Proc. 1999 Computer Measurement Group Conference (CMG'99), Reno, NV, Dec. 1999.

PLANNING THE CAPACITY OF A WEB SERVER: AN EXPERIENCE REPORT

Daniel A. Menascé
Department of Computer Science, MS 4A5
George Mason University
Fairfax, VA 22030-4444
Menasce@cs.gmu.edu

Robert Peraino
University Computing
George Mason University
Fairfax, VA 22030-4444
Peraino@gmu.edu

Nikki Dinh
SRA International, Inc.
4300 Fair Lakes Court
Fairfax, VA 22033-4232
nikki_dinh@sra.com

Quan T. Dinh
Lockheed Martin Federal Systems, Inc.
9500 Godwin Dr.
Manassas, VA 20110-4157
Quan.t.dinh@lmco.com

Web servers have become a key aspect of most mission-critical applications. This paper describes our experience in planning the capacity of a production server at our organization. The methodology used here is model-based and allows for performance prediction. The paper describes how each step of the methodology was carried out with special emphasis in workload characterization, performance model development, validation, and calibration.

1. Introduction

At present, George Mason University does not have a good understanding of its web service workload, or the subsequent resource requirements created by this workload. As a result, the university can neither quantify current resource utilization, nor predict future resource needs. Through the use of a model-based capacity planning methodology [MA98], we seek to paint an analytical “picture” of the current web service workload, and the resulting resource utilization. We will also predict maximum throughput, and suggest changes that will increase maximum throughput. This paper will step through the stages of the model-based capacity planning methodology, to include environmental description, workload characterization, workload model development and validation, performance model development, performance model validation and calibration, and performance prediction.

As the World Wide Web has become an integral part of our society, www.gmu.edu has become one of the most important, and most visible representatives of the university. The server (whose formal name is **jiju**) is a 24-by-7 operation. Because it is a highly visible production server, if jiju were to be down for any length of time, phones would start ringing almost immediately. This need for constant availability severely limited our ability to perform benchmarks and other activities that would normally occur on off-line equipment. Another challenge to our analysis is the constantly changing nature of jiju. Just prior to the start of our analysis, the administrator wanted to upgrade the server software. Since this might change the performance behavior of the server, we opted to wait for the upgrade to be completed before starting our analysis. And as we were finishing our analysis, we were pressured to do so quickly, since another software upgrade was planned, as well as memory and RAID disk upgrades. This is the volatile nature of mission-critical components.

In this paper, we use the capacity planning methodology described in [MA98] to plan the capacity of a public Web server. The methodology consists of the following steps: understanding the environment, workload characterization, performance model development, model validation, performance prediction, workload forecasting, and cost-performance analysis. We describe in the following sections our experience with applying the methodology.

2. Understanding the Environment

This first step in the methodology is concerned with describing the hardware, the software, the connectivity of the server, and the peak usage periods. We gathered our initial information by interviewing the administration to get an overview of the hardware and software configuration, and the services provided.

Figure 1 details the hardware of the server which consists of a dual 168 MHz processor Sun Server with 256 MB of RAM running Solaris 2.6. The HTTP server is Apache version 1.3.4. The storage subsystem is composed of three disks. The operating system and the logs it generates are stored on disk 1, the HTTP access and error logs are stored on disk 2, and the document tree is stored on disk 3.

The machine is dedicated to being a Web server. The only persons who would log directly into the system are the administrators.

