Advanced Computer Networks

Congestion control in TCP

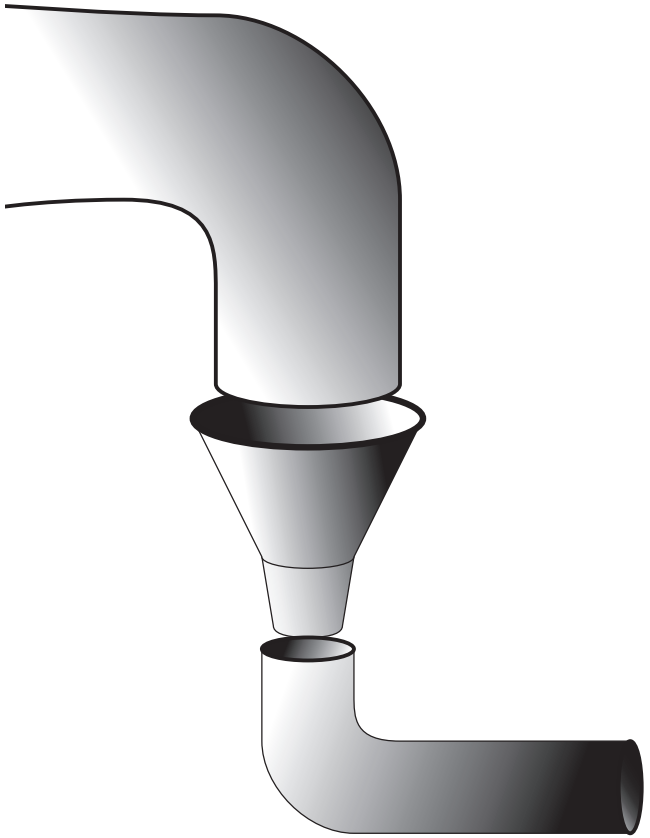
Prof. Andrzej Duda

duda@imag.fr

http://duda.imag.fr

Contents

- AIMD - Additive increase, multiplicative decrease
- Principles of CC in TCP
- TCP Tahoe congestion control states
  - Slow Start
  - Congestion Avoidance
- TCP Reno congestion control states
  - Slow Start
  - Congestion Avoidance
  - Fast Recovery
- TCP RENO fairness
- TCP Cubic
- TCP BBR
- ECN



End-to-end congestion control

- End-to-end congestion control
  - binary feedback from the network: congestion or not
  - rate adaptation mechanism: decrease or increase
- Modeling
  - I sources, rate $x_i(t)$, $i = 1, ..., I$
  - link capacity: C
  - discrete time, feedback cycle = one time unit
  - during one time cycle, the source rates are constant, and the network generates a binary feedback signal $y(t) \in \{0, 1\}$
  - sources: increase the rate if $y(t) = 0$ and decrease if $y(t) = 1$
  - feedback

$$y(t) = [if\ (\sum_{i=1}^{I} x_i(t) \le c)\ then\ 0\ else\ 1]$$

Linear adaptation algorithm

- Find constants u_0, u_1, v_0, v_1, such that

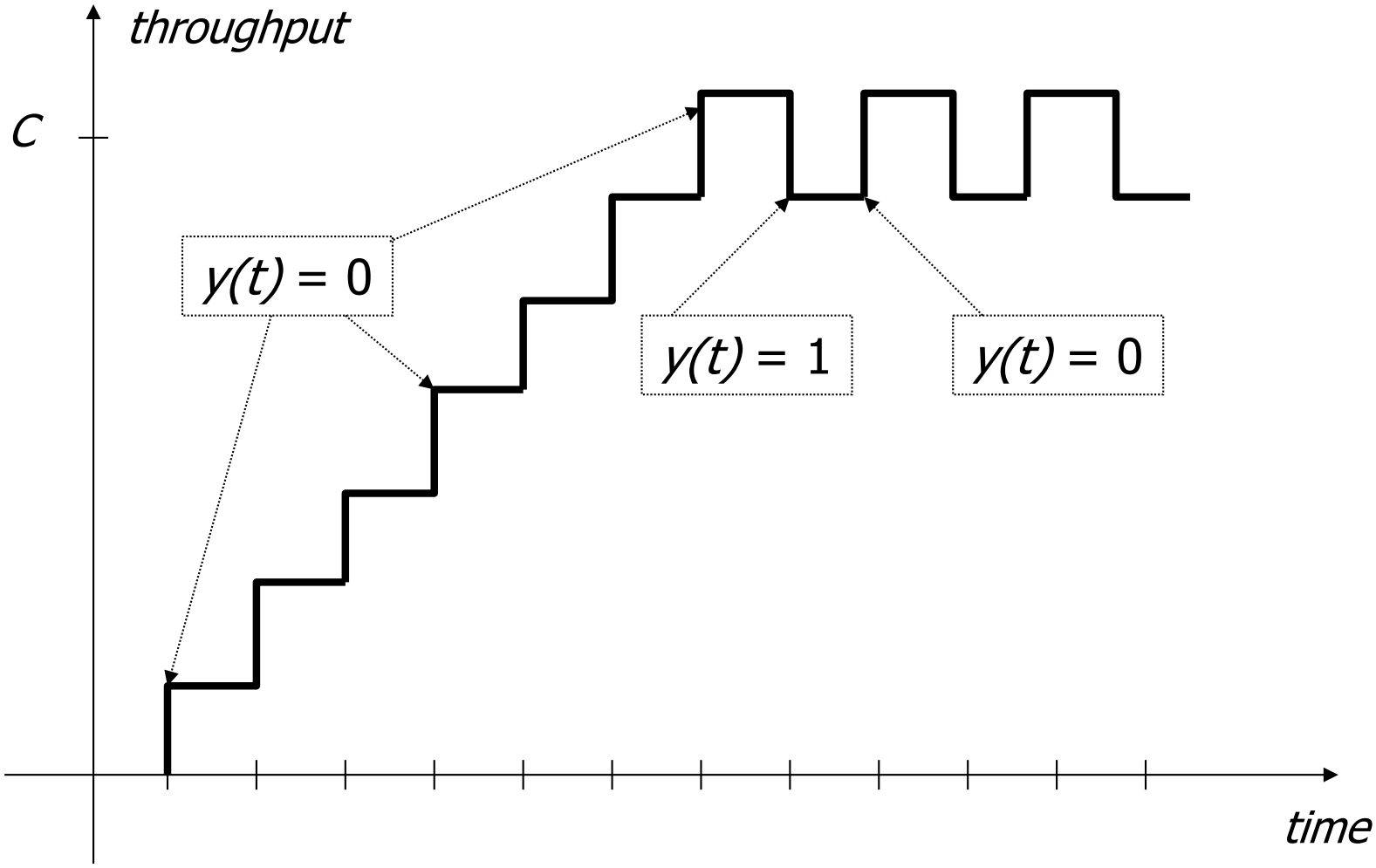
$$x_i(t+1) = u_{y(t)}\, x_i(t) + v_{y(t)}$$

  - we want to converge towards a fair allocation
  - one single bottleneck, so all fairness criteria are equivalent
  - we should have $x_i = C/I$
  - the total throughput

$$f(t) = \sum_{i=1}^{I} x_i(t)$$

    should oscillate around C : it should remain below C until it exceeds it once, then return below C

Linear adaptation algorithm



Necessary conditions

$$f(t+1) = u_{y(t)} f(t) + v_{y(t)}$$

- we must have

  $u_0 f + v_0 > f$, increase rate if feedback 0

  $u_1 f + v_1 < f$, decrease rate if feedback 1

- this gives the following conditions

  $u_1 < 1$ and $v_1 \leq 0$ (A)

  or

  $u_1 = 1$ and $v_1 < 0$ (B)

  and

  $u_0 > 1$ and $v_0 \geq 0$ (C)

  or

  $u_0 = 1$ and $v_0 > 0$ (D)

Ensure fairness

$u_1 < 1$ and $v_1 \leq 0$ (A)

or

$u_1 = 1$ and $v_1 < 0$ (B)

