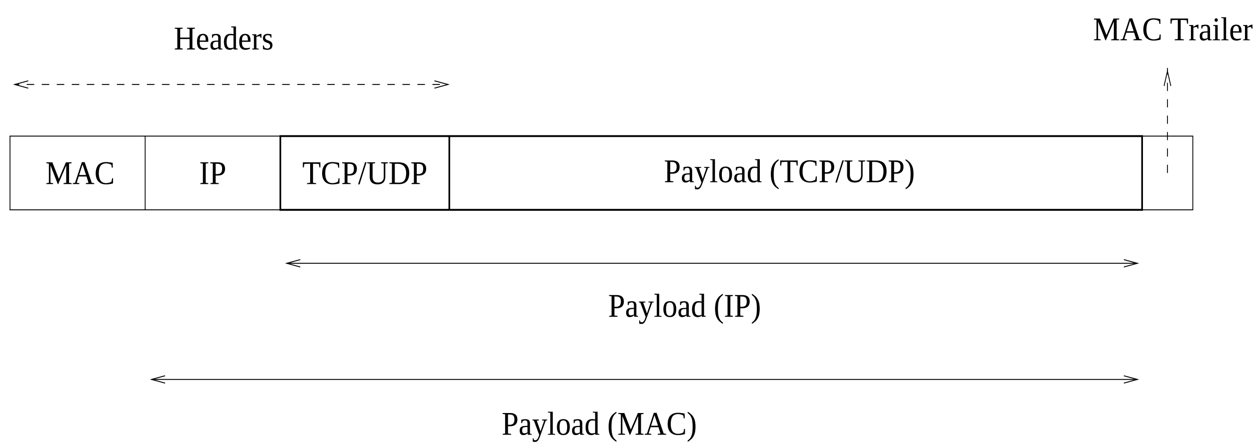
Transport Protocols: TCP and UDP

- ⟶ end-to-end protocol
- ⟶ runs on top of network layer protocols
- ⟶ treat network layer & below as black box

Three-level encapsulation:



- ⟶ meaning of protocol "stack": push/pop headers
- ⟶ common TCP payload: HTTP

Network layer (IP) assumptions:

- unreliable
- out-of-order delivery
- absence of QoS guarantees (delay, throughput, etc.)
- insecure (IPv4)

  → IPsec

Additional performance properties:

