Inferring Client Response Time at the Web Server

David P. Olshefski
IBM T.J. Watson Research
19 Skyline Drive
Hawthorne, NY 10532
olshef@us.ibm.com

Jason Nieh
Dept. of Computer Science
Columbia University
1214 Amsterdam Avenue
New York, NY 10027-7003
nieh@cs.columbia.edu

Dakshi Agrawal
IBM T.J. Watson Research
19 Skyline Drive
Hawthorne, NY 10532
agrawal@us.ibm.com

ABSTRACT

As businesses continue to grow their World Wide Web presence, it is becoming increasingly vital for them to have quantitative measures of the client perceived response times of their web services. We present Certes (CliEnt Response Time Estimated by the Server), an online server-based mechanism for web servers to measure client perceived response time, as if measured at the client. Certes is based on a model of TCP that quantifies the effect that connection drops have on perceived client response time, by using three simple server-side measurements: connection drop rate, connection accept rate and connection completion rate. The mechanism does not require modifications to http servers or web pages, does not rely on probing or third party sampling, and does not require client-side modifications or scripting. Certes can be used to measure response times for any web content, not just HTML. We have implemented Certes and compared its response time measurements with those obtained with detailed client instrumentation. Our results demonstrate that Certes provides accurate server-based measurements of client response times in HTTP 1.0/1.1 [14] environments, even with rapidly changing workloads. Certes runs online in constant time with very low overhead. It can be used at web sites and server farms to verify compliance with service level objectives.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Measurements, Models, Operational analysis*

General Terms

Algorithms, Management, Measurement, Performance, Experimentation.

Keywords

Web server, client perceived response time.

1. INTRODUCTION

The focus of web server performance is shifting from throughput and utilization benchmarks [22, 4, 24] to guaranteeing delay bounds for different classes of clients [20, 34, 17, 28, 12, 2, 27, 9, 5]. Providers of web services are faced with the challenge of providing differentiated services that guarantee bounds on client perceived response times while at the same time maximizing throughput. In order for a web site to guarantee delay bounds for its clients, it should be able to determine, in real-time, the client perceived response time. This information can then be used to verify compliance with service level objectives and to identify potential problems that may exist on the server or in the network. Unfortunately, the problem of obtaining an accurate measure of client response time remains a key factor preventing delay bounded web services from being realized.

Several approaches have been proposed for determining the response time that is experienced by a client. One approach being taken by a number of companies [18, 21, 13, 33] is to periodically measure response times obtained at a geographically distributed set of monitors. This approach can at best provide an approximation of the response time perceived by actual clients. The information gathered is generally not available at the web server in real-time, limiting the ability of a web server to respond to changes in response time to meet delay bound guarantees. Service providers are also known to place servers near the monitors used by these companies to artificially improve these performance measurements [11].

A second approach is to instrument existing web pages with client-side scripting in order to gather client response time statistics [29]. A 'post-connection' approach as this does not account for time due to failed connection attempts or waiting in kernel queues. It also does not work for non-HTML files that cannot be instrumented, such as PDF and Postscript files. It may also not work for older browsers or browsers with scripting capabilities disabled. Instrumenting the client web browser itself to gather response time statistics could avoid some of these limitations, but would require changing and upgrading all client browsers. Client browser measurements cannot accurately decompose the response time into server and network components and therefore provide no insight into whether server or network providers would be responsible for problems.

A third approach is based on having the web server application track when requests arrive and complete service [19, 20, 17, 2]. This approach has the desirable property that it only requires information available at the web server. However, server latency measures at the application level do not properly include network interactions and provide no information on network problems that might occur and affect client perceived response time. They also do not account for overheads associated with the TCP protocol underlying HTTP, including the time due to failed TCP connection attempts or waiting in kernel queues. These times can be significant, especially for servers which discard connection attempts to avoid overloading the server [34] or limiting input queue lengths of an application server [12] in order to provide a bound on the time spent in the application layer.

A fourth approach reconstructs the client response time from network packet traces. This can be done either offline, by analyzing packet trace logs [31], or the analysis can be performed online as the network packets are passively captured from the communication line [23]. Scalability can be a drawback with the online approach since the packet capturing and analysis may not be able to keep pace with the high traffic rate entering and leaving a busy server farm, requiring a number of monitoring machines. The cost of buying and managing monitoring machines may be prohibitive.

We have created Certes (CliEnt Response Time Estimated by the Server), an online mechanism that accurately estimates client perceived response time using only information available at the web server. Certes combines a model of TCP retransmission and exponential back-off mechanisms with three simple server-side measurements: connection drop rate, connection accept rate, and connection completion rate. The model and measurements are used to quantify the time due to failed connection attempts and determine their effect on perceived client response time. Certes then measures both time spent waiting in kernel queues as well as time to retrieve requested web data. It achieves this by going beyond application-level measurements to using a kernel-level measure of the time from the very beginning of a successful connection until it is completed. Our approach does not require probing or third party sampling, and does not require modification of web pages, http servers, or client-side modifications. Certes uses a model that is inherently able to decompose response time into various server and network components to help determine whether server or network providers are responsible for performance problems. Certes can be used to measure response times for any web content, not just HTML.

We have implemented Certes and verified its response time measurements against those obtained via detailed client-side instrumentation. Our results demonstrate that Certes provides accurate server-based measurements of client response times in HTTP 1.0/1.1 environments, even with rapidly changing workloads. Our results show that Certes is particularly useful under overloaded server conditions when web server application-level and kernel-level measurements can be grossly inaccurate. We further demonstrate the need for Certes measurement accuracy in web server control mechanisms that manipulate inbound kernel queue limits to achieve response time goals.

This paper is outlined as follows. Section 2 provides some necessary background on HTTP and TCP protocols and presents an overview of the Certes model. A mathematical construction of the Certes model follows, focusing on how it accounts for the time due to failed connection attempts. Section 3 presents experimental results demonstrating the effectiveness of Certes in estimating client response time at the server with various dynamic workloads for both HTTP 1.0/1.1. Section 4 discusses related work. Finally, we present some concluding remarks.

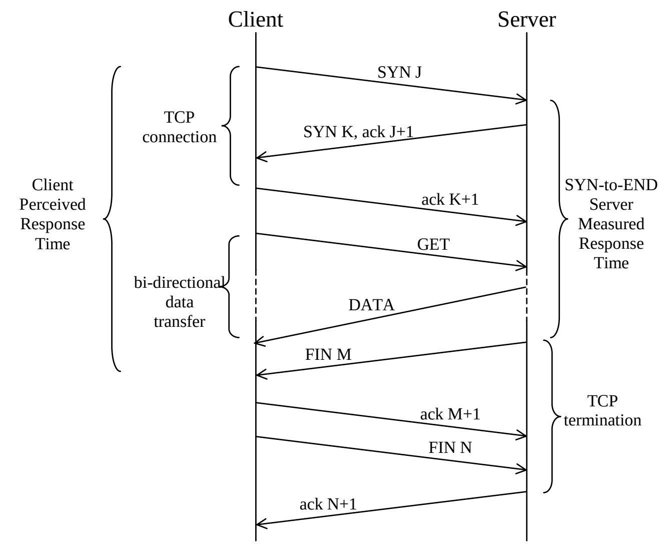
Client
Server
SYN J
SYN K, ack J+1
ack K+1
GET
DATA
FIN M
ack M+1
FIN N
ack N+1
TCP connection
bi-directional data transfer
Client Perceived Response Time
SYN-to-END Server Measured Response Time
TCP termination

Figure 1: Typical TCP Client-Server interaction.

2. CLIENT RESPONSE TIME ESTIMATED BY THE SERVER (CERTES)

2.1 Overview

We present an overview of the Certes model in the context of the HTTP and TCP protocols used for web services. A more detailed mathematical construction of the model is presented in Section 2.2.

