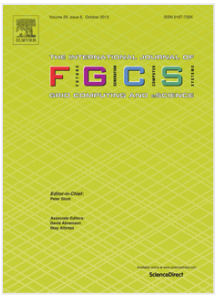
Accepted Manuscript



Simulation of SLA-based VM-scaling algorithms for cloud-distributed applications

Alexandru-Florian Antonescu, Torsten Braun

PII: S0167-739X(15)00032-1
DOI: http://dx.doi.org/10.1016/j.future.2015.01.015
Reference: FUTURE 2705

To appear in: *Future Generation Computer Systems*

Received date: 3 October 2014
Revised date: 22 January 2015
Accepted date: 24 January 2015

Please cite this article as: A.-F. Antonescu, T. Braun, Simulation of SLA-based VM-scaling algorithms for cloud-distributed applications, *Future Generation Computer Systems* (2015), http://dx.doi.org/10.1016/j.future.2015.01.015

Simulation of SLA-Based VM-Scaling Algorithms for Cloud-Distributed Applications

Alexandru-Florian Antonescu[a,b,*], Torsten Braun[b]

[a] *SAP Switzerland, Products & Innovation, Research, Regensdorf, Switzerland*
[b] *University of Bern, Communication and Distributed Systems, Bern, Switzerland*

Abstract

Cloud Computing has evolved to become an enabler for delivering access to large scale distributed applications running on managed network-connected computing systems. This makes possible hosting Distributed Enterprise Information Systems (dEISs) in cloud environments, while enforcing strict performance and quality of service requirements, defined using Service Level Agreements (SLAs). SLAs define the performance boundaries of distributed applications, and are enforced by a cloud management system (CMS) dynamically allocating the available computing resources to the cloud services. We present two novel VM-scaling algorithms focused on dEIS systems, which optimally detect most appropriate scaling conditions using performance-models of distributed applications derived from constant-workload benchmarks, together with SLA-specified performance constraints. We simulate the VM-scaling algorithms in a cloud simulator and compare against trace-based performance models of dEISs. We compare a total of three SLA-based VM-scaling algorithms (one using prediction mechanisms) based on a real-world application scenario involving a large variable number of users. Our results show that it is beneficial to use autoregressive predictive SLA-driven scaling algorithms in cloud management systems for guaranteeing performance invariants of distributed cloud applications, as opposed to using only reactive SLA-based VM-scaling algorithms.

Keywords: Cloud Computing, Service Level Agreements, Horizontal Scaling, Prediction, Simulation

*Corresponding author
Email addresses: alexandru-florian.antonescu@sap.com (Alexandru-Florian Antonescu), braun@iam.unibe.ch (Torsten Braun)

1. Introduction

Cloud Computing [1] has evolved to become an enabler for delivering access to large-scale distributed applications[2] running inside managed environments composed of network-connected computing systems. This made possible hosting of Distributed Enterprise Information Systems (dEISs) in cloud environments, while enforcing strict performance and quality of service requirements, defined using Service Level Agreements (SLA).

SLAs are contracts defining the performance and quality of service (QoS) boundaries of distributed applications. A cloud management system (CMS) enforces SLAs by dynamically allocating available computing resources to cloud services. A CMS monitors both the software cloud resources as well as the underlying physical network and computing resources. It uses this information for deciding the actions to be taken, such as increasing the number of VMs (scaling-out), decreasing (scaling-in), or migrating software components in order to maintain the conditions defined in the SLAs and for maximising provider-specific metrics (e.g. energy efficiency).

It is often the case that cloud applications exhibit predictable and repeatable patterns in their resource utilisation levels, caused by the execution of repeatable workloads (e.g. with hourly, daily, weekly patterns). A CMS can benefit from detecting such repeatable patterns by combining this information with prediction mechanisms in order to estimate the near-term utilisation level of both software and physical resources, and then to optimise the allocation of resources based on the SLAs.

Also, the specific way of packing cloud applications in Virtual Machines (VMs) allows a CMS to scale cloud-distributed applications by means of "horizontal"-scaling, where the number of VMs allocated to application-services is increased of decreased according to variations in the external workload. Therefore, using SLAs for specifying the performance of cloud applications could enable the CMS to better perform VM-scaling by correlating the SLA guarantees with the actual number of VMs allocated to cloud applications, their QoS metrics and the size of the distributed workload.

We define the research question as: *"How can a CMS dynamically scale the number of VMs allocated to cloud services, so that the SLA-defined performance constraints are maintained under variable workload conditions such as fluctuating number of users?"*.

We present an approach for designing and testing SLA scaling algorithms for dEIS systems by using performance-models of cloud-distributed applications (built with the help of constant-workload benchmarks) and then simulating the scaling algorithms in a cloud simulator against the performance models. We extend the work in [3] and [4] by presenting and evaluating two new SLA-based VM-scaling algorithms. In total, we compare three SLA-based VM-scaling algorithms (one using prediction mechanisms) based on (1) a real-world application scenario involving a large variable number of users, and (2) pre-recorded monitoring traces of an actual distributed enterprise application.

Our results show that it is valuable to use a predictive SLA-driven VM-scaling algorithm in a cloud management system for guaranteeing performance SLA invariants of distributed cloud applications.

Our main contributions can be summarised as follows. We present an approach for analysing the performance boundaries of a distributed application using batches of benchmarks. We then show how Little's Law can be combined with the benchmark's results and SLA-defined performance conditions in order to identify optimal scaling conditions for the distributed application. We also show how multi-step linear regression can be used to efficiently predict application workloads, and then we integrate the prediction mechanism into a SLA-based VM-scaling algorithm. In total, we analyse three SLA-based VM-scaling algorithms.

The rest of our paper is organised as follows. Section 2 presents the related work in the field of distributed enterprise applications, cloud computing simulators, prediction models, and SLA-based scaling of cloud services. Section 3 introduces the problem of predicting time series. Section 4 introduces an algorithm for doing multi-step prediction using linear regression models. Section 5 presents a benchmarking methodology based on Little's Law for exploring the relations between system's workload, occupancy (concurrency) and the average execution time. We then use these relations for finding the maximum processing capacity of the corresponding VM-instances based on SLA-defined performance conditions. Section 6 introduces two SLA-based VM-scaling algorithms that use the mechanisms presented in Sections 4 and 5. Section 7 discusses the results of evaluating the SLA-based VM-scaling algorithms using a simulation of a real-world multi-user workload in a cloud simulator. Finally, Section 8 draws conclusions.

2. Related Work

We split the related work section into four subsections, as follows: (1) distributed enterprise information systems, (2) cloud computing simulator, (3) time series prediction mechanisms, and (4) SLA-based scaling of cloud services.

2.1. Distributed Enterprise Information Systems

Distributed (Cloud) Enterprise Applications [2] are component-based, distributed, scalable, and complex business applications, usually mission-critical. Commonly, they are multi-user applications handling large datasets in a parallel and/or distributed manner, and their purpose is to solve specific business problems in a robust way. Often these applications are running in managed computing environments, such as datacenters [5]. Enterprise Resource Planning (ERP)[6] applications are a type of distributed enterprise applications, which provide an integrated view of core business processes, using multiple database systems.

According to Marston et al. [7] SLAs play an important role in the enterprise environment especially for mitigating risks associated with variability in availability of cloud resources. They often contain a model of guarantees and penalties, which can be used by infrastructure management systems for allocating and optimising the use of datacenter resources.

Antonescu et al. [3] describe a model of concurrent workload processing in distributed enterprise information systems, by analysing the effect of concurrent processing of distributed transactions on the physical resources. We extend this model by using linear regression and log-normal distributions to simulate higher levels of concurrency.

2.2. CloudSim Cloud Simulator

CloudSim [8] positions itself as a generic cloud simulator for both applications and cloud infrastructures, as it allows modelling of hardware and software cloud resources. It allows representing physical host entities, network links and datacenters. The modelled software entities are virtual machines (VMs), brokers (services) and cloudlets (tasks). The mentioned entities are manipulated using a Java API. The simulator is implemented using discrete event communication. CloudSim provides a wide selection of resource allocation policies for VM-to-host and task-to-VM mapping. It is worth noting that

network links are only modelled through their bandwidth and fixed transmission delay values. NetworkCloudSim improves these network-modelling aspects by introducing additional simulation entities, such as routers and network packets. However, for the purpose of this work, CloudSim's network-modelling capabilities were enough.

Buyya et al. [9] presented an approach for simulating large scale cloud environments using CloudSim, describing the steps required for simulating a large number of hosts and network connections. However, they did not focus on how to model applications using CloudSim. Our work extends [4] and it enables simulating large-scale cloud-distributed applications with dynamic VM-scaling capabilities.

Garg et al. [10] describe an extension of CloudSim, which allows simulating complex tasks composed of multiple computational and communication-intensive subtasks. They also allow better modelling of network topologies, by introducing the concepts of switches and data flows. While their work is more focussed on message-passing interface (MPI) applications, our work emphasises the concurrent execution of CPU-intensive tasks in enterprise systems.

Antonescu et al. [4] present SLA-driven simulation of multi-tenant scalable cloud-distributed enterprise applications. The authors describe a SLA-based VM-scaling algorithm for distributed systems, and its implementation in CloudSim. We extend the presented reactive VM-scaling algorithm with two additional VM-scaling algorithms, which we implement and validate using CloudSim.

2.3. Time Series Prediction

Visan et al. [11] describe a bio-inspired prediction algorithm based on a Cascade-Correlation neural network, which uses a genetic algorithm for initialising the network's weights. The authors use their algorithm for performing both one-step and multi-step predictions of a large-scale distributed experiment, with good results.

Islam et al. [12] present an approach for predicting the aggregated percentage of CPU utilisation of VMs running a distributed web benchmark, using both error correction neural networks (ECNN) and linear regression (LR). Their results suggest that although using ECNN yields better prediction results than, the need to retrain the neural network might be a disadvantage compared to the use of LR. We focus our work on using LR in the context of SLA-driven scaling of cloud services, showing how prediction can

be used for mitigating the disadvantages caused by the delay in instantiating VMs.

Roy et al. [13] present a VM-allocation algorithm, which uses a second order autoregressive moving average prediction method for optimising the utility of the application over a prediction horizon. We also use the prediction of the arrival rate of users for sizing the distributed system. We directly use SLAs for solving the VM-scaling problem.

Antonescu et al. [14] present an algorithm for allocating VMs to hosts using a genetic algorithm, which simultaneously tries to optimise multiple utility functions and uses prediction mechanisms (triple exponential smoothing) for forecasting the resource utilisation. We take a similar approach in this paper. We focus on the scaling requirements of the cloud services, instead of the VMs.

2.4. SLA-Based Scaling of Cloud Services

Garca et al. [15] describe Cloudcompaas, a SLA-aware Platform-as-aService (PaaS) Cloud platform for managing resource lifecycle, using WS-Agreement as a basis for the SLA specification, as well as representation of cloud resources. They present multiple experiments in which the virtual infrastructure is scaled based on the number of incoming requests and defined SLAs.

