Transport Layer Protocols
(*Reliable Transport, Flow & Congestion Control*)

References: http://www.tcpipguide.com/free/t_TransportLayerProtocols.htm et al.

This document is covers three topics: **reliable transport**, **flow control** and **congestion control** techniques. The transport layer protocol, TCP (Transmission Control Protocol), provides these services. Note that, UDP (User Datagram Protocol), which is another transport layer protocol of Internet protocol suite, does not have these capabilities.

1 Reliable Transport

Reliable transport includes the ability to *recover from packet error or loss*. To accomplish this TCP uses a type of ARQ (*automatic repeat request*) scheme which *retransmits a packet in the event of a bit-error or packet loss*. The principle of an ARQ scheme is presented below with a specific focus on its implementation in TCP. This section will cover the principle of reliable transport in general as well as its specific implementation in TCP.

1.1 Principle of ARQ (Automatic Repeat Request)

A retransmission scheme corrects a transmission error by retransmitting the lost or corrupted data packets. To implement this capability, each packet must carry a sequence number.

Stop-and-Wait Retransmission Scheme

The simplest retransmission scheme uses the following technique:

1. Send
2. Get ACK / NAK *(ACK = acknowledgement, NAK = negative acknowledgement)*
3. Send (retransmit same packet if NAK, send next one if ACK).

Such a scheme, though simple, suffers from *high latency and low throughput* (see the example below).

Example: Consider a 1-Gbps geostationary satellite IP data link (with Stop-and-Wait scheme). What would be the throughput (effective bit-rate) with the above basic retransmission scheme?

- 1 Gbps link → One bit time, T = 1 ns
- IP packet has a max of (2^{16} – 1) bytes/packet
- One IP packet time (maximum) = (2^{16} -1) bytes/packet x 8 bits/byte x 1 ns/bit ≈ 0.5 ms / packet
- Minimum roundtrip distance, d ≈ 71600 km
- Minimum roundtrip delay ≈ 71600 km / (3 x 10^5km/s) ≈ 240 ms
- The roundtrip delay per packet = 240 ms (ignore processing time)
- Need to receive ACK/NAK every 240 ms so...
    - Max packet rate = 1 packet/(240 ms) ≈ 4 packets/s
    - Average bitrate = 4 packets/s x 2^{16} bytes/packet x 8 bits/byte = 2.1 x Mbps (476 times slower than its capacity bit rate, of 1 Gbps)

Most of the time this rate is not acceptable.

Go-Back-to-N vs. Selective-Repeat Retransmission Schemes

The solution to the problem with stop and wait is to use a system that allows up to a certain number of packets (N) sent without receiving the ACK. If there is any error in some of those packets, the sender resends them as soon as the sender understands the error. One question though: when there is an error or packet loss, should the sender resend that particular packet only, or all the packets that are already sent after that packet?

- If all of them, the scheme is called **Go-Back-to-N** scheme.
- If it is just that one packet, the scheme is called **Selective Repeat** (*TCP uses this one, we will discuss the detail later in the section on 'Selective Acknowledgment' in Section 3 of these notes*). This requires less repetition (and hence has higher throughput) but causes out of sequence delivery. For this to be implemented, therefore, the *receiver requires re-sequencing capability*, which is already provided by TCP.

