Implementing Distributed Server Groups for the World Wide Web

Michael Garland, Sebastian Grassia, Robert Monroe, Siddhartha Puri

25 January 1995
CMU-CS-95-114

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

The World Wide Web (WWW) has recently become a very popular facility for the dissemination of information. As a result of this popularity, it is experiencing rapidly increasing traffic load. Single machine servers cannot keep pace with the ever greater load being placed upon them. To alleviate this problem, we have implemented a distributed Web server group. The server group can effectively balance request load amongst its members (within about 10% of optimal), and client response time is no worse than in the single server case. Client response time was not improved because the measured client traffic consumed all available network throughput. The distributed operation of the server groups is completely transparent to standard Web clients.

This research is sponsored in part by the Wright Laboratory, Aeronautical SystemsCenter, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Support also came from the Air Force Materiel Command under contract number F19628-93-C-0171. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory or the United States Government.

Keywords: World Wide Web, distributed computing, hypertext transfer protocol (HTTP), load balancing, server groups, distributed information systems.

1. Introduction

As the exponential growth of the World Wide Web continues with no near-term end in sight, the load placed on popular Web servers is often too great for a single server machine to handle. When the volume of requests to a server becomes prohibitively high the server tends to drop requests and eventually crashes. The service provider needs more computing power to meet the demands of the clients requesting the service. One solution is to simply buy a bigger machine that can handle the large load of service requests. This solution is, however, unacceptable; it provides only temporary relief from server overload. If server usage continues to grow rapidly, even the more powerful server will eventually be overwhelmed. This problem is likely to become more significant over the next few years as the number of commercial service providers and the size of the data objects being exchanged are likely to increase dramatically.

A fairly obvious solution to this problem is to implement a distributed Web server which will spread incoming request load among several machines. There is, however, a significant problem with this approach. The HyperText Transfer Protocol (HTTP/1.0) used by Web clients and servers identifies a resource on the Web using a valid hostname. Consequently, a distributed server needs to present a single hostname to maintain compatibility with existing Web clients. We can alleviate this problem by extending the concept of distributed process groups to the Web server level. A distributed server group exports a single logical name and address to the outside world. From the viewpoint of a client sending an HTTP request, the server group is a single server that will handle the HTTP request in the same way that any other Web server would. Once the server group receives a request it transparently allocates it to an appropriate member of the group for processing.

By supporting distributed Web server groups, we can provide a mechanism for smoothly scaling the server capacity of Web service providers. The use of server groups can reduce the latency of servicing a request provided there is sufficient available network bandwidth to accommodate higher request throughput. Our Web server group implementation can effectively balance request load, and is transparent to Web clients which support the HTTP/1.0 protocol.

2. The World Wide Web

2.1. The HTTP Protocol

The fundamental mechanism underlying the Web is the HTTP protocol. The protocol is stateless, object-oriented, and textual. A complete description of the current protocol can be found on-line at [CERN]. For our purposes, we are only interested in the basic structure of the protocol.

HTTP is an RPC-like protocol; clients make requests of servers and receive responses. The general format of a request is:

method	URL	protocol version	**CrLf**
optional header information			**CrLf**
Body			

Figure 1: HTTP request format.

```
GET /doc.html HTTP/1.0 <CrLf>
<CrLf>
<end>
```

Figure 2: Example HTTP request.

This figure needs some explanation. A *URL* is simply a name for an object; typically, the hierarchical naming scheme of the Unix file system is used. The *method* given in the request is a method to be invoked on the object named by the given URL. The **CrLf** tag represent the pair of characters carriage return and line feed. The *optional header information* is a series of MIME headers which can be used to specify a variety of things including client name, user name, encoding type, date, authorization information, etc. The *body* of the request is interpreted by the method which is invoked; it is in essence the argument list for the method.

