Proc. ACM Conf. E-commerce, Tampa, FL, October 14-17, 2001.

Preserving QoS of E-commerce Sites Through Self-Tuning: A Performance Model Approach

Daniel A. Menascé
E-Center for E-Business
Dept. of Computer Science
George Mason University
Fairfax, VA 22030, USA
menasce@cs.gmu.edu

Daniel Barbará
E-Center for E-Business
Dept. of Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
dbarbara@ise.gmu.edu

Ronald Dodge
E-Center for E-Business
Dept. of Computer Science
George Mason University
Fairfax, VA 22030, USA
rdodge@gmu.edu

ABSTRACT

The Quality of Service (QoS) of e-commerce sites plays a crucial role in attracting and retaining customers. The workload experienced by these sites tends to vary in a very dynamic way. The complexity of the sites combined with the large short-terms variations of the workload calls for automated methods for site configuration. This paper describes a method for dynamically monitoring and tuning e-commerce sites so that desired QoS levels are attained. Our approach uses hill climbing techniques combined with analytic queuing models to guide the search for the best combination of configuration parameters. We validate our approach in an experimental setting by comparing the QoS levels of a TPC-W e-commerce site with and without control. We showed that under increasing loads, the controlled system meets its QoS goals, while the uncontrolled site fails to do so.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

E-Business, QoS, QoS control, queuing networks, TPC-W

1. INTRODUCTION

The Quality of Service (QoS) of e-commerce sites plays a crucial role in attracting and retaining customers. Frustrated customers leave these sites and do not return, causing revenue to be lost. The performance of an e-commerce site is a function of a large number of parameters such as the number of threads at each level, the maximum number of connections accepted, the maximum number of requests served by each thread, cache sizes, cache replacement policies and parameters, as well as load balancing policies and parameters.

The workload of information providing web sites has been extensively studied [6, 17]. An analysis of the workload seen by e-commerce sites is presented in [12, 15] and it was shown that these workloads tend to vary very dynamically and exhibit short-term fluctuations. The challenge for e-commerce sites is how to best use their existing resources to cope with short-term fluctuations in the workload in a manner that the desired QoS levels are met.

This problem has been addressed in many different ways. In [5], a session-admission control mechanism is proposed. Requests may be classified into high, medium, or low priority based on the configured policy. Priority levels are used to determine admission priority and performance-level. When the site cannot provide the desired QoS, new sessions are rejected so that the current ones can continue to experience good performance. These techniques were later incorporated in HP's WebQoS product [7]. While this approach works well for sessions in progress, it does not deal with an important QoS metric, namely the probability that a request is rejected. Another approach to QoS control is the one incorporated in Peakstone's eAssurance, which uses statistical models, including Bayesian and stochastic modeling to model site behavior [16]. These statistical models are constantly updated based on observations of changes in applications, infrastructure, or traffic. The models used by eAssurance are different in nature from the ones we propose in our work. Our models are based on predictive queuing models of computer systems. Moreover, given that the methods used by eAssurance are proprietary, it is difficult to make a more thorough comparison. In [13], the authors propose a family of resource management policies that dynamically assign priorities to customers. This approach is aimed at using the site's existing resources to optimize business metrics such as revenue throughput but does not provides guarantees in terms of QoS.

This paper addresses a method by which e-commerce sites can track their workload and the value of QoS metrics to dynamically determine how different configuration parameters should be changed to meet QoS requirements. We consider three basic QoS metrics, although others could easily be incorporated into our framework: site response time, site throughput, and probability that a request is rejected.

EC'01, October 14-17, 2001, Tampa, Florida, USA.

Our proposed approach is general enough and can be used to dynamically change any parameters that can be changed at run time, including request and or session priority, as in WebQoS, any software reconfigurable parameters, or even the number of CPUs. Some vendors already allow the number of CPUs to be dynamically repartitioned across domains. An example of that is Sun's Automatic Dynamic Reconfiguration (ADR), available in Enterprise 10000 servers [19].

We built a controller that monitors the site and uses a hill-climbing technique guided by a queuing network model to determine new values of various configuration parameters. The configuration is then changed dynamically according to the results of the search in order to ensure that the site shows as little deviation as possible from the desired QoS levels.

Our approach was validated experimentally by integrating our controller into an e-commerce site we developed following the TPC-W benchmark guidelines [20]. We also built a parameterized TPC-W workload generator to drive the site at various patterns of arrival rates. We showed that as the arrival rate increases and reaches its peak value, the QoS of the controlled system manages to remain positive, thereby meeting the QoS requirements, while that of the uncontrolled site goes deeply into negative territory.

The rest of the paper is organized as follows. Section 2 provides the background and basic concepts needed in the following sections. Section 3 presents an overview of the basic approach used by the QoS controller. The next Section describes the queuing network model used by the controller. Section 5 presents the experimental testbed used to assess the efficiency of the controller. Results of the experiments are discussed in section 6 and Section 7 presents some concluding remarks, and future work.

2. BACKGROUND

An e-commerce site is typically composed of multiple layers including web servers, application servers, and database servers as illustrated in Fig. 1.

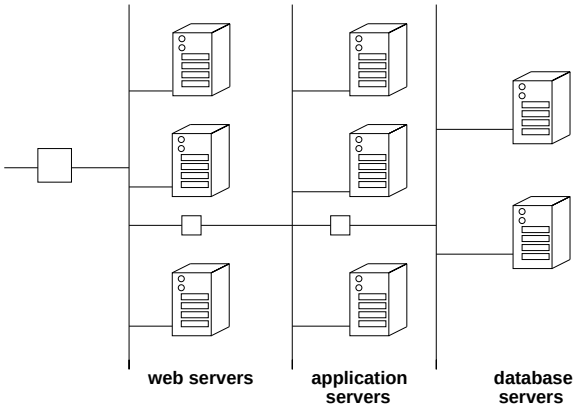

Figure 1: An e-commerce site with a multi-layered architecture.

A request to execute an e-business function is first handled by one of the Web servers. In almost all cases, the HTML page that is returned to the client is dynamically generated by an application server that may need to access a database server more than once during the execution of the application to obtain data needed to build the page. If the request is for an in-line image, it can be satisfied directly by a web server.

Web servers, application servers, and database servers are usually multi-threaded. So, an arriving request to any of the servers needs to first queue for a thread. In many cases, there is a limit on the maximum number of requests that can be either waiting for a thread to become available or being serviced by a thread. In these cases, when an arriving requests find the queue at its maximum size, the request is rejected.

We consider in this work three quality of service metrics:

- Server-side response time (R): time elapsed since a request arrives the site until it is completely processed and a reply is sent to the client. This time, for our purposes, does not include any network time external to the site.
- Probability of rejection (P_{rej}): probability that an arriving request will be rejected because any of the queues is at its maximum capacity.
- Site throughput (X_0): number of requests per second that complete execution from the site.