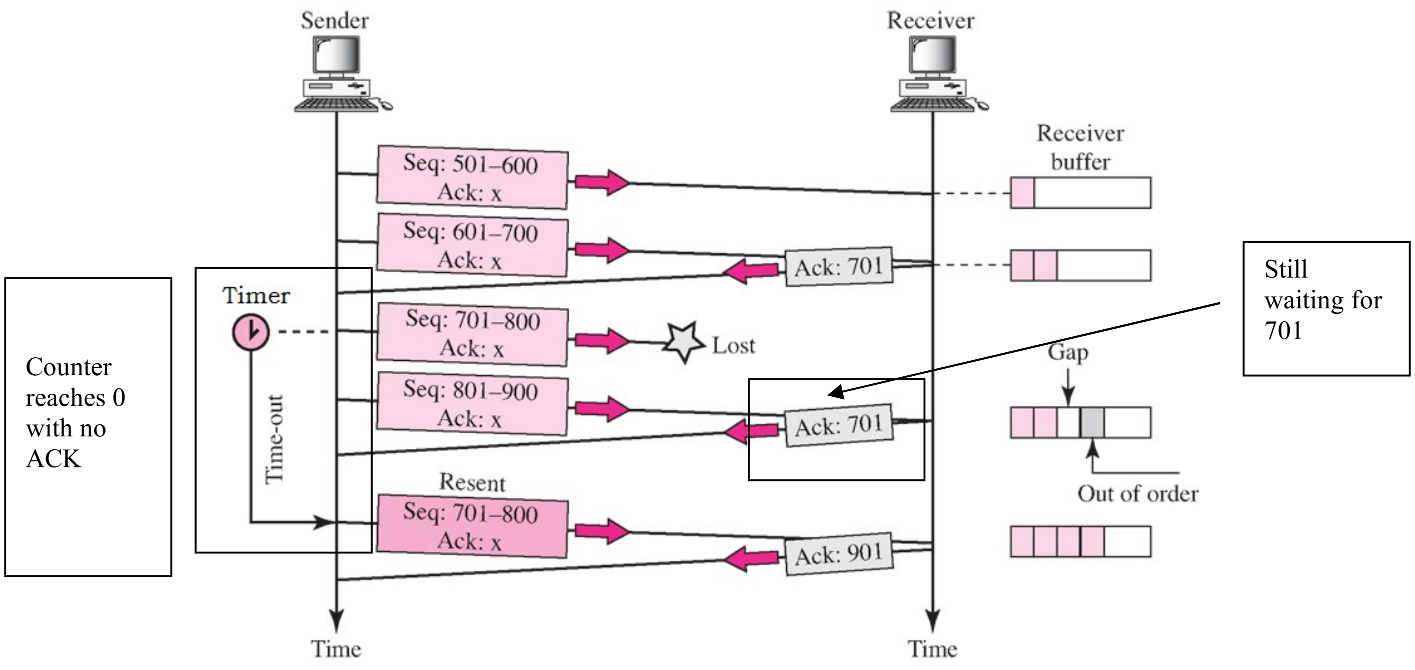
Error and Packet Loss – How does the transmitter know?

There are *three components a transmitter may rely on*:

1. **ACK** (acknowledgement) message from the receiver to sender when there is no bit error. Upon reception of the ACK the transmitter sends the next data packet.
2. **NAK** (Negative ACK) message from the receiver to sender when there is an error. Up on reception of the NAK the data sender re-sends the data packet.
3. **Time-out** of sender's timer for lost packet. The sender sends a packet and sets a count-down timer that starts decrement as soon as the packet is transmitted. *If the counter decrements to zero before any ACK/NAK arrival, the sender retransmits*.

TCP uses ACK and timer but no explicit NAK. TCP's ACK is the sequence number of the data byte it is expecting to receive next from the transmitter. The meaning of an ACK is '*I received all other bytes before this one*'. If the transmitter keeps transmitting new segments but receives *duplicate ACK numbers repeatedly*, the transmitter assumes 'segment-loss' and retransmits the segment. This retransmission is called **fast retransmission.** There is also another type of re-transmission that is triggered by the time-out of a counter.

Example:



Ref: Data Communications and Networking, By Forouzan, McGraw-Hill

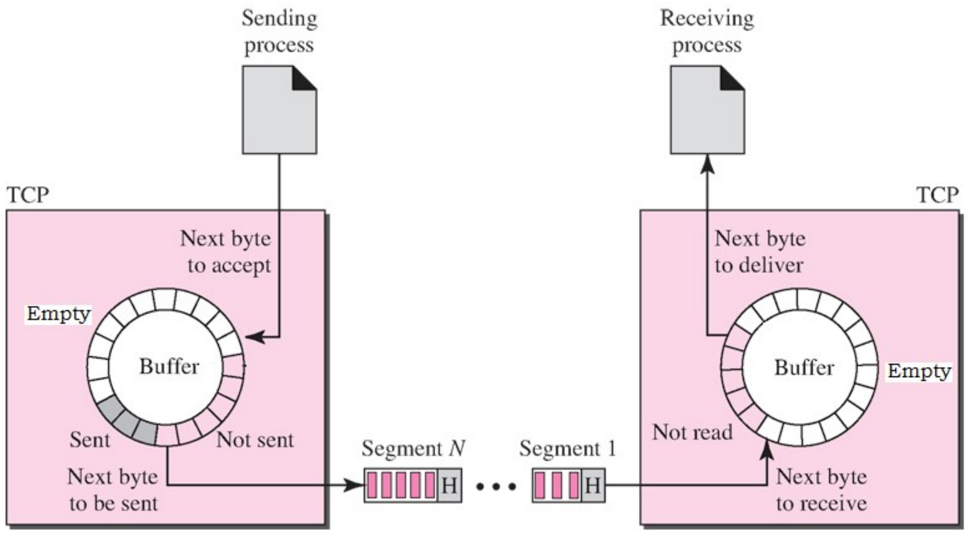
The selective-repeat scheme of TCP protocol requires the following set of techniques:

1. Sequence numbering (SEQ)
2. Error detection scheme (checksum)
3. ACK scheme
4. A sender-TCP count-down timer
5. Sender-TCP buffer
6. A receiver-TCP buffer
7. Sender-TCP sliding window
8. Receiver-TCP sliding window

1.2 *Sliding Window Protocol in TCP*

Ref: http://www.tcpipguide.com/free/t_TCPSlidingWindowDataTransferandAcknowledgementMech-5.htm

The following figure illustrates the sender-TCP and receiver-TCP buffers per process-to-process (port-to-port) link. Each location in the circular buffer is one byte. For the sake of simplicity of the illustration the buffer length is shown to be 20 bytes only. In practice the buffer size can be tens of thousands of bytes.



Ref: Data Communications and Networking, By Forouzan, McGraw-Hill

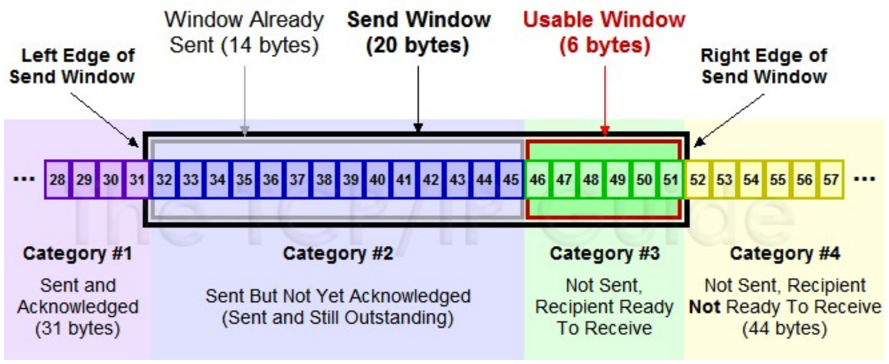
Sender Side	**Receiver Side**
Circular buffer for <u>outgoing</u> data (for outgoing TCP segments)	Circular buffer for <u>incoming</u> data (for incoming TCP segments)
Pointer P_{TX1} to locate the buffer space for next byte from the upper layer	Pointer P_{RX1} to locate the next byte to be transferred to the upper layer
Pointer P_{TX2} to locate the next byte to be packed in next outgoing TCP segment	Pointer P_{RX2} to locate the buffer space for next received byte
'Not Sent' section of the buffer has bytes that are waiting to be sent	'Not Read' section of the buffer has bytes that are waiting to be delivered to the upper layer
'Sent' section of the buffer means that these bytes are sent but their ACK is not received yet (these are kept for retransmission, if needed)	
'Empty' section means empty locations, which can be used to store new bytes from the upper layer	'Empty' section means empty locations, which can be used to store new received bytes

Note: *If inflow rate is higher than outflow rate the buffer will overflow*. Obviously, the **buffers need flow control**. When the buffer is nearly full and crossing a pre-defined threshold, the scheme must signal the 'sender' to slow down or stop for a while.

