Towards a Scalable Distributed WWW Server on Workstation Clusters

Daniel Andresen Tao Yang Oscar H. Ibarra
Department of Computer Science
University of California
Santa Barbara, CA 93106
{dandrese, tyang, ibarra}@cs.ucsb.edu

Abstract

In this paper, we investigate the issues involved in developing a scalable World Wide Web (WWW) server called SWEB on a cluster of workstations. The objective is to strengthen the processing capabilities of such a server in order to match huge demands in simultaneous access requests from the Internet, especially when these requests involve delivery of large digitized documents. The scheduling component of the system actively monitors the usages of CPU, disk I/O channels and the interconnection network to effectively distribute HTTP requests across processing units to exploit task and I/O parallelism. We analyze the maximum number of requests that can be handled by the system and present several experiments to examine the performance of this system.

1 Motivation

The Scalable Web server (SWEB) project grew out of the needs of the Alexandria Digital Library (ADL) project at UCSB [A96]. Digital library systems, which provide the on-line retrieval and processing of digitized documents through Internet, have increasingly turned into a topic of national importance. Our work is motivated by the fact that the Alexandria digital library WWW server has a potential to become the bottleneck in delivering large digitized documents over high-speed Internet connections. Popular WWW sites such as the *Lycos* and *AltaVista* [LYC95] receive three to ten million accesses a day.

For WWW-based network information systems such as digital libraries, the servers involve much more intensive I/O and heterogeneous CPU activities than simple file servers.. To meet such demands, the SWEB project makes use of networked computers and inexpensive disk resources to strengthen the processing and storage capabilities of WWW servers. We expect that using the idle cycles of those processing units and retrieving files in parallel from inexpensive disks can significantly improve the scalability of the server in response to a large amount of simultaneous HTTP requests.

Numerous other initiatives to create high-performance HTTP servers have been reported. The *Inktomi* server at UC Berkeley is based on the NOW technology [BR96]. NCSA [KBM94] has built a multi-workstation HTTP server based on round-robin domain name resolution (DNS) to assign requests to workstations. The round-robin technique is effective when HTTP requests access HTML information of relatively uniform size and the load and computing powers of workstations are relatively comparable. Our assumption is that the computing powers of workstations and parallel machine resources can be heterogeneous. They can be used for other computing needs, and can leave and join the system resource pool at any time. Thus scheduling techniques which are adaptive to the dynamic change of system load and configuration are desirable. Round-robin DNS assignment cannot predict those changes. Another weakness of the technique is the degree of *name caching* which occurs. DNS caching enables a local DNS system to cache the name-to-IP address mapping, so that most recently accessed hosts can quickly be mapped. The downside is that all requests for a period of time from a DNS server's domain will go to a particular IP address [MFM95].

It should be noted that WWW applications only represent a special class of Internet information systems. There are other information servers dealing with huge file sizes and large numbers of users, for example multi-media servers [HL95+]. Our situation has several differences. First, our current system has no real-time processing constraints for displaying digital movies or audio clips. Secondly, many of our users tend to browse different text and images information, rather than focusing on one particular document such as one film beginning to end. Thirdly, we require lossless document delivery, keeping consistency between the source library holding and what is sent to the client.

We have implemented our scheme on a cluster of SUN SPARC nodes connected by Meiko CS-2 Elan network, and clusters of SUN and DEC workstations connected by Ethernet. Each processing unit (i.e. a SUN SPARC node) is linked to a local disk and may be capable of handling a user request. The distinguishing feature of SWEB is effective resource utilization by close collaboration of multiple processing units. Server scalability is achieved by actively monitoring the run-time CPU, disk I/O, and network loads of system resource units, and dynamically scheduling user HTTP requests to a proper node for the minimum response time.

Our dynamic scheduling scheme is closely related to the previous work on load balancing on distributed systems, for which a collection of papers is available in [SHK95]. In these studies, tasks arrivals may temporarily be uneven among processors and the goal of load balancing is to adjust the imbalance between processors by appropriately transferring tasks from overloaded processors to underloaded processors. The task resource requirements are unknown, and the criteria for task migration are based on a single system parameter, i.e., the CPU load. We call them single-faceted scheduling strategies. In WWW applications, there are multiple parameters that affect the system performance, including CPU loads, interconnection network performance and disk channel usages. The optimal HTTP request assignment to processors does not solely depend on CPU loads. Thus we need to develop a multi-faceted scheduling scheme that can effectively utilize the system resources by considering the aggregate impact of multiple parameters on system performance. In [GDI93], multiple resource requirements are predicted and suggested to guide the load sharing, but the authors utilize only the CPU factor in predicting response times in their analysis.

The paper is organized as follows: Section 2 gives the background and problem definition. Section 3 discusses the possible approaches for building a scalable WEB server and the SWEB organization and load balancing strategies. Section 3.3 provides analytic results on a performance bound of SWEB. Section 4 presents the experimental results showing the performance of SWEB and compares our approach with others. Section 5 discusses conclusions and future work.

2 Background: The World Wide Web

The World Wide Web is based on three critical components: the Uniform Resource Locator (URL), the HyperText Markup Language (HTML), and the HyperText Transfer Protocol (HTTP). The URL defines which resource the user wishes to access, the HTML language allows the information to be presented in a platform-independent but still well-formatted manner, and the HTTP protocol is the application-level mechanism for achieving the transfer of information [HT95].

