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

Example: aggravated in data center networks

$\rightarrow$ called TCP incast

Data center network topology evolved to accommodate horizontal (or east-west traffic)

- Clos/leaf-spine connectivity

  $\rightarrow$ facilitates east-west traffic flow of data center apps

- application traffic can generate many-to-one simultaneous data transfer

  $\rightarrow$ hot spot at spine switches

  $\rightarrow$ significant packet losses and degraded throughput

After aggressive backoff to prevent TCP throughput collapse, want to recover faster than linear increase to utilize high speed links.

$\rightarrow$ important application domain of congestion control

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:

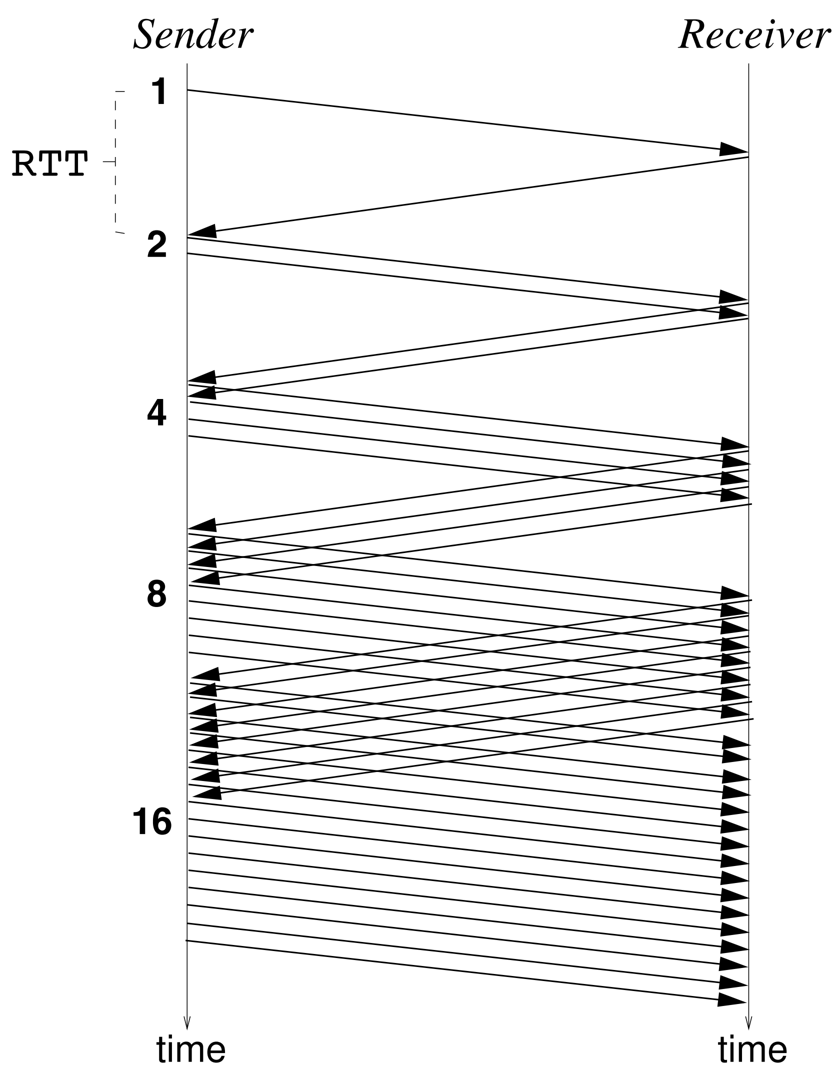
$$\texttt{CongestionWindow} \leftarrow \texttt{CongestionWindow} + 1$$

Upon timeout:

