Web Load Balancing Through More Accurate Server Report

C. Edward Chow, Dept. of CS, Univ. of Colorado at Colorado Springs, Colorado Springs, CO 80933-7150, USA, chow@cs.uccs.edu
Indira Semwal, Dept. of CS, Univ. of Colorado at Colorado Springs, Colorado Springs, CO 80933-7150, USA, isemwal@cs.uccs.edu

Abstract

In this paper we present the load balancing results of a web server clusters where the server load reporting of Apache web servers are improved by accurately computing the remaining document size to be transmitted and dynamically estimating document retrieving and transmission speed. The performance of the web server cluster is significantly improved over round robin scheduling. We also consider the impact of network path information in selecting the servers in a WAN cluster. The architecture and an experimental testbed of such load balancing system is presented.

Keywords: Internet Computing, Cluster Computing, Load Sharing and Balancing, Apache Web Server

1. Introduction

It is often necessary to replicate databases and network services available over the Internet in order to serve a large number of clients more efficiently [1, 2, 3]. Additionally when servers are replicated over several geographic locations on the Internet, they reduce the congestion on backbone and international links and provide faster access to the users. Cluster systems, web switches, and bandwidth control devices were proposed and developed for such environment.

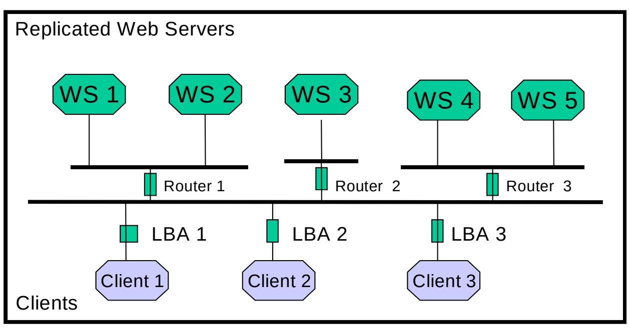


Figure 1.1. Load Balancing System

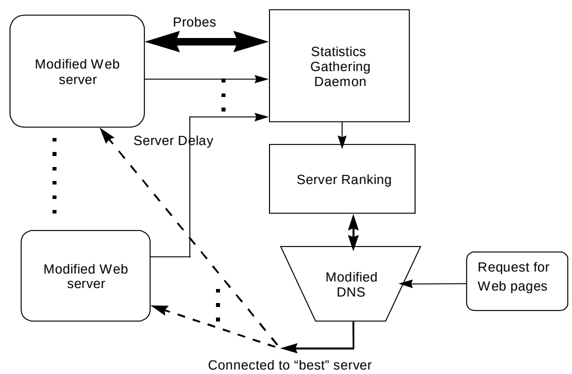


Figure 1.2. Load Balancing Agent

Whenever the replicated database is available, the user needs to determine which of the replicated servers will provide a "better" service. One could make the choice based on the geographic proximity of the server or simply choose the first choice offered. However, this method is static in nature and may not necessarily provide the user with the best service. The load over the networks and at any given server can change dynamically and so it would be advantageous to have the server selection be based on the dynamic nature of the network and the server load.

In this paper we have studied the factors that affect the quality of connection or the response time of a remote server. In our study the choice of server takes into account the traffic along the network connection to each of the replicated servers, as well as the load at each of the servers.

The algorithm developed in this paper provides the user with the IP address of the server that has the least response time. This selection is made from several statistical data collected by a Load Balancing Agent (LBA). The frequency with which the statistics about the network are collected can also be adjusted. This frequency could depend upon the required accuracy of the result, as well as some other factors, e.g., the time of day. For instance, during off-peak hours, the networks are not as dynamic, so the statistics need not be collected as often. During high traffic period, some statistical data could be collected to make better decisions regarding the server. The Load balancing agent (LBA) collects three kinds of data:

- The bottleneck bandwidth along a given path.
- The available bandwidth along a path.
- The expected delay at the web server, based on the size of the pending queue at the web server and its processing power.

The LBA collects network path statistics using two probing tools Bprobe and Cprobe [3]. The delay at the server is measured by the server itself. This was accomplished by modifying the Apache Web Server source code [6]. Once the statistics are available, the LBA calculates the ranking of each replicated server.