- Works “ok”
- Can break down under high load conditions

  → Atlanta Olympics

  → DoS and worm attack

- Wide behavioral range

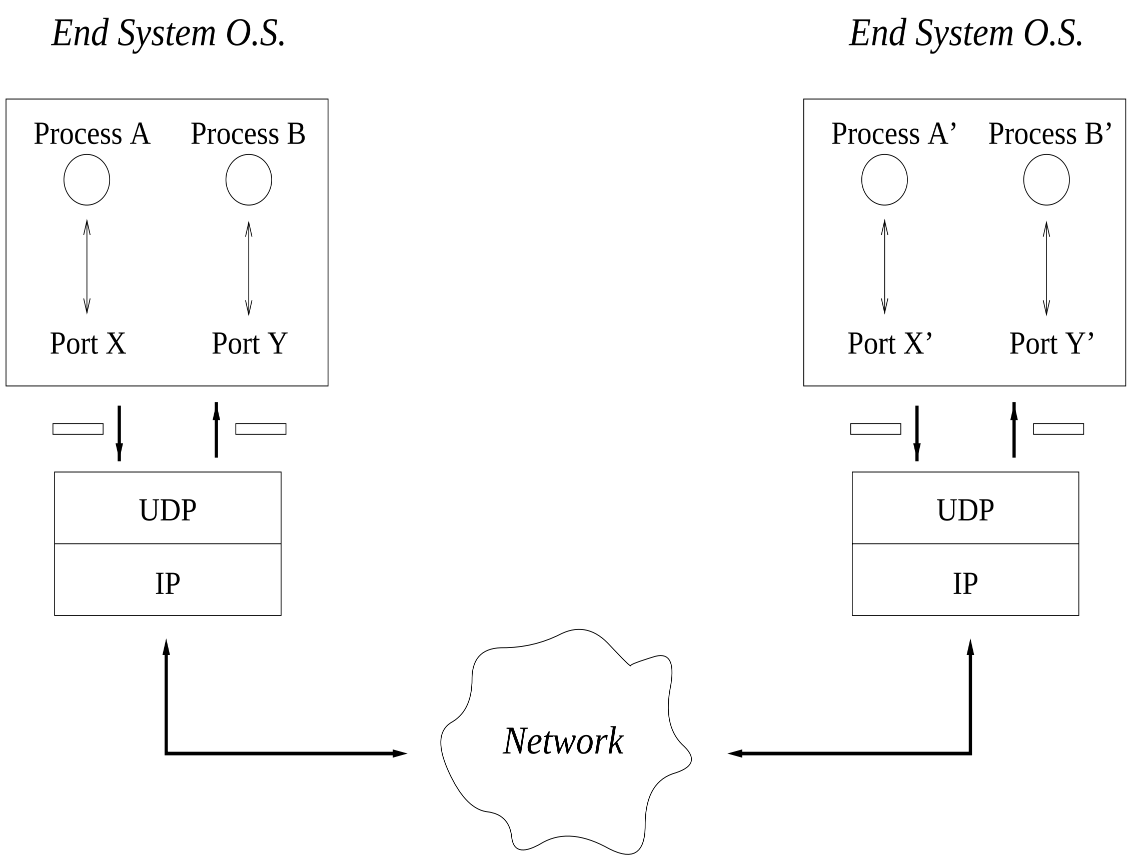
  → sometimes good, so so, or bad

Goal of UDP (User Datagram Protocol):

$\longrightarrow$ process identification

$\longrightarrow$ port number as demux key

$\longrightarrow$ minimal support beyond IP



UDP packet format:

2	2
Source Port	Destination Port
Length	Checksum
Payload	

Checksum calculation (pseudo header):

4		
Source Address		
Destination Address		
$00 \cdots 0$	Protocol	UDP Length

$\longrightarrow$ pseudo header, UDP header and payload

UDP usage:

- multimedia streamining

  → lean and nimble

  → at minimum requires process identification

  → congestion control carried out above UDP

- stateless client/server applications

  → persistent state a hinderance

  → lightweight

Goals of TCP (Transmission Control Protocol):

- process identification
- reliable communication: ARQ
- speedy communication: congestion control
- segmentation

$\longrightarrow$ connection-oriented, i.e., stateful

$\longrightarrow$ complex mixture of functionalities

Segmentation task: provide "stream" interface to higher level protocols

$\longrightarrow$ exported semantics: contiguous byte stream

$\longrightarrow$ recall ARQ

- segment stream of bytes into blocks of fixed size
- segment size determined by TCP MTU (Maximum Transmission Unit)
- actual unit of transmission in ARQ

TCP packet format:

<table>
<tr><th colspan="8">2</th><th>2</th></tr>
<tr><td colspan="8">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="9">Sequence Number</td></tr>
<tr><td colspan="9">Acknowledgement Number</td></tr>
<tr><td>Header Length</td><td></td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Window Size</td></tr>
<tr><td colspan="8">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="9">Options (if any)</td></tr>
<tr><td colspan="9">DATA (if any)</td></tr>
</table>

- Sequence Number: position of first byte of payload
- Acknowledgement: next byte of data expected (receiver)
- Header Length (4 bits): 4 B units
- URG: urgent pointer flag
- ACK: ACK packet flag
- PSH: override TCP buffering
- RST: reset connection
- SYN: establish connection
- FIN: close connection
- Window Size: receiver's advertised window size
- Checksum: prepend pseudo-header
- Urgent Pointer: byte offset in current payload where urgent data begins
- Options: MTU; take min of sender & receiver (default 556 B)

