LARGE-SCALE ELASTIC ARCHITECTURE FOR DATA AS A SERVICE

Project Number:	FP7-ICT-318809
Project Title:	Large-Scale Elastic Architecture for Data as a Service
Deliverable Number:	D4.3 Text
Title of Deliverable:	Prototype of scheduling and data placement supporting both vertical and horizontal agile elasticity
Contractual Date of Delivery:	M24 – 9/30/2014
Actual Date of Delivery:	9/30/2014

Abstract

In this deliverable, we present our approach for providing vertical and horizontal elasticity. We achieve vertical elasticity by tuning the CPU resources when we experience sudden load spikes while we scale horizontally when the underlying physical machine has reached its resource limits. In our evaluation, we show that we can achieve a cost reduction by factor of 2 when we use vertical scaling and reduce costs by 54% using horizontal scaling in comparison to commercial offerings such as Amazon auto-scale.

List of Contributors

Name	Organization	E-mail
Wojciech Barczynski	Cloud & Heat	wojciech.barczynski@cloudandheat.com
Marcel Gädig	Cloud & Heat	marcel.gaedigk@cloudandheat.com
Jens Struckmeier	Cloud & Heat	jens.struckmeier@cloudandheat.com
Anja Strunk	Cloud & Heat	anja.strunks@cloudandheat.com
Emmanuel Bernard	Red Hat	ebernard@redhat.com
Jonathan Halliday	Red Hat	jonathan.halliday@redhat.com
Mark Little	Red Hat	mlittle@redhat.com
Mircea Markus	Red Hat	mmarkus@redhat.com
Pedro Ruivo	Red Hat	pedro@infinispan.org
Eleftherios Chatzilaris	TSI	echatzilaris@softnet.tuc.gr
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Nikolaos Giatrakos	TSI	ngiatrakos@softnet.tuc.gr
Ekaterini Ioannou	TSI	ioannou@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
Nikolaos Pavlakis	TSI	npavlakis@softnet.tuc.gr
Ioakim Perros	TSI	imperros@softnet.tuc.gr
Evagelos Vazeos	TSI	vagvaz@softnet.tuc.gr
Christof Fetzer	TUD	christof.fetzer@tu-dresden.de
André Martin	TUD	andre.martin@tu-dresden.de
Do Le Quoc	TUD	do@se.inf.tu-dresden.de
Frezewd Lemma Tena	TUD	frezewd_lemma.tena@mailbox.tu-dresden.de
Frank Busse	TUD	frank.busse@tu-dresden.de

Document Approval

	Name	Email	Date
Approved by WP Leader	Christof Fetzer	christof.fetzer@tu-dresden.de	2014-09-15
Approved by GA Member 1	Xiao Bai	xbai@yahoo-inc.com	2014-09-30
Approved by GA Member 2	Minos Garofalakis	minos@softnet.tuc.gr	2014-09-25

Contents

List of Figures

GLOSSARY

EU	European Union
FP7	Seventh Framework Programme
Micro-cloud	A cluster of machines with virtualization capabilities
Node	A machine inside a micro-cloud
VM	Virtual Machine
IaaS	Infrastructure-as-a-service
DaaS	Data-as-a-service

1. Executive Summary

We have witnessed a tremendous growth in data volume during the past decade. In order to cope with such a growth, scalable systems must be built. The LEADS platform allows its users to perform data crawling and processing at large scale, however, contrary to companies such as Google or Yahoo!, LEADS uses a deployment infrastructure formed of micro-clouds spread geographically. Especially in this context, scalability is essential for providing reliable and scalable services.

In this deliverable we will present our approach for providing scalability in such an environment. Our approach is based on two dimensions, vertical elasticity which allows us to tolerate short-term bursts as well as horizontal elasticity, where additional resources are added or removed during runtime based on the evolution of the demand.

We present our controller based approach which allows reducing the execution time of MapReduce-like jobs running on LEADS through CPU tuning if short-term bursts are experienced, while new resources i.e., virtual machines (VMs) are acquired if the physical capacity has reached its limits in order to reduce runtime of jobs deployed afterwards. In our evaluation, we show that vertical elasticity can significantly improve execution time and cut costs compared to state-of-the-art elasticity mechanisms such as offered by vendors such as Amazon AWS auto-scale. We are able to cut costs to 54% using horizontal scaling and reduce cost by factor of 2 when using vertical scaling.

2. Vertical Scalability

2.1 Introduction

Many applications running in the cloud have time-varying workload. These applications are sensitive to resource over-provisioning, because response time and throughput quickly degrade if a server is highly utilized. For MapReduce applications, this results in a longer job completion time. Therefore it is important to avoid such situations. The resource assignment mechanism should be able to quickly adapt to the changing application requirements. Traditional techniques such as horizontal scaling [1] require significantly more time to scale an application. Average start time of a VM is about 1 minute [2]. Moreover VM consolidation causes overhead for the network resource in case of VM migration, since the VM needs to be moved from one host to another. Virtualization technology on the basis of Linux control groups provides a vertical scaling mechanism which is a lightweight and fast way to provision a VM. Instead of moving the VM to another host or starting a new VM, one can assign more or less resources to the VM on-the-fly. Vertical scaling has low overhead. For example, assigning additional CPU power to the VM takes less than one second [2]. Cloud providers such as CloudSigma [3] and ProfitBricks [4] already support vertical scaling. CloudSigma offers REST interface to allocate/deallocate resources to the VM. ProfitBricks allows changing the VM capacity over WEB interface. However, none of these providers offer solutions for automatic vertical scaling.

The focus of our work is to provide mechanisms to quickly adapt application resource assignment for MapReduce-like applications (Queries deployed on LEADS) or custom plugins running on LEADS with respect to the changing demand, and to improve utilization of allocated resources. Figure 1 presents average CPU utilization of worker nodes in a Hadoop cluster for different MapReduce jobs. The first graph shows how each job utilizes the CPU differently. Moreover, there is a significant difference between the map and reduce phase for applications such as PageRank and WordCount jobs commonly used in LEADS, whereas a simple Sort job has almost equal CPU utilization during its map and reduce phase. The second graph in Figure 1 depicts the time spent in each phase. For example, the reduce phase of the PageRank job takes up to 70% of the completion time of the entire job. Hence, 70% of the time a Hadoop cluster will be underutilized by 20% if we assigned resources statically based on the utilization of the map phase. Moreover, if we do not change the resource assignment but run another job, for example a sort job, the cluster will be underutilized at around 20% during the entire sort. These examples illustrate that resource assignments should consider the entire job run time rather each single job.

