An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures

Ahmed Ali-Eldin, Johan Tordsson and Erik Elmroth
Department of Computing Science, Umeå University
Umeå, Sweden
Email:{ahmeda, tordsson, elmroth}@cs.umu.se

***Abstract*—Cloud elasticity is the ability of the cloud infrastructure to rapidly change the amount of resources allocated to a service in order to meet the actual varying demands on the service while enforcing SLAs. In this paper, we focus on horizontal elasticity, the ability of the infrastructure to add or remove virtual machines allocated to a service deployed in the cloud. We model a cloud service using queuing theory. Using that model we build two adaptive proactive controllers that estimate the future load on a service. We explore the different possible scenarios for deploying a proactive elasticity controller coupled with a reactive elasticity controller in the cloud. Using simulation with workload traces from the FIFA world-cup web servers, we show that a hybrid controller that incorporates a reactive controller for scale up coupled with our proactive controllers for scale down decisions reduces SLA violations by a factor of 2 to 10 compared to a regression based controller or a completely reactive controller.**

I. Introduction

With the advent of large scale data centers that host outsourced IT services, cloud computing [1] is becoming one of the key technologies in the IT industry. A cloud is an elastic execution environment of resources involving multiple stakeholders and providing a metered service at a specified level of quality [2]. One of the major benefits of using cloud computing compared to using an internal infrastructure is the ability of the cloud to provide its customers with elastic resources that can be provisioned on demand within seconds or minutes. These resources can be used to handle flash crowds. A flash crowd, also known as a slashdot effect, is a surge in traffic to a particular service that causes the service to be virtually unreachable [3]. Flash crowds are very common in today's networked world. Figure I shows the traces of the FIFA 1998 world cup website. Flash crowds occur frequently before and after matches. In this work, we try to automate and optimize the management of flash crowds in the cloud by developing an autonomous elasticity controller.

Autonomous elastic cloud infrastructures provision resources according to the *current* actual demand on the infrastructure while enforcing service level agreements (SLAs). Elasticity is the ability of the cloud to rapidly vary the allocated resource capacity to a service according to the current load in order to meet the quality of service (QoS) requirements specified in the SLA agreements. Horizontal elasticity is the ability of the cloud to rapidly increase or decrease the number of virtual machines (VMs) allocated to a service according to the current load. Vertical elasticity is the ability of the cloud to change the hardware configuration of VM(s) already running to increase or decrease the total amount of resources allocated to a service running in the cloud.

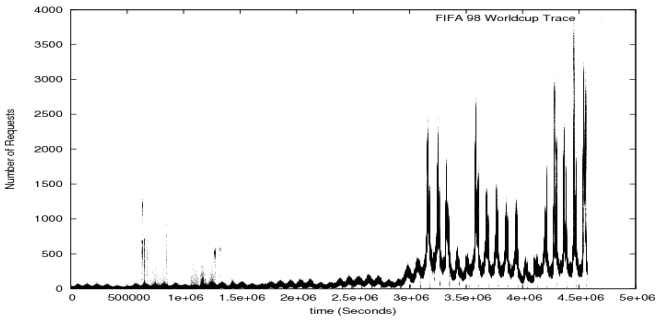


Fig. 1. Flash crowds illustrating the rapid change in demand for the FIFA world cup website.

Building elastic cloud infrastructures that scale up and down with the actual demand of the service is a problem far from being solved [2]. Scale up should be fast enough in order to prevent breaking any SLAs while it should be as close as possible to the actual required load. Scale down should not be premature, i.e., scale down should occur when it is anticipated that the service does not need these resources in the near future. If scale down is done prematurely, resources are allocated and deallocated in a way that causes oscillations in the system. These resource oscillations introduce problems to load balancers and add some extra costs due to the frequent release and allocation of resources [4]. In this paper we develop two adaptive horizontal elasticity controllers that control scale up and scale down decisions and prevent resource oscillations.

This paper is organized as follows; in Section II, we describe the design of our controllers. In Section III we describe our simulation framework, our experiments and discuss our results. In Section IV we describe some approaches to building elasticity controllers in the literature. We conclude in Section V.

II. Building an Elastic Cloud Controller

In designing our controllers, we view the cloud as a control system. Control systems are either closed loop or open loop systems [5]. In an open loop control system, the control action does not depend on the system output making open loop control generally more suited for simple applications where no

978-1-4673-0269-2/12/$31.00

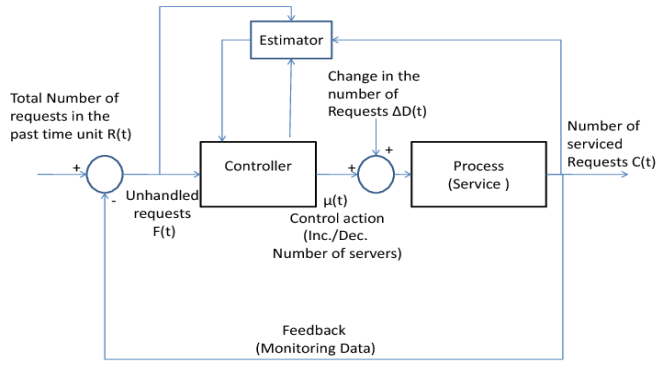


Fig. 2. Adaptive Proactive Controller Model.

