Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computer Communications xx (2005) 1–14
computer communications
www.elsevier.com/locate/comcom

Delay-based early congestion detection and adaptation in TCP: impact on web performance

Michele C. Weigle[a,*], Kevin Jeffay[b], F. Donelson Smith[b]

[a]Department of Computer Science, Clemson University, Clemson, SC 29634, USA
[b]Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, USA

Received 13 January 2004; revised 13 November 2004; accepted 24 November 2004

Abstract

Concerns over the scalability of TCP's end-to-end approach to congestion control and its AIMD congestion adaptation have led to proposals for router-based congestion control, specifically, active queue management (AQM). In this paper we present an end-to-end alternative to AQM—a new congestion detection and reaction mechanism for TCP based on measurements of one-way transit times of TCP segments within a TCP connection. Our design, called Sync-TCP, places timestamps in TCP headers, measures variation in one-way transit times, and uses these measurements as a form of early congestion notification. We demonstrate empirically that: (1) Sync-TCP provides better throughput and HTTP response-time performance than TCP Reno, (2) Sync-TCP provides better early congestion detection and reaction than the Adaptive Random Early Detection with Explicit Congestion Notification AQM mechanism, (3) Sync-TCP's congestion detection and adaptation mechanisms are robust against clock drift, (4) Sync-TCP is an incrementally deployable protocol—Sync-TCP connections can co-exist with TCP Reno connections in a network, and (5) the performance of TCP Reno connections are improved with the addition of even a small percentage of Sync-TCP connections.

Keywords: Congestion control; Simulations

1. Introduction

Since its inception, congestion control on the Internet has been the responsibility of end-system hosts. In the current Internet, TCP implementations detect packet losses and interpret them as indicators of congestion. The canonical (and indeed required [1]) response to these congestion indications is for end-systems to 'back-off' by reducing the rate at which they are currently transmitting data. The specific response invoked by present-day TCP, TCP Reno, reduces the amount of data a connection can have unacknowledged in the network by at least a factor of two. The connection then slowly probes for available bandwidth by increasing the amount of unacknowledged data each round-trip time (RTT). This congestion detection mechanism and the 'additive increase, multiplicative decrease' (AIMD[1]) congestion reaction eliminated congestion collapse events on the Internet and has enabled the growth of the Internet to its current size.

However, despite the success of TCP Reno, concerns have been raised about the future of pure end-to-end approaches to congestion avoidance [2]. In response to these concerns, router-based congestion control mechanisms such as active queue management (AQM) have been developed and proposed for deployment [3–6]. With a router in the congestion control loop, and specifically with the use of AQM, it is now possible for end-systems to receive a signal of incipient congestion prior to the actual occurrence of congestion. The signal can be an implicit signal realized

* Corresponding author. Tel.: +1 864 656 6753; fax: +1 864 656 0145.
E-mail address: mweigle@cs.clemson.edu (M.C. Weigle).

0140-3664/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.comcom.2004.11.011

[1] Additive increase is defined as $w(t+1)=\alpha+w(t)$, where $w(t)$ is the size of the current congestion window in segments at time t and the time unit is one RTT. During TCP Reno congestion avoidance, $\alpha=1$. For every ACK received, the congestion window is increased by $1/w(t)$, which results in an increase of at most one segment in one RTT. Multiplicative decrease is defined as $w(t+1)=\beta w(t)$. In TCP Reno, $\beta=0.5$.

by a router dropping a packet from a connection even though resources exist in the router to enqueue and forward the packet, or an explicit signal realized by setting a congestion notification bit in the packet's header (for those connections that have previously indicated that they are able to interpret this bit). All proposals for router-based congestion control tie the generation of the congestion signal to a decision procedure based on measures of the length of the queue in the router. Numerous mechanisms have been developed to determine when a router should signal congestion, but all require information on the current size of the queue.

This paper presents a new congestion control design that seeks to provide the benefits of the existing AIMD adaptation mechanism while also providing an end-to-end-only form of early congestion detection. The new design, called Sync-TCP, works by placing timestamps in the TCP header and using these timestamps to monitor one-way transit times (OTTs) of TCP segments. A congestion detection and adaptation algorithm based on observations of variations in OTT within a TCP connection is developed and is shown empirically to both provide better throughput and HTTP response-time performance than TCP Reno and better early congestion detection and adaptation than a proposed AQM mechanism, Adaptive Random Early Detection (ARED) with Explicit Congestion Notification (ECN). Moreover, we show that Sync-TCP connections can co-exist with TCP Reno connections in a network (i.e. that Sync-TCP is an incrementally deployable protocol) and that the performance of TCP Reno connections are improved with even the addition of a small percentage of Sync-TCP connections. This evaluation is performed in the context of HTTP traffic, which consists of a large number of short-lived flows and a small number of long-lived flows. Any end-to-end congestion control mechanism will have little direct effect on the short-lived flows, because the large majority of these flows will have completed before any congestion information is relayed back to the sender. In this way, HTTP traffic is almost a worst-case for an end-to-end congestion control mechanism since it can only affect a small portion of the traffic. The case we make here is that Sync-TCP is effective in improving the performance of web traffic as a whole while only directly affecting the behavior of a few flows.

The remainder of this paper is organized as follows. Section 2 presents related work in congestion control modifications proposed for TCP and in router-based congestion control and early congestion notification. Section 3 presents the design of Sync-TCP. Section 4 presents the methodology we use to evaluate Sync-TCP, and Section 5 presents our experimental results.

2. Related work

There have been two main approaches to detecting congestion before router buffers overflow: end-to-end methods and router-based mechanisms. End-to-end methods are focused on making changes to TCP Reno. Most of these approaches try to detect and react to congestion earlier than TCP Reno (i.e. before segment loss occurs) by monitoring the network using end-to-end measurements. Router-based mechanisms, such as AQM, make changes to the routers so that they notify senders when congestion is occurring but before packets are dropped. AQM, in theory, gives the best performance because congested routers are in the best position to know when congestion is occurring. Drawbacks to using AQM methods include the complexity involved in configuring the various parameters [7] and the need to change routers in the network (and possibly the end system).

2.1. End-to-end approaches

Several end-to-end congestion control algorithms have been proposed as alternatives to TCP Reno. Delay-based approaches, such as TCP Vegas and TCP Santa Cruz attempt to use network measurements to detect congestion before packet loss occurs. FAST TCP is a variation on TCP Vegas with the goal of utilizing the extra capacity available in large high-speed links.

2.1.1. TCP Vegas

To address the sometimes large number of segment losses with TCP Reno, Brakmo et al. [8] proposed several modifications to the congestion control algorithm. The resulting protocol, TCP Vegas, included three main modifications: a fine-grained RTT timer used to detect segment loss, congestion avoidance using expected throughput, and congestion control during TCP's slow start phase. TCP Vegas uses a decrease in a connection's throughput as an early indication of network congestion. The basis of TCP Vegas' congestion avoidance algorithm is to keep just the right amount of 'extra data' in the network. TCP Vegas defines extra data as the data that would not have been sent if the connection's sending rate exactly matched the bandwidth available to it in the network. This extra data will be queued at the bottleneck link. If there is too much extra data, network queues could grow and the connection itself could become a cause of congestion. If there is not enough extra data in the network, the connection does not receive feedback (in the form of RTT estimates provided by the return of ACKs) quickly enough to adjust to congestion. TCP Vegas relies on RTT estimates to detect congestion. This could cause the sender to reduce its sending rate when there was congestion on the ACK path (thus, delaying ACKs and increasing the RTT) rather than on the data path.

2.1.2. TCP Santa Cruz

TCP Santa Cruz [9] makes changes to TCP Reno's congestion avoidance and error recovery mechanisms. The congestion avoidance algorithm in TCP Santa Cruz uses changes in delay, in addition to segment loss, to detect

congestion. Modifications to TCP Reno's error recovery mechanisms utilize a TCP SACK-like *ACK Window* to more efficiently retransmit lost segments. (TCP with selective acknowledgments (SACK) adds information to ACKs about contiguous blocks of correctly received data so that senders can better identify lost segments.) TCP Santa Cruz also includes changes to the RTT estimate and changes to the retransmission policy of waiting for three duplicate ACKs before retransmitting. The congestion detection and reaction mechanisms in TCP Santa Cruz are much like that of TCP Vegas, except that congestion detection uses change in forward delay instead of change in throughput (which relies on RTTs) to detect congestion.