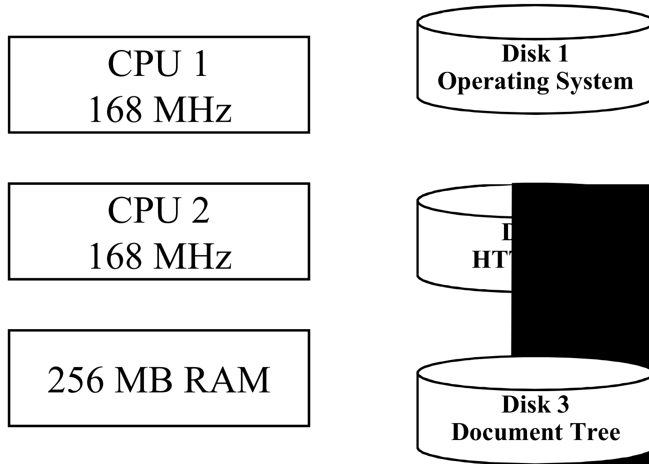

Figure 1 - Hardware Configuration of the Server

Figure 2 illustrates the connectivity. Jiju is connected to a local network via a 100Mbps Ethernet interface using the TCP/IP protocols. This interface is connected directly to a router, and then out to the Internet via a T3 line. Jiju is a publicly accessible server; 50 percent of all requests originate off-campus. Our analysis concentrates on the server plus the link that connects the server to the Internet.

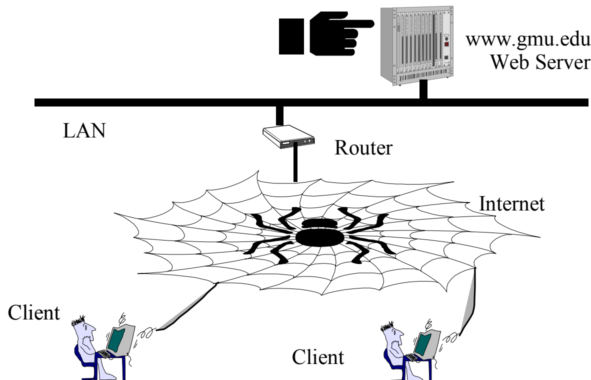

Figure 2 - Server Connectivity

In order to determine the peak usage periods, we analyzed the web server access log for the month of February, which consists of 13.2 million requests. For each minute of the month, an average arrival rate was calculated, in transactions per second (tps). This represents approximately 40,000 samples. Figure 3 is a graph of the tps rate for every minute of the month. The X-axis is labeled by hour of day. We can clearly see that the peak hours are from 9 AM to 5 PM. The mean tps rate over the entire month is 5.49 tps. The maximum rate, averaged over one minute, is 28.4 tps. The minimum rate is 0.017 tps. This means that the server was never idle, over any one-minute interval.

To further understand the data, we added a third dimension to the data, which is Day of the Week (0 to 6 in Figure 4; 0 is Monday and 6 is Sunday). We could then see how the usage fluctuates throughout the week. From the figure it becomes clear that weekends represent periods of lower activity than weekdays. The peak period was determined to be weekdays from 9 AM to 5PM

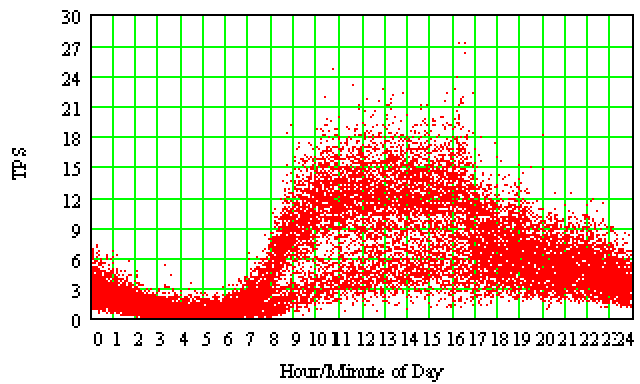

Figure 3 - Workload Intensity (in tps) for the Month of February

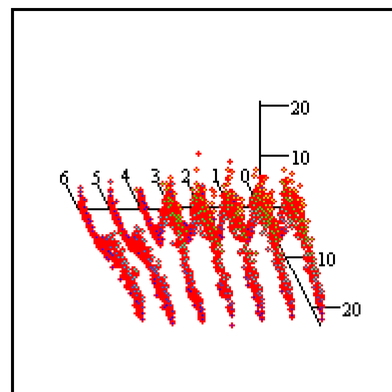

Figure 4 - Workload Intensity by Hour of the Day and Day of the Week.

