Smartacking: Improving TCP Performance from the Receiving End

Daniel K. Blandford, Sally A. Goldman, Sergey Gorinsky, Yan Zhou, and Daniel R. Dooly

***Abstract*— We present *smartacking*, a technique that improves performance of Transmission Control Protocol (TCP) via adaptive generation of acknowledgments (ACKs) at the receiver. When the bottleneck link is underutilized, the receiver transmits an ACK for each delivered data segment and thereby allows the connection to acquire the available capacity promptly. When the bottleneck link is at its capacity, the smartacking receiver sends ACKs with a lower frequency reducing the control traffic overhead and slowing down the congestion window growth to utilize the network capacity more effectively. To promote quick deployment of the technique, our primary implementation of smartacking modifies only the receiver. This implementation estimates the sender's congestion window using a novel algorithm of independent interest. We also consider different implementations of smartacking where the receiver relies on explicit assistance from the sender or network. Experiments in a wide variety of settings show beneficial impacts of smartacking on TCP performance, especially in environments with low levels of connection multiplexing on bottleneck links.**

***Index Terms*– TCP congestion control, acknowledgment frequency, receiver algorithms.**

I. Introduction

Internet hosts support efficient and fair sharing of traversed network links by participating in congestion control protocols that regulate the amount of transmitted data in response to the observed network performance. In particular, Internet applications routinely rely on Transmission Control Protocol (TCP) [5] which establishes a connection between two communicating end hosts and enforces a dynamic *congestion window* as an upper limit on the amount of sent data that the receiver has not yet acknowledged. Initially, the congestion window is small and hence prevents the sender from injecting a lot of data into the network. The window increases when a timely acknowledgment (ACK) confirms data delivery. However, if the stream of ACKs indicates loss, the TCP connection reduces the congestion window and thereby curbs the transmission.

TCP is an end-to-end protocol that does not require any network help beyond best-effort unreliable delivery of sent data segments and their ACKs. Whereas the end-to-end property improves the protocol extensibility, achieving an efficient and fair network usage exclusively from the end points is challenging. In TCP, the sender carries most of this burden by performing a variety of computations, e.g., maintaining a retransmission timer based on RTT (round-trip time) estimation, counting duplicate ACKs, and adjusting the congestion window.

In contrast to the sender, the receiving end of the TCP connection has a simple role: after delivery of a data segment, the receiver generates an ACK. The receiver can delay transmitting the ACK for up to half a second but has to send at least one ACK for each two delivered data segments. Lowering the frequency of ACKs reduces the protocol processing overhead and frees network resources for communicating data in the opposite direction. On the other hand, the delayed acknowledgment mechanism can disrupt the RTT estimation and slow down the growth of the congestion window.

In this paper, we present *smartacking*, a technique for adaptive generation of ACKs at the receiver. Beside reducing the amount of control traffic in the network, smartacking strives to improve the bottleneck link utilization by TCP. The algorithm is derived from an observation that the gap between delivered segments becomes close to uniform when traffic exhausts the capacity of the bottleneck link. Note that it is the receiver – not the sender – that can monitor the inter-segment arrival times. When the bottleneck link is at its capacity, the smartacking receiver sends ACKs with a lower frequency reducing the control traffic overhead and slowing down the congestion window growth. On the other hand, when the bottleneck link is underutilized, the receiver transmits an ACK for each delivered data segment and thereby preserves the rate at which TCP acquires the available capacity. Our experiments confirm that smartacking helps TCP to utilize the network capacity more effectively, including in topologies with asymmetric data flows and high bandwidth-delay products. We also show that smartacking TCP interacts fairly with standard TCP traffic.

Our primary implementation of smartacking modifies only the receiver. Although it would be easier to implement smartacking by changing the sender as well, we deliberately pursue a receiver-only implementation for two reasons. First, the receiver-only approach enables us to design mechanisms of independent relevance, such as an algorithm for estimating the congestion window at the receiver. Second, altering a protocol at only one end gives the enhancement better chances for attaining widespread deployment.

The rest of the paper is organized as follows. Section II provides a brief overview of related work. Section III describes smartacking. Section IV presents methodology of our evaluation. Section V reports experimental results. Finally, Section VI concludes the paper with a summary.

II. Related Work

TCP is not a rigid design and has evolved substantially, with congestion control becoming one of the most important additions. Since then numerous extensions – including Reno [19], SACK [24], Vegas [11], and NewReno [15] – have been proposed for TCP congestion control. However, the diverse

Manuscript received May 7, 2006; revised November 1, 2006.

This work was performed at Washington University in St. Louis and supported in part by U.S. National Science Foundation (NSF) grant CCR-9734940.

Daniel K. Blandford (dkb@andrew.cmu.edu) is with the Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA. Sally A. Goldman (sg@cs.wustl.edu) and Sergey Gorinsky (gorinsky@wustl.edu) are with the Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130, USA. Yan Zhou (zhou@cis.usouthal.edu) is at the School of Computer and Information Sciences, University of South Alabama, Mobile, AL 36688, USA. Daniel R. Dooly (ddooly@siue.edu) is with the Department of Computer Science, Southern Illinois University, Edwardsville, IL 62026, USA.

extensions employ the same tool to regulate transmission: a *congestion window* serves as an upper limit on the amount of transmitted data that the receiver has not yet acknowledged. Although the congestion window measures data in bytes, TCP passes data to the network in segments that do not exceed one MSS (maximum segment size). For example, MSS is sometimes set to 1460 bytes. Initially, the congestion window allows only a small amount of unacknowledged data. The initial window size commonly equals one or two MSS [4], [26], [29], [33]. The sender increases the window when timely acknowledgments confirm data delivery. However, if the stream of ACKs indicates loss, the sender decreases the congestion window. The goal of the window adjustment algorithm is to stabilize the transmission at a level that supports fair and efficient utilization of the network.

The window adjustment algorithm in the most widely deployed TCP versions operates in two modes – *slow start* and *congestion avoidance*. In the slow-start mode, the sender increases the congestion window by one MSS per non-duplicate ACK. When the bottleneck link is underutilized, the slow start doubles the window every RTT and thereby enables the connection to acquire the available capacity promptly. In the congestion-avoidance mode, a non-duplicate ACK increases the congestion window by an inverse of its current value; these adjustments grow the window by approximately one MSS per RTT and are supposed to supply convergence to the fair share of the bottleneck link capacity. Every TCP connection starts in the slow-start mode. Upon inference of congestion, the sender sets a threshold to the half of the current congestion window. When a triple duplicate ACK serves as the congestion indication, the sender reduces the window to the threshold and switches to the congestion-avoidance mode. If the congestion is inferred from a retransmission timeout, the sender reduces the window to one MSS and reenters the slow-start mode. The sender switches from the post-congestion slow start to the congestion-avoidance mode when the growing window reaches the threshold. Whereas the sender also performs a variety of other computations, the receiver's participation in the window adjustment is limited to transmitting an ACK upon delivery of a data segment.

The idea to delay ACKs at the receiver is far from being new. Even before TCP was enhanced with congestion control, Clark proposed postponing an ACK as a means to reduce TCP processing overhead and release network resources [12]. Later, delayed ACKs were found to be particularly beneficial in asymmetric networks [6]. In modern TCP versions, the receiver does not delay an ACK beyond half a second and transmits at least one ACK for each two delivered data segments [9]. Recently, Oliveira and Braun explored delayed ACKs as means to improve TCP performance and power consumption in multihop wireless networks [27].

Lowering the frequency of ACKs can result in a slower growth of the congestion window because the sender increases the window in response to non-duplicate ACKs. In the slow-start mode, reducing the window growth interferes with the objective of acquiring the available capacity promptly and is therefore undesirable. ABC (Appropriate Byte Counting) [2] and SABC (Scaled Appropriate Byte Counting) [3] are instantiations of a *byte counting* technique where the sender increases the congestion window in response to acknowledged bytes rather than acknowledged segments; ACKs that confirm delivery of more data trigger larger increases in the congestion window. Byte counting does not change the receiving end of the connection.

ACC (Acknowledgment Congestion Control) [6] is an alternative design that modifies both the sender and receiver. ACC also relies on support from network routers: when the router detects congestion on its output link, the router sets the ECN (Explicit Congestion Notification) [31] bit in the headers of packets forwarded to the link. The receiver checks the headers of delivered packets for set ECN bits and uses these observations to adjust a delay factor d. Possible values of d vary from 1 to a maximum determined by the congestion window which the sender communicates to the receiver explicitly. The receiver doubles d upon receiving a packet with the set ECN bit. While no such packets arrive, the receiver decreases the delay factor by one per RTT. In ACC, the receiver transmits one ACK for every d delivered data packets.

Related to the improvement of congestion control from the receiving end is an issue of receiver misbehavior [14], [32]. The receiver of a TCP connection can misbehave by providing feedback incorrectly in order to trick the sender into transmitting data at an unfairly high rate. In particular, it has been shown that by generating ACKs more frequently than prescribed by the protocol, the misbehaving receiver can substantially increase the reliable throughput of the connection at the expense of competing traffic [32]. In contrast to such misbehaving receivers, the receiver that follows smartacking as described in Section III generates ACKs *less* frequently with an objective of benefiting the overall efficiency and fairness of the network usage. Our experiments in Section V confirm that smartacking improves fairness of TCP.

III. SMARTACKING

In this section, we present *smartacking*, our technique for adaptive generation of ACKs at the TCP receiver. First, Section III-A discusses the main ideas behind smartacking. Then, Section III-B describes our implementation of the technique.

A. General Technique

In the slow-start mode, the sender of a TCP connection commonly injects a burst of data segments into the network. After the burst passes through the bottleneck link of the connection, gaps between the segments increase, and the burst spreads out [18], [21]–[23]. Eventually, the congestion window contains a large enough number of segments to utilize the bottleneck link fully. At this point, the segments arriving to the receiver are spread out the most over RTT of the connection. Also, since the bottleneck link reaches its capacity, continuing the aggressive growth of the congestion window does not improve the link utilization.

Note that it is the receiver – not the sender – that can monitor gaps between the arriving segments to determine the link saturation. This observation leads us to propose *smartacking*, a technique for improving TCP performance from the receiving end. In smartacking TCP, the receiver generates ACKs depending on its measurements of gaps between delivered data segments. When the bottleneck link is underutilized, the receiver transmits one ACK per segment and thereby preserves the rate at which TCP acquires the available capacity. When the bottleneck link is at its capacity, the receiver sends ACKs less frequently reducing the control traffic overhead and slowing down the congestion window growth.

For how long should the receiver delay ACKs when transmitting them with a lower frequency? Delaying an ACK creates a potential danger of underutilizing the bottleneck link. If the ACK is delayed for too long, the sender stops transmitting after the amount of unacknowledged data covers the congestion window, and subsequently the bottleneck link runs out of packets to forward. Hence, smartacking strives to generate ACKs with the lowest frequency that keeps the bottleneck link fully utilized. To achieve this goal, the receiver estimates a time interval `LastSegmentTime` after which it will receive the last segment allowed by the congestion window. Then, the receiver transmits the ACK with delay $\texttt{LastSegmentTime} - \texttt{RTT}$ so that the sender will receive the ACK just before exhausting the congestion window.

