Using the ECN Nonce to detect Spurious Loss Events in TCP

Michael Welzl
University of Innsbruck
Innsbruck, Austria
Email: michael.welzl@uibk.ac.at

***Abstract*—A sudden delay spike or reordering in the network can cause TCP to experience a loss event. Since loss is interpreted as a sign of congestion in TCP, this causes the protocol to reduce its sending rate. Several mechanisms for detecting and reacting to such spurious loss events have been proposed; each of them has some advantages and disadvantages. We extend this space with a new detection mechanism which complements the existing ones well. Furthermore, the mechanism is easy to implement because it only needs the sender to intelligently interpret feedback from ECN nonce.**

I. Introduction

There are currently two ways for a standard TCP sender to detect congestion and react:

1) When no ACKs arrive at the sender for a long time, a timeout occurs. The timer, which is calculated as a function of round-trip time (RTT) measurements, is designed to be robust against delay fluctuations. When it fires, the sender assumes that it has no more packets in flight. Therefore, the state of the network is unknown, and the most appropriate decision is to restart with the initial window size (i.e. go into slow start mode). All unacknowledged packets are retransmitted by TCP at this point.
2) It halves its congestion window and resends the (first or only) missing packet upon arrival of three duplicate ACKs (DupACKs). This reaction, which is called "fast retransmit / fast recovery (FR/FR)", is based on the assumption that the packet preceding the three packets which caused the receiver to send the DupACKs was dropped because of congestion in the network. The arrival of DupACKs, however, signifies that there are still packets in flight; hence, there is no need to react as conservatively as in the timeout case.

Both of these congestion signals can also occur because of other reasons, misleading the sender which then reacts as if there was congestion when in fact there was none. Such occurrences are called *spurious loss events*. A spurious timeout can be caused by a sudden delay spike, which significantly delays all the packets that are currently in flight, and spurious FR/FR can occur when a single packet is significantly delayed (i.e. the packets are reordered on the way to the receiver, or ACKs are reordered on the way back). While the effect of a spurious FR/FR may seem to be much less significant than a spurious timeout, the unnecessary retransmissions (called *false fast retransmits* in [1]) that it brings about can cause the problem to occur several times in a row, repeatedly reducing the sender's congestion window.

Making TCP robust against such occurrences can make it operate better in the presence of adverse network effects such as wireless handovers or route oscillation. Doing so may also enable the future deployment of mechanisms which reorder packets but would otherwise be beneficial, such as multi-path routing or exploiting parallelism in packet forwarding; more details can be found in [1].

Several mechanisms for detecting spurious loss events have been proposed, all of which have some advantages and disadvantages. We present another one, which could be used to complement the existing mechanisms in environments where certain features are not desired or available, and to increase the likelihood of successfully detecting and reacting to the problem. A striking feature of our mechanism is its simplicity and ease of implementation — the only change needed to TCP, in addition to supporting the (currently experimental) ECN nonce, is to interpret the incoming bits in a slightly more sophisticated way.

How a sender should react once a spurious loss event was detected is not the focus of this paper. In our simulations, we follow the common approach in related work, which is to restore the original congestion state at the sender to the state before congestion was (wrongly) detected. A detailed discussion of the reaction to a spurious loss event can be found in [2], and a full specification of the sender's reaction to a spurious timeout is given in RFC 4015 [3]. Notably, while this RFC does not specify a reaction to spurious FR/FR because of disagreement regarding the way that the DupACK threshold should be updated, this does not mean that an appropriate reaction will never be defined. RFC 4015 only foresees reverting to the previous congestion control state in the absence of an ECN signal from the receiver; our ECN nonce based mechanism, therefore, only operates when there is no congestion. This also means that, although the mechanism is related to ECN, no router support for ECN is needed.

After a review of related work in the next section and an introduction to the ECN nonce mechanism in Section 3, we will explain our mechanism in Section 4. A simulation study is presented in Section 5, and Section 6 concludes.

