University of Helsinki
Department of Computer Science
Series of Publications C, No. C-2001-53

Making TCP Robust Against Delay Spikes

Andrei Gurtov

Helsinki, November 2001

Report C-2001-53

University of Helsinki
Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

The papers in the series are intended for internal use and are distributed by the author. Copies may be ordered from the library of Department of Computer Science.

Making TCP Robust Against Delay Spikes

Andrei Gurtov

Department of Computer Science, University of Helsinki

Report C-2001-53
November 2001
14 pages

Abstract. This document discusses optimizations for a TCP sender that are most helpful in the presence of delays spikes, but are seemingly suitable for general deployment. The motivation for this work is increasing popularity of links (e.g. provided by cellular networks) that have delay spikes exceeding the usual link latency by several times. The effect of a delay spike on TCP Tahoe, Reno, NewReno and SACK is described. The document recommends timing every segment and restarting the retransmit timer to achieve a more conservative RTO estimate. Furthermore, it discusses how a series of DUPACKs should be treated.

Key Words: TCP, cellular networks, delay spike, spurious timeout

CR Classification: C.2.1

Contents

1. Introduction **1**

2. Sources of delay spikes **2**

3. Delays and interactions with TCP mechanisms **2**

4. Making TCP robust against delay spikes **5**

4.1 Restarting the retransmit timer 5

4.2 Timing every segment 6

4.3 Treating a DUPACK series 8

4.4 Ignoring DUPACKs for oldest outstanding segment after RTO 11

5. Conclusions **12**

References **13**

1. Introduction

The increasing number of users access the Internet via data links provided by cellular Wide Area Wireless Networks (W-WANs). Due to link outages, handovers, and priority blocking delay spikes in order of tens of seconds can occur leading to spurious TCP timeouts and unnecessary retransmissions.

Making TCP robust against delay spikes may include mechanisms on signalling [LU01a], detection [LU01b] and response [LG01] [BA01c] to spurious timeouts. Recommendations made in this documents are expected to be suitable for general deployment even when these mechanisms are implemented.

This document proposes several mechanisms to increase the conservativeness of the TCP retransmission timer. It has a positive effect of making it more tolerable to delays spikes. However, a more conservative RTO timer also has the drawback of a lengthy recovery in case the RTO has not been spurious, i.e. occurred due to segment losses [AP99]. To avoid this performance drawback, non-spurious RTOs should be avoided as much as possible. Thus, all relevant mechanisms that reduce a probability of RTO in presence of packet losses are recommended. Two most important such mechanisms are SACK [RFC2018] [BA01a] and the Limited Transmit algorithm [RFC3042]. The New Reno algorithm [RFC2582] may be used by the TCP sender when the SACK option is not available on the connection. A list of other experimental methods for enhancing loss recovery and avoiding non-spurious RTOs can be found in [RFC3155].

An important observation is that for W-WAN users and operators the battery power consumption and radio resource usage are often as important as the throughput of the link. This suggests that the amount of data sent over the wireless link should be minimized even at the trade-off of extra delay in data delivery. With regard to this document it means that TCP features that avoid unnecessary packet retransmissions have an extra value.

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in [RFC2119]. A segment that acknowledges new data is referred as an ACK. The DupThresh corresponds to the number of DUPACKs necessary to trigger the fast retransmit algorithm, the default value is three DUPACKs.

2. Sources of delay spikes

TCP does not react well to large delays (several times the usual RTT) that occur suddenly. Without a chance to adapt its retransmission timer to such a delay, TCP has to assume that outstanding segments were lost and retransmits them. There is a number of possible reasons why delay spikes in order of tens of seconds can occur [GU01a].

(1) A long delay spike can be a result of link layer recovery from a link outage due to temporal loss of radio coverage for example while driving into a tunnel or stepping into an elevator.

(2) During a handover the mobile terminal may have to perform some time-consuming actions before data can be transmitted in a new cell. Many W-WANs in such a case try to provide seamless mobility, that is internally re-route packets from the old to the new base station at the expense of additional delay.

(3) Blocking by high-priority traffic may occur when an arriving circuit-switch call or higher priority data user temporally preempts the radio channel.

Delay spikes in the Internet can occur for example due to routing changes, but are less frequent than in cellular networks [AP99]. Delay spikes are often coupled with increased likelihood of packets losses in the network. In addition, the network path conditions can change heavily after a delay, for example the available bandwidth can shrink tenfold, after a handover from a fast to a slower cell.

3. Delays and interactions with TCP mechanisms

This section briefly summarizes reaction of a conformant TCP to a delay spike. The Reno description is borrowed from [LK00]. Description of Tahoe, New Reno and SACK is based on experiments in the NS2 simulator [NS] version 2.1b8. The experiment included inserting a delay approximately three times the RTT in the beginning of the TCP connection without data losses. The line rate was 9.6 kbps and the latency 300 ms. Traces are available at [GU01b].

