2014

TCP Congestion Avoidance Algorithm Identification

Peng Yang

Juan Shao

Wen Luo

Lisong Xu
University of Nebraska-Lincoln, xu@cse.unl.edu

Jitender S. Deogun
University of Nebraska-Lincoln, jdeogun1@unl.edu

See next page for additional authors

Pages: 1311 - 1324, DOI: 10.1109/TNET.2013.2278271

TCP Congestion Avoidance Algorithm Identification

Peng Yang, *Member, IEEE*, Juan Shao, Wen Luo, Lisong Xu, *Member, IEEE*, Jitender Deogun, *Member, IEEE*, and Ying Lu, *Member, IEEE*

***Abstract*—The Internet has recently been evolving from homogeneous congestion control to heterogeneous congestion control. Several years ago, Internet traffic was mainly controlled by the traditional RENO, whereas it is now controlled by multiple different TCP algorithms, such as RENO, CUBIC, and Compound TCP (CTCP). However, there is very little work on the performance and stability study of the Internet with heterogeneous congestion control. One fundamental reason is the lack of the deployment information of different TCP algorithms. In this paper, we first propose a tool called TCP Congestion Avoidance Algorithm Identification (CAAI) for actively identifying the TCP algorithm of a remote Web server. CAAI can identify all default TCP algorithms (e.g., RENO, CUBIC, and CTCP) and most non-default TCP algorithms of major operating system families. We then present the CAAI measurement result of about 30 000 Web servers. We found that only $3.31\% \sim 14.47\%$ of the Web servers still use RENO, 46.92% of the Web servers use BIC or CUBIC, and $14.5\% \sim 25.66\%$ of the Web servers use CTCP. Our measurement results show a strong sign that the majority of TCP flows are not controlled by RENO anymore, and a strong sign that the Internet congestion control has changed from homogeneous to heterogeneous.**

***Index Terms*—Heterogeneous congestion control, Internet measurement, TCP congestion control.**

I. Introduction

THE INTERNET has recently been evolving from homogeneous congestion control to heterogeneous congestion control. A few years ago, Internet traffic was mainly controlled by the same TCP congestion control algorithm—the standard Additive-Increase-Multiplicative-Decrease algorithm [2], [3] which is usually called RENO.[1] However, Internet traffic is now controlled by multiple different TCP algorithms. For example, Table I lists all the TCP algorithms available in two major operating system families: Windows family (e.g., Windows XP/Vista/7/Server) and Linux family (e.g., RedHat, Fedora, Debian, Ubuntu, SuSE). Both Windows and Linux users can change their TCP algorithms with only a single line of command. Linux developers can even design and then add their own TCP algorithms.

TABLE I
TCP Algorithms Available in Major Operating System Families

Operating Systems	TCP algorithms
Windows family	RENO [2], and CTCP [8]
Linux family	RENO, BIC [12], CUBIC [13], HSTCP [14], HTCP [15], HYBLA [16], ILLINOIS [17], LP [18], STCP [19], VEGAS [20], VENO [21], WESTWOOD+ [22], and YEAH [23]

There is, however, very little work [4]–[6] on the performance and stability study of the Internet with heterogeneous congestion control. One fundamental reason is the lack of the deployment information of different TCP algorithms in the Internet. As an analogy, if we consider the Internet as a country, an Internet node as a house, and a TCP algorithm running at a node as a person living at a house, the process of obtaining the TCP deployment information can be considered as the TCP algorithm census in the country of the Internet. Just like the population census is vital for the study and planning of the society, the TCP algorithm census is vital for the study and planning of the Internet.

Question 1: Are the Majority of TCP Flows Still Controlled by Reno?: This is an important question because most of recently proposed congestion control algorithms, such as CUBIC [7], CTCP [8], DCCP [9], and SCTP [10], are designed to perform well when competing with the traditional RENO, but yet be friendly with the competing RENO traffic (usually referred to as TCP friendliness). If the majority of TCP flows are not controlled by RENO anymore, it is necessary to reevaluate not only the performance but also the design goals of these congestion control algorithms. For example, if CTCP becomes the dominating algorithm in the Internet, should new congestion control algorithms be designed to be friendly to CTCP instead of RENO?

Question 2: What Percentage of Internet Nodes Use a Specific TCP Algorithm?: This is an important question for designing new congestion control algorithms and evaluating existing congestion control algorithms, such as studying interprotocol fairness issues [4], [6] among different TCP algorithms. More importantly, this is an important question for designing and dimensioning other Internet components. As an example, the sizes of router buffers [11] are usually determined by assuming that all TCP flows are controlled by RENO, and for instance the well-known rule-of-thumb that sets the buffer size of a link to its bandwidth delay product directly comes from the RENO assumption. For an Internet with heterogeneous congestion control, it is important to know the answer to question 2

Manuscript received September 25, 2012; revised May 27, 2013; accepted July 11, 2013; approved by IEEE/ACM Transactions on Networking Editor S. Fahmy. Date of publication September 09, 2013; date of current version August 14, 2014. This work was supported in part by the NSF under Grants CAREER CNS-0644080 and CNS-1017561. Part of this work was presented at the IEEE International Conference on Distributed Computing Systems (ICDCS), Minneapolis, MN, USA, June 20–24, 2011.

The authors are with the Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE 68588-0115 USA (e-mail: pyang@cse.unl.edu; jshao@cse.unl.edu; wluo@cse.unl.edu; xu@cse.unl.edu; deogun@cse.unl.edu; ylu@cse.unl.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2013.2278271

[1]In this paper, we use RENO to refer to the traditional congestion control algorithm used in both Reno, NewReno, and SACK.

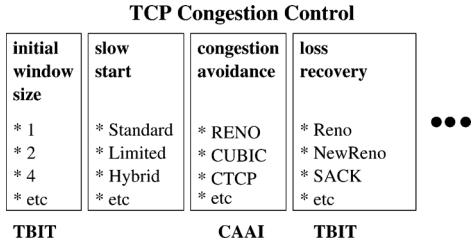


Fig. 1. TCP congestion control components.

in order to determine the sizes of router buffers. As another example, the design of Active Queue Management (AQM) mechanisms is also highly dependent on the TCP algorithms used by Internet nodes.

This paper has two main contributions. *First*, we propose a tool called *TCP Congestion Avoidance Algorithm Identification* (CAAI) for actively identifying the TCP algorithm of a remote Web server. The reason that we consider Web servers is that Web traffic comprises a significant portion of the total Internet traffic (between 16% and 34% in different regions according to ipoque [24] in 2009, and between 16% and 25.9% in different regions according sandvine [25] in 2012). CAAI can identify all default TCP algorithms (i.e., RENO, BIC, CUBIC, and CTCP) and most non-default TCP algorithms of major operating system families and can be used to conduct the TCP algorithm census. It is very challenging to develop CAAI due to the fact that Internet nodes do not explicitly report their TCP algorithms. With the population census analogy, it would be very challenging to gather the population information if people did not tell their information. After an overview of CAAI in Section III, we describe the three steps of CAAI in Sections IV–VI, respectively: 1) how to design and emulate some specific network environments in which different TCP algorithms behave differently; 2) how to extract the unique features of a TCP algorithm from the collected TCP behavior traces; 3) how to identify the TCP algorithm of a Web server based on its TCP features.

Second, we demonstrate the potential applications of CAAI by presenting our measurement results of about 30 000 Web servers in Section VII. We found that only $3.31\% \sim 14.47\%$ of the Web servers still use RENO, 46.92% of the Web servers use BIC or CUBIC, and $14.5\% \sim 25.66\%$ of the Web servers use CTCP. In addition, we found that some TCP algorithms have several versions, and the early versions are still used by a large fraction of Web servers. For example, $13.41\% \sim 24.57\%$ of the Web servers still use an early version of CTCP. We also found that some Web servers use non-default TCP algorithms (such as HTCP).

II. TCP Congestion Control and Related Works

TCP congestion control consists of several important components, such as the initial window size, slow start, congestion avoidance, loss recovery, etc., as illustrated in Fig. 1. The initial window size could be 1, 2 [26], 3, 4 [27], or even 10 [28] packets. The slow start algorithm could be the standard slow start [26], limited slow start [29], hybrid slow start [30], etc. The congestion avoidance algorithm could be RENO [2], CUBIC [13], CTCP [8], etc. The loss recovery mechanism could be Reno [31], NewReno [32], SACK [33], DSACK [34], etc. Note that, we can create different TCP congestion control algorithms with different combinations of these components. For example, CUBIC can be combined with the standard slow start or the hybrid slow start or other slow start algorithms, and it can be combined with NewReno or SACK or other loss recovery mechanisms.

CAAI proposed in this paper only considers how to identify the TCP congestion avoidance component of a Web server. The initial window size and the loss recovery components of a Web server can be identified by TBIT [35] (described later in this section). Very few slow start algorithms have been implemented in major operating systems, and therefore we do not consider how to identify them in this paper.

Because this paper considers only the congestion avoidance component of a TCP congestion control algorithm, we use the term "*a TCP congestion avoidance algorithm*" or "*a TCP algorithm*" to refer to the congestion avoidance component of a TCP congestion control algorithm. For example, when we say that a TCP algorithm is CUBIC, it means that the congestion avoidance component of the TCP congestion control algorithm is CUBIC. We first review related works on identifying TCP congestion avoidance components, and then review related works on inferring other TCP congestion control components.

1) Related Works on Identifying TCP Congestion Avoidance Components: Because most TCP algorithms listed in Table I were proposed recently, there are very few papers on identifying them. Oshio *et al.* [36] propose a cluster analysis-based method for a router to distinguish between two different TCP algorithms. Feyzabadi *et al.* [37] consider how to detect whether the TCP algorithm of a Web server is RENO or CUBIC. Our proposed CAAI is different from their works in that: 1) CAAI can distinguish among most TCP algorithms available in major operating system families, whereas their works consider only two different TCP algorithms; 2) we have solved many Web and TCP issues so that we can conduct a large scale of Internet experiments, whereas their works are mainly based on simulations.

Our early work [38] proposes a method to infer the TCP multiplicative decrease parameter of a Web server, which is an important TCP feature for identifying TCP algorithms. This paper differs from our early work [38] in the following ways: 1) this paper considers how to distinguish among different TCP algorithms, whereas our early work only considers how to extract a TCP feature—the TCP multiplicative decrease parameter; and 2) this paper presents, for the first time, the TCP deployment information of about 30 000 Web servers.

2) Related Works on Inferring Other TCP Congestion Control Components: There are a large number of papers on inferring other TCP congestion control components, and they can be classified into two categories: active measurements [35], [39]–[41], which actively measure the TCP behaviors of Internet nodes, and passive measurements [42]–[46], which measure the TCP behaviors of Internet flows in passively collected packet traces.