2.1.3. FAST TCP

FAST TCP [10] is focused on improving performance of flows on large, high-bandwidth links. It shares the goal of keeping a certain number of packets in the network with TCP Vegas. FAST TCP uses changes in estimated queuing delays as a sign of congestion, along with packet losses, but bases the queuing delays on changes in RTTs. This has the same problem as TCP Vegas of not being able to distinguish between congestion on the data path vs congestion on the ACK path. Both Sync-TCP and FAST TCP use queuing delays as a form of multi-bit congestion feedback. In both algorithms, the change to the congestion window depends on the degree of congestion detected.

2.2. AQM strategies

Internet routers today employ traditional FIFO queuing (called drop-tail queue management). Active queue management (AQM) is a class of router-based congestion control mechanisms where a router monitors its queue size and makes decisions on how to admit packets to the queue. Traditional routers use a drop-tail policy, where packets are admitted whenever the queue is not full. AQM routers potentially drop packets before the queue is full. This action is based on the fundamental assumption that most of the traffic in the network employs a congestion control mechanism, such as TCP Reno, where the sending rate is reduced when packets are dropped. AQM algorithms are designed to maintain a relatively small average queue but allow short bursts of packets to be enqueued without dropping them. In order to keep a small queue, packets are dropped early, i.e. before the queue is full. A small queue results in lower delays for packets that are not dropped. The low delay resulting from a small queue potentially comes at the cost of higher packet loss than would be seen with losses caused only by an overflowing queue as with drop-tail routers.

2.2.1. Random early detection

Random Early Detection (RED) [3] is an AQM mechanism that seeks to reduce the long-term average queue length in routers. Under RED, as each packet arrives, routers compute a weighted average queue length that is used to determine when to notify end-systems of incipient congestion. Congestion notification in RED is performed by marking a packet. If, when a packet arrives, congestion is deemed to be occurring, the arriving packet is marked. For standard TCP end-systems, a RED router drops marked packets in order to signal congestion through packet loss. If the TCP end-system understands packet-marking (as explained below), a RED router marks and then forwards the marked packet. The more packets a connection sends, the higher the probability that its packets will be marked. In this way, RED spreads out congestion notifications proportionally to the amount of space in the queue that a connection occupies. Evaluations of RED [7], though, showed that choosing appropriate parameter settings is difficult, the optimal setting of certain parameters depends on the traffic mix flowing through the router, and inappropriate parameter settings could be harmful to traffic performance.

2.2.2. Explicit congestion notification

Explicit Congestion Notification (ECN) [4] is an method for AQM mechanisms to mark packets and for end systems to respond to those marks. When an ECN-capable router signals congestion, it marks a packet by setting a bit in the packet's header. ECN-capable receivers echo this mark in ACKs sent back to the sender. Upon receiving an ACK with the ECN bit set, the sender reacts in the same way as it would react to a packet loss (i.e. by halving the congestion window). Thus, a TCP sender implementing ECN receives two different notifications of congestion: ECN and packet loss. This allows senders to be more adaptive to changing network conditions.

2.2.3. Adaptive RED

RED performance depends upon how its four parameters $\min_{th}$ (minimum threshold), $\max_{th}$ (maximum threshold), $\max_p$ (maximum drop probability) and w_q (weight given to new queue size measurements) are set [7]. Adaptive RED [6] is a modification to RED which addresses the difficulty of appropriately setting RED parameters. Adaptive RED adapts the value of $\max_p$ between 1 and 50% so that the average queue size is halfway between $\min_{th}$ and $\max_{th}$. Adaptive RED includes another modification to RED, called 'gentle RED' [5]. In gentle RED, when the average queue size is between $\max_{th}$ and $2\max_{th}$, the drop probability is varied linearly from $\max_p$ to 1, instead of being set to 1 as soon as the average is greater than $\max_{th}$. Additionally, when the average queue size is between $\max_{th}$ and $2\max_{th}$, selected packets are no longer marked, but always dropped, even if packets carry ECN markings.

3. Sync-TCP

Sync-TCP is an end-to-end early congestion detection and reaction mechanism that is based on TCP Reno, but

measures one-way transit times (OTTs) to detect network congestion. A connection's forward path OTT is the amount of time it takes a segment to traverse all links from the sender to the receiver and includes both propagation and queuing delays. Queues in routers build up before they overflow, resulting in increased OTTs. If senders directly measure changes in OTTs and back off when increasing OTTs indicate that congestion is occurring, congestion could be alleviated.

Any congestion control mechanism is built upon detecting congestion and then reacting to the congestion. TCP Reno detects congestion through segment loss and reacts to that congestion notification by reducing the congestion window. Sync-TCP detects congestion based on how its measured OTTs change. Sync-TCP reacts to segment loss in the same manner as TCP Reno (both by retransmitting the lost segment and by reducing the congestion window), but also reacts to additional congestion indications derived from OTTs. Sync-TCP adjusts the congestion window based on the degree of congestion present in the network. Unlike other delay-based end-to-end congestion control protocols, such as TCP Vegas and TCP Santa Cruz, Sync-TCP's change to TCP Reno is limited to the congestion control mechanism itself, making no changes to TCP Reno's error recovery or loss detection mechanisms.

3.1. OTTs vs RTTs

The queuing delay of a path can be estimated by taking the difference between the minimum-observed OTT and the most recent OTT. The minimum-observed OTT from a flow is assumed to be close to the propagation delay on the link. Any additional delay is assumed to be caused by queuing. Since the estimation of the queuing delay involves the difference between OTTs (rather than the exact values of OTTs), synchronized clocks are not required. OTTs can more accurately reflect queuing delay caused by network congestion than round-trip times (RTTs). Using RTTs there is no way to accurately estimate the forward-path queuing delay. If there is an increase in the RTT, the sender cannot distinguish between the cause being congestion on the forward (data) path, congestion on the reverse (ACK) path, or both.

3.2. Sync-TCP timestamp option

End systems can determine the OTTs between them by exchanging timestamps. For Sync-TCP, we added an option to the TCP header, based on the RFC 1323 timestamp option [11]. Fig. 1 pictures the full TCP header[2] along with a cut-out of the Sync-TCP timestamp option. This TCP header option includes the OTT in microseconds calculated for the last segment received (*OTT*), the current segment's sending

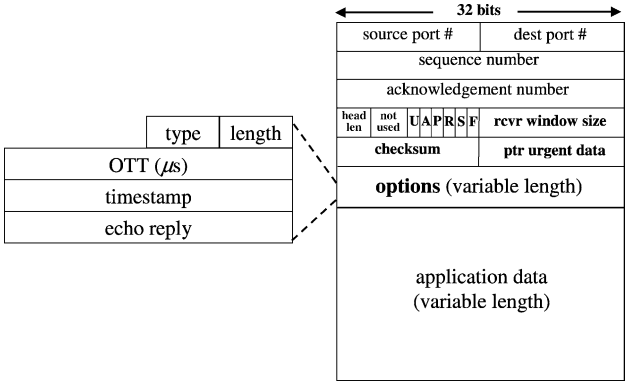

Fig. 1. TCP header and Sync-TCP timestamp option.

time (*timestamp*), and the sending time of the last segment received (*echo reply*). When a receiver sees any segment with the Sync-TCP timestamp option, it calculates the segment's OTT by subtracting the time the segment was sent from the time the segment was received. The OTT is then inserted into the next segment the receiver sends to the sender (generally an ACK). Upon receiving the ACK, the sender can estimate the current queuing delay by subtracting its minimum-observed OTT from the OTT present in the ACK.

3.3. Congestion detection

Sync-TCP uses changes in forward-path queuing delay to detect congestion. A Sync-TCP sender obtains a new queuing delay estimate for every ACK received. These queuing delays are averaged, and the trend of the average queuing delay is determined. The trend and magnitude of the average queuing delay are provided to the congestion reaction mechanism of Sync-TCP.

For every ACK received, Sync-TCP computes the weighted average of the estimated queuing delay. Sync-TCP uses the same smoothing factor as TCP's RTO estimator, which is 1/8. Once the average is computed, its magnitude falls into one of four regions, defined as a percentage of the *maximum*-observed queuing delay: 0–25, 25–50, 50–75, or 75–100%. This uniform partitioning of the space roughly corresponds to regions of negligible congestion, low congestion, moderate congestion, and heavy congestion, respectively. The maximum-observed queuing delay represents an estimate of the total amount of queuing resources available in the network.

