Network Working Group
Request for Comments: 2525
Category: Informational

V. Paxson
Editor
ACIRI / ICSI
M. Allman
NASA Glenn Research Center/Sterling Software
S. Dawson
Real-Time Computing Laboratory
W. Fenner
Xerox PARC
J. Griner
NASA Glenn Research Center
I. Heavens
Spider Software Ltd.
K. Lahey
NASA Ames Research Center/MRJ
J. Semke
Pittsburgh Supercomputing Center
B. Volz
Process Software Corporation

March 1999

Known TCP Implementation Problems

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

This memo catalogs a number of known TCP implementation problems. The goal in doing so is to improve conditions in the existing Internet by enhancing the quality of current TCP/IP implementations. It is hoped that both performance and correctness issues can be resolved by making implementors aware of the problems and their solutions. In the long term, it is hoped that this will provide a reduction in unnecessary traffic on the network, the rate of connection failures due to protocol errors, and load on network servers due to time spent processing both unsuccessful connections and retransmitted data. This will help to ensure the stability of the global Internet.

Each problem is defined as follows:

Name of Problem
: The name associated with the problem. In this memo, the name is given as a subsection heading.

Classification
: One or more problem categories for which the problem is classified: "congestion control", "performance", "reliability", "resource management".

Description
: A definition of the problem, succinct but including necessary background material.

Significance
: A brief summary of the sorts of environments for which the problem is significant.

Implications
Why the problem is viewed as a problem.

Relevant RFCs
The RFCs defining the TCP specification with which the problem conflicts. These RFCs often qualify behavior using terms such as MUST, SHOULD, MAY, and others written capitalized. See RFC 2119 for the exact interpretation of these terms.

Trace file demonstrating the problem
One or more ASCII trace files demonstrating the problem, if applicable.

Trace file demonstrating correct behavior
One or more examples of how correct behavior appears in a trace, if applicable.

References
References that further discuss the problem.

How to detect
How to test an implementation to see if it exhibits the problem. This discussion may include difficulties and subtleties associated with causing the problem to manifest itself, and with interpreting traces to detect the presence of the problem (if applicable).

How to fix
For known causes of the problem, how to correct the implementation.

2. Known implementation problems

2.1.

Name of Problem
No initial slow start

Classification
Congestion control

Description
When a TCP begins transmitting data, it is required by RFC 1122, 4.2.2.15, to engage in a "slow start" by initializing its congestion window, cwnd, to one packet (one segment of the maximum size). (Note that an experimental change to TCP, documented in [RFC2414], allows an initial value somewhat larger than one packet.) It subsequently increases cwnd by one packet for each ACK it receives for new data. The minimum of cwnd and the

receiver's advertised window bounds the highest sequence number the TCP can transmit. A TCP that fails to initialize and increment cwnd in this fashion exhibits "No initial slow start".

Significance

In congested environments, detrimental to the performance of other connections, and possibly to the connection itself.

Implications

A TCP failing to slow start when beginning a connection results in traffic bursts that can stress the network, leading to excessive queueing delays and packet loss.

Implementations exhibiting this problem might do so because they suffer from the general problem of not including the required congestion window. These implementations will also suffer from "No slow start after retransmission timeout".

There are different shades of "No initial slow start". From the perspective of stressing the network, the worst is a connection that simply always sends based on the receiver's advertised window, with no notion of a separate congestion window. Another form is described in "Uninitialized CWND" below.

Relevant RFCs

RFC 1122 requires use of slow start. RFC 2001 gives the specifics of slow start.

Trace file demonstrating it

Made using tcpdump [Jacobson89] recording at the connection responder. No losses reported by the packet filter.

```
10:40:42.244503 B > A: S 1168512000:1168512000(0) win 32768
                        <mss 1460,nop,wscale 0> (DF) [tos 0x8]
10:40:42.259908 A > B: S 3688169472:3688169472(0)
                        ack 1168512001 win 32768 <mss 1460>
10:40:42.389992 B > A: . ack 1 win 33580 (DF) [tos 0x8]
10:40:42.664975 A > B: P 1:513(512) ack 1 win 32768
10:40:42.700185 A > B: . 513:1973(1460) ack 1 win 32768
10:40:42.718017 A > B: . 1973:3433(1460) ack 1 win 32768
10:40:42.762945 A > B: . 3433:4893(1460) ack 1 win 32768
10:40:42.811273 A > B: . 4893:6353(1460) ack 1 win 32768
10:40:42.829149 A > B: . 6353:7813(1460) ack 1 win 32768
10:40:42.853687 B > A: . ack 1973 win 33580 (DF) [tos 0x8]
10:40:42.864031 B > A: . ack 3433 win 33580 (DF) [tos 0x8]
```

After the third packet, the connection is established. A, the connection responder, begins transmitting to B, the connection initiator. Host A quickly sends 6 packets comprising 7812 bytes, even though the SYN exchange agreed upon an MSS of 1460 bytes (implying an initial congestion window of 1 segment corresponds to 1460 bytes), and so A should have sent at most 1460 bytes.

The ACKs sent by B to A in the last two lines indicate that this trace is not a measurement error (slow start really occurring but the corresponding ACKs having been dropped by the packet filter).

A second trace confirmed that the problem is repeatable.

Trace file demonstrating correct behavior

Made using tcpdump recording at the connection originator. No losses reported by the packet filter.

```
12:35:31.914050 C > D: S 1448571845:1448571845(0)
                         win 4380 <mss 1460>
12:35:32.068819 D > C: S 1755712000:1755712000(0)
                         ack 1448571846 win 4096
12:35:32.069341 C > D: . ack 1 win 4608
12:35:32.075213 C > D: P 1:513(512) ack 1 win 4608
12:35:32.286073 D > C: . ack 513 win 4096
12:35:32.287032 C > D: . 513:1025(512) ack 1 win 4608
12:35:32.287506 C > D: . 1025:1537(512) ack 1 win 4608
12:35:32.432712 D > C: . ack 1537 win 4096
12:35:32.433690 C > D: . 1537:2049(512) ack 1 win 4608
12:35:32.434481 C > D: . 2049:2561(512) ack 1 win 4608
12:35:32.435032 C > D: . 2561:3073(512) ack 1 win 4608
12:35:32.594526 D > C: . ack 3073 win 4096
12:35:32.595465 C > D: . 3073:3585(512) ack 1 win 4608
12:35:32.595947 C > D: . 3585:4097(512) ack 1 win 4608
12:35:32.596414 C > D: . 4097:4609(512) ack 1 win 4608
12:35:32.596888 C > D: . 4609:5121(512) ack 1 win 4608
12:35:32.733453 D > C: . ack 4097 win 4096
```

References

This problem is documented in [Paxson97].

How to detect

For implementations always manifesting this problem, it shows up immediately in a packet trace or a sequence plot, as illustrated above.

How to fix
If the root problem is that the implementation lacks a notion of a congestion window, then unfortunately this requires significant work to fix. However, doing so is important, as such implementations also exhibit "No slow start after retransmission timeout".

2.2.

Name of Problem
No slow start after retransmission timeout

Classification
Congestion control

Description
When a TCP experiences a retransmission timeout, it is required by RFC 1122, 4.2.2.15, to engage in "slow start" by initializing its congestion window, cwnd, to one packet (one segment of the maximum size). It subsequently increases cwnd by one packet for each ACK it receives for new data until it reaches the "congestion avoidance" threshold, ssthresh, at which point the congestion avoidance algorithm for updating the window takes over. A TCP that fails to enter slow start upon a timeout exhibits "No slow start after retransmission timeout".

Significance
In congested environments, severely detrimental to the performance of other connections, and also the connection itself.

Implications
Entering slow start upon timeout forms one of the cornerstones of Internet congestion stability, as outlined in [Jacobson88]. If TCPs fail to do so, the network becomes at risk of suffering "congestion collapse" [RFC896].

Relevant RFCs
RFC 1122 requires use of slow start after loss. RFC 2001 gives the specifics of how to implement slow start. RFC 896 describes congestion collapse.

The retransmission timeout discussed here should not be confused with the separate "fast recovery" retransmission mechanism discussed in RFC 2001.

Trace file demonstrating it
Made using tcpdump recording at the sending TCP (A). No losses reported by the packet filter.

```
10:40:59.090612 B > A: . ack 357125 win 33580 (DF) [tos 0x8]
10:40:59.222025 A > B: . 357125:358585(1460) ack 1 win 32768
10:40:59.868871 A > B: . 357125:358585(1460) ack 1 win 32768
10:41:00.016641 B > A: . ack 364425 win 33580 (DF) [tos 0x8]
10:41:00.036709 A > B: . 364425:365885(1460) ack 1 win 32768
10:41:00.045231 A > B: . 365885:367345(1460) ack 1 win 32768
10:41:00.053785 A > B: . 367345:368805(1460) ack 1 win 32768
10:41:00.062426 A > B: . 368805:370265(1460) ack 1 win 32768
10:41:00.071074 A > B: . 370265:371725(1460) ack 1 win 32768
10:41:00.079794 A > B: . 371725:373185(1460) ack 1 win 32768
10:41:00.089304 A > B: . 373185:374645(1460) ack 1 win 32768
10:41:00.097738 A > B: . 374645:376105(1460) ack 1 win 32768
10:41:00.106409 A > B: . 376105:377565(1460) ack 1 win 32768
10:41:00.115024 A > B: . 377565:379025(1460) ack 1 win 32768
10:41:00.123576 A > B: . 379025:380485(1460) ack 1 win 32768
10:41:00.132016 A > B: . 380485:381945(1460) ack 1 win 32768
10:41:00.141635 A > B: . 381945:383405(1460) ack 1 win 32768
10:41:00.150094 A > B: . 383405:384865(1460) ack 1 win 32768
10:41:00.158552 A > B: . 384865:386325(1460) ack 1 win 32768
10:41:00.167053 A > B: . 386325:387785(1460) ack 1 win 32768
10:41:00.175518 A > B: . 387785:389245(1460) ack 1 win 32768
10:41:00.210835 A > B: . 389245:390705(1460) ack 1 win 32768
10:41:00.226108 A > B: . 390705:392165(1460) ack 1 win 32768
10:41:00.241524 B > A: . ack 389245 win 8760 (DF) [tos 0x8]
```

The first packet indicates the ack point is 357125. 130 msec after receiving the ACK, A transmits the packet after the ACK point, 357125:358585. 640 msec after this transmission, it retransmits 357125:358585, in an apparent retransmission timeout. At this point, A's cwnd should be one MSS, or 1460 bytes, as A enters slow start. The trace is consistent with this possibility.

B replies with an ACK of 364425, indicating that A has filled a sequence hole. At this point, A's cwnd should be 1460*2 = 2920 bytes, since in slow start receiving an ACK advances cwnd by MSS. However, A then launches 19 consecutive packets, which is inconsistent with slow start.

A second trace confirmed that the problem is repeatable.

Trace file demonstrating correct behavior

Made using tcpdump recording at the sending TCP (C). No losses reported by the packet filter.

```
12:35:48.442538 C > D: P 465409:465921(512) ack 1 win 4608
12:35:48.544483 D > C: . ack 461825 win 4096
12:35:48.703496 D > C: . ack 461825 win 4096
12:35:49.044613 C > D: . 461825:462337(512) ack 1 win 4608
```

```
12:35:49.192282 D > C: . ack 465921 win 2048
12:35:49.192538 D > C: . ack 465921 win 4096
12:35:49.193392 C > D: P 465921:466433(512) ack 1 win 4608
12:35:49.194726 C > D: P 466433:466945(512) ack 1 win 4608
12:35:49.350665 D > C: . ack 466945 win 4096
12:35:49.351694 C > D: . 466945:467457(512) ack 1 win 4608
12:35:49.352168 C > D: . 467457:467969(512) ack 1 win 4608
12:35:49.352643 C > D: . 467969:468481(512) ack 1 win 4608
12:35:49.506000 D > C: . ack 467969 win 3584
```

After C transmits the first packet shown to D, it takes no action in response to D's ACKs for 461825, because the first packet already reached the advertised window limit of 4096 bytes above 461825. 600 msec after transmitting the first packet, C retransmits 461825:462337, presumably due to a timeout. Its congestion window is now MSS (512 bytes).

D acks 465921, indicating that C's retransmission filled a sequence hole. This ACK advances C's cwnd from 512 to 1024. Very shortly after, D acks 465921 again in order to update the offered window from 2048 to 4096. This ACK does not advance cwnd since it is not for new data. Very shortly after, C responds to the newly enlarged window by transmitting two packets. D acks both, advancing cwnd from 1024 to 1536. C in turn transmits three packets.

References

This problem is documented in [Paxson97].

How to detect

Packet loss is common enough in the Internet that generally it is not difficult to find an Internet path that will force retransmission due to packet loss.

If the effective window prior to loss is large enough, however, then the TCP may retransmit using the "fast recovery" mechanism described in RFC 2001. In a packet trace, the signature of fast recovery is that the packet retransmission occurs in response to the receipt of three duplicate ACKs, and subsequent duplicate ACKs may lead to the transmission of new data, above both the ack point and the highest sequence transmitted so far. An absence of three duplicate ACKs prior to retransmission suffices to distinguish between timeout and fast recovery retransmissions. In the face of only observing fast recovery retransmissions, generally it is not difficult to repeat the data transfer until observing a timeout retransmission.

Once armed with a trace exhibiting a timeout retransmission, determining whether the TCP follows slow start is done by computing the correct progression of cwnd and comparing it to the amount of data transmitted by the TCP subsequent to the timeout retransmission.

How to fix
If the root problem is that the implementation lacks a notion of a congestion window, then unfortunately this requires significant work to fix. However, doing so is critical, for reasons outlined above.

2.3.

Name of Problem
Uninitialized CWND

Classification
Congestion control

Description
As described above for "No initial slow start", when a TCP connection begins cwnd is initialized to one segment (or perhaps a few segments, if experimenting with [RFC2414]). One particular form of "No initial slow start", worth separate mention as the bug is fairly widely deployed, is "Uninitialized CWND". That is, while the TCP implements the proper slow start mechanism, it fails to initialize cwnd properly, so slow start in fact fails to occur.