<u>The Sliding Window Protocol</u>

The flow control protocol is called the **Sliding Window Protocol** since the sections of the <u>windows</u> (empty, sent and to read/send) *slides over the buffers locations* as transmission and reception continue.

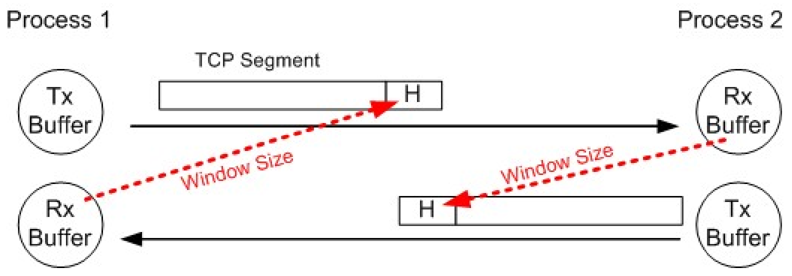
- **Category 1**: bytes sent and acknowledged. The space can now be used to receive more data from the upper layer.
- **Category 2**: bytes are sent but not yet acknowledged. The space remains occupied by the sent bytes until valid ACK is received and they are moved to Category #1.
- **Category 3**: bytes from the upper layer waiting to be sent. The receiver has buffer space to receive this many new bytes, and sender can send anytime.
- **Category 4**: bytes from the upper layer that are waiting to be sent. The receiver, however, has no buffer space to receive these bytes (not ready to receive), and *sender must not send these bytes until the receiver allows to do so by updating its window size*.


Left Edge of Send Window
Window Already Sent (14 bytes)
Send Window (20 bytes)
Usable Window (6 bytes)
Right Edge of Send Window
... 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 ...
Category #1
Sent and Acknowledged (31 bytes)
Category #2
Sent But Not Yet Acknowledged (Sent and Still Outstanding)
Category #3
Not Sent, Recipient Ready To Receive
Category #4
Not Sent, Recipient Not Ready To Receive (44 bytes)

<u>http://www.tcpipguide.com/free/t_TCPSlidingWindowAcknowledgmentSystemForDataTranspo-6.htm</u>
(The sender window is shown as a linear buffer for simplicity, but it is actually a circular buffer).

2 Flow Control

TCP has an end-to-end flow control mechanism. The **receiver**-TCP layer dynamically sets the maximum number of bytes that the **sender**-TCP layer's can send without receiving an ACK (using the *Window size* element in the TCP header). See the following illustration.



A process (say Process1 #1) checks its receiving buffer Rx Buffer in Process 1, to estimate how many more bytes (say N bytes) it can accept from the other process (Process-2) without any risk of overflow. *In the next outgoing TCP segment* the *process (Process-1) sets the Window-Size field in TCP header accordingly* (Window-Size = N) in order to let the other side (Process-2) know how many bytes Process-1 can receive from Process-2. The other side (Process-2) then estimates how many bytes are already sent in the 'pipeline' (say, M bytes). It can then send up to (N – M) bytes. The following figure illustrates this mechanism in detail.

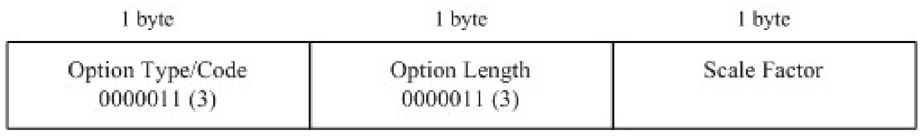
<u>How to Change the SEND Window?</u>

Overflow and/or congestion can happen dynamically, that is why the window size is dynamic. There are two ways to change the window size:

1. The TCP format provides a field called "*Window Size*" to dynamically inform the sender the size of receiver window (in number of bytes)

2. Maximum "base size" of the widow is (2^{16}-1 ~= 64 kb), which is approximately equal to one maximum IP datagram size. However, if the delay and/or bit-rate (or bandwidth) is large (in other words the delay-bandwidth product is large) then the window size needs to be bigger than just 64 kilobytes. The *Window Size* field cannot make the window larger than this, but the *<u>Window Scaling Option</u>* in the header can be used to do this.

