Administering Very High Volume Internet Services

Dan Mosedale, William Foss, and Rob McCool
Netscape Communications

Administering Very High Volume Internet Services

Dan Mosedale, William Foss, and Rob McCool – Netscape Communications

ABSTRACT

Providing WWW or FTP service on a small scale is already a well-solved problem. Scaling this to work at a site that accepts millions of connections per day, however, can easily push multiple machines and networks to the bleeding edge. In this paper, we give concrete configuration techniques that have helped us get the best possible performance out of server resources. Our analysis is mostly centered on WWW service, but much of the information applies equally well to FTP service. Additionally we discuss some of the tools that we use for day-to-day management.

We don't have a lot of specific statistics about exactly how much each configuration change helped us. Rather, this paper represents our many iterations through the "watch the load increase; see various failures; fix what's broken" loop. The intent is to help the reader configure a high-performance, manageable server from the start, and then to supply ideas about what to look for when it becomes overloaded.

Our Site

Netscape Communications runs what we believe to be one of the highest volume web services on the Internet. Our machines currently take a total of between six and eight million HTTP hits per day, and this number continues to grow. Furthermore, we make the Netscape Navigator, our web browser, available for downloading via FTP and HTTP[1].

The web site[2] contains online documentation for the Netscape Navigator, sales and marketing information about our entire product line, many general interest pages including various directory services, as well as home pages for Netscape employees. All of the machines run the Netscape Server, but most of the strategies in this paper should apply to other HTTP and even FTP servers also.

At various times, we have tried out various configurations of machines running the given operating systems; Figure 1 shows a list.

Each of our WWW servers has an identical content tree uploaded to it (more on this later). Before the Netscape Navigator was released to the Internet community for the first time, we thought about the web pages we intended to serve and debated how we would spread the load across multiple machines when that became necessary. Because of problems reported using DNS round-robin techniques[3], we chose to instead implement a randomization scheme inside of the Netscape Navigator itself. In short, when a copy of the Navigator is accessing home.mcom.com or home.netscape.com, it periodically queries the DNS for a hostname of the form homeX.netscape.com, where X is a random number between 1 and 16. Each of our web servers has a number of the homeX aliases pointing to it. Since this strategy is not something that will be available to most sites, we won't spend more time on it here.

Another scheme which we have looked into is nameserver-based load balancing. This scheme depends upon a nameserver periodically polling each content server at site to find out how loaded they are (though a less functional version could simply use static weighting). The nameserver then resolves the domain name to the IP address of the server which currently has the least load. This has the added benefit of not sending requests to a machine that is overloaded or unavailable, effectively creating a poor man's failover system. It can, however, leave a

Sun uniprocessor (60 Mhz) SPARCserver 20	Solaris 2.3 & 2.4
SGI 2-processor Challenge-L	IRIX 5.2
SGI Indy (150 Mhz)	IRIX 5.2 & 5.3
SGI Challenge-S (175 Mhz)	IRIX 5.3
HP 9000/816	HP-UX 9.04
90 Mhz Pentium PC	Windows NT 3.5
90 Mhz Pentium PC	BSD/OS 2.0

Figure 1: Tested hardware/software configurations

dead machine in the server pool for as long a duration as the DNS TTL. Two DNS load balancing schemes that we plan to investigate further are RFC 1794[4] and lbnamed[5].

Figures 2 and 3 summarize some of the performance statistics for the various servers on July 26, 1995 along with their assigned relative load. Our setup allows us to give each machine a fraction of the total traffic to our site measured in sixteenths of the total.

Note: In addition to assuming 1/16 of the WWW load, the 150 MHz Indy also wears the aliases www.netscape.com, home.netscape.com, and currently runs all CGI processes for the entire site. Total CGI load for this day accounted for 49,887 of its 1,548,859 HTTP accesses (more on this later).