When a sudden delay that exceeds the current value of the TCP retransmission timer occurs in the data transfer, TCP times out and retransmits the oldest outstanding segment, as shown in Figure 1. Since data segments are delayed but not lost, the retransmission is unnecessary and the timeout is spurious. The sender interprets the ACK generated by the receiver in response to a delayed segment as related to the retransmission, not the original segment. This happens due to the retransmission ambiguity problem as the ACK bears no information which segment, original or retransmitted,

has generated it. Encouraged by arriving ACKs, TCP retransmits all outstanding segments using the slow start algorithm. Also, a number of new segments allowed by the congestion window are transmitted. Such a retransmission policy is called go-back-N since the sender forgets about all segments it has earlier transmitted. When retransmitted segments arrive to the receiver they generate DUPACKs since the original segments have already been delivered. When the threshold of three DUPACKs is reached at the sender, a spurious fast retransmit is triggered. The presumably missing segment is retransmitted and the congestion window is reduced that causes a pause in transmission of new segments. From this point, TCP behavior depends on the flavor of the TCP implementation.

TCP receiver sends DUPACKs in response to out-of-order segments. In other words, a DUPACK series appears due to unnecessary retransmissions, if segments have been duplicated by the network, or due to a packet loss. A DUPACK series triggers a fast retransmit when the DupThresh is reached, unless not prevented by the ”bug fix” or SACK information (Section 4.3).

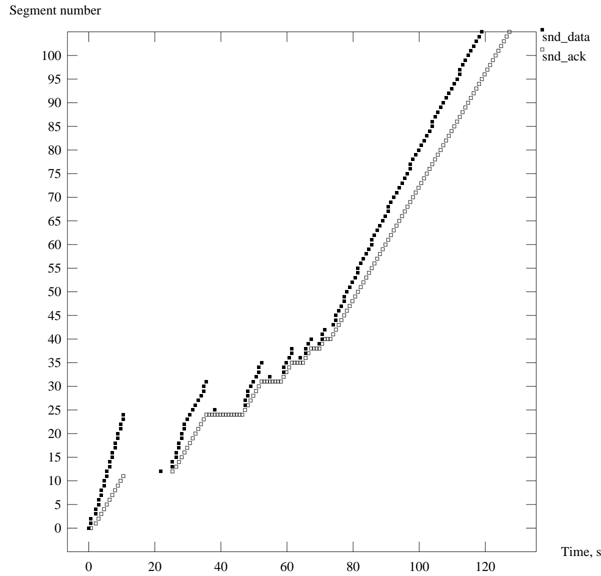
TCP Tahoe in Figure 1(a) ignores arriving DUPACKs after the fast retransmit, as it does not implement fast recovery. However, when partial ACKs start to arrive, Tahoe retransmits outstanding segments unnecessary using the slow start algorithm. This in turn leads to a sequence of DUPACKs causing a fast retransmit and repeating the cycle until the flight size is reduced to the point when the fast retransmit cannot be triggered anymore. After that, the connection proceeds normally slowly increasing the window in congestion avoidance.

TCP Reno in Figure 1(b) enters the fast recovery phase after the false fast retransmit and does not perform any additional unnecessary retransmissions unless the RTO timer expired during fast recovery (Section 4.4)

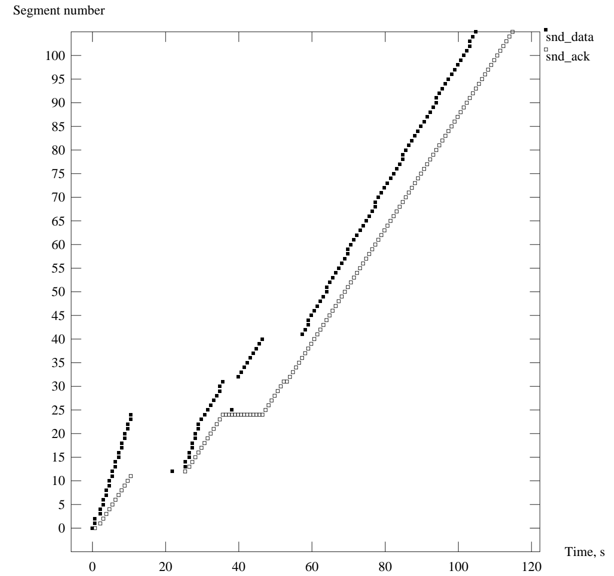
ACKs arriving after false fast retransmit are partial because they are not confirming the reception of the foremost outstanding segment at that time. The New Reno [RFC2582] algorithm retransmits the presumably missing segment at each new partial acknowledgment in Figure 1(c). A new DUPACK series is triggered by these unnecessary retransmissions, in a similar way as for Tahoe. This continues over and over until too few packets are in flight to trigger a spurious fast retransmit.

Fortunately, preventing the first false fast retransmit after the spurious timeout by the ”bug fix” (Section 4.3) also solves the problem of continues unnecessary retransmits for Tahoe and New Reno.