The focus of our work is quantifying the impact of failed connection attempts on client perceived response time. Due to space constraints, we limit our discussion to an estimate of response time based on the duration of a TCP connection. For HTTP 1.0 where each HTTP request uses a separate TCP connection, this estimate corresponds to measuring the response time of individual HTTP requests. However, for HTTP 1.1 where multiple HTTP requests may be served over a single connection, this estimate may include the time for multiple requests. Since a given web page may require multiple HTTP requests in order to be displayed, determining the response time for downloading a web page may require correlating the response times of multiple HTTP requests. Although important, these issues are orthogonal to the focus of this paper and beyond the scope of our discussion. For further discussion of these issues see [15].

In the absence of packet loss, the typical client-server interaction is shown in Figure 1. Before a client can send an HTTP request to the web server, a TCP connection must first be established, via the TCP three-way handshake mechanism [7, 2]. First, the client sends a SYN packet to the server. Second, the server acknowledges the client request for connection by sending a SYN-ACK back to the client. Third, the client responds by sending an ACK to the server, completing the process of establishing a connection. Once the TCP connection is established, a series of HTTP requests are sent to the web server to request data. During the data transfer phase, one or more web pages can be transferred on the same TCP connection to the client, depending on whether HTTP 1.0 or 1.1 is used. When the data transfer is completed, the connection is terminated. To terminate the connection, the server first sends a FIN packet to the client. The client responds by sending an ACK and its own FIN to the server. Finally, the server sends an ACK back to the client and terminates the connection.

Figure 1, the common case, shows that the client perceived response time is the time from when the initial SYN is sent from the client until the time when the client receives the last byte of data from the server. The SYN-to-END time, which is the server's perception of the response time, fails to include one round trip time (RTT) which is composed of the transit time for the initial SYN J, and the transit time for the last data packet. In this case, the client perceived response time is:

$$CLIENT\_RT = \text{SYN-to-END} + RTT$$

This also holds if the server terminates the connection before sending any data by sending a FIN or RST. Likewise, if the client sends a FIN or RST to the server, then the client perceived response time is simply equal to the SYN-to-END time.

The SYN-to-END time can be decomposed into two components: the time taken to establish the TCP connection after receiving the initial SYN, and the time taken to receive and process the HTTP request(s) from the client. In certain circumstances, for example when the web server is lightly loaded and the data transfer is large, the first component of the SYN-to-END time can be ignored, and the second component can be used as an approximation to the processing time spent in the application-level server. In such cases, measuring the processing time in the application-level server can provide a good estimate of the SYN-to-END time. In general, the processing time in the application-level server is not a good estimate of the SYN-to-END time. If the web server is heavily loaded, it may delay sending the SYN-ACK back to the client, or it may delay delivering the HTTP request from the client to the application-level server. In such cases, the time to establish the TCP connection may constitute a significant component of the SYN-to-END time. Thus, to obtain an accurate measure of the SYN-to-END time, measurements must be done at the kernel level. A simple way to measurement the SYN-to-END time is to perform kernel-based timestamping of the arrival of the SYN as well as the end of transaction. If the kernel does not already provide such a packet timestamp mechanism, it can be added with minor modifications. As part of this work we modified the Linux kernel to track this information for each TCP connection.

A server-based method that can be used to determine RTT is to use the time from when the SYN-ACK is sent from the server to the time when the server receives the ACK back from the client. The RTT time measured in this way includes the time spent by the client in processing the SYN-ACK and preparing its reply. Our experience indicates that typically the time taken by clients to process a SYN-ACK packet and send a reply is not significant, and this method yields an accurate measure of RTT. This method requires the kernel at the web server to timestamp the transmission of the SYN-ACK and timestamp the arrival of the return ACK from the client. Again, these timestamps can be added to the kernel with minor modifications. Alternatively, any other approach to estimate RTT can be used [1].

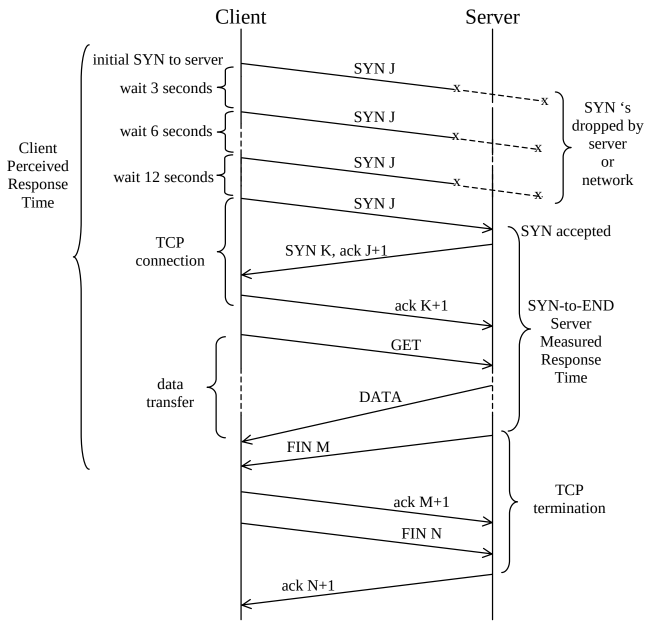

Figure 2: Effect of SYN drops on client response time.

The previous discussion assumes there are no packet losses during TCP connection establishment. Figure 2 shows a client-server interaction in the presence of SYN drops either at the server or in the network [32]. When the initial SYN is dropped, the server does not send the corresponding ACK packet. As a result, the client incurs a TCP timeout and retransmits the initial SYN to the server. Due to TCP timeout and exponential back-off mechanisms, the client may have to wait 3 seconds, 9 seconds, 21 seconds, etc., before its SYN packet is accepted by the server[6]. This wait time to initiate a TCP connection is often larger than the time required to transfer the actual web data. Dropping a SYN does not represent a denial of access in this case, but rather a delay in establishing the connection.

We refer to the time taken by the failed connection attempts as the CONN-FAIL time, which is now included in the client perceived response time:

$$CLIENT\_RT = \text{CONN-FAIL} + \text{SYN-to-END} + RTT$$

Any failure to complete the 3-way handshake after the SYN is accepted by the server is captured by the SYN-to-END time. For example, delays caused by dropped SYN-ACKs from the server to the client (the second part of the 3-way

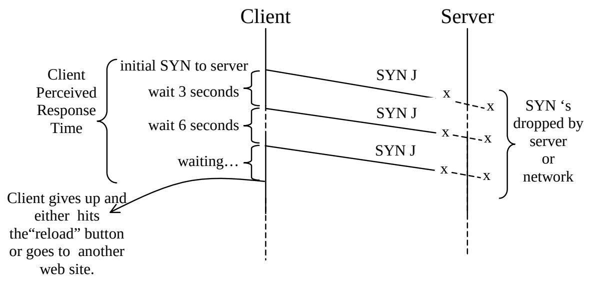
Client
Server
Client Perceived Response Time
initial SYN to server
wait 3 seconds
wait 6 seconds
waiting…
SYN J
SYN J
SYN J
x
x
x
SYN 's dropped by server or network
Client gives up and either hits the"reload" button or goes to another web site.

Figure 3: Client gets frustrated waiting for connection.

handshake) are accounted for in the SYN-to-END time.

While determining SYN-to-END is relatively straight forward, determining the CONN-FAIL time is a difficult problem and is a key focus of this work. The main problem is that when a server accepts a SYN and processes the connection, the server is unaware of how many failed connection attempts have been made by the client prior to this successful attempt. The TCP header [16] and the data payload of a SYN packet do not provide any indication of which attempt the accepted SYN represents. As a result, the server cannot examine the accepted SYN to determine whether it is an initial attempt at connecting, or a first retry at connecting, or an N^{th} retry at connecting. Even in the cases where the server is responsible for dropping the initial SYN and causing a retry, it is difficult for the server to remember the time the initial SYN was dropped and correlate it with the eventually accepted SYN for a given connection. For such a correlation, the server would be required to retain additional state for each dropped SYN at precisely the time when the server's input network queues are probably near capacity, which could result in performance scalability problems for the server.