Normally when a user tries to access services of a remote host, a Domain Name System (DNS) server [5], provides a mapping that converts host names into IP addresses. We used a modified load balancing DNS [8]. A cluster of servers is associated with a logical domain name, e.g. best.csnet.uccs.edu. Our modified DNS reads the file created by the LBA with the server names and their ranks, it then selects the highest ranked server and forms a query response to the clients. Thus the DNS returns the host IP address of the server that is ranked the best. This work has been explained in [7, 8].

The combination of our LBA and the modified DNS connects the user with the best-ranked server. It automates the dynamic selection process for the client.

1.1. RELATED RESEARCH

In the past, many static schemes for server selection have been developed. Schemes for static server selection mainly use the geographical location of the client and server or the distance measured using network hops. More recently, dynamic algorithms were developed [1-4]. A survey and performance analysis of load balancing algorithms is in [9]. Some dynamic algorithms use the Round-Robin scheme to rotate through a set of identically configured servers [1]. In other dynamic server selection methods, the dynamic nature of the network is taken into account. [2, 3]. This means that the client must have knowledge about the traffic on the path from the client to each server before a selection is made. As the load on servers also varies, the load factor must also be taken into account when making a selection [4].

In [3], bandwidth-probing tools were developed, to dynamically select the best server[1]. This work is a major improvement to the static server selection methods and their experiments improved the selection process by 50%. A valuable contribution is that they developed path-probing tools that measure path characteristics very accurately. These tools BPROBE and CPROBE were ported in another project, for use in our labs and have been used in our experiments. They are included as a part of the Load Balancing Agent in our load balancing implementation.

A drawback of the implementation in [3] is that it does not take into account the server load. The criterion in [3] for server selection is based only on the bandwidth along the given path and the congestion.

2. APACHE WEB SERVER WITH ACCURATE SERVER LOAD REPORTING

The Apache web server [6] provides a server status web page, which reports the history of CPU load, server throughput, and connection rate. A more accurate server load will be measured by calculating the pending queue size of requests. The queue size divided by the server processing power will give a more accurate delay estimation of the server. The Apache web server was modified so that the current status of the server load can be calculated more accurately, and make it dynamically available to the load balancing agents. This Section gives details about the Apache web server code that was modified for the load balancing implementation.

2.1. Enhancements made to the Apache 1.3.9 source code .

We begin this section with a brief summary of how the Apache Web server processes incoming requests. Then we explain the role of the status handler in reporting the server-status. Following this, the changes that were made in the status handler in order to obtain extensive status reporting will be described. Finally the procedure for communicating the status of the server to the LBA, or more precisely, the expected delay at the server will be described.

2.2. Request Processing by Apache.

Apache preforks several child processes that listen to incoming requests. When a child process receives a request, it does the following:

- URI to filename translation,
- Authorized ID checking,

---

1 These tools BPROBE and CPROBE were used in our LBA implementation.

- Authorization access checking,
- Access checking other than authorization,
- Determining MIME type of object requested,
- Check hook for possible extensions,
- Actually sending a response back to the client, and
- Logging the request.

The phases are handled by a succession of modules, each of the modules invokes a handler for the phase. The sole argument to the handlers is the request_rec structure. Handlers for most phases set the respective fields of this data structure. The response handler sends an actual response to the client. The primitives for this are ap_rputc and ap_rprintf, for internally generated output, and ap_send_fb_length, ap_send_fd_length, and ap_send_mmap to copy the contents of a file straight to the client. Modifications had to be made to functions ap_send_fb_length, ap_send_fd_length, and ap_send_mmap.

Each child process keeps track of its status by updating the fields of the child scoreboard, also called the short_score. The parent process keeps track of its children by reading the child's scoreboard and updating its own scoreboard fields in the parent score. As mentioned earlier, this is implemented using shared memory. The short_score, the parent_score and a global_score are tied up in the scoreboard structure.

```
Typedef struct {
      short score servers[HARD_SERVER_LIMIT];
      parent_score parent[HARD_SERVER_LIMIT ];
      global_score global;
      } scoreboard ;
```

Code Listing 2.1

The data structures for short_score, parent_score and global_score are in the file apache/apache_1.3.9/src/include/scoreboard.h.

