Middleware for Processing Message Queues with Elasticity Support and Sequential Integrity of Asynchronous Message Processing

Eduardo Henrique Teixeira
Department of Computer Science - CIC, University of Brasilia (UnB), Brasília, DF - Brazil
E-mail: edu.henr@gmail.com

Aletéia Patrícia Favacho de Araújo
Department of Computer Science - CIC, University of Brasilia (UnB), Brasília, DF - Brazil
E-mail: aleteia@cic.umb.br

Abstract. Elasticity in computing refers to dynamically adjusting the amount of allocated resources to process a distributed application. In order to achieve this, mechanisms are needed to avoid the phenomenon of the elasticity threshold detection moving constantly up or down. The existing work fails to deliver sequential integrity of asynchronous messages processing and the asymmetries of data distribution to achieve parallel consumption. This paper fills this gaps and proposes a middleware solution to dynamically analyze the flow of message queue, and a mechanism to increase the parallelized consumption based on the output behavior. An architecture for IOD (Increase On Demand) middleware is presented, with support for the increase and decrease of thread's to cope with the growth of message queues, using the technique of limit-based heuristics over a given period of time and grouping messages into sub-queues based on classification criteria.

1. Introduction
Elasticity is the characteristic of an environment that defines the degree to which a system is able to adapt dynamically to changes in workload, through automatic provisioning and release of resources [10]. In short, elastic computing is the dynamic provisioning of resources [17]. Elastic computing brings significant benefits to application providers, including cost savings and prevention of super- and sub-provisioning of IT resources. This occurs through the monitoring of demand and the acquisition of resources required by applications to achieve a high level of quality [12].

The aim of elasticity is that the amount of resources allocated to a service is what it really needs. It is therefore possible, for example, to reduce the number of servers required to handle the FIFO (First In First Out) [7] message queues (interprocess communication mechanisms to access data asynchronously from multiple processes) of a distributed system and, consequently, to save computational resources. It is also possible to determine the current processing load on a cluster, which can be measured in terms of the flow of outgoing messages from a queue and may be limited by CPU consumption, in order to avoid saturating the servers. This is
where a middleware solution can be used: one that is capable of transparently and dynamically implementing the concept of elasticity in order to ensure that processing can adapt to the growth of message queues and, to avoid messages accumulating before they are processed.

In addition to elasticity, fault tolerance is essential in distributed platforms in order to allow one or more processes to fail without harming the rest of the system [20]. In distributed systems, failures are common and may occur due to errors in hardware or software [28]. Hardware failures generally result from physical degradation of components, while software ones occur due to errors in design or implementation, and are also known as bugs [28]. In this context, the tolerance of software failures guarantees correct application behavior, regardless of the number and type of failures occurring [4].

This article proposes the IOD (Increase On Demand) middleware as a valuable tool to ensure elasticity and software fault tolerance in distributed systems with message queue-based architectures, such as high performance and high availability clusters. IOD middleware dynamically analyzes the behavior of the message queue outputs to determine whether to increase or decrease the number of threads responsible for handling messages. The goal of the proposed middleware is to prevent the accumulation of messages to be processed. The IOD also analyzes the behavior of CPU consumption to determine the limits of scalability of servers so as to avoid saturation of the cluster. In addition, it uses mechanisms for fast recovery in case of failures, ensuring that services maintained by the cluster become available again quickly.

IOD middleware’s focus is to use elastic computing to determine how to most efficiently utilize the processing cores available in the cluster. To achieve this, the number of threads required to handle the message queues is calculated dynamically according to the behavior of the average queue output and the average CPU consumption of each node. Distributed applications that use message queues as an IPC (Interprocess Communication) mechanism [7] can also benefit from the techniques outlined in this paper. This is possible because the allocation of resources in a cluster is performed at each node, avoiding CPU idle periods.

The remainder of this paper is divided into four sections. Section 2 contains an overview of related elasticity and fault tolerance work. Section 3 describes the architecture of the proposed IOD middleware. Section 4 presents the evaluation and analysis of tests done with the IOD middleware, initially in a simulated environment and then in a real distributed application that uses message queues. Finally, Section 5 presents some conclusions and outlines the next steps of this work.

2. Related Work
To achieve transparent and automatic elasticity, the approaches [14][23] are appropriate when sequential integrity of the processing of messages in the queues is not a functional requirement. The messages in the queues can therefore be redistributed according to the required output, without worrying about the order in which they are processed. However, this can be a disadvantage when it is necessary to maintain sequential integrity in processing the queues for the consumers. In this case, the events generated by consumers need to respect the time constraint in which the messages were generated.

