Measuring the Capacity of a Web Server under Realistic Loads

Gaurav Banga Peter Druschel

Department of Computer Science
Rice University
Houston, TX 77005
gaurav@cs.rice.edu

Abstract

The World Wide Web and its related applications place substantial performance demands on network servers. The ability to measure the effect of these demands is important for tuning and optimizing the various software components that make up a Web server. To measure these effects, it is necessary to generate realistic HTTP client requests in a test-bed environment. Unfortunately, the state-of-the-art approach for benchmarking Web servers is unable to generate client request rates that exceed the capacity of the server being tested, even for short periods of time. Moreover, it fails to model important characteristics of the wide area networks on which most servers are deployed (e.g. delay and packet loss).

This paper examines pitfalls that one encounters when measuring Web server capacity using a synthetic workload. We propose and evaluate a new method for Web traffic generation that can generate bursty traffic, with peak loads that exceed the capacity of the server. Our method also models the delay and loss characteristics of WANs. We use the proposed method to measure the performance of widely used Web servers. The results show that actual server performance can be significantly lower than indicated by standard benchmarks under conditions of overload and in the presence of wide area network delays and packet losses.

1 INTRODUCTION

The explosive growth in size and usage of the World Wide Web results in increasing load on its constituent networks and servers, and stresses the protocols that the Web is based on. Improving the performance of the Web has been the subject of much recent research, addressing various aspects of the problem such as better Web caching [Bestavros *et al.* 1995; Braun and Claffy 1994; Chankhunthod *et al.* 1996; Seltzer and Gwertzman 1995; Williams *et al.* 1996], HTTP protocol enhancements [Fielding *et al.* 1997; Mogul *et al.* 1997; Padmanabhan and Mogul 1994; Banga *et al.* 1997; Spasojevic *et al.* 1994], better HTTP servers and proxies [Apache 1998; Chankhunthod *et al.* 1996; ACME 1998; Zeus 1998] and server OS implementations [Banga *et al.* 1999; Mogul 1995b; Compaq 1998; Druschel and Banga 1996; Mogul and Ramakrishnan 1997; Banga *et al.* 1998; Hu *et al.* 1997a; Banga and Mogul 1998].

To date most work on measuring Web software performance has concentrated on accurately characterizing Web server workloads in terms of request file types, transfer sizes, locality of reference in URLs requested and other related statistics [Arlitt and Williamson 1996; Bestavros *et al.* 1995; Braun and Claffy 1994; Crovella and Bestavros 1996; Cunha *et al.* 1995; Kwan *et al.* 1995]. Some researchers have tried to evaluate the performance of Web servers and proxies using real workloads directly [Maltzahn *et al.* 1997; Mogul 1995a; Banga and Mogul 1998]. However, this approach suffers from the experimental difficulties involved in non-intrusive measurement of a live system and the inherent irreproducibility of live workloads.

Recently, there has been some effort towards Web server evaluation through generation of synthetic HTTP client traffic, based on invariants observed in real Web traffic [SPEC 1996; Web66 1996; Mecklermedia 1996; Mindcraft 1998; Almeida *et al.* 1996]. Unfortunately, there are pitfalls that arise in generating heavy and realistic Web traffic using a limited number of client machines. These problems can lead to significant deviation of benchmarking conditions from reality and fail to predict the performance of a given Web server.

In a Web server evaluation test-bed consisting of a small number of client machines, it is difficult to simulate many independent clients. Typically, a load generating scheme is used that equates client load with the number of client processes in the test system. Adding client processes is thought to increase the total client request rate. Unfortunately, some peculiarities of the TCP protocol limit the traffic generating ability of such a simple scheme. Because of this, generating request rates that exceed the server's capacity is nontrivial, leaving the effect of request bursts on server performance unevaluated.

The commonly used scheme generates client traffic that has little resemblance in its temporal characteristics to real-world Web traffic. In addition, there are fundamental differences between the delay and loss characteristics of WANs and the LANs used in test-beds. All of these factors may cause certain important aspects of Web server performance to remain unevaluated. Finally, care must be taken to ensure that limited resources in the simulated client systems do not distort the server performance results.

In this paper, we examine these issues and their effect on the process of Web server evaluation. We propose a new methodology for HTTP request generation that complements the work on Web workload modeling. Our work focuses on those aspects of the request generation method that are important for providing a scalable means of generating realistic HTTP requests, including peak loads that exceed the capacity of the server. We also address the need to model the delay and loss characteristics of WANs. We expect that this request generation methodology, in conjunction with a representative HTTP request data set like the one used in the SPECWeb benchmark [SPEC 1996] and a representative temporal characterization of HTTP traffic, will result in a benchmark that can more accurately predict actual Web server performance.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the dynamics of a typical HTTP server running on a UNIX-based TCP/IP network subsystem. Section 3 identifies problems that arise when trying to measure the performance of such a system. In Section 4 we describe our methodology. Section 5 gives a quantitative evaluation of our methodology, and presents measurements of Web servers using the proposed method. Finally, Section 6 covers related work and Section 7 offers some conclusions.

2 DYNAMICS OF AN HTTP SERVER

In this section, we briefly describe the working of a typical HTTP server on a machine with a UNIX-based TCP/IP implementation. The description provides background for the discussion in the following sections. For simplicity, we focus our discussion on a BSD based network subsystem [McKusick *et al.* 1996; Wright and Stevens 1995]. The behavior of other implementations of TCP/IP, such as those found in UNIX System V [Bach 1986] and Windows NT [Custer 1993], is similar.

In the HTTP protocol, for each URL fetched, a browser establishes a new TCP connection to the appropriate server, sends a request on this connection and then reads the server's response[1)]. To display a typical Web page, a browser may need to initiate several HTTP transactions to fetch the various components (HTML source, images) of the page.

Figure 1 shows the sequence of events in the connection establishment phase of an HTTP transaction. The dashed flow on the right side of the figure shows the processing performed by a user-level HTTP process, while the left picture shows the events in the network and in the server's in-kernel TCP/IP code. After starting, the Web server process *listens* for connection requests on a socket bound to a well known port—typically port 80. When a connection establishment request (TCP SYN packet) from a client is received on this socket (Figure 1, position 1), the server TCP responds with a SYN-ACK TCP packet, creates a

[1)] HTTP 1.1 supports persistent connections where several requests may be sent serially over one connection.

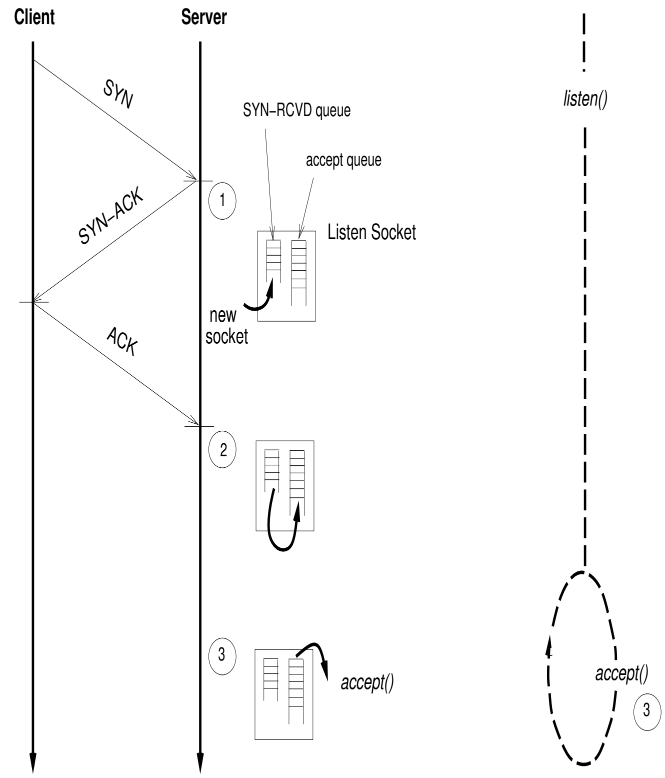

Figure 1: HTTP Connection Establishment Timeline

socket for the new, incomplete connection, and places it in the listen socket's SYN-RCVD queue. Later, when the client responds with an ACK packet to the server's SYN-ACK packet (position 2), the server TCP removes the socket created above from the SYN-RCVD queue and places it in the listen socket's queue of connections awaiting acceptance (accept queue). Each time the WWW server process executes the `accept()` system call (position 3), the first socket in the accept queue of the listen socket is removed and returned. After accepting a connection, the WWW server—either directly or indirectly by passing this connection to a helper process—reads the HTTP request from the client, sends back an appropriate response, and closes the connection.

In most UNIX-based TCP/IP implementations, the kernel variable `somaxconn` limits the maximum *backlog* on a listen socket. This backlog is an upper bound on the sum of the lengths of the SYN-RCVD and accept queues. In the context of the discussion above, the server TCP drops incoming SYN packets (Figure 1, position 1) whenever this sum exceeds a value of 1.5 times the backlog[2)]. When the client TCP misses the SYN-ACK packet, it goes into an exponential backoff paced SYN retransmission mode until it either receives a SYN-ACK, or its connection establishment timer expires[3)].

[2)] In the System V Release 4 flavors of UNIX (e.g. Solaris) this sum is limited by $1 \times backlog$ rather than $1.5 \times backlog$.