We needed to ensure that each child process keeps account of its dynamic queue size. The queue size at each child process can be determined by the number of bytes remaining to be transferred by this child.

bytes remaining = file size - bytes already transferred

The Apache code for file transfer was modified. File transfers are made by either calling ap_send_fd_length(), or ap_send_fb_length(), or ap_send_mmap().

These functions are in apache/apache_1.3.9/src/main/http_protocol.c. The partial code segments are shown below:

```
API_EXPORT(long) ap_send_fd_length(FILE *f, request_rec *r, long length) {
    char buf[IOBUFSIZE];
    long total_bytes_sent = 0;
    register int n, w, o, len;
    int mychild_num;
1   short_score *ss;
    if (length == 0) return 0;
2   ap_sync_scoreboard_image();
3   mychild_num = r->connection->child_num ;
4   ss = &ap_scoreboard_image->servers[mychild_num];
      ..........
    while (n && !r->connection->aborted) {
       w = ap_bwrite(r->connection->client, &buf[o], n);
       if (w > 0) {
          ap_reset_timeout(r); /* reset timeout after successful write */
          total_bytes_sent += w;
          n -= w;
          o += w;
5         ss->mybytes_sent = total_bytes_sent;
6         ss->mybytes_remaining = r->finfo.st_size -  total_bytes_sent ;
7         put_scoreboard_info(mychild_num, ss);
```

```
            } else if (w < 0) {
               if (!r->connection->aborted) {
                  ap_log_rerror(APLOG_MARK, APLOG_INFO, r,
                      "client stopped connection before send body completed");
                  ap_bsetflag(r->connection->client, B_EOUT, 1);
                  r->connection->aborted = 1;
               }
               break;
            }
         }
    }
    ap_kill_timeout(r);
8   put_scoreboard_info(mychild_num, ss);
    SET_BYTES_SENT(r);
    SET_BYTES_SENT(r);
    return total_bytes_sent;
}
```

Code Listing 2.2

In the above code segment, line 1, creates an instance of the scoreboard into which the child can write. Line 2, ap_sync_scoreboard_image() will read the current scoreboard which contains information about all child processes. When a child is given a request to process, a request_rec data structure is passed to it. The child can find out all the details of the request from the fields of this data structure, as well as its own child number, mychild_num = r->connection-> child_num. The child can access its scoreboard once it knows its child_num. Lines 3 and 4 were added to find the child_num and to access the scoreboard for this child. When this write is successful, the next IOBUFSIZE bytes are read and so on. Normally the function returns when the entire file is written out.

As IOBUFSIZE bytes are transmitted, a running score of the bytes sent is saved in the mybytes_sent field of the short_score, line 5. Similarly a running score of the bytes remaining is saved in the mybytes_remaining field of the short score, line 6. Line 7, updates the scoreboard. Line 8 updates the scoreboard again when the file transmission is complete. Similar changes were made to ap_send_fb_length() and ap_send_mmap ().

2.3. The Role of the status_handler for Status Reports.

The status_handler module reads the scoreboard and outputs the server status in html format. The scoreboard fields are in shared memory and can be accessed from anywhere in the program. Some of the variables reported in the status report are:

Total access = count

where count is the sum of requests made to all the children

Total traffic = kbcount,

where kbcount is the sum of bytes served by each child.

Bytes served = kbcount

Bytes/sec = kbcount / uptime,

where uptime is the difference in the current time and time when the server started.

Bytes per request = kbcount /count

*CPU load = (tu+ ts +tcu+ tcs) /tick * uptime..Eq. 2.1,*

where, tu : user time in seconds; ts: system time in seconds;

tcu: user time in microseconds; tcs: system time in microseconds

Request/sec = count/uptime

In order to report statistics on the web server, the status handler loops through all the child processes that are busy and calculates the above mentioned parameters, based on the scoreboard values. It then outputs these parameters in the html format.

2.4. Enhancements made for Extensive Status reporting.

One of the main goals in our project was to dynamically obtain the status of each child. What we observed was that the bytes_left field, before our modifications, was either always equal to the size of the file or it was 0. This prompted us to check when Apache was updating the bytes_sent field. Evidently, the scoreboard is normally updated only when an entire file is written to the output buffer.

