A New TCP for Persistent Packet Reordering

Stephan Bohacek, João P. Hespanha, Junsoo Lee, Chansook Lim, and Katia Obraczka

***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 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. 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. Furthermore, TCP-PR only requires changes to the TCP sender side making it easier to deploy.

I. 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 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. Specifically, 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." This is the case for reordering of both data and acknowledgment packets. Indeed, it is well known that TCP performs poorly under significant packet reordering (which may not be necessarily caused by packet losses) [1].

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 for out-of-order packets observed in the Internet today [2]. Internet experiments performed through MAE-East and reported in [3] show that 90% of all connections tested experience packet reordering. Researchers at SLAC performed similar experiments and found that 25% of the connections monitored reordered packets [4]. 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 source, sink, as well as forwarder of traffic. 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 [5]–[7].

Mechanisms that provide different quality-of-service (QoS) by differentiating traffic may introduce packet reordering. An example of such mechanisms is DiffServ [8], which has been proposed to provide different QoS on the Internet. In the case of Expedited Forwarding, packets receive preferential treatment as long as the flow obeys negotiated bandwidth constraints. If the flow exceeds these constraints, the non-conformant packets are typically dropped. However, an al-

This work has been partially supported by the National Science Foundation under Grant Nos. ANI-0322476 and CCR-0311084.

Stephan Bohacek is with the Department of Electrical & Computer Engineering, University of Delaware, Newark, DE 19716 USA (e-mail: bohacek@udel.edu)

João P. Hespanha is with Department of Electrical & Computer Engineering, Univ. of California, Santa Barbara, CA 93106 USA (e-mail: hespanha@ece.ucsb.edu)

Junsoo Lee is with the Department of Computer Science, Sookmyung Women's Univ., Seoul, Korea 140-742 (email: jslee@sookmyung.ac.kr

Chansook Lim is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089 USA (e-mail: chansool@usc.edu)

Katia Obraczka is with the Department of Computer Engineering, University of California, Santa Cruz, CA 95064 USA (e-mail: katia@cse.ucsc.edu)

ternative to dropping these packets is to lower their priority. In this case, the packets will be placed in different queues and will likely experience different latency, resulting in out-of-order delivery to the final destination. While this alternative is atypical, RFC 3246 simply specifies that packets "should not" be reordered, a weaker requirement than "must not" be reordered.

While packet reordering is often considered to be pathological in today's Internet, as shown in [3], it is actually part of normal operation for a number of routers containing parallel paths through the switch. Due to the scheduling algorithms used, different packet sizes and arrivals times may result in the reordering of packets entering on a single interface. While the exact cause of packet reordering lies in the details of the scheduling algorithm, a more general reason is that parallel paths are employed for economic reasons; it is cheaper to build multiple moderate speed paths than a single very high-speed path. The result of seeking this increase in cost efficiency is that packets may sometimes be reordered. TCP-PR is a transport protocol compatible with multipath routing, hence it will not limit the drive for efficiency at the lower layers.

Beyond router design, there are other areas that stand to gain efficiency if multiple paths are permitted. For example, load balancing is greatly simplified if single flows are permitted to use different paths. When a flow is restricted to use a single path, then optimal load balancing reduces to an NP-hard integer programming problem (cf. [9, p. 359]) but if the single path restriction is lifted, then optimal load balancing is a simpler linear programming problem. In [10], different flows may be split along multiple paths in order to meet QoS requirements. Permitting even single flows to be split results in a more efficient use of network resources. In the case of MANETs, spreading packets across different links also decreases the battery drain on any particular mobile node and may increase the lifetime of the network.

While efficiency is one area that may benefit from multipath routing, fault tolerance and security can also be improved by utilizing multiple paths. For example, in wired networks, multipath routing has been shown to reduce the impact of link failures [11]. Similarly, multipath routing can increase robustness to eavesdropping by spreading packets across different paths, thus forcing the attacker to sniff multiple links [11]. Multipath routing can take advantage of the considerable path redundancy that already exists in today's Internet. For example, in [12], it was shown that in the US Sprintlink topology, 90% of PoP pairs are connected through at least 4 distinct paths.

In MANETs, alternate path routing has been an active area of research. In [13], it is suggested that alternate paths be found and stored in an attempt to anticipate failures in the primary path. However, it was shown in [14] that alternate paths may grow stale and no longer exist when the primary path fails. One way to learn that alternate paths have failed is to send part of the data stream along them, as in multipath routing.

While multipath routing has many advantages, it leads to persistent packet reordering. Today's implementations of TCP are not compatible with networks that reorder packets and suffer great reductions in throughput when faced with persistent packet reordering. TCP's incompatibility with persistent packet reordering has been a major deterrent to the deployment of the mechanisms mentioned above on the Internet or on other networks in which TCP is prevalent. There are a number of methods for improving TCP's performance in packet-reordering prone environments, but most of them try to recover from occasional reordering and rely on the packet ordering itself to detect drops. However, under persistent reordering conditions, packet ordering conveys very little information on what is actually happening inside the network.

In this paper, we describe TCP-PR, a transport protocol that performs well under persistent packet reordering (Section III). The key feature of TCP-PR is that duplicate ACKs are not used as an indication of packet loss. Rather, TCP-PR relies exclusively on timeout. Both worst-case analysis and Internet traces are used to ensure that the timeout threshold is not too small and only actual packet losses cause retransmissions (Section IV). Through extensive ns-2 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 (Section VI). We find that under persistent packet reordering, TCP-PR achieves significantly higher throughput. We also test TCP-PR's compatibility and fairness to standard TCP variants, specifically TCP-SACK (Section V). In the absence of packet reordering, TCP-PR is shown to have similar performance and competes fairly with TCP-SACK. TCP-PR neither requires changes to the TCP receiver nor uses any special TCP header option. Hence, TCP-PR is suitable for incremental deployment.

II. 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 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 timestamp every packet sent. The receiver echoes back the timestamp 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[1].

DSACK [16] 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

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

spurious retransmission detected through DSACK[2]. Besides recovering the congestion state prior to the spurious retransmission, 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 the 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 [17].

Time-delayed fast-recovery (TD-FR), which was first proposed in [18] and analyzed in [1], addresses 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 set to $\max(RTT/2, DT)$, where DT is the difference between the arrival of the first and third DUPACK.

A number of mechanisms to improve TCP's performance in MANET environments have been proposed. For example, TCP-DOOR [5] 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. Upon detecting out-of-order packets (internally or informed by the receiver[3]), the TCP sender responds by either: (1) temporarily disabling congestion control for a given time interval (i.e., freezing the congestion control state, which includes the congestion window `cwnd` and the retransmission timer RTO), or (2) resetting the state to its value prior entering congestion avoidance. More recently, TCP-DCR [19], another variant of TCP for wireless networks, has been developed. Similarly to TD-FR, TCP-DCR delays response to DUPACKs. However, the delay of one RTT imposed by TCP-DCR is longer than that of TD-FR.

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 of congestion. However, if packets are persistently reordered, packet ordering conveys little information regarding congestion and thus should not be used to trigger congestion control. Consequently, as shown in Section VI, TD-FR and methods that use DSACK along with adjusting $dupthresh$ perform poorly when faced with persistent packet reordering.