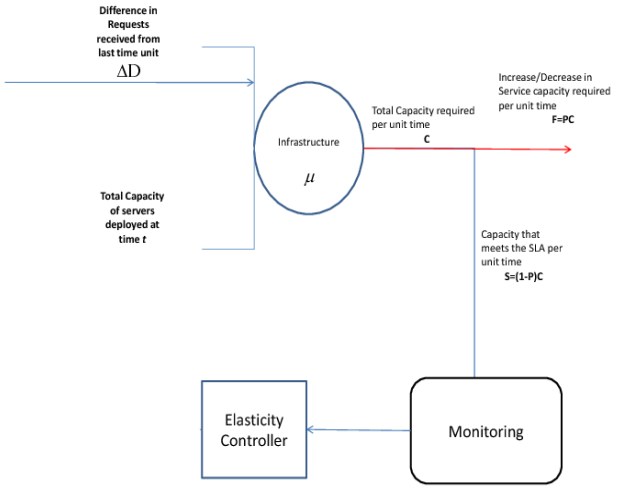


Fig. 3. Queuing Model for a service deployed in the cloud.

feedback is required and no system monitoring is performed. Contrarily, a closed loop control system is more suited for sophisticated application as the control action depends on the system output and on monitoring some system parameters. The general closed-loop control problem can be stated as follows: The controller output $\mu(t)$ tries to force the system output $C(t)$ to be equal to the reference input $R(t)$ at any time t irrespective of the disturbance ΔD. This general statement defines the main targets of any closed loop control system irrespective of the controller design.

In this work, we model a service deployed in the cloud as a closed loop control system. Thus, the horizontal elasticity problem can be stated as follows: The elasticity controller output $\mu(t)$ should add or remove VMs to ensure that the number of service requests $C(t)$ is equal to the total number of requests received $R(t) + \Delta D(t)$ at any time unit t with an error tolerance specified in the SLA irrespective of the change in the demand ΔD while maintaining the number of VMs to a minimum. The model is simplified by assuming that servers start up and shut down instantaneously.

We design and build two adaptive proactive controllers to control the QoS of a service as shown in Figure 2. We add an estimator to adapt the output of the controller with any change in the system load and the system model.

A. Modeling the state of the service

Figure 3 shows a queuing model representing the cloud infrastructure. The infrastructure is modeled as a $G/G/N$ stable queue in which the number of servers N required is variable [6]. In the model, the total number of requests per second R_{total} is divided into two inputs to the infrastructure, the first input $R(t)$ represents the total amount of requests the infrastructure is capable of serving during time unit t. The second input, ΔD represents the change in the number of requests from the past time unit. Since the system is stable, the output of the queue is the total service capacity required per unit time and is equal to R_{total}. P represents the increase or decrease in the number of requests to the current service capacity $R(t)$.

The goal of a cloud provider is to provide all customers with enough resources to meet the QoS requirements specified in the SLA agreements while reducing over provisioning to a minimum. The cloud provider monitors a set of parameters stated in the SLA agreements. These parameters represent the controlled variables for the elasticity controller. Our controllers are parameter independent and can be configured to use any performance metric as the controlled parameter. For the evaluation of our controllers, we choose the number of concurrent requests received for the past time unit to be the monitored parameter because this metric shows both the amounts of over provisioned and under provisioned resources which is an indicator to the costs incurred due to the elasticity engine. Most of the previous work on elasticity considers response time to be the controlled parameter. Response time is software and hardware dependent and is not well suited for comparing the quality of different elasticity approaches [7].

B. Estimating future usage

From Figure 3, the total future service capacity required per unit time, $C(t+1)$, is $C(t+1) = \Delta D(t) + R(t)$, where $R(t)$ is the current service capacity and $\Delta D(t)$ is the change in the current service capacity required in order to meet the SLA agreement while maintaining the number of VMs to minimum. A successful proactive elasticity engine is able to estimate the change in future demand $\Delta D(t)$ and add or remove VMs based on this proactive estimation. $\Delta D(t)$ can be estimated by

$$\Delta D(t) = P(t)C(t) \quad (1)$$

where $P(t)$ is the gain parameter of the controller. $P(t)$ is positive if there is an increase in the number of requests, negative if there is a decrease in the number of requests, or zero if the number of requests is equal to the current service capacity.

We define $\widehat{C}$ to be the infrastructure's average periodical service rate over the past T_d time units. $\widehat{C}$ is calculated

for the whole infrastructure and not for a single VM. Thus, $\widehat{C} = \frac{\sum_i^{T_d} n_i t_i}{T_d}$, where T_d is a system parameter specifying the period used for estimating the average periodical service rate and t_i is the time for which the number of requests received per unit time for the whole infrastructure stay constant at n_i requests per unit time before the demand changes. Thus, $\sum_i^{T_d} t_i = T_d$. We also define $\overline{n}$, the average service rate over time as $\overline{n} = \frac{\sum_t n(t)}{T}$.

From equation 1 and since the system is stable ,

$$F = \widehat{C}\ P, \tag{2}$$

where F, the estimated increase or decrease of the load, is calculated using the gain parameter of the controller P every time unit. The gain parameter represents the estimated rate of adding or removing VMs. We design two different controllers with two different gain parameters.

For the first controller P_{C1}, the gain parameter P_1 is chosen to be the periodical rate of change of the system load,

$$P_1 = \frac{\Delta D_{T_d}}{T_D}. \tag{3}$$

As the workload is a non-linear function in time, the periodical rate of change of the load is the derivative of the workload function during a certain period of time. Thus, the gain parameter represents the load function changes over time.

For the second controller P_{C2}, the gain parameter P_2 is the ratio between the change in the load and the average system service rate over time,

$$P_2 = \frac{\Delta D_{T_d}}{\overline{n}}. \tag{4}$$

This value represents the load change with respect to the average capacity. By substituting this value for P in Equation 1, the predicted load change is the ratio between the current service rate and the average service rate multiplied by the change in the demand over the past estimation period.

