Performance Study of Dispatching Algorithms in Multi-tier Web Architectures

Mauro Andreolini [1] Michele Colajanni [2] Ruggero Morselli [3]

Abstract

The number and heterogeneity of requests to Web sites are increasing also because the Web technology is becoming the preferred interface for information systems. Many systems hosting current Web sites are complex architectures composed by multiple server layers with strong scalability and reliability issues. In this paper we compare the performance of several combinations of centralized and distributed dispatching algorithms working at the first and second layer, and using different levels of state information. We confirm some known results about load sharing in distributed systems and give new insights to the problem of dispatching requests in multi-tier cluster-based Web systems.

1 Introduction

Web technology is becoming the preferred standard interface for accessing many services exploited through the Internet. While the first generation of Web sites was largely based on static and read-only information, an increasing percentage of Web sites provide information and services that are personalized for the client or created dynamically by the execution of some application process. The consequence is that dynamic pages and services are becoming essential in modern sites where Web-based technologies have emerged as a valid alternative to traditional client-server computing. Because of the complexity of the Web infrastructure, performance problems may arise in many points during an interaction with Web-based systems. Upgrading the power of a single server will not solve the Web scalability problem in a foreseeable future. The alternative architecture we consider in this paper is a locally distributed Web system composed by multiple nodes that are typically organized in layers.

An example of multi-tier cluster-based Web system (briefly, *Web cluster*), is shown in Figure 1. The only visible address is the Virtual IP (VIP) corresponding to the front-end device which is located in front of the set of servers. This device, hereafter called *Web switch*, interfaces the rest of the Web cluster nodes with the Internet, thus making the distributed nature of the site architecture completely transparent. The first set of *Web server* nodes run the HTTP daemons. They listen on some network port for the client requests assigned by the Web switch, prepare the content requested by the clients, send the response back to the clients or to the Web switch, and finally return to the listen status. The Web server nodes are capable of handling requests for static content, whereas they forward requests for dynamic content to *back-end* servers hosting databases and other (legacy) applications. It may consist of a layer of separate nodes or thin gateway processes running on the Web server nodes that accept requests from the Web server and interact with the database server or other legacy applications at the back-end layer. A middle layer may be interposed between the Web and the back-end layer. The load reaching this Web system must be evenly distributed among the server nodes, so as to improve performance. Hence, we have to include some component that routes client requests among the servers with the goal of load sharing maximization. The level of multiple indirections in a multi-tier Web cluster architecture where each layer consists of multiple server nodes opens several interesting performance problems because request routing and dispatching can be implemented at different levels, for example, at the Web switch and at the Web server layer. The Web switch receives all inbound packets that clients send to the VIP address, and routes them to a Web server node. In such a way, it acts as the *centralized* dispatcher of a distributed system with fine-grained control on client requests assignments. This topic is widely investigated in literature. For a complete survey see [5].

Request dispatching and load sharing at the internal layers are implemented in most commercial products, but they are not widely studied topics in the research community. Indeed, the selection of a back-end server can be done by some *centralized* entity (the Web switch itself or some master node, e.g. [25]) or in a *distributed* way by any Web server. The combination of feasible alternatives is wide. This paper gives one of the first contributions to this area. We compare the performance of several combinations of centralized and distributed dispatching algorithms working at the first and second layer, and using different levels of state information.

The main focus is on the second dispatching-level because the non-uniformity of the load and non-cacheability of most documents (in spite of many efforts [12, 20]) introduce additional degrees of complexity to the request dispatching issue. Indeed, burst arrivals and hot spots can be faced by the Web switch and servers through caching techniques so that a Web

[1] Department of Computer Engineering, University of Roma "Tor Vergata", Roma, Italy 00133, andreolini@ing.uniroma2.it

[2] Department of Information Engineering, University of Modena, Modena, Italy 41100, colajanni@unimo.it

[3] University of Modena, Italy, and Department of Computer Science, University of Maryland, ruggero@cs.umd.edu

server node can typically deliver several hundreds of static files per second. On the other hand, to share the load for dynamic requests is quite difficult because they often require orders of magnitude higher service times. We also analyze the efficiency and the limitations of the different solutions and the tradeoff among the alternatives with the aim of identifying the characteristics of centralized and distributed approaches and their impact on performance. A partially unexpected result is that at the second layer the most simple dispatching policies work better than the more sophisticated ones, even when the system is highly loaded. Instead, detailed information about the server, especially past load, does not appear useful, and can often lead to bad dispatching choices. Our experimental results confirm in a quite different context the well known results by Eager et al. [13] and others [9] obtained through analytical and simulative models.

Zhu et al. [25] have studied the problem of request dispatching in a multi-tier architecture, where a second-layer switch is integrated in each Web server node, called *master node*. When a Web server node receives a request for dynamic content, it queries the master to choose the back-end server to which it has to forward the request. The back-end node selection is based on a prediction model that estimates the expected cost for processing the dynamic request on each slave node. Similar multi-tier architectures have been also analyzed in [4].

The generation of dynamic content opens other performance issues that are beyond the scope of this paper. Indeed, the alternative solutions depend also on the application software, the chosen middleware and database technology. For example, commercial Web service software, such as BEA WebLogic and IBM WebSphere, have evolved from simple Web servers into complex Web application servers that use CGI, Java Server Pages, Microsoft Active Server Pages, XML, and other technologies. A different solution for load balancing based on CORBA middleware technology is proposed in [21] where several dispatching strategies have been also proposed and evaluated. Other interesting research fields in multi-tier Web architectures are not oriented to load balancing, but to mechanisms that can cache query results and dynamic content [12, 20].

The rest of this paper is organized in the following sections. In Section 2, we describe the dispatching algorithms that can be used by the Web switch and Web servers in a centralized or distributed way. In Section 3 and 4, we examine the test-bed prototype and the benchmarking parameters that we used in the experiments, respectively. In Section 5, we discuss the experimental results which are obtained for two main workload scenarios. In Section 6, we outline our conclusions.

