Self-* through self-learning: overload control for distributed web systems

Novella Bartolini, Giancarlo Bongiovanni, Simone Silvestri

Department of Computer Science,
University of Rome "Sapienza", Italy
E-mail: {bartolini,bongiovanni,simone.silvestri}@di.uniroma1.it

Abstract

Overload control is a challenging problem for web-based applications, which are often prone to unexpected surges of traffic. Existing solutions are still far from guaranteeing the necessary responsiveness under rapidly changing operative conditions. We contribute an original Self-* Overload Control (SOC) algorithm that self-configures a dynamic constraint on the rate of incoming new sessions in order to guarantee the fulfillment of the quality requirements specified in a Service Level Agreement (SLA). Our algorithm is based on a measurement activity that makes the system capable of self-learning and self-configuring even in the case of rapidly changing traffic scenarios, dynamic resource provisioning or server faults. Unlike other approaches, our proposal does not require any prior information about the incoming traffic, or any manual configuration of key parameters.

We ran extensive simulations under a wide range of operating conditions. The experiments show how the proposed system self-protects from overload, meeting SLA requirements even under intense workload variations. Moreover, it rapidly adapts to unexpected changes in available capacity, as in the case of faults or voluntary architectural adjustments. Performance comparisons with other previously proposed approaches show that our algorithm has better performance and more stable behavior.

Key words:
admission control, flash crowd, scalability, self-learning, self-configuration

1 Introduction

Quality of Service (QoS) management for web-based applications is typically considered a problem of system sizing: sufficient resources are to be provisioned to meet quality of service requirements under a wide range of operating conditions. While this approach is beneficial in making site performance satisfactory in most common working situations, it still leaves the site unable to face sudden and unexpected surges in traffic. In these situations, in fact, it is impossible to predict the intensity of the overload. The architecture in use, although over-dimensioned, may not be sufficient to meet the desired QoS under every possible traffic scenario. For this reason, unexpected increases in requests and, above all, flash crowds are considered the bane of every internet based application.

Due to the ineffectiveness of static resource over-provisioning, several alternative approaches have been proposed for overload management in web systems, such as dynamic provisioning, dynamic content adaptation, performance degradation and admission control. Most of the previously proposed work on this topic relies on laborious parameter-tuning and manual configuration that preclude fast adaptation of the control policies.

This study is motivated by the need to formulate a fast reactive and autonomous approach to admission control. We contribute an original Self-* Overload Control algorithm (SOC) that, as the name suggests, has some fundamental autonomic properties, namely self-configuration, self-adaptation and self-protection.

The proposed algorithm is designed to be adopted by web cluster dispatching points (DP) and does not require any modification of the client and/or server software. DPs intercept requests and make decisions to block or accept incoming new sessions to meet the service level requirements detailed in a Service Level Agreement (SLA). Decisions whether to accept or refuse new sessions are made on the basis of a dynamically adjusted constraint on the admission rate. This constraint is updated and kept consistent with system capacity and time-varying traffic behavior by means of an autonomous and continuous self-learning activity. This latter is of primary importance if human supervision is to be avoided. In particular, the proposed system is capable of self-configuring its component level parameters according to performance requirements. It rapidly adapts its configuration even under time-varying system capacity as in the case of server faults and recovery, or during runtime system maintenance. Furthermore, it self-protects from overloads by autonomously tuning its own responsiveness.

Unlike previous solutions, our approach ensures the reactivity necessary to deal with flash crowds. These are detected as soon as they arise by a simple change detection mechanism. The faster the traffic changes, the higher the rate of policy updates. This rate will be set back to lower values when the workload conditions return to normality.

Our proposal is oriented to the management of web-based traffic, and for this reason provides admission control at session granularity. Nevertheless, it does not require any prior knowledge about incoming traffic, and can be applied to non-session based traffic as well.

We designed a synthetic traffic generator, based on two industrial standard benchmarks, namely SPECWEB2005 and TPC-W, which we used to run simulations under a wide range of operating conditions. We compared SOC with other commonly adopted approaches: it turns out that SOC is superior to previous solutions in terms of performance and stability even in the presence of flash crowds.

A wide range of experiments have been conducted to test the sensitivity of the proposed solution to the configuration of the few start-up parameters. The experiments show that the behavior of our policy is not dependent on initial parameter setting, while other policies achieve an acceptable performance only when perfectly tuned and in very stable scenarios.

The experiments also highlight the ability of the system to react autonomously to flash crowds, server faults and runtime capacity adjustments, which, to the best of our knowledge, were never addressed by other previously proposed policies.

Our paper is organized as follows: in section 2 we formulate the problem of overload control in distributed web systems. In section 3 we sketch the basic actions of the proposed overload control policy. In section 4 we introduce our algorithm in deeper detail and in Section 5 we explain the self-learning mechanism at the basis of our approach. In section 6 we introduce some previous proposals that we compare with our own in section 7. Section 8 outlines the state of the art of admission control in autonomic distributed web systems while section 9 sets forth our conclusions.

2 The problem

We tackle the problem of admission control for web based services. In this context, the user interacts with the application by issuing a sequence of requests and waiting for the related responses. Such requests are logically related and form a so-called navigation session. As justified by [1, 2] we make the admission control work at session granularity, i.e., requests belonging to admitted sessions are never filtered by the admission controller.

Since the system should promptly react to traffic anomalies, any type of solution that requires human intervention is to be excluded. For this reason we address this problem by applying the autonomic computing [3] design paradigm.

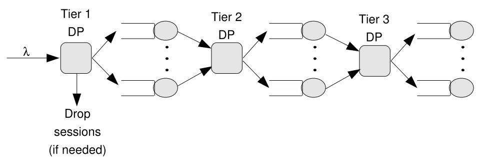


Fig. 1. Three tier architecture of a web cluster

We consider a multi-tier architecture [4, 5]. Each request may involve execution at different tiers. The most typical example of such an architecture is the three-tier cluster organization, where pure http requests involve the http server tier, servlet requests involve the application tier, while database requests reach the third tier requiring the execution of a query to a database. The reference architecture is shown in figure 1. Each tier may be composed of several replicated servers, while a front-end dispatcher hosts the admission control and dispatch module. The processing time of requests at different tiers may vary significantly.

The quality of service of web applications is usually regulated by an SLA. Although our work may be applied to different formulations of the SLA, we introduce the following one that takes into account the heterogeneity of the cluster tiers:

- $RT^{i}_{\mathtt{SLA}}$: maximum acceptable value of the 95th percentile of the response time for requests of type $i \in \{1, 2, \ldots, K\}$, where K is the number of cluster tiers;
- $\lambda_{\mathtt{SLA}}$: minimum guaranteed admission rate. If $\lambda_{\mathtt{in}}(t)$ is the rate of incoming sessions, and $\lambda_{\mathtt{adm}}(t)$ is the rate of admitted sessions, this agreement imposes that $\lambda_{\mathtt{adm}}(t) \geq \min\{\lambda_{\mathtt{in}}(t), \lambda_{\mathtt{SLA}}\}$;
- $T_{\mathtt{SLA}}$: observation interval between two subsequent verifications of the SLA constraints.

Meeting these quality requirements under sudden traffic variations, dynamic capacity changes or even faults, requires novel techniques that guarantee the necessary responsiveness. In such cases, respecting the agreement on response time is a challenging problem. Some other performance issues arise as well, such as the necessity of avoiding the oscillatory behavior that typically affects some previous proposals, as we show in the experimental section 7.

3 The idea

We designed SOC, a session-based admission control policy that dynamically self-configures a constraint on the incoming rate of new sessions. This constraint corresponds to the maximum load that the system can sustain without violating the agreements on the quality of service. It cannot be set off-line because it depends on the particular traffic rate and profile that the system has to face. Indeed, the same session rate may induce very different workloads on the system and dramatically different response times.

In order to make our proposal work without any prior assumptions about incoming traffic, we introduce a *self-learning activity* that makes the system aware of its capacity to sustain each particular traffic profile as the profile is when it comes. To this purpose, we make the system measure the relationship between the rate of admitted sessions and the corresponding measure of response time. By accurately processing raw measurements, the system can "learn" the maximum session admission rate that can be adopted in observance of the SLA requirements. A proper admission probability is therefore calculated considering the incoming session rate and the maximum tolerable rate.

Our algorithm is sketched in figure 2. SOC works in two modes, namely *normal mode* and *flash crowd management mode*, switching from one to the other according to the traffic scenario under observation. During stable load situations the system works in the normal mode and performance controls are regularly paced at time intervals of length $T_{\mathtt{AC}}^{\mathtt{SOC}}$. If a sudden change in traffic scenario is detected, the system enters the flash crowd management mode, during which policy updates are made at every new session arrival in order to control the overload. The introduction of this mode ensures accuracy and responsiveness at the same time.

