# TCP-PR: TCP for Persistent Packet Reordering*

## Extended Version

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*

February 2002[†]

### 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-or-order acknowledgements) has no effect on TCP-PR's performance.

Through extensive simulations, we show that evaluate TCP-PR performs consistently better than existing mechanisms that try to make TCP more robust to packet reordering. In 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

[*]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.

[†]Revised on March 2003.

rate by decreasing its *congestion window.* TCP uses two strategies for detecting packet loss. The first one is based on the sender's retransmission timeout (RTO) expiring and is sometimes referred to as *coarse timeout.* 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 timeout and reduces its congestion window in half. The basic idea behind fast retransmit is to improve TCP's throughput by avoiding the sender to timeout (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 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 [1]. 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 [2]. 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 [3, 4, 5].

A number of mechanisms that have been recently proposed to enhance the original Internet architecture also result in (persistent) packet reordering. Multi-path 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

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

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 [6], 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 [7] 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 time-stamp 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 [8] 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 [1], 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 [9].

Time-delayed fast-recovery (TD-FR), which was first proposed in [10] and analyzed in [1], 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 [3], 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.

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, they were not intended to be used when packet reordering is persistent.

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

[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 [3], one way the TCP receiver can notify the sender is by setting a *OOO* bit in the TCP ACK packet

packet is assumed to be lost[5]. 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. This is discussed in Section 4. The second challenge, covered in Section 5, is to maintain fairness with current implementations of TCP. In Section 6, 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. 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 $\texttt{to}-\texttt{be}-\texttt{sent}$ list contains all the packets whose transmission is pending, waiting for an "opening" in the congestion window. The $\texttt{to}-\texttt{be}-\texttt{ack}$ 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 $\texttt{to}-\texttt{be}-\texttt{sent}$ list; when the congestion window allows it, the packet is sent to the receiver and moved to the $\texttt{to}-\texttt{be}-\texttt{ack}$ list; finally when an ACK for that packet arrives from the receiver, it is removed from the $\texttt{to}-\texttt{be}-\texttt{ack}$ list (under cumulative ACKs, many packets will be simultaneously removed from $\texttt{to}-\texttt{be}-\texttt{ack}$). Alternatively, when it is detected that a packet was dropped, it is moved from the $\texttt{to}-\texttt{be}-\texttt{ack}$ list back into the $\texttt{to}-\texttt{be}-\texttt{sent}$ 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 $\texttt{to}-\texttt{be}-\texttt{ack}$ list, it is timestamped. When a packet remains in the $\texttt{to}-\texttt{be}-\texttt{ack}$ 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 timestamp in the $\texttt{to}-\texttt{be}-\texttt{ack}$ list plus an estimated maximum possible round-trip-time $\texttt{mxrtt}$.

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

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

[5] This means that out-of-order ACKs are automatically taken care of.

where $\beta$ is a constant larger than 1 and $\texttt{srtt}$ an exponentially weighted average of past RTTs. Whenever a new ACK arrives, we update $\texttt{srtt}$ as follows:

$$\texttt{srtt} = \max\left\{\alpha^{\frac{1}{\texttt{cwnd}}} \times \texttt{srtt}, \texttt{sample} - \texttt{rtt}\right\}, \tag{1}$$

where $\alpha$ denotes a positive constant smaller than 1, $\texttt{cwnd}$ the current window size, and $\texttt{sample} - \texttt{rtt}$ the RTT for the packet whose acknowledgment just arrived. The reason to raise $\alpha$ to the power $1/\texttt{cwnd}$ is that in one RTT the formula in (1) is iterated $\texttt{cwnd}$ times. This means that, e.g., if there were a sudden decrease in the RTT then $\texttt{srtt}$ 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 $\alpha$ can therefore be interpreted as a smoothing factor in units of RTTs. As discussed in Section 4, the performance of the algorithm is actually not very sensitive to changes in the parameters $\beta$ and $\alpha$, provided they are chosen in appropriate ranges.

Two modes exist for the update of the congestion window: $slow - start$ and $congestion - avoidance$. The sender always starts in $slow - start$ and will only go back to $slow - start$ after periods of extreme losses (cf. Section 3.2). In this mode, $\texttt{cwnd}$ starts with the value one and increases exponentially (one for each ACK received). Once the first loss is detected, $\texttt{cwnd}$ is halved and the sender transitions to the $congestion - avoidance$ mode, where $\texttt{cwnd}$ increases linearly ($1/\texttt{cwnd}$ for each ACK received). Subsequent drops cause further halving of $\texttt{cwnd}$, without the sender ever leaving $congestion - avoidance$. An important but subtle point in halving $\texttt{cwnd}$ is that when a packet is sent, not only a timestamp but the current value of $\texttt{cwnd}$ is saved in the $\texttt{to} - \texttt{be} - \texttt{ack}$ list. When a packet drop is detected, then $\texttt{cwnd}$ is actually set equal to half the value of $\texttt{cwnd}$ at the time the packet was sent and not half the current value of $\texttt{cwnd}$. 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 $\texttt{cwnd}$, once a drop is detected a snapshot of the $\texttt{to} - \texttt{be} - \texttt{sent}$ list is taken and saved into an auxiliary list called $\texttt{memorize}$. As packets are acknowledged or declared as dropped, they are removed from the $\texttt{memorize}$ list so that this list contains only those packets that were sent before $\texttt{cwnd}$ was halved and have not yet been unaccounted for. When a packet in this list is declared dropped, it does not cause $\texttt{cwnd}$ 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.

