IEICE TRANS. INF. & SYST., VOL.E86–D, NO.7 JULY 2003

PAPER

Content Sniffer Based Load Distribution in a Web Server Cluster*

Jongwoong HYUN†, *Student Member*, **Inbum JUNG**††, **Joonwon LEE**†, *and* **Seungryoul MAENG**†, *Nonmembers*

SUMMARY Recently, layer-4 (L4) switches have been widely used as load balancing front-end routers for Web server clusters. The typical L4 switch attempts to balance load among the servers by estimating load using the load metrics measured in the front-end and/or the servers. However, insufficient load metrics, measurement overhead, and feedback delay often cause misestimate of server load. This may incur significant dynamic load imbalance among the servers particularly when the variation of requested content is high. In this paper, we propose a new content sniffer based load distribution strategy. By sniffing the requests being forwarded to the servers and by extracting load metrics from them, the L4 switch with our strategy more timely and accurately estimates server load without the help of back-end servers. Thus it can properly react to dynamic load imbalance among the servers under various workloads. Our experimental results demonstrate substantial performance improvements over other load balancing strategies used in the typical L4 switch.

key words: *clusters, load balancing, World Wide Web, performance evaluation*

1. Introduction

The rapid growth of World Wide Web has introduced a few significant changes to server systems. Even the Web sites with large server capacity have been often overwhelmed by continuous and disproportionate increase in client requests. Growing demand on scalability has led many sites to deploy cluster-based servers that are cost effective and scalable compared to a single huge server [1], [10], [15]. The vast majority of traffic in the early days of Web servers was for delivering relatively small static files. Today, however, its significant fraction is for graphics and multimedia content of various sizes. As the Web is increasingly used as an interface to new applications such as e-commerce, the proportion of requests for dynamic content is increasing [14], [25]. Generating a dynamic page is much more resource intensive than retrieving a static file because it requires execution of additional server programs to construct information when a request is made. High variation of content may cause skewed utilization of the servers. For example, if at worst all dynamic page requests were routed to the same server, the server would be loaded much more than the others in the cluster.

Those changes require the cluster to employ a sophisticated load balancing strategy for attaining scalable performance. Due to architecture transparency and centralized control of load distribution, layer-4 (L4) switches are recently widely being deployed as load balancing front-end routers in Web server clusters [8], [9], [12]. Since the HTTP works on top of the connection-oriented TCP, the L4 switch attempts to balance load among the back-end servers by means of distributing connection requests across them. The scheduling granularity is per connection. For an incoming connection, a server is content-blindly selected mainly on the basis of server load. The subsequent HTTP request(s) on a connection is routed to the same server. Thus, timely and accurate load estimate is vital to achieve the load balancing purpose. As it is infeasible to measure a single metric that directly reflects each server's load at the switch itself, some form of load metrics feedback from the servers is necessary. Unfortunately, the overhead imposed on each server and the feedback delay are not negligible. More overhead is unavoidable to obtain more specific load metrics, and further, it is also not easy to decide the priorities of different metrics. These difficulties may often lead to misestimate of server load, and cause dynamic load imbalance among the servers particularly when content is highly varying.

As an alternative, content-aware router, called layer-7 (L7) switch, intends to route requests to the servers best suited to respond by first establishing a connection with each client, and then by inspecting the content (e.g., Uniform Resource Identifier, URI) in the subsequent request packet [3], [18], [20]. Once a server is chosen, the established connection and the request have to be delivered to the server through the connection migration mechanism such as TCP splicing or TCP handoff [17], [18]. The L7 switch allows many new capabilities such as cache affinity scheduling [18], sophisticated load balancing [7], session integrity [3], and special content deployment [23] at the cost of routing throughput. Due to the overheads of connection establishment, L7 protocol processing, and connection mi-

Manuscript received July 22, 2002.
Manuscript revised December 31, 2002.

†The authors are with the Department of Electrical Engineering & Computer Science, Korea Advanced Institute of Science and Technology, Daejon, Korea.
††The author is with the Division of Electrical and Computer Engineering, Kangwon National University, Chunchon, Korea.

*This work was supported in part by National Research Laboratory Program funded by Ministry of Science and Technology, Korea.

gration, the front-end L7 switch may become a bottleneck even for the typical workload [5], [21].

In this paper, we propose a new content sniffer based load distribution (CSLD) strategy for the L4 switch used in clustered Web servers that provide dynamic as well as static content. The motivation is as follows. If the front-end could properly estimate load of the back-end servers without load metrics feedback from them, more performance improvement would be achieved because of timely load estimation and no server overhead for load measurement. However, our proposed strategy must not significantly reduce the routing throughput unlike the content-aware request distribution strategies used in the L7 switches. For this purpose, the front-end could sniff the requests being forwarded to the servers, and by using the extracted URIs, it could estimate servers' CPU and disk load that would be caused by those requests. As a result, it could direct new connection requests to lightly loaded servers.

Effectiveness of CSLD depends on the accuracy in load estimate. We notice that content in a Web site can be classified based on a knowledge of it. Static files can be classified by the sizes because their fetching costs are proportional to the sizes. In a similar way to that proposed in [25], dynamic content can be classified using the URI patterns such as the desired application and query parameters. The administrator can a priori measure the average processing cost for each class of content requests on a real server. The classification results and a priori measured costs are stored as a few tables in the front-end. Given such tables, sniffing the URIs enables relatively accurate estimation of each server's load. As a result, the L4 switch with our CSLD strategy can greatly reduce dynamic load imbalance among the servers even for workloads with bursty requests and high variation of content, and thus improve the overall server throughput and user perceived latency.

The rest of this paper is organized as follows. Section 2 provides some background information and related work. Section 3 describes our CSLD strategy. In Section 4, we present the simulation results and discuss the performance potential of CSLD strategy. Section 5 presents conclusions and future work.

2. Background and Related Work

Figure 1 shows an example for a Web site that consists of a front-end L4 switch and clustered back-end servers. The L4 switch is implemented as a hardware switch fabric or a software switch, which integrates the TCP router mechanism and the dispatcher within the kernel TCP/IP stack on a workstation or PC. Every client packet for requesting an HTTP service from the site reaches the front-end that has a network interface configured with a virtual IP address (VIPA), the only public IP address for the service [6]. The TCP router maintains a connection table indexed by client IP address and port number. Each table entry includes information on a connection such as timestamp, connection state, and target server. For an incoming packet, the TCP router looks up the table. If an existing connection is found, it directs the packet to the target server. Otherwise, if the packet has the TCP SYN flag set (i.e., connection request), it asks the dispatcher to select a server, creates a new table entry, and then routes the packet to the selected server. The connection table entry is updated according to the flag bits in the TCP header and the timestamp of the packet.

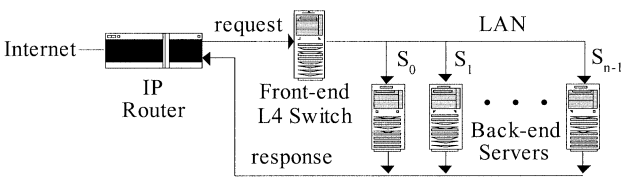

Fig. 1 Web server cluster front-ended by a layer-4 switch.