One way the bug can occur is if, during the connection establishment handshake, the SYN ACK packet arrives without an MSS option. The faulty implementation uses receipt of the MSS option to initialize cwnd to one segment; if the option fails to arrive, then cwnd is instead initialized to a very large value.

Significance
In congested environments, detrimental to the performance of other connections, and likely to the connection itself. The burst can be so large (see below) that it has deleterious effects even in uncongested environments.

Implications
A TCP exhibiting this behavior is stressing the network with a large burst of packets, which can cause loss in the network.

Relevant RFCs
RFC 1122 requires use of slow start. RFC 2001 gives the specifics of slow start.

Trace file demonstrating it

This trace was made using tcpdump running on host A. Host A is the sender and host B is the receiver. The advertised window and timestamp options have been omitted for clarity, except for the first segment sent by host A. Note that A sends an MSS option in its initial SYN but B does not include one in its reply.

```
16:56:02.226937 A > B: S 237585307:237585307(0) win 8192
      <mss 536,nop,wscale 0,nop,nop,timestamp[|tcp]>
16:56:02.557135 B > A: S 1617216000:1617216000(0)
      ack 237585308 win 16384
16:56:02.557788 A > B: . ack 1 win 8192
16:56:02.566014 A > B: . 1:537(536) ack 1
16:56:02.566557 A > B: . 537:1073(536) ack 1
16:56:02.567120 A > B: . 1073:1609(536) ack 1
16:56:02.567662 A > B: P 1609:2049(440) ack 1
16:56:02.568349 A > B: . 2049:2585(536) ack 1
16:56:02.568909 A > B: . 2585:3121(536) ack 1

   [54 additional burst segments deleted for brevity]

16:56:02.936638 A > B: . 32065:32601(536) ack 1
16:56:03.018685 B > A: . ack 1
```

After the three-way handshake, host A bursts 61 segments into the network, before duplicate ACKs on the first segment cause a retransmission to occur. Since host A did not wait for the ACK on the first segment before sending additional segments, it is exhibiting "Uninitialized CWND"

Trace file demonstrating correct behavior

See the example for "No initial slow start".

References

This problem is documented in [Paxson97].

How to detect

This problem can be detected by examining a packet trace recorded at either the sender or the receiver. However, the bug can be difficult to induce because it requires finding a remote TCP peer that does not send an MSS option in its SYN ACK.

How to fix

This problem can be fixed by ensuring that cwnd is initialized upon receipt of a SYN ACK, even if the SYN ACK does not contain an MSS option.

2.4.

Name of Problem
Inconsistent retransmission

Classification
Reliability

Description
If, for a given sequence number, a sending TCP retransmits different data than previously sent for that sequence number, then a strong possibility arises that the receiving TCP will reconstruct a different byte stream than that sent by the sending application, depending on which instance of the sequence number it accepts.

Such a sending TCP exhibits "Inconsistent retransmission".

Significance
Critical for all environments.

Implications
Reliable delivery of data is a fundamental property of TCP.

Relevant RFCs
RFC 793, section 1.5, discusses the central role of reliability in TCP operation.

Trace file demonstrating it
Made using tcpdump recording at the receiving TCP (B). No losses reported by the packet filter.

```
12:35:53.145503 A > B: FP 90048435:90048461(26)
                           ack 393464682 win 4096
                                        4500 0042 9644 0000
                 3006 e4c2 86b1 0401 83f3 010a b2a4 0015
                 055e 07b3 1773 cb6a 5019 1000 68a9 0000
data starts here>504f 5254 2031 3334 2c31 3737*2c34 2c31
                 2c31 3738 2c31 3635 0d0a
12:35:53.146479 B > A: R 393464682:393464682(0) win 8192
12:35:53.851714 A > B: FP 90048429:90048463(34)
                          ack 393464682 win 4096
                                        4500 004a 965b 0000
                 3006 e4a3 86b1 0401 83f3 010a b2a4 0015
                 055e 07ad 1773 cb6a 5019 1000 8bd3 0000
data starts here>5041 5356 0d0a 504f 5254 2031 3334 2c31
                 3737*2c31 3035 2c31 3431 2c34 2c31 3539
                 0d0a
```

The sequence numbers shown in this trace are absolute and not adjusted to reflect the ISN. The 4-digit hex values show a dump of the packet's IP and TCP headers, as well as payload. A first sends to B data for 90048435:90048461. The corresponding data begins with hex words 504f, 5254, etc.

B responds with a RST. Since the recording location was local to B, it is unknown whether A received the RST.

A then sends 90048429:90048463, which includes six sequence positions below the earlier transmission, all 26 positions of the earlier transmission, and two additional sequence positions.

The retransmission disagrees starting just after sequence 90048447, annotated above with a leading '*'. These two bytes were originally transmitted as hex 2c34 but retransmitted as hex 2c31. Subsequent positions disagree as well.

This behavior has been observed in other traces involving different hosts. It is unknown how to repeat it.

In this instance, no corruption would occur, since B has already indicated it will not accept further packets from A.

A second example illustrates a slightly different instance of the problem. The tracing again was made with tcpdump at the receiving TCP (D).

```
22:23:58.645829 C > D: P 185:212(27) ack 565 win 4096
                                     4500 0043 90a3 0000
                   3306 0734 cbf1 9eef 83f3 010a 0525 0015
                   a3a2 faba 578c 70a4 5018 1000 9a53 0000
data starts here>504f 5254 2032 3033 2c32 3431 2c31 3538
                   2c32 3339 2c35 2c34 330d 0a
22:23:58.646805 D > C: . ack 184 win 8192
                                     4500 0028 beeb 0000
                   3e06 ce06 83f3 010a cbf1 9eef 0015 0525
                   578c 70a4 a3a2 fab9 5010 2000 342f 0000
22:31:36.532244 C > D: FP 186:213(27) ack 565 win 4096
                                     4500 0043 9435 0000
                   3306 03a2 cbf1 9eef 83f3 010a 0525 0015
                   a3a2 fabb 578c 70a4 5019 1000 9a51 0000
data starts here>504f 5254 2032 3033 2c32 3431 2c31 3538
                   2c32 3339 2c35 2c34 330d 0a
```

In this trace, sequence numbers are relative. C sends 185:212, but D only sends an ACK for 184 (so sequence number 184 is missing). C then sends 186:213. The packet payload is identical to the previous payload, but the base sequence number is one higher, resulting in an inconsistent retransmission.

Neither trace exhibits checksum errors.

Trace file demonstrating correct behavior
(Omitted, as presumably correct behavior is obvious.)

References
None known.

How to detect
This problem unfortunately can be very difficult to detect, since available experience indicates it is quite rare that it is manifested. No "trigger" has been identified that can be used to reproduce the problem.

How to fix
In the absence of a known "trigger", we cannot always assess how to fix the problem.

In one implementation (not the one illustrated above), the problem manifested itself when (1) the sender received a zero window and stalled; (2) eventually an ACK arrived that offered a window larger than that in effect at the time of the stall; (3) the sender transmitted out of the buffer of data it held at the time of the stall, but (4) failed to limit this transfer to the buffer length, instead using the newly advertised (and larger) offered window. Consequently, in addition to the valid buffer contents, it sent whatever garbage values followed the end of the buffer. If it then retransmitted the corresponding sequence numbers, at that point it sent the correct data, resulting in an inconsistent retransmission. Note that this instance of the problem reflects a more general problem, that of initially transmitting incorrect data.

2.5.

Name of Problem
Failure to retain above-sequence data

Classification
Congestion control, performance

Description
When a TCP receives an "above sequence" segment, meaning one with a sequence number exceeding RCV.NXT but below RCV.NXT+RCV.WND, it SHOULD queue the segment for later delivery (RFC 1122, 4.2.2.20). (See RFC 793 for the definition of RCV.NXT and RCV.WND.) A TCP that fails to do so is said to exhibit "Failure to retain above-sequence data".

It may sometimes be appropriate for a TCP to discard above-sequence data to reclaim memory. If they do so only rarely, then we would not consider them to exhibit this problem. Instead, the particular concern is with TCPs that always discard above-sequence data.

Significance
In environments prone to packet loss, detrimental to the performance of both other connections and the connection itself.

Implications
In times of congestion, a failure to retain above-sequence data will lead to numerous otherwise-unnecessary retransmissions, aggravating the congestion and potentially reducing performance by a large factor.

Relevant RFCs
RFC 1122 revises RFC 793 by upgrading the latter’s MAY to a SHOULD on this issue.

Trace file demonstrating it
Made using tcpdump recording at the receiving TCP. No losses reported by the packet filter.

B is the TCP sender, A the receiver. A exhibits failure to retain above sequence-data:

```
10:38:10.164860 B > A: . 221078:221614(536) ack 1 win 33232 [tos 0x8]
10:38:10.170809 B > A: . 221614:222150(536) ack 1 win 33232 [tos 0x8]
10:38:10.177183 B > A: . 222150:222686(536) ack 1 win 33232 [tos 0x8]
10:38:10.225039 A > B: . ack 222686 win 25800
```

Here B has sent up to (relative) sequence 222686 in-sequence, and A accordingly acknowledges.

```
10:38:10.268131 B > A: . 223222:223758(536) ack 1 win 33232 [tos 0x8]
10:38:10.337995 B > A: . 223758:224294(536) ack 1 win 33232 [tos 0x8]
10:38:10.344065 B > A: . 224294:224830(536) ack 1 win 33232 [tos 0x8]
10:38:10.350169 B > A: . 224830:225366(536) ack 1 win 33232 [tos 0x8]
10:38:10.356362 B > A: . 225366:225902(536) ack 1 win 33232 [tos 0x8]
```

```
10:38:10.362445 B > A: . 225902:226438(536) ack 1 win 33232 [tos 0x8]
10:38:10.368579 B > A: . 226438:226974(536) ack 1 win 33232 [tos 0x8]
10:38:10.374732 B > A: . 226974:227510(536) ack 1 win 33232 [tos 0x8]
10:38:10.380825 B > A: . 227510:228046(536) ack 1 win 33232 [tos 0x8]
10:38:10.387027 B > A: . 228046:228582(536) ack 1 win 33232 [tos 0x8]
10:38:10.393053 B > A: . 228582:229118(536) ack 1 win 33232 [tos 0x8]
10:38:10.399193 B > A: . 229118:229654(536) ack 1 win 33232 [tos 0x8]
10:38:10.405356 B > A: . 229654:230190(536) ack 1 win 33232 [tos 0x8]
```

A now receives 13 additional packets from B. These are above-sequence because 222686:223222 was dropped. The packets do however fit within the offered window of 25800. A does not generate any duplicate ACKs for them.

The trace contributor (V. Paxson) verified that these 13 packets had valid IP and TCP checksums.

```
10:38:11.917728 B > A: . 222686:223222(536) ack 1 win 33232 [tos 0x8]
10:38:11.930925 A > B: . ack 223222 win 32232
```

B times out for 222686:223222 and retransmits it. Upon receiving it, A only acknowledges 223222. Had it retained the valid above-sequence packets, it would instead have ack'd 230190.

```
10:38:12.048438 B > A: . 223222:223758(536) ack 1 win 33232 [tos 0x8]
10:38:12.054397 B > A: . 223758:224294(536) ack 1 win 33232 [tos 0x8]
10:38:12.068029 A > B: . ack 224294 win 31696
```

B retransmits two more packets, and A only acknowledges them. This pattern continues as B retransmits the entire set of previously-received packets.

A second trace confirmed that the problem is repeatable.

Trace file demonstrating correct behavior

Made using tcpdump recording at the receiving TCP (C). No losses reported by the packet filter.

```
09:11:25.790417 D > C: . 33793:34305(512) ack 1 win 61440
09:11:25.791393 D > C: . 34305:34817(512) ack 1 win 61440
09:11:25.792369 D > C: . 34817:35329(512) ack 1 win 61440
09:11:25.792369 D > C: . 35329:35841(512) ack 1 win 61440
09:11:25.793345 D > C: . 36353:36865(512) ack 1 win 61440
09:11:25.794321 C > D: . ack 35841 win 59904
```

A sequence hole occurs because 35841:36353 has been dropped.

```
09:11:25.794321 D > C: . 36865:37377(512) ack 1 win 61440
09:11:25.794321 C > D: . ack 35841 win 59904
09:11:25.795297 D > C: . 37377:37889(512) ack 1 win 61440
09:11:25.795297 C > D: . ack 35841 win 59904
09:11:25.796273 C > D: . ack 35841 win 61440
09:11:25.798225 D > C: . 37889:38401(512) ack 1 win 61440
09:11:25.799201 C > D: . ack 35841 win 61440
09:11:25.807009 D > C: . 38401:38913(512) ack 1 win 61440
09:11:25.807009 C > D: . ack 35841 win 61440
(many additional lines omitted)
09:11:25.884113 D > C: . 52737:53249(512) ack 1 win 61440
09:11:25.884113 C > D: . ack 35841 win 61440
```

Each additional, above-sequence packet C receives from D elicits a duplicate ACK for 35841.

```
09:11:25.887041 D > C: . 35841:36353(512) ack 1 win 61440
09:11:25.887041 C > D: . ack 53249 win 44032
```

D retransmits 35841:36353 and C acknowledges receipt of data all the way up to 53249.

References

This problem is documented in [Paxson97].

How to detect

Packet loss is common enough in the Internet that generally it is not difficult to find an Internet path that will result in some above-sequence packets arriving. A TCP that exhibits "Failure to retain ..." may not generate duplicate ACKs for these packets. However, some TCPs that do retain above-sequence data also do not generate duplicate ACKs, so failure to do so does not definitively identify the problem. Instead, the key observation is whether upon retransmission of the dropped packet, data that was previously above-sequence is acknowledged.

Two considerations in detecting this problem using a packet trace are that it is easiest to do so with a trace made at the TCP receiver, in order to unambiguously determine which packets arrived successfully, and that such packets may still be correctly discarded if they arrive with checksum errors. The latter can be tested by capturing the entire packet contents and performing the IP and TCP checksum algorithms to verify their integrity; or by confirming that the packets arrive with the same checksum and contents as that with which they were sent, with a presumption that the sending TCP correctly calculates checksums for the packets it transmits.

