QoS in Parallelized E-Commerce Systems

Bruno Diniz Rodrigo Pereira Wagner Meira Jr. Virgílio Almeida

[1] e-SPEED - Computer Science Department
Federal University of Minas Gerais
Belo Horizonte — Brazil
{diniz,rpereira,meira,virgilio}@dcc.ufmg.br

August 2000

Abstract—
The proliferation and variety of e-commerce services make the maintenance of the quality of the services (QoS) provided an essential feature for successful sites. In this paper we propose an application-level strategy for differentiating e-commerce services that is based on variating the number of threads employed for satisfying a request. In order to validate and evaluate our strategy, we implemented an ad server that provides two levels of service. For sake of service differentiation, our strategy showed to be effective, allowing high-priority requests to be answered up to 60% faster than low-priority requests, although the differentiation reduces as the workload that is submitted to the server increases.

Keywords— **WWW, Internet, Parallel Computing, Quality of Service.**

I. Introduction

The Internet grew above all expectations in a short period of time. Current Internet technology allows all kinds of interactions and the execution of diverse tasks, including commercial transactions. In particular, most of the companies are discovering the Internet as a way to expand their market, taking advantage of the large number of potential customers in the network. For instance, a recent research among Brazilian Internet users pointed a current user population of almost 5 million, and 28 million potential users, which is greater than the overall population of whole countries. In parallel, the amount of information available in the Internet is also increasing, making easier to get information hardly reached before [Var95].

A similar trend is also being observed in e-commerce, which has been also growing. The strategy of the companies offering products through the Internet is to expand the number of services provided to users, in order to make them faithful and to get more and more traffic. The variety and efficiency of services are becoming essential to be succesfull in this information economy, however the user-perceived latency can become a really big problem. Users don't want to wait anymore. Instead, they want a high quality service with a small response time. As a result, assuring quality of service (QoS) in e-commerce services is becoming an important issue for an effective Internet-based commercial strategy.

The very first issue in differentiating services is to identify what are the levels to be provided and how we identify to which level each user is assigned [MAFM99a]. A common strategy in this case is to cluster users into groups that share similar behavior, and each group is assigned to a different service level according to their potential for generating profit, that is, more profitable users are granted a higher priority [MAFM99b, MAFM99a].

A second issue in providing differentiated quality of service is the system support necessary. In this case, we need to define the premises that rule the e-commerce server behavior, such as the service level agreements and the strategy when the server gets overloaded (e.g., discard new requests or low-profit users). In all cases, the implementation of such schemes is difficult because almost all commercial operating systems do not support differentiated services and guarantees regarding resource allocation of processing time, disk and network bandwidth. There are some recent work towards supporting QoS primitives in commercial Unix systems, which are discussed in Section I.A.

This paper discusses the implementation of differentiated services in e-commerce servers without employing any OS support other than threads. Basically, we verify how effective a pure application-level strategy can be despite workload and task complexity. Our strategy is tested in a electronic ad server that has been parallelized, and provides two levels of service. In Section II, we present the architecture of e-commerce systems, including its modules and functionalities. Section III shows how QoS is being used to improve this e-commerce systems. In Section IV, we describe the ad server we implemented to validate a QoS policy based on thread scheduling, describing the experiments and analysing the results in Section V. Finally, in Section VI we draw some conclusions from our work and present what we are planning to do from now.

A. Related Work

As mentioned before, there has been several efforts towards supporting scalable and differentiated services in WWW servers. Most of the proposals are for novel mechanisms and primitives for supporting differentiated levels of services.

Flash [PDZ99] proposes a new web architecture, the

AMPED (asymmetric multi-process event-driven architecture), that combines the high performance of single-process event-driven servers with multi-threaded servers for diskbound workloads. Although the results are very good in terms of scalability, the use of traditional event-driven mechanisms prevent the implementation of differentiated level of services.