We review the two most relevant works. TBIT [35] is a popular active measurement tool for inferring TCP behavior of a remote Web server. It can infer various TCP behaviors such as the initial window size, the loss recovery mechanisms, congestion window halving, etc. However, it cannot identify the congestion avoidance algorithms, simply because most congestion avoidance algorithms listed in Table I were proposed after TBIT was developed. CAAI is implemented by extending the source code of TBIT. Specifically, CAAI only uses part of the TBIT

code to communicate raw TCP packets with a Web server. We wrote our own code to emulate two network environments, to extract two TCP features, and to identify the TCP congestion avoidance algorithm.

NMAP [41] is a popular active measurement tool for inferring the information, such as the operating system, of a remote Internet node. However, it is hard to infer the TCP algorithm of a remote Internet node, even if we can detect the operating system of the node for the following reasons. Even though an operating system has a default TCP algorithm, a user can easily change the default algorithm.

III. CAAI Overview

A. Design Goals

CAAI is designed to actively identify the TCP congestion avoidance algorithm of a remote Web server.

We have the following design goals for CAAI.

- Design goal 1: It can identify all default TCP algorithms and most non-default TCP algorithms of major operating system families.
- Design goal 2: It is insensitive to the operating system of a Web server, insensitive to network conditions, and insensitive to TCP components other than the congestion avoidance component.

For the first design goal, we consider a total of 14 TCP algorithms: RENO [2], BIC [12], CTCP′ and CTCP″ [8], CUBIC′ and CUBIC″ [13], HSTCP [14], HTCP [15], ILLINOIS [17], STCP [19], VEGAS [20], VENO [21], WESTWOOD+ [22], and YEAH [23]. RENO is the default TCP of some Windows operating systems and some Linux operating systems. CTCP is the default TCP of some Windows operating systems. We found that there are two versions of CTCP: The earlier one implemented in Windows Server 2003 and XP is referred to as CTCP′, and the later one implemented in Windows Server 2008, Vista, and 7 is referred to as CTCP″. BIC and CUBIC are the default TCP of some Linux operating systems. Since CUBIC was included into Linux Kernel in 2006, it has had several major changes [13]. We consider two major versions of CUBIC: The one implemented in Linux kernel 2.6.25 and before is referred to as CUBIC′, and the one implemented in Linux kernel 2.6.26 and after is referred to as CUBIC″. Finally, among all TCP algorithms listed in Table I, we do not consider two TCP algorithms, HYBLA [16] and LP [18], because they are not designed for Web servers. Specifically, HYBLA [16] is primarily designed for satellite connections, and LP [18] is designed for background file transfer.

The second design goal enables us to accurately identify the TCP algorithms of as many Web servers as possible. Insensitivity to the operating system of a Web server is desirable because the same TCP algorithm can be implemented into different operating systems. Insensitivity to network conditions (e.g., packet loss, delay, reordering, and duplication) is desirable because we have no control of the network condition between a CAAI computer and a remote Web server. Insensitivity to TCP components other than congestion avoidance is desirable because the TCP behavior of a Web server is controlled not only by its TCP congestion avoidance component, but also by many other TCP components (as listed in Fig. 1).

B. TCP Algorithm Features

A TCP congestion avoidance algorithm can be well described by the following two features.

- *Feature 1: Multiplicative Decrease Parameter (denoted by β)*, which determines the slow start threshold (i.e., the boundary congestion window size between the slow start and congestion avoidance states).
- *Feature 2: Window Growth Function (denoted by $g(\cdot)$)*, which determines how a TCP algorithm grows its congestion window size in the congestion avoidance state.

Because this paper considers only the congestion avoidance component of a TCP congestion control algorithm, we use the term "*window*" to refer to the congestion window of a TCP congestion control algorithm. Let $loss\_cwnd$ denote the window size just before a loss event or a timeout. In case of a loss event, TCP usually sets both its slow start threshold and window size to $\beta \times loss\_cwnd$. In case of a time out, TCP sets its slow start threshold to $\beta \times loss\_cwnd$, and usually sets its window size to 1 packet. Different TCP algorithms usually have different multiplicative decrease parameters. For example, RENO sets $\beta = 0.5$; CUBIC sets $\beta = 0.7$; and STCP sets $\beta = 0.875$. Some TCP algorithms have a variable β that depends on $loss\_cwnd$ and the network environment such as the duration of a round-trip time (RTT), the minimum RTT, and the maximum RTT. For example, BIC sets $\beta = 0.8$ if $loss\_cwnd > 14$, and sets $\beta = 0.5$ if $loss\_cwnd \leq 14$; HSTCP sets β between 0.5 and 0.9 depending on $loss\_cwnd$; HTCP sets β between 0.5 and 0.8 depending on the ratio of the minimum RTT and the maximum RTT.

Different TCP algorithms usually have different window growth functions. The window growth function of a TCP algorithm is usually a function of the elapsed number of RTTs in the congestion avoidance state (denoted by x) and $loss\_cwnd$. For example, RENO has a linear window growth function of x (i.e., $g(x, loss\_cwnd) = 0.5 \times loss\_cwnd + x$), and STCP has an exponential window growth function of x (i.e., $g(x, loss\_cwnd) = 0.875 \times loss\_cwnd \times 1.02^x$ for nondelayed ACKs). Some TCP algorithms have a window growth function that depends not only on x, but also on the network environment. For example, the CUBIC function depends on both x and the duration of an RTT, and the CTCP function depends on x, the duration of an RTT, and the minimum RTT.

Note that different TCP algorithms may show different features in one network environment, but show the same features in another network environment. Therefore, an important part of CAAI is to emulate some network environments in which different TCP algorithms have different features so that they can be distinguished from one another.

C. CAAI Steps

CAAI identifies the TCP algorithm of a remote Web server by analyzing its TCP behaviors in some emulated network environments. CAAI has the following three steps.

- *Step 1: Trace Gathering.* CAAI gathers the TCP window traces of a remote Web server in some emulated network environments.
- *Step 2: Feature Extraction.* CAAI extracts the two TCP algorithm features from the gathered TCP window traces.
- *Step 3: Algorithm Classification.* CAAI finally identifies the TCP algorithm by comparing the extracted features to the training features.

D. Design Challenges

It is very challenging to design CAAI for the following reasons.

1) It might be easy to find a network environment to distinguish two TCP algorithms. However, it is nontrivial to find a small set of network environments to clearly distinguish a large number of TCP algorithms, like 14 TCP algorithms.
2) We do not have the control of the network condition between a CAAI computer and a remote Web server. The condition of the network from a remote Web server to a CAAI computer greatly influences the TCP data packets sent from the Web server. The condition of the network from a CAAI computer to a Web server greatly influences the TCP ACK packets received by the Web server. Therefore, it is hard to emulate desired network environments, measure the TCP window sizes of a Web server, extract TCP features from the measured window traces, and identify the TCP algorithm based on the extracted TCP features.
3) We do not have the control of the content on a Web server, and most Web pages are very short. Therefore, it is hard to maintain a TCP connection between a CAAI computer and a remote Web server long enough so that CAAI can gather sufficiently long traces of TCP window sizes.

IV. CAAI Step 1: Trace Gathering

A. Overview

The first step of CAAI gathers the traces of TCP window sizes of a remote Web server in some emulated network environments. These network environments are carefully chosen so that different TCP algorithms have different features and thus they can be distinguished from one another. Specifically, for each network environment, we have the following.

- *Subtask 1:* CAAI creates a TCP connection to a remote Web server and emulates the network environment.
- *Subtask 2:* CAAI measures the TCP window sizes of the Web server in the emulated network environment.
- *Subtask 3:* CAAI maintains the TCP connection until it has gathered a sufficiently long trace of window sizes.

B. Emulated Network Environments

CAAI emulates the following two network environments with parameters $cwnd\_threshold$ and mss for a Web server.

- *Network Environment A:* CAAI sends back an ACK packet to acknowledge each TCP data packet from the Web server (i.e., nondelayed ACKs). The TCP data packets sent from the Web server are not lost until the TCP window size of the Web server becomes greater than $cwnd\_threshold$ packets. Then, the packet loss leads to a TCP timeout of the Web server (i.e., $loss\_cwnd \geq cwnd\_threshold$). After the timeout, there is again no data packet loss. In addition, there is always no data packet reordering in the emulated network. The maximum TCP segment size (MSS) is mss bytes, and the RTT between CAAI and the Web server is always 1.0 s.
- *Network Environment B:* Same as network environment A, except that the RTT is 0.8 or 1.0 s as specified in Fig. 2.

Why These Two Network Environments?: Fig. 3 shows the traces of window sizes of all 14 TCP algorithms in these two network environments. We can see that network environment A or B alone is insufficient to distinguish among 14 TCP algorithms. For example, RENO [Fig. 3(a)] and VEGAS [Fig. 3(k)] have the same trace in network environment A, and RENO and VENO [Fig. 3(l)] have very similar traces in network environment B. Both A and B together with $cwnd\_threshold = 512$ packets can clearly distinguish among all 14 TCP algorithms. Network environment A is used to collect the behavior of a TCP algorithm in a network environment with a fixed RTT, in which we can extract the two TCP features for a fixed RTT. Network environment B is used to collect the behavior of the TCP algorithm in another network environment with a varying RTT, in which we can extract another two TCP features for a varying RTT. Before the timeout in network environment B, RTT increases from 0.8 to 1.0 s after the third RTT, and this is used to check whether the β feature depends on RTT [e.g., ILLINOIS shown in Fig. 3(i) and VENO shown in Fig. 3(l)]. After the timeout, RTT increases from 0.8 to 1.0 s after the 12th RTT, and this is used to check whether the $g(\cdot)$ feature depends on RTT [e.g., CTCP$''$ shown in Fig. 3(d) and YEAH shown in Fig. 3(n)]. As explained next, $cwnd\_threshold$ is no more than 512 packets, and therefore TCP has already entered the congestion avoidance state after 12 RTTs.

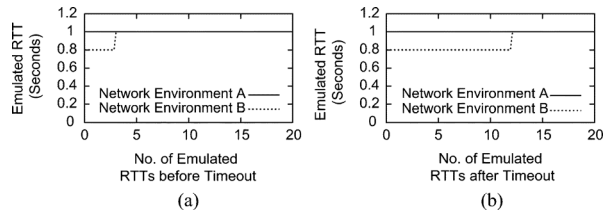


Fig. 2. RTTs of the two emulated network environments A and B. (a) Before timeout. (b) After timeout.

Values of $cwnd\_threshold$: Most TCP algorithms typically have the same or similar behavior as the traditional RENO for small window sizes (e.g., CTCP$''$ = RENO when their window sizes are less than 41), and have different behaviors than RENO for large window sizes. Therefore, window traces obtained with a large $cwnd\_threshold$ can be used to accurately distinguish among different TCP algorithms. For example, Fig. 3 shows that two network environments A and B with $cwnd\_threshold = 512$ packets can be used to clearly distinguish among all 14 TCP algorithms. However, window traces with a very large $cwnd\_threshold$ are hard to obtain because they require a very long Web page to be downloaded from a Web server, which is usually time-consuming and sometimes impossible to find on the Web server, and because the maximum achievable window size is affected by many factors such as the bandwidth-delay product of the network and the service load of the Web server. CAAI tries four $cwnd\_threshold$ values in the decreasing order of 512, 256, 128, and finally 64 packets. This is because traces with $cwnd\_threshold$ greater than 512 are hard to obtain, and traces with $cwnd\_threshold$ less than 64 are almost useless for distinguishing among different TCP algorithms. We notice that RENO, CTCP$'$, and CTCP$''$ have similar traces when $cwnd\_threshold = 64$ or 128 packets as illustrated in Fig. 3(o), therefore we do not distinguish among these three algorithms when $cwnd\_threshold = 64$ or 128 packets.

