The LC* Assignment Policy for Cluster-Based Servers

Victoria Ungureanu
Department of MSIS, Rutgers University
180 University Ave., Newark, NJ 07102
ungurean@research.rutgers.edu

Benjamin Melamed
Department of MSIS, Rutgers University
94 Rockafeller Rd., Piscataway, NJ 08854
melamed@rbs.rutgers.edu

Michael Katehakis
Department of MSIS, Rutgers University
180 University Ave., Newark, NJ 07102
mnk@andromeda.rutgers.edu

Abstract

A cluster-based server consists of a front-end dispatcher and multiple back-end servers. The dispatcher receives incoming jobs, and then decides how to assign them to back-end servers, which in turn serve the jobs according to some discipline. Cluster-based servers have been broadly deployed as they combine good performance with low cost.

Several assignment policies have been proposed for cluster-based servers, most of which aim to balance the load among back-end servers. There are two main strategies for load balancing: The first strategy aims at balancing the amount of work *at back-end servers, while the second strategy aims at balancing the* number of jobs *assigned to back-end servers. Example of policies using these strategies are JSQ (Join Shortest Queue) and LC (Least Connected), respectively.*

In this paper we propose a policy, called LC, which combines the two aforementioned strategies. The paper shows experimentally that when preemption is admitted (i.e. jobs are executed concurrently by back-end servers), LC substantially outperforms* both *JSQ and LC. This improved performance is achieved by using only information readily available to the dispatcher, and therefore LC* is a practical policy in regards to implementation.*

Keywords: cluster-based servers, back-end server architecture, job preemption, simulation.

1 Introduction

Web servers are becoming increasingly critical as the Internet assumes an ever more central role in the telecommunications infrastructure. Applications that handle heavy loads, commonly use a *cluster-based* architecture for Web servers because it combines good performance with low cost. A cluster-based server consists of a front-end dispatcher and several back-end servers (see Figure 1). The dispatcher receives incoming jobs and then decides how to assign them to back-end servers, which in turn process the jobs according to some discipline. The dispatcher is also responsible for passing incoming data pertaining to a job from a client to a back-end server. Accordingly, for each job in progress at a back-end server there is an open connection between the dispatcher and that server [13, 17].

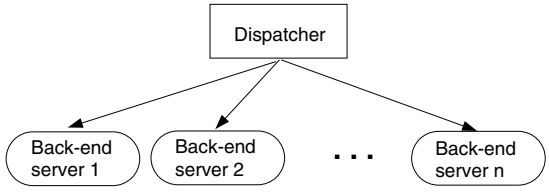


Figure 1. A cluster-based Web server.

Several dispatcher assignment policies have been proposed for this type of architecture (see, e.g. [4, 16]). One of the most well-known policies is JSQ (Join the Shortest Queue). Under JSQ, the dispatcher assigns an incoming job to the back-end server with the smallest amount of residual work, i.e., the sum of service demands of all jobs in the server queue plus the residual work of the jobs currently being served. It has been shown in [18] that JSQ is the *optimal* policy (i.e. it minimizes the expected response time) if: (a) jobs arrive at the dispatcher according to a Poisson process, (b) job sizes follow an exponential distribution, and (c) jobs are served FCFS (first-come first-served) at each server

Proceedings of the Third IEEE International Symposium on Network Computing and Applications (NCA'04)
0-7695-2242-4/04 $ 20.00 IEEE

queue.

However, the optimality of JSQ comes with several caveats. First, there is a great deal of evidence suggesting that the sizes of files traveling on the Internet do not follow an exponential-type distribution. Rather, these sizes appear to follow power-law (heavy-tailed) distributions [3, 7, 8] of the form $\mathbb{P}[X > x] \sim \frac{c}{x^\alpha}$, where X is the random file size, $c > 0$, and $1 \leq \alpha \leq 2$. For power-law job-size distributions, *a relatively small fraction of jobs accounts for a relatively large fraction of the overall load.*

Second, JSQ requires the dispatcher to know the status of all back-end servers at all times. This type of information is difficult, if at all possible, to acquire, and consequently, JSQ has not been used in practice [13, 17]. Rather, commercial products predominantly use the LC (Least-Connected) policy [5], under which the dispatcher assigns a job to the back-end server currently processing the smallest number of jobs (i.e., the one with the least number of open connections to the dispatcher). Note that both JSQ and LC aim to balance the load among back-end servers. However they employ *different strategies* to achieve this goal: JSQ aims to balance the *amount of work* at back-end servers, and LC aims to balance the *number of jobs* at them.