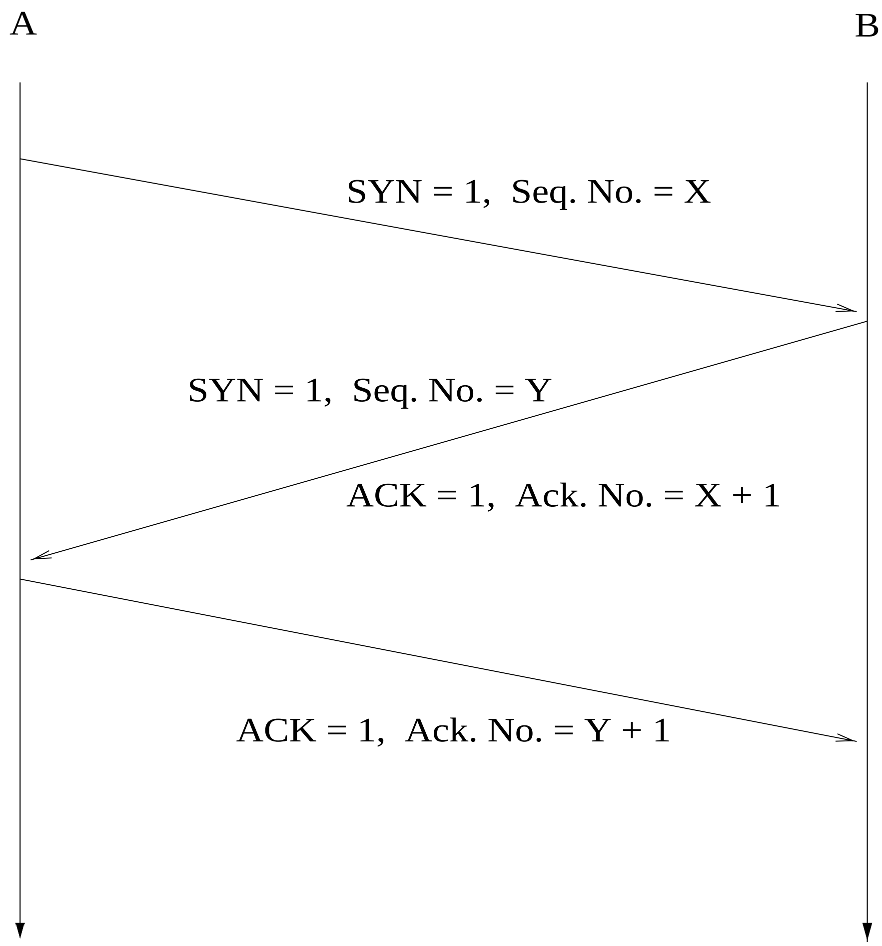
Checksum calculation (pseudo header):

4

<table>
<tr><td colspan="3">Source Address</td></tr>
<tr><td colspan="3">Destination Address</td></tr>
<tr><td>00 $\cdots$ 0</td><td>Protocol</td><td>TCP Segment Length</td></tr>
</table>

$\longrightarrow$ pseudo header, TCP header and payload

TCP connection establishment (3-way handshake):



- X, Y are chosen randomly

  $\longrightarrow$ sequence number prediction
- piggybacking

2-person consensus problem: are A and B in agreement about the state of affairs after 3-way handshake?