II. Related work

The *Eifel* detection algorithm [4], specified in RFC 3522 [5], uses the Timestamps option to distinguish an ACK that was caused by the original packet from an ACK that was caused by its retransmission. The Timestamps option was introduced in RFC 1323 [6] as a means to fix the *retransmission ambiguity* problem: a sender adds a timestamp to a packet, which is reflected by the receiver in the corresponding ACK and later used by the sender to calculate the RTT as $current\ time - timestamp\ in\ the\ ACK$. Since both timestamps are generated by the same host, no time synchronization is needed for this process. The purpose of this option was merely to enable using retransmitted packets and their corresponding ACKs for updating the timeout value (which is normally not done since the possibility of falsely updating this value because of a delayed packet or ACK was described in [7]).

If the timestamp in the ACK is smaller (earlier) than the timestamp that the sender stored for calculating the RTT when it retransmitted the packet, a spurious loss event is identified by the Eifel algorithm. The significant disadvantage of this method is its use of the Timestamps option. This option needs 10 bytes in every packet. It can be exploited by operators to detect Network Address Translators (NATs) because a sequence of outgoing packets from various NAT'ed hosts will share the same source address yet may not show a strictly increasing timestamp, and it can make the system vulnerable to an attack.[1]

DSACK ("Duplicate SACK") [8] is an extension to the "Selective Acknowledgment" (SACK) option in TCP which enables a receiver to inform the sender about the reception of duplicate packets. As described in [9], this information can be used by the sender to detect a spurious loss event. A detailed specification of this process is given in RFC 3708 [10]. The detection method is potentially slower than Eifel: the sender can only detect the problem with DSACK when the ACK corresponding to the retransmitted packet arrives at the sender.

F-RTO [11], specified in RFC 4138 [12], is a pure sender-side mechanism which can detect a spurious timeout (but, in the currently proposed version, no spurious FR/FR) without requiring any TCP options. Roughly, it works as follows: upon expiry of the timer, slow start is entered and a packet is transmitted as usual. In addition to standard TCP behavior, F-RTO more carefully monitors incoming ACKs. If the first ACK after the first retransmission advances the window, the timeout might be spurious. In order to test whether this really is the case, instead of retransmitting the next two packets, up to two new ones are sent in the next step. If the timeout is not a spurious one, these two packets cause DupACKs. If, however, the next ACK also advances the window, the retransmit timeout (RTO) is declared spurious.

Note that this algorithm cannot *always* detect spurious timeouts, as it does not normally react upon DupACKs. Thus, if no ACKs arrive which advance the window, F-RTO cannot detect that the timeout was spurious. While the mechanism can make use of DupACKs if SACK is used, the operation of F-RTO is also limited in that it only considers incoming ACKs after the timeout which acknowledge a number that is smaller than the highest sequence number that was used so far. This is done to prevent misinterpreting an ACK corresponding to a single delayed packet (from a previous FR/FR) as an indication of a spurious timeout.

Eifel, DSACK and F-RTO are the only spurious loss event detection methods that have been published as RFCs up to now. There are however a few other related efforts: *STODER*, for example, avoids the retransmission ambiguity problem by letting the sender transmit a packet which is smaller than the original one instead of retransmitting the packet as it is. An ACK caused by the smaller packet can then be distinguished from an ACK that was caused by the original one by examining the number of bytes that are acknowledged. While this mechanism was shown to perform very well in [13], it is arguably quite a significant change to the TCP implementation in comparison to most other methods (and especially the one presented here).

A Performance Enhancing Proxy (PEP) based approach is described in [14] — here, the performance of TCP is enhanced by filtering out DupACKs to the sender when a spurious timeout has happened in order to prevent it from unnecessarily retransmitting packets. This method is based on the assumption that the sender is unable to detect a spurious timeout by itself. It nicely complements existing work because it is easy to deploy. Another mechanism which falls in the category of transparently coping with the problem is described in [15]: here, the idea is to let the link layer avoid a spurious timeout by artificially increasing the RTT before a wireless handover occurs, such that the delay spike does not become significant enough for TCP's RTO timer to fire.

