Response-Time SLO Management with Containers

Francesc D. Muñoz-Escoí, José-Ramón García-Escrivá,
José Ramón González de Mendívil, José M. Bernabéu-Aubán

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

fmunyoz@iti.upv.es, rgarcia@upv.es, mendivil@unavarra.es, josep@iti.upv.es

Technical Report TR-IUMTI-SIDI-2017/001

Response-Time SLO Management with Containers

Francesc D. Muñoz-Escoí, José-Ramón García-Escrivá,
José Ramón González de Mendívil, José M. Bernabéu-Aubán

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
46022 Valencia (SPAIN)

Technical Report TR-IUMTI-SIDI-2017/001

e-mail: fmunyoz@iti.upv.es, rgarcia@upv.es, mendivil@unavarra.es, josep@iti.upv.es

Revised edition (December 2018)

Abstract

Elasticity management strategies for PaaS systems usually rely on service instance create/destroy operations. In the regular case, those service instances are deployed using virtual machines. Recently, light-weight containers have been introduced for a faster management of those create/destroy actions. They also admit resume/pause operations for achieving the same functionality, although with a higher resource consumption.

Response time is a very challenging SLO for interactive services. Proactive elasticity strategies seem to be mandatory for managing that SLO. However, when those elastic components have moderate or small memory requirements, a pool of paused containers may be kept, and a resume/pause container management strategy may introduce enough elasticity immediacy for dealing with this SLO without requiring any complex proactive performance model. This paper analyses an example of reactive elasticity management using those immediate scaling actions, providing encouraging results.

KEYWORDS: Elastic service, container management, elasticity management.

1 Introduction

Cloud providers that follow the *Platform as a Service* (PaaS) model [11] should automate the elasticity decisions for managing the services deployed in their platforms. To this end, the platform should contain resource monitors that provide information to a platform element that behaves as both a resource manager and activity scheduler that with either a proactive (i.e., predictive) or a reactive approach (or even a combination of both) should dynamically adapt the amount of instances of each service component to respect the compromised objectives stated in the *service level agreement* (SLA) [15]. One of the most challenging *service level objectives* (SLO) is response time since its value may depend on many parameters.

Proactive approaches [2] build a performance model for each deployed service and analyse the current input workload to predict the values for a selected set of performance indicators. If those performance indicators generate values that endanger SLA compliance, the service is adapted, reconfiguring it with the appropriate amount of resource instances to reach its intended performance values. On the other hand, reactive mechanisms [3, 13, 16, 20, 21] choose a given set of metrics that should be periodically assessed using several scaling rules. If those rules are satisfied, their associated scaling actions are undertaken. In the end, both approaches execute some actions in order to adapt the service, either scaling it in or out. The main problem is that those actions do not have an immediate effect, since either adding or stopping component instances demands a non-negligible time interval.

In order to minimise the time spent in those scaling actions, different kinds of hosting elements for each component instance have been recently proposed and compared [4, 14]. Thus, instead of only relying on hypervisors [19] that manage multiple *virtual machines* (VM) in each host computer, there are also lightweight container managers [18, 22] that are able to create, start, stop or remove their instances much faster than with VMs.

If those instance managing actions were fast enough, the performance models to be used in the adaptive approach could also be much simpler than they are when their actions should be applied onto sets of VMs. The goal of this paper is to check whether this hypothesis is true or not. To this end, Docker [12] has been chosen as the intended container manager and its configuration will be optimised trying to minimise the intervals demanded for activating new instances or deactivating exceeding instances when needed. Thus, our main goal is to test whether an almost immediate scale-in/scale-out mechanism will provide a *good* elasticity management. That mechanism may be based on container pausing/unpausing when service components have small memory and network-bandwidth requirements. In this scope, *good* means that it will only need a simple single-rule reactive strategy, instead of requiring an elaborate proactive performance model.

Our approach for minimising the interval being needed for resuming a server instance in case of scaling out a service (or, respectively, pause a server instance in case of scaling in) consists in managing a small pool of paused containers in each host. To this end, a broker agent has been implemented. It filters every request message sent by clients and every reply sent by servers. Thus, this broker knows and periodically reports to the elasticity manager the average response time on each server instance and the current trends of both client workload and server response time. With that information, the elasticity manager may command each scaling in or scaling out action in time, to ensure that the response time SLO is kept in the appropriate range.

However, several challenges exist in this resume/pause strategy, associated to the following questions: (1) Which is the most appropriate information reporting interval to be used by the broker agent?, (2) Once the minimal service time for a given set of operations is known (using to this end a minimal workload that does not saturate server instances), how close may the response time SLO be to that minimal service time? An experimental evaluation is presented in order to answer those questions, showing that the proposed approach is feasible and reasonable.

The rest of this paper is structured as follows. Section 2 describes the architecture of our elasticity management system. Section 3 presents the criteria being used in the scale-in and scale-out decisions. Section 4 explains the results of a series of experimental evaluations of this system, assessing the quality of its elasticity decisions and the benefits introduced by a resume/pause container mechanism when it is compared with the classical create/remove one. Section 5 discusses related work. Finally, Section 6 gives our conclusions.

2 System Architecture

Our system requires a manifesto for each service to be deployed in the platform. That manifesto states which are the service components and their interaction dependences. The platform has a hierarchical elasticity management consisting of two levels. The top level is a centralised *global elasticity manager* (GEM) that coordinates the deployment of every service component for all existing services. Besides it, there are multiple *local elasticity manager*s (LEM), one per host. Each LEM interacts with its local container manager, requesting the start, stop, resume or remove actions on the locally existing containers. It receives from GEM the response time SLO value to be respected by each deployed component and manages their scaling actions, considering their current workload. Thus, GEM knows which hosts have been used in order to deploy a given component. LEMs locally decide how many instances should be deployed for each component, reporting their variations to GEM as soon as they happen.

Every service component is in two or more hosts in order to guarantee their availability in case of host failures. Client requests are adequately forwarded to those hosts by other platform elements. On each host, there is a local broker per component that filters all client requests forwarded to that component. Note that this broker is inherently replicated, since its associated component is deployed in more than one host. In case of host failure, both the broker and all its associated component instances fail at once. So, there is no need of a broker replacement in those cases.

Once each request has been served, their corresponding replies are filtered by the broker. Thus, brokers deal with four complementary tasks: (1) load balancing among all local component instances, (2) computation of the response time of each filtered request/reply pair, (3) reporting of response time statistics to LEM, and (4) selection of the component instances to be stopped or removed in scale-in actions. When LEM requires a scale-in action, the broker chooses one of the free instances in order to destroy or pause it, depending on the configured operational strategy. If there is no free instance at that time, the broker chooses that with the lowest current workload and passes its ID to LEM once the latter has processed and replied all its currently assigned requests. When LEM gets such ID, it destroys or pauses that instance.

GEM manages system-wide scalability issues, using to this end LEMs as its supporting mechanism. This paper is focused on LEM management, in order to illustrate and discuss its convenience. Therefore, a single node is needed in order to assess such LEM convenience, since LEM manages a set of local instances that should respect a given response time goal.

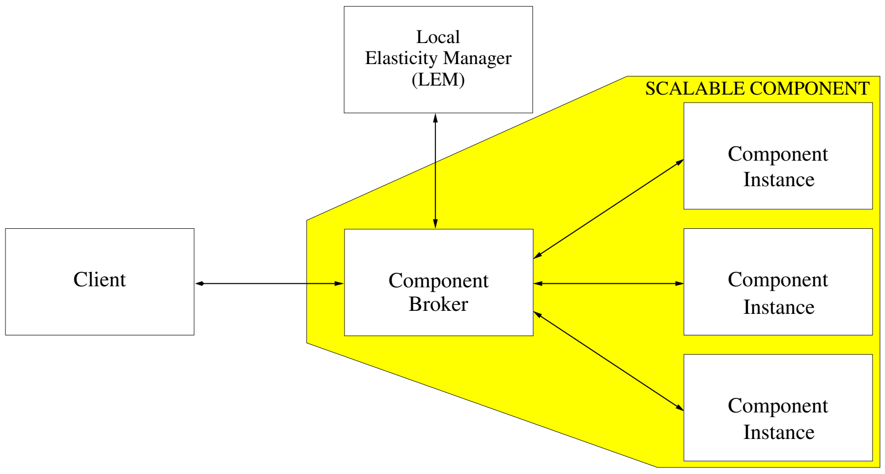


Figure 1: System architecture assuming a single service component.

The LEM-based elasticity management architecture is depicted in Figure 1, considering a single component. There is a unique LEM in each node that interacts with all deployed components. For each component, these elements exist:

- *Broker*. The broker deals with these tasks:
  - It routes incoming messages to the server component instances. We assume that all the interactions supported by a component follow a request-reply pattern. Both request and reply messages are filtered by this broker. Thus, the broker balances the load among the existing server instances. To this end, every instance –at its start– sends an initial CONNECT message to the broker. The broker keeps the instance identities in a queue of available instances.

    When client requests arrive, the broker dequeues the first available instance ID and forwards the request to it. Once the reply is sent to the client, the instance identity is enqueued again.