To estimate `LastSegmentTime`, the receiver maintains the following three variables measured in MSSs: (1) `CwndEst` is an estimate of the current congestion window, (2) `LastReceived` records the last received data, and (3) `LastACKed` denotes the last acknowledged data. $\texttt{CwndEst} + \texttt{LastACKed} - \texttt{LastReceived}$ represents the number of additional segments that the receiver expects from the current congestion window. The receiver also computes `ISAT`, an average of inter-segment arrival times, and estimates `LastSegmentTime` as $(\texttt{CwndEst} + \texttt{LastACKed} - \texttt{LastReceived}) \cdot \texttt{ISAT}$.

When the network capacity is underutilized, $\texttt{CwndEst} \cdot \texttt{ISAT} - \texttt{RTT}$ is negative, and the receiver transmits an ACK immediately. The immediate response allows the window to grow at the highest rate. On the other hand, as the bottleneck link becomes fully utilized, the ACK is delayed for up to $\texttt{LastSegmentTime} - \texttt{RTT}$, but no longer than the maximum time interval allowed by TCP. The delayed response slows the window growth without stalling the sender after the sender finishes transmitting its previous window of segments.

B. Receiver-Only Implementation

In addition to `ISAT` observable at the receiver, implementing the technique requires knowledge of the congestion window and round-trip time, i.e., information that is readily available at the sender. Hence, it might seem reasonable to implement smartacking in a distributed manner where the receiver sets its `CwndEst` and `RTT` variables to values communicated explicitly by the sender. We, however, deliberately pursue a receiver-only implementation of smartacking. This choice is chiefly due to deployment considerations. Experience shows that TCP extensions that upgrade only one of the communicating ends are more likely to enjoy wide adoption than those extensions that require changes at both ends. For example, both SACK [24] and NewReno [15] have been proposed for addressing multiple losses within the congestion window; although SACK is an earlier and technically superior solution than NewReno, the latter enjoyed wide deployment much quicker because NewReno upgrades only the sender whereas SACK modifies both the sender and receiver [28]. We hope that our receiver-only implementation will help smartacking to become widely deployed.

To implement smartacking with no support from the sender, the receiver can obtain most of the needed information either straightforwardly or using well-known estimation mechanisms. For example, the receiver has direct knowledge of `LastACKed` and `LastReceived`. In our implementation, the smartacking receiver computes `ISAT` as an exponentially weighted moving average (EWMA) with a gain of 0.25. The averaging ignores measurements triggered by the first segment arrival from any window. Also, the receiver ignores up to two consecutively measured values that are at least three times larger than the current `ISAT`. To compute `RTT`, the receiver reuses the standard TCP mechanism for RTT estimation; when the receiver is not transmitting its own data to the sender, the receiver estimates RTT by sending keep-alive messages once per second.

Our receiver-only implementation of smartacking faces a novel challenge of estimating the congestion window. Since different extensions of TCP use different algorithms to regulate the congestion window at the sender, the congestion window estimation at the receiver might also need to be extension-specific. While automatic detection of the sender's algorithm at the receiver is an interesting topic for future research, Figure 1 describes our mechanism for estimating the congestion window in NewReno that does not employ byte counting. The presented procedure `EstimateCwnd` has three phases corresponding to the initial slow-start, post-congestion slow-start, and congestion-avoidance modes of the sender. During the initial slow start, the receiver tracks the congestion window using the variable `StartingCwnd`. This variable is initially set to one segment and is incremented every time when the receiver transmits a non-duplicate ACK. After sending the third duplicate ACK, the receiver leaves the initial slow start and marks the transition by setting `StartingCwnd` to -1. Subsequently, the boolean variable `PastSsthresh` determines the current phase of the receiver: `True` corresponds to congestion avoidance, and `False` denotes post-congestion slow start. During congestion avoidance, the receiver tracks the congestion window using variable `CwndEst` and increases this variable by $1/\texttt{CwndEst}$ when transmitting a non-duplicate ACK. To initialize `CwndEst` when switching to congestion avoidance, as well as to estimate the congestion window in the post-congestion slow start, the receiver maintains a timer expiring once per RTT. When the timer expires at the end of a one-RTT interval, the receiver records the number of delivered in-order segments and transmitted non-duplicate ACKs in variables `NumSegmentsThisInterval` and `NumACKsThisInterval` respectively. Also, `NumACKsLastInterval` and `NumSegmentsLastInterval` save respectively the previous values of `NumACKsThisInterval` and `NumSegmentsThisInterval` recorded after the timeout one RTT earlier. Then, the receiver computes `CwndIfSlowStart` as a sum of `NumSegmentsLastInterval`, `NumACKsLastInterval`, and `NumACKsThisInterval` to estimate the congestion window during post-congestion slow start.

Our novel mechanism for the congestion window estimation plays an important role in our implementation of smartacking. However, this mechanism is also independently valuable because of its potential usefulness to other designs – such as ACC – where the receiver must know the congestion window.

IV. Evaluation Methodology

We evaluate smartacking and alternative approaches in a variety of network configurations. After Section IV-A describes the evaluated designs, Section IV-B presents our experimental setup. Then, Section IV-C discusses metrics for assessing the performance.

```
int EstimateCwnd()
  {if (StartingCwnd > -1) return StartingCwnd;
   if(PastSsthresh) return (int) CwndEst;
   Total = NumSegmentsLastInterval + NumACKsLastInterval + NumACKsThisInterval;
   if (Total > SsthreshEst) return max(SsthreshEst,NumSegmentsLastInterval) + 2; else return Total;}
```

When sending a non-duplicate ACK:

```
   NumACKsThisInterval++;
   if (StartingCwnd > -1) StartingCwnd++;     // initial slow-start phase
   if (CwndEst > 0) CwndEst += 1.0/CwndEst;
```

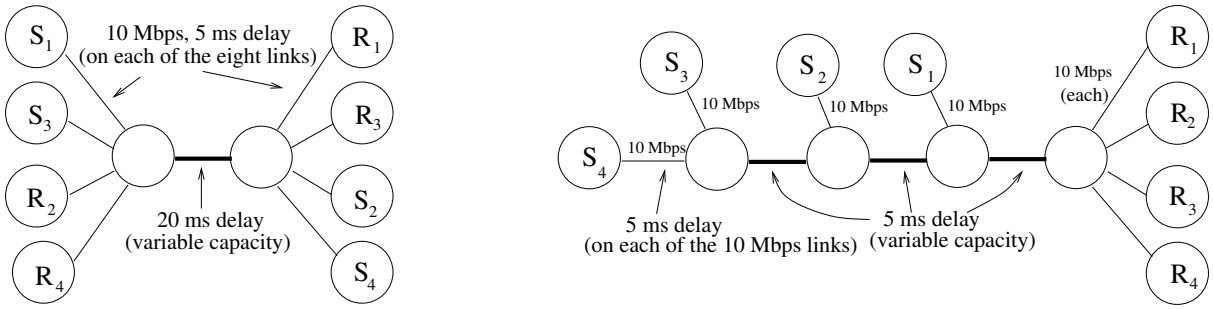
Whenever an in-order segment arrives:

```
  NumSegmentsThisInterval++
```

Once per RTT:

```
   if (!PastSsthresh && CwndIfSlowStart > NumSegmentsThisInterval + 2 && NumACKsLastInterval > 2
       && !SeenLossRecently)   // switching to the congestion-avoidance phase
      {PastSsthresh = True; CwndEst= NumSegmentsThisInterval + 1;}
   if (NumSegmentsThisInterval  == 0 && NumSegmentsLastInterval == 0) SeenLossThisInterval = True;
   if (SeenLossThisInterval && !SeenLossRecently)
      {SsthreshEst = max((int)(EstimateCwnd()/2), 2); StartingCwnd = -1; PastSsthresh = False;}
   if (NumSegmentsThisInterval > 1) SeenLossRecently = SeenLossThisInterval;
   SeenLossThisInterval = False;
   CwndIfSlowStart = NumSegmentsLastInterval + NumACKsLastInterval + NumACKsThisInterval;
   NumSegmentsLastInterval = NumSegmentsThisInterval; NumACKsLastInterval = NumACKsThisInterval;
   NumSegmentsThisInterval = NumACKsThisInterval = 0;
```

Fig. 1. Estimating the congestion window for TCP NewReno.



(a) Dumbbell topology (2-2 configuration)

(b) Parking-Lot topology

Fig. 2. Network topologies in our experiments.

A. Evaluated Designs

In Section III-B, we presented our implementation of the smartacking technique for TCP NewReno. We refer to this implementation as *NewReno with smartacking* and compare it with schemes that can be classified into two categories: existing designs and different implementations of smartacking.

Existing Designs. We include *NewReno* in our studies because NewReno is a widely deployed TCP extension that serves as a basis for our smartacking implementation. This design employs delayed ACKs as described in RFC 1122 [9]. In addition, we report results for *AckEvery*, a NewReno implementation that acknowledges every segment immediately. To compare smartacking with other proposals for overcoming the negative impact of delayed ACKs on the congestion window growth, we also evaluate *ACC* and *ABC*. Although we conducted experiments with SABC as well, we do not report these results here because of their similarity to the results reported for ABC.

Different Implementations of Smartacking. Due to deployment considerations, we deliberately pursued a receiver-only solution while implementing the smartacking technique in Section III-B. On the other hand, support from the sender or network can lead to more effective implementations of smartacking. In our experiments, we quantify potential benefits from such support mechanisms.

If the bottleneck link of a TCP connection is fully utilized, the smartacking receiver of the connection expects from arriving segments to be uniformly distributed over RTT. This ideal scenario happens when the connection is the only source of packets on the bottleneck link. However, when the saturated link carries other traffic as well, the receiver can observe a burstier distribution of the arriving segments: instead of being spread out over RTT, the segments from the congestion window can clump within a smaller portion of RTT. Then, the receiver can transmit ACKs prematurely. To alleviate the distortion, the sender and routers can employ mechanisms that improve packet mixing on shared bottleneck links.

Pacing is a mechanism where the sender smooths the bursty pattern of TCP operation by distributing segment transmissions throughout RTT [1], [34]. Pacing helps not only with improved packet mixing on shared links but also in networks with extremely low reverse-path capacities. Since a burst of segment arrivals can cause a burst of ACKs, the bursty transmission of data segments can congest the low-capacity reverse path with ACKs. Ensued queueing of ACKs increases the RTT estimate and shortens the delays of ACKs at the smartacking receiver. Pacing addresses this problem by reducing the possibility of the reverse-path congestion. However, spreading of data segments over the whole RTT would interfere with operation of smartacking: since the smartacking receiver detects the availability of the bottleneck capacity via gaps between

successive congestion windows, the spread of segments over the whole RTT would always trick the receiver into perceiving the bottleneck link as saturated even when the link is actually underutilized. Hence, we implement pacing by distributing the transmitted segments from a congestion window evenly over an interval equal to 0.9 RTT. We refer to the enhanced implementation as *NewReno with smartacking and pacing*.

