TCP/IP Guide

Author : Tony Hill
Date : 2nd April 2013
Version : v1-0
OS : Windows Vista Home Premium SP2

INDEX

1 Introduction 3
2 TCP/IP Features & Functionality 3
2.1 Ethernet Frame 3
2.2 TCP Connection Establishment Parameters 3
2.3 TCP Finite State Machine 5
2.4 TCP Byte Stream 6
2.5 Bandwidth Delay Product 7
2.6 Receive Window 9
2.7 Receive Window Scaling 10
2.8 TCP Sliding Window 10
2.9 Congestion Control 11
2.9.1 Slow Start 11
2.9.2 Congestion Avoidance 12
2.9.3 Send Rate Decrease 13
2.9.4 Fast Recovery 13
2.9.5 Windows NG TCP & CTCP 14
2.9.6 Graphical Output of Slow Start 14
2.10 ACK Schemes 16
2.10.1 Early ACK Mechanism & Retransmission 16
2.10.2 Cumulative ACK Mechanism 18
2.10.3 Delayed ACK Mechanism 19
2.10.4 Selective ACK Mechanism 21
2.10.5 Duplicate Selective ACK 22
2.11 RTO Calculation 23
2.11.1 Initial Measurement 23
2.11.2 Subsequent Measurements 24
2.11.3 Sample RTO Calculations 24
2.12 Timestamps 25
2.12.1 PAWS 26
2.12.2 RTTM 26
2.13 ECN 27
2.13.1 ECN Overview 27
2.13.2 ECN Packets 28
2.13.3 ECN Connection Establishment 29
2.13.4 ECN Bit Settings 29
2.13.5 ECN Nonce 30
2.13.6 Adaptive Queue Management (WRED) 31

1 Introduction

After several years of working with TCP/IP in one form or another I have accumulated lots of notes and disparate pieces of information on the subject. To be honest, I had forgotten a bit about some of the more esoteric aspects of the protocol. During a recent exercise when trying to get the Windows half of my dual-boot laptop to browse the web as rapidly and efficiently as the Linux half, I had to revisit some of the mechanisms and re-read some of the RFCs. For future reference, I decided to document in one place everything that I had looked into. Having started to do this it dawned on me that some of this information might be useful to others - hence this paper.

Please feel free to disseminate it as you see fit but, more importantly, don't hesitate to let me know via the contact tab on my web site if you encounter any glaring errors or if there is anything that I have not explained clearly.

The result of my Windows tuning exercise?

Linux is still a fair bit quicker due, in part, to Windows being more of a black box with fewer configurable TCP parameters in the registry. Not to mention new mechanisms like auto tuning. However, I have managed to improve response times by around 20% to 30%, so will settle for that, for now anyway.

2 TCP/IP Features & Functionality

2.1 Ethernet Frame

There are several types of Ethernet frame - Ethernet II, 802.3, 802.2 LLC and SNAP. However, the Ethernet II frame is the most widely used in today's LAN and Internet networks. Figure 2-1 below shows the Ethernet II frame without VLAN tags, which, if present, would consist of one or more 32-bit fields sitting in between the Source MAC and Frame Type fields. The Ethernet frame format is "Canonical" i.e. the bytes are stored left to right with each byte being transmitted right to left on the wire (LSB first).

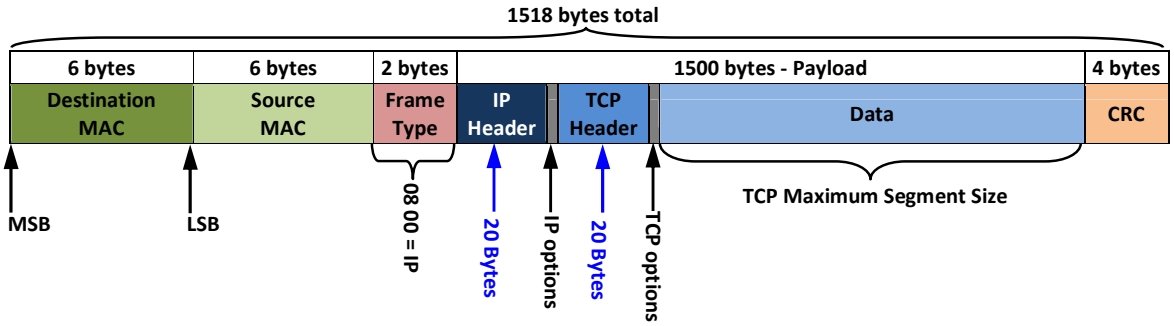


Figure 2-1: Ethernet II Frame Format

This paper makes several references to those aspects of the frame that relate to the IP and TCP protocols, namely IP and TCP header, TCP Options, Maximum Transmission Unit (MTU) and Maximum Segment Size (MSS).

2.2 TCP Connection Establishment Parameters

When Host A wishes to communicate with Host B it establishes a TCP connection using the "three-way handshake".

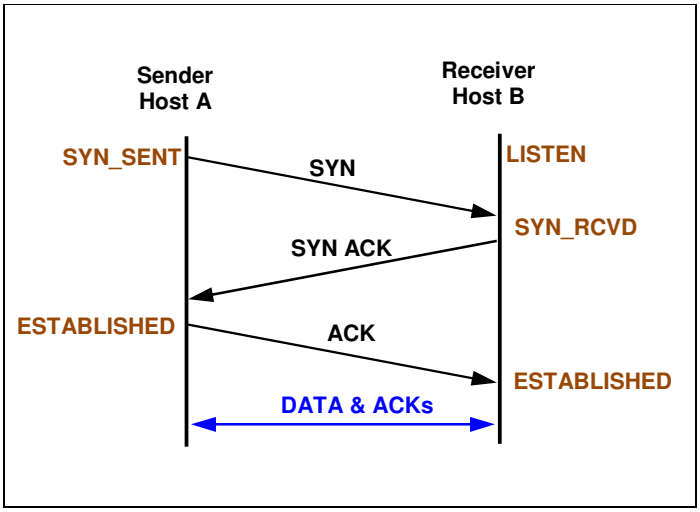


Figure 2-2: TCP Three-Way Handshake

Each host conveys important information in the TCP header and TCP options fields of the SYN packet that determines the behaviour of the connection. Examples of this information are:

- Window Size
- TCP Options
  - Maximum Segment Size
  - Window Scale
  - TCP SACK (Selective Acknowledgement)
  - Timestamps

```
⊟ Transmission Control Protocol, Src Port: 49682 (49682), Dst Port: http (80), Seq: 0, Len: 0
    Source port: 49682 (49682)
    Destination port: http (80)
    [Stream index: 3]
    Sequence number: 0    (relative sequence number)
    Header length: 32 bytes
  ⊟ Flags: 0x02 (SYN)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...0 .... = Acknowledgement: Not set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    ⊞ .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
    Window size: 8192
  ⊞ Checksum: 0x9e47 [validation disabled]
  ⊟ Options: (12 bytes)
      Maximum segment size: 1260 bytes
      NOP
      Window scale: 2 (multiply by 4)
      NOP
      NOP
      TCP SACK Permitted Option: True
```

Figure 2-3: Host A SYN Packet with Options

The initial window size of 8,192 bytes tells Host B that Host A is able to receive and buffer this many bytes before it sends an ACK to Host B. However, this window size may be scaled up in subsequent packet exchanges. The maximum segment size tells Host B that it should only send IP packets to Host A with a TCP payload content of 1,260 bytes (MTU - 40 explained later). Selective Acknowledgements are supported for the connection.

```
Transmission Control Protocol, Src Port: http (80), Dst Port: 49683 (49683), Seq: 0, Ack: 1, Len: 0
  Source port: http (80)
  Destination port: 49683 (49683)
  [Stream index: 2]
  Sequence number: 0    (relative sequence number)
  Acknowledgement number: 1    (relative ack number)
  Header length: 32 bytes
  Flags: 0x12 (SYN, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Nonce: Not set
    .... 0... .... = Congestion Window Reduced (CWR): Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgement: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..1. = Syn: Set
    .... .... ...0 = Fin: Not set
  Window size: 14600
  Checksum: 0x86d8 [validation disabled]
  Options: (12 bytes)
    Maximum segment size: 1412 bytes
    NOP
    NOP
    TCP SACK Permitted Option: True
    NOP
    Window scale: 7 (multiply by 128)
```

Figure 2-4: Host B SYN ACK Packet with Options

The initial window size of 14,600 bytes tells Host A that Host B is able to receive and buffer this many bytes before it sends an ACK to Host A. However, this window size may be scaled up in subsequent packet exchanges. The maximum segment size tells Host A that it should only send IP packets to Host B with a TCP payload content of 1,412 bytes (MTU - 40). Selective Acknowledgements are supported for the connection.

Figure 2-5 below shows the four-way handshake that terminates a TCP connection. Either host can terminate the connection by sending a TCP FIN packet.

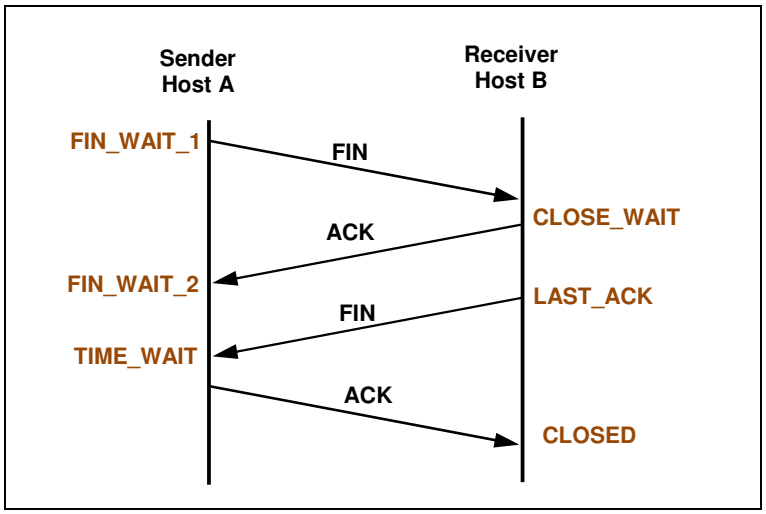


Figure 2-5: TCP Session Closedown

2.3 TCP Finite State Machine

This paper focuses on the features and congestion control mechanisms of TCP rather than the various stages and status of its connections. However, the TCP FSM is shown below in Figure 2-6 for completeness.

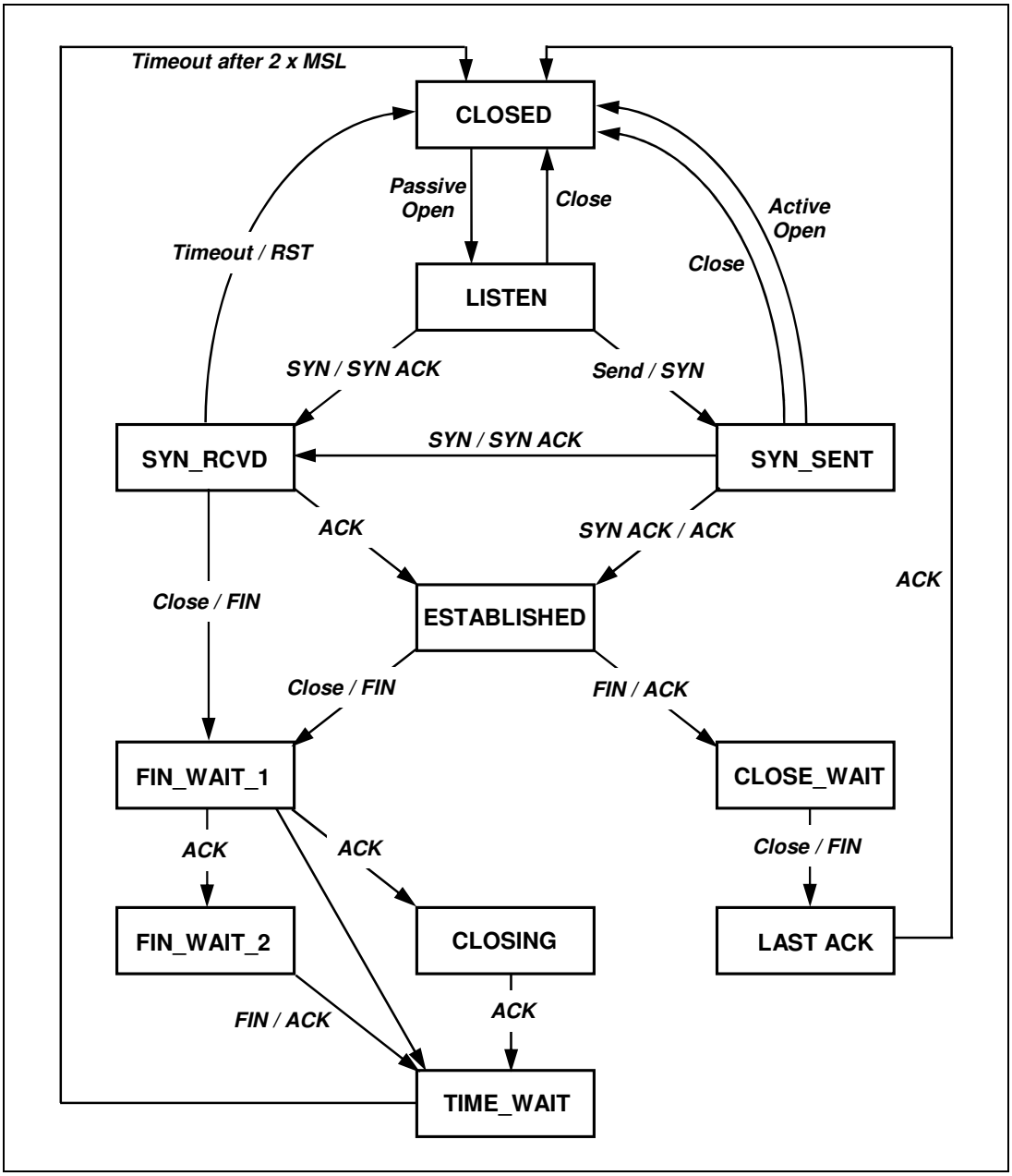


Figure 2-6: TCP FSM

2.4 TCP Byte Stream

A TCP host transmits data in streams of consecutive bytes. It "chunks" the bytes and encapsulates them in TCP segments each of which can be up to the maximum segment size specified by the receiving host in the SYN / SYN ACK exchange. TCP uses sequence numbers to keep track of the individual bytes that are transmitted. The receiving host acknowledges the receipt of bytes in the stream by specifying the sequence number the next expected byte.

