2004 IEEE Intl. Conf. on Autonomic Computing, New York, NY, May 17-18, 2004.

Assessing the Robustness of Self-Managing Computer Systems under Highly Variable Workloads

Mohamed N. Bennani and Daniel A. Menascé
Dept. of Computer Science, MS 4A5
George Mason University
4400 University Dr.
Fairfax, VA 22030
{mbennani,menasce}@cs.gmu.edu

Abstract

Computer systems are becoming extremely complex due to the large number and heterogeneity of their hardware and software components, the multi-layered architecture used in their design, and the unpredictable nature of their workloads. Thus, performance management becomes difficult and expensive when carried out by human beings. A new approach, called self-managing computer systems, is to build into the systems the mechanisms required to self-adjust configuration parameters so that the Quality of Service requirements of the system are constantly met. In this paper, we evaluate the robustness of such methods when the workload exhibits high variability in terms of the inter-arrival time and service times of requests. Another contribution of this paper is the assessment of the use of workload forecasting techniques in the design of QoS controllers.

1. Introduction

Computer systems are becoming extremely complex. Complexity stems from the large number and heterogeneity of a system's hardware and software components, from the multi-layered architecture used in the system's design, and from the unpredictable nature of the workloads, especially in Web-based systems [11]. Therefore, performance management of complex systems is difficult and expensive when carried out by human beings. A new approach, called self-managing computer systems, is to build into the systems the mechanisms required to self-adjust configuration parameters so that the Quality of Service (QoS) requirements of the system are constantly met. There has been a growing interest in self-managing systems as illustrated by the papers in a recent workshop [3] and in [2, 4, 5, 6, 7, 9, 10, 14, 15]. In this paper, we evaluate the robustness of the QoS controller we designed and described in [9] and expand its design. That approach combines analytic performance models with combinatorial search techniques to design controllers that run periodically (e.g., every few minutes) to determine the best possible configuration for a system given its workload.

An evaluation of the robustness of this method when the workload exhibits high variability in terms of the inter-arrival time and service times of requests is presented. The results indicate that the approach is robust for relatively high values of the coefficients of variation of the inter-arrival time and service time distributions.

As an extension of the controller described in [9], workload forecasting techniques were integrated into the controller to make it react to the expected workload as opposed to recently observed workload intensity. The results show that, at a 95% confidence level, the controller that uses workload forecasting is able to maintain significantly higher values of QoS at times when the workload intensity is rising towards its peak levels or reducing from its peak level.

The rest of this paper is organized as follows. As a background to the remaining sections, section two describes the basic approach to the design of self-management systems. Section three describes the QoS metric used by the controller for optimization purposes. Section four presents the experimental setting and the next section describes the experiments and results used to illustrate the robustness of the method with respect to highly varying inter-arrival times and highly variable service times. Section six discusses how the QoS controller uses workload forecasting. Finally, section seven presents some concluding remarks.

2. Controller Approach

The controller, discussed in greater detail in [9], is based on the notion that a computer system is enhanced with a QoS controller that i) monitors system performance, ii)

monitors the resource utilization of the various resources of the system, iii) executes, at regular intervals, called controller intervals (CI), a controller algorithm to determine the best configuration for the system (see Fig. 1). As a result of running the controller algorithm, reconfiguration commands are generated to instruct the system to change its configuration.

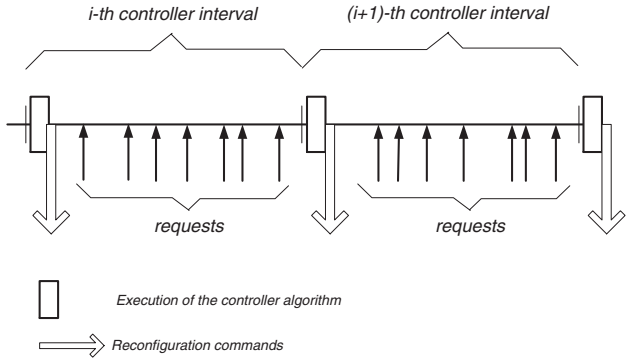


Figure 1. Controller intervals.

The architecture of the QoS controller is best described with the help of Fig. 2. The QoS controller has four main components: Service Demand Computation (2), Workload Analyzer (3), QoS Controller Algorithm (5), and Performance Model Solver (4). The Service Demand Computation (2) component collects utilization data (1) on all system resources (e.g., CPU and disks) as well as the count of completed requests (7), which allows the component to compute the throughput. The service demand of a request, i.e., the total average service time of a request at a resource, can be computed as the ratio between the resource utilization and the system throughput [12]. The service demands (8) computed by this component are used as input parameters to a Queuing Network (QN) model [12] of the computer system solved by the Performance Model Solver component.