[3)] 4.4BSD's TCP retransmits at 6 seconds and 30 seconds after the first SYN is sent before finally giving up at 75 seconds. Other TCP implementations behave similarly.

The average length of the SYN-RCVD queue depends on the average round-trip delay between the server and its clients, and the connection request rate. This is because a socket stays on this queue for a period of time equal to the round-trip delay. Long round-trip delays and high request rates increase the length of this queue. The accept queue's average length depends on how fast the HTTP server process calls accept() (i.e., the rate at which it serves requests,) and the request rate. If a server is operating at its maximum capacity, it cannot call accept() fast enough to keep up with the connection request rate and the queue grows.

Each socket's protocol state is maintained in a data structure called a Protocol Control Block (PCB). TCP maintains a table of the active PCBs in the system. A PCB is created when a socket is created, either as a result of a system call, or as a result of a new connection being established. A TCP connection is closed either actively by one of the peers executing a close() system call, or passively as a result of an incoming FIN control packet. In the latter case, the PCB is deallocated when the application subsequently performs a close() on the associated socket. In the former case, a FIN packet is sent to the peer and after the peer's FIN/ACK arrives and is ACKed, the PCB is kept around for an interval equal to the so-called TIME-WAIT period of the implementation[4)]. The purpose of this TIME-WAIT state is to be able to retransmit the closing process's ACK to the peer's FIN if the original ACK gets lost, and to allow the detection of delayed, duplicate TCP segments from this connection.

A well-known problem exists in many traditional implementations of TCP/IP that limits the throughput of a Web server. These systems have small default and maximum values for somaxconn. Since this threshold can be reached when the accept queue and/or the SYN-RCVD queue fills, a low value can limit throughput by refusing connection requests needlessly. As discussed above, the SYN-RCVD queue can grow because of long round-trip delays between server and clients, and high request rates. If the limit is too low, an incoming connection may be dropped even though the Web server may have sufficient resources to process the request. Even in the case of a long accept queue, it is usually preferable to accept a connection, unless the queue already contains enough work to keep the server busy for at least the client TCP's initial retransmission interval (6 seconds for 4.4BSD). To address this problem, some vendors have increased the maximum value of somaxconn (e.g. Digital UNIX uses 32767, Solaris uses 1000). In Section 3, we will see how this fact interacts with WWW request generation.

Commonly used Web server applications use a number of different process architectures. Some servers, e.g. Apache [Apache 1998], use a process-per-connection architecture. This is similar to the UNIX server model, although a set of persistent pre-forked processes is used in order to reduce forking overhead. Multi-process servers can suffer from large context-switching overhead, so many recent servers use a single-process

[4)] This TIME-WAIT period should be set equal to twice the Maximum Segment Lifetime (MSL) of a packet on the Internet (RFC 793 [Postel 1981] specifies the MSL as 2 minutes, but many implementations use a much shorter value.)

architecture. There are two types of single-process servers: event-driven servers, which use the `select()` system call to allow a single thread[5)] to perform processing for all connections being handled by the server, and multi-threaded servers, where each connection is assigned to a dedicated thread. Zeus [Zeus 1998] is an example of a single-process server. In Section 5, we will discuss the effect of WAN characteristics on the performance of a process-per-connection server and a single-process server.

3 PROBLEMS IN GENERATING SYNTHETIC HTTP REQUESTS

This section identifies problems that arise when trying to measure the performance of a Web server, using a test-bed consisting of a limited number of client machines. For reasons of cost and ease of control, it is desirable to use a small number of client machines to simulate a large client population. We first describe a straightforward, commonly-used scheme for generating Web traffic, and identify problems that arise.

In the simple method, a set of N Web client processes[6)] execute on P client machines. Usually, the client machines and the server share a LAN. Each client process repeatedly establishes a HTTP connection, sends a HTTP request, receives the response, waits for a certain time (think time), and then repeats the cycle. The sequence of URLs requested comes from a database designed to reflect realistic URL request distributions observed on the Web. Think times are chosen such that the average URL request rate equals a specified number of requests per second. N is typically chosen to be as large as possible given P, so as to allow a high maximum request rate. To reduce cost and for ease of control of the experiment, P must be kept low. All the popular Web benchmarking efforts that we know of use a load generation scheme similar to this [SPEC 1996; Web66 1996; Mecklermedia 1996; Mindcraft 1998].

Several problems arise when trying to use the simple scheme described above to generate realistic HTTP requests. We describe these problems in detail in the following subsections.

3.1 Inability to generate excess load

In the World Wide Web, HTTP requests are generated by a huge number of clients, where each client has a think time distribution with large mean and variance. Furthermore, the think time of clients is not independent; factors such as human users' sleep/wake patterns, and the publication of Web content at scheduled times causes high correlation of client HTTP requests. As a result, HTTP request traffic arriving at a server is bursty with the burstiness being observable at several scales of observation [Crovella and

[5)] An event-driven server on a multi-processor machine uses as many threads as there are processors in the system.

[6)] In this discussion we use the terms client processes to denote either client processes or client threads, as this distinction makes no difference to our method.

Bestavros 1996], and with peak rates exceeding the average rate by factors of 8 to 10 [Mogul 1995a; Stevens 1996]. Furthermore, peak request rates can easily exceed the capacity of the server.

By contrast, in the simple request generation method, a small number of clients have independent think time distributions with small mean and variance. As a result, the generated traffic has little burstiness. The simple method generates a new request only after a previous request is completed. This, combined with the fact that only a limited number of clients can be supported in a small testbed, implies that the clients stay essentially in lock-step with the server. That is, the rate of generated requests never exceeds the capacity of the server.

Consider a Web server that is subjected to HTTP requests from an increasing number of clients in a test-bed using the simple method. For simplicity, assume that the clients use a constant think time of zero seconds, i.e., they issue a new request immediately after the previous request is completed. For small document retrievals, a small number of clients (3–5 for our test system) are sufficient to drive the server at full capacity. If additional clients are added to the system, the only effect is that the accept queue at the server will grow in size, thereby adding queuing delay between the instant when a client perceives a connection as established, and the time at which the server accepts the connection and handles the request. This queuing delay reduces the rate at which an individual client issues requests. Since each client waits for a pending transaction to finish before initiating a new request, the aggregate connection request rate of all the clients remains equal to the throughput of the server.

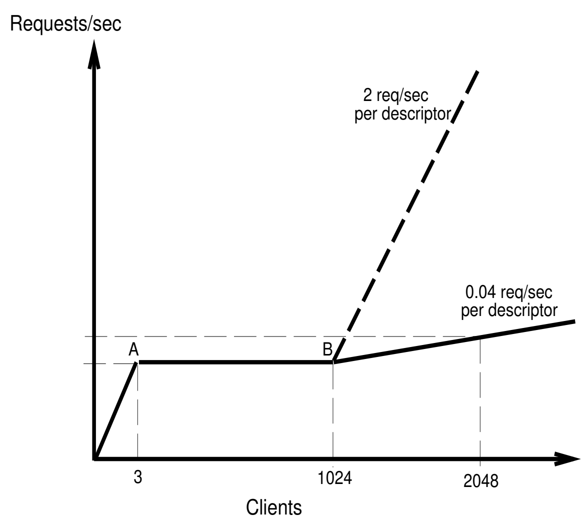

Figure 2: Request Rate versus no. of Clients

As we add still more clients, the server's accept queue eventually fills. At that point, the server TCP starts to drop connection establishment requests that arrive while the sum of the SYN-RCVD and accept queues is at its limit. When this happens, the clients whose connection requests are dropped go into TCP's exponential backoff and generate further requests at a very low rate. (For 4.4BSD based systems, the rate is 3 requests in 75 seconds.) This behavior is shown graphically by the solid line in Figure 2. The server saturates at point A, and then the request rate remains equal to the throughput of the server until the accept queue fills up (point B). Thereafter the rate increases as in the solid line at 0.04 requests/second per added client.

To generate a significant rate of requests beyond the capacity of the server, one would have to employ a huge number of client processes. Suppose that for a certain size of requested file, the capacity of a server is 100 connections/sec, and we want to generate requests at 1100 requests/sec. One would need on the order of 25000 client processes $((1100-100)/(3/75))$ beyond the maximum size of the listen socket's accept queue to achieve this request rate. Recall from Section 2 that many vendors now configure their systems with a large value of somaxconn to avoid dropping incoming TCP connections needlessly. Thus, with somaxconn $= 32767$, one would need 74151 processes $(1.5 \times 32767 + 25000)$ to generate 1100 requests/sec. Efficiently supporting such large numbers of client processes on a small number of client machines is not feasible.

A real Web server, on the other hand, can easily be overloaded by the huge (practically infinite) client population existing on the Internet. As mentioned above, it is not unusual for a server to receive bursts of requests at rates that exceed the average rate by factors of 8 to 10. The effect of such bursts is to temporarily overload the server.

Many UNIX and non-UNIX based network subsystems suffer from poor overload behavior [Banga 1998; Druschel and Banga 1996; Mogul and Ramakrishnan 1997]. Under heavy network load these interrupt-driven systems can enter a state called *receiver-livelock* [Ramakrishnan 1992]. In this state, the system spends all its resources processing incoming network packets (in this case TCP SYN packets), only to discard them later because there is no CPU time left to service the receiving application program (in this case the Web server). It is therefore important to evaluate Web server performance under overload.

