Telecommunication Systems 16:3,4, 361–378, 2001

Web Server Performance Modeling

R.D. VAN DER MEI [*], R. HARIHARAN [**] and P.K. REESER
AT&T Labs, 200 Laurel Avenue, Middletown, NJ 07748, USA

Abstract. The advent of Web technology has made Web servers core elements of future communication networks. Although the amount of traffic that Web servers must handle has grown explosively during the last decade, the performance limitations and the proper tuning of Web servers are still not well understood. In this paper we present an end-to-end queueing model for the performance of Web servers, encompassing the impacts of client workload characteristics, server harware/software configuration, communication protocols, and interconnect topologies. The model has been implemented in a simulation tool, and performance predictions based on the model are shown to match very well with the performance of a Web server in a test lab environment. The simulation tool forms an excellent basis for development of a Decision Support System for the configuration tuning and sizing of Web servers.

Keywords: World Wide Web, HTTP, Web server, httpd, performance, throughput, delay, response time, blocking, TCP/IP, Internet, intranets

1. Introduction

Over the past few years, the World Wide Web (WWW) has experienced tremendous growth, which is not likely to slow down in the near future. The explosion of Internet Commerce service offerings (e.g., [2]) has insured that the “Web” will remain at the center of mainstream communications. Furthermore, the recent emergence of Internet Telephony (IT) service offerings has brought the heretofore-separate world of the Internet into the realm of traditional telecommunications. IT services range from simple “click-to-dial” offerings that use the Internet for voice call setup (e.g., [3]) to end-to-end voice communications that use the Internet for packetized voice transport (e.g., [4]).

At the heart of most Internet Commerce and Telephony service offerings is the Web server. Web servers, which are typically based on the Hypertext Transfer Protocol (HTTP) running over TCP/IP, are expected to perform millions of transaction requests per day at an “acceptable” Quality of Service (QoS) level in terms of transaction throughput (connect and error rates) and latency (packet transfer and response times) experienced by the end users. To cope with the increasing volume of transaction requests, as well as the increasing demands of real-time voice communications, a thorough understanding of the performance capabilities and limitations of HTTP Web servers is crucial.

Web server performance is a complicated interplay between a variety of components, such as server hardware platform, Web server software, server operating system,

[*] Current work address: KPN Research, Leidschendam, The Netherlands.
[**] Current work address: Sun Microsystems, Austin, TX, USA.

network bandwidth, file sizes, caching, etc. Experience has taught that the performance of Web servers can be increased tremendously by proper tuning of the components of the server. In order to properly configure these different components, it is crucial to understand how these components interact, and how they impact the end-to-end performance.

To compare the performance of different Web server platforms, several benchmarking tools have been brought to the market (e.g., [15]). These tools typically “fire off” a large number of transaction requests and measure the responsiveness of the server. Although these tools are certainly useful to compare the performance of different server platforms, there are a number of drawbacks. First, most benchmarking tools to some extent consider the Web server as a “black box”, and as such fail to provide insight into which of the components of the server are the performance limiting factors for a given parameter setting. Second, performing benchmarking experiments in a test environment is extremely time consuming. Due to the lack of insight into the impact of the individual components of the Web server on the performance, experiments must be done for many different workload scenarios.

In the literature, a significant number of papers have appeared focusing on workload characterization of the traffic on the Internet and in intranet environments, based on traffic measurements. Arlitt and Williamson [1] present a workload characterization study for Internet Web servers, based on a variety of data sets. Their main conclusions (in the context of the present paper) are that the mean transfer size is small but that the transfer-size distribution is heavy tailed, that the successive reference epochs to the same file are well-modeled by a Poisson process, and that a small number of pages account for the vast majority of the page requests. Paxson and Floyd [11] analyze traffic traces for Wide Area Networks (WANs) and show that user-initiated “session” arrivals are well-modeled by Poisson processes, but that packet-level traffic streams may deviate considerably from Poisson processes, and may exhibit self-similarity over different time scales. We refer to [6,10, and references therein], for discussions on the phenomenon of self-similarity of the traffic streams in LAN and WAN environments.

Only a few papers in the literature are focused on the modeling of Web server performance. Slothouber [13] proposes to model a Web server as an open queueing network. However, the model ignores essential lower-level details of HTTP and TCP/IP protocols, even though they strongly impact the Web server performance. In an excellent piece of work, Heidemann et al. [9] present analytic models for the interaction of HTTP with several transport layers (such as TCP, T-TCP and UDP), including the impact of slow-start algorithms. Dilley et al. [7] present a high-level layered model of an HTTP server, and build a tool framework to collect and analyze empirical data. Although the papers mentioned here provide significant insights into the performance of Web servers, none of the papers provide a model for the end-to-end performance of the communication between the client and the server.

In this paper, we propose an end-to-end performance model for Web servers, encompassing the impacts of client workload characteristics, server harware/software configuration, communication protocols, and interconnection topologies. HTTP transactions proceed along a number of phases in successive order. Therefore, the transaction flows within a Web server can be described by a tandem queueing model, consisting of the following submodels:

(1) a multi-server zero-buffer blocking model for the TCP connection setup phase,

(2) a multi-server finite-buffer blocking model for the HTTP application processing, and

(3) a finite-buffer polling model for the network I/O controller.

The interactions between the different submodels are discussed in detail. In the present paper, we focus on HTTP version 1.0 [5]. However, we emphasize that the model can be extended to incorporate improved versions of HTTP in a straightforward manner. The model has been implemented in a simulation tool that can be used to obtain insights into how the different components of the model interact, and how they impact the end-to-end performance. The performance predictions have been validated by experiments performed in a test lab. The results demonstrate that the predictions based on the simulation tool are very close to the test results. The simulation tool forms an excellent basis for the development of a Decision Support System for the proper tuning and sizing of Web servers.

The remainder of this paper is organized as follows. In section 2 we describe and model the transaction flows within an HTTP server. In section 3 the performance predictions based on the model are validated against performance results in a test lab environment. We have performed a number of simulation experiments to obtain a better understanding of how the performance-limiting component within a Web server varies for different parameter settings. The results of these experiments are outlined in section 4. Finally, section 5 contains several concluding remarks and addresses a number of topics for further research.

2. Transaction flows and modeling

An HTTP transaction proceeds through a Web server along three successive phases: (1) TCP connection setup, (2) HTTP layer processing and (3) network I/O processing. In this section, we describe and model the dynamics of each of these phases. Combining these per-phase models, we obtain a tandem queueing model, as illustrated in figure 1. We emphasize that in practice the interplay between the different Web server components is extremely complicated and may be highly implementation specific. The model described herein aims to give a generic simplified description of the transaction flows within the Web server, covering the main performance limiting components, but omitting many (possibly relevant) details. In order to keep the model tractable and to limit the size of the parameter space, several assumptions must be made. We emphasize that the model discussed below should be viewed from that perspective.

