10 TCP Flow Control / Error Handling

1 Thema des Praktikums

Im folgenden Praktikum werden die in der Theorie besprochenen Mechanismen zur Flusskontrolle und zur Fehlerbehandlung von TCP untersucht. Hierzu werden Optionen im TCP Header betrachtet, sowie detailliert die Fenstermechanismen und die Fehlerbehandlung bei kurzzeitigem Verbindungsunterbruch untersucht.

2 Vorbereitung

Um den Versuch erfolgreich durchführen zu können, sollten Sie alle TCP Segmente in Wireshark sehen. Hierzu müssen gewisse TCP Offloading Funktionen (https://lwn.net/Articles/358910/) deaktiviert werden.

Wozu wird TCP Offloading verwendet?

In den letzten Jahren ist die Kommunikationsgeschwindigkeit in Ethernet-Systemen schneller gestiegen als die Geschwindigkeit des Computerprozessors. Dies führt zu einem Input/Output (I/O)-Engpass. Der Prozessor, der in erster Linie für Computing und nicht für I/O ausgelegt ist, kann mit den durch das Netzwerk fließenden Daten nicht mithalten. Dadurch wird der TCP/IP-Flow mit einer Geschwindigkeit verarbeitet, die langsamer ist als die Geschwindigkeit des Netzwerks. TOE löst dieses Problem, indem es die Belastung (Entlastung) des Mikroprozessors und des I/O-Subsystems beseitigt.

Wozu dient «tso», «gso» und «rso»?

TSO: TCP Segmentation Offload -> Netzwerkschnittstellenkarte (NIC) teilt die Multipacket-Puffer in separate Pakete auf.

GSO: Generic Segmentation Offload -> analog TSO einfach generisch

RSO:

Inwiefern könnte die Versuchsdurchführung beeinflusst werden, wenn TCP Offloading aktiv ist?

Unterschiedliche Durchlaufzeit

2.1 Vorbereitung TCP Header Optionen

Unter https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml finden Sie die offizielle Aufstellung der IANA zu den festgelegten TCP Header Optionen (Abschnitt «TCP Option Kind Numbers»). Lesen Sie im RFC 7323 nach, wie die Option «Window Scale» verwendet wird.

Welche Werte kann das Datenfeld dieser Option annehmen? Welches ist somit die maximale Fenstergrösse, die mit dieser Option dem Sender mitgeteilt werden kann?

max: <2^31

In welchen Datagrammen (beim TCP Verbindungsaufbau und beim Datenaustausch) erwarten Sie, diese Option vorzufinden?

Welche Optionen erwarten Sie in jedem TCP Segment während der Datenübertragung vorzufinden, welche nur beim Verbindungsaufbau (Betrachten Sie Option Kind 1-5 und 8)?

Option Kind	Anzahl Bytes im TCP Header	Bezeichnung	Verwendung beim Verbindungsaufbau	Verwendung beim Datenaustausch
1			[ ]	[ ]
2			[ ]	[ ]
3			[ ]	[ ]
4			[ ]	[ ]
5			[ ]	[ ]
8			[ ]	[ ]

Tabelle 1: TCP Optionen

2.2 Vorbereitung Sliding Window bei TCP

Welches der beiden folgenden Fenster, die bei Fluss- und Überlastkontrolle eine Rolle spielen, erwarten Sie in der Wireshark Aufzeichnung feststellen zu können, welches nicht oder nur indirekt?

Advertised Window:

Congestion Window:

Gemäss Skript wird als Anfangswert für den Slow Start Threshold (ssthresh) ein Wert von 64 kByte gewählt. RFC 5681 schlägt eine andere Wahl vor.

Welcher Wert wird vorgeschlagen?

Was bewirkt diese Wahl?

Warum, denken Sie, ist die neue Wahl sinnvoll?

- **Zeigen Sie Ihre Vorbereitungen dem Laborbetreuer.**

3 Versuchsdurchführung zu den TCP Optionen

Um Störungen zu vermeiden, verwenden wir ein lokales Netz und das Ethernet-Interface eth1 der Rechner.

