TCP-PR: TCP for Persistent Packet Reordering[*]

Stephan Bohacek[†] João P. Hespanha[‡] Junsoo Lee[§] Chansook Lim[§] Katia Obraczka[††]

bohacek@eecis.udel.edu hespanha@ece.ucsb.edu junsoole@usc.edu chansool@usc.edu katia@cse.ucsc.edu

[†]*Dept. Electrical & Computer Engineering, Univ. of Delaware, Newark, DE 19716*
[‡]*Dept. Electrical & Computer Engineering, Univ. of California Santa Barbara, CA 93106-9560*
[§] *Department of Computer Science, Univ. of Southern California Los Angeles, CA 90089*
[††]*Computer Engineering Department, University of California Santa Cruz, CA 95064*

Abstract

Most standard implementations of TCP perform poorly when packets are reordered. In this paper, we propose a new version of TCP that maintains high throughput when reordering occurs and yet, when packet reordering does not occur, is friendly to other versions of TCP. The proposed TCP variant, or TCP-PR, does not rely on duplicate acknowledgments to detect a packet loss. Instead, timers are maintained to keep track of how long ago a packet was transmitted. In case the corresponding acknowledgment has not yet arrived and the elapsed time since the packet was sent is larger than a given threshold, the packet is assumed lost. Because TCP-PR does not rely on duplicate acknowledgments, packet reordering (including out-of-order acknowledgments) has no effect on TCP-PR's performance.

Through extensive simulations, we show that TCP-PR performs consistently better than existing mechanisms that try to make TCP more robust to packet reordering. When the case that packets are not reordered, we verify that TCP-PR maintains the same throughput as typical implementations of TCP (specifically, TCP-SACK) and shares network resources fairly.

1 Introduction

The design of TCP's error and congestion control mechanisms was based on the premise that packet loss is an indication of network congestion. Therefore, upon detecting loss, the TCP sender backs off its transmission rate by decreasing its *congestion window*. TCP uses two strategies for detecting packet loss. The first one is based on the sender's retransmission time-out (RTO) expiring and is sometimes referred to as *coarse time-out*. When the sender times out, congestion control responds by causing the sender to enter *slow-start*, drastically decreasing its congestion window to one segment. The other loss detection mechanism originates at the receiver and uses TCP's sequence number. Essentially, the receiver observes the sequence numbers of packets it receives; a "hole" in the sequence is considered indicative of a packet loss. Because TCP mainly uses cumulative acknowledgments[1], the receiver generates a "duplicate acknowledgment" (or DUPACK) for every "out-of-order" segment it receives. Note that until the lost packet is received, all other packets with higher sequence number are considered "out-of-order" and will cause DUPACKs to be generated. Modern TCP implementations adopt the *fast retransmit* algorithm which infers that a packet has been lost after the sender receives a few DUPACKs. The sender then retransmits the lost packet without waiting for a time-out and reduces its congestion window in half. The basic idea behind fast retransmit is to improve TCP's throughput by avoiding the sender to time-out (which results in slow-start and consequently the shutting down of the congestion window to one).

Fast retransmit can substantially improve TCP's performance in the presence of sporadic reordering but it still operates under the assumption that out-of-order packets indicate packet loss and therefore congestion. Consequently, its performance degrades considerably in

[*]The research presented in this paper was supported by DARPA and NSF. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

[1]More recently, TCP SACK has been proposed and enables the TCP receiver to selectively acknowledge out-of-sequence segments.

the presence of "persistent reordering." Indeed, it is well known that TCP performs poorly under significant packet reordering that is not necessarily caused by packet losses [3]. This is the case not only for re-ordering of data- but also of acknowledgment packets.

Packet reordering is generally attributed to transient conditions, pathological behavior, and erroneous implementations. For example, oscillations or "route flaps" among routes with different round-trip times (RTTs) are a common cause of out-of-order packets observed in the Internet today [17]. However, networks with radically different characteristics (when compared to the Internet, for example) can exhibit packet reordering as a result of their normal operation. This is the case of wireless networks, in particular multi-hop mobile ad-hoc networks (MANETs). In MANETs, which are also known as "networks without a network," there is no fixed infrastructure and every node can be traffic source and sink, as well as traffic forwarder. The potential for unconstrained mobility poses many challenges to routing protocols including frequent topology changes. Thus MANET routing protocols need to recompute routes often which may lead to (persistent) packet reordering. In fact, improving the performance of TCP in such environments (by trying to differentiate out-of-order packets from congestion losses) has been the subject of several recent research efforts [20, 8, 13].

A number of mechanisms that have been recently proposed to enhance the original Internet architecture also result in (persistent) packet reordering. Multipath routing protocols are examples of these. The main idea behind them is to use the Internet's inherent path redundancy and route packets between one particular source and destination over multiple paths. The benefits of multi-path routing include: increased end-to-end throughput, better load balancing across network elements (this is especially important in resource-constrained environments, like MANETs, where power is scarce), and improved immunity to attacks (spreading traffic over a number of paths makes attacks such as packet interception, eavesdropping, and traffic analysis much harder to carry out). However, multiple routes are likely to exhibit different RTTs, causing considerable packet reordering. TCP will therefore not perform well if run atop multi-path routing protocols.

