PeakForecast: Efficient Predictive Elasticity Resource Scaling for Cloud Systems to Absorb Traffic Surges

Daniel Rene Fouomene Pewo
University Yaounde 1
Cameroon
rene.fouemene@uy1.uninet.cm

Romain Rouvoy
Univ. Lille / Inria / IUF
France
romain.rouvoy@inria.fr

Lionel Seinturier
Univ. Lille / Inria / IUF
France
lionel.seinturier@inria.fr

Marcel Fouda Ndjodo
University Yaounde 1
Cameroon
marcel.fouda@uy1.uninet.cm

ABSTRACT

There is a lot of prior work that deals with dynamic resource management, to efficiently utilize elasticity of clouds, since over-provisioning leads to resource wastage and extra monetary cost, while under-provisioning causes performance degradation and violation of service-level agreement (SLA). These works can be classified into predictive, reactive, and mixed approaches. While these approaches can handle gradual changes in load, they cannot handle abrupt changes, especially traffic surge that occur almost instantaneously, like Slashdot effect.Unfortunately, such traffic variations are generally unplanned, of great amplitude, within a very short period, and a variable delay to return to a normal regime. In this paper, we introduce PeakForecast[1](PF) to deal that issue. PF is an elastic distributed resource scaling approach for IaaS cloud infrastructures that provide a medium-term resource demand prediction. PF can efficiently scale cloud resources up and down to absorb such traffic surges. We describe our resource allocation algorithm (PF algorithm) based on simple exponential smoothing (SES) prediction method and MAPE-K (Monitoring, Analysis, Planning, and Execution) loop, which auto-scaling resources by allocates or deallocates resources based on traffic of user requests. We also present the design and implementation of a prototype elastic middleware solution, based on PF. We validate our approach by experimental results by demonstrating that, our prototype can provide spontaneous elasticity of resources for traffic surges observed on the Japanese version of Wikipedia during the Fukushima Daiichi nuclear disaster in March 2011 and a dataset acquired from the FIFA 1998 World Cup web site.

Keywords

IaaS management, Auto-scaling, Forecasting model, Traffic surge, Slashdot effect.

1. INTRODUCTION

Elastic resource provisioning is a key feature of cloud computing, allowing users to scale up or down resource allocation for their applications at run-time. To date, most practical approaches to managing elasticity are based on allocation/de-allocation of the virtual machine (VM) instances to the application. This VM-level elasticity typically incurs both considerable overhead and extra costs, especially for applications with rapidly fluctuating demands [44].

Unfortunately, abrupt changes in load, or traffic surge, are all too common in today's data centers. Important events [3], such as the September 11 attacks [45, 46], earthquakes or other natural disasters [47], Slashdot effects [48], Black Friday shopping [49], or sporting events, such as the Super Bowl [50] or the Soccer World Cup [51], are common causes of load spikes for website traffic. Service outages [52] or server failures [53] can also result in abrupt changes in load caused by a sharp drop in capacity. While some of the above events are predictable, most of them cannot be predicted in advance, like the Slashdot effect, occur when a distributed system is overloaded by a huge number of requests, potentially leading to its

[1] https://github.com/Spirals-Team/peak-forecast

unavailability. There is a lot of prior work that deals with dynamic resource management [3], to efficiently utilize elasticity of clouds. These works can be classified into predictive [28, 33, 24, 23, 7, 8], reactive [44, 3, 4, 5, 6] and mixed [54, 55, 56, 57, 11, 10, 9] approaches. While these approaches can handle gradual changes in load, they cannot handle abrupt changes, especially traffic surge that occur almost instantaneously [3]. In this context, it is legitimate to raise the following research questions: *Is it possible to anticipate the effects of a traffic surge before under-provisioning resources? If so, is it possible to protect the targeted system from unavailability?*

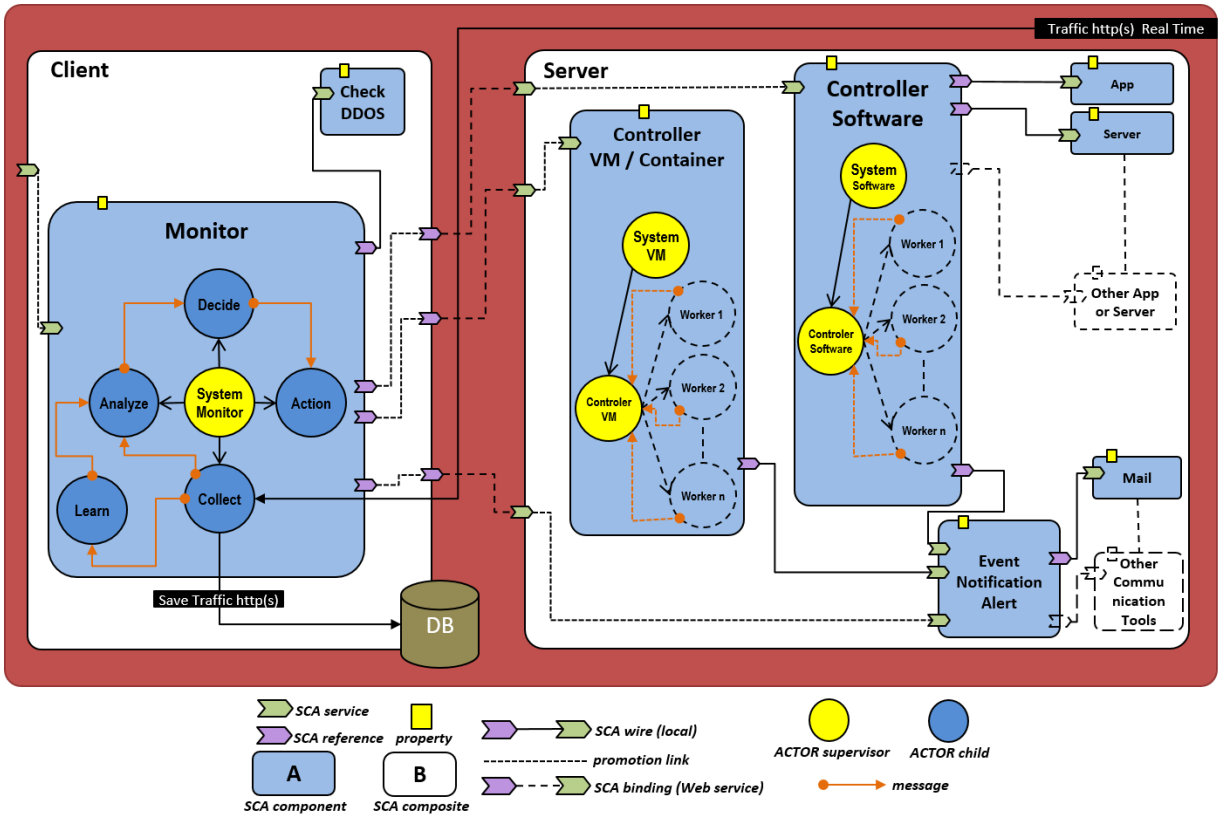

Figure 1: Overall architecture of PeakForecast

In this paper, to address this challenge, we propose an elastic distributed resource scaling approach for IaaS cloud infrastructures, named PeakForecast (PF). PF provide a medium-term resource demand prediction to absorb traffic surges before under-provisioning, Figure 1 shows its overall architecture.

Our mix approach consists, from a trace of queries received in the last seconds, minutes or hours: *i)* Detecting Potential Traffic Surges, *ii)* After detecting the traffic surge, forecasting the Upcoming Traffic by using a forecasting model, *iii)* Estimating the number resources required to absorb the remaining traffic to come, *iv)* Auto-scaling resources by quickly automatically adding resources to absorb traffic surges. This approach offers benefits for multi-tier applications that are already implemented using multiple VMs by improving the resource utilization between them as application demands vary.

We make the following contributions in this paper:

- We describe PeakForecast (PF), an elastic distributed resource scaling approach for IaaS cloud infrastructures that provide a medium-term resource demand prediction to absorb traffic surges before under-provisioning. Figure 1 shows its generic architecture. PF can efficiently scale cloud resources up and down to absorb such traffic surges.

- We describe a series of experiments that test the simple exponential smoothing (SES) prediction method that we use and compare predicted traffic with actual traffic and a set of alternative prediction algorithms, using a traffic surge that occurred on the Japanese version of Wikipedia the minutes before and after the tsunami on March 11th, 2011. The evaluation results demonstrate that SES is effective.

- We describe our resource allocation algorithm (PF algorithm).

- We design and implementation of a prototype elastic middleware solution, based on PF. We validate our approach by experimental results, by demonstrating that our prototype elastic middleware solution can provide spontaneous elasticity of resources for traffic surges in IaaS cloud infrastructures.

The generic architecture of PF based on the Service Component Architecture (SCA) standard, our resource allocation algorithm (PF algorithm) based on simple exponential smoothing (SES) prediction method and MAPE-K (Monitoring, Analysis, Planning, and Execution) loop, which auto-scaling resources by allocates or deallocates resources based on traffic of user requests.

The remainder of this paper is organized as follows. We discuss related works (cf. Section 2) in this domain. We introduce a case study highlighting the limitations of the state of the art (cf. Section 3) that we use to present our approach PF, we present the overall architecture of PF by describing its key components (cf. Section 4) and before validating our prototype elastic middleware solution by reporting on experimental results (cf. Section 5). Finally concluding (cf. Section 6).

2. RELATED WORKS

For a comprehensive study regarding the state of the art of elasticity in the cloud, We highlight the most prominent elasticity related solutions regarding (1) reactive approaches, (2) predictive approaches, and (3) mixed approaches. For a more detail review, refer to [62, 61, 58, 59, 60].

Reactive approaches such as threshold-based rules or policies are the most popular technique for automatic scaling in cloud computing (commercial systems such as provided by Amazon Web Services (AWS) [21], RightScale [22], Microsoft [36]), presumaly because of their apparent simplicity. The Amazon Web Services (AWS) [21] cloud platform offers static threshold scaling through specific metrics such as the CPU utilization. Amazon then monitors these metrics and scales when the set threshold rules are breached. Amazon offers a few different techniques on how the scale performs such as a fixed change, adding/negating from the current machine count or using a percentage change for the machines. Microsoft Azure [36] offering of static threshold-based rules runs the choice of two metrics, CPU utilization and the number of work tasks in the message queue [41]. RightScale offer the same core use of static threshold-based rules as the other two providers mentioned. RightScale have extended static threshold-based rules with RightScale's auto-scaling algorithm [37]. It is a simple democratic voting process whereby, if a majority of the VMs agree that they should scale up or down, that action is taken; otherwise no action occurs. Each VM votes to scale up or down based on a set of rules. After each scaling action, there is a period called the resize calm time (equivalent to the inertia or cooldown time), where no action can be performed. It prevents the algorithm from continually allocating resources as new instances boot [38]. As RightScale's voting system is based on rules,it has the same disadvantage [38]: the algorithm is highly dependent on user-defined threshold values, and therefore, to the workload characteristics. This was the conclusion reached by Kupferman et al. [39] after comparing RightScale with other algorithms. Static threshold-based rules presented the most simple and intuitive approach for auto scaling. However, setting the appropriate thresholds is a very tricky task, and can lead to instability in the system. Moreover, static thresholds become invalid if the behavior of the application changes suddenly during peak traffic. Setting the correct threshold also presents a challenge as this is done manually which requires knowledge of the application. If the threshold is wrong there is the risk of oscillation in the scaling, underutilization of resources or over utilization [40].

Predictive approaches use heuristics and analytical techniques together with historical data to predict future demand and proactively allocate resources. A number of different methods are used in the workload prediction. In [23], it develops a model-predictive algorithm for the workload prediction in which a second order autoregressive moving average method filter is used. PRESS [24] developed a hybrid online resource demand prediction model that combines a Markov model and a fast Fourier transform-based technique. Previous prediction schemes either focus on short-term prediction or need to assume cyclic workload patterns. Gmach et al. [25] used a Fourier transformbased scheme to perform offline extraction of long-term cyclic workload patterns. In comparison, PEAKFORECAST does not assume the workload is cyclic, and can predict resource demands for arbitrary workload patterns. Chen et al. [26] used sparse periodic auto-regression to perform load prediction. However, their approach is tailored towards long prediction intervals (e.g.,hours) and assumes that the repeating period is known in advance [24]. The experiments [24] have shown that auto-regression is computationally intensive, which makes it impractical for short-term online VM resource scaling. Saripalli et al. [27] use a two-step approach, using cubic spline interpolation combined with a hotspot detection algorithm for sudden spikes. Predicted values depend on the window width chosen. Hence, large amounts of available data are needed to produce confident predictions [28].