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 because it requires the sender to maintain the list* $\texttt{to} - \texttt{be} - \texttt{ack}$ *of packets whose acknowledgment is pending, but is not significantly more demanding than TCP-SACK. It does maintain the extra list* $\texttt{memorize}$ *that is used to detect drop bursts, but this list is empty most of the time.*

### 3.2 Extreme losses

When a large number of packets are lost, TCP-NewReno or SACK often lead to timeouts. In particular, when approximately half (or more) packets are lost within a window, TCP-NewReno/SACK will timeout in fast-recovery mode. This is because, not enough ACKs are received for the congestion window to open and allow for the retransmissions needed and eventually a timeout occurs. When this happens persistently, these protocols start an exponential back-off of the timeout 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 $slow - start$ 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` [11]. 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 pseudo-code in Table 3 implements this algorithm. In this pseudo-code, we also inhibited increments of `cwnd` while the list `memorize` is not empty. This was also done to improve fairness with respect to TCP-NewReno/SACK, because it emulates the fact that in these algorithms `cwnd` only goes back to the usual increase of $1/\texttt{cwnd}$ per ACK after the sender leaves fast-recovery mode.

## 4 Selection of TCP-PR Parameters

In this section we discuss the selection of the $\alpha$ and $\beta$ parameters used by TCP-PR to estimate the maximum round-trip time `mxrtt` in order to achieve high throughput. When the time elapsed since a packet was sent exceeds a threshold and its acknowledgment has not yet arrived, TCP-PR assumes the packet was dropped and divides the congestion window by two. However, there is the risk that if the threshold is incorrectly set, the algorithm will assume that a packet has been lost when it merely experienced a large round-trip time. We refer to such events as *spurious timeouts*. While occasional spurious timeouts are of little consequence, the throughput may suffer severely if they occur too frequently. We examine two situations where spurious timeouts may occur. The first is caused by rapid variations of the round-trip time, in particular, sudden increases of the round-trip time, possibly leading to a division by two of the congestion window. The second (and more challenging) situation arises under multi-path routing when one of the paths is rarely used and has significantly larger latency than the others.

## 4.1 Parameter Selection for Variable Round-Trip Time

Under single path routing, a spurious timeout may occur when there is a large jump in the round-trip time. Although a slow increase in the round-trip time is easily absorbed by the averaging provided by (1), sudden increases may lead to spurious timeouts. Thus, it is not the variance of the round-trip time, but rather its high frequency component that may lead to spurious timeouts.

While rapidly changing round-trip times can lead to spurious timeouts, large transmission and propagation delay can make spurious timeouts rare. To see this, we determine the probability of a spurious timeout. Let $R_k$ be the round-trip time experienced by the $k^{\text{th}}$ packet. The sequence of round-trip times is $\{R_k : -\infty < k < \infty\}$. For ease of presentation, we assume that the congestion window is constant and takes the value `cwnd`. This assumption has little impact on the final result. Then

$$P\left(\text{spurious timeout}\right) = P\Big(R_0 > \max\left(\beta R_{-1}, \beta\alpha^{1/\texttt{cwnd}} R_{-2}, \cdots, \beta\alpha^{(k-1)/\texttt{cwnd}} R_{-k}, \cdots\right)\Big).$$

The round-trip time can be decomposed into two parts: a fixed component that depends on propagation and transmission delay, and a time-varying one, which is mostly due to queuing delay. We denote the former by $T$ and the probability density function and cumulative distribution of the latter by $p$ and $F$, respectively. By making the "worst-case" assumption that the round-trip times are independent, we conclude that the probability of a timeout is given by

$$\begin{aligned}
&P\left(\text{spurious timeout}\right)\\
&= P(R_0 > \beta R_{-1}) P(R_0 > \beta\alpha^{1/\texttt{cwnd}} R_{-2}) \cdots P(R_0 > \beta\alpha^{(k-1)/\texttt{cwnd}} R_{-k}) \cdots\\
&= P\Big(R_{-1} < \frac{R_0}{\beta}\Big) P\Big(R_{-2} < \frac{R_0}{\beta\alpha^{1/\texttt{cwnd}}}\Big) \cdots P\Big(R_{-k} < \frac{R_0}{\beta\alpha^{(k-1)/\texttt{cwnd}}}\Big) \cdots\\
&= \int_T^\infty p(r-T) \prod_{k=0}^{\infty} F\Big(\frac{r}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}} - T\Big) dr\\
&= \int_0^\infty p(u) \prod_{k=0}^{\infty} F\Big(\frac{u+T}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}} - T\Big) du = \int_0^\infty p(u) \prod_{k=0}^{\infty} F\Big(\frac{u+T\left(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\right)}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}}\Big) du.
\end{aligned} \tag{2}$$