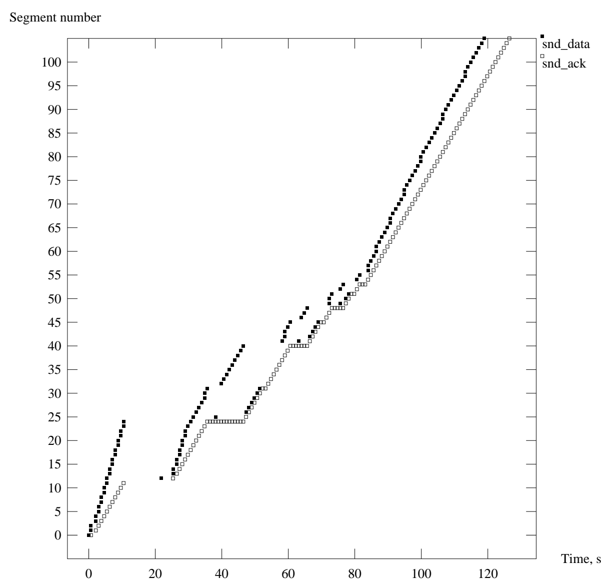
TCP SACK in Figure 1(d) can avoid the false fast retransmit but cannot avoid the go-back-N behavior. The information on the retransmitted segments during go-back-N comes only in DUPACKs. Using the D-SACK extension of SACK (duplicate-SACK) allows reporting to the sender



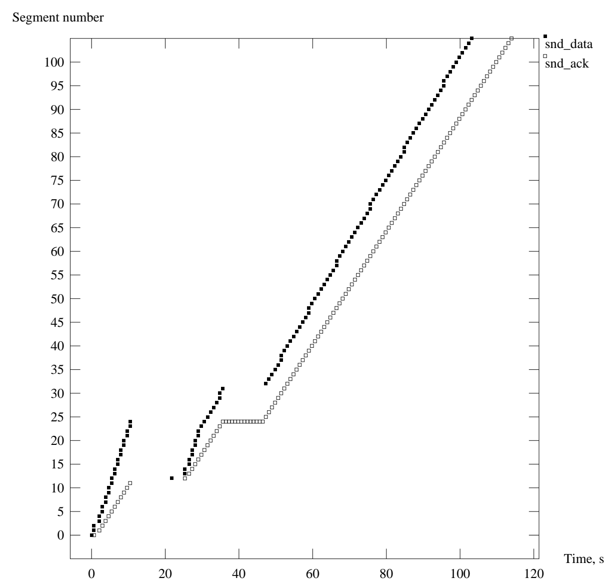
(a) Tahoe



(b) Reno



(c) NewReno



(d) SACK

Figure 1: Response to a spurious timeout by different TCPs. Timestamps are enabled and the ”bug fix” is disabled.

the sequence number of a packet that triggered a DUPACK [RFC2883]. D-SACK info comes too late to avoid go-back-N retransmissions, but it can be used to learn about unnecessary retransmissions [BA01b] and adapt the for the future.

In our experience, the impact of delay spikes on real-world TCPs is worse than have been described above. Almost every TCP tested against a delay spike revealed implementation bugs [GU01b]. One goal of this document is to make the TCP developers aware of the negative effect of delay spikes.

4. Making TCP robust against delay spikes

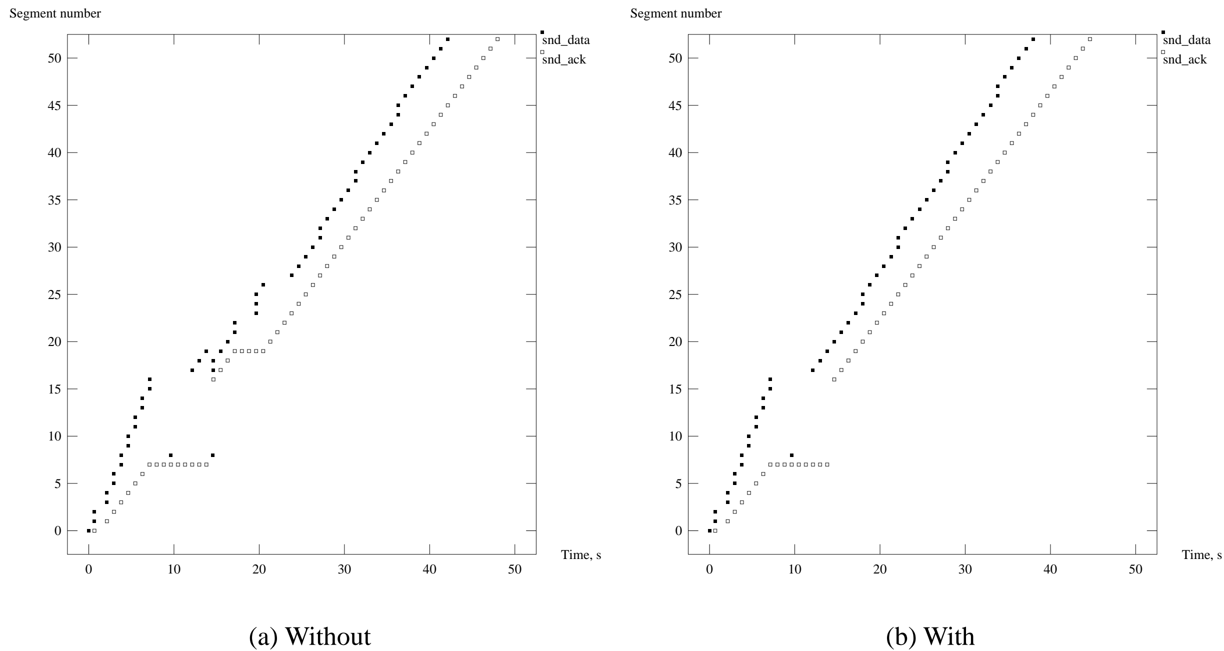
4.1 Restarting the retransmit timer

The obvious way to reduce the number of spurious RTOs in the presence of long sudden delays is to make the RTO timer more conservative than [RFC2988], which recommends restarting RTO only upon an ACK that acknowledges new data. Restarting the RTO timer also when a segment is retransmitted or upon a DUPACK is clearly more conservative approach which is explicitly allowed by [RFC2581].