and

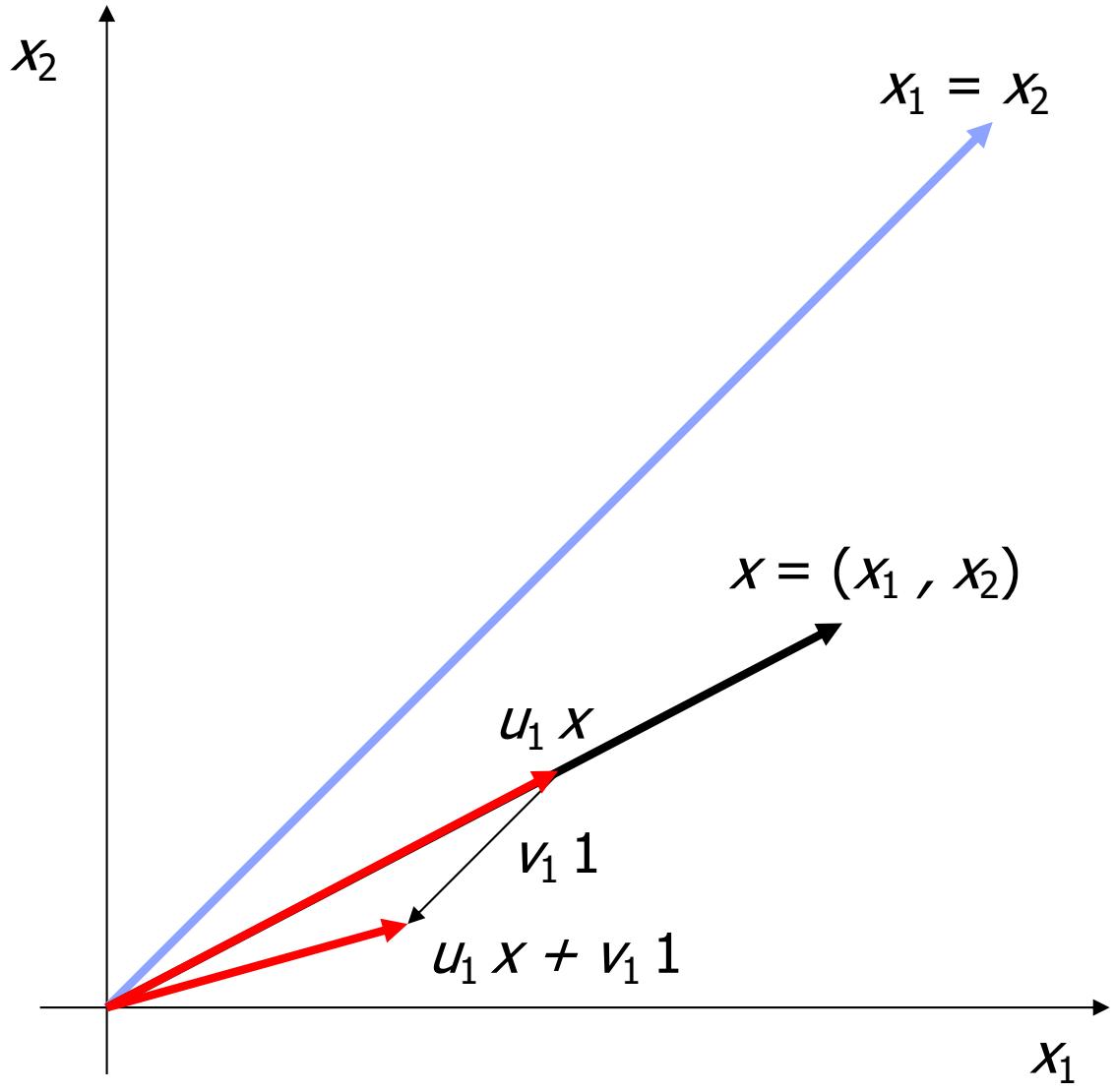
$u_0 > 1$ and $v_0 \geq 0$ (C)

or

$u_0 = 1$ and $v_0 > 0$ (D)

how to decrease rate?

$v_1 = 0$, $u_1 < 1$



Ensure fairness

$u_1 < 1$ and $v_1 \leq 0$ (A)

or

$u_1 = 1$ and $v_1 < 0$ (B)

and

$u_0 > 1$ and $v_0 \geq 0$ (C)

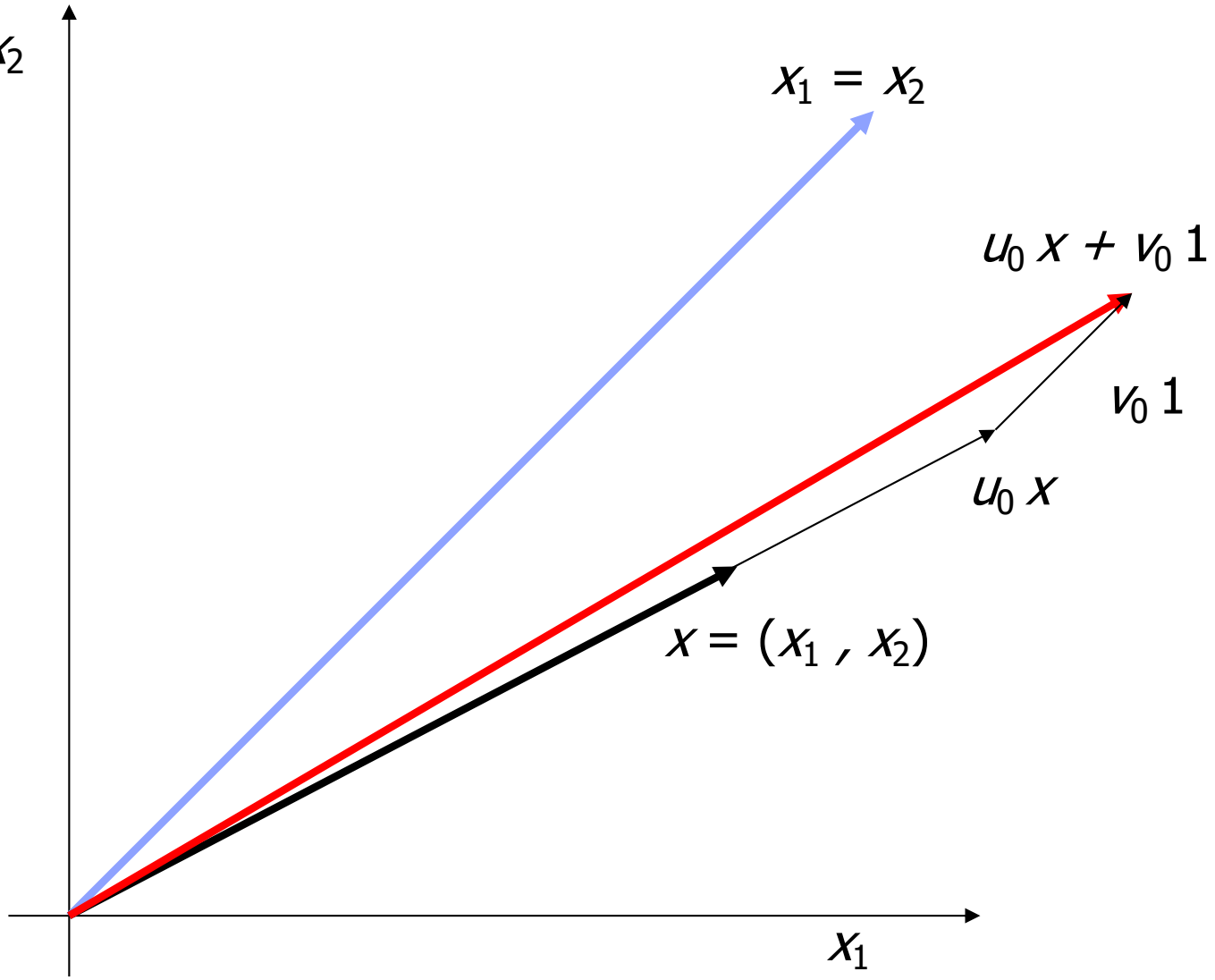
or

$u_0 = 1$ and $v_0 > 0$ (D)

how to increase rate?

$v_0 > 0$, $u_0 > 1$ or

$v_0 > 0$, $u_0 = 1$



Ensure fairness

$u_1 < 1$ and $v_1 \leq 0$ (A)

or

$u_1 = 1$ and $v_1 < 0$ (B)

and

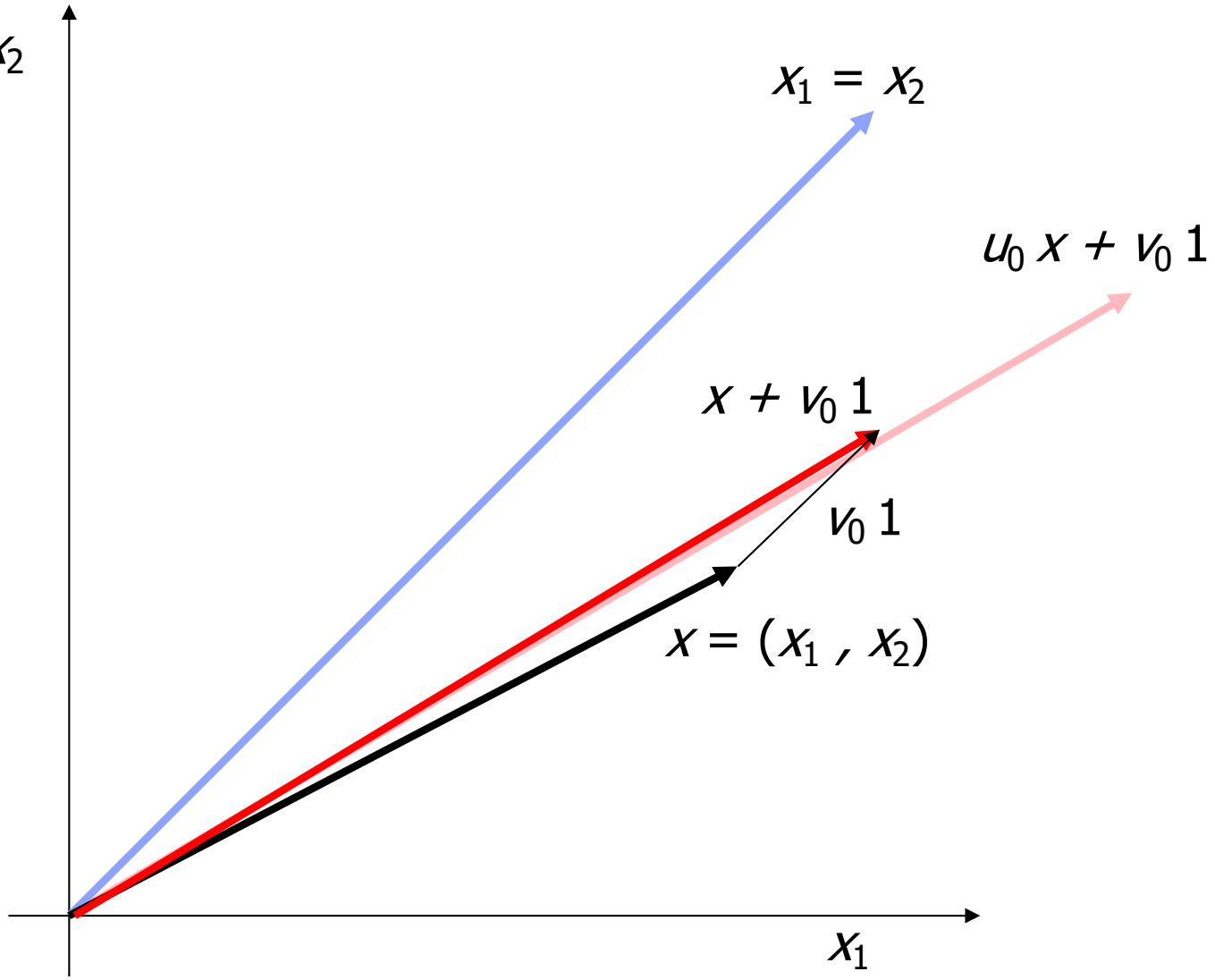
$u_0 > 1$ and $v_0 \geq 0$ (C)

or

$u_0 = 1$ and $v_0 > 0$ (D)

how to increase rate?

