Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computer Communications xxx (2006) xxx–xxx
computer communications
www.elsevier.com/locate/comcom

Quantifying the effects of recent protocol improvements to 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 29634-1906, USA
[b] Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, USA

Received 15 March 2005; received in revised form 19 April 2005; accepted 1 March 2006

Abstract

We assess the state of Internet congestion control and error recovery through a controlled study that considers the integration of standards-track TCP error recovery and both TCP and router-based congestion control. The goal is to examine and quantify the benefits of deploying standards-track technologies for contemporary models of Internet traffic as a function of the level of offered network load. We limit our study to the dominant and most stressful class of Internet traffic: bursty HTTP flows. We find that for HTTP flows (1) using SACK only improves performance for larger-than-typical HTTP transfers, (2) unless congestion is a serious concern (i.e., unless average link utilization is approximately 80% or higher), there is little benefit to using RED queue management, (3) above 80% link utilization there is potential benefit to using Adaptive RED with ECN marking, however, complex performance trade-offs exist and the results are dependent on parameter settings.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Congestion control; Simulations; TCP

1. Introduction

Improvements to TCP's error recovery and congestion control/avoidance mechanisms have been a mainstay of contemporary networking research. Representative innovations in error control include the use of fast transmissions (TCP Reno), fast retransmission in the face of multiple losses (TCP New Reno), and selective acknowledgements (TCP SACK). Representative innovations in congestion control include the congestion avoidance and the additive-increase, multiplicative decrease algorithm (TCP Tahoe), fast recovery (TCP Reno), early congestion detection in routers (RED), and explicit congestion notification (ECN).

While simulation studies have shown performance improvements with the addition of each new piece of networking technology, the evaluations have often been simplistic and have largely considered each improvement in isolation. In this paper, which is based on our preliminary work [1], we assess the state of congestion control and error recovery proposed for the Internet through a controlled study that considers the integration of standards-track TCP error recovery and TCP/router-based congestion control. The goal is to examine and quantify the benefits of deploying these technologies for contemporary models of Internet traffic as a function of the level of offered network load.

Although numerous modifications to TCP have been proposed, we limit our consideration to proposals that have either been formally standardized or are being proposed for standardization as these are the most likely to be widely deployed on the Internet. We report the results of an extensive study into the impact of using TCP Reno versus TCP SACK in networks employing drop-tail queue management versus random early detection queue management (specifically, adaptive, gentle RED) in routers. We assess the performance of combinations of error recovery and router queue management through traditional

* 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 © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comcom.2006.03.001

network-centric measures of performance such as link utilization and loss-rates, as well as through user-centric measures of performance such as response time distributions for Web request–response transactions.

Considering both network and end-user-centric measures of performance, for bursty HTTP traffic sources:

- There is no difference in performance between Reno and SACK for typical short-lived HTTP flows independent of load and pairing with queue management algorithm. For larger flows, using SACK does improve performance over that of Reno.
- Adaptive RED (ARED) with ECN marking performs better than ARED with packet dropping, and the value of marking increases as the offered load increases. ECN also offers more significant gains in performance when the target delay parameter is small (5 ms).
- However, unless congestion is a serious concern (i.e., for average link utilizations of 80% or higher with bursty sources), there is little benefit to using RED queue management in routers.
- In congested networks, a fundamental trade-off exists between optimizing response time performance of short responses versus long responses. If one favors short responses, ARED with ECN marking and a small target delay (5 ms) performs better than drop-tail independent of the level of congestion (at the expense of long responses). This conclusion should be tempered with the caution that ARED performance is quite sensitive to parameter settings. At all loads there is little difference between the performance of drop-tail and ARED with ECN and a larger target delay (60 ms). If one favors long responses, drop-tail performs better than ARED with ECN marking. In addition, as load increases drop-tail also results in higher link utilization.

In total, we conclude that for user-centric measures of performance, router-based congestion control (active queue management), especially when integrated with end-system protocols (explicit congestion notification), has a greater impact on performance than protocol improvements for error recovery. However, for lightly to moderately loaded networks (e.g., 50% average utilization) neither queue management nor protocol improvements significantly impact performance.

The following sections provide background on the specific error recovery and congestion control improvements we are considering as well as a summary of past evaluations of each. Section 5 explains our experimental methodology, and Section 6 presents a summary of our main results.

2. Background – congestion control and avoidance

2.1. TCP Reno

TCP Reno is de facto standard TCP implementation. Reno congestion control consists of two major phases: *slow-start* and *congestion avoidance*. In addition, congestion control is integrated with related *fast retransmit* and *fast recovery* error recovery mechanisms.

Slow-start restricts the rate of packets entering the network to the same rate that acknowledgments (ACKs) return from the receiver. The receiver sends cumulative ACKs which acknowledge the receipt of all bytes up to the sequence number carried in the ACK (i.e., the receiver sends the sequence number of the next packet it expects to receive). The congestion window, *cwnd*, controls the rate at which data is transmitted and loosely represents the amount of unacknowledged data in the network. For every ACK received during slow-start, *cwnd* is incremented by one segment.

The congestion avoidance phase conservatively probes for additional bandwidth. The slow-start threshold, *ssthresh*, is the threshold for moving from slow-start to congestion avoidance. When *cwnd* is greater than *ssthresh*, slow-start ends and congestion avoidance begins. Once in congestion avoidance, *cwnd* is increased by one segment every round-trip time, or as commonly implemented, by $1/cwnd$ of a segment for each ACK received.

When packet loss occurs, *ssthresh* is set to 1/2 the current value of *cwnd*, and as explained below, the value of *cwnd* is set depending on how the loss was detected. In the simplest case, a loss that is detected by the expiration of a timer, the connection returns to slow-start after a loss.

2.2. Random early detection

Internet routers today employ traditional FIFO queuing (called ''drop-tail'' queue management). Random Early Detection (RED) is a router-based congestion control mechanism that seeks to reduce the long-term average queue length in routers [2]. A RED router monitors queue length statistics and probabilistically drops arriving packets even though space exists to enqueue the packet. Such ''early drops'' are performed as a means of signaling TCP flows (and others that respond to packet loss as an indicator of congestion) that congestion is incipient in the router. Flows should reduce their transmission rate in response to loss (as outlined above) and thus prevent router queues from overflowing.

Under RED, routers compute a running weighted average queue length that is used to determine when to send congestion notifications back to end-systems. Congestion notification is referred to as ''marking'' a packet. For standard TCP end-systems, a RED router drops marked packets to signal congestion through packet loss. If the TCP end-system understands packet-marking, a RED router marks and then forwards the marked packet.

RED's marking algorithm depends on the average queue size and two thresholds, min_{th} and max_{th}. When the average queue size is below min_{th}, RED acts like a drop-tail router and forwards all packets with no modifications. When the average queue size is between min_{th} and max_{th}, RED marks incoming packets with a certain

probability. When the average queue size is greater than max_{th}, all incoming packets are dropped. The more packets a flow 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 flow occupies.

The RED thresholds min_{th} and max_{th}, the maximum drop probability max_p, and the weight given to new queue size measurements w_q, play a large role in how the queue is managed. Recommendations [3] on setting these RED parameters specify that max_{th} should be set to three times min_{th}, w_q should be set to 0.002, or 1/512, and max_p should be 10%.

2.3. Adaptive RED

Adaptive RED (ARED) [4] is a modification to RED which addresses the difficulty of setting appropriate RED parameters. ARED adapts the value of max_p so that the average queue size is halfway between min_{th} and max_{th}. The maximum drop probability, max_p is kept between 1% and 50% and is adapted gradually. ARED includes another modification to RED, called "gentle RED" [5]. In gentle RED, when the average queue size is between max_{th} and $2 \times max_{th}$, 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}. When the average queue size is between max_{th} and $2 \times max_{th}$, selected packets are no longer marked, but always dropped.