Restarting the retransmit timer after performing fast retransmit gives a TCP sender more time to wait for the retransmitted segment to be acknowledged. Otherwise, spurious RTO can occur during fast recovery as shown in Figure 2(a), even when no delay spike is present and the RTT samples are collected frequently. Figure 2(b) shows that restarting the retransmit timer when the fast retransmit is triggered prevents the spurious timeout. Restarting the retransmit timer on fast retransmit is a common implementation strategy among existing TCPs. The limited transmit algorithm can increase fast recovery by two DUPACKs, thus raising the likelihood of a spurious RTO, especially if the retransmit timer is not restarted.

Restarting the retransmit timer on DUPACKs is discussed in more detail in Section 4.3. The general argument to consider here is that TCP would not want to timeout while it gets some feedback that segments are being delivered by the network. In addition, DUPACKs could have useful SACK information.

Restarting the retransmit timer on partial ACKs is discussed in [RFC2582]. The Impatient variant of NewReno restarts the retransmit timer only on the first partial ACK, while the Slow-but-Steady variant upon each partial ACK. The Impatient version may timeout during a lengthy fast recovery, and proceed with the go-back-N and slow start. This may result in a quicker recovery when a large number of segments is lost. The Slow-but-Steady version can stay in fast recovery for



(a) Without (b) With

Figure 2: Fast retransmit due to a lost segment with and without reset of the retransmit timer. TCP Reno with timestamps enabled.

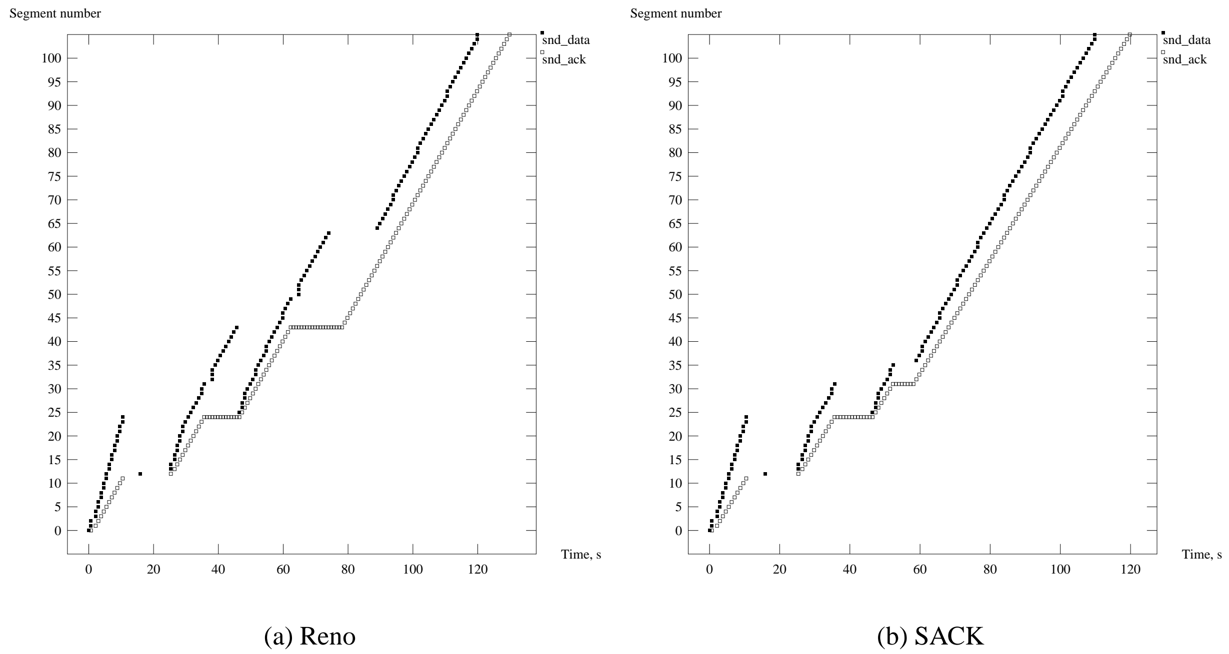
a long time, however avoiding unnecessary retransmissions which are likely during go-back-N. If a delay spike occurs during the fast recovery phase, the Impatient version is more likely experience a spurious timeout. In the presence of delay spikes and when unnecessary retransmissions are costly, the TCP MAY prefer the Slow-but-Steady version, that is restarting the timer on each DUPACK.

Recommendation: The retransmit timer SHOULD be restarted after fast retransmit. TCP MAY restart the retransmit timer on partial ACKs when unnecessary retransmissions are costly and delay spikes are likely.