- Bauen Sie die Versuchskonfiguration gemäss Abbildung 1 auf: Die Rechner werden via eth1 über zwei Hirschmann Switches verbunden und mit Linux gestartet.
  Wichtig: Das Ethernet-Interface eth0 bleibt mit dem ZHAW-Netz verbunden.

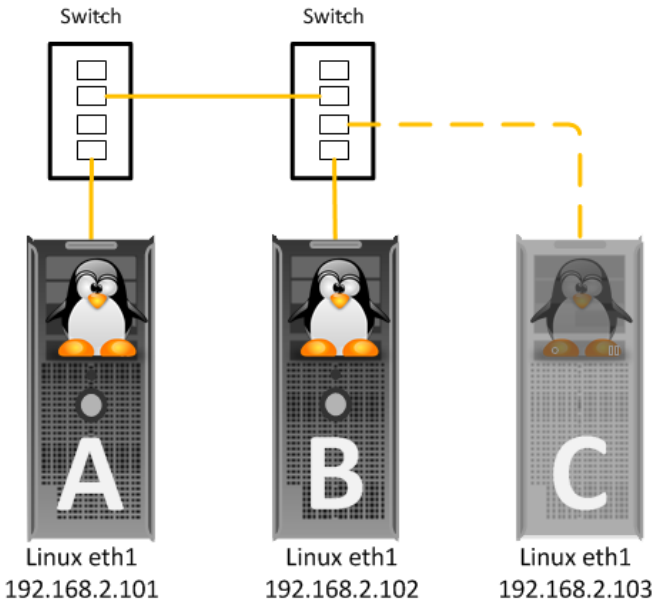


Abbildung 1: Versuchsaufbau «TCP Optionen»

- Setzen Sie die Switches wie in den vorangegangenen Versuchen mittels der USB Sticks zurück.
- Öffnen Sie auf jedem Rechner ein Terminal, laden Sie die Dateien mit dem Script **download-kt** zurück und wechseln Sie dann ins Verzeichnis dieses Praktikums:
  ```
  download-kt
  cd /home/ktlabor/praktika/10_tcp_flow_control
  ```
- **Am Ende des Praktikums** führen Sie das Script `reset-kt-home` aus. Das Script löscht das Home-Verzeichnis des Users ktlabor und erstellt es neu.

Es werden im Prinzip die gleichen Client-Server-Programme wie in den letzten Praktika verwendet, ausser dass mehr Daten geschickt werden und die „_slow“-Varianten mit `usleep()` künstlich gebremst wurden.

- Compilieren Sie auf den Rechnern die vorhandenen C-Programme. Zum Beispiel so:
  ```
  for f in *.c; do gcc -Wall -o ${f%.c} $f; done
  ```
- Damit im Wireshark die effektiven Pakete angezeigt werden, müssen Sie das Offloading ausschalten:
  ```
  ethtool -K eth1 tso off
  ethtool -K eth1 gso off
  ethtool -K eth1 gro off
  ```
- Überprüfen Sie die Einstellungen mit dem Befehl:
  ```
  ethtool -k eth1 | grep offload
  ```
  Alle Einstellungen sollten auf "off" sein (ausser VLAN-Einstellungen).

3.1 Untersuchung der TCP-Optionen

- Starten Sie auf einem der Rechner Wireshark.
- Starten Sie das Programm server_slow und machen Sie mit client_fast einen Zugriff.
  `./client_fast <server-IP>`
- Untersuchen Sie mit Wireshark die SYN-Frames.

Welche Optionen sind vorhanden (beide Richtungen einzeln betrachten) und was bedeuten sie?

Wie gross sind die vom Client signalisierten initialen Window-Grössen?