2.1. TCP connection setup phase

Before data can be transmitted between the client and the server, a two-way connection (a TCP socket) must be established. The TCP subsystem consists of a so-called TCP

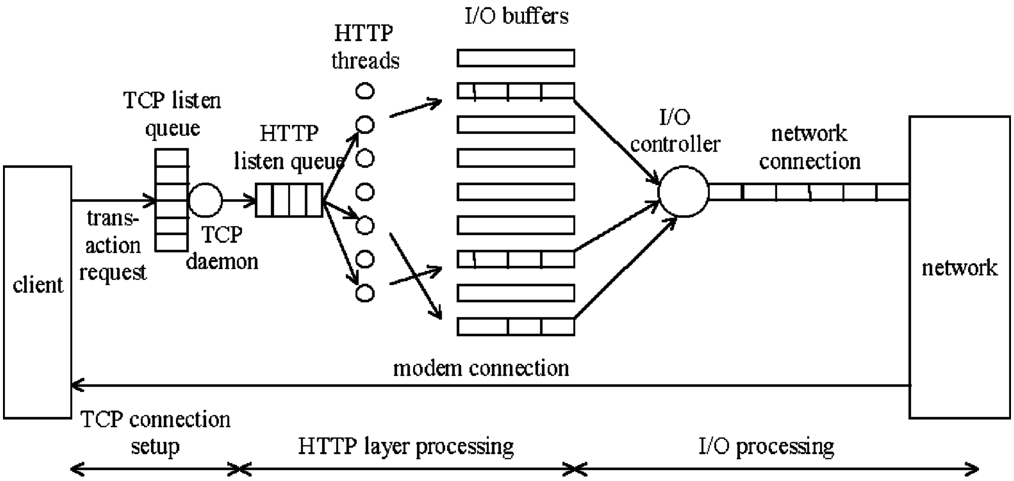

Figure 1. Queueing model for a Web server.

Listen Queue served by a server daemon. A TCP connection is established by a three-way handshake procedure, which proceeds along the following steps (see, for instance, [14] for more details):

- The client sends a connection request (SYN) to the server.
- If there is a slot available at the TCP Listen Queue, then the request occupies one slot and the server daemon sends an acknowledgment (SYN-ACK) to the client; otherwise, the connection request is rejected.
- Upon receipt of a SYN-ACK, the client sends an acknowledgement (ACK) and a transaction request (e.g., GET) to the server. Upon arrival of the transaction request, the TCP Listen Queue slot is released.

Immediately after the TCP socket has been established, the server daemon forwards the transaction request to the HTTP subsystem (discussed in section 2.2). After the transaction has been processed, the server typically sends a FIN message to the client to terminate the TCP socket.

To model the TCP connection setup phase, denote by N_{TCP} the size of the TCP Listen Queue (i.e., the number of slots). The TCP connection setup phase can be modeled as a blocking model with N_{TCP} servers and zero waiting buffer space, where a “server” represents a slot in the TCP Listen Queue and customers represent connection requests. If an incoming customer finds all servers busy (i.e., all slots are occupied by other pending connection requests), then the request is refused; otherwise, the request is taken into service immediately. A service time represents the time between (1) the arrival of the connection request at the TCP Listen Queue and (2) the time at which the transaction request (after receiving the SYN-ACK) arrives, as illustrated in figure 2. In this way, the duration of a service time corresponds to one network round-trip time (RTT) between the server and the client. The size of the TCP Listen Queue (N_{TCP}) is configurable.

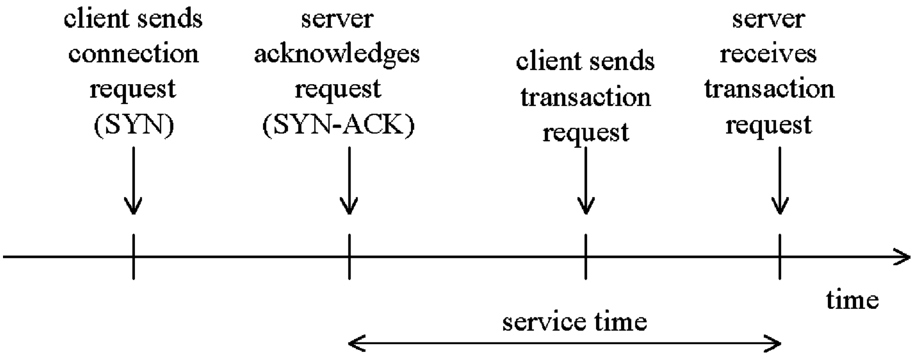
client sends connection request (SYN)
server acknowledges request (SYN-ACK)
client sends transaction request
server receives transaction request
time
service time

Figure 2. The 3-way handshake procedure during the TCP connection setup phase.

2.2. HTTP processing phase

After a TCP connection has been established, the transaction request is ready to be parsed and interpreted by the HTTP subsystem, which consists of an HTTP Listen Queue served by one or more multi-threaded HTTP daemons. The dynamics of the HTTP subsystem are described as follows:

- If an HTTP thread is available, then the thread fetches the requested file (either from a file system or from cache memory) and puts the file into a network I/O buffer (if available). The thread is then released to process the next transaction request.
- If all I/O buffers are occupied at that time, then the HTTP thread remains idling until an I/O buffer becomes available. If there is more than one thread waiting for an I/O buffer to become available, some (implementation-specific) assignment rule is used to determine in which order free-coming I/O buffers are assigned to the waiting threads.
- If there is no HTTP thread available (i.e., all threads are busy), then the transaction request enters the so-called HTTP Listen Queue (if possible), and waits until it gets assigned a thread to handle the request.
- If the HTTP Listen Queue is full, the transaction request is rejected, the connection is torn down, and the client receives a connection refused message.

Note that connection refusal messages may be generated in two ways: (1) blocking at the TCP subsystem (when all slots at the TCP Listen Queue are occupied, see section 2.1), and (2) blocking at the HTTP subsystem (when the HTTP Listen Queue is full).