4.2 Timing every segment

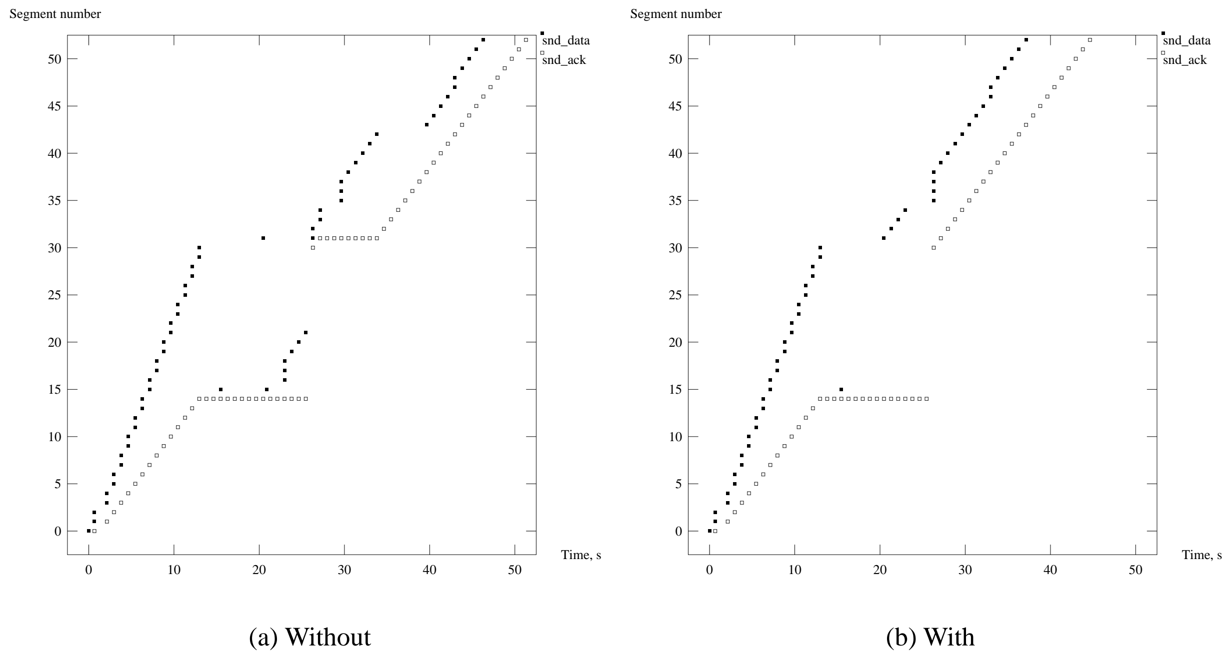
Traditionally, TCPs have been collecting one RTT sample per a window of data [Jac88]. This can lead to underestimating the link RTT and spurious RTOs.

During the slow start phase the queuing delay is increasing rapidly and the RTO value applied just before a new RTT sample is collected may underestimate the current RTT. Increasing the RTTvar coefficient from two to four prevents a spurious timeout during the slow start [Jac88]



(a) Reno (b) SACK

Figure 3: Response to a spurious timeout when timestamps are disabled.



(a) Without (b) With

Figure 4: Fast retransmit in TCP Reno due to a lost segment with and without timestamps.

assuming no delay spikes are present. Still, the standard RTO timer [RFC2988] may exceed the link RTT only by a small edge allowing even a small RTT spike to cause a spurious timeout.

In congestion avoidance, a spurious RTO without a delay spike is still possible with the standard RTO timer, however, only if a very large window is used on a bandwidth-limited link [Jac88]. Unfortunately, overbuffering is seems to be a frequent case for slow links [LU99].

Timing every segment eliminates the effect of lagging RTO behind a rapidly increasing link RTT, thus decreasing the likelihood of a spurious timeout. Timing every segment can be implemented with or without the timestamp option [RFC1323]. Using the timestamp option has the advantage in allowing use of retransmitted segments for RTT measurement which is otherwise blocked by the Karn's algorithm [KP87].

Experiments using NS show that the RTT spike tolerated by TCP without a spurious timeout could be twice higher when every segment is used for RTT estimation [GU01b]. Figure 3(a) and 3(b) illustrate the response to the same delay spike as in Figure 1(b) and 1(d) when the RTT sample is collected only once per window. The retransmit timer is clearly more aggressive without timestamps, as the first retransmission occurs 4 secs earlier. Furthermore, both Reno and SACK experience a second spurious RTO during a DUPACK sequence when timestamps are not used.