3. Workload Characterization

In this step of the capacity planning methodology, we describe the main components of the global workload, the decomposition into classes, and how we arrived at these results.

3.1 Identifying the Basic Component

The service provided by jiju supports the HTTP Protocol. This protocol defines a series of request types, with ancillary data. The basic component of this system is the HTTP GET request. Analysis of one month of the Apache web server access logs shows that 99.72 percent of all activity is of this type.

3.2 Choice of the Characterizing Parameters

The choice of the characterizing parameters is straightforward. The workload intensity of requests is characterized by the arrival rate, measured in transactions per second (tps). Service demands are computed for the three disks, and the two CPUs.

3.3 Partitioning the Workload

For the purpose of making the workload model more representative, we partitioned the workload according to file sizes. Utilizing the web server access log for the peak periods as defined in section 2, we were able to obtain file sizes and frequencies. We wanted to see if there were areas of file size concentrations that would make the partitions obvious. The data was so disparate as to make any visual display useless. We then implemented the minimum spanning tree (MST) [MAD94, MA98] clustering algorithm so that we could perform a one-dimensional clustering on file sizes. We initially ran the clustering algorithm with 150 clusters. Upon inspection, we noticed that the top 4 clusters represented 98 percent of all requests. Additionally, zero-length transfers were the result of 45.76 percent of all requests. A request may report a zero-length transfer for several reasons including:

- The version in the client cache is up to date.
- The requested file was not found.
- An access violation prevented file transfer.

In each of these cases, the request must be processed, which creates CPU demand, and a search must be done for the file, which creates disk demand. So, the zero-length transfers are a very real part of the system activity. Using the clustering results as a starting point, along with the above observations, we decided on four classes of requests, as shown in Table 1.

Class	Range (Bytes)	% Requests	% Data	Derived from
1	0	45.76	0.00	From cluster 0
2	1 - 4372	43.47	23.09	Remainder of cluster 0
3	4373 - 19284	8.78	35.41	Clusters 1, 2, and 3
4	> 19284	1.99	41.49	Remainder of clusters

Table 1 - Class Characterization.

Note that 10.77% of the requests (classes 3 and 4) are responsible for retrieving about 77% of the data. This is typical of Web servers [AW96,CB96].

3.4 Data Collection

This section discusses the tools used for data collection and describes how they were used to obtain the service demands at the CPU and disks.

3.4.1 Tools Used for Data Collection

From the perspective of the system, we used `iostat` and `process accounting`. `iostat` provided us with the necessary device utilization for both the CPU and disks. `Process accounting` provided us with a very detailed look at the system activity. This allowed us to ensure that we accounted for all major resource usage.

From the perspective of the Apache server, we relied on the server access log. In determining what our data requirements would be, we decided that the existing access log did not provide sufficient information about server activity. Since the source code for Apache is available, we could instrument the server, to write additional information into the log. By mid-March, the following additional fields were being recorded in the production log: PID, service time, CPU time, and system date/time. We explain each of these additional fields in what follows.

PID: the Process ID. The Apache server works in a master/slave arrangement. A master process spawns slave tasks to service requests. When a slave server has handled a certain pre-configured number of requests, the master kills the slave and starts a new one in its place. The intent behind this design is to avoid memory leaks in key system functions, on certain OS platforms. Our intention in recording the PID of the slave on each log record was to be able to find all requests handled by a single slave, and then match that slave to a process accounting record.

Service Time. Also defined as the server-side response time, this is the elapsed time in microseconds, from the start of processing of a request, until the data is completely sent. The intent of recording this was to have a direct measure of response time for validation of the performance model, and for benchmarking purposes.

CPU Time. This is the accumulated CPU time of the request, in milliseconds. This field allows us to have a direct measure of CPU service demand for each request. Our plan was to find the average CPU service demand for each class of request. It turns out that this field proved useless as explained in section 3.5.1.

System Date/Time in Seconds. Although the date and time is recorded in the HTTP log, it is not recorded in a format useful for time manipulation. This field records the system internal clock in accumulated

seconds, since Time of Epoch. This allows us to find the elapsed time between any two log records by simply subtracting the time values, and was used to compute request inter-arrival times.