Once the server receives a request from a client, it processes the request and returns a response. A reply from the server has the following structure:

protocol version	status code	reason	**CrLf**
optional header information			**CrLf**
Body			

Figure 3: HTTP response format.

```
HTTP/1.0 200 OK <CrLf>
Server: NCSA/1.3
Content-type: text/html <CrLf>
HTML document
<end>
```

Figure 4: Example HTTP response.

The *status code* indicates the result of the request; the *reason* field provides a human-readable textual explanation of the status code. The rest of the fields of the response fulfill similar roles to their counterparts in the request.

In current patterns of Web use, the `GET` method is by far the most frequently used. In fact, Web sessions typically consist exclusively of `GET`s. This reflects the fact that the Web is currently used primarily as a means of fetching files, albeit in many formats, including hypertext.

2.2. Web Servers and Clients

Servers and clients use the HTTP protocol described above to exchange information. The general model of the Web currently in use is this: servers export a hierarchical file space to clients. This file space is populated by “documents” of various types. The primary function of Web clients is to acquire documents from this file space in response to user actions. The client and server must also negotiate a document format which is acceptable to them both. Web servers and clients communicate with each other over TCP/IP streams. The general model of an HTTP transaction is:

- The client opens a TCP/IP connection to the server,
- The client sends its request over this stream,
- The server sends its response back,
- And the connection is closed.

In the case of the most widely used Web server (NCSA’s `httpd`), the server forks a new process for each connection. That process receives the request from the client, processes it, sends back a response, and exits.

3. A Distributed Web Server Group

Since it’s inception, the Web has experienced very rapid growth in usage and amount of data being provided. The growth of the user community has generated increasing demand for data, and the raw amount of data has also increased due to the proliferation of large multimedia resources. The result has been continually increasing stress on Web servers.

A natural solution is to group several server machines into a single server group. The server group would provide a single logical service but would physically share incoming requests amongst its member machines. The need for this is indeed real; NCSA has implemented a similar scheme for their server after a single server machine was no longer able to cope with the heavy load [KBM94].

In order for a Web server group to be useful it must operate transparently within the existing Web. In other words, it

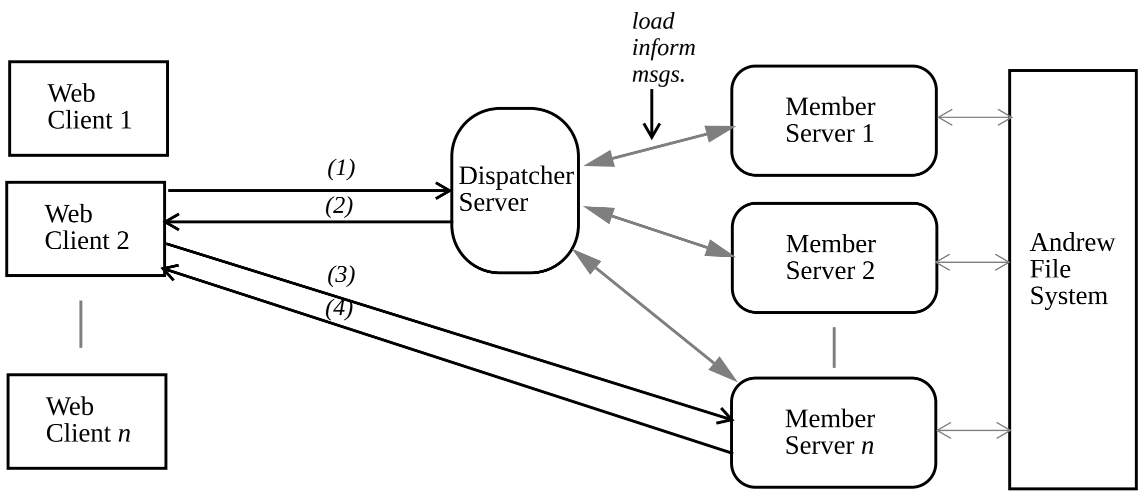


Figure 5: Architecture and Protocol for the Distributed Web Server Group. (1) Client sends an HTTP request. (2) Dispatcher calculates least-loaded server and sends forwarding address. (3) Client resends HTTP request to forwarding address (member server). (4) Member server satisfies request.

must be implemented within the framework of the HTTP protocol and it must be completely compatible with existing Web clients. The server group implementation is also constrained by the current URL specification. A server group must be identified by a unique Internet address. In addition to all this, we also want the server group to perform well: it must balance request load effectively and introduce as little new overhead as possible.