The Tools

The traffic and content size of our web site grew very quickly, and we collected a number of tools to help us manage the machines and content. We used a number of existing programs, some of which we enhanced, and developed a few internally as well. We'll go over many of these here; the intent is to focus on tools that are directly useful in managing web servers and content. We will avoid discussing HTML authoring programs and utilities; for those who are interested, see http://home.netscape.com/home/how-to-create-web-services.html for pointers to many such tools.

Document Control: CVS

Since our content comes from several different sources within the company, we chose to use CVS[6] to manage the document tree. This has worked moderately well, but is not an ideal solution for our environment.

In some some ways, the creation of content resembles a mid-to-large programming environment. A document revision system became necessary to govern the creation of our web site content as multiple "contributing editors" added and deleted material from the content tree. CVS provided a reasonably easy method to retrieve older source or detailed logs of changes made to HTML dating back to the creation of the content tree.

One drawback of CVS is that many of the folks who design our content found it difficult to use and understand due to a lack of experience with UNIX. A cross-platform GUI-based tool would be especially well-suited to this market niche.

Host type	Load Fraction	Hits	Redirects	Server Errors	# unique URL's	Unique hosts	KB transferred
SGI 150Mhz R4400 Indy	1/16	1548859	68962	7253	4558	120712	12487021
SGI 175 Mhz R4400 Challenge S	6/16	2306930	47154	23	2722	249791	11574007
SGI 175 Mhz R4400 Challenge S	5/16	2059499	43441	43	2681	225055	10626111
BSD/OS 90 Mhz Pentium	4/16	1571804	31919	23	2351	192726	7936917

Figure 2: WWW Server activity for the period between 25/Jul/1995:23:58:04 and 26/Jul/1995:23:58:59

Host type	Load Fraction	Bytes/Hits	Bytes/Hits	Bytes/Hits	Bytes/Hits	Bytes/Hits
Host type SGI 150 Mhz R4400 Indy	1/16	430600/82	345132/61	227618/61	224849/60	236678/59
Host type SGI 175 Mhz R4400 Challenge S	6/16	613621/128	646112/119	656412/110	545699/108	520256/107
Host type SGI 175 Mhz R4400 Challenge S	5/16	466244/93	430870/88	358186/84	375964/84	531421/82
Host type BSD/OS 90 Mhz Pentium	4/16	375696/81	256143/77	417655/76	394878/72	298561/70

Figure 3: Five busiest minutes for the tested hosts

Content Push

Once we had multiple machines serving our WWW content, it became necessary to come up with a reasonable mechanism for getting copies of our master content tree to all of the servers outside our firewall. NCSA distributes their documents among server machines by keeping their content tree on the AFS distributed filesystem[3].

It seemed to us that another natural solution to this problem was rdist[7], a program specifically designed for keeping trees of files in sync with a master copy. However, we felt we couldn't use this unmodified, as its security depended entirely on a .rhosts file, which is a notoriously thin layer of protection. With the help of some other developers, we worked on incorporating SSL[8] into rdist in order to provide for encryption as well as better authentication of both ends. With SSL, we no longer need to rely on a client's IP address for its identity; cryptographic certificates provide that authentication instead.

In the development of our SSLified rdist, we decided that it would be a good idea to use the latest rdist from USC, in part because it has the option of using *rsh*(1) for its transport rather than *rcmd*(3). Because it doesn't use *rcmd*(3), it no longer needs to be setuid root, which is a real security win. One side effect of this is that we now have an SSLified version of *rsh*(1), which we use to copy log files from our servers back to our internal nets.

Monitoring

During the course of our server growth, we wrote and/or borrowed a number of tools for monitoring our web servers. These include a couple of tools to check response time, a log analyzer, and a program to page us if one of the servers goes down.