    On the other hand, if a request arrives and finds that the available queue is empty, this means that all instances are busy. If so, this incoming request is placed in a queue of pending requests. The oldest pending request is served when the next reply is processed by the broker.
  - Since both requests and replies pass through the broker, the broker monitors the response time of each request. Thus, it may collect as much information as needed about the average response time in each server instance, periodically reporting (by default, every one second) that information to LEM.

    The set of metrics to be collected by brokers is configurable. It initially consists of the following information:

1. Current number of component server instances in this host.
2. Service time of the latest 50 completed requests, expressed in nanoseconds.
3. Queuing time (in the broker incoming queues) of the latest 50 completed requests, in nanoseconds.
4. Request arrival rate (req/sec), in this last reporting period.
5. Current amount of request messages kept in the broker incoming queue.

All those parameters provide an accurate snapshot of the workload variability trends and the service capacity of that component.

- *Instances*. Components may have a variable amount of instances. Figure 1 depicts a scenario where the deployed component has three instances. That amount of instances is dynamically adjusted by LEM, depending on the current workload and the target response time specified in the SLA.

Each one of the elements shown in Figure 1 is deployed using a separate Docker container. We assume a dynamic resource allocation policy for containers. For instance, the amount of main memory reserved for each container may be varied at run time. Thus, paused containers use a low amount of resources that is later increased when needed.

When a new host is added, a default pool of containers is started and paused, using the images of the components that may run in that host.

3 Elasticity Management

As previous sections have outlined, our elasticity management strategy consists in using a pool of already created and paused containers. With it, LEM may resume one or more of those paused containers in case of applying a scale-out action or, respectively, it may pause one or more running containers in case of a scale-in action. Pause and resume actions have an immediate effect on containers. Therefore, there is no need of a stabilisation interval once any of those actions is applied. We refer to this strategy as "*resume/pause*".

Traditionally, in order to scale service components, a conservative "*create/destroy*" strategy is assumed. In it, each time a component instance needs to be started, a new container or VM is created using its component image as a base. That creation (with start) action needs a non-negligible time interval: fewer than 6 seconds in case of containers or several tens of seconds in case of virtual machines, depending on the host computer and the operating system and set of programs installed in the VM image. Because of these intervals, each scaling action needs a subsequent stabilisation interval in order to ensure that its effects are already noticeable before applying any other new scaling action.

We use by default the resume/pause strategy. With it, elasticity management consists in periodical interactions between the brokers of each deployed component and the LEM placed in their local host computer. This generates a *reactive elasticity management* (REM) algorithm that behaves as follows:

1. Periodically (once per second by default), each broker reports to LEM the collected data described in Section 2, i.e., current amount of instances (ci), service time of the latest 50 requests (st_{50}), queueing time of the latest 50 completed requests (qt_{50}), request arrival rate (λ), and current amount of enqueued pending requests (pr).

2. With those values, LEM computes the average service time for those 50 latest requests (st) and the average queueing time for them (qt).

3. If $qt = 0$, a scale-in action may be convenient. In order to assess this, Little's Law [9] is used for computing the target amount of instances (ti) to maintain:

$$ti = \left\lceil \lambda \times st \right\rceil \qquad (1)$$

4. Otherwise, Little's Law is used again for computing the convenience of a scale-out action. In this case, LEM also considers the current amount of pending requests and the intended maximum response time (mrt), i.e., the response time SLO:

$$ti = \left\lceil \lambda \times st + \frac{pr \times st}{mrt} \right\rceil \tag{2}$$

5. If $ti \neq ci$, then LEM asks the broker about the identity of $\mid ci - ti \mid$ instances to be paused or restarted. Once the broker returns that information, LEM commands the local container manager to pause or unpause those containers.

Note that no complex computation or performance model is needed in REM. Indeed, it only uses one of the classical laws of queueing theory.

3.1 GEM Overview

In order to complement this elasticity management description, a brief summary of the *global elasticity management* (GEM) is needed. GEM controls in which nodes each component is initially deployed. In order to avoid single points of failure, two nodes are used for initial deployment. To this end, the corresponding containers are created and its component image is started in order to be immediately paused. The number of created instances per node is static and set to N. The current amount of nodes where a component c has been deployed is represented by n_c. There are $n_c \times N$ created instances for component c. Let ac_c be the amount of currently active (i.e., created and not paused) instances of component c. Thus, $0 < ac_c \leq n_c \times N$.

The default global scale-in and scale-out reactive rules for managing every component are:

- *Scale-out rule*: When $ac_c > 0.8 \times n_c \times N$, then $n_c = n_c + 1$.
- *Scale-in rule*: When $N > 2 \wedge ac_c < 0.4 \times n_c \times N$, then $n_c = n_c - 1$.

The 0.8 (up) and 0.4 (down) thresholds may be dynamically adapted for each component, depending on its behaviour.

These rules define the intended size of the containter pool per component. Let us assume a component class with $N = 5$. Thus, initially, every component has $2N$ (i.e., 10) instances. It is scaled out to $3N$ (15) instances when there are at least $1.6N$ (8) instances in active state, and it will be scaled again to $4N$ (20) instances when there are at least $2.4N$ (12) instances active, and so on. Note that this 20% margin in order to raise the scale-out action is needed for choosing the appropriate new host computer (in the common case, a VM in an IaaS system), start a LEM and broker in it, download the component image in that host (if it is not present in the default container image to be used), start all those new N containers and pause them as soon as they send a registration message to the broker placed in that machine.

On the other hand, when we have already deployed $4N$ (20) instances, then we will destroy N of them when there are fewer than $1.6N$ (8) active instances. It is scaled in again to $2N$ (10) instances, when there are fewer than $1.2N$ (6) active instances. No other scale-in transition can be applied, since the minimal number of nodes to be used is 2.

Besides this default reactive strategy, GEM may use a proactive global strategy, e.g., one based on machine learning [15]. Thus, both strategies may be compared periodically, choosing that with the best behaviour.

4 Experimental Evaluation

In order to check the benefits that may be obtained from a resume/pause elasticity management strategy, a series of evaluating experiments has been carried out. This section describes the results of those tests and it is structured as follows. Section 4.1 describes the hosting environment used in those evaluations. Section 4.2 explains the service types used in these experiments. Section 4.3 outlines client behaviour. Section

4.4 summarises the results obtained in a monitoring overhead evaluation. Finally, Section 4.5 presents the results of our elasticity evaluation.

Note that we do not need realistic workloads. The most important problem to solve is a high and immediate increase in the workload intensity, demanding a large (and also immediate) scale-out action.

The results depicted in all figures shown in subsequent sections represent the averages obtained in a set of ten replays of each experiment.

4.1 System Configuration

The architecture described in Section 2 has been deployed and tested on multiple computers, using pools of containers directly managed by the host computer. In this way, no hypervisor overhead is added to that introduced by container management.

In order to run our experimental evaluation, a computer (let us call it PhC1) with these characteristics has been chosen: Intel Core i7-3632QM CPU (4 cores at 2.2 GHz with hyper-threading and a maximum frequency of 3.2 GHz), 8 GB of DDR3 RAM at 1600 MHz and a 500 GB S-ATA hard disk at 5400 rpm. Its operating system is a Linux kernel version 4.15.0 within an Ubuntu 18.04 LTS distribution. Containers are managed by Docker, version 18.06.1-ce. The programs run in these tests have been implemented in NodeJS, version 10.9.0, using version 4.2.0 of the ∅MQ asynchronous communication library. Each architecture element uses a separate Docker container, and all containers run in that same computer.

A physical computer like PhC1 should provide a more stable set of results than a VM from an IaaS cloud provider. In order to compare its performance with those VMs, a benchmarking tool may be used. Thus, we have run the open-source Phoronix Test Suite [17] in its version 7.8.0 to this end, since a set of results of that test suite on multiple Amazon EC2 VM types was reported in [8].

From such a test suite, the following tests have been run: `pts/openssl` (in signatures per second, more is better), `pts/compress-7zip` (in MIPS, more is better), and `pts/pybench` (total time in ms, less is better). The first two are intended for processor benchmarking while the last one for system benchmarking.

The achieved results for these tests, using their default number of trial runs, are:

Machine	openssl	compress-7zip	pybench
m5.large	242	5090	1559
m5.4xlarge	1860.93	36692	1637
PhC1	491.23	16587	1720

The Amazon EC2 m5.large and m5.4xlarge are general purpose VMs with the following characteristics:

Machine	vCPU	ECU	Memory (GiB)
m5.large	2	10	8
m5.4xlarge	16	61	64

As it has been stated above, PhC1 has the same amount of RAM than m5.large and 8 logical cores. Since PhC1 used a Core-i7 processor at 2.2 GHz, those 8 logical cores would be similar to 16 ECUs (EC2 Compute Unit [ECU] is the equivalent to one Xeon processor at 1.2 GHz from year 2007). Thus the results achieved by PhC1 are intermediate to those from m5.large and m5.4xlarge in regard to CPU performance (`pts/openssl` and `pts/compress-7zip` tests) and comparable to them in regard to overall system performance (i.e., `pts/pybench` test).

