Measuring the Capacity of a Web Server*

Gaurav Banga and Peter Druschel †

Department of Computer Science
Rice University
Houston, TX 77005

Abstract

The widespread use of the World Wide Web and related applications places interesting 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. Unfortunately, accurate generation of such traffic in a testbed of limited scope is not trivial. In particular, the commonly used approach is unable to generate client request-rates that exceed the capacity of the server being tested even for short periods of time. 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. Finally, we use the proposed method to measure the performance of a Web server.

1 Introduction

The explosive growth in the use of the World Wide Web has resulted in increased 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 [5, 6, 7, 23, 31], HTTP protocol enhancements [4, 20, 25, 18], better HTTP servers and proxies [2, 33, 7] and server OS implementations [16, 17, 10, 24].

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 [3, 5, 6, 8, 9, 12]. Some researchers have tried to evaluate the performance of Web servers and proxies using real workloads directly [13, 15]. 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 [26, 28, 29, 30, 1]. 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 testbed 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 naive 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. In addition, a naive scheme generates client traffic that has little resemblance in its temporal characteristics to real-world Web traffic. Moreover, there are fundamental differences between the delay and loss characteristics of WANs and the LANs used in testbeds. Both 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

*This paper will appear in the Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, Dec 1997.

†This work was supported in part by National Science Foundation Grant CCR-9503098

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 expect that this request generation methodology, in conjunction with a representative HTTP request data set like the one used in the SPECWeb benchmark [26] 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 a Web server 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 give a brief overview of 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 [14, 32] based network subsystem. The working of many other implementations of TCP/IP, such as those found in Unix System V and Windows NT, 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 text, images) of the page.

Figure 1 shows the sequence of events in the connection establishment phase of an HTTP transaction. When starting, a 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 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.

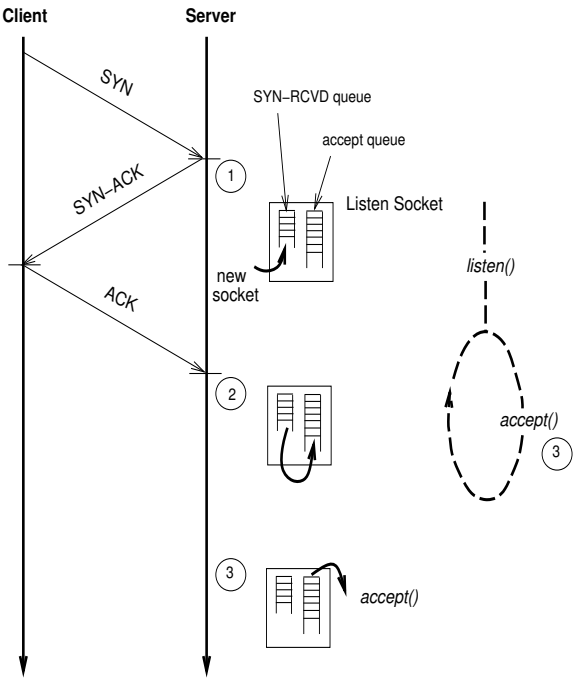

Figure 1: HTTP Connection Establishment Timeline

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].

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

[1] HTTP 1.1 supports persistent connections, but most browsers and servers today do not use HTTP 1.1.

[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.

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 with many traditional implementations of TCP/IP that limits the throughput of a Web server. Many BSD based 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 and ship their systems with large maximum values (e.g. Digital Unix 32767, Solaris 1000). In Section 3, we will see how this fact interacts with WWW request generation.

[4]This TIME-WAIT period should be set equal to twice the Maximum Segment Lifetime (MSL) of a packet on the Internet (RFC 793[21] specifies the MSL as 2 minutes, but many implementations use a much shorter value.)

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 testbed consisting of a limited number of client machines. For reasons of cost and ease of control, one would like to use a small number of client machines to simulate a large Web 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[5] 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 [26, 28, 29, 30].

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 user's 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 [8], and with peak rates exceeding the average rate by factors of 8 to 10 [15, 27]. 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

[5]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.

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 lockstep with the server. That is, the rate of generated requests never exceeds the capacity of the server.