C. Determining suitable estimation intervals

The interval between two estimations, T_d, represents the period for which the estimation is valid, is a crucial parameter affecting the controller performance. It is used for calculating $\widehat{C}$ for both controllers and for P_1 in the case of the first controller. T_d controls the controllers' reactivity. If T_d is set to one time unit, the estimations for the system parameters are done every time unit and considers only the system load during past time unit. At the other extreme, if T_d is set to ∞, the controller does not perform any predictions at all. As the workload observed in data centers is dynamic [8], setting an adaptive value for T_d that changes with the load dynamics is one of our goals.

We define K to be the tolerance level of a service i.e. the number of requests the service does not serve on time before making a new estimation, in other words,

$$T_d = \frac{K}{\widehat{C}}. \tag{5}$$

TABLE I
OVERVIEW OF THE NINE DIFFERENT WAYS TO BUILD A HYBRID CONTROLLER.

Engine Name	Scale up mechanism	Scale down mechanism
UR-DR	Reactive	Reactive
UR-DP	Reactive	Proactive
UR-DRP	Reactive	Reactive and Proactive
URP-DRP	Reactive and Proactive	Reactive and Proactive
URP-DR	Reactive and Proactive	Reactive
URP-DP	Reactive and Proactive	Proactive
UP-DP	Proactive	Proactive
UP-DR	Proactive	Reactive
UP-DRP	Proactive	Reactive and Proactive

K is defined in the SLA agreement with the service owner. If K is specified to be zero, T_d should always be kept lower than the maximum response time to enforce that no requests are served slower by the system.

D. An elasticity engine for scale-up and scale-down decisions

The main goal of any elasticity controller is to enforce the SLAs specified in the SLA agreement. For today's dynamical network loads [3], it is very hard to anticipate when a flash crowd is about to start. If the controller is not able to estimate the flash crowd on time, many SLAs are likely to be broken before the system can adapt to the increased load.

Previous work on elasticity considers building hybrid controllers that combines reactive and proactive controllers [9] and [10]. We extend on this previous work and consider all possible ways of combining reactive and proactive controllers for scaling of resources in order to meet the SLAs. We define an elasticity engine to be an elasticity controller that considers both scale-up and scale-down of resources. There are nine approaches in total to build an elasticity engine using a reactive and a proactive controller. These approaches are listed in Table I. Some of these combinations are intuitively not good, but for the sake of completeness we evaluate the results of all of these approaches. In order to facilitate our discussion, we use the following naming convention to name an elasticity engine; an elasticity engine consists of two controllers, a scale up (U) and a scale down (D) controller. A controller can be either reactive (R) or proactive (P). P_{C1} and P_{C2} are a special case from proactive controllers e.g. URP-DRP elasticity engine has a reactive and proactive controller for scale up and scale down while a UR-DP_{C1} is an engine having a reactive scale up controller and P_{C1} for scale down.

III. EXPERIMENTAL EVALUATION

In order to validate the controllers, we designed and built a discrete event simulator that models a service deployed in the cloud. The simulator is built using Python. We used the complete traces from the FIFA 1998 world cup as input to our model [11]. The workload contains 1.3 billion Web requests recorded in the period between April 30, 1998 and July 26, 1998. We have calculated the aggregate number of requests per second from these traces. They are by far the most used traces in the literature. As these traces are quite old, we multiply the

number of requests received per unit time by a constant in order to scale up these traces to the orders of magnitude of today's workloads. Although there are newer traces available such as the Wikipedia trace [12], but they do not have the number of peaks seen in the FIFA traces. We assume perfect load balancing and quantify the performance of the elasticity engines only.

A. Nine Approaches to build an elasticity engine

In this experiment we evaluate the nine approaches to a hybrid controller and quantify their relative performance using P_{C1} and P_{C2}. We use the aggregate number of requests per unit time from the world cup traces multiplied by a constant equal to 50 as input to our simulator. This is almost the same factor by which the number of Internet users increased since 1997 [13]. To map the number of service requests to the number of servers, we assume that each server can serve up to 500 requests per unit time. This number is an average between the number of requests that can be handled by a Nehalem Server running the MediaWiki application [14] and a Compaq ProLiant DL580 server running a database application [15]. We assume SLAs that specify the maximum number of requests not handled per unit time to be fewer than 5% of the maximum capacity of one server.

The reactive controller is reacting to the current load while the proactive controller is basing its decision on the history of the load. Whenever a reactive controller is coupled with a proactive controller and the two controllers give contradicting decisions, the decision of the reactive controller is chosen. For the UR-DR controller, scale down is only done if the number of unused servers is greater than two servers in order to reduce oscillations.

To compare all the different approaches, we monitor and sum the number of servers the controllers fail to provision on time to handle the increase in the workload, S^-. This number can be viewed as the number of extra servers to be added to avoid breaking all SLAs, or as the quality of estimation. $\overline{S^-}$ is the average number of requests the controller fails to provision per unit time. Similarly, we monitor the number of extra servers deployed by the infrastructure at any unit time. The summation of this number indicates the provisioned unused server capacity, S^+. $\overline{S^+}$ is the averaged value over time. These two aggregate metrics are used to compare the different approaches.

Table II shows the aggregate results when P_{C1} and P_{C2} are used for the proactive parts of the hybrid engine. The two right-most columns in the table show the values of S^- and S^+ as percentages of the total number of servers required by the workload respectively. We compare the different hybridization approaches with a UR-DR elasticity engine [16].

