Network Working Group
Internet-Draft
Intended status: Experimental
Expires: November 20, 2017

M. Bagnulo
UC3M
B. Briscoe
Simula Research Lab

May 19, 2017

ECN++: Adding Explicit Congestion Notification (ECN) to TCP Control Packets
draft-bagnulo-tcpm-generalized-ecn-04

Abstract

This document describes an experimental modification to ECN when used with TCP. It allows the use of ECN on the following TCP packets: SYNs, pure ACKs, Window probes, FINs, RSTs and retransmissions.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 20, 2017.

Copyright Notice

Table of Contents

1. Introduction

RFC 3168 [RFC3168] specifies support of Explicit Congestion Notification (ECN) in IP (v4 and v6). By using the ECN capability, switches performing Active Queue Management (AQM) can use ECN marks instead of packet drops to signal congestion to the endpoints of a communication. This results in lower packet loss and increased performance. RFC 3168 also specifies support for ECN in TCP, but solely on data packets. For various reasons it precludes the use of ECN on TCP control packets (TCP SYN, TCP SYN-ACK, pure ACKs, Window probes) and on retransmitted packets. RFC 3168 is silent about the use of ECN on RST and FIN packets. RFC 5562 [RFC5562] is an experimental modification to ECN that enables ECN support for TCP SYN-ACK packets.

This document defines an experimental modification to ECN [RFC3168] that enables ECN support on all the aforementioned types of TCP packet. [I-D.ietf-tsvwg-ecn-experimentation] is a standards track procedural device that relaxes standards track requirements in RFC 3168 that would otherwise preclude these experimental modifications.

The present document also considers the implications for common derivatives and variants of TCP, such as SCTP [RFC4960], if the experiment is successful. One particular variant of TCP adds accurate ECN feedback (AccECN [I-D.ietf-tcpm-accurate-ecn]), without which ECN support cannot be added to SYNs. Nonetheless, ECN support can be added to all the other types of TCP packet whether or not AccECN is also supported.

1.1. Motivation

The absence of ECN support on TCP control packets and retransmissions has a potential harmful effect. In any ECN deployment, non-ECN-capable packets suffer a penalty when they traverse a congested bottleneck. For instance, with a drop probability of 1%, 1% of connection attempts suffer a timeout of about 1 second before the SYN is retransmitted, which is highly detrimental to the performance of short flows. TCP control packets, such as TCP SYNs and pure ACKs, are important for performance, so dropping them is best avoided.

Non-ECN control packets particularly harm performance in environments where the ECN marking level is high. For example, [judd-nsdi] shows that in a data centre (DC) environment where ECN is used (in conjunction with DCTCP), the probability of being able to establish a new connection using a non-ECN SYN packet drops to close to zero even when there are only 16 ongoing TCP flows transmitting at full speed. In this data centre context, the issue is that DCTCP’s aggressive response to packet marking leads to a high marking probability for

ECN-capable packets, and in turn a high drop probability for non-ECN packets. Therefore non-ECN SYNs are dropped aggressively, rendering it nearly impossible to establish a new connection in the presence of even mild traffic load.

Finally, there are ongoing experimental efforts to promote the adoption of a slightly modified variant of DCTCP (and similar congestion controls) over the Internet to achieve low latency, low loss and scalable throughput (L4S) for all communications [I-D.briscoe-tsvwg-l4s-arch]. In such an approach, L4S packets identify themselves using an ECN codepoint. With L4S and potentially other similar cases, preventing TCP control packets from obtaining the benefits of ECN would not only expose them to the prevailing level of congestion loss, but it would also classify control packet into a different queue with different network treatment, which may also lead to reordering, further degrading TCP performance.

1.2. Experiment Goals

The goal of the experimental modifications defined in this document is to allow the use of ECN on all TCP packets. Experiments are expected in the public Internet as well as in controlled environments to understand the following issues:

- How SYNs, Window probes, pure ACKs, FINs, RSTs and retransmissions that carry the ECT(0), ECT(1) or CE codepoints are processed by the TCP endpoints and the network (including routers, firewalls and other middleboxes). In particular we would like to learn if these packets are frequently blocked or if these packets are usually forwarded and processed.

- The scale of deployment of the different flavours of ECN, including [RFC3168], [RFC5562], [RFC3540] and [I-D.ietf-tcpm-accurate-ecn].

- How much the performance of TCP communications is improved by allowing ECN marking of each packet type.

- To identify any issues (including security issues) raised by enabling ECN marking of these packets.

The data gathered through the experiments described in this document, particularly under the first 2 bullets above, will help in the design of the final mechanism (if any) for adding ECN support to the different packet types considered in this document. Whenever data input is needed to assist in a design choice, it is spelled out throughout the document.

Success criteria: The experiment will be a success if we obtain enough data to have a clearer view of the deployability and benefits of enabling ECN on all TCP packets, as well as any issues. If the results of the experiment show that it is feasible to deploy such changes; that there are gains to be achieved through the changes described in this specification; and that no other major issues may interfere with the deployment of the proposed changes; then it would be reasonable to adopt the proposed changes in a standards track specification that would update RFC 3168.

1.3. Document Structure

The remainder of this document is structured as follows. In Section 2, we present the terminology used in the rest of the document. In Section 3, we specify the modifications to provide ECN support to TCP SYNs, pure ACKs, Window probes, FINs, RSTs and retransmissions. We describe both the network behaviour and the endpoint behaviour. Section 5 discusses variations of the specification that will be necessary to interwork with a number of popular variants or derivatives of TCP. RFC 3168 provides a number of specific reasons why ECN support is not appropriate for each packet type. In Section 4, we revisit each of these arguments for each packet type to justify why it is reasonable to conduct this experiment.

2. Terminology

The keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in this document, are to be interpreted as described in [RFC2119].

Pure ACK: A TCP segment with the ACK flag set and no data payload.

SYN: A TCP segment with the SYN (synchronize) flag set.

Window probe: Defined in [RFC0793], a window probe is a TCP segment with only one byte of data sent to learn if the receive window is still zero.

FIN: A TCP segment with the FIN (finish) flag set.

RST: A TCP segment with the RST (reset) flag set.

Retransmission: A TCP segment that has been retransmitted by the TCP sender.

ECT: ECN-Capable Transport. One of the two codepoints ECT(0) or ECT(1) in the ECN field [RFC3168] of the IP header (v4 or v6). An

ECN-capable sender sets one of these to indicate that both transport end-points support ECN. When this specification says the sender sets an ECT codepoint, by default it means ECT(0). Optionally, it could mean ECT(1), which is in the process of being redefined for use by L4S experiments [I-D.ietf-tsvwg-ecn-experimentation] [I-D.briscoe-tsvwg-ecn-l4s-id].

Not-ECT: The ECN codepoint set by senders that indicates that the transport is not ECN-capable.

CE: Congestion Experienced. The ECN codepoint that an intermediate node sets to indicate congestion [RFC3168]. A node sets an increasing proportion of ECT packets to CE as the level of congestion increases.

3. Specification

3.1. Network (e.g. Firewall) Behaviour

Previously the specification of ECN for TCP [RFC3168] required the sender to set not-ECT on TCP control packets and retransmissions. Some readers of RFC 3168 might have erroneously interpreted this as a requirement for firewalls, intrusion detection systems, etc. to check and enforce this behaviour. Section 4.3 of [I-D.ietf-tsvwg-ecn-experimentation] updates RFC 3168 to remove this ambiguity. It require firewalls or any intermediate nodes not to treat certain types of ECN-capable TCP segment differently (except potentially in one attack scenario). This is likely to only involve a firewall rule change in a fraction of cases (at most 0.4% of paths according to the tests reported in Section 4.2.2).

In case a TCP sender encounters a middlebox blocking ECT on certain TCP segments, the specification below includes behaviour to fall back to non-ECN. However, this loses the benefit of ECN on control packets. So operators are RECOMMENDED to alter their firewall rules to comply with the requirement referred to above (section 4.3 of [I-D.ietf-tsvwg-ecn-experimentation]).

3.2. Endpoint Behaviour

The changes to the specification of TCP over ECN [RFC3168] defined here solely alter the behaviour of the sending host for each half-connection. All changes can be deployed at each end-point independently of others.

The feedback behaviour at the receiver depends on whether classic ECN TCP feedback [RFC3168] or Accurate ECN (AccECN) TCP feedback [I-D.ietf-tcpm-accurate-ecn] has been negotiated. Nonetheless,

neither receiver feedback behaviour is altered by the present specification.

For each type of control packet or retransmission, the following sections detail changes to the sender's behaviour in two respects: i) whether it sets ECT; and ii) its response to congestion feedback. Table 1 summarises these two behaviours for each type of packet, but the relevant subsection below should be referred to for the detailed behaviour. The subsection on the SYN is more complex than the others, because it has to include fall-back behaviour if the ECT packet appears not to have got through, and caching of the outcome to detect persistent failures.

TCP packet type	ECN field if AccECN f/b negotiated*	ECN field if RFC3168 f/b negotiated*	Congestion Response
SYN	ECT	not-ECT	Reduce IW
SYN-ACK [RFC5562]	ECT	ECT	Reduce IW as in [RFC5562]
Pure ACK	ECT	ECT	Usual cwnd response and optionally [RFC5690]
W Probe	ECT	ECT	Usual cwnd response
FIN	ECT	ECT	None or optionally [RFC5690]
RST	ECT	ECT	N/A
Re-XMT	ECT	ECT	Usual cwnd response

Window probe and retransmission are abbreviated to W Probe an Re-XMT.
* For a SYN, "negotiated" means "requested".

Table 1: Summary of sender behaviour. In each case the relevant section below should be referred to for the detailed behaviour

It can be seen that the sender can set ECT in all cases, except if it is not requesting AccECN feedback on the SYN. Therefore it is RECOMMENDED that the experimental AccECN specification [I-D.ietf-tcpm-accurate-ecn] is implemented (as well as the present specification), because it is expected that ECT on the SYN will give the most significant performance gain, particularly for short flows. Nonetheless, this specification also caters for the case where AccECN feedback is not implemented.

3.2.1. SYN

3.2.1.1. Setting ECT on the SYN

With classic [RFC3168] ECN feedback, the SYN was never expected to be ECN-capable, so the flag provided to feed back congestion was put to another use (it is used in combination with other flags to indicate that the responder supports ECN). In contrast, Accurate ECN (AccECN) feedback [I-D.ietf-tcpm-accurate-ecn] provides two codepoints in the SYN-ACK for the responder to feed back whether or not the SYN arrived marked CE.

Therefore, a TCP initiator MUST NOT set ECT on a SYN unless it also attempts to negotiate Accurate ECN feedback in the same SYN.

For the experiments proposed here, if the SYN is requesting AccECN feedback, the TCP sender will also set ECT on the SYN. It can ignore the prohibition in section 6.1.1 of RFC 3168 against setting ECT on such a SYN.

The following subsections about the SYN solely apply to this case where the initiator sent an ECT SYN.

> MEASUREMENTS NEEDED: Measurements are needed to verify that if SYN packets with the ECT(0)/ECT(1)/CE codepoints are properly delivered by the network. We need to learn if there are cases if SYN packets are dropped because having the the ECT(0)/ECT(1)/CE codepoints. We also need to learn if the network clears SYN packet with the the ECT(0)/ECT(1)/CE codepoints. In addition, we need measurements to learn how current deployed base of servers react to SYN packets with ECT(0)/ECT(1)/CE codepoints whether they discard it, or process it an return a SYN/ACK packet proceeding with the connection. It would be also useful to measure how the network elements and the servers react to all possible combinations of ECN codepoints and NS/CWR/ECE flags.

3.2.1.2. Caching Lack of Support for ECT on SYNs

Until AccECN servers become widely deployed, a TCP initiator that sets ECT on a SYN (which implies the same SYN also requests AccECN, as required above) SHOULD also maintain a cache per server to record any failure of previous attempts.

The initiator will record any server's SYN-ACK response that does not support AccECN. Subsequently the initiator will not set ECT on a SYN to such a server, but it can still always request AccECN support (because the response will state any earlier stage of ECN evolution that the server supports with no performance penalty). The initiator will discover a server that has upgraded to support AccECN as soon as it next connects, then it can remove the server from its cache and subsequently always set ECT for that server.

If the initiator times out without seeing a SYN-ACK, it will also cache this fact (see fall-back in Section 3.2.1.4 for details).

There is no need to cache successful attempts, because the default ECT SYN behaviour performs optimally on success anyway. Servers that do not support ECN as a whole probably do not need to be recorded separately from non-support of AccECN because the response to a request for AccECN immediately states which stage in the evolution of ECN the server supports (AccECN [I-D.ietf-tcpm-accurate-ecn], classic ECN [RFC3168] or no ECN).

The above strategy is named "optimistic ECT and cache failures". It is believed to be sufficient based on initial measurements and assumptions detailed in Section 4.2.1, which also gives alternative strategies in case larger scale measurements uncover different scenarios.

3.2.1.3. SYN Congestion Response

If the SYN-ACK returned to the TCP initiator confirms that the server supports AccECN, it will also indicate whether or not the SYN was CE-marked. If the SYN was CE-marked, the initiator MUST reduce its Initial Window (IW) and SHOULD reduce it to 1 SMSS (sender maximum segment size).

If the SYN-ACK shows that the server does not support AccECN, the TCP initiator MUST conservatively reduce its Initial Window and SHOULD reduce it to 1 SMSS. A reduction to greater than 1 SMSS MAY be appropriate (see Section 4.2.1). Conservatism is necessary because a non-AccECN SYN-ACK cannot show whether the SYN was CE-marked.

If the TCP initiator (host A) receives a SYN from the remote end (host B) after it has sent a SYN to B, it indicates the (unusual) case of a simultaneous open. Host A will respond with a SYN-ACK. Host A will probably then receive a SYN-ACK in response to its own SYN, after which it can follow the appropriate one of the two paragraphs above.

In all the above cases, the initiator does not have to back off its retransmission timer as it would in response to a timeout following no response to its SYN [RFC6298], because both the SYN and the SYN-ACK have been successfully delivered through the network. Also, the initiator does not need to exit slow start or reduce ssthresh, which is not even required when a SYN is lost [RFC5681].

If an initial window of 10 (IW10 [RFC6928]) is implemented, Section 5 gives additional recommendations.

3.2.1.4. Fall-Back Following No Response to an ECT SYN

An ECT SYN might be lost due to an over-zealous path element (or server) blocking ECT packets that do not conform to RFC 3168. However, loss is commonplace for numerous other reasons, e.g. congestion loss at a non-ECN queue on the forward or reverse path, transmission errors, etc. Alternatively, the cause of the blockage might be the attempt to negotiate AccECN, or possibly other unrelated options on the SYN.

To expedite connection set-up if, after sending an ECT SYN, the retransmission timer expires, the TCP initiator SHOULD send a SYN with the not-ECT codepoint in the IP header. If other experimental fields or options were on the SYN, it will also be necessary to follow their specifications for fall-back too. It would make sense to co- ordinate all the strategies for fall-back in order to isolate the specific cause of the problem.

If the TCP initiator is caching failed connection attempts, it SHOULD NOT give up using ECT on the first SYN of subsequent connection attempts until it is clear that the blockage persistently and specifically affects ECT on SYNs. This is because loss is so commonplace for other reasons. Even if it does eventually decide to give up on ECT on the SYN, it will probably not need to give up on AccECN on the SYN. In any case, the cache should be arranged to expire so that the initiator will infrequently attempt to check whether the problem has been resolved.

Other fall-back strategies MAY be adopted where applicable (see Section 4.2.2 for suggestions, and the conditions under which they would apply).

3.2.2. SYN-ACK

3.2.2.1. Setting ECT on the SYN-ACK

For the experiments proposed here, the TCP implementation will set ECT on SYN-ACKs. It can ignore the requirement in section 6.1.1 of RFC 3168 to set not-ECT on a SYN-ACK.

The feedback behaviour by the initiator in response to a CE-marked SYN-ACK from the responder depends on whether classic ECN feedback [RFC3168] or AccECN feedback [I-D.ietf-tcpm-accurate-ecn] has been negotiated. In either case no change is required to RFC 3168 or the AccECN specification.

Some classic ECN implementations might ignore a CE-mark on a SYN-ACK, or even ignore a SYN-ACK packet entirely if it is set to ECT or CE. This is a possibility because an RFC 3168 implementation would not necessarily expect a SYN-ACK to be ECN-capable.

> FOR DISCUSSION: To eliminate this problem, the WG could decide to prohibit setting ECT on SYN-ACKs unless AccECN has been negotiated. However, this issue already came up when the IETF first decided to experiment with ECN on SYN-ACKs [RFC5562] and it was decided to go ahead without any extra precautionary measures because the risk was low. This was because the probability of encountering the problem was believed to be low and the harm if the problem arose was also low (see Appendix B of RFC 5562).
>
> MEASUREMENTS NEEDED: Server-side experiments could determine whether this specific problem is indeed rare across the current installed base of clients that support ECN.

3.2.2.2. SYN-ACK Congestion Response

A host that sets ECT on SYN-ACKs MUST reduce its initial window in response to any congestion feedback, whether using classic ECN or AccECN. It SHOULD reduce it to 1 SMSS. This is different to the behaviour specified in an earlier experiment that set ECT on the SYN-ACK [RFC5562]. This is justified in Section 4.3.

The responder does not have to back off its retransmission timer because the ECN feedback proves that the network is delivering packets successfully and is not severely overloaded. Also the responder does not have to leave slow start or reduce ssthresh, which is not even required when a SYN-ACK has been lost.

The congestion response to CE-marking on a SYN-ACK for a server that implements either the TCP Fast Open experiment (TFO [RFC7413]) or the

initial window of 10 experiment (IW10 [RFC6928]) is discussed in Section 5.

3.2.2.3. Fall-Back Following No Response to an ECT SYN-ACK

After the responder sends a SYN-ACK with ECT set, if its retransmission timer expires it SHOULD resend a SYN-ACK with not-ECT set. If other experimental fields or options were on the SYN, it will also be necessary to follow their specifications for fall-back too. It would make sense to co-ordinate all the strategies for fall-back in order to isolate the specific cause of the problem.

The server MAY cache failed connection attempts, e.g. per client access network. If the TCP server is caching failed connection attempts, it SHOULD NOT give up using ECT on the first SYN-ACK of subsequent connection attempts until it is clear that the blockage persistently and specifically affects ECT on SYN-ACKs. This is because loss is so commonplace for other reasons (see Section 3.2.1.4). The cache should be arranged to expire so that the server will infrequently attempt to check whether the problem has been resolved.

This fall-back strategy is the same as that for ECT SYN-ACKs in [RFC5562]. Other fall-back strategies MAY be adopted if found to be more effective, e.g. one retransmission attempt using ECT before reverting to not-ECT.

3.2.3. Pure ACK

For the experiments proposed here, the TCP implementation will set ECT on pure ACKs. It can ignore the requirement in section 6.1.4 of RFC 3168 to set not-ECT on a pure ACK.

A host that sets ECT on pure ACKs MUST reduce its congestion window in response to any congestion feedback, in order to regulate any data segments it might be sending amongst the pure ACKs. It MAY also implement AckCC [RFC5690] to regulate the pure ACK rate, but this is not required. Note that, in comparison, TCP Congestion Control [RFC5681] does not require a TCP to detect or respond to loss of pure ACKs at all; it requires no reduction in congestion window or ACK rate.

The question of whether the receiver of pure ACKs is required to feed back any CE marks on them is a matter for the relevant feedback specification ([RFC3168] or [I-D.ietf-tcpm-accurate-ecn]). It is outside the scope of the present specification. Currently AccECN feedback is required to count CE marking of any control packet including pure ACKs. Whereas RFC 3168 is silent on this point, so

feedback of CE-markings might be implementation specific (see Section 4.4.1).

DISCUSSION: An AccECN deployment or an implementation of RFC 3168 that feeds back CE on pure ACKs will be at a disadvantage compared to an RFC 3168 implementation that does not. To solve this, the WG could decide to prohibit setting ECT on pure ACKs unless AccECN has been negotiated. If it does, the penultimate sentence of the Introduction will need to be modified.

MEASUREMENTS NEEDED: Measurements are needed to learn how the deployed base of network elements and servers react to pure ACKs marked with the ECT(0)/ECT(1)/CE codepoints, i.e. whether they are dropped, codepoint cleared or processed.

3.2.4. Window Probe

For the experiments proposed here, the TCP sender will set ECT on window probes. It can ignore the prohibition in section 6.1.6 of RFC 3168 against setting ECT on a window probe.

A window probe contains a single octet, so it is no different from a regular TCP data segment. Therefore a TCP receiver will feed back any CE marking on a window probe as normal (either using classic ECN feedback or AccECN feedback). The sender of the probe will then reduce its congestion window as normal.

A receive window of zero indicates that the application is not consuming data fast enough and does not imply anything about network congestion. Once the receive window opens, the congestion window might become the limiting factor, so it is correct that CE-marked probes reduce the congestion window. However, CE-marking on window probes does not reduce the rate of the probes themselves. This is unlikely to present a problem, given the duration between window probes doubles [RFC1122] as long as the receiver is advertising a zero window (currently minimum 1 second, maximum at least 1 minute [RFC6298]).

MEASUREMENTS NEEDED: Measurements are needed to learn how the deployed base of network elements and servers react to Window probes marked with the ECT(0)/ECT(1)/CE codepoints, i.e. whether they are dropped, codepoint cleared or processed.

3.2.5. FIN

A TCP implementation can set ECT on a FIN.

The TCP data receiver MUST ignore the CE codepoint on incoming FINs that fail any validity check. The validity check in section 5.2 of [RFC5961] is RECOMMENDED.

A congestion response to a CE-marking on a FIN is not required.

After sending a FIN, the endpoint will not send any more data in the connection. Therefore, even if the FIN-ACK indicates that the FIN was CE-marked (whether using classic or AccECN feedback), reducing the congestion window will not affect anything.

After sending a FIN, a host might send one or more pure ACKs. If it is using one of the techniques in Section 3.2.3 to regulate the delayed ACK ratio for pure ACKs, it could equally be applied after a FIN. But this is not required.

> MEASUREMENTS NEEDED: Measurements are needed to learn how the deployed base of network elements and servers react to FIN packets marked with the ECT(0)/ECT(1)/CE codepoints, i.e. whether they are dropped, codepoint cleared or processed.

3.2.6. RST

A TCP implementation can set ECT on a RST.

The "challenge ACK" approach to checking the validity of RSTs (section 3.2 of [RFC5961] is RECOMMENDED at the data receiver.

A congestion response to a CE-marking on a RST is not required (and actually not possible).

> MEASUREMENTS NEEDED: Measurements are needed to learn how the deployed base of network elements and servers react to RST packets marked with the ECT(0)/ECT(1)/CE codepoints, i.e. whether they are dropped, codepoint cleared or processed.

3.2.7. Retransmissions

For the experiments proposed here, the TCP sender will set ECT on retransmitted segments. It can ignore the prohibition in section 6.1.5 of RFC 3168 against setting ECT on retransmissions.

Nonetheless, the TCP data receiver MUST ignore the CE codepoint on incoming segments that fail any validity check. The validity check in section 5.2 of [RFC5961] is RECOMMENDED. This will effectively mitigate an attack that uses spoofed data packets to fool the receiver into feeding back spoofed congestion indications to the

sender, which in turn would be fooled into continually halving its congestion window.

If the TCP sender receives feedback that a retransmitted packet was CE-marked, it will react as it would to any feedback of CE-marking on a data packet.

> MEASUREMENTS NEEDED: Measurements are needed to learn how the deployed base of network elements and servers react to retransmissions marked with the ECT(0)/ECT(1)/CE codepoints, i.e. whether they are dropped, codepoint cleared or processed.

4. Rationale

This section is informative, not normative. It presents counter-arguments against the justifications in the RFC series for disabling ECN on TCP control segments and retransmissions. It also gives rationale for why ECT is safe on control segments that have not, so far, been mentioned in the RFC series. First it addresses over-arching arguments used for most packet types, then it addresses the specific arguments for each packet type in turn.

4.1. The Reliability Argument

Section 5.2 of RFC 3168 states:

> "To ensure the reliable delivery of the congestion indication of the CE codepoint, an ECT codepoint MUST NOT be set in a packet unless the loss of that packet [at a subsequent node] in the network would be detected by the end nodes and interpreted as an indication of congestion."

We believe this argument is misplaced. TCP does not deliver most control packets reliably. So it is more important to allow control packets to be ECN-capable, which greatly improves reliable delivery of the control packets themselves (see motivation in Section 1.1). ECN also improves the reliability and latency of delivery of any congestion notification on control packets, particularly because TCP does not detect the loss of most types of control packet anyway. Both these points outweigh by far the concern that a CE marking applied to a control packet by one node might subsequently be dropped by another node.

The principle to determine whether a packet can be ECN-capable ought to be "do no extra harm", meaning that the reliability of a congestion signal’s delivery ought to be no worse with ECN than without. In particular, setting the CE codepoint on the very same packet that would otherwise have been dropped fulfills this

criterion, since either the packet is delivered and the CE signal is delivered to the endpoint, or the packet is dropped and the original congestion signal (packet loss) is delivered to the endpoint.

The concern about a CE marking being dropped at a subsequent node might be motivated by the idea that ECN-marking a packet at the first node does not remove the packet, so it could go on to worsen congestion at a subsequent node. However, it is not useful to reason about congestion by considering single packets. The departure rate from the first node will generally be the same (fully utilized) with or without ECN, so this argument does not apply.

4.2. SYNs

RFC 5562 presents two arguments against ECT marking of SYN packets (quoted verbatim):

> "First, when the TCP SYN packet is sent, there are no guarantees that the other TCP endpoint (node B in Figure 2) is ECN-Capable, or that it would be able to understand and react if the ECN CE codepoint was set by a congested router.
>
> Second, the ECN-Capable codepoint in TCP SYN packets could be misused by malicious clients to "improve" the well-known TCP SYN attack. By setting an ECN-Capable codepoint in TCP SYN packets, a malicious host might be able to inject a large number of TCP SYN packets through a potentially congested ECN-enabled router, congesting it even further."

The first point actually describes two subtly different issues. So below three arguments are countered in turn.

4.2.1. Argument 1a: Unrecognized CE on the SYN

This argument certainly applied at the time RFC 5562 was written, when no ECN responder mechanism had any logic to recognize or feed back a CE marking on a SYN. The problem was that, during the 3WHS, the flag in the TCP header for ECN feedback (called Echo Congestion Experienced) had been overloaded to negotiate the use of ECN itself. So there was no space for feedback in a SYN-ACK.

The accurate ECN (AccECN) protocol [I-D.ietf-tcpm-accurate-ecn] has since been designed to solve this problem, using a two-pronged approach. First AccECN uses the 3 ECN bits in the TCP header as 8 codepoints, so there is space for the responder to feed back whether there was CE on the SYN. Second a TCP initiator can always request AccECN support on every SYN, and any responder reveals its level of ECN support: AccECN, classic ECN, or no ECN. Therefore, if a

responder does indicate that it supports AccECN, the initiator can be sure that, if there is no CE feedback on the SYN-ACK, then there really was no CE on the SYN.