There are many performance studies, which compare experimentally these and other assignment policies, but commonly, these studies preclude *job preemption*, i.e. assume that jobs are executed sequentially, in FCFS order. In contrast, this paper studies the performance of JSQ and LC, with and without job preemption at back-end servers. Our choice is motivated by the fact that in practice, most, if not all, back-end servers use preemption, i.e process jobs concurrently (see Section 2 for further details). The study shows that preemption affects dramatically the performance of these policies. When preemption is precluded, then JSQ outperforms LC as expected. However, when preemption is allowed, then LC outperforms JSQ by a factor of two! This is a surprising outcome in view of JSQ's optimality. Moreover, LC with preemption still outperforms JSQ without preemption by a substantial margin. The last observation suggests that deploying LC with preemptive back-end servers is not only a practical choice, but should also yield better performance.

The results of the aforementioned study suggest that if jobs can be preempted, balancing the number of jobs at back-end servers is a better strategy than balancing the amount of work there. On the face of it, this contention appears to be counter-intuitive, because JSQ uses more information than LC. However, our results lend support to the contention that this apparently more detailed information is not as relevant as intuition would lead us to believe. We will elaborate on this point in Section 3.

In this context, an interesting question is whether *combining the two strategies* would improve even more the performance of a cluster employing preemption. To answer this question, we propose a new policy, called LC*, which aims to balance the number of jobs at back-end servers in a manner that avoids creating large disparities in the amount of work at them. In a nutshell, LC* operates as follows: The dispatcher uses a threshold parameter to classify incoming jobs into *short* and *long*; short jobs are assigned to the least connected back-end server, while long jobs are assigned to a back-end server, not currently processing a long job. In particular, if all back-end server-queues contain a long job, then the assignment of an incoming long job is deferred until a back-end server completes its large job.

The proposed LC* policy does not achieve perfect balancing of either the number of jobs or the amount of work. However, we argue heuristically that LC* tends to give rise to only relatively minor disparities (deviations from perfect balance) in these metrics across back-end servers. First, there are no large disparities in the amount of work, because a back-end server queue may contain at most one large job at any given time. Second, there are only minor disparities in the number of jobs: at any given time, the maximal possible difference in the number of jobs assigned to any two back-end server is 2. To see that, note that under LC, the maximal possible difference is 1. However, because, LC* does not require a large job to be assigned to the least connected server, the maximal possible difference increases to 2, since only one large job may be processed by a back-end server at any given time.

To gauge the performance of the LC* policy, we exercised it on empirical data traces measured at Internet sites serving the 1998 World Cup. We mention that Arlitt and Jin [3] show that job sizes from these traces do indeed follow a power-law distribution with $\alpha = 1.37$. In particular, for the trace considered files with sizes greater than 30KB, make up less than 3% of the files requested, but account for over 50% of the overall workload. We show that when files in excess of 30 KB are classified as long, LC* outperforms substantially both LC and JSQ. Thus, the study demonstrates that the careful assignment of a small number of jobs can have a profound impact on overall response time performance. It is worth pointing out that this increase in performance is achieved by using only information readily available to the dispatcher. Consequently, LC* is a *practical policy with regard to implementation*.

The rest of the paper is organized as follows. Section 2 presents the common back-end server architectures and examines how jobs are processed under various architectures. Section 3 discusses the effects of preemption on the performance of JSQ and LC by presenting a performance study driven by World Cup data traces. Section 4 presents in detail the LC* policy and illustrates its performance. Finally, Section 5 concludes the paper.

2 Back-end Server Architectures

The most common architectures employed for back-end servers are:

- **MP (Multi-Process).** In this architecture a back-end server employs multiple processes; each process serves a job to completion before accepting a new one. A process runs for a predefined time interval (quantum) or until it blocks, after which the operating system selects another process to run.

