This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the WCNC 2010 proceedings.

On Detection Algorithms for Spurious Retransmissions in TCP

Zheng Wen and Kwan L. Yeung
Department of Electrical and Electronic Engineering
The University of Hong Kong
Pokfulam, Hong Kong
E-mail: {wenzheng, kyeung}@eee.hku.hk

***Abstract* — In TCP, a spurious packet retransmission can be caused by either spurious timeout (STO) or spurious fast retransmit (SFR). The "lost" packets are unnecessarily retransmitted and the evoked congestion control process causes network underutilization. In this paper, we focus on spurious retransmission detection. We first present a survey on some important and interesting spurious retransmission detection algorithms. Based on the insights obtained, we propose a novel yet simple detection algorithm called split-and-retransmit (SnR). SnR only requires a minor modification to the TCP sender while leaving the receiver intact. The key idea is to split the retransmitted packet into two smaller ones before retransmitting them. As the packet size is different, the ACK triggered will carry different ACK numbers. This allows the sender to easily distinguish between the original transmission and the retransmission of a packet without relying on, e.g., TCP options. We then compare our SnR with STODER, F-RTO and Newreno under both loss-free and lossy network environments. We show that our SnR is resilient to packet loss and yields good performance under various simulation settings.**

I. INTRODUCTION

Transmission control protocol (TCP) has been designed, implemented and tuned to work efficiently in wired networks where bit error rate is very low and packet loss is most probably caused by congestion. In wireless networks, transmission is, however, no longer reliable [1]. Sudden delay spike and packet reordering as a result of handoff, link outage and link layer retransmission can be observed frequently. A packet loss may be incorrectly inferred by the sender where the delayed or reordered packets may finally arrive at the receiver. The retransmission of the "lost" packet is thus spurious and the associated congestion control process is also unnecessary.

In general, there are two causes of spurious retransmissions [2], spurious timeout (STO) and spurious fast retransmission (SFR). STO refers to the case that packets have been delayed in the network for a long time that exceeds the previously set RTO timer, which evokes slow start, where congestion window (*cwnd*) is set to one and slow start threshold (*ssthresh*) is halved. This is often a result of sudden delay spike which can be triggered by handoff, link level recovery and physical disconnection of wireless link. On the other hand, SFR corresponds to the case that packets get reordered to the extent exceeding the *Dupthresh* (with a default value of three). According to conventional TCP, this will trigger fast retransmit, where *cwnd* and *ssthresh* are halved. Route oscillation, link layer retransmission and multipath routing in wireless networks can all be accused for causing SFR. It should also be noticed that among packets experiencing extensive delay spike or packet reordering, some may indeed be lost due to buffer overflow or transmission error. By definition, if the lost packet is the (oldest) packet triggered retransmission, such a retransmission is not a spurious retransmission. In general, it is more challenging to handle delay spike (or packet reordering) coupled with real packet loss.

TCP performance can suffer badly from spurious retransmission [4]. Particularly, in the case of spurious timeout (STO), it may subsequently cause further fast retransmit or timeout at the sender. But if a TCP sender can detect the spurious retransmission (a posteriori), appropriate actions can be taken to rectify the situation. In this paper, we only focus on designing efficient detection algorithms for spurious retransmission.

The nature of cumulative ACK in TCP makes the spurious retransmission detection difficult. When an ACK that acknowledges/covers the retransmitted packet arrives, the sender cannot distinguish if this ACK corresponds to the original transmission or the retransmission of the packet -- a phenomenon known as acknowledgment ambiguity [3]. If the ambiguity could be removed, the sender can declare that the earlier retransmission is spurious if the received ACK corresponds to the original transmission; otherwise [1], the retransmission is deemed necessary/non-spurious. Some important and interesting work on spurious retransmission detection include Eifel algorithm [2,5], F-RTO [4,6], DSACK based algorithm [7], STODER [8], and more recently, ECN nonce based algorithm [9]. Among them, the first three have been published as IETF RFCs. In this paper, a comparative study on the existing detection algorithms is first carried out. Based on it, we propose our own algorithm split-and-retransmit

This work is supported by Hong Kong Research Grant Council General Research Fund HKU 719108E.