A simple HTTP request would typically activate a sequence of events from initiation to completion as shown in Figure 1. First, the client determines the host name from the URL, and uses the local Domain Name System (DNS) server to determine its IP address. The local DNS may not know the IP address of the destination, and may need to contact the DNS system on the destination side to complete the resolution. After receiving the IP address, the client then sets up a TCP/IP connection to a well-known port on the server where the HTTP process is listening. The request is then passed in through the connection. After parsing the request, the server sends back a numeric response code (e.g., 202 in the HTTP protocol stands for "OK. File found.", and 404 is "File not found.") followed by the results of the query. The connection is then closed by either the client or the server.

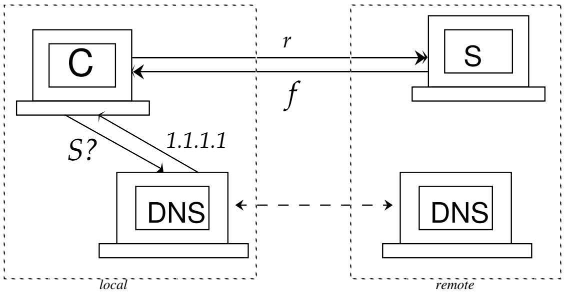

Figure 1: *A simple HTTP transaction.* Client C looks up the IP address of server S, sends over request r, and receives response f.

3 SWEB: A scalable WWW server

We call a system *scalable* if the system response time for individual requests is kept as small as theoretically possible when the number of simultaneous HTTP requests increases, while maintaining a low request drop rate and achieving a high peak request rate. We define the response time for a request as the length of the time period from when a request is initiated until all the requested information arrives at the client.

For a single-workstation server, there is an upper bound for the number of requests per second (RPS) that the server can handle. For example, the NCSA has performed a number of tests using high-end workstations, and discovered in their working environment approximately 5-10 RPS could be dealt with using the NCSA httpd server [KBM94], which cannot match the current and future loads (e.g. a digital library server). Thus multiple servers are needed for achieving scalable performance.

Our overall objective is to reduce and sustain the response time under large numbers of simultaneous requests. Goals considered in designing this system are twofold. First, we hope to demonstrate how to utilize existing inexpensive commodity networks, heterogeneous workstations, and disks to build a scalable WWW server. Second, we attempt to develop dynamic scheduling algorithms for exploiting task and I/O parallelism, adaptive to changing system resource loads and availabilities. The scheduler needs to incorporate multiple system performance parameters to assign user HTTP requests to a proper unit for efficient processing, while adding minimal overhead.

There are several performance factors that affect the response time in processing HTTP requests. These include processor load, caused by the overhead necessary to send bytes out on the network properly packetized and marshaled; disk I/O, which limits how quickly data can be delivered to the processor; the local network, over which remote data requests must be fetched; and the internet connection to the client, which often is a severe bottleneck.

Our goal is to get the requested information out of the server as fast as possible. Thus our scheduler primarily monitors the first three performance factors. The Internet bandwidth information is used partially in request redirection. Our scheduling algorithm is multi-faceted in the sense that the proper decision on routing HTTP requests needs to aggregate the impact of the above multiple system parameters on the overall system response time. We will present the architecture for SWEB and its scheduling algorithm next.

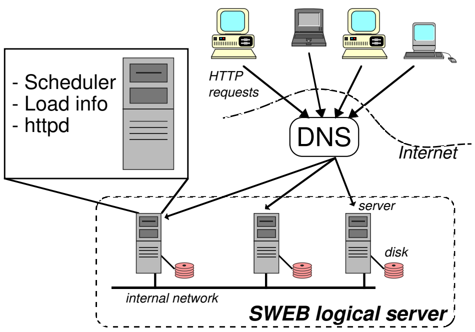

Figure 2: *The computing and storage architecture of SWEB.* DNS directs tasks to servers running task scheduler.

3.1 SWEB Architecture

There are two techniques for designing the scheduler. One is to have a centralized scheduler running on one processor such that all HTTP requests go through this processor. The scheduler monitors the usages of all system resources, makes assignment decisions based on this information, and routes requests to appropriate processors. We did not take this approach mainly because the single central distributor becomes a single point of failure, making the entire system more vulnerable, unless significant low-level precautions are taken [APB96].

The current version of SWEB uses a distributed scheduler. The user requests are first evenly routed to SWEB processors via DNS rotation, as indicated in Figure 2. The rotation on available workstation network IDs is in a round-robin fashion. This functionality is available in current DNS systems. The major advantages of this technique are simplicity, ease of implementation, and reliability [KBM94].

The DNS assigns the requests without consulting dynamically-changing system load information. Then SWEB conducts a further assignment of requests. Each processor in SWEB contains a scheduler and those processors collaborate with each other to exchange system load information. After a request is routed to a processor via DNS, the scheduler in that processor makes a decision regarding whether to process this request or assign it to another processor. Two approaches, URL redirection or request forwarding, could be used to achieve reassignment and we use the former. Request forwarding, which would involve transferring the TCP/IP connection to another server without the client's knowledge, is difficult to implement within HTTP [APB96]. URL redirection gives us excellent compatibility with current browsers and near-invisibility to users. Any HTTP request is not allowed to be redirected more than once to avoid the ping-pong effect.