Erstes Client Packet (SYN):

Zweites Client Packet (ACK):

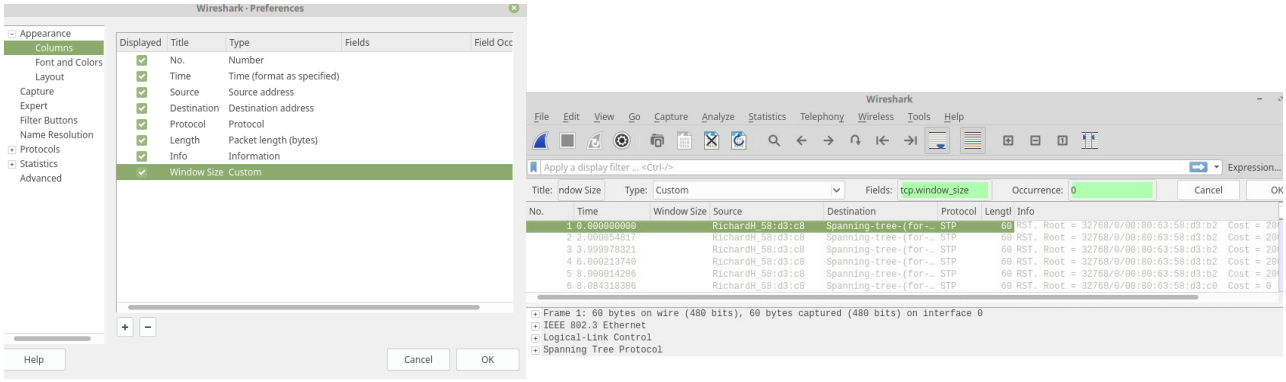
- **Zeigen Sie die Resultate dem Laborbetreuer.**

4 Sliding Window Mechanismen von TCP

4.1 Sliding-Window-Mechanismus ohne Reduktion der Window-Size

Zunächst soll der Fall betrachtet werden, wo der Flow-Control-Mechanismus die Window-Size nicht reduziert, d.h. der Client alle Daten des Servers verarbeiten kann.

- Damit die Auswertung der *Window-Grösse* in Excel einfacher zu handhaben ist, wird in Wireshark eine zusätzliche Spalte für die *Window-Grösse* erstellt:
    - Edit Preferences öffnen → Columns → Add (eine neue Spalte hinzufügen) → umbenennen in «Window Size». In der Auswahl «Type» `Custom` anwählen.
    - Die Spalte auswählen, im Kontextmenu «Edit Column» auswählen, dann in der Eingabezeile «Fields:» `tcp.window_size` eintragen.



- Starten Sie server-slow (ist mit usleep künstlich verlangsamt) und machen Sie mit client-fast einen Zugriff darauf (Dauer ca. 50 Sek.). Zeichnen Sie diese Verbindung mit Wireshark auf.
- Exportieren Sie die im Wireshark aufgezeichneten Daten in eine CSV-Datei (Comma Separated Values): Lassen Sie Wireshark laufen, setzen Sie einen Filter auf TCP und exportieren Sie die Pakete via File → Export Packet Dissections → As „CSV"..., Packet-Range="Displayed").
- Öffnen Sie die exportierte Datei (mit Libre-Office öffnen). Die Tabelle sollte nun folgende Spalten aufweisen: No. / Time / Source / Destination / Protocol / Info / Window Size
- Erstellen Sie nun eine Grafik, die den Verlauf der Window Size des Clients zeigt. Beispiel:
    - Markieren Sie die ganze Tabelle (*Ctrl-A)* und setzen die Filterfunktion *(Data →Filter →Autofilter)*
    - Wählen Sie in der Spalte *Source* die vom Client benutze IP-Adresse aus, so dass nur der relevante Datenverkehr sichtbar bleibt (Client→Server).
    - Die Spalte *Window-Size* markieren und eine Linien-Grafik einfügen *(Insert →Chart →Chart Type=Line, Lines Only)*.

Was passiert mit der Window-Grösse des Clients?

Wie verläuft die Window-Grösse des Servers?

4.2 Sliding-Window-Mechanismus mit Flow-Control

Nun soll der Fall betrachtet werden, wo der Flow-Control-Mechanismus das Window reduziert, weil der Client die Daten des Servers sonst nicht rechtzeitig verarbeiten kann.

- Starten Sie dazu server-fast und machen Sie mit client-slow den Zugriff (Dauer ca. 20 Sek.).
  `./client_slow <server-IP>`
- Erstellen Sie auch für diese Verbindung eine Line-Grafik, die den Verlauf der Window Size darstellt.
- Interpretieren Sie das Diagramm:

Wodurch wird die grosse Spitze am Anfang bestimmt?

