Scalable Store and Forward Messaging

Ahmed El Rheddane, Noël De Palma, Alain Tchana
LIG/UJF, Grenoble, France
{*ahmed.el-rheddane, noel.de_palma, alain.tchana*}*@imag.fr*

***Abstract*—Since the emergence of the Internet, and particularly with the outburst of cloud computing, the production of reliable and scalable distributed applications is an important area of research. Various middleware technologies were designed for that purpose, among which we find Message-Oriented Middleware (MOM), which provides reliable asynchronous communication through message queueing techniques. MOMs have been standardized using the AMQP protocol, and in the Java world, with the JMS API.**

In this paper, we extend a store and forward mechanism to improve the scalability of an end-to-end reliable asynchronous messaging infrastructure while remaining compliant to the standard JMS API. We design a flow control based load balancing policy that, on the one hand, reduces the risk of consumer queues' failures while maintaining a near optimal throughput; and on the other hand, insures the scalability of our load balancing mechanism on the producer's side. We report the evaluation of our solution deployed on a cloud computing infrastructure and implemented within Joram, an open source implementation of the JMS API and the AMQP queuing protocol. This work is now part of the Joram distribution available on the OW2 consortium.

***Keywords*-JMS; message queues; scalability; load balancing; flow control**

I. INTRODUCTION

Today's applications often run on distributed resources. One of the most commonly used ways to simply yet reliably integrate the different components of a distributed software system is through a message-oriented middleware (MOM). MOMs use messages as the only structure to communicate, coordinate and synchronize, thus allowing the components to run asynchronously. MOMs offer two communication paradigms: *one-to-one*, producers send messages to a queue where they are stored till they are consumed by one and only one consumer; and *one-to-many* or *publish-subscribe*, a producer sends a message to a topic that broadcasts it to all the subscribed consumers. Java, with a concern of providing the community with a universal messaging interface has standardized the Java Message Service API (JMS) [1]. This, while making sure that all message-oriented applications would be easily integrated, gives the developers the choice of the implementation beneath depending on their specific needs with regard to reliability and overall performance.

The most intuitive MOM configuration consists in having one server, with the desired queue, generally on the consumer's side, thus rendering the distant communication channel between the producer and the queue vulnerable in the case of failures. Instead, a more reliable MOM ensures a store and forward mechanism. This mechanism requires a reliable communication model between producers and queues based on the following properties:

- *Asynchrony:* the asynchronous property decouples producers from queues. They do not need to be both ready for execution at the same time. This property enables a deferred access to queues and a loose coupling between producers and consumers.
- *Reliability:* once a message is sent, it is guaranteed to be delivered despite network failures or system crashes.

In this work, we consider the specific case of applications with symmetric consumers, i.e., all the consumers process the same tasks. We also position ourselves in the context of cloud computing, where the consumers might belong to different clouds and their performance varies depending on the load of the cloud, since the virtual machines might share the same physical resources thus affecting each other's performances. Taking this into consideration, we aim to improve the scalability of the store and forward mechanism with clustered queues: we propose a new load balancing policy based on flow control, which dynamically adapts the messages' load on each of the cluster's queues to its consumption rate; this will be highlighted by comparing our scalable store and forward solution to a static load balancing policy such as round-robin. Load balancing is moreover done on the producer's side so as to allow intercloud consumers' deployment. Last but not least, our solution includes a failover mechanism in order to enhance its reliability.

We implemented and evaluated our solution using Joram, for Java Open Reliable Asynchronous Messaging [2], deployed on a cloud computing infrastructure. Joram is a pure Java implementation of the JMS API. It also implements the Advanced Message Queuing Protocol (AMQP) [3].

The rest of this paper is organized as follows: Section II describes our store and forward mechanism and shows how we improve its scalability; Section III formally describes the scalability of queue messaging; in Section IV we detail the proposed load balancing strategy, which we later evaluate in Section V; then we present the related work in Section VI before finally concluding this work in Section VII.