The functional structure of the scheduler at each processor is depicted in Fig. 3. It contains a httpd daemon based on NCSA httpd code for handling httpd requests, with a *broker* module which determines the best possible processor to handle a given request. The broker consults with two other modules, the *oracle* and the *loadd*. The *oracle* is a miniature expert system, which uses a user-supplied table to characterize the CPU and disk demands for a particular task. The *loadd* daemon is responsible for updating the system CPU, network and disk load information periodically (every 2-3 seconds), and marking those processors which have not responded in a given period of time as unavailable. When a processor leaves or joins the resource pool, the *loadd* daemon will be aware of the change.

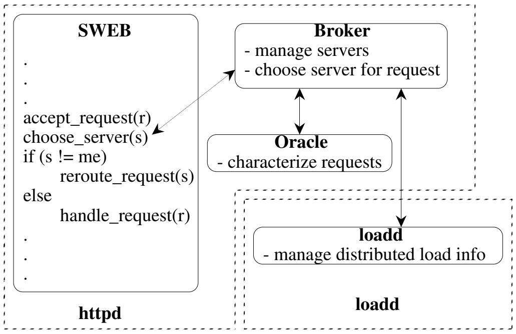

Figure 3: The functional modules of a SWEB scheduler in a processor.

3.2 The multi-faceted scheduling algorithm

In this subsection we present the algorithm that decides where a HTTP request should be routed. As we discussed before, several system load parameters affect such a decision. In a single-faceted scheduling system, a processor can be classified as lightly loaded and heavily loaded based on one parameter, e.g. CPU load. One purpose of such a classification is to update load information only when a classification changes to reduce unnecessary overhead, e.g. [SHK95]. In our problem context, it is hard to classify a processor as heavily or lightly loaded since there are several load parameters. A processor could have a light CPU load but its local disk may receive many access requests from the network file system.

We address how multiple system load parameters can be used together in deciding the assignment of HTTP requests. Since our goal is to minimize the response time for each request, we designed the heuristic based on the estimated cost for processing each request using the following formula:

$$t_s = t_{redirection} + t_{data} + t_{CPU} + t_{net}$$

$t_{redirection}$ is the cost to redirect the request to another processor, if required. t_{data} is the time to transfer the required data from the disk drive, or from the remote disk if the file is not local. t_{CPU} is the time to fork a process, perform disk reading to handle a HTTP request, plus any known associated computational cost if the request is a CGI operation. t_{net} is the cost for transferring the processing results over the Internet. We discuss and model these individual cost terms as follows.

- $$t_{data} = \begin{cases} t_{lstartup} + \frac{\text{Requested file size}}{b_{disk} \times \delta_1} \\ t_{rstartup} + \frac{\text{Requested file size}}{min(b_{disk} \times \delta_1, b_{net} \times \delta_2)} \end{cases}$$

  If the file is local, the time required to fetch the data is simply the file size divided by the available bandwidth of the local storage system, b_{disk}, plus some startup overhead $t_{lstartup}$. We also measure the disk channel load δ_1. If there are many requests, the disk transmission performance degrades accordingly.

If the data is remote, then the file must be retrieved through the interconnection network. The local network bandwidth, b_{net}, and load δ_2 must be incorporated, plus the startup overhead t_{rstart}. Experimentally, we found on the Meiko approximately a 10% penalty for a remote NFS access, and on the SUN workstations connected by Sparc/Ethernet the cost increases by 50%-70%.

In our current setting, $t_{lstartup}$ and $t_{rstartup}$ are neglected because if the files are large, these costs are relatively small and if the files are small, redirection $t_{redirection}$ and network overhead will dominate.

- $t_{CPU} = CPU_{load} \frac{\text{No. of operations required}}{CPU_{speed}}$.

  The t_{CPU} term estimates the amount of processing time required to complete the task. This is based on the speed of the server, the estimated CPU load on a destination node (CPU_{load}), and the estimated number of operations required for the task. CPU_{load} represents the number of active jobs sharing the CPU, thus we multiply the required computation time by this factor to approximate t_{CPU}. The computation requirement for a particular request is estimated by the *oracle* component of the system (see Figure 3). The parameters for different architectures are saved in a configuration file. It should be noted that some estimated CPU cycles may overlap with network and disk time and the overall cost may be overestimated slightly, but this conservative estimation works well in our experience.

  The load estimation of remote processors is based on the periodic updating of information given by those remote processors. It is possible that a processor p_x is incorrectly believed to be lightly loaded by other processors, and many requests will be redirected to it. To avoid this unsynchronized overloading, we conservatively increase the CPU load of p_x by σ. This strategy is found to be effective in [SHK95]. We use $\sigma = 30\%$.

- $t_{net} = \frac{\#\_bytes\_required}{net\_bandwidth}$

  This term is used to estimate the time necessary to return the results back to the client over the network. When the scheduler compares processors, we assume all processors will have basically the same cost for this term, so it is not estimated. Our research goal is to produce the query result for a HTTP request as fast as possible in the server site.