Usually, managers of e-commerce sites specify bounds on the values of the QoS levels and monitor the site to ensure that these levels are being met. We define the following QoS requirements:

- Maximum average server-side response time ($R_{\max}$): maximum value for the average response time that one is willing to tolerate.
- Maximum probability of rejection ($P_{\text{rej}}^{\max}$): maximum acceptable value for the probability of rejection.
- Minimum site throughput ($X_0^{\min}$): acceptable lower bound on the throughput.

We can now define *QoS deviations* for each of the QoS metrics as below.

$$\Delta QoSR = \frac{R_{\max} - R}{R_{\max}} \quad (1)$$

$$\Delta QoSX_0 = \frac{X_0 - X_0^{\min}}{X_0^{\min}} \quad (2)$$

$$\Delta QoSP_{\text{rej}} = \frac{P_{\text{rej}}^{\max} - P_{\text{rej}}}{P_{\text{rej}}^{\max}} \quad (3)$$

The definitions above have the following property for any of the three QoS metrics: i) a QoS deviation has no dimensions and represents the percent deviation from the required QoS level, ii) the QoS deviation is non negative if the site meets the QoS requirement for that metric and negative otherwise.

While response time and probability of rejection are customer-perceived QoS metrics, throughput is a site-wide metric. Also, an increase in throughput may come at the expense of an undesirable increase in response time. Site managers have to determine how they want to balance the QoS requirements. For that purpose, we define a QoS function that combines all three QoS metrics as

$$QoS = w_R \times \Delta QoSR + w_X \times \Delta QoSX_0 + w_P \times \Delta QoSP_{\text{rej}}, \quad (4)$$

where w_R, w_X, and w_P are weights assigned by site management to each QoS deviation. These weights must sum to one and reflect the importance given by management to each QoS metric.

3. APPROACH

The approach we follow to control an e-commerce site is based on searching the space of values of configurable parameters for a point where the aggregate metric QoS define in Eq. (4) is maximized or close to being maximized.

Figure 2 shows the architecture of the QoS controller and its relationship to the e-commerce site. The main modules of the controller are the Workload Monitor, Performance Monitor, Configuration Controller, Performance Model Solver, and the QoS Monitor. Their main functions are described in what follows using the numbers in parentheses in Fig. 2, which refer to flow of information between the e-commerce site and the controller (dashed lines) and flow of information between modules of the controller (solid lines).

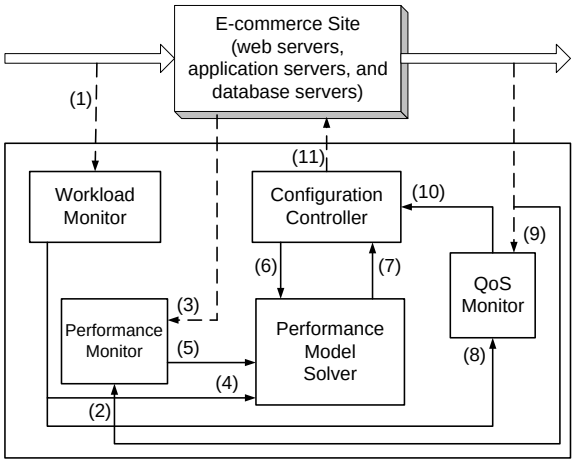

Figure 2: Architecture of the QoS controller.

The QoS controller collects data during intervals of time called *controller intervals (CIs)*. At the end of each CI, the QoS controller decides, according to one of the control policies described later in the section, if reconfiguration needs to take place and how. The length of a CI may be variable depending on the control policy in use by the controller. In our experiments, the duration of a CI was 120 sec for the fixed CI policies.

The Workload Monitor collects information about the arrival process of requests (1) and computes the average arrival rate of requests to the site during the CI. The Performance Monitor measures device (e.g., CPU, disks) utilizations (3) for all Web, application, and database server machines of the e-commerce site. This information, along with the throughput (2) is used by the Performance Monitor to compute the service demands at each device during the CI. Service demand is defined as the total service time per request at a given device. This time does not include any queuing time at the device [14]. For example, the service demand $D_{\text{cpu}}^{\text{WS}}$ at the CPU of a web server is computed as $U_{\text{cpu}}^{\text{WS}}/X_0$, where $U_{\text{cpu}}^{\text{WS}}$ is the observed CPU utilization of the Web server processes during the CI and X_0 the site throughput. So, the Performance Monitor produces at the end of each CI, a service demand vector $\vec{D} = (\vec{D}_{\text{WS}}, \vec{D}_{\text{AS}}, \vec{D}_{\text{DS}})$ where $\vec{D}_{\text{WS}}, \vec{D}_{\text{AS}}$, and $\vec{D}_{\text{DS}}$ are the vectors of service demands for all devices at the web servers, application servers, and database servers, respectively.

The QoS Monitor checks, at the end of each CI, if any of the QoS metrics was violated by receiving information (9) on completing requests from the e-commerce site. The QoS monitor decides if there is a need to change the configuration. In the affirmative case, it instructs (10) the Configuration Controller to determine a new configuration for the site.

We explain now how the Configuration Controller determines a new configuration. Let $\vec{C} = (c_1, c_2, \cdots, c_P)$ be a vector of P configuration parameters that can be dynamically changed. Every parameter c_i has a range given by $(c_i^{\min}, c_i^{\max})$. In our implementation we used as configuration parameters the number of threads at the web and application servers and the maximum queue size of requests at each of these servers. The Configuration Controller uses a simple hill-climbing technique (a well known search technique widely used in optimization algorithms), to search the space of configurations for one that improves the QoS value. While hill-climbing does not guarantee optimality, it is a simple heuristic that performs well in a variety of applications. Let $QoS(\vec{C})$ be the QoS value for configuration $\vec{C}$. When the Configuration Controller is informed that it needs to find a new configuration by the QoS Monitor, it also received from it the QoS value for the current configuration $\vec{C}_0$.

The search for a new configuration is based on a hill-climbing method. From the current configuration we examine all neighbor configurations. A neighbor configuration is defined as one in which the value of one of the configuration parameters is incremented by moving one step. If we assume that the domain of each configuration parameter is the set of integers in the range $(c_i^{\min}, c_i^{\max})$, this step is taken by incrementing one of the parameters by plus or minus one. Defining the vector $\vec{1}_i$ as a vector $(0, 0, \cdots, 1, \cdots, 0)$ with a 1 in the i-th position and zero everywhere else, we can say that $\vec{C} - \vec{1}_i$ is an example of a neighbor configuration of $\vec{C}$. For every possible neighbor configuration, the algorithm uses a predictive queuing network model of the site, as explained in Section 4, to compute the QoS for the neighboring configuration. The Performance Model Solver computes the QoS value for each configuration it receives (6) from the Configuration Controller, using service demands received from the Performance Monitor (5) and the arrival rate received from the Workload Monitor (4).