Values of mss: Since the maximum window size is limited by the ratio of the bandwidth-delay product to the MSS, we should set mss to a smaller value in order to have a higher maximum window size. CAAI tries four mss values in the increasing order of 100, 300, 536, and 1460 B. This is because a large fraction of Web servers can accept an MSS as low as 100 B, and all Web

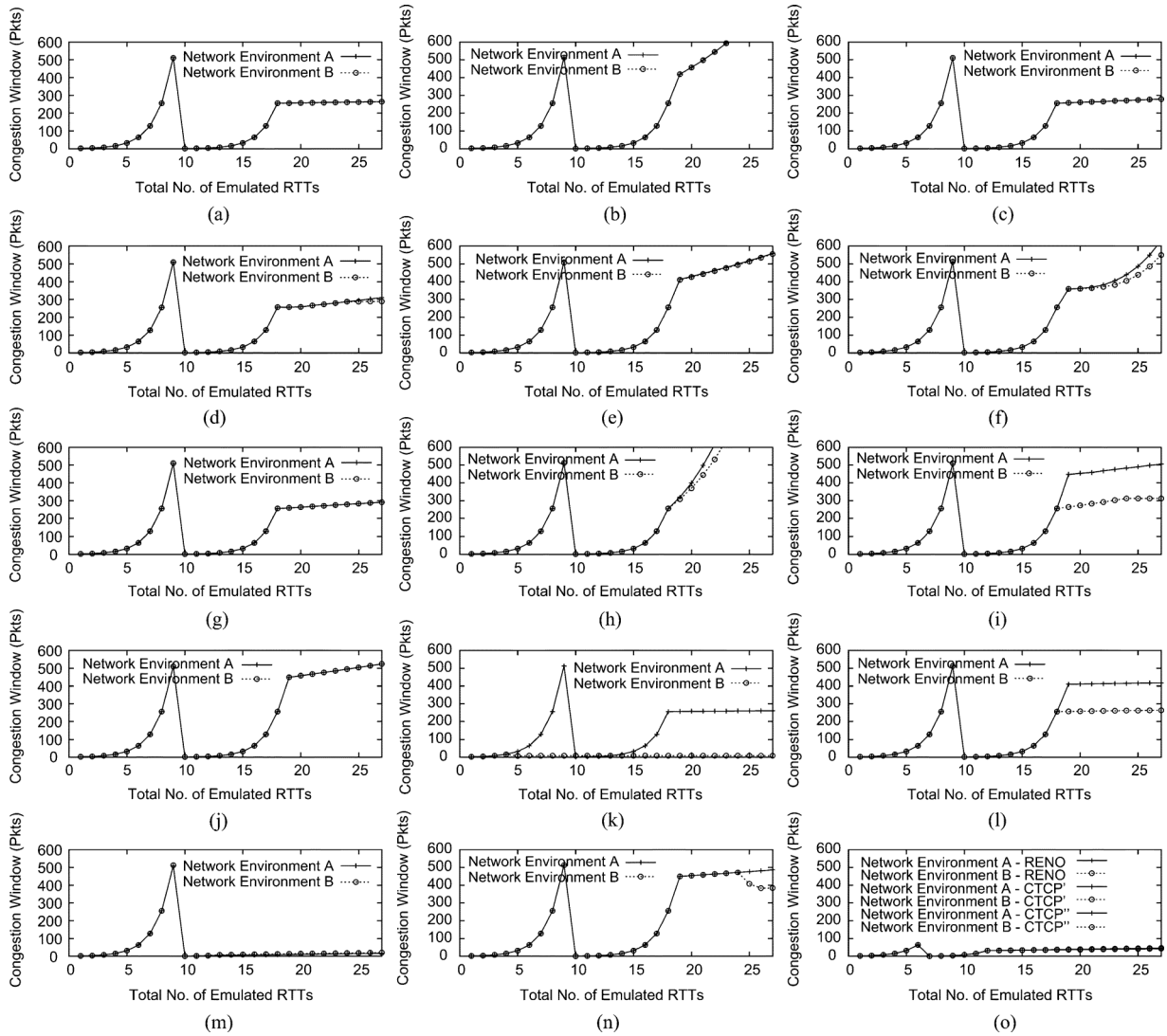


Fig. 3. Traces of window sizes of all 14 TCP algorithms in network environments A and B measured on our local testbed with a 0% packet-loss rate. (a)–(n) are obtained with $cwnd\_threshold = 512$ packets, and (o) is obtained with $cwnd\_threshold = 64$ packets. The results remain the same for different values of mss. Unless explicitly indicated, the operating system is Linux kernel 2.6.27. We can see that two network environments A and B with $cwnd\_threshold = 512$ packets can be used to clearly distinguish among all 14 TCP algorithms. (a) RENO. (b) BIC. (c) CTCP′ (Windows Server 2003). (d) CTCP′′ (Windows Server 2008). (e) CUBIC′ (Linux kernel 2.6.25). (f) CUBIC′′ (Linux kernel 2.6.27). (g) HSTCP. (h) HTCP. (i) ILLINOIS. (j) STCP. (k) VEGAS. (l) VENO. (m) WESTWOOD+. (n) YEAH. (o) RENO, CTCP′, and CTCP′′.

TABLE II
MINIMUM SEGMENT SIZES OF WEB SERVERS IN OUR EXPERIMENTS

Minimum segment size	Percentage of the web servers
100 Bytes	87.51%
300 Bytes	8.10%
536 Bytes	3.40%
1460 Bytes	0.99%

servers should accept an MSS of 1460 B. Table II shows the percentages of about 60 000 Web servers for each mss value in our experiments described in Section VII-B. We can see that most Web servers accept an mss of 100 B, and there are also a nontrivial number of Web servers that only accept an mss larger than 100 B.

Why Emulating an RTT of 1.0 s?: Because we can only emulate an RTT longer than the actual RTT between a CAAI computer and a Web server, and because we want to emulate the same two network environments for all Web servers, the emulated RTT should be longer than all actual RTTs. However,

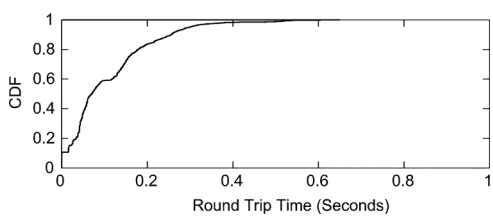


Fig. 4. CDF of the RTT of 5000 Web servers. Measured in 2010. One RTT per server.

a very long emulated RTT may cause undesired TCP timeouts (actual TCP timeouts, not our emulated TCP timeouts). Fig. 4 shows the cumulative distribution function (CDF) of the actual RTTs of 5000 popular Web servers that we measured in 2010 using Ping, and we can see that almost all actual RTTs are less than 0.8 s. In addition, the initial TCP timeout period is usually between 2.5 and 6.0 s [47]. Therefore, we can choose an emulated RTT in the range between 0.8 and 2.5 s, and CAAI chooses 1.0 s.

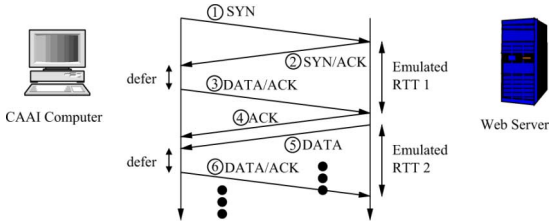


Fig. 5. TCP packets between CAAI and a remote Web server.

Value of TCP Receive Window Size: For every ACK packet, CAAI sets the receive window size field to $2^{16} - 1$ B and sets the window scale option field to 14. Therefore, the actual receive window size is $(2^{16} - 1) \times 2^{14} \approx 2^{30}$ B, which is 1 GB. This is sufficiently large so that the receive window size does not limit the rate of a TCP flow. Note that CAAI does not require such a large amount of memory since it does not save the received data.

Why Emulating a Timeout (i.e., No ACK Packet Until the Timeout) Instead of a Loss Event (i.e., Three Duplicate ACK Packets)? This is because right after a loss event, the window size depends not only on the congestion avoidance algorithm, but also heavily on some other TCP components such as burstiness control in Linux. Linux uses a special burstiness control [48], [49] to prevent TCP from sending a burst of packets to the Internet since bursty traffic may cause long queueing delay and high packet loss. However, the burstiness control interferes with congestion control on controlling the TCP window size. For example, for a Linux Web server, the window size right after a loss event may be far less than $\beta \times loss\_cwnd$, and therefore it is hard to accurately measure the two TCP features after a loss event.

Why Not Combining Multiple Network Environments Into One Longer Network Environment? A longer network environment requires a longer Web page to be downloaded from a Web server, which is usually time-consuming and sometimes impossible to find. Therefore, we prefer multiple short network environments rather than a long network environment.

C. Subtask 1: Emulating A Network Environment

Fig. 5 illustrates how a CAAI computer communicates with a Web server and how it emulates a network environment. CAAI first establishes a TCP connection to the Web server by sending a SYN packet (number 1 in the figure). The SYN packet contains a TCP option to set MSS to mss and another TCP option to enable the window scaling. After receiving the SYN/ACK packet (number 2) from the Web server, CAAI defers sending the first DATA/ACK packet (number 3) for a while so that the first RTT experienced by the Web server is equal to the first RTT of the emulated network environment. This DATA/ACK packet not only acknowledges the SYN/ACK packet, but also contains the first few HTTP request messages.

The Web server sends back an ACK packet (number 4) to acknowledge the DATA/ACK packet, and then sends its data packets (only the first packet, number 5, is shown in the figure) that contain HTTP response messages. CAAI again defers sending the next DATA/ACK packet (number 6) so that the second RTT experienced by the Web server is equal to the second RTT of the emulated network environment. This DATA/ACK packet not only acknowledges the received data packet (number 5), but also contains the next few HTTP request messages. CAAI continues acknowledging each received data packet, until the TCP window size of the Web server is greater than $cwnd\_threshold$. Then, CAAI stops sending any packet, and thus the TCP algorithm of the Web server will finally time out and retransmit the lost packets. For each data packet received after the timeout, CAAI sends back a DATA/ACK that acknowledges all data packets received so far. CAAI finally stops after the Web server does not send any more packets or after it has gathered a sufficient long trace.

