Estimation-Based Load-Balancing with Admission Control for Cluster Web Servers

Saeed Sharifian, Seyed Ahmad Motamedi, and Mohammad Kazem Akbari

The growth of the World Wide Web and web-based applications is creating demand for high performance web servers to offer better throughput and shorter user-perceived latency. This demand leads to widely used cluster-based web servers in the Internet infrastructure. Load balancing algorithms play an important role in boosting the performance of cluster web servers. Previous load balancing algorithms suffer a significant performance drop under dynamic and database-driven workloads. We propose an estimation-based load balancing algorithm with admission control for cluster-based web servers. Because it is difficult to accurately determine the load of web servers, we propose an approximate policy. The algorithm classifies requests based on their service times and tracks the number of outstanding requests from each class in each web server node to dynamically estimate each web server load state. The available capacity of each web server is then computed and used for the load balancing and admission control decisions. The implementation results confirm that the proposed scheme improves both the mean response time and the throughput of clusters compared to rival load balancing algorithms and prevents clusters being overloaded even when request rates are beyond the cluster capacity.

Keywords: Cluster web server, load-balancing algorithm, layer-7 switch, admission control.

Manuscript received Mar. 13, 2008; revised Dec. 16, 2008; accepted Feb. 18, 2009.

Saeed Sharifian (phone: +98 21 66462691, email: sharifian_s@aut.ac.ir) and Seyed Ahmad Motamedi (email: motamedi@aut.ac.ir) are with the Department of Electrical Engineering, Amirkabir University of Technology, Tehran, Iran.

Mohammad Kazem Akbari (email: akbarif@aut.ac.ir) is with the Department of Computer Engineering & IT, Amirkabir University of Technology, Tehran, Iran.

I. Introduction

Nowadays, the Web has become business-oriented and is the preferred interface for information and services around the world. The web community is growing day by day, exponentially increasing the load that web sites must support. On the other hand, users have come to expect low site downtime and short response times. Therefore, web service providers should offer services with superior performance in order to retain existing users and attract new ones [1]. One of the most popular solutions for these challenges is the cluster web server [2]. More Internet service providers run their services on a cluster of servers and this trend is accelerating.

A typical cluster web server architecture is shown in Fig. 1. The main components of the cluster are a set of web servers, a set of database servers, and a web switch. The web switch acts as a centralized global scheduler that receives requests and dispatches them to the web servers.

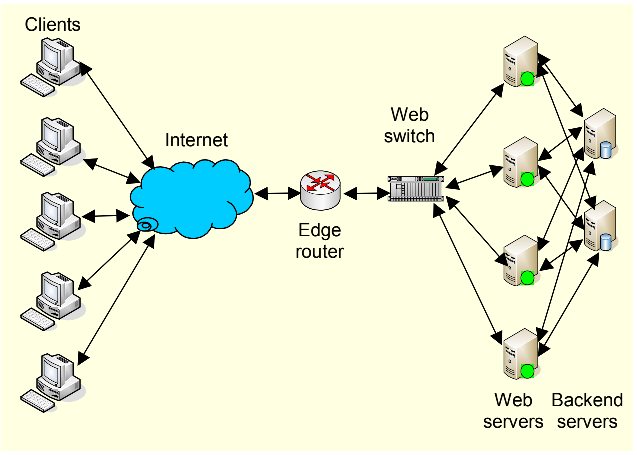

Fig. 1. Typical architecture of cluster web server.

Any client intending to request a page, first contacts the web switch. The web switch selects the server best suited to handle the request and assigns the request to it. If the request involves data stored in the database servers, the web server sends a query to the database and converts the results to HTML format. After the server has finished processing the request, it sends a response back to the client through the web switch.

To avoid some of the web servers becoming idle while others are overloaded, a load balancing algorithm is employed in the cluster. A load balancing algorithm which runs in the web switch plays an important role in boosting cluster performance. Load balancers make decisions regarding which server is best suited to assign a new request to it. The use of a fine load balancing algorithm increases cluster throughput, reduces response times, and improves reliability.

Current cluster web servers have to overcome two problems to keep clients satisfied. First, dynamic workloads which are becoming popular in current web sites impose significant performance drops in clusters due to weak load balancing algorithms. In addition to data-rich online web services, even seemingly static web pages are usually generated dynamically in order to include personalization and advertising features. However, dynamic contents make significantly higher resource demands than static web pages [3], [4] and create performance problems in the absence of a proper load balancing algorithm in cluster web servers. Second, current clusters are subjected to enormous variations in demand, often in an unpredictable fashion, and this results in flash crowds. Admission control helps the cluster serve the maximum number of requests in overload conditions and maintain response times at an acceptable level. Therefore, admission control is a critical issue in keeping a web server cluster operational in the presence of overload, even when the incoming requests rate is several times greater than the cluster capacity.

