© 2003 Menascé and Bennani. All Rights Reserved.
In Proc. 2003 Computer Measurement Group Conf., Dec. 7-12, 2003, Dallas, TX.

ON THE USE OF PERFORMANCE MODELS TO DESIGN SELF-MANAGING COMPUTER SYSTEMS

Daniel A. Menascé Mohamed N. Bennani
Department of Computer Science, MS 4A5
George Mason University
Fairfax, VA 22030-4444
{menasce,mbennani}@cs.gmu.edu

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. Because of these reasons, 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 requirements of the system are constantly met. In this paper, we describe an approach in which analytic performance models are combined with combinatorial search techniques to design controllers that run periodically (e.g., every few minutes) to determine the best possible configuration for the system given its workload. We first illustrate and motivate the ideas using a simulated multithreaded server. Then, we provide experimental results, obtained by using the techniques described here, to an actual Web server subject to a workload generated by SURGE.

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. Because of these reasons, 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 [CGK 2003] and in [MDB 2001, Menascé 2003, DGHPT 2002]. In this paper, we describe an approach in which analytic performance models are combined with combinatorial search techniques to design controllers that run periodically (e.g., every few minutes) to determine the best possible configuration for the system given its workload. We first illustrate and motivate the ideas using a simulated multithreaded server. Then, we provide experimental results obtained by using the techniques described here in an actual Web server subject to a workload generated by the Scalable URL Reference Generator (SURGE) [BC 1998].

The rest of this paper is organized as follows. 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 describes the multithreaded server system used to illustrate the approach as well as an analytic model for the system. The next section describes the simulation experiments used to illustrate the efficiency of the method under various circumstances. Section six describes the results of using our methods to an actual Web server. Finally, section seven presents some concluding remarks.

2. Basic Approach

Our approach to self-managing systems 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

© 2003 Menascé and Bennani. All Rights Reserved.

intervals, called *controller intervals*, a controller algorithm to determine the best configuration for the system (see Fig. 2.1). As a result of running the controller algorithm, reconfiguration commands are generated to instruct the computer system to change its configuration.

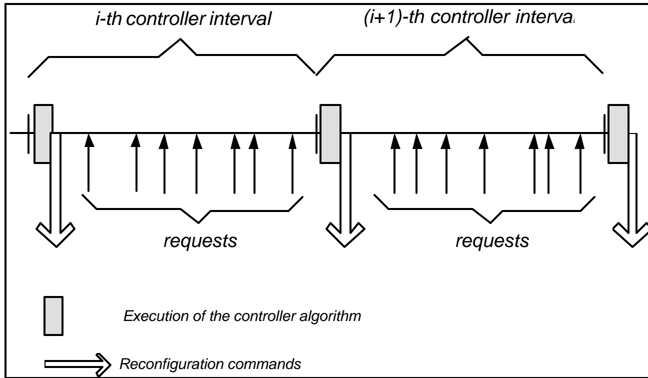

Figure 2.1 – Controller intervals.

The architecture of the QoS controller is best described with the help of Figure 2.2. 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 [MAD 1994]. The service demands computed by this component (8) are used as input parameters of a Queuing Model (QN) of the computer system solved by the Performance Model Solver component (see below).