The authors of [16] devised a spurious timeout detection algorithm which operates on historical trace files, and used it to study the frequency of such events in the Austrian UMTS and GPRS networks, with the result that spurious timeouts rarely occurred in this environment.

III. ECN and the ECN Nonce

The ECN nonce mechanism is inherently unrelated to the detection of spurious loss events; however, it is inevitable to briefly explain its operation together with a short overview of ECN itself because our mechanism is based upon it.

Active Queue Management mechanisms like RED [17], which are designed to regulate fairness among flows and prevent traffic phase effects [18] can improve the overall performance of the system by dropping packets at the onset of congestion. Since there would not be any real need to actually drop packets early except for communicating the message "there was congestion, reduce your rate" to senders, the same can be achieved by letting these mechanisms set a bit in packets. This bit is a message to the receiver, informing it to tell the sender that it should react as if the packet had been

[1] `http://www.kb.cert.org/vuls/id/637934`

dropped. This is what Explicit Congestion Notification (ECN) does [19].

One downside of this design is that it is not incentive compatible — it is not in the interest of the receiver to inform the sender of congestion in the network in most standard usage scenarios (e.g. when the receiving TCP is feeding its data into a web browser). Hence, a selfish receiver could simply decide not to include this notification in its ACKs. This problem is addressed by the ECN nonce [20], [21].

The idea behind this scheme is to let the sender insert a random number (a "nonce") in every packet. A router which sets a bit to inform the receiver (and consequently the sender) of congestion is supposed to delete this number, and the sender only believes the receiver that there was *no* congestion if it receives the correct random number back. Using the nonce is usually in the interest of the sender, as it helps ensure that all receivers are equally served (which is desirable from the perspective of a web server, for example).

The specifications of ECN and its nonce are somewhat more complicated because of several reasons, including backward compatibility (routers need to be informed if a packet belongs to a TCP connection where the sender and receiver are ECN-capable before deciding whether it should be dropped or marked) and space (it is not feasible to store a long random number in the IP header). Here is an overview of how the mechanism really operates:

- A packet carries a two-bit "ECN field" in its IP header. The defined values of this field are:
  - **00:** ECN is not used
  - **01:** This value is called ECT(1), which means "ECN-capable transport" (i.e. routers are told that the sender and receiver understand ECN) with a nonce value of 1.
  - **10:** This value is called ECT(0), which has the same meaning as ECT(1) but with a nonce value of 0.
  - **11:** This value can be set by ECN-capable routers as a congestion signal if the value of the field was ECT(0) or ECT(1) before.
- A sender randomly sets the value of the field to ECT(0) or ECT(1), thereby choosing a nonce value of 0 or 1, respectively.
- In the absence of congestion, the packet arrives at the receiver with this field intact.
- The receiver will then echo a nonce sum (calculated as the previously received nonce XOR the new one) back to the sender in a field called "Nonce Sum". This field is located in the TCP header because it is not necessary for routers to see it. The reason for using a sum is that, since ACKs are unreliable in TCP, a receiver could otherwise simply skip an ACK in order to hide a congestion signal from the sender. Since this detail is irrelevant for the operation of our algorithm, we assume that the receiver simply sends back the nonce as it is in the context of this paper for the sake of simplicity.

In the presence of congestion, the router sets the ECN field to "11", which means that the receiver does not obtain a nonce. It then either sends the previous old nonce sum or "0" back and additionally sets a flag called "ECN Echo" (ECE) in the TCP header on all consecutive ACKs until a retransmitted packet arrives where the "Congestion Window Reduced" (CWR) flag is set. This flag causes the receiver to stop setting ECE=1.