2 Dispatching algorithms

In a multi-tier Web cluster there is a dispatching policy carried out by the Web switch that selects a Web server, and another

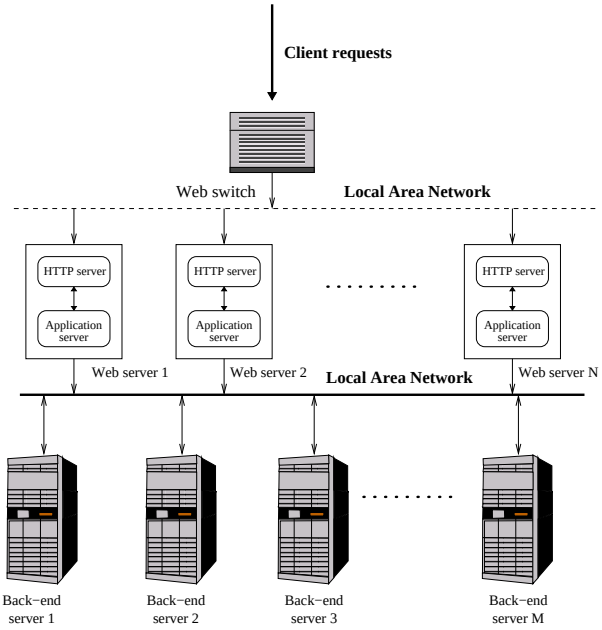

Figure 1: An example of multi-tier architecture for a cluster-based Web system.

dispatching choice when a back-end server has to be chosen for a dynamic request. The common rule is that these dispatchers cannot use highly sophisticated algorithms because they have to take immediate decisions for hundreds or thousands of requests per second. There are several alternatives for dispatching algorithms described in literature, however for the scope of this paper the most important choices are among *centralized* vs. *distributed*, and *state-blind* vs. *state-aware* algorithms.

The front-end architecture with a single Web switch that receives all inbound packets drives the choice to centralized dispatching policies. Hence, the real alternative for the Web switch dispatching-layer is among *state-blind vs.* state-aware algorithms. On the other hand, for the second dispatching-layer, we may consider both *distributed policies carried out by the Web servers and* centralized solutions where the decision is taken by the Web switch or by another server selected as a master.

State-blind (or static) algorithms are the fastest solution because they do not rely on the current state of the system at the time of decision making. However, the common belief is that these algorithms may make poor assignment decisions. *State-aware* (or dynamic) algorithms have the potential to outperform static algorithms by using some state information to help dispatching decisions. The main problems with dynamic algorithms are that they require mechanisms to collect and analyze data, thereby incurring in potentially expensive overheads and stale information about state conditions.

State-aware algorithms can take into account a variety of system state information that depends also on the protocol stack

layer at which the dispatcher operates. They can be further classified according to the level of system state information being used by the dispatcher. A Web server operates at the application layer and can apply *content-aware dispatching* algorithms, while a Web switch can operate at the same layer or at the TCP layer where only *content-blind dispatching* algorithms are allowed. We consider the following four classes of algorithms.

- In *state-blind* policies, the dispatcher assigns requests using a static algorithm and no dynamic information.
- In *client state-aware* policies, the dispatcher routes requests on the basis of some client information. For example, a dispatcher working at application layer can examine the entire HTTP request and take decisions on the basis of more detailed information about the client. (Indeed *client info-aware* would be a more correct definition for this class of algorithms, but we prefer to use client state-aware for symmetry reasons.)
- In *server state-aware* policies, the dispatcher assigns requests on the basis of some server state information, such as current and past load condition, latency time, and availability.
- In *client and server state-aware* policies, the dispatcher routes requests by combining client and server state information.

We consider the dispatching algorithms which are representative of each of these four classes. Table 1 summarizes the policies analyzed in this paper and whether they are applied at the Web switch or at the Web server.

2.1 State-blind algorithms

State-blind policies do not consider any system state information. Typical examples are **Random** and **Round-Robin** (**RR**) algorithms. Random distributes the incoming requests uniformly through the server nodes with equal probability of reaching any server. RR uses a circular list and a pointer to the last selected server to make dispatching decisions.

Both Random and RR policies can be easily extended to treat servers with different processing capacities by making the assignment probabilistic on the basis of server capacity. Different processing capacities can be also treated by using the so-called **static Weighted Round-Robin**, which comes as a variation of the Round-Robin policy. Each server is assigned an integer weight w_i that indicates its capacity. Specifically, $w_i = C_i/min(C)$, where $min(C)$ is the minimum server capacity. The dispatching sequence will be generated according to the server weights.

2.2 Server state-aware algorithms

When we consider dispatching algorithms that use some server state information we have to decide the *server load index*, how and when to compute it, how and when to transmit this information to the dispatcher. These are well known problems in a networked system [11, 15].

Once a server load index is selected, the dispatcher can apply different algorithms. A common scheme is to have the new request assigned to the server with the lowest load index. The **Least Loaded** algorithm (**LL**) actually denotes a set of policies that depend on the index chosen to characterize the server load. The three main factors that affect the latency time of a Web request are loads on CPU, disk, and network resources of the Web server nodes. Typical server state information includes the CPU utilization evaluated over a short interval, the instantaneous CPU queue length periodically observed, the disk or I/O storage utilization, the instantaneous number of active connections, the number of active processes, and the object latency time, that is, the completion time of an object request at the Web cluster side. These indexes need a process monitor on the servers and a communication mechanism from the servers to the dispatcher. For example, in the **Least Connections** policy (**LL-CONN**), which is usually adopted in commercial products (e.g., Cisco's LocalDirector [7], F5 Networks' BIG-IP [14]), the dispatcher assigns the new request to the server with the fewest active connections. In this paper we consider also other load metrics, such as CPU and disk utilization. In the **LL-CPU** policy and **LL-DISK** policy the dispatcher assigns requests to the server having lowest CPU and disk utilization, respectively.

The **dynamic Weighted Round-Robin** algorithm (WRR) is a variation of the version that considers static information (server capacity) as a weight. This policy associates each server with a dynamically evaluated weight that is proportional to the server load state, typically estimated as a number of connections [16]. Periodically, the dispatcher gathers load index information from the servers and computes the weights, that are dynamically incremented for each new connection assignment.

2.3 Client state-aware algorithms

The complexity and overheads of a dispatcher that can examine the HTTP request at the application layer motivates the use of more sophisticated content-aware distribution policies. The proposed algorithms may use information about the requested URL for different purposes, such as: to improve reference locality in the server caches so to reduce disk accesses (*cache affinity*); to use specialized server nodes to provide different Web services (*specialized servers*), such as streaming content, dynamic content; to partition the Web content among the servers, for increasing secondary storage scalability (*service partitioning*); to increase load sharing among the server nodes (*load sharing*).