[1] Throughout the paper, we assume that there is no out-of-order between the retransmitted packet and the packets sent before retransmission.

(SnR). In SnR, the acknowledgement ambiguity is elegantly removed by splitting the retransmitted packet into two smaller ones before retransmitting them. When an ACK *partially* acknowledged the retransmitted packet arrives, this must be triggered by the receiving of the first split-and-retransmitted packet at the receiver. That implies the earlier retransmission is necessary. Instead, if the ACK (cumulatively) acknowledges the entire retransmitted packet, this must be due to the late arrival of the original transmission of the packet. Then the earlier retransmission is spurious. A striking feature of our SnR algorithm is its simplicity and ease of implementation: only a minor modification is required at the TCP sender, while the TCP receiver is kept untouched.

The rest of the paper is organized as follows. In Section II, we present a critical survey of major spurious retransmission detection algorithms. In Section III, SnR algorithm is introduced. In Section IV, the performance of SnR is compared with other detection algorithms by simulations. In Section V, some salient points of our SnR are highlighted using examples. Finally, we conclude the paper in Section VI. Throughout the paper, “packet” and “segment” will be used interchangeably.

II. Survey on Spurious Retransmission Detection Algorithms

The problem of acknowledgement ambiguity is the root for TCP to suffer from spurious retransmissions. Since a TCP sender can not distinguish an ACK for the original transmission from the one for segment retransmission, on receiving an ACK, the sender can not tell whether the earlier retransmission is spurious or not. Based on the way used to tackle the acknowledgement ambiguity problem, existing work on spurious retransmission detection is summarized and compared in Table I, and we elaborate on each of them below.

TABLE I
COMPARISON OF EXISTING ALGORITHMS

	Eifel	**ECN based**	**DSACK based**	**STODER**	**F-RTO**
TCP Option	Timestamp	N/A*	SACK	N/A	N/A
RFC	RFC 3522	N/A	RFC 3708	N/A	RFC 4138
Detection Time	<RTT	<RTT	>RTT	<RTT	>RTT
STO	Yes	Yes	Yes	Yes	Yes
SFR	Yes	Yes	Yes	Yes	N/A
Tx_Modifi	Yes	Yes	Yes	Yes	Yes
Rx_Modifi	Yes	Yes	Yes	No/A	No/A

* ECN nonce bit in IP datagram header is required.
[a] Tx_Modifi / Rx_Modifi refers to sender/receiver side modification.

A. Eifel algorithm

Eifel algorithm [2], specified in RFC 3522, relies on the TCP timestamp option [10] to remove acknowledgement ambiguity. The underlying idea is that packet retransmission must happen later than the original transmission. The sender adds a timestamp to each out-going packet. On receiving a packet, the receiver reflects back the timestamp value in the corresponding ACK. The sender will declare a (previous) retransmission spurious if the timestamp in the *first* received ACK that advances congestion window *cwnd* (i.e. acknowledges some outstanding packets) is smaller/earlier than the timestamp corresponding to the retransmit packet. Assuming no ACK loss, the time required for the arrival of the first ACK (that advances *cwnd*) after retransmission is at most one round trip time (RTT) between the sender and receiver. So the spurious retransmission detection time of Eifel algorithm is less than a RTT.

The side effects of Eifel are obvious. Firstly, the adoption of TCP timestamp option introduces an overhead of up to 12 bytes for every outgoing packet (including ACKs); this decreases TCP goodput [4]. Secondly, the TCP timer granularity (of 500 ms) may not be sensitive enough for the Eifel algorithm to detect spurious retransmission where retransmission occurs in less than a RTT (typically 200ms), which implies that the timestamps in both original and retransmission may be the same. Thirdly, when being implemented with TCP Reno, Eifel algorithm may not function well if spurious delay is accompanied by packet loss [4]. This is because RFC1323 [10] requires that the echoed timestamp should respond to the most recent data segment that advanced the window. When there is a packet loss, the echoed timestamp may not be reliable for the sender to make a judgment.