$v_0 > 0$, $u_0 = 1$



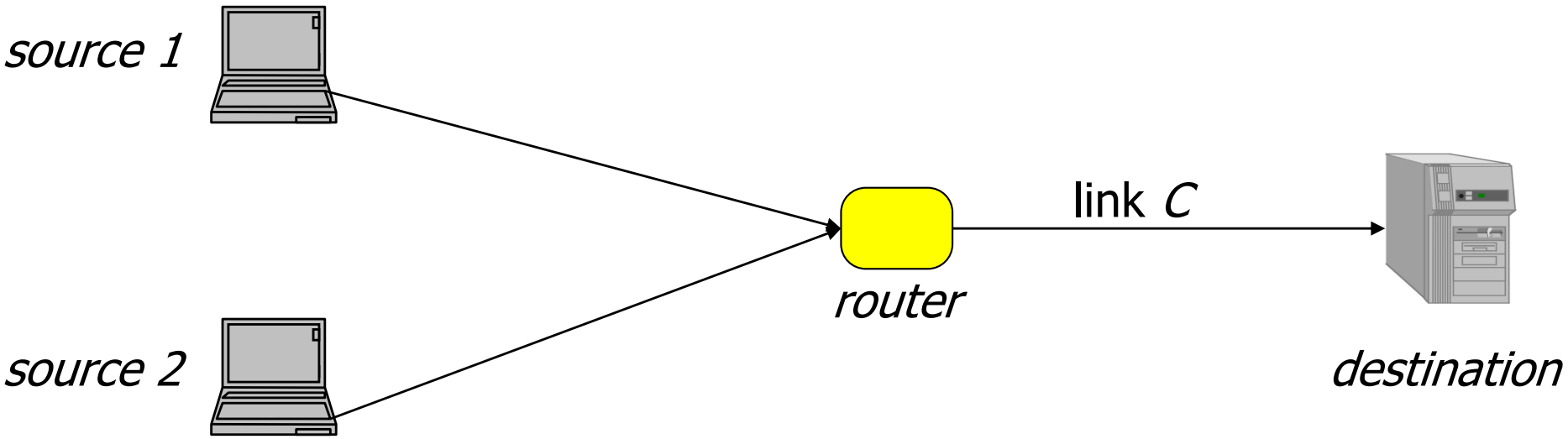
Ensure fairness

- When we apply a multiplicative increase or decrease, the unfairness is unchanged
- An additive increase decreases the unfairness, whereas an additive decrease increases the unfairness
- To obtain that unfairness decreases or remains the same, and such that in the long term it decreases
  - $v_1 = 0$ decrease must be **multiplicative**
  - $u_0 = 1$ increase must be **additive**

Result

- Fact
  - In order to satisfy efficiency and convergence to fairness, we must have a multiplicative decrease (namely, $u_1 < 1$ and $v_1 = 0$ and a non-zero additive component in the increase (namely, $u_0 \geq 1$ and $v_0 > 0$).
  - If we want to favour a rapid convergence towards fairness, then the increase should be additive only (namely, $u_0 = 1$ and $v_0 > 0$).
- **AIMD - Additive increase, Multiplicative decrease**

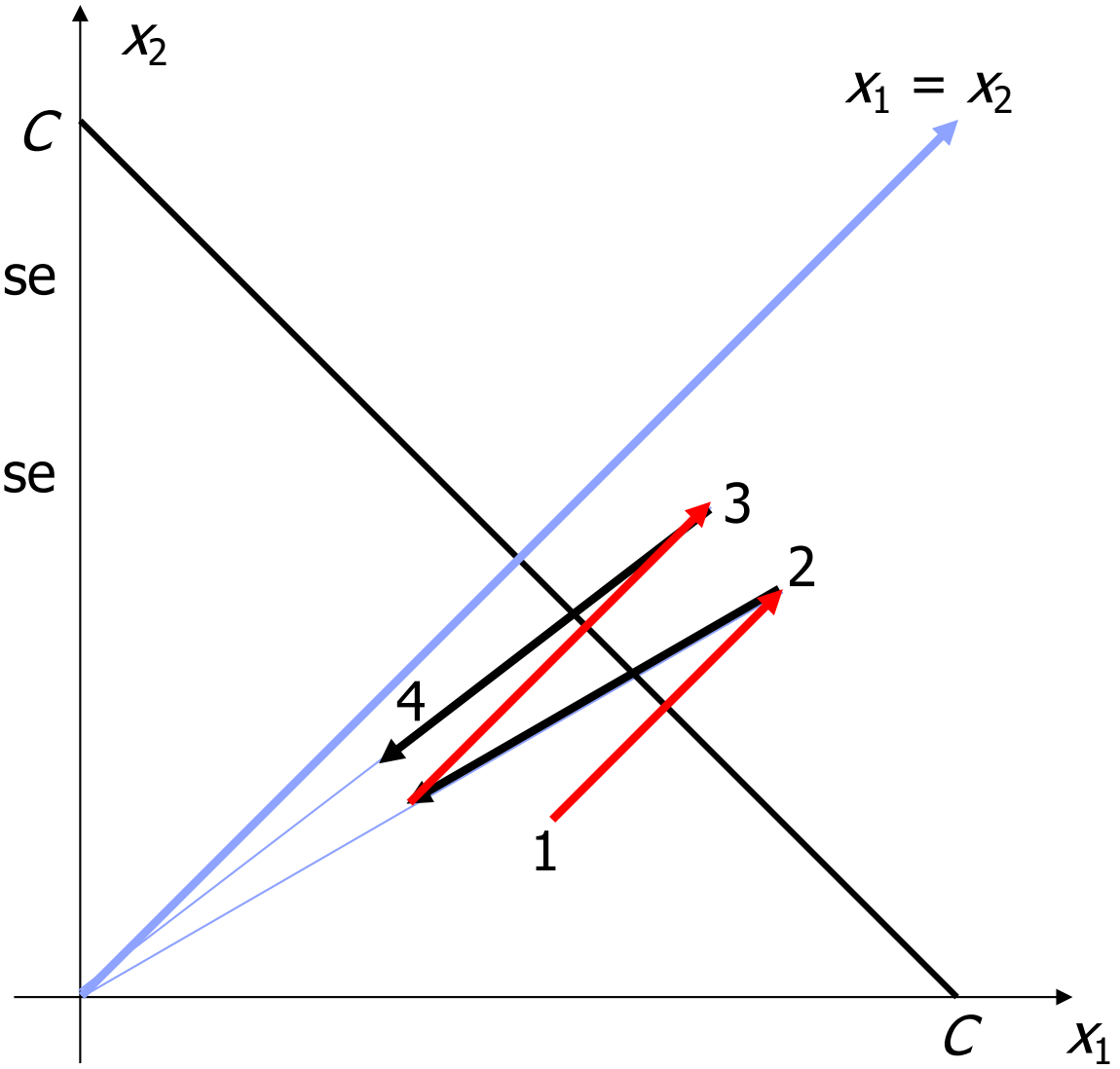
Why AI-MD works?



- Simple scenario with two sources sharing a bottleneck link of capacity *C*

Throughput of sources

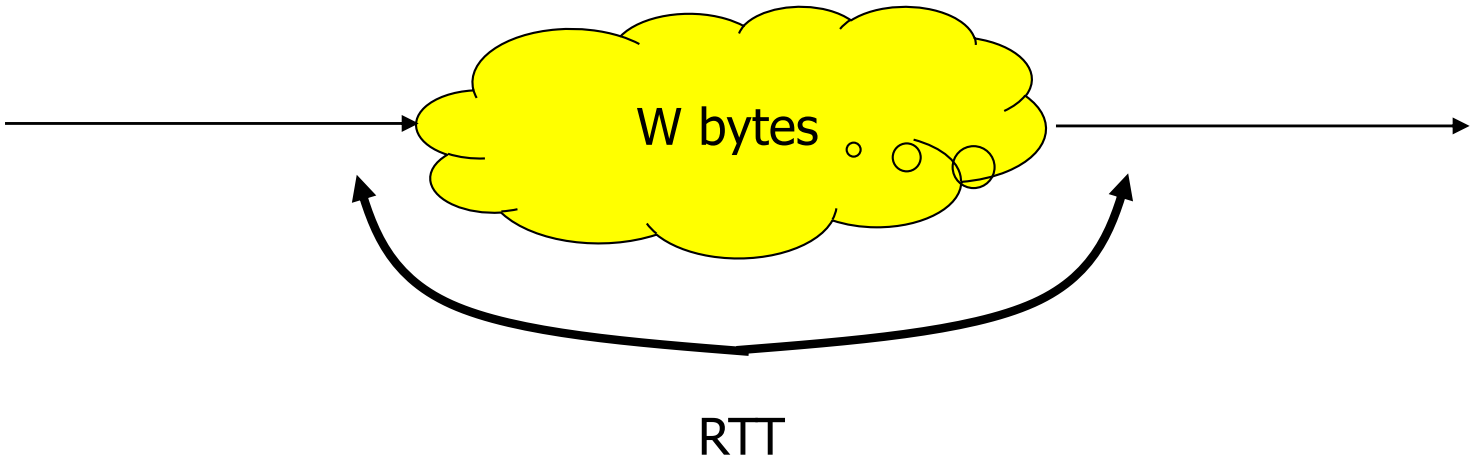
1. Additive increase
2. Multiplicative decrease
3. Additive increase
4. Multiplicative decrease