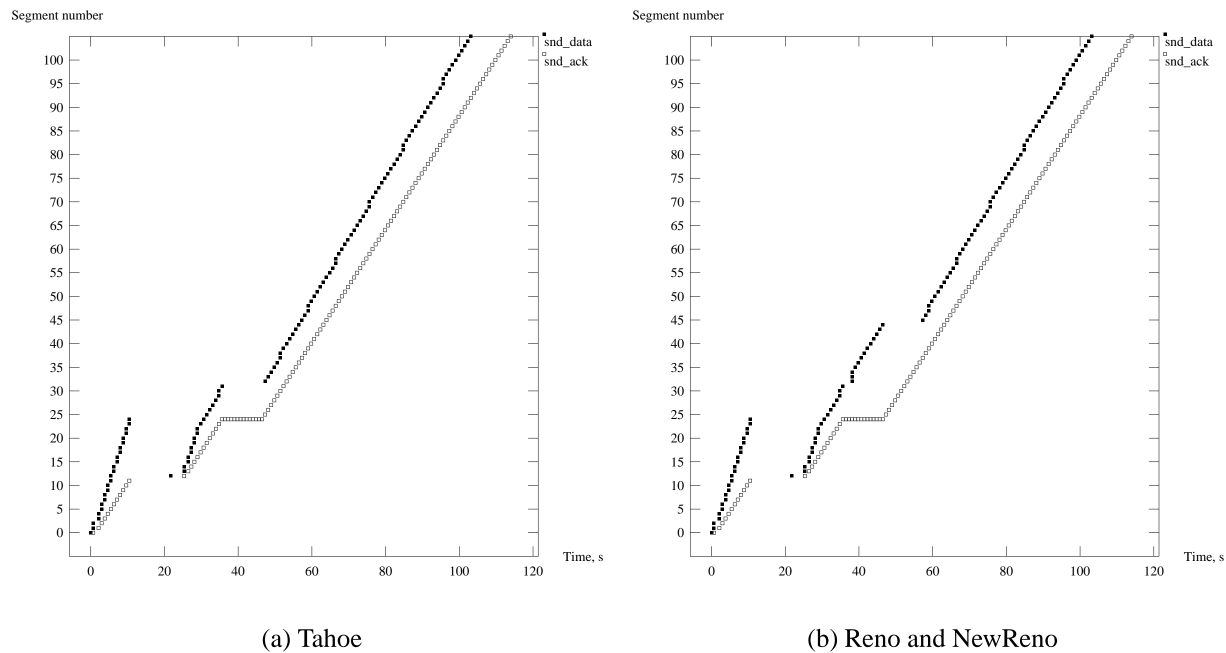
Figure 4(a) shows another example when the spurious RTO occurs during fast recovery. Waiting for 5 secs more would avoid the timeout in this example. Using the timestamp option in Figure 4(b) allows to complete the fast recovery phase without a spurious timeout.

Recommendation: TCP SHOULD collect RTT samples more frequently than once per RTT to decrease the likelihood of a spurious RTO.

4.3 Treating a DUPACK series

Despite of a conservative retransmit timer, a spurious RTO can still occur. The resulting go-back-N behavior produces a large number of DUPACKs triggered by unnecessary retransmissions. The DUPACK series can cause a spurious fast retransmit and a spurious RTO.

Without SACK support on the connection, the receiver has no knowledge whether a DUPACK has been due to a unnecessary retransmission or due to a lost segment. To prevent unnecessary fast retransmits after a RTO, a "bug fix" has been suggested [RFC2582]. The "bug fix" disables fast retransmits until all segments outstanding at the time when RTO occurred are acknowledged. A less careful version of this restriction allows the fast retransmit when DUPACKs arrive for the foremost outstanding packet, while a more careful version does not. A spurious timeout without



(a) Tahoe

(b) Reno and NewReno

Figure 5: Response to a spurious timeout when the ”bug fix” is enabled.

lost segments presents exactly the situation when DUPACKs arrive for the foremost outstanding segment. The careful version is recommended [RFC2582]. Response to a spurious timeout with the careful ”bug fix” is shown in Figure 5(a) and 5(b).

When SACK is supported by the connection, receiving a DUPACK without a SACK block or with a D-SACK block pointing below the cumulative ACK indicates that the DUPACK was triggered by a unnecessary retransmission (excluding a pathological case when the receiver has agreed to use SACK but does not send SACK info). Thus, there is no reason to enter the loss recovery phase, and the same behavior should be followed as without SACK when the fast retransmit is prevented by the ”bug fix”.

At the moment, the safest way is to follow [RFC2582] and ignore DUPACKs not covering the highest outstanding segment. Ignoring DUPACKs means no segment (re)transmission or changes to congestion control state. However, if a TCP sender simply ignores DUPACKs arriving when fast retransmit is disabled, it can lead to loosing the ACK clock. Whether incoming DUPACKs can be used to trigger transmission of segments is an open problem, and below we provide some facts to be considered. Note, that for DUPACKs before DupThresh, transmission of segments is covered by the limited transmit algorithm. Using the limited transmit algorithm when the ”bug fix”

is enabled shares the considerations below.