B. ECN nonce based algorithm

The ECN nonce based algorithm [9] makes use of the ECN nonce codepoints [15] in the IP datagram header. The ECN-capable sender sets ECN nonce value equal to 1, i.e. ECT(1), for original transmission and ECT(0) for retransmission. The ECN-capable receiver reflects the ECN nonce value back to the sender by using the “Nonce Sum” field in the TCP header. On receipt of ACK, sender will declare a spurious retransmission if nonce value in ACK equals to 1. The smart use of the nonce value achieves the same effect as that of Eifel algorithm while removing the timestamp option overhead. However, an ECN receiver does not directly reflect the nonce value, instead it “reflects” the calculated Nonce Sum (previous nonce XOR the new one). In other words, this detection algorithm relies on the deployment of ECN protocol but at the same time, may not be fully compatible with it.

C. STODER algorithm

The STODER algorithm [8] tackles the acknowledgement ambiguity problem using repacketization (same as our SnR algorithm). On TCP sender timeout, a *k*-byte smaller packet is generated and retransmitted instead of the original one (whereas in our SnR, two equally split packets are sent). Then the sequence number of the last data byte in the retransmit packet, denoted by s-redge, is stored. When an ACK arrives, TCP sender declares the previous retransmission spurious if it acknowledges more data beyond s-redge. Otherwise, it reacts as in a normal timeout situation. Notably, *all* outstanding

packets in the retransmission queue[2] must be repacketized for retransmission as the packet delineation/boundary is shifted (to left) by *k* bytes. The TCP receiver may also need to deal with the payload overlap problem if some of the outstanding packets have already been received. Another issue [8] with STODER is that the TCP sender needs to keep an extra state variable (s-redge) for the first retransmit packet when Nagle's algorithm is in function, or a series of variables for tracing every outstanding packet if Nagle's algorithm is not adopted. Nevertheless, in terms of protocol overhead and compatibility, STODER outperforms Eifel and the ECN-based algorithm.

D. DSACK

As an extension to SACK [14], DSACK [11] uses the first SACK block to specify the segment that triggers a duplicate acknowledgement (Dup_ACK) at the receiver. The DSACK based algorithm [7] makes use of this feature to detect spurious retransmission. The general idea is straightforward: if a retransmitted packet has been acknowledged for the second time, the earlier retransmission is spurious. In the DSACK based algorithm, TCP sender keeps a packet scoreboard, marking each packet as RE_tx, ACK and Dup_ACK.. The TCP sender will declare a spurious retransmission if a retransmitted packet has been acknowledged for the second time.

The detection speed using DSACK is essentially slower than Eifel, the ECN nonce based algorithm and STODER, since the TCP sender can only detect spurious retransmission upon receiving the DSACK corresponding to the retransmit packet, which requires at least on RTT. Besides, apart from the overhead of 12 bytes, the DSACK based algorithm is not robust to ACK loss. Because DSACK option will be installed only once, the loss of the corresponding ACK disables the detection process.

E. F-RTO

F-RTO [4], as specified in RFC 4138, scrutinizes the incoming ACK numbers for inferring if a retransmission is spurious. Unlike the algorithms discussed above, F-RTO can only detect spurious retransmission due to timeout (i.e. STO). Due to the heuristic nature of F-RTO, it is designed to be conservative. Accordingly, it requires a longer time to detect a spurious retransmission.

A key idea of F-RTO is that receipt of ACK for any non-retransmitted segment after timeout indicates a spurious retransmission. On the RTO timer expiry, TCP sender enters slow start phase and retransmits one outstanding packet. If the first incoming ACK after retransmission advances the *cwnd*, TCP sender will transmit another two *new* packets (i.e. packets are not transmitted previously). (But if the sending of the two new packets is prohibited by the current *cwnd* size or the availability of data for sending, F-RTO gives up the detection and treats the timeout as real,) These two packets will trigger Dup_ACKs in case of real timeout. But if the second incoming ACK still advances the *cwnd* (i.e. not a Dup_ACK), it is interpreted as an indication of spurious timeout. Otherwise, a "real" timeout occurs and the sender will revert to the conventional slow start algorithm to retransmit the outstanding packets.