An initiator can combine AccECN with three possible strategies for setting ECT on a SYN:

(S1): Pessimistic ECT and cache successes: The initiator always requests AccECN in the SYN, but without setting ECT. Then it records those servers that confirm that they support AccECN in a cache. On a subsequent connection to any server that supports AccECN, the initiator can then set ECT on the SYN.

(S2): Optimistic ECT: The initiator always sets ECT optimistically on the initial SYN and it always requests AccECN support. Then, if the server response shows it has no AccECN logic (so it cannot feed back a CE mark), the initiator conservatively behaves as if the SYN was CE-marked, by reducing its initial window.

A. No cache: The optimistic ECT strategy ought to work fairly well without caching any responses.

B. Cache failures: The optimistic ECT strategy can be improved by recording solely those servers that do not support AccECN. On subsequent connections to these non-AccECN servers, the initiator will still request AccECN but not set ECT on the SYN. Then, the initiator can use its full initial window (if it has enough request data to need it). Longer term, as servers upgrade to AccECN, the initiator will remove them from the cache and use ECT on subsequent SYNs to that server.

(S3): ECT by configuration: In a controlled environment, the administrator can make sure that servers support ECN-capable SYN packets. Examples of controlled environments are single-tenant DCs, and possibly multi-tenant DCs if it is assumed that each tenant mostly communicates with its own VMs.

For unmanaged environments like the public Internet, pragmatically the choice is between strategies (S1) and (S2B):

- The "pessimistic ECT and cache successes" strategy (S1) suffers from exposing the initial SYN to the prevailing loss level, even if the server supports ECT on SYNs, but only on the first connection to each AccECN server.

- The "optimistic ECT and cache failures" strategy (S2B) exploits a server's support for ECT on SYNs from the very first attempt. But if the server turns out not to support AccECN, the initiator has to conservatively limit its initial window - usually unnecessarily. Nonetheless, initiator request data (as opposed to server response data) is rarely larger than 1 SMSS anyway {ToDo: reference? (this information was given informally by Yuchung Cheng)}.

The normative specification for ECT on a SYN in Section 3.2.1 uses the "optimistic ECT and cache failures" strategy (S2B) on the assumption that an initial window of 1 SMSS is usually sufficient for client requests anyway. Clients that often initially send more than 1 SMSS of data could use strategy (S1) during initial deployment, and strategy (S2B) later (when the probability of servers supporting AccECN and the likelihood of seeing some CE marking is higher). Also, as deployment proceeds, caching successes (S1) starts off small then grows, while caching failures (S2B) becomes large at first, then shrinks.

> MEASUREMENTS NEEDED: Measurements are needed to determine whether one or the other strategy would be sufficient for any particular client, or whether a particular client would need both strategies in different circumstances.

4.2.2. Argument 1b: Unrecognized ECT on the SYN

Given, until now, ECT-marked SYN packets have been prohibited, it cannot be assumed they will be accepted. According to a study using 2014 data [ecn-pam] from a limited range of vantage points, out of the top 1M Alexa web sites, 4791 (0.82%) IPv4 sites and 104 (0.61%) IPv6 sites failed to establish a connection when they received a TCP SYN with any ECN codepoint set in the IP header and the appropriate ECN flags in the TCP header. Of these, about 41% failed to establish a connection due to the ECN flags in the TCP header even with a Not-ECT ECN field in the IP header (i.e. despite full compliance with RFC 3168). Therefore adding the ECN-capability to SYNs was increasing connection establishment failures by about 0.4%.

> MEASUREMENTS NEEDED: In order to get these failures fixed, data will be needed on which of the possible causes below is behind them.

RFC 3168 says "a host MUST NOT set ECT on SYN [...] packets", but it does not say what the responder should do if an ECN-capable SYN arrives. So perhaps some responder implementations are checking that the SYN complies with RFC 3168, then silently ignoring non-compliant SYNs (or perhaps returning a RST). Also some middleboxes (e.g.

firewalls) might be discarding non-compliant SYNs. For the future, [I-D.ietf-tsvwg-ecn-experimentation] updates RFC 3168 to clarify that middleboxes "SHOULD NOT" do this, but that does not alter the past.

Whereas RSTs can be dealt with immediately, silent failures introduce a retransmission timeout delay (default 1 second) at the initiator before it attempts any fall back strategy. Ironically, making SYNs ECN-capable is intended to avoid the timeout when a SYN is lost due to congestion. Fortunately, where discard of ECN-capable SYNs is due to policy it will occur predictably, not randomly like congestion. So the initiator can avoid it by caching those sites that do not support ECN-capable SYNs. This further justifies the use of the "optimistic ECT and cache failures" strategy in Section 3.2.1.

> MEASUREMENTS NEEDED: Experiments are needed to determine whether blocking of ECT on SYNs is widespread, and how many occurrences of problems would be masked by how few cache entries.

If blocking is too widespread for the "optimistic ECT and cache failures" strategy (S2B), the "pessimistic ECT and cache successes" strategy (Section 4.2.1) would be better.

> MEASUREMENTS NEEDED: Then measurements would be needed on whether failures were still widespread on the second connection attempt after the more careful ("pessimistic") first connection.

If so, it might be necessary to send a not-ECT SYN soon after the first ECT SYN (possibly with a delay between them - effectively reducing the retransmission timeout) and only accept the non-ECT connection if it returned first. This would reduce the performance penalty for those deploying ECT SYN support.

> FOR DISCUSSION: If this becomes necessary, how much delay ought to be required before the second SYN? Certainly less than the standard RTO (1 second). But more or less than the maximum RTT expected over the surface of the earth (roughly 250ms)? Or even back-to-back?

However, based on the data above from [ecn-pam], even a cache of a dozen or so sites ought to avoid all ECN-related performance problems with roughly the Alexa top thousand. So it is questionable whether sending two SYNs will be necessary, particularly given failures at well-maintained sites could reduce further once ECT SYNs are standardized.

4.2.3. Argument 2: DoS Attacks

[RFC5562] says that ECT SYN packets could be misused by malicious clients to augment "the well-known TCP SYN attack". It goes on to say "a malicious host might be able to inject a large number of TCP SYN packets through a potentially congested ECN-enabled router, congesting it even further."

We assume this is a reference to the TCP SYN flood attack (see https://en.wikipedia.org/wiki/SYN_flood), which is an attack against a responder end point. We assume the idea of this attack is to use ECT to get more packets through an ECN-enabled router in preference to other non-ECN traffic so that they can go on to use the SYN flooding attack to inflict more damage on the responder end point. This argument could apply to flooding with any type of packet, but we assume SYNs are singled out because their source address is easier to spoof, whereas floods of other types of packets are easier to block.

Mandating Not-ECT in an RFC does not stop attackers using ECT for flooding. Nonetheless, if a standard says SYNs are not meant to be ECT it would make it legitimate for firewalls to discard them. However this would negate the considerable benefit of ECT SYNs for compliant transports and seems unnecessary because RFC 3168 already provides the means to address this concern. In section 7, RFC 3168 says "During periods where ... the potential packet marking rate would be high, our recommendation is that routers drop packets rather then set the CE codepoint..." and this advice is repeated in [RFC7567] (section 4.2.1). This makes it harder for flooding packets to gain from ECT.

Further experiments are needed to test how much malicious hosts can use ECT to augment flooding attacks without triggering AQMs to turn off ECN support (flying "just under the radar"). If it is found that ECT can only slightly augment flooding attacks, the risk of such attacks will need to be weighed against the performance benefits of ECT SYNs.

4.3. SYN-ACKs

The proposed approach in Section 3.2.2 for experimenting with ECN-capable SYN-ACKs is identical to the scheme called ECN+ [ECN-PLUS]. In 2005, the ECN+ paper demonstrated that it could reduce the average Web response time by an order of magnitude. It also argued that adding ECT to SYN-ACKs did not raise any new security vulnerabilities.

The IETF has already specified an experiment with ECN-capable SYN-ACK packets [RFC5562]. It was inspired by the ECN+ paper, but it

specified a much more conservative congestion response to a CE-marked SYN-ACK, called ECN+/TryOnce. This required the server to reduce its initial window to 1 segment (like ECN+), but then the server had to send a second SYN-ACK and wait for its ACK before it could continue with its initial window of 1 SMSS. The second SYN-ACK of this 5-way handshake had to carry no data, and had to disable ECN, but no justification was given for these last two aspects.

The present ECN experiment uses the ECN+ congestion response, not ECN+/TryOnce. First we argue against the rationale for ECN+/TryOnce given in sections 4.4 and 6.2 of [RFC5562]. It starts with a rather too literal interpretation of the requirement in RFC 3168 that says TCP's response to a single CE mark has to be "essentially the same as the congestion control response to a *single* dropped packet." TCP's response to a dropped initial (SYN or SYN-ACK) packet is to wait for the retransmission timer to expire (currently 1s). However, this long delay assumes the worst case between two possible causes of the loss: a) heavy overload; or b) the normal capacity-seeking behaviour of other TCP flows. When the network is still delivering CE-marked packets, it implies that there is an AQM at the bottleneck and that it is not overloaded. This is because an AQM under overload will disable ECN (as recommended in section 7 of RFC 3168 and repeated in section 4.2.1 of RFC 7567). So scenario (a) can be ruled out. Therefore, TCP's response to a CE-marked SYN-ACK can be similar to its response to the loss of _any_ packet, rather than backing off as if the special _initial_ packet of a flow has been lost.

How TCP responds to the loss of any single packet depends what it has just been doing. But there is not really a precedent for TCP's response when it experiences a CE mark having sent only one (small) packet. If TCP had been adding one segment per RTT, it would have halved its congestion window, but it hasn't established a congestion window yet. If it had been exponentially increasing it would have exited slow start, but it hasn't started exponentially increasing yet so it hasn't established a slow-start threshold.

Therefore, we have to work out a reasoned argument for what to do. If an AQM is CE-marking packets, it implies there is already a queue and it is probably already somewhere around the AQM's operating point - it is unlikely to be well below and it might be well above. So, it does not seem sensible to add a number of packets at once. On the other hand, it is highly unlikely that the SYN-ACK itself pushed the AQM into congestion, so it will be safe to introduce another single segment immediately (1 RTT after the SYN-ACK). Therefore, starting to probe for capacity with a slow start from an initial window of 1 segment seems appropriate to the circumstances. This is the approach adopted in Section 3.2.2.

4.4. Pure ACKs

Section 5.2 of RFC 3168 gives the following arguments for not allowing the ECT marking of pure ACKs (ACKs not piggy-backed on data):

> "To ensure the reliable delivery of the congestion indication of the CE codepoint, an ECT codepoint MUST NOT be set in a packet unless the loss of that packet in the network would be detected by the end nodes and interpreted as an indication of congestion.
>
> Transport protocols such as TCP do not necessarily detect all packet drops, such as the drop of a "pure" ACK packet; for example, TCP does not reduce the arrival rate of subsequent ACK packets in response to an earlier dropped ACK packet. Any proposal for extending ECN-Capability to such packets would have to address issues such as the case of an ACK packet that was marked with the CE codepoint but was later dropped in the network. We believe that this aspect is still the subject of research, so this document specifies that at this time, "pure" ACK packets MUST NOT indicate ECN-Capability."

Later on, in section 6.1.4 it reads:

> "For the current generation of TCP congestion control algorithms, pure acknowledgement packets (e.g., packets that do not contain any accompanying data) MUST be sent with the not-ECT codepoint. Current TCP receivers have no mechanisms for reducing traffic on the ACK-path in response to congestion notification. Mechanisms for responding to congestion on the ACK-path are areas for current and future research. (One simple possibility would be for the sender to reduce its congestion window when it receives a pure ACK packet with the CE codepoint set). For current TCP implementations, a single dropped ACK generally has only a very small effect on the TCP’s sending rate."

We next address each of the arguments presented above.

The first argument is a specific instance of the reliability argument for the case of pure ACKs. This has already been addressed by countering the general reliability argument in Section 4.1.

The second argument says that ECN ought not to be enabled unless there is a mechanism to respond to it. However, actually there _is_ a mechanism to respond to congestion on a pure ACK that RFC 3168 has overlooked - the congestion window mechanism. When data segments and pure ACKs are interspersed, congestion notifications ought to regulate the congestion window, whether they are on data segments or

on pure ACKs. Otherwise, if ECN is disabled on Pure ACKs, and if (say) 70% of the segments in one direction are Pure ACKs, about 70% of the congestion notifications will be missed and the data segments will not be correctly regulated.

So RFC 3168 ought to have considered two congestion response mechanisms - reducing the congestion window (cwnd) and reducing the ACK rate - and only the latter was missing. Further, RFC 3168 was incorrect to assume that, if one ACK was a pure ACK, all segments in the same direction would be pure ACKs. Admittedly a continual stream of pure ACKs in one direction is quite a common case (e.g. a file download). However, it is also common for the pure ACKs to be interspersed with data segments (e.g. HTTP/2 browser requests controlling a web application). Indeed, it is more likely that any congestion experienced by pure ACKs will be due to mixing with data segments, either within the same flow, or within competing flows.

This insight swings the argument towards enabling ECN on pure ACKs so that CE marks can drive the cwnd response to congestion (whenever data segments are interspersed with the pure ACKs). Then to separately decide whether an ACK rate response is also required (when they are ECN-enabled). The two types of response are addressed separately in the following two subsections, then a final subsection draws conclusions.

4.4.1. Cwnd Response to CE-Marked Pure ACKs

If the sender of pure ACKs sets them to ECT, the bullets below assess whether the three stages of the congestion response mechanism will all work for each type of congestion feedback (classic ECN [RFC3168] and AccECN [I-D.ietf-tcpm-accurate-ecn]):

Detection: The receiver of a pure ACK can detect a CE marking on it:

* Classic feedback: the receiver will not expect CE marks on pure ACKs, so it will be implementation-dependent whether it happens to check for CE marks on all packets.

* AccECN feedback: the AccECN specification requires the receiver of any TCP packets to count any CE marks on them (whether or not control packets are ECN-capable).

Feedback: TCP never ACKs a pure ACK, but the receiver of a CE-mark on a pure ACK can feed it back when it sends a subsequent data segment (if it ever does):

* Classic feedback: the receiver (of the pure ACKs) would set the echo congestion experienced (ECE) flag in the TCP header as normal.

* AccECN feedback: the receiver continually feeds back a count of the number of CE-marked packets that it has received (and, if possible, a count of CE-marked bytes).

Congestion response: In either case (classic or AccECN feedback), if the TCP sender does receive feedback about CE-markings on pure ACKs, it will react in the usual way by reducing its congestion window accordingly. This will regulate the rate of any data packets it is sending amongst the pure ACKs.

4.4.2. ACK Rate Response to CE-Marked Pure ACKs

Reducing the congestion window will have no effect on the rate of pure ACKs. The worst case here is if the bottleneck is congested solely with pure ACKs, but it could also be problematic if a large fraction of the load was from unresponsive ACKs, leaving little or no capacity for the load from responsive data.

Since RFC 3168 was published, Acknowledgement Congestion Control (AckCC) techniques have been documented in [RFC5690] (informational). So any pair of TCP end-points can choose to agree to regulate the delayed ACK ratio in response to lost or CE-marked pure ACKs. However, the protocol has a number of open deployment issues (e.g. it relies on two new TCP options, one of which is required on the SYN where option space is at a premium and, if either option is blocked by a middlebox, no fall-back behaviour is specified). The new TCP options addressed two problems, namely that TCP had: i) no mechanism to allow ECT to be set on pure ACKs; and ii) no mechanism to feed back loss or CE-marking of pure ACKs. A combination of the present specification and AccECN addresses both these problems, at least for ECN marking. So it might now be possible to design an ECN-specific ACK congestion control scheme without the extra TCP options proposed in RFC 5690. However, such a mechanism is out of scope of the present document.

Setting aside the practicality of RFC 5690, the need for AckCC has not been conclusively demonstrated. It has been argued that the Internet has survived so far with no mechanism to even detect loss of pure ACKs. However, it has also been argued that ECN is not the same as loss. Packet discard can naturally thin the ACK load to whatever the bottleneck can support, whereas ECN marking does not (it queues the ACKs instead). Nonetheless, RFC 3168 (section 7) recommends that an AQM switches over from ECN marking to discard when the marking

probability becomes high. Therefore discard can still be relied on to thin out ECN-enabled pure ACKs as a last resort.

4.4.3. Summary: Enabling ECN on Pure ACKs

In the case when AccECN has been negotiated, the arguments for ECT (and CE) on pure ACKs heavily outweigh those against. ECN is always more and never less reliable for delivery of congestion notification. The cwnd response has been overlooked as a mechanism for responding to congestion on pure ACKs, so it is incorrect not to set ECT on pure ACKs when they are interspersed with data segments. And when they are not, packet discard still acts as the "congestion response of last resort". In contrast, not setting ECT on pure ACKs is certainly detrimental to performance, because when a pure ACK is lost it can prevent the release of new data. Separately, AckCC (or perhaps an improved variant exploiting AccECN) could optionally be used to regulate the spacing between pure ACKs. However, it is not clear whether AckCC is justified.

In the case when Classic ECN has been negotiated, there is still an argument for ECT (and CE) on pure ACKs, but it is less clear-cut. Some existing RFC 3168 implementations might happen to (unintentionally) provide the correct feedback to support a cwnd response. Even for those that did not, setting ECT on pure ACKs would still be better for performance than not setting it and do no extra harm. If AckCC was required, it is designed to work with RFC 3168 ECN.

4.5. Window Probes

Section 6.1.6 of RFC 3168 presents only the reliability argument for prohibiting ECT on Window probes:

> "If a window probe packet is dropped in the network, this loss is not detected by the receiver. Therefore, the TCP data sender MUST NOT set either an ECT codepoint or the CWR bit on window probe packets.
>
> However, because window probes use exact sequence numbers, they cannot be easily spoofed in denial-of-service attacks. Therefore, if a window probe arrives with the CE codepoint set, then the receiver SHOULD respond to the ECN indications."

The reliability argument has already been addressed in Section 4.1.

Allowing ECT on window probes could considerably improve performance because, once the receive window has reopened, if a window probe is lost the sender will stall until the next window probe reaches the

receiver, which might be after the maximum retransmission timeout (at least 1 minute [RFC6928]).

On the bright side, RFC 3168 at least specifies the receiver behaviour if a CE-marked window probe arrives, so changing the behaviour ought to be less painful than for other packet types.

4.6. FINs

RFC 3168 is silent on whether a TCP sender can set ECT on a FIN. A FIN is considered as part of the sequence of data, and the rate of pure ACKs sent after a FIN could be controlled by a CE marking on the FIN. Therefore there is no reason not to set ECT on a FIN.

4.7. RSTs

RFC 3168 is silent on whether a TCP sender can set ECT on a RST. The host generating the RST message does not have an open connection after sending it (either because there was no such connection when the packet that triggered the RST message was received or because the packet that triggered the RST message also triggered the closure of the connection).

Moreover, the receiver of a CE-marked RST message can either: i) accept the RST message and close the connection; ii) emit a so-called challenge ACK in response (with suitable throttling) [RFC5961] and otherwise ignore the RST (e.g. because the sequence number is in-window but not the precise number expected next); or iii) discard the RST message (e.g. because the sequence number is out-of-window). In the first two cases there is no point in echoing any CE mark received because the sender closed its connection when it sent the RST. In the third case it makes sense to discard the CE signal as well as the RST.

Although a congestion response following a CE-marking on a RST does not appear to make sense, the following factors have been considered before deciding whether the sender ought to set ECT on a RST message:

- As explained above, a congestion response by the sender of a CE-marked RST message is not possible;
- So the only reason for the sender setting ECT on a RST would be to improve the reliability of the message’s delivery;
- RST messages are used to both mount and mitigate attacks:

    * Spoofed RST messages are used by attackers to terminate ongoing connections, although the mitigations in RFC 5961 have considerably raised the bar against off-path RST attacks;

    * Legitimate RST messages allow endpoints to inform their peers to eliminate existing state that correspond to non existing connections, liberating resources e.g. in DoS attacks scenarios;

- AQMs are advised to disable ECN marking during persistent overload, so:

    * it is harder for an attacker to exploit ECN to intensify an attack;

    * it is harder for a legitimate user to exploit ECN to more reliably mitigate an attack

- Prohibiting ECT on a RST would deny the benefit of ECN to legitimate RST messages, but not to attackers who can disregard RFCs;

- If ECT were prohibited on RSTs

    * it would be easy for security middleboxes to discard all ECN-capable RSTs;

    * However, unlike a SYN flood, it is already easy for a security middlebox (or host) to distinguish a RST flood from legitimate traffic [RFC5961], and even if a some legitimate RSTs are accidentally removed as well, legitimate connections still function.

So, on balance, it has been decided that it is worth experimenting with ECT on RSTs. During experiments, if the ECN capability on RSTs is found to open a vulnerability that is hard to close, this decision can be reversed, before it is specified for the standards track.

4.8. Retransmitted Packets.

RFC 3168 says the sender "MUST NOT" set ECT on retransmitted packets. The rationale for this consumes nearly 2 pages of RFC 3168, so the reader is referred to section 6.1.5 of RFC 3168, rather than quoting it all here. There are essentially three arguments, namely: reliability; DoS attacks; and over-reaction to congestion. We address them in order below.

The reliability argument has already been addressed in Section 4.1.

Protection against DoS attacks is not afforded by prohibiting ECT on retransmitted packets. An attacker can set CE on spoofed retransmissions whether or not it is prohibited by an RFC. Protection against the DoS attack described in section 6.1.5 of RFC 3168 is solely afforded by the requirement that "the TCP data receiver SHOULD ignore the CE codepoint on out-of-window packets". Therefore in Section 3.2.7 the sender is allowed to set ECT on retransmitted packets, in order to reduce the chance of them being dropped. We also strengthen the receiver's requirement from "SHOULD ignore" to "MUST ignore". And we generalize the receiver's requirement to include failure of any validity check, not just out-of-window checks, in order to include the more stringent validity checks in RFC 5961 that have been developed since RFC 3168.

A consequence is that, for those retransmitted packets that arrive at the receiver after the original packet has been properly received (so-called spurious retransmissions), any CE marking will be ignored. There is no problem with that because the fact that the original packet has been delivered implies that the sender's original congestion response (when it deemed the packet lost and retransmitted it) was unnecessary.

Finally, the third argument is about over-reacting to congestion. The argument goes that, if a retransmitted packet is dropped, the sender will not detect it, so it will not react again to congestion (it would have reduced its congestion window already when it retransmitted the packet). Whereas, if retransmitted packets can be CE tagged instead of dropped, senders could potentially react more than once to congestion. However, we argue that it is legitimate to respond again to congestion if it still persists in subsequent round trip(s).

Therefore, in all three cases, it is not incorrect to set ECT on retransmissions.

5. Interaction with popular variants or derivatives of TCP

The following subsections discuss any interactions between setting ECT on all all packets and using the following popular variants or derivatives of TCP: SCTP, IW10 and TFO. This section is informative not normative, because no interactions have been identified that require any change to specifications. The subsection on IW10 discusses potential changes to specifications but recommends that no changes are needed.

TCP variants that have been assessed and found not to interact adversely with ECT on TCP control packets are: SYN cookies (see

Appendix A of [RFC4987] and section 3.1 of [RFC5562]), TCP Fast Open (TFO [RFC7413]) and L4S [I-D.briscoe-tsvwg-l4s-arch].

5.1. SCTP

Stream Control Transmission Protocol (SCTP [RFC4960]) is a standards track protocol derived from TCP. SCTP currently does not include ECN support, but Appendix A of RFC 4960 broadly describes how it would be supported and a draft on the addition of ECN to SCTP has been produced [I-D.stewart-tsvwg-sctpecn]. This draft avoids setting ECT on control packets and retransmissions, closely following the arguments in RFC 3168. When ECN is finally added to SCTP, experience from experiments on adding ECN support to all TCP packets ought to be directly transferable to SCTP.

5.2. IW10

IW10 is an experiment to determine whether it is safe for TCP to use an initial window of 10 SMSS [RFC6928].

This subsection does not recommend any additions to the present specification in order to interwork with IW10. The specifications as they stand are safe, and there is only a corner-case with ECT on the SYN where performance could be occasionally improved, as explained below.

As specified in Section 3.2.1.1, a TCP initiator can only set ECT on the SYN if it requests AccECN support. If, however, the SYN-ACK tells the initiator that the responder does not support AccECN, Section 3.2.1.1 advises the initiator to conservatively reduce its initial window to 1 SMSS because, if the SYN was CE-marked, the SYN-ACK has no way to feed that back.

If the initiator implements IW10, it seems rather over-conservative to reduce IW from 10 to 1 just in case a congestion marking was missed. Nonetheless, the reduction to 1 SMSS will rarely harm performance, because:

- as long as the initiator is caching failures to negotiate AccECN, subsequent attempts to access the same server will not use ECT on the SYN anyway, so there will no longer be any need to conservatively reduce IW;

- currently it is not common for a TCP initiator (client) to have more than one data segment to send {ToDo: evidence/reference?} - IW10 is primarily exploited by TCP servers.

If a responder receives feedback that the SYN-ACK was CE-marked, Section 3.2.2.2 mandates that it reduces its initial window to 1 SMSS. When the responder also implements IW10, it is particularly important to adhere to this requirement in order to avoid overflowing a queue that is clearly already congested.

5.3. TFO

TCP Fast Open (TFO [RFC7413]) is an experiment to remove the round trip delay of TCP's 3-way hand-shake (3WHS). A TFO initiator caches a cookie from a previous connection with a TFO-enabled server. Then, for subsequent connections to the same server, any data included on the SYN can be passed directly to the server application, which can then return up to an initial window of response data on the SYN-ACK and on data segments straight after it, without waiting for the ACK that completes the 3WHS.

The TFO experiment and the present experiment to add ECN-support for TCP control packets can be combined without altering either specification, which is justified as follows:

- The handling of ECN marking on a SYN is no different whether or not it carries data.

- In response to any CE-marking on the SYN-ACK, the responder adopts the normal response to congestion, as discussed in Section 7.2 of [RFC7413].

6. Security Considerations

Section 3.2.6 considers the question of whether ECT on RSTs will allow RST attacks to be intensified. There are several security arguments presented in RFC 3168 for preventing the ECN marking of TCP control packets and retransmitted segments. We believe all of them have been properly addressed in Section 4, particularly Section 4.2.3 and Section 4.8 on DoS attacks using spoofed ECT-marked SYNs and spoofed CE-marked retransmissions.

7. IANA Considerations

There are no IANA considerations in this memo.

8. Acknowledgments

Thanks to Mirja Kuehlewind and David Black for their useful reviews.

The work of Marcelo Bagnulo has been performed in the framework of the H2020-ICT-2014-2 project 5G NORMA. His contribution reflects the

consortiums view, but the consortium is not liable for any use that may be made of any of the information contained therein.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, DOI 10.17487/RFC5562, June 2009, <http://www.rfc-editor.org/info/rfc5562>.

[RFC5961] Ramaiah, A., Stewart, R., and M. Dalal, "Improving TCP's Robustness to Blind In-Window Attacks", RFC 5961, DOI 10.17487/RFC5961, August 2010, <http://www.rfc-editor.org/info/rfc5961>.

[I-D.ietf-tcpm-accurate-ecn] Briscoe, B., Kuehlewind, M., and R. Scheffenegger, "More Accurate ECN Feedback in TCP", draft-ietf-tcpm-accurate-ecn-02 (work in progress), October 2016.