The tool to check response time is designed to be run from a machine external to the server being monitored. Every so often, it wakes up and sends a request to an HTTP server for a typical document, such as the home page. It measures the amount of time that it took from start to finish; that is, from before it calls connect() to after it gets a zero from read() indicating that the server has closed the connection. If you choose a relatively small document, this time can give you a good general indication of how long people are unnecessarily waiting for documents (since under ideal conditions a small document should come back nearly instantaneously). In our typical monitoring setup, we run monitor programs from remote, well-connected sites as well as from locally-networked machines. This allows us to see when problems are a result of network congestion as opposed to lossage on the server machines.

The logfile analyzer, which is now a standard part of the Netscape Communications and Commerce server products, provides information about the busiest hours or minutes of the day, and about how much data transfer client document caching has saved our site. The analyzer can be very helpful in determining which hours are peak hours and will require the most attention. Because of the high volume of traffic at our site, we designed it to process large log files quickly.

The program to page us when a server becomes unreachable is similar to our response time program. The difference is that when it finds that a server does not respond within a reasonable time frame for three consecutive tries, it sends an e-mail message to us along with a message to our alphanumeric-pager gateway to make sure we know that a server needs attention.

At times when we knew that we wouldn't be able to come in and reboot the server, we used a UNIX box with an RS232-controlled on/off switch to automatically hard boot any system not responding to three sequential GET requests. A small PC with two serial ports is enough to individually monitor 10 systems and provides recovery for most non- fatal system errors (e.g., most problems other than hardware failure or logfile partitions filling up).

Performance

Previous works [9, 10, 11] have explored HTTP performance and have come to the conclusion that HTTP in its current form and TCP are particularly ill-suited to one another when it comes to performance. The authors of some of these articles have suggested a number of ways to improve the situation via protocol changes. For the present, however, we are more interested in making do with what we have.

More practically speaking, most TCP stacks have never been abused in quite this way before, so it's not too surprising that they don't deal well with this level of load. The standard UNIX model of forking a new server each time a connection opens doesn't scale particularly well either, and the Netscape Server uses a process-pool model for just this reason.

Kernels

The problems that took the most time for us to solve involved the UNIX kernel. Not having sources for most platforms makes it something of a black box. We hope that sharing some hard-won insights in this area will prove especially useful to the reader.

TCP Tuning

There are several kernel parameters which one can tweak that will often improve the performance of a web or FTP server significantly.

The size of the listen queue corresponds to the maximum number of connections pending in the kernel. A connection is considered pending when it has not been fully established, or when it has been established and is waiting for a process to do an *accept*(2). If the queue size is too small, clients will sometimes see "connection refused" or "connection timed out" messages. If it is too big, results are sporadic: some machines seem to function nicely, while others of similar or identical configuration become hopelessly bogged down. You will need to experiment to find the right size for your listen queue. Version 1.1 of the Netscape Communications and Commerce Servers will never request a listen queue larger than 128.

In kernels that have BSD-based TCP stacks, the size of the listen queue is controlled by the SOMAXCONN parameter. Historically, this has been a #define in the kernel, so if you don't have access to your OS source code, you will probably need to get a vendor patch which will allow you to tune it. In Solaris, this parameter is called tcp_conn_req_max and can be read and written using *ndd*(1M) on /dev/tcp. Sun has chosen to limit the size to which one can raise tcp_conn_req_max using ndd to 32. Contact Sun to find out how to raise this limit further.

Additionally, the kernel on a server machine needs to have enough memory to buffer all the data that it is sending out. In variants of UNIX that use a BSD-based TCP stack, these buffers are called mbufs. The default number of mbufs in most kernels is way too small for TCP traffic of this nature; reconfiguration is usually required. We have found that trial and error is required to find the right number: if 'netstat -m' shows that requests for memory are being denied, you probably need more mbufs. Under IRIX, the parameter you will need to raise is called nm_clusters and it lives in /var/sysgen/master.d/bsd.