Mixed approaches. Mixed or Hybrid approaches combine reactive and predictive approaches to determine when to acquire resources over short and long time scales respectively (e.g., [54]). In this category, some approaches use predictive techniques for releasing resources and reactive techniques for acquiring resources(e.g., [55, 56]). Laura R. et al [55] presented a Platform Insights is a hybrid elasticity controller employing both reactive rule-based and predictive model-based elasticity mechanisms together in a coordinated manner. However, Platform Insights not handle multiple QoS objectives at once and not incorporate an algorithm to detect change in workload mix. In comparison, PEAKFORECAST detects change in workload mix. From a trace of requests received in the last seconds, minutes or hours, PEAKFORECAST detects if the underlying system is facing a traffic surge or not

Previous work has applied control theory [29], [30], [31] or reinforcement learning [32] to adaptively adjust resource allocations based on service level objectives (SLO) conformance feedback. However, those approaches often have parameters that need to be speciffed or tuned offine, and need some time to converge to the optimal(near-optimal) [24]. In comparison, PEAFFORECAST directly predicts resource allocation based on historical request time series.

Huber et al. [35] presented a self-adaptive resource management algorithm which leverages workload prediction and a performance model [34] that predicts application's performance under different configurations and workloads [33]. In comparison, PEAKFORECAST does not require any prior application knowledge.

Vertical vs. Horizontal scaling. In practice [61], existing solutions for auto-scaling enable horizontal scaling, i.e., acquiring or realizing node instances, while vertical scaling, i.e., increasing computing power of node instances, is not considered. It

has been attributed to impossibility of changing the size of nodes at the hypervisors level [62] [63]. SmartScale [57] proposed an auto-scaling method that minimized resource usage costs for bag-of-tasks jobs. It used horizontal scaling which added or removed VMs and vertical scaling which expanded or reduced the size of a VM. However, it is still deficient for resource requirements of dynamic workloads because it lacks consideration of resource usage during execution of an application. PeakForecast enables horizontal scaling.

It should be noted that the predictive models and the various theories mentioned above are mainly reserved for the scientific world. Indeed, industrial solutions prefer a reactive approach based on threshold-based rules. This is due to the fact that this type of solution is simpler to implement and more easily understandable by a non-scientific community that is struggling to apprehend theories that are too far from the industrial world. The main disadvantage of reactive techniques is that they do not anticipate unexpected changes in workload, and therefore, resources can not be provided in advance.

In order to handle traffic surges, the authors advocate either having spare servers that are always available (*i.e.*, over-provisioning), or finding a way to lower setup times. This is the case in [3], which considers that the architecture of the datacenter includes a *caching* tier consisting of machines that are always available. They temporarily use the resources of these machines to process requests as long as new servers are being deployed in the application tier. The elastic part is limited to the application tier, which is the largest consumer of CPU resources. When the load is high, [3] needs to be coupled with techniques like those in [13, 14, 15, 16] to minimize the damage caused by load spikes. However, by using PeakForecast, we do not have to pay for any additional resources, which is not the case when over-provisioning via spare servers. Furthermore, we used mixed approaches (reactive and predictive), which allows us to manage traffic surges in less than 60 seconds for case of VMs or in less than 10 seconds for case of Containers, in order to respond with the best reactivity as possible to user demands and maintaining the quality of service online site continuously.

There is also work, such as [17, 18], which addresses elastic control for multi-tier application services that allocate and release resources in discrete units, such as virtual server instances of pre-determined sizes. It focuses on an elastic control of the storage tier, in which adding or removing a storage node or *brick* requires rebalancing stored data across the nodes, and are not the focus of our paper.

Table 1: Summary of the PeakForcast comparison with existing works

References	Approaches		Method Scaling		Limitations
	Reactive	Predictive	Vertical	horizontal	
[21](2017)	x			x	Reactive approaches presented the most simple and intuitive approach for auto scaling. However, setting the appropriate thresholds is a very tricky task, and can lead to instability in the system. Moreover, static thresholds become invalid if the behavior of the application changes suddenly during peak traffic.
[22, 37](2017)	x			x	
[36](2017)	x			x	
[23](2011)		x		x	Previous prediction schemes either focus on short-term prediction or need to assume cyclic workload patterns. In comparison, PeakForecast does not assume the workload is cyclic, and can predict resource demands for arbitrary workload patterns
[24](2010)		x		x	Their approach is tailored towards long prediction intervals (e.g, hours) and assumes that the repeating period is known in advance.
[27](2011)		x		x	Predicted values depend on the window width chosen. Hence, large amounts of available data are needed to produce condent predictions.
[29, 30, 31](2009)		x		x	Those approaches often have parameters that need to be specified or tuned offine, and need some time to converge to the optimal(near-optimal) decisions. In comparison, PeakForecast directly predicts resource allocation based on historical request time series
[33](2013)		x		x	Presented a self-adaptive resource management algorithm which leverages workload prediction and a performance model that predicts application's performance under different configurations and workloads. In contrast, PeakForecast does not require any prior application knowledge.
[55](2013)	x	x		x	Platform Insights not handle multiple QoS objectives at once and not incorporate an algorithm to detect change in workload mix.In comparison,Peakforecast detects change in workload mix.
[57](2013)	x	x	x	x	it is still deficient for resource requirements of dynamic workloads because it lacks consideration of resource usage during execution of an application.

2.1 Our Contributions

We make the following contributions :

- We describe PeakForecast (PF), an elastic distributed resource scaling approach for IaaS cloud infrastructures that provide a medium-term resource demand prediction to absorb traffic surges before under-provisioning. Figure 1 shows its generic architecture. PF can efficiently scale cloud resources up and down to absorb such traffic surges.

- We describe a series of experiments that test the simple exponential smoothing (SES) prediction method that we use and compare predicted traffic with actual traffic and a set of alternative prediction algorithms, using a traffic surge that occurred on the Japanese version of Wikipedia the minutes before and after the tsunami on March 11th, 2011. The evaluation results demonstrate that SES is effective.

- We describe our resource allocation algorithm (PF algorithm).

- We design and implementation of a prototype elastic middleware solution, based on PF. We validate our approach by experimental results, by demonstrating that our prototype elastic middleware solution can provide spontaneous elasticity of resources for traffic surges in IaaS cloud infrastructures.

The generic architecture of PF based on the Service Component Architecture (SCA) standard, our resource allocation algorithm (PF algorithm) based on simple exponential smoothing (SES) prediction method and MAPE-K (Monitoring, Analysis, Planning, and Execution) loop, which auto-scaling resources by allocates or deallocates resources based on traffic of user requests.

3. OUR APPROACH

In this section, we illustrate our approach on a case study that consists in absorbing a traffic surge that occurred on the Japanese version of Wikipedia the minutes before and after the tsunami on March 11th 2011.

3.1 Detecting Potential Traffic Surges

In Figure 2, we can observe that, between 12:49 and 12:50, a traffic surge suddenly occurred because people constantly reported and consulted live information related to the disaster. One can observe that the intensity and the duration of such a surge are extreme compared to more traditional traffic loads. In particular, the short duration of this event makes it difficult to apply standard provisioning and deployment approaches to adjust the resources reactively. For example, threshold-based techniques cannot keep the pace with such a situation as the amplitude of the surge may lead to inappropriate decisions.

The intensity of such a traffic surge depends on a coefficient C_s that can be defined as:

$$C_s = \frac{\text{Number of requests at } P_n}{\text{Number of requests at } P_{n-1}}$$

The highest C_s, the more intense the traffic surge. Figure 3 illustrates the evolution of C_s between 12:28 and 13:22 on March 11th, and detecting a surge suspicion around 12:49.

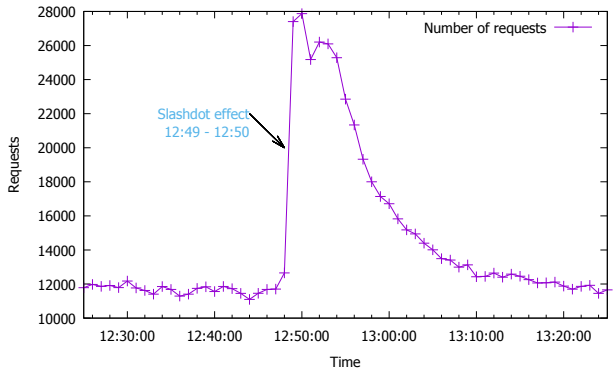

Figure 2: Wikipedia request throughput on March 11th 2011.

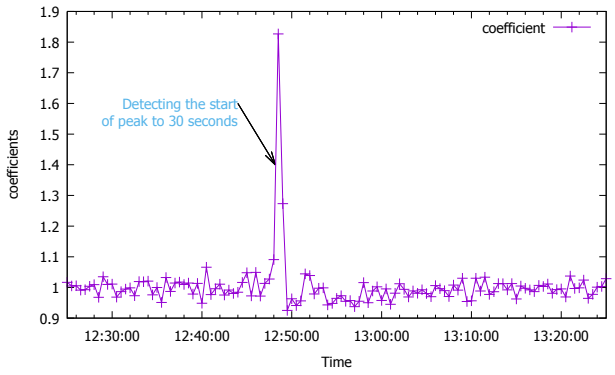

Figure 3: Evolution of the surge coefficient on March 11th 2011.

3.2 Forecasting the Upcoming Traffic

After detecting the traffic surge, we need to predict the traffic to be absorbed in the upcoming seconds. We need a method which can adjust its model quickly according to the variation trend of traffic. As a result of these, in this paper, we use a *simple exponential smoothing*(SES) prediction method for time series [1] to solve this problem. The *simple exponential smoothing* model is one of the most popular forecasting methods. Its principle is to give more importance to recent observations. It is therefore more reactive than rolling averages or regression models because it takes into account a quickly changing trend. It applies to time series without trend and seasonality, but with a change of level (abrupt as it is the case with slashdot effects). The procedure gives heaviest weight to more recent observations and smaller weight to observations in the more distant past. The *simple exponential smoothing* provides a short-term prediction at P_{n+1}. Given an observed time series defined as $y_1, y_2, ..., y_n$, the forecasting formula is therefore defined as:

$$\hat{y}_{i+1} = \alpha.y_i + (1-\alpha).\hat{y}_i, 0 < \alpha < 1, i > 0 \quad (1)$$

Where y_i is the actual known series value for time period i, $\hat{y}_i$ is the forecast value of the variable Y for time period i, $\hat{y}_{i+1}$ is the forecast value for time period $i+1$ and α is the *smoothing constant.* α is the weight assigned to the most recent observations in the time series. Essentially, the forecasting model is based on the actual value for this period, and the value forecast for this period, which recursively was estimated from forecasts for previous periods. Equation (1) can therefore be rewritten as:

$$\hat{y}_{i+1} - \hat{y}_i = \alpha.(y_i - \hat{y}_i) \quad (2)$$

Thus, the accuracy of the forecast value is proportional to the forecasting error, that is:

$$e_i = y_i - \hat{y}_i \quad (3)$$

Where residual e_i is the forecasting error for time period i. Thus, we choose α minimizing $\sum e_i^2$.

We applied the simple exponential smoothing method on the requests throughput in a time interval enclosing the traffic surge (cf. Table 2). In Table 2, the first column are the intervals of 30 seconds from 12:46:30 to 12:53:30; for example #1=[12:46:30 - 12:47:00], #2=[12:47:30 - 12:47:30], #3=[12:47:30 - 12:48: 00]. The second column, the number of requests received in the time interval. Columns 3 to 5 are the predictions of requests for each of its *smoothing constant*t values; For example in the column for α=0.9, at interval #3=[12:47:30 - 12:48:00], the query number is 5,890, however the value that was predicted is 5,830 and the prediction of the next interval is 5,884. The table 3 are the squared of forecasting error per interval. Similarly for Table 4 and **??**.

