SWEB: Towards a Scalable World Wide Web Server on Multicomputers

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

Abstract

We investigate the issues involved in developing a scalable World Wide Web (WWW) server on a cluster of workstations and parallel machines, using the Hypertext Transport Protocol (HTTP). The main objective is to strengthen the processing capabilities of such a server by utilizing the power of multicomputers to match huge demands in simultaneous access requests from the Internet. We have implemented a system called SWEB on a distributed memory machine, the Meiko CS-2, and networked SUN and DEC workstations. The scheduling component of the system actively monitors the usages of CPU, I/O channels and the interconnection network to effectively distribute HTTP requests across processing units to exploit task and I/O parallelism. We present the experimental results on the performance of this system. Our results indicate that the system delivers good performance on multi-computers and obtains significant improvements over other approaches.

1 Motivation

The Scalable Web server (SWEB) project grew out of the needs of the Alexandria Digital Library (ADL) project at UCSB [A95]. 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. The Alexandria Project is focused on the design, implementation, and deployment of a distributed digital library for spatially-indexed information. The collections of the library currently involve geographically-referenced materials, such as maps, satellite images, digitized aerial photographs, and associated metadata. The fundamental goal of the Alexandria Project is to permit users who are distributed over the Internet to access broad classes of spatially-indexed materials that may themselves be distributed over the Internet and to derive useful information from such materials. A rapid prototype system for the Alexandria digital library has already been developed which allows users to browse spatially-indexed maps and images [A95]. To expose the system to a broad user community, the next version of the system (available at the end of 1995) will be connected to the World Wide Web (WWW) using the Hypertext Transport Protocol (HTTP).

Our work is motivated by the fact that the Alexandria digital library WWW server has a potential to become the bottleneck in delivering digitized documents over high-speed Internet. Popular WWW sites such as the *Lycos* and *Yahoo* [LYC95] receive over one million accesses a day. For example, in 1993 alone, the weekly access rate for NCSA's HTTP server at UIUC increased from 91K to 1.5M. The peak request arrival rate at NCSA exceeds more than 20 requests/second, but a single, high-end workstation can handle only a limited number of requests per second [KB94]. The requests to the NCSA server typically involve only simple file retrieval operations. For WWW-based network information systems such as digital libraries, the servers involve much more intensive I/O and heterogeneous CPU activities. This is because such systems process a large amount of digitized images. And content-based searching and multi-resolution image browsing [A95] have large demands in computation speed, disk storage and network bandwidth in the server. To meet such demands, the SWEB project makes use of existing networked computers and inexpensive disk resources to strengthen the processing and storage capabilities of WWW servers. During the last few years, it has become commonplace that local networks span large numbers of powerful workstations connected with inexpensive disks. Those machines are idle much of time. Also the price/performance ratio of parallel machines has dropped rapidly. We expect that recycling 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.

We have developed the preliminary version of a scalable WWW server (SWEB) [A95] running on a Meiko CS-2 distributed memory machine and a network of workstations (NOW) such as SUN and DEC machines. Each processing unit is capable of handling a user request following the HTTP protocol. The distinguishing feature of SWEB is effective resource utilization by a close collaboration of multiple processing units. Each processing unit is a processor (e.g. SUN SPARC or a Meiko CS-2 node) possibly linked to a local disk. Several resource constraints affect the performance of the server: the CPU speed and memory size of one processing unit, the existing system load, the transmission bandwidth between the processing unit and its local disk, the network latency and bandwidth between a processing unit and a remote disk when the accessed files are not stored in the local disk, and disk contention when multiple I/O requests access the same disk, can all affect the performance. 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 efficient processing.

The paper is organized as follows: Section 2 discusses the related work. Section 3 gives the background and problem definition. Section 4 discusses the possible approaches for building a scalable

WEB server and the SWEB organization and load balancing strategies. Section 5 presents the experimental results showing the performance of SWEB and compares our approach with others. Section 6 provides analytic results on a performance bound of SWEB and verifies it with the actual experiment results. Section 7 discusses conclusions and future work.

2 Related Work

Numerous other initiatives to create high-performance HTTP servers have been reported. NCSA [KB94] 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 chunks 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. The domain name resolution in a round-robin fashion 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. For example, all the requests from UCSB might go to server $S2$, if its IP address happened to be cached in some previous lookup, even though server $S1$ might be relatively idle.