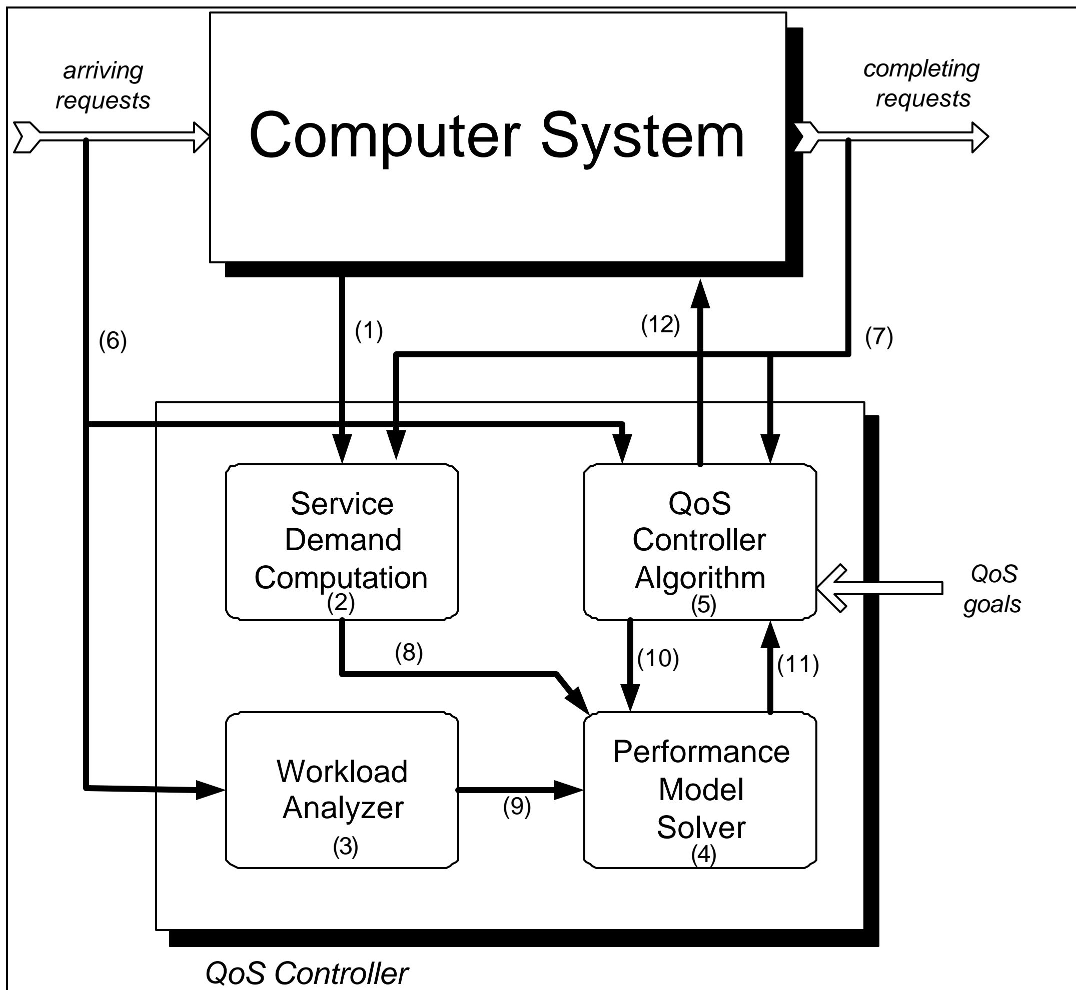

Figure 2.2 – Architecture of a QoS Controller

© 2003 Menascé and Bennani. All Rights Reserved.

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 [ALL 1983] 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 Model (QN) solved by the Performance Model Solver component.

The Performance Model Solver (4) 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 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.

The QoS Controller Algorithm (5) component runs the controller algorithm at the beginning of each controller interval (see Figure 2.1). The 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) [ROR 1996] 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. A formula to compute the QoS value is 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 the QoS Value

The QoS controller attempts, at each controller interval, to optimize the QoS metric, QoS, defined in this section. This metric combines relative deviations of the average response time, average throughput, and probability of rejection, with respect to their desired goals. More specifically, the relative deviation ΔQoS_R of the average response time is defined as

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

where $R_{\max}$ is the maximum average response time tolerated and $R_{measured}$ is the measured response time. The definition of Eq. (1) has the following properties:

- $\Delta QoS_R = 0$ if the response time exactly meets its SLA (i.e., $R_{measured} = R_{\max}$).
- $\Delta QoS_R > 0$ if the response time exceeds its SLA (i.e., $R_{measured} < R_{\max}$). Given that the measured response time $R_{measured}$ is at least equal to the sum $\sum_{i=1}^{K} D_i$ of the service demands D_i for all *K* resources (see [MAD94]), then, using Eq. (1), it follows that $\Delta QoS_R \leq 1 - (\sum_{i=1}^{K} D_i) / R_{\max} < 1$.
- $\Delta QoS_R < 0$ if the response time does not meet its SLA (i.e., $R_{measured} > R_{\max}$). Then, from Eq. (1) it follows that $-1 < -(1 - R_{\max} / R_{measured}) \leq \Delta QoS_R$

An intuitive interpretation of the definition in Eq. (1) is that of a relative gain (or loss) with respect to the SLA (or to the measured response time) . For example, if the measured response time is 3 seconds and the maximum response time is 4 seconds, then ΔQoS_R = (4-3)/4= 0.25. So, there is a gain in response time of 25% relative to its SLA. If the maximum response time is 3 seconds and the measured response time is 4 seconds, then ΔQoS_R = (3-4)/4= - 0.25. So, there is a 25% loss (i.e., a negative gain) with respect to the measured response time. In other words, it would be necessary to cut down 25% of the measured response time to meet the SLA.

