Using Certes to Infer Client Response Time at the Web Server

DAVID OLSHEFSKI
IBM T.J. Watson Research Center and Columbia University
JASON NIEH
Columbia University
and
DAKSHI AGRAWAL
IBM T.J. Watson Research Center

As businesses continue to grow their World Wide Web presence, it is becoming increasingly vital for them to have quantitative measures of the mean client perceived response times of their web services. We present Certes (CliEnt Response Time Estimated by the Server), an online server-based mechanism that allows web servers to estimate mean 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 mean client perceived 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 estimate response times for any web content, not just HTML. We have implemented Certes and compared its response time estimates with those obtained with detailed client instrumentation. Our results demonstrate that Certes provides accurate server-based estimates of mean client response times in HTTP 1.0/1.1 environments, even with rapidly changing workloads. Certes runs online in constant time with very low overhead. It can be used at websites and server farms to verify compliance with service level objectives.

Categories and Subject Descriptors: D.4.8 [**Operating Systems**]: Performance—*measurements, models, operational analysis*

Parts of this work appeared as OLSHEFSKI, D., NIEH, J., AND AGRAWAL, D. Inferring client response time at the web server. In *ACM Sigmetrics Conference Proceedings* (Marina Del Rey, Calif.). ACM, New York, 2002, pp. 160–171.
This work was supported in part by an NSF Career Award, NSF grant ANI-0117738, and an IBM SUR Award.
Authors' addresses: D. Olshefski, IBM T. J. Watson Research, 3S-F32, 19 Skyline Drive, Hawthorne, N.Y. 10532, email: olshef@us.ibm.com; J. Nieh, Columbia University, 450 Computer Science MC0401, 500 West 120th Street, New York, NY 10027; D. Agrawal, IBM T. J. Watson Research, 3S-E53, 19 Skyline Drive, Hawthorne, N.Y. 10532.

General Terms: Algorithms, Management, Measurement, Performance, Experimentation
Additional Key Words and Phrases: Web server, client perceived response time

1. INTRODUCTION

The focus of web server performance is shifting from throughput and utilization benchmarks [Nahum et al. 1999; Barford and Crovella 1999; Nielsen et al. 1997] to guaranteeing delay bounds for different classes of clients [Lu et al. 2001; Voigt et al. 2001; Kanodia and Knightly 2000; Parekh et al. 2001; Eggert and Heidemann 1999; Almeida et al. 1998; Pandey et al. 1998; Chen et al. 2001; Bhatti and Friedrich 1999]. 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 website 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.

We have created Certes (CliEnt Response Time Estimated by the Server), an online mechanism that accurately estimates mean 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 mean client perceived 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 mean 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 queuing or that perform admissions control to achieve response time goals.

This article is outlined as follows: Section 2 provides some necessary background on the components of response time discusses related work. Section 3 presents an overview of the Certes approach, the mathematical construction of the Certes model focusing on how it accounts for time attributed to failed connection attempts, and a fast online implementation of the Certes model. Section 4 describes our implementation of Certes on Linux. Section 5 presents experimental results demonstrating the effectiveness of Certes in estimating mean client response time at the server with various dynamic workloads for both HTTP 1.0/1.1. Finally, we present some concluding remarks.

2. BACKGROUND AND RELATED WORK

To understand the issues involved in measuring response time, we begin by presenting an anatomical view of the client/server behavior that occurs when a web client accesses a remote Internet website. Once a URL, such as `http://www.cnn.com/US/index.html`, is entered into a web browser, the following ten steps occur to download and display the web page:

(1) *URL parsing*. The client browser parses the URL to obtain the name of the remote host, `www.cnn.com`, from which to obtain the web page, `/US/index.html`. Web browsers maintain a cache of web pages, so if the web page is in cache and has not expired, processing can be performed locally and Steps (2)–(7) below can be skipped.

(2) *DNS lookup*. In order to contact the website (i.e., `www.cnn.com`), the browser must first obtain its IP address from DNS [Mockapetris 1987a, 1987b]. Since the browser maintains a local cache containing the IP addresses of frequently accessed websites, contacting the DNS server for this information is only performed on a cache miss, which often implies that the website is being visited for the first time.

(3) *TCP connection setup*. The client establishes a TCP connection with the remote web server. Before a client can send the HTTP request to the web server, a TCP connection must first be established, via the TCP three-way handshake mechanism [Cardwell et al. 2000; Almeida et al. 1998]. 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. Note that if the client's web browser already had an established TCP connection to the server and persistent HTTP connections are used, the browser may reuse this connection, skipping this step.

(4) *HTTP request sent*. The browser requests the web content, `/US/index.html`, from the remote site by sending an HTTP request over the established TCP connection.

(5) *HTTP request received*. When the web server machine receives an HTTP packet, the operating system determines which application should receive

the message. The HTTP request is then passed to an HTTP server application such as Apache which is typically executing in user space.

(6) *HTTP request processed*. The HTTP server application processes the request by obtaining the content either from a disk file, CGI script or other such program.

(7) *HTTP response sent*. The HTTP server application passes the content to the operating system, which in turn, sends the content to the client.

(8) *HTTP response processed*. Upon receiving the response to the HTTP request, the client processes the web content. If the content consists of an HTML page, the browser parses the HTML, identifies any embedded objects such as images, and begins rendering the web page on the display.

(9) *Embedded objects retrieved*. The browser opens additional connections to retrieve any embedded objects, allowing the browser to make multiple, simultaneous requests for the embedded objects. This parallelism helps to reduce overall latency. Depending on where the embedded objects are located, connections may be to the same server, other web servers, or content delivery networks (CDNs). If the connections are persistent and embedded objects are located on the same server, then several embedded objects will be obtained over each connection. Otherwise, a new connection will be established for each embedded object.

(10) *Rendering*. Once all the embedded objects have been obtained, the browser can fully render the web page on the display.

This ten-step process may repeat itself at any point in time, preempting any of the Steps (2) through (10). For example, the user may click on a hyperlink when the web page is not fully rendered. Such behavior causes an immediate halt to the current activity and a jump to Step (1).

A complete measure of the time to download and display a web page would account for the time spent across all ten steps. The only way to completely measure the actual client perceived response time is to measure the response time on the client machine. This requires the ability to instrument the web browser on every client, and requires that most users use the instrumented browser. Furthermore, for websites to use such information, clients would need to include mechanisms to send the measurements back to the respective websites for them to use this information to verify compliance with service-level objectives. Unfortunately, this direct browser instrumentation is not possible in practice. As a result, several pragmatic approaches have been developed to determine client response time without requiring client browser modification. These approaches must be considered as methods to estimate client perceived response time, though some may be more accurate than others. We provide an overview of these approaches and how they account for response time associated with each of the ten steps for downloading and displaying a web page. We also discuss other related work below.

One approach being taken by a number of companies [KeyNote; Mercury Interactive; Exodus; StreamCheck] is to periodically measure response times obtained by a geographically distributed set of monitors. These monitors can

be fully instrumented to provide a complete measurement of response time across all of the ten steps previously discussed, as perceived by the monitors. However, this approach suffers from five important limitations. First, no actual client transactions are being measured—only the response time for transactions generated by the monitors are reported. Second, any approach based on coarsed-grained sampling may suffer from statistical biases. Third, monitors are limited to performing transactions that do not affect other users or modify state in backend databases. For example, it would be unwise to configure a monitor to actually purchase an airline ticket or trade stock on an open exchange. Fourth, the information gathered by monitors 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. Lastly, CDN providers are known to place servers near monitors used by these companies to artificially improve their own performance measurements [Danzig 2001].

A second approach is to instrument existing web pages with client-side scripting in order to gather client response time statistics [Rajamony and Elnozahy 2001]. The approach can be used to account for actual client transactions. However, client-side scripting will always consider the start of the transaction to be sometime after the first byte of HTTP data is received by the client and the client begins processing the HTTP response. A "post-connection" approach as this does not account for any delays that occur in Steps (1) through (7), including time due to TCP connection setup or waiting in kernel queues on the web server. Client-side scripting also cannot be applied to 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. 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 to have the web server application track when requests arrive and complete service [Li and Jamin 2002; Lu et al. 2001; Kanodia and Knightly 2000; Almeida et al. 1998]. This approach has the desirable properties that it only requires information available at the web server and can be used for non-HTML content. However, this approach only measures Step (6) of the total response time. 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 TCP connection setup or waiting in kernel queues. These times can be significant, especially for servers which discard connection attempts to avoid overloading the server [Voigt et al. 2001], or for servers which limit input queue lengths of an application server [Eggert and Heidemann 1999] in order to provide a bound on the time spent in the application layer.

A fourth approach is to capture network packet traffic to the web server and use those traces to reconstruct the client response time. This can be done either offline, by analyzing packet trace logs [Smith et al. 2001], or online as the network packets are passively captured from the communication line [NetQoS].

To use packet traces to determine web page response time, EtE [Fu et al. 2002] provides an offline server-side approach for obtaining a value for client response time, based on correlating the activity across multiple connections.

This approach does account for Steps (4) to (9), but does not account for TCP connection setup time. Since EtE is a server-based approach like Certes, it also does not account for DNS lookup times and browser rendering times.

The approach also does not account for any delays that may be due to web objects that reside outside of the web server, such as objects stored on other servers or CDNs. Scalability can also be a drawback with the offline 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 packet monitoring machines. The cost of buying and managing monitoring machines may be prohibitive. In addition to the above, offline analysis fails to provide information in real-time.

In addition to the above approaches that focus on web performance, a number of analytical models have been proposed for modeling TCP behavior [Pahdye and Floyd 2001; Padhye et al. 1998; Cardwell et al. 2000]. For example, Padhye et al. [1998] derived steady state throughput of a TCP bulk transfer for a given like loss rate and round trip time. This model is further extended in Cardwell et al. [2000] to include the effects of TCP three-way handshake and TCP slow start. The extended model can accurately estimate throughput for TCP transfers of any length. These analytical models focus on estimating TCP transfer throughput instead of estimating client perceived response time. They also assume a fixed packet loss rate that remains constant over time and is known a priori. These assumptions are often not valid in measuring web server performance. For example, SYN packet loss rates may change frequently due to server load or if a web server uses SYN drops to manipulate its quality of service (QoS). Appendix 6 discusses further some of the queuing modeling issues.

Many recent approaches have proposed methods 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 et al. [2001] proposed TCP SYN policing and prioritized accept queues to support different service classes. Another approach is to dynamically manage a system resource by using a control feedback that depends on the measurements of client perceived response time [Parekh et al. 2001; Chen and Mohapatra 1999]. Such approaches can result in a high probability of TCP SYN drops, resulting in failed connection attempts that increase the client perceived response time. For these QoS mechanisms to work as desired, it is important that the effect of TCP SYN drops on the client perceived response time is measured accurately by accounting for the time in Step (3) above. Unfortunately, previous response time measurement methods, including client-side scripting, web server application-level measurements, and using packet traces, do not accurately account for the time due to TCP connection setup in the presence of failed connection attempts.

If failed connection attempts did not contribute significantly to client perceived response time, omitting the time due to TCP connection setup would not be an issue. However, Figures 1 and 2 illustrate that failed connection

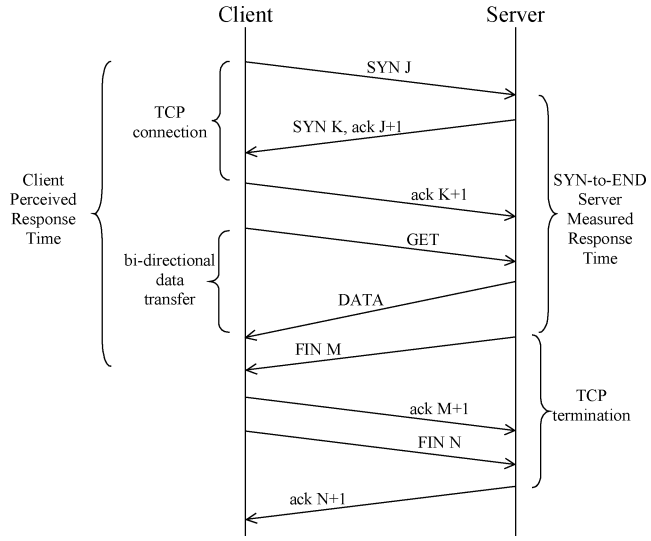

Fig. 1. Typical TCP client–server interaction.

attempts can contribute to substantial increases in response time. The figures show a TCP-oriented view of the process of downloading and displaying a web page with a focus on TCP connection establishment. URL parsing and web page rendering do not usually account for a significant portion of response time and DNS lookups are often cached to reduce their impact on response time. As a result, Figures 1 and 2 only illustrate Steps (3) to (9) with a single client and server to simplify our discussion.

Figure 1 shows the more common case in which there is no packet loss. Here, the TCP connection is established via the TCP three-way handshake mechanism, then a series of HTTP requests are sent to the web server to request data. In this scenario, the time due to TCP connection setup for transmitting and processing the initial SYN, SYN/ACK, and ACK is likely to be small compared to the time for processing the HTTP requests and may not contribute much to the overall client perceived response time. Figure 2 shows the same client-server interaction in the presence of SYN drops at the server due to server overload or admissions control [Stevens 1994]. 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 [Braden 1989]. This wait time to initiate a TCP connection is often larger than the time required to transfer the actual web data and will be a dominant factor in the overall client perceived response time. Dropping a SYN does not represent a denial of access in this case, but rather a delay in establishing the connection. The latency associated with this behavior needs to be quantified.