Note that in the previous examples the CPU utilization has been the limiting factor and reason for degraded performance we experience in our micro-benchmarks. However, network and disk bandwidths are additional limiting factors which are not considering for vertical scaling in this work as those resources cannot be adjusted on the fly through common hypervisor technologies such as KVM.

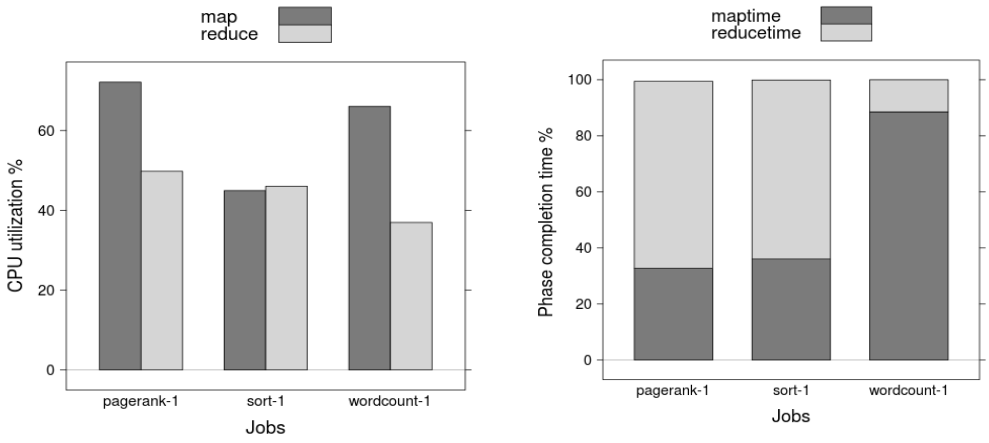

Figure 1 MapReduce job per-phase CPU utilization and per-phase job completion time.

2.2 Tuning CPU Speed

Tuning the speed of a virtual CPU is one of the techniques of vertical scaling which is based on Linux control groups [5]. The basic idea is to assign a physical CPU (pCPU) quota to each virtual CPU. The CPU quota value specifies the total amount of time spent in microseconds per second by the physical CPU running a VM task. Figure 2 presents the relationship between the Hadoop job completion time and the virtual CPU quota. For simplicity, we present the CPU quota limit in percent of the physical CPU. The graph shows that a decrease of the CPU quota causes almost the same slowdown for each job. Allocating a lower CPU speed (under-provisioning) to the VM during job runtime leads to a longer job completion time. Therefore it is important to quickly react on the increased CPU usage and assign the required CPU speed.

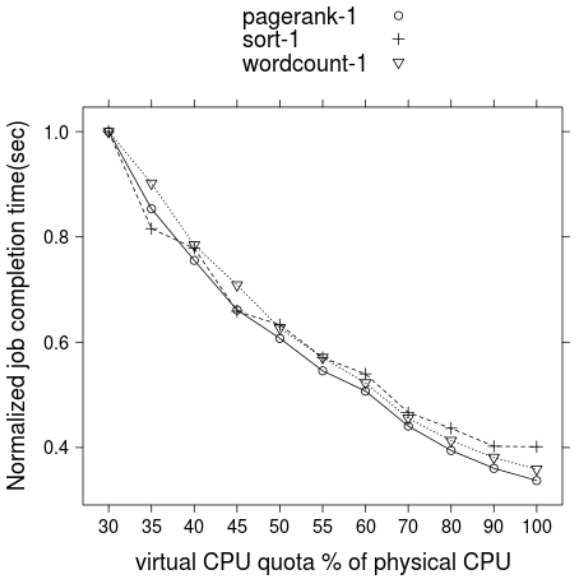

Figure 2 Effect of virtual CPU quota on job completion time.

2.3 Scaling Rules

We describe the dynamic resource allocation in the form of elasticity rules. Figure 3 presents an example of such rule. The rule consists of two parts: The *condition* and an *action*. The *condition* specifies a metric <*Metric*>, for example CPU or memory utilization to be tracked and the threshold <*Th*> to be checked. The *action* determines the amount of resource <*Resource*> capacity <*P*> to allocate/de-allocate.

The rules are used in a following way: For each time interval <T> the metric <*Metric*> is compared against the defined threshold <*Th*>. If the condition holds for a time window <*Tw*> then the defined action is triggered. After the action has been triggered, the elasticity rule can also specify a cool down period <Tc> for which the system has to wait prior to a novel evaluating of the condition.

MONITOR <*Metric*> *EVERY* <*T*>
  IF <*Metric*> {<, >} <*Th*> *FOR* <*Tw*>
CHANGE <*Resource*> *by* <*P*>
WAIT FOR <*Tc*>

Figure 3 Elasticity rule syntax

One of the difficult parts of configuring dynamic resource scaling actions is to define the parameters of the elasticity rule. In our work we fix the threshold value. However, we will change other parameters such as capacity allocation value <P> and cool down period <Tc> to evaluate how they affect the quality of the resource scaling. Therefore we investigated several alternative rules.

2.4 Vertical Elasticity Controller

Our elasticity controller performs CPU provisioning with respect to the current CPU demand. The controller collects statistics of the CPU utilization of each node. The data is stored in a key-value store such as Infinispan ISPN. Every second, the controller requests from the KV store the CPU usage over the last period and checks if one of the scaling actions can be executed. Scaling actions are performed based on the rules configured for the controller. Two such sets of rules are defined: scaling up and scaling down. The first rule prevents performance degradation by allocation additional CPU capacity while the second rule reduces resource usage cost by releasing/freeing virtual CPU quota.

2.5 Evaluation

We performed experiments to evaluate our controller implementation. For a test bed, we used a micro-cloud located in Dresden hosting Nutch, our web-crawler. As Nutch is a mature web-crawler implementation and open source, it has been used as a code basis and extended for operations with ISPN within WP1 of the LEADS project. Although the original Nutch implementation uses Hadoop, an open source implementation of MapReduce, we extended it to store fetched contents directly in ISPN rather than on a distributed file system.