The following packet exchange shows Host A issuing a TCP HTTP GET request to Host B, a web server. Host A's acknowledgements to Host B's sequence numbers are shown in blue and Host B's acknowledgements to Host A's sequence numbers are shown in red. Please note that ACK packets have a length of 0 bytes and that only data segments cause the next expected byte value to increment.

[Host A	Host B	Description]

```
Seq 1          GET                          Host A sends HTTP GET byte 1
Len 608   -------------->                   Host A  GET segment length = 608 bytes
Ack 1                                       Host A next expected Data byte 1 from Host B

               ACK          Seq 1           Host B byte sequence 1
          <--------------   Len 0           Host B ACK segment length = 0 bytes (ACK is not data)
                            Ack 609         Host B next expected Data byte 609 from Host A

               DATA         Seq 1           Host B byte sequence 1 of Data Stream
          <--------------   Len 395         Host B Data segment length = 395 bytes
                            Ack 609         Host B next expected Data byte 609 from Host A

               DATA         Seq 396         Host B byte sequence 395 + 1 of Data Stream
          <--------------   Len 1260        Host B Data segment length = 1260 bytes
                            Ack 609         Host B next expected Data byte 609 from Host A

Seq 609        ACK                          Host A byte sequence 609
Len 0     -------------->                   Host A ACK length = 0 bytes (ACK not data)
Ack 1656                                    Host A next expected Data byte 395 + 1260 +1 from Host B

               DATA         Seq 1656        Host B byte sequence1656 of Data Stream
          <--------------   Len 1260        Host B Data segment length = 1260 bytes
                            Ack 609         Host B next expected Data byte 609 from Host A

               DATA         Seq 2916        Host B byte 1656 + 1260 of Data Stream
          <--------------   Len 1260        Host B Data segment length = 1260 bytes
                            Ack 609         Host B next expected Data byte 609 from Host A

Seq 609        ACK                          Host A byte sequence 609
Len 0     -------------->                   Host A ACK length = 0 bytes (ACK not data)
Ack 4177                                    Host A next expected Data byte 2916 + 1260 +1 from Host B

Etc.
```

2.5 Bandwidth Delay Product

The BDP is a theoretical calculation yielding the potential throughput in bytes of the path between two hosts. Throughput is maximised when both hosts transmit as much data as the path between them allows. Ironically, the increase in bandwidth of transmission media has created some challenges rather than solved all the problems for the developers of TCP. TCP's transmission and congestion management mechanisms needed (and continue) to be optimised regularly to enable it to make full and efficient use of the bandwidth that is available.

BDP is calculated by multiplying the bandwidth of the path by the latency for packets of a given size that are transmitted over that path.

BDP in bytes = (BW Kbps / 8) * RTT ms

Amongst other things, the BDP indicates the amount of buffer space a receiving host should reserve to receive all of the in-transit data for a connection; the buffer space being the TCP Receive Window (RWND).

The following example is a manual BDP calculation to demonstrate how RWND can be estimated for a client PC that accesses a web server to retrieve its landing page. The connection to the internet is over VDSL running at approximately 20Mbps.

- Determine the path MTU between the client host and web server using ICMP pings
- Ascertain the Maximum Round Trip Time (RTT) of the path using ICMP pings
- Calculate the BDP and determine the RWND in bytes

The client PC's IP MTU = 1,492 bytes. Therefore, the ping length needs to be **1,464** bytes (1,492 IP MTU - 20 byte IP header - 8 byte ICMP header = 1,464).

Confirm the maximum IP size that all devices along the path are able to forward without fragmentation by pinging the target host using the above ping length with the IP do not fragment (DF) bit set.

```
C:\Users\comet>ping -f -l 1464 www.google.com
Pinging www.google.com [74.125.132.106] with 1464 bytes of data:
Reply from 74.125.132.106: bytes=64 (sent 1464) time=223ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=21ms TTL=44
```

To prove that this is the IP MTU of the path, increase the length of the ping packet by one byte to 1,465 and ping again.

```
C:\Users\comet>ping -f -l 1465 www.google.com
Pinging www.google.com [74.125.132.106] with 1465 bytes of data:
Packet needs to be fragmented but DF set.
Packet needs to be fragmented but DF set.
```

The ping fails because a device in the path reports that it cannot transmit the packet without fragmenting it. So, 1,492 bytes is indeed the MTU of the path.

Send 10 x maximum size ping packets to the target host to determine the maximum Round Trip Time (RTT). Please note that the RTT of the first ping may be a lot longer due to the PC sending an ARP request to the default gateway. Send a few pings, cancel and send a further 10 x pings to obtain a representative value of RTT.

```
C:\Users\comet>ping -f -l 1464 www.google.com -n 10
Pinging www.google.com [74.125.132.106] with 1464 bytes of data:
Reply from 74.125.132.106: bytes=64 (sent 1464) time=21ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=20ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=23ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=30ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=20ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=20ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=23ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=37ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=20ms TTL=44
Reply from 74.125.132.106: bytes=64 (sent 1464) time=21ms TTL=44
Ping statistics for 74.125.132.106:
    Packets: Sent = 10, Received = 10, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 20ms, Maximum = 37ms, Average = 23ms
```

Calculate the BDP to ascertain the effective throughput of the path i.e. the maximum number of bytes that could be in transit over the path at any given moment in time.

BDP_{bytes} = (BW Kbps / 8) * RTT ms

BDP_{bytes} = (20,000 Kbps / 8) * 37 ms

BDP_{bytes} = 2,500 bytes per second * 37 ms

BDP_{bytes} = 92,500 bytes

Determine the client PC's TCP MSS size. MSS = 1,492 IP MTU - 20 byte IP header - 20 byte TCP header = **1,452** bytes.

Calculate the client PC's RWND. By default, TCP specifies the size of the RWND using a 16-bit field, which allows it contain up to a maximum of 65,535 bytes. Determine the multiple of MSS segments that the default RWND can accommodate.

65,535 / 1,452 MSS = 45 (rounded down from 45.31)

45 * 1,452 = 65,340 bytes of RWND (un-scaled)

This calculation shows that the client PC's maximum, un-scaled RWND is not quite sufficient to accommodate the maximum BDP of the path and, therefore, throughput is not maximised. Section 2.8 describes the TCP mechanism to scale the RWND size beyond the default maximum.

For other types of connection the bandwidth and RTT may be quite different. For example, for an MTU of 1,492 bytes over an 8Mbps ADSL connection with a maximum latency of 200ms the BDP calculation would be:

BDP_{bytes} = (8,000 Kbps / 8) * 200 ms

BDP_{bytes} = 1,000 bytes per second * 200 ms

BDP_{bytes} = 200,000 bytes

65,535 / 1,452 MSS = 45 (rounded down)

44 * 1,452 = 65,340 bytes

The un-scaled RWND is an order of magnitude too small to accommodate the BDP. Once again, the RWND should be scaled up to accommodate the BDP. In this case the RWND scaling factor could be 2 (2^2 = 4) thereby quadrupling the default RWND size.

65,430 x 2^1 = 130,860 bytes

65,430 x 2^2 = 261,720 bytes

2.6 Receive Window

A host specifies its RWND size in the TCP packets it transmits to the remote host. The RWND is the number of bytes the receiving host can store in its receive buffer before it must send an acknowledgement to the sending host.

The un-scaled RWND size is specified using a 16-bit value so its maximum size can be up to 65,535 bytes. The receiving host can increase or decrease its RWND during the connection to reflect achievable throughput. However, neither host has any knowledge of the underlying network conditions when the connection is initiated so the initiating host specifies a default starting RWND value. In the Microsoft NG TCP stack, the initiating host usually specifies an initial RWND size of 8,192 bytes.

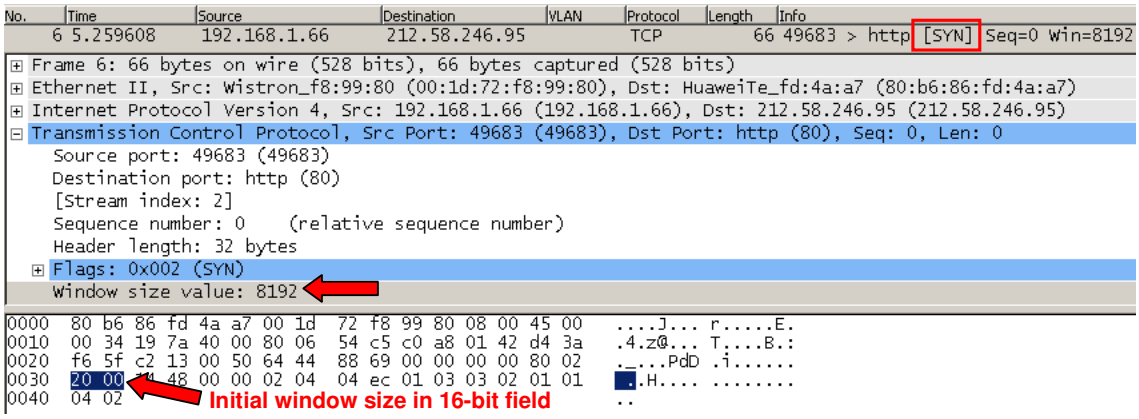


Figure 2-7: Initial RWND Size

Rapid ACK reception indicates that latency of the underlying network must be low. If a host calculates rapid RTT values, even after only a single packet has been exchanged, it informs the other host that it has increased its RWND. In Figure 2-8 below, the initiating host specifies in its TCP SYN packet an initial RWND of 8,192 bytes and receives a SYN ACK in around 11ms. It immediately increases its RWND from 8,192 to 66,780 bytes to increase throughput and informs the other host.

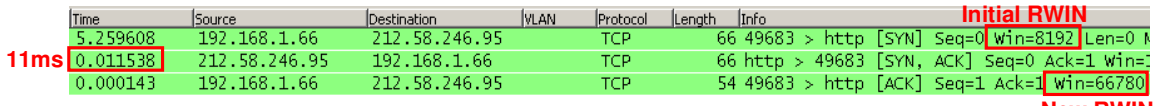


Figure 2-8: RWND Increase

Conversely, a host may decrease its RWND if network conditions deteriorate, indicated by slower ACK RTTs.

Please note: Although it is important for RWND to be sized appropriately there are many other factors that govern the throughput of a TCP connection - the acknowledgement scheme and sender's Congestion Window (CWND), for example, amongst other things. These and the "other things" are described in detail later on.

2.7 Receive Window Scaling

RFC1323 describes the Window Scaling mechanism. The receiving host specifies its initial RWND using a 16-bit field in the TCP header, which allows a maximum RWND size of 65,535 bytes.

In Figure 2-8 above, the receiving host increased its RWND to 66,780 bytes, but this is above the maximum value denoted by a 16-bit field. It does this using a TCP header option. TCP options can be up to 40 bytes in length (only 12 bytes in this example) and are identified by the option “Kind” byte.

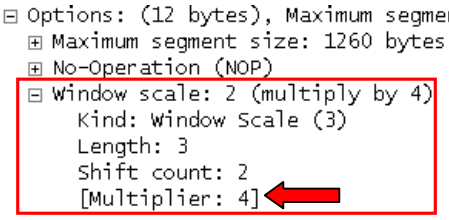


Figure 2-9: Window Scale Multiplier

The Window Scale option is a “Kind 3”, 3-byte field with a length byte = 3 and a shift count byte. For this connection, the RWND may be scaled by up to 4 times [Multiplier: 4].

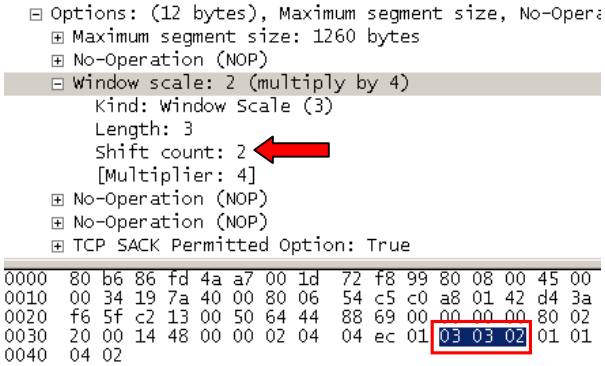


Figure 2-10: Window Scale Field

The Shift Count byte (value 2) specifies the binary shift to the left required to obtain the scaling factor i.e. $2^0 = 1$, shift left one place $2^1 = 2$, shift left two places $2^2 = 4$. In this example the shift count is 2 so the host’s RWND may be scaled by up to $2^2 = 4$ times.