It is considerably easier to verify that an implementation does NOT exhibit this problem. This can be done by recording a trace at the data sender, and observing that sometimes after a retransmission the receiver acknowledges a higher sequence number than just that which was retransmitted.

How to fix
If the root problem is that the implementation lacks buffer, then then unfortunately this requires significant work to fix. However, doing so is important, for reasons outlined above.

2.6.

Name of Problem
Extra additive constant in congestion avoidance

Classification
Congestion control / performance

Description
RFC 1122 section 4.2.2.15 states that TCP MUST implement Jacobson's "congestion avoidance" algorithm [Jacobson88], which calls for increasing the congestion window, cwnd, by:

```
MSS * MSS / cwnd
```

for each ACK received for new data [RFC2001]. This has the effect of increasing cwnd by approximately one segment in each round trip time.

Some TCP implementations add an additional fraction of a segment (typically MSS/8) to cwnd for each ACK received for new data [Stevens94, Wright95]:

```
(MSS * MSS / cwnd) + MSS/8
```

These implementations exhibit "Extra additive constant in congestion avoidance".

Significance
May be detrimental to performance even in completely uncongested environments (see Implications).

In congested environments, may also be detrimental to the performance of other connections.

Implications

The extra additive term allows a TCP to more aggressively open its congestion window (quadratic rather than linear increase). For congested networks, this can increase the loss rate experienced by all connections sharing a bottleneck with the aggressive TCP.

However, even for completely uncongested networks, the extra additive term can lead to diminished performance, as follows. In congestion avoidance, a TCP sender probes the network path to determine its available capacity, which often equates to the number of buffers available at a bottleneck link. With linear congestion avoidance, the TCP only probes for sufficient capacity (buffer) to hold one extra packet per RTT.

Thus, when it exceeds the available capacity, generally only one packet will be lost (since on the previous RTT it already found that the path could sustain a window with one less packet in flight). If the congestion window is sufficiently large, then the TCP will recover from this single loss using fast retransmission and avoid an expensive (in terms of performance) retransmission timeout.

However, when the additional additive term is used, then cwnd can increase by more than one packet per RTT, in which case the TCP probes more aggressively. If in the previous RTT it had reached the available capacity of the path, then the excess due to the extra increase will again be lost, but now this will result in multiple losses from the flight instead of a single loss. TCPs that do not utilize SACK [RFC2018] generally will not recover from multiple losses without incurring a retransmission timeout [Fall96,Hoe96], significantly diminishing performance.

Relevant RFCs

RFC 1122 requires use of the "congestion avoidance" algorithm.
RFC 2001 outlines the fast retransmit/fast recovery algorithms.
RFC 2018 discusses the SACK option.

Trace file demonstrating it

Recorded using tcpdump running on the same FDDI LAN as host A. Host A is the sender and host B is the receiver. The connection establishment specified an MSS of 4,312 bytes and a window scale factor of 4. We omit the establishment and the first 2.5 MB of data transfer, as the problem is best demonstrated when the window has grown to a large value. At the beginning of the trace excerpt, the congestion window is 31 packets. The connection is never receiver-window limited, so we omit window advertisements from the trace for clarity.

```
11:42:07.697951 B > A: . ack 2383006
11:42:07.699388 A > B: . 2508054:2512366(4312)
11:42:07.699962 A > B: . 2512366:2516678(4312)
11:42:07.700012 B > A: . ack 2391630
11:42:07.701081 A > B: . 2516678:2520990(4312)
11:42:07.701656 A > B: . 2520990:2525302(4312)
11:42:07.701739 B > A: . ack 2400254
11:42:07.702685 A > B: . 2525302:2529614(4312)
11:42:07.703257 A > B: . 2529614:2533926(4312)
11:42:07.703295 B > A: . ack 2408878
11:42:07.704414 A > B: . 2533926:2538238(4312)
11:42:07.704989 A > B: . 2538238:2542550(4312)
11:42:07.705040 B > A: . ack 2417502
11:42:07.705935 A > B: . 2542550:2546862(4312)
11:42:07.706506 A > B: . 2546862:2551174(4312)
11:42:07.706544 B > A: . ack 2426126
11:42:07.707480 A > B: . 2551174:2555486(4312)
11:42:07.708051 A > B: . 2555486:2559798(4312)
11:42:07.708088 B > A: . ack 2434750
11:42:07.709030 A > B: . 2559798:2564110(4312)
11:42:07.709604 A > B: . 2564110:2568422(4312)
11:42:07.710175 A > B: . 2568422:2572734(4312) *

11:42:07.710215 B > A: . ack 2443374
11:42:07.710799 A > B: . 2572734:2577046(4312)
11:42:07.711368 A > B: . 2577046:2581358(4312)
11:42:07.711405 B > A: . ack 2451998
11:42:07.712323 A > B: . 2581358:2585670(4312)
11:42:07.712898 A > B: . 2585670:2589982(4312)
11:42:07.712938 B > A: . ack 2460622
11:42:07.713926 A > B: . 2589982:2594294(4312)
11:42:07.714501 A > B: . 2594294:2598606(4312)
11:42:07.714547 B > A: . ack 2469246
11:42:07.715747 A > B: . 2598606:2602918(4312)
11:42:07.716287 A > B: . 2602918:2607230(4312)
11:42:07.716328 B > A: . ack 2477870
11:42:07.717146 A > B: . 2607230:2611542(4312)
11:42:07.717717 A > B: . 2611542:2615854(4312)
11:42:07.717762 B > A: . ack 2486494
11:42:07.718754 A > B: . 2615854:2620166(4312)
11:42:07.719331 A > B: . 2620166:2624478(4312)
11:42:07.719906 A > B: . 2624478:2628790(4312) **

11:42:07.719958 B > A: . ack 2495118
11:42:07.720500 A > B: . 2628790:2633102(4312)
11:42:07.721080 A > B: . 2633102:2637414(4312)
11:42:07.721739 B > A: . ack 2503742
11:42:07.722348 A > B: . 2637414:2641726(4312)
```

```
11:42:07.722918 A > B: . 2641726:2646038(4312)
11:42:07.769248 B > A: . ack 2512366
```

The receiver's acknowledgment policy is one ACK per two packets received. Thus, for each ACK arriving at host A, two new packets are sent, except when cwnd increases due to congestion avoidance, in which case three new packets are sent.

With an ack-every-two-packets policy, cwnd should only increase one MSS per 2 RTT. However, at the point marked "*" the window increases after 7 ACKs have arrived, and then again at "**" after 6 more ACKs.

While we do not have space to show the effect, this trace suffered from repeated timeout retransmissions due to multiple packet losses during a single RTT.

Trace file demonstrating correct behavior

Made using the same host and tracing setup as above, except now A's TCP has been modified to remove the MSS/8 additive constant. Tcpdump reported 77 packet drops; the excerpt below is fully self-consistent so it is unlikely that any of these occurred during the excerpt.

We again begin when cwnd is 31 packets (this occurs significantly later in the trace, because the congestion avoidance is now less aggressive with opening the window).

```
14:22:21.236757 B > A: . ack 5194679
14:22:21.238192 A > B: . 5319727:5324039(4312)
14:22:21.238770 A > B: . 5324039:5328351(4312)
14:22:21.238821 B > A: . ack 5203303
14:22:21.240158 A > B: . 5328351:5332663(4312)
14:22:21.240738 A > B: . 5332663:5336975(4312)
14:22:21.270422 B > A: . ack 5211927
14:22:21.271883 A > B: . 5336975:5341287(4312)
14:22:21.272458 A > B: . 5341287:5345599(4312)
14:22:21.279099 B > A: . ack 5220551
14:22:21.280539 A > B: . 5345599:5349911(4312)
14:22:21.281118 A > B: . 5349911:5354223(4312)
14:22:21.281183 B > A: . ack 5229175
14:22:21.282348 A > B: . 5354223:5358535(4312)
14:22:21.283029 A > B: . 5358535:5362847(4312)
14:22:21.283089 B > A: . ack 5237799
14:22:21.284213 A > B: . 5362847:5367159(4312)
14:22:21.284779 A > B: . 5367159:5371471(4312)
14:22:21.285976 B > A: . ack 5246423
14:22:21.287465 A > B: . 5371471:5375783(4312)
```

```
14:22:21.288036 A > B: . 5375783:5380095(4312)
14:22:21.288073 B > A: . ack 5255047
14:22:21.289155 A > B: . 5380095:5384407(4312)
14:22:21.289725 A > B: . 5384407:5388719(4312)
14:22:21.289762 B > A: . ack 5263671
14:22:21.291090 A > B: . 5388719:5393031(4312)
14:22:21.291662 A > B: . 5393031:5397343(4312)
14:22:21.291701 B > A: . ack 5272295
14:22:21.292870 A > B: . 5397343:5401655(4312)
14:22:21.293441 A > B: . 5401655:5405967(4312)
14:22:21.293481 B > A: . ack 5280919
14:22:21.294476 A > B: . 5405967:5410279(4312)
14:22:21.295053 A > B: . 5410279:5414591(4312)
14:22:21.295106 B > A: . ack 5289543
14:22:21.296306 A > B: . 5414591:5418903(4312)
14:22:21.296878 A > B: . 5418903:5423215(4312)
14:22:21.296917 B > A: . ack 5298167
14:22:21.297716 A > B: . 5423215:5427527(4312)
14:22:21.298285 A > B: . 5427527:5431839(4312)
14:22:21.298324 B > A: . ack 5306791
14:22:21.299413 A > B: . 5431839:5436151(4312)
14:22:21.299986 A > B: . 5436151:5440463(4312)
14:22:21.303696 B > A: . ack 5315415
14:22:21.305177 A > B: . 5440463:5444775(4312)
14:22:21.305755 A > B: . 5444775:5449087(4312)
14:22:21.308032 B > A: . ack 5324039
14:22:21.309525 A > B: . 5449087:5453399(4312)
14:22:21.310101 A > B: . 5453399:5457711(4312)
14:22:21.310144 B > A: . ack 5332663              ***

14:22:21.311615 A > B: . 5457711:5462023(4312)
14:22:21.312198 A > B: . 5462023:5466335(4312)
14:22:21.341876 B > A: . ack 5341287
14:22:21.343451 A > B: . 5466335:5470647(4312)
14:22:21.343985 A > B: . 5470647:5474959(4312)
14:22:21.350304 B > A: . ack 5349911
14:22:21.351852 A > B: . 5474959:5479271(4312)
14:22:21.352430 A > B: . 5479271:5483583(4312)
14:22:21.352484 B > A: . ack 5358535
14:22:21.353574 A > B: . 5483583:5487895(4312)
14:22:21.354149 A > B: . 5487895:5492207(4312)
14:22:21.354205 B > A: . ack 5367159
14:22:21.355467 A > B: . 5492207:5496519(4312)
14:22:21.356039 A > B: . 5496519:5500831(4312)
14:22:21.357361 B > A: . ack 5375783
14:22:21.358855 A > B: . 5500831:5505143(4312)
14:22:21.359424 A > B: . 5505143:5509455(4312)
14:22:21.359465 B > A: . ack 5384407
```

```
14:22:21.360605 A > B: . 5509455:5513767(4312)
14:22:21.361181 A > B: . 5513767:5518079(4312)
14:22:21.361225 B > A: . ack 5393031
14:22:21.362485 A > B: . 5518079:5522391(4312)
14:22:21.363057 A > B: . 5522391:5526703(4312)
14:22:21.363096 B > A: . ack 5401655
14:22:21.364236 A > B: . 5526703:5531015(4312)
14:22:21.364810 A > B: . 5531015:5535327(4312)
14:22:21.364867 B > A: . ack 5410279
14:22:21.365819 A > B: . 5535327:5539639(4312)
14:22:21.366386 A > B: . 5539639:5543951(4312)
14:22:21.366427 B > A: . ack 5418903
14:22:21.367586 A > B: . 5543951:5548263(4312)
14:22:21.368158 A > B: . 5548263:5552575(4312)
14:22:21.368199 B > A: . ack 5427527
14:22:21.369189 A > B: . 5552575:5556887(4312)
14:22:21.369758 A > B: . 5556887:5561199(4312)
14:22:21.369803 B > A: . ack 5436151
14:22:21.370814 A > B: . 5561199:5565511(4312)
14:22:21.371398 A > B: . 5565511:5569823(4312)
14:22:21.375159 B > A: . ack 5444775
14:22:21.376658 A > B: . 5569823:5574135(4312)
14:22:21.377235 A > B: . 5574135:5578447(4312)
14:22:21.379303 B > A: . ack 5453399
14:22:21.380802 A > B: . 5578447:5582759(4312)
14:22:21.381377 A > B: . 5582759:5587071(4312)
14:22:21.381947 A > B: . 5587071:5591383(4312) ****
```

"***" marks the end of the first round trip. Note that cwnd did not increase (as evidenced by each ACK eliciting two new data packets). Only at "****", which comes near the end of the second round trip, does cwnd increase by one packet.

This trace did not suffer any timeout retransmissions. It transferred the same amount of data as the first trace in about half as much time. This difference is repeatable between hosts A and B.

References

[Stevens94] and [Wright95] discuss this problem. The problem of Reno TCP failing to recover from multiple losses except via a retransmission timeout is discussed in [Fall96,Hoe96].

How to detect
If source code is available, that is generally the easiest way to detect this problem. Search for each modification to the cwnd variable; (at least) one of these will be for congestion avoidance, and inspection of the related code should immediately identify the problem if present.

The problem can also be detected by closely examining packet traces taken near the sender. During congestion avoidance, cwnd will increase by an additional segment upon the receipt of (typically) eight acknowledgements without a loss. This increase is in addition to the one segment increase per round trip time (or two round trip times if the receiver is using delayed ACKs).

Furthermore, graphs of the sequence number vs. time, taken from packet traces, are normally linear during congestion avoidance. When viewing packet traces of transfers from senders exhibiting this problem, the graphs appear quadratic instead of linear.

Finally, the traces will show that, with sufficiently large windows, nearly every loss event results in a timeout.

How to fix
This problem may be corrected by removing the "+ MSS/8" term from the congestion avoidance code that increases cwnd each time an ACK of new data is received.

2.7.