The TCP routing mechanisms are divided into packet rewriting mechanism and packet forwarding mechanism. The former replaces the destination IP address (VIPA) of an incoming packet with the target server's IP address. The source IP address of the response packet also has to be replaced with the VIPA, which is done by the server or the TCP router depending on the specific packet rewriting techniques. The latter requires that the front-end and the servers are physically connected on a LAN. Once a target server is decided, the TCP router reroutes the packet to the server by changing the MAC address of the packet's Ethernet frame to that of the server, and then by placing it on the LAN. The server directly sends the response to the client. Though the latter lacks geographical scalability, it achieve higher routing throughput than the former which involves address translation and checksum recalculation in the front-end or the servers.

The load balancing strategies used in the L4 switch (dispatcher) are as follows. Round-robin (RR) strategy distributes connections across the servers in a round-robin manner. Least-connection (LC) strategy directs a new connection to the server with the least number of active connections. It does not perform well when the servers have different CPU power. The weighted least-connection (WLC) strategy selects the servers so that the number of active connections in each server may increase in proportion to its CPU power.

The weighted round-robin (WRR) strategy selects the least loaded server based on each server's weight (capacity available for serving new connections). The weights are computed using various load metrics measured in the front-end and the servers. For instance, IBM's Network Dispatcher [12] uses three types of load metrics as follows. Input metrics measured in the front-end includes the number of active connections and the request rate for each server. The front-end obtains for-

ward metrics by periodic server-polling (i.e., sends a simple HTTP "GET /" request to each server and measures the response delay). Host metrics, measured by user level daemon processes in each server, include very specific load metrics such as the number of active processes, the memory usage, the number of open sockets, and the length of wait queue for disk I/O. A user level daemon in the front-end gathers those metrics, and through complex aggregation procedures, computes a load index, which determines the weight.

Authors in [7] proposed a content-aware load balancing policy called CAP for L7 switches. Though we independently proposed our CSLD strategy, there are some similarities between them. Both focus on load balancing in clustered Web servers that provide dynamic as well as static content which have different resource requirements. Both attempt to classify client requests based on their expected impact on system resources. However, a few fundamental differences exist between them. CAP attempts to keep the servers evenly loaded, at the cost of system scalability. As in other content-aware request dispatching strategies, the L7 switch may result in the bottleneck. Per-request dispatching granularity is used to improve server performance for HTTP/1.1 persistent connections, which imposes additional connection migration overhead. CAP uses a coarse-grained classification of requests, such as CPU-bound, disk-bound, both CPU- and disk-bound, and non-resource-intensive classes, together with a single high processing cost assumed for all dynamic content requests. It distributes the classes among the servers so that it may avoid skewed assignment of any particular resource-intensive class to a server. On the other hand, CSLD is a connection dispatching strategy for L4 switches. It intends to compensate content-blindness of L4 switch with content sniffing based load estimation and distribution while not significantly reducing the routing throughput of L4 switch. Further, CSLD allows a fine-grained content classification based on file size, URI pattern, and a priori measured average processing costs for each class of requests.

3. Content Sniffer Based Load Distribution

3.1 Main Idea

In this section, we propose our content sniffer based load distribution (CSLD) strategy. For simplicity, the following assumptions hold for the rest of this paper. As shown in Fig. 1, the cluster consists of N homogeneous back-end servers and a front-end software L4 switch that integrates the dispatcher and the packet forwarding TCP router within the kernel. All servers are equally able to respond to any content request, and run Apache that follows the process-driven model [16].

A client request usually passes through a sequence of processing steps in the server. Its processing cost includes a fixed portion and a variable one. The former corresponds to the CPU time used in the kernel and the user-level Apache code for the basic processing steps, such as establishing and tearing down a connection, parsing an HTTP header, and authorizing and logging the request. The latter depends on the requested content, and includes the costs for fetching a file from disk or RAM, creating a dynamic content by invoking an appropriate handler, and transmitting the response.

The objective of CSLD is to balance load among the servers by properly estimating load without costly load metrics feedback from the servers. The basic idea is as follows. The front-end dispatcher sniffs each client request being forwarded to the target server. Using the sniffed URI, it estimates the server's CPU and disk I/O time that would be consumed to process the request, which are called CPU cost and disk cost (denoted by *ccost* and *dcost*) respectively. They are used to compute the current server load, on the basis of which new connections are directed to lightly loaded servers. To put the idea into practice, we need an effective way to estimate those costs by URIs. We notice that the administrator can possibly classify content in the site into several classes based on a knowledge of it. Static files can be classified by their sizes because the cost for fetching a file is proportional to the size. Dynamic content can be classified according to the URI patterns such as desired application and query parameters.

As shown in Fig. 2, the dispatcher that implements CSLD consists of content sniffer, load estimator, connection dispatcher, and the following tables which are used to classify the sniffed URIs and to estimate the costs. The file table is a global hash table indexed by URIs, and includes an entry for every static file in the site. Each entry has a few fields such as file name (URI), size, last modification time, and latest reference time in each server. All the fields but the last one are given by the administrator. The cost table includes CPU time breakdowns for the basic request processing steps, *ccost* for transmitting a unit size of response, and *dcost* for reading a data block from disk, a priori measured by the administrator on a real server machine.

The class table includes the average *ccost*, *dcost*,

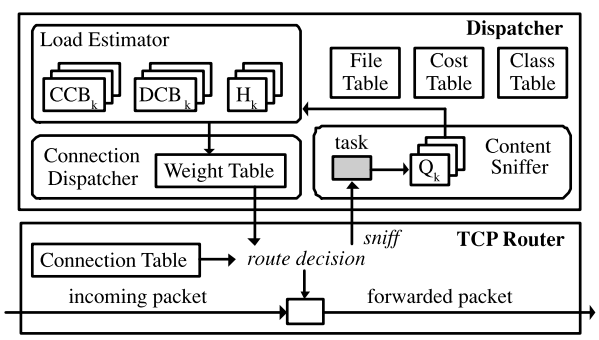

Fig. 2 L4 switch that implements our CSLD strategy.

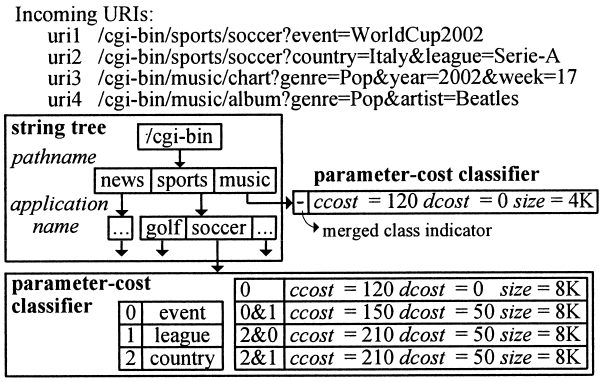

Fig. 3 An example of class search in the class table.

and size for each class of dynamic content requests, a priori measured on a real server. It consists of a string tree proposed in [25] and parameter-cost classifiers. For a sniffed dynamic URI, the content sniffer searches the string tree to determine the pathname and application name, and looks up the parameter-cost classifier to locate the matching class. Then, the average *ccost*, *dcost*, and size of the matching class is obtained. Our simple measurements in a few small Web sites showed that URIs with the same pathname or application name and similar parameters often result in similar costs and page sizes. Dynamic content requests that are similar in the costs as well as in the URI patterns are merged into the same class. Figure 3 shows such an example. Despite the same pathname (/cgi-bin/sports) and application name (soccer), uri1 and uri2 are classified into different classes because their parameters and the corresponding costs in the parameter-cost classifier are different. On the other hand, uri3 and uri4 are classified into the same class because their application names are different, but their pathnames and costs are the same. When there are a lot of merged classes, the class search overhead may be reduced. However, the overhead for URI classification and cost estimation will largely depend on the service characteristics of a real Web site.

