Internet Engineering Task Force
INTERNET-DRAFT
draft-jarvinen-tcpm-sack-recovery-entry-01.txt
Intended status: Standards Track
Expires: February 2010

I. Jarvinen
M. Kojo
University of Helsinki

5 August 2009

Using TCP Selective Acknowledgement (SACK) Information to Determine Duplicate Acknowledgements for Loss Recovery Initiation

Status of this Memo

Copyright Notice

Abstract

This document describes a TCP sender algorithm to trigger loss recovery based on the information gathered on a SACK scoreboard instead of simply counting the number of arriving duplicate acknowledgements in the traditional way. The given algorithm is more robust to ACK losses, ACK reordering, missed duplicate acknowledgements due to delayed acknowledgements, and extra duplicate acknowledgements due to duplicated segments and out-of-window segments. The algorithm allows not only a timely initiation of TCP loss recovery but also reduces false fast retransmits. It has a low implementation cost on top of the SACK scoreboard defined in RFC 3517.

Table of Contents

TO BE DELETED BY THE RFC EDITOR UPON PUBLICATION:

Changes from draft-jarvinen-tcpm-sack-recovery-entry-00.txt

* TODO items embedded: Improvements with window update, clarify dupack counting

* Modified ACK reordering scenario in appendix, shows now a scenario where recovery is triggered in a more timely manner.

* IDnits

* Handle small segments case using duplicate ACKs counter paraller to the SACK blocks based detection.

* Add a placeholder for SACK splitting

* Mentioned FACK as some ideas are inherited from there

END OF SECTION TO BE DELETED.

1. Introduction

The Transmission Control Protocol (TCP) [RFC793] has two methods for triggering retransmissions. First, the TCP sender relies on incoming duplicate acknowledgement (ACKs) [RFC2581bis], indicating receipt of out-of-order segments at the TCP receiver. After receiving a required number of duplicate ACKs (usually three), the TCP sender retransmits the first unacknowledged segment and continues with a fast recovery algorithm such as Reno [RFC2581], NewReno [RFC3782] or SACK-based loss recovery [RFC3517]. Second, the TCP sender maintains a retransmission timer that triggers retransmission of segments, if the retransmission timer expires before the segments have been acknowledged.

While the conservative loss recovery algorithm defined in [RFC3517] takes full advantage of SACK information during a loss recovery, it does not consider the very same information during the pre-recovery detection phase. Instead, it simply counts the number of arriving duplicate ACKs and leans on the number of duplicate ACKs in deciding when to enter loss recovery. However, this traditional heuristics of simply counting the number of duplicate ACKs to trigger a loss recovery fails in several cases to determine correctly the actual number of valid out-of-order segments the receiver has successfully received. First, trusting on duplicate ACKs alone utterly fails to get hold of the whole picture in case of ACK losses and ACK reordering, resulting in delayed or missed initiation of fast

retransmit and fast recovery. Similarly, the delayed ACK mechanism tends to conceal the first duplicate ACK as the delayed cumulative ACK becomes combined with the first duplicate ACK when the first out-of-order segment arrives at the receiver (in case of an enlarged ACK ratio such as with ACK congestion control [FARI08], even more significant portion is affected). Second, segment duplication or out-of-window segments increase the risk of falsely triggering loss recovery as they trigger duplicate ACKs.

ADDME: window updates can be used to determine current state when other ACKs where lost, they would not even be dupacks (Thanks to corridor talk with Anna Brunstrom).

The algorithm specified in this document uses TCP Selective Acknowledgement Option [RFC2018] to determine duplicate ACKs and to trigger loss recovery based on the information gathered on the SACK scoreboard [RFC3517]. It works in the pre-recovery state giving a more accurate heurestic for determining the number of out-of-order segments arrived at the TCP receiver. The information gathered on the scoreboard reveals missing ACKs and allows detecting duplicate events. Therefore, the algorithm enables a timely triggering of Fast Retransmit. In addition, it allows the use of Limited Transmit regardless of lost ACKs and also in the cases where the SACK information is piggybacked to a cumulative ACK due to delayed ACKs. This, in turn, allows keeping ACK clock running more accurately.

