Analytic Model of Web Servers in Distributed Environments

Paul Reeser
AT&T Labs
D5-3B20, 200 Laurel Ave
Middletown, NJ 07748
+1 732 420 3693
preeser@att.com

Rema Hariharan
AT&T Labs
D5-3D17, 200 Laurel Ave
Middletown, NJ 07748
+1 732 420 3743
rhariharan@att.com

ABSTRACT

In this paper, we illustrate a model-based approach to Web server performance evaluation, and present an analytic queueing model of Web servers in distributed environments. Performance predictions from the analytic model match well with the performance observed from simulation. The model forms an excellent basis for a decision support tool to allow system architects to predict the behavior of new systems prior to deployment, or existing systems under new workload scenarios.

Keywords

HTTP, Web, distributed, OO, Java, servlet, script, performance.

1. INTRODUCTION

Web technologies are currently being employed to provide end-user interfaces in distributed computing environments. The core element of these Web solutions is a Web server based on the HyperText Transfer Protocol (HTTP) running over TCP/IP. Web servers are required to perform millions of transaction requests per day at an acceptable Quality of Service (QoS) level in terms of client response time and server throughput. Consequently, a thorough understanding of the performance capabilities and limitations of Web servers is critical.

In many applications, the Web server performs significant dynamic server-side scripting. In these applications, a Web server retrieves a file, parses the file for scripting language content, interprets the scripting statements, and executes embedded code, possibly requiring a connection to a remote application for data processing/transfer. To facilitate this functionality, many servers implement the Common Gateway Interface (CGI) standard. However, for each invocation of a CGI application, a new process is forked and executed, causing significant performance problems on the server. To overcome this performance penalty, many Web servers implement an Application Programming Interface (API) to perform server-side processing without spawning a new process, either by interpreting embedded scripting on Web pages, or by dynamically loading precompiled code.

WOSP 2000, Ontario, Canada

One approach to performing this server-side scripting is to implement a script-engine (SE) dedicated to processing server-side scripts. Examples of SE implementations are the Active Server Pages (ASP) technology in Microsoft's Internet Information Server (IIS), or the Java Server Pages (JSP) technology in both IIS and Netscape's Enterprise Server (NES). In ASP applications, for example, IIS retrieves a file, parses the file for scripting language content, and interprets the scripting statements. Since scripts are typically interpreted, a complex script will impede SE performance.

Web server performance in these distributed environments is a complex interplay between a variety of factors (e.g., hardware platform, server operating system, script execution environment, workload characteristics, network bandwidth, etc). Experience has shown that the performance of Web servers can be impacted tremendously by the proper tuning of the server components. In order to properly configure these different components, however, it is crucial to understand how these components interact and how they impact user-perceived end-to-end performance. Although testing is essential for assessing performance, there are inherent limitations to a testing approach for validating Web server performance. Consequently, modeling is critical to further understand the performance capabilities and limitations of Web servers that participate in distributed computing.

In this paper, we illustrate a model-based approach to Web server performance evaluation, and present an analytic queueing model of Web servers in distributed environments. Performance predictions from the analytic model match well with the performance observed from simulation. The model forms an excellent basis for a decision support tool to allow system architects to predict the behavior of new systems prior to deployment, or existing systems under new workload scenarios.

2. RELATION TO PREVIOUS WORK

A number of queueing models of Web server performance exist in the literature. Slothouber [14] proposes to model a Web server as an open queueing network. Menascé and Almeida [9] provide an analysis of the Web application and workload characteristics that impact performance, and develop queueing models of the server hardware elements. However, while these models are useful to identify performance tradeoffs and perform high-level capacity planning, they ignore essential low-level details of the HTTP and TCP/IP protocols and software, despite the fact that these details strongly impact server performance. Van der Mei, etal [17] propose a detailed simulation model of the low-level hardware and software components of a Web server, and Reeser, etal [13] propose a detailed analytic model. However, these models fail to address Web servers with significant server-side processing that participate in distributed computing.

In this paper, we present an enhanced end-to-end analytic queueing model of Web servers that incorporates dynamic server-side computing in a distributed environment. This model represents a substantial extension of the model proposed in [13] to include enhancements to the existing HTTP and I/O sub-system models, and introduction of new Script Engine and distributed backend sub-system models. The basic analytic model presented here has been validated by comparing performance predictions from the model to those from a simulation [17]. The simulation, in turn, has been validated by comparing to the performance observed in a test lab environment [16], as well as in a real-world production environment with actual users, authentic arrival patterns, and realistic Web content and scripting logic [8].

Analytically solving a queueing model with this level of detail requires a number of simplifying assumptions. The basic assumptions underlying this model are that inter-arrival times and holding times at each sub-system are exponentially distributed, and output file sizes are geometrically distributed. Obviously, these assumptions seem unrealistic for any real-world scenario. Therefore, we outline a number of extensions to the basic model to address these assumptions, including enhancements to model HTTP 1.1 (persistent connections), non-Poisson arrival processes, and heavy-tailed distributions. However, since these particular extensions have not yet been incorporated or validated, this work should be viewed accordingly as preliminary work in progress.

Despite these limitations, early evidence from the simulation model [8] suggests that while these real-world characteristics (non-Poisson arrivals and heavy-tailed service times) do impact performance at the packet level, they may not strongly manifest themselves in the average end-to-end user-perceived application performance measures (such as response time and blocking).

This analytic model forms an excellent basis for development of a decision support tool for evaluating the performance of Web servers in a distributed environment, allowing system architects to predict the behavior of new systems prior to their deployment, or the behavior of existing systems under new workload scenarios.

3. BASIC DESCRIPTIVE MODEL

3.1 Basic HTTP Transaction Flow

The flow of a basic HTTP 1.0 transaction through a Web server is shown in Figure 1. (Extension of the basic model to HTTP 1.1 is discussed in section 6.1.) Transactions proceed through a Web server along four successive phases (sub-systems): TCP/IP connection setup, HTTP application processing, Script Engine (SE) dynamic processing (including interaction with a distributed backend server), and network I/O processing and transmission. These phases are discussed in more detail below.

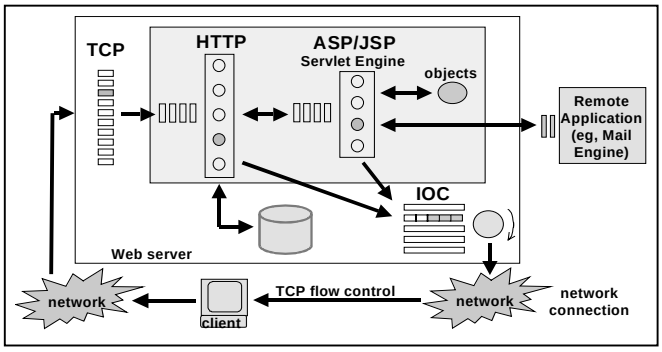

Figure 1: Model of Basic HTTP Transaction Flows

3.2 TCP Connection Setup Phase

Before information can be exchanged between the client and server, a two-way connection (TCP socket) must be established. The TCP sub-system consists of a TCP Listen Queue (TCP-LQ) served by a "listener" daemon (httpd in the case of HTTP traffic). A TCP connection is established by the well-known 3-way handshake procedure (SYN, SYN-ACK, ACK). Immediately after the TCP socket is established, the transaction request is forwarded to the HTTP sub-system for processing. If all slots in the TCP-LQ are occupied upon arrival of a request, then the request is rejected, and the client receives a “connection refused” message.