TCP and Congestion Control

- TCP is used to avoid congestion in the Internet
  - a TCP source adjusts its sending window to the congestion state of the network
  - this avoids congestion collapse and ensures some fairness
- TCP sources interpret losses as a negative feedback
  - used to reduce the sending rate
- Window-based control
- UDP sources are a problem for the Internet
  - use for long lived sessions (ex: RealAudio) is a threat: congestion collapse
  - UDP sources should imitate TCP : “TCP friendly”

Sending window

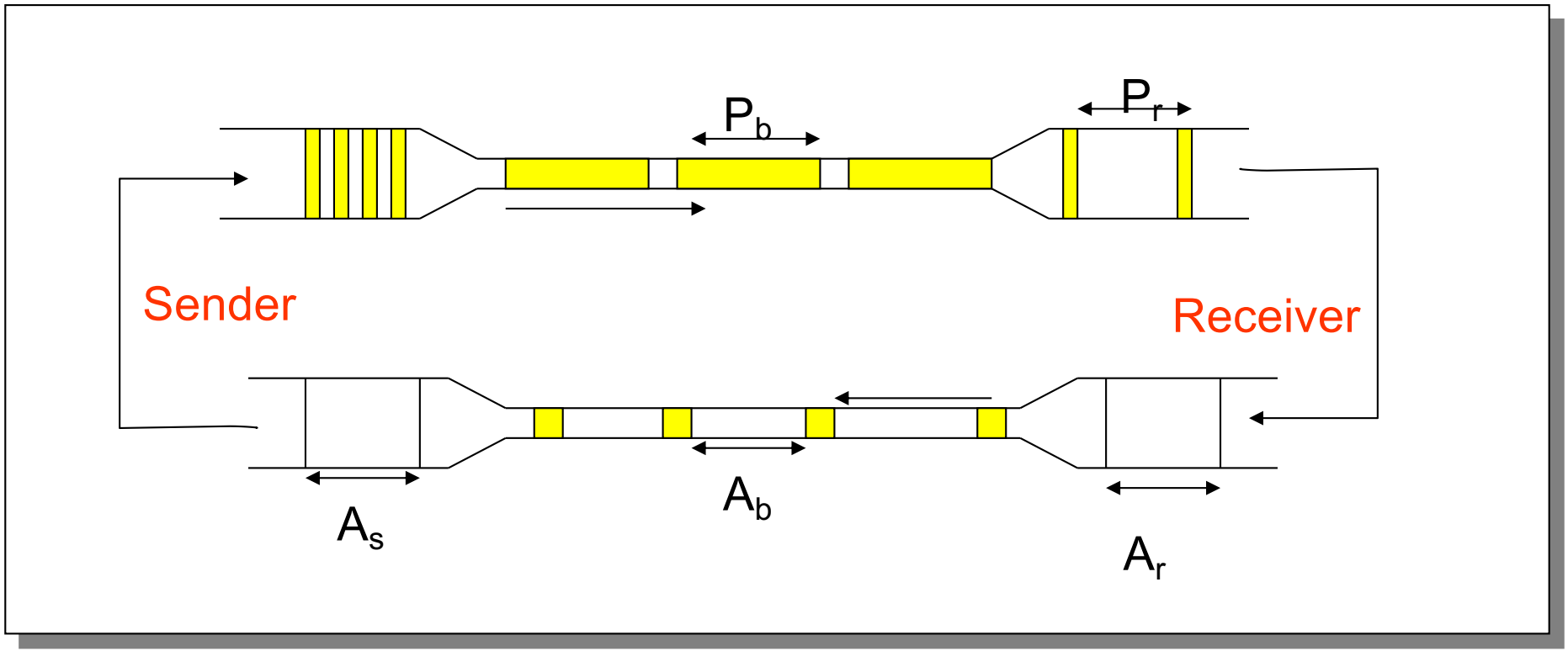


- W - the number of non ACKed bytes
  - throughput = W/RTT (Little's formulae)
- If congestion
  - RTT increases, automatic reduction of the source rate
  - additional control: decrease W

Sending window

- Sending window - number of non ACKed bytes
  - W = min (**`cwnd`**, **`OfferedWindow`**)
  - **`cwnd`**
    - congestion window - maintained by TCP source
  - **`OfferedWindow`**
    - announced by destination in TCP header
    - flow control
    - reflects free buffer space
- Same mechanism used for flow control and for congestion control

Self-clocking or ACK Clock



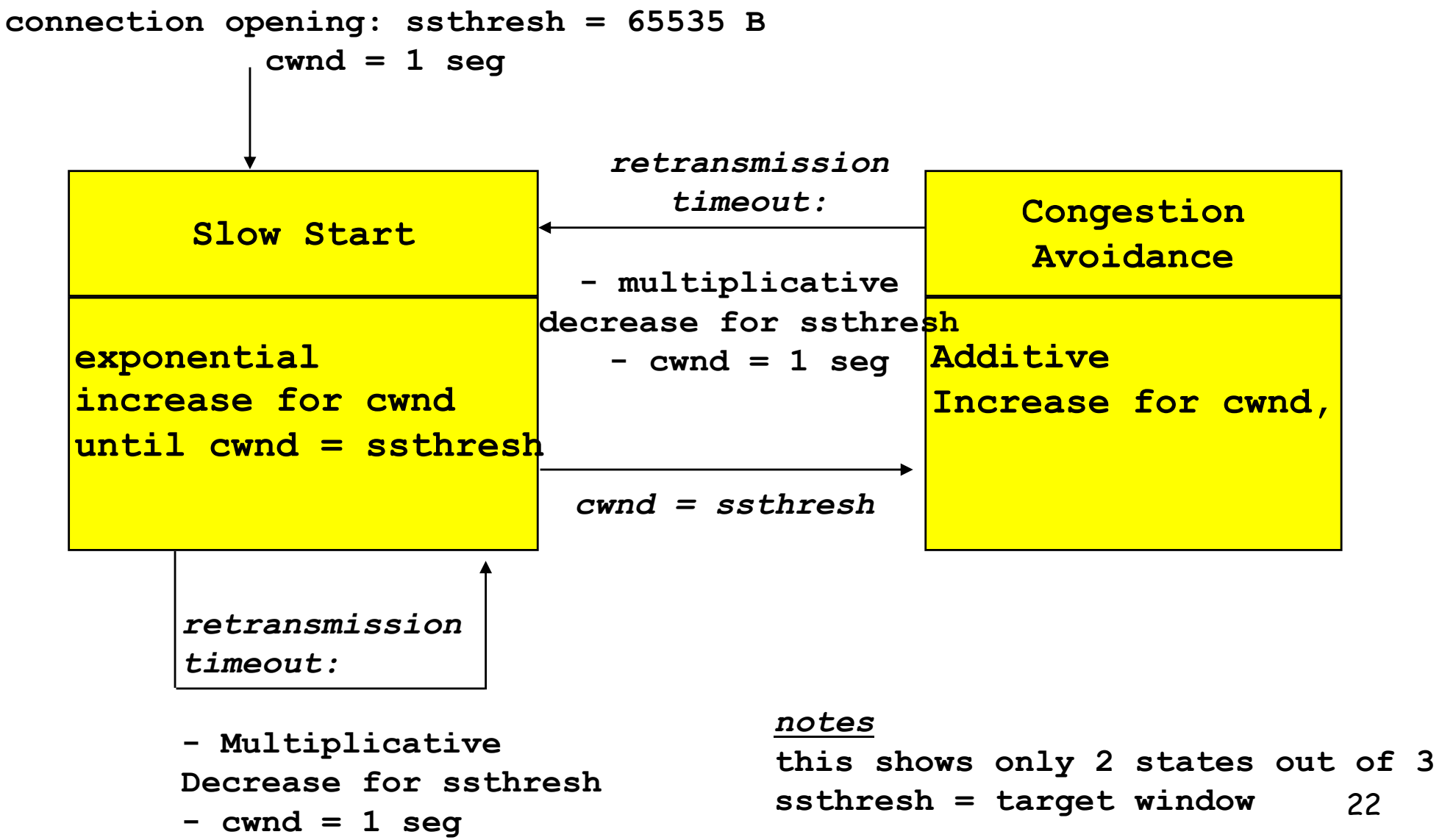
- Self-clocking systems tend to be very stable under a wide range of bandwidths and delays.
- The principal issue with self-clocking systems is getting them started.

