Practices for TCP Senders in the Face of Segment Reordering

Abstract

This document outlines actions a TCP sender can take after determining that a spurious fast retransmission has been sent due to packet reordering. The document outlines the method a TCP connection can use to "undo" needless changes made to the congestion control state. In addition, several methods to help prevent future fast retransmits are outlined.

Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1 Introduction

As outlined in [RFC793] TCP receivers always indicate the next in-order byte of data they expect to receive in the acknowledgments (ACKs) they send. Therefore, if a segment arrives that does not include the next expected byte of data, the receiver generates a
duplicate ACK. This situation can arise from either packet loss or from packet reordering in the network.

Meanwhile, TCP’s fast retransmit algorithm [Jac88, RFC2581] uses the arrival of three duplicate ACKs as an indication that the network lost the packet containing the indicated data byte. Upon detection of a lost segment via fast retransmit the sender resends the given data segment before the expiration of the retransmission timer (RTO). The sender waits for three duplicate ACKs in an attempt to distinguish between actual packet loss and small amounts of packet reordering.

Several Internet measurement studies have shown that waiting for the arrival of three duplicate ACKs may not be sufficient to distinguish between packet loss and packet reordering over some network paths [Pax97, BPS99]. Therefore, in some cases a TCP sender may send a spurious fast retransmit. This spurious retransmit wastes network resources on needless work, as well as causing the TCP sender to infer that the network is experiencing congestion and thus the sender should reduce its sending rate. [BA02a] shows the performance impact of various levels of reordering on TCP connections.

This document offers several experimental strategies for mitigating the impact of spurious fast retransmits. The first mitigation is to allow the sender to revert to its previous sending rate when spurious retransmissions are detected. The second mitigation is to dynamically vary the number of duplicate ACKs required to trigger fast retransmit in an attempt to prevent spurious retransmissions.

Explicitly out of scope for this document is the subject of spurious RTOs. This subject is covered by others (as outlined in section 3 on related work).

We hope the community will experiment with the mechanisms presented in this document and provide their experience and input to a future decision on which mechanisms (if any) may be appropriate to include as part of the TCP’s standard congestion control algorithms.

This document is organized as follows. Section 2 offers definitions. Section 3 outlines related work. Section 4 details how a TCP sender can revert to a previous congestion control state after a spurious retransmit has been detected. Section 5 outlines the methods for dynamically adjusting the number of duplicate ACKs required to trigger fast retransmit. Section 6 discusses interactions between adjusting the duplicate ACK threshold and the RTO. Section 7 sketches future work in this area. Section 8 outlines security considerations.

2 Definitions

It is assumed that the reader is familiar with the congestion control concepts and terminology for TCP from [RFC2581].
In addition, the following terms will be used:

“dupthresh” represents the number of duplicate ACKs required to trigger a fast retransmit. [RFC2581] defines dupthresh to be 3 duplicate ACKs.

3 Related Work

[Editor’s note: Additional related work outlined in [SL02] will be included in a later version of this document.]

A number of documents have been written on the subject of spurious retransmits. In this space, there are three categories of mechanisms:

(1) Mechanisms for detecting spurious retransmits.

The changes specified in this document require a mechanism to detect when a TCP sender has needlessly retransmitted. We do not specify any one algorithm that must be used with our mitigation strategies because the exact method employed does not have a direct impact on this specification. (That is not to say that the choice of detection mechanism is immaterial in a larger sense.)

[BA02b] outlines a scheme that uses the DSACK TCP option [RFC2883] to detect spurious retransmits.

[LM02] outlines the Eifel algorithm that uses the TCP timestamp option [RFC1323] to detect unneeded retransmissions. (An additional variant of the Eifel algorithm uses reserved bits from the TCP header rather than timestamps [LK00].)

[SK02] outlines the F-RTO algorithm that uses a slight modification to TCP’s sending pattern after a retransmission timeout (RTO) and the resulting ACKs to detect spurious RTOs.

(2) Mechanisms for undoing needless changes to congestion control state.

Section 4 of this document specifies the appropriate method for reverting to a previous set of congestion control parameters after reliably detecting a needless retransmit (and in the presence of no other congestion signals). The scheme specified in this document draws on the spirit outlined in [RFC2883].

[LG02] outlines the response to spurious retransmits (both spurious fast retransmits and spurious timeouts) after detection by the Eifel detection algorithm [LM02]. The mechanism for undoing needless changes to the congestion control state of the connection is similar to the mechanism we outline in section 4.

(3) Mechanisms for making TCP more tolerant to reordering events such that future spurious retransmits are avoided.
[LG02] outlines one particular method for adjusting dupthresh in an attempt to prevent future needless fast retransmissions. (In addition, [LG02] provides a method for adjusting the RTO after a spurious RTO.)

[ZKFP02] outlines a scheme whereby the TCP sender's SACK scoreboard data structure is annotated with information about packet reordering in an attempt to adjust the dupthresh to prevent packet reordering from causing needless retransmits.