Sync-TCP uses a simple trend analysis algorithm to determine when the average queuing delay is increasing or decreasing. The trend analysis algorithm is based on that by Jain and Dovrolis [13], which uses the trend of OTTs to measure available bandwidth. We chose to use nine samples for trend analysis in Sync-TCP, which marks a tradeoff between detecting longer term trends and quickly making decisions. Sync-TCP first gathers nine average queuing delay samples and splits them into three groups of three

[2] See Stevens [12] for a description of the header fields.

samples, in the order of their computation. The median, m_i, of each of the three groups is computed. Since there are only three medians, the trend is determined to be increasing if $m_0 < m_2$. Likewise, the trend is determined to be decreasing if $m_0 > m_2$. Sync-TCP computes a new trend every three ACKs by replacing the three oldest samples with the three newest samples for trend analysis. Due to the nine-sample trend analysis, no early congestion detection will occur for connections that receive fewer than nine ACKs (i.e. send fewer than 18 segments, if delayed ACKs are used by the receiver).

Each time an ACK is received, the congestion detection mechanism in Sync-TCP reports one of two outcomes: (1) not enough samples have been gathered to compute the trend, or (2) enough samples have been gathered, and the region where the average queuing delay lies in relation to the maximum-observed queuing delay and the direction of the trend are reported. We provide pseudocode of the Sync-TCP congestion detection algorithm in Algorithm 1. Fig. 2 shows the Sync-TCP congestion detection algorithm at work. This example comes from the transfer of 20 long-lived FTP connections in each direction over one bottleneck link. The connections in this example used Sync-TCP congestion detection but did not take any congestion control actions other than normal TCP Reno actions. The gray line is the actual queuing delay faced by packets entering the bottleneck. Each point (star, square, or circle) is the average queuing delay as estimated by one sender, when an ACK returns, with a 53 ms base (minimum) RTT. The stars indicate when there were not enough samples to compute the trend (the trend is only computed every three ACKs). The hollow boxes indicate when Sync-TCP detected the trend was decreasing, and the solid circles indicate when Sync-TCP detected the trend was increasing. The four average queuing delay regions defined above are outlined with dotted lines at 25, 50, 75, and 100% of the maximum-observed queuing delay. The average queuing delay tracks the actual queuing delay relatively well. The lag between the actual queuing delay and the average queuing delay is caused by both the time between the packet experiencing the delay and its ACK returning (delayed ACKs were used) and the average weighting factor of 1/8. Fig. 2 demonstrates that the trend analysis algorithm used in Sync-TCP can perform well in tracking the trend of the average queuing delay.

Algorithm 1: Sync-TCP congestion detection

Upon receiving a packet:
  Extract SendTime, EchoTime, and OTT from packet header
  OutOTT ← CurrentTime − SendTime
  Update MinOTT
  CurQDelay ← OTT − MinOTT
  Update MaxQDelay
  AvgQDelay ← 0.875AvgQDelay + 0.125 CurQDelay
  if *nine queuing delay samples gathered* **then**
    Update Trend
  else
    Not enough samples
  end
Before sending a packet:
  Insert CurrentTime, SendTime, and OutOTT into packet header

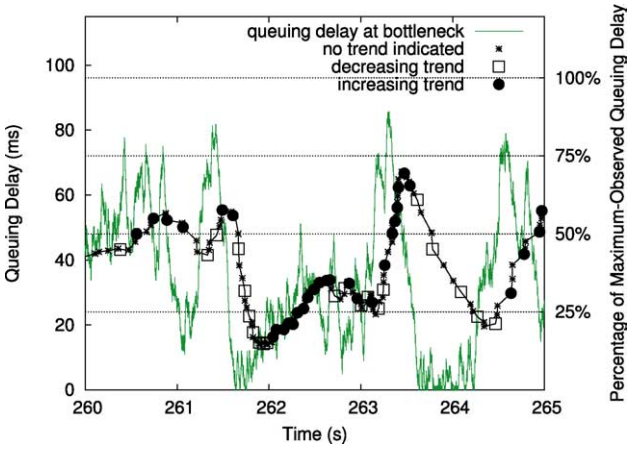

Fig. 2. Sync-TCP congestion detection. The graph pictures one connection's estimated average queuing delay during a portion of a long-lived file transfer.

3.4. Congestion reaction

Sync-TCP makes adjustments to the congestion window (*cwnd*) based on the indication provided by its congestion detection mechanism. Sync-TCP leverages the existing AIMD framework for congestion window adjustment but adjusts AIMD's α and β parameters differently than TCP Reno ($\alpha = 1$, $\beta = 0.5$). Following AIMD, when *cwnd* is to be increased, Sync-TCP incrementally increases *cwnd* for every ACK returned, and when *cwnd* is to be decreased, Sync-TCP makes the decrease immediately.

The additional congestion window decreases in Sync-TCP (as compared to TCP Reno) when the average queuing delay trend is increasing are balanced by the more aggressive congestion window increases when the average queuing delay trend is decreasing. The congestion detection mechanism in Sync-TCP will return a new congestion detection result every three ACKs (according to the trend analysis algorithm), causing Sync-TCP to potentially adjust α or β every three ACKs.

The adjustments made to *cwnd* if the trend of average queuing delays is *increasing* are described below:

- If the average queuing delay is less than 25% of the maximum-observed queuing delay, *cwnd* is increased by one segment over one RTT (α is set to 1). This region corresponds to negligible congestion, so the goal is to grow the congestion window to obtain additional available bandwidth. This is the same increase that TCP Reno uses during congestion avoidance.
- If the average queuing delay is between 25 and 50% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 10% (β is set to 0.9). This region corresponds to low congestion. Since the trend is increasing, but the queuing delay is relatively low, Sync-TCP reacts quickly but with a small reduction.
- If the average queuing delay is between 50 and 75% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 25% (β is set to 0.75). This region corresponds to moderate congestion. Since the trend is increasing and the queuing delay is higher than the previous region, the congestion window reduction should be larger than before. The idea is that a 10% reduction (from the previous region) was not enough to alleviate congestion.
- If the average queuing delay is above 75% of the maximum-observed queuing delay, *cwnd* is decreased immediately by 50% (β is set to 0.5). This region corresponds to heavy congestion and likely imminent packet loss. This is the same decrease that TCP Reno would make if a segment loss were detected by the receipt of three duplicate ACKs and that an ECN-enabled TCP sender would make if a segment were received with the congestion-experienced bit set.

The adjustments made to *cwnd* if the trend of average queuing delays is *decreasing* are described below:

- If the average queuing delay is less than 25% of the maximum-observed queuing delay, *cwnd* is increased by 50% over one RTT (α is set to $x_i(t)/2$, where $x_i(t)$ is the value of *cwnd* at time t, in order to achieve a 50% increase). This region has a very low queuing delay and a decreasing trend. This indicates that there is bandwidth available for use. The flow should quickly increase its congestion window in order to obtain some of this available bandwidth. Note that this sharp increase is only possible because if the average queuing delay starts increasing, the congestion window will be reduced—without waiting for packet loss to occur. If the congestion detection signal remained in this region, *cwnd* would be doubled in two RTTs. This increase is slower than TCP Reno's increase during slow start, but more aggressive than during congestion avoidance.
- If the average queuing delay is between 25 and 50% of the maximum-observed queuing delay, *cwnd* is increased by 25% over one RTT (α is set to $x_i(t)/4$ to achieve a 25% increase). This region corresponds to low congestion and a decreasing trend. Since there is little danger of packet loss, the congestion window is increased. Again, an aggressive increase is balanced by an early decrease if the average queuing delay starts increasing.
- If the average queuing delay is between 50 and 75% of the maximum-observed queuing delay, *cwnd* is increased by 10% over one RTT (α is set to $x_i(t)/10$ to achieve a 10% increase). This region corresponds to moderate delay, but a decreasing trend. The goal is to keep data flowing in the network, so the congestion window is gently increased. The 10% increase would typically be a very slow increase in the congestion window.
- If the average queuing delay is above 75% of the maximum-observed queuing delay, *cwnd* not adjusted (β is set to 1). This region corresponds to heavy congestion, but a decreasing queuing delay trend. No change to the congestion window is made because appropriate adjustments will be made if the conditions change. In the future, if the average remained in this region, but the trend began to increase, *cwnd* would be reduced by 50%. If the trend remained decreasing, the average queuing delay would eventually decrease below 75% of the maximum, causing *cwnd* to be increased slightly.

We provide pseudocode of the Sync-TCP congestion reaction algorithm in Algorithm 2.