How to Emulate a Network Without Any Loss or Reordering of Data Packets Except the Emulated Timeouts?: Since CAAI defers sending ACK packets, it can detect most lost and reordered packets by checking the sequence numbers of data packets received from the Web server. In case of packet loss or reordering, CAAI still sends the correct ACK packets as if there is no packet loss or reordering. Note that, however, CAAI cannot guarantee that ACK packets will be successfully delivered to the Web server, and this is one important reason for the inaccuracy of CAAI identification results.

How to Deal With Forward RTO-Recovery (F-RTO) [50]?: The emulated TCP timeout may be detected by a Web server using F-RTO as a spurious retransmission timeout. In this case, the Web server does not have a regular slow start after the emulated timeout, which is however required by CAAI to accurately determine the two TCP features. Therefore, for Web servers using F-RTO, CAAI sends a duplicate ACK after the emulated timeout in order to stop the F-RTO recovery and start the conventional TCP timeout recovery.

How to Deal With Slow Start Threshold Caching?: Usually, the initial slow start threshold of a new TCP connection is set to infinite. However, a Web server using slow start threshold caching (as part of TCP auto-tuning) sets the initial slow start threshold of a new TCP connection to the slow start threshold of the previous TCP connection of the same Web client. In this case, if CAAI emulates network environment B immediately after network environment A, the Web server exits the slow start state very early and takes a very long time to reach $cwnd\_threshold$. Therefore, for Web servers using slow start threshold caching, CAAI waits for some time (like 10 min) between emulating network environments A and B.

D. Subtask 2: Measuring the Congestion Window Sizes

We estimate the TCP window size of a Web server by the number of data packets that it sends in an emulated RTT. There are two difficulties. 1) After CAAI receives a data packet from the Web server, how to determine whether it belongs to the previous RTT or the current RTT? 2) Since a packet may be lost or duplicated in the Internet, the number of data packets received by CAAI may not be equal to the number of data packets sent by the Web server.

The first difficulty can be solved by emulating an RTT long enough so that the bandwidth-delay product is much larger than $cwnd\_threshold \times mss$. By doing so, the Web server sends all data packets belonging to the same emulated RTT in a short time interval at the beginning of an emulated RTT. After receiving all corresponding ACK packets in a short time interval at the beginning of the next emulated RTT, the Web server sends all data packets belonging to the next emulated RTT in a short time interval at the beginning of the next emulated RTT. Therefore, there is a time gap between two consecutive data packets belonging to two different emulated RTTs. Considering that the maximum $cwnd\_threshold$ of CAAI is 512 packets and the maximum mss of CAAI is 1460 B, if the bandwidth from a

Web server to a CAAI computer is at least 10 Mb/s, an emulated RTT should be longer than $512 \times 1460 \times 8/10^7 = 0.6$ s. All our emulated RTTs (i.e., 1.0 and 0.8 s) are longer than 0.6 s. If the bandwidth from a Web server to the CAAI computer is far less than 10 Mb/s, CAAI only works for very small MSS like 100 B, which is fortunately supported by most Web servers as shown in Table II. In general, we recommend to run CAAI on a computer with a fast Internet connection (e.g., 10 Mb/s and above).

The second difficulty can be solved by using the highest sequence number among all data packets that CAAI receives in an emulated RTT. CAAI measures the window size w_k of the Web server at RTT k as follows: $w_k = (h_k - h_{k-1})/mss$, where h_k is the highest sequence number among all data packets that CAAI receives in the kth RTT. In this way, as long as CAAI receives the data packet with the highest sequence number, it can accurately measure the window size. Even if the data packet with the highest sequence number is lost, CAAI can still reasonably accurately measure the window size as long as it receives the data packets with the next highest sequence numbers.

E. Subtask 3: Maintaining A TCP Connection

In order to distinguish among different TCP algorithms, CAAI must gather sufficiently long traces of window sizes. Because $cwnd\_threshold$ is no more than 512 packets, the slow start state usually takes no more than 9 RTTs. Therefore, CAAI gathers 18 RTTs of window sizes after the timeout so that TCP has usually entered the congestion avoidance state for at least 9 RTTs, which are sufficient to distinguish among all 14 TCP algorithms when $cwnd\_threshold = 512$ packets as shown in Fig. 3. Accordingly, CAAI continuously gathers a trace until the 18th RTT after the timeout, and we define *a valid trace* to be a trace that has 18 RTTs of window sizes after the timeout.

The difficulty is how to maintain the TCP connection so that CAAI can gather a valid trace of window sizes. For example, for network environment A and B with $cwnd\_threshold = 512$ packets and $mss = 100$ B, it requires about 379 kB of data for a Web server with RENO to send a total of 28 RTTs of data packets (10 RTTs before timeout and 18 RTTs after timeout). For $mss = 300$, 536, and 1460 B, it requires about 1137, 2032, and 5536 kB, respectively. CAAI uses the following two methods together to solve the problem.

CAAI repeatedly sends the same HTTP request message to a Web server using HTTP pipelining. By default, CAAI repeatedly sends a HTTP request 12 times. One might think that it is sufficient to repeatedly request the default index.html of a Web server. However, there are two issues.

1) A considerable fraction of Web servers only accept the first or the first few HTTP requests and discard the remaining requests. Fig. 6 shows the CDF of the maximum numbers of repeated HTTP requests accepted by about 60 000 Web servers in our experiments described in Section VII-B. For example, about 47% of the Web servers accept only one HTTP request, and nearly 60% of the Web servers accept three or less repeated HTTP requests.
2) Some Web servers have a very short default Web page, such as index.html and index.htm. Fig. 7 shows the CDF of the default Web page sizes. For example, only about 12% of the Web servers have a default Web page longer than 100 kB.

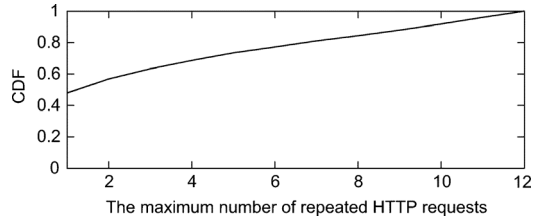


Fig. 6. CDF of the maximum numbers of repeated HTTP requests accepted by the Web servers in our experiments.

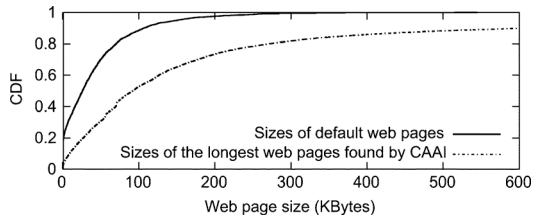


Fig. 7. CDF of the sizes of the default Web page and the longest Web pages found by CAAI.

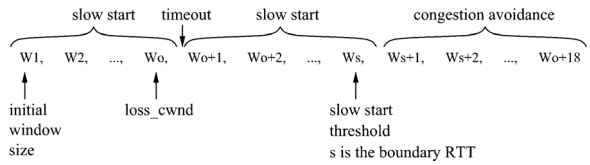


Fig. 8. Valid trace of window sizes.

Second, CAAI sends repeated HTTP request messages for a long Web page. We have developed a Web page searching tool to automatically search a Web server for a long Web page (e.g., html files, image files, or executable files). Specifically, for a Web server, our tool first uses httrack [51] to find as many Web pages as possible in 5 min (while taking care of http redirection), uses the dig service to find Web pages with the same DNS name server as the Web server, obtains the Web page headers to find their sizes without actually downloading them, and finally finds the longest Web pages. It turns out that this is the most time-consuming part of our experiments, and therefore we run this tool simultaneously on hundreds of PlanetLab nodes [52]. With the Web page searching tool, we are able to find long Web pages on a Web server. Fig. 7 shows the CDF for the sizes of the longest Web pages found by CAAI. Compared to the default Web pages, the longest Web pages found by CAAI are considerably longer. For example, the Web pages longer than 100 kB now account for about 48% of the total Web servers, as opposed to about 12% when we consider the default Web pages.

V. CAAI Step 2: Feature Extraction

This section describes how CAAI extracts the two TCP features from a valid trace. Fig. 8 illustrates a valid trace with window sizes: $w_1, w_2, \ldots, w_o, w_{o+1}, \ldots, w_{o+18}$, where w_1 is the initial window size, w_o is the window size right before the timeout, w_{o+1} is the first nonzero window size after the timeout, and w_{o+18} is the last window size of the valid trace.

In order to extract the two TCP features, CAAI first determines at which RTT (denoted by $o < s \leq o + 18$, and called *the boundary RTT*) TCP leaves the slow start state. That is, the slow start threshold is w_s. Once the boundary RTT is determined, the two TCP features can be extracted.

A. Determining the Boundary RTT

The determination of the boundary RTT is based on the fact that the standard TCP slow start is usually the default one, and the hybrid slow start [30] used by CUBIC behaves the same as the standard slow start in our emulated network environments A and B (since the RTTs of the slow start state after the timeout remain unchanged, and the emulated RTT is relatively long). That is, a TCP algorithm increases the window size by one for every received ACK in the slow start state and increases the window size relatively slowly in the congestion avoidance state. The challenge is how to check whether the window size of a Web server is increased by one for every ACK packet when ACK packets may be lost. To solve this problem, CAAI first estimates the maximum ACK loss rate in the slow start state, and then uses it to determine the boundary RTT.

First, at an RTT $k > o$ in the slow start state after the timeout, CAAI estimates the maximum ACK loss rate (denoted by q_k) on the path from CAAI to a Web server using

$$q_k = \frac{1}{n}\sum_{i=o+1}^{k-1} p_i + \frac{1.96}{\sqrt{n(n-1)}}\sqrt{\sum_{i=o+1}^{k-1} p_i^2 - \frac{1}{n}\Big(\sum_{i=o+1}^{k-1} p_i\Big)^2}. \tag{1}$$

The equation obtains q_k using the confidence interval of a total of $n = k - 1 - o$ random variables $p_{o+1}, p_{o+2}, \ldots, p_{k-1}$, where p_i denotes the ACK loss rate in RTT i. We have $p_i = (2w_i - w_{i+1})/w_i$, where the denominator w_i is the total number of ACKs in RTT i, and the numerator $2w_i - w_{i+1}$ is an estimate of the number of ACK packets lost in RTT i. This is because CAAI sends w_i ACK packets at RTT i, and if all of them are successfully received by the Web server, the window size w_{i+1} at RTT $i+1$ should be $2w_i$. The first and second terms of (1) are the average and the 95% confidence interval [53] of these n random variables, respectively. To avoid abnormal q_k values, we limit the maximum q_k to be 60% and the minimum q_k to be 15%.

Next, CAAI detects whether the window size at RTT k is increased by one for every ACK packet by checking whether $w_{k+1} > w_k + w_k(1 - q_k)$ or equivalently whether $w_{k+1}/w_k > 2 - q_k$. Starting from the smallest RTT $s > o$ such that $w_s \geq w_o/2$, CAAI searches for three consecutive RTTs s, $s+1$, and $s+2$, for all of which the window size is not increased by one for every ACK packet. RTT s is then the boundary RTT.