Garg et al. [16] propose an admission control mechanism that maximises the resource utilisation and profit, while it also ensures that the SLA-specified QoS requirements are met. The authors used a forecasting model based on a multi-layer feed-forward neural network for predicting the utilisation of server resources. We focus our work on predicting the incoming rate of requests to the cloud services, and we use this as input to the algorithms responsible for scaling the virtual infrastructure.

Antonescu et al. [17] describe a resource allocation and VM-scaling algorithm based on SLA constraints, combined with dynamically discovered correlations between the service monitoring metrics and prediction of optimal instantiation time for VMs based on detected patterns in resource utilisation levels. In the current paper we describe a benchmark-driven approach for discovering the maximum processing capacity of the cloud-VMs using Little's Law. We combine this mechanism with autoregression-based prediction and novel control algorithms for optimal VM-scaling.

3. Time Series Prediction

We define the SLA Cloud Management Optimization (SLA-CMO) problem [14] [18] [19] [20] as: improving the efficiency of allocating datacenter's computing resources by dynamically changing the number of VMs allocated to cloud services, so that SLA-defined performance requirements are met under variable workload conditions.

Solving the SLA-CMO problem depends directly on having a reliable source of monitoring information reflecting the state (e.g. number of allocated VMs, system's throughput, requests arrival rate, response time) of the managed distributed systems. The management system will then take corrective actions for ensuring that the SLA contracts guaranteeing the system's performance are not violated. However, the actions' effects on the underlying system might be delayed, creating a time window during which the system might be in an undesirable state (e.g. under-performing). Such SLA violations can be avoided [17] if the SLA management system can timely predict the trend and variation of the critical monitoring system's parameters, allowing it to actuate (e.g. scale-out) at such a time moment that the newly added virtual resources (e.g. VMs) become active just as the workload would surpass the previous capacity of the cloud system.

The prediction problem can be defined as finding the next n values of a dependent system's parameter P, using p previous values of one or more predictors as shown in Equation 1

$$(P_{t+1}, P_{t+2}, ..., P_{t+n}) = f(X_t, X_{t-1}, ..., X_{t-p}) \tag{1}$$

where t is the current time moment, X_t is the value of the predicting vector at time t, and f is a function.

The above definition is true for parameters whose values are depending on other system parameters, for example, the average execution time of a distributed application depends on the number of VM instances allocated to the service, the workload's arrival rate and the application's average occupancy (as defined by Little's law [21]).

Another class of parameters are the independent ones, whose values are not determined by other parameters. An example of such system parameter is the workload arrival rate, which is determined only by factors external to the system. Such independent parameters can also be predicted, by observing certain patterns in the distribution of data and then recognising when

the data flow will start following a certain learned pattern. This can be formulated as a dependency of the current and future values of the parameter on the previous own values, as shown in Equation 2, where the mathematical terms are the same as the ones from Equation 1.

$$(P_{t+1}, P_{t+2}, ..., P_{t+n}) = f(P_t, P_{t-1}, ..., P_{t-p}) \tag{2}$$

Figure 1 shows a snapshot of such an independent parameter - the request arrival rate of a transaction processing application.

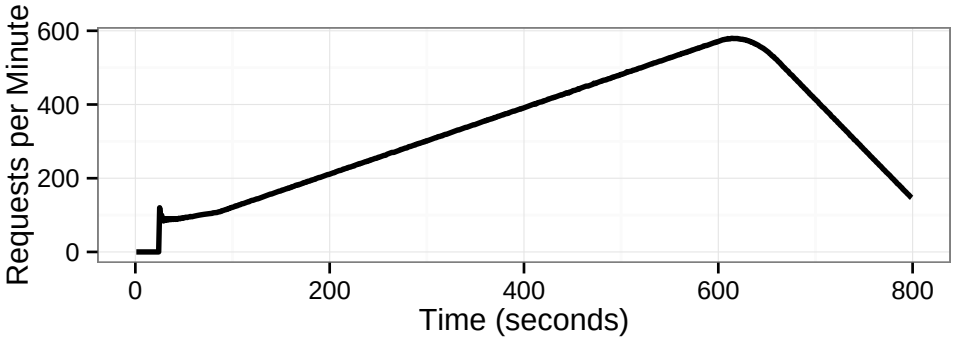


Figure 1: Requests Arrival Rate

When identifying repeatable patterns in temporal data series, it is important to pay attention to the statistical parameters of the data series, such as variance, mean, trend, etc. For example, a large variance in the data could hide a repeating pattern. Filtering the data series by applying a moving average or kernel smoothing [22] could expose the underlying trend of the data. By analysing the trend of the temporal data series, two types of variations could be observed: linear and non-linear. We shortly discuss the mathematical fundaments for linear regression in Appendix A, error calculation in Appendix B and non-linear regression in Appendix C.

4. Multi-Step Prediction using Linear Autoregression

We investigate autoregression for predicting multiple future values of an independent variable. As underlying example we will use the time series shown in Figure 1 containing a window of data representing the arrival rate of requests of an ERP system. As we want to use the data for predicting the

future values, this means that it will be processed in a streaming fashion, as it becomes available.

ALGORITHM 1: Streaming Linear Prediction using Autoregression

```
1  model ← NULL;
2  previous.data ← empty;
3  previous.time ← empty;
4  win.len ← p seconds;
5  prediction.length ← n seconds;
6  while Data Stream is Open do
7      win.data ← buffer(win.len);
8      win.time ← time(win.data) ;
9      if model is not NULL then
10         prediction.current ← predict(model, win.time);
11         err.mape ← MAPE(win.data, prediction.current);
12         if err.mape < ϵ then
13             t ← current time;
14             prediction.time ← (t + 1, ..., t + prediction.length);
15             win.predicted ← predict(model, prediction.time);
16             use win.predicted;
17         else
18             model ← NULL;
19             previous.data ← empty;
20             previous.time ← empty;
21         end
22     end
23     append win.data to previous.data;
24     append win.time to previous.time;
25     model ← linear regression(previous.data, previous.time);
26     model.accuracy ← accuracy(model);
27 end
```

Two important properties of the prediction algorithm are the following: (1) immunity to small variations (non-uniformity) in the data sampling period, and (2) auto-alignment of the regression with the actual signal's trend. The first property ensures that the prediction produced by the algorithm will always be sampled with the same period, allowing a deterministic use of the predicted data, independent of the actual sampling period of the predicted

signal. The second property ensures that the algorithm will auto-adjust to changes in the signal's trend (e.g. slope) by monitoring the prediction error and adapting the length of the window of data used for calculating the "learning" model.

The prediction algorithm works as follows. In the first iteration, a regression model is calculated (e.g. using R's *lm* function of the *stats* package [23]) using the current window of data in line 25. In the second iteration the previously calculated linear model will be used for predicting the current data window by applying the same regression *model* to the time moments corresponding to he sampled data (*win.time*), producing the *prediction.current* data vector in line 10. Next, the mean average percentage error (*err.mape*) is calculated from the current data window (*win.data*) and the model's prediction (*prediction.current*). This accuracy measure is effectively the out-of-sample error measure of the regression model, as the error was calculated with data coming from outside the time window known by the regression model. If the error is lower than a predefined threshold ϵ (e.g. 1.5%) then (1) the model is considered valid and it is used in line 15 for predicting the next n seconds of signal's values at times $t+1, ..., t+n$, and (2) the current data window and its sampling time moments is added to the previous data window, respectively to the previous sampling time window. If the error is larger than the specified threshold, than the *model* is dropped, together with the accumulated data.

By calculating the signal's prediction at times $t+1, ..., t+n$, the immunity to variations in the signal's sampling period is guaranteed, which allows the subsequent components to use the produced data forecast at any arbitrary time horizon smaller than n. Also, by accumulating the signal's data samples and calculating the regression model with an increasing length of the data window this ensures that the regression model's slope will be aligned with the signal's slope with an error of at most ϵ. As soon as the out-of-sample $MAPE$ error of the regression's model exceeds ϵ, the model and accumulated data will be dropped, and a new *model* will be calculated, thus satisfying the second property of the prediction algorithm.

The linear regression model's accuracy represented by the RMSE and MAPE errors are displayed in Figure 2, when applying linear regression to the data from Figure 1 (average arrival rates), with a regression window *win.len* of 10 seconds. As expected the graphs show two peaks because of the two changes in the underlying data's trend at time 20 and 600.

Figure 3 shows the evolution of both RMSE and MAPE errors for the

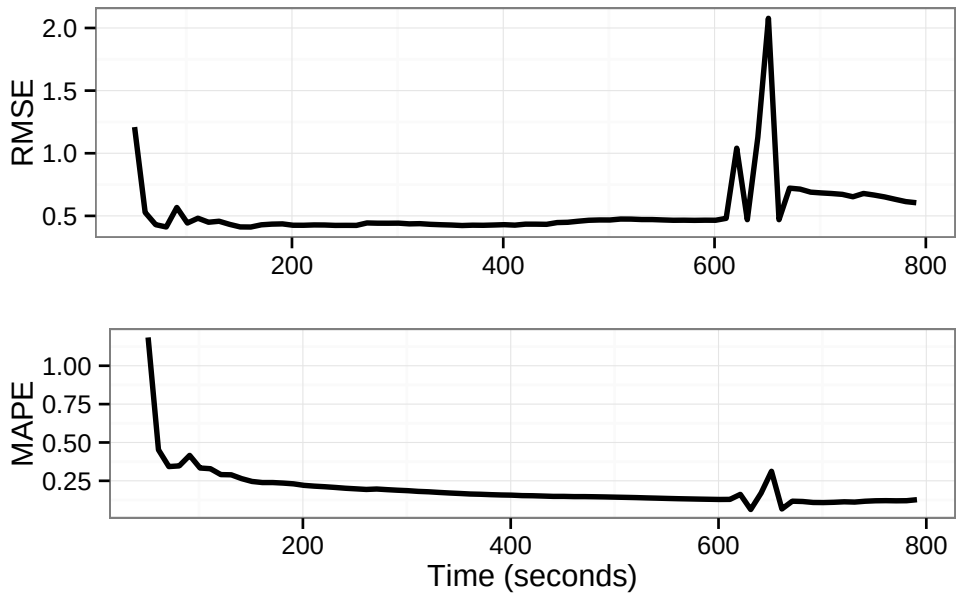


Figure 2: Regression Errors. Top: Root Mean Standard Error. Bottom: Mean Absolute Percentage Error

out-of-sample accuracy. As expected, there are two regions where the errors have large values, in the beginning, around time moment 0, and around time moment 600. This is because in the beginning of the time series the data experienced a slight instability when transitioning from 0 to 60 requests per minute. Also, at time moment 600 the trend of the data changed from increasing to decreasing, leading to the an decrease of the accuracy of the previous regression model valid only until the time moment 600.

For calculating the signal's prediction of the next n seconds (equal to 40 in our example), first a data window *prediction.time* is created (line 14) containing the time values between the current time $t + 1$ and $t + n$. The regression model is then used in line 15 for calculating the data prediction *win.prediction* corresponding to time interval *prediction.time*.