- $$t_{redirection} = \begin{cases} 0 \\ 2 * t_{client-server\ latency} + t_{connect} \end{cases}$$

  This term measures the time necessary to move a HTTP request from one server to another. This is set to twice the estimated latency of the connection between the server and the client ($t_{client-server\ latency}$) plus the time for a server to set up a connection ($t_{connect}$). The conceptual model is that of a very short reply going back to the client browser, who then automatically issues another request to the new server address. The transfer time is zero if the task is already local to the target server. The estimate of the link latency is available from the TCP/IP implementation, but in the initial implementation is hand-coded into the server.

Given the arrival of HTTP request r at processor x, the scheduler at processor x goes through the following steps.

1. *Preprocess a request* The server parses the HTTP commands, and completes the pathname given, checking appropriate permissions along the way. It also determines whether the requested document exists, has moved, or is a CGI program to execute.

2. *Analyze request* The server then determines whether itself or another server should fulfill the request. It does so by checking the results from the preprocessing phase. If r is already determined to be a redirection, does not exist, or is not a retrieval of information[1], then the request is always completed at x. If not, then the request is passed to the broker for analysis. The broker then:

   (a) Determines the server on whose local disk the file resides (if any).

   (b) Calculates an estimated time t_s for each available server-node for request r.

   Having determined the estimated time for each server to fill the request, the broker indicates its choice (determined by the minimum time to completion) to the main process.

3. *Redirection* If the chosen server is not x, the request is redirected appropriately.

4. *Fulfillment* At this point the request is processed in the normal HTTP server manner, with any CGI's executed as needed, and any client responses shuttled out the appropriate socket.

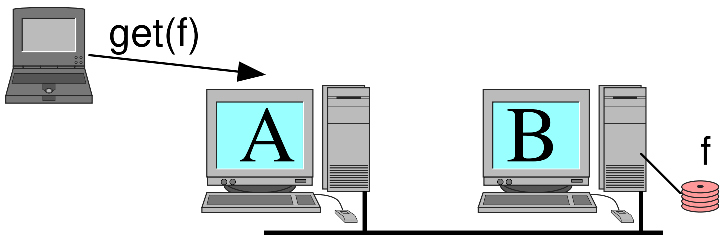

Figure 4: **Example.** Request r comes to server A for scheduling.

We give an example to demonstrate the decision process. Assume a request comes to server A (see Figure 4) for a file f of size 1.5MB physically located on server B. Assume local disk bandwidth is 1.1 MB/s, with $\delta_1 = 91\%$ available, and remote disk access bandwidth is 1.0 MB/s with $\delta_2 = 80\%$ available . Assume the latency to transfer a request is 10ms. File fetches take $\approx$ 10 instructions/byte(derived experimentally). Assume the load factor is 2.0 and 3.5 on A and B, respectively, and both machines have a computing power of 40 MIPS, with $\sigma = 30\%$. Ignoring t_{net}, we estimate the cost if this request is handled in A or B. For A,

$$t_{redirection} = 0, \quad t_{data} = \frac{1.5}{1.0 \times 80\%} = 1.875, \quad t_{CPU} = \frac{1.5 \times 10 \times (1 + 2.0)}{40MIPS} = 1.125.$$

Thus $t_A = 3.00sec$. Similarly, the cost t_B is computed below with the over-estimation rate 30%.

$$t_B = 10ms + \frac{1.5}{1.1 \times 91\%} + \frac{(1 + 30\%) \times 1.5 \times 10 \times (1 + 3.5)}{40MIPS} = 3.70sec.$$

Thus the request would be serviced at A. In this situation, due to the rough parity between network and local disk access rates, the equation is dominated by the difference in CPU loads. Conversely, if the difference between local and remote disk access is large, as in the case of an Ethernet-connected network with fast local drives, the t_{data} term will tend to override differences in CPU load. Under this condition, this request could be assigned to server node B to follow file locality.

[1] SWEB currently focuses on GET and related commands used in the HTTP protocol. Other commands (e.g., POST) are not handled, but SWEB could be extended to do so in the future.

It should be noted that modeling the cost associated with processing a HTTP request accurately is not easy. We still need to investigate further the design of such a function. Our experiments show that the current cost function does reflect the impact of multiple parameters on the overall system response performance, and that SWEB delivers acceptable performance based on such a heuristic function, adapting to dynamically-changing system resource loads. In our experiments, we will show that the overhead of SWEB scheduling is insignificant compared to the system load used for request fulfillment and other activities.

3.3 Performance Analysis

In this section, we examine the maximum sustained requests per second (RPS) achievable by our system in accessing relatively large files. This is shown by studying the performance of SWEB when it enters a steady state. At that state, the system receives a uniform number of requests per second, and produces a stable throughput of information requested. In the following analysis, we assume that the system is homogeneous in the sense that all nodes have the same CPU speed and the initial load, and each node has a local disk with the same bandwidth. We further assume that all computing resources are in use. We define the following terms:

- R – sustained RPS for the entire system
- p – the number of nodes.
- r – sustained RPS per processor. $r = \frac{R}{p}$.
- r^* – the theoretical upper bound on the sustained RPS.
- E – the average computation per request.
- F – the average file size requested.
- b_1 – the bandwidth of local disk accessing.
- b_2 – the bandwidth of remote disk accessing.
- c – the probability of a processed request to access a local disk.
- d – the average redirection probability
- d – the average redirection probability (it is stable when the system enters the steady state).
- A – the overhead in preprocessing a request, and deciding a redirection.
- O – the overhead of redirection.
- H – the average processing time for each request.