In order to dynamically obtain the "number of bytes remaining" on each request, additional fields were added to the short_score, in apache/apache_1.3.9 /src/include/scoreboard.h. Additional fields needed were bytes_left and time_to_completion. As explained in the previous section, the bytes_left field of the child score were updated by the child during file transfer, the modified file is src/main/http_protocol.c. The modified status handler mod_status.c in file src/modules/standard/mod_status.c, then reads the bytes_left value from the scoreboard and computes the time_to_completion value.

As the scoreboard is updated with each write into the output buffer, any time the scoreboard is read by the status handler, it gets the most recent value of bytes_left. To write out the values into the scoreboard the put_scoreboard_info() function is called. This function is declared as a static function in /src/main/http_main.c . To allow this function to be called by the functions ap_send_fd_length() and ap_send_mmap(), both of which are in /src/main/http-protocal.c, the declaration of the put_short_score function had to be redefined to extend its scope for all Apache files.

Now that the status handler can read the most current value of the bytes_left, a small code segment was included in the status_handler function to print out the bytes_left for each child process. This segment also calculates the running sum of bytes_left for all the processes and outputs that.

Total remaining bytes , $B = \sum B_i$*...Eq. 2.2,*

where, B_i are the number of bytes remaining on each pending request.

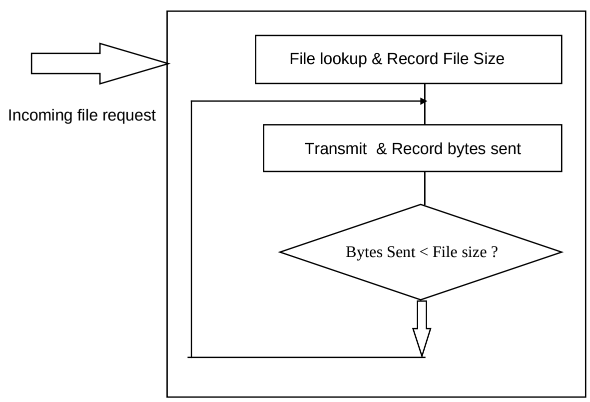


Figure 2.2 Dynamic Queue Size

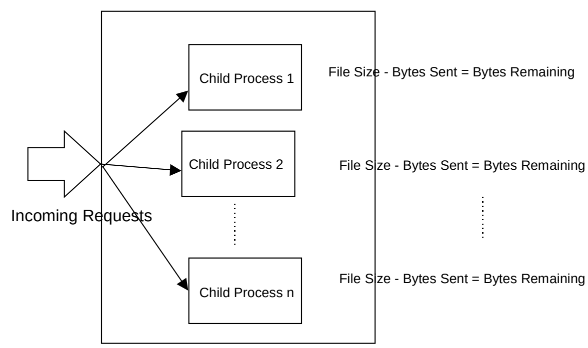


Figure 2.3 Modified Status Reporting

Additionally we were interested in the delay that will be encountered at the server at any given time. To calculate this we need to also know the speed at which the server is processing the requests. This is determined by looking at the size of file transferred for each request and the time taken to process the request. Each child process keeps track of its "time taken" values from its scoreboard.

Child Speed = $(\Sigma \text{Bytes sent} / \Sigma \text{time taken}) / N$*..Eq. 2.3,*

where N is the number of children with pending requests and Bytes sent is the number of bytes sent for the i-th request.

The Average speed is calculated using the speed for each child process and dividing it by the number of "alive" child processes:

Average Speed = ΣChild Speed/ Number of alive children processes....Eq. 2.4

We use the following formula to compute the time remaining for the server:

Time Left = B / Average Speed . ...Eq. 2.5

2.5. THE LOAD BALANCING AGENT

The main task of the LBA is to dynamically collect the statistics for all the web servers that belong to the cluster of replicated servers. In our implementation the web servers need not be on the same subnet. They can be distributed all over the Internet. Testing of the LBA was done for servers within a UCCS subnet. The statistics being collected by the LBA are:

- Network bandwidth for each replicated web server
- Network Congestion Bandwidth
- Expected Delay at the server

The available and the bottleneck network bandwidth are estimated using our modified version of BPROBE® and CPROBE®, bandwidth estimation tools originally developed by Carter and Crovella [3]. As we started our experiments, there were a few options that we considered for reading the bandwidth information that was being generated by the probes. The LBA launches the probes using the system command as given below:

system(comm);

where "comm" is the command to launch bprobe i.e.

comm = "./bprobe hostname or "comm = "./cprobe hostname

This function system (), invokes the Bourne shell to execute the specified command *string*.

