A Decision Support System for Tuning Web Servers in Distributed Object Oriented Network Architectures

R.D. van der Mei
AT&T Labs
Middletown, NJ, USA
+1 732 420 3716
rvandermei@att.com

W.K. Ehrlich
AT&T
Red Hill, NJ, USA
+1 732 615 5721
wehrlich@att.com

P.K. Reeser
AT&T Labs
Middletown, NJ, USA
+1 732 420 3693
preeser@att.com

J.P. Francisco
AT&T
Red Hill, NJ, USA
+1 732 615 4132
shemp@att.com

ABSTRACT

Web technologies are currently being employed to provide end user interfaces in diverse computing environments. The core element of these Web solutions is a Web server that is based on the Hypertext Transfer Protocol (HTTP) running over TCP/IP. Web servers are required to respond to millions of transaction requests per day at an "acceptable" Quality of Service (QoS) level with respect to the end-to-end response time and the server throughput. In many applications, the server performs significant server-side processing in distributed, object-oriented (OO) computing environments. In these applications, a Web server retrieves a file, parses the file for scripting language content, interprets the scripting statements and then executes embedded code, possibly requiring a TCP connection to a remote application for data transfer. In this paper, we present an end-to-end model that addresses this new class of Web servers that engage in OO computing. We have implemented the model in a simulation tool. Performance predictions based on the simulations are shown to match well with performance observed in a test environment. Therefore, the model forms an excellent basis for a Decision Support System for system architects, allowing them to predict the behavior of systems prior to their creation, or the behavior of existing systems under new load scenarios.

Keywords

World Wide Web, HTTP, Web server, httpd, performance, distributed, object-oriented, computing, architecture, configuration tuning, Decision Support System

1. INTRODUCTION

Over the past few years, the World Wide Web has experienced tremendous growth, which is not likely to slow down in the near future. The explosion of Internet Commerce service offerings [1] 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 [2] to end-to-end voice communications that use the Internet for packetized voice transport [3]. At the heart of these Web solutions is a Web server that is based on the Hypertext Transfer Protocol (HTTP) running over TCP/IP. Web servers are required to respond to millions of transaction requests per day at an "acceptable" QoS level with respect to the end-to-end response time and the server throughput. 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 Web servers is crucial.

Web technologies are currently being employed to provide end user interfaces in distributed computing environments, possibly requiring a connection to a remote application for data transfer (see Figure 1). In many applications, the server performs significant server-side scripting. To this end, 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 side. To overcome this, Web servers may 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.

One approach to performing server-side scripting is to implement a script-engine dedicated to process server-side scripts. A typical example of a script engine implementation is the Active Server Pages (ASP) technology in Microsoft's Internet Information Server (IIS) running on Windows NT. In ASP applications, IIS retrieves a file, parses the file for scripting language content, and interprets the scripting statements. Since a script is interpreted, a complex script may slow down the script engine. Consequently, some script engines (e.g., VBScript, JavaScript) enable instances of objects (e.g., compiled C++ or Java code) to be created on the Web server. In an object-oriented (OO) Web environment, an object's methods, properties and events are directly accessible from the script.

Web server performance in a distributed, OO environment is a complex interplay between a variety of components (e.g., hardware platform, threading model, object scope model, server operating system, network bandwidth, disk file sizes, caching). However, existing models [4,5] fail to address servers with significant server-side processing that participate in distributed, OO computing. In this paper, we present an end-to-end