Crovella et. al. [CFHB99] shows that non-traditional service ordering is an effective strategy for improving the throughput of WWW servers. Their strategy is based on granting priorities to short connections, that is, connections associated with requests to small files. Again, the use of such strategy in e-commerce servers is difficult because the response-size variance is not as high as in traditional WWW servers, and the size of the response is usually not known in advance. Banga et. al. [BDM98, BDM99] proposed several improvements in the implementation of OS primitives that benefit WWW applications, such as resource containers. Resource containers are a general strategy that allows fine-grain management of resources in servers, making possible the establishment of several service differentiation policies. Although resource containers provide all primitives necessary for implementing differentiated QoS, they imply significant changes in the OS and are not available in any commercial systems. A similar strategy is proposed by Eclipse [BGOS98], but with the same drawbacks.

In [KaGM00], we investigated the impact of interrupted requests on e-commerce servers and verified that simple changes in the semantics of TCP protocols may result in significant improvements (reducing the server response time in up to 70%) in the overall performance of the server. We are not aware of any other work that target specifically ecommerce servers and one of our goals is to verify whether the characterizations that motivated the aforementioned efforts are still applicable to e-commerce services. Early results [PMA+00] show that there are several aspects that should be considered in e-commerce systems while supporting their execution efficiently. One of the goals of this paper is to assess such aspects, which shall be used as basis for novel QoS-related strategies.

II. E-COMMERCE SYSTEMS

Current e-commerce architectures can be divided in layers [BM99], where each layer represents a different service provided by an e-commerce application. A taxonomy of layers is shown in Figure 1 and described in the sections that follow.

A. Network Services

This is the layer where the network-related services (Internet, Virtual Private Network (VPN) or Extranet) are located. Among the services present in this layer, there are network protocol implementations, secure connections and network

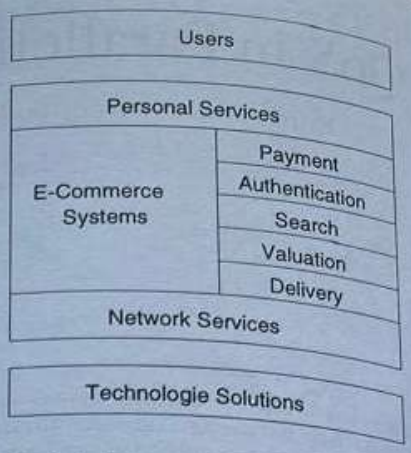

Fig. 1. Architecture of E-Commerce Systems

resource management. These services are the basis for ecommerce services, and should assure safety and reliability to the final user.

B. Commerce Services

This layer implements the business-related services that are provided by the server, such as search, payment, valuation and authentication, that is the services that implement the logic of the automated business. In this paper, we differentiated services provided in this layer, in particular the banner generation of an ad server, as described in Section IV.

C. Personal Services

Services in this layer are responsible for generating different responses to users, depending on their profile, actual conditions and criteries. As the the amount of information about a user increases, the amount of personalized services also increases. This layer is also essential to provide differentiated services, since the information gathered at this level is used to determine the quality of service provided by lower levels.

III. QOS IN E-COMMERCE SYSTEMS

In this section we discuss how the efforts mentioned in Section I.A can be used for implementing differentiated ecommerce services and present our approach.

The basic feature necessary for providing differentiated services is the ability to manage the allocation and usage of resources such as disks, memory, processors and network bandwidth. In this case, simple priority-based schemes do not suffice, since we may want to reserve 20% of the CPU processing time to a given group of users. An example of resource management strategy is shown in Figure 2.