Once the probing is launched, it provides analysis of intermediate results. At the end it includes the number for the estimated final bandwidth. One option to read the final bandwidth value was to write the output to a file and the parse it. This option was not implemented, as the output file could be quite large and parsing would slow down the probing process.

The second option was to modify the source code of the probes and use sockets to send the final bandwidth information to the load balancing agent. This option has the drawback that we do not know in advance the port numbers of the load balancing agent to send the bandwidth information, and this parameter would have to be entered, once the programs were running and the port numbers were assigned. Then the LBA and the probe tools would advertise their port numbers and the user would type the port number of the LBA so that the probe tool would scan it and send the bandwidth information to the LBA.

There is a simpler solution to the above problem that we implemented. We made a minor modification to the probe tool code at bprobe.c and congtool.c, so that the probing tools, apart from doing a screen dump, would write out the final bandwidth values to a file along with the name of the probed host. The LBA then reads the hostnames and bandwidth values from the files, one file for bprobe and the other for cprobe, and saves the values in its internal data structure.

The delay value at a web server is calculated by the modified Apache source code as explained in Section3 of this paper.

The weight for each server is calculated using the following formula

*weight = (1+ ms->b_Result + ms->c_Result)/((1 + ms->delay) * 10000)*

.............*Eq. 2.6*

Our main reason for coming up with this formula was to consider that there is a direct relationship between higher weight and greater bandwidth and an inverse relationship between delay and weight, i.e. higher delay means lower weight. Weight calculation has also been discussed in [7,8].

3. WEBSTONE BENCHMARKING RESULTS

In this Section we discuss the testbed set up for measuring the performance of the LBA system and the performance results. Hardware configuration for web server:

4 Pentium 500 MHz PCs (128MB Memory, 256 MB Swap, 3Com 3c905B Ethernet Interface)

LAN: 100Mbps

Operating system: Redhat 6.2

Web Servers: Apache 1.3.9

Hardware configuration for the Load Balancing Agent and the Load balancing DNS:

1 Pentium 240 MHz PC (128MB Memory, 128 MB Swap, IDE)

Network: AMD PLNET 32 10 Mbps

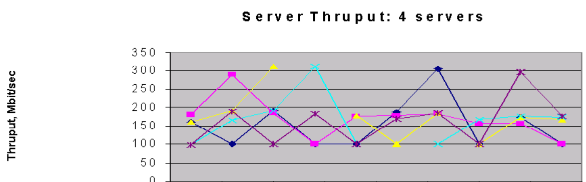
LAN: 10 Mbps Ethernet

Operating System: Redhat 7.0

Benchmarking was done using a modified Webstone Benchmarking tool WebStone 2.5 [10]. Modifications were made to the source code of WebStone. These modifications take into consideration that the IP address of the web server could be dynamic. Each web client in the modified WebStone resolves the web server IP address dynamically. We used the modified code to further test the load balancing DNS. Benchmarking results are in the following graphs.

The LBA was tested with sets of web servers ranging in number from one to six. With each web server configuration, the load was increased, by increasing the number of clients requesting web pages. The numbers of clients were increased from 10 to 100. Tests were also run to monitor the performance of the LBA depending on the frequency with which we update the information about server load. The frequency for updating server load information was increased from one second, then to five, ten, fifteen and finally at 20 seconds.

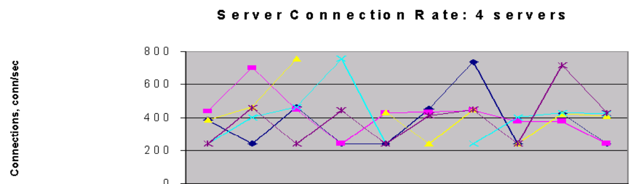
We observed that when the number of servers in our replicated configuration is increased from one server to six servers, the throughput increases. The system throughput and connection rate fluctuate more, as the number of clients that probe the system with web page requests, is increased from 10 clients to 100 clients, as shown in Figures 3.1 and 3.2. The system as a whole performed better, when the LBA updated server load information every 5 seconds. This could be attributed to the oscillating effect that can occur when the best performing web server is selected, and a large number of incoming requests are sent to this server. The performance of the server degrades, and the next set of requests are routed to other servers, thus the performance of this server farm as a whole falls, but it then rises as the load is balanced.