F-RTO requires no modification of the receiver, and only limited change to the TCP sender. But its efficiency is limited due to its heuristic nature. Firstly, F-RTO does not support the detection of spurious fast retransmit. Secondly, the detection time (about 2RTT) is typically longer than other algorithms because the need for two ACKs to arrive after retransmission. Thirdly, F-RTO is not able to detect all the cases of spurious timeout. Notably, if either of the first two ACKs is a Dup_ACK, F-RTO will declare the retransmission non-spurious. Although the detection accuracy can be improved by a SACK-enhanced version of F-RTO [12] (when the second ACK is a Dup_ACK), TCP option header must be used.

[2] TCP sender would normally buffer the outstanding packets in a retransmission queue until they are acknowledged [13].

III. SPLIT-AND-RETRANSMIT ALGORITHM

In typical TCP implementation [13], the sender buffers all the outstanding packets in a retransmission queue until they are acknowledged. If a retransmission is required, the first/oldest outstanding packet in the queue is resent. In our proposed split-and-retransmit (SnR) algorithm, the retransmit packet is equally split into two. They are then transmitted together as if the original outstanding packet is resent. Note that the retransmission queue only needs to store the original (i.e. non-split) packet. The receiver acts as normal and sends back ACK upon receiving packets. At the sender, when the first ACK that advances the *cwnd* after retransmission arrives, the sender declares the earlier retransmission spurious if all the data bytes in the first outstanding packet (in the retransmission queue) are acknowledged. Otherwise, if the received ACK partially acknowledges the first outstanding packet, which implies that the ACK is triggered by the arrival of the first split-and-retransmitted packet, the earlier retransmission is deemed necessary. It is worthwhile to note that the TCP sender has the flexibility to split an outstanding packet into any number of smaller packets according to the need. However, over-splitting should be avoided because each additional packet will incur a 40-byte overhead, 20-byte for TCP and 20-byte for IP. Finally, Fig. 1 and Fig. 2 show the packet trace of SnR under spurious timeout and real timeout, where the Eifel response algorithm [2] of fully restoring the values of *cwnd* and *ssthresh* after detecting a spurious timeout is adopted.

Both SnR and STODER uses repacketization of retransmit packet for removing acknowledgement ambiguity, but SnR does not require to maintain the extra state variable (s-redge), and does not need to repacketize all the outstanding packets in the retransmission queue in case of non-spurious retransmission. Besides, as each of the split packets is capable of triggering an ACK at the receiver, and the arrival interval between the two ACKs is much less than a RTT, the sender can collect and react to the network/receiver status in a more timely manner, e.g. if there is a real packet loss among the delayed outstanding packets, the extra (duplicate) ACK (corresponding to the second split packet) can allow the sender to detect the loss faster, i.e. to exceed the *Dup_Threshold* for fast retransmit

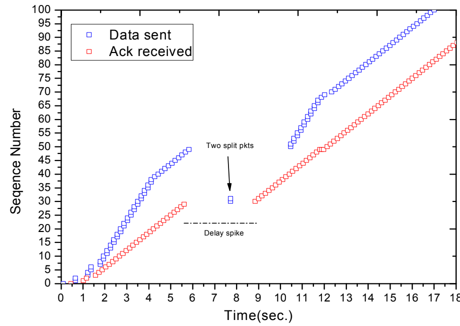

Fig. 1. Packet trace of Split-and-Retransmit after a spurious timeout

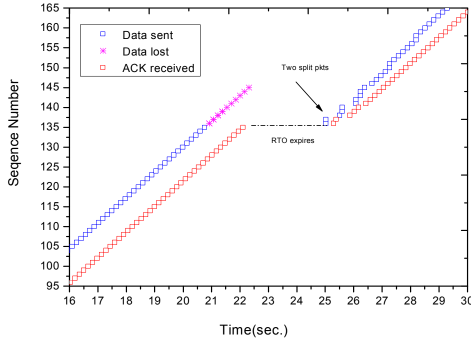

Fig. 2. Packet trace of Split-and-Retransmit after a real timeout

quicker. (Please also refer to the example in Section V. B.) .