The assignments made by publishers to subscribers can generate asymmetric distribution of data [6][13][23]. These asymmetries are also common in systems with multiple message queues and multiple consumer [13][23]. Identifying the best candidate for consumption of the queues is necessary to reduce latency and avoid saturation of overloaded consumers. Thus, it is possible to determine these candidates identifying the highest average flow [13][23]. The lowest average CPU consumption can also be used as a metric to identify the best candidates for the consumption of new items [13], or by load balancing policy based on flow control [24].

In distributed systems with CPU-bound characteristics, it is possible to explore the transparency of elasticity of an application [11][14][23]. This transparency can be achieved using the metric of load in terms of CPU consumption [11][13][19]. However, if the number of
messages remains close to the detection limit of elasticity, problems of rapidly expanding and reducing the thread pool can occur. To solve this problem and keep the output at acceptable levels, [11][13][23] propose the use of limit-based heuristics, for a given amount of time. In this way, the created threads are kept for a longer period, thereby minimizing the cost of restarting them and reducing the latency of messages consumption. This approach is a potential solution to the problem of up-and-down elasticity threshold detection.

Approaches [8][11][12][14][19][23][24][25] mention that elasticity must be achieved transparently and automatically. They also discuss the use of the publisher/subscriber design pattern [6][13][23][24] to achieve elasticity and the decoupling of producers and consumers. With the approaches cited in [11][12][13][19], elasticity is achieved according to the required load demand, which is measured by CPU consumption. Elastic operations can be made transparently without any knowledge of the application, such as the expected time for consuming messages [25]. However, handling elasticity at application level in systems that are composed of redundant components that recover from failures controlled by a middleware also needs to be considered [26].

Many of the approaches to fault tolerance use replication so that another set of processes can take over in case of hardware or software failure [1][2][9][16]. However, techniques such as fast failure recovery [2][3][9][27] are interesting because they avoid the cost of additional memory and communication between replicated processes and the exchange of state information. Papers [3][16][24] propose support for fault tolerance transparently in the middleware. This approach is attractive because it abstracts away from the user application the details of detection and recovery of implemented features. In order to achieve this transparency, [2][16] detect failure by monitoring the connection between processes. A fail-over mechanism that resend a given message transparently to another queue if its initial destination is unavailable [24].

3. Proposed IOD Middleware

To create efficient elastic environments, existing services must be extended with elastic computing capabilities and policies of resource provisioning on demand [15]. The IOD middleware proposes support for elasticity by means of dynamic adaptation of the number of threads for handling messages, based on an analysis of the inbound and outbound throughput of the queues and server CPU consumption. Support for fault tolerance, in IOD middleware, is achieved by detecting the failure and recovering with a quick restart. In this way, the messages in the queues continue to be processed from the point where the failure occurred, without the need for a replication process. These features will be presented in the following sections.

3.1. Message’s Handling Architecture

To achieve parallel consumption, it is first necessary to separate the messages from a queue into distinct groups, according to a classification criterion. Each queue thus gives rise to several sub-queues, each indexed by a key that identifies the group. This should be done to increase the number of worker threads that will be responsible for processing the messages, distributing the groups among the various initiated worker threads and thus increasing the throughput by means of parallel consumption.

A system with multiple consumer queues can generate distribution asymmetries [6][13][23], which leads to an imbalance in the system when consuming the messages in the sub-queues. The IOD middleware deals with this by sorting the groups, when the worker threads are created or destroyed, starting from the thread that has the most messages through to the one that has the fewest. A round robin algorithm is then used to distribute these groups among the available worker threads. Furthermore, at each round of consumption, one message from each group is processed in accordance with FQ (Fair Queueing), an algorithm that allows multiple message queues to share the same processing capacity, ensuring fairness in consumption and avoiding
starvation due to heavy flows. First Come First Served algorithm [22] is also used to preserve the sequential integrity in the consumption of messages in the same group.

In order to achieve elasticity in the IOD middleware, it is necessary to analyze message queue throughput, the limitation of creating worker threads per CPU and the standard deviation of the average output per group, as well as failure recovery with elasticity, which will be described in the following subsections.