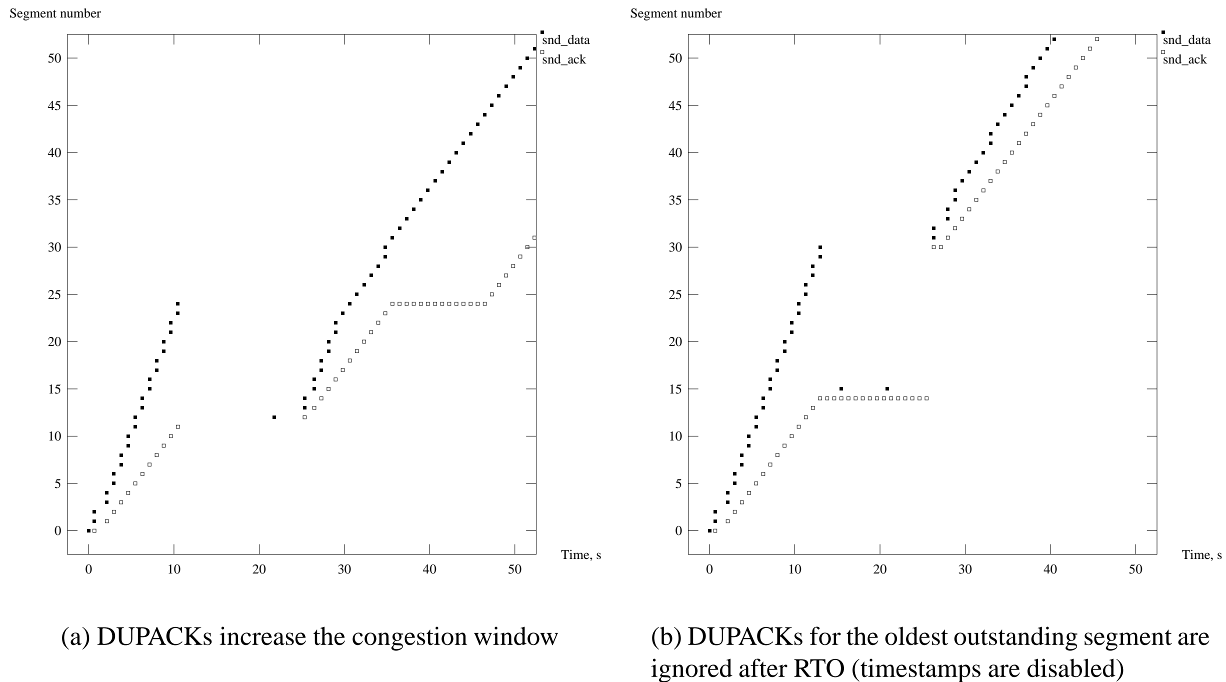
TCP Reno in NS2 uses DUPACKs to rigger transmission of segments during 37-50 secs as shown in Figure 5(b). When an ACK arrives at 50 sec, the congestion window is deflated that produces a pause in packet transmission during 50-58 sec. The alternative behavior when the congestion window is not inflated is shown in Figure 5(a). Correspondingly, segments are not transmitted upon DUPACKs and there is no pause due to deflating of the congestion window when an ACK arrives at 50 sec. There is no clear benefit in either approach, as illustrated by the equal download time for both connections. A possible modification is shown in Figure 6(a) when DUPACKs trigger transmission of segments and the congestion window is not deflated. However, this algorithm produces unstable behavior when evaluated in environment with congestion losses and therefore cannot be recommended. In opposite, transmitting a new segment every *second* DUPACK (similar to the Rate-Halving algorithm) reduces the transmission rate, but still preserves the ACK clock. Experiments show that this approach is stable in presence of congestion losses and improves the throughput.

If segments are transmitted on DUPACKs for the segment below the highest outstanding segment, they are retransmissions. If these retransmissions happen to be unnecessary, a new DUPACK series is created later. On the other hand, segments transmitted upon DUPACKs for the highest outstanding segment are new transmissions. Thus, one option would be to allow transmitting segments on DUPACKs only for the highest outstanding segment.

The retransmit timer may be restarted safely upon DUPACKs if no segments are transmitted after DupThresh. Restarting the timer decreases likelihood of spurious RTOs during a DUPACK series when delay spikes occur. However, timing every segment and restarting the timer on reaching DupThresh seem to provide a conservative enough retransmit timer in many cases. Restarting the retransmit timer on DUPACKs can lead to a lengthy recovery when the segment was lost.

The reason for banning transmission of a segment AND restarting the retransmit timer on DUPACKs is a situation when the last outstanding segment is lost. The receiver will keep sending DUPACKs until the lost segment is received. A newly transmitted segment on a DUPACK will trigger another DUPACK in the future creating an endless loop. The retransmit timer in this case serves as an back-up way to interrupt the loop. With SACK, this problem can appear only in a pathological case when the receiver does not report losses.

Recommendation: TCP without SACK SHOULD implement the careful version of the ”bug fix”. TCP with SACK SHOULD NOT enter the loss recovery phase when DUPACKs do not have a SACK block indicating a lost segment. TCP MAY restart the retransmit timer upon receiving a DUPACK. Transmitting segments upon DUPACKs above the oldest outstanding segment is an



(a) DUPACKs increase the congestion window

(b) DUPACKs for the oldest outstanding segment are ignored after RTO (timestamps are disabled)

Figure 6: Effect of DUPACKs when the "bug fix" is enabled.

open issue. However, TCP MUST NOT transmit a new segment and restart the retransmit timer for DUPACKs above the DupThresh.

4.4 Ignoring DUPACKs for oldest outstanding segment after RTO

TCP can time out during a series of DUPACKs, either during fast recovery phase as shown in Figure 2(a) and 4(a), or after a large number of unnecessary retransmissions as shown in Figure 4(a). If DUPACKs are delayed or lost, RTO can occur despite of restarting the retransmit timer upon DUPACKs. RTO in this situation may or may not be spurious; the recommendation in this section applies in both cases.

The proper behavior after RTO is specified in [RFC2581], that is to retransmit the oldest outstanding segment, wait for an ACK and back-off the RTO timer if it expires again. However, a fairly common behavior among TCPs is to use DUPACKs arriving after RTO to inflate the congestion window and clock out retransmissions of segments, as happens in Figure 4(a) at 25 secs. This behavior is observed at least in current versions of FreeBSD, Windows, and NS2 TCPs. This behavior can be a special case of the problem as unnecessary fast retransmits which is discussed in

Table 1: Summary of recommendations.