Was verursacht die kleinen Rippel?

- Betrachten Sie dieselbe Grösse (*Window-Size*) als Funktion der Zeit (xy-Diagramm) und erklären Sie den Unterschied der Grafiken (Spalten Time & Window selektieren, Insert→Chart→Chart Type = XY (Scatter), Lines Only).

Warum ist die Spitze am Anfang so kurz im Vergleich zur obigen Grafik? (Wodurch ist die Zeit gegeben?)

Woher kommt die lange Rampe bzw. Peak am Ende? Was geschieht da? (Betrachten Sie auch die Rohwerte in der Aufzeichnung).

- **Zeigen Sie die Resultate dem Laborbetreuer.**

5 Verhalten bei einem kurzzeitigen Verbindungsunterbruch

- Starten Sie den client-slow und server-fast. Unterbrechen Sie die Verbindung zwischen den Switches nach ca. 2 Sekunden für einige Sekunden. Analysieren Sie das Verhalten mit Wireshark **auf beiden Hosts**.
- Führen Sie zwei Versuche durch: beim ersten Versuch stellen Sie die Verbindung nach ca. 3 Sekunden wieder her, beim zweiten Versuch warten Sie 15 Sekunden mit dem Wiedereinstecken.

Was passiert mit der Übertragung? Wie verhält sich der Server?

Wie verhält sich der Client?

Was passiert, wenn die Verbindung wiederhergestellt wird?

- **Zeigen Sie die Resultate dem Laborbetreuer.**

6 Anhang: Unterlagen zu TCP

6.1 Anhang 1: Offloading Mechanismen

Quelle: https://lwn.net/Articles/358910/

(...)

Given the importance of per-packet overhead, one might well ask whether it makes sense to raise the MTU. That can be done; the "jumbo frames" mechanism can handle packets up to 9KB in size. The problem, (...) is that "the Internet happened." Most connections of interest go across the Internet, and those are all bound by the lowest MTU in the entire path. Sometimes that MTU is even less than 1500 bytes. Protocol-based mechanisms for finding out what that MTU is exist, but they don't work well on the Internet; in particular, a lot of firewall setups break it. So, while jumbo frames might work well for local networks, the sad fact is that we're stuck with 1500 bytes on the wider Internet.

If we can't use a larger MTU, we can go for the next-best thing: pretend that we're using a larger MTU. For a few years now Linux has supported network adapters which perform "TCP segmentation offload," or TSO. With a TSO-capable adapter, the kernel can prepare much larger packets (64KB, say) for outgoing data; the adapter will then re-segment the data into smaller packets as the data hits the wire. That cuts the kernel's per-packet overhead by a factor of 40. TSO is well supported in Linux; for systems which are engaged mainly in the sending of data, it's sufficient to make 10GB work at full speed.

The kernel actually has a generic segmentation offload mechanism (called GSO) which is not limited to TCP. It turns out that performance improves even if the feature is emulated in the driver. But GSO only works for data transmission, not reception. That limitation is entirely fine for broad classes of users; sites providing content to the net, for example, send far more data than they receive. But other sites have different workloads, and, for them, packet reception overhead is just as important as transmission overhead.

Solutions on the receive side have been a little slower in coming, and not just because the first users were more interested in transmission performance. Optimizing the receive side is harder because packet reception is, in general, harder. When it is transmitting data, the kernel is in complete control and able to throttle sending processes if necessary. But incoming packets are entirely asynchronous events, under somebody else's control, and the kernel just has to cope with what it gets.

Still, a solution has emerged in the form of "large receive offload" (LRO), which takes a very similar approach: incoming packets are merged at reception time so that the operating system sees far fewer of them. This merging can be done either in the driver or in the hardware; even LRO emulation in the driver has performance benefits. LRO is widely supported by 10G drivers under Linux.