IV. PERFORMANCE EVALUATION

To evaluate the proposed SnR algorithm, we implement SnR in NS2 version 2.27 and test it in various situations. The performance of (TCP NewReno with) SnR is compared to plain NewReno, (Newreno with) F-RTO and (NewReno with) STODER. As our focus is on spurious retransmission detection, the same Eifel response algorithm of fully restoring the *cwnd* and *ssthresh* value after detecting a spurious timeout is adopted by all detection algorithms.

We set up a simple topology to emulate the last-hop wireless link, where the TCP sender is connected to a wireless gateway before reaching the TCP receiver through wireless link. We select the link parameters to approximate the properties of a typical scenario where the link data rate is set to 1.4Mbps with a one way propagation delay of 200ms. The router buffer size is set to be sufficiently large to remove the influence of buffer overflow. For TCP, we use 512 bytes maximum segment size and 64 Kbytes receiver buffer size. To trigger spurious timeouts, we adopt the delayer module, a build-in module in NS2, to generate delay spikes.

For each scenario, we run the simulation 30 times for each detection algorithm. Each simulation run lasts for 100 seconds and the sender always has data to send. For fair comparison, the same random number seed is used for all algorithms in a particular scenario. We choose goodput, the successfully delivered bytes over simulation time, as our performance metric. The results are depicted using a box-plot graph, where the central line represents the median goodput and the central mark refers to the mean value of data sets. The lower and upper bound of the box present the first and third quartiles of the results, respectively. The top and bottom ends of the vertical line refer to the ninety five and five percentile, respectively. The maximum and minimum values we get are shown using the star marks at the top and bottom area.

A. Delay spike

As the primary objective of our SnR is to detect spurious timeout caused by delay spike, we first consider delay spikes without packet loss. The lengths of the delay spikes are exponentially distributed with a mean of 1.5 seconds. The time intervals between every two consecutive delay spikes are also exponentially distributed with a mean of 20 seconds. Fig. 3 shows the goodput distribution of different algorithms. As expected, STODER and SnR outperform TCP NewReno and F-RTO in terms of median goodput value, achieving an improvement of up to 31.7% with respect to NewReno and 15.6% with respect to F-RTO. This confirms that SnR and STODER have higher detection accuracy. (Note that if the sending of two new packets upon receiving the first acceptable ACK is limited by the available congestion window size, F-RTO will give up the detecting process and treat the retransmission as real.) Although STODER and SnR give the same performance in terms of median goodput, STODER performs slightly better in terms of mean goodput. This could be due to the additional 40-byte packet header overhead of SnR in split-and-retransmit a packet.

B. Delay spike with packet loss

We now consider the case of delay spike coupled with packet loss. We adopt the two-state Markov loss model [16] for properly characterizing the packet loss due to congestion and transmission errors. The duration of good state and bad state are exponentially distributed with an average of 20 seconds and 3 seconds, respectively. The state transition matrix is set to [0.9 0.1; 0.7 0.3]. In other words, the transition probability from good to bad is 0.1 and from bad to good is 0.7. Fig. 4 shows the state diagram of the Mokov loss model. Two loss scenarios are studied. In Scenario I, all packets sent during bad states are lost with probability 1, whereas in good state no packet will be lost. Scenario I corresponds to a bursty loss model, where packets are dropped (mainly) due to buffer overflow. In Scenario II, packets are lost following the Bernoulli model with a probability of 1% in good state and 90% in bad state. Scenario II tries to capture the loss due to transmission errors on wireless links. In both scenarios, we assume that ACKs are always reliably delivered, and delay spikes are generated same as the case without packet loss.