4.2 Service Types

A resume/pause strategy for managing containers allows a fast reactive elasticity management, but may introduce problems of resource exhaustion (e.g., if a static resource allocation policy was used, each paused container would require a large portion of the host main memory). Therefore, it is convenient to evaluate this strategy in different scenarios, depending on the type of server component:

- *Null server* (type 0): The component has no useful code nor any memory requirement. It returns immediately a reply, without doing any computation. This server is intended for assessing the overhead introduced by the response time monitoring machinery placed in the broker.
- *Light server* (type A): The component has very light CPU and memory requirements. This is the most favourable case for using this reactive strategy, since many paused instances may stay without tight constraints in the same host.

  An example of light server has been implemented, providing a single operation in its interface. That operation returns its result in 10 ms, but only demands 100 μs of CPU time in that interval; i.e., it only uses the processor in 1% of that interval. Although the regions of its virtual address space need about 1 GB of memory, its actual resident set size only demands 30 MB of memory. Thus, its memory demands are also very low.

  In case of using servers of this type, the elasticity management should focus on the time spent in each server reception queue by the incoming requests. That queuing time will comprise a large portion of the response time perceived by client agents.
- *Moderate server* (type B): This component example has moderate CPU and memory requirements. It also provides an interface consisting of a single operation. However, it uses an array of 20 million integers that is randomly filled with values in the 0..100 range at initialisation. Its public operation admits an integer argument, and looks for the amount of times that such value is contained in the array slots, returning that result.

  In this case, that operation needs a minimum of 34 ms of service time to be completed (considering the host computer described in Section 4.1 and no other competing process), and this component requires 1.4 GB for its virtual address space and 480 MB for its resident set size. This limits the amount of instances that may be simultaneously run in a given host, since each instance should receive at least 500 MB of memory in order to run without page thrashing.

  This is a stressing type of server regarding response time evaluation. It introduces quite a long interval of moderate CPU usage with continuous accesses to different addresses in main memory. In case of memory scarcity this immediately leads to page thrashing, highly increasing the overall service time of each request.

Type B servers introduce a significantly higher workload in containers than those of type A. Their processing demands are 340 times higher (34 ms vs 100 μs) and their main memory demands are 16 times bigger (480 MB vs 30 MB). However, their default service time is only 3.4 times higher (34 ms vs 10 ms). This implies that multiple instances of a service of type B may easily stress a host machine regarding its CPU and main memory requirements, while a similar amount of service instances of type A do not introduce any significant workload in both regards. Additionally, those server programs have been intentionally implemented as single-threaded processes and they do not start the processing of a new request until they have replied the current one.

4.3 Client Behaviour

The client processes used in all these experiments run very short NodeJS programs and are I/O-bound. They have a resident set size of 35 MB, but 25 MB of that set belong to shared libraries. They use a request-reply synchronous communication pattern in order to interact with the service broker and they are run in the same host than the assessed components.

We have deployed the clients in that host in order to introduce a controllable overhead on that host machine, emulating the overhead that may be found when all these components are deployed on a VM in an IaaS system [14]. However, the overhead to be found in a real IaaS system cannot be controlled nor precisely bound, since it depends on the workload being managed in the physical computer where such VM is deployed.

4.4 Monitoring Overhead

Section 2 has explained that the broker element in each deployed component monitors a set of metrics and reports them to LEM. Let us evaluate now which overhead is introduced in this filtering of every incoming request and outgoing reply. To this end a configuration with a "raw" broker (i.e., one that simply forwards messages, without collecting any metric) is compared with a fully functional monitoring broker. The broker uses its default reporting period; i.e., 1 second. Since the communication channel between a broker and LEM follows an asynchronous pattern, no pause nor significant delay is introduced in the broker process for sending its metric-report messages.

In this use case, the deployed component is of type 0. It immediately replies every request. Both request and reply messages carry a short constant string (5 characters). Our goal is to figure out which delay is introduced by the broker onto the fastest possible request-reply pattern. Thus, when this pattern uses the raw broker, it provides the minimal round-trip time, and it will be compared against the round-trip time generated by a monitoring and reporting broker.

The workload is generated by synchronous clients that start a sequence of 200000 requests each one.

Figure 2 shows how that overhead evolves depending on the number of server instances. The graphic shows the average round-trip time per request. There are as many client processes as server instances. The overhead ranges from 7.4% (291.39 μs on a monitoring broker vs. 271.27 μs on a raw broker) with 1 client to 22.8% (732.36 μs vs. 596.20 μs) with 8 clients. In any case, this overhead is directly proportional to the number of clients; i.e., to their generated workload: close to 20 μs per client.

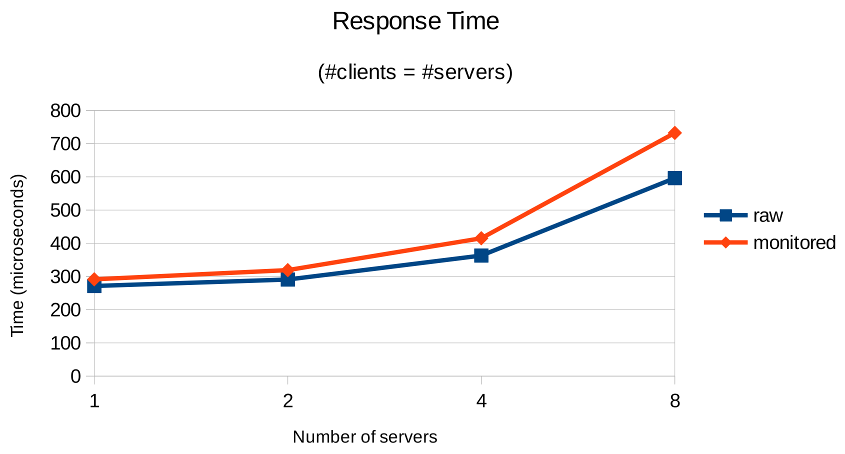


Figure 2: Overhead introduced by a monitoring broker.

This overhead directly depends on the amount of requests, since each request compels the broker to execute short code fragments in order to collect several metrics. The host machine has a processor with four cores and hyper-threading. The Linux kernel reports eight cores in that host. Thus, a configuration with up to four server instances, the broker and one client does not use yet more processes than processing cores exist. In those configurations, process switches and broker-server ROUTER connection management do not introduce any noticeable overhead for the raw broker. Indeed, its response time remains constant in that range. Figure 3 shows configurations with a single client and different numbers of server instances. In them, the measured overhead does not depend heavily on the amount of server instances.

The round-trip time using a raw broker varies 1.17% from its configuration with 1 server (271.27 μs) to the configuration with 8 servers (274.44 μs), and 3.83% to the configuration with 16 servers (281.64 μs). A similar behaviour is shown using the monitoring broker: 0.87% of variation in the 1 to 8 range (293.94 μs vs. 291.39 μs) and 2.3% in the 1 to 16 range (298.12 μs vs. 291.39 μs). Note that the workload in these examples is heavily I/O-bound. Indeed, using the raw broker, the global CPU utilisation in the host is in the [8.5,9.5] percentage range for user time, [3.5,4.5] for system time and around 87% idle, while the configurations with a monitoring broker set a CPU utilisation 2 units greater for user time, equal for

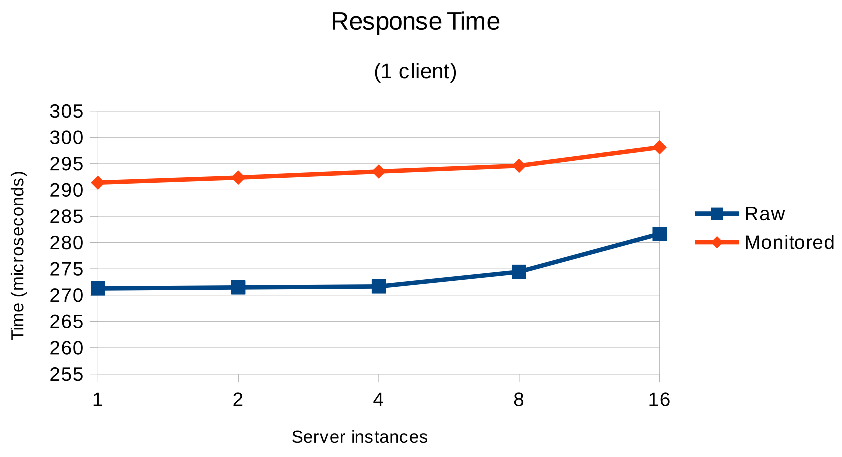


Figure 3: Monitoring broker overhead. Constant workload.

system time and 85% idle. Thus, when the raw broker is used, its demanded CPU intervals are very short, leading immediately to the subsequent I/O operation: to remain suspended waiting for the next message reception. This leads to frequent context switches when eight or more server instances are used. This explains the increasing response time trend in the workloads managed by the raw broker. On the other hand, the monitoring broker has longer CPU intervals, since its message forwarding and accounting tasks require a longer processing time. Due to this, context switches do not heavily impact its response time curve in Figure 3.

Comparing the monitoring broker with the raw broker, its overhead simply adds around 20 μs in the assessed configurations, and it is close to 7% in those cases. Note that this 7% overhead is the worst possible case, since the server immediately returns the answer in the assessed scenario. When the operation to be served by the invoked component instance has a non-negligible processing time, those 20 μs of overhead will generally represent less than 1% of the global round-trip time.