Mechanisms that provide different quality-of-service (QoS) by differentiating traffic are also likely to introduce packet reordering as part of their normal operation. An example of such mechanisms is DiffServ [2], which has been proposed to provide different QoS on the Internet. Typically, these protocols work by marking packets so that when they get to a QoS-capable router, they may be placed in different queues and get forwarded over different routes. This will likely result in packet reordering at the ultimate destination.

TCP's poor performance under persistent packet reordering has been a major deterrent to the deployment of the mechanisms mentioned above on the Internet or other networks in which TCP is prevalent. A number of methods for improving TCP's performance in packet-reordering prone environments have been proposed. Most of them try to recover from occasional reordering and rely on packet ordering itself to distinguish drops from reordering. However, under persistent reordering conditions, packet ordering conveys very little information on what is actually happening inside the network.

In this paper, we propose to solve TCP's poor performance under persistent packet reordering by relying solely on timers. Besides its robustness to the size of the reordering event, TCP-PR neither requires changes to the TCP receiver nor uses any special TCP header option. Through extensive simulations, we evaluate the performance of TCP-PR comparing it to a number of existing schemes that address TCP's poor performance under packet reordering. We also test TCP-PR's compatibility and fairness to standard TCP variants, specifically TCP-SACK. In the absence of packet re-ordering, TCP-PR is shown to have similar performance and competes fairly with TCP-SACK. Moreover, in the presence of persistent packet re-ordering, it behaves significantly better than the other algorithms tested.

2 Related Work

As previously mentioned, several mechanisms that address TCP's lack of robustness to packet reordering have been recently proposed. This section summarizes them and puts our work on TCP-PR in perspective.

Upon detecting spurious retransmissions, the Eifel algorithm [15] restores TCP's congestion control state to its value prior to when the retransmission happened. The more spurious retransmissions of the same packet are detected, the more conservative the sender gets. For spurious retransmission detection, Eifel uses TCP's timestamp option and has the sender time-stamp every packet sent. The receiver echoes back the time-stamp in the corresponding acknowledgment (ACK) packets so that the sender can differentiate among ACKs generated in response to the original transmission as well as retransmissions of the same packet[2].

[2]As an alternative to time-stamping every packet, Eifel can also use a single bit to mark the segment generated by the original transmission.

DSACK [11] proposes another receiver-based mechanism for detecting spurious retransmission. Information from the receiver to the sender is carried as an option (the DSACK option) in the TCP header. The original DSACK proposal does not specify how the TCP sender should respond to DSACK notifications. In [3], a number of responses to DSACK notifications were proposed. The simplest one relies on restoring the sender's congestion window to its value prior to the spurious retransmission detected through DSACK[3]. Besides recovering the congestion state prior to the spurious retransmission, the other proposed strategies also adjust the DUPACK threshold (`dupthresh`). The different `dupthresh` adjustment mechanisms proposed include: (1) increment `dupthresh` by a constant; (2) set the new value of `dupthresh` to the average of current `dupthresh` and the number of DUPACKs that caused the spurious retransmission; and (3) set `dupthresh` to an exponentially weighted moving average of the number of DUPACKs received at the sender. Recently, another scheme that relies on adjusting the `dupthresh` has been proposed [21].

Time-delayed fast-recovery (TD-FR), which was first proposed in [18] and analyzed in [3], handles packet reordering. This method stands out from the others in that it utilizes timers as well as DUPACKs. It sets a timer when the first DUPACK is observed. If DUPACKs persists longer than a threshold, then fast retransmit is entered and the congestion window is reduced. The timer threshold is $\max(RTT/2, DT)$, where DT is the difference between the arrival of the first and third DUPACK.

More recently, another scheme for improving the performance of TCP has been proposed. TCP-DOOR [20], which specifically targets MANET environments, detects out-of-order packets by using additional sequence numbers (carried as TCP header options). To detect out-of-order data packets, the TCP sender uses a 2-byte TCP header option called *TCP packet sequence number* to count every data packet including retransmissions. For out-of-order DUPACK detection, the TCP receiver uses a 1-byte header option to record the sequence in which DUPACKs are generated. The TCP sender, upon detecting out-of-order packets (itself or informed by the receiver in the case of out-of-order data packets[4]), responds by either: (1) temporarily disabling congestion control (i.e., keeping congestion control state, such as the retransmission timer RTO and congestion window `cwnd`, constant) for a time interval T_1, or (2) if in congestion avoidance mode, recovering state prior to entering congestion avoidance.

[3]Instead of instantaneously increasing the congestion window to the value prior to the retransmission event, the sender slow-starts up to that value in order to avoid injection of sudden bursts into the network.

