TCP connection establishment (3-way handshake):

A   B

SYN = 1, Seq. No. = X

SYN = 1, Seq. No. = Y

ACK = 1, Ack. No. = X + 1

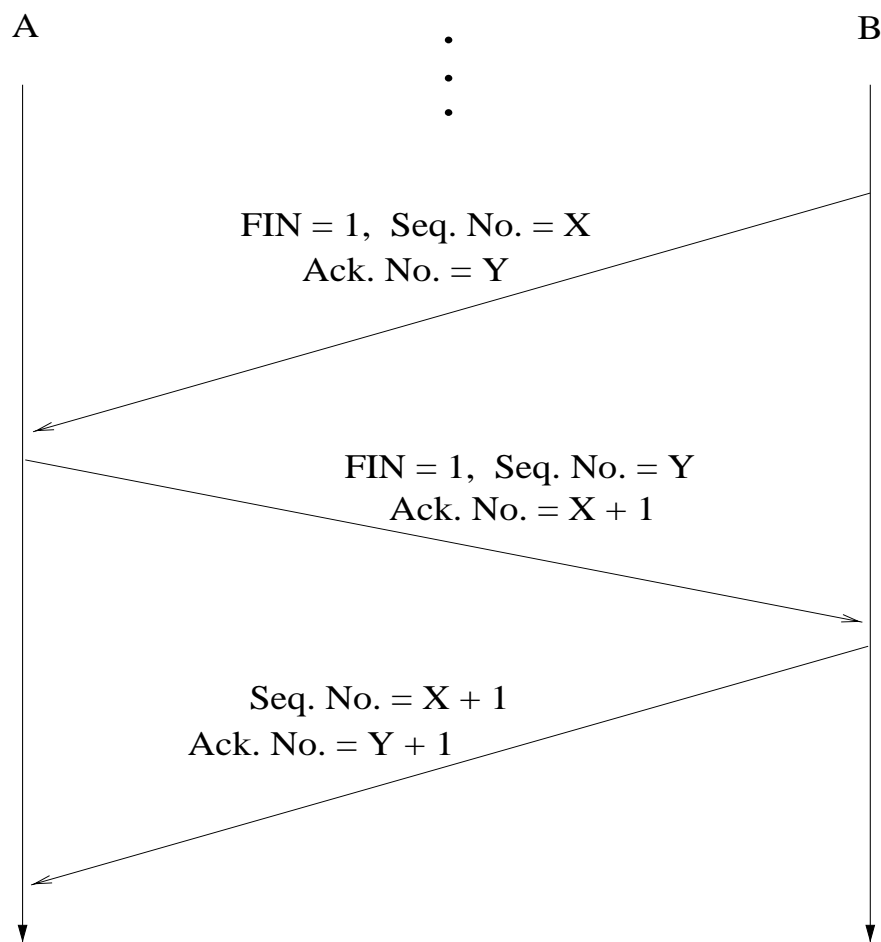ACK = 1, Ack. No. = Y + 1
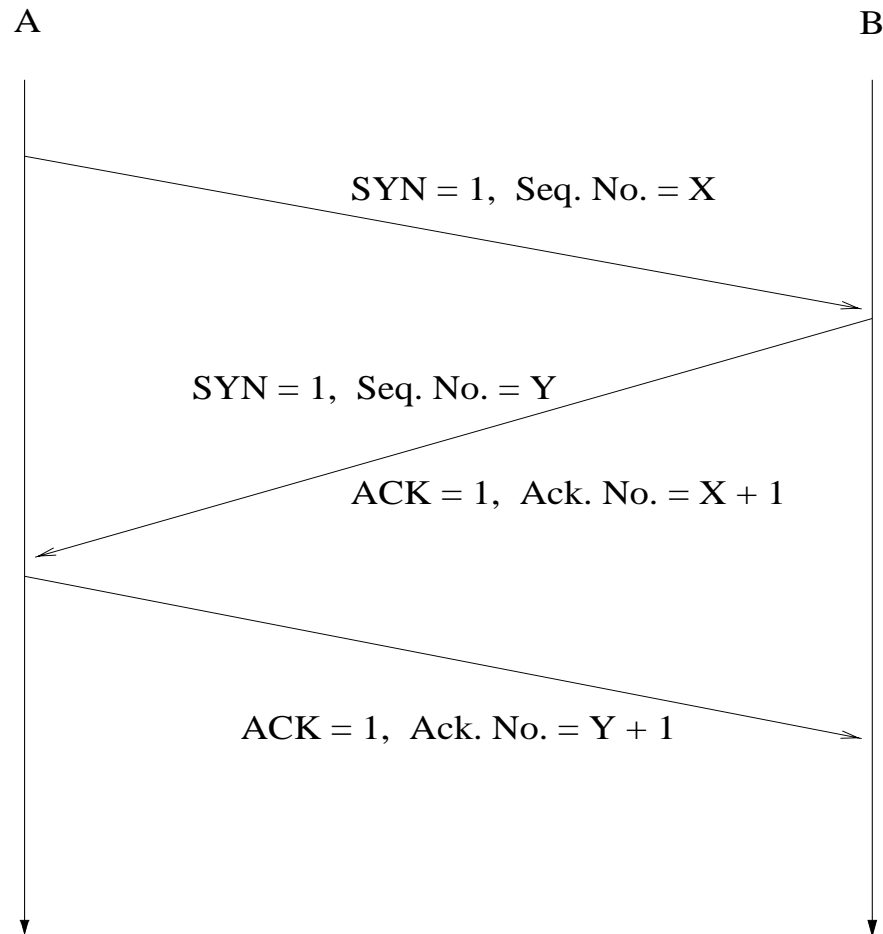
- X, Y are chosen randomly

