Internet Engineering Task Force (IETF)
Request for Comments: 6675
Obsoletes: 3517
Category: Standards Track
ISSN: 2070-1721

E. Blanton
Purdue University
M. Allman
ICSI
L. Wang
Juniper Networks
I. Jarvinen
M. Kojo
University of Helsinki
Y. Nishida
WIDE Project

August 2012

A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP

Abstract

This document presents a conservative loss recovery algorithm for TCP that is based on the use of the selective acknowledgment (SACK) TCP option. The algorithm presented in this document conforms to the spirit of the current congestion control specification (RFC 5681), but allows TCP senders to recover more effectively when multiple segments are lost from a single flight of data. This document obsoletes RFC 3517 and describes changes from it.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6675.

1. Introduction

This document presents a conservative loss recovery algorithm for TCP that is based on the use of the selective acknowledgment (SACK) TCP option. While the TCP SACK option [RFC2018] is being steadily deployed in the Internet [All00], there is evidence that hosts are not using the SACK information when making retransmission and congestion control decisions [PF01]. The goal of this document is to outline one straightforward method for TCP implementations to use SACK information to increase performance.

[RFC5681] allows advanced loss recovery algorithms to be used by TCP [RFC793] provided that they follow the spirit of TCP's congestion control algorithms [RFC5681] [RFC2914]. [RFC6582] outlines one such advanced recovery algorithm called NewReno. This document outlines a loss recovery algorithm that uses the SACK TCP option [RFC2018] to enhance TCP's loss recovery. The algorithm outlined in this document, heavily based on the algorithm detailed in [FF96], is a conservative replacement of the fast recovery algorithm [Jac90] [RFC5681]. The algorithm specified in this document is a straightforward SACK-based loss recovery strategy that follows the guidelines set in [RFC5681] and can safely be used in TCP implementations. Alternate SACK-based loss recovery methods can be used in TCP as implementers see fit (as long as the alternate algorithms follow the guidelines provided in [RFC5681]). Please note, however, that the SACK-based decisions in this document (such as what segments are to be sent at what time) are largely decoupled from the congestion control algorithms, and as such can be treated as separate issues if so desired.

This document represents a revision of [RFC3517] to address several situations that are not handled explicitly in that document. A

summary of the changes between this document and [RFC3517] can be found in Section 9.

2. Definitions

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 BCP 14, RFC 2119 [RFC2119].

The reader is expected to be familiar with the definitions given in [RFC5681].

The reader is assumed to be familiar with selective acknowledgments as specified in [RFC2018].

For the purposes of explaining the SACK-based loss recovery algorithm, we define six variables that a TCP sender stores:

"HighACK" is the sequence number of the highest byte of data that has been cumulatively ACKed at a given point.

"HighData" is the highest sequence number transmitted at a given point.

"HighRxt" is the highest sequence number which has been retransmitted during the current loss recovery phase.

"RescueRxt" is the highest sequence number which has been optimistically retransmitted to prevent stalling of the ACK clock when there is loss at the end of the window and no new data is available for transmission.

"Pipe" is a sender's estimate of the number of bytes outstanding in the network. This is used during recovery for limiting the sender's sending rate. The pipe variable allows TCP to use fundamentally different congestion control than the algorithm specified in [RFC5681]. The congestion control algorithm using the pipe estimate is often referred to as the "pipe algorithm".

"DupAcks" is the number of duplicate acknowledgments received since the last cumulative acknowledgment.

For the purposes of this specification, we define a "duplicate acknowledgment" as a segment that arrives carrying a SACK block that identifies previously unacknowledged and un-SACKed octets between HighACK and HighData. Note that an ACK which carries new SACK data is counted as a duplicate acknowledgment under this definition even

if it carries new data, changes the advertised window, or moves the cumulative acknowledgment point, which is different from the definition of duplicate acknowledgment in [RFC5681].

We define a variable "DupThresh" that holds the number of duplicate acknowledgments required to trigger a retransmission. Per [RFC5681], this threshold is defined to be 3 duplicate acknowledgments. However, implementers should consult any updates to [RFC5681] to determine the current value for DupThresh (or method for determining its value).

Finally, a range of sequence numbers [A,B] is said to "cover" sequence number S if A <= S <= B.

3. Keeping Track of SACK Information

For a TCP sender to implement the algorithm defined in the next section, it must keep a data structure to store incoming selective acknowledgment information on a per connection basis. Such a data structure is commonly called the "scoreboard". The specifics of the scoreboard data structure are out of scope for this document (as long as the implementation can perform all functions required by this specification).