[4]As suggested in [20], one way the TCP receiver can notify the sender is by setting a *000* bit in the TCP ACK packet

To some extent, the approaches described above still utilize packet ordering to detect drops. Indeed, when reordering is not persistent, packet ordering is still somewhat indicative of drops and therefore congestion. However, if packets are persistently reordered, packet ordering convey little information regarding congestion and thus are not good heuristics for congestion control. Consequently, while these approaches can recover from occasional out-of-order packets, their performance degrades under persistent packet reordering, as shown in Section 5.

We propose to neglect DUPACKs altogether and rely solely on timers to detect drops: if the ACK for a packet has not arrived and the elapsed time since the packet is sent exceeds a threshold, then the packet is assumed to be lost. In the next section we describe the TCP-PR algorithm in detail.

There are two main design challenges in developing an adaptive timer threshold. First, the threshold must be chosen such that it is only surpassed when a packet has actually been lost. For lack of space we do not discuss this issue here and refer the reader to [5] for details. The second challenge, covered in Section 4, is to maintain fairness with current implementations of TCP. In Section 5, through extensive simulations, we show that TCP-PR performs better than existing packet reordering recovery methods under persistent reordering conditions.

3 TCP-PR

As mentioned above, the basic idea behind TCP-PR is to detect packet losses through the use of timers instead of duplicate acknowledgments. This is prompted by the observation that, under persistent packet reordering, duplicate acknowledgments are a poor indication of packet losses. Because TCP-PR relies solely on timers to detect packet loss, it is also robust to acknowledgment losses. This is because the algorithm does not distinguish between data- (on the forward path) or acknowledgment-losses (on the reverse path).

The proposed algorithms only require changes in the TCP sender and is therefore backward-compatible with any TCP receiver. TCP-PR's sender algorithm is still based on the concept of a congestion window, but the update of the congestion window follows slightly different rules than standard TCP. However, significant care was placed in making the algorithm fair with respect to other versions of TCP to make sure they can coexist.

3.1 The Basic Algorithm

Packets being processed by the sender are kept in one of two lists: The `to-be-sent` list contains all the packets whose transmission is pending, waiting for an "opening" in the congestion window. The `to-be-ack` list contains those packets that were already sent but have not yet been acknowledged. Typically, when an application produces a packet it is first placed in the `to-be-sent` list; when the congestion window allows it, the packet is sent to the receiver and moved to the `to-be-ack` list; finally when an ACK for that packet arrives from the receiver, it is removed from the `to-be-ack` list (under cumulative ACKs, many packets will be simultaneously removed from `to-be-ack`). Alternatively, when it is detected that a packet was dropped, it is moved from the `to-be-ack` list back into the `to-be-sent` list.

As mentioned above, drops are always detected through timers. To this effect, whenever a packet is sent to the receiver and placed in the `to-be-ack` list, it is timestamped. When a packet remains in the `to-be-ack` list more than a certain amount of time it is assumed dropped. In particular, we assume that a packet was dropped at time t when t exceeds the packet's time-stamp in the `to-be-ack` list plus an estimated maximum possible round-trip-time `mxrtt`.

As packets are sent and ACKs received, an estimate `mxrtt` of the maximum possible round-trip-time is continuously updated. The estimate used is given by:

$$\texttt{mxrtt} := \beta \times \texttt{ewrtt},$$

where β is a constant larger than 1 and `ewrtt` an exponentially weighted average of past RTTs. Whenever a new ACK arrives, we update `ewrtt` as follows:

$$\texttt{ewrtt} = \max\left\{\alpha^{\frac{1}{\texttt{cwnd}}} \times \texttt{ewrtt}, \texttt{sample-rtt}\right\}, \quad (1)$$

where α denotes a positive constant smaller than 1, `cwnd` the current window size, and `sample-rtt` the RTT for the packet whose acknowledgment just arrived[5]. The reason to raise α to the power $1/\texttt{cwnd}$ is that in one RTT the formula in (1) is iterated `cwnd` times. This means that, e.g., if there were a sudden decrease in the RTT then `ewrtt` would decrease by at a rate of $(\alpha^{\frac{1}{\texttt{cwnd}}})^{\texttt{cwnd}} = \alpha$ per RTT, independently of the current value of the congestion window. The parameter α can therefore be interpreted as a memory factor in units of RTTs. Note that `ewrtt` is not a smoothed version RTT. Hence, this approach is not like early versions of TCP's RTO calculation that were solely based on estimates of the mean RTT. To the contrary, if a past RTT observation is large, this large RTT is responsible for the value of `ewrtt` for sometime. In this way, `ewrtt` will reflect spikes in RTT. This is similar to Van Jacboson's algorithm where recent variations in RTT have a significant impact on RTO. As discussed in [5], the performance of the algorithm is actually not very sensitive to changes in the parameters β and α, provided they are chosen in appropriate ranges.

Two modes exist for the update of the congestion window: and . The sender always starts in and will only go back to after periods of extreme losses (cf. Section 3.2). In this mode, `cwnd` starts with the value one and increases exponentially (one for each ACK received). Once the first loss is detected, `cwnd` is halved and the sender transitions to the mode, where `cwnd` increases linearly ($1/\texttt{cwnd}$ for each ACK received). Subsequent drops cause further halving of `cwnd`, without the sender ever leaving . An important but subtle point in halving `cwnd` is that when a packet is sent, not only a time-stamp but the current value of `cwnd` is saved in the `to-be-ack` list. When a packet drop is detected, then `cwnd` is actually set equal to half the value of `cwnd` at the time the packet was sent and not half the current value of `cwnd`. This makes the algorithm fairly insensitive to the delay between the time a drop occurs until it is detected.