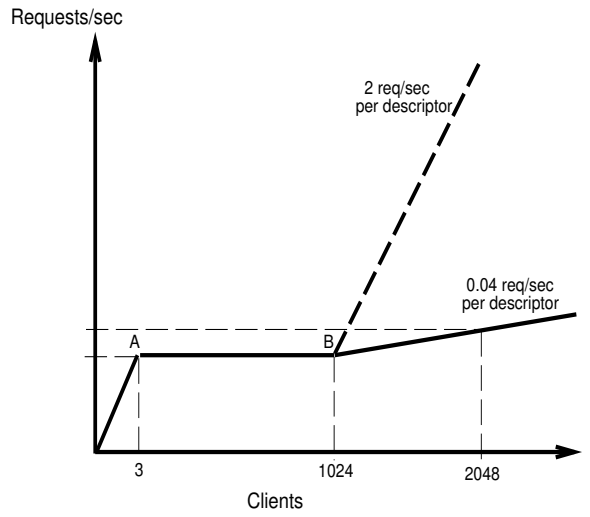

Figure 2: Request Rate versus no. of Clients

Consider a Web server that is subjected to HTTP requests from an increasing number of clients in a testbed 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 sees 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 net connection request rate of all the clients remains equal to the throughput of the server.

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 this is 3 requests in 75 seconds.) The behavior is depicted 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 15000 client processes ($(1100-100)/(3/75)$) beyond a number equal to 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, we need 64151 processes ($1.5 \times 32767 + 15000$) 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 at all 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. It is important to evaluate Web server performance under overload. For instance, it is a well known fact that many Unix and non-Unix based network subsystems suffer from poor overload behavior [11, 19]. Under heavy network load these interrupt-driven systems can enter a state called *receiver-livelock*[22]. 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 programs (in this case the Web server).

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 Additional Problems

The WAN-based Web has network characteristics that differ from the LANs on which Web servers are usually evaluated. 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 which are known to cause long SYN-RCVD queues in the server's listening socket. Also, packet losses due to congestion are absent in LAN-based testbeds. Maltzahn et al. [13] discovered a large difference in Squid proxy performance from the idealized numbers reported in [7]. A lot of this degradation is attributed to such WAN effects, which tend to keep server

resources such as memory tied up for extended periods of time.

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 [30] 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[15] 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. We will return to the issue of client bottlenecks in Section 4, and show how to account for client resource constraints in setting up a testbed.

4 A Scalable Method for Generating HTTP Requests

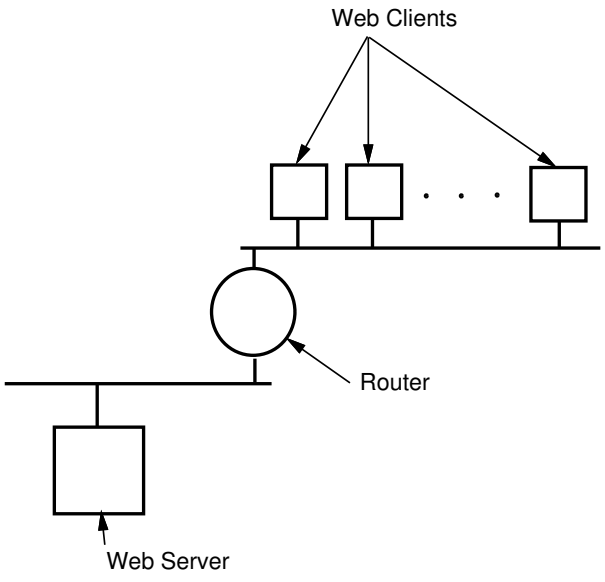

Figure 3: Testbed Architecture

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 [5, 6, 7, 23, 31], and can easily be used in conjunction with it.

4.1 Basic Architecture

The basic architecture of our testbed 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. If WAN effects are to be evaluated, the client machines should be connected to the server through a router that has sufficient capacity to carry the maximum traffic anticipated. The purpose of the router is to simulate WAN delays by introducing an artificial delay in the router's forwarding mechanism.

4.2 S-Clients

A S-Client 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.

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 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 on this socket. In the former case, the process sends a HTTP request on the newly established connection, hands off this connection