II. STORE AND FORWARD WITH LOAD BALANCING

To provide a store and forward mechanism, a MOM must insure both the asynchrony between producers and the

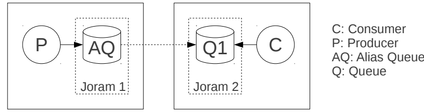

Figure 1. Alias queue's principle

queues deployed on consumers' side, and the communication reliability between them. For that purpose, a solution is to use a special destination called an *alias queue*. An alias queue is a special persistent queue that automatically forwards the messages it is sent to another, generally distant, persistent queue on the consumer's side (see Figure 1). It is set to write-only mode as the "real" destination is meant to be the queue to whom the messages are forwarded. The alias queue would thus be an intermediate destination on the producer's side where messages will be stored, and *visible* (i.e., can be monitored), till they successfully reach their final destination. This persistent pair of queues enforces the asynchronous property. To enforce reliability, the forwarding mechanism involves a distributed transaction between the alias queue and the related queue. This transaction insures, despite network or system failures, that a message is either stored on the persistent alias queue on the producer's side or on the persistent queue on the consumer's side.

The aim of this paper is to improve the scalability of this store and forward mechanism. We propose a new load balancing policy based on flow control described in Section IV. To implement this policy, we extended the alias queue mechanism to support load balancing. This extension is based on a well-known load balancing pattern similar to Web-based system (e.g., JK Apache Tomcat Connector [4]). Each producer is assigned to an alias queue that would distribute the messages to a set of distant clustered queues each corresponding to a set of local consumers (see Figure 2). We also integrated a failover mechanism that allows messages to be re-sent to another queue if their initial destination is unavailable. Note that this pattern is not exactly the same as the one used for Web systems since: (i) load balancing is achieved on the producers' side; and (ii) both the producers' and the consumers' sides can be controlled. Also, this is different from the one-to-many messaging paradigm provided by topics, as one message will be forwarded to one and only one of the cluster's queues. We will see in the following sections how this affects MOM's scalability and what the different strategies of distributing messages between our multiple destinations are.

III. Scalable Messaging

In this section, we discuss the different factors that affect the performance of a messaging system. First, we will start with the case of a standard queue then generalize our approach to clustered queues using an alias queue as a forwarding mechanism.

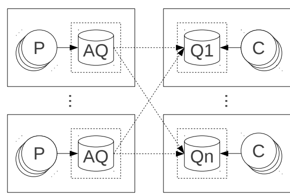

Figure 2. Scalable queueing with enhanced alias queues

A. Standard Queues

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*.
- $\Delta l < 0$: In this case, the consumption rate is higher than the potential reception rate and receivers are blocked waiting for new messages to come. The queue is still *unstable* and we say that it is *draining*. This means that the queue's ressources are underutilized.
- $\Delta l = 0$: 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.

B. Clustered Queues

In this case, our alias queue, to which the messages are sent, is wired to n queues, on which the messages are received. Let p be the production rate on the alias queue, 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:

1) Global Scalability: Let L be the total number of waiting messages in all the consumers' queues. We have:

$$L = \sum_{i=1}^{n} l_i \; and \; \Delta L = p - \sum_{i=1}^{n} c_i \tag{1}$$

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 fails to guarantee that on each consumer queue, the forwarded load is properly handled. This will be guaranteed by *local scalability*.

2) 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 queue. Thus, for each $i \in \{1..n\}$ we have:

$$\Delta l_i = r_i.p - c_i \tag{2}$$

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-271-4

Local scalability is then given by:

$$\forall i \in \{1..n\}; \Delta l_i = 0 \quad (3)$$

Note that local scalability implies global scalability as:

$$\forall i \in \{1..n\}; \Delta l_i = 0 \Rightarrow \Delta L = \Delta \sum_{i=1}^{n} l_i = \sum_{i=1}^{n} \Delta l_i = 0 \quad (4)$$

In the remaining of this paper, we will suppose that global scalability is verified and try to achieve local scalability by tuning our load balancing strategies.

IV. FLOW CONTROL POLICY

The main question that arises when forwarding messages to different destinations is how to achieve load balancing, i.e., how to distribute the received messages over the clustered destination queues. In this work, we propose a dynamic load balancing strategy based on flow control, i.e., the consumption rates of our consumers. As a reference load balancing strategy, we choose round-robin; we could also have chosen random, which is statistically equivalent, and would ultimately give the same results.