Every incoming packet is routed by the TCP router to the target server. For a connection request, the connection dispatcher selects the server with the least load (largest weight) on the basis of the weight table, which is periodically updated by the load estimator. It uses the same weighted round-robin algorithm as that described in [12], [24]. For the HTTP request arriving on an existing connection, the target server is already specified in the connection table maintained by the TCP router. The request is sniffed by the content sniffer, and the corresponding server load is estimated by the load estimator, as described in the next subsections.

3.2 Content Sniffer (CS)

Once the target server S_k ($k = 0, \ldots, N-1$) is decided for an incoming packet, the TCP router passes CS some parameters such as the request timestamp, the target server identifier, the pointer to the packet data, and the round-trip-time (RTT) of the connection. Making these parameters does not impose additional overhead because the TCP router always maintains such information for its own duties.

The packet is sniffed and classified as follows. CS creates a data structure called the task that includes the timestamp, RTT, class, *ccost*, and *dcost* fields. The timestamp and RTT fields are set using the passed parameters. If the packet has a SYN, FIN, or RST flag set in the TCP header, it is regarded as a connection request. In the case of request packet, the HTTP header and URI are parsed. If the URI is a dynamic content, it is regarded a dynamic request, and the class table is searched to find the class that matches the URI pattern. Otherwise, a hashing function is computed on the URI to look up the file table. If the table has no entry for the URI, the packet is regarded an invalid (INV) request. If the HTTP header has an If-Modified-Since (IMS) field with the value larger than the last modification time in the corresponding file table entry, the packet is expected to cause a "Not-Modified" response. Other valid URI is regarded a static file request. The *ccost* for a connection, INV, or IMS request is set appropriately by referring to the cost table. The *ccost* and *dcost* for a dynamic request are set to those of the matching class in the class table. For a static request, the *ccost* is set to the default value for the basic request processing steps, specified in the cost table, whereas setting the *dcost* is deferred until being done by the load estimator. When sniffing is finished, the task is placed in the queue Q_k specified for the target server.

3.3 Load Estimator (LE)

LE is invoked periodically. Suppose that the period is P and LE is invoked at time T_{i+1}. The queue Q_k will contain the tasks sniffed from T_i to T_{i+1}, which correspond to the requests routed to the server S_k during that period. LE works in two steps as follows.

- First Step - Cost Estimation

In this step, while taking the tasks out of the queue Q_k one by one, LE estimates when and how much *ccost* and *dcost* would be required to process the corresponding request in the server S_k. The process execution for a request is modeled as a sequence of CPU bursts and I/O bursts. LE maintains two circular arrays to record *ccost* and *dcost* for each server. They are called CPU cost board, denoted by CCB_k, and disk cost board, denoted by DCB_k, respectively. Figure 4 shows their structure where *max_slots* is the total number of slots in an array and *slot_len* is the maximum cost value that can be recorded in a slot. For example, when *slot_len* is 50 milliseconds (ms) and *max_slots* is 200, one array can record up to 10 seconds of *ccost* or *dcost*, which

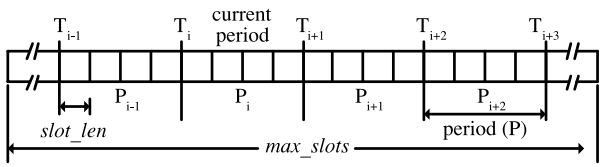

Fig. 4 Structure of the cost boards (CCB_k and DCB_k).

corresponds to 50 periods when P is 200 ms. Since the cost boards are circular arrays, all the slots in the period P_i are cleaned at the beginning of the first step. LE maintains two variables TC_k and TD_k that indicate the start (or finish) time of a CPU bursts and a disk I/O bursts respectively. They are used to index the slots in CCB_k and those in DCB_k respectively.

LE checks the timestamp and the class in a task drawn from Q_k. If TC_k is smaller than the timestamp in the task, it is updated. The *ccost* in the task is added to the slot in CCB_k indexed by TC_k, and then added to TC_k as well. If the value in the slot becomes larger than *slot_len*, the difference is added to the subsequent slot(s). TD_k is updated to TC_k value if it is smaller. In the case of a dynamic request, (if any) *dcost* is added to the slot in DCB_k indexed by TD_k, and then added to TD_k as well. If the class indicates a static file request, LE needs to predict whether the request results in a cache miss in the server because its processing may involve disk read. Various techniques may be possible for this purpose, including our proposed one described below. For each server, LE maintains a history table H_k to periodically (e.g., every second) record the estimated total size of data that has been read from disk in recent hours (e.g., 6 hours). LE refers to the corresponding file table entry for the latest reference time of the URI in the server S_k, denoted by TR_k(URI). If it is zero, the request will result in a cold cache miss. Otherwise, by referring to the history table, LE estimates the total size read from disk since TR_k(URI). If the size is larger than the server cache size, the request is expected to cause a cache miss. Then, LE computes *dcost* according to the file size and the cost table, adds it to the slot in DCB_k indexed by TD_k, and adds it to TD_k as well. Assuming that TD_k is the time when disk data is written into memory, LE adds the file size to the appropriate H_k table entry. The TR_k(URI) is updated to TD_k (or TC_k in the case of a cache hit).

The *ccost* for transmitting the response is computed based on the size and the cost table. If the size is smaller than the TCP send buffer size, a whole data will be immediately transmitted. Small static files, relatively small dynamic pages, responses to INV requests, and "Not-Modified" responses belong to such a case. Then, the *ccost* is added to the slot in CCB_k indexed by TC_k, and then adds it to TC_k as well. Otherwise, the transmission time depends on the TCP's slow start, sliding window protocol, and congestion window size

```
/* Statistics estimated in LE */
MR     - Overall cache miss ratio in the cluster
MR_k   - Cache miss ratio in S_k
C_r    - Average ccost per request
C_m    - Average dcost per cache miss
C_p    - Average ccost per period
N_r    - Average number of requests per period
F_dreq - Proportion of dynamic requests
T_dreq - Average ccost of dynamic requests

/* computation of each server's aggregate load index L_k */
if (MR < miss_l) {
  if (C_p < load_l)
    L_k = (total ccost for P_i+1 in CCB_k)/C_r;
  else if (C_p < load_m)
    L_k = (total ccost for P_i+1 and P_i+2 in CCB_k)/C_r;
  else
    L_k = (total ccost for all the next periods in CCB_k)/C_r;
  if (MR_k > miss_m)
    L_k + = (total dcost for P_i+1 and P_i+2 in DCB_k)/C_m;
}
else {
  L_k = (total dcost for all the next periods in DCB_k)/C_m;
  if (MR < miss_h) {
    L_k + = (total ccost for P_i+1 in CCB_k)/C_r;
    if (C_p > load_h || (C_p > load_m && MR < miss_m))
      L_k + = (total ccost for P_i+2 in CCB_k)/C_r;
  }
}
/* computation of each server's weight W_k */
for (min=max=k=0; k<N; k++)
  if (L_k > max)        max = L_k;
  else if (L_k < min)   min = L_k;
t_diff = N_r/N;         /* adjustable based on statistics */
dec = (max − min ≤ t_diff) ? 1.0 : t_diff / (max − min);
for (k=0; k<N; k++)
  W_k = (max. allowable weight) − (L_k − min) * dec;
```

Fig. 5 Pseudo-code example of load aggregation in LE.