The size of the requested file is generally unknown beforehand. Therefore, the file size may exceed the I/O buffer size. In that case, the file is partitioned into a number of parts $P_1, \dots, P_k$ each of which fills one I/O buffer (except for the trailing part P_k). In the case $k > 1$, all segments of a given file must make use of the same I/O buffer. Thus, an HTTP server is not allowed to dump different parts of the same file into different I/O buffers. Therefore, if $k > 1$ then P_1 (i.e., the first segment of the file) is placed into the I/O buffer. Then, the HTTP thread responsible for handling the transaction has to remain

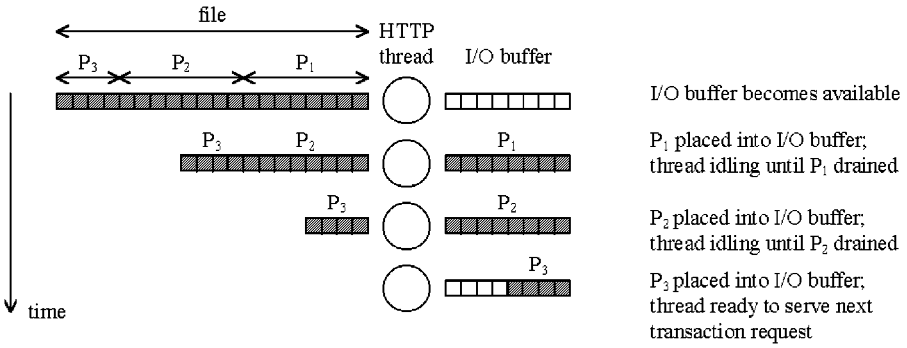
file
HTTP thread
I/O buffer
P3
P2
P1
time
I/O buffer becomes available
P1 placed into I/O buffer; thread idling until P1 drained
P2 placed into I/O buffer; thread idling until P2 drained
P3 placed into I/O buffer; thread ready to serve next transaction request

Figure 3. File partitioning during the HTTP processing phase.

idling (blocked) until the buffer has been drained completely before it can place P_2 into the buffer, and so on. An exception is made for P_k (the final part of the partitioned file): as soon as P_k has been placed into the I/O buffer, the thread is ready to serve another transaction request (and does not have to idle until P_k has been drained completely). Figure 3 illustrates the file partitioning for $k = 3$.

The HTTP subsystem can be modeled by a multi-server finite-buffer blocking system with N_{HTTP} servers and buffer size B_{HTTP}. The servers represent the HTTP threads, the customers represent transaction requests, and the buffer represents the HTTP Listen Queue. If a server is available, then the customer is taken into service immediately. Otherwise, the customer enters the HTTP Listen Queue; if the queue is full, then the customer is rejected. Define the service time τ_{trans} of a customer (transaction request) as the time interval between (1) the time at which a thread starts to fetch the requested file and (2) the time at which all parts of the file have been placed into an I/O buffer. The service time, τ_{trans}, consists of the following three parts:

$$\tau_{\text{trans}} = \tau_{\text{fetch}} + \tau_{\text{wait}} + \tau_{\text{drain}}, \tag{1}$$

where τ_{fetch} represents the time required to fetch the file, τ_{wait} represents the time the server has to wait to get access to an I/O buffer and τ_{drain} represents the time to put the entire file into an I/O buffer (possibly in parts). In general, the requested file is partitioned into parts $P_1, \ldots, P_k$, where $k := N_{\text{file}}$, the number of I/O buffers that the file needs (in figure 3 we have $N_{\text{file}} = 3$). Therefore, τ_{drain} can be further decomposed as

$$\tau_{\text{drain}} = \tau_{\text{drain}}^{(1)} + \cdots + \tau_{\text{drain}}^{(k-1)}, \tag{2}$$

where $\tau_{\text{drain}}^{(i)}$ is the time needed to put P_i into an I/O buffer and drain P_i $(i = 1, \ldots, k-1)$. Recall that the time needed to drain the trailing part of the file does not contribute to the "service time", so that $\tau_{\text{drain}}^{(k)}$ is excluded here. Denoting the file size by F and the I/O buffer by B_{IO}, N_{file} is given by

$$N_{\text{file}} = \left\lceil \frac{F}{B_{\text{IO}}} \right\rceil, \tag{3}$$

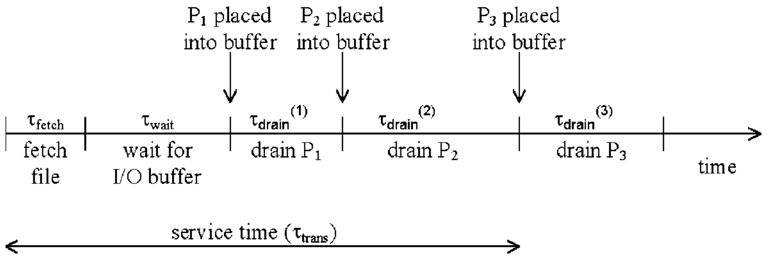

Figure 4. The HTTP processing phase.

where the $\lceil x \rceil$ is defined as the smallest integer that is larger than or equal to x. It is assumed that the time to place a part of the file into an I/O buffer (after an I/O buffer has been assigned to the responsible HTTP thread) is negligible. Figure 4 illustrates the components of a "service time" in the HTTP subsystem for the case $N_{\text{file}} = 3$ (see also figure 3). In general, τ_{fetch} is an independent random variable, while the probability distributions of the random variables τ_{wait}, N_{file} and τ_{drain}, and their correlation structure, are output parameters that generally depend on the performance of the I/O subsystem (modeled in the next section).

The size of the I/O buffers (B_{IO}) is configurable, and is typically tuned to entirely store the vast majority of the requested files, in the sense that $\text{Prob}\{F \leqslant B_{\text{IO}}\} \leqslant \alpha$, where α is close to 1 (e.g., 0.95 or 0.99). Hence, in most cases, the requested file fits within a single I/O buffer and, therefore, need not be partitioned. The buffer size (B_{IO}), the number of I/O buffers (N_{IO}), the size of the HTTP Listen Queue (B_{HTTP}), and the number of threads (N_{HTTP}) are configurable.

2.3. I/O processing phase

The different I/O buffers are "drained" over a common network connection to the network (e.g., the Internet or an intranet). The scheduling of access for the different output buffers to the network connection is done by a so-called I/O controller that "visits" the buffers in some order (e.g., in a round-robin fashion). The communication between the server and the client is based on the TCP/IP protocol suite. TCP/IP is a connection-oriented protocol, and is controlled by a windowing mechanism (see [14] for more details). The transmission unit for the TCP/IP-based network connection is the Maximal Segment Size (MSS), i.e., the largest amount of data that TCP will send to the client in one segment. Therefore, the files residing in the I/O buffers are (virtually) partitioned into blocks of 1 MSS (except for the trailing part of the file). The windowing mechanism implies that a block of a file residing in an output buffer can only be transmitted if the TCP window is open; that is, if blocks can still be transmitted before receiving an acknowledgment. Notice that the arrival of acknowledgments generally depends on the congestion in the network. Therefore, the rate at which each of the I/O buffers can "drain" their contents is affected by congestion in the network.

The dynamics of the I/O subsystem can be modeled as a single-server polling model with N_{IO} queues, each of size B_{IO}, where N_{IO} is the number of I/O buffers and

B_{IO} is the I/O buffer size. The server represents the I/O controller and the queues represent the I/O buffers. The “service times” represent the time to “drain” (i.e., transmit and acknowledge) 1 file block.