After transmitting a SYN packet with scaling factor 4 and initial RWND of 8,192 bytes, the initiating host receives a SYN ACK packet. The initiating host then completes the three-way handshake with an ACK packet. The ACK packet contains a RWND of 16,695 bytes, which scaled up 4 times = 66,780 bytes.

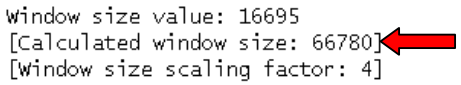


Figure 2-11: Window Scale Calculation

2.8 TCP Sliding Window

The sliding window mechanism is one of the fundamental parts of TCP protocol operation. Used in conjunction with RWND and an acknowledgement scheme it provides dynamic flow control that adapts to changing conditions in the underlying network.

The sequence below demonstrates how the sliding window operates for Host A sending TCP segments to Host B. For the purpose of illustration, Host B has specified to Host A an MSS of 100 bytes with a RWND of 400 bytes. Host A can transmit up to 4 x segments at once to “fill the pipe” and keep Host B’s RWND replete. Assume that the acknowledgement scheme dictates that Host B must acknowledge every second segment. Remember that the ACKs contain the value of the next expected byte in the next segment.

```
[          Host A                                                Host B              ]
```

```
                                        0 1 2 3
[0 1 2 3] 4 5 6 7 8 9 10 11 12 ----------------> [0 1 2 3] 4 5 6 7 8 9 10 11 12 13
                                      ACK 2
                                <---------------- 0 1 [2 3 4 5] 6 7 8 9 10 11 12 13
                                      ACK 4
                                <---------------- 0 1 2 3 [4 5 6 7] 8 9 10 11 12 13
                                    4 5 6 7
0 1 2 3 [4 5 6 7] 8 9 10 11 12 ----------------> 0 1 2 3 [4 5 6 7] 8 9 10 11 12 13
                                      ACK 6
                                <---------------- 0 1 2 3 4 5 [6 7 8 9] 10 11 12 13
                                 ACK 8 [WIN = 3]
                                <---------------- 0 1 2 3 4 5 6 7 [8 9 10] 11 12 13
                                     8 9 10
0 1 2 3 4 5 6 7 [8 9 10] 11 12 ----------------> 0 1 2 3 4 5 6 7 [8 9 10] 11 12 13
                                   ACK 10 LOST
                                <-------X------- 0 1 2 3 4 5 6 7 8 9 [10 11 12] 13
                                     ACK 11
                                <---------------- 0 1 2 3 4 5 6 7 8 9 10 [11 12 13]
                               RETRANSMIT 8 9 10
0 1 2 3 4 5 6 7 [8 9 10] 11 12 ----------------> 0 1 2 3 4 5 6 7 8 9 10 [11 12 13]
                                     ACK 12
                                <---------------- 0 1 2 3 4 5 6 7 8 9 10 11 [12 13..]
Etc.
```

Overall throughput decreases when Host B reduces its RWND to 300 bytes (3 x segments). When ACK 10 (which acknowledges the bytes in segments 8 & 9) is lost, Host A must retransmit the entire window even though it receives an ACK for the bytes in segment 11.

Host A and Host B keep track of exactly how many bytes (segments) have been sent, how many have been received and how many have been acknowledged. Figure 2-12 below shows the window from Host A's view of the world.

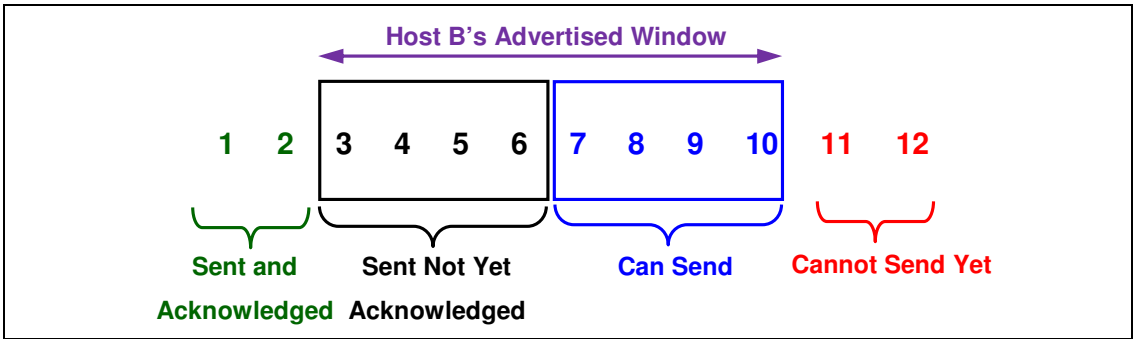


Figure 2-12: Sender's Sliding Window

2.9 Congestion Control

TCP must also control the amount of data a sender is able to transmit into the network. This is particularly necessary when it starts transmitting for the first time because it does not know whether there will be congestion, or whether the underlying network is unreliable and likely to lose packets. The mechanisms described below are specified in RFC5681.

2.9.1 Slow Start

Slow start is used to control transmission when a sender initiates a new connection, and after it detects packet loss due to the expiry of its re-transmission timer. In actual fact, slow start is not as slow as the name implies. The sender increases its transmission rate fairly rapidly until it reaches the slow start threshold. Once it reaches the threshold it enters the congestion avoidance stage during which it increases transmission one segment at a time, effectively probing the network to see whether delays will start to creep in or whether packets will start getting mislaid.

This is why TCP transmission profile often looks like a saw tooth - ramp up, slow down, packet loss, rapid decrease, ramp up, slow down etc. etc.

The sender maintains two state variables - CWND (congestion window) and "ssthresh" (slow start threshold). CWND is the amount of data that the sender can transmit at any given moment in time and "ssthresh" is a threshold that determines the point at which the sender transitions from the slow start state to the congestion avoidance state. Unlike RWND, the CWND and "ssthresh" are internal TCP variables that are not advertised in TCP packet exchanges.

Before transmitting, the sender sets **ssthresh = RWND** of the receiving host.

The sender sets its CWND to a value called the Initial Window (IW), where SMSS is the Sender's Maximum Segment Size:

```
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
```

Once the three-way handshake is complete, the sender transmits using the IW determined above. Upon receipt of an ACK the sender increases CWND by adding one segment to the number of previously acknowledged segments, effectively doubling CWND each time. This is shown in the packet sequence below, where IW =1 and MSS = 1024 to illustrate.

```
[ Host A                                Host B ]
Segment-1    1024 ------------->
                  <------------- ACK 1025
Segment-2    2048 -------------> 1 previous + 1 = 2 segments allowed
Segment-3    3072 ------------->
                  <------------- ACK 3073
Segment-4    4096 -------------> 3 previous + 1 = 4 segments allowed
Segment-5    5120 ------------->
Segment-6    6144 ------------->
Segment-7    7168 ------------->
                  <------------- ACK 7169
Segment-8    8192 -------------> 7 previous + 1 = 8 segments allowed
Segment-9    9216 ------------->
Segment-10  10240 ------------->
Segment-11  11264 ------------->
Segment-12  12288 ------------->
Segment-13  13312 ------------->
Segment-14  14336 ------------->
Segment-15  15360 ------------->
                  <------------- ACK 15361
Etc.
```

This exponential (not so slow) growth in transmission continues until CWND = "ssthresh" = RWND. When this happens, the sender enters the congestion avoidance state incrementing CWND more gradually until it detects congestion and / or packet loss.

2.9.2 Congestion Avoidance

The sender enters congestion avoidance mode as soon as "ssthresh" is reached. During congestion avoidance, the sender may only increment by one the number of segments it transmits after the reception of each ACK. This provides a linear growth in CWND rather than the exponential growth of the slow start phase.

```
[ Host A                                Host B ]
Segment-1    1024 ------------->
                  <------------- ACK 1025
Segment-2    2048 -------------> 1 previous + 1 = 2 segments allowed
Segment-3    3072 ------------->
                  <------------- ACK 3073
Segment-4    4096 -------------> 2 previous + 1 = 3 segments allowed
Segment-5    5120 ------------->
Segment-6    6144 ------------->
                  <------------- ACK 6145
Segment-8    8192 -------------> 3 previous + 1 = 4 segments allowed
Segment-9    9216 ------------->
```

```
Segment-9   10240 ------------->
Segment-10  11264 ------------->
                  <------------- ACK 11265
Etc.
```

The congestion avoidance phase forces the sender to "push the boundaries" of the connection by probing the network to discover how much it can transmit until congestion and /or loss are experienced.

2.9.3 Send Rate Decrease

The sender detects congestion and possible segment loss when it is forced to re-transmit due to an ACK that doesn't arrive causing the sender's re-transmission timer to expire. When this happens, the sender reduces sharply rate of transmission into the network. The "ssthresh" variable is reduced as follows:

ssthresh = maximum of (In-flight Data / 2) or 2 x SMSS

Where "in-flight data" is the amount of un-acknowledged data in transit to the receiver and SMSS is the Sender Maximum Segment Size.

In parallel, the sender's CWND is set to a Loss Window (LW) value of 1 x MSS, irrespective of the Initial Window (IW) determined at the beginning of the slow start phase.

The sender then re-enters the slow start phase until the now lower "ssthresh" is reached. After "ssthresh" is reached the sender enters into congestion avoidance and increments its CWND linearly i.e. one MSS at a time.

2.9.4 Fast Recovery

Fast Recovery is described in RFC5681. Fast recovery is a mechanism that allows the sender to resume sending after packet loss without reverting to step one of the slow start mechanism. Fast recovery uses Fast Retransmit to re-send lost segments. The two main causes of lost segments are - (1) a QoS policy in a transit router drops packets, (2) parallel load sharing paths with different latencies cause out-of-order packet delivery.

The sender invokes fast retransmit when it receives three duplicate ACKs from the receiver and re-sets "ssthresh" to a lower value. Having recovered from the packet loss the sender then resumes where it left off, either increasing transmission exponentially until it reaches "ssthresh" or, if "ssthresh" has been reached, entering congestion avoidance.

The packet sequence below illustrates how fast recovery works in conjunction with fast retransmit. MSS = 1024 in the example.

```
[ Host A                          Host B ]
Segment-1    1024 ------------->
                  <------------- ACK 1025
Segment-2    2048 ----> LOST
Segment-3    3072 ------------->
                  <------------- ACK 2049 Where's Segment-2?
Segment-4    4096 ------------->
                  <------------- ACK 2049 1st Duplicate ACK
Segment-5    5120 ------------->
                  <------------- ACK 2049 2nd Duplicate ACK
Segment-6    6144 ------------->
                  <------------- ACK 2049 3nd Duplicate ACK

Set ssthresh = (In-flight Data / 2) or 2 x SMSS

Segment-2    2048 -------------> Immediate re-transmit before timeout

Set CWND = ssthresh + 3 x MSS (INFLATE CWND to transmit buffered data)

Segment-7    7168 ------------->
Segment-8    8192 ------------->
Segment-9    9216 ------------->
                  <------------- ACK 9217 (cumulative ACK)
```

```
Set CWND = ssthresh (DEFLATE CWND)

Segment-10 10240 -------------->
                 <-------------- ACK 10241
Etc.
```

RFC2582 specifies the “New Reno Modification to TCP Recovery”, which improves fast recovery when more than one packet is lost.

Only one packet was lost in the example packet exchange above so ACK 9217 to Segment-9 is taken as a “full” cumulative acknowledgement of all the preceding segments, including re-transmitted Segment-2. If multiple packets are lost the ACK to Segment-9 may only be a “partial” acknowledgement but the sender would only know this after further duplicate acknowledgements are received.

To improve efficiency, New Reno uses an additional variable called “recover” for the sender to keep track of transmitted segments. The highest sequence number of all the transmitted segments up to receipt of the 3rd duplicate ACK is stored in the recover variable. This would be Segment-6 in the above example.

If the sequence number of an arriving ACK matches the content of “recover”, the system knows that the ACK encompasses all of the intermediate segments for that re-transmission event. If the ACK sequence number does not match the content of “recover”, the system starts to re-transmit the intermediate segments without waiting for further duplicate ACKs to arrive.

New Reno also introduces a further modification to RFC5861. The re-transmit timer is reset after receipt of every partial ACK, which prevents slow start from kicking in due to re-transmit timer expiry.

2.9.5 Windows NG TCP & CTCP

Microsoft introduced its Compound TCP algorithm into the Vista and Windows Server 2008 NG TCP protocol stack to optimise transmission over large BDP connections. As shown in Section 2.6, large BDP values are obtained for higher latency, lower bandwidth connections.

CTCP is intended to provide the benefits of HSTCP (RFC3649 - High Speed TCP, not covered in this paper), which uses a modified algorithm to increase the senders CWND more aggressively. However, CTCP combines the conventional RTT congestion avoidance mechanism with a delay-based queuing approach to provide fairness for multiple TCP streams that are competing for bandwidth.