This algorithm is close to what Linux TCP implementation has used for a very long time. A similar approach is briefly mentioned along ACK congestion control [FARI08] but as the usefulness of the algorithm in this document is more general and not limited to ACK congestion control we specify it separately. We also note that the defination of a duplicate acknowledgement already suggests that an incoming ACK can be considered as a duplicate ACK if it "contains previously unknown SACK information" [RFC2581bis]. While similarities with this algorithm and Forward Acknowledgement (FACK) [MM96] exist, they differ in how the quantity of data outstanding in the network is determined.

1.1. Conventions and 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 BCP 14, RFC 2119 [RFC2119] and indicate requirement levels for protocols.

1.2. Definitions

The reader is expected to be familiar with the definitions given in [RFC2581bis], [RFC2018], and [RFC3517].

2. Algorithm Details

In order to use this algorithm, a TCP sender MUST have TCP Selective Acknowledgement Option [RFC2018] enabled and negotiated for the TCP connection. A TCP sender MUST maintain SACK information in an approproate data structure such as scoreboard defined in [RFC3517]. This algorithm uses functions IsLost (SeqNum), Update(), and SetPipe () and variables DupThresh, HighData, HighRxt, Pipe, and RecoveryPoint, as defined in [RFC3517].

A TCP sender using this algorithm MUST take following steps:

1) Upon the receipt of any ACK containing SACK information:

   If no previous loss event has occurred on the connection OR RecoveryPoint is less than SND.UNA (the oldest unacknowledged sequence number [RFC793]), continue with the other steps of this algorithm. Otherwise, continue the ongoing loss recovery.

2) Update the scoreboard via the Update () function as outlined in [RFC3517].

3) Only if SACK blocks contained previously unknown in-window SACK information, count the acknowledgment as duplicate ACK [RFC2581bis]. TODO: clarify how to behave on cumulative ACKs and with non-dupacks.

4) Determinate if a loss recovery should be initiated:

   If IsLost(SND.UNA) returns false AND the sender has received less than DupThresh duplicate ACKs, goto step 5A. Otherwise goto step 5B.

5A) Invoke optional Limited Transmit:

   Set HighRxt to SND.UNA and run SetPipe(). The TCP sender MAY transmit previously unsent data segments according the guidelines of Limited Transmit [RFC3042], with the exception that the amount of octets that can be send is determined by Pipe and cwnd.

If cwnd - pipe >= 1 SMSS, the TCP sender can transmit one or more segments as follows:

Send Loop:

a) If available unsent data exists and the receiver's advertised window allows, transmit one segment of up to SMSS octets of previously unsent data starting with sequence number HighData+1 and update HighData to reflect the transmission of the data segment. Otherwise, exit Send Loop.

b) Run SetPipe() to re-calculate the number of outstanding octets in the network. If cwnd - pipe >= 1 SMSS, go to step a) of Send Loop. Otherwise, exit Send Loop.

5B) Invoke Fast Retransmit and enter loss recovery:

Initiate a loss recovery phase, per the fast retransmit algorithm outlined in [RFC2581] and continue with a fast recovery algorithm, such as the SACK-based loss recovery algorithm outlined in [RFC3517].

3. Discussion

In scenarios where no ACK losses nor reordering occur and the first acknowledgement with SACK information is not the ACK held due to delayed acknowledgements mechanism, the new SACK information with each duplicate ACK covers a single segment. In such a case, this algorithm will trigger loss recovery after three duplicate acknowledgements and will allow transmission of a single new segment using Limited Transmit on each acknowledgement. This is identical to the behavior that would occur without this algorithm (assuming DupThresh is 3 and that all segments are SMSS sized). This scenario together with other scenarios describing the behavior of the algorithm are depicted in Appendix A.

A set of potential issues to consider with the algorithm are discussed in the following.