ARED's developers provide guidelines for the automatic setting of min_{th}, max_{th}, and w_q. Setting min_{th} results in a trade-off between throughput and delay. Larger queues increase throughput, but at the cost of higher delays. The rule of thumb suggested by the authors is that the average queuing delay should only be a fraction of the round-trip time (RTT). If the target average queuing delay is $target_{delay}$ and C is the link capacity in packets per second, then min_{th} should be set to $target_{delay} \times C/2$. The guideline for setting max_{th} is that it should be $3 \times min_{th}$, resulting in a target average queue size of $2 \times min_{th}$. The weighting factor w_q controls how fast new measurements of the queue affect the average queue size and should be smaller for higher speed links. This is because a given number of packet arrivals on a fast link represents a smaller fraction of the RTT than for a slower link. It is suggested that w_q be set as a function of the link bandwidth, specifically, $1 - \exp(-1/C)$.

Although many extensions to RED have been proposed, we use ARED in our study because it includes the "gentle" mode, which is now the recommended method of using RED, and because we believe ARED is the most likely RED variant to be standardized and deployed.

2.4. Explicit congestion notification

Explicit congestion notification (ECN) [6,7] is an optimization of active queue management that allows routers to notify end systems when congestion is present in the network. When an ECN-capable router detects that its average queue length has reached a threshold, it marks packets by setting the CE ("congestion experienced") bit in the packets' TCP headers. (The decision of which packets to mark depends on the queue length monitoring algorithm in the router.) When an ECN-capable receiver sees a packet with its CE bit set, an ACK with its ECN-Echo bit set is returned to the sender. Upon receiving an ACK with the ECN-Echo bit set, the sender reacts in the same way as it would react to a packet loss (i.e., by halving the congestion window). Ramakrishnan and Floyd [7] recommend that since an ECN notification is not an indication of packet loss, the congestion window should only be decreased once per RTT, unless packet loss does occur. A TCP sender implementing ECN thus receives two notifications of congestion, ECN and packet loss. This allows senders to be more adaptive to changing network conditions.

ECN is recommended for use in routers that monitor their average queue lengths over time (e.g., routers running RED), rather than those that can only measure instantaneous queue lengths. This allows short bursts of packets without triggering congestion notifications.

3. Background – error recovery

3.1. Error recovery in TCP Reno

TCP Reno provides two methods of detecting packet loss: the expiration of a timer and the receipt of three duplicate acknowledgements. Whenever a new packet is sent, the retransmission timer (RTO) is set. If the RTO expires before the packet is acknowledged, the packet is assumed to be lost. When the RTO expires, the packet is retransmitted, *ssthresh* is set to 1/2 *cwnd*, *cwnd* is set to 1 segment, and the connection re-enters slow-start.

Fast retransmit specifies that a packet can be assumed lost if three duplicate ACKs are received. This allows TCP Reno to avoid a lengthy timeout during which no data is transferred. When packet loss is detected via three duplicate ACKs, fast recovery is entered. In fast recovery, *ssthresh* is set to 1/2 *cwnd*, and *cwnd* is set to *ssthresh* + 3. For each additional duplicate ACK received, *cwnd* is incremented by 1 segment. New segments can be sent as long as *cwnd* allows. When the first ACK arrives for the retransmitted packet, *cwnd* is set back to *ssthresh*. Once the lost packet has been acknowledged, TCP leaves fast recovery and returns to congestion avoidance.

Fast recovery also provides a transition from slow-start to congestion avoidance. If a sender is in slow-start and detects packet loss through three duplicate ACKs, after the loss has been recovered, congestion avoidance is entered. The only other way to enter congestion avoidance is if $cwnd > ssthresh$. In many cases, though, the initial value of *ssthresh* is set to a very large value, so packet loss is often the only trigger to enter congestion avoidance.

TCP Reno can only recover from one packet loss during fast retransmit and fast recovery. Additional packet losses in the same window may require that the RTO expire before being retransmitted. The exception is when *cwnd* is greater than 10 segments. In this case, Reno could recover from two packet losses by entering fast recovery twice in succession. This causes *cwnd* to effectively be reduced by 75% in two RTTs [8].

3.2. Selective acknowledgments

A recent addition to the standard TCP implementation is the selective acknowledgment option (SACK) [8,9]. The SACK option contains up to four (or three, if RFC 1323 timestamps are used) SACK blocks, which specify contiguous blocks of received data. Each SACK block consists of two sequence numbers which delimit the range of data the receiver holds. A receiver can add the SACK option to ACKs it sends back to a SACK-enabled sender. In the case of multiple losses within a window, the sender can infer which packets have been lost and should be retransmitted using the information in the SACK blocks.

A SACK-enabled sender can retransmit multiple lost packets in one RTT. The SACK recovery algorithm only operates once fast recovery has been entered via the receipt of three duplicate ACKs. Whenever an ACK with new information is received, the sender adds to a list of packets (called the ''scoreboard'') that have been acknowledged. These packets have sequence numbers past the current value of the highest cumulative ACK. At the beginning of fast recovery, the sender estimates the amount of unacknowledged data ''in the pipe'' based on the highest packet sent, the highest ACK received, and the number of packets in the scoreboard. This estimate is saved in the variable *pipe*. Each time a packet is sent the value of *pipe* is incremented. The *pipe* counter is decremented whenever a duplicate ACK arrives with a SACK block indicating that new data was received. When *pipe* is less than *cwnd*, the sender can either send retransmissions or transmit new data. When the sender is allowed to send data, it first looks at the scoreboard and sends any packets needed to fill gaps at the receiver. If there are no such packets, then the sender can transmit new data. The sender leaves fast recovery when all of the data that was unacknowledged at the beginning of fast recovery has been acknowledged.

Allowing up to three SACK blocks per SACK option ensures that each SACK block is transmitted in at least three ACKs, providing some amount of robustness in the face of packet loss.

4. Related work

TCP SACK has been studied by many and the reviews are mixed. Some researchers [10] have shown that using TCP SACK does not hurt competing TCP Reno flows. It was also shown that TCP SACK flows avoid some retransmissions and receive better throughput than if TCP Reno had been used [8]. Further research [11,12] has narrowed the benefits of TCP SACK to environments with medium loss levels. With low levels of loss, TCP Reno can use fast retransmit and fast recovery to repair lost packets without experiencing a timeout. With high levels of loss, TCP SACK flows also experience timeouts because timeouts are necessary even in TCP SACK to recover from lost retransmissions or the loss of several consecutive ACKs. There is also the potential with TCP SACK to cause large persistent queues when coupled with drop-tail routers [13]. TCP SACK can avoid timeouts, which hurt a flow's performance, but also serve to help drain the queue during times of severe congestion. Finally, a study of a busy web server [14] found that although 50% of the packet drops were detected via timeouts, TCP SACK would have only avoided 4% of the timeouts.

RED has been put forth by the IETF as an essential part of congestion control in the Internet [15]. Subsequent studies [16,17] of RED in the context of web traffic has shown that the use of RED routers offers little or no improvement over drop-tail routers. Additionally, these studies also showed that setting the parameters of RED for best performance was non-intuitive. One study [18] looked at the combination of TCP SACK running over RED routers. They found that the benefit of TCP SACK was diminished when coupled with RED. Since RED smoothes packet loss over many flows, there were fewer packets drops for a single flow in the same window. Because of this, TCP Reno's fast retransmit and fast recovery mechanisms could recover from the loss without needing SACK information.

RED coupled with ECN marking has the same parameter setting difficulty as RED with dropping, but should offer some benefit to TCP flows. Studies of RED with ECN [19] have found that with bulk transfer traffic, using ECN results in low levels of loss. With web-like traffic, the advantage of using ECN increased with the level of congestion in the network. Another study of RED with ECN marking [20] found that using ECN did reduce the number of packet drops, but did not necessarily improve the goodput received by the flows.

There have been many studies that look at TCP SACK and RED (with and without ECN marking) individually. We extend these previous studies to analyze the interactions of TCP SACK, RED, and ECN marking and their effects on two-way web traffic. Our traffic model (and RTT model) is based on the analysis of recent Internet traffic traces, combining both short-lived and long-lived flows.