Synthetic requests generated using the simple method cannot reproduce the bursty aspect of real traffic, and therefore fail to evaluate the behavior of Web servers under overload.

3.2 Inability to model characteristics of real networks

The Internet based Web has network characteristics that differ from the LANs on which Web servers are usually benchmarked. Performance aspects of a server that are dependent on such network characteristics are not evaluated. In particular, the simple method does not model high and variable WAN delays. Packet losses due to congestion are also absent in LAN-based test-beds. Both of these phenomena induce large amounts of state in real-world servers, which is known to adversely affect performance.

For example, WAN delays and packet losses cause long SYN-RCVD queues in a server's listen socket, which may cause poor performance, as discussed in Section 2. Delays and packet losses also increase the lifetime of connections, thus inducing a large number of simultaneous connections at a server, which can cause significant performance degradation of event-driven servers [Banga and Mogul 1998; Maltzahn *et al.* 1997; Fox *et al.* 1997] and other types of servers.

WAN delays also cause an increase in the bandwidth-delay product experienced by a TCP connection. Therefore, the server TCP needs to provide increased amounts of buffer space (in the form of socket send buffers) for Web transfers to proceed at full speed. This increased demand for buffer space may reduce the amount of main memory available for the document cache. Current Web server benchmarks do not expose these performance aspects of servers.

Other aspects of the network subsystem, such as the server TCP's timeout mechanism, are never exercised during benchmarking and may perform poorly in practice. Our experiments, described in detail later in Section 5, suggest that these factors are important in practice.

3.3 Client and network resource constraints

When generating synthetic HTTP requests from a small number of client machines, care must be taken that resource constraints on the *client* machine do not accidentally distort the measured server performance. With an increasing number of simulated clients per client machine, client-side CPU and memory contention are likely to arise. Eventually, a point is reached where the bottleneck in a Web transaction is no longer the server but the client. Designers of commercial Web server benchmarks have also noticed this pitfall. The WebStone benchmark [Mindcraft 1998] explicitly warns about this potential problem, but gives no systematic method to avoid it.

The primary factor in preventing client bottlenecks from affecting server performance results is to limit the number of simulated clients per client machine. In addition, it is important to use an efficient implementation of TCP/IP (in particular, an efficient PCB table implementation) on the client machines, and to avoid I/O operations in the simulated clients that could affect the rate of HTTP transactions in uncontrolled ways. For example, writing logging information to disk can affect the client behavior in complex and undesirable ways.

Similarly, while benchmarking a Web server, it is important to ensure that the bandwidth of the network connecting the client machines to the server is not a bottleneck factor. Many modern workstations can saturate a single 100Mbps link. Therefore, it may be necessary to use multiple network interfaces in the Web server machine to measure its true capacity.

4 A NEW METHOD FOR GENERATING HTTP REQUESTS

In this section, we describe the design of a new method to generate Web traffic. This method addresses the problems raised in the previous section. It should be noted that our work does not by itself address the problem of accurate simulation of Web workloads in terms of the request file types, transfer sizes and locality of reference in URLs requested; instead, we concentrate on mechanisms for generating heavy concurrent traffic that has a temporal behavior similar to that of real Web traffic. Our work is intended to complement the existing work done on Web workload characterization [Bestavros *et al.* 1995; Braun and Claffy 1994; Chankhunthod *et al.* 1996; Seltzer and Gwertzman 1995; Williams *et al.* 1996], and can be easily used in conjunction with it.

4.1 Basic architecture

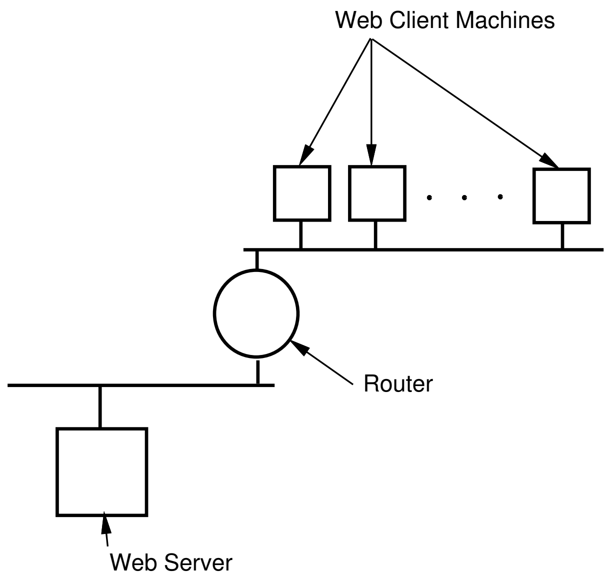

Figure 3: Test-bed architecture

The basic architecture of our test-bed is shown in Figure 3. A set of P client machines are connected to the server machine being tested. Each client machine runs a number of S-Client (short for Scalable Client) processes. The structure of a S-Client, and the number of S-Clients that run on a single machine are critical to our method and are described in detail below. The client machines are connected to the server through a router that has sufficient capacity to carry the maximum client traffic anticipated. The purpose of the router is to simulate WAN effects by introducing an artificial delay and/or dropping packets at a controlled rate.

4.2 S-Clients

A S-Client (Figure 4) consists of a pair of processes connected by a UNIX domain socketpair. One process in the S-Client, *the connection establishment process*, is responsible for generating HTTP requests at a certain rate and with a certain request distribution. After a connection is established, the connection establishment process sends a HTTP request to the server, then it passes on the connection to the *connection handling process*, which handles the HTTP response.

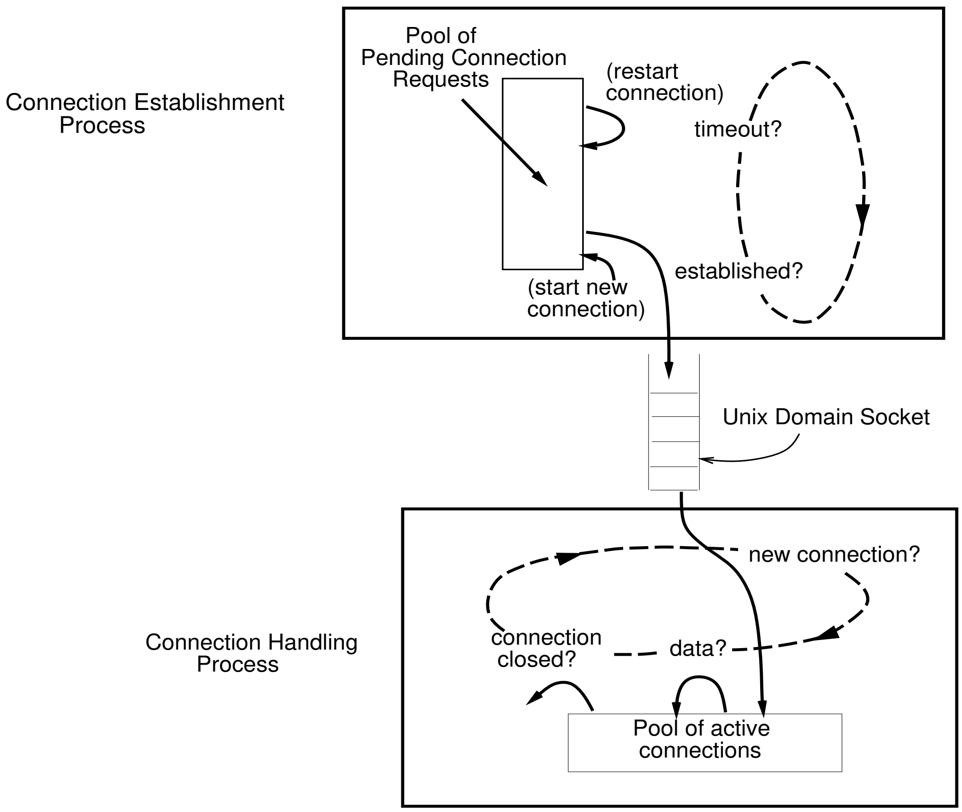

Figure 4: A Scalable Client

The connection establishment process of a S-Client works as follows: The process opens D connections to the server using D sockets in non-blocking mode. These D connection requests[7] are spaced out over T milliseconds. T is required to be larger than the maximal round-trip delay between client and server (remember that an artificial delay may be added at the router).

After the process executes a non-blocking `connect()` to initiate a connection, it records the current time in a variable associated with the used socket. In a tight loop, the process checks if for any of its D active sockets, the connection is complete, or if T milliseconds have elapsed since a `connect()` was performed

[7] In this paper, we model only HTTP/1.0, which uses a dedicated connection for each distinct HTTP request.

on this socket. In the former case, the process sends a HTTP request on the newly established connection, hands off this connection to the other process of the S-Client through the UNIX domain socketpair, closes the socket, and then initiates another connection to the server. In the latter case, the process simply closes the socket and initiates another connection to the server. Notice that closing the socket in both cases does not generate any TCP packets on the network. In the first case, the close merely releases a reference to the corresponding socket. In the second case, the close prematurely aborts TCP's connection establishment timeout period and releases socket resources in the kernel.