Name of Problem
Initial RTO too low

Classification
Performance

Description
When a TCP first begins transmitting data, it lacks the RTT measurements necessary to have computed an adaptive retransmission timeout (RTO). RFC 1122, 4.2.3.1, states that a TCP SHOULD initialize RTO to 3 seconds. A TCP that uses a lower value exhibits "Initial RTO too low".

Significance
In environments with large RTTs (where "large" means any value larger than the initial RTO), TCPs will experience very poor performance.

Implications

Whenever RTO < RTT, very poor performance can result as packets are unnecessarily retransmitted (because RTO will expire before an ACK for the packet can arrive) and the connection enters slow start and congestion avoidance. Generally, the algorithms for computing RTO avoid this problem by adding a positive term to the estimated RTT. However, when a connection first begins it must use some estimate for RTO, and if it picks a value less than RTT, the above problems will arise.

Furthermore, when the initial RTO < RTT, it can take a long time for the TCP to correct the problem by adapting the RTT estimate, because the use of Karn's algorithm (mandated by RFC 1122, 4.2.3.1) will discard many of the candidate RTT measurements made after the first timeout, since they will be measurements of retransmitted segments.

Relevant RFCs

RFC 1122 states that TCPs SHOULD initialize RTO to 3 seconds and MUST implement Karn's algorithm.

Trace file demonstrating it

The following trace file was taken using tcpdump at host A, the data sender. The advertised window and SYN options have been omitted for clarity.

```
07:52:39.870301 A > B: S 2786333696:2786333696(0)
07:52:40.548170 B > A: S 130240000:130240000(0) ack 2786333697
07:52:40.561287 A > B: P 1:513(512) ack 1
07:52:40.753466 A > B: . 1:513(512) ack 1
07:52:41.133687 A > B: . 1:513(512) ack 1
07:52:41.458529 B > A: . ack 513
07:52:41.458686 A > B: . 513:1025(512) ack 1
07:52:41.458797 A > B: P 1025:1537(512) ack 1
07:52:41.541633 B > A: . ack 513
07:52:41.703732 A > B: . 513:1025(512) ack 1
07:52:42.044875 B > A: . ack 513
07:52:42.173728 A > B: . 513:1025(512) ack 1
07:52:42.330861 B > A: . ack 1537
07:52:42.331129 A > B: . 1537:2049(512) ack 1
07:52:42.331262 A > B: P 2049:2561(512) ack 1
07:52:42.623673 A > B: . 1537:2049(512) ack 1
07:52:42.683203 B > A: . ack 1537
07:52:43.044029 B > A: . ack 1537
07:52:43.193812 A > B: . 1537:2049(512) ack 1
```

Note from the SYN/SYN-ACK exchange, the RTT is over 600 msec. However, from the elapsed time between the third and fourth lines (the first packet being sent and then retransmitted), it is apparent the RTO was initialized to under 200 msec. The next line shows that this value has doubled to 400 msec (correct exponential backoff of RTO), but that still does not suffice to avoid an unnecessary retransmission.

Finally, an ACK from B arrives for the first segment. Later two more duplicate ACKs for 513 arrive, indicating that both the original and the two retransmissions arrived at B. (Indeed, a concurrent trace at B showed that no packets were lost during the entire connection). This ACK opens the congestion window to two packets, which are sent back-to-back, but at 07:52:41.703732 RTO again expires after a little over 200 msec, leading to an unnecessary retransmission, and the pattern repeats. By the end of the trace excerpt above, 1536 bytes have been successfully transmitted from A to B, over an interval of more than 2 seconds, reflecting terrible performance.

Trace file demonstrating correct behavior

The following trace file was taken using tcpdump at host C, the data sender. The advertised window and SYN options have been omitted for clarity.

```
17:30:32.090299 C > D: S 2031744000:2031744000(0)
17:30:32.900325 D > C: S 262737964:262737964(0) ack 2031744001
17:30:32.900326 C > D: . ack 1
17:30:32.910326 C > D: . 1:513(512) ack 1
17:30:34.150355 D > C: . ack 513
17:30:34.150356 C > D: . 513:1025(512) ack 1
17:30:34.150357 C > D: . 1025:1537(512) ack 1
17:30:35.170384 D > C: . ack 1025
17:30:35.170385 C > D: . 1537:2049(512) ack 1
17:30:35.170386 C > D: . 2049:2561(512) ack 1
17:30:35.320385 D > C: . ack 1537
17:30:35.320386 C > D: . 2561:3073(512) ack 1
17:30:35.320387 C > D: . 3073:3585(512) ack 1
17:30:35.730384 D > C: . ack 2049
```

The initial SYN/SYN-ACK exchange shows that RTT is more than 800 msec, and for some subsequent packets it rises above 1 second, but C’s retransmit timer does not ever expire.

References

This problem is documented in [Paxson97].

How to detect
This problem is readily detected by inspecting a packet trace of the startup of a TCP connection made over a long-delay path. It can be diagnosed from either a sender-side or receiver-side trace. Long-delay paths can often be found by locating remote sites on other continents.

How to fix
As this problem arises from a faulty initialization, one hopes fixing it requires a one-line change to the TCP source code.

2.8.

Name of Problem
Failure of window deflation after loss recovery

Classification
Congestion control / performance

Description
The fast recovery algorithm allows TCP senders to continue to transmit new segments during loss recovery. First, fast retransmission is initiated after a TCP sender receives three duplicate ACKs. At this point, a retransmission is sent and cwnd is halved. The fast recovery algorithm then allows additional segments to be sent when sufficient additional duplicate ACKs arrive. Some implementations of fast recovery compute when to send additional segments by artificially incrementing cwnd, first by three segments to account for the three duplicate ACKs that triggered fast retransmission, and subsequently by 1 MSS for each new duplicate ACK that arrives. When cwnd allows, the sender transmits new data segments.

When an ACK arrives that covers new data, cwnd is to be reduced by the amount by which it was artificially increased. However, some TCP implementations fail to "deflate" the window, causing an inappropriate amount of data to be sent into the network after recovery. One cause of this problem is the "header prediction" code, which is used to handle incoming segments that require little work. In some implementations of TCP, the header prediction code does not check to make sure cwnd has not been artificially inflated, and therefore does not reduce the artificially increased cwnd when appropriate.

Significance
TCP senders that exhibit this problem will transmit a burst of data immediately after recovery, which can degrade performance, as well as network stability. Effectively, the sender does not

reduce the size of cwnd as much as it should (to half its value when loss was detected), if at all. This can harm the performance of the TCP connection itself, as well as competing TCP flows.

Implications

A TCP sender exhibiting this problem does not reduce cwnd appropriately in times of congestion, and therefore may contribute to congestive collapse.

Relevant RFCs

RFC 2001 outlines the fast retransmit/fast recovery algorithms. [Brakmo95] outlines this implementation problem and offers a fix.

Trace file demonstrating it

The following trace file was taken using tcpdump at host A, the data sender. The advertised window (which never changed) has been omitted for clarity, except for the first packet sent by each host.

```
08:22:56.825635 A.7505 > B.7505: . 29697:30209(512) ack 1 win 4608
08:22:57.038794 B.7505 > A.7505: . ack 27649 win 4096
08:22:57.039279 A.7505 > B.7505: . 30209:30721(512) ack 1
08:22:57.321876 B.7505 > A.7505: . ack 28161
08:22:57.322356 A.7505 > B.7505: . 30721:31233(512) ack 1
08:22:57.347128 B.7505 > A.7505: . ack 28673
08:22:57.347572 A.7505 > B.7505: . 31233:31745(512) ack 1
08:22:57.347782 A.7505 > B.7505: . 31745:32257(512) ack 1
08:22:57.936393 B.7505 > A.7505: . ack 29185
08:22:57.936864 A.7505 > B.7505: . 32257:32769(512) ack 1
08:22:57.950802 B.7505 > A.7505: . ack 29697 win 4096
08:22:57.951246 A.7505 > B.7505: . 32769:33281(512) ack 1
08:22:58.169422 B.7505 > A.7505: . ack 29697
08:22:58.638222 B.7505 > A.7505: . ack 29697
08:22:58.643312 B.7505 > A.7505: . ack 29697
08:22:58.643669 A.7505 > B.7505: . 29697:30209(512) ack 1
08:22:58.936436 B.7505 > A.7505: . ack 29697
08:22:59.002614 B.7505 > A.7505: . ack 29697
08:22:59.003026 A.7505 > B.7505: . 33281:33793(512) ack 1
08:22:59.682902 B.7505 > A.7505: . ack 33281
08:22:59.683391 A.7505 > B.7505: P 33793:34305(512) ack 1
08:22:59.683748 A.7505 > B.7505: P 34305:34817(512) ack 1 ***
08:22:59.684043 A.7505 > B.7505: P 34817:35329(512) ack 1
08:22:59.684266 A.7505 > B.7505: P 35329:35841(512) ack 1
08:22:59.684567 A.7505 > B.7505: P 35841:36353(512) ack 1
08:22:59.684810 A.7505 > B.7505: P 36353:36865(512) ack 1
08:22:59.685094 A.7505 > B.7505: P 36865:37377(512) ack 1
```

The first 12 lines of the trace show incoming ACKs clocking out a window of data segments. At this point in the transfer, cwnd is 7 segments. The next 4 lines of the trace show 3 duplicate ACKs arriving from the receiver, followed by a retransmission from the sender. At this point, cwnd is halved (to 3 segments) and artificially incremented by the three duplicate ACKs that have arrived, making cwnd 6 segments. The next two lines show 2 more duplicate ACKs arriving, each of which increases cwnd by 1 segment. So, after these two duplicate ACKs arrive the cwnd is 8 segments and the sender has permission to send 1 new segment (since there are 7 segments outstanding). The next line in the trace shows this new segment being transmitted. The next packet shown in the trace is an ACK from host B that covers the first 7 outstanding segments (all but the new segment sent during recovery). This should cause cwnd to be reduced to 3 segments and 2 segments to be transmitted (since there is already 1 outstanding segment in the network). However, as shown by the last 7 lines of the trace, cwnd is not reduced, causing a line-rate burst of 7 new segments.

Trace file demonstrating correct behavior

The trace would appear identical to the one above, only it would stop after the line marked "***", because at this point host A would correctly reduce cwnd after recovery, allowing only 2 segments to be transmitted, rather than producing a burst of 7 segments.

References

This problem is documented and the performance implications analyzed in [Brakmo95].

How to detect

Failure of window deflation after loss recovery can be found by examining sender-side packet traces recorded during periods of moderate loss (so cwnd can grow large enough to allow for fast recovery when loss occurs).

How to fix

When this bug is caused by incorrect header prediction, the fix is to add a predicate to the header prediction test that checks to see whether cwnd is inflated; if so, the header prediction test fails and the usual ACK processing occurs, which (in this case) takes care to deflate the window. See [Brakmo95] for details.

2.9.

Name of Problem

Excessively short keepalive connection timeout

Classification
Reliability

Description
Keep-alive is a mechanism for checking whether an idle connection is still alive. According to RFC 1122, keepalive should only be invoked in server applications that might otherwise hang indefinitely and consume resources unnecessarily if a client crashes or aborts a connection during a network failure.

RFC 1122 also specifies that if a keep-alive mechanism is implemented it MUST NOT interpret failure to respond to any specific probe as a dead connection. The RFC does not specify a particular mechanism for timing out a connection when no response is received for keepalive probes. However, if the mechanism does not allow ample time for recovery from network congestion or delay, connections may be timed out unnecessarily.

Significance
In congested networks, can lead to unwarranted termination of connections.

Implications
It is possible for the network connection between two peer machines to become congested or to exhibit packet loss at the time that a keep-alive probe is sent on a connection. If the keep-alive mechanism does not allow sufficient time before dropping connections in the face of unacknowledged probes, connections may be dropped even when both peers of a connection are still alive.

Relevant RFCs
RFC 1122 specifies that the keep-alive mechanism may be provided. It does not specify a mechanism for determining dead connections when keepalive probes are not acknowledged.

Trace file demonstrating it
Made using the Orchestra tool at the peer of the machine using keep-alive. After connection establishment, incoming keep-alives were dropped by Orchestra to simulate a dead connection.

```
22:11:12.040000 A > B: 22666019:0 win 8192 datasz 4 SYN
22:11:12.060000 B > A: 2496001:22666020 win 4096 datasz 4 SYN ACK
22:11:12.130000 A > B: 22666020:2496002 win 8760 datasz 0 ACK
(more than two hours elapse)
00:23:00.680000 A > B: 22666019:2496002 win 8760 datasz 1 ACK
00:23:01.770000 A > B: 22666019:2496002 win 8760 datasz 1 ACK
00:23:02.870000 A > B: 22666019:2496002 win 8760 datasz 1 ACK
00:23.03.970000 A > B: 22666019:2496002 win 8760 datasz 1 ACK
```

```
00:23.05.070000 A > B: 22666019:2496002 win 8760 datasz 1 ACK
```

The initial three packets are the SYN exchange for connection setup. About two hours later, the keepalive timer fires because the connection has been idle. Keepalive probes are transmitted a total of 5 times, with a 1 second spacing between probes, after which the connection is dropped. This is problematic because a 5 second network outage at the time of the first probe results in the connection being killed.

Trace file demonstrating correct behavior

Made using the Orchestra tool at the peer of the machine using keep-alive. After connection establishment, incoming keep-alives were dropped by Orchestra to simulate a dead connection.

```
16:01:52.130000 A > B: 1804412929:0 win 4096 datasz 4 SYN
16:01:52.360000 B > A: 16512001:1804412930 win 4096 datasz 4 SYN ACK
16:01:52.410000 A > B: 1804412930:16512002 win 4096 datasz 0 ACK
(two hours elapse)
18:01:57.170000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:03:12.220000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:04:27.270000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:05:42.320000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:06:57.370000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:08:12.420000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:09:27.480000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:10:43.290000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:11:57.580000 A > B: 1804412929:16512002 win 4096 datasz 0 ACK
18:13:12.630000 A > B: 1804412929:16512002 win 4096 datasz 0 RST ACK
```

In this trace, when the keep-alive timer expires, 9 keepalive probes are sent at 75 second intervals. 75 seconds after the last probe is sent, a final RST segment is sent indicating that the connection has been closed. This implementation waits about 11 minutes before timing out the connection, while the first implementation shown allows only 5 seconds.