5. Methodology

5.1. Experimental setup

HTTP traffic consists of communication between web clients and web servers. In non-persistent 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 ran simulations in NS-2[1] [21] with varying levels of two-way non-persistent HTTP 1.0 traffic. These two-way traffic loads provide roughly equal levels of congestion on both the "forward" path (server-to-client) and "reverse" path (client-to-server). The following pairings of error recovery and queue management techniques were tested: Reno with drop-tail queuing in routers, Reno with ARED using packet drops, ECN-enabled Reno with ARED using ECN packet-marking, SACK with drop-tail, SACK with ARED using packet drops, and ECN-enabled SACK with ARED using packet marking. Table 1 presents a summary of the error-recovery and queue management pairings that were run.

The HTTP traffic we generate comes from the PackMime model [22] developed at Bell Labs. This model is based on the analysis of HTTP connections in a trace of a 100 Mbps Ethernet link connecting an enterprise network of 3000 hosts to the Internet [23,24]. The fundamental parameter of PackMime is the TCP/HTTP connection initiation rate (a parameter of the distribution of connection interarrival times). The model also includes distributions of the size of HTTP requests and the size of HTTP responses.

The request size distribution and the response size distribution are heavy-tailed (Figs. 1 and 2). There are a large number of small request sizes and a few very large request sizes. Almost 90% of the HTTP requests are under 1 KB and can fit in a single 1420-byte[2] packet. The largest request is almost 1 MB. Sixty percent of the HTTP responses can fit into one packet, and 90% of the responses fit into 10 packets, yet the largest response size is over 100 MB. Using this distribution, we will have many short-lived transfers, but also some very long-lived flows. These PackMime HTTP request and response size distributions are similar to those presented in a recent traffic measurement study [25].

In each experiment, we examine the behavior of traffic that consists of over 250,000 flows, with a total simulated time of 40 min. Crovella and Lipsky [26] have shown that simulations with heavy-tailed workloads (such as HTTP response sizes) can take a very long time to reach steady-state. Running a simulation for only a few minutes would take into account a small portion of the rich behavior of the traffic model. We ran our simulations as long as the available hardware and software environments would support to capture a significant amount of this behavior.

We implemented PackMime traffic generation in NS-2 using Full-TCP, which includes bi-directional TCP connection flows, connection setup, connection teardown, and variable packet sizes. In our implementation, one

Table 1
Experiments

TCP	Queuing method	Queue length (1250 B packets)	ARED delay $delay_{target}(min_{th},max_{th})$
Reno	Drop-tail	111 pckts (1.5 × BDP) 148 pckts (2 × BDP)	
Reno	ARED	370 pckts (5 × BDP)	5 ms (5, 15) 60 ms (30, 90)
Reno	ARED + ECN	370 pckts (5 × BDP)	5 ms (5, 15) 60 ms (30, 90)
SACK	Drop-tail	111 pckts (1.5 × BDP) 148 pckts (2 × BDP)	
SACK	ARED	370 pckts (5 × BDP)	5 ms (5, 15) 60 ms (30, 90)
SACK	ARED + ECN	370 pckts (5 × BDP)	5 ms (5, 15) 60 ms (30, 90)

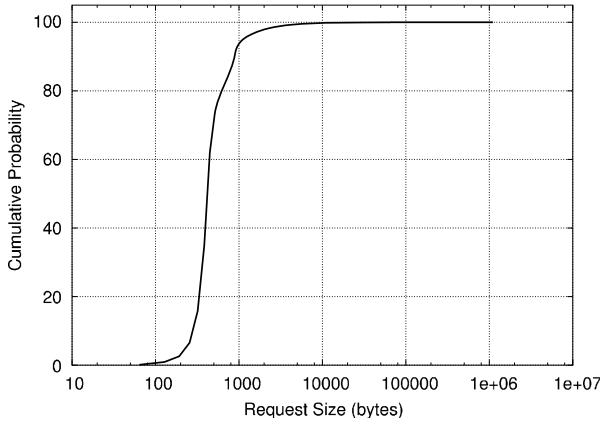


Fig. 1. Distribution of HTTP request sizes for a typical experiment (250,000 request/response pairs) using PackMime.

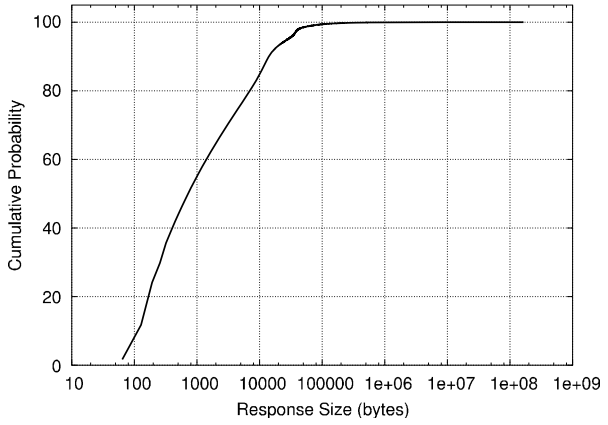


Fig. 2. Distribution of HTTP response sizes for a typical experiment (250,000 request/response pairs) using PackMime.

[1] In order to use SACK and ECN in Full-TCP, we had to modify the ns-2.1b9 source code. See http://www.cs.clemson.edu/~mweigle/ns/ for details.

[2] We assume an Ethernet MTU of 1500 bytes and use a TCP MSS of 1420 bytes, counting for 40 bytes of base TCP/IP header and 40 bytes maximum of TCP options.

PackMime node represents a cloud of HTTP clients or servers. The traffic load is driven by the user-supplied connection rate parameter, which is the number of new connections starting per second. The connection rate corresponding to each desired link loading was determined by a calibration procedure described below. New connections begin at their appointed time, whether or not any previous connection has completed.

The network we simulate consists of two clouds of web servers and clients positioned at each end of a 10 Mbps bottleneck link (Fig. 3). There is a 10 Mbps bottleneck link between the two routers, a 20 Mbps link between each PackMime cloud and its corresponding aggregation node, and a 100 Mbps link between each aggregation node and the nearest router. This configuration is designed to ensure that all congestion in the network occurs on the bottleneck link between the two routers.

The aggregation nodes in our simulations are NS-2 nodes that we developed called DelayBox to delay packets in the simulation. DelayBox is an NS analog to *dummynet* [27], which is used in FreeBSD network testbeds to delay packets. With DelayBox, packets from a TCP connection can be delayed before being passed on to the next node. This allows each TCP connection to experience a different minimum delay, and hence a different round-trip time, based on random sampling from a delay distribution. In our experiments DelayBox uses an empirical delay distribution from the PackMime model. This results in RTTs ranging from 1 ms to 3.5 s. The median RTT is 54 ms, the mean is 74 ms, and the 90th percentile is 117 ms. RTTs are assigned independently of request or response size, represent only propagation delay, and do not include queuing delays.

The mean packet size for the HTTP traffic (excluding pure ACKs, but including headers) is 1250 bytes. This includes the HTTP responses for the forward path and the HTTP requests for the reverse path. For a target bottleneck bandwidth of 10 Mbps, we compute the bandwidth-delay product (BDP) to be 74 1250-byte packets. In all cases, we set the maximum send window for each TCP connection to the BDP.

The simulation parameters used for TCP and Adaptive RED are listed in Tables 2 and 3.

5.2. Queue management

5.2.1. Drop-tail settings

Christiansen et al. [16] recommend a maximum queue size between $1.25 \times \text{BDP}$ and $2 \times \text{BDP}$ for reasonable response times for drop-tail queues. The maximum queue buffer sizes tested in our drop-tail experiments were $1.5 \times \text{BDP}$ and $2 \times \text{BDP}$.