In cache affinity policies, the file space is typically partitioned among the server nodes. A hash function applied to the URL (or to a substring of the URL) can be used to perform a static partitioning of the files. The dispatching policy running on the Web switch (namely, **URL hashing** algorithm) uses the same

Table 1: Dispatching policies to distribute client requests among Web and back-end servers.

Policy	Location	Knowledge
RR	switch/server	state-blind
LL-CONN	server	server state-aware
LL-CPU	server	server state-aware
LL-DISK	server	server state-aware
WRR	server	server state-aware
PART	switch	client info-aware
CAP	server	client info-aware
CAP-CENTR	switch	client info-aware
LARD	switch	client and server state-aware
CAP-ALARM	server	client and server state-aware

function. However, the solution combining Web object partitioning and URL hashing can be applied to Web sites providing static content only. Moreover, it ignores load sharing completely, as it is difficult to partition the file space in such a way that requests are balanced out. Indeed, if a small set of files accounts for a large fraction of requests (a well-known characteristic of Web workloads, e.g., [2, 10]), the server nodes serving those critical files will be more loaded than others. For cacheable documents, these critical load peaks can be lowered by architectures that integrate proxy server solutions, but the problems remain for most dynamic services.

For Web sites providing heterogeneous Web services, the requested URL can be used to statically partition the servers according to the file space and service type they handle. The goal is to exploit the locality of references in the server nodes, to achieve the best cache hit rate for static documents, and possibly to employ specialized servers for certain types of requests, such as dynamic content, multimedia files, streaming video [24]. We refer to this policy as to **Service Partitioning** (**PART**). Most commercial content-aware switches deploy this type of approach (for example, F5 Networks' BIG-IP [14], Resonate's Central Dispatch [23]) although they use different names.

Another goal of the content-aware dispatching algorithms is to improve load sharing among the server nodes. These strategies do not require static partitioning of the file space and the Web services. Let us consider one policy belonging to this class, called **Client Aware Policy** (**CAP**) [6], that is oriented to Web sites providing services with different computational impact on system resources. The basic observation of CAP is that when the Web site provides heterogeneous services, each client request could stress a different Web system resource, e.g., CPU, disk, network. Although the Web switch cannot estimate the service time of a static or dynamic request accurately, it can distinguish the class of the request from the URL and estimate its main impact on one Web system resource. A feasible classification for CAP is to consider disk bound, CPU bound, and network bound services, but other choices are possible depending on the Web content. To improve load sharing in Web clusters that provide multiple services, the Web switch manages a circular list of server assignments for each class of Web services. The CAP goal is to share multiple load classes among all servers so that no component of a node is overloaded. CAP in its initial form is a client state-aware policy, however it can be easily combined with some server state information.

It is worth noting that the client state-aware policies do not require a hard tuning of parameters, which is typical of the server state-aware policies, because the service classes are decided in advance and the dispatching choice is determined statically once the requested URL has been classified.

2.4 Client and server state-aware algorithms

Client information is often combined with some server state information, for example to provide the so called *client affinity* [16, 17]. In policies based on client affinity, client and server information have a different weight: when available, client information usually overrides server information for assignment decisions. Instead of assigning each new connection to a server on the basis of its server state regardless of any past assignment, performance or functional reasons motivate the assignment of the same request to the same server.

The **Locality-Aware Request Distribution** (**LARD**) policy is a content-aware request distribution that considers both locality and load balancing [3, 22]. The basic principle of LARD is to direct all requests for the same Web object to the same server node as long as its utilization is below a given threshold. By so doing, the requested object is more likely to be found into the disk cache of the server node. Some check on the server utilization is useful to avoid overloading servers and, indirectly, to improve load sharing.

A modified version of the CAP policy, namely **CAP-ALARM**, uses an asynchronous communication mechanism that exchanges alarm messages only when necessary, in addition to the request content for dispatching. The server load status is computed periodically; if it exceeds a threshold value, an alarm message is sent to the dispatcher. When the load falls clearly below the threshold, a wakeup message is sent to the dispatcher. The requests are distributed using the

CAP algorithm, unless the chosen server has sent an alarm message, indicating that its utilization has exceeded the maximum capacity. In this case, CAP-ALARM chooses the next server which is not overloaded, if it exists, or the least loaded server. The overloaded server will be used again as soon as the dispatcher receives a wakeup message.

3 Test-bed architecture

The Web cluster consists of a Web switch node, connected to the back-end nodes and the Web servers through one or multiple high speed LANs, as in Figure 1. The distributed architecture of the cluster is hidden to the HTTP client through a unique Virtual IP (VIP) address.

In this paper we use a layer-7 Web switch implemented at the application layer, thus allowing content-aware request distribution. Different mechanisms were proposed to implement a layer-7 Web switch at various operating system levels. The most efficient solutions are TCP hand-off [22] and TCP splicing [8], both implemented at the kernel level. Application layer solutions are undoubtedly less efficient than kernel level mechanisms, but their implementation is cheaper and sufficient for the purposes of this paper which intends to compare dispatching algorithms instead of Web switch solutions. There is no doubt that a commercial real system should work at the kernel level, but addressing the switch performance issues is out of the scope of this paper.

In particular, we used the TCP gateway mechanism (also known as reverse proxy or surrogate): a proxy running on the Web switch at the application layer mediates the communication between the client and the server. When a request arrives, the proxy on the Web switch accepts the client connection and forwards the client request to the target server. When the response arrives from the server, the proxy forwards it to the client through the client-switch connection.

We implemented the TCP gateway mechanism through the Apache Web server software, that was sufficient to implement and test any layer-7 dispatching algorithm without modifications at the kernel level.