   The **Window Scaling Option (Option Type/Code = 03)** is another way to define the window size.



1 byte	1 byte	1 byte
Option Type/Code 0000011 (3)	Option Length 0000011 (3)	Scale Factor

   <u>Example:</u>

- If Scale factor = 0000 0100 which is 4
- Window size = Base_Size (binary) x 2^4
- Left-shift Base_Size bits (binary) by 4 (i.e. '<<4') $\leftarrow$ (multiplication factor of 16)

3 Congestion Control

3.1 The Small Packet Problem: Nagle's Algorithm

Problem

When TCP sends an acknowledgment (ACK) for every received segment and the transmitted segments are small (e.g., in **Telnet applications** that send one TCP segment per keystroke), a significant inefficiency arises. This inefficiency is due to the **overhead** of *TCP and IP headers, which may be larger than the payload itself*.

This leads to:
1. Wasted bandwidth: A large proportion of transmitted data consists of headers rather than useful payload.
2. Increased network congestion: A high number of small packets unnecessarily consume network resources.

Nagle's Algorithm

(http://en.wikipedia.org/wiki/Nagle%27s_algorithm).

A strategy on the **sender side** to address the **Small Packet Problem** by *combining multiple small payloads into larger TCP segments*.

How it Works
1. **Send the first byte immediately**: When data is available for transmission, the first byte is sent immediately, even if it forms a small packet.
2. **Accumulate additional data while waiting for an ACK**: The algorithm holds off on sending any further data until the acknowledgment (ACK) for the previous packet is received. During this wait, the sender accumulates new outgoing data.
3. **Send accumulated data**: Once an ACK is received (or enough data has been accumulated to fill a full TCP segment), the buffered data is sent as a single packet.
4. **Repeat the process**: The sender waits for the next ACK while continuing to accumulate additional data for subsequent transmissions.

Send first byte → Wait for ACK while *accumulating new bytes* → Send on ACK or large enough segment → repeat

Advantages of Nagle's Algorithm
- **Reduced Overhead**: By combining multiple small packets, the proportion of header-to-payload size decreases, making transmissions more efficient.
- **Better Bandwidth Utilization**: The number of packets transmitted is reduced, decreasing congestion in the network.

Downsides of Nagle's Algorithm
- **Increased Latency**:
  - Data may be delayed while waiting for ACKs or while accumulating enough data to form a full TCP segment.
  - This can be especially noticeable in real-time or interactive applications like Telnet, where users expect immediate responses to their inputs.

Nagle's Algorithm strikes a balance between efficient **bandwidth** utilization and transmission **latency**. While it is effective in reducing the overhead caused by small packets, it may introduce latency in scenarios requiring real-time responsiveness. For such cases, disabling Nagle's Algorithm might be appropriate.

Is the TCP PUSH flag in the TCP Header used to disable Nagle's Algorithm?

- No, the **TCP PSH (Push)** flag is not directly used to disable Nagle's Algorithm. However, it plays a complementary role in signaling the need for immediate delivery of data.

- The **PSH flag** is set by the sender to indicate that the **receiving** TCP stack should deliver the data to the application immediately, rather than buffering it. This ensures that the data is processed promptly upon receipt, without unnecessary delays.

So how is Nagle's algorithm disabled?

- The **TCP_NODELAY** flag can be used when setting up the transmitters **socket** to disable Nagle's Algorithm.

```
socket (  socket_fd,    IPPROTO_TCP,   TCP_NODELAY);    // pseudocode
```

- When this option is enabled, TCP will send packets immediately without waiting to accumulate more data, thereby bypassing Nagle's Algorithm. This is particularly useful for applications requiring low-latency communication, such as real-time systems, online gaming, or interactive applications like Telnet.