The total time to drain the contents of an I/O buffer $\tau_{\text{drain I/O buffer}}$ can be expressed as follows:

$$\tau_{\text{drain I/O buffer}} = \sum_{k} \tau^{(k)}_{\text{I/O block}}, \tag{4}$$

where $\tau^{(k)}_{\text{I/O block}}$ is the time needed to drain the kth block in the I/O buffer. $\tau^{(k)}_{\text{I/O block}}$ can be further decomposed as

$$\tau^{(k)}_{\text{I/O block}} = \tau^{(k)}_{\mathrm{RC}} + \tau^{(k)}_{\mathrm{DB}}, \tag{5}$$

where $\tau^{(k)}_{\mathrm{RC}}$ denotes the time until the server visits the I/O buffer in question (i.e., the residual cycle time), and $\tau^{(k)}_{\mathrm{DB}}$ denotes the time to “drain” the file block. The latter, in turn, can be decomposed further as follows:

$$\tau^{(k)}_{\mathrm{DB}} = \tau^{(k)}_{\mathrm{link}} + \tau^{(k)}_{\mathrm{inet}} + \tau^{(k)}_{\mathrm{client}}, \tag{6}$$

where $\tau^{(k)}_{\mathrm{link}}$ is the time to put the block on the output link, $\tau^{(k)}_{\mathrm{inet}}$ is the time to send the block and return an acknowledgment, and $\tau^{(k)}_{\mathrm{client}}$ is the time required by the client interface (e.g., a modem or LAN card) to read the block. Figure 5 illustrates the dynamics of the model for the case where the I/O buffer contains 3 blocks, numbered B_1, B_2 and B_3, respectively. The time to put a file block of size $B^{(k)}$ onto a network connection is given by the following expression:

$$\tau^{(k)}_{\mathrm{link}} = \frac{B^{(k)}}{\mu_{\mathrm{NC}}}, \tag{7}$$

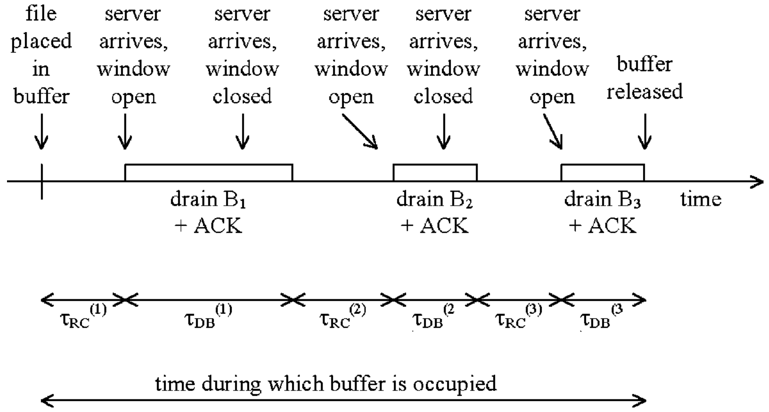
file placed in buffer
server arrives, window open
server arrives, window closed
server arrives, window open
server arrives, window closed
server arrives, window open
buffer released
drain B₁ + ACK
drain B₂ + ACK
drain B₃ + ACK
time
τRC(1)
τDB(1)
τRC(2)
τDB(2
τRC(3)
τDB(3
time during which buffer is occupied

Figure 5. I/O processing phase.

where μ_{NC} is the line speed of the connection to the network. Moreover, the time required by the client to read a file block of size $B^{(k)}$ is given by

$$\tau^{(k)}_{\text{client}} = \frac{B^{(k)}}{\mu_{\text{client}}}, \tag{8}$$

where μ_{client} is the rate the client reads incoming data (e.g., 28.8 Kbit/s for modem). $\tau^{(k)}_{\text{inet}}$ is a random variable with the same distribution as a network RTT.

Remark 2.1. The model explains how congestion in the network may lead to rejection of incoming transaction requests. To this end, suppose the network is congested for some time period. Then the network RTT increases, and consequently, TCP acknowledgments of the receipt of file blocks are delayed, so that the throughput of file blocks from the I/O buffer to the client over the TCP connection decreases. This means that the files are "drained" at a lower rate. This, in turn, implies that I/O buffers become available to the HTTP threads at a lower rate, so that HTTP threads may have to wait (idle) for a longer time period to get access to an I/O buffer to "dump" the file. Consequently, the HTTP Listen Queue will tend to fill up, leading eventually to rejection of incoming transaction requests. In this way, congestion in the network for some sustained period of time may cause the server itself to run into performance problems.

Remark 2.2. With the advent of transactions involving dynamic content, Web servers must handle requests for non-HTML (e.g., script output) files, which are generally much more CPU-intensive than static HTML files. Files with dynamic content are typically implemented into common scripting standards such as Common Gateway Interface (CGI) and Application Programmng Interface (API). In many Web server implementations, the HTTP thread responsible for running a script must wait for the execution of the script to be completed before handling another transaction request. In the model presented above, the impact of dynamic content on the Web server performance can be incorporated by modifying the "service-time distribution" of the HTTP threads accordingly. Note that in other server implementations (e.g., Microsoft Internet Information Server 3.0 and later), the server is equipped with a dedicated script engine. The HTTP thread checks if a script must be run, and if so, the HTTP thread forwards the request to the script engine. As soon as the script object has been delivered to the script engine, the HTTP thread is ready to handle newly incoming transaction requests (so the HTTP thread does not wait for the script execution to complete). Extension of the model to incorporate such a dedicated script engine is fairly straightforward.

3. Validation

We have implemented the model in a simulation tool to predict the Web server performance. To assess the validity of the model, we have performed experiments to compare performance predictions based on the simulation model with measurements in a test lab environment. The results are outlined below.