In comparison to standalone MapReduce jobs that can run on top of Infinispan, Nutch comprises of a number of different jobs which are repeated in several rounds in order to carry out the crawling. The jobs have different CPU consumption pattern. For our evaluation, we limited the number of rounds to five. We furthermore applied six different sets of resource allocation rules. Table 1 summarizes these rules. The first two rules are based on static allocation which represents a fixed VM size allocation. The controller assigns the resources only during start-up. The first CPU assignment rule considers the average CPU demand while the second previously reported peaks demand which guarantees that the application has sufficient resources over the whole course of time. However, this leads to

resource wastage as CPU demand can vary over the time. The first two rules are used as a baseline for the resource utilization and the performance experiments while the remaining ones perform the desired dynamic resource allocation. Each set of rules contain two distinct rules: for scaling up and down. However, they differ in terms of the capacity allocation value <P> and cool down period <Tc>. Rules #3 and #4 have fixed capacity allocation value <P> while for rule #5 and #6 the value of <P> is dynamic. These two latter rules use linear and exponential multipliers, respectively. We apply a mean average to compute the CPU usage over the time window <Tw>. We set the upper and lower thresholds to 60% and 30% respectively.

N	***Rule name***	***Condition***	***Action P***	***Cool down period***
1	static avg	average demand allocation	None	N/A
2	static peak	peak demand allocation	None	N/A
3	cool down fixedP	if CPU > 60% for 5 sec.	20% pCPU	5 sec.
		if CPU < 30% for 5 sec.	- 5% pCPU	5 sec.
4	reactive fixedP	if CPU > 60% for 5 sec.	20% pCPU	No
		if CPU < 30% for 5 sec.	- 5% pCPU	No
5	reactive linear factorP	if CPU > 60% for 5 sec.	*linear_v	No
		if CPU < 30% for 5 sec.	- 5% pCPU	No
6	reactive exp factorP	if CPU > 60% for 5 sec.	*exp_v	No
		if CPU < 30% for 5 sec.	- 5% pCPU	No

Table 1 Rules

Figure 4 depicts the results obtained for five rounds of a Nutch crawler-run, against the normalized completion time. This normalization is performed with respect to peak allocation rule. Rule #1 yields the worst completion time, since the demands above average CPU usage are not sufficiently provisioned. This leads to a slowdown that requires four times of the execution time in comparison to peak as well as dynamic allocation. The remaining rules increase completion time by only 30%. The experiments using rule #3 and #6 have completion times close to rule #1.

Rule #3 has a cool down period <Tc> which prevents an oscillating behaviour due to sudden CPU load spikes. Hence there are periods a VM is under provisioned for at most <Tc> seconds. The cool down period for rule #4 is set to zero in order to trigger a de-allocation of resources as soon as the CPU utilization drops below the lower threshold. However, if the load suddenly increases then it requires triggering rule #4 couple of times until the allocation reaches the desired level. As result rule #4 has longer than rule #3 completion time. The rule with exponential multiplier eliminates the drawback of rule #4. It can quickly increase value of <P> if the demand suddenly increases. Therefore it gives similar to rule #3 completion time.

Figure 5 depicts the efficiency of the different rules applied. It presents a normalized CPU utilization across several Nutch nodes. The CPU utilization is normalized against highest CPU usage values. Rules that do not utilize a cool down period provide in general a higher utilization for nodes while the rule #3 has the second lowest utilization value as the controller waits until the cool down period has passed prior to releasing underutilized resources.

Finally, we computed the total cost when running the Nutch crawler for five rounds. The cost is the sum of the virtual CPU allocations over five rounds. The normalization is performed against the lowest cost. The results are shown in Figure 6. The graph shows that a static resource allocation based on average resource demand results in higher costs than a peak demand allocation. The use of dynamic resource scaling (rule #3) allows the reduction of costs.

Rules #4-6 do not employ cool down periods, hence, they give an improvement from 40% to 60% over the cost as when using rule #3. The lowest possible costs can be achieved by solely applying rule #6 as it outperforms rule #5 due to the exponential multiplier and its allocation value <P>.
The presented evaluation reveals that we can save twice the costs using the exponential multiplier rule in comparison to the peak allocation scheme while only increasing 18% the overall job completion time.