References

This problem is documented in [Dawson97].

How to detect

For implementations manifesting this problem, it shows up on a packet trace after the keepalive timer fires if the peer machine receiving the keepalive does not respond. Usually the keepalive timer will fire at least two hours after keepalive is turned on, but it may be sooner if the timer value has been configured lower, or if the keepalive mechanism violates the specification (see Insufficient interval between keepalives problem). In this

example, suppressing the response of the peer to keepalive probes was accomplished using the Orchestra toolkit, which can be configured to drop packets. It could also have been done by creating a connection, turning on keepalive, and disconnecting the network connection at the receiver machine.

How to fix
This problem can be fixed by using a different method for timing out keepalives that allows a longer period of time to elapse before dropping the connection. For example, the algorithm for timing out on dropped data could be used. Another possibility is an algorithm such as the one shown in the trace above, which sends 9 probes at 75 second intervals and then waits an additional 75 seconds for a response before closing the connection.

2.10.

Name of Problem
Failure to back off retransmission timeout

Classification
Congestion control / reliability

Description
The retransmission timeout is used to determine when a packet has been dropped in the network. When this timeout has expired without the arrival of an ACK, the segment is retransmitted. Each time a segment is retransmitted, the timeout is adjusted according to an exponential backoff algorithm, doubling each time. If a TCP fails to receive an ACK after numerous attempts at retransmitting the same segment, it terminates the connection. A TCP that fails to double its retransmission timeout upon repeated timeouts is said to exhibit "Failure to back off retransmission timeout".

Significance
Backing off the retransmission timer is a cornerstone of network stability in the presence of congestion. Consequently, this bug can have severe adverse affects in congested networks. It also affects TCP reliability in congested networks, as discussed in the next section.

Implications
It is possible for the network connection between two TCP peers to become congested or to exhibit packet loss at the time that a retransmission is sent on a connection. If the retransmission mechanism does not allow sufficient time before dropping

connections in the face of unacknowledged segments, connections may be dropped even when, by waiting longer, the connection could have continued.

Relevant RFCs

RFC 1122 specifies mandatory exponential backoff of the retransmission timeout, and the termination of connections after some period of time (at least 100 seconds).

Trace file demonstrating it

Made using tcpdump on an intermediate host:

```
16:51:12.671727 A > B: S 510878852:510878852(0) win 16384
16:51:12.672479 B > A: S 2392143687:2392143687(0)
                         ack 510878853 win 16384
16:51:12.672581 A > B: . ack 1 win 16384
16:51:15.244171 A > B: P 1:3(2) ack 1 win 16384
16:51:15.244933 B > A: . ack 3 win 17518  (DF)

<receiving host disconnected>

16:51:19.381176 A > B: P 3:5(2) ack 1 win 16384
16:51:20.162016 A > B: P 3:5(2) ack 1 win 16384
16:51:21.161936 A > B: P 3:5(2) ack 1 win 16384
16:51:22.161914 A > B: P 3:5(2) ack 1 win 16384
16:51:23.161914 A > B: P 3:5(2) ack 1 win 16384
16:51:24.161879 A > B: P 3:5(2) ack 1 win 16384
16:51:25.161857 A > B: P 3:5(2) ack 1 win 16384
16:51:26.161836 A > B: P 3:5(2) ack 1 win 16384
16:51:27.161814 A > B: P 3:5(2) ack 1 win 16384
16:51:28.161791 A > B: P 3:5(2) ack 1 win 16384
16:51:29.161769 A > B: P 3:5(2) ack 1 win 16384
16:51:30.161750 A > B: P 3:5(2) ack 1 win 16384
16:51:31.161727 A > B: P 3:5(2) ack 1 win 16384

16:51:32.161701 A > B: R 5:5(0) ack 1 win 16384
```

The initial three packets are the SYN exchange for connection setup, then a single data packet, to verify that data can be transferred. Then the connection to the destination host was disconnected, and more data sent. Retransmissions occur every second for 12 seconds, and then the connection is terminated with a RST. This is problematic because a 12 second pause in connectivity could result in the termination of a connection.

Trace file demonstrating correct behavior

Again, a tcpdump taken from a third host:

```
16:59:05.398301 A > B: S 2503324757:2503324757(0) win 16384
16:59:05.399673 B > A: S 2492674648:2492674648(0)
                        ack 2503324758 win 16384
16:59:05.399866 A > B: . ack 1 win 17520
16:59:06.538107 A > B: P 1:3(2) ack 1 win 17520
16:59:06.540977 B > A: . ack 3 win 17518  (DF)

<receiving host disconnected>

16:59:13.121542 A > B: P 3:5(2) ack 1 win 17520
16:59:14.010928 A > B: P 3:5(2) ack 1 win 17520
16:59:16.010979 A > B: P 3:5(2) ack 1 win 17520
16:59:20.011229 A > B: P 3:5(2) ack 1 win 17520
16:59:28.011896 A > B: P 3:5(2) ack 1 win 17520
16:59:44.013200 A > B: P 3:5(2) ack 1 win 17520
17:00:16.015766 A > B: P 3:5(2) ack 1 win 17520
17:01:20.021308 A > B: P 3:5(2) ack 1 win 17520
17:02:24.027752 A > B: P 3:5(2) ack 1 win 17520
17:03:28.034569 A > B: P 3:5(2) ack 1 win 17520
17:04:32.041567 A > B: P 3:5(2) ack 1 win 17520
17:05:36.048264 A > B: P 3:5(2) ack 1 win 17520
17:06:40.054900 A > B: P 3:5(2) ack 1 win 17520

17:07:44.061306 A > B: R 5:5(0) ack 1 win 17520
```

In this trace, when the retransmission timer expires, 12 retransmissions are sent at exponentially-increasing intervals, until the interval value reaches 64 seconds, at which time the interval stops growing. 64 seconds after the last retransmission, a final RST segment is sent indicating that the connection has been closed. This implementation waits about 9 minutes before timing out the connection, while the first implementation shown allows only 12 seconds.

References

None known.

How to detect

A simple transfer can be easily interrupted by disconnecting the receiving host from the network. tcpdump or another appropriate tool should show the retransmissions being sent. Several trials in a low-rtt environment may be required to demonstrate the bug.

How to fix

For one of the implementations studied, this problem seemed to be the result of an error introduced with the addition of the Brakmo-Peterson RTO algorithm [Brakmo95], which can return a value of zero where the older Jacobson algorithm always returns a

positive value. Brakmo and Peterson specified an additional step of min(rtt + 2, RTO) to avoid problems with this. Unfortunately, in the implementation this step was omitted when calculating the exponential backoff for the RTO. This results in an RTO of 0 seconds being multiplied by the backoff, yielding again zero, and then being subjected to a later MAX operation that increases it to 1 second, regardless of the backoff factor.

A similar TCP persist failure has the same cause.

2.11.

Name of Problem
Insufficient interval between keepalives

Classification
Reliability

Description
Keep-alive is a mechanism for checking whether an idle connection is still alive. According to RFC 1122, keep-alive may be included in an implementation. If it is included, the interval between keep-alive packets MUST be configurable, and MUST default to no less than two hours.

Significance
In congested networks, can lead to unwarranted termination of connections.

Implications
According to RFC 1122, keep-alive is not required of implementations because it could: (1) cause perfectly good connections to break during transient Internet failures; (2) consume unnecessary bandwidth ("if no one is using the connection, who cares if it is still good?"); and (3) cost money for an Internet path that charges for packets. Regarding this last point, we note that in addition the presence of dial-on-demand links in the route can greatly magnify the cost penalty of excess keepalives, potentially forcing a full-time connection on a link that would otherwise only be connected a few minutes a day.

If keepalive is provided the RFC states that the required inter-keepalive distance MUST default to no less than two hours. If it does not, the probability of connections breaking increases, the bandwidth used due to keepalives increases, and cost increases over paths which charge per packet.

Relevant RFCs

RFC 1122 specifies that the keep-alive mechanism may be provided. It also specifies the two hour minimum for the default interval between keepalive probes.

Trace file demonstrating it

Made using the Orchestra tool at the peer of the machine using keep-alive. Machine A was configured to use default settings for the keepalive timer.

```
11:36:32.910000 A > B: 3288354305:0      win 28672 datasz 4 SYN
11:36:32.930000 B > A: 896001:3288354306 win 4096  datasz 4 SYN ACK
11:36:32.950000 A > B: 3288354306:896002 win 28672 datasz 0 ACK

11:50:01.190000 A > B: 3288354305:896002 win 28672 datasz 0 ACK
11:50:01.210000 B > A: 896002:3288354306 win 4096  datasz 0 ACK

12:03:29.410000 A > B: 3288354305:896002 win 28672 datasz 0 ACK
12:03:29.430000 B > A: 896002:3288354306 win 4096  datasz 0 ACK

12:16:57.630000 A > B: 3288354305:896002 win 28672 datasz 0 ACK
12:16:57.650000 B > A: 896002:3288354306 win 4096  datasz 0 ACK

12:30:25.850000 A > B: 3288354305:896002 win 28672 datasz 0 ACK
12:30:25.870000 B > A: 896002:3288354306 win 4096  datasz 0 ACK

12:43:54.070000 A > B: 3288354305:896002 win 28672 datasz 0 ACK
12:43:54.090000 B > A: 896002:3288354306 win 4096  datasz 0 ACK
```

The initial three packets are the SYN exchange for connection setup. About 13 minutes later, the keepalive timer fires because the connection is idle. The keepalive is acknowledged, and the timer fires again in about 13 more minutes. This behavior continues indefinitely until the connection is closed, and is a violation of the specification.

Trace file demonstrating correct behavior

Made using the Orchestra tool at the peer of the machine using keep-alive. Machine A was configured to use default settings for the keepalive timer.

```
17:37:20.500000 A > B: 34155521:0       win 4096 datasz 4 SYN
17:37:20.520000 B > A: 6272001:34155522 win 4096 datasz 4 SYN ACK
17:37:20.540000 A > B: 34155522:6272002 win 4096 datasz 0 ACK

19:37:25.430000 A > B: 34155521:6272002 win 4096 datasz 0 ACK
19:37:25.450000 B > A: 6272002:34155522 win 4096 datasz 0 ACK
```

```
21:37:30.560000 A > B: 34155521:6272002 win 4096 datasz 0 ACK
21:37:30.570000 B > A: 6272002:34155522 win 4096 datasz 0 ACK

23:37:35.580000 A > B: 34155521:6272002 win 4096 datasz 0 ACK
23:37:35.600000 B > A: 6272002:34155522 win 4096 datasz 0 ACK

01:37:40.620000 A > B: 34155521:6272002 win 4096 datasz 0 ACK
01:37:40.640000 B > A: 6272002:34155522 win 4096 datasz 0 ACK

03:37:45.590000 A > B: 34155521:6272002 win 4096 datasz 0 ACK
03:37:45.610000 B > A: 6272002:34155522 win 4096 datasz 0 ACK
```

The initial three packets are the SYN exchange for connection setup. Just over two hours later, the keepalive timer fires because the connection is idle. The keepalive is acknowledged, and the timer fires again just over two hours later. This behavior continues indefinitely until the connection is closed.

References
This problem is documented in [Dawson97].

How to detect
For implementations manifesting this problem, it shows up on a packet trace. If the connection is left idle, the keepalive probes will arrive closer together than the two hour minimum.

2.12.

Name of Problem
Window probe deadlock

Classification
Reliability

Description
When an application reads a single byte from a full window, the window should not be updated, in order to avoid Silly Window Syndrome (SWS; see [RFC813]). If the remote peer uses a single byte of data to probe the window, that byte can be accepted into the buffer. In some implementations, at this point a negative argument to a signed comparison causes all further new data to be considered outside the window; consequently, it is discarded (after sending an ACK to resynchronize). These discards include the ACKs for the data packets sent by the local TCP, so the TCP will consider the data unacknowledged.

Consequently, the application may be unable to complete sending new data to the remote peer, because it has exhausted the transmit buffer available to its local TCP, and buffer space is never being freed because incoming ACKs that would do so are being discarded. If the application does not read any more data, which may happen due to its failure to complete such sends, then deadlock results.

Significance

It's relatively rare for applications to use TCP in a manner that can exercise this problem. Most applications only transmit bulk data if they know the other end is prepared to receive the data. However, if a client fails to consume data, putting the server in persist mode, and then consumes a small amount of data, it can mistakenly compute a negative window. At this point the client will discard all further packets from the server, including ACKs of the client's own data, since they are not inside the (impossibly-sized) window. If subsequently the client consumes enough data to then send a window update to the server, the situation will be rectified. That is, this situation can only happen if the client consumes 1 < N < MSS bytes, so as not to cause a window update, and then starts its own transmission towards the server of more than a window's worth of data.

Implications

TCP connections will hang and eventually time out.

Relevant RFCs

RFC 793 describes zero window probing. RFC 813 describes Silly Window Syndrome.