Our solution for determining the CONN-FAIL time is based on a statistical model that estimates the number of first, second, third, etc., retries during each time interval. We determine the number of retries that occurred before a SYN is accepted using three server-side measurements: the number of SYNs dropped and accepted, and the number of connections completed. All three measurements can be obtained using simple counters at the server. This information is combined with an understanding of the TCP exponential backoff mechanism to correlate accepted SYNs with the number of SYN drops that occurred in previous time intervals. Section 2.2 describes the approach in detail.

The Certes model also includes the impact of clients canceling the connection request due to frustration while waiting to connect. This scenario is shown in Figure 3. To include the impact of cancelled requests, the Certes model includes a limit, referred to as the client frustration timeout (FTO), which is the longest amount of time a client is willing to wait for an indication of a successful connection. In other words, the FTO is a measure of the upper bound on the number of connection attempts that a client's TCP implementation will make before the client hits 'reload' on the browser or goes to another website. It is possible to use a distribution of the FTO derived from real world web browsing traffic to include in the Certes model. For simplicity, we used a constant default value of 21 seconds for the FTO in our experiments in section 3.

2.2 Mathematical Construction of The Certes Model

We present a step-by-step construction of the Certes model. We begin by defining the necessary measurements and parameters, and then, by example, expose the relationships that exist between them. We conclude this section with an equation for calculating the mean client response time from the measurements and parameters.

Certes divides time into discrete intervals for grouping connections by their temporal relationship. For ease of exposition, we will assume that time is divided into one second intervals, but in general any interval size less than the initial TCP retry timeout value of three seconds may be used. The three basic server-side measurements that are taken for each interval are:

$DROPPED_i$ the total number of SYN packets that the server dropped during the i^{th} interval.

$ACCEPTED_i$ the total number of SYN packets that the server did *not* drop during the i^{th} interval.

$COMPLETED_i$ the total number of connections that completed during the i^{th} interval.

The offered load, that is, the number of SYN packets arriving at the server, in the i^{th} interval is given by,

$$OFFERED\_LOAD_i = ACCEPTED_i + DROPPED_i$$

It follows that the mean SYN drop rate at the server for the i^{th} interval can be calculated as:

$$DR_i = DROPPED_i / OFFERED\_LOAD_i \qquad (1)$$

We decompose each of these quantities, $DROPPED_i$, $ACCEPTED_i$, $COMPLETED_i$, and $OFFERED\_LOAD_i$, as a sum of terms. Let R_i^j be the number of SYN's *that arrived* at the server as a j^{th} retry during the i^{th} interval, starting with R_i^0 as the number of *initial* attempts to connect to the server during interval i. Let k be the maximum number of retries attempted by any client (based on the FTO). For each interval i we have the following decomposition:

$$\begin{aligned} OFFERED\_LOAD_i &= \textstyle\sum_{j=0}^{k} R_i^j \\ DROPPED_i &= \textstyle\sum_{j=0}^{k} D_i^j \\ ACCEPTED_i &= \textstyle\sum_{j=0}^{k} A_i^j \\ COMPLETED_i &= \textstyle\sum_{j=0}^{k} C_i^j \end{aligned} \qquad (2)$$

where D_i^j is the number of SYN's that arrived at the server as a j^{th} retry during the i^{th} interval *but were dropped by the server*, A_i^j is the number of SYN's that arrived at the server as a j^{th} retry during the i^{th} interval and *were accepted by the server*, and C_i^j is the number of connections completed during the i^{th} interval that *were accepted by the server* as a j^{th} retry. These equations express the key measured values of the model as a sum of terms that have associations to connection attempts.

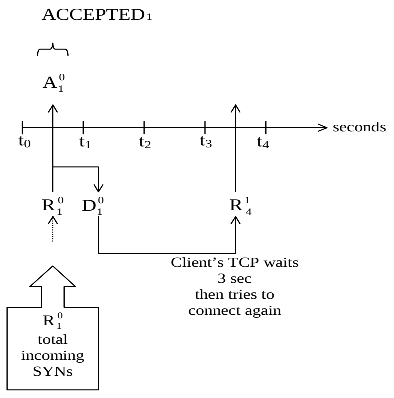

Figure 4: Initial connection attempts that get dropped become retries three seconds later.

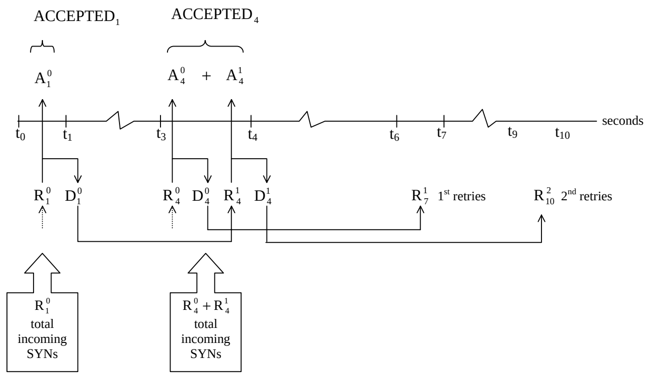

Figure 5: A second attempt at connection, that gets dropped, becomes a retry six seconds later.

It turns out the values for R_i^j, A_i^j, D_i^j, C_i^j can be inferred based on their interdependence and their relationship to the total measured values for $DROPPED_i$, $ACCEPTED_i$, $COMPLETED_i$, and $OFFERED\_LOAD_i$. As shown in Figure 4, assume that the server is booted at time t_0 (or there is a period of inactivity prior to t_0). During the first interval $[t_0,t_1]$ the server measures $ACCEPTED_1$ and $DROPPED_1$. Note that $A_1^0 = ACCEPTED_1$, $D_1^0 = DROPPED_1$, and $R_1^0 = OFFERED\_LOAD_1$, i.e. all SYN's arriving, accepted or dropped during the first interval are *initial* SYNs. The dropped SYN's, D_1^0 , will return to the server as 1^{st} retries three seconds later as R_4^1 during interval $[t_3,t_4]$.

Moving ahead in time to interval $[t_3,t_4]$, as shown in Figure 5, the server measures $ACCEPTED_4$ and $DROPPED_4$ and calculates the SYN drop rate for the 4^{th} interval, DR_4, using Equation 1. The web server cannot distinguish between an initial SYN or a 1^{st} retry, therefore, the drop rate applies to both R_4^0 and R_4^1 equally, giving $D_4^1 = DR_4 \cdot R_4^1$, and then $A_4^1 = R_4^1 - D_4^1$. From equations 2, $A_4^0 = ACCEPTED_4 - A_4^1$ and $D_4^0 = DROPPED_4 - D_4^1$. Finally, the number of initial SYN's arriving during the 4^{th} interval is $R_4^0 = A_4^0 + D_4^0$. We have determined the values for all terms in Figure 5.

Note that the D_4^1 dropped SYN's will return to the server as 2^{nd} retries six seconds later during interval $[t_9,t_{10}]$, as R_{10}^2 , when those clients experience their second TCP timeout and that the D_4^0 dropped SYN's will return to the server as 1^{st} retries, as R_7^1 , three seconds later during interval $[t_6,t_7]$.
By continuing in this manner it is possible to recursively compute all values of R_i^j, A_i^j and D_i^j for all intervals, for a given k. Figure 6 depicts the 10^{th} interval, including those intervals that *directly* contribute to the values in the 10^{th} interval. Clients that give up after k connection attempts are depicted as ending the transaction.

Figure 7 shows the final model defining the relationships between the incoming, accepted, dropped and completed connections during the i^{th} interval. Connections accepted during the i^{th} interval complete during the $(i+\text{SYN-to-END})^{th}$ interval. The client frustration timeout is specified in seconds and the term $R^j_{i+[FTO-3\times 2^{k-1}]}$ indicates that clients who do not get accepted during the i^{th} interval on the k^{th} retry will cancel their attempt for service during the $i + [FTO - 3 \times 2^{k-1}]$ interval. The Certes online model in Figure 7 can be implemented in a web server by using a simple data structure with a sliding window.