Due to the recurring formula of simple exponential smoothing, we must choose a value from which forecasts can be applied. This value is of little importance if the series is long and in our case (cf. Table 2), we chose the first observation—*i.e.* the number of requests for the interval #1 (5,981). Concerning the choice of the *smoothing constant* α, which directly affects the predictions, no direct mathematical method gives its optimum value. For this, it is obviously necessary to rely on data stored and therefore, in our case, based on requests history. Thus, we have compared three values of α—0.1, 0.4, and 0.9—to observe their effect on the predictions of our forecasting model.

Period Id	# of requests	Forecast α=0.4	Forecast α=0.1	Forecast α=0.9
1	5,981	5,981	5,981	5,981
2	5,813	5,981	5,981	5,981
3	5,890	5,914	5,964	5,830
4	6,051	5,904	5,957	5,884
5	6,600	5,963	5,966	6,034
6	12,056	6,218	6,030	6,543
7	15,349	8,553	6,632	11,505
8	14,198	11,271	7,504	14,965
9	13,674	12,442	8,173	14,275
10	12,870	12,935	8,723	13,734
11	12,306	12,909	9,138	12,956
12	12,851	12,668	9,455	12,371
13	13,351	12,741	9,794	12,803
14	13,065	12,985	10,150	13,296
15		13,017	10,441	13,088

Table 2: Traffic predictions per interval of 30 seconds from 12:46:30 to 12:53:30 on March 11th 2011.

Id	# of requests	Residual α=0.4	Residual α=0.1	Residual α=0.9
1	5,981			
2	5,813	28,224	28,224	28,224
3	5,890	566	5,505	3,624
4	6,051	21,527	8,877	27,896
5	6,600	405,810	401,700	320,019
6	12,056	34,084,803	36,317,716	30,388,430
7	15,349	46,184,685	75,982,191	14,778,312
8	14,198	8,564,747	44,810,958	587,636
9	13,674	1,517,665	30,257,578	360,789
10	12,870	4204	17,194,457	746,610
11	12,306	363,492	10,035,958	423,029
12	12,851	33,583	11,533,918	230,361
13	13,351	372,045	12,649,019	300,300
14	13,065	6,396	8,496,592	53,454
$\sum e_i^2$		9.15×10^7	2.47×10^8	4.82×10^7

Table 3: Residuals $e_i^2 = (y_i - \hat{y}_i)^2$ of Table 2.

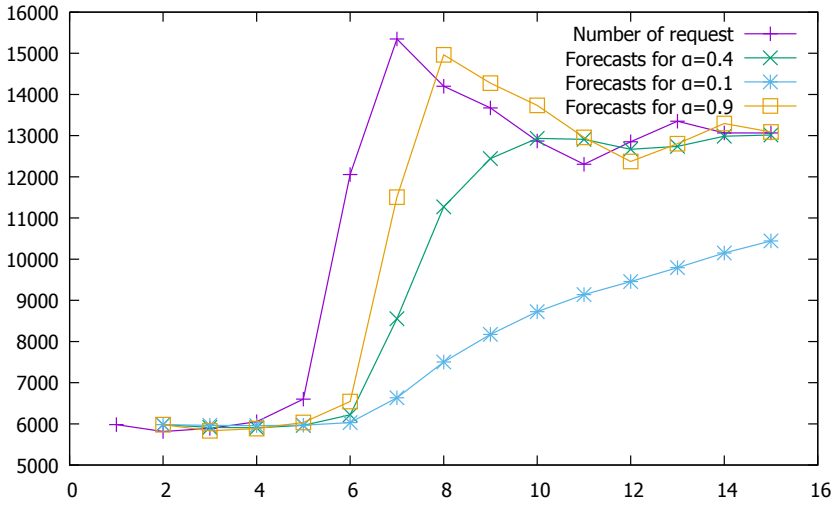

Figure 4: Traffic predictions from 12:46:30 to 12:53:30 on March 11th 2011.

From Table 3, we can observe that the *smoothing constant* minimizing $\sum e_i^2$ and provides predictions with an acceptable precision—*i.e.*, which is very close to reality—is $\alpha = 0.9$. This can be acknowledged in Figure 4.

With this *constant smoothing* of 0.9, when we detect a traffic surge at the interval #6 (cf. Table 2), we predict an increase of $11,504$ requests during the next period. Adding this estimated traffic to previous requests, we obtain a forecast traffic of $23,560$ requests from the 30^{th} second of the minute marking the traffic surge, as shown in Figure 2.

This prediction is close to the peak of real traffic, which reached $27,405$ requests per minute (adding the interval number 6 and 7 of Table 2), with an average of 457 requests per second.

We compare predicted traffic with actual traffic and a set of alternative prediction algorithms. We applied these methods on the requests throughput in a time interval enclosing the traffic surge (cf. Table 2). The *Double Exponential Smoothing*(DES) [42] like regular exponential smoothing, except includes a component to pick up trends. The *Holt–Winters*(HW) [42] is a direct extension of Holt's [42] method for trended data that models the seasonal variation in a time-series as either a multiplicative component, or an additive component. The *Moving Average*(MA) [42] is a calculation to analyze data points by creating series of averages of different subsets of the full data set. The *Fast Fourier transform* algorithms for smoothing(FFS) [43] is to transform a time series into its Fourier coordinates, then remove part of the higher frequencies, and then transform the coordinates back to a signal. This new signal is a smoothed series. The *Autoregressive Integrated Moving Average*(ARIMA) model [42] is generally referred to as an ARIMA(p,d,q) model where parameters p,d and q are non - negative integers that refer to the order of the autoregressive, integrated, and moving average parts of the model respectively.

Period Id	# of requests	SES α=0.9	MA	Holt-Winters α=0.2 β=0.2	DES α=0.9	ARIMA (1,1,1)	FFS p =0.3
1	5,813	5,918	5,918	5,918	5,918	5,918	7,761
2	5,890	5,813	6,088	5,813	5,913	5,833	5,277
3	6,051	5,882	7,282	5,831	6,597	5,947	5,281
4	6,600	6,034	9,189	5,887	6,257	6,184	7,887
5	12,056	6,543	10,850	6,070	7,075	7,440	11,620
6	15,349	11,504	12,375	7,547	16,519	15,725	14,431
7	14,198	14,964	13,629	9,699	18,925	14,698	15,089
8	13,674	14,274	13,679	11,370	13,980	13,816	13,932
9	12,870	13,734	13,179	12,695	13,164	13,542	12,448
10	12,306	12,956	13,010	13,601	12,121	12,300	12,032
11	12,851	12,371	12,888	14,161	11,702	12,379	12,881
12	13,351	12,803	12,927	14,665	13,168	13,260	13,867
13	13,065	13,296	13,083	15,117	13,825	13,384	13,487
14	13,065	13,088	13,160	15,338	12,933	12,791	11,141

Table 4: Traffic predictions per interval of 30 seconds from 12:46:30 to 12:53:30 on March 11th 2011 with a set of alternative prediction algorithms.

Period Id	# of requests	Residual SES	Residual MA	Residual Holt-Winters	Residual DES	Residual ARIMA	Residual FFS
1	5,813	11,025	11,025	11,025	11,025	11,025	3,796,835
2	5,890	5,929	39,402	5,929	566	3,204	375,483
3	6,051	28,459	1,515,361	48,189	298,650	10,811	592,394
4	6,600	320,208	6,703,956	508,019	117,320	172,298	1,657,129
5	12,056	30,388,615	1,452,507	35,830,198	24,810,053	21,302,714	189,789
6	15,349	14,778,324	8,842,296	60,869,053	1,369,441	141,573	841,907
7	14,198	587,635	323,305	20,237,500	22,352,622	250,898	794,423
8	13,674	360,789	29	5,304,110	94,168	20,412	66,677
9	12,870	746,609	95,976	30,452	86,491	452,429	177,928
10	12,306	423,028	496,179	1,677,856	33,947	30	74,931
11	12,851	230,360	1,413	1,717,069	1,319,720	222,275	906
12	13,351	300,299	179,267	1,729,210	33,458	8,153	267,244
13	13,065	53,453	324	4,211,091	578,976	101,876	178,655
14	13,065	534	9,088	5,169,668	17,421	74,794	3,701,438
Residuals $\sum e_i^2 = \sum(y_i - \hat{y}_i)^2$		4.82×10^7	1.96×10^7	1.37×10^8	5.11×10^7	2.27×10^7	1.27×10^7

Table 5: Residuals $\sum e_i^2 = \sum(y_i - \hat{y}_i)^2$ of Table 4.

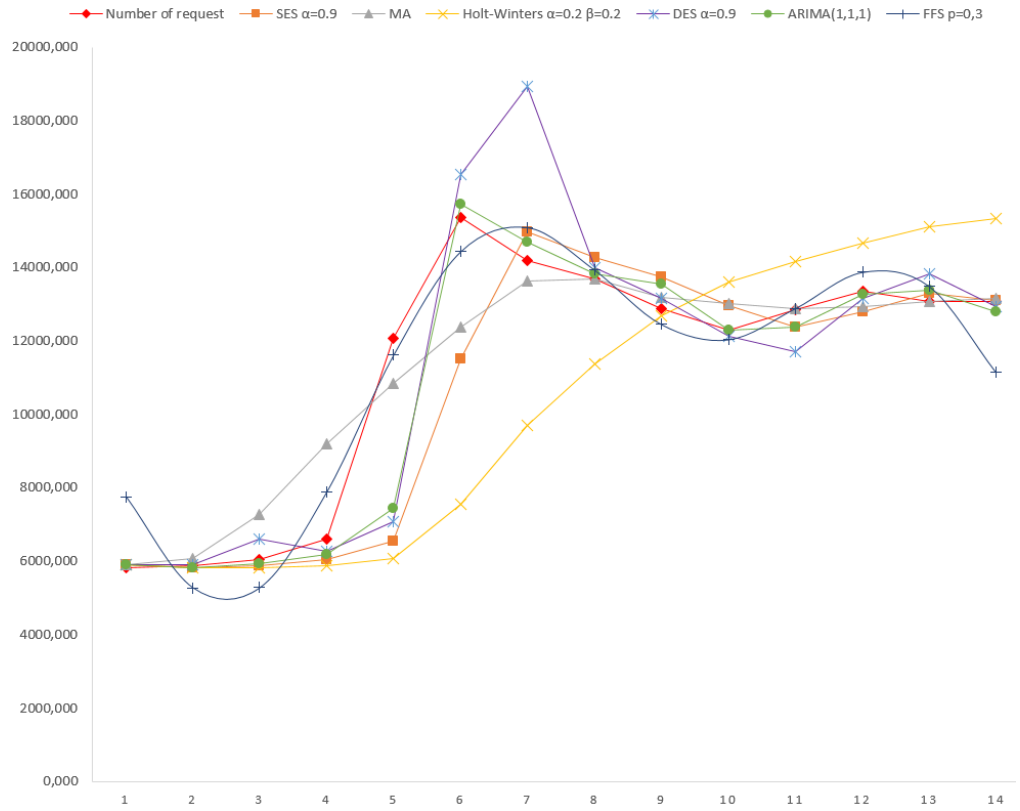

Figure 5: Methods Traffic predictions from 12:46:30 to 12:53:30 on March 11th 2011 with a set of alternative prediction algorithms.

Experimental results are shown in Figure 5 . We can see that in most situations, the *Simple Exponential Smoothing*(SES) outperforms other methods, and has a lowest prediction error. *Simple exponential smoothing* can adjust its function with the traffic trend, so it can predict the traffic timely and accurately. By contrast, *Double Exponential Smoothing*(DES) is higher than the actual traffic most of the time, *Holt-Winters* prediction is less sensitive of traffic variations, *ARIMA* prediction is always dragging a little behind, *Moving Average*(MA) prediction is less sensitive of traffic variations. *FFS* prediction is always dragging a little behind and is higher than the actual traffic most of the time.

