TCP congestion control:

→ general data transport

→ focus: speedy transport of large files

Issue:

- too fast: significant losses lead to degraded performance

  → e.g., TCP congestion collapse

- too slow: underutilize bandwidth

  → especially pronounced in today's high-speed networks

Interface between TCP sliding window and congestion control:

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

where

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

→ how to adjust CongestionWindow to increase throughput

→ prevent congestion collapse

→ based on method B

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:

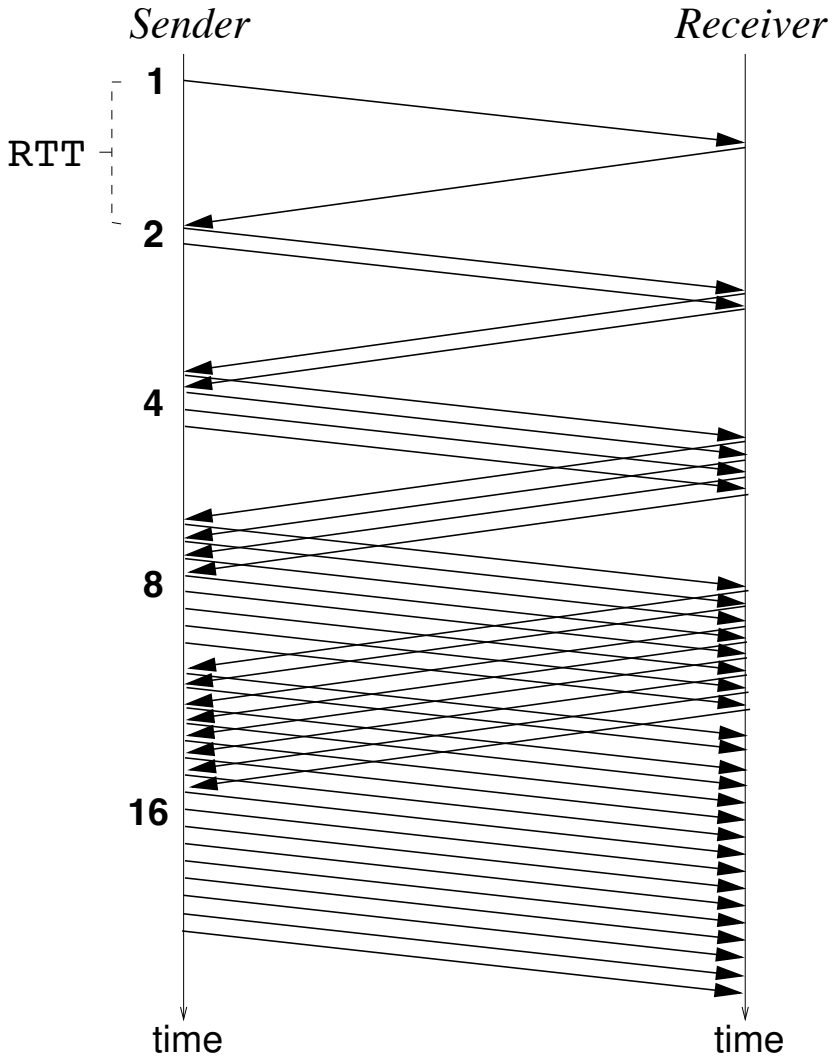
$$\texttt{CongestionWindow} \leftarrow \texttt{CongestionWindow} + 1$$

Upon timeout:

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

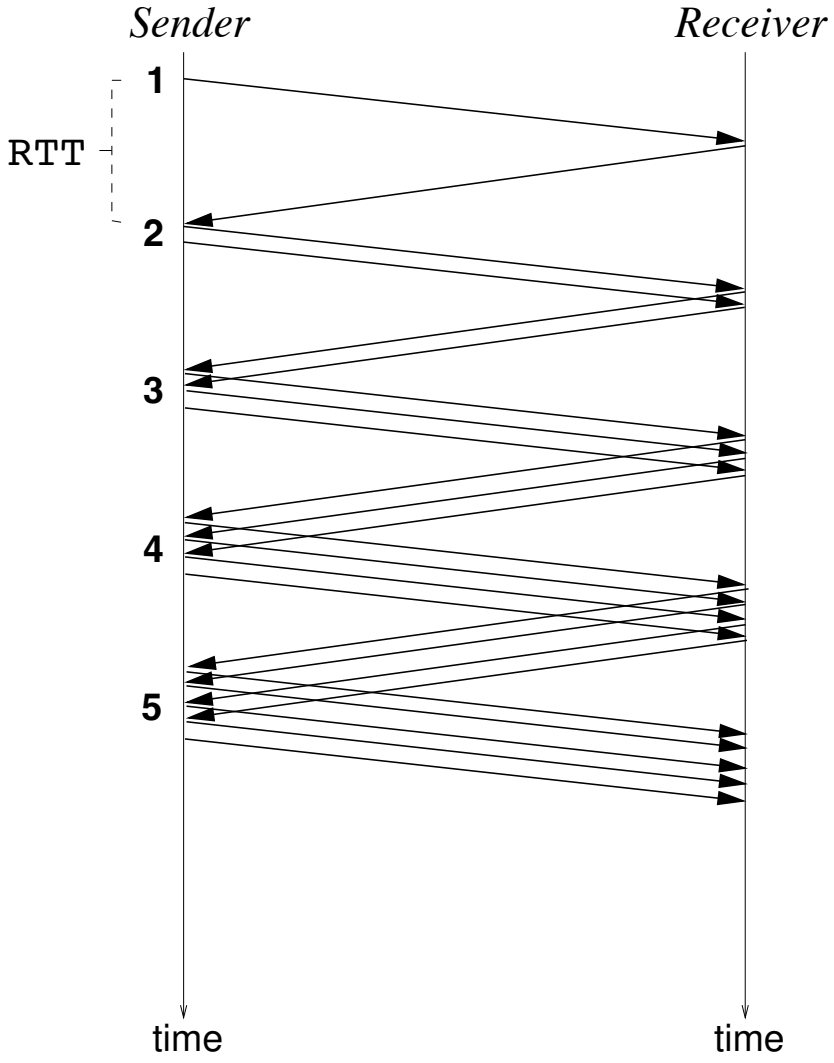
But is it correct...

"Linear increase" time diagram:



$\longrightarrow$ results in exponential increase

What we want:



$\longrightarrow$ 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

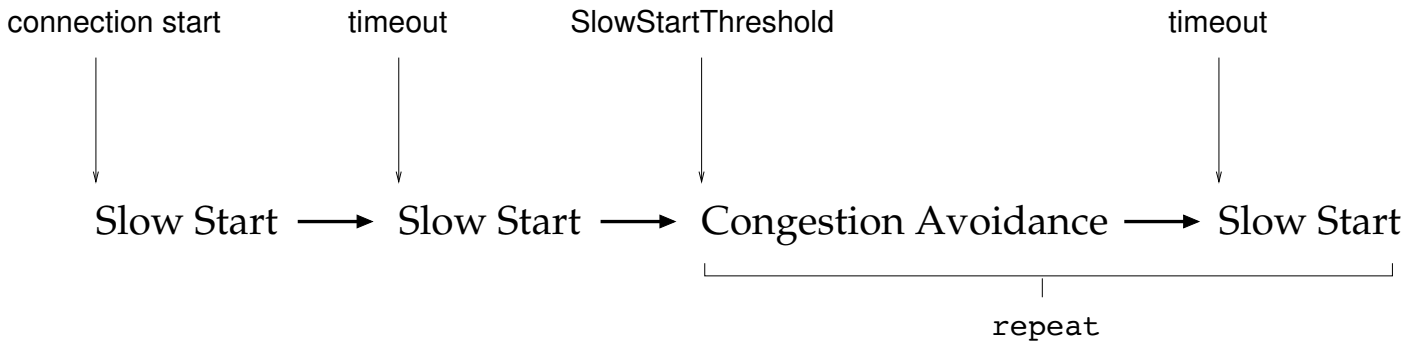
  $\longrightarrow$ "slow" is a misnomer