4.5 Elasticity Evaluation

Let us evaluate the elasticity techniques described in Section 3. To this end, Section 4.5.1 shows a direct comparison between two operational strategies: create/destroy and resume/pause. The goal in all these experiments is that the response time measured by the broker element for each client request never exceeds 800 ms. The actual server processing time for each request is around 10 ms, using to this end the *light server* implementation described in Section 2. With this, the workload is I/O-bound (i.e., communication-oriented) and the response time perceived by clients depends heavily on the communication and queueing times. Subsequently, Section 4.5.2 refines that evaluation using smaller SLO limits and assessing different elasticity evaluation intervals until a good set of results is found. Finally, Section 4.5.3 assesses the behaviour of the resume/pause operational strategy when *type-B* servers are used.

4.5.1 Basic Evaluation

Let us compare two alternatives in the operational strategy. To this end, a service execution consisting of the following steps is used:

1. At time 0 seconds, the system is started with a minimal configuration composed by LEM, one reporting broker (since only one service is deployed), one server instance and one client process.

2. A loop of four workload increasing iterations is started. In each iteration:

    - A pause of twenty seconds is done.

- Three additional client instances are started. Thus, once the last iteration completes, there are 13 clients.

3. Twenty seconds later, 12 clients are stopped and removed. With this, a single client remains in the system.
4. Fifteen seconds later, all system agents are stopped. The stopping order is: client first, followed by any remaining server instances and, finally, the broker.

These stages are driven by a shell script, in a synchronous way. The exact time at which each stage is initiated depends on the time (Δ) needed for serving the commands run in the previous stage. This means that step 1 is started at time 0, step 2.1 at time 20+Δ seconds, step 2.2 at time 40+3Δ seconds, step 2.3 at time 60+4Δ seconds, step 2.4 at time 80+5Δ seconds, step 3 at time 100+6Δ seconds and, finally, step 4 at time 115+6Δ seconds.

Each client sends a request every 15 ms. This introduces a constant workload per client that is slightly lighter than what a server instance may process. Thus, each server instance should be able to deal with at least one client.

The goal of this service execution is to stress the adaptation capability of the platform elasticity management strategy. Thus, the first workload increasing iteration (initiated at time 20 sec) sets a 300% increase on the workload (from 1 to 4 clients), while the second iteration only increases a moderate 75% (from 4 to 7 clients). Those two iterations are the most difficult to handle.

The first strategy (*create/destroy*) consists in using the Docker Compose `scale` command for specifying in each scaling decision how many server instances should be used from that time on. This means that a given amount of containers should be created and started (in case of a scale-out decision) or stopped and removed (in case of a scale-in). Those two kinds of operations demand quite a long time in a system like that described in Section 4.1.

Considering that scenario, the curve defined by the client and server instances is depicted in Figure 4.

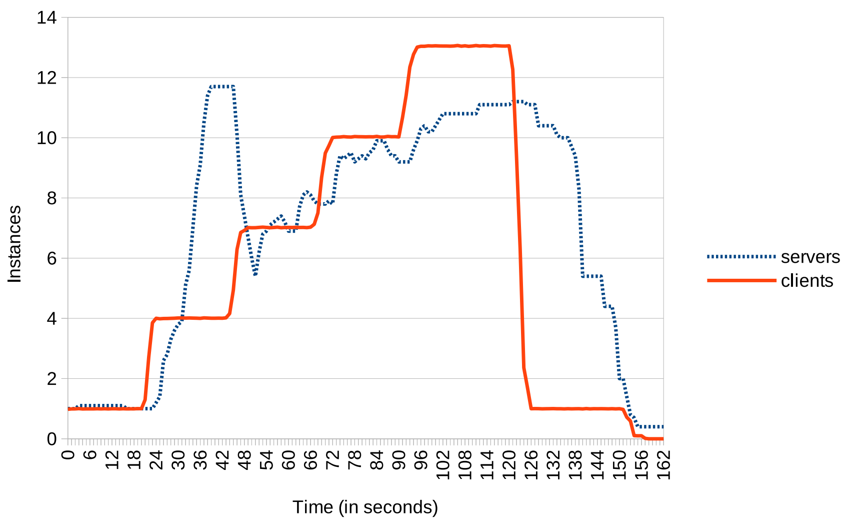


Figure 4: Execution with the create/destroy operational strategy: instances vs. time.

As we can see in the figure, Δ is quite close to 6 s in this set of executions. For instance, step 4 is started at time 151 s, and it is scheduled at time 115+6Δ s. This means that $\Delta = \frac{151-115}{6} = \frac{36}{6} = 6$ s. Although the broker reporting period is 1 second, the LEM decisions took at least 4 s to have any initial effect (in case of scaling out, exceeding 8 s in case of scaling in). Because of this, the first scale-out decision is conditioned by those too long reaction times. Thus, at time 23 s, the amount of clients has reached four units, but the amount of running server instances is still 1. Indeed, on average, at that time there will be more than 200 requests in the reception queue of the broker, and the average queueing time there already exceeds 800 ms (the compromised SLO for the request response time). To counteract this fact, LEM starts 10 additional server instances, but it is too late and the SLA has already been violated. Those instances do not become fully operational till time 38 s, cleaning the incoming queue at that time. Once that queue is clean, the amount of server instances is correctly adapted to the incoming workload until the penultimate execution stage is reached. At that point, 12 clients are removed from the system and, again, the create/destroy operational approach needs more than 20 s for adequately removing those exceeding servers.

The second strategy (*resume/pause*) consists in generating a pool of started server instances when the system is initiated. Those pre-started instances are immediately paused and kept in that state till they are needed. All this management is applied in the first stage of the evaluated executions. Later on, when an instance is needed, an `unpause` action is requested. When this second strategy is used, the curve of client and server instances is depicted in Figure 5.

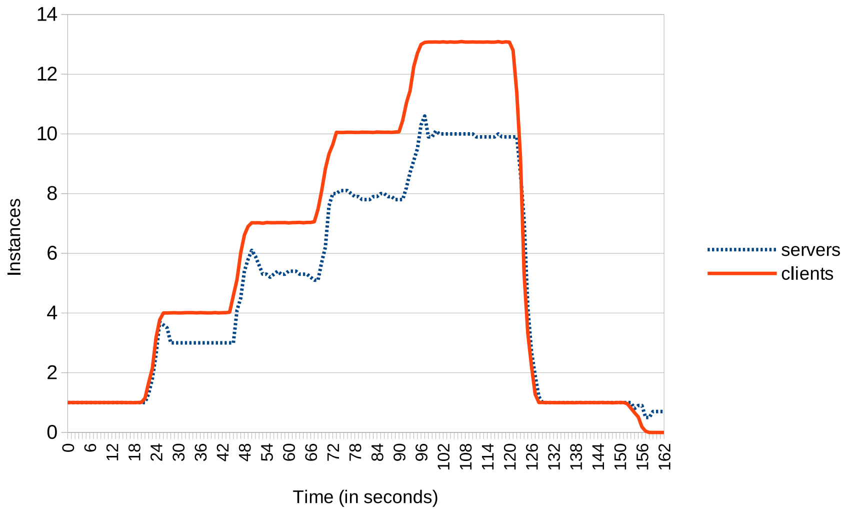


Figure 5: Execution with the resume/pause operational strategy: instances vs. time.

This second strategy has an almost-zero reaction time. With this, the elasticity manager is able to take immediately its scale-out actions. For instance, clients start their transition from 10 to 13 instances at time 76 sec, completing it at time 79 sec, and the server instances start their transition from 8 to 10 instances at time 77 and complete it at time 80 sec; i.e., with a minimal delay equal to the reporting interval. In all this transition, the average queueing time does not exceed 52 ms.

Once these two figures have been analysed, it seems interesting to compare the response times of these two operational strategies. Figure 6 shows them.

The execution has four workload increasing transitions. Each one adds three client instances. The one most difficult to handle is the first one, since it implies an increase of 300% in the workload to be supported by the deployed service. The create/destroy operational strategy is unable to handle that increase. Its response time surpasses 2515 ms at time 28 s and exceeds 800 ms (the SLO) from time 23 to 32 s, inclusive. The reason for this has already been explained before: its reaction time is too long, and many requests should wait for a long interval in the incoming broker queue.

The other three subsequent workload increasing transitions do not introduce any problem. They do not represent an important workload increase percentage (75% at most), and the increase in response time is proportional to that percentage.