As soon as the overload is under control, the system goes back to the normal mode and starts again performing periodic updates of the admission policy and of the knowledge basis.

Under both modes, capacity changes may occur at runtime, as a consequence of server failures or maintenance operations. The system is informed regarding the occurrence of such events because either a server does not reply as requested, as in the case of server failures, or an explicit message comes from the administrator who is performing maintenance procedures.

In both cases, the acquired knowledge is scaled and reinterpreted on the basis of the available information regarding the occurred capacity change. This makes our system capable of working jointly with another overload control technique: dynamic provisioning [6].

According to our proposal, the admission controller should operate at the application level of the protocol stack. This is because session information is necessary to discriminate which requests are to be accepted (namely requests belonging to already ongoing sessions), and which can be refused (requests that imply the creation of a new session). The cluster dispatcher can discriminate between new requests and requests belonging to ongoing sessions

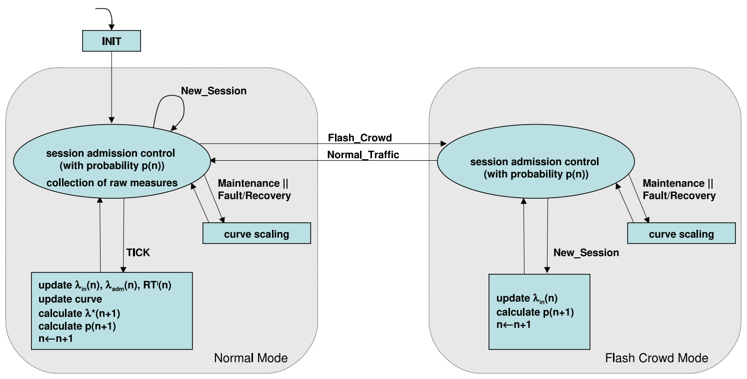

INIT
New_Session
session admission control (with probability p(n)) collection of raw measures
Flash_Crowd
Normal_Traffic
session admission control (with probability p(n))
Maintenance || Fault/Recovery
curve scaling
TICK
update $\lambda_{in}(n)$, $\lambda_{adm}(n)$, $RT^i(n)$
update curve
calculate $\lambda^*(n+1)$
calculate p(n+1)
$n \leftarrow n+1$
Normal Mode
Maintenance || Fault/Recovery
curve scaling
New_Session
update $\lambda_{in}(n)$
calculate p(n+1)
$n \leftarrow n+1$
Flash Crowd Mode

Fig. 2. Self-* Overload Control (SOC) algorithm

because either a cookie or an http parameter is appended to the request. This technique ensures two important benefits: 1) the admission controller can be implemented on DPs, and does not require any modification of client and server software, 2) the dispatcher can immediately respond to non-admitted requests, sending an "*I am busy*" page to inform the client of the overload situation. This avoids the expiration of protocol time-outs affecting the user-perceived performance and also mitigates the retrial phenomenon.

4 Self-* Overload Control (SOC) Policy

SOC provides a probabilistic admission control mechanism which filters incoming sessions according to an adaptive rate limit λ^*. This limit is calculated on the basis of the information gathered during a continuous self-learning activity. In order to perform this activity, the system takes measurements to learn the relationship between the observed Response Time (RT) and the rate of admitted sessions. The value of λ^* is then calculated as the highest rate that the site can support without violating the constraints on the response time defined in the SLAs. The admission control policy is then formulated by varying the admission probability according to a prediction of the future workload and to the estimated value of λ^*.

As described in figure 2, algorithm actions are triggered by different events. Under normal mode, new sessions, whose arrival determines the event `New_Session`, are blocked by the admission controller with a given probability. In this mode, the system continuously takes raw measurements related to the session interarrival time and to the response time of requests belonging to admitted sessions. The clock tick, identified by the event `TICK`, makes the system process the raw measurements and update the knowledge basis (self-learning activity). After this knowledge update, the system calculates a new upper limit on the session rate and consequently a new admission probability. The detection of a traffic surge, associated to the event `Flash_Crowd`, brings the system to the flash crowd mode. Under this second operative mode the system adjusts the admission probability at each event `New_Session` (i.e. much more frequently than in the normal mode). Notice that the system is not allowed to alter the knowledge basis when working in the flash crowd mode, given the sudden variability of the observed traffic pattern. When the system detects that the flash crowd is under control, in correspondence to the event `Normal_Traffic`, the algorithm switches back to normal mode. Under both modes, the events `Maintenance` and `Fault/Recovery` cause the execution of a knowledge scaling procedure as described in section 5.3.

The transition between the two modes is regulated by a change detection mechanism and may occur at any time t from the start of the current iteration (last `TICK` event).

In order to describe this mechanism, we introduce some notations. We denote by N the number of admitted sessions in the last t seconds. We also denote by σ_λ the standard deviation of the admitted rate observed at runtime when the admission control is actually blocking some traffic. The parameter σ_λ measures the intensity of the inherent variability of the admitted rate $\lambda_{\mathtt{adm}}$.

The system switches from normal to flash crowd mode if both the following conditions are met:

- the currently admitted rate exceeds the limit λ^* by q times the measured standard deviation of the admitted rate, that is $((N/t) > (\lambda^* + q \cdot \sigma_\lambda))$,
- N exceeds the expectations for a single iteration, that is $(N > \lambda^* \cdot T_{\mathtt{AC}}^{\mathtt{SOC}})$,

where the second condition is introduced as a complement to the first, to avoid the occurrence of change detections in consequence of occasionally short interarrival times between subsequent sessions.

The system returns to normal mode only when the instantaneous rate of admitted sessions returns under the limit λ^*. In this case we can assume the unexpected surge is under control and the policy can return to normal mode, during which performance controls are paced at a slower and regular rate.

4.1 Normal mode

Under the normal operative mode, described in the left block of figure 2, the system admits new sessions with a probability that is updated every $T_{\mathtt{AC}}^{\mathtt{SOC}}$ seconds, that is, at each `TICK` event. This probability update is necessary to ensure the fulfillment of the SLA under time varying traffic conditions. The update mechanism works on the basis of the information made available by the self-learning activity detailed in section 5.

This mode provides a probabilistic session admission control. At the beginning of the n-th iteration, the admission probability $p(n)$ is calculated on the basis of a prediction of the system capacity to serve incoming traffic in observance of the SLA requirements. To this end the system estimates the future incoming session rate $\hat{\lambda}_{\mathtt{in}}(n)$ and the maximum rate $\lambda^*(n)$ that it can sustain without violating the agreements under the observed traffic profile.

The estimate of the incoming session rate $\hat{\lambda}_{\mathtt{in}}(n)$, for the next iteration interval $[t_n, t_{n+1})$, is obtained by means of an exponential moving average with weight $\alpha = 0.5$.

We assume that an estimate of the current session arrival rate $\hat{\lambda}_{\mathtt{in}}(n)$ can be calculated as follows $\hat{\lambda}_{\mathtt{in}}(n) = \alpha\lambda_{\mathtt{in}}(n-1) + (1-\alpha)\hat{\lambda}_{\mathtt{in}}(n-1)$. The algorithm is sufficiently robust to possible false predictions, as they will be corrected at the next iterations, making use of updated statistics. Notice that although a complete analysis of the sensitivity of this load prediction is out of the scope of this paper, experiments have been conducted by using prediction based on only instantaneous values of the interarrival session rate, with similar results. The introduction of the exponential average only reduces the noise in the measurements.

In order to estimate the upper limit $\lambda^*(n)$ of the incoming session rate to be adopted to meet the agreements, the algorithm makes use of the information collected by means of the self-learning activity. For the sake of clarity, we defer the description of this activity to section 5. In this section we only say that, through this activity, the system obtains an updated estimate of the functions that relate the admitted session rate $\lambda_{\mathtt{adm}}$ and the response time RT^i at each tier i, $i = 1, \ldots, K$. We indicate such functions by $\hat{f}^i_{(n)}(\lambda)$, as calculated at the n-th iteration.

The upper limit $\lambda^*(n)$ is obtained by inverting the functions $\hat{f}^i_{(n)}(\lambda)$ in correspondence to the value of the maximum tolerable response time $RT^i_{\mathtt{SLA}}$ specified in the SLA and calculating $\lambda^*(n) = \min_{i=1,\ldots,K} \lambda^*_i(n)$, where $\lambda^*_i(n) = \hat{f}^{i^{-1}}_{(n)}(RT^i_{\mathtt{SLA}})$.

The DP can configure the session admission probability according to the forecast of incoming traffic given by $\hat{\lambda}_{\mathtt{in}}(n)$ and admit new sessions with probability $p(n) = \min\{1, \lambda^*(n)/\hat{\lambda}_{\mathtt{in}}(n)\}$.