Trace file demonstrating it

Trace made from a version of tcpdump modified to print out the sequence number attached to an ACK even if it's dataless. An unmodified tcpdump would not print seq:seq(0); however, for this bug, the sequence number in the ACK is important for unambiguously determining how the TCP is behaving.

```
[ Normal connection startup and data transmission from B to A.
  Options, including MSS of 16344 in both directions, omitted
  for clarity. ]
16:07:32.327616 A > B: S 65360807:65360807(0) win 8192
16:07:32.327304 B > A: S 65488807:65488807(0) ack 65360808 win 57344
16:07:32.327425 A > B: . 1:1(0) ack 1 win 57344
16:07:32.345732 B > A: P 1:2049(2048) ack 1 win 57344
16:07:32.347013 B > A: P 2049:16385(14336) ack 1 win 57344
16:07:32.347550 B > A: P 16385:30721(14336) ack 1 win 57344
16:07:32.348683 B > A: P 30721:45057(14336) ack 1 win 57344
16:07:32.467286 A > B: . 1:1(0) ack 45057 win 12288
```

```
16:07:32.467854 B > A: P 45057:57345(12288) ack 1 win 57344

[ B fills up A's offered window ]
16:07:32.667276 A > B: . 1:1(0) ack 57345 win 0

[ B probes A's window with a single byte ]
16:07:37.467438 B > A: . 57345:57346(1) ack 1 win 57344

[ A resynchronizes without accepting the byte ]
16:07:37.467678 A > B: . 1:1(0) ack 57345 win 0

[ B probes A's window again ]
16:07:45.467438 B > A: . 57345:57346(1) ack 1 win 57344

[ A resynchronizes and accepts the byte (per the ack field) ]
16:07:45.667250 A > B: . 1:1(0) ack 57346 win 0

[ The application on A has started generating data.  The first
  packet A sends is small due to a memory allocation bug. ]
16:07:51.358459 A > B: P 1:2049(2048) ack 57346 win 0

[ B acks A's first packet ]
16:07:51.467239 B > A: . 57346:57346(0) ack 2049 win 57344

[ This looks as though A accepted B's ACK and is sending
  another packet in response to it.  In fact, A is trying
  to resynchronize with B, and happens to have data to send
  and can send it because the first small packet didn't use
  up cwnd. ]
16:07:51.467698 A > B: . 2049:14337(12288) ack 57346 win 0

[ B acks all of the data that A has sent ]
16:07:51.667283 B > A: . 57346:57346(0) ack 14337 win 57344

[ A tries to resynchronize.  Notice that by the packets
  seen on the network, A and B *are* in fact synchronized;
  A only thinks that they aren't. ]
16:07:51.667477 A > B: . 14337:14337(0) ack 57346 win 0

[ A's retransmit timer fires, and B acks all of the data.
  A once again tries to resynchronize. ]
16:07:52.467682 A > B: . 1:14337(14336) ack 57346 win 0
16:07:52.468166 B > A: . 57346:57346(0) ack 14337 win 57344
16:07:52.468248 A > B: . 14337:14337(0) ack 57346 win 0

[ A's retransmit timer fires again, and B acks all of the data.
  A once again tries to resynchronize. ]
16:07:55.467684 A > B: . 1:14337(14336) ack 57346 win 0
```

```
16:07:55.468172 B > A: . 57346:57346(0) ack 14337 win 57344
16:07:55.468254 A > B: . 14337:14337(0) ack 57346 win 0

Trace file demonstrating correct behavior
   Made between the same two hosts after applying the bug fix
   mentioned below (and using the same modified tcpdump).

[ Connection starts up with data transmission from B to A.
  Note that due to a separate bug (the fact that A and B
  are communicating over a loopback driver), B erroneously
  skips slow start. ]
17:38:09.510854 A > B: S 3110066585:3110066585(0) win 16384
17:38:09.510926 B > A: S 3110174850:3110174850(0)
                         ack 3110066586 win 57344
17:38:09.510953 A > B: . 1:1(0) ack 1 win 57344
17:38:09.512956 B > A: P 1:2049(2048) ack 1 win 57344
17:38:09.513222 B > A: P 2049:16385(14336) ack 1 win 57344
17:38:09.513428 B > A: P 16385:30721(14336) ack 1 win 57344
17:38:09.513638 B > A: P 30721:45057(14336) ack 1 win 57344
17:38:09.519531 A > B: . 1:1(0) ack 45057 win 12288
17:38:09.519638 B > A: P 45057:57345(12288) ack 1 win 57344

[ B fills up A's offered window ]
17:38:09.719526 A > B: . 1:1(0) ack 57345 win 0

[ B probes A's window with a single byte.  A resynchronizes
  without accepting the byte ]
17:38:14.499661 B > A: . 57345:57346(1) ack 1 win 57344
17:38:14.499724 A > B: . 1:1(0) ack 57345 win 0

[ B probes A's window again.  A resynchronizes and accepts
  the byte, as indicated by the ack field ]
17:38:19.499764 B > A: . 57345:57346(1) ack 1 win 57344
17:38:19.519731 A > B: . 1:1(0) ack 57346 win 0

[ B probes A's window with a single byte.  A resynchronizes
  without accepting the byte ]
17:38:24.499865 B > A: . 57346:57347(1) ack 1 win 57344
17:38:24.499934 A > B: . 1:1(0) ack 57346 win 0

[ The application on A has started generating data.
  B acks A's data and A accepts the ACKs and the
  data transfer continues ]
17:38:28.530265 A > B: P 1:2049(2048) ack 57346 win 0
17:38:28.719914 B > A: . 57346:57346(0) ack 2049 win 57344

17:38:28.720023 A > B: . 2049:16385(14336) ack 57346 win 0
17:38:28.720089 A > B: . 16385:30721(14336) ack 57346 win 0
```

```
17:38:28.720370 B > A: . 57346:57346(0) ack 30721 win 57344

17:38:28.720462 A > B: . 30721:45057(14336) ack 57346 win 0
17:38:28.720526 A > B: P 45057:59393(14336) ack 57346 win 0
17:38:28.720824 A > B: P 59393:73729(14336) ack 57346 win 0
17:38:28.721124 B > A: . 57346:57346(0) ack 73729 win 47104

17:38:28.721198 A > B: P 73729:88065(14336) ack 57346 win 0
17:38:28.721379 A > B: P 88065:102401(14336) ack 57346 win 0

17:38:28.721557 A > B: P 102401:116737(14336) ack 57346 win 0
17:38:28.721863 B > A: . 57346:57346(0) ack 116737 win 36864
```

References

None known.

How to detect

Initiate a connection from a client to a server. Have the server continuously send data until its buffers have been full for long enough to exhaust the window. Next, have the client read 1 byte and then delay for long enough that the server TCP sends a window probe. Now have the client start sending data. At this point, if it ignores the server's ACKs, then the client's TCP suffers from the problem.

How to fix

In one implementation known to exhibit the problem (derived from 4.3-Reno), the problem was introduced when the macro MAX() was replaced by the function call max() for computing the amount of space in the receive window:

```
tp->rcv_wnd = max(win, (int)(tp->rcv_adv - tp->rcv_nxt));
```

When data has been received into a window beyond what has been advertised to the other side, rcv_nxt > rcv_adv, making this negative. It's clear from the (int) cast that this is intended, but the unsigned max() function sign-extends so the negative number is "larger". The fix is to change max() to imax():

```
tp->rcv_wnd = imax(win, (int)(tp->rcv_adv - tp->rcv_nxt));
```

4.3-Tahoe and before did not have this bug, since it used the macro MAX() for this calculation.

2.13.

Name of Problem

Stretch ACK violation

Classification
Congestion Control/Performance

Description
To improve efficiency (both computer and network) a data receiver may refrain from sending an ACK for each incoming segment, according to [RFC1122]. However, an ACK should not be delayed an inordinate amount of time. Specifically, ACKs SHOULD be sent for every second full-sized segment that arrives. If a second full-sized segment does not arrive within a given timeout (of no more than 0.5 seconds), an ACK should be transmitted, according to [RFC1122]. A TCP receiver which does not generate an ACK for every second full-sized segment exhibits a "Stretch ACK Violation".

Significance
TCP receivers exhibiting this behavior will cause TCP senders to generate burstier traffic, which can degrade performance in congested environments. In addition, generating fewer ACKs increases the amount of time needed by the slow start algorithm to open the congestion window to an appropriate point, which diminishes performance in environments with large bandwidth-delay products. Finally, generating fewer ACKs may cause needless retransmission timeouts in lossy environments, as it increases the possibility that an entire window of ACKs is lost, forcing a retransmission timeout.

Implications
When not in loss recovery, every ACK received by a TCP sender triggers the transmission of new data segments. The burst size is determined by the number of previously unacknowledged segments each ACK covers. Therefore, a TCP receiver ack'ing more than 2 segments at a time causes the sending TCP to generate a larger burst of traffic upon receipt of the ACK. This large burst of traffic can overwhelm an intervening gateway, leading to higher drop rates for both the connection and other connections passing through the congested gateway.

In addition, the TCP slow start algorithm increases the congestion window by 1 segment for each ACK received. Therefore, increasing the ACK interval (thus decreasing the rate at which ACKs are transmitted) increases the amount of time it takes slow start to increase the congestion window to an appropriate operating point, and the connection consequently suffers from reduced performance. This is especially true for connections using large windows.

Relevant RFCs
RFC 1122 outlines delayed ACKs as a recommended mechanism.

Trace file demonstrating it

Trace file taken using tcpdump at host B, the data receiver (and ACK originator). The advertised window (which never changed) and timestamp options have been omitted for clarity, except for the first packet sent by A:

```
12:09:24.820187 A.1174 > B.3999: . 2049:3497(1448) ack 1
    win 33580 <nop,nop,timestamp 2249877 2249914> [tos 0x8]
12:09:24.824147 A.1174 > B.3999: . 3497:4945(1448) ack 1
12:09:24.832034 A.1174 > B.3999: . 4945:6393(1448) ack 1
12:09:24.832222 B.3999 > A.1174: . ack 6393
12:09:24.934837 A.1174 > B.3999: . 6393:7841(1448) ack 1
12:09:24.942721 A.1174 > B.3999: . 7841:9289(1448) ack 1
12:09:24.950605 A.1174 > B.3999: . 9289:10737(1448) ack 1
12:09:24.950797 B.3999 > A.1174: . ack 10737
12:09:24.958488 A.1174 > B.3999: . 10737:12185(1448) ack 1
12:09:25.052330 A.1174 > B.3999: . 12185:13633(1448) ack 1
12:09:25.060216 A.1174 > B.3999: . 13633:15081(1448) ack 1
12:09:25.060405 B.3999 > A.1174: . ack 15081
```

This portion of the trace clearly shows that the receiver (host B) sends an ACK for every third full sized packet received. Further investigation of this implementation found that the cause of the increased ACK interval was the TCP options being used. The implementation sent an ACK after it was holding 2*MSS worth of unacknowledged data. In the above case, the MSS is 1460 bytes so the receiver transmits an ACK after it is holding at least 2920 bytes of unacknowledged data. However, the length of the TCP options being used [RFC1323] took 12 bytes away from the data portion of each packet. This produced packets containing 1448 bytes of data. But the additional bytes used by the options in the header were not taken into account when determining when to trigger an ACK. Therefore, it took 3 data segments before the data receiver was holding enough unacknowledged data (>= 2*MSS, or 2920 bytes in the above example) to transmit an ACK.

Trace file demonstrating correct behavior

Trace file taken using tcpdump at host B, the data receiver (and ACK originator), again with window and timestamp information omitted except for the first packet:

```
12:06:53.627320 A.1172 > B.3999: . 1449:2897(1448) ack 1
    win 33580 <nop,nop,timestamp 2249575 2249612> [tos 0x8]
12:06:53.634773 A.1172 > B.3999: . 2897:4345(1448) ack 1
12:06:53.634961 B.3999 > A.1172: . ack 4345
12:06:53.737326 A.1172 > B.3999: . 4345:5793(1448) ack 1
12:06:53.744401 A.1172 > B.3999: . 5793:7241(1448) ack 1
12:06:53.744592 B.3999 > A.1172: . ack 7241
```

```
12:06:53.752287 A.1172 > B.3999: . 7241:8689(1448) ack 1
12:06:53.847332 A.1172 > B.3999: . 8689:10137(1448) ack 1
12:06:53.847525 B.3999 > A.1172: . ack 10137
```

This trace shows the TCP receiver (host B) ack'ing every second full-sized packet, according to [RFC1122]. This is the same implementation shown above, with slight modifications that allow the receiver to take the length of the options into account when deciding when to transmit an ACK.

References
This problem is documented in [Allman97] and [Paxson97].

How to detect
Stretch ACK violations show up immediately in receiver-side packet traces of bulk transfers, as shown above. However, packet traces made on the sender side of the TCP connection may lead to ambiguities when diagnosing this problem due to the possibility of lost ACKs.

2.14.

Name of Problem
Retransmission sends multiple packets

Classification
Congestion control

Description
When a TCP retransmits a segment due to a timeout expiration or beginning a fast retransmission sequence, it should only transmit a single segment. A TCP that transmits more than one segment exhibits "Retransmission Sends Multiple Packets".

Instances of this problem have been known to occur due to miscomputations involving the use of TCP options. TCP options increase the TCP header beyond its usual size of 20 bytes. The total size of header must be taken into account when retransmitting a packet. If a TCP sender does not account for the length of the TCP options when determining how much data to retransmit, it will send too much data to fit into a single packet. In this case, the correct retransmission will be followed by a short segment (tinygram) containing data that may not need to be retransmitted.

A specific case is a TCP using the RFC 1323 timestamp option, which adds 12 bytes to the standard 20-byte TCP header. On retransmission of a packet, the 12 byte option is incorrectly

interpreted as part of the data portion of the segment. A standard TCP header and a new 12-byte option is added to the data, which yields a transmission of 12 bytes more data than contained in the original segment. This overflow causes a smaller packet, with 12 data bytes, to be transmitted.

Significance

This problem is somewhat serious for congested environments because the TCP implementation injects more packets into the network than is appropriate. However, since a tinygram is only sent in response to a fast retransmit or a timeout, it does not effect the sustained sending rate.

Implications

A TCP exhibiting this behavior is stressing the network with more traffic than appropriate, and stressing routers by increasing the number of packets they must process. The redundant tinygram will also elicit a duplicate ACK from the receiver, resulting in yet another unnecessary transmission.

Relevant RFCs

RFC 1122 requires use of slow start after loss; RFC 2001 explicates slow start; RFC 1323 describes the timestamp option that has been observed to lead to some implementations exhibiting this problem.