3.1. System Architecture

We have chosen the following general architecture for our Web server groups:

- There is a special server (the *dispatcher*) which coordinates and controls the server group. Requests intended for the server group are received by the dispatcher.
- Attached to the dispatcher is a collection of *member* servers. These servers are the actual repository of data and fulfill requests made of the group.

The dispatcher is the single entity representing the group to the outside world. Incoming requests are received by the dispatcher, and they are redirected to one of the member servers for actual processing. It is also the task of the dispatcher to balance the load placed upon the various members under its control. In essence, it tries to always dispatch incoming requests to the least loaded member. Each member exports an identical copy of some data repository. In our case, this data is shared via a common AFS directory. Figure 5 diagrams both the system architecture and communication protocol.

We have implemented our new servers as extensions to the NCSA Web server (`httpd`). This server has several attractive properties: it is well supported, it is in wide use, and it is quite portable. One major consequence of this choice has affected our design; since the `httpd` server forks a new process for every connection, a process handling an incoming request cannot modify the global server state.

We have decided that the dispatcher should not fork on connection like the standard `httpd` server; it only forks when sending messages to its members. This lowers the latency of request dispatching and allows the global server state to be modified during request processing. Throughout the design process we felt that the dispatcher was likely to be a bottleneck, since it is a single point through which all requests flow. Processing each request serially before accepting the next connection introduced an even greater danger of creating a bottleneck. Therefore we have tried to keep the amount of processing performed by the dispatcher to a minimum. Fortunately, in our experience the load on the dispatcher never reached a critical point. It remained only marginally loaded.

3.2. Interfacing to Web Clients

In order to make the server group work, there needs to be some mechanism for the dispatcher to redirect a client's request to another server. Fortunately, such a mechanism already exists and is supported by all major Web clients (including Mosaic, Lynx, and W3).

Recall that every response from a server to a client includes a status code. The status code of particular interest to us is: `Found (302)`. It instructs a client that the requested object has been found on another server, whose address is included in the body of the response. The client is expected to re-request that object from the given alternative server. This process of redirection is completely invisible to the user.

3.3. Protocol Extensions

To support server groups, we needed some means for communication between the dispatcher and its members. One possibility would be to build a dedicated protocol for this purpose, possibly using UDP instead of TCP. However, we decided to implement this communication through extensions to the HTTP protocol. As a result, the intragroup communication fits smoothly within the overall structure of the server. This has a number of advantages; the chief advantage being that our code should be easily portable to future server versions as well as other servers.

The extensions we have made to the HTTP protocol are fairly minor. Most of these changes involve simply adding new methods, and they only affect the dialog between the dispatcher and its members. These extensions are completely invisible to clients. The first set of extensions involves adding methods for administering server groups. In these cases, the dispatcher is acting as a server and a member is communicating with it in the role of a client. The new methods are as follows:

Method: GROUP-ADD
Body: URL of a member server
This method adds the specified member server to the server group administered by the dispatcher.

Method: GROUP-EXIT
Body: URL of a member server
This method removes the specified member server from the dispatcher's server group.

Method: GROUP-QUERY
Body: None
Returns status information about the server group and its members. Currently this is meant only for debugging purposes. A real system might choose to ignore this request.

The next extension actually diverges from the HTTP specification. It is implemented as a *message* rather than as a request. A response is neither required nor expected. The format of the message is:

Method: LOAD-INFORM
Body: Load value and time stamp
This message communicates load information from a member to the dispatcher. This information is then used by the dispatcher in its load balancing decisions.