This on-line self-tuning of the admission probability has several benefits. On the one hand, the highest possible rate of incoming sessions is admitted. On the other hand, it prevents the system from being overloaded, by quickly reducing the admission probability as the traffic grows.

The initial parameter setting of our algorithm has little impact on its performance thanks to its self-configuration and self-adaptation capabilities. Indeed, even if the starting setup is incorrect, the self-learning activity will soon create the knowledge basis needed to adapt the algorithm configuration. As initial setting we use $p(0) = 1$ and continue admitting all incoming sessions until the first estimate of $\hat{f}^i_{(n)}(\lambda)$ becomes available.

4.2 Flash crowd mode

The flash crowd mode is introduced when incoming traffic shows a sudden surge, possibly related to the occurrence of a flash crowd. This mode is described in the right block of figure 2. Its execution makes statistical metrics be updated every time a new session is admitted, thus ensuring both reactivity and adaptivity.

Although statistical metrics are updated at each session arrival, no learning mechanism is activated in flash crowd management mode, due to the high variability of incoming traffic. The purpose of statistical metrics updating is the continuous monitoring of the intensity of incoming traffic $\lambda_{\mathtt{in}}$. This value is used to update the admission probability $p(n)$ accordingly, by setting $p(n) = \lambda^*/\lambda_{\mathtt{in}}$.

In flash crowd mode, all statistical metrics are calculated over a moving window which comprises the set of the last $\lfloor \lambda^* \cdot T^{\mathtt{SOC}}_{\mathtt{AC}} \rfloor$ admitted sessions. The higher the incoming session rate $\lambda_{\mathtt{in}}$, the smaller the time interval during which the statistical metrics are calculated, thus ensuring system reactivity without compromising the reliability of the measurements.

5 Self-learning activity

The aim of this self-learning activity is to store and update information regarding the functions $\hat{f}^i_{(n)}(\cdot)$. These functions represent an estimate of the relationship between the observed traffic rate $\lambda_{\mathtt{adm}}$ and the corresponding response time RT^i, $i = 1, \ldots, K$ at each tier. This activity is of primary importance for determining the autonomous behavior of SOC. It is performed at runtime when the system is working in normal operative mode and is made possible by the continuous collection of raw measurements related to session interarrival time and to request execution time.

All actions mandated by the normal operative mode are iteratively executed at regular time intervals of the same length as the admission control period, that is, $T^{\mathtt{SOC}}_{\mathtt{AC}}$ seconds.

5.1 *Measurement analysis*

During each iteration the system takes measurements of the response time of the different tiers. It should be noted that request differentiation occurs only for measurement purposes, that is, after the request execution. For this reason it occurs without mis-classification problems.

We define $\mathcal{T}_i^n$ as the set of raw measurements of response time for requests of type $i, i \in \{1, 2, \ldots, K\}$ taken during the n-th iteration interval $[t_n, t_{n+1})$ of length $T_{\mathtt{AC}}^{\mathtt{SOC}}$.

At the end of each iteration, the system calculates the following statistical metrics:

- RT_n^i, that is, the 95th percentile of the set $\mathcal{T}_i^n$, for $i \in \{1, 2, \ldots, K\}$;
- $\lambda_{\mathtt{in}}(n)$, that is, the average incoming rate of new sessions observed during the time interval $[t_n, t_{n+1})$;
- $\lambda_{\mathtt{adm}}(n)$, that is, the average rate of admitted sessions during the time interval $[t_n, t_{n+1})$.

It should be emphasized that a statistical metric calculated from samples of raw measurements taken during a single iteration interval is insufficiently reliable for two reasons. First, workload is subject to variations that may cause transient effects. Second, the number of samples may not be sufficient to ensure an acceptable confidence level. The use of longer sampling periods may allow the collection of more numerous samples, but it is impossible to define the length of such a period for any possible traffic situation. Indeed, the incoming workload may change before a sufficiently representative set of samples is available. Moreover, too long a sampling period may lead to the low responsiveness of the admission policy.

The idea at the basis of our proposal is to collect these statistics under the range of workload levels observed during the past history of the system.

With this aim, let us consider the set of *stat pairs*:
$\mathcal{R}_i \triangleq \{(\lambda_{\mathtt{adm}}(n), RT_n^i), n \in \{0, 1, \ldots\}\}$.

We partition the Cartesian plane into rectangular intervals, named *slices*, of length l_λ along the $\lambda_{\mathtt{adm}}$ axis, as shown in figure 3. In the experimental section 7.1, we show that experiments prove the independence of the algorithm performance from the initial setting of l_λ. This is due to the aggregation technique that we detail as follows.

For every interval $[(k-1)l_\lambda; kl_\lambda)$, with $k = 1, 2, \ldots$ we define $\mathcal{P}_k^i = \{(\lambda_{\mathtt{adm}}, RT^i) | \lambda_{\mathtt{adm}} \in [(k-1)l_\lambda; kl_\lambda)\}$ that is the set of i-th tier stat pairs lying in the k-th slice.

Then we calculate the *barycenter* $B_k^i = (\lambda_k^B, RT_k^{Bi})$ of the k-th interval as the point with average coordinates over the set $\mathcal{P}_k^i$. In particular,

$$B_k^i \triangleq \Big(\sum_{(\lambda_z, RT_z^i) \in \mathcal{P}_k^i} \lambda_z / |\mathcal{P}_k^i|, \sum_{(\lambda_z, RT_z^i) \in \mathcal{P}_k^i} RT_z^i / |\mathcal{P}_k^i| \Big)$$

for each tier i, where $i \in \{1, 2, \ldots, K\}$.

An interval has no barycenter if $\mathcal{P}_k^i = \emptyset$.

Figure 3 shows the collected stat pairs taken at run-time at the database tier of a typical scenario. It also highlights the barycenters for each interval. This picture shows the curve labeled *RT curve* obtained through offline benchmarks in order to give a visual representation of the ideal curve that would be constructed in case of perfect knowledge.

Every time a new point is added to a set $\mathcal{P}_k^i$, the system updates the cardinality of the set being modified, and recalculates the related barycenter. It should be noted that this update is performed for only one set at a time (sets that have not been modified do not require statistic updates) and is incrementally calculated with respect to a short statistical representation named *barycenter summary*. The use of such a summary avoids computational and storage costs that the system would incur if all pairs had to be considered. In particular the summary of the barycenter B_k^i is composed of the tuple $< (\lambda_k^B, RT_k^{Bi}), (\sigma_\lambda(B_k^i), \sigma_{RT}(B_k^i)), N(B_k^i) >$ where the first element represents the barycenter coordinates, the second element is the standard deviation in both coordinates, and the third element is the cardinality of the related slice, that is, $N(B_k^i) = |\mathcal{P}_k^i|$.

We note that not all barycenters are alike in terms of representativeness and reliability. We quantitatively evaluate their representativeness in terms of standard error in both coordinates. Barycenters calculated with a standard error lower than $t_{\texttt{err}}$ in any coordinate are considered *reliable*, while the others are discarded. In particular we consider a barycenter B_k^i reliable if:

$$\max\left\{ \frac{\sigma_\lambda(B_k^i)}{\sqrt{N(B_k^i)}}, \frac{\sigma_{RT}(B_k^i)}{\sqrt{N(B_k^i)}} \right\} \leq t_{\texttt{err}}.$$

It should be emphasized that other metrics of reliability can be introduced instead of the standard error without significant impact on the proposed methodology. In a previous study [7] we had already achieved appreciable results by considering a measure of reliability as simple as the cardinality $N(B_k^i)$.

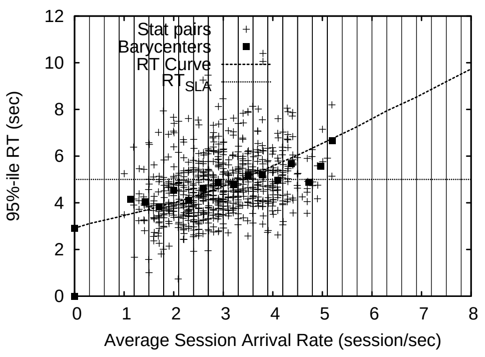


Fig. 3. Regular slice barycenters

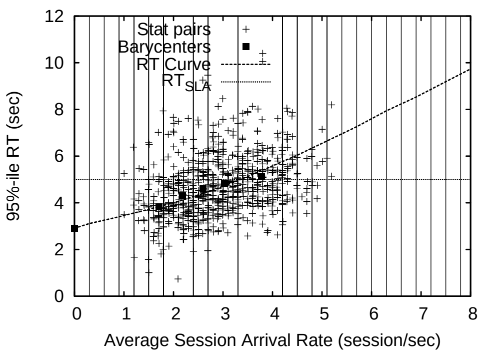


Fig. 4. Aggregated slice barycenters

5.2 Curve construction

Let $\mathcal{B}_i$ be the set of reliable barycenters for the i-th tier, ordered on the basis of the first coordinate $\lambda_{\mathtt{adm}}$. Since we know that the relation between $\lambda_{\mathtt{adm}}$ and RT^i is monotonically not decreasing, we can assume that if two subsequent barycenters of $\mathcal{B}_i$ do not satisfy this basic monotonicity property, the corresponding slices can be aggregated to improve measurement reliability. To this end the sets of stat pairs related to the corresponding intervals are aggregated and a new barycenter is calculated for the aggregated slice. The slice aggregation process continues until $\mathcal{B}_i$ contains only barycenters in growing order in both the coordinates, as shown in figure 4. We note that this procedure permits a further validation of the measurements, in addition to the previously performed test on the standard error values. This makes our proposal less sensitive to the particular barycenter reliability metric in use.