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. This is discussed in Section IV. The second challenge, covered in Section V, is to maintain fairness with current implementations of TCP. Section VI presents extensive simulation results that show that under persistent packet reordering, TCP-PR performs significantly better than existing packet reordering recovery methods.

[2]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.

[3]As suggested in [5], the TCP receiver can notify the sender by setting a *OOO* bit in the TCP ACK packet

III. 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 as the algorithm does not distinguish between data- (on the forward path) or acknowledgment (on the reverse path) losses.

The proposed algorithms only require changes in the TCP sender and are 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 ensure they can coexist.

A. 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 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, a timestamp is saved. 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 `mxrtt`.

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

$$\texttt{mxrtt} := \beta \times \texttt{srtt}, \tag{1}$$

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

$$\texttt{srtt} = \max\left\{\alpha^{\frac{1}{\lfloor \texttt{cwnd} \rfloor}} \times \texttt{srtt}, \texttt{sample} - \texttt{rtt}\right\}, \tag{2}$$

where α denotes a positive constant smaller than 1, $\lfloor \texttt{cwnd} \rfloor$ the floor of the current congestion window size, and $\texttt{sample} - \texttt{rtt}$ the RTT for the packet whose acknowledgment just arrived [4]. The reason to raise α to the power $1/\lfloor \texttt{cwnd} \rfloor$ is that in one RTT the formula in (2) is iterated $\lfloor \texttt{cwnd} \rfloor$ times. This means that, e.g., if there were a sudden decrease in the RTT then srtt would decrease by a rate of $(\alpha^{\frac{1}{\lfloor \texttt{cwnd} \rfloor}})^{\lfloor \texttt{cwnd} \rfloor} = \alpha$ per RTT, independently of the current value of the congestion window. The parameter α can therefore be interpreted as a smoothing factor in units of RTTs. As discussed in Section IV, 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.

Figure **??** shows how srtt and mxrtt "track" RTT. Note that srtt tracks the peaks of RTT. The rate that srtt decays after a peak is controlled by α. The right-hand plot shows how large jumps can cause $RTT >$ mxrtt (for this data set, occurrences at 15s, 45s, 75s, etc.) resulting in spurious timeouts (note that the jumps in RTT in the right-hand plot were artificially generated). In order for these jumps to cause a spurious timeouts, the jumps in RTT could occur no sooner than every 15 seconds. In this case, 1500 packets were delivered between these jumps. If the jumps occurred more frequently, then, as can be seen from the figure, mxrtt would not have decayed to a small enough value and spurious timeouts would not occur. Furthermore, if the jumps were larger, then the time between jumps to cause a timeout would be no smaller. The issue of spurious timeouts is closely examined in Section IV.

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 III-B). In $slow - start$, cwnd starts at 1 and increases exponentially (increases by one for each ACK received). Once the first loss is detected, cwnd is halved and the sender transitions to $congestion - avoidance$, where cwnd increases linearly (1/cwnd for each ACK received). Subsequent drops cause further halving of cwnd, without the sender ever leaving $congestion - avoidance$. An important but subtle point in halving cwnd is that when a packet is sent, not only a timestamp but the current value of cwnd is saved in the $\texttt{to} - \texttt{be} - \texttt{ack}$ 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 $\texttt{to} - \texttt{be} - \texttt{sent}$ 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.

The pseudo-code in Table I corresponds to the algorithm just described. Table II 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 are pending, but is not significantly more demanding than TCP-SACK. It does maintain the extra memorize list used to detect drop bursts, but this list is empty most of the time and otherwise only needs to contain pointers to packets also in the $\texttt{to} - \texttt{be} - \texttt{ack}$ list. Recall that the transport layer must maintain the data to be transmitted until the packet has been ACKed by a cumulative ACK. If static memory allocation is used, the transport layer must allocate enough memory to hold a maximum sender's window's worth of packets. In the typical case where the MSS is several hundred to over a thousand bytes, TCP-PR's requirement of two bytes per packet for timestamps and lists of pointers results in a relatively minor increase in the transport layer's memory requirements. On the other hand, $\alpha^{\frac{1}{\lfloor \texttt{cwnd} \rfloor}}$ must be computed every time $\lfloor \texttt{cwnd} \rfloor$ is incremented and requires a number of multiplications, divisions and additions. Alternatively, $\alpha^{\frac{1}{\lfloor \texttt{cwnd} \rfloor}}$ can be tabulated requiring table look-up operations. Clearly, table look-ups are less computationally intensive, but require more memory (again, this memory demand is small compared to what is required for packet buffering).

B. Extreme Losses

When more than half of a window's worth of packets is dropped, TCP-NewReno/SACK may timeout in the fast-recovery mode [20]. This is because not enough ACKs are received for the congestion window to open and allow for the sender to perform the needed retransmissions. The occurrence of a timeout typically depends on the number of packets dropped, the congestion window size cwnd, the round-trip

[4] We currently have an implementation of TCP-PR in the Linux kernel. In order to compute $\alpha^{\frac{1}{\lfloor \texttt{cwnd} \rfloor}}$ in the kernel, we employ Newton's method through the following loop:

1 $x :=$ result from last calculation or 1 if there has not been a previous calculation of $\alpha^{\frac{1}{\lfloor \texttt{cwnd} \rfloor}}$
2 while $\left| x - \left(\frac{\lfloor \texttt{cwnd} \rfloor - 1}{\lfloor \texttt{cwnd} \rfloor} x + \frac{\alpha}{\lfloor \texttt{cwnd} \rfloor \, x^{\lfloor \texttt{cwnd} \rfloor - 1}} \right) \right| > (1-\alpha)10^{-3}$
3 $\quad x := \frac{\lfloor \texttt{cwnd} \rfloor - 1}{\lfloor \texttt{cwnd} \rfloor} x + \frac{\alpha}{\lfloor \texttt{cwnd} \rfloor \, x^{\lfloor \texttt{cwnd} \rfloor - 1}}$
4 end

The larger the value of n, the better the approximation. In our implementation, we are using $n = 10$. However, if cwnd is bounded by a small enough value, it might be simpler to save $\alpha^{\frac{1}{\lfloor \texttt{cwnd} \rfloor}}$ in a look-up table

TABLE I: Pseudo-code for TCP-PR (cf. notation in Table II)

Event		Code
initialization	1	`mode` := *slow-start*
	2	`cwnd` := 1
	3	`ssthr` := $+\infty$
	4	`memorize` := $\emptyset$
`time` > `time`(n) + `mxrtt` (drop detected for packet n)	5	remove(`to-be-ack`, n)
	6	add(`to-be-sent`, n)
	7	if not is-in(`memorize`, n) then /* new drop */
	8	`memorize` := `to-be-ack`
	9	`cwnd` := `cwnd`(n)/2
	10	`ssthr` := `cwnd`
	11	else /* other drops in burst */
	12	remove(`memorize`, n)
	13	`flush-cwnd()`
ack received for packet n	14	`srtt` = $\max\{\alpha^{\frac{1}{\texttt{cwnd}}} \times \texttt{srtt}, \texttt{time} - \texttt{time}(n)\}$
	15	`mxrtt` := $\beta \times$ `srtt`
	16	remove(`to-be-ack`, n)
	17	remove(`memorize`, n)
	18	if `mode` = *slow-start* and `cwnd` + 1 $\leq$ `ssthr` then
	19	`cwnd` := `cwnd` + 1
	20	else
	21	`mode` := *congestion-avoidance*
	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)
	28	`time`(k) = `time`

TABLE II: Notation used in Tables I and III

`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

time, and the value of RTO. In the ns-2 simulations whose results we report, we observed timeouts in TCP SACK when more than `cwnd`/2+1 packets are dropped within a window, which is consistent with the results in [20]. TCP-NewReno and SACK also enter the timeout mode when the retransmitted packets are lost or when drops occur while the congestion window size is smaller than 4. In the latter case, there are not enough ACKs to trigger fast recovery, so a timeout eventually occurs.