The connection handling process of a S-Client waits for 1) data to arrive on any of the active connections, or 2) for a new connection to arrive on the UNIX domain socket connecting it to the other process. In case of new data on an active socket, it reads this data; if this completes the server's response, it closes the socket. A new connection arriving at the UNIX domain socket is simply added to the set of active connections.

The rationale behind the structure of a S-Client is as follows. The two key ideas are to (1) shorten TCP's connection establishment timeout, and (2) to maintain a constant number of unconnected sockets (simulated clients) that are trying to establish new connections. Condition (1) is accomplished by using non-blocking connects and closing the socket if no connection was established after T seconds. The fact that the connection establishment process tries to establish another connection immediately after a connection was established ensures condition (2).

The purpose of (1) is to allow the generation of request rates beyond the capacity of the server with a reasonable number of client sockets. Its effect is that each client socket generates SYN packets at a rate of at least $1/T$. Shortening the connection establishment timeout to $500ms$ by itself would cause the system's request rate to follow the dashed line in Figure 2.

The idea behind (2) is to ensure that the generated request rate is independent of the rate at which the server handles requests. In particular, once the request rate matches the capacity of the server, the additional queuing delays in the server's accept queue no longer reduce the request rate of the simulated clients. Once the server's capacity is reached, adding more sockets (descriptors) increases the request rate at $1/T$ requests per descriptor, eliminating the flat portion of the graph in Figure 2.

To increase the maximal request generation rate, we can either decrease T or increase D. As mentioned before, T must be larger than the maximal round-trip time between client and server. This is to avoid the case where the client aborts an incomplete connection in the SYN-RCVD state at the server, but whose SYN-ACK from the server (see Figure 1) has not yet reached the client. Given a value of T, the maximum value of D is usually limited by OS imposed restrictions on the maximum number of open descriptors in a single process. However, depending on the capacity of the client machine, it is possible that one S-Client with a large D may saturate the client machine.

Therefore, as long as the client machine is not saturated, D can be as large as the OS allows. When multiple S-Clients are needed to generate a given rate, the largest allowable value of D should be used, as this keeps the total number of processes low, thus reducing overhead due to context switches and memory contention between the various S-Client processes. How to determine the maximum rate that a single client machine can safely generate without risking distortion of results due to client side bottlenecks is the subject of the next section.

The proposed method for generating Web traffic scales linearly with the number of client machines as long as network contention is not an issue. This is because the load generating ability of a S-Client is independent of the capacity of the server, as we work around TCP's exponential backoff and because we separate the request generation process and hence the generation rate from the connection handling process.

The presented scheme generates HTTP requests with a trivial think time distribution, i.e., it uses a constant think time chosen to achieve a certain constant request rate. It is possible to generate more complex request processes by adding appropriate think periods between the point where a S-Client detects a connection was established and when it next attempts to initiate another connection. In this way, any request arrival process can be generated whose peak request rate is lower than or equal to the maximum raw request rate of the system. In particular, the system can be parameterized to generate self-similar traffic [Crovella and Bestavros 1996].

4.3 Request generating capacity of a client machine

As noted in the previous section, while evaluating a Web server, it is very important to operate client machines in load regions where they are not limiting the observed performance. Our method for finding the maximum number of S-Clients that can be safely run on a single machine—and thus determine the value of P needed to generate a certain request rate—is as follows. The work that a client machine has to do is largely determined by the sum of the number of sockets D of all the S-Clients running on that machine. Since we do not want to operate a client near its capacity, we choose this value as the largest number N for which the throughput vs. request rate curve when using a single client machine is identical to that resulting from the use of two client machines[8)]. The corresponding number of S-Clients we need to use is found by distributing these N descriptors into as few processes as the OS permits. We call the request rate generated by these N descriptors the maximum raw request rate of a client machine.

It is possible that a single process's descriptor limit (imposed by the OS) is smaller than the average

[8)] A two-machine client configuration may suffer more packet collisions than a single machine configuration on a shared Ethernet. In all our experiments, we use a switched network with sufficient capacity in the switch.

number of simultaneous active connections in the connection handling process of a S-Client. In this case we have no option but to use a larger number of S-Clients with smaller D values to generate the same rate. Due to increased memory contention and context switching, this may actually cause a lower maximum raw request rate for a client machine than if the OS limit on the number of descriptors per process was higher. The number of machines needed to generate a certain request rate may be higher in this case.

4.4 Design of the delay router

This section discusses the design of the router shown in Figure 3. As mentioned in Section 4.1, the purpose of this router is to simulate the effects of WANs. Our goal is to be able use the router to simultaneously model multiple wide area network paths between the various clients and the server, each of which have independent and adjustable latency, bandwidth and loss characteristics.

We implemented the router by modifying the forwarding mechanism in the 4.4BSD TCP/IP protocol stack. As a result, the router software is easily portable to any BSD based UNIX operating system. Our changes to the forwarding mechanism were as follows: The vanilla BSD forwarding mechanism forwards an IP packet destined for another machine as soon as it is processed in `ip_input()` (or after a fragmented packet has been reassembled) by calling `ip_forward()`. In contrast, our delay router queues packets in a kernel data structure, to be forwarded after a specified delay. A dedicated kernel process continuously checks for packets that are due to be forwarded; once a packet is ready, it calls `ip_forward()` to transmit this packet.

Continuous polling is used instead of a periodic, clock-interrupt driven check in order to conserve the timing of packets on the network. Our router is able to conserve temporal packet spacing, and scale the spacing correctly when modeling limited bandwidth paths. If two back-to-back packets were separated in the incoming packet train at the router by T, their spacing in the outgoing train is T scaled by a factor equal to the ratio of the bandwidth of the incoming path to the bandwidth of the path being modeled by the router, assuming that the former is greater than the latter.

In contrast, a periodic check based scheme introduces burstiness in the outgoing packet train, because all packets due for transmission within a check period are transmitted simultaneously. Even with a check period of $10ms$, which corresponds to the clock-interrupt frequency of many commonly-used operating systems, up to 43 back-to-back packets may be sent during each check period if the simulated path bandwidth is $50Mbps$. This issue is important because TCP performance is known to be affected by phenomena such as ACK compression [Shenker *et al.* 1990; Mogul 1993], which also disturb the timing of packets in the network.

The forwarding delay for a packet is computed as the sum of three components. Algebraically, this is expressed as:

$$delay_{packet} = latency_{path} + \frac{size_{packet}}{bandwidth_{path}} + queueing\text{-}time_{path}$$

Essentially, this equation models a limited-bandwidth network pipe between a client and the server. The first component in the equation is a propagation delay, which models the latency of the path. Next, we have the transmission delay, which models the time required to transfer a packet given a certain path bandwidth. The last component models queuing delay inside the router. This is the time that a packet has to wait while the preceding packets, which are being sent along the same path, are transmitted. This is computed as:

$$queueing\text{-}time_{path} = \sum_{i=0}^{queued\ packets\ on\ path} \frac{size_{packet_i}}{bandwidth_{path}}$$

The delay router can support multiple simulated network paths. For example, the router may be configured such that all packets between client machine A and the server traverse one network path with its set of delay and bandwidth parameters. Packets between machine B and the server might use a different path with its own set of delay and bandwidth parameters, and so on. The router can also drop packets on a given simulated path at a controlled rate and with a given loss model. Also, while the current implementation does not model fluctuations in path bandwidth, this could be easily implemented in our framework.

The router allows us to accurately model fairly complex scenarios. For example, we can model a scenario where a certain fraction of a server's load is from clients that are connected to the server via fast networks, such as a LAN. Another fraction of the clients may be assumed to be home users, connected via modems. A third fraction of clients may be connected via moderate latency and high-bandwidth cross-country backbone networks. Finally, a fourth set of hosts may make requests of the server over lossy networks, such as wireless networks.

5 QUANTITATIVE EVALUATION

In this section we present experimental data to quantify the problems identified in Section 3, and to evaluate the performance of our proposed method. We first measure the request generation limitations of the simple approach and evaluate the S-Client based request generation method proposed in Section 4. We then describe the use of S-Clients to measure the overload performance of a Web server. Finally, we present an evaluation of the effect of WAN characteristics on Web server performance.

5.1 Experimental Setup

We use two experimental setups. Our first set of experiments (described in Sections 5.2 through 5.4) were performed in a test-bed consisting of 4 Sun Microsystems SPARCstation 20 model 61 workstations (60MHz SuperSPARC+, 36KB L1, 1MB L2, SPECint92 98.2) as the client machines. The workstations are equipped with 32MB of memory and run SunOS 4.1.3_U1. Our server is a dual processor SPARCStation 20 constructed from 2 erstwhile SPARCStation 20 model 61 machines. This machine has 64MB of memory and runs Solaris 2.5.1. A 155 Mbps ATM local area network connects the machines, using FORE Systems SBA-200 network adaptors. For our HTTP server, we used the NCSA httpd server software, revision 1.5.1. The server's OS kernel was tuned using Web server performance enhancing tips advised by Sun. That is, we increased the total pending connections (accept+SYN-RCVD queues) limit to 1024. In this set of experiments, we used no artificial delay in the router connecting the clients and the server. We will refer to this test-bed as "Testbed-1" in the following discussion.