3.3 HTTP Layer Processing Phase

The HTTP sub-system consists of an HTTP Listen Queue (HTTP-LQ) served by a multi-threaded HTTP daemon that coordinates the processing performed by a number of (worker) threads. The dynamics of the HTTP sub-system are as follows:

1. If an HTTP worker thread is available, then the thread retrieves the requested file. If the file requires script processing, then the transaction is forwarded to the SE sub-system (see section 3.4). Otherwise, the static file content is retrieved and forwarded to the I/O sub-system (see section 3.5). If all I/O buffers are occupied at that time, then the HTTP thread remains idling until an I/O buffer becomes available.
2. If there is no HTTP worker thread available, then the transaction request enters the HTTP-LQ (if possible), and waits until a thread is assigned to handle the request.
3. If the HTTP-LQ if full, then the transaction request is rejected, the clients receives a “connection refused” message, and the TCP connection is torn down.

3.4 SE Layer Processing Phase

The script-engine (SE) dynamics generally depend heavily on the Web server implementation and server operating system. In this section, we attempt to highlight the general performance issues that arise in SE implementations, without focusing too heavily on a particular technology. The SE sub-system consists of an SE Listen Queue (SE-LQ) and a pool of handler threads dedicated to interpreting scripting statements and executing embedded code (e.g., C++, Java). During execution, communication with a remote backend server may be needed (e.g., to perform a database query), possibly requiring a TCP connection to the backend server to be established and subsequently torn down. The dynamics of the SE sub-system are described as follows:

1. If an SE handler thread is available, then the thread executes the embedded code. If the HTTP/SE implementation operates in *blocking* mode (that is, the HTTP thread blocks on a response from the SE – see Figure 2), then the SE thread forwards the transaction request back to the HTTP thread for further handling (see section 3.3). Otherwise, if the HTTP/SE implementation operates in *non-blocking* mode (that is, the HTTP thread is released as soon as the request is forwarded to the SE sub-system – see Figure 2), then the SE thread forwards the transaction request directly to the I/O sub-system (see section 3.5). If all I/O buffers are occupied at that time, then the SE thread remains idling until an I/O buffer becomes available.
2. If there is no SE handler thread available, then the transaction request enters the SE-LQ (if possible), and waits until a thread is assigned to handle the request.

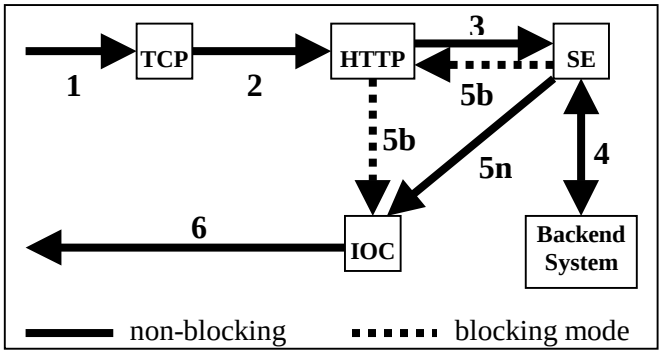

Figure 2: Modes of HTTP/SE Implementation

3. If the SE-LQ if full, then the transaction request is rejected, the clients receives a "connection refused" message, and the TCP connection is torn down.

Note that Microsoft's IIS/ASP implementation operates in non-blocking mode, while most other implementations (e.g., NES/JSP) typically operate in blocking mode.

3.5 Network I/O Processing Phase

The I/O sub-system consists of a number of parallel I/O buffers, an I/O controller (IOC), the network interface card (NIC), and the connection (link) from the Web server to the network. The contents of the I/O buffers are drained over the network link to the client as scheduled by the IOC. The IOC visits the different I/O buffers in a round-robin fashion, checks whether the I/O buffers have any data to send, and if so, places a chunk of data onto the network connection.

The communication between the server and client is based on the TCP flow control mechanism (see [15] for details). The transmission unit for TCP is the maximum segment size (MSS), the largest amount of data that TCP will send at one time. Files residing in I/O buffers are (virtually) partitioned into blocks of one MSS (except for the trailing part of the file). The window mechanism implies that a block can only be transmitted if the TCP window is open. The arrival of acknowledgments generally depends on network congestion. Therefore, the rate at which I/O buffers drain their contents is affected by network congestion.

Numerous approaches have been suggested for analytically modeling multi-layered communication protocol architectures such as TCP/IP (see [2] and references therein).

3.6 Sub-System Interactions

To understand the end-to-end performance of Web servers, it is important to understand the interactions between the sub-systems discussed in sections 3.2-3.5. To this end, consider what happens if the network connection between the Web server and the client is congested. Then the network round-trip time (RTT) increases, so that TCP acknowledgments (from client to server) of the receipt of file blocks by the client are delayed, implying that the "drain rate" of the I/O buffers decreases. This, in turn, implies that I/O buffers become available to the HTTP/SE threads at a slower rate, so that these threads may have to wait for a longer time period to get access to an I/O buffer. Since the threads are idling as long as they are waiting for an I/O buffer to become available, the availability of idle threads goes down. The HTTP and SE LQs then tend to fill up and overflow, leading to the blocking of incoming transaction requests. In this way, performance problems in the network will lead to performance problems throughout the Web server itself.

The key to providing a tractable analytic model of these intricate dependencies and interactions lies in the ability to de-couple them, while at the same time accurately capturing the impacts of "downstream" congestion. The approach taken here is to treat each sub-system as a separate Markovian queue, compute its performance in isolation, and feed the results back into adjacent sub-systems. The resulting model consists of a sequence of (independent yet coupled) queues, where the arrival rates and holding times at each node are iteratively adjusted to account for downstream performance. The resulting system is solved recursively until convergence.

4. BASIC ANALYTIC MODEL

4.1 Markovian Queue Refresher

Prior to analyzing the various sub-systems, it is convenient to review the solution to the general M/M/N/Q Markovian queueing system. Assume that arrivals form an i.i.d. Poisson process with rate λ, and that service times are i.i.d. exponentially distributed with mean τ. Let $A = \lambda\tau$ denote the offered load. Let N denote the number of independent service stations, and let Q denote the number of FCFS waiting stations (not including the service stations). We denote the resulting system as an $M(\lambda)/M(\tau)/N/Q$ Markovian queueing system [3]. The relevant queueing measures follow directly by applying basic queueing results. In particular, one can readily compute:

$\pi \equiv \pi(A,N,Q)$ = the probability of queueing (all service stations occupied),

$\beta \equiv \beta(A,N,Q)$ = the probability of blocking (all service + waiting stations occupied),

n = the average number of occupied service stations,

q = the average number of occupied waiting stations, and

ω = the average waiting time prior to service.

Note that if Q=0, then $\beta(A,N,0)$ represents the Erlang B ("blocked calls cleared") formula [3], and if $Q=\infty$, then $\pi(A,N,\infty)$ represents the Erlang C ("blocked calls delayed") formula [3].

4.2 TCP Sub-System Model

As described in section 3.2, the TCP sub-system behaves like a group of N_{tcp} separate servers, where each "server" represents a slot in the TCP-LQ. A service time represents the time between the arrival of the connection request at the TCP-LQ and the completion of the TCP handshake, corresponding to one round-trip time (RTT) between the server and the client. We assume that incoming requests form a Poisson process with rate λ_{file} (in files/second), each with an exponentially distributed holding time with mean τ_{net} (in seconds) corresponding to the network RTT. Accordingly, we model this TCP sub-system as an $M(\lambda_{file})$ / $M(\tau_{net})$ / N_{tcp} / 0 blocking system. (Extension of this basic model to non-Poisson arrival processes will be discussed in section 6.2.)