Note that this document refers to keeping account of (marking) individual octets of data transferred across a TCP connection. A real-world implementation of the scoreboard would likely prefer to manage this data as sequence number ranges. The algorithms presented here allow this, but require the ability to mark arbitrary sequence number ranges as having been selectively acknowledged.

Finally, note that the algorithm in this document assumes a sender that is not keeping track of segment boundaries after transmitting a segment. It is possible that there is a more refined and precise algorithm available to a sender that keeps this extra state than the algorithm presented herein; however, we leave this as future work.

4. Processing and Acting Upon SACK Information

This section describes a specific structure and control flow for implementing the TCP behavior described by this standard. The behavior is what is standardized, and this particular collection of functions is the strongly recommended means of implementing that behavior, though other approaches to achieving that behavior are feasible.

The definition of Sender Maximum Segment Size (SMSS) used in this section is provided in [RFC5681].

For the purposes of the algorithm defined in this document, the scoreboard SHOULD implement the following functions:

Update ():

Given the information provided in an ACK, each octet that is cumulatively ACKed or SACKed should be marked accordingly in the scoreboard data structure, and the total number of octets SACKed should be recorded.

Note: SACK information is advisory and therefore SACKed data MUST NOT be removed from the TCP's retransmission buffer until the data is cumulatively acknowledged [RFC2018].

IsLost (SeqNum):

This routine returns whether the given sequence number is considered to be lost. The routine returns true when either DupThresh discontiguous SACKed sequences have arrived above 'SeqNum' or more than (DupThresh - 1) * SMSS bytes with sequence numbers greater than 'SeqNum' have been SACKed. Otherwise, the routine returns false.

SetPipe ():

This routine traverses the sequence space from HighACK to HighData and MUST set the "pipe" variable to an estimate of the number of octets that are currently in transit between the TCP sender and the TCP receiver. After initializing pipe to zero, the following steps are taken for each octet 'S1' in the sequence space between HighACK and HighData that has not been SACKed:

(a) If IsLost (S1) returns false:

Pipe is incremented by 1 octet.

The effect of this condition is that pipe is incremented for packets that have not been SACKed and have not been determined to have been lost (i.e., those segments that are still assumed to be in the network).

(b) If S1 <= HighRxt:

Pipe is incremented by 1 octet.

The effect of this condition is that pipe is incremented for the retransmission of the octet.

Note that octets retransmitted without being considered lost are counted twice by the above mechanism.

NextSeg ():

This routine uses the scoreboard data structure maintained by the Update() function to determine what to transmit based on the SACK information that has arrived from the data receiver (and hence been marked in the scoreboard). NextSeg () MUST return the sequence number range of the next segment that is to be transmitted, per the following rules:

(1) If there exists a smallest unSACKed sequence number 'S2' that meets the following three criteria for determining loss, the sequence range of one segment of up to SMSS octets starting with S2 MUST be returned.

(1.a) S2 is greater than HighRxt.

(1.b) S2 is less than the highest octet covered by any received SACK.

(1.c) IsLost (S2) returns true.

(2) If no sequence number 'S2' per rule (1) exists but there exists available unsent data and the receiver's advertised window allows, the sequence range of one segment of up to SMSS octets of previously unsent data starting with sequence number HighData+1 MUST be returned.

(3) If the conditions for rules (1) and (2) fail, but there exists an unSACKed sequence number 'S3' that meets the criteria for detecting loss given in steps (1.a) and (1.b) above (specifically excluding step (1.c)), then one segment of up to SMSS octets starting with S3 SHOULD be returned.

(4) If the conditions for (1), (2), and (3) fail, but there exists outstanding unSACKed data, we provide the opportunity for a single "rescue" retransmission per entry into loss recovery. If HighACK is greater than RescueRxt (or RescueRxt is undefined), then one segment of up to SMSS octets that MUST include the highest outstanding unSACKed sequence number SHOULD be returned, and RescueRxt set to RecoveryPoint. HighRxt MUST NOT be updated.

Note that rules (3) and (4) are a sort of retransmission "last resort". They allow for retransmission of sequence numbers even when the sender has less certainty a segment has been

lost than as with rule (1). Retransmitting segments via rule (3) and (4) will help sustain the TCP's ACK clock and therefore can potentially help avoid retransmission timeouts. However, in sending these segments, the sender has two copies of the same data considered to be in the network (and also in the pipe estimate, in the case of (3)). When an ACK or SACK arrives covering this retransmitted segment, the sender cannot be sure exactly how much data left the network (one of the two transmissions of the packet or both transmissions of the packet). Therefore, the sender may underestimate pipe by considering both segments to have left the network when it is possible that only one of the two has.