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, we have found that without special attention to the behavior during extreme losses, TCP-PR is unfair to today's implementations of TCP. In order to maintain fairness (a key goal of this work), we propose a version of TCP-PR that mimics TCP-SACK's timeout, i.e., upon detecting an extreme loss situation, TCP-PR sets SSTHRESH to `cwnd`/2, resets `cwnd` to one, performs exponential back-off, etc. It should be emphasized that this variation of TCP-PR is not required for the proper functioning of TCP-PR in sense that throughput is maintained without this enhancement. Rather, this variation results in lower, but more fair, throughput in high loss situations.

TCP-PR detects 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 is 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 during a TCP-NewReno or TCP-SACK timeout, we check if `cburst` (and therefore the number of drops in a burst) exceeds `cwnd`/2 + 1, or if a drop is detected while `cwnd` < 4, or if a retransmitted packet is dropped. When one of these conditions occurs, we reset `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` [21]. If further (new) drops occur while `cwnd` = 1, instead

of dividing cwnd by two we double mxrtt, thus emulating the usual exponential back-off. The pseudo-code in Table III implements this algorithm. In this pseudo-code, we also inhibited increments of cwnd while the memorize list 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 the fast-recovery mode.

IV. Selection of TCP-PR Parameters

In this section we discuss the selection of the parameters α and β used in the estimation of the maximum round-trip time mxrtt. When the time elapsed since a packet was sent exceeds mxrtt 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 mxrtt is set too low, 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, throughput may suffer severely if they occur too frequently.

In order to determine adequate values for α and β that reduce the occurrence of spurious timeouts, we employ two different methods: (1) an analytical approach that determines a worst-case relationship between the rate of spurious timeouts and the values of the parameters, and (2) an empirical method that utilizes RTT traces to determine the probability of spurious timeouts in today's Internet.

A. Worst-case Analysis of Spurious Timeouts

A spurious timeout occurs when the estimate mxrtt of the maximum round-trip time is actually smaller than RTT. Since mxrtt adapts online to the current RTT, this can occur when RTT takes small values for a period of time and suddenly increases. We consider a worst-case situation where packets exhibit one of two possible RTTs: a small value $RTT_{\min}$ and a large one $RTT_{\max}$. This would occur, e.g., if multipath routing were employed in the network depicted in Figure 1. In this network periodic spurious timeouts can occur if several

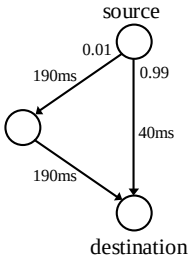

Fig. 1: Worst-case topology for spurious timeouts. When the packet latency along the longer path is at least β times longer than that of the shorter path and the packets take the shorter path infrequently enough, a spurious timeout will occur every time a packet follows the longer path.

packets take the shorter path until srtt essentially becomes equal to $RTT_{\min}$ and then a packet takes the longer path. A timeout will then occur if

$$\beta \times \texttt{mxrtt} \approx \beta \times RTT_{\min} < RTT_{\max}. \tag{3}$$

Suppose that this is indeed the case. One can then ask under what conditions a spurious timeout can occur again. To answer this question note that, right after a packet goes through the longer path, srtt jumps to $RTT_{\max}$. In the worst-case it will be followed by several packets taking the shorter path as otherwise mxrtt will not decrease and further spurious timeouts will not occur. As discussed in Section III, after K congestion windows worth of packets have been ACKed, $\texttt{srtt} = \alpha^K \times RTT_{\max}$. Therefore, a spurious timeout will be produced by a packet taking the long route $K \times RTT_{\min}$ seconds after the previous one, as long as

$$\beta \times \texttt{srtt} = \beta \times \alpha^K \times RTT_{\max} < RTT_{\max}.$$

We conclude from here that the minimum number of shorter round-trip times between spurious timeouts is equal to $K = \frac{\log \beta}{-\log \alpha}$. In this worst-case situation, most packets take the shorter path and therefore the average RTT is essentially $RTT_{\min}$. We can summarize our conclusions as follows. When (3) does not hold there will be no spurious timeouts, otherwise the time between spurious timeouts must be larger than

$$\frac{\log \beta}{-\log \alpha} \times \overline{RTT},$$

where $\overline{RTT}$ is the average round-trip time. This means that $\beta = 3$ and $\alpha = 0.999$ result in low rates for spurious timeouts (approximately $1000 \times \overline{RTT}$ seconds between spurious timeouts) and yield good performance, as confirmed in the following sections.

Remark 2: This worst-case analysis also applies to conditions such as highly variable processing delay within a router, or delay variations due to link layer ARQ or media access in wireless channels which may also cause spurious timeouts. Similarly to the scenario investigated above, the worst case situation is when the latency mostly takes a small value and occasionally jumps to a large value.

B. Analysis of RTT Traces

The above analysis determined the worst-case frequency of spurious timeouts. RTT traces can be utilized to estimate the probability of getting a spurious timeout in the more typical case of today's Internet. Using a July 25, 2001 snapshot of round-trip times from the NLANR data set [22], we computed empirical 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 (one sample per 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 α and β. Figure 2 shows the probability of a drop versus the parameters α and β.

TABLE III: Pseudo-code for TCP-PR with extreme losses (cf. notation in Table II).

Event		Code
initialization	1	mode := *slow-start*
	2	cwnd := 1
	3	ssthr := $+\infty$
	4	memorize := $\emptyset$
	5	waituntil := $+\infty$
	6	to-be-ack := $\emptyset$
	7	cburst = 0
time $>$ time(n) + mxrtt (drop detected for packet n)	8	remove(to-be-ack, n)
	9	add(to-be-sent, n)
	10	if mode $\neq$ *waiting* then
	11	if not is-in(memorize, n) then /* new drop */
	12	memorize := to-be-ack
	13	cburst := 1
	14	if cwnd$(n) > 1$ then
	15	cwnd := cwnd$(n)/2$
	16	ssthr := cwnd
	17	else /* other drops in burst */
	18	remove(memorize, n)
	19	cburst := cburst + 1
	20	if cburst $>$ cwnd + 1 or cwnd < 2 or
	21	packet n is a retransmitted packet then
	22	cwnd := 1
	23	mxrtt := $\max\{2 \times$ mxrtt, $1sec\}$
	24	waituntil := time + mxrtt
	25	mode := *waiting*
ack received for packet n	26	remove(to-be-ack, n)
	27	remove(memorize, n)
	28	if mode $\neq$ *waiting* then
	29	srtt $= \max\left\{\alpha^{\frac{1}{\text{cwnd}}} \times \text{srtt}, \text{time} - \text{time}(n)\right\}$
	30	mxrtt := $\beta \times$ srtt
	31	if memorize $= \emptyset$ then
	32	cburst := 0
	33	if mode = *slow-start* and cwnd + 1 $\leq$ ssthr then
	34	cwnd := cwnd + 1
	35	else
	36	mode := *congestion-avoidance*
	37	cwnd := cwnd $+ 1/$cwnd
	38	flush-cwnd()
flush-cwnd()	39	if mode $\neq$ *waiting* then
	40	while cwnd $>$ |to-be-ack| do
	41	k=send(to-be-sent)
	42	remove(to-be-sent, k)
	43	add(to-be-ack, k)
	44	time(k) = time
time $>$ waituntil	45	put contents of memorize into to-be-sent
	46	memorize := $\emptyset$
	47	to-be-ack := $\emptyset$
	48	mode := *slow-start*
	49	waituntil := $+\infty$
	50	flush-cwnd()