In this paper, we present a new load balancing algorithm with admission control for cluster web servers. The algorithm makes decisions based on estimated available capacity of each web server in a cluster. Our contributions in this paper are the following. First, we classify requests based on their service times. Given the big differences in the service demands of web workloads, classification provides an opportunity to better manage workloads. Second, we propose a load estimation mechanism based on the number of requests from each class in a system and their resource demands. Third an adaptive load balancing algorithm with an admission control mechanism is proposed based on the estimated load (available capacity) of each web server in a cluster. We run some experiments on a prototype cluster to evaluate the effectiveness of the algorithm and compare it to rival algorithms. The implementation results indicate significant gains with the proposed load balancing algorithm in terms of the mean response time and cluster throughput. Also, the proposed admission control mechanism prevents performance drop in overload conditions.

The rest of this paper is organized as follows. Related work is presented in section II. Section III describes the architecture of the cluster web server. Section IV presents a method to classify the web workload. Estimation of the available capacity of web servers is presented in section V. We propose our scheduling and admission control algorithms in sections VI and VII, respectively. Section VIII presents a performance evaluation system, and experimental results are given in section IX. Finally, conclusions and future works are outlined in section X.

II. Related Work

Various academic and commercial proposals confirm the increasing interest in web clusters regarding load balancing [5]-[10] architecture design; performance optimization [4]; overload and admission control [1], [11]-[15]; and load balancing on a geographical scale [16]. A detailed survey of general load balancing algorithms and their classification into layer-4 and layer-7 algorithms is provided in [2].

First-generation load balancing algorithms such as random (RAN) and round-robin (RR) are static algorithms and do not consider server load information in load balancing decisions. This shortcoming was improved in the second-generation load balancing algorithms, such as weighted round-robin (WRR), least connections (LC), and weighted least connections (WLC) [2], [6]. These algorithms collect instantaneous load status information of web servers (such as CPU load, disk usage, and the number of active network connections) as load descriptors and use them in server selection decisions. These load statuses obtained via direct measurement fluctuate at different time scales and become obsolete quickly [6], [17], [18]. Therefore, a decision which is made based on direct resource measurement of load status may be risky if not completely wrong. Moreover, the communication cost of load measurement within a cluster for these algorithms is relatively high for a large number of nodes in the cluster.

Third-generation load balancing algorithms use workload information such as type of URL and cookies, in addition to server load information. Proactive request distribution (PRORD) [10] and ADAPTLOAD [9] are two examples of third-generation load balancing algorithms that aim to improve the cache hit rate in web server nodes. These algorithms work fine in clusters that host traditional static web publishing services and benefit from a cache [4], [6], [7]. These algorithms mainly focus on improving the performance of clusters for static workloads and do not consider dynamic workloads which impose very different processing requirements on the cluster. These approaches have been shown by experiment [7] to be unsuitable for today's clusters with dynamic contents. Content aware policy (CAP) [4], [7], [8] is another third-generation load balancing algorithm which uses request classification and the multi-class round robin scheme for load balancing. The shortcoming of CAP is that, this algorithm does not consider server load states in load balancing decisions.

The effect of overload on web servers has been covered in several works which have taken different approaches to protecting web sites from overload. Chen [12] implements a control theoretic approach which uses a proportional integral controller in a single web server to guarantee service delay by admission control. Andreolini [1] proposes an admission control mechanism for cluster web servers based on the maximum number of connections that each server can support. Xiong [14] implements the same strategy.

Our proposal combines important aspects that previous works have considered in isolation or simply ignored. First, we consider classification of dynamic and static web workloads in our scheme. Second, we focus on load balancing with admission control. Third, our estimation-based load balancing and admission control algorithms are fully adaptive to the available resources in a cluster and workload characteristics instead of using untrustworthy direct measured data such as CPU load.

III. Proposed Cluster Web Server Architecture

As shown in Fig. 2, the web switch is used to fairly distribute incoming workloads in the cluster by request classification. The order and number of requests which will be processed is controlled (scheduling and admission control), and a suitable web server is dynamically selected for the request assignment (dispatching).

We introduce the concept of class to separate requests with widely differing CPU demands. Requests with similar CPU demands are mapped into one class of requests denoted by j (j=1,⋯,C) according to offline profiling procedure. The classification module in the web switch parses each incoming request URL to extract its filename. The classification module then searches in a lookup table to find the class of each request. The class of each request is attached to it as a tag which is used in scheduler and admission control as well as load balancer sections.