Note that different initial window sizes do not affect the accuracy of CAAI for the following reasons. Different TCP algorithms may use different initial window sizes, which is the window size of a new TCP connection (i.e., w_1). For example, the initial window size could be 1, 2 [26], 3, 4 [27], or even 10 [28] packets. After a timeout, TCP algorithms usually restart the window size from one packet instead of their initial window sizes [28]. CAAI only uses the window size right before the timeout and the window sizes after the timeout (i.e., $w_o, w_{o+1}, \ldots, w_{o+18}$) and does not use the initial window size (i.e., w_1). Therefore, the accuracy of CAAI does not depend on the initial window size. In addition, when determining the boundary RTT, CAAI only checks the relative ratio of two adjacent window sizes (i.e., whether $w_{k+1}/w_k > 2 - q_k$) and does not check the absolute values of the window sizes. Therefore, the accuracy of CAAI does not depend on the first window size after a timeout (i.e., w_{o+1}) either.

B. Feature 1: Multiplicative Decrease Parameter

Feature β can be obtained by $\beta = w_s/loss\_cwnd$ where w_s is the window size at the boundary RTT, and $loss\_cwnd$ is the window size right before the timeout (i.e., $loss\_cwnd = w_o$). In most cases, CAAI can find the boundary RTT and then calculate the β value. In these cases, to avoid abnormal β values, we limit the minimum β to be 0.5 (the lowest value of all 14 TCP algorithms, except WESTWOOD+), and the maximum β to be 2.0. Sometimes, CAAI could not find the boundary RTT because the congestion window sizes $w_{o+1}, w_{o+2}, \ldots, w_{o+18}$ are always below $w_o/2$ [e.g., for WESTWOOD+ as illustrated in Fig. 3(m)]. In these cases, we set β to 0.

C. Feature 2: Window Growth Function

The window growth function of a trace can be described by the window sizes after boundary RTT s. Specifically, we use $w_{s+4} - w_{s+1}$ and $w_{s+9} - w_{s+1}$ to describe the window growth function of a trace. There are two reasons. First, we use only two window sizes (i.e., w_{s+4} and w_{s+9}) instead of all the window sizes after boundary RTT s (i.e., $w_{s+1}, w_{s+2}, \ldots, w_{s+9}, \ldots, w_{o+18}$. Note that, $s + 9 \leq o + 18$). This is because, within the first few RTTs of the congestion avoidance state, the window growth functions of these 14 TCP algorithms can be sufficiently well approximated by two line segments so that they can still be distinguished from one another. Second, we use the offset window size (i.e., $w_{s+4} - w_{s+1}$ and $w_{s+9} - w_{s+1}$) instead of the actual window size (i.e., w_{s+4} and w_{s+9}). This is because, for most TCP algorithms, the offset window sizes remain the same (or almost same) for traces with different $loss\_cwnd$. For example, $w_{s+4} - w_{s+1}$ of RENO is always 3 for any $loss\_cwnd$, but w_{s+4} depends on the specific $loss\_cwnd$.

D. Feature Vector of a Web Server

CAAI emulates two network environments A and B and gathers two traces from a Web server. All the features of a Web server can be described by a feature vector as follows:

$$V = (\beta^A,\ w^A_{s+4} - w^A_{s+1},\ w^A_{s+9} - w^A_{s+1},\ \beta^B,\ w^B_{s+4} - w^B_{s+1},\ w^B_{s+9} - w^B_{s+1},\ loss\_cwnd^B). \tag{2}$$

Features with superscripts ^{A} and ^{B} are for network environments A and B, respectively. The vector element $loss\_cwnd^B$ is mainly used for VEGAS as illustrated in Fig. 3(k) because its maximum congestion window size could not reach even 64 in network environment B. We set $loss\_cwnd^B$ to 0 if the $loss\_cwnd$ in network environment B is below 64, and to 1 otherwise. We can see that a feature vector consists of a total of seven feature vector elements.

VI. CAAI Step 3: Algorithm Classification

This section describes how CAAI identifies (or classifies) the TCP algorithm of a Web server based on its feature vector V. The terms "identification" and "classification" are used interchangeably in the paper. The challenge is that we may get different feature vectors for different Web servers using the same TCP algorithm or for the same Web sever but at different times. This is because the window trace gathered from a Web

server depends on the network condition, especially the instantaneous ACK loss rate on the path from a CAAI computer to the Web server. To solve this problem, we first create a training set that contains the feature vectors of all TCP algorithms in some network conditions (details in Section VII), and then classify the TCP algorithm of a Web server using a machine learning algorithm.

We have compared the performance of several machine learning algorithms including K Nearest Neighbor methods, Decision Tree methods, Artificial Neural Network methods, Naive Bayes methods, Support Vector Machine methods, and Random Forest methods using Weka [54]. Weka is a free and powerful collection of machine learning algorithms for data mining tasks and is written in Java by the University of Weikato. Our Weka results [55] show that random forest consistently achieves the highest classification accuracy among all these methods, and therefore CAAI uses random forest to classify the TCP algorithm of a Web server.

We first explain how random forest [56] grows multiple classification trees, and then describe how random forest classifies the TCP algorithm of a Web server.

Random forest grows multiple classification trees using a training set of feature vectors. Each tree is grown using the bootstrap aggregation (or bagging) method, which uses a random subset of the training set by randomly sampling feature vectors with replacement from the training set. Each node of a tree is split using the random subspace method, which randomly selects K elements from the total of seven elements of a feature vector, and then chooses the element with the best split among these K feature vector elements. There is no pruning when growing a tree.

To classify the TCP algorithm of a Web server, random forest first gets the classification voted by each tree, and then chooses the classification voted by the most trees as the final classification result. Weka also calculates the percentage of the trees voting for the final classification result as a classification confidence level.

Random forest has two important parameters: 1) the total number of trees denoted by $I > 1$, and 2) the number of randomly selected feature vector elements at each node denoted by K. According to random forest [56], we can choose as large I as we want because random forest does not overfit. However, K must be much less than the total number of feature vector elements. We will set the values of these two parameters in Section VII.

VII. CAAI Experiments

In this section, we describe our test-bed and Internet experiment results of CAAI.

A. Testbed Evaluation and Parameter Setting

1) Testbed Setup: We use our lab testbed to collect a training set of feature vectors for CAAI as illustrated in Fig. 9. The testbed consists of four computers: one CAAI computer, one Linux Web server, one Windows Web server, all connected to a Linux router. The Linux Web server runs Apache, and the Windows Web server runs IIS. We run Netem [57] on the Linux router to emulate various network conditions between the CAAI computer and the two Web servers.

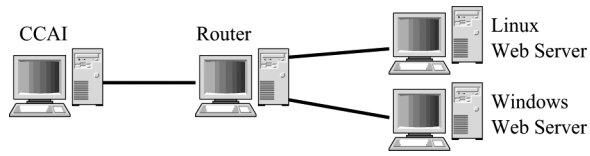


Fig. 9. Our lab testbed.

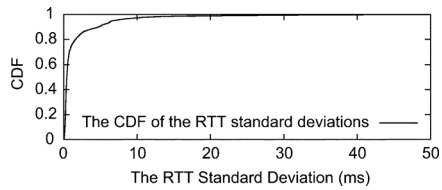


Fig. 10. CDF of the measured RTT standard deviations.

The feature vectors of CTCP′ are obtained using an IIS Web server on Windows Server 2003, and the feature vectors of CTCP″ are obtained using an IIS Web server on Windows Server 2008 (dual boot on the Windows Web server). The feature vectors of CUBIC′ are obtained using an Apache server on Linux kernel 2.6.25, and the feature vectors of all other 11 TCP algorithms are obtained using an Apache server on openSUSE 11.1 with Linux kernel 2.6.27. Note that we use the feature vectors of RENO only on Linux. This is because we have obtained the feature vectors of both Linux RENO and Windows RENO, and they are very similar to each other.

2) Collecting a Training Set of Feature Vectors: To collect a training set of feature vectors of the 14 TCP algorithms, we attempt to emulate realistic Internet network conditions on our testbed. This is because the window trace gathered from a Web server depends on the network condition, especially the instantaneous ACK loss rate on the path from a CAAI computer to the Web server. As a result, we may get different feature vectors for different Web servers using the same TCP algorithm or for the same Web sever but at different times. We first describe how we measure the network conditions between our local CAAI computers and remote Web servers, and then describe the feature vectors collected on the testbed that emulates the measured network conditions.

We describe the network condition between a local CAAI computer and a remote Web server by its average RTT, RTT standard deviation, and packet-loss rate. Since MSS has no impact on the feature vectors, we do not measure the MSS of a network condition. We collected a database of network conditions by measuring 5000 popular Web servers in 2010 and 2011. The RTT information is obtained by "pinging" these Web servers from our local CAAI computers. Ping measures the RTT between a CAAI computer and a Web server for several times and outputs the average and standard deviation of RTTs. The CDF of the average RTTs is shown in Fig. 4, and the CDF of the RTT standard deviations is shown in Fig. 10. The packet-loss rates are obtained from a number of PCAP files. Each PCAP file is generated by TCPdump [58] and contains the packet trace when we download a Web page from a Web server to a CAAI computer. By analyzing the sequence numbers in the packet trace, we can calculate the packet-loss rate that is defined as the ratio between the number of lost packets and the total number of packets (including the lost packets) in a PCAP file. The CDF of the packet-loss rates is shown in Fig. 11.

TABLE III
IDENTIFICATION ACCURACY (IN PERCENTAGE) PER TCP ALGORITHM OF THE TRAINING FEATURE VECTORS

%	RC-small	RENO-big	CTCP'-big	CTCP"-big	BIC	CUBIC'	CUBIC"	HSTCP	HTCP	ILLI.	STCP	VEGA.	VENO	WEST.	YEAH
RC-small	**99.16**	0	0	0	0	0	0	0.17	0	0	0	0	0.5	0.17	0
RENO-big	0	**94**	2	2	0	0	0	0	0	0	0	0	2	0	0
CTCP'-big	0	2	**90**	2.5	0	0	0	1	1	1.5	1	0	0	0	1
CTCP"-big	0	2.5	2.5	**91**	0	0	0	2.5	0	1.5	0	0	0	0	0
BIC	0	0	0	0	**98.5**	0.25	0.25	0	1	0	0	0	0	0	0
CUBIC'	0	0	0	0	0.5	**97.75**	0	0	0.5	0.5	0.5	0	0	0	0.25
CUBIC"	0	0	0	0	0.25	0	**97.75**	0	2	0	0	0	0	0	0
HSTCP	0	0	1.25	2	0	0	0	**96.75**	0	0	0	0	0	0	0
HTCP	0	0	0.5	0.25	0.25	0.25	0.5	0	**97.5**	0.75	0	0	0	0	0
ILLINOIS	0	0	0.25	0	0	0.5	0	0	0.25	**96.25**	0.5	0	2.25	0	0
STCP	0	0	0	0	0	0	0	0	0.5	0.5	**97.25**	0	0	0	2.75
VEGAS	0	0	0	0	0	0	0	0	0	0	0	**100**	0	0	0
VENO	5	1.25	0.25	0.25	0	0	0	0	0	0.75	0	0	**92.5**	0	0
WESTWOOD+	0	0	0	0	0	0	0	0	0	0	0	0	0	**100**	0
YEAH	0.25	0	0.25	0	0	0	0	0	0	1.25	1	0	0	0	**97.25**

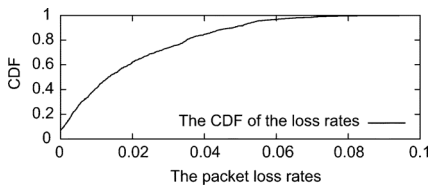


Fig. 11. CDF of the measured packet-loss rates .