A. Round-Robin

The first implemented strategy is the simplest. It consists in forwarding messages uniformly over our destinations: we would forward the first message to the first queue, the second to the next one etc. Till we are out of destinations, in which case we go back to send to the first queue and so on.

If we take up the forwarding ratios introduced in the previous section, this strategy can be described as follows:

$$\forall i \in \{1..n\}; r_i = \frac{1}{n} \quad (5)$$

n being the number of queues wired to our alias queue.

While this strategy is straightforward to implement, and can even be effective if all the consumer queues have the same consumption rate; it can also result in local instability if our queues have different consumption rates. Besides, it is static, which makes it unable to follow the potential variation of our distributed messaging system. Thus, a more sophisticated adaptive strategy is needed.

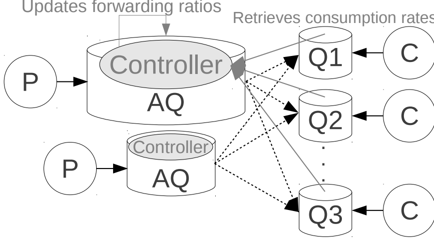

Figure 3. Load balancing controller

Algorithm 1 Flow control's algorithm

```
while TRUE do
  for each consumer queue c do
    rate[c] ← c.monitorConsumptionRate()
    load[c] ← c.monitorLoad()
  end for
  for each consumer queue c do
    weight[c] ← computeConsumerWeight(rate[])
    if load[c] > MAX_LOAD then
      weight[c] ← weight[c]*9/10
    end if
  end for
  p.updateWeights(weight[])
  sleep(period)
end while
```

B. Flow Control Principle

Flow control is a dynamic strategy that allows a consumption-aware message distribution. Its mechanism, described by Figure 3, relies on a controller integrated with our alias queues, which has a representation of their interconnections with the consumers' queues. The controller's integration guarantees our solution's scalability with regard to producers since each alias queue has its own load balancer instead of having one centralized load balancing controller. It is also easy to use for an end-user as load balancing is done transparently without any extra configuration.

Our controller periodically monitors the system, retrieving particularly the consumption rates of the consumer queues, i.e., the number of messages each of the cluster's queues has been asked to deliver over the last period. The decision process can then be formally described as follows: let us say that for the k-th period, we retrieved $c_i(k)$ as consumption rates for our queues. In order to make sure that the more a queue consumes messages, the more messages it will be sent, the expression of our $r_i(k+1)$ for the next period is:

$$\forall i \in \{1..n\}; r_i(k+1) = \frac{c_i(k)}{\sum_{i=1}^{n} c_i(k)} \quad (6)$$

As for the overload that might occur on a queue before its forwarding ratio is regulated, we propose to define a maximum load limit per queue, above which its forwarding ratio will be artificially decreased so as it can handle part of its pending messages.

Naturally, the controller executes its decision by replacing the old $r_i(k)$ with the newly computed $r_i(k+1)$. Technically, Algorithm 1 details the different steps that our controller goes through, where *computeConsumerWeight* implements $r_i(k+1)$'s expression. The weights used in our implementation are directly proportional to our forwarding ratios, they represent the number of messages that will be forwarded to the same queue before changing destinations.

The only question left to be answered is how to determine the period of our control loop. While having a shorter loop increases the reactivity of our system, it also induces a greater overhead as it involves exchanging monitoring messages more frequently. The solution we propose aims at maximizing the reactivity of our system while controlling its induced overhead. We do not fix the period itself, but we fix a tolerated overhead, i.e., the ratio of monitoring messages to the produced throughput: at each iteration, we determine the next period based on last period's throughput so as to stay within the tolerated overhead.

Our consumption-aware load balancing strategy takes into consideration the differences between our consumer queues in terms of consumption rates, which, a priori, vary with time, and should improve the performance of our system. The second part of the evaluation section verifies this assumption.

V. Evaluation

Now that our scalable distributed messaging system is properly geared, we have to check its efficiency. To do so, we started by evaluating the proper overhead of the alias queue, and we went on to compare the performances of our two load balancing strategies. Note that overhead always refers to the effect of using an alias queue on performance.