Congestion control states

- TCP connection may be in three states with respect to congestion
  - **Slow Start** (Démarrage Lent) after loss detected by retransmission timer
  - **Fast Recovery** (Récupération Rapide) after loss detected by Fast Retransmit (three duplicated ACKs)
  - **Congestion Avoidance** (Évitement de Congestion) otherwise
- Terminology
  - *ssthresh* – target window
  - *cwnd – congestion window*
  - *flightSize* - the amount of data that has been sent but not yet acknowledged, roughly *cwnd*

TCP Tahoe

Slow Start and Congestion Avoidance



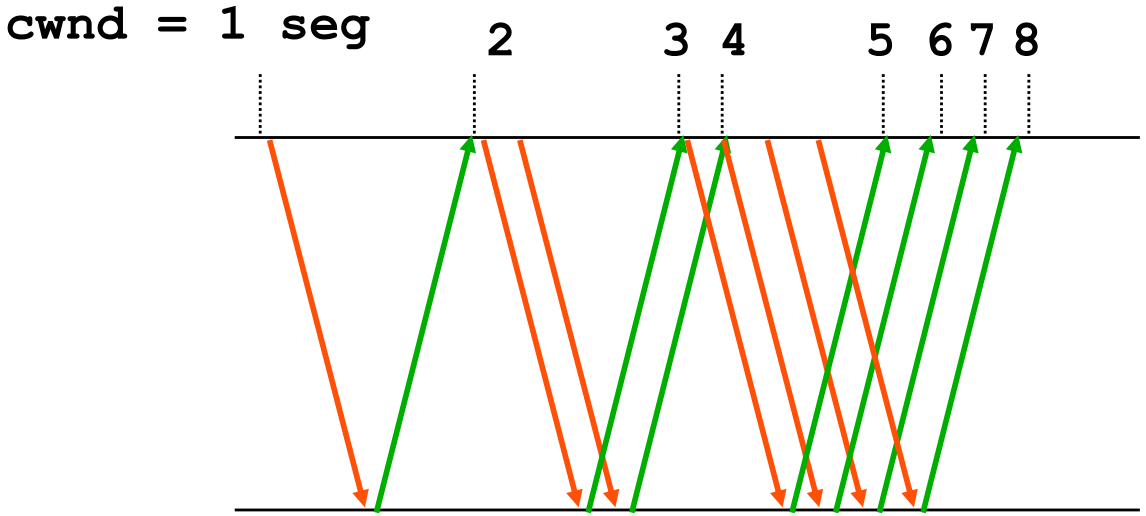
Slow Start

```
/ * exponential increase for cwnd */

  non dupl. ack received during slow start ->
            cwnd = cwnd + MSS (in bytes)
  if cwnd = ssthresh then transition to
  congestion avoidance
```

- Window increases rapidly up to the value of **`ssthresh`**
  Not so slow, rather exponential

Slow Start

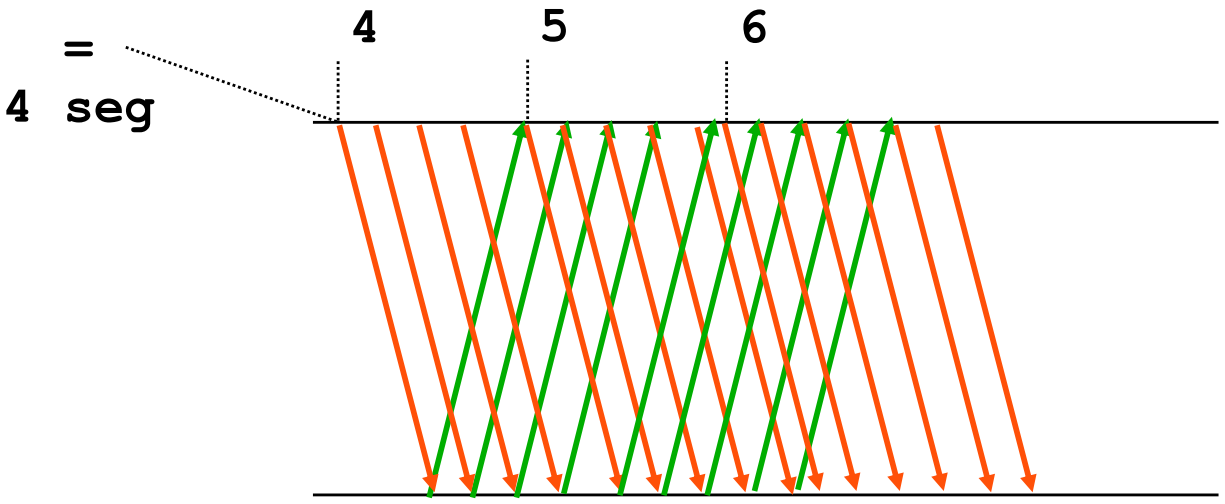


- purpose of this phase: avoid bursts of data at the beggining or after a retransmission timeout

Increase/decrease

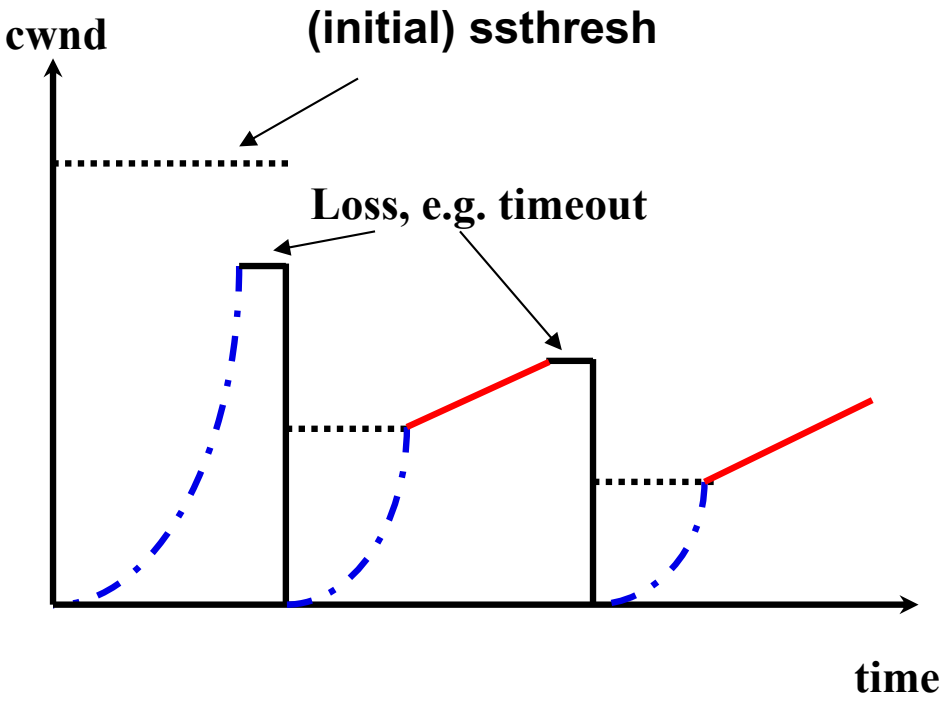
- Multiplicative decrease
  - **`ssthresh = 0.5 cwnd`**
  - **`ssthresh = max (ssthresh, 2 MSS)`**
  - **`cwnd = 1 MSS`**
- Additive increase
  - for each ACK
    - **`cwnd = cwnd + MSS×MSS / cwnd`**
    - **`cwnd = min (cwnd, max-size) (64KB)`**
  - **`cwnd`** is in bytes, counting in segments, this means that
    - we receive (**`cwnd/MSS`**) ACKs per RTT
    - for each ACK: **`cwnd/MSS`** ← 1/W
    - for a full window: W ← W + 1 MSS

`cwnd` Additive Increase



- during one round trip + interval between packets: increase by 1 MSS (linear increase)