Fig. 5 compares the goodput performance of the four detection algorithms under a bursty loss model, i.e. Scenario I. We can see that SnR provides the highest mean and median throughput. SnR outperforms STODER because it benefits from the extra ACK triggered by the second split packet: when a real packet loss occurs, the extra ACK allows the sender to detect the loss faster, i.e. to exceed the *Dup_Threshold* for fast retransmit earlier. Fig. 6 shows the goodput performance under the loss Scenario II. In this case, the performance is dominated by packet loss instead of delay spikes. It is interesting to note that the gain of using STODER is largely offset by its mediocre performance in dealing with packet loss. Thus it shows almost the same goodput as NewReno. SnR is more resilient to loss and it improves the goodput by about 13% compared to NewReno and STODER. In other words, the advantage of retransmitting the oldest outstanding packet as two split packets

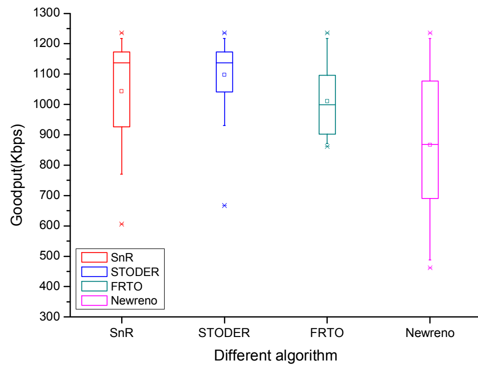

Fig. 3. Goodput comparison under delay spike without packet loss

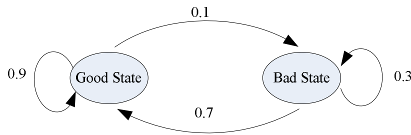

Fig. 4. Two-state Markov loss model

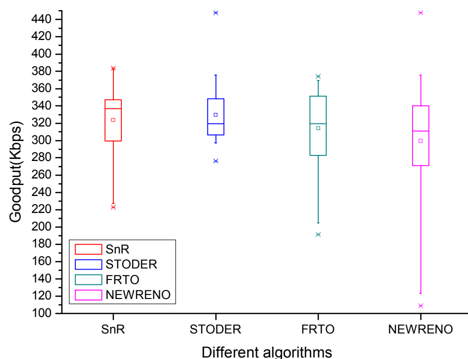

Fig. 5. Goodput performance under delay spike with loss Scenario I Ⅰ

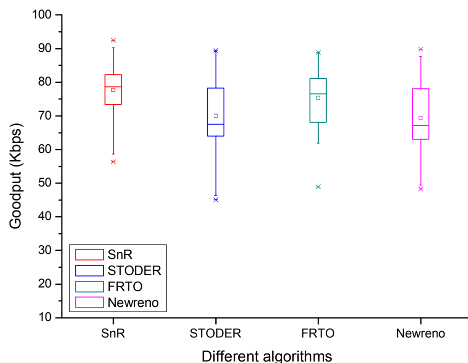

Fig. 6. Goodput performance under delay spike with loss Scenario Ⅱ

is more noticeable over lossy links, as the ACKs triggered by the split packets will help the sender getting into the fast retransmit faster. It is also worthwhile to note that F-RTO performs relatively better than Fig. 3 (without packet loss) and Fig. 5 (with congestion loss). This can be partly explained by the fact that in this Scenario II, the TCP goodput performance is largely determined by packet loss, whereas a conservative algorithm like F-RTO is more resilient to packet loss.

V. CASE BASED ANALYSIS

Notably, in our simulation study above, only delay spike coupled with/without packet loss are considered, just like [4, 8]. In practice, out-of-order packet arrival and the loss of ACKs are unavoidable. In this section, we use specific examples to highlight the additional advantages of our SnR in the presence of out-of-order packet arrival and loss of ACKs.

A. *When delayed packets are disordered*

During delay spike, when the oldest outstanding packet gets reordered, the first incoming ACK after timeout would normally be a duplicate one. Fig. 7 depicts an example. Suppose B is the oldest outstanding packet that triggers a spurious timeout. As B is reordered and arrives after packet C, the first ACK the sender receives will be a Dup_ACK(ACK_B) triggered by the receipt of packet C. This Dup_ACK will disable the F-RTO detection process. As a result, it would mistakenly declare a spurious timeout to be real. Note that in slow start, TCP sender would not open *cwnd* until receiving an ACK that acknowledges new data (Acceptable_ACK). In other words, F-RTO in this case will wait for another real timeout to retransmit packets.