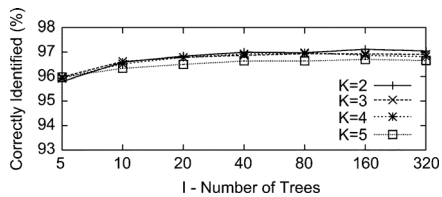


Fig. 12. Percentage of correctly identified feature vectors in the 10-fold cross validation.

When we emulate a random network environment, we randomly select an average RTT, an RTT standard deviation, and a packet-loss rate from our network condition database and set the parameters of Netem accordingly. Specifically, the emulated RTTs for a network condition follow the normal distribution whose average and standard deviation are the given average and standard deviation of RTTs, respectively. For each pair of 14 TCP algorithms and 4 $cwnd\_threshold$ values, we emulate 100 network conditions, each of which randomly selects an average RTT, an RTT standard deviation, and a packet-loss rate. Therefore, there are a total of $14 \times 4 \times 100 = 5600$ feature vectors in the training set.

As illustrated in Fig. 3(o), RENO, CTCP′, and CTCP″ behave very similar to each other when $cwnd\_threshold$ is 64 or 128 packets, and consequently, RENO, CTCP′, and CTCP″ have very similar feature vectors in these cases. This is because CTCP [8] is designed to be very friendly to RENO. Therefore, when $cwnd\_threshold = 64$ or 128, we do not distinguish among RENO, CTCP′, and CTCP″, and we call them together as RC-small. We refer to RENO with $cwnd\_threshold = 256$ or 512 as RENO-big, refer to CTCP′ with $cwnd\_threshold = 256$ or 512 as CTCP′-big, and refer to CTCP″ with $cwnd\_threshold = 256$ or 512 as CTCP″-big.

3) Cross Validation and Parameter Setting: We use the 10-fold cross validation to evaluate the accuracy of the random forest of CAAI. That is, we evenly and randomly divide the total of 5600 feature vectors into 10 groups, and then we run the validation for 10 rounds. In each round, one group is selected for validating random forest and the remaining nine groups are used for training random forest. In all 10 rounds, each group has one and only one chance to be selected for validation. Finally, the accuracy is measured by the percentage of correctly identified feature vectors in all 10 rounds.

Fig. 12 shows the percentage of correctly identified feature vectors, as we vary the two random forest parameters: I that is the number of trees, and K that is the number of randomly selected feature vector elements from the total of seven feature vector elements at each node of a tree. We can see that the accuracy improves as parameter I increases, and when $I >= 40$, the accuracy remains almost the same. Therefore, CAAI sets parameter I to 80. That is, CAAI grows 80 trees in random forest. We can also see that the accuracy does not change much as parameter K changes, except for very big K values such as 5. Therefore, CAAI sets parameter K to 4, which is the default value of the Weka random forest parameter K (specifically by default Weka sets $K = \lfloor log_2(7+1) \rfloor + 1 = 4$).

Now we take a deeper look at the identification accuracy. When CAAI sets parameter I to 80 and parameter K to 4, the overall identification accuracy of the 10-fold cross validation is 96.98%. Table III shows the individual identification accuracy of each TCP algorithm (called the confusion matrix). A value at row X and column Y represents the percentage of the feature vectors of TCP algorithm X being classified as TCP algorithm Y. For example, among all CUBIC″ feature vectors, 97.75% are correctly identified as CUBIC″, but 0.25% and 2% are mistakenly identified as BIC and HTCP, respectively. As another example, among all RC-small feature vectors that contain all RENO, CTCP′, and CTCP″ with $cwnd\_threshold = 64$ or 128 feature vectors, 99.16% are correctly identified as RC-small, but 0.17%, 0.5%, and 0.17% are mistakenly identified as HSTCP, VENO, and WESTWOOD+, respectively. We can see that random forest can reasonably accurately identify each individual TCP algorithm.

B. Internet Measurement

We used CAAI to identify the TCP algorithms of 63 124 popular Web servers (using the Alexa traffic rank [59]) in Spring and Summer 2011. For a Web server, we first run our Web page searching tool (described in Section IV-E) on a PlanetLab node to find a long Web page on the Web server, and then run CAAI on our lab computers to download the Web page and to identify

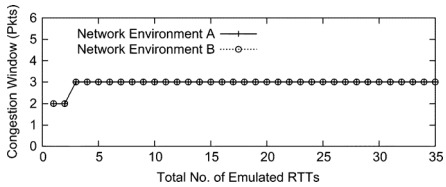


Fig. 13. Invalid trace without any timeout. Because the congestion window size of the Web server is always below $cwnd\_threshold$.

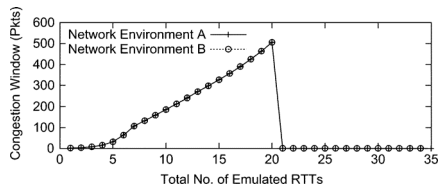


Fig. 14. Valid trace with "Remaining at 1 Packet." That is, the window size remains at 1 packet after the timeout.

the TCP algorithm of the Web server. If a Web server has multiple IP addresses, we only test one of them. A short message is added into the header of every HTTP request message to indicate our contact information and the research purpose of our experiments.

Our lab computers that we used to run CAAI are connected to the Gigabit campus network that is then connected to the Internet through a 10-Gb/s link. Note that the lab computers we used to measure the network conditions for Figs. 4, 10, and 11 are the same as the lab computers that we used to run the CAAI experiments of the Web servers. This eliminates the potential identification inaccuracy (if any) caused by location-dependent network condition measurements.

1) Web Server Information: In this section, we provide some basic information of the Web servers measured in our experiments.

The Web servers exhibit good geographical diversity and are located all around the world. Among all the Web servers, 0.54% are located in Africa, 21.46% are located in Asia, 0.83% are located in Australia, 43.28% are located in Europe, 31.92% are located in North America, and 1.97% are located in South America.

From the headers of HTTP response messages, we obtain the information of Web server software. Among all the Web servers, 70.20% use Apache, 11.13% use IIS, 12.85% use Ngnix, 1.36% use LiteSpeed, and 4.46% use other software.

Note that if a Web server uses a TCP proxy (such as a load balancer) that splits the end-to-end TCP connection between CAAI and the Web server, then the TCP algorithm identified by CAAI is actually the TCP algorithm used by the TCP proxy instead of the Web server. This is one possible reason that for about 15% of IIS Web servers in our experiments, the TCP algorithms identified by CAAI are not RENO, CTCP′, or CTCP″.

2) Web Servers With Invalid Traces: For 53% of the Web servers (i.e., about 33 000 Web servers), CAAI could not gather valid window traces (i.e., 18 RTTs of window sizes after a timeout, as described in Section IV-E) even with $cwnd\_threshold = 64$ packets. The major reasons are: 1) CAAI could not find a sufficiently long Web page on a Web server; and 2) a Web server accepts only one HTTP request or very few repeated HTTP requests in the same TCP connection. Intuitively, the file transfer of these Web servers is mainly controlled by the TCP slow start algorithm, and thus it is not necessary to identify their congestion avoidance algorithms. Some other reasons are: 1) the congestion window size of a Web server is always below $cwnd\_threshold$, and thus CAAI does not emulate a timeout. Fig. 13 shows such an example; 2) the congestion window size of a Web server reaches $cwnd\_threshold$, but somehow the Web server does not respond to the emulated timeout.

3) Web Servers With Valid Traces: For 47% of the Web servers (i.e., about 30 000 Web servers), CAAI successfully gathered valid traces as summarized in Table IV. As described in Section IV-B, CAAI starts with $cwnd\_threshold = 512$ packets. If not successful, CAAI tries $cwnd\_threshold = 256$, 128, and finally 64 packets. Each column of Table IV shows the information of the Web servers gathered with a $cwnd\_threshold$, and the last column shows the overall information. We can see that for 63.84%, 14.02%, 14.24%, and 7.92% of the Web servers with valid traces, CAAI successfully gathered window traces with $cwnd\_threshold = 512$, 256, 128, and 64 packets, respectively. In the remaining part of this section, we consider only the Web servers with valid traces (i.e., about 30 000 Web servers), and the percentage is calculated with respect to these Web servers.

TABLE IV
IDENTIFICATION RESULTS (IN PERCENTAGE) OF WEB SERVERS

$cwnd\_threshold$	512	256	128	64	Total
Total	63.84	14.02	14.24	7.92	100
RENO	2.46	0.85	6.69	4.47	3.31∼14.47
CTCP'	11.26	2.15			13.41∼24.57
CTCP"	0.82	0.27			1.09∼12.25
BIC	11.9	5.51	2.09	0.95	20.45
CUBIC'	11.71	0.67	0.3	0.13	12.81
CUBIC"	12.05	1.19	0.24	0.18	13.66
HSTCP	1.16	0.21	0.09	0.11	1.57
HTCP	4.14	0.43	0.25	0.07	4.89
ILLINOIS	0.23	0.32	0.16	0.13	0.83
STCP	0.47	0.08	0.28	0.07	0.9
VEGAS	0.09	0.04	0.02	0.04	0.19
VENO	0.58	0.36	0.18	0.33	1.45
WESTWOOD+	0.36	0.16	0.04	0.02	0.58
YEAH	0.59	0.23	0.32	0.6	1.74
Unsure TCP	3.41	0.47	0.28	0.16	4.32
Remaining at 1 Packet	0.51	0.15	0.43	0.21	1.31
Nonincreasing Window	0.6	0.47	0.43	0.26	1.75
Approaching $loss\_cwnd$	0.41	0.08	2.31	0	2.81
Bounded Window	1.1	0.35	0.12	0.18	1.76

After manually checking these valid traces, we notice that there are four special cases of valid traces measured in the Internet experiments, which we do not observe in our testbed experiments.

1) After the timeout, the window size of a Web server remains at 1 packet for a very long time. We call this case "Remaining at 1 Packet" in Table IV, and Fig. 14 shows such an example.
2) In the congestion avoidance state, the window size of a Web server never increases (i.e., always less than or equal to w_s). We call this case "Nonincreasing Window." Fig. 15 shows such an example.