3.1. Small Segment Sender

If a TCP sender is sending small segments (usually intentionally overriding Nagle algorithm [RFC896]), the IsLost(SND.UNA) used in step 4 of the algorithm might fail to detect the need for loss recovery on the third duplicate acknowledgement because not enough octets have been SACKed to cover DupThresh * SMSS bytes above

SND.UNA. If the SACKed octets are discontiguous (the second rule of IsLost()), the loss recovery is still triggered on time. Otherwise, the traditional duplicate ACKs algorithm needs to be used as a fallback. Step 3 and the latter condition of step 4 implement the traditional algorithm paraller to the SACK block based detection, however, it comes with a cost of lost robustness against ACK losses as expected.

Alternatively, a TCP sender that is able to discern segment boundaries accurately can consider full segments in IsLost() regardless of segment size. Therefore, such a TCP sender can avoid the problem with small segments using IsLost(SND.UNA) check alone which means that step 3 and the latter condition of step 4 are redundant and do not have to be implemented.

Note: the small segments problem is not unique to this algorithm but also the SACK-based loss recovery [RFC3517] encounters it because of how IsLost() is defined.

3.2. One Segment is Small

A variant of small segment sender case is the case where only one of the SACKed segments is smaller than SMSS (possible even with Nagle enabled). If TCP sender lacks ability to use the improved method by discerning segment boundaries but still wants robustness against ACK losses in this case, it MAY replace IsLost(SND.UNA) with test:

```
SACKed octets > SMSS * (DupThresh - 1)
```

3.3. SACK Capability Misbehavior

If the receiver represents such a SACK misbehavior that it advertizes SACK capability but never sends any SACK blocks when it should, this algorithm fails to enter loss recovery and retransmission timeout is required for recovery. However, such misbehavior does not allow SACK-based loss recovery [RFC3517] to work either, and a TCP sender will anyway require a timeout to recover.

3.4. Compatibility with Duplicate ACK based Loss Recovery Algorithms

This algorithm SHOULD NOT be used together with a fast recovery algorithm that determines the segments that have left the network based on the number of arriving duplicate acknowledgements (e.g., NewReno [RFC3782]), instead of the actual segments reported by SACK.

In presence of ACK reordering such an algorithm will count the delayed duplicate acknowledgements during the fast recovery algorithm as extra while detemining the number of packets that have left the network.

In general there should be very little reason to combine this algorithm with a loss recovery algorithm that is based on inferior, non-SACK based information only.

4. Security Considerations

A malicious TCP receiver may send false SACK information for sequence number ranges which it has not received in order to trigger Fast Retransmit sooner. Such behavior would only be useful when out-of-order segments have arrived because otherwise the flow undergoes a loss recovery with a window reduction. This kind of lying involves guessing which segments will arrive later. In case the guess was wrong, the performance of the flow is ruined because the TCP sender will need a retransmission timeout as it will not retransmit the segments until it assumes SACK reneging. On a successful guess the attacker is able to trigger the recovery slightly earlier. The later segments would have allowed reporting the very same regions with SACK anyway. Therefore, the gain from this attack is small, hardly justifiable considering the drastic effect of a misguess. Also, a similar attack can be made with the duplicate acknowledgment based algorithm (even if the new SACK information rule is applied) by sending false duplicate acknowledgements with false SACK ranges, and trivially without the new SACK information rule.

A variation of the lying attack discards reliability of the flow but as soon as the reliability is not a concern of the receiver, a number of simpler ways exist to attack TCP independently of this algorithm. Thus this algorithm is not considered to weaken TCP security properties against false information.

PLACEHOLDER: SACK splitting to make recovery to start sooner than it should or to trigger more segments (with less bandwidth in the opposite direction than using multiple duplicate ACKs).

5. IANA Considerations

This document has no actions for IANA.

6. Acknowledgements

Appendix

A. Scenarios

A.1. Basic Case

In this scenario no Delayed ACK, ACK losses, reordering or other "abnormal" behavior happens. For simplicity all the segments are SMSS sized.