Consider a Web server with the following parameter settings: 4 CPUs, $N_{\mathrm{TCP}} = 1024$, $B_{\mathrm{HTTP}} = 500$, $N_{\mathrm{HTTP}} = 40$, $B_{\mathrm{IO}} = 1024$, $N_{\mathrm{IO}} = 60$, $\tau_{\mathrm{fetch}} = 9.4$ ms (consisting of 1.2 ms for HTTP processing and 8.2 ms for running dynamic content, see also remark 2.2), TCP maximum window size $=$ 8 Kbytes, MSS $=$ 1460 bytes, RTT $=$ 1 ms

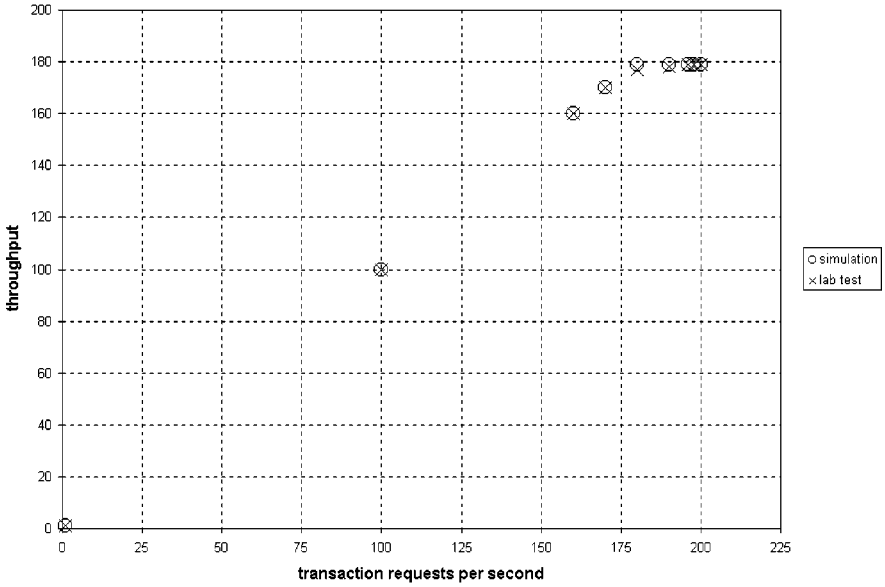

Figure 6. Throughput as a function of the transaction request rate.

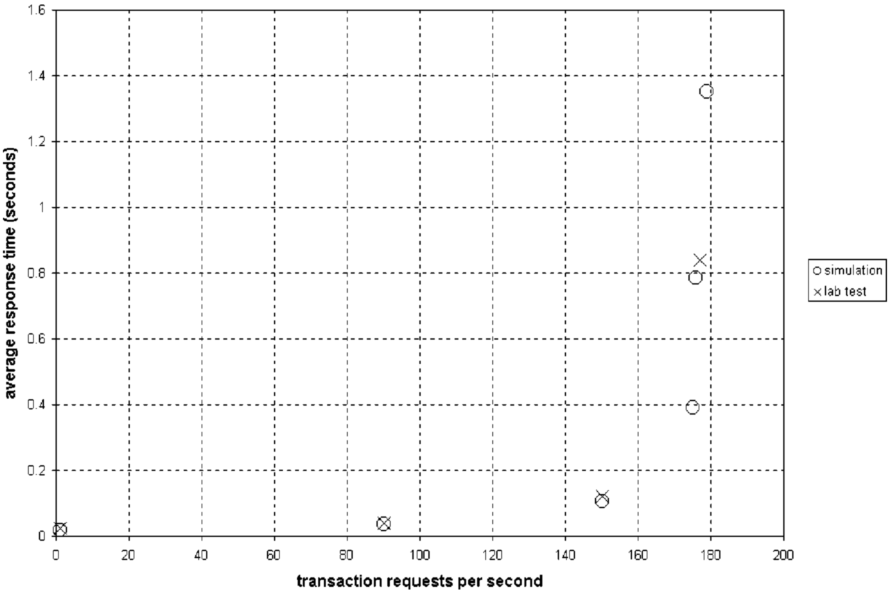

Figure 7. Average response time as a function of the transaction request rate.

(based on measurements). The clients and the server were connected via a 100 Mbit/s Fast Ethernet, so that $\mu_{\mathrm{NC}} = \mu_{\mathrm{client}} = 1.5$ Mbit/s. A load generator was used to "bombard" the server with transaction requests at given transaction rates. Figure 6 shows the server throughput as a function of transaction request rate (in transactions per second), and figure 7 shows the average end-to-end response time (in seconds) as a function of transaction request rate. The results in figures 6 and 7 show that the performance predictions based on the simulation model closely match the results in the test lab. We have performed various other validation experiments, and the accuracy of those results was found to be comparable to that shown in figures 6 and 7.

4. Simulation

We have performed a variety of numerical experiments to obtain insight into the performance capabilities and limitations of Web servers. The results are outlined below. The simulation tool is very useful to answer "what–if" questions to understand the performance under various parameter settings, and can ultimately be used to develop tuning guidelines for the configuration of Web servers. However, derivation of tuning guidelines requires a deeper insight into how the performance-limiting component of the server varies for different parameter configurations, which requires many more simulation experiments to be performed. We emphasize that the simulation results discussed below are only a first step in that direction, and are only valid in the specific settings considered.

Consider the model with the following parameters (referred to throughout as model I): 4 CPUs, $N_{\mathrm{TCP}} = 1024$, $B_{\mathrm{HTTP}} = 128$, $N_{\mathrm{HTTP}} = 128$, $N_{\mathrm{IO}} = 256$, $B_{\mathrm{IO}} = 30$ Kbytes, MSS $= 512$ bytes, τ_{fetch} is exponentially distributed with mean 5 ms, the file size is geometrically distributed with mean 4 Kbytes, the network (Internet) RTTs are exponentially distributed with mean 250 ms, $\mu_{\mathrm{NC}} = 1.5$ Mbit/s, $\mu_{\mathrm{client}} = 28.8$ Kbit/s, maximum TCP window size $= 1$ Kbyte. File-retrieval requests arrive according to a batch Poisson process, where the batch-size distribution (representing the number of in-line images per Web page) is geometrically distributed with mean 10. See remark 4.2 for a discussion on the assumptions. Figure 8 shows the throughput (i.e., the number of successful transactions per second) as a function of transaction request rate, and figure 9 shows the end-to-end response time (in seconds) as a function of transaction request rate (model I). Figure 8 shows that the transaction throughput increases linearly with request rate up to a certain threshold value. Close examination of the results in figure 8 indicates that the maximum throughput is limited by the bandwidth of the T1 connection, restricting the number of successful transactions to about 46 per second. Figure 9 shows that the response time is fairly constant when the server is lightly loaded, but increases rather sharply when the load approaches the capacity of the first bottleneck (i.e., the T1 line speed). When the offered load exceeds the capacity of the server, the server saturates, transactions are rejected, and the average response time tends to a constant threshold. Note that the response time does not increase without