When a new request arrives at the web switch, the classification module uses a request URL and a lookup table to determine the class of the request. The classification module uses the URL field in the HTTP header as input, and reads the mapping information from the lookup table. The lookup table

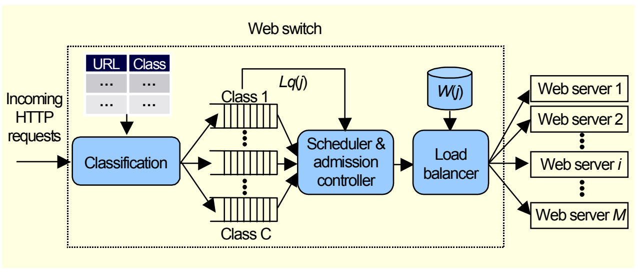

Fig. 2. Block Diagram of cluster web server architecture.

is constructed from CPU demand profiles of various requests. After a request has been classified, the queue module is invoked. The queue module implements a set of first-in first-out (FIFO)-like queues, one for each request class. The queue module suspends incoming requests and adds each of them to the queue corresponding to its class. After this phase, the scheduler selects requests from queues according to the scheduling algorithm and assigns them to the admission controller. The scheduler runs while at least one nonempty queue exists. The admission controller makes a decision to accept or reject a request. Accepted requests are assigned to a dispatcher module. The scheduler and admission controller module use the number of waiting requests in each queue as $Lq(j)$ to make decisions for the next request selection and acceptance. After the dispatcher receives a request from the admission controller, it selects one of the M web servers in the cluster based on the load balancing algorithm. The algorithm selects web server i (i=1,⋯,M) which is estimated to have a higher available capacity and assigns the request to it. The dispatcher estimates the available capacity of each server by an algorithm described in section V.

The dispatcher has a sufficient number of counters (equal to the number of classes multiplied by the number of servers) to hold the status of each web server for each class. These counters are used to track the number of outstanding requests. When a request is received by the load balancer, it reads the tag of the request to determine its class. The load balancer then assigns the request to a web server with higher available capacity. The dispatcher continuously updates the status counters and available capacity when a request is assigned to a web server or when the processing of a request is finished in the web server. When a web server completes processing a request, the dispatcher sends the response to the client and updates the available capacity of the related server. In the following sections the detailed functionality of each part of the system is presented.

IV. Web Server Workload Classification

There are several types of web objects that are generally served by web servers [7], [8]. In most cases, we can classify the web objects into dynamic and static requests. Static requests include HTML pages with embedded objects, such as small pictures which can be cached in memory. Each static web object is a file and can be classified into a certain range of sizes. Since the service time of a static request is proportional to the size of the file [19], static requests can be classified based on their sizes. Note that static requests have small CPU demands [7], [11]. The processing of static requests consists of two tasks: reading a file from a disk or cache memory and transferring it through the network interface. In the past, disk and network resources both created bottlenecks of web servers for static services. Nowadays, with high bandwidth networks and a large amount of RAM in servers for caching contents, the bottleneck problem for static requests has shifted to CPUs because of the context switching overhead of static requests.

Nowadays, most web sites support dynamic contents for rich Internet applications. Dynamic requests consist of dynamic contents that are generated by server side scripting languages (such as PHP, PERL, and JSP) or by enterprise web applications (such as EJB and ASP.NET). Therefore, dynamic contents cannot be fully cached. The contents of dynamic requests are not known in advance and must be retrieved from the web and database servers. Dynamic requests may be as simple as the sum of bill items which do not require intensive CPU resources, or as complex as the content of an e-commerce secure site which requires SSL protocol processing with intensive use of CPUs [7], [8]. Also, dynamic contents which are generated by database-driven web applications make intensive use of CPUs both in the web and database servers.

As mentioned above, to have a better estimation of the impact of each request on the web server load, we classify dynamic requests into several classes based on their impact on server resources. Since the CPU is the main source of bottlenecks in the generation of dynamic contents [3], [11], dynamic requests can be classified according to their CPU demands [5], [7], [8].

The files which were used in our experiments were generated using the specifications in Tables 1 and 2. The workloads are classified into 7 classes, C1 to C7. A name associated with the file size is assigned to each static file. A PHP script is used for dynamic loads. The script receives a variable parameter which determines the execution time of the script. The script reads the input parameter and repeats a one-millisecond operation in a loop according to the requested execution time parameter. The dynamic request URL may take the form of the following example: http://www.example.com/test.php?time=50. Here, 50 is the input parameter, which can be varied by the user. Generally, in a real web site, all of the files which are used as content are known in advance. A web site uses a limited number of files, and this was also true in our experiments. After all the files were generated, we ran an offline workload profiling procedure on a web server. With the help of profiling each request separately, we could determine CPU demand, service time, and the number of critical connections ($Nc(j)$) for each file. Therefore, we could use clustering techniques to classify the files with similar CPU demands (similar service time) into one class. Note that our proposed algorithm has a higher degree of accuracy when we use a higher number of classes, but at the same time its processing overhead will also increase. A solution is to heuristically determine the number of classes as K and use algorithms such as K-mean clustering for classification. After the request classification is finished, the average CPU demand and the average $Nc(j)$ in each class are computed and used in the weight ($W(j)$) determination procedure. In addition, a lookup table is generated from the results of the classification (Fig. 2). The lookup table is a mapping between file names in a workload and their associated classes.