Recently NCSA has come out with version 1.4 of their httpd software, which in many cases doubles the performance of previous versions by utilizing a pre-forking strategy [NCH95]. This technique can be incorporated in our system and will be beneficial. However we have not done so at this stage and in this paper we focus on alternative means of achieving the scalability through dynamic scheduling and request redirection.

Other projects have focused on improving the performance of a single-workstation server. The Apache project has programmed a set of changes to the early versions of NCSA httpd to make the server more efficient. The CERN server has achieved significant improvements. The Harvest project [BD94] has introduced a HTTP cache which can be a major accelerator for pages without a dynamically created component. All these efforts are concentrating on the single workstation server level and can be embodied in our system, while our work focuses on a collaborating multi-workstation server.

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 one particular document such as one film beginning to end. Thirdly, we assume lossless document delivery, keeping consistency between the source library holding and what is sent to the client.

Our dynamic scheduling scheme is closely related to the previous work on load balancing on distributed systems [ELZ86, BCS95, WR93]. 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 most well known techniques are called receiver-initiated and sender-initiated strategies. In those studies, 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 the 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 only depend on CPU loads. Thus we need to develop a multi-faceted scheduling scheme that can effectively utilize the system resources by considering multiple performance parameters.

3 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, BC95, BL95].

- HTML is derived from the Standard Generalized Markup Language (SGML) optimized for presenting hypertext information. It operates on the basis of *tags*, where each piece of the text is marked to indicate the format to be used. HTML is similar to other markup languages such as LaTeX and RTF. HTML version 1.0 contained little beyond hypertext links and standard formatting functions such as bold, center, etc. Later versions have been extended to include the *forms* interface, where users can fill in data and submit the form to a server; *tables*, which mimic the layout of a spreadsheet; and inline images. In addition to the formal standard, many browser companies (e.g. Netscape) have added custom extensions to their implementations.

- The Uniform Resource Locator (URL) serves as an index to a particular service or document, giving the hostname, the access method, and document/application path in a single concise format. For example, the URL http://www.cs.ucsb.edu/ would fetch the default document (or page) from the www server at the Computer Science department at UCSB using the HTTP protocol. ftp://www.cs.ucsb.edu/myfile uses the FTP protocol to fetch the document "myfile" from the same machine. URLs are not limited to simply fetching files, however, but can specify a program to execute. E.g., http://www.cs.ucsb.edu/cgi-bin/uptime causes the "uptime" program to execute on the www server and return its results to the client. This ability to transparently select a resource which might be the results of a dynamic execution or simply a file fetch makes a number of internet services possible without adding complexity for the user.

- The HTTP protocol (HTTP) is a stateless, connection-oriented application-level protocol used for network hypermedia information systems. A number of WWW browsers and servers use such protocols, for example, NetCom's NetScape browser, the NCSA and CERN HTTP servers[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 response code (e.g., 202 in the HTTP 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.

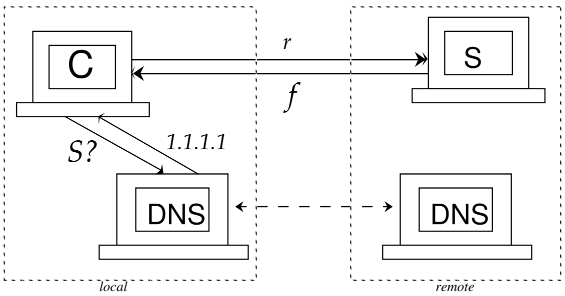

Figure 1: **A simple HTTP transaction.** Client C looks up the address of server S, sends over request r, and receives response f.

4 SWEB: A scalable WWW server

We call a WWW server *scalable* if the system response time for individual requests remains relatively constant when the the number of simultaneous HTTP requests increases. 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. 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 [KB94], which cannot match the current and future loads (e.g. a digital library server). Thus multiple servers are needed for achieving scalable performance.

We assume that the architecture of our system contains a set of networked workstations. Some of the workstations are connected to SCSI disks or mass storage RAID subsystems in which HTML files are stored. The disks are included in a shared file system among all processing units. Our current implementation runs on NOWs using Ethernet (we will use ATM in the future) and a Meiko CS-2 machine which contains a set of distributed memory SPARC processors connected by a fat tree based communication network. All files are available to each processor via NFS mounts. The NOW version runs on a set of SUN Sparc stations with Solaris operating systems, and DEC Alpha-based workstations with OSF1. Fig. 2 depicts the architecture assumptions of SWEB. In this paper, the term workstation unit and processor are interchangeable. We also assume that each CPU unit may be used by other applications and it could leave and join the resource pool at any time.