Table 2
TCP parameters

TCP parameter	Value
Initial window size	2 segments
Timestamp option	False
TCP tick	10 ms
BSD 4.2 bug fix	True
Min RTO	1 s
Initial RTO	6 s
RFC 2988 RTT calculation	True
Delayed ACK	False
SACK block size	8 bytes
Max SACK blocks	3

Table 3
Adaptive RED parameters

ARED parameter	Value
Packet mode	True
Alpha	0.01
Beta	0.9
Interval for adaptations	500 ms
max max_p	0.5
min max_p	0.01

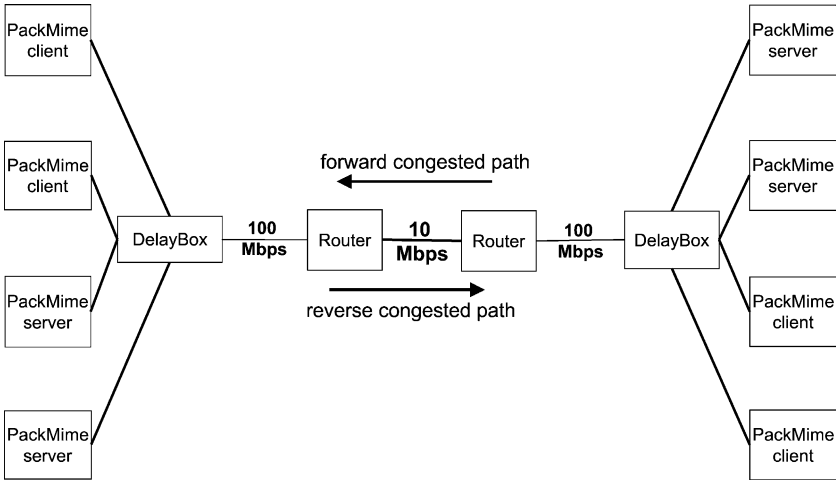


Fig. 3. Simulated network environment.

5.2.2. Adaptive RED settings

We ran sets of ARED experiments using the default ARED settings in NS-2 (target delay = 5 ms) and with parameters similar to those suggested by Christiansen, $min_{th} = 30$ and $max_{th} = 90$, giving a target delay of 60 ms. Note that in both cases, min_{th} and max_{th} are computed automatically based on the mean packet size, target delay, and link speed. The maximum router queue length was set to $5 \times$ BDP. This ensured that there would be no tail drops, in order to isolate the effects of ARED.

5.3. Performance metrics

In each experiment, we measured the HTTP response times (the time from sending an HTTP request to receiving the entire HTTP response), link utilization, throughput (number of bytes entering the bottleneck), average loss rate, average percentage of flows that experience loss, and average queue size. These summary statistics are given for the Reno experiments in Figs. 4–7 (as reported below, we found no large difference between Reno and SACK performance). HTTP response time is our main metric of performance. We report the CDFs of response times for responses that complete in 1500 ms or less. When discussing the CDFs, we discuss the percentage of flows that complete in a given amount of time. It is not the case that only small responses complete quickly and only large responses take a long time to complete. For example, between a 500 KB response that has a RTT of 1 ms and a 1 KB response that has a RTT of 1 s, the 500 KB response will likely complete before the smaller response.

5.4. Levels of offered load and data collection

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 our network at 100 Mbps and determined the PackMime connection rates (essentially the HTTP request rates) that will result in average link utilizations (in both forward and reverse directions) of 5, 6, 7, 8, 8.5, 9, 9.5, 10, and 10.5 Mbps. The connection rate that results in an average utilization of 8% of the (clearly uncongested) 100 Mbps link will be used to generate an offered load on the 10 Mbps link of 8 Mbps or 80% of 10 Mbps link. Note that this 80% load (i.e., the connection rate that results in 8 Mbps of traffic on the 100 Mbps link) will not actually

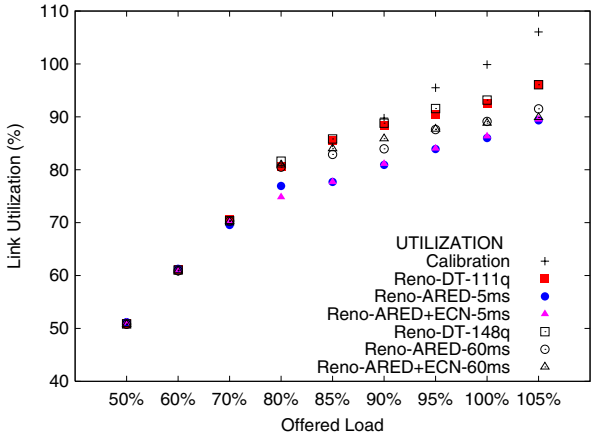


Fig. 4. Average link utilization.

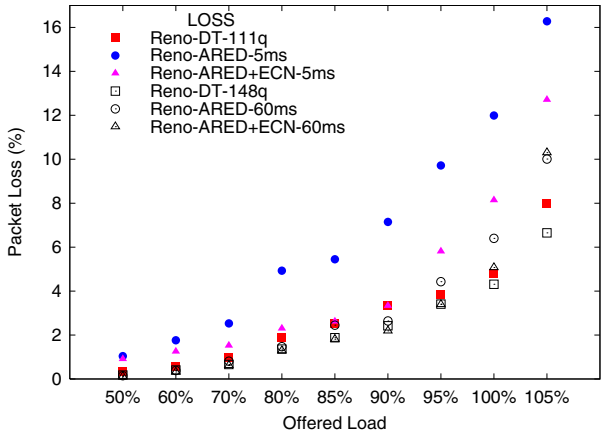


Fig. 5. Average loss rates.

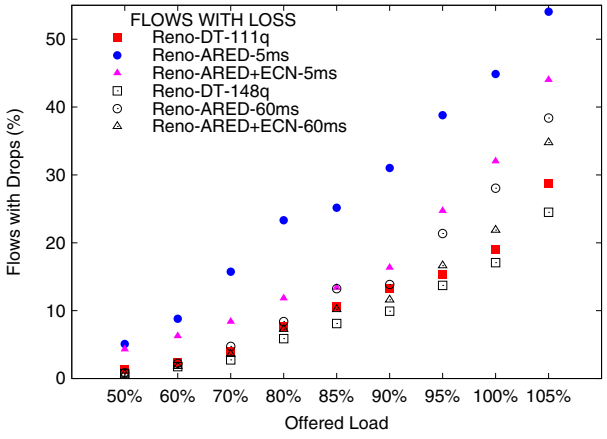


Fig. 6. Average percentage of flows experiencing drops.

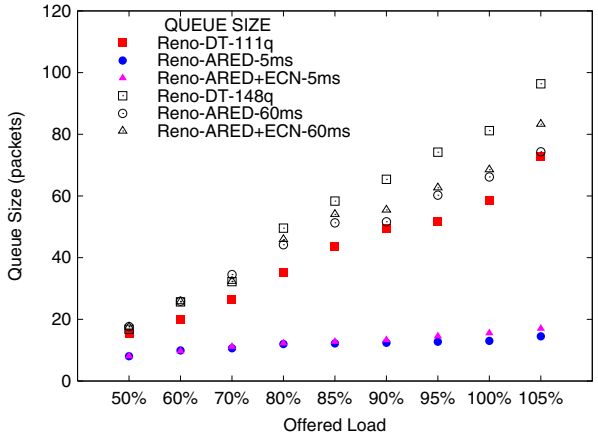


Fig. 7. Average queue sizes.

result in 8 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 scheme used. (And the link utilization achieved by a given level of offered load is a metric for comparing protocol/queue management combinations. See Fig. 4.)

Given the bursty nature of our HTTP traffic sources, we used a 120-s "warm-up" interval before collecting any data. After the warm-up interval, we allow the simulation to proceed until 250,000 HTTP request–response pairs have been completed. We also require that at least one 10 MB response has started a transfer before 1000 s after the warm-up period. This ensures that we will have some very long transfers in the network along with the typical short-lived web transfers.