For our evaluation we used virtual machine instances of type m1.small as described by Amazon EC2 [5], i.e., 2GB memory and 1 VCPU, provisioned on a private cloud running racks with two 6 cores Intel(R) Xeon(R) CPU E5645 @ 2.40GHz, 32GB RAM, 1GBps isolated LAN and managed by OpenStack [6]. All our results are computed over campaigns of $1,000,000$ messages of 1kB each. Our solution has been implemented and tested using Joram.

A. Alias Queue's Overhead

In our first set of experiments, we want to evaluate the maximum capacity of Joram with and without using alias queue. Our metric here is the maximum throughput, i.e., maximum number of consumed messages per second. Figure 4 shows the results of these experiments, presented in pairs: either using an alias queue or not.

The first two experiments put both the consumer and producer on the same virtual machine, and use only one Joram server. We can see that the general throughput slightly decreases when using an alias queue as an intermediate message, along with its acknowledgement, is added. This overhead is however less than 4%, as intra-server communication is highly optimized in Joram.

The experiments 3 and 4, add a new Joram server, to evaluate the overhead when messages go through an intermediate server instead of directly reach their final destination; this corresponds to the reliable set-up discussed in the store and forward, subsection of section II, even though both servers are co-located on the same virtual machine. We can see that,

No	Configuration	msg/s
1	P → Q1 ← C	2291
2	P → AQ → Q1 ← C	2211
3	P → Q1 ← C	2052
4	P → AQ → Q1 ← C	2001
5	P → Q1 ← C	1944
6	P → AQ → Q1 ← C	1918
7	P → AQ → Q1 ← C; AQ → Q2 ← C	3678

Figure 4. Alias queue's overhead evaluation

in this case the overhead is even smaller (2.5%), as extra messages are needed for the forwarding even without the alias queue.

In the scenario depicted by the experiments 5 and 6, which is the most realistic since the communication is done between two different virtual machines, we can see that the alias queue's overhead drops to about 1%. Moreover, in this particular case, the virtual machines are co-located; should we consider the latency as well, the alias queue's overhead can fairly be neglected.

Now that we have established that alias queues' utilization has almost no overhead, the 7th experiment of Figure 4 shows how this mechanism can be used to enhance the scalability of our messaging system. The resulting throughput, which is roughly two times the previous one (experiment 6), shows that adding consumer queues to the alias queue linearly increases the system's performance.

In this particular case, the consumers were both identical, as they were both running on maximum speed, on similar virtual machine instances. Thus, the simple round-robin strategy was enough. In the next part, we will see how flow control is sometimes necessary for Joram to work properly.

B. Flow Control Evaluation

To evaluate our dynamic load balancing strategy, we regulate the sending and receiving rates of our clients and calculate the total time needed to receive the $1,000,000$ sent messages. We also monitor the system, particularly the queues load during the experiments. Based on the previously

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-271-4

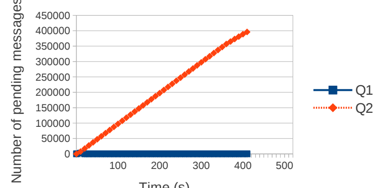

Figure 5. Consumer queues' load evolution

Figure 6. Consumer queues' load evolution in flow control mode

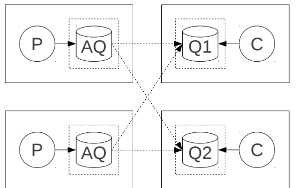

Figure 7. 2 producers, 2 consumers configuration

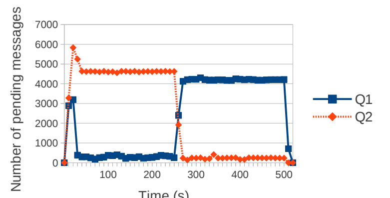

Figure 8. Consumer queues' loads with changing rates

computed maximum throughput (experiment 4, Figure 4), the production rate used for the following experiments is: **2000msgs/s**. The consumption rates of the queues are given as a percentage of the production rate and varies as follows:

1) One producer and two identical consumers: The configuration with 1 producer and 2 consumers is similar to the one set for the 7th experiment of Figure 4. Our experiments show that round-robin takes a total time of **500.0s**, whereas our flow control policy results in a total reception time of **500.5s**. This is the ideal case where both consumers receive messages at the same rate (50% of the produced load each), round-robin is here the perfect solution. However, we see that even when our flow control mechanism is activated, it gives us about the same performance. The overhead is due to expected side-effects in the computation of weights as we had to settle for a level of granularity.