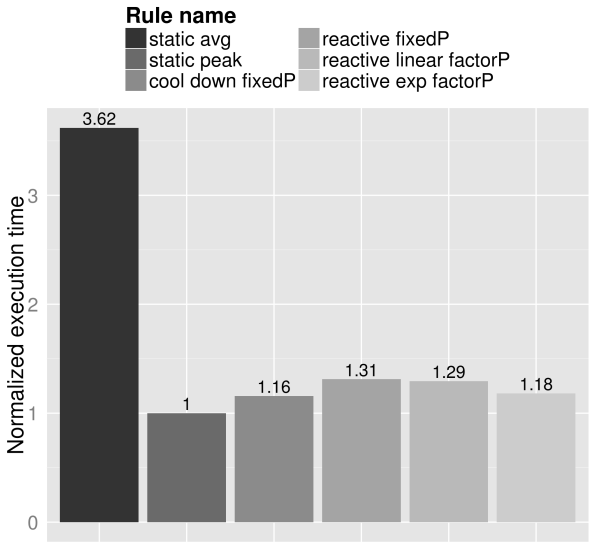

Figure 4 Job completion time

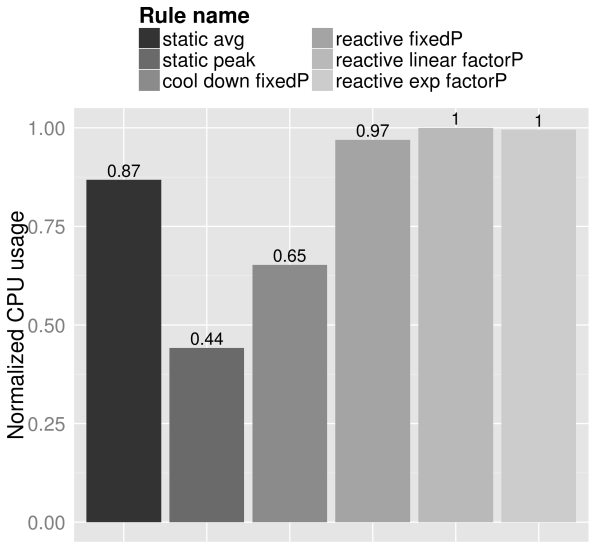

Figure 5 CPU usage

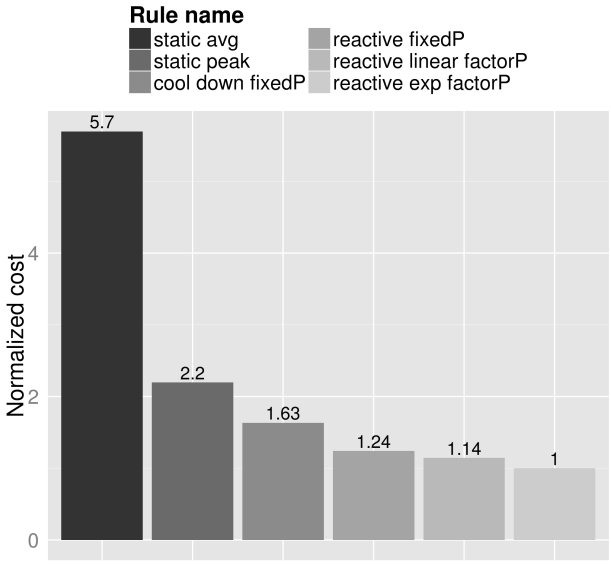

Figure 6 Resource cost

3. Horizontal Elasticity

3.1 Overview & Motivation

In this section, we describe our approach on providing horizontal elasticity in LEADS. Horizontal elasticity allows us to add and/or remove processing as well as storage nodes during runtime. This mechanism is required whenever vertical elasticity has reached its limits due to the available resources (CPU and Memory) of the underlying hardware, or when nodes are underutilized over long periods of time and when shutting down nodes can lead to savings monetary and energy wise.

We will first discuss challenges when providing horizontal elasticity, targeting the data processing as well as the crawling layer of LEADS. The crawling layer is carried out using Nutch, the open-source web-crawler which has been extended within WP1 to meet requirements of the LEADS project. The first release of the crawler will be carried out in a non-distributed version which is going to be extended during the following months to work in a distributed fashion.

For both layers, we assume an unpredictable and erratic load behaviour which is caused by the runtime behaviour of the operators used in (*i*) queries issued by LEADS customers, (*ii*)user plugins running on the platform and the (*iii*) web-crawler itself.

Deciding whether to scale out, i.e., acquiring new resources or maintaining the current pool of resources is not a trivial task as it depends on several factors. A naive approach for providing horizontal elasticity is the monitoring of the resource consumption and the acquisition of new resources as soon as the system exceeds its accumulated capacity. However, since future requirements cannot be predicted, a scale out might waste resources if the load increase lasts only for a small period of time, which can be the case when processing data using custom plugins where the nature of the operator does not expose a stable load over the course of processing.

Another challenge is the delay until a newly-acquired resource is available to the system, as starting up VMs in a cloud environment can take up to several seconds to minutes, in addition to the start-up time of the deployed storage and data processing system used in LEADS. Hence, resources must be acquired well in advance in order to prevent unresponsiveness of the service due to rapidly increasing load. Vertical scaling as described in Section 2 helps us to prevent unresponsiveness during a horizontal scale out.

Considering these challenges, we can establish the following requirements when designing an elasticity controller for LEADS: First the controller must be able to cope with arbitrary and unpredictable workloads that comprise long and short-term load bursts, and second, the controller must consider delays of resource acquisition and load balancing that cannot be accommodated through vertical scaling.

3.2 Self-Adaptive Elasticity Controller

In the following section, we will describe our approach for providing horizontal scaling in LEADS. Horizontal elasticity is achieved through a self-adaptive elasticity controller. Self-adaptation is necessary as the load behaviour is unpredictable where both long-term and short-term bursts must be properly handled.

In order to scale elastically, the controller component constantly monitors metrics such as the CPU utilization of all participating nodes in the cluster and observes them over a sliding window. The size

of the sliding window will be adjusted over time by the controller in order to adapt to the various workload as we will describe more in detail in the following. Measurements that fall in such a sliding window are averaged using an arithmetic average.

The performance probes collected by the controller consist of CPU, memory as well as network utilization. This allows us to perform a scale out also when resources such as the network bandwidth saturates and prevents swapping of the system in case of memory exhaustion.

We use a single controller that monitors each micro-cloud as a whole as well as its individual nodes. If the accumulated capacity load has reached its limit, new resources must be acquired while an overload of a single node can be handled through simple load balancing techniques (moving the computation to another, less loaded node) if other nodes have still sufficient spare resources available.

However, the data storage layer as well as the data processing technology must support appropriate mechanisms for balancing load across its nodes. Such mechanisms include the migration of data as well as processing tasks. Infinispan supports load balancing on storage level which we will leverage for horizontal elasticity, hence, data is spread across newly acquired nodes using its x-transfer feature when new nodes are added to a micro-cloud. However, data processing tasks encapsulated as MapReduce will not be migrated during runtime as they are characterized through finite execution times contrary to event stream processing. Furthermore, migrating MapReduce tasks would imply shuffling around Terabytes of data which is impractical in the micro-cloud environment where bandwidth is limited between micro-clouds.

In our approach, we consider both scale out and load balancing techniques as it leads to a highly scalable system that dynamically adjusts with the amount of data needed to be stored as well as to be processed. In order to evaluate our approach, we are using StreamMine3G, an elastic data processing system which fits well in the plugin architecture developed within WP3 (processing layer) where continues streams of data are processed through the Listener Interface provided by Infinispan. StreamMine3G supports load balancing during runtime using operator migration (of stateful operators) as well as the addition and removal of nodes for a scale-in and out.

Our controller component uses two thresholds for determining if a node is overloaded or underutilized: The ***upper threshold UT*** is used for moving processing tasks to less loaded nodes within a micro-cloud if the current value of the monitored resource exceeds the given threshold while the ***lower threshold LT*** determines the boundary for a contraction, i.e., freeing resources. Both thresholds are specified by the user and given in percentage as of the maximum achievable utilization.