Our system implementation is based on off-the-shelf hardware and software components. The clients and servers of the system are connected through a switched 100Mbps Ethernet that does not represent the bottleneck for our experiments. The cluster is made up of a Web switch, two Web servers and four back-end servers. Each machine is a Dual PentiumIII-833Mhz PC with 512MB of memory. All nodes of the cluster use a 3Com 3C905C 100bTX network interface. They are all equipped with a Linux operating system (kernel release 2.4.17). Apache 1.3.12 is used as the Web server software, while PHP 3.0.17 has been chosen as the dynamic page support. On the Web switch and on the Web servers, the Apache Web server is configured as a reverse proxy through the modules *mod_proxy* and *mod_rewrite*. The dispatching algorithms are implemented as C modules that are activated at startup of the Apache server. The dispatching module communicates with the *rewriting engine*, which is provided by *mod_rewrite*, over its *stdin* and *stdout* file handles. For each map-function lookup, the dispatching module receives the key to lookup as a string on *stdin*. After that, it has to return the looked-up value as a string on *stdout*.

Some dispatching policies require information about the server loads. Therefore, we implemented load monitors for the following load metrics: number of active connections, CPU utilization and number of I/O requests issued to the disk. Moreover, several load update strategies have been implemented on the dispatcher modules.

The combination of a set of load indexes into a single index that reflects the server load is an interesting research issue that has not yet been investigated in the context of Web clusters. In our test-bed system, we are considering the last five load coefficients coming from each server and we implemented the following three load update strategies:

***Last value*:** this is the most obvious load update strategy. The last load coefficient coming from the server is used as the server state information.

***Arithmetical mean*:** the state load of a server is computed as the arithmetical mean of the last three load coefficients coming from that server.

***Weighted mean*:** the server state information is computed as the weighted mean of the five most recent load samples coming from the server. Each value is combined with a weight obtained by a decay distribution. We use $L_i = \sum_{j=1}^{5} (1/G) l_i(j) e^{-j/2}$, where $l_i(j)$ for $j \in \{1, \ldots, 5\}$ are the load samples of the server i evaluated in the past five intervals of 5 seconds each, and $G = \sum_{j=1}^{5} e^{-j/2}$ is the normalizing factor.

In addition to the choice of the server load index, all server state aware policies face the problem of updating load information. The intervals between load index updates need to be chosen carefully to make sure that the system remains stable. If the interval is too long, performance may be poor because the system is responding to old information about the server loads. On the other hand, too short intervals may result in system over-reaction and instability. A strategy for interpreting stale load information that can apply to layer-4 Web switches has been proposed in [11]. Space limitations do not allow us to address such important issue in this paper.

4 Workload model

The Web cluster is subject to a mixed workload consisting of both static and dynamic documents. Static pages and objects are served by the Web servers, while dynamic content is

Table 2: Workload model for static requests and client behavior.

Category	*Distribution*	*PMF*	*Range*	*Parameters*
Requests per session	Inverse Gaussian	$\sqrt{\frac{\lambda}{2\pi x^3}} e^{\frac{-\lambda(x-\mu)^2}{2\mu^2 x}}$	$x > 0$	$\mu = 3.86$, $\lambda = 9.46$
User think time	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1.4$, $k = 1$
Objects per request	Pareto	$\alpha k^\alpha x^{-\alpha-1}$	$x \geq k$	$\alpha = 1.33$, $k = 2$
HTML object size	Lognormal Pareto	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$ $\alpha k^\alpha x^{-\alpha-1}$	$x > 0$ $x \geq k$	$\mu = 7.630$, $\sigma = 1.001$ $\alpha = 1$, $k = 10240$
Embedded object size	Lognormal	$\frac{1}{x\sqrt{2\pi\sigma^2}} e^{\frac{-(\ln x-\mu)^2}{2\sigma^2}}$	$x > 0$	$\mu = 8.215$, $\sigma = 1.46$

processed by the back-ends. Dynamic workload is obtained by means of PHP scripts which are passed to the back-end servers. The same dynamic service may yield very different results depending on when it is requested. This affects response times tremendously. To take this aspect into account, for each request we pre-generate a random number ranging from 1Kb to 100Kb, representing the size of the buffer on which we perform dynamic computations. We chose not to generate bigger buffers since we do not want to perform stress-test analysis, but a fair policy comparison. We have considered four classes of dynamic services. *Type 1* emulates the retrieval of data which is cached at the back-end and does not require intensive CPU or disk usage. *Type 2* emulates a query to a database. *Type 3* emulates an activity that stresses the CPU, such as ciphering. *Type 4* emulates a heavy Web transaction that requires a database query and ciphering of the results.

In the experiments we compare the performance of dispatching strategies under two workload scenarios. Both of them consist of 80% of dynamic requests and 20% of static requests. This choice was motivated to put more stress on the back-end nodes, but it is also confirmed by some workload characterizations, e.g. [1]. Among the dynamic requests, 50% are of *Type 1*, 25% are of *Type 2*, 12.5% are of *Type 3*, 12.5% are of *Type 4*. In the *light scenario* and in the *heavy scenario* client inter-arrival times are exponentially distributed with mean equal to 10 seconds and 1 second, respectively. We have verified that these load mixes do not overload the Web switch and the Web servers, but tend to stress the back-end servers of the test-bed prototype. This is inline with the purposes of this paper that is more oriented to investigate dispatching at the second layer. We have also verified that a different mix of dynamic requests does not change the results significantly, provided that the bottleneck remains at the back-end level and not at the Web switch or at the Web server layer.

To emulate realistic Web workloads, we have also considered concepts like user sessions, user think-time, embedded objects per Web page, realistic file sizes and popularity. Table 2 summarizes the probability mass function (PMF), the range, and the parameter values of the workload model for static requests and client behavior.

The performance metric of interest for this paper is the response time of a client request, consisting of the base HTML page and all embedded objects. Web workloads are characterized by heavy-tailed distributions, thus response times may assume highly variable values with non-null probability. We have verified, as other authors did in advance, that the mean is not at all representative of the system behavior. Hence, we have preferred to use the cumulative distribution and the 90-percentile of the response time of a client request. These metrics are able to represent much better the different behaviors of the dispatching policies examined in the next section.

Because of this choice, we have also modified the *httperf* tool [19] (version 0.8) because no available benchmark was able to return the 90-percentile and the cumulative distribution of the response time. Client requests are obtained from a pre-generated trace whose characteristics follow the model described above. This approach guarantees also the replicability of the tests, that is fundamental for a fair comparison of the performance of the dispatching policies. During each experiment, at least 1000 user sessions are generated; this value increases as the mean inter-arrival time decreases, to keep test times sufficiently long. We have planned 5 runs for each experiment and consider the mean result; in this way, we limit the fluctuations derived from the high variability of the workload.