After a few aggregations, the function $f_i(\cdot)$ that relates $\lambda_{\mathtt{adm}}$ and RT^i is estimated by means of linear interpolation of the set $\mathcal{B}_i$, thus obtaining the piecewise linear function $\hat{f}^i_{(n)}(\cdot)$, where the subscript (n) stands for the iteration step at which the estimate is updated. In particular, the interpolation between the last two points is extended to provide at least a rough estimate even for values of $\lambda_{\mathtt{adm}}$ that are greater than those actually sampled.

For the start-up setting of the curve construction phase, we use $\mathcal{B}_i = \{P^i_{\mathtt{bench}}\}$, where $P^i_{\mathtt{bench}} = (0, RT^i_{\mathtt{bench}})$ represents a lower bound on the 95th percentile of the response times of type i requests. This point is the 95th percentile of response time measured at the i-th tier, when the system is in a completely idle state, that is when $\lambda_{\mathtt{adm}} \cong 0$. In order to calculate the average response time in such a situation, we use an offline benchmark, obtaining the points $P^i_{\mathtt{bench}} = (0, RT^i_{\mathtt{bench}})$, $i \in \{1, 2, \ldots, K\}$.

The proper setting of the points $P^i_{\mathtt{bench}}$ is not a key point in the algorithm, since as self-learning activity progresses, more measures are available to build $\hat{f}^i_{(n)}(\cdot)$ with growing accuracy.

Due to the frequent updates, the constructed curve constitutes a highly dynamic structure that continuously adapts itself to changing workload situations. The function $\hat{f}^i_{(n)}(\cdot)$ can be used to obtain a prediction of the response time corresponding to any possible incoming workload rate.

We note that the use of common regression techniques as an alternative to linear interpolation is inadvisable, because it would require a prior assumption about the type of functions being parametrized for the regression. Experiments conducted using different traffic profiles (e.g., with SPECWEB2005 [8] and TPC-W [9] oriented traffic generators) show that, apart from monotonicity, no other structural property is generally valid for all possible traffic scenarios. This would make it difficult to choose the correct type of regression (e.g. polynomial, exponential, power law).

5.3 Self-learning in the case of capacity variations

The resource availability of a web cluster may vary over time for several reasons. For example, capacity variations may occur when service providers reinforce the server pool by adding new servers or substituting old ones. Dynamic provisioning schemes may imply autonomous architectural readjustments [6] as well. Lastly, resource unavailability may occur as a consequence of server failures.

This self-learning activity needs to take this variability into account since it means it is necessary to update the learned information on the basis of the new resource configuration.

Before the system collects sufficient measurements reliably to update self-learned information, there is a time interval during which an insufficient number of post-capacity variation measurements are available to reliably construct the functions $\hat{f}^i_{(n)}(\cdot)$. For this reason, we let the system use the information gathered prior to the capacity change by scaling it proportionally to the degree of the capacity variation.

We assume the existence of a mechanism appropriate to permit the system to detect a capacity variation and its extent as soon as it occurs. Let $C_{\texttt{pre}}$ and $C_{\texttt{post}}$ be the initial and final capacity respectively, in terms of capacity units. Note that the concept of capacity unit does not imply server uniformity. Each server capacity can be measured in terms of capacity units by executing an offline benchmark before the addition of the server to the cluster. We measure the extent of the capacity variation by means of the ratio $\rho \triangleq C_{\texttt{post}}/C_{\texttt{pre}}$. This ratio is used to scale the knowledge basis constructed before the variation.

We remind the reader that each barycenter is associated with a summary of its statistic properties, namely coordinates, standard deviation and cardinality of the related slice. The scaling procedure consists of constructing a new set $\mathcal{B}^i_\rho$ by scaling the values of the barycenter summaries for all barycenters in $\mathcal{B}^i$. In particular, the barycenter B^i_k, summarized by the tuple $< (\lambda^B_k, RT^{Bi}_k), (\ \sigma_\lambda(B^i_k),\ \sigma_{RT}(B^i_k)),\ N(B^i_k)\ >$, is scaled to the barycenter $B^i_{k\rho}$ with the summary $< s_1, s_2, s_3 >$, where:

$$s_1 = (\rho\lambda^B_k, RT^{Bi}_k);$$

$$s_2 = (\rho\sigma_\lambda(B^i_k),\ \sigma_{RT}(B^i_k));$$

$$s_3 = \max\left\{\left[\frac{\rho\sigma_\lambda(B^i_k)}{t_{\texttt{err}}}\right]^2, \left[\frac{\sigma_{RT}(B^i_k)}{t_{\texttt{err}}}\right]^2\right\}.$$

Note that the value of s_3 is chosen as the minimum cardinality that still ensures barycenter reliability. In this way, the scaled knowledge basis is responsive to the addition of new measurements that will either confirm or invalid barycenter reliability.

6 Other admission control strategies

In this section we describe other previously proposed QoS policies, and make performance comparisons in the next section. These policies can be formulated in a variety of ways depending on the performance objective under consideration. We limit our analysis to the optimization of response time, since this is closely related to user perceptions of quality.

6.1 Threshold-based admission control

Fixed threshold policies have been proposed in many fields of computer science, and in particular for web applications with several variants [1, 10, 11].

According to the Threshold Based Admission Control (TBAC) policy, the DP makes periodic evaluations of the 95th percentile of response time of each tier, every $T^{\texttt{TBAC}}_{\texttt{AC}}$ seconds. If there is at least one tier for which the 95th percentile of response time exceeds a threshold $RT^{\texttt{TBAC}}$, the DP rejects new sessions and only accepts requests that belong to ongoing sessions.

On the contrary, if the value of the 95th percentile of response time at each tier is lower than $RT^{\texttt{TBAC}}$, all new sessions are accepted for the next $T^{\texttt{TBAC}}_{\texttt{AC}}$ seconds.

Similarly to all threshold-based policies, this policy implies a typical on/off behavior of the admission controller, causing unacceptable response time oscillations. Furthermore, its performance is particularly sensitive to parameter set-up (i.e., the choice of the threshold $RT^{\texttt{TBAC}}$ and of the period between two succeeding decisions $T_{\texttt{AC}}^{\texttt{TBAC}}$). For this reason it cannot be used in traffic scenarios characterized by highly variable workloads.

6.2 Probabilistic Admission Control

The Probabilistic Admission Control (PAC) is a well known technique in control theory and is commonly used when oscillations need to be avoided. This policy was proposed for Internet services in [12], while a similar version was also proposed for web systems in [13].

According to this policy, a new session is admitted with a certain probability whose value depends on the measured response time. Every $T_{\texttt{AC}}^{\texttt{PAC}}$ seconds, at the n-th iteration, the DP evaluates the 95th percentile of response time of each tier, RT_n^i, $i = 1, \ldots, K$, and compares it with two thresholds, $RT_{\texttt{low}}^{\texttt{PAC}}$ and $RT_{\texttt{high}}^{\texttt{PAC}}$.

The session admission probability is a piece-wise linear function of RT_n^i formulated as follows: $p(n+1) = \min_{i=1,\ldots,K} p_i(RT_n^i)$, where $p_i(RT_n^i)$, abbreviated with p_i^n, has the following value:

$$p_i^n = \begin{cases} 1 & \text{if } RT_n^i \le RT_{\texttt{low}}^{\texttt{PAC}} \\ \frac{RT_{\texttt{high}}^{\texttt{PAC}} - RT_n^i}{RT_{\texttt{high}}^{\texttt{PAC}} - RT_{\texttt{low}}^{\texttt{PAC}}} & \text{if } RT_{\texttt{low}}^{\texttt{PAC}} < RT_n^i \le RT_{\texttt{high}}^{\texttt{PAC}} \\ 0 & \text{if } RT_n^i > RT_{\texttt{high}}^{\texttt{PAC}} \end{cases} \tag{1}$$

Note that the two threshold values $RT_{\texttt{high}}^{\texttt{PAC}}$ and $RT_{\texttt{low}}^{\texttt{PAC}}$ that characterize this policy are arbitrarily set offline independently of the observed incoming session rate and of the time $T_{\texttt{AC}}^{\texttt{PAC}}$ between two succeeding decisions. As we show in section 7, the performance of this policy is heavily dependent on the proper tuning of these parameters.