But LRO is a bit of a flawed solution, according to Herbert; the real problem is that it "merges everything in sight." This transformation is lossy; if there are important differences between the headers in incoming packets, those differences will be lost. And that breaks things. If a system is serving as a router, it really should not be changing the headers on packets as they pass through. LRO can totally break satellite-based connections, where some very strange header tricks are done by providers to make the whole thing work. And bridging breaks, which is a serious problem: most virtualization setups use a virtual network bridge between the host and its clients. One might simply avoid using LRO in such situations, but these also tend to be the workloads that one really wants to optimize. Virtualized networking, in particular, is already slower; any possible optimization in this area is much needed.

The solution is generic receive offload (GRO). In GRO, the criteria for which packets can be merged is greatly restricted; the MAC headers must be identical and only a few TCP or IP headers can differ. In fact, the set of headers which can differ is severely restricted: checksums are necessarily different, and the IP ID field is allowed to increment. Even the TCP timestamps must be identical, which is less of a restriction than it may seem; the timestamp is a relatively low-resolution field, so it's not uncommon for lots of packets to have the same timestamp. As a result of these restrictions, merged packets can be resegmented losslessly; as an added benefit, the GSO code can be used to perform resegmentation.

One other nice thing about GRO is that, unlike LRO, it is not limited to TCP/IPv4.

6.2 Anhang 2: Auszug aus RFC 5618

DRAFT STANDARD
Errata Exist

Network Working Group M. Allman
Request for Comments: 5681 V. Paxson
Obsoletes: 2581 ICSI
Category: Standards Track E. Blanton
Purdue University
September 2009

TCP Congestion Control

Abstract

This document defines TCP's four intertwined congestion control algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery. In addition, the document specifies how TCP should begin transmission after a relatively long idle period, as well as discussing various acknowledgment generation methods. This document obsoletes RFC 2581.

Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (c) 2009 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents in effect on the date of publication of this document (http://trustee.ietf.org/license-info). Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table Of Contents

1. Introduction

This document specifies four TCP [RFC793] congestion control algorithms: slow start, congestion avoidance, fast retransmit and fast recovery. These algorithms were devised in [Jac88] and [Jac90]. Their use with TCP is standardized in [RFC1122]. Additional early work in additive-increase, multiplicative-decrease congestion control is given in [CJ89].

Note that [Ste94] provides examples of these algorithms in action and [WS95] provides an explanation of the source code for the BSD implementation of these algorithms.

In addition to specifying these congestion control algorithms, this document specifies what TCP connections should do after a relatively long idle period, as well as specifying and clarifying some of the issues pertaining to TCP ACK generation.

This document obsoletes [RFC2581], which in turn obsoleted [RFC2001].

This document is organized as follows. Section 2 provides various definitions that will be used throughout the document. Section 3 provides a specification of the congestion control algorithms. Section 4 outlines concerns related to the congestion control algorithms and finally, section 5 outlines security considerations.

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

2. Definitions

This section provides the definition of several terms that will be used throughout the remainder of this document.

SEGMENT: A segment is ANY TCP/IP data or acknowledgment packet (or both).

SENDER MAXIMUM SEGMENT SIZE (SMSS): The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery [RFC1191, RFC4821] algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.

RECEIVER MAXIMUM SEGMENT SIZE (RMSS): The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup. Or, if the MSS option is not used, it is 536 bytes [RFC1122]. The size does not include the TCP/IP headers and options.

FULL-SIZED SEGMENT: A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).

RECEIVER WINDOW (rwnd): The most recently advertised receiver window.

CONGESTION WINDOW (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

INITIAL WINDOW (IW): The initial window is the size of the sender's congestion window after the three-way handshake is completed.

LOSS WINDOW (LW): The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission timer.

RESTART WINDOW (RW): The restart window is the size of the congestion window after a TCP restarts transmission after an idle period (if the slow start algorithm is used; see section 4.1 for more discussion).

FLIGHT SIZE: The amount of data that has been sent but not yet cumulatively acknowledged.

DUPLICATE ACKNOWLEDGMENT: An acknowledgment is considered a "duplicate" in the following algorithms when (a) the receiver of the ACK has outstanding data, (b) the incoming acknowledgment carries no data, (c) the SYN and FIN bits are both off, (d) the acknowledgment number is equal to the greatest acknowledgment received on the given connection (TCP.UNA from [RFC793]) and (e) the advertised window in the incoming acknowledgment equals the advertised window in the last incoming acknowledgment.