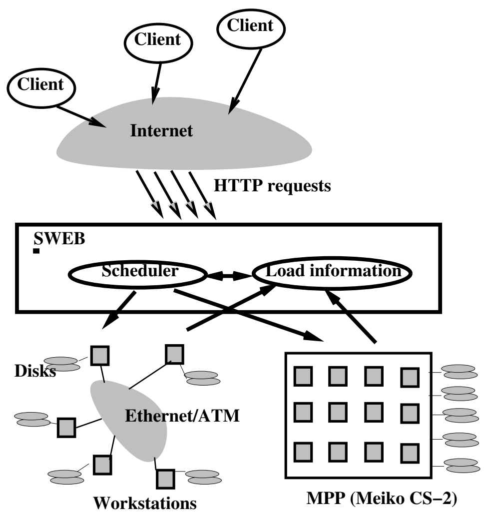

Figure 2: The computing and storage architecture of SWEB.

The overall objective of the system is to reduce and sustain the response time under large numbers of simultaneous requests. Goals considered in designing this system are:

- To demonstrate how to utilize existing inexpensive commodity network, workstation, and disk resources to build a scalable WWW server. The computing environment could be heterogeneous and workstation/processor units with different speeds and different loads at any time.
- To develop sophisticated dynamic scheduling algorithms for exploiting task and I/O parallelism adaptive to the run-time change of system resource loads and availabilities. The system needs to provide good system resource estimation to assist the scheduler. The scheduler needs to incorporate multiple system performance parameters to assign user HTTP requests to a proper unit for efficient processing. The overhead involved for current resource load assessment and scheduling decision making should be minimized.

In this subsection we will first discuss factors that affect scalability of the system, introduce the functional modules of SWEB, and discuss the approaches for routing a HTTP request to a processor based on the scheduling decision (since such a routing scheme affects the design of the scheduling heuristic). Then we will present our scheduling algorithm for determining the processor assignment of a given HTTP request.

4.1 Performance factors

There are several performance factors that affect the response time in processing HTTP requests. We discuss them in more detail as follows.

1. *Processor load.*

   A HTTP request is handled through a TCP/IP connection and then subsequently through a forked subprocess. This subprocess may retrieve a file or invoke a CGI program, which requires varying amounts of CPU time. Notice that a processing unit can also be used for applications other than WWW. The load of a processing unit must be monitored so that HTTP requests can be distributed to relatively lightly loaded processors.

2. *Disk I/O.*

   Since most HTTP requests retrieve files from disks, the I/O requirements can present further bottlenecks. For instance, whether the files are on local disk or fetched over a local network can affect performance substantially. The raw data transfer rate of disks is in the range of 2-3 MByte per second while the RAID subsystems can achieve substantially faster performance

(20MB or more per second). If many requests are accessing the same disk at the same time, the I/O performance can degrade significantly due to disk channel contention. If simultaneous user requests are accessing different disks, then parallel I/O leads to a higher throughput in delivering documents.

3. *Interconnection network.*

   HTTP requests usually access HTML files, but they could activate the execution of a program. The local interconnection network bandwidth affects the performance of file retrieval since many files may not reside in the local disk of a processor. Remote file retrieval through the network file system will be involved. The network traffic congestion could dramatically slow the HTTP request processing. However, in some cases, given a very fast network and NFS server, it is possible that a remote access can actually be faster than pulling data from the local disk, if the local disk has a high channel contention.

4. *Internet.*

   The Internet speed significantly affects the performance in information delivery. Currently, Internet bandwidth is increasing rapidly with the advent of ATM, BISDN, and other WAN technologies. However, the NCSA work [KB94] and our experiment show that the Internet is not necessarily the bottleneck in a high traffic system, and the server is another bottleneck for document delivery.

Our goal is to get the requested information out of the server as fast as possible. Thus our scheduler primarily monitors the above first three performance factors. The Internet bandwidth information is used partially in request re-direction. 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.

4.2 Distributed vs. centralized scheduler

There are two approaches in designing the scheduler. One is to have a centralized scheduler running 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.