[I-D.ietf-tsvwg-ecn-experimentation] Black, D., "Explicit Congestion Notification (ECN) Experimentation", draft-ietf-tsvwg-ecn-experimentation-02 (work in progress), April 2017.

9.2. Informative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, DOI 10.17487/RFC3540, June 2003, <http://www.rfc-editor.org/info/rfc3540>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <http://www.rfc-editor.org/info/rfc4960>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <http://www.rfc-editor.org/info/rfc4987>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <http://www.rfc-editor.org/info/rfc5681>.

[RFC5690] Floyd, S., Arcia, A., Ros, D., and J. Iyengar, "Adding Acknowledgement Congestion Control to TCP", RFC 5690, DOI 10.17487/RFC5690, February 2010, <http://www.rfc-editor.org/info/rfc5690>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <http://www.rfc-editor.org/info/rfc6298>.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., and M. Mathis, "Increasing TCP's Initial Window", RFC 6928, DOI 10.17487/RFC6928, April 2013, <http://www.rfc-editor.org/info/rfc6928>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <http://www.rfc-editor.org/info/rfc7413>.

[RFC7567] Baker, F., Ed. and G. Fairhurst, Ed., "IETF Recommendations Regarding Active Queue Management", BCP 197, RFC 7567, DOI 10.17487/RFC7567, July 2015, <http://www.rfc-editor.org/info/rfc7567>.

[I-D.briscoe-tsvwg-ecn-l4s-id] Schepper, K., Briscoe, B., and I. Tsang, "Identifying Modified Explicit Congestion Notification (ECN) Semantics for Ultra-Low Queuing Delay", draft-briscoe-tsvwg-ecn-l4s-id-02 (work in progress), October 2016.

[I-D.briscoe-tsvwg-l4s-arch]
Briscoe, B., Schepper, K., and M. Bagnulo, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture", draft-briscoe-tsvwg-l4s-arch-02 (work in progress), March 2017.

[I-D.stewart-tsvwg-sctpecn]
Stewart, R., Tuexen, M., and X. Dong, "ECN for Stream Control Transmission Protocol (SCTP)", draft-stewart-tsvwg-sctpecn-05 (work in progress), January 2014.

[judd-nsdi]
Judd, G., "Attaining the promise and avoiding the pitfalls of TCP in the Datacenter", USENIX Symposium on Networked Systems Design and Implementation (NSDI'15) pp.145-157, May 2015.

[ecn-pam] Trammell, B., Kuehlewind, M., Boppart, D., Learmonth, I., Fairhurst, G., and R. Scheffenegger, "Enabling Internet-Wide Deployment of Explicit Congestion Notification", Int'l Conf. on Passive and Active Network Measurement (PAM'15) pp193-205, 2015.

[ECN-PLUS]
Kuzmanovic, A., "The Power of Explicit Congestion Notification", ACM SIGCOMM 35(4):61--72, 2005.

Authors' Addresses

Marcelo Bagnulo
Universidad Carlos III de Madrid
Av. Universidad 30
Leganes, Madrid 28911
SPAIN

Phone: 34 91 6249500
Email: marcelo@it.uc3m.es
URI: http://www.it.uc3m.es

Bob Briscoe
Simula Research Lab

Email: ietf@bobbriscoe.net
URI: http://bobbriscoe.net/

TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: May 4, 2017

Y. Cheng
N. Cardwell
N. Dukkipati
Google, Inc

October 31, 2016

RACK: a time-based fast loss detection algorithm for TCP
draft-ietf-tcpm-rack-01

Abstract

This document presents a new TCP loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time, instead of packet or sequence counts, to detect losses, for modern TCP implementations that can support per-packet timestamps and the selective acknowledgment (SACK) option. It is intended to replace the conventional DUPACK threshold approach and its variants, as well as other nonstandard approaches.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 4, 2017.

Copyright Notice

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

1. Introduction

This document presents a new loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time instead of the conventional packet or sequence counting approaches for detecting losses. RACK deems a packet lost if some packet sent sufficiently later has been delivered. It does this by recording packet transmission times and inferring losses using cumulative acknowledgments or selective acknowledgment (SACK) TCP options.

In the last couple of years we have been observing several increasingly common loss and reordering patterns in the Internet:

1. Lost retransmissions. Traffic policers [POLICER16] and burst losses often cause retransmissions to be lost again, severely increasing TCP latency.

2. Tail drops. Structured request-response traffic turns more losses into tail drops. In such cases, TCP is application-limited, so it cannot send new data to probe losses and has to rely on retransmission timeouts (RTOs).

3. Reordering. Link layer protocols (e.g., 802.11 block ACK) or routers' internal load-balancing can deliver TCP packets out of order. The degree of such reordering is usually within the order of the path round trip time.

Despite TCP stacks (e.g. Linux) that implement many of the standard and proposed loss detection algorithms [RFC3517][RFC4653][RFC5827][RFC5681][RFC6675][RFC7765][FACK][THIN-STREAM][TLP], we've found that together they do not perform well. The main reason is that many of them are based on the classic rule of counting duplicate acknowledgments [RFC5681]. They can either detect loss quickly or accurately, but not both, especially when the sender is application-limited or under reordering that is unpredictable. And under these conditions none of them can detect lost retransmissions well.

Also, these algorithms, including RFCs, rarely address the interactions with other algorithms. For example, FACK may consider a packet is lost while RFC3517 may not. Implementing N algorithms while dealing with N^2 interactions is a daunting task and error-prone.

The goal of RACK is to solve all the problems above by replacing many of the loss detection algorithms above with one simpler, and also more effective, algorithm.

2. Overview

The main idea behind RACK is that if a packet has been delivered out of order, then the packets sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681][RFC3517][FACK]. But the key innovation in RACK is to use a per-packet transmission timestamp and widely deployed SACK options to conduct time-based inferences instead of inferring losses with packet or sequence counting approaches.

Using a threshold for counting duplicate acknowledgments (i.e., dupthresh) is no longer reliable because of today's prevalent reordering patterns. A common type of reordering is that the last "runt" packet of a window's worth of packet bursts gets delivered first, then the rest arrive shortly after in order. To handle this effectively, a sender would need to constantly adjust the dupthresh to the burst size; but this would risk increasing the frequency of RTOs on real losses.

Today's prevalent lost retransmissions also cause problems with packet-counting approaches [RFC5681][RFC3517][FACK], since those approaches depend on reasoning in sequence number space. Retransmissions break the direct correspondence between ordering in sequence space and ordering in time. So when retransmissions are lost, sequence-based approaches are often unable to infer and quickly repair losses that can be deduced with time-based approaches.

Instead of counting packets, RACK uses the most recently delivered packet's transmission time to judge if some packets sent previous to that time have "expired" by passing a certain reordering settling window. On each ACK, RACK marks any already-expired packets lost, and for any packets that have not yet expired it waits until the reordering window passes and then marks those lost as well. In either case, RACK can repair the loss without waiting for a (long) RTO. RACK can be applied to both fast recovery and timeout recovery, and can detect losses on both originally transmitted and retransmitted packets, making it a great all-weather recovery mechanism.

3. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment

[RFC2018] RFCs. Familiarity with the conservative SACK-based recovery for TCP [RFC6675] is not expected but helps.

RACK has three requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018].

2. For each packet sent, the sender MUST store its most recent transmission time with (at least) millisecond granularity. For round-trip times lower than a millisecond (e.g., intra-datacenter communications) microsecond granularity would significantly help the detection latency but is not required.

3. For each packet sent, the sender MUST remember whether the packet has been retransmitted or not.

We assume that requirement 1 implies the sender keeps a SACK scoreboard, which is a data structure to store selective acknowledgment information on a per-connection basis. For the ease of explaining the algorithm, we use a pseudo-scoreboard that manages the data in sequence number ranges. But the specifics of the data structure are left to the implementor.

RACK does not need any change on the receiver.

4. Definitions of variables

A sender needs to store these new RACK variables:

"Packet.xmit_ts" is the time of the last transmission of a data packet, including retransmissions, if any. The sender needs to record the transmission time for each packet sent and not yet acknowledged. The time MUST be stored at millisecond granularity or finer.

"RACK.packet". Among all the packets that have been either selectively or cummulatively acknowledged, RACK.packet is the one that was sent most recently (including retransmission).

"RACK.xmit_ts" is the latest transmission timestamp of RACK.packet.

"RACK.end_seq" is the ending TCP sequence number of RACk.packet.

"RACK.RTT" is the associated RTT measured when RACK.xmit_ts, above, was changed. It is the RTT of the most recently transmitted packet that has been delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.reo_wnd" is a reordering window for the connection, computed in the unit of time used for recording packet transmission times. It is used to defer the moment at which RACK marks a packet lost.

"RACK.min_RTT" is the estimated minimum round-trip time (RTT) of the connection.

"RACK.ack_ts" is the time when all the sequences in RACK.packet were selectively or cumulatively acknowledged.

Note that the Packet.xmit_ts variable is per packet in flight. The RACK.xmit_ts, RACK.RTT, RACK.reo_wnd, and RACK.min_RTT variables are to keep in TCP control block per connection. RACK.packet and RACK.ack_ts are used as local variables in the algorithm.

5. Algorithm Details

5.1. Transmitting a data packet

Upon transmitting a new packet or retransmitting an old packet, record the time in Packet.xmit_ts. RACK does not care if the retransmission is triggered by an ACK, new application data, an RTO, or any other means.

5.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained in [RFC6298] or [RFC7323] to update the estimated minimum RTT in RACK.min_RTT. The sender can track a simple global minimum of all RTT measurements from the connection, or a windowed min-filtered value of recent RTT measurements. This document does not specify an exact approach.

Step 2: Update RACK.reo_wnd.

To handle the prevalent small degree of reordering, RACK.reo_wnd serves as an allowance for settling time before marking a packet lost. By default it is 1 millisecond. We RECOMMEND implementing the reordering detection in [REORDER-DETECT][RFC4737] to dynamically adjust the reordering window. When the sender detects packet reordering RACK.reo_wnd MAY be changed to RACK.min_RTT/4. We discuss more about the reordering window in the next section.

Step 3: Advance RACK.xmit_ts and update RACK.RTT and RACK.end_seq

Given the information provided in an ACK, each packet cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Among all

the packets newly ACKed or SACKed in the connection, record the most recent Packet.xmit_ts in RACK.xmit_ts if it is ahead of RACK.xmit_ts. Ignore the packet if any of its TCP sequences has been retransmitted before and either of two condition is true:

1. The Timestamp Echo Reply field (TSecr) of the ACK's timestamp option [RFC7323], if available, indicates the ACK was not acknowledging the last retransmission of the packet.

2. The packet was last retransmitted less than RACK.min_rtt ago. While it is still possible the packet is spuriously retransmitted because of a recent RTT decrease, we believe that our experience suggests this is a reasonable heuristic.

If this ACK causes a change to RACK.xmit_ts then record the RTT and sequence implied by this ACK:

```
RACK.RTT = Now() - RACK.xmit_ts
RACK.end_seq = Packet.end_seq
```

Exit here and omit the following steps if RACK.xmit_ts has not changed.

Step 4: Detect losses.

For each packet that has not been fully SACKed, if RACK.xmit_ts is after Packet.xmit_ts + RACK.reo_wnd, then mark the packet (or its corresponding sequence range) lost in the scoreboard. The rationale is that if another packet that was sent later has been delivered, and the reordering window or "reordering settling time" has already passed, the packet was likely lost.

If a packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the given packet lost. Using the basic algorithm above, the sender would wait for the next ACK to further advance RACK.xmit_ts; but this risks a timeout (RTO) if no more ACKs come back (e.g, due to losses or application limit). For timely loss detection, the sender MAY install a "reordering settling" timer set to fire at the earliest moment at which it is safe to conclude that some packet is lost. The earliest moment is the time it takes to expire the reordering window of the earliest unacked packet in flight.

This timer expiration value can be derived as follows. As a starting point, we consider that the reordering window has passed if the RACK.packet was sent sufficiently after the packet in question, or a sufficient time has elapsed since the RACK.packet was S/ACKed, or some combination of the two. More precisely, RACK marks a packet as

lost if the reordering window for a packet has elapsed through the sum of:

1. delta in transmit time between a packet and the RACK.packet

2. delta in time between when RACK.ack_ts and now

So we mark a packet as lost if:

```
RACK.xmit_ts > Packet.xmit_ts
        AND
(RACK.xmit_ts - Packet.xmit_ts) + (now - RACK.ack_ts) > RACK.reo_wnd
```

If we solve this second condition for "now", the moment at which we can declare a packet lost, then we get:

```
now > Packet.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)
```

Then (RACK.ack_ts - RACK.xmit_ts) is just the RTT of the packet we used to set RACK.xmit_ts, so this reduces to:

```
now > Packet.xmit_ts + RACK.RTT + RACK.reo_wnd
```

The following pseudocode implements the algorithm above. When an ACK is received or the RACK timer expires, call RACK_detect_loss(). The algorithm includes an additional optimization to break timestamp ties by using the TCP sequence space. The optimization is particularly useful to detect losses in a timely manner with TCP Segmentation Offload, where multiple packets in one TSO blob have identical timestamps. It is also useful when the timestamp clock granularity is close to or longer than the actual round trip time.

```
RACK_detect_loss():
min_timeout = 0

For each packet, Packet, in the scoreboard:
    If Packet is already SACKed, ACKed,
       or marked lost and not yet retransmitted:
        Skip to the next packet

    If Packet.xmit_ts > RACK.xmit_ts:
        Skip to the next packet
    /*  Timestamp tie breaker */
    If Packet.xmit_ts == RACK.xmit_ts AND
       Packet.end_seq > RACK.end_seq:
        Skip to the next packet

    timeout = Packet.xmit_ts + RACK.RTT + RACK.reo_wnd + 1
    If Now() >= timeout:
        Mark Packet lost
    Else If (min_timeout == 0) or (timeout is before min_timeout):
        min_timeout = timeout

If min_timeout != 0
    Arm a timer to call RACK_detect_loss() after min_timeout
```

6. Tail Loss Probe: fast recovery on tail losses

This section describes a supplemental algorithm, Tail Loss Probe (TLP), which leverages RACK to further reduce RTO recoveries. TLP triggers fast recovery to quickly repair tail losses that can otherwise only be recoverable by RTOs. After an original data transmission, TLP sends a probe data segment within one to two RTTs. The probe data segment can either be new, previously unsent data, or a retransmission. In either case the goal is to elicit more feedback from the receiver, in the form of an ACK (potentially with SACK blocks), to allow RACK to trigger fast recovery instead of an RTO.

An RTO occurs when the first unacknowledged sequence number is not acknowledged after a conservative period of time has elapsed [RFC6298 [1]]. Common causes of RTOs include:

1. Tail losses at the end of an application transaction.

2. Lost retransmits, which can halt fast recovery if the ACK stream completely dries up. For example, consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. On receipt of a SACK for P3, RACK marks P1 and P2 as lost and retransmits them as R1 and R2. Suppose R1 and R2 are lost as

well, so there are no more returning ACKs to detect R1 and R2 as lost. Recovery stalls.

3. Tail losses of ACKs.

4. An unexpectedly long round-trip time (RTT). This can cause ACKs to arrive after the RTO timer expires. The F-RTO algorithm [RFC5682 [2]] is designed to detect such spurious retransmission timeouts and at least partially undo the consequences of such events (though F-RTO cannot be used in many situations).

6.1. Tail Loss Probe: An Example

Following is an example of TLP. All events listed are at a TCP sender.

(1) Sender transmits segments 1-10: 1, 2, 3, ..., 8, 9, 10. There is no more new data to transmit. A PTO is scheduled to fire in 2 RTTs, after the transmission of the 10th segment. (2) Sender receives acknowledgements (ACKs) for segments 1-5; segments 6-10 are lost and no ACKs are received. The sender reschedules its PTO timer relative to the last received ACK, which is the ACK for segment 5 in this case. The sender sets the PTO interval using the calculation described in step (2) of the algorithm. (3) When PTO fires, sender retransmits segment 10. (4) After an RTT, a SACK for packet 10 arrives. The ACK also carries SACK holes for segments 6, 7, 8 and 9. This triggers RACK-based loss recovery. (5) The connection enters fast recovery and retransmits the remaining lost segments.

6.2. Tail Loss Probe Algorithm Details

We define the terminology used in specifying the TLP algorithm:

FlightSize: amount of outstanding data in the network, as defined in [RFC5681 [3]].

RTO: The transport’s retransmission timeout (RTO) is based on measured round-trip times (RTT) between the sender and receiver, as specified in [RFC6298 [4]] for TCP. PTO: Probe timeout is a timer event indicating that an ACK is overdue. Its value is constrained to be smaller than or equal to an RTO.

SRTT: smoothed round-trip time, computed as specified in [RFC6298 [5]].

Open state: the sender has so far received in-sequence ACKs with no SACK blocks, and no other indications (such as retransmission timeout) that a loss may have occurred.

The TLP algorithm has three phases, which we discuss in turn.

6.2.1. Phase 1: Scheduling a loss probe

Step 1: Check conditions for scheduling a PTO.

A sender should schedule a PTO after transmitting new data or receiving an ACK if the following conditions are met:

(a) The connection is in Open state. (b) The connection is either cwnd-limited (the data in flight matches or exceeds the cwnd) or application-limited (there is no unsent data that the receiver window allows to be sent). (c) SACK is enabled for the connection.

(d) The most recently transmitted data was not itself a TLP probe (i.e. a sender MUST NOT send consecutive or back-to-back TLP probes).

(e) TLPRtxOut is false, indicating there is no TLP retransmission episode in progress (see below).

Step 2: Select the duration of the PTO.

A sender SHOULD use the following logic to select the duration of a PTO:

```
If an SRTT estimate is available:
     PTO = 2 * SRTT
Else:
     PTO = initial RTO of 1 sec
If FlightSize == 1:
     PTO = max(PTO, 1.5 * SRTT + WCDelAckT)
     PTO = max(10ms, PTO)
     PTO = min(RTO, PTO)
```

Aiming for a PTO value of 2*SRTT allows a sender to wait long enough to know that an ACK is overdue. Under normal circumstances, i.e. no losses, an ACK typically arrives in one SRTT. But choosing PTO to be exactly an SRTT is likely to generate spurious probes given that network delay variance and even end-system timings can easily push an ACK to be above an SRTT. We chose PTO to be the next integral multiple of SRTT. Similarly, current end-system processing latencies and timer granularities can easily push an ACK beyond 10ms, so senders SHOULD use a minimum PTO value of 10ms. If RTO is smaller than the computed value for PTO, then a probe is scheduled to be sent at the RTO time.

WCDelAckT stands for worst case delayed ACK timer. When FlightSize is 1, PTO is inflated additionally by WCDelAckT time to compensate

for a potential long delayed ACK timer at the receiver. The RECOMMENDED value for WCDelAckT is 200ms, or the delayed ACK interval value explicitly negotiated by the sender and receiver, if one is available.

6.2.2. Phase 2: Sending a loss probe

When the PTO fires, transmit a probe data segment:

```
    If a previously unsent segment exists AND
       the receive window allows new data to be sent:
        Transmit that new segment
        FlightSize += SMSS
        The cwnd remains unchanged
        Record Packet.xmit_ts
    Else:
        Retransmit the last segment
        The cwnd remains unchanged
```

6.2.3. Phase 3: ACK processing

On each incoming ACK, the sender should ancel any existing loss probe timer. The timer will be re-scheduled if appropriate.

6.3. TLP recovery detection

If the only loss in an outstanding window of data was the last segment, then a TLP loss probe retransmission of that data segment might repair the loss. TLP loss detection examines ACKs to detect when the probe might have repaired a loss, and thus allows congestion control to properly reduce the congestion window (cwnd) [RFC5681 [6]].

Consider a TLP retransmission episode where a sender retransmits a tail packet in a flight. The TLP retransmission episode ends when the sender receives an ACK with a SEG.ACK above the SND.NXT at the time the episode started. During the TLP retransmission episode the sender checks for a duplicate ACK or D-SACK indicating that both the original segment and TLP retransmission arrived at the receiver, meaning there was no loss that needed repairing. If the TLP sender does not receive such an indication before the end of the TLP retransmission episode, then it MUST estimate that either the original data segment or the TLP retransmission were lost, and congestion control MUST react appropriately to that loss as it would any other loss.

Since a significant fraction of the hosts that support SACK do not support duplicate selective acknowledgments (D-SACKs) [RFC2883 [7]]

the TLP algorithm for detecting such lost segments relies only on basicRFC 2018 [8] SACK support [RFC2018 [9]].

Definitions of variables

TLPRtxOut: a boolean indicating whether there is an unacknowledged TLP retransmission.

TLPHighRxt: the value of SND.NXT at the time of sending a TLP retransmission.

6.3.1. Initializing and resetting state

When a connection is created, or suffers a retransmission timeout, or enters fast recovery, it should reset TLPRtxOut to false

6.3.2. Recording loss probe states

Senders must only send a TLP loss probe retransmission if TLPRtxOut is false. This ensures that at any given time a connection has at most one outstanding TLP retransmission. This allows the sender to use the algorithm described in this section to estimate whether any data segments were lost.

Note that this condition only restricts TLP loss probes that are retransmissions. There may be an arbitrary number of outstanding unacknowledged TLP loss probes that consist of new, previously-unsent data, since the retransmission timeout and fast recovery algorithms are sufficient to detect losses of such probe segments.

Upon sending a TLP probe that is a retransmission, the sender set TLPRtxOut to true and TLPHighRxt to SND.NXT

Detecting recoveries done by loss probes

Step 1: Track ACKs indicating receipt of original and retransmitted segments

A sender considers both the original segment and TLP probe retransmission segment as acknowledged if either (i) or (ii) are true:

(i) This is a duplicate acknowledgment (as defined in [RFC5681 [10]], section 2), and all of the following conditions are met:

(a) TLPRtxOut is true

(b) SEG.ACK == TLPHighRxt

(c) SEG.ACK == SND.UNA

(d) the segment contains no SACK blocks for sequence ranges above TLPHighRxt

(e) the segment contains no data

(f) the segment is not a window update

(ii) This is an ACK acknowledging a sequence number at or above TLPHighRxt and it contains a D-SACK; i.e. all of the following conditions are met:

(a) TLPRtxOut is true

(b) SEG.ACK >= TLPHighRxt and

(c) the ACK contains a D-SACK block

If either conditions (i) or (ii) are met, then the sender estimates that the receiver received both the original data segment and the TLP probe retransmission, and so the sender considers the TLP episode to be done, and records that fact by setting TLPRtxOut to false.

Step 2: Mark the end of a TLP retransmission episode and detect losses

If the sender receives a cumulative ACK for data beyond the TLP loss probe retransmission then, in the absence of reordering on the return path of ACKs, it should have received any ACKs for the original segment and TLP probe retransmission segment. At that time, if the TLPRtxOut flag is still true and thus indicates that the TLP probe retransmission remains unacknowledged, then the sender should presume that at least one of its data segments was lost, so it SHOULD invoke a congestion control response equivalent to the response to any other loss.

More precisely, on each ACK, after executing step (5a) the sender SHOULD reset the TLPRtxOut to false, and invoke the congestion control about the loss event that TLP has successfully repaired.

7. RACK and TLP discussions

7.1. Advantages

The biggest advantage of RACK is that every data packet, whether it is an original data transmission or a retransmission, can be used to detect losses of the packets sent prior to it.

Example: tail drop. Consider a sender that transmits a window of three data packets (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. RACK will mark P1 as lost when the SACK of P2 is received, and this will trigger the retransmission of P1 as R1. When R1 is cumulatively acknowledged, RACK will mark P3 as lost and the sender will retransmit P3 as R3. This example illustrates how RACK is able to repair certain drops at the tail of a transaction without any timer. Notice that neither the conventional duplicate ACK threshold [RFC5681], nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses, because of the required packet or sequence count.

Example: lost retransmit. Consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. When P3 is SACKed, RACK will mark P1 and P2 lost and they will be retransmitted as R1 and R2. Suppose R1 is lost again (as a tail drop) but R2 is SACKed; RACK will mark R1 lost for retransmission again. Again, neither the conventional three duplicate ACK threshold approach, nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses. And such a lost retransmission is very common when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit. Retransmissions are often lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.

Example: (small) degree of reordering. Consider a common reordering event: a window of packets are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload due to application-limited behavior. Suppose the sender has detected reordering previously (e.g., by implementing the algorithm in [REORDER-DETECT]) and thus RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within min_RTT/4, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still falsely mark P1 and P2 lost. We discuss how to reduce the false positives in the end of this section.

The examples above show that RACK is particularly useful when the sender is limited by the application, which is common for interactive, request/response traffic. Similarly, RACK still works when the sender is limited by the receive window, which is common for applications that use the receive window to throttle the sender.

For some implementations (e.g., Linux), RACK works quite efficiently with TCP Segmentation Offload (TSO). RACK always marks the entire TSO blob lost because the packets in the same TSO blob have the same transmission timestamp. By contrast, the counting based algorithms (e.g., [RFC3517][RFC5681]) may mark only a subset of packets in the TSO blob lost, forcing the stack to perform expensive fragmentation of the TSO blob, or to selectively tag individual packets lost in the scoreboard.

7.2. Disadvantages

RACK requires the sender to record the transmission time of each packet sent at a clock granularity of one millisecond or finer. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording packet transmission times will need to add per-packet internal state (commonly either 4 or 8 octets per packet) to track transmission times. In contrast, the conventional approach requires one variable to track number of duplicate ACK threshold.

7.3. Adjusting the reordering window

RACK uses a reordering window of min_rtt / 4. It uses the minimum RTT to accommodate reordering introduced by packets traversing slightly different paths (e.g., router-based parallelism schemes) or out-of-order deliveries in the lower link layer (e.g., wireless links using link-layer retransmission). Alternatively, RACK can use the smoothed RTT used in RTT estimation [RFC6298]. However, smoothed RTT can be significantly inflated by orders of magnitude due to congestion and buffer-bloat, which would result in an overly conservative reordering window and slow loss detection. Furthermore, RACK uses a quarter of minimum RTT because Linux TCP uses the same factor in its implementation to delay Early Retransmit [RFC5827] to reduce spurious loss detections in the presence of reordering, and experience shows that this seems to work reasonably well.

One potential improvement is to further adapt the reordering window by measuring the degree of reordering in time, instead of packet distances. But that requires storing the delivery timestamp of each packet. Some scoreboard implementations currently merge SACKed packets together to support TSO (TCP Segmentation Offload) for faster scoreboard indexing. Supporting per-packet delivery timestamps is difficult in such implementations. However, we acknowledge that the current metric can be improved by further research.

7.4. Relationships with other loss recovery algorithms

The primary motivation of RACK is to ultimately provide a simple and general replacement for some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653] and nonstandard ones [FACK][THIN-STREAM]. While RACK can be a supplemental loss detection on top of these algorithms, this is not necessary, because the RACK implicitly subsumes most of them.

[RFC5827][RFC4653][THIN-STREAM] dynamically adjusts the duplicate ACK threshold based on the current or previous flight sizes. RACK takes a different approach, by using only one ACK event and a reordering window. RACK can be seen as an extended Early Retransmit [RFC5827] without a FlightSize limit but with an additional reordering window. [FACK] considers an original packet to be lost when its sequence range is sufficiently far below the highest SACKed sequence. In some sense RACK can be seen as a generalized form of FACK that operates in time space instead of sequence space, enabling it to better handle reordering, application-limited traffic, and lost retransmissions.

