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?

⟶ linear increase/exponential decrease

⟶ AIMD

⟶ method B

TCP congestion control components:

(i) Congestion avoidance

⟶ linear increase/exponential decrease

⟶ additive increase/exponential decrease (AIMD)

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

Upon receiving ACK:

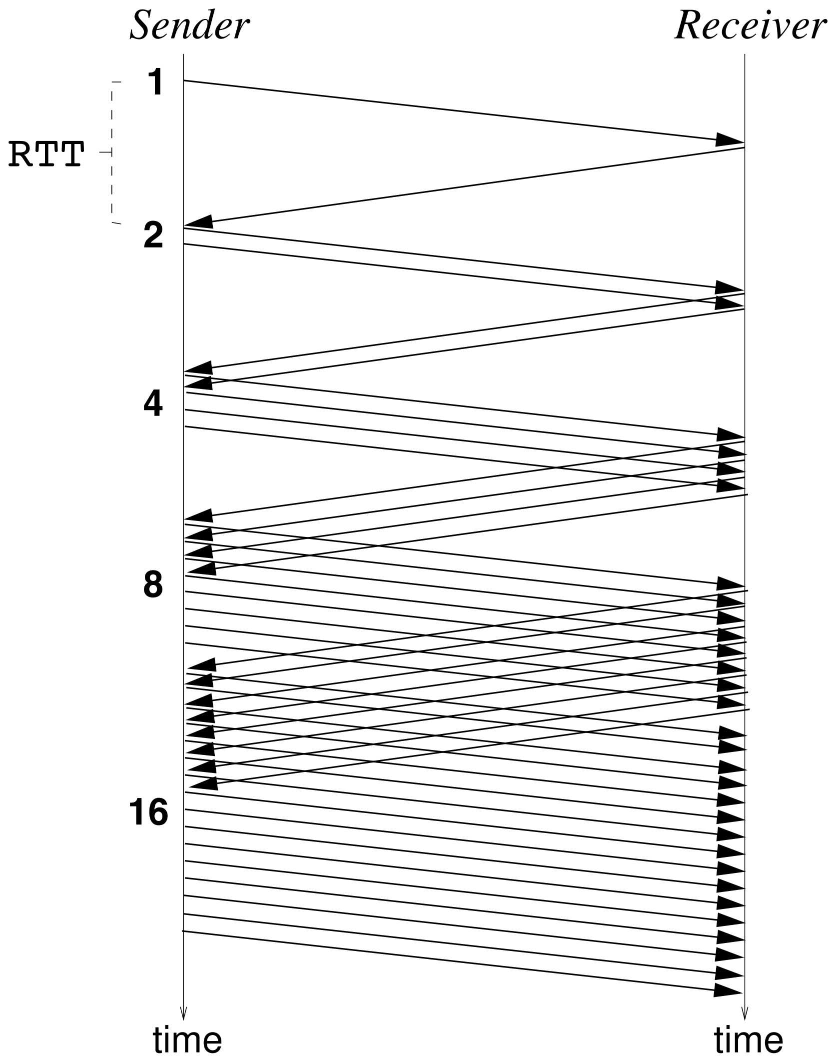
$$\texttt{CongestionWindow} \leftarrow \texttt{CongestionWindow} + 1$$

Upon timeout:

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

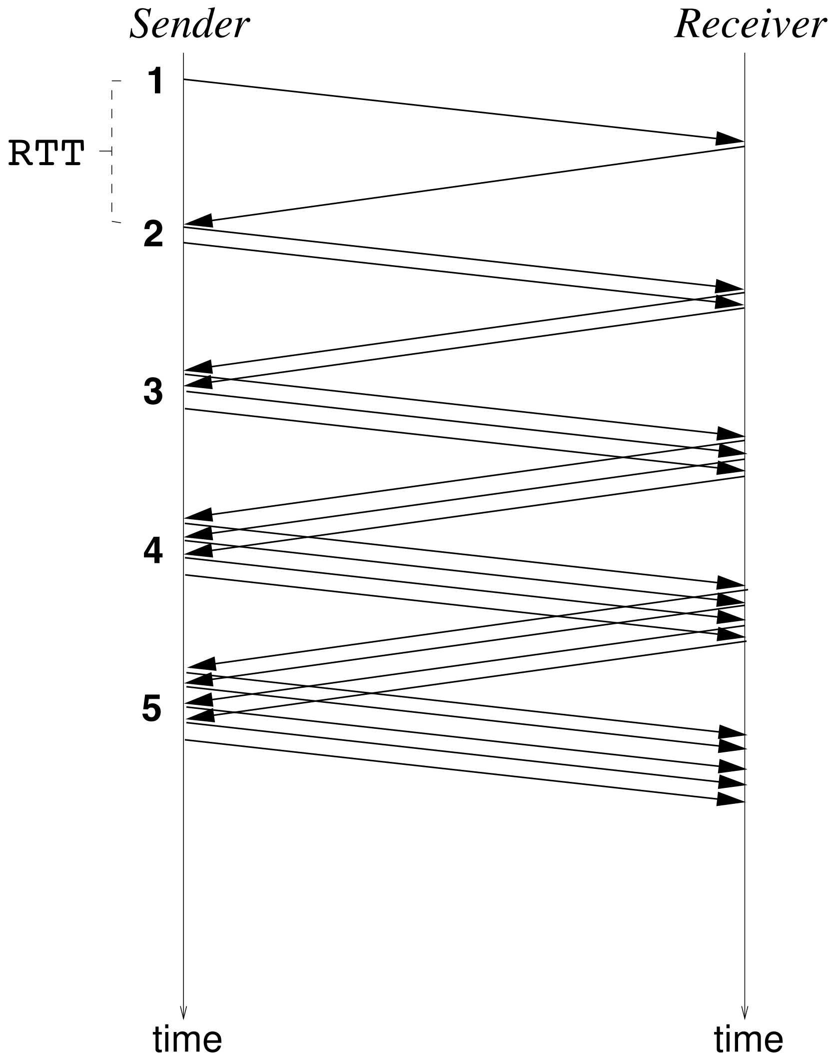
But is it correct…

"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

  → rapidly probe for available bandwidth

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

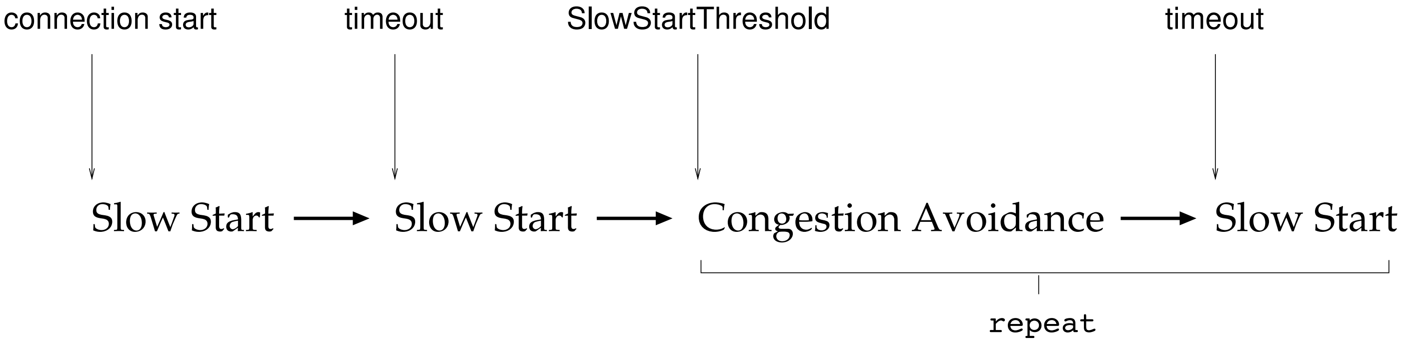
  → rapidly climb to safe level

⟶ "slow" is a misnomer

⟶ exponential increase is super-fast

Basic dynamics:

- ⟶ after connection set-up
- ⟶ before connection tear-down



connection start → Slow Start

timeout → Slow Start

SlowStartThreshold → Congestion Avoidance

timeout → Slow Start

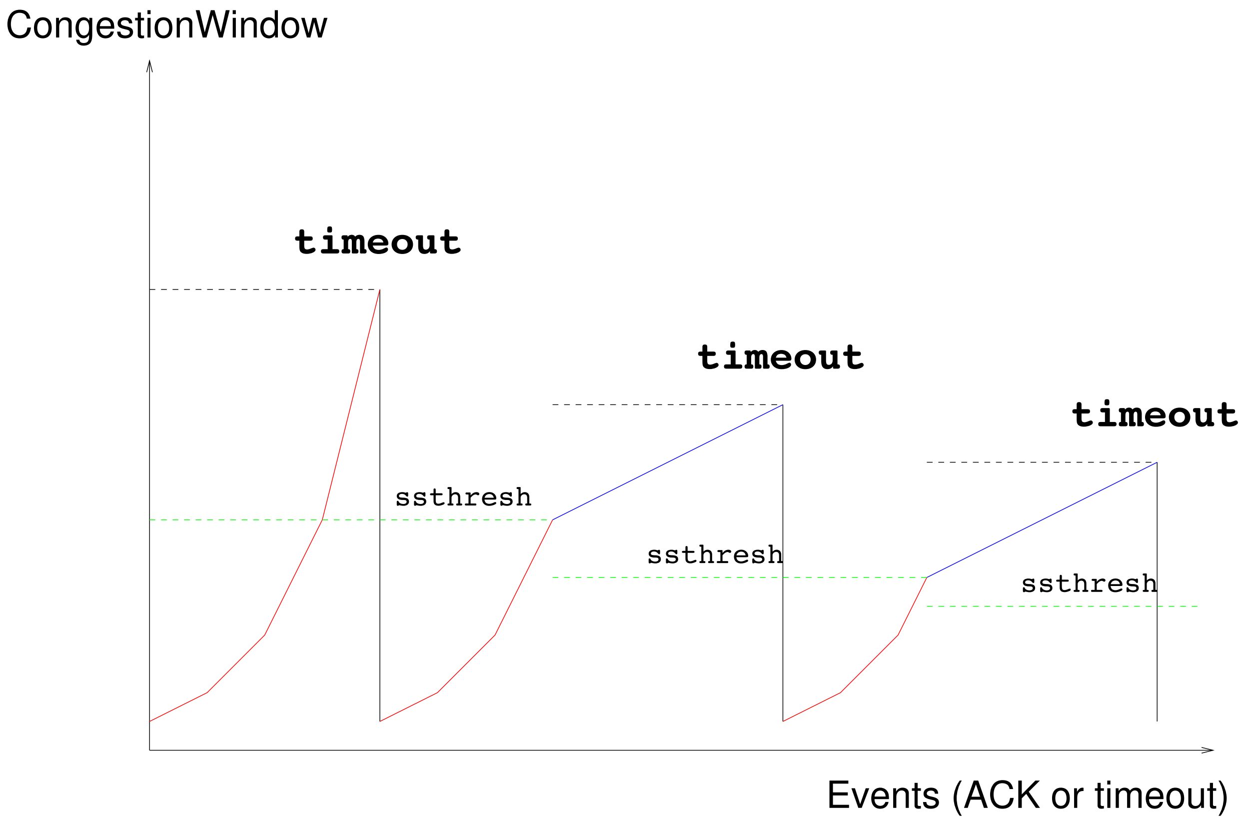
Slow Start ⟶ Slow Start ⟶ Congestion Avoidance ⟶ Slow Start

`repeat`

- ⟶ 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 and commence Congestion Avoidance

  → dup ACKs: likely spurious loss
- Insert "inflationary" phase just before Congestion Avoidance

Additional changes and recent TCP variants.