We assume that arrival rates ($r = \frac{R}{p}$) for all nodes after the division via DNS are uniform and the file sizes are uniform as shown in Figure 5. We assume the probability of accessing one of the server disks is equal (1/p) among r received requests for a node. Then $r * 1/p$ requests are accessing the local disk.

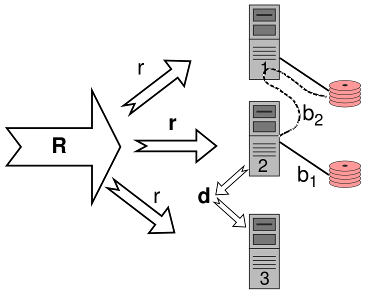

Figure 5: R requests are divided into r per machine, and server 2 in turn redirects dr to others. Server 2 has a local disk bandwidth of b_1, remote of b_2.

Among those r requests, dr of them will be redirected to other nodes (since d is the probability that a request will be redirected, so dr in total will be redirected), but dr requests will also be redirected to this node (we also assume that redirection is uniformly distributed because of the homogeneous system). Our experiments show that in such cases, the redirected requests tend to follow file locality. Thus the total number of requests processed at each processor after redirection is r requests per second. Among them, the total number of requests accessing the local disk is r/p from the original arrival tasks, plus additional d redirected requests. Then the probability of accessing a local disk for those r requests is:

$$c = \frac{\frac{r}{p} + d * r}{r} = \frac{1}{p} + d.$$

During the H seconds spent for each request, there are at least costs associated with CPU overhead for preprocessing a request, and accessing the local disk or remote disk.

Thus

$$H \geq W_1 + W_2 + W_3.$$

where W_1 is time spent for local disk access, W_2 is for remote disk access, and W_3 is the average CPU time spent for this request.

Since each request lasts H seconds, and there are r new requests arriving at each second, the number of old and new requests in progress in each processor at any particular moment is rH. For the case of accessing a local disk, the disk bandwidth is shared by rH requests with probability of c. For the case of accessing a remote disk, the network bandwidth is shared by rH requests with probability of $(1-c)$. Thus we have

$$W_1 = c * \frac{F}{\frac{b_1}{rH}}, \quad W_2 = (1-c) * \frac{F}{\frac{b_2}{rH}}.$$

The computation time is the non-overlapped CPU cycles for processing a request. Notice that CPU cycles are shared by rH requests.

$$W_3 = rH(A + E + d(A + E + O)).$$

So

$$H \geq c\frac{F}{b_1}rH + (1-c)\frac{F}{b_2}rH + (A + E + d(A + E + O))rH.$$

and substituting for c gives

$$H \geq (\frac{1}{p} + d)\frac{F}{b_1}rH + (1 - \frac{1}{p} - d)\frac{F}{min(b_1, b_2)}rH + (A + E + d(A + E + O))rH.$$

Thus

$$r \leq \frac{1}{(\frac{1}{p} + d)\frac{F}{b_1} + (1 - \frac{1}{p} - d)\frac{F}{min(b_1, b_2)} + A + E + d(A + E + O)} = r^*.$$

Then the upper bound for the sustained maximum RPS for the entire system at the steady state is:

$$R = pr^* \approx \frac{p}{(\frac{1}{p} + d)\frac{F}{b_1} + (1 - \frac{1}{p} - d)\frac{F}{min(b_1, b_2)} + A + E + d(A + E + O)}.$$

We present an example and compare with our experimental results. For the Meiko CS-2, the disk reading performance is affected by the disk file caching. It is shown in [FKC95+] that the local disk read bandwidth is about $b_1 = 5MB/s$ for a set of files with size 1.5 MB and the remote disk read bandwidth is about $b_2 = 4.5MB/s$. Our experiments on the Meiko show that d is about 0.10. The CPU cycles spent for each request for preprocessing and computation is about $0.02s$.The cost of redirection is insignificant, $O \approx 0$. With $F = 1.5MB$ and $p = 6$,

$$r^* \approx \frac{1}{(\frac{1}{6} + 0.1)\frac{1.5}{5} + (1 - \frac{1}{6} - 0.1)\frac{1.5}{4.5} + 0.02 + 0.1(0.02 + 0)} = 2.88.$$

Thus the estimated upper bound for sustained RPS on 6 nodes is R = 17.28. This is verified experimentally in Figure 1 to be 16. Thus our theoretical bound is quite close the actual sustained performance. We have tested other cases, and shown this formula is quite accurate for large files. For small files other overhead factors which have not been accounted for, such as the network OS code limitations, dominate the results and the estimates are inaccurate.

4 Experimental Results

Our primary experimental testbed consists of a cluster of workstation nodes connected by the Meiko Elan fast network. Each node has a scalar processing unit (a 40Mhz SuperSparc chip) with 32MB of RAM running Solaris 2.3. We use six CS-2 nodes, each of which is connected to a dedicated 1GB hard drive on which the test files reside. Disk service is available to all other nodes via NSF mounts. The Elan network has a peak bandwidth of 40MBytes/s. We call this configuration as Sparc/Elan. Our secondary testbed is a network of 4 SparcStation LX's running Solaris 2.4 connected via standard 10 Mb/s Ethernet. Each LX has a local 525 MB hard drive and 16MB of RAM. The effective bandwidth of this Ethernet is low since it is shared by other UCSB machines. We call this configuration as Sparc/Ethernet.