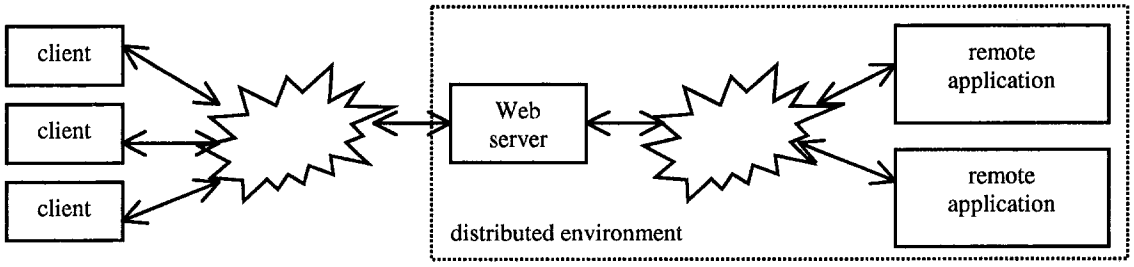

Figure 1. Illustration of a Web server in a distributed computing environment

performance model for the communication between the client and the server performance that incorporates server-side processing in a distributed OO environment. In general, the transaction flows depend on the implementation of the Web server and on the Operating System. In this paper, we focus on the dynamics of the ASP technology for the Microsoft IIS server for Windows NT. However, we emphasize that analogous constructs are also applicable beyond the IIS server. We have implemented the model in a simulation tool. The model is validated by comparing performance predictions based on the model to performance observed in a test environment. The simulation tool forms an excellent basis for the development of a Decision Support System for evaluating the performance of Web servers in a distributed OO environment, allowing system architects to predict the behavior of systems prior to their creation, or the behavior of existing systems under new load scenarios.

2. TRANSACTION FLOWS

Each HTTP transaction proceeds through a Web server along four successive phases: (1) TCP connection setup, (2) HTTP layer processing, (3) script-engine processing, and (4) network I/O processing. The different phases (see Figure 2) are discussed in more detail in sections 2.1 to 2.4.

2.1 TCP Connection Setup Phase

Before information can be exchanged between the client and server, a two-way connection (a TCP socket) must be established. The TCP sub-system consists of a TCP Listen Queue (TCP-LQ) served by a server daemon. A TCP connection is established by the well-known three-way handshake procedure (see [6] for details). Immediately after the TCP socket has been established, the transaction request is forwarded to the HTTP sub-system for further processing. If all slots in the TCP-LQ are occupied upon arrival of a connection request, then the request is rejected and the client must resubmit a connection setup request.

The TCP connection setup phase is modeled by a multi-server blocking model and zero waiting buffer space. Each "server" represents a slot in the TCP-LQ, and the number of servers equals the size of the TCP-LQ. Each customer represents a connection request. If an incoming customer finds all servers busy (i.e., all slots are occupied by other pending connection requests), then the customer is rejected; otherwise, the customer is taken into service immediately. A service time represents the time between the arrival of the connection request at the TCP-LQ and the time at which the three-way handshake is completed. In this way, the service time of a customer corresponds to one round-trip time (RTT) between the server and the client.

2.2 HTTP Layer Processing Phase

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

1. If an HTTP thread is available, then the thread retrieves the requested file. If the file requires script processing, then the transaction is forwarded to the script engine for further processing (see section 2.3); otherwise, the file content is forwarded to the I/O sub-system (see section 2.4).
2. If all I/O buffers are occupied at that time, then the HTTP thread remains idling until an I/O buffer becomes available.
3. If there is no HTTP thread available, then the transaction request enters the HTTP-LQ (if possible), and waits until it gets assigned a thread to handle the request.
4. If the HTTP-LQ if full, then the transaction request is rejected, the TCP connection is torn down, and the clients receives a "connection refused" message.

The HTTP sub-system can be modeled by a multi-server finite-buffer blocking system. The servers represent the HTTP threads, the customers represent transaction requests, and the buffer represents the HTTP-LQ. The number of servers equals the number of threads, and the buffer size is the length of the HTTP-LQ. If a server is available, then the customer is taken into service immediately. Otherwise, the customer enters the HTTP-LQ; if the queue is full, then the customer is rejected. The customer occupies the thread from the time at which a thread starts to handle the transaction request until the thread becomes available for handling another transaction request.