Our second set of experiments (described in Sections 5.5 and 5.6) were performed on a test-bed consisting of 2 166Mhz Pentium Pro PCs, each with 64MB of memory, as the client machines. Our server was a 333 Mhz Pentium II PC equipped with 128MB of memory. The client machines were connected over a 100Mbps switched Fast Ethernet to a router (300 Mhz Pentium Pro with 128MB of memory). The server was connected to the router over another 100Mbps Fast Ethernet. Both the router and the server had four 100Mbps Fast Ethernet interfaces each. For our server software, we used Apache v1.2.4 [Apache 1998] and Zeus v1.3.0 [Zeus 1998]. All machines, including the router, ran FreeBSD 2.2.5. The router's kernel was slightly modified to incorporate the new forwarding code described in Section 4.4. We will refer to this test-bed as "Testbed-2" below.

5.2 Request generation rate

The purpose of our first experiment is to quantitatively characterize the limitations of the simple request generation scheme described in Section 3. In this experiment, we use Testbed-1. We run an increasing number of client processes distributed across 4 client machines. Each client tries to establish a HTTP connection to the server, sends a request, receives the response and then repeats the cycle. Each HTTP request is for a single file of size 1294 bytes. We measure the request rate (incoming SYNs/second) at the server.

In a similar test, we ran 12 S-Clients distributed across the 4 client machines with an increasing number of descriptors per S-Client and measured the request rate seen at the server. Each S-Client had the

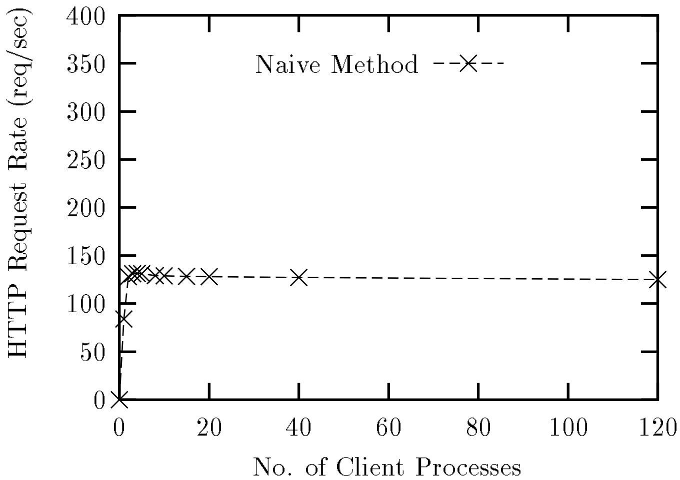

Figure 5: Request rate versus number of clients

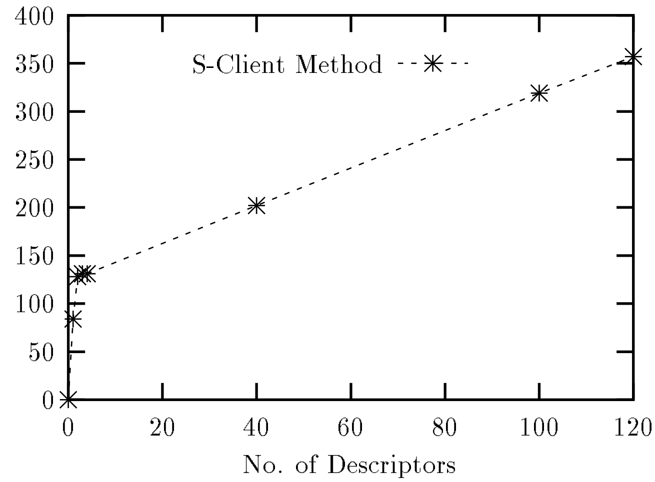

Figure 6: Request rate versus number of descriptors

connection establishment timeout period T set to $500ms$. The same file was requested as in the case of the simple clients. We ensured that the network was not a bottleneck in this experiment.

Figure 5 plots the total connection request rate seen by the server versus the total number of client processes for the simple client test. Figure 6 plots the same metric for the S-Client test, but with the total number of descriptors in the S-Clients on the x-axis. For the reasons discussed earlier, the simple scheme generates no more than about 130 requests/second (which is the capacity of our server for this request size). At this point, the server can accept connections at exactly the rate at which they are generated. As we add more clients, the queue length at the accept queue of the server's listen socket increases and the request rate remains nearly constant at the capacity of the server.

With S-Clients, the request rate increases linearly with the total number of descriptors being used for establishing connections by the client processes. To highlight the difference in behavior of the two schemes in this figure, we do not show the full curve for S-Clients. The complete curve shows a linear increase in request rate all the way up to 2065 requests/second with our setup of four client machines. Beyond this point, client capacity resource limitations set in and the request rate ceases to increase. More client machines are needed to achieve higher rates. Thus we see that S-Clients enable the generation of request loads that greatly exceed the capacity of the server. The generated load also scales very well with the number of descriptors being used.

5.3 Overload Behavior of a Web Server

Being able to reliably generate high request rates, we used the new method to evaluate how a typical commercial Web server behaves under high load. We measured the HTTP throughput achieved by the server in transactions per second. The same 1294 byte file was used in this test.

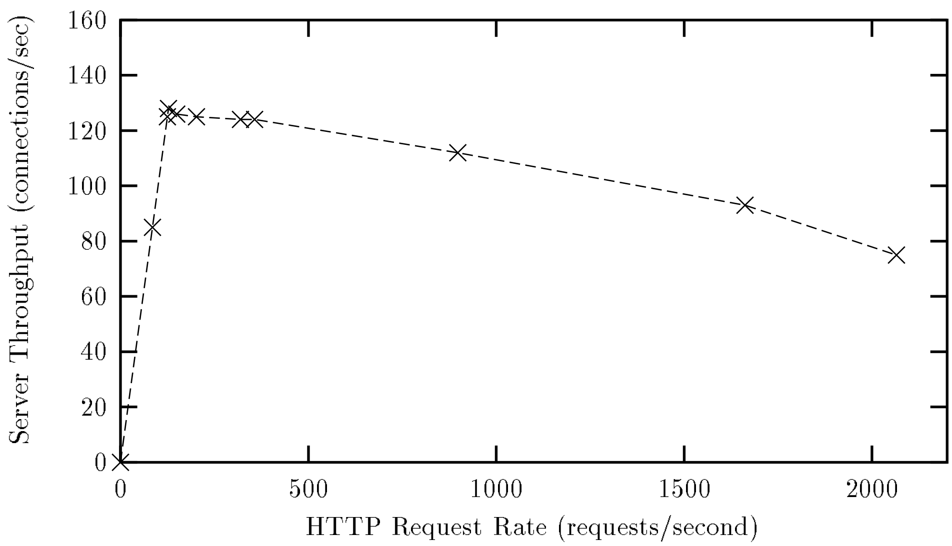

Figure 7: Web server throughput versus request rate

Figure 7 plots the server throughput versus the total connection request rate. As before, the server saturates at about 130 transactions per second. As we increase the request rate beyond the capacity of the server, the server throughput declines, initially somewhat slowly, and then more rapidly reaching about 75 transactions/second at 2065 requests/second. This decline in throughput with increasing request rate is due to the CPU resources spent on protocol processing for incoming requests (SYN packets) that are eventually dropped due to the backlog on the listen socket (i.e., the full accept queue).

The reason behind this poor overload behavior of Web servers has been explained elsewhere [Banga 1998]. Essentially, the interrupt-driven nature of the protocol stack causes strictly higher priority to be assigned to non-flow-controlled kernel activities, such as the processing of incoming SYN packets. Packet processing associated with established HTTP connections is flow-controlled by the progress rate of user-level applications. Thus, under overload, the progress of of established HTTP connections suffers as the system spends increasing amounts of time processing SYN packets only to drop these later at the full listen socket.

The slope of the throughput drop corresponds to about 325 μsec worth of processing time per SYN packet. While this may seem large, it is consistent with our observation of the performance of a server system based on a 4.4BSD network subsystem retrofitted into SunOS 4.1.3_U1 on the same hardware.

The large drop in throughput of an overloaded server highlights the importance of evaluating the overload behavior of a Web server. Note that it is impossible to evaluate this aspect of Web server performance with current benchmarks that are based on the simple scheme for request generation.

5.4 Throughput under Bursty Conditions

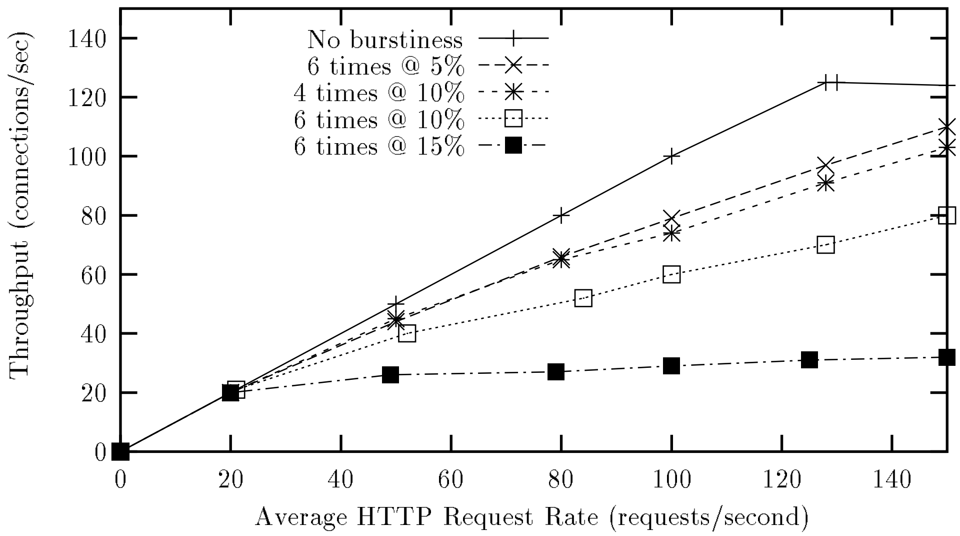

Figure 8: Web server throughput under bursty conditions versus request rate

In Section 3, we point out that one of the drawbacks of the simple traffic generation scheme is the lack of burstiness in the request traffic. A burst in request rate may temporarily overload the server beyond its capacity. Since Figure 7 indicates degraded performance under overload, we were motivated to investigate the performance of a Web server under bursty conditions.