Fair queueing of packets at the bottleneck link is a router-based mechanism that also can improve packet mixing [7], [13]. To assess the impact of fair queueing, we consider a simple round-robin algorithm that allocates a separate queue per flow and visits the queues in a round-robin manner to select a packet for transmission to the link. We use *NewReno with smartacking and fair queueing* to denote results achieved by our smartacking implementation in networks with round-robin scheduling.

An alternative to modifying the sender or routers is to implement smartacking differently at the receiver. Although the receiver cannot affect packet mixing directly, computations of ACK delays at the receiver can account for distortions in the distribution of arriving segments. Hence, we extend smartacking by introducing a parameter α to calculate the ACK delay as $\texttt{LastSegmentTime} - \alpha \cdot \texttt{RTT}$ rather than $\texttt{LastSegmentTime} - \texttt{RTT}$. When the arriving segments spread over the whole RTT, the receiver should use $\alpha = 1$ and compute the ACK delay in the same way as our smartacking implementation in Section III-B. However, when the bottleneck link is fully utilized but the segments still clump within a smaller portion of RTT, the receiver should keep the value of α between 0 and 1 to balance the underestimation of `LastSegmentTime`. Setting α to 0 corresponds to the scenario when the receiver transmits the ACK upon arrival of the last segment from the congestion window. We implement the following algorithm for adjustment of α. Initially, the receiver sets α to 1 and does not reduce its value until generating a third duplicate ACK. Subsequently, the receiver multiplies α by 0.9 whenever the congestion window estimate decreases. The receiver also maintains two auxiliary parameters β and γ initialized to 1 and 0 respectively. Parameter β takes real values between 0 and 1 and acts with respect to α in the same way as the threshold acts toward the congestion window at the sender. Parameter γ is an integer reflecting the frequency of ACKs confirming a single segment. Whenever the receiver transmits an ACK, γ is adjusted: if the ACK confirms a single segment, the receiver increments γ; otherwise, the receiver decrements γ. When γ becomes negative with its absolute value exceeding $4 \cdot \texttt{CwndEst}$ (i.e., when most ACKs confirm multiple segments), the receiver resets γ to 0 and updates α as follows: if $\beta > \alpha$, then the receiver increases α by 0.05; otherwise, the receiver decreases α by 0.01. We refer to this enhanced implementation as *NewReno with extended smartacking*.

B. Experimental Setup

In this section, we discuss network topologies, traffic patterns, and protocol settings in our experiments. Since we expect that smartacking provides most of its benefits in environments with small degrees of connection multiplexing on bottleneck links, we use few connections in most of the experiments. However, we also conduct some experiments with a large number of connections to study the impact of smartacking in networks with high levels of multiplexing. We follow suggestions from Floyd and Jacobson [16] and focus on settings where data flows traverse bottleneck links in both directions. This focus is important because congestion on the reverse path enables us to evaluate how smartacking performs under loss of ACKs. We also consider some simpler settings with no data flows on the reverse path. We examine each TCP extension discussed in Section IV-A in a bulk-transfer scenario where the sender communicates a large file containing ten thousand data segments. Packets that carry the data segments are 1000-byte long. The size of packets carrying ACKs is 40 bytes. The maximum value for an ACK delay is set to 200 ms (since our results for the maximum ACK delay of 100 ms are similar, we do not report them here). The start times of connections stagger by 100 ms. We set the advertised window of each receiver so that it never limits the congestion window at the sender. In some experiments, we add on-off UDP (User Datagram Protocol) [30] cross traffic to study the reaction to sharp changes in the available network capacity.

In most of our experiments, we use routers with *Droptail* buffer management because of their prevalence in the modern Internet. The size of a Droptail buffer is usually set to the half of the bandwidth-delay product. However, we also report results for our experiments with different buffer sizes. Finally, we briefly report on a small portion of results from our extensive experiments with networks of *RED* (Random Early Detection) routers [10], [17] where RED parameters are configured as $\min_{\text{th}} = 5$, $\max_{\text{th}} = 15$, $w_q = 0.002$, and $\max_p = 0.1$.

We conduct the experiments in two general network topologies. Figure 2a depicts a *Dumbbell topology* common in congestion control studies: each connection traverses three hops, and the link between the two routers is the middle hop for each connection. Figure 3 describes examined configurations of the Dumbbell topology. Figure 2b presents a *Parking-Lot topology* which is commonly used to assess fairness of congestion control protocols. Our configuration of the topology has four TCP connections and three bottleneck links. S_i and R_i refer respectively to the sender and receiver of connection i. S_1, S_2, S_3, and S_4 start transmitting their files at time 0, 100 ms, 200 ms, and 300 ms respectively.

We perform all the experiments in our own discrete-event packet-level simulator, which is similar to the popular NS-2 [25] but offers lower simulation times. To validate the simulator, we repeated some of the experiments in NS-2 and received results that are indistinguishable from those reported below.

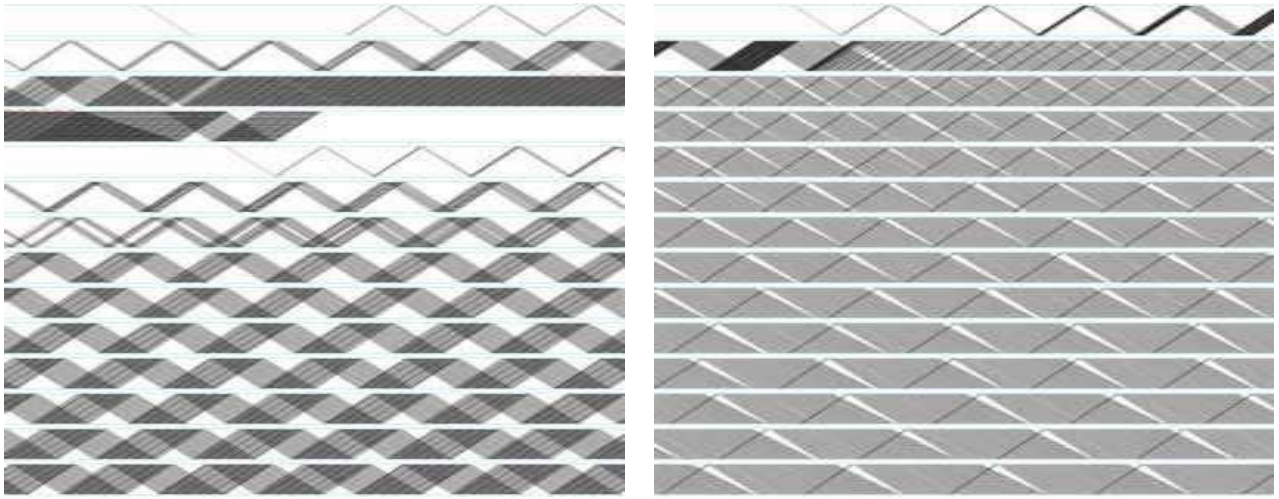
C. Performance Metrics

Completion time of a TCP connection, defined as the amount of time between the transmission of the first data segment and delivery of the last ACK, is the main measure of performance in our studies. Completion time also serves as a basis for other long-term performance metrics. In experiments with multiple TCP connections, the reported completion time is the average of their individual completion times. *Relative completion time* is computed as the ratio of the completion times for a pair of compared connections. To measure fairness of network sharing, we use a *fairness index* [20]:

$$F(t_1, t_2, \ldots, t_n) = \frac{\left(\sum_{i=1}^{n} t_i\right)^2}{n \cdot \sum_{i=1}^{n} t_i^2} \qquad (1)$$

Configuration	Description
1-0	One TCP connection with 50 ms one-way propagation delay and no data traffic on the reverse path.
Asymmetric	Adopted from Balakrishnan, Padmanabhan, and Katz [6]: each of the edge links has a capacity of 10 Mbps and propagation delay of 1 ms in both directions; however, the middle link is asymmetric; it has a capacity of 10 Mbps and propagation delay of 5 ms along the data path of the only TCP connection in the configuration; in the reverse direction, the link has a propagation delay of 50 ms and capacity that varies between 5 Kbps and 30 Kbps.
2-2	Symmetric: four TCP connections transmit data over the bottleneck link in opposite directions so that each router forwards to the bottleneck link a mix of ACKs and data packets; the first connection starts sending its data at time 0; another connection starts its transmission in the same direction 200 ms later; the two connections that send data in the reverse direction start at time 100 ms and 300 ms respectively; the middle bottleneck link has a propagation delay of 20 ms and variable capacity; each of the edge links has a capacity of 10 Mbps and propagation delay of 5 ms; hence, the round-trip propagation time for the connections is 60 ms.
High-Multiplexing	Replaces every connection in the 2-2 configuration with 50 parallel TCP connections, i.e., 100 connections transmit data in one direction of the bottleneck link, and the other 100 connections send data in the opposite direction.
Different-RTT	Variation of the 2-2 configuration: in each direction, one of the two connections has an increased propagation RTT of 120 ms; the propagation RTT for the other connection remains 60 ms.
TCP-UDP	Replaces one of the two TCP connections in each direction of the 2-2 configuration with an on-off UDP flow that changes its mode of operation once per 5 seconds; when the UDP flow is on, it injects packets into the network at a constant rate equal to the half of the bottleneck link capacity.

Fig. 3. Configurations of the Dumbbell topology.



(a) NewReno, 29% (b) NewReno with smartacking, 61%

Fig. 4. Timeline diagrams for the 1-0 configuration with the bottleneck link capacity of 9 Mbps.

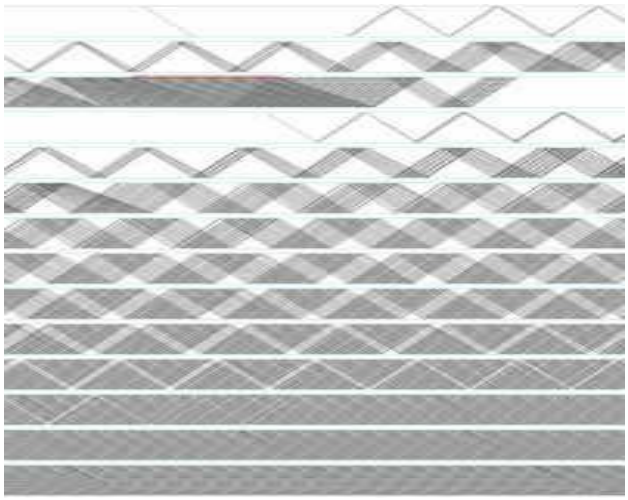
where n is the number of connections, and t_i denotes the completion time of connection i. In addition to the long-term metrics, we monitor dynamics of TCP connections on shorter timescales and record the congestion window and threshold at the sender, congestion window estimate at the receiver, transmission and arrival times of data segments and ACKs.