3.2. Analysis of Average Input and Output Flow

During a burst of incoming messages, if a distributed application is unable to process them as quickly as they arrive, an accumulation in the queues will take place, resulting in delays in processing and, consequently, a reduction in the quality of the provided services. To determine the number of worker threads required to cope with queue buildup in the presence of message bursts, it is necessary to calculate the average input and output flow, and the average ratio of incoming to outgoing messages. These calculations are performed from the time queue growth is detected, here called Growth Detection \((GD)\), until the Critical Point \((CP)\), which is the moment when new worker threads are created. The \(GD\) occurs when the ratio of incoming messages to output messages becomes larger than one, causing queueing. The \(GD\) is detected by Equation 1,

\[
\left(\frac{\text{Input}}{\text{Output}} > 1\right)
\]  

(1)

where \(\text{Input}\) is the number of incoming messages and \(\text{Output}\) is the number of outgoing messages. The mean ratio between the incoming and outgoing messages is called \(\text{avgIO}\).

New worker threads are created at the time when the accumulation in the queue cannot be dealt with before the end of the maximum time set for handling messages. This moment occurs at the \(CP\) and the action of increasing worker threads is called Scale Up \((SU)\). The \(CP\) is found when the number of messages in the queue \((\text{QueueSize})\) is greater than the \(\text{MaxQueueSize}\), given by Equation 2,

\[
(\text{avgTOUT} \times (\text{schTIME} - (\text{hNOW} - \text{hSTART})))
\]  

(2)

where variable \(\text{avgTOUT}\) is the average output flow, \(\text{schTIME}\) is the maximum time for the treatment of messages in the queue, \(\text{hNOW}\) is the current time and \(\text{hSTART}\) is the start time, i.e. the moment at which the value given by Equation 1 becomes true.

The Scale Down \((SD)\) occurs at the Exit Point, which is the time where the worker threads are removed. It is obtained by means of Equation 3,

\[
(\text{QueueSize} < \text{avgTIN})
\]  

(3)

where \(\text{QueueSize}\) is the current size of the queue, and \(\text{avgTIN}\) is average input flow since the \(CP\).

In addition to the Contention Threads \((CT)\), which are the worker threads intended for burst containment, another set of worker threads is needed, called Zero Threads \((ZT)\), to consume the messages that have accumulated in the queue between the \(GD\) and the \(CP\). Therefore, various worker threads are created, of types \(ZT\) or \(CT\), and each one is associated with several message sub-queues separated by groups, so that throughput increases as a result of the parallelized consumption. The \(CT\) and \(ZT\) threads are shown in Figure 1.

The calculation of the total number of worker threads started in a \(SU\), is shown in Algorithm 1. When \(\text{QueueSize}\) is greater than the \(CP\), line 2, the value of \(\text{avgIO}\) is used to find the number of \(CT\) worker threads, line 3. The variable \(\text{avgTOUT}\) divided by the total number of worker threads results in the average flow by thread, line 4, which is used as the rate of consumption,
line 5, to obtain the number of \( ZT \) worker threads, line 6, necessary to reset the messages that have been accumulated. To avoid the problem of elasticity threshold detection [11], the IOD middleware starts one worker thread more than the sum of \( CT \) and \( ZT \), line 7. This started thread is a \( CT \) type and is intended to avoid the problem of elasticity threshold detection [11].

\[ \text{Algorithm 1: Worker Threads Started in the SU.} \]

1. MaxQueueSize = \((\text{avgTOUT} \times (\text{schTIME} - (\text{hNOW} - \text{hSTART}))\);
2. if \((\text{QueueSize} > \text{MaxQueueSize})\) then
3.   ContentionThreads = \text{avgIO};
4.   ThroughPutByWorkerThread = \((\text{avgTOUT} / \text{TotalWorkerThreads})\); 
5.   ConsumptionRate = \text{QueueSize} / \((\text{schTIME} - (\text{now} - \text{hSTART})\));
6.   ZeroThreads = ConsumptionRate / ThroughPutByWorkerThread;
7.   TotalThreadsToStart = ContentionThreads + ZeroThreads + 1;
8. end if

The calculation of the number of threads that are removed in the \( SD \), is shown in Algorithm 2. To perform the \( SD \) it doesn’t matter if the threads that will be removed are the type \( CT \) or \( ZT \), and it is necessary that the value given by Equation 3 is true, as shown in line 1 of Algorithm 2. Then the throughput per thread is calculated, line 2, which is used in the calculation of threads to be removed, line 3. Again, to avoid the phenomenon of elasticity threshold detection [11], one fewer thread is removed, line 4.

3.3. Analysis of CPU Consumption

Knowing the average CPU consumption by worker thread is useful to determine limits for the \( SU \) (Scale Up). Thus, when bursts of incoming messages are detected, a process is started for
Algorithm 2: Worker Threads Removed in the SD.