The values used here to adjust the congestion window are based on extensive experimentation [14] and a desire to react according to the degree of congestion. A precise evaluation of different settings is the subject of future work,

Algorithm 2: Sync-TCP congestion reaction

```
if Trend is increasing then
    if AvgQDelay is 0–25% of MaxQDelay then
        Increase cwnd by one segment over one RTT
    else if AvgQDelay is 25–50% of MaxQDelay then
        Decrease cwnd by 10%
    else if AvgQDelay is 50–75% of MaxQDelay then
        Decrease cwnd by 25%
    else if AvgQDelay is 75–100% of MaxQDelay then
        Decrease cwnd by 50%
else if Trend is decreasing then
    if AvgQDelay is 0–25% of MaxQDelay then
        Increase cwnd by 50% over one RTT
    else if AvgQDelay is 25–50% of MaxQDelay then
        Increase cwnd by 25% over one RTT
    else if AvgQDelay is 50–75% of MaxQDelay then
        Increase cwnd by 10% over one RTT
    else if AvgQDelay is 75–100% of MaxQDelay then
        Do nothing
end
```

but the empirical evaluation given here shows that these reasonable settings work well.

4. Evaluation methodology

Here we describe the experimental setup of our evaluation of Sync-TCP, including the HTTP traffic model used, the metrics we use to evaluate performance, and a description of the experiments that were run.

Initial evaluations of network protocols often use a simple traffic model: a small number of long-lived flows with equal RTTs, with data flowing in one direction, ACKs flowing in the opposite direction, and every segment immediately acknowledged. These scenarios may be useful in understanding and illustrating the basic operation of a protocol, but they are not sufficiently complex to be used in making evaluations of one protocol versus another. We evaluate the performance of Sync-TCP with non-persistent HTTP 1.0 traffic[3], because it has a mix of long and short flows and represents a worst-case for end-system congestion control mechanisms. Additionally, a recent measurement study [15] shows that persistent HTTP connections account for only 15% of all connections. We note that it would be better to test against realistic mixes of traffic, but such models are not available, so the HTTP model is the most realistic traffic model available.

4.1. Experimental setup

We ran simulations with two-way HTTP traffic using the *ns* network simulator [16]. The two-way traffic loads were designed to provide roughly equal levels of congestion on both the forward path and reverse path in the network. Fig. 3 is a simplified depiction of the topology used. Each circle and hexagon in the figure represent five *ns* nodes (each connected by a separate link to its nearest router) in the actual experimental topology. Each *ns* node represents a 'cloud' of HTTP clients or servers (i.e. end systems sharing an aggregation link). The interior squares are routers. The dashed lines represent the direction of the flow of data traffic in the network. Traffic generated by circles 1–4 does not leave the 'middle' of the network. Traffic generated by circles 0 and 5 traverses all routers. The links between the routers carry two-way traffic. This topology is referred to here as the '6Q parking lot' topology because of the six routers used and the topology's resemblance to a row in a parking lot. This network setup was first proposed by Floyd [17] to allow testing of multiple congested routers using cross-traffic on the interior links. In Fig. 3, we distinguish between traffic on the network as follows:

[3] An evaluation of Sync-TCP using competing long-lived FTP flows is given in [14].

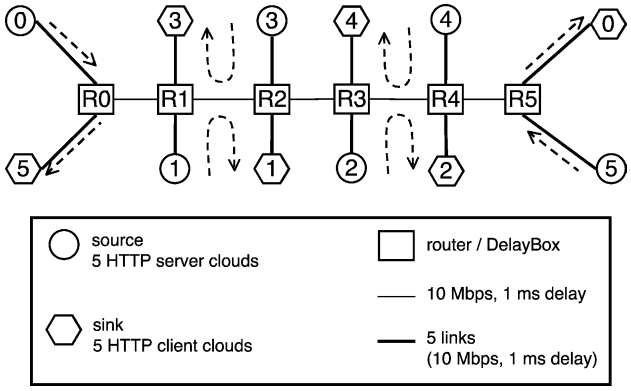

Fig. 3. Simplified 6Q parking lot topology.

- *Forward-path end-to-end traffic*. Data from source 0 to sink 0 and ACKs from sink 5 to source 5.
- *Reverse-path end-to-end traffic*. Data from source 5 to sink 5 and ACKs from sink 0 to source 0.
- *Forward-path cross-traffic*. Data from source 1 to sink 1 and from source 2 to sink 2 and ACKs from sink 3 to source 3 and from sink 4 to source 4.
- *Reverse-path cross-traffic*. Data from source 3 to sink 3 and from source 4 to sink 4 and ACKs from sink 1 to source 1 and from sink 2 to source 2.

Each link has a 10 Mbps capacity and a 1 ms propagation delay. The 1 ms propagation delay on each link results in a minimum RTT of 14 ms for any end-to-end flow (each end-to-end flow traverses seven links in each direction), although, as explained below, all flows see a different minimum delay. Since there are actually five *ns* nodes feeding into each router on each interface (each *ns* node logically has its own interface to the router), there is a maximum incoming rate of 50 Mbps to each router from each circle shown in Fig. 3. The data presented herein comes from measurements of the end-to-end traffic on the forward path in the network.

Three different high-level scenarios will be used to evaluate Sync-TCP:

- *One bottleneck*. Congestion is caused by the aggregation of traffic between routers R0 and R1 on the forward path and between routers R5 and R4 on the reverse path. There is a maximum of 50 Mbps of traffic entering R0 and R5. These routers can forward at a maximum of 10 Mbps, so congestion will result.
- *Two bottlenecks*. Congestion is caused by two-way cross-traffic between routers R1 and R2 and routers R3 and R4. For example, R1 will have a maximum of 10 Mbps of traffic coming from R0 and a maximum of 50 Mbps of additional traffic coming from circle 1 (along with the ACKs generated by hexagon 3) to forward over a 10 Mbps link.
- *Three bottlenecks*. Congestion is caused by the aggregation present in the one bottleneck case plus two-way

cross-traffic between routers R1 and R2 and routers R3 and R4.

4.2. HTTP traffic generation

HTTP traffic consists of communication between web clients and web servers. In HTTP 1.0, a client opens a TCP connection to make a single request from a server. The server receives the request and sends the response data back to the client. Once the response has been successfully received, the TCP connection is closed. The time elapsed between the client opening the TCP connection and closing the TCP connection is the *HTTP response time* and represents the completion of a single HTTP request–response pair.

We use the PackMime HTTP traffic model [18,19], developed at Bell Labs, to generate synthetic web traffic. The level of traffic generated by this model is based on the number of new connections that are generated per second by a 'cloud' of web clients. PackMime provides distributions for HTTP request sizes, HTTP response sizes, and the time between new HTTP connections based on the user-supplied connection rate parameter.

Previous studies have shown that the size of files transferred over HTTP is heavy-tailed [20,21]. With heavy-tailed file sizes, there are a large number of small files and a non-negligible number of extremely large files. Fig. 4 shows the CDF of HTTP response sizes generated by the PackMime model. Note that the *x*-axis is on a log scale. The median response size is under 1 KB, but the largest response generated is almost 50 MB.

The levels of offered load used in our experiments are expressed as a percentage of the capacity of a 10 Mbps link. We initially ran the network at 100 Mbps and determined the PackMime connection rates (essentially the HTTP request rates) that would result in average link utilizations (in both forward and reverse directions) of 5, 6, 7, 7.5, 8, 8.5, 9, 9.5, 10 and 10.5 Mbps. These rates are then used to represent a percentage of the capacity of a 10 Mbps link. For example, the connection rate that results in an average utilization of 8.5 Mbps, or 8.5% of the (clearly uncongested) 100 Mbps link, will be used to generate an offered load on the 10 Mbps link of 8.5 Mbps, or 85% of the 10 Mbps link. Note that this '85% load' (i.e. the connection rate that results in 8.5 Mbps of traffic on the 100 Mbps link) will not actually result in 8.5 Mbps of traffic on the 10 Mbps link. The bursty HTTP sources will cause congestion on the 10 Mbps link and the actual utilization of the link will be a function of the protocol and router queue management mechanism used.

The routers in these simulations are *ns* nodes that we developed, called DelayBoxes, which, in addition to forwarding segments, also delay segments. DelayBox is an *ns* analog to dummynet [22] and NIST Net [23], which are used, respectively, in FreeBSD and Linux network testbeds to delay segments. With DelayBox, segments from a TCP connection can be delayed before being passed on to the next *ns* node. This allows each TCP connection to experience a different minimum delay (and hence a different minimum RTT), based on random sampling from a delay distribution, and thus, allows us to simulate a larger network. In these experiments, DelayBox uses an empirical RTT distribution from the PackMime model. In this RTT distribution, the shortest RTT is 15 ms, the longest RTT is over 9 s, and the median RTT is 78 ms. Fig. 5 shows the CDF of the minimum RTTs (including the 14 ms propagation delay added by the network topology) for end-to-end flows in the simulation. Note that the *x*-axis is on a log scale. The median base RTT is under 100 ms, but the largest base RTT is over 9 s. This distribution of RTTs is similar to that found in a recent study of RTTs on the Internet [24].