V. Experimental Results

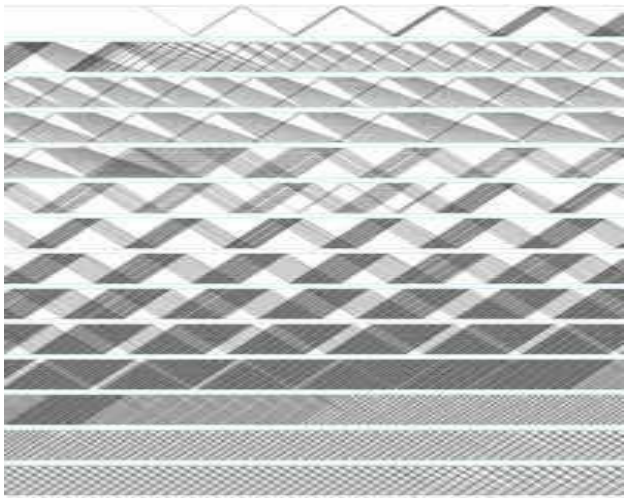
This section presents results from our experimental evaluation of smartacking. First, we explore the impact of smartacking on dynamics of TCP communications in Section V-A. Then, Section V-B focuses on long-term performance in networks with Droptail routers. Section V-C reports results for networks of RED routers. Section V-D studies performance of smartacking in the Asymmetric configuration. Section V-E quantifies the reaction of the examined protocols to sharp changes in the available capacity. Section V-F evaluates smartacking in network configurations with high levels of connection multiplexing on bottleneck links. Finally, Section V-G investigates the impact of smartacking on fairness of network sharing.

A. Impact on TCP Dynamics

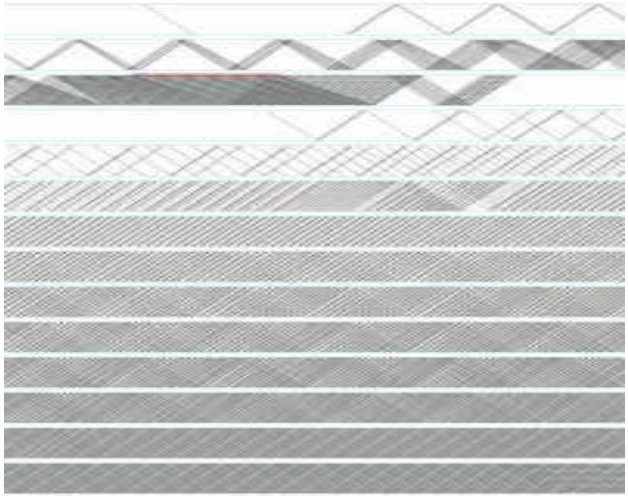
Before evaluating the widely deployed TCP designs and their extensions in terms of long-term performance, it is important to understand how short-term dynamics of the schemes affect the cumulative metrics. We use *timeline diagrams* to reflect TCP behavior over a range of shorter timescales, starting from the under-RTT timescale to intervals that capture a sequence of transitions between the congestion control modes of the connection. In each of our timeline diagrams, a horizontal band represents interactions between the sender and receiver over a fixed period. Whereas the top edge of



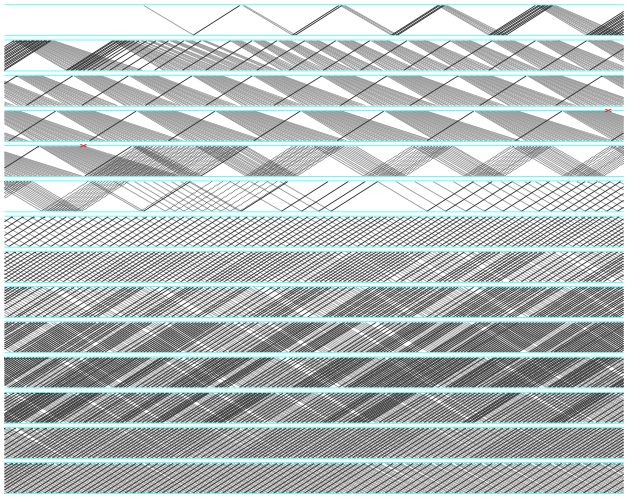
(a) NewReno, 23%



(b) NewReno with smartacking, 27%

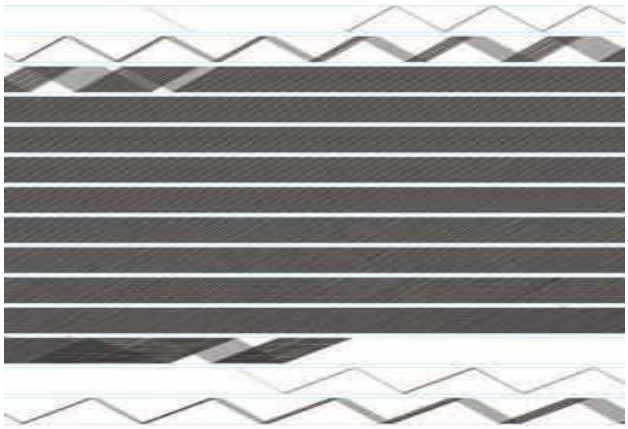


(c) NewReno with pacing, 22%

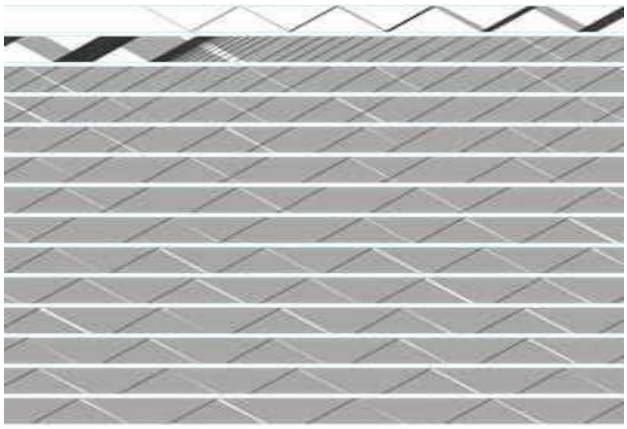


(d) NewReno with smartacking and pacing, 25%

Fig. 5. Timeline diagrams for the 1-0 configuration with the bottleneck link capacity of 5 Mbps.



(a) NewReno, 49%



(b) NewReno with smartacking, 68%

Fig. 6. Timeline diagrams for the 2-2 configuration with the bottleneck link capacity of 20 Mbps.

the band corresponds to the sender, the bottom edge denotes the receiver. Time advances from the left to the right. Every timeline diagram consists of 14 horizontal bands stacked in their temporal order: the left edge of the top band corresponds to time 0, the right edge of this band represents the same time as the left edge of the band immediately below, and so on until the right edge of the bottom band shows the last time reflected in the diagram. While gray lines that descend to the right within a band represent communications of data segments,

black lines that rise to the right denote ACKs. Then, darkness of a band region indicates intensity of communications during the corresponding time interval. We annotate each timeline diagram with the name of the examined protocol and also report what percentage of the file is communicated by the end of the timeline.

To show the impact of smartacking, Figure 4 presents two timeline diagrams for the 1-0 configuration with the bottleneck link capacity of 9 Mbps. The diagram in Figure 4a demonstrates that NewReno stalls after delivering the first data segment because the receiver delays acknowledging the segment for the maximum duration allowed. When the transmission resumes, the sender doubles the congestion window each RTT until a burst of data segments saturates the bottleneck link and causes losses. After a retransmission timeout ends the ensued second stall, NewReno reduces the window to one MSS and then goes through the post-congestion slow start into congestion avoidance. However, the connection fails to recapture the bottleneck capacity and communicates only 29% of the file by the end of the timeline. As Figure 4b shows, NewReno with smartacking achieves a higher throughput of 61%. Smartacking improves the throughput partly by avoiding both stalls. Upon receiving the first data segment or whenever the bottleneck link is underutilized, the smartacking receiver transmits an ACK immediately and does not slow down the growth of the congestion window. As the transmission approaches the bottleneck link capacity, NewReno with smartacking reduces the frequency of ACKs and converges to a pattern where the receiver provides the sender with one ACK per window so that – as the narrow white triangles in Figure 4b indicate – data from a new window starts arriving to the bottleneck link router before the link runs out of packets to forward. In this phase, the connection grows the congestion window slowly and induces no losses.

We repeat the above experiments in the 1-0 configuration where the bottleneck link capacity is 5 Mbps, and loss rates are higher. Figures 5a and 5b confirm that smartacking reduces the reverse-path traffic and improves the connection throughput, albeit the improvement is less dramatic: from 23% to 27%. Figures 5c and 5d show that pacing reduces NewReno throughput slightly due to the smoother transmission and has a similar effect on NewReno with smartacking. However, NewReno with smartacking and pacing still outperforms plain NewReno.

Figure 6 shows that in the 2-2 configuration with the bottleneck link capacity of 20 Mbps, plain NewReno stalls twice as well: (1) due to the delayed ACK of the first data segment, and (2) recovering from losses via a retransmission timeout. On the other hand, NewReno with smartacking acquires the available capacity promptly and then maintains smooth data delivery with a low frequency of ACKs and no retransmission timeouts. 49%-to-68% throughput increase quantifies the improvement in performance.

To understand the reasons for the throughput improvements offered by smartacking, we trace the congestion window at the sender. Figure 7 reports the traced values for eight of the examined TCP extensions in the 1-0 configuration with the bottleneck link capacity of 5 Mbps. In the schemes with no smartacking (NewReno, NewReno with pacing, AckEvery, ABC, and ACC), the window in the congestion-avoidance mode grows at approximately the same rate regardless of the bottleneck link utilization. On the other hand, the smartacking designs (NewReno with smartacking, NewReno with smartacking and pacing, and NewReno with extended smartacking) reduce the window growth when the bottleneck link gets saturated. The smoother increase prolongs the periods when the bottleneck link capacity is fully utilized. Since the area under the congestion window curve is correlated to the connection throughput, Figure 7 illustrates why the adjustment of the ACK frequency allows smartacking to improve the throughput.

The graphs also plot our estimate of the congestion window at the receiver and demonstrate accuracy of the proposed estimation mechanism. Although we trace the congestion window in the 2-2 configuration as well, below we describe respective graphs only briefly due to space constraints (the interested reader can see the graphs in our longer report [8]): despite the higher degree of connection multiplexing, estimation of the congestion window at the receiver remains precise; besides, pacing helps smartacking to smooth oscillations of the congestion window when the bottleneck link is fully utilized.