Our final extension is also a message rather than a request.

Method: LOAD-QUERY
Body: None
This message is used by the dispatcher to query load information from a member. The member is expected to respond later with a LOAD-INFORM message.

This last requirement may seem slightly strange; the member is required to respond to a LOAD-QUERY message with another message rather than a direct response. The reason for this is related to the structure of the dispatcher. Since the dispatcher forks when sending messages, any response would be unable to affect the global state. To get around this difficulty, we require the member to send a LOAD-INFORM back to the parent process.

3.4. Load Balancing

The task of the dispatcher is to distribute incoming requests to members of the server group. However, simply doing this without making informed decisions about *where* to send requests would not be very desirable. We want the dispatcher to distribute requests in such a way as to balance the load placed on the various member servers. This requires two things: the dispatcher must have some knowledge of the load on members, and there must be some scheduling policy used to select which member to dispatch the next request to.

The first issue that needs to be addressed is how to measure the load on a member server. At first glance, you might suspect that the dispatcher should have enough information to compute the load on a member. After all, every request that arrives at a member was directed there by the dispatcher. In reality, this is not true. When examining actual HTML documents it becomes evident that many links are either relative to the current server or point to a remote server. This has the following consequence: when first contacting a server group, a client will be dispatched to a particular member. Subsequently, all relative links will be resolved by going *directly* to that member. This property actually has the desirable effect of helping to balance the load [CHE91]. In addition, since each client connection is for a single request, there is no way of predicting how long a particular client will continue making requests of the server. The dispatcher cannot know the load on a member, so it must be informed of the member's load (using the LOAD-INFORM message described above).

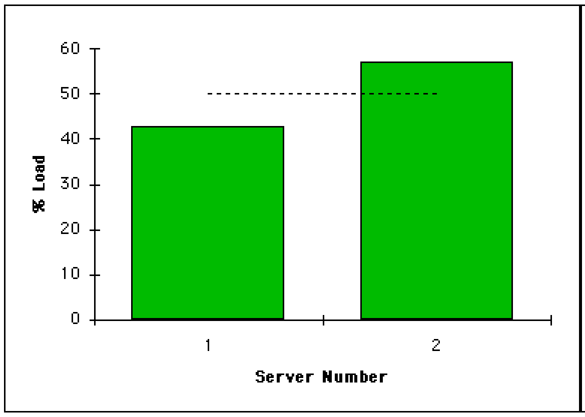
A good load measure for the members should account for two sources of load: the traffic of requests and the general load on the machine. Obviously, request traffic is an important source of load. Including the load on the machine as well is necessary because there might easily be other active processes on the same machine. In particular, there might be multiple members per machine. The formula we use to compute the load on a member is $Load = 10\,(r + m)$ where r is the number of requests per second being received by the member and m is the machine load as measured by the number of processes in the run queue. The scaling factor of 10 is used to allow the load to be represented as an integer, for faster processing. Note that with our servers, there is a direct correlation between requests being processed and number of processes. This load information is transmitted periodically to the dispatcher (every 15 sec. in our experimental configuration).

The dispatcher uses a simple scheduling policy. It keeps all the members in a doubly-linked priority queue, sorted by increasing load. Thus, dispatching a request simply involves looking at the head of the queue and returning that member address to the requesting client. The dispatcher also attempts to estimate the change of member load over time by adding a constant factor to the load every time a request is dispatched to that member. If the dispatcher does not receive a LOAD-INFORM query from a member after some long amount of time, it will send a LOAD-QUERY message to it. If this fails for some reason, the member is assumed to have failed and is removed from the server pool.