Regarding the second strategy, all those transitions have been handled without problems. The largest response time happens at time 24 s and only reaches 243 ms, quite far from the upper limit (800 ms). It corresponds, as in the other strategy, to the first increasing transition. In spite of this, it is worth noting that the data depicted in Figure 6 corresponds to the average results, per second, from the data collected in ten different executions. Therefore, it is possible that there are values larger than such averages. Revising the

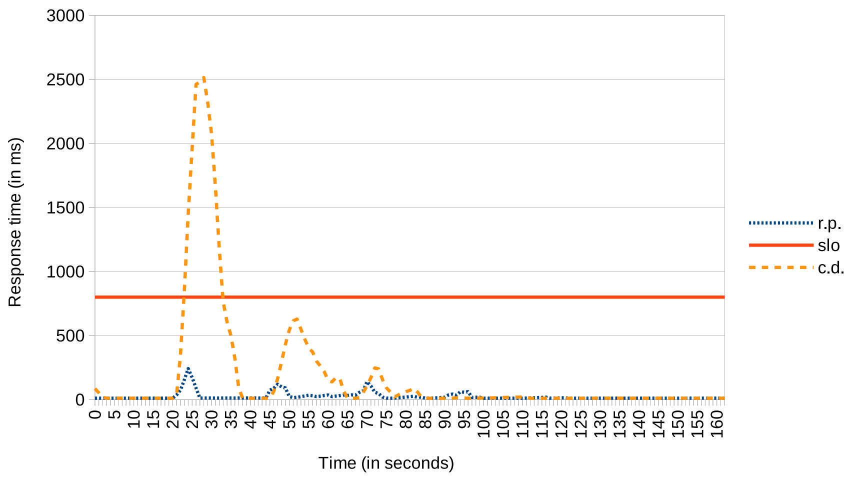


Figure 6: Average response times of the operational strategies.

data for each execution, the maximum response time happened at time 25 s in one of those executions, and its value is 523.64 ms, more than twice larger than the average, but without surpassing the target maximum of 800 ms.

Let us see on the following how a static activation frequency (with a period of 1 sec) is able to deal with tighter SLOs. Since the create/destroy strategy has been unable to manage a very loose SLO target of 800 ms, it will not be considered in subsequent analyses. Its main problem is its too long reaction time, needing at least 4 s to apply any scaling action. When the workload may vary rapidly and with large variations, those slow scaling approaches may only work as intended when they are combined with accurate proactive elasticity strategies.

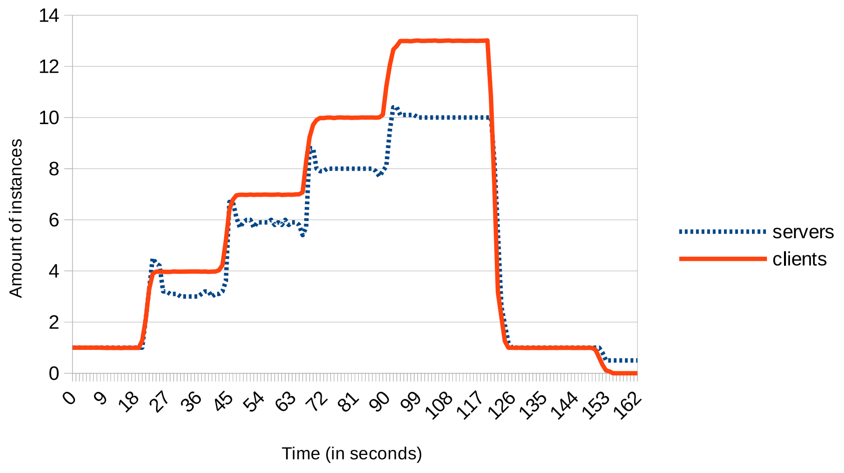


Figure 7: Execution with a resume/pause strategy and 200 ms SLO: instances vs. time.

4.5.2 Tight SLO Evaluation

In order to deal with increasing workloads, Equation 2 considers both the current request arrival rate (that has been collected in the last reported interval) and the current amount of pending requests in the broker reception queue. The results of Equation 2 are appropriate when the increases in the broker reception queue length are detected and adjusted as soon as possible. In Section 4.5.1 the reporting period (1 second) was only 25% longer than the maximum admitted response time (800 ms). Let us see what happens in other cases.

To begin with, Figure 6 has shown that the average response time achieved by the resume/pause strategy for the first and largest workload increase (from 1 to 4 clients, generating each one around 65 req/sec) is 243 ms. Let us see how Equation 2 behaves, combined with the resume/pause operational variant, if the SLO is set to 200 ms and the activation period is 1 second.

Thus, Figure 7 shows a slightly different behaviour when it is compared with Figure 5. Both figures show how the amount of server instances varies when the incoming workload is increased. Both configurations stabilise in the same amount of server instances. Thus, when there are 4 clients, 3 servers are needed; with 7 clients, almost 6 servers are used; with 10 clients, 8 servers, and with 13 clients, 10 servers need to run. However, with a loose SLO (Figure 5) at most one additional server is started and used for cleaning the reception queue, before stabilising the set of server instances, while with a tight SLO (Figure 7) two additional server instances need to be started in order to process all enqueued requests in the first workload increase. This suggests that such queue of pending requests may become too large with tight SLOs, endangering SLA compliance.

Figure 8 confirms that suspicion. Although the average response time is much lower than the SLO limit, the maximal response times observed in these executions clearly surpass the limit.

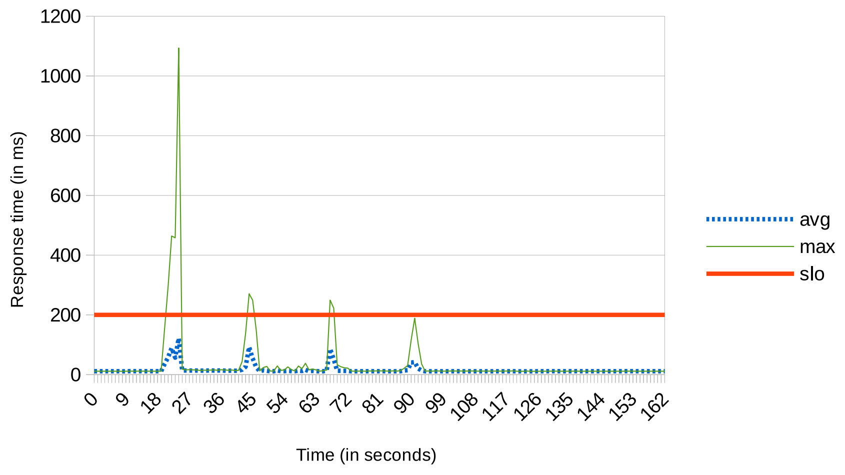


Figure 8: Response times for a resume/pause strategy with 200 ms SLO.

In the first transition the average response time reaches a value of 121.7 ms, while the second transition leads to 92.6 ms, the third to 86.1 ms and the last to 41.8 ms. These values are quite far from the 200 ms limit stated in the SLA. In spite of this, the maximum response time values reach 1093 ms in the first workload increase, 270.5 ms in the second, 249.4 ms in the third and 188.5 ms in the last one.

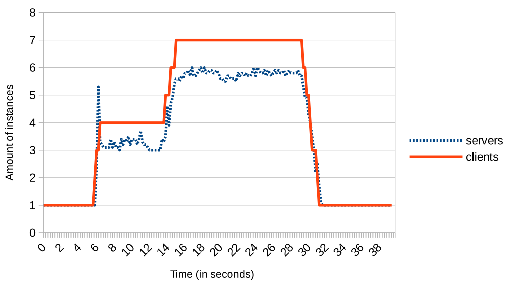
These results show that a long static evaluation period is not appropriate for managing tight SLO requirements, specially when both the broker-report interval is much longer than the SLA response time limit. A first solution to this problem may consist in adjusting that period to the SLO limit.

Thus, let us analyse how close should the evaluation period be to the maximal response time stated in the SLA. To this end, the SLO limit is set now at 50 ms and the execution presented in Section 4.5.1 is adapted using shorter pauses (5 sec. instead of 20 sec.) and using only their two first workload increases (those that caused SLO violations with the largest response times, as depicted in Figure 8). With these requirements, SLA violations should be easier to generate than in our previous evaluations.

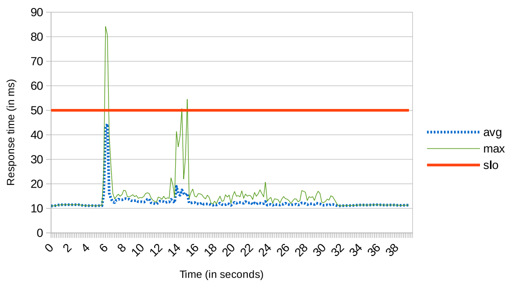
To begin with, the broker-report interval will be set to a length four times longer than the SLO limit. With this, some SLA violations will occur. Later on, those intervals will be shortened until no violation happens. Then, some conclusion could be got from those final intervals.

Therefore, the first case to be studied consists in intervals of 200 ms combined with a SLO limit of

50 ms. Figure 9 shows the amount of instances (a) and service response time (b), respectively, in that case. Although the averages shown in Figure 9.b do not exceed the SLO limit, the maximal response times registered in those ten executions raised SLA violations in the first workload increase in several of the assessed executions. This suggests, as expected, that shorter reporting and evaluation intervals are needed to correctly manage a SLO like this.