  $\longrightarrow$ exponential increase is super-fast

Basic dynamics:

$\longrightarrow$ after connection set-up

$\longrightarrow$ before connection tear-down

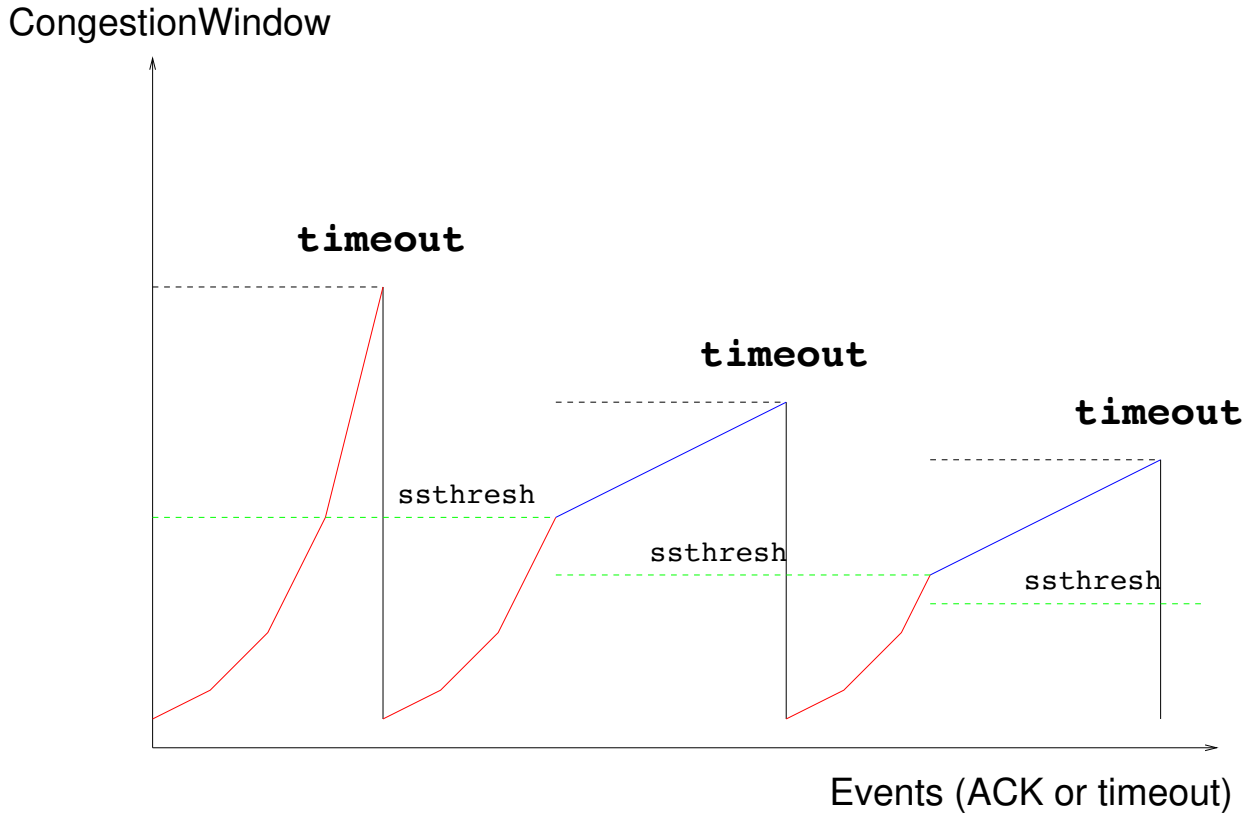


$\longrightarrow$ many TCP transfers are small

$\longrightarrow$ small TCP flows don't escape `Slow Start`

`CongestionWindow` evolution:

$\longrightarrow$ relevant for larger flows



(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, continue sending new data before switching to Congestion Avoidance

