2011 Third IEEE International Conference on Coud Computing Technology and Science

EQS: an Elastic and Scalable Message Queue for the Cloud

Nam-Luc Tran
Euranova R&D
Belgium
Email: namluc.tran@euranova.eu

Sabri Skhiri
Euranova R&D
Belgium
Email: sabri.skhiri@euranova.eu

Esteban Zimányi
Université Libre de Bruxelles
Belgium
Email: ezimanyi@ulb.ac.be

***Abstract*—With the emergence of cloud computing, on-demand resources usage is made possible. This allows applications to elastically scale out according to the load. One design pattern that suits this paradigm is the event-driven architecture (EDA) in which messages are sent asynchronously between distributed application instances using message queues. However, existing message queues are only able to scale for a certain number of clients and are not able to scale out elastically. We present the Elastic Queue Service (EQS), an elastic message queue architecture and a scaling algorithm which can be adapted to any message queue in order to make it scale elastically. EQS architecture is layered onto multiple distributed components and its management components can be integrated with the cloud infrastructure management. We have implemented a prototype of EQS and deployed it on a cloud infrastructure. A series of load testings have validated our elastic scaling algorithm and show that EQS is able to scale out in order to adapt to an applied load. We then discuss about the elastic scaling of the management layers of EQS and their possible integration with the cloud infrastructure management.**

I. Introduction

Nowadays, many applications are developed on a cloud environment. However, in order to make them scalable and fully elastic, we need to adopt new design patterns. The event-driven architecture (EDA) is probably the most important architectural pattern used in the cloud for making applications scale. Indeed, this pattern enables asynchronous communication between distributed application instances and fits perfectly to the dynamic and elastic nature of the cloud. Going further, many components of the Infrastructure as a Service (IaaS) management require adapted messaging middlewares in order to efficiently transport the tremendous number of control messages they receive. These components comprise for instance the decision layer used for auto-scaling decisions, the monitoring system and other components of the cloud infrastructure. These messaging middlewares would have to be highly scalable and should scale elastically.

Today, the most efficient message buses only comply with the first condition: they are scalable. Worse, they are designed to be scalable only for a few number of producers and listeners, which is not the case in a cloud environment where one could face several hundred thousand listeners and producers. Finally, they are not elastic and cannot scale dynamically according the load.

In this paper we propose an algorithm that can be adapted to any message queue in order to render it fully elastic. By elastic we mean exploiting the on-demand computing resources of a cloud by dynamically mobilizing resources and dismissing them when they are not needed any more. In the case of an elastic message queue, this means that according to the message throughput, the message queue can be automatically and dynamically distributed on newly created instances (scaling out) when it is required, or in the opposite case, can be redistributed to a subset of queue instance nodes (scaling in) when the load is less important.

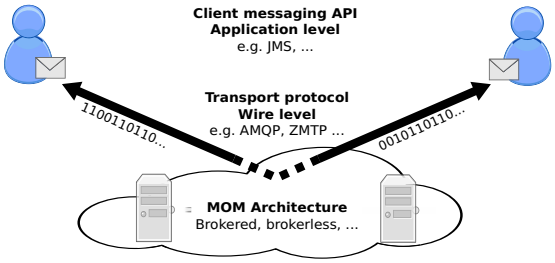

Fig. 1. The three layers in the stack of Messaging Oriented Middleware (MOM) products. The idea behind EQS is to bring the architecture of the MOM on the cloud in order to make use of the dynamic resource allocation and scale elastically.

The general idea of the algorithm is to dynamically adapt the load by isolating the most important producers and listeners and redirect them to a newly created instance of the same queue. The objective is to balance the message throughput on the different existing instances. We have developed a prototype based on ZeroMQ[1] and have evaluated its performance on Amazon EC2[2].

II. Background and Related Work

When referring to current existing Message Oriented Middlewares (MOMs), there are actually three distinct layers to take into account. Figure 1 summarizes these three layers.

The first layer is the API at the application level which defines a standardized set of calls between clients. One of the most well-known API used to send messages between client applications is the Java Message Service (JMS)[3].

[1]ZeroMQ: http://www.zeromq.org
[2]Amazon EC2: http://aws.amazon.com/ec2/
[3]JMS-J2EE: http://www.oracle.com/technetwork/java/jms/

978-0-7695-4622-3/11 $26.00 © 2011 IEEE
DOI 10.1109/CloudCom.2011.59

Beneath the API application layer is the transport protocol which is used to carry the messages at the wire level. Two different MOM products which are interoperable should be compliant at the API level but also use the same wire-level protocol, while abstracting the complexity of the architecture [1]. An attempt at standardizing the wire-level protocol is the Advanced Messaging Queuing Protocol (AMQP)[4].