Alternatively, a TCP that utilizes selective acknowledgments (SACKs) [RFC2018, RFC2883] can leverage the SACK information to determine when an incoming ACK is a "duplicate" (e.g., if the ACK contains previously unknown SACK information).

3. Congestion Control Algorithms

This section defines the four congestion control algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery, developed in [Jac88] and [Jac90]. In some situations, it may be beneficial for a TCP sender to be more conservative than the algorithms allow; however, a TCP MUST NOT be more aggressive than the following algorithms allow (that is, MUST NOT send data when the value of cwnd computed by the following algorithms would not allow the data to be sent).

Also, note that the algorithms specified in this document work in terms of using loss as the signal of congestion. Explicit Congestion Notification (ECN) could also be used as specified in [RFC3168].

3.1. Slow Start and Congestion Avoidance

The slow start and congestion avoidance algorithms MUST be used by a TCP sender to control the amount of outstanding data being injected into the network. To implement these algorithms, two variables are added to the TCP per-connection state. The congestion window (cwnd) is a sender-side limit on the amount of data the sender can transmit

into the network before receiving an acknowledgment (ACK), while the receiver's advertised window (rwnd) is a receiver-side limit on the amount of outstanding data. The minimum of cwnd and rwnd governs data transmission.

Another state variable, the slow start threshold (ssthresh), is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission, as discussed below.

Beginning transmission into a network with unknown conditions requires TCP to slowly probe the network to determine the available capacity, in order to avoid congesting the network with an inappropriately large burst of data. The slow start algorithm is used for this purpose at the beginning of a transfer, or after repairing loss detected by the retransmission timer. Slow start additionally serves to start the "ACK clock" used by the TCP sender to release data into the network in the slow start, congestion avoidance, and loss recovery algorithms.

IW, the initial value of cwnd, MUST be set using the following guidelines as an upper bound.

```
If SMSS > 2190 bytes:
    IW = 2 * SMSS bytes and MUST NOT be more than 2 segments
If (SMSS > 1095 bytes) and (SMSS <= 2190 bytes):
    IW = 3 * SMSS bytes and MUST NOT be more than 3 segments
if SMSS <= 1095 bytes:
    IW = 4 * SMSS bytes and MUST NOT be more than 4 segments
```

As specified in [RFC3390], the SYN/ACK and the acknowledgment of the SYN/ACK MUST NOT increase the size of the congestion window. Further, if the SYN or SYN/ACK is lost, the initial window used by a sender after a correctly transmitted SYN MUST be one segment consisting of at most SMSS bytes.

A detailed rationale and discussion of the IW setting is provided in [RFC3390].

When initial congestion windows of more than one segment are implemented along with Path MTU Discovery [RFC1191], and the MSS being used is found to be too large, the congestion window cwnd SHOULD be reduced to prevent large bursts of smaller segments. Specifically, cwnd SHOULD be reduced by the ratio of the old segment size to the new segment size.

The initial value of ssthresh SHOULD be set arbitrarily high (e.g., to the size of the largest possible advertised window), but ssthresh MUST be reduced in response to congestion. Setting ssthresh as high as possible allows the network conditions, rather than some arbitrary host limit, to dictate the sending rate. In cases where the end systems have a solid understanding of the network path, more carefully setting the initial ssthresh value may have merit (e.g., such that the end host does not create congestion along the path).

The slow start algorithm is used when cwnd < ssthresh, while the congestion avoidance algorithm is used when cwnd > ssthresh. When cwnd and ssthresh are equal, the sender may use either slow start or congestion avoidance.