The complete SWEB system is based on NCSA httpd 1.3 source with modifications to support the distributed functionality. We plan to move to version 1.5 in the immediate future. All software uses the sockets library built on the Solaris TCP/IP streams implementation. The use of the built-in TCP/IP library was a deliberate decision based on several factors, including the following:

1. *Compatibility.* We wanted to use current WWW/HTTP compatible clients for testing purposes, as well as our own programs. Current browsers typically use TCP/IP and sockets or the equivalent.

2. *Portability*. With an implementation built on standard libraries available on virtually any UNIX system, our software can easily be moved to a heterogeneous network of workstations with little or no modifications.

There are, of course, disadvantages. The primary disadvantage is certainly performance: we were only able to achieve approximately 5-15% of the peak communication performance on Sparc/Elan, where a DMA transfer handled by the built-in Elan communications co-processor can achieve speeds near the peak bandwidth. We also felt that additional code complexity was caused by not using the built-in messaging libraries such as NX/2 or Active Messages.

We ran a series of tests where a burst of requests would arrive nearly simultaneously, simulating the action of a graphical browser such as Netscape where a number of simultaneous connections are made, one for each graphics image on the page. The clients were primarily situated within UCSB. We also tested SWEB via requests from the East coast of the US (Rutgers University) to examine the benefits of scheduling in a high network latency situation. We concentrate on the UCSB data on the scalability of the system believing this more accurately reflects the high-bandwidth networks we anticipate.

It should be noted that the results we report are average performances by running the same tests multiple times. Individual test performance is affected by dynamically-changed system loads since the machines are shared by many active users at UCSB.

We report our experiments to examine the performance of our system in the following aspects: scalability of overall performance, scheduling strategy comparison, and overhead costs.

4.1 Scalability of overall performance

Maximum RPS. The first experiment was run to determine how many requests per second SWEB could process. This depends on the average file sizes requested and the number of nodes. In [KBM94], it is reported that a high-end workstation running NCSA httpd could fulfill approximately 5 RPS. We examine how a one-node NCSA httpd 1.3 server performs, and compare it with the 6-node SWEB on Sparc/Elan and the 4-node SWEB on Sparc/Ethernet. The maximum RPS is determined by fixing the average file size and increasing the RPS until requests start to fail, which indicates that the system limit is reached. The duration of the test which simulates the burst of simultaneous requests also affects the experimental results. The requests coming in a short period can be queued and processed gradually. But the requests continuously generated in a long period cannot be queued without actively processing them since there are new requests coming after each second. We use two types of tests. One is a short period as a duration of 30 seconds and at each second a constant number of requests are launched. The long period has 120 seconds, in order to obtain the sustained maximum RPS.

Table 1: *Maximum RPS* for a test duration of 30s and 120s.

	Short period (30s)				*Sustained (120 s)*			
	Sparc/Elan		Sparc/Ethernet		Sparc/Elan		Sparc/Ethernet	
file sizes	1K	1.5M	1K	1.5M	1K	1.5M	1K	1.5M
Single server	45	9	24	4	16	2	10	< 1
SWEB	82	45	76	11	45	16	28	1

It can be seen from Table 1 that the maximum RPS of a single-node server is improved significantly by the multi-node server. The speedups vary for different file sizes. The maximum RPS obtained in a short period is much higher than that in a long period because requests accumulated in a short period can be queued. For Ethernet results with 1.5MB file size, 11 RPS is reached for duration of 30s but only 1 is achieved for sustained maximum RPS. This is because the maximum disk and Ethernet bandwidth limit is reached for this case. These results have been confirmed via the analysis in Section 3.3, which gave an analytical maximum sustained 17.8 RPS for 1.5M files on Sparc/Elan, consistent with the 16 RPS achieved in practice. It should be noted that in practice requests come in periodic bursts. Thus we use a 30-second test period in the rest of experiments, representing a non-trivial but limited burst of requests [MFM95].

Response time and drop rate. In Table 2, we report the response time (the time from after the client sends a request until the completion of this request) when we vary the number of server nodes. The system starts to drop requests if the server reaches its RPS limit. In general, the performance of Sparc/Elan is much better than that of Sparc/Ethernet with the same number of nodes, mainly because of the network performance difference.

For small file requests (1K), the multi-node server performs much better than one-node server but the response time remains constant when using 2+ processors. This is because none of the theoretical or practical limits on bandwidth or processing power have been reached for small files. The response time does increase to a certain degree after the number of processors exceeds 2, which reflects the overhead required by the scheduling algorithm and distributed file system.

For relatively large files (1.5MB), the processing time is substantially longer. When the number of processors increases, the SWEB provides substantially better performance. These results are consistent with those of NCSA, and also strongly confirm the notion that under heavy loads a distributed server solution can achieve significant speedups. Under especially heavy loads, which would tend to occur during peak hours at popular sites, a single server tends to drop almost half or more of the connections made, whereas a distributed server might have a larger overall average response time and fill every request. The superlinear speedup we obtain reflects the fact that the total size of memory in SWEB is much larger than on a one-node server, and that the multi-node server accommodates more requests within main memory while one-node server spends more time in swapping between memory and the disk. Additionally, the network overhead is distributed among multiple servers rather than concentrated at a single node.