- **MT (Multi-Threaded).** In this architecture, back-end servers spawn multiple threads within a single address space; each thread serves a job to completion before accepting a new one. There are two main approaches to thread scheduling: preemptive and non-preemptive [6]. In *preemptive* scheduling, a thread may be suspended even when it is runnable, in which case, another thread is chosen for execution. In *non-preemptive* (coroutine) scheduling, a thread is allowed to run to completion unless it blocks.

From this brief description it follows that MP back-end servers can serve several jobs concurrently. The same holds true for MT back-end servers employing preemptive thread scheduling. The only practical case where it can be assumed to a certain degree that jobs are executed in FCFS order is when MT back-end servers use non-preemptive scheduling for threads.

We mention that Apache, the Web server with the broadest installed base today [1, 11], comes in two flavors: the MP architecture over UNIX, and the MT architecture over Microsoft Windows. It should be noted that other types of back-end server architectures exist, including single-process event-driven (SPED) [19] and asymmetric multi-process event-driven (AMPED) [14]. Both of these can serve multiple jobs concurrently.

3 The Impact of Preemption on JSQ and LC

This section presents a simulation study driven by real-life data traces, which shows that when back-end servers admit preemption (i.e. jobs are executed concurrently), the strategy of balancing the *number of jobs* at back-end servers outperforms the strategy of balancing the *amount of work*.

To compare the performance of JSQ and LC, we simulated a cluster of four back-end servers, driven by a World Cup trace, described below. The experiments were subject to the assumptions that communication times between the dispatcher and back-end servers and the overhead incurred by the dispatcher to select (job, server) pairs are negligible.

3.1 Simulation Data

Our study used trace data from Internet sites serving the 1998 World Cup. The data used are available on the Internet Traffic Archive (see [3] and http://ita.ee.lbl.gov/html/traces.html). This repository provides detailed information about the 1.3 billion requests received by World-Cup sites over 92 days – from April 26, 1998 to July 26, 1998. We mention again that Arlitt and Jin [3] have shown that job sizes from these traces follow a heavy-tail distribution.

From this repository, we selected a trace covering 1 hour from June 26 data, containing approximately 6 million requests. The relevant statistics of this trace are described next. Figure 2(a) depicts the number of requests received by the World-Cup cluster in successive one-second intervals, while Figure 2 (b) plots the number of bytes requested from the same cluster in successive one-second intervals. To further underline the power-law distribution of job-sizes we point out the following aspects of this trace:

- Approximately 75% of the files requested have sizes of less than 2KB, which account for less than 12% of the transferred data.

- Files with sizes greater than 30KB, which make up less than 3% of the files requested, account for over 50% of the overall workload. Even more striking, files in excess of 100KB, which make up less than 0.04% of all files requested, account for 7% of the transferred data.

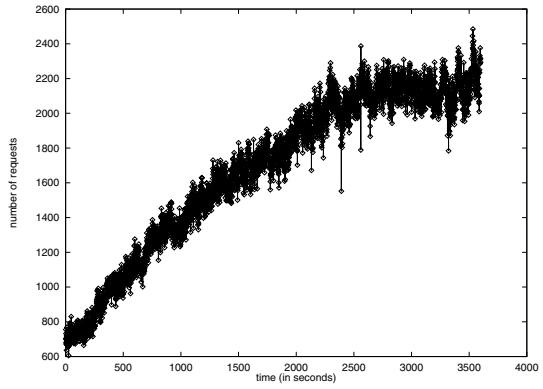
The selection of the particular data trace was motivated by the fact that it exhibits arrival-rate fluctuations corresponding to light, medium and heavy loadings in this temporal order, as evidenced by Figure 2. More specifically, the trace allows us to study policy performance under various loading conditions, as follows:

- **Light loading.** In the time interval [0, 1200], the arrival rate is relatively low (below 1600 requests/sec), and the resultant utilization coefficient is also low ($\approx 40\%$).