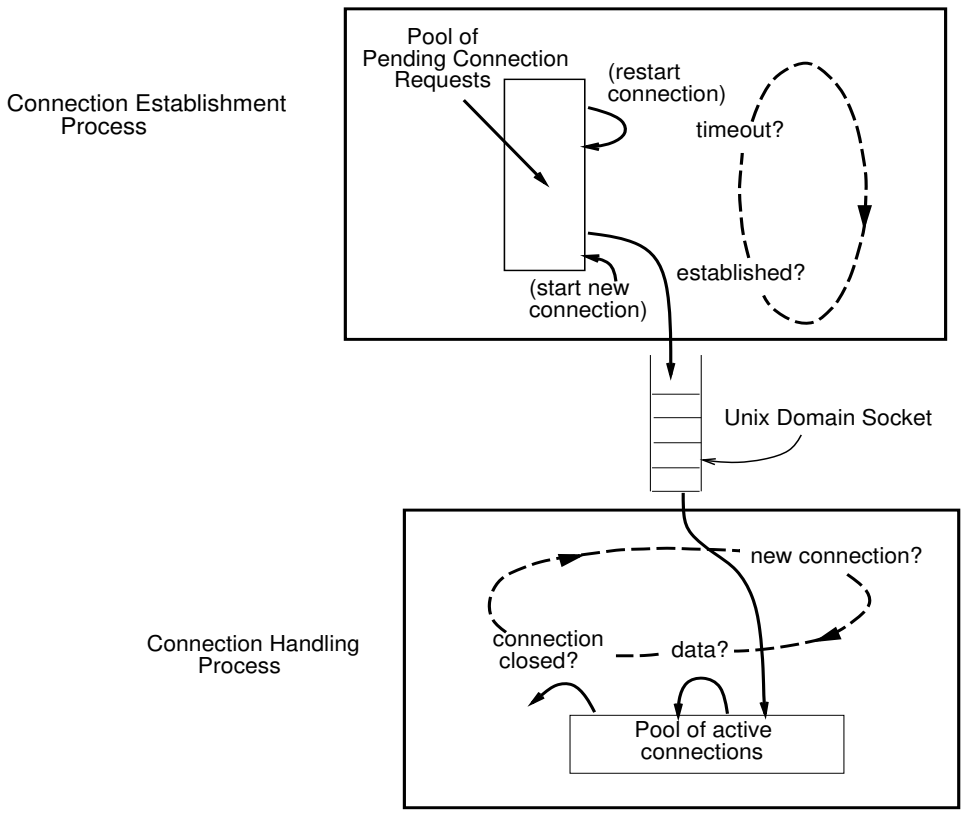

Figure 4: A Scalable Client

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 effect, it prematurely aborts TCP's connection establishment timeout period. The close merely releases socket resources in the OS.

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 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.

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 unchanged from the same curve when using 2 client machines. 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 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. Because of this, the number of machines needed to generate a certain request rate may be higher in this case.

4.4 Think Time Distributions

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 [8].

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 measure the request generation limitations of the naive approach and evaluate the S-Client based request generation method proposed in Section 4. We also measure the performance of a Web server using our method.

5.1 Experimental Setup

All experiments were performed in a testbed 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 Mbit/s 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. In our experiments we used no artificial delay in the router connecting the clients and the server. We have not yet quantitatively evaluated the effect of WAN delays on server performance.

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 and decreased the TIME-WAIT period to 3 seconds.

