* Window scale: allows shift of window size field up to 16 bits left (i.e., \( W' = 2^{w + \text{scale}} \)) to permit more packets to be in flight for high BW-delay product links.

Note: BW-delay product (byte rate \( \times \) one-way delay) = "# of pipelined bytes in flight."

* Set Selective Repeat instead of Go-Back-\( n \) ARQ.

- **TCP Flow Control**
  - TCP flow controls aim to
    1. Avoid receive buffer overflows.
    2. Reduce transport overheads.

  - The buffer overflow problem is obvious—if XMTRs send data faster than RCVRs can process it, buffers overflow.

  - To avoid overflows, in each outgoing Window Size field, TCP entities state their available buffer space (i.e., the # of bytes they will accept beyond the current seq #).

  - By dynamically varying these declarations, TCP entities can limit the number of segs sent per \( RTT' \) and thereby indirectly control their peer's transmission rate, i.e.,

    \[
    \text{rate} \approx \frac{\# \text{ segs sent}}{RTT'} \approx \frac{\text{window size}}{RTT}.
    \]

  - **In practice**
    - Many TCP implementations delay RCVR ACKs and window updates up to 500 msecs in hope of hitching a "free" ride on a reverse data seg.

    - Many also implement "Nagle's algorithm," i.e., for character data they
      - Send the first byte immediately, but buffer subsequent bytes until the ACK returns.
      - Send the buffered bytes, but buffer subsequent bytes until the next ACK returns.
      - and so on ...

    - Obviously, delaying data segs in this fashion is not a great idea when the segs contain timely info, e.g., a cursor movement.

    - The "push" flag can be used override all of these policies and force entities to empty their buffers.

    - Other problems that sap transport efficiency include the "silly window" syndrome.

  - The transport overhead problem is best illustrated by example. Suppose a telnet user is sending isolated characters to a remote editor. Then ...  

  

<table>
<thead>
<tr>
<th>USER ACTION</th>
<th>PACKET GENERATED</th>
<th>EDITOR ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>type &quot;a&quot;</td>
<td>41-byte (20+20+1) segment</td>
<td>begin read TCP ACK</td>
</tr>
<tr>
<td>see &quot;a&quot;</td>
<td>41-byte (20+20+1) segment</td>
<td>TCP window update</td>
</tr>
</tbody>
</table>

  - Thus the transport efficiency is only 1/62! 

  - For blocks of 10 chars. the efficiency \( \sim \) to 1/18.

  - For blocks of 100 chars. the efficiency \( \sim \) to 5/18.

* The silly window syndrome can arise when users only read isolated characters from remote senders, e.g.,

  - Suppose a large receiver buffer is full.

  - When the next byte is read, a one byte buffer ACK is generated.

  - Regardless of the sender window size, when a one byte buffer ACK is received one byte is sent.

  and so on ...

* Regardless of the window sizes only one byte messages are generated, and the efficiency is poor.

* One solution is to force the receiver to wait until a sizable buffer space has been freed before advertising that space is available.
TCP Congestion Control

- TCP congestion controls aim to
  1. Avoid congestion (network buffer overflows).
  2. Allocate congested resources fairly.
- Congestion occurs whenever, due to poor provisioning, or excessive demand, the traffic traversing some network resource exceeds its capacity.
- "Fair" allocation of a congested resource means roughly, dividing it equally among competing connections.
- On an end-to-end basis this means TCP should ideally transmit at rate
\[
R = \min \left\{ \frac{C_1}{J_1}, \ldots, \frac{C_N}{J_N} \right\}.
\]
where
\[
C_i = \text{capacity of the } i\text{th link (bps)} \quad J_i = \# \text{ of competing } i\text{th link connections}
\]
- Under TCP, rate allocation is accomplished in a distributed fashion by, as necessary, imposing limits on connection transmission rates above and beyond those imposed by the flow controls, i.e., by further limiting the number of segments sent per $RTT$.

The sender may only send frames with sequence numbers in the range of BOTH windows,

\[\text{last ACKed seq } \#+1, \text{ last ACKed seq } \#+ \min \{ \text{CW, RxW} \}\]
i.e., frames may be sent no faster than the slowest component, the receiver or the network.

- The sender assumes that congestion is present whenever a sent TCP segment times out.
- Given this assumption it updates the CW size as follows:
  initialize: CW=MSS, Thres=64K bytes

loop: if (ACK received & CW<\min\{Thres,RxW\})
  new CW = old CW x 2
  [slow start]
else if (ACK received & Thres<CW<RxW)
  new CW = old CW + MSS
  [congestion avoidance]
else if (segment timeout)
  Thres=CW/2
  CW=MSS

In words, as long as no timeouts occur
- CW grows exponentially up to \(\min\{\text{Thres}, \text{RxW}\}\)
- CW grows linearly for \(\text{Thres}<\text{CW}<\text{RxW}\)
When a timeout occurs
- The threshold is halved
- CW is reset to MSS (maximum segment size)

More specifically, if a connection sends \(w\) segs of maximum segment size (MSS) per $RTT$ its rate is
\[
\text{rate} = \frac{w \ast \text{MSS}}{RTT}.
\]
TCP starts with a small \(w\) value and probes for spare BW by \(\sqrt{w}\) until a seg loss occurs (detected by a timeout) at which time it \(\sqrt{w}\) to a "safe" level, and begins probing again by increasing \(w\) more slowly.