(cwnd), and the RTT. However, we took a simplified model for TCP bulk data transfer. LE assumes that the server writes cwnd segments into the send buffer every RTT while exponentially increasing the cwnd value. The *ccost* is divided into multiple parts, and each part is added to an appropriate slot in CCB_k.

Through periodically performing the first step for all N servers, LE gathers and maintains some workload statistics for recent periods, as shown in Fig. 5.

- Second Step - Load Aggregation

When this step starts, all the slots in CCB_k and DCB_k except for those for the current period P_i may include CPU and I/O requirement of those tasks which were not yet completed in the server S_k until the end of current period (T_{i+1}). These slots correspond to the aggregate load which would remain in the server at worst until time $T_{i+1} + max\_slots * slot\_len - P$. Figure 5 is an example pseudo-code for load aggregation. The threshold variables miss_h, miss_m, and miss_l specify high, medium, and low cache miss ratio respectively. They are given by the administrator through analyzing the daily average cache miss ratio using the statistics gath-

ered in LE or the real server logs. With those thresholds, we merely intend to take into account the situation of significant cache misses after the server startup (cold misses) and to detect (if any) abrupt changes in miss ratio. As reported in [11], [14], [19] and the references cited therein, workloads are usually CPU intensive and disk load has little effect on the overall performance in busy Web servers with large memory, because locality of reference tends to be very high in such systems. The variables load_h, load_m, and load_l are used to differentiate the degree of CPU load given to the entire cluster. The threshold t_diff specifies the upper bound on the difference between the number of new connections directed to the most loaded server and that directed to the least loaded server during the next period.

LE computes each server's aggregate load index and weight as follows. While comparing the cache miss ratio and the average *ccost* per period with the given thresholds, LE selects the slots which belong to all or part of the next periods in CCB_k and DCB_k as the candidates. For example, when the overall load is low, it selects only the next period P_{i+1}. The reason is that the subsequent periods may include the *ccost* for file transmission over the connections with long RTTs, and thus we had better exclude them from computing current server load. By considering the cache miss ratio and the degree of overall load, we can decide the relative importance between CPU load and disk load, and possibly avoid overestimate of server load. The sum of *ccost* and that of *dcost* in those candidate slots are divided by the corresponding unit costs (C_r and C_m) respectively, and then summed up to obtain an aggregate load index L_k. Since the scheduling granularity is per connection, we need to prevent the dispatcher from too sensitively reacting to load imbalance among the servers. Therefore, each server's weight W_k is computed by proportionally reducing the L_k so that the maximum difference among the L_k values may not exceed the threshold t_diff. The weights are recorded in the weight table. LE can be optimized to dynamically adjust those threshold variables according to the workload characteristics. However, to observe the basic performance behavior of CSLD, the same algorithm as Fig. 5 with some fixed threshold values was used in all simulations of the next section.

4. Simulation

4.1 Workload and Simulator

To evaluate the performance of different load balancing strategies under various workloads, we developed a configurable Web server cluster simulator, and conducted trace-driven simulations using the access logs of the 1998 World Cup Web site, gathered on its peak day [22]. Table 1 summarizes the trace characteristics.

Our simulator consists of three models that simulate clients, front-end L4 switch, and back-end servers.

Table 1 Summary of trace characteristics.

No. logs	% static	% dynamic	% IMS	% INV	% HTTP/1.1
73.3M	71.76	0.02	27.87	0.35	19.69

The client model generates a stream of tokenized requests based on the access logs. A token includes the fields such as timestamp, client/server identifier, URI, file size, and HTTP response code. Because the logs lack a lot of information useful for a detailed simulation [4], we made the client model change some fields in the token according to the specified simulation options. Because the logs do not include connection establishment/teardown requests, the client model explicitly generates them when necessary to simulate the connection-oriented HTTP. For those requests, TCP's three-way/four-way handshake procedures are replayed among three models. When all processes are busy and the number of connections pending in the listen socket's queue exceeds the specified limit called the backlog, the back-end server model starts to reject new connection requests. Each rejected connection request is resubmitted to the front-end model, at worst two times using an exponential backoff-based retransmission timer.

To obtain various workloads, we used the following simulation options. The option N specifies the cluster size. The option R_{req} specifies the request rate. The timestamp in a token indicates the time when the token arrives at the front-end model. If this option is explicitly specified, the client model sets the timestamps by generating exponential inter-arrival times. Otherwise, the timestamps are generated based on the timestamps in the access logs. The option RTT specifies the client-to-server round-trip time. Once a new connection is accepted at the server, the subsequent HTTP request arrives at the front-end after at least one RTT. In practice, the client may be connected to the server through the networks with low capacity or high traffic. To study whether the WAN delays may affect load-balancing performance, the client model generates the RTTs using a long-tailed pareto distribution function, and assigns them to new connections. In the next subsection, all simulations use a relatively long mean RTT value (about 150 ms) unless otherwise mentioned.

The option P_{dreq} specifies the proportion of dynamic requests. Since the trace includes few dynamic requests, we made the client model generate additional dynamic requests according to the given P_{dreq}. The option C_{dreq} specifies the set of classes of dynamic requests generated in the client model. The logs lack information on the processing costs of dynamic requests. For simplicity, we assume that dynamic requests are CPU intensive and their processing rate may range from 1/10 to 1/100 of the processing rate of static requests based on the previous study [13]. Thus, dynamic requests are classified into up to 10 classes in the client

model. For example, $C_{dreq} = \{2, 5\}$ indicates that the class 2 and 5 will be generated with equal probability of occurrence. Assuming that the average cost for satisfying static requests is 1 ms, the client model assigns 20 ms and 50 ms to the class 2 and 5 respectively as the mean processing costs. The processing costs for dynamic requests in each class are given by a normal distribution function with the specified mean value. The standard deviation of the distribution is set to 20% of the mean value specified for each class in all simulations of the next subsection unless otherwise mentioned.

The front-end model simulates the TCP router and the dispatcher. The TCP router forwards the tokens from the client model to the target back-end models. For a connection request token, the dispatcher selects a target server according to the specified load balancing strategy. The RR, LC and CSLD strategies work in the same way as described in the previous sections.