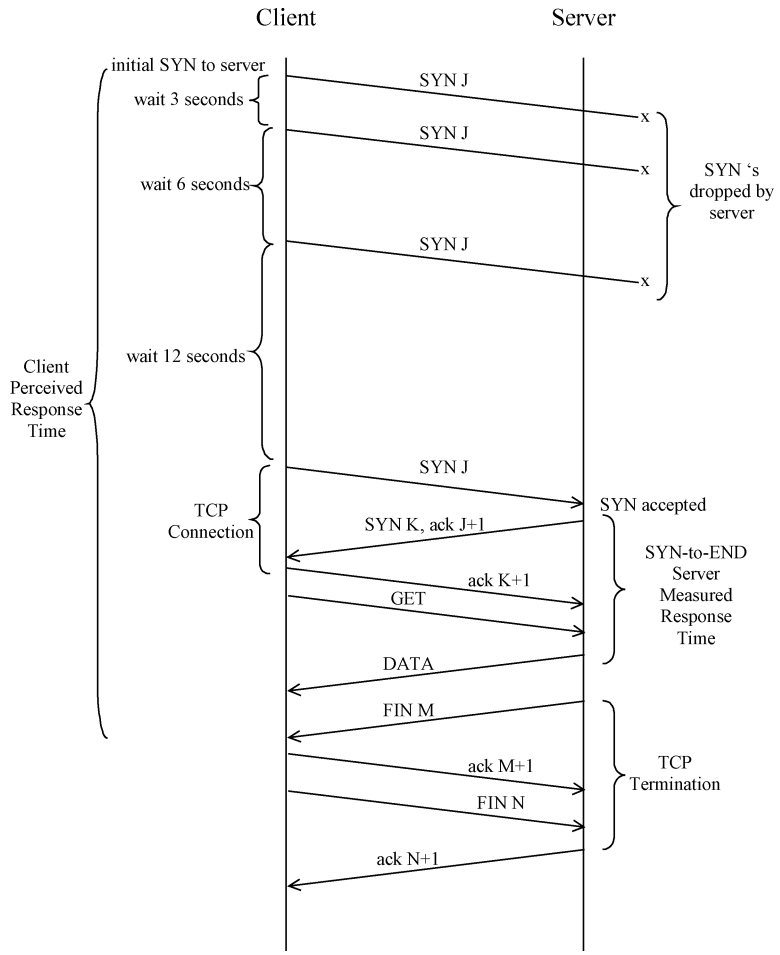

Fig. 2. Effect of SYN drops on client response time.

3. THE CERTES MODEL

The main contribution of Certes is to provide a server-side measure of mean client perceived response time that includes the impact of failed TCP connection attempts on web server performance. To simplify our discussion and focus on the issue of failed TCP connection attempts, we make the following assumptions:

(1) We focus on measuring the response time due to TCP connection setup through retrieving embedded objects, Steps (3) through (9) in Section 2. We do not consider Steps (1), (2), and (10). We assume that URL parsing and web page rendering times are small and DNS lookups are generally cached to reduce their impact on response time.
(2) We focus on determining the contribution to client perceived response time due to the performance of a given web server. We do not quantify delays that may be due to web objects residing on other servers or CDNs.

(3) We limit our discussion to an estimate of response time based on the duration of a TCP connection. For nonpersistent connections where each HTTP request uses a separate TCP connection, this estimate corresponds to measuring the response time for individual HTTP requests. For persistent connections where multiple HTTP requests may be served over a single connection, this estimate will include the time for multiple requests. Since web page with embedded objects 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. Complementary work [Fu et al. 2002] on correlating connections to web pages can be used for this purpose. Although important, these issues are orthogonal to the focus of this article.

Given these assumptions, a measure of client-perceived response time should include the time starting from when the first SYN packet is sent from the client to the server until the last HTTP response data packet is received from the server by the client. For a given connection, we define CONN-FAIL as the time between when the first SYN packet is sent from the client and when the last SYN packet is sent from the client. This is the time due to failed TCP connection attempts. When there are no failed connection attempts, CONN-FAIL is zero. For a given connection, we define SYN-to-END as the time between when the server receives the last SYN packet until the time when the server sends the last data packet. This is essentially the server's perception of response time in the absence of SYN drops. The client perceived response time is equal to CONN-FAIL and SYN-to-END plus one round trip time (RTT) to account for the the time it takes to send the SYN packet from the client to the server plus the time it takes to send the last data packet from the server to the client. The client perceived response time over the connection is:

$$CLIENT\_RT = \text{CONN-FAIL} + \text{SYN-to-END} + RTT. \quad (1)$$

Determining client perceived response time then reduces to determining CONN-FAIL, SYN-to-END, and RTT. Note that any failure to complete the 3-way handshake after the SYN is accepted by the server is captured by SYN-to-END. For example, delays caused by dropped SYN/ACKs from the server to the client (the second part of the 3-way handshake) are accounted for in the SYN-to-END time (as shown in Figure 3). The equation also holds if the server terminates the connection before sending any data by sending a FIN or RST.

Determining the SYN-to-END component of the client perceived response time is relatively straightforward. 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

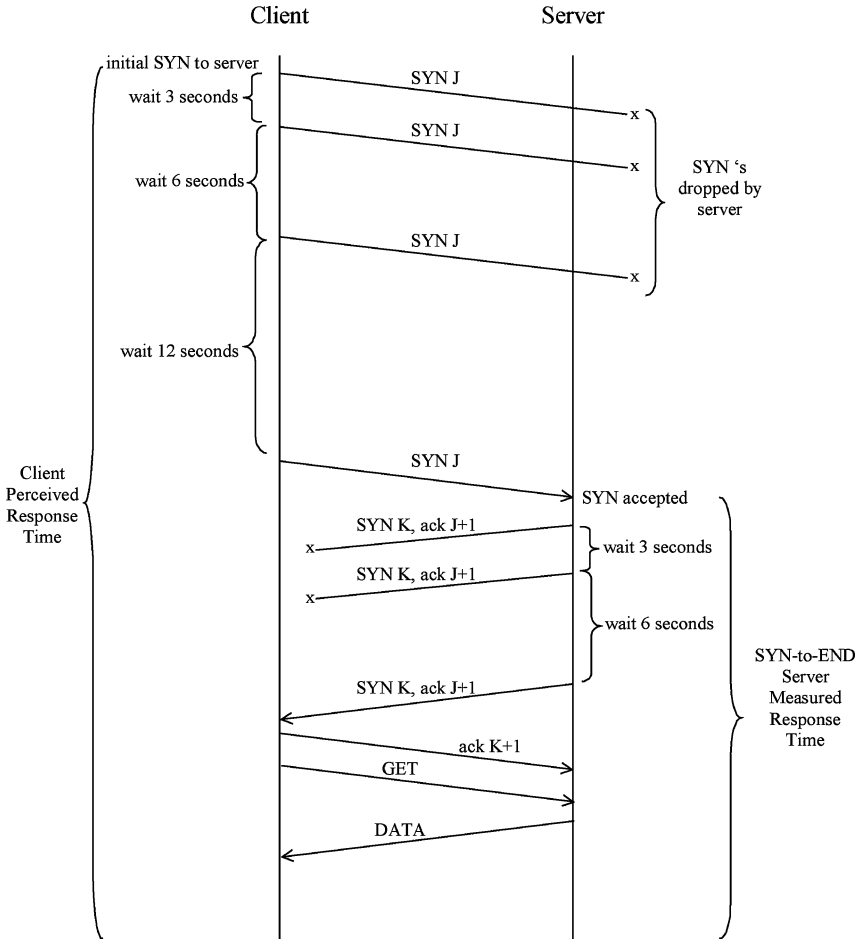

Fig. 3. Dropped SYN/ACK from server to client captured in SYN-to-END time.

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 measure SYN-to-END is by timestamping in the kernel when the last SYN packet is received by the server and when the last data packet is sent from the server. If the kernel does not already provide such a packet timestamp mechanism, it can be added with minor modifications. Section 4 describes in further detail how we measured SYN-to-END for our Certes Linux implementation.

Determining the RTT component of the client perceived response time is also relatively straightforward. RTT can be determined at the server by measuring

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. Other approaches for estimating RTT can also be used [Allman 2000]. For both SYN-to-END and RTT measurements, the kernel at the web server must provide the respective timestamps. As discussed in Section 4, these timestamps can be added with minor modifications.

However, determining CONN-FAIL is a difficult problem. The 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 [Postel 1981] 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 Nth 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.

Certes solves this problem by taking advantage of two properties of server mechanisms for supporting SYNs. First, since the server cannot distinguish between whether a SYN packet is an initial attempt or Nth retry, it must treat them all equally. Second, it is easy for a server to simply count the number of SYNs that are dropped versus accepted since it only requires a small amount of state. As a result, Certes can compute the probability that a SYN is dropped and apply that probability equally to all SYNs during a given time period to estimate the number of SYN retries that occur. This information is then combined with a understanding of the TCP exponential backoff mechanism to correlate accepted SYNs with the number of SYN drops that occurred to determine how many retries were needed before establishing a connection.

Certes can then determine CONN-FAIL based on how many retries were needed and the amount of time necessary for those retries to occur. In particular, due to TCP timeout and exponential backoff mechanisms specified in RFC 1122 [Braden 1989], the first SYN retry occurs 3 seconds after the initial SYN, the second SYN retry occurs 6 seconds after the first retry, the third SYN retry occurs 12 seconds after the second retry, etc. Certes does assume that all clients adhere to this exact exponential behavior on SYN retries from RFC 1122. This is a reasonable assumption given that RFC 1122 is supported by all major operating systems, including Microsoft operating systems [Microsoft], Linux [RedHat], FreeBSD [FreeBSD], NetBSD 1.5 [NetBSD], AIX 5.x, and Solaris. OneStat.com [OneStat 2002] estimates that 97.46% of the web server accesses

on the Internet are from users running a Windows operating system. The rest they attribute to Macintosh and Linux users (1.43% and 0.26%, respectively).

Section 3.1 presents a more detailed step-by-step construction of the Certes model. In particular, we discuss the impact of the variance of RTT on when retries arrive at the server and how Certes accounts for this variability. Section 3.2 describes a more simplified Certes model that can be implemented efficiently and yet still yields good response time results.

3.1 Mathematical Construction of The Certes Model

Certes determines the mean client perceived response time by accounting for CONN-FAIL using a statistical model that estimates the number of first, second, third, etc., retries that occur during a specified time interval. Certes divides time into discrete intervals for grouping connections by their temporal relationship. Without loss of generality, 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. For ease of exposition, let $m = 3$ be the number of discrete time intervals that occur during the initial TCP retry timeout value of three seconds.

Certes determines the number of retries that occurred before a SYN is accepted by using simple counters to take three aggregate server-side measurements for each time interval. The measurements are:

$DROPPED_i$	the total number of SYN packets that the server dropped during the ith interval.
$ACCEPTED_i$	the total number of SYN packets that the server did *not* drop during the ith interval.
$COMPLETED_i$	the total number of connections that completed during the ith interval.

Using these three measurements, we can compute for a given interval the offered load at the server, which is the number of SYN packets arriving at the server. The offered load in the ith interval is:

$$OFFERED\_LOAD_i = ACCEPTED_i + DROPPED_i. \tag{2}$$

Certes decomposes each of these measured quantities, $OFFERED\_LOAD_i$, $DROPPED_i$, $ACCEPTED_i$, and $COMPLETED_i$ as a sum of terms that have associations to connection attempts. Let R_i^j be the number of SYNs that arrived at the server as a jth retry during the ith interval, starting with R_i^0 as the number of initial attempts to connect to the server during interval i. Let D_i^j be the number of SYNs that arrived at the server as a jth retry during the ith interval but were dropped by the server. Let A_i^j be the number of SYNs that arrived at the server as a jth retry during the ith interval and were accepted by the server. Let C_i^j be the number of connections completed during the ith interval that were accepted by the server as a jth retry. Let k be the maximum number of retries attempted by any client. For each interval i, we have the

following decomposition:

$$\begin{aligned} OFFERED\_LOAD_i &= \textstyle\sum_{j=0}^{k} R_i^j \\ DROPPED_i &= \textstyle\sum_{j=0}^{k} D_i^j \\ ACCEPTED_i &= \textstyle\sum_{j=0}^{k} A_i^j \\ COMPLETED_i &= \textstyle\sum_{j=0}^{k} C_i^j . \end{aligned} \tag{3}$$

For each time interval i, Certes determines the mean client perceived response time for those web transactions that are completed during the time interval. This includes both connections that are completed during the time interval as well as connections that give up during the interval after exceeding the maximum number of retries attempted by any client. $COMPLETED_i$ is the number of transactions that completed during the ith interval and R_i^{k+1} is the number of clients that gave up during the interval. Applying Eq. (1) to a time interval, Certes computes the mean client response time for the ith interval as:

$$CLIENT\_RT_i = \frac{R_i^{k+1} \cdot 3[2^{k+1} - 1] + \sum_{j=1}^{k} C_i^j \cdot 3[2^j - 1] + \sum \text{SYN-to-END} + \sum RTT}{COMPLETED_i + R_i^{k+1}}. \tag{4}$$

Equation (4) essentially divides the sum of the response times by the number of transactions to obtain mean response time. In the denominator, Eq. (4) sums the total number of transactions that completed and clients that gave up. In the numerator, there are four terms summed together. The first term $R_i^{k+1} \cdot 3[2^{k+1} - 1]$ is the amount of time that clients waited before giving up based on the TCP exponential backoff mechanism. The second term $\sum_{j=1}^{k}[C_i^j \cdot 3[2^j - 1]]$ represents the total CONN-FAIL time experienced by those clients that completed in the ith interval. The third term $\sum$ SYN-to-END is the sum of the measured SYN-to-END times for all transactions completed in the ith interval. The fourth term $\sum RTT$ is the sum of one round trip time for all transactions completed during the ith interval. For example, if $k = 2$, then Eq. (4) reduces 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.

To compute the mean client perceived response time for each interval, Certes uses Eq. (3) to derive the values of C_i^j and R_i^{k+1} from the measured quantities $OFFERED\_LOAD_i$, $DROPPED_i$, $ACCEPTED_i$, and $COMPLETED_i$. We start from the observation that the TCP header [Postel 1981] and the data payload of a SYN packet do not provide any indication of which connection attempt a dropped SYN represents. As a result, the server's TCP implementation cannot distinguish a SYN packet containing a jth SYN retry from a SYN packet containing a kth SYN retry. This implies that all types of SYN packets are dropped

or accepted with equal probability. The mean SYN drop rate at the server for the ith interval can be computed from $OFFERED\_LOAD_i$ and $DROPPED_i$:

$$DR_i = DROPPED_i / OFFERED\_LOAD_i. \tag{5}$$

A key hypothesis of Certes is that the drop rate must therefore be equal for all R_i^j in the ith interval. This results in the following relations between R_i^j and D_i^j:

$$\begin{aligned} D_i^0 &= DR_i \cdot R_i^0 \\ D_i^1 &= DR_i \cdot R_i^1 \\ D_i^2 &= DR_i \cdot R_i^2 \\ &\vdots \\ D_i^k &= DR_i \cdot R_i^k. \end{aligned} \tag{6}$$

Each individual connection that completes during the ith interval was accepted during the (i − SYN-to-END)th interval. Because each connection may have a different SYN-to-END time, connections that complete during the ith interval may have been accepted during different intervals. Let $ACCEPTED_{p,i}$ be the number of connections that were accepted during the pth interval and completed during the ith interval. Therefore,

$$COMPLETED_i = \sum_p ACCEPTED_{p,i}. \tag{7}$$

Let

$$ACCEPTED_{p,i} = \sum_{j=0}^{k} A_{p,i}^j, \tag{8}$$

where $A_{p,i}^j$ is the number of SYNs that were accepted during the pth interval as a jth retry and completed during the ith interval. Therefore,

$$C_i^j = \sum_p A_{p,i}^j. \tag{9}$$

As mentioned above, 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. Therefore, there is no direct method for determining the number of retries associated with a specific connection. As such, there is no direct method for obtaining $A_{p,i}^j$. We estimate the value of $A_{p,i}^j$ from the ratio of A_p^j to $ACCEPTED_p$:

$$A_{p,i}^j = \left[\frac{A_p^j}{ACCEPTED_p} \right] \cdot ACCEPTED_{p,i}. \tag{10}$$

Since the SYNs that do not get dropped get accepted, Eq. (6) implies that A_i^j is:

$$A_i^j = R_i^j - D_i^j = R_i^j - \left[DR_i \cdot R_i^j \right]. \tag{11}$$

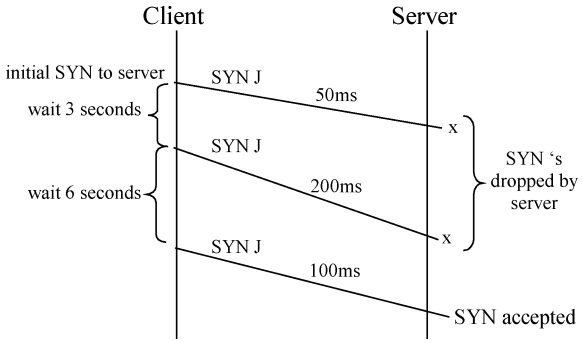

Fig. 4. Variance in RTT affects arrival time of retries.