We should also note the importance of differentiated level

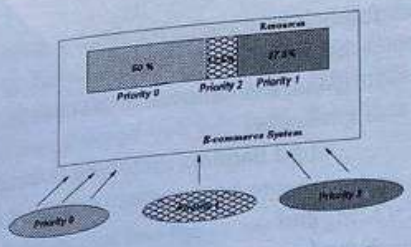

Fig. 2. E-Commerce System with QoS

of services in e-commerce systems. These servers are experiencing an explosive increase in the number of customers, eventually causing the degradation of the quality of the services provided, both in terms of response time and throughput. Another cause for QoS degradation is the the increasing variety of services, since the servers provide not only its own services, but a range of related services that, sometimes, are loosely related to the main activity of the automated business. In all cases, supporting various levels of service is demanded for a successful server.

A. QoS Using Thread Scheduling

Some of the proposals presented in Section I.A modify the kernel of the operating system to support QoS, changing scheduler-related criteria and parameters that are used to select which process will execute. In practice, the scheduler should guarantee that some process has the amount of resource usage desired like CPU time or network bandwidth.

Our goal in this work is to provide QoS without changing the kernel or being intrusive in the operating system execution. Our idea is to parallelize the services provided by the site and to differentiate the level of service provided by granting a different number of execution threads to each service, depending on the level it is assigned to. Thus, high level requests are performed by a greater number of threads than low level requests. This strategy benefits high level requests by reducing their elapsed response time, simulating a modified scheduler.

IV. Case Study: Ad Server

In this section we describe the ad server we implemented. This implementation is used to analyze the costs and the benefits of the inclusion of QoS primitives in an e-commerce application. We also present the server, its main features, and also discuss some QoS metrics used to compare the various approaches. It is worth to say that we didn't intend to implement a good ad server algorithm, that could be parallelized and executed faster. We used an ad server only to show that we can gain QoS primitives in the application level, with portability, based only on threads. Any other e-commerce server could have been used, since it had computationally intensive tasks to execute.

First of all, we describe how the ad server works. Without loss of generality, we identify three entities that interact with the ad server: clients, advertisers, and publishers. Advertisers are the companies that contract a number of impressions of their banners for a given time interval. These impressions may or may not be associated with a given set of concepts[1] that are semantically related to the nature of the banner. Publishers are the sites that place banners served by the ad server in their pages, usually being paid for that. Clients are users that request pages to publishers, and get banners from the adserver. These entities and their relationships are shown in Figure 3. We should note that each banner and each slot has a set of concepts, also called keywords, associated with them. These concepts are useful to determine which banners can be placed in a given slot. The parameters used to make this decision, including the concepts, are part of the request (an HTTP request) made by clients to the ad server.

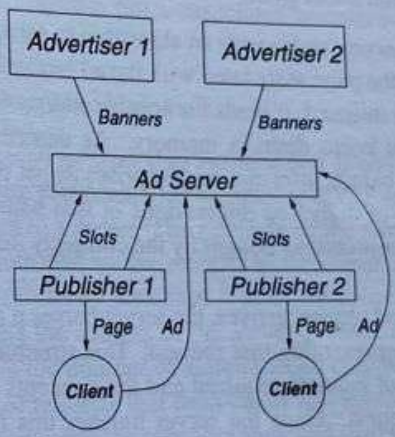

Fig. 3. Ad Server Architecture

From this short description of our operational scenario, we can define three basic tasks that are performed by the ad server:

Accept contracts: The server should handle contracts for placing banners, including concepts that should be associated to the banners. In this case it is important to verify whether there is impressiom capacity for the contracted advertisement.

Manage slots: The server should control the placement of banners for the possible slots in order to maximize the number of contracts and thus the profit generated by the server. Furthermore, it should also perform the accounting of placement and click-through as a measure of effectiveness of the publisher.

Provide banners: The ad server should answer to client requests promptly and consistently, that is, the banners

[1] concepts are keywords associated with banners or pages. For example, a car page in a web site could have the concepts "car", "road" and "drive" associated with it.

should not take long to be delivered and basic consistency criteria such as not placing repeated or paradoxal banners in the same page should be enforced. Temporal distribution of banners is also an issue while placing banners.