The results shown in the two tables indicate that using an UR-DP_{C2} engine reduces S^- by a factor of 9.1 compared to UR-DR elasticity engine, thus reducing SLA violations by the same ratio. This comes at the cost of using 14.33% extra servers compared to 1.4% in the case of a UR-DR engine. Similar results are obtained using a URP_{C2}-DP_{C2} engine. These results are obtained because the proactive scale down controller does not directly release resources when the load decreases instantaneously but rather makes sure that this decrease is not instantaneous. Using a reactive controller for scale down on the other hand reacts to any load drop by releasing resources. It is also observed that the second best results are obtained using an UR-DP_{C1} elasticity engine. This setup reduces S^- by a factor of 4, from 1.63% to 0.41% compared to a UR-DR engine at the expense of increasing the number of extra servers used from 1.4% to 9.44%.

A careful look at the table shows that elasticity engines with reactive components for both scale up and scale down show similar results even when a proactive component is added. We attribute this to the premature release of resources due to the reactivity component used for the scale down controller. The premature release of resources causes the controller output to oscillate with the workload. The worst performance is seen when a proactive controller is used for scale up with a reactivity component in the scale down controller. This engine is not able to react to sudden workload surges. In addition it releases resources prematurely.

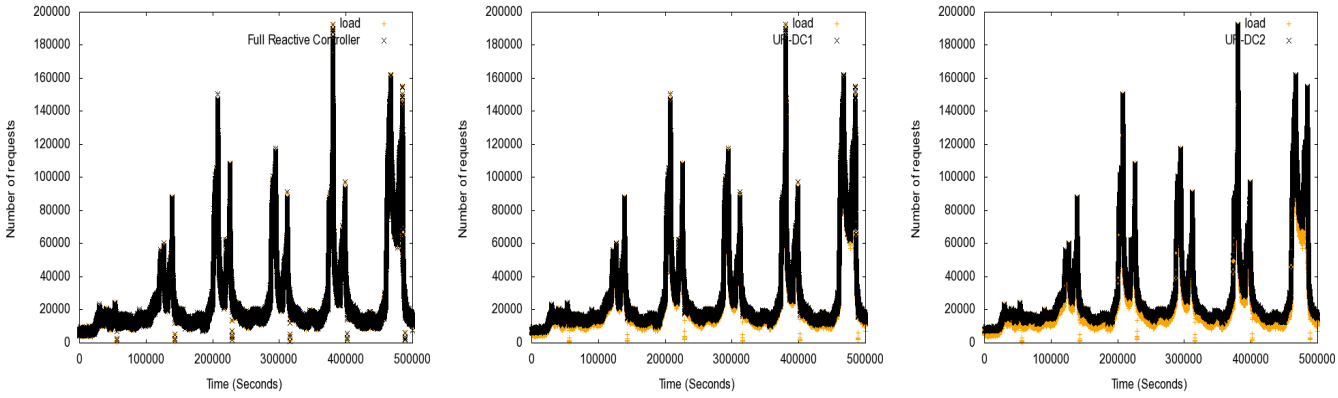
Figures 4(a), 4(b) and 4(c) shows the performance of a UR-DR, UR-DP_{C1} and a UR-DP_{C2} elasticity engines over part of the trace from 06:14:32, the 21^{st} of June,1998 to 01:07:51 27^{th} of June,1998. Figures 4(d), 4(e) and 4(f) shows an in depth view of the period between 15:50:00 the 23^{rd} of June, 1998 till 16:07:00 on the same day (between time unit 208349 and 209349 on the left hand side figures).

The UR-DR elasticity engine releases resources prematurely as seen in Figure 4(d). These resources are then reallocated when there is an increase in demand causing resource allocation and deallocation to oscillate. The engine is always following the demand but is never ahead. On the other hand, figures 4(e) and 4(f) show different behavior where the elasticity engine tries not to deallocate resources prematurely in order to prevent oscillations and to be ahead of the demand. It is clear in Figure 4(f) that the elasticity engine estimates the future load dynamics and forms an *envelope* over the load. An envelope is defined as the smooth curve that takes the general shape of the load's amplitude and passes through its peaks [17]. This delay in the deallocation comes at the cost of using more resources. These extra resources improve the performance of the service considerably as it will be always ahead of the load. We argue that this additional cost is well justified considering the gain in service performance.

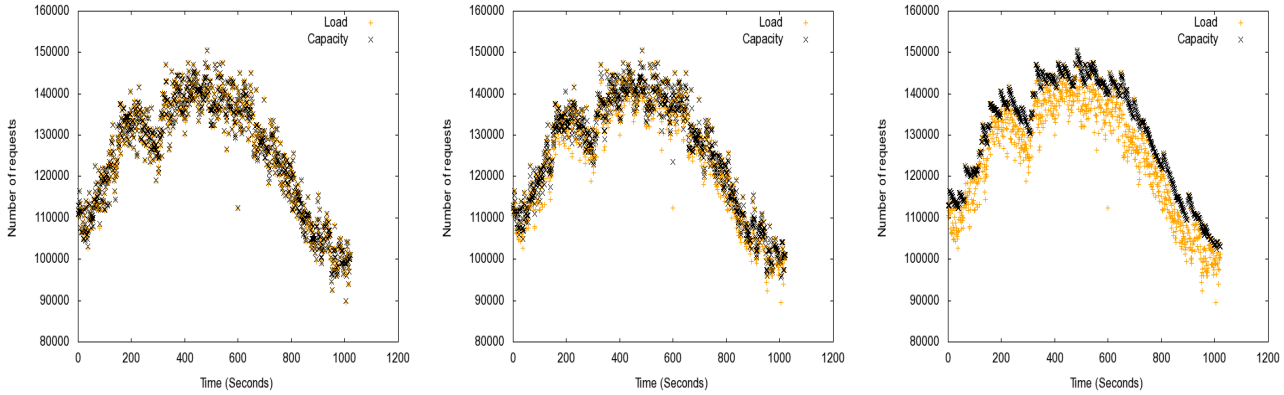
1) Three classes of SLAs: An infrastructure provider can have multiple controller types for different customers and different SLA agreements. The results shown in table II suggest having three classes of customers namely, gold, silver and bronze. A gold customer pays more in order to get the best service at the cost of some extra over-provisioning and uses a UR-DP_{C2} elasticity engine. A silver customer uses the UR-DP_{C1} elasticity engine to get good availability while a bronze customer uses the UR-DR and gets a reduced, but acceptable, QoS but with very little over-provisioning. These three different elasticity engines with different degrees