Nevertheless RACK is still an experimental algorithm. Since the oldest loss detection algorithm, the 3 duplicate ACK threshold [RFC5681], has been standardized and widely deployed, we RECOMMEND TCP implementations use both RACK and the algorithm specified in Section 3.2 in [RFC5681] for compatibility.

RACK is compatible with and does not interfere with the the standard RTO [RFC6298], RTO-restart [RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is because RACK only detects loss by using ACK events. It neither changes the timer calculation nor detects spurious timeouts.

Furthermore, RACK naturally works well with Tail Loss Probe [TLP] because a tail loss probe solicit seither an ACK or SACK, which can be used by RACK to detect more losses. RACK can be used to relax TLP’s requirement for using FACK and retransmitting the the highest-sequenced packet, because RACK is agnostic to packet sequence numbers, and uses transmission time instead. Thus TLP can be modified to retransmit the first unacknowledged packet, which can improve application latency.

7.5. Interaction with congestion control

RACK intentionally decouples loss detection from congestion control. RACK only detects losses; it does not modify the congestion control algorithm [RFC5681][RFC6937]. However, RACK may detect losses earlier or later than the conventional duplicate ACK threshold approach does. A packet marked lost by RACK SHOULD NOT be

retransmitted until congestion control deems this appropriate (e.g. using [RFC6937]).

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [RFC5681]. The distinction between fast recovery or RTO recovery is not necessary because RACK is purely based on the transmission time order of packets. When a packet retransmitted by RTO is acknowledged, RACK will mark any unacked packet sent sufficiently prior to the RTO as lost, because at least one RTT has elapsed since these packets were sent.

7.6. TLP recovery detection with delayed ACKs

Delayed ACKs complicate the detection of reparies done by TLP, since with a delayed ACK the sender receives one fewer ACK than would normally be expected. To mitigate this complication, before sending a TLP loss probe retransmission, the sender should attempt to wait long enough that the receiver has sent any delayed ACKs that it is withholding. The sender algorithm described above features such a delay, in the form of WCDelAckT. Furthermore, if the receiver supports duplicate selective acknowledgments (D-SACKs) [RFC2883] then in the case of a delayed ACK the sender's TLP loss detection algorithm (in step (4)(a)(ii), above) can use the D-SACK information to infer that the original and TLP retransmission both arrived at the receiver.

If there is ACK loss or a delayed ACK without a D-SACK, then this algorithm is conservative, because the sender will reduce cwnd when in fact there was no packet loss. In practice this is acceptable, and potentially even desirable: if there is reverse path congestion then reducing cwnd is prudent.

However, in practice sending a single byte of data turned out to be problematic to implement and more fragile than necessary. Instead we use a full segment to probe but have to add complexity to compensate for the probe itself masking losses.

7.7. RACK for other transport protocols

RACK can be implemented in other transport protocols. The algorithm can skip step 3 and simplify if the protocol can support unique transmission or packet identifier (e.g. TCP echo options). For example, the QUIC protocol implements RACK [QUIC-LR] .

8. Experiments and Performance Evaluations

RACK and TLP have been deployed at Google including the connections to the users in the Internet and internally. We conducted an performance evaluation experiment on RACK and TLP on a small set of Google Web servers in western-europe that serve most European and some African countries. The length of the experiments was five days (one weekend plus 3 weekdays) in October 2016, where the servers were divided evenly into three groups.

Group 1 (control): RACK off, TLP off

Group 2: RACK on, TLP off

Group 3: RACK on, TLP on

All groups use Linux using the Cubic congestion control with an initial window of 10 packets and fq/pacing qdisc. In term of specific recovery features, all of them enable RFC3517 (Conservative SACK-based recovery) and RFC5682 (F-RTO) but disable FACK because it is not an IETF RFC. The goal of this setup is to compare RACK and TLP to RFC-based loss recoveries instead of Linux-based recoveries.

The servers sit behind a load-balancer that distributes the connections evenly across the three groups.

Each group handles similar amount of connections and send and receive similar amount of data. We compare total amount of time spent in loss recovery across groups. The recovery time is from when the recovery and retransmit starts, till the remote has acknowledge beyond the highest sequence at the time the recovery starts. Therefore the recovery includes both fast recoveries and timeout recoveries. Our data shows that Group 2 recovery latency is only 2% lower than the Group 1 recovery latency. But Group 3 recovery latency is 25% lower than Group 1 by reducing 40% of the RTOs triggered recoveries! Therefore it is very important to implement both TLP and RACK for performance.

We want to emphasize that the current experiment is limited in terms of network coverage. The connectivities in western-europe is fairly good therefore loss recovery is not a performance bottleneck. We plan to expand our experiments in regions with worse connectivities, in particular on networks with strong traffic policing. We also plan to add the fourth group to disable RFC3517 to use solely RACK and TLP only to see if RACK plus TLP can completely replace all other SACK based recoveries.

9. Security Considerations

RACK does not change the risk profile for TCP.

An interesting scenario is ACK-splitting attacks [SCWA99]: for an MSS-size packet sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This would not fool RACK. RACK.xmit_ts would not advance because all the sequences of the packet are transmitted at the same time (carry the same transmission timestamp). In other words, SACKing only one byte of a packet or SACKing the packet in entirety have the same effect on RACK.

10. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

11. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael Welzl for his per-packet timer idea that inspired this work. Eric Dumazet, Randy Stewart, Van Jacobson, Ian Swett, and Jana Iyengar contributed to the algorithm and the implementations in Linux, FreeBSD and QUIC.

12. References

12.1. Normative References

[RFC793] Postel, J., "Transmission Control Protocol", September 1981.

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

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", May 2013.

[RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", RFC 4737, November 2006.

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, August 2012.

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

[RFC5827] Allman, M., Ayesta, U., Wang, L., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, April 2010.

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, September 2009.

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

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

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

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", September 2014.

12.2. Informative References

[FACK] Mathis, M. and M. Jamshid, "Forward acknowledgement: refining TCP congestion control", ACM SIGCOMM Computer Communication Review, Volume 26, Issue 4, Oct. 1996. , 1996.

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Drops", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), August 2013.

[RFC7765] Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", February 2016.

[REORDER-DETECT]
Zimmermann, A., Schulte, L., Wolff, C., and A. Hannemann, "Detection and Quantification of Packet Reordering with TCP", draft-zimmermann-tcpm-reordering-detection-02 (work in progress), November 2014.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Recovery And Congestion Control", draft-tsvwg-quic-loss-recovery-01 (work in progress), June 2016.

[THIN-STREAM]
Petlund, A., Evensen, K., Griwodz, C., and P. Halvorsen, "TCP enhancements for interactive thin-stream applications", NOSSDAV , 2008.

[SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control With a Misbehaving Receiver", ACM Computer Communication Review, 29(5) , 1999.

[POLICER16]
Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., and R. Govindan, "An Analysis of Traffic Policing in the Web", ACM SIGCOMM , 2016.

12.3. URIs

[1] https://tools.ietf.org/html/rfc6298

[2] https://tools.ietf.org/html/rfc5682

[3] https://tools.ietf.org/html/rfc5681

[4] https://tools.ietf.org/html/rfc6298

[5] https://tools.ietf.org/html/rfc6298

[6] https://tools.ietf.org/html/rfc5681

[7] https://tools.ietf.org/html/rfc2883

[8] https://tools.ietf.org/html/rfc2018

[9] https://tools.ietf.org/html/rfc2018

[10] https://tools.ietf.org/html/rfc5681

Authors' Addresses

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043
USA

Email: ycheng@google.com

Neal Cardwell
Google, Inc
76 Ninth Avenue
New York, NY 10011
USA

Email: ncardwell@google.com

Nandita Dukkipati
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043
USA

Email: nanditad@google.com

TCP Maintenance Working Group
Internet-Draft
Intended status: Experimental
Expires: October 28, 2019

Y. Cheng
N. Cardwell
N. Dukkipati
P. Jha
Google, Inc

April 26, 2019

RACK: a time-based fast loss detection algorithm for TCP
draft-ietf-tcpm-rack-05

Abstract

This document presents a new TCP loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time, instead of packet or sequence counts, to detect losses, for modern TCP implementations that can support per-packet timestamps and the selective acknowledgment (SACK) option. It is intended to replace the conventional DUPACK threshold approach and its variants, as well as other nonstandard approaches.

1. Introduction

This document presents a new loss detection algorithm called RACK ("Recent ACKnowledgment"). RACK uses the notion of time instead of the conventional packet or sequence counting approaches for detecting losses. RACK deems a packet lost if some packet sent sufficiently later has been delivered. It does this by recording packet transmission times and inferring losses using cumulative acknowledgments or selective acknowledgment (SACK) TCP options.

In the last couple of years we have been observing several increasingly common loss and reordering patterns in the Internet:

1. Lost retransmissions. Traffic policers [POLICER16] and burst losses often cause retransmissions to be lost again, severely increasing TCP latency.

2. Tail drops. Structured request-response traffic turns more losses into tail drops. In such cases, TCP is application-limited, so it cannot send new data to probe losses and has to rely on retransmission timeouts (RTOs).

3. Reordering. Link-layer protocols (e.g., 802.11 block ACK), link bonding, or routers' internal load-balancing can deliver TCP packets out of order. The degree of such reordering is usually within the order of the path round trip time.

Despite TCP stacks (e.g. Linux) that implement many of the standard and proposed loss detection algorithms [RFC4653][RFC5827][RFC5681][RFC6675][RFC7765][FACK][THIN-STREAM], we've found that together they do not perform well. The main reason is that many of them are based on the classic rule of counting duplicate acknowledgments [RFC5681]. They can either detect loss quickly or accurately, but not both, especially when the sender is application-limited or under reordering that is unpredictable. And under these conditions none of them can detect lost retransmissions well.

Also, these algorithms, including RFCs, rarely address the interactions with other algorithms. For example, FACK may consider a packet is lost while RFC6675 may not. Implementing N algorithms while dealing with N^2 interactions is a daunting task and error-prone.

The goal of RACK is to solve all the problems above by replacing many of the loss detection algorithms above with one more effective algorithm to handle loss and reordering.

2. Overview

The main idea behind RACK is that if a packet has been delivered out of order, then the packets sent chronologically before that were either lost or reordered. This concept is not fundamentally different from [RFC5681][RFC6675][FACK]. But the key innovation in RACK is to use a per-packet transmission timestamp and widely deployed SACK options to conduct time-based inferences instead of inferring losses with packet or sequence counting approaches.

Using a threshold for counting duplicate acknowledgments (i.e., DupThresh) alone is no longer reliable because of today's prevalent reordering patterns. A common type of reordering is that the last "runt" packet of a window's worth of packet bursts gets delivered first, then the rest arrive shortly after in order. To handle this effectively, a sender would need to constantly adjust the DupThresh to the burst size; but this would risk increasing the frequency of RTOs on real losses.

Today's prevalent lost retransmissions also cause problems with packet-counting approaches [RFC5681][RFC6675][FACK], since those approaches depend on reasoning in sequence number space. Retransmissions break the direct correspondence between ordering in sequence space and ordering in time. So when retransmissions are lost, sequence-based approaches are often unable to infer and quickly repair losses that can be deduced with time-based approaches.

Instead of counting packets, RACK uses the most recently delivered packet's transmission time to judge if some packets sent previous to that time have "expired" by passing a certain reordering settling window. On each ACK, RACK marks any already-expired packets lost, and for any packets that have not yet expired it waits until the reordering window passes and then marks those lost as well. In either case, RACK can repair the loss without waiting for a (long) RTO. RACK can be applied to both fast recovery and timeout recovery, and can detect losses on both originally transmitted and retransmitted packets, making it a great all-weather loss detection mechanism.

3. Design Rationale for Reordering Tolerance

The reordering behavior of networks can evolve (over years) in response to the behavior of transport protocols and applications, as well as the needs of network designers and operators. From a network

or link designer's viewpoint, parallelization (eg. link bonding) is the easiest way to get a network to go faster. Therefore their main constraint on speed is reordering, and there is pressure to relax that constraint. If RACK becomes widely deployed, the underlying networks may introduce more reordering for higher throughput. But this may result in excessive reordering that hurts end to end performance:

1. End host packet processing: extreme reordering on high-speed networks would incur high CPU cost by greatly reducing the effectiveness of aggregation mechanisms, such as large receive offload (LRO) and generic receive offload (GRO), and significantly increasing the number of ACKs.

2. Congestion control: TCP congestion control implicitly assumes the feedback from ACKs are from the same bottleneck. Therefore it cannot handle well scenarios where packets are traversing largely disjoint paths.

3. Loss recovery: Having an excessively large reordering window to accommodate widely different latencies from different paths would increase the latency of loss recovery.

An end-to-end transport protocol cannot tell immediately whether a hole is reordering or loss. It can only distinguish between the two in hindsight if the hole in the sequence space gets filled later without a retransmission. How long the sender waits for such potential reordering events to settle is determined by the current reordering window.

Given these considerations, a core design philosophy of RACK is to adapt to the measured duration of reordering events, within reasonable and specific bounds. To accomplish this RACK places the following mandates on the reordering window:

1. The initial RACK reordering window SHOULD be set to a small fraction of the round-trip time.

2. If no reordering has been observed, then RACK SHOULD honor the classic 3-DUPACK rule for initiating fast recovery. One simple way to implement this is to temporarily override the reorder window to 0.

3. The RACK reordering window SHOULD leverage Duplicate Selective Acknowledgement (DSACK) information [RFC3708] to adaptively estimate the duration of reordering events.

4. The RACK reordering window MUST be bounded and this bound SHOULD be one round trip.

As a flow starts, either condition 1 or condition 2 or both would trigger RACK to start the recovery process quickly. The low initial reordering window and use of the 3-DUPACK rule are key to achieving low-latency loss recovery for short flows by risking spurious retransmissions to recover losses quickly. This rationale is that spurious retransmissions for short flows are not expected to produce excessive network traffic.

For long flows the design tolerates reordering within a round trip. This handles reordering caused by path divergence in small time scales (reordering within the round-trip time of the shortest path), which should tolerate much of the reordering from link bonding, multipath routing, or link-layer out-of-order delivery. It also relaxes ordering constraints to allow sending flights of TCP packets on different paths dynamically for better load-balancing (e.g. flowlets).

However, the fact that the initial RACK reordering window is low, and the RACK reordering window's adaptive growth is bounded, means that there will continue to be a cost to reordering and a limit to RACK's adaptation to reordering. This maintains a disincentive for network designers and operators to introduce needless or excessive reordering, particularly because they have to allow for low round trip time paths. This means RACK will not encourage networks to perform inconsiderate fine-grained packet-spraying over highly disjoint paths with very different characteristics. There are good alternative solutions, such as MPTCP, for such networks.

To conclude, the RACK algorithm aims to adapt to small degrees of reordering, quickly recover most losses within one to two round trips, and avoid costly retransmission timeouts (RTOs). In the presence of reordering, the adaptation algorithm can impose sometimes-needless delays when it waits to disambiguate loss from reordering, but the penalty for waiting is bounded to one round trip and such delays are confined to longer-running flows.

This document provides a concrete and detailed reordering window adaptation algorithm for implementors. We note that the specifics of the algorithm are likely to evolve over time. But that is a separate engineering optimization that's out of scope for this document.

4. Requirements

The reader is expected to be familiar with the definitions given in the TCP congestion control [RFC5681] and selective acknowledgment [RFC2018] RFCs. Familiarity with the conservative SACK-based recovery for TCP [RFC6675] is not expected but helps.

RACK has three requirements:

1. The connection MUST use selective acknowledgment (SACK) options [RFC2018].

2. For each packet sent, the sender MUST store its most recent transmission time with (at least) millisecond granularity. For round-trip times lower than a millisecond (e.g., intra-datacenter communications) microsecond granularity would significantly help the detection latency but is not required.

3. For each packet sent, the sender MUST remember whether the packet has been retransmitted or not.

We assume that requirement 1 implies the sender keeps a SACK scoreboard, which is a data structure to store selective acknowledgment information on a per-connection basis ([RFC6675] section 3). For the ease of explaining the algorithm, we use a pseudo-scoreboard that manages the data in sequence number ranges. But the specifics of the data structure are left to the implementor.

RACK does not need any change on the receiver.

5. Definitions

5.1. 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 [RFC2119 [1]]. In this document, these words will appear with that interpretation only when in UPPER CASE. Lower case uses of these words are not to be interpreted as carrying [RFC2119 [2]] significance.

The reader is expected to be familiar with the definitions given in [RFC793], including SND.UNA, SND.NXT, SEG.ACK, and SEG.SEQ.

5.2. Definitions of variables

A sender implementing RACK needs to store these new RACK variables:

"Packet.xmit_ts" is the time of the last transmission of a data packet, including retransmissions, if any. The sender needs to record the transmission time for each packet sent and not yet acknowledged. The time MUST be stored at millisecond granularity or finer.

"RACK.packet". Among all the packets that have been either selectively or cumulatively acknowledged, RACK.packet is the one that was sent most recently (including retransmissions).

"RACK.xmit_ts" is the latest transmission timestamp of RACK.packet.

"RACK.end_seq" is the ending TCP sequence number of RACK.packet.

"RACK.rtt" is the RTT of the most recently transmitted packet that has been delivered (either cumulatively acknowledged or selectively acknowledged) on the connection.

"RACK.rtt_seq" is the SND.NXT when RACK.rtt is updated.

"RACK.reo_wnd" is a reordering window computed in the unit of time used for recording packet transmission times. It is used to defer the moment at which RACK marks a packet lost.

"RACK.dupthresh" is a constant specifying the number of duplicate acknowledgments, or selectively acknowledged segments, that can (under certain conditions) trigger fast recovery, similar to [RFC6675]. As in [RFC5681] and [RFC6675], this threshold is defined to be 3.

"RACK.min_RTT" is the estimated minimum round-trip time (RTT) of the connection.

"RACK.ack_ts" is the time when all the sequences in RACK.packet were selectively or cumulatively acknowledged.

"RACK.reo_wnd_incr" is the multiplier applied to adjust RACK.reo_wnd

"RACK.reo_wnd_persist" is the number of loss recoveries before resetting RACK.reo_wnd

"RACK.dsack" indicates if a DSACK option has been received since last RACK.reo_wnd change "RACK.pkts_sacked" returns the total number of packets selectively acknowledged in the SACK scoreboard.

"RACK.reord" indicates the connection has detected packet reordering event(s)

"RACK.fack" is the highest selectively or cumulatively acknowledged sequence

Note that the Packet.xmit_ts variable is per packet in flight. The RACK.xmit_ts, RACK.end_seq, RACK.rtt, RACK.reo_wnd, and RACK.min_RTT variables are kept in the per-connection TCP control block. RACK.packet and RACK.ack_ts are used as local variables in the algorithm.

6. Algorithm Details

6.1. Transmitting a data packet

Upon transmitting a new packet or retransmitting an old packet, record the time in Packet.xmit_ts. RACK does not care if the retransmission is triggered by an ACK, new application data, an RTO, or any other means.

6.2. Upon receiving an ACK

Step 1: Update RACK.min_RTT.

Use the RTT measurements obtained via [RFC6298] or [RFC7323] to update the estimated minimum RTT in RACK.min_RTT. The sender can track a simple global minimum of all RTT measurements from the connection, or a windowed min-filtered value of recent RTT measurements. This document does not specify an exact approach.

Step 2: Update RACK stats

Given the information provided in an ACK, each packet cumulatively ACKed or SACKed is marked as delivered in the scoreboard. Among all the packets newly ACKed or SACKed in the connection, record the most recent Packet.xmit_ts in RACK.xmit_ts if it is ahead of RACK.xmit_ts. Sometimes the timestamps of RACK.Packet and Packet could carry the same transmit timestamps due to clock granularity or segmentation offloading (i.e. the two packets were sent as a jumbo frame into the NIC). In that case the sequence numbers of RACK.end_seq and Packet.end_seq are compared to break the tie.

Since an ACK can also acknowledge retransmitted data packets, RACK.rtt can be vastly underestimated if the retransmission was spurious. To avoid that, ignore a packet if any of its TCP sequences have been retransmitted before and either of two conditions is true:

1. The Timestamp Echo Reply field (TSecr) of the ACK's timestamp option [RFC7323], if available, indicates the ACK was not acknowledging the last retransmission of the packet.

2. The packet was last retransmitted less than RACK.min_rtt ago.

If the ACK is not ignored as invalid, update the RACK.rtt to be the RTT sample calculated using this ACK, and continue. If this ACK or SACK was for the most recently sent packet, then record the RACK.xmit_ts timestamp and RACK.end_seq sequence implied by this ACK. Otherwise exit here and omit the following steps.

Notice that the second condition above is a heuristic. This heuristic would fail to update RACK stats if the packet is spuriously retransmitted because of a recent minimum RTT decrease (e.g. path change). Consequentially RACK may not detect losses from ACK events and the recovery resorts to the (slower) TLP or RTO timer-based events. However such events should be rare and the connection would pick up the new minimum RTT when the recovery ends to avoid repeated similar failures.

Step 2 may be summarized in pseudocode as:

```
RACK_sent_after(t1, seq1, t2, seq2):
    If t1 > t2:
        Return true
    Else if t1 == t2 AND seq1 > seq2:
        Return true
    Else:
        Return false

RACK_update():
    For each Packet newly acknowledged cumulatively or selectively:
        rtt = Now() - Packet.xmit_ts
        If Packet.retransmitted is TRUE:
            If ACK.ts_option.echo_reply < Packet.xmit_ts:
               Return
            If rtt < RACK.min_rtt:
               Return

        RACK.rtt = rtt
        If RACK_sent_after(Packet.xmit_ts, Packet.end_seq
                           RACK.xmit_ts, RACK.end_seq):
            RACK.xmit_ts = Packet.xmit_ts
```

Step 3: Detect packet reordering

To detect reordering, the sender looks for original data packets being delivered out of order in sequence space. The sender tracks the highest sequence selectively or cumulatively acknowledged in the RACK.fack variable. The name fack stands for the most forward ACK originated from the [FACK] draft. If the ACK selectively or cumulatively acknowledges an unacknowledged and also never retransmitted sequence below RACK.fack, then the corresponding packet has been reordered and RACK.reord is set to TRUE.

The heuristic above only detects reordering if the re-ordered packet has not yet been retransmitted. This is a major drawback because if RACK has a low reordering window and the network is reordering packets, RACK may falsely retransmit frequently. Consequently RACK may fail to detect reordering to increase the reordering window, because the reordered packets were already (falsely) retransmitted.

DSACK [RFC3708] can help mitigate this issue. The false retransmission would solicit DSACK option in the ACK. Therefore if the ACK has a DSACK option covering some sequence that were both acknowledged and retransmitted, this implies the original packet was reordered but RACK retransmitted the packet too quickly and should set RACK.reord to TRUE.

```
RACK_detect_reordering():
    For each Packet newly acknowledged cumulatively or selectively:
        If Packet.end_seq > RACK.fack:
            RACK.fack = Packet.end_seq
        Else if Packet.end_seq < RACK.fack AND
                Packet.retransmitted is FALSE:
            RACK.reord = TRUE

    For each Packet covered by the DSACK option:
        If Packet.retransmitted is TRUE:
            RACK.reord = TRUE
```

Step 4: Update RACK reordering window

To handle the prevalent small degree of reordering, RACK.reo_wnd serves as an allowance for settling time before marking a packet lost. This section documents a detailed algorithm following the design rationale section. RACK starts initially with a conservative window of min_RTT/4. If no reordering has been observed, RACK uses RACK.reo_wnd of 0 during loss recovery, in order to retransmit quickly, or when the number of DUPACKs exceeds the classic DUPACK threshold. The subtle difference of this approach and conventional one [RFC5681][RFC6675] is dicussed later in the section of "RACK and TLP discussions".

Further, RACK MAY use DSACK [RFC3708] to adapt the reordering window, to higher degrees of reordering, if DSACK is supported. Receiving an ACK with a DSACK indicates a spurious retransmission, which in turn suggests that the RACK reordering window, RACK.reo_wnd, is likely too small. The sender MAY increase the RACK.reo_wnd window linearly for every round trip in which the sender receives a DSACK, so that after N distinct round trips in which a DSACK is received, the RACK.reo_wnd becomes (N+1) * min_RTT / 4, with an upper-bound of SRTT. The inflated RACK.reo_wnd would persist for 16 loss recoveries and after which it resets to its starting value, min_RTT / 4.

The following pseudocode implements above algorithm. Note that extensions that require additional TCP features (e.g. DSACK) would work if the feature functions simply return false.

```
RACK_update_reo_wnd():
    RACK.min_RTT = TCP_min_RTT()
    If DSACK option is present:
        RACK.dsack = true

    If SND.UNA < RACK.rtt_seq:
        RACK.dsack = false  /* React to DSACK once per round trip */

    If RACK.dsack:
        RACK.reo_wnd_incr += 1
        RACK.dsack = false
        RACK.rtt_seq = SND.NXT
        RACK.reo_wnd_persist = 16 /* Keep window for 16 recoveries */
    Else if exiting loss recovery:
        RACK.reo_wnd_persist -= 1
        If RACK.reo_wnd_persist <= 0:
            RACK.reo_wnd_incr = 1


    If RACK.reord is FALSE:
        If in loss recovery:  /* If in fast or timeout recovery */
            RACK.reo_wnd = 0
            Return
        Else if RACK.pkts_sacked >= RACK.dupthresh:
            RACK.reo_wnd = 0
            return
    RACK.reo_wnd = RACK.min_RTT / 4 * RACK.reo_wnd_incr
    RACK.reo_wnd = min(RACK.reo_wnd, SRTT)
```

Step 5: Detect losses.

For each packet that has not been SACKed, if RACK.xmit_ts is after Packet.xmit_ts + RACK.reo_wnd, then mark the packet (or its

corresponding sequence range) lost in the scoreboard. The rationale is that if another packet that was sent later has been delivered, and the reordering window or "reordering settling time" has already passed, then the packet was likely lost.

If another packet that was sent later has been delivered, but the reordering window has not passed, then it is not yet safe to deem the unacked packet lost. Using the basic algorithm above, the sender would wait for the next ACK to further advance RACK.xmit_ts; but this risks a timeout (RTO) if no more ACKs come back (e.g, due to losses or application limit). For timely loss detection, the sender MAY install a "reordering settling" timer set to fire at the earliest moment at which it is safe to conclude that some packet is lost. The earliest moment is the time it takes to expire the reordering window of the earliest unacked packet in flight.