Another issue in implementing ad servers is the server architecture and how the components interact. In [DMA00], we describe, discuss, and evaluate three models to provide outsourcing services. All of them can be employed for implementing our ad server. In the current implementation, the communication with the ad server initiates through a HTTP request that specifies the generation parameters. The server's response provides the banner identifiers to be placed in the client page. The number of banners in the page, their type and other parameters, as previously mentioned, are passed as parameters of the HTTP request.

A. Algorithm Description

The ad server implements an algorithm to select a suitable banner for the page associated with the ad request. When the ad server is initiated, it loads the specifications of contracted banners and keeps them in memory. As described in Section IV, the banner information comprises its list of concepts, the number of delivered impressions of that banner and the number of impressions bought by the company that owns the banner.

When an ad request arrives, the server parses it and creates a structure representing that request. This structure contains the number of banners required and the concepts associated with that request. Once the server finishes this first phase, that is, creating a request object to represent the request (and thus the associated page), it starts the algorithm to determine which n banners will be placed in the page, where n is the number of slots in the same page.

The algorithm then determines the candidate banners for each concept, that is, banners that fulfill all requirements associated with the request, such as size and other slot features. All candidate banners are grouped in a common set and each banner is assigned to a weight that expresses its placement probability. In the current implementation of the ad server, the placement probability is proportional to the difference between the number of impressions contracted and the number of impressions delivered. In summary, the server tries to deliver all banners contracted homogeneously, given more weight to those banners that demand more impressions for contract completion. The last step is to select the banners, which is performed through a "roullette" strategy, where the space associated with each banner is its placement probability. A high level description of the algorithm is presented in Figure 4.

The implementation of the banner selection algorithm employed an array where we store, in each position, the difer-

```
1   foreach concept in request
2     begin
3     if there is banners with concept
4       begin
5       find banners matching concept;
6       merge banners into common list;
7       end
8     end
9   select banner from commom list;
```

Fig. 4. Banner selection algorithm

ence between the number of impressions contracted and the number of impressions delivered of the banner. The larger this diference, the larger the placement probability of the banner. The array is created cumulatively, that is, the second entry contains the probability of the first plus its probability and so on. After creating this array, we choose a randomically generated number greater than zero and smaller than the sum of all weights, that is equal to the probability value of the last position of the array. We then determine in which position of the array the generated number belongs. This position will be the index of the banner to be selected.

For example, assume that there are five banners contracted, b_1, b_2, b_3, b_4 and b_5, at a given time, their state (i.e., impressions contracted and delivered) is presented in Table I, and some client requested two banners, we would select the banners as follows:

Banner	Contracted	Delivered	Diff
b_1	10,000	3,240	6,760
b_2	8,000	2,560	5,440
b_3	20,000	4,210	15,790
b_4	30,000	1,250	28,750
b_5	25,000	6,430	18,570

TABLE I
STATE OF BANNERS AT THE TIME OF A REQUEST

With the values of the diference between the number of impressions bought and the number of impressions actually delivered, we build the array of weights. The resulting array for the example given is presented in Table II.

Pos.	0	1	2	3	4
Val.	6,760	12,220	27,990	56,740	75,310

TABLE II
EXAMPLE OF AN ARRAY OF WEIGHTS

We should observe that the values are acumulatives. In th

case, we choose a randomically generated number between 0 and 75.310. For example, suppose that we choose 10.534. Checking the array, we determine that the number is between the values of entry 0 and entry 1, indicating that the banner selected is entry 1.

This algorithm is efficient because, theoretically, it chooses the banner that has the larger weight. Ocasionally, it may choose a banner with less weight, but as requests arrive, the "space" that previously delivered banners use decreases, as well as their weight in the array. We evaluted this algorithm through some tests and experiments, and the result was satisfactory, that is, it distributed the banners evenly across time. In spite of being statistical-based, the algorithm provided real gains because of its adaptative nature.