Mechanism	Use	SACK	Section
Restarting the retransmit timer			
after DUPACK for DupThresh	SHOULD	both	4.1
after partial ACK	MAY	both	4.1
after DUPACK above DupThresh	MAY	both	4.3
Timing every segment	SHOULD	both	4.2
Careful ”bug fix”	SHOULD	w/o	4.3
No recovery on DUPACKs without loss info	SHOULD	with	4.3
New segment on every second DUPACK (without restarting the retransmit timer)	MAY	both	4.3
Ignoring DUPACKs after RTO	SHOULD	both	4.4

Section 4.3. In this situation the fast retransmit does not make sense, since only a single segment retransmitted after RTO is assumed to be outstanding, and cannot cause enough DUPACKs to trigger the fast retransmit. Furthermore, since RTO is taken as an indication of severe congestion, it is unwise to retransmit segments on DUPACKs after RTO without getting any feedback for the first retransmission.

It has been observed with real TCPs that transmitting segments on DUPACKs after RTO can lead to a series of spurious timeouts as follows. TCP times out during a long DUPACK series caused by go-back-N retransmissions. After RTO, DUPACKs are triggering unnecessary retransmission of segments; resulting DUPACKs in the future cause RTO again. When DUPACKs are ignored after RTO as shown in Figure 6(b), a spurious RTO during a DUPACK series can only lead to unnecessary reduction of the congestion window and slow start threshold, but does not produce a series of spurious RTOs.

Recommendation: TCP SHOULD ignore DUPACKs for the oldest outstanding segment after RTO.

5. Conclusions

A TCP connection can experience delay spikes due to various reasons like handovers, priority blocking, temporal link outages or route changes. We described the response of Tahoe, Reno, New Reno and SACK TCP to a spurious timeout resulting from a delay spike. We have studied the behavior of the retransmit timer and ways to treat a DUPACK series using the NS simulator. The resulting summary of recommendations is shown in Table 1.

Acknowledgements

Many thanks to Reiner Ludwig, Mark Allman, and Sally Floyd for discussions on the contents of this document and to Timo Alanko for supporting this work.

References

[AP99] M. Allman and V. Paxson, On Estimating End-to-End Network Path Properties, ACM SIGCOMM '99, September 1999, Cambridge, MA.

[BA01a] E. Blanton, M. Allman. A Conservative SACK-based Loss Recovery Algorithm for TCP. Internet-Draft draft-allman-tcp-sack-07.txt, July 2001, work in progress.

[BA01b] E. Blanton, M. Allman. Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions, August 2001, work in progress.

[BA01c] E. Blanton, M. Allman, Adjusting the Duplicate ACK Threshold to Avoid Spurious Retransmits, work in progress, July 2001.

[Jac88] V. Jacobson, "Congestion Avoidance and Control", In proceedings of ACM SIGCOMM'88, 1988.

[GU01a] A. Gurtov, Effect of Delays on TCP Performance, In Proceedings of IFIP Personal Wireless Communications, August 2001.

[GU01b] A. Gurtov, Traces of TCP connections experiencing a delay spike,
http://www.cs.helsinki.fi/u/gurtov/tcp/, November 2001.

[RFC1323] V. Jacobson, R. Braden, D. Borman, TCP Extensions for High Performance, RFC 1323, May 1992.

[RFC2018] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowledgement Options, RFC 2018, October 1996.

[RFC2119] S. Bradner, Key words for use in RFCs to Indicate Requirement Levels, RFC 2119, March 1997.

[RFC2026] S. Bradner. The Internet Standards Process – Revision 3, RFC 2026, October 1996

[RFC2581] M. Allman, V. Paxson, W. Stevens, TCP Congestion Control, RFC 2581, April 1999.

[RFC2582] S. Floyd and T. Henderson. The NewReno modification to TCP's fast recovery algorithm. IETF RFC 2582, April 1999.

[RFC2883] S. Floyd, J. Mahdavi, M. Mathis, M. Podolsky, A. Romanow, An Extension to the Selective Acknowledgement (SACK) Option for TCP, RFC 2883, July 2000.

[RFC2988] V. Paxson, M. Allman, Computing TCP's Retransmission Timer, RFC 2988, November 2000.

[RFC3042] Allman, M., Balakrishnan, H. and S. Floyd, Enhancing TCP's Loss Recovery Using Limited Transmit, RFC 3042, January, 2001.

[RFC3155] S.Dawkins, G. Montenegro, M. Kojo, V. Magret, N. Vaidya. End-to-end Performance Implications of Links with Errors, RFC3155, August 2001.

[KP87] P. Karn, C. Partridge, Improving Round-Trip Time Estimates in Reliable Transport Protocols, In Proceedings of ACM SIGCOMM 87.

[LK00] R. Ludwig, R. H. Katz, The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions, ACM Computer Communication Review, Vol. 30, No. 1, January 2000.

[LU99] R. Ludwig, B. Rathonyi, A. Konrad, K. Oden, and A. Joseph. Multi-layer tracing of TCP over a reliable wireless link. In Proceedings of the ACM SIGMETRICS, May 1999.

[LG01] R. Ludwig, A. Gurtov, Responding to Spurious Timeouts in TCP, work in progress, November 2001.

[LU01a] R. Ludwig, TCP Retransmit (RXT) Flag, work in progress, November 2001.

[LU01b] R. Ludwig, The Eifel Algorithm for TCP, work in progress, November 2001.

[NS] ISI, Network Simulator 2, http://www.isi.edu/nsnam/ns