7 Simulation Results

In this section we present a comparative analysis of the SOC, TBAC and PAC policies. We also conduct a performance study of the SOC policy under several working scenarios. This serves to evidence its stability and its capabilities to react promptly to sudden workload changes and capacity variations.

We developed a simulator on the basis of the OPNET modeler software [14].

In our simulation setting, we assume that the interarrival time of new sessions follows a negative exponential distribution with average $1/\lambda_{\mathtt{in}}$. The interarrival time of requests belonging to the same session is more complex and follows the phase model of an industrial standard benchmark: SPECWEB2005 [8].

Upon reception of a response, the next request is sent after a think-time interval $T_{\mathtt{think}}$. Our model of $T_{\mathtt{think}}$ is based on TPC-W [9,15] and on other studies in the area of web traffic analysis such as [16]. As in the TPC-W model, we assume an exponential distribution of think times with a lower bound of 1 sec. Therefore $T_{\mathtt{think}} = \max\{-\log(r)\mu, 1\}$ where r is uniformly distributed over the interval [0,1] and $\mu = 10$ sec.

To model user behavior realistically, we also introduce a timeout representing the maximum response time tolerable by users. If the timeout expires before the reception of a response, the client abandons the system.

In our experiments we refer to a typical three-tier cluster organization such as the one depicted in figure 1.

We map each phase of the session state model onto a specific tier of the three available ones.

We use an approximate estimate of the average processing times of the different tiers on the basis of the experiments detailed in [10]. We assume each session phase requires an exponentially distributed execution time set as follows: the average execution times of pure http, servlet and database requests are 0.001 sec, 0.01 sec and 1 sec, respectively.

For the sake of brevity, we conduct our analysis on the database tier which is the bottleneck of the architecture considered in these simulations. Thus we simplify the previously introduced notation leaving out the tier index. This way the limit on the database response time defined in the SLA is indicated with $RT_{\mathtt{SLA}}$.

Unless stated otherwise, all the experiments of this section are conducted with 20 application servers, a client timeout of 8 sec and $RT_{\mathtt{SLA}} = 5$ sec. The fixed threshold $T^{\mathtt{TBAC}}_{\mathtt{AC}}$ of the TBAC policy is set in agreement with the SLA constraints on the 95th percentile of database response time, therefore $T^{\mathtt{TBAC}}_{\mathtt{AC}} = RT_{\mathtt{SLA}}$. The thresholds of the PAC policy are set as follows: $T^{\mathtt{PAC}}_{\mathtt{low}} = 3$ sec and $T^{\mathtt{PAC}}_{\mathtt{high}} = RT_{\mathtt{SLA}}$, in agreement with SLA constraints.

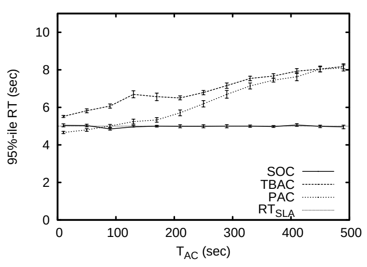


Fig. 5. 95th percentile of RT

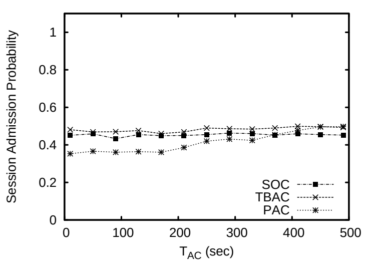


Fig. 6. Session admission probability

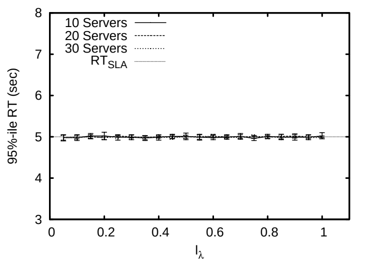


Fig. 7. 95th percentile of RT

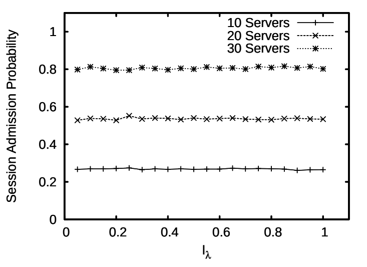


Fig. 8. Session admission probability

7.1 Parameter sensitivity

This set of experiments is introduced to study the sensitivity of these policies to the parameter setting. The focus of this study is two key parameters: the length of the admission control period $T_{\mathtt{AC}}$, common to the three policies under consideration, and the amplitude of the intervals partitioning the $\lambda_{\mathtt{adm}}$ axis, l_λ, used in in the SOC policy proposed in this paper.

7.1.1 Sensitivity to the setting of T_{AC}^{SOC}

The aim of these experiments is to study the performance sensitivity of SOC to the $T_{\mathtt{AC}}$ parameter, that is, the length of the admission control period.

Figures 5 and 6 show that the performance of the SOC policy does not vary significantly with variations in $T_{\mathtt{AC}}$. The same cannot be said for the other two policies, TBAC and PAC. To understand the reasons for the different behavior of the three policies, let us first consider the case of SOC. For short values of $T_{\mathtt{AC}}$, statistical metrics are calculated frequently on the basis of poor sets of raw measurements. The low confidence level of such metrics is compensated by the high number of points that are used to construct the set of barycenters $\mathcal{B}$.

Such points contribute to the quantitative evaluation of the reliability measure defined on the set of barycenters. The choice of reliable barycenters allows our algorithm to reconstruct the correct relationship between the rate of admitted sessions and the 95th percentile of response time and then to find the maximum admission rate that can be adopted while still respecting quality agreements. With increasing $T_{\mathtt{AC}}$, fewer points are available for the construction of the curve set. In this case, although the curve set is constructed on the basis of a smaller number of points, it still permits a good estimate of the maximum adoptable admission rate because these points have a higher confidence level.

Although the period between subsequent decisions is a parameter that requires manual sizing, this set of experiments shows that there is no need to perform fine and laborious tuning, since the behavior of the SOC policy is almost the same for a wide range of values of this parameter.

The TBAC and PAC policy obviously benefit from shorter periods between successive decisions. Short decision periods make the system capable of rapidly correcting wrong decisions and of reacting to workload changes. By contrast, long decision periods induce more intense oscillations in the case of high loads, and cause scarce utilization in low load situations.

The TBAC and PAC policies take advantage of short periods, while the SOC policy is virtually independent of this setting. Thus, in order to compare the three policies fairly we use the following parameter setting for the subsequent experiments: $T_{\mathtt{AC}}^{\mathtt{TBAC}} = T_{\mathtt{AC}}^{\mathtt{PAC}} = 10$ sec, which is a sufficiently short period to allow the TBAC and PAC policies to work optimally. Since this setting has no impact on the behavior of the SOC policy, we set $T_{\mathtt{AC}}^{\mathtt{SOC}} = 60$ sec. This relieves the system of the unnecessarily frequent decisions that would be made with shorter inter-observation periods.

7.1.2 Sensitivity to the setting of l_λ

The SOC policy proposed in this paper, requires manual definition of the l_λ parameter, that is, the amplitude of the intervals partitioning the λ_{adm} axis. The purpose of these experiments is to show that the setting of this parameter does not influence the performance of our policy, permitting a correct self-configuration of the optimum admission rate λ^* under a wide range of values of l_λ. To this end, we analyzed performance in terms of 95th percentile of response time and session admission probability, varying the amplitude of the intervals. Moreover, we considered three different architectures with 10, 20 and 30 application servers, respectively.

Figure 7 shows the measured 95th percentile of response time as a function of l_λ. This figure highlights how the SOC policy can meet the SLA independently of the amplitude of the intervals and also under different system capacities. Figure 8 reveals that manual tuning of the l_λ is not labor-intensive because the system behaves properly under a wide range of values.

Moreover, the slice aggregation technique further reduces the dependency of

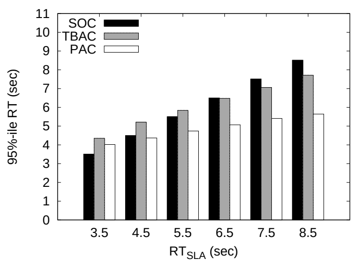


Fig. 9. 95th percentile of RT

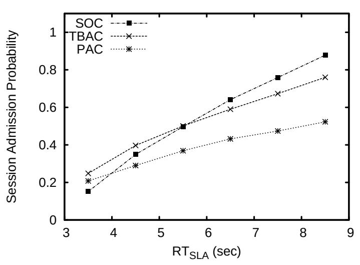


Fig. 10. Session admission probability

SOC behavior on the tuning of this parameter. Indeed, the system is able to self-configure a proper limit on the rate of admitted sessions, guaranteeing the fulfillment of the SLA and an optimal utilization of system resources, independently of the setting of l_λ and system capacity.