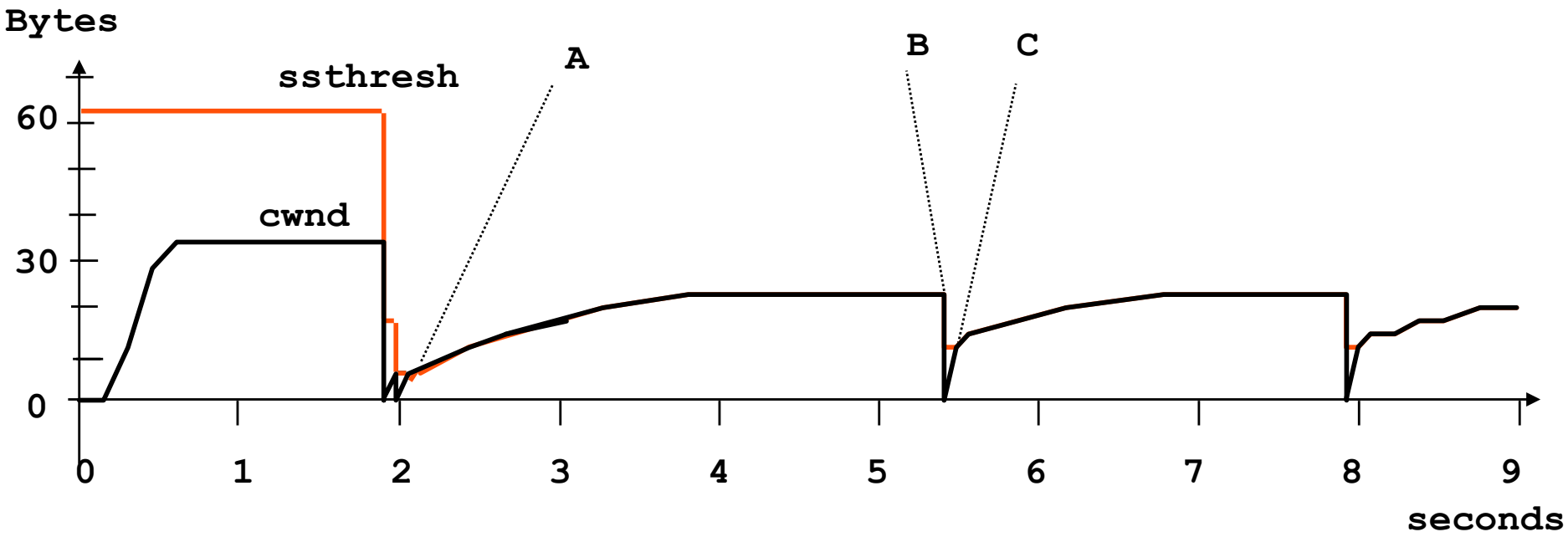
Example



slow start – in bleu

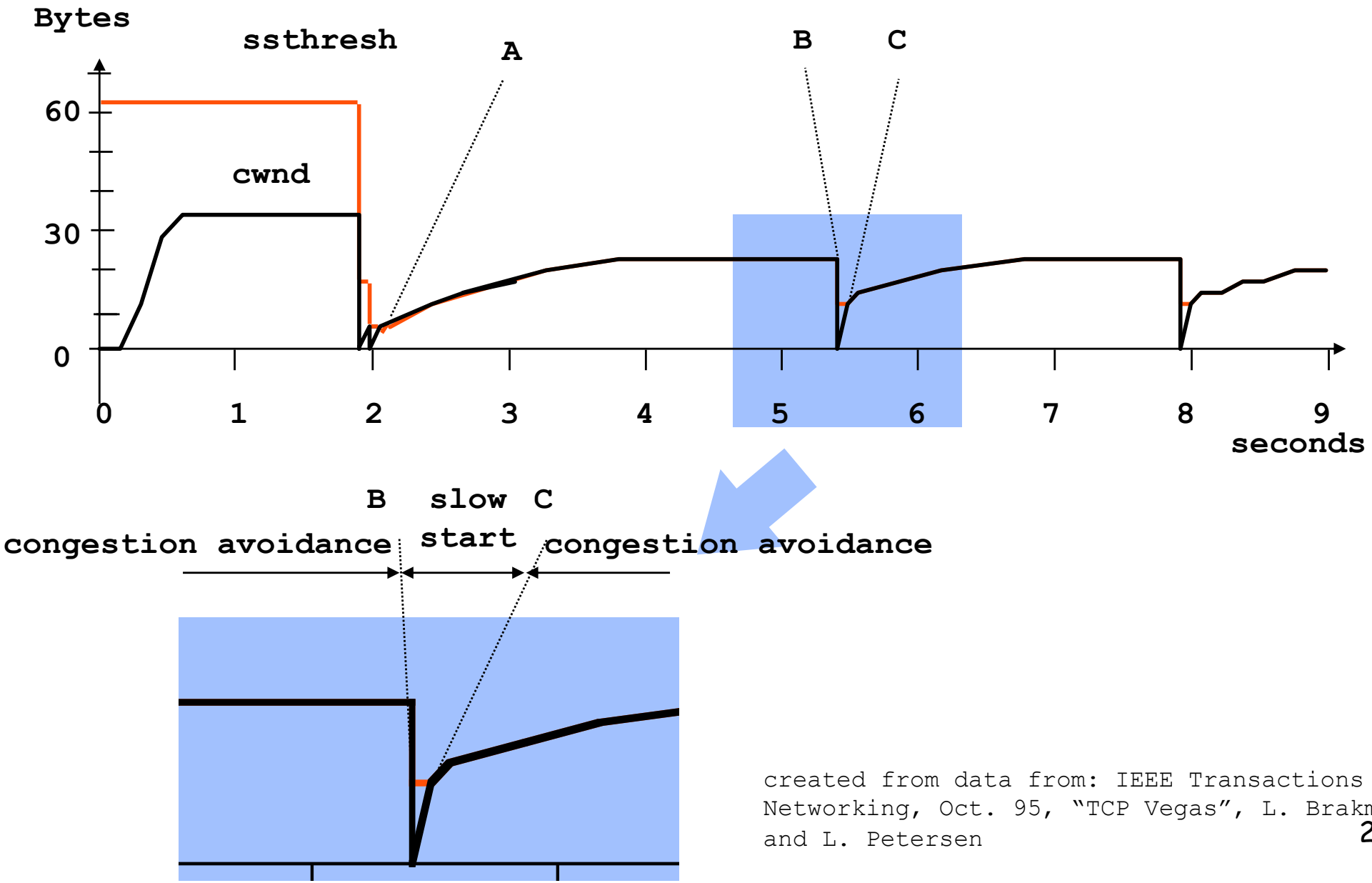
congestion avoidance – in red

Example



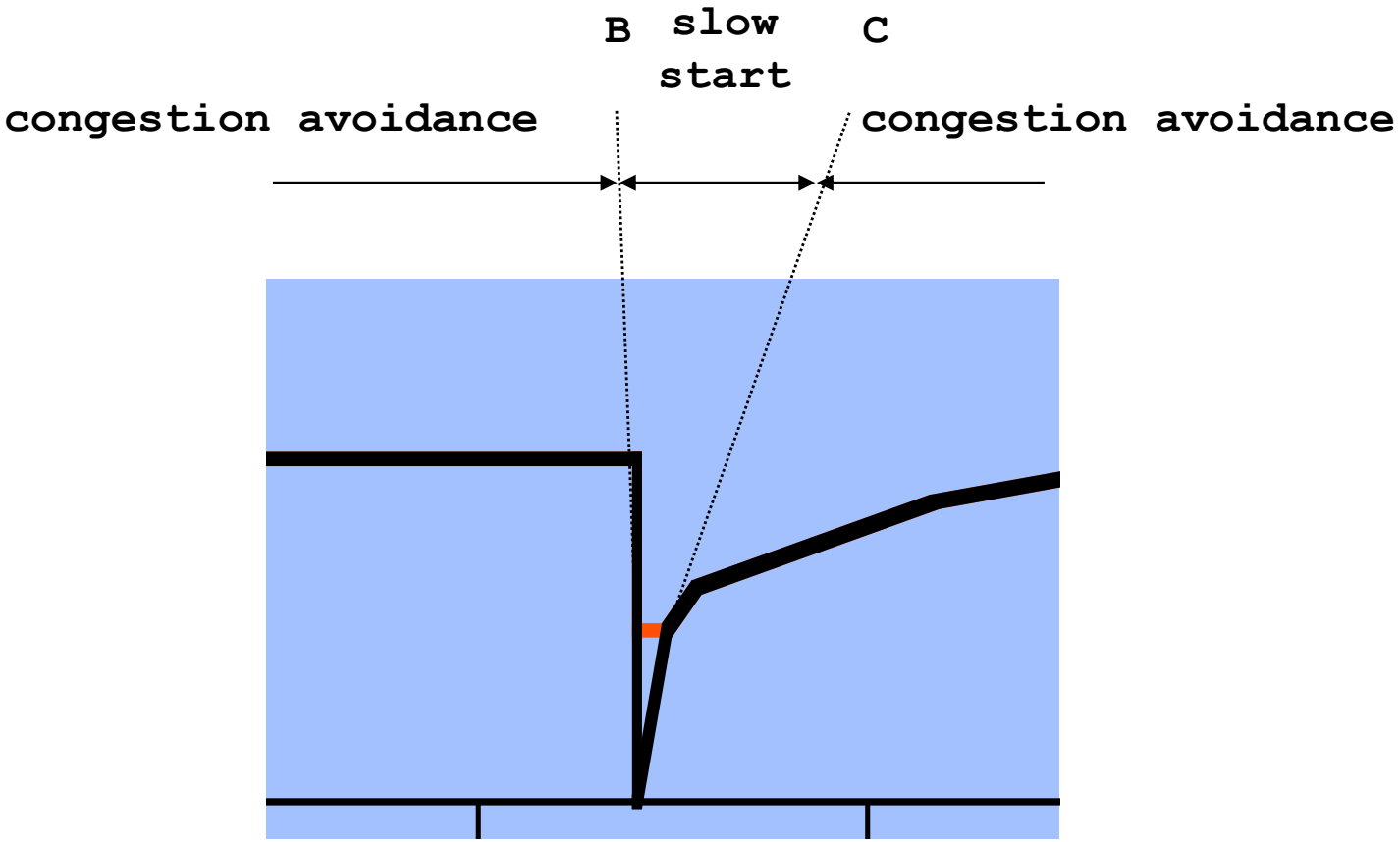
created from data from: IEEE Transactions on Networking, Oct. 95, "TCP Vegas", L. Brakmo and L. Petersen

Example



created from data from: IEEE Transactions on Networking, Oct. 95, "TCP Vegas", L. Brakmo and L. Petersen