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. We ran an increasing

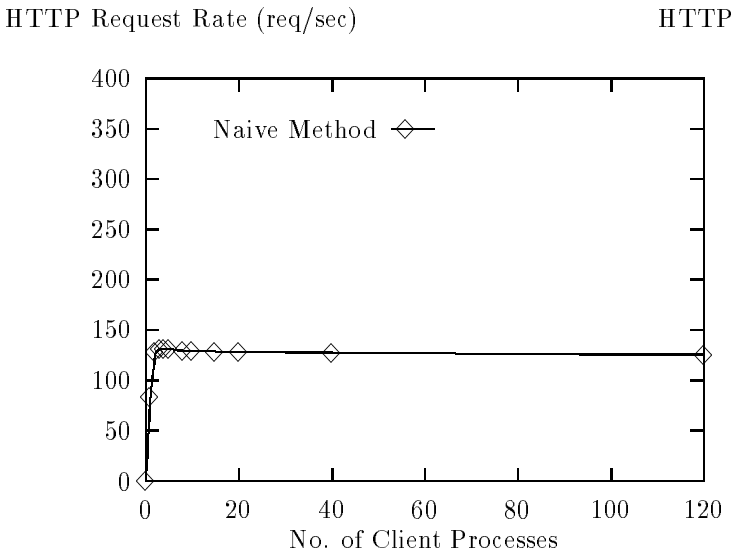

Figure 5: Request rate versus number of clients

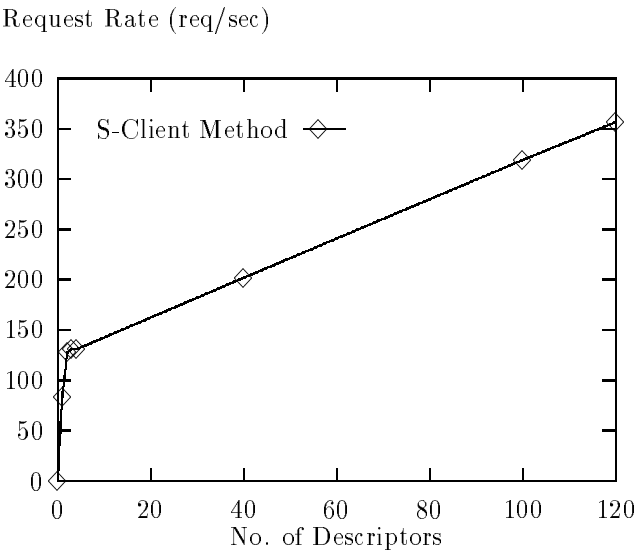

Figure 6: Request rate versus number of descriptors

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 measured 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 connection establishment timeout period T set to $500ms$. The same file was requested as in the case of the simple clients.

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 per 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 per 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 terms of transactions per second. The same 1294 byte file as before was used in this test.

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 fall 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 (the full accept queue).

The slope of the throughput drop corresponds to about 325 usec worth of processing time per SYN packet. While this may seem large, it is consistent with our ob-

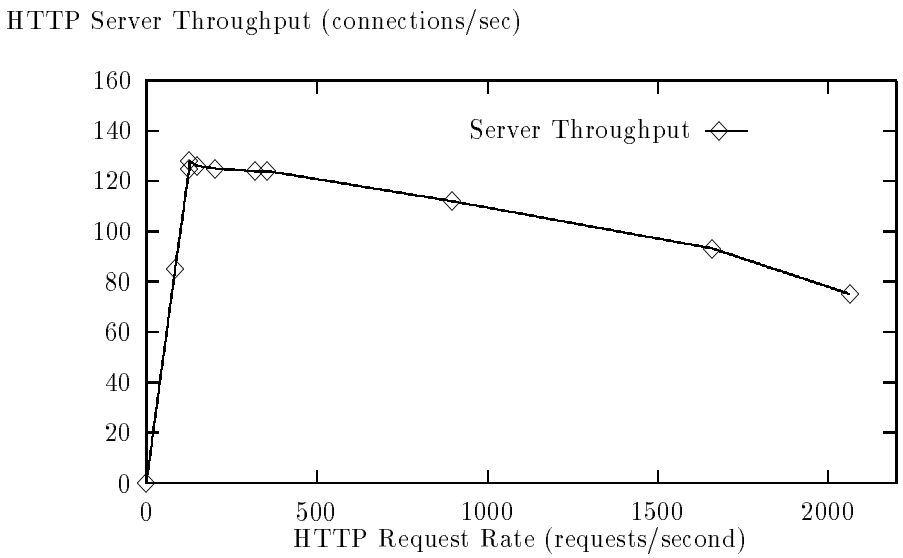

Figure 7: Web server throughput versus request rate

servation 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.

This 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 bench marks that are based on the simple scheme for request generation.

5.4 Throughput under Bursty Conditions

In Section 3, we point out that one of the drawbacks of the naive 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 times 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 exceeded 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. 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.

Note that our workload only approximates what one would see on the real WWW. 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 such burstiness on server performance is significant.

6 Related Work

There is much existing work towards characterizing the invariants in WWW traffic. Most recently, Arlitt and Williamson [3] 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 [8] looked at Self-Similarity in WWW traffic. The invariants reported by these efforts have been used in evaluating the

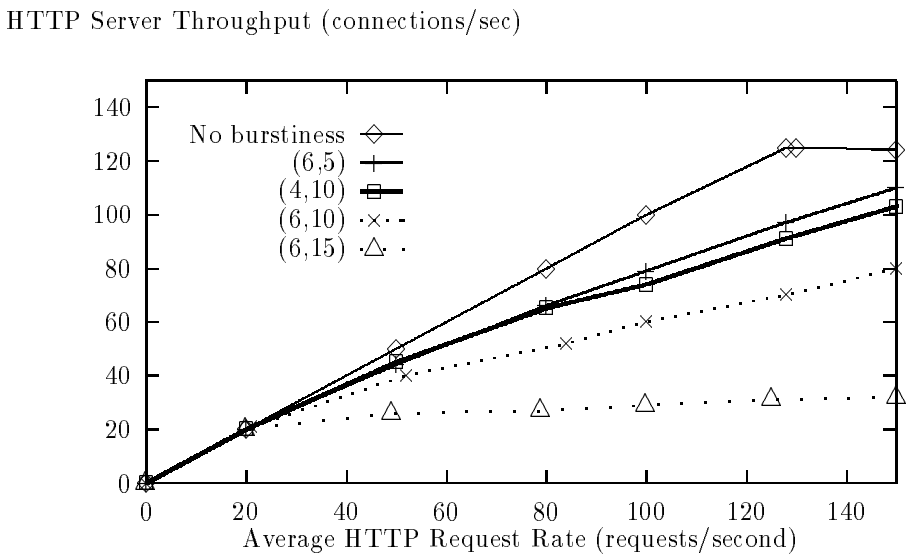

Figure 8: Web server throughput under bursty conditions versus request rate

performance of Web servers, and the many methods proposed by researchers to improve WWW performance.