Obviously, with a nonce that can only have the values 0 and 1, there is a 50% chance for a malicious receiver to guess the correct value. Proper security is attained by requiring the receiver to correctly set the nonce sum over the full time of the connection and immediately considering it an attack if the value of this field is incorrect.

IV. Detecting Spurious Loss Events with the ECN Nonce

The idea of our algorithm is simple: if an original packet was sent with the nonce value X and, upon retransmission, the nonce value Y was used, the nonce that is reflected by the receiver can be used to eliminate the retransmission ambiguity problem. If the receiver sends back Y, everything is normal, but if it sends back X, the ACK corresponds to the original packet and the retransmission was unnecessary, which means that a spurious loss event occurred. Note that, as mentioned before, we will assume that the receiver simply reflects the nonce value back to the sender instead of calculating a sum for the sake of simplicity.

What are the possible values of X and Y? Since ECN must be disabled for retransmitted packets according to the specification, they carry no nonce — RFC 3540 [21] does not define the value to be returned in such a case, and states that the receiver behavior is otherwise unchanged from RFC 3168 [19], where the "Nonce Sum" is a reserved field. This means that it should be set to 0 by the receiver. Y is therefore always 0, and it is only possible for the mechanism to detect a spurious loss event when $X = 1$. The chance for this to happen is 50%.

In order to increase the detection probability, and to avoid relying on the receiver to consistently set Y to 0, the sender can wait for more than one ACK until the loss event is declared spurious. This is a trade-off between robustness and reaction speed of the mechanism. We introduce a parameter called $ecnsp\_param$, which denotes the number of additional ACKs that the mechanism waits for. For values of 1, 2, 3, 4 and 5, the chances of the mechanism to detect a spurious loss event (provided that enough ACKs arrive during its operation) are then 75%, 87.5%, 93.7%, 96.8% and 98.4%, and the chances for a malicious receiver to fool the sender are 25%, 12.5%, 6.25%, 3.2% and 1.6%, respectively.

V. Simulations

We simulated the behavior of our mechanism with the ns-2 network simulator[2], using a TCP SACK sender and receiver which are interconnected with a single link. Andrei Gurtov's implementation of the Eifel algorithm was the starting point