2.3 Script Engine Processing Phase

The dynamics of a script-engine (SE) generally depend on the Web server implementation and on the Operating System. In this section, we focus on the dynamics of the Active Server Pages (ASP) technology for the IIS Web server running on Microsoft Windows NT. However, the constructs discussed below are also applicable beyond the IIS server. We emphasize that the aim of discussion is to give the reader a general understanding of the type of performance issues related to different SE implementations, not to discuss the ASP technology in great detail. To this end, the terminology used will be generic and may deviate from the ASP-specific terminology. The reader is referred to Microsoft documentation for extensive discussions of the ASP internals and the ASP terminology.

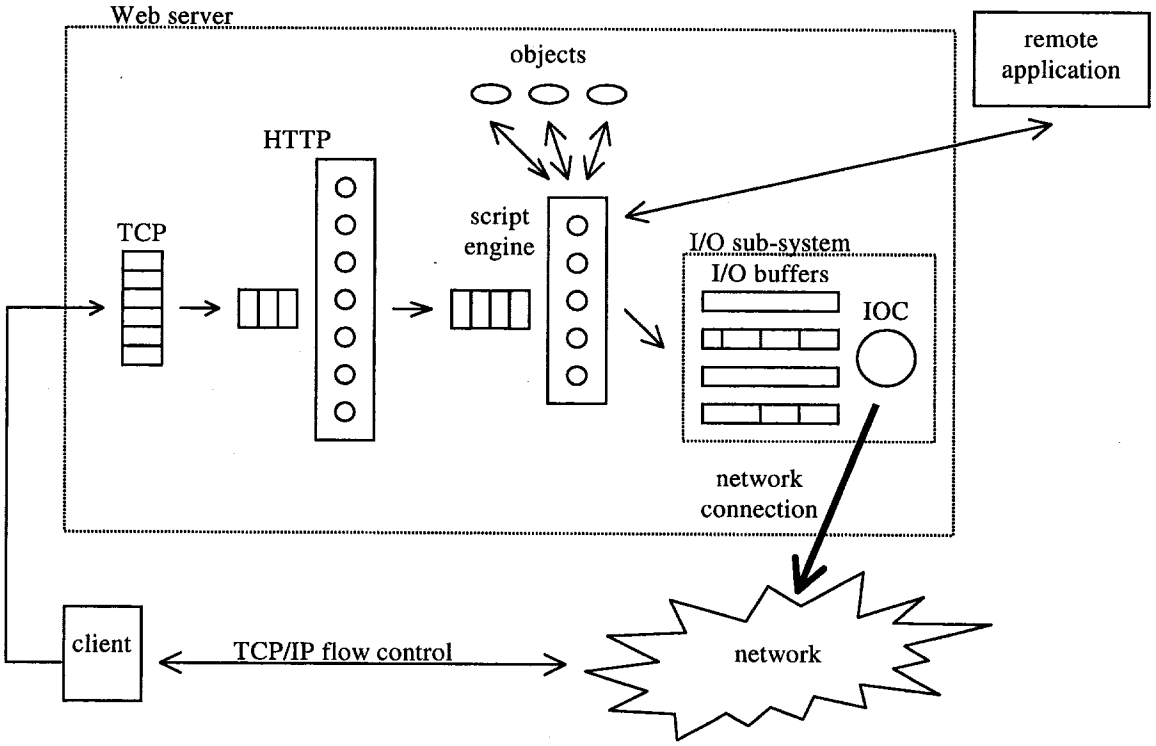

Figure 2. Model of the transaction flows within Web server

The SE sub-system is equipped with a SE Listen Queue (SE-LQ) and a pool of threads dedicated to interpreting scripting statements and executing embedded code (e.g., C++, Java). During object execution, communication with a remote backend server may be needed (e.g., to perform a database query), possibly requiring a TCP/IP connection to the backend server to be established and subsequently torn down. The transaction flows related to the SE depend on the object scoping and threading models. Below we outline the basic ideas of object threading and scoping models (for the ASP technology), and their impact on the performance of the SE sub-system.