3.3 Estimating the resource requirements

While the duration of a traffic surge cannot be predicted, another key constant to be considered is the time required to bring a computing resource to be ready to process request.

Case of *virtual machines* (VMs): This includes the time to provision a virtual appliance, the time to install an application (*e.g.*, MediaWiki takes 10 seconds on average to install), a server or middleware platform (*e.g.*, MongoDB takes 11 seconds on average to install) on a virtual machine. By taking into account this delay and the maximum throughput supported by the considered components, we can estimate the number of virtual appliances and application stacks required to absorb the traffic surge along the upcoming period. In particular, we use the following formula where the number of virtual appliances to deploy N is estimated as:

$$N = \frac{S_{t+1} - Y_t}{X}$$

Case of *Docker containers* (Containers): We use the following formula where the number of virtual appliances to deploy N is estimated as:

$$N = \frac{S_{t+1}}{X}$$

Where S_{t+1} is the *estimated traffic* at time $t+1$, Y_t is the *monitored traffic* at time t, X is the *maximum throughput* supported by a given application stack, about 600 requests per second in our case study (this value is obtained from benchmarks). The functions respectively calculating the number of resources to provision for VMs and containers are different, because in containers the number of resources calculate includes resources in use. We do it in relation to the Kubernate[2] command which allows us to scale the number of containers(For example, with command *$ Kubectl scale —-replicas = 3* , if initially there are 2 containers then Kubernate will only add 1 container).

4. PEAKFORECAST MIDDLEWARE

In the context of virtual infrastructures (cf. Figures 8 and 9), PeakForecast is aelastic middleware solution we developed to automatically address the elasticity issues imposed by traffic surges not only by adding resources to absorb traffic surges, but also to release these resources once the load decreases. PeakForecast therefore monitors the incoming traffic, makes predictions on the volume of this traffic and adjust the size of the infrastructure according to these predictions. Adding more resources mainly concerns instances of *Virtual Machines* (VMs) or *Docker containers*[3] (Containers) hosting the presentation layer and the business logic tier of the website, because they are usually considered as the bottleneck caused by traffic surges. In particular, we use Fabric[4], a Python library for streamlining the use of SSH, for operating the VMs deployment or the replication containers. Once instances of VMs are deployed, the load balancer (here Nginx[5]) is automatically reconfigured to consider the new instances made available.

In this section, we provide a detailed overview of our tool, PeakForecast, in a virtual web infrastructure (Figure 8 and 9). PeakForecast is a distributed software system based on the SCA standard (see Figure 6). The *Service Component Architecture* (SCA) standard is a set of specifications for building distributed applications based on *Service–Oriented Architectures* (SOA) and *Component–Based Software Engineering* (CBSE) principles [2]. We use the FraSCAti[6] middleware platform, which enables the development and the execution of distributed SOA applications based on SCA. To build an efficient and scalable solution, we built PeakForecast with the Akka event-driven middleware framework.[7] Akka is a toolkit for building highly concurrent, distributed, and fault tolerant event–driven applications on the JVM. Akka implements a programming model based on actors. The strength of Akka is its high performance: 50 million messages processed per second on a single machine, and small memory footprint; 2.7 million actors per GB of heap.

As illustrated in Figure 6, PeakForecast is mainly composed of two components. The first component exposes a web service to administrate the VMs or containers of the virtual web infrastructure. It contains four essential components ControllerSoftware, ControllerVM/Container, ControllerKubectl and EventNotificationAlert. The component ControllerSoftware provides services to *install, uninstall, status, reconfig, stop, start, restart* a component on one or more VMs. Here, one component may be a server or an application. In case the service must be running on multiple VMs, the operation is performed by the actors Worker"n" to operate in parallel and coordinated by the supervisor actor ControllerSoftware. The component ControllerVM/-Container provides services to *start, restart, pause, resume, clone, create, delete* VMs and Containers. All these services are applied to one or more VMs and Containers. The component ControllerKubectl provides services to replicate Containers. The component EventNotificationAlert provides services to send emails, tweets and SMS notifications or to alert to administrators of the virtual web infrastructure. The administrators can receive a notification message even if they are on the move to keep being informed and up-to-date about the condition of virtual web infrastructure.

[2] http://kubernetes.io
[3] https://www.docker.com
[4] http://fabfile.org
[5] http://wiki.nginx.org
[6] http://frascati.ow2.org
[7] http://akka.io

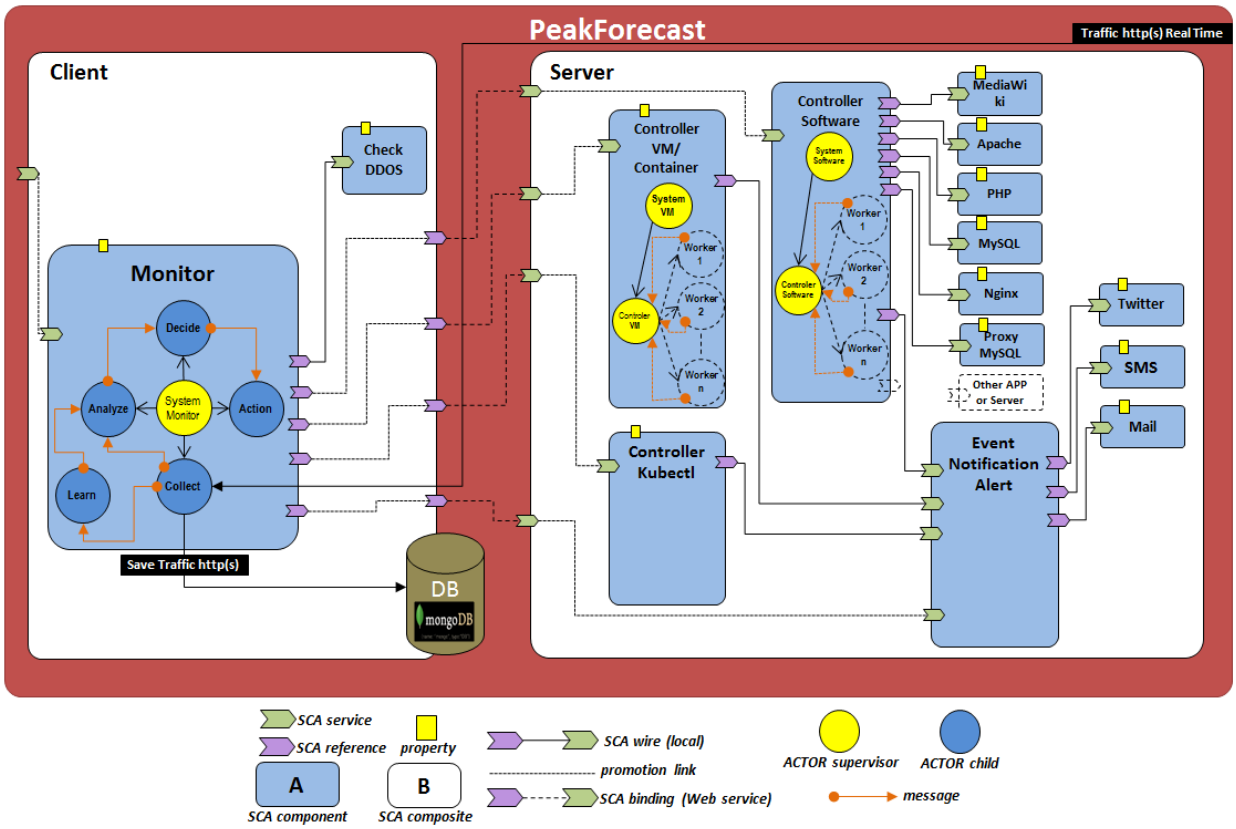

Figure 6: Architecture of the PeakForecast Middleware.

The second component allows PeakForecast to monitor HTTP traffic, detecting traffic surges and reacting accordingly by consuming the services provided by the first component. It contains two essential components: Monitor and CheckDDOS. The component Monitor implements a MAPE-K loop [19], and is mainly composed of five actors: Collect, Learn, Analyse, Decide, *Act*(Action), communicating with each others by exchanging messages (see Figure 7). The actor Collect collects traffic information per time interval. The number of requests collected by the actor Collect are sent as a message to the actor Analyse and actor Learn. The actor Analyse analyzes the number of requests to detect when we are facing a traffic surge (see Section 2.1), or the prediction gives an estimate of upcoming traffic (see Section 2.2). It is within this actor that we implemented the *simple exponential smoothing* formula. With this prediction of the upcoming traffic, PeakForecast estimates the number of VMs or Containers to absorb the traffic surge (see Section 2.3). The actor Analyse sends the number of VMs or Containers as a message to the actor *Decide*. Based on the latency of VMs in use, the actor *Decide* determines the exact number of VM or container instances to deploy in order to absorb the traffic surge, verifies if the virtual web infrastructure is facing a *Distributed Deny of Service* (DDoS), and then sends this number to the actor Act(Action), which takes care of making available the requested VM or container instances, allowing the virtual web infrastructure to anticipate the traffic surge ahead. The actor Learn allows calculation (learning process) to part of a historic request (*e.g.*, 30 numbers of requests received), to determine the best *smoothing constant* α and to send α as a message to the actor Analyse in order to increase and maintain the reliability of prediction model. In order to manage the positive faults, the actor Learn which allows us to part of a historical request (eg 30 numbers of requests received), to determine at each moment, the best smoothing constant of the moment that minimizes the sum of the squares of the last third of the errors. Thus, the value of the smoothing constant is dynamically updated in order to increase and maintain the reliability of prediction model.

The component CheckDDOS allows the component Monitor, to check if a server is under DDoS before adding resources within a virtual web infrastructure. This architecture of PeakForecast (cf. Figure 6) is an architecture that can be reused for any Cloud applications that can be horizontally scaled by spawning new replicas.

The monitoring algorithm of our autonomous system PeakForecast is implemented by the component *Monitor*, which coordinates the activation of the various components of the architecture in order to quickly arrive at an optimal preference decision. To provide this decision-making capacity, the monitoring algorithm uses active rules or ECA (*Event-Condition-Action*) rules, widely studied in the field of active databases.

intervalTime	Represents the time interval the actor Collect will use to collect requests. For example, every 30 seconds, 1 minute ...
intervalHistoryRequests	Represents the interval of the last analysis requests that the actor Learn will use during the automatic learning process.
constantSmoothing	Represents the smoothing constant of the predictive model by the Simple Exponential Smoothing method.
surgeCoefficientThreshold	Represents the highest C_s, the more intense the traffic surge (cf. SubSection 2.1)
numberDetectedPeak	Represents the number of detected traffic surges with an added VMs or Containers action.
numberRequestCurrent equal *numberRequestPrevious*	Represents the number of request collected after an *intervalTime* time recorded in the *numberRequestCurrent* variable. *NumberRequestPrevious* contains the previous value of *numberRequestCurrent*. Initially they have the same value.
listNumberRequestBeforePeak	Represents the list containing the values of the number of requests collected before the detected traffic peaks.
listNumberVMsUp	Represents the list of VMs or Containers numbers added by the actor Act(action).
scaleUp	Allows to trigger the action of adding the VMs or Containers.
scaleDown	Allows to trigger the removal action of the VMs or Containers.

Table 6: List of variables.

Monitoring Algorithm 1 describes a control loop mechanism that runs as long as the virtual infrastructure is active. *Line 3* (collect) corresponds to the collection of the number of requests per time interval (for example one second, 15 seconds, 30 seconds, 1 minute) of the virtual infrastructure in real time. *Line 4* (learn) corresponds to the analysis (automatic learning process) of the numbers of requests collected in order to improve the predictions or resource estimates needed to absorb the peak of future traffic. *Line 5* (anlyze) corresponds to the analysis of the number of requests as they are collected; When an upcoming peak is detected, an estimate of the number of resources (VMs or Docker containers) needed to absorb the upcoming traffic peak is returned. *Line 6* (decision) corresponds to the decision of the exact number of resources (VMs or Docker containers) to add to absorb the peak of traffic depending on the customer profile of the virtual infrastructure and resources in use. *Line 7* (action) corresponds to the execution of adding or removing a number of resources (VMs or Docker containers). The functions collect, learn, analyze, decision, execute are described respectively in *Algorithms 3, 4, 5, 6*. *Algorithm 3* describes a sequence of collects traffic information per time interval, save in the database, and which are returned. *Algorithm 4* describes a sequence that allows , to part of a historic request (*e.g.*, 30 numbers of requests received), to determine (learning process), the best *smoothing constant* α and and return it. *Algorithm 5* describes a sequence that corresponds to the analysis of the number of requests as they are collected; When an upcoming peak is detected, an estimate of the number of resources (VMs or Docker containers) needed to absorb the upcoming traffic peak and return it. *Algorithm 6* describes a sequence that corresponds to the decision of the exact number of resources (VMs or Docker containers) to add to absorb the peak of traffic depending on the customer profile of the virtual infrastructure and resources in use and return it.