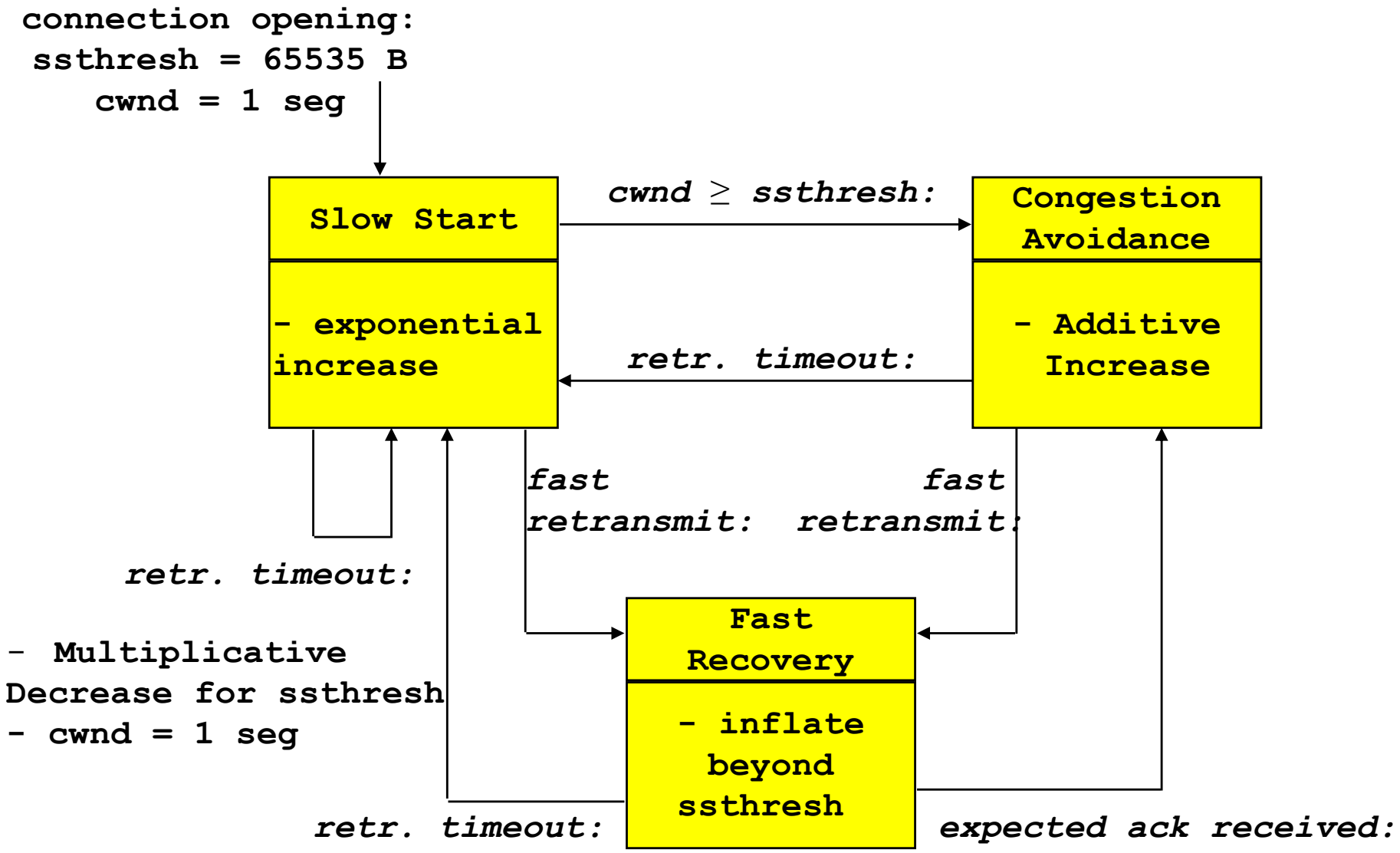
Slow Start and Congestion Avoidance



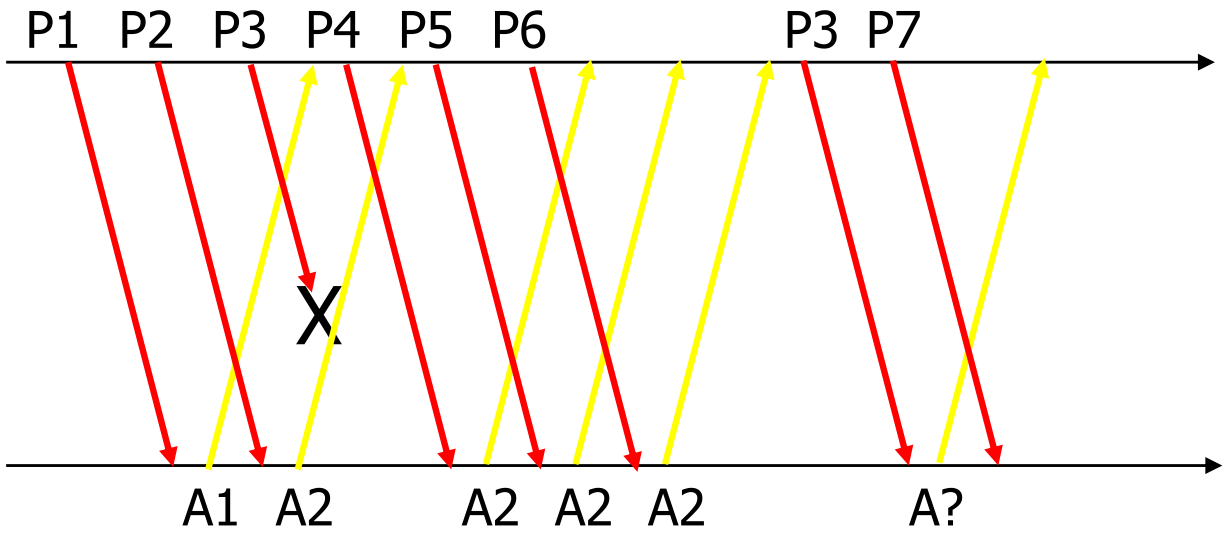
created from data from: IEEE Transactions on Networking, Oct. 95, "TCP Vegas", L. Brakmo and L. Petersen

TCP Reno

Congestion Control States



Fast Retransmit



- Fast Retransmit
  - retransmit timer can be large
  - optimize retransmissions similarly to Selective Retransmit
  - if sender receives 3 duplicated ACKs, retransmit missing segment

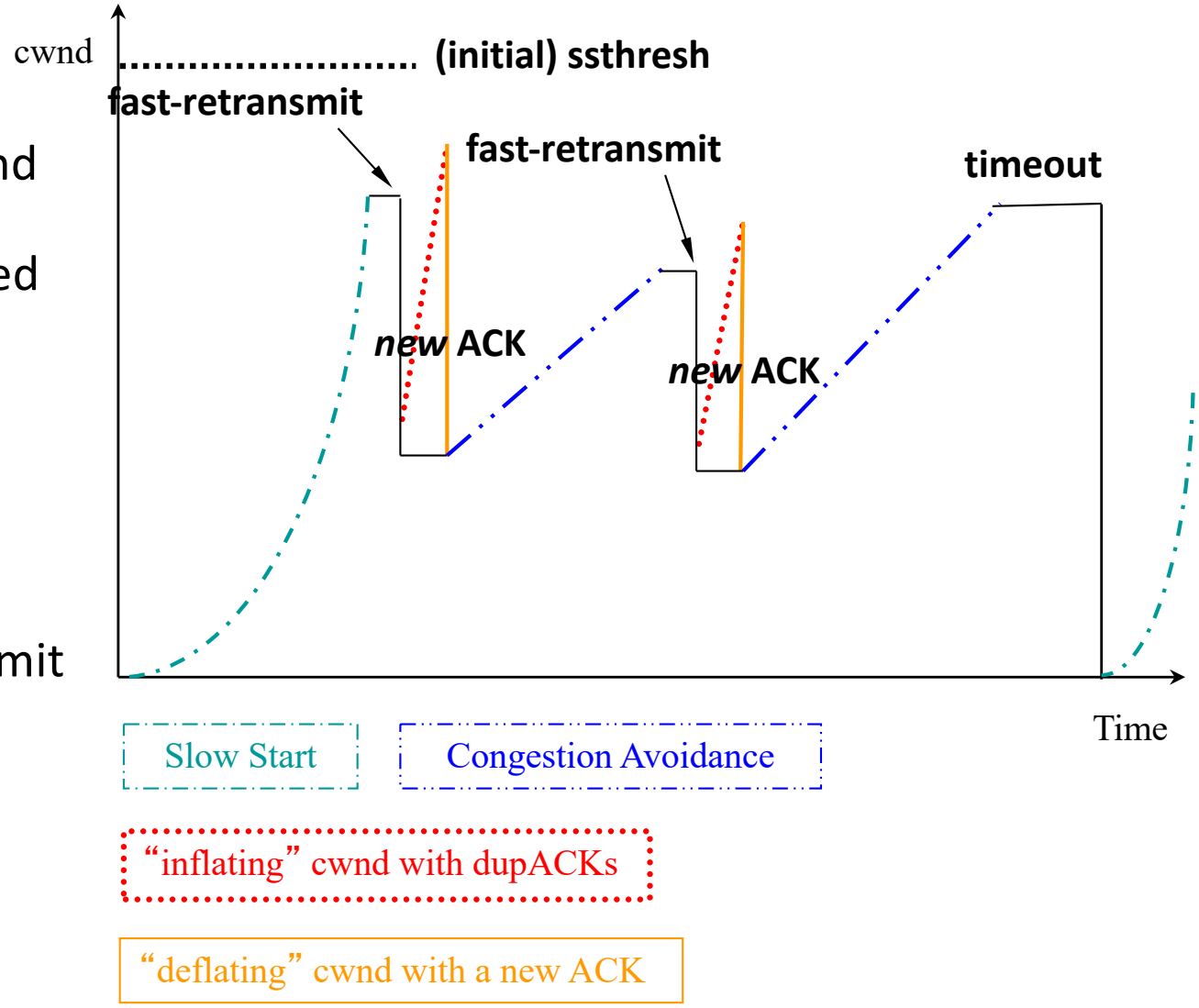
Fast Recovery

Concept:

- After fast retransmit, reduce cwnd by half, and continue sending segments at this reduced level.