Combining Eq. (10) and (11) allows us to rewrite Eq. (9) as:

$$C_i^j = \sum_p \left[\frac{R_p^j - \left[DR_p - R_p^j \right]}{ACCEPTED_p} \cdot ACCEPTED_{p,i} \right]. \tag{12}$$

Equation (12) solves for C_i^j in terms of R_p^j, DR_p and $ACCEPTED_{p,i}$. We can substitute Eq. (12) into our equation for calculating $CLIENT\_RT_i$, effectively removing C_i^j from Eq. (4). We now turn our attention to solving for R_i^j.

Drops occurring during the ith interval return as retries in future intervals. Based on the TCP exponential backoff mechanism, the timing of the return depends on whether it was an initial SYN, a 1st retry, a 2nd retry, etc. As a result, the number of retries arriving during the ith interval is a function of the number of drops that occurred in prior intervals:

$$\begin{aligned} R_i^1 &= D_{i-m}^0 \\ R_i^2 &= D_{i-2m}^1 \\ R_i^3 &= D_{i-4m}^2 \\ &\vdots \\ R_i^{k+1} &= D_{i-2^{k-1}m}^k. \end{aligned} \tag{13}$$

Equation (13) assumes that retries arrive at the server exactly when expected based on the TCP specification (i.e., in 3 seconds, 6 seconds, etc.). Due to variance in RTT, this assumption may not hold in practice. Such a scenario is shown in Figure 4, where the network delay changes between connection attempts for a specific client. This has the effect of skewing the estimates for R_i^j, since retries may not always arrive at the server exactly when expected (i.e., in 3 seconds, 6 seconds, etc.). Note that it is the variance in RTT for a specific client that affects the model and not the differences in RTT between clients. For example, the server will observe the 3-second, 6-second, 12-second, etc. retry delay for each client with a consistent RTT, regardless of the magnitude of the RTT.

This effect can be accounted for by treating R_i^j as a weighted distribution over the D_i^j of past intervals instead of just using a single interval. Let $W_{p,i}^j$ be

the portion of D_p^j that will return as R_i^{j+1}. The following holds:

$$1 = \sum_i W_{p,i}^j. \tag{14}$$

Using these weights, we can modify Eq. (13) so that R_i^j is a combination of drops occurring in a small set of prior intervals, rather than the number of drops that occurred in one specific prior interval:

$$\begin{aligned}
R_i^1 = \cdots + & \left[W_{i-m-1,i}^0 \cdot D_{i-m-1}^0\right] + \\
& \left[W_{i-m,i}^0 \cdot D_{i-m}^0\right] + \\
& \left[W_{i-m+1,i}^0 \cdot D_{i-m+1}^0\right] + \cdots \\
R_i^2 = \cdots + & \left[W_{i-2m-1,i}^1 \cdot D_{i-2m-1}^1\right] + \\
& \left[W_{i-2m,i}^1 \cdot D_{i-2m}^1\right] + \\
& \left[W_{i-2m+1,i}^1 \cdot D_{i-2m+1}^1\right] + \cdots \\
R_i^3 = \cdots + & \left[W_{i-4m-1,i}^2 \cdot D_{i-4m-1}^2\right] + \\
& \left[W_{i-4m,i}^2 \cdot D_{i-4m}^2\right] + \\
& \left[W_{i-4m+1,i}^2 \cdot D_{i-4m+1}^2\right] + \cdots \\
& \vdots \\
R_i^k = \cdots + & \left[W_{i-2^{k-1}m-1,i}^{k-1} \cdot D_{i-2^{k-1}m-1}^{k-1}\right] + \\
& \left[W_{i-2^{k-1}m,i}^{k-1} \cdot D_{i-2^{k-1}m}^{k-1}\right] + \\
& \left[W_{i-2^{k-1}m+1,i}^{k-1} \cdot D_{i-2^{k-1}m+1}^{k-1}\right] + \cdots.
\end{aligned} \tag{15}$$

Equation (6) allows us to rewrite Eq. (15) in terms of DR_i, $W_{p,i}^j$ and R_i^j by substituting $DR_i \cdot R_i^j$ for D_i^j:

$$\begin{aligned}
R_i^1 = \cdots + & \left[W_{i-m-1,i}^0 \cdot DR_{i-m-1} \cdot R_{i-m-1}^0\right] + \\
& \left[W_{i-m,i}^0 \cdot DR_{i-m} \cdot R_{i-m}^0\right] + \\
& \left[W_{i-m+1,i}^0 \cdot DR_{i-m+1} \cdot R_{i-m+1}^0\right] + \cdots \\
R_i^2 = \cdots + & \left[W_{i-2m-1,i}^1 \cdot DR_{i-2m-1} \cdot R_{i-2m-1}^1\right] + \\
& \left[W_{i-2m,i}^1 \cdot DR_{i-2m} \cdot R_{i-2}^1\right] + \\
& \left[W_{i-2m+1,i}^1 \cdot DR_{i-2m+1} \cdot R_{i-2m+1}^1\right] + \cdots \\
R_i^3 = \cdots + & \left[W_{i-4m-1,i}^2 \cdot DR_{i-4m-1} \cdot R_{i-4m-1}^2\right] + \\
& \left[W_{i-4m,i}^2 \cdot DR_{i-4m} \cdot R_{i-4m}^2\right] + \\
& \left[W_{i-4m+1,i}^2 \cdot DR_{i-4m+1} \cdot R_{i-4m+1}^2\right] + \cdots \\
& \vdots
\end{aligned} \tag{16}$$

$$
\begin{aligned}
R_i^k = \cdots + & \left[W_{i-2^{k-1}m-1,i}^{k-1} \cdot DR_{i-2^{k-1}m-1} \cdot R_{i-2^{k-1}m-1}^{k-1}\right] + \\
& \left[W_{i-2^{k-1}m,i}^{k-1} \cdot DR_{i-2^{k-1}m} \cdot R_{i-2^{k-1}m}^{k-1}\right] + \\
& \left[W_{i-2^{k-1}m+1,i}^{k-1} \cdot DR_{i-2^{k-1}m+1} \cdot R_{i-2^{k-1}m+1}^{k-1}\right] + \cdots .
\end{aligned}
$$

By recursive substitution of the R_i^j terms, we can transform these k equations into terms of the unknowns R_i^0 and $W_{p,i}^j$. For $k = 2$ and $m = 3$, the result is:

$$
R_i^1 = \left[W_{i-4,i}^0 \cdot DR_{i-4} \cdot R_{i-4}^0\right] + \left[W_{i-3,i}^0 \cdot DR_{i-3} \cdot R_{i-3}^0\right] + \left[W_{i-2,i}^0 \cdot DR_{i-2} \cdot R_{i-2}^0\right]
$$

$$
\begin{aligned}
R_i^2 = \; & W_{i-7,i}^1 \cdot DR_{i-7} \cdot \left[\left[W_{i-11,i-7}^0 \cdot DR_{i-11} \cdot R_{i-11}^0\right] + \right. \\
& \left[W_{i-10,i-7}^0 \cdot DR_{i-10} \cdot R_{i-10}^0\right] + \\
& \left.\left[W_{i-9,i-7}^0 \cdot DR_{i-9} \cdot R_{i-9}^0\right]\right] + \\
& W_{i-6,i}^1 \cdot DR_{i-6} \cdot \left[\left[W_{i-10,i-6}^0 \cdot DR_{i-10} \cdot R_{i-10}^0\right] + \right. \\
& \left[W_{i-9,i-6}^0 \cdot DR_{i-9} \cdot R_{i-9}^0\right] + \\
& \left.\left[W_{i-8,i-6}^0 \cdot DR_{i-8} \cdot R_{i-8}^0\right]\right] + \\
& W_{i-5,i}^1 \cdot DR_{i-5} \cdot \left[\left[W_{i-9,i-5}^0 \cdot DR_{i-9} \cdot R_{i-9}^0\right] + \right. \\
& \left[W_{i-8,i-5}^0 \cdot DR_{i-8} \cdot R_{i-8}^0\right] + \\
& \left.\left[W_{i-7,i-5}^0 \cdot DR_{i-7} \cdot R_{i-7}^0\right]\right].
\end{aligned}
\tag{17}
$$

From Eq. (3), we have:

$$
OFFERED\_LOAD_i = R_i^0 + R_i^1 + R_i^2 \tag{18}
$$

and by substituting Eq. (17) into Eq. (18), we get:

$$
\begin{aligned}
& OFFERED\_LOAD_i = \\
& R_i^0 + \\
& \left[W_{i-4,i}^0 \cdot DR_{i-4} \cdot R_{i-4}^0\right] + \left[W_{i-3,i}^0 \cdot DR_{i-3} \cdot R_{i-3}^0\right] + \left[W_{i-2,i}^0 \cdot DR_{i-2} \cdot R_{i-2}^0\right] + \\
& W_{i-7,i}^1 \cdot DR_{i-7} \cdot \left[\left[W_{i-11,i-7}^0 \cdot DR_{i-11} \cdot R_{i-11}^0\right] + \right. \\
& \qquad \left[W_{i-10,i-7}^0 \cdot DR_{i-10} \cdot R_{i-10}^0\right] + \\
& \qquad \left.\left[W_{i-9,i-7}^0 \cdot DR_{i-9} \cdot R_{i-9}^0\right]\right] + \\
& W_{i-6,i}^1 \cdot DR_{i-6} \cdot \left[\left[W_{i-10,i-6}^0 \cdot DR_{i-10} \cdot R_{i-10}^0\right] + \right. \\
& \qquad \left[W_{i-9,i-6}^0 \cdot DR_{i-9} \cdot R_{i-9}^0\right] + \\
& \qquad \left.\left[W_{i-8,i-6}^0 \cdot DR_{i-8} \cdot R_{i-8}^0\right]\right] + \\
& W_{i-5,i}^1 \cdot DR_{i-5} \cdot \left[\left[W_{i-9,i-5}^0 \cdot DR_{i-9} \cdot R_{i-9}^0\right] + \right. \\
& \qquad \left[W_{i-8,i-5}^0 \cdot DR_{i-8} \cdot R_{i-8}^0\right] + \\
& \qquad \left.\left[W_{i-7,i-5}^0 \cdot DR_{i-7} \cdot R_{i-7}^0\right]\right].
\end{aligned}
\tag{19}
$$

Equation (19) provides *one* equation for each interval i, in terms of $OFFERED\_LOAD_i$ (which is measured), DR_i (which is measured), R_i^0 (which is unknown) and $W_{p,i}^j$ (which is unknown). Once solutions for R_i^0 are found, they can be used to calculate $R_i^j, \forall i, j$. Additionally, the presence of $W_{p,i}^j$ introduces nonlinearity. Each interval i contains seven unknowns: R_i^0, $W_{i,i+2}^0$, $W_{i,i+3}^0$, $W_{i,i+4}^0$, $W_{i,i+5}^1$, $W_{i,i+6}^1$, and $W_{i,i+7}^1$.

From Eq. (14), we have the following equations for each interval i:

$$\begin{aligned} 1 &= W_{i,i+2}^0 + W_{i,i+3}^0 + W_{i,i+4}^0 \\ 1 &= W_{i,i+5}^1 + W_{i,i+6}^1 + W_{i,i+7}^1. \end{aligned} \tag{20}$$

All values in Eq. (19) must be positive, and hence we have the constraints:

$$0 \leq R_i^0, W_{p,i}^j \forall i, j, p. \tag{21}$$

Of course, if the values for $W_{p,i}^j$ were somehow magically known, then Eq. (19) could be solved directly since it reduces to a linear system of N equations in N unknowns. In practice, however, $W_{p,i}^j$ are unknown and need to be estimated. We describe one approach to a solution whose general steps are as follows:

(1) Determine an initial estimate for all $W_{p,i}^j$ over a window of prior intervals. Errors in the estimates for $W_{p,i}^j$ are directly related to the errors in R_i^0. As such, determining the bounds for this error is a known solved problem: bounding the error in solving a system of linear equations whose coefficients may contain experimental error [Golub and Loan 1996].
(2) Solve Eq. (19) using these $W_{p,i}^j$ estimated values.
(3) If there is no solution in Step (2), (i.e., Eq. (21) is not satisfied) or there is a positive change in the optimization objective, then change the values for $W_{p,i}^j$ and iterate.

Let W_I be the initial vector of $W_{p,i}^j$ estimated values. The objective of the optimization may be to minimize $\|W_I - W_S\|$, where W_S is the final solution vector of weights. In other words, assuming that the initial best estimate is based on prior fact, the solution vector ought not to deviate significantly from it.