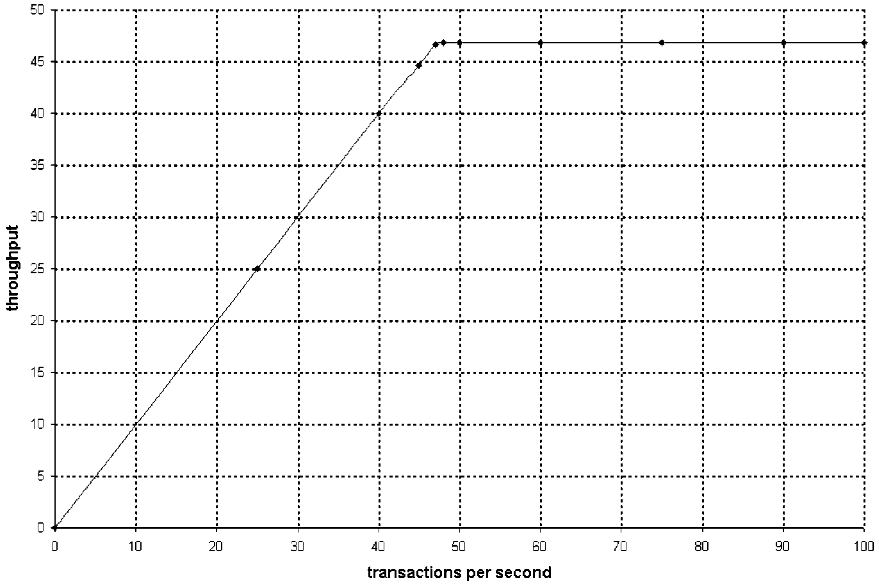

Figure 8. Throughput as a function of the transaction request rate (for model I).

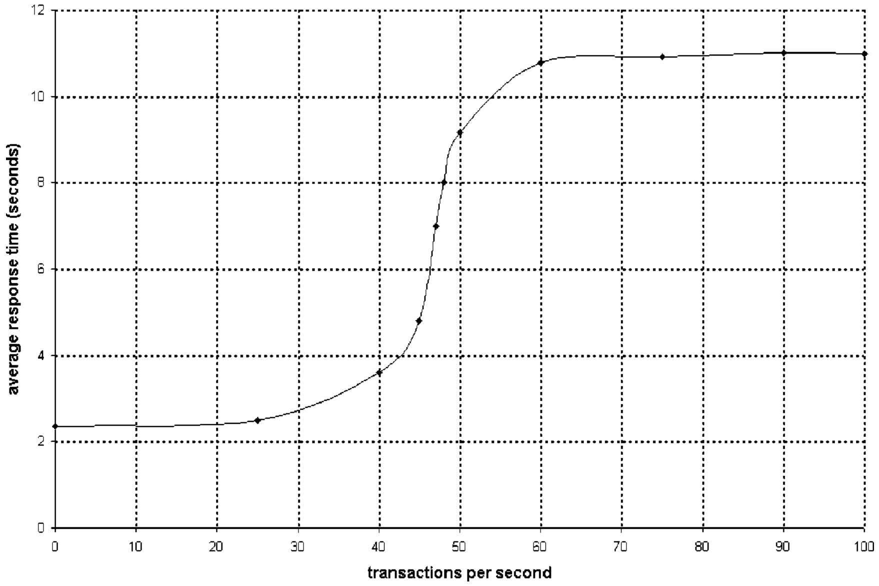

Figure 9. Average response time as a function of the transaction request rate (for model I).

bound, because the HTTP Listen Queue is finite and transaction requests finding a full buffer are rejected.

Next, consider what happens if the observed bottleneck in model I is removed by replacing the T1 network connection by a T3 line (which can process as much as

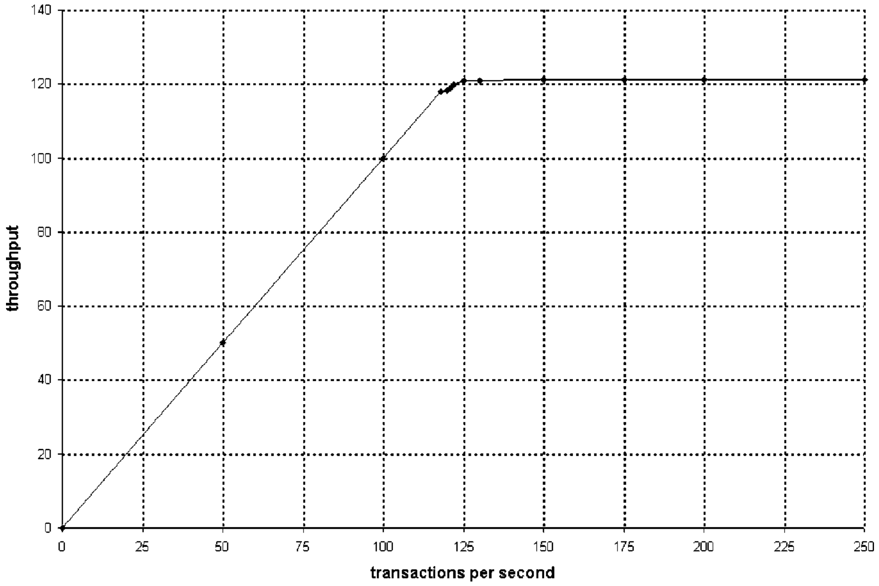

Figure 10. Throughput as a function of the transaction request rate (for model II).

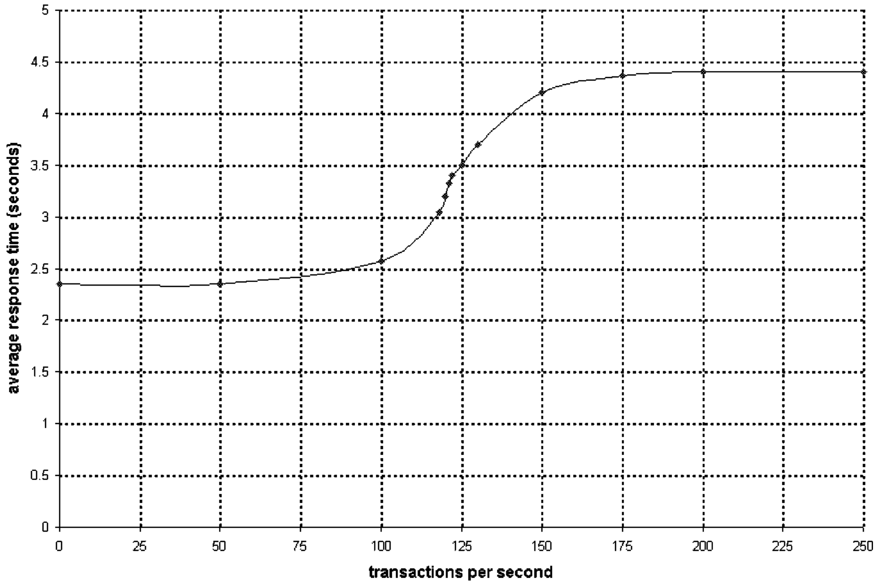

Figure 11. Average response time as a function of the transaction request rate (for model II).