[2]Code is available from `http://www.welzl.at/research/projects/spurious`

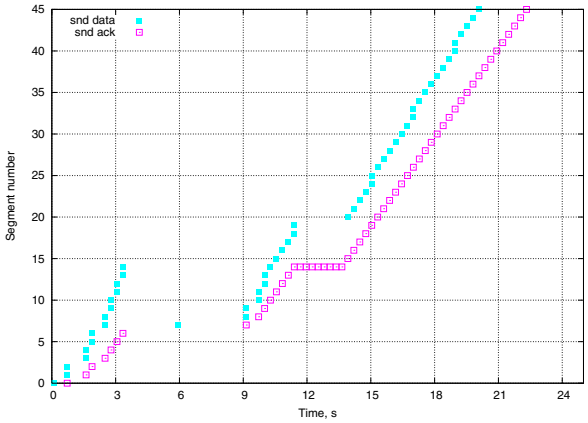

Fig. 1. Standard TCP reaction to a spurious timeout

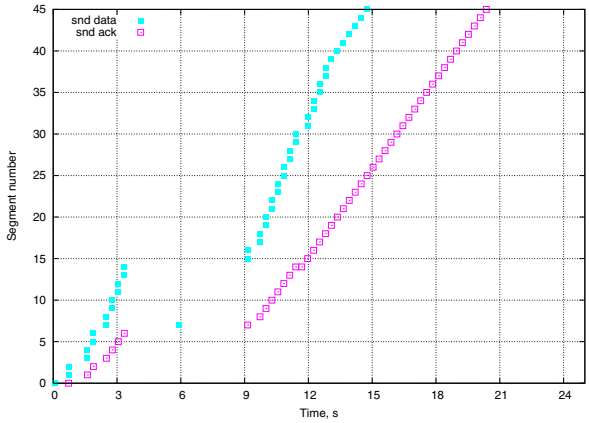

Fig. 2. ECN nonce based spurious timeout detection with $ecnsp\_param = 0$

for our code; delay spikes were generated using the implementation of the "hiccup" tool as well as the "deadline drop" patch by Andrei Gurtov. This code is available from his homepage. Since it does not support reordering (which we needed in order to cause a spurious FR/FR), we created this effect by rerouting a single packet along a second path with more delay. In our implementation it is assumed that detection always works (i.e. the nonce has the right value). Unless otherwise noted, the reaction upon detecting a spurious loss event was similar to the one implemented in the Eifel ns-2 code — the congestion window $cwnd$ and the slow start threshold $ssthresh$ were restored to their values prior to the wrongly triggered timeout or FR/FR phase.

A. Spurious timeout

Figure 1 is a time/sequence plot of TCP, showing its reaction to a spurious timeout, which we caused by generating a delay spike after 3 seconds; the timeout occurs after 5.92 seconds. At this point, 14 packets were sent and 6 packets were acknowledged. There are still 8 unacknowledged packets in the network. The sender enters the slow start phase and starts by retransmitting the first unacknowledged packet (number 7), as it is assumed that all outstanding packets are lost. After 9.14 seconds, an ACK arrives at the sender. Without a mechanism like Eifel or ours, the sender cannot properly detect whether this ACK was caused by a packet which was sent before the timeout happened; therefore, it has no significant effect. From 9.14 seconds until 10.28 seconds, the sender retransmits all outstanding packets.

The sequence of incoming ACKs that was caused by the retransmitted packets begins to arrive after 9.72 seconds. This process continues until 11.4 seconds, when an ACK for packet 14 arrives at the sender. Then, another problem happens: at 11.37 seconds the receiver generates the first of a series of 8 DupACKs, which are caused by the retransmitted packets which were in fact already received earlier. Since this is a consecutive series of DupACKs carrying increasing sequence numbers, with only one DupACK for each packet that arrived at the receiver, this behavior does not trigger FR/FR, but the sender is stalled as these ACKs do not advance the window. After 13.92 seconds all the problems are resolved and the sender can continue sending packets in the normal manner. No marking or dropping from congestion happens in our scenario — we only study the effect of abruptly delaying a series of packets or a single packet on a TCP sender.

Figure 2 shows how our ECN nonce based spurious timeout detection scheme can fix the problem in the same scenario: as before, the sender enters the slow start phase, retransmits packet number 7, and at 9.14 seconds the first ACK after the timeout arrives. This time, because the ACK carries the nonce value 1 but the nonce is always 0 for retransmitted packets, the sender is able to determine that this ACK was caused by the original packet number 7, and that the timeout was spurious. The sender can now continue sending new packets. Moreover, it can restore the state of $cwnd$ and $ssthresh$ to their last known values. After 9.72 seconds, normal ACKs for the newly sent packets arrive at the sender, i.e. the series of DupACKs in the case without spurious timeout detection does not occur.

Figure 3 shows the impact of a spurious timeout on $cwnd$ with and without our mechanism: after the timeout, the standard TCP sender immediately enters slow start. It then increases the window exponentially until half of the previous window size ($ssthresh$) is reached at 10 seconds. At 18.9 seconds, $cwnd$ has reached its original value again. With our spurious timeout detection scheme, the previous state is quickly restored and the connection can continue in slow start mode until it reaches the higher $ssthresh$ value of 20.

The impact of waiting for more than one ACK (thereby rendering our mechanism more reliable) is shown in Figures 4 and 5, respectively. While the reaction is slightly delayed, causing disturbances such as additional DupACKs arriving because of unnecessarily retransmitted packets, the performance is still quite good. This even holds for $ecnsp\_param = 3$.