Step (1). One approach for determining W_I to account for the impact of variance in RTT shown in Figure 4 would be to base W_I on average historical measures of the changes in RTT over time. Let χ_k be the probability density function of ΔRTT over a period of length $3[2^k]m$. Given that the arrivals of R_i^0 are uniformly distributed over the ith interval (defined by the probability

density function t_i), then

$$\begin{aligned}
E\left[W^0_{i,i+2}\right] &= \int_{i+1}^{i+2} f_{\chi_0}(x)\,dx \\
E\left[W^0_{i,i+3}\right] &= \int_{i+2}^{i+3} f_{\chi_0}(x)\,dx \\
E\left[W^0_{i,i+4}\right] &= \int_{i+3}^{i+4} f_{\chi_0}(x)\,dx \\
E\left[W^1_{i,i+5}\right] &= \int_{i+4}^{i+5} f_{\chi_1}(x)\,dx \\
E\left[W^1_{i,i+6}\right] &= \int_{i+5}^{i+6} f_{\chi_1}(x)\,dx \\
E\left[W^1_{i,i+7}\right] &= \int_{i+6}^{i+7} f_{\chi_1}(x)\,dx.
\end{aligned} \tag{22}$$

Where $f_{\chi_k}(t)$ is the convolution of t_i and χ_k:

$$\begin{aligned}
f_{\chi_0}(t) &= 3 + \int_{-\infty}^{\infty} \chi_0(x) t_i(t-x)\,dx \\
f_{\chi_1}(t) &= 9 + \int_{-\infty}^{\infty} \chi_1(x) t_i(t-x)\,dx.
\end{aligned} \tag{23}$$

In other words, $E[W^0_{i,i+2}]$ is the mean portion of R^0_i that is expected to return during the $(i+2)$nd interval as R^1_{i+2}. Note that, in Eq. (22), the $E[W^j_{p,i}]$ terms are independent of p. We now set W_I to $E[W^j_{p,i}]$, in effect, replacing $W^j_{p,i}$ in Eq. (19) with its historical mean, $E[W^j_{p,i}]$. By replacing the variables $W^j_{p,i}$ by their means, the error can be quantified using Chernoff's Bound [Papoulis and Pillai 2001].

Step (2). Substituting the current estimated values of $W^0_{p,i}$ and $W^1_{p,i}$ into Eq. (19) translates the problem into a linear system of N equations in N unknowns, for N intervals (i.e., since $W^0_{p,i}$ and $W^1_{p,i}$ are now constants, the only unknowns left are R^0_i). During system initialization, note that all SYNs arriving, accepted or dropped during the first interval are *initial* SYNs. Likewise, $R^j_i = 0$ for $1 \le j \le k$, $1 \le i \le 3$ (no 1st, 2nd, 3rd, ..., kth, retries can occur in the first three intervals) and $R^j_i = 0$ for $2 \le j \le k$, $4 \le i \le 9$ (no 2nd, 3rd, ..., kth, retries can occur during the 4th and 9th intervals). In general,

$$\begin{aligned}
&R^j_i = 0 \quad \text{for} \quad i \le 3(2^z - 1),\ j \ge z,\ 1 \le z \le k, \\
&R^j_i = 0 \quad \text{for} \quad i \le 0, \forall j.
\end{aligned} \tag{24}$$

For the initial N intervals, there are only N unknowns:

$$\begin{aligned}
OFFERED_LOAD_1 &= R_1^0 \\
OFFERED_LOAD_2 &= R_2^0 \\
OFFERED_LOAD_3 &= R_3^0 \\
OFFERED_LOAD_4 &= R_1^0 \cdot [W_{1,4}^0 \cdot DR_1] + \\
&\quad\ R_2^0 \cdot [W_{2,4}^0 \cdot DR_2] + \\
&\quad\ R_4^0 \\
OFFERED_LOAD_5 &= R_1^0 \cdot [W_{1,5}^0 \cdot DR_1] + \\
&\quad\ R_2^0 \cdot [W_{2,5}^0 \cdot DR_2] + \\
&\quad\ R_3^0 \cdot [W_{3,5}^0 \cdot DR_3] + \\
&\quad\ R_5^0
\end{aligned} \tag{25}$$

Step (3). If Step (2) does not produce a satisfactory solution, an adjustment is made to the values of $W_{p,i}^0$ and $W_{p,i}^1$. There are several ways to perform this adjustment. One method is based on the partial derivatives of R_i^0 with respect to $W_{p,i}^0$ and $W_{p,i}^1$, as defined by the gradient matrix:

$$G = \begin{bmatrix}
\frac{\partial R_i^0}{\partial W_{i-2,i}^0} & \frac{\partial R_{i-1}^0}{\partial W_{i-2,i}^0} & \frac{\partial R_{i-2}^0}{\partial W_{i-2,i}^0} & \cdots \\
\frac{\partial R_i^0}{\partial W_{i-3,i}^0} & \frac{\partial R_{i-1}^0}{\partial W_{i-3,i}^0} & \frac{\partial R_{i-2}^0}{\partial W_{i-3,i}^0} & \cdots \\
\frac{\partial R_i^0}{\partial W_{i-4,i}^0} & \frac{\partial R_{i-1}^0}{\partial W_{i-4,i}^0} & \frac{\partial R_{i-2}^0}{\partial W_{i-4,i}^0} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}. \tag{26}$$

The number of columns in G is equal to the number of intervals in the sliding window and the number of rows in G is equal to the total number of $W_{p,i}^j$ in the sliding window. Using G we can formulate a linear program to determine $W_{p,i}^j$ for the next iteration:

$$\underbrace{\begin{bmatrix}
\frac{\partial R_i^0}{\partial W_{i-2,i}^0} & \frac{\partial R_i^0}{\partial W_{i-3,i}^0} & \frac{\partial R_i^0}{\partial W_{i-4,i}^0} & \cdots \\
\frac{\partial R_{i-1}^0}{\partial W_{i-2,i}^0} & \frac{\partial R_{i-1}^0}{\partial W_{i-3,i}^0} & \frac{\partial R_{i-1}^0}{\partial W_{i-4,i}^0} & \cdots \\
\frac{\partial R_{i-2}^0}{\partial W_{i-2,i}^0} & \frac{\partial R_{i-2}^0}{\partial W_{i-3,i}^0} & \frac{\partial R_{i-2}^0}{\partial W_{i-4,i}^0} & \cdots \\
\vdots & \vdots & \vdots & \ddots
\end{bmatrix}}_{G^T}
\underbrace{\begin{bmatrix}
\Delta W_{i-2,i}^0 \\
\Delta W_{i-3,i}^0 \\
\Delta W_{i-4,i}^0 \\
\Delta W_{i-7,i}^1 \\
\Delta W_{i-6,i}^1 \\
\Delta W_{i-5,i}^1 \\
\vdots
\end{bmatrix}}_{\Delta W}
=
\underbrace{\begin{bmatrix}
\Delta R_i^0 \\
\Delta R_{i-1}^0 \\
\Delta R_{i-2}^0 \\
\Delta R_{i-3}^0 \\
\Delta R_{i-4}^0 \\
\Delta R_{i-5}^0 \\
\vdots
\end{bmatrix}}_{\Delta R^0}. \tag{27}$$

The column vector ΔW is the amount of (unknown) change to apply to the $W_{p,i}^{j}$ for the next iteration. The column vector ΔR^0 is the amount of change we would like to witness for each R_i^0 by applying the new values for $W_{p,i}^{j}$. In this case,

$$\Delta R_i^0 = \begin{cases} \|0 - R_i^0\| & \text{if} \quad R_i^0 < 0 \\ 0 & \text{otherwise.} \end{cases} \tag{28}$$

Essentially, Eq. (27) uses the gradient matrix G^T to determine how much each weight ought to be changed in order to achieve a viable solution. Equation (27) can be solved using a linear least squares method [Press et al. 1992] to obtain a best fit solution for the ΔW.

Final Step. Once Step (2) produces a satisfactory solution for R_i^0 and $W_{p,i}^{j}$, these values can be plugged into Eq. (16) to obtain the values for R_i^j. The values for R_i^j can then be used in Eq. (12) to determine C_i^j. Having determined the values for R_i^j and C_i^j for the ith interval, we use these values in Eq. (4) to obtain the mean client response time.

3.2 Fast Online Approximation of The Certes Model

Section 3.1 describes a computationally expensive algorithm: solving a system of nonlinear equations. We now present a fast, online, implementation of Certes that produces near optimal results based on a noniterative approach. We simplify the mathematical approach in two ways:

(1) We assume that all transactions that complete during the ith interval have roughly the same SYN-to-END time. If variance in SYN-to-END time leads to an inconsistency in the model, we make an online adjustment similar to Eq. (12) but based on the mean SYN-to-END time for a given interval. For the remainder of the article, when referring to SYN-to-END time, we imply the mean SYN-to-END time for a given interval.

(2) We compute an initial estimate of weights, W_I, by assuming RTT has no variance. If this assumption leads to an inconsistency in the model, we make simple online adjustments to $W_{p,i}^{j}$ in the current and future time intervals.

What follows is a step-by-step example exposing this approach.

Step (1). An alternative to the approach given in the prior section for determining W_I is to begin with the assumption that the RTT has no variance. Given an assumption of zero variance in the RTT, the initial values for W_I become:

$$\begin{aligned} 0 &= W_{i-m-1,i}^{0} = W_{i-2m-1,i}^{1} = W_{i-4m-1,i}^{2} = \cdots = W_{i-2^{k-1}m-1,i}^{k-1} \\ 1 &= W_{i-m,i}^{0} \quad = W_{i-2m,i}^{1} \quad = W_{i-4m,i}^{2} \quad = \cdots = W_{i-2^{k-1}m,i}^{k-1} \\ 0 &= W_{i-m+1,i}^{0} = W_{i-2m+1,i}^{1} = W_{i-4m+1,i}^{2} = \cdots = W_{i-2^{k-1}m+1,i}^{k-1}. \end{aligned} \tag{29}$$

If, by using this assumption, a solution cannot be found, we add-in or adjust for RTT variance by increasing or decreasing the values for $W_{p,i}^{j}$ using simple online heuristics in Step (3). These adjustments serve as an alternative to iterating over Eq. (19) to determine *optimal* values for $W_{p,i}^{j}$.

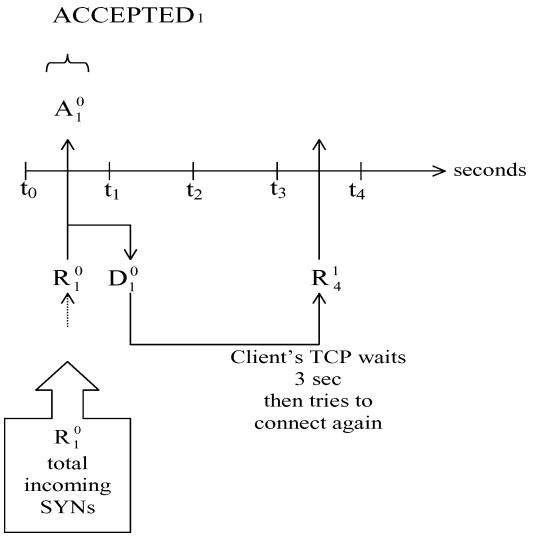

Fig. 5. Initial connection attempts that get dropped become retries three seconds later.

Step (2). The following demonstrates how to efficiently solve Eq. (19) via online direct substitution over a sliding window of intervals. Assume that the server is booted at time t_0 (or there is a period of inactivity prior to t_0), as shown in Figure 5. Certes assumes that all SYNs arriving during the first interval $[t_0, t_1]$ are initial SYNs. During the first interval $[t_0, t_1]$, the server measures $ACCEPTED_1$ and $DROPPED_1$ and can use those measurements to determine $A_1^0 = ACCEPTED_1$, $D_1^0 = DROPPED_1$, and $R_1^0 = OFFERED\_LOAD_1$. Appendix 6 shows the results when Certes is applied when SYNs in the first interval are not all initial SYNs. The dropped SYNs, D_1^0 , will return to the server as 1st 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 6, the server measures $ACCEPTED_4$ and $DROPPED_4$ and calculates the SYN drop rate for the 4th interval, DR_4, using Eq. (5). The web server cannot distinguish between an initial SYN or a 1st 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 Eq. (3), $A_4^0 = ACCEPTED_4 - A_4^1$ and $D_4^0 = DROPPED_4 - D_4^1$. Finally, the number of initial SYNs arriving during the 4th interval is $R_4^0 = A_4^0 + D_4^0$. We have determined the values for all terms in Figure 6.

Note that the D_4^1 dropped SYNs will return to the server as 2nd 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 SYNs will return to the server as 1st 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 7 depicts the 10th interval, including those intervals that *directly* contribute to the values in the 10th interval. Clients that give up after k connection attempts are depicted as ending the transaction.

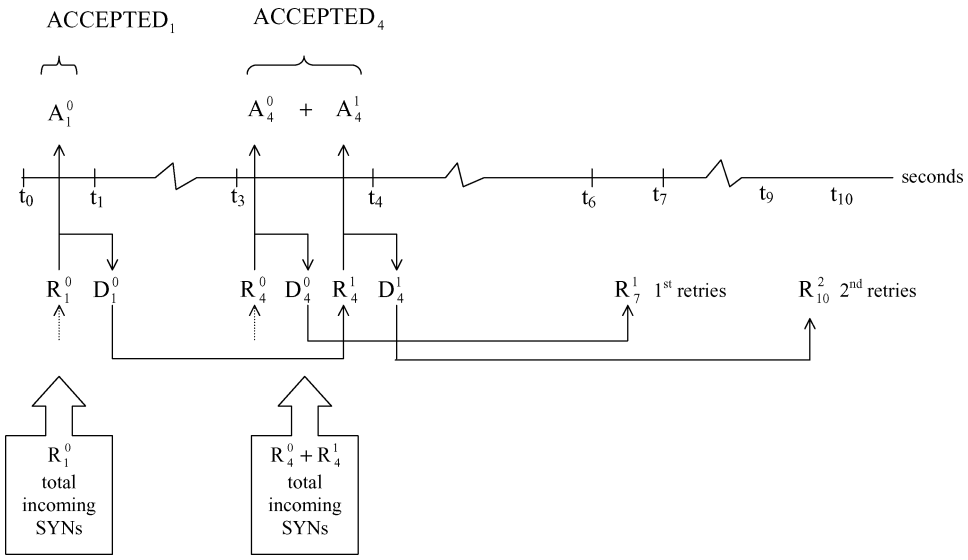

Fig. 6. A second attempt at connection, that gets dropped, becomes a retry six seconds later.

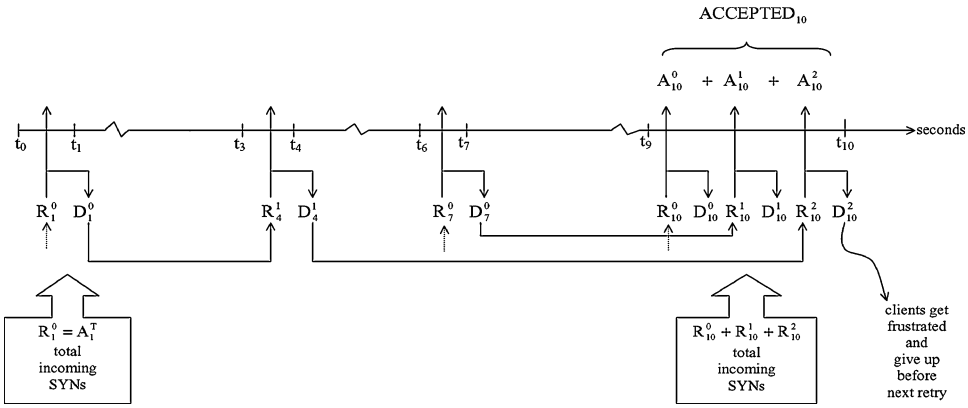

Fig. 7. After three connection attempts the client gives up.