The relative deviation ΔQoS_P of the probability of rejection is defined similarly to ΔQoS_R. Namely,

$$\Delta QoS_P = \frac{P_{\max} - P_{measured}}{\max(P_{\max}, P_{measured})} \qquad (2)$$

where $P_{\max}$ is the maximum probability of rejection tolerated and $P_{measured}$ is the measured probability of rejection. The definition of Eq. (2) has the following properties:

- $\Delta QoS_P = 0$ if the probability of rejection exactly meets its SLA (i.e., $P_{measured} = P_{\max}$).
- $0 < \Delta QoS_P \leq 1$ if the probability of rejection exceeds its SLA (i.e., $P_{measured} < P_{\max}$).
- $-1 \leq \Delta QoS_P < 0$ if the probability of rejection does not meet its SLA (i.e., $P_{measured} > P_{\max}$).

© 2003 Menascé and Bennani. All Rights Reserved.

The relative deviation of the average throughput is defined as

$$\Delta QoS_X = \frac{X_{measured} - X^*_{\min}}{\max(X_{measured}, X^*_{\min})} \qquad (3)$$

where $X_{measured}$ is the measured throughput, $X^*_{\min} = \min(\boldsymbol{l}, X_{\min})$ is the minimum value between the arrival rate $\boldsymbol{l}$ and the minimum required throughput $X_{\min}$. The reason for using $X^*_{\min}$ as the SLA as opposed to $X_{\min}$ in Eq. (3) is that 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 level of throughput. The definition of Eq. (3) has the following properties:

- $\Delta QoS_X = 0$ if the throughput meets its SLA (i.e., $X_{measured} = X^*_{\min}$).
- $0 < \Delta QoS_X < 1$ if the throughput exceeds its SLA (i.e., $X_{measured} > X^*_{\min}$).
- $-1 < \Delta QoS_X < 0$ if the throughput does not meet its SLA (i.e., $X_{measured} < X^*_{\min}$).

We can now define the single metric QoS as a weighted sum of the three QoS deviations defined above. Thus,

$$QoS = w_R \times \Delta QoS_R + w_P \times \Delta QoS_P + w_X \times \Delta QoS_X$$

where w_R, w_P, and w_X are weights, in the interval [0,1], determined by management, to indicate the relative importance of response time, throughput, and probability of rejection. Note that the value of QoS is a dimensionless number between –1 and 1. If all three metrics meet or exceed their SLAs, $QoS \geq 0$. If QoS < 0, then at least one of the metrics does not meet its SLA.

4. A Multithreaded Server Example

We describe in this section an example of a computer system that we will use to illustrate the ideas described in section 2.

4.1 System Description

The computer system shown in Figure 4.1 consists of a multithreaded server that receives requests at a rate of λ requests/sec. The system has *m* threads and the maximum number of requests that can be in the system either waiting for a thread or using a thread is equal to *n* (*n* > *m*). Thus, requests that arrive and find *n* requests in the system are rejected. When a thread is executing a request, it is using physical resources such as the CPU and disk. So, the response time of a request can be broken down into the following components: waiting for a thread (i.e., software contention), waiting for a physical resource (e.g., CPU or disk), and using a physical resource.

4.2 Analytic Model

The analytic model used to obtain the average response time, *R*, average throughput, *X*, and probability that requests are rejected, P_{rej}, is described here. The model consists of a combination of a Markov Chain [KLEI 1975] and a queuing network (QN) model. The Markov Chain is used to model the waiting queue for threads and the set of *m* execution threads. The QN model is used to obtain the rate at which threads complete their execution.

Let *X (k), k* = 1, …, *m* be the rate at which a thread completes its execution when there are *k* requests in execution. This rate can be obtained by solving the QN model for the hardware subsystem composed of the CPU and disk when there are *k* concurrent threads in execution. The solution of this queuing network can be obtained through Mean Value Analysis (MVA) [MA 2002, MA 2000, MAD 1994].

The Markov chain model of Figure 4.2 has *n* + 1 states. A state *k* (*k* = 0, …, *n*) represents the number of requests in the system (waiting for a thread or using a thread). The arrival rate in the diagram is the arrival rate of requests, λ, and the completion rate is the completion rate of threads. Note that since there can be at most *m* threads in execution, the departure rate is constant and equal to *X* (*m*) for any state *k* > *m*.