The current version of SWEB uses a distributed scheduler. The user requests are first evenly routed to SWEB processors via the DNS scheme as shown in Fig. 3. The use of DNS rotation to provide

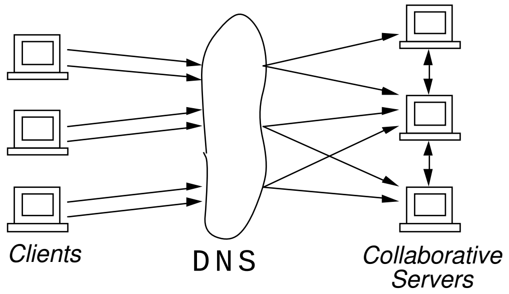

Figure 3: Distributed collaborative scheduler.

the initial assignment of HTTP requests is used in the NCSA multi-workstation server [KB94]. In this scheme, multiple real machines are mapped to the same IP name. As shown in Fig. 4, when a client requests the network ID of the machine name (e.g., www.cs.ucsb.edu), the DNS at the server site will rotate the network IDs, pick up one (e.g., 1.1.1.1) and send back to the client. 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 [KB94].

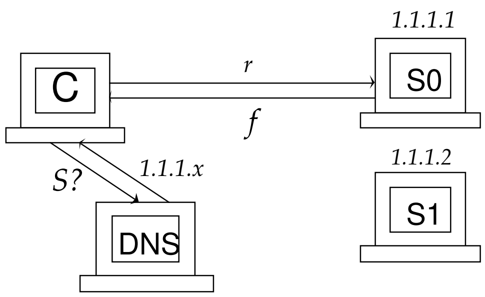

Figure 4: **DNS Rotation.** The DNS server returns differing IP addresses for a given name in round-robin fashion from a list of servers.

The DNS assigns the requests without consulting dynamically-changing system load information. Thus the SWEB conducts a further re-direction of requests. Each processor in SWEB contains a scheduler and those processors collaborate with each others 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 redirect it to another processor. Any HTTP request is not allowed to be re-routed more than once to avoid the ping-pong effect. This architecture is substantially less prone to single-point-of-failure breakdowns, is more distributed, and the overhead of load balancing is distributed across a number of nodes.

The functional structure of the scheduler at each processor is depicted in Fig. 5. 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 preset period of time as unavailable. When a processor leaves or joins the resource pool, the *loadd* daemon will be aware of the change.

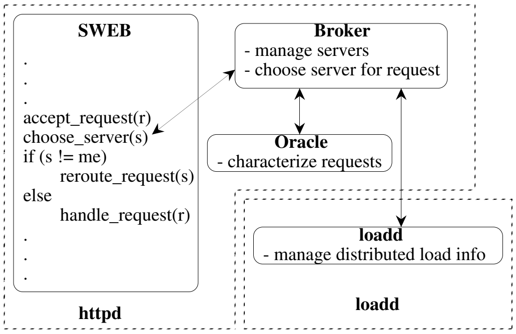

Figure 5: The functional modules of a SWEB scheduler in a processor.

4.3 Routing HTTP requests

After a processor accepts a HTTP request through TCP connection and a system decides to assign a request to a particular processor, this request needs to be *transparently* routed to this processor.

The best situation would be to modify the UNIX sockets package to change the semantics of the "accept" system call. Currently the http daemon at the server sits listening on a pre-defined port. When a client wishes to connect, the server then negotiates another port over which to conduct the transaction, while continuing to listen on the original port. Setting this new port to be on a separate machine is difficult since it requires a substantial modification of the UNIX operating system kernel.

The approach that we eventually decided to implement is based on the fact that the HTTP protocol allows for a response called "URL Redirection". As shown in Fig 6, when client C sends a request to server $S0$, $S0$ returns a rewritten URL r' and a response code indicating the information is located at r'. C then follows r' to retrieve the resulting file f. Most Net browsers and clients automatically query the new location, so redirection is virtually transparent to the user. The SWEB algorithm that incorporates the URL redirection with the scheduler is shown in Fig. 7.

The primary advantages of URL redirection are the simplicity of implementation and universal compatibility. An simple control program can be set up within a modified WWW server, where the main complexity lies in the routines to determine the optimal server for a particular request.

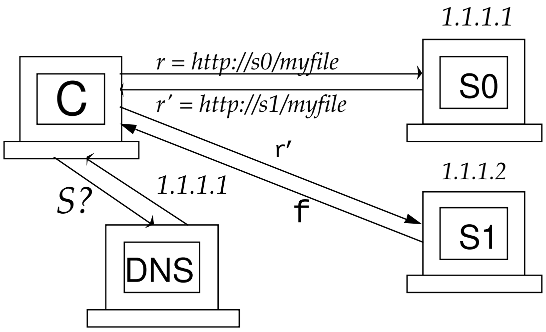

Figure 6: The URL redirection.

```
while(true)
    listen_on_well_known_port()
    accept_URL_request(x)
    parse(x)
    s = scheduler_determine_optimal_processor(x)
    if( s == me )
     fulfill_request()
    else
             x' = rewrite_URL(s,x)
             return_as_redirection(x')
    endif
endwhile
```

Figure 7: the URL redirection algorithm.