TCP employs a mechanism called keepalive that is designed to make sure that when one host of a TCP connection loses contact with its peer host, and either host is waiting for data from its peer, the waiting system does not wait indefinitely for data to arrive. Under the sockets interface, if the socket the system is waiting for is configured to have the SO_KEEPALIVE option turned on, the system will send a keepalive packet to the remote system after it has been waiting for a certain period of time. It will continue sending a packet periodically, and will give up and close the connection if the system does not respond after a certain number of tries.

Many systems provide a mechanism for changing the interval between TCP keepalive probes. Typically, the period of time before a system will send a keepalive packet is measured in hours. This is to make sure that the system does not send large numbers of keepalive packets to hosts which, for example, have idle telnet sessions that simply don't have data to send for long periods of time.

With a web server, an hour is an awfully long time. If a browser does not send information the server is waiting for within a few minutes, it is likely that the remote machine has become unreachable. In the past, router failures were the typical cause of hosts becoming unreachable. In today's Internet, that problem still exists, while at the same time an increasingly large number of users are using a modem with SLIP or PPP as their connection to the Internet. Our experience has shown that these types of connections are unstable, and cause the most situations where a host suddenly becomes silent and unreachable. Most HTTP servers have a timeout built in so that if they have waited for data from a client for a few minutes, they will forcibly close that connection. The situations where a server is not actively waiting for data are the ones most important for keepalive.

If "netstat -an" shows many idle sockets in the kernel or idle HTTP servers waiting for them, you should first check whether your server software sets the SO_KEEPALIVE socket option. The Netscape Communications and Commerce servers do so. The second thing you should check is whether your system allows you to change the interval between keepalive probes. Many systems such as IRIX and Solaris provide mechanisms for changing the keepalive interval to minutes instead of hours. Most systems we've encountered have a default of two hours; we typically truncate it to 15 minutes. If your system is not a dedicated web server system, you should consider keeping the value relatively high so idle telnet sessions don't cause unnecessary network traffic. The third thing you should check with your vendor is whether their TCP implementation allows sockets to time out during the final stages of a TCP close. Certain versions of the BSD TCP code on which many of today's systems are based do not use keepalive timeouts during close situations. This means that a connection to a system that becomes unreachable before it has fully acknowledged the close can stay in your machine's kernel indefinitely. If you see this situation, contact your vendor for a patch.

Your Vendor Is Your Friend; Or, The Value of the Patch

In almost every case, we have gotten quite a bit of value from working directly with the vendor. Since very high volume TCP service of the nature we describe is a fairly new phenomenon, OS vendors are only beginning to adapt their kernels for this.

There are patches to allow the system administrator to increase the listen queue size for IRIX 5.2 and 5.3, as well as enabling the TCP keepalive timer while a socket is in closing states. These patches also fix other problems including a few related to multiprocessor correctness and performance. Contact SGI for the current patch numbers; if you are using a WebFORCE system you should already have them. With these patches, most parameters the administrator will need to edit are in /var /sysgen/master.d/bsd.

If you are using Solaris 2.3, you will definitely want to get the most recent release of the kernel jumbo patch, number 101318. In general, we've found Solaris 2.4 able to handle much more traffic than even a 2.3 system with scalability patches installed. If upgrading to 2.4 is an option, we highly recommend it especially when your traffic starts to reach the range of multiple hundreds of thousands of hits per day. The 2.4 jumbo patch number 101945 is also recommended, both for security as well as stability reasons.

Logging

Generally, the less information that you need to log, the better performance you will get. We found that by turning off logging entirely, we typically realized a performance gain of about 20%. In the future, many servers will offer alternative log file formats to the current "common log format," which will provide better performance as well as record only the information most important to the site administrator.

Many servers offer the ability to perform reverse DNS lookups on the IP addresses of the clients that access your server. While it is very useful information, having your server do it at run-time tends to be a performance problem. Since many DNS lookups either timeout or are extremely slow, the server then generates extra traffic on the local network, and devotes some (often non-trivial) amount of networking resources to waiting for DNS response packets.