V. Server Available Capacity Estimation

Throughput of a web server is a good criterion of server capacity. Usually, the throughput curve shows an inverted U shape with increments in a load. Throughput rises initially, as the rate of requests increases, and then peaks when a bottleneck resource (in this case CPU) on the web server reaches to maximum utilization limit. Once a resource reaches its maximum usage, queuing for that resource begins, causing throughput to drop. This point is called the saturation point, and the number of requests at this point is called the critical number of requests. To ensure that a server can handle requests with an acceptable mean response time, any number of requests close to the critical number of requests should be avoided. Because the saturation point of a web server is workload dependent, we need to determine the critical number of requests for each class of requests separately.

As previously mentioned, we introduce the concept of class to separate requests with widely differing CPU demands. The web switch maps requests with similar CPU demands into one class of requests. We give each class j a normalized weight $W(j)$ which shows the average CPU demand of class j in comparison to other classes of requests. Without loss of generality in our approach, we assume a homogenous cluster architecture. With a heterogonous cluster, it is necessary to determine $W(j)$ for each server node separately.

If the critical number of requests for each class j obtained from profiling is assumed to be $Nc(j)$, the minimum value of the critical number of requests among all classes can be determined as

$$Nm = \min_{j}\left\{Nc(1),\cdots,Nc(j),\cdots,Nc(C)\right\}. \qquad (1)$$

The value of *Nm* is related to a class of requests with a higher CPU demand. Therefore, we can calculate weight *W*(*j*) for each class *j* of requests as

$$W(j) = \frac{Nc(j)}{Nm}, \qquad j \in \{1,\cdots,C\}. \qquad (2)$$

We assume that the maximum capacity of each web server *i* in the cluster is *Nm* requests from the highest CPU intensive class. From this, we can estimate each server load as a weighted sum of the number of requests from each class *j*. Assume that in each instance of time in the web switch a vector *N*(*i*) is associated with each web server *i* in the cluster, and the number of outstanding requests from each class *j* in web server *i* is tracked as follows:

$$N(i) = \left\{n(i,j) \middle| j = 1,\cdots,C\right\}. \qquad (3)$$

From this, we can approximate the load of each web server *i* as *L*(*i*) which is calculated as

$$L(i) = \sum_{j=1}^{C} \frac{n(i,j)}{W(j)}. \qquad (4)$$

Weight *W*(*j*) determines the contribution of requests from class *j* in the total load of each web server. According to (2) and (4), the available capacity of each web server *i*, *AC*(*i*), can be calculated as

$$AC(i) = Nm - \sum_{j=1}^{C} \frac{n(i,j)}{W(j)}. \qquad (5)$$

We use the available capacity *AC*(*i*) as a criterion of the web server's ability to service requests. The dispatcher selects the web server *i* which has the maximum *AC*(*i*) value of all servers and assigns the next request to it. Upon the request assignment to server *i* or when the processing of a request in that server is finished, the *AC*(*i*) value is updated. We simply add/remove 1/*W*(*j*) term to/from the value of *AC*(*i*) in response to any changes in the number of requests. Therefore, in each instance of time, *AC*(*i*) dynamically shows the current available capacity value in web server *i*.

VI. Scheduling Algorithm

We employ a new probabilistic scheduling algorithm which gives processing preference to a class of requests which has a larger number of requests waiting in the queue of the web switch. Due to the highly variable nature of queue length, we use an exponentially weighted moving average scheme to smooth the results from consecutive epochs. Assume that the actual length of queue *j* is *Lq*(*j*), sampled at every Δ*t* units of time. The smoothed length of queue *j* can be calculated from (6) every Δ*t* units of time:

$$L_j(t) = \alpha Lq(j) + (1-\alpha)L_j(t-1), \qquad 0 < \alpha < 1. \qquad (6)$$

Here, α is a constant. A higher value of α causes the system to response faster to changes of queue length. Therefore, we use α=0.8 as a typical value through our experiments. According to the smoothed length of the queue, we define access probability for each class of requests as

$$P_j(t) = \frac{L_j(t)}{\sum_{j=1}^{C} L_j(t)}. \qquad (7)$$

Access probability dynamically illustrates the relative contribution of requests from each class in total workload. We define a probability range for each class *j* based on its access probability as

$$\left(\sum_{j=0}^{i-1} P_j(t), \sum_{j=0}^{i} P_j(t)\right), \quad i = 1,2,\cdots,C, \qquad (8)$$

$$\text{where} \quad P_0(t) = 0 \quad \text{and} \quad \sum_{j=0}^{C} P_j(t) = 1.$$

We use the probability ranges to schedule requests in each interval Δ*t*. Generally, the priority levels are assigned to the class of requests stored in a queue. Each queue's priority is related to its $P_j(t)$ value. These priorities are dynamic and change according to the incoming workload mixture. A queue with higher $P_j(t)$ has higher priority, and more requests are read from it. The scheduler first should select a queue from which to fetch the next request. Then, the scheduler generates a random number between 0 and 1. The value of the random number falls into one of the ranges defined in (8), so the queue with index *j* related to the selected range is chosen. The scheduler then fetches a request from the selected queue and assigns it to the admission controller. Therefore, lower priority queues have a lower chance than higher priority queues to flow.