5 Experimental results

In this section, we present the results of several dispatching algorithms with a focus on the the second-layer. We indicate the solutions that use a centralized policy at the Web switch layer, and a distributed algorithm at the Web server layer through the <policy1>/<policy2> notation.

We first consider the class of server state-aware policies to the purpose of showing the difficulty of using server load information for dispatching. Next, we analyze server state-blind policies which do not take into account any kind of server load information, and we verify that these algorithms perform generally better than server state-aware policies. The final goal is to compare centralized and distributed solutions.

5.1 Instability of server state information

Various issues need to be addressed when we consider dispatching policies based on some server state information: first of all, the choice for one or more *server load index(es)*; then the way to compute the load state information and the frequency of the samples; finally, due to the cluster architecture, most indexes are not immediately available at the dispatcher, so we have to decide how to transmit them and how frequently.

The focus of this section is on distributed dispatching at the second layer that is, between the Web and the back-end servers. We consider the LARD centralized policy at the Web switch that performs well for static documents. We compare nine server state-aware policies that come as variants of the basic Least Loaded algorithm. In particular, we consider three load indexes evaluated on the back-end servers (that is, LL-CPU, LL-DISK, LL-CONN), and the three metrics "instantaneous load", "arithmetical mean" (AM) and "weighted mean" (WM), because we are also interested to analyze the impact on performance of using present and/or past samples. For comparison reasons, we finally report the results of another state-aware policy, CAP-ALARM, that uses an asynchronous alarm mechanism for skipping overloaded back-end servers.

There are various ways for reading the results reported in the Figures 2- 7. Two preliminary observations are in order: the figures on the left side refer to the light workload scenario and response time scales are between 0 and 8 seconds; the figures on the right side refer to the heavy workload scenario and response time scales are between 0 and 12 seconds.

The first important conclusion is a confirmation of the importance of the load index and update interval. Any of these choices has a strong impact on final performance to the extent that the same server LL policy can behave much better or much worse than other algorithms depending on an adequate selection of the load indexes and of the update interval. The dependency on these choices is impressive: the 90-percentiles of response times go from 2.3 to 7 seconds for the light scenario, and from 6 to 12 seconds for the heavy scenario.

If we focus on the sample analysis, we can see that in most instances, even if some instability exists, the shortest update interval (5 seconds) gives the best results. Moreover, the lowest response time is achieved when the last sample or the weighted samples (WM) are considered. On the other hand, the arithmetical mean (AM) that gives the same importance to all samples never performs as the best. These results indicate that it is preferable to make dispatching decisions based on the most recent load information only. The motivation can be attributed to the dynamics of the multi-tier Web cluster system. Due to the large variability of client requests assigned to a server, the load information becomes obsolete quickly and it is poorly correlated with future server load.

From the Figures 2- 7 we can also derive other interesting considerations. The CPU utilization (LL-CPU) is undoubtedly the best index for representing the load on the back-end servers. The curves in Figures 4 and 5 are the most stable, to the extent that this algorithm is almost no sensitive to the update intervals and to present and past samples. Stability of the results is of primary importance for a Web system subject to highly variable traffic. Moreover, LL-CPU gives the lowest response times with respect to LL-CONN and LL-DISK that is, always below 3 seconds and below 7 seconds for the light and heavy scenario, respectively.

The number of active TCP connections (LL-CONN) is not a good measure of the server load, because almost anything can be requested to a back-end server over a TCP connection, from a small document, possibly cached, to a stressing query. This result was actually expected, while the bad performance of the LL-DISK results more surprising. As many queries to the back-end servers are also disk bound, we expected that the number of I/O requests issued in the last update interval would be a good measure of the back-end load. On the other hand, Figures 6 and 7 show that the 90-percentiles of response times are much higher than those achieved by the LL-CPU, and comparable, although more stable, to those obtained by the LL-CONN. We give the following motivation to this result. Counting the number of I/O requests suffers from a drawback similar to that discussed about the number of active TCP connections. An access to disk does not reveal much about the computational weight of the involved query. Moreover, as the update interval increases this measure tends to become inaccurate, especially if I/O requests are concentrated in bursts during short periods of time.

We finally compare the results of the LL policies against those of the CAP-ALARM algorithm that does not use server load information to take dispatching decisions, but it just alerts the dispatcher to skip an overloaded or unreachable server. It should be noted that for the CAP-ALARM policy, the update interval concerns only the recomputation of the server load status, because an alarm (wakeup) message is sent only if the server exceeds (falls below) a given load threshold. Whereas CAP-ALARM performs similarly to LL-CPU when the load is light, for the heavy scenario we have that simply tracking the heavily loaded servers reduces the response times by almost 50% (Figure 5). This result is important because this performance improvement can be achieved at a minimum increase in the communication overheads.

Actually, for a fair comparison, it must be pointed out that the CAP-ALARM policy tends to perform better when the load recomputation interval is small, since alarm and wakeup messages are delivered more promptly to the dispatcher. This intuitive result is confirmed by the results of the CAP-ALARM algorithm with respect to the server load recomputation interval in a light and heavy load scenario (Figure 8). It is interesting to observe that we have two clear choices for solving the tradeoff between performance results and overhead: either we prefer a very small interval (e.g., about 1 second) with its consequent high costs or we limit overheads and choose larger observation intervals (e.g., 6 or more seconds).