What is the key difference between TCP_NODELAY and the TCP PSH Flag?

TCP_NODELAY:
- Disables Nagle's Algorithm, ensuring that small packets are **sent immediately by the sender**, without waiting to accumulate more data or for an acknowledgment.
- It affects the **sender's behavior** at the TCP layer.

PSH Flag:
- This is a per-packet signal that tells the **receiver** to pass the data to the application as soon as it is received, without waiting for additional data.
- It affects the **receiver's behavior** at the TCP layer.

Key Point?

While the **PSH flag** and **TCP_NODELAY** both aim to reduce delays, they address different aspects of TCP behavior. Disabling Nagle's Algorithm with **TCP_NODELAY** ensures that data is sent immediately, whereas the **PSH flag** ensures that data is delivered promptly to the receiving application.

3.2 Silly Window Syndrome: Clark's Algorithm

(http://en.wikipedia.org/wiki/Silly_window_syndrome)

Silly Window Syndrome occurs when a slow receiver (e.g., due to limited processing power or high system load) processes only a small amount of incoming data and immediately sends an acknowledgment (ACK) with a very small **window size**.

This leads to:

1. The sender transmitting very small segments of data
2. High overhead - as each small segment has a relatively large header (at least 40 bytes for TCP and IP) compared to its data payload.
3. Poor network efficiency and reduced throughput, as many small packets increase congestion and processing load.

This behavior is undesirable and is termed **Silly Window Syndrome (SWS)** because of its inefficiency.

Clark's Algorithm
A strategy implemented on the **receiver side** to mitigate SWS. It aims to *avoid advertising a small window size* and promotes efficient data transfer by managing how the receiver processes and advertises its available buffer space.

How Clark's Algorithm Works

- When the receiver's advertised window is small, the receiver delays sending an ACK.
- Receiver waits (continues freeing buffer space) until:
    - At least half of the receive buffer is free, or
    - A sufficiently large window size (e.g., based on a predefined threshold) can be advertised.

Risk

- Buffer Overflow: While delaying ACKs, there is a potential risk of buffer overflow if the incoming data rate exceeds the receiver's processing capacity.

Summary

1. *Receiver waits for a minimum window size (e.g. half the receive buffer) before sending ACK*
2. *Process data* (free up buffer space) while waiting

Risk: Receive buffer could overflow if incoming data rate is faster than processing rate

3.3 Congestion Collapse

When there is congestion in the path, packets are dropped and the sender's Retransmission Timer expires, so ...

sender retransmits → increased congestion → more dropped packets and time-outs
→ sender retransmits → increased congestion → more dropped packets and timeouts ... → Congestion Collapse

The TCP Tahoe (a.k.a. Slow Start) Algorithm

The **Slow Start Algorithm** addresses this by ensuring that the sender does not overwhelm the network with data.

Key Concept:
A new variable, the **Congestion Window (***CongWin***)**, is introduced to dynamically reduce the size of the send window (SW) and limits thet packet population (congestion) in the network.

The send window is determined as the minimum of:

1. The sender's buffer size (SendBuf),
2. The receiver's advertised window (RcvWin), and
3. The congestion window (*CongWin*).

Send Window is usually limited by the *CongWin*:

SW = min (Sender's Buffer Size, RcvWin, *CongWin*)
= *CongWin* (where *CongWin* << RcvWin << SendBuf)

Congestion Window Behavior

The congestion window is an integer multiple (w) of the maximum segment size (MSS)

$$CongWin = w \times MSS$$

* MSS = 536 by default but can be defined in the TCP option field.

1. Congestion Window Increases

- **Slow Start initialized**:
    - Initially w = 1
    - This allows the sender to start transmitting 1 segment.