For high-volume logging, syslogd also causes performance problems; we suggest avoiding it. If one is logging 10 connections per second, and each connection causes two pieces of data to be logged (as it does for us), this could mean up to 20 context-switches into syslogd and 20 out of it per second. This overhead is in addition to any logging-related I/O and all processing related to actual content service.

On our site, the logs are rotated once every 24 hours and compressed into a staging directory. A separate UNIX machine inside of our firewall uses an SSLified *rsh*(1) to bring the individual logs to a 4 gigabyte partition where the logs are uncompressed, concatenated and piped to our analysis software. Reverse DNS lookups are done at this point rather than at run time on the server, which allows us to do only one reverse lookup per IP address that connected during that day. Processing and lookups on the logs from all of the machines on our site takes approximately an hour to complete.

A single compressed log file is approximately 70 megabytes, and consequently, we end up with over 250 MB of log files daily. Our method of log manipulation allows for an automated system of backing up and processing a months worth of data with little or no human intervention. Tape backups are generated onto 8mm tape once monthly.

Overall analysis of the log files shows consistent data supporting the following:

a) Peak loads occur between 12 and 3 o'clock PM, PST. Peak connection rates were between 120-140 connections per second, per machine. A second peak of roughly half the amplitude occurs between 5 and 6 o'clock PM, PST.
b) Wednesday is the highest load day of the week, generating more than both weekend days combined.

Equipment

Networks

We found that one UNIX machine doing high-volume content service was about all that an Ethernet could handle. Putting two such machines on a single Ethernet caused the performance of both machines to degrade badly as the net became saturated with traffic and collisions. More analysis of our network data is still needed. We are finding it to be more cost effective to have one Ethernet per host than to purchase FDDI equipment for all of them.

As an aside, we have found SGI's addition of the "-C" switch to *netstat*(1) in IRIX to be extremely useful. It displays the data collected by netstat in a full-screen format which is updated dynamically.

Memory

This is fairly simple: get lots of it. You want to have enough memory both for buffering network data and for your filesystem cache to keep most of the frequently accessed files that it serves in memory. The filesystem read cache hit-rate percentage on our web servers is almost always 90% or above. Most modern UNIXes automatically pick a reasonable size for the buffer cache, but some may require manual tuning. Many OS vendors include useful tools for monitoring your cache hit rates: System V derivatives have sar; we also found HP/UX's *monitor*(1M) and IRIX's *osview*(1) helpful.

Our Typical Configuration

A typical webserver at our site is a workstation class machine (e.g., Sun SPARC 20, SGI Indy, or Pentium P90) running between 128 and 150 processes. For UNIX machines at least, we have found 128 megabytes of memory to be about the most that our machines can use. With this much memory, we have all the network buffer space we need, we get a high filesystem read-cache hit rate, and usually have a few (or even tens of) megabytes to spare (depending on the UNIX version).

Generally one gigabyte or more of disk is necessary. These days, each of our servers generates over 200 megs of log data per day (before compression), and the amount of HTML content we are housing continues to grow. Data from sar and kernel profiling code suggest that our boxes are spending between 5% and 20% of their time waiting for disk I/O. Given the high read-cache hit-rate, we expect that by moving our log files onto a separate fast/wide SCSI drive and experimenting with filesystem parameters, this percentage will decrease fairly significantly.

Miscellaneous Points

In this section, we will discuss a few random things that we have learned during our tenure managing web servers.

A Bit About Security

In addition to the normal security concerns of sites on the Internet[12], web servers have some unique security "opportunities". One of the most notable is CGI programs[13]. These allow the author to add all sorts of interesting functionality to a web site. Unfortunately, they can also be a real security problem: since they generally take data entered by a web user as their input, they need to be very careful about what such data is used for.