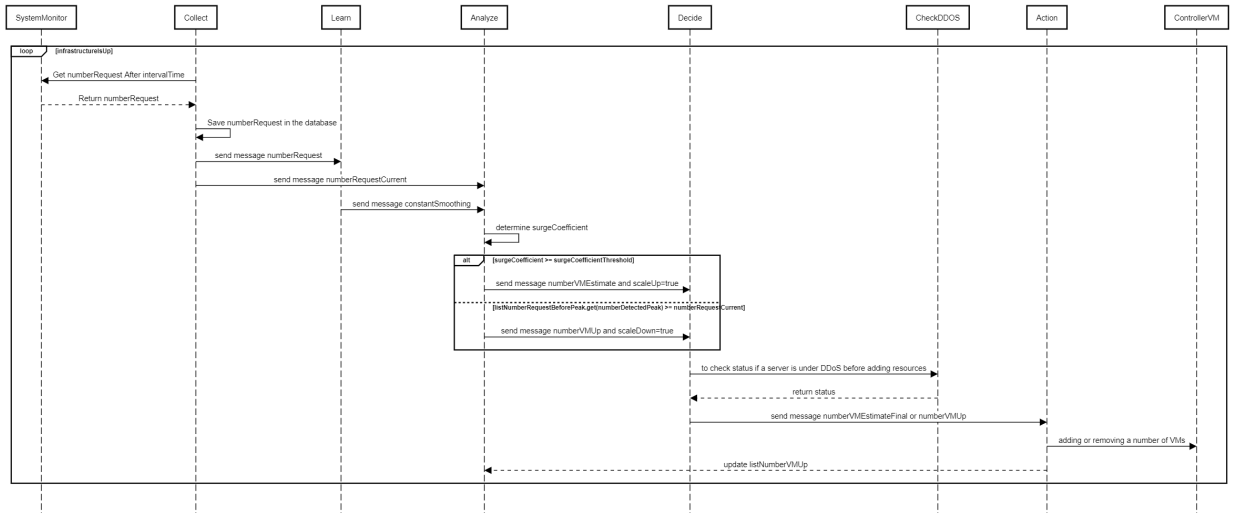

Figure 7: Coordination between PeakForecast Middleware's components.

Algorithm 1 Supervision of the Autonomous System PeakForecast

Require: INITIALIZATIONVARIABLES()
while $infrastructureIsUp$ **do**
  $numberRequestCurrent \leftarrow$ COLLECT($intervalTime$)
  $constantSmoothing \leftarrow$ LEARN($intervalHistoryRequests$)
  $numberVMEstimate \leftarrow$ ANALYZE($numberRequestCurrent, numberPreviousRequest, constantSmoothing$)
  $numberVMEstimateFinal \leftarrow$ DECIDE($numberVMEstimate$)
  ACTION(numberVMEstimateFinal)
  $numberRequestPrevious \leftarrow numberRequestCurrent$
end while

Algorithm 2 Variable Initialization Procedure

procedure INITIALIZATIONVARIABLES()
  $intervalTime \leftarrow 30$ ▷ 30 secondes for example
  $intervalHistoryRequests \leftarrow 100$ ▷ 100 requests for example
  $constantSmoothing \leftarrow 0.9$ ▷ 0.9 for example based on requests history
  $surgeCoefficientThreshold \leftarrow 2$
  $numberDetectedPeak \leftarrow 0$
  $numberRequestCurrent = numberRequestPrevious$
  $listNumberRequestBeforePeak \leftarrow NULL$ ▷ empty list of number request detects before traffic peak
  $listNumberVMsUp \leftarrow NULL$ ▷ empty list of VMs add to absorb traffic peak
  $scaleUp \leftarrow false$
  $scaleDown \leftarrow false$
end procedure

Algorithm 3 Collection function of real-time request

function COLLECT($intervalTime$)
  Get $numberRequest$ After $intervalTime$
  Save $numberRequest$ in the database
  return $numberRequest$
end function

Algorithm 4 Learning function on the number of requests collected

function LEARN($intervalHistoryRequests$)
  With the $intervalHistoryRequests$, calculate the best $constantSmoothing$ of simple exponential smoothing method for prediction
  return $constantSmoothing$
end function

Algorithm 5 Function to analyze the number of requests collected in real time

function ANALYZE($numberRequestCurrent, numberRequestPrevious, constantSmoothing$)
  $surgeCoefficient \leftarrow numberRequestCurrent/numberRequestPrevious$
  if $surgeCoefficient \geq surgeCoefficientThreshold$ **then**
    $numberDetectedPeak \leftarrow numberDetectedPeak + 1$
    $listNumberRequestBeforePeak.add(numberRequestPrevious)$
    With the simple exponential smoothing prediction method and the parameters $numberRequestCurrent$, $constantSmoothing$, calculate the $numberRequestFuture$
    With the $numberRequestFuture$ estimating the number of VMs $numberVMEstimate$ requirements to absorb traffic surge
    $scaleUp \leftarrow true$
    return $numberVMEstimate$
  end if
  if $listNumberRequestBeforePeak.isNotEmpty()$ **then**
    if $listNumberRequestBeforePeak.get(numberDetectedPeak) \geq numberRequestCurrent$ **then**
      $numberVMUp \leftarrow listNumberVMUp.get(numberDetectedPeak)$
      $scaleDown \leftarrow true$
      return $numberVMUp$
    end if
  end if
end function

Algorithm 6 Decision-making function regarding the number of VMs or Containers to be added or removed

```
1: function DECIDE(numberVMEstimate)
2:     if scaleDown = true then
3:         return numberVMEstimateFinal
4:     else
5:         With latency VMs running, the IAAS customer profil, numberVMEstimate, determine the final number of VMs
   numberVMEstimateFinal
6:         return numberVMEstimateFinal
7:     end if
8: end function
```

5. VALIDATION

As a matter of validation, the virtual web infrastructure (see Figures 8 and 9) has been developed.

Case of the VMs (cf. Figure 8): The physical machine uses Virtual Box[8] as an hypervisor and has the following characteristics: CORE i3 CPU 2.50GHz with 4 GB of RAM, and for each VM: CPU 1 GHz with 500 MB of RAM. A VM front-end hosts the load balancer NGINX[9] responsible for distributing incoming HTTP requests among the VMs hosting the MEDIAWIKI[10] website (the business logic tier of the website). While another VM contains the MYSQL load balancer responsible for distributing SQL queries to the VMs hosting the databases (the data layer of the website). As illustrated in Figure 8, MYSQL Proxy [11] redirects read SQL queries to MYSQL server master and slaves, while write queries are processed by the MYSQL server master before forwarding the master replica to slaves.

Case of the Containers (cf. Figure 9): The physical machine uses Kubernetes[12] for orchestrating, managing Docker containers and has the following characteristics: CORE i3 CPU 2.50GHz with 4 GB of RAM, and for each Container: CPU 1 GHz with 500 MB of RAM. The block "frontend-controller" which contains the container "*PeakForecastClient NginxProxy*", allows to monitor HTTP traffic, detect traffic surges and react accordingly.

For the purpose of this validation, adding resources concerns instances of VMs or Containers containing MEDIAWIKI, because they are considered as the bottleneck in traffic surges. To perform our benchmark, we used the tool SIEGE[13]. SIEGE can stress a single URL with a user-defined number of simulated users, or it can read many URLs into memory and stress them simultaneously. The program reports the total number of hits recorded, bytes transferred, response time, concurrency, and return status. Siege supports HTTP/1.0 and 1.1 protocols, the `GET` and `POST` directives, cookies, transaction logging, and basic authentication.

Without PeakForecast: we use a single VM or Container (`apache1`) containing an instance of MEDIAWIKI and thus treating all HTTP requests sent by the NGINX load balancer. After some minutes, the web site becomes overloaded, slows down and even becomes temporally unavailable.

With PeakForecast: In Figure 10(a), we also have a single VM (`apache1`) containing an instance of MEDIAWIKI and thus serving all HTTP requests sent by NGINX load balancer. PEAKFORECAST quickly detects that the infrastructure is facing a traffic surge, predicts the upcoming traffic, estimates the number of VMs required to absorb the traffic surge (2 VMs: apache 2 & 3) and starts the process of provisioning these VMs. In Figure 10(b), PEAKFORECAST installs MEDIAWIKI on VMs `apache 2 & 3` (in approximately 10 seconds), then add the VMs in the configuration file of the NGINX load balancer to make them visible to NGINX. Figure 10(c) reports treatments, such as requests sent by VMs `apache 2 & 3`, which explains the evolution of CPU and RAM in VM `apache1`. In Figure 10(d), PEAKFORECAST removes the VMs `apache 2 & 3` once the traffic surge has been absorbed.

We also have a single container (`apache1`) containing MEDIAWIKI and thus serving all HTTP requests sent by frontend-controller. In Figure 11(b) PEAKFORECAST quickly detects that the infrastructure is facing a traffic surge, predicts the upcoming traffic, estimates the number of Containers required to absorb the traffic surge (2 Containers: `apache 2 & 3`) and starts the process of provisioning these Containers by replication mediawiki-controller (in approximately 5 seconds). Figure 11(c) reports treatments, such as requests sent by Containers `apache 2 & 3`, which explains the evolution of CPU and RAM in container apache 1. In Figure 11(d), PEAKFORECAST removes the Container `apache 2 & 3` once the traffic surge has been absorbed.

[8] `https://www.virtualbox.org/wiki/Downloads`
[9] `http://nginx.org/en/download.html`
[10] `https://www.mediawiki.org/wiki/Download`
[11] `http://download.nust.na/pub6/mysql/downloads/mysql-proxy/index.html`
[12] `http://kubernetes.io`
[13] `http://www.joedog.org/siege-home`