Incoming connection requests that arrive when all N_{tcp} LQ slots are occupied are blocked ("connection refused"), and retry with probability π_{retry}. We account for the reattempts by inflating the offered load as follows (see Figure 3): The initial offered load a_{file} (in Erlangs) is given by

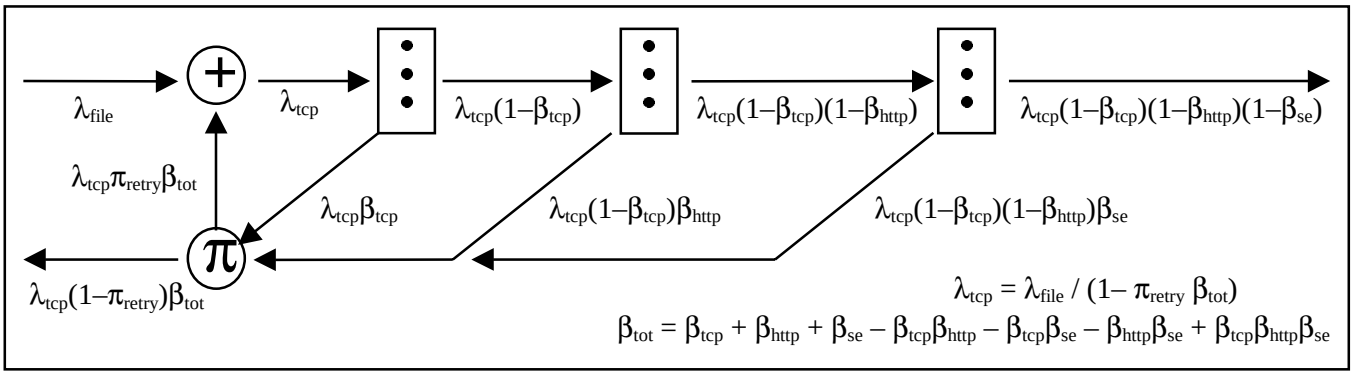

Figure 3: Modes of HTTP/SE Implementation

$$a_{file} = \lambda_{file}\tau_{net} . \quad (2.1)$$

The total offered load A_{tcp} (including reattempts) is then given by

$$Atcp = afile + Atcp\ \pi retry\ \beta tot , \quad (2.2)$$

where β_{tot} is the total (TCP + HTTP + SE) LQ blocking experienced by the incoming requests (analyzed in section 4.4). Solving for A_{tcp} yields

$$A_{tcp} = a_{file} / (1 - \pi_{retry}\ \beta_{tot}) = \lambda_{file}\tau_{net} / (1 - \pi_{retry}\ \beta_{tot}) . \quad (2.3)$$

The probability of finding all TCP LQ slots occupied (ie, the probability of blocking) β_{tcp} and the average number of occupied TCP LQ slots n_{tcp} follow directly from basic results for the M/M/N/0 blocking system. Then, the effective file request rate λ_{tcp} at the TCP sub-system is given by

$$\lambda_{tcp} = \lambda_{file} / (1 - \pi_{retry}\ \beta_{tot}) , \quad (2.4)$$

and the effective file request rate λ_{http} offered to the HTTP server sub-system is given by

$$\lambda_{http} = \lambda_{tcp} (1 - \beta_{tcp}) = \lambda_{file} (1 - \beta_{tcp}) / (1 - \pi_{retry}\ \beta_{tot}) . \quad (2.5)$$

Note that the impact of "downstream" congestion on the TCP sub-system is captured in the calculation of total offered load A_{tcp} (including reattempts), which in turn impacts the calculations of the TCP LQ blocking β_{tcp} and the total (TCP + HTTP + SE) LQ blocking β_{tot}.

4.3 HTTP Sub-System Model

The HTTP sub-system consists of a LQ of size Q_{http} served by an HTTP daemon with N_{http} worker threads. The customer occupies the thread from the time at which a thread is assigned to the request until the thread becomes available to handle another request (including any time the thread waits for a response from the SE sub-system, or for an I/O buffer to become available). We assume that incoming transaction requests form a Poisson process with rate λ_{http}, each with an exponentially distributed HTTP thread holding time with mean τ_{http}. We model this HTTP sub-system as an $M(\lambda_{http}) / M(\tau_{http}) / N_{http} / Q_{http}$ queueing/blocking system. (Extension to non-Poisson arrival processes and heavy-tailed service times will be discussed in sections 6.2 and 6.3.)

Incoming requests that arrive when all Q_{http} LQ slots are occupied are blocked, and retry with probability π_{retry}. As before, we account for these reattempts by inflating the offered load as follows (see Figure 3): The total offered load A_{http} (including reattempts) is given by

$$A_{http} = \lambda_{http}\ \tau_{http} = \lambda_{file}\ \tau_{http}\ (1–\beta_{tcp}) / (1– \pi_{retry}\ \beta_{tot}) . \quad (3.1)$$

The probability of finding all HTTP LQ slots occupied β_{http}, the average number of occupied HTTP threads n_{http}, the average number of occupied HTTP LQ slots q_{http}, and the average waiting time in the HTTP LQ ω_{http} all follow directly from basic results for the M/M/N/Q queueing system. Then, the effective file request rate λ_{se} offered to the SE sub-system is given by

$$\lambda_{se} = \lambda_{http}\ (1–\beta_{http}) = \lambda_{file}\ (1–\beta_{tcp})\ (1–\beta_{http}) / (1– \pi_{retry}\ \beta_{tot}). \quad (3.2)$$

Next, we compute the average HTTP thread holding time τ_{http}. In the case of a *non-blocking* SE implementation, τ_{http} consists simply of the time to process the requested file for scripting language content τ_{proc}, after which the request is forwarded to the SE sub-system for script execution. The time to process the file τ_{proc} is given by

$$\tau_{proc} = \tau_{cpu} \max \{ [\min(n_{http}+ I^*, N_{http}) + n_{se}] / N_{cpu} , 1\}, \quad (3.3)$$

where τ_{cpu} is the CPU execution time for the HTTP thread to process the file, n_{se} is the average number of occupied SE threads (analyzed in section 4.4), N_{cpu} is the number of CPUs, and $I^* = 1$ if $N_{http} > 1$ (and 0 otherwise).

Note that the computation of τ_{proc} takes into account the fact that the hardware resources (CPUs) are shared among software threads. The behavior captured in equation 3.3 is best described as "time-slicing", where each active request gets an equal proportion of the CPU, and the request service time is linearly elongated by multiplying by the number of active requests. In contrast, the behavior of the CPU model in [13] is best described as "processor-sharing", where request service time is non-linearly elongated by dividing by {1 – the CPU utilization}. As we will see in section 5, this change represents a substantial enhancement over the model in [13]. Comparison to simulation and test results has demonstrated that the time-slicing model is more accurate.

In the case of a *blocking* SE implementation, τ_{http} consists of the time to process the requested file τ_{proc}, plus the time the HTTP thread is blocked waiting for the SE sub-system to return control, plus the time the HTTP thread is blocked waiting for an output buffer to become available (in the case that all output buffers are occupied), plus the time the HTTP thread is blocked waiting to refill the output buffer (in the case that the requested file does not fit entirely into the buffer in one pass). Therefore, τ_{http} is given by

$$\tau_{http} = \tau_{proc} + \omega_{se} + \tau_{se} + \omega_{ioq} + \omega_{buf} , \quad (3.4)$$

where ω_{se} is the average waiting time in the SE LQ and τ_{se} is the average SE thread holding time (both analyzed in section 4.4), ω_{ioq} is the average waiting time to receive a network output buffer (analyzed in section 4.5), and ω_{buf} is the average waiting time to refill the output buffer.

The average waiting time to refill the network output buffer ω_{buf} is determined as follows: Assuming that the file size is geometrically distributed with parameter $p=1/n_{file}$, where n_{file} is the average file size (in blocks), then the probability π_{buf} that the requested file size exceeds the output buffer size N_{iob} (in blocks) is given by

$$\pi_{buf} = 1 - P\{\text{file size} \le N_{iob}\} = 1 - p\sum_{i=1}^{N}(1-p)^{i-1} = (1-p)^{N} \quad (3.5)$$

for $p = 1/n_{file}$ and $N = N_{iob}$. Similarly, it can be shown that the probability of exceeding two buffer sizes is π_{buf}^2, and so on. Therefore, the expected number of stages required to write the entire file into the network output buffer is given by $1 / (1 - \pi_{buf})$, and the expected number of additional stages (beyond the first mandatory write) is given by $1/(1-\pi_{buf}) - 1 = \pi_{buf} / (1 - \pi_{buf})$.