4.3. Performance metrics

In each experiment, we measured network-level metrics, including the average packet loss percentage at each congested router, the average queue size at each congested

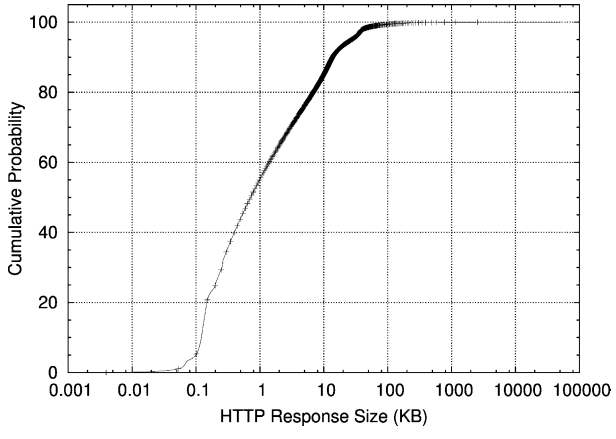

Fig. 4. HTTP response sizes.

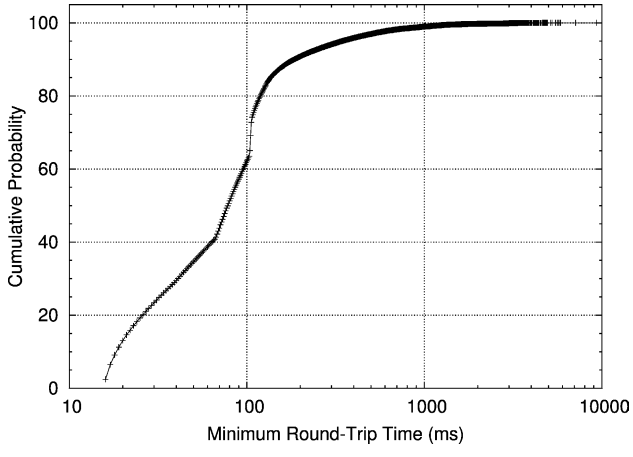

Fig. 5. Round-trip times.

router, the average throughput, the average goodput (data arriving to the web clients), and the average link utilization at the link closest to the forward-path HTTP clients. We also measured application-level metrics, such as the HTTP response times and the average goodput per response. The application-level metrics are measured only for those flows that have completed their request–response pair. The network-level metrics are measured for all traffic, including flows that still have data in the network when the experiment ends.

HTTP response time cumulative distribution functions (CDFs) are the main metric we use for evaluating the performance of HTTP traffic. The goal is for a large percentage of request–response pairs to complete in a short amount of time, so, in our figures, the closer the CDF curve is to the top-left corner, the better the performance.

4.4. Experiments

We evaluated Sync-TCP over drop-tail routers against TCP Reno over drop-tail routers and against ECN-enabled TCP SACK over Adaptive RED routers (SACK-ARED-ECN). A study of recent TCP protocol improvements and AQM mechanisms in the context of HTTP traffic [14,25] showed that performance tradeoffs existed between drop-tail and Adaptive RED queue management. For this reason, we chose to look at both drop-tail and Adaptive RED queue management mechanisms as comparisons to Sync-TCP.

We ran each offered load (50–105% of the 10 Mbps link) over one congested bottleneck for each of the three TCP protocols tested. The experiments were run until 250,000 HTTP request–response pairs completed. For multiple bottlenecks, we ran experiments so that the total offered load on the interior links (with both end-to-end and cross-traffic flows) averaged 75, 90, and 105% of the 10 Mbps link. Because of the additional memory and run time involved in simulating multiple congested links, the multiple bottleneck experiments were run until 150,000 HTTP request–response pairs completed.

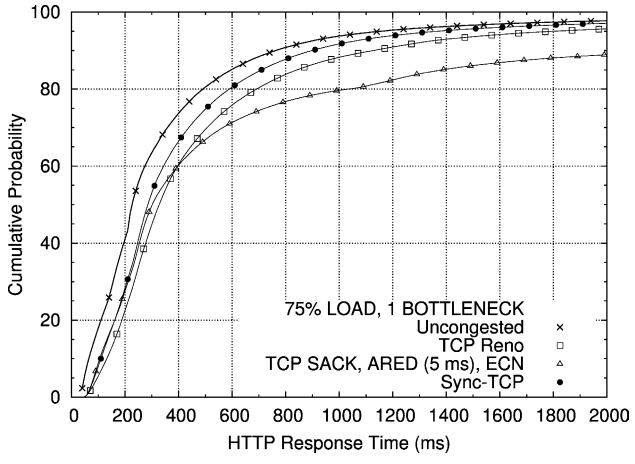

Fig. 6. HTTP response times, 75% load, 1 bottleneck.

All experiments were run with two-way traffic and delayed ACKs. Unless otherwise specified, each TCP protocol was tested in isolation (e.g. all connections used Sync-TCP during the Sync-TCP experiments). All experiments used the same random number generator seed for sampling from the PackMime distributions, so the same file sizes were generated and transferred in each experiment. For the drop-tail experiments, the maximum queue length at each router was set to twice the bandwidth-delay product, which was 68 1420-byte segments, resulting in queue lengths of 136 packets. For the Adaptive RED experiments, the maximum queue length was set to 5 times the bandwidth-delay product to eliminate tail drops. Additionally, for Adaptive RED, the target delay was set to 5 ms. This is the default setting in *ns* and also has been shown to provide short queues and good response-time performance for short-lived HTTP connections [14,25].

5. Results

Here, we present results from the experiments described above. We show results from experiments with 1, 2, and 3 congested links. We also looked at the impact that clock drift has on Sync-TCP and the feasibility of incremental deployment of Sync-TCP alongside TCP Reno.

5.1. Single bottleneck

With one bottleneck, congestion on the forward path is caused only at the aggregation point, router R0. Figs. 6–8 show the HTTP response time CDFs for 75, 90, and 95% offered load, respectively[4]. In these figures, for comparison we also include the HTTP response times obtained by the same PackMime connection rate on an uncongested network where all links had 100 Mbps capacity, which represents the best possible performance. At 75% load, SACK-ARED-ECN slightly outperforms Reno for responses that complete in 350 ms or less (i.e. a larger percentage of HTTP request–response pairs completed in 350 ms or less with SACK-ARED-ECN than with Reno). There is a crossover point in performance between SACK-ARED-ECN and Reno at 400 ms. After that point, Reno performs significantly better. With Sync-TCP, there is no crossover point with either Reno or SACK-ARED-ECN; Sync-TCP always performs better. At 90 and 95% loads (Figs. 7 and 8), Sync-TCP continues to perform better than both Reno and SACK-ARED-ECN. There is a slightly larger percentage of flows that complete in 200 ms or less with SACK-ARED-ECN than either Sync-TCP or Reno, but after that crossover point, SACK-ARED-ECN performs the worst of the three protocols.

[4] For space considerations, we present only a subset of the experiments that were run. The full set of results is presented in [14].

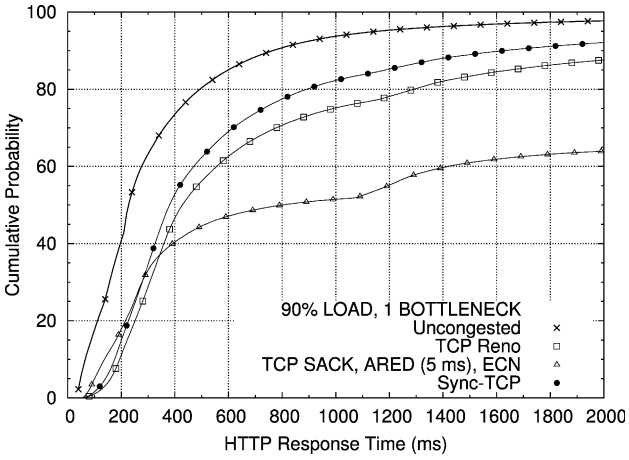

Fig. 7. HTTP response times, 90% load, 1 bottleneck.

We present summary statistics at 90% load in Table 1. The average completed response size is interesting because, for all of the protocols, the same response sizes were generated, but for each protocol, a different subset of the responses actually completed in the first 250,000 responses. If large responses stall (through excessive packet loss), smaller responses will continue to complete and reduce the average response size. Therefore, the larger the average completed response size, the better the performance that large flows received. The average goodput per response measures the average of the goodput achieved by each of the 250,000 HTTP responses.