3.3 Scale-Out

Scale-out describes the process of acquiring new resources when an overload situation has been detected. In order to detect such an overload, the elasticity controller monitors the utilization of several performance metrics such as CPU, network and memory utilization. If at least one of those metrics exceeds the previously defined threshold *UT*, an overload situation is detected which can cause unresponsiveness to the service. However, in order to prevent unavailability of the service, new resources must be acquired well in advanced and before exceeding the nodes capacity.

We therefore perform a short-term prediction of utilization of the evolution of each monitored resource using historical measurements which we will illustrate in an example shortly. If this short-term

prediction indicates that any of the monitored metrics may surpass the defined threshold *UT*, new resources are acquired.

The prediction is re-computed each time a new performance probe is available. It also considers the time it takes until the newly acquired resource is available based on historically available data and considering the limits of the vertical scalability. As an example, consider a system utilization of 80% and a threshold UT of 90%. Let us assume the utilization constantly increases by 1% with every minute, hence, our prediction reveals that after 10 minutes the *UT* threshold will be reached. In case the acquisition of a new VM and the shifting of the processing task may require three minutes, the latest point in time the new resources must be acquired in order to be available on time is when the utilization has reached 87% assuming that the load increase stays constant with a 1% increase per minute. An example is shown in Figure 7.

The estimation is performed by iterating over all nodes and predicting the future CPU, memory and network consumption. If any of those metrics exceed the previously set threshold, a scale out will be triggered.

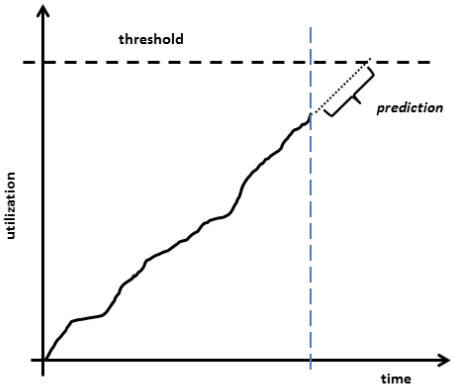

Figure 7 Prediction Example

The elasticity controller uses an arithmetic average for all metrics in order to cope with load spikes and monitors the recent history of performance measurements using a sliding window of 5 or 15 seconds. The arithmetic average filters load spikes by smoothen the observed metric. The duration of the window is adjusted based on the experienced workload as we will detail in the Section 3.5.

In order to use as few resources as possible, each node should maximize the utilization of the available resources while still being responsive, hence, a scale-out should be performed as late as possible while keeping the utilization always close to the given *UT* threshold. However, this can lead to situations where multiple nodes require a scale out at the same time.

In order to prevent unresponsiveness of the system, the scale-out-factor defines the number of new nodes the controller acquires in a batch as soon as an overload situation is detected. The factor is adjusted based on the evolution of the resource utilization, hence, if the utilization only increased very slowly during the past hour, fewer nodes will be acquired compared to the situation when the load suddenly increased by multiples of magnitude during a short period of time. The number of nodes to be acquired is determined by the sum of nodes that indicate an overload.

3.4 Scale-In

Scale-in describes the process of freeing unused resources, i.e., contracting the cluster. Similar to Scale-out, a sliding window is used for deciding when nodes can be released or not. A scale-in usually requires the migration of processing tasks from low utilized nodes to nodes that still have sufficient capacity for the task in order to free and release the node.

Candidates for a scale-in are chosen based on their utilization and costs they would incur while continue running. The costs are defined by the cost model which considers also energy consumption aspects that will be addressed during the final year of the project in T4.4 and 4.5 within WP4. Hence, nodes that would incur very low costs are kept running until its costs would increase again. This has the advantage of having already spare resources in case the load increases suddenly. The algorithm iterates of all computational tasks running on a node determining nodes with spare resources for a scale-in.

One difficulty of the proposed solution is the algorithm's complexity. Essentially, it is a bin packing problem which is NP-hard where the first-fit algorithm has a $O(n \log n)$ time complexity. In our case, n would be the number of tasks on the nodes needed to be re-distributed across the cluster. Assuming a large amount of tasks and a micro-cloud consisting of around 20 nodes would result in multiple executions of the scale-in algorithm and, therefore, the bin packing problem for one scale-in process.

In order to decrease the computational complexity, we group tasks on a single node in tasks packs. The idea is not to execute the scale-in algorithm with a large number of tasks but with a much smaller number of tasks packs. If a node is overloaded and has several hundred tasks deployed, each task often accounts only for less than 1% of the node's utilization. Thus, grouping tasks into packs, each with a combined utilization of approximately 3%, reduces the number of objects for the scale-in algorithm dramatically. A node with 1000 concurrently running tasks and 50% CPU utilization can be reduced to 17 packs. Solving the bin packing problem for such a small number is computationally feasible. However, a disadvantage is the lost granularity as a single node has to have sufficient capacity for a task pack with a utilization of at least 3%.

3.5 Self-Tuning

As the load behaviour is unpredictable and might change over time, we adjust the *UT* based on the load behaviour observed in the past. To make a better distinction between the *UT* specified by the user and the *UT* adjusted by the controller, we refer to the *UT* used internally at the controller as current upper threshold ($UT_{current}$). The $UT_{current}$ value operates between 50% and the user set *UT* threshold. In addition to the *UT* value, the controller adjusts the window size of the sliding window used for resource monitoring. Based on the results of micro benchmarks we performed, we use a 15 seconds window which is optimal for workloads with heavy load spikes while we use a five seconds long window for more moderate and less burst-y workloads.

3.5.1 Current Upper Threshold Adjustment

The controller uses a vector of $UT_{current}$ where each entry in the vector represents a different resource that is monitored such as CPU and/or network utilization etc. In order to prevent an oscillation of the controller, small load burst are filtered out by averaging measurements within a three-second period.

The $UT_{current}$ value is adjusted once every four minutes. Within such an adjustment cycle, we count the amount of times a monitored resource exceeds the threshold that was previously set. As such an overload does often lead to unresponsiveness of the system, the time a system resides in such a state should be minimized, and hence, we define an unresponsive threshold which is the ratio of time the system stayed or did not stay within this bound.