4. Performance Analysis

4.1. Experimental Overview:

The most appropriate test for our distributed Web server would involve observing its performance in a heavily accessed location over a period of months. However, practical considerations obviously preclude this type of testing from the scope of this project. Instead, we have attempted to simulate heavy (localized) load on the servers and observe how they perform under a variety of file access patterns. We have attempted to use file access patterns which we feel are fairly representative of common Web usage patterns. Our testing has focused on the latency for satisfying



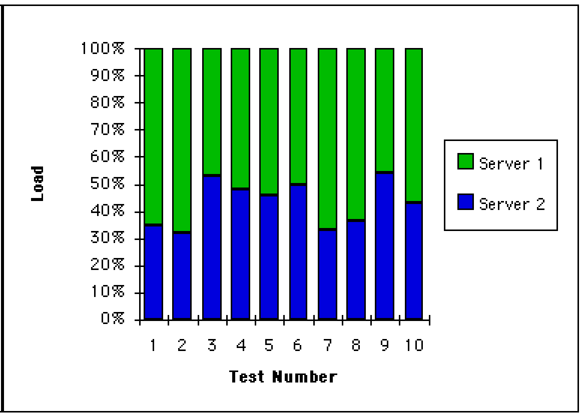


Figure 6: Total Load Balance (2 servers)

Figure 7: Instantaneous Load Balance

a request (from the user's perspective), and on the balancing of load amongst the group members. These metrics reflect the perception of a user browsing the Web, and server hardware utilization, respectively.

4.2. Experimental Procedure:

We developed a test suite of 125 tests to observe how the server groups performed as various aspects of the system were stressed. As a control test, we performed each of the test series on a vanilla `httpd` server. To measure server performance, the member servers were instrumented to periodically dump their load information into log files.

To measure request latency as seen by clients, we constructed a programmable HTTP client (using Tcl) that measured the time required to satisfy an HTTP request. The phases timed by the client were: connection to the dispatcher, redirecting to the member server, and finally receiving the requested data. All timing measurements were made in C code to avoid Tcl overhead, and the data read by the client was not interpreted in any way. Many machines were required to test system performance; the final test suite was performed in Carnegie Mellon's computing labs early in the morning over Thanksgiving break, 1994. The campus network was very lightly loaded. We used 10 Sun Sparc-5 machines as clients and 5 SGI Indigo2 machines as servers. The entire test suite took about five hours to complete. External load on the network remained consistently negligible, but present, throughout the duration of the test.

4.3. Test Cases:

The three parameters which were varied during our tests were: the number of member servers, the number of concurrent clients, and the file access pattern. Specifically, we looked at 1 vanilla `httpd` server, 1, 2, and 4 member servers, 1, 5, and 10 clients, and ten different file access patterns. The file access patterns were designed to simulate common Web access patterns. All of the file access patterns were some combination of small (4KB), medium (100KB), and large (2MB) files. These file sizes were designed to represent a small HTML home page, a medium JPEG image, and a short MPEG video clip, respectively. We ran test cases for each possible combination of server count, client count, and file access pattern.

4.4. Experimental Results

Server Performance Observations: The long term load balancing performance of the servers is displayed in Figures6 and 8. As you can see, the dispatcher was able to maintain reasonable load balancing among the member servers. The dashed line on each graph indicates the optimal case. As the graphs show, the actual load observed was within 7% of optimal with 2 servers and within 9% of optimal with 4 servers. We consider this performance to be quite acceptable. Figure7 shows a more fine grained load balancing analysis over ten consecutive test cases with 2

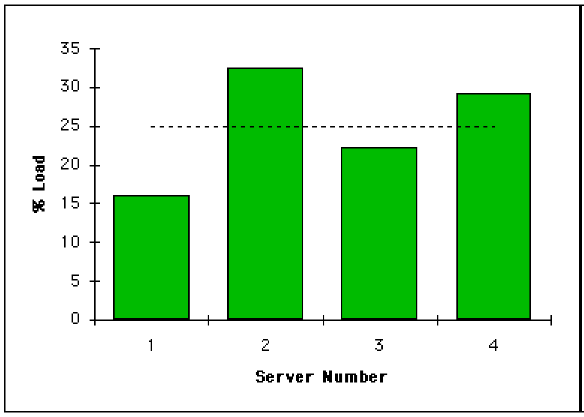


Figure 8: Total Load Balance (4 servers)