To prevent bursts of drops from causing excessive decreases in `cwnd`, once a drop is detected a snapshot of the `to-be-sent` list is taken and saved into an auxiliary list called `memorize`. As packets are acknowledged or declared as dropped, they are removed from the `memorize` list so that this list contains only those packets that were sent before `cwnd` was halved and have not yet been unaccounted for. When a packet in this list is declared dropped, it does not cause `cwnd` to be halved. The rational for this is that the sender already reacted to the congestion that caused that burst of drops. This type of reasoning is also present in TCP-NewReno and TCP-SACK.

In TCP-PR packets are only sent when the congestion window allows it. In particular, when `cwnd` exceeds the number of packets in the `to-be-ack` list. In practice, this means that most packets are sent when an acknowledgment arrives and a packet is removed from the

[5]To compute $x := \alpha^{\frac{1}{\texttt{cwnd}}}$ in a Linux kernel, we employ Newton's method to solve the equation $x^{\texttt{cwnd}} = \alpha$. This leads to the following loop

```
1   $x := 1$
2   for $i := 1$ to $n$
3       $x := \frac{\texttt{cwnd}-1}{\texttt{cwnd}} x + \frac{\alpha}{\texttt{cwnd}\, x^{\texttt{cwnd}-1}}$
4   end
```

In our implementation, we used $n = 2$.

`to-be-ack` list. Therefore, TCP-PR exhibits the type of self-clocking common to other versions of TCP. Packets may also be sent when a drop is detected since when this happens a packet is also removed from the `to-be-ack` list.

The pseudo-code in Table 1 corresponds to the algorithm just described. Table 2 summarizes the notation used in the code.

Remark 1 *From a computational view-point, TCP-PR is more demanding than TCP-(New)Reno. This additional complexity is due to the computation of* `ewrtt`*, which involves approximating* $\alpha^{\frac{1}{cwnd}}$ *using Newton's method. We currently only use two iterations for the approximation algorithm, so the added computation is very small. In terms of storage requirements, TCP-PR has similar complexity to TCP-(New)Reno and SACK as they make use of the kernel socket buffers (e.g.,* `sk_buff` *in Linux) data structure to store packets. The* `memorize` *list can be implemented through a flag that marks the packets in the* `to-be-ack` *list that should also be in the* `memorize` *list. In the case of Linux, unused parts of the* `sk_buff` *data structure can be used to hold the value of the flag. Hence, no additional memory is required.*

3.2 Extreme Losses

When half (or more) packets are lost within a window, TCP-NewReno/SACK will time-out in fast-recovery mode. This is because not enough ACKs are received for the congestion window to open and allow for the sender to perform the needed retransmissions. Eventually a time-out occurs. When this happens persistently, these protocols start an exponential back-off of the time-out interval until packets are able to get through.

The "correct" behavior of congestion control under extreme losses is somewhat controversial and perhaps the more reasonable approach is to leave to the application to decide what to do in this case. However, and to be compatible with previous versions of TCP, we propose a version of TCP-PR that resets `cwnd` to one and performs exponential back-off under extreme loss conditions.

We detect extreme losses by counting the number of packets lost in a burst. This can be done using a counter `cburst` that is incremented each time a packet is removed from the `memorize` list due to drops and reset to zero when this list becomes empty. We recall that this list is usually kept empty but when a drop occurs it "memorizes" the packets that were outstanding. In the spirit of TCP-NewReno and TCP-SACK, packets from this list that are declared dropped do not lead to further halving of the congestion window.

To emulate as close as possible what happens with TCP-NewReno and SACK, when `cburst` (and therefore the number of drops in a burst) exceeds $\texttt{cwnd}/2 + 1$, we reset $\texttt{cwnd} = 1$ and transition to the [illegible] mode. Moreover, and for fairness with implementations of TCP-NewReno/SACK that use coarse-grained timers, we increase `mxrtt` to one second and delay sending packets by `mxrtt` [1]. If further (new) drops occur while $\texttt{cwnd} = 1$, instead of dividing `cwnd` by two, we double `mxrtt`, which emulates the usual exponential back-off. The reader is referred to [5] for the pseudo-code that implements this algorithm.