- The idea is that as all competing connections probe and back off simultaneously the resource allocation approaches that of the fair allocation.

TCP simultaneously implements its flow and congestion control schemes using two windows:

* A FLOW CONTROL WINDOW set by the Rx
  \[
  \text{[last ACKed seq } \#+1, \text{ last ACKed seq } \#+\text{RxW}]\]
  where RxW denotes the available Rx buffer capacity.
* A separate CONGESTION WINDOW set by the sender based on a sender estimate of congestion
  \[
  \text{[last ACKed seq } \#+1, \text{ last ACKed seq } \#+\text{CW}]\]
  where CW denotes the sender's current estimate of the available network capacity.

For MSS=1024 bytes, after a first timeout at 0

\begin{center}
\begin{tikzpicture}
  \draw[->] (0,0) -- (5,0) node[below] {Transmission number};
  \draw[->] (0,0) -- (0,6) node[left] {Congestion window size};
  \draw (0,0) -- (5,5.5) node[above] {Threshold};
  \draw (0,0) -- (5,0.5) node[above] {Timeout};
  \draw (0,0) -- (5,4) node[above] {Threshold};
\end{tikzpicture}
\end{center}

- Modifications and Extensions
  - The preceding scheme (the one in the text) is often referred to as TCP Tahoe.
  - Under Tahoe, lost segments block further transmissions until a timer expires.
  - As these delays slow adaptation, most OSs implement a variant of Tahoe known as TCP Reno.
  - TCP Reno differs from TCP Tahoe in two respects:
    1. It implements a FAST RETRANSMISSION mechanism that retransmits segments as soon as three copies of the same ACK are received.
2. It implements a FAST RECOVERY mechanism that essentially cancels the slow start phase after a fast retransmission.

A third variant of Tahoe that is not widely implemented is TCP Vegas. Whereas, Tahoe and Reno react to congestion, Vegas attempts to avoid it. The basic idea is:

1. To use RTT measurements to anticipate imminent segment loss, e.g., the longer the RTT's the greater the presumed congestion, and
2. To lower the segment transmission rate linearly when imminent segment losses are anticipated.

- To get a feel for the steady-state rates actually achievable under TCP Tahoe (or Reno) we

  - First assume that
    * The TCP RCV buffer is very very large.
      $\Rightarrow$ that the receive window can be ignored
    * The TCP sender has much data to send.
      $\Rightarrow$ the connection is never idle for lack of data

- Next we note that if we ignore TCP's slow start phase (reasonable because its exponential growth phase is short relative to the linear congestion avoidance phase) TCP's congestion window . . .

  grows linearly to some critical size $W \cdot MSS$, is reduced to $W \cdot MSS/2$ when congestion occurs,

  $W \cdot MSS$ to $W \cdot MSS / 2$

- Over each sawtooth congestion avoidance cycle of length $W/2$ RTT's the connection's rate varies linearly from $W \cdot MSS$ to $W \cdot MSS / 2$.

- If follows that the average rate is

  $$R_{ave} = \frac{1}{2} \left( \frac{W \cdot MSS}{2 \cdot RTT} + \frac{W \cdot MSS}{RTT} \right)$$

  $= 0.75 \cdot \frac{W \cdot MSS}{RTT}$

  i.e., the average rate is 3/4ths of the peak rate and as the MSS $\searrow$, the RTT $\nearrow$, and/or congestion $\nearrow$.

- Is this rate allocation fair? In practice, NO!

  - Recall that TCP's congestion controls aim to deliver equal shares of BW to all connections sharing each link.

  - If all apps ran over TCP, and no app opened parallel connections, TCP would do a reasonable job of BW sharing.

  One way to do this could be to
  
  * Have each connection track its loss rate (e.g., packets lost/packets sent), and
  * Then apply the relation

    $$TCP - equiv. rate \approx 1.22 \cdot \frac{MSS}{RTT \cdot \sqrt{L}}$$

    you derived in HW3 problem 9.

  * Transmit at rates no greater than these "TCP-equivalent" rates, i.e., mimic TCP rate behavior.
    
    * E.g., in the face of congestion a video app might be required to $\searrow$ its resolution (to $\searrow$ its rate).
    
    * This needn't mean the app get substantially worse QoS. Had it maintained its original rate packet losses would have been higher.