For these computations, it was assumed that cwnd $= 1$. However, for cwnd > 1 we have that $\alpha < \alpha^{1/\text{cwnd}}$ and therefore

$$\begin{aligned}\text{mxrtt}_{k+1} &= \beta \times \text{srtt}_{k+1} \\ &= \beta \times \max\left\{\alpha^{1/\text{cwnd}} \times \text{srtt}_k, \text{sampleRTT}\right\} \\ &\geq \beta \times \max\left\{\alpha^{1} \times \text{srtt}_k, \text{sampleRTT}\right\}.\end{aligned}$$

Hence, assuming cwnd $= 1$ actually reduces mxrtt, leading to an overestimate on the number of spurious timeouts. The data used was also collected at one minute intervals, whereas TCP-PR would likely sample RTT much more frequently. Since RTT is positively correlated [23], large jumps are observed less frequently when RTT is sampled at closely spaced intervals. The more frequent sampling that occurs within TCP-PR would likely drive down the probability of spurious timeouts even further.

Despite these two conservative assumptions, Figure 2 shows that as long as $\beta > 1$ and $\alpha > 0.99$ the probability of a spurious timeout occurring is less than 10^{-7}. For $\alpha > 0.999$ and $\beta \geq 2$, the probability of a spurious timeout becomes vanishingly small.

Standard implementations of TCP [24] compute RTO which is a filtered version of RTT that is used to trigger TCP's timeout. Typically, RTO is computed using Van Jacobson's

algorithm [25]

$$srtt_{k+1} = \frac{7}{8} srtt_k + \frac{1}{8} RTT_k$$
$$DevRTT_{k+1} = \frac{3}{4} DevRTT_k + \frac{1}{4} |srtt_k - RTT_k|$$
$$RTO_k = \max\left(srtt_k + K \times DevRTT_k, MinRTO\right),$$

While the typical value of K is 4, there is some discrepancy over the value of $MinRTO$. Often, it is assumed that $MinRTO = 1$ second (as specified in RFC 2988), but some implementations use different values, e.g., BSD and MS Windows uses 500ms, and Linux uses 200ms. Figure 2 shows the empirical probability of $RTT > RTO$ from the NLARN data set under the assumption that $MinRTO = 0$. For $K = 4$, we obtained $P\left(RTT > RTO\right) \approx 0.0124$, i.e., a little more than 1% of all packets would timeout. It is difficult to determine the best value of $MinRTO$. If $RTT > MinRTO$, then we can expect 1% of packets sent will trigger a spurious timeout. For RTO too large, the throughput of TCP-PR would be reduced. While it is possible that an intermediate value of MinRTO would result in less than 1% spurious timeouts while maintaining high throughput, it is difficult to find a single MinRTO that works well in many situations. A similar conclusion was also reached in [26] where no obvious "sweet spots" were found for $MinRTO$ when used in TCP.

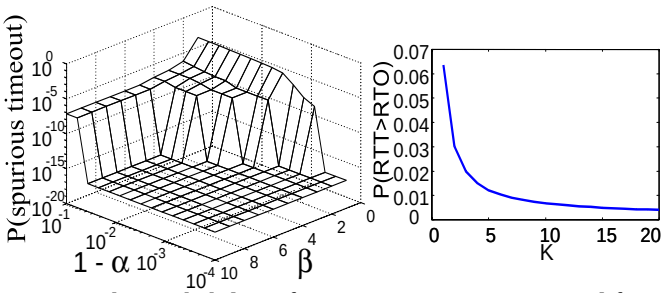

Fig. 2: Right: Probability of spurious timeouts computed from the NLANR data set. For many pairs of α, β there were no spurious timeouts. However, in order to view the data on a log scale a perturbation of 10^{-16} has added. Hence, all the pairs α, β that show a probability of 10^{-16} actually had no spurious timeouts at all. Left: probability of spurious timeouts when Van Jacobson's Algorithm is used.

V. Performance and Fairness without Packet Reordering

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 VI, we also show that, when no packet reordering occurs, TCP-PR achieves the same throughput as other TCP implementations.

We performed extensive ns-2 [27] simulations to show that TCP-PR is fair with respect to standard TCP implementations, for a wide range of network conditions and topologies. In this section, a sample of our simulation results is presented, with attention focused on the compatibility with TCP-SACK [28]. One of the topologies we use is the dumbbell topology, also known as *single-bottleneck*. 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 fairness) of TCP-SACK and an implementation of the "TCP-friendly" formula [29]. The other topology used is the *parking-lot* topology, which includes multiple bottleneck links and has also been employed in a number of recent performance studies of network protocols including [30] and [31]. Figure 3 shows the parking-lot and dumbbell topologies used, including the source and destination nodes for the cross traffic. Previous studies that used the parking-lot topology only included cross traffic between nodes CS1→CD1, CS2→CD2, and CS3→CD3. In our simulations, we also considered cross traffic between CS1→CD2, CS1→CD3, and CS2→CD3. For the single-bottleneck (dumbbell) topology, we ran simulations both with and without HTTP background traffic. When background traffic was used, it corresponded to around 10% of the total traffic [5]. HTTP traffic flowed from node 1 to node 2; more specifically, we set up five Web server-client pairs, each of which running ten concurrent connections. All Web clients ran in node 2 and all Web servers in node 1 so that the HTTP traffic direction coincides with the TCP flows under study. For the parking-lot topology, we always considered HTTP cross traffic. Eight Web server-client pairs were configured each of which with ten concurrent active sessions. While CS1→CD1 and CS3→CD3 have 2 pairs of Web server-client each, all other pairs have a single Web server-client pair. In both topologies, each Web session used the following parameters: inter-page time was exponentially distributed with mean 1 second, the number of objects per page was uniformly distributed with mean 1, the inter-object time was exponentially distributed with mean 10 ms, and object size was Pareto-distributed with mean 40KB and shape parameter equal to 1.2 [6]. A large number of distinct link speeds and number of flows were investigated.

Following the approach taken in [29], the fairness of TCP-

[5]Generating HTTP background traffic to correspond to 10% of the total traffic was motivated by the observation that, "while the exact fraction of short-lived traffic found on the Internet is unknown, it appears that short-lived flows make up for at least 10% of the total Internet traffic" [32].

[6]These are the same parameters used by sample ns-2 scripts included in the ns-2 distribution; furthermore, these same parameters have also been used in simultaions conducted by other researchers (e.g., [33].)