The predicted data *win.prediction* is finally used in line 16. An example of using the predicted data is given in Section 6, where the predicted arrival rate of requests is used as input to a SLA-based VM-Scaling algorithm.

The predicted data obtained by running the algorithm is displayed along the initial data in Figure 4. The very good accuracy of the prediction algo-

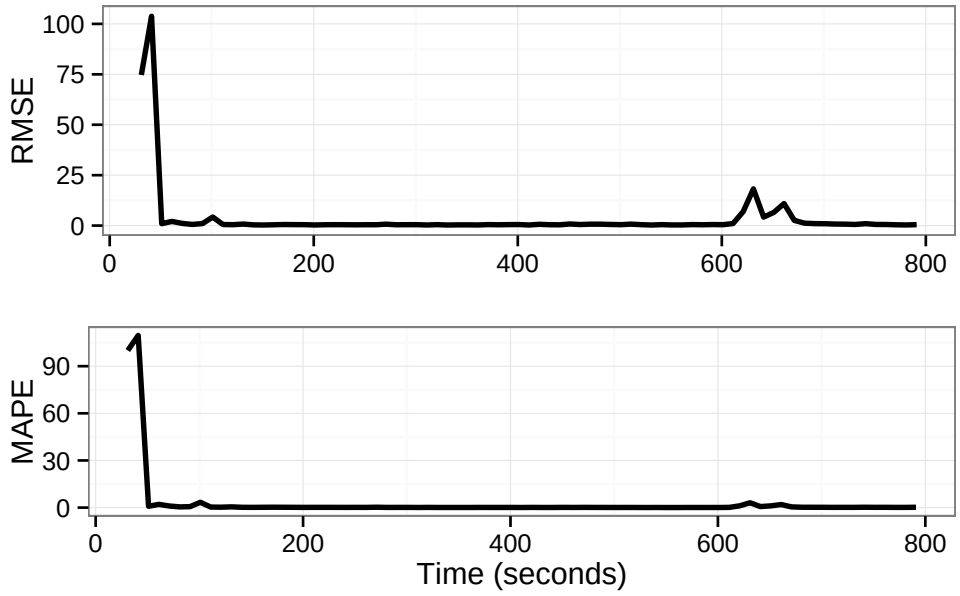


Figure 3: Forecast Errors. Top: Root Mean Standard Error. Bottom: Mean Absolute Percentage Error

rithm can be observed, as the predicted data closely follows the input data. Also, it can be noticed that the predicted data starts at around time moment 70, after the data's trend stabilises itself.

5. Performance Profiling of Cloud-Distributed Applications

In this section we present a performance profiling analysis of a cloud-distributed Enterprise Information System (dEIS) ([3] [4] [24] [17] [14] [19]) based on Little's law [21]. The purpose of this analysis is to determine the dependencies between the average arrival rate of requests to a distributed system, the system's average throughput, the average number of concurrent requests executed by the system and the average execution time.

Once these relations are known, we will use them to create an improved SLA-based service scaling policy, extending the algorithm presented in [4]. We describe the enhanced scaling algorithms in Section 6, and we show the simulation of these SLA policies in Section 7.

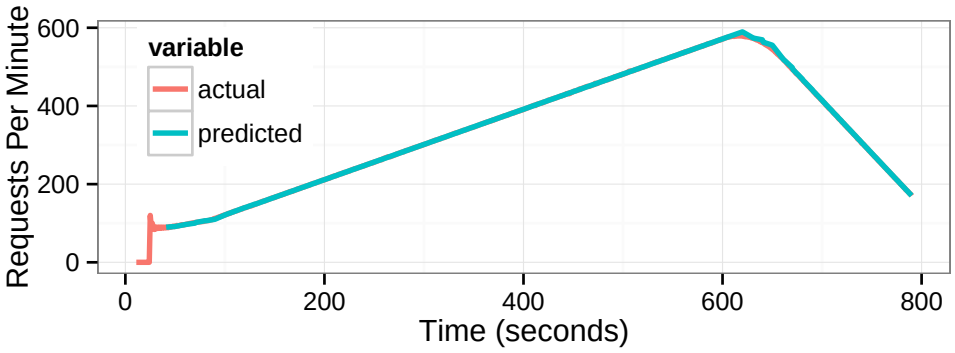


Figure 4: Forecast vs. Actual Data

5.1. Little's Law in the Context of Distributed Computing Systems

Little's Law [21] applies to users-processing systems, and it is a result from the queueing theory stating that the long term average number of users (L) in a system is equal to the product of the rate (λ) at which users arrive, and the average waiting time (W) that a user spends in the system, as expressed algebraically in Equation 3.

$$L = \lambda W \tag{3}$$

Another form of Little's Law applies to the relation between the average system's throughput (Th), mean number of users in the system (L), and the average execution time (W), as expressed by Equation 4.

$$W = \frac{L}{Th} \tag{4}$$

It is important to note that Equation 3 uses the arrival rate, while Equation 4 uses the system's throughput. The two equations are equivalent under conservation of flow conditions, when the average arrival rate (λ) is equal to the average departure rate (or throughput Th). Also, all the jobs entering the system must exit the system at a given point, so the system must report also the exceptional cases when a job fails, as long as that job was considered in the calculation of the arrival rate. Finally, the system needs to be stable [21], by occasionally having $L = 0$ (empty system).

We will use Equation 4 in Section 5.3 when we will be presenting a methodology for benchmarking a distributed system in order to find out the

dependencies between the average arrival rate, the system's average throughput, the average number of concurrent requests executed by the system and the average execution time.

5.2. Distributed Enterprise Information System under Test

We apply the management scaling problem to a Distributed Enterprise Information System (dEIS) composed of multiple scalable tiers, as described in this section.

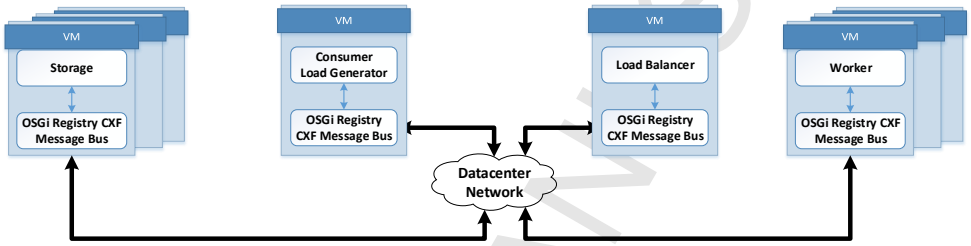
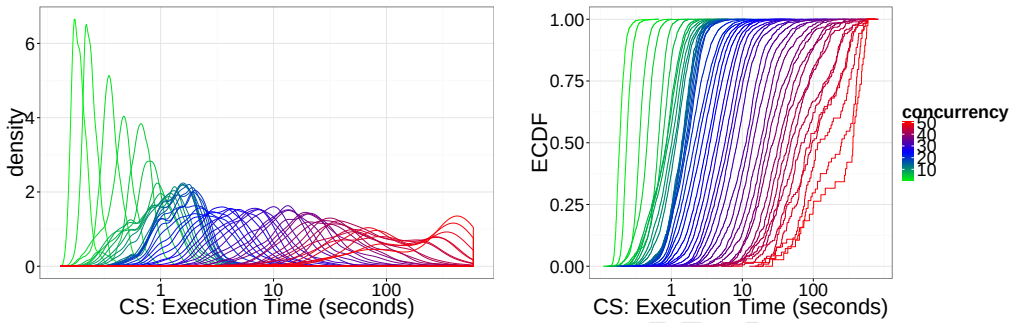


Figure 5: Relations between dEIS Services

Fig. 5 provides an overview of the overall EIS topology. We shortly present the structure of the EIS system used, with more details found in [19], [17]. This class of systems is representative for core enterprise management systems, such as ERP [6].

As representative dEIS cloud-distributed application we used the one described in ([3], [17], [14], [19], [20]). The targeted dEIS system is composed of four core services: one (or more) Thin Clients (CS), a Load Balancer (LB), one or more Worker services (WK), and one or more Database Storage services (ST). Each service runs in its own VM and communicates asynchronously with the other services using a distributed service messaging bus (CXF [25]). The communication between the services located in different VMs is handled by a Distributed OSGi (d-OSGi)[26] registry. The ST service contains a TPC-H [27] generated database.

The CS service contains the graphical user interface, as well as logic for initiating data sessions and issuing requests. The LB service provides load balancing logic, while also maintaining session information about connected clients. The LB's VM also hosts the d-OSGi service registry. The WK services implement data queries, analysis, transactional and arithmetic logic for the application. The ST service contains interfaces and mechanisms for



(a) (b)

Figure 6: CS Service Execution Time vs. Concurrency. a) Density Plot. b) ECDF Plot

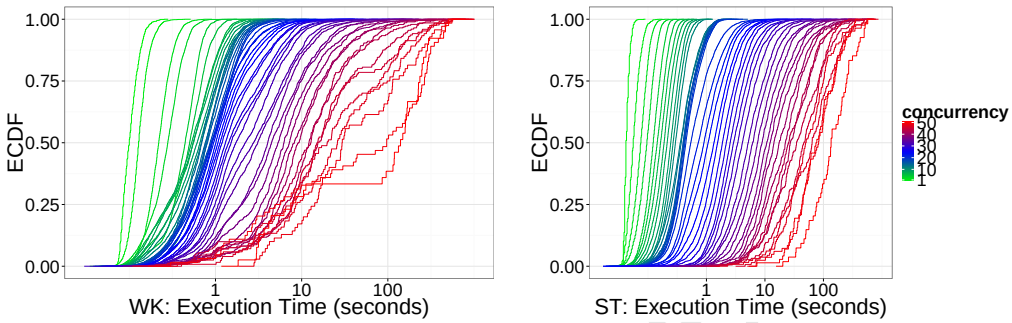
creating, reading, updating and deleting store data. A detailed presentation of the performance model of dEIS can be found in [3].

5.3. Little's Law-Based Application Performance Benchmarking

In order to build a performance profile of the dEIS application we will use the results of applying Little's Law to stable instances of the dEIS system. Intuitively, as long as the arrival rate (λ) of dEIS requests remains below the maximum processing capacity of the system, we expect the average execution time (W) to increase linearly. After a certain value of λ will be exceeded, W will begin increasing exponentially due to accumulation of jobs in the system, leading to a drop in the value of system's throughput (Th).

Mainly, we want to identify the dependency between the average execution time and the system's throughput at constant concurrency (occupancy). For this we will run 50 batches of benchmark tests, where the overall system's concurrency is kept constant for 10 minutes at a value $\lambda \in (1...50)$, before dropping back to 0. This ensures that we will get an accurate picture on (1) the distribution of execution times at all the dEIS services, and (2) the average achievable throughput corresponding to λ.