4 Performance without Packet Reordering: Performance and Fairness

Two issues arise when considering TCP-PR over networks without packet reordering: performance and fairness. The first issue is whether TCP-PR performs as well as other TCP implementations under "normal" conditions, i.e., no packet reordering. Specifically, for a fixed topology and background traffic, does TCP-PR achieve similar throughput as standard TCP implementations? The second concern is whether TCP-PR and standard TCP implementations are able to coexist fairly. To some extent, the fairness issue encompasses the performance issue: if TCP-PR competes fairly against standard TCP implementations in a variety of network conditions, then it seems reasonable that TCP-PR and other TCP implementations are able to achieve similar throughput (and thus perform similarly) when exposed to similar network conditions. Therefore, while this section focuses on fairness, it indirectly addresses the performance issue. Additionally, in Section 5, we also show that, when no packet reordering occurs, TCP-PR achieves the same throughput as other TCP implementations. While the TCP implementations studied in Section 5 are not the standard implementations of today, it has been shown that when no packet reordering occurs, these implementations perform exactly the same as standard TCP [3].

We have performed extensive ns-2 [19] simulations to show that, for a wide range of network conditions and topologies, TCP-PR is fair to standard TCP implementations. In this section, a sample of our simulation results is presented with attention focused on the compatibility with TCP-SACK [16] over two topologies. The first topology used is the dumbbell topology, which is also known as *single-bottleneck* as it includes just one bottleneck link. A number of simulation-based studies have used the dumbbell topology to evaluate the performance of network protocols. One recent example is the comparison between the performance (including

Event		Code
initialization	1	mode :=
	2	cwnd := 1
	3	ssthr := $+\infty$
	4	memorize := $\emptyset$
time $>$ time(n) + mxrtt (drop detected for packet n)	5	*to-be-ack*, n
	6	*to-be-sent*, n
	7	*memorize*, n
	8	*memorize* := *to-be-ack*
	9	*cwnd* := *cwnd*$(n)/2$
	10	*ssthr* := *cwnd*
	11	
	12	*memorize*, n
	13	*flush-cwnd()*
ACK received for packet n	14	*ewrtt* $= \max\left\{\alpha^{\frac{1}{cwnd}} \times$ *ewrtt*, *time* $-$ *time*$(n)\right\}$
	15	*mxrtt* := $\beta \times$ *ewrtt*
	16	*to-be-ack*, n
	17	*memorize*, n
	18	*mode* = *cwnd* $+ 1 \leq$ *ssthr*
	19	*cwnd* := *cwnd* $+ 1$
	20	
	21	*mode* :=
	22	*cwnd* := *cwnd* $+ 1/$*cwnd*
	23	*flush-cwnd()*
flush-cwnd()	24	while cwnd $>$ |to-be-ack| do
	25	k=send(to-be-sent)
	26	remove(to-be-sent, k)
	27	add(to-be-ack, k)

Table 1. Pseudo-code for TCP-PR (cf. notation used in Table 2)

fairness) of TCP-SACK and an implementation of the "TCP-friendly" formula [10]. The second topology we use is the *parking-lot* topology, which, like the dumbbell, has also been employed in a number of recent performance studies of network protocols including [9] and [14]. Parking-lot is a generalization of the dumbbell topology as it includes multiple bottleneck links. Figure 1 shows the parking-lot topology we used, including the source and destination nodes for the long-lived TCP-SACK flows that were employed as cross traffic. We should point out that previous studies that used the parking-lot topology only included cross traffic flowing between node pairs CS1$\rightarrow$CD1, CS2$\rightarrow$CD2, and CS3$\rightarrow$CD3. We have these as well as between CS1$\rightarrow$CD2, CS1$\rightarrow$CD3, and CS2$\rightarrow$CD3.

Following the approach taken in [10], the fairness of TCP-PR to TCP-SACK is judged by simulating an equal number of TCP-PR and TCP-SACK flows. These flows have a common source and destination. The steady state fairness can be quantified with a single number, the *mean normalized throughput*. If there are n flows, then the *normalized throughput* of flow i is

$$T_i = \frac{x_i}{\frac{1}{n}\sum_{j=1}^{n} x_j},$$

where the throughput, x_i, is the total data sent during the last 60 seconds of the simulation. The mean normalized throughput for a particular protocol is the average value of T_i, averaged over all the flows of that protocol. Note that if $T_i = 1$, then flow i has received the average throughput. Similarly, if the mean normalized throughout is one, then the two implementations received the same average throughput.

Figure 2 shows the normalized throughput and mean normalized throughput for various numbers of TCP-PR and TCP-SACK flows. Results from the dumbbell and parking-lot topologies are shown in the right and left-hand plots, respectively. In these experiments, TCP-PR α and β were fixed at 0.995 and 3.0, respectively. From the graphs, it is clear that the two versions of TCP-PR and TCP-SACK compete fairly over a wide range of traffic conditions and thus exhibit similar performance.

While the mean normalized throughput describes the average behavior of all flows, the *coefficient of variation*

`time`	current time
`time`(n)	time at which time packet n was sent
`cwnd`(n)	congestion window at the time packet n was sent
is-in(`list`, k)	returns true if the packet k is in the list `list`
add(`list`, k)	add the packet k to the list `list`
remove(`list`, k)	remove the packet k from list `list` (if k is not in `list` do nothing)
$\vert$`list`$\vert$	number of elements in the list `list`
k=send(`list`)	send the packet in list `list` with smallest seq. number, returning the seq. number

Table 2. Notation used in Table 1

describes the variation of the throughput. Specifically, let I be the set of flows of a particular protocol. The coefficient of variation is

$$CoV = \frac{1}{\sum_{i\in I} T_i}\sqrt{\sum_{i\in I}\left(T_i - \frac{1}{|I|}\sum_{i\in I} T_i\right)^2},$$

where $|I|$ is the number of elements in the set I. Figure 3 shows the coefficient of variation for ten simulations as well as the mean coefficient of variation for the simulation set. From Figures 2 and 3, we conclude that the mean and variance of the throughput for TCP-PR and TCP-SACK are similar.