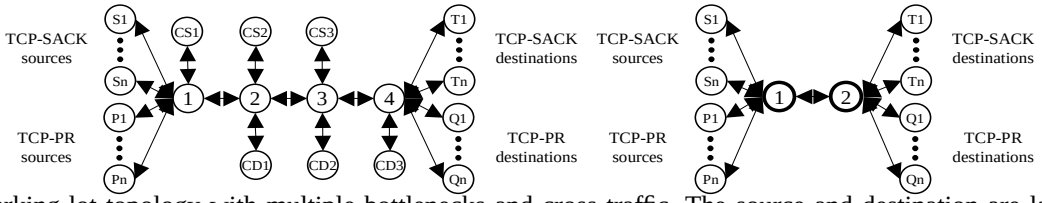

Fig. 3: Left: Parking-lot topology with multiple bottlenecks and cross traffic. The source and destination are labeled S and D respectively. The cross-traffic connections are CS1→CD1, CS1→CD2, CS1→CD3, CS2→CD2, CS2→CD3, and CS3→CD3. The data rates are: 5Mbps for CS1→1, 1.66Mbps for CS2→2, 2.5Mbps for CS3→3, and 15Mbps for all other links. This results in the following three bottlenecks: 1→2, 2→3, and 3→4. Right: The dumbbell topology.

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 achieves the average throughput. Similarly, if the mean normalized throughout of both protocols is one, then they achieved the same average throughput.

Figures 4–10 show the normalized throughput and the mean normalized throughput for different numbers of flows, topologies, link speeds, queue disciplines, propagation delays, and with and without cross traffic. In these experiments, α and β were fixed at 0.999 and 3.0, respectively. For comparison purposes, Figures 6 and 8 compare the throughput when TCP-SACK and TCP-Reno compete for bandwidth. 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.

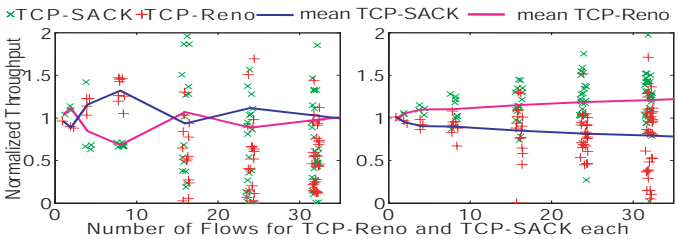

Fig. 8: Normalized Throughput of TCP-Reno and TCP-SACK. The left-hand plot shows the throughput of TCP-SACK and TCP-Reno for the drop-tail queue discipline, while the right-hand plot shows the results for RED queue discipline. In these simulations the topology was a single bottleneck topology with a 1.5Mbps bandwidth bottleneck. The round-trip propagation delay was 20ms and the queue size was 25 packets.

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

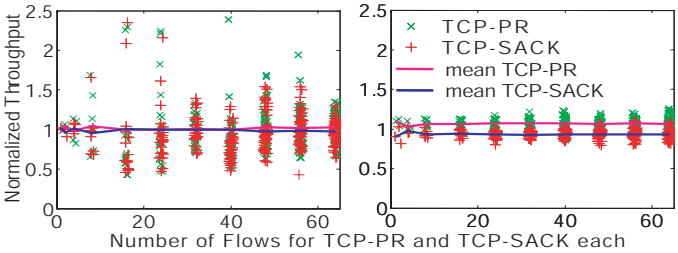

Fig. 10: Normalized Throughput for TCP-PR and TCP-SACK with HTTP Cross Traffic. The left-hand plot shows the results for the drop-tail queue discipline while the right-hand plot shows the results for RED queue discipline. In these simulations the topology was the parking-lot topology with a 15Mbps links. Each link has the round-trip propagation delay of 20ms with a queue size of 250 packets.

the variation of the throughput and is defined by

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

where I denotes the set of flows of a particular protocol, and $|I|$ the number of elements in the set I. Figure 11 shows the coefficient of variation for ten simulations as well as the mean coefficient of variation for the simulation set. From Figures 4–11, we conclude that the mean and variance of the throughput for TCP-PR and TCP-SACK are similar. In light of these results, incremental TCP-PR deployment should have no adverse effects on competing flows that use other implementations of TCP.

Figure 12 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). Surprisingly, fairness is maintained for a wide range of α and β. Note that for $\beta = 1$, TCP-SACK exhibits higher 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 throughput is essentially the same for $1 < \beta < 5$. Such extreme loss scenarios are not of

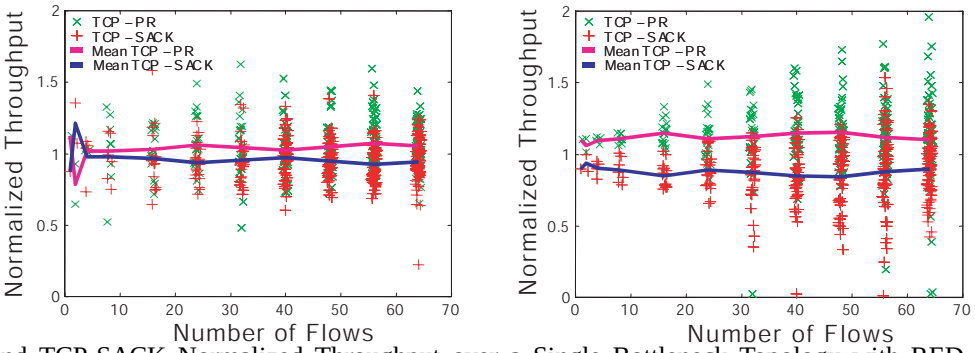

Fig. 4: TCP-PR and TCP-SACK Normalized Throughput over a Single Bottleneck Topology with RED Queue Discipline. The left-hand figure shows the case of 200ms round-trip propagation delay and a 250 packet queue, while the right-hand plot shows the case of 20ms round-trip propagation delay and a 25 packet queue. In both cases, the bottleneck link bandwidth was 15Mbps.

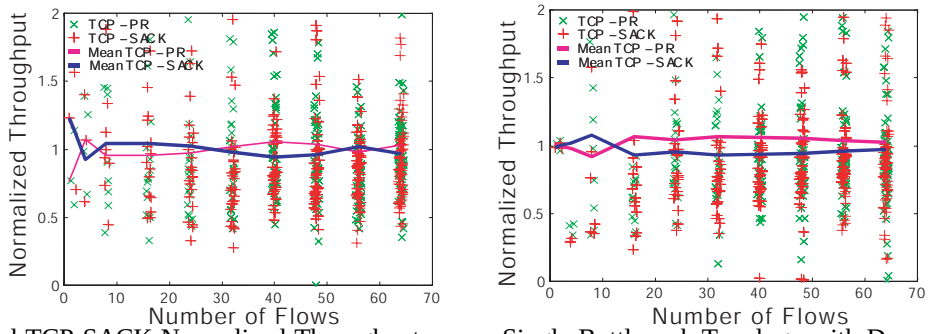

Fig. 5: TCP-PR and TCP-SACK Normalized Throughput over a Single Bottleneck Topology with Drop-Tail Queue Discipline. The left-hand figure shows the case of 200ms round-trip propagation delay and a 250 packet queue, while the right-hand plot shows the case of 20ms round-trip propagation delay and a 250 packet queue. In both of these cases, the bottleneck link bandwidth was 15Mbps.