This timer expiration value can be derived as follows. As a starting point, we consider that the reordering window has passed if the RACK.packet was sent sufficiently after the packet in question, or a sufficient time has elapsed since the RACK.packet was S/ACKed, or some combination of the two. More precisely, RACK marks a packet as lost if the reordering window for a packet has elapsed through the sum of:

1. delta in transmit time between a packet and the RACK.packet

2. delta in time between RACK.ack_ts and now

So we mark a packet as lost if:

```
 RACK.xmit_ts >= Packet.xmit_ts
        AND
 (RACK.xmit_ts - Packet.xmit_ts) + (now - RACK.ack_ts) >= RACK.reo_wnd
```

If we solve this second condition for "now", the moment at which we can declare a packet lost, then we get:

```
now >= Packet.xmit_ts + RACK.reo_wnd + (RACK.ack_ts - RACK.xmit_ts)
```

Then (RACK.ack_ts - RACK.xmit_ts) is just the RTT of the packet we used to set RACK.xmit_ts, so this reduces to:

```
Packet.xmit_ts + RACK.rtt + RACK.reo_wnd - now <= 0
```

The following pseudocode implements the algorithm above. When an ACK is received or the RACK timer expires, call RACK_detect_loss(). The algorithm includes an additional optimization to break timestamp ties by using the TCP sequence space. The optimization is particularly

useful to detect losses in a timely manner with TCP Segmentation Offload, where multiple packets in one TSO blob have identical timestamps. It is also useful when the timestamp clock granularity is close to or longer than the actual round trip time.

```
RACK_detect_loss():
    timeout = 0

    For each packet, Packet, not acknowledged yet:
        If Packet.lost is TRUE AND Packet.retransmitted is FALSE:
            Continue /* Packet lost but not yet retransmitted */

        If RACK_sent_after(RACK.xmit_ts, RACK.end_seq,
                           Packet.xmit_ts, Packet.end_seq):
            remaining = Packet.xmit_ts + RACK.rtt +
                        RACK.reo_wnd - Now()
            If remaining <= 0:
                Packet.lost = TRUE
            Else:
                timeout = max(remaining, timeout)

    If timeout != 0
        Arm a timer to call RACK_detect_loss() after timeout
```

Implementation optimization: looping through packets in the SACK scoreboard above could be very costly on large-BDP networks since the inflight could be very large. If the implementation can organize the scoreboard data structures to have packets sorted by the last (re)transmission time, then the loop can start on the least recently sent packet and abort on the first packet sent after RACK.time_ts. This can be implemented by using a seperate list sorted in time order. The implementation inserts the packet at the tail of the list when it is (re)transmitted, and removes a packet from the list when it is delivered or marked lost. We RECOMMEND such an optimization because it enables implementations to support high-BDP networks. This optimization is implemented in Linux and sees orders of magnitude improvement in CPU usage on high-speed WAN networks.

6.3. Tail Loss Probe: fast recovery for tail losses

This section describes a supplemental algorithm, Tail Loss Probe (TLP), which leverages RACK to further reduce RTO recoveries. TLP triggers fast recovery to quickly repair tail losses that can otherwise be recovered by RTOs only. After an original data transmission, TLP sends a probe data segment within one to two RTTs. The probe data segment can either be new, previously unsent data, or a retransmission of previously sent data just below SND.NXT. In either case the goal is to elicit more feedback from the receiver, in

the form of an ACK (potentially with SACK blocks), to allow RACK to trigger fast recovery instead of an RTO.

An RTO occurs when the first unacknowledged sequence number is not acknowledged after a conservative period of time has elapsed [RFC6298]. Common causes of RTOs include:

1. The entire flight of data is lost.

2. Tail losses of data segments at the end of an application transaction.

3. Tail losses of ACKs at the end of an application transaction.

4. Lost retransmits, which can halt fast recovery based on [RFC6675] if the ACK stream completely dries up. For example, consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. On receipt of a SACK for P3, RACK marks P1 and P2 as lost and retransmits them as R1 and R2. Suppose R1 and R2 are lost as well, so there are no more returning ACKs to detect R1 and R2 as lost. Recovery stalls.

5. An unexpectedly long round-trip time (RTT). This can cause ACKs to arrive after the RTO timer expires. The F-RTO algorithm [RFC5682] is designed to detect such spurious retransmission timeouts and at least partially undo the consequences of such events, but F-RTO cannot be used in many situations.

6.4. Tail Loss Probe: An Example

Following is an example of TLP. All events listed are at a TCP sender.

1. Sender transmits segments 1-10: 1, 2, 3, ..., 8, 9, 10. There is no more new data to transmit. A PTO is scheduled to fire in 2 RTTs, after the transmission of the 10th segment.

2. Sender receives acknowledgements (ACKs) for segments 1-5; segments 6-10 are lost and no ACKs are received. The sender reschedules its PTO timer relative to the last received ACK, which is the ACK for segment 5 in this case. The sender sets the PTO interval using the calculation described in step (2) of the algorithm.

3. When PTO fires, sender retransmits segment 10.

4. After an RTT, a SACK for packet 10 arrives. The ACK also carries SACK holes for segments 6, 7, 8 and 9. This triggers RACK-based loss recovery.

5. The connection enters fast recovery and retransmits the remaining lost segments.

6.5. Tail Loss Probe Algorithm Details

We define the terminology used in specifying the TLP algorithm:

FlightSize: amount of outstanding data in the network, as defined in [RFC5681].

RTO: The transport's retransmission timeout (RTO) is based on measured round-trip times (RTT) between the sender and receiver, as specified in [RFC6298] for TCP. PTO: Probe timeout (PTO) is a timer event indicating that an ACK is overdue. Its value is constrained to be smaller than or equal to an RTO.

SRTT: smoothed round-trip time, computed as specified in [RFC6298].

The TLP algorithm has three phases, which we discuss in turn.

6.5.1. Phase 1: Scheduling a loss probe

Step 1: Check conditions for scheduling a PTO.

A sender should check to see if it should schedule a PTO in the following situations:

1. After transmitting new data that was not itself a TLP probe

2. Upon receiving an ACK that cumulatively acknowledges data

3. Upon receiving a SACK that selectively acknowledges data that was last sent before the segment with SEG.SEQ=SND.UNA was last (re)transmitted

A sender should schedule a PTO only if all of the following conditions are met:

1. The connection supports SACK [RFC2018]

2. The connection has no SACKed sequences in the SACK scoreboard

3. The connection is not in loss recovery

If a PTO can be scheduled according to these conditions, the sender should schedule a PTO. If there was a previously scheduled PTO or RTO pending, then that pending PTO or RTO should first be cancelled, and then the new PTO should be scheduled.

If a PTO cannot be scheduled according to these conditions, then the sender MUST arm the RTO timer if there is unacknowledged data in flight.

Step 2: Select the duration of the PTO.

A sender SHOULD use the following logic to select the duration of a PTO:

```
TLP_timeout():
    If SRTT is available:
        PTO = 2 * SRTT
        If FlightSize = 1:
           PTO += WCDelAckT
        Else:
           PTO += 2ms
    Else:
        PTO = 1 sec

    If Now() + PTO > TCP_RTO_expire():
        PTO = TCP_RTO_expire() - Now()
```

Aiming for a PTO value of 2*SRTT allows a sender to wait long enough to know that an ACK is overdue. Under normal circumstances, i.e. no losses, an ACK typically arrives in one SRTT. But choosing PTO to be exactly an SRTT is likely to generate spurious probes given that network delay variance and even end-system timings can easily push an ACK to be above an SRTT. We chose PTO to be the next integral multiple of SRTT.

Similarly, network delay variations and end-system processing latencies and timer granularities can easily delay ACKs beyond 2*SRTT, so senders SHOULD add at least 2ms to a computed PTO value (and MAY add more if the sending host OS timer granularity is more coarse than 1ms).

WCDelAckT stands for worst case delayed ACK timer. When FlightSize is 1, PTO is inflated by WCDelAckT time to compensate for a potential long delayed ACK timer at the receiver. The RECOMMENDED value for WCDelAckT is 200ms.

Finally, if the time at which an RTO would fire (here denoted "TCP_RTO_expire") is sooner than the computed time for the PTO, then a probe is scheduled to be sent at that earlier time.

6.5.2. Phase 2: Sending a loss probe

When the PTO fires, transmit a probe data segment:

```
TLP_send_probe():
    If an unsent segment exists AND
       the receive window allows new data to be sent:
        Transmit the lowest-sequence unsent segment of up to SMSS
        Increment FlightSize by the size of the newly-sent segment
    Else:
        Retransmit the highest-sequence segment sent so far
    The cwnd remains unchanged
```

When the loss probe is a retransmission, the sender uses the highest-sequence segment sent so far. This is in order to deal with the retransmission ambiguity problem in TCP. Suppose a sender sends N segments, and then retransmits the last segment (segment N) as a loss probe, and then the sender receives a SACK for segment N. As long as the sender waits for any required RACK reordering settling timer to then expire, it doesn't matter if that SACK was for the original transmission of segment N or the TLP retransmission; in either case the arrival of the SACK for segment N provides evidence that the N-1 segments preceding segment N were likely lost. In the case where there is only one original outstanding segment of data (N=1), the same logic (trivially) applies: an ACK for a single outstanding segment tells the sender the N-1=0 segments preceding that segment were lost. Furthermore, whether there are N>1 or N=1 outstanding segments, there is a question about whether the original last segment or its TLP retransmission were lost; the sender estimates this using TLP recovery detection (see below).

Note that after transmitting a TLP, the sender MUST arm an RTO timer, and not the PTO timer. This ensures that the sender does not send repeated, back-to-back TLP probes. This is important to avoid TLP loops if an application writes periodically at an interval less than PTO.

6.5.3. Phase 3: ACK processing

On each incoming ACK, the sender should check the conditions in Step 1 of Phase 1 to see if it should schedule (or reschedule) the loss probe timer.

6.6. TLP recovery detection

If the only loss in an outstanding window of data was the last segment, then a TLP loss probe retransmission of that data segment might repair the loss. TLP recovery detection examines ACKs to detect when the probe might have repaired a loss, and thus allows congestion control to properly reduce the congestion window (cwnd) [RFC5681].

Consider a TLP retransmission episode where a sender retransmits a tail packet in a flight. The TLP retransmission episode ends when the sender receives an ACK with a SEG.ACK above the SND.NXT at the time the episode started. During the TLP retransmission episode the sender checks for a duplicate ACK or D-SACK indicating that both the original segment and TLP retransmission arrived at the receiver, meaning there was no loss that needed repairing. If the TLP sender does not receive such an indication before the end of the TLP retransmission episode, then it MUST estimate that either the original data segment or the TLP retransmission were lost, and congestion control MUST react appropriately to that loss as it would any other loss.

Since a significant fraction of the hosts that support SACK do not support duplicate selective acknowledgments (D-SACKs) [RFC2883] the TLP algorithm for detecting such lost segments relies only on basic SACK support [RFC2018].

Definitions of variables

TLPRxtOut: a boolean indicating whether there is an unacknowledged TLP retransmission.

TLPHighRxt: the value of SND.NXT at the time of sending a TLP retransmission.

6.6.1. Initializing and resetting state

When a connection is created, or suffers a retransmission timeout, or enters fast recovery, it executes the following:

```
TLPRxtOut = false
```

6.6.2. Recording loss probe states

Senders MUST only send a TLP loss probe retransmission if TLPRxtOut is false. This ensures that at any given time a connection has at most one outstanding TLP retransmission. This allows the sender to

use the algorithm described in this section to estimate whether any data segments were lost.

Note that this condition only restricts TLP loss probes that are retransmissions. There may be an arbitrary number of outstanding unacknowledged TLP loss probes that consist of new, previously-unsent data, since the retransmission timeout and fast recovery algorithms are sufficient to detect losses of such probe segments.

Upon sending a TLP probe that is a retransmission, the sender sets TLPRxtOut to true and TLPHighRxt to SND.NXT.

Detecting recoveries accomplished by loss probes

Step 1: Track ACKs indicating receipt of original and retransmitted segments

A sender considers both the original segment and TLP probe retransmission segment as acknowledged if either 1 or 2 are true:

1. This is a duplicate acknowledgment (as defined in [RFC5681], section 2), and all of the following conditions are met:

    1. TLPRxtOut is true

    2. SEG.ACK == TLPHighRxt

    3. SEG.ACK == SND.UNA

    4. the segment contains no SACK blocks for sequence ranges above TLPHighRxt

    5. the segment contains no data

    6. the segment is not a window update

2. This is an ACK acknowledging a sequence number at or above TLPHighRxt and it contains a D-SACK; i.e. all of the following conditions are met:

    1. TLPRxtOut is true

    2. SEG.ACK >= TLPHighRxt

    3. the ACK contains a D-SACK block

If either of the conditions is met, then the sender estimates that the receiver received both the original data segment and the TLP

probe retransmission, and so the sender considers the TLP episode to be done, and records that fact by setting TLPRxtOut to false.

Step 2: Mark the end of a TLP retransmission episode and detect losses

If the sender receives a cumulative ACK for data beyond the TLP loss probe retransmission then, in the absence of reordering on the return path of ACKs, it should have received any ACKs for the original segment and TLP probe retransmission segment. At that time, if the TLPRxtOut flag is still true and thus indicates that the TLP probe retransmission remains unacknowledged, then the sender should presume that at least one of its data segments was lost, so it SHOULD invoke a congestion control response equivalent to fast recovery.

More precisely, on each ACK the sender executes the following:

```
if (TLPRxtOut and SEG.ACK >= TLPHighRxt) {
    TLPRxtOut = false
    EnterRecovery()
    ExitRecovery()
}
```

7. RACK and TLP discussions

7.1. Advantages

The biggest advantage of RACK is that every data packet, whether it is an original data transmission or a retransmission, can be used to detect losses of the packets sent chronologically prior to it.

Example: TAIL DROP. Consider a sender that transmits a window of three data packets (P1, P2, P3), and P1 and P3 are lost. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond by default) after the transmission of the previous packet. RACK will mark P1 as lost when the SACK of P2 is received, and this will trigger the retransmission of P1 as R1. When R1 is cumulatively acknowledged, RACK will mark P3 as lost and the sender will retransmit P3 as R3. This example illustrates how RACK is able to repair certain drops at the tail of a transaction without any timer. Notice that neither the conventional duplicate ACK threshold [RFC5681], nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses, because of the required packet or sequence count.

Example: LOST RETRANSMIT. Consider a window of three data packets (P1, P2, P3) that are sent; P1 and P2 are dropped. Suppose the transmission of each packet is at least RACK.reo_wnd (1 millisecond

by default) after the transmission of the previous packet. When P3 is SACKed, RACK will mark P1 and P2 lost and they will be retransmitted as R1 and R2. Suppose R1 is lost again but R2 is SACKed; RACK will mark R1 lost for retransmission again. Again, neither the conventional three duplicate ACK threshold approach, nor [RFC6675], nor the Forward Acknowledgment [FACK] algorithm can detect such losses. And such a lost retransmission is very common when TCP is being rate-limited, particularly by token bucket policers with large bucket depth and low rate limit. Retransmissions are often lost repeatedly because standard congestion control requires multiple round trips to reduce the rate below the policed rate.

Example: SMALL DEGREE OF REORDERING. Consider a common reordering event: a window of packets are sent as (P1, P2, P3). P1 and P2 carry a full payload of MSS octets, but P3 has only a 1-octet payload. Suppose the sender has detected reordering previously and thus RACK.reo_wnd is min_RTT/4. Now P3 is reordered and delivered first, before P1 and P2. As long as P1 and P2 are delivered within min_RTT/4, RACK will not consider P1 and P2 lost. But if P1 and P2 are delivered outside the reordering window, then RACK will still falsely mark P1 and P2 lost. We discuss how to reduce false positives in the end of this section.

The examples above show that RACK is particularly useful when the sender is limited by the application, which is common for interactive, request/response traffic. Similarly, RACK still works when the sender is limited by the receive window, which is common for applications that use the receive window to throttle the sender.

For some implementations (e.g., Linux), RACK works quite efficiently with TCP Segmentation Offload (TSO). RACK always marks the entire TSO blob lost because the packets in the same TSO blob have the same transmission timestamp. By contrast, the algorithms based on sequence counting (e.g., [RFC6675][RFC5681]) may mark only a subset of packets in the TSO blob lost, forcing the stack to perform expensive fragmentation of the TSO blob, or to selectively tag individual packets lost in the scoreboard.

7.2. Disadvantages

RACK requires the sender to record the transmission time of each packet sent at a clock granularity of one millisecond or finer. TCP implementations that record this already for RTT estimation do not require any new per-packet state. But implementations that are not yet recording packet transmission times will need to add per-packet internal state (commonly either 4 or 8 octets per packet or TSO blob) to track transmission times. In contrast, the conventional [RFC6675] loss detection approach does not require any per-packet state beyond

the SACK scoreboard. This is particularly useful on ultra-low RTT networks where the RTT is far less than the sender TCP clock grainularity (e.g. inside data-centers).

RACK can easily and optionally support the conventional approach in [RFC6675][RFC5681] by resetting the reordering window to zero when the threshold is met. Note that this approach differs slightly from [RFC6675] which considers a packet lost when at least #DupThresh higher-sequenc packets are SACKed. RACK's approach considers a packet lost when at least one higher sequence packet is SACKed and the total number of SACKed packets is at least DupThresh. For example, suppose a connection sends 10 packets, and packets 3, 5, 7 are SACKed. [RFC6675] considers packets 1 and 2 lost. RACK considers packets 1, 2, 4, 6 lost.

7.3. Adjusting the reordering window

When the sender detects packet reordering, RACK uses a reordering window of min_rtt / 4. It uses the minimum RTT to accommodate reordering introduced by packets traversing slightly different paths (e.g., router-based parallelism schemes) or out-of-order deliveries in the lower link layer (e.g., wireless links using link-layer retransmission). RACK uses a quarter of minimum RTT because Linux TCP used the same factor in its implementation to delay Early Retransmit [RFC5827] to reduce spurious loss detections in the presence of reordering, and experience shows that this seems to work reasonably well. We have evaluated using the smoothed RTT (SRTT from [RFC6298] RTT estimation) or the most recently measured RTT (RACK.rtt) using an experiment similar to that in the Performance Evaluation section. They do not make any significant difference in terms of total recovery latency.

7.4. Relationships with other loss recovery algorithms

The primary motivation of RACK is to ultimately provide a simple and general replacement for some of the standard loss recovery algorithms [RFC5681][RFC6675][RFC5827][RFC4653], as well as some nonstandard ones [FACK][THIN-STREAM]. While RACK can be a supplemental loss detection mechanism on top of these algorithms, this is not necessary, because RACK implicitly subsumes most of them.

[RFC5827][RFC4653][THIN-STREAM] dynamically adjusts the duplicate ACK threshold based on the current or previous flight sizes. RACK takes a different approach, by using only one ACK event and a reordering window. RACK can be seen as an extended Early Retransmit [RFC5827] without a FlightSize limit but with an additional reordering window. [FACK] considers an original packet to be lost when its sequence range is sufficiently far below the highest SACKed sequence. In some

sense RACK can be seen as a generalized form of FACK that operates in time space instead of sequence space, enabling it to better handle reordering, application-limited traffic, and lost retransmissions.

Nevertheless RACK is still an experimental algorithm. Since the oldest loss detection algorithm, the 3 duplicate ACK threshold [RFC5681], has been standardized and widely deployed. RACK can easily and optionally support the conventional approach for compatibility.

RACK is compatible with and does not interfere with the the standard RTO [RFC6298], RTO-restart [RFC7765], F-RTO [RFC5682] and Eifel algorithms [RFC3522]. This is because RACK only detects loss by using ACK events. It neither changes the RTO timer calculation nor detects spurious timeouts.

Furthermore, RACK naturally works well with Tail Loss Probe [TLP] because a tail loss probe solicits either an ACK or SACK, which can be used by RACK to detect more losses. RACK can be used to relax TLP's requirement for using FACK and retransmitting the the highest-sequenced packet, because RACK is agnostic to packet sequence numbers, and uses transmission time instead. Thus TLP could be modified to retransmit the first unacknowledged packet, which could improve application latency.

7.5. Interaction with congestion control

RACK intentionally decouples loss detection from congestion control. RACK only detects losses; it does not modify the congestion control algorithm [RFC5681][RFC6937]. However, RACK may detect losses earlier or later than the conventional duplicate ACK threshold approach does. A packet marked lost by RACK SHOULD NOT be retransmitted until congestion control deems this appropriate. Specifically, Proportional Rate Reduction [RFC6937] SHOULD be used when using RACK.

RACK is applicable for both fast recovery and recovery after a retransmission timeout (RTO) in [RFC5681]. RACK applies equally to fast recovery and RTO recovery because RACK is purely based on the transmission time order of packets. When a packet retransmitted by RTO is acknowledged, RACK will mark any unacked packet sent sufficiently prior to the RTO as lost, because at least one RTT has elapsed since these packets were sent.

The following simple example compares how RACK and non-RACK loss detection interacts with congestion control: suppose a TCP sender has a congestion window (cwnd) of 20 packets on a SACK-enabled connection. It sends 10 data packets and all of them are lost.

Without RACK, the sender would time out, reset cwnd to 1, and retransmit the first packet. It would take four round trips (1 + 2 + 4 + 3 = 10) to retransmit all the 10 lost packets using slow start. The recovery latency would be RTO + 4*RTT, with an ending cwnd of 4 packets due to congestion window validation.

With RACK, a sender would send the TLP after 2*RTT and get a DUPACK. If the sender implements Proportional Rate Reduction [RFC6937] it would slow start to retransmit the remaining 9 lost packets since the number of packets in flight (0) is lower than the slow start threshold (10). The slow start would again take four round trips (1 + 2 + 4 + 3 = 10). The recovery latency would be 2*RTT + 4*RTT, with an ending cwnd set to the slow start threshold of 10 packets.

In both cases, the sender after the recovery would be in congestion avoidance. The difference in recovery latency (RTO + 4*RTT vs 6*RTT) can be significant if the RTT is much smaller than the minimum RTO (1 second in RFC6298) or if the RTT is large. The former case is common in local area networks, data-center networks, or content distribution networks with deep deployments. The latter case is more common in developing regions with highly congested and/or high-latency networks. The ending congestion window after recovery also impacts subsequent data transfer.

7.6. TLP recovery detection with delayed ACKs

Delayed ACKs complicate the detection of repairs done by TLP, since with a delayed ACK the sender receives one fewer ACK than would normally be expected. To mitigate this complication, before sending a TLP loss probe retransmission, the sender should attempt to wait long enough that the receiver has sent any delayed ACKs that it is withholding. The sender algorithm described above features such a delay, in the form of WCDelAckT. Furthermore, if the receiver supports duplicate selective acknowledgments (D-SACKs) [RFC2883] then in the case of a delayed ACK the sender's TLP recovery detection algorithm (see above) can use the D-SACK information to infer that the original and TLP retransmission both arrived at the receiver.

If there is ACK loss or a delayed ACK without a D-SACK, then this algorithm is conservative, because the sender will reduce cwnd when in fact there was no packet loss. In practice this is acceptable, and potentially even desirable: if there is reverse path congestion then reducing cwnd can be prudent.

7.7. RACK for other transport protocols

RACK can be implemented in other transport protocols. The algorithm can be simplified by skipping step 3 if the protocol can support a unique transmission or packet identifier (e.g. TCP timestamp options [RFC7323]). For example, the QUIC protocol implements RACK [QUIC-LR].

8. Experiments and Performance Evaluations

RACK and TLP have been deployed at Google, for both connections to users in the Internet and internally. We conducted a performance evaluation experiment for RACK and TLP on a small set of Google Web servers in Western Europe that serve mostly European and some African countries. The experiment lasted three days in March 2017. The servers were divided evenly into four groups of roughly 5.3 million flows each:

Group 1 (control): RACK off, TLP off, RFC 6675 on

Group 2: RACK on, TLP off, RFC 6675 on

Group 3: RACK on, TLP on, RFC 6675 on

Group 4: RACK on, TLP on, RFC 6675 off

All groups used Linux with CUBIC congestion control, an initial congestion window of 10 packets, and the fq/pacing qdisc. In terms of specific recovery features, all groups enabled RFC5682 (F-RTO) but disabled FACK because it is not an IETF RFC. FACK was excluded because the goal of this setup is to compare RACK and TLP to RFC-based loss recoveries. Since TLP depends on either FACK or RACK, we could not run another group that enables TLP only (with both RACK and FACK disabled). Group 4 is to test whether RACK plus TLP can completely replace the DupThresh-based [RFC6675].

The servers sit behind a load balancer that distributes the connections evenly across the four groups.

Each group handles a similar number of connections and sends and receives similar amounts of data. We compare total time spent in loss recovery across groups. The recovery time is measured from when the recovery and retransmission starts, until the remote host has acknowledged the highest sequence (SND.NXT) at the time the recovery started. Therefore the recovery includes both fast recoveries and timeout recoveries.

Our data shows that Group 2 recovery latency is only 0.3% lower than the Group 1 recovery latency. But Group 3 recovery latency is 25% lower than Group 1 due to a 40% reduction in RTO-triggered recoveries! Therefore it is important to implement both TLP and RACK for performance. Group 4's total recovery latency is 0.02% lower than Group 3's, indicating that RACK plus TLP can successfully replace RFC6675 as a standalone recovery mechanism.

We want to emphasize that the current experiment is limited in terms of network coverage. The connectivity in Western Europe is fairly good, therefore loss recovery is not a major performance bottleneck. We plan to expand our experiments to regions with worse connectivity, in particular on networks with strong traffic policing.

9. Security Considerations

RACK does not change the risk profile for TCP.

An interesting scenario is ACK-splitting attacks [SCWA99]: for an MSS-size packet sent, the receiver or the attacker might send MSS ACKs that SACK or acknowledge one additional byte per ACK. This would not fool RACK. RACK.xmit_ts would not advance because all the sequences of the packet are transmitted at the same time (carry the same transmission timestamp). In other words, SACKing only one byte of a packet or SACKing the packet in entirety have the same effect on RACK.

10. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

11. Acknowledgments

The authors thank Matt Mathis for his insights in FACK and Michael Welzl for his per-packet timer idea that inspired this work. Eric Dumazet, Randy Stewart, Van Jacobson, Ian Swett, Rick Jones, Jana Iyengar, Hiren Panchasara, Praveen Balasubramanian, Yoshifumi Nishida, Bob Briscoe, Felix Weinrank, and Michael Tuexen contributed to the draft or the implementations in Linux, FreeBSD, Windows and QUIC.

12. References

12.1. Normative References

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

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

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

[RFC4737] Morton, A., Ciavattone, L., Ramachandran, G., Shalunov, S., and J. Perser, "Packet Reordering Metrics", RFC 4737, November 2006.

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

[RFC5682] Sarolahti, P., Kojo, M., Yamamoto, K., and M. Hata, "Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP", RFC 5682, September 2009.

[RFC5827] Allman, M., Ayesta, U., Wang, L., Blanton, J., and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)", RFC 5827, April 2010.

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

[RFC6675] Blanton, E., Allman, M., Wang, L., Jarvinen, I., Kojo, M., and Y. Nishida, "A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP", RFC 6675, August 2012.

[RFC6937] Mathis, M., Dukkipati, N., and Y. Cheng, "Proportional Rate Reduction for TCP", May 2013.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, "TCP Extensions for High Performance", September 2014.

[RFC793] Postel, J., "Transmission Control Protocol", September 1981.

12.2. Informative References

[FACK] Mathis, M. and M. Jamshid, "Forward acknowledgement: refining TCP congestion control", ACM SIGCOMM Computer Communication Review, Volume 26, Issue 4, Oct. 1996. , 1996.

[POLICER16] Flach, T., Papageorge, P., Terzis, A., Pedrosa, L., Cheng, Y., Karim, T., Katz-Bassett, E., and R. Govindan, "An Analysis of Traffic Policing in the Web", ACM SIGCOMM , 2016.