We configured a S-Client with think time values such that it generates bursty request traffic. We characterize the bursty traffic by 2 parameters, a) the ratio between the maximum request rate and the average request rate, and b) the fraction of time for which the request rate exceeds the average rate. Whenever the request rate is above the mean, it is equal to the maximum. The period is 100 seconds. For four different combination of these parameters we varied the average request rate and measured the throughput of the server. We used Testbed-1 in this experiment.

Figure 8 plots the throughput of the Web server versus the average request rate. The first parameter in the label of each curve is the factor a) above, and the second is factor b) above, expressed as a percentage. For example, (6, 5) refers to the case where for 5% of the time the request rate is 6 times the average request rate.

As expected, even a small amount of burstiness can degrade the throughput of a Web server. For the case with 5% burst ratio and peak rate 6 times the average, the throughput for average request rates well below the server's capacity is degraded by 12-20%. In general, high burstiness both in parameter a) and in b) degrades the throughput substantially. This is to be expected given the reduced performance of a server beyond the saturation point in Figure 7.

The bursty traffic model in this experiment is only a crude approximation of actual traffic conditions at a real WWW server. The point of this experiment is to show that the use of S-Clients *enables* the generation of request distributions of complex nature and with high peak rates. This is not possible using a simple scheme for request generation. Moreover, we have shown that the effect of burstiness on server performance is significant.

5.5 Effect of WAN delays

In our next experiment, we use our new method to characterize the effect of WAN delays on Web server throughput. We configured Testbed-2 with 4 S-Clients distributed across two client machines. Each S-Client generates a realistic workload which was derived from Rice University's Computer Science departmental Web server logs. Only requests for static documents were extracted from the log. We use a real workload in this experiment in order to observe the expected secondary effects of excess state in a server, i.e. the effects of the increased memory requirements resulting from WAN conditions on the effective size of the server's main memory file cache. In a control experiment, we use a trivial workload where the S-Clients make requests for the same 1KB file.

The router was configured to introduce a fixed round-trip delay T. We varied T from 0 to 200ms to model typical round-trip delays in the United States portion of the Internet. In this experiment, we do not model the effects of limited path bandwidth; each client-router-server path had a real and simulated bandwidth of 100Mbps. The router always had sufficient CPU to perform its forwarding role without becoming a bottleneck. We measured the throughput of the server as a function of T. Our results for the Apache server are shown as the curve labeled "Apache server throughput" in Figure 9. The request generated by the clients was matched to the capacity of the server in this experiment, i.e., the server was not overloaded.

As we can see, the throughput of the Apache server declines significantly as the network delay increases. With 200ms round-trip delay, the throughput is about 54% lower then the throughput observed with 0ms simulated delay (the LAN case).

To understand the reason behind this throughput drop, we instrumented the server to find out where

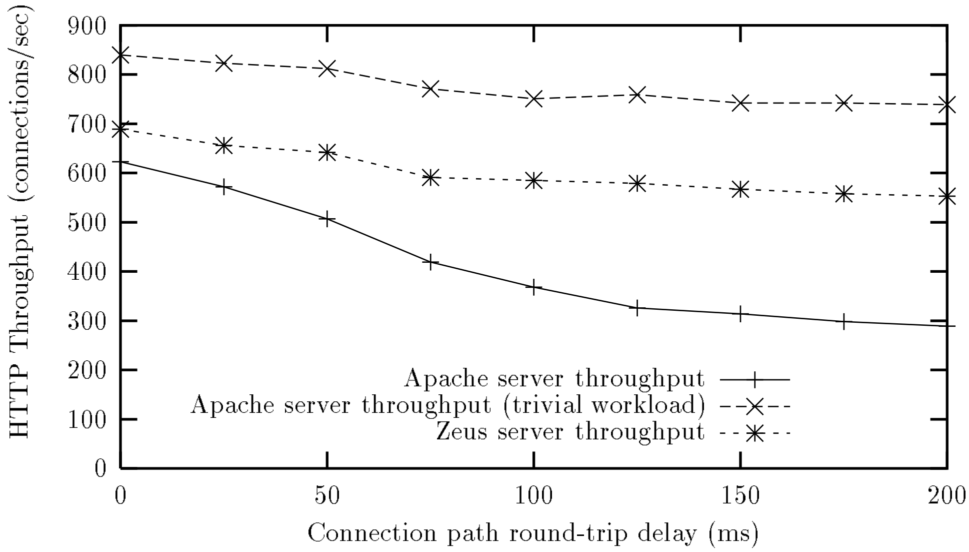

Figure 9: Server throughput versus WAN delays

the system's time was being spent. We noticed that WAN delays induce a large number of simultaneously active connections at the server. This is because with larger round-trip network delays, each TCP (and thus HTTP) connection at the server lasts longer. From queuing theory, we know that for a given service rate, longer service times lead to a larger number of simultaneously active jobs. Thus, with longer HTTP connection times, the average number of simultaneously active connections is larger. This larger number of active connections at the server is directly responsible for the decreased server throughput, for reasons discussed below.

First, the large number of connections increase the memory pressure at the server, due to the per-connection state that needs to be maintained. This state includes per-connection state maintained by Apache (Apache uses a UNIX process for each connection) and the per-connection state maintained by the OS kernel. The latter includes the socket buffers, the size of which increases with the network delay.

Second, with a process-per-connection server such as Apache, a large number of simultaneously active connections increases the context-switching overhead in the server system. The system continuously switches between a large number of server processes while handling the HTTP requests on various connections.

The total number of active connections and Apache server processes as a function of network delay is shown in Figure 10. The solid curve (labeled "Active connections") shows the average number of HTTP connections that are in the TCP's ESTABLISHED state for a particular value of network delay. The dotted curve depicts the total number of Apache server processes. As we can see, the number of active connections and server processes increases rapidly with network delay. The total number of server processes levels out

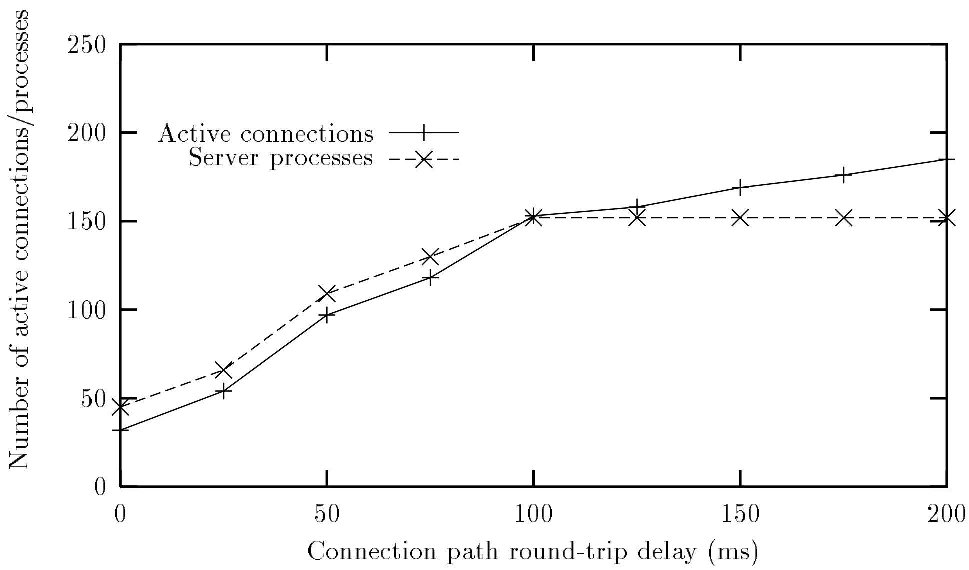

Figure 10: Variation of # of active connections and server processes with WAN delays.

at 152 active processes. This corresponds to an Apache configuration value which limits the total number of server processes to the default value of 152. The active connection count continues to increase even beyond this point. Connections that do not have a process assigned to them wait to be accepted in the accept queue of the Web server's listen socket.