Since any specific load estimation technique for the WRR strategy is not known, we experiment with its six variants, shown in Table 2, using three types of load metrics as in [12]. Input metric M_i is the number of active connections. Forward metric M_p is the server-polling response time. Host metric M_h includes four metrics, M_{h1} to M_{h4}, which are the number of connections waiting for being accepted, and the number of processes that are waiting for disk I/O, fetching static files, and creating dynamic pages respectively. In practice, those metrics can be hardly obtained without modifying the server code or running additional processes. WRR_p is the simplest variant since it uses M_p only. The others use all three types of metrics. The most idealized variant WRR_i assumes zero measurement overhead for M_h and the shortest period. To favor the WRR strategy, we made the following ideal assumptions. 1) Those load metrics are periodically measured and gathered with minimal overhead and no delay. 2) The average CPU time for processing static requests and dynamic requests, denoted by T_{sreq} and T_{dreq} respectively, are given by the administrator. 3) The authors in [12] argue that the configuration parameters of different load metrics can be dynamically tuned to the changing nature of workload. However, through the simulation we found that such an automatic tune-up is very difficult. Instead, the best configuration parameters obtained from the simulation were used.

Equation (1) shows how those six WRR variants compute each server's aggregate load index L_k. For all variants but WRR_p, the configuration parameters of input, forward, and host metrics, f_i, f_p, and f_h are set to 0.1, 0.45, and 0.45 respectively. The weight factor $f_{I/O}$ is used to make L_k reflect the overall disk load in the cluster. When disk load is low, medium, or high, it is set to 1, 2, or 4. We multiply M_p by the average request rate given to the server because the server would have been loaded in proportion to the server-polling response time. Since the average CPU cost for dynamic requests is usually larger than that for static requests, we multiply M_{h4} by T_{dreq}/T_{sreq}. We set f_{dreq} to a proper value obtained from the simulation. We found that it should be small (e.g., 0.1) due to the following reasons: 1) When M_{h4} is measured, every process that has been already in execution for creating a dynamic page will use further CPU time smaller than T_{dreq}. 2) Load caused by those processes was already partially reflected on the other host metrics. 3) Since it is not known that how many connections among M_{h1} will request dynamic content, the metric M_{h4} may inaccurately reflects the number of pending dynamic requests.

$$\begin{aligned} L_k = {} & M_i * f_i + M_p * (\text{request\_rate\_assigned\_to\_}S_k) * f_p \\ & + (M_{h1} + M_{h2} * f_{I/O} + M_{h3} \\ & + M_{h4} * (T_{dreq}/T_{sreq}) * f_{dreq}) * f_h \end{aligned} \quad (1)$$

Using the aggregate load indexes, the front-end model computes the weights in the same way as in Fig. 5. Through the simulation, we learned that when the overall load is too low or too high, the threshold t_diff should be reduced to prevent the dispatcher from overestimating load imbalance among the servers.

The back-end model simulates the back-end nodes that run Apache 1.3.16 server, configured with its default key configuration directives [2], [16], on the Linux. It approximates the OS behavior for managing the system resources. We set the process time slice to 100 ms. CPU is shared among server processes by a priority (credit) based preemptive time-sharing scheduling algorithm. The back-end cache uses the LRU replacement policy. We set the cache size to 64 MB. For each incoming token, this model performs a detailed queuing model simulation based on the cost parameters. Table 3 lists the average costs for the basic request processing steps, measured on a 700 MHz Pentium III machine with 256 MB RAM and a 5,400 RPM disk drive.

Table 2 Six WRR variants. (o: used, x: not used, unit: ms)

	M_i	M_p	M_h	Period	Overhead for M_h
WRR_i	o	o	o	200	0
WRR_a	o	o	o	500	5
$WRR_{a'}$	o	o	o	500	10
WRR_b	o	o	o	1,000	5
$WRR_{b'}$	o	o	o	1,000	10
WRR_p	x	o	x	500	0

Table 3 Cost parameters used in the simulator.

Parameters	Cost (us)
connection establishment or teardown	132
process fork	1,400
context switch	35
basic processing steps (parsing URI, etc.)	213
response to an IMS or INV request	30
transmission of 512 Bytes to network	37
disk seek and rotational latency	15,600
disk transfer time for 4 KB block	300

Table 4 Average server latencies in seconds achieved under various workload.

	C_{dreq}	P_{dreq}	R_{req}	RR	LC	WRR_i	WRR_a	WRR_b	$WRR_{a'}$	$WRR_{b'}$	WRR_p	CSLB
Case-1	{3}	5.0%	4000/s	0.0226	0.2172	0.0186	0.0209	0.0229	0.0219	0.0230	0.0245	0.0223
Case-2	{3}	6.4%	6400/s	0.3469	0.5041*	0.1309	0.1759	0.1805	1.4940*	0.2113	0.1835	0.1500
Case-3	{4}	4.0%	4000/s	0.0267	0.2247*	0.0225	0.0254	0.0273	0.0256	0.0275	0.0298	0.0262
Case-4	{4}	6.0%	4800/s	0.1202	0.4606	0.0799	0.0932	0.1008	0.1003	0.1049	1.7166*	0.0829
Case-5	{5}	4.0%	4000/s	0.0428	0.2674	0.0347	0.0363	0.0423	0.0370	0.0429	0.7917*	0.0395
Case-6	{5}	6.8%	4000/s	0.9975	0.6065	0.2564	0.3802	0.4503	2.2888*	0.8784	2.6801*	0.3139

4.2 Simulation Result

Our simulator outputs a few metrics such as throughput, user perceived latency, server latency, and rejection rate. Among others, we select the average server latency as the main performance metric because it gives a better view of the load balancing effect on the clustered servers for various workloads. The server latency is the delay from receiving a client request to sending the first TCP segment of response. The rejection rate indicates the rejection probability of new connections in the servers. It is used as a complementary performance metric when the servers are overloaded.

4.2.1 Comparison of the Basic Performance

Table 4 lists the average server latencies achieved by RR, LC, six WRR variants, and CSLD for various workloads. The asterisk (*) indicates that the rejection rate is not zero. The simulation conditions are as follows. The cluster size (N) was set to 16. Exponential inter-arrival times were generated at the specified request rate (R_{req}). Simulation was conducted for 30 minutes after 15 minutes of warming-up. Case-1, 3, and 5 are relatively light workload. Though Case-2, 4, and 6 are a little heavy workloads, they do not seem to incur peak load because RR shows zero rejection rate.

The most idealized WRR_i performs best of all for any case. Since RR performs better than what we expected, we will evaluate the performance of the others on the basis of RR. CSLD also performs fairly well for all cases. It performs slightly worse than a few WRR variants when the overall load and the variation of content are low (Case-1, 3, and 5). However, it always outperforms RR for any workload. Due to shorter load measurement period, WRR_a outperforms WRR_b in all cases, and WRR_b sometimes performs worse than RR for light workload with low variation of content (Case-1 and 3). This implies that a longer period has an impact on the performance of WRR strategy.

$WRR_{a'}$ and $WRR_{b'}$ are the same as WRR_a and WRR_b respectively except that load measurement overhead is doubled, but perform much worse. $WRR_{a'}$ significantly increases the server latency and causes rejection of many connection requests particularly for heavy workload (Case-2 and 6). This implies that the potential benefits of using host metrics may be reduced or even outweighed by the imposed overhead. So, it may be better to choose a long load measurement period when the overhead is high. WRR_p performs better than RR only when the request rate and the proportion of dynamic requests are high, but the average processing cost for dynamic requests is low (Case-2). For higher C_{dreq}, the server latencies significantly increase even when the overall load is modest because WRR_p often incurs load imbalance among the servers. We repeated the simulation while changing the period of server-polling, but the results were similar. This implies that load estimation based on server-polling only is not sufficient to cope with the variation of content.