6. Results

We first present the results for different queue management algorithms when paired with TCP Reno end-systems. Next, results for queue management algorithms paired with TCP SACK end-systems are presented and compared to the Reno results. Finally, the two best scenarios are compared. In the following response time CDF plots, the response times obtained in a calibration experiment with an uncongested 100 Mbps link are included for a baseline reference as this represents the best possible performance. Table 4 lists the labels we use to identify experiments.

We report here the results from experiments at offered loads of 80% and 105% (i.e., connection rates that would generate 8 and 10.5 Mbps of traffic, respectively, on an uncongested network). These offered loads correspond intuitively to "moderate" and "severe" levels of congestion on a 10 Mbps link. Complete results, for all offered loads, can be found in [28]. The results included here are also supported by the data from those experiments.

6.1. Reno + droptail

Fig. 8 shows the CDFs of response times for Reno-DT-111q and Reno-DT-148q. There is little performance difference between the two queue sizes, though there is a crossover point in the response times. The crossover is described here only for illustration purposes, since the difference is minimal. At 80% load, the crossover is at coordinate (700 ms, 80%). This means that for both DT-111q and DT-148q, 80% of the HTTP responses completed in 700 ms or less. For a given response time less than 700 ms, DT-111q produces a slightly higher percentage of responses that complete in that time or less than does DT-148q. For a given response time greater than 700 ms, DT-148q yields a slightly higher percentage of responses that complete in that time or less than DT-111q does. For simplicity, when comparing drop-tail to other queuing methods, we will show only DT-148q results.

Table 4
Summary of labels and abbreviations

Abbreviation	Description
DT-111q	Drop-tail with 111 packet queue (1.5 × BDP)
DT-148q	Drop-tail with 148 packet queue (2 × BDP)
ARED-5 ms	Adaptive RED with 5 ms target delay
ARED-60 ms	Adaptive RED with 60 ms target delay
ARED + ECN-5 ms	Adaptive RED with ECN and 5 ms target delay
ARED + ECN-60 ms	Adaptive RED with ECN and 60 ms target delay

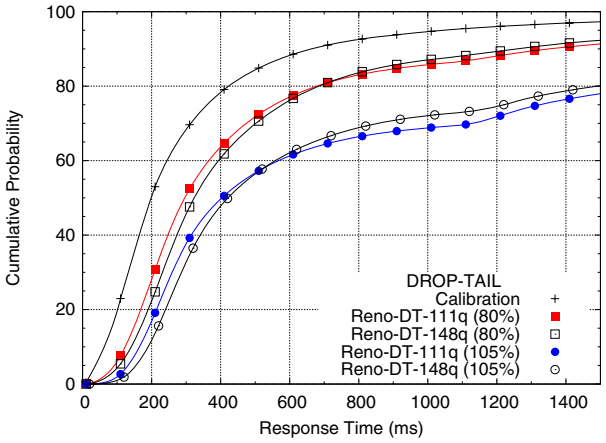


Fig. 8. Distribution of HTTP response times for drop-tail queuing.

6.2. Reno + adaptive RED

Response time CDFs for Reno-DT-148q, Reno-ARED-5ms, and Reno-ARED-60 ms are shown in Figs. 9 and 10. At almost all loads both of the drop-tail queues perform no worse than ARED-60 ms. There is a distinct crossover point between Reno-ARED-5ms and Reno-ARED-60 ms (and Reno-DT-148q and Reno-DT-111q) near 400 ms. This points to a trade-off between improving response times for some flows and causing worse response times for others. For responses that complete in less than 400 ms, ARED-5ms offers better performance. For responses that complete in over 400 ms, ARED-60 ms,

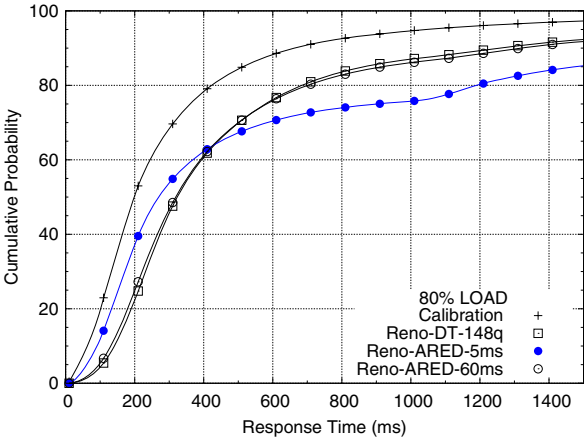


Fig. 9. Distribution of HTTP response times for drop-tail and ARED at 80% offered load.

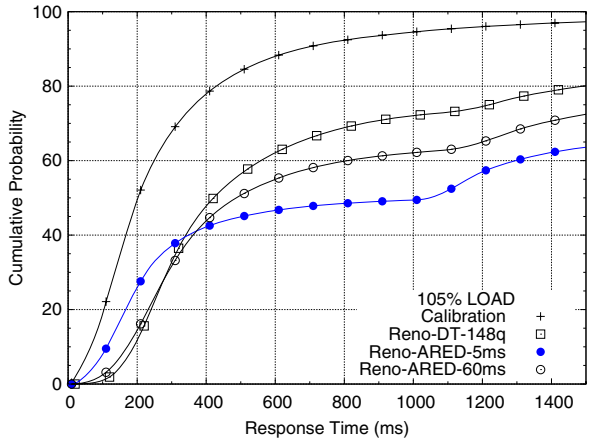


Fig. 10. Distribution of HTTP response times for drop-tail and ARED at 105% offered load.

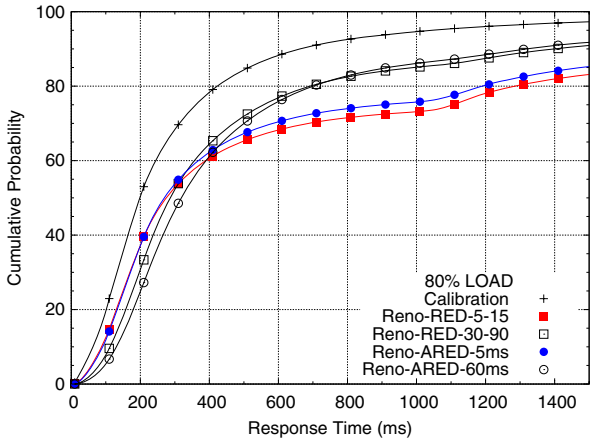


Fig. 11. Distribution of HTTP response times for RED and ARED at 80% offered load.

Reno-DT-111q, or Reno-DT-148q are preferable. As the load increases, the crossover remains near a response time of 400 ms, but the percentage of completed responses in that time or less decreases. Also, as load increases, the performance of ARED-5ms for longer responses is poor.

ARED-5ms keeps a much shorter average queue than ARED-60 ms (Fig. 7), but at the expense of longer response times for many responses. With ARED-5ms, many flows experience packet drops. Many of these connections are very short-lived, often consisting of a single packet (60% of responses consist of only one packet and 80% consist of five or fewer packets) and when they experience packet loss, they are forced to suffer a retransmission timeout, increasing their HTTP response times. For ARED-5ms, the CDF of response times levels off after the crossover point and does not increase much until after 1 s, which is the minimum RTO in our simulations. This indicates that a significant portion of the flows suffered timeouts.

As congestion increased toward severe levels, the response time benefits from the drop-tail queue became substantially greater. At 105% load about 60% of responses (those taking longer than 300–400 ms to complete) have better response times with the drop-tail queue while only 40% of responses are better with ARED-5ms. For this same 60% of responses, ARED-60 ms is also superior to ARED-5ms.