Note that F-RTO (which we tested using the original code from the author's web page) reacts at the same time as our mechanism with $ecnsp\_param = 1$ in this example, as our

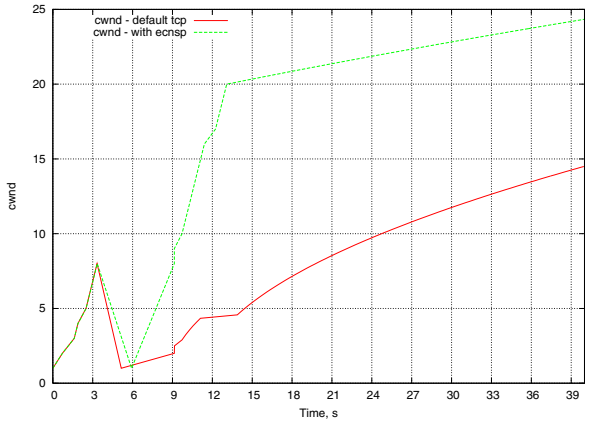

Fig. 3. Congestion window evolution with and without ECN nonce based spurious timeout detection ($ecnsp\_param = 0$)

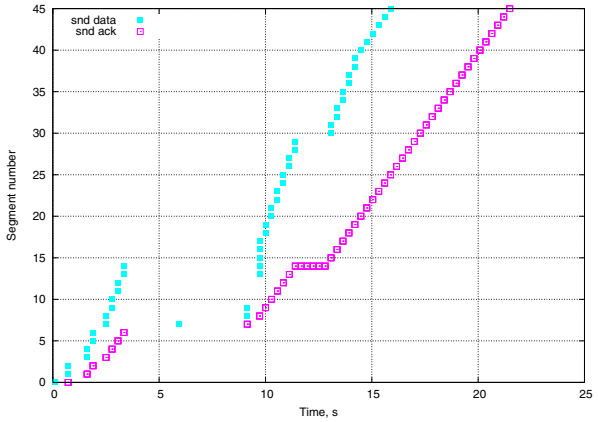

Fig. 4. ECN nonce based spurious timeout detection with $ecnsp\_param = 1$

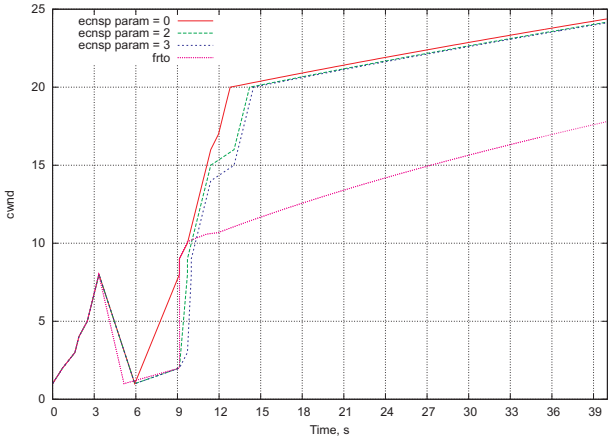

Fig. 5. Comparison of some $ecnsp\_param$ values and F-RTO

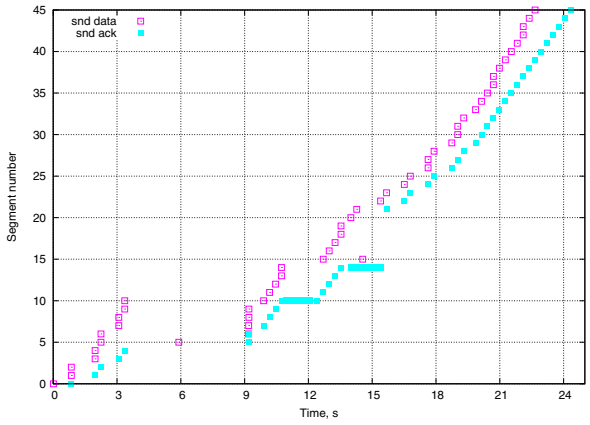

Fig. 6. Standard TCP reaction to spurious fast recovery

delay spike was designed to delay the reception of ACKs by just as much as it takes for the ACK that was caused by the first retransmitted packet to arrive before the genuine one. If the delay spike was slightly less pronounced, the speed of F-RTO would not increase but our mechanism would react faster. In this figure, TCP with F-RTO appears to enter congestion avoidance as $cwnd$ reaches 10 instead of 20. We do not have an explanation for this behavior, which should not occur with default parameters, but we stress that this is irrelevant in the context of this paper. We only consider the speed of detection and do not evaluate different recovery strategies.

Generally, as our mechanism reacts after $ecnsp\_param - 1$ incoming ACKs that correspond to packets which were sent before the spurious loss event, the reaction time of our mechanism is exactly the same as with Eifel if $ecnsp\_param = 0$. This was also confirmed by our simulations, for which we used the Eifel implementation by Andrei Gurtov.

B. Spurious fast retransmit / fast recovery

In Figure 6, our spurious timeout scenario was extended with an additional spurious FR/FR phase. Because a single packet is significantly delayed, at second 14.2 the sender retransmits it. As the plot shows, standard TCP can swiftly recover from this problem, but entering the FR/FR phase has a significant impact on $cwnd$. With our mechanism, the sender can detect that the FR/FR was spurious and accordingly react. This is shown in Figure 7.