1 if (QueueSize < avgTIN) then
2   ThroughPutByThread = (avgTOUT / TotalWorkerThreads);
3   ThreadsToRemove = TotalWorkerThreads - (avgTIN / ThroughPutByThread);
4   ThreadsToRemove = ThreadsToRemove - 1;
5 end if

measuring average CPU consumption, per worker thread and per cluster node, to determine if the required throughput to contain growth and drain the queue can be achieved without saturating the server CPU.

3.4. Analysis of Standard Deviation of Average Output Flow per Group
The bursts of incoming messages of a distributed system that handle message queues may originate from several different groups. To handle the bursts from n groups, the IOD middleware creates a number of worker threads, according to the average output flow, calculated during the growth of the queue, as described in the analysis of the average input and output flow, presented in Section 3.2.

However, when the bursts originate from a small group of messages, creating many worker threads can mean a waste of resources and, depending on the consumption throughput, may not be enough to contain the growth of the queue and still generate recursive SUs (Scale Up), without solving the problem of queue growth. To solve this problem, before the SU and during the burst of messages, the average output flow per group is measured and, this will be used to calculate the standard deviation at the moment the SU is performed.

A random variable in a normal distribution has a 95% chance of being less than two standard deviations from its mean [5]. Thus, since the average output flow was normalized, in at most one message from each group per round of the FQ algorithm, after n rounds, when the average output moves more than two standard deviations away from the mean, it is possible to detect bursts coming from specific groups. In this way, the amount of worker threads created is limited by the number of groups having an average output flow greater than twice the standard deviation identified during the burst. This approach aims to create worker threads specifically for the groups in which bursts where detected.

3.5. Fault Tolerance
In the IOD middleware, software fault tolerance features were supported for process failure, but hardware failures were not considered. Therefore, the messages in the queues could continue to be processed from the point where the failure occurred, without requiring the use of replication processes.

To support fast failure recovery mechanisms [2][3][9][27], if a process fails and the number of messages in the queue is greater than the CP (Critical Point), mechanisms of elasticity in the middleware are required to perform the SU (Scale Up). Thus, when the processes are restarted after a failure, if the input bursts continue, the SU occurs rapidly by detecting the CP larger than the QueueSize. The incomplete message being processed by the failing process goes back to the queue to be handled by the process when it restarts. However, at this time another mechanism is needed to provide elasticity, if the bursts ended after the failure point but before the process restart. In this situation, if the number of messages in the queue is above avgTOUT x schTIME (the measurements of throughput) before deciding to perform the SU, it’s performed using only 10% of schTIME. This is necessary to perform the SU more quickly, which is the
case when the bursts end after the failure and before the quick restart of the process. The same scenario occurs when bursts of messages do not stop, even after the quick restart of the process.

The efficient implementation of fault tolerance in a distributed environment, relies on a mechanism of elasticity, exactly as implemented by the IOD middleware.

4. Results

The tests for the IOD middleware were performed in both a simulated environment and a real environment. For the simulated tests of the IOD middleware, a simulator was created with two threads: a producer that writes packets to the message queue and a consumer that reads and delivers the messages to the IOD middleware. For the simulated tests, \textit{schTIME} was given the value of 80 seconds because this is maximum time of message handling required by the real distributed application that will use the IOD middleware. The production thread was programmed to generate messages at a rate five times greater than the consuming thread. The purpose of this setting is to cause message accumulation, and demonstrate the operation of the elasticity proposed by the IOD middleware. After testing in a simulated environment, the IOD middleware was deployed in a real distributed application that uses message queues to validate if the middleware achieved the goal of dynamically adapting to the throughput conditions and avoided queueing. The tests were divided into five groups, as described below.

4.1. Creating and Terminating Threads Dynamically

The tests presented in this section are intended to verify the functioning of dynamic creation and termination of worker threads, based on the average flow of incoming and outgoing messages.

Tests were carried out with the producer creating a burst of packets every 10 milliseconds and the consumer processing them every 50 milliseconds, i.e. generation of messages at a rate five times greater than consumption. Accordingly, the experiments showed that there was an adequate increase of the number of worker threads to increase the flow rate of consumption, as shown in Figure 2, in which the $x$-axis corresponds to elapsed time and the $y$-axis corresponds to the value of the variables $QueueSize$, $MaxQueueSize$, $avgTIN$ and $avgTOUT$. These variables correspond respectively to the values of the queue length; the value given by Equation 2, average input flow, and the average output flow.