Figure 6 plots (on a logarithmic scale) the execution time (W) in seconds against the concurrency (L) measured at the CS service. Figure 6a shows the density distribution function of the execution time, while Figure 6b shows its empiric cumulative distribution function (ECDF). It is easy to notice on the ECDF plot that for concurrency values above 30 all the execution times (W) are above 10 seconds. Above this concurrency level (L), any small increase in L will produce a very large increase in W.



(a) (b)

Figure 7: ECDF Plot of Service Execution Time vs. Concurrency. a) WK service. b) ST service.

Figure 7a further explains the rapid increase in the mean of the system's execution time after concurrency level 30, by plotting the ECDF distribution of the W for each L level, for the WK service. Similarly, Figure 7b shows the EDCF plot per concurrency level of W for the ST service. The plots also suggest that a management system should not allow the dEIS system to slide into a region with concurrency (L) above 20, as the execution time will increase very fast with only slight increases in L. This corresponds to the dense green ECDF curves on the WK plot, respectively to the dense blue ECDF curves on the ST plot.

WK				**ST**			
L	W	λ	Th	L	W	λ	Th
0.85	106.25	118.0	133.40	0.64	44.1	118.0	133.37
1.49	135.89	236.0	263.61	1.47	59.5	236.0	260.45
1.97	236.68	333.3	366.39	1.91	78.5	333.3	363.10
2.47	296.38	355.5	392.94	2.23	99.7	355.5	395.65
3.01	461.58	315.7	362.82	2.77	129.5	315.7	381.52
3.60	532.35	358.8	406.07	2.95	152.4	358.8	418.43
4.39	615.45	377.8	419.50	3.05	175.7	433.7	433.72
5.11	619.07	414.8	462.41	3.12	201.7	476.1	476.12
5.83	666.97	435.4	483.38	3.30	229.4	435.4	497.15
6.63	733.06	442.2	491.27	3.55	262.0	442.2	510.37

Table 1: Dependency between average concurrency (L), average execution time in milliseconds (W), average arrival rate (λ), and average throughput (Th) for WK and ST services

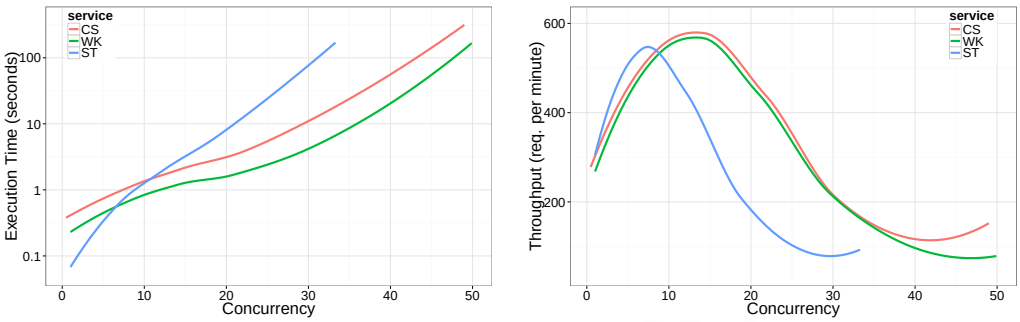


Figure 8: Per Service a) Average Concurrency vs. Average Execution Time. b) Average Concurrency vs. Average Throughput

After executing the benchmark described in this section, we produce a lookup table for the WK and ST dEIS services, linking the average system's occupancy (L), average execution time (W), the average arrival rate λ, and the average throughput (Th). Later, in Section 6 we will present an algorithm using this lookup table as input for ensuring that the dEIS system is properly scaled so that it can handle the volume of workload directed at it.

Ten of out fifty entries in the lookup table are shown in Table 1. The entries in the table approximately obey the Little's Law, which is due to how the average values of the W, λ, Th, and L metrics were calculated, especially the system's occupancy. In the actual VM deployment, the metrics were averaged over a sliding time window of one minute. This, combined with the fact that the arrival rate was not constant due to the constraint of having the CS-occupancy (almost) constant, produced slight variations in the averaged values, which are not influencing the SLA-based scaled algorithms as only the values for W and Th are used as inputs.

However, it is worth observing that indeed, the execution time increases with the increase in system's occupancy. Figure 8a shows the relation between W and L for CS, WK and LB dEIS services, on a logarithmic scale for W. This points to the fact that W starts increasing fast after L equal to 15 measured at CS service. This rapid increase of W is explained by Figure 8b, which shows that the service's throughput starts dropping after a certain value of the service's occupancy (L). The change in Th's trend happens around $L = 15$ for CS and WK services, and around $L = 8$ for ST service.

This delay between the occupancy's levels at ST and WK services is due to the fact that a job is processed twice [3] by the WK service, but only once by the ST service, causing the occupancy at the ST service be lower than the occupancy at the WK service. This difference is not noticeable between CS and WK service because of the smaller processing time in the second computing round [3] at CS service.

6. SLA-Based VM-Scaling Algorithms for CMS

As we have seen in the previous section, the processing capacity of dEIS applications can be easily saturated if the system's occupancy (L) approaches a critical region. Once the system enters into this hazardous region, the average execution time (W) will quickly increase from below one second to tens of seconds, lowering the quality of experience as a result of large delays in the processing of dEIS-requests.

In order for the CMS to prevent this behaviour where the quality of experience drops below acceptable limits we take the following actions: (1) we define a SLA specifying the maximum average execution time for the dEIS service, and (2) we enable the CMS to use a SLA-based VM-Scaling policy for ensuring that the application's distributed processing capacity is appropriately sized for handling the incoming flow of requests.

In [4] we have presented a reactive SLA-based VM-scaling algorithm that monitors the average execution time (W) of a service and compares it with a SLA-defined maximum value W_{max}. As soon as W exceeds 80% of W_{max}, the algorithm will trigger creating one additional VM for processing the extra workload.

The problem with this scaling approach is that when the incoming workload leads to an increase of the average system's occupancy (L), placing L in the critical region where W will start increasing very fast. Additionally, the latency associated with starting a new VM might cause a very large accumulation of workload, as the newly arrived requests could use the entire system's processing capacity. This, in turn, will result in a very large value for the system's occupancy, leading to even higher average execution times. When the newly created VM will begin processing jobs, all the new requests will be directed at it (as the load balancer will select it as it will have the lowest value of allocated workload), quickly saturating it and creating the conditions for a new VM scale-out. This process will repeat itself as long as the flow of incoming requests will continue to be greater than zero. If there

will be no more requests coming for a period long enough for the accumulated requests to finish their execution, then the system might recover.

In order to prevent this degradation in system's performance, we designed a new SLA-based VM-Scaling algorithm using the results of applying Little's Law. This new scaling algorithm uses relations between the average execution time (W), system's average occupancy (L), and system's average throughput (Th) for ensuring that the processing capacity dEIS system remains in a safe region, where the incoming workload is being processed without leading to an unsafe increase in system's occupancy.

Additionally, we enhanced the throughput-based VM-Scaling algorithm by adding prediction capabilities to it. Next, we will describe both SLA-based VM-Scaling algorithms.

6.1. λ-Based VM-Scaling Algorithm

Algorithm 2 describes the steps taken for sizing the number of VMs based on the current average arrival rate of requests, the current average system's throughput (Th), the maximum value for the average execution time defined by SLA, and the benchmark-obtained value of the Th.

The algorithm receives as input the SLA containing the maximum execution time (W_{max}) across all the VM instances of the considered service. As part of the initialisation sequence, the algorithm will first search in the tuples (λ, W, L) displayed in Fig. 8 for the benchmark entry (e) with the average execution time ($e.W$) closest, but lower, than the SLA threshold (W_{max}). The maximum value for the throughput will be stored in Th_{max}, and if no such value exists, then the program's execution will be terminated in line 8.

The scaling algorithm will be executed every N seconds. In line 11, the average arrival rate (λ) during the last minute will be compared to 80% of the maximum throughput (Th_{max}) multiplied with the current number of VMs (vm). If $\lambda > 0.8\, vm\, Th_{max}$ then in line 12 the number of VMs (vm^*) necessary for processing this workload will be calculated as the upper part of the division of λ by $0.8Th_{max}$. If vm^* is greater than the current number of VMs (vm) plus the current number of VMs being instantiated (vm^+), then the current scale-out scaling step (out) is calculated as the difference between the planned number of VMs (vm^*) and the total number of VMs, including the ones being instantiated ($vm + vm^+$).

ALGORITHM 2: λ-Based VM-Scaling Algorithm

Data: SLA and *benchmark* table containing (L, W, Th, λ) tuples

1 $W_{max} \leftarrow W\ defined\ in\ SLA$ and $Th_{max} \leftarrow 0$;
2 **for** $e \in benchmark$ **do**
3 **if** $e.W < W_{max}$ ***AND*** $e.Th > Th_{max}$ **then**
4 $Th_{max} \leftarrow e.Th$;
5 **end**
6 **end**
7 **if** $Th_{max} = 0$ **then**
8 terminate execution
9 **end**
10 **repeat** *every N seconds*
11 **if** $\lambda > 0.8\, vm\, Th_{max}$ **then**
12 $vm^* \leftarrow \left\lceil \frac{\lambda}{0.8 Th_{max}} \right\rceil$;
13 **if** $vm^* > vm + vm^+$ **then**
14 $out \leftarrow vm^* - vm - vm^+$;
15 **end**
16 **else if** $\lambda < 0.3\, vm\, Th_{max}$ ***AND*** *time(last scaling)* $<$ *cool-down* **then**
17 $vm^* \leftarrow \left\lceil \frac{\lambda}{0.8 Th_{max}} \right\rceil$;
18 **if** $vm^* < vm - vm^-$ **then**
19 $in \leftarrow vm^* - vm - vm^-$;
20 **end**
21 **end**
22 **if** $Th > 0.8\, vm\, Th_{max}$ **then**
23 $vm^* \leftarrow \left\lceil \frac{Th}{0.8 Th_{max}} \right\rceil$;
24 **if** $vm^* > vm + vm^+ + out$ **then**
25 $out \leftarrow vm^* - vm - vm^+$ **AND** $in \leftarrow 0$;
26 **end**
27 **end**
28 perform scaling
29 **end**

Similarly, the conditions for scale-in are checked in line 16. If λ is lower than 30% of the maximum throughput of all VMs then the optimal number of VMs (vm^*) is calculated in line 17 as upper part of the division of λ by

$0.8Th_{max}$. If vm^* is lower than the current number of VMs (vm), minus the number of VMs currently being decommissioned (vm^-), than the VM scale-in step is calculated as $vm^* - vm - vm^-$. The scale-in operation is executed only if there was no scale-out or scale-in in the last *cool-down* seconds (e.g. 30).

Finally, in line 22, the average throughput (Th) during the last minute is compared to 80% of the maximum throughput across all VMs, and if it is larger, then the planned number of VMs (vm^*) is calculated as the upper part of the division of Th by $0.8Th_{max}$. If vm^* is larger than the current number of VMs (vm), plus the number of VMs being instantiated (vm^+) and the planned scale-out, then the new scale-out step is $vm^* - vm - vm^+$. This prevents that the current processing capacity of all VMs is exceeded by a rapidly increasing incoming workload.