Underneath the wire-level protocol lies the architecture of the MOM. In its most simple form, the architecture may consist in direct point-to-point communication between applications, without requiring an intermediate element such as a broker. In this category, we can cite for example the ZeroMQ library, which offers enhanced socket communication allowing N-N relationships between sockets and basic message filtering.

While in such architecture there are less elements to manage and no apparent single point of failure, the lack of an intermediate element between clients implies that client applications need to exactly know the address and nature of the recipients they are sending messages to, and that both clients should be alive at the moment of sending (no time decoupling). Furthermore, the complexity of the message delivery is pushed to the clients which become each a performance bottleneck. Such architecture offers no messaging element to scale out in order to adapt the performance of the messaging to the actual load and can thus not benefit from the dynamic resource allocation made possible by cloud computing infrastructures.

In most of the MOM architectures there is a central element, the broker, which is the common end-point for the clients of the MOM. The broker asynchronously delivers messages from a producer to a consumer (synchronization decoupling). The producer does not need to know the exact location or nature of the listener, it just delivers its message to the broker which then in turn routes it to the corresponding consumers (space decoupling). The broker thus provides space, time and synchronization decoupling. As this interaction paradigm between message producers and listeners is adapted to the deployment of scalable and loosely coupled systems, the architecture of the underlying MOMs should also be designed with scalability in mind [2].

The AMQP protocol specifies a broker-based architecture made of exchanges, bindings and queues. Exchanges are the endpoints of the producers and host the queues. They receive each message sent and route them to the corresponding queue which are endpoints for the consumers. The routing of the messages is based on the binding and the routing key contained in the header of the message. Figure 2 describes a typical brokered AMQP setup.

Because of its central nature, the broker is often seen as a single point of failure and bottleneck in many MOM architectures. As many works have already pointed out [3], [4], the broker-centric design of AMQP represents a bottleneck to both performance and scaling. Some implementations of AMQP brokers, such as RabbitMQ[5] for example, allow

[4]The 1.0 draft of the protocol has just been released: http://www.amqp.org

[5]VMWare RabbitMQ: http://www.rabbitmq.com/server.html

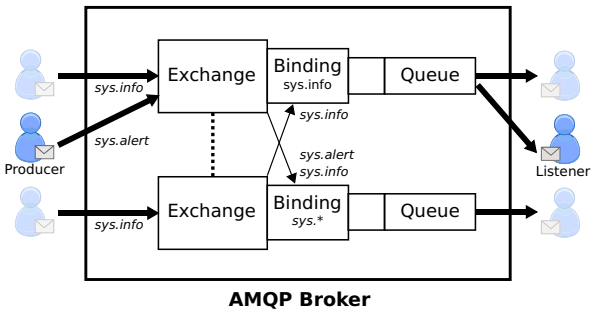

Fig. 2. Description of a typical AMQP broker setup. The binding actually links a queue to an exchange and filters the messages that will be delivered to the queue.

exchanges to work in a clustered mode. In this setup, a broker consists of a cluster of several exchanges working together. The state is shared among the nodes thanks to a shared database. RabbitMQ for example uses the Erlang in-memory shared database Mnesia [5]. However, queues are not load balanced on the exchanges and the messages remain stored on the exchange where the queue has been declared. When accessing a queue from another exchange, the request is simply routed to the exchange where the queue is assigned. This architecture involving a shared state does not allow a cluster of an arbitrarily large number of nodes. RabbitMQ clusters usually consists of a few nodes and their overall performance even decreases as the size of the cluster increases.

This design thus does not allow to increase the global performance of the queue by scaling out new instances of the MOM. This limitation prevents AMQP-based messaging products to be deployed on a cloud computing infrastructure in order to make use of the dynamic resource allocation so that the queue performance is dynamically adapted to the actual load.

To our knowledge the only successful implementation of a broker-based MOM using a cloud computing infrastructure is the Amazon Simple Queue Service (SQS)[6]. Many reports have however stated that SQS does not scale well with the number of concurrent users [6]. Message throughput is also low and latencies are high when compared to traditional MOM products.

III. The Elastic Queuing Service

In this section we present the architecture of the Elastic Queue Service (EQS), a message bus designed to enable elastic scalability, performance and high-availability. Its design is targeted for a deployment on a cloud computing infrastructure using commodity hardware. EQS contributes on the architecture level of the MOM stack (Figure 1).

A. Goals

The EQS architecture aims at answering the following goals and requirements:

[6]Amazon SQS: http://aws.amazon.com/sqs/

1) Elastic scalability: EQS copes with load increases by scaling out instances. It makes use of dynamic resource allocation provided by the cloud infrastructure to scale out elastically. The performance is linear with the number of instances.

2) High throughput: by spawning out additional instances, EQS is able to adapt to any applied load and reach a high amount of messages processed per time unit. The high performance of EQS is not focused on its individual components nor on single-instance configurations but comes from its ability to elastically scale out.