In case the system is declared as unresponsive, the *UT* value is decreased incrementally within the four minutes long cycle by 4%. The decrease implies the automatic removal of computational tasks from the node the overload has been detected on. In case the system did not pass (i.e., no load spike) the previously set *UT* threshold during the cycle, its value is increased by 1% which allows the shifting of additional computational tasks to that node for a better overall resource utilization. The controller keeps the *UT* constant as long as the amount of unresponsiveness within a cycle resides in the previously defined limits.

3.5.2 Window Size Adjustment

In addition to the adjustment of the *UT* value, our controller also adjusts the size of the sliding observation window based on the workload characteristics. If more frequent spikes are observed, the window size is increased in order to prevent too frequent scaling actions while on more moderate workloads the window size is decreased to quickly react if the load changes suddenly.

The observation window size is initially set to give seconds and updated every 500 milliseconds where the derivative (steepness) for each node is computed. If the average across all nodes is larger than 40%, the window size is increased to 15 seconds while an average of less than 15 % will decrease the sliding window size back to five seconds again. Listing 1 summarizes the behaviour.

Algorithm Self-tuning window size algorithm.

```
input : Cluster
1  sumSteepness = 0;
2  sumSystemDemand = 0;
3  foreach Node in Cluster do
4  |   sumSteepness += Node.getSteepness();
5  |   sumSystemDemand += Node.getAverageDemand();
6  end
7  ratio = sumSteepness / sumSystemDemand;
8  if ratio > 40 % then SetWindowSize(30);
9  else
10 |   if ratio < 15 % then SetWindowSize(10);
11 end
```

Listing 1 Self-tuning window size

After an increase of the sliding window size, no adjustment will be performed for at least 15 seconds in order to prevent oscillation of the system (cool down period). The 15 seconds cool down period has been determined empirically. Figure 8 summarizes the hysteresis behaviour of the self-tuning window size mechanism in detail.

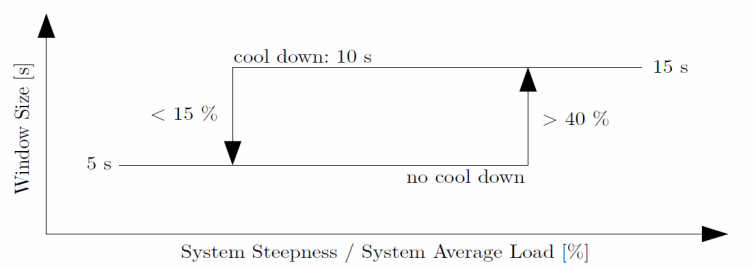

Figure 8 Hysteresis behaviour of the self-tuning window size mechanism.

3.6 Scaling Across Micro-clouds

In this section, we will describe our approach for horizontal scalability across micro-clouds as the capacity of a single micro-cloud is limited. We achieve horizontal scalability across micro-clouds through the replication of data provided by the storage layer developed within WP2 and the data placement strategies carried out within WP4: Data items are replicated across racks within a single micro-cloud as well as across micro-clouds. Since data processing tasks in the distributed executor of Infinispan are placed on nodes holding the appropriate data caches, our scheduler can hence freely choose where to start (amongst micro-clouds holding the data), i.e., an execution of a data processing tasks. Alternatively, Emsemble cache described in WP2, Deliverable D2.3 can be used, however, with the cost of remote accesses which can incur additional and high network traffic.

With regards to elasticity, we harness the replication of data by switching off data processing tasks in micro-clouds in case the micro-cloud's capacity limit is about to reach. In order to do so, we select candidates of processing tasks for a tear down where replicas exists on less loaded micro-clouds. The candidates are ranked according to the incurred costs when re-initiating the processing task on the remote micro-cloud.

3.7 Evaluation

In the following section, we will present several experiments we executed in order to evaluate our approach. For evaluation and demonstration the advancement over state-of-the-art, we implemented Amazon's auto-scale elasticity controller for a comparison with regards to costs incurred by the two controllers and unresponsiveness of the elastic service when exposing it to different workloads.

3.7.1 Workloads

Our experiments are simulation driven. The simulator mimics the micro-cloud environment as well as the data processing layer. For the experiments, we use workloads that expose various different characteristics for evaluating of the controller's performance, where each workload comprises a set of the three metrics (CPU, memory and network). In order to have reproducible results, the workloads are based on recordings from real world scenarios. The real world scenario comprises of a Twitter-data stream captured in 2013 and server logs from webservers.

In our first experiment, we perform so called sanity checks with the following experiment setup: An initial node count of one node, 120 processing task representing running plugins as described in WP3, Deliverable D3.3, and an upper threshold set to 85% CPU, 80% memory and 85% network utilization. The observation window is set to five seconds. Figure 9 depicts the result of the experiment where we successively increase the load of the nodes for 25 minutes after we slowly decrease it again.