Certes is resilient to minor inconsistencies in client TCP behavior. For example, due to inconsistencies in network delays the 1^{st} retry from a client may not arrive at the server exactly three seconds later, rather it may arrive in the interval prior to or after the interval it was expected to arrive. Likewise, since the measurement for SYN-to-END is not constant, there will be instances where $C^j_{i+\text{SYN-to-END}} \neq A_i^j$; in otherwords, some of the j retries accepted in the i^{th} interval may complete prior to or after the $i+\text{SYN-to-END}^{th}$ interval. These occurrences relate to the choice for interval size. As the interval size grows smaller, the probability that the 1^{st} retry from a client will arrive in the interval which is exactly three seconds later grows smaller. We addressed these inconsistencies in our implementation by performing online adjustments to ensure that relationships within and between intervals remained consistent. More generally, one could include distributions in the model. For example, the server can calculate the distribution of the SYN-to-END, using it to determine C_i^j over a range of prior intervals. Alternatively, one can formulate a best-fit, optimization problem, and then use linear least squares to determine the 'best-fit' for all parameters, across a sliding window of intervals. As shown in Section 3.2, the results obtained by using our online adjustments were sufficiently accurate to limit the utility of using a costlier linear least squares approach.

2.2.1 Packet Loss in the Network

Packet drops that occur in the network (and not explicitly by the server) are included in the model to refine the client response time estimate. Since the client-side TCP reacts to network drops in the same manner as it does to server-side drops, network drops are estimated and added to the drop counts, D_i^j . As shown in Figure 7, SYNs dropped by the network (NDS_i^j) are combined with those dropped at the server.

To estimate the SYN drop rate in the network, one can use a general estimate of a 2-3% [36, 37] packet loss rate in the Internet or, in the case of private networks, obtain

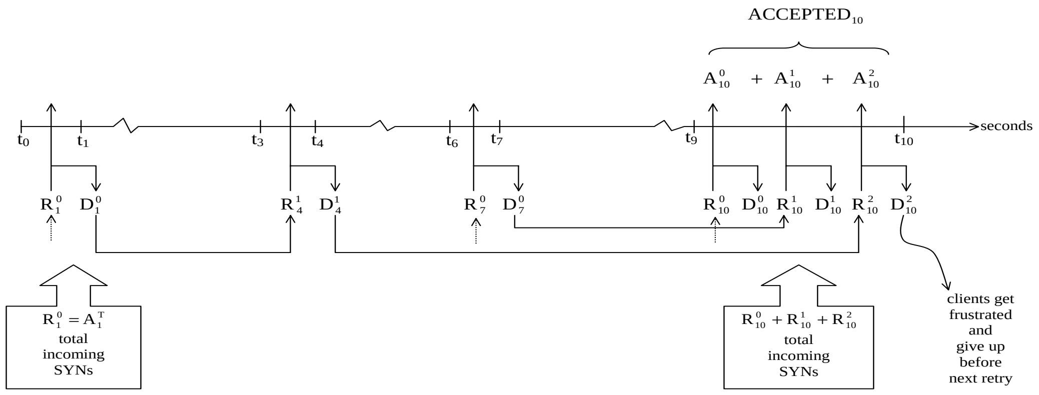
ACCEPTED$_{10}$
A^0_{10} + A^1_{10} + A^2_{10}
t_0
t_1
t_3
t_4
t_6
t_7
t_9
t_{10}
seconds
R^0_1 D^0_1
R^1_4 D^1_4
R^0_7 D^0_7
R^0_{10} D^0_{10} R^1_{10} D^1_{10} R^2_{10} D^2_{10}
$R^0_1 = A^T_1$ total incoming SYNs
$R^0_{10} + R^1_{10} + R^2_{10}$ total incoming SYNs
clients get frustrated and give up before next retry

Figure 6: After three connection attempts the client gives up.

packet loss probabilities from routers. Another approach is to assume that the packet loss rate from the client to the server is equal to the loss rate from the server to the client. The server can estimate the packet loss rate to the client from the number of TCP retransmissions.

2.3 Calculating Client Perceived Response Time

The online implementation of Certes estimates the mean client response time for the current interval, i, using the following equation:

$$CLIENT\_RT_i = \frac{\sum \text{SYN-to-END} + \sum RTT + R_i^{k+1} \cdot 3[2^{k+1} - 1] + \sum_{j=1}^{k} C_i^j \cdot 3[2^j - 1]}{COMPLETED_i + R_i^{k+1}} \quad (3)$$

Equation 3 essentially divides the sum of the response times by the number of transactions to obtain mean response time. In the denominator, R_i^{k+1} is the number of clients that gave-up during the current interval, and is added to the number of transactions that completed during the current interval. In the numerator, $\sum$ SYN-to-END is the sum of the measured SYN-to-END times and $\sum RTT$ is the sum of one round trip time (for all connections completed during the current interval). The term $R_i^{k+1} \cdot 3[2^{k+1} - 1]$ represents the amount of time that clients waited before giving-up. The term $\sum_{j=1}^{k} [C_i^j \cdot 3[2^j - 1]]$ represents the amount of time clients waited between SYN retries. A running total of the SYN-to-END and RTT measures can be kept, allowing Equation 3 to be calculated in constant time. For example, if $k = 2$, then Equation 3 resolves to

$$CLIENT\_RT_i = \frac{\sum \text{SYN-to-END} + \sum RTT + 21R_i^{k+1} + 9C_i^2 + 3C_i^1}{COMPLETED_i + R_i^{k+1}}$$

C_i^1 indicates the number of clients that waited an additional 3 seconds due to a SYN drop, C_i^2 is the number of clients that waited an additional 9 seconds due to two SYN drops, and R_i^{k+1} is the number of clients that gave-up after waiting 21 seconds.

2.4 Limitations of the Model

We identify two aspects of the model that we consider to be potential sources of estimation error and give solutions.

If the client frustration timeout is:	then the number of retries will be
less than 3 sec	k=0
at least 3sec but less than 9sec	k=1
at least 9sec but less than 21sec	k=2
at least 21sec but less than 45sec	k=3
at least 45sec but less than 1.55min	k=4
at least 1.55min but less than 3.15min	k=5

Table 1: Relationship between client frustration timeout and number of connection attempts

They are a) incorrect FTO assumptions and b) SYN floods. DNS lookup time is not included in our model nor are pages obtained from content distribution networks or distributed caches.

The first potential source of estimation error involves using an inaccurate estimate of the client frustration timeout. Table 1 specifies the relationship between FTO and the value for k, the maximum number of retries a client is willing to attempt before giving up. Fortunately, the value chosen for k covers a range of client behavior - unfortunately, that range will not cover all client behavior. For our experiments we chose a default value of $k = 2$. If the distribution for k was known (via historical measurements) the distribution can easily be included into the model. Most operating systems set k to 4, 5 or 6 and Microsoft's Internet Explorer web browser has a default FTO of five minutes.

The second potential source of estimation error arises during a SYN flood (denial of service) attack. During a SYN flood, the attackers keep the server's SYN queue full by continually sending large numbers of initial SYNs. This essentially reduces the FTO to zero. The end result is that the server stands idle, with a full SYN queue, while very few client connections are established and serviced. A SYN flood attack is very different from a period of heavy load. The perpetrators of a SYN attack do not adhere to the TCP timeout and exponential back-off mechanisms, never respond to a SYN-ACK and never establish a connection with the server; no transactions are serviced. On the other hand, in heavy load conditions, clients adhere to the TCP protocol and large numbers of transactions are serviced (excluding situations where the server enters a thrashing state).