Fig. 7 and Table 1 together show that SACK-ARED-ECN has a larger percentage of flows finish under 200 ms than Reno and Sync-TCP because it keeps the average queue size smaller. Unfortunately, the average queue size is larger than the target queue size, so SACK-ARED-ECN keeps the queue small by dropping packets, even though ECN is used (because the average queue size is large enough that Adaptive RED will drop packets even from ECN-capable connections). Sync-TCP, on the other hand, keeps smaller queues than TCP Reno, but also reduces packet loss. The smaller queues with Sync-TCP are caused by connections backing off as they see increased queuing delays. The flows do not see diminished performance as compared to Reno because the queues are smaller and packet losses are avoided.

Table 1
90% Total load, 1 bottleneck

	TCP Reno	SACK ARED ECN	Sync-TCP	100 Mbps
%Packet drops at R0	3.9	13.9	2.0	0
Avg. queue size (pckts)	64.4	19.4	53.9	0
Avg. completed rspsz (B)	6960	6353	7202	7256
Goodput/response (kbps)	89.4	83.9	99.6	226.8
Median rsptime (ms)	430	810	390	230
Mean rsptime (ms)	1268.1	4654.7	943.6	406.1

Sync-TCP early congestion detection does not take effect until after nine ACKs have been received by an HTTP server. With delayed ACKs, this means that the HTTP response must be larger than 18 segments (about 25 KB) for the sender to use Sync-TCP early congestion detection. Until the 9th ACK is received and the first trend can be computed, the sender will use Reno congestion control. In these experiments, only 5% of the completed responses are greater than 25 KB, so only a fraction of the HTTP flows are actually using Sync-TCP early congestion detection and reaction. The other 95% of the connections are essentially using Reno. *The overall improved response times are achieved by applying early congestion detection and reaction to only 5% of the connections*. To show how these 5% of connections perform when using Sync-TCP early congestion detection, Fig. 9 shows the HTTP response times up to 20 s for flows that have responses greater than 25 KB. In this graph, the largest Sync-TCP flows perform no worse than the corresponding flows from Reno. Sync-TCP succeeded in providing better performance to all flows while not penalizing longer flows. The longer flows that use early congestion detection also can take advantage of the more aggressive increase of *cwnd* when the queuing delay is low.

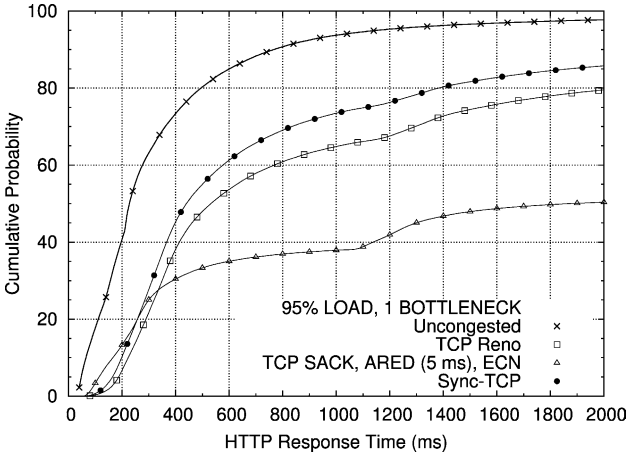

Fig. 8. HTTP response times, 95% load, 1 bottleneck.

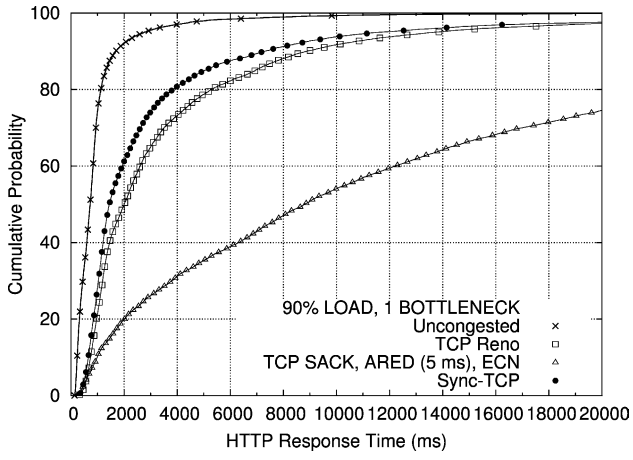

Fig. 9. HTTP response times, 90% load, responses >25 KB.

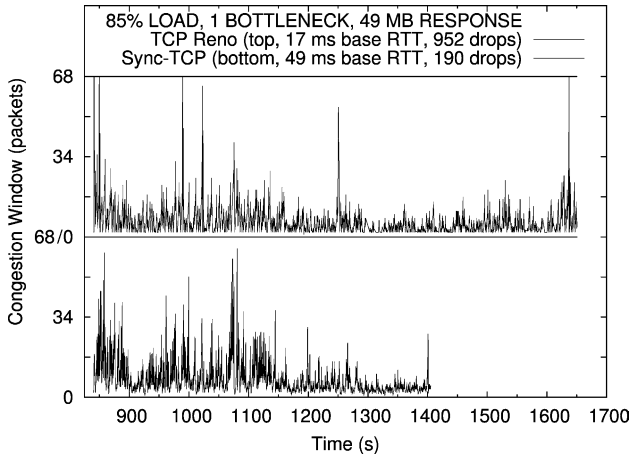

Fig. 10. Congestion window comparison.

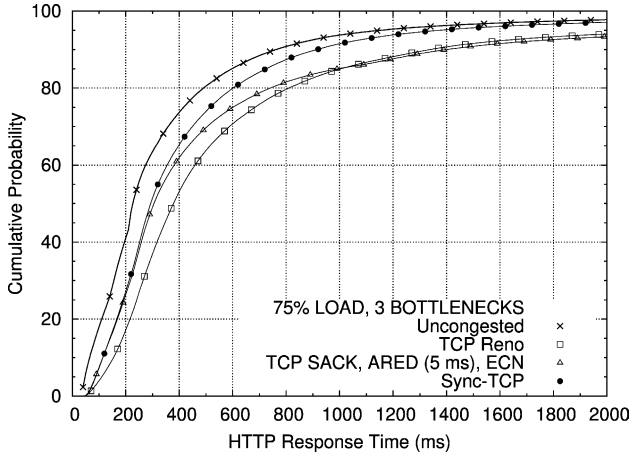

Fig. 12. HTTP response times, 75% load, 3 bottlenecks.

Fig. 10 shows the congestion window over time for the largest response that completed at 85% load. The graph shows the transfer of a 49 MB file that starts at around the same time in both the Sync-TCP and Reno experiments. The Reno congestion window is on the top half of the graph, and the Sync-TCP congestion window is on the bottom half of the graph. The maximum send window is 68 packets. The congestion window for SACK-ARED-ECN is not shown because although this particular response started at the same time as in the Reno and Sync-TCP experiments, it did not complete before time 2400, when the simulation ended. The interesting things about this graph are that the Sync-TCP flow finishes over 200 s before the Reno flow, the Sync-TCP flow sees 5 times fewer packet drops than the Reno flow, and the Sync-TCP flow has a minimum RTT that is almost 3 times larger than the Reno flow. With Sync-TCP, large responses are not being harmed, but rather, they are benefiting, along with smaller responses, from the small queues and low drop rates that result when larger flows react to increasing queuing delay.

5.2. Multiple bottlenecks

In the multiple bottleneck experiments, two-way cross-traffic is introduced onto the link between R1 and R2 and onto the link between R3 and R4 (Fig. 3). Figs. 11 and 12 show the HTTP response times of end-to-end traffic for Reno, SACK-ARED-ECN, and Sync-TCP with 2 and 3 bottlenecks at 75% offered load on the interior links. Figs. 13 and 14 show the HTTP response times with 2 and 3 bottlenecks at 90% offered load on the interior links. For SACK-ARED-ECN, each router is running Adaptive RED with a 5 ms target delay, and all of the cross-traffic is running SACK. For Sync-TCP, all of the cross-traffic is running Sync-TCP. At 75% load, the relative performance of Reno, SACK-ARED-ECN, and Sync-TCP is similar to that with one bottleneck (Fig. 6). Sync-TCP still performs better than the other two protocols. The crossover point between Reno and SACK-ARED-ECN shifts, with SACK-ARED-ECN performing better relative to Reno as the number of bottlenecks increases.