3) Integration on a cloud: with all these considerations in mind, EQS was designed from the beginning as a product running on a cloud infrastructure. Many of its features should integrate with functionalities provided by cloud infrastructure providers (IaaS) and cloud platforms management. When available, these functionalities and requirements should be compliant with existing and upcoming standards for cloud computing.

B. EQS Components

The architecture of EQS is distributed among different components. In the following we provide a description of them.

1) Producer and listener: These are the basic components of EQS. A *producer* is a client which connects to a specific queue in order to deliver a message. A *listener* is a client which connects to a specific queue in order to retrieve a message. There is no major restriction on the format or the nature of the message as EQS only transmits bytes.

2) Exchange: The exchange is the low-level component connecting producers and listeners. It receives a message on its frontend and transmits it to the listeners connected to its backend in a N-N relationship. The exchange is totally stateless. It can perform specific delivery to the listeners connected to its backend based on a routing key contained in the message. Thanks to their stateless nature, exchanges are the EQS components suitable for scaling-out in order to adapt to the load.

3) Queue: A queue is the logical concept of the entity used by producer and listener clients to transmit messages concerning a particular subject. An EQS queue is comprised of one or more exchanges, each one transmitting messages concerning the same subject. Queues hide the complexity behind exchanges and the other components of EQS to the producers and listeners. A producer or listener which wants to connect to a specific queue should only know the name or the unique url of the queue. Figure 3 describes the topology of an EQS queue.

Each queue is defined by a set of limit Key Performance Indicators (KPIs) issued from a Service Level Agreement (SLA) definition set. These limit KPIs define the quality of service level that is expected and are specified by the customer owning the queue. They describe, for example, the minimum guaranteed throughput (number of messages delivered per time unit) of the queue, which is defined for a fixed number of producers and listeners connected to the queue and for a fixed message payload.

4) Queue management: The queue management is the organizational layer on top of the queue. It keeps a permanent record of all the existing queues including their name, their URLs, the associated SLA, the list of exchanges composing the queue and their physical location. These information constitute the *metadata* associated to a queue.

5) Monitoring: The monitoring component aggregates information over the health of the EQS service. It monitors the KPIs at the EQS level and at the infrastructure level.

- At the EQS level, metrics specific to EQS are monitored in order to determine the status of EQS exchanges. Those metrics consist of the total number of messages processed by the exchanges, the number of connected producer and listener clients and the available bandwidth for each exchange.
- At the infrastructure level, system metrics such as component availability and resource usage are monitored.

At the same time, the monitoring component is able to induce failover actions related to failure handling:

- producers and listeners are relocated to alternative free exchanges, or new exchanges are created, in case of failure.
- components of EQS are restarted when outage occurs.

In the case where EQS is integrated with the cloud management such as mentioned in the introduction, the monitoring component, given its ubiquitous nature, is best placed on the infrastructure and platform manager of the cloud provider. In this context, monitoring of EQS metrics should be defined in a standard way in order to be integrated with existing cloud platform management standards. Some cloud platforms management projects, such as CLAUDIA [7] for example, already propose that functionality. Figure 3 describes the integration of the EQS management components to the management of the cloud infrastructure.

6) Rules and scaling management: This component manages the rules and the triggered actions regarding the defined SLA. The rules and scaling management decide which actions to take based on measures aggregated by the monitoring component of EQS and also on external SLA and billing rules. Triggered actions include:

- Scaling out and in of exchanges when load increases or decreases, or when there is no more budget to keep instances alive on the underlying cloud infrastructure.
- Relocation of producers on other free exchanges, when limit KPIs are reached.
- Relocation of producers and listeners on existing exchanges when scaling in.

The scaling out of instances is based on the elastic algorithm described in Section III-D

C. Interactions between EQS components

All the components described in the previous section work together in order to compose the EQS. In this section we

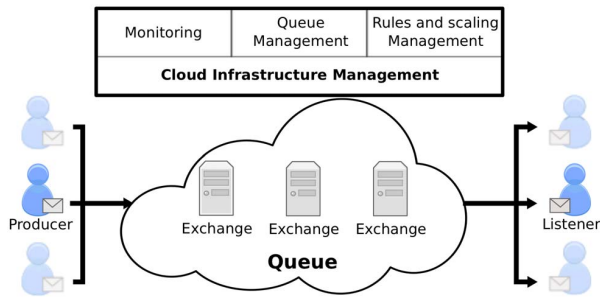

Fig. 3. EQS components deployed on a cloud computing infrastructure. An EQS queue is comprised of several exchanges, each transiting a portion of total traffic on the queue from producers to listeners.

will present these interactions along particular use cases and workloads which describe the behavior of the EQS while in production.