The neighboring configuration with the largest QoS is selected as the next configuration to examine and the process repeats itself from that configuration. The search continues until either no improvement can be made or we reached a limit on the maximum number of hops in the path to the new configuration. This latter type of limitation may be needed to avoid unstable behavior. A pictorial description of the hill climbing process for the case of two parameters c_1 and c_2 is given in Fig. 3. The numbers next to each configuration are the QoS values for the configuration. The configurations with a white interior are the abandoned ones.

A complete description of the algorithm is given in Fig. 4. As can be seen in the algorithm, the complexity of the search

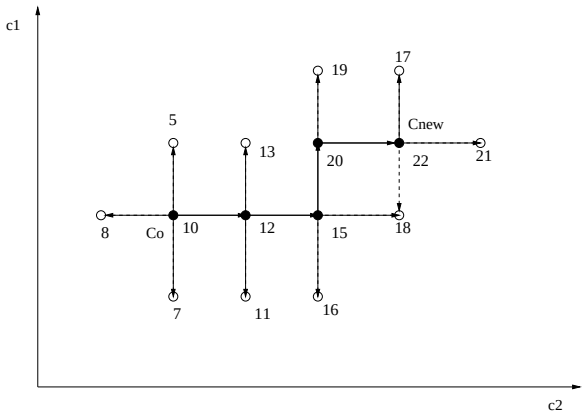

Figure 3: Example of the Hill Climbing Approach.

is proportional to $NumHops \times 2 \times P$.

```
NumHops ← 1;
C⃗curr ← C⃗0;
C⃗new ← C⃗0;
Repeat
  Improved ← False;
  MaxQoS ← QoS(C⃗curr);
  For i = 1 to P Do
    Begin
      If ci − 1 ≥ cimin
      Then If QoS(C⃗curr − 1⃗i) > MaxQoS
           Then Begin
                  MaxQoS ← QoS(C⃗curr − 1⃗i);
                  C⃗new ← C⃗curr − 1⃗i;
                  Improved ← True
                End;
      If ci + 1 ≤ cimax
      Then If QoS(C⃗new + 1⃗i) > MaxQoS
           Then Begin
                  MaxQoS ← QoS(C⃗curr + 1⃗i);
                  C⃗new ← C⃗curr + 1⃗i;
                  Improved ← True
                End
    End;
    C⃗curr ← C⃗new;
    NumHops ← NumHops + 1;
Until (¬ Improved | (Numhops = MaxHops))
```

Figure 4: New Configuration Search Algorithm

4. THE QUEUING NETWORK MODEL

The hill climbing procedure described in the previous section requires the computation of the QoS value for a given configuration. This is obtained through the use of a predictive queuing model discussed here. We start by first discussing a model for a single tier site and then extend it to multiple tiers.

4.1 Single Tier Model

We describe now a model that can be used to represent both contention for software resources (e.g., server threads) as well as hardware resources (e.g., CPU and disks) on a single tier (e.g., web server tier). Figure 5 depicts the combined model. Requests arrive to be serviced by one of the m server threads. If the number of requests k waiting or being served by a thread is equal to n, the arriving request is rejected. Otherwise, it queues for a thread. Once a request is able to obtain a thread, it starts to use the physical resources of the tier. A thread at this tier may request service from a thread from a lower level tier. For example, a web server tier may need to request service from the application server. So, a thread is busy while a request is either a) using a physical resource, b) waiting for a physical resource, or c) waiting for a response from a lower level tier.

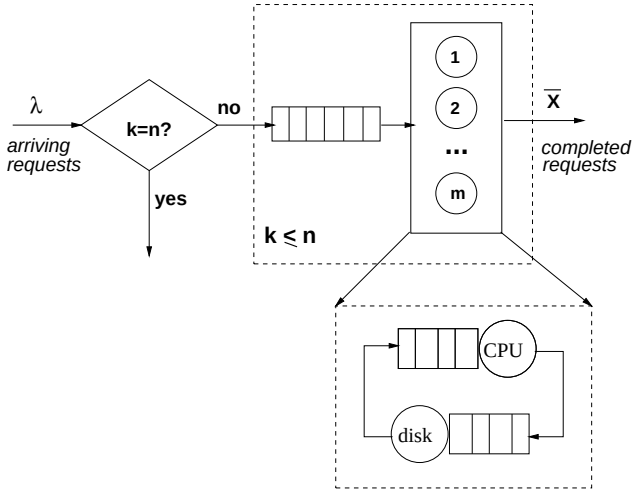

Figure 5: A queuing model for a single tier server.

Let $X(k), k = 1, \cdots, n$ be the rate at which a thread completes execution. This rate depends on the number of concurrent threads in execution and on the response time of the lower level tier, if any. If we know the values of $X(k)$ we can compute the probability P_k that k requests are in the system. This can be done using a birth-death process [8] with states $0, 1, \cdots, n$, arrival rate λ and departure rate μ_k defined as $\mu_k = X(k)$ for $k \leq m$ and $\mu_k = X(m)$ for $k = m+1, \cdots, n$.

This type of system is solved in [11] and the solution is

$$P_k = \begin{cases} P_0 \ \lambda^k/\beta(k) & k = 1, \cdots, m \\ P_0 \ \rho^k \ X(m)^m/\beta(m) & k = m+1, \cdots, n \end{cases} \quad (5)$$

where $\beta(k) = X(1) \times X(2) \times \cdots X(k)$, $\rho = \lambda/X(m)$ and

$$P_0 = \left[1 + \sum_{k=1}^{m} \lambda^k/\beta(k) + \frac{\rho \ \lambda^m \ (1-\rho^{n-m}}{\beta(m)(1-\rho)} \right]^{-1}. \quad (6)$$

Once the probabilities P_k are known one can easily obtain three QoS metrics of interest: average throughput ($\overline{X}$), average response time (R), and probability that a request is

rejected (P_{rej}) as follows.

$$\overline{X} = \sum_{k=1}^{m} X(k)\ P_k + X(m) \sum_{k=m+1}^{n} P_k \quad (7)$$

$$R = (1/\overline{X}) \sum_{k=1}^{n} k\ P_k \quad (8)$$

$$P_{\text{rej}} = P_n \quad (9)$$

Equation (8) follows directly from Little's Law [8] since the summation is the average number of requests in the system.