Surprisingly, fairness is maintained for a wide range of α and β. Figure 4 shows TCP-SACK's mean normalized throughput for different values of α and β. For these simulations, the number of flows was held constant at 64 total flows (32 TCP-SACK and 32 TCP-PR flows). Note that for $\beta = 1$, TCP-SACK exhibits better throughput. However, for β larger than 1, both implementations achieve nearly identical performance. A large number of simulations show that these results are consistent for different levels of background traffic and different topologies. We noticed that even in situations where cross traffic causes extreme loss conditions (over 15% drop probability), TCP-SACK only gets up to 20% more throughput when $\beta = 10$, while the throughputs are the same for $1 < \beta < 5$. Such extreme loss are not particularly relevant since TCP's throughput is very low when loss probability is large.

These results under normal traffic conditions are not so much evidence of the remarkableness of TCP-PR, but rather they attest to the robustness of the AIMD scheme. The important feature of the AIMD scheme is that if two flows detect drops at the *same* rate, then their congestion windows will converge. It is shown in [7] and, in more detail, in [4] that TCP flows over a dumbbell topology will converge to the same bandwidth exponentially fast. While these proofs rely on the protocols being identical, they also point to the inherent stability of the AIMD scheme which is witnessed in the simulation results presented here.

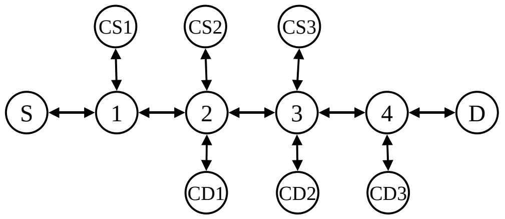

Figure 1. Parking-lot topology with multiple bottlenecks and cross traffic. The source and destination are labeled `S` and D respectively. The cross-traffic connections are `CS1`→`CD1`, `CS1`→`CD2`, `CS1`→`CD3`, `CS2`→`CD2`, `CS2`→`CD3`, and `CS3`→`CD3`. The link bandwidths are: `CS1`→`1`=5Mbps, `CS2`→`2`=1.66Mbps, `CS3`→`3`=2.5Mbps, and all other bandwidths are 15Mbps. For these values, the three links `1`→`2`, `2`→`3`, and `3`→`4` become bottleneck links.

5 Performance under Packet Reordering: Comparison with other Methods

This section compares the performance of TCP-PR against existing algorithms that try to make TCP more robust to packet reordering. As before, we run extensive simulations using ns-2 to compare the performance of these methods in the face of persistent packet reordering due to multi-path routing. The topology for this comparison is shown in Figure 5. Two classes of simulations were performed, the first set fixed the propagation delay for each link at 10ms, while the second set fixed the propagation delay at 60ms.

Many multi-path routing strategies are possible over this topology. We have developed a family of strategies that is parameterized by a single variable ε (see [12, 6] for details). This parameter controls the degree to which

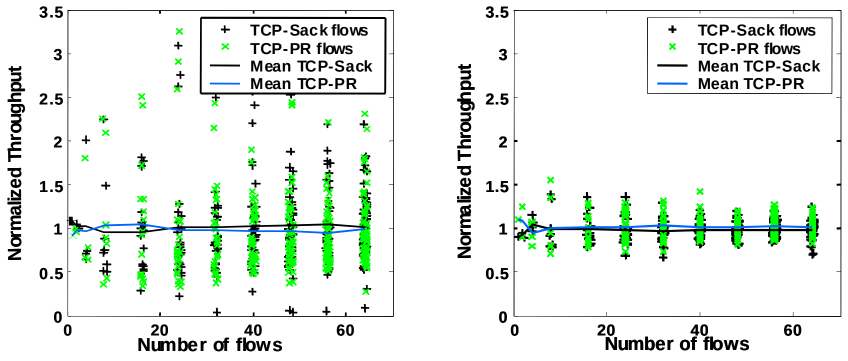

Figure 2. Fairness of TCP-PR competing with TCP-SACK. Normalized throughput for the dumbbell and parking-lot topologies are shown in the left and right plots, respectively.