As a conclusion of this set of experiments, we can state that although SOC is based on the off-line configuration of l_λ, this does not significantly reduce its autonomy. In the following experiments we set $l_\lambda = 0.3$, without significant impact on the achieved results.

7.2 Performance under different SLA requirements

The purpose of this set of simulations (figure 9 and 10) is to test policy behavior under different SLAs. We tested the SOC, TBAC and PAC policies under six different values of the SLA threshold $RT_{\mathtt{SLA}}$, ranging from 3 to 8 seconds. We set $T^{\mathtt{TBAC}}_{\mathtt{AC}} = T^{\mathtt{PAC}}_{\mathtt{high}} = RT_{\mathtt{SLA}}$, while $T^{\mathtt{PAC}}_{\mathtt{low}} = 3$ sec. The incoming session rate was set to $\lambda_{\mathtt{in}} = 2.3$ sessions/sec.

Figures 9 and 10 show the 95th percentile of database response time and the new session admission probability respectively. These figures underscore the main drawbacks of the TBAC and PAC policies discussed earlier. In the case of low threshold (3.5-6.5 seconds), TBAC does not prevent SLA violations, while for higher thresholds it causes system under-utilization. This is demonstrated by the trend of the session admission probability shown in figure 10: too many sessions are accepted or refused for low and high SLA threshold values respectively. The same can be said for the PAC policy, for which figures 9 and 10 exhibit a SLA violation when $RT_{\mathtt{SLA}} \leq 4.5$ and system under-utilization for higher threshold values.

In contrast, our policy, instead, never violates the agreements. Indeed, thanks to its learning capabilities, SOC properly adapts the admission probability to the given performance requirements.

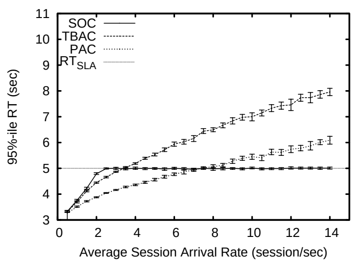


Fig. 11. 95th percentile of RT

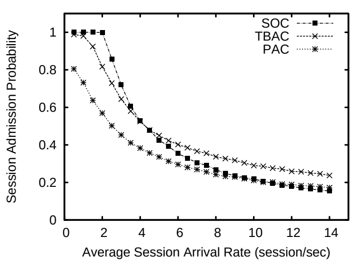


Fig. 12. Session admission probability

7.3 Performance under different workload intensities

The aim of this set of results (figures 11 and 12) is to study the TBAC, PAC and SOC policies under different system workloads.

The SOC policy correctly self-learns the relation between the response time and the rate of admitted sessions, independently of the incoming workload. Moreover, it self-optimizes resource utilization, admitting as many new sessions as possible, and finding the highest admission rate that respects SLA limits. On the one hand, as shown in figure 11, when traffic load is high, our policy finds the suitable session arrival rate and admits as many sessions as possible, while remaining within SLA limits. On the other hand, when traffic is low, it accepts almost all incoming sessions, as figure 12 shows.

In contrast to SOC, non-adaptive policies such as TBAC and PAC typically under-utilize system resources in low workload conditions, and violate QoS agreements when the workload is high.

This behavior of the TBAC policy is due to its on/off nature. Even in a low workload scenario, momentary peaks of requests can cause the DP to measure high values of the 95th percentile of response time, possibly registering an SLA violation for a short time interval. The TBAC policy reacts to this situation by refusing all new sessions for one or even more intervals, likely causing system under-utilization. The TBAC policy demonstrates performance problems in high load scenarios as well. When measurements reveal that new sessions can be admitted under SLA constraints, the TBAC policy admits all incoming new sessions for the subsequent inter-observation period, possibly causing an uncontrolled growth of the system load and oscillations of response time.

The PAC policy has similar problems, due to the way it calculates session admission probability for the next interval. Equation (1) assumes a linear dependence of response time on session admission probability. Given a value of the 95th percentile of response time, the PAC policy selects the corresponding value of the admission probability, independently of the incoming session rate. However, this value may not be suited to different workload scenarios.

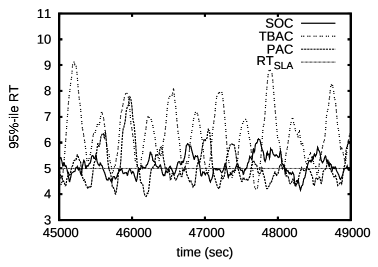


Fig. 13. 95th percentile of RT

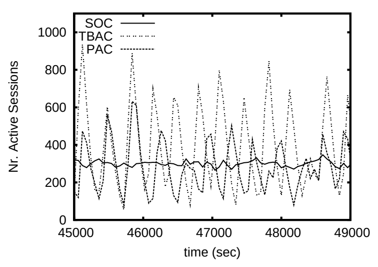


Fig. 14. Number of active sessions

Furthermore, this policy shows a high dependence on the tuning of the inter-observation period. This parameter is of primary importance because session-based admission decisions do not have an immediate effect on response time. An incorrect tuning of this parameter leads to oscillatory policy behavior, although less severe than in the case of the TBAC policy.

7.4 Performance stability

This set of results (figures 13 and 14) studies the problem of performance stability over time. To this end we analyze how the number of active users and the 95th percentile of response time vary over the simulation time.

The TBAC policy shows oscillatory behavior due to its on/off nature. The PAC policy was introduced with the explicit goal of reducing the oscillatory behavior that characterizes common threshold-based policies. Nonetheless, it also shows a high variability of the number of active sessions and response time.

One of the main reasons why these policies cannot guarantee stable behavior is that the admission controller works at session granularity. In fact, an admitted session traverses the cluster tiers according to a given life-cycle phase model, and for this reason it does not have an immediate impact on all cluster tiers. At the same time, a decision to stop admitting new sessions has an impact on perceived performance only after the end of a sufficient number of sessions. This can lead the TBAC policy to accept all incoming sessions for too many rounds before the measured response time exceeds the threshold. Once the threshold is exceeded, this policy reacts by refusing all incoming sessions. However, the response time remains over the threshold until the end of a sufficient number of sessions. The PAC policy reduces the amplitude of the oscillations compared with TBAC, but it does not solve this problem completely.

By contrast, the SOC policy maintains stable behavior for the response time and the number of active sessions. Its self-learning activity makes it possible to build a consolidated knowledge of system capacity relative to incoming traffic, which in turn is used to derive a good and stable estimation of the optimal admission rate. Through the self-monitoring phase, the admission probability is properly set with respect to the incoming load, thus stabilizing the number of active sessions, and consequently the response time.

7.5 Performance under time varying traffic load

In the following experiments we studied the capabilities of SOC to react properly to sudden changes in incoming traffic, with and without activating the flash crowd management mode described in section 4.2.

Figure 15 characterizes the traffic scenario of this set of experiments. It shows a session arrival rate subject to several sudden surges of growing intensity.

In figures 16, 17, 18 and 19, the complete version of SOC with both modes is named *Flash Crowd Management* (FCM). A basic version of SOC working only in normal mode (that is, obtained by de-activating the change detection mechanism) is called *Base*.

Figure 16 shows how the introduction of the flash crowd mode makes the system capable of significantly mitigating spikes in response time caused by the occurrence of flash crowds. In contrast, these spikes are evident in figure 17, which shows that without proper flash crowd management, a violation of the service level agreements is unavoidable.

Figures 18 and 19 focus on the management of the flash crowd that occurs at 100000 seconds of simulations.

These figures highlight the increased reactivity of SOC when using flash crowd management support. The Base version takes almost 40 seconds to discover the occurrence of a flash crowd and consequently modify admission probability; while the enhanced version reacts almost immediately.

Is is important to note the time-scale difference between figures 18 and 19, as well as the fact that a 40 second delay in discovering a flash crowd implies the system is in overload for almost 500 seconds. This is largely because the admission controller works at session granularity.