→ assumption: dup ACKs indicate spurious loss

→ can afford to be more aggressive

→ avoid stalling

Set `CongestionWindow` to `SlowStartThreshold` + 3

- account for 3 dup ACKs

  → allows transmission of new data

- keep incrementing `CongestionWindow` for additional dup ACKs
- upon receiving new ACK transition to Congestion Avoidance

  ⟶ TCP Reno

Recent variants:

→ more aggressive than plain linear increase

→ targeted at modern high-speed networks

TCP BIC:

- Upon loss set:
  - → $W_{\max}$ congestion window size at time of loss
  - → $W_{\min}$ congestion window size after multiplicative backoff
- Perform binary search between $W_{\min}$ and $W_{\max}$ to rapidly identify sustainable high throughput window size
  - → updated congestion window: $(W_{\min} + W_{\max})/2$
  - → upper bound window increase

TCP Cubic:

→ based on BIC

→ window increase follows cubic function

- from W_{min} to W_{max}

  → concave increase: similar to binary search

- above W_{max}

  → convex increase: rapid probing similar to slow start

- congestion window update not determined by RTT

  → use time elapsed since last window reduction

  → akin to open-loop window increase

. . . default TCP congestion control in Linux, MacOS, Windows.

Case for exponential backoff

- For multimedia streaming (e.g., pseudo real-time) with limited prefetch, AIMD (Method B) not suited

  → can use Method D, variants

  → under long prefetch, can use reliable transport (e.g., TCP)

- For unimodal case—throughput decreases when system load is excessive—instability concern

  → asymmetry in control law to curb instability

  → worst-case: congestion collapse

Potential for unfairness

→ unintentional: TCP friendliness

→ intentional: deployment hurdle for noncooperative actor

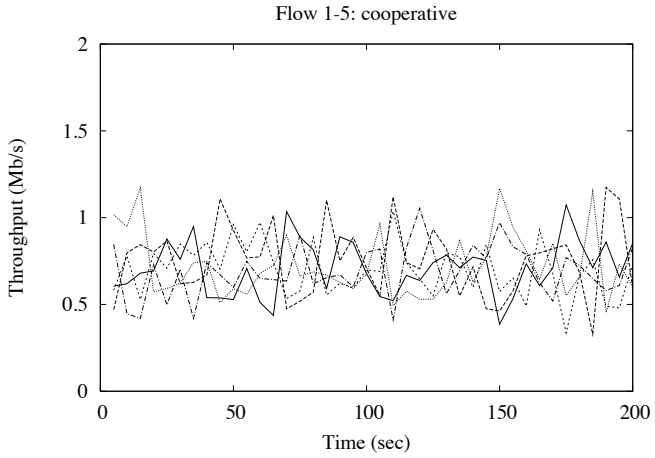
In general: subject of noncooperative game theory

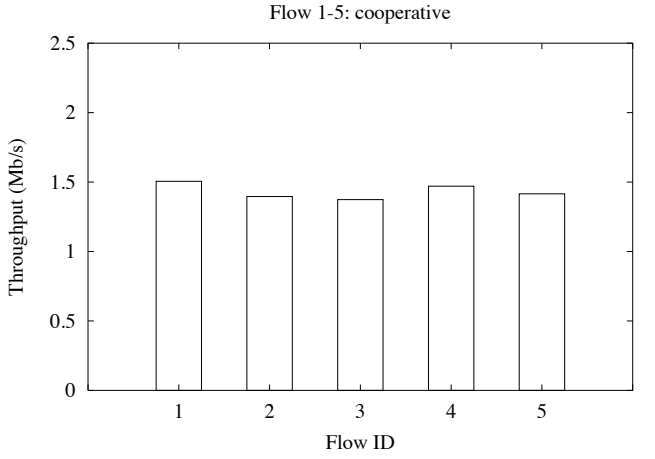
→ e.g., mechanism design

→ hurdles to exploiting selfishness

→ disincentivize

Throughput share of 5 homogenous TCP flows:





4 regular (cooperative) TCP flows and 1 noncooperative TCP flow:

$\rightarrow$ starts behaving selfishly at time 100s