The question now is how to compute the values of $X(k), k = 1, \cdots, m$ needed to compute the probabilities $P_k, k = 1, \cdots, n$. These values can be computed by solving a closed queuing network model (see [11, 14]) composed of all K physical resources (e.g., CPU and disks) of the tier as well as a virtual device that represents the time D_{lower} spent by a thread waiting for service provided by the lower tier. This virtual device is modeled in the queuing network as a delay device with no queuing. One can then use Mean Value Analysis [18, 11, 14] to solve the closed queuing network and obtain the throughput values $X(k)$, using the following procedure

$\bar{n}_i(0) = 0$ for $i = 1, \cdots, K$
For $k = 1, \cdots, m$ **do**

$$R_i^{'}(k) = D_i \times [1 + \bar{n}_i(k-1)] \quad \text{for } i = 1, \cdots, K$$

$$X(k) = \frac{k}{D_{\text{lower}} + \sum_{i=1}^{K} R_i^{'}(k)}$$

$$\bar{n}_i(k) = X(k) \times R_i^{'}(k) \quad \text{for } i = 1, \cdots, K$$

End For

where $\bar{n}_i(k)$ is the average number of requests at physical resource i when there are k busy threads and $R_i^{'}(k)$ is the average total time spent, queuing and receiving service, by a request at physical resource i when there are k busy threads. Let us now define functions that return the average response time and probability of rejection of any given tier as a function of the following parameters:

- λ: average arrival rate of requests to the tier,
- $\vec{D}$: vector of service demands for the physical resources of the tier,
- D_{lower}: total time spent by a thread waiting for service from the lower tier,
- m: number of threads, and
- n: maximum number of requests.

So, using the model just discussed above one can obtain the functions $RT(\lambda, \vec{D}, D_{\text{lower}}, m, n)$ and $PRej(\lambda, \vec{D}, D_{\text{lower}}, m, n)$, for the average response time and probability of rejection, respectively. The effective arrival rate λ_{eff} at the tier, i.e., the rate of requests not rejected, is then equal to $\lambda \times (1 - P_{\text{rej}})$.

4.2 Multiple Tier Model

We now consider a complete e-commerce site composed of three tiers: web servers, applications servers, and database servers. The queuing model for the entire site is built through a composition of the the single-tier model described in the previous subsection. The complete queuing model is shown in Fig. 6.

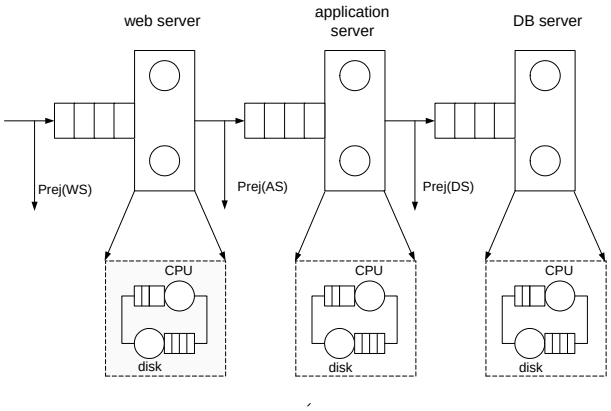

Figure 6: The queuing network model for the E-commerce site.

Once we combine the three tiers, we introduce a dependency between results obtained in each as illustrated in Fig. 7, which shows how the different metrics in each layer depend on one another. The arrows in the figure mean "depends on."

Due to the cyclic dependency that exists in Fig. 7, we need to use an iterative approach to solve the model. Before we present the algorithm, some definitions are in order:

- λ^{WS}: overall arrival rate of requests to the e-commerce site,
- f: fraction of requests processed by the Web server that require service from the application server, and
- N_{db}: average number of DB calls per execution of an application server thread.

The iterative algorithm to solve the model is given in Fig. 8.

4.3 Model Validation

The model presented in the previous subsections is an approximation and needs to be validated. We should emphasize however, that the goal of the model is to compute the QoS value for a given configuration with sufficient precision to be useful by the controller. Thus, the model must be able to track the trends in performance reasonably well to be useful. To verify this, we compared performance predictions by the model with measurements obtained in the experimental setting described in Section 5. Figures 9 and 10 present a comparison between measured and modeled response times and probability of rejection, respectively. The response time curves also show 95% confidence interval bars for each measured average response time value. The results obtained with the analytic model are within the confidence intervals for the measured values.

The pictures show that the model tracks reasonable well the measurements for the purpose of being used by the controller algorithm.

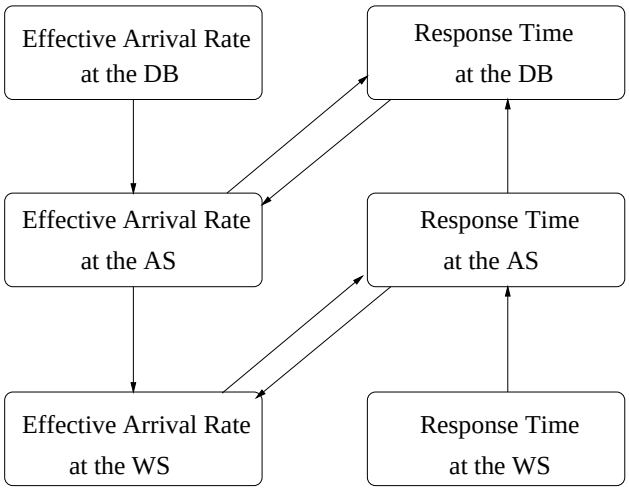

Figure 7: Dependency Between the Models at the Three Tiers

Initialize the probability of rejection to zero
$P_{\text{rej}}^{\text{DB}} \leftarrow 0$; $P_{\text{rej}}^{\text{AS}} \leftarrow 0$; $P_{\text{rej}}^{\text{WS}} \leftarrow 0$; $R_{\text{old}} \leftarrow 0$;
Repeat
  Compute arrival rate to DB server
  $\lambda^{\text{DB}} \leftarrow \lambda^{\text{WS}} \times (1 - P_{\text{rej}}^{\text{WS}}) \times f \times (1 - P_{\text{rej}}^{\text{AS}}) \times N_{\text{db}}$
  Compute the DB response time
  $R_{\text{DB}} \leftarrow RT(\lambda^{\text{DB}}, \vec{D}_{\text{DB}}, 0, m_{\text{DB}}, n_{\text{DB}})$
  Compute the DB rejection probability
  $P_{\text{rej}}^{\text{DB}} \leftarrow PRej(\lambda^{\text{DB}}, \vec{D}_{\text{DB}}, 0, m_{\text{DB}}, n_{\text{DB}})$
  Compute the arrival rate to the AS server
  $\lambda^{\text{AS}} \leftarrow \lambda^{\text{WS}} \times (1 - P_{\text{rej}}^{\text{WS}}) \times f$;
  Compute the AS response time
  $R_{\text{AS}} \leftarrow RT(\lambda^{\text{AS}}, \vec{D}_{\text{AS}}, N_{\text{db}} \times R_{\text{DB}}, m_{\text{AS}}, n_{\text{AS}})$
  Compute the AS rejection probability
  $P_{\text{rej}}^{\text{AS}} \leftarrow PRej(\lambda^{\text{AS}}, \vec{D}_{\text{AS}}, N_{\text{db}} \times R_{\text{DB}}, m_{\text{AS}}, n_{\text{AS}})$
  Compute the WS response time
  $R_{\text{WS}} \leftarrow RT(\lambda^{\text{WS}}, \vec{D}_{\text{WS}}, f \times R_{\text{AS}}, m_{\text{WS}}, n_{\text{WS}})$
  Compute the WS rejection probability
  $P_{\text{rej}}^{\text{WS}} \leftarrow PRej(\lambda^{\text{WS}}, \vec{D}_{\text{WS}}, f \times R_{\text{AS}}, m_{\text{WS}}, n_{\text{WS}})$
  Compute the error in this iteration
  Error $\leftarrow | \frac{R_{\text{WS}} - R_{\text{old}}}{R_{\text{WS}}} |$
  Prepare for next iteration
  $R_{\text{old}} \leftarrow R_{\text{WS}}$
Until (Error $<$ Tolerance)
Compute Final Metrics
$R \leftarrow R_{\text{WS}}$;
$P_{\text{rej}} \leftarrow P_{\text{rej}}^{\text{WS}} + (1 - P_{\text{rej}}^{\text{WS}})(P_{\text{rej}}^{\text{AS}} + (1 - P_{\text{rej}}^{\text{AS}})\ P_{\text{rej}}^{\text{DB}})$;
$X_0 \leftarrow \lambda^{\text{WS}}\ (1 - P_{\text{rej}})$

