TCP congestion control

Recall:

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

where

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

Key question: how to set `CongestionWindow` which, in turn, affects ARQ's sending rate?

$\longrightarrow$ linear increase/exponential decrease

$\longrightarrow$ AIMD

TCP congestion control components:

(i) Congestion avoidance

$\longrightarrow$ linear increase/exponential decrease

$\longrightarrow$ additive increase/exponential decrease (AIMD)

As in Method B, increase `CongestionWindow` linearly, but decrease exponentially

Upon receiving ACK:

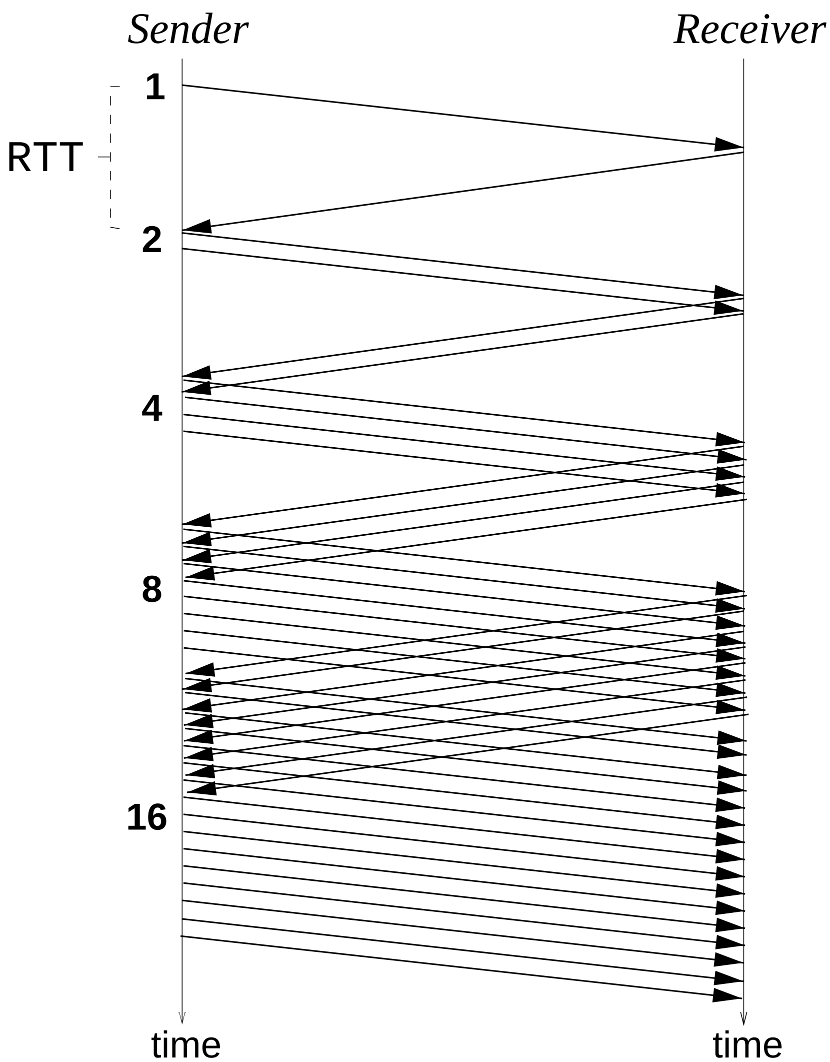
$$\texttt{CongestionWindow} \leftarrow \texttt{CongestionWindow} + 1$$

Upon timeout:

$$\texttt{CongestionWindow} \leftarrow \texttt{CongestionWindow} \,/\, 2$$

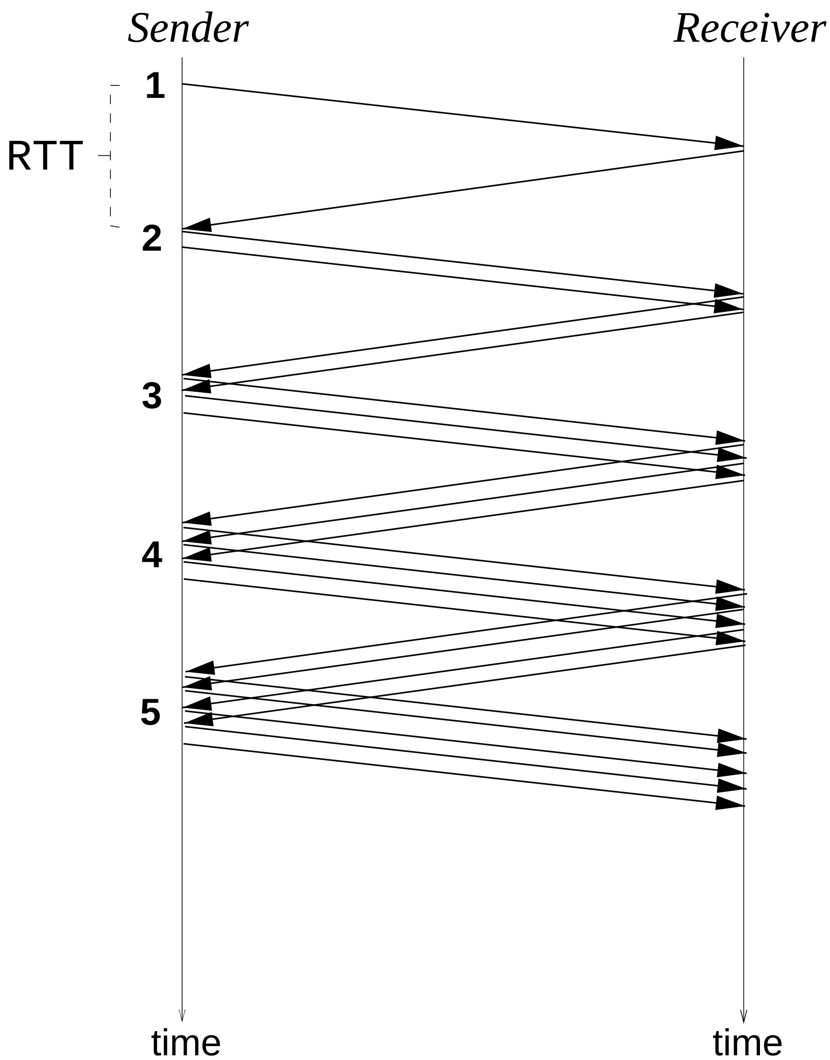
But is it correct. . .

“Linear increase” time diagram:



⟶ results in exponential increase

What we want:



⟶ increase by 1 every window

Thus, linear increase update:

$$\texttt{CongestionWindow} \leftarrow \texttt{CongestionWindow} + (1\,/\,\texttt{CongestionWindow})$$

Upon timeout and exponential backoff,

$$\texttt{SlowStartThreshold} \leftarrow \texttt{CongestionWindow}\,/\,2$$

(ii) Slow Start

Reset `CongestionWindow` to 1

Perform exponential increase

$$\texttt{CongestionWindow} \leftarrow \texttt{CongestionWindow} + 1$$

- Until timeout at start of connection

  $\rightarrow$ rapidly probe for available bandwidth

- Until `CongestionWindow` hits `SlowStartThreshold` following Congestion Avoidance