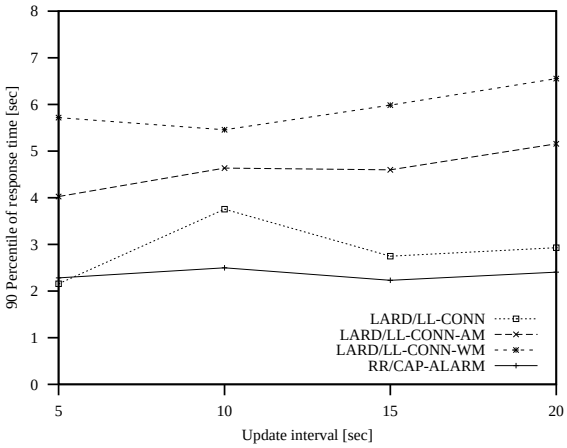

Figure 2: LARD/LL-CONN algorithms (*Light load scenario*)

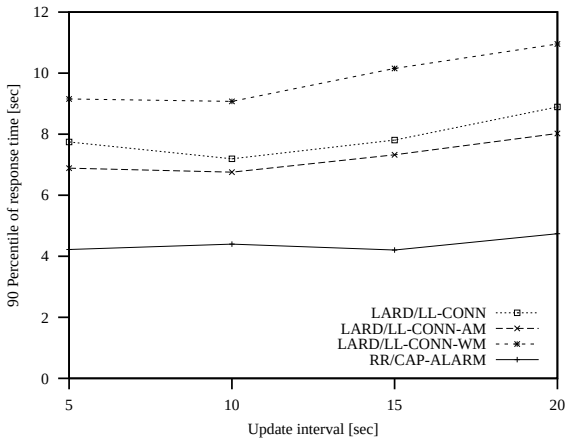

Figure 3: LARD/LL-CONN algorithms (*Heavy load scenario*)

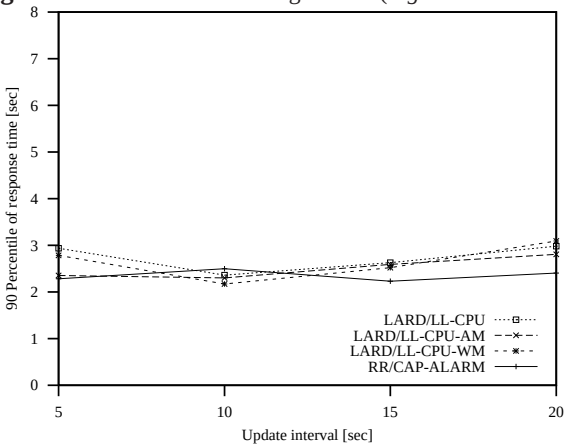

Figure 4: LARD/LL-CPU algorithms (*Light load scenario*)

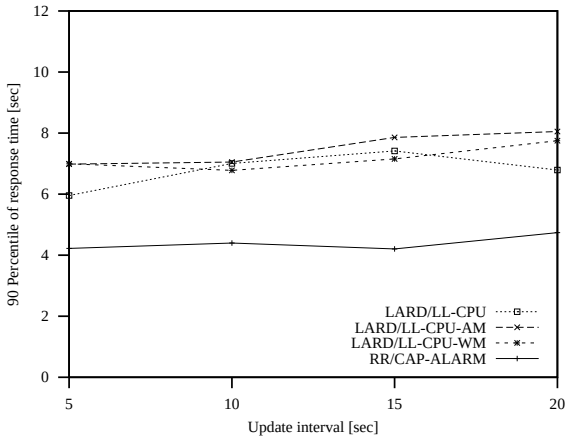

Figure 5: LARD/LL-CPU algorithms (*Heavy load scenario*)

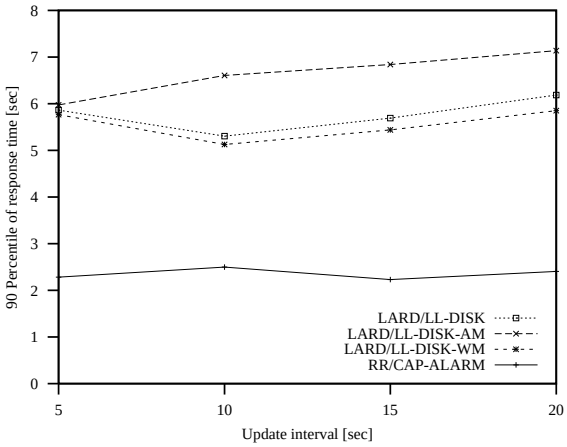

Figure 6: LARD/LL-DISK algorithms (*Light load scenario*)

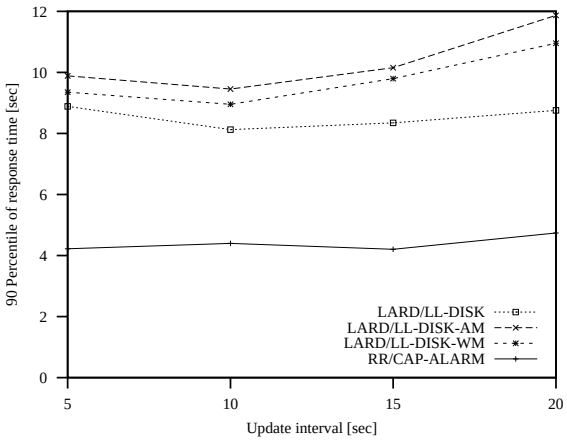

Figure 7: LARD/LL-DISK algorithms (*Heavy load scenario*)

Similarly to [13], we can conclude that in a distributed system the threshold policy using a small amount of state information provides substantial performance improvements. On the other hand, the LL strategy, which tends to assign every dynamic request to the same back-end for the entire update interval can easily unbalance the server loads and can possibly overload the selected server even for light scenarios.

An important final observation is in order. Although the quantitative values of the performance results are by no means independent of the considered workload model, the relative conclusions about the load indexes, metrics and sample are

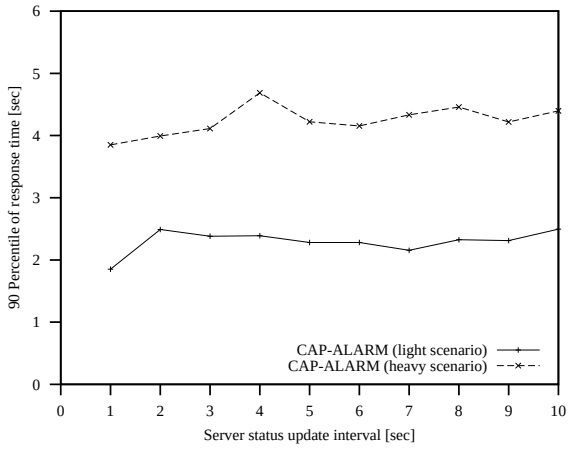

Figure 8: Sensitivity of the CAP-ALARM algorithm to the frequency of status information updating

rather stable.

5.2 Performance of server state-blind policies

In this section we consider the policies which do not take into account any server state information for dispatching. There are two classes of server state-blind policies: pure state-blind (e.g., RR) and client state-aware algorithms (e.g., PART, CAP). The figures report also the best server state-aware policy (LLopt) whose performance must be considered as an optimal target which is absolutely not guaranteed for all workload conditions.