[QUIC-LR] Iyengar, J. and I. Swett, "QUIC Loss Recovery And Congestion Control", draft-tsvwg-quic-loss-recovery-01 (work in progress), June 2016.

[RFC7765] Hurtig, P., Brunstrom, A., Petlund, A., and M. Welzl, "TCP and SCTP RTO Restart", February 2016.

[SCWA99] Savage, S., Cardwell, N., Wetherall, D., and T. Anderson, "TCP Congestion Control With a Misbehaving Receiver", ACM Computer Communication Review, 29(5) , 1999.

[THIN-STREAM] Petlund, A., Evensen, K., Griwodz, C., and P. Halvorsen, "TCP enhancements for interactive thin-stream applications", NOSSDAV , 2008.

[TLP] Dukkipati, N., Cardwell, N., Cheng, Y., and M. Mathis, "Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Drops", draft-dukkipati-tcpm-tcp-loss-probe-01 (work in progress), August 2013.

12.3. URIs

[1] https://tools.ietf.org/html/rfc2119

[2] https://tools.ietf.org/html/rfc2119

Authors' Addresses

Yuchung Cheng
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043
USA

Email: ycheng@google.com

Neal Cardwell
Google, Inc
76 Ninth Avenue
New York, NY 10011
USA

Email: ncardwell@google.com

Nandita Dukkipati
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043

Email: nanditad@google.com

Priyaranjan Jha
Google, Inc
1600 Amphitheater Parkway
Mountain View, California 94043

Email: priyarjha@google.com

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: May 3, 2017

Y. Nishida
GE Global Research
H. Asai
The University of Tokyo

October 30, 2016

Increasing Maximum Window Size of TCP
draft-nishida-tcpm-maxwin-02.txt

Abstract

This document proposes to increase the current max window size allowed in TCP. It describes the current logic that limits the max window size and provides a rationale to relax the limitation as well as the negotiation mechanism to enable this feature safely.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 3, 2017.

Table of Contents

1. Introduction

TCP throughput is determined by two factors: Round Trip Time and Receive Window size. It can never exceed Receive Window size divided by RTT. This implies larger window size is important to achive better performance. Original TCP's maximum window size defined in RFC793 [RFC0793] is 2^16 -1 (65,535), however, RFC7323 [RFC7323] defines TCP Window Scale option which allows TCP to use larger window size. Window Scale uses a shift count stored in 1-byte field in the option. The receiver of the option uses left-shifted window size value by the shift count as actual window size. When Window Scale is used, TCP can extend maximum window size to 2^30 - 2^14 (1,073,725,440). This is because the maximum shift count is 14 as described in the Section 2.3 of RFC7323 [RFC7323]. However, since TCP's sequence number space is 2^32, we believe it is still possible to use larger window size than this while careful design of the logic that can identify segments inside the window is required. In this document, we propose to increase the maximum shift count to 15, which extend window size to 2^31 - 2^15.

2. 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 [RFC2119].

3. Increasing Maximum Window Size

RFC7232 requires maximum window size to be less than 2^30 as described below.

"
TCP determines if a data segment is "old" or "new" by testing whether its sequence number is within 2^31 bytes of the left edge of the window, and if it is not, discarding the data as "old". To insure that new data is never mistakenly considered old and vice versa, the left edge of the sender's window has to be at most 2^31 away from the right edge of the receiver's window. The same is true of the sender's right edge and receiver's left edge. Since the right and left edges of either the sender's or receiver's window differ by the window size, and since the sender and receiver windows can be out of phase by at most the window size, the above constraints imply that two times the maximum window size must be less than 2^31, or

max window < 2^30
"

However, TCP does not necessarily need to determine if a segment is old or new. Because important point is to determine if a receive segment is inside of the window or not. It basically does not matter if a segment is too old (left side of the window) or too new (right side of the window) as long as it is outside of the window. Based on this viewpoint, we propose to extend maximum window to 2^31 - 2^15, which can be attained by increasing maximum shift count to 15.

To demonstrate the feasibility of the proposal, we would like to use the following worst case example where the sender and the receiver windows are completely out of phase. In this example, we define S as the sender's left edge of the window and W as the sender's window size. Hence, the sender's right edge of the window is S+W. Also, the receiver's left edge of the window is S+W+1 and the right edge of the window is S+2W+1, as they are out of phase. This situation can happen when the sender sent all segments in the window and the receiver received all segments while no ACK has been received by the sender yet. Now, we presume a segment that contains sequence number S has arrived at the receiver. This segment should be excluded by the receiver, although it can easily happen when the sender retransmits segments.

In case of W=2^31, the receiver cannot exclude this segment as S+2W = S. It is considered inside of the window. (S+W+1 < S < S+2W+1) However, our proposed window size is W=2^31-X, where X is 2^15. In this case, when segment S has arrived, the following checks will be performed. First, TCP checks it with the left edge of the window and it considers the segment is left side of the left edge. (S < S+W+1 Note: W=2^31-X) Second, TCP checks it with the right edge of the window and it considers the segment is right of the right edge. (S > S+2W+1) You might notice that the result of the second check is not expected one as the segment S is actually an old segment. This is

the problem that the referred paragraphs from RFC7232 [RFC7323] describe. However, the segment is properly excluded by the receiver as both checks indicate it is outside of the window. It should be noted that the principle of TCP requires to accept the segment S only when it has passed both checks successfully, which means S must satisfy the following condition.

S >= left edge && S <= right edge

As we have shown in the example, our proposed maximum window size: W=2^31-2^15 does not affect this principle.

4. Usage for New Shift Count

In this proposal, an endpoint that supports new maximum window size simply sets 15 to the shift count in Window Scale option. A potential problem of this approach is that an endpoint cannot verify if the peer supports shift count 15 unless the peer also uses shift count 15. However, we believe this is not a significant problem. As specified in [RFC7323], if an endpoint does not support shift count 15, it simply interprets it as shift count 14. while the window size used by the sender will be smaller than the actual available window size at the receiver, this will not cause any fatal issues

In order to avoid the performance degradation caused by misinterpretation of shift count, an endpoint MAY allocate more than 65535 * 2^15 + 65535 * 2^14 bytes memory for connections that use shift count 15. In this case, the endpoint that supports shiftcount 15 can always offer maximum window value and shift count 15 when the peer does not support new shift count. Because the maximum window value conventional endpoint can consume is 65535 * 2^14 bytes. As long as an endpoint can offer shift count 15 with maximum window value, conventional endpoints won’t have performance issue.

5. Use Cases, Benefits to Explore Maximum Window Size

One of the use cases of the extended maximum window size is high volume data transfer over paths with long RTT delays and high bandwidth, called long fat pipes. The proposed extension improves and doubles at most the maximum throughput when bandwidth-latency product is greater than 1 GB. As propagation delay in an optical fiber is around 20 cm/ns, RTT will be over 100 milliseconds when the distance of the transmission is more than 10000km. This distance is not extraordinary for trans-pacific communications. In this case, the maximum throughput will be limited to 80 Gbps with the current maximum window size, although network technologies for more than 100 Gbps are becoming common these days.

As the current TCP sequence number space is limited to 32 bits, it will not be possible to increase maximum window size any further. However, TCP may eventually have other extensions to increase sequence number space, for example, [RFC7323] and [RFC1263] mention about increasing sequence number space to 64 bits. We believe the information in this document will be useful when such extensions are proposed as they need to define new maximum window size.

6. Acknowledgments

The authors gratefully acknowledge significant inputs for this document from Richard Scheffenegger and Ilpo Jarvinen.

7. Security Considerations

It is known that an attacker can have more chances to insert forged packets into a TCP connection when large window size is used. This is not a specific problem of this proposal, but a generic problem to use larger window. Using PAWS can mitigate this problem, however, it is recommended to consult the Security Considerations section of RFC7323 [RFC7323] to check its implications.

8. IANA Considerations

This document does not create any new registries or modify the rules for any existing registries managed by IANA.

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <http://www.rfc-editor.org/info/rfc7323>.

9.2. Informative References

[RFC1263] O'Malley, S. and L. Peterson, "TCP Extensions Considered Harmful", RFC 1263, DOI 10.17487/RFC1263, October 1991, <http://www.rfc-editor.org/info/rfc1263>.

Authors' Addresses

Yoshifumi Nishida
GE Global Research
2623 Camino Ramon
San Ramon, CA 94583
USA

Email: nishida@wide.ad.jp

Hirochika Asai
The University of Tokyo
7-3-1 Hongo
Bunkyo-ku, Tokyo 113-8656
JP

Email: panda@wide.ad.jp

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: August 3, 2017

Y. Nishida
GE Global Research
H. Asai
The University of Tokyo
M. Bagnulo
UC3M
January 30, 2017

Increasing Maximum Window Size of TCP
draft-nishida-tcpm-maxwin-03.txt

Abstract

This document proposes to increase the current max window size allowed in TCP. It describes the current logic that limits the maximum window size and provides a rationale to relax the limitation as well as the negotiation mechanism to enable this feature safely.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 3, 2017.

Table of Contents

1. Introduction

TCP throughput is determined by two factors: Round Trip Time and Receive Window size. It can never exceed Receive Window size divided by RTT. This implies larger window size is important to achieve better performance. Original TCP's maximum window size defined in RFC793 [RFC0793] is 2^16 -1 (65,535), however, RFC7323 [RFC7323] defines TCP Window Scale option which allows TCP to use larger window size. Window Scale uses a shift count stored in 1-byte field in the option. The receiver of the option uses left-shifted window size value by the shift count as actual window size. When Window Scale is used, TCP can extend maximum window size to 2^30 - 2^14 (1,073,725,440). This is because the maximum shift count is 14 as described in the Section 2.3 of RFC7323 [RFC7323]. However, since TCP's sequence number space is 2^32, we believe it is still possible to use larger window size than this while careful design of the logic that can identify segments inside the window is required. In this document, we propose to increase the maximum shift count to 15, which extend window size to 2^31 - 2^15.

2. 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 [RFC2119].

3. Increasing Maximum Window Size

RFC7323 requires maximum window size to be less than 2^30 as described below.

> "
> TCP determines if a data segment is "old" or "new" by testing whether its sequence number is within 2^31 bytes of the left edge of the window, and if it is not, discarding the data as "old". To insure that new data is never mistakenly considered old and vice versa, the left edge of the sender's window has to be at most 2^31 away from the right edge of the receiver's window. The same is true of the sender's right edge and receiver's left edge. Since the right and left edges of either the sender's or receiver's window differ by the window size, and since the sender and receiver windows can be out of phase by at most the window size, the above constraints imply that two times the maximum window size must be less than 2^31, or
>
> max window < 2^30
>
> "

However, TCP does not necessarily need to determine if a segment is old or new. Because important point is to determine if a receive segment is inside of the window or not. It basically does not matter if a segment is too old (left side of the window) or too new (right side of the window) as long as it is outside of the window. Based on this viewpoint, we propose to extend maximum window to 2^31 - 2^15, which can be attained by increasing maximum shift count to 15.

To demonstrate the feasibility of the proposal, we would like to use the following worst case example where the sender and the receiver windows are completely out of phase. In this example, we define S as the sender's left edge of the window and W as the sender's window size. Hence, the sender's right edge of the window is S+W. Also, the receiver's left edge of the window is S+W+1 and the right edge of the window is S+2W+1, as they are out of phase. This situation can happen when the sender sent all segments in the window and the receiver received all segments while no ACK has been received by the sender yet. Now, we presume a segment that contains sequence number S has arrived at the receiver. This segment should be excluded by the receiver, although it can easily happen when the sender retransmits segments.

In case of W=2^31, the receiver cannot exclude this segment as S+2W = S. It is considered inside of the window. (S+W+1 < S < S+2W+1) However, our proposed window size is W=2^31-X, where X is 2^15. In this case, when segment S has arrived, the following checks will be performed. First, TCP checks it with the left edge of the window and

it considers the segment is left side of the left edge. (S < S+W+1 Note: W=2^31-X) Second, TCP checks it with the right edge of the window and it considers the segment is right of the right edge. (S > S+2W+1) You might notice that the result of the second check is not expected one as the segment S is actually an old segment. This is the problem that the referred paragraphs from RFC7323 [RFC7323] describe. However, the segment is properly excluded by the receiver as both checks indicate it is outside of the window. It should be noted that the principle of TCP requires to accept the segment S only when it has passed both checks successfully, which means S must satisfy the following condition.

S >= left edge && S <= right edge

As we have shown in the example, our proposed maximum window size: W=2^31-2^15 does not affect this principle.

Using the larger window size implies that the sequence number space can wrap around in less than 3 RTTs. This can pose problems to distinguish old retransmitted packets from new packets solely using the same sequence number. Because of this, a sender using the larger window size defined in this specification is recommended to use Protection Against Wrapped Sequences (PAWS) as defined in RFC7323 [RFC7323].

4. Updating the Window Scale Option

As shown in Figure 1, the Window Scale Option (WSO) defined in [RFC7323] has three 1-byte fields, the Kind field (which specifies the option type), the Length field (set to 3 because the WSO is 3 bytes long) and the shift.cnt field (which specifies the shift count applied to the window to scale it).

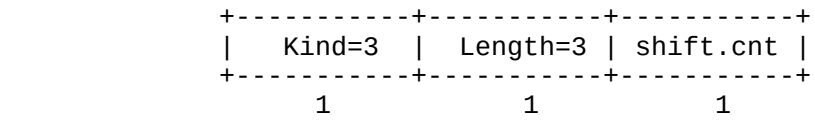


Figure 1: Window Scale Option (WSO) format

RFC7323 [RFC7323] defines that the shift.cnt field can have a maximum value of 14 and upon reception of a larger value in this field, the receiver must proceed as if it had received a shift.cnt of 14.

This specification updates the shift.cnt field definition. Figure 2 represents the new format of the shift.cnt field. The eight bits contained in the shift.cnt field are formatted as "SSSSLRRR".

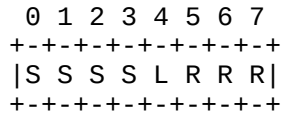


Figure 2: New shift.cnt field format

These bits are parsed as follows:

- The four leftmost bits "SSSS" express the shift-count, as in RFC7323 [RFC7323], only that now the maximum shift count value allowed is 15.

- The "L" bit expresses if the sender supports the large window defined in this specification i.e. the bit is set if the sender supports this specification.

- The three rightmost bits "RRR" are reserved for future use and MUST be set to zero.

This new format for the shift.count field allows an updated client to initiate a TCP connection and express that it supports the larger window by setting the "L" bit, while still conveying information about the shift count that it wants to use for its own RCV.WND in the four leftmost bits "SSSS" (which do not necessarily have to be set to 15). A server that supports this specification that receives a SYN with the WSO with the "L" bit set knows that it can reply using a shift count of 15. A legacy server that receives the WSO with the "L"" bit set will interpret it using the RFC7323 format and will then read it as a shift count value larger than 14. As per RFC7323 the server MUST then assume a shift count of 14. The legacy server will then reply with a WSO with the "L" bit set to zero, so the client knows that the server does not support this specification and that the server will assume a shift count of 14 for the client’s receive window.

5. Use Cases, Benefits to Explore Maximum Window Size

One of the use cases of the extended maximum window size is high volume data transfer over paths with long RTT delays and high bandwidth, called long fat pipes. The proposed extension improves and doubles at most the maximum throughput when bandwidth-latency product is greater than 1 GB. As propagation delay in an optical fiber is around 20 cm/ns, RTT will be over 100 milliseconds when the distance of the transmission is more than 10000km. This distance is not extraordinary for trans-pacific communications. In this case, the maximum throughput will be limited to 80 Gbps with the current

maximum window size, although network technologies for more than 100 Gbps are becoming common these days.

As the current TCP sequence number space is limited to 32 bits, it will not be possible to increase maximum window size any further. However, TCP may eventually have other extensions to increase sequence number space, for example, [RFC7323] and [RFC1263] mention about increasing sequence number space to 64 bits. We believe the information in this document will be useful when such extensions are proposed as they need to define new maximum window size.

6. Acknowledgments

The authors gratefully acknowledge significant inputs for this document from Richard Scheffenegger and Ilpo Jarvinen.

7. Security Considerations

It is known that an attacker can have more chances to insert forged packets into a TCP connection when large window size is used. This is not a specific problem of this proposal, but a generic problem to use larger window. Using PAWS can mitigate this problem, however, it is recommended to consult the Security Considerations section of RFC7323 [RFC7323] to check its implications.

8. IANA Considerations

If approved, this document overrides the definition of the WSO option defined in RFC7323 and so the IANA registry should be update accordingly (at least to add a pointer to this specification as reference for the WSO in the IANA registry).

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <http://www.rfc-editor.org/info/rfc793>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <http://www.rfc-editor.org/info/rfc7323>.

9.2. Informative References

[RFC1263] O'Malley, S. and L. Peterson, "TCP Extensions Considered Harmful", RFC 1263, DOI 10.17487/RFC1263, October 1991, <http://www.rfc-editor.org/info/rfc1263>.

Authors' Addresses

Yoshifumi Nishida
GE Global Research
2623 Camino Ramon
San Ramon, CA 94583
USA

Email: nishida@wide.ad.jp

Hirochika Asai
The University of Tokyo
7-3-1 Hongo
Bunkyo-ku, Tokyo 113-8656
JP

Email: panda@wide.ad.jp

Marcelo Bagnulo
UC3M

Email: marcelo@it.uc3m.es

INTERNET-DRAFT
Intended Status: Standard Track
Expires: May 4, 2017

M.Sun
HUAWEI Technologies

October 31, 2016

An Improvement of ECN to Enhance TCP Fairness Performance
draft-sun-tcpm-ecn-improvement-00

Abstract

This document describes TCP fairness performance enhancement scheme by use of two parameters congestion degree and link idle rate. It uses IP header flag reserved field and ECN 2 bits field to form a new IP congestion and Spare Flag (ICSF) and uses 3bits of reserved bits in the TCP header to compose the TCP Congestion and Spare Flag (TCSF) field. This method identifies the congestion and link idle status to make sure that every TCP flow can receive the same degree of fairness and can improve TCP send window adjustment speed and transmission efficiency.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

Copyright and License Notice

Table of Contents

1 Introduction

ECN (Explicit Congestion Notification)[RFC3168]is an extension to the Internet Protocol and to the Transmission Control Protocol and is defined in RFC 3168. ECN allows end-to-end notification of network congestion without dropping packets. ECN is an optional feature that may be used between two ECN-enabled endpoints when the underlying network infrastructure also supports it. An ECN-aware router may set a mark in the IP header instead of dropping a packet in order to signal impending congestion. The receiver of the packet echoes the congestion indication to the sender, which reduces its transmission rate as if it detected a dropped packet.

This draft specifies use of congestion degree and idle rate information in IP/TCP packets as an improvement of ECN. This scheme can help network equipments to obtain accurate congestion and idle status of the whole link in real time, and then feedback to the TCP transmitter. TCP transmitter can be more accurate to adjust the transmission window to avoid network congestion better. Packets always carry the worst port congestion or idle state and there will be no overlap of the case. In addition the scheme provides the same network status in each TCP flow and better TCP fairness.

This document requests one flag (1 bit) from IP header flag reserved field and 3bits of reserved bits in the TCP header. It describes use of extending options in IP and TCP.

1.1 Terminology

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

TCP: Transmission Control Protocol

RTT: Round Trip Time

CWND: Congestion Window

ACK: Acknowledgement

SSThresh:Slow Start Threshold

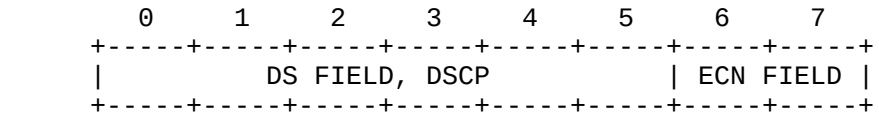
2. Generic ECN Overview

2.1 Detailed information of ECN

The traditional congestion control algorithm is usually designed on the basic assumption that network is like a black box. These algorithms use TCP detection to realize network congestion and delay.The send window at sending end increases gradually until congestion or packets loss occurs in network.

Although these congestion control algorithms can adapt to best effort transmission manner, they are not sensitive to packets loss or network delay, either cannot apply to the application of interactive class, including telnet,web browsing,audio/video etc.

ECN method helps routers announce TCP when network is about to get congested. Our method can not only avoid network packets loss circumstances, but also network delay due to router packet buffer overflow.



DSCP: differentiated services codepoint
ECN: Explicit Congestion Notification

Figure 1 ECN field in IP message

In Fig.1, we use 2 bit ECN FIELD in IP header to record whether ECN is enable and to represent congestion of network.

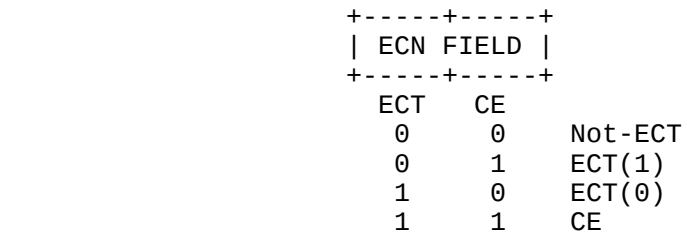


Figure 2 The ECN FIELD in IP

In Fig.2, ECN’s value is 00 which means not ECT (ECN-Capable Transport). 10 or 01 means ECT. While 11 means CE (Congestion Experienced).

There are two fields added in TCP header: CWR and ECE,as shown in Figure

3.

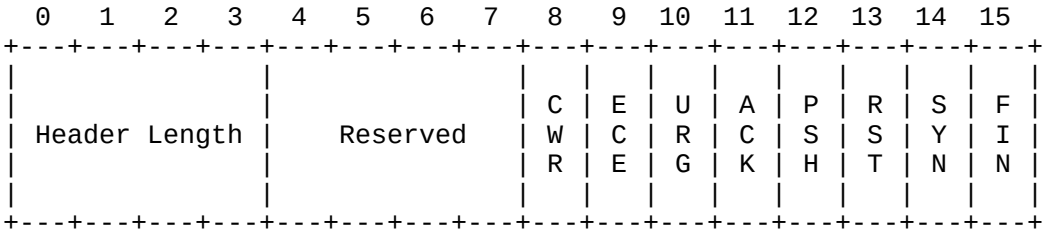


Figure 3 The new definition of bytes 13 and 14 of the TCP Header

CWR (Congestion Window Reduced): The sender will announce that send window has already reduced. The receiver will stop mark ECE in TCP ACK when it receives CWR message.

ECE (ECN Echo): If ECN FIELD’s value is 11 in messages received by TCP receiver, the network is congested. So receiver will send ECE(ECN Echo)in TCP ACK back to sender to notify congestion and messages will be marked as congestion Experienced CE. Besides the ECE can be used by TCP to negotiate whether both receiver and sender support ECN function.

The interaction process of ECN is described as following.

Step1: Using two fields of ECE and CWR in TCP SYN header to consult ECN. If both two ends set ECE and CWR in SYN message, these two ends support ECN.

Step2: If both ends support ECN, TCP sender will enable ECN function in IP ECN field and set IP ECN field 10 or 01.

Step3: When message arrives at network equipment, the equipment will check output port buffer situation. If buffer overflows the threshold of ECN and IP ECN filed is ECT, IP ECN filed will be modified to 11 and marked as CE.

Step4: Send messages out through output port.

Step5-7: are as same as steps 3-4.

Step8: When IP ECN field is marked as CE in IP messages, the receiver will add ECE mark in TCP ACK and notify sender that network has already congested.

Step9: According to the total amount of messages and marked ECE messages, the sender will confirm the congestion level and determine the size of send window.

2.2 Shortage

The main purpose of ECN is to notify the sender to reduce the window when the network device buffer achieves the threshold value as one method. But this congestion handle method lacks good solutions for light load in network. When light load happened, network is unable to notify TCP sender rapidly about idle rate information so it will be helpful if we can rapidly adjust send window to get faster transmission speed and better physical bandwidth utilization.

ECN can not accurately reflect network congestion status. Theoretically, when multiple network devices in the path exceed a threshold value of buffer, the congestion status should be the worst node status. But ECN signs may be changed by each network device, so the proportion of packets labeled with ECN is larger than that of the most congested node. That will lead network congestion status received by the sender is far greater than the actual congestion on the path.

TCP fairness problem. Because of the different stream order and the number of packets sent by RTT, it will lead to different congestion levels, which leads to the unfairness of the flow.

3 TCP Fairness Performance Improvement

3.1 Congestion Degree

The degree of link congestion is mainly refers to the number of packets in the link buffer cache relative to the physical bandwidth of the link, how long it takes to complete the transmission. For example, the link buffer cache size of message is 20MB, convert into 20*8Mb for the bit, the link bandwidth is 1Gbps. Then it takes 20*8Mb/1Gbps=160/1024=0.16 seconds. It takes 0.16 seconds to complete message transferred in the cache and the link congestion level can be 16%.

3.2 Idle Rate

Link idle rate is a concept which is opposite to link usage. For example, 1Gbps link with 600Mbps traffic, then the link usage is 60%, while the link idle rate is 40%.

3.3 Full Link Congestion and Idle Information

Step1: TCP sender sends message.

Step2: After receiving this message, the first network device can find out the link congestion degree/idle rate of its output port based on the forwarding table, and increase the link congestion information/idle rate into the message.

Step3: The message is sent out from the outlet port of the network device.

Step4: The following network equipment will execute look-up forwarding table and obtain the degree of congestion as Step2, and judge the congestion degree/idle rate of port and message before sending. If current congestion degree/idle rate of its port is worse than that in the message, update the congestion/idle information in the message, otherwise change noting.

Step5: Then the message is sent out from the outlet port of the network device.

Step6-7: are as same as Step 3-4.

Step8: TCP receiver adds the link congestion degree/idle rate of the message to the ACK, and the worst congestion degree/idle rate information of the link is notified to sender.

Step9: TCP sender determine the scope of the reduction/increase of the send window according to the received TCP ACK with the worst congestion degree/idle rate and current congestion window size.

3.4 Send Window Adjusting Method

3.4.1 Using the Worst Congestion Degree to Adjust Sending Window

The TCP sender adjusts the decrease rate of the window according to the worst congestion degree in the received TCP ACK and the current send window. Assuming that the worst congestion degree in TCP ACK is 10%, the current window of FLOW1 is 1000 and the window of FLOW2 is 200. Since the worst degree of congestion is 10%, which means that the current traffic exceeds 10% of the link bandwidth, the total traffic of all current flows is (1 + 10%) of the available bandwidth. Therefore, the ratio of the flow rate to be reduced is 10% / (1 + 10%) = 9.09%. For FLOW1, the current send window is 1000, so the window needs to be reduced by 90.9 (1000 * 9.09%), while FLOW2’s send window needs to be reduced by 18.2 (200 * 9.09%).

3.4.2 Using the Worst Idle Rate to Adjust Sending Window

The TCP sender adjusts the increase in window size based on the worst idle rate in the received TCP ACK and the current send window. Assuming that the worst idle rate in TCP ACK is 40%, the current window of FLOW1 is 1000 and the window of FLOW2 is 200. Since the worst idle rate is 40%, so the current link utilization is 60%. The flow rate of all current flow is 60%, and 40% of the free space, so the flow rate can be increased by 40% / (1-40%) = 66.67%.For FLOW1, the current send window

is 1000, so the window needs to increase 667 (1000 * 66.67%), while FLOW2's send window is 200, so the window needs to increase 133 (200 * 66.67%).