3.5 Computing Service Demands

The service demand $D_{i,r}$ of requests of class r at device i is defined as the total service time of requests of that class at that device [MAD94, MA98] and can be computed as

$$D_{i,r} = \frac{U_{i,r} \times T}{C_r} \quad (1)$$

where $U_{i,r}$ is the utilization of device i due to requests of class r, T is the interval during which the utilization was measured and C_r is the number of requests of class r processed in the interval of duration T. While it is easy to obtain the overall utilization of a device, it is not always easy to obtain the utilization of that device due to a specific type of request, especially when the server is a production server that cannot be taken offline for running controlled benchmarks. Also, we did not have access to any similar machine on which we could run equivalent benchmarks. This placed a tremendous limitation on our options for computing service demands for the resources and classes. We decided to rely on a rule of proportionality for certain resources. In other words, we assumed, that for certain devices, the utilization of a specific class at that device is proportional to the fraction of data retrieved by that class from the total number of bytes retrieved. We also decided to distribute system overhead proportionally among the classes.

The utilization of system resources was recorded for one interval of the peak period time window. The service demands for each resource and class were derived from these utilization values based on the formula:

$$D_{i,r} = U_i \times \frac{f_r \times T}{C_r} \quad (2)$$

where, $D_{i,r}$ is the service demand of requests of class r at resource i, U_i is the total utilization of resource i (obtained from `iostat`), T the measurement period, C_r is the number of requests of class r processed during the interval (obtained from the HTTP log), and f_r is a proportionality factor that indicates the fraction of the busy period of resource i that can be attributed to class r.

3.5.1 CPU Service Demand

At first, we planned to derive the CPU service demands directly from the instrumented server log. We were disappointed to discover that this would not be possible due to the granularity with which the system records CPU time accumulation. The system can only update the CPU time at the end of a process timeslice. On jiju, this time period is 10 msec. Testing verified that CPU time would always be reported in increments of 10 msec. Unfortunately, the service demand of most requests is less than 10 msec. As a result, the time reported in the log for small files is either 0 msec or 10 msec, neither of which is correct. We decided to work with the premise that the CPU service demand is proportional to the file size. To test this supposition, we benchmarked the CPU time required to request a set of files of increasing size. The files had to be of sufficient size such that the granularity of the CPU time reported by the system would not have a significant impact on the measurements. We benchmarked files of size 1 MB to 10MB in increments of 1MB. Each file size was benchmarked 10 times. We then performed a linear regression on the resulting data.

Figure 5 shows the results of the benchmarks, and the linear regression. Assuming that CPU demand is proportional to file size implies that class 1 (zero-length files) requires no time. But those requests must use **some** time, so we rely on the linear regression and set the service demand to be the y-intercept, which would be the expected value for a file of zero bytes. To calculate the service demands for the other classes, we use Eq. (2), but we first must subtract from the CPU busy time ($2 \times U_{CPU} \times T$) the activity represented by class 1, which was calculated independently. Also, the factor f_r is the percentage of bytes b_r retrieved by class r requests. So the CPU service demand is given by

$$D_{cpu,r} = \frac{\left(2 \times U_{cpu} \times T - D_{cpu,1} \times C_1\right) \times b_r}{C_r} \quad (3)$$

where C_1 is the number of class 1 requests executed during time T. Note that the factor 2 in the total CPU time ($2 \times U_{CPU} \times T$) is due to the fact that the server has two CPUs and the CPU utilization reported by `iostat` is the average utilization of the two processors.