Because of the given long mean RTT (150 ms), the LC strategy does not perform well in most cases. It performs better than RR, only when there are many long lived connections due to high variation of content (Case-6). Additional simulations confirmed that given shorter mean RTTs, it performs better. This implies that using input metric only is not sufficient to properly estimate server load in a WAN environment. We will exclude LC, WRR_b, $WRR_{a'}$, $WRR_{b'}$, and WRR_p from the results presented in the rest of the simulations because of their low performance.

Table 5 Average server latencies in seconds achieved by RR as R_{req} increases when C_{dreq}={4} and P_{dreq}=5%.

3.0 K/s	4.0 K/s	5.0 K/s	6.0 K/s	6.2 K/s	6.3 K/s
0.0211	0.0341	0.0567	0.1734	0.4644	1.5899

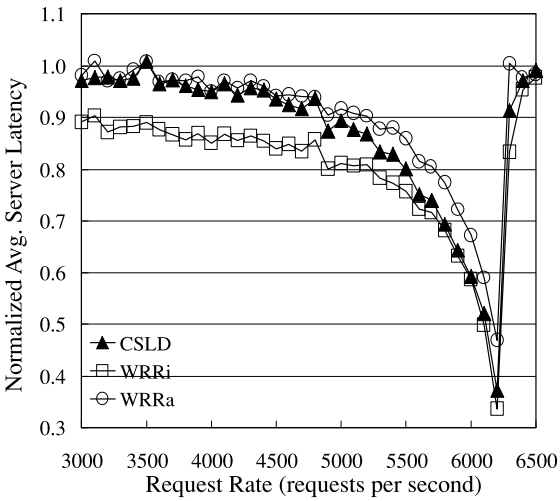

Fig. 6 Normalized avg. server latency as R_{req} increases.

Table 5 shows the average server latencies achieved by RR as the request rate (R_{req}) increases when C_{dreq} is {4} and P_{dreq} is 5%. Figure 6 shows the average server latencies of WRR_i, WRR_a, and CSLD, normalized by those of RR. As R_{req} increases, CSLD achieves higher performance improvement ratio, compared to RR. For heavy workloads, its performance comes close to that of the most idealized WRR_i. When R_{req} is 6,300/s, about 11.35 million of requests are generated for 30 minutes, incurring peak load. In such case, the overall rejection rates caused by RR, WRR_i, WRR_a, and CSLD were about 3.6%, 1.8%, 3.9%, and 2.1% respectively.

4.2.2 Performance Comparison of WRR and CSLD

We conducted the simulations while changing the degree of content variation (C_{dreq} and P_{dreq}). The time for each run of simulation is 1.5 hours with 15 minutes of warming-up. The request rate ranges from 1,860/s to 3,683/s. Inter-arrival times are based on the timestamps in the logs. Figure 7 to 11 show the average server latencies achieved by WRR_i, WRR_a, and CSLD for the workloads with different C_{dreq}, normalized by the corresponding ones of RR. The simulation results showed the behavior that coincides with the results shown in Table 5 and Fig. 6. As the rate of dynamic requests increases, the server latency and user perceived latency drastically increase because of rapid increase in the variation of content and the overall load.

The variation of content (C_{dreq}) grows in the order of Fig. 7 to Fig. 11. Given the same P_{dreq}, the workload of Fig. 7 and that of Fig. 8 incur the same overall load, but the latter involves a little higher variation of content. Similarly, the workload of Fig. 11 incurs only a little more load than that of Fig. 10, but it has much higher content variation. As C_{dreq} increases, CSLD

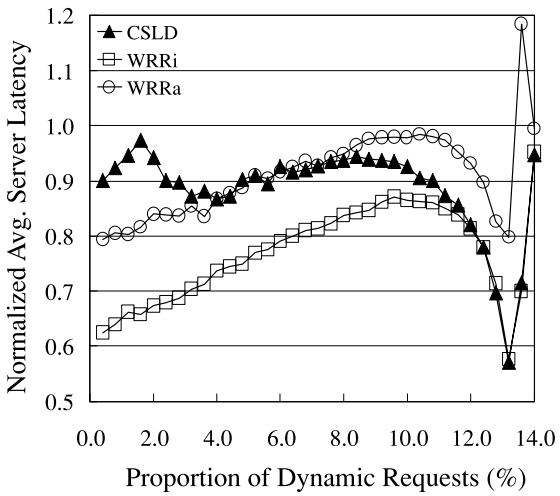

Fig. 7 Normalized avg. server latency as the proportion of dynamic requests (P_{dreq}) increases when C_{dreq} ={3}.

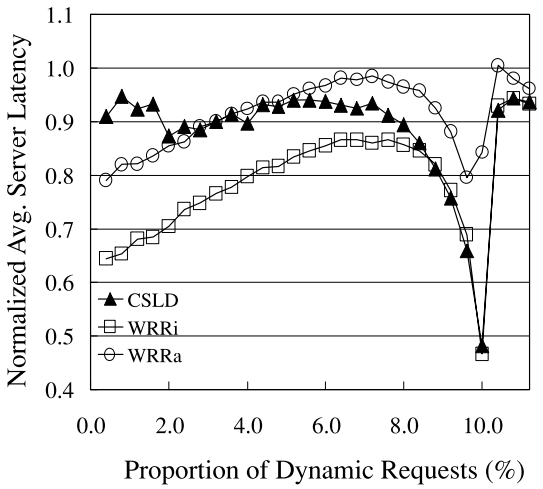

Fig. 9 Normalized avg. server latency as P_{dreq} increases when C_{dreq} ={4}.

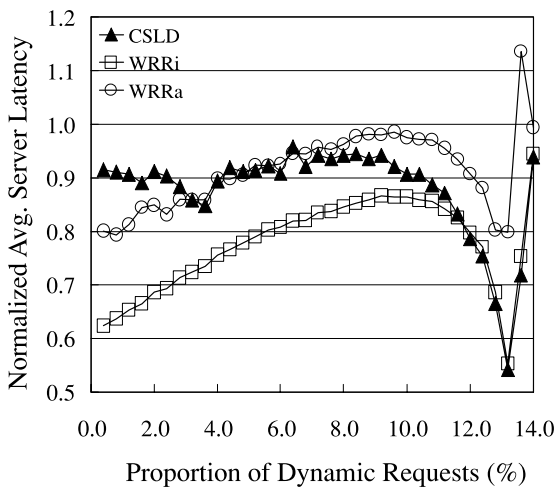

Fig. 8 Normalized avg. server latency as P_{dreq} increases when C_{dreq} ={1,2,3,4,5}.

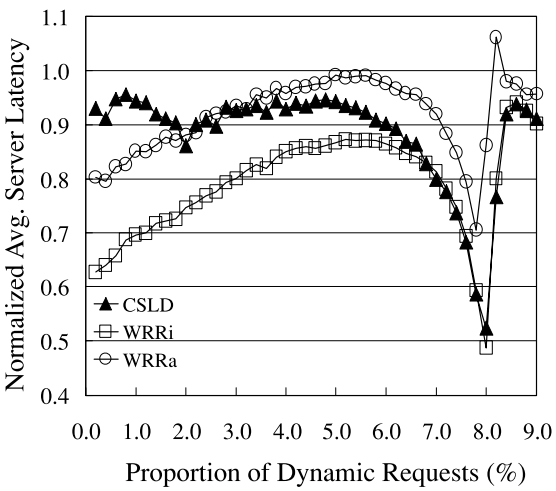

Fig. 10 Normalized avg. server latency as P_{dreq} increases when C_{dreq} ={5}.