Table 2: *Performance in terms of response times and drop rates.* On Sparc/Elan, 16 RPS, 30 sec. duration. On Sparc/Ethernet, 16 RPS for 1k file, 8 RPS for 1.5M file. The duration is 30s. Time units are in seconds, averaged over all requests. *Single server test timed out after no responses were received.

	Sparc/Elan						*Sparc/Ethernet*			
#nodes	1	2	3	4	5	6	1	2	3	4
1K file time	9.8	0.40	0.41	0.44	0.49	0.62	10.9	0.83	0.40	0.26
1K drop rate	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
1.5M file time	98.2	72.5	30.7	23.7	16.5	9.9	> 600	201.4	170.6	167.6
1.5M drop rate	37.3%	5.0%	3.5%	3.5%	3.5%	0.0%	*	20.5%	0%	0%

4.2 Comparison between different scheduling strategies

Our scheduling strategy takes into consideration the locality of requested files, and also the current resource loads. We compare our approach with others: one is the NCSA approach that uniformly distributes requests to nodes (round robin); another is to purely exploit the file locality by assigning requests to the nodes that own the requested files. We present experimental results in retrieving a set of files with uniform and non-uniform sizes.

Requests with non-uniform file size. With non-uniform file sizes, the load distribution between processors by the initial DNS assignment is heterogeneous. We expect that the round-robin approach cannot adapt such load variations. We tested the ability of the system to handle requests with sizes varying from short, approximately 100 bytes, to relatively long, approximately 1.5MB. Table 3 shows the actual response time in seconds for this case. For lightly loaded systems, SWEB performs comparably with the others. For heavily loaded systems (RPS ≥ 20), SWEB has an advantage of 15-60% over round robin and file locality.

Table 3: *Performance under non-uniform requests.* 1.5MB file size, 30 sec. duration, 0% drop rate, Sparc/Elan.

RPS	8	16	20	24	32
Round Robin	1.5	2.4	5.2	6.9	15.3
File Locality	1.7	2.5	4.4	7.9	9.3
SWEB	1.7	2.6	3.5	5.9	6.2

It is important to exploit file locality and also consider CPU load. We performed a skewed test to illustrate the fundamental weakness of the file locality heuristic where each client accessed the same file located on a single server, effectively reducing the parallel system to a single server. In this situation, round-robin handily outperforms file locality, with and average response times of 3.7s and 81.4s, respectively. This test was performed with six servers, 8 RPS, for 45s, and file size of 1.5MB.

We also ran clients on the east coast (Rutgers University, New Jersey), and the tests results show a typical performance gain of over 10% using file locality instead of round robin from an Ethernet-linked server, in spite of the poor bandwidth and long latency over the connection from the east coast to the west coast.

Requests with uniform file size. For uniform file sizes, the DNS scheme assigns HTTP requests evenly to nodes, which leads to a homogeneous distribution of CPU load, but the network communication load is not minimized. We expect that the advantages of the re-scheduling will manifest in a high communication cost environment. On Sparc/Elan, we conducted several tests and the result shows that three strategies have similar performance. This is because NFS is implemented in Sparc/Elan through the fast Elan fat tree network and the chance of network contention is much smaller than on Ethernet. In a relatively slow, bus-type Ethernet, the advantage of exploiting file locality is more clear. This is verified in an experiment shown in Table 4.

4.3 Overhead of SWEB

Overhead distribution from the client point of view. SWEB code was instrumented to determine the overhead cost for a request. Table 5 shows the case of a 1.5MB file fetched over a fairly heavily loaded system. We examine the performance in one case as follows; the results from other cases are similar. The total time from request initiation

Table 4: *Performance under uniform requests on Sparc/Ethernet.* File size 1.5MB, 0% drop rate.

RPS	2	4	8
Round Robin	48.1	128.3	243.7
File locality	42.25	124.2	219.0
SWEB	41.6	124.8	210.4

to receiving the last byte of information was approximately 5.4 seconds, of which the server spent 4.9 transmitting data, with approximately 10% of the total time spent acquiring the connection, handling the redirection, and a mere 5 msec. spent in the scheduling algorithm. The algorithm requires approximately 4 msec. for analysis when the URL has not been previously redirected, and about 1 msec. when it has. The results indicate that the overall overhead introduced by SWEB analysis and scheduling algorithm is insignificant. For small files (approximately 1K), the data transfer time is smaller, but the overhead of SWEB compared to the preprocessing time for parsing HTTP commands is still very small. The direct cost of analysis is typically about 1-4 ms. for cost estimation and 4 ms. to generate a redirection if necessary. Indirect costs which affect the time required to complete a redirection depend heavily on the latency between client and server. For a client fetching a 1.5M file on the Sparc/Elan, of the 5.4 sec. total time, well over 90% is spent doing data transfer.

Table 5: *Cost distribution in average response time.* 1.5M file size, Sparc/Elan. Items marked "SWEB" are introduced by the SWEB system. All other times are due to standard HTTPd functionality. Times for smaller requests will vary in the data transfer component, but remain essentially the same for the others.