Figure 8: Performance Model for the E-commerce Site

5. THE EXPERIMENTAL TESTBED

To assess the effectiveness of the QoS controller, we designed and implemented a prototype of an e-commerce site and a workload generator. The e-commerce site follows the TPC-W benchmark specifications [20]. The workload generator generates customer sessions based on the Customer Behavior Model Graph (CBMG) [12] specified in the TPC-W benchmark. Figure 11 shows the architecture of the experimental testbed. All the machines are Intel-based PCs running Windows 2000.

In the following sub-sections we discuss the implementation of each component of the prototype.

5.1 The Web Server

The Web server uses a modified version of the Open SA Apache Web Server integrated with SSL support. The modifications to the web server include application server integration, a finite, modifiable client request queue and a configuration management process. Figure 12 shows how requests are processed by the Web server. A more detailed description follows.

The application server coordination is implemented through a handler module, integrated with the Apache API. The module communicates with the application server and sends the dynamically generated pages to the client. To implement the connection management, a queue management process is integrated into the Apache core module. Running as a separate thread, the queue manager communicates with the system controller module using Windows NT named pipes to receive configuration change instructions. When a change is received, the manager issues instructions for the required configuration change. As required, the manager either creates more threads or issues a thread reduction command. As threads complete servicing a request and return to the request queue for a job, the new thread total is checked. If the number of active threads is above the new maximum, the thread exits. This implementation protects the integrity of a client request and prevents configuration changes from effecting a current request. Additionally, the manager may increase or decrease the maximum number of connections

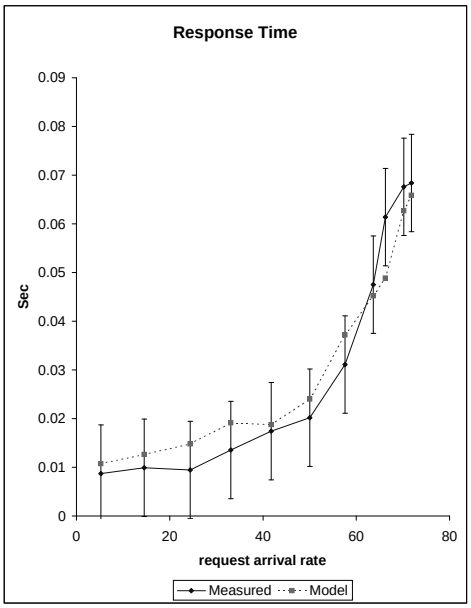

Figure 9: Measured and modeled response time.

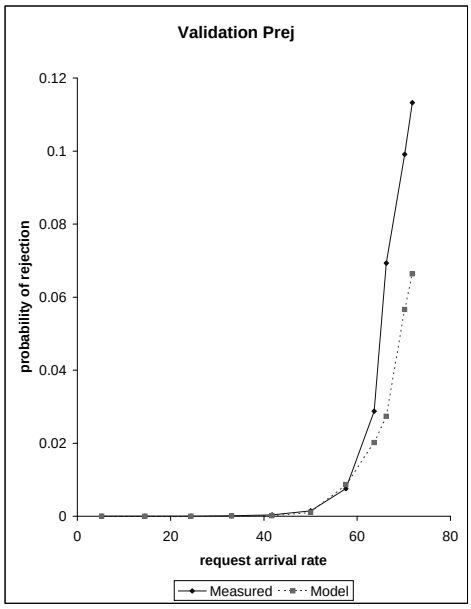

Figure 10: Measured and modeled probability of rejection.

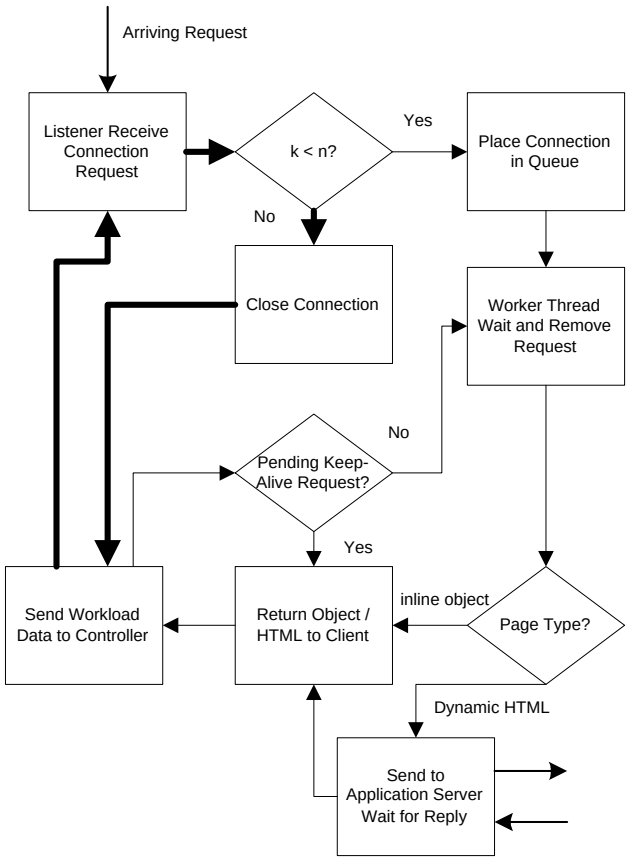

Figure 12: Web Server Request Processing.