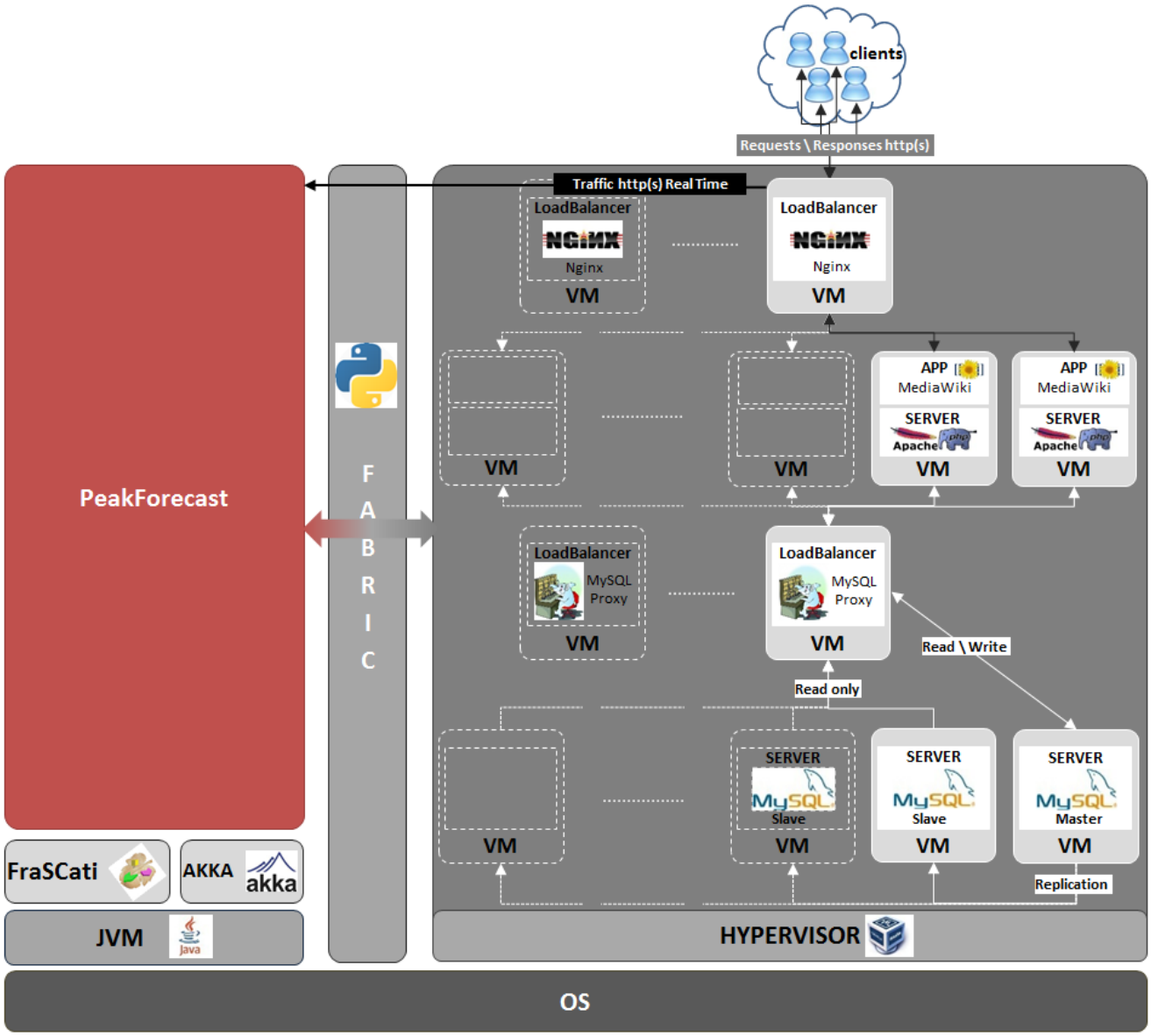

Figure 8: PeakForecast Infrastructure for VMs.

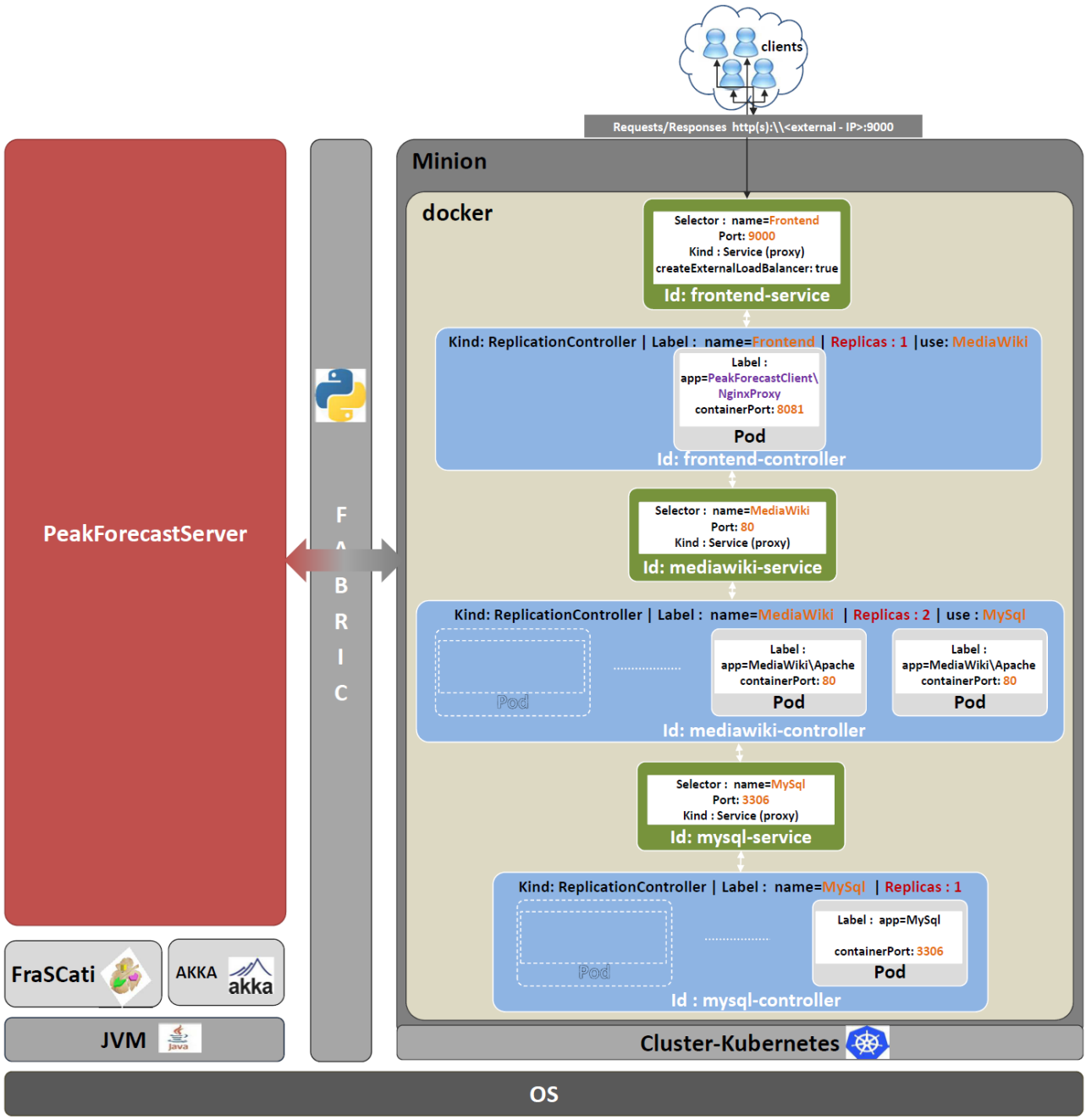

Figure 9: PeakForecast Infrastructure for Containers.