Figure 7 shows that the window at times of congestion reaches about 60 MSS under the non-smartacking schemes versus nearly 80 MSS with smartacking. The higher value under smartacking is due to delayed ACKs: as the bottleneck link becomes completely utilized, the receiver starts delaying ACKs; the extra delay results in larger RTT and window when the bottleneck link buffer overflows. This side effect of smartacking is positive since TCP does not recover from packet losses nicely if the window is at most few MSS.

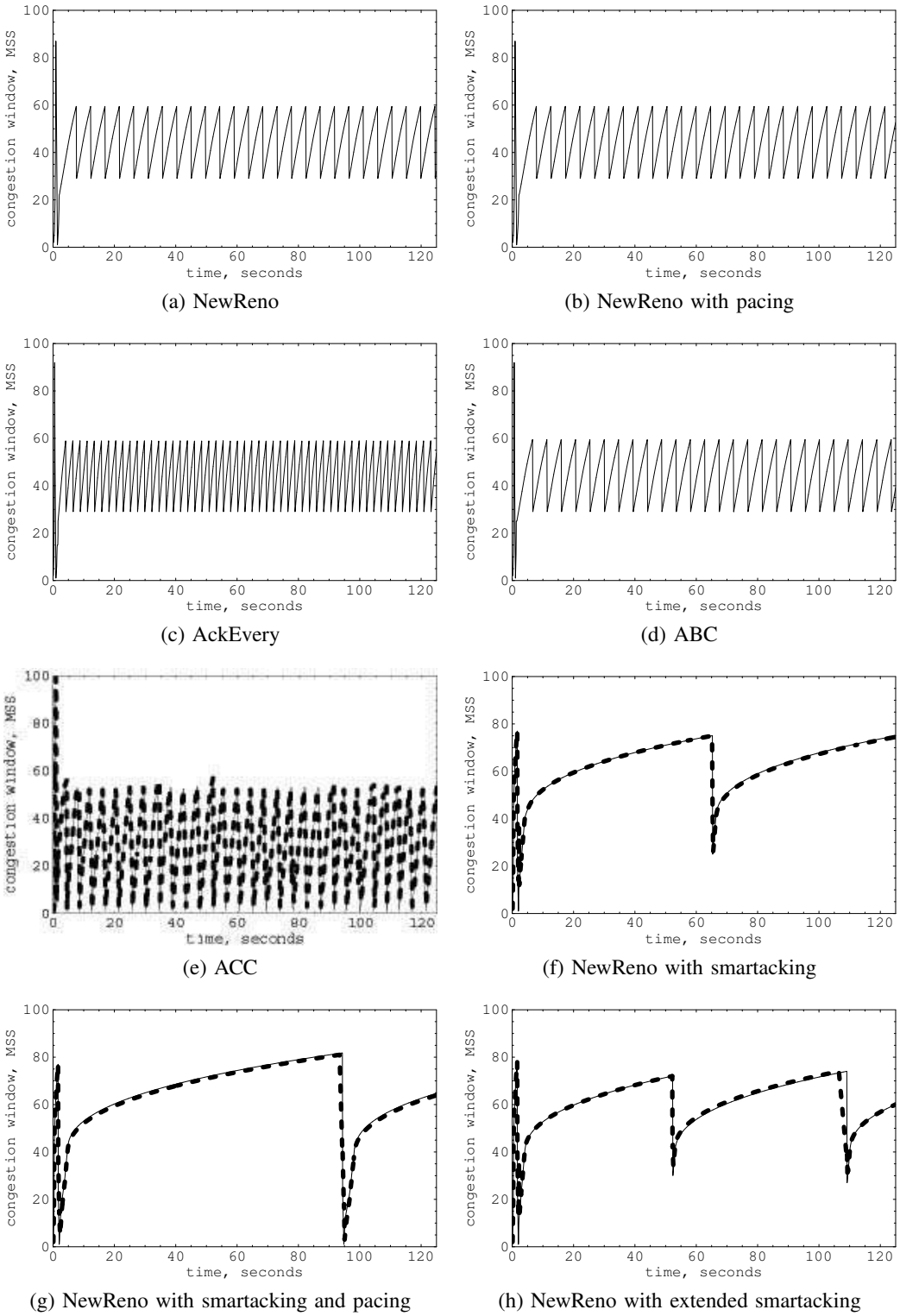
B. Long-Term Performance

For the 1-0 configuration, Figure 8a shows completion times of connections using NewReno, NewReno with pacing, NewReno with smartacking, or NewReno with smartacking and pacing. Figure 8b plots relative completion times computed as ratios of the connection completion times to the completion time of the NewReno connection. The implementations with smartacking perform consistently better than the schemes without smartacking, with up to 20% reduction in the completion time. Both smartacking implementations achieve similar performance without any substantial benefits from pacing.

Figure 9 compares smartacking (represented by NewReno with smartacking and pacing) with byte counting (represented by ABC) in the 1-0 and 2-2 configurations while NewReno, AckEvery, and NewReno with pacing form a background for the comparison. By increasing the rate of the window growth during the slow start, ABC typically provides a smaller completion time than NewReno. However, smartacking consistently yields a much larger reduction. This further improvement is due to the smoother window growth when the bottleneck link capacity becomes saturated.

Figure 10 presents experimental results for the Different-RTT configuration. In general, the smartacking versions provide lower completion times. However, to achieve the improvement for smaller capacities of the bottleneck link shared by connections with different RTTs, the receiver needs the sender's assistance in the form of pacing or α adjustments.

It is well known that TCP performance depends on the size of the bottleneck link buffer. A common rule of thumb prescribes setting the buffer size to one bandwidth-delay product (BDP) [19]. To evaluate the impact of the buffer size on the benefits from smartacking, we conduct experiments in the 2-2 configurations where the bottleneck link has capacities of 7 Mbps and 15 Mbps, and its buffer size varies from 0.3 BDP to 3 BDP. Figure 11 confirms that the buffer size of 1 BDP provides the traditional TCP versions with near optimal



(a) NewReno (b) NewReno with pacing

(c) AckEvery (d) ABC

(e) ACC (f) NewReno with smartacking

(g) NewReno with smartacking and pacing (h) NewReno with extended smartacking

Fig. 7. Congestion windows at the sender (solid lines) and their estimates at the receiver (dotted lines) in the 1-0 configuration with the bottleneck link capacity of 5 Mbps.

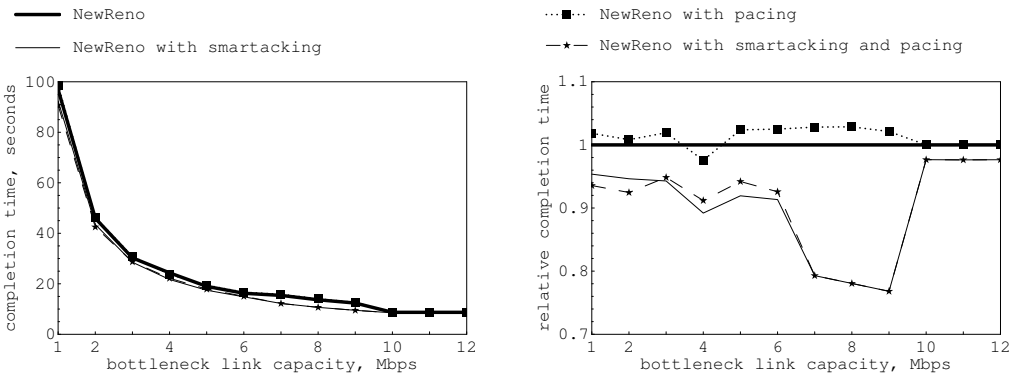
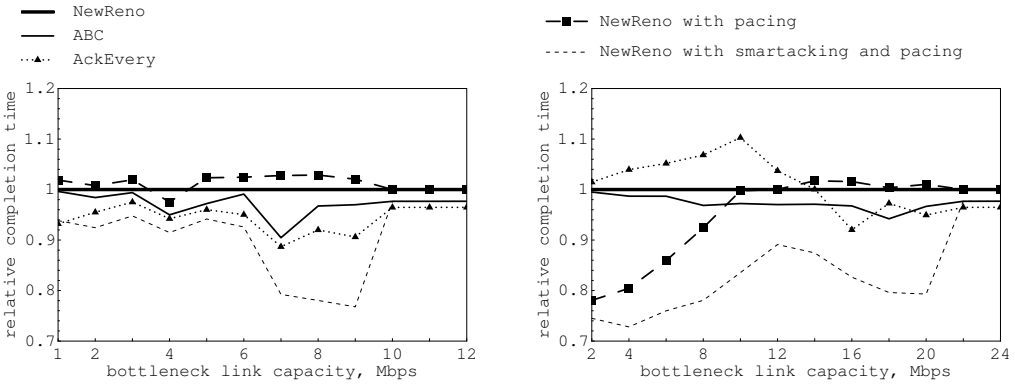


Fig. 8. Completion times in the 1-0 configuration.



(a) 1-0 configuration (b) 2-2 configuration

Fig. 9. Comparison of smartacking with byte counting.

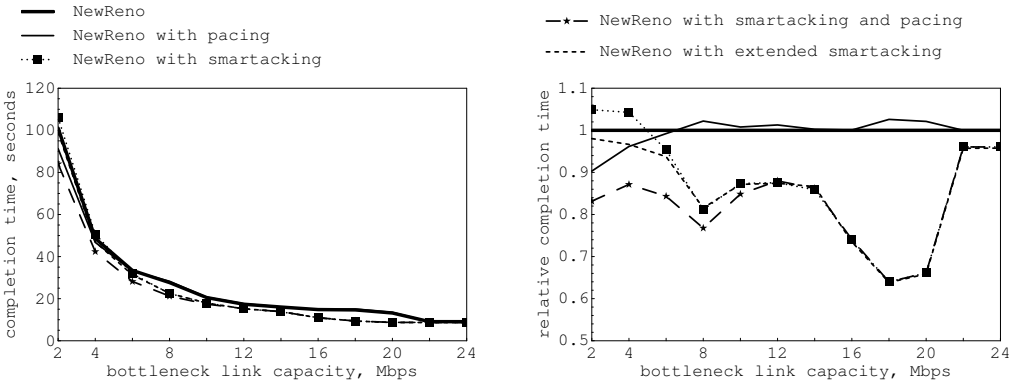
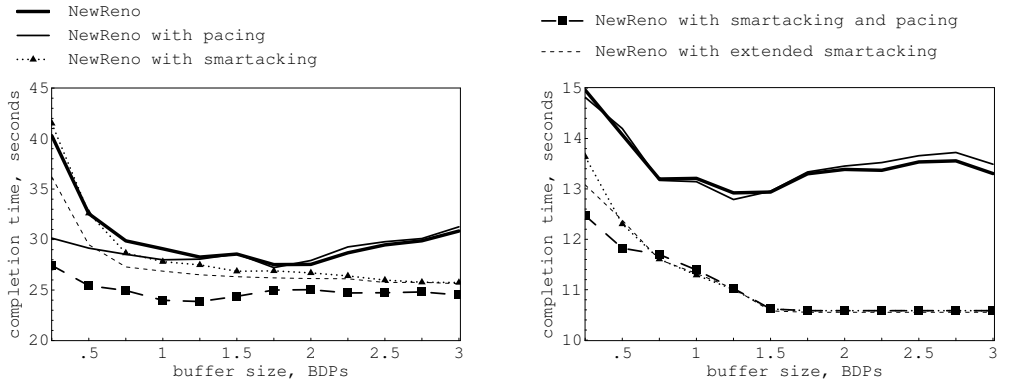
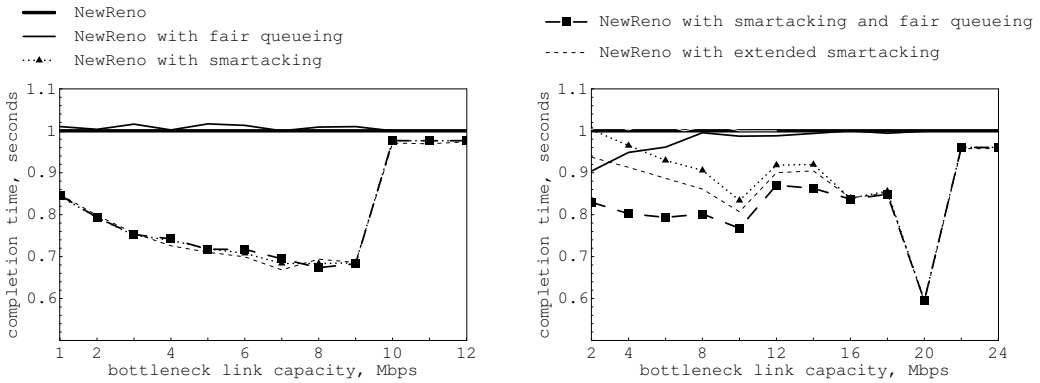