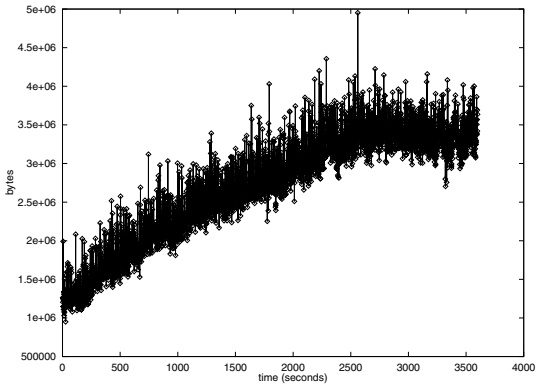
- **Medium loading.** In the time interval (1200, 2400], the arrival rate is between 1600 and 2000 requests/sec, and the resultant utilization coefficient is intermediate.

- **Heavy loading.** In the time interval (2400, 3600], the arrival rate exceeds 2000 requests/sec, and the corresponding utilization coefficient is high ($\approx$75%).

From each trace record, we extracted only the request arrival time and the size of the requested file. We mention that the recorded time stamps are in integer seconds, with arrivals on the order of several hundreds of requests



(a) Number of request arrivals per second;



(b) Total bytes requested per second

Figure 2. Empirical request statistics from a World Cup trace.

per second. Consequently, we have distributed request arrivals uniformly over each second. Since no service time information was recorded, the simulation estimates the service time as the sum of the (constant) time to establish and close a connection, and the (variable, size-dependent) time required to retrieve and transfer a file. The justification for this estimation method may be found in [13, 15, 17].

3.2 Simulation Experiments

The performance metric used to compare these assignment policies is *slowdown*, defined as the ratio between a job's response time (the time interval from the moment a job arrives at the dispatcher and up until it ends processing at the corresponding back-end server) and its service time.

3.2.1 Job Processing Without Preemption

Figure 3 displays average slowdowns in successive one second intervals for the two assignment policies considered, under the assumption that jobs are not preempted (i.e., jobs are executed sequentially in the order of their arrival at back-end servers). The figure shows that JSQ outperforms LC over all time intervals, and therefore, under all simulated loading conditions.

3.2.2 Job Processing With Preemption

These experiments simulate MP back-end servers that schedule jobs for execution in round-robin manner. Figure 4 displays average slowdowns in successive 60-second intervals. Interestingly, under the MP-architecture, LC outperforms JSQ over *all* time intervals. Moreover, the relative advantage of LC over JSQ increases as the load increases. For light loadings their performance is very close, but for heavy loadings, LC outperforms JSQ by as much as a factor of 3.

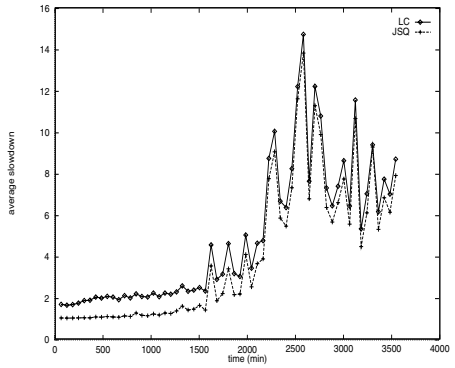


Figure 3. Successive average slowdowns of LC and JSQ when jobs are processed without preemption.

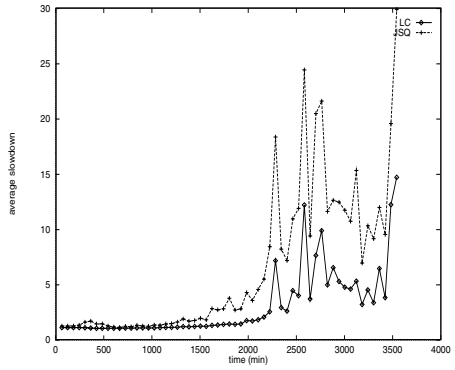


Figure 4. Successive average slowdowns of LC and JSQ when jobs are processed with preemption.