TABLE II
S^- AND S^+ FOR P_{C1} AND P_{C2}

Name	S^-	$\overline{S^-}$	S^+	$\overline{S^+}$	$S^-\%$	$S^+\%$
UR-DR	-1407732	-0.3	120641	0.026	-1.63%	1.4%
P_{C1} results						
UR-DP$_{C1}$	-354814	-0.077	8159220	1.78	-0.41%	9.44%
UR-DRP$_{C1}$	-1412289	-0.3	1202806	0.26	-1.63%	1.4%
URP$_{C1}$-DRP$_{C1}$	-1411678	-0.3	1203170	0.26	-1.63%	1.4%
URP$_{C1}$-DR	-1407036	-0.3	1206391	0.26	-1.62%	1.4%
URP$_{C1}$-DP$_{C1}$	-354127	-0.077	8160627	1.78	-0.41%	9.4%
UP$_{C1}$-DP$_{C1}$	-4147953	-0.9	1827431	0.399	-4.8%	2.1%
UP$_{C1}$-DR	-8474040	-1.85	408447	0.399	-9.8%	2.1%
UP$_{C1}$-DRP$_{C1}$	-11408704	-2.49	190427.0	0.041	-10%	0.27%
P_{C2} results						
UR-DP$_{C2}$	-159029	-0.0347	12386346.0	2.7	-0.18%	14.33%
UR-DRP$_{C2}$	-1418949.0	-0.31	1176239.0	0.257	-1.64%	1.36%
URP$_{C2}$-DRP$_{C2}$	-1419269.0	-0.31	1175393.0	0.257	-1.64%	1.35%
URP$_{C2}$-DR	-1407732.0	-0.31	1206407.0	0.263	-1.63%	1.4%
URP$_{C2}$-DP$_{C2}$	-159029	-0.0347	12386346.0	2.707	-0.18%	14.33%
UP$_{C2}$-DP$_{C2}$	-4350841.0	-0.951	2216866.0	0.485	-5.03%	2.6%
UP$_{C2}$-DR	-11245521	-2.458	396697	0.0867	-13%	0.46%
UP$_{C2}$-DRP$_{C2}$	-11408704	2.49	190427	0.0416	-13.2%	0.22%



(a) UR-DR performance in a period of 6 days. (b) UR-DP$_{C1}$ performance in a period of 6 days. (c) UR-DP$_{C2}$ performance in a period of 6 days.



(d) UR-DR: Zooming on a period of 17 minutes. (e) UR-DP$_{C1}$: Zooming on a period of 17 minutes. (f) UR-DP$_{C2}$: Zooming on a period of 17 minutes.

Fig. 4. Performance of UR-DR, UR-DP$_{C1}$ and, UR-DP$_{C2}$ elasticity engines with time: The Figures show how the different engines detect future load. It can be observed that the UR-DR engine causes the capacity to oscillate with the load while UR-DP$_{C1}$ and UR-DP$_{C2}$ predict the envelope of the workload.

TABLE III
COMPARISON BETWEEN THE UR-DREGRESSION, UR-DP$_{C1}$, UR-DP$_{C2}$, AND UR-DR ELASTICITY ENGINES

Name	S^-	S^+	S^-	S^+
UR-DRegression	-74791.7	1568047.8	-2.24%	47%
UR-DP$_{C1}$	-50307.2	1076236.3	-1.51%	32.24%
UR-DP$_{C2}$	-35818.6	1326841.7	-1.07%	39.75%
UR-DR	-99801.8	653082.9	-2.99%	19.57%

of over provisioning and qualities of estimation give cloud providers convenient tools to handle customers of different importance classes and thus increase their profit and decrease their penalties. Current cloud providers usually have a general SLA agreement for all their customers. RackSpace [18] for example guarantees 100% availability with a penalty equal to 5% of the fees for each 30 minutes of network or data center downtime for the cloud servers. It guarantees 99.9% availability for the cloud files. The network is considered not available in case of [18]: (i) The Rackspace Cloud network is down, or (ii) the Cloud Files service returns a server error response to a valid user request during two or more consecutive 90 second intervals, or (iii) the Content Delivery Network fails to deliver an average download time for a 1-byte reference document of 0.3 seconds or less, as measured by The Rackspace Cloud's third party measuring service. For an SLA similar to the RackSpace SLA or Amazon S3 [19], using one of our controllers significantly reduces penalties paid due to server errors, allowing the provider to increase profit.