Next, the VM manager will be informed about either scaling-out or scaling-in the number of VMs. In case of dEIS, the scale-in operation is coordinated with the LB service, so that no further workload is directed at the VMs selected for decommission.

6.2. Predictive λ-Based VM-Scaling Algorithm

In case of "reactive" scaling, the monitoring algorithm detects the scale-out or scale-in conditions and then it informs the VM manager for performing the scaling. However, as the creation of new VMs is not performed instantly, there will be a time window during which the system will be in a state in which the SLA-defined conditions might be violated.

Given the delay in instantiating VMs, it is beneficial to predict the conditions for scale-out and and to initiate the scale-out operation in advance so that the VMs are already operational at the time they will be needed.

In order to test our assumptions, we modified the Algorithm 2 to include the prediction of the arrival rate, and to trigger scaling-out the VMs before the actual workload reaches the scale-out condition. Algorithm 3 lists the details of the predictive scaling algorithm. As multi-step prediction method we use the one presented in Section 4.

We initialise the algorithm by first determining the value for the maximum throughput (Th_{max}) given (1) the maximum execution time (W_{max}) defined in the SLA, and (2) the benchmark-obtained value for the Th. Also, the regression model (RM) is set to null, and the number of predicted VMs ($Pred$) is initialised with an empty set.

ALGORITHM 3: Predictive λ-Based VM-Scaling Algorithm

Data: SLA contract with the maximum execution time

1 $RM \leftarrow NULL$;
2 $Pred \leftarrow empty$;
3 determine Th_{max};
4 **repeat** *every N seconds*
5 $\quad \Lambda \leftarrow \{\lambda(i) | i < t - M\}$;
6 $\quad T \leftarrow sampling(\Lambda)$;
7 $\quad$ **if** *RM is not NULL* **then**
8 $\quad\quad MAPE = accuracy(RM, T, \Lambda)$;
9 $\quad\quad$ **if** $MAPE < \epsilon_{max}$ **then**
10 $\quad\quad\quad T^* \leftarrow (t+1, t+2, \cdots, t+N+D)$;
11 $\quad\quad\quad \Lambda^* \leftarrow predict(RM, T^*)$;
12 $\quad\quad\quad Pred[T^*] \leftarrow \left\lceil \frac{\Lambda}{0.8 Th_{max}} \right\rceil$;
13 $\quad\quad$ **else**
14 $\quad\quad\quad Pred \leftarrow empty$;
15 $\quad\quad\quad$ drop RM;
16 $\quad\quad$ **end**
17 $\quad$ **end**
18 $\quad RM \leftarrow regression(\Lambda, T)$;
19 $\quad$ **if** *Pred[t+D] exists* **then**
20 $\quad\quad out \leftarrow max(0, Pred[t+D] - vm - vm^+)$;
21 $\quad$ **end**
22 $\quad$ **if** $out = 0$ ***AND*** $vm^+ = 0$ **then**
23 $\quad\quad$ calculate scale-in step using another algorithm;
24 $\quad$ **end**
25 **end**

Next, the management loop starts, by repeating the following operations every N seconds. Let t be the current time in seconds. In line 5, the values for λ in the time window $(t, t-M)$ are retrieved and stored in Λ. T is then set to the sampling time of the values in Λ.

If RM has already been calculated (is not $NULL$), then it will be used for predicting the arrival rates corresponding to the time moments in T, and then the out-of-sample accuracy ($MAPE$) will be calculated using the actual values from Λ. If $MAPE$ is below a threshold (ϵ_{max}), then a prediction (Λ^*) of the next $N + D$ seconds will be calculated, where D is the time necessary

for a VM to be instantiated and to become operational. Next, in line 12 the number of VMs at time $(t+1, t+2, \cdots, t+N+D)$ is calculated using the method described in Section 6.1, Algorithm 2. If the prediction accuracy is higher than ϵ_{max}, then the predicted number of VMs is dropped, ensuring that no scaling decision is taken based on unreliable information.

Next, in line 18, the regression model RM is calculated using Λ and T, by applying the method described in Algorithm 1, where the current values for Λ and T are appended to the previous ones, as long as their trend is maintained, as explained in Section 4.

Next, in line 19 a check is made for determining if a prediction exists for time $t+D$. By looking at the necessary number of VMs D seconds in advance, we ensure that any VM needed in the near future will actually be ready at that time. If the prediction exists, then a scale-out step is calculated as the difference between the predicted number of VMs ($Pred[t+D]$) at time $t+D$ and the total number of VMs (vm), including the ones currently being instantiated (vm^+).

If the algorithm determines that no scale-out is needed and there are no VMs currently being instantiated, then in line 23 the scale-in step will be calculated using the Algorithm 2.

7. Evaluation Results

In order to evaluate the two new SLA-based VM-Scaling algorithms previously presented in Sections 6.1 and 6.2 we implemented them in CloudSim [28], which allowed us to run multiple simulations against the dEIS distributed application.

Next, we describe some implementation details about the integration of the new scaling algorithms in CloudSim, and the implementation of the prediction mechanisms. We continue with comparing the λ-based and predictive λ-based VM-Scaling algorithms using a synthetic workload. Finally, we describe a real-world scenario where the incoming workload received by a system grew with four orders of magnitude, and then we use this workload-description to simulate and compare the three scaling algorithms described: reactive, λ-based and predictive λ-based.

7.1. Implementation of Simulations

For building a system able to compare the SLA-based VM-Scaling policies, we extended our dEIS CloudSim-based simulator presented in [3] and

[4]. We added two new CloudSim scaling policies for the algorithms described in Sections 6.1 and 6.2, supported by two new additional monitoring metrics (arrival rate and throughput) at the CloudSim datacenter-broker level.

For integrating the multi-step prediction mechanism described in Section 4 we wrote a series of analytic scripts in R [23] (for calculating the linear regression model, prediction from the regression model, and prediction's accuracy), which were invoked by our CloudSim extended datacenter broker using the TCP/IP Rserve [29] library. Given that Rserve does not allow transferring of arbitrary complex R objects, we had to cache the regression's results in R, by allocating unique model identifiers to each pair of cloud tenant and service. In this way, CloudSim would transmit the model identifier when a regression model was calculated, and then the same identifier would be used when CloudSim required calculating a prediction based on the regression model.

7.2. Evaluation of the Predictive VM-Scaling Algorithm

In order to evaluate the predictive VM-scaling algorithm, we created a CloudSim simulation with 500 available servers in which the workload was increased linearly from 60 requests per minute to 1000 requests per minute during 1200 seconds, and then it was linearly decreased back to 60 requests per minute during 400 seconds. We ran the simulation first with the λ VM-Scaling algorithm, and then with the predictive-λ VM-Scaling algorithm. The prediction algorithm used a forecasting window of 50 seconds. The simulated time for instantiating a VM was 18 seconds for the WK VM, respectively 23 seconds for the ST VM. The maximum error value (ϵ_{max}) for considering a prediction as valid was set to 1.5%. The VM-Scaling algorithm was executed every 5 seconds.

Figure 9 shows the distribution of workload across simulation's duration, measured at the WK service. The prediction horizon refers to the time distance from the time moment when the prediction is calculated to the time when the workload is forecasted. The black line represents the actual value of the workload calculated over a moving window of one minute. As it can be seen, the prediction closely follows the actual values of the workload. At simulated time 1200, when the workload's trend changes from increasing to decreasing, it can be observed that the prediction continues to increase, however, it quickly realigns itself with the new direction. This confirms that the prediction mechanism used was appropriate for predicting this type

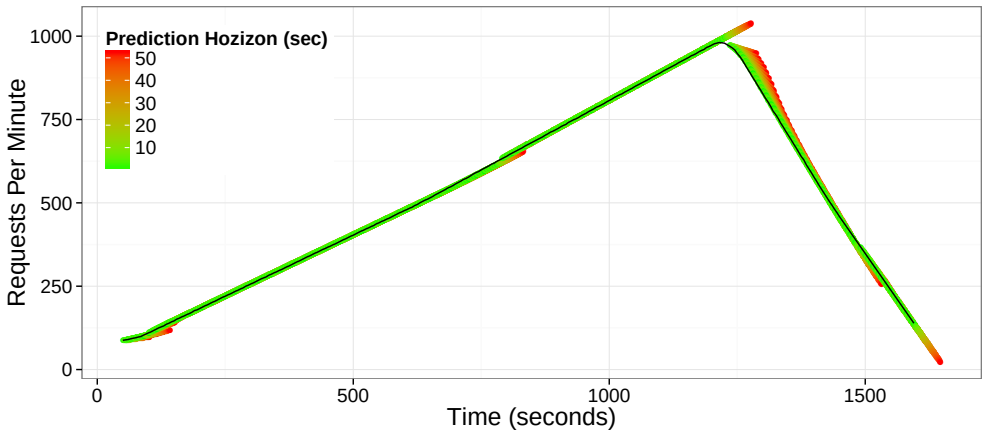
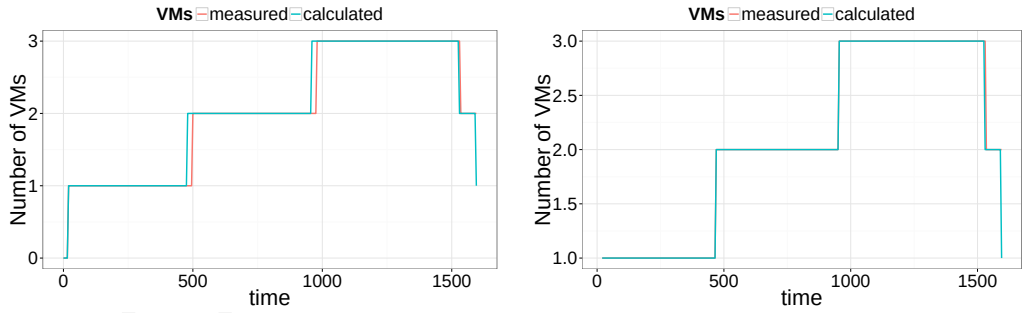


Figure 9: Actual Arrival Rate (black) vs. Predicted Arrival Rate

of workload, and that incorrect predictions do not affect the algorithm's correctness as the predictions are only used for scale-out and not for scale-in.

Figure 10 shows the actual evolution of the number of VMs belonging to the WK service. In the simulation shown in Figure 10a we used the λ-based VM-Scaling algorithm, while in Figure 10b we used the predictive λ-based VM-Scaling algorithm. It can be seen that in Figure 10b the plot of the actual number of VMs is perfectly aligned with the calculated number of VMs (the two lines overlap), while in Figure 10a there is a noticeable gap



(a) (b)

Figure 10: Evolution of the number of VMs of the WK service in case of applying a) λ-based scaling algorithm b) predictive λ-based scaling algorithm.