Figures 9 and 10 show the cumulative distribution functions of the response time in the case of a light and heavy scenario, respectively. When the load is low, all server state-blind policies perform similarly. When we pass to consider a heavier dynamic workload, the performance results of RR/RR and PARTopt/RR are clearly the best. The slight better results of PARTopt/RR over RR/RR were expected because the document partition was done on the basis of a known workload, so to maximize load sharing. This is an optimal condition for the PART policy because different workload distributions could lead the PART policy to perform much worse than PARTopt, especially if requests tend to be concentrated to subsets of Web document directories.

On the other hand, the good results of RR/RR, even better than RR/CAP, were quite surprising (and, we can add, disappointing, because they contradicted the results obtained when RR and CAP were applied to a centralized Web switch [6]). We give the following motivation. It is not convenient to use a multi-class dispatching list for each Web server when there are multiple independent dispatchers, because this augments the probability that two or more requests of the same class are sent to the same back-end server with high risks of overloading it. Instead, dispatching the requests in a plain RR way (that is, through a single class list for each independent dispatcher) limits the risks of conflicts that instead the CAP policy tends to augment.

5.3 Centralized vs. distributed algorithms

After the analysis of several distributed policies, we study the performance of a centralized algorithm for dispatching requests to the back-end servers. There are two main ways for implementing a centralized mechanism: to select a master node as in [25] or to let the Web switch decide for both first- and second-layer dispatching. In this paper we refer to the latter choice, because we verified that a query to an external master node for each request dispatching slows down the response time.

Hence, we consider the CAP policy implemented at the Web switch, namely CAP-CENTR, that parses the URL and classifies the type of the request (we recall that there are four classes of dynamic requests and one of static requests). The static requests are assigned to the Web servers in a RR way, while for each dynamic request the Web switch selects a back-end server on the basis of the CAP multi-class list.

Figure 11 and 12 show the 90-percentile of the response times for the CAP-CENTR algorithm and other distributed algorithms that show the best results. (It should be noted that two of them, that is LARD/LLopt and PARTopt/WRRopt, are optimal versions of the considered policies, for which the shown results are not guaranteed at all.) Again, the light scenario (Figure 11) does not show appreciable differences.

In a heavier load scenario, the convenience of a centralized (CAP-CENTR) and a state-blind solution (RR/RR) over distributed server state-aware dispatching becomes evident. Indeed, with a distributed mechanism, the dispatchers tend to influence each other negatively, unless some sort of back-end partitioning scheme is adopted. We recall that the PARTopt results shown in these figures are by far an optimal choice that is not guaranteed for other workload conditions.

6 Conclusions

Systems with multiple nodes are the leading architectures to build highly accessed Web sites that have to guarantee scalable services and to support ever increasing request load. The performance problems of Web-based architectures is worsening also because of the need of client authentication and system security, the increased complexity of middleware and application software, and the high availability requirements of corporate data centers and e-commerce Web sites. In this paper, we analyze dispatching algorithms that are suitable for multi-tier distributed Web systems. We have analyzed the efficiency and limitations of different centralized and distributed policies and evaluated the tradeoff among the considered alternatives. From the performance study carried out in this paper we can take several conclusions.

- To balance the load across the nodes organized in mul-

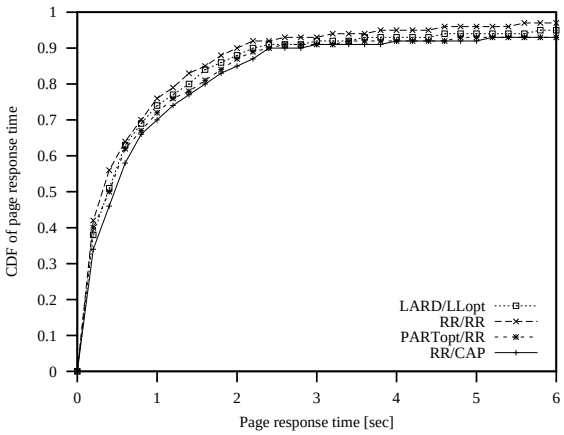

Figure 9: State-blind and state-aware policies (*Light load scenario*)

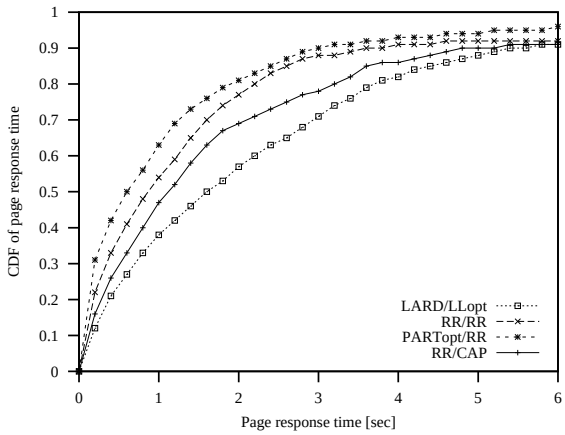

Figure 10: State-blind and state-aware policies (*Heavy load scenario*)

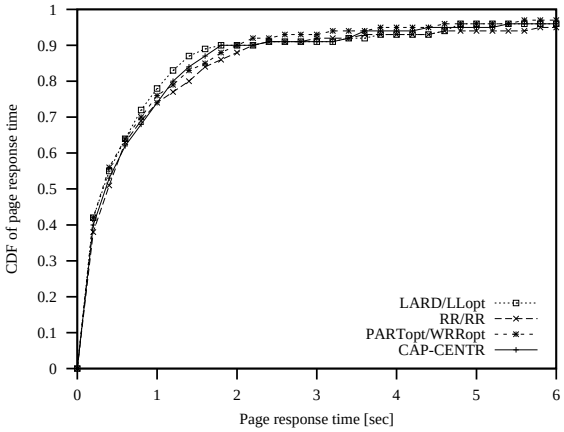

Figure 11: Centralized *vs* distributed policies (*Light load scenario*)