B. Comparison with regression based controllers

In this experiment we compare our controllers with the controller designed by Iqbala et al. [10] who design a hybrid elasticity engine with a reactive controller for scale-up decisions and a predictive controller for scale-down decisions. When the capacity is less than the load, a scale up decision is taken and new VMs are added to the service. For scale down, their predictive component uses second order regression. The regression model is recomputed for the full history every time a new measurement data is available. If the current load is less than the provisioned capacity for k time units, a scale down decision is taken using the regression model. If the predicted number of servers is greater than the current number of servers, the result is ignored. Following our naming convention, we denote their engine UR-DRegression. As regression is recomputed every time a new measurement data is available on the full history, simulation using the whole world cup traces would be time consuming. Instead, in this experiment we used part of the trace from 09:47:41 on the 13^{th} of May, 1998 to 17:02:49 on the 25^{th} of May, 1998. We multiply the number of concurrent requests by 10 and assume that the servers can handle up to 100 requests. We assume that the SLA requires that a maximum of 5% of the capacity of a single server is not serviced per unit time.

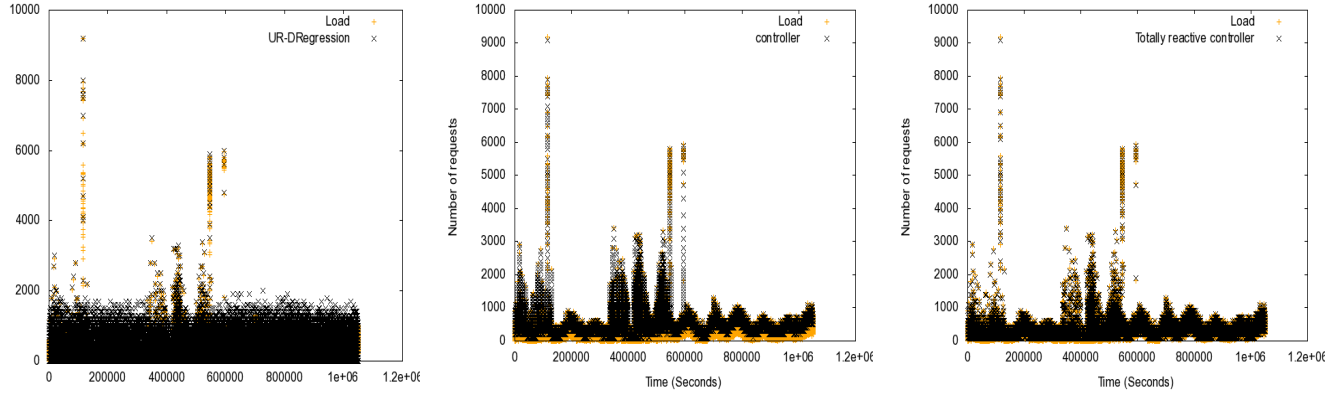
Table III shows the aggregated results for four elasticity engines; UR-DRegression, UR-DP$_{C1}$, UR-DP$_{C2}$ and UR-DR. Although all the proactive approaches reduce the value of S^- compared to a UR-DR engine, P$_{C2}$ still shows superior results. The number of unused server that get provisioned by the regression controller S^+ is 50% more than for P$_{C1}$ and 15% more than P$_{C2}$ although both P$_{C1}$ and P$_{C2}$ reduces S^- more. The $UR-DR$ controller has a higher SLA violation rate (3%) while maintaining a much lower over-provisioning rate (19.57%). As we evaluate the performance of the controller on a different part of the workload and we multiply the workload by a different factor, the percentages of the capacity the controller fail to provision on time and the unused provisioned capacity changed from the previous experiment.

Figures 5(a), 5(b) and 5(c) show the load compared to the controller outputs for the UR-DR, UR-DP$_{C2}$, and UR-DRegression approaches. The amount of unused capacity using a regression based controller is much higher than the unused capacity for the other controllers. The controller output for the UR-DRegression engine completely over-estimates the load causing prediction oscillations between the crests and the troughs. One of the key advantages of P$_{C1}$ and P$_{C2}$ is that they depend on simple calculations. They are both scalable with time compared to the regression controller. The highest observed estimation time for the UR-DRegression is 6.5184 seconds with an average of 0.97695 seconds compared to 0.000512 seconds with an average of 5.797×10^{-6} in case of P$_{C1}$ and P$_{C2}$.

C. Performance impact of the workload size

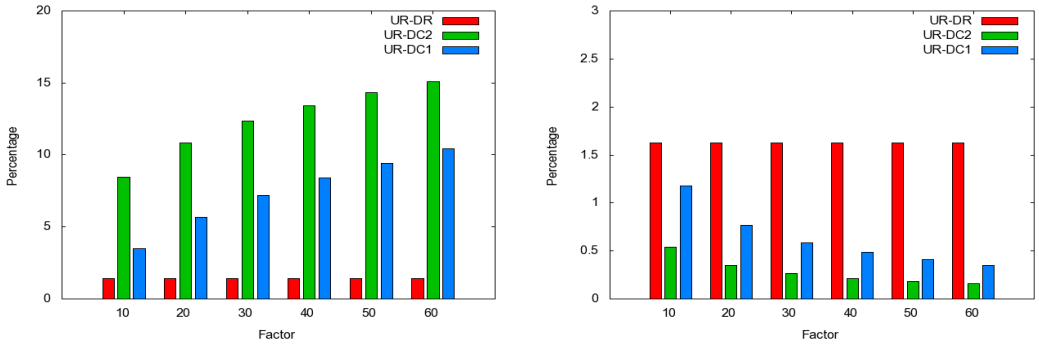
In this experiment we investigate the effect of changing the load and server power on the performance of our proposed elasticity engines. We constructed six new traces using the world cup workload traces by multiplying the number of requests per second in the original trace by a factor of 10, 20, 30, 40, 50, and 60. We ran experiments with the new workloads using the UR-DR, UR-DP$_{C1}$ and UR-DP$_{C2}$ elasticity engines. For each simulation run, we assume that the number of requests that can be handled by any server is 10 times the factor by which we multiplied the traces, e.g., for an experiment run using a workload trace where the number of requests is multiplied by 20, we assume that the server capacity is up to 200 requests per second. We also assume that for each experiment the SLA specifies the maximum unhandled number of requests to be 5% of the maximum capacity of a single server.