B. Parallelized Algorithm

By observing the algorithm presented, we can identify some parallelization opportunities, which would allow us to employ mechanisms such as threads and differentiate the services provided. We parallelized the banner selection algorithm by distributing the per-concept banner selection (lines 3 to 7 in Figure 4) so that all threads select candidate banners in parallel, one concept per thread. The tradeoff is between the number of threads allocated and the number of concepts in a request. In our implementation, as mentioned, we provide two levels of service. The high priority level is assigned to several (in fact a pre-defined number) threads while low priority level is always assigned to one thread, regardless the number of concepts.

As each thread takes a concept to process, if the number of concepts is lower than the number of pre-defined threads that will serve a high priority request, some threads will be idle and the gain will be smaller. This way, as the number of concepts grows, the high level requests will have smaller response times than low level ones, because in case of low level requests only one thread will be responsible for processing and in case of high level requests the execution will be shared among the available threads.

C. Metrics of QoS

In order to validate the strategy sugested in Section IV.B, it is necessary to define some metrics of QoS that will be gathered during the experiments. We employed two metrics:

Response Time: It is the elapsed time between the client starts requesting the banner and it is received. By assessing this metric, we can verify whether high priority requests are served better than lower priority requests.

Predictability: It is variance among the response times associated with requests from a given service level. By assessing this metric, we determine how predictable the response time is.

V. Experimental Analisys

In this section we describe the experiments we performed using our ad server and discuss the results gathered.

A. Experimental Environment

Our ad server runs on a quad-processor Pentium Pro 200 MHz machine with 128 Mb RAM and executes Solaris 8. The server was implemented in our laboratory and store all data necessary for the server operation in the same machine. There are two other machines representing clients, generating workload for the ad server. The clients are K6 II 450 MHz with 128 Mb RAM and all machines are located in the same LAN. One of the parameters of each request handled by the ad server is the type of the request, that could assume the values 0 or 1, representing low and high priority requests, respectively. This information allows not only to differentiate the service provided by the server, but also to distinguish between the response times associated to each level in the client programs.

B. Experiments Description

The workload submitted to the ad server is generated by a client program that reads a file containing HTTP requests and submits them to a server. The intensity of the workload generated is controlled by the number of simultaneous requests that are submitted by the client. The client program receives as parameters the address of the server to make the requests (IP number), the number of simultaneous clients and the total number of requests that each client will submit. For each request, it computes the response time and stores the request information and its response time in an output file.

By using this workload generator, we could vary the number of simultaneous clients that sent request to the ad server. We performed experiments that employed 10, 20, 60 and 100 simultaneous clients, each of them sending 300 requests. We also evaluated the ad server regarding the complexity of the request submitted, that is, the number of concepts (keywords) in the request. We varied the number of concepts by generating three different workloads, where all requests comprise 3, 6 or 10 concepts, respectively. Finally, we verified the impact of varying the number of threads that are employed to satisfy high priority requests, from 2 to 8, which we call the priority level.

C. Results and Analysis

After running all described experiments, we verified that the proposed strategy was succesful. As detailed next, we managed to differentiate service between requests by simply allocating a more threads to high priority requests. For instance, by using 10 simultaneous clients and 6 threads to process high priority requests, we improved by 50%, on av-

erage, the response time, as shown in Figure 6 and Figure 7.

The other QoS metric described in Section C is predictability. We can see in Figure 5 and in Table III that the response time of high priority requests did not vary as much as the low priority requests. In fact, the response time was almost constant for all requests, while low priority requests presented a large variance. We can conclude that, beyond the better response time, our stategy for providing differentiated QoS was also sucessful in terms of predictability.

As described in Section V.B, we verified the impact of varying the priority level, the number of concepts and the workload intensity. The results are shown in the sections that follow.