45 Mbit/s). This model is referred to as model II. Figures 10 and 11 show the throughput and average response time as a function of transaction request rate. The results in figure 10 show that by replacing the T1 with a T3 network connection, the maximum throughput has increased considerably, but the server can not saturate the T3 line. In

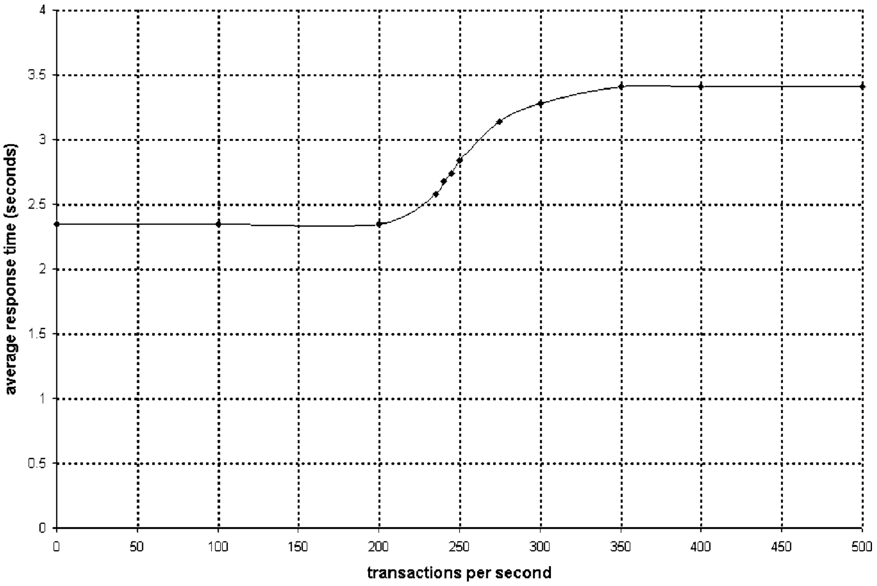

Figure 12. Average response time as a function of the transaction request rate (for model III).

fact, examination of the results shows that the number of I/O buffers (assumed to be $N_{\mathrm{IO}} = 256$) has now become the performance-limiting factor, restricting the throughput to about 121 transactions per second. Note that at maximum server throughput, the T3 network connection is only 11% loaded. Similar to the results for model I, we observe that the average delay increases fairly sharply when the server approaches its performance bottleneck. We also observe in figure 11 that the threshold for average delay has dropped from about 11 s to only about 4.5 s.

To proceed, suppose that in an attempt to remove the observed bottleneck in model II, it is decided to double the number of I/O buffers (so that $N_{\mathrm{IO}} = 512$ instead of 256). This model is referred to as model III. Figure 12 shows the average response time as a function of the transaction request rate for this model. The results in figure 12 show that by doubling the number of I/O buffers, the response times tend to increase when the transaction rate approaches about 240 (about two times the maximum throughput observed in model II). In fact, the results show that in model III the number of I/O buffers (although doubled compared to model II) is *still* the performance-limiting factor. We also observe that the average response time has dropped slightly compared to model II.

The numerical examples considered above assumed that the average RTT is 250 ms, which is rather typical in Wide Area Networks (WANs) such as the Internet. For comparison, let us consider the performance of Web servers in a Local Area Network (LAN) environment. To this end, we consider the model with the same parameters as in model III, but where the average RTT has decreased to only 1 ms and where the MSS is 1460 bytes (realistic in LAN environments, see [9]). The model is referred to as model IV. Figure 13 shows the average response time as a function of the transaction request rate for model IV. The results show that the maximum throughput has increased

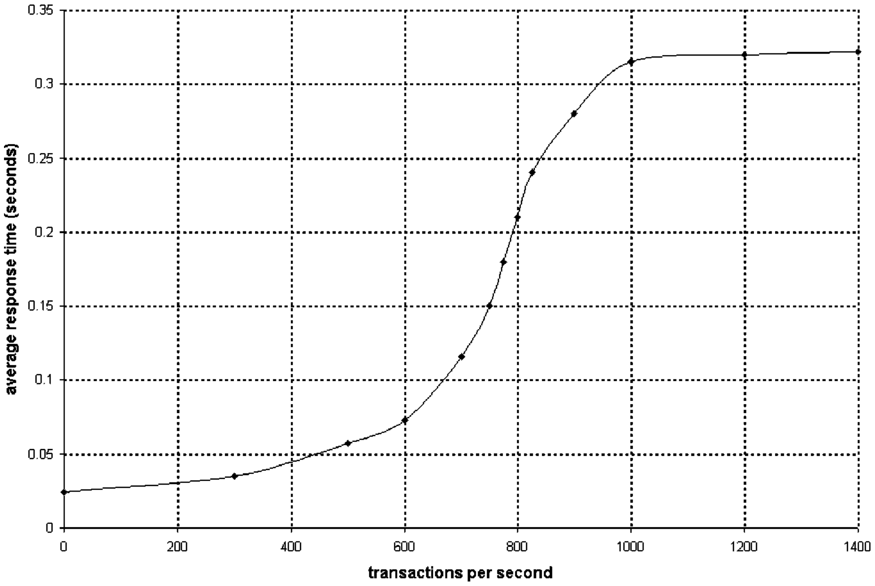

Figure 13. Average response time as a function of the transaction request rate (for model IV).

dramatically to about 800 transactions per second and that the average response time has decreased significantly (compared to the models II and III). The performance-limiting factor has now become the CPU processing speed, rather than the number of I/O buffers.

Remark 4.1. The results in figures 12 and 13 address an interesting observation regarding network environments and server benchmarking. In a WAN environment, the network RTT is significant, so that the TCP window flow control mechanism results in a relatively slow draining of the I/O buffers, making the I/O subsystem a serious potential performance bottleneck. In a LAN environment, in contrast, the network RTT is relatively short and the I/O subsystem drains the output buffers quickly, making the I/O subsystem a less likely bottleneck. Web server benchmarking experiments are typically performed in a high-speed LAN environment, where the TCP and I/O processing phases are unlikely to limit performance. Therefore, applying the absolute outcomes of the benchmarking experiments as an estimation of the actual Web server performance capabilities may lead to overly optimistic performance predictions when the server operates in a WAN environment. Ignoring the network environment may lead to highly inaccurate and overly optimistic performance predictions based on benchmarking.

