent		Transient	
Setup	localCF	tcpCF	localCF	tcpCF
1 queue	25 309	13 862	41 726	19 669
2 queues	25 052	13 456	33 971	16 828
4 queues	22 318	13 338	25 741	28 450

Figure 3: MQPerf

2 queues without significant performance decrease.

Thus, the resource unit is no longer the VM instance, but the available *slots* that we can provision queues on. Co-provisioning allows us to diversify our provisioning policies: if our main concern is performance, we might want to have each queue on a new VM instance, and provided we are using a private cloud, we might even want to create this VM instance on the least loaded physical machine. Other policies might have energy efficiency as the main concern. This is the case for the basic policy that we have implemented.

However, co-provisioning only reduces the impact of VM provisioning lag, for once all the slots on an instance are filled we still have to provision a new VM. Pre-provisioning deals with this issue once and for all.

2) Pre-provisioning: In order to optimize our solution even more, we have looked into reducing the time needed to add new queues, particularly when it involves provisioning a new virtual machine instance. The solution we propose is pre-provisioning a certain number of unneeded VM instances, which will be maintained as long as our cloud messaging system runs. This means that when a new node is needed, we use a pre-existing node, which renders our system more reactive, the used VM is then asynchronously replaced, which means that the creation of the new VM instance will not affect the latency of our system, thus improving its performance.

In order to evaluate the number of necessary pre-provisioned VMs, we need the Service Level Agreement to specify not only the maximum tolerated latency, which defines our $maxLoad$, but also the maximum supported increase of the production rate during a unit of time. Considering the following parameters:

- $SLA.delta$: The maximum increase of the production's rate in 1s (msg/s^2).
- $VM.startup$: The average startup time of virtual machines (s).
- $VM.capacity$: The maximum consumption capacity of a virtual machine, provided all its slots are filled (msg/s).

The number of virtual machines to be pre-provisioned NPP is given by:

$$NPP = ceil(\frac{SLA.delta \times VM.startup}{VM.capacity})$$

The numerator expresses, in the worst case, the extra production load that might occur during the startup of a

virtual machine. This should be handled by our pool of pre-provisioned VMs, thus, it should be equal to $NPP \times VM.capacity$, hence the formula above.

Next, we present the implemented provisioning policy.

3) Provisioning policy: Our provisioning policy is energy-efficiency-driven and aims at having an automatically consolidated park of queues. Should we have control over the cloud infrastructure as well, this consolidation is achieved on both levels: (i) having our virtual machines on the minimum possible number of physical machines (PM) and (ii) having the provisioned queues on the minimum number of VMs.

This is achieved on scaling up, by always trying to provision the new queue on an available slot in an existing VM instance, and only create a new instance if all the available slots on the last created VM instance are filled; and when creating the new VM instance always try to use the current physical machine and only use another if the first cannot host the new instance. This automatically minimizes both the numbers of utilized PMs an VM instances. On scaling down, in order to maintain the automatic consolidation, we always remove the last added queue, if it was the last one on its VM instance, than we can destroy one pre-provisioned VM and if this VM was the last one on its corresponding PM, the latter can be put into an energy-saving mode.

The next section studies the performance of our elastic cloud messaging system and shows the specific improvement due to each optimization.

IV. Evaluation

In this section, we validate our elastic messaging system and discuss its performance. We will not only validate our implementation as a whole, but also highlight the performance gain provided by each optimization, i.e., co-provisioning and pre-provisioning. All the following experiments have been done on Amazon EC2, using *m1.small* instances.

A. Effect of co-provisioning

In these two first experiments, the system is subjected to a production rate that gradually goes up to $750msg/s$, a single worker is configured to consume at most $100msg/s$, our elasticity algorithm's $minLoad$ and $maxLoad$ are respectively $50msg/s$ and $200msg/s$.

Figure 4 shows the results of allowing at most one worker per VM, whereas Figure 5 shows the results with provisioning up to two workers on the same VM. In both cases, no VM has been pre-provisioned.