Finally, section 5 of this document outlines several schemes to avoid needlessly evoking the fast retransmit algorithm. The mechanisms in section 5 are based on [BA02a]. Each of the above proposals has its pros and cons. An in-depth analysis of each proposal is beyond the scope of this document.

4 Reverting Congestion Control State

[RFC2883, LK00, BA02a] all mention the possibility of reverting congestion control state when a sender determines that the congestion window was reduced unnecessarily due to a spurious retransmission. This document specifies that a sender that reliably detects a spurious retransmission MAY "undo" the needless congestion control state changes. If the sender wishes to undo needless changes it MUST use the following algorithm:

1. When retransmitting a segment and before reducing cwnd or ssthresh, save the value of cwnd as cwnd_prev and the value of ssthresh as ssthresh_prev.

2. Upon the detection of a spurious retransmit, set ssthresh to cwnd_prev.

   This causes a 1 RTT slow start back to the cwnd in use before the sender needlessly changed the congestion control state. This rule causes a smooth increase of cwnd, whereas simply setting cwnd to cwnd_prev upon detection of a spurious retransmit would cause the transmission of a potentially large burst.

3. Optional. When cwnd reaches ssthresh set ssthresh to ssthresh_prev.

   This rule allows a TCP that is using slow start when a spurious retransmission is detected to resume that slow start after detection. Without this rule the cwnd would be reverted to its previous value, however further cwnd growth would be limited to the linear growth of congestion avoidance.

If a sender uses the above algorithm, one of the algorithms specified in the next section MUST be used in an attempt to prevent further needless retransmissions.
5 Preventing Spurious Fast Retransmits

Undoing congestion control changes caused by reordering events increases the overall performance of a TCP connection [BA02a]. However, [BA02a] shows that when the steps in section 4 are taken alone the number of unnecessary retransmits (and, therefore, wasted network resources) still remains significantly high. Therefore, further performance gains and network resource savings can be realized if we can prevent subsequent spurious fast retransmits. Simulations in [BA02a] show that a strategy of reversing erroneous congestion control decisions, coupled with a fast retransmit mitigation scheme, achieves good performance (within 1-2% of a transfer that experiences no reordering) while reducing the number of unnecessary retransmits by roughly a factor of 6.

This document purposes several options that deal with postponing the fast retransmit algorithm to allow the reordered segments to be acknowledged by the receiver. This is achieved by increasing the duplicate acknowledgment threshold, or directly delaying fast retransmit by an increment of time when a spurious retransmission has occurred previously.

Regardless of which approach is taken to avoid spurious retransmits, caution should be taken in their implementation. For example, more aggressive manipulations of dupthresh can result in faster convergence in the face of a connection facing persistent reordering, but may increase the chance of relying on the RTO to recover from real loss. The RTO is often lengthy and cuts cwnd to one segment and invokes slow start.

5.1 Increase dupthresh By K

A TCP sender that implements this strategy MUST increase dupthresh by some small constant, K, each time a spurious fast retransmit is detected. In [BA02a] K is set to increment dupthresh by one after a spurious retransmission is detected.

This method requires minimal state to implement, but a conservative increment of K may be slow to converge in connections where persistent large reordering events are present. In connections where the reordering event lengths are varied convergence becomes problematical at best.

5.2 Increase dupthresh Based On Reordering Length

A TCP sender that implements this strategy MUST increase dupthresh based on the length of the most recent reordering event that causes a spurious fast retransmit. With this method a TCP sender determines the number of duplicate ACKs, C, that would have disambiguated the reordering event from the perceived loss. C is then averaged with the current value of dupthresh, per

\[ \text{dupthresh} = \frac{(C + \text{dupthresh})}{2} \]  

(1)
If the new value for dupthresh is less than or equal to its predecessor, dupthresh SHOULD be incremented by one.

This method allows for a varied growth of dupthresh, accommodating the varied reordering event lengths mentioned in the previous option. Unfortunately, reordering events significantly longer than the connection’s average reordering events may result in skewing dupthresh to an overly optimistic value, preventing fast retransmit before the RTO timer expires.

5.3 Delay of Fast Retransmit

As proposed in [Pax97], the fast retransmit can be delayed by some small amount of time after dupthresh duplicate ACKs have been received in an effort to distinguish between loss and reordering. A sender that implements this strategy MUST track the duration in time, T, of the reordering event from the reception of the first duplicate ACK to the first cumulative or partial ACK. This represents the time necessary to disambiguate the longest reordering event in the current connection from its perceived loss. The sender MUST then, on subsequent perceived loss events, wait T seconds after the third duplicate acknowledgment for subsequent reordering/loss events before invoking the fast retransmit algorithm. Should a cumulative ACK be received before this timer expires, the sender MUST NOT invoke fast retransmit.

Similar in approach to the scheme sketched in section 5.2, this method considers the length of the reordering event. In using the passage of time, as opposed to counting duplicate ACKs, the strategy is more robust where ACK loss is present. Unfortunately this potential advantage comes at a cost of requiring more overhead than in the previously outlined methods in the form of an additional timer for each TCP connection.