	1	2	4	5	6	7	9	10
update: 1 s	159.8	99.58	99.23	99.36	186.5	304.5	174.6	99.45
update: 5 s	179.6	289	99.49	176.6	177.6	182.1	155.5	99.33
update: 10 s	160.4	191.2		178	99.42	184.4	173.3	168
update: 15 s	98.52	165.7	311.8	99.45		99.27	176.8	173.8
update: 20 s	98.89	189.4	183	99.43	169.5	185.5	295.4	177.1

Clients (in unit of 10)

Figure 3.1 Server throughput vs. # of clients with different server load reporting period.



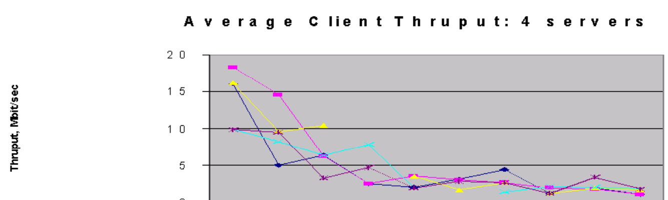
	10	20	40	50	60	70	90	100
update: 1 s	388.1	242	241.1	241.4	453.1	739.7	424.3	241.6
update: 5 s	436.4	702.2	241.7	429	431.5	442.5	377.9	241.3
update: 10 s	389.8	464.5		432.6	241.5	448	421.1	408.3
update:15 s	239.4	402.6	757.5	241.6		241.2	429.6	422.3
update: 20 s	240.3	460	444.7	241.6	411.8	450.6	717.7	430.2

Clients

Figure 3.2 Server connection rate vs. # of clients with different server load reporting period.

As we increased the number of clients probing the server, for any given configuration of web servers, we noticed that the average client throughput fell more rapidly, Fig. 3.3, with an increase in the number of clients, when there were fewer web servers configured in the system. For example, with two web servers configured, the average client throughput fell rapidly when the number of clients increased from 10 clients to 20 clients. With five web servers configured, the average client throughput fell rapidly when the number of clients was increased from 20 clients to 30 clients. Again, a higher throughput threshold was maintained when the LBA updated server performance statistics every 5 seconds.

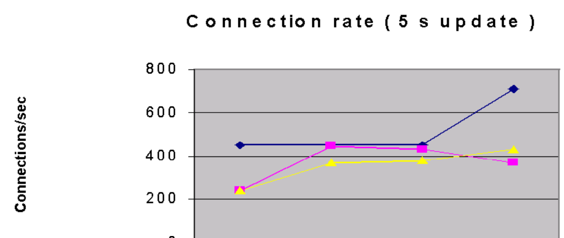
The connection rate in our experiments increased with the addition of web servers to the system configuration. We also observed that even though the connection rate increased with the addition of web servers, the connection rate fluctuated more when we increase the number of clients requesting web pages. The most stable performance was noted when the LBA updated server load information every 5 seconds.



	10	20	40	50	70	80	100
update:1s	16.08	4.99	2.5	2.02	4.44	1.26	1.03
update:5s	18.28	14.51	2.49	3.58	2.65	2	1.03
update:10s	16.15	9.68		3.61	2.69	1.29	1.72
update:15s	9.88	8.32	7.83	2	1.46	2.13	1.81
update:20s	9.92	9.58	4.7	2	2.69	1.28	1.8

Client

Figure 3.3 Average client throughput vs. # of clients with different server load reporting period.



	1	2	3	4
30 clients	451.56	447.42	448.93	706.03
50 clients	237.89	445.4	429.01	364.99
90 clients	238.75	367.97	377.85	430

Servers

Figure 3.4. Connection rate vs. the number of servers.

Figure 3.4 shows a steady connection rate increases in general as more servers are added, except that with 50 clients the connection rates actually decreases. This may due to transient network traffic.