Figure 6(a) shows the percentage of servers the engines failed to provision on time to handle the increase in demand for each workload size (S^-) while Figure 6(b) shows the percentages of extra servers provisioned for each workload size. It is clear that the UR-DR engine exhibits the same performance with changing workloads. For the UR-DP$_{C1}$ and the UR-DP$_{C2}$ engines on the other hand, the performance depends on the workload and the server capacity. As the factor by which we multiply the workload increases, the percentage of servers the two engines failed to provision on time decreases. Inversely, the percentage of extra servers provisioned increases. These results indicate that the quality of estimation changes with any change in the workload. We



(a) UR-DRegression elasticity engine. (b) UR-DP$_{C2}$ elasticity engine. (c) UR-DR elasticity engine

Fig. 5. Performance Comparison of UR-DR, UR-DP$_{C2}$ and UR-DRegression elasticity engines. The UR-DRegression controller over-provisions many servers to cope with the changing workload dynamics.



(a) The effect of changing load size on the percentage of S^- to the total number of servers. (b) The effect of changing load size on the percentage of S^+ to the total number of servers.

Fig. 6. The effect of changing the workload size and the server capacity on the UR-DR, UR-DP$_{C1}$ and UR-DP$_{C2}$ elasticity engines.

attribute the improvement in the quality of estimation when the load increases using the UR-DP$_{C1}$ and UR-DP$_{C2}$ engines to the ability of both estimators to predict the envelope of the workload, thus decreasing the number of prematurely deallocated resources. Although the number of requests increases in the different workloads, the number of times the controllers deallocate resources prematurely also increases, but at a slower rate than the load. We have performed similar experiments with the Wikipedia traces [12] and obtained similar results [20]. Due to lack of space we omit those results.

D. Discussion

Although our proactive controllers P$_{C1}$ and P$_{C2}$ are designed using the change in the load as the controller parameter, they can be generalized to be used with any hardware parameter such as CPU load, memory consumption, network load and disk load or any server level parameter such as response time. When P$_{C1}$ or P$_{C2}$ controller is used with hardware measured parameter, e.g., CPU load, $C(t)$ becomes the total CPU capacity needed by the system to handle the CPU load per unit time. ΔD is the change in the load. $\widehat{C}$ becomes the average periodical measurement of the CPU load and $\overline{n}$ the average measurement of the CPU load over time. The definition of the two controllers remains the same.

Both the UR-DP$_{C1}$ and UR-DP$_{C2}$ engines can be integrated in the model proposed by Lim et al. [21] to control a storage cloud. In storage clouds, adding resources does not have an instantaneous effect on the performance since data must be copied to the new allocated resources before the effect of the control action takes place. For such a scenario, P$_{C1}$ and P$_{C2}$ are very well suited since they predict the envelope of the demand. The engines can also replace the elasticity controllers designed by Urgaonkar et al. [9] or Iqbala et al. [10] for a multi-tier service deployed in the cloud.

IV. Related Work

The problem of dynamic provisioning of resources in computing clusters has been studied for the past decade. Cloud elasticity can be viewed as a generalization of that problem. Our model is similar to the model introduced in [22]. In that work, the authors tried to estimate the availability of a machine in a distributed storage system in order to replicate its data.

Toffetti et al. [1] use Kriging surrogate models to approximate the performance profile of virtualized, multi-tier Web

applications. The performance profile is specific to an application. The Kriging surrogate model needs offline training. A change in the workload dynamics results in a change in the service model. Adaptivity of the service model of an application is vital to cope with the changing load dynamics in todays Internet [3].

Lim et al. [21] design an integral elasticity controller with proportional thresholding. They use a dynamic target range for the set point. The integral gain is calculated offline making this technique suitable for a system where no sudden changes to the system dynamics occur as the robustness of an integral controller is affected by changing the system dynamics [23].

Urgaonkar et al. [9] propose a hybrid control mechanism that incorporates both a proactive controller and a reactive controller. The proactive controller maintains the history of the session arrival rate seen. Provisioning is done before each hour based on the worst load seen in the past. No short term predictions can be done. The reactive controller acts on short time scales to increase the resources allocated to a service in case the predicted value is less than the actual load that arrived. No scale down mechanism is available.

In [24], the resource-provisioning problem is posed as one of sequential optimization under uncertainty and solved using limited look-ahead control. Although the solution shows very good theoretical results, it exhibits an exponential increase in computation time as the number of servers and VMs increase. It takes 30 minutes to compute the required elasticity decision for a system with 60 VMs and 15 physical servers. Similarly, Nilabja et al. use limited lookahead control along with model predictive control for automating elasticity decisions. Improving the scalability of their approach is left as a future direction to extend their work.

Chacin and Navaro [25] propose an elastic utility driven overlay network that dynamically allocate instances to a service using an overlay network. The instances of each services construct an overlay while the non-allocated instances construct another overlay. The overlays change the number of instances allocated to a service based on a combination of an application provided utility function to express the service's QoS, with an epidemic protocol for state information dissemination and simple local decisions on each instance.