$\longrightarrow$ in general: impossible

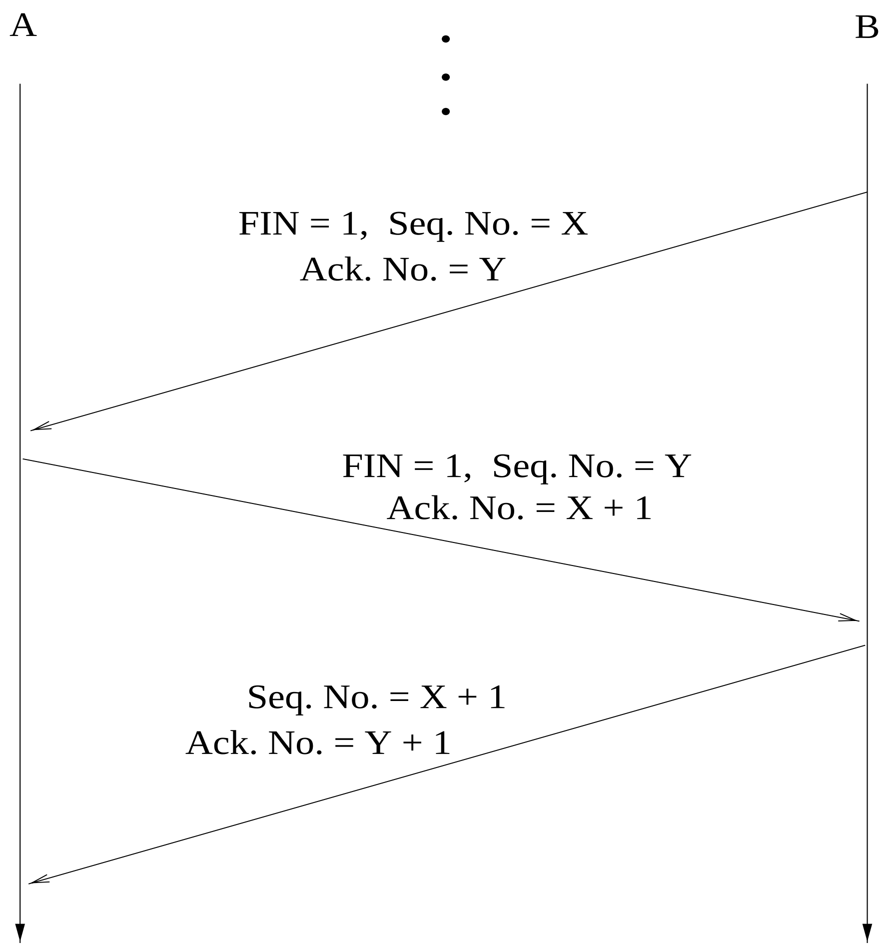
$\longrightarrow$ can be proven

$\longrightarrow$ "acknowledging the ACK problem"

$\longrightarrow$ also TCP session ending

$\longrightarrow$ lunch date problem

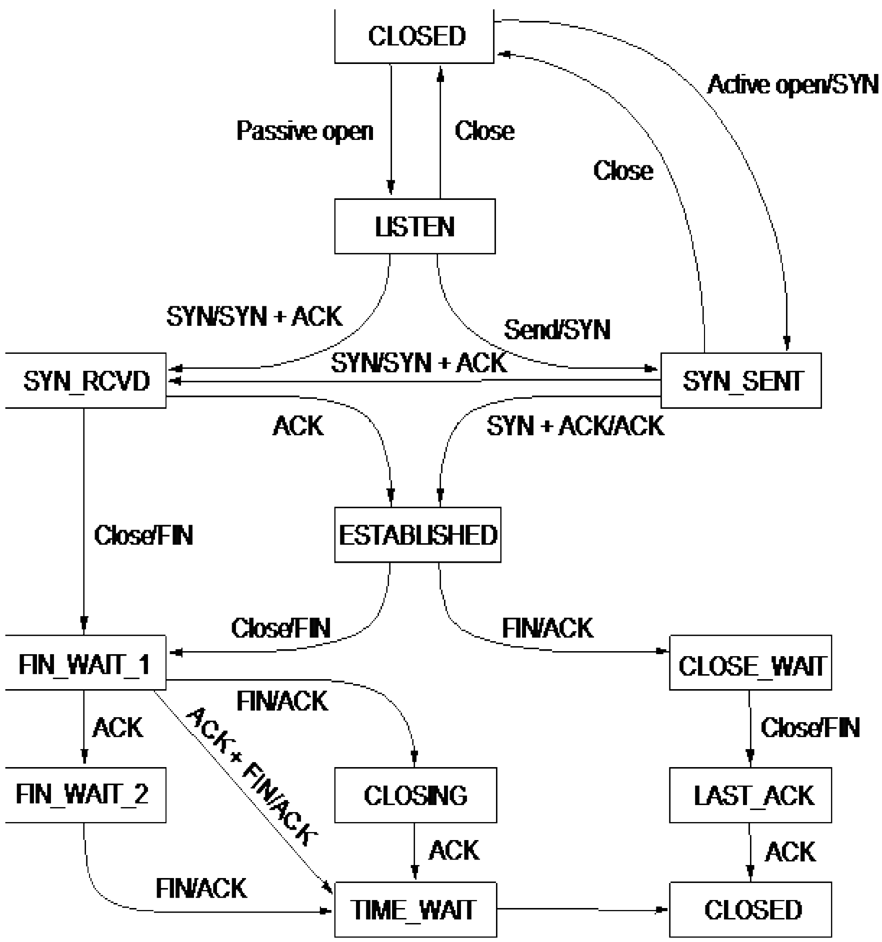
TCP connection termination:



- full duplex
- half duplex

More generally, finite state machine representation of TCP's control mechanism:

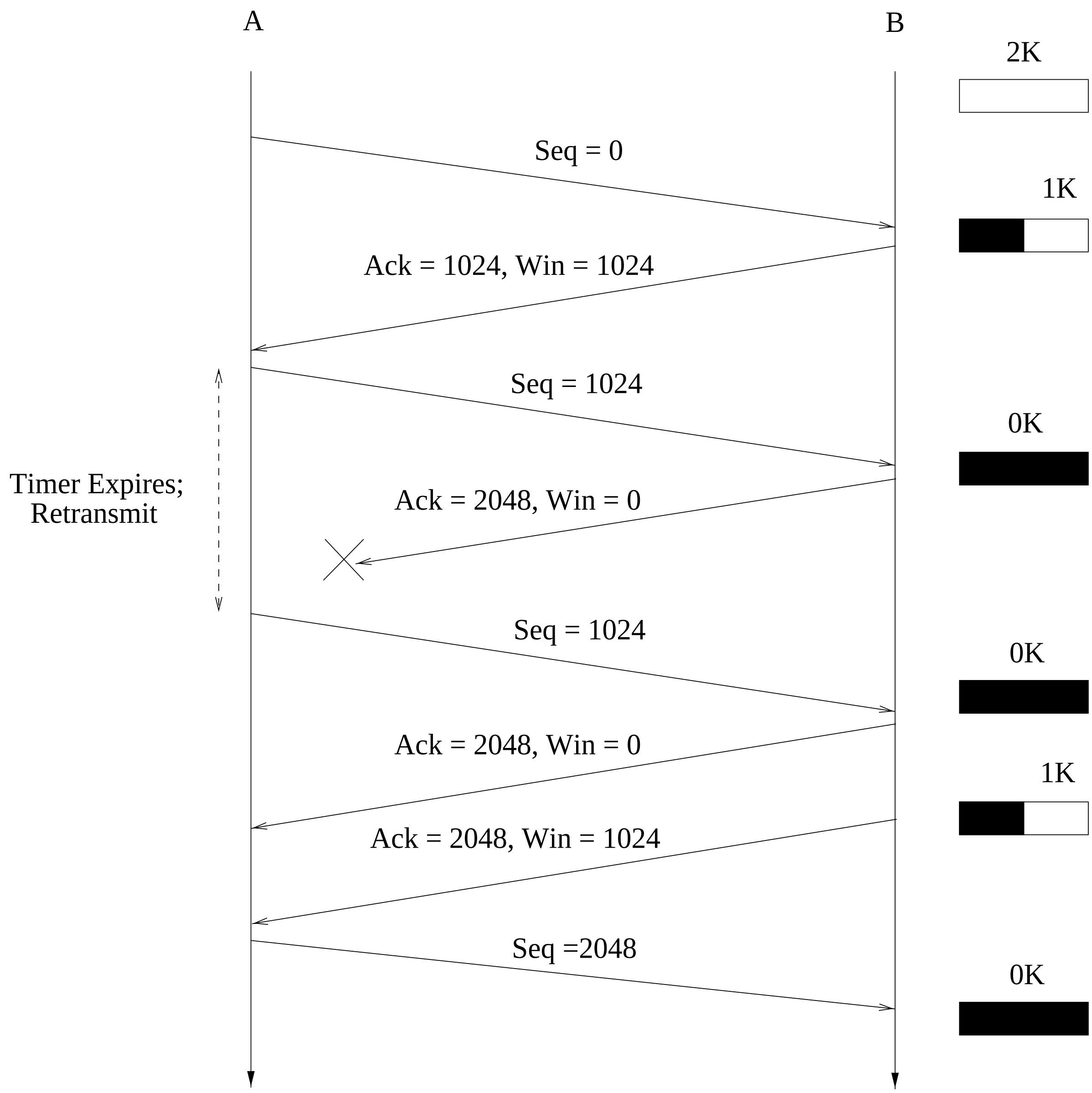
$\longrightarrow$ state transition diagram