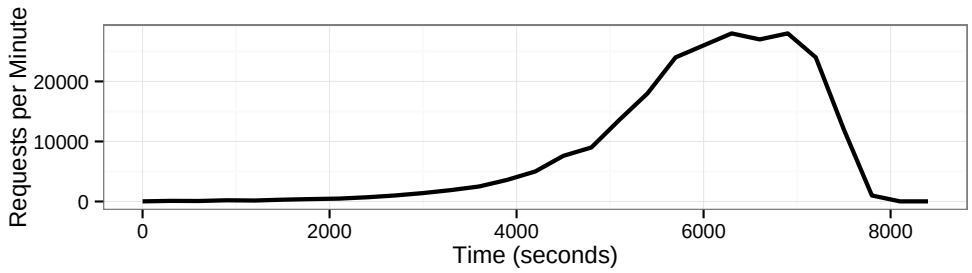


Figure 11: "Schwingen"-Like dEIS Workload

between the calculated and the actual measured number of VMs at times 500 and 1000, caused by the delay of 18 seconds in instantiating the WK VM.

These two simulations have shown that it is advantageous to use prediction in conjunction with the λ-based VM-Scaling algorithm. The prediction-enabled VM-Scaling algorithm has the advantage of eliminating the effect of the delays when scaling-out the VMs, as VMs become operational at exactly the right moment compared to the increase in the workload responsible for triggering the scale-out.

7.3. Real-World Application Scenario

For testing the ability of the VM-scaling algorithms to dynamically increase and decrease the number of VMs allocated to distributed services, while also complying with the SLAs regarding the maximum value of the execution times, we have selected a simulation scenario based on a real-world event [30].

In order to cover a Schwingen (Swiss sport) event and for reporting real-time on the performance of sportsmen, as well as live-tracking the scores of the fights, a cloud computing infrastructure was prepared and a set of mobile applications were developed. The app combined real-time time processing and analytics with a mobile platform, while running in the cloud environment.

However, due to the huge success of the event, the mobile application was downloaded 70000 times, creating a very large load for the computing infrastructure and the network. This, combined with sub-optimal scaling of the computing infrastructure (according to [30]), led to some very poor performance on the mobile side, with very long waiting times.

We attempt to simulate a similar load for the dEIS distributed application, by creating a workload of up to 28000 dEIS-users (each user corresponding to a CS-issued request), while at the same time keeping the combined response time for the WK and ST services below one second. Figure 11 shows the distribution of workload across time. The workload first increases to 500 concurrent users during 1800 seconds, then to 1000 users in 700 seconds, then it approximately doubles at every 600 seconds, until it reaches 11000 users at time moment 4800. From there on the workload increases with roughly 10000 users at every 1600 seconds, reaching 28000 users at time 6300. The workload will stay at this level for about 600 seconds, after which it begins decreasing to towards 25 users during approximately 1200 seconds. In total we simulate 8400 seconds, or 2 hours and 20 minutes.

The goal of the simulation scenario is to test the ability of the VM-scaling algorithms to maintain the execution time below the one specified in the SLA, and implicitly, to prevent the dEIS system from becoming overloaded with requests.

7.4. Comparison of VM-Scaling Algorithms

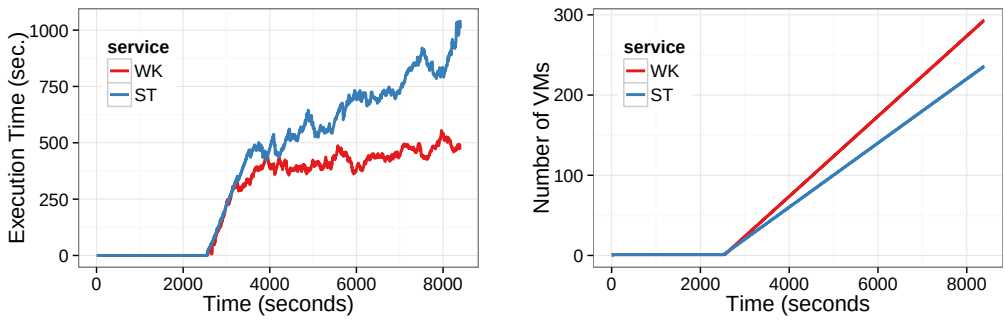
For comparing the presented VM-Scaling algorithms, we used the workload described in Section 7.3 together with a SLA policy defining a maximum combined response time of 1 second, divided between WK and ST services with a ratio of 7:3. The resulting SLA specified a maximum execution time of 700ms for the WK service, respectively 300ms for the ST service.

The workload was simulated against the dEIS model [4] (constructed using recorded dEIS monitoring traces) [3] in the CloudSim simulator.

We set to compare the *reactive* SLA-based VM-Scaling algorithm [4] with the *λ-based* algorithm described in Section 6.1, and the *predictive λ-based* algorithm presented in Section 6.2. For all three algorithms we will analyse the performance of WK and ST service, by considering the distributions of (1) execution times, (2) the rate of incoming, processed, and dropped requests, and (3) the total number of VMs.

7.4.1. Reactive SLA-Based VM-Scaling Algorithm

We first tested the reactive SLA-based VM-Scaling algorithm, by initially simulating creating one VM for each of the dEIS application's services (CS, LB, WK and ST). After all 4 VMs were created, the workload generator was started at the CS service, which began generating requests according to the workload pattern shown in Figure 11.



(a) (b)

Figure 12: Simulation of the Reactive SLA-Based VM-Scaling Algorithm a) Execution time corresponding to WK and ST services. b) Number of VMs corresponding to WK and ST.

Figure 12a shows the average execution time (in seconds) measured at WK and ST services. At simulation time equal to 2400sec the *SLA ratio* parameter, calculated as the ratio between the average execution time (W) measured during the last 60 seconds and the SLA-defined maximum value of W, exceeded the scaling threshold of 0.8, triggering creation of a pair of WK-ST VMs, which were instantiated after 18, respectively 23 seconds, as it can be seen in Figure 12b.

The system's behaviour is explained by the fact that once the processing capacity of the available VMs is exceeded, the incoming workload will only delay the execution of the requests that are already being executed. This will lead to an increase for the execution time for all the requests. Combining this with the fact that the considered dEIS application does not queue requests (they begin their execution as soon as they are received), and that the workload never drops to zero during the simulation, helps explain why the execution time keeps increasing until the simulation's end.

Figures 13a and 13b show the number of inbound (red line), processed (blue line) and dropped (green line) requests at WK, respectively at ST services. Soon after simulation time 2400sec (where the first VM-Scaling for both WK and ST services was performed) it can be observed that the number of requests being processed by the dEIS distributed system drops below the number of inbound requests, creating an imbalance both at the WK and ST services.

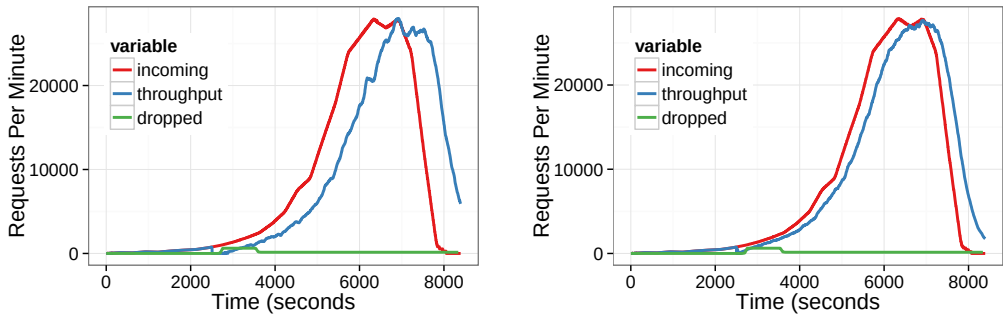


Figure 13: Distribution of the arrival rate (red), processing rate (blue), and dropped requests during the simulation of the Reactive SLA-Based VM-Scaling Algorithm for dEIS a) WK service. b) ST service.

The imbalance in processing of requests observed at both WK and ST services caused the accumulation of requests being processed, increasing the system's occupancy and leading to the increase in the average execution time (W), clearly visible in Figure 12a.

As the W metric continued to increase, the reactive SLA-based VM-Scaling algorithm continued to observe a *SLA ratio* value above the scaling threshold, triggering the continuous creation of VMs at 20 seconds intervals for the WK service, respectively 25 seconds for the ST service, as the VMs have a instantiation delay of 18 seconds (WK), respectively 23 seconds (ST), and the scaling algorithm was executed at every 5 seconds. This explains the ascending trend of the number of VMs from Figure 12b and the different slopes of WK and ST horizontal scaling plots.

Regarding the reason of this behaviour, it lies in the fact the dEIS distributed system was first scaled at a time when the average throughput was higher than the optimal one, as described in Section 5.3. This, combined with a constant increasing number of new arrivals, led to the continual accumulation of requests, and an imbalance in the dEIS system's inbound-outbound flow of requests.

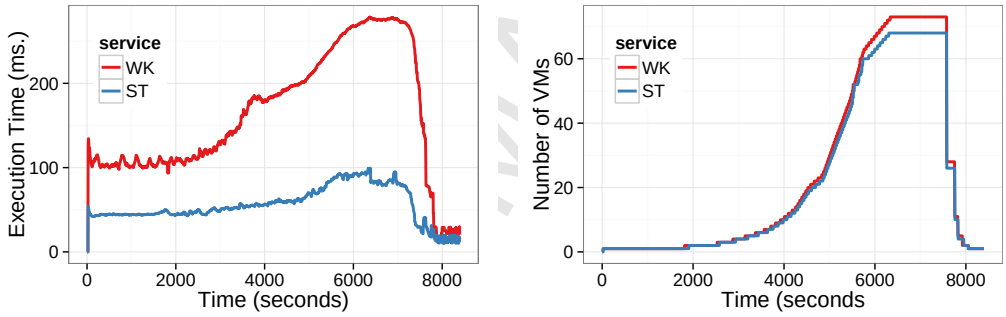
The processing imbalance remained present throughout the simulation, as it can be observed in Figures 13a and 13b, which show a difference between number of incoming and system's throughput until simulation's end. This also explains why the number of VMs in Figure 12b does not drop to 1 at

the end of simulation - because there were still requests being processed and the execution time was over the SLA-defined maximum value.

The defined SLAs were violated during 71% of the simulation's duration as the algorithm did not manage to keep the distributed system in a SLA-compliant state. This shows that the reactive SLA-based VM-Scaling is not suited for scaling system with fast-increasing workloads, however, we will show that the λ-based and predictive-λ-based VM-Scaling algorithms are very suited for such tasks.

7.4.2. λ-Based VM-Scaling Algorithm

The λ-based VM-scaling algorithm was validated with a simulation of the same workload described in Section 7.3, Figure 11, which has been also used in the previous subsection as well.



(a) (b)

Figure 14: Simulation of the λ-Based VM-Scaling Algorithm a) Execution time corresponding to WK and ST services. b) Number of VMs corresponding to WK and ST.

The simulation started with one VM per dEIS-service and the target SLAs defined a maximum execution time of 700ms for the WK service, respectively of 300ms for the ST service.