As expected, the latency of VM provisioning results in overload pikes that might require the provisioning of extra workers to be handled, even though our algorithm has a safety interval in which he awaits the scaling decision to take effect. We can see comparing both Figures 4 and 5 that the overload pikes have been halved, as half the times in the second experiment, a worker doesn't have to wait for the provisioning of a new VM but can directly be provisioned on an existing VM. It is as well worth mentioning that in the second case, we only use half the number of virtual machines, which is a significant improvement in terms of energy efficiency.

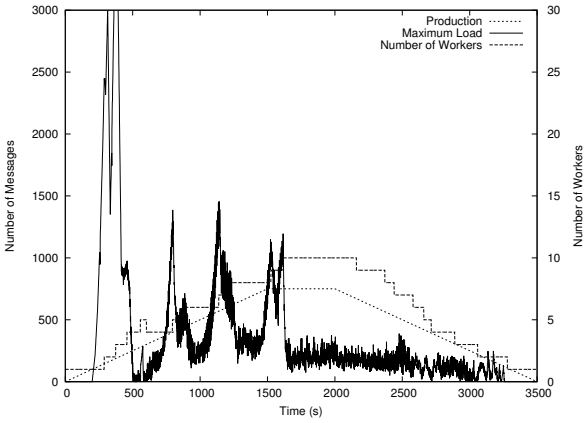

Figure 4: 1 worker per VM, no provisioning

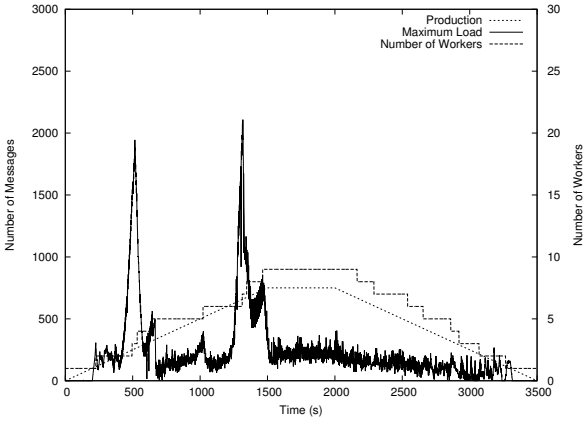

Figure 5: 2 workers per VM, no provisioning

B. Effect of pre-provisioning

Using the same parameters as above, we made a third experiment, where, in addition to provisioning two workers on the same VM, we pre-provision a VM. The results are depicted by Figure 6.

The pre-provisioned VM completely removed the impact of VM startup latency on our system, as we no longer need to wait for a VM to start: we always have an available VM to use and we replace it asynchronously.

C. Size of the pre-provisioning pool

In the previous experiment, one VM was enough for our system to work properly, as the production rate's acceleration was not very high. In the following two experiments, we

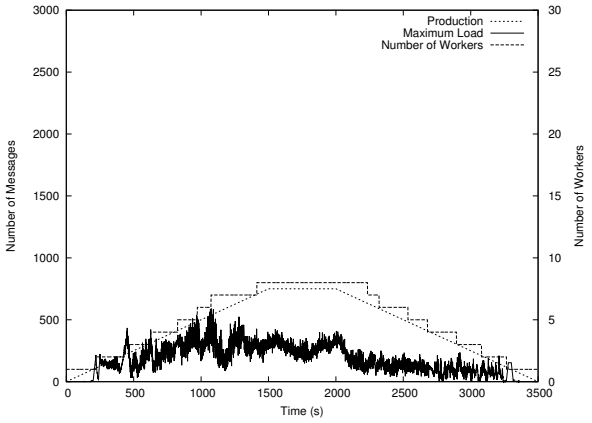

Figure 6: 2 workers per VM, 1 pre-provisioned VM

multiply this acceleration by eight. Figures 7 and 8 show the results with respectively one and two pre-provisioned VMs.