  $\rightarrow$ rapidly climb to safe level

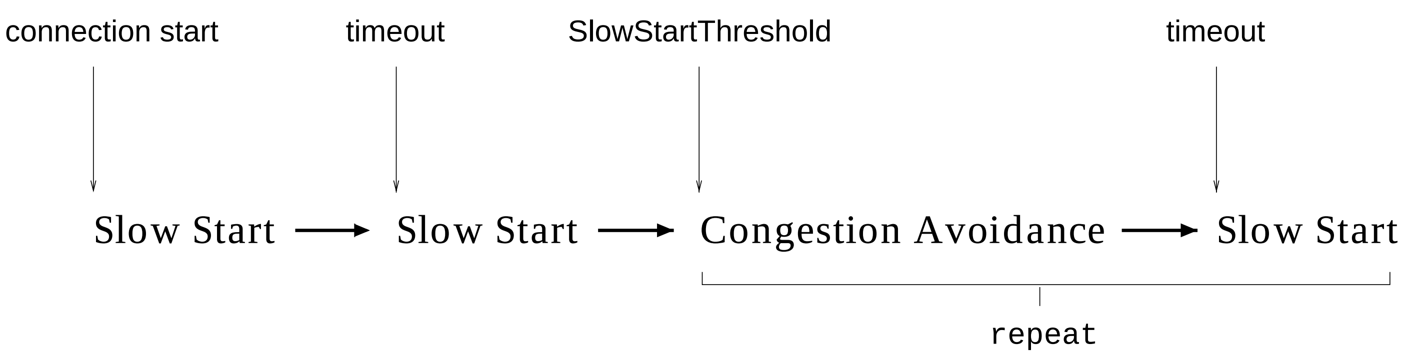
$\longrightarrow$ "slow" is a misnomer

$\longrightarrow$ exponential increase is super-fast

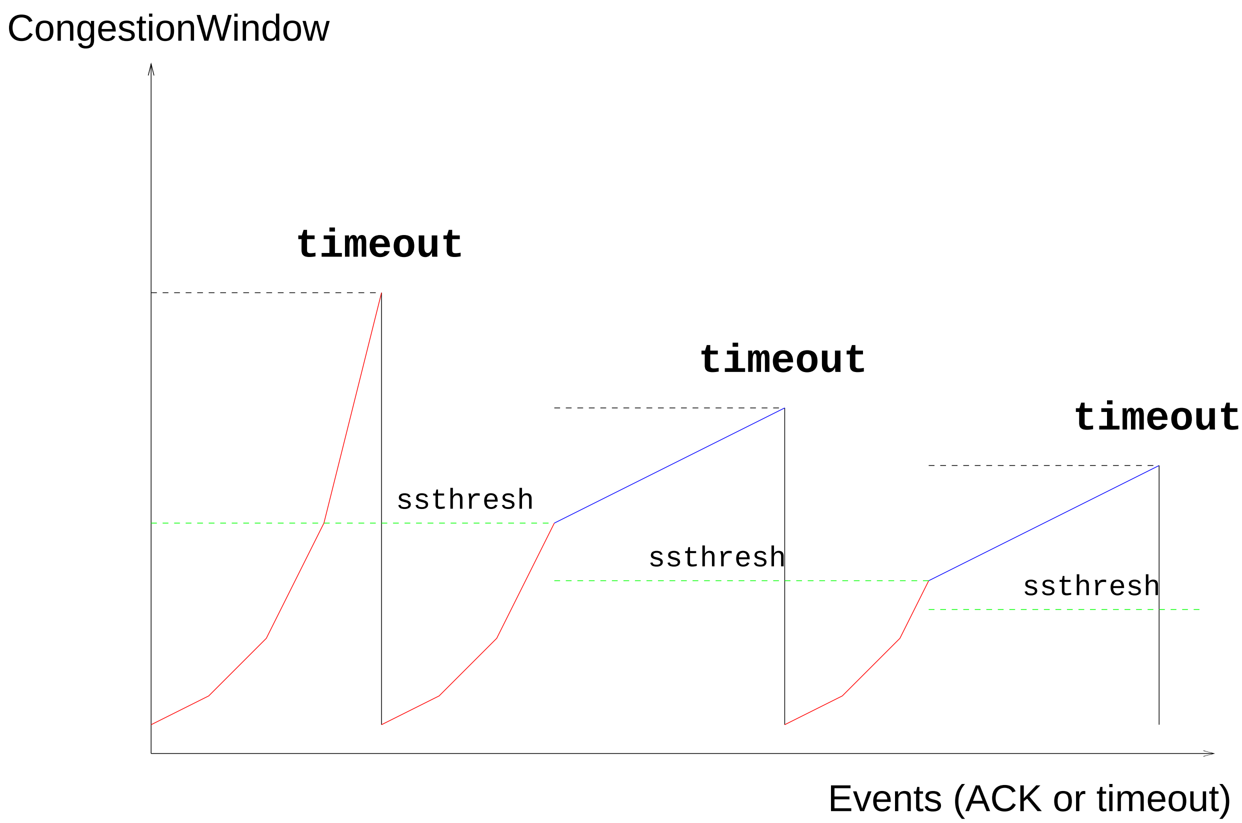
Basic dynamics:

⟶ after connection set-up

⟶ before connection tear-down



`CongestionWindow` evolution:



$\longrightarrow$ what happens if receiver window size hits max?

$\longrightarrow$ DOE, supercomputing centers, etc.

(iii) Exponential timer backoff

$$\texttt{TimeOut} \leftarrow 2 \cdot \texttt{TimeOut} \qquad \text{if retransmit}$$

(iv) Fast Retransmit

Upon receiving three duplicate ACKs:

- Transmit next expected segment

  $\rightarrow$ segment indicated by ACK value

- Perform exponential backoff and commence Slow Start

  $\longrightarrow$ three duplicate ACKs: likely segment is lost

  $\longrightarrow$ react before timeout occurs

TCP Tahoe: features (i)-(iv)

(v) Fast Recovery

Upon Fast Retransmit:

- Skip Slow Start and commence Congestion Avoidance

  $\longrightarrow$ dup ACKs: likely spurious loss

- Insert "inflationary" phase just before Congestion Avoidance

Inflationary phase:

- `SlowStartThreshold` $\leftarrow$ `CongestionWindow` / 2
- `CongestionWindow` $\leftarrow$ `SlowStartThreshold` + 3
- On each additional duplicate ACK, increment `CongestionWindow`
- On first non-dup ACK, commence Congestion Avoidance

  `CongestionWindow` $\leftarrow$ `SlowStartThreshold`

TCP Reno: features (i)-(v)

$\longrightarrow$ pre-dominant form

Many more versions of TCP:

$\longrightarrow$ NewReno w/ SACK, w/o SACK, Vegas, etc.

$\longrightarrow$ wireless, ECN, multiple time scale

$\longrightarrow$ mixed verdict; pros/cons

Given sawtooth behavior of TCP's linear increase/exponential backoff:

Why use exponential backoff and not Method D?

- For multimedia streaming (e.g., pseudo real-time), AIMD (Method B) is not appropriate

  $\longrightarrow$ use Method D

- For unimodal case—throughput decreases when system load is excessive—story is more complicated

  $\longrightarrow$ asymmetry in control law needed for stability