Trace file demonstrating it

Made using tcpdump recording at a machine on the same subnet as Host A. Host A is the sender and Host B is the receiver. The advertised window and timestamp options have been omitted for clarity, except for the first segment sent by host A. In addition, portions of the trace file not pertaining to the packet in question have been removed (missing packets are denoted by "[...]" in the trace).

```
11:55:22.701668 A > B: . 7361:7821(460) ack 1
    win 49324 <nop,nop,timestamp 3485348 3485113>
11:55:22.702109 A > B: . 7821:8281(460) ack 1
[...]

11:55:23.112405 B > A: . ack 7821
11:55:23.113069 A > B: . 12421:12881(460) ack 1
11:55:23.113511 A > B: . 12881:13341(460) ack 1
11:55:23.333077 B > A: . ack 7821
11:55:23.336860 B > A: . ack 7821
11:55:23.340638 B > A: . ack 7821
11:55:23.341290 A > B: . 7821:8281(460) ack 1
11:55:23.341317 A > B: . 8281:8293(12) ack 1
```

```
11:55:23.498242 B > A: . ack 7821
11:55:23.506850 B > A: . ack 7821
11:55:23.510630 B > A: . ack 7821

[...]

11:55:23.746649 B > A: . ack 10581
```

The second line of the above trace shows the original transmission of a segment which is later dropped. After 3 duplicate ACKs, line 9 of the trace shows the dropped packet (7821:8281), with a 460-byte payload, being retransmitted. Immediately following this retransmission, a packet with a 12-byte payload is unnecessarily sent.

Trace file demonstrating correct behavior

The trace file would be identical to the one above, with a single line:

```
11:55:23.341317 A > B: . 8281:8293(12) ack 1
```

omitted.

References

[Brakmo95]

How to detect

This problem can be detected by examining a packet trace of the TCP connections of a machine using TCP options, during which a packet is retransmitted.

2.15.

Name of Problem

Failure to send FIN notification promptly

Classification

Performance

Description

When an application closes a connection, the corresponding TCP should send the FIN notification promptly to its peer (unless prevented by the congestion window). If a TCP implementation delays in sending the FIN notification, for example due to waiting until unacknowledged data has been acknowledged, then it is said to exhibit "Failure to send FIN notification promptly".

Also, while not strictly required, FIN segments should include the PSH flag to ensure expedited delivery of any pending data at the receiver.

Significance
The greatest impact occurs for short-lived connections, since for these the additional time required to close the connection introduces the greatest relative delay.

The additional time can be significant in the common case of the sender waiting for an ACK that is delayed by the receiver.

Implications
Can diminish total throughput as seen at the application layer, because connection termination takes longer to complete.

Relevant RFCs
RFC 793 indicates that a receiver should treat an incoming FIN flag as implying the push function.

Trace file demonstrating it
Made using tcpdump (no losses reported by the packet filter).

```
10:04:38.68 A > B: S 1031850376:1031850376(0) win 4096
                   <mss 1460,wscale 0,eol> (DF)
10:04:38.71 B > A: S 596916473:596916473(0) ack 1031850377
                   win 8760 <mss 1460> (DF)
10:04:38.73 A > B: . ack 1 win 4096 (DF)
10:04:41.98 A > B: P 1:4(3) ack 1 win 4096 (DF)
10:04:42.15 B > A: . ack 4 win 8757 (DF)
10:04:42.23 A > B: P 4:7(3) ack 1 win 4096 (DF)
10:04:42.25 B > A: P 1:11(10) ack 7 win 8754 (DF)
10:04:42.32 A > B: . ack 11 win 4096 (DF)
10:04:42.33 B > A: P 11:51(40) ack 7 win 8754 (DF)
10:04:42.51 A > B: . ack 51 win 4096 (DF)
10:04:42.53 B > A: F 51:51(0) ack 7 win 8754 (DF)
10:04:42.56 A > B: FP 7:7(0) ack 52 win 4096 (DF)
10:04:42.58 B > A: . ack 8 win 8754 (DF)
```

Machine B in the trace above does not send out a FIN notification promptly if there is any data outstanding. It instead waits for all unacknowledged data to be acknowledged before sending the FIN segment. The connection was closed at 10:04.42.33 after requesting 40 bytes to be sent. However, the FIN notification isn't sent until 10:04.42.51, after the (delayed) acknowledgement of the 40 bytes of data.

Trace file demonstrating correct behavior
Made using tcpdump (no losses reported by the packet filter).

```
10:27:53.85 C > D: S 419744533:419744533(0) win 4096
                   <mss 1460,wscale 0,eol> (DF)
10:27:53.92 D > C: S 10082297:10082297(0) ack 419744534
                   win 8760 <mss 1460> (DF)
10:27:53.95 C > D: . ack 1 win 4096 (DF)
10:27:54.42 C > D: P 1:4(3) ack 1 win 4096 (DF)
10:27:54.62 D > C: . ack 4 win 8757 (DF)
10:27:54.76 C > D: P 4:7(3) ack 1 win 4096 (DF)
10:27:54.89 D > C: P 1:11(10) ack 7 win 8754 (DF)
10:27:54.90 D > C: FP 11:51(40) ack7 win 8754 (DF)
10:27:54.92 C > D: . ack 52 win 4096 (DF)
10:27:55.01 C > D: FP 7:7(0) ack 52 win 4096 (DF)
10:27:55.09 D > C: . ack 8 win 8754 (DF)
```

Here, Machine D sends a FIN with 40 bytes of data even before the original 10 octets have been acknowledged. This is correct behavior as it provides for the highest performance.

References
This problem is documented in [Dawson97].

How to detect
For implementations manifesting this problem, it shows up on a packet trace.

2.16.

Name of Problem
Failure to send a RST after Half Duplex Close

Classification
Resource management

Description
RFC 1122 4.2.2.13 states that a TCP SHOULD send a RST if data is received after "half duplex close", i.e. if it cannot be delivered to the application. A TCP that fails to do so is said to exhibit "Failure to send a RST after Half Duplex Close".

Significance
Potentially serious for TCP endpoints that manage large numbers of connections, due to exhaustion of memory and/or process slots available for managing connection state.

Implications

Failure to send the RST can lead to permanently hung TCP connections. This problem has been demonstrated when HTTP clients abort connections, common when users move on to a new page before the current page has finished downloading. The HTTP client closes by transmitting a FIN while the server is transmitting images, text, etc. The server TCP receives the FIN, but its application does not close the connection until all data has been queued for transmission. Since the server will not transmit a FIN until all the preceding data has been transmitted, deadlock results if the client TCP does not consume the pending data or tear down the connection: the window decreases to zero, since the client cannot pass the data to the application, and the server sends probe segments. The client acknowledges the probe segments with a zero window. As mandated in RFC1122 4.2.2.17, the probe segments are transmitted forever. Server connection state remains in CLOSE_WAIT, and eventually server processes are exhausted.

Note that there are two bugs. First, probe segments should be ignored if the window can never subsequently increase. Second, a RST should be sent when data is received after half duplex close. Fixing the first bug, but not the second, results in the probe segments eventually timing out the connection, but the server remains in CLOSE_WAIT for a significant and unnecessary period.

Relevant RFCs

RFC 1122 sections 4.2.2.13 and 4.2.2.17.

Trace file demonstrating it

Made using an unknown network analyzer. No drop information available.

```
client.1391 > server.8080: S 0:1(0) ack: 0 win: 2000 <mss: 5b4>
server.8080 > client.1391: SA 8c01:8c02(0) ack: 1 win: 8000 <mss:100>
client.1391 > server.8080: PA
client.1391 > server.8080: PA 1:1c2(1c1) ack: 8c02 win: 2000
server.8080 > client.1391: [DF] PA 8c02:8cde(dc) ack: 1c2 win: 8000
server.8080 > client.1391: [DF] A 8cde:9292(5b4) ack: 1c2 win: 8000
server.8080 > client.1391: [DF] A 9292:9846(5b4) ack: 1c2 win: 8000
server.8080 > client.1391: [DF] A 9846:9dfa(5b4) ack: 1c2 win: 8000
client.1391 > server.8080: PA
server.8080 > client.1391: [DF] A 9dfa:a3ae(5b4) ack: 1c2 win: 8000
server.8080 > client.1391: [DF] A a3ae:a962(5b4) ack: 1c2 win: 8000
server.8080 > client.1391: [DF] A a962:af16(5b4) ack: 1c2 win: 8000
server.8080 > client.1391: [DF] A af16:b4ca(5b4) ack: 1c2 win: 8000
client.1391 > server.8080: PA
server.8080 > client.1391: [DF] A b4ca:ba7e(5b4) ack: 1c2 win: 8000
server.8080 > client.1391: [DF] A b4ca:ba7e(5b4) ack: 1c2 win: 8000
```

```
client.1391 > server.8080: PA
server.8080 > client.1391: [DF] A ba7e:bdfa(37c) ack: 1c2 win: 8000
client.1391 > server.8080: PA
server.8080 > client.1391: [DF] A bdfa:bdfb(1) ack: 1c2 win: 8000
client.1391 > server.8080: PA

[ HTTP client aborts and enters FIN_WAIT_1 ]

client.1391 > server.8080: FPA

[ server ACKs the FIN and enters CLOSE_WAIT ]

server.8080 > client.1391: [DF] A

[ client enters FIN_WAIT_2 ]

server.8080 > client.1391: [DF] A bdfa:bdfb(1) ack: 1c3 win: 8000

[ server continues to try to send its data ]

client.1391 > server.8080: PA < window = 0 >
server.8080 > client.1391: [DF] A bdfa:bdfb(1) ack: 1c3 win: 8000
client.1391 > server.8080: PA < window = 0 >
server.8080 > client.1391: [DF] A bdfa:bdfb(1) ack: 1c3 win: 8000
client.1391 > server.8080: PA < window = 0 >
server.8080 > client.1391: [DF] A bdfa:bdfb(1) ack: 1c3 win: 8000
client.1391 > server.8080: PA < window = 0 >
server.8080 > client.1391: [DF] A bdfa:bdfb(1) ack: 1c3 win: 8000
client.1391 > server.8080: PA < window = 0 >

[ ... repeat ad exhaustium ... ]

Trace file demonstrating correct behavior
   Made using an unknown network analyzer.  No drop information
   available.

client > server D=80 S=59500 Syn Seq=337 Len=0 Win=8760
server > client D=59500 S=80 Syn Ack=338 Seq=80153 Len=0 Win=8760
client > server D=80 S=59500 Ack=80154 Seq=338 Len=0 Win=8760

[ ... normal data omitted ... ]

client > server D=80 S=59500 Ack=14559 Seq=596 Len=0 Win=8760
server > client D=59500 S=80 Ack=596 Seq=114559 Len=1460 Win=8760

[ client closes connection ]

client > server D=80 S=59500 Fin Seq=596 Len=0 Win=8760
```

```
server > client D=59500 S=80 Ack=597 Seq=116019 Len=1460 Win=8760

[ client sends RST (RFC1122 4.2.2.13) ]

client > server D=80 S=59500 Rst Seq=597 Len=0 Win=0
server > client D=59500 S=80 Ack=597 Seq=117479 Len=1460 Win=8760
client > server D=80 S=59500 Rst Seq=597 Len=0 Win=0
server > client D=59500 S=80 Ack=597 Seq=118939 Len=1460 Win=8760
client > server D=80 S=59500 Rst Seq=597 Len=0 Win=0
server > client D=59500 S=80 Ack=597 Seq=120399 Len=892 Win=8760
client > server D=80 S=59500 Rst Seq=597 Len=0 Win=0
server > client D=59500 S=80 Ack=597 Seq=121291 Len=1460 Win=8760
client > server D=80 S=59500 Rst Seq=597 Len=0 Win=0
```

"client" sends a number of RSTs, one in response to each incoming packet from "server". One might wonder why "server" keeps sending data packets after it has received a RST from "client"; the explanation is that "server" had already transmitted all five of the data packets before receiving the first RST from "client", so it is too late to avoid transmitting them.

How to detect

The problem can be detected by inspecting packet traces of a large, interrupted bulk transfer.

2.17.

Name of Problem

Failure to RST on close with data pending

Classification

Resource management

Description

When an application closes a connection in such a way that it can no longer read any received data, the TCP SHOULD, per section 4.2.2.13 of RFC 1122, send a RST if there is any unread received data, or if any new data is received. A TCP that fails to do so exhibits "Failure to RST on close with data pending".

Note that, for some TCPs, this situation can be caused by an application "crashing" while a peer is sending data.

We have observed a number of TCPs that exhibit this problem. The problem is less serious if any subsequent data sent to the now-closed connection endpoint elicits a RST (see illustration below).

Significance

This problem is most significant for endpoints that engage in large numbers of connections, as their ability to do so will be curtailed as they leak away resources.

Implications

Failure to reset the connection can lead to permanently hung connections, in which the remote endpoint takes no further action to tear down the connection because it is waiting on the local TCP to first take some action. This is particularly the case if the local TCP also allows the advertised window to go to zero, and fails to tear down the connection when the remote TCP engages in "persist" probes (see example below).

Relevant RFCs

RFC 1122 section 4.2.2.13. Also, 4.2.2.17 for the zero-window probing discussion below.

Trace file demonstrating it

Made using tcpdump. No drop information available.

```
13:11:46.04 A > B: S 458659166:458659166(0) win 4096
                    <mss 1460,wscale 0,eol> (DF)
13:11:46.04 B > A: S 792320000:792320000(0) ack 458659167
                    win 4096
13:11:46.04 A > B: . ack 1 win 4096 (DF)
13:11.55.80 A > B: . 1:513(512) ack 1 win 4096 (DF)
13:11.55.80 A > B: . 513:1025(512) ack 1 win 4096 (DF)
13:11:55.83 B > A: . ack 1025 win 3072
13:11.55.84 A > B: . 1025:1537(512) ack 1 win 4096 (DF)
13:11.55.84 A > B: . 1537:2049(512) ack 1 win 4096 (DF)
13:11.55.85 A > B: . 2049:2561(512) ack 1 win 4096 (DF)
13:11:56.03 B > A: . ack 2561 win 1536
13:11.56.05 A > B: . 2561:3073(512) ack 1 win 4096 (DF)
13:11.56.06 A > B: . 3073:3585(512) ack 1 win 4096 (DF)
13:11.56.06 A > B: . 3585:4097(512) ack 1 win 4096 (DF)
13:11:56.23 B > A: . ack 4097 win 0
13:11:58.16 A > B: . 4096:4097(1) ack 1 win 4096 (DF)
13:11:58.16 B > A: . ack 4097 win 0
13:12:00.16 A > B: . 4096:4097(1) ack 1 win 4096 (DF)
13:12:00.16 B > A: . ack 4097 win 0
13:12:02.16 A > B: . 4096:4097(1) ack 1 win 4096 (DF)
13:12:02.16 B > A: . ack 4097 win 0
13:12:05.37 A > B: . 4096:4097(1) ack 1 win 4096 (DF)
13:12:05.37 B > A: . ack 4097 win 0
13:12:06.36 B > A: F 1:1(0) ack 4097 win 0
13:12:06.37 A > B: . ack 2 win 4096 (DF)
13:12:11.78 A > B: . 4096:4097(1) ack 2 win 4096 (DF)
```

```
13:12:11.78 B > A: . ack 4097 win 0
13:12:24.59 A > B: . 4096:4097(1) ack 2 win 4096 (DF)
13:12:24.60 B > A: . ack 4097 win 0
13:12:50.22 A > B: . 4096:4097(1) ack 2 win 4096 (DF)
13:12:50.22 B > A: . ack 4097 win 0
```

Machine B in the trace above does not drop received data when the socket is "closed" by the application (in this case, the application process was terminated). This occurred at approximately 13:12:06.36 and resulted in the FIN being sent in response to the close. However, because there is no longer an application to deliver the data to, the TCP should have instead sent a RST.