(5) If the conditions for each of (1), (2), (3), and (4) are not met, then NextSeg () MUST indicate failure, and no segment is returned.

Note: The SACK-based loss recovery algorithm outlined in this document requires more computational resources than previous TCP loss recovery strategies. However, we believe the scoreboard data structure can be implemented in a reasonably efficient manner (both in terms of computation complexity and memory usage) in most TCP implementations.

5. Algorithm Details

Upon the receipt of any ACK containing SACK information, the scoreboard MUST be updated via the Update () routine.

If the incoming ACK is a cumulative acknowledgment, the TCP MUST reset DupAcks to zero.

If the incoming ACK is a duplicate acknowledgment per the definition in Section 2 (regardless of its status as a cumulative acknowledgment), and the TCP is not currently in loss recovery, the TCP MUST increase DupAcks by one and take the following steps:

(1) If DupAcks >= DupThresh, go to step (4).

Note: This check covers the case when a TCP receives SACK information for multiple segments smaller than SMSS, which can potentially prevent IsLost() (next step) from declaring a segment as lost.

(2) If DupAcks < DupThresh but IsLost (HighACK + 1) returns true -- indicating at least three segments have arrived above the current cumulative acknowledgment point, which is taken to indicate loss -- go to step (4).

(3) The TCP MAY transmit previously unsent data segments as per Limited Transmit [RFC5681], except that the number of octets which may be sent is governed by pipe and cwnd as follows:

(3.1) Set HighRxt to HighACK.

(3.2) Run SetPipe ().

(3.3) If (cwnd - pipe) >= 1 SMSS, there exists previously unsent data, and the receiver's advertised window allows, transmit up to 1 SMSS of data starting with the octet HighData+1 and update HighData to reflect this transmission, then return to (3.2).

(3.4) Terminate processing of this ACK.

(4) Invoke fast retransmit and enter loss recovery as follows:

(4.1) RecoveryPoint = HighData

When the TCP sender receives a cumulative ACK for this data octet, the loss recovery phase is terminated.

(4.2) ssthresh = cwnd = (FlightSize / 2)

The congestion window (cwnd) and slow start threshold (ssthresh) are reduced to half of FlightSize per [RFC5681]. Additionally, note that [RFC5681] requires that any segments sent as part of the Limited Transmit mechanism not be counted in FlightSize for the purpose of the above equation.

(4.3) Retransmit the first data segment presumed dropped -- the segment starting with sequence number HighACK + 1. To prevent repeated retransmission of the same data or a premature rescue retransmission, set both HighRxt and RescueRxt to the highest sequence number in the retransmitted segment.

(4.4) Run SetPipe ()

Set a "pipe" variable to the number of outstanding octets currently "in the pipe"; this is the data which has been sent by the TCP sender but for which no cumulative or selective acknowledgment has been received and the data has not been determined to have been dropped in the network. It is assumed that the data is still traversing the network path.

(4.5) In order to take advantage of potential additional available cwnd, proceed to step (C) below.

Once a TCP is in the loss recovery phase, the following procedure MUST be used for each arriving ACK:

(A) An incoming cumulative ACK for a sequence number greater than RecoveryPoint signals the end of loss recovery, and the loss recovery phase MUST be terminated. Any information contained in the scoreboard for sequence numbers greater than the new value of HighACK SHOULD NOT be cleared when leaving the loss recovery phase.

(B) Upon receipt of an ACK that does not cover RecoveryPoint, the following actions MUST be taken:

(B.1) Use Update () to record the new SACK information conveyed by the incoming ACK.

(B.2) Use SetPipe () to re-calculate the number of octets still in the network.

(C) If cwnd - pipe >= 1 SMSS, the sender SHOULD transmit one or more segments as follows:

(C.1) The scoreboard MUST be queried via NextSeg () for the sequence number range of the next segment to transmit (if any), and the given segment sent. If NextSeg () returns failure (no data to send), return without sending anything (i.e., terminate steps C.1 -- C.5).

(C.2) If any of the data octets sent in (C.1) are below HighData, HighRxt MUST be set to the highest sequence number of the retransmitted segment unless NextSeg () rule (4) was invoked for this retransmission.

(C.3) If any of the data octets sent in (C.1) are above HighData, HighData must be updated to reflect the transmission of previously unsent data.

(C.4) The estimate of the amount of data outstanding in the network must be updated by incrementing pipe by the number of octets transmitted in (C.1).

(C.5) If cwnd - pipe >= 1 SMSS, return to (C.1)

Note that steps (A) and (C) can potentially send a burst of back-to-back segments into the network if the incoming cumulative acknowledgment is for more than SMSS octets of data, or if incoming SACK blocks indicate that more than SMSS octets of data have been lost in the second half of the window.