CTCP is enabled by default in Vista and Windows Server 2008:

```
C:\Users\comet>netsh interface tcp show global
Querying active state...

TCP Global Parameters
----------------------------------------------
Receive-Side Scaling State          : enabled
Chimney Offload State               : enabled
Receive Window Auto-Tuning Level    : normal
Add-On Congestion Control Provider  : ctcp
ECN Capability                      : disabled
RFC 1323 Timestamps                 : disabled
```

Please refer to Section 3 for an analysis of the benefits of enabling and disabling CTCP.

2.9.6 Graphical Output of Slow Start

The following graph is generated from a Wireshark packet capture of an HTTP session to a geographically distance web site. The site was chosen because its landing page has a lot of content and the site’s distance extends path latency. The graph is produced selecting the first web server TCP data segment, clicking on Statistics in the top menu bar followed by TCPStreamGraph and selecting Time-Sequence Graph (Stevens). The Y-axis is sequence number values and the X-axis time in seconds.

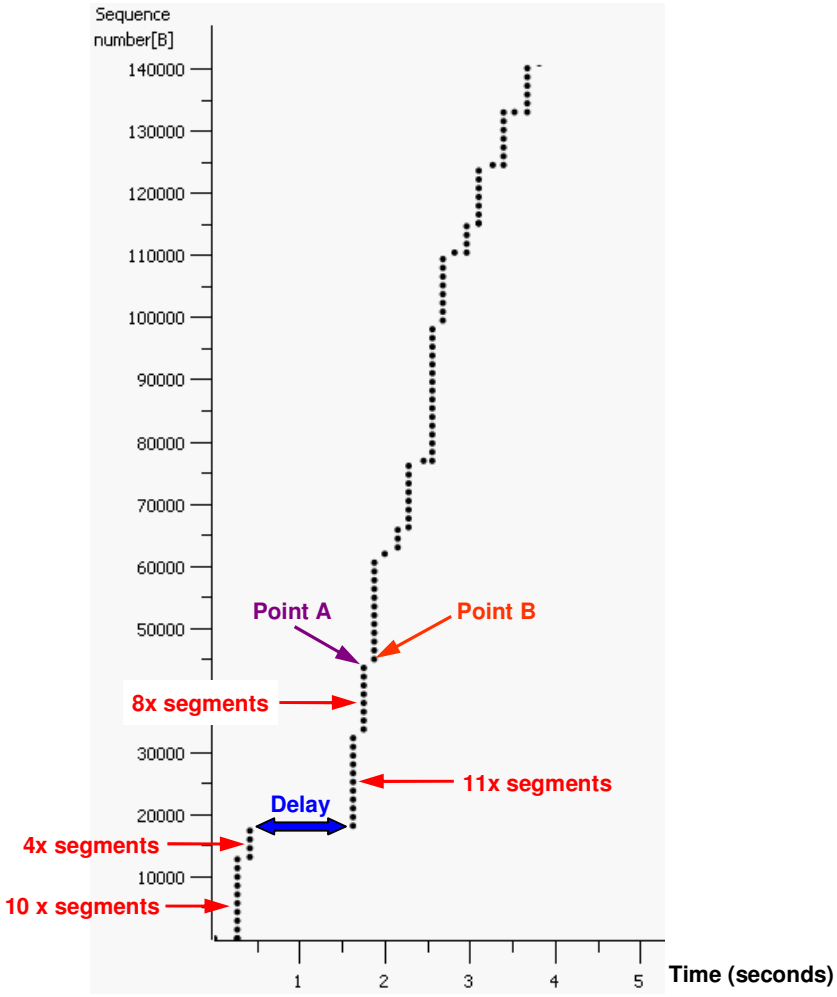


Figure 2-13: Slow Start Graph

The graph shows the sender's slow start and congestion avoidance operating - each dot is an individual packet and each group of dots a CWND "event". The first burst of activity occurs after approximately 250ms when the sender transmits 10 x segments. 100ms later the sender transmits 4 x segments. There is a delay of just over 1 second (in blue) and the sender then resumes transmitting with 11 x segments followed by 8 etc. What this graph demonstrates is that whilst all of the windowing and congestion control mechanisms we've discussed so far are definitely operating you will rarely see a smooth, tidy graph of exponential and linear growth in the real world.

Correlating this graph with the packet capture demonstrates another very important aspect of TCP. Point A (purple arrow) in the graph represents packet capture numbers 313 to 322 and Point B (orange arrow) packets 415 to 429. Please note that the packet numbers are not contiguous in the capture simply because a filter is configured to display only the packets that belong to this conversation.

Figure 2-14 below shows that the receiver acknowledges every fourth segment from the sender because this is how the delayed acknowledgement scheme is configured. Examining the time column on the left reveals that there is a jump of around 122ms between packets 322 and 415. This represents the CWND transition indicated by the arrows at Points A & B in the graph. In other words, it is equally important to understand what is happening from a timeline rather than just a segment and acknowledgement perspective.

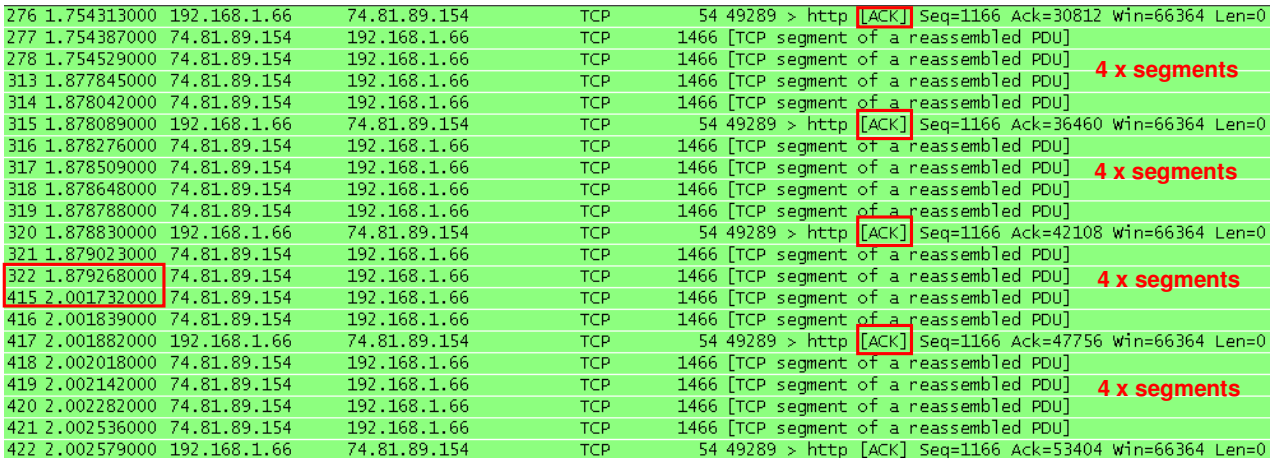


Figure 2-14: Correlation-1 of Packets with Graph

Please note: The packet capture refers to the segments as "TCP segment of a reassembled PDU". This is simply because Wireshark analyses the TCP sequence numbers and knows that each packet is part of the same stream. The packets are not fragmented at the IP layer and each packet has its own perfectly formed TCP header.

2.10 ACK Schemes

2.10.1 Early ACK Mechanism & Retransmission

One of the principal challenges in the formative days of TCP/IP was the "small packet problem". TCP must not only transport large volumes of web page or file transfer data; it must also support character-by-character applications such as Telnet. If a Telnet user types a single character which is sent immediately, the single data byte would have an overhead of 40 bytes (20 x IP header bytes + 20 x TCP header bytes). To overcome this inefficiency problem, John Nagle stipulated (1984) that a period of time should elapse before segments are sent to allow data to accumulate until either there is sufficient data to fill a segment, or an acknowledgement for a previously transmitted segment is received. TCP still follows this approach today but the big question is - how long should TCP wait before it transmits, or re-transmits, un-acknowledged data?

The user types a character on the keyboard and the first segment is sent (irrespective of whether it contains one or more bytes of data). The time interval until the corresponding ACK returns is measured - this is the Round Trip Time (RTT). The RTT is smoothed (SRTT) using a constant (α) to eliminate any large, rogue latency variations in the calculation. The SRTT is then multiplied by weighting factor (β) to yield the Re-transmission Time Out (RTO).

The formula below is from an early RTO computation [RFC793]. SRTT of the very first packet is set to the same value as the RTT.

$$\mathbf{SRTT = SRTT \times \alpha + (1 - \alpha) \times RTT_{ACK}}$$

$$\mathbf{RTO = \beta \times SRTT}$$

Where α **(smoothing constant)** = 0.875 (typically) and β **(RTO factor)** = 2. Lowering α decreases smoothing, increases RTT and consequently RTO. Increasing β increases RTO to provide more tolerance for RTT delays. Table 2-1 below shows RTO values calculated for sample RTTs that are increasing steadily.

RTT (sec)	SRTT (sec)	RTO (sec)
0.34	0.34 = 0.34	0.34 x 2 = 0.68
0.62	0.34 x 0.875 + (1 - 0.875) x 0.62 = 0.3750	0.38 x 2 = 0.76
0.76	0.38 x 0.875 + (1 - 0.875) x 0.76 = 0.4231	0.42 x 2 = 0.85

0.91	**0.42** x 0.875 + (1 - 0.875) x **0.91** = 0.4840	**0.48** x 4 = **0.97**
1.20	**0.48** x 0.875 + (1 - 0.875) x **1.20** = 0.5735	0.57 x 4 = **1.15**

Table 2-1: RFC793 RTO Algorithm

In the early TCP implementations, every segment received a corresponding ACK. When combined with the RTO calculation this provided the most reasonable way at the time to send and receive data for both file-orientated and interactive applications.

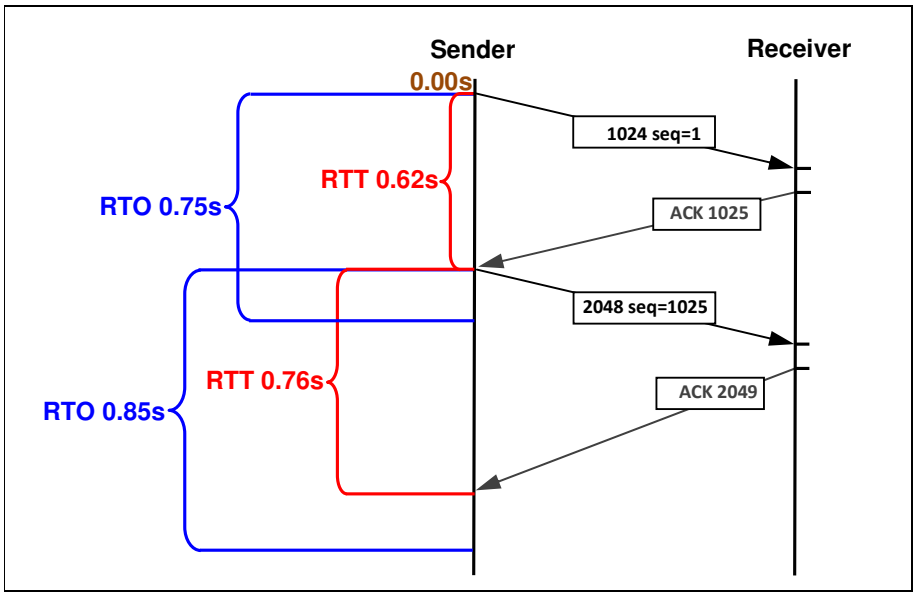


Figure 2-15: Early RTO & ACK Scheme

However, this model proved not to be resilient to frequent packet loss. Re-transmissions in particular presented the greatest difficulty. Should the RTT be measured between original data segment transmission and ACK reception or between data segment re-transmission and ACK reception?

Either of these measurements could significantly skew the RTT computation, and hence RTO.

Phil Karn (1987) proposed an alternative approach to overcome this dilemma that consisted of two rules:

- Do not compute the RTT of any re-transmitted segments and their corresponding ACKs
- Back-off (double) the RTO of each successive re-transmission drastically reduce re-transmission events

The first rule removes the ambiguity about how to measure the RTT. The second rule lengthens the RTO to such an extent that it gives the protocol more time to recover but slows its operation down considerably.