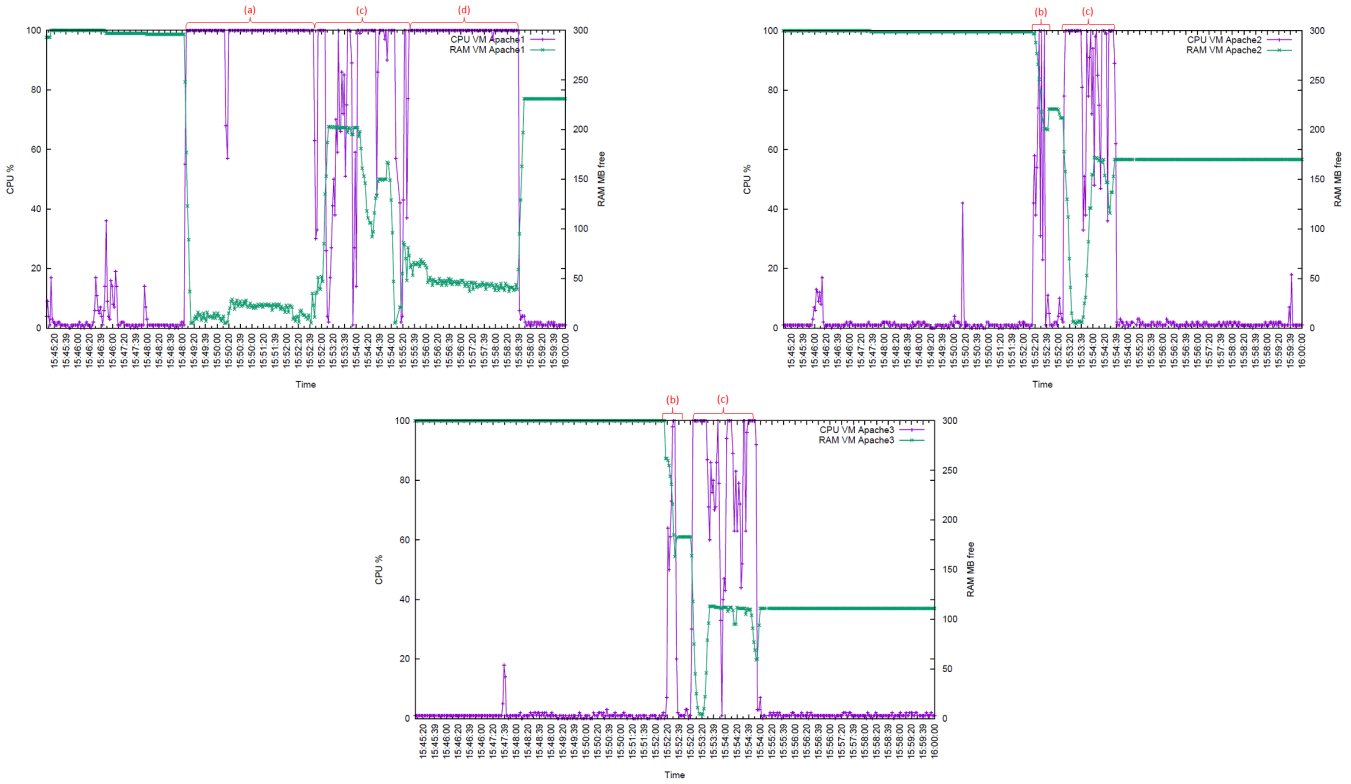

Figure 10: Benchmark 1 with PeakForecast VMs

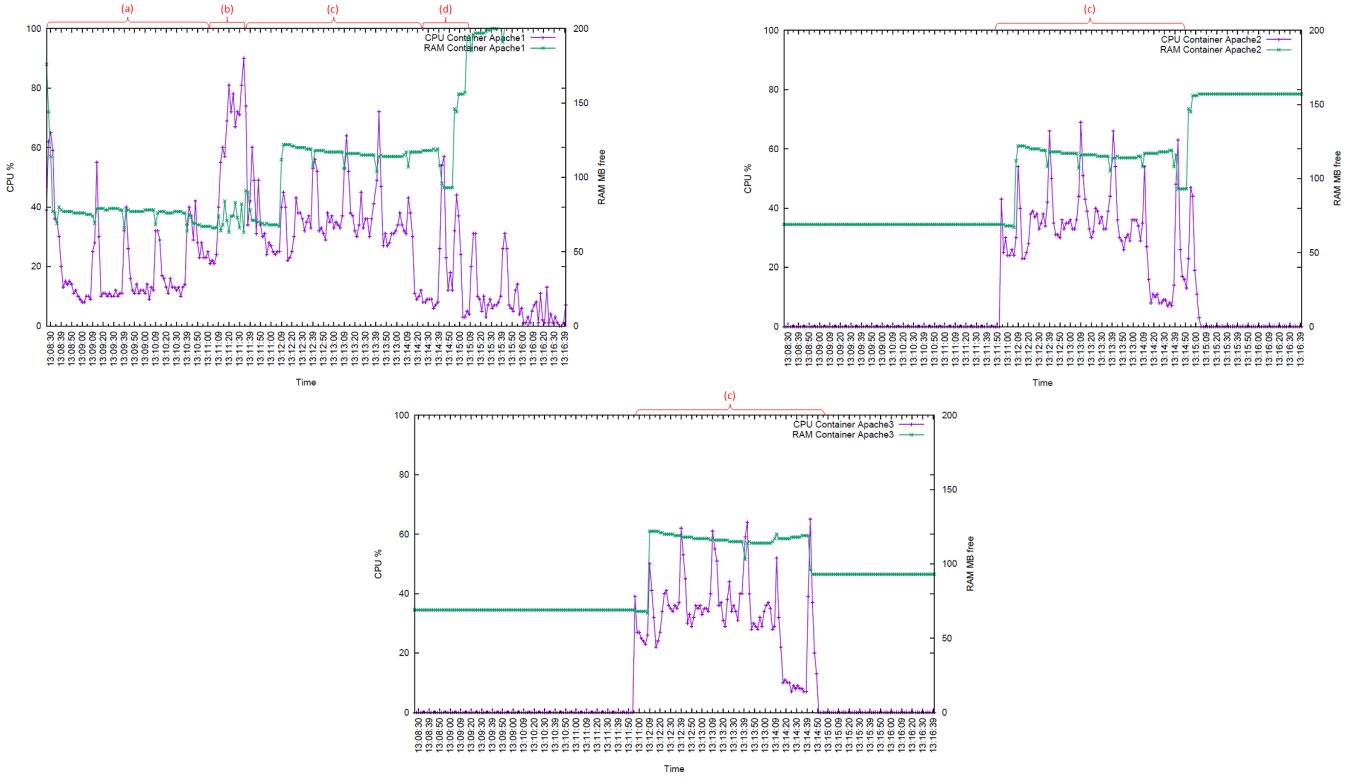

Figure 11: Benchmark 2 with PeakForecast Containers

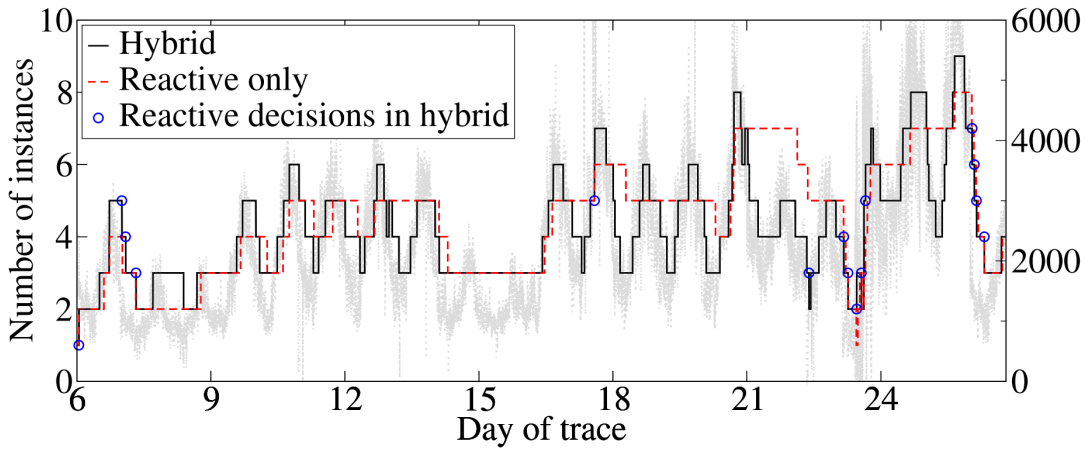

Figure 12: Auto-scaling simulation using 1998 FIFA World Cup trace with Platform Insights of Laura R. et al [55]

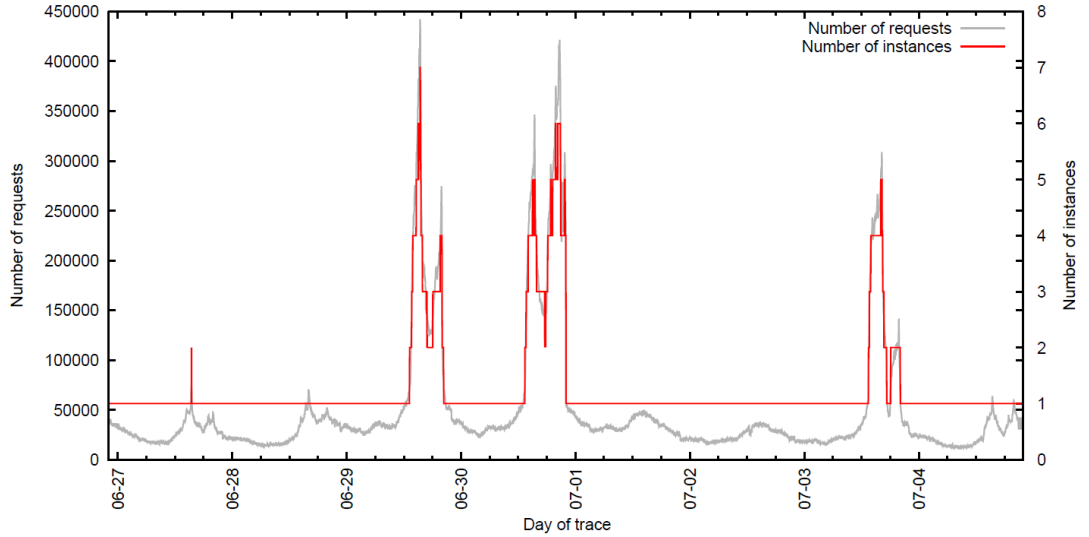

Figure 13: Auto-scaling simulation using 1998 FIFA World Cup trace with PeakForecast

To evaluate the performance of the elasticity, a simulation has been carried out also on a real dataset the FIFA 1998 World Cup Access logs [20], were used to simulate the incoming load to the web site. The log files were summarized to extract the number of requests arriving every 2 minutes. Figure 13 shows how resources are allocated during the period from June 26 to July 4 by PeakForecast. It clearly shows that the base resource required is an instance and it auto-scaling between 1 to 7 instance when the load increases or decreases. PeakForecast adjusts resource more appropriately, and hence will result in better utilization, than does the purely reactive controller and Platform Insights of Laura R. et al [55] that not incorporate an algorithm to detect change in workload mix.

6. CONCLUSIONS AND FUTURE WORK

In this paper, to response the following research questions: Is it possible to anticipate the effects of a traffic surge? If so, is it possible to protect the targeted system from unavailability? we have presented an elastic distributed resource scaling approach for IaaS cloud infrastructures that provide a medium-term resource demand prediction to absorb traffic surges before under-provisioning, named PeakForecast(PF). From a trace of queries received in the last seconds, minutes or hours, PF: *i)* Detecting Potential Traffic Surges, *ii)* After detecting the traffic surge, Forecasting the Upcoming Traffic by using a forecasting model, *iii)* Estimating the number resources required to absorb the remaining traffic to come, *iv)* Auto-scaling resources by quickly automatically adding resources to absorb traffic surges. We have evaluated our approach using MediaWiki and web application traffic traces that were collected on the Japanese version of Wikipedia some days before and after the tsunami on March 11th, 2011, in an interval where the traffic surge has been observed and a dataset acquired from the FIFA 1998 World Cup web site. The experimental results confirm that our prototype elastic middleware solution, based on PF, can provide instantaneous elasticity of resources during traffic surges.

As for future work, we planned: *i)* To give PF the possibility of quickly changing some VM parameters (e.g., memory,

CPU) to delay the peak of traffic before considering the deployment of additional VMs or Containers. *ii)* In real commercial scenarios, allocating extra VMs or Containers will have to be paid for a whole hour. Thus, we should allow PF, based on predictions, to consider the expected duration of the traffic surge in order to adjust the minimum number of VMs or Containers that would maintain an acceptable QoS and minimize wasting of resources.

Appendices

Listing 1: Client Composite of the PeakForecast Middleware.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
                    xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
                    name="peakforecast-ws-client"
                    targetNamespace="http://frascati.ow2.org/peakforecast-ws">
          <service name="r" promote="monitor/r">
                <interface.java interface="java.lang.Runnable"/>
          </service>
          <component name="monitor">
                <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
                    lib.Monitor"/>
                <service name="r">
                  <interface.java interface="java.lang.Runnable"/>
                </service>
                <reference name="controllerSoftwareService">
                  <binding.ws wsdli:wsdlLocation="http://192.168.1.108:9000/
                      ControllerSoftwareService?wsdl"
                                        wsdlElement="http://api.peakforecast.fabric.
                                            akka.frascati.ow2.org/#wsdl.port(
                                            ControllerSoftwareService/
                                            ControllerSoftwareServicePort)"
                                        />
                </reference>
                <reference name="controllerVMService">
                  <binding.ws wsdli:wsdlLocation="http://192.168.1.108:9000/
                      ControllerVMService?wsdl"
                                        wsdlElement="http://api.peakforecast.fabric.
                                            akka.frascati.ow2.org/#wsdl.port(
                                            ControllerVMService/ControllerVMServicePort
                                            )"
                                        />
                </reference>
                <reference name="controllerKubectlService">
                  <binding.ws wsdli:wsdlLocation="http://192.168.1.108:9000/
                      ControllerKubectlService?wsdl"
                                        wsdlElement="http://api.peakforecast.fabric.
                                            akka.frascati.ow2.org/#wsdl.port(
                                            ControllerKubectlService/
                                            ControllerKubectlServicePort)"
                                        />
                </reference>
                <reference name="eventNotificationAlertService">
                  <binding.ws wsdli:wsdlLocation="http://192.168.1.108:9000/
                      EventNotificationAlertService?wsdl"
                                        wsdlElement="http://api.peakforecast.fabric.
                                            akka.frascati.ow2.org/#wsdl.port(
                                            EventNotificationAlertService/
                                            EventNotificationAlertServicePort)"
                                        />
                </reference>
          </component>
          <component name="shell">
                <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
                    lib.ShellImpl"/>
          </component>
          <wire source="monitor/sh" target="shell/ShellService"/>
</composite>
```

Listing 2: Server Composite of the PeakForecast Middleware.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite  xmlns="http://www.osoa.org/xmlns/sca/1.0"
  xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
  targetNamespace="http://frascati.ow2.org/peakforecast-ws"
  name="peakforecast-ws-server">

          <!--  SCA binding (Web service)   -->
          <service name="controllersoftwareservice" promote="controllersoftware/
              controllersoftwareservice">
                <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.api
                    .ControllerSoftwareService" />
                <binding.ws uri="http://192.168.1.108:9000/ControllerSoftwareService"/>
          </service>
          <service name="controllervmservice" promote="controllervm/controllervmservice"
              >
                <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.api
                    .ControllerVMService" />
                <binding.ws uri="http://192.168.1.108:9000/ControllerVMService"/>
          </service>
          <service name="controllerkubectlservice" promote="controllerkubectl/
              controllerkubectlservice">
                <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.api
                    .ControllerKubectlService" />
                <binding.ws uri="http://192.168.1.108:9000/ControllerKubectlService"/>
          </service>
          <service name="eventnotificationalertservice" promote="eventnotificationalert/
              eventnotificationalertservice">
                <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.api
                    .EventNotificationAlertService" />
                <binding.ws uri="http://192.168.1.108:9000/EventNotificationAlertService
                    "/>
          </service>

          <!--  SCA Component   -->
          <component name="controllersoftware">
                <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
                    lib.ControllerSoftwareImpl"/>
                <service name="controllersoftwareservice">
                  <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.
                      api.ControllerSoftwareService" />
                </service>
          </component>
          <component name="controllervm">
                <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
                    lib.ControllerVMImpl"/>
                <service name="controllervmservice">
                  <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.
                      api.ControllerVMService" />
                </service>
          </component>
          <component name="controllerkubectl">
                <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
                    lib.ControllerKubectlImpl"/>
                <service name="controllerkubectlservice">
                  <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.
                      api.ControllerKubectlService" />
                </service>
          </component>
          <component name="eventnotificationalert">
                <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
                    lib.EventNotificationAlertImpl"/>
                <service name="eventnotificationalertservice">
                  <interface.java interface="org.ow2.frascati.akka.fabric.peakforecast.
                      api.EventNotificationAlertService" />
                </service>
          </component>

          <component name="shell">
                <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
                    lib.ShellImpl"/>
          </component>
```

```
<component name="mediawiki">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.MediaWiki"/>
</component>
<component name="apache">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.Apache"/>
</component>
<component name="php">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.PHP"/>
</component>
<component name="mysql">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.MySql"/>
</component>
<component name="nginx">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.Nginx"/>
</component>
<component name="proximysql">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.ProxyMysql"/>
</component>

<component name="twitter">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.TwitterImpl"/>
</component>
<component name="sms">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.SMSImpl"/>
</component>
<component name="email">
      <implementation.java class="org.ow2.frascati.akka.fabric.peakforecast.
          lib.EmailImpl"/>
</component>

<!--  SCA wire (local)   -->
<wire source="mediawiki/sh" target="shell/ShellService"/>
<wire source="apache/sh" target="shell/ShellService"/>
<wire source="php/sh" target="shell/ShellService"/>
<wire source="mysql/sh" target="shell/ShellService"/>
<wire source="nginx/sh" target="shell/ShellService"/>
<wire source="proximysql/sh" target="shell/ShellService"/>

<wire source="controllersoftware/cmsMediawiki" target="mediawiki/
    SoftwareService"/>
<wire source="controllersoftware/serverApache" target="apache/SoftwareService"
    />
<wire source="controllersoftware/libphp" target="php/SoftwareService"/>
<wire source="controllersoftware/serverMysql" target="mysql/SoftwareService"/>
<wire source="controllersoftware/loadbalancerNginx" target="nginx/
    SoftwareService"/>
<wire source="controllersoftware/loadbalancerProximysql" target="proximysql/
    SoftwareService"/>
<wire source="controllersoftware/sh" target="shell/ShellService"/>
<wire source="controllersoftware/event" target="eventnotificationalert/
    eventnotificationalertservice"/>

<wire source="controllervm/sh" target="shell/ShellService"/>
<wire source="controllervm/event" target="eventnotificationalert/
    eventnotificationalertservice"/>

<wire source="controllerkubectl/sh" target="shell/ShellService"/>
<wire source="controllerkubectl/event" target="eventnotificationalert/
    eventnotificationalertservice"/>

<wire source="eventnotificationalert/twitter" target="twitter/TwitterService"
    />
<wire source="eventnotificationalert/sms" target="sms/SMSService"/>
<wire source="eventnotificationalert/email" target="email/EmailService"/>
```

```
</composite>
```