It can then be shown that the average waiting time ω_{buf} to refill the output buffer is given by

$$\omega_{buf} = N_{iob}\, \tau_{iob}\, \pi_{buf} / (1 - \pi_{buf}) , \quad (3.6)$$

where τ_{iob} is average block service time (analyzed in section 4.5).

Note that the impact of "downstream" congestion (in terms of SE and I/O system occupancy and I/O buffer size) is captured in the calculation of the average HTTP thread holding time τ_{http}.

4.4 SE Sub-System Model

The SE sub-system consists of a LQ of size Q_{se} served by N_{se} separate script handler threads. We assume that incoming requests form a Poisson process with rate λ_{se}, each with an exponentially distributed SE thread holding time with mean τ_{se}. We model this SE sub-system as an $M(\lambda_{se}) / M(\tau_{se}) / N_{se} / Q_{se}$ queueing/blocking system. (Extension to non-Poisson arrival processes and heavy-tailed service times will be discussed in sections 6.2 and 6.3.)

Incoming requests that arrive when all Q_{se} LQ slots are occupied are blocked, and retry with probability π_{retry}. As before, we account for these reattempts by inflating the offered load as follows (see Figure 3): The total offered load A_{se} (including reattempts) is given by

$$A_{se} = \lambda_{se}\, \tau_{se} = \lambda_{file}\, \tau_{se}\, (1-\beta_{tcp})\, (1-\beta_{http}) / (1- \pi_{retry}\, \beta_{tot}). \quad (4.1)$$

The probability of finding all SE LQ slots occupied β_{se}, the average number of occupied SE threads n_{se}, the average number of occupied SE LQ slots q_{se}, and the average waiting time in the SE LQ ω_{se} all follow directly from basic results for the M/M/N/Q queueing system. Then, the total (TCP + HTTP + SE) LQ blocking β_{tot} experienced by incoming requests is given by

$$\beta_{tot} = \beta_{tcp} + \beta_{http}\,(1- \beta_{tcp}) + \beta_{se}\,(1-\beta_{http})\,(1- \beta_{tcp}). \quad (4.2)$$

Next, we compute the average SE thread holding time τ_{se}. In the case of a *blocking* SE implementation, τ_{se} consists simply of the time to execute the requested script τ_{exec} (including any distributed backend delay), after which the request is returned to the HTTP thread for final processing. The time to execute the script τ_{exec} is given by

$$\tau_{exec} = \tau_{script}\, \max\{[\min(n_{se}+1, N_{se})+n_{http}]/N_{cpu} , 1\} + \tau_{backend}, \quad (4.3)$$

where τ_{script} is the CPU execution time for the SE thread to run the script, $\tau_{backend}$ is the delay associated with communication to the distributed backend server (eg, a legacy database), and $I^* = 1$ if $N_{se} > 1$ (0 otherwise). Again, note that the computation of τ_{exec} takes into account the fact that the hardware resources (CPUs) are shared among software threads in a time-slicing manner.

In the case of a *non-blocking* SE implementation, τ_{se} consists of the time to execute the requested script τ_{exec} (including any backend delay), plus the time the SE thread is blocked waiting for an output buffer to become available (in the case that all output buffers are occupied), plus the time the SE thread is blocked waiting to refill the output buffer (in the case that the requested file does not fit entirely in the output buffer in one pass). Therefore, τ_{se} is given by

$$\tau_{se} = \tau_{exec} + \omega_{ioq} + \omega_{buf} . \quad (4.4)$$

Then, the effective file request rate λ_{buf} offered to the I/O sub-system is given by

$$\lambda_{buf} = \lambda_{se}(1-\beta_{se}) = \lambda_{file}(1-\beta_{tcp})(1-\beta_{http})(1-\beta_{se})/(1-\pi_{retry}\beta_{tot}). \quad (4.5)$$

Note that the impact of "downstream" congestion (in terms of I/O sub-system buffer size and occupancy) is captured in the calculation of the average SE thread holding time τ_{se}.

4.5 I/O Sub-System Model

The I/O sub-system consists of N_{buf} network output buffers, served by a single I/O controller in a round-robin (polling) fashion. We assume that incoming requests form a Poisson process with rate λ_{buf}, each with an exponentially distributed buffer service time with mean τ_{buf}. We model the assignment of I/O buffers as an $M(\lambda_{buf}) / M(\tau_{buf}) / N_{buf} / \infty$ queueing system, where the average buffer holding time τ_{buf} captures the impacts of the I/O controller and the TCP flow control.

Specifically, the I/O sub-system holding time is a function of the average number of cycles n_{cycle} required by the I/O controller to send the file. The average number of cycles, in turn, is determined by the dynamics of the TCP flow control as follows: assuming that file size is geometrically distributed, the expected number of cycles n_{cycle} is given by

$$n_{cycle} = p\sum_{i=1}^{\infty} E\{\#\text{ cycles} \mid \text{file size}=i\}\,(1-p)^{i-1} \text{ for } p = 1/n_{file}, \quad (5.1)$$

where the $E\{\text{number of cycles} \mid \text{file size} = i\}$ is chosen to reflect the TCP flow control algorithm and network congestion scenario.

For instance, consider an optimistic scenario (see Table 1): if the window size doubles with each successful transmission (1 block, then 2 blocks, then 4 blocks, then 8 blocks, etc), then the $E\{\text{number of cycles} \mid \text{file size} = i\} = 1, 2, 2, 3, 3, 3, 3, 4, 4, 4,\ldots$ for file size $i = 1, 2, 3,\ldots$

Table 1: Number of Cycles vs. File Size (Optimistic Scenario)

File Size (blocks)	Cycle 1 WS = 1	Cycle 2 WS = 2	Cycle 3 WS = 4	Cycle 4 WS = 8	**Number of Cycles**
1	1				**1**
2	1	1			**2**
3	1	2			**2**
4	1	2	1		**3**
5	1	2	2		**3**
6	1	2	3		**3**
7	1	2	4		**3**
8	1	2	4	1	**4**
9	1	2	4	2	**4**
10	1	2	4	3	**4**

On the other hand, consider a pessimistic scenario (see Table 2): if the window size follows the example sample path: 1 block, then 2 blocks, then 4 blocks (2 of which are not acknowledged), then 1 block (back-off due to congestion), etc, then the E{number of cycles | file size = i} = 1, 2, 2, 3, 3, 4, 5, 5, 6, 6,… for i = 1,2,3,… The average window size (in blocks) is then given by n_{file} / n_{cycle} .

Table 2: Number of Cycles vs. File Size (Pessimistic Scenario)

FileSize (blocks)	Cycle1 WS=1	Cycle2 WS=2	Cycle3 WS=4	Cycle4 WS=1	Cycle5 WS=2	Cycle6 WS=4	**# of Cycles**
1	1						**1**
2	1	1					**2**
3	1	2					**2**
4	1	2	1				**3**
5	1	2	2				**3**
6	1	2	2	1			**4**
7	1	2	2	1	1		**5**
8	1	2	2	1	2		**5**
9	1	2	2	1	2	1	**6**
10	1	2	2	1	1	2	**6**

Next, the average elapsed time for each cycle τ_{cycle} consists of the waiting time ω_{link} to acquire control of the output link, plus the time τ_{link} to put a chunk of data on the output link, plus the PC modem time τ_{modem} to read the data, plus the network RTT τ_{net} to send the data and return the ACK. Therefore, τ_{cycle} is given by

$$\tau_{cycle} = \omega_{link} + \tau_{link} + \tau_{net} + \tau_{modem} . \quad (5.2)$$

Equation 5.2 is optimistic from the standpoint that the model assumes that network and modem service times are independent of the load on the Web server (no contention between requests). This assumption is reasonable for most network environments, since one server typically will not perturb a network. However, this assumption may be optimistic for the client modem, where (up to 4) files from the same Web server may be downloading in parallel. Therefore, a pessimistic bound may be obtained by multiplying τ_{modem} by 4 in equation 5.2.