2) One producer and two unbalanced consumers: In this case, our consumers have significantly different consumption rates (70% and 30%). Round-robin is not at all suited for such a configuration, it expectedly resulted in the time-out of the slowest consumer: it couldn't receive all the forwarded messages in a reasonable time. This is mainly due to the overload on the consumer's queue, as on each round, it keeps 20% of the forwarded messages, which later affects its ability to respond to the consumer's client requests. Figure 5 shows the evolution of the slow consumer's queue load.

We can see that the number of waiting messages on the slowest consumer's queue is growing linearly, and while this queue is flooded, the other is draining. This badly affects the overall performance of the system. Flow control, on the other hand, achieves a total reception time of **510s**, which is not very far from the ideal 500s. The delay is due to the fact that the flow control loop's initial period is 10s, which means that it takes 10s for the first flow control regulation to take place. Figure 6 shows that flow control regulated the forwarded messages to insure a balanced load on both consumers' queues.

3) Two producers and two variable consumers: Figure 7 shows the configuration set up for this experiment, which is meant to prove two things: first, that our mechanism can work with more than one producer (i.e., alias queue); more importantly it shows that our flow control effectively adapts to any consumption rates' variation as we start with consumers receiving with 70%-30% rates and invert them on $t = t_0 + 250s$ to 30%-70%. Figure 8 describes the queues' loads during this experiment.

As you can see in Figure 8, the queues' loads are stabilized throughout the experiment, which results in a total reception time of **506s**. This surely concludes the effectiveness of the flow control mechanism.

VI. RELATED WORK

While in our present work, we apply load balancing policies to message-oriented middleware, many previous works have detailed different load balancing strategies, particularly for web-based applications [7], [8], [9], [10]. These policies have been classified as content-blind or content-aware based on whether they take into account requests being forwarded. *Round-robin* and *weighted round-robin* are obviously content-blind. Other content-blind policies are *random*, which dispatches messages randomly between the worker servers; *least connection* and *least loaded*, which forward messages respectively to the server with the least number of connections and the one with the least load, with regard to the server's capacity and current utilization. Content-aware policies aim to achieve better efficiency by taking into account for instance the sessions established between the clients and servers and forward the packets belonging to the same session to the same servers, these are then called *sticky sessions* [11]. Another content-aware policy consists in taking into account the locality of the clients and forward their requests to the nearest servers. While these policies in general aim at optimizing the performance of the system, other studies [12], [13] focus on the energy efficiency of such policies. Our flow control policy is therefore content-blind, it also differs from the previous policies by its integrated store-and-forward mechanism.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-271-4

Load balancing has also been widely addressed in the context of high performance computing in grids or multiprocessor machines[14], [15], [16] where distributed load balancing, which involves exchanging loads between neighbor computing nodes is rather privileged.

Commercial message-oriented middlewares do also integrate load balancing, IBM's WebSphere MQ [17], for instance, provides a basic round-robin policy for its cluster queues that can be enhanced by statically specifying weights for queues in order to manage their priority. Another example is HornetQ [18], which also distributes the loads over its queue clusters on a round-robin basis, it excludes however the queues with no connected consumer. Finally, Oracle's BEA WebLogic [19] JMS implementation also offers load balancing with policies limited to round-robin and random.

In the specific case of Joram, a previous work [20] has addressed scalability differently: producers are statically affected each to a specific consumer queue; these consumer queues are interconnected (clustered queues) in a way that each draining queue will see if the the others have extra messages and "steal" them, likewise, once a queue's load reach a certain limit it distributes, if possible, the extra load over the other queues. Whereas this is a corrective policy that handles problems when they occur, which results in extra traffic on our system, as a message is first sent to a queue, then it is potentially forwarded as many times as necessary; our work is based on a predictive policy that tries to forward the messages to the "right" queues in the first place.