Activity	*Time*
Preprocessing Req.	70 msec.
Analysis (SWEB)	1 or 4 msec.
Redirection (SWEB)	4 msec.
Data Transfer	4.9 sec.
Network Costs	0.5 sec.
Total Client Time	5.4 sec.

Overhead distribution from the server point of view. From the server's point of view, we need to examine how much load the SWEB contributes for scheduling decisions and resource information collection. Table 6 shows that in processing requests for files of sizes 1.5MB when 16 RPS, 4.4% of CPU cycles are used for parsing the HTML commands, but less than 0.01% time is used for collecting load information and making scheduling decisions. Approximately 0.2% of the available CPU is used for load monitoring. We have also tested the overhead with small files (1K), and the percentage of overhead for load information collection and scheduling again is insignificant. The relative percentages would change as the processing/fulfillment phase decreased with the file size.

5 Conclusions

We have implemented a WWW server running on multi-computer systems. We show that it is important to incorporate multiple load factors (CPU, I/O and network) in one scheduling framework. Our scheme effectively monitors and utilizes multiple system resources, and the overhead required to support such a scheme is extremely low. Our scheduler gives a performance advantage of up to 50% over other commonly used techniques in our tests.

Table 6: *CPU time distribution at a SWEB server node.* The test is for 1K and 1.5M file sizes on Sparc/Elan with RPS=16.

task	1K (%)	1.5M (%)
Request Parsing	70	4.4
Server Analysis (SWEB)	0.6	< 0.01
Load monitoring(SWEB)	< 0.2	< 0.2
Redirection(SWEB)	< 0.01	< 0.01
Processing/fulfillment	28	95.5

We are currently migrating the system to be the primary server for the ADL and investigating its performance in a heterogeneous environment [WA95+].

Acknowledgments

This work was supported in part by funding from NSF IRI94-11330 and CDA-9529418, and a Navy NRaD grant. We would like to thank Omer Egecioglu, Terry Smith, Cong Fu and the Alexandria Digital Library team for many valuable discussions, Vegard Holmedahl for programming and debugging the system. We would also like to thank the anonymous referees for their useful comments.

References

[A96] The Alexandria Digital Library Project, http://alexandria.sdc.ucsb.edu/.

[APB96] E. Anderson, D. Patterson, E. Brewer, "The Magicrouter, an Application of Fast Packet Interposing", submitted to the Second Symposium on Operating System Design and Implementation (OSDI96), 1996. Also available at http://HTTP.CS.Berkeley.EDU/~eanders/262/262paper.ps.

[AC95+] D.Andresen, L.Carver, R.Dolin, C.Fischer, J.Frew, M.Goodchild, O.Ibarra, R.Kothuri, M.Larsgaard, B.Manjunath, D.Nebert, J.Simpson, T.Smith, T.Yang, Q.Zheng, "The WWW Prototype of the Alexandria Digital Library", *Proceedings of ISDL'95: International Symposium on Digital Libraries,* Japan August 22 - 25, 1995.

[AY95+] D.Andresen, T.Yang, V.Holmedahl, O.Ibarra, "SWEB: Towards a Scalable World Wide Web Server on Multicomputers", *Dept. of Computer Science Tech Rpt. TRCS95-17* U.C. Santa Barbara, Sept., 1995, http://www.cs.ucsb.edu/ Research/rapid_sweb/SWEB.html. A short version appears in *Proc. of 10th IEEE International Symp. on Parallel Processing (IPPS'96)*, pp. 850-856, April, 1996.

[SHK95] B. A. Shirazi, A. R. Hurson, and K. M. Kavi (Eds), Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE CS Press, 1995.

[BR96] E. Brewer, Personal communication, http://inktomi.berkeley.edu, Jan., 1996.

[FKC95+] C. Fu, M. Kim, D. Chang, O. H. Ibarra, T. Yang, Performance Evaluation of Parallel I/O on the Meiko CS-2. Tech Report, UCSB 1995.

[GDI93] K. Goswami, M. Devarakonda, R. Iyer, Prediction-based Dynamic Load-sharing Heuristics, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 638-648, June, 1993.

[HL95+] J. Hsieh, M. Lin, J. Liu, D. Du, Performance of A Mass Storage System for Video-On-Demand, Special Issue of the *Journal of Parallel and Distributed Computing* on Multimedia Processing and Technology, Vol.30, No.2, pp.147-167, November, 1995.

[HT95] Hypertext Transfer Protocol(HTTP): A protocol for networked information, http://www.w3.org/hypertext/WWW/Protocols/, June 26, 1995.

[LYC95] Lycos Usage: Accesses per Day, http://lycos.cs.cmu.edu/usage-day.html.

[KBM94] E.D. Katz, M. Butler, R. McGrath, A Scalable HTTP Server: the NCSA Prototype, *Computer Networks and ISDN Systems.* vol. 27, 1994, pp. 155-164.

[MFM95] D. Mosedale, W. Foss, R. McCool, "Administering Very High Volume Internet Services", *Proc. of 1995 LISA IX*, Monterey, CA, September, 1995.

[WA95+] R. Wolski, C. Anglano, J. Schopf, F. Berman, Developing Heterogeneous Applications Using Zoom and HeNCE, *Proceedings of the Heterogeneous Computing Workshop, HCW '95*, pp. 12-21, Santa Barbara, CA, IEEE, April, 1995.