- **Slow Start Phase (Exponential Growth)**:
    - *CongWin* increases **exponentially** (doubles) every round-trip time (RTT) ***iff*** no packet loss occurs.
    - So after each successful RTT:

      $w = w \times 2$

    - This phase continues until the **Threshold** is reached.

- **Congestion Avoidance Phase (Linear Growth)**:
    - Once *w* reaches the **Threshold**, growth slows so that

      $w = w + 1$
    - This phase continues until *CongWin* reaches the receiver's advertised window, the sender's buffer size or if *packet loss* occurs (enters Loss phase below).

2. Loss Phase

- **On Packet Loss**:
    - Packet loss is detected by **retransmission timeout** or receiving **duplicate ACKs**
    - On packet loss TCP Tahoe:
        1. Resets w so that $w = 1$ and re-enters the **Slow Start** phase
        2. **Reduces the Threshold** to half its value at time of loss

$$\text{Threshold} = w / 2$$

Summary of Key Phases

1.	Slow Start:	w grows exponentially up to the Threshold.
2.	Congestion Avoidance:	w grows linearly beyond the Threshold.
3.	Loss:	w is reset to 1 (re-enters slow start phase) and threshold to w / 2

This adaptive behavior allows TCP Tahoe to balance efficient data transmission with network stability, avoiding excessive congestion.

Selective Acknowledgment (SACK) in TCP (RFC 2018)

Problem

In traditional TCP with **cumulative acknowledgment**, the sender is informed only about the highest byte successfully received in order. If multiple packets are lost within a single window, this approach significantly reduces TCP throughput because:

1. The sender can detect only one lost packet per round-trip time (RTT).
2. The sender might retransmit packets unnecessarily, including those that have already been received successfully, leading to inefficiency.

Solution

Selective acknowledgment (SACK) allows TCP to handle multiple lost packets more efficiently by allowing the receiver to inform the sender explicitly about *which packets* (or ranges) have been successfully received. The sender can then retransmit only the missing packets.

Benefits

1. Only missing packets are retransmitted, conserving bandwidth and reducing network congestion.
2. Multiple lost packets can be recovered within one RTT, reducing delays and improving throughput.

Example

Suppose a sender transmits data in the byte range 0–10,000, and packets corresponding to byte ranges 2,000–3,000 and 6,000–7,000 are lost.

- **Without SACK**:
    - The sender retransmits all packets starting from byte 2,000, even though packets 3,001–5,999 were successfully received.

- **With SACK**:
    - The receiver sends a SACK option indicating that bytes 0–1,999, 3,001–5,999, and 7,001–10,000 were received.
    - The sender retransmits **only** the missing ranges: 2,000–3,000 and 6,000–7,000.

3.4 Explicit Congestion Notification (ECN)

http://en.wikipedia.org/wiki/Explicit_Congestion_Notification

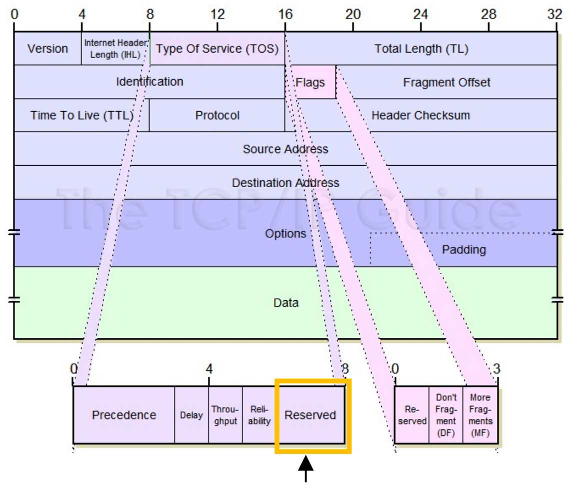
Explicit Congestion Notification (ECN) is another scheme (RFC3168), which is also a TCP/IP protocol enhancement. This scheme provides a *mechanism for reporting intermediate congestion in the route to the destination*. This is a broader class of congestion control that uses active queue management (AQM), which has the capability to detect early signs of congestion.