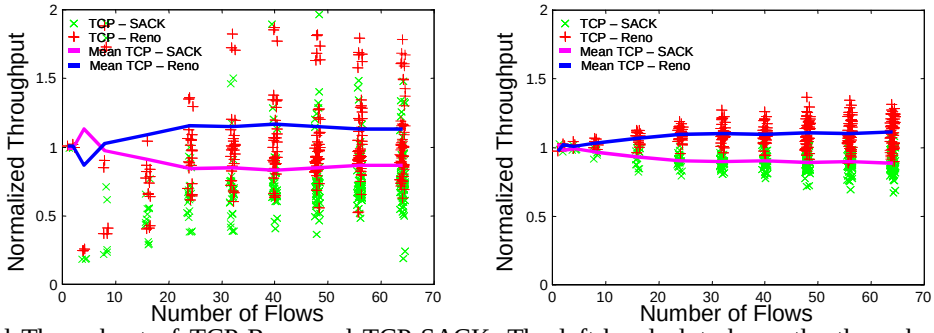

Fig. 6: Normalized Throughput of TCP-Reno and TCP-SACK. The left-hand plot shows the throughput of TCP-SACK and TCP-Reno for the drop-tail queue discipline, while the right-hand plot shows the results for RED queue discipline. In these simulations the topology was a single bottleneck topology with a 15Mbps bandwidth bottleneck. The round-trip propagation delay was 20ms and the queue size was 250 packets.

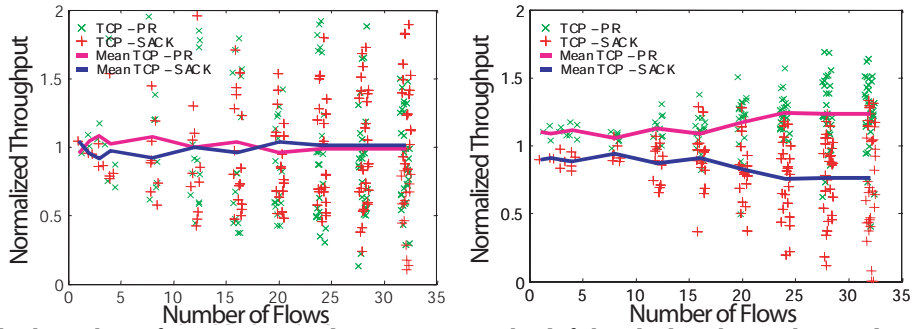

Fig. 7: Normalized Throughput for TCP-PR and TCP-SACK. The left-hand plot shows the results for the drop-tail queue discipline while the right-hand plot shows the results for RED queue discipline. In these simulations the topology was a single bottleneck topology with a 1.5Mbps bandwidth bottleneck. The round-trip propagation delay was 20ms and the queue size was 25 packets.

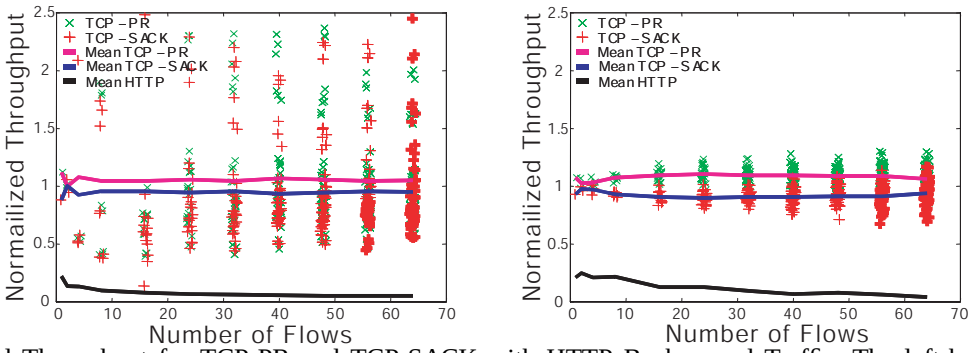

Fig. 9: Normalized Throughput for TCP-PR and TCP-SACK with HTTP Background Traffic. The left-hand plot shows the results for the drop-tail queue discipline while the right-hand plot shows the results for RED queue discipline. In these simulations the topology was a single bottleneck topology with a 15Mbps bandwidth bottleneck. The round-trip propagation delay of 20ms and a queue size of 250 packets.

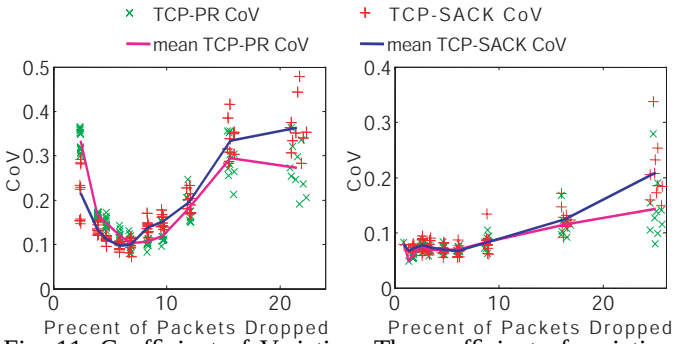

Fig. 11: Coefficient of Variation. The coefficient of variation as a function of packet loss probability. The variation in loss probability was simulated by decreasing the link bandwidth. The left plot is the coefficient of variation for the dumbbell topology and the right plot is for the parking lot topology. The single bottleneck had a round-trip propagation delay of 20ms and a queue size of 250 packets. The parking-lot topology had a round-trip propagation delay of 20ms with a queue size of 250 packets. In both cases, the link speed was 15Mbps and the drop-tail queueing discipline was used. The parking-lot topology had HTTP cross traffic.

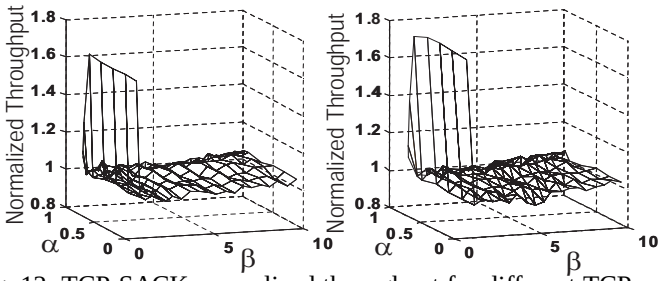

Fig. 12: TCP-SACK normalized throughput for different TCP-PR parameters. The left plot shows the mean normalized throughput of TCP-SACK using the single bottleneck topology, while the right plot shows normalized throughput for the parking-lot topology. The single bottleneck had a round-trip propagation delay of 20ms and a queue size of 250 packets. The parking-lot topology had a two-way propagation delay of 20ms with a queue size of 250 packets. In all cases, the link speed was 15Mbps and the drop-tail queueing discipline was used. The parking-lot topology had HTTP cross traffic.

particular concern since TCP's throughput is very low when the loss probability is this large.

Under normal traffic conditions, fairness is not so much evidence of the remarkableness of TCP-PR, but rather it attests to the robustness of additive-increase/multiplicative-decrease (AIMD) schemes. An important feature of these schemes is that if two flows detect drops at the *same* rate, then their congestion windows will converge to the same value. In fact, it was shown in [34] and, in more detail, in [35] that, at least for a dumbbell topology, competing TCP flows 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.