The drop in Apache's throughput in Figure 9 up to about 100ms network delay is primarily due to increased memory pressure and context switching in the system as Apache increases its number of server processes. Each Apache process has an incremental memory requirement of about 500KB. The memory consumed by these server processes decreases the total amount of memory available for the main memory file cache, which in turns decreases throughput. Instrumentation of the kernel confirmed this; the size of the file cache decreases from about 97MB at 0ms delay to roughly 28MB at 100ms delay. Since the working set of our workload is about 80MB in size, this causes the server system to move from a CPU bound scenario, where most requests are satisfied from the file cache, to a disk-bound scenario, where frequent file cache misses degrade server throughput significantly.

Beyond 100ms of network delay, Apache does not increase the number of server processes further since its configuration limit was reached. The subsequent decrease in server throughput is due to increased memory pressure in the system as the amount of data dedicated to socket buffers increases. At 200ms delay, the total memory consumption of all network buffers is about 3.6MB. Note that this contribution to memory pressure is likely to increase further in the near future, as networks with very high bandwidth-delay products, such as intercontinental high-speed networks, are deployed.

The dotted curve in Figure 9, labeled “Apache server throughput (trivial workload)”, shows the throughput of Apache with increasing WAN delay when it faces a trivial workload, i.e., all requests to the server are for the same 1K file. Here, all requests hit in the file cache. The purpose of plotting this curve is to separate out the effect of factors other than the reduced main memory cache hit rate. There is roughly a 12% drop in throughput over the range of network delays. We attribute this throughput drop primarily to the context-switching overhead as the number of Apache processes increases.

We also measured the throughput of the Zeus server with increasing WAN delays. In contrast to Apache, Zeus is a single-process, event-driven server. It does not use a separate process for every connection; instead all connections are handled within the same server process. Our objective in benchmarking Zeus was to see how its performance is impacted by WAN delays given that it has reduced per-connection memory requirements and less context-switching overhead.

Our results are shown in the curve labeled “Zeus server throughput” in Figure 9. There is a decrease in server performance of roughly 20% over the range of network delays. Instrumentation of the system shows that the primary cause is poor scalability of the select() system call with respect to the total number of connections being simultaneously handled. Event-driven servers like Zeus makes heavy use of select() to manage multiple HTTP connections on a non-blocking fashion. This effect (and a solution to the poor performance of select()) are covered in detail elsewhere [Banga and Mogul 1998; Banga *et al.* 1998].

A caveat is in order here. From the difference in throughput degradation rates of Apache and Zeus, the reader might conclude that event-driven servers like Zeus are a partial solution for the performance problems of Apache in the presence of WAN delays. However, event-driven servers may have performance problems of their own, which degrade their throughput for disk-bound workloads.

Being single-threaded, the lack of support for non-blocking disk I/O [Banga *et al.* 1998] in many operating systems (including FreeBSD) causes purely event-driven servers to be blocked for hundreds of milliseconds on each disk I/O. During this period, no processing happens for any connection being handled by the server. Thus, in the presence of memory cache misses, single-threaded event-driven servers effectively serve documents at disk speed (60–120 requests/second).

Zeus can be configured to use multiple processes to overcome this problem; however, the added memory requirements and context-switching overheads may then results in performance problems similar to those experienced by Apache. Unfortunately, we were unable to verify this experimentally, as multi-process configurations of Zeus are not supported under FreeBSD.

In summary, wide-area network delays have a significant impact on the performance of Web servers. This impact remains unevaluated by standard benchmarks, which are based on LAN test-beds.

5.6 Effect of packet losses

Our final experiment attempts to measure the effect of packet losses on Web server throughput. We configured the router of Testbed-2 to drop packets. The router was configured to cause packet losses with a rate varying from 0% to 5%. The round-trip delay between the client and the server was constant at 100ms. As in the previous experiment, the generated request rate was matched to the capacity of the server.

Our results indicate that packet losses increase the amount of state at the server, due to an increase in the number of active connections. The reason is that, like WAN delays, packet losses increase the duration of an HTTP transaction. At a 5% packet loss rate, the number of simultaneously active connections that we observed at the server was about 320, compared to 149 without losses. At this loss rate, the maximum throughput of the server is 237 requests/sec. This corresponds to a degradation of about 36% from the throughput measured with no losses and 100ms network delay.

This result shows that packet losses, like WAN delays, have a significant impact on Web server performance, which remains unevaluated by existing benchmarks.

6 RELATED WORK

Operating system researchers have devoted much effort to characterizing and improving Web server performance. In an early study, Mogul analyzed Web server performance in the context of the 1994 California election server [Mogul 1995a; Mogul 1995b]. This evaluation was performed by observing server behavior while facing a live workload. Another early study was performed by McGrath at NCSA [McGrath 1995]. This study used an HTTP request generation method very similar to the simple method that we described in Section 3.

Since then, there have been a large number of efforts to evaluate and improve Web and proxy server performance. These include efforts to support zero-copy I/O [Pai *et al.* 1999], to improve the architecture of servers [Chankhunthod *et al.* 1996], to reduce the number of system calls in the typical I/O path [Hu *et al.* 1997a; Hu *et al.* 1998; Hu *et al.* 1997b], and to quantify the interaction of Web servers and operating systems [Almeida *et al.* 1996; Yates *et al.* 1997; Schechte and Sutaria 1996]. Kaashoek et al. have developed a prototype high-performance Web server based on a customized operating system that is tailored specially for servers [Kaashoek *et al.* 1997; Kaashoek *et al.* 1996]. The request generation method used in all these studies is similar to the simple method of Section 3.

There is also much existing work towards characterizing the invariants in WWW traffic. Arlitt and Williamson [Arlitt and Williamson 1996] characterized several aspects of Web server workloads such as request file type distribution, transfer sizes, locality of reference in the requested URLs and related statistics. Crovella and Bestavros [Crovella and Bestavros 1996] looked at self-similarity in WWW traffic. The invariants reported by these efforts have been used in evaluating the performance of Web servers, and the many methods proposed by researchers to improve WWW performance.

Formal Web server benchmarking efforts have relatively recent origins. WebStone [Mindcraft 1998] was one of the earliest Web server benchmarks. WebStone is similar to the simple scheme described in Section 3 and suffers from its limitations. Other early efforts [Web66 1996; Mecklermedia 1996] were similar to WebStone. SPECWeb96 [SPEC 1996] is a standardized Web server benchmark from SPEC, with a workload derived from the study of typical servers on the Internet. The request generation method of this benchmark is also similar to that of the simple scheme and suffers from the same limitations.

The Wisconsin Proxy Benchmark (WPB) described by Almeida and Cao was designed to specifically measure the performance of a proxy server, as opposed to a Web server [Almeida and Cao 1998]. The first version of this benchmark uses a variant of the simple request generation method described in Section 3 that has been parameterized to allow the generation of requests to a long-tailed distribution of document sizes.

One aim of the WPB work was to measure the effect of slow modem connections on proxy server throughput. For this, a modem emulator implemented in the client machine kernel spaces out IP packets according to the delay and bandwidth characteristics of modem links. This study was, however, mainly concerned with the effect of proxy performance on the response time seen by a client connected to the Internet via a slow modem link. The effect on proxy performance of well-connected clients that access the proxy via long-delay, but relatively high-bandwidth links was not studied. In addition, the overload behavior of proxy servers was not evaluated. A new version of the WPB, which uses our S-Client method to generate requests, is currently being used to study proxy performance under overload.

Mosberger and Jin describe httperf, a tool that attempts to provide a comprehensive benchmark for Web server measurement [Mosberger and Jin 1998]. httperf supports HTTP/1.1-style persistent connections, request pipelining, and "chunked" transfer-encoding [Fielding *et al.* 1997; Nielsen *et al.* 1997]. An important concern in the design of this tool is the complete separation of the issues related to the actual generation of HTTP calls from those related to the specific workload and measurements that should be used. httperf uses a specially constructed method, which is similar to and partly based on the S-Client architecture, for generating requests that exceed the capacity of the Web server being measured. This tool, however, does not model the effects of WANs on server performance.

Banga and Mogul describe the effect of WAN characteristics on the performance of event-driven Web servers [Banga and Mogul 1998]. Their measurements were performed in the context of a live workload, and by using a specially constructed benchmark which used a number of slow client connections to artificially increase the number of simultaneously active connections. While useful, this benchmarking approach does not generalize to modeling arbitrary WANs and combinations of networks. This approach also fails to accurately model the exact timing of packets and packet losses on WANs.

In summary, most Web benchmarks that we know of evaluate Web Servers only by modeling aspects of server workloads that pertain to request file types, transfer sizes and locality of reference in URLs requested. No benchmark, other than those that are based on our method, attempts to accurately model the effects of request overloads, or WAN characteristics on server performance. Our method based on S-Clients enables the generation of HTTP requests with burstiness and high rates. Furthermore, the use of a controlled delay/loss inducing router allows us to model the effect of WANs on server performance. Our work is intended to complement the workload characterization efforts to evaluate Web servers.

7 CONCLUSION

This paper examines pitfalls that arise when generating synthetic Web server workloads in a testbed consisting of a small number of client machines. It exposes limitations of the simple request generation scheme that underlies state-of-the-art Web server benchmarks. In particular, current benchmarks do not drive a server beyond its capacity and they do not model the effect of delays and packet losses in the network.

We propose and evaluate a new strategy that addresses these problems by using a novel client process architecture (S-Clients) and a WAN-modeling router. S-Clients allow the generation of request loads that exceed the server's capacity. The router makes it possible to measure the effect of packet losses and delays on the server's performance.