On the other hand, our SnR only begins to act on receiving the first Acceptable_ACK (i.e. ACK_C) which corresponds to the arrival of packet B, as shown in the bottom case in Fig. 7. Dup_ACK (ACK_B) received after timeout would simply be ignored. This shows that SnR is robust to the reordering among delayed packets. The relative performance gain would be more noticeable if the reordering is more severe, e.g. the oldest packet B is reordered to arrive after D (instead of C).

B. *A simple response algorithm*

Eifel algorithm suffers from real packet losses. A well-cited corner case [4] is that when all outstanding packets are lost except the oldest one, the oldest one times out (because the associated ACK is also lost) and sender retransmits it. The retransmitted packet triggers an ACK with a timestamp corresponding to the first arrival of the oldest packet at the receiver. When this ACK arrives at the sender, Eifel declares a spurious timeout (because the timestamp in the ACK is older then the one stored). The sender then starts to transmit new packets, which will lead to another real timeout (due to the inability of the fast recovery algorithm in TCP Reno).

In the corner case above, as far as detection is concerned, Eifel is accurate in declaring the retransmission of the oldest packet is spurious, because the receiver already has it. The poor performance is due to the response algorithm used: if spurious retransmission is detected, transmit new packets instead of outstanding ones. We found that such a response algorithm is too aggressive because there are true packet losses in the network. We propose to modify the response algorithm (of Eifel) as follows: the sender transmits new packets on detection of spurious retransmission, and retransmits outstanding packets if the *second/next* ACK is a duplicate. The second ACK is used for loss detection and if the second ACK is a Dup_ACK, it is a strong indication of packet loss in the network because the lost packets (as well as other outstanding packets) were sent before sender timeout. Indeed, F-RTO follows the same design principle.

For our SnR, the simple response algorithm above can be more efficient than Eifel and F-RTO. Fig. 8 shows an example. For the case of Eifel algorithm, consider B is the oldest

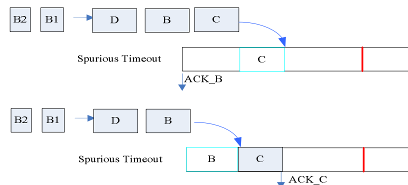

Fig. 7. An example when delayed packets got reordered

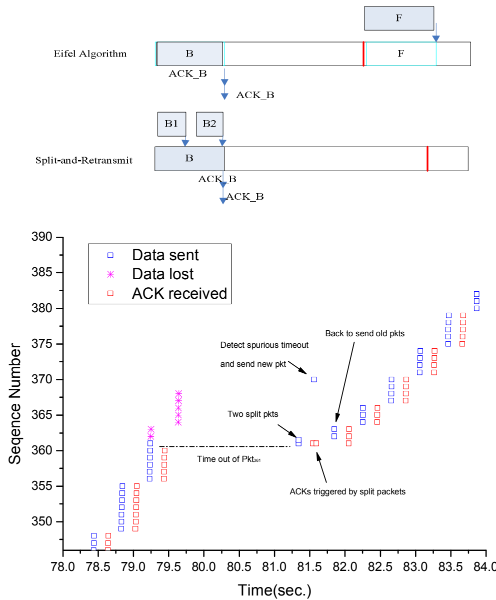

Fig. 8. Tackling the problem of packet loss coupled with spurious timeout

outstanding packet buffered at the receiver before timeout and original ACK_B was lost during delay spike. On the arrival of the retransmit of B, ACK_B is generated. When sender receives it, it declares the earlier retransmission of B is spurious, and a new packet, F is sent, which in turn triggers another (duplicate) ACK_B back to the sender. Note that this duplicate ACK_B will arrive one RTT after its detection of spurious retransmission. With our proposed simple response algorithm, the sender realizes packet C was lost, and thus it falls back to send outstanding packet C. For the case of SnR, when split-packet B1 arrives, (new) ACK_B is generated. When B2 arrives, duplicate ACK_B is generated. By the time the second/duplicate ACK_B arrives at the sender, the sender can revert to retransmitting outstanding packet C much faster. Fig. 8 also shows the corresponding packet trace of using SnR to handle the corner case that all outstanding packets are lost except the oldest one.