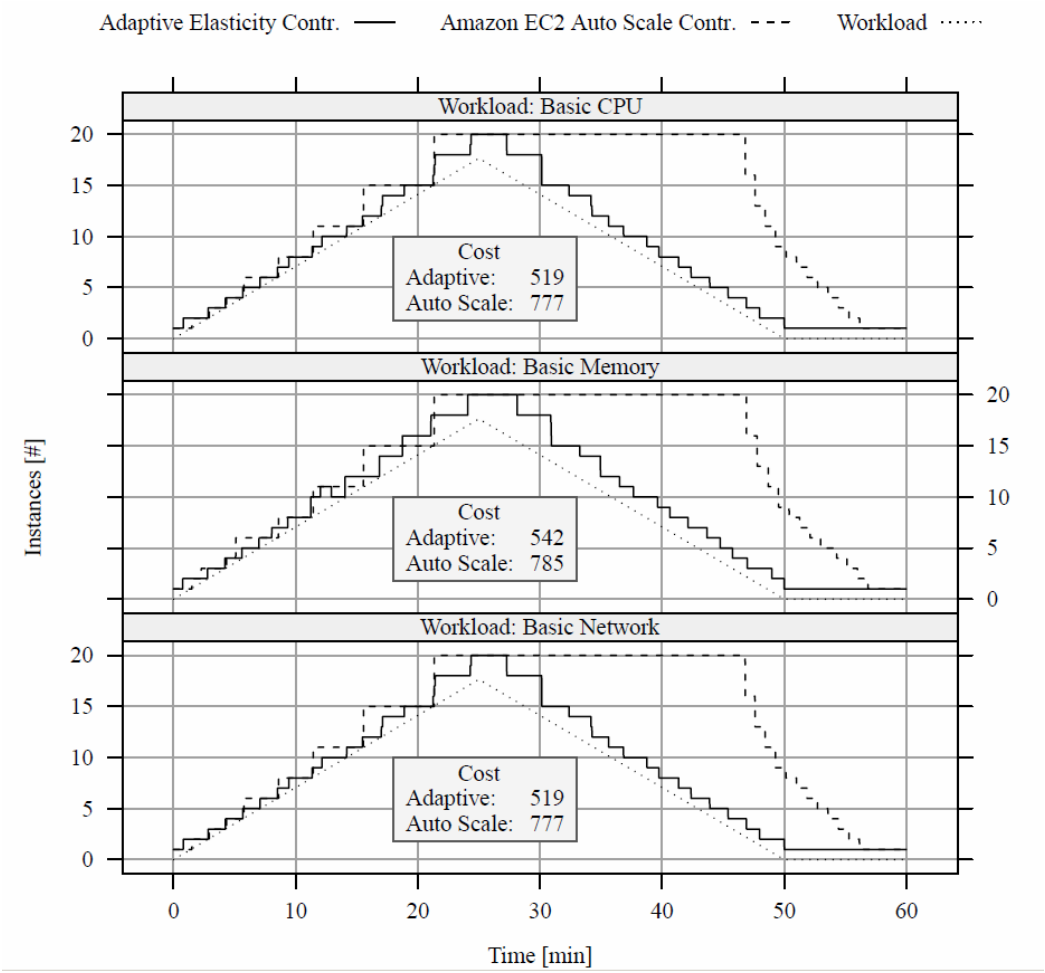

Figure 9 Sanity-check workloads and associated costs.

Concerning the scale-out behaviour, both controllers expose similar behaviours where additional nodes are required as soon as the thresholds are exceeded. However, Amazon's auto-scale keeps the acquired nodes for a long period of time even though these resources are not required anymore. The graphs show also the costs in machine hours incurred by the two different controllers. The large delay in releasing nodes in Amazon auto-scale is due to the cool down phases. Note, the costs do not include network transfer costs.

In the next experiment, we are comparing the costs when varying the *UT* threshold. A higher threshold leads to high monetary savings; however, it also increases the risk of service unresponsiveness. For the experiment we applied four different workloads to the controller. Two realistic workloads were chosen. The first one is characterized by heavy load spikes that can occur when processing crawled web pages online through customer's plugins while the second one is taken from periodically crawling of Twitter data. In addition to those real workloads, we generated a sinusoid and a steadily increasing workload similar to the one in our sanity check experiment. The results of the experiment are depicted in Figure 10 and Figure 11.

Regardless of the different characteristics of the workloads, our adaptive controller implementation always incurred less costs compared to Amazon's auto-scale. However, with regards to unresponsiveness, Amazon's auto-scale controller performs slightly better for the sinusoid workload. This is due to the fact that nodes are not immediately released causing higher costs.

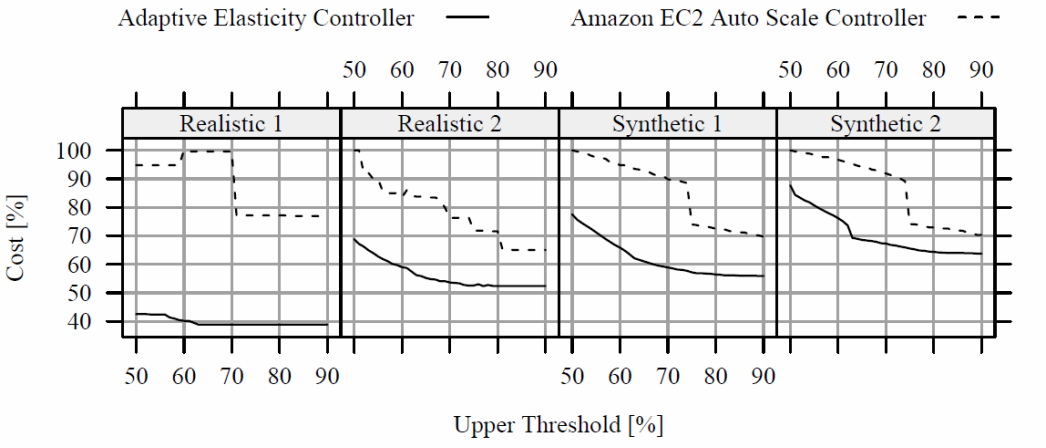

Figure 10 Costs with varying upper threshold.

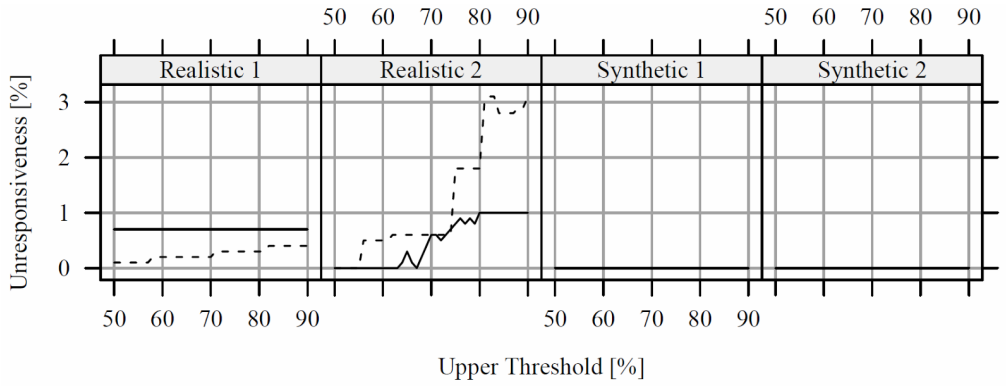

Figure 11 Unresponsiveness with varying upper threshold.