Since $\prod_{k=0}^{\infty} F\Big(\frac{u+T\left(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\right)}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}}\Big) \neq 0$ only if $u + T\big(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\big) > 0$ for all $k$, we can restrict our attention to the case $u > T(\beta - 1)$. Thus,

$$P\left(\text{spurious timeout}\right) = \int_{T(\beta-1)}^{\infty} p\left(u\right) \prod_{k=0}^{\infty} F\Big(\frac{u+T\big(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\big)}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}}\Big) du \leq \int_{T(\beta-1)}^{\infty} p\left(u\right) du.$$

Chebyshev's inequality then yields

$$P\left(\text{spurious timeout}\right) \leq \frac{\sigma^2}{\left(T\left(\beta-1\right)-\mu\right)^2}, \tag{3}$$

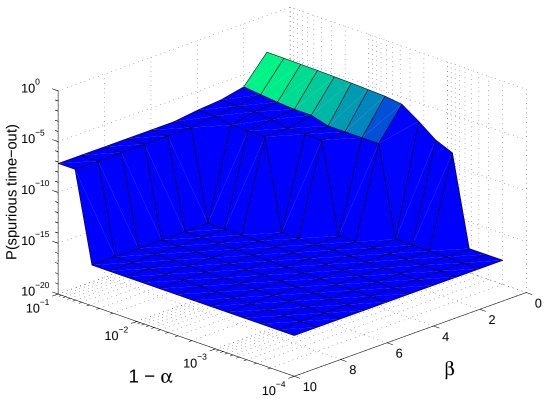

Figure 1: Probability of Spurious timeout. This plot is from the NLANR data set. For many pairs of $\alpha$, $\beta$ there were no spurious timeouts. However, in order to view the data on a log scale a perturbation of $10^{-16}$ has added. Hence, all the pairs $\alpha$, $\beta$ that show a probability of $10^{-16}$ actually had no spurious timeouts at all.

where $\mu$ and $\sigma^2$ denote the mean and variance of the variable part round-trip time and it is assumed that $T(\beta - 1) > \mu$. From (3), we conclude that when the variance is small, $\beta > 1$, and the propagation delay is large, then spurious timeouts will be rare. This result implies, for example, that over high performance networks such as Internet 2, where delay is dominated by the time-invariant component of the round-trip time (therefore $\sigma$ is small), TCP-PR will have few spurious timeouts. Note, that this is a "worst-case" estimate since we assumed that the round-trip times are I.I.D. The probability of spurious timeouts should be lower when the round-trip time is sampled frequently, as happens in the TCP-PR algorithm. In particular, (3) does not imply that if the variance is large then there will be a larger probability of spurious timeouts.

While (2) and (3) can be evaluated for different distribution of round-trip times, more insight is gained by analyzing real data. Using the July 25, 2001 snapshot of round-trip times from the NLANR data set [12], we estimated the probability of spurious timeouts. The total data set consists of nearly 13000 connections between 122 sites and 17.5 million round-trip time measurements. This data consisted of time series of round-trip times for each connection with each time series containing 1440 round-trip times sampled once a minute over the entire day. For each time series, the `srtt` and `mxrtt` were computed and spurious timeouts noted. This process was repeated for several values of $\alpha$ and $\beta$. Figure 1 shows the probability of a drop versus the parameters $\alpha$ and $\beta$. For these computations, it was assumed that `cwnd` = 1 (a conservative assumption). Furthermore, since the round-trip time was sampled once a minute, there was low correlation between closely spaced round-trip time measurements that are usually present when TCP-PR observes the round-trip time. Since, high correlation can reduce spurious timeouts, this data provides a "worst-case" analysis of spurious timeout. Despite this, Figure 1 shows that as long as $\beta > 1$, the probability of a spurious timeout occurring is less than $10^{-7}$.

## 4.2 Packet Reordering Due to Multi-path Routing

The most important feature of TCP-PR is the ability to maintain high throughput in the face of persistent packet reordering. When massive reordering occurs, the round-trip times experienced by packets will be dissimilar. This variation of the observed round-trip time will be especially significant when packets take different paths as a result of multi-path routing.