VI. Conclusion

In this paper, we first conducted a comparative study of some interesting and important spurious retransmission detection algorithms. Based on the insights obtained, we proposed a new detection algorithm called split-and-retransmit (SnR). In SnR, instead of retransmitting a packet, we split it into two and send both as a single retransmission. When an ACK partially acknowledged the retransmitted packet (before splitting) arrives, we know this is triggered by the receiving of the first split-and-retransmitted packet at the receiver. So the earlier retransmission is necessary. If the ACK (cumulatively) acknowledges the entire retransmitted packet, this must be due to the late arrival of the original transmission of the packet. Then the sender declares that the earlier retransmission is spurious. Compared to TCP Newreno, simulation results showed that our SnR provides up to 31.7% increase in goodput when spurious timeouts occur in a loss free environment, and up to 10% when delay spike is coupled with packet loss.

References

[1] Jianhao Hu, G. Feng and Kwan L. Yeung, “Hierarchical Cache Design for Enhancing TCP over Heterogeneous Networks with Wired and Wireless Links,” IEEE Transactions on Wireless Communications, Vol.2, No.2, pp.205–217, March, 2003.

[2] R. Ludwig and M. Meyer, “The Eifel algorithm: making TCP robust against spurious retransmissions,” *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 1, pp. 30-36, 2000.

[3] P. Karn and C. Partridge, “Improving round –trip time estimates in reliable transport protocols,” ACM Transactions on Computer Systems, vol. 2, pp. 2-7, 1987.

[4] P. Sarolahti, M. Kojo, and K. Raatikainen, “F-RTO: an enhanced recovery algorithm for TCP retransmission timeout,” *SIGCOMM Comput. Commun. Rev.*, vol. 33, no.2, pp. 51-63, 2003.

[5] R. Ludwig and M. Meyer, “The Eifel Algorithm for TCP,” RFC3522 (Experimental), Apr. 2003.

[6] P. Sarolahti and M. Kojo, “Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP),” RFC 4138 (Experimental), Aug. 2005.

[7] E. Blanton and M. Allman, “Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions,” RFC 3708 (Experimental), Feb. 2004.

[8] K. Tan, Q. Zhang and W. Zhu, “ STODER: a robust and efficient algorithm for handling spurious retransmit timeouts in TCP,” GLOBECOM’05, pp.3692–3696, St. Louis, Missouri, USA, Dec. 2005.

[9] M. Welzl, “Using the ECN nonce to detect spurious loss events in TCP,” *GLOBECOM’08*, pp.1-6, New Orleans, LA, USA, Dec. 2009,.

[10] V. Jacobason, R. Braden, and D. Borman, “TCP Extensions for High Performance,” RFC 1323 (Proposed Standard), May 1992.

[11] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, “An Extension to the Selective Acknowledgement (SACK) Option for TCP,” RFC 2883 (Proposed Standard), Jul. 2000.

[12] P. Sarolahti, “Congestion control on spurious TCP retransmission timeouts,” *GLOBECOM’03*, pp. 682- 686, San Francisco, USA, Dec. 2003,.

[13] R. Braden, “Requirements for Internet Hosts -- Communication Layers,” RFC1122, Oct. 1989.

[14] M.Mathis, J.Mahdavi, S. Floyd and A. Romanow, “TCP Selective Acknowledgement Options,” RFC 2018 (Proposed Standard), Oct. 1996.

[15] K. Ramakrishnan, S. Floyd and D.Black, “The Addition of Explicit Congestion Notification (ECN) to IP,” RFC 3168 (Proposed Standard), Sep. 2001.

[16] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley, "Measurement and modeling of the temporal dependence in packet loss," INFOCOM’99, pp. 345–352, Los Angeles, USA, March 2009,.