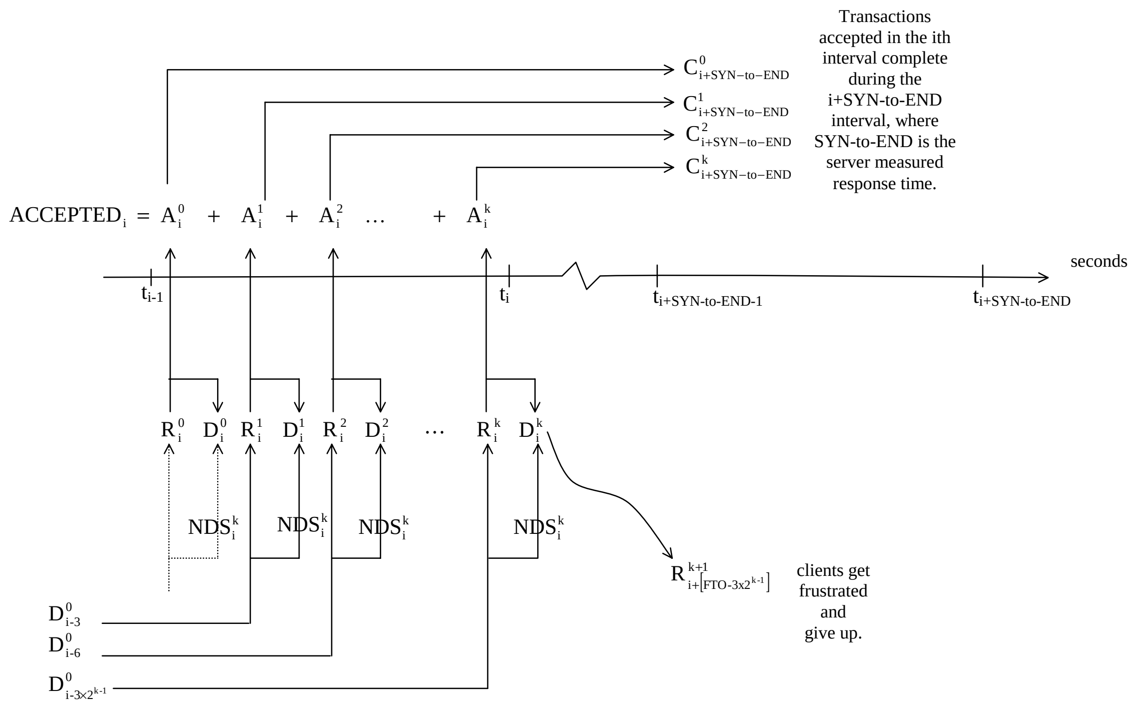
Transactions accepted in the ith interval complete during the i+SYN-to-END interval, where SYN-to-END is the server measured response time.
$C^0_{i+SYN-to-END}$
$C^1_{i+SYN-to-END}$
$C^2_{i+SYN-to-END}$
$C^k_{i+SYN-to-END}$
$ACCEPTED_i = A^0_i + A^1_i + A^2_i \dots + A^k_i$
seconds
t_{i-1}
t_i
$t_{i+SYN-to-END-1}$
$t_{i+SYN-to-END}$
R^0_i D^0_i R^1_i D^1_i R^2_i D^2_i … R^k_i D^k_i
NDS^k_i
NDS^k_i
NDS^k_i
NDS^k_i
$R^{k+1}_{i+[FTO-3x2^{k-1}]}$
clients get frustrated and give up.
D^0_{i-3}
D^0_{i-6}
$D^0_{i-3\times 2^{k-1}}$

Figure 7: Relationship between incoming, accepted, dropped, completed requests and network SYN drops.

Certes works well under heavy load conditions due to the adherence of clients to the TCP protocol. During a SYN flood attack, Certes faces the problem of identifying the distribution of the FTO. Our solution involves identifying when a SYN attack is underway, allowing Certes to switch from the FTO distribution currently in use to one that is representative of a SYN attack. While identifying a SYN attack is relatively simple, it is not easy to construct a representative FTO distribution for a SYN flood attack, which is beyond the scope of this paper.

3. EXPERIMENTAL RESULTS

To demonstrate the effectiveness of Certes, we implemented Certes on Linux and evaluated its performance in HTTP 1.0/1.1 environments, under constant and changing workloads. The results presented here focus on evaluating the accuracy of Certes for determining client perceived response time in the presence of failed connection attempts. The accuracy of Certes is quantified by comparing its estimate of client perceived response time with client-side measurements obtained through detailed client instrumentation. Section 3.1 describes the experimental design and the testbed used for the experiments. Section 3.2 presents the client perceived response time measurements obtained for various HTTP 1.0/1.1 web workloads. Section 3.3 demonstrates how a web server control mechanism can use Certes to measure its own ability to manage client response time.

3.1 Experimental Design

The testbed consisted of six machines: a web server, a WAN emulator, and four client machines. Each machine was an IBM Netfinity 4500R with dual 933 MHz CPUs, 100/10 Mbps Ethernet, 512 MB RAM, and 9.1 GB SCSI HD. Both the server and clients ran RedHat Linux 7.1 while the WAN emulator ran FreeBSD 4.4. The client machines were connected to the web server via two 10/100 Mbps Ethernet switches and a WAN emulator, used as a router between the two switches. The client-side switch was a 3Com SuperStack II 3900 and the server-side switch was a Netgear FS508. The WAN emulator software used was DummyNet [30], a flexible and commonly used FreeBSD tool. The WAN emulator simulated network environments with different network latencies, ranging from .3 to 150 ms of round-trip time, as would be experienced in LAN and cross-country WAN environments, respectively. The WAN emulator simulated networks with no packet loss and 3% packet loss, which is not uncommon over the Internet.

The web server machine ran the latest stable version of the Apache http server, V1.3.20. Apache was configured to run 255 daemons and a variety of test web pages and CGI scripts were stored on the web server. The number of test pages was small and the page sizes were 1 KB, 5 KB, 10 KB, and 15 KB. The CGI scripts would dynamically generate a set of pages of similar sizes.

Certes also executed on the server machine, independently from Apache. The Certes implementation was designed to periodically obtain counters and aggregate SYN-to-END time from the kernel and perform modeling calculations in user space. Periodically Certes would log the modelling results to disk. For our experiments, the Certes implementation was configured to use 250 ms second measurement intervals and a default frustration timeout of 21 seconds (except where noted).

Since RedHat 7.1 is not fully instrumented for Certes, minor modifications were made to the kernel. We calculated

Test	Total Number of Clients	Pages Types	Pages per Connection	Network Drop Rate	HTTP	ping RTT (ms) min/avg/max	Connections per Second	SYN Drop Rate
A	2000	static	1	0	1.0	1/8/21	1210-1670	11%-22%
B	2000	static+cgi	1	0	1.0	0.2/0.5/5	330-580	11%-33%
C	2000	static+cgi	1	0	1.0	141/152/165	320-675	0.5%-26%
D	2000	cgi	1	0	1.0	0.2/0.4/4	175-320	26%-44%
E	2000	static	15	0	1.1	4/11/17	80-150	45%-63%
F	1400	static	15	0	1.1	141/153/167	50-96	0.5%-36%
G	2000	static+cgi	5	0	1.1	0.2/0.7/6	97-173	42%-59%
H	1600	static+cgi	5	0	1.1	140/152/165	95-175	9%-37%
I+	2000	static+cgi	5	0	1.1	120/133/147	30-185	0% - 54%
J+	4800	static	1	0	1.0	142/151/165	55-470	0%-78%
K	2000	static+cgi	5	3%	1.1	0.2/0.6/6	103-177	35%-56%
L	2000	static	1	3%	1.0	0.2/0.9/8	340-1310	0%-30%
M	1600	static+cgi	5	3%	1.1	140/151/161	50-115	14%-54%
N	2000	static	1	3%	1.0	144/151/164	145-400	0.5%-34%
O*	1500	static+cgi	5	3%	1.1	140/150/161	57-147	8%- 53%
P*	1800	static+cgi	1	3%	1.0	140/151/161	180-400	5%-38%

Table 2: Test configurations included HTTP 1.0/1.1, with static and dynamic pages.

the SYN-to-END time for TCP connections by tracking the timestamp of the SYN when the device driver posts the incoming packet to the kernel and by correlating that with a timestamp at the end of the transaction. We also added ACCEPTED and COMPLETED counters, but re-used the existing SNMP/TCP counter for DROPPED. This totaled less than 50 lines of code. All these values were exposed to user space by a kernel module that extended the /proc directory.