ARED should give better performance than the original RED design without the gentle option. At high loads, a significant number of packets arrive at the queue when the average queue size is greater than max_{th}. Without the gentle option, these packets would all be dropped, rather than being dropped probabilistically. To see the full difference between ARED including the gentle option and the original RED design we compared the results between the two designs at loads of 80% (Fig. 11) and 90% (Fig. 12). In these comparisons, two configurations of the original RED minimum and maximum thresholds were used: (5, 15) and (30, 90) which correspond roughly to the 5

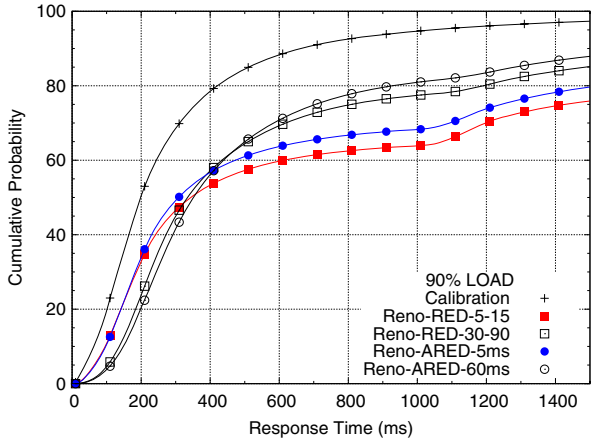


Fig. 12. Distribution of HTTP response times for RED and ARED at 90% offered load.

and 60 ms target queue lengths used for ARED. For the original RED experiments, we used the recommended settings of $w_q = 1/512$ and $max_p = 10\%$. For the ARED experiments, w_q was set automatically based on link speed to 0.001, and max_p was adapted between 1% and 50%.

Taken together, Figs. 11 and 12 show that the adaptation of max_p and the linear increase in drop probability from max_p to 1.0 in ARED is an improvement over the original RED design and that the improvement is more significant as the level of congestion increases.

6.3. Reno + adaptive RED + ECN

The full value of ARED is realized only when it is coupled with a more effective means of indicating congestion to the TCP endpoints than the implicit signal of a packet drop. ECN is the congestion signaling mechanism intended to be paired with ARED. Fig. 13 (80% load) and Fig. 14 (105% load) present the response time CDFs for Reno-ARED-5 ms, Reno-ARED-60 ms, Reno-ARED + ECN-5 ms, and Reno-ARED + ECN-60 ms. Up to 90% load,

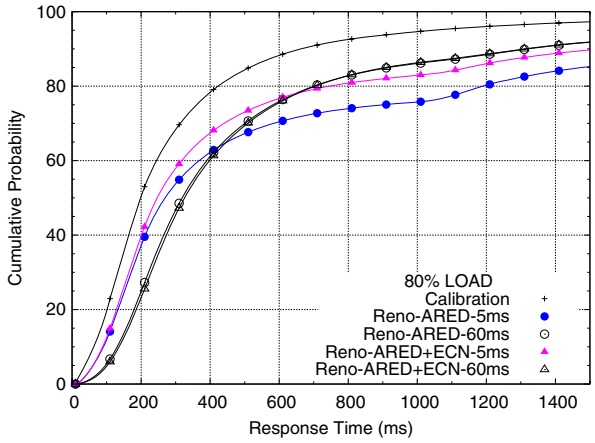


Fig. 13. Distribution of HTTP response times for ARED with and without ECN at 80% offered load.

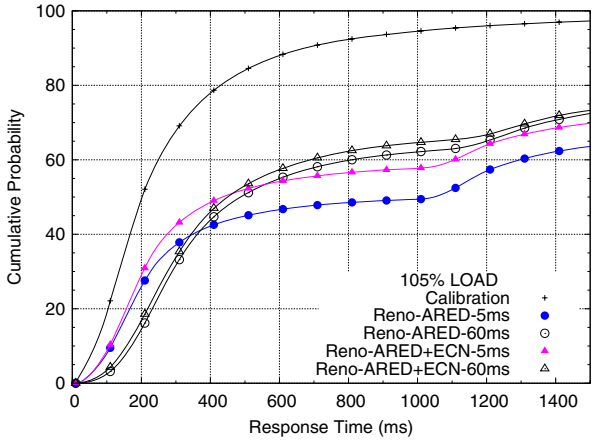


Fig. 14. Distribution of HTTP response times for ARED with and without ECN at 105% offered load.

ARED + ECN-5 ms delivers superior or equivalent performance for all response times. Further, ECN has a more significant benefit when the ARED target queue is small. As before, there is a trade-off where the 5 ms target delay setting performs better before the crossover point and the 60 ms setting performs slightly better after the crossover point. The trade-off when ECN is paired with ARED is much less significant. ARED + ECN-5ms does not see as much drop-off in performance after the crossover point. For the severely congested case, ECN provides even more advantages, especially when coupled with a small target delay.

Figs. 4–7 give summary network-centric performance measures. Overall, ARED + ECN-5 ms produces lower link utilization than ARED + ECN-60 ms. This trade-off between response time and link utilization is expected. ARED + ECN-5 ms keeps the average queue size small (Fig. 7) so that packets see low delay as they pass through the router. Flows that experience no packet loss (Fig. 6) should see very low queuing delays, and therefore, low response times. On the other hand, larger flows may receive ECN notifications and reduce their sending rates so that the queue drains more often. As expected, drop-tail results in the highest link utilization and the lowest drop rates.

We also ran a set of experiments with one-way traffic to see how well ECN and ARED would perform in a less complex environment. By one-way traffic, we mean that there was HTTP response traffic flowing in only one direction on the link between routers. Thus the reverse-path carrying ACKs and HTTP request packet was very lightly loaded. This means that ACKs are much less likely to be lost or "compressed" at the router queue and that (1) TCP senders will receive a smoother flow of ACKs to "clock" their output segments and (2) ECN congestion signals will be more timely. In some sense, this is a best case scenario for ARED with ECN. The results for this case are given in Fig. 15. With two-way traffic, performance is significantly reduced over the one-way case especially for the severe congestion at 105% offered load. These results clearly illustrate the importance of considering the effects of congestion in both directions of flow between TCP endpoints.

6.4. SACK

The experiments described above were repeated by pairing SACK with the different queue management mechanisms in place of Reno. Figs. 16 and 17 show a comparison between Reno and SACK based on response time CDFs when paired with drop-tail and ARED + ECN. Overall, SACK provides no better performance than Reno. When paired with ARED + ECN, SACK and Reno are essentially identical independent of load. When paired with drop-tail, Reno appears to provide somewhat superior response times especially when congestion is severe.

We expected to see improved response times with SACK over Reno with drop-tail queues. SACK should prevent some of the timeouts that Reno would have to experience before recovering from multiple drops in a window. Why is there not a large improvement with SACK over Reno

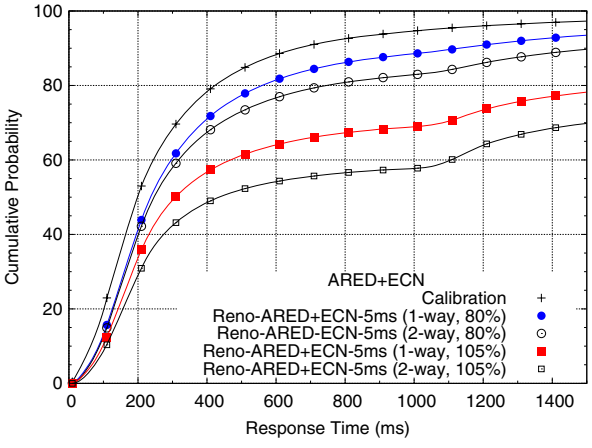


Fig. 15. Distribution of HTTP response times for ARED/ECN with 1-way and 2-way traffic.

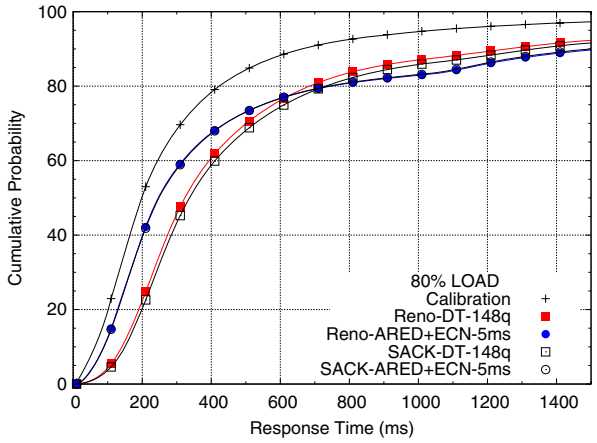


Fig. 16. Distribution of HTTP response times for drop-tail and ARED/ECN at 80% offered load.