Figure 8 shows the final model defining the relationships between the incoming, accepted, dropped and completed connections during the ith interval. Connections accepted during the ith interval complete during the $(i + \text{SYN-to-END})$th interval. The client frustration timeout is specified in seconds and the term $R^j_{i+[FTO-2^{k-1}m]}$ indicates that clients who do not get accepted during the ith interval on the kth retry will cancel their attempt for service during the $i + [FTO - 2^{k-1}m]$ interval.

The model in Figure 8 can be implemented in a web server by using a simple data structure with a sliding window. Note that during each time interval, only the aggregate counters for $DROPPED_i$, $ACCEPTED_i$, and $COMPLETED_i$ are incremented. At the end of each time interval, the more detailed counters for R^j_i, A^j_i, D^j_i, C^j_i are computed using a fixed number of computations.

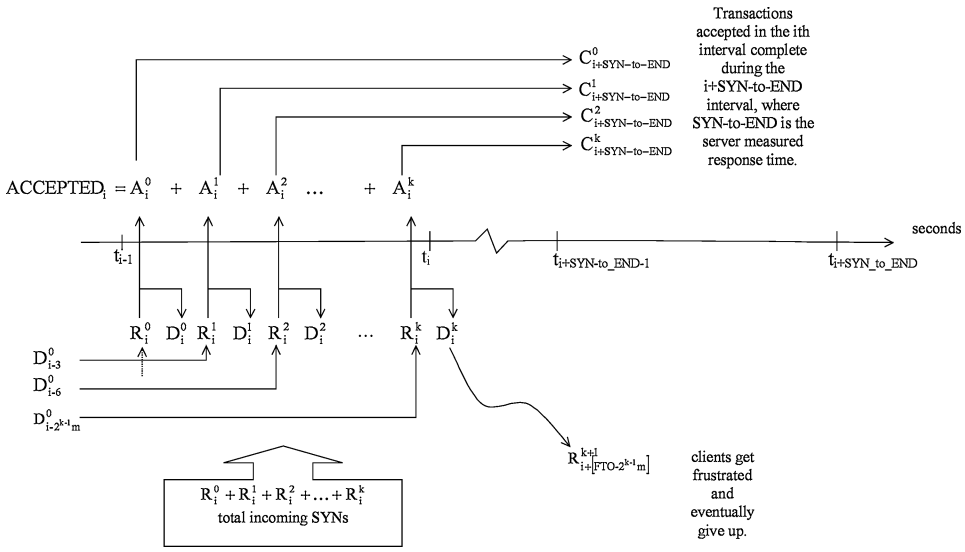

Fig. 8. Relationship between incoming, accepted, dropped, completed requests.

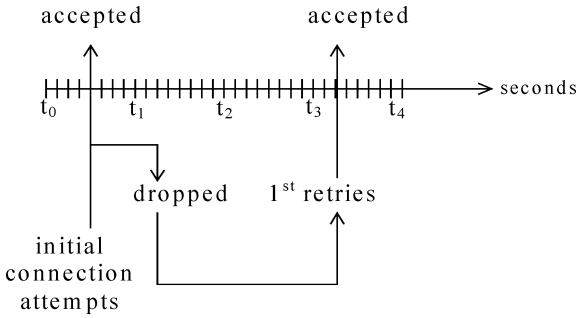

Fig. 9. The smaller the interval, the more difficult to accurately discretize events.

Step 3. As mentioned in Section 3.1, due to inconsistencies in network delays the 1st 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 $C^{j}_{i+\text{SYN-to-END}} \neq A^{j}_{i}$; in other words, some of the j retries accepted in the ith interval may complete prior to or after the $(i + \text{SYN-to-END})$th interval.

These occurrences relate to an interesting aspect of the choice for interval length. In general, when sampling techniques are used, the smaller the sampling period (more frequent the sampling), the more accurate the result. Certes is *not* a sampling based approach—yet one might intuit that using shorter intervals would somehow provide for better results—just the opposite is true. As shown in Figure 9, as the size of the interval is reduced below a certain point, the probability that events happen when expected reduces as well. For example, the probability that a dropped initial SYN will arrive back at the server during

the interval that is *exactly* three seconds later becomes zero as the size of the interval is reduced to zero. This is similar in nature to the problem of variance in RTT that is specified in Figure 4. Likewise, with small-sized intervals, the probability of events occurring on an interval boundary increases.

These inconsistencies can be accounted for by performing online adjustments to $W^j_{p,i}$ to ensure that relationships within and between intervals remain consistent. The function *rint*, round to integer, is used to ensure that certain values for the model remain integral (i.e., we do not allow a fractional number of dropped SYNs). The first heuristic we use is:

$$\begin{aligned}
&\text{if } \left(OFFERED_LOAD_i < R_i^1 + R_i^2\right) \text{ then}\\
&\quad overload = \left(R_i^1 + R_i^2\right) - OFFERED_LOAD_i\\
&\quad R_i^1 \;\; = R_i^1 \;\; - rint\left(overload \cdot \left\lfloor \frac{R_i^1}{R_i^1 + R_i^2} \right\rfloor\right)\\
&\quad R_{i+1}^1 = R_{i+1}^1 + rint\left(overload \cdot \left\lfloor \frac{R_i^1}{R_i^1 + R_i^2} \right\rfloor\right)\\
&\quad R_i^2 \;\; = R_i^2 \;\; - rint\left(overload \cdot \left\lfloor \frac{R_i^2}{R_i^1 + R_i^2} \right\rfloor\right)\\
&\quad R_{i+1}^2 = R_{i+1}^2 + rint\left(overload \cdot \left\lfloor \frac{R_i^2}{R_i^1 + R_i^2} \right\rfloor\right).
\end{aligned} \tag{30}$$

If the number of retries exceeds the measured offered load, we simply *delay* a portion of the overload until the next interval.

The second heuristic we use is:

$$\begin{aligned}
&\text{if } \left(ACCEPTED_{i-\text{SYN-to-END}} \neq COMPLETED_i\right) \text{ then}\\
&\quad C_i^0 = rint\left(COMPLETED_i \cdot \left\lfloor \frac{A^0_{i-\text{SYN-to-END}}}{ACCEPTED_{i-\text{SYN-to-END}}} \right\rfloor\right)\\
&\quad C_i^1 = rint\left(COMPLETED_i \cdot \left\lfloor \frac{A^1_{i-\text{SYN-to-END}}}{ACCEPTED_{i-\text{SYN-to-END}}} \right\rfloor\right)\\
&\quad C_i^2 = rint\left(COMPLETED_i \cdot \left\lfloor \frac{A^2_{i-\text{SYN-to-END}}}{ACCEPTED_{i-\text{SYN-to-END}}} \right\rfloor\right).
\end{aligned} \tag{31}$$

Since our approximation uses the mean SYN-to-END time, the number of completed connections may not equal the number of accepted connections. We adjust for this difference by using the ratio $A^0_{i-\text{SYN-to-END}} : A^1_{i-\text{SYN-to-END}} : A^2_{i-\text{SYN-to-END}}$ to calculate the ratio for $C_i^0 : C_i^1 : C_i^2$. This attempts to adjust for variance in the SYN-to-END time.

As shown in Section 5.2, the results obtained by using these heuristics were sufficiently accurate to allow us to bypass the use of the costlier optimization approach defined in Section 3.1.

Final Step. Having determined the values for R_i^j and C_i^j for the ith interval, we use these values in Eq. (4) to obtain the mean client response time.

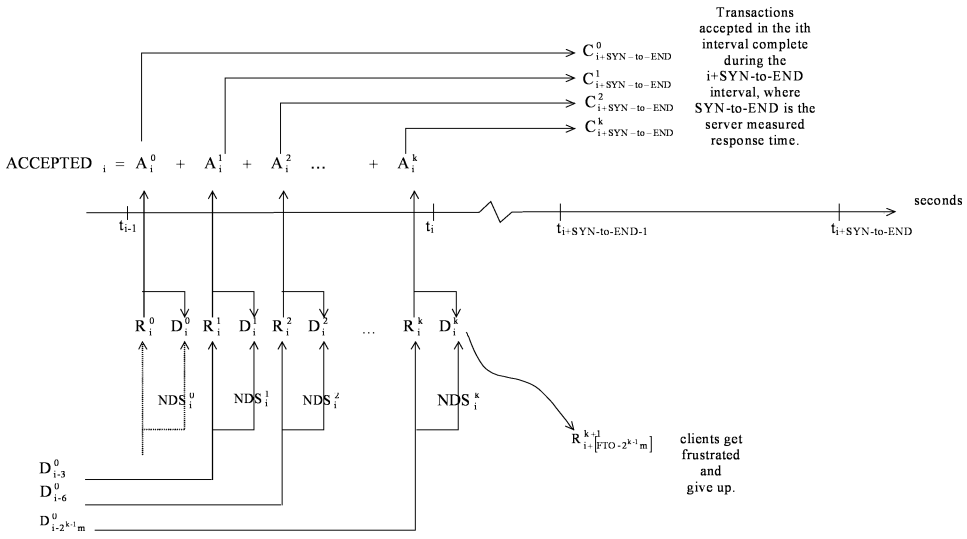

Fig. 10. Addition of network SYN drops to the model.

3.2.1 *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 10, 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% [Yajnik et al. 1999; Zhang et al. 2000] packet loss rate in the Internet or, in the case of private networks, obtain 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.

3.2.2 *Client Frustration Time Out (FTO).* A scenario that is very often neglected when calculating response times occurs when the client cancels the connection request due to frustration while waiting to connect. This is shown in Figure 11. Any client is only willing to wait a certain amount of time before hitting reload on the browser or going to another site. Such failed transactions must be included when determining client response time. To include this latency, the Certes model defines 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.

Table I specifies the relationship between FTO and the value for k, which was introduced in Section 3.1 as the maximum number of retries a client is willing to attempt before giving up. Fortunately, the value chosen for the number of

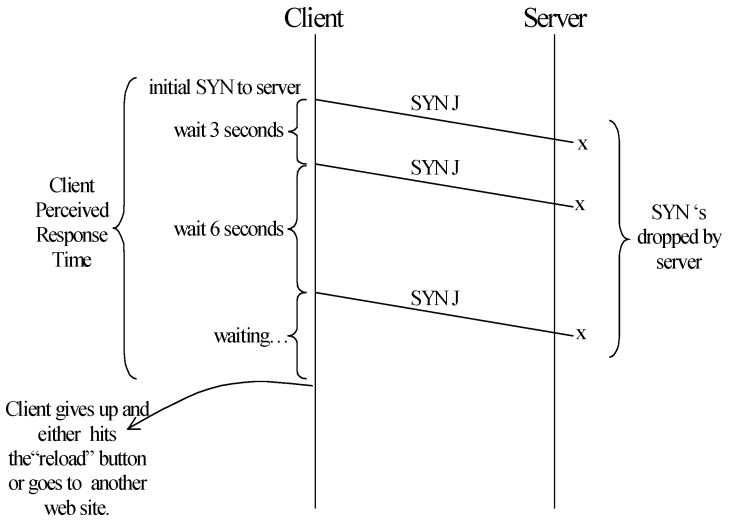

Fig. 11. Client gets frustrated waiting for connection.

Table I. Relationship between Client Frustration Timeout and Number of Connection Attempts

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

retries covers a range of client behavior—unfortunately, that range will not cover all client behavior. Although it is possible to use a distribution of the FTO derived from real-world web-browsing traffic, for simplicity, we used a constant default value of 21 seconds ($k = 2$) in most of our experiments. In Section 5, we look more carefully at the impact of using an incorrect assumption for the FTO.

3.2.3 *SYN Flood Attacks.* Another scenario that is very often neglected when calculating response times 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).

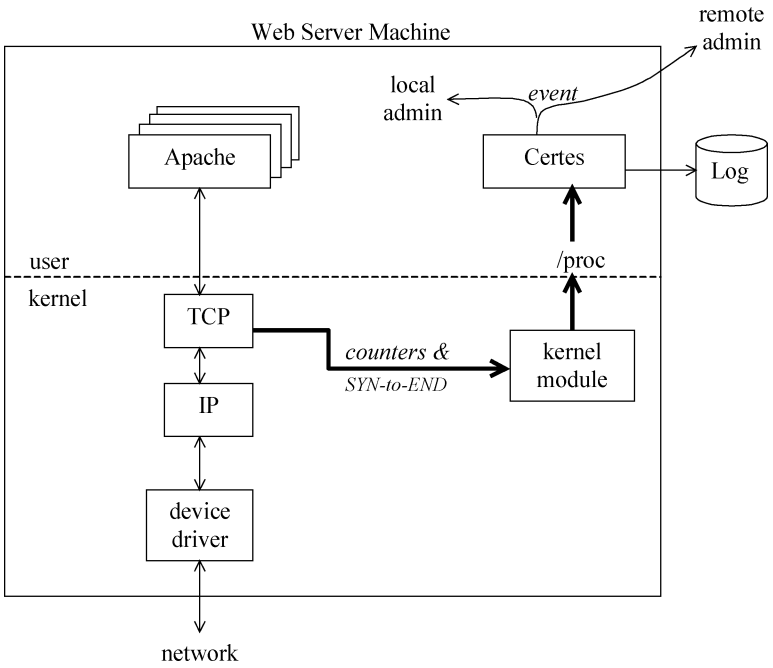

Fig. 12. Certes implementation on a Linux web server.

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 approach to a 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 simple to construct a representative FTO distribution for a SYN flood attack. Implementing this approach is beyond the scope of this paper and left for future work.

3.2.4 *Categorization.* Certes can be used, in parallel, to obtain response time estimates for multiple classes of transactions. Since Certes is based on the drop activity associated with SYN packets, the classification of a dropped SYN is limited to the information contained in the SYN packet which includes, the device the packet arrived on, source IP address and port, and destination IP address and port. NAT [Srisuresh and Holdredge 1999; Srisuresh and Egevang 2001] and IP aliasing can be used to reduce or eliminate some of these restrictions, but requires additional configuration on the part of the server's administrator.