During slow start, a TCP increments cwnd by at most SMSS bytes for each ACK received that cumulatively acknowledges new data. Slow start ends when cwnd exceeds ssthresh (or, optionally, when it reaches it, as noted above) or when congestion is observed. While traditionally TCP implementations have increased cwnd by precisely

SMSS bytes upon receipt of an ACK covering new data, we RECOMMEND that TCP implementations increase cwnd, per:

```
cwnd += min (N, SMSS)                    (2)
```

where N is the number of previously unacknowledged bytes acknowledged in the incoming ACK. This adjustment is part of Appropriate Byte Counting [RFC3465] and provides robustness against misbehaving receivers that may attempt to induce a sender to artificially inflate cwnd using a mechanism known as "ACK Division" [SCWA99]. ACK Division consists of a receiver sending multiple ACKs for a single TCP data segment, each acknowledging only a portion of its data. A TCP that increments cwnd by SMSS for each such ACK will inappropriately inflate the amount of data injected into the network.

During congestion avoidance, cwnd is incremented by roughly 1 full-sized segment per round-trip time (RTT). Congestion avoidance continues until congestion is detected. The basic guidelines for incrementing cwnd during congestion avoidance are:

* MAY increment cwnd by SMSS bytes

* SHOULD increment cwnd per equation (2) once per RTT

* MUST NOT increment cwnd by more than SMSS bytes

We note that [RFC3465] allows for cwnd increases of more than SMSS bytes for incoming acknowledgments during slow start on an experimental basis; however, such behavior is not allowed as part of the standard.

The RECOMMENDED way to increase cwnd during congestion avoidance is to count the number of bytes that have been acknowledged by ACKs for new data. (A drawback of this implementation is that it requires maintaining an additional state variable.) When the number of bytes acknowledged reaches cwnd, then cwnd can be incremented by up to SMSS bytes. Note that during congestion avoidance, cwnd MUST NOT be increased by more than SMSS bytes per RTT. This method both allows TCPs to increase cwnd by one segment per RTT in the face of delayed ACKs and provides robustness against ACK Division attacks.

Another common formula that a TCP MAY use to update cwnd during congestion avoidance is given in equation (3):

```
cwnd += SMSS*SMSS/cwnd                   (3)
```

This adjustment is executed on every incoming ACK that acknowledges new data. Equation (3) provides an acceptable approximation to the underlying principle of increasing cwnd by 1 full-sized segment per RTT. (Note that for a connection in which the receiver is acknowledging every-other packet, (3) is less aggressive than allowed -- roughly increasing cwnd every second RTT.)

Implementation Note: Since integer arithmetic is usually used in TCP implementations, the formula given in equation (3) can fail to increase cwnd when the congestion window is larger than SMSS*SMSS. If the above formula yields 0, the result SHOULD be rounded up to 1 byte.

Implementation Note: Older implementations have an additional additive constant on the right-hand side of equation (3). This is incorrect and can actually lead to diminished performance [RFC2525].

Implementation Note: Some implementations maintain cwnd in units of

bytes, while others in units of full-sized segments. The latter will find equation (3) difficult to use, and may prefer to use the counting approach discussed in the previous paragraph.

When a TCP sender detects segment loss using the retransmission timer and the given segment has not yet been resent by way of the retransmission timer, the value of ssthresh MUST be set to no more than the value given in equation (4):

ssthresh = max (FlightSize / 2, 2*SMSS) (4)

where, as discussed above, FlightSize is the amount of outstanding data in the network.

On the other hand, when a TCP sender detects segment loss using the retransmission timer and the given segment has already been retransmitted by way of the retransmission timer at least once, the value of ssthresh is held constant.

Implementation Note: An easy mistake to make is to simply use cwnd, rather than FlightSize, which in some implementations may incidentally increase well beyond rwnd.

Furthermore, upon a timeout (as specified in [RFC2988]) cwnd MUST be set to no more than the loss window, LW, which equals 1 full-sized segment (regardless of the value of IW). Therefore, after retransmitting the dropped segment the TCP sender uses the slow start algorithm to increase the window from 1 full-sized segment to the new value of ssthresh, at which point congestion avoidance again takes over.

As shown in [FF96] and [RFC3782], slow-start-based loss recovery after a timeout can cause spurious retransmissions that trigger duplicate acknowledgments. The reaction to the arrival of these duplicate ACKs in TCP implementations varies widely. This document does not specify how to treat such acknowledgments, but does note this as an area that may benefit from additional attention, experimentation and specification.