The Workload Analyzer (3) component analyzes the stream of arriving requests (6) and computes statistics for the workload intensity, such as average arrival rate, and uses statistical techniques [1] to forecast the intensity of the workload in the next controller interval. The current or predicted workload intensity values (9) computed by this component are also used as input parameters of the Queuing Network model solved by the Performance Model Solver component (4). This component receives requests (10) from the QoS Controller Algorithm to solve the QN model corresponding to a specific configuration of the system. This component takes as input parameters to the QN model the

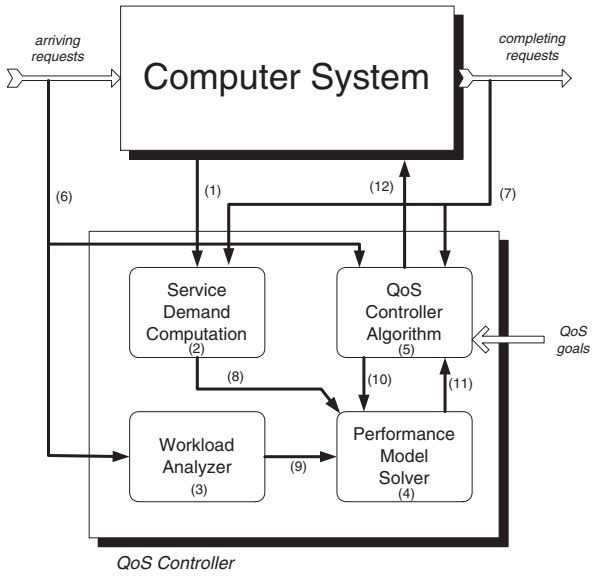


Figure 2. Architecture of the QoS Controller.

configuration parameter values (10), service demand values (8), and workload intensity values (9). The output of the QN model is the resulting QoS value (11) for the configuration used as input by the QoS Controller algorithm. At the beginning of each controller interval (see Fig. 1), the QoS Controller Algorithm (5) component runs the controller algorithm. This algorithm takes into account the desired QoS goals, the arrival and departure processes, and performs a combinatorial search (e.g., beam search or hill-climbing) [13] of the state space of possible configuration points in order to find a close-to-optimal configuration. The cost function associated to each point in the space of configuration points is the QoS value of the configuration described in section 3. This QoS value has to be computed by the Performance Model Solver for each point in the space of configuration points examined by the QoS controller algorithm. Once the QoS controller determines the best configuration for the workload intensity levels provided by the Workload Analyzer, it sends reconfiguration commands (12) to the computer system.

3. Computing QoS Values

The QoS metric, QoS, computed at the end of each controller interval is defined as $QoS = w_R \times \Delta QoS_R + w_X \times \Delta QoS_X + w_P \times \Delta QoS_P$, where ΔQoS_R, ΔQoS_X, and ΔQoS_P are relative deviations of the average response time, average throughput, and probability of rejection, with respect to their desired goals, and w_R, w_X, and w_P are the relative weights of these deviations with respect to the QoS

value.

The relative deviation ΔQoS_R is defined as

$$\Delta QoS_R = \frac{R_{\max} - R_{\text{measured}}}{\max(R_{\max}, R_{\text{measured}})} \quad (1)$$

where $R_{\max}$ is the maximum average response time tolerated and R_{measured} is the measured response time.

The relative deviation ΔQoS_X is defined as

$$\Delta QoS_X = \frac{X_{\text{measured}} - X^*_{\min}}{\max(X_{\text{measured}}, X^*_{\min})} \quad (2)$$

where $X^*_{\min} = \min(\lambda, X_{\min})$ is the minimum value between the arrival rate λ and the minimum required throughput $X_{\min}$. $X^*_{\min}$ is used as the Service Level Agreement (SLA) instead of $X_{\min}$ in Eq. (2) because it would not make sense to expect a system to meet a given minimum throughput requirement if the workload intensity is not large enough to drive the system to that throughput level.

The relative deviation ΔQoS_P is defined as

$$\Delta QoS_P = \frac{P_{\max} - P_{\text{measured}}}{\max(P_{\max}, P_{\text{measured}})} \quad (3)$$

where $P_{\max}$ is the maximum probability of rejection tolerated and P_{measured} is the measured probability of rejection.

The deviations in Eqs. (1)-(3) are defined in such a way that i) the deviation is a dimensionless number in the interval (-1,1), ii) the deviation is zero when the SLA is exactly met, negative when the SLA is violated, and positive when the SLA is exceeded.

4. The Experimental Setting

Our experiments simulate a computer system that consists of a multi-threaded server. The server has m threads and a maximum system size (i.e., total number of requests in the system, waiting or using a thread) equal to $n (n > m)$. Arriving requests that find n requests in the system are rejected. When a thread is serving a request it will use physical resources (e.g., CPU and disk). Therefore, the response time of a request can be broken down into the waiting time for a thread (software contention), waiting times for physical resources, and service times at physical resources. The configurable parameters n and m are adjusted dynamically at the end of every controller interval (2 minutes), so that the QoS is maximized. The SLA values used for all experiments, except for those on SLA sensitivity, are: $R_{\max} = 1.2$ seconds, $X_{\min} = 5$ requests/sec, and $P_{\max} = 0.05$.