For a round-robin configuration with four servers, the connection rate was 327.6. This number is used as a based line in Figure 3.5 to compare the performance of LBA with different number of server load updates, ranging from 1 to 12 per second. The experiments show the average connection rate for the 4-server configuration with the LBA was 615 connections per second. The connection rate fluctuates when the number of the server load updates is between 1 and 8, after 8 we saw consistent increase in performance. Further studies are needed to explain the phenomena.

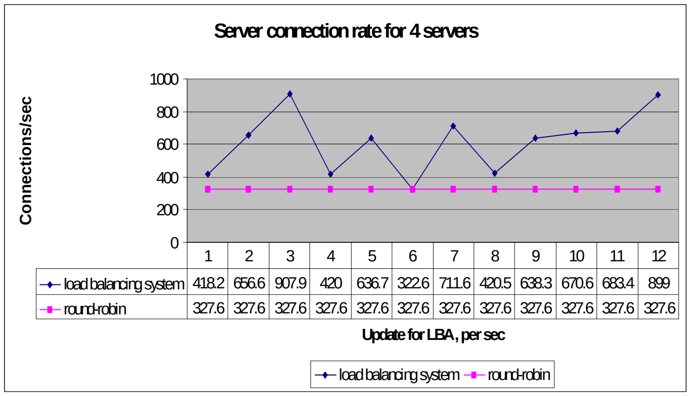


Figure 3.5. Compare round-robin with LBA using accurate server load.

4. CONCLUSIONS AND FUTURE WORK

A load balancing system was implemented for improving web server farm performance by collecting statistics about web server load and path characteristics, and distributing the load accordingly. An analysis is made about the performance of the load-balancing agent. The WebStone benchmark results on the server cluster based on the LBA shows a consistent improvement as the number of web servers increases. The results also show an improvement over the round-robin method. The impact of the server load update frequency on the cluster was also studied.

The preliminary experiments indicate that there is a peak in performance near the 5-second update interval. Our conclusion after studying the results was that web server performance improved as additional servers were added to the configuration. However, we saw degradation in performance when the available memory on the local disk was low. Server throughput increased with the addition of more servers, and was evaluated to be most stable when the LBA was updating server information every 5 to 10 seconds.

Our load balancing agent has been designed and implemented with limited functionality. It can be extended by adding a configuration protocol that automatically update the server list when a server joins or leaves a cluster. This work can also be enhanced to a system with several LBA's that serve in a WAN environment. In such an extended system, the LBA's could exchange information with web servers as well as with each other. Multicasting could be used for simplicity and effectiveness. The LBA's could be designed to collect server load and path information for servers within close geographic proximity. Not only would this give an accurate picture of the load at a web site, it would reduce the congestion that probing creates over the backbone links. However, due to the relative long delay in WAN environment, further studies are needed to investigate the impact of delay. Also the child process speed can be estimated at initial set up time by retrieving certain document size and avoid impact of the connection bandwidth on its estimation.

References

[1] E.D. Katz et al. "A scalable HTTP server: The NCSA prototype". Computer Networks, vol. 27(2) pp. 155-164, 1994.
[2] Zegura E. W., et. al. "Application-layer anycasting: a server selection architecture and use in a replicated Web service," ACM Trans. Networking, vol. 8(4), pp. 455 – 466, Aug. 2000.
[3] Carter R. L and Crovella M, "Measuring Bottleneck Link speed in Packet-Switched Networks". Tech. Rep. BU-CS-96-006, Computer science Department, Boston University, Boston, MA, 1996.
[4] Dahlin M. "Interpreting Stale Load information". UTCS Technical report TR98-20, Department of Computer science, University of Texas at Austin, 1998.
[5] Albitz P. and Liu C. DNS and BIND. O'Reilly & Assoc. 1997.
[6] Apache Web Server. http://www.apache.org.
[7] Emery S. M, " Dynamic Load Balancing of Virtual Web Servers". Thesis report. Department of Computer Science, CU-Colorado Springs, CO. March 2000.
[8] Schemers R. J III. "lbnamed: A Load balancing Name Server in Perl". Sun Soft Inc. LISA IX Sept. 1995.
[9] Blais A. L. and Chow C. E. Performance Analysis of Load Balancing Algorithms. Technical Report EAS-CS-2000-2.
[10] Webstone 2.5 Benchmarking tool. http://www.mindcraft.com/webstone.