2.3.1 Object Threading Model

Objects can be classified as single-threaded or multi-threaded.[1] A single-threaded object has so-called thread affinity. That is, the object's methods can only be executed by a dedicated thread, so that the object can never be accessed concurrently. If another thread needs to access the object's methods, it will have to request the thread that owns the object to access the object on its behalf. Hence, all method calls to a single-threaded object must be serialized on a specific thread. In contrast, a multi-threaded object is not owned by a specific thread, and can be accessed concurrently by multiple threads, with no guarantee concerning which thread will execute a given method invocation. Implementers must therefore protect resources (e.g., shared static variables) used by a single instance of the object against concurrent access. We refer to [7] (and references therein) for more details on threading models.

2.3.2 Object Scope

Objects are also classified according to their so-called scope. The scope of an object is associated with the lifetime of the object. Two extreme examples are Transaction Scope Objects[2] (TSOs) and Application Scope Objects (ASOs). A TSO lives only for the duration of a single object request. A TSO is created, executed, and dereferenced over the course of the request. Multiple TSO object requests result in multiple instances of the object. In contrast, an ASO lives for the complete duration of the application. An ASO is created when the application is started. Only a single instance of the object will exist for the duration of the application. Session Scope Objects (SSOs) fall in between TSOs and ASOs in the sense that they live only for the duration of a user's session. The reader is referred to [9,10] (and references therein) for more information on object scoping models.

2.3.3 Performance Modeling

The object scoping and threading models may have a strong impact on the performance of the Web server. The dynamics of the SE sub-system depend on the scoping and threading models, which lead to different performance models, depending on the

[1] In ASP terminology, an object can live in a so-called apartment, which can be single-threaded or multi-threaded. An object can be "Single-", "Apartment-", "Free-" or "Both-" threaded.

[2] Also referred to as Page Scope Objects.

implementation of the Web server. Several possible performance modeling approaches are addressed below.

The simplest model is obtained in the case when all objects are TSOs. A TSO is created, executed, and dereferenced over the course of a request by a specific thread. Because the TSO only lives during the course of a single request, there is no concurrency in accessing the TSO by other threads. Therefore, this scenario can be modeled by a multi-server queueing model, where the servers represent the threads, the customers represent transaction requests entering the SE sub-system, and the time needed to create, execute, and dereference the TSO is incorporated into the service time.

A different performance model is obtained for single-threaded ASOs (and SSOs), which are owned by a specific thread, T*, but which are not dereferenced after execution. In this case, the ASO's methods can only be accessed by T* and transaction requests handled by other threads that need to execute the ASO are serialized through T* to execute the object. These dynamics can be modeled by a single-server queue, where the server represents T*, the customers represent transaction requests owned by other threads that need to access the ASO, and the service times represent the CPU time required to serialize a transaction and to execute the ASO. Note that it may be possible (although not desirable) that T* owns multiple ASOs, which leads to a similar model.

Another situation arises for multi-threaded ASOs (or SSOs), which are not owned by a specific thread, and can be accessed concurrently by multiple threads. This type of situation may be modeled by a multiple- (possibly infinite-) server queueing model, where the servers represent the access ports to the concurrently accessible ASO, and the customers represent transactions or threads that need to access the ASO. The speed at which the active servers work generally depends on the amount of "critical sections" in the ASO object code. In the extreme case where all sections are critical, the server node may be a Processor Sharing node, i.e., where the speed of the active servers is inversely proportional to the number of active servers (see for example [11]). In the other extreme case where the amount of critical sections is negligible, the server speed may be assumed to be independent of the number of active servers.

We reemphasize that the discussion of the modeling of different combinations of threading and scoping models is intended only to give a global idea of the type of performance issues related to different threading and scoping models in an OO environment. Clearly, refinement of the models may be required in specific implementations.