The initial values for n and m are $n = 7$ and $m = 2$ for the experiments reported in section 5, and $n = 30$ and $m = 10$ for the experiments of section 6. Different initial configurations were used in these sections because different workloads were used. For the same reason, different weights for the SLAs were used for the experiments reported in section 5 ($w_R = 0.25$, $w_X = 0.30$, and $w_P = 0.45$) and for the experiments of section 6 ($w_R = 0.35$, $w_X = 0.25$, $w_P = 0.40$). In this case, we wanted to give a higher and a smaller importance to the response time and the throughput, respectively, as the workload intensity exceeds, at times, the maximum theoretical value of 20 req/sec.

Also, in section 6, we only used beam search as the heuristic search technique because the curves in section 5 indicate that there is no statistically significant difference at the 95% level between using beam search and hill climbing. CSIM's library (www.mesquite.com) was used for simulating the multithreaded server and IMSL's library (www.vni.com) was used for the polynomial regression models needed for the forecasting experiments.

5. Highly Variable Interrarival and Service Times

Many real workloads exhibit some sort of high variability in their intensity and/or service demands at the different resources. Therefore, it is very important to investigate the behavior of the proposed technique for self-managing computer systems in such environments. To this end, we conducted a set of experiments to study the impact of the variability in the request inter-arrival and service times distributions at both system resources (i.e., cpu and disk). The variability of these distributions is represented by their respective coefficients of variation (COV) (i.e., the standard deviation divided by the mean): C_a and C_s. We used the values 1.0, 2.0, and 4.0 for C_a and C_s for a total of 9 combinations of the values of these two coefficients of variation.

5.1. Generating Distributions with Varying Coefficients of Variation

We used the exponential distribution for a COV equal to 1. To synthesize a distribution with a given mean, μ, and a COV > 1, we used a 2-stage Coxian distribution [8], where each stage is exponentially distributed with an average equal to μ_i for i = 1, 2 (see Fig. 3). As shown in the figure, one moves from stage 1 to 2 with probability (1 - q). One can exit the server right from stage 1 with probability q ($0 < q < 1$). The average time spent in this server is equal to

$$\mu = \mu_1 + (1 - q)\mu_2. \quad (4)$$

The variance is given by

$$\sigma^2 = \mu_1^2 + (1 - q^2)\mu_2^2. \quad (5)$$

Therefore, the COV is given by:

$$COV = \frac{\sigma}{\mu} = \frac{\sqrt{\mu_1^2 + (1 - q^2)\mu_2^2}}{\mu_1 + (1 - q)\mu_2}. \quad (6)$$

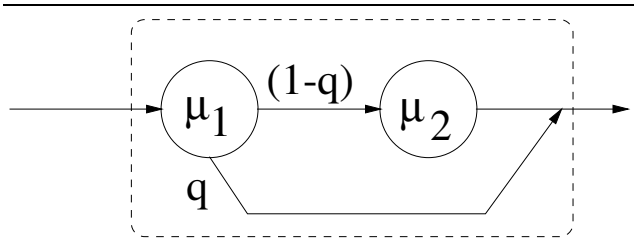


Figure 3. Two-phase Coxian distribution.

The question becomes, then, how to choose μ_1, μ_2, and q to obtain a distribution with given values μ and COV for the mean and coefficient of variation. We start by using Eq. (4) to write μ_1 in terms of μ_2 and q: $\mu_1 = \mu - (1-q) \times \mu_2$. By replacing this expression for μ_1 in Eq. (6) one obtains the following quadratic equation on the unknown μ_2:

$$2(1-q)\mu_2^2 - 2\mu(1-q)\mu_2 + (1 - COV^2)\mu^2 = 0. \quad (7)$$

We can now solve Eq. (7) for q varying from 0.1 to 0.95 in increments of 0.05 and choose one of the values of q that results in a positive value for μ_1 and μ_2.

5.2. Results

Figure 4 depicts the variation of the workload intensity, measured in requests/sec, as a function of time, measured in controller interval units for the experiments related to the variability of the workload. The duration of each experiment was 30 CIs (i.e., 60 minutes since each CI was set to 2 minutes). The mean service demands at the cpu and the disk were 0.03 seconds and 0.05 seconds, respectively. Thus, the maximum theoretical arrival rate supported by the system is 20 req/sec (i.e., 1 / max [0.03, 0.05]) [12].

The average arrival rate starts at a low value of 5 req/sec and reaches a peak of 19 req/sec, close to the theoretical maximum, at CI = 19. The workload intensity stays at this level for three CIs and then starts to decreases towards 14 req/sec. Ten experiments were run for each combination of C_a and C_s and 95% confidence intervals for the average of the QoS value were computed at the end of each CI. Results were obtained for three scenarios: one in which the controller is disabled and two others with the QoS controller active. The two results in which the controller is active differ in the combinatorial optimization technique used by the controller: beam search and hill-climbing. Figure 5 shows results obtained in our previous study [9] for the case of exponentially distributed inter-arrival and service times ($C_a = C_s = 1.0$). The controlled system maintains much