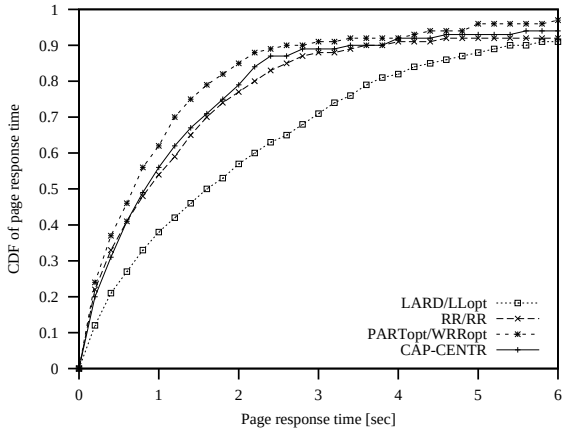

Figure 12: Centralized *vs* distributed policies (*Heavy load scenario*)

tiple layers, a Web cluster has two control knobs: the dispatching policy carried out by the Web switch for selecting a Web server for static requests and a dispatching mechanism for selecting a back-end server for dynamic requests. Just exploring one dispatching component alone is inadequate to address uneven distributions of the traffic reaching a Web site.

- Among the most used dispatching algorithms in commercial Web clusters, we found that Round-Robin gives good results and is quite stable, while the Least-Loaded policies are inadequate because they tend to drive servers to saturation as all requests are sent to the same server until new information is propagated. This "herd effect" is well known in distributed systems [11, 18], yet it is surprising that the Least-Loaded approach is commonly proposed in commercial products. On the other hand, server state-blind algorithms make not so poor assignment decisions as we and other authors expected.

- Another initially unexpected result for the server state-aware policies is that the best performance is achieved by algorithms that use only limited state information for checking whether a server is overloaded or not. On the other hand, detailed load information, especially the less recent load information, does not appear useful, and can often lead to bad dispatching choices. The most promising server state-aware algorithms use asynchronous alarms (when the load at a server crosses a threshold) combined with some client information. However, server load remains an unreliable information which is costly to get and difficult to manage adequately, because it is highly sensitive to the instantaneous state conditions.

- Pure client state-aware policies have a great advantage over policies that use also server information, as they do not require expensive and hard-to-tune mechanisms for monitoring and evaluating the load on each server, gathering the results, and combining them to make dispatching decisions. However, even client state-aware policies should take into account at least a binary server information in order to avoid routing the requests to temporarily unavailable or overloaded servers.

- Static partitioning algorithms tend to ignore load sharing, as it is almost impossible to partition the Web services in such a way that client requests are balanced out. On the other hand, if these algorithms are combined with dispatching policies that guarantee high cache hit rates, such as LARD, they give best results for Web sites providing static information and some simple database information.

The study of optimal resource management in multi-tier architectures is challenging because the multitude of involved technologies and complexity of process interactions occurring at the middle-tier let the vast majority of commercial products prefer quite naive dispatching solutions. Combining load balancing and caching of dynamic content in multi-tier systems is worthy of further investigation.

References

[1] M. Arlitt, D. Krishnamurthy, and J. Rolia. Characterizing the scalability of a large web-based shopping system. *ACM Trans. on Internet Technology*, 1(1):44–69, Aug. 2001.

[2] M. F. Arlitt and T. Jin. A workload characterization study of the 1998 World Cup Web site. *IEEE Network*, 14(3):30–37, May/June 2000.

[3] M. Aron, P. Druschel, and Z. Zwaenepoel. Efficient support for P-HTTP in cluster-based Web servers. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 185–198, Monterey, CA, June 1999.

[4] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July/Aug. 2001.

[5] V. Cardellini, E. Casalicchio, M. Colajanni, and P. Yu. The state of the art in locally distributed web-server system. *ACM Computing Surveys*, 34(2), June 2002.

[6] E. Casalicchio and M. Colajanni. A client-aware dispatching algorithm for Web clusters providing multiple services. In *Proceedings of the 10th International World Wide Web Conference*, pages 535–544, Hong Kong, May 2001.

[7] Cisco Systems Inc. `http://www.cisco.com/`.

[8] A. Cohen, S. Rangarajan, and H. Slye. On the performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct. 1999.

[9] M. Colajanni, P. S. Yu, and D. M. Dias. Analysis of task assignment policies in scalable distributed Web-server systems. *IEEE Trans. Parallel and Distributed Systems*, 9(6):585–600, June 1998.

[10] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Trans. Networking*, 5(6):835–846, Dec. 1997.

[11] M. Dahlin. Interpreting stale load information. *IEEE Trans. Parallel and Distributed Systems*, 11(10):1033–1047, Oct. 2000.

[12] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A middleware system which intelligently caches query results. In *Proceedings of ACM/IFIP Middleware 2000*, pages 24–44, Palisades, NY, Apr. 2000.

[13] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Engineering*, 12(5):662–675, May 1986.

[14] F5 Networks Inc. `http://www.f5labs.com/`.

[15] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In *Proceedings of Performance 1987*, pages 515–528, North-Holland, The Netherlands, 1987.

[16] G. S. Hunt, G. D. H. Goldszmidt, R. P. King, and R. Mukherjee. Network Dispatcher: A connection router for scalable Internet services. *Computer Networks*, 30(1-7):347–357, 1998.

[17] Linux Virtual Server project. `http://www.linuxvirtualserver.org/`.

[18] M. Mitzenmacher. How useful is old information. *IEEE Trans. Parallel and Distributed Systems*, 11(1):6–20, Jan. 2000.

[19] D. Mosberger and T. Jin. httperf - A tool for measuring web server performance. In *Proceedings of Workshop on Internet Server Performance*, Madison, Wisconsin, 1998.

[20] Oracle. Oracle9*i*as web Cache. `http://www.oracle.com/ip/deploy/ias/caching/index.html`.

[21] O. Othman, C. O'Ryan, and D. C. Schmidt. Strategies for CORBA middleware-based load balancing. *IEEE Distributed Systems Online*, 2(3), Mar. 2001.

[22] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, San Jose, CA, Oct. 1998.

[23] Resonate Inc. `http://www.resonate.com/`.

[24] C.-S. Yang and M.-Y. Luo. A content placement and management system for distributed Web-server systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, pages 691–698, Taipei, Taiwan, Apr. 2000.

[25] H. Zhu, B. Smith, and T. Yang. Scheduling optimization for resource-intensive Web requests on server clusters. In *Proceedings of the 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 13–22, June 1999.