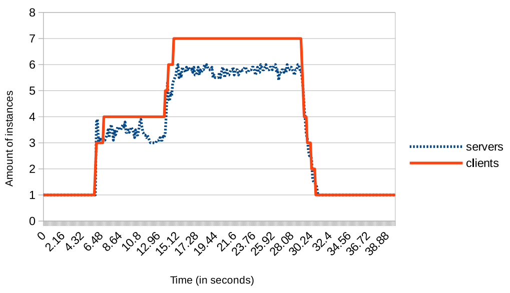
(a)



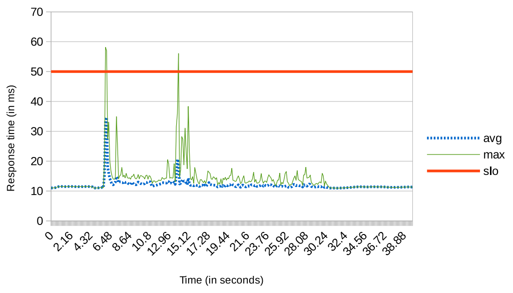
(b)

Figure 9: Reporting period of 200 ms and SLO=50 ms: (a) instances and (b) response time.

Thus, a second configuration is tested. In this case, the interval lasts 120 ms, with the same SLO limit (50 ms). The results are shown in Figure 10. When Figures 9.a and 10.a are compared, the first workload increase needs initially 5 server instances to be managed with an interval of 200 ms (Figure 9.a). This is due to a fast and large increase in the incoming request queue length, because of its too long monitoring interval. This causes a large number of SLA violations when the broker-report intervals is set to 200 ms. In Figure 10.a, that interval has been shortened to a bit more than a half of their original length. With this, only 4 server instances have been used. Regarding the average response times shown in Figure 10.b, now their values are below the SLO limit; reaching 34.7 ms in the first workload increase and 20.6 ms in the second one. In spite of this, there are still a few SLA violations in the assessed set of 10 executions, and the maximal response time still surpasses 50 ms. So, the shorter the intervals are, the better the response times will be.



(a)

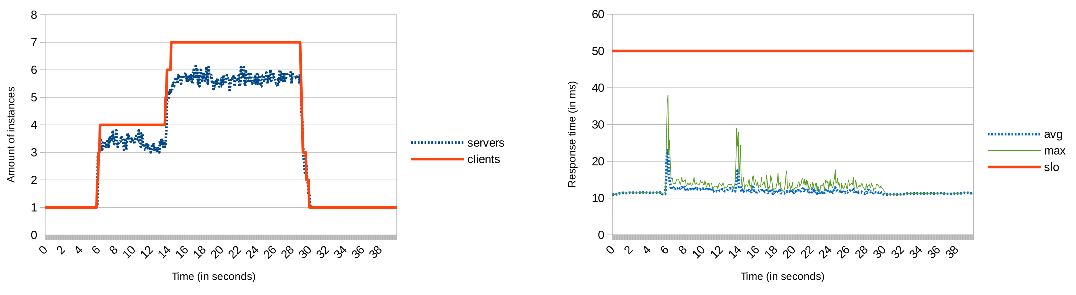


(b)

Figure 10: Reporting period of 120ms and SLO=50ms: (a) instances and (b) response time.

A new configuration is still needed. Now, the intervals are set to 80 ms and its results are depicted in Figure 11. In this case, no SLA violation happens in the set of these ten executions. Indeed, the maximum response time occurs in the first workload increase, as expected, but it only reaches a value of 38 ms; i.e., 12 ms lower than the SLO limit. Therefore, it seems that a reporting period set to a value 60% greater than the intended limit (since 80 ms is 60% larger than 50 ms) may be a good configuration strategy. In case of problems, the length of these periods could be shortened again, until a larger margin between the actual response time and the intended limit is obtained.

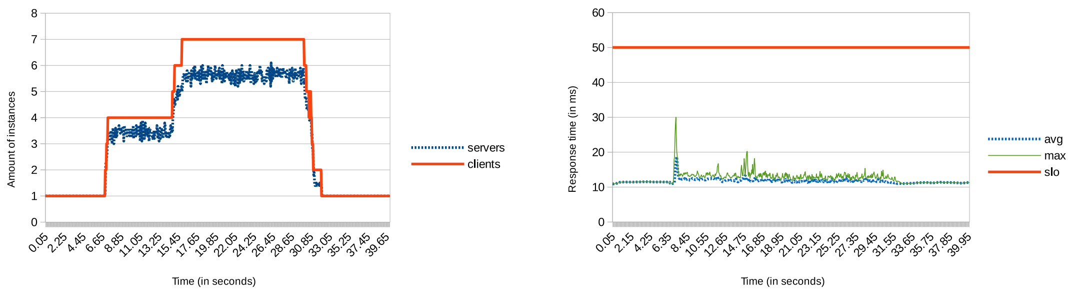
Following that advice, Figure 12 shows what happens when the broker-report period is identical to the SLO limit (i.e., 50 ms in this example). No SLA violation happens in those executions, of course, but we



(a) (b)

Figure 11: Reporting period of 80ms and SLO=50ms: (a) instances and (b) response time.

should take care of how the maximal response time has decreased when it is compared with an 80 ms-period. Now, the average response time is a bit lower: it does not surpass 18.7 ms. Moreover, the maximal response time registered in all these ten executions is now 30 ms (8 ms lower than the value obtained with a reporting period of 80 ms). Therefore, this confirms that shorter reporting periods provide more accurate elasticity managements.



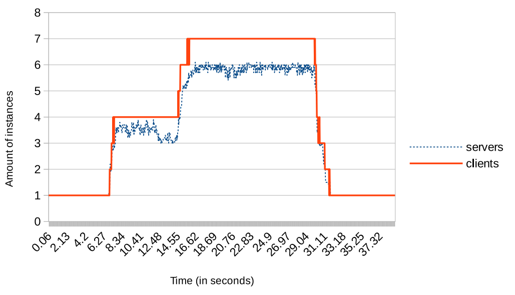
(a) (b)

Figure 12: Reporting period of 50ms and SLO=50ms: (a) instances and (b) response time.

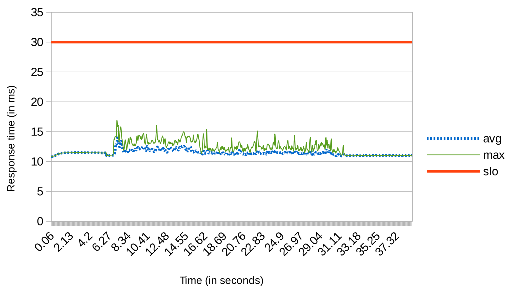
Indeed, Figure 13 does the same assuming a SLO limit of 30 ms. As in the previous case (50 ms), no SLA violation has occurred in these executions. The maximal response time observed in all those executions is 26.9 ms, and the average response time does not surpass 17.6 ms. These results encourage additional runs with lower SLO limits. However, periods lower than 30 ms are very hard to manage with a standard Docker deployment. With those short intervals, it is difficult to compute the request arrival rate with enough accuracy in the first workload increasing transition, since its arrival rate is too low. Thus, the lowest practical limit for the reporting period is 30 ms in the host system configuration described in Section 4.1.

Once these results have been presented, and looking at part (b) in Figures 9 to 13, it can be realised that the maximal response time observed for this type-A server application has never exceeded 60% of the reporting and evaluation periods, when they are set to the same length. This means, for that server application, that with a broker-report period set to P, the SLO to be assured may have a minimum value of 0.6P. However, this is not a general rule to be applied to every server application to be deployed in our platform. Each application should be carefully characterised and assessed in a preliminary stage in order to compute their appropriate reporting and evaluation periods.

In spite of this, for non-CPU-bound applications, these results confirm the hypothesis suggested in Section 1: with an almost immediate mechanism for applying scaling actions, a reactive strategy minimises the amount of SLA violations and it does not need any complex performance model to take its decisions. Indeed, for the simple type-A service, a regular container management strategy based on classical create/destroy operations was not able to support a very loose SLO requirement (800 ms of maximal response time, for operations requiring 11 ms of execution), while the management based on resume/pause oper-



(a)



(b)

Figure 13: Reporting period of 30ms and SLO=30ms: (a) instances and (b) response time.

ations has been able to manage tight SLO requirements (20 ms of maximal response time for the same operations). In spite of this, these first results should be still complemented considering other factors. Thus, other kinds of operations (e.g., CPU-intensive) and services (e.g., much more memory demanding) should be analysed for evaluating the adequacy of this reactive strategy based on container resume/pause actions. To this end, type-B servers are used in the next section.

4.5.3 Dealing with Moderate Memory Requirements

Type-B server instances introduce several problems for an operational strategy based on resume/pause container operations. To begin with, they require more memory: at least 500 MB should be assigned to each server instance in order to avoid page swapping. Indeed, 800 MB are recommended for achieving stable process behaviour. This means that only 10 server instances may run simultaneously in the host computer described in Section 4.1. Besides this, those instances are also CPU-bound. This means that the higher the amount of instances, the higher the response time will be, since all instances compete for the same set of CPU cores.