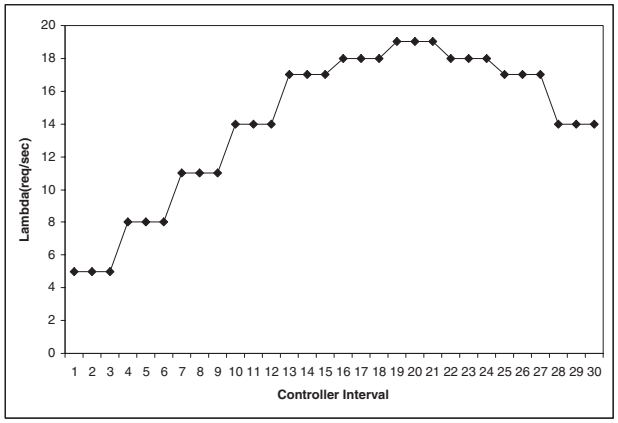


Figure 4. Workload intensity variation for the high variability experiments.

higher QoS values than the non-controlled system even at high peak loads.

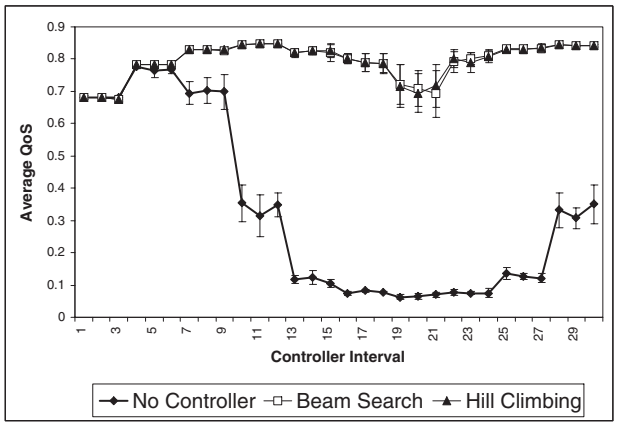
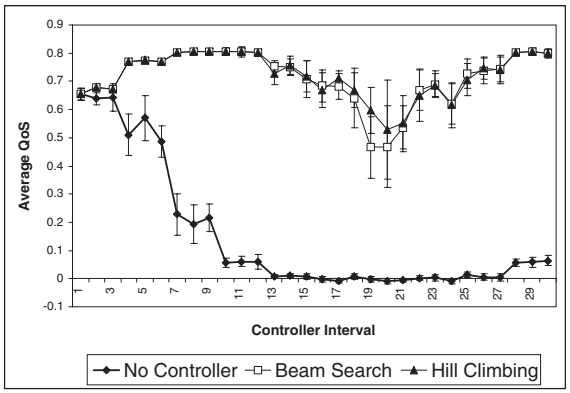
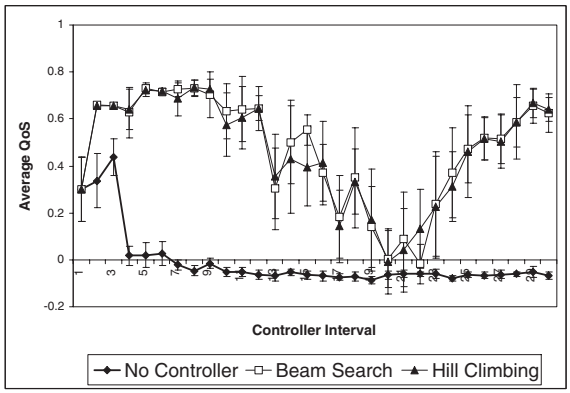


Figure 5. QoS Controller Performance for $C_a = C_s = 1.0$.

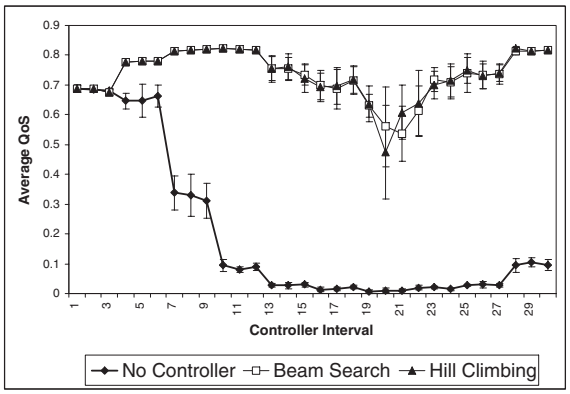
Figures 6 shows the results for all the scenarios in which either the inter-arrival time or the service time or both are not exponentially distributed. First, it should be noted that the controlled system always exhibits higher QoS values than the non-controlled (NC) system. Also, as expected, confidence intervals become wider as either or both COV increase. But, confidence intervals for the controlled system tend to be wider than those for the non-controlled system (NC) because the system itself is varying due to dy-