The link holding time is clearly impacted by the server load. Therefore, we model the output link as an $M(\lambda_{link}) / M(\tau_{link}) / 1 / \infty$ queueing system (where $\lambda_{link} = n_{cycle} \lambda_{buf}$) in order to compute the waiting time ω_{link} to acquire control of the output link.

Finally, the average number of occupied buffers n_{buf}, the probability that all N_{buf} output buffers are occupied (the probability of queueing) π_{ioq}, the average number of queued files (that is, the average number of HTTP/SE threads that are blocked, waiting for an output buffer) q_{ioq}, and the average waiting time (that is, the average time that the HTTP/SE thread is blocked, waiting for a network output buffer) ω_{ioq} follow directly from basic results for the M/M/N/∞ queueing system.

4.6 End-to-End Performance

The system of equations defined in sections 4.2-4.5 can easily be solved iteratively until convergence. Although we have not yet proven that the algorithm is guaranteed to converge to a unique solution, we have always reached convergence in practice. For points outside of a very narrow region around the bottleneck saturation throughput (clarified in section 5), the computational cost is trivial (10's of iterations, and 10's of CPU milliseconds). For points within this very narrow region, convergence is achieved through judicious use of a damping algorithm to minimize oscillations, and computational cost is still very reasonable (100's of iterations, and a few CPU seconds).

Once the iteration is complete, we can compute the end-to-end, user-perceived performance metrics. First, the user-perceived file "connect" time τ_{conn} consists of the TCP service time plus the HTTP waiting time. Accordingly, the average user-perceived "connect" time τ_{conn} is given by

$$\tau_{conn} = \tau_{net} + \omega_{http} . \quad (6.1)$$

The user-perceived "response" time τ_{resp} consists of the HTTP processing time, plus the SE waiting and execution times, plus the I/O waiting and service times. Accordingly, the average user-perceived "response" time τ_{resp} is given by

$$\tau_{resp} = \tau_{proc} + \omega_{se} + \tau_{exec} + \omega_{ioq} + \tau_{buf} . \quad (6.2)$$

The end-to-end service time is then the sum of "connect" and "response" times, $\tau_{conn} + \tau_{resp}$. Also, the effective end-to-end file "connect" rate λ_{conn} is given by

$$\lambda_{conn} = \lambda_{file} (1-\beta_{tcp}) (1-\beta_{http}) (1-\beta_{se}) / (1- \pi_{retry} \beta_{tot}), \quad (6.3)$$

and the effective end-to-end file "error" rate λ_{error} is given by

$$\lambda_{error} = \lambda_{file} - \lambda_{conn} = \lambda_{file} (1-\pi_{retry}) \beta_{tot} / (1- \pi_{retry} \beta_{tot}). \quad (6.4)$$

5. BASIC MODEL VALIDATION

A simulation model was written to validate the accuracy of the basic analytic approximation. The simulation assumes Poisson arrivals and geometric file sizes, but not exponential holding times. Another difference is that the simulation faithfully captures sub-system interactions and correlations. The simulation has been validated extensively against lab measurements; the results of that validation are documented in [17,16]. The test lab environment used more realistic arrival patterns, service times and file sizes. In addition, the simulation has been validated against a real-world production environment with actual users, authentic arrival patterns, and realistic Web content and scripting logic [8].

We performed a variety of numerical experiments. For a number of realistic scenarios, we computed the server throughput, blocked load, and average end-to-end response time as a function of the transaction request rate (TRR). In order to demonstrate the enhancements over the proposed model in [13], we consider the same two Internet scenarios. The results are outlined below.

In the first scenario, the Web server is connected to the Internet via a T1 (1.544 Mbps) line. Clients are connected to the Internet via 28.8 Kbps modems. The Internet RTT is exponentially distributed with mean 250ms. The Web server runs on a single CPU machine, and the total CPU time required by an HTTP thread is 60ms/file. Clients do not reattempt when blocked. The TCP LQ size is 1024, the HTTP LQ size is 128, the number of HTTP threads is 128, and the number of I/O buffers is 256. The MTU size is 512B, and the average output file size is 4KB. This scenario is realistic and typical for clients connected to a Web server via the Internet.

The effective server throughput λ_{conn}, the blocked load λ_{error}, and the average end-to-end service time $\tau_{conn} + \tau_{resp}$, are shown in Figures 4a-c (respectively) as a function of the TRR λ_{file}. As can be seen in Figure 4a, the throughput increases linearly up to some threshold T*, and remains constant as the TRR exceeds T*. One can readily verify that this maximum throughput T* corresponds to the performance limit of the CPU (1/0.06 ≈ 17 requests/sec). As can be seen in Figure 4b, the blocked load remains constant at 0 up to T*, then increases linearly as the TRR exceeds T*.