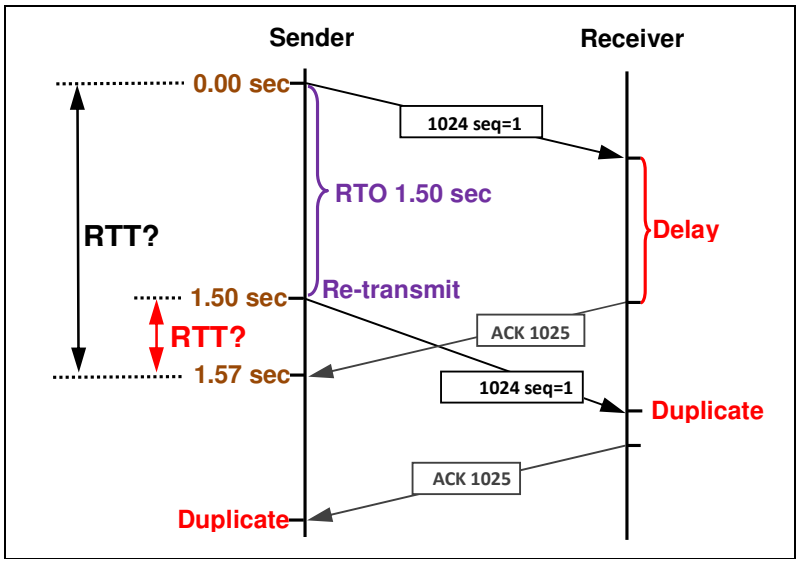


Figure 2-16: Problems with Early ACK and RTO Mechanisms

The RTT smoothing algorithm also proved not to be suitable for networks in which RTT measurements fluctuated considerably, so Jacobson / Karel produced an enhanced algorithm that introduced a RTT deviation component. With moderate levels of RTT fluctuation, deviation is small and RTO increases marginally. But, with significant levels of fluctuation the deviation is large and RTO is allowed to increase much more significantly [RFC1122]. The revised algorithm is shown below:

Difference = RTT - SRTT

SRTT = SRTT + (α x Difference)

Deviation = Deviation + (ϕ x [Difference - Deviation])

RTO = SRTT + (β x Deviation)

Where initial SRTT = RTT and initial Deviation = 0, α = 0.125, ϕ= 0.25, β = 4.

Taking the same RTT values from Table 2-1 and applying them to the calculations in Table 2-2 below the RTO increases significantly for those values where deviation is higher.

RTT (sec)	Diff (sec)	SRTT (sec)	Dev (sec)	RTO (sec)
0.34	0.00	0.34	0.00	0.34 + (4 x 0.00) = 0.34
0.62	0.62 - 0.34 = 0.28	0.34 + (0.125 x 0.28) = 0.38	0.00 + (0.25 x (0.28 - 0.00)) = 0.07	0.38 + (4 x 0.07) = 0.66
0.76	0.76 - 0.38 = 0.38	0.38 + (0.125 x 0.38) = 0.43	0.07 + (0.25 x (0.38 - 0.07)) = 0.15	0.43 + (4 x 0.15) = 1.03
0.91	0.91 - 0.43 = 0.48	0.43 + (0.125 x 0.48) = 0.49	0.15 + (0.25 x (0.48 - 0.15)) = 0.23	0.49 + (4 x 0.23) = 1.41
1.20	1.20 - 0.49 = 0.71	0.49 + (0.125 x 0.71) = 0.58	0.23 + (0.25 x (0.71 - 0.23)) = 0.35	0.58 + (4 x 0.35) = 1.98

Table 2-2: Jacobson Karel Algorithm

The following sections describe enhancements to the TCP stack that have mitigated many of these problems. Section 2.12 describes in detail the specification of the latter day RTO calculation.

2.10.2 Cumulative ACK Mechanism

The cumulative ACK mechanism is TCP's default acknowledgement method. TCP sends streams of bytes which it sends in chunks, or segments. The relationship between bytes and segments is a source of great confusion when discussing ACK schemes. The thing to bear in mind is that acknowledgements specify byte numbers rather than segment numbers so it is perfectly acceptable for a receiver to use a single acknowledgement for multiple segments, provided that it has received ALL of the preceding byes in ALL of the previous segments. If not, selective acknowledgements must be used to retrieve the missing information.

The downside of cumulative acknowledgements is that if a receiver waits too long to send an ACK and the sender's data is lost in transit, the sender has to re-transmit everything again - which could be an entire window. This is why for large MSS packets TCP generally acknowledges each segment at a time.

Figure 2-17 below illustrates the cumulative ACK mechanism with each segment carrying 100 bytes.

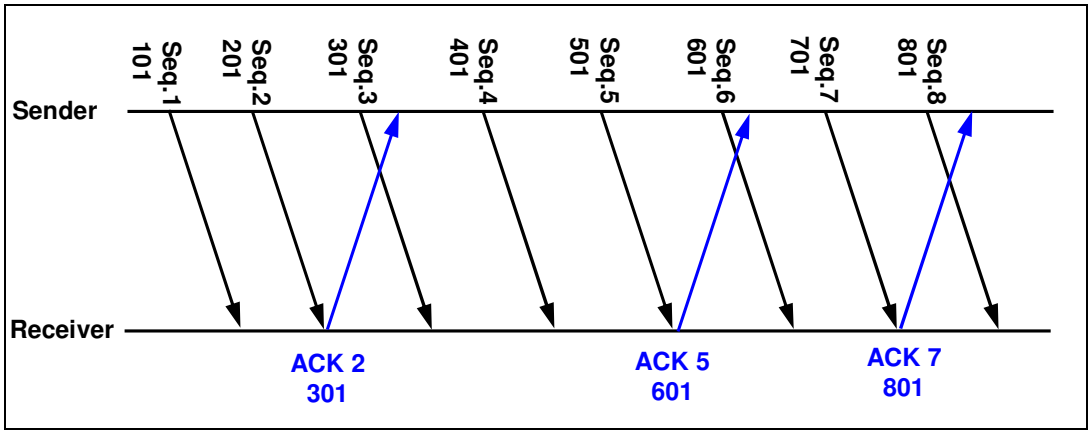


Figure 2-17: Cumulative ACK Mechanism

2.10.3 Delayed ACK Mechanism

The delayed ACK mechanism is described in RFC 1122. Delayed ACK modifies the default TCP ACK method by reducing the number of ACK packets a receiver needs to send. Instead of sending an ACK for every received segment, the receiver sends an ACK for every second full-size segment. The receiver must acknowledge receipt of the first segment within 500ms, even if the second segment has not arrived, thus preventing the receiver from waiting indefinitely for the second segment. Figure 2-18 below illustrates how delayed ACK works.

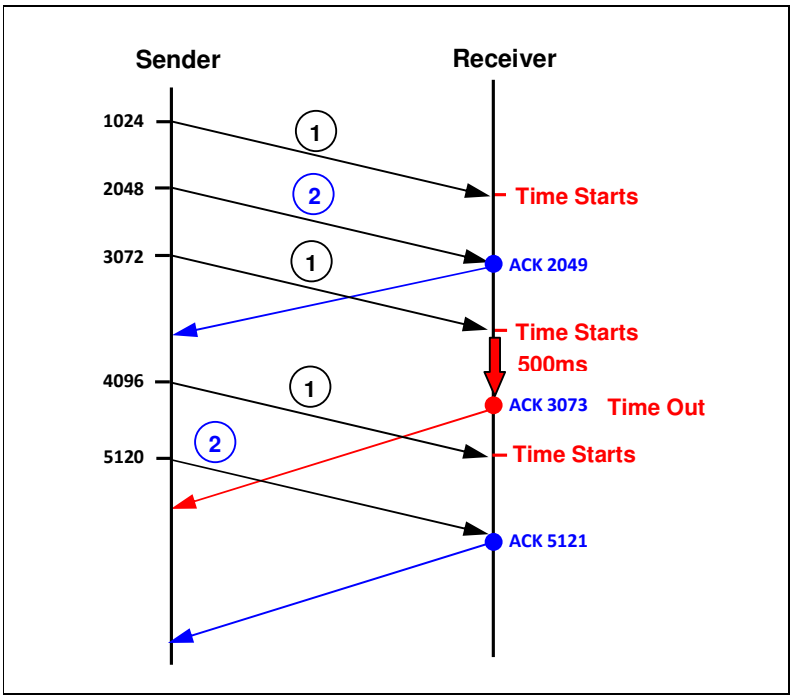


Figure 2-18: Delayed ACK as per RFC

Two registry entries govern the use of delayed ACK in the Windows NG TCP implementation:

- TcpAckFrequency (default 2 = two unacknowledged segments)

– TcpDelAckTicks (default 2 = 200ms)

TcpAckFrequency is the number of segments received after which the receiver must send an ACK and TcpDelAckTicks is the ACK timer in units of 100ms.

I haven't been able to verify this myself because I do not have the necessary test equipment. However, I have come across information that suggests that the Windows delayed ACK implementation behaves slightly differently. Apparently, the timer is not re-set upon receipt of the first full-size segment but expires regardless at regular 200ms intervals. If this is the case, the receiver will send an ACK for a segment arriving adjacent to the clock boundary irrespective of whether it is the first or second segment. This is illustrated in Figure 2-19 below.

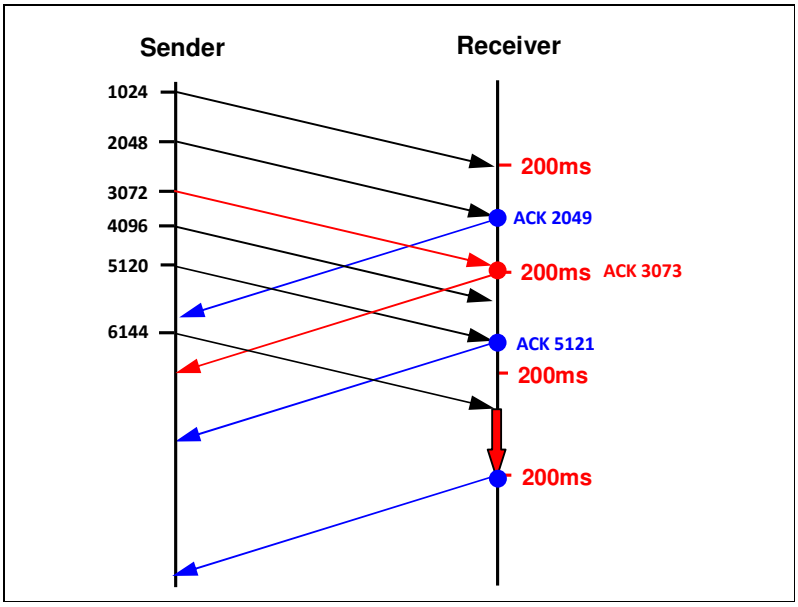


Figure 2-19: Windows Delayed ACK

Figure 2-20 below shows delayed ACK in operation with TcpAckFrequency set to 2.

Time	Source	Destination	VLAN	Protocol	Length	Info	
0.000031	192.168.1.66	212.58.246.95		TCP	54	49683 > http [ACK] Seq=609 Ack=4740 Win=66216 Len=0	
0.000021	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	2 x segments
0.000030	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	
0.000033	192.168.1.66	212.58.246.95		TCP	54	49683 > http [ACK] Seq=609 Ack=7260 Win=66780 Len=0	
0.000082	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	2 x segments
0.000107	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	
0.000035	192.168.1.66	212.58.246.95		TCP	54	49683 > http [ACK] Seq=609 Ack=9780 Win=66780 Len=0	
0.011093	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	2 x segments
0.000123	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	
0.000043	192.168.1.66	212.58.246.95		TCP	54	49683 > http [ACK] Seq=609 Ack=12300 Win=66780 Len=0	
0.000033	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	2 x segments
0.000182	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	
0.000044	192.168.1.66	212.58.246.95		TCP	54	49683 > http [ACK] Seq=609 Ack=14820 Win=66780 Len=0	
0.000053	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	2 x segments
0.000112	212.58.246.95	192.168.1.66		TCP	1314	[TCP segment of a reassembled PDU]	
0.000036	192.168.1.66	212.58.246.95		TCP	54	49683 > http [ACK] Seq=609 Ack=17340 Win=66780 Len=0	

Figure 2-20: Delayed ACK Default Settings

Figure 2-21 below shows delayed ACK in operation with TcpAckFrequency set to 4.

Time	Source	Destination	VLAN	Protocol	Length	Info
0.000047000	192.168.1.66	212.58.246.92		TCP	54	57388 > http [ACK] Seq=601 Ack=8569 Win=66364 Len=0
0.000504000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000246000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.009446000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000094000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000052000	192.168.1.66	212.58.246.92		TCP	54	57388 > http [ACK] Seq=601 Ack=14217 Win=66364 Len=0
0.000145000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000241000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000130000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000190000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000049000	192.168.1.66	212.58.246.92		TCP	54	57388 > http [ACK] Seq=601 Ack=19865 Win=66364 Len=0
0.000133000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000111000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000166000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000273000	212.58.246.92	192.168.1.66		TCP	1466	[TCP segment of a reassembled PDU]
0.000043000	192.168.1.66	212.58.246.92		TCP	54	57388 > http [ACK] Seq=601 Ack=25513 Win=60716 Len=0

4 x segments

4 x segments

4 x segments

Figure 2-21: Delayed ACK with 4 x ACKs

2.10.4 Selective ACK Mechanism

TCP's standard cumulative acknowledgement scheme is very inefficient when packets are lost. If a sender transmits 10 segments to a receiver and the first segment is lost the receiver cannot request specific re-transmission of the first segment, resulting in the sender having to re-transmit all 10 segments. With SACK, the receiver acknowledges successful receipt of the contiguous block containing segments 2 to 10, which, by implication, prompts the sender to re-transmit segment 1. Selective ACK and Duplicate Selective ACK are defined in RFC2018 and RFC2883 respectively.