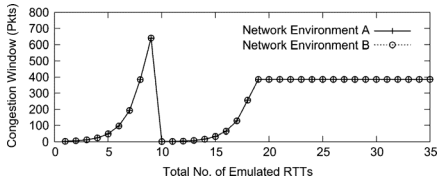


Fig. 15. Valid trace with "Nonincreasing Window." That is, the window size never increases in the congestion avoidance state.

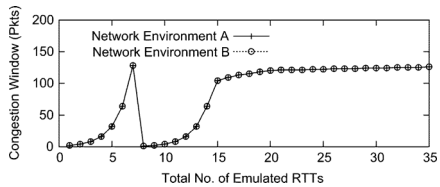


Fig. 16. Valid trace with "Approaching $loss\_cwnd$." That is, the window size increases and slowly approaches $loss\_cwnd$.

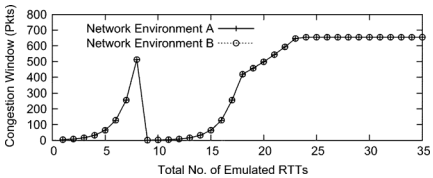


Fig. 17. Valid trace with "Bounded Window." That is, the window size is bounded by some upper bound.

3) In the congestion avoidance state, the window size of a Web server initially increases quickly, and then increases slowly as it approaches to $loss\_cwnd$. We call this case "Approaching $loss\_cwnd$." Fig. 16 shows such an example.
4) In the congestion avoidance state, the window size of a Web server increases beyond $loss\_cwnd$, and then is bounded by some upper bound. One possible reason is that the window size is bounded by some factors, such as the TCP send buffer size of a Web server. We call this case "Bounded Window." Fig. 17 shows such an example.

Except these special valid traces, CAAI uses the Weka implementation of random forest to classify the rest of the valid traces. In addition to the final classification result, Weka also outputs a confidence level that is the percentage of the trees voting for the final classification result among all random forest trees. CAAI does not report the classification result of random forest if the confidence level is lower than 40% (that is, less than 40% of all random forest trees vote for the final classification result). Table IV shows that 4.32% of Web servers are labeled as "Unsure TCP" because their confidence levels are lower than 40%. Fig. 18 shows such an example. There are some possible reasons for the low confidence levels. First, a Web server uses some TCP algorithm different from all 14 TCP algorithms considered in this paper. Second, the trace of a Web server is not smooth possibly due to some network and Web server factors, such as a very high packet-loss rate or a very busy Web server.

We can see that *only* $3.31\% \sim 14.47\%$ *of Web servers still use the traditional RENO*. The reason for the range is that we do not distinguish among RENO, CTCP′, and CTCP″ when $cwnd\_threshold = 64$ or 128 (called RC-small in Section VII-A). Specifically, $2.46 + 0.85 = 3.31\%$ of Web servers use RENO when $cwnd\_threshold = 512$ or 256 (called RENO-big in Section VII-A), and $6.69 + 4.47 = 11.16\%$ of

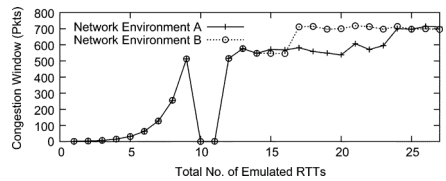


Fig. 18. Valid trace with "Unsure TCP." That is, CAAI could not identify its TCP algorithm due to the low confidence level of random forest.

Web servers use RC-small when $cwnd\_threshold = 128$ or 64. Thus, at least 3.31% of Web servers and at most $3.31 + 11.16 = 14.47\%$ of Web servers still use the traditional RENO.

We can see that $13.41\% \sim 24.57\%$ of Web servers use CTCP′ (the early version of CTCP), and only $1.09\% \sim 12.25\%$ of Web servers use CTCP″ (the later version of CTCP). *Overall,* $14.5\% \sim 25.66\%$ *of Web servers use the Windows default TCP algorithms—CTCP (including both CTCP′ and CTCP″).* We can also see that *a significant percentage (i.e.,* $20.45 + 12.81 + 13.66 = 46.92\%$*) of Web servers use the Linux default TCP algorithms—BIC/CUBIC (CUBIC includes CUBIC′ and CUBIC″), and among them, BIC (developed earlier than CUBIC′ and CUBIC″) has the largest number of Web servers*.

Table IV shows that a nontrivial percentage of Web servers use those non-default TCP algorithms (i.e., TCP algorithms other than RENO, BIC, CUBIC, and CTCP). For example, 4.89% of Web servers use HTCP. While some of them may be identification errors, we find that there are indeed some Web servers using non-default TCP algorithms. One possible reason for the Web servers to use HTCP is that HTCP is recommended by some Linux TCP tuning Web sites [60], [61] for high-speed data transfer.

C. Discussion

Our CAAI measurement results show a strong sign that the majority of TCP flows are not controlled by RENO anymore, and a strong sign that Internet congestion control has already changed from homogeneous to heterogeneous.

When evaluating a new congestion control algorithm, it is important to study whether the new algorithm can achieve good performance when competing with not only the traditional RENO, but also other new dominating algorithms such as BIC, CUBIC, and CTCP. This is especially important for delay-based congestion control algorithms because a delay-based algorithm whose parameters are tuned for competing with RENO may achieve poor performance when competing with more aggressive algorithms such as BIC and CUBIC.

When designing a traffic generator to generate realistic Internet traffic, it is important to consider not only traditional traffic characteristics such as packet sizes, packet interarrival times, RTTs, connection durations, connection establishment rates, but also new traffic characteristics such as congestion control algorithms.

When determining the sizes of router buffers, it is important to consider the distribution of TCP congestion control algorithms. The router buffer sizes are usually determined by assuming that all TCP flows are controlled by RENO, and for instance the well-known rule-of-thumb that sets the buffer size of a link to its bandwidth delay product directly comes from the multiplicative decrease parameter of RENO [11]. Because quite a few TCP congestion control algorithms, such as BIC, CUBIC,

HSTCP, HTCP, and STCP, use multiplicative decrease parameters different from that of RENO, the rule-of-thumb may not hold true.

Finally, note that this paper focuses on the congestion avoidance component of TCP algorithms, and this is because Internet traffic heavily depends on the congestion avoidance component. A large portion of Internet flows such as Web flows and peer-to-peer (P2P) flows are mice flows that have short sizes and thus mainly depend on the slow start component of a TCP algorithm, whereas a small portion of Internet flows are elephant flows that have long sizes and thus mainly depend on the congestion avoidance component of a TCP algorithm. However, because of the heavy-tailed distribution of Internet flow sizes, a significant portion of Internet traffic [62] including Web and P2P traffic is generated by elephant flows, and thus heavily depends on the congestion avoidance component.

VIII. Conclusion

In this paper, we proposed a tool called CAAI for identifying the TCP algorithm of a remote Web server and presented our measurement results of the TCP deployment information of about 30 000 Web servers.

There are some limitations of the current work. The current CAAI does not consider some other TCP congestion control algorithms, such as FAST [63], which is not available in any operating system but has been used by some Web servers, and does not consider XCP [64], VCP [65], and PERT [66], which have recently been proposed but not yet incorporated into any operating system. We plan to add the feature vectors of more operating systems (e.g., FreeBSD, OpenBSD, Mac OS X, and Solaris) into our training set, so that we can more accurately identify their TCP algorithms.

References

[1] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu, "TCP congestion avoidance algorithm identification," in *Proc. IEEE ICDCS*, Minneapolis, MN, USA, Jun. 2011, pp. 310–321.

[2] V. Jacobson, "Congestion avoidance and control," in *Proc. ACM SIGCOMM*, Stanford, CA, USA, Aug. 1988, pp. 314–326.

[3] D. Chiu and R. Jain, "Analysis of the increase/decrease algorithms for congestion avoidance in computer networks," *J. Comput. Netw. ISDN*, vol. 17, no. 1, pp. 1–14, Jun. 1989.

[4] A. Tang, J. Wang, S. Low, and M. Chiang, "Equilibrium of heterogeneous congestion control: Existence and uniqueness," *IEEE/ACM Trans. Netw.*, vol. 15, no. 4, pp. 824–837, Aug. 2007.

[5] K. Munir, M. Welzl, and D. Damjanovic, "Linux beats Windows!—Or the worrying evolution of TCP in common operating systems," in *Proc. PFLDNet*, Marina Del Rey, CA, USA, Feb. 2007, pp. 1–6.

[6] M. Weigle, P. Sharma, and J. Freeman, "Performance of competing high-speed TCP flows," in *Proc. NETWORKING*, Coimbra, Portugal, May 2006, pp. 476–487.

[7] I. Rhee and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," in *Proc. PFLDNet*, Feb. 2005, pp. 1–6.

[8] K. Tan, J. Song, Q. Zhang, and M. Sridharan, "A compound TCP approach for high-speed and long distance networks," in *Proc. IEEE INFOCOM*, Barcelona, Spain, Apr. 2006, pp. 1–12.

[9] E. Kohler, M. Handley, and S. Floyd, "Datagram congestion control protocol (DCCP)," RFC 4340, Mar. 2006.

[10] R. Stewart, "Stream control transmission protocol," RFC 4960, Sep. 2007.

[11] G. Appenzeller, I. Keslassy, and N. McKeown, "Sizing router buffers," in *Proc. ACM SIGCOMM*, Portland, OR, USA, Aug. 2004, pp. 281–292.

[12] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion (BIC) control for fast long-distance networks," in *Proc. IEEE INFOCOM*, Hong Kong, Mar. 2004, pp. 2514–2524.

[13] S. Ha, I. Rhee, and L. Xu, "CUBIC: A new TCP-friendly high-speed TCP variant," *Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, Jul. 2008.

[14] S. Floyd, "Highspeed TCP for large congestion windows," RFC 3649, Dec. 2003.

[15] R. N. Shorten and D. J. Leith, "H-TCP: TCP for high-speed and long-distance networks," in *Proc. PFLDNet*, Argonne, IL, Feb. 2004, pp. 1–6.

[16] C. Caini and R. Firrincieli, "TCP-Hybla: A TCP enhancement for heterogeneous networks," *Int. J. Satell. Commun. Netw.*, vol. 22, no. 5, pp. 547–566, Sep. 2004.

[17] S. Liu, T. Basar, and R. Srikant, "TCP-Illinois: A loss and delay-based congestion control algorithm for high-speed networks," in *Proc. VALUETOOLS*, Pisa, Italy, Oct. 2006, Art. no. 55.

[18] A. Kuzmanovic and E. W. Knightly, "TCP-LP: A distributed algorithm for low priority data transfer," in *Proc. IEEE INFOCOM*, San Francisco, CA, USA, Apr. 2003, pp. 1691–1701.

[19] T. Kelly, "Scalable TCP: Improving performance in highspeed wide area networks," *Comput. Commun. Rev.*, vol. 33, no. 2, pp. 83–91, Apr. 2003.

[20] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New techniques for congestion detection and avoidance," *Proc. ACM SIGCOMM*, pp. 24–35, Aug. 1994.