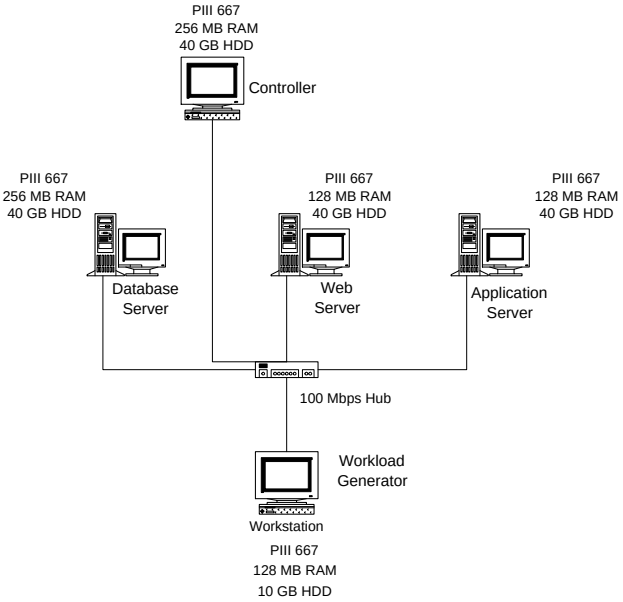

Figure 11: Experimental Testbed.

allowed in the system. As the Apache listener process receives TCP connection requests from clients, it adds the requests to a semaphore-based queue. The number of current requests in the system is checked at each addition to the requests queue. If the addition of the new request will exceed the maximum system size the connection is rejected.

The web server also reports workload statistics to the QoS controller module using named pipes. If the Listener process rejects a request, a rejection message is sent to the controller. Once a worker thread removes the request from the queue it parses the request to determine its type. If the request is for an in-line object (e.g., image) the thread retrieves the object and sends it back to the client. If the request is for an HTML page, the handler module for the application server is called. The handler module uses the web server worker threads process Identification number (PID) to create a unique named pipe and sends the request and the pipe name to the application server. The thread then sleeps on the pipe, waiting to receive the HTML page. Upon receipt, the page is sent to the client. The worker thread computes the request response time, sends it to the controller, and decreases the current queue size by one. In the case of a pending keep-alive request, the worker thread increases the current maximum queue size and processes the request. If there is not a pending keep-alive request, the worker thread returns to the job queue for another request. The worker thread increases the queue size counter without regard to the maximum queue size by design. This is intended to

preserve the integrity of a request. An alternate approach would be to treat each request as a new arrival to the system and subject it to rejection.

5.2 Application Server

The Application server is responsible for generating dynamic HTML web pages. The server is an original design and uses multithreading to process simultaneous requests. The server consists of three modules, a listener module, a request handler module, and a controller module. Figure 13 shows how requests are processed by the application server. A more detailed description follows.

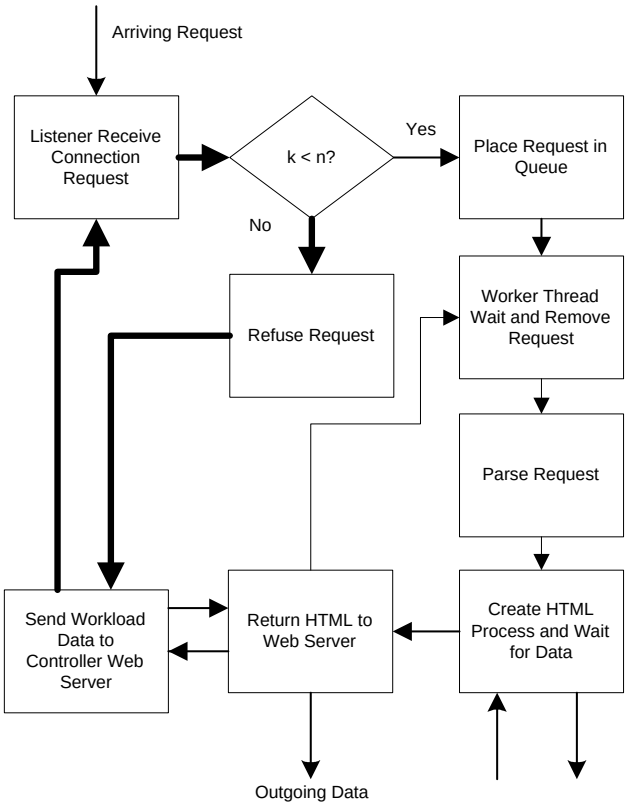

Figure 13: Application Server Request Processing.

The application server uses named pipes to receive requests from the web server. Similarly to the web server, when a request is received, the listener thread that received the request attempts to place the request on the request queue. This is implemented as a semaphore queue in a similar manner as the web server's connection queue. If the listener thread rejects the request, it sends a rejection notification to the controller and to the web server. The web server then closes the connection and the web server worker thread returns to the job queue. After a request has been placed on the request queue at the application server, a worker thread from the request handler module removes the request. The request handler worker thread parses the request to retrieve the request body and to determine which web server thread to send the HTML page back to. The worker thread then creates a process to generate the dynamic HTML page. The worker thread receives the HTML page from the process and forwards it to the web server. As part of the HTML page returned by the process, a counter of the number of database transactions performed during generation of the HTML page is also returned. This counter along with the response time for the request is sent to the QoS controller. The worker thread then returns to the request queue.

5.3 Database Server

The database server for our prototype uses Microsoft SQL Server 2000 and is not included in our configuration parameters. The performance parameters (device service demands and average response time per database request) however are included in the QN model solver. The application server is responsible for reporting to the QoS controller the number of database calls N_{DB}. This information is used to compute the arrival rate to the database server which is used in the solution of the performance model as discussed previously.

5.4 Workload Generator

The workload is generated from a multi-threaded Browser Emulator application that we developed. This workload generator runs on a dedicated machine connected to the e-commerce site by a 100Mbps Ethernet LAN. The Browser application uses multi-threading to emulate a group of client browsers and consists of a controller process and a variable number of browser threads. Each of the browser threads presents a unique workload to the e-commerce site. The navigation of a given thread through the e-commerce system follows the TPC-W navigation probabilities.

To provide repeatable performance from the Browser application, each thread seeds its random number generator with the same number each time it is run. When additional threads are created to increase the number of active sessions, they will continue the series and seed the random number generator accordingly. While this provides a very high degree of repeatability, the randomness with which the e-commerce system will refuse connections when it becomes heavily loaded will inject variability in the results. This however is unavoidable and would occur even if each browser used a trace for its workload generation.

Each browser thread emulates an HTTP/1.1 compliant browser. The browser thread first requests the HTML home page for the e-commerce site (dynamically generated). After receiving the response from the server, the browser parses the page, extracting information such as in-line objects to request (images), customer ID and session ID, and any items that may be in the shopping cart. Once the page has been parsed, the browser thread divides up the in-line requests between a set of reader threads (three in our experiments) that use a pipelined/keep-alive request framework to retrieve the images. This technique, common in today's browsers, involves combining a series of requests in to a single request message (pipelining) and then retrieving the response for each segment of the message without opening a new TCP connection to the server (keep-alive).