Web server benchmarking efforts have much more recent origins. SGI's WebStone [30] was one of the earliest Web server benchmarks and is the de facto industry standard, although there have been several other efforts [28, 29]. WebStone is very similar to the simple scheme that we described in Section 3 and suffers from its limitations. Recently SPEC has released SPECWeb96 [26], which is a standardized Web server benchmark with a workload derived from the study of some typical servers on the Internet. The request generation method of this benchmark is also similar to that of the simple scheme and so it too suffers from the same limitations.

In summary, all 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 we know of attempts to accurately model the effects of request overloads on server performance. Our method based on S-Clients enables the generation of HTTP requests with burstiness and high rates. It is intended to complement the workload characterization efforts to evaluate Web servers.

7 Conclusion

This paper examines pitfalls that arise in the process of generating synthetic Web server workloads in a testbed consisting of a small number of client machines. It exposes the limitations of the simple request generation scheme that underlies state-of-the-art Web server benchmarks. We propose and evaluate a new strategy that addresses these problems using a set of specially constructed client processes. Initial experience in using this method to evaluate a typical Web server indicates that measuring Web server performance under overload and bursty traffic conditions gives new and important insights in Web server performance. Our new methodology enables the generation of realistic, bursty HTTP traffic and thus the evaluation of an important performance aspect of Web servers.

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

References

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

[2] Apache. http://www.apache.org/.

[3] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

[4] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proceedings of the 1997 Usenix Technical Conference*, Jan. 1997.

[5] A. Bestavros, R. Carter, M. Crovella, C. Cunha, A. Heddaya, and S. Mirdad. Application-Level Document Caching in the Internet. Technical Report TR-95-002, Boston University, CS Dept., Boston MA, Feb. 1995.

[6] H. Braun and K. Claffy. 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, Oct. 1994.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Technical Conference*, Jan. 1996.

[8] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of the ACM SIGMETRICS '96 Conference*, Philadelphia, PA, Apr. 1996.

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

[10] DIGITAL UNIX Tuning Parameters for Web Servers. http://www.digital.com/info/internet/document/ias/tuning.html.

[11] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proc. 2nd Symp. on Operating Systems Design and Implementation*, Seattle, WA, Oct. 1996.

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

[13] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the ACM SIGMETRICS '97 Conference*, Seattle, WA, June 1997.

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

[15] J. C. Mogul. Network behavior of a busy web server and its clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, CA, 1995.

[16] J. C. Mogul. Operating system support for busy internet servers. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[17] J. C. Mogul. Personal communication, Oct. 1996.

[18] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the SIGCOMM '97 Conference*, Cannes, France, Sept. 1997.

[19] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. of the 1996 Usenix Technical Conference*, pages 99–111, 1996.

[20] V. N. Padmanabhan and J. C. Mogul. Improving HTTP Latency. In *Proceedings of the Second International WWW Conference*, Chicago, IL, Oct. 1994.

[21] J. B. Postel. Transmission Control Protocol. RFC 793, Sept. 1981.

[22] K. K. Ramakrishnan. Scheduling issues for interfacing to high speed networks. In *Proc. Globecom'92 IEEE Global Telecommunications Conference*, pages 622–626, Orlando, FL, Dec. 1992.

[23] M. Seltzer and J. Gwertzman. The Case for Geographical Pushcaching. In *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[24] Solaris 2 TCP/IP. http://www.sun.com/sunsoft/solaris/networking/tcpip.html.

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

[26] SPECWeb96. http://www.specbench.org/osg/web96/.

[27] W. Stevens. *TCP/IP Illustrated Volume 3*. Addison-Wesley, Reading, MA, 1996.

[28] Web 66. http://web66.coled.umn.edu/gstone/info.html.

[29] WebCompare. http://webcompare.iworld.com/.

[30] WebStone. http://www.sgi.com/Products/WebFORCE/Resources/res_webstone.html.

[31] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the SIGCOMM '96 Conference*, Palo Alto, CA, Aug. 1996.

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

[33] Zeus. http://www.zeus.co.uk/.