Furthermore, the approach lies well within our design parameters, by not requiring a modification of the HTTP protocol, being reasonably efficient, and being able to support sophisticated optimization algorithms.

The primary disadvantage of URL redirection in practice is the added overhead of an additional connect/pass request/parse/respond cycle after the redirection occurs. We will show that such overhead is more than negated by improved performance overall, especially in situations where multi-megabyte files are a common request. If the request is for a small document, it is generally better for a processor to handle the request itself without redirection. Our scheduler will consider such effects.

4.4 The multi-faceted scheduling algorithm

In this subsection we discuss 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. [ELZ86, WR93]. In our problem context, it is hard to classify a processor as heavily or lighted 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. Thus each processor in SWEB has a load daemon to detect its own CPU, disk, and network load, periodically broadcasting such information to other processors. In our experiments we will show that such overhead is insignificant.

Our initial algorithm took into account only the location of the file to be fetched, and rerouted the request to that processor. We called this strategy *file locality.* Initial results showed that under many conditions an improvement over simple round-robin was apparent. In some cases, such as many requests for a single file, performance fell off drastically.

We then addressed how multiple system load parameters can be used together in deciding the assignment of the HTTP requests. Since our goal is to minimize the response time for each request, we design 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.

- $$t_{data} = \begin{cases} \frac{\text{Requested file size}}{\text{Disk bandwidth}_{\times\delta_1}} & \text{if file is local} \\ \frac{\text{Requested file size}}{min(\text{Disk bandwidth}_{\times\delta_1},\ \text{Local network bandwidth}_{\times\delta_2})} & \text{if file is remote} \end{cases}$$

  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. 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 and load δ_2 must be incorporated. Experimentally, we found on the Meiko approximately a 10% penalty for a NFS access, and on the SUN workstations connected by Ethernet the cost increases by 50%-70%.

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

  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 load on a destination node (CPU_{load}), and the estimated number of operations required for the task. The computation requirement for a particular request is estimated by the *Oracle* component of the system (see Figure 5). Currently we mainly estimate the CPU cost involved in requesting a file. 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 such estimation still helps us determine the slowdown in file transfer when the CPU is shared by others.

  One surprising aspect of our research to date is the experimental result that a simple file fetch from a http server results in substantial CPU activity. For a node on the Meiko or SUN, each concurrent request adds .2 or more to the running load average (a load of 2 or more is considered heavy using the UNIX load estimate function uptime()). In this paper, we assume most requests are retrieving HTML or other files. We will incorporate cost functions for CGI operations as accurately as possible as the system is further developed for specialized applications such as digital libraries where knowledge on special CGI operations are available. It should be noted that an inaccurate estimate of computational cost is possible, but the system tries to respond to such a dynamic situation by accessing the current information on processor load as it is updated periodically and adjusting its scheduling decision accordingly.

  The load estimation of remote processors is based on the periodic updating of information informed by those remote processors (every 2-3 seconds). 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 [WR93]. We use $\sigma = 30\%$ in our system.

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

  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. Thus we do not estimate this term. Our goal is to produce the query result for a HTTP request as fast as possible in the server site.

- $$t_{redirection} = \begin{cases} 0 & \text{if task is executed locally} \\ 2 * t_{client-server\ latency} + t_{connect} & \text{if redirection is used} \end{cases}$$

This term measures the time necessary to move the HTTP request from one server to another. This is set to twice the estimated latency of the connection between the server and the client plus the time for a server to set up a connection. 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, determining 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 fill 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. Additionally, if the request has already been redirected by SWEB it will always be accepted. If this is not the case, 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 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.

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 experiment shows that the current cost function does reflect the impact of multiple parameters on the overall

[1]SWEB currently focuses on GET and related commands used mostly the HTTP protocol. Other commands such as POST are not handled, but could be extended in future.

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.

5 Experimental Results

Our primary experimental testbed consists of a Meiko CS-2 distributed memory machine at UCSB. Each node has a scalar processing unit (a 40Mhz SuperSparc chip) with 32MB of RAM running Solaris 2.3 [ME94]. We mainly 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 These nodes are connected via a modified fat-tree network with a peak bandwidth of 40MB/s. 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. All performance numbers in the following analysis refer to the primary testbed unless otherwise noted. All time units are seconds unless otherwise noted.