In the scenario presented in Figure 2, the \textit{CP} was detected with 1130 messages in the queue, 14 seconds after the start of the message burst, as shown in Table 1. At 9 seconds after the \textit{SU}, the middleware decided to terminate five threads so that the average output flow would be compatible with the average input flow. Here, the IOD middleware proved satisfactory in terms of creation and termination of worker threads based on the throughput, because it was able to create 12 worker threads and end the bottleneck in a period of 23 seconds, i.e., within the limit of message handling defined as 80 seconds. The number of messages reached a controlled level, because the stabilization of input and output messages occurred with an acceptable number (33) in which the number of messages in the queue doesn’t grow beyond the Critical Point.

| Moment         | Duration (sec.) | Input Messages | Output Messages | Threads | Queueing |
|----------------|-----------------|----------------|-----------------|---------|----------|
| Start          | 14              | 99             | 19              | 1       | 0        |
| Scale Up       | 9               | 99             | 197             | 12      | 1.130    |
| Scale Down     | 2               | 99             | 99              | 7       | 33       |

Table 1: Times of the Elasticity Algorithm.
4.2. Creation of Threads Limited by Average CPU Consumption

The purpose of this test is to determine if the elasticity algorithm prevents saturation of CPU consumption on the servers by checking the behavior of the increase of worker threads limited by the average CPU consumption. The worker threads were configured to use the busywait technique in order to simulate high CPU usage in the simulation. The tests were performed with the producer thread generating messages every 10 milliseconds, and the consumer thread processing them every 50 milliseconds, resulting in message accumulation.

The results of the message queue, CPU and threads are shown in Figures 3a and 3b. In Figure 3a, the x-axis is the elapsed time, and the y-axis shows the variables QueueSize, MaxQueueSize, the avgTIN and avgTOUT. These variables correspond, respectively, to the values of the queue length, the value given by the Equation 2, average input flow, and the average output flow. In Figure 3b, the x-axis is the elapsed time in seconds and the y-axis is the percentage of CPU usage of the process and the machine, or the number of threads in the process.

As can be seen in Figure 3, the reduction in the number of messages occurs more slowly because the number of threads is limited based on the average CPU usage of the worker threads so as not to exceed 100% of the server. Like the previous experiment the decrease is also linear but it happens in a lower rate, as shown in Figure 2, because not all the worker threads necessary for processing the entire queue in the required time were created. According to Table 2, the middleware should have created five worker threads based on the average input flow. However, as shown in Table 3, the average CPU consumption by worker thread was 15.28%, and would exceed the limit of the residual server CPU, which was 83.60%. The middleware therefore created only four worker threads (1 fewer) to avoid saturating the CPU usage of the machine. At the time of Scale Down, the average output flow was at 107 messages per second and therefore below the average input flow, which was 115.5 messages per second. As a result, the middleware did not detect the need to remove worker threads.

It is important to consider the saturation of the environment in order to make the distributed
application capable of dynamically identifying the limits of the workload that it is capable of processing given the size of its infrastructure. If the system identifies that it needs to increase its capacity but cannot do so due to imposed limitation, it can generate alarms so that additional adjustments can be made according to the required load.
Table 2: Behavior of the Threads with CPU Limits.

| Moment       | Threads Needed | Created Threads | Removed Threads |
|--------------|----------------|-----------------|----------------|
| Scale Up     | 5              | 4               | N/A            |
| Scale Down   | N/A            | N/A             | 0              |

Table 3: Behavior of the CPU in the Elasticity Algorithm with Limits.

| Moment       | Worker Thread Mean | Total CPU Needed | Residual CPU |
|--------------|-------------------|------------------|--------------|
| Scale Up     | 15.28%            | 91.72%           | 83.60%       |
| Scale Down   | N/A               | N/A              | N/A          |

4.3. Creating Threads Limited by Standard Deviation

These tests aim to verify the operation of creating worker threads limited by the number of groups, whose average output flow exceed two standard deviations from the mean. The idea is to analyze if, during a burst of packets from a few groups, the worker threads are created based only on the analysis of the average output flow, because in this case resources are wasted with threads that will not contain the growth of the queue. This procedure is designed to test the behavior of the IOD middleware before the detection of groups that have isolated bursts of messages and check whether the decreased use of resources occurred through the creation of fewer worker threads.