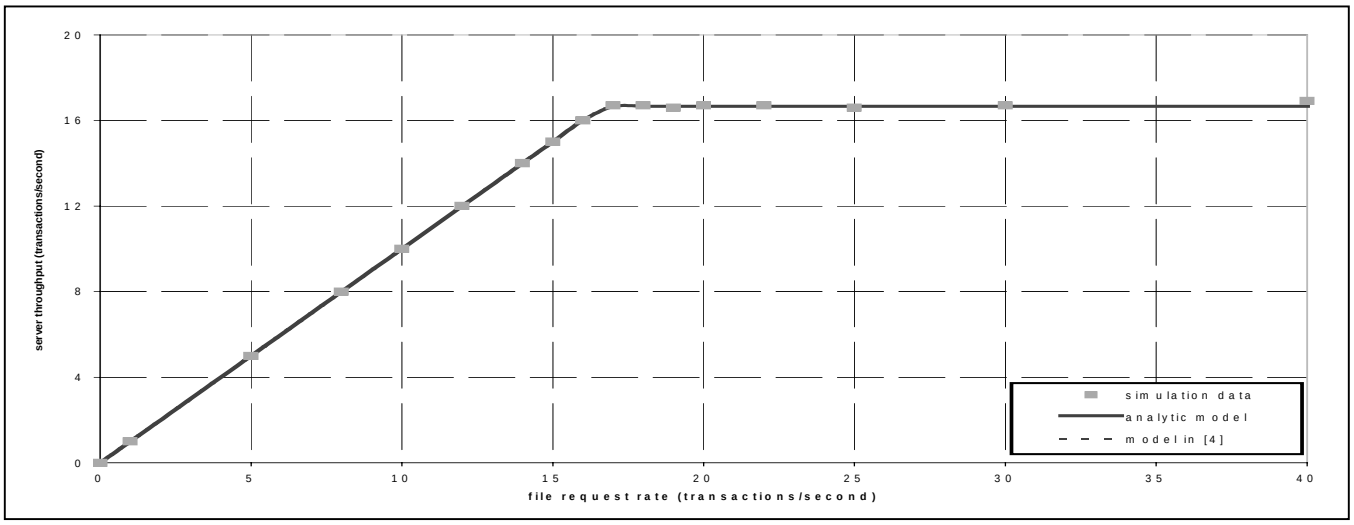

Figure 4a: Server Throughput vs. TRR – Internet Scenario 1

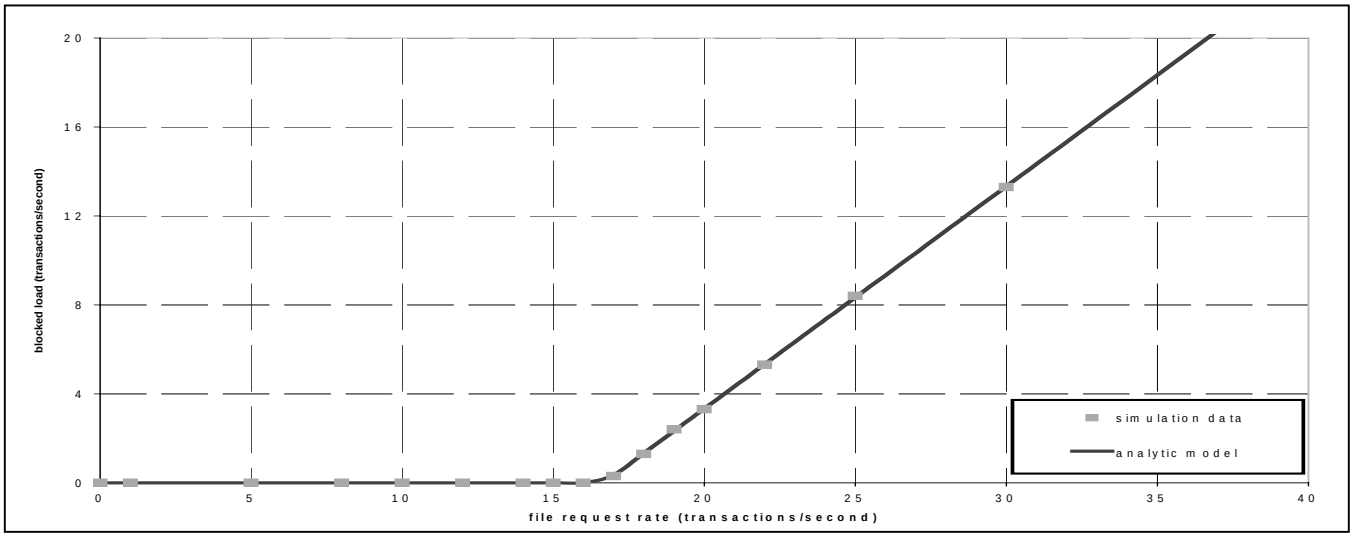

Figure 4b: Blocked Load vs. TRR – Internet Scenario 1

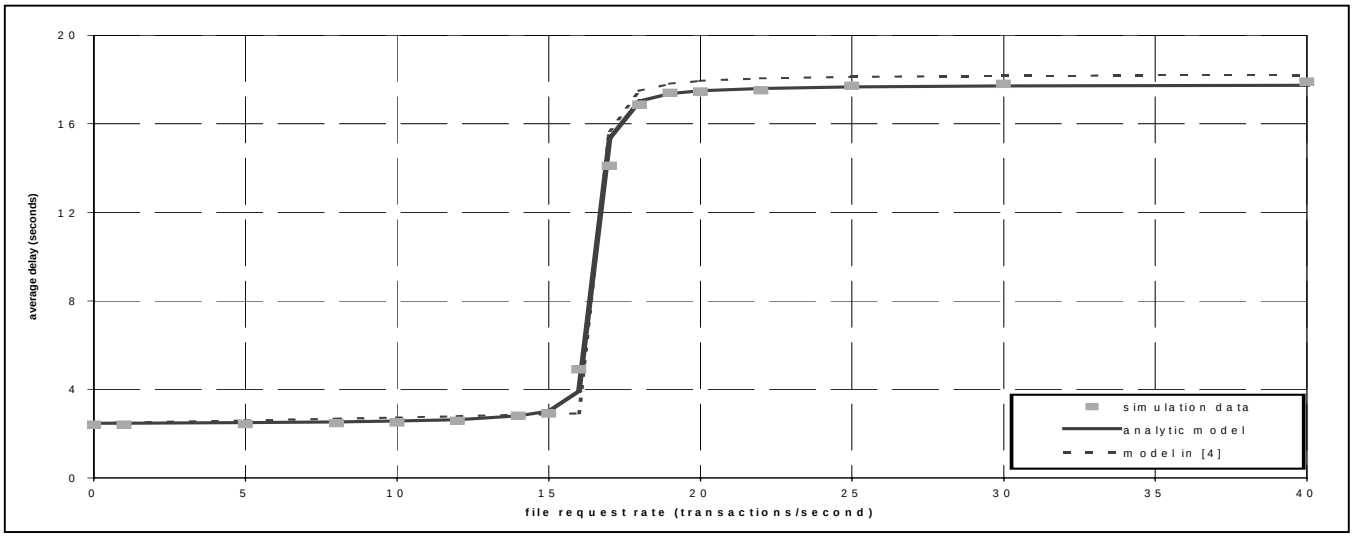

Figure 4c: Client Response Time vs. TRR – Internet Scenario 1

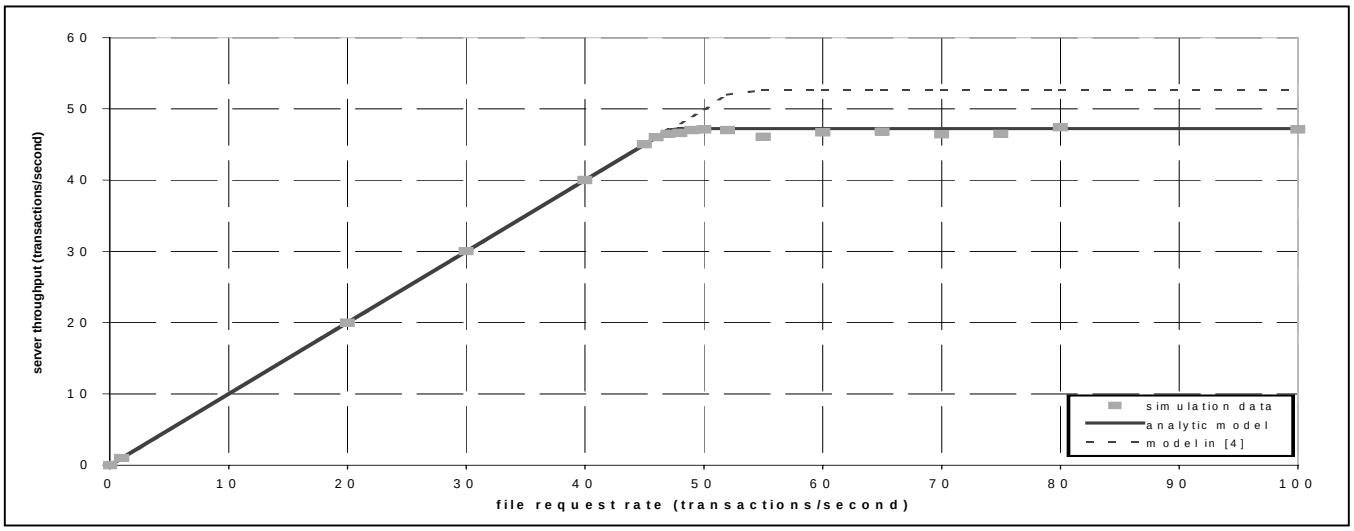

Figure 5a: Server Throughput vs. TRR – Internet Scenario 2

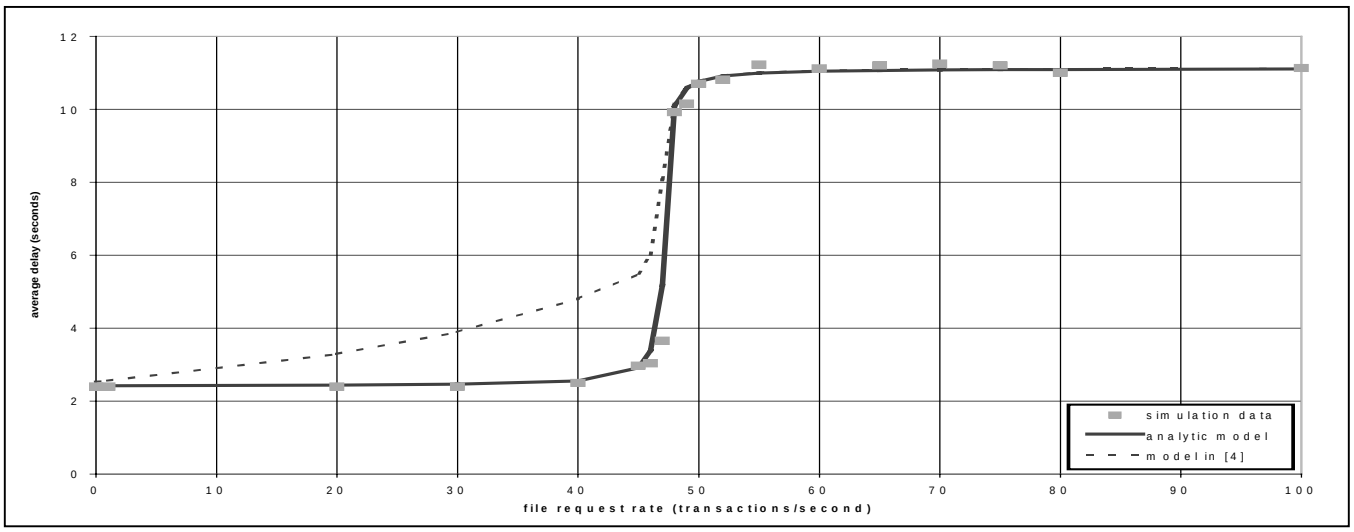

Figure 5b: Client Response Time vs. TRR – Internet Scenario 2

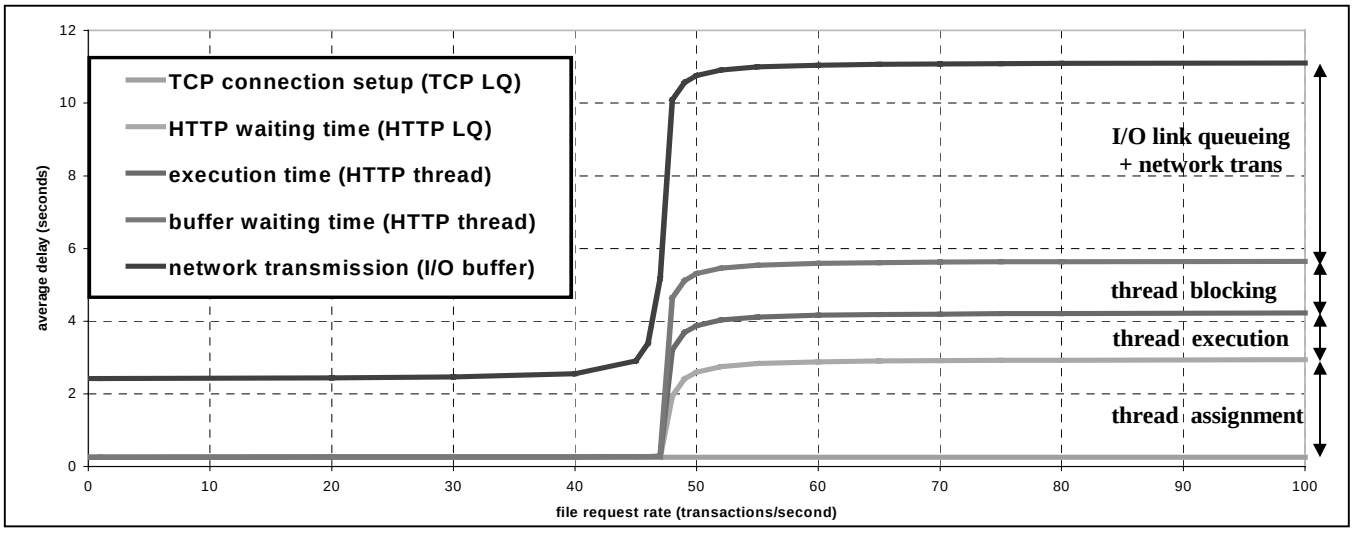

Figure 5c: Decomposition of Response Time vs. TRR – Internet Scenario 2

As can be seen in Figure 4c, the end-to-end response time remains fairly constant for TRR below T*, then transitions to a higher plateau when the TRR exceeds T* (where additional requests are blocked). These lower and upper plateaus can be readily explained as follows: in the case of the lower plateau, the contention between requests (thus, queueing delay) is negligible. Therefore, the end-to-end service time is determined by summing the no-load service times at each of the sub-systems. In the case of the upper plateau, the delay is determined by the bottleneck resource (in this case, the CPU) at the saturation load T*. Each request that is ultimately served (not blocked) arrives when all N_{http} threads are busy and Q_{http}-1 LQ slots are occupied (thus seizing the last available slot in the HTTP LQ). The remaining components of the end-to-end delay for the non-bottlenecked resources are then determined based on a TRR of T*.

We observe that the analytic approximation is highly accurate for all values of TRR. Furthermore, the model presented here yields an improvement over the model proposed in [13]. In particular, the enhancement to the HTTP holding time analysis (time-slicing instead of processor-sharing) more accurately captures both the “knee” and "plateau" of the response time curve.

In the second scenario, consider what happens if the HTTP processing bottleneck is alleviated (e.g., by upgrading the CPU or improving the cache performance). Assume that total CPU time is decreased from 60ms to 10ms. For this scenario, Figures 5a-b depict the throughput and response time as a function of TRR. Figure 5a shows that decreasing the required CPU time by a factor of 6 increases the maximum throughput by only a factor of less than 3, suggesting that the bottleneck has moved to another sub-system. In fact, one can readily verify that the capacity of the T1 link has become the limiting bottleneck, restricting the maximum throughput to T* = 1544/32 ≈ 47 requests/sec.