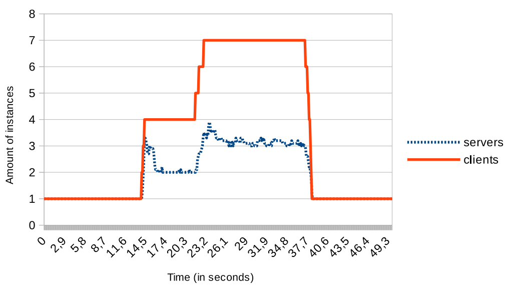
Let us start our analysis with a moderate set of requirements:

- The SLO limit is placed in a response time of 100 ms. It is an apparently light requirement, since a response time of 35 ms may be obtained when a single client and a single server instance run in that host computer.
- The broker reporting interval is set at 100 ms, too. As we have seen in Section 4.5.2, those values should be appropriate for managing light to moderate workloads without problems.
- Each client has a sending period identical to the SLO limit (100 ms), generating a sending rate of 10 req/sec.

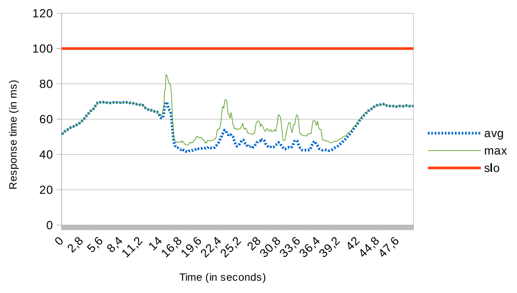
The results obtained with that configuration are depicted in Figure 14. Only four server instances are required in the worst cases; i.e., when workload is increased because the amount of clients is also increased. The average response time varies from 40 to 70 ms, but the maximum response time reaches 85.27 ms in the first workload increase. At that time, the amount of server instances transits from an initial single instance to, momentarily, four instances. Those four instances are needed for clearing the queue of pending requests that is expanding in the broker. But those instances should compete for the scarce host processing cores. In the end, this is translated into a high response time peak.

That response time peak will be higher if clients introduce a slightly higher workload. Let us see what happens in that case. To this end, the client sending period is changed from 100 ms to 90 ms, providing a sending rate of 11.1 req/sec. The new results are depicted in Figure 15.

This increase in the sending rate of clients (11%) has demanded a transition from 1 to 5 server instances when the amount of clients is increased from 1 to 4 instances, and from 3 to 6 server instances in the second client transition (from 4 to 7 clients). This has introduced a high increase in the average and maximal response times in such first client transition, breaking the SLO limit. Indeed, in this set of executions the

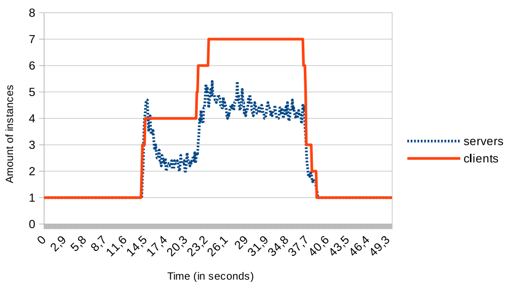


(a)

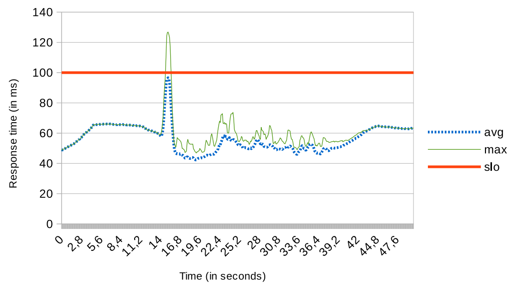


(b)

Figure 14: Type-B server, SLO=100ms, sending rate=10 req/sec: (a) instances and (b) response time.



(a)

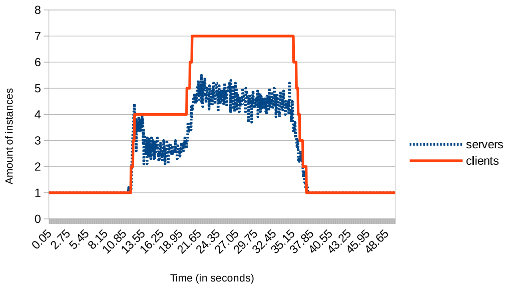


(b)

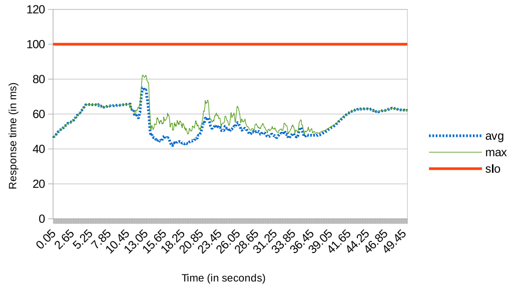
Figure 15: Type-B server, SLO=100ms, sending rate=11.1 req/sec: (a) instances and (b) response time.

maximum response time reaches 126.83 ms and the average response time at that same point is 96.64 ms. Therefore, those initial transitions (from 1 client to a higher amount) may introduce serious problems when server instances are CPU-bound with moderate memory requirements. Fortunately, the second transition has been adequately managed. Both Figures 14 and 15 show that in that case the average response time does not reach 60 ms and the maximum response time is around 70 ms, clearly lower than the 100 ms limit.

Therefore, some management variation is needed in order to adequately deal with this slightly heavier workload (11.1 req/sec per client vs. 10 req/sec per client). The solution to this problem may be found with a careful revision of the experiments made with type-A servers. In those cases, when the reporting period was much longer than the SLO limit, as it is shown in Figure 9, the first workload increase was being detected too late, placing many pending requests in the broker incoming queue. This also generated SLO limit violations. The solution in that case consisted in using shorter reporting periods. Let us try that solution here. Thus, instead of using a reporting period identical to the SLO limit, we will use reporting periods of 50 ms; i.e., a half of the SLO limit.



(a)



(b)

Figure 16: Type-B server, as in Figure 15 but with 50 ms as its reporting period.

Figure 16 shows the new results. As it is shown, shorter reporting periods are able to adequately manage this heavier load. This confirms that a pausing/restarting strategy for elastic container management may be able to handle a large variety of workloads with a minimal (even none) workload modelling effort.

4.5.4 Summary

Section 4.5.1 has shown that a resume/pause policy may manage without problems sudden high workload increases (e.g., a 4-times increase) while a create/destroy policy cannot manage those workload variations. Later, Sections 4.5.2 and 4.5.3 have shown that when the resume/pause policy is used, the proposed elasticity management behaves adequately using default reporting periods equal to the SLO target. However, when the SLO to be respected is too close to the minimal service response time, the unique existing solution consists in further reducing those periods. But that strategy has its limits. Thus, in the assessed deployment (i.e., depending on the physical characteristics of the host computer) those periods should not be shorter than 30 ms.

Section 4.4 has shown that the overhead introduced by this elasticity management does not depend on the reporting frequency, but only on the workload introduced by clients. Indeed, global overhead directly, and linearly, depends on the request arrival rate.

Taking those facts into account, a general policy could be set, consisting of these aspects:

- GEM should consider the response time SLO limit for each deployed service. It should initialise the reporting period of each service to its specific SLO limit.
- In regular cases, no SLO violation will happen using that default reporting period value. However, if any violation arises, that period will be shortened. Each shortening action should reduce a 20% of the period length being used up to that point.
- A minimal length for that period should be empirically found, assessing in a pre-production phase the service to be deployed in a host computer identical to that to be used at the production stage. That minimal period length should be respected. If a shorter period seems to be needed, this means that the set of server instances to be deployed should use a larger amount of host computers. In the latter, the intended number of containers will find their demanded resources without endangering their SLO limit.

5 Related Work

Response time SLO management is a challenging objective and has been the target of other previous papers. Let us review the approaches used in that related work and the results they obtained.

To begin with, Iqbal et al. (2011) [6] describe a VM-based system where reactive scale-out techniques are combined with predictive scale-in ones. That system is able to manage a two-tier Web service, appropriately scaling their instances and using a minimal amount of deployed instances without compromising response time SLO compliance. Scale-out decisions are based on setting rules for CPU utilisation levels, while scale-in decisions are based on a regression model that considers different configurable log scanning intervals (30, 60 or 120 seconds). Their experimental results show that most scaling decisions are accurate and that the SLA violations only affect to 3.5% of client operations in the worst case. Those results are very good, but they correspond to two components that require a joint service time lower than 50 ms with a response time SLO of 1000 ms. Our technique may deal with tighter constraints.

Virtual machines may become slow in order to apply scaling actions. Because of this, Han et al. (2012) [5] propose a mechanism for adapting the amount of resources (i.e., CPU share, memory size and I/O bandwidth) assigned to each VM in a first scaling stage. That strategy may be complemented with create-VM or destroy-VM actions in subsequent stages. The resulting LS algorithm provides some of the benefits previously mentioned for container-based elasticity management. Thus, the main advantage of VM resource adaptation is that it only needs a few milliseconds to be completed, while create-VM or destroy-VM actions need from 60 to 120 seconds in the system analysed in [5]. Thus, compared with the PBS (policy-based scaling) and TDS (tier-dividing scaling) VM-based scaling algorithms, LS minimises both the amount of server instances, scaling time and provision costs. In spite of this, the experimental evaluation presented in [5] sets an upper bound on response time to 2 seconds, and such threshold is temporarily exceeded (reaching about 3.3 seconds) when the workload increases either 50%, 100% or 200%. In those scenarios, it takes at least 100 seconds to return such response time to values that comply with the SLA.