1) Producer/listener queue connection: A producer or listener which wants to connect to a queue in order to send or retrieve messages first contacts the queue management component and addresses the name of the queue they want to connect to. The queue name is specified by the underlying application on both producer and subscriber sides. As the queue management component holds the information about the exchanges that compose a queue and is synced with the monitoring, it will return the address of a free exchange of the queue to connect to. If the queue does not exist yet, the queue management component creates it:

- new exchanges are started.
- the metadata is updated with the addresses, the location of the exchanges and the required SLA.
- the new exchanges are registered with the monitoring component.

2) Queue scaling out: When the monitoring detects that the limit KPIs for a queue are reached, a scaling out action may be needed in order for the queue to cope with the actual load. The limit KPIs that trigger this action are defined in the SLA at the creation of the queue. They may comprise situations such as, for example:

- the queue has reached a critical throughput in its actual exchange configuration.
- average latency of message delivery is higher than a threshold.
- the average number of producers and listeners connected to an exchange has reached a critical number.

When any of these situation is detected by the monitoring, the rules and scaling component triggers a scaling out action based on the monitoring values, the topology of the existing exchanges and the SLA. The scaling out of exchanges is performed following the elastic scaling algorithm presented in Section III-D. Producers are then relocated to the additional exchange created which relieves part of the load from the existing exchanges. This created exchange is then registered in the queue management component and for monitoring.

3) Queue scaling in: This is the opposite case of the scaling out situation: an exchange is tagged for imminent shutdown. This can be caused by situations such as:

- the exchange is underused and the load it actually handles can be distributed among the other exchanges. In a cloud pay-per-use model, this means that the cost of maintaining this instance alive is higher than its benefits and thus it can be shut down.
- the billing rules state that there is no more credit for that queue. Exchanges are shut down and the queue is brought to a minimal service level, as agreed on in the SLA.

These situations are detected by the rules component and actions are inferred with respect to the scaling rules. When an exchange is shut down, its connected producers are relocated on the remaining exchanges, which addresses are provided by the queue management and in a load balanced manner.

4) Failover on exchange down: An exchange which encounters an outage is detected by the monitoring. The queue management and the scaling management then relocate the producers and listeners which were connected to that exchange on the remaining exchanges through a broadcasted message on a dedicated channel.

D. The EQS exchange elastic scaling algorithm

The scaling out of exchange instances is based on the elastic scaling algorithm:

```
Queue q;                                                          1
if q.load > kpi_queue                                             2
  Exchange new_exchange createNewExchange();                      3
  List<Exchange> L = q.getOverloadedExchanges();                  4
  while(new_exchange.isNotOverloaded)                             5
    for Exchange e in L                                           6
        AND e.getNbOfConnectedProducers > 1                       7
      Producer P = e.getMostProductiveProducerID();               8
      P.relocate(new_exchange);                                   9
  q.getListeners.updateExchanges(new_exchange);                   10
```

Basically the algorithm creates a new exchange if a queue has reached its limit KPIs (line 2, we do not represent here the process of checking if the queue is eligible for a scaling out, for example, if there is enough credit or resource) and relocates (line 9) for each overloaded exchange (line 4) its most productive producer (line 8), that is the one which has sent the biggest number of messages over the previous time period, on the newly created exchange. It then updates all the listener clients of the queue to listen to that exchange (10). The limit situation occurs when an exchange has reached its limit KPIs with a single producer connected (line 7), in which case it is not possible to relocate that producer elsewhere. If there is no available exchange with higher KPIs, advanced actions such as notifying the producer or the queue owner should be considered (not shown here).

IV. Implementation

In this section we describe how we have actually implemented EQS and tested it on a cloud infrastructure.

We have decided to start from an existing messaging product and develop the components of EQS on top of it. The reason behind this choice is that we want to focus EQS development on the architecture and the elastic scalability, and delegate the common messaging functions such as message publish and subscribe, topic routing, in-memory message caching and disk swapping, to an existing messaging library. In order to be able to develop our functions on top of it, the messaging library would have to remain quite at low-level. We have settled to develop our prototype of EQS on top of the ZeroMQ messaging library.

ZeroMQ is a messaging library written in C which handles communications between sockets. It provides messaging features between sockets with on a brokerless architecture where queues are pushed on the edges at each client. The following delivery models are supported:

- publish-subscribe with basic topic key matching.
- request-reply with advanced distribution patterns.

In the scope of our implementation we will focus on the publish-subscribe messaging model but our architecture can be easily adapted to a request-reply model. We rely on TCP/IP transport.

A. Producer and listener implementation

We have used the off-the-shelf ZeroMQ implementation of the publish-subscribe client. A publisher client opens a ZeroMQ socket, connects it to an endpoint on the network and starts sending messages. A listener client opens a socket, connects it to an endpoint, declare a topic key, and starts consuming messages related to that topic. A listener socket can connect to multiple publisher sockets at the same time. In the publish/subscribe configuration the ZeroMQ library handles the listener subscriptions at the client side, there is no exchange between a producer and listener which maintains the state of the subscriptions.