The initial software environment consisted of one copy of unmodified NCSA httpd 1.3 running on each server node being tested, and a custom www client running on each client node. The complete SWEB system is based on NCSA httpd 1.3 source with extensive modifications to support the distributed functionality. Each custom client accesses the broker, and is then redirected (if necessary) to the appropriate server to fulfill the request. 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 NOW with 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 the Meiko, where a DMA transfer handled by the built-in Elan communications coprocessor 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.

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. But we mainly examine the UCSB data on the scalability of the system since our goal is to get the data out of the server as fast as possible.

It should be noted that the results we report are average performances by running the same tests multiple times over an extended period. The 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.** We will determine the RPS (request per second) limit that is achievable by our system, compared with the one-node performance. For the current system, if the number of requests exceeds the RPS limit, the requests will be dropped. We will report the drop rate of the system, and examine whether speedups in terms of response time can be achieved using multiple nodes in the SWEB server.

- **Comparative study**. We will compare our approach with the other approaches. One is the NCSA round-robin method, which cyclically distributes the requests among processing nodes. We also examine the importance of exploiting file locality by assigning requests to processors who are physically attached to the disks containing the requested files.

- **Overhead distribution**. We examine how much overhead is imposed on our system in monitoring the system resource utilization, making decisions and redirection. We study the overhead of distribution from the point of view of both client and server.

5.1 Scalability of overall performance

Maximum RPS on Meiko CS-2 and NOW. 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 [KB94], it is reported that a high-end workstation running NCSA httpd could fulfill approximately 5 requests per second (rps). We examine how a one-node NCSA httpd 1.3 server performs, and compare it with the 6-node SWEB on Meiko CS-2 and the 4-node SWEB on NOW. 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. In Section 6, we will provide an analysis of the sustained maximum RPS and verify this result.

	Short period (30s)				*Sustained (120 s)*			
	Meiko		NOW		Meiko		NOW	
file sizes	1K	1.5M	1K	1.5M	1K	1.5M	1K	1.5M
Single server	45	9	24	4	16	2	10	< 1
Multi-node server	82	45	76	11	45	16	28	1

Table 1: *Maximum RPS* for a test duration of 30s and 120s on Meiko CS-2 (left part) and NOW (right).

It can be seen 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 NOW 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. However it should be noticed that in practice, a burst of requests does come periodically but not constantly. Thus we will mainly use the 30-second test period in the rest of experiments.

Response time and drop rate on Meiko CS-2 and NOW. 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 RPS reaches the limit. For small file requests (1K), the multi-node server performs much better than one-node server but the response remains constant when using 2+ processors, because none of the theoretical or practical limits on bandwidth or processing power have been reached. 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.

	Meiko						*NOW*			
#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%

Table 2: *Performance in terms of response times and drop rates.* On Meiko CS-2, RPS is 16 and the duration is 30s. On NOW, RPS=16 for 1k file, RPS=8 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.

5.2 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 the requests to nodes (round robin); another is to purely exploit the file locality by assigning requests to the nodes that own the requested files.

Requests with uniform file sizes. Table 3 lists two performance tables to compare the round robin and the SWEB approach when varying the RPS, keeping the number of servers constant at six and the file size uniformly at 1KB and 1.5MB respectively. The response times do not vary substantially, and our scheduling algorithm provides little benefit compared to pure round-robin.

Thus on Meiko CS-2, exploiting file locality does not improve the performance dramatically for the requests with uniform sizes. This is because NFS is implemented in Meiko CS-2 through the fast Elan fat tree network, where due to increased bandwidth the chance of network contention is much smaller than on Ethernet, and the throughput is correspondingly higher. According to a series of tests performed on the Meiko, the bandwidth of a remote NFS file accessing only degrades 10%

RPS	8	16	20	24	32
Round Robin	0.63	0.56	0.55	0.58	0.67
file locality	0.52	0.48	0.50	0.48	0.54
SWEB	0.56	0.48	0.48	0.52	0.48

RPS	8	16	20	24	32
Round Robin	3.4	8.7	14.1	20.0	27.4
File Locality	3.4	8.8	11.6	19.0	29.6
SWEB	3.4	8.5	14.0	19.3	26.7

Table 3: The top table lists the response times in seconds under small uniform requests with 1KB file size on Meiko. The duration is 30 sec. The drop rate is 0% for all cases. The bottom table lists response times under uniform requests with size 1.5MB.

compared to local disk accessing. Also requests with uniform file sizes do not lead to heterogeneous load variations between processors and the round-robin performs well for such cases.