ECN is an *optional add-on capability*. To use it the TCP-points (both source and destination) as well as intermediate routers need to be ECN-capable.

Conventionally, when there is congestion in the network it is signalled by dropped packets. An ECN enabled **intermediate router**, however, can instead signal congestion in the network by setting a mark in the IP header instead of dropping the packet. The **receiver** that gets this signal then echoes the congestion indicator back to the **sender** to tell it to reduce its transmission rate.

Since routers operate at the 'Internet Layer' in the TCP/IP stack (the layer just below the transport layer), they must signal congestion via the IP header. (Note that the transmission rate, however, is handled by the sender/receiver endpoints at the transport layer.)

ECN routers signal congestion to the sender/receiver using the **two least significant bits (7th and 8th bit from the left) of type-of-service field** of the IP datagram header (Internet layer).



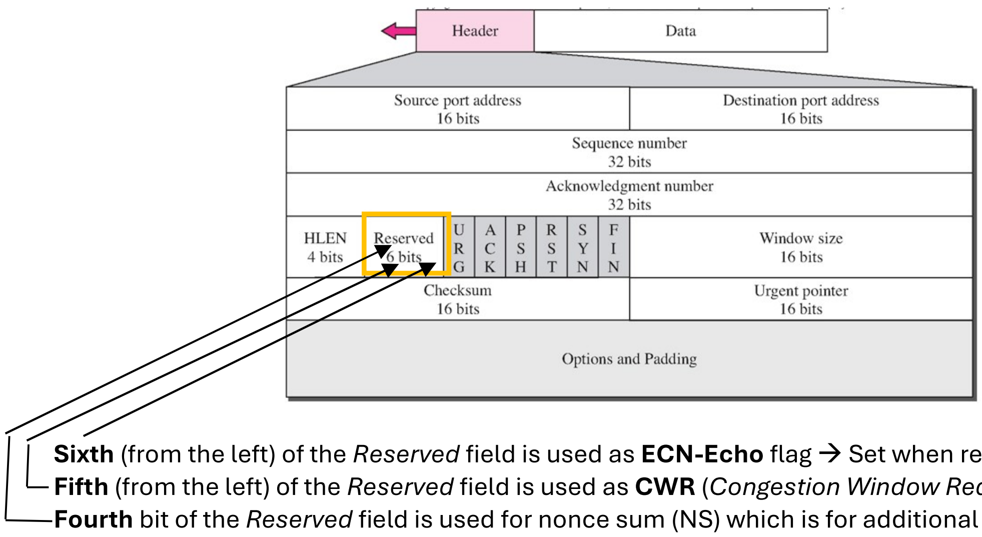
These two bits

- 00 – Non ECN-Capable Transport, Non-ECT
- 10 – ECN Capable Transport, ECT(0)
- 01 – ECN Capable Transport, ECT(1)
- 11 – Congestion Encountered, CE $\rightarrow$ This must be handled by transport layer protocols (TCP)

ECT(0) and ECT(1) both mean ECN capable and either one can be used by TCP of a node. During TCP connection the end-points must negotiate which one they will use.

When there is any congestion, a router sets these bits to 11 to signal that congestion has been encountered. The receivers IP forwards this CE (congestion encountered) status to its TCP layer and the receivers TCP then reports it to the senders TCP.

For the receivers TCP to send a congestion report to the senders TCP, it uses **three reserve bits of the TCP packet header**:



Sixth (from the left) of the *Reserved* field is used as **ECN-Echo** flag → Set when receiver sees congestion
Fifth (from the left) of the *Reserved* field is used as **CWR** (*Congestion Window Reduced)* flag
Fourth bit of the *Reserved* field is used for nonce sum (NS) which is for additional security against unwanted erasure of CE (congestion encountered) bit.

When the receivers TCP gets CE (congestion encountered) from the IP-layer

- It sends <u>ECN-Echo = 1</u> in the next TCP-segment to the sender.
- The sender reduces its congestion window and sets the CWR = 1 in its next TCP segment to let the other side know that the congestion window is reduced.