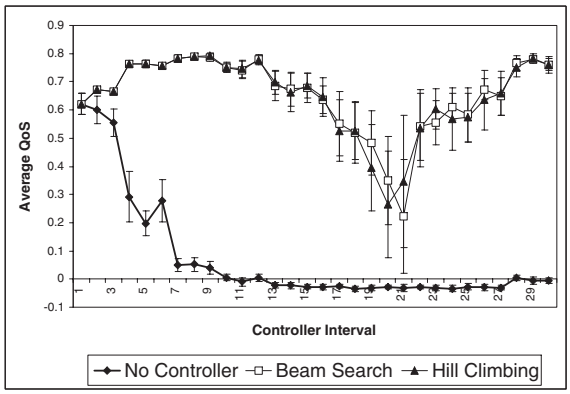
(a) $C_a = 1.0$ and $C_s = 2.0$



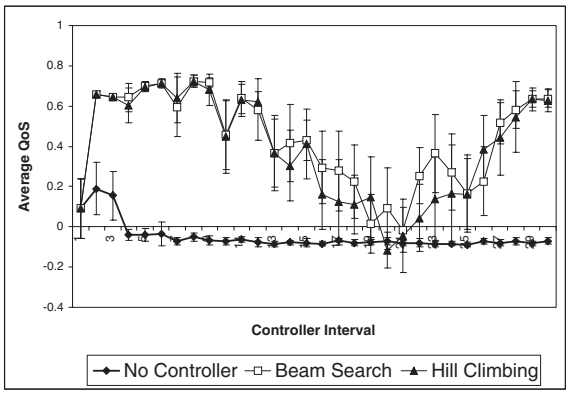
(b) $C_a = 1.0$ and $C_s = 4.0$



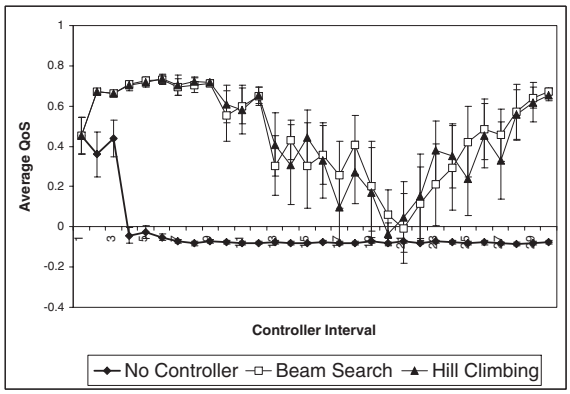
(c) $C_a = 2.0$ and $C_s = 1.0$



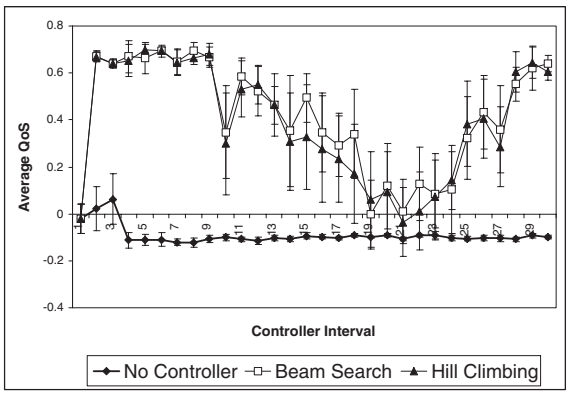
(d) $C_a = 2.0$ and $C_s = 2.0$



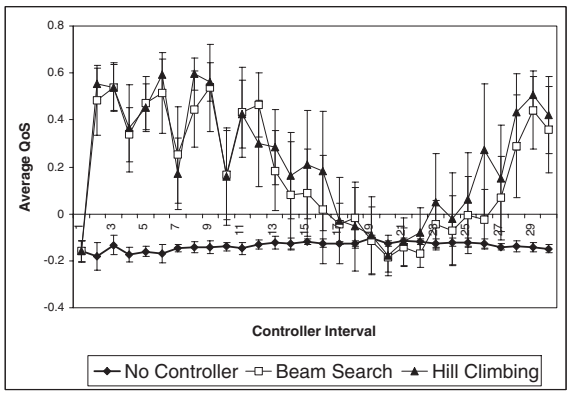
(e) $C_a = 2.0$ and $C_s = 4.0$



(f) $C_a = 4.0$ and $C_s = 1.0$



(g) $C_a = 4.0$ and $C_s = 2.0$



(h) $C_a = 4.0$ and $C_s = 4.0$

Figure 6 - QoS Controller Performance vs. C_a and C_s

namic adjustment of parameters. Another clear observation is that as the variability increases, the performance of the NC system starts to deviate from that of the controlled systems at an earlier stage. For example, when $C_a = 1.0$ and $C_s = 1.0$ (Fig. 5), the difference in QoS starts at CI = 7 ($\lambda = 11$) req/sec. As C_s increases for the same value of C_a, the difference between the two cases becomes apparent at CI = 4 ($\lambda = 8$), and CI = 2 ($\lambda = 5$) (see Figs. 6 (a)-(b)).