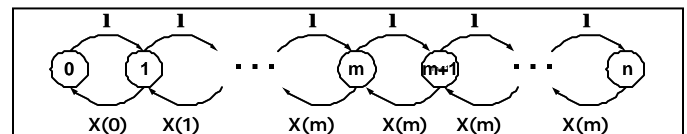

Figure 4.2 – Markov Chain for the thread system

The solution of this Markov Chain, i.e., the values of the probabilities P_k (*k* = 0, …, *n*) of finding *k* requests in the system, can be obtained using the methods in [KLEI 1975, MA 2002]. The solution is given below.

$$P_k = \begin{cases} P_0 \boldsymbol{l}^k / \boldsymbol{b}(k) & k = 1,\ldots,m \\ P_0 \boldsymbol{r}^k X(m)^m / \boldsymbol{b}(m) & k = m+1,\ldots,n \end{cases}$$

where $\boldsymbol{b}(k) = X(1) \times X(2) \times \ldots \times X(k)$, $\boldsymbol{r} = \boldsymbol{l} / X(m)$, and

© 2003 Menascé and Bennani. All Rights Reserved.
In Proc. 2003 Computer Measurement Group Conf., Dec. 7-12, 2003, Dallas, TX.

$$P_0 = \left[1 + \sum_{k=1}^{m} \frac{\lambda^k}{\beta(k)} + \frac{\rho \times \lambda^m (1-\rho^{n-m})}{\beta(m)(1-\rho)} \right]^{-1}.$$

The metrics of interest can be easily computed from the state probabilities and from Little's Law [KLEI 1975] as follows.

$$P_{rej} = P_n$$

$$X = \lambda(1 - P_{rej}) = \sum_{k=1}^{n} X(k) \times P_k$$

$$R = \frac{\sum_{k=1}^{n} k \times P_k}{X}.$$

5. Simulation Experiments

In order to analyze the effectiveness of the controller described above, we developed a simulation program of the multithreaded server in C and C++ using CSim [CSim]. The simulation program also implements the controller code exactly as the controller would operate in an actual system. The controller implemented in this experiment uses two different types of combinatorial search techniques: hill-climbing and beam search [ROR 1996]. The two configuration parameters to be changed by the controller are the values of *n* and *m*. Thus, a configuration point is the pair (*n*, *m*) as defined previously.

Hill-climbing is a very simple search technique that works as follows. Starting from the current configuration Co = (n_o, m_o): examine all "neighbor" configurations to Co and move to the one with the highest value of QoS, computed by the Performance Model solver. A "neighbor" configuration is defined as one in which one of the configuration parameter values is changed by +1 or –1. So, the neighbor configurations of Co are (n_o+1, m_o), (n_o-1, m_o), (n_o, m_o-1), and (n_o, m_o+1). The search is repeated at each new point visited until either i) the value of the QoS does not improve or ii) a threshold on the number of points traversed has been exceeded. One of the drawbacks of hill-climbing is that the search can be prematurely stopped at a local optimum. An illustration of this method is shown in Figure 5.1, which shows various points in the space of possible configurations. A circle represents each configuration. The number within each circle indicates the QoS value of the configuration. Assume that configuration A in Figure 5.1 is the initial configuration. The neighbors of A are B, C, D, and E. Neighbor E is the one with the highest QoS. So, the next point visited is E, which has F as the neighbor with the highest QoS value. F becomes then the next point to be visited and so on.

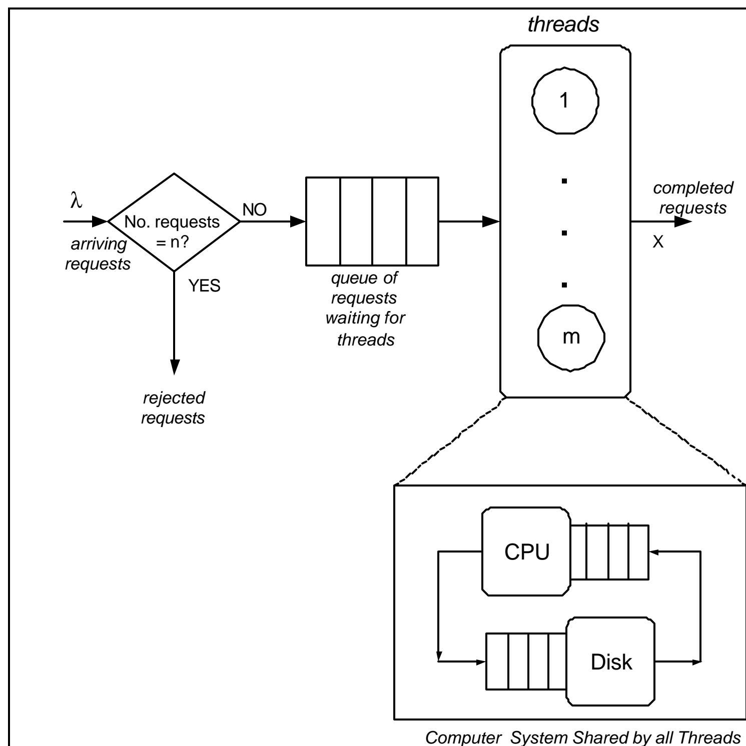

Figure 4.1 - Software and hardware queues.