Thr	2 Concepts		6 Concepts		10 Concepts	
	Low	High	Low	High	Low	High
2	82.8	83.9	84.9	80.1	89.7	81.9
3	75.8	76.5	89.2	76.3	89.9	76.5
4	82.7	83.7	84.8	80.5	90.0	81.8
5	85.8	70.8	84.6	81.1	89.0	69.9
6	74.6	73.5	80.8	72.9	88.6	71.4
7	85.8	70.9	84.8	72.0	90.0	74.3
8	83.2	73.2	83.6	72.9	89.3	78.3

TABLE III

RELATIVE STANDARD DEVIATION FOR 10 SIMULTANEOUS CLIENTS

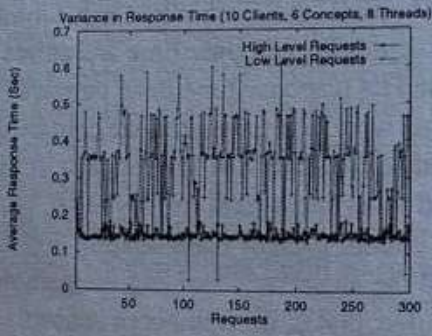

Fig. 5. Response time for streams of requests

C.1 Priority Level

The priority level is the number of threads employed to satisfy high priority requests. Results for different number of clients can be seen in Figures 6, 7, and 8. The first two graphs show the QoS gain, that is, the ratio between the response times of high and low priority requests. As expected, the gain increases with the priority level. There is no gain when we employ just 2 threads, on the other hand, the ad server answers high priority requests up to 60% faster. The improvement can be clearly observed in Figure 8, where we can see that the response time for high priority requests using 8 threads is half the response time for low priority threads.

C.2 Simultaneous Clients

We verified that an increase in the number of simultaneous clients minimizes the effectiveness of our approach, since the system gets saturated. However, we were always able to provide a better service to high priority requests, although the gains are much smaller when we service more clients, as can be observed in Table IV and by comparing the graphs in Figures 6 and 7.

C.3 Number of Concepts

The number of concepts in each request bounds the QoS gain, since it limits the potential parallelism. For instance, if we employ 8 threads to satisfy high priority requests, but the number of concepts is always less than 2, we will waste system resources with idle threads.

We can observe such problem in Figures 6 and in Figure 7, where the gains for requests comprising 3 concepts decrease as the number of threads increases, while the same could not be observed for requests with 6 and 10 concepts.

Cli.	2 Concepts		6 Concepts		10 Concepts	
	Low	High	Low	High	Low	High
10	0.124	0.071	0.184	0.073	0.251	0.126
20	0.136	0.113	0.260	0.163	0.366	0.269
60	0.350	0.317	0.675	0.571	1.066	0.954
100	0.806	0.738	1.123	1.014	1.759	1.642

TABLE IV

RESPONSE TIME USING PRIORITY LEVEL 6

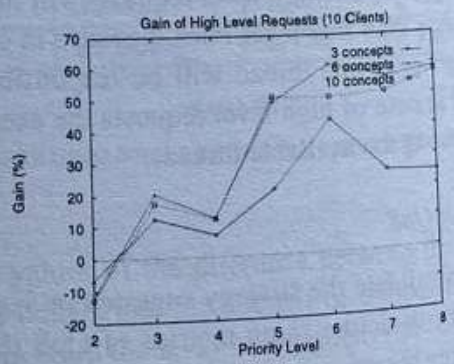

Fig. 6. QoS Gain for 10 clients

VI. CONCLUDING REMARKS

In this paper we proposed the use of variable level of parallelism for differentiating the quality of service provided by e-commerce servers. We show the effectiveness of the proposed approach by employing it in a two-level priority ad