Consecutive burst tests were performed for three different groups, and once again the producer thread generated messages at intervals of 10 milliseconds and the consumer thread processed at intervals of 50 milliseconds. For the first group the bursts were generated for the first 16 seconds, for the second group they were generated from seconds 17-38, and for third group they were generated from seconds 39-75. Figure 4 shows the behavior of the message queue, where the x-axis corresponds to the elapsed time and the y-axis corresponds to the value of the variables QueueSize, MaxQueueSize, avgTIN, avgTOUT and Threads. These variables correspond, respectively, to the values of the queue length, the value given by Equation 2, the average input flow, the average output flow, and the number of threads in the process (on the scale of 100:1).

Despite three threads having been created, one for each group at the three critical points, the message queue cannot be contained, as shown in Figure 4. This behavior occurs because the message generation is five times higher than consumption, and the middleware cannot create more than one thread per group due to the restriction of sequential integrity in processing data from the same group. Therefore, despite the increase in the number of threads, the number of messages in the queues does not decrease because there is no possibility of separating the data from the same group for parallel processing. Moreover, it is possible to observe in Figure 4 that, after the first SU, despite the queue size being above the value of MaxQueueSize, the next SU only occurs after the queue doubles in size compared to the previous SU, to avoid SUs occurring very close to each other. It was possible to save resources by creating worker threads for the treatment of specific groups with isolated bursts and, hence, fewer threads than determined by the algorithm associated with the analysis of the flow, as shown in Table 4.
Figure 4: Behavior of the Message Queue with Bursts in Three Groups.

Table 4: Creating Threads Limited by Standard Deviation.

| Moment            | Threads (Throughput) | Threads (Standard Deviation) |
|-------------------|----------------------|-----------------------------|
| Critical Point 1  | 6                    | 1                           |
| Critical Point 2  | 9                    | 1                           |
| Critical Point 3  | 11                   | 1                           |

4.4. Elasticity with Fault Tolerance

The objective of this test is to validate if, after a failure while processing messages, the elasticity is initiated without the presence of packet bursts, and after a quick restart of the process. The simulation involved a process crash followed by a quick restart, without any message bursts and the queue containing more messages than $MaxQueueSize$.

In Figure 5, the $x$-axis corresponds to the elapsed time and the $y$-axis corresponds to the value of the variables $QueueSize$, $MaxQueueSize$, $avgTIN$ and $avgTOUT$. These variables correspond, respectively, to the values of the queue length, the value given by Equation 2, the average input flow, and the average output flow. After restarting the process, despite the $QueueSize$ exceeding the $MaxQueueSize$, an increase in the number of consumer threads only occurs after 10% of $schTIME$ (8 seconds). This approach aims to analyze the behavior of input and output flows before a $SU$, generating the averages for the calculations of the number of worker threads required to drain the message queue before the end of the maximum message processing time ($schTIME$). A few seconds after the restart of the process, increases in the consumption of messages indicate an increase in the number of worker threads to drain the messages from the queue before the end of $schTIME$. This behavior demonstrates that, even with the process crash, the IOD middleware can be elastic and tolerant against a demand generated and terminated before the quick restart.
4.5. Distributed Real Application

This section analyzes the tests of the IOD middleware in a real distributed application running on a high performance cluster that consists of 11 servers, each one with four cores and four gigabytes of memory. All 11 servers run Microsoft Windows 2008 R2 64 bit operating system, and are virtualized from two servers 360C BLADE G8, each one with two Intel Xeon Quad Core/Eight Threads model E5540 2.53 GHz processors and 96 gigabytes of memory.

Each server in the cluster runs a distributed application with 19 processes, one being the coordinator that performs IPC between coordinated processes through MSMQ (Microsoft Message Queue). Each of these processes is responsible for a distinct feature of the distributed application and has an independent message queue which is fed every 80 seconds. Before using the IOD middleware in a real distributed application, it was necessary to modify the IPC protocol by adding a group identifier to indicate with which group the messages are associated. This procedure allows the IOD middleware to group related messages and to then divide sub-queues between worker threads.

Initially, data was collected for the time required for consumption of the messages from the queues of the 11 servers, over a period of 6 hours, under heavy message traffic. It was found that the average time for processing messages during this period was 5.2 seconds. Given that the maximum treatment time of the messages (schTIME) is 80 seconds, the distributed application’s servers used the CPU in very short bursts and were idle most of the time (93.5%). In order to utilize the available resources on this platform more efficiently, the 11 servers of the cluster were grouped into a single server. Whereas the treatment time of each message is not affected by grouping servers (because all nodes have the same hardware configuration) the time for processing messages is 6.5% (5.2 seconds of 80) on average. Therefore, to reduce the number of servers by a factor of 11, an increase of 71.5% of the server’s processing capacity is required. Doubling the capacity of a server is more than sufficient in this case, so the virtualized server had its configuration increased from four to eight cores and four to eight gigabytes of memory.
In this setup, when bursts of messages occur, the distributed application has a buffer of CPU and memory to contain the queueing and ensure scalability of the environment.