Let us now examine the effect of the variation of C_s for a fixed value of C_a. For $C_a = 1.0$ and $C_s = 1.0$ (Fig. 5), the controlled system keeps the QoS value higher than 0.7 throughout the experiments while the NC system exhibits a marked drop in QoS (to about 0.1) when λ reaches its peak value. For this value of C_s the QoS for the NC case is still positive. When C_s increases to 2.0 (Fig. 6 (a)), the QoS for the controlled case drops to about 0.45 at the peak value of λ and the QoS for the non-controlled case goes to zero for most of the experiment ($13 \leq$ CI ≤ 27). For $C_s = 4.0$ (Fig. 6 (b)), a high value of the service time COV, the NC case exhibits a negative QoS for most of the experiment while the controlled system only gets slightly lower than zero at peak load and then recovers. The NC system remains in negative territory.

We now examine the variation of the QoS as C_a varies for a fixed value of C_s. For $C_s = 1.0$ and $C_a = 1.0$ and 2.0 (Figs. 5 and 6 (a), respectively), the NC system exhibits marked drops in the QoS value as soon as λ starts to increase but still remains in positive territory. The controlled system maintains a high QoS value at peak load even for $C_a = 2.0$. For example, in this case, the average QoS value at peak load is 0.45 for the controlled system while it is very close to zero for the NC system. When $C_s = 1.0$ and $C_a = 4.0$ (Fig. 6 (b)), the NC system displays a negative QoS throughout most of the experiment (from CI = 4 onwards). The controlled system only gets slightly lower than zero at peak load.

At extreme cases, where both C_a and C_s are very high (i.e., equal to 4.0 as shown in Fig. 6 (h)), the NC system has a negative QoS value throughout the entire experiment. The controlled system reaches some negative points at peak load but recovers when the load decreases.

In order to explore the sensitivity of the controller to the space of SLA values, we ran experiments for $C_a = C_s = 2.0$ for stricter and more relaxed SLA values than the ones used in Fig. 6. Figure 7 illustrates the relative variation φ of the QoS with respect to the base value QoS_{base} shown in Fig. 6. The value of φ was defined as

$$\varphi = \frac{QoS - QoS_{\text{base}}}{\mid QoS_{\text{base}} \mid}. \quad (8)$$

The values for the more relaxed and stricter SLAs are: $R_{\max} = 1.5$ seconds, $X_{\min} = 4$ requests/sec, $P_{\max} = 0.1$; and $R_{\max} = 1.0$ seconds, $X_{\min} = 7$ requests/sec, $P_{\max} = 0.03$, respectively. As the figure indicates, the controlled system is much less sensitive to variations in the SLA values than the NC system.

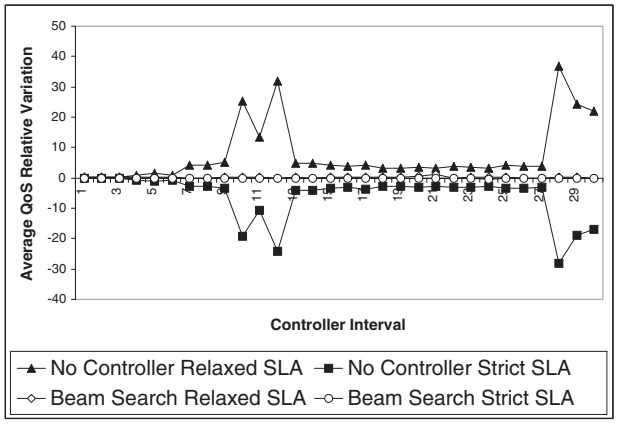


Figure 7. Effect of stricter and more relaxed SLAs on the controller performance.

6. The Workload Forecasting Algorithm

In the self-managing computer systems that we proposed in [9], the QoS optimizer module uses the average arrival rate of requests obtained in the previous controller interval, CI, as an estimate of the expected workload intensity for the next CI. This value is then used by the performance model to compute the QoS value for a given set of configuration parameters. The drawback of such an approach is that it overlooks any increasing or decreasing trends in the workload for the past CI. This could result, consequently, in a very inaccurate estimate of the next expected arrival rate and an inappropriate choice of configuration values.

To overcome this shortcoming, we added a module responsible for short-term workload forecasting. This module keeps a sliding window of N values for the last average arrival rates observed for the last N small sub-intervals. Each of these sub-intervals is of length Δ seconds. N and Δ are chosen so that $N \times \Delta$ does not exceed the length of a controller interval (2 minutes in our case).

Many techniques can be used for short-term forecasting. However, no particular technique gives good forecasting results for all kind of data. Therefore, the forecasting module uses three techniques: exponential smoothing, weighted moving averages, and polynomial regression [11].

Exponential smoothing was included because it is known to be good for making predictions from time series data that exhibit upwards and/or downwards trends. Exponential smoothing computes a prediction as follows: Predicted-

Value = $\alpha\times$ PreviousActualValue + $(1-\alpha)\times$ PreviousPredictedValue. We used $\alpha = 0.6$.