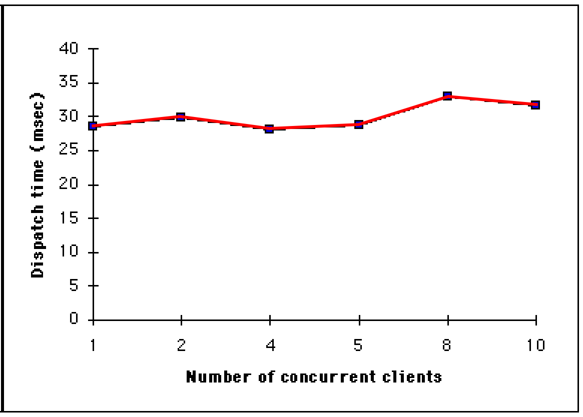


Figure 9: Dispatch time with increasing client load.

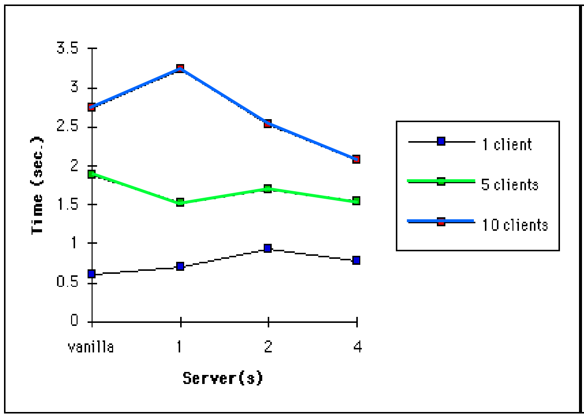


Figure 10: Time to access 10 4K files.

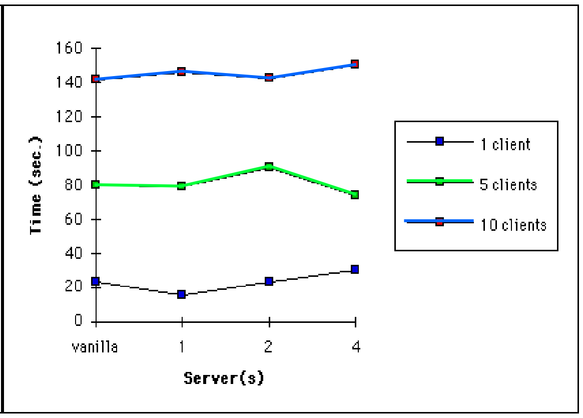


Figure 11: Time to access 5 2Mb files.

servers. As you can see, the instantaneous load balance varies somewhat. This could be improved by reducing the granularity of LOAD-INFORM messages from the members, but this would probably lower total throughput. It is practically impossible to ensure optimal load balancing. Once a client has a member's address, it can contact that member any number of times; these accesses cannot possibly be accurately predicted by the servers.

User Performance Perception: Unfortunately, the performance witnessed by the clients was not as promising as the load balancing results. While conducting the tests it became rapidly apparent that the bandwidth of the Ethernet was insufficient to keep up with even a small number of clients and servers, let alone a barrage from four servers and ten clients. Throughout the tests, it was not uncommon for the number of Ethernet collisions in the server pool to exceed the number of successful outgoing packets. As a result, the client latency figures are fairly erratic, and show only a slight improvement as servers are added. Figures10 and 11 show request latency for very small and very large file access patterns. The curves are somewhat erratic. Adding servers to the group did not generally improve request latency, but it was also generally no worse than that of the stand-alone vanilla server. It would certainly seem that in our experimental setup, the network, and not the server machines, was the chief limiting factor on performance.