- One other potential weakness of TCP

  * TCP assumes that all losses are due to congestion.

  * As a consequence it performs poorly when operated (unmodified) over error-prone (e.g., wireless) links.

  * The reason? When error-loss occurs, TCP needlessly reduces its rate rather than quickly retransmitting.

  * Proposed fixes include:

113

- In practice, however,

  * Many apps (e.g., multimedia apps) do not run over TCP precisely because TCP may throttle their rate.
    
    * Non-TCP connections (e.g., UDP) need not, when faced with congestion, $\searrow$ their rates (as TCP does).
    
    * Hence, they may consume an unfair share of BW when run side-by-side with "good neighbor" TCP.

  * Many other apps (most notably Web browsers) regularly open multiple parallel TCP connections.
    
    * As TCP strives to equalize per connection BW, the apps with the most connections get the most BW.
    
    * E.g., when 5 1-connection and 15-connection TCP apps compete, the latter gets 1/2 the BW.

- Why is fairness a concern?

  * A single "unruly" connection can degrade the QoS (e.g., loss & delay) of all other link connections.

  * Apps that react inappropriately to congestion increase the Internet's susceptibility to congestive collapse.

- One proposed approach to managing non-TCP-generated congestion would require that all connections

  * Maintain, at all times, estimates of the rate at which they would transmit if they were TCP connections.
- EXPLICIT FEEDBACK: Incorporating an explicit congestion indication into TCP/IP so that congestion no longer need be inferred from loss.

- INDIRECT TCP: Spitting individual TCP connections the traverse wireless links into separate connections, so that TCP behavior can be tailored to the local link, e.g., congestion slows down retransmission, while error-loss speeds it up.

- SNOOPING: Stationing "snooping agents," at the boundaries of wireless links, that cache incoming wireline TCP segs and invisibly retransmit those lost due to errors.

**TCP Timing**

- TCP's timer settings determine whether
  - * Timers timeout prematurely thereby flooding the network with unnecessary retransmissions.
  - * Timers timeout too late introducing undesirable retransmission delays.
  - * Unusual network conditions induce deadlock.
  - * Improperly terminated connections are detected.

- That goes off to remind the sender to probe the receiver for a window size update.
- Is intended to prevent the deadlocks that could arise from the loss of RCVR window size updates.

* A KEEPALIVE TIMER
  - That runs independently in both the sender and receiver.
  - That is reset every time channel activity occurs.
  - That goes off to remind the sender or receiver to check whether their peer entity is still "alive" and terminate the connection if they are not.

* A WAIT STATE TIMER
  - That runs independently in both the sender and receiver.
  - That is used to delay closing a connection for > twice the max frame lifetime to ensure that before closing all connection frames have died.

- UDP (the User Datagram Protocol) is the Internet’s "no frills" transport service.

- UDP is connectionless. It provides
  - * IP datagram service between two endpoints.
  - * No flow control or error recovery (i.e., no ACKs).

- Unless the settings are regularly updated to account for current network conditions performance suffers.

- A RETRANSMISSION TIMER is started every time a TCP segment is sent. A separate timer is maintained for each destination and set to
  
  $$\text{timeout} = RTT + 4D$$

  where $D$, a crude estimate of the standard deviation of the round-trip delay ($RTT$) is maintained recursively as
  
  $$D = \alpha_1 D + (1 - \alpha_1) |RTT - M|,$$

  $RTT$ is updated by the recursion
  
  $$RTT = \alpha_2 RTT + (1 - \alpha_2) M,$$

  $M$ denotes the last measured round trip delay, and $\alpha_1$ and $\alpha_2$ are forgetting factors.

- When it is not clear, due to a retransmission, what ACK should be used to measure the current $RTT$, $M$
  - * $RTT$ is not updated, but
  - * The timeout period is doubled until segs once again get through the network the first time they're sent.

- Other timers include:
  - * A PERSISTENCE TIMER
    - That is started by the sender when it receives a receiver ACK with a window size of 0.

- Optional (checksum) error detection.
    - Checksum applies to the pseudo-header (same as TCP) and the UDP segment.
    - When not in use it is set to zero.

- **UDP Header Format:**

<table>
<thead>
<tr>
<th>32 Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
</tr>
<tr>
<td>UDP length</td>
</tr>
</tbody>
</table>

- * Source Port #: source process ID (16 bits)
- * Destination Port #: destination process ID (16 bits)
- * UDP Length: length of 8 byte UDP header + data (16 bits)
- * Checksum: ones complement of the sum modulo $2^{16} - 1$ of all 16-bit words in
  - Entire UDP segment (with checksum set to 0)
  - Pseudo header consisting of: IP source & destination addresses, protocol & UDP seg length

As for TCP, inclusion of the pseudo header allows the UDP seg to protect itself from misdelivery by IP.