  $\rightarrow$ 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:

A   B
$$\vdots$$

FIN = 1, Seq. No. = X
Ack. No. = Y

FIN = 1, Seq. No. = Y
Ack. No. = X + 1

Seq. No. = X + 1
Ack. No. = Y + 1

• full duplex

• half duplex

More generally, finite state machine representation of TCP's control mechanism:
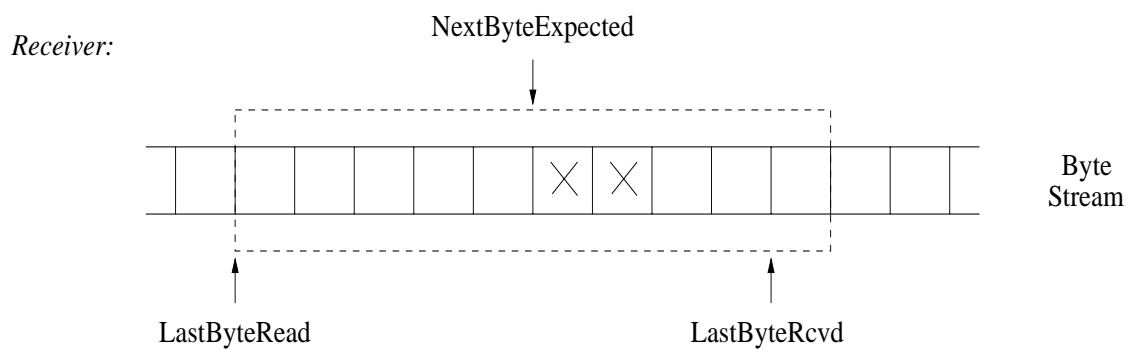
$\longrightarrow$ state transition diagram
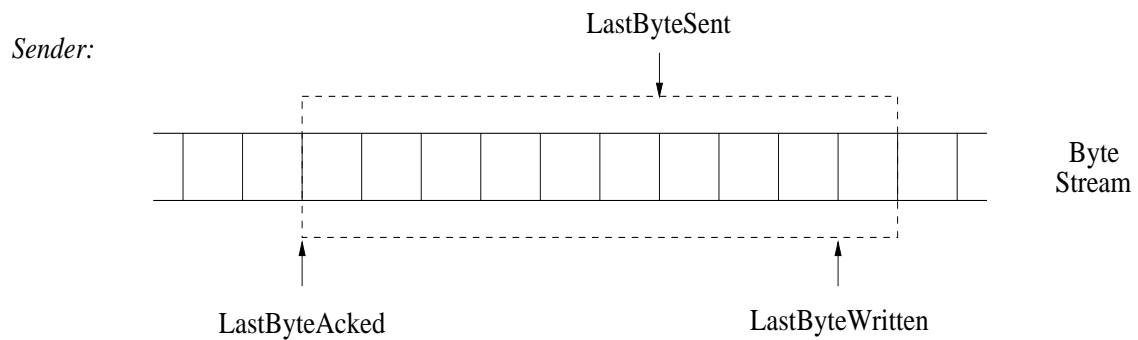
Features to notice:

- 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

TCP's sliding window protocol

Sender:

LastByteSent

Byte
Stream

LastByteAcked   LastByteWritten

Receiver:

NextByteExpected

Byte
Stream

LastByteRead   LastByteRcvd

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

Note asynchrony between TCP module and application.