Communicating hosts indicate their support of selective acknowledgement by specifying the SACK Permitted option in the TCP options field of the SYN and SYN ACK packets exchanged during the three-way handshake. The SACK permitted option is 2 bytes long [Kind =4, Length =2].

```
⊟ Options: (12 bytes), Maximum segment size, No-Opera
  ⊞ Maximum segment size: 1452 bytes
  ⊞ No-Operation (NOP)
  ⊞ Window scale: 2 (multiply by 4)
  ⊞ No-Operation (NOP)
  ⊞ No-Operation (NOP)
  ⊟ TCP SACK Permitted Option: True
      Kind: SACK Permission (4)
      Length: 2

0000  80 b6 86 fd 4a a7 00 1d  72 f8 99 80 08 00 45 00
0010  00 34 01 8d 40 00 80 06  93 61 c0 a8 01 42 4a 51
0020  59 9a c0 89 00 50 89 0c  54 42 00 00 00 00 80 02
0030  20 00 4b 20 00 00 02 04  05 ac 01 03 03 02 01 01
0040  04 02
```

Figure 2-22: SACK Permitted Option

During the session, the receiver uses ACK segments to convey SACK information to the sender. It embeds the sequence numbers of contiguous blocks of segments it has received in TCP options field [Kind =5, Length = n]. The option field contains two 32-bit numbers; one specifies the first byte of a segment being acknowledged and the other specifies the first byte of the next expected segment. In Figure 2-23 below, the SACK option is acknowledging receipt of the segment beginning with byte 12709 and requesting the next segment in the stream that begins with byte 13087.

```
⊟ Options: (12 bytes), No-Operation (NOP), No-Operat
  ⊞ No-Operation (NOP)
  ⊞ No-Operation (NOP)
  ⊟ SACK: 12709-13087
      left edge = 12709 (relative)
      right edge = 13087 (relative)
  ⊞ [SEQ/ACK analysis]

0000  80 b6 86 fd 4a a7 00 1d  72 f8 99 80 08 00 45 00
0010  00 34 01 99 40 00 80 06  93 55 c0 a8 01 42 4a 51
0020  59 9a c0 89 00 50 89 0c  55 61 39 2e 79 49 80 10
0030  3f 6e 11 49 00 00 01 01  05 0a 39 2e 7e cd 39 2e
0040  80 47
```

Figure 2-23: SACK Information Transport

The SACK mechanism does not replace but works in concert with the standard, cumulative acknowledgement scheme. The receiver continues to acknowledge receipt of the highest sequence number using cumulative ACKs and only uses the SACK option to notify the sender that it must re-

transmit previously transmitted segments. In addition, SACK operation must not interfere with periods of aggressive congestion control, such as when the fast recovery mechanism is active. In any case, transmission is often throttled back to one segment at a time during these recovery periods.

SACK uses blocks to delimit segments and bytes. The byte at the left-hand edge of a block is the sequence number of the first byte of that block. The byte at the right-hand edge is the sequence number of the first byte of the next block, following the convention of ACK values denoting the next expected byte in the stream. The sender maintains a sequence-ordered re-transmission queue containing the segments that have not yet been acknowledged. Upon receipt of a SACK packet, the sender compares the sequence numbers in the SACK block with those in its re-transmission queue and re-transmits those packets which sequence numbers are not specified in the SACK block.

The following example is based on an example in RFC 2018. A sender transmits 8 x 500 byte TCP segments to a receiver - 5000, 5500, 6000, 6500, 7000, 7500, 8000 and 8500. Segments 2, 4, 6 and 8 are lost. The 3rd, 5th and 7th packets trigger the receiver to send the following SACK options to the sender:

	Triggering Segment	ACK	First Block Left Edge	First Block Right Edge	2nd Block Left Edge	2nd Block Right Edge	3rd Block Left Edge	3rd Block Right Edge
1)	5000	5500						
2)	5500	[lost]						
3)	6000	5500	6000	6500				
4)	6500	[lost]						
5)	7000	5500	7000	7500	6000	6500		
6)	7500	[lost]						
7)	8000	5500	8000	8500	7000	7500	6000	6500
8)	8500	[lost]						

(1) The receiver acknowledges segment 5000 in the normal way by sending an ACK with next expected byte 5500.

(3) Receipt of segment 6000 triggers a SACK event because segment 5500 is missing. The receiver re-sends the ACK for segment 5500 and includes a SACK block containing left-hand block value 6000 and right-hand block value 6500. It is telling the sender that it still needs block 5500, that it is acknowledging receipt of the segment beginning with byte 6000 and now expects the segment beginning with byte 6500.

(5) Receipt of segment 7000 triggers another SACK event because segments 5500 and 6500 are missing. The receiver re-sends the ACK for segment 5500 containing two SACK blocks. The first SACK block is pre-pended to the front of the block list. In block one, the receiver acknowledges receipt of segment 7000 and expects the next segment beginning with byte 7500. In block two, the receiver acknowledges segment 6000 and expects the next segment beginning with byte 6500.

The same process is repeated for lost segments 7500 (6) and 8500 (8) with the latest block being pre-pended to the list each time.

Please note that a maximum of 4 x blocks can be specified in the options field of the TCP header.

2.10.5 Duplicate Selective ACK

Duplicate selective ACK (D-SACK) is described in RFC2883. D-SACK is an extension to the SACK functionality described in RFC2018. However, RFC2018 does not describe how the receiver uses the SACK option to signal that it has received duplicate segments. A duplicate segment arrives at a receiver when the sender re-transmits it either because the sender's RTO timer has expired due to the ACK being delayed, or because the ACK has been lost completely.

The duplicate selective ACK mechanism uses exactly the same fields and structure of the SACK option. The only difference, of course, is that the receiver knows it has received the same segment twice because it observes the same segment sequence numbers in two different packets. The receiver signals a duplicate segment to the sender using the first block of the SACK options field.

The following example is taken from the RFC. The sender transmits 4 x segments but the network drops the return ACK packets for the first two, (1) & (2). To complicate matters, the network also

drops the sender's third segment, (3). The sender's RTO timer expires because it hasn't received ACKs for (1) & (2) so it re-transmits the first segment, (5).

```
   Transmitted     Received       ACK Sent
   Segment         Segment        (Including SACK Blocks)

1) 3000-3499       3000-3499      3500 (ACK dropped)
2) 3500-3999       3500-3999      4000 (ACK dropped)
3) 4000-4499       (data packet dropped)
4) 4500-4999       4500-4999      4000, SACK =4500-5000
5) 3000-3499       3000-3499      4000, DSACK=3000-3500, SACK=4500-5000
```

(4) The receiver doesn't yet know that its ACK packets for segments (1) and (2) have been lost, or that the sender has transmitted segment (3), which has also been lost. The receiver does receive segment 4500 (4). The receiver sends a SACK packet for segment 4500 (4) re-specifying cumulative ACK 4000 and includes SACK options in the packet. The first SACK block acknowledges segment 4500 and specifies that it is expecting the next segment starting with byte 5000.

(5) In the meantime, a re-transmission of segment 3000 (5) arrives at the receiver. The receiver sends an ACK packet re-specifying cumulative ACK 4000 and includes a D-SACK option in the first block specifying that it has already received 3000 but is still expecting 3500. The second block is a normal SACK block acknowledging receipt of segment 4500 and requesting the next segment beginning with byte 5000.

2.11 RTO Calculation

The Retransmission Time Out calculation is one of the most important components of the TCP stack. Without an accurate RTO calculation, fast and efficient transmission is simply not possible. Section 2.11.1 described the evolution of RTO algorithms and the problems that developers had to overcome. RFC6298 specifies the algorithm that is used in latter day TCP stacks.

As with its immediate predecessor, the algorithm includes Smooth Round Trip Time (SRTT) and RTT variation components. These are maintained in two variables - SRTT and RTTVAR. In addition, many of the rules of previous RTO calculations are still observed e.g. Karn's rule for not sampling the RTT of re-transmitted segments and corresponding ACKs, and the doubling (back-off) of RTO when the re-transmission timer expires.

2.11.1 Initial Measurement

The RFC specifies that the sender should set RTO to 1 second until it is able to measure the RTT for the first segment. In addition, when the first RTT measurement is taken, the following rules MUST apply:

```
SRTT = RTT
RTTVAR = RTT/2
RTO = SRTT + maximum of (G, K * RTTVAR)
```

Where constant **K = 4** and **G** is the granularity of the system's clock. If K * RTTVAR = 0 then variance must be rounded to **G** seconds.

The RFC states that clock granularity is not itself a part of the RTO computation i.e. a clock granularity value is not specified anywhere in the calculation. But, clock granularity does have a significant bearing on the system's ability to measure RTT to a fine degree of accuracy. If the clock is so coarse (as it is in many older systems) that RTT measurement is poor resulting consistently in RTTVAR being zero, then the variable **G** must be used instead, but its value must reflect the granularity of the clock. For example, a system clock with a granularity of 500ms would result in RTT and SRTT measurements that were in the order of seconds rather than milliseconds and the value of RTTVAR would be meaningless. Therefore, **G** would need to be specified in seconds accordingly.

Moreover, the RFC also states that the RTO should be rounded up to 1 second even if it is calculated to be less than 1 second. This provides protocol stability and ensures consistency for communicating systems that have very different processing capabilities. So, even though systems with fine-grained

clocks would benefit in every other aspect of protocol dependent computation, they would certainly not benefit from consistently and accurately computing RTO values of less than 1 second.

Please note: Processor speed and clock granularity are two different things. For example, Linux running on a 2MHz CPU may define its frequency (granularity) as 1000Hz, which provides 10 X more fine grained accuracy than a frequency of 100Hz.

2.11.2 Subsequent Measurements

When subsequent RTT measurements are taken, the system MUST use the following algorithm:

```
RTTVAR = (1 – beta) * RTTVAR + beta * (SRTT – RTT)

SRTT = (1 – alpha) * SRTT + (alpha * RTT)
```

Where values for alpha and beta SHOULD be: **alpha = 1/8, beta = 1/4**.

If beta approaches 1, the current RTTVAR sample is predicated more on current RTT so reaction to variation is more immediate. If beta approaches 0, the historical value of variation has more influence on the calculation and reaction to variation is more gradual.

If alpha approaches 1, the current SRTT sample is predicated more on current RTT so reaction to variation is more immediate. If alpha approaches 0, the historical value of RTT has more influence on the calculation and reaction to variation is more gradual. The recommended values of alpha and beta have, presumably, been tested and optimised.

Finally, RTO is calculated using the following algorithm:

```
RTO = SRTT + max (G, K*RTTVAR)
```

2.11.3 Sample RTO Calculations

Table 2-3 below contains sample RFC6298 RTO calculations. The RTTs used are the same as those used in Tables 2-1 and 2-2. To paraphrase the RFC - the value of SRTT used in the RTTVAR calculation is the SRTT value from the previous SRTT measurement i.e. before SRTT is updated for the current RTT. Therefore, RTTVAR must be calculated first followed by SRTT in the order shown below:

beta = 0.25, alpha = 0.125
RTTVAR = (1 - beta) * RTTVAR + beta * (SRTT - RTT)
SRTT = (1 - alpha) * SRTT + (alpha * RTT)
RTO = SRTT + max (G, K*RTTVAR)

RTT (sec)	RTTVAR (sec)	SRTT (sec)	RTO (sec)
0.34	0.34 / 2 = 0.17	0.34 = 0.34	0.34 + (4 x 0.17) = **1.02**
0.62	(0.75 x 0.17) + 0.25(0.62 - 0.34) = 0.20	(0.875 x 0.34) + (0.125 x 0.62) = 0.38	0.38 + (4 x 0.20) = **1.18**
0.76	(0.75 x 0.20) + 0.25(0.76 - 0.38) = 0.25	(0.875 x 0.38) + (0.125 x 0.76) = 0.43	0.43 + (4 x 0.25) = **1.43**
0.91	(0.75 x 0.25) + 0.25(0.91 - 0.43) = 0.31	(0.875 x 0.43) + (0.125 x 0.91) = 0.49	0.49 + (4 x 0.31) = **1.73**
1.20	(0.75 x 0.31) + 0.25(1.20 - 0.49) = 0.41	(0.875 x 0.49) + (0.125 x 1.20) = 0.86	0.86 + (4 x 0.41) = **2.50**

Table 2-3: RFC6298 RTO Calculation

When comparing the RTO results in this table with those in Tables 2-1 and 2-2 it is clear that the calculation according to RFC6298 is much more conservative than for the previous algorithms. This is in keeping with the following statement in the RFC:

Traditionally, TCP implementations use coarse grain clocks to measure the RTT and trigger the RTO, which imposes a large minimum value on the RTO. Research suggests that a large minimum RTO is needed to keep TCP conservative and avoid spurious retransmissions [AP99]. Therefore, this specification requires a large minimum RTO as a conservative approach, while at the same time acknowledging that at some future point, research may show that a smaller minimum RTO is acceptable or superior.

Please note: The calculation for RTTVAR specifies **SRTT - RTT**. But, SRTT is smoothed and, therefore, generally less than RTT, which would sometimes yield a meaningless negative result for RTO. SRTT - RTT is meant to signify the difference between the two variables and is a positive value.

2.12 Timestamps

RFC1323 specifies the use of timestamps in TCP packets to provide two key features:

1) Protection Against Wrapped Sequence Numbers (PAWS)
2) Facilitation of dynamic Round Trip Time Measurement (RTTM)

Both PAWS and RTTM use the same 32-bit timestamp fields in the options part of the TCP header. The fields are called TSval (timestamp value) and TSecr (timestamp echo reply).

Sending and receiving hosts maintain their own internal system clock, each of which has its own granularity depending on the hardware and OS it is running. Granularity in this context means the OS processing frequency e.g. 100Hz, 200Hz, 1000Hz etc. rather than CPU speed.