3.5 IP Option Extend

3.5.1 Congestion Degree IP Extend

The degree of congestion carried in IP packets can be achieved by extending the IP option.

Type	Length	Value
node congestion degree	4 bytes	0.1

The worst degree of congestion carried by TCP ACK can be extended by TCP option as following.

Type	Length	Value
the worst congestion degree	4 bytes	0.1

3.5.2 Idle Rate IP Extend

The idle rate carried in IP packets can be achieved by extending the IP option.

Type	Length	Value
node idle rate	4 bytes	0.45

The worst idle rate carried by TCP ACK can be extended by TCP option as following.

Type	Length	Value
the worst idle rate	4 bytes	0.45

4 References

4.1 Normative References

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

[RFC1925] Callon, R., "The Twelve Networking Truths", RFC 1925, April 1 1996.

4.2 Informative References

[RFC2481] Ramakrishnan, K. and S. Floyd, "A Proposal to add Explicit Congestion Notification (ECN) to IP", RFC 2481, January 1999.

[RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, June 2003.

[RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP’s SYN/ACK Packets", RFC 5562, June 2009.

Authors’ Addresses

Marcus Sun

HUAWEI TECHNOLOGIES CO.,LTD
12 E. Mozhou Rd. Jiangning Dist.
Nanjing,Jiangsu
China

EMail: marcus.sun@huawei.com

TCPM WG
Internet Draft
Intended status: Informational
Expires: April 2017

J. Touch
USC/ISI
M. Welzl
S. Islam
University of Oslo
J. You
Huawei
October 28, 2016

TCP Control Block Interdependence
draft-touch-tcpm-2140bis-01.txt

Status of this Memo

This Internet-Draft will expire on April 28, 2016.

Abstract

This memo describes interdependent TCP control blocks, where part of the TCP state is shared among similar concurrent or consecutive connections. TCP state includes a combination of parameters, such as connection state, current round-trip time estimates, congestion control information, and process information. Most of this state is maintained on a per-connection basis in the TCP Control Block (TCB), but implementations can (and do) share certain TCB information across connections to the same host. Such sharing is intended to improve overall transient transport performance, while maintaining backward-compatibility with existing implementations. The sharing described herein is limited to only the TCB initialization and so has no effect on the long-term behavior of TCP after a connection has been established.

Table of Contents

1. Introduction

TCP is a connection-oriented reliable transport protocol layered over IP [RFC793]. Each TCP connection maintains state, usually in a data structure called the TCP Control Block (TCB). The TCB contains information about the connection state, its associated local process, and feedback parameters about the connection's transmission properties. As originally specified and usually implemented, most TCB information is maintained on a per-connection basis. Some implementations can (and now do) share certain TCB information across connections to the same host.. Such sharing is intended to lead to better overall transient performance, especially for numerous short-lived and simultaneous connections, as often used in the World-Wide Web [Be94],[Br02].

This document discusses TCB state sharing that affects only the TCB initialization, and so has no effect on the long-term behavior of TCP after a connection has been established. Path information shared across SYN destination port numbers assumes that TCP segments having the same host-pair experience the same path properties, irrespective of TCP port numbers. The observations about TCB sharing in this document apply similarly to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

2. Conventions used in this document

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

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

3. Terminology

Host - a source or sink of TCP segments associated with a single IP address

Host-pair - a pair of hosts and their corresponding IP addresses

Path - an Internet path between the IP addresses of two hosts

4. The TCP Control Block (TCB)

A TCB describes the data associated with each connection, i.e., with each association of a pair of applications across the network. The TCB contains at least the following information [RFC793]:

```
Local process state
    pointers to send and receive buffers
    pointers to retransmission queue and current segment
    pointers to Internet Protocol (IP) PCB
Per-connection shared state
    macro-state
        connection state
        timers
        flags
        local and remote host numbers and ports
        TCP option state
    micro-state
        send and receive window state (size*, current number)
        round-trip time and variance
        cong. window size (snd_cwnd)*
        cong. window size threshold (ssthresh)*
        path maximum transmission unit (PMTU)*
        max window size seen*
        MSS#
        round-trip time and variance#
```

The per-connection information is shown as split into macro-state and micro-state, terminology borrowed from [Co91]. Macro-state describes the finite state machine; we include the endpoint numbers and components (timers, flags) used to help maintain that state. Macro-state describes the protocol for establishing and maintaining shared state about the connection. Micro-state describes the protocol after a connection has been established, to maintain the reliability and congestion control of the data transferred in the connection.

We further distinguish two other classes of shared micro-state that are associated more with host-pairs than with application pairs. One class is clearly host-pair dependent (#, e.g., MSS, RTT), and the other is host-pair dependent in its aggregate (*, e.g., congestion window information, current window sizes, etc.).

5. TCB Interdependence

There are two cases of TCB interdependence. Temporal sharing occurs when the TCB of an earlier (now CLOSED) connection to a host is used to initialize some parameters of a new connection to that same host, i.e., in sequence. Ensemble sharing occurs when a currently active connection to a host is used to initialize another (concurrent) connection to that host.

6. An Example of Temporal Sharing

The TCB data cache is accessed in two ways: it is read to initialize new TCBs and written when more current per-host state is available. New TCBs are initialized using context from past connections as follows:

TEMPORAL SHARING - TCB Initialization

Cached TCB	New TCB
old_PMTU	old_PMTU
old_MSS	old_MSS
old_RTT	old_RTT
old_RTTvar	old_RTTvar
old_option	(option specific)
old_ssthresh	old_ssthresh
old_snd_cwnd	old_snd_cwnd

Most cached TCB values are updated when a connection closes. Two exceptions are PMTU, which is updated after Path MTU Discovery [RFC4821], and MSS, which is updated whenever the MSS option is received in a TCP header.

Sharing MSS information affects only data in the SYN of the next connection, because MSS information is typically included in most TCP segments.

[TBD - complete this section with details for TFO and other options whose state may, must, or must not be shared] The way in which other TCP option state can be shared depends on the details of that option. E.g., TFO state includes the TCP Fast Open Cookie [RFC7413] or, in case TFO fails, a negative TCP Fast Open response (from [RFC 7413]: "The client MUST cache negative responses from the server in order to avoid potential connection failures. Negative responses include the server not acknowledging the data in the SYN, ICMP error messages, and (most importantly) no response (SYN-ACK) from the server at all, i.e., connection timeout."). TFOinfo is cached when a connection is established.

Other TCP option state might not be as readily cached. E.g., TCP-AO [RFC5925] success or failure between a host pair for a single SYN destination port might be usefully cached. TCP-AO success or failure to other SYN destination ports on that host pair is never useful to cache because TCP-AO security parameters can vary per service.

TEMPORAL SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_PMTU	curr_PMTU	PMTUD	current (cur)_PMTU
old_MSS	curr_MSS	MSSopt	cur_MSS
old_RTT	curr_RTT	CLOSE	merge(curr,old)
old_RTTvar	curr_RTTvar	CLOSE	merge(curr,old)
old_option	curr option	ESTAB	depends on option)
old_ssthresh	curr_ssthresh	CLOSE	merge(curr,old)
old_snd_cwnd	curr_snd_cwnd	CLOSE	merge(curr,old)

Caching PMTU and MSS is trivial; reported values are cached, and the most recent values are used. The cache is updated when the MSS option is received or after PMTUD (i.e., when an ICMPv4 Fraqmentation Needed [RFC1191] or ICMPv6 Packet Too Big message is received [RFC1981] or the equivalent is inferred, e.g. as from PLMTUD [RFC4821]), respectively, so the cache always has the most recent values from any connection. For MSS, the cache is consulted

only at connection establishment and not otherwise updated, which means that MSS options do not affect current connections. The default MSS is never saved; only reported MSS values update the cache, so an explicit override is required to reduce the MSS. Other options are copied or merged depending on the details of each option. E.g., TFO state is updated when a connection is established and read before establishing a new connection.

RTT values are updated by a more complicated mechanism [RFC1644][Ja86]. Dynamic RTT estimation requires a sequence of RTT measurements. As a result, the cached RTT (and its variance) is an average of its previous value with the contents of the currently active TCB for that host, when a TCB is closed. RTT values are updated only when a connection is closed. The method for merging old and current values needs to attempt to reduce the transient for new connections. [THESE MERGE FUNCTIONS NEED TO BE SPECIFIED, considering e.g. [DM16] - TBD].

The updates for RTT, RTTvar and ssthresh rely on existing information, i.e., old values. Should no such values exist, the current values are cached instead.

7. An Example of Ensemble Sharing

Sharing cached TCB data across concurrent connections requires attention to the aggregate nature of some of the shared state. For example, although MSS and RTT values can be shared by copying, it may not be appropriate to copy congestion window or ssthresh information (see section 8 for a discussion of congestion window or ssthresh sharing).

ENSEMBLE SHARING - TCB Initialization

Cached TCB	New TCB
old_PMTU	old_PMTU
old_MSS	old_MSS
old_RTT	old_RTT
old_RTTvar	old_RTTvar
old_option	(option-specific)

ENSEMBLE SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_PMTU	curr_PMTU	PMTUD/PLPMTUD	curr_PMTU
old_MSS	curr_MSS	MSSopt	curr_MSS
old_RTT	curr_RTT	update	rtt_update(old,cur)
old_RTTvar	curr_RTTvar	update	rtt_update(old,cur)
old_option	curr option	(depends)	(option specific)

For ensemble sharing, TCB information should be cached as early as possible, sometimes before a connection is closed. Otherwise, opening multiple concurrent connections may not result in TCB data sharing if no connection closes before others open. The amount of work involved in updating the aggregate average should be minimized, but the resulting value should be equivalent to having all values measured within a single connection. The function "rtt_update" in the ensemble sharing table indicates this operation, which occurs whenever the RTT would have been updated in the individual TCP connection. As a result, the cache contains the shared RTT variables, which no longer need to reside in the TCB [Ja86].

Congestion window size and ssthresh aggregation are more complicated in the concurrent case. When there is an ensemble of connections, we need to decide how that ensemble would have shared these variables, in order to derive initial values for new TCBs.

Any assumption of this sharing can be incorrect, including this one, because identical endpoint address pairs may not share network paths. In current implementations, new congestion windows are set at an initial value of 4-10 segments [RFC3390][RFC6928], so that the sum of the current windows is increased for any new connection. This can have detrimental consequences where several connections share a highly congested link.

There are several ways to initialize the congestion window in a new TCB among an ensemble of current connections to a host, as shown below. Current TCP implementations initialize it to four segments as standard [rfc3390] and 10 segments experimentally [RFC6928] and T/TCP hinted that it should be initialized to the old window size [RFC1644]. In the former cases, the assumption is that new connections should behave as conservatively as possible. In the

latter T/TCP case, no accommodation is made for concurrent aggregate behavior.

In either case, the sum of window sizes can increase, rather than remain constant. A different approach is to give each pending connection its "fair share" of the available congestion window, and let the connections balance from there. The assumption we make here is that new connections are implicit requests for an equal share of available link bandwidth, which should be granted at the expense of current connections. [TBD - a new method for safe congestion sharing will be described]

8. Compatibility Issues

For the congestion and current window information, the initial values computed by TCB interdependence may not be consistent with the long-term aggregate behavior of a set of concurrent connections between the same endpoints. Under conventional TCP congestion control, if a single existing connection has converged to a congestion window of 40 segments, two newly joining concurrent connections assume initial windows of 10 segments [RFC6928], and the current connection’s window doesn’t decrease to accommodate this additional load and connections can mutually interfere. One example of this is seen on low-bandwidth, high-delay links, where concurrent connections supporting Web traffic can collide because their initial windows were too large, even when set at one segment.

[TBD - this paragraph needs to be revised based on new recommendations] Under TCB interdependence, all three connections could change to use a congestion window of 12 (rounded down to an even number from 13.33, i.e., 40/3). This would include both increasing the initial window of the new connections (vs. current recommendations [RFC6928]) and decreasing the congestion window of the current connection (from 40 down to 12). This gives the new connections a larger initial window than allowed by [RFC6928], but maintains the aggregate. Depending on whether the previous connections were in steady-state, this can result in more bursty behavior, e.g., when previous connections are idle and new connections commence with a large amount of available data to transmit. Additionally, reducing the congestion window of an existing connection needs to account for the number of packets that are already in flight.

Because this proposal attempts to anticipate the aggregate steady-state values of TCB state among a group or over time, it should avoid the transient effects of new connections. In addition, because it considers the ensemble and temporal properties of those

aggregates, it should also prevent the transients of short-lived or multiple concurrent connections from adversely affecting the overall network performance. There have been ongoing analysis and experiments to validate these assumptions. For example, [Ph12] recommends to only cache ssthresh for temporal sharing when flows are long. Sharing ssthresh between short flows can deteriorate the overall performance of individual connections[Ph12, Nd16], although this may benefit overall network performance. [TBD - the details of this issue need to be summarized and clarified herein].

[TBD - placeholder for corresponding RTT discussion]

Due to mechanisms like ECMP and LAG [RFC7424], TCP connections sharing the same host-pair may not always share the same path. This does not matter for host-specific information such as RWIN and TCP option state, such as TFOinfo. When TCB information is shared across different SYN destination ports, path-related information can be incorrect; however, the impact of this error is potentially diminished if (as discussed here) TCB sharing affects only the transient event of a connection start or if TCB information is shared only within connections to the same SYN destination port. In case of Temporal Sharing, TCB information could also become invalid over time. Because this is similar to the case when a connection becomes idle, mechanisms that address idle TCP connections (e.g., [RFC7661]) could also be applied to TCB cache management.

There may be additional considerations to the way in which TCB interdependence rebalances congestion feedback among the current connections, e.g., it may be appropriate to consider the impact of a connection being in Fast Recovery [RFC5861] or some other similar unusual feedback state, e.g., as inhibiting or affecting the calculations described herein.

TCP is sometimes used in situations where packets of the same host-pair always take the same path. Because ECMP and LAG examine TCP port numbers, they may not be supported when TCP segments are encapsulated, encrypted, or altered - for example, some Virtual Private Networks (VPNs) are known to use proprietary UDP encapsulation methods. Similarly, they cannot operate when the TCP header is encrypted, e.g., when using IPsec ESP. TCB interdependence among the entire set sharing the same endpoint IP addresses should work without problems under these circumstances. Moreover, measures to increase the probability that connections use the same path could be applied: e.g., the connections could be given the same IPv6 flow label. TCB interdependence can also be extended to sets of host IP address pairs that share the same network path conditions, such as when a group of addresses is on the same LAN (see Section 9).

9. Implications

There are several implications to incorporating TCB interdependence in TCP implementations. First, it may reduce the need for application-layer multiplexing for performance enhancement [RFC7231]. Protocols like HTTP/2 [RFC7540] avoid connection reestablishment costs by serializing or multiplexing a set of per-host connections across a single TCP connection. This avoids TCP's per-connection OPEN handshake and also avoids recomputing MSS, RTT, and congestion windows. By avoiding the so-called, "slow-start restart," performance can be optimized. TCB interdependece can provide the "slow-start restart avoidance" of multiplexing, without requiring a multiplexing mechanism at the application layer.

TCB interdependence pushes some of the TCP implementation from the traditional transport layer (in the ISO model), to the network layer. This acknowledges that some state is in fact per-host-pair or can be per-path as indicated solely by that host-pair. Transport protocols typically manage per-application-pair associations (per stream), and network protocols manage per-host-pair and path associations (routing). Round-trip time, MSS, and congestion information could be more appropriately handled in a network-layer fashion, aggregated among concurrent connections, and shared across connection instances [RFC3124].

An earlier version of RTT sharing suggested implementing RTT state at the IP layer, rather than at the TCP layer [Ja86]. Our observations are for sharing state among TCP connections, which avoids some of the difficulties in an IP-layer solution. One such problem is determining the associated prior outgoing packet for an incoming packet, to infer RTT from the exchange. Because RTTs are still determined inside the TCP layer, this is simpler than at the IP layer. This is a case where information should be computed at the transport layer, but could be shared at the network layer.

Per-host-pair associations are not the limit of these techniques. It is possible that TCBs could be similarly shared between hosts on a subnet or within a cluster, because the predominant path can be subnet-subnet, rather than host-host. Additionally, TCB interdependence can be applied to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

There may be other information that can be shared between concurrent connections. For example, knowing that another connection has just tried to expand its window size and failed, a connection may not attempt to do the same for some period. The idea is that existing

TCP implementations infer the behavior of all competing connections, including those within the same host or subnet. One possible optimization is to make that implicit feedback explicit, via extended information associated with the endpoint IP address and its TCP implementation, rather than per-connection state in the TCB.

Like its initial version in 1997, this document’s approach to TCB interdependence focuses on sharing a set of TCBs by updating the TCB state to reduce the impact of transients when connections begin or end. Other mechanisms have since been proposed to continuously share information between all ongoing communication (including connectionless protocols), updating the congestion state during any congestion-related event (e.g., timeout, loss confirmation, etc.) [RFC3124]. By dealing exclusively with transients, TCB interdependence is more likely to exhibit the same behavior as unmodified, independent TCP connections.

10. Implementation Observations

The observation that some TCB state is host-pair specific rather than application-pair dependent is not new and is a common engineering decision in layered protocol implementations. A discussion of sharing RTT information among protocols layered over IP, including UDP and TCP, occurred in [Ja86]. Although now deprecated, T/TCP was the first to propose using caches in order to maintain TCB states (see Appendix A for more information).

The table below describes the current implementation status for some TCB information in Linux kernel version 4.6, FreeBSD 10 and Windows (as of October 2016).

TCB data	Status
old_MSS	Cached and shared in Linux
old_RTT	Cached and shared in FreeBSD
old_RTTvar	Cached and shared in FreeBSD
old PMTU	Cached and shared in FreeBSD and Windows
old TFOinfo	Cached and shared in Linux and Windows
old_snd_cwnd	Not shared
old_ssthresh	Cached and shared in FreeBSD and Linux: FreeBSD: arithmetic mean of ssthresh and previous value if a previous value exists; Linux: depending on state, max(cwnd/2, ssthresh) in most cases

11. Security Considerations

These suggested implementation enhancements do not have additional ramifications for explicit attacks. These enhancements may be susceptible to denial-of-service attacks if not otherwise secured. For example, an application can open a connection and set its window size to zero, denying service to any other subsequent connection between those hosts.

TCB sharing may be susceptible to denial-of-service attacks, wherever the TCB is shared, between connections in a single host, or between hosts if TCB sharing is implemented within a subnet (see Implications section). Some shared TCB parameters are used only to create new TCBs, others are shared among the TCBs of ongoing connections. New connections can join the ongoing set, e.g., to optimize send window size among a set of connections to the same host.

Attacks on parameters used only for initialization affect only the transient performance of a TCP connection. For short connections, the performance ramification can approach that of a denial-of-service attack. E.g., if an application changes its TCB to have a

false and small window size, subsequent connections would experience performance degradation until their window grew appropriately.

The solution is to limit the effect of compromised TCB values. TCBs are compromised when they are modified directly by an application or transmitted between hosts via unauthenticated means (e.g., by using a dirty flag). TCBs that are not compromised by application modification do not have any unique security ramifications. Note that the proposed parameters for TCB sharing are not currently modifiable by an application.

All shared TCBs MUST be validated against default minimum parameters before used for new connections. This validation would not impact performance, because it occurs only at TCB initialization. This limits the effect of attacks on new connections to reducing the benefit of TCB sharing, resulting in the current default TCP performance. For ongoing connections, the effect of incoming packets on shared information should be both limited and validated against constraints before use. This is a beneficial precaution for existing TCP implementations as well.

TCBs modified by an application SHOULD NOT be shared, unless the new connection sharing the compromised information has been given explicit permission to use such information by the connection API. No mechanism for that indication currently exists, but it could be supported by an augmented API. This sharing restriction SHOULD be implemented in both the host and the subnet. Sharing on a subnet SHOULD utilize authentication to prevent undetected tampering of shared TCB parameters. These restrictions limit the security impact of modified TCBs both for connection initialization and for ongoing connections.

Finally, shared values MUST be limited to performance factors only. Other information, such as TCP sequence numbers, when shared, are already known to compromise security.

12. IANA Considerations

There are no IANA implications or requests in this document.

This section should be removed upon final publication as an RFC.

13. References

13.1. Normative References

[RFC793] Postel, Jon, "Transmission Control Protocol," Network Working Group RFC-793/STD-7, ISI, Sept. 1981.

[RFC1191] Mogul, J., Deering, S., "Path MTU Discovery," RFC 1191, Nov. 1990.

[RFC1981] McCann, J., Deering. S., Mogul, J., "Path MTU Discovery for IP version 6," RFC 1981, Aug. 1996.

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

[RFC4821] Mathis, M., Heffner, J., "Packetization Layer Path MTU Discovery," RFC 4821, Mar. 2007.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., Jain, A., "TCP Fast Open", RFC 7413, Dec. 2014.

13.2. Informative References

[Br02] Brownlee, N. and K. Claffy, "Understanding Internet Traffic Streams: Dragonflies and Tortoises", IEEE Communications Magazine p110-117, 2002.

[Be94] Berners-Lee, T., et al., "The World-Wide Web," Communications of the ACM, V37, Aug. 1994, pp. 76-82.

[Br94] Braden, B., "T/TCP -- Transaction TCP: Source Changes for Sun OS 4.1.3,", Release 1.0, USC/ISI, September 14, 1994.

[Co91] Comer, D., Stevens, D., Internetworking with TCP/IP, V2, Prentice-Hall, NJ, 1991.

[FreeBSD] FreeBSD source code, Release 2.10, http://www.freebsd.org/

[Ja86] Jacobson, V., (mail to public list "tcp-ip", no archive found), 1986.

[Nd16] Dukkipati, N., Yuchung C., and Amin V., "Research Impacting the Practice of Congestion Control." ACM SIGCOMM CCR (editorial).

[DM16] Matz, D., "Optimize TCP's Minimum Retransmission Timeout for Low Latency Environments", Master's thesis, Technical University Munich, 2016.

[Ph12] Hurtig, P., Brunstrom, A., "Enhanced metric caching for short TCP flows," 2012 IEEE International Conference on Communications (ICC), Ottawa, ON, 2012, pp. 1209-1213.

[RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification," RFC-1644, July 1994.

[RFC1379] Braden, R., "Transaction TCP -- Concepts," RFC-1379, September 1992.

[RFC3390] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window," RFC 3390, Oct. 2002.

[RFC7231] Fielding, R., J. Reshke, Eds., "HTTP/1.1 Semantics and Content," RFC-7231, June 2014.

[RFC3124] Balakrishnan, H., Seshan, S., "The Congestion Manager," RFC 3124, June 2001.

[RFC4340] Kohler, E., Handley, M., Floyd, S., "Datagram Congestion Control Protocol (DCCP)," RFC 4340, Mar. 2006.

[RFC4960] Stewart, R., (Ed.), "Stream Control Transmission Protocol," RFC4960, Sept. 2007.

[RFC5861] Allman, M., Paxson, V., Blanton, E., "TCP Congestion Control," RFC 5861, Sept. 2009.

[RFC5925] Touch, J., Mankin, A., Bonica, R., "The TCP Authentication Option," RFC 5925, June 2010.

[RFC6824] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Jan. 2013.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., Mathis, M., "Increasing TCP's Initial Window," RFC 6928, Apr. 2013.

[RFC7424] Krishnan, R., Yong, L., Ghanwani, A., So, N., Khasnabish, B., "Mechanisms for Optimizing Link Aggregation Group (LAG) and Equal-Cost Multipath (ECMP) Component Link Utilization in Networks", RFC 7424, Jan. 2015

[RFC7540] Belshe, M., Peon, R., Thomson, M., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, May 2015.

[RFC7661] Fairhurst, G., Sathiaseelan, A., Secchi, R., "Updating TCP to Support Rate-Limited Traffic", RFC 7661, Oct. 2015

14. Acknowledgments

The authors would like to thank for Praveen Balasubramanian for information regarding TCB sharing in Windows, and Yuchung Cheng and Michael Scharf for comments on earlier versions of the draft. This work has received funding from a collaborative research project between the University of Oslo and Huawei Technologies Co., Ltd., and is partly supported by USC/ISI's Postel Center.

This document was prepared using 2-Word-v2.0.template.dot.

Authors' Addresses

Joe Touch
USC/ISI
4676 Admiralty Way
Marina del Rey, CA 90292-6695
USA

Phone: +1 (310) 448-9151
Email: touch@isi.edu

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no

Jianjie You
Huawei
101 Software Avenue, Yuhua District
Nanjing 210012
China

Email: youjianjie@huawei.com

15. Appendix A: TCB sharing history

T/TCP proposed using caches to maintain TCB information across instances (temporal sharing), e.g., smoothed RTT, RTT variance, congestion avoidance threshold, and MSS [RFC1644]. These values were in addition to connection counts used by T/TCP to accelerate data delivery prior to the full three-way handshake during an OPEN. The goal was to aggregate TCB components where they reflect one association - that of the host-pair, rather than artificially separating those components by connection.

At least one T/TCP implementation saved the MSS and aggregated the RTT parameters across multiple connections, but omitted caching the congestion window information [Br94], as originally specified in [RFC1379]. Some T/TCP implementations immediately updated MSS when the TCP MSS header option was received [Br94], although this was not addressed specifically in the concepts or functional specification [RFC1379][RFC1644]. In later T/TCP implementations, RTT values were updated only after a CLOSE, which does not benefit concurrent sessions.

Temporal sharing of cached TCB data was originally implemented in the SunOS 4.1.3 T/TCP extensions [Br94] and the FreeBSD port of same [FreeBSD]. As mentioned before, only the MSS and RTT parameters were cached, as originally specified in [RFC1379]. Later discussion of T/TCP suggested including congestion control parameters in this cache [RFC1644].

TCPM WG
Internet Draft
Intended status: Informational
Obsoletes: 2140
Expires: July 2019

J. Touch
Independent Consultant
M. Welzl
S. Islam
University of Oslo
January 4, 2019

TCP Control Block Interdependence
draft-touch-tcpm-2140bis-06.txt

Status of this Memo

Abstract

This memo updates and replaces RFC 2140's description of interdependent TCP control blocks, in which part of the TCP state is shared among similar concurrent or consecutive connections. TCP state includes a combination of parameters, such as connection state, current round-trip time estimates, congestion control information, and process information. Most of this state is maintained on a per-connection basis in the TCP Control Block (TCB), but implementations can (and do) share certain TCB information across connections to the same host. Such sharing is intended to improve overall transient transport performance, while maintaining backward-compatibility with existing implementations. The sharing described herein is limited to only the TCB initialization and so has no effect on the long-term behavior of TCP after a connection has been established.

Table of Contents

1. Introduction

TCP is a connection-oriented reliable transport protocol layered over IP [RFC793]. Each TCP connection maintains state, usually in a data structure called the TCP Control Block (TCB). The TCB contains information about the connection state, its associated local process, and feedback parameters about the connection's transmission properties. As originally specified and usually implemented, most TCB information is maintained on a per-connection basis. Some implementations can (and now do) share certain TCB information across connections to the same host [RFC2140]. Such sharing is intended to lead to better overall transient performance, especially for numerous short-lived and simultaneous connections, as often used in the World-Wide Web [Be94],[Br02].