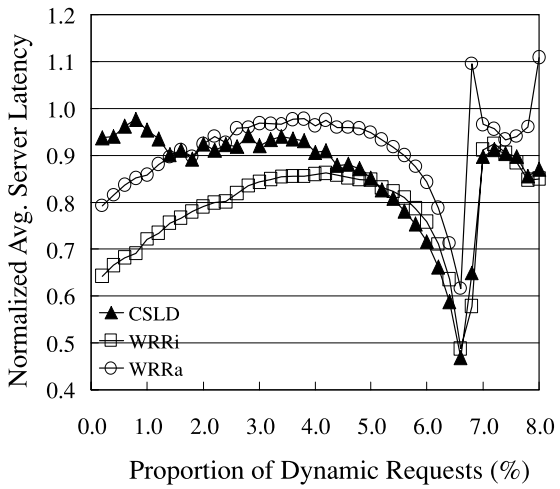

Fig. 11 Normalized avg. server latency as P_{dreq} increases when C_{dreq} ={2,4,6,8,10}.

starts to outperform WRR_a at lower P_{dreq}. For example, it starts to outperform WRR_a at P_{dreq}=4% in Fig. 7, whereas it does so at P_{dreq}=2% in Fig. 10. The reason is that content sniffing based load estimation enables CSLD to alleviate dynamic load imbalance among the servers caused by the variation of content.

Every curve in all figures has a valley of "V" shape. Those points located after a valley indicate the normalized average server latencies under peak loads. All strategies showed low rejection rates because the request rate was not high (at most 3,683/s) and further, the duration of peak request rate was relatively short. Another reason is that when a server rejects a connection request, the client model retransmits it at worst two times using an exponential backoff-based timer, and the retransmitted request is likely to be accepted at the server after a transient peak of load. RR and WRR_a rejected new connection requests much more than WRR_i and CSLD. For example, CSLD showed zero rejection rate when C_{dreq}={3} and P_{dreq}=13.6%, whereas WRR_a incurred rejection of 58,059 connection requests. WRR_a performs slightly worse than RR at the point just after each valley because host metrics measurement overhead (5 ms), relatively large compared to the period (500 ms), has a significant impact on the performance under such workloads.

Around the middle of each curve in all figures, the performance improvement ratios (compared to RR) of WRR_i, WRR_a, and CSLD are low. The reason is as follows. Since the simulator uses a random function to generate additional dynamic requests, the resulting load variation is relatively uniform. Given modest workload, RR relatively performs well because currently occurring load imbalance among the servers will be often naturally reduced more or less by subsequent requests that may involve another load imbalance. All workloads in Fig. 7 to 11 have relatively long periods with low request rate. Thus, the "average" server latency is a favorable metric to WRR_a that performs well for light workload, whereas it is a little unfavorable to CSLD that performs better for heavy workload. However, CSLD outperforms or competes with WRR_a.

It seems to be very difficult to develop a practically good load estimation technique for the WRR strategy. Though WRR_i performs best for most workloads, it is infeasible. Despite many ideal assumptions (e.g., ideal load metrics, minimal host metrics measurement overhead, short period, and zero feedback delay, and optimized configuration parameters), WRR_a performs well for not every workload. The simulation results of other variants also showed that the performance is very sensitive to those conditions. In this paper, we attempted to use the WRR strategy for dynamically balancing servers load. We doubt whether this attempt was not appropriate. Authors in [12] proposed WRR as a load-sharing strategy to smooth out transient peak overload periods on some servers rather than as a load-balancing strategy. However, it also requires dynamic load metrics feedback from the servers and tuning up the configuration parameters to the changing workload characteristics. Since it uses a longer period, it less frequently imposes overhead in the servers. If the other conditions were the same, it could perform as well as or slightly better than WRR_a for peak load periods. However, due to a longer period, it is likely to fail to balance load among the servers for the workloads with high variation of content. It is also practically not easy to implement such a sophisticated load-sharing technique. CSLD performed fairly well for all workloads. Every figure showed that it competes with the most idealized WRR_i especially for heavy load. CSLD performed worse than WRR_a only when the overall load is very low. But, this will not cause a significant problem because user perceived latency as well as the server latency are very low in such case. It should be noted that any optimization was not applied to CSLD because we wanted to observe the basic performance of CSLD. Given a little optimization, it achieves better performance for light workload as well.

We did not present the simulation results for the workload with C_{dreq}={1} or {2}. In either case, the performance improvement achieved by WRR_i, WRR_a, and CSLD was at most 21% even if the proportion of dynamic requests and the request rate are very high. Performance behavior of CSLD was similar to those described above. We also tested the scalability. Though WRR_i, WRR_a, and CSLD always scaled better than RR for large cluster sizes, RR also showed relatively good scalability. Given heavy load and small cluster size (N<12), the scalability of WRR_a was not as good as those of WRR_i and CSLD.

All workloads mentioned above include HTTP/1.0 requests only. We repeated the simulations while generating HTTP/1.1 requests in the trace logs. Since the

logs lack related information, the length of a persistent connection was determined solely by the given Apache configuration directives. As a result, the average number of requests delivered on a persistent connection was very large. Such workloads somewhat decreased the effectiveness of WRR_i, WRR_a, and CSLD because the scheduling granularity is per connection. Despite that we optimized WRR_a to adjust the threshold t_diff based on the proportion of HTTP/1.1 requests and the changes in overall load, it showed only a little performance improvement. On the other hand, given such optimization, CSLD achieved more improvement.

4.3 Discussion - Robustness and Overhead of CSLD

CSLD achieved 79-92% of accuracy in cache miss prediction for static requests in the simulations using the World Cup trace that has a relatively small working set due to high locality of reference. Since we wanted to experiment on different traces, we repeated the simulations using the UC Berkeley Web proxy trace [22]. The results were similar, except that the performance of CSLD degraded a little more when both the request rate and the variation of content are very low. This implies that light workload with a large working set may increase the impact of erroneous cache miss prediction on the performance a little but not so critically. Since it is common today to see the servers with large memory, the impact of load caused by static requests on server performance is more and more decreasing.

To mimic WAN delays, a pareto distribution with mean 150 ms was used for RTTs. When the rate of static requests is very high, long RTTs are unfavorable to CSLD that uses a simpler TCP transmission model than that used in the back-end model. Though, CSLD performs at least better than RR for such a workload. Given a shorter mean RTT, it outperforms the others even for light workloads with low content variation.

The standard deviation of dynamic request processing costs (normal distribution) was set to a large value (20% of the mean specified for each class), unfavorable to CSLD, in the previous subsection. Thus, there is a possibility of misestimate of server load because CSLD knows only the mean values. Though, the performance of CSLD was fairly good. We repeated the simulation while increasing the standard deviation. The result showed that the sensitivity of CSLD to such misestimate is not high. Its performance never significantly degraded until the standard deviation is set to 40% of each specified mean, and was still better than that of WRR_a for all workloads but very light one.