Figure 14a shows the average execution time measured over a moving time window of one minute, which was well below the maximum limit set by SLA. The reason why this happened in contrast to the simulation presented in Section 7.4.1 is that this algorithm considered a maximum processing capacity per VM of 435 requests per minute for the WK service, respectively of 456 requests per minute for the ST service. These values were calculated

based on the defined SLAs and the results of the benchmark described in Section 5.3.

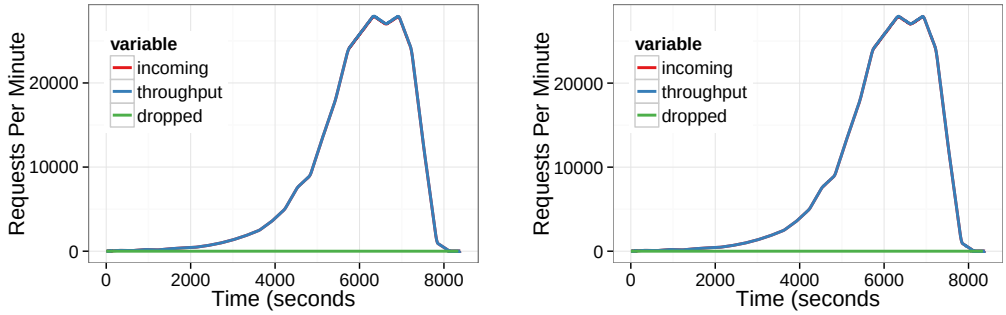


Figure 15: Distribution of the arrival rate (red), processing rate (blue), and dropped requests during the simulation of the λ-Based VM-Scaling Algorithm for dEIS a) WK service. b) ST service.

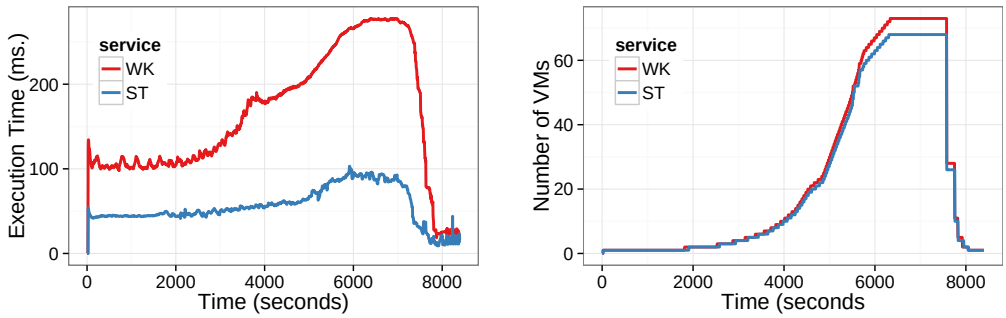
Figure 14b shows the evolution of the number of VMs of both WK and ST throughout the simulation. As it can be seen, the first scale-out was performed earlier than in the case of the reactive VM scaling algorithm, after 1815 seconds from the simulation's start, when the average arrival rate, equal to 388 requests per minute (for WK service), exceeded the algorithm's scaling capacity threshold (80% of 435 req. per minute). It is worth observing that during the time when the VM scaling-out was signalled and the actual time when the VMs become operational, the number of incoming requests continue to increase, however it did not exceed the services' processing capability. In section 7.4.3 we will show how this risk was also mitigated.

The maximum number of VMs corresponding to WK service was 73, while for the ST service, the number was 68, given the slightly higher processing capacity of ST service, according to the previously presented benchmark.

As shown in Figures 15a and 15b, the processing rate (throughput) of both WK and ST followed closely the arrival rate of requests, validating the algorithm's capacity of maintaining the distributed system in fully SLA-compliant state.

7.4.3. Predictive λ-Based VM-Scaling Algorithm

The predictive-λ-based VM-scaling algorithm was validated with the same workload used in the previous two simulations, described in Section 7.3.



(a) (b)

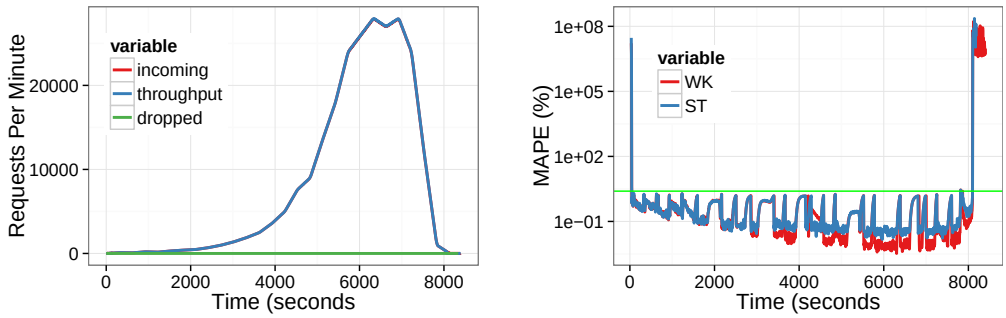
Figure 16: Simulation of the Predictive-λ-Based VM-Scaling Algorithm a) Execution time corresponding to WK and ST services. b) Number of VMs corresponding to WK and ST.

The simulation's results match closely the ones from the previous simulation described in Section 7.4.2 (Figure 14), given that the scaling algorithm extended the λ-based VM-scaling algorithm with prediction capabilities. The SLA compliance was 100% for both algorithms, but in the case of the the predictive λ algorithm the scaling-out of VMs was scheduled to happen so that the VMs become operational *exactly* at the time signalled by the control algorithm. This increased the system's protection to sudden increases in workload and kept the spare processing capacity constant instead of diminishing it due to the delay in VMs' instantiation.

Figure 16a shows that the average execution time for both the WK and ST services stayed below the SLA-defined maximum values of 700ms for the WK service, respectively 300ms for the ST service. Figure 16b shows the evolution of the number of VMs for both WK and ST services, which is very similar to the one presented in Figure 14b, with the difference that in the case of the predictive-λ-based VM-scaling algorithm the instantiation of VMs happened *exactly* at the moment determined by the scaling algorithm due to predicting the rate of incoming requests.

Figure 17a presents the evolution of processing capacity of the WK service, which was perfectly balanced with the incoming workload. Figure 17b shows the prediction accuracy, which remained below the threshold of 2.5% (green line) most of the time, enabling using the results of the forecasting as input for the VM-scaling algorithm.

The algorithm achieved maintaining a SLA compliance rate of 100% per-



(a) (b)

Figure 17: a) Distribution of the arrival rate (red), processing rate (blue), and dropped requests during the simulation of the Predictive-λ-Based VM-Scaling Algorithm for the dEIS WK service. b) Accuracy of predicting the arrival rate at the WK service.

cent, while also creating a larger margin for the variation of the arrival rate, as the whole system was scaled faster due to the use of prediction.

8. Conclusions

Cloud Computing has evolved to become an enabler for delivering access to large-scale distributed applications running on managed environments composed of network-connected computing systems. This makes possible hosting Distributed Enterprise Information Systems (dEIS) in cloud environments, while allowing Cloud Management Systems (CMS) to enforce strict performance and quality of service requirements, defined using Service Level Agreements (SLA).

In this paper we presented two new VM-scaling algorithms focused on dEIS systems, which can be used by cloud infrastructure management systems to optimally detect the most appropriate scaling conditions using performance-models of distributed applications derived from constant-workload benchmarks. We have shown how to combine benchmark results, with Little's Law and SLAs for identifying the optimal processing capacity of cloud services, and then we used it in a SLA and arrival rate-based VM-scaling algorithm.

We have also extended the arrival rate-based VM scaling algorithm to consider prediction of the arrival rate, by using linear regression and multi-step forecasting.

We have evaluated a total of three VM-scaling algorithms by simulating them in a cloud simulator against trace-based performance models of dEIS, using a real-world application scenario involving a large variable number of users. Our results show that using predictive SLA-driven scaling algorithms in cloud management systems for guaranteeing performance invariants of distributed cloud applications improves the management efficiency of infrastructure-cloud management systems, as opposed to using only reactive SLA-based VM-scaling algorithms.

Appendix A. Statistical Linear Models

A linear statistical model (or linear regression model) represents the mathematical relation between a dependent variable Y and one or more predictor variables X, as shown in Equation A.1.

$$y = x_0 + \beta X + \epsilon \tag{A.1}$$

y is the dependent value, x_0 is the intercept (free term), β is a transposed vector of scalar coefficients, X is a vector of independent variables, and ϵ is the error term.

If Y is a scalar variable and X is also a scalar variable, then the regression is called *simple*. If X is a vector of independent variables, the regression is called *multiple*. If both X and Y are vectors then the statistical model is called *general linear model*.

When the modelled variable is independent and it is represented as a time series, we are dealing with *autoregression*, as shown in Equation A.2.

$$Y_t = y_0 + \beta Y_{t-1} + \epsilon \tag{A.2}$$

Y_t and Y_{t-1} are values of variable Y at time t and $t-1$ respectively, y_0 is the model's intercept term, ϵ is the error term, and β is the regression coefficient.

By assuming that the model's accuracy is *acceptable* over a time window of length p, it is possible to predict the next n values of Y by applying n times the formula in Equation A.2, starting with Y_t. A linear model's accuracy is defined in terms of errors, as described in Appendix B. In Section 4 we show an example of how autoregression can be applied for predicting the time series shown in Figure 1.

Appendix B. Errors in Statistical Models

The linear model calculates for each value X_t of the independent variable X a corresponding $\hat{X}_t$ value according to equation A.2. The difference between X_t and $\hat{X}_t$ represents the regression error. By applying the regression over a time series X, a series of error values called *residuals* are produced. When the errors are calculated between the predicted values and the actual values of the independent variable, the errors are then called *out of sample* errors. Residuals are important for calculating the accuracy of modelling,

while the out of sample errors are important for calculating the prediction's accuracy.

We present two metrics for quantifying the regression errors: Root Mean Standard Deviation (or Error) and Mean Absolute Percentage Error.

The Root Mean Standard Deviation ($RMSD$) is a scale-dependent measure (dependent on the variable's maximum value) of estimation error, equal to the square root of the mean square error, as shown in Equation B.1.

$$RMSD = \sqrt{\frac{\sum_{t=1}^{n}(\hat{Y}_t - Y_t)^2}{n}} \tag{B.1}$$

The Mean Absolute Percentage Error ($MAPE$) is calculated as the mean of the error's modulus, as shown in Equation B.2

$$MAPE = \frac{\sum_{t=1}^{n}|\hat{Y}_t - Y_t|}{n} \tag{B.2}$$

and it expresses accuracy as percentage. We will use $MAPE$ error for calculating the regression's accuracy and for deciding whether the prediction can be used for SLA-based scaling in Section 6.

Appendix C. Statistical Non-Linear Models

Non-linear regression is a form of regression in which the mathematical relation describing the dependency between the predictors and the dependent variable is a non-linear combination of parameters, for example exponential addition. Exponential smoothing is an example of non-linear modelling, where a time series is *smoothed* by approximating its terms with computations of an exponential function.