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

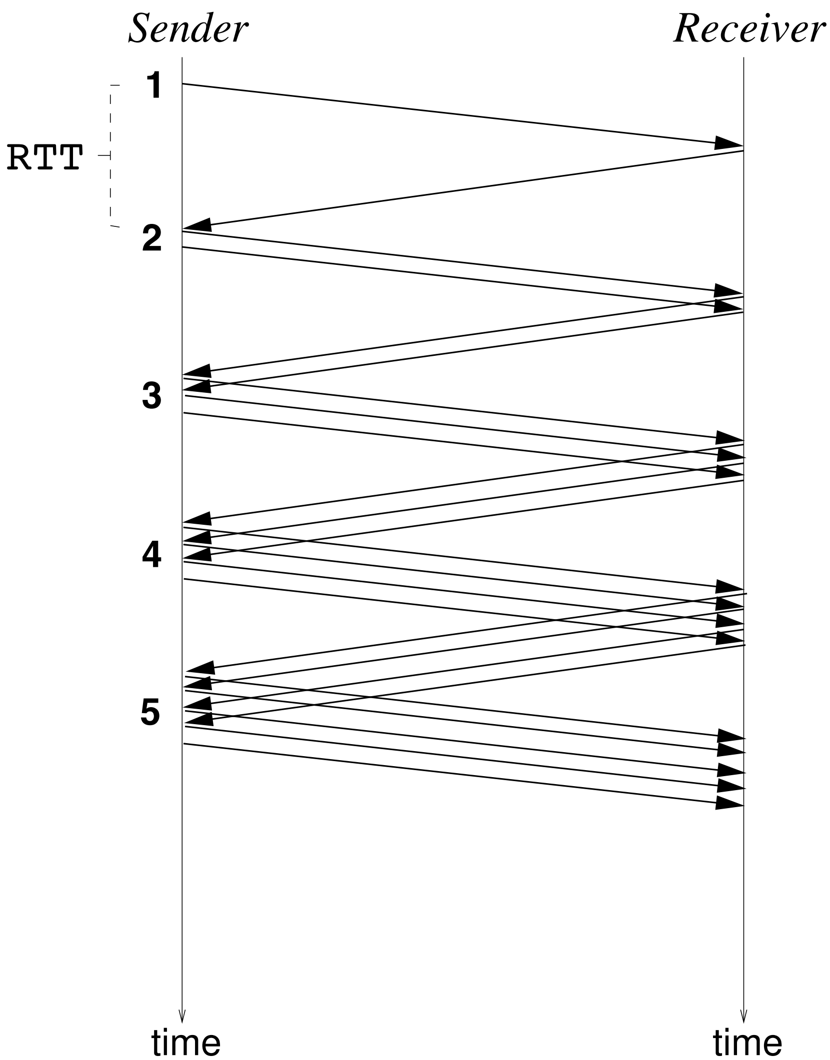
But is it correct…

"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

  → rapidly probe for available bandwidth

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

  → rapidly climb to safe level

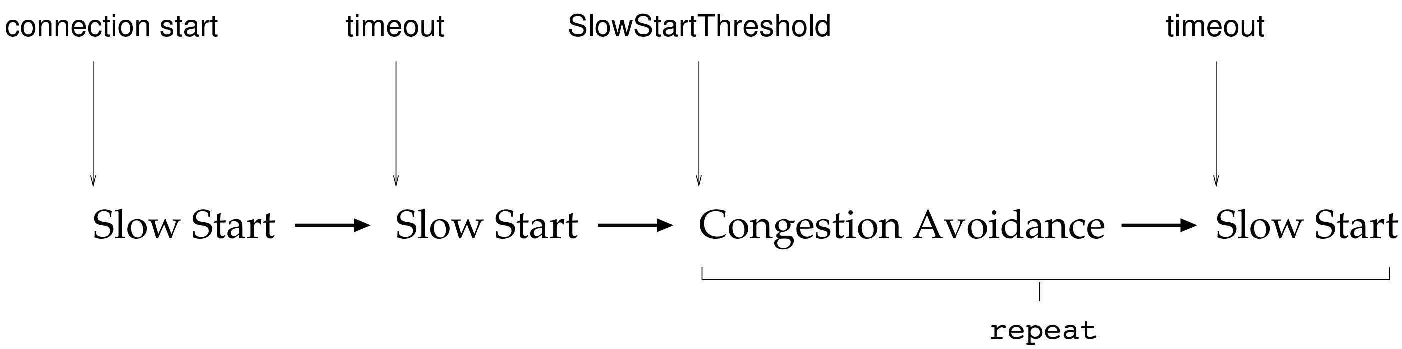
⟶ "slow" is a misnomer

⟶ exponential increase is super-fast

Basic dynamics:

⟶ after connection set-up

⟶ before connection tear-down

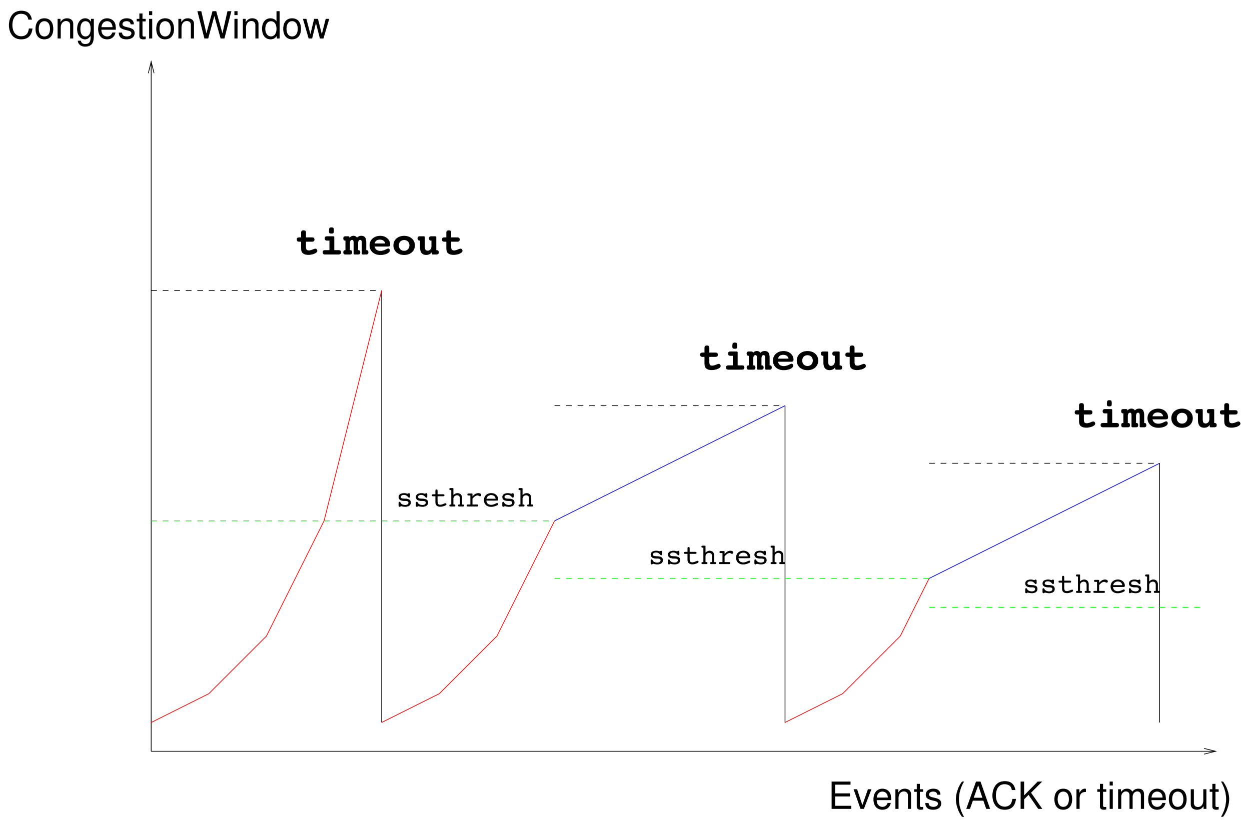


⟶ many TCP transfers are small

⟶ small TCP flows don't escape `Slow Start`

`CongestionWindow` evolution:

⟶ 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

  → segment indicated by ACK value

- Perform exponential backoff and commence Slow Start

⟶ three duplicate ACKs: likely segment is lost

⟶ 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

$\longrightarrow$ TCP Reno

TCP New Reno:

- stay in Fast Recovery longer

  → handle multiple spurious losses within packet train

  → delay transition to Congestion Avoidance

- selective ACK (SACK)

  → receiver informs sender of contigous blocks of data received

  → i.e., sender able to identify holes

. . . drawback: more complex

Recent variants:

→ more aggressive than plain linear increase

→ targeted at modern high-speed networks

TCP BIC:

- Upon loss set:

  → $W_{\rm max}$ congestion window size at time of loss

  → $W_{\rm min}$ congestion window size after multiplicative backoff

- Perform binary search between $W_{\rm min}$ and $W_{\rm max}$ to rapidly identify sustainable high throughput window size

  → updated congestion window: $(W_{\rm min} + W_{\rm max})/2$

  → 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.

Compound TCP: delay-based

- estimate queueing delay from RTT

  → closer to method D

  → other variants of delay based TCP (e.g., Vegas)

- hybrid: uses sum of two windows

  → delay based window that increases rapidly

  → loss based window following linear increase

  → susceptible to congestion collapse

Used in Windows 10 until transition to TCP Cubic.