In a relatively slow, bus-type Ethernet in a NOW environment, the advantage of exploiting file locality is significant as shown in Table 4. We ran clients at the east coast (Rutgers University, New Jersey), the tests results show a performance gain of over 10% using file locality instead of round robin, in spite of the poor bandwidth and long latency over the connection from the east coast to the west coast.

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

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

However, exploiting file locality without consideration of CPU load does not lead to a good performance. A skewed test was performed 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.

Request with non-uniform file size. With uniform file sizes, round-robin performs well on the Meiko CS-2, but not in the NOW where file locality is important. For the non-uniform file size, the performance of round-robin will not be comparable with SWEB because the non-uniform file access leads to heterogeneous load distribution of the system, and the round-robin approach cannot adapt such load variations.

We tested the ability of the system to handle requests which randomly varied file sizes requested (either very short, c. 100 bytes, or relatively long, c. 1.5MB). We anticipated that round-robin and simple file locality would prove less efficient in this case and the results bore this out. Table 5 shows the actual performance and SWEB has an advantage of over 50% over round robin.

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

Table 5: Performance under nonuniform requests. 30 sec. duration, 0% drop rate, Meiko CS-2.

5.3 Overhead of SWEB

Overhead distribution from the client point of view. SWEB code was instrumented to determine the overhead cost for a request. Table 6 shows the case of a 1.5MB file fetched over a fairly heavily loaded system. The total time from request initiation 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 with 1K, the data transferring time is reduced, but the overhead of SWEB compared to the preprocessing time for parsing HTTP commands is still very small.

It should be noted that the times in Table 6 are measured using UNIX function $gettimeofday()$. For example, 70 msec spent on preprocessing requests actually take 20 msec of CPU cycles since CPUs are shared by many users.

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 7 shows that in processing requests for files of sizes 1.5MB when RPS=16, 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. 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 proportionally with the file size.

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.

Table 6: *Cost distribution in average response time.* This test is for 1.5M file size on Meiko. Items marked "SWEB" are introduced by the SWEB system. All other times are due to standard HTTP server functionality. Times for smaller requests (c. 1K) will vary in the data transfer component, but remain essentially the same for the others.

task	%
Request Parsing	4.4
Server Analysis	< 0.01
Load monitoring	< 0.2
Redirection	< 0.01
Processing/fulfillment	95.5

Table 7: *CPU time distribution at a SWEB server node.* The test is for 1.5M file size on Meiko CS-2 with RPS=16.

6 An analysis on maximum sustained RPS

In this section, we verify the maximum sustained RPS achieved 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 define the following terms:

- p — the number of nodes.
- r — sustained RPS per processor. r^* — the theoretical upper bound on the sustained RPS.
- 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 (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) for all nodes are uniform and the file sizes are uniform. 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.

Among those r requests, dr of them will be redirected to other node but dr requests will be redirected to this node (we also assume that redirection is uniformly distributed because of the homogeneous system). Our experiment shows that in such cases, the redirected requests follow the file locality. Thus the total number of requests processed at each processor after redirection is r requests per second. Among them, the total request 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 handled by each processor at any particular moment is rH. For the case of accessing 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 cost is the non-overlapped CPU cycles for processing a request. Notice that CPU cycles are shared by rH requests.

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

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

Then the upper bound for the sustained maximum RPS per processor at the steady state is:

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

For the Meiko CS-2, the disk reading performance is affected by the disk file caching. It is tested in [FKC95+] that the local disk read bandwidth is about $b_1 = 5MBytes/s$ 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 for Meiko show that d is about 0.10. The CPU cycle spent for each request for preprocessing is about $0.02s$. The cost of redirection is insignificant, $O \approx 0$. With $F = 1.5MB$ and $p = 6$, $r^* \approx 2.88$. Thus the estimated upper bound for sustained RPS on 6 nodes is 17.28. Notice our experimental number in Section 5 is 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.

7 Conclusions and Future Work

We have implemented a WWW server running on a distributed memory machine and a NOW. We show that our scheduling algorithm effectively monitors and utilizes the multiple system resources, and the overhead required to support such a scheme is extremely low. The resulting performance gains show a decided advantage over the current approaches such as round-robin.