There are times when the workload maintains an almost constant intensity for quite a while before changing significantly. Weighted moving averages is an appropriate technique for these situations. In our experiments, we compute the forecasted value based on the three most recent average arrival rates in the sliding window. The chosen weights give more importance to the newest values. Hence, the forecasted value is given by: ForecastValue = (0.45 $\times$ LatestEntryInSlidingWindow + 0.35 $\times$ SecondLatestEntryInSlidingWindow + 0.25 $\times$ ThirdLatestEntryInSlidingWindow).

The third forecasting technique, polynomial regression, was chosen as polynomials have the ability of approximating fairly well any continuous function. The higher the degree of the polynomial the better is the fitting. However, in order not to introduce a severe overhead on the controller when computing the regression model, we used a moderately high value for polynomial degree: six.

All three models are rebuilt each time a new average arrival rate entry is inserted into the sliding window. At this time, we compute what would be the forecasted value according to each of the three models. We also compute the R^2 value, based on the method of least squares errors, for each of these models to assess the quality of the fits. At this stage, the forecasting module returns the forecasted value provided by the model with the highest R^2 value. There is an exception to this rule, however. In the case of a downward trend in the workload intensity, the polynomial regression model may forecast a negative value for the expected arrival rate. In such a case, even though the polynomial regression model might produce the highest R^2 value, the forecasting module returns the forecasted value that comes from the model with the second highest R^2 value, instead.

6.1. Results

Figure 8 compares the expected arrival rate at every controller interval, when the forecasting module was enabled/disabled, to the actual measured arrival rate. Note that in this figure we start from the 2nd controller interval as it is only at this time that data is available in the sliding window so that forecasting can be carried out. The actual workload has two peaks at 30 req/sec at CI = 8 and CI = 24. The curve for the expected arrival rate when forecasting is not used is simply a one-time unit shift to the right of the curve of the measured arrival rate. When the forecasting module is enabled, the system succeeded in finding quite close estimates of the arrival rate when that was possible at all. The largest gaps between the forecasted and the measured arrival rates happened at the 10-th and 11-th controller intervals. At these points, the forecasted values were 41.27 req/sec and 43.28 req/sec, whereas the measured workload intensities were 30 req/sec and 26.03 req/sec, respectively. However, since for both of these cases, the measured arrival rates for the immediate previous controller intervals (9 and 10) were 30 req/sec, these gaps did not significantly impact the QoS. This is due to the fact that 30 req/sec exceeds by far system's maximum throughput (20 req/sec). Therefore, the system configuration was already set at its minimum size.

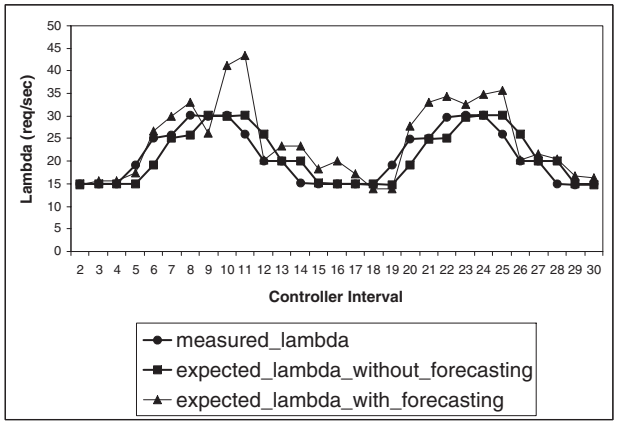


Figure 8. Workload intensity variation for the workload forecasting experiments.

From this figure, we also observe that at controller intervals 6, 8, 20, and 22, the values for the measured arrival rates were 25, 30, 24.92, and 29.74 req/sec, respectively. The expected values for the arrival rates at these same CIs when forecasting was not used were significantly smaller (20, 25, 20, and 25 req/sec, respectively). Whereas the corresponding values, when forecasting was used, were 26.66, 32.96, 27.71, and 34.45 req/sec, respectively. As a result, the QoS values, at these CIs, were significantly higher when forecasting was used than when it was not. This is illustrated in Fig. 9.

Figure 9 shows the results of the average QoS obtained for 10 runs of the simulation when the forecasting module was enabled and when it was disabled along with the 95% confidence intervals for the average QoS. We can see from this figure that the average QoS obtained when forecasting is enabled is statistically better for exactly 8 out of the 30 controller intervals. For the other controller intervals the 95% confidence intervals overlap and therefore no conclusion can be reached at. These eight controller intervals are: CI = 6 (λ = 25 req/sec), CI = 7 (λ = 26 req/sec), CI = 8 (λ = 30 req/sec), CI = 12 (λ = 20 req/sec), CI = 20 (λ = 24.92 req/sec), CI = 21 (λ = 25.2 req/sec), CI =

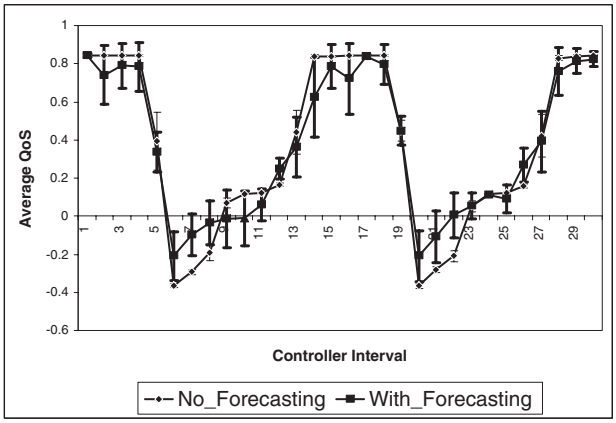


Figure 9. Impact of workload forecasting.