Window scaling:

- 16-bit window size field limits receiver buffer size to 64 KB.
- Increase window size by scaling factor.
- During SYN handshake, exchange scaling factor using option field.
- If scaling factor is c, multiply window size by 2^{16+c}

  → shift operation

  → c limited to 14

BIC-TCP, TCP CUBIC: loss-based

- Instead of linear increase in Congestion Avoidance, use binary search (BIC)

  → concave shape: fast then slow when nearing window size of congestion event ($W_{\max}$)

  → convex shape: after $W_{\max}$ switch to probing mode

  → TCP CUBIC uses cubic function directly

  → Linux

TCP Vegas, Compound TCP: delay-based, hybrid

- Estimate queueing delay from RTT

  → use minimum as reference point

- If RTT increases assume queueing at bottleneck link(s)

  → slow down linearly

  → closer to method D

  → susceptible to congestion collapse

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

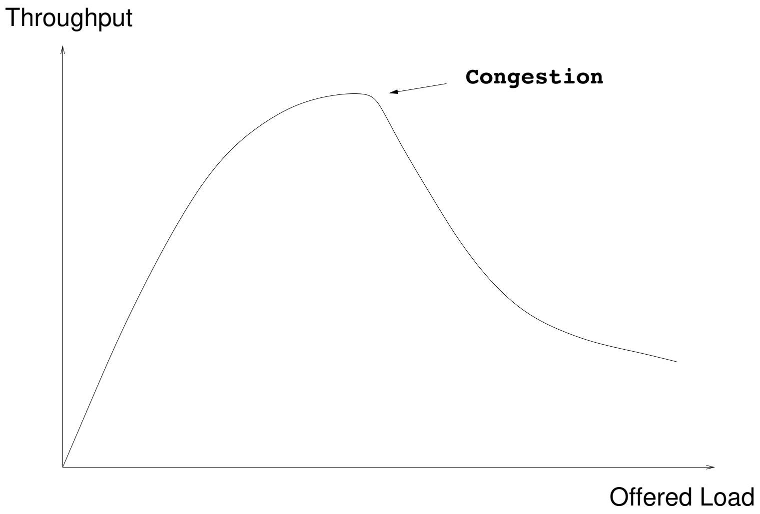
  → worst-case: congestion collapse

Congestion control and selfishness:

⟶ to be or not to be selfish . . .

⟶ John von Neumann, John Nash, . . .

Ex.: “tragedy of commons,” Garrett Hardin, ’68



- if everyone acts selfishly, no one wins

  → in fact, everyone loses
- can this be prevented?

Ex.: Prisoner's Dilemma game

$\longrightarrow$ formalized by Tucker in 1950

$\longrightarrow$ "cold war"

- both cooperate (i.e., stay mum): 1 year each
- both selfish (i.e., rat on the other): 5 years each
- one cooperative/one selfish: 9 vs. 0 years

		Bob	
		C	N
Alice	C	$1, 1$	$9, 0$
	N	$0, 9$	$5, 5$

$\longrightarrow$ payoff matrix

$\longrightarrow$ what would "rational" prisoners do?

When cast as congestion control game:

		Bob	
		C	N
Alice	C	5, 5	0, 9
	N	9, 0	1, 1

Alice and Bob share network bandwidth

→ (a, b): throughput (Mbps) achieved by Alice/Bob

→ large is desirable

Upon congestion: back off or escalate?

→ equivalent to Prisoner's dilemma

Rational: in the sense of seeking selfish gain

→ both choose strategy “N”

→ called Nash equilibrium

→ steady-state or stable fixed-point

Reason:

→ whatever choice the other player makes, “N” yields better payoff over “C”

→ i.e., strategy “N” dominates strategy “C”

In some systems, selfish behavior results in system optimal outcome

→ theoretical foundation of Adam Smith’s “invisible hand”