Once the TCP receiver gets first out-of-order segment, it sends a duplicate ACK with SACK information about the received octets. The following two out-of-order segments trigger a duplicate ACK each, with the corresponding range SACKed in addition to the previously know information. The sender gets those duplicate ACKs in-order, each of them will SACK a new previously unknown segment.

This algorithm triggers loss recovery on third duplicate ACK because IsLost returns true as DupThresh * SMSS bytes became SACKed above the SND.UNA on the same acknowledgement, thus the behavior is identical to that of a sender which is using duplicate acknowledgments. If Limited Transmit is in use, two first duplicate ACKs allow a single segment to be sent with either of the algorithms (Pipe is decremented by SMSS by the SACKed octets per ACK allowing SMSS worth of new octets).

ACK Received	Transmitted Segment	Received Segment	ACK Sent (Including SACK Blocks)
1000			
	3000-3499	3000-3499	(delayed ACK)
	3500-3999	3500-3999	4000
2000			
	4000-4499	(dropped)	
	4500-4999	4500-4999	4000, SACK=4500-5000
3000			
	5000-5499	5000-5499	4000, SACK=4500-5500
	5500-5999	5500-5999	4000, SACK=4500-6000
4000			
	6000-6499	6000-6499	4000, SACK=4500-6500
	6500-6999	6500-6999	4000, SACK=4500-7000
4000, SACK=4500-5000			
	7000-7499	7000-7499	4000, SACK=4500-7500

```
4000, SACK=4500-5500
                7500-7999       7500-7999   4000, SACK=4500-8000
4000, SACK=4500-6000
                4000-4499       4000-4499   8000
4000, SACK=4500-6500
```

A.2. Delayed ACK

A basic case with delayed ACK send the first ACK with SACK information but since the previous ACK was sent with a lower sequence number because an acknowledgment is held by delayed ACK, the sender will not considered it as duplicate ACK. Because the segment contains SACK information that is identical to the basic case, the sender can use Limited Transmit with the same segments as in the basic case and will start loss recovery at the third acknowledgment, i.e., with the second duplicate acknowledgment. In the same situation the duplicate ACK based sender will have to wait for one more duplicate ACK to arrive to do the same as the first acknowledgment is fully "wasted".

Technically an acknowledgement with a sequence number higher than what was previously acknowledged is not a duplicate acknowledgement but a presence of the SACK block tells another story revealing the receiver which used delayed ACK, and thus the missing duplicate acknowledgement in between. The response of a TCP sender taking advantage of such inferred duplicate acknowledgements is well within the guidelines of packet conservation principle [Jac88] as it still sends only when segments have left the network.

```
ACK             Transmitted     Received    ACK Sent
Received        Segment         Segment     (Including SACK Blocks)

1500
                3000-3499       3000-3499   3500
                3500-3999       3500-3999   (delayed ACK)
2500
                4000-4499       (dropped)
                4500-4999       4500-4999   4000, SACK=4500-5000
3500
                5000-5499       5000-5499   4000, SACK=4500-5500
                5500-5999       5500-5999   4000, SACK=4500-6000
4000, SACK=4500-5000
                6000-6499       6000-6499   4000, SACK=4500-6500
                6500-6999       6500-6999   4000, SACK=4500-7000
4000, SACK=4500-5500
                7000-7499       7000-7499   4000, SACK=4500-7500
4000, SACK=4500-6000
```

```
                4000-4499      4000-4499   7500
    4000, SACK=4500-6500
```