When the sender transmits a segment it uses its clock to generate a number (timestamp) that could, for example, be the time elapsed since system boot. It inserts the number into the 32-bit TCP TSval options field of a segment and transmits the segment to the receiver. The receiver generates an ACK segment and copies the sender's TSval timestamp into the TSecr field. The receiver generates its own timestamp, inserts this into the TSval field of the ACK and transmits the ACK to the sender. When the sender receives the ACK it compares the value (that it generated originally) in the TSecr field with its current system clock and derives a fairly accurate RTT value.

This echo mechanism means that it is not necessary for the system clocks of both hosts to be synchronised.

In Figure 2-24 below, Host A generates a timestamp number (1743327258), which it inserts into the TCP TSval options field of a segment to transmit to Host B. Host A copies the latest TSval (196281) it received from Host B into the TSecr field and transmits the segment to Host B. The timestamp option is 10-bytes in length [Kind = 8, Length, TSval, TSecr].

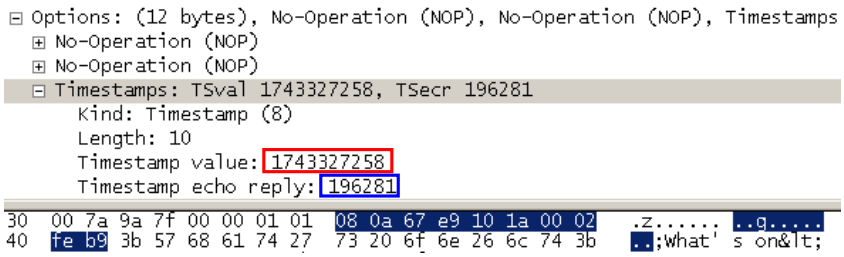


Figure 2-24: Timestamp Information Encoding

A host may transmit timestamps in the SYN segments it uses to establish connections to remote hosts. A host may only send timestamps if it receives SYN segments from other hosts that contain the timestamp option. The echo reply field (TSecr) is only valid in segments that have the ACK bit set in the TCP header.

The packet sequence below illustrates how the mechanism works starting with a Host A generated SYN packet.

```
[  Host A                                                    Host B  ]
1) TSval=100,TSecr=0,Seq.0 SYN  ------------->
2)                              <-------------- SYN-ACK 1,TSval=10,TSecr=100
3) TSval=101,TSecr=10,Seq.1 ACK ------------->
4) TSval=101,TSecr=10,Seq.101   ------------->
5)                              <-------------- ACK 201,TSval=11,TSecr=101
6) TSval=102,TSecr=11,Seq.201   ------------->
7)                              <-------------- ACK 301,TSval=12,TSecr=102
8) TSval=103,TSecr=12,Seq.301   ------------->
                                <-------------- ACK 401,TSval=13,TSecr=103
Etc.
```

In step (1), Host A sets its TSval number to 100 and establishes a connection to Host B. The TSecr is zero because this is the first segment of the connection and there is not yet anything to echo back to Host B. In step (2), Host B sets its own TSval number to 10 and echo replies Host A's timestamp in the TSecr field. At this point, or at any time later in the conversation, each host compares the echoed TSecr value with its current system clock and computes the RTT.

2.12.1 PAWS

PAWS uses the same TCP timestamps option as the RTTM mechanism described above to fulfil a different requirement. TCP sequence numbers are specified using 32-bit fields, which for high-speed, large window transfers could result in sequence number wrap-around relatively quickly.

Wrap around (Maximum Sequence Lifetime) is calculated as follows:

MSL = 2^{31} / Bytes per second

Taking some typical values for Ethernet:

Type	B/W	Bytes/Sec	MSL Secs	MSL Mins
Ethernet	10Mbps	1,250,000	1,718	28.6
Ethernet	100Mbps	12,500,000	172	2.9
Ethernet	1Gbps	125,000,000	17	0.3

Recall that BDP (throughput) is determined using the following calculation:

BDP_{bytes} = (BW Kbps / 8) * RTT ms

For a 1Gbps LAN connection assuming a RTT of 2ms, the BDP (throughput) would be:

BDP_{bytes} = (1000000 / 8) * 2

BDP_{bytes} = 125,000 * 2

BDP_{bytes} = 250,000

Even if the BDP (throughput) was half this value, it would still only be a matter of seconds until wrap around occurred.

Rather than using timestamps to determine RTT, the timestamps may be used to validate the currency of the 32-bit TCP sequence numbers. Segments received within the current “timestamp window” would be determined and accepted using the following calculation for Maximum Segment Lifetime (MSL):

MSL 2^{31} > t2 - t1 > 0

Where, t2 is the timestamp of the most recent, valid segment and t1 is the timestamp of an earlier segment that may have a sequence number that is overlapped.

2.12.2 RTTM

As described above, communicating hosts may use the TCP timestamp option to determine the Round Trip Time Measurement of segments. But, there are some rules that must be observed for the mechanism to be used successfully. The following scenarios outline these rules.

2.12.2.1 Periods of Silence

Host A transmits segments 1, 3 and 5 to Host B and then ceases transmission for a period of 50 seconds. Host B identifies the gap in transmission and does not use the received timestamp to update its RTT computation.

```
[  Host A                                                    Host B  ]
1) TSval=1,TSecr=10,Seq.101     -------------->
2)                              <-------------- ACK 201,TSval=11,TSecr=1
3) TSval=2,TSecr=11,Seq.201     -------------->
4)                              <-------------- ACK 301,TSval=12,TSecr=2
5) TSval=3,TSecr=12,Seq.301     -------------->
6)                              <-------------- ACK 401,TSval=13,TSecr=3

   [Period of 50 seconds silence]

7) TSval=53,TSecr=13,Seq.401    -------------->
                                <-------------- ACK 501,TSval=63,TSecr=53
8) TSval=54,TSecr=63,Seq.501    -------------->
Etc.
```

Host B does not use the TSecr = 13 information in segment 7 to update its RTT information because too much time has elapsed since it received segment 5. Host B only updates RTT information if new information arrives that it uses to update its receive window. This is what happens when segment 8 arrives so Host B compares its system clock to the echoed value 63.

2.12.2.2 Delayed ACKs

Host A transmits 3 delayed ACK segments to Host B.

```
[  Host A                                                    Host B  ]
1) TSval=1,TSecr=10,Seq.101     -------------->
2) TSval=2,TSecr=10,Seq.201     -------------->
3) TSval=3,TSecr=10,Seq.301     -------------->
4)                              <-------------- ACK 401,TSval=11,TSecr=1
```

Host B uses the TSval of the first rather than last received segment to echo the timestamp value back to Host A. This reflects the effective RTT of the connection preventing Host A from using the TSval in the last segment (3) to compute an artificially low RTO, potentially causing unnecessary time outs and re-transmissions.

2.12.2.3 Out of Order Packets

Host A transmits 6 segments to Host B, one of which is lost in step (3). This results in out-of-order segment delivery at Host B and causes the fast re-transmit mechanism to be invoked.

```
[  Host A                                                    Host B  ]
1)  TSval=1,TSecr=10,Seq.101    -------------->
2)                              <-------------- ACK 201,TSval=11,TSecr=1
3)  TSval=2,TSecr=11,Seq.201    ---- LOST---->
4)  TSval=3,TSecr=11,Seq.301    -------------->
5)                              <-------------- ACK 201,TSval=12,TSecr=1 DUP #1
6)  TSval=4,TSecr=12,Seq.401    -------------->
7)                              <-------------- ACK 201,TSval=13,TSecr=1 DUP #2
8)  TSval=5,TSecr=13,Seq.501    --------------
9)                              <-------------- ACK 201,TSval=14,TSecr=1 DUP #3
10) TSval=6,TSecr=14,Seq.201    -------------->
11)                             <-------------- ACK 601,TSval=15,TSecr=6
12) TSval=7,TSecr=15,Seq.601    -------------->
13)                             <-------------- ACK 701,TSval=16,TSecr=7
```

Host A re-transmits segment 2 during step (10). Host B continues to set TSecr in the duplicate ACKs with the TSval of the segment in step (1) because this was the segment that last caused Host B to update its receive window. By delaying to update the TSecr value, Host B forces Host A to perform conservative RTT computations effectively extending its RTO.

When Host B receives the re-transmitted segment in step (10) it is able to acknowledge all of the previously missing segments and update its receive window once more. It can now resume using the TSecr information to update its RTT calculation.

2.13 ECN

TCP operates at layer-4 enforcing its own congestion management schemes and effectively treating the network as a black box. With ECN, TCP works together with layer-3 IP allowing the queuing mechanisms in network routers to participate in congestion control.

RFC3168 describes the Explicit Congestion Notification (ECN) protocol and RFC3540 describes use of the single-bit NONCE mechanism to validate ECN responses.

2.13.1 ECN Overview

Figure 2-25 below illustrates the operation of ECN across an IP network. For ECN to work, the end devices and network devices must all understand and support the protocol.

1) Host A and Host B establish a TCP connection using the three-way handshake. Two bits are set in the TCP header of the SYN and SYN ACK packets to signal ECN protocol support.

2) The hosts set the ECN Capable Transport (ECT) bits in the lower two bits of the IP header DSCP byte and transmit TCP segments. The ECT bits tell the routers that the hosts are using ECN. Neither host sets the ECT bits in ACK packets.

3) The hosts continue to exchange TCP segments with the ECT bits set in the IP header of non-ACK packets.

4) A router on the path experiences congestion. Rather than discard the end hosts' packets, Weighted Random Early Discard (WRED) remarks the lower two bits of the IP header DSCP byte to Congestion Experienced (CE).

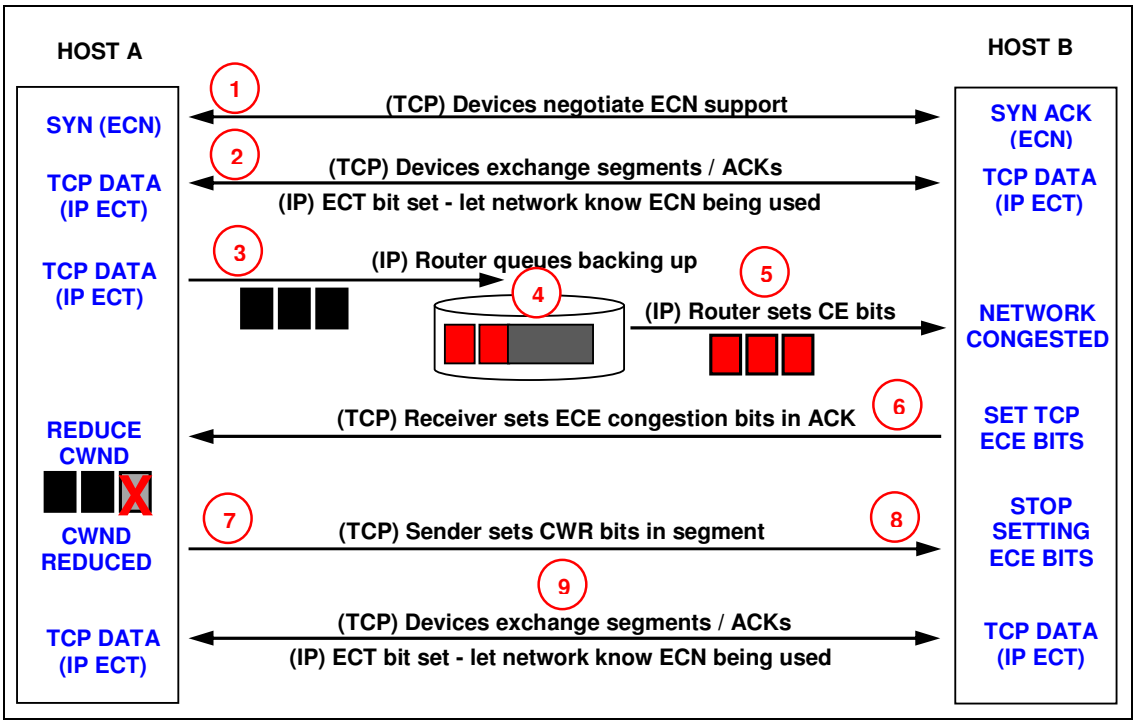


Figure 2-25: ECN Operation

5) Host B receives the CE marked IP packets and sets the ECN Echo (ECE) bit in the TCP header of its ACK segment and sends the segment to Host A.

6) Host A receives the TCP ACK with ECE set and initiates slow start reducing its CWND.

7) Host A sets the Congestion Window Reduced (CWR) bit in the TCP header of the next segment it sends to Host B.

8) Host B receives the segment with the CWR bit and knows that Host A has reduced its window size so it stops setting ECE bits in the segments it sends to Host A.

9) Normal bi-directional TCP communication carries on.

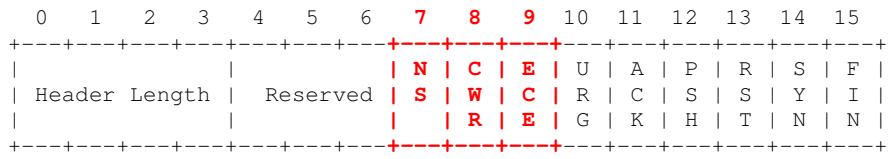
2.13.2 ECN Packets

Three new bits are used in the reserved field of the TCP header for ECN operation, as shown below.