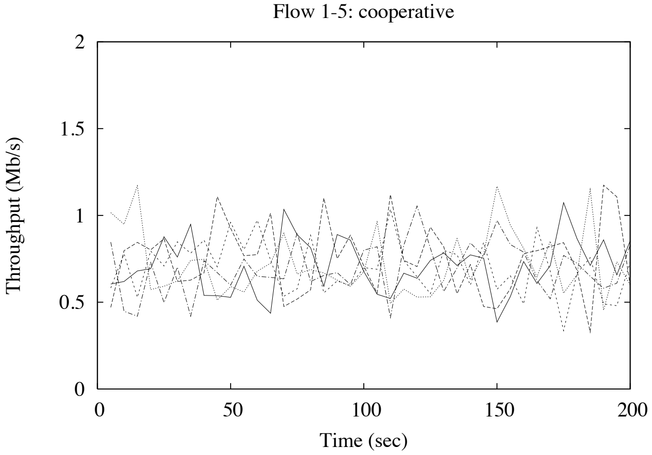
→ in general, not the case

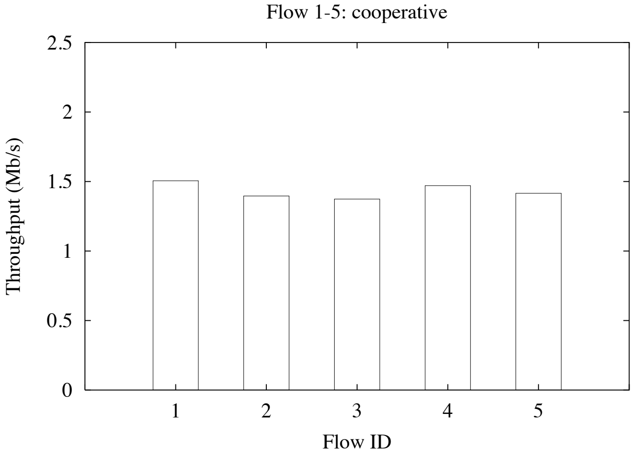
→ cooperation is better but can it be enforced?

Impact in networks:

→ 5 regular (cooperative) TCP flows

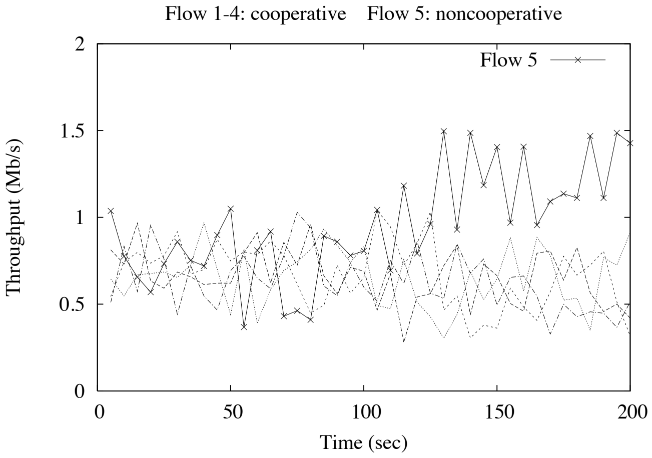
→ share 11 Mbps WLAN bottleneck link

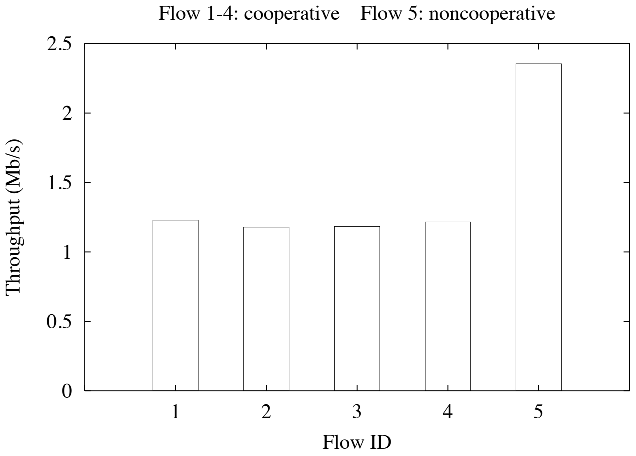




4 regular (cooperative) TCP flows and 1 noncooperative TCP flow:

→ starts behaving selfishly at time 100s





Potential danger for:

→ unfairness

→ overall system performance

Is it being exploited in today's Internet?

→ no one knows

→ technical implementation issues

→ e.g., interoperability with legacy protocols

→ e.g., shooting oneself in the foot