While the focus here is on fairness, packet delay is also of interest. TCP-PR results in delays that, on average, are similar to today's implementations of TCP, even though their loss detection mechanisms differ. In the case of a single packet loss in the middle of a file transfer, today's implementations of TCP will deliver the lost packet $(\beta-1)RTTs$ sooner than TCP-PR. We note that our simulations use $\beta = 3$. On the other hand, if there are over β losses, TCP-PR will deliver packets sooner than TCP New Reno. In the multiple loss scenario, the packet delay of TCP-SACK compared to TCP-PR depends (in a complicated way) on the total number of losses and the window size. In general, as the number of losses increases, the difference between TCP-PR and TCP-SACK decreases.

In the case of small file transfers, it is likely that a packet loss will not invoke triple duplicate ACKs (either because `cwnd` is small, or because the lost packet is near the end of the file). In the case that the packet is at the end of the file, TCP-PR will deliver the packet sooner than current TCP implementations. On the other hand, if the packet loss occurs when `cwnd` is small, TCP-PR will recognize the loss sooner than many of today's TCP implementations. Exceptions are implementations that use *Limited Transmit* [36], which allows packets to be sent even when DUP-ACKs arrive. Such implementations would detect the loss two RTTs after the lost packet is sent, whereas TCP-PR would detect it $\beta \times RTT$ after it was sent. Implementations that do not use Limited Transmit enter timeout in such situations. In order to maintain fairness

with implementations that do not use Limited Transmit, TCP-PR will enter the extreme loss state and will deliver the packet at the same time as these implementations.

VI. Performance under Packet Reordering: Comparison with other Methods

This section compares the performance of TCP-PR against existing algorithms that make TCP more robust to packet reordering. Two types of packet reordering are investigated: packet reordering due to queue swaps which might occur within a switch as suggested by [1], and reordering due to multipath routing. Our goal in designing TCP-PR is to provide a transport protocol that is suitable for environments that exhibit persistent reordering, yet achieving adequate performance (including fairness) in environments with no or occasional reordering. While queue swaps may lead to the latter scenarios, as discussed in Section IV, multipath routing typically results in persistent packet reordering and can be especially demanding since packets sent back-to-back may experience very different latencies.

Besides TCP-PR, several other approaches to TCP are considered. These include TCP-SACK with the DSACK feature enabled. In this case, spurious drops are detected, but no mitigation is performed. This method is labeled *DSACK-NM*. In [1], several methods were examined and are considered here as well. These methods use "limited transmit" [36] so that packets are still sent when duplicate ACKs arrive. These methods also adjust *dupthresh*. In the graphs that follow, *Inc by 1* refers to the approach that increments the *dupthresh* by one every time a spurious retransmission is detected. Upon detecting a spurious retransmission, the method labeled *Inc by N* increases the *dupthresh* such that the just observed spurious retransmission would not have occurred. The method labeled *EWMA* varies the *dupthresh* according to an exponentially weighted moving average filter. Another method first suggested in [18] and further investigated in [1] is referred to as *time-delayed fast-retransmit* (*TD-FR*). In this case, fast retransmit is only entered when a triple duplicate ACK is observed and a certain amount of time has passed since the packet was sent. Recently another method for adapting *dupthresh* has been suggested [17]. Since a simulation implementation of this method is not yet available, it was not included in this comparison.

A. Performance under Packet Reordering Due to Queue Swaps

In [1] the different versions of TCP were compared by examining their performance in the face of queue swaps. A queue swap is when two packets in a queue are exchanged. We assume that queue swap events occur every one second, and each event consists of K individual packet swaps. The ns-2 simulations presented refer to a single bottleneck topology with drop-tail queuing, a maximum queue size of 250 packets, and one flow. The experiments were repeated for different propagation delays. Since the critical concern is whether or not the throughput is affected by queue swaps, Figure 13 shows the relationship between throughput and the number K of packet swaps per queue swap event.

The left plot in Figure 13 shows the throughput for a 30ms propagation delay and the right plot for a 180ms propagation delay. In the low propagation delay case, most algorithms work relatively well. Indeed, even TCP-SACK with no special mitigation for packet reordering achieves a throughput that is merely 1.5% smaller than the other algorithms. However, for higher propagation delays the situation is quite different. In the right plot of Figure 13, we observe that most methods only achieve a throughput that is 25% smaller than TCP-PR and TD-FR. In both cases the throughput achieved by TCP-PR is nearly independent of the number of packet swaps per swap event. This result is to be expected since reordering of packets in a queue will merely produce duplicate ACKs. The ACK arrival rate is not changed and hence TCP-PR's throughput is not affected.

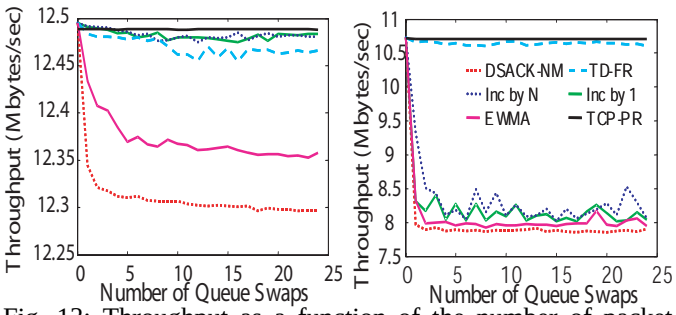

Fig. 13: Throughput as a function of the number of packet swaps per swap event. The left plot shows the throughput for a 30ms propagation delay and the right plot for a 180ms propagation delay.

B. Performance under Packet Reordering Due to Multipath Routing

As before, we ran extensive simulations using ns-2 to compare the performance of the different algorithms in the face of persistent packet reordering due to multipath routing. The tests presented were performed on the topology shown in Figure 14. Different sets of simulations were performed. In the first set, the propagation delay for each link was set to 10ms, while in the second set it was set to 60ms. These simulations were performed with and without background traffic.

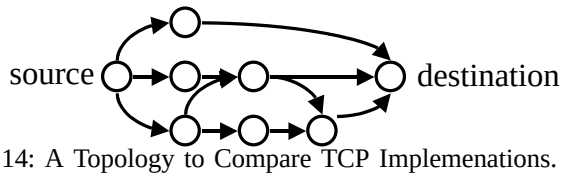

Fig. 14: A Topology to Compare TCP Implemenations. Each link has a delay of 20ms, bandwidth of 10Mbps and queue has a size of 100 packets.

Many multipath routing strategies are possible over this topology. We developed a family of strategies that is parameterized by a single variable ε (cf. [37] for details). This parameter controls the degree to which routing accounts for link cost: When $\varepsilon = \infty$ the link cost is heavily penalized, resulting in minimum-hop routing. When $\varepsilon = 0$ the link cost is not penalized at all and all independent paths from source to

destination are used with equal probability. Intermediate values of ε correspond to compromises between these two extreme cases. We compared the performance of TCP-PR with that of the various TCP versions with $dupthresh$ compensation schemes in [1]. This was done for several fixed routing strategies, each corresponding to a distinct value of ε. In these simulations only one flow was active at a time.

Figure 15 shows the throughput for various values of ε. The simulations show that for $\varepsilon = 500$ (single-path routing), all methods achieve the same throughput. For $\varepsilon = 0$ (full multi-path routing) most protocols other than TCP-PR suffer drastic decreases in throughput. The exception is time-delayed fast-recovery (TD-FR) and TCP-DCR, which still achieves a reasonable throughput for small values of ε when the propagation delay is small (the left plot in Figure 15). However, as shown in Figure 15 and Figure 16, TD-FR and TCP-DCR still suffer a large decrease in throughput when the propagation delay is increased. The reason for this drop in throughput is that TD-FR and TCP-DCR make use of both $dupthresh$ and timers. While the "limited transmit algorithm" attempts to reduce it, 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. This confirms that duplicate ACKs are indicative of packet loss in single path routing, but their occurrence convey little information when multi-path routing is utilized.