After the in-line images have been received, the browser thread determines the composition of its next request through a routing table based on a given client class. Each possible request available to the browser has a set of tasks associated with it according to the TPC-W specification. For example, if the next state is the shopping cart page, the browser thread determines if it has any item in its cart, and which items to increase, decrease or remove from the cart.

The controller process in the workload generator varies the workload presented to the e-commerce system by increasing or decreasing the number of active sessions. When the controller process needs to reduce the workload, it signals the browser threads that the number of active sessions has been reduced. At the completion of each request, a browser thread checks to see if the browser population has been reduced to the new level. If not, it decrements the population count by one and sets its own session_stop flag to true. The next time the browser thread completes a home page request it will exit. This is done to retain the probability distribution in the CBMG. If additional browsers are needed, the controller increments the population counter by the additional browser count and spawns the required number of browsers.

6. EXPERIMENTAL RESULTS

To assess the efficiency of the QoS controller we ran several experiments using the testbed described in the previous section. Client threads at the TPC-W workload generator start sessions that follow the navigational pattern of the TPC-W benchmark. The number of concurrent sessions is increased in steps of ten every five minutes from a starting number of five until a maximum of 65 concurrent sessions. Then, at every five minutes, the number of sessions is decreased to 35. This entire variation of the arrival rate lasts for thirty controller intervals (CI), where each CI is fixed at 120 seconds. We made ten independent runs of each such experiment using a different and independent seed for the random number generators used by the TPC-W Workload Generator. For each of the ten runs we repeated the sequence with the QoS controller enabled and disabled. This gave us then a total of twenty runs. At the end of each of the thirty CIs, we compute the QoS value as defined in Eq. (4). An average of about 246,000 requests were submitted in each of the twenty runs. The experiments reported in this section assume the thresholds and weights for the response time, throughput, and probability of rejection shown in Table 1.

	Response Time (msec)	Throughput (req/sec)	Probability of Rejection
Max/Min	200	85	5%
Weight	0.5	0.2	0.3

Table 1: QoS Requirements and Weights for the Experiments.

The four configuration parameters that could be changed by the QoS controller were the maximum number of requests waiting or being processed at the Web server, the number of Web server threads, the maximum number of requests waiting or being processed at the application server, and the the number of application server threads.

Figure 14 illustrates the resulting variation of the arrival rate during an experiment. So, the average arrival rate of requests varies from about 14 requests/sec to 90 requests/sec. It is important to emphasize that the theoretical maximum throughput for this system computed as the inverse of the maximum service demand [14] is 102 requests/sec. So the system was driven to about 88% of its maximum possible throughput during the experiment. The values depicted in the graph represent the average values at each CI for all the experiments. It should be pointed out that the coefficient of variation (i.e., standard deviation over the mean) of the value of the average arrival rate in each CI is rather small, on the order of 0.1.

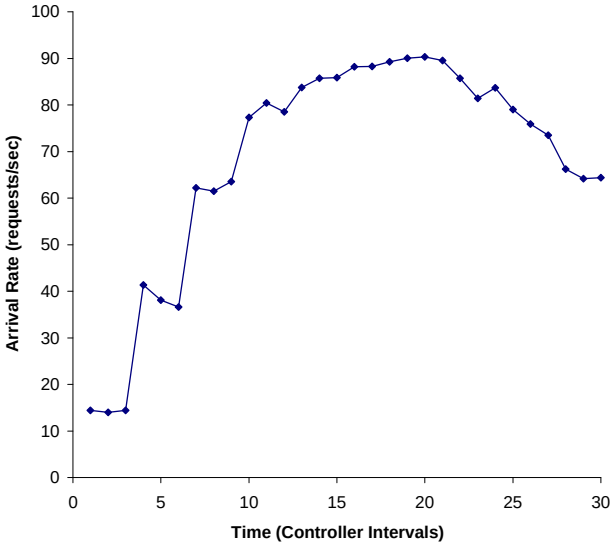

Figure 14: Variation of the Arrival Rate During the Experiments.

Figure 15 shows the variation of the average QoS value over all experiments with the QoS controller enabled and disabled as a function of time measured in CIs. There are 30 points in each curve corresponding to each of the 30 CIs. The x-axis is labeled with the average value of the arrival rate in each CI.

The following important observations can be made:

- During the first 11 CIs, the arrival rate is in its increasing phase. There is virtually no difference in the QoS levels between the controlled and uncontrolled systems.
- As the arrival rate reaches its peak value, the QoS value of the uncontrolled system starts to decrease an enters negative territory, indicating violation of one or more of the QoS levels. The QoS for the controlled system manages to stay positive throughout the entire experiment.

To establish if there is statistical difference between the QoS values QoS_c with the controller and the QoS value QoS_u without the controller, we computed a 95% confidence interval for the mean of the differences $QoS_c - QoS_u$ between the QoS for the controlled and uncontrolled cases. This confidence interval is shown in Table 2 for each of the 30 CIs. The average value of the arrival rate is shown in column two of the table, the average difference between the QoS values in column three, and the 95% confidence interval in column four. If the confidence interval includes zero, then we cannot say that the two systems are different. This is indicated in the last column of the table, which shows that after CI 12 the controlled system is better than the uncontrolled one at a 95% confidence level.

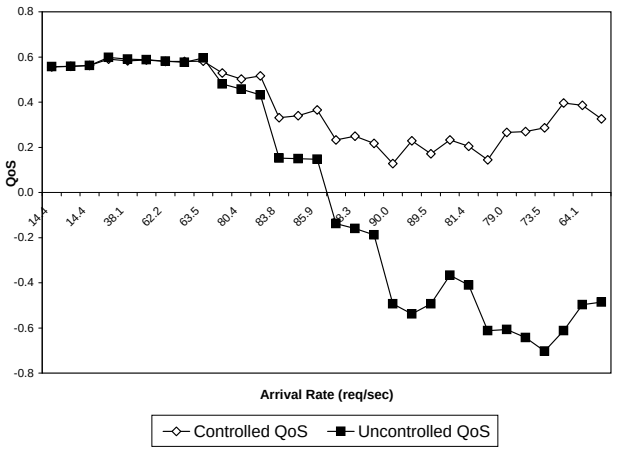

Figure 15: QoS Values With and Without Control.