B. Exchange implementation

The exchange is implemented as a proxy comprised of a listener socket on its frontend with a topic key matching all possible keys and a publisher socket on its backend. It receives messages from the frontend and resends the message immediately to the listeners connected to the backend socket.

C. Monitoring, rules and scaling management components

As with the current state of standards in cloud platforms and infrastructure it is not yet possible to seamlessly merge custom application monitoring and rules triggering with cloud management infrastructures, we have decoupled these components at each level.

At the infrastructure level, we use the cloud provider metrics for monitoring the system health. This comprises the following metrics: CPU usage, memory and bandwidth available. We also use the alerts and rules triggering methods featured by the cloud platform management based on these metrics. Actions triggered by these metrics are: start-up of a new instance when available memory is below a threshold and shut-down of an instance when it is underused.

At the EQS level, we have implemented a specific type of instance which monitors the health of the exchanges. This monitoring receives heartbeats and health values of all the exchanges under its supervision. The monitoring instance is monitored by the cloud infrastructure monitoring. It also triggers actions with regard to the elastic scaling algorithm (see Section III-D).

We make use of the service abstraction provided by existing cloud platform management tools in order to manage the metadata and KPIs of the running exchanges and for the infrastructure monitoring. Existing platforms which provides these functionalities comprise Amazon CloudWatch[7], Scalr[8] and Rightscale[9]. The platform management tool that we used in this case is Scalr. Figure 4 summarizes our implementation of EQS and shows the integration of each component with the cloud platform management.

D. Queue management

We have developed a queue management service instance which stores the metadata of the queues: existing queues names and the addresses of the exchanges they are made of. It also stores a basic set of KPI requirements comprised of available bandwidth and number of processed messages on the exchange. The producer and listener clients initiate the connection to an exchange by first contacting the queue management using request-reply communication. In turn they receive the address of a free exchange to connect to. The queue management is synced with the exchange monitoring which makes it always up to date concerning the available exchanges. In practice, we have merged the queue management instance on the same instance as the EQS monitoring instance.

E. Elastic scaling algorithm implementation

The elastic scaling of EQS is decoupled, as is the monitoring, at the cloud infrastructure and EQS levels. First, exchanges system limit KPIs are detected by the infrastructure management which creates a new instance each time the available memory or bandwidth go below a threshold. When the new instance is created, the EQS monitoring instance is notified on this newly created exchange. It then triggers the scaling algorithm and relocates the most productive producers from the existing exchanges which are above the limit KPIs to this new exchange.

V. Evaluation

We have validated the elastic scaling ability of our EQS implementation through a test involving one queue. We start with an initial queue made of 1 exchange. Producers then start to connect and gradually increase the load of the queue. The architecture will be able to automatically scale-out by adding additional exchanges in order to cope with the load and with

[7]Amazon CloudWatch: http://aws.amazon.com/cloudwatch/
[8]Scalr: https://scalr.net/
[9]Rightscale: http://www.rightscale.com/

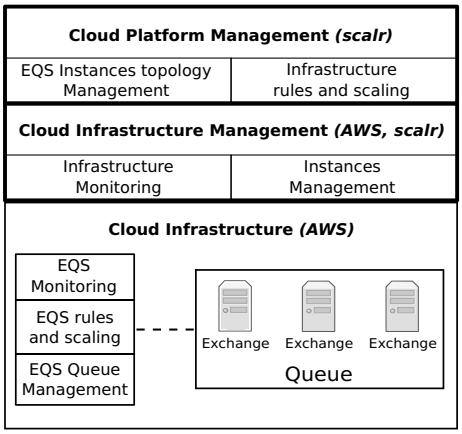

Fig. 4. Our implementation of EQS on top of existing cloud management platforms. Monitoring and scaling management have been decoupled between infrastructure and EQS levels. A dedicated instance type monitors EQS metrics.

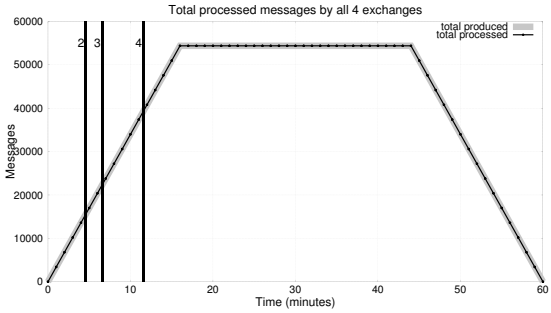

Fig. 5. EQS test workload results: 16 producers connect at intervals of 60 seconds, each sending messages of 5 kB at a rate of 3400 messages/min during 45 minutes. This plot represents the average number of messages received from the exchanges and passed to the listeners over the 5 runs. On overlay is the total number of messages produced by the 16 producers. Vertical lines represent the mean minute of the test when a new exchange is started by the monitoring following a scaling out event. Exchange 1 is started at minute zero.

regard to the rules and KPIs. We have used the latest ZeroMQ version at the time of writing (2.1.7) along with the Java binding (*jzmq*). Amazon instances of type *small* have been used for the exchanges, the monitor, the producer and listener clients. The hosted version of Scalr has been used to manage the metadata and to trigger the rules related to instances system metrics.