4. CERTES LINUX IMPLEMENTATION

We have implemented the fast online Certes model from Section 3.2 in RedHat Linux 7.1. Figure 12 shows that the Certes implementation was designed to execute on the same machine as the web server application. Apache is shown

as the web server application in Figure 12, but any web server application can be used since Certes runs completely independently.

Certes was built with the expectation that it would be part of a control loop. As such, a local or remote administrator (or control module) can subscribe to Certes to receive notification when response time thresholds are exceeded. Certes also periodically logs its modeling results to disk to provide a history of web server performance that can be used for additional performance analysis.

Certes was mostly implemented as a user-space application that obtains kernel measurements at the end of each time interval, and then uses the measurements to perform modeling calculations in user space. This time interval is the same one introduced in Section 3.1, and can be set to any value less than the initial TCP retry timeout value of 3 seconds. The kernel measurements required by Certes are the total number of accepted, dropped, and completed connections, and the total SYN-to-END time for all completed connections during an interval. We implemented these as global running counters within the kernel. These variables are monotonically increasing from the time at which the machine is booted, regardless of whether or not the Certes model is executing in user space.

If the kernel already provides these four measurements, then Certes can be implemented without any kernel modifications. However, since RedHat 7.1 is not fully instrumented for Certes, minor modifications were made to the kernel. These modifications totaled less than 50 lines of code. To expose the ACCEPTED, COMPLETED and DROPPED counters, and the SYN-to-END measurement to user space, we wrote a simple kernel module that extended the `/proc` directory. User space programs can then obtain the kernel values by simply reading a file in the `/proc` directory. This is the de facto method in Linux for obtaining kernel values from user space.

To provide further details on our kernel modifications, we describe the steps by which the Linux kernel manages TCP connection establishment and termination. We then discuss our instrumentation code that obtains the ACCEPTED, COMPLETED and DROPPED counters, and the SYN-to-END measurement.

Figure 13 shows the structure of the TCP/IP connection establishment implementation in Linux. The three important data structures are the rx queue, the SYN queue, and the accept queue. The rx queue contains incoming packets that have just arrived. The SYN queue, which is actually implemented as a hash table, contains those connections which have yet to complete the TCP three-way handshake. The accept queue contains those connections which have completed the three-way handshake but have not yet been accepted by the Apache web server application. The accept queue is often referred to as the listen queue since the socket it is attached to is a listening socket. Figure 13 is numbered according to the following steps that occur during TCP connection establishment in an unmodified Linux kernel:

(1) A SYN arrives and is timestamped (denoted *ts* in the figure).
(2) The incoming SYN packet is placed onto the rx queue during the hardware interrupt. The rx queue does have a limit, but this limit can be changed and rarely do packets get dropped due to the rx queue being full.

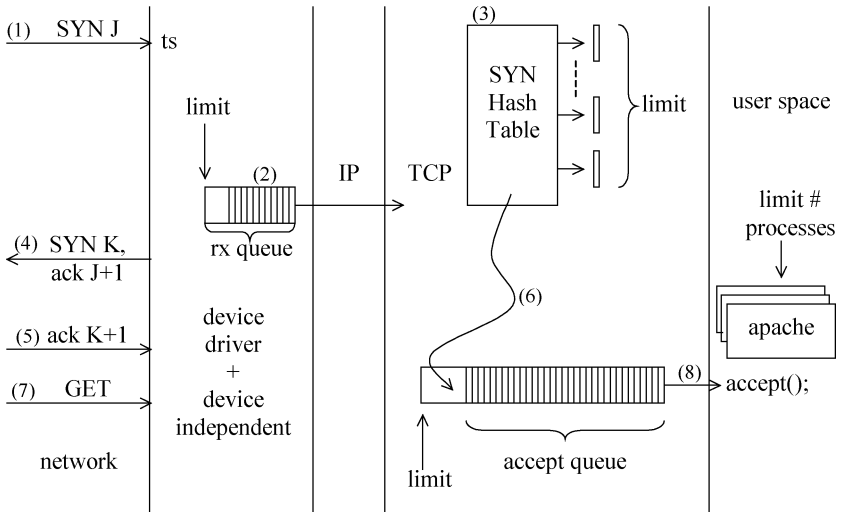

Fig. 13. TCP/IP connection establishment on Linux.

(3) During post interrupt handling, the IP layer will route the incoming SYN to TCP. If the SYN hash table is full, TCP drops the incoming SYN. Otherwise, TCP creates an open (connection) request and places it into the SYN hash table. Note that Linux does not save the timestamp for the initial SYN packet in the open request structure.

(4) TCP will respond to the incoming SYN immediately by sending a SYN/ACK to the client. If TCP cannot immediately send a SYN/ACK to the client (i.e., the tx queue in Figure 14 is full), TCP will drop the incoming SYN.

(5) The client completes the TCP 3-way handshake by sending an ACK to the server.

(6) Once TCP receives the third part of the TCP 3-way handshake from the client, the open request will be placed onto the accept queue for processing. At this point, the new child socket is created and pointed to by the open request. The connection is considered to be established at this point.

(7) The GET request could arrive at the server prior to the child connection being accepted by Apache, in which case the GET request is simply attached to the child socket as an inbound data packet.

(8) Apache accepts the newly established child connection and proceeds to process the request. The speed at which Apache can process requests, along with the limit on the number of running Apache processes affects the length of the accept queue.

Our kernel modifications with respect to Steps (1) through (8) are as follows: We added an 8-byte timestamp field to both the open request structure and the socket structure so that the timestamp of the initial SYN could be saved across the life time of the connection. In Step (3), the timestamp in the SYN is copied to the open request structure and in Step (6) it is copied from the open request structure to the child socket structure. The ACCEPTED counter

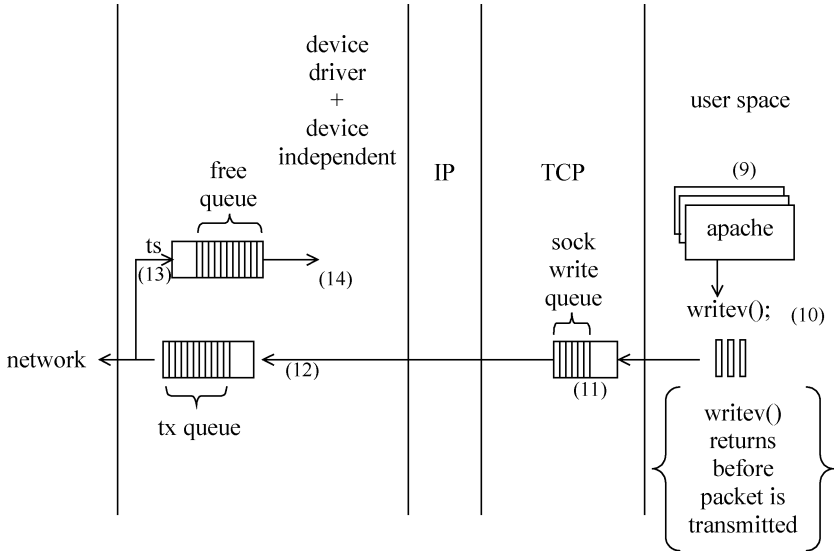

Fig. 14. TCP/IP outbound data transmission on Linux.

is also incremented during Step (6). For our DROPPED counter, we just used the existing SNMP/TCP drop counter, but fixed several functions in the kernel that either failed to increment the counter when necessary or, due to incorrect logic, incremented the counter more than once for the same SYN drop.

Figure 14 shows the outbound processing that occurs when Apache is sending data to the client. Figure 14 is numbered according to the following steps that occur during TCP outbound data transmission in an unmodified Linux kernel:

(9) Apache compiles a response to the GET request. This may include executing a program (such as CGI script or Java program) or reading a file from disk.

(10) Once the response is composed, Apache makes a socket system call to send the data (i.e., `writev`). If there is space available in the kernel for the response data, the data is copied into the kernel and then `writev()` immediately returns. If not, `writev()` will block the Apache process until space becomes available.

(11) Once the data is copied to kernel space, TCP will immediately attempt to queue the data for transmission by placing it onto the tx queue. If the tx queue is full, TCP places the data on the socket outbound write queue. If that queue is full, TCP will cause the Apache process to block.

(12) Placed onto the tx queue, the data waits to be transmitted onto the network.

(13) After the data is transmitted onto the network, the data packet is placed onto the free queue.

(14) The data packet is freed.

Our kernel modifications infringed upon Step (13). As a data packet is being placed onto the free queue, the current time is stored in the socket structure. Likewise, if the server application (i.e., Apache) closes the connection, or a TCP FIN or RST packet is received from the client, the current time is also saved in the socket structure in another timestamp field. This is also the point at which the COMPLETED counter is incremented. In this manner, we are able to identify when the server finished sending data to the client and when either the server or the client closed the connection. We choose the lesser of these two as the end of the transaction. Subtracting the timestamp obtained from the initial SYN allows us to determine the SYN-to-END time for the connection (which is then added to the running total). In other words, we defined the end of the transaction to be whichever occurs first: the last data packet is sent from the server to the client or the first arrival/transmission of a TCP RST or FIN packet.

In this section, we provided some insights into the key kernel modifications we performed, all of which were relatively minor. Other modifications, not included in the above discussion, were too far removed from the purpose of this paper to be discussed. Suffice it to say, a thorough investigation of the linux kernel TCP/IP stack was undertaken to ensure that all code paths relevant to Certes were examined. Although we provided all of the above by directly modifying and rebuilding the kernel, it would be possible to provide the identical support using a kernel module (but with greater implementation difficulty).

5. 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 5.1 describes the experimental design and the testbed used for the experiments. Section 5.2 presents the client perceived response time measurements obtained for various HTTP 1.0/1.1 web workloads. Section 5.3 demonstrates how a web server control mechanism can use Certes to evaluate its own ability to manage client response time.

5.1 Experimental Design

The testbed consisted of six machines: a web server, a WAN emulator, and four client machines (Figure 15). Each machine was an IBM Netfinity 4500R with dual 933 MHz CPUs, 10/100 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

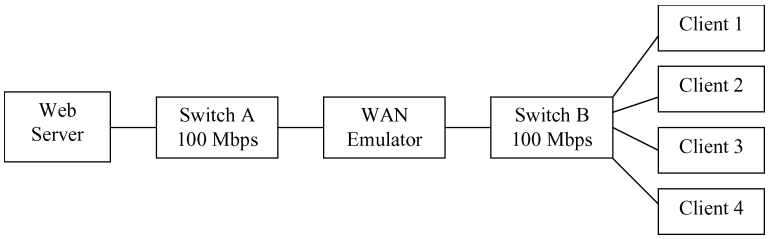

Fig. 15. Test Bed.

was DummyNet [Rizzo 1997], a flexible and commonly used FreeBSD tool. The WAN emulator simulated network environments with different network latencies, ranging from 0.3 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 (shown in Figure 12). 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 modeling 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).

The client machines ran an improved version of the Webstone 2.5 web traffic generator [WebStone]. 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 both HTTP 1.0 and HTTP 1.1 support persistent and nonpersistent connections, we configured the traffic generators to run HTTP 1.0 over nonpersistent connections and HTTP 1.1 over persistent connections. Although recent studies indicate that nonpersistent connections are still used far more frequently than

Table II. Test Configurations Included HTTP 1.0/1.1, with Static and Dynamic Pages

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%

persistent connections in practice [Smith et al. 2001], the use of persistent connections increases the duration of each connection and reduces the number of connection attempts, 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 [Smith et al. 2001].

The characteristics of the sixteen workloads are summarized in Table II. In an attempt to cover a broad range of conditions, we varied the workloads along the following dimensions:

(1) static pages and dynamic content (Perl and C)
(2) HTTP 1.0 and 1.1
(3) 1 to 15 pages per connection
(4) 0% to 3% network drop rate
(5) 5 ms to 150 ms network delays
(6) 1400 to 4800 clients (30 to 1670 conn/sec)
(7) CPU and bandwidth bound
(8) consistent and variable load.

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. For each workload, we measured at the server the steady-state number of connections per second and mean SYN

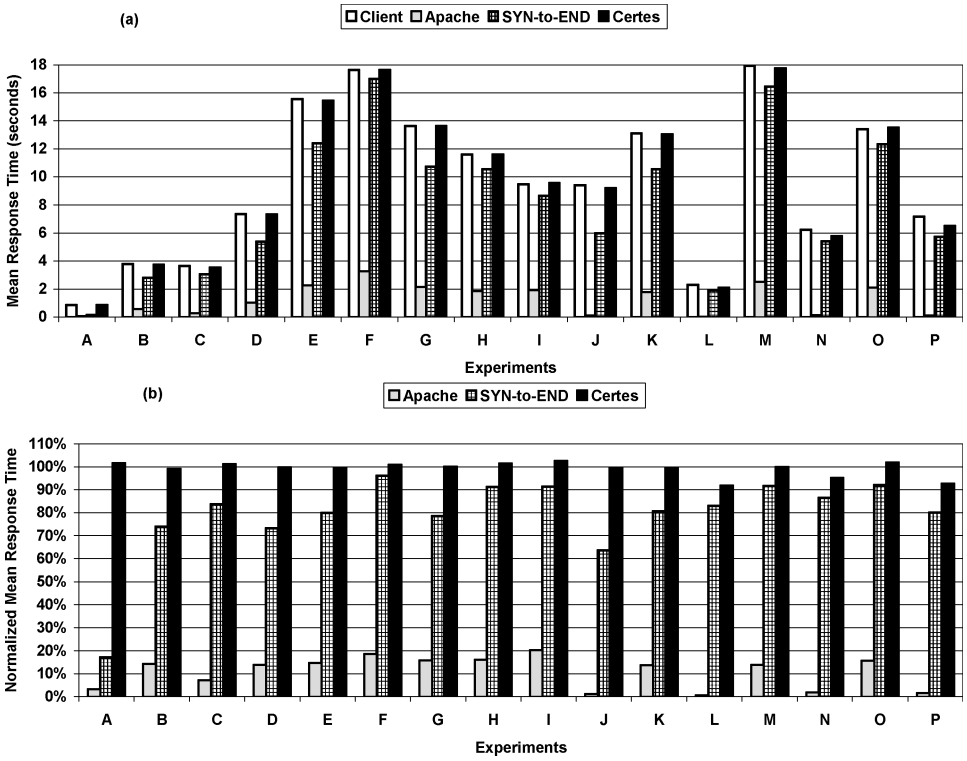

Fig. 16. Certes accuracy and stability in various environments.

drop rate during successive one-second time intervals. These measurements provide an indication of the load imposed on the server.