While non-linear models are more suitable to represent complex patterns in time series, they also introduce additional complexity as their output usually depends on correctly identifying the underlying signal's period, needed by some prediction algorithms for decomposing the signal into its seasonal, trend and noise components. Also, some non-linear prediction algorithms, such as Holt-Winters [31] require having a time series with a length of at least two periods.

Let us present a simple example. Figure C.18 shows a time series containing a repetitive pattern (closely resembling the more complex patterns usually found in business-critical enterprise applications), with some added

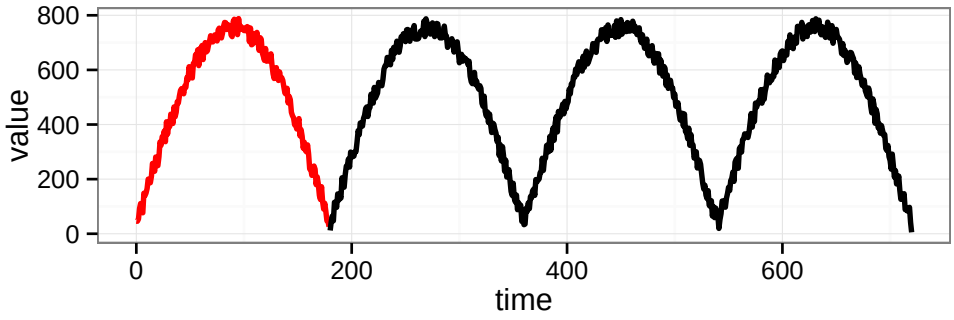


Figure C.18: Sample Time Series with a Periodic Pattern

noise. The first period is coloured red. The data was generated by calculating the absolute value of function *sinus* multiplied with a constant factor, as shown in Equation C.1.

$$Y = 1 + 720 \left| sin(x\frac{\pi}{180}) \right| + \epsilon, x \in (1...720) \tag{C.1}$$

There are multiple ways of extracting the period of a signal, for example by using the autocorrelation function as in [17], or by calculating the estimated spectral density of the time series using the periodogram [32] and then converting the frequency with the largest spectral density into the signal's period. Naturally, the period of Y is equal to 180 (e.g. seconds), as the function *sinus* has a period of 2π and $|sin(x)|$ has a period equal to π.

Figure C.19 shows the prediction output for the time series given above, by applying the Holt-Winters algorithm, first with the correct period of the time series (180), and then with an approximative value for the period (145). This shows the importance of correctly identifying the signal's period before attempting to predict it, which can be difficult in practice. The lower image shows a bad prediction caused by the incorrect detection of the trend of the time series, caused by the shorter value of the period - the signal being identified as having 6 periods instead of 5.

The non-linear prediction models are suited for forecasting time series with seasonal variations, such as hourly, daily or monthly. The predictions can then be used for optimising processes with longer time horizons [11], such as the allocation of physical computing resources [14]. For the work presented in Section 6 we will use a combination of linear models and error

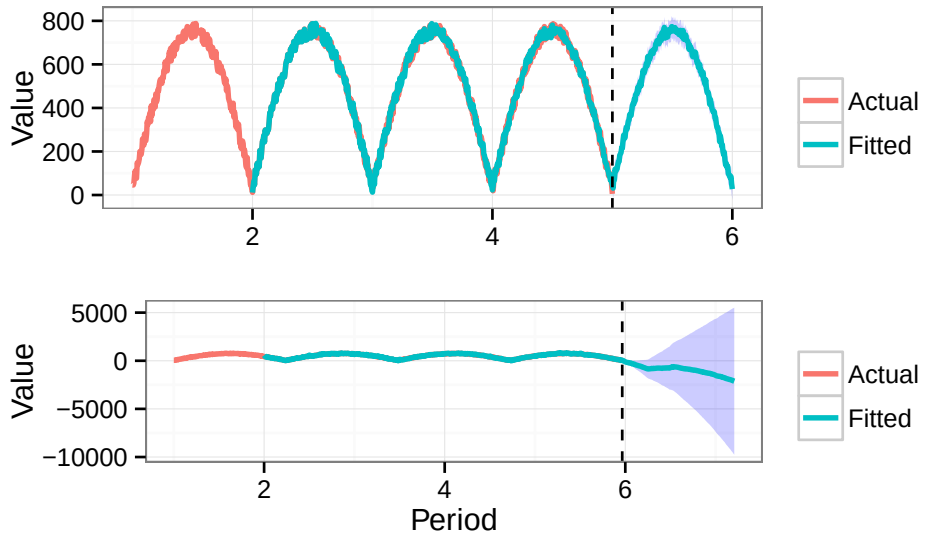


Figure C.19: Holt-Winters Prediction. Top: Signal's Period:180. Bottom: Signal's Period: 145

estimation, as the time horizon of the prediction is usually short, in the range of tens of seconds.

References

[1] P. Mell, T. Grance, The NIST definition of cloud computing, NIST special publication.

[2] D. Woods, Enterprise Services: Architecture, O'Reilly Media, Inc., 2003.

[3] A.-F. Antonescu, T. Braun, Modeling and simulation of concurrent workload processing in cloud-distributed enterprise information systems, in: ACM SIGCOMM Workshop on Distributed Cloud Computing (DCC 2014), 2014.

[4] A.-F. Antonescu, T. Braun, SLA-driven simulation of multi-tenant scalable cloud-distributed enterprise information systems, in: ACM PODC Workshop on Adaptive Resource Management and Scheduling for Cloud Computing (ARMS-CC), 2014.

[5] J. Bozman, Cloud computing: The need for portability and interoperability, IDC Analyze the Future.

[6] A. Leon, Enterprise resource planning, Tata McGraw-Hill Education, 2008.

[7] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, A. Ghalsasi, Cloud computing—the business perspective, Decision Support Systems 51 (1) (2011) 176–189.

[8] G. Lab, Cloud simulator cloudsim, `http://code.google.com/p/cloudsim` (2014).

[9] R. Buyya, et al., Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities, in: Int. Conf. on High Performance Computing & Simulation, 2009. HPCS'09., IEEE, 2009, pp. 1–11.

[10] S. K. Garg, et al., Networkcloudsim: Modelling parallel applications in cloud simulations, in: Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on, IEEE, 2011, pp. 105–113.

[11] A. Visan, M. Istin, F. Pop, V. Cristea, Bio-inspired techniques for resources state prediction in large scale distributed systems, International Journal of Distributed Systems and Technologies (IJDST) 2 (3) (2011) 1–18.

[12] S. Islam, J. Keung, K. Lee, A. Liu, Empirical prediction models for adaptive resource provisioning in the cloud, Future Generation Computer Systems 28 (1) (2012) 155–162.

[13] N. Roy, A. Dubey, A. Gokhale, Efficient autoscaling in the cloud using predictive models for workload forecasting, in: Cloud Computing (CLOUD), 2011 IEEE International Conference on, IEEE, 2011, pp. 500–507.

[14] A.-F. Antonescu, P. Robinson, T. Braun, Dynamic SLA management with forecasting using multi-objective optimizations, in: Proc. 13th IFIP/IEEE Symposium on Integrated Network Management (IM), 2013.

[15] A. García García, I. Blanquer Espert, V. Hernández García, SLA-driven dynamic cloud resource management, Future Generation Computer Systems 31 (2014) 1–11.

[16] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, R. Buyya, SLA-based virtual machine management for heterogeneous workloads in a cloud datacenter, Journal of Network and Computer Applications.

[17] A.-F. Antonescu, T. Braun, Improving management of distributed services using correlations and predictions in SLA-driven cloud computing systems, in: Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS), 2014.

[18] H. N. Van, F. Tran, J.-M. Menaud, SLA-aware virtual resource management for cloud infrastructures, in: Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on, Vol. 1, 2009, pp. 357–362. doi:10.1109/CIT.2009.109.

[19] A.-F. Antonescu, P. Robinson, T. Braun, Dynamic topology orchestration for distributed cloud-based applications, in: Proc. 2nd IEEE Symposium on Network Cloud Computing and Applications (NCCA), 2012.

[20] A.-F. Antonescu, A.-M. Oprescu, et al., Dynamic optimization of SLA-based services scaling rules, in: Proc. 5th IEEE Internetional Conference on Cloud Computing Technology and Science (CloudCom), 2013.

[21] J. D. Little, S. C. Graves, Little's law, in: Building Intuition: Insights From Basic Operations Management Models and Principles, Springer, 2008, pp. 81–100.

[22] F. E. Croxton, et al., Applied general statistics., Prentice-Hall, Inc, 1939.

[23] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria (2013). URL `http://www.R-project.org`

[24] F. Pop, M. Potop-Butucaru, Adaptive Resource Management and Scheduling for Cloud Computing, Vol. 8907 of Lecture Notes in Computer Science (LNCS) / Theoretical Computer Science and General Issues, Springer, 2015.

[25] Apache CXF, http://cxf.apache.org/ (2013).

[26] Distributed OSGi, http://cxf.apache.org/distributed-osgi.html (2013).

[27] T. P. P. Council, TPC-H benchmark specification, Published at http://www.tcp.org/hspec.html.

[28] T. Goyal, et al., Cloudsim: simulator for cloud computing infrastructure and modeling, Procedia Engineering 38 (2012) 3566–3572.

[29] S. Urbanek, Rserve – a fast way to provide r functionality to applications, in: Proc. of the 3rd International Workshop on Distributed Statistical Computing (DSC), 2003.

[30] ComputerWorld, SAP-schwing-app ein PR-GAU (engligh translation: SAP swing app a PR meltdown), Engligh translation: http://goo.gl/Zmmjwx Original webpage: http://www.computerworld.ch/news/software/artikel/sap-schwing-app-ein-pr-gau-64170/ (Sept 2013).

[31] C. C. Holt, Forecasting seasonals and trends by exponentially weighted moving averages, International Journal of Forecasting 20 (1) (2004) 5–10.

[32] P. Bloomfield, Fourier analysis of time series: an introduction, John Wiley & Sons, 2004.

Torsten Braun got his Ph.D. degree from University of Karlsruhe (Germany) in 1993. From 1994 to 1995 he has been a guest scientist at INRIA Sophia-Antipolis (France). From 1995 to 1997 he has been working at the IBM European Networking Centre Heidelberg (Germany) as a project leader and senior consultant. He has been a full professor of Computer Science at the University of Bern (Switzerland) and head of the research group "Communication and Distributed Systems" since 1998. He has been member of the SWITCH (Swiss education and research network) board of trustees since 2001. Since 2011, he has been vice president of the SWITCH foundation.

Florian Antonescu is a Research Associate in the department of Products & Innovation at SAP Switzerland. He is expected to receive his PhD from University of Bern (Switzerland) in 2015. Previously he obtained his Master in Management of Information Technology, and Diploma in Computer Science from University "Politehnica" of Bucharest (Romania). His research interests include distributed computing, scalability of cloud systems, large-scale statistical data analysis, and mobile computing. For his PhD he investigated the use of Service Level Agreements in cloud environments for scaling distributed infrastructures.

Florian Antonescu

Click here to download high resolution image