VII. Admission Control Algorithm

The ideal behavior of an overloaded web server is to keep serving requests at its maximum capacity, even when the imposed load is beyond its capacity; however, servers are usually overloaded. To prevent overloading, an admission control mechanism should be devised in the web switch. Simple admission control mechanisms typically drop requests regardless of their impact on server resources. However, requests should not be treated equally. Different types of requests have different CPU demands; therefore, it is essential for the admission controller to consider these differences. Also, service providers are advised to rely on rough guidelines for total utilization and avoid peak CPU utilization of over 70% [20]. Therefore, our admission controller algorithm begins its work when the average available capacity (ACC) of web servers in cluster becomes less than 0.3:

$$ACC = \frac{\sum_{i=1}^{M} \frac{AC(i)}{M}}{Nm} \le 0.3. \qquad (9)$$

The proposed admission control algorithm considers both the CPU demand of various classes of requests and their population in the web-switch queue to make decisions. We define the acceptance probability vector *PA*(*t*) which is calculated every Δ*t* units of time (when the ACC value is less than 0.3) as

$$PA(t) = \{Pa(1), \cdots, Pa(j), \cdots, Pa(C)\}, \qquad (10)$$

where *Pa*(*j*) is the probability of accepting a request from class *j* and can be calculated as

$$Pa(j) = \frac{P_j(t) \times W(j)}{\sum_{j=1}^{C} P_j(t) \times W(j)} \times (ACC \times k). \qquad (11)$$

The first term is related to the weighted access probability. Therefore, a request from a class with a lower CPU demand and longer queue length has a higher chance of being accepted. The second term in (11) is a reduction term which permits only some requests to be accepted. This term is related to the ACC. Constant *k* is a scaling factor which maps the range [0,0.3] into [0,1], and its value in this case is 1/0.3=3.333.

When a request arrives at the admission controller, it generates a random number between 0 and 1. If the generated number is smaller than the accepted probability of the request class, the request is accepted and assigned to the dispatcher; otherwise, it is dropped. In the implemented system, when the admission controller rejects a request, an error message (HTTP code 500) is sent to the client, and at the same time, the associated network socket is closed. A benchmarking tool recognizes the request as unsuccessful and logs it. Note that when the ACC is higher than 0.3, all requests are accepted.

VIII. Performance Evaluation

To evaluate the proposed load balancing algorithm, it was implemented on a cluster web server, and cluster throughput and average response time were selected as criteria [4]-[8]. We also implemented two commonly used load balancing algorithms WRR [4]-[8] and CAP [5]-[8] for comparison. We used the average CPU load as a load index for the WRR algorithm. We used the same classification method for CAP and the proposed estimation-based algorithm. In the following sections, the workloads and implementation setups used in our experiments are given.

Table 1. Static workload specifications.

Class	File size range (kB)	Probability of access
C1	0.1<X<1	0.35
C2	1<X<10	0.5
C3	10<X<100	0.14
C4	100<X<1000	0.01

Table 2. Dynamic workload specifications.

Class	Processing time (ms)	Probability of access
C5	10	0.5
C6	50	0.3
C7	100	0.2

1. Experimental Workload

As previously mentioned, an actual web site may serve different percentages of static and dynamic contents. Therefore, we considered two main types of requests (static and dynamic) in synthetic workloads in our experiments. In [19], it was confirmed that the proportion of static requests in the Web has a heavy-tail distribution. In this kind of distribution, most documents are small in size (a few kB) and a small number of files are larger. Large files tend to contribute to the majority of server loads. Also, the service times of static files depend directly on the size of the files [19]. Larger files require longer service times; therefore, we used four classes of files with various size ranges and various access probabilities as the static file sets in our synthetic workload. The choice of workload and its parameters was adopted from frequently used benchmarks such as WebStone [21] and SPECweb99 [22]. Table 1 summarizes the specifications used for static file sets in the synthetic workload.

As in [7], [8], we also considered three classes of dynamic requests with various ranges of service time and occurrence probabilities as dynamic file sets in the synthetic workload. According to Table 2, the requests in class C5 emulate an activity that stresses the CPU, such as ciphering in a SSL connection. The requests in class C6 emulate queries to a database, and the requests in class C7 emulate queries to a database and ciphering the results. The request classes are characterized by a negative exponential distribution for the service time and have means of 10, 50, and 100 ms, respectively. Table 2 summarizes the specifications for dynamic file sets of the synthetic workload.