© 2003 Menascé and Bennani. All Rights Reserved.

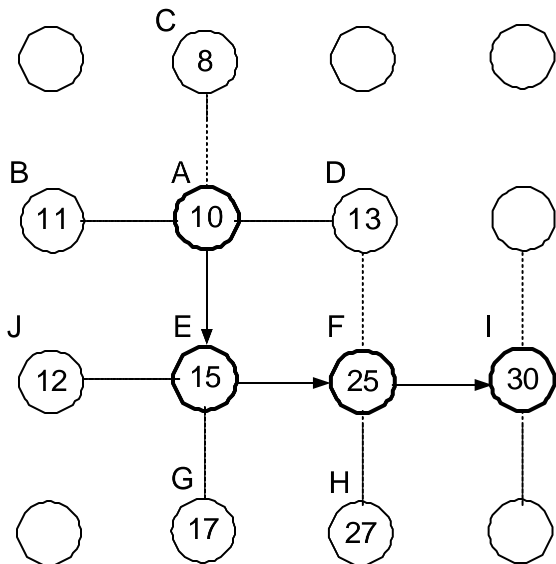

Figure 5.1 – Hill-climbing

Beam search is a combinatorial search procedure that works as follows. Starting from the initial configuration, the QoS value for all the neighbors are computed and the points with the *k* highest values are used to continue the search. The value of *k* is called the *beam*. Then, the neighbors of each of the *k* selected points are evaluated and, again, the *k* highest values among all these points are kept for further consideration. This process repeats itself until a given number of *levels* is reached. Then, the overall highest value is returned. Figure 5.2 illustrates how beam search with *k* = 2 operates starting from the point at level 1. Four neighbors are evaluated but only the two with highest QoS values (15 and 18) are kept. The four neighbors of these two points are evaluated and they constitute level 2 of the tree. The two points with the highest QoS among the eight (22 and 25) are kept. Their neighbors are evaluated and configurations with QoS values 40 and 39 are selected as the two with the two highest values. In this example, the search is limited to four levels.

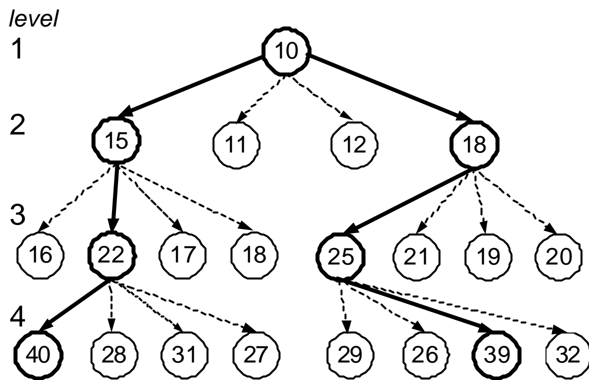

Figure 5.2 Beam search

Our simulation experiments consider the system of Figure 4.1 with one CPU and one disk. The service demands at the CPU and disk are 0.03 sec and 0.05 sec, respectively. The SLAs and respective weights are:

- $R \leq$ 1.2 seconds and w_R = 0.25,
- $X \geq$ 5 requests/sec and w_X = 0.30, and
- $P_{rej} \leq$ 0.05 and w_P = 0.45.

During each experiment, the arrival rate of requests started at a low value of 5 requests/sec and was increased to a peak value of 19 requests/sec and was reduced to 14 requests/sec during a period of one hour, which was divided into 30 controller intervals, as illustrated in Figure 5.1. At the maximum value of 19 requests/sec the utilization of the bottleneck resource (i.e., the disk) is close to 100%. Therefore, we did not increase the arrival rate any further, otherwise the probability of rejection would be too high.

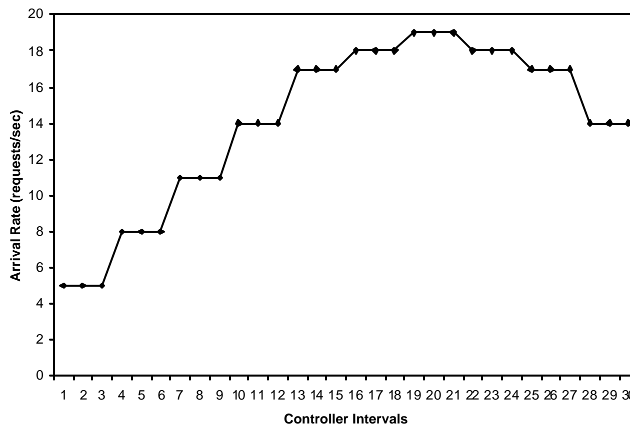

Figure 5.1 – Evolution of the workload intensity.