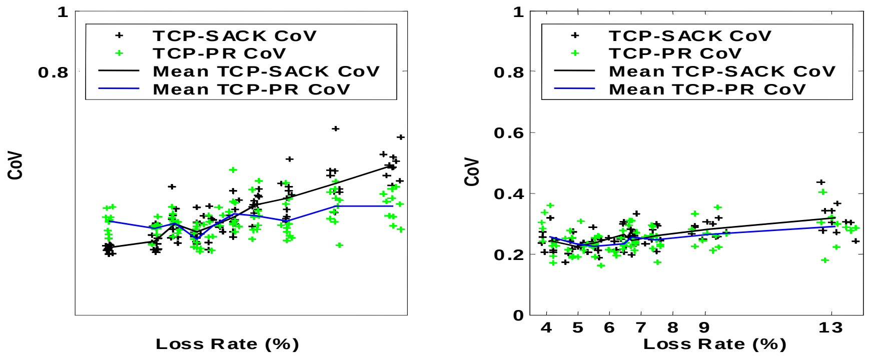

Figure 3. Coefficient of variation. The coefficient of variation as a function of packet loss probability. The variation in loss probability was simulated by decreasing the link bandwidth. The left plot is the coefficient of variation for the dumbbell topology and the right plot is for the parking lot topology.

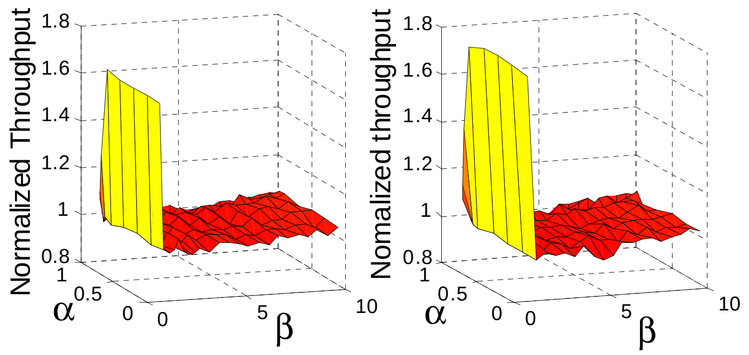

Figure 4. TCP-SACK normalized throughput for different TCP-PR parameters. The left plot shows the mean normalized throughput of TCP-SACK over the dumbbell topology, while the right plot shows the normalized throughput for the parking lot topology.

delay is taken into consideration when the routing is designed. In particular, when $\varepsilon = \infty$, delay is heavily penalized and shortest path routing is used, whereas when $\varepsilon = 0$, delay is not penalized at all and full multi-path routing is used, leading to all independent paths from source to destination being used with equal probability. Intermediate values of ε correspond to compromises between these two extreme cases.

For a fixed routing strategy (a fixed ε), TCP-PR and TCP with various `dupthresh` compensation schemes discussed in [3] were each tested independently, hence, only one flow was active at a time. Furthermore, for these simulations, there was no background traffic. The rationale behind these choices is that the objective in this section is, rather than compare how the different versions of TCP interact with each other, to investigate how the different methods are able to cope with persistent packet reordering.

Figure 6 shows the throughput for various values of ε. These simulations show that for $\varepsilon = 0$ (full multi-path routing), most of the other protocols suffer drastic decrease in throughput. For $\varepsilon = 500$ (single-path routing), all methods achieve the same throughput. The exception is time-delayed fast-recovery (TD-FR), which still achieves a reasonable throughput for small values of ε if the propagation delay is small (the left plot in Figure 6). However, as the propagation delay is increased, the throughput decrease. To some degree this decrease in throughput is due to an increase in the round-trip time. (Notice that at $\varepsilon = 500$, all the throughputs are smaller on the right than on the left.) However, at $\varepsilon = 0$, TD-FR suffers a very large drop in throughput when the propagation delay is increased. The reason for this drop in throughput is that TD-FR makes use of both `dupthresh` and timers. Specifically, the `dupthresh` is larger when the round-trip time is larger. As discussed in [3], an increase in the `dupthresh` can lead to burstiness. While the "limited transmit algorithm" attempts to reduce the burstiness, burstiness remains a problem for TD-FR over connections with long latency. These simulations demonstrate the effectiveness of TCP-PR's timer-based packet drop detection. While DUPACKS are indicative of packet loss in single path routing, their occurrence convey little when multi-path routing is utilized.

Recently another method for adapting `dupthresh` has been suggested [21]. Since the simulation implementation of this method is not yet available, it was not included in this comparison.

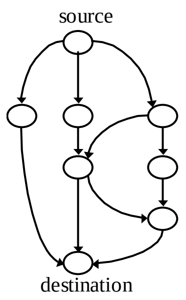

Figure 5. A topology to compare TCP implementations. Each link has a bandwidth of 10Mbps and queue has a size of 100 packets. Two types of simulations were performed. In one, the delay for each link was 10ms and in the other, the delay was 60ms.

6 Conclusions

In this paper we proposed and evaluated the performance of TCP-PR, a variant of TCP that is specifically designed to handle persistent reordering of packets (both data and acknowledgment packets). Our simulation results show that TCP-PR is able to achieve high throughput when packets are reordered and yet is fair to standard TCP implementations, exhibiting similar performance when packets are delivered in order. From a computational view-point, TCP-PR is more demanding than TCP-(New)Reno but carries essentially the same overhead as TCP-SACK.