3.2. Fast Retransmit/Fast Recovery

A TCP receiver SHOULD send an immediate duplicate ACK when an out-of-order segment arrives. The purpose of this ACK is to inform the sender that a segment was received out-of-order and which sequence number is expected. From the sender's perspective, duplicate ACKs can be caused by a number of network problems. First, they can be caused by dropped segments. In this case, all segments after the dropped segment will trigger duplicate ACKs until the loss is repaired. Second, duplicate ACKs can be caused by the re-ordering of data segments by the network (not a rare event along some network paths [Pax97]). Finally, duplicate ACKs can be caused by replication of ACK or data segments by the network. In addition, a TCP receiver SHOULD send an immediate ACK when the incoming segment fills in all or part of a gap in the sequence space. This will generate more timely information for a sender recovering from a loss through a retransmission timeout, a fast retransmit, or an advanced loss recovery algorithm, as outlined in section 4.3.

The TCP sender SHOULD use the "fast retransmit" algorithm to detect and repair loss, based on incoming duplicate ACKs. The fast retransmit algorithm uses the arrival of 3 duplicate ACKs (as defined in section 2, without any intervening ACKs which move SND.UNA) as an

indication that a segment has been lost. After receiving 3 duplicate ACKs, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.

After the fast retransmit algorithm sends what appears to be the missing segment, the "fast recovery" algorithm governs the transmission of new data until a non-duplicate ACK arrives. The reason for not performing slow start is that the receipt of the duplicate ACKs not only indicates that a segment has been lost, but also that segments are most likely leaving the network (although a massive segment duplication by the network can invalidate this conclusion). In other words, since the receiver can only generate a duplicate ACK when a segment has arrived, that segment has left the network and is in the receiver's buffer, so we know it is no longer consuming network resources. Furthermore, since the ACK "clock" [Jac88] is preserved, the TCP sender can continue to transmit new segments (although transmission must continue using a reduced cwnd, since loss is an indication of congestion).

The fast retransmit and fast recovery algorithms are implemented together as follows.

1. On the first and second duplicate ACKs received at a sender, a TCP SHOULD send a segment of previously unsent data per [RFC3042] provided that the receiver's advertised window allows, the total FlightSize would remain less than or equal to cwnd plus 2*SMSS, and that new data is available for transmission. Further, the TCP sender MUST NOT change cwnd to reflect these two segments [RFC3042]. Note that a sender using SACK [RFC2018] MUST NOT send new data unless the incoming duplicate acknowledgment contains new SACK information.

2. When the third duplicate ACK is received, a TCP MUST set ssthresh to no more than the value given in equation (4). When [RFC3042] is in use, additional data sent in limited transmit MUST NOT be included in this calculation.

3. The lost segment starting at SND.UNA MUST be retransmitted and cwnd set to ssthresh plus 3*SMSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.

4. For each additional duplicate ACK received (after the third), cwnd MUST be incremented by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.

   Note: [SCWA99] discusses a receiver-based attack whereby many bogus duplicate ACKs are sent to the data sender in order to artificially inflate cwnd and cause a higher than appropriate sending rate to be used. A TCP MAY therefore limit the number of times cwnd is artificially inflated during loss recovery to the number of outstanding segments (or, an approximation thereof).

   Note: When an advanced loss recovery mechanism (such as outlined in section 4.3) is not in use, this increase in FlightSize can cause equation (4) to slightly inflate cwnd and ssthresh, as some of the segments between SND.UNA and SND.NXT are assumed to have left the network but are still reflected in FlightSize.

5. When previously unsent data is available and the new value of cwnd and the receiver's advertised window allow, a TCP SHOULD send 1*SMSS bytes of previously unsent data.

6. When the next ACK arrives that acknowledges previously unacknowledged data, a TCP MUST set cwnd to ssthresh (the value set in step 2). This is termed "deflating" the window.

   This ACK should be the acknowledgment elicited by the retransmission from step 3, one RTT after the retransmission (though it may arrive sooner in the presence of significant out-of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost.

Note: This algorithm is known to generally not recover efficiently from multiple losses in a single flight of packets [FF96]. Section 4.3 below addresses such cases.

(…)