5.1. Retransmission Timeouts

In order to avoid memory deadlocks, the TCP receiver is allowed to discard data that has already been selectively acknowledged. As a result, [RFC2018] suggests that a TCP sender SHOULD expunge the SACK information gathered from a receiver upon a retransmission timeout (RTO) "since the timeout might indicate that the data receiver has reneged." Additionally, a TCP sender MUST "ignore prior SACK information in determining which data to retransmit." However, since the publication of [RFC2018], this has come to be viewed by some as too strong. It has been suggested that, as long as robust tests for reneging are present, an implementation can retain and use SACK information across a timeout event [Errata1610]. While this document does not change the specification in [RFC2018], we note that implementers should consult any updates to [RFC2018] on this subject. Further, a SACK TCP sender SHOULD utilize all SACK information made available during the loss recovery following an RTO.

If an RTO occurs during loss recovery as specified in this document, RecoveryPoint MUST be set to HighData. Further, the new value of RecoveryPoint MUST be preserved and the loss recovery algorithm outlined in this document MUST be terminated. In addition, a new recovery phase (as described in Section 5) MUST NOT be initiated until HighACK is greater than or equal to the new value of RecoveryPoint.

As described in Sections 4 and 5, Update () SHOULD continue to be used appropriately upon receipt of ACKs. This will allow the recovery period after an RTO to benefit from all available information provided by the receiver, even if SACK information was expunged due to the RTO.

If there are segments missing from the receiver’s buffer following processing of the retransmitted segment, the corresponding ACK will contain SACK information. In this case, a TCP sender SHOULD use this SACK information when determining what data should be sent in each segment following an RTO. The exact algorithm for this selection is not specified in this document (specifically NextSeg () is inappropriate during loss recovery after an RTO). A relatively straightforward approach to "filling in" the sequence space reported as missing should be a reasonable approach.

6. Managing the RTO Timer

The standard TCP RTO estimator is defined in [RFC6298]. Due to the fact that the SACK algorithm in this document can have an impact on the behavior of the estimator, implementers may wish to consider how the timer is managed. [RFC6298] calls for the RTO timer to be re-armed each time an ACK arrives that advances the cumulative ACK point. Because the algorithm presented in this document can keep the ACK clock going through a fairly significant loss event (comparatively longer than the algorithm described in [RFC5681]), on some networks the loss event could last longer than the RTO. In this case the RTO timer would expire prematurely and a segment that need not be retransmitted would be resent.

Therefore, we give implementers the latitude to use the standard [RFC6298]-style RTO management or, optionally, a more careful variant that re-arms the RTO timer on each retransmission that is sent during recovery MAY be used. This provides a more conservative timer than specified in [RFC6298], and so may not always be an attractive alternative. However, in some cases it may prevent needless retransmissions, go-back-N transmission, and further reduction of the congestion window.

7. Research

The algorithm specified in this document is analyzed in [FF96], which shows that the above algorithm is effective in reducing transfer time over standard TCP Reno [RFC5681] when multiple segments are dropped from a window of data (especially as the number of drops increases). [AHKO97] shows that the algorithm defined in this document can greatly improve throughput in connections traversing satellite channels.

8. Security Considerations

The algorithm presented in this paper shares security considerations with [RFC5681]. A key difference is that an algorithm based on SACKs is more robust against attackers forging duplicate ACKs to force the TCP sender to reduce cwnd. With SACKs, TCP senders have an additional check on whether or not a particular ACK is legitimate. While not fool-proof, SACK does provide some amount of protection in this area.

Similarly, [CPNI309] sketches a variant of a blind attack [RFC5961] whereby an attacker can spoof out-of-window data to a TCP endpoint, causing it to respond to the legitimate peer with a duplicate cumulative ACK, per [RFC793]. Adding a SACK-based requirement to trigger loss recovery effectively mitigates this attack, as the

duplicate ACKs caused by out-of-window segments will not contain SACK information indicating reception of previously un-SACKED in-window data.

9. Changes Relative to RFC 3517

The state variable "DupAcks" has been added to the list of variables maintained by this algorithm, and its usage specified.

The function IsLost () has been modified to require that more than (DupThresh - 1) * SMSS octets have been SACKed above a given sequence number as indication that it is lost, which is changed from the minimum requirement of (DupThresh * SMSS) described in [RFC3517]. This retains the requirement that at least three segments following the sequence number in question have been SACKed, while improving detection in the event that the sender has outstanding segments which are smaller than SMSS.