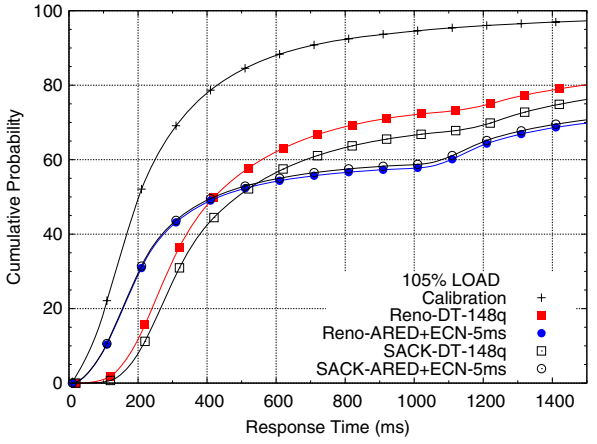


Fig. 17. Distribution of HTTP response times for drop-tail and ARED/ECN at 105% offered load.

with drop-tail? Recall that with HTTP most TCP connections only send a few segments and for loss events in these connections, SACK never comes into effect.

For longer flows (those with responses larger than 50 KB), we find improvement in response times when using SACK as opposed to Reno. In Fig. 18, we show response time CDFs up to 20,000 ms at 80% load for flows that contain HTTP responses larger than 50 KB. Drop-tail and ARED + ECN are the queue management techniques shown. Although we have show that ARED + ECN with a 5 ms target delay performs well for flows that have short responses and finish quickly, the performance for larger flows is very poor. Using SACK improves performance for ARED + ECN a bit, but it is still far below that of drop-tail. With drop-tail, SACK improves performance of these long flows over that of Reno. This performance improvement is attributed to the ability of the flows using SACK to avoid timeouts by recovering from multiple losses in a single fast retransmission period.

Even though the difference between Reno and SACK and drop-tail and ARED + ECN are dramatic for larger

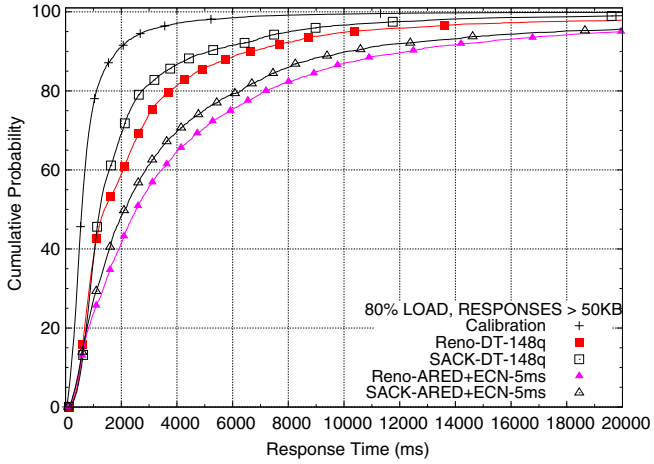


Fig. 18. Distribution of HTTP response times for flows with responses larger than 50 KB for Reno and SACK with drop-tail and ARED/ECN at 80% offered load.

flows, these flows only represent 1.4% of the total flows (but almost half of the total bytes) in our workload. These performance differences are only revealed when we look at the larger flows in isolation.

6.5. Drop-tail vs. ARED + ECN

Here, we compare the performance of the two "best" error recovery and queue management combinations for overall web traffic. Figs. 19 and 20 present the response time CDFs for Reno-DT-148q and Reno-ARED + ECN-5 ms. With these scenarios, the fundamental trade-off between improving response times for some responses and making them worse for other responses is clear. Further, the extent of the trade-off is quite dependent on the level of congestion. At 80% load, Reno-ARED + ECN-5 ms offers better response-time performance for nearly 75% of responses but marginally worse for the rest. At levels of severe congestion the improvements in response times for Reno-ARED + ECN-5 ms apply to around 50% of responses while the response times of the other 50% are degraded significant-

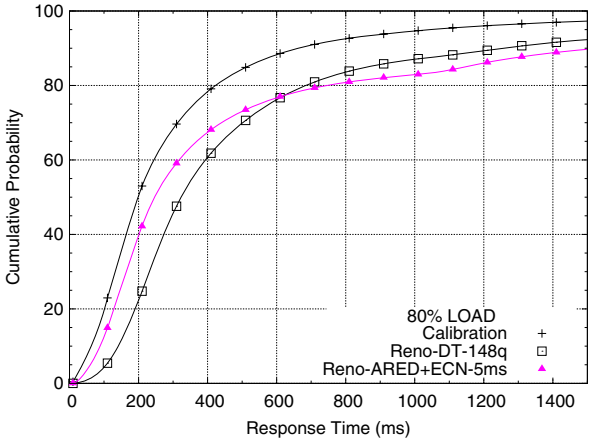


Fig. 19. Distribution of HTTP response times for best drop-tail and best AQM at 80% offered load.

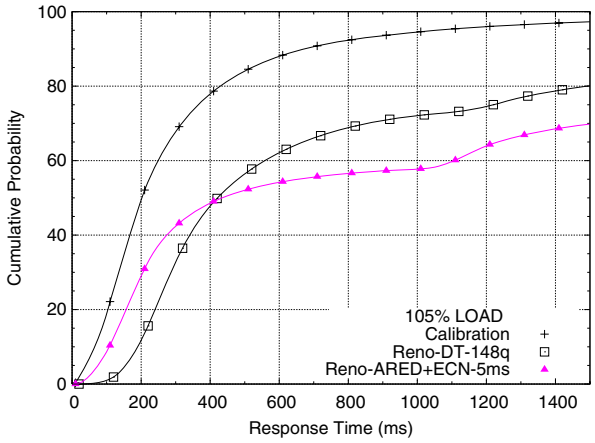


Fig. 20. Distribution of HTTP response times for best drop-tail and best AQM at 105% offered load.

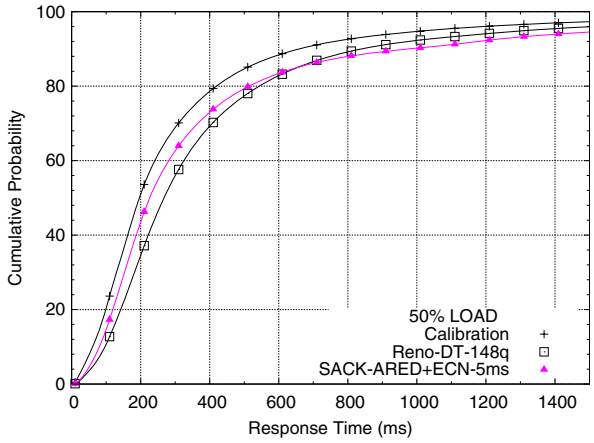


Fig. 21. Distribution of HTTP response times for best drop-tail and best AQM at 50% offered load.

ly, Reno-DT-148q and Reno-ARED + ECN-5 ms are on the opposite ends of the queue-management spectrum, yet, they each offer better HTTP response times for different portions of the total set of responses. Further complicating the trade-off is the result that Reno-DT-148q gives higher link utilization along with a high average queue size, while Reno-ARED + ECN-5 ms gives low average queue sizes, but also lower link utilization.