[21] C. Fu and S. Liew, "TCP Veno: TCP enhancement for transmission over wireless access networks," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 2, pp. 216–228, Feb. 2003.

[22] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang, "TCP Westwood: Bandwidth estimation for enhanced transport over wireless links," in *Proc. ACM MobiCom*, Rome, Italy, Jul. 2001, pp. 287–297.

[23] A. Baiocchi, A. Castellani, and F. Vacirca, "YeAH-TCP: Yet another highspeed TCP," in *Proc. PFLDNET*, Los Angeles, CA, USA, Feb. 2007, pp. 1–6.

[24] Ipoque, Leipzig, Germany, "Internet study 2008/2009," 2009 [Online]. Available: http://www.ipoque.com/en/resources/internet-studies

[25] Sandvine, Inc., Waterloo, ON, Canada, "Global Internet phenomena report," 2012 [Online]. Available: http://www.sandvine.com/news/global_broadband_trends.asp

[26] M. Allman, V. Paxson, and W. Stevens, "TCP congestion control," RFC 2581, Apr. 1999.

[27] M. Allman, S. Floyd, and C. Partridge, "Increasing TCP's initial window," RFC 3390, Oct. 2002.

[28] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, "An argument for increasing TCP's initial congestion window," *Comput. Commun. Rev.*, vol. 40, no. 3, pp. 27–33, Jul. 2010.

[29] S. Floyd, "Limited slow-start for TCP with large congestion windows," RFC 3742, Mar. 2004.

[30] S. Ha and I. Rhee, "Hybrid slow start for high-bandwidth and long-distance networks," in *Proc. PFLDNET*, Manchester, U.K., Mar. 2008, pp. 1–6.

[31] M. Allman, V. Paxson, and E. Blanton, "TCP congestion control," RFC 5681, Sep. 2009.

[32] S. Floyd, T. Henderson, and A. Gurtov, "The newreno modification to TCP's fast recovery algorithm," RFC 3782, Apr. 2004.

[33] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," RFC 2018, Oct. 1996.

[34] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An extension to the selective acknowledgement (SACK) option for TCP," RFC 2883, Jul. 2000.

[35] J. Padhye and S. Floyd, "On inferring TCP behavior," in *Proc. ACM SIGCOMM*, San Diego, CA, USA, Aug. 2001, pp. 287–298.

[36] J. Oshio, S. Ata, and I. Oka, "Identification of different TCP versions based on cluster analysis," in *Proc. IEEE ICCCN*, San Francisco, CA, USA, Aug. 2009, pp. 1–6.

[37] S. Feyzabadi and J. Schonwalder, "Identifying TCP congestion control algorithms using active probing," presented at the PAM, Zurich, Switzerland, Apr. 2010, Poster.

[38] P. Yang, W. Luo, and L. Xu, "Towards measuring the deployment information of different TCP congestion control algorithms: The multiplicative decrease parameter," in *Proc. IEEE GLOBECOM*, Miami, FL, USA, Dec. 2010, pp. 1–5.

[39] D. Comer and J. Lin, "Probing TCP implementations," in *Proc. USENIX Summer Conf.*, Boston, MA, USA, Jun. 1994, p. 17.

[40] A. Medina, M. Allman, and S. Floyd, "Measuring the evolution of transport protocols in the Internet," *Comput. Commun. Rev.*, vol. 35, no. 2, pp. 37–52, Apr. 2005.

[41] "Network Mapper (NMAP)," [Online]. Available: http://nmap.org/

[42] V. Paxson, "Automated packet trace analysis of TCP implementations," in *Proc. ACM SIGCOMM*, Cannes, France, Sep. 1997, pp. 167–179.

[43] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of Internet flow rates," in *Proc. ACM SIGCOMM*, Pittsburgh, PA, USA, Aug. 2002, pp. 309–322.
[44] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley, "Inferring TCP connection characteristics through passive measurements," in *Proc. IEEE INFOCOM*, Hong Kong, Mar. 2004, pp. 1582–1592.
[45] S. Rewaskar, J. Kaur, and F. Smith, "A performance study of loss detection/recovery in real-world TCP implementations," in *Proc. IEEE ICNP*, Beijing, China, Oct. 2007, pp. 256–265.
[46] F. Qian, A. Gerber, Z. Mao, S. Sen, O. Spatscheck, and W. Willinger, "TCP revisited: a fresh look at TCP in the wild," in *Proc. ACM IMC*, Chicago, IL, USA, Nov. 2009, pp. 76–89.
[47] N. Seddigh and M. Devetsikiotis, "Studies of TCP's retransmission timeout mechanism," in *Proc. IEEE ICC*, Helsinki, Finland, Jun. 2001, pp. 1834–1840.
[48] P. Sarolahti and A. Kuznetsov, "Congestion control in Linux TCP," in *Proc. FREENIX Track: 2002 USENIX Ann. Tech. Conf.*, Berkeley, CA, USA, Jun. 2002, pp. 49–62.
[49] D. Wei, S. Hegdesan, and S. Low, "A burstiness control for TCP," in *Proc. PFLDNet*, Feb. 2005, pp. 1–6.
[50] P. Sarolahti, M. Kojo, K. Yamamoto, and M. Hata, "Forward RTO-recovery (F-RTO): An algorithm for detecting spurious retransmission timeouts with TCP," RFC 5682, Sep. 2009.
[51] "HTTrack website copier," 2013 [Online]. Available: http://www.httrack.com/
[52] PlanetLab, "An open platform for developing, deploying, and accessing planetary-scale service," 2002 [Online]. Available: http://www.planet-lab.org/
[53] H. Perros, "Computer simulation techniques—The definitive introduction," 2003 [Online]. Available: http://www.csc.ncsu.edu/faculty/perros/simulation.pdf
[54] The University of Waikato, Hamilton, New Zealand, "Weka 3: Data mining software in Java," 2013 [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/
[55] J. Shao, "Identification of TCP algorithms," Master thesis, Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE, USA, Dec. 2012.
[56] L. Breiman, "Random forests," *Mach. Learning*, vol. 45, no. 1, pp. 5–32, 2001.
[57] S. Hemminger, "Network emulation with NetEm," in *Proc. 6th Australia's Nat. Linux Conf.*, Apr. 2005, pp. 1–9.
[58] "Tcpdump," 2010 [Online]. Available: http://www.tcpdump.org/
[59] Alexa Internet, Inc., "Top 500 global sites," 2013 [Online]. Available: http://www.alexa.com/topsites
[60] Department of Energy, Energy Sciences Network, Berkeley, CA, USA, "Linux tuning," 2012 [Online]. Available: http://fasterdata.es.net/host-tuning/linux/
[61] TestMy Net, LLC, "How to setup a speed test server," 2013 [Online]. Available: http://testmy.net/make-a-speed-test
[62] N. Basher, A. Mahanti, A. Mahanti, C. Williamson, and M. Arlitt, "A comparative analysis of Web and peer-to-peer traffic," in *Proc. WWW*, Beijing, China, Apr. 2008, pp. 287–296.
[63] C. Jin, D. X. Wei, and S. H. Low, "Fast TCP: motivation, architecture, algorithms, performance," in *Proc. IEEE INFOCOM*, Hong Kong, Mar. 2004, pp. 2490–2501.
[64] D. Katabi, M. Handley, and C. Rohrs, "Congestion control for high bandwidth-delay product networks," in *Proc. ACM SIGCOMM*, Pittsburgh, PA, USA, Aug. 2002, pp. 89–102.
[65] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman, "One more bit is enough," in *Proc. ACM SIGCOMM*, Philadelphia, PA, USA, Aug. 2005, pp. 37–48.
[66] S. Bhandarkar, A. Reddy, Y. Zhang, and D. Loguinov, "Emulating AQM from end hosts," in *Proc. ACM SIGCOMM*, Kyoto, Japan, Aug. 2007, pp. 349–360.

Peng Yang (M'11) received the Bachelor's degree in computer science and technology and Master's degree in computer application technology from Central China Normal University, Wuhan, China, in 2004 and 2007, respectively, and is currently pursuing the Doctoral degree in computer science at the University of Nebraska–Lincoln, Lincoln, NE, USA.

His research interests include wireless communication, transmission control protocols, utility-based rate control, and peer-to-peer networks.

Juan Shao received the Bachelor degree in computer science from An Hui University, Hefei, China, in 1997, and the Master's degree in computer science from the University of Nebraska–Lincoln, Lincoln, NE, USA, in 2012.

She joined the Holland Computing Center, University of Nebraska–Lincoln, Lincoln, NE, USA, in 2013. Her research interests are in the areas of network Qos, TCP/IP protocols, and high-performance networking.

Wen Luo received the Bachelor of Science degree in mathematics and computer science double major with honors from the University of Nebraska–Lincoln, Lincoln, NE, USA, and is currently pursuing the master's degree in financial engineering at Cornell University, Ithaca, NY, USA.

He worked with Dr. Lisong Xu's team in his senior year at the University of Nebraska–Lincoln for undergraduate academic research experience program.

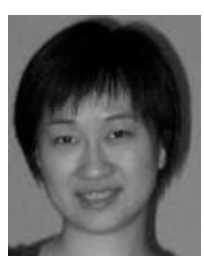
Lisong Xu (M'04) received the B.E. and M.E. degrees from the University of Science and Technology Beijing, Beijing, China, in 1994 and 1997, respectively, and the Ph.D. degree from North Carolina State University, Raleigh, NC, USA, in 2002, all in computer science.

He joined the University of Nebraska–Lincoln (UNL), Lincoln, NE, USA, in 2004 and is currently an Associate Professor in computer science and engineering.

Dr. Xu is a recipient of the NSF CAREER Award in 2007 and the UNL CSE Student Choice Outstanding Teaching Award in 2006, 2007, 2008, 2010, and 2011.

Jitender Deogun (A'86–M'05) received the B.S. (Hons.) degree from Punjab University, Chandigarh, India, in 1967, the M.Sc. degree from Delhi University, New Delhi, India, and the M.S. and Ph.D. degrees from the University of Illinois at Urbana–Champaign, Urbana, IL, USA, in 1974 and 1979, respectively.

He is a Full Professor of computer science and engineering with the University of Nebraska–Lincoln, Lincoln, NE, USA. His research interests include optical networking, optical switch design, data center architectures, ontologies for mental health, bioinformatics, design and analysis of algorithms, and structural and algorithmic graph theory.



Ying Lu (M'05) received the B.S. degree from the Southwest Jiaotong University, Chengdu, China, in 1996, and the M.CS. and Ph.D. degrees in from the University of Virginia, Charlottesville, VA, USA, in 2001 and 2005, respectively, all in computer science.

Since 2005, she has been with the University of Nebraska–Lincoln, Lincoln, NE, USA, where she is an Associate Professor with the Department of Computer Science and Engineering. Her research interests include cluster and cloud computing, real-time systems, autonomic computing, and Web systems. She has done significant work on feedback control of computing systems.