After running the real distributed application, it was found that one of the 19 processes had a backlog of messages in its queue, as shown in Figure 6. To contain this accumulation, 15 seconds after the start of the burst of messages the IOD middleware enabled the creation of more worker threads, by observing that the \textit{QueueSize} (1.226) was greater than the \textit{MaxQueueSize} (1182). Six more worker threads were created with an estimated time of 65 seconds to drain the message queue, below the maximum processing time (\textit{schTIME}) of 80 seconds. After the creation of the worker threads the \textit{avgTOUT} was greater than the \textit{avgTIN}, indicating increased consumption parallelism achieved by means of elasticity, to contain the burst of messages as shown in Figure 6.

As the average CPU usage per worker thread was 0.10\%, and the residual CPU server was 99.57\%, the IOD middleware did not detect the need to limit the creation of worker threads due to CPU consumption. The IOD middleware also detected no cases of groups with isolated bursts, thereby not creating worker threads based on the standard deviation of the average output flow per group.

The tests have shown that is possible to use the IOD middleware to implement elasticity in a real distributed application through the creation and termination of worker threads in accordance with the workload. Another important contribution was to use consumption parallelism in message queues through the clustering of correlated messages in sub-queues. The use of elasticity in the IOD middleware led to a decrease in infrastructure costs, due to being able to reduce the number of servers needed to process the message queue and making more efficient use of each node cluster, to avoid CPU idle time. Creating mechanisms that provide increased throughput of consumption significantly improves the efficiency and the use of infrastructure resources, since the same mass amount of data can be processed in less time and by fewer servers.
5. Conclusions and Future Work

The increasingly high cost of infrastructure equipment shows that it is essential to use available resources optimally and efficiently. Dynamically and automatically scaling resources close to the amount of required processing is important to service providers, as it impacts directly on the cost savings with no loss of service quality.

The IOD middleware proved that it is possible to achieve the requirements of elasticity, adapting to the conditions of the processing, load by increasing or decreasing the number of worker threads according to demand. Moreover, the IOD middleware proved that it is capable of optimizing the use of IT resources, through task parallelism, using multiple worker threads to process the message queues. This study was able to effectively achieve a saving in infrastructure resources, reducing the number of servers from 11 to a single server with double the current configuration needed to process the message queues of a real distributed application. This reduced the cost of the hardware required to run the distributed application and the redundant equipment can be used for other applications as well as in the failover of the cluster that is hardware available in the case of failure or abnormal termination of the previously active system.

A contribution from this paper was the creation of Equations 2 and 3 to dynamically determine the critical points of entry and exit for the parallel processing of message queues, according to the input and output flow, and the detection of growth given by Equation 1. These equations are based on the dynamic behavior of each server and the time limits imposed for the processing of messages. None of the workers cited in Section 2 has this transparent and dynamic adjusting of resources at middleware level according to the necessary time for consuming messages based on QoS (Quality of Service) parameters, defined with each client of the distributed application that uses message queues, such as maximum time for message’s handling or maximum CPU usage. In additional, none of the approaches studied concern themselves with sequential integrity of asynchronous messages processing, nor with the asymmetries of data distribution to achieve parallel consumption, which is achieved by the IOD middleware.

A current limitation of the IOD middleware is linked to the fact that it cannot parallelize the processing of messages in the same group, due to the sequential integrity constraint that imposes ordered processing. As future work, we intend to implement instruction pipelining to optimize the execution of the processing of messages in the same group, without generating sequential integrity conflicts.