The client machines ran an improved version of the Webstone 2.5 web traffic generator [35]. Five improvements were made to the traffic generator. First, we removed all interprocess communication (IPC) and made each child process autonomous to avoid any load associated with IPC. Second, we modified the WebStone log files to be smaller yet contain more information. Third, we extended the error handling mechanisms and modified how and when timestamps were taken to obtain more accurate client-side measurements. Fourth, we implemented a client frustration timeout mechanism after discovering the one provided in WebStone was only triggered during the select() function call and was not a true wall clock frustration timeout mechanism. Fifth, we added an option to the traffic generator that would produce a variable load on the server by switching between 'on' and 'sleep' states.

The traffic generators were used on the four client machines to impose a variety of workloads on the web server. The results for sixteen different workloads are presented, half of which were HTTP 1.0, the other half HTTP 1.1. While recent studies indicate that HTTP 1.0 is still used far more frequently than HTTP 1.1 in practice [31], HTTP 1.1 employs persistent connections, which increases the duration of each connection and reduces the number of connections, thereby reducing the effect that SYN drops have on client response time. Measuring both HTTP 1.0/1.1 workloads provides a way to quantify the benefits of using Certes for different versions of HTTP versus only using simpler SYN-to-END measurements. For the HTTP 1.1 workloads considered, the number of web objects per connection ranged from 5 to 15, consistent with recent measurements of the number of objects (i.e. banners, icons, etc) typically found in a web page [31].

The characteristics of the sixteen workloads are summarized in Table 2. The workloads accessed a varying mix of static web pages and CGI-generated web pages. All of the sixteen workloads imposed a constant load on the server except for Test I+ and Test J+, which imposed a highly-varying load on the server. Each experimental workload was run for 20 minutes. As shown in Table 2, for each workload, we measured at the server the steady-state number of connections per second and mean SYN drop rate during successive one-second time intervals. These measurements provide an indication of the load imposed on the server.

3.2 Measurements and Results

Figure 8a shows the client-side, Certes, SYN-to-END and Apache measured response times, measured in seconds, for each experiment. Figure 8b shows the same results normalized with respect to the client-side measurements.

The results show that the SYN-to-END measurement consistently underestimated the client-side measured response time, with the error ranging from 5% to more than 80%. The Apache measurements for response time, which by definition will always be less than the SYN-to-END time, were extremely inaccurate, with an error of at least 80% in all test cases. In contrast, the Certes estimate was consistently very close to the client-side measured response time, with the error being less than 2.5% in all cases except Tests L, N and P*, which were less than 7.4%.

Figures 9a and 9b, show the response time distributions for Test D using HTTP 1.0 and Test G using HTTP 1.1. These results show that Certes not only provides an accurate aggregate measure of client perceived response time, but that Certes provides an accurate measure of the distribution of client perceived response times. Figure 9 again shows how

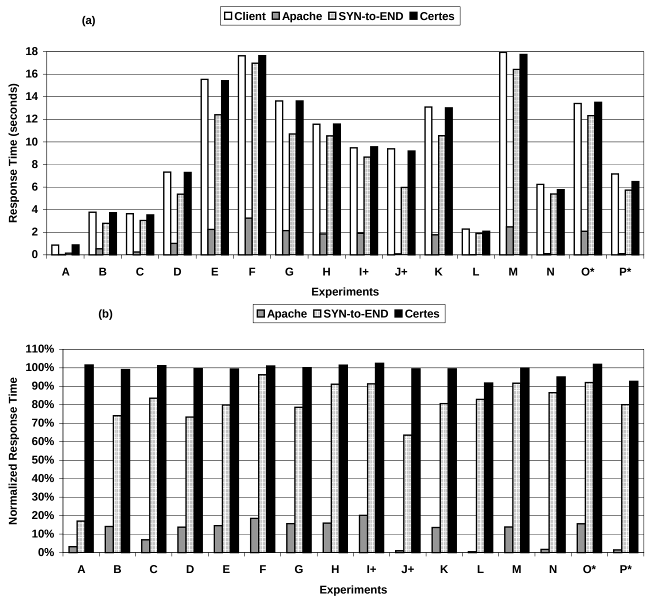

Figure 8: Certes accuracy and stability in various environments.

erroneous the SYN-to-END time measurements are in estimating client perceived response time.

Figures 10a and 10b show how the response time varies over time for Test A using HTTP 1.0 and Test G using HTTP 1.1. The figures show the mean response time at one-second time intervals as determined by each of the four measurement methods. The client-side measured response time increases at the beginning of each test run then reaches a steady state during most of the test run while the traffic generated is relatively constant. At the end of the experiment the clients are terminated, the generated traffic drops off, and the response time drops to zero.

Figure 10 shows that Certes can track in real-time the variations in client perceived response time for both HTTP 1.0/1.1 environments. The figure also indicates that Certes is effective at tracking both smaller and larger scale response times, and that Certes is able to track client perceived response time over time in addition to providing the accurate long term aggregate measures of mean response time shown in Figure 8. Again, Certes provides a far more accurate real-time measure of client perceived response time than SYN-to-END times or Apache. The large amount of overlap in the figures between the Certes response time measurements and client-side response time measurements show that the measurements are very close. In contrast, the SYN-to-END and Apache meassurements have almost no overlap with the client-side measurements and are substantially lower.

To gain insight on Certes' sensitivity to the FTO, Test O* and Test P* were executed using false assumptions for the number of retries k. In these two cases the FTO was distributed across clients: 1/3 of the transactions were from clients configured to have an FTO of 9 seconds ($k = 1$), 1/3 were from clients configured to have an FTO of 21 seconds ($k = 2$), and 1/3 from clients configured to have a

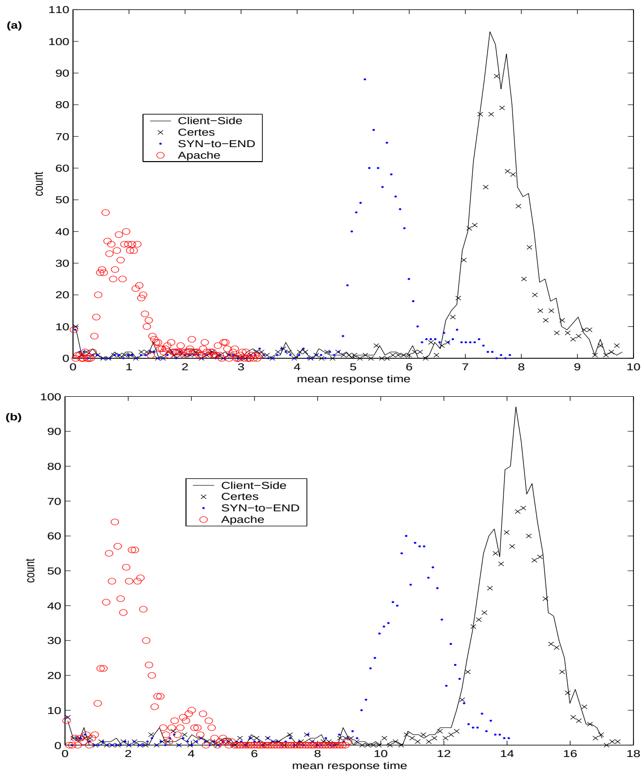

Figure 9: Certes response time distribution approximates that of the client for Tests D and G.