Dispatcher Server Performance: One of the promising results of our experiments was the performance of the dispatcher. We had anticipated that the dispatcher would be the primary bottleneck in our experiments; however, it turned out to be remarkably fast and scalable. We attempted to determine where the dispatcher would break by having 1 to 10 clients send a barrage of requests to the dispatcher; the forwarding responses were merely discarded. With the same 10 client and 4 server configuration that rapidly saturated the Ethernet when data was actually transferred, we were unable to force the dispatcher to refuse any connections or dramatically increase the time required to process a request. As Figure 8 shows, the time required to process a request stays fairly constant at around 30 milliseconds, even as the number of clients is increased from 1 to 10. This was a surprising result. With the 10 client and 4 server machines we had at our disposal, we were unable to significantly degrade the dispatcher's performance. Furthermore, the 30 msec. overhead induced by the redirection is small enough to be imperceptible to a human user. Obviously, the

30 msec. delay is very sensitive to network distance travelled, but the results paint a very encouraging picture of the general cost of dispatching.

5. Related Work

Distributed process groups have been explored in the ISIS system [BIR91] and the V system [CHE85]. These two systems implement slightly different forms of process groups, neither of which are sufficient to completely solve our problem. However, they do establish the basic concepts of distributed process groups, and hence distributed server groups.

Load balancing methods can be broken down into two main categories: those based on communication and those based on dispatching patterns [HSU86]. In the first category, a group of equal servers dispatches jobs to its members, trying to optimize (or at least improve) some performance criteria. This requires that each server be kept informed of the load on every other server. The second approach views the problem as one of routing, where a central dispatcher routes all jobs to servers based on a routing policy which may involve optimization. The critical difference for our purposes is that in this scheme the servers whose loads we are trying to balance need to communicate only with the dispatcher. Schemes for dynamically scheduling requests to balance load have proven successful in several application areas similar to our own [ALO87, WIK91, HSU86, PAR92].

The NCSA has also implemented a distributed Web server mechanism [NCSA]. Their server, which is probably the most heavily loaded server on the Web, was beginning to experience actual system failures due to extreme load. They report that the system was incapable of handling more than 20 requests per second without crashing. To solve this problem, they extended the BIND name resolution server at their site. It now allows them to specify multiple physical machines for the same logical name. When HTTP connections are requested on the logical name, the BIND server resolves that name to one of the physical machines. Machines are scheduled in a purely round-robin fashion. They report that load was not very well balanced (1 of their 4 machines consistently got 50% of the load), but it did solve their problem of server failure.

6. Future Directions

There are several ways in which our work could be extended in the future. To support active use in the Web at large, there are a few features which would make our servers more complete. As with most experimental systems, our code is not in a state acceptable for widespread use. There are also areas in which our servers could be extended or altered to improve functionality and performance.

As they are currently implemented, our servers implement the basic server group functionality. However, there are several extensions that could be made. The GROUP-ADD and GROUP-EXIT requests could be extended to allow multiple servers to be added/removed in a single request. We could also extend the dispatcher to support anonymous members. It is conceivable that some servers might not want to advertise their location to the rest of the world. In this case, the dispatcher could serve as an intermediary between the client and the anonymous server. A request for anonymity would be made in the GROUP-ADD request. Currently, every member exports identical data. It would be nice to allow specific members to export data objects unique to themselves. This would complicate the dispatcher in that it would have to actually interpret the URLs handed to it, look them up in a table, and determine whether they had a special location or could be dispatched to anyone. With the current, non-forking, architecture of the dispatcher, one would need to carefully consider whether this name resolution would add unacceptable overhead.

This point highlights what is probably the most important improvement which should be made to our servers, and indeed to all Web servers. They should be multithreaded. Servers currently pay a high price for accepting a connection, a price particularly noticeable during small requests. In addition, the server process interacting with the client cannot modify the global server state. A multithreaded server should perform significantly better, and obviate the contortions one must go through to maintain server global state.

An important area of inquiry which was beyond the scope of our current project, but which could be very informative, is the long term measurement of server group performance under actual use. Although we have tried to examine "reasonable" server usage, our experiments have used inherently artificial access patterns. It is nontrivial to predict just how our measured performance would translate into actual performance. With our fairly limited time and performance data, we were unable to seriously explore variations in the actual dispatching policy. Such inquiry might lead to significant performance improvements.