In particular, figure 19 shows how the Base version of SOC is incapable of managing such a flash crowd. This can be seen especially from the high values of the 95th percentile of response time, which not only do violate SLA constraints but even exceed user time-out. This means that users are abandoning the site due to poor performance or system unavailability. On the contrary, the use of flash crowd mode makes the system capable of maintaining response

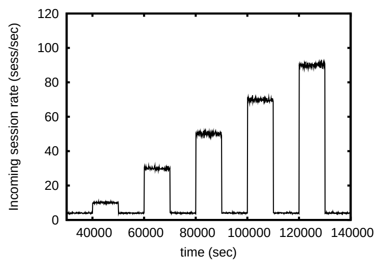
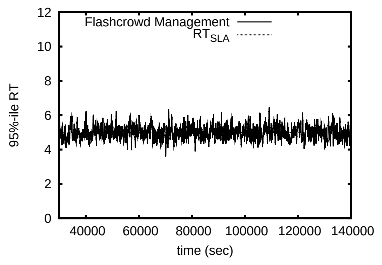


Fig. 15. Session arrival rate



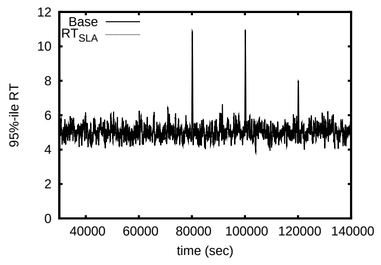
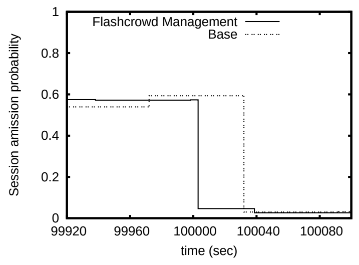


Fig. 16. 95th percentile of RT (FCM) Fig. 17. 95th percentile of RT (Base)



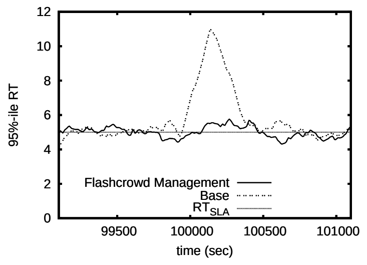


Fig. 18. Admission probability Fig. 19. 95th percentile of RT

time at acceptable levels by rapidly reducing session admission probability as soon as the surge in demand is detected.

7.6 Run-time Capacity Variations

The following experiments demonstrate the behavior of our algorithm under time-varying system capacity and stable traffic profile. In all the experiments of this subsection we varied only the capacity of the database tier. The first two tiers both maintained 10 equally equipped servers, whose average service time was 0.001 sec and 0.01 sec respectively. We introduce three sets of experiments conducted by increasing and decreasing the cluster capacity with homogeneous and heterogeneous servers.

7.6.1 Cluster of homogeneous servers

We start our analysis with a set of experiments in which the available capacity varies due to the addition of homogeneous servers. The number of servers at

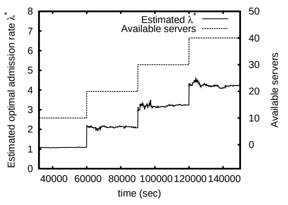


Fig. 20. Rate limit (homogeneous capacity increase)

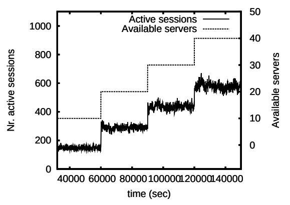


Fig. 21. Active sessions (homogeneous capacity increase)

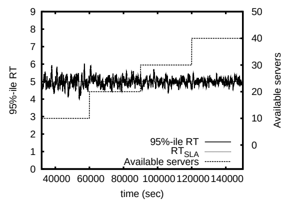


Fig. 22. Response time (homogeneous capacity increase)

the database tier (initially set to 10) grows by ten units every 30000 seconds. All the servers have an exponentially distributed service time whose average value is 1 sec. The average session arrival rate is set to 6 sessions/sec.

Figure 20 shows that the estimated rate limit properly follows the increment in the number of servers, and grows in proportion. This result is obtained because of the scaling procedure introduced in section 5.3 .

Analogous results are detailed in figure 21, showing how the number of active sessions in the system also grows proportionally to system capacity. Both figures 20 and 21 enlighten the rapid reactivity of the algorithm to the new system configuration. Figure 22 shows that although there is a significant runtime change in cluster architecture and in session acceptance rate, SLA requirements are always met. In fact, response time remains steadily close to the SLA limit without perceivable variations even when architectural capacity changes. This result is particularly important as it shows how capacity improvements can be made at run-time in a way that is completely transparent to the user. Note that the variance of response time decreases with time due to the increased rate of accepted sessions, leading to a more numerous sample set.

7.6.2 Cluster of heterogeneous servers

We now introduce experiments conducted on a cluster of heterogeneous servers for which we consider capacity variations at the database tier only. For simplicity we divide the available servers into two types: top- and low-quality servers. The capacity of a low-quality server is measured as 1 capacity unit, and corresponds to an average execution time of 1 second per request (at the database tier), while a top-quality server has two capacity units and a corresponding average execution time of 0.5 seconds.

Maintenance
In this subsection we consider the case of runtime maintenance, where added servers belong to the top type. The initial cluster configuration provides 20 servers, of which 10 are top- and 10 are low-quality servers (for a total of

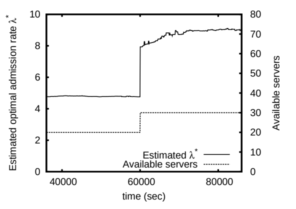


Fig. 23. Rate limit (capacity increase)

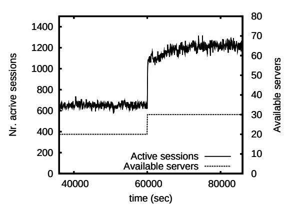


Fig. 24. Act. sessions (capacity increase)

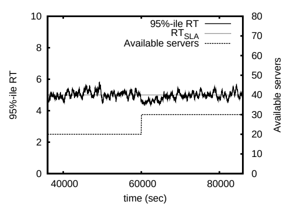


Fig. 25. Response time (capacity increase)

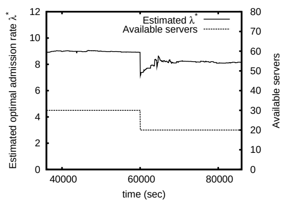


Fig. 26. Rate limit (low server fault)

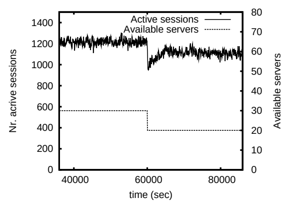


Fig. 27. Active sessions (low server fault)

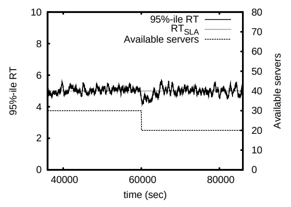


Fig. 28. Response time (low server fault)

30 capacity units). The maintenance intervention consists in the addition of 10 top-quality servers after 60000 seconds (leading to a total of 50 capacity units).

In this case the rough estimate of the new rate limit conducted with the scaling procedure causes a momentary underestimation of the system capacity, as seen in figure 23. After a short transient period learning activity causes the acquisition of more reliable information, thus allowing a rapid correction of the estimated rate limit. Figures 24 and 25 show how this momentary underestimation causes a short period in which the system is slightly underutilized with respect to the agreements.

Fault
In this subsection we consider the case of an architectural fault, where a group of servers suddenly becomes unavailable. We consider two scenarios, namely the fault of a group of top-quality servers and the fault of a group of low-quality servers. In both scenarios, the initial cluster configuration provides 30 servers of which 20 are top- and 10 are low-quality servers (for a total of 50 capacity units).

In the first scenario the fault causes the sudden unavailability of 10 low-quality servers after 60000 seconds (leading to a total of 40 capacity units). In this case the scaling procedure has effects similar to those described in the case of maintenance, and causes a momentary underestimation of the system capacity, as shown in figures 26, 27 and 28.

In the second scenario, the fault causes the sudden unavailability of 10 top quality servers after 60000 seconds (leading to a total of 30 capacity units).

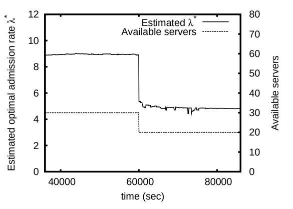


Fig. 29. Rate limit (top server fault)

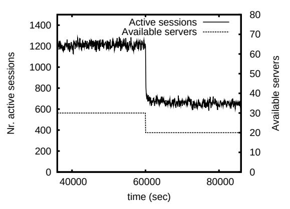


Fig. 30. Active sessions (top server fault)

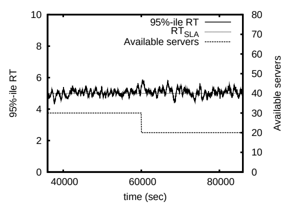


Fig. 31. Response time (top server fault)

In this case the scaling procedure produces a momentary overestimation of system capacity and a slight temporary violation of the agreements on quality for a very short transient period as depicted in figures 29, 30 and 31. Nevertheless, the self-learning capabilities of the proposed algorithm allow rapid correction of the estimated capacity and quick adaptation of the admission control policy to return the system to SLA constraints on response time.

8 Related Work

There is an impressively growing interest in autonomic computing and self-managing systems, starting from several industrial initiatives from IBM [3], Hewlett Packard [17] and Microsoft [18]. Although self-adaptation capabilities could dramatically improve web system reactivity and overload control during flash crowds, little effort has been devoted to the problem of autonomous tuning of QoS policies for web systems.