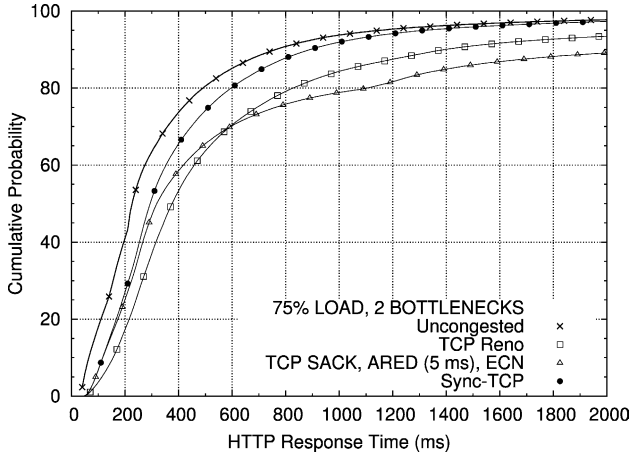

Fig. 11. HTTP response times, 75% load, 2 bottlenecks.

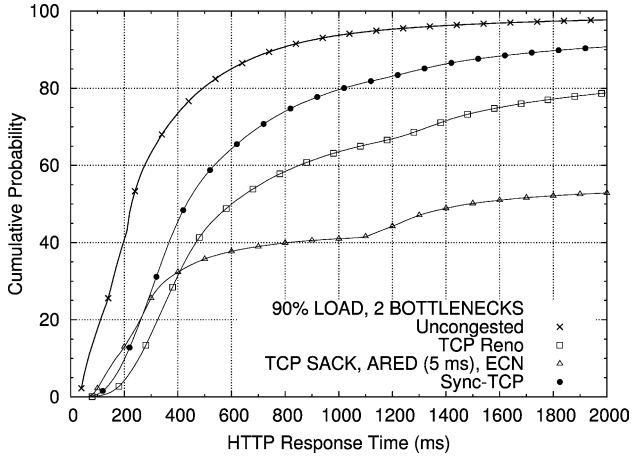

Fig. 13. HTTP response times, 90% load, 2 bottlenecks.

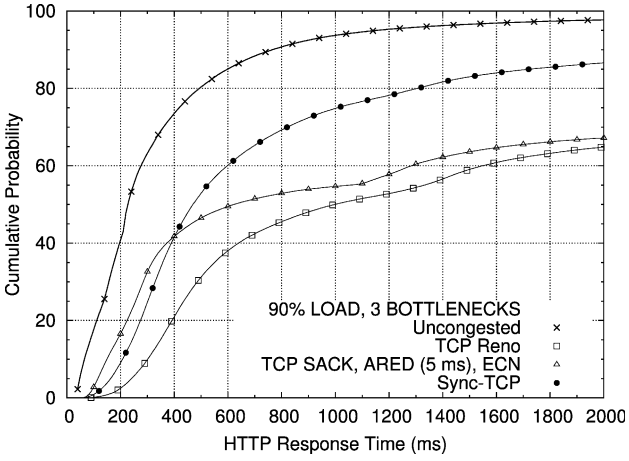

Fig. 14. HTTP response times, 90% load, 3 bottlenecks.

With Reno, performance degrades as more congested links are added, as expected. SACK-ARED-ECN, though, performs better with 3 bottlenecks than with either 1 or 2 bottlenecks. We suspect this is because of the chain of Adaptive RED routers keep all of the queues small. The performance of Sync-TCP degrades only slightly as the number of bottlenecks increases. In fact, the performance of Sync-TCP with 3 bottlenecks is comparable to that of Reno with only 1 bottleneck.

5.3. Impact of clock drift

Sync-TCP does not require that computer clocks be synchronized in time because it uses changes in OTT to detect congestion. Sync-TCP is dependent, though, on the degree of clock drift between two end systems. Clock drift is the result of computers' clocks not being synchronized in frequency and, thus, running at different rates. Mills [26] performed a survey of 20,000 Internet hosts and found clock drift rates as high as 500 μs/s (with a median drift of 78 μs/s). To test how this amount of clock drift would affect Sync-TCP, we ran experiments with 75, 90, and 95% offered load over a single bottleneck where each flow was assigned a clock drift rate sampled from a uniform distribution with a maximum of 500 μs/s. The results of these experiments are presented in Fig. 15. There is no large difference between the experiments that include clock drift and those that have no clock drift. Only 5% of flows use the computation of queuing delays to detect and react to congestion, so the number of flows that are affected by the clock drift is minimal. Additionally, the round-trip propagation delay in the network is 14 ms. For the clock drift to reach values on that order, a flow would have to have the maximum drift of 500 μs/s and take over 28 s to complete. At 95% offered load, 97% of the HTTP responses larger than 25 KB finish in under 28 s.

5.4. Incremental deployment of Sync-TCP

Previous work by Martin et al. [27] has suggested that delay-based congestion avoidance (DCA) protocols that detect and react to congestion based on increasing RTTs, such as TCP Vegas, are not suitable for incremental deployment on the Internet. This finding is based on the premise that for incremental deployment to be viable, every connection that uses DCA must see higher throughput and lower loss than if the connection had used Reno. Martin et al. showed that a single DCA connection will not receive better performance because increasing RTTs are not a good predictor of packet loss. We claim that OTTs are a better indicator of the state of the forward-path of the network than RTTs and are less susceptible to noise. Additionally, with Sync-TCP, we are not trying to predict packet loss, but instead, to react to congestion (which can, but does not always, lead to packet loss). Further, new protocols are not deployed one connection at a time, but rather, are tied to specific operating system releases, causing many connections to use the new protocol at once. The impact on all connections (both those that make the change and those

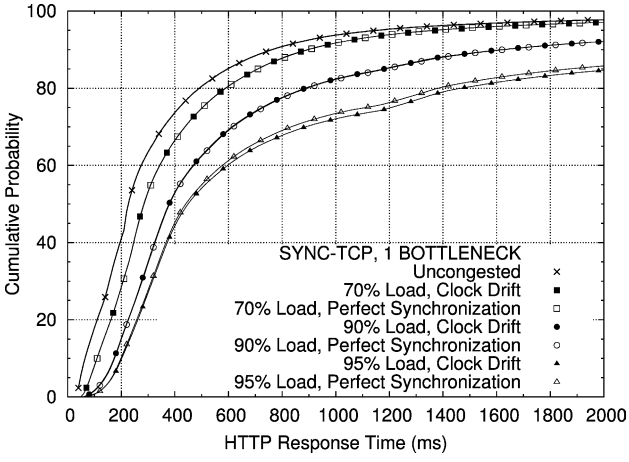

Fig. 15. HTTP response times, Sync-TCP with clock drift.

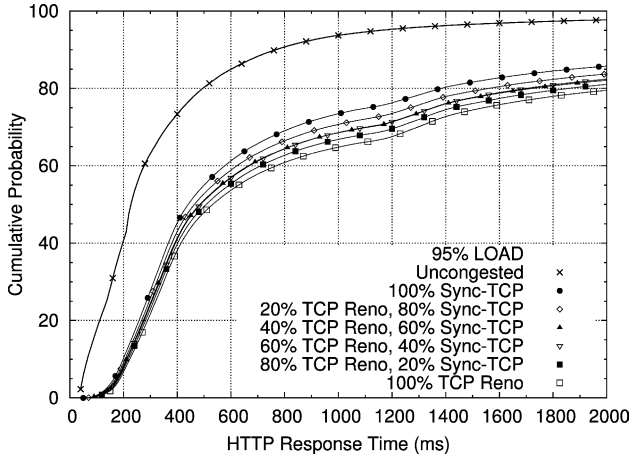

Fig. 16. HTTP response times, 95% load, Reno/Sync-TCP mix.

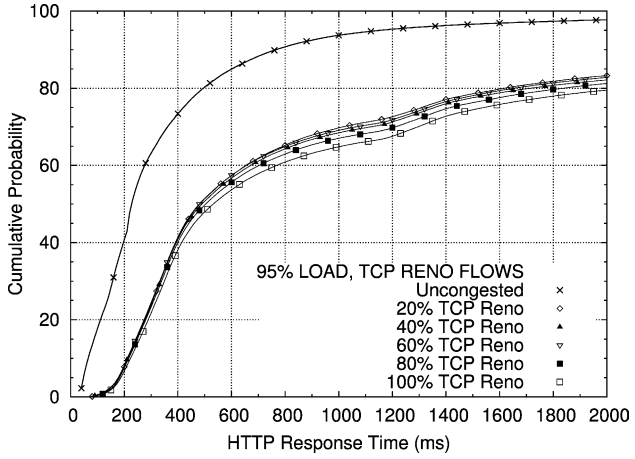

Fig. 17. HTTP response times, 95% load, Reno connections, mix of Reno and Sync-TCP.

that do not) must also be considered when deciding to deploy a new protocol.