In the last Figure 12, we show the overall saving that can be achieved using our adaptive solution. The error bars indicate different configurations such as observation window sizes. In summary, we can save up to 54% of costs when using our adaptive solution compared to commercial offerings such as Amazon EC2.

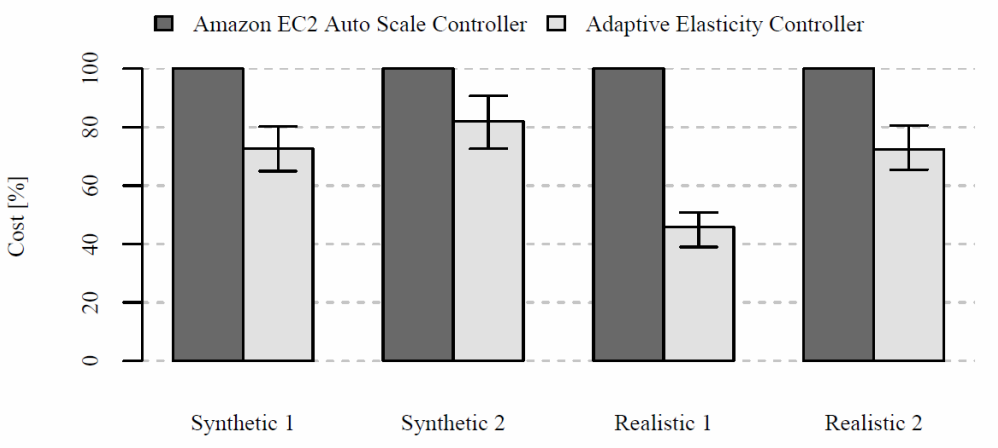

Figure 12 Cost savings comparison with Amazon EC2 Auto-scale

3.8 UCC Cloud Challenge 2014

In order to further evaluate our approach for horizontal elasticity and making it accessible to a broader audience, we extended our evaluation by processing energy consumption data and submitted this as a short paper to the UCC2014 conference. Our proposal was accepted and selected as one of the finalists for this year's Cloud Challenge 2014; hence, our system will be showcased during the Cloud Challenge session in London in December 2014. The submission is available in Appendix II.

4. Conclusion

In this deliverable we presented our approach for vertical and horizontal elasticity: Vertical elasticity allows us to tolerate short time bursts and guarantee high responsiveness of our service. We are able to adjust CPU speed tuning for each VM hosted within LEADS. However, when the limits of the underlying hardware are reached, we use horizontal elasticity to spread the work across more nodes which can go even beyond a single micro-cloud. Our evaluation reveals that we can (*i*) reduce task processing execution time using vertical elasticity as well as (*ii*) saving costs when scaling horizontally using our adaptive controller compared to state-of-the-art mechanisms such as auto-scale from Amazon.

5. Bibliography

[1] “Amazon Elastic Compute Cloud,” [Online]. Available: http://aws.amazon.com/ec2.

[2] S. G. A. V. a. B. V. S. Dutta, “Smartscale: Automatic application scaling in enterprise clouds,” in *IEEE Fifth International Conference on Cloud Computing*, Washington, DC, USA, 2012.

[3] “Cloudsigma, Infrastructure-as-a-Service provide,” [Online]. Available: http://www.cloudsigma.com.

[4] “ProfitBricks, Infrastructure-as-a-Service provider,” [Online]. Available: http://www.profitbricks.com.

[5] “Control Groups,” [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html.

[6] “OpenStack,” [Online]. Available: http://www.openstack.org/.

[7] Fabric. [Online]. Available: http://www.fabfile.org/.

Appendix I

Integration across Work Packages / LEADS Platform

The following section will describe the technical efforts taken in order to integrate vertical and horizontal elasticity in the technologies used in LEADS.

Vertical Elasticity

Cloud&Heat uses OpenStack [6] for virtual machine provisioning, however, the current implementation of the OpenStack software stack does not allow to adjustment of CPU resources during runtime, although KVM, the underlying hypervisor technology supports runtime adjustments using its Command Line Tools (virsh). We therefore developed a REST API Interface which allows us to adjust the resources addressed in this deliverable on-the-fly:

Request currently set resource quotas

Call format

```
POST /getquota<VMID> HTTP/1.1
Accept: application/json
Content-Type: application/json
```

Arguments

Parameter *vmID* specifies the name of the VM the current active quota values are requested.

Return format

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: <size>
{
  "scheduler"      : posix
  "cpu_shares"     : <value>
  "vcpu_period"    : <value>
  "vcpu_quota"     : <value>
  "emulator_period": <value>
  "emulator_quota" : -1
}
```

Modify currently set resource quotas

Call format

```
POST /setquota<VMID> HTTP/1.1
Accept: application/json
Content-Type: application/json
{
  "vcpu_quota"     : <value>
}
```

Arguments

Parameter *vmID* specifies the name of the VM where the current quota should be updated whereas *value* specifies the new value.

Return format

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: <size>
{
  "result"      : OK
}
```

The calls on the Cloud&Heat side are then transformed in CLI calls such as:

```
$ virsh schedinfo VMNAME
```

Horizontal Elasticity

Concerning horizontal elasticity, we are using the Python OpenStack Nova API for acquiring and releasing virtual machines such as in the following example code snippet.

```
from novaclient.v1_1 import client

nova_client = client.Client(USER, PASS, TENANT, AUTH_URL)
flavor_xs = nova_client.flavors.find(name="aocloud.xs")
image = nova_client.images.find(name="Ubuntu 14.04.1 LTS Server")
instance = nova_client.servers.create(name="LEADS", image=image, fla-
vor=flavor_xs)
```

Virtual machines are equipped with **fabric scripts** [7] in order to automatically install the Nutch crawler as well as Infinispan on newly acquired nodes. Infinispan as well as Nutch workers nodes register automatically at their master nodes and incorporate themselves in the network providing additional spare resources.

Reporting Statistics

For collecting several statistics such as CPU, memory and network utilization within virtual machines and across nodes and micro-clouds, we are using DoLen (https://bitbucket.org/lenin_army/dolen) an open source tool developed within the LEADS project. The collected statistics serve as a source for vertical as well as horizontal scalability decisions.