2.4 I/O Processing Phase

The I/O sub-system consists of a number of parallel I/O buffers, an I/O controller (IOC), and the connection from the server to the network. The contents of the I/O buffers are "drained" over the network connection to the network. The draining of the different I/O buffers over the network connection is 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 the client is based on the TCP flow control mechanism (see [6] for 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 at a time. 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 window mechanism implies that a block of a file residing in an output buffer can only be transmitted if the TCP window is open. Notice that the arrival of acknowledgments generally depends on the congestion on the network. Therefore, the rate at which I/O buffers can drain their contents may be affected by congestion on the network.

The dynamics of the I/O subsystem can be modeled as a single-server multi-queue polling model with finite buffers. Each queue represents an I/O buffer. The server represents the IOC, and the service times represent the time to transmit a file block. When the server arrives at a (non-empty) queue, it transmits one or more file blocks (depending on the TCP window size); if the window is closed, then the server immediately proceeds to the next queue. We assume that the time for the server to proceed from one queue to the next is negligible. The dynamics of the TCP flow control mechanism can be modeled along the lines of Heidemann et al. [4] for a variety of slow-start algorithms.

To understand the end-to-end performance of the Web server, it is important to understand the interactions between the different sub-systems discussed in sections 2.1-2.4. To this end, let us consider what happens if the network connection between the Web server and the client is congested for some time period. Then the network RTT increases, so that the TCP acknowledgments (from the client to the 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 and 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 to "dump" the output content. Since the threads are idling as long as they are waiting for an I/O buffer to become available, the availability of the threads will go down, and the HTTP and SE LQs will tend to fill up and overflow, which may lead to blocking of incoming transaction requests. In this way, performance problems in the network may imply performance problems in the Web server itself.

3. IMPLEMENTATION

To obtain and validate performance predictions based on the model discussed in the previous section, we have implemented the model in a simulation tool called Q+. The tool provides a GUI for debugging and demonstration purposes. In addition, we have implemented a C Programming Interface (CPI) to the model. The CPI allows the user to disable the GUI and run the model in the background, which is significantly faster. The GUI implementation of the model (see Figure 3) is outlined below.

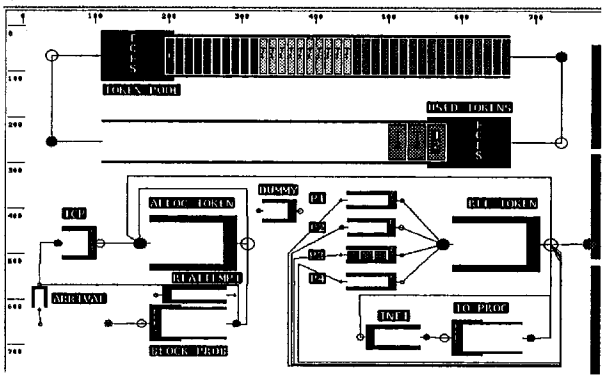

Figure 3. Implementation of the model

To implement the dynamics of the model, we have used a token-pool mechanism. Each entry in the TCP-LQ, each buffer spot in the HTTP-LQ or SE-LQ, each HTTP or SE thread, and each I/O buffer corresponds to a unique token. Free tokens reside in the TOKEN_POOL node, and occupied tokens reside in the USED_TOKENS node. Each transaction enters the system at the ARRIVAL node and leaves in the sink, either via the REL_TOKEN node (when the transaction has been performed successfully) or via the BLOCK_PROB node (when the transaction has failed). A transaction that needs to allocate a token (e.g., to reserve a buffer spot, to access a thread, or to reserve an I/O buffer), enters the ALLOC_TOKEN node. If no token of the type required is available (e.g., when a buffer is full, when all threads are occupied, or when all I/O buffers are occupied), then the transaction may either wait (idle) in the ALLOC_TOKEN node until a token becomes available (e.g., when allocating an I/O buffer), or the transaction may enter the REATTEMPT node (for reattempts) or the BLOCK_PROB node (when the transaction is rejected). When a transaction is assigned a thread, the transaction is executed on one of the processors (P1 to P4). A token is released by entering the REL_TOKEN node. The I/O buffers and IOC are implemented in the I/O_PROC node, the TCP flow control is implemented in the INET node, and backend communication is implemented in the BACKEND node (not shown in Figure 3).