A. EQS test workload

Sixteen producers connect at intervals of 60 seconds to the queue and send messages at a rate of 3,400 messages per minute with a payload of 5 kilobytes per message. They each send messages during 45 minutes then disconnect from the queue. All the producers are hosted on the same instance. From the beginning, there are twenty listeners connected to the queue. They subscribe to every message of the queue, this means that each listener receives all the messages produced by a producer (fan-out). The twenty listeners are hosted on 4 different instances, which makes 5 listeners per instance. The test is run five times independently at different moments on the Amazon infrastructure and mean values have been derived.

The rules for the workload are: create a new exchange instance when its outgoing bandwidth is higher than 15 Mbps/s, and relocate producers on free exchanges when the exchanges transit more than 15 Mbits/s of messages. The SLA business rule we set up is to maintain no more than 4 exchange instances at the same time.

B. Results

Figure 5 shows the number of messages produced by the producers at each minute of the run and the number of message that has been processed by the exchanges, that means, the number of messages that have been received by the exchanges and transferred to the connected listeners. We can observe that the infrastructure and EQS monitoring are able to trigger the right exchanges scale-out actions with regard to the scaling algorithm which permits the queue to cope with its actual load. Figure 7 shows the individual number of messages processed by each of the 4 exchanges of the queue. We observe that over different runs new exchanges are not always started at the same minute during the test run but within a time interval of 2-3 minutes. This is due to many factors of the platform we work on. Firstly, we rely on the Amazon infrastructure which provides best effort service level what concerns network I/O. Secondly, we rely on the Scalr platform for the triggering of infrastructure rules which polls the metrics at intervals of 1 minute. Figure 6 depicts the actual number of messages that are received by the twenty connected listeners. With this setup our system is able to handle more than one million messages per minute with 5 kilobytes of payload.

We have measured the latency of the message delivery on a subset of 400 messages sent each minute[10]. Figure 8 represents the mean latencies of the delivery of those 400 messages at each minute of the test run. We observe that the mean latency remains in the same range throughout the run, independently of the load applied to the queue, thanks to the scaling out of exchange instances. This variation in the latency results from the best effort nature of Amazon EC2 network I/O.

It is worth mentioning that we have performed the same test with the same setup on our private cluster composed of commodity hardware (gigabit private LAN with max. 1 hop between instances). The mean latency measured then was under the millisecond for a message. We also achieved performance 5 times higher without message dropping in our lab. The depicted results here show the highest throughput that we could achieve on EC2 *small* instances without message dropping by the ZeroMQ library with a setup of 4 exchanges. Therefore these should not be taken as pure performance results but more as a validation of the elastic and dynamic nature of our algorithm.

[10]In order to avoid clock drifting on cloud instances, the producer and listener of these 400 messages were hosted on the same instance synced with a public NTP server using the appropriate hypervisor time management flags.

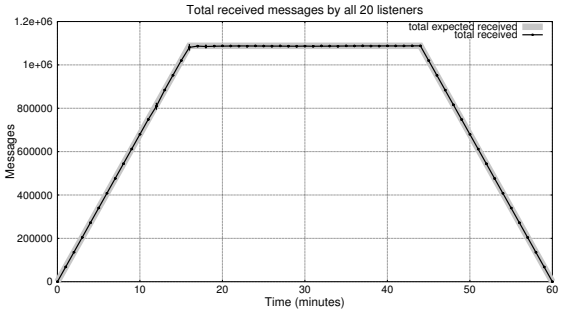

Fig. 6. Total delivered messages cumulated on the 20 listeners. Messages are sent in fan-out mode, which means each message sent by one of the 16 producers will be received by all the 20 listeners. This plot represents the average number of messages that is actually received by the 20 listeners over the 5 runs. On overlay is the total number of messages generated by the 16 producers.