Data Center TCP (DCTCP):

→ invoke router/switch assistance: AQM

→ ECN (explicit congestion notification)

→ established at TCP connection set-up

→ IPv4: 2-bit ECN in TOS field

Congestion control: selfishness and fairness

- different versions of TCP co-exist on Internet
- aggressiveness that increases throughput for one connection may come at the expense of another

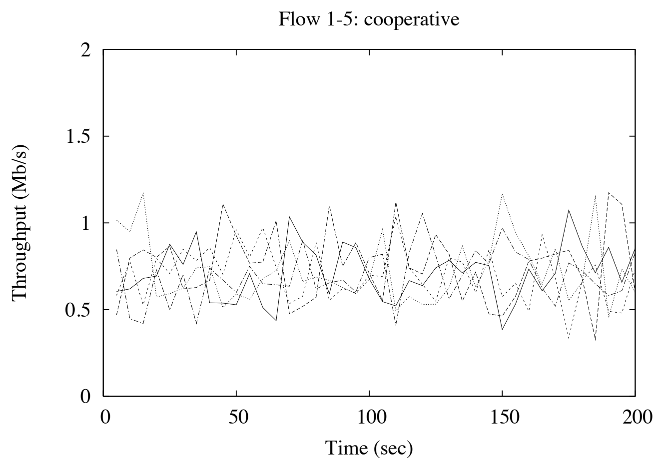
→ referred to as TCP-friendliness

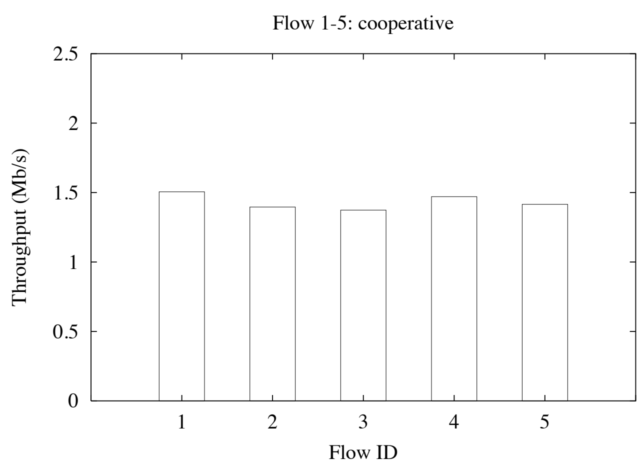
Example:

→ 5 regular (cooperative) TCP flows

→ share 11 Mbps WLAN bottleneck link

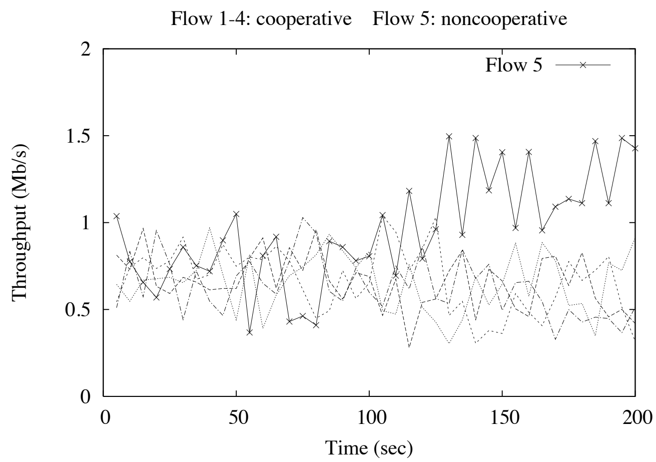
Throughput share of 5 homogenous TCP flows:

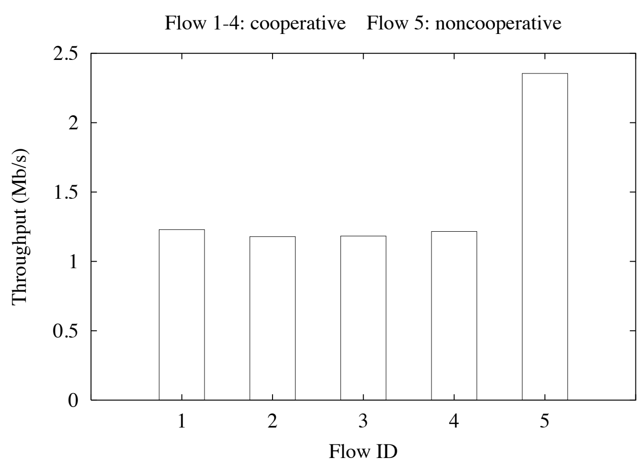




4 regular (cooperative) TCP flows and 1 noncooperative TCP flow:

→ starts behaving selfishly at time 100s





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