CI	λ (req/sec)	$QoS_c - QoS_u$	95% conf. interval	zero in interval
1	14.4	-0.0001	(-0.0171, 0.0168)	Y
2	14.0	-0.0003	(-0.0133, 0.0128)	Y
3	14.4	0.0000	(-0.0112, 0.0111)	Y
4	41.3	-0.0073	(-0.0157, 0.0012)	Y
5	38.1	-0.0070	(-0.0167, 0.0028)	Y
6	36.6	-0.0018	(-0.0111, 0.0075)	Y
7	62.2	-0.0013	(-0.0186, 0.0161)	Y
8	61.5	0.0030	(-0.0169, 0.0229)	Y
9	63.5	-0.0148	(-0.0342, 0.0046)	Y
10	77.3	0.0487	(-0.0098, 0.1072)	Y
11	80.4	0.0455	(-0.0086, 0.0996)	Y
12	78.5	0.0847	(0.0044, 0.1650)	N
13	83.8	0.1859	(0.1034, 0.2685)	N
14	85.7	0.1841	(0.0702, 0.2980)	N
15	85.9	0.2190	(0.0948, 0.3431)	N
16	88.2	0.3714	(0.2273, 0.5156)	N
17	88.3	0.4083	(0.2692, 0.5474)	N
18	89.2	0.4046	(0.3261, 0.4832)	N
19	90.0	0.6211	(0.4943, 0.7478)	N
20	90.3	0.7667	(0.6106, 0.9228)	N
21	89.5	0.6652	(0.5612, 0.7692)	N
22	85.7	0.5984	(0.4401, 0.7567)	N
23	81.4	0.6146	(0.3572, 0.8720)	N
24	83.7	0.7561	(0.4674, 1.0447)	N
25	79.0	0.8729	(0.5753, 1.1705)	N
26	75.9	0.9116	(0.5992, 1.2241)	N
27	73.5	0.8900	(0.5734, 1.2065)	N
28	66.2	0.8582	(0.5413, 1.1751)	N
29	64.1	0.7330	(0.3649, 1.1011)	N
30	64.4	0.6406	(0.3288, 0.9524)	N

Table 2: Difference Between QoS Values with and Without the QoS Controller.

7. CONCLUSIONS AND FUTURE WORK

This paper presented a novel framework to control the QoS of e-commerce sites. This approach can be applied to dynamically change the values of any configuration parameters that can be changed at run time. Examples include maximum number of connections, number of threads, cache sizes, load balancing policies and parameters. The technique uses a combination of heuristic optimization techniques guided by predictive queuing network models.

With the aid of a very detailed prototype of a three-tiered e-commerce site that follows the TPC-W specification guidelines, we have been able to demonstrate the usefulness of the approach. In particular, we showed that as the arrival rate increases and reaches its peak value, the QoS of the controlled system manages to remain positive, thereby meeting the QoS requirements, while that of the uncontrolled site goes deeply into negative territory.

We plan to continue our work in several important directions. First, it is worth exploring the space of configuration parameters to discover other controllable features that can impact QoS and therefore can be subject to tuning. Secondly, our current implementation is *reactive*. That is, the system looks at the current workload and configuration parameters, decides if a new operational point needs to be found, and in that case proceeds to greedily find a better one through hill-climbing. We plan to apply forecasting techniques [1] in an attempt to anticipate workload variations in the near future and let the system adjust to those estimated variations before they occur. One idea is to use workload data (as generated by the workload generator) to define clusters of workload points and track their movement (using techniques as the one described in [2]) to come up with models of how the workload evolves in time. These models can then be used to predict the type of workload that will be received by the site in the near future.

Hill-climbing was used in these experiments due to the speed at which the solutions can be found. However, it is well-known that hill-climbing can get stuck in a local optimal point, rendering a sub-optimal solution. In our case, this characteristic would pose a problem if this sub-optimal solution happens to violate the QoS parameters (which did not occur in our experiments, but it is a distinct possibility). Alternative ways of searching for improved solutions have to be tried and compared with hill-climbing. Among them, we will try the classic *depth first search, breadth first search,* and *best first search* techniques, which explore possible states using different traversal algorithms. For that, the search state has to be organized like a graph, in which nodes represent states (parameter vectors) and edges connect nodes that represent adjacent parameter assignments (e.g., moving one unit for one of the parameter values, such as number of threads).

Acknowledgements

This research was partially funded by Virginia's Center for Innovative Technology (CIT) under award no. INF-00-022 and by the TRW Foundation.

8. REFERENCES

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

[2] D. Barbará and P. Chen, "Tracking Clusters in Evolving Data Sets," *Proceedings of the 14th AAAI International Flairs Conference*, Key West, FL, 2001.

[3] V. Almeida, M. Crovella, A. Bestavros, and A. Oliveira "Characterizing Reference Locality in the WWW," *Proc. IEEE/ACM International Conference on Parallel and Distributed System (PDIS)*, December 1996.

[4] M. Arlitt and C. Williamson, "Web Server Workload Characterization," *Proc. 1996 SIGMETRICS Conference on Measurement of Computer Systems*, ACM, May 1996.

[5] L. Cherkasova and P. Phaal, "Session Based Admission Control: A Mechanism for Improving the Performance of an Overloaded Web Server," HPL-98-119, HP Labs Technical Reports, 1998.

[6] M. Crovella and A. Bestavros, "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Transactions on Networking*, 5(6), pp. 835–846, December 1997.

[7] Hewlett Packard, WebQos, www.hp.com/products1/webqos/products/

[8] L. Kleinrock, Queueing Systems, Vol. I, John Wiley, NY, 1975.

[9] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the self-similar nature of Ethernet traffic (extended version)," *IEEE/ACM Trans. Networking*, pp. 1–15, 1994.

[10] J. D. Little, "A proof of the queuing formula L = λ *W*," *Operations Research*, Vol. 9, 1961, pp. 383-387.

[11] D. Menascé and V. Almeida, *Capacity Planning for Web Performance: metrics, models, and methods*, Prentice Hall, Upper Saddle River, NJ, 1998.

[12] D. A. Menascé, V. Almeida, R. Fonseca, and M. Mendes, "A Methoodology for Workload Characterization for E-Commerce Servers," *Proc. 1999 ACM Conference in Electronic Commerce*, Denver, CO, Nov. 3-5, pp 119-128.

[13] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes, "Business-oriented Resource Management Policies for E-Commerce Servers," *Performance Evaluation*, September 2000.

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

[15] D. A. Menascé, V. A. F. Almeida, R. Riedi, F. Pelegrinelli, R. Fonseca, and W. Meira Jr., "In Search of Invariants for E-Business Workloads," Proc. Second ACM Conference on Electronic Commerce, Minneapolis, MN, October 17-20, 2000.

[16] Peakstone Corporation, www.peakstone.com.

[17] Pitkow, J., Summary of WWW characterizations, *World Wide Web*, No. 2, 1999.

[18] M Reiser and S. Lavenberg, "Mean-Value Analysis of Closed-Multi Chain Queueing Networks," J. ACM, vol. 27, no. 2, 1980.

[19] Sun Microsystems, High End Servers, Sun Enterprise 10000, www.sun.com/servers/highend/

[20] Transaction Processing Council, The TPC-W Benchmark, www.tpc.org.