The controller interval duration is equal to 120 seconds. During that interval, 2280 requests arrive on average during the peak load of 19 requests/sec.

Figure 5.2 illustrates the variation of the QoS metric defined in section 3 during the duration of the experiment. The x-axis is labeled with the values of the average arrival rates observed at each controller interval. There are four curves in Figure 5.2: two of them use the two search heuristics (beam search and hill-climbing) described before, another curve corresponds to the case in which the controller is disabled, and the last one corresponds to the optimal QoS values obtained through an exhaustive search. This value is computed off-line at the end of the simulation run by examining all 9,801 possible configurations at each controller interval and determining the best. It should be noted that for each

© 2003 Menascé and Bennani. All Rights Reserved.

point, the performance model has to be solved in order to compute the QoS value for that point. The two heuristics considered evaluated no more than 120 points per controller interval, i.e., 1.2% of the total number of points.

In Figure 5.2, the optimal QoS curve shows the best QoS for the next controller interval assuming that the arrival rate for that interval is the same as in the current one. The other curves show what was actually measured from the system at the beginning of each interval. These measurements are expected to be lower than the optimal values provided that the arrival rate does not change. Since the arrival rates change from interval to interval, small variations will be observed.

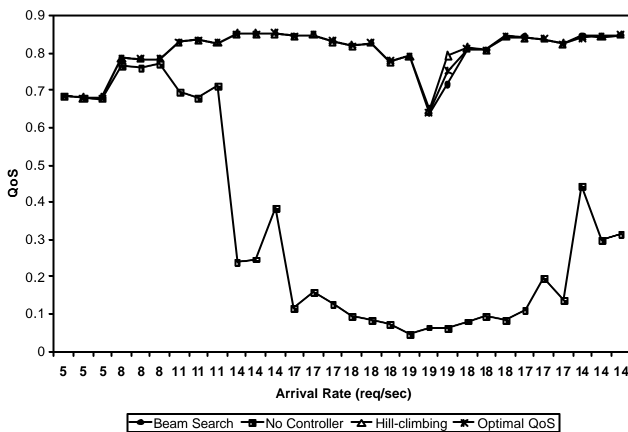

Figure 5.2 – QoS values for beam search, hill-climbing, no control, and exhaustive search.

The following observations can be drawn from Figure 5.2:

- At low loads, the controlled and uncontrolled cases provide almost the same value for the QoS.
- As the arrival rate increases, the controlled system manages to keep the QoS very close to its maximum value, while the QoS for the uncontrolled system falls precipitously to almost zero.
- Both beam search and hill-climbing provide almost the same value for the QoS as the optimal value. In fact, the relative error between any of the two heuristics and the exhaustive search does not exceed 0.7%.

Figure 5.3 shows the utilization of the bottleneck device as a function of the arrival rate of requests. As shown in the figure, when the controller is enabled, the utilization of the disk is always higher than when the controller is turned off. In fact, as the arrival rate reaches its peak value of 19 requests/sec, the disk utilization reaches 97% for the controller case and only 77% for the uncontrolled case. The reason is that the when the controller is disabled, a significant number of requests are rejected, as illustrated in Figure 5.4. At the peak load, 21% of the incoming requests are rejected because the system is not properly configured. Figure 5.3 illustrates a very interesting feature of the QoS controller: existing system resources can be better utilized while providing better QoS to requests.

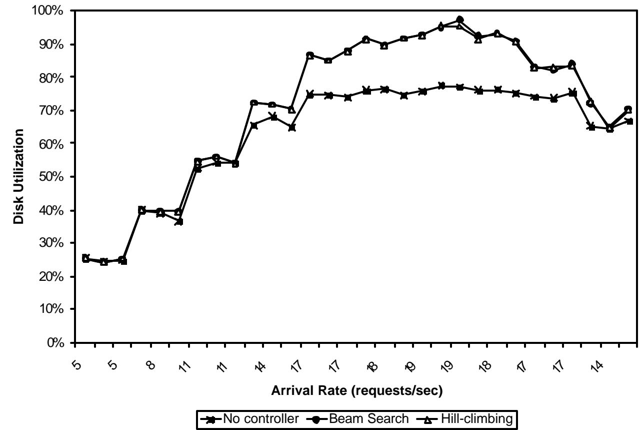

Figure 5.3 – Disk utilization vs. arrival rate of requests.

Figure 5.4 shows that the controlled system adjusts itself to avoid rejecting any requests, even at high loads. The uncontrolled case violates the SLA of 0.05 for the probability of rejection as soon as the arrival rate exceeds 12 requests/sec.