In the future we plan to migrate the server into the production environment as the primary server for the Alexandria Digital Library project. We feel this will introduce a large number of interesting research problems relating to scheduling wildly varying demands on a heterogeneous system with nonuniform load distributions. As part of this issue, many of the Alexandria-specific functions need to be characterized in terms of cpu, disk, and network demands. We are also incorporating the pre-forking strategy [NCH95] into our scheme to further reduce the server overhead.

Acknowledgments

This work was supported in part by funding from NSF, ARPA, and NASA under NSF IRI94-11330 and NSF CCR-9409695, and a startup fund from University of California at Santa Barbara. We would like to thank Omer Egecioglu, Terry Smith, Chris Fischer, Cong Fu and the Alexandria Digital Library team for many valuable discussions and suggestions. We would also like to acknowledge the student researchers who contributed to this project, including Dona Agustin and Hongbing Wang.

References

[A95] 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.

[BCS95] J. Baumgartner, D. J. Cook, and B. Shirazi, Genetic solutions to the load balancing problem. Proc of ICPP 95 Workshop, pp 72-78.

[BD94] C. M. Bowman, P. B. Danzig, D. Hardy, U. Manber and M Schwartz, The Harvest Information Discovery and Access System. Proceedings of the Second International World Wide Web Conference, pp. 763-771, Chicago, Illinois, October 1994.

[APA95] Apache HTTP Server Project, *http://www.apache.org/index.html*, June 26, 1995.

[BC95] T. Bemers-Lee and D. Connolly, Hypertext Markup Language - 2.0, *http://www.w3.org/hypertext/WWW/MarkUp/html-spec/html-spec_toc.html*, June 16, 1995.

[BL95] T. Bemers-Lee, Uniform Resource Locators, *http://www.w3.org/hypertext/WWW/Addressing/URL/*, 1995.

[ELZ86] D. L. Eager, E. D. Lazowska, and J. Zahojan, Adaptive load sharing in homogeneous distributed systems, IEEE Trans on Software Engineering, 12(5):662-675. 1986.

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

[GM95] J. Gosling and H. McGilton, The Java Language Environment: A White Paper, *http://java.sun.com/*, 1995.

[HL95] J. Hsieh, M. Lin, J. Liu, T. Ruwart, and D. H.C. Du, Performance of A Mass Storage System for Video-On-Demand , Submitted to Journal of Parallel and Distributed Computing, Special issue on Multimedia Processing and Technology.

[HT95] Hypertext Transfer Protocol(HTTP): A protocol for networked information, $http://www.w3.org/hypertext/WWW/Protocols/HTTP/HTTP2.html$, June 26, 1995.

[LYC95] Lycos Usage: Accesses per Day, $http://lycos.cs.cmu.edu/usage-day.html$, June 17, 1995.

[KB94] 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.

[MC95] R. McGrath, Performance of Several HTTP Demons on an HP 735 Workstation, $http://hoohoo.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html$, April 25, 1995.

[ME94] Meiko Corp., Computing Surface CS-2 Product Description, Meiko, Waltham, MA, 1994

[NC95] NCSA development team, The Common Gateway Interface, $http://hoohoo.ncsa.uiuc.edu/cgi/$, June, 1995.

[NCH95] NCSA development team, The NCSA HTTPd Home Page, $http://hoohoo.ncsa.uiuc.edu/$, June 13, 1995.

[SS95] M.G. Siriram and Mukesh Singhai, Measures of the Potential for Load Sharing in Distributed Computing Systems, *IEEE Trans. on Software Engineering* vol. 21, no. 5, May, 1995, pp. 468-475.

[ST90] W. R. Stevens, **UNIX Network Programming**, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[SU95] W. Shu and M. Y. Wu, An incremental parallel scheduling approach to solving dynamic and irregular problems. Proceedings of Inter. Conf. on Parallel Processing, 1995. II. 143-150.

[WR93] M. Willebeek-LeMair and A. Reeves, Strategies for Dynamic Load Balancing on Highly Parallel Computers, *IEEE Trans. on Parallel and Distributed Systems,* vol. 4, no. 9, September, 1993, pp. 979-993.

[JX93] J. Ju, G. Xu, and K. Yang, An Intelligent Dynamic Load Balancer for Workstation Clusters *IEEE Trans. on Software Engineering* vol. 21, no. 5, May, 1995.

[MZ95] F.J. Muniz and E.J. Zaluska Parallel Load-balancing: An extension to the gradient model, *Parallel Computing,*, vol 21, 1995, pp. 287-301.

[WS94] G. Wright and W. R. Stevens, **TCP/IP Illustrated, Volume 2**, Addison-Wesley, Reading, MA, 1994.