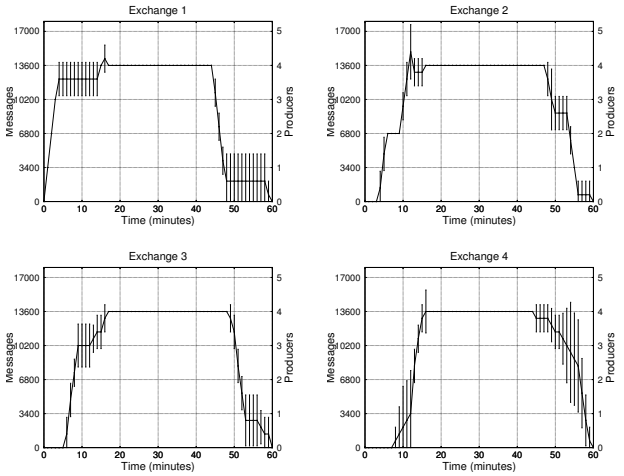

Fig. 7. Number of messages processed by the 4 queue exchanges. The plots depict the mean values and standard deviation computed on 5 independant runs. At the beginning, only Exchange 1 is active. As producers connect to the queue and the load increases, Exchanges 2,3 and 4 are started by the rules and scaling management. The second y-axis on the right depicts the number of users connected to an exchange corresponding to a processed number of messages.

VI. Discussion and Future Work

The results from the previous section have shown that our proposed messaging architecture is able to scale by creating additional exchanges in response to increases in the number of messages being sent. The elastic scaling ability of the exchange enables EQS to cope with actual messaging loads. What must be also taken in consideration is the scaling of the management components of EQS. These should also scale when there are thousands of queues, producers and listeners to manage, and several millions of exchanges per queue. Note that even if we have performed the evaluation on one queue, the monitoring and management components are self-contained per queue. Queues are not related between them and have each their own set of monitoring, management and exchange components. When dealing with multiple queues,

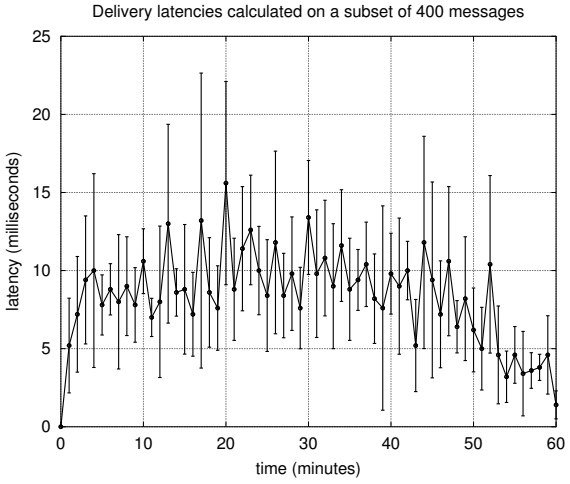

Fig. 8. Mean latency of the message delivery and standard deviation during the run, calculated on 5 different runs. Latency is calculated on a subset of 400 messages each minute.

these sets of components are duplicated for each queue.

In the current design of the queue management system, the metadata storage and the auto-scaling decision layer are not scalable. The queue monitoring system is implemented by a central entity which has a global view and understanding of the system. The metadata storage maintains the most important information about the queue composition in terms of existing exchanges, SLA, geographical locations, their current load and ability to scale. It is also a central entity.

An evolution of this architecture would be to replace this central metadata storage by a distributed system on top of a distributed storage such as HBase [8] or Cassandra [9]. In front of this storage, we will implement a central queue that will distribute control events to a set of workers which will in turn update the metadata and contextual data on the distributed storage. Going further, we can also imagine implementing the concept of data affinity by forwarding the metadata update requests to a worker running on the same machine where the data is located. In order to implement such mechanisms, we need to coordinate the event routing with the sharding policy of the underlying distributed storage. Figure 9 summarizes this approach, using Apache ZooKeeper[11] for the cluster management. The advantage of having a worker pool is to be able to smoothly handle multiple requests on the same data object and even to implement priority rules as described in the LMAX architecture [10], [11]. This architecture is inspired by the Staged Event-Driven Architecture (SEDA) concept developed by [12].

Concerning the infrastructure monitoring system, we can use an IaaS monitoring subsystem or use existing solutions [13] used for monitoring large data centers such as Nagios. Many of those solutions are also used within cloud providers for the infrastructure management [14].

[11] Apache ZooKeeper: http://zookeeper.apache.org

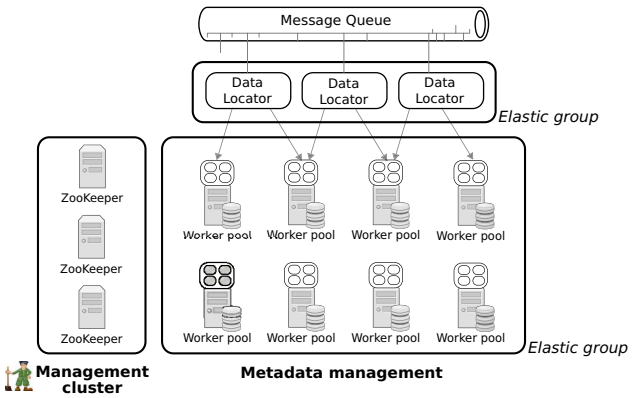

Fig. 9. Architecture of the distributed metadata management. The Data Locators are aware of the data sharding policy, such as the consistent hashing used in Cassandra for example, and can then route the update request to the right node. Then a stateless worker from a worker pool is assigned to execute the update request. Apache Zookeeper is used here for the cluster management.