Fig. 10. Completion times in the Different-RTT configuration.



(a) bottleneck link capacity of 7 Mbps (b) bottleneck link capacity of 15 Mbps

Fig. 11. Impact of the buffer size of the bottleneck link on completion times in two 2-2 configurations with different link capacities.



(a) 1-0 configuration (b) Different-RTT configuration

Fig. 12. Completion times in networks with RED routers.

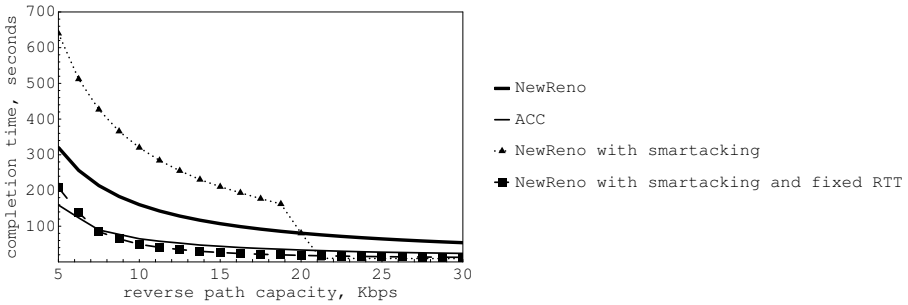


Fig. 13. Completion times in the Asymmetric configuration.

performance. The optimal buffer size for the smartacking implementations appears to be somewhat larger. We attribute the increase in the optimal buffer size to extra delay imposed by smartacking on ACKs at the receiver. The graphs show that smartacking improves TCP performance over a wide range of examined buffer sizes. Furthermore, the relative performance improvement becomes higher as the buffer size of the bottleneck link increases.

C. Networks with RED Routers

Figure 12 reports some of our extensive experimental results for networks with RED routers. In general, smartacking TCP extensions (represented in our graphs by NewReno with smartacking, NewReno with extended smartacking, and NewReno with smartacking and fair queueing) consistently outperform schemes with no smartacking (represented by NewReno and NewReno with fair queueing). In the 1-0 configuration, all the implementations of smartacking behave similarly. However, in the Different-RTT configuration where two connections compete for the bottleneck link capacity in each direction, the smartacking extensions with advanced features – such as dynamic α or fair queueing – provide larger reductions in the completion time than the reductions offered by our receiver-only implementation of smartacking.

D. Asymmetric Networks

We compare performance of smartacking implementations and ACC in the Asymmetric configuration used originally in the studies proposing ACC. Our results confirm that ACC provides shorter completion times than NewReno over the whole range of reverse-path capacities between 5 Kbps and 30 Kbps. We also observe that NewReno with smartacking and pacing reduces the completion times even further. However, Figure 13 demonstrates that smartacking performs less consistently without support from pacing. For reverse-path capacities of at least 21 Kbps, NewReno with smartacking yields lower completion times than ACC. On the other hand, even plain NewReno outperforms NewReno with smartacking when the reverse-path capacity is less than 20 Kbps. Our analysis links this degradation in performance to the RTT estimate at the receiver. When the reverse path has a low capacity, ACKs are frequent enough to saturate the capacity and increase the reverse-path queueing delay. Consequently, the receiver increases its RTT estimate and transmits one ACK per data segment arrival even when the forward-path capacity is fully utilized.

We validate the above analysis by repeating the experiment for NewReno with smartacking when the receiver does not change the RTT estimate after measuring it in the beginning of the connection. Figure 13 denotes the new scheme as *NewReno with smartacking and fixed RTT*. The extension behaves similarly to NewReno with smartacking and pacing. Hence, not only sender-dependent implementations (such as the one with pacing) but also receiver-only smartacking implementations have a potential to outperform ACC consistently in asymmetric networks with an extremely low reverse-path capacity. Realizing this potential in an integrated implementation of smartacking is a topic for future research.

E. Reaction to Capacity Changes

To evaluate smartacking when the available capacity changes, we compare traditional TCP implementations (NewReno, AckEvery, and NewReno with pacing) to ACC and NewReno with smartacking and pacing in the TCP-UDP configuration. Figure 14 shows that addition of smartacking

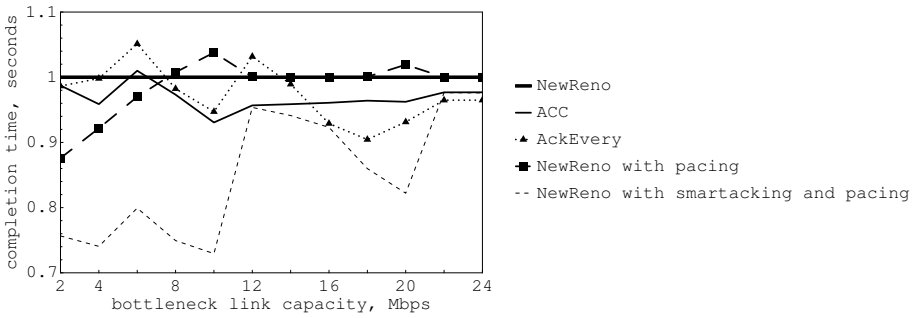
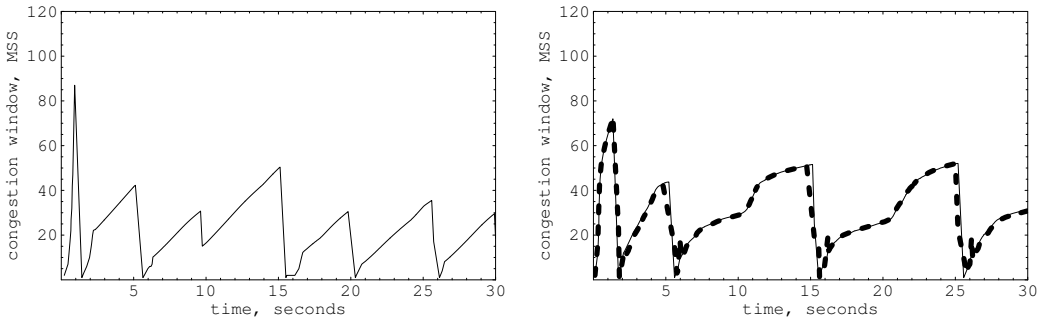


Fig. 14. Completion times in the TCP-UDP configuration.



(a) NewReno with pacing (b) NewReno with smartacking and pacing

Fig. 15. Congestion windows at the sender (solid lines) and their estimates at the receiver (dotted lines) in the TCP-UDP configuration with the bottleneck link capacity of 5 Mbps.