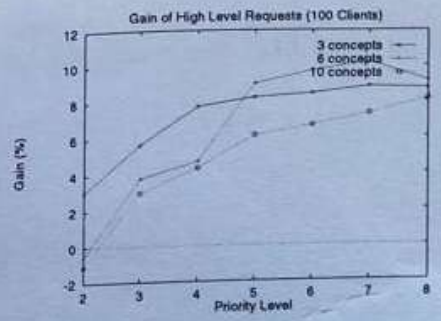

Fig. 7. QoS Gain for 100 clients

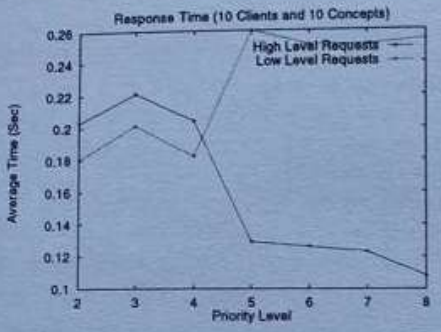

Fig. 8. Response time for 100 clients

server. Although the strategy was not effective in some cases (insufficient paralllelism), we provided response up to 60% faster to high-priority requests.

One limitation of our approach is the lack of control on the gain provided. This limitation is explained because we did not modify the scheduler of operating system. Thus, we guarantee a better service, but cannot establish service level agreements, which is a target for further investigation. To support such initiative, we plan to change the PTH [Pro] package so that server threads are scheduled according to pre-defined priorities, similarly to the Eclipse approach [BGOS98]. Another problem to be pointed is that our approach isn't suitable for shared servers, as the resource management can't be predicted (it is good for dedicated servers). In [BDM99], the authors introduce the concept of *Resource Containers*, that can be useful in the context of shared servers to minimize this problem. Finally, we plan to better characterize the workload to banner servers, by mixing requests comprising a variable number of requests and demanding our server to adapt to the instantaneous workload represented by the queries being satisfied.

REFERENCES

[BDM98] Gaurav Banga, Peter Druschel, and Jeff Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance*, Madison, WI, June 1998.

[BDM99] Gaurav Banga, Peter Druschel, and Jeff Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings od the Third Symposium on Operating System Design and Implementation*, New Orleans, LA, February 1999.

[BGOS98] J. Bruno, E. Gabber, B. Özden, and A. Silberchatz. The eclipse operating system: Providing quality of service vi a reservation domains. In *Proc. of the Usenix Annual Technical Conference*, New Orleans, LO, June 1998.

[BM99] A. Basu and S. Muylle. Customization in online trade processes. In *Proc. of the International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems*, pages 120–127, April 1999.

[CFHB99] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers, 1999.

[DMA00] B. Diniz, W. Meira Jr., and V. Almeida. Análise de desempenho da terceirização de serviços de comércio eletrônico. In *Anais do XVIII Simpósio Brasileiro de Redes de Computadores*, Belo Horizonte, MG, Maio 2000. SBC.

[KaGM00] E. Kraemer, G. Paix ao, D. Guedes, and W. Meira Jr. Minimizing the impact of orphan requests in e-commerce services. In *Proc. of the Performance and Arqchitecture of Web Servers*, June 2000. Held in conjunction the ACM SIGMETRICS 2000.

[MAFM99a] D. A. Menascé, V. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *Proc. 1999 ACM Conference on Electronic Commerce*, November 1999.

[MAFM99b] D. A. Menascé, V. Almeida, R. Fonseca, and M. A. Mendes. Resource management policies for e-commerce servers. In *Proc. Second Workshop on Internet Server Performance*, Atlanta, GA, May 1st 1999. in conjunction with ACM SIGMETRICS 99/FCRC.

[PDZ99] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. of the 1999 Annual Usenix Technical Conference*, Monterrey, CA, June 1999.

[PMA+00] G. Paixão, W. Meira Jr., V. Almeida, D. Menascé, and A. Pereira. Design and implementation of a tool for measuring the performance of complex e-commerce sites. In *Proceedings of the 11th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance Tools 2000)*, March 2000.

[Pro] Gnu Project. Pth: Portable threads library. http://www.gnu.org/software/pth/.

[Var95] H. Varian. Information economy. *Scientific American*, pages 201–202, September 1995.