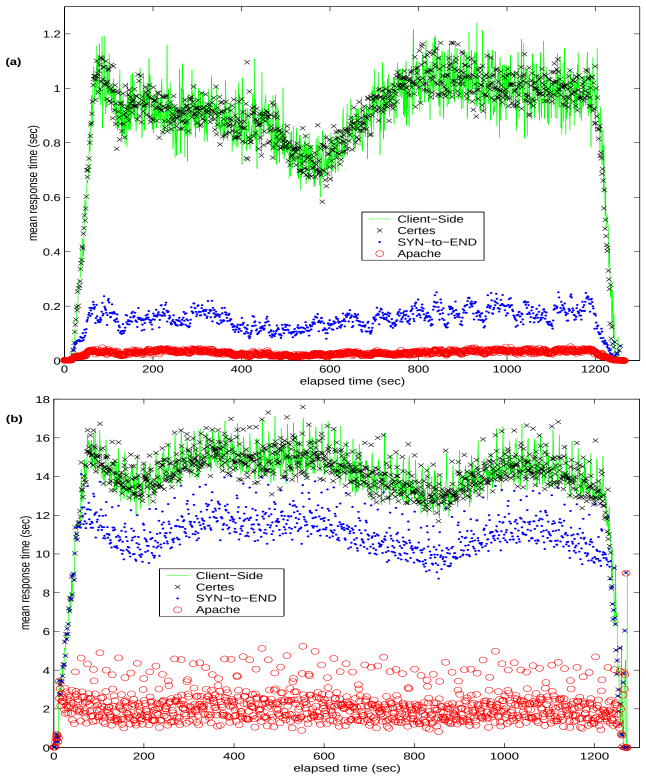

Figure 10: Certes online tracking of the client response time in Test A and Test G.

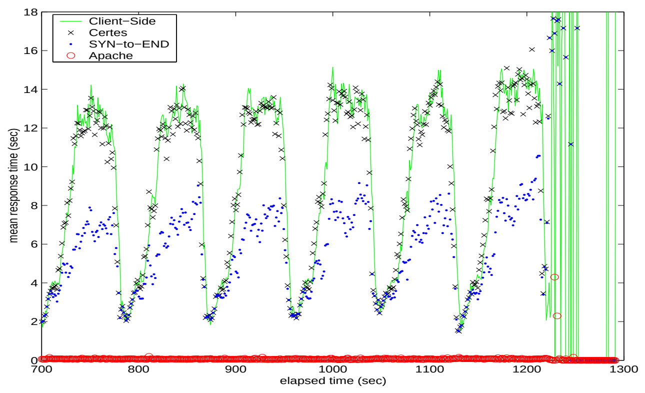

Figure 11: Certes online tracking of the client response time in Test J, in on-off mode.

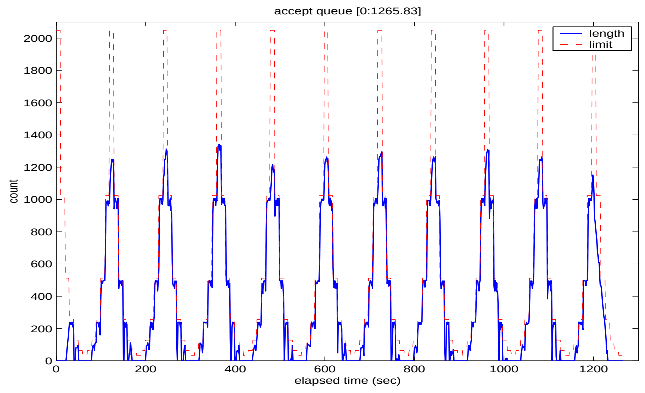

Figure 12: Web server control manipulating the Apache listen queue limit.

client FTO of 45 seconds ($k = 3$); the online model used the incorrect assumption that all clients had an FTO of 21 seconds ($k = 2$). The results for Tests O* and P* show that the Certes response time measurements were still within 2% and 7.4%, respectively, of the client-side response time measurements. For Test O* the resulting Certes estimate was only off by 108 ms and for Test P* the difference was 677 ms. As mentioned earlier, if the distribution for k was known (via historical measurements) the distribution can easily be included into the model. Further study is needed to determine if error bounds exist for Certes and under which specific conditions Certes is least accurate and why.

One of the key requirements for an online algorithm such as Certes is to be able to quickly observe rapid changes in client response time. Figure 11 shows how Certes is able to track the client response time as it rapidly changes over time. There is no significant lag in Certes reaction time to these changes. This is an important feature for any mechanism to be used in real-time control. As expected, the SYN-to-END measurement tracks the client perceived response time during the time intervals in which SYN drops do not occur. During the interval in which SYN drops occur, the SYN-to-END measurement reaches a maximum (i.e. about 6 seconds in Figure 11), which indicates the inaccuracy of the SYN-to-END time for those connections that are accepted when the listen queue is nearly full. We note for completeness that Figure 11 is zoomed in to show detail and does not contain information from the entire experiment. The 'chaos' at the end of the test run is indicative of the time-dependant nature of SYN dropping. These relatively few clients experienced SYN drops *prior* to these last few intervals, increasing the overall mean client response time during a period when the load on the system is actually very light. The mean client response time during these intervals actually reflects heavy load in the recent past.

An important consideration in using an online measurement tool such as Certes is ensuring that the measurement overhead does not adversely affect the performance of the web server. To determine the overhead of Certes, we re- executed Tests A, G and H on the server without the Certes instrumentation and found the difference in throughput and client response time to be insignificant. This suggests that Certes imposes little or no overhead on the server.

3.3 Listen Queue Management

In this section we demonstrate how Certes can be combined with a web server control mechanism to better manage client response time. Web server control mechanisms often manipulate inbound kernel queue limits as a way to achieve response time goals [19, 20, 17, 28, 2, 9, 8]. Unfortunately, there is a serious pitfall that can occur when post-TCP connection measurements are used as an estimate of the client response time. Using these types of measurements as the response time goal can lead the control mechanism to take actions that may result in having the exact opposite effect on client perceived response time from that which is intended. Without a model such as Certes, the control mechanism will be unaware of this effect.

To emulate the effects of a control mechanism at the web server, we modified the Linux kernel to dynamically change the Apache listen queue limit over time. Figure 12 shows the listen queue limit changing every 10 seconds between the values of 2^5 and 2^{11}. Figure 13 shows the effect this has on the client perceived response time. In this experiment, 1000 clients requested static pages using HTTP 1.0 while DummyNet imposed a 152 ms ping delay. The SYN drop rate varied from 0 to 81%, depending on the accept queue limit; likewise the number of completed transactions varied from 185 to 1020 per second.

When the queue limit is small, such as near the 200^{th} interval, the response time at the clients is high due to failed connection attempts, but the SYN-to-END time is small due to short queue lengths at the server. The pitfall occurs when the control mechanism decides to shorten the listen queue to reduce response time, causing SYN drops, which in turn increases mean client response time. Certainly, the control mechanism must be aware of the effect that SYN drops have on the client perceived response time and include this as an input when deciding on the proper queue limits.

4. RELATED WORK

Previous approaches in estimating client perceived response time by third party sampling [18, 21, 13, 33], client-side scripting [29], and web server application-level measurement

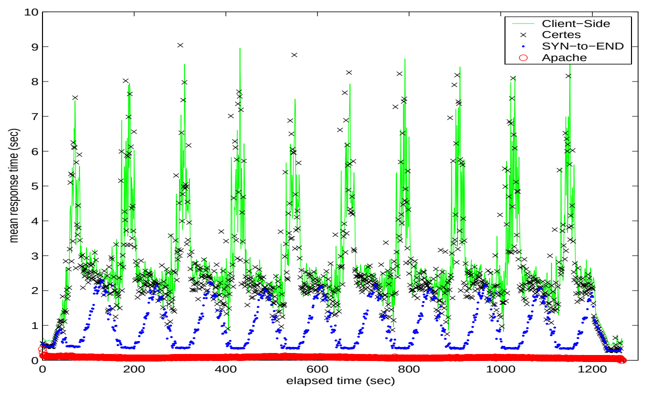

Figure 13: Client response time increases as listen queue limit decreases.