A.3. ACK Losses

This case with ACK loss shares much behavior with the case with delayed ACK. If hole at rcv.nxt is filled, the sender will notice that cumulative ACK advanced. In case of out-of-order segments the first ACK which gets through to the sender includes SACK blocks up to the quantity the SACK block redundancy is able to cover. With this algorithm the sender immediately takes use of all the information that is made available by the incoming ACK.

```
    ACK           Transmitted    Received    ACK Sent
    Received      Segment        Segment     (Including SACK Blocks)

    1000
                  3000-3499      3000-3499   (delayed ACK)
                  3500-3999      3500-3999   4000
    2000
                  4000-4499      (dropped)
                  4500-4999      4500-4999   4000, SACK=4500-5000
                                             (dropped)
    3000
                  5000-5499      5000-5499   4000, SACK=4500-5500
                  5500-5999      5500-5999   4000, SACK=4500-6000
    4000
                  6000-6499      6000-6499   4000, SACK=4500-6500
                  6500-6999      6500-6999   4000, SACK=4500-7000
    4000, SACK=4500-5500 (two segments left the network)
                  7000-7499      7000-7499   4000, SACK=4500-7500
                  7500-7999      7500-7999   4000, SACK=4500-8000
    4000, SACK=4500-6000
                  4000-4499      4000-4499   8000
    4000, SACK=4500-6500
```

A.4. ACK Reordering

With ACK reordering an ACK is postponed. Due to redundancy the next ACK after postponed one contains not only its own information but also the information of the reordered ACK (similar to the ACK losses case). Then when the reordered ACK arrives, the sender already knows about the information it provides and therefore no actions are taken with this algorithm.

```
    ACK           Transmitted    Received    ACK Sent
```

```
Received      Segment        Segment     (Including SACK Blocks)

1000
              3000-3499      3000-3499   (delayed ACK)
              3500-3999      3500-3999   4000
2000
              4000-4499      (dropped)
              4500-4999      4500-4999   4000, SACK=4500-5000
                                         (delayed)
3000
              5000-5499      5000-5499   4000, SACK=4500-5500
              5500-5999      5500-5999   4000, SACK=4500-6000
4000
              6000-6499      6000-6499   4000, SACK=4500-6500
              6500-6999      6500-6999   4000, SACK=4500-7000
4000, SACK=4500-5500
              7000-7499      7000-7499   4000, SACK=4500-7500
              7500-7999      7500-7999   4000, SACK=4500-8000
4000, SACK=4500-6000
              4000-4499      4000-4499   8000
4000, SACK=4500-5000 (has only redundant information)
4000, SACK=4500-6500
```

A.5. Packet Duplication

Packet duplication happens either due to unnecessary retransmission or hardware duplication. It adds a redundant ACK which has only redundant information or a data segment to the stream which will triggers a redundant duplicate ACK (possibly with SACK and/or DSACK [RFC2883] information). Because neither adds any new SACKed octets at the sender, this algorithm will not do anything while duplicate ACK based receiver would falsely consider it as a duplicate ACK.

If one of the redundant ACKs is lost, the effect of duplication is just negated.

It is possible for the sender to detect this case using DSACK alone.

A.6. Mitigation of Blind Throughput Reduction Attack

In case an attacker knows or is able to guess 4-tuple of a TCP connection, it may apply a blind throughput reduction attack. In this attack TCP is tricked to send duplicate ACK to the other endpoint using out-of-window segments which it is considerably easier to achieve than a match with sequence numbers. If more than dupThresh duplicate ACKs can be triggered in row without any

legimate segment that advances acknowledged sequence number, the other end acts according that false congestion signal and halves the window.

With this algorithm such duplicate ACKs are filtered because they do not have any new in-window SACK blocks (DSACK [RFC2883] might be present though).

References

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.

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

[RFC2581bis] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 2581bis, May 2009.

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

[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.

Informative References

[FARI08] Floyd, S., Arcia, A., Ros, D., and J. Iyengar, "Adding Acknowledgement Congestion Control to TCP", Internet-Draft, draft-floyd-tcpm-ackcc-06, July 2009.

[Jac88] Jacobson, V., "Congestion Avoidance and Control", In Proc. ACM SIGCOMM 88.

[MM96] M. Mathis, J. Mahdavi, "Forward Acknowledgment: Refining TCP Congestion Control," Proceedings of SIGCOMM'96, August 1996, Stanford, CA.

[RFC896] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, January 1984.

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

[RFC3782] Floyd, S., Henderson, T., and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 3782, April 2004.

AUTHORS' ADDRESSES

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