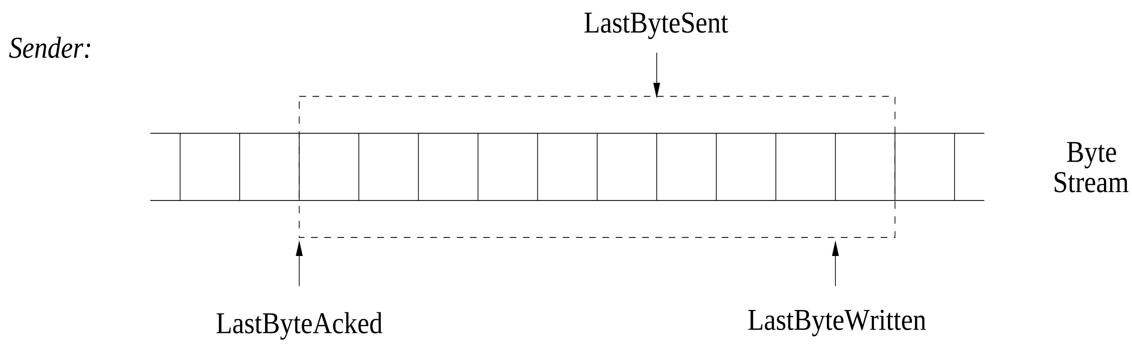
Features to notice:

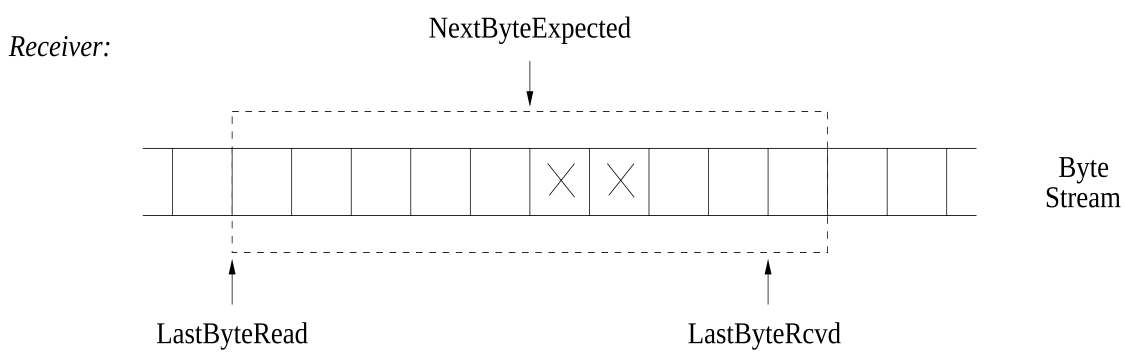
- Connection set-up:
  - client's transition to `ESTABLISHED` state without ACK
  - how is server to reach `ESTABLISHED` if client ACK is lost?
  - `ESTABLISHED` is macrostate (partial diagram)
- Connection tear-down:
  - three normal cases
  - special issue with `TIME WAIT` state
  - employs hack