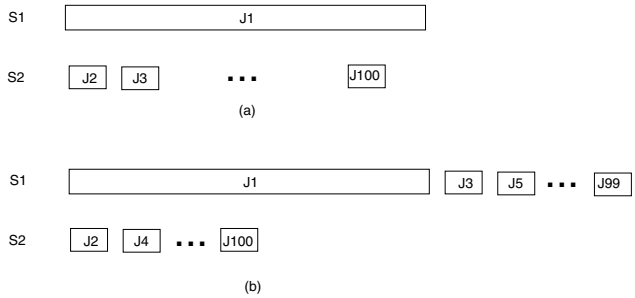


Figure 5. Server queues under (a) JSQ, and (b) LC.

These observations can be explained by the fact that performance is primarily affected here by the number of jobs contending for resources. To gain insight into this explanation, consider the scenario, where a sequence of jobs, J_1,. . .,J_{100}, arrives back-to-back at a cluster consisting of two back-end servers, S_1 and S_2 (see Figure 5). Assume that J_1 requires a service time of 100 s, while all others require a service time of 1 s. Under JSQ, S_1 is assigned the large job (J_1), while S_2 is assigned all other (small) jobs. Accordingly, the amount of work is almost evenly distributed among the two back-end servers, as illustrated in Figure 5(a). If processes are scheduled for execution in round-robin manner, for a quantum of 0.1 s, the average slowdown incurred is $\approx 95s$. To see why, note that the response times for the small jobs are 90 s, 90.1 s, 90.2 s, ..., 100 s, and therefore their slowdown are 90, 90.1, 90.2, ..., 100. In contrast, the large job incurs a slowdown of only 1 since it is served in 100 s.

Under LC, S_1 is assigned the large job and 49 small ones, while S_2 is assigned 50 small jobs. Accordingly, the number of jobs is divided evenly among the two back-end servers, as illustrated in Figure 5(b). Consequently, the response times of small jobs assigned to S_2 are just 45 s, 45.1 s, 45.2 s,....,50 s. Small jobs assigned to S_1 have similar response times, while the response time of the large job is 150 s, and therefore its slowdown is 1.5. Consequently, the average slowdown incurred is ≈47, which is better by a factor of 2 as compared to JSQ. Intuitively, the improvement is due to the fact that LC arranges to double the throughput of small jobs as compared to JSQ, and noting that the response times of small jobs dominate the overall response time.

This example leads us to contend that the larger the variability in the number of jobs across server queues, the poorer JSQ performs. This contention is borne out by the experiments: JSQ performs far worst than LC for heavy loadings (where the length of back-end server queues and their variability tend to be large) than under light loading (where the length of server queues and their variability tend to be small).

	average slowdown	
policy	no job preemption	job preemption
JSQ	4.69	7.7
LC	5.58	3.53

Table 1. Comparative statistics for JSQ and LC

We conclude this section by comparing the average performance of JSQ and LC with and without preemption, over the simulation time horizon (see Table 1). We make three observations from this data. First, JSQ yields better performance without preemption than with preemption. Second, LC's behavior is opposite, namely, it yields better performance with preemption than without preemption. Finally, LC with preemption outperforms JSQ without preemption.

4 The LC* Policy

The experiments and the example presented in the previous section support the contention that aiming to balance the number of jobs assigned to back-end servers is more important than balancing the amount of work there. We shall now proceed to demonstrate experimentally that response-time performance may be further improved if *balancing the number of jobs is combined with balancing the amount of work*.

To support this contention, consider the following motivating example. Suppose that two large jobs and two small jobs arrive back-to-back at a cluster consisting of two back-end servers. Then, under LC, there are two possible assignments of these jobs. The first assigns a large job and a small job to each back-end server, while the second assigns the two large jobs to one back-end server and the two small ones to the other. Note that the first assignment balances the amount of work, while the second does not. To quantify how this disparity affects performance, assume that the service times of large and small jobs are 100 s and 1 s, respectively. Then the response time of each large job under the first assignment is ≈100 s, while under the second assignment, it is ≈200 s! This outcome is due to the large disparity of workload across back-end servers under the second assignment.

This example suggests that balancing both the number of jobs and the amount of work may lead to improved performance. It further suggests that assigning multiple large jobs to the same back-end server exacts a large response-time penalty, and therefore such assignments should be avoided. These observations motivate our LC* policy, defined as fol-

lows.