VI. Conclusion

Unlike Eifel, which needs Timestamps, or DSACK based spurious loss event detection, our mechanism does not need any TCP options. If the ECN nonce is used, it can be deployed without changing the receiver. While F-RTO is the most easily deployable mechanism, as it does not require any special TCP mechanisms on the receiver side, its reaction can be slower than our scheme. Moreover, F-RTO (as it is currently specified in RFC 4138 [12]) cannot detect a spurious fast recovery and will fail if the first two incoming ACKs after going into slow start are DupACKs. This is no problem for our mechanism.

Clearly, ECN nonce based loss event detection cannot be seen as a replacement for F-RTO or even Eifel (which always yields the same performance as our mechanism with $ecnsp\_param = 0$), as it can miss spurious loss events depending on the value of the nonces and $ecnsp\_param$. We therefore regard our scheme as complementary to its

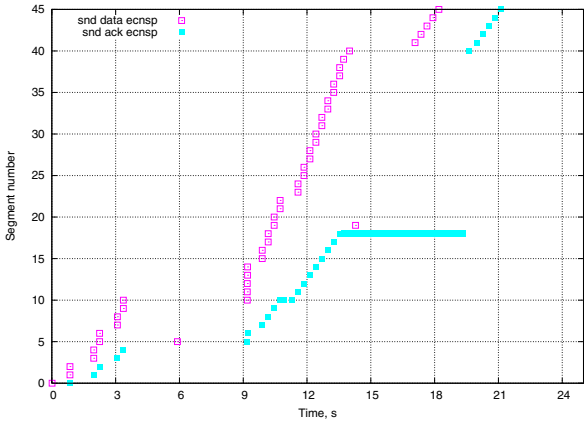

Fig. 7. ECN nonce based spurious fast recovery detection with $ecnsp\_param = 0$

alternatives. For instance, if both F-RTO and our mechanism are used, ours could improve the reaction speed and detect loss events that F-RTO misses, while F-RTO could detect spurious timeouts that are missed by the ECN nonce based detection method.

Whether the ECN nonce specification in RFC 3540 [21] should advance to Proposed Standard status or not is currently a matter of debate in the IETF. This is because of an alternate proposal [22] which would need to use the ECT codepoint in a different way than proposed in RFC 3540. While our mechanism builds upon this RFC, we do not make any claims regarding the benefits of the mechanism proposed by us as opposed to the benefits of the alternative proposal. This paper simply documents another use of the ECN nonce. However, we do believe that it may serve as useful additional input to this discussion.