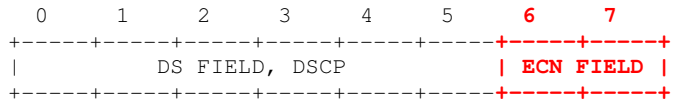
NS - Nonce Sum [bit 7]

CWR - Congestion Window Reduced [bit 8]

ECE - ECN Echo [bit 9]



The IP header contains a 6-bit Differentiated Services Code Point (DSCP) field and a 2-bit Explicit Congestion Notification Field (ECN).



The two bits of the ECN field have the following meaning:

ECN FIELD		
0	0	End device is not ECN capable
0	1	ECT(1) - ECN Capable Transport set by end device (NONCE 1)
1	0	ECT(0) - ECN Capable Transport set by end device (NONCE 0)
1	1	CE - Congestion Experienced set by router to indicate congestion to end device

An end device may set either of the ECT bits in the IP header to indicate that it is ECN capable. These bits are also used as a NONCE (Number used Once) checksum so that the transmitting host knows the receiver is complying with the ECN rules of operation.

2.13.3 ECN Connection Establishment

The initiating host must set the CWR and ECE bits in the TCP header of the SYN packet. The receiving host must respond with only the ECE bit set in the TCP header of the SYN ACK packet. Any other combinations are invalid and ignored.

The initiating host's SYN packet tells the receiver that it is participating as both an ECN sender and receiver. The receiver's SYN ACK packet tells the sender that it is participating as both an ECN receiver and sender.

At any time during the connection, the sending host may or may not set the ECT bits in the segments it transmits. If it sets the ECT bits, the receiving host must respond accordingly. If the sender does not set the ECT bits the receiving host must also not set the ECT bits in its responses. Those packets are treated as non-ECN capable and routed through the network as non-ECN connection packets.

2.13.4 ECN Bit Settings

The ECN bits in the DSCP byte of the IP header are set as follows:

ECN FIELD		
0	0	End device is not ECN capable
0	1	ECT(1) - ECN Capable Transport set by end device (NONCE 1)
1	0	ECT(0) - ECN Capable Transport set by end device (NONCE 0)
1	1	CE - Congestion Experienced set by router to indicate congestion to end device

1) The sending host (which can be either host) sets the ECN bits in the IP header of TCP data segments to either ECT(0) or ECT(1) randomly. Both bit combinations tell the network devices that the sender is ECN capable. These bits are also used to compute the NONCE (described in the next section).
2) When a router in the path to the receiver experiences congestion its Adaptive Queue Management (AQM) mechanism e.g. WRED (described later) sets the ECN bits to CE (1, 1) rather than dropping the packet. Setting the CE bits erases the previous ECT bit settings.
3) If congestion in a router is such that it drops the packet, then normal TCP congestion control mechanisms still apply i.e. the segment is lost so the sending host re-transmits if it doesn't receive and ACK before its RTO expires.

4) Statistically, the congested router(s) will eventually set and forward the IP packet with the CE bits set.
5) Any other routers that receive the IP packet with the CE bits set process the packet by either re-writing the CE bits, or leaving them unchanged before forwarding the packet towards the receiving host.
6) The receiving host receives the IP packet from the network and the CE bits tell it that there is congestion somewhere along the network path. The host computes the cumulative NONCE, sets the corresponding NS and ECE bits in the TCP header of the ACK segment and transmits it towards the sending host. The receiving host does not set the ECT bits in the ACK packet. Please note that the NONCE will now be incorrect at the sending host because the router modified the ECT bits.
7) If the ACK packet is lost, the sending host's RTO timer expires and it re-transmits the data segment with the ECT bits set.
8) When the sending host receives the ACK segment with the ECE bit set, it ignores the incorrect NONCE, adjusts its CWND, sets the CWR bit in the TCP header of the next data segment, re-sets its NONCE variable to that of the receiving host, sets the ECT bits and transmits the segment.
9) If the data segment with the CWR bit set is lost, the sending host re-transmits the segment when its RTO timer expires.
10) The receiving host receives the segment and stops setting the ECE bits in the ACK segments that it transmits.

The ECN congestion control mechanism should apply only to single RTT events (segment and ACK exchanges), including lost and re-transmitted segments, to avoid multiple congestion events causing instability.

2.13.5 ECN Nonce

When the sender initiates an ECN connection it must be sure that the receiver is honouring the ECN bit settings and co-operating with the sender to manage congestion control. If a poorly implemented receiver agrees to participate in an ECN connection but does respond correctly to ECN packets, it could be gaining a performance advantage over other ECN compliant receivers using the same network path. The sender uses a NONCE mechanism to verify that the receiver is operating correctly. If it is not, the sender can take drastic measures, such as reducing its CWND to penalise the receiver.

The NONCE mechanism works as follows:

1) The sender randomly sets either the ECT(0) or ECT(1) bits in the IP header of the TCP segments indicating to the network that it is ECN capable.
2) The receiver computes the NONCE cumulatively for all the segments it is acknowledging and sets the NONCE Sum (NS) bit in the TCP header of the ACK segment.
3) Upon receiving the ACK segment, the sender checks the value of the NS bit to verify whether the receiver is complying with the ECN mechanism.

The use of a single-bit NONCE to fulfil this function is necessary because bit space in the TCP header is restricted. However, a single bit NONCE is still sufficient to detect receiver misbehaviour. A single bit only gives the sender a 50/50 chance of detecting a cheating receiver for each ACK, but the probability of detection increases significantly as consecutive ACK segments are received during the lifetime of the connection.

Figure 2-26 below shows how the NONCE is derived at the receiver and checked at the sender. ECT(1) is NONCE=1 and ECT(0) is NONCE=0. The sender transmits either at random. If the router sets the CE bits the NONCE is deemed to be cleared i.e. NONCE=0.

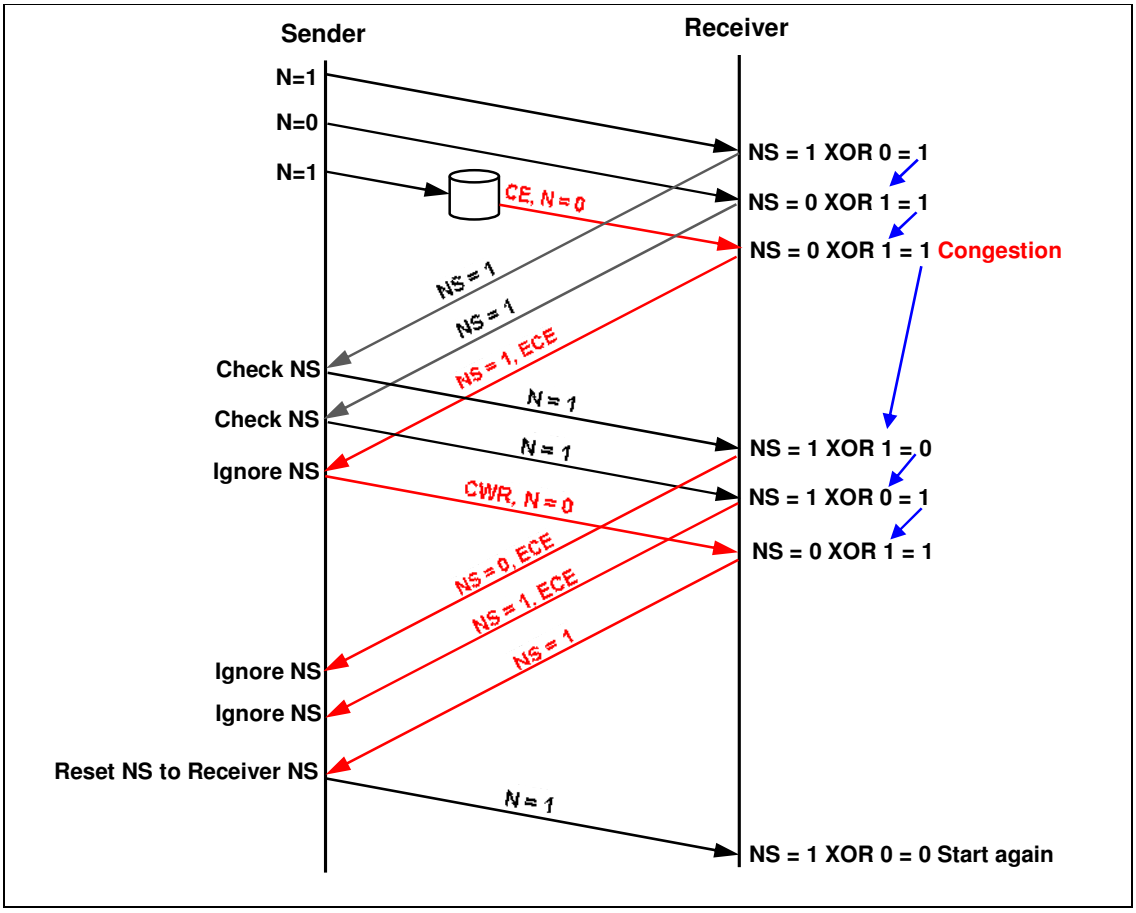


Figure 2-26: NONCE Calculation

2.13.6 Adaptive Queue Management (WRED)

Random Early Discard (RED) and Weighted Random Early Discard (WRED) are mechanisms that routers use to deliberately drop packets from the front of a queue that is becoming congested. This differs from standard queuing behaviour, which drops packets from the tail of the queue when it is full. With tail dropping, the network administrator has no control over which packets to drop because they never enter the queue in the first place, they just fall off the end.

RED and WRED are often used to deliberately drop TCP packets to force TCP senders to enter slow start, thereby reducing congestion on high utilisation links.

WRED must be used for ECN because the “weighted” part means that it can distinguish IP packets with different DSCP and ECN values in the IP header, whereas RED cannot.

WRED is configured to drop different quantities of packets with specific DSCP/ECN values based on different queue fill levels. This is referred to as the packets’ drop probability.

And here’s the most important part. Having decided that a packet is eligible to be dropped, WRED does NOT have to drop it. Instead, a policy can be configured to re-write the DSCP/ECN bits in the IP header and forward the packet. This is EXACTLY what is needed for ECN to set the CE bits.

Figure 2-27 below shows a representation of how WRED would be used for ECN.

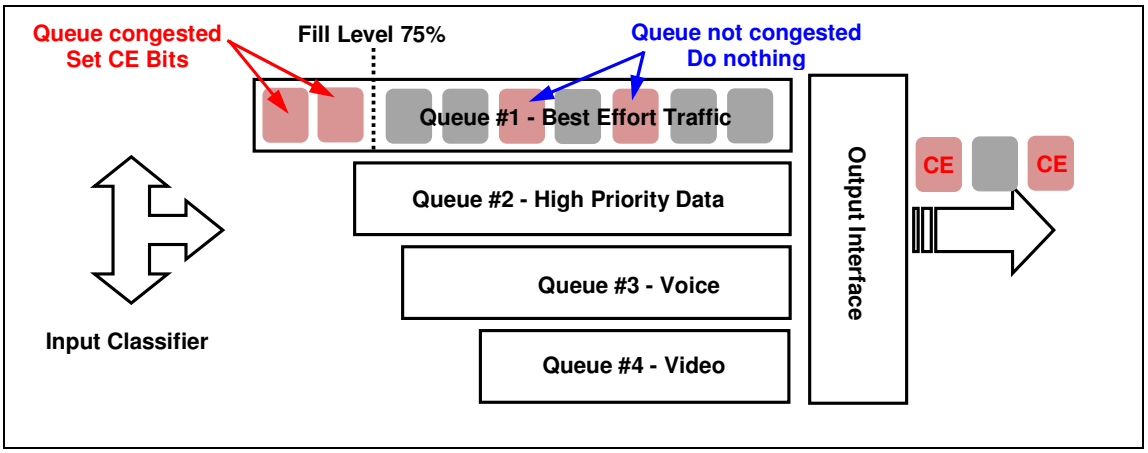


Figure 2-27: ECN WRED Queuing

Figure 2-28 below shows a possible, simple re-write probability configuration for ECN IP packets. In other words, when the queue is 75% full, re-write the CE bits to "1 1" in the ECN field of every single IP packet (100% of the packets) that is transporting TCP frames.

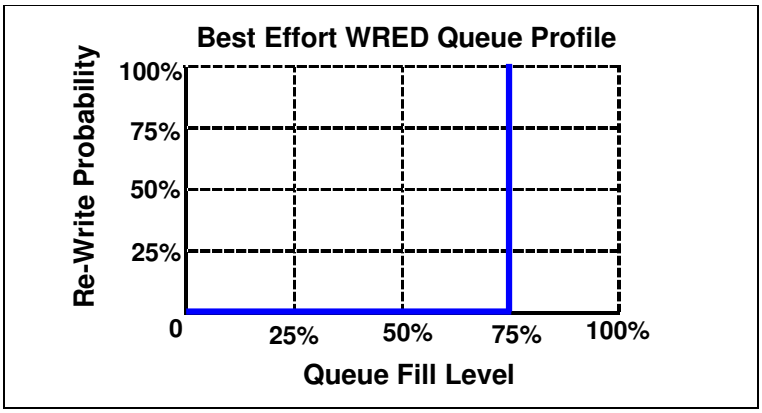


Figure 2-28: ECN WRED Re-Write Probability