The model is rather complex and has a fairly large number of input parameters. The main input parameters are the transaction request rate, the TCP-LQ size, the HTTP-LQ size, SE-LQ size, the number of HTTP threads, the number of SE threads, the HTTP thread CPU-time, the SE thread CPU time, the percentage of transactions that require script processing, the number of I/O buffers, the I/O buffer size, the average file size, the number of CPUs, the MSS, the network connection speed, the modem speed, the client-server acknowledgment Round Trip Time (RTT) distribution, the maximum TCP window size, the fraction of transactions that require backend communication, the server-backend RTT distribution, the reattempt probability, the distribution of the reattempt time interval, amongst others.

The output parameters in the current implementation are (a) the effective server throughput, (b) the end-to-end response time (mean and standard deviation), and (c) the fraction of transactions that are blocked at the TCP, HTTP and SE sub-systems, respectively.

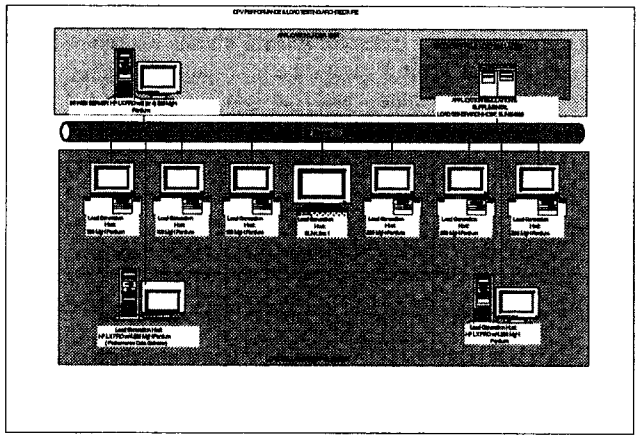

Figure 4. Test configuration

4. VALIDATION

To validate the simulation model, we compared model predictions with observed measurements taken on IIS under load test for a variety of datasets. The results are outlined below.

4.1 Test Environment

The Web Server test configuration is shown in Figure 4. The top left machine in the first row represents the Web Server under Test (SUT) executing on an HP LX Pro workstation. The SUN-E4000 shown in the top row on the right was used as a load generator machine for simulating client requests and as a backend-end server running an application that participated in distributed OO computing. The other workstations were used as load generators, while the leftmost machine in the bottom row also stored performance test data. The servers were connected via a 100Mbit/s Fast Ethernet LAN.

4.2 Results

We have performed experiments with a variety of datasets. The details are omitted for compactness of the paper. In all cases, the SUT was exposed to different transaction request rates, and the responsiveness of the SUT was measured in terms of the throughout (i.e., the number of transactions per time unit) and the residence time (defined as the time between the transaction entering service at the HTTP sub-system until the output file was

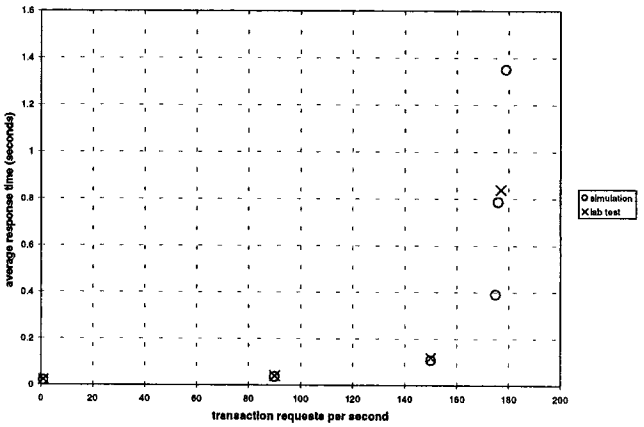

Figure 5. Simulated and empirical average residence times

entirely placed into an I/O buffer). Figure 5 shows the predicted and empirical residence times for several transaction request rates for dataset IV, and Table 1 shows the maximum throughput (TP), both the measured values and the values predicted by the simulation model, for the different datasets.