Remark 4.2. To limit the number of simulations, we made a number of assumptions that may or may not be accurate in specific cases. For instance, we assume incoming transaction requests form a batch Poisson process. The literature, however, is not conclusive regarding the nature of the arrival process of transaction requests offered to a Web server. Interestingly, the results in [11] show that the user-initiated arrivals of “sessions” are well-modeled by a homogeneous Poisson arrival process, while lower-level (e.g., packet-level) traffic streams are not captured well by a Poisson process. In fact, recent studies based on traffic measurements have revealed that packet-level traffic streams in telecommunication networks may have a complicated autocorrelation structure, exhibiting long-range dependence (cf., e.g., [6,10,11]). Moreover, we assume that the network RTTs are exponentially distributed. Again, the literature is not conclusive regarding the stochastic behavior of RTTs. Furthermore, we assume that the number of file-retrieval requests generated by a single page-retrieval request is geometrically distributed with mean 10. This assumption may be unrealistic in some situations. In this context, note that number of simultaneously outstanding TCP connections per client may be limited; the maximum number of parallel TCP connections is a tunable browser parameter. We acknowledge that these assumptions may have a significant impact on the results presented above. It would be very useful and interesting to study the impact of the assumptions on the observed results.

In the context of the simulation model proposed here, we emphasize that *these assumptions are not restrictions posed by the model*: the model *itself* does not impose *any* restriction on the request arrival process, nor on the number of files per page, nor on the distribution of RTTs and their correlation structures.

Remark 4.3. The dynamics of the TCP flow control mechanism are notoriously complex, and the performance of TCP depends on a significant number of TCP parameter settings, such as the slow-start algorithm, the maximum send and receive window sizes, etc. We refer to [14] for an excellent detailed description of the TCP transaction flows. The primary focus of this paper is to present an end-to-end model of the transaction flows related to the application protocol HTTP, rather than to provide a detailed model for the TCP transaction flows. In order to cover the impact of the TCP flow control mechanism to some extent, while keeping the parameter space limited, we have implemented a simplified version of the TCP protocol stack in the model used for the simulations. We implement the dynamics of the TCP flow control mechanism according to the model described in [9]. Note that many simulation tools include a standard TCP flow control module.

5. Concluding remarks and topics for further research

We have presented a new queueing model for the end-to-end performance of Web servers. The model describes the impacts and interactions of the TCP subsystem, HTTP subsystem, I/O subsystem, and network. To predict the performance of Web servers (in terms of end-to-end response time and effective throughput), the model has been implemented in a simulation tool. The performance predictions based on the simulation model have been validated extensively with results in a test lab environment. To illustrate the usefulness of the model, we have performed a number of preliminary simulation runs, and the results provide several interesting insights into the performance capabilities of

Web servers. Since performance testing in a lab environment is extremely time consuming, the simulation tool provides an excellent vehicle for answering "what–if" questions in support of decisions regarding Web server configuration tuning.

The model described in this paper presents many opportunities and challenges to perform further research in various directions. First, in order to identify performance tuning guidelines, we must obtain a better understanding of the impact of the different system parameters (e.g., the arrival process, the buffer sizes, the file-size distribution, the number of HTTP threads, the number and size of the I/O buffers, etc.) on the performance of Web servers. To this end, we must perform many more simulation runs to compare the performance of Web servers under many configuration settings.

In addition, the model can be extended in several directions. For example, in order to process transaction requests involving dynamic content (e.g., CGI/API scripts, Java servlets), many servers are equipped with a script engine (with a set of dedicated threads). The model may be extended to explicitly cover the impact of such a script engine implementation. Furthermore, the performance of Web servers may be significantly impacted by the threading and object scope models that are employed by the Web server. For instance, in some threading models it may occur that different threads need to wait (idle) for access to the same object, leading to additional contention and delay. It would be interesting to explicitly model the impact of the threading and object scoping models.

In the current model, the transaction request rate is independent of the number of transactions in progress. In many applications, however, the customer population is fixed (e.g., in a corporate Intranet environment with a limited user population). Those scenarios may be modeled by a closed queueing network.

Next, in the current model a TCP connection is established for every file transfer (i.e., HTTP version 1.0 [5]), causing significant overhead. To overcome this performance penalty, the concept of persistent connections has been proposed (HTTP 1.1 [8]). It would be interesting to implement persistent connections in the model, and quantify the impact on the performance of Web servers.

Although the simulation tool has been found to be very effective in predicting Web server performance, the computation time may be significant, especially when the buffer sizes and numbers of threads are large. Therefore, it may be desirable to complement the simulation model with fast-to-evaluate analytic approximations to obtain performance predictions within a negligible time interval (at the expense of some accuracy). In [12], we propose such an analytic approximation, and show that the predictions match well with the simulation results.

Acknowledgements

The authors wish to thank Pravin Johri for implementing the simulation model. They also want to thank the anonymous referees, whose comments and suggestions have led to a significant improvement in the presentation of this paper.

References

[1] M.F. Arlitt and C.L. Williamson, Internet Web servers: workload characterization and performance implications, IEEE Transactions on Networking 5 (1997) 631–645.

[2] AT&T Easy World Wide Web service, `http://www.att.com/easywww/`.

[3] AT&T Just4Me service, `http://www.att.com/just4me/`.

[4] AT&T Connect 'N Save VoIP service, `http://www.connectnsave.att.com/`.

[5] T. Berners-Lee, R. Fielding and H. Frystyk, Hypertext transfer protocol – HTTP/1.0, RFC 1945, Internet request for comments (1995).

[6] M. Crovella and A. Bestavros, Self-similarity in World-Wide Web traffic: evidence and possible causes, in: *Proc. of ACM Sigmetrics*, Philadelphia, PA, 1996, pp. 160–169.

[7] J. Dilley, R. Friedrich, T. Jin and J. Rolia, Web server performance measurements and modeling techniques, Performance Evaluation 33 (1998) 5–26.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk and T. Berners-Lee, Hypertext transfer protocol – HTTP/1.1, RFC 2068, Internet request for comments (1997).

[9] J. Heidemann, K. Obraczka and J. Touch, Modeling the performance of HTTP over several transport protocols, IEEE Transactions on Networking 5 (1997) 616–630.

[10] W. Leland, M. Taqqu, W. Willinger and D. Wilson, On the self-similar nature of Ethernet traffic, IEEE Transactions on Networking 2 (1994) 1–15.

[11] V. Paxson and S. Floyd, Wide area traffic: the failure of Poisson, IEEE Transactions on Networking 3 (1995) 226–244.

[12] P.K. Reeser, R.D. van der Mei and R. Hariharan, An analytic model of an HTTP Web server, in: *Teletraffic Engineering in a Competitive World*, eds. P. Key and D. Smith, *Proc. of 16th International Teletraffic Congress* (Elsevier, Amsterdam, 1999) pp. 1199–1208.

[13] L.P. Slothouber, A model of Web server performance, `http://louvx.biap.com/white-papers/performance/overview/`.

[14] W.R. Stevens, *TCP/IP Illustrated*, Vol. 1 (Addison-Wesley, Reading, MA, 1996).

[15] WebStone, Mindcraft Inc., `http://www.mindcraft.com/`.

[16] S. Weiberle, Private communications with support engineers from Sun Microsystems.