[19, 20, 17, 2] were discussed earlier. In addition, a number of analytical models have been proposed for modeling TCP behavior [26, 25, 7]. The focus of this line of research is on estimating TCP transfer throughput, and as a corollary, on estimating the client perceived response time. For example, Padhye et. al [25] derive steady state throughput of a TCP bulk transfer for a given like loss rate and round trip time. Their model is further extended by Cardwell, Savage, and Anderson [7] to include the effects of TCP three-way handshake and TCP slow start. Their extended model can accurately estimate throughput for TCP transfers of any length. Our work differs from these analytical models in several ways: First, the focus of these models is on estimating TCP transfer throughput for a given set of link conditions, while our work focuses on estimating the client perceived response times. Second, the analytical models assume a fixed packet loss rate. If the server uses SYN drops to manipulate its quality of service (QoS), then the loss rate for the three-way handshake would be different than the loss rate during the rest of the connection. Third, these models assume that the packet loss rate remains constant over the time. This paper is concerned with the case when the packet loss rate for the SYN packets may be changing frequently. Finally, these models require that the loss rate and round trip time be known a priori in order to estimate the client perceived response time, while our methods propose to estimate the same in real time.

Our work is complementary to many recent proposals for controlling QoS at web servers. One approach entails implementing kernel mechanisms that differentiate among TCP connections of different service classes during the TCP connection establishment phase. For example, Voigt, Tewari, and Mehra have proposed TCP SYN policing and prioritized listen queue to support different service classes [34]. Another approach is to dynamically manage a system resource by using a control feedback that depends on the measurements of client perceived response time [28, 8]. Such approaches can result in a high probability of TCP SYN drops, increasing the client perceived response time. Unless the effect of TCP SYN drops on the client perceived response time is measured accurately, these mechanisms would not work as desired. Our work complements these efforts to control the QoS at web servers by providing an accurate means of measuring the client perceived response time.

5. CONCLUSIONS

This paper presented Certes, an online server-based mechanism that enables web servers to measure client perceived response time. Certes is based on a model of TCP that 1) quantifies the effect that SYN drops have on client response time, 2) requires three simple server-side measurements, 3) does not require modifications to client browsers, http servers, or web pages, and does not rely on probing or third party sampling. Certes was shown to provide accurate estimates in the HTTP 1.0/1.1 environments, with both static and dynamically created pages, under constant and variable loads of differing scale.

A key result of Certes is its robustness and accuracy. Certes can be applied over long periods of time and does not drift or diverge from the perceived client response time, that any errors that may be introduced into the model do not accumulate over time. Certes is able to accurately track, online, the client perceived response time. This includes the subtle changes that can occur under constant load as well as the rapid changes that occur under bursty conditions. Certes can determine the distribution of the client perceived response time. This is extremely important, since service level objectives may not only specify mean response time targets, but also indicate variability measures such as mode, maximum, standard deviation and variance.

Certes can be readily applied in a number of contexts. Certes is particularly useful to web servers that provide QoS by controlling inbound kernel queue limits. Certes allows such servers to avoid the pitfalls associated with using application level or kernel level SYN-to-END measurements of response time. Algorithms that manage resource allocation, reservations or congestion [3] can benefit from the short-term forecasting [10] of connection retries modelled by Certes.

6. ACKNOWLEDGMENTS

This work was supported in part by an NSF CAREER Award, NSF grant ANI-0117738, and an IBM SUR Award.

7. REFERENCES

[1] M. Allman. A Web Server's View of the Transport Layer. *ACM Computer Communication Review*, 30(4):133–142, October 2000.

[2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *Workshop on Internet Server Performance Conference Proceedings*, June 1998.

[3] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. *ACM SIGCOMM Computer Communication Review*, 29(4):175–187, 1999.

[4] P. Barford and M. Crovella. A performance evaluation of hyper text transfer protocols. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):188–197, 1999.

[5] N. Bhatti and R. Friedrich. Web Server Support for Tiered Services. *IEEE Network*, 13(5):6764–71, Sept.-Oct. 1999.

[6] R. Braden. Requirements for Internet Hosts - communication layers. RFC 1122, October 1989.

[7] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP Latency. In *IEEE INFOCOMM Conference Proceedings*, volume 3, pages 1742–1751, 2000.

[8] X. Chen and P. Mohapatra. Providing Differentiated Service from an Internet Server. In *8th Int. Conf. On Computer Communications and Networks Conference Proceedings*, pages 214–217, 1999.

[9] X. Chen, P. Mohapatra, and H. Chen. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *10th International World Wide Web Conference Proceedings*, pages 545–554, 2001.

[10] E. Cohen, B. Krishnamurthy, and J. Rexford. Efficient Algorithms for Predicting Requests to Web Servers. In *IEEE INFOCOM Conference Proceedings*, pages 284–293, 1999.

[11] P. Danzig. Keynote talk presented at NOSSDAV. http://www.nossdav.org/2001/keynote_nossdav2001.ppt, 2001.

[12] L. Eggert and J. Heidemann. Application-Level Differentiated Services for Web Servers. *World Wide Web Journal*, 3(2):133–142, August 1999.

[13] Exodus. http://www.exodus.com/.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol- HTTP 1.1. RFC 2068, January 1997.

[15] Y. Fu, L. Cherkasova, W. Tang, and A. Vahdat. EtE: Passive End-to-End Internet Service Performance Monitoring. In *USENIX Conference Proceedings*, 2002.

[16] E. J. Postel. Transmission Control Protocol. RFC 793, September 1981.

[17] V. Kanodia and E. Knightly. Multi-Class Latency-Bounded Web Services. In *IEEE/IFIP IWQoS Conference Proceedings*, June 2000.

[18] KeyNote. http://www.keynote.com/.

[19] K. Li and S. Jamin. A Measurement-Based Admission-Controlled Web Server. In *IEEE INFOCOMM Conference Proceedings*, pages 651–659, 2000.

[20] C. Lu, T. Abdelzaher, J. Stankovic, and S. H. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Server. In *IEEE Real-Time Technology and Applications Symposium*, pages 51–62, June 2001.

[21] MercuryInteractive. http://www-heva.mercuryinteractive.com/.

[22] E. Nahum, T. Barzilai, and D. Kandlur. Performance issues in WWW servers. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):216–217, 1999.

[23] NetQoS. http://www.netqos.com/.

[24] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. *ACM SIGCOMM Computer Communication Review*, 27(4):155–166, 1997.

[25] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: a simple model and its empirical validation. *ACM SIGCOMM Computer Communication Review*, 28(4):303–314, 1998.

[26] J. Pahdye and S. Floyd. On inferring TCP behavior. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 287–298. ACM Press, 2001.

[27] R. Pandey, J. F. Barnes, and R. Olsson. Supporting quality of service in HTTP servers. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 247–256. ACM Press, 1998.

[28] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using Control Theory to Achieve Service Level Objectives In Performance Management. In *IFIP/IEEE International Symposium on Integrated Network Management Conference Proceedings*, pages 841–854, 2001.

[29] R. Rajamony and M. Elnozahy. Measuring Client-Perceived Response Times on the WWW. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS) Conference Proceedings*, March 2001.

[30] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[31] F. D. Smith, F. H. Campos, K. Jeffay, and D. Ott. What TCP/IP protocol headers can tell us about the web. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):245–256, 2001.

[32] W. R. Stevens. *TCP/IP Illustrated, Volume 1 The Protocols*. Addison-Wesley, Massachusetts, 1994.

[33] StreamCheck. http://www.streamcheck.com/.

[34] T. Voigt, R. Tewari, A. Mehra, and D. Freimuth. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *USENIX Conference Proceedings*, 2001.

[35] WebStone. http://www.mindcraft.com/.

[36] M. Yajnik, S. Moon, J. Kurose, and D. Towsley. Measurement and Modeling of the Temporal Dependence in Packet Loss. In *IEEE INFOCOM Conference Proceedings*, pages 345–352, 1999.

[37] Y. Zhang, V. Paxson, and S. Shenker. The Stationarity of Internet Path Properties: Routing, Loss and Throughput. In *Technical Report, ACIRI*, May 2000.