Basic TCP data transfer:



TCP's sliding window protocol





- sender, receiver maintain buffers `MaxSendBuffer`, `MaxRcvBuffer`

Note asynchrony between TCP module and application.

Sender side: maintain invariants

- $\texttt{LastByteAcked} \le \texttt{LastByteSent} \le \texttt{LastByteWritten}$
- $\texttt{LastByteWritten} - \texttt{LastByteAcked} < \texttt{MaxSendBuffer}$

  $\longrightarrow$ buffer flushing (advance window)

  $\longrightarrow$ application blocking

- $\texttt{LastByteSent} - \texttt{LastByteAcked} \le \texttt{AdvertisedWindow}$

Thus,

$$\begin{aligned}\texttt{EffectiveWindow} = \texttt{AdvertisedWindow} - \\ (\texttt{LastByteSent} - \texttt{LastByteAcked})\end{aligned}$$

$\longrightarrow$ upper bound on new send volume

Actually, one additional refinement:

$\longrightarrow$ `CongestionWindow`

`EffectiveWindow` update procedure:

$$\texttt{EffectiveWindow} = \texttt{MaxWindow} - (\texttt{LastByteSent} - \texttt{LastByteAcked})$$

where

$$\texttt{MaxWindow} = \min\{\,\texttt{AdvertisedWindow},\ \texttt{CongestionWindow}\,\}$$

How to set `CongestionWindow`.

$\longrightarrow$ domain of TCP congestion control

Receiver side: maintain invariants

- $\texttt{LastByteRead} < \texttt{NextByteExpected} \leq \texttt{LastByteRcvd} + 1$
- $\texttt{LastByteRcvd} - \texttt{NextByteRead} < \texttt{MaxRcvBuffer}$

$\longrightarrow$ buffer flushing (advance window)

$\longrightarrow$ application blocking

Thus,

$$\begin{aligned}\texttt{AdvertisedWindow} = \texttt{MaxRcvBuffer} - \\ (\texttt{LastByteRcvd} - \texttt{LastByteRead})\end{aligned}$$

Issues:

How to let sender know of change in receiver window size after `AdvertisedWindow` becomes 0?

- trigger ACK event on receiver side when `AdvertisedWindow` becomes positive
- sender periodically sends 1-byte probing packet

$\longrightarrow$ design choice: smart sender/dumb receiver

$\longrightarrow$ same situation for congestion control

Silly window syndrome: Assuming receiver buffer is full, what if application reads one byte at a time with long pauses?

- can cause excessive 1-byte traffic
- if `AdvertisedWindow` $<$ MSS then set `AdvertisedWindow` $\leftarrow 0$

Do not want to send too many 1 B payload packets.

Nagle's algorithm:

- rule: connection can have only one such unacknowledged packet outstanding
- while waiting for ACK, incoming bytes are accumulated (i.e., buffered)

... compromise between real-time constraints and efficiency.