In the final synthetic workload, 80% of the file sets are dynamic, and 20% are static. The high percentage of dynamic file sets in the final synthetic workload conforms with workload characterizations of typical E-commerce sites [23], [24].

2. Experimental Architecture

The cluster web server consisted of 17 machines. One machine was used as the web switch, and the other 16 machines were used as web server nodes. The web server nodes were AMD Athlon 3.2 GHz CPUs with 512 MB of DDR RAM. The web-switch node was a dual AMD Opteron 2218 dual-core 2.6 GHz CPU with 2 GB of DDR RAM. All the nodes were connected through a high-speed gigabit LAN switch. We used enough 2 GHz AMD Athlon machines as the client emulators to ensure that they would not become bottlenecked in any of our experiments.

The distributed architecture of the cluster was hidden from the clients via a unique virtual IP address of the web switch. All the machines in the cluster ran Linux kernel 2.6 as an operating system. Also, we used Apache v.1.3.39 [25] as a web server, configured with a PHP v.5 [26] module as the server side scripting engine. We increased the maximum number of processes for each Apache instance to 512 to avoid connection refusals from the server when numerous clients simultaneously requested services. We observed that with that value, the number of Apache processes never limited the performance. The client workloads for the experiments were generated using a modified version of the synthetic workload generator and a web performance measurement tool called Httperf [27]. We first generated offline traces of synthetic user sessions and then replayed these traces using Httperf. Using the pre-generated traces guaranteed the repeatability of the tests, which is fundamental for a fair comparison between the load balancing algorithms. We varied the load on the site by varying the number of concurrent clients. In the synthetic workload, each *user session* consisted of a sequence of requests separated by *user think time*. The think time and session time were generated from a negative exponential distribution with means of 7 seconds and 15 minutes, respectively. These numbers conform to the TPC-W specifications [24].

IX. Experimental Results

In the following sections, the results of our experiments are presented in terms of mean response time and throughput. Generally, an algorithm that achieves higher throughput and lower average response time better utilizes the cluster resources and fairly balances the loads among server nodes. Web servers normally work below the nominal peak throughput. Note that there is almost a linear relationship between the average response time and the system load in this situation. However, under overload conditions, when a web server receives more requests than its maximum capacity, the response time of the web server starts to fluctuate and grows rapidly with the number of clients. The admission control mechanism helps to alleviate this problem by dropping the excessive requests and keeping the average response time within a certain range. We investigated these two important working conditions, low-load and overload conditions, in our experiments.

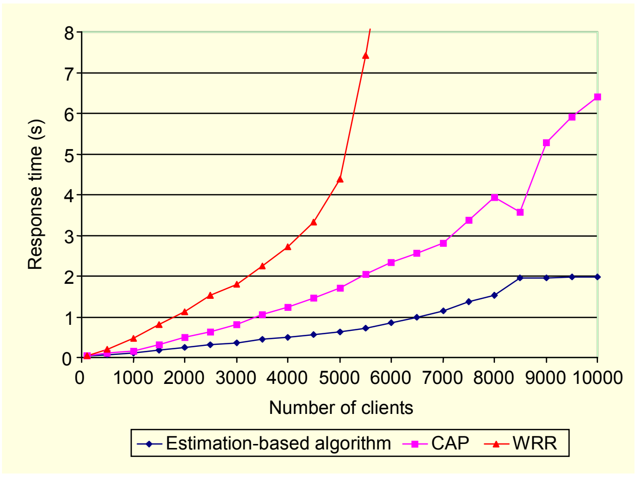

Fig. 3. Mean response time variation vs. number of clients.

1. Mean Response Time

Figure 3 shows the average response time of the cluster web server for three load balancing algorithms under the synthetic workload.

In the low-load situation, the average response time was slightly lower for the estimation-based algorithm in comparison to rival schemes. The WRR scheme with 2 seconds average response time served 3,250 clients. In contrast, the CAP with the same average response time served 5,500 clients. At the same time, the estimation-based algorithm served 8,400 clients in 2 seconds and did not serve any extra clients.

The overload situation starts from the saturation point where the average response time for the WRR and the CAP algorithms become unstable and increase exponentially due to the lack of an admission control mechanism. It can be concluded that a better load balancing algorithm and use of admission control in the proposed estimation-based algorithm improves the performance of the cluster to accept more clients than the other two algorithms.