If a CGI script takes an email address and hands it off to sendmail on the command line, the script needs to go through and make sure that no unescaped shell characters are given to the shell that might cause it to do something unexpected. Such unexpected interplay between different programs is a common cause of security violations. Since many users who want to provide programmatic functionality on their web pages are not intimately familiar with the ins and outs of UNIX security, one approach to this problem is to simply forbid CGI programs in users' personal web pages.

However, we suggest an alternative: mandate use of taintperl[14] for CGI programs written by users. Perl is already one of the predominant scripting languages used to write CGI programs; it is extremely powerful for manipulating data of all sorts and producing HTML output. taintperl is a version of perl which keeps track of the source of the data that it uses. Any data input by the user is considered by the taintperl interpreter to be tainted, and can't be used for dangerous operations unless explicitly untainted. This means that such scripts can be reasonably easily audited by a security officer by grepping for the untaint command and carefully analyzing the variables on which it is used.

Web Server Heterogeneity

An interesting feature of our site is that it is an ideal testbed for ports of our server to new platforms. Since it was clear to us from the beginning that we would be using it this way, we needed to think about how we would deal with CGI programs and their environment. After some thought, we decided that it just wasn't practical to try to port and test all of our CGI content (and force our users to do the same for their home pages) to every new platform that we wanted to test. It turned out that we were very fortunate to have considered this early, as we eventually ended up testing our Windows NT port on our web site.

We designated a single alias to a machine that would run all of our CGI programs. We then created the guideline that all HTML pages should simply point all CGI script references to this alias. A nice side effect is that this type of content is partitioned to its own machine which can be specifically tailored to CGI service. If the machine pool contains many incompatible machines, this setup avoids having to maintain binaries for CGI programs compiled for each machine. An average day at our site shows 50,000 (out of 7.5 million) requests for CGI scripts (note that this ratio is almost certainly very dependent on the type of content served).

An additional experiment would be to partition other types of content (e.g., graphics) to their own machines in the same way. If necessary, DNS-based load-balancing could be used to spread out load across multiple machines of the same type (e.g., gif.netscape.com could be used to refer to multiple machines whose only purpose in life is to serve GIF files).

FTP vs. HTTP

Both FTP and HTTP offer a easy way to handle file transfer, each with relative strengths.

FTP provides a "busy signal", that is, feedback to the user indicating that a site is currently processing too many transactions. That limit is easily set by the site administrator. HTTP provides a mechanism for this as well, however it is not implemented by many HTTP servers primarily due to the fact that it can be very confusing for users. When a user connects to an FTP site, they are allowed to transfer every document they need in that session. Due to HTTP's stateless nature and the fact that it uses a new connection for each file transfer, a user can easily get an HTML document and then be refused service when asking for the document's inlined images. This makes for a very confusing user experience. It is hoped that future work in HTTP development will help to alleviate this problem. Further work in URL or URN arenas will hopefully provide more formal mechanisms for defining alternative distribution machines.

In a system planning sense, FTP should be considered to be a separate service, and therefore can be cleanly served from a completely different computer. This offers easier log analysis of file delivery vs. html served, and also aids in security efforts. A system running only one service, correctly configured, is less likely to be breached, and if breached, does not mean loss of security for our entire site.

Performance gains are also likely. Content typically served by FTP is composed of large files, compared to HTTP-served data which is typically designed to be small and quickly accessible by modem users. A document and its inlined images are short enough to be delivered in short periods. Mixing the two different types can make it hard to pin down system bottlenecks, especially if FTP and HTTP are being served from a single machine. Many times the two services will compete for the same resources, making it hard to track down problems in both areas.

While running both FTP and HTTP servers on one machine, we found that 128 HTTP daemon processes and an imposed limit of 50 simultaneous FTP connections was about all a workstation-class system would tolerate. Further growth beyond that would cause each service to be periodically denied network resources. Once separated, however, 250 simultaneous FTP connections on a workstation-class machine was handled easily.