$\longrightarrow$ useful for `telnet`-type applications

Sequence number wrap-around problem: recall sufficient condition

$$\texttt{SenderWindowSize} < (\texttt{MaxSeqNum} + 1)/2$$

$\longrightarrow$ 32-bit sequence space/16-bit window space

However, more importantly, time until wrap-around important due to possibility of roaming packets.

bandwidth	time until wrap-around †
T1 (1.5 Mbps)	6.4 hrs
Ethernet (10 Mbps)	57 min
T3 (45 Mbps)	13 min
F/E (100 Mbps)	6 min
OC-3 (155 Mbps)	4 min
OC-12 (622 Mbps)	55 sec
OC-24 (1.2 Gbps)	28 sec

RTT estimation

... important to not underestimate nor overestimate.

Karn/Partridge: Maintain running average with precautions

$$\texttt{EstimateRTT} \leftarrow \alpha \cdot \texttt{EstimateRTT} + \beta \cdot \texttt{SampleRTT}$$

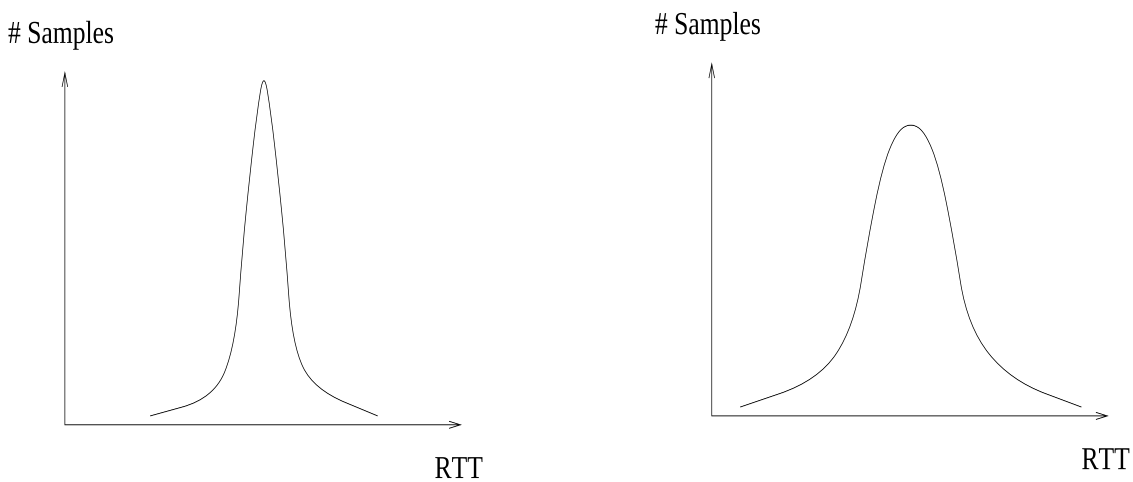
- `SampleRTT` computed by sender using timer
- $\alpha + \beta = 1$; $0.8 \leq \alpha \leq 0.9$, $0.1 \leq \beta \leq 0.2$
- $\texttt{TimeOut} \leftarrow 2 \cdot \texttt{EstimateRTT}$ or
  $\texttt{TimeOut} \leftarrow 2 \cdot \texttt{TimeOut}$ (if retransmit)

$\longrightarrow$ need to be careful when taking `SampleRTT`

$\longrightarrow$ infusion of complexity

$\longrightarrow$ still remaining problems

Hypothetical RTT distribution:



$\longrightarrow$ need to account for variance

$\longrightarrow$ not nearly as nice

Jacobson/Karels:

- $\texttt{Difference} = \texttt{SampleRTT} - \texttt{EstimatedRTT}$
- $\texttt{EstimatedRTT} = \texttt{EstimatedRTT} + \delta \cdot \texttt{Difference}$
- $\texttt{Deviation} = \texttt{Deviation} + \delta(|\texttt{Difference}| - \texttt{Deviation})$

Here $0 < \delta < 1$.

Finally,

- $\texttt{TimeOut} = \mu \cdot \texttt{EstimatedRTT} + \phi \cdot \texttt{Deviation}$

where $\mu = 1$, $\phi = 4$.

$\longrightarrow$ persistence timer

$\longrightarrow$ how to keep multiple timers in UNIX