Dataset	Measured TP	Predicted TP
I	125	123
II	54	55
III	13	13
IV	177	179

Table 1. Measured and predicted maximum throughput for different datasets

The results in Figure 5 and Table 1 indicate that the simulation model results are consistent with the empirical test results.

5. MODEL EXTENSIONS

The model can be extended in several directions. First, in the current model a separate TCP connection is established and torn down for each individual file, which is known to degrade the server performance. To overcome this, one may use the concept of persistent connections, where multiple files are transmitted over the same TCP connection. It would be interesting to incorporate the concept of persistent connections into the model and study its impact of the end-to-end performance of the Web server. Second, in the current implementation of the model the objects are assumed to be of transaction scope. To study the impact of the object scoping models in more detail, we are currently implementing the dynamics of session scope and application scope objects into the model (see section 2.3 for more details on the modeling). Third, in this paper the modeling of the script-engine node has been focused on the dynamics of the Active Server Component (ASC) of the Microsoft's IIS server. It is a challenging topic for further research to incorporate other script engine implementations, such as servlet technology, into the model and study its impact on the end-to-end performance of the server.

6. CONCLUDING REMARKS

The final objective for both testing and modeling Web Server performance is to identify appropriate configuration guidelines for deploying Web Servers. Although TESTING is an important technique for assessing Web Server performance, it has several severe drawbacks:

1. load/stress testing is extremely time-consuming and tedious
2. testing alone is of limited applicability beyond the test workload and is not generalizable
3. testing alone cannot predict the performance tradeoffs in advance of major new software releases.

Hence, MODELING is critical to further understand the performance capabilities and limitations of Web servers. We reemphasize that a simulation model of a Web server is extremely useful as a Decision Support System for system architects, allowing them to predict the behavior of systems prior to their creation, or the behavior of existing systems under new load scenarios.

For Web servers that engage in OO computing, it is important to analyze factors that affect threads responsible for script and object execution. These factors include whether the scripting thread pool is synchronous versus asynchronous [8], whether objects are single or multi-threaded, whether the object executes within the Web server's process space, thread affinity, and thread message filtering. Note that these factors can be contrasted with factors previously investigated in Web server performance studies (e.g., file-size distribution, transaction request arrival process, TCP window control algorithms, different versions of HTTP). A simulation model offers an ideal tool for such an investigation.

7. REFERENCES

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

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

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

[4] Heidemann, J., Obraczka, K. and Touch, J. (1997), Modeling of the performance of HTTP over several transport protocols. IEEE Trans. Netw. 5, 616-630

[5] Slothouber, L.P., A model of Web server performance, http://louvx.biap.com/whitepapers/performance/overview/

[6] Stevens, R.W. (1994). TCP/IP Illustrated, Vol 1 (Addison Wesley)

[7] Box, D. (1998), Essential COM (Addison Wesley Longman, Inc.)

[8] Hu, J., Pyarali, I. and Schmidt, D.C. (1997), Measuring the impact of event dispatching and concurrency models on Web server performance over high-speed Networks, IEEE Proceedings of the 2nd Global Internet Conference

[9] Corning, M., Elfanbaum, S. and Melnick, D. (1997), Working with Active Server Pages (Que Corporation, Indianapolis, IN)

[10] Box, D., Active Server Pages and COM apartments, http://www.develop.com/dbox/aspapt.asp

[11] Kleinrock, L.K. (1976). Queueing Systems, Vol. 2 (Wiley & Sons, New York)