Initial experience in using this method to evaluate widely used Web servers indicates that measuring server performance under overload, bursty traffic conditions, and in the presence of WAN packet losses and delays, gives new and important insights in Web server performance. Our new methodology enables the evaluation of this important aspect of Web server performance.

Source code and additional technical information about our benchmarking method can be found at http://www.cs.rice.edu/CS/Systems/Web-measurement/.

Acknowledgments

We are grateful to the anonymous referees for their detailed comments, which helped us to improve this paper. This work was supported in part by NSF Grants CCR-9803673, CCR-9503098, and by Texas TATP Grant 003604.

REFERENCES

ACME (1998), "ACME Laboratories, thttpd," http://www.acme.com/software/thttpd/.

Almeida, J., V. Almeida, and D. Yates (1996), "Measuring the Behavior of a World-Wide Web Server," Technical Report TR-96-025, Boston University, CS Dept., Boston MA.

Almeida, J. and P. Cao (1998), "Measuring Proxy Performance with the Wisconsin Proxy Benchmark," Presented at the 3rd Web Caching Workshop, Manchester, England.

Apache (1998), "Apache HTTP Server Project," http://www.apache.org/.

Arlitt, M. F. and C. L. Williamson (1996), "Web Server Workload Characterization: The Search for Invariants," In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, pp. 126–137.

Bach, M. J. (1986), *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ.

Banga, G. (1998), "The Design and Implementation of a New Network Subsystem Architecture for Server Systems," Master's thesis, Rice University, Houston, TX.

Banga, G., F. Douglis, and M. Rabinovich (1997), "Optimistic Deltas for WWW Latency Reduction," In *Proceedings of the 1997 USENIX Annual Technical Conference*, Anaheim, CA, pp. 289–303.

Banga, G., P. Druschel, and J. C. Mogul (1998), "Better operating system features for faster network servers," In *Proceesings of the Workshop on Internet Server Performance*, Madison, WI, pp. 69–79.

Banga, G., P. Druschel, and J. C. Mogul (1999), "Resource containers: A new facility for resource management in server systems," In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA (*to appear*).

Banga, G. and J. C. Mogul (1998), "Scalable kernel performance for Internet servers under realistic loads," In *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, pp. 1–12.

Bestavros, A., R. Carter, M. Crovella, C. Cunha, A. Heddaya, and S. Mirdad (1995), "Application-Level Document Caching in the Internet," Technical Report TR-95-002, Boston University, Department of Computer Science, Boston MA.

Braun, H. and K. Claffy (1994), "Web Traffic Characterization: An Assessment of the Impact of Caching Documents from NCSA's Web Server," In *Proceedings of the Second International WWW Conference*, Chicago, IL, pp. 1007–1027.

Chankhunthod, A., P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell (1996), "A Hierarchical Internet Object Cache," In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, pp. 153–163.

Compaq (1998), "Compaq Computer Corporation, Digital UNIX Tuning Parameters for Web Servers," http://www.digital.com/info/internet/document/ias/tuning.html.

Crovella, M. and A. Bestavros (1996), "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, pp. 160–169.

Cunha, C., A. Bestavros, and M. Crovella (1995), "Characteristics of WWW Client-Based Traces," Technical Report TR-95-010, Boston University, CS Dept., Boston MA.

Custer, H. (1993), *Inside Windows NT*, Microsoft Press, Redmond, WA.

Druschel, P. and G. Banga (1996), "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Seattle, WA, pp. 261–276.

Fielding, R., J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee (1997), "Hypertext Transfer Protocol – HTTP/1.1," RFC 2068.

Fox, A., S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier (1997), "Cluster-Based Scalable Network Services," In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, San Malo, France, pp. 78–91.

Hu, J. C., S. Mungee, and D. C. Schmidt (1998), "Techniques for developing and measuring high-performance Web servers over ATM networks," In *Proceedings of the IEEE Infocom Conference*, San Francisco, CA.

Hu, J. C., I. Pyrali, and D. C. Schmidt (1997a), "Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks," In *Proceedings of the 2nd Global Internet Conference*, Phoenix, AZ.

Hu, Y., A. Nanda, and Q. Yang (1997b), "Measurement, analysis, and performance improvement of the Apache web server," Technical Report TR 1097-0001, University of Rhode Island, ECE Dept.

Kaashoek, M. F., D. R. Engler, G. R. Ganger, H. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie (1997), "Application performance and flexibility on Exokernel systems," In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Saint-Malo, France, pp. 52–65.

Kaashoek, M. F., D. R. Engler, G. R. Ganger, and D. A. Wallach (1996), "Server Operating Systems," In *Proceedings of the 1996 ACM SIGOPS European Workshop*, Connemara, Ireland, pp. 141–148.

Kwan, T. T., R. E. McGrath, and D. A. Reed (1995), "User access patterns to NCSA's World-wide Web server," Technical Report UIUCDCS-R-95-1934, Dept. of Computer Science, Univ. IL.

Maltzahn, C., K. J. Richardson, and D. Grunwald (1997), "Performance Issues of Enterprise Level Web Proxies," In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, pp. 13–23.

McGrath, R. E. (1995), "Performance of Several HTTP Demons on an HP 735 Workstation," http://www.ncsa.uiuc.edu/InformationServers/Performance/V1.4/report.html.

McKusick, M. K., K. Bostic, M. J. Karels, and J. S. Quarterman (1996), *The Design and Implementation* *of the 4.4BSD Operating System*, Addison-Wesley Publishing Company.

Mecklermedia (1996), "Mecklermedia Corporation, *WebServer Compare*," http://webcompare.iworld.com/.

Mindcraft (1998), "Mindcraft Inc., *WebStone*," http://www.mindcraft.com/webstone/.

Mogul, J. C. (1993), "Observing TCP Dynamics in Real Networks," In *Proceedings of the ACM SIGCOMM '92 Conference*, pp. 281–292.

Mogul, J. C. (1995a), "Network Behavior of a Busy Web Server and its Clients," Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA.

Mogul, J. C. (1995b), "Operating System Support for Busy Internet Servers," In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA.

Mogul, J. C., F. Douglis, A. Feldmann, and B. Krishnamurthy (1997), "Potential benefits of delta encoding and data compression for HTTP," In *Proceedings of the ACM SIGCOMM '97 Conference*, Cannes, France, pp. 181–194.

Mogul, J. C. and K. K. Ramakrishnan (1997), "Eliminating Receive Livelock in an Interrupt-driven Kernel," *ACM Transactions on Computer Systems 15*, 3, 217–252.

Mosberger, D. and T. Jin (1998), "httperf—A Tool for Measuring Web Server Performance," In *Proceedings of the Workshop on Internet Server Performance*, Madison, WI, pp. 59–68.

Nielsen, H., J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley (1997), "Network performance effects of HTTP/1.1, CSS1, and PNG," In *Proceedings of the ACM SIGCOMM '97 Conference*, Cannes, France, pp. 155–166.

Padmanabhan, V. N. and J. C. Mogul (1994), "Improving HTTP Latency," In *Proceedings of the Second International WWW Conference*, Chicago, IL, pp. 995–1005.

Pai, V. S., P. Druschel, and W. Zwaenepoel (1999), "IO-Lite: A unified I/O buffering and caching system," In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA (*to appear*).

Postel, J. B. (1981), "Transmission Control Protocol," RFC 793.

Ramakrishnan, K. K. (1992), "Scheduling Issues for Interfacing to High Speed Networks," In *Proceedings of the IEEE Global Telecommunications Conference*, Orlando, FL, pp. 622–626.

Schechte, S. E. and J. Sutaria (1996), "A Study of the Effects of Context Switching and Caching on HTTP Server Performance," http://www.eecs.harvard.edu/~stuart/Tarantula/FirstPaper.html.

Seltzer, M. and J. Gwertzman (1995), "The Case for Geographical Pushcaching," In *Proceedings of the 1995 Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, pp. 51–55.

Shenker, S., L. Zhang, and D. Clark (1990), "Some Observations on the Dynamics of a Congestion Control Algorithm," *ACM Computer Communication Review 20*, 4, 30–39.

Spasojevic, M., M. Bowman, and A. Spector (1994), "Using a Wide-Area File System Within the World-Wide Web," In *Proceedings of the Second International WWW Conference*, Chicago, IL.

SPEC (1996), "The Standard Performance Evaluation Corporation, *SPECWeb96*," http://www.specbench.-org/osg/web96/.

Stevens, W. (1996), *TCP/IP Illustrated Volume 3*, Addison-Wesley, Reading, MA.

Web66 (1996), "Web66 GStone Testing Laboratory Preliminary Report," http://web66.coled.umn.edu/gstone/.

Williams, S., M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox (1996), "Removal Policies in Network Caches for World-Wide Web Documents," In *Proceedings of the ACM SIGCOMM '96 Conference*, Palo Alto, CA, pp. 293–305.

Wright, G. and W. Stevens (1995), *TCP/IP Illustrated Volume 2*, Addison-Wesley, Reading, MA.

Yates, D., V. Almeida, and J. Almeida (1997), "On the interaction between an operating system and Web server," Technical Report TR-97-012, Boston University, CS Dept., Boston MA.

Zeus (1998), "Zeus," http://www.zeus.co.uk/.