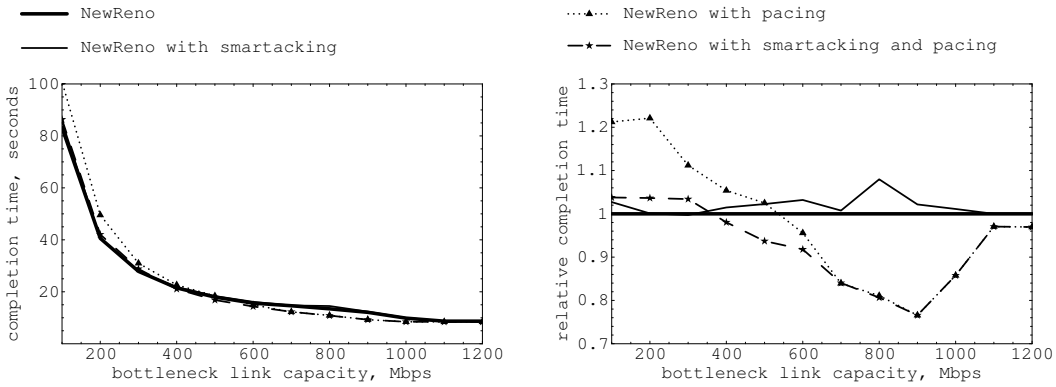


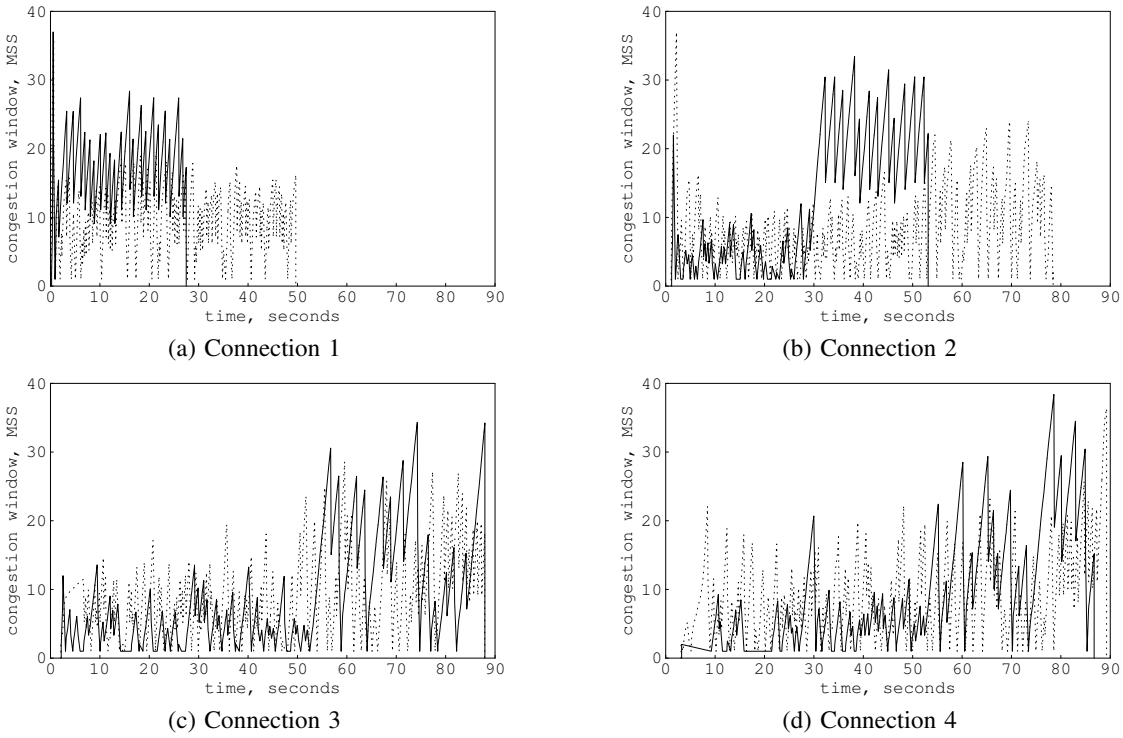
Fig. 16. Completion times in the High-Multiplexing configuration.

reduces the completion time and consistently provides a lager improvement in performance than ACC.

Figure 15 illustrates reasons why smartacking is beneficial in networks with variable capacities. The graphs trace the congestion windows for NewReno with pacing and NewReno with smartacking and pacing in the TCP-UDP configuration where the bottleneck link capacity is 5 Mbps. When the UDP flow starts transmitting, the influx of its packets congests the network and causes losses. Both NewReno implementations notice the congestion, curb their transmission, and then probe for the new available capacity. Initially, the smartacking receiver resumes sending one ACK per data segment arrival. As the utilization of the bottleneck link returns to its capacity, the smartacking receiver decreases the ACK frequency, and this yields the desired reduction in the congestion window growth. However, when the UDP flow turns quiet and thereby releases the half of the bottleneck link capacity, the smartacking receiver reacts to the change by returning to the highest frequency of ACKs. Once again, as the bottleneck link gets saturated, NewReno with smartacking and pacing reduces the ACK frequency and congestion window growth. In contrast, NewReno with pacing notices neither the newly released capacity nor the approaching saturation of the bottleneck link and continues to grow the window in the same linear fashion. Our experiments also show that smartacking recognizes the availability of the new capacity much faster than ACC which takes up to sixteen RTTs to increase the ACK frequency to one ACK per data segment arrival.

F. High Levels of Connection Multiplexing

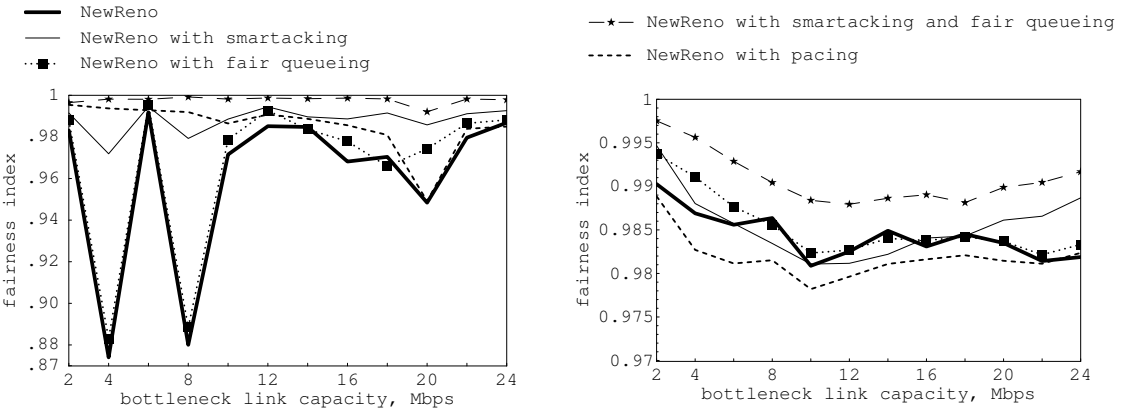
So far, we experimented in environments where the degree of connection multiplexing on bottleneck links was low. What happens for higher levels of connection multiplexing? Figure 16 reports completion times for the High-Multiplexing configuration with 200 concurrent connections. The results are representative of all our experiments with high levels of connection multiplexing: smartacking provides no tangible improvements in efficiency of congestion control when the number of connections is large. This conclusion is not sur-



(a) Connection 1 (b) Connection 2

(c) Connection 3 (d) Connection 4

Fig. 17. Congestion windows in the Parking-Lot topology with NewReno (solid lines) and NewReno with smartacking (dotted).



(a) Droptail routers (b) RED routers

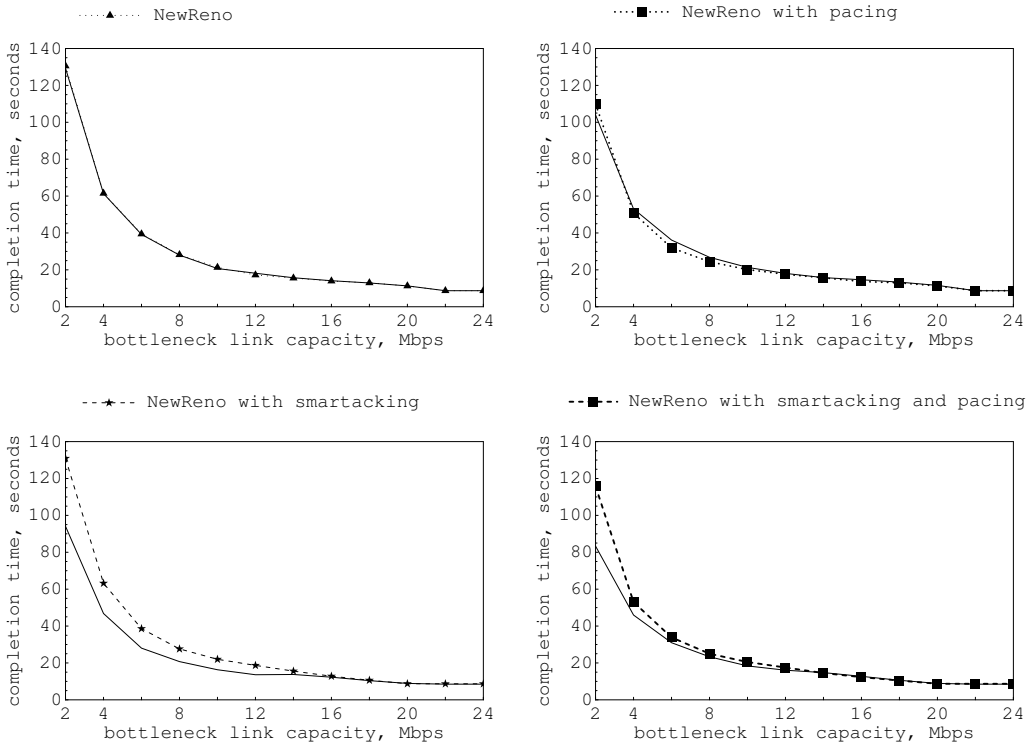
Fig. 18. Intra-protocol fairness in the Parking-Lot topology.

prising. With many connections sharing the bottleneck link, inefficiencies of an individual connection are counterbalanced by other connections. Hence, smartacking has a smaller headroom for improving the efficiency. Furthermore, our ISAT-based mechanism for detecting available capacity becomes less precise. On the other hand, our experiments with smartacking in high-multiplexing settings reveal no negative impact substantial enough to offset the benefits from smartacking in low-multiplexing environments. Furthermore, our results in Section V-G indicate that even configurations with many concurrent connections can benefit from smartacking because smartacking improves fairness of network sharing.

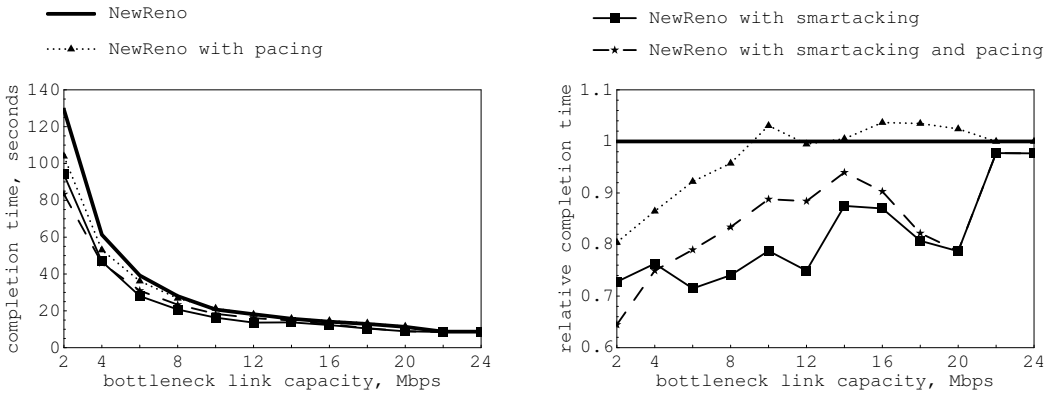
G. Fairness of Network Sharing

We study the impact of smartacking on fairness by considering first the issue of intra-protocol fairness and then shifting our attention to TCP-friendliness.

Figure 17 traces the congestion windows of the four connections in the Parking-Lot topology with Droptail routers and shared link capacities of 4 Mbps. Since the differences between the starting times of the connections are minor in comparison to the completion times, the last of the three shared links serves as a bottleneck for all the connections most of the time. Hence, under a fair allocation of the bottleneck link capacity, the connections should have similar completion times. However, traditional TCP extensions discriminate against connections with long RTTs. Figure 17 confirms this phenomenon for NewReno. Connection 1, which has the smallest RTT, grabs most of the bottleneck capacity and finishes much earlier than the three others. Having the second smallest RTT, connection 2 inherits the domination over the bottleneck link and finishes much before connections 3 and 4 which share the longest RTT. Then, the remaining connections 3 and 4 take turns in grabbing a larger portion of the bottleneck capacity. Figure 17 also demonstrates that smartacking helps NewReno to improve the fairness of the bottleneck link sharing. The congestion windows of all the



(a) completion times for a NewReno connection (solid line) and parallel connection (as identified above each graph)



(b) completion times for the NewReno connection with different types of cross traffic

Fig. 19. TCP-friendliness of smartacking in the 2-2 configuration.

connections become more stable and similar to each other. Consequently, the completion times of the connections become less diverse as well.