1. *The dispatcher classifies incoming jobs into long or short relative to a cutoff parameter c.*
2. *The dispatcher assigns short jobs to the least connected back-end server.*
3. *The dispatcher assigns a large job to a back-end server whose queue does not already contain a large job. If there is no such back-end server, the dispatcher defers the assignment until a back-end server completes its large job.*

We draw the reader's attention to the following points. First, the classification into long and short jobs is based on the *size of requested documents*. Although *job service time* is a more relevant classification metric pro forma, we nevertheless chose *size*, because the dispatcher has this type of information readily available. Furthermore, it has been shown that the time to service a request is well-approximated by the size of the requested file [10].

Secondly, the dispatcher can estimate job size only for *static* requests (dynamic files are created on the fly by the server in response to a request). Consequently, LC* implicitly treats dynamic requests as short. Even though this classification may be inaccurate for certain dynamic requests, we argue that the errors incurred do not affect substantially LC*'s performance. This is because evidence suggests that while the number of dynamic requests is growing, the majority of the requests at most web servers are static [2, 10, 12]. For example, it was shown in [3] that for World Cup traces only 0.02% of requests were for dynamic files.

Finally, LC* is practical to implement in that the extra information required is readily available to the dispatcher, and the processing of this information is quite fast.

We next proceed to demonstrate experimentally, via a case study, that LC* outperforms both JSQ and LC. The study simulates a cluster of four MP back-end servers that process the jobs recorded by the trace of Section 3.1. The simulation sets the value of the cutoff parameter to 30K (i.e., under LC*, a request for a file whose size is at least 30K is classified as a large job). Recall that for the trace considered files with with sizes greater than 30KB, which make up less than 3% of the files requested, account for over 50% of the overall workload.

Figure 6 displays average slowdowns of LC and LC* in successive 60-second intervals. Table 2 displays slowdown averages under various loading regimes (light, medium and heavy), as well as the overall time horizon.

We next proceed to compare the average slowdowns of LC to those of LC*:

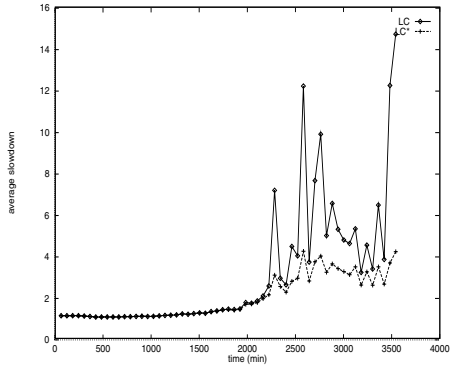


Figure 6. Successive average slowdowns for LC **and** LC*.

- **Light loading.** The average slowdowns of both policies in the lightly-loaded interval [0, 1200] is very close to 1, meaning that jobs are served almost without delay. These statistics are due to the fact that when the arrival rate is low, a job is predominantly assigned to an idle back-end server.
- **Medium loading.** As the arrival rate and subsequent utilization increase in the interval [1200,2400], LC* slightly outperforms LC. This is due to the fact that LC* has only few occasions to make better assignments than LC.
- **Heavy loading.** As the loadings becomes heavy in the interval [2400, 3600], LC* outperforms LC by almost a factor of 2. Moreover, there are periods during which LC* outperforms LC by as much as a factor of 4! (See, for example, the terminal period.) This behavior is due to the fact that under heavy loadings LC* has many occasions to make better assignments than LC.

We conclude this section with the following observations. First, LC* outperforms JSQ with and without preemption by a factor of 3 and 2, respectively (see Tables 1 and 2). Second, LC* dramatically outperforms LC under heavy loading regimes, while under light and medium regimes their performance is similar. This observation suggests that the dispatcher should employ an adaptive policy: it should use LC in light traffic, and switch to LC* in heavy traffic. Thus, under such an adaptive policy, the overhead attendant to LC* is incurred only when it gives rise to substantially improved performance.

5 Conclusion

Several conclusions can be drawn from the studies presented in this paper. First, the back-end server architecture

policy	average slowdown			
	light loading	medium loading	heavy loading	overall
LC	1.06	1.90	6.17	3.53
LC*	1.06	1.59	3.22	2.16

Table 2. Comparative statistics for LC and LC*

affects profoundly the performance of a policy. Specifically, the paper shows experimentally that the performance of JSQ is far better when preemption is precluded than when it is supported.