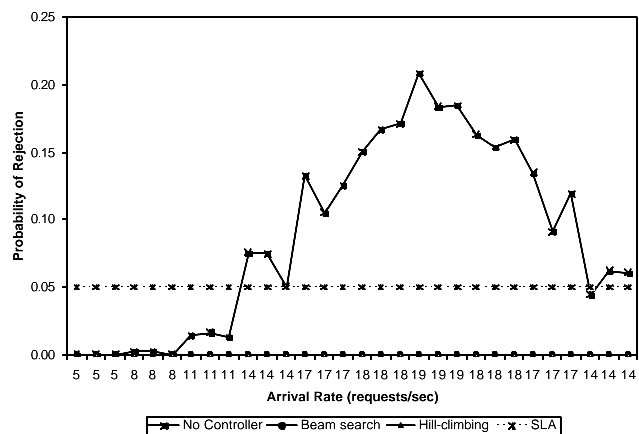

Figure 5.4 – Probability of rejection vs. arrival rate of requests.

Figure 5.5 shows the variation of the average response time vs. the average arrival rate of requests. As it can be seen, as the workload intensity reaches its peak, the controlled system moves towards its response time SLA of 1.2 seconds and even violates it for a very short time interval. The response time of the uncontrolled system is lower than the one for the controlled system because more than 20% of the requests are rejected at high loads and are kept out of

© 2003 Menascé and Bennani. All Rights Reserved.

the system. Thus, the controlled system adjusts itself to meet, as close as possible, the response time SLA while minimizing the probability of rejection. This in turn provides a higher throughput as seen in Figure 5.6.

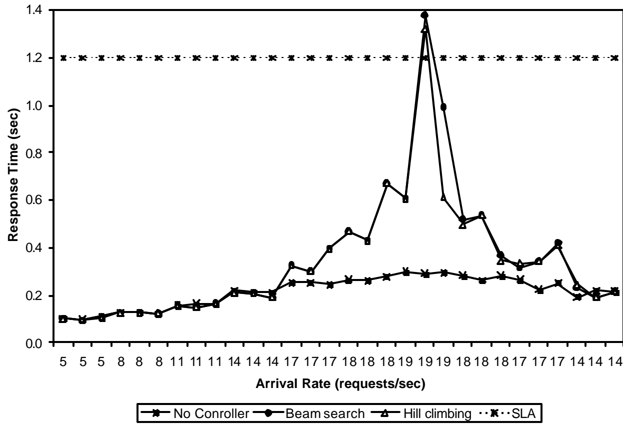

Figure 5.5 – Response time vs. arrival rate of requests.

Figure 5.6 shows that the controlled and uncontrolled systems satisfy the throughput SLA of at least 5 request/sec. However, at peak loads, the controlled system is able to process 19 requests/sec while the uncontrolled system can only process 15.5 requests/sec due to the rejected requests.

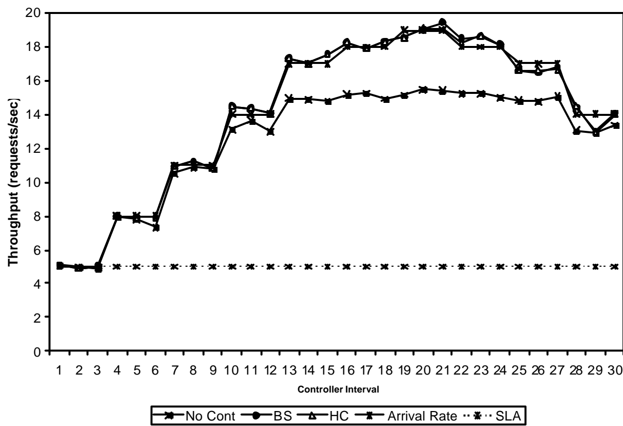

Figure 5.6 – Throughput and arrival rate vs. controller interval.

6. QoS Control of an Actual Web Server

In this section we show the results of applying the techniques described above to the QoS control of an actual Web server. The HTTP server is Apache 1.3.12 which was modified to allow for a dynamic change of the number of active threads (m) and the maximum number of requests in the system (n). The workload used to drive the server is generated by SURGE, a workload generator for Web servers [BC 1998], using two client machines sending requests to a third machine that runs the Web server. SURGE generates references matching empirical measurements regarding file size distributions, relative file popularity, embedded file references, and temporal locality of references. This workload generator was selected because it was demonstrated [BC 98] that, unlike other Web server benchmarks, it exercises servers in a manner that is consistent with actual empirical distributions observed in Web traffic. A fourth machine runs the QoS controller. All four machines are Intel-based and run either Windows 2000 Professional or Windows XP Professional. All machines are connected through a 100 Mbps LAN switch.