The definition of a "duplicate acknowledgment" has been modified to utilize the SACK information in detecting loss. Duplicate cumulative acknowledgments can be caused by either loss or reordering in the network. To disambiguate loss and reordering, TCP's fast retransmit algorithm [RFC5681] waits until three duplicate ACKs arrive to trigger loss recovery. This notion was then the basis for the algorithm specified in [RFC3517]. However, with SACK information there is no need to rely blindly on the cumulative acknowledgment field. We can leverage the additional information present in the SACK blocks to understand that three segments lying above a gap in the sequence space have arrived at the receiver, and can use this understanding to trigger loss recovery. This notion was used in [RFC3517] during loss recovery, and the change in this document is that the notion is also used to enter a loss recovery phase.

The state variable "RescueRxt" has been added to the list of variables maintained by the algorithm, and its usage specified. This variable is used to allow for one extra retransmission per entry into loss recovery, in order to keep the ACK clock going under certain circumstances involving loss at the end of the window. This mechanism allows for no more than one segment of no larger than 1 SMSS to be optimistically retransmitted per loss recovery.

Rule (3) of NextSeg() has been changed from MAY to SHOULD, to appropriately reflect the opinion of the authors and working group that it should be left in, rather than out, if an implementor does not have a compelling reason to do otherwise.

10. Acknowledgments

The authors wish to thank Sally Floyd for encouraging [RFC3517] and commenting on early drafts. The algorithm described in this document is loosely based on an algorithm outlined by Kevin Fall and Sally Floyd in [FF96], although the authors of this document assume responsibility for any mistakes in the above text.

[RFC3517] was co-authored by Kevin Fall, who provided crucial input to that document and hence this follow-on work.

Murali Bashyam, Ken Calvert, Tom Henderson, Reiner Ludwig, Jamshid Mahdavi, Matt Mathis, Shawn Ostermann, Vern Paxson, and Venkat Venkatsubra provided valuable feedback on earlier versions of this document.

We thank Matt Mathis and Jamshid Mahdavi for implementing the scoreboard in ns and hence guiding our thinking in keeping track of SACK state.

The first author would like to thank Ohio University and the Ohio University Internetworking Research Group for supporting the bulk of his work on RFC 3517, from which this document is derived.

11. References

11.1. Normative References

[RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

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

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

11.2. Informative References

[AHKO97] Mark Allman, Chris Hayes, Hans Kruse, Shawn Ostermann, "TCP Performance Over Satellite Links", Proceedings of the Fifth International Conference on Telecommunications Systems, Nashville, TN, March, 1997.

[All00] Mark Allman, "A Web Server's View of the Transport Layer", ACM Computer Communication Review, 30(5), October 2000.

[CPNI309] Fernando Gont, "Security Assessment of the Transmission Control Protocol (TCP)", CPNI Technical Note 3/2009, <http://www.gont.com.ar/papers/tn-03-09-security-assessment-TCP.pdf>, February 2009.

[Errata1610] RFC Errata, Errata ID 1610, RFC 2018, <http://www.rfc-editor.org>.

[FF96] Kevin Fall and Sally Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", Computer Communication Review, July 1996.

[Jac90] Van Jacobson, "Modified TCP Congestion Avoidance Algorithm", Technical Report, LBL, April 1990.

[PF01] Jitendra Padhye, Sally Floyd "Identifying the TCP Behavior of Web Servers", ACM SIGCOMM, August 2001.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 6582, April 2012.

[RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, September 2000.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, June 2011.

[RFC3517] Blanton, E., Allman, M., Fall, K., and L. Wang, "A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP", RFC 3517, April 2003.

[RFC5961] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, August 2010.

Authors' Addresses

Ethan Blanton
Purdue University Computer Sciences
305 N. University St.
West Lafayette, IN 47907
United States
EMail: elb@psg.com

Mark Allman
International Computer Science Institute
1947 Center St. Suite 600
Berkeley, CA 94704
United States
EMail: mallman@icir.org
http://www.icir.org/mallman

Lili Wang
Juniper Networks
10 Technology Park Drive
Westford, MA 01886
United States
EMail: liliw@juniper.net

Ilpo Jarvinen
University of Helsinki
P.O. Box 68
FI-00014 UNIVERSITY OF HELSINKI
Finland
EMail: ilpo.jarvinen@helsinki.fi

Markku Kojo
University of Helsinki
P.O. Box 68
FI-00014 UNIVERSITY OF HELSINKI
Finland
EMail: kojo@cs.helsinki.fi

Yoshifumi Nishida
WIDE Project
Endo 5322
Fujisawa, Kanagawa 252-8520
Japan
EMail: nishida@wide.ad.jp