3.5.2 Disk 1 Service Demand

Disk 1 contains the operating system and its support files such as the process accounting log. As mentioned in section 3.4.1, slave server processes are constantly being created and killed. When a process ends, a process accounting log record is written. Therefore, the higher the arrival rate of requests, more process accounting records are generated. Instead of a proportional distribution of activity, each class has the same service demand since all accounting records

are the same size regardless of class. The service demand for disk 1 is then computed as

$$D_{disk1,r} = \frac{U_{disk1} \times T}{\sum_{r=1}^{4} C_r} \qquad \forall r \qquad (4)$$

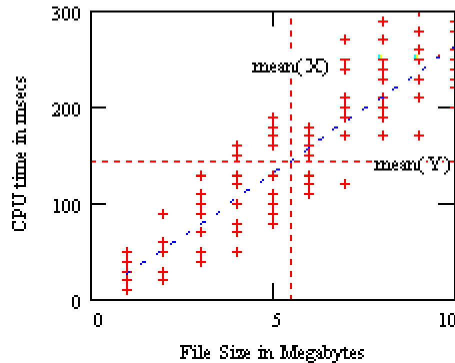

Figure 5 - CPU time (in msec) versus file size (in Mbytes)

3.5.3 Disk 2 Service Demand

Disk 2 holds the web server log. Just like disk1, the service demand necessary to write a log record is the same for all requests, regardless of class. So the service demand is computed in the same manner as disk 1.

3.5.4 Disk 3 Service Demand

Disk 3 is the server file repository. Service demands must be derived in a manner similar to the CPU demand. As was the case with the CPU service demand, the class 1 service demand could not be based on the class 1 proportion of data retrieved, since the class 1 file size is zero. And as mentioned before, it was not possible to take the machine off-line for a controlled benchmark. To get a rough estimate of the service demand of disk 3 for class 1, we performed an on-line benchmark. Considering the reasons why a request would fall into this class (mentioned in section 3.3), the benchmark merely needed to search for files, and not open them. The benchmark accessed the I-node information for all files in several file systems, while the disk utilization was measured with `iostat`. This was performed by issuing the UNIX command `'ls -lR'` over entire file systems, monitoring the utilization of the disk. We used entire file systems to avoid picking up anything in the cache. We used the long listing (`-l`) option, to force the system to look at the file i-node, which should be equivalent to the amount of work performed by 0-length transfers. After the run, we counted the number of files in the file system. This would become C_1. We then computed the service demand at disk 3 for class 0 as $D=U^*(T/C_1)$. We did this for three different file systems on jiju and averaged the results. The remaining class service demands were computed using the same method as the CPU classes.

The resulting service demands are then given in Table 2. Note that we have added the incoming link and outgoing links as additional devices. The service demand at these devices was obtained from the file size for each class and from the protocol (TCP/IP + HTTP) overhead incurred in transmitting files of that size.

	1	2	3	4
CPU 1	0.400	8.520	64.760	334.400
CPU 2	0.400	8.520	64.760	334.400
Disk1	0.740	0.740	0.740	0.740
Disk2	0.670	0.670	0.670	0.670
Disk3	1.740	1.840	13.990	72.250
In-link	0.036	0.036	0.036	0.036
Out-link	0.000	0.409	2.103	5.333

Table 2 - Service demands (msec)

4. Workload Model Validation

Workload model validation would entail the development of a synthetic workload based on the workload characterization, which would be run on the system so that performance results could be compared to production performance results. Because jiju is a production server, we were not permitted to bring down the server so that we could run a synthetic workload. Had we been able to do this, we would have built a script that would submit synthetic requests of four types based on the four classes. The file sizes would be based on the average file sizes for each class. The distribution of the requests would be based on the proportion of requests that each class represents.

5. Performance Model Development and Performance Prediction

The performance model used to represent the server and the incoming and outgoing links is a multiclass open queuing network (QN) model (see Fig. 6). To model the dual processor CPU we used the approximation suggested by Seidman et al [SSS87].

Figure 7 shows the average response time for the four classes as a function of the aggregate arrival rate of requests. The proportion of requests in each class is the same shown in Table 1. The model shows that after 120 tps, the utilization of the CPU reaches 100%. Class 4 is the one with the largest response time and with the largest growth rate in response time.