6.6. Performance for offered loads less than 80%

We have presented results for offered loads of 80% and 105% as these are representative of moderately and severely congested networks. For load levels lower than 80%, there is an advantage for ARED + ECN-5 ms over DT-148q for shorter responses. The same trade-off is present for 50–80% load as with loads over 80% and hence is a fundamental trade-off. ARED + ECN-5 ms has lower average queue sizes and the corresponding better response time performance for shorter responses, with similar link utilization as DT-148q. DT-148q performs better for responses that take longer than 600 ms to complete.

Much below 80% offered load, SACK and Reno have identical performance, and ARED + ECN performs only marginally better than drop-tail. At these loads, there is no difference in link utilization between any of the queue management techniques. There is also very little packet loss (no average loss over 3%). For loads under 80%, given the complexity of implementing RED, there is no compelling reason to use ARED + ECN over drop-tail. Fig. 21 is representative of these results, comparing the performance for our best queue management/protocol combination at 50% offered load.

7. Conclusions

Our results provide an evaluation of the state-of-the-art in TCP error recovery and congestion control in the context of Internet traffic composed of "mice" and "elephants." We used bursty HTTP traffic sources generating a traffic mix with a majority of flows sending few segments (less than 5), a small number sending many segments (more than 50), and a number in the [5, 50] segment range.

Using NS-2 simulations, we evaluated how well various pairings of TCP Reno, TCP SACK, drop-tail, Adaptive RED (with both packet-marking and dropping), and ECN perform in the context of HTTP traffic. Our primary metric of performance was the response time to deliver each HTTP object requested along with secondary metrics of link utilization, router queue size and packet loss percentage. Our main conclusions based on HTTP traffic sources and these metrics are:

- There is no difference in performance between Reno and SACK for typical short-lived HTTP flows independent of load and pairing with queue management algorithm. For larger flows, using SACK does improve performance over that of Reno.
- As expected, ARED with ECN marking performs better than ARED with packet dropping and the value of ECN marking increases as the offered load increases. ECN also offers more significant gains in performance when the target delay is small (5 ms).
- Unless congestion is a serious concern (i.e., for average link utilizations of 80% or higher with bursty sources), there is little benefit to using RED queue management in routers.
- ARED with ECN marking and a small target delay (5 ms) performs better than drop-tail with $2 \times$ BDP queue size at offered loads having moderate levels of congestion (80% load). This finding should be tempered with the caution that, like RED, ARED is also sensitive to parameter settings.
- At loads that cause severe congestion, there is a complex performance trade-off between drop-tail with $2 \times$ BDP queue size and ARED with ECN at a small target delay. ARED can improve the response time of about half the responses but worsens the other half. Link utilization is significantly better with drop-tail.

- At all loads there is little difference between the performance of drop-tail with $2 \times$ BDP queue size and ARED with ECN marking and a larger target delay (60 ms).

Acknowledgements

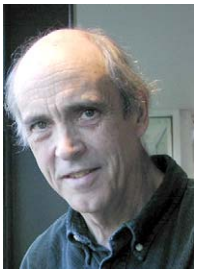
We thank Jin Cao, Bill Cleveland, Yuan Gao, Dong Lin, and Don Sun from Bell Labs for help in implementing their PackMime model in NS-2. This work supported in parts by grants from the National Science Foundation (grants ITR-0082870, CCR-0208924, EIA-0303590, and ANI-0323648), Cisco Systems Inc., and the IBM Corporation.

References

[1] M. C. Weigle, K. Jeffay, F. D. 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.
[2] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM Transactions on Networking 1 (4) (1993) 397–413.
[3] S. Floyd. RED: discussions of setting parameters, <http://www.icir.org/floyd/REDparameters.txt/> (Nov. 1997).
[4] S. Floyd, R. Gummadi, S. Shenker, Adaptive RED: an algorithm for increasing the robustness of RED's active queue management, Technical Note (August 2001).
[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, TCP and explicit congestion notification, ACM Computer Communication Review 24 (5) (1994) 10–23.
[7] K. K. Ramakrishnan, S. Floyd. A proposal to add explicit congestion notification (ECN) to IP, RFC 2481, experimental (January 1999).
[8] K. Fall, S. Floyd, Simulation-based comparisons of Tahoe, Reno, and SACK TCP, ACM Computer Communication Review 26 (3) (1996) 5–21.
[9] M. Mathis, J. Mahdivi, S. Floyd, A. Romanow, TCP selective acknowledgement options, RFC 2018, October 1996.
[10] S. Floyd, Issues of TCP with SACK, Tech. rep., LBL (March 1996).
[11] R. Bruyeron, B. Hemon, L. Zhang, Experimentations with TCP selective acknowledgement, ACM Computer Communication Review 28 (2) (1998) 54–77.
[12] J. Bolliger, U. Hengartner, T. Gross, The effectiveness of end-to-end congestion control mechanisms. Tech. rep., ETH Zurich (February 1999).
[13] M. Mathis, J. Semke, J. Mahdivi, The macroscopic behavior of the tcp congestion avoidance algorithm, ACM Computer Communication Review 27 (3) (1997) 67–82.
[14] H. Balakrishnan, V. Padmanabhan, S. Seshan, M. Stemm, R. Katz. TCP behavior of a busy Internet server: analysis and improvements, in: Proceedings of IEEE INFOCOM, 1998.
[15] 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).
[16] M. Christiansen, K. Jeffay, D. Ott, F.D. Smith, Tuning RED for web traffic, IEEE/ACM Transactions on Networking 9 (3) (2001) 249–264.
[17] M. May, J.-C. Bolot, C. Diot, B. Lyles, Reasons not to deploy RED, in: Proceedings of IWQoS, London, UK, 1999.
[18] Y. Zhang, L. Qiu, Understanding the end-to-end performance impact of RED in a heterogeneous environment, Tech. Rep. 2000-1802, Cornell CS (July 2000).
[19] J.H. Salim, U. Ahmed, Performance evaluation of explicit congestion notification (ECN) in IP Networks, RFC 2884 (July 2000).
[20] K. Pentikousis, H. Badr, B. Kharmah, On the performance gains of TCP with ECN, in: Proceedings of the 2nd European Conference on Universal Multiservice Networks (ECUMN), Colmar, France, 2002.
[21] 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.
[22] 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.
[23] 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.
[24] J. Cao, W.S. Cleveland, D. Lin, D.X. Sun, On the nonstationarity of Internet traffic, in: Proceedings of ACM SIGMETRICS, Cambridge, MA, 2001, pp. 102–112.
[25] 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.
[26] M. Crovella, L. Lipsky, Long-lasting transient conditions in simulations with heavy-tailed workloads, in: Proceedings of the 1997 Winter Simulation Conference, 1997, pp. 1005–1012.
[27] L. Rizzo, Dummynet: a simple approach to the evaluation of network protocols, ACM Computer Communication Review 27 (1) (1997) 31–41.
[28] M.C. Weigle, Investigating the use of synchronized clocks in TCP congestion control. Ph.D. thesis, University of North Carolina at Chapel Hill (August 2003).

Michele Weigle is an Assistant Professor of Computer Science at Clemson University. She received her Ph.D. from the University of North Carolina at Chapel Hill in 2003. Her research interests include network protocol evalation, network simulation and modeling, Internet congestion control, and mobile ad-hoc networks.

Kevin Jeffay is S. Shepard Jones Distinguished Term Professor of Computer Science in the Department of Computer Science at the University of North Carolina at Chapel Hill. He also serves as the director of Undergraduate Studies for the department. He received his Ph.D. from the University of Washington in 1989. His research and teaching interests are in real-time systems, operating systems, networking, and multimedia systems. He and his students focus on the general problem of providing real-time computation and communication services across the Internet. Dr. Jeffay is currently the editor-in-chief of the ACM/Springer-Verlag journal Multimedia Systems. He is the program chair of the 2000 IEEE Real-Time Systems Symposium and the 2000 International Workshop on Network and perating System Support for Digital Audio and Video (NOSSDAV).



F. Donelson Smith is a Research Professor of Computer Science at the University of North Carolina at Chapel Hill. His current research interests include networking, operating systems, and distributed systems.