Here we show the results at 95% load of various mixtures between Sync-TCP and Reno connections. We ran experiments with 20, 40, 60, and 80% Reno traffic. Fig. 16 shows the HTTP response times for all connections. We also show the 100% Reno and 100% Sync-TCP cases for comparison. The higher the percentage of Reno traffic, the closer the HTTP response time curve is to the 100% Reno case. Adding in just 20% Sync-TCP traffic improves the overall performance. Additionally, with 20% Sync-TCP, only 1% of connections use early congestion detection and reaction (i.e. send responses that are larger than 25 KB). Fig. 17 shows the performance from these same experiments for only those connections using Reno. This shows that adding Sync-TCP does not have a detrimental effect on the Reno connections. In fact, adding Sync-TCP traffic improves the performance of the Reno connections. Fig. 18 shows the corresponding figure for those connections that used Sync-TCP. Performance is quite good with only 20% Sync-TCP and increases with the percentage of Sync-TCP traffic. Further, with 20% Sync-TCP traffic, the Sync-TCP connections performed better than if the corresponding connections had used Reno.

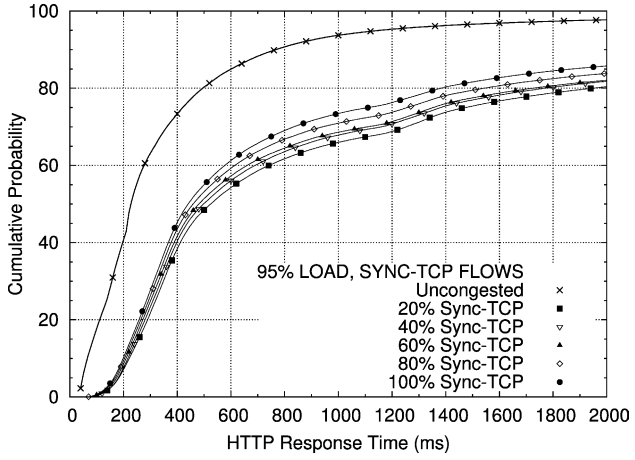

Fig. 18. HTTP response times, 95% load, Sync-TCP connections, mix of reno and Sync-TCP.

6. Conclusions

We have presented Sync-TCP, a delay-based early congestion detection and reaction mechanism that uses one-way transit times to estimate forward-path queuing delay. If senders detect when queuing delays are increasing, they can adjust their sending rates accordingly to help lower queue sizes and reduce the occurrence of packet loss. Additionally, monitoring queuing delays allows connections to detect when congestion is subsiding and take advantage of the excess capacity in the network by increasing their sending rates more aggressively than TCP Reno.

We evaluated Sync-TCP in the context of two-way bursty HTTP traffic with and without cross-traffic. Sync-TCP provides better throughput, better HTTP response-time performance, and lower packet loss rates than TCP Reno over drop-tail routers and ECN-enabled SACK over Adaptive RED routers. Sync-TCP achieves the goal of keeping queues small by detecting increasing queuing delays and reacting to that congestion signal, as opposed to active queue management techniques, which often must drop packets to signal congestion.

We have shown that Sync-TCP does not require the use of synchronized clocks and that typical computer clock drift rates do not greatly affect the performance of Sync-TCP. We have also shown that Sync-TCP can co-exist in a network with TCP Reno and that the addition of even a small percentage of Sync-TCP connections can improve the performance of competing TCP Reno connections.

References

[1] M. Allman, V. Paxson, W.R. Stevens, TCP congestion control, RFC 2581, April 1999.

[2] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenkar, J. Wroclawski, L. Zhang, Recommendations on queue management and congestion avoidance in the Internet, RFC 2309, April 1998.

[3] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM Transactions on Networking 1 (4) (1993) 397–413.

[4] K. Ramakrishnan, S. Floyd, A proposal to add explicit congestion notification (ECN) to IP, RFC 2481, experimental, January 1999.

[5] S. Floyd, Recommendation on using the gentle variant of RED, Technical Note, http://www.icir.org/floyd/red/gentle.html, March 2000.

[6] S. Floyd, R. Gummadi, S. Shenker, Adaptive RED: an algorithm for increasing the robustness of RED's active queue management, Technical Note, August 2001.

[7] M. Christiansen, K. Jeffay, D. Ott, F.D. Smith, Tuning RED for web traffic, IEEE/ACM Transactions on Networking 9 (3) (2001) 249–264.
[8] L. Brakmo, S. O'Malley, L. Peterson, TCP Vegas: new techniques for congestion detection and avoidance, in: Proceedings of ACM SIGCOMM, London, UK, 1994, pp. 24–35.
[9] C. Parsa, J.J. Garcia-Luna-Aceves, Improving TCP congestion control over internets with heterogeneous transmission media, in: Proceedings of the Seventh IEEE International Conference on Network Protocols (ICNP), Toronto, Canada, 1999, pp. 213–221.
[10] C. Jin, D.X. Wei, S.H. Low, FAST TCP: motivation, architecture, algorithms, performance, in: Proceedings of IEEE INFOCOM, Hong Kong, 2004.
[11] V. Jacobson, R. Braden, D. Borman, TCP extensions for high performance, RFC 1323, May 1992.
[12] W.R. Stevens, TCP/IP Illustrated, The Protocols, vol. 1, Addison-Wesley, Reading, MA, 1994.
[13] M. Jain, C. Dovrolis, End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput, in: Proceedings of ACM SIGCOMM, Pittsburgh, PA, 2002, pp. 295–308.
[14] M.C. Weigle, Investigating the use of synchronized clocks in TCP congestion control, PhD thesis, University of North Carolina at Chapel Hill, August 2003.
[15] F. Hernandez-Campos, F.D. Smith, K. Jeffay, Tracking the evolution of web traffic: 1995–2003, in: Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS), Orlando, FL, 2003, pp. 16–25.
[16] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, H. Yu, Advances in network simulation, IEEE Computer 33 (5) (2000) 59–67.
[17] S. Floyd, Connections with multiple congested gateways in packet-switched networks part 2: two-way traffic, Technical Note, ftp://ftp.ee.lbl.gov/papers/gates2.ps.Z, 1991.
[18] J. Cao, W.S. Cleveland, Y. Gao, K. Jeffay, F.D. Smith, M.C. Weigle, Stochastic models for generating synthetic HTTP source traffic, in: Proceedings of IEEE INFOCOM, Hong Kong, 2004.
[19] W.S. Cleveland, D. Lin, D.X. Sun, IP packet generation: statistical models for TCP start times based on connection-rate superposition, in: Proceedings of ACM SIGMETRICS, Santa Clara, CA, 2000, pp. 166–177.
[20] F.D. Smith, F. Hernandez-Campos, K. Jeffay, D. Ott, What TCP/IP protocol headers can tell us about the web, in: Proceedings of ACM SIGMETRICS, Cambridge, MA, 2001, pp. 245–256.
[21] P. Barford, M.E. Crovella, Generating representative web workloads for network and server performance evaluation, in: Proceedings ACM SIGMETRICS, Madison, WI, 1998, pp. 151–160.
[22] L. Rizzo, Dummynet: a simple approach to the evaluation of network protocols, ACM SIGCOMM Computer Communication Review 27 (1) (1997) 31–41.
[23] M. Carson, D. Santay, NIST Net: a Linux-based network emulation tool, ACM SIGCOMM Computer Communication Review 33 (3) (2003) 111–126.
[24] J. Aikat, J. Kaur, F.D. Smith, K. Jeffay, Variability in TCP round-trip times, in: Proceedings of the ACM SIGCOMM Internet Measurement Conference (IMC'03), Miami, FL, 2003, pp. 279–284.
[25] M. Weigle, K. Jeffay, F. Smith, Quantifying the effects of recent protocol improvements to standards-track TCP, in: Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS), Orlando, FL, 2003, pp. 226–229.
[26] D.L. Mills, The network computer as precision timekeeper, in: Proceedings of the 28th Precision Time and Time Interval (PTTI) Applications and Planning Meeting, Reston, VA, 1996, pp. 96–108.
[27] J. Martin, A. Nilsson, I. Rhee, Delay-based congestion avoidance for TCP, IEEE/ACM Transactions on Networking 11 (2) (2003) 356–369.