22 (λ = 29.74 req/sec), and CI = 26 (λ = 20 req/sec). For most of these controller intervals, the QoS is negative. However, when forecasting is enabled, the QoS values are significantly higher than otherwise. For example, at the 6-th controller interval, the average QoS when forecasting is not used is -0.36 whereas it is only -0.20 if forecasting is used. This is an improvement of about 44%. The forecasting module was able to notice that λ went up from 14.91 req/sec at CI = 4 to 19.12 req/sec at CI =5 and therefore predicted a value of 26.66 req/sec for CI = 6. The actual measured value of λ was 25 req/sec.

Another scenario that shows the importance of the added forecasting module is the measured QoS at the 26-th controller interval ($\lambda = 20$ req/sec). The measured QoS is 0.27 when forecasting is enabled and only 0.16 when it is disabled. This is an improvement of about 69%. The forecasting module noticed that λ went down from 30 req/sec at CI = 24 to 26 req/sec at CI =25 and predicted a value of 20 req/sec for CI = 26. The actual measured value of λ for CI = 26 is exactly 20 req/sec.

7. Concluding Remarks

The experiments reported in this paper clearly show the robustness of analytic models when used for QoS control. Even though these models assume exponential service and inter-arrival times (i.e., $C_s = 1.0$ and $C_a = 1.0$), they do a good job at predicting the trends of QoS metrics when these assumptions are violated. In our case, it is more important to correctly compare, QoS-wise, two points in the search space rather than knowing their absolute QoS values. The results in the paper also show that the use of workload forecasting can improve the QoS of a controlled system especially when the workload intensity is getting close to its saturation value. It was also shown that the controlled system is much less sensitive to the values of the SLAs than the non-controlled one.

References

[1] B. Abraham, J. Leodolter, and J. Ledolter, *Statistical Methods for Forecasting*, John Wiley & Sons, 1983.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, A. Veitch, "Hippodrome: running circles around system administration," *Proc. Conf. File and Storage Technologies (FAST'02)*, Monterey, CA, Jan. 2002

[3] J. Chase, M. Goldszmidt, and J. Kephart. eds., *Proc. First ACM Workshop on Algorithms and Architectures for Self-Managing Systems*, San Diego, CA, June 11, 2003.

[4] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle, "Managing Energy and Server Resources in Hosting Centers," *Proc. 18th Symp. Operating Systems Principles*, Oct. 2001.

[5] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics With Application to the Apache Web Server," *Proc. IEEE/IFIP Network Operations and Management Symp.*, Florence, Italy, April 15-19, 2002.

[6] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat, "Model-Based Resource Provisioning in a Web Service Utility," *Proc. Fourth USENIX Symposium on Internet Technologies and Systems*, March 2003.

[7] D. Garlan, S. Cheng, and B. Schmerl, "Increasing System Dependability through Architecture-based Self-repair," *Architecting Dependable Systems*, R. de Lemos, C. Gacek, A. Romanovsky (eds.), Springer-Verlag, 2003.

[8] L. Kleinrock, *Queuing Systems, Volume I: Theory*, Wiley-Interscience, NY, 1975.

[9] D.A. Menascé and M. Bennani, "On the Use of Performance Models to Design Self-Managing Computer Systems," *Proc. 2003 Computer Measurement Group Conf.*, Dallas, TX, Dec. 7-12, 2003.

[10] D.A. Menascé, R. Dodge, and D. Barbará, "Preserving QoS of E-commerce Sites through Self-Tuning: A Performance Model Approach," *Proc. 2001 ACM Conf. E-commerce*, Tampa, FL, Oct. 14-17, 2001.

[11] D.A. Menascé and V.A.F. Almeida, *Capacity Planning for Web Services: metrics, models, and methods,* Prentice Hall, PTR, 2002.

[12] D.A. Menascé, V.A.F. Almeida, and L.W. Dowdy, *Capacity Planning and Performance Modeling: from mainframes to client-server systems*, Prentice Hall, 1994.

[13] V.J. Rayward-Smith, I.H. Osman, C.R. Reeves, eds., *Modern Heuristic Search Methods,* John Wiley & Sons, Dec. 1996.

[14] F. Schintke, T. Schutt, A. Reinefeld, "A Framework for Self-Optimizing Grids Using P2P Components," *Proc. Intl. Workshop on Autonomic Computing Systems*, Sept. 2003.

[15] R. Wickremisinghe, J. Vitter, and J. Chase, "Distributed Computing with Load-Managed Active Storage," *Proc. IEEE Int. Symp. High Performance Distr. Computing*, July 2002.