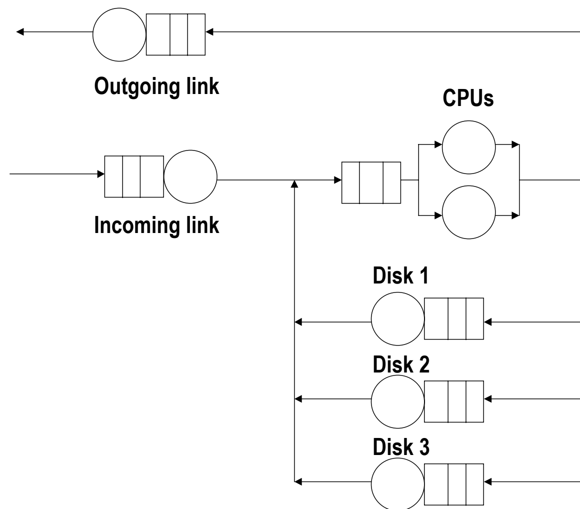

Figure 6- QN model for the Web server.

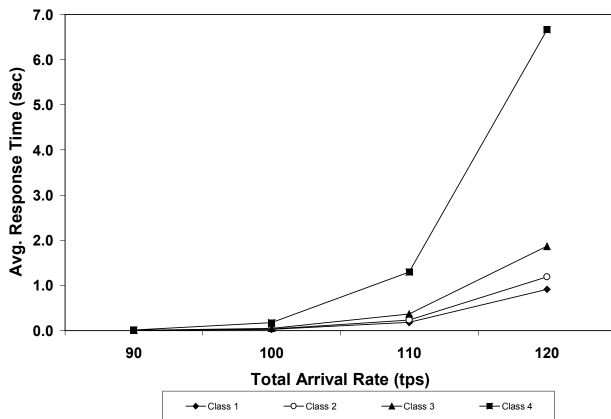

Figure 7 – Response Time vs. Arrival Rate

6. Performance Model Validation and Calibration

In this step, we compared the model predictions with the actual reported system activity. We describe the validation problems and how we fixed them.

When we tested the model for the first time, the results were wildly inaccurate, and inconsistent. Each set of data we ran from different days provided differing amounts of error. So we turned back to our environmental review, to see what we had missed.

To get a closer look at the system activity, we were permitted by the system administrator to review the process accounting logs and the `cron` configuration, to get a much more precise look at the system activity. From the process accounting log review, we identified a process that used a tremendous amount of resources, and ran every hour. We traced the process to a job that was scheduled in the `cron` configuration. The job tabulated usage statistics by reading the entire access log every hour. As the log grew in size, so did the resource requirements of this process. Thus, the resource requirements were different for every hour of every day of the month. Since the access log is restarted only at the end of the month, it will grow to about 1.5 gigabytes in size before being removed. As a result, by the end of the month, this statistics job required 45 minutes of CPU time each hour, which is much more than even the server requires! We estimated that the statistics job accounts for approximately 37 percent of the reported aggregate CPU utilization, and this is consistent with the activity we were seeing with `iostat`. Before we decided how to incorporate this activity into the model, we brought our findings to the attention of the administration. They did not know that the job was running, and determined that it only needed to run once per week, during off-hours. The rescheduling of this job precluded any need to include it in the model. We felt that we achieved much success already, since we saved the university almost an entire processor!

Once the statistics usage job was addressed, our model accuracy in terms of utilization became quite acceptable. The maximum percent absolute error in the utilization was 6.5% and that happened for class 2.

We planned to validate the model response time by comparing it to the recorded server-side response time as defined in section 3.4.1. We were terribly disappointed to discover that although the response was being accurately recorded, it was not strictly server-side response time. This effect was a characteristic of being on an open network. When we randomly sampled the log, we found that response times could vary from 0.5 seconds to 60 seconds for the same file size. Causes for this behavior include network sluggishness, timeouts, and slow modem speed at the client. Of course, the response times reported by the model do not consider these issues.

Slow modem speed is probably the single greatest cause of response time variability. It should be noted that response time cannot be recorded in the HTTP log until all of the data for a request is sent. If the client request is coming over low speed modem connections, this will limit the effective bandwidth of the TCP connection used to send the results back to the client and will therefore increase response time.