Sender side: maintain invariants

- $\texttt{LastByteAcked} \leq \texttt{LastByteSent} \leq \texttt{LastByteWritten}$

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

    $\longrightarrow$ buffer flushing (advance window)

    $\longrightarrow$ application blocking

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

Thus,

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

    $\longrightarrow$ upper bound on new send volume

Actually, one additional refinement:

$\longrightarrow$ `CongestionWindow`

`EffectiveWindow` update procedure:

`EffectiveWindow` = `MaxWindow`−

$\qquad$ (`LastByteSent` − `LastByteAcked`)

where

`MaxWindow` =

$\qquad$ min{ `AdvertisedWindow`, `CongestionWindow` }

How to set `CongestionWindow`.

$\longrightarrow$ domain of TCP congestion control

Receiver side: maintain invariants

- $\mathtt{LastByteRead} < \mathtt{NextByteExpected} \leq$
  $\mathtt{LastByteRcvd} + 1$

- $\mathtt{LastByteRcvd} - \mathtt{NextByteRead} < \mathtt{MaxRcvBuffer}$

  $\longrightarrow$ buffer flushing (advance window)

  $\longrightarrow$ application blocking

Thus,

$$\mathtt{AdvertisedWindow} = \mathtt{MaxRcvBuffer}-$$
$$(\mathtt{LastByteRcvd} - \mathtt{LastByteRead})$$

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 method:

- 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.

$\rightarrow$ useful for `telnet/ssh`-type interactive 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

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

- $\texttt{SampleRTT}$ computed by sender using timer

- $\alpha + \beta = 1;\ \ 0.8 \leq \alpha \leq 0.9,\ 0.1 \leq \beta \leq 0.2$

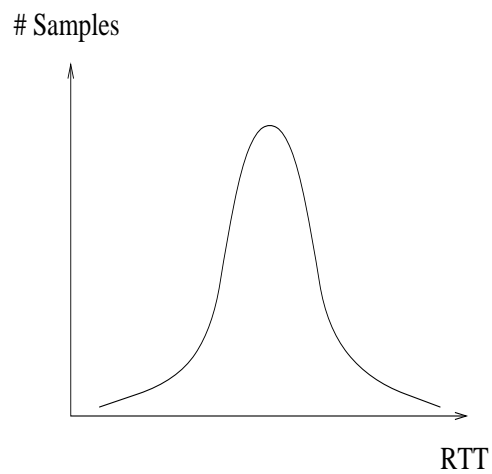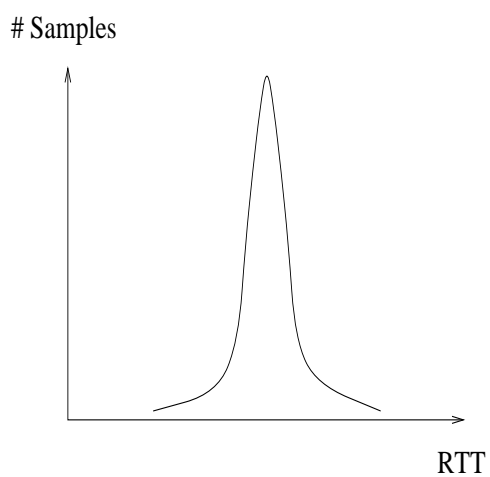- $\texttt{TimeOut} \leftarrow 2 \cdot \texttt{EstimateRTT}$ or

  $\texttt{TimeOut} \leftarrow 2 \cdot \texttt{TimeOut}$ (if retransmit)

    $\longrightarrow$ need to be careful when taking $\texttt{SampleRTT}$

    $\longrightarrow$ 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