Figure 2 shows a topology over which multi-path routing was implemented using the ns-2 [13] network simulator. The routing is such that 1% of the packets take the path with much longer latency. Thus, the round-trip times observed by the sender are typically small, but occasionally quite large. While simple, this type of routing/topology, with one long delay path that is used infrequently and a much shorter delay path used most frequently, characterizes a worst-case scenario. As will be explained, cross-traffic and intermediate routers are of little consequence. The reason that such a topology can lead to spurious timeouts is that since the maximum round-trip time estimate `mxrtt` uses a smoothed version of the round-trip time, there is a risk that the smooth round-trip time will "forget" the occasional long round-trip time and trigger a reduction of `cwnd` whenever a packet takes the path with longer delay. If these spurious timeouts occur too frequently, then the throughput will suffer. Figure 3 shows the throughput of TCP-PR over this worst-case topology/routing scenario for different parameters $\alpha$ and $\beta$. Note that the throughput is maintained as long as $\alpha$ or $\beta$ are large. This conclusion is reasonable since if $\alpha$ is near to one, the past values of the round-trip time are remembered for a long time. Furthermore, even if the smoothed round-trip time "forgets" the occasional large round-trip time, the rate at which spurious timeouts occur can still be reduced by increasing $\beta$.

This worst-case scenario can also be studied analytically. Suppose that one path has significantly more delay than the other paths. Thus, if this path with the long delay is used infrequently, there is a risk that when it is used, the long delay will trigger a spurious timeout. Suppose that the longer path is used with probability $\rho$ and spurious timeouts only occur when packets take this path. As above, assume that the congestion window is constant. This assumption merely simplifies the presentation of the analysis, as will be noted, it has little impact of the results. Under these assumptions, the probability of a spurious timeout is given by

$$\begin{aligned}
&P\left(\text{spurious timeout}\right)\\
&= P\left(R_0 > \max\left(\beta R_{-1}, \beta\alpha^{1/\mathtt{cwnd}}R_{-2}, \beta\alpha^{2/\mathtt{cwnd}}R_{-3}, ..., \beta\alpha^{k/\mathtt{cwnd}}R_{-k}, ...\right)\right)\\
&= P\Big(R_{-1} < \frac{1}{\beta}R_0\Big)\cdots P\Big(R_{-k} < \frac{1}{\beta\alpha^{k\frac{1}{\mathtt{cwnd}}}}R_0\Big)\cdots\\
&= \int_T^\infty \rho p(r-T)\prod_{k=0}^{\infty}\Big((1-\rho)+\rho F\Big(\frac{r}{\beta\alpha^{k\frac{1}{\mathtt{cwnd}}}}-T\Big)\Big)dr.
\end{aligned}$$

Note that since spurious timeouts can only occur when the longer path is used, we assume that the round-trip time over the short path is always smaller than `mxrtt`. Thus $P\big(R_{-k} < \frac{1}{\beta a^k}R_0\big) = (1-\rho) + \rho F\big(\frac{r}{\beta\alpha^k} - T\big)$, where $F$ is the distribution of the time-varying part of the round-trip time for the longer path and $T$ is the time-invariant part of the round-trip time for the longer path. With a change of variables, the above becomes

$$\begin{aligned}
&P\left(\text{spurious timeout}\right)\\
&= \int_0^\infty \rho p\left(u\right) \prod_{k=0}^{\infty}\left(\left(1-\rho\right)+\rho F\left(\frac{u+T}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}}-T\right)\right)du\\
&= \int_0^\infty \rho p\left(u\right) \prod_{k=0}^{\infty}\left(\left(1-\rho\right)+\rho F\left(\frac{u+T\left(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\right)}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}}\right)\right)du.
\end{aligned}$$

While the above probability can be evaluated for various distributions $p$, it is when $T$ is large that presents the possibility of spurious timeouts. This corresponds to the case where the propagation and transmission delay are far larger on the longer path than on the shorter paths. As shown by the data analysis in the previous section, the variation in the round-trip time is of not much concern, hence cross-traffic and multiple hops are not directly responsible for spurious timeouts. Therefore, the case where $T$ is large is investigated. To this end, note that for $T$ large, $u+T\left(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\right)$ is either negative (when $\beta\alpha^{k\frac{1}{\texttt{cwnd}}}>1$) or large (when $\beta\alpha^{k\frac{1}{\texttt{cwnd}}}<1$). Thus, $F\left(\frac{u+T\left(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\right)}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}}\right)$ is either zero or one. Specifically, $F\left(\frac{u+T\left(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\right)}{\beta\alpha^{k\frac{1}{\texttt{cwnd}}}}\right)=1$ when $\beta\alpha^{k\frac{1}{\texttt{cwnd}}}<1$, i.e., when $k>-\log(\beta)/\log\left(\alpha^{\frac{1}{\texttt{cwnd}}}\right)$. Thus,

$$\prod_{k=0}^{\infty}\left(\left(1-\rho\right)+\rho F\left(\frac{u+T\left(1-\beta\alpha^{k\frac{1}{\texttt{cwnd}}}\right)}{\beta\alpha^{k\overline{\frac{1}{\texttt{cwnd}}}}}\right)\right)\approx\prod_{k=0}^{-\frac{\log(\beta)}{\log(\alpha)\frac{1}{\texttt{cwnd}}}}\left(1-\rho\right)=\left(1-\rho\right)^{-\frac{\log(\beta)}{\log(\alpha)\frac{1}{\texttt{cwnd}}}}.$$