Figure 18a reports the fairness index for five TCP extensions in the Parking-Lot topology with Droptail routers. Addition of smartacking to New Reno or NewReno with pacing improves the intra-protocol fairness consistently. As expected, NewReno with smartacking and fair queueing yields the highest fairness index among the three examined implementations of smartacking. Figure 18b shows the fairness index for the Parking-Lot topology with RED routers. By discarding packets probabilistically before the link buffer gets full, the RED router of the bottleneck link allows New Reno to raise its low fairness index in the scenarios where Droptail buffer management starves a connection by sending it into a series of retransmission timeouts. In general, switching to RED routers results in a smoother fairness index for each of the examined TCP extensions.

We evaluate TCP-friendliness of smartacking in the 2-2 configuration where three connections employ NewReno while the fourth connection uses either NewReno, or NewReno with pacing, or NewReno with smartacking, or NewReno with smartacking and pacing. Figure 19a reports completion times for the fourth connection as well as for the NewReno connection that delivers data in the same direction. Figure 19b shows that adoption of smartacking by the fourth connection not only does not harm the parallel NewReno connection but also helps the NewReno connection to reduce its completion time.

VI. Conclusion

We presented *smartacking*, a technique that improves performance of TCP via adaptive generation of ACKs at the receiver: when the bottleneck link is underutilized, the receiver transmits an ACK for each delivered data segment and thereby allows the connection to acquire the available capacity quickly;

when the bottleneck link is at its capacity, the receiver sends ACKs with a lower frequency reducing the control traffic overhead and slowing down the congestion window growth to utilize the network capacity more effectively. Smartacking estimates availability of the network capacity by measuring the inter-segment arrival time (ISAT) at the receiver. Our experiments confirmed that this estimation mechanism operates precisely. In particular, when UDP or TCP cross traffic ceases, ISAT measurements promptly reflect the freed network capacity and boost the rate of ACKs; then, the triggered faster growth of the congestion window enables the connection to capture the released capacity aggressively. Also, since smartacking allows a new TCP connection to raise its congestion window quickly, the technique is particularly beneficial in networks with many short-lived connections.

To promote quick deployment of smartacking, our primary implementation of the technique modified only the receiver. This implementation estimates the sender's congestion window using a novel algorithm that has independent value, e.g., for ACC and other protocols where the receiver must know the congestion window. We also considered different implementations of smartacking where the sender or network provides the receiver with explicit assistance such as pacing or fair queueing.

Our experiments in a wide variety of settings showed that all the considered implementations of smartacking help TCP to be more efficient in networks with low levels of connection multiplexing. In networks with high levels of connection multiplexing, efficiency gains from smartacking are negligible because of two reasons. First, the ISAT-based mechanism for detecting availability of the network capacity is less precise when the number of connections is large. Second, with a high level of connection multiplexing, TCP utilizes the bottleneck link quite efficiently even without smartacking. On the other hand, our experiments indicated that networks with many connections on bottleneck links can also benefit from smartacking because smartacking improves fairness of network sharing.

Based on our findings, we believe that smartacking represents a promising approach for improving TCP. However, additional extensive studies over real networks are needed before the technique becomes ready for wide deployment. Whereas this paper presented implementations of smartacking for TCP NewReno, another direction for future work is to implement smartacking for other TCP versions such as SACK. Also, it seems enticing to combine smartacking with ACK filtering [6] in order to derive a more effective protocol for asymmetric networks.

ACKNOWLEDGMENT

The authors would like to thank Sally Floyd, Guru Parulkar, Jon Turner, George Varghese, Marcel Waldvogel, and Ellen Zegura for their extremely valuable feedback.

REFERENCES

[1] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," in *Proc. IEEE INFOCOM 2000*, Tel-Aviv, Israel, March 2000.

[2] M. Allman, "On the Generation and Use of TCP Acknowledgments", *ACM Computer Communication Review*, vol. 28, no. 5, October 1998, pp. 4–21.

[3] M. Allman, "TCP Byte Counting Refinements", *ACM Computer Communication Review*, vol. 29, no. 3, July 1999, pp. 14–22.

[4] M. Allman, C. Hayes, and S. Ostermann, "An Evaluation of TCP with Larger Initial Windows", *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 3, July 1998, pp. 41–52.

[5] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.

[6] H. Balakrishnan, V. Padmanabhan, and R. Katz, "The Effects of Asymmetry on TCP Performance", in *Proc. ACM/IEEE MobiCom 1997*, Budapest, Hungary, September 1997.

[7] J. Bennett and H. Zhang, "Hierarchical Packet Fair Queueing Algorithms", *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, October 1997, pp. 675–689.

[8] K. Blandford, S. Goldman, S. Gorinsky, Y. Zhou, and D. Dooly, "Smartacking: Improving TCP Performance from the Receiving End", *www.arl.wustl.edu/~gorinsky/pdf/WUCSE-TR-2005-4.pdf*, Department of Computer Science and Engineering, Washington University in St. Louis, Tech. Rep. WUCSE-2005-4, January 2005.

[9] R. Braden, "Requirements for Internet Hosts – Communication Layers", RFC 1122, October 1989.

[10] R. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, L. Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, April 1998.

[11] L. Brakmo, S. O'Malley, and L. Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance", in *Proc. ACM SIGCOMM 1994*, London, UK, August-September 1994.

[12] D. Clark, "Window and Acknowledgement Strategy in TCP", RFC 813, July 1982.

[13] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm", in *Proc. ACM SIGCOMM 1989*, Austin, USA, September 1989.

[14] D. Ely, N. Spring, D. Wetherall, S. Savage, and T. Anderson, "Robust Congestion Signaling", in *Proc. IEEE ICNP 2001*, Riverside, USA, November 2001.

[15] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 2582, April 1999.

[16] S. Floyd and V. Jacobson, "On Traffic Phase Effects in Packet-Switched Gateways", *Internetworking: Research and Experience*, vol. 3, no. 3, September 1992, pp. 115–156.

[17] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance", *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, August 1993, pp. 397–413.

[18] V. Jacobson, "Congestion Avoidance and Control", in *Proc. ACM SIGCOMM 1988*, Stanford, USA, August 1988.

[19] V. Jacobson, "Modified TCP Congestion Control Algorithm", End2end-interest mailing list, April 1990.

[20] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley-Interscience, April 1991.

[21] S. Keshav, "Packet-Pair Flow Control," *www.cs.uwaterloo.ca/~keshav/*, February 1995.

[22] J. Kulik, R. Coulter, D. Rockwell, and C. Partridge, "Paced TCP for High Delay-Bandwidth Networks", in *Proc. IEEE Globecom 1999*, Rio de Janeiro, Brazil, December 1999.

[23] J. Kulik, R. Coulter, D. Rockwell, and C. Partridge, "A Simulation Study of Paced TCP", BBN, Tech. Rep. CR-2000-209416, *gltrs.grc.nasa.gov/reports/2000/CR-2000-209416.pdf*, January 2000.

[24] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, October 1996.

[25] S. McCanne and S. Floyd, *ns Network Simulator. http://www.isi.edu/nsnam/ns/.*

[26] A. Medina, M. Allman, and S. Floyd, "Measuring the Evolution of Transport Protocols in the Internet", *ACM Computer Communication Review*, vol. 35, no. 2, April 2005, pp. 37–52.

[27] R. Oliveira and T. Braun, "A Dynamic Adaptive Acknowledgment Strategy for TCP over Multihop Wireless Networks", in *Proc. IEEE INFOCOM 2005*, Miami, USA, March 2005.

[28] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack, "Upgrading Transport Protocols using Untrusted Mobile Code", in *Proc. ACM SOSP 2003*, New York, USA, October 2003.

[29] K. Poduri and K. Nichols, "Simulation Studies of Increased Initial TCP Window Size", RFC 2415, September 1998.

[30] J. Postel, "User Datagram Protocol", RFC 768, October 1980.

[31] K. Ramakrishnan and S. Floyd, "A Proposal to Add Explicit Congestion Notification (ECN) to IP", RFC 2481, January 1999.

[32] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP Congestion Control with a Misbehaving Receiver", *ACM Computer Communication Review*, vol. 29, no. 5, October 1999, pp. 71–78.

[33] T. Shepard and C. Partridge, "When TCP Starts Up With Four Packets Into Only Three Buffers", RFC 2416, September 1998.

[34] L. Zhang, S. Shenker, and D. Clark, "Observations and Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic", in *Proc. ACM SIGCOMM 1991*, Zurich, Switzerland, September 1991.

Sally Goldman received a B.Sc. from Brown University and a M.S. and Ph.D. from Massachusetts Institute of Technology under the supervision of Ronald Rivest. Dr. Goldman is currently on the faculty of Washington University in St. Louis where she is a Full Professor and also the Associate Chair of the Department of Computer Science and Engineering. Her primary research interests are machine learning, content-based image retrieval, and computational learning theory. She was a recipient of the NSF National Young Investigator award. Dr. Goldman's work has appeared in many top conferences including FOCS, STOC, ICML, CVPR, ACM Multimedia, NIPS, COLT, and in journals such as the Journal of the ACM, SIAM Journal on Computing, Information and Computation, Journal of Computer and System Sciences, Journal of Machine Learning Research, and Machine Learning Journal. She is currently on the editorial board for the Journal of Machine Learning Research and the Journal of Computer and Systems Sciences.

Sergey Gorinsky is a native of Skhodnya, Russia. He received the degree of Engineer at Moscow Institute of Electronic Technology, Zelenograd, Russia and M.S. and Ph.D. degrees from the University of Texas at Austin, USA. Dr. Gorinsky is currently with Washington University in St. Louis, USA where he works as an Assistant Professor at the Applied Research Laboratory in the Department of Computer Science and Engineering. His primary research interests are in computer networking and distributed systems. Dr. Gorinsky's work appeared at top conferences and journals such as ACM SIGCOMM, IEEE INFOCOM, and IEEE/ACM Transactions on Networking. He has been serving on Technical Program Committees of IEEE INFOCOM and other networking conferences.

Yan Zhou is an Assistant Professor in the School of Computer and Information Sciences at the University of South Alabama. She received the Doctor of Science degree from Washington University in St. Louis in 2001. Her recent and forthcoming publications are in the areas of machine learning and data mining.

Daniel R. Dooly is an Assistant Professor of Computer Science at Southern Illinois University Edwardsville. He holds a B.A. in Mathematics from Greenville College, an M.S. in Mathematics from Southern Illinois University Edwardsville, and a D.Sc. in Computer Science from Washington University in St. Louis. His current research interests are in algorithms and machine learning. Dr. Dooly is a member of ACM and IEEE Computer Society. His papers have appeared in the Journal of Computer and System Sciences, Journal of Machine Learning Research, Journal of the ACM, Naval Research Logistics, and Telecommunication Systems.