VII. Conclusion

Message-oriented middlewares have proven to be an effective way to integrate the components of a distributed software system, both guaranteeing asynchrony and end-to-end reliability thanks to their store and forward mechanisms. In this paper, we described and extended the store and forward mechanism of a MOM infrastructure in order to improve its scalability with regard to both the producers and the consumers, while maintaining the JMS API compatibility. Our extension includes the design of a flow control based load balancing policy to insure the local stability of the clustered queues. This has been done with the concern of providing a scalable distributed mechanism that would be totally transparent to the end-user. The evaluation of our solution, carried out on a cloud computing infrastructure, shows the effectiveness of our design compared to a basic load balancing policy. As a future work, we intend to enhance our solution to support the elasticity of message-oriented middleware using the flexibility offered by cloud computing infrastructures. We will thus go beyond the static dimensioning the queues and develop a dynamic provisioning mechanism that would scale automatically the clustered queues based on the total load of our system.

Ackowledgement

We'd like to thank ANR INFRA for supporting this work.

References

[1] "JMS Concepts," [retrieved: Mar., 2013]. Available: http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html

[2] "Joram home page," [retrieved: Mar., 2013]. Available: http://joram.ow2.org/

[3] "AMQP home page," [retrieved: Mar., 2013]. Available: http://www.amqp.org/

[4] "The Apache Tomcat Connector," [retrieved: Mar., 2013]. Available: http://tomcat.apache.org/connectors-doc/index.html

[5] "Amazon Elastic Compute Cloud home page," [retrieved: Mar., 2013]. Available: http://aws.amazon.com/ec2/

[6] "OpenStack home page," [retrieved: Mar., 2013]. Available: http://openstack.org/

[7] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A scalable and highly available web server," in Proceedings of the 41st IEEE Computer Conference (COMPCON), 1996, pp. 85–.

[8] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on web-server systems," in IEEE Internet Computing, vol. 3, May 1999, pp. 28–39.

[9] K. Gilly, C. Juiz, N. Thomas, and R. Puigjaner, "Adaptive admission control algorithm in a qos-aware web system," in Inf. Sci., vol. 199, Sep. 2012, pp. 58–77.

[10] B. Yagoubi and Y. Slimani, "Dynamic load balancing strategy for grid computing," in Transactions on Engineering, Computing and Technology, 2006.

[11] L. Cherkasova and P. Phaal, "Session-based admission control: A mechanism for peak load management of commercial web sites," in IEEE Trans. Comput., vol. 51, 2002.

[12] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2008, pp. 337–350.

[13] E. M. Elnozahy, M. Kistler, and R. Rajamony, "Energy-efficient server clusters," in Proceedings of the 2nd Workshop on Power-Aware Computing Systems, 2002, pp. 179–196.

[14] C. Xu and F. Lau, "Iterative dynamic load balancing in multicomputers," in Journal of Operational Research Society, vol. 45, 1994, pp. 786–796.

[15] R. Diekmann, B. Monien, and R. Preis, "Load balancing strategies for distributed memory machines," in Multi-Scale Phenomena and Their Simulation, 1997, pp. 255–266.

[16] Y. Li and Z. Lan, "A survey of load balancing in grid computing," in Proceedings of the First international conference on Computational and Information Science (CIS), 2004, pp. 280–285.

[17] "WebSphere MQ V6 Fundamentals," [retrieved: Mar., 2013]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247128.pdf

[18] "HornetQ home page." [retrieved: Mar., 2013]. Available: http://www.jboss.org/hornetq/

[19] "Introduction to WebLogic JMS," [retrieved: Mar., 2013]. Available: http://docs.oracle.com/cd/E13222_01/wls/docs81/jms/intro.html

[20] C. Taton, N. De Palma, J. Philippe, and S. Bouchenak, "Self-optimization of clustered message-oriented middleware," in Proceedings of the 4th International Conference on Autonomic Computing (ICAC), 2007.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-271-4