Hence, when $T$ is large,

$$P\left(\text{spurious timeout}\right)\approx\rho\left(1-\rho\right)^{-\frac{\log(\beta)}{\log(\alpha)\frac{1}{\texttt{cwnd}}}}. \tag{4}$$

If the congestion window is not constant and $\alpha\approx1$, then the above approximation is

$$P\left(\text{spurious timeout}\right)\approx\rho\left(1-\rho\right)^{-\frac{\log(\beta)}{\log(\alpha)\overline{\frac{1}{\texttt{cwnd}}}}},$$

where $\overline{\frac{1}{\texttt{cwnd}}}$ is the average value of $\frac{1}{\texttt{cwnd}}$. In order to gain more insight, we continue to focus on the case where `cwnd` is constant. It is straightforward to find the $\rho$ that maximizes (4), yielding the worst-case probability of a spurious timeout. Figure 4 shows this worst-case probability as a function of $\alpha^{\frac{1}{\texttt{cwnd}}}$ and $\beta$.

When $\alpha\approx1$, the worst-case probability of a spurious timeout is approximately

$$\text{worst-case } P\left(\text{spurious timeout}\right)\approx\frac{-\log\left(\alpha\right)}{\log\left(\beta\right)}\frac{1}{\texttt{cwnd}}\frac{1}{e}.$$

Thus, in the worst-case, the typical number of packets between spurious timeouts is $\frac{e\log(\beta)\texttt{cwnd}}{\log(\alpha)}$. Since `cwnd` packets are sent every round-trip time, the typical time between spurious timeouts is

$$\text{Time between timeouts } = \frac{e\log\left(\beta\right)}{\log\left(\alpha\right)}\times\text{round-trip time.}$$

Thus, in the worst-case, if $\alpha=0.99$ and $\beta=3$, then there will be nearly 300 round-trip times between spurious timeouts. This number goes to nearly 3000 round-trip times if $\alpha=0.999$.

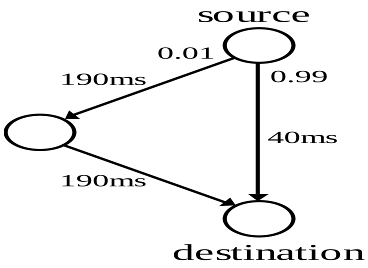

Figure 2: The Topology for Multi-path Routing. The path with shorter delay is taken by 99% of the packets, while the path with longer delay is taken by 1% of the packets.

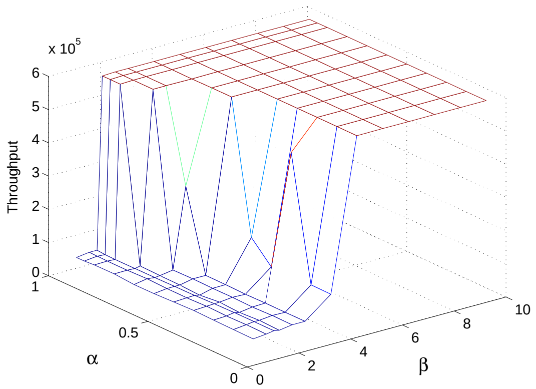

Figure 3: Throughput of TCP-PR with Multi-path routing.

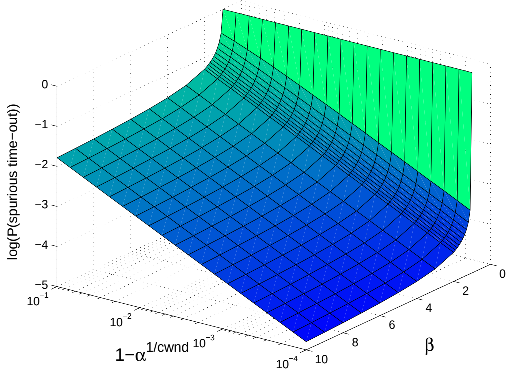

Figure 4: Worst Case Probability of Spurious timeout over Multi-path Routing.

# 5 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 6, 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 6 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 [1].

We have performed extensive ns-2 [13] 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 [14] over two topologies. The first topology is the dumbbell topology. The dumbbell topology was used in [15] to demonstrate the fairness of TCP-SACK and an implementation of the "TCP-friendly" formula. The second topology we use is the parking-lot topology. This topology includes cross traffic and is shown in Figure 5. The cross traffic used are long-lived TCP-SACK flows.