Later on, Jayathilaka et al.(2015) [7] propose the Cerebro system. Cerebro combines off-line static code and application control structure analysis with on-line performance monitoring in order to predict and assess upper bounds on the application response time. Thus, they use a hybrid approach: code analysis gives a prediction, while performance monitoring generates time series that both adjust the prediction and assess its quality, allowing the system to react when the current prediction would lead to a SLA violation. In spite of that possibility of reaction, the authors only evaluate in [7] the quality of their off-line predictions. Their goal is to use those results for setting the response time threshold in the SLA of services that should be deployed in that system. However, response time is not the main SLA metric to consider in order to manage service elasticity decisions. Those decisions depend on other metrics. Indeed, the off-line analysis considers a very light workload (1 req/min) to find that threshold. Thus, this system example confirms that accurately managing response time SLOs, using a reactive strategy, is a challenging task.

On the other hand, Abranches and Solis (2016) [1] actually use a reactive policy. They propose a platform architecture called PAS (*PID-based AutoScaler*) to assess response time SLO compliance. As in our case, a load balancer (equivalent to our service broker) is the key element in that architecture, although it consists of many other elements:

1. The load balancer itself is implemented taking HAProxy as its basis. However, this load balancer writes its reporting information in a log that is later processed by other components.

2. Flume sends the load balancer logs, in text format, to Apache Spark.

3. Apache Spark, using its Spark Streaming API, is utilised for processing those logs, filtering and timestamping the logged elements that are needed by the Request Monitor, building in this way a time series.

4. Such time series is temporarily held in an in-memory Redis database.

5. That data is read by the PID Auto Scaler (PAS) component, equivalent to our elasticity manager, that computes the intended actions to be sent periodically as commands to the Kubernetes orchestrator. The default commanding period is 10 seconds in this system.

6. Finally, Kubernetes applies the intended horizontal scalability commands onto its managed Docker containers.

PAS is more general than our proposal, since the former manages a cluster of hosts, while the latter only administers a single host. The overall architecture is quite similar in both systems and it is based on the same principles: a load balancer (or broker) is used for distributing the incoming workload onto all server instances, registering the service time for each request. However, our solution is light-weight compared with PAS since our system does not need so many components in order to provide information to the elasticity manager. In spite of this, Abranches and Solis are able to obtain results that are much better than the default Kubernetes HPA (*horizontal POD autoscaler*): (a) PAS response times are lower than those obtained with HPA, and (b) PAS is able to manage the same workloads with less than a half of the containers needed by HPA.

Unfortunately, PAS cannot avoid SLO violations when the workload is highly increased at the beginning of an execution. For instance, multiple experiments are presented in [1] with a target response time of 50 ms. Those executions last 1000 seconds, and the response time is finally stabilised below 50 ms, as intended, but it took a bit more than 200 seconds to reach that stabilisation point and, in the meantime, the average response time exceeded 150 ms in one of the experiments and 250 ms in the other. Those violations happened at the start of the execution, when the appropriate amount of needed containers were started. But it took too long to achieve stabilisation (indeed, 200 seconds, i.e., the length of the time series used in their model), and this means that almost 20% of the incoming requests violated its SLA in those execution examples.

We have solved that same problem relying on paused containers. Their resumption is easy and fast. With this, high workload increases can be adequately managed, without needing any predictive performance model. However, this is not for free, since those paused containers have received a set of resources from their host computers that cannot be used by any other service agent. Thus, our approach is simply a variation of overprovisioning.

Overprovisioning is not a novelty. Let us consider other examples of that technique. One of them is the aggressive resource provisioning mechanism proposed by Liu et al. (2015) [10] in their SPRNT system. The SPRNT adaptive strategy is based on reinforcement learning: it is continuously assessing the resource-performance relationships. The resources assumed in SPRNT are virtual machines (they are more difficult to administer than containers), and response time is the main performance metric in that system. Reinforcement learning is a dynamic predictive model [15] that considers recent history in order to define several workload categories, assessing the resource needs of each identified category. Progressively, that model is refined considering the actual past resource needs for each workload category, improving in this way its precision. This strategy introduces a problem: in many cases, it needs a very long learning stage before providing accurate predictions. Liu et al. [10] partially heal that problem applying an overprovision when SPRNT recommends a scaling action. Those additional virtual machines may appropriately deal with forthcoming high workload increases, reducing SLA violations in those cases. Those exceeding virtual machines are stopped once the workload stabilises. However, this overprovisioning could be problematic if the forecast workload increase never arrives. Thus, the adequacy of this solution highly depends on the reinforcement learning precision.

Our solution is not subjected to uncertainty and requires a light and small machinery. Therefore, it might be used in the initial stages of systems like SPRNT and PAS, while their prediction accuracy was still low.

6 Conclusions

A prototype elasticity management system for a cloud platform has been presented. It is based on two main components: an elasticity manager and a service broker. There are as many service brokers as deployed services are in each host. Each broker monitors the response time of each served client request, propagating periodically those monitored metrics to the elasticity manager. The elasticity manager evaluates those metrics and takes the appropriate scaling actions. To this end, it uses an operational strategy based on two container operations: pause and resumption.

That strategy has an almost immediate response. Therefore, it is better than the classical one based on create and destroy operations if the SLO being considered is response time. Indeed, in some experiments, the create/destroy strategy already fails when the SLO imposes a limit 80 times larger than the service time, while the resume/pause strategy has worked adequately even for limits less than twice larger than the service time.

Although these results are encouraging, the resume/pause strategy introduces several problems that should be considered: (1) its request filtering overhead is small (20 μs per request), but it might be non-negligible in applications with short response times, (2) paused containers hold some hardware resources (e.g., main memory) that cannot be assigned to other candidates. In spite of this, its results are quite close to those of a hypothetical optimal management and may be used, as a minimum, for handling elasticity decisions while other more ellaborate proactive elasticity management alternatives, used in the same system, are building their historical base in order to improve their accuracy.

References

[1] Marcelo Cerqueira de Abranches and Priscila Solis. An algorithm based on response time and traffic demands to scale containers on a cloud computing system. In *15th IEEE International Symposium*

on Network Computing and Applications (NCA), pages 343–350, Cambridge, Boston, MA, USA, November 2016.

[2] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Software Eng.*, 30(5):295–310, May 2004.

[3] Emiliano Casalicchio and Luca Silvestri. Mechanisms for SLA provisioning in cloud-based service providers. *Computer Networks*, 57(3):795–810, February 2013.

[4] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and Linux containers. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, Philadelphia, PA, USA, March 2015.

[5] Rui Han, Li Guo, Moustafa Ghanem, and Yike Guo. Lightweight resource scaling for cloud applications. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing CCGrid*, pages 644–651, Ottawa, Canada, 2012.

[6] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Comp. Syst.*, 27(6):871–879, 2011.

[7] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Response time service level agreements for cloud-hosted web applications. In *6th ACM Symposium on Cloud Computing (SoCC)*, pages 315–328, Kohala Coast, Hawaii, USA, 2015.

[8] Michael Larabel. Benchmarking Amazon EC2 instances vs. various Intel/AMD CPUs, February 2018. Available at: https://www.phoronix.com/scan.php?page=article&item=amazon-ec2-feb2018&num=1.

[9] John Dutton Conant Little. A proof of the queueing formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961.

[10] Jinzhao Liu, Yaoxue Zhang, Yue-Zhi Zhou, Di Zhang, and Hao Liu. Aggressive resource provisioning for ensuring QoS in virtualized environments. *IEEE Trans. Cloud Computing*, 3(2):119–131, 2015.

[11] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Recommendations of the National Institute of Standards and Technology, Special Publication 800-145, September 2011.

[12] Dirk Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 239:76–91, March 2014. URL: http://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.

[13] Mohamed Mohamed, Mourad Amziani, Djamel Belaïd, Samir Tata, and Tarek Melliti. An autonomic approach to manage elasticity of business processes in the cloud. *Future Gener. Comp. Sys.*, 50:49–61, September 2015.

[14] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 386–393, Tempe, AZ, USA, March 2015.

[15] Francesc D. Muñoz-Escoí and José M. Bernabéu-Aubán. A survey on elasticity management in PaaS systems. *Computing*, 99(7):617–656, July 2017.

[16] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys Conference*, pages 289–302, Lisbon, Portugal, March 2007.

[17] Phoronix Media. Phoronix Test Suite - Linux testing and benchmarking platform, automated testing, open-source benchmarking, December 2018. Available at: https://www.phoronix-test-suite.com/.

[18] Rob Pike, David L. Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, April 1993.

[19] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[20] Shriram Rajagopalan. *System Support for Elasticity and High Availability*. PhD thesis, The University of British Columbia, Vancouver, Canada, March 2014.

[21] Rhodney Simoes and Carlos Alberto Kamienski. Elasticity management in private and hybrid clouds. In *7th IEEE International Conference on Cloud Computing (CLOUD)*, pages 793–800, Anchorage, AK, USA, June 2014.

[22] Steven J. Vaughan-Nichols. New approach to virtualization is a lightweight. *IEEE Computer*, 39(11):12–14, November 2006.