HTTP service, on the other hand, offers a method to gather useful information from the requestor via forms before allowing file transfer to take place. This became a necessity at Netscape, as the encryption technology in our software required a certain amount of legal documentation to be agreed to prior to download.

Conclusion and Future Directions

Although sites such as ours are currently the exception, we expect that they will soon become the rule as the Internet continues its exceedingly rapid growth. Additionally, we expect content to become vastly more dynamic in the future, both on the front end (using mechanisms such as server push[14] and Java[15]) and the backend (where using SQL databases and search engines will become even more common). This promises to provide many new challenges, especially in the area of performance measurement and management.

We hope that the techniques and information in this paper will prove helpful to folks who wish to administer sites providing a very high volume of service.

Acknowledgements

Special thanks to Brendan Eich for his toolsmithing and his willingness to get his hands dirty with the kernel, to Jeff Weinstein for SSLrsh and SSLrdist, to Rod Beckwith for general network magic, and to Gene Tran for help with performance measurement and the SGI kernel profiler. Thanks to Bill Earl, Mukesh Kacker, Mike Karels and Vernon Schryver for TCP bug-fixes and general advice. We also appreciate the input from the folks who took the time to read drafts of our paper.

About the Authors

The authors have been responsible for the design, implementation, and babysitting of the Netscape web site since its inception.

Dan Mosedale <dmose@netscape.com> is Lead UNIX Administrator at Netscape. He has been managing UNIX boxes for long enough to dislike them thoroughly. Dan likes playing around with weird Internet stuff and wrote a FAQ list about getting connected to the MBONE.

William Foss <bill@netscape.com> is Webmaster at Netscape. His current focus is on how to make the site scale even further in an economical fashion. Prior to Netscape, he had the enviable job of playing with large scale UNIX systems and working for Jim Clark at Silicon Graphics.

Rob McCool <robm@netscape.com> is a member of technical staff at Netscape. He designed and implemented the Netsite Communications and Commerce servers. Prior to Netscape he designed, implemented, tested, documented and supported NCSA httpd from its inception through version 1.3.

Bibliography

[1] http://home.netscape.com/comprod/mirror/index.html

[2] http://home.netscape.com

[3] Kwan, T., McGrath, R., & Reed, D., "User Access Patterns to NCSA's World Wide Web Server," http://www-pablo.cs.uiuc.edu/Papers/WWW.ps.Z.

[4] Brisco, T., "DNS Support for Load Balancing," RFC 1794, USC/Information Sciences Institute, April 1995. ftp://ds.internic.net/rfc/rfc1794.txt

[5] Schemers, Roland J. "lbnamed: A Load Balancing Name Server in Perl" LISA IX Conference Proceedings, http://www-leland.stanford.edu/~schemers/docs/lbnamed/lbnamed.html

[6] ftp://prep.ai.mit.edu/pub/gnu/cvs-1.5.tar.gz

[7] Cooper, M., "Overhauling Rdist for the 90's," LISA VI Conference Proceedings, pp. 175-188. ftp://usc.edu/pub/rdist

[8] http://home.netscape.com/info/security-doc.html

[9] Padmanabhan, V., & Mogul, J., "Improving HTTP Latency," Proceedings of the Second International WWW Conference (October 1994), pp. 995-1005. http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html

[10] Spero, S., "Analysis of HTTP Performance Problems," 1994. http://sunsite.unc.edu/mdma-release/http-prob.html

[11] Viles, C., & French, J., “Availability and Latency of World Wide Web Information servers,” Computing Systems, Vol. 8, No 1, pp. 61-91, USENIX Association.

[12] When looking for UNIX security information, a couple of excellent places to start are http://www.alw.nih.gov/Security/security.html and http://www.yahoo.com/Computers_and_Internet/Security_and_Encryption/

[13] http://hoohoo.ncsa.uiuc.edu/cgi/

[14] http://home.netscape.com/assist/net_sites/pushpull.html

[15] http://java.sun.com