Following the approach taken in [15], 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 6 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 $\alpha$ and $\beta$ 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* 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 7 shows the coefficient of variation for ten simulations as well as the mean coefficient of variation for the simulation set. From Figures 6 and 7, we conclude that the mean and variance of the throughput for TCP-PR and TCP-SACK are similar.

Surprisingly, this fairness is maintained for a wide range of $\alpha$ and $\beta$. Figure 8 shows TCP-SACK's mean normalized throughput for different values of $\alpha$ and $\beta$. 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 $\beta$ 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. Furthermore, these days, it seems that loss rates of 15% are rare.

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 [16] and, in more detail, in [17] 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.

# 6 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 9. 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 $\varepsilon$ (see [18] for details). This parameter controls the

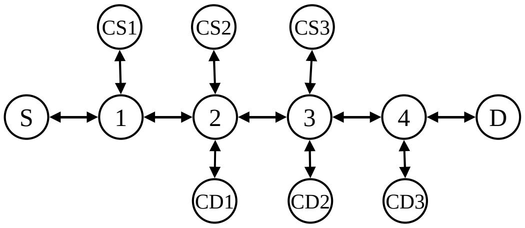

Figure 5: Parking-Lot Topology. This topology has multiple bottlenecks and cross traffic. The source and destination are labeled S and D respectively. The cross-traffic connections are CS1→CD1, CS1→CD2, CS1→CD3, CS2→CD2, CS2→CD3, and CS3→CD3. The data rates are: CS1→1 = 5Mbps, CS2→2=1.66Mbps, CS3→3=2.5Mbps, and all other links are 15Mbps. In this way the three links, 1→2, 2→3, and 3→4 are all bottleneck links.

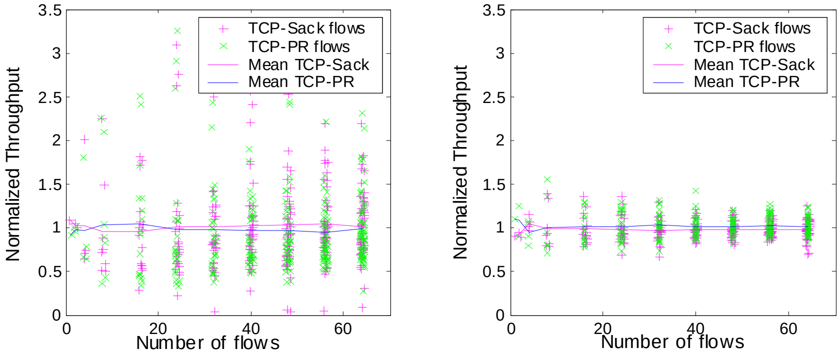

Figure 6: Fairness of TCP-PR Competing with TCP-SACK. The left plot shows the normalized throughput for the dumbbell topology, while the right plot shows the normalized throughput for the parking lot topology.

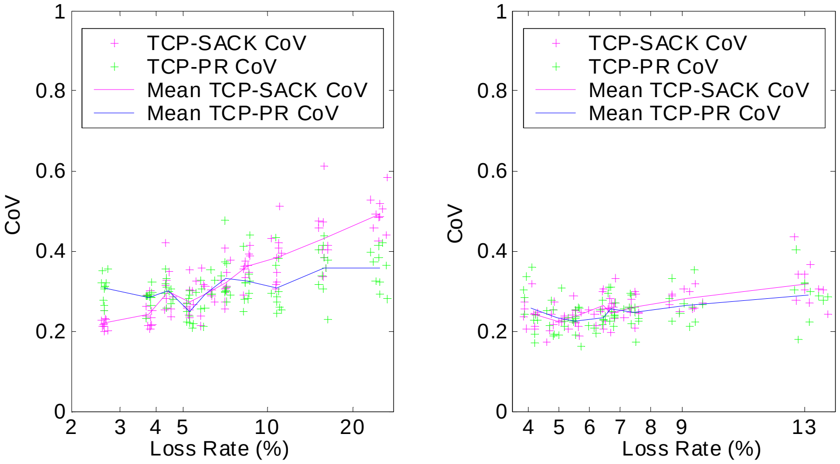

Figure 7: Coefficient of Variation. The coefficient of variation as a function of packet loss probability. The variation in loss probability was simulated by decreasing the link bandwidth. The left plot is the coefficient of variation for the dumbbell topology and the right plot is for the parking lot topology.