Note: Machine A's zero-window probing is also broken. It is resending old data, rather than new data. Section 3.7 in RFC 793 and Section 4.2.2.17 in RFC 1122 discuss zero-window probing.

Trace file demonstrating better behavior

Made using tcpdump. No drop information available.

Better, but still not fully correct, behavior, per the discussion below. We show this behavior because it has been observed for a number of different TCP implementations.

```
13:48:29.24 C > D: S 73445554:73445554(0) win 4096
                    <mss 1460,wscale 0,eol> (DF)
13:48:29.24 D > C: S 36050296:36050296(0) ack 73445555
                    win 4096 <mss 1460,wscale 0,eol> (DF)
13:48:29.25 C > D: . ack 1 win 4096 (DF)
13:48:30.78 C > D: . 1:1461(1460) ack 1 win 4096 (DF)
13:48:30.79 C > D: . 1461:2921(1460) ack 1 win 4096 (DF)
13:48:30.80 D > C: . ack 2921 win 1176 (DF)
13:48:32.75 C > D: . 2921:4097(1176) ack 1 win 4096 (DF)
13:48:32.82 D > C: . ack 4097 win 0 (DF)
13:48:34.76 C > D: . 4096:4097(1) ack 1 win 4096 (DF)
13:48:34.84 D > C: . ack 4097 win 0 (DF)
13:48:36.34 D > C: FP 1:1(0) ack 4097 win 4096 (DF)
13:48:36.34 C > D: . 4097:5557(1460) ack 2 win 4096 (DF)
13:48:36.34 D > C: R 36050298:36050298(0) win 24576
13:48:36.34 C > D: . 5557:7017(1460) ack 2 win 4096 (DF)
13:48:36.34 D > C: R 36050298:36050298(0) win 24576
```

In this trace, the application process is terminated on Machine D at approximately 13:48:36.34. Its TCP sends the FIN with the window opened again (since it discarded the previously received data). Machine C promptly sends more data, causing Machine D to

reset the connection since it cannot deliver the data to the application. Ideally, Machine D SHOULD send a RST instead of dropping the data and re-opening the receive window.

Note: Machine C's zero-window probing is broken, the same as in the example above.

Trace file demonstrating correct behavior

Made using tcpdump. No losses reported by the packet filter.

```
14:12:02.19 E > F: S 1143360000:1143360000(0) win 4096
14:12:02.19 F > E: S 1002988443:1002988443(0) ack 1143360001
                    win 4096 <mss 1460> (DF)
14:12:02.19 E > F: . ack 1 win 4096
14:12:10.43 E > F: . 1:513(512) ack 1 win 4096
14:12:10.61 F > E: . ack 513 win 3584 (DF)
14:12:10.61 E > F: . 513:1025(512) ack 1 win 4096
14:12:10.61 E > F: . 1025:1537(512) ack 1 win 4096
14:12:10.81 F > E: . ack 1537 win 2560 (DF)
14:12:10.81 E > F: . 1537:2049(512) ack 1 win 4096
14:12:10.81 E > F: . 2049:2561(512) ack 1 win 4096
14:12:10.81 E > F: . 2561:3073(512) ack 1 win 4096
14:12:11.01 F > E: . ack 3073 win 1024 (DF)
14:12:11.01 E > F: . 3073:3585(512) ack 1 win 4096
14:12:11.01 E > F: . 3585:4097(512) ack 1 win 4096
14:12:11.21 F > E: . ack 4097 win 0 (DF)
14:12:15.88 E > F: . 4097:4098(1) ack 1 win 4096
14:12:16.06 F > E: . ack 4097 win 0 (DF)
14:12:20.88 E > F: . 4097:4098(1) ack 1 win 4096
14:12:20.91 F > E: . ack 4097 win 0 (DF)
14:12:21.94 F > E: R 1002988444:1002988444(0) win 4096
```

When the application terminates at 14:12:21.94, F immediately sends a RST.

Note: Machine E's zero-window probing is (finally) correct.

How to detect

The problem can often be detected by inspecting packet traces of a transfer in which the receiving application terminates abnormally. When doing so, there can be an ambiguity (if only looking at the trace) as to whether the receiving TCP did indeed have unread data that it could now no longer deliver. To provoke this to happen, it may help to suspend the receiving application so that it fails to consume any data, eventually exhausting the advertised window. At this point, since the advertised window is zero, we know that

the receiving TCP has undelivered data buffered up. Terminating the application process then should suffice to test the correctness of the TCP's behavior.

2.18.

Name of Problem
Options missing from TCP MSS calculation

Classification
Reliability / performance

Description
When a TCP determines how much data to send per packet, it calculates a segment size based on the MTU of the path. It must then subtract from that MTU the size of the IP and TCP headers in the packet. If IP options and TCP options are not taken into account correctly in this calculation, the resulting segment size may be too large. TCPs that do so are said to exhibit "Options missing from TCP MSS calculation".

Significance
In some implementations, this causes the transmission of strangely fragmented packets. In some implementations with Path MTU (PMTU) discovery [RFC1191], this problem can actually result in a total failure to transmit any data at all, regardless of the environment (see below).

Arguably, especially since the wide deployment of firewalls, IP options appear only rarely in normal operations.

Implications
In implementations using PMTU discovery, this problem can result in packets that are too large for the output interface, and that have the DF (don't fragment) bit set in the IP header. Thus, the IP layer on the local machine is not allowed to fragment the packet to send it out the interface. It instead informs the TCP layer of the correct MTU size of the interface; the TCP layer again miscomputes the MSS by failing to take into account the size of IP options; and the problem repeats, with no data flowing.

Relevant RFCs
RFC 1122 describes the calculation of the effective send MSS. RFC 1191 describes Path MTU discovery.

Trace file demonstrating it

Trace file taking using tcpdump on host C. The first trace demonstrates the fragmentation that occurs without path MTU discovery:

```
13:55:25.488728 A.65528 > C.discard:
        P 567833:569273(1440) ack 1 win 17520
        <nop,nop,timestamp 3839 1026342>
        (frag 20828:1472@0+)
        (ttl 62, optlen=8 LSRR{B#} NOP)

13:55:25.488943 A > C:
        (frag 20828:8@1472)
        (ttl 62, optlen=8 LSRR{B#} NOP)

13:55:25.489052 C.discard > A.65528:
        . ack 566385 win 60816
        <nop,nop,timestamp 1026345 3839> (DF)
        (ttl 60, id 41266)
```

Host A repeatedly sends 1440-octet data segments, but these hare fragmented into two packets, one with 1432 octets of data, and another with 8 octets of data.

The second trace demonstrates the failure to send any data segments, sometimes seen with hosts doing path MTU discovery:

```
13:55:44.332219 A.65527 > C.discard:
        S 1018235390:1018235390(0) win 16384
        <mss 1460,nop,wscale 0,nop,nop,timestamp 3876 0> (DF)
        (ttl 62, id 20912, optlen=8 LSRR{B#} NOP)

13:55:44.333015 C.discard > A.65527:
        S 1271629000:1271629000(0) ack 1018235391 win 60816
        <mss 1460,nop,wscale 0,nop,nop,timestamp 1026383 3876> (DF)
        (ttl 60, id 41427)

13:55:44.333206 C.discard > A.65527:
        S 1271629000:1271629000(0) ack 1018235391 win 60816
        <mss 1460,nop,wscale 0,nop,nop,timestamp 1026383 3876> (DF)
        (ttl 60, id 41427)
```

This is all of the activity seen on this connection. Eventually host C will time out attempting to establish the connection.

How to detect

The "netcat" utility [Hobbit96] is useful for generating source routed packets:

```
1% nc C discard
(interactive typing)
^C
2% nc C discard < /dev/zero
^C
3% nc -g B C discard
(interactive typing)
^C
4% nc -g B C discard < /dev/zero
^C
```

Lines 1 through 3 should generate appropriate packets, which can be verified using tcpdump. If the problem is present, line 4 should generate one of the two kinds of packet traces shown.

How to fix

The implementation should ensure that the effective send MSS calculation includes a term for the IP and TCP options, as mandated by RFC 1122.

3. Security Considerations

This memo does not discuss any specific security-related TCP implementation problems, as the working group decided to pursue documenting those in a separate document. Some of the implementation problems discussed here, however, can be used for denial-of-service attacks. Those classified as congestion control present opportunities to subvert TCPs used for legitimate data transfer into excessively loading network elements. Those classified as "performance", "reliability" and "resource management" may be exploitable for launching surreptitious denial-of-service attacks against the user of the TCP. Both of these types of attacks can be extremely difficult to detect because in most respects they look identical to legitimate network traffic.

4. Acknowledgements

Thanks to numerous correspondents on the tcp-impl mailing list for their input: Steve Alexander, Larry Backman, Jerry Chu, Alan Cox, Kevin Fall, Richard Fox, Jim Gettys, Rick Jones, Allison Mankin, Neal McBurnett, Perry Metzger, der Mouse, Thomas Narten, Andras Olah, Steve Parker, Francesco Potorti', Luigi Rizzo, Allyn Romanow, Al Smith, Jerry Toporek, Joe Touch, and Curtis Villamizar.

Thanks also to Josh Cohen for the traces documenting the "Failure to send a RST after Half Duplex Close" problem; and to John Polstra, who analyzed the "Window probe deadlock" problem.

5. References

[Allman97] M. Allman, "Fixing Two BSD TCP Bugs," Technical Report CR-204151, NASA Lewis Research Center, Oct. 1997. http://roland.grc.nasa.gov/~mallman/papers/bug.ps

[RFC2414] Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's Initial Window", RFC 2414, September 1998.

[RFC1122] Braden, R., Editor, "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1122, October 1989.

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

[Brakmo95] L. Brakmo and L. Peterson, "Performance Problems in BSD4.4 TCP," ACM Computer Communication Review, 25(5):69-86, 1995.

[RFC813] Clark, D., "Window and Acknowledgement Strategy in TCP," RFC 813, July 1982.

[Dawson97] S. Dawson, F. Jahanian, and T. Mitton, "Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool," to appear in Software Practice & Experience, 1997. A technical report version of this paper can be obtained at ftp://rtcl.eecs.umich.edu/outgoing/sdawson/CSE-TR-298-96.ps.gz.

[Fall96] K. Fall and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP," ACM Computer Communication Review, 26(3):5-21, 1996.

[Hobbit96] Hobbit, Avian Research, netcat, available via anonymous ftp to ftp.avian.org, 1996.

[Hoe96] J. Hoe, "Improving the Start-up Behavior of a Congestion Control Scheme for TCP," Proc. SIGCOMM '96.

[Jacobson88] V. Jacobson, "Congestion Avoidance and Control," Proc. SIGCOMM '88. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z

[Jacobson89] V. Jacobson, C. Leres, and S. McCanne, tcpdump, available via anonymous ftp to ftp.ee.lbl.gov, Jun. 1989.

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

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, November 1990.

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

[Paxson97] V. Paxson, "Automated Packet Trace Analysis of TCP Implementations," Proc. SIGCOMM '97, available from ftp://ftp.ee.lbl.gov/papers/vp-tcpanaly-sigcomm97.ps.Z.

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

[RFC2001] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC 2001, January 1997.

[Stevens94] W. Stevens, "TCP/IP Illustrated, Volume 1", Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[Wright95] G. Wright and W. Stevens, "TCP/IP Illustrated, Volume 2", Addison-Wesley Publishing Company, Reading Massachusetts, 1995.

6. Authors' Addresses

Vern Paxson
ACIRI / ICSI
1947 Center Street
Suite 600
Berkeley, CA 94704-1198

Phone: +1 510/642-4274 x302
EMail: vern@aciri.org

Mark Allman <mallman@grc.nasa.gov>
NASA Glenn Research Center/Sterling Software
Lewis Field
21000 Brookpark Road
MS 54-2
Cleveland, OH 44135
USA

Phone: +1 216/433-6586
Email: mallman@grc.nasa.gov

Scott Dawson
Real-Time Computing Laboratory
EECS Building
University of Michigan
Ann Arbor, MI 48109-2122
USA

Phone: +1 313/763-5363
EMail: sdawson@eecs.umich.edu

William C. Fenner
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
USA

Phone: +1 650/812-4816
EMail: fenner@parc.xerox.com

Jim Griner <jgriner@grc.nasa.gov>
NASA Glenn Research Center
Lewis Field
21000 Brookpark Road
MS 54-2
Cleveland, OH 44135
USA

Phone: +1 216/433-5787
EMail: jgriner@grc.nasa.gov

Ian Heavens
Spider Software Ltd.
8 John’s Place, Leith
Edinburgh EH6 7EL
UK

Phone: +44 131/475-7015
EMail: ian@spider.com

Kevin Lahey
NASA Ames Research Center/MRJ
MS 258-6
Moffett Field, CA 94035
USA

Phone: +1 650/604-4334
EMail: kml@nas.nasa.gov

Jeff Semke
Pittsburgh Supercomputing Center
4400 Fifth Ave
Pittsburgh, PA 15213
USA

Phone: +1 412/268-4960
EMail: semke@psc.edu

Bernie Volz
Process Software Corporation
959 Concord Street
Framingham, MA 01701
USA

Phone: +1 508/879-6994
EMail: volz@process.com

7. Full Copyright Statement