To study the effects on response time of the client access speed, we developed a mixed QN model with one closed class representing a client making requests and four open classes identical to the ones in the previous model (see Fig. 8). Figure 8 shows three delay devices, represented by circles, corresponding to the browser, client access link, and the Internet. The service demand at the browser is the client think time.

The service demand at the client access link is the total time to transmit requests and receive replies. It is assumed that client access link is dedicated, as is the case with modem access. When this is not true, as in the case of LAN-based access, we used the effective bandwidth seen by the client to compute the service demand. Using the solution methods for mixed QNs given in [MAD94], we obtained the response time as a function of the client-side access speed. This is depicted in Fig. 9 for class 4 for an aggregate arrival rate for the open classes equal to 120 tps and for an Internet round trip delay of 200 msec. Response times vary from 9.51 sec to 6.70 sec depending on the client-side access speed. Note that for the same arrival rate of 120 tps, the server-side response time obtained from the open QN model is 6.66 sec. This model explains the large variability in response time observed in the log. The response times for requests coming from campus over a high speed ATM backbone and 100 Mbps LANs only were more in line with those predicted by the open QN model.

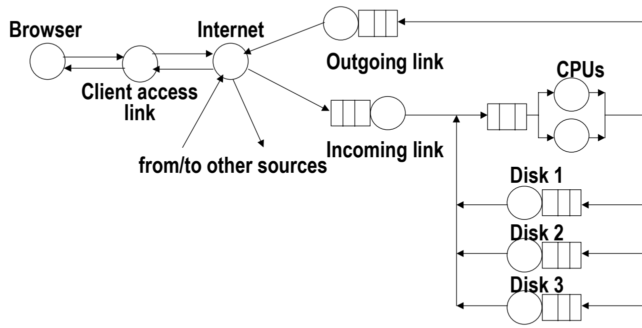

Figure 8 – Mixed QN Model

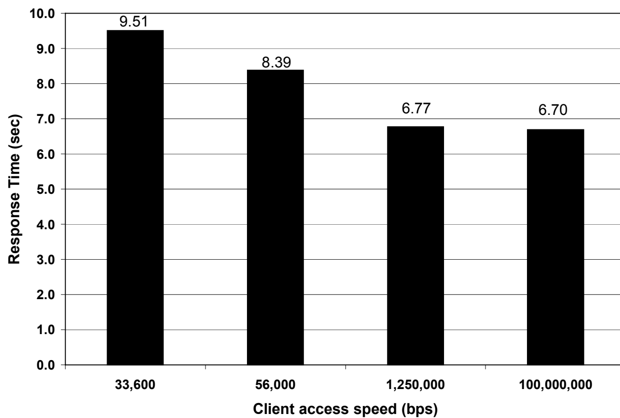

Figure 9 – Response Time vs. Client-side Access Speed.

Our analytic model showed a saturation around 120 tps. We wanted to verify through measurements if this was indeed the case. To that end, we ran a benchmark using the *InetLoad* benchmark tool [InetLoad] to drive the server to its maximum capacity, determined by a CPU, our bottleneck, close to 100% utilized. We chose this tool because it provided two key features; we could use actual log activity to drive the benchmark, and we could mimic the arrival rate characteristics, which would maintain the burstiness of our workload. In fact, if we look at a plot of the arrival rate over time we see clearly the bursty nature of the workload. Figure 10 illustrates the effects of burstiness. This graph is a 191-second interval randomly taken from the access log on April 26, 1999 at 15:37:05, showing the arrival rate in requests per second. The burstiness is clearly visible. While the average arrival rate over this interval is only 13.53 requests per second, the bursts are considerably higher.

We generated an *InetLoad* script by processing the production log to obtain the file references and the request inter-arrival times. Bursts of requests are handled in the following manner: if 10 requests arrived within one second, the script will ensure that the benchmark will request those 10 files within one second. This is how we maintain the exact nature of the workload. The arrival rate can be increased without compromising the arrival rate characteristics, by "compressing" the real time interval during which an interval of log activity will be performed. This can be accomplished by reducing the inter-arrival time.