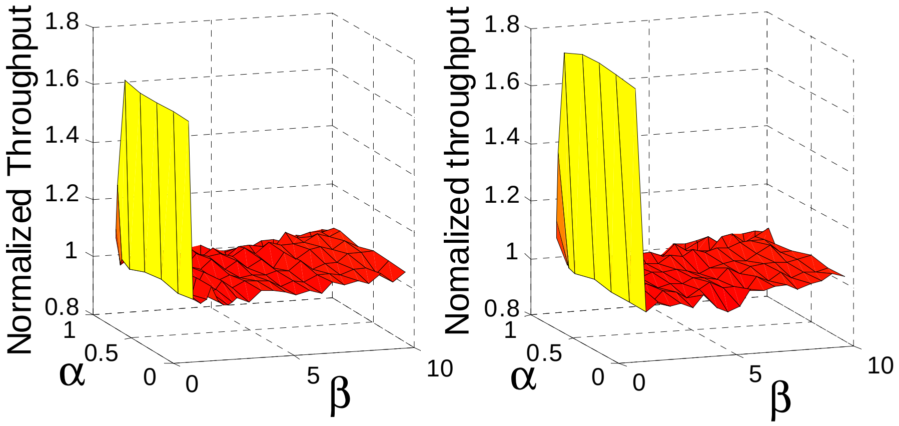

Figure 8: TCP-SACK Normalize Throughput for Different TCP-PR Parameters. The left plot shows the mean normalized throughput of TCP-SACK over the dumbbell topology, while the right plot shows the normalized throughput for the parking lot topology.

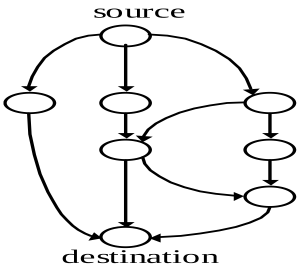

Figure 9: A Topology to Compare TCP Implemenations. Each link has a delay of 20ms, bandwidth of 10Mbps and queue has a size of 100 packets.

degree to which 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 $\varepsilon$ correspond to compromises between these two extreme cases.

For a fixed routing strategy (a fixed $\varepsilon$), TCP-PR and TCP with various *dupthresh* compensation schemes discussed in [1] 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 10 shows the throughput for various values of $\varepsilon$. 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 $\varepsilon$ if the propagation delay is small (the left plot in Figure 10). 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 [1], 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 [9]. Since the simulation implementation of this method is not yet available, it was not included in this comparison.

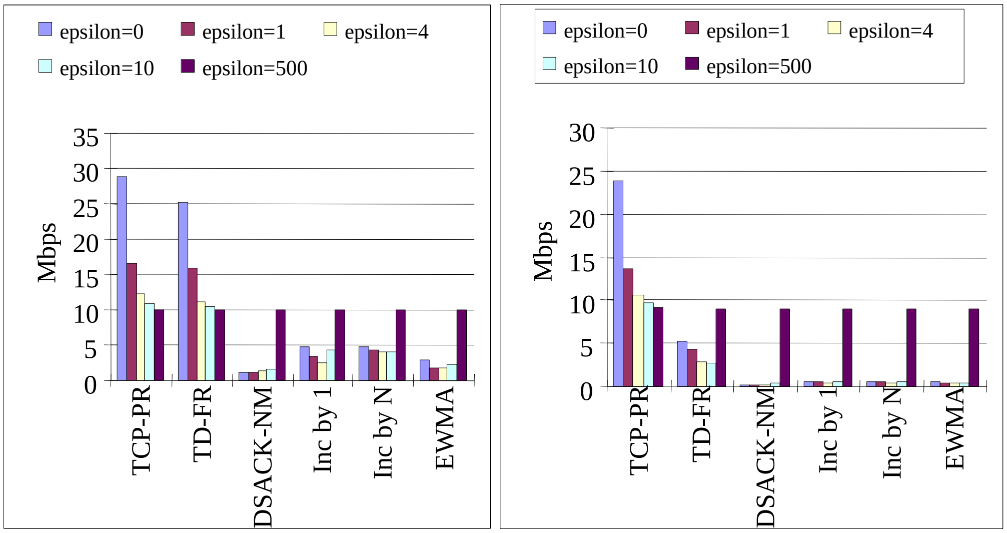

Figure 10: The throughput for different TCP implementations and different degrees of multi-path routing. Single path routing corresponds to the case when $\varepsilon = 500$. For $\varepsilon$ smaller, alternative paths are used more frequently. In the limit, $\varepsilon = 0$, all paths are used with equal probability. The plot on the right corresponds to the topology in Figure 9 with each link propagation delay of 10ms. The plot on the right is the same, but the propagation of each link is 60ms.

# 7 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 [6]); 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.

# References

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

[2] V. Paxson, “End-to-end routing behavior in the internet,” in *ACM SIGCOMM*, 1996.

[3] F. Wang and Y. Zhang, “Improving TCP performance over mobile ad-hoc networks with out-of-order detection and response,” in *ACM MOBIHOC*, 2002.

[4] T. Dyer and R. Boppana, “A comparison of TCP performance over three routing protocols for mobile ad hoc networks,” in *ACM MOBIHOC*, 2001.