The second conclusion is that if jobs can be preempted, balancing the number of jobs at back-end servers is a better strategy than balancing the amount of work there. In particular, the paper shows experimentally that LC, which uses the former strategy, performs far better than JSQ, which uses the latter strategy.

Finally, the study supports the contention that combining the two balancing strategies yields performance superior to that of each constituent strategy. Specifically, the paper proposes a policy, LC*, which improves over both LC and JSQ. A notable feature of the proposed policy is that it has only modest informational and computational requirements, which renders it a practical candidate for real-life implementation.

Acknowledgments

This work was supported in part by the Center for Supply Chain Management at Rutgers University and by DIMACS under the contract STC-91-19999.

References

[1] "The Apache HtextrmP Server Project". http://httpd.apache.org/

[2] Arlitt, M., Friedrich, R. and Jin, T. Workload Characterization of a Web proxy in a cable modem environment. *ACM Performance Evaluation Review*, 27(2), 25-36, 1999.

[3] Arlitt, M. and T. Jin. "Workload Characterization of the 1998 World Cup Web Site," *IEEE Network*, 14(3), 30-37, May/June 2000. Extended version: Tech Report HPL-1999-35R1, Hewlett-Packard Laboratories, September 1999.

[4] Bruckner, P. *Scheduling Algorithms*, Third Edition, Springer-Verlag, 2001.

[5] Cardellini, V., E. Casalicchio, M. Colajanni and P. S. Yu. "The state of the art in locally distributed Web-server systems", *ACM Computing Surveys*, 34(2):263-311, 2002.

[6] Coulouris, G., J. Dollimore and T. Kindberg. "Distributed Systems - Concepts and Design", (3rd edition), Addison-Wesley, 2001.

[7] Crovella, M.E., M.S. Taqqu and A. Bestavros. "Heavy-tailed Probability Distributions in the World Wide Web," In *A Practical Guide To Heavy Tails*, Chapman Hall, New York, 3–26, 1998.

[8] Faloutsos, M., P. Faloutsos and C. Faloutsos. "On Power-Law Relationships of the Internet Topology," In Proceedings of *ACM SIGCOMM '99*, 251-262, Aug. 1999.

[9] Harchol-Balter, M. "Task Assignment with Unknown Duration," *Journal of the ACM*, Vol. 49, No. 2, 260-288, March 2002.

[10] Harchol-Balter M., B. Schroeder, N. Bansal, M. Agrawal. "Size-based Scheduling to Improve Web Performance." *ACM Transactions on Computer Systems*, 21(2), May 2003.

[11] Hu, Y., A. Nanda and Q. Yang. "Measurement, Analysis and Performance Improvement of the Apache Web Server", in *The International Journal of Computers and Their Applications*, 8(4),217-231, 2001.

[12] Krishnamurthy, B. and Rexford, J. Web Protocols and Practice : HtextrmP 1.1, Networking Protocols, Caching, and Traffic Measurement. Addison-Wesley, 2001.

[13] Pai, V.S., M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel and E. Nahum. "Locality-aware Request Distribution in Cluster-based Network Servers", in the *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.

[14] Pai, V.S., P. Druschel and W. Zwaenepoel. "Flash: An Efficient and Portable Web Server", In the *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[15] Riska, A., W. Sun, E. Smirni and G. Ciardo. "AdaptLoad: Effective Balancing in Clustered Web Servers Under Transient Load Conditions," in *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002.

[16] Pinedo, M. *Scheduling: Theory, Algorithms, and Systems*, Prentice Hall, 2002.

[17] Teo Y.M. and R. Ayani, "Comparison of Load Balancing Strategies on Cluster-based Web Servers", In *Simulation, The Journal of the Society for Modeling and Simulation International*, 77(5-6), 185-195, November-December 2001.

[18] Winston, W. "Optimality of the Shortest Line Discipline," *Journal of Applied Probability*, 14, 181–189, 1977.

[19] Zeus Technology. "Zeus Web Server". http://www.zeus.com/serve/.