5.4 Average Duplicate ACK Tracking

In an attempt to use a connection’s history while still being adaptive to reordering events which are uncharacteristically divergent, an Exponentially Weighted Moving Average (EWMA) is tried in [BA02a]. The EWMA gives the values needed to mitigate recent spurious fast retransmits a higher weight over the values used in previous fast retransmit mitigation attempts in the connections history. Unfortunately, initial results for this mechanism yielded little or no advantages over TCP variants which did not use any mechanisms to mitigate spurious fast retransmits. Therefore, we do not recommend using an EWMA scheme in this document, but rather offer it as a future research area.

6 Preventing Spurious Timeouts

In section 5, we outline several methods to increase dupthresh, or insert delay before fast retransmit, in hopes of maintaining cwnd to mitigate TCP performance issues in the face of reordering. Regardless of the method used, since reordering events may vary in
length and ACK loss will effect the number of duplicate ACKs returned to the sender there exists a possibility that dupthresh will be set to an overly optimistic value. In this case the RTO will expire before enough duplicate ACKs are received for fast retransmit. In addition to the often lengthy timeout imposed by the RTO [RFC2988] the TCP sender will reduce cwnd to one segment and restart transmission with slow start. This document sketches several additional methods for coping with RTOs caused by an overly optimistic dupthresh in the next subsections. In addition, the outline of future work below (section 7) offers some thoughts in this area.

6.1 Reducing dupthresh

In addition to the performance issues involved with the expiration of the RTO, the sender should consider an RTO as an indication that the current dupthresh is no longer a valid estimate of the reordering conditions in the network. The TCP sender MUST reset dupthresh to the default value of from [RFC2581] (currently 3). This conservative approach guards against subsequent RTOs stemming from an optimistic dupthresh value, but may allow spurious retransmissions to resume until the sender’s dupthresh estimate re-converges.

6.2 Bounding dupthresh

This document specifies that the dupthresh MUST never be larger than cwnd-1 segments. Furthermore, to be more conservative we RECOMMEND that dupthresh never be more than 90% of cwnd. This means that whenever cwnd is reduced in the course of a TCP transfer the stack must also ensure that dupthresh is likewise reduced, if necessary.

6.3 Extending Limited Transmit

A final strategy that MAY be used by a TCP sender in an attempt to prevent RTOs is to extend the limited transmit algorithm [RFC3042]. Limited transmit calls for a TCP sender to transmit one previously unsent data segment upon reception of each of the first two duplicate ACKs. This document allows that a TCP sender MAY transmit a previously unsent data segment on every second duplicate ACK received after the first two and before dupthresh is met. So, when used in conjunction with [RFC3042], a TCP sender with a dupthresh of 7 could transmit new data segments on duplicate ACKs 1, 2, 4, and 6.

This mechanism allows the TCP sender to sustain the ACK clock and possibly generate new duplicate ACKs that will help trigger fast retransmit and prevent an RTO. The mechanism is conservative because in the case when loss is the cause of the duplicate ACKs (rather than reordering) the rate the TCP sender is transmitting new segments into the network is gradually being reduced by sending only on every second duplicate ACK.

This mechanism is similar to the rate-halving proposals [].
7 Future Work

All the schemes for varying dupthresh in this document involve increasing the number of duplicate ACKs required to trigger fast retransmit. While we provide one scheme to reduce dupthresh (collapsing back to the default value after an RTO) [ZKFP02] shows that schemes that offer both dupthresh increase and decrease offer more performance benefit. One area for future work is to extend the strategies outlined in section 5 to allow for decreasing dupthresh.

8 Security Considerations

Reversion of congestion control state has no obvious security considerations given an accurate signal that indicates a needless retransmit. Any security considerations pertaining to the method of determining that a fast retransmit was spurious are directly applicable to this document.

The adjustment of dupthresh may be vulnerable to any third party who wishes to illegitimately gain network resources and/or cause a denial of service between the sender and receiver. The vulnerability exists when a third party sends fictitious duplicate acknowledgments to the sender with a sequence number within the sender’s current cwnd. As a result, the sender may needlessly retransmit and also grow dupthresh to an overly optimistic value, resulting in future RTOs. Such RTOs cause the sender to drop cwnd to 1 segment, relinquishing network resources at the cost of the connection’s performance.

Acknowledgments

The authors would like to thank Sally Floyd, Reiner Ludwig, Vern Paxson and Randall Stewart for spurring this work through their previous work for providing input during the formative stages of this draft.

References


Authors’ Addresses:

Ethan Blanton
Purdue University Computer Sciences
1398 Computer Science Building
West Lafayette, IN 47907
eblanton@cs.purdue.edu
Robert Dimond  
Verizon FNS/NASA Glenn Research Center  
Lewis Field  
21000 Brookpark Rd. MS 142-1  
Cleveland, OH 44135  
Phone: 216-433-8468  
bdimond@grc.nasa.gov

Mark Allman  
BBN Technologies/NASA Glenn Research Center  
Lewis Field  
21000 Brookpark Rd. MS 54-5  
Cleveland, OH 44135  
Phone: 216-433-6586  
Fax: 216-433-8705  
mallman@bbn.com  
http://roland.grc.nasa.gov/~mallman