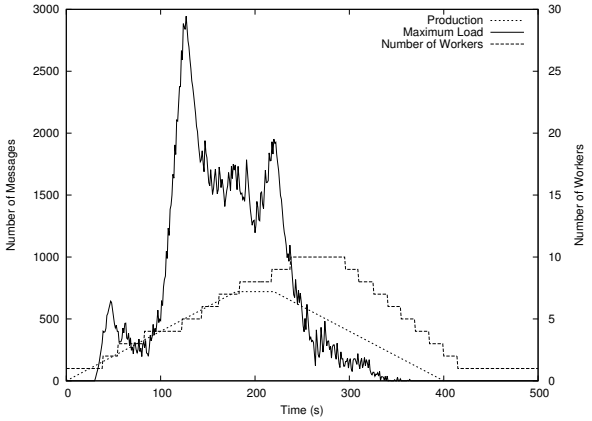

Figure 7: 1 worker per VM, no provisioning

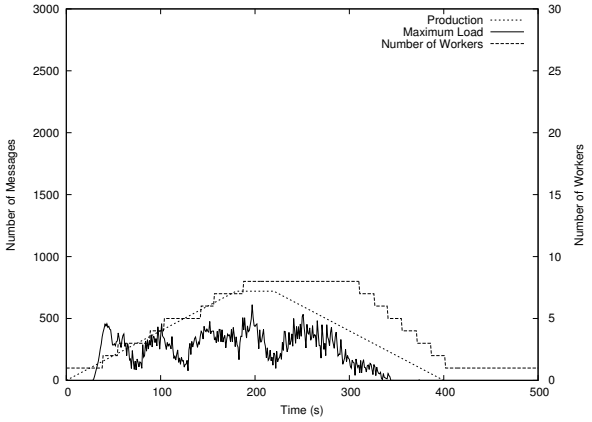

Figure 8: 2 workers per VM, no provisioning

It is clear from Figure 7 that, in this case, provisioning one VM isn't enough, the provisioning of the four first workers happens normally, since they use the first VM (the first two), and the pre-provisioned VM (the third and fourth), however, when a fifth worker is needed, it has to wait for the new pre-provisioned VM to start up, as it didn't have enough time to launch.

This can be expected as, if we take up the formula expressed in III-B2, the production rate increases by $100msg/s$ each $25s$, which corresponds to an $SLA.delta$ of $4msg/s^2$, and given that the mean startup time of an EC2 linux instance is $VM.startup = 96.9s$ [5], and that each VM contains 2 workers which consume $100msg/s$ each, which means that $VM.capacity = 200msg/s$, the minimum number of pre-provisioned VMs should be:

$$N = ceil(\frac{4 \times 96.9}{200}) = 2$$

Sure enough, pre-provisioning two VMs results in a proper functioning of our system as shown by Figure 8.

V. Related Work

This work is part of the general context of elasticity and dynamic resources provisioning of Internet services. Many works have previously addressed this matter either using heuristics to adapt the size of provisioned resources [6]–[11] or establishing mathematical models to characterize the systems to scale [12]–[16].

In the case of Oceano [6], the resources are not virtualized and the system only manages the physical resources allocated to the different applications, which makes the provisioning delay quite important and forces Oceano to have the resources for the most reactive parts allocated statically. OnCall [7], which uses virtualized resources, is more reactive to load spikes and allows us to have a 100% elastic resources pool. Cataclysm [10], [11] is another hosting platform that responds to overloads by adding extra resources, moreover, it uses a request classifier, to dynamically degrade the service during overloads. [8] and [9] specifically target the elasticity of a databases' cluster, the proposed solution involves keeping a set of idle nodes in order to improve the provisioning latency.

On the model-based approach, [13] for instance, formally describe the structure of multi-tier internet applications as a network of queues. This model is then used to achieve accurate capacity planning. Another example is SmartScale [16] which uses estimation models to coordinate vertical and horizontal scaling decisions in order to optimize the system's performance.