[5] G. Holland and N. Vaidya, “Analysis of TCP performance over mobile ad-hoc networks,” in *ACM MOBICOM*, 1999.

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

[7] R. Ludwig and R. Katz, “The Eifel algorithm: Making TCP robust against spurious retransmissions,” *ACM Computer Communication Review*, vol. 30, no. 1, 2000.

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

[9] N. Zhang, B. Karp, S. Floyd, and L. Peterson, “RR-TCP: A reordering-robust TCP with DSACK,” Tech. Rep. TR-02-006, ICSI, Berkeley, CA, July 2002.

[10] V. Paxson, “End-to-end internet packet dynamics,” in *ACM SIGCOMM*, 1997.

[11] M. Allman and V. Paxson, “Computing TCP’s retransmission timer,” *RFC 2988*, p. 13, Nov. 2000.

[12] N. L. for Applied Network Research (NLANR). http://www.nlanr.net/.

[13] 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 http://www.isi.edu/nsnam/ns/ns-documentation.html.

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

[15] S. Floyd, M. Handley, J. Padhye, and J. Widmer, “Equation-based congestion control for unicast applications,” in *SIGCOMM 2000*, (Stockholm, Sweden), 2000.

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

[17] 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.

[18] J. Hespanha and S. Bohacek, “Preliminary results in routing games,” in *American Control Conference*, (Arlington, VA), IEEE, June, 2001.

```
Event: initialization
    mode := slow-start
    cwnd := 1
    ssthr := $+\infty$
    memorize := $\emptyset$

Event: time > time(n) + mxrtt
(drop detected for packet n)
    remove(to-be-ack, n)
    add(to-be-sent, n)
    if not is-in(memorize, n) then /* new drop */
        memorize := to-be-ack
        cwnd := cwnd(n)/2
        ssthr := cwnd
    else /* other drops in burst */
        remove(memorize, n)
    flush-cwnd()

Event: ack received for packet n
    srtt = $\max\{\alpha^{\frac{1}{cwnd}} \times srtt, time - time(n)\}$
    mxrtt := $\beta \times$ srtt
    remove(to-be-ack, n)
    remove(memorize, n)
    if mode = slow-start and cwnd + 1 $\leq$ ssthr then
        cwnd := cwnd + 1
    elseif then
        mode := congestion-avoidance
        cwnd := cwnd + 1/cwnd
    flush-cwnd()

Event: flush-cwnd()
    while cwnd > |to-be-ack| do
        k=send(to-be-sent)
        remove(to-be-sent, k)
        add(to-be-ack, k)
```

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

| | |
|---|---|
| `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) |
| \|`list`\| | 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 Tables 1 and 3

```
Event                                   Code

initialization                          1    mode := slow-start
                                        2    cwnd := 1
                                        3    ssthr := $+\infty$
                                        4    memorize := $\emptyset$
                                        4a   cburst = 0
                                        4b   slowdown = time

time > time(n) + mxrtt                  5    remove(to-be-ack, n)
(drop detected for packet n)            6    add(to-be-sent, n)
                                        7    if not is-in(memorize, n) then /* new drop */
                                        8        memorize := to-be-ack
                                        8a       if cwnd(n) > 1 then
                                        9            cwnd := cwnd(n)/2
                                        9a       else
                                        9b           mxrtt := 2 × mxrtt
                                        10       ssthr := cwnd
                                        11   else /* other drops in burst */
                                        12       remove(memorize, n)
                                        12a      cburst := (memorize = $\emptyset$) ? 0 : cburst + 1
                                        12b      if cburst > cwnd/2 + 1 then
                                        12c          cwnd := 1
                                        12d          mode := slow-start
                                        12e          mxrtt := max{mxrtt, 1sec}
                                        12f          slowdown := time + mxrtt
                                        13   flush-cwnd()

ack received for packet n               14   srtt = max $\left\{\alpha^{\frac{1}{\mathtt{cwnd}}} \times \mathtt{srtt}, \mathtt{time} - \mathtt{time}(n)\right\}$
                                        15   mxrtt := $\beta$ × srtt
                                        16   remove(to-be-ack, n)
                                        17   remove(memorize, n)
                                        17a  if memorize = $\emptyset$ then
                                        17b      cburst := 0
                                        18       if mode = slow-start and cwnd + 1 ≤ ssthr then
                                        19           cwnd := cwnd + 1
                                        20       elseif then
                                        21           mode := congestion-avoidance
                                        22           cwnd := cwnd + 1/cwnd
                                        23   flush-cwnd()

flush-cwnd()                            24a  while slowdown ≤ time and cwnd > |to-be-ack| do
                                        25       k=send(to-be-sent)
                                        26       remove(to-be-sent, k)
                                        27       add(to-be-ack, k)
```

Table 3: Pseudo-code for TCP-PR with extreme losses (cf. notation used in Table 2).