CSLD imposes CPU overhead on the front-end L4 switch, mainly for parsing URIs together with file table lookup or class table search. However, our simple measurements showed that it never significantly decreases the routing throughput. It was at most about 200 us even for parsing a dynamic request that involves searching a matching class in a relatively complex class table which has a few tens of entries per parameter-cost classifier. CSLD requires not an exact prediction about when and what amount of load will be caused by each request but an estimate for relative degree of each server's load compared to load of the others. Therefore, there is room for further optimization. We are developing a simpler cache miss prediction technique with low CPU and memory overhead, rather than a more accurate one. In addition, we are studying a technique for hiding the overhead of content sniffing by inter-request times so that it may not be on the critical path of TCP routing. In Section 3, we described that the content sniffer is invoked for each request being forwarded to the target server. Instead, we may consider copying incoming requests to a temporary buffer so that they can be parsed and classified in a periodic batch-process.

5. Conclusion

We present a new content sniffer based load distribution (CSLD) strategy for L4 switch. A simulation study shows that the performance advantages of CSLD over WRR increase as the variation of content or the overall load increases, and CSLD performs at least better than RR for any workload. Performance results suggest that load balancing strategies should take into account dynamic load imbalance among the servers caused by the variation of content in order to improve the Web server cluster performance. Our proposed content sniffing based load estimation technique enables not only timely and proper reaction to dynamic load imbalance but also easy implementation, compared to that of WRR.

We developed a prototype CSLD as a loadable kernel module and integrated it into a software L4 switch called the Linux Virtual Server [24]. We are currently optimizing it to improve the performance and to reduce the imposed overhead. Some future work is necessary for evaluating its performance in a real web server environment. To simulate WAN delays, we are attempting to add artificial delays to the packet routing mechanism in the IP router. For a fair comparison, we should implement a sophisticated load estimation technique for the WRR strategy. We need a web server cluster benchmark which can not only generate a realistic request traffic but also support dynamic content.

References

[1] D. Andersen, T. Yang, and O. Ibarra, "Toward a scalable distributed WWW server on workstation clusters," J. Parallel and Distributed Computing, vol.42, no.1, pp.91–100, 1997.
[2] Apache Http Server Project, http://httpd.apache.org.
[3] G. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha, "Design, implementation and performance of a content-based switch," Proc. IEEE INFOCOM 2000, pp.1117–1126, 2000.

[4] M. Arlitt and T. Jin, "Workload characterization of the 1998 world cup web site," IEEE Network, vol.14, no.3, pp.30–37, 2000.
[5] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, "Scalable content-aware request distribution in cluster-based network servers," Proc. USENIX Annual Technical Conference, pp.323–336, 2000.
[6] V. Cardellini, M. Colajanni, and P. Yu, "Dynamic load balancing on web-server systems," IEEE Internet Computing, vol.3, no.3, pp.28–39, 1999.
[7] E. Casalicchio, V. Cardellini, and M. Colajanni, "Content-aware dispatching algorithms for cluster based web servers," Cluster Computing, vol.5, no.1, pp.65–74, 2002.
[8] Cisco Systems Inc., LocalDirector, http://www.cisco.com.
[9] Foundry Networks Inc., ServerIron, http://www.foundrynet.com.
[10] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, "Cluster-based scalable network services," Proc. 16th ACM Symposium on Operating System Principles, pp.78–91, 1997.
[11] Y. Hu, A. Nanda, and Q. Yang, "Measurement, analysis and performance improvement of the apache web server," Proc. 18th International Performance, Computing and Communications Conference, 1999.
[12] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee, "Network dispatcher: A connection router for scalable internet services," Computer Networks and ISDN Systems, vol.30, no.1-7, pp.347–357, 1998.
[13] A. Iyengar, E. MacNair, and T. Nguyen, "An analysis of web server performance," Proc. IEEE GLOBECOM, 1997.
[14] A. Iyengar, M. Squillante, and L. Zhang, "Analysis and characterization of large-scale web server access patterns and performance," World Wide Web Journal, vol.2, no.1, pp.85–100, 1999.
[15] E.D. Katz, M. Butler, and R. McGrath, "A scalable HTTP server: The NCSA prototype," Computer Networks and ISDN Systems, vol.27, no.2, pp.155–164, 1994.
[16] B. Krishnamurthy and J. Rexford, Web protocols and practice: HTTP/1.1, networking protocols, caching, and traffic measurement, 1st ed., Addison Wesley, 2001.
[17] D. Maltz and P. Bhagwat, "TCP splicing for application layer proxy performance," IBM TR RC-21139, 1998.
[18] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-aware request distribution in cluster-based network servers," Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.205–216, 1998.
[19] J. Redstone, S. Eggers, and H. Levy, "An analysis of operating system behavior on a simultaneous multithreaded architecture," Proc. 9th ASPLOS, pp.245–256, 2000.
[20] Resonate Inc., Central Dispatch, http://www.resonate.com.
[21] J. Song, E. Levy, A. Iyengar, and D. Dias, "Design alternatives for scalable web server accelerators," Proc. IEEE International Symposium on Performance Analysis of Systems and Software, pp.184–192, 2000.
[22] The Internet Traffic Archive, http://ita.ee.lbl.gov.
[23] C. Yang and M Luo, "Efficient support for content-based routing in web server clusters," Proc. USENIX Symposium on Internet Technologies and Systems, pp.221–232, 1999.
[24] W. Zhang, "Linux virtual servers for scalable network services," Proc. Ottawa Linux Symposium, 2000.
[25] H. Zhu and T. Yang, "Cachuma: Class-based cache management for dynamic web content," Proc. IEEE INFOCOM 2001, pp.1215–1224, 2001.

Jongwoong Hyun received the B.S. degree from Korea University, in 1986 and the M.S. degree in Computer Science from Korea Advanced Institute of Science and Technology (KAIST), in 1998. He is currently a Ph.D. student working in the computer architecture group at KAIST. From 1986 to 1998, he was with Samsung Electronics Co. Ltd., Korea. His research interests include computer architectures, cluster computing, and web server.

Inbum Jung received the B.S. degree from Korea University, in 1985 and the M.S. and Ph.D. degrees in Computer Science from KAIST, in 1994 and 2000, respectively. From 1984 to 1995, he was with Samsung Electronics Co. Ltd., Korea. He is currently a faculty member at Kangwon National University. His research interests include operating systems, computer architectures, parallel processing, and cluster computing.

Joonwon Lee received the B.S. degree from Seoul National University, in 1983 and the M.S. and Ph.D. degrees from the College of Computing, Georgia Institute of Technology, in 1990 and 1991, respectively. From 1991 to 1992, he was with IBM research centers. He is currently a faculty member at KAIST. His research interests include computer architectures, operating systems, parallel processing, and grid & mobile computing.

Seungryoul Maeng received the B.S. degree in Electronic Engineering at Seoul National University in 1977 and the M.S. and Ph.D. degrees in Computer Science from KAIST, in 1979 and 1984, respectively. He has been a faculty member at KAIST since 1984. His research interests include computer architectures, parallel systems, cluster computing, and grid computing.