Because of its robustness to persistent packet reordering, TCP-PR would work well if mechanisms that introduce packet reordering as part of their normal operation were deployed on the Internet. Such mechanisms include proposed enhancements to the original Internet architecture such as multi-path routing for increased throughput, load balancing, and security; protocols that provide differentiated services (e.g., DiffServ [2]); and traffic engineering approaches.

Furthermore, TCP-PR will work well in wireless multi-hop environments allowing wireless routing protocols to make use of multiple paths when available. While the protocol described in this paper focuses on wired networks, we plan to adapt it for wireless environments as part of our future work.

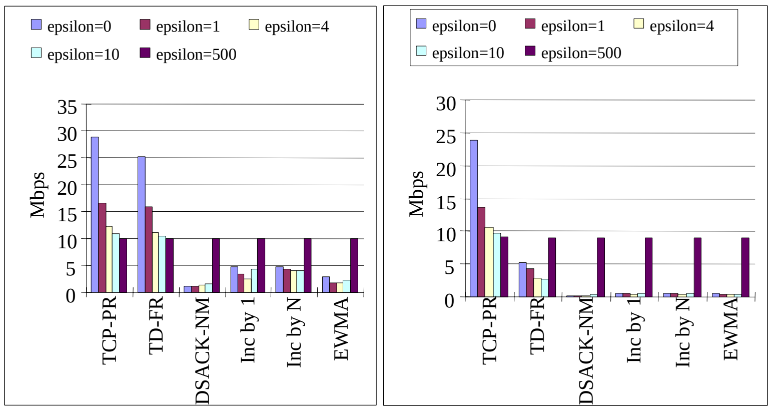

Figure 6. Throughput for different TCP implementations and different degrees of multi-path routing for the topology in Figure 5. Single path routing corresponds to $\varepsilon = 500$. For smaller ε, alternative paths are used more frequently and in the limit, $\varepsilon = 0$, all paths are used with equal probability. The propagation delays were the same for all links, equal to 10ms for left plot and 60ms for the right plot.

References

[1] M. Allman and V. Paxson. Computing TCP's retransmission timer. *RFC 2988*, page 13, Nov. 2000.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Whang, and W. Weiss. An architecture for differentiated services. RFC 2475, 1998.

[3] E. Blanton and M. Allman. On making TCP more robust to packet reordering. *ACM Computer Communications Review*, 32, 2002.

[4] S. Bohacek, J. Hespanha, K. Obraczka, and J. Lee. Analysis of a TCP hybird model. In *Proceedings of the 39th Annual Allerton Conference on Communication, Control and Computing*, 2001.

[5] S. Bohacek, J. P. Hespanha, J. Lee, C. Lim, and K. Obraczka. TCP-PR: TCP for persistent packet reordering. Extended version. Technical report, University of California, Santa Barbara, May 2003. Available at .

[6] S. Bohacek, J. P. Hespanha, and K. Obraczka. Saddle policies for secure routing in communication networks. In *Proc. of the 41th Conf. on Decision and Contr.*, Dec. 2002.

[7] D. Chiu and R. Jain. Analysis of the Increase/Decrease algorithms for congestion avoidance in computer networks. *Journal of Computer Networks and ISDN*, 17:1–14, 1989.

[8] T. Dyer and R. Boppana. A comparison of TCP performance over three routing protocols for mobile ad hoc networks. In *Proc. of the ACM MOBIHOC*, 2001.

[9] S. Floyd. Connections with multiple congested gateways in packet-switched networks part 1: One-way traffic. *ACM Computer Communication Review*, 21(5):30–47, Oct. 1991.

[10] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proc. of the ACM SIGCOMM*, 2000.

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

[12] J. P. Hespanha and S. Bohacek. Preliminary results in routing games. In *Proc. of the 2001 Amer. Contr. Conf.*, June 2001.

[13] G. Holland and N. Vaidya. Analysis of TCP performance over mobile ad-hoc networks. In *Proc. of the ACM MOBICOM*, 1999.

[14] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for future high bandwidth-delay product environments. In *Proc. of the ACM SIGCOMM*, Aug. 2002.

[15] R. Ludwig and R. Katz. The Eifel algorithm: Making TCP robust against spurious retransmissions. *ACM Computer Communication Review*, 30(1), 2000.

[16] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgement options. RFC 2018, 1996.

[17] V. Paxson. End-to-end routing behavior in the Internet. In *Proc. of the ACM SIGCOMM*, 1996.

[18] V. Paxson. End-to-end Internet packet dynamics. In *Proc. of the ACM SIGCOMM*, 1997.

[19] The VINT Project, a collaboratoin between researchers at UC Berkeley, LBL, USC/ISI, and Xerox PARC. *The ns Manual (formerly ns Notes and Documentation)*, Oct. 2000. Available at .

[20] F. Wang and Y. Zhang. Improving TCP performance over mobile ad-hoc networks with out-of-order detection and response. In *Proc. of the ACM MOBIHOC*, 2002.

[21] N. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A reordering-robust TCP with DSACK. Technical Report TR-02-006, ICSI, Berkeley, CA, July 2002.