We are currently working on a Linux implementation of our mechanism, which we will make available from our corresponding website.[3] We expect the outcome to shed some more light not only on the usefulness of our scheme, but also on the added-complexity-versus-benefit trade-off that must be considered for all extras that are added to the highly complicated protocol that TCP has become. Given the simplicity of our mechanism, we believe that this result will be very favorable.

ACKNOWLEDGMENTS

The authors would like to thank Thomas Raffler and Kashif Munir for their substantial contributions to this paper.

[3] http://www.welzl.at/research/projects/spurious

REFERENCES

[1] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: A reordering-robust TCP with DSACK," ICSI, Berkeley, CA, Tech. Rep. TR-02-006, July 2002.
[2] A. Gurtov and R. Ludwig, "Responding to spurious timeouts in TCP," in *IEEE Infocom*, San Francisco, CA, March 2003.
[3] R. Ludwig and A. Gurtov, "The Eifel Response Algorithm for TCP," RFC 4015 (Proposed Standard), Feb. 2005.
[4] R. Ludwig and R. H. Katz, "The Eifel algorithm: making TCP robust against spurious retransmissions," *SIGCOMM Comput. Commun. Rev.*, vol. 30, no. 1, pp. 30–36, 2000.
[5] R. Ludwig and M. Meyer, "The Eifel Detection Algorithm for TCP," RFC 3522 (Experimental), Apr. 2003.
[6] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," RFC 1323 (Proposed Standard), May 1992.
[7] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 1, pp. 66–74, 1995.
[8] 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.
[9] E. Blanton and M. Allman, "On making TCP more robust to packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 1, pp. 20–30, 2002.
[10] ——, "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.
[11] P. Sarolahti, M. Kojo, and K. Raatikainen, "F-RTO: an enhanced recovery algorithm for TCP retransmission timeouts," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 51–63, 2003.
[12] 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.
[13] K. Tan, Q. Zhang, and W. Zhu, "STODER: a robust and efficient algorithm for handling spurious retransmit timeouts in TCP," in *GLOBECOM'05*, vol. 6, no. 2, St. Louis, Missouri, USA, December 2005, pp. 3692–3696.
[14] Y.-C. Kim and D.-H. Cho, "Considering spurious timeout in proxy for improving TCP performance in wireless networks," *Elsevier Computer Networks*, vol. 44, no. 5, pp. 599–616, April 2004.
[15] K. A. Qazi, V. Kolli, and P. J. Wilhelm, "Method and apparatus for preventing a spurious retransmission after a planned interruption of communications," U.S. Patent No. 7321567, January 2008.
[16] F. Vacircaa, T. Ziegler, and E. Hasenleithner, "An algorithm to detect TCP spurious timeouts and its application to operational UMTS/GPRS networks," *Elsevier Computer Networks*, vol. 50, no. 16, pp. 2981–3001, November 2006.
[17] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, 1993.
[18] ——, "Traffic phase effects in packet-switched gateways," *SIGCOMM Comput. Commun. Rev.*, vol. 21, no. 2, pp. 26–42, 1991.
[19] K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," RFC 3168 (Proposed Standard), Sep. 2001. [Online]. Available: http://www.ietf.org/rfc/rfc3168.txt
[20] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, pp. 71–78, 1999.
[21] N. Spring, D. Wetherall, and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces," RFC 3540 (Experimental), Jun. 2003.
[22] B. Briscoe, A. Jacquet, T. Moncaster, and A. Smith, "Re-ECN: Adding accountability for causing congestion to TCP/IP," Internet-draft draft-briscoe-tsvwg-re-ecn-tcp-05, work in progress, January 2008.