There are also some studies discussing vertical elasticity [26]. Jung et al. [4] design a middleware for generating cost sensitive adaptation actions such as elasticity and migration actions. Vertical elasticity is enforced using adaptation action in fixed steps predefined in the system. To allocate more VMs to an application a migration action is issued from a pool of dormant VMs to the pool of the VMs of the target host followed by an increase adaptation action that allocates resources on the migrated VM for the target application. These decisions are made using a combination of predictive models and graph search techniques reducing scalability. The authors leave the scalability of their approach for future work.

V. Conclusions and Future Work

In this paper, we consider the problem of autonomic dynamic provisioning for a cloud infrastructure. We introduce two adaptive hybrid controllers P_{C1} and P_{C2}, that use both reactive and proactive control to dynamically change the number of VMs allocated to a service running in the cloud based on the current and the predicted future demand. Our controllers detect the workload envelope and hence do not deallocate resources prematurely. We discuss the different ways of designing a hybrid elasticity controller that incorporates both reactive and proactive components. Our simulation results show that using a reactive controller for scale up and one of our proactive controllers for scale down improves the SLA violations rate two to ten times compared to a totally reactive elasticity engine. We compare our controllers to a regression based elasticity controller using a different workload and demonstrate that our engines over-allocate between 32% and 15% less resources compared to a regression based engine. The regression based elasticity engine SLA violation rate is 1.48 to 2.1 times the SLA violation rate for our engines. We also investigate the effect of the workload size on the performance of our controllers. For increasing loads, our simulation results show a sublinear increase in the number of SLAs violated using our controllers compared to a linear increase in the number of SLAs violations for a reactive controller. In the future, we plan to integrate vertical elasticity control in our elasticity engine and modify the controllers to consider the delay required for VM start up and shut down.

VI. Acknowledgments

This work is supported by the OPTIMIS project (http://www.optimis-project.eu/) and the Swedish government's strategic research project eSSENCE. It has been partly funded by the European Commissions IST activity of the 7th Framework Program under contract number 257115 . This research was conducted using the resources of High Performance Computing Center North (http://www.hpc2n.umu.se/).

References

[1] P. Mell and T. Grance, "The nist definition of cloud computing," *National Institute of Standards and Technology*, vol. 53, no. 6, 2009.

[2] D. Kossmann and T. Kraska, "Data management in the cloud: Promises, state-of-the-art, and open questions," *Datenbank-Spektrum*, vol. 10, pp. 121–129, 2010, 10.1007/s13222-010-0033-3. [Online]. Available: http://dx.doi.org/10.1007/s13222-010-0033-3

[3] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long, "Managing flash crowds on the Internet," 2003.

[4] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "A cost-sensitive adaptation engine for server consolidation of multitier applications," in *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '09. Springer-Verlag New York, Inc., 2009, pp. 9:1–9:20.

[5] K. Ogata, *Modern control engineering*. Prentice Hall, 2009.

[6] H. Li and T. Yang, "Queues with a variable number of servers," *European Journal of Operational Research*, vol. 124, no. 3, pp. 615 – 628, 2000.

[7] P. Bodık, "Automating datacenter operations using machine learning," Ph.D. dissertation, University of California, 2010.

[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, ser. SIGCOMM '09. New York, NY, USA: ACM, 2009, pp. 51–62.
[9] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier Internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, p. 1, 2008.
[10] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871 – 879, 2011.
[11] M. Arlitt and T. Jin. (1998, August) "1998 world cup web site access logs". [Online]. Available: http://www.acm.org/sigcomm/ITA/
[12] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009, http://www.globule.org/publi/WWADH_comnet2009.html.
[13] (2011, July) Internet growth statistics. [Online]. Available: http://www.internetworldstats.com/emarketing.htm
[14] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz, "Napsac: design and implementation of a power-proportional web cluster," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 102–108, 2011.
[15] P. Dhawan. (2001, October) Performance comparison: Exposing existing code as a web service. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms978401.aspx
[16] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma, "Towards autonomic workload provisioning for enterprise grids and clouds," in *Grid Computing, 2009 10th IEEE/ACM International Conference on*. IEEE, 2009, pp. 50–57.
[17] W. M. Hartmann, *Signals, sound, and sensation*. Amer Inst of Physics, 1997.
[18] (2009, June) Rackspace hosting: Service level agreement. [Online]. Available: http://www.rackspace.com/cloud/legal/sla/
[19] (2007, October) Amazon S3 service level agreement. [Online]. Available: http://aws.amazon.com/s3-sla/
[20] A. J. Ferrer, F. Hernndez, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forg, T. Sharif, and C. Sheridan, "Optimis: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 66 – 77, 2012.
[21] H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *Proceeding of the 7th international conference on Autonomic computing*, ser. ICAC '10. New York, NY, USA: ACM, 2010, pp. 1–10.
[22] A. Duminuco, E. Biersack, and T. En-Najjary, "Proactive replication in distributed storage systems using machine availability estimation," in *Proceedings of the 2007 ACM CoNEXT conference*, ser. CoNEXT '07. New York, NY, USA: ACM, 2007, pp. 27:1–27:12.
[23] M. Morari, "Robust stability of systems with integral control," *Automatic Control, IEEE Transactions on*, vol. 30, no. 6, pp. 574–577, 1985.
[24] D. Kusic, J. Kephart, J. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster Computing*, vol. 12, no. 1, pp. 1–15, 2009.
[25] P. Chacin and L. Navarro, "Utility driven elastic services," in *Distributed Applications and Interoperable Systems*. Springer, 2011, pp. 122–135.
[26] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *Proceedings of the 6th international conference on Autonomic computing*, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 117–126.