References

[1] Heithem Abbes, Christophe Cerin, Mohamed Jenni, and Walid Saad. Fault tolerance based on the publish-subscribe paradigm for the bonjourgrid middleware. 11th IEEE/ACM International Conference on Grid Computing, pages 57–64, 2010.
[2] Tekin Bicer, Wei Jiang, and Gagan Agrawal. Supporting fault tolerance in a data-intensive computing middleware. 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pages 1–12, 2010.
[3] Marcela Castro, Dolores Rexachs, and Emilio Luque. Transparent fault tolerance middleware at user level. 2012 International Conference on High Performance Computing and Simulation (HPCS), pages 566–572, 2012.
[4] George Coulouris, Jean Dollimore, and Tim Kindberg. Distributed Systems Concepts and Design. Publisher Pearson Prentice Hall, fourth edition, 2009.
[5] Douglas Downing and Jeffrey Clark. Applied Statistic. Publisher Saraiva, third edition, 2011.
[6] Wenjing Fang, Beihong Jin, Biao Zhang, Yuwei Yang, and Ziyuan Qin. Design and evaluation of a pub/sub service in the cloud. International Conference on Cloud and Service Computing, pages 32–39, 2011.
[7] John Shapley Gray. Interprocess Communications in Linux: The Nooks & Crannies. Publisher Pearson Prentice Hall, first edition, 2003.
[8] Yike Guo, Rui Hann, Benjamin Satzger, and Hong-Linh Truong. Programming directives for elastic computing. Internet Computing and IEEE, pages 72–77, 2012.
[9] Chen He, Derek Weitzel, David Swanson, and Ying Lu. Hog: Distributed hadoop mapreduce on the grid. *SC Companion: High Performance Computing and Networking Storage and Analysis*, pages 1276–1283, 2012.

[10] Nikolas Roman Herbst, Samuel Kouneif, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX.

[11] Shigeru Imai, Thomas Chestna, and Carlos A. Varela. Elastic scalable cloud computing using application-level migration. *IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 91–98, 2012.

[12] Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. *5th IEEE International Conference on Service-Oriented Computing and Applications*, pages 1–8, 2012.

[13] Ming Li, Fan Ye, Minkyong Kim, Han Chen, and Hui Lei. A scalable and elastic publish/subscribe service. *IEEE International Parallel & Distributed Processing Symposium*, pages 1254–1265, 2011.

[14] Ricky K.K. Ma, King Tin Lam, Cho-Li Wang, and Chenggang Zhang. A stack-on-demand execution model for elastic computing. *39th International Conference on Parallel Processing*, pages 208–217, 2010.

[15] Paul Marshall, Henry Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. *26th International Parallel and Distributed Processing Symposium Workshops & PhD Forums*, pages 1085–1094, 2012.

[16] Rolando Martins, Priya Narasimhan, Lu Lopes, and Fernando Silva. Lightweight fault-tolerance for peer-to-peer middleware. *29th IEEE International Symposium on Reliable Distributed Systems*, pages 313–317, 2010.

[17] Julien Perez, Ccile Germain-Renaud, Balzs Kgl, and Charles Loomis. Responsive elastic computing. *ACM/IEEE International Conference on Autonomic Computing*, pages 55–64, 2009.

[18] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschman. *Pattern-Oriented Software Architecture*. Publisher John Wiley & Sons, first edition, 2000.

[19] Akiyoshi Sugiki and Kazuhiko Kato. An extensible cloud platform inspired by operating systems. *Fourth IEEE International Conference on Utility and Cloud Computing*, pages 306–311, 2011.

[20] Andrew S. TANENBAUM and Maarten Van Steen. *Distributed systems: principles and paradigms*. Publisher Pearson Prentice Hall, second edition, 2008.

[21] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. Publisher Pearson Prentice Hall, fifth edition, 2010.

[22] Andrew S. TANENBAUM and Albert S. Woodhull. *Operating Systems Design and Implementation*. Publisher Pearson Prentice Hall, third edition, 2007.

[23] Nam-Luc Tran, Sabri Shkiri, and Esteban Zimnyi. Eqs: an elastic and scalable message queue for the cloud. *Third IEEE International Conference on Cloud Computing Technology and Science*, pages 391–398, 2011.

[24] Ahmed El Rheddane, Noel de Palma, Alain Tchana and Daniel Hagimont. Elastic Message Queues. *IEEE International Conference on Cloud Computing*, 2014.

[25] Rodrigo da Rosa Righi, Vinicius Facco Rodrigues, Cristiano Andr da Costa, Guilherme Galante, Luis Bona, Tiago Ferreto. AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud. *IEEE Transactions on Cloud Computing*, 2015.

[26] Maria Toeroe, Neha Pawar and Ferhat Khendek. Managing Application Level Elasticity and Availability. *10th CNSM and Workshop*, 2014.

[27] Jun Wang, Yu Deng, Jian-Wen Chen, and Di Zheng. Research of the middleware based fault tolerance for the complex distributed simulation applications. *International Conference on Computational Intelligence and Software Engineering (CiSE)*, pages 1–4, 2009.

[28] Jonathan M. Smith. A survey of software fault tolerance techniques. pages 165–170, 1986.