We have not yet implemented the persistent aspect of the message queues. The idea would be to forward each message to a pool of workers dedicated to write messages in the NoSQL storage. A final acknowledge should guarantee the message persistence. In case of failure of the exchange during the message forwarding, the producer client library should be notified.

An elastic queuing system such as EQS can be used for communication between applications but also at different levels of the IaaS management. One of the next steps of EQS is to implement a PoC of a scalable monitoring system receiving message probes from a lot of different components such as VMs (Virtual Machines), application servers, applications, hardware, switches, and to transport them to a set of distributed CEP instances. This PoC will be implemented on OpenStack[12]. OpenStack is a good example of an IaaS management implemented using the EDA paradigm. When a VM provisioning request is sent to the Nova-API module, a completely distributed orchestration process is launched in order to create the VM, mount the volumes, and register it to the network. Each step of this provisioning process is done through a message queue. However, this queuing system is currently based on Rabbit MQ and thus is not elastic [15]. EQS could be used at that level in order to support a completely elastic provisioning process.

VII. Conclusion

Emergence of cloud computing and on-demand resource usage schemes have enabled applications to scale out elastically. In this context, we have presented EQS, and elastic message queue which is able to scale out elastically in order to adapt with the existing load. This type of message queue is suitable for large scale applications built around event-driven architectures but also for cloud infrastructures management. We have shown with load tests that EQS is able to scale out elastically in order to adapt with the applied messaging load. We have proposed architecture tracks for all the management components of EQS which can make them scale elastically, as with the current state of the implementation only the messaging component is able to scale out elastically. We have also discussed about possible integration of EQS to the cloud infrastructure management.

[12]OpenStack Compute: http://www.openstack.org/projects/compute/

References

[1] P. Pietzuch, D. Eyers, S. Kounev, and B. Shand, "Towards a common api for publish/subscribe," in *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, ser. DEBS '07. New York, NY, USA: ACM, 2007, pp. 152–157. [Online]. Available: http://doi.acm.org/10.1145/1266894.1266924

[2] P. T. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, pp. 114–131, 2003.

[3] H. Subramoni, G. Marsh, S. Narravula, and D. K. Panda, "Design and evaluation of benchmarks for financial applications using Advanced Message Queuing Protocol (AMQP) over InfiniBand," in *Proceedings of the Workshop on High Performance Computational Finance, WHPCF 2008*. IEEE, Nov. 2008, pp. 1–8.

[4] G. Marsh, A. Sampat, S. Potluri, and D. Panda, "Scaling Advanced Message Queuing Protocol (AMQP) Architecture with Broker Federation and InfiniBand," Ohio State University, Tech. Rep. OSU-CISRC-5/09-TR17, 2008. [Online]. Available: ftp://ftp.cse.ohio-state.edu/pub/tech-report/2009/TR17.pdf

[5] H. Mattsson, H. Nilsson, C. Wikstrm, and E. T. Ab, "Mnesia - a distributed robust dbms for telecommunications applications," in *Practical Applications of Declarative Languages: Proceedings of the PADL1999 Symposium, G. Gupta, Ed. Number 1551 in LNCS*. Springer, 1999, pp. 152–163.

[6] B. Uddin, B. He, and R. Sion, "Cloud Performance Benchmark Series - Amazon Simple Queue Service (SQS)," Stony Brook Network Security and Applied Cryptography Lab (NSAC), Tech. Rep., 2011.

[7] L. Rodero-Merino, L. M. Vaquero, V. Gil, F. Galán, J. Fontán, R. S. Montero, and I. M. Llorente, "From infrastructure delivery to service management in clouds," *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1226–1240, Oct. 2010.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 24, no. 2, 2008.

[9] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[10] D. Farley and M. Thompson, "LMAX - How to do over 100K contended complex business transactions per second at less than 1ms latency," 2010. [Online]. Available: http://www.infoq.com/presentations/LMAX

[11] M. Fowler, "The LMAX Architecture," 2011. [Online]. Available: http://martinfowler.com/articles/lmax.html

[12] M. Welsh and D. Culler, "SEDA: an architecture for well-conditioned, scalable internet services," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, Dec. 2001.

[13] K. Buytaert, T. De Cooman, F. Descamps, and B. Verwilst, "Systems Monitoring Shootout: Finding your way in the Maze of Monitoring tools," in *Proceedings of the Linux Symposium*, Ottawa, Canada, 2008, pp. 53–62.

[14] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, ser. CCGRID '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2009.93

[15] Rackspace Cloud Computing, "RabbitMQ and Nova," 2010. [Online]. Available: http://nova.openstack.org/devref/rabbit.html