A great deal of research [1, 2, 10, 19–22] proposes non autonomic solutions to the problem of overload control in web server architectures.

The application of the autonomic computing paradigm to the problem of overload control in web systems poses some key problems concerning the design of a measurement framework. The authors of [23] propose a technique for learning dynamic patterns of web user behavior. A finite state machine representing typical user behavior is constructed on the basis of system past history, and is used for prediction and prefetching techniques. In reference [24], the problem of delay prediction is analyzed on the basis of a learning activity exploiting passive measurements of query executions. Such predictive capability is exploited to enhance traditional query optimizers.

The proposals presented in [23, 24] can partially contribute to improving the QoS of web systems. However, in contrast to our work, none of these directly formulates a complete autonomic solution that also shows how to take measurements and make corresponding admission control decisions for web cluster architectures.

The authors of [25] also address a very important decision problem in the design of the monitoring module: the timing of performance control. They propose to adapt the time interval between successive decisions to the size of workload dependent system parameters, such as the processor queue length. The dynamic adjustment of this interval is of primary importance for threshold based policies for which a constant time interval between decisions may lead to oscillatory behavior in high load scenarios, as we have shown in Section 7. Simulations reveal that our algorithm is not subject to oscillations and shows a very little dependence on the time interval between decisions.

The problem of designing adaptive component-level thresholds is analyzed in [26] in the general context of autonomic computing. The mechanism proposed in the paper consists in monitoring the threshold values in use by keeping track of false alarms with respect to possible violations of service level agreements. A regression model is used to fit the observed history. When a sufficiently confident fit is attained, the thresholds are calculated accordingly. On the contrary, if the required confidence is not attained, the thresholds are set to random values as if there were no history. A critical problem for this proposal is the fact that the most common threshold policies cause on/off behavior that often results in unacceptable performance. By contrast, our proposal is based on a probabilistic approach and a learning technique, dynamically creating a knowledge basis for the online evaluation of the best decision to be made even for traffic situations that never occurred in past history.

The problem of autonomously configuring a computing cluster to satisfy SLA requirements is addressed in [27]. This paper introduces a design strategy for autonomic computing that divides the problem into different phases, called *monitor*, *analyze*, *plan* and *execute* (MAPE, according to the terminology in use by IBM [28]) in order to meet SLA requirements in terms of response time and server utilization. Unlike our work, the authors of this paper designed a policy whose decisions concern the reconfiguration of resource allocation to services.

The design of SOC is inspired by the policies we introduced in our previous articles [7, 29]. Here we have added a change detection and a decision rate adaptation mechanism to manage flash crowds, as well as the scaling procedure to handle capacity variations.

9 Conclusion

In this paper we address the problem of overload control for web based systems. We introduce an original autonomic policy named SOC, characterized by self-configuration, self-adaptation and self-protection properties. SOC exploits a change detection mechanism in order to switch between two modes according to the time variability of incoming traffic.

When incoming traffic is stable the policy works in normal mode, in which performance controls are paced at a regular rate. The policy switches to flash crowd management mode as soon as a rapid surge in demand is detected. It then increases the rate of performance controls until incoming traffic becomes more stable. This permits a fast reaction to sudden changes in traffic intensity, and a high system responsiveness.

Our policy does not require any prior knowledge about incoming traffic, or any assumptions about the probability distribution of request inter-arrival and service times.

Unlike other proposals in the field, our policy works under a wide range of operating conditions without the need for laborious manual parameter tuning. It is implemented entirely at dispatching points, without requiring any modification of client and server software.

We have compared our policy to previously proposed approaches. Extensive simulations have shown that it permits an excellent utilization of system resources while always respecting the limits on response time imposed by service level agreements. We have shown that our policy reduces the oscillations in response time which are common to other policies. Simulation results also highlight the ability of our proposal to react properly to flash crowds and to capacity variations that may occur as a consequence of failures or runtime maintenance procedures. We have demonstrated how SOC rapidly adapts the admission probability in such situations so as to keep the overload under control.

References

[1] L. Cherkasova and P. Phaal, “Session based admission control: a mechanism for peak load management of commercial web sites,” *IEEE Transactions on Computers, 2002*, vol. 51, no. 6, 2002.

[2] J. Carlstrom and R. Rom, “Application aware admission control and scheduling in web servers,” *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2002.

[3] “Ibm: the vision of autonomic computing,” *http://www.research.ibm.com/autonomic/manifesto.*

[4] V. Cardellini, E. Casalicchio, and M. Colajanni, “The state of the art in locally distributed web server systems,” *ACM Computing Surveys*, vol. 34, no. 2, pp. 263–311, 2002.

[5] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," *IEEE Transactions on the Web*, vol. 1, no. 1, 2007.

[6] 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, March 2008.

[7] N. Bartolini, G. Bongiovanni, and S. Silvestri, "An autonomic admission control policy for distrbuted web systems," *The IEEE Proceedings of the International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunicaion Systems (MASCOTS)*, 2007.

[8] "Specweb2005 design document," *http://www.spec.org/web2005/docs/designdocument.html.*

[9] "The transaction processing council (tpc). tpc-w," *http://www.tpc.org/tpcw.*

[10] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel, "A method for transparent admission control and request scheduling in e-commerce web sites," *Proceedings of the ACM World Wide Web Conference(WWW)*, May 2004.

[11] N. Bartolini, G. Bongiovanni, and S. Silvestri, "Distributed server selection and admission control in replicated web systems," *The IEEE Proceedings of the 6th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2007.

[12] Z. Xu and G. v. Bochmann, "A probabilistic approach for admission control to web servers," *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, July 2004.

[13] J. Aweya, M. Ouelette, D. Y. Montuno, B. Doray, and K. Felske, "An adaptive load balancing scheme for web servers," *International Journal of Network Management*, vol. 12, pp. 3–39, 2002.

[14] "Opnet technologies inc." *http://www.opnet.com.*

[15] D. Menasce, "Tpc-w: A benchmark for e-commerce," *IEEE Internet Computing*, May/June 2002.

[16] H. Weinreich, H. Obendorf, E. Herder, and M. Mayer, "Off the beaten tracks: Exploring three aspects of web navigation," *Proceedings of the ACM World Wide Web Conference (WWW)*, 2006.

[17] "Hewlett packard: Adaptive enterprise design principles," *http://h71028.www7.hp.com/enterprise/cache/80425-0-0-0-121.html.*

[18] "Microsoft: The drive to self-managing dynamic systems," *http://www.microsoft.com/windowsserversystem/dsi/default.mspx.*

[19] A. Verma and S. Ghosal, "On admission control for profit maximization of networked service providers," *Proceedings of ACM World Wide Web (WWW)*, 2003.

[20] D. Henriksson, Y. Lu, and T. Abdelzaher, "Improved prediction for web server delay control," *Proceedings of the IEEE Euromicro Conference on Real-time systems (ECRTS)*, 2004.

[21] C. Lu, Y. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son, "Feedback control architecture and design methodology for service delay guarantees in web servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 9, pp. 1014–1027, 2006.

[22] X. Chen and J. Heidemann, "Flash crowd mitigation via adaptive admission control based on application-level observation," *ACM Transactions on Internet Technology*, vol. 5, no. 3, pp. 532–569, 2005.

[23] S-Pradeep, C. Ramachandran, and S. Srinivasa, "Towards autonomic websites based on learning automata," *Proceedings of the ACM World Wide Web Conference (WWW)*, 2005.

[24] J.-R. Gruser, L. Raschid, V. Zadorozhny, and T. Zhan, "Learning response time for websources using query feedback and application in query optimization," *The International Journal on Very Large Data Bases*, vol. 9, no. 1, March 2000.

[25] X. Liu, R. Zheng, J. Heo, Q. Wang, and L. Sha, "Timing performance control in web server systems utilizing server internal state information," *Proceedings of the IEEE Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services (ICAS/ICNS)*, 2005.

[26] D. Breitgand, E. Henis, and O. Shehory, "Automated and adaptive threshold setting: enabling technology for autonomy and self-management," *Proceedings of the International Conference on Autonomic Computing (ICAC)*, 2005.

[27] Y. Li, K. Sun, J. Qiu, and Y. Chen, "Self-reconfiguration of service-based systems: a case study for service level agreements and resource optimization," *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 2005.

[28] B. Miller, "The autonomic computing edge: The role of knowledge in autonomic systems," *http://www-128.ibm.com/developerworks/autonomic/library/ac-edge6/.*

[29] N. Bartolini, G. Bongiovanni, and S. Silvestri, "Self-* overload control for distributed web systems," *The IEEE Proceedings of the International Workshop on Quality of Service (IWQoS)*, 2008.