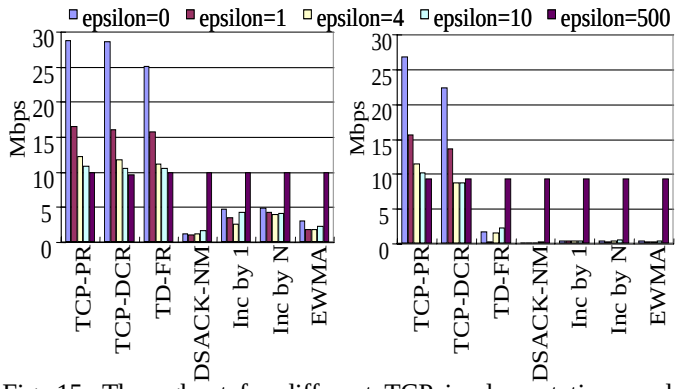

Fig. 15: Throughput for different TCP implementations and different degrees of multi-path routing. $\varepsilon = 500$ corresponds to single path routing, whereas for smaller values of ε alternative paths are sometimes used. In the limit $\varepsilon = 0$, all paths are used with equal probability. The left plot corresponds to the topology in Figure 14 with a 10ms propagation delay for each link and the right plot corresponds to the same topology but with a 60ms propagation delay for each link.

VII. 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

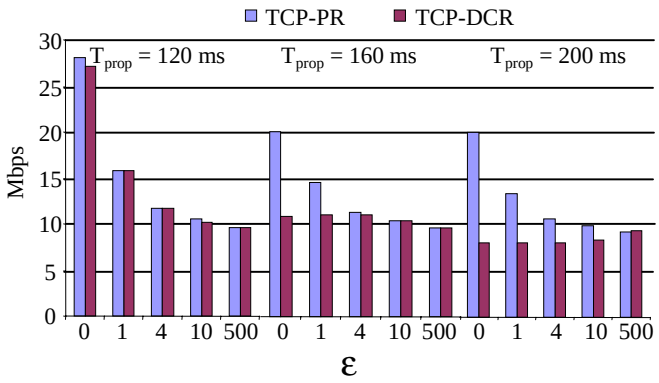

Fig. 16: Throughput of TCP-PR and TCP-DCR for different values of ϵ and different propagation delays.

demanding than TCP-(New)Reno but carries essentially the same overhead as TCP-SACK.

Because of its robustness to persistent packet reordering, TCP-PR allows mechanisms that introduce packet reordering as part of their normal operation to be deployed in 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 [8]); and traffic engineering approaches.

A Linux implementation of TCP-PR is under development and is available at [38]. Furthermore, TCP-PR is expected to 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] J. Bennett and C. Partridge, "Packet reordering is not pathological network behavior," *IEEE/ACM Transactions on Networking*, vol. 7, no. 6, 1999.
[4] L. Cottrell, "Packet reordering," 2000.
[5] F. Wang and Y. Zhang, "Improving TCP performance over mobile ad-hoc networks with out-of-order detection and response," in *ACM MOBIHOC*, 2002.
[6] T. Dyer and R. Boppana, "A comparison of TCP performance over three routing protocols for mobile ad hoc networks," in *ACM MOBIHOC*, 2001.
[7] G. Holland and N. Vaidya, "Analysis of TCP performance over mobile ad-hoc networks," in *ACM MOBICOM*, 1999.
[8] S. Blake, D. Black, M. Carlson, E. Davies, Z. Whang, and W. Weiss, "An architecture for differentiated services." RFC 2475, 1998.
[9] D. Bertsekas, *Network Optimization: Continuous and Discrete Models*. 1998.
[10] N. Taft-Plotkin, B. Bellur, and R. Ogier, "Quality-ofservice routing using maximally disjoint paths," 1999.
[11] S. Bohacek, J. Hespanha, K. Obraczka, J. Lee, and C. Lim, "Secure stochastic routing," in *ICCCN*, 2002.
[12] R. Teixeira, K. Marzullo, S. Savage, and G. M. Voelker, "Characterizing and measuring path diversity of Internet topologies," in *SIGMETRICS*, 2003.
[13] A. Nasipuri and S. Das, "Demand multipath routing for mobile ad hoc networks," in *Networks, Proceedings of the 8 Th Annual IEEE Internation Conference on Computer Communications and Networks (ICCCN)*, 1999.

[14] M. Pearlman, Z. Haas, P. Sholander, and S. Tabrizi, "the impact of alternate path routing for load balancing in mobile ad hoc networks," in *Proceedings of the ACM MobiHoc*, 2000.

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

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

[17] 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.

[18] V. Paxson, "End-to-end internet packet dynamics," in *ACM SIGCOMM*, 1997.

[19] S. Bhandarkar, N. Sadry, A. L. N. Reddy, and N. Vaidya, "Tcp-dcr: A novel protocol for tolerating wireless channel errors," *IEEE Transactions on Mobile Computing*, 2004.

[20] B. Sikdar, S. Kalyanaraman, and K. S. Vastola, "Analytic models and comparative study of the latency and steady-state throughput of TCP Tahoe, Reno and SACK," *IEEE/ACM Transactions on Networking*, 2003.

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

[22] N. L. for Applied Network Research (NLANR).

[23] S. Bohacek, "A stochastic model of tcp and fair video transmission," in *INFOCOM*, 2003.

[24] R. Braden, "Requirements for Internet hosts – communication layers," *RFC 1122*, Oct. 1989.

[25] V. Jacobson, "Congestion avoidance and control," *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, vol. 18, 4, pp. 314–329, 1988.

[26] M. Allman and V. Paxson, "On estimating end-to-end network path properties," in *Proc. of ACM SIGCOMM '99*, 1999.

[27] 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.

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

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

[30] S. Floyd, "Connections with multiple congested gateways in packet-switched networks part 1: One-way traffic," *ACM Computer Communication Review*, vol. 21, no. 5, pp. 30–47, October 1991.

[31] D. Katabi, M. Handley, and C. Rohrs, "Internet congestion control for future high bandwidth-delay product environments," 2002.

[32] F. Hernández-Campos, J. S. Marron, G. Samorodnitsky, and F. D. Smith, "Variable heavy tail duration in internet traffic," in *Proc. of IEEE/ACM MASCOTS 2002*, 2002.

[33] S. Floyd, "Scripts for adaptive red simulations." Available from http://www.icir.org/floyd/adaptivered/papersims/single1.tcl.

[34] 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.

[35] S. Bohacek, J. Hespanha, K. Obraczka, and J. Lee, "Analysis of a TCP hybrid model," in *Proc. of the 39th Annual Allerton Conference on Communication, Control and Computing*, 2001.

[36] M. Allman, H. Balakrishnan, and S. Floyd, "Enhancing TCP's loss recovery using limited transmit," *RFC 3042*, 2001.

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

[38] "The TCP-PR web page. available at http://eecis.udel.edu/ bohacek/tcp-pr.htm."