5.2 Measurements and Results

Figure 16(a) compares the client-side, Certes, SYN-to-END and Apache measured mean response times for each experiment. The values shown are the response times, calculated on a per second interval, averaged over the 20-minute test period. Figure 16(b) 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 13 and 14 explain why the Apache level measure of response time is so short compared to the mean client perceived response time. Apache does not measure all the inbound kernel queuing that occurs nor the time it takes to

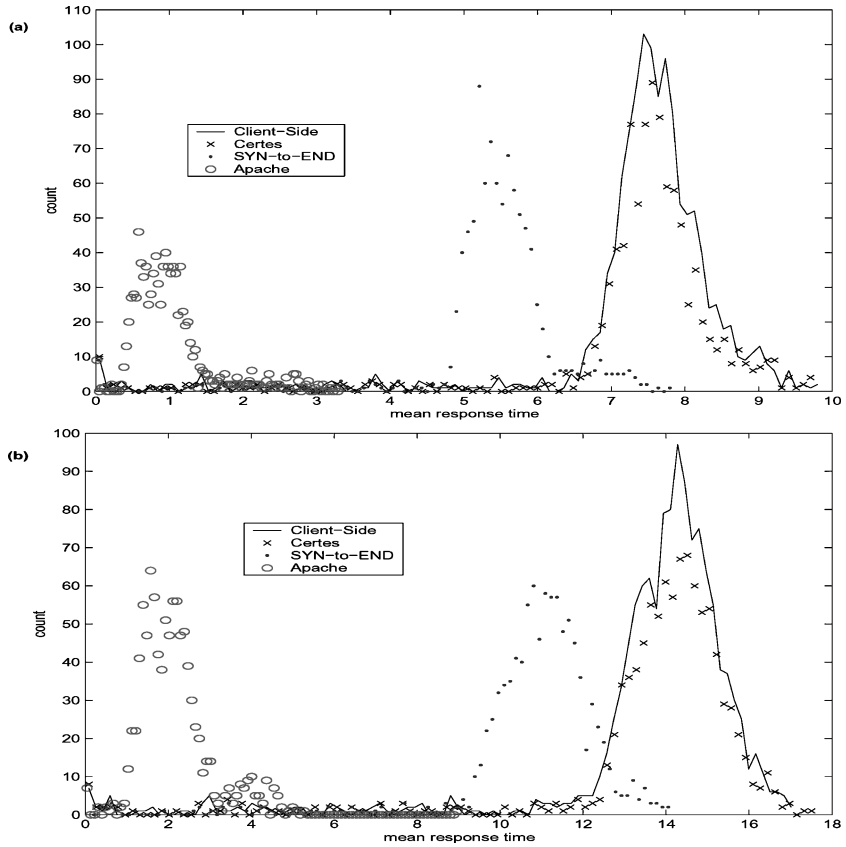

Fig. 17. Certes response time distribution approximates that of the client for Tests D and G.

perform the TCP three-way handshake. On outbound, Apache measures end of transaction before the data is transmitted (i.e., as soon as the *writev()* returns).

Figures 17(a) and 17(b), 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 17 again shows how erroneous the SYN-to-END time measurements are in estimating client perceived response time.

Figures 18(a) and 18(b) 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 18 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

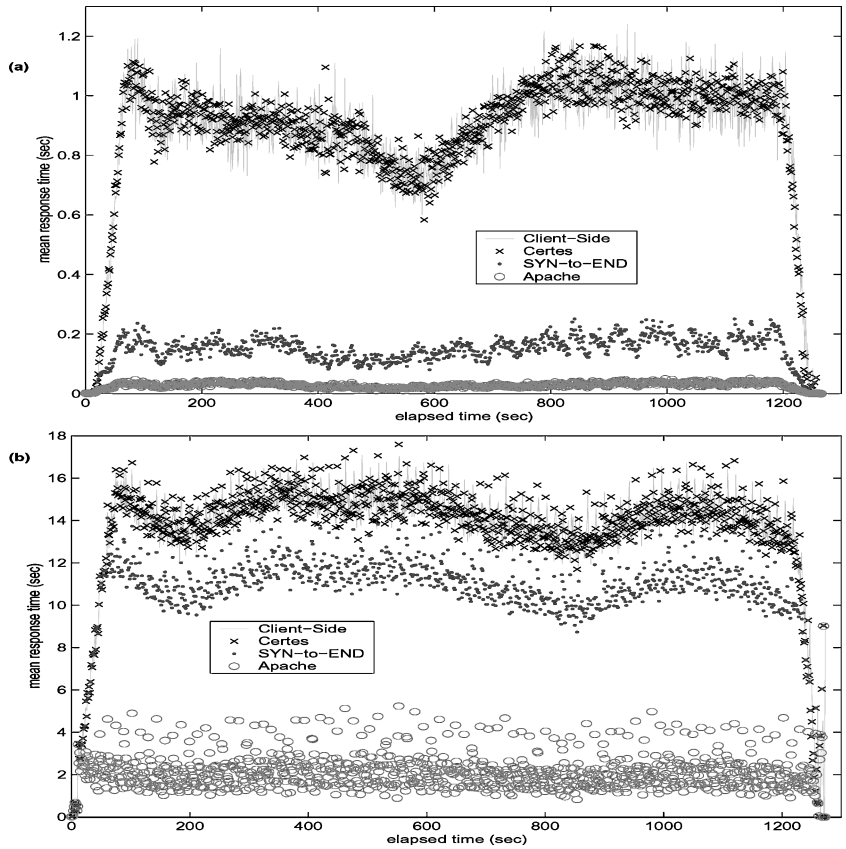

Fig. 18. Certes online tracking of the client response time in Test A and Test G.

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 16. 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 measurements 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 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

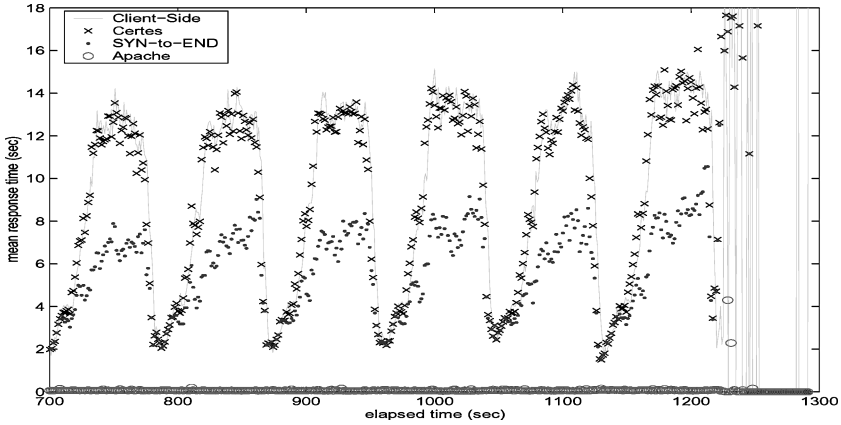

Fig. 19. Certes online tracking of the client response time in Test J, in on-off mode.

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 19 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 19), which indicates the inaccuracy of the SYN-to-END time for those connections that are accepted when the accept queue is nearly full. We note for completeness that Figure 19 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-dependent 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.

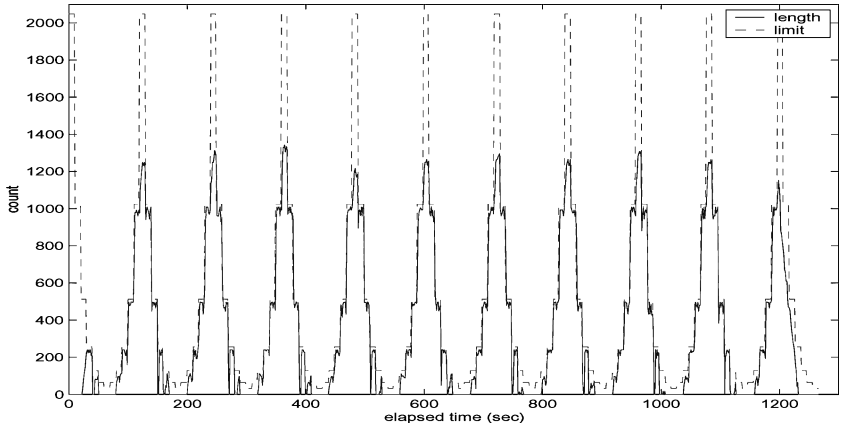

Fig. 20. Web server control manipulating the Apache accept queue limit.

5.3 Accept 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 [Li and Jamin 2002; Lu et al. 2001; Kanodia and Knightly 2000; Parekh et al. 2001; Almeida et al. 1998; Chen et al. 2001; Chen and Mohapatra 1999]. 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 server to dynamically change the Apache accept queue limit over time. Figure 20 shows the accept queue limit changing every 10 seconds between the values of 2^5 and 2^{11}. Figure 21 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 200th 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 accept 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.

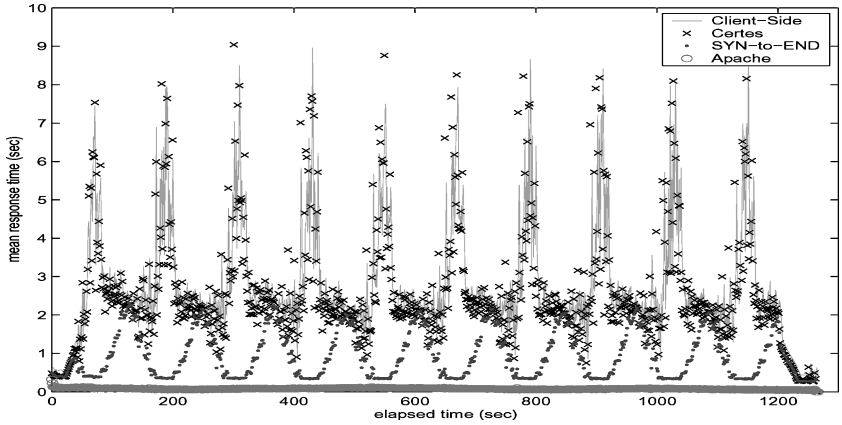

Fig. 21. Client response time increases as accept queue limit decreases.

6. CONCLUSIONS

This article 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 quantifies the effect that SYN drops have on client perceived response time by using three simple server-side measurements. Certes does not suffer from any of the drawbacks associated with the addition of new hardware, having to modify existing web pages or HTTP servers, and does not rely on third party sampling. Certes can also be used for the delivery of non-HTML objects such as PDF or PS files.

A key result of Certes is its robustness and accuracy. 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. Certes can be applied over long periods of time and does not drift or diverge from the client perceived response time; any errors that may be introduced into the model do not accumulate over time. Certes is computationally inexpensive and can be used online at the web server to provide information in real-time. Certes captures the subtle changes that can occur under constant load as well as the rapid changes that occur under bursty conditions. Certes can also determine the distribution of the client perceived response time, which 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 manage QoS by performing admissions control. Certes allows such servers to quantify the effect that admission rejection has on client perceived response time as well as allowing them to avoid the pitfalls associated with using application level or kernel level SYN-to-END measurements of response time. Certes is accurate under heavy server load, the moment at which admissions control or scheduling algorithms must make critical decisions. Algorithms that manage resource allocation, reservations or congestion [Balakrishnan et al. 1999] can benefit from the

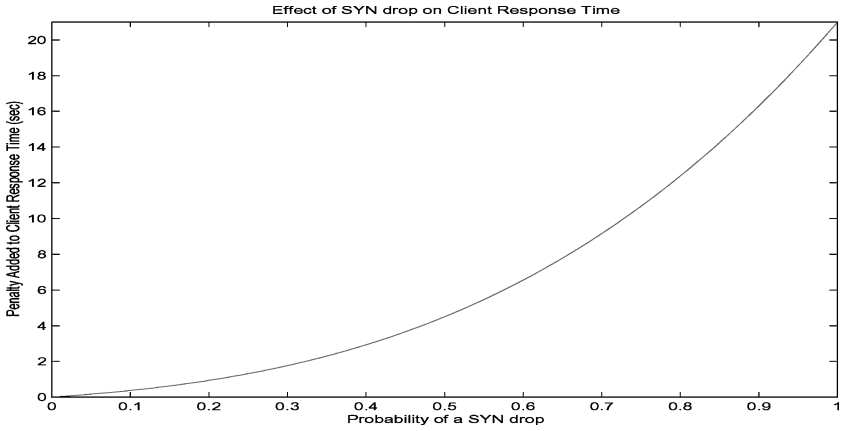

Fig. 22. Effect of SYN drop rate on client response time, as modeled as an /M/M/1 queuing system.

short-term forecasting [Cohen et al. 1999] of connection retries modeled by Certes.

APPENDIXES

A. SHORTCOMINGS OF THE (STRICTLY) QUEUING THEORETIC APPROACH

As discussed in Section 2, the related work on modeling TCP that we cite assumes that the SYN drop rate remains consistent over time (and is based on network drop probabilities and not drop rates at the server). We show here that such a queuing theoretic approach leads to an error prone result that is not nearly as accurate as the Certes model. Using an /M/M/1 queuing system to represent the web server, the steady-state expected client response time is:

$$CLIENT\_RT = (t_s + t_q) \cdot (1 - p^3) + 3p + 6p^2 + 12p^3 + \cdots, \tag{32}$$

where

t_s is the service time of the request, that is, $\frac{1}{\mu}$
t_q is the time spent waiting on the queue
p is the probability of dropping a connection request.

The assumptions for this overly simplified model is that the offered load remains constant over time and that t_s remains constant regardless of the offered load. Nevertheless, Figure 22 is a plot of Eq. (32) (with $t_s + t_q = 0.010$ seconds) showing the additional time added to the mean service time under the given SYN drop rate. For example, a drop rate of approximately 20% adds 1 second to the mean service time.

Substituting SYN-to-END for $t_s + t_q$ in Eq. (32), we can obtain the results that the /M/M/1 model produces for Test J. Figure 23 shows Figure 19 overlaid with the /M/M/1 results. The /M/M/1 model fails to track the client perceived response time as effective as Certes. This is due to its inability to track the

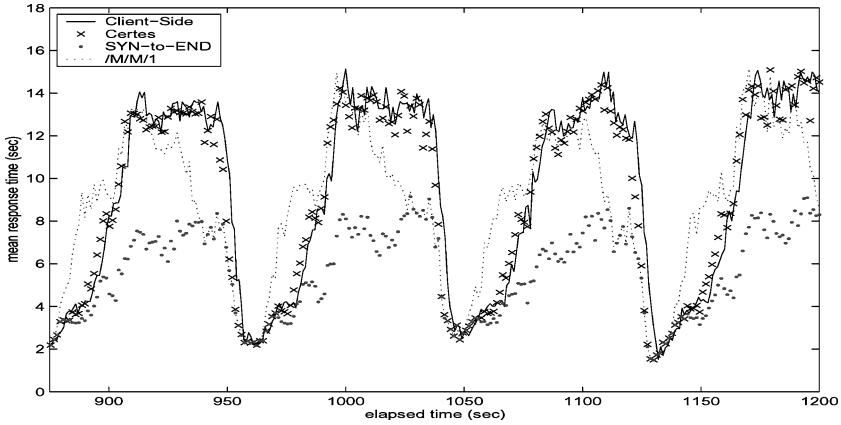

Fig. 23. Modeling as an /M/M/1 queuing system fails to accurately track client perceived response time.