The SLAs and respective weights for the experiment are:

- $R \leq$ 0.3 seconds and w_R = 0.5,
- $X \geq$ 50 requests/sec and w_X = 0.2, and
- $P_{rej} \leq$ 0.05 and w_P = 0.3.

Figure 6.1 shows the variation of the QoS during the experiment. As it can be seen, the QoS for the uncontrolled Web server becomes negative when the load gets to its peak value, indicating that at least one of the three metrics is not meeting its SLA. On the other hand, the QoS for the controlled Web server always remains in positive territory for both hill-climbing and beam search.

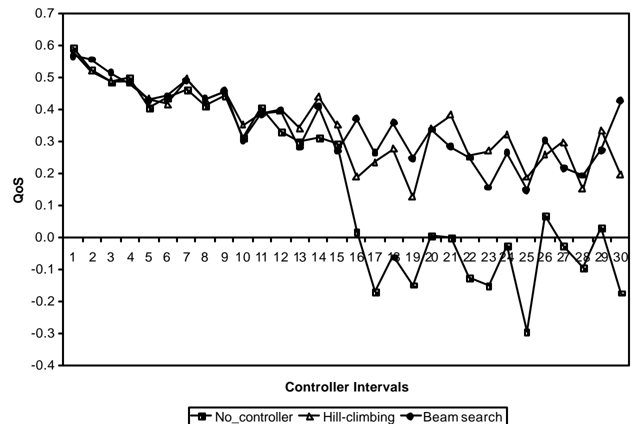

Figure 6.1 – QoS for an actual Web Server with a SURGE workload.

7. Concluding Remarks

The results of this paper provide a novel use for analytic performance models. Traditionally, these models were used to predict system performance for medium- to long-term time intervals as required by capacity planning studies. In our esearch, we used these models to predict QoS values in short time

© 2003 Menascé and Bennani. All Rights Reserved.

intervals of the order of a few minutes. These analytic models have proved to be robust enough to successfully guide heuristic search techniques to provide best configuration parameters for a computer system subject to a varying workload.

We are currently extending the results of this paper along several lines:

- *Load-adjustable controller interval length*: the length of the controller interval should adjust to match the variability of the workload intensity. During periods of stable workload intensity, the controller interval can be increased to reduce the overhead of running the controller.
- *Sliding window procedure for measuring the QoS metrics*: instead of measuring response time, throughput, and probability of rejection in a single controller interval, we are experimenting with using the last *k* controller intervals, with a higher weight on measurements obtained from more recent intervals.
- *Workload forecasting*: to make the controller more proactive as opposed to reactive, we are considering using statistical forecasting techniques for the workload intensity value used by the analytic model.

References

[ALL 1983] B. Abraham, J. Leodolter, and J. Ledolter, ``Statistical Methods for Forecasting,'' John Wiley & Sons, 1983.

[BC 1998] P. Barford and M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," Proc. 1998 ACM Sigmetrics, Madison, Wisconsin, June 22-26, 1998.

[CGK 2003] J. Chase, M. Goldszmidt, and J. Kephart, eds., Proc. First ACM Workshop on Algorithms and Architectures for Self-Managing Systems (Self-Manage '03), San Diego, CA, June 11, 2003.

[CSIM] CSim Development Toolkit for Simulation & Modeling, Mesquite Software, www.mesquite.com.

[DGHPT 2002] 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 Symposium (NOMS 2002), Florence, Italy, April 15-19, 2002, pp. 219—234.

[KLEI 1975] Kleinrock, L., *Queuing Systems*, Vol. I: Theory, John Wiley, 1975.

[MA 2002] Menascé, D. A. and V. A. F. Almeida, *Capacity Planning for Web Services: metrics, models, and methods,* Prentice Hall, Upper Saddle River, NJ, 2002.

[MA 2000] Menascé, D. A. and V. A. F. Almeida, *Scaling for E-business: technologies, models, performance, and capacity planning*, Prentice Hall, Upper Saddle River, NJ, 2000.

[MAD94] Menascé, D. A., V. A. F. Almeida, and L. W. Dowdy, *Capacity Planning and Performance Modeling: from mainframes to client-server systems*, Prentice Hall, Upper Saddle River, NJ, 1994.

[Menascé 2003] Menascé, D. A., "Automatic QoS Control," *IEEE Internet Computing*, January/February 2003, Vol. 7, No. 1.

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

[ROR96] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, eds., *Modern Heuristic Search Methods*, John Wiley & Sons, December 1996.