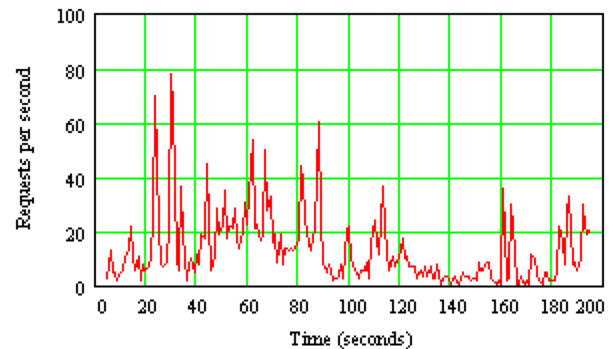

Figure 10- Bursty Nature of the Workload.

We ran our benchmark several times and averaged the throughput obtained. Simultaneously with each run we had iostat active so that we could verify that both CPUs were close to 100% utilized. The average throughput observed was 131 tps, 9.2% above the maximum throughput predicted by our model.

To determine the maximum arrival rate under non-bursty conditions, we used *InetLoad* to perform a benchmark consisting of requests submitted at a constant rate. The non-bursty benchmark was able to drive the server at 202 tps, well above the values obtained with the bursty workload. These findings corroborate the ones given in [Banga97].

Additional discussion of load testing can be found in [Michalsky98].

7. Answering What-If Questions

The university is considering moving the Web server to a machine with RAID disks for increased fault tolerance and a different processor. One of the machines being considered by the administration had two slightly slower processors. We advised against it knowing that the CPU was the bottleneck. The other option had to 250 MHz processors. To obtain a very rough estimate of the gains to be achieved by using the faster processors we ran our model again. We ran our model again by multiplying the CPU service demand by 168/250 = 0.672. To account for the increase in processor speed. We could also have used the ratio of SPECint95 ratings. For an arrival rate value of 120 ts, class 4 now has a response time of 0.592 sec as opposed to 6.6 sec with the slower processors. Also the maximum throughput supported by the Web server increases to 183 tps, an increase of about 52% with respect to the previous situation. So, our recommendation to the administration was to upgrade to the configuration with two 250 MHz processors.

8. Concluding Remarks

The purpose of this paper was to present a very practical view of how we applied a model-based capacity planning methodology to a production web-server. We also discussed where things went apparently wrong and what steps had to be taken to validate and calibrate the model. One of the biggest challenges in any capacity planning and performance modeling study is making sure one has the right data, both for the purpose of computing input parameters to analytic models, as well as for validating model outputs such throughput, utilization, and response times.

References

[AW96] Arlitt, M. and C. Williamson, Web Server Workload Characterization: the search for invariants, Proc. 1996 ACM Sigmetrics Conference, Philadelphia, May 1996.

[Banga97] Banga, G. and P. Druschel, Measuring the Capacity of a Web Server, Usenix Symposium on Internet Technologies and Systems, December 1997.

[CB96] Crovella, M. and A. Bestravos, Self-Similarity in World Wide Web Traffic: Evidence and Causes, Proc. 1996 ACM Sigmetrics Conference, Philadelphia, May 1996.

[InetLoad] http://www.microsoft.com/msdownload/inetload/inetload.htm

[MENA94] 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.

[MENA98] Menascé, D. A. and V. A. F. Almeida, *Capacity Planning for Web Performance: metrics, models, and methods*, Prentice Hall, Upper Saddle River, NJ, 1998.

[Michalsky98] Michalsky, R. J., Load Testing in an Internet World, Proc. 1998 CMG Conference, Dec. 6-11, Anaheim, CA, pp. 291-299.

[SSS87] Seidman, A., P. Schweitzer, and S. Shalev-Oren, Computerized closed queueing network models of flexible manufacturing systems, *Large Scale Syst. J.*, North Hollland, vol. 12, pp. 91-107, 1987.