Again, we observe that the analytic approximation is highly accurate for all values of TRR. Furthermore, the model presented here again yields a dramatic improvement over the model proposed in [13]. In particular, the enhancement to the I/O buffer holding time analysis more accurately captures the throughput threshold (Figure 5a) and response time "knee" (Figure 5b).

In Figure 5c, we decompose the end-to-end response time into the various sub-system component delays. In particular, Figure 5c shows the cumulative contributions from the TCP LQ (connection setup), the HTTP LQ (waiting for thread assignment), the HTTP thread (request execution time as well as waiting time for buffer assignment), and the I/O buffer (network transmission).

As can be seen, at low loads the delay is dominated by the modem read time (4KB @ 28.8Kbps ≈ 1.1s) and the network RTT (1 for TCP setup + 4 for data transmission ≈ 1.3s). At high loads, the delay consists of 1.1s modem time and 1.3s transmission time (as before), plus 3.3s waiting for the network link, 1.4s waiting for an output buffer, 1.3s HTTP thread execution time, and 2.7s waiting for an HTTP thread. These values are consistent with our intuition and understanding of the Web server internals. As the network link (bottleneck) saturates, the I/O buffers fill, causing the HTTP threads to block, in turn causing the HTTP threads and LQ to saturate. Thus, the HTTP thread *execution* time consists of 128 [threads] x 0.01s [CPU time] ≈ 1.3s. The HTTP thread *holding* time consists of 1.3s [execution] + 1.4s [waiting] ≈ 2.7s. Thus, the HTTP LQ waiting time consists of 128 [LQ slots] x 2.7s [thread holding time] ÷ 128 [threads] ≈ 2.7s.

Note that even though the I/O sub-system is the bottleneck in this scenario, it still accounts for less than 1/2 of the end-to-end delay (due to sub-system interactions and congestion migration).

6. MODEL EXTENSIONS

6.1 HTTP 1.1 (Persistent Connections)

The basic model assumes that the TCP connection is set up and torn down for each transaction (file) request (HTTP 1.0). With HTTP 1.1, multiple file requests can be "pipelined" across the same (persistent) TCP socket [10]. This protocol enhancement is easily accommodated in the analytic model by adding a feedback loop before the HTTP sub-system, as shown in Figure 6.

Let n_{conn} denote the average number of file requests per TCP connection. Then the probability of feedback $\pi_{feedback}$ is given by $(n_{conn} - 1) / n_{conn}$. For example, if the requested Web page consists of 1 index file plus 11 inline files (images, applets, etc), and the browser allows up to 4 concurrent sockets per page request, then $n_{conn} = 12/4 = 3$ files/connection and $\pi_{feedback} = 2/3$.