7. REFERENCES

[1] R.G. Brown, R.F. Meyer: The fundamental theory of exponential smoothing, Operations Research, 9, 1961, p. 673–685

[2] M. Beisiegel, et al. Service Component Architecture: Building Systems using a Service Oriented Architecture. white paper 0.9, BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase, Nov 2005.

[3] A. Gandhi, T. Zhu, M. Harchol–Balter, M.A. Kozuch: SOFTScale: Stealing Opportunistically for Transient Scaling, Middleware 2012, p. 142–163.

[4] J.C.B. Leite, D.M. Kusic, and D. Mosse: Stochastic approximation control of power and tardiness in a three–tier web–hosting cluster, In ICAC 2010, Washington, DC, USA : 41–50.

[5] R. Nathuji, A. Kansal, and A. Ghaffarkhah.: Q–clouds: Managing performance interference effects for qos-aware clouds, In EuroSys 2010, Paris, France, p. 237–250.

[6] P. Padala, K.-Y. Hou, K.G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant: Automated control of multiple virtualized resources, In EuroSys 2009, Nuremberg, Germany, p. 13–26.

[7] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz. Napsac: Design and implementation of a power–proportional web cluster, In Green Networking 2010, New Delhi, India, p. 15–22.

[8] T. Horvath and K. Skadron: Multi-mode energy management for multi-tier server clusters, In PACT 2008, Toronto, ON, Canada, p. 270–279.

[9] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah: Minimizing data center sla violations and power consumption via hybrid resource provisioning, In IGCC 2011, Orlando, FL, USA, p. 1–8.

[10] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper: Adaptive quality of service management for enterprise services, ACMTrans. Web, 2(1):1–46, 2008.

[11] B. Urgaonkar and A. Chandra: Dynamic provisioning of multi–tier internet applications, In ICAC 2005, Washington, DC, USA, p. 217–228.

[12] A. Chandra and P. Shenoy: Effectiveness of dynamic resource allocation for handling internet flash crowds, Technical Report TR03–37, Department of Computer Science, Umas, November 2003.

[13] B. Urgaonkar and P. Shenoy: Cataclysm: Scalable overload policing for internet applications, Journal of Network and Computer Applications, 31(4):891–920, 2008.

[14] A. Adya, W.J. Bolosky, R. Chaiken, J.R. Douceur, J. Howell, and J. Lorch: Load management in a large-scale decentralized file system, MSR-TR, 2004–60, July 2004.

[15] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra: Kernel mechanisms for service differentiation in overloaded web servers. In USENIX ATC 2001, Boston, MA, USA, p. 189–202.

[16] L. Cherkasova and P. Phaal: Session-based admission control: A mechanism for peak load management of commercial web sites, IEEE Trans. Comput., 51, June 2002, p. 669–685.

[17] H. C. Lim, S. Babu, and J. S. Chase: Automated control for elastic storage, in ICAC10. New York,NY, USA: ACM, 2010, p. 19–24.

[18] B. Trushkowsky, P. Bodik, A. Fox, M.J. Franklin, M.I. Jordan, and D.A. Patterson: The scads director: scaling a distributed storage system under stringent performance requirements, In FAST 2011, San Jose, CA, USA, p. 163–176.

[19] M. Micheal, B. Ivan, E. Vincent C and B. Ivona: Revealing the MAPE loop for the autonomic management of Cloud infrastructures, In IEEE Symposium on Computers and Communications (ISCC), pages 147–152.IEEE, 2011.

[20] The Internet Traffic Archive http://ita.ee.lbl.gov/html/traces.html Accessed 12th March 2013.

[21] Amazon web services, 2017, last visited: 2017/04/08. [Online]. Available: https://aws.amazon.com

[22] RightScale 2017, last visited: 2017/04/08. [Online]. Available: http://www.rightscale.com

[23] N. Roy, A. Dubey, A. Gokhale, Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting, Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing, pp. 500–507, 2011.

[24] Z. Gong, J. Wilkes, and X. Gu, Press: Predictive elastic resource scaling for cloud systems, in 2010 International Conference on Network and Service Management. IEEE, 10 2010, pp. 9–16.

[25] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity management and demand prediction for next generation data centers. In International Conference on Web Services, 2007.

[26] G. Chen and et al., Energy-aware server provisioning and load dispatching for connection-intensive internet services, in Proc. NSDI, 2008.

[27] P. Saripalli, G. Kiran, R. R. Shankar, H. Narware, and N. Bindal, Load prediction and hot spot detection models for autonomic cloud computing, 2011 Fourth IEEE International Conference on Utility and Cloud Computing, pp. 397–402, 12 2011.

[28] Joao Loff et Joao Garcia. Vadara : Predictive Elasticity for Cloud Applications. In Proceedings of the 2014 IEEE 6th International Conference on Cloud Computing Technology and Science, CLOUDCOM14, pages 541–546 Washington, DC, USA, 2014. IEEE Computer Society.

[29] X. Zhu and et al., 1000 Islands: Integrated capacity and workload management for the next generation data center, in Proc. ICAC, Jun. 2008

[30] E. Kalyvianaki, T. Charalambous, and S. Hand, Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters, in Proc. ICAC, 2009.

[31] P. Padala and et al., Adaptive control of virtualized resources in utility computing environments, in Proc. Eurosys, 2007.

[32] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, VCONF: A reinforcement learning approach to virtual machines auto-configuration, in Proc. ICAC, 2009.

[33] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In Proc. of the USENIX International Conference on Automated Computing (ICAC13). San Jose, CA, 2013.

[34] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In Automated Software Engineering, 2011.

[35] N. Huber, F. Brosig, and S. Kounev. Model-based self-adaptive resource allocation in virtualized environments. In Software Engineering for Adaptive and SelfManaging Systems, 2011.

[36] Microsoft Azure 2017, last visited: 2017/08/25. [Online]. Available: https://azure.microsoft.com.
[37] RightScale, Set up Autoscaling using Voting Tags, last visited: 2017/08/25 [Online]. Available: http://support.rightscale.com/12-Guides/Dashboard_Users_Guide/Manage/Arrays/Actions/Set_up_Autoscaling_using_Voting_Tags/index.html
.
[38] Tania Lorido-Botrean, Josee Miguel-Alonso, Josee A. Lozano, Auto-scaling Techniques for Elastic Applications in Cloud Environments, Technical Report September 5, 2012.
[39] Jonathan Kupferman, Jeff Silverman, Patricio Jara, and Jeff Browne. Scaling into the cloud. Technical report, University of California, Santa Barbara; CS270 - Advanced Operating Systems, 2009.
[40] SPICER, James. CloudPlan: An Elastic Management System for Cloud Planning. Discovery, Invention & Application, [S.l.], june 2016. Available at: https://computing.derby.ac.uk/ojs/index.php/da/article/view/101. Date accessed: 25 aug. 2017.
[41] Thraka, Andy . How to scale a cloud service, Azure 2015. url: http://azure.microsoft.com/en-gb/documentation/articles/cloud-services-how-to- scale/.
[42] Hyndman, Rob J, Koehler, Anne B ; Ord, J Keith ; Snyder, Ralph D : Forecasting with exponential smoothing : the state space approach, Springer, 2008. 368 p.
[43] A. Jain, J. Jasiulek, Fast Fourier transform algorithms for linear estimation, smoothing and Riccati equations, IEEE Transactions on Signal Processing 1991, P. 137–147.
[44] R. Han, L. Guo, M. M. Ghanem, Y. Guo, Lightweight Resource Scaling for Cloud Applications, Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 644–651, 2012
[45] William LeFebvre. CNN.com: Facing A World Crisis. Invited Talk, USENIX ATC 2002.
[46] Jim Hu and Greg Sandoval. Web acts as hub for info on attacks. CNET news, September 2001.
[47] Lisa A. Wald and Stan Schwarz. The 1999 southern california seismic network bulletin. Seismological Research Letters, 71:401–422, July 2000
[48] Stephen Adler. The Slashdot Effect: An Analysis of Three Internet Publications. http://ssadler.phy.bnl.gov/adler/SDE/SlashDotEffect.html.
[49] Josh Constine. Walmarts black friday disaster: Website crippled, violence in stores. http://techcrunch.com/2011/11/25/walmart–black–friday, November 2011
[50] Kathleen Ohlson. Victorias secret knows ads, not the web. Computer World, February 1999
[51] Martin Arlitt and Tai Jin. Workload characterization of the 1998 world cup web site. IEEE Network, 1999.
[52] Peter Pachal. Amazon apologizes for cloud outage, issues credit to customers. PCMag, April 2011.
[53] Bianca Schroeder, Eduardo Pinheiro, and Wolf–Dietrich Weber. DRAM errors in the wild: a large–scale field study. In SIGMETRICS 2009, Seattle, WA, USA.
[54] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal and T. Wood, Agile dynamic provisioning of multi-tier internet applications, ACM TAAS, 2008.
[55] Kathryn Bean Laura R. Moore and Tariq Ellahi. A coordinated reactive and predictive approach to cloud elasticity. CLOUD COMPUTING 2013 : The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization, 2013
[56] A. Ali-Eldin, J. Tordsson and E. Elmroth, An adaptive hybrid elasticity controller for cloud infrastructures in Network Operations and Management Symposium (NOMS), IEEE, 2012, pp. 204–212)
[57] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. Smartscale : Automatic application scaling in enterprise clouds. In Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pages 221–228. IEEE, 2012
[58] E. Caron, L. Rodero-Merino, F. Desprez and A. Muresan, Autoscaling, load balancing and monitoring in commercial and opensource clouds, 2012.
[59] T. Lorido–Botran, J. Miguel–Alonso and J. A. Lozano, Auto–scaling Techniques for Elastic Applications in Cloud Environments, University of Basque Country, Tech. Rep. EHUKAT–IK–09–12, 2012
[60] A. Gambi, G. Toffetti and M. Pezze, Assurance of self–adaptive controllers for the cloud, in Assurances for Self-Adaptive Systems, pp 311-339, 2013.
[61] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Autonomic resource provisioning for cloud–based software. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self–Managing Systems, pages 95–104. ACM, Pages 95-104 , 2014.
[62] E. Caron, L. Rodero–Merino, F. Desprez and A. Muresan, Autoscaling load balancing and monitoring in commercial and opensource clouds, RR-7857, INRIA 2012, pp.27.
[63] L. M. Vaquero, L. Rodero-Merino and R. Buyya, Dynamically scaling applications in the cloud, ACM SIGCOMM Computer Communication Review,Pages 45–52, 2011.