In the particular case of message queues' elasticity, besides the proprietary Amazon Simple Queue Service, there is EQS [17], which proposes an AMQP-based queue that can be replicated based on the connection loads of consumers and producers. Unlike our solution, EQS follows the one-to-many communication paradigm. Finally, [18] is a JMS compatible solution that introduces clustered queues: on

clients' admission, the client which connects to a generic connection to all the queues is forwarded to the least loaded queue, if all the queues are overloaded, a new queue is added to the cluster. This differs from our solution as (i) we do not depend on clients' connection to scale our system and (ii) the client-queue connections are not static and can be changed to achieve for better load balancing.

VI. Conclusion

We have presented an elastic message queuing system that adapts the number of queues and consumers to the load of messages. Our system has 3 main assets, (i) its flow control based load balancing makes sure that the provisioned resources are used to their maximum capacity; (ii) in the case of overload, our pre-provisioning and co-provisioning techniques achieve high reactivity while minimizing the cost and (iii) removal of unnecessary resources is done gradually in order to minimize the number of wrong decisions which would affect badly the performance of our system. Our work has been evaluated on a public cloud and particular care has been taken to show the benefit of each of our provisioning techniques. In the future, we intend to study the impact of different provisioning strategies on the behavior of our messaging system and generalize our approach to the one-to-many messaging paradigm.

Acknowledgement

We would like to thank FSN Datalyse and ANR Ctrl-Green for supporting this work.

References

[1] (2014, January) Java Message Service Concepts. [Online]. Available: http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

[2] (2014, January) AMQP home page. [Online]. Available: http://www.amqp.org/

[3] (2014, January) JORAM home page. [Online]. Available: http://joram.ow2.org/

[4] (2014, January) Amazon Elastic Cloud Compute home page. [Online]. Available: http://aws.amazon.com/ec2/

[5] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 423–430.

[6] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger, "Oceano-sla based management of a computing utility," in *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, 2001, pp. 855 –868.

[7] J. Norris, K. Coleman, A. Fox, and G. Candea, "Oncall: Defeating spikes with a free-market application cluster," in *Proceedings of the First International Conference on Autonomic Computing*, ser. ICAC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 198–205.

[8] G. Soundararajan and C. Amza, "Autonomic provisioning of backend databases in dynamic content web servers," in *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC*.

[9] G. Soundararajan, C. Amza, and A. Goel, "Database replication policies for dynamic content applications," in *In EuroSys06*. ACM, 2006, pp. 89–102.

[10] B. Urgaonkar and P. Shenoy, "Cataclysm: Handling extreme overloads in internet services," Department of Computer Science, University of Massachussetts, Tech. Rep., November 2004.

[11] ——, "Cataclysm: policing extreme overloads in internet applications," in *Proceedings of the 14th international conference on World Wide Web*, ser. WWW '05, New York, NY, USA, 2005, pp. 740–749.

[12] B. Urgaonkar and A. Chandra, "Dynamic provisioning of multi-tier internet applications," in *Proceedings of the Second International Conference on Automatic Computing*, ser. ICAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 217–228.

[13] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "Analytic modeling of multitier internet applications," *ACM Trans. Web*, vol. 1, no. 1, May 2007.

[14] Q. Zhang, L. Cherkasova, and E. Smirni, "A regression-based analytic model for dynamic resource provisioning of multi-tier applications," in *Proceedings of the Fourth International Conference on Autonomic Computing*, ser. ICAC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 27–.

[15] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 71–84.

[16] S. Dutta, S. Gera, A. Verma, and B. Viswanathan, "Smartscale: Automatic application scaling in enterprise clouds," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, june 2012, pp. 221 –228.

[17] N.-L. Tran, S. Skhiri, and E. Zimányi, "Eqs: An elastic and scalable message queue for the cloud," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 391–398.

[18] C. Taton, N. D. Palma, S. Bouchenak, and D. Hagimont, "Improving the performances of JMS-based applications," *Int. J. Autonomic Comput.*, vol. 1, no. 1, pp. 81–102, Apr. 2009.