6.2 Non-Poisson Arrival Processes

The basic model assumes that incoming TCP-level file requests form a Poisson process. Clearly, this assumption is approximate. It has been shown that HTTP application-level page requests are reasonably modeled by a Poisson process [11], but IP-level packet requests exhibit long-range dependence [c.f. 18] that is likely caused by TCP flow control [1]. File requests therefore exhibit some measure of "burstiness" that falls in between these two cases (that is, file requests are more bursty than page requests, but less correlated than packet requests).

There are a number of approaches to capturing this burstiness and correlation that are easily amenable to the analytic modeling techniques employed here. First, we can assume that TCP file requests form a batch Poisson process, with mean batch size equal to the average number of files per page. Second, we can assume that file requests are modeled by a Markov-modulated Poisson process [7]. Finally, we can use any number of two-moment approximations (e.g. [12]).

Note that HTTP 1.1 (section 6.1) drives the arrival process closer to that of page requests (i.e., a Poisson process). That is, the behavior of the TCP connection request arrival process falls in between that of page requests and that of file requests. In other words, HTTP 1.1 tends to reduce the burstiness and correlation between arrivals to the TCP sub-system, making the Poisson assumption of the basic model somewhat more reasonable.

6.3 Heavy-Tailed File Distributions

The basic model assumes that HTTP thread, SE thread, and I/O buffer holding times are exponentially distributed. Again, these assumptions are clearly approximate. It has been shown that sizes of static files (i.e., those without significant dynamic content) are heavy-tailed [c.f. 4]. To our knowledge, there is no empirical evidence that dynamic files (i.e., script execution times)

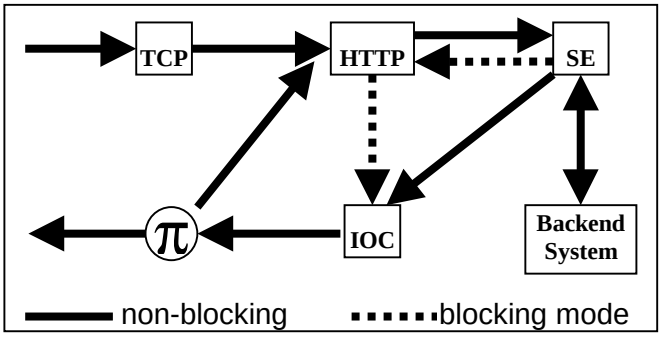

Figure 6: Modes of HTTP/SE Implementation

are similarly heavy-tailed. Thus, it is not clear that these exponential assumptions are unrealistic in the case of Web servers with significant server-side processing (the focus of this analysis). Nevertheless, it is appropriate to consider a model where holding times are heavy-tailed, so that the model can be easily "tuned" to the particular application workload under consideration.

It has been shown [4] that heavy-tailed file sizes can be reasonably modeled by multiple job classes, each with different arrival rates and holding times. The analysis here can be modified to accommodate multiple job classes. A number of reasonable approximations have been proposed to compute the blocking seen by different job classes offered to a common service system (typically referred to as "parcel" blocking in teletraffic literature). Many of these approaches arose out of the need to size secondary trunk groups that handle peaked overflow traffic from multiple primary trunk groups. Fredericks [6], for example, proposes an analytic approximation that can be readily incorporated into the basic model, thereby accommodating heavy-tailed distributions.

7. CONCLUSIONS & FURTHER WORK

In this paper, we have presented a detailed analytic queueing approximation to predict the user-perceived performance provided by Web servers with significant server-side processing in distributed computing environments. The "basic" analytic model has been validated against simulation as well as lab test measurements, and has been shown to be remarkably accurate for the scenarios modeled to date. In addition, the mechanics to extend the model to accommodate non-Poisson arrivals, long-range dependence, heavy-tailed distributions, and persistent connections have been outlined. The resulting "extended" model is well suited to handle a diverse range of application workloads. This analytic model forms an excellent basis for development of a decision support tool for evaluating the performance of Web servers in distributed environments, allowing system architects to predict the behavior of new systems prior to their deployment, or the behavior of existing systems under new workload scenarios.

The analytic model has not yet been validated against a real-world production scenario. However, a variant of the simulation model used to benchmark the analytic model has been validated against an actual distributed Web server deployment, and has been shown to be accurate. This evidence suggests that while the real-world characteristics (non-Poisson arrivals and heavy-tailed service times) do impact performance at the packet level, they do not strongly manifest themselves in the average end-to-end user-perceived application performance measures (such as response time and blocking). Therefore, we are optimistic that the analytic model will be reasonably accurate when benchmarked against real-world scenarios (despite its simplifying assumptions).

Further work is planned to incorporate the extensions outlined in section 6. First, we plan to add a feedback loop to model the effects of HTTP 1.1. Next, we plan to introduce a two-moment (mean and peakedness [c.f. 5]) characterization of the arrival process, with two job classes. We can then treat each sub-system as a separate $GI_2 / M_2 / N / Q$ renewal system (along the lines of [12] and [6]), and solve the system of equations iteratively (as with the present model).

8. ACKNOWLEDGMENTS

The authors would like to thank the referees on the WOSP technical program committee for their insightful comments.

9. REFERENCES

[1] Arvidsson and Karlsson, “On Traffic Models for TCP/IP”, proceedings of the 16th ITC, Teletraffic Engineering in a Competitive World, Elsevier ©1999.

[2] Conway, “A Perspective on the Analytical Performance Evaluation of Multi-Layered Communication Protocol Architectures”, IEEE Journal on Selected Areas in Communications ©1991.

[3] Cooper, Introduction to Queueing Theory, 2nd Edition, North Holland ©1981.

[4] Crovella and Bestravos, “Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes”, proceedings of the ACM/SIGMETRICS conference, ACM ©1996.

[5] Eckberg, “Generalized Peakedness of Teletraffic Processes”, proceedings of the 10th ITC ©1983.

[6] Fredericks, “A New Approach to Parcel Blocking via State Dependent Birth Rates”, Bell Laboratories internal memorandum MM73-3425-3, 29 June 1973.

[7] Grandell, Doubly Stochastic Poisson Processes, Springer-Verlag ©1976.

[8] Hariharan, van der Mei, Ehrlich, Reeser, “Modeling the Performance of Web Servers Engaged in OO Computing”, accepted for publication in an upcoming special issue of IEEE Transactions on Modeling and Computer Simulation.

[9] Menascé and Almeida, Capacity Planning for Web Performance, Prentice Hall PTR ©1998.

[10] Padmanabhan and Mogul, “Improving HTTP Latency”, proceedings of the 2nd WWW Conference ©1994.

[11] Paxson and Floyd, “Wide Area Traffic: Failure of Poisson Modeling”, IEEE/ACM Transactions on Networking ©1995.

[12] Reeser, “Simple Approximation for Blocking Seen by Peaked Traffic with Delayed, Correlated Reattempts”, proceedings of 12th ITC, Teletraffic Science for New Cost-Effective Systems, Networks and Services, Elsevier ©1989.

[13] Reeser, van der Mei, Hariharan, “An Analytic Model of a Web Server”, proceedings of the 16th ITC, Teletraffic Engineering in a Competitive World, Elsevier ©1999.

[14] Slothouber, “A Model of Web Server Performance”, StarNine Technologies ©1996.

[15] Stevens, TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley ©1994.

[16] van der Mei, Ehrlich, Reeser, Francisco, “A DSS for Tuning Web Servers in Distributed OO Network Architectures”, proceedings of the 2nd WISP, ACM ©1999.

[17] van der Mei, Hariharan, Reeser, “Web Server Performance Modeling”, proceedings of 4th Informs Telecom Conference, special issue of Telecommunication Systems ©2000.

[18] Willinger, Taqqu, Sherman and Wilson, “Self-Similarity Through High Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level”, IEEE/ACM Transactions on Networking ©1997.