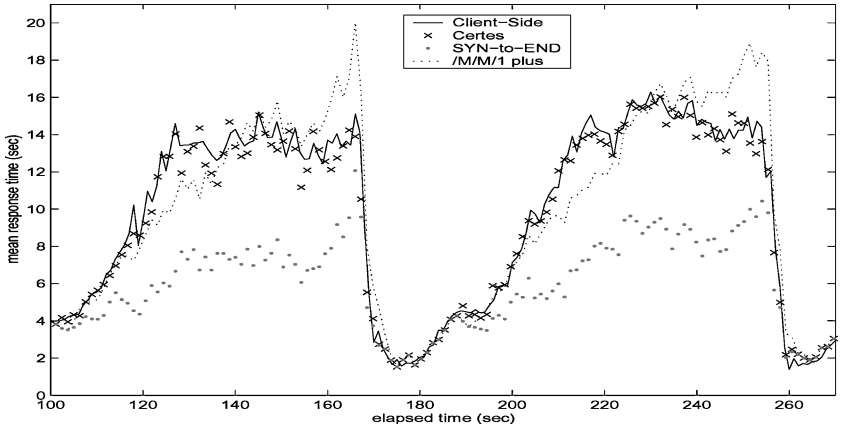

Fig. 24. Using a sliding window of drop probabilities fails to capture all the dependences between time intervals.

dependencies between time intervals. Note that this model still requires collecting the SYN-to-END time, and the number of dropped, accepted and completed for the current interval.

Equation (33) shows a more accurate approximation, using the drop probabilities from prior intervals:

$$\begin{aligned} CLIENT\_RT = \; & \\ & mean(\text{SYN-to-END}_i) \\ & 3 \cdot DR_{i-\text{SYN-to-END}-3} + \\ & 6 \cdot DR_{i-\text{SYN-to-END}-6} \cdot DR_{i-\text{SYN-to-END}-6} + \\ & 12 \cdot DR_{i-\text{SYN-to-END}-12} \cdot DR_{i-\text{SYN-to-END}-18} \cdot DR_{i-\text{SYN-to-END}-21}. \end{aligned} \tag{33}$$

This approach captures some, but not all, of the dependences that exist between time intervals. The results from applying Eq. (33) to Test J are shown in Figure 24. Note that to apply this approach, a sliding window of the number of

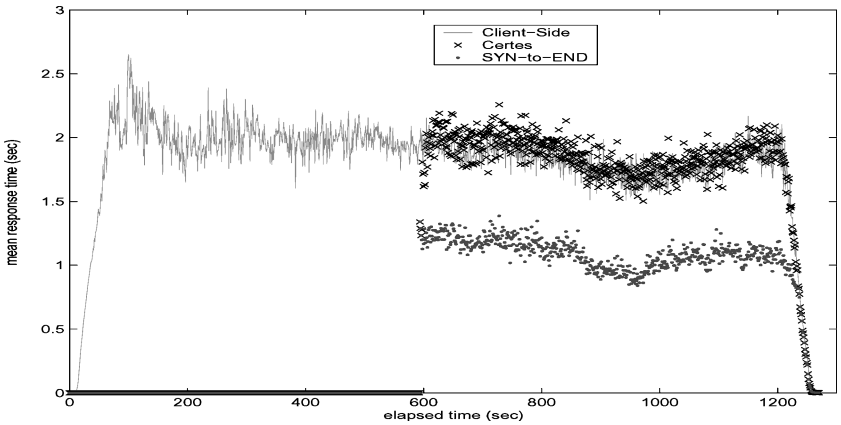

Fig. 25. Certes begins modeling at the 600th interval during a consistent load test.

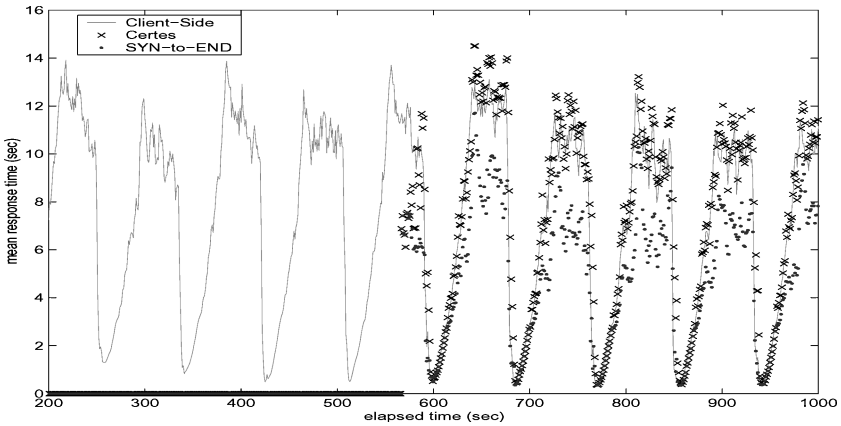

Fig. 26. Certes begins modeling at the 575th interval (in the middle of a peak) during a variable load test.

dropped, accepted and completed is required—exactly that which is required by Certes. Therefore, Certes gives a more accurate result using the same information at an equivalent computational cost.

B. CONVERGENCE

The online implementation of Certes makes the assumption that during the first interval, all SYNs are initial SYNs. Certes will converge if this assumption is not true—that is, if Certes begins modeling during the ith interval. Figures 25 and 26 are the results of starting the Certes modeling halfway through the execution of a consistent and variable load experiment. Figure 25 is similar to Test A except that the accept queue limit was set to 512 and Figure 26 is a re-execution of Test J. Figures 25 and 26 represent worst-case scenarios in the sense that none of the measurements for prior intervals are available when Certes begins modeling. In both cases, Certes converges after 21 seconds, which is the FTO.

REFERENCES

ALLMAN, M. 2000. A web server's view of the transport layer. *ACM Comput. Commun. Rev. 30*, 4 (Oct.), 133–142.

ALMEIDA, J., DABU, M., MANIKUTTY, A., AND CAO, P. 1998. Providing differentiated levels of service in web content hosting. In Tech. Rep. CS-TR-1998-1364. Computer Sciences Department, University of Wisconsin-Madison.

BALAKRISHNAN, H., RAHUL, H. S., AND SESHAN, S. 1999. An integrated congestion management architecture for internet hosts. *ACM SIGCOMM Comput. Commun. Rev. 29*, 4 (Oct.), 175–187.

BARFORD, P. AND CROVELLA, M. 1999. A performance evaluation of hyper text transfer protocols. *ACM SIGMETRICS Perf. Eval. Rev. 27*, 1 (June), 188–197.

BHATTI, N. AND FRIEDRICH, R. 1999. Web server support for tiered services. *IEEE Net. 13*, 5 (Sept.-Oct.), 64–71.

BRADEN, R. 1989. *RFC 1122: Requirements for Internet Hosts—Communication layers*. IETF, `http://www.ietf.org`.

CARDWELL, N., SAVAGE, S., AND ANDERSON, T. 2000. Modeling TCP Latency. In *IEEE INFOCOMM Conference Proceedings* (Tel-Aviv, Israel). IEEE Computer Society Press, Los Alamitos, Calif., 1742–1751.

CHEN, X. AND MOHAPATRA, P. 1999. Providing differentiated service from an internet server. In *8th International Conference on Computer Communications and Networks Conference Proceedings* (Boston, Mass.). IEEE Computer Society Press, Los Alamitos, Calif., 214–217.

CHEN, X., MOHAPATRA, P., AND CHEN, H. 2001. An admission control scheme for predictable server response time for web accesses. In *10th International World Wide Web Conference Proceedings* (Hong Kong, China). 545–554.

COHEN, E., KRISHNAMURTHY, B., AND REXFORD, J. 1999. Efficient algorithms for predicting requests to web servers. In *IEEE INFOCOM Conference Proceedings* (Orlando, Fla.). IEEE Computer Society Press, Los Alamitos, Calif., 284–293.

DANZIG, P. 2001. Ideas for next generation content delivery. In *NOSSDAV 2001* (Port Jefferson, N.Y.). ACM, New York, `http://www.nossdav.org/2001/keynote_nossdav2001.ppt`.

EGGERT, L. AND HEIDEMANN, J. 1999. Application-level differentiated services for web servers. *WWW J. 3*, 2 (Aug.), 133–142.

EXODUS. `http://www.exodus.com/`.

FREEBSD. `http://www.FreeBSD.org/`.

FU, Y., CHERKASOVA, L., TANG, W., AND VAHDAT, A. 2002. EtE: Passive end-to-end internet service performance monitoring. In *USENIX Conference Proceedings* (Monterey, Calif.). 115–130.

GOLUB, G. H. AND LOAN, C. F. V. 1996. *Matrix Computations*. The John Hopkins University Press, Baltimore, Md.

KANODIA, V. AND KNIGHTLY, E. 2000. Multi-class latency-bounded web services. In *IEEE/IFIP IWQoS Conference Proceedings* (Pittsburgh, Pa).

KEYNOTE. `http://www.keynote.com/`.

LI, K. AND JAMIN, S. 2002. A measurement-based admission-controlled web server. In *IEEE INFOCOMM Conference Proceedings* IEEE, New York, NY, 651–659.

LU, C., ABDELZAHER, T., STANKOVIC, J., AND SON, S. H. 2001. A feedback control approach for guaranteeing relative delays in web server. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium* (Taipei, Taiwan). IEEE Computer Society Press, Los Alamitos, Calif.

MERCURY INTERACTIVE. `http://www-heva.mercuryinteractive.com/`.

MICROSOFT. `http://www.MicroSoft.com/`.

MOCKAPETRIS, P. 1987a. RFC 1034: Domain names concepts and facilities. IETF, `http://www.ietf.org`.

MOCKAPETRIS, P. 1987b. RFC 1035: Domain names implementation and specification. IETF, `http://www.ietf.org`.

NAHUM, E., BARZILAI, T., AND KANDLUR, D. 1999. Performance issues in WWW servers. *ACM SIGMETRICS Performance Evaluation Review 27*, 1 (May), 216–217.

NETBSD. `http://www.NetBSD.org/`.

NETQOS. `http://www.NetQoS.com/`.

NIELSEN, H. F., GETTYS, J., BAIRD-SMITH, A., PRUD'HOMMEAUX, E., LIE, H. W., AND LILLEY, C. 1997. Network performance effects of HTTP/1.1, CSS1, and PNG. *ACM SIGCOMM Comput. Commun. Rev. 27*, 4 (Oct.), 155–166.

OLSHEFSKI, D., NIEH, J., AND AGRAWAL, D. 2002. Inferring client response time at the web server. In *ACM SIGMETRICS Conference Proceedings* (Marina Del Rey, Calif.). ACM, New York, 160–171.

ONESTAT. 2002. Microsoft's windows OS global market share is more than 97% according to OneStat.com. *OneStat Press Release*.

PADHYE, J., FIROIU, V., TOWSLEY, D., AND KUROSE, J. 1998. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM Comput. Commun. Rev. 28*, 4 (Oct.), 303–314.

PAHDYE, J. AND FLOYD, S. 2001. On inferring TCP behavior. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (San Diego, Calif.). ACM New York, 287–298.

PANDEY, R., BARNES, J. F., AND OLSSON, R. 1998. Supporting quality of service in HTTP servers. In *Proceedings of the 17th annual ACM Symposium on Principles of Distributed Computing* (Puerto Vallarta, Mexico). ACM New York, 247–256.

PAPOULIS, A. AND PILLAI, S. U. 2001. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill Series in Electrical Engineering.

PAREKH, S., GANDHI, N., HELLERSTEIN, J., TILBURY, D., JAYRAM, T., AND BIGUS, J. 2001. Using control theory to achieve service level objectives in performance management. In *IFIP/IEEE International Symposium on Integrated Network Management Conference Proceedings* (Seattle, Wash.). IEEE Computer Society Press, Los Alamitos, Calif. 841–854.

POSTEL, J. 1981. RFC 793: Transmission Control Protocol. IETF, `http://www.ietf.org`.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 1992. *Numerical Recipes in C: The Art of Scientific Computing, 2nd Edition*. Cambridge University Press, Cambridge, United Kingdom.

RAJAMONY, R. AND ELNOZAHY, M. 2001. Measuring client-perceived response times on the WWW. In *3rd USENIX Symposium on Internet Technologies and Systems (USITS) Conference Proceedings (USITS '01)* (San Francisco, Calif.).

REDHAT. `http://www.RedHat.com/`.

RIZZO, L. 1997. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Comput. Commun. Rev. 27*, 1 (Jan.), 31–41.

SMITH, F. D., CAMPOS, F. H., JEFFAY, K., AND OTT, D. 2001. What TCP/IP protocol headers can tell us about the web. *ACM SIGMETRICS Perf. Eval. Rev. 29*, 1 (June), 245–256.

SRISURESH, P. AND EGEVANG, K. 2001. *RFC 3022: Traditional IP Network Address Translator (Traditional NAT)*. IETF, `http://www.ietf.org`.

SRISURESH, P. AND HOLDREDGE, M. 1999. *RFC 2663: IP Network Address Translator (NAT) Terminology and Considerations*. IETF, `http://www.ietf.org`.

STEVENS, W. R. 1994. *TCP/IP Illustrated, Volume 1 The Protocols*. Addison-Wesley, Reading, Mass.

STREAMCHECK. `http://www.streamcheck.com/`.

VOIGT, T., TEWARI, R., MEHRA, A., AND FREIMUTH, D. 2001. Kernel mechanisms for service differentiation in overloaded web servers. In *USENIX Conference Proceedings* (Boston, Mass.). 189–202.

WEBSTONE. `http://www.mindcraft.com/`.

YAJNIK, M., MOON, S., KUROSE, J., AND TOWSLEY, D. 1999. Measurement and modeling of the temporal dependence in packet loss. In *IEEE INFOCOM Conference Proceedings* (Orlando, Fla.). IEEE Computer Society Press, Los Alamitos, Calif., 345–352.

ZHANG, Y., PAXSON, V., AND SHENKER, S. 2000. The Stationarity of Internet Path Properties: Routing, Loss and Throughput. In Tech. Rep. ACIRI.

Received August 2002; revised January 2003; accepted September 2003