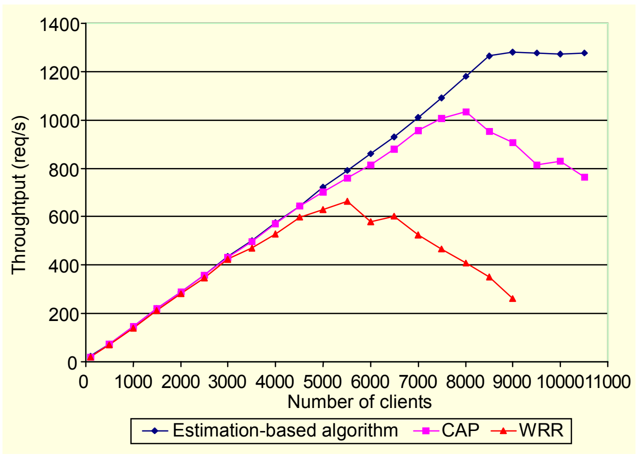

Fig. 4. Throughput variation vs. number of clients.

2. Throughput

Figure 4 shows the throughput of the algorithms on the cluster in terms of the number of requests per second. For light loads, the throughput of a growing number of clients increases linearly.

The estimation-based algorithm achieved slightly higher throughput in comparison to the other schemes. With additional clients, at the level of 1,280 req/s, the throughput reached its peak for the estimation-based algorithm when it was serving 8,500 clients. The saturation point of the CAP algorithm was at 1,020 req/s for 8,000 clients, and that of WRR was at 648 req/s for 5,500 clients. The lower throughput with the WRR and CAP schemes result from the fact that one or more web server(s) in the cluster reaches saturation point. When there are excessive requests in the web server, consequently many of them time out. The total overhead time of each timed-out request adds to unproductive work, while some requests do not even get any service. The wasted time causes drastic drops in the throughput of the cluster web server, while resource utilization remains at 100%.

These results clearly indicate that the load balancer of the estimation-based algorithm works better than that of the two other algorithms. Also, under overload conditions, the estimation-based algorithm provides stable throughput due to the use of the admission control mechanism, while the two other algorithms face unstable conditions and their throughputs of CAP and WRR are diminished. In brief, the average request rate that can be served by the estimation-based algorithm scheme is about 1.25 times higher than with CAP and 1.97 times higher than with WRR.

X. Conclusion

In this paper, we proposed a novel estimation-based load balancing algorithm with an admission control mechanism for cluster-based web servers. The proposed scheme classifies incoming requests based on their service time. We dynamically estimated server loads by tracking the number of outstanding requests from each class in each web server and considered their service demands. Then, the available capacity of each web server was used in load balancing and admission control decisions. We also considered the dynamically evaluated access probability of each class in the incoming workload for proposed scheduling and admission control decisions.

The proposed scheme was implemented in a prototype cluster web server. The experimental results obtained from a synthetic workload showed that, due to better load balancing and admission control of the proposed scheme, the web cluster can accept a higher number of concurrent clients and keep the mean response time of the cluster lower than with the WRR and the CAP load balancing algorithms. However, the dispatcher can become bottlenecked in our proposed scheme. In this situation, we can use a cluster of cluster web servers. It consists of a layer-4 web switch in front of some layer-7 web switches. The layer-4 switch receives all requests and dispatches them among layer-7 switches with simple (RR, WRR, LC) algorithms. Each layer-7 switch implements our proposed algorithm and manages part of the web server. Current layer-4 hardware switches can handle many concurrent clients, so the proposed layer-7 switch does not seem to lead to bottlenecking for the cluster of cluster web servers.

References

[1] M. Andreolini and E. Casalicchio, "A Cluster-Based Web System Providing Differentiated and Guaranteed Services," *Cluster Computing*, vol. 7, 2004, pp. 7-19.

[2] T. Schroeder, S. Goddard, and B. Ramamurthy, "Scalable Web Server Clustering Technologies," *IEEE Network*, vol. 14, no. 3, May/June 2000, pp. 38-45.

[3] J. Challenger et al., "Efficiently Serving Dynamic Data at Highly Accessed Web Sites," *IEEE/ACM Trans. on Networking*, vol. 12, 2004, pp. 223-233.

[4] M. Andreolini, M. Colajanni, and R. Morselli, "Performance Study of Dispatching Algorithms in Multi-tier Web Architectures," *ACM SIGMETRICS Perf. Eval. Review*, vol. 30, no. 2, Sept. 2002, pp. 10-20.

[5] V. Cardellini et al., "The State of the Art in Locally Distributed Web-Server Systems," *ACM Computing Surveys (CSUR)*, vol. 31, June 2002, pp. 263-311.

[6] E. Casalicchio and S. Tucci, "Static and Dynamic Scheduling Algorithms for Scalable Web Server Farm," *Proc. Euromicro Workshop on Parallel and Dist. Proc.*, 2001, pp. 199-176.