Our tests show that the constraints of network bandwidth overshadow the constraints of CPU load for our distributed server. This suggests that we could incorporate more intelligent request processing into our servers, and the added computation would not impair performance. We could also allow each server to accept a higher CPU load by providing more intensive services such as numerical computation or database lookup, instead of simply sending files. In this case we would expect our distributed servers to provide a significant performance enhancement.

The most obvious way to avoid a network bottleneck is to distribute the servers geographically so that they reside on separate networks. The dispatcher would then take network load and proximity into account when deciding which server to transfer a client to. This gives rise to the paradigm of "transparent mirrors," which could be seen as extending the current practice of geographically distributed mirror sites by providing a transparent centralized directory service. The most difficult problem that this approach raises is determining proximity. While some way of calculating geographical distance may be a good heuristic, what we would really like is a distance metric based on network topology. Finding a good way for the dispatcher to dynamically keep track of the topology of the internet and the probable available bandwidth at each link is wide open for further study.

7. Conclusion

The increasing load placed upon Web servers is clearly a serious problem. The NCSA has already become so overloaded that they were forced to deploy a distributed server mechanism. While their method works, it is not particularly clean, and it does not balance request load very well. Server groups provide a much nicer abstraction for distributed Web servers, and they support a wider range of possible uses. With our current architecture, server groups could also be deployed without any changes to system software as required by the NCSA approach.

Our implementation demonstrates that server groups are both feasible and effective. Server group functionality can be added to existing HTTP servers without modifying the overall structure of the servers. Our servers have also proven capable of distributing request load fairly amongst group members. For all this, the server group does not incur any significant cost in client response time.

8. Bibliography

Process groups:

[BIR91] Birman, Kenneth; The Process Group Approach to Reliable Distributed Computing; *Communications of the ACM*, December 1993, pp. 36-53.

[CHE85] Cheriton, David R., and Zwaenepoel, Willy; Distributed Process Groups in the V Kernel; *ACM Transactions on Computer Systems*, May 1985, pp 77-107.

Load Balancing:

[ALO87] Alonso, Rafael; An Experimental Evaluation of Load Balancing Strategies; Technical Report CS-TR112-87, Department of Computer Science, Princeton University.

[CHE91] Cheng, William C., Muntz, Richard R.; Optimal routing for closed queueing networks. *Performance*

Evaluation, vol. 13, no.1; Sept. 1991 pp. 3-15.

[HSU86] Hsu, Chi-Yin Huang, Liu, Jane W.S.; Dynamic Load Balancing Algorithms in Homogenous Distributed Systems; Technical Report UIUCDCS-R-86-1261, Department of Computer Science, U of Illinois, Urbana-Champaign.

[LIN91] Lin, H.-C., Raghavendra, C.S.; A dynamic load balancing policy with a central job dispatcher. Proceedings of 11th Intl. Conf. on Distributed Computing Systems; May 1991; pp. 264-271.

[PAR92] Parris, Colin, Zhang, Hui, Ferrari, Domenico; A Mechanism for Dynamic Re-routing of Real-time Channels; Technical TR-92-053, Computer Science Division, UC Berkeley.

[WIK91] Wikstrom, Milton C. et al; A Meta-Balancer for Dynamic Load Balancers; Technical Report TR#91-04, Department of Comuter Science, Iowa State University.

HTTP and Distributed Web Servers:

[CERN] HTTP specification -- working document specifying the hypertext transfer protocol (HTTP) 1.0, available on the WWW at the URL *"http://info.cern.ch/"*.

[KBM94] Katz, Butler, McGrath: "A Scalable HTTP Server: The NCSA Prototype" - working document from NCSA. Available on the WWW at the URL *"http://www.elsevier.nl/cgi-bin/query/WWW94/FinalProgramme.html?katz"*.