This document updates RFC 2140's discussion of TCB state sharing and provides a complete replacement for that document. This state sharing affects only TCB initialization [RFC2140] and thus has no effect on the long-term behavior of TCP after a connection has been established. Path information shared across SYN destination port numbers assumes that TCP segments having the same host-pair experience the same path properties, irrespective of TCP port numbers. The observations about TCB sharing in this document apply similarly to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

2. Conventions used in this document

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

In this document, these words will appear with that interpretation only when in ALL CAPS. Lower case uses of these words are not to be interpreted as carrying significance described in RFC 2119.

In this document, the characters ">>" preceding an indented line(s) indicates a statement using the key words listed above. This convention aids reviewers in quickly identifying or finding the portions of this RFC covered by these keywords.

3. Terminology

Host - a source or sink of TCP segments associated with a single IP address

Host-pair - a pair of hosts and their corresponding IP addresses

Path - an Internet path between the IP addresses of two hosts

4. The TCP Control Block (TCB)

A TCB describes the data associated with each connection, i.e., with each association of a pair of applications across the network. The TCB contains at least the following information [RFC793]:

```
Local process state
    pointers to send and receive buffers
    pointers to retransmission queue and current segment
    pointers to Internet Protocol (IP) PCB
Per-connection shared state
    macro-state
        connection state
        timers
        flags
        local and remote host numbers and ports
        TCP option state
    micro-state
        send and receive window state (size*, current number)
        round-trip time and variance
        cong. window size (snd_cwnd)*
        cong. window size threshold (ssthresh)*
        max window size seen*
        sendMSS#
        MMS_S#
        MMS_R#
        PMTU#
        round-trip time and variance#
```

The per-connection information is shown as split into macro-state and micro-state, terminology borrowed from [Co91]. Macro-state describes the protocol for establishing the initial shared state about the connection; we include the endpoint numbers and components (timers, flags) required upon commencement that are later used to help maintain that state. Micro-state describes the protocol after a connection has been established, to maintain the reliability and congestion control of the data transferred in the connection.

We further distinguish two other classes of shared micro-state that are associated more with host-pairs than with application pairs. One class is clearly host-pair dependent (#, e.g., MSS, MMS, PMTU, RTT), and the other is host-pair dependent in its aggregate (*, e.g., congestion window information, current window sizes, etc.).

5. TCB Interdependence

There are two cases of TCB interdependence. Temporal sharing occurs when the TCB of an earlier (now CLOSED) connection to a host is used to initialize some parameters of a new connection to that same host, i.e., in sequence. Ensemble sharing occurs when a currently active connection to a host is used to initialize another (concurrent) connection to that host.

6. An Example of Temporal Sharing

The TCB data cache is accessed in two ways: it is read to initialize new TCBs and written when more current per-host state is available. New TCBs can be initialized using context from past connections as follows:

TEMPORAL SHARING - TCB Initialization

Cached TCB	New TCB
old_MMS_S	old_MMS_S or not cached
old_MMS_R	old_MMS_R or not cached
old_sendMSS	old_sendMSS
old_PMTU	old_PMTU
old_RTT	old_RTT
old_RTTvar	old_RTTvar
old_option	(option specific)
old_ssthresh	old_ssthresh
old_snd_cwnd	old_snd_cwnd

Sections 8 and 9 discuss compatibility issues and implications of sharing the specific information listed above. Section 10 gives an overview of known implementations.

Most cached TCB values are updated when a connection closes. The exceptions are MMS_R and MMS_S, which are reported by IP [RFC1122], PMTU which is updated after Path MTU Discovery [RFC1191][RFC4821][RFC8201], and sendMSS, which is updated if the MSS option is received in the TCP SYN header.

Sharing sendMSS information affects only data in the SYN of the next connection, because sendMSS information is typically included in most TCP SYN segments. Caching PMTU can accelerate the efficiency of PMTUD, but can also result in black-holing until corrected if in error. Caching MMS_R and MMS_S may be of little direct value as they are reported by the local IP stack anyway.

The way in which other TCP option state can be shared depends on the details of that option. E.g., TFO state includes the TCP Fast Open Cookie [RFC7413] or, in case TFO fails, a negative TCP Fast Open response. RFC 7413 states, "The client MUST cache negative responses from the server in order to avoid potential connection failures. Negative responses include the server not acknowledging the data in the SYN, ICMP error messages, and (most importantly) no response (SYN-ACK) from the server at all, i.e., connection timeout." [RFC 7413]. TFOinfo is cached when a connection is established.

Other TCP option state might not be as readily cached. E.g., TCP-AO [RFC5925] success or failure between a host pair for a single SYN destination port might be usefully cached. TCP-AO success or failure to other SYN destination ports on that host pair is never useful to cache because TCP-AO security parameters can vary per service.

The table below gives an overview of option-specific information that can be shared.

TEMPORAL SHARING - Option info

Cached	New
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

TEMPORAL SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_MMS_S	curr_ MMS_S	OPEN	curr MMS_S
old_MMS_R	curr_ MMS_R	OPEN	curr_MMS_R
old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
old_PMTU	curr_PMTU	PMTUD	curr_PMTU
old_RTT	curr_RTT	CLOSE	merge(curr,old)
old_RTTvar	curr_RTTvar	CLOSE	merge(curr,old)
old_option	curr option	ESTAB	(depends on option)
old_ssthresh	curr_ssthresh	CLOSE	merge(curr,old)
old_snd_cwnd	curr_snd_cwnd	CLOSE	merge(curr,old)

Caching PMTU and sendMSS is trivial; reported values are cached, and the most recent values are used. The cache is updated when the MSS option is received in a SYN or after PMTUD (i.e., when an ICMPv4 Fraqmentation Needed [RFC1191] or ICMPv6 Packet Too Big message is received [RFC8201] or the equivalent is inferred, e.g. as from PLPMTUD [RFC4821]), respectively, so the cache always has the most recent values from any connection. For sendMSS, the cache is consulted only at connection establishment and not otherwise updated, which means that MSS options do not affect current connections. The default sendMSS is never saved; only reported MSS values update the cache, so an explicit override is required to reduce the sendMSS. There is no particular benefit to caching MMS_S and MMS R as these are reported by the local IP stack.

TCP options are copied or merged depending on the details of each option, where "merge" is some function that combines the values of "curr" and "old". E.g., TFO state is updated when a connection is established and read before establishing a new connection.

RTT values are updated by a more complicated mechanism [RFC1644][Ja86]. Dynamic RTT estimation requires a sequence of RTT measurements. As a result, the cached RTT (and its variance) is an average of its previous value with the contents of the currently active TCB for that host, when a TCB is closed. RTT values are updated only when a connection is closed. The method for merging old

and current values needs to attempt to reduce the transient for new connections.

The updates for RTT, RTTvar and ssthresh rely on existing information, i.e., old values. Should no such values exist, the current values are cached instead.

TEMPORAL SHARING - Option info Updates

Cached	Current	when?	New Cached
old_TFO_Cookie	old_TFO_Cookie	ESTAB	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure	ESTAB	old_TFO_Failure

7. An Example of Ensemble Sharing

Sharing cached TCB data across concurrent connections requires attention to the aggregate nature of some of the shared state. For example, although MSS and RTT values can be shared by copying, it may not be appropriate to simply copy congestion window or ssthresh information; instead, the new values can be a function (f) of the cumulative values and the number of connections (N).

ENSEMBLE SHARING - TCB Initialization

Cached TCB	New TCB
old_MMS_S	old_MMS_S
old_MMS_R	old_MMS_R
old_sendMSS	old_sendMSS
old_PMTU	old_PMTU
old_RTT	old_RTT
old_RTTvar	old_RTTvar
old ssthresh sum	f(old ssthresh sum, N)
old snd_cwnd sum	f(old snd cwnd sum, N)
old_option	(option-specific)

Sections 8 and 9 discuss compatibility issues and implications of sharing the specific information listed above.

The table below gives an overview of option-specific information that can be shared.

ENSEMBLE SHARING Option info

Cached	New
old_TFO_Cookie	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure

ENSEMBLE SHARING - Cache Updates

Cached TCB	Current TCB	when?	New Cached TCB
old_MMS_S	curr_MMS_S	OPEN	curr_MMS_S
old_MMS_R	curr_MMS_R	OPEN	curr_MMS_R
old_sendMSS	curr_sendMSS	MSSopt	curr_sendMSS
old_PMTU	curr_PMTU	PMTUD /PLPMTUD	curr_PMTU
old_RTT	curr_RTT	update	rtt_update(old,curr)
old_RTTvar	curr_RTTvar	update	rtt_update(old,curr)
old ssthresh	curr ssthresh	update	adjust sum as appopriate
old snd_cwnd	curr snd_cwnd	update	adjust sum as appopriate
old_option	curr option	(depends)	(option specific)

For ensemble sharing, TCB information should be cached as early as possible, sometimes before a connection is closed. Otherwise, opening multiple concurrent connections may not result in TCB data sharing if no connection closes before others open. The amount of work involved in updating the aggregate average should be minimized, but the resulting value should be equivalent to having all values measured within a single connection. The function "rtt_update" in

the ensemble sharing table indicates this operation, which occurs whenever the RTT would have been updated in the individual TCP connection. As a result, the cache contains the shared RTT variables, which no longer need to reside in the TCB [Ja86].

Congestion window size and ssthresh aggregation are more complicated in the concurrent case. When there is an ensemble of connections, we need to decide how that ensemble would have shared these variables, in order to derive initial values for new TCBs.

ENSEMBLE SHARING - Option info Updates

Cached	Current	when?	New Cached
old_TFO_Cookie	old_TFO_Cookie	ESTAB	old_TFO_Cookie
old_TFO_Failure	old_TFO_Failure	ESTAB	old_TFO_Failure

Any assumption of this sharing can be incorrect because identical endpoint address pairs may not share network paths. In current implementations, new congestion windows are set at an initial value of 4-10 segments [RFC3390][RFC6928], so that the sum of the current windows is increased for any new connection. This can have detrimental consequences where several connections share a highly congested link.

There are several ways to initialize the congestion window in a new TCB among an ensemble of current connections to a host. Current TCP implementations initialize it to four segments as standard [rfc3390] and 10 segments experimentally [RFC6928] and T/TCP hinted that it should be initialized to the old window size [RFC1644]. In the former cases, the assumption is that new connections should behave as conservatively as possible. In the latter T/TCP case, no accommodation is made for concurrent aggregate behavior. The algorithm described in [Ba12] adjusts the initial cwnd depending on the cwnd values of ongoing connections.

8. Compatibility Issues

For the congestion and current window information, the initial values computed by TCB interdependence may not be consistent with the long-term aggregate behavior of a set of concurrent connections between the same endpoints. Under conventional TCP congestion control, if a single existing connection has converged to a congestion window of 40 segments, two newly joining concurrent connections assume initial windows of 10 segments [RFC6928], and the

current connection's window doesn't decrease to accommodate this additional load and connections can mutually interfere. One example of this is seen on low-bandwidth, high-delay links, where concurrent connections supporting Web traffic can collide because their initial windows were too large, even when set at one segment.

The authors of [Hu12] recommend caching ssthresh for temporal sharing only when flows are long. Some studies suggest that sharing ssthresh between short flows can deteriorate the performance of individual connections [Hu12, Du16], although this may benefit aggregate network performance.

Due to mechanisms like ECMP and LAG [RFC7424], TCP connections sharing the same host-pair may not always share the same path. This does not matter for host-specific information such as RWIN and TCP option state, such as TFOinfo. When TCB information is shared across different SYN destination ports, path-related information can be incorrect; however, the impact of this error is potentially diminished if (as discussed here) TCB sharing affects only the transient event of a connection start or if TCB information is shared only within connections to the same SYN destination port. In case of Temporal Sharing, TCB information could also become invalid over time. Because this is similar to the case when a connection becomes idle, mechanisms that address idle TCP connections (e.g., [RFC7661]) could also be applied to TCB cache management, especially when TCP Fast Open is used [RFC7413].

There may be additional considerations to the way in which TCB interdependence rebalances congestion feedback among the current connections, e.g., it may be appropriate to consider the impact of a connection being in Fast Recovery [RFC5861] or some other similar unusual feedback state, e.g., as inhibiting or affecting the calculations described herein.

TCP is sometimes used in situations where packets of the same host-pair always take the same path. Because ECMP and LAG examine TCP port numbers, they may not be supported when TCP segments are encapsulated, encrypted, or altered - for example, some Virtual Private Networks (VPNs) are known to use proprietary UDP encapsulation methods. Similarly, they cannot operate when the TCP header is encrypted, e.g., when using IPsec ESP. TCB interdependence among the entire set sharing the same endpoint IP addresses should work without problems under these circumstances. Moreover, measures to increase the probability that connections use the same path could be applied: e.g., the connections could be given the same IPv6 flow label. TCB interdependence can also be extended to sets of host IP

address pairs that share the same network path conditions, such as when a group of addresses is on the same LAN (see Section 9).

It can be wrong to share TCB information between TCP connections on the same host as identified by the IP address if an IP address is assigned to a new host (e.g., IP address spinning, as is used by ISPs to inhibit running servers). It can be wrong if Network Address (and Port) Translation (NA(P)T) [RFC2663] or any other IP sharing mechanism is used. Such mechanisms are less likely to be used with IPv6. Other methods to identify a host could also be considered to make correct TCB sharing more likely. Moreover, some TCB information is about dominant path properties rather than the specific host. IP addresses may differ, yet the relevant part of the path may be the same.

9. Implications

There are several implications to incorporating TCB interdependence in TCP implementations. First, it may reduce the need for application-layer multiplexing for performance enhancement [RFC7231]. Protocols like HTTP/2 [RFC7540] avoid connection reestablishment costs by serializing or multiplexing a set of per-host connections across a single TCP connection. This avoids TCP's per-connection OPEN handshake and also avoids recomputing the MSS, RTT, and congestion window values. By avoiding the so-called, "slow-start restart," performance can be optimized [Hu01]. TCB interdependence can provide the "slow-start restart avoidance" of multiplexing, without requiring a multiplexing mechanism at the application layer.

TCB interdependence pushes some of the TCP implementation from the traditional transport layer (in the ISO model), to the network layer. This acknowledges that some state is in fact per-host-pair or can be per-path as indicated solely by that host-pair. Transport protocols typically manage per-application-pair associations (per stream), and network protocols manage per-host-pair and path associations (routing). Round-trip time, MSS, and congestion information could be more appropriately handled in a network-layer fashion, aggregated among concurrent connections, and shared across connection instances [RFC3124].

An earlier version of RTT sharing suggested implementing RTT state at the IP layer, rather than at the TCP layer [Ja86]. Our observations are for sharing state among TCP connections, which avoids some of the difficulties in an IP-layer solution. One such problem is determining the associated prior outgoing packet for an incoming packet, to infer RTT from the exchange. Because RTTs are

still determined inside the TCP layer, this is simpler than at the IP layer. This is a case where information should be computed at the transport layer, but could be shared at the network layer.

Per-host-pair associations are not the limit of these techniques. It is possible that TCBs could be similarly shared between hosts on a subnet or within a cluster, because the predominant path can be subnet-subnet, rather than host-host. Additionally, TCB interdependence can be applied to any protocol with congestion state, including SCTP [RFC4960] and DCCP [RFC4340], as well as for individual subflows in Multipath TCP [RFC6824].

There may be other information that can be shared between concurrent connections. For example, knowing that another connection has just tried to expand its window size and failed, a connection may not attempt to do the same for some period. The idea is that existing TCP implementations infer the behavior of all competing connections, including those within the same host or subnet. One possible optimization is to make that implicit feedback explicit, via extended information associated with the endpoint IP address and its TCP implementation, rather than per-connection state in the TCB.

Like the initial version of this document [RFC2140], this update's approach to TCB interdependence focuses on sharing a set of TCBs by updating the TCB state to reduce the impact of transients when connections begin or end. Other mechanisms have since been proposed to continuously share information between all ongoing communication (including connectionless protocols), updating the congestion state during any congestion-related event (e.g., timeout, loss confirmation, etc.) [RFC3124]. By dealing exclusively with transients, TCB interdependence is more likely to exhibit the same behavior as unmodified, independent TCP connections.

10. Implementation Observations

The observation that some TCB state is host-pair specific rather than application-pair dependent is not new and is a common engineering decision in layered protocol implementations. A discussion of sharing RTT information among protocols layered over IP, including UDP and TCP, occurred in [Ja86]. Although now deprecated, T/TCP was the first to propose using caches in order to maintain TCB states (see Appendix A for more information).

The table below describes the current implementation status for some TCB information in Linux kernel version 4.6, FreeBSD 10 and Windows (as of October 2016). In the table, "shared" only refers to temporal sharing.

TCB data	Status
old MMS_S	Not shared
old MMS_R	Not shared
old_sendMSS	Cached and shared in Linux (MSS)
old PMTU	Cached and shared in FreeBSD and Windows (PMTU)
old_RTT	Cached and shared in FreeBSD and Linux
old_RTTvar	Cached and shared in FreeBSD
old TFOinfo	Cached and shared in Linux and Windows
old_snd_cwnd	Not shared
old_ssthresh	Cached and shared in FreeBSD and Linux: FreeBSD: arithmetic mean of ssthresh and previous value if a previous value exists; Linux: depending on state, max(cwnd/2, ssthresh) in most cases

11. Updates to RFC 2140

This document updates the description of TCB sharing in RFC 2140 and its associated impact on existing and new connection state, providing a complete replacement for that document [RFC2140]. It clarifies the previous description and terminology and extends the mechanism to its impact on new protocols and mechanisms, including multipath TCP, fast open, PLPMTUD, NAT, and the TCP Authentication Option.

The detailed impact on TCB state addresses TCB parameters in greater detail, addressing RSS in both the send and receive direction, MSS and send-MSS separately, adds path MTU and ssthresh, and addresses the impact on TCP option state.

New sections have been added to address compatibility issues and implementation observations. The relation of this work to T/TCP has

been moved to an appendix discussion on history, partly to reflect the deprecation of that protocol.

Finally, this document updates and significantly expands the referenced literature.

12. Security Considerations

These presented implementation methods do not have additional ramifications for explicit attacks. They may be susceptible to denial-of-service attacks if not otherwise secured. For example, an application can open a connection and set its window size to zero, denying service to any other subsequent connection between those hosts.

TCB sharing may be susceptible to denial-of-service attacks, wherever the TCB is shared, between connections in a single host, or between hosts if TCB sharing is implemented within a subnet (see Implications section). Some shared TCB parameters are used only to create new TCBs, others are shared among the TCBs of ongoing connections. New connections can join the ongoing set, e.g., to optimize send window size among a set of connections to the same host.

Attacks on parameters used only for initialization affect only the transient performance of a TCP connection. For short connections, the performance ramification can approach that of a denial-of-service attack. E.g., if an application changes its TCB to have a false and small window size, subsequent connections would experience performance degradation until their window grew appropriately.

13. IANA Considerations

There are no IANA implications or requests in this document.

This section should be removed upon final publication as an RFC.

14. References

14.1. Normative References

[RFC793] Postel, Jon, "Transmission Control Protocol," Network Working Group RFC-793/STD-7, ISI, Sept. 1981.

[RFC1191] Mogul, J., Deering, S., "Path MTU Discovery," RFC 1191, Nov. 1990.

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

[RFC2140] Touch, J., "TCP Control Block Interdependence", RFC 2140, April 1997.

[RFC4821] Mathis, M., Heffner, J., "Packetization Layer Path MTU Discovery," RFC 4821, Mar. 2007.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., Jain, A., "TCP Fast Open", RFC 7413, Dec. 2014.

[RFC8201] McCann, J., Deering. S., Mogul, J., Hinden, R. (Ed.), "Path MTU Discovery for IP version 6," RFC 8201, Jul. 2017.

14.2. Informative References

[Br02] Brownlee, N. and K. Claffy, "Understanding Internet Traffic Streams: Dragonflies and Tortoises", IEEE Communications Magazine p110-117, 2002.

[Be94] Berners-Lee, T., et al., "The World-Wide Web," Communications of the ACM, V37, Aug. 1994, pp. 76-82.

[Br94] Braden, B., "T/TCP -- Transaction TCP: Source Changes for Sun OS 4.1.3,", Release 1.0, USC/ISI, September 14, 1994.

[Co91] Comer, D., Stevens, D., Internetworking with TCP/IP, V2, Prentice-Hall, NJ, 1991.

[FreeBSD] FreeBSD source code, Release 2.10, http://www.freebsd.org/

[Ja86] Jacobson, V., (mail to public list "tcp-ip", no archive found), 1986.

[Du16] Dukkipati, N., Yuchung C., and Amin V., "Research Impacting the Practice of Congestion Control." ACM SIGCOMM CCR (editorial), on-line post, July, 2016.

[Hu01] Hugues, A., Touch, J., Heidemann, J., "Issues in Slow-Start Restart After Idle", draft-hughes-restart-00 (expired), Dec., 2001.

[Hu12] Hurtig, P., Brunstrom, A., "Enhanced metric caching for short TCP flows," 2012 IEEE International Conference on Communications (ICC), Ottawa, ON, 2012, pp. 1209-1213.

[Ba12] Barik, R., Welzl, M., Ferlin, S., Alay, O., " LISA: A Linked Slow-Start Algorithm for MPTCP", IEEE ICC, Kuala Lumpur, Malaysia, 23-27 May 2016.

[RFC1122] Braden, R. (ed), "Requirements for Internet Hosts -- Communication Layers", RFC-1122, Oct. 1989.

[RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification," RFC-1644, July 1994.

[RFC1379] Braden, R., "Transaction TCP -- Concepts," RFC-1379, September 1992.

[RFC2663] Srisuresh, P., Holdrege, M., "IP Network Address Translator (NAT) Terminology and Considerations", RFC-2663, August 1999.

[RFC3390] Allman, M., Floyd, S., Partridge, C., "Increasing TCP's Initial Window," RFC 3390, Oct. 2002.

[RFC7231] Fielding, R., J. Reshke, Eds., "HTTP/1.1 Semantics and Content," RFC-7231, June 2014.

[RFC3124] Balakrishnan, H., Seshan, S., "The Congestion Manager," RFC 3124, June 2001.

[RFC4340] Kohler, E., Handley, M., Floyd, S., "Datagram Congestion Control Protocol (DCCP)," RFC 4340, Mar. 2006.

[RFC4960] Stewart, R., (Ed.), "Stream Control Transmission Protocol," RFC4960, Sept. 2007.

[RFC5861] Allman, M., Paxson, V., Blanton, E., "TCP Congestion Control," RFC 5861, Sept. 2009.

[RFC5925] Touch, J., Mankin, A., Bonica, R., "The TCP Authentication Option," RFC 5925, June 2010.

[RFC6824] Ford, A., Raiciu, C., Handley, M., Bonaventure, O., "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824, Jan. 2013.

[RFC6928] Chu, J., Dukkipati, N., Cheng, Y., Mathis, M., "Increasing TCP's Initial Window," RFC 6928, Apr. 2013.

[RFC7424] Krishnan, R., Yong, L., Ghanwani, A., So, N., Khasnabish, B., "Mechanisms for Optimizing Link Aggregation Group (LAG) and Equal-Cost Multipath (ECMP) Component Link Utilization in Networks", RFC 7424, Jan. 2015

[RFC7540] Belshe, M., Peon, R., Thomson, M., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, May 2015.

[RFC7661] Fairhurst, G., Sathiaseelan, A., Secchi, R., "Updating TCP to Support Rate-Limited Traffic", RFC 7661, Oct. 2015

15. Acknowledgments

The authors would like to thank for Praveen Balasubramanian for information regarding TCB sharing in Windows, and Yuchung Cheng, Lars Eggert, Ilpo Jarvinen and Michael Scharf for comments on earlier versions of the draft. Earlier revisions of this work received funding from a collaborative research project between the University of Oslo and Huawei Technologies Co., Ltd. and were partly supported by USC/ISI's Postel Center.

This document was prepared using 2-Word-v2.0.template.dot.

16. Change log

This section should be removed upon final publication as an RFC.

06:

- Changed to update 2140, cite it normatively, and summarize the updates in a separate section

05:

- Fixed some TBDs.

04:

- Removed BCP-style recommendations and fixed some TBDs.

03:

- Updated Touch's affiliation and address information

02:

- Stated that our OS implementation overview table only covers temporal sharing.

- Correctly reflected sharing of old_RTT in Linux in the implementation overview table.

- Marked entries that are considered safe to share with an asterisk (suggestion was to split the table)

- Discussed correct host identification: NATs may make IP addresses the wrong input, could e.g. use HTTP cookie.

- Included MMS_S and MMS_R from RFC1122; fixed the use of MSS and MTU

- Added information about option sharing, listed options in the appendix

Authors' Addresses

Joe Touch

Manhattan Beach, CA 90266
USA

Phone: +1 (310) 560-0334
Email: touch@strayalpha.com

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Safiqul Islam
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 84 08 37
Email: safiquli@ifi.uio.no

17. Appendix A: TCB sharing history

T/TCP proposed using caches to maintain TCB information across instances (temporal sharing), e.g., smoothed RTT, RTT variance, congestion avoidance threshold, and MSS [RFC1644]. These values were in addition to connection counts used by T/TCP to accelerate data delivery prior to the full three-way handshake during an OPEN. The goal was to aggregate TCB components where they reflect one association - that of the host-pair, rather than artificially separating those components by connection.

At least one T/TCP implementation saved the MSS and aggregated the RTT parameters across multiple connections, but omitted caching the congestion window information [Br94], as originally specified in [RFC1379]. Some T/TCP implementations immediately updated MSS when the TCP MSS header option was received [Br94], although this was not addressed specifically in the concepts or functional specification [RFC1379][RFC1644]. In later T/TCP implementations, RTT values were updated only after a CLOSE, which does not benefit concurrent sessions.

Temporal sharing of cached TCB data was originally implemented in the SunOS 4.1.3 T/TCP extensions [Br94] and the FreeBSD port of same [FreeBSD]. As mentioned before, only the MSS and RTT parameters were cached, as originally specified in [RFC1379]. Later discussion of T/TCP suggested including congestion control parameters in this cache [RFC1644].

18. Appendix B: Options

In addition to the options that can be cached and shared, this memo also lists all options for which state should *not* be kept. This list is meant to avoid work duplication and should be removed upon publication.

Obsolete (MUST NOT keep state):

ECHO

ECHO REPLY

PO Conn permitted

PO service profile

CC

CC.NEW

CC.ECHO

Alt CS req

Alt CS data

No state to keep:

EOL

NOP

WS

SACK

TS

MD5

TCP-AO

EXP1

EXP2

MUST NOT keep state:

Skeeter (DH exchange - might be obsolete, though)

Bubba (DH exchange - might really be obsolete, though)

Trailer CS

SCPS capabilities

S-NACK

Records boundaries

Corruption experienced

SNAP

TCP Compression

Quickstart response

UTO

MPTCP (can we cache when this fails?)

TFO success

MAY keep state:

MSS

TFO failure (so we don't try again, since it's optional)

MUST keep state:

TFP cookie (if TFO succeeded in the past)