[7] E. Casalicchio, V. Cardellini, and M. Colajanni, "Client-Aware

Dispatching Algorithms for Cluster-Based Web Servers," *Cluster Comp.*, vol. 5, no. 1, Jan. 2002, pp. 65-74.

[8] M. Andreolini, M. Colajanni, and M. Nuccio, "Scalability of Content-Aware Server Switches for Cluster-Based Web Information Systems," *Proc. IEEE World Wide Web Conf.*, 2003.

[9] Q. Zhang et al., "Workload-Aware Load Balancing for Clustered Web Servers," *IEEE Trans. Parallel and Distributed Systems*, vol. 16, Mar. 2005, pp. 219-233.

[10] H.K. Lee, "A PROactive Request Distribution (PRORD) Using Web Log Mining in a Cluster-Based Web Server," *International Conference on Parallel Processing (ICPP)*, 2006, pp. 559-568.

[11] A. Chandra et al., "An Observation-Based Approach Towards Self-Managing Web Servers," *Computer Communications*, vol. 29, May 2006, pp. 1174-1188.

[12] L. Chen, Y. Lu, and T.F. Abdelzaher, "Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers," *IEEE Parallel and Distributed Systems*, vol. 17, 2006, pp. 1014-1027.

[13] V. Cardellini et al., "Web Switch Support for Differentiated Services," *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, Sept. 2001, pp. 14-19.

[14] Z. Xiong and P. Yan, "A Solution for Supporting QoS in Web Server Cluster," *Proc. of International Conference on Wireless Communications, Networking and Mobile Computing*, vol. 2, no. 23-26, Sept. 2005, pp. 834-839.

[15] V. Cardellini et al., "Mechanisms for Quality of Service in Web Clusters," *Computer Networks*, vol. 17, Dec. 2001, pp. 761-771.

[16] V. Cardellini, M. Colajanni, and P. Yu, "Request Redirection Algorithms for Distributed Web Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 14, no. 4, April 2003, pp. 355-368.

[17] M. Mitzenmacher, "How Useful is Old Information," *IEEE Trans. Parallel and Distributed Systems*, vol. 11, Jan. 2000, pp. 6-20.

[18] M. Dahlin, "Interpreting Stale Load Information," *IEEE Trans. Parallel Distributed System*, vol. 11, no. 10, Oct. 2000, pp. 1033-1047.

[19] B. Schroeder and M.H. Balter, "Web Servers under Overload: How Scheduling Can Help," ACM *Trans. Internet Technology (TOIT)*, vol. 6, no. 1, Feb. 2006, pp. 20-52.

[20] A. Cockcroft and B. Walker, *Capacity Planning for Internet Services*, SUN Press, 2001.

[21] Webstone: http://www.mindcraft.com/webstone.

[22] Specweb99: http://www.spec.org.

[23] RUBIS benchmark: http://rubis.objectweb.org/

[24] Transaction Processing Council, http://www.tpc.org/.

[25] Apache: http://www.apache.org.

[26] MySQL Database: http://www.mysql.com/.

[27] D. Mosberger and T. Jin, "Httperf: A Tool to Measure Web Server Performance," *Proc. USENIX Symp. Internet Technologies and Systems*, 1997, pp. 59-76.

Saeed Sharifian received his BSc degree in electrical engineering from the KNT University of Technology, Tehran, Iran, in 2000, and his MSc degree in digital electronic engineering from the Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran, in 2002. He is currently a member of the Iranian High Performance Computing Research Centre (HPCRC) as a researcher. He is now a PhD candidate with the Department of Electrical Engineering, Amirkabir University of Technology, Tehran, Iran. His research interests include high-performance web server architecture, parallel computing and programming, sensor networks, as well as performance modeling and evaluation.

Seyed Ahmad Motamedi received the BS degree in electronic engineering from Amirkabir University of Technology, Tehran, Iran, in 1979. He received the MS degree in computer hardware in 1981, and the PhD degree in informatics systems (computer hardware) in 1984, both from University of Pierre & Marie Curie (Paris VI), France. Currently, he is a full professor of electrical engineering technology. He was the President of the Iranian Research Organization for Science and Technology (IROST) from 1986 to 2001. His research interests include parallel processing, image processing, microprocessor systems, automation, and biomedical engineering. He has published papers in more than 60 international conference proceedings and scientific journals. He is also the author of three books.

Mohammad Kazem Akbari received his BSc degree in computer engineering from the National (Beheshti) University, Tehran, Iran, in 1984, and the MSc and PhD degrees in computer engineering from the Case Western Reserve University, Cleveland, Ohio, USA, in 1991 and 1995, respectively. He is currently a faculty member with the Department of Computer Engineering and IT at Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran. He is also the Chair of the Iranian High Performance Computing Research Centre. His research interests include parallel processing, grid and cluster computing systems, as well as mathematical modeling. Dr. Akbari is a member of the ACM and the scientific committee of the Computer Society of Iran.