Problems:

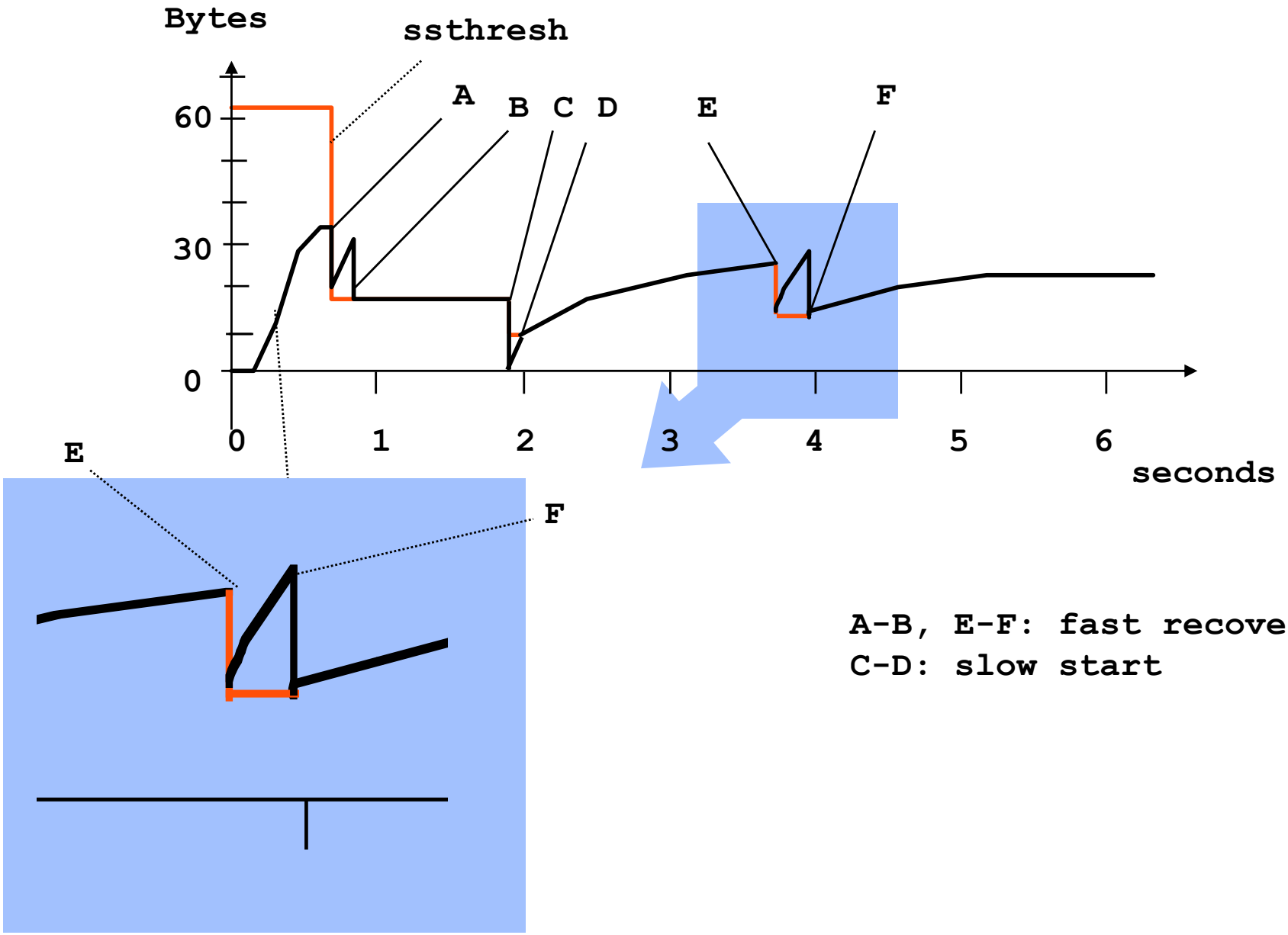
- Sender has too many outstanding segments.
- How does sender transmit packets on a dupACK? Need to use a "trick" - inflate cwnd.



Fast Recovery

- Multiplicative decrease (Reno + NewReno)
  - `ssthresh = 0.5 cwnd`
  - `ssthresh = max (ssthresh, 2 MSS)`
- Fast Recovery (Reno + NewReno)
  - `cwnd = ssthresh + 3 MSS (inflate)`
  - `cwnd = min (cwnd, 64K)`
  - `retransmit the missing segment (n)`
- For each duplicated ACK (Reno + NewReno)
  - `cwnd = cwnd + MSS (keep inflating)`
  - `cwnd = min (cwnd, 64K)`
  - `keep sending segments in the current window`
- For partial ACK (NewReno)
  - `retransmit the first unACKed segment`
  - `cwnd = cwnd – ACKed + MSS (deflate/inflate)`

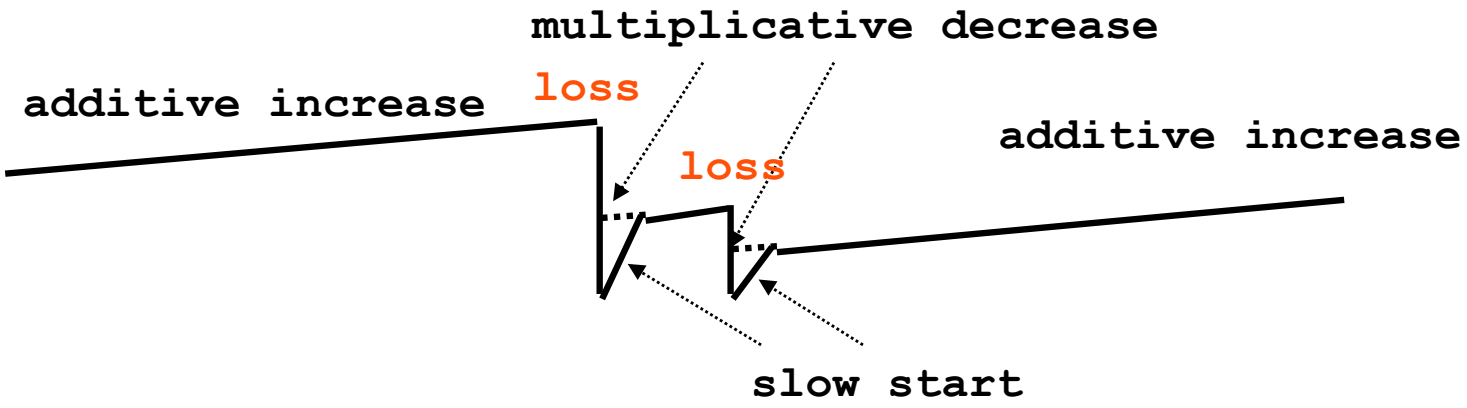
Fast Recovery Example



A-B, E-F: fast recovery
C-D: slow start

TCP Congestion Control

- TCP performs congestion control in end-systems
- Principle
  - sender increases its sending window until loss occurs, then decreases
- Target window
  - additive increase (no loss)
  - multiplicative decrease (loss)



TCP Congestion Control

- 3 phases
  - **slow start**
    - starts with 1, exponential increase up to **`twnd`**
  - **congestion avoidance**
    - additive increase until loss or max window
  - **fast recovery**
    - fast retransmission of one segment
- Slow start entered at setup or after retransmission timeout
- Fast recovery entered at fast retransmit
- Congestion avoidance entered when **`cwnd ≥ ssthresh`**

Summary of TCP Behavior

TCP Variation	Response to 3 dupACKs	Response to Partial ACK of Fast Retransmission	Response to "full" ACK of Fast Retransmission
Tahoe	Do fast retransmit, enter slow start	++cwnd	++cwnd
Reno	Do fast retransmit, enter fast recovery	Exit fast recovery, deflate window, enter congestion avoidance	Exit fast recovery, deflate window, enter congestion avoidance
NewReno	Do fast retransmit, enter modified fast recovery	Fast retransmit and deflate window – remain in modified fast recovery	Exit modified fast recovery, deflate window, enter congestion avoidance

TCP Flavors

- TCP-Tahoe
  - **`cwnd`** =1 on triple dupACK (Fast Retransmit -> Slow Start)
- TCP-Reno
  - **`cwnd`** =1 on timeout
  - **`cwnd`** = **`cwnd`**/2 on triple dupACK (Fast Recovery)
  - **`cwnd`** += **`1`** on dupACK (Fast Recovery)
- TCP-newReno
  - TCP-Reno + improved fast recovery
- TCP-SACK
  - incorporates selective acknowledgements

Quick Review

- **Slow-start**: **`cwnd`** starts at 1MSS
  - ACK of new data:
    - **`cwnd`** → **`cwnd`** + 1 (units of **`cwnd`**)
  - Switch to Congestion Avoidance when **`cwnd`** ≥ **`ssthresh`**

- **Congestion Avoidance**: AIMD
  - 3 dupACKs: **`cwnd`** → **`cwnd`** /2
  - ACK of new data:
    - **`cwnd`** → **`cwnd`** + MSS^2/ **`cwnd`** (bytes)
    - **`cwnd`** → **`cwnd`** + 1/**`cwnd`** (units of **`cwnd`**)

- **Time-out**:
  - **`ssthresh`** → **`cwnd`**/2 (AIMD)
  - **`cwnd`** → 1MSS
  - Do slow-start

Quick Review

Fast Recovery:

- If dupACKcount = 3
  - **`ssthresh`** = **`cwnd`**/2 (**`ssthresh`** just being used to store value)
  - **`cwnd`** = **`ssthresh`** + 3 MSS
- While in fast recovery
  - **`cwnd`** = **`cwnd`** + 1 MSS for each additional duplicate ACK
  - This allows source to send an additional packet…
  - …to compensate for the packet that arrived (generating dupACK)
- Exit fast recovery after receiving new ACK
  - set **`cwnd`** = **`ssthresh`** (which had been set to **`cwnd`**/2 after loss)

Event: ACK (new data)

- If in **Slow Start**
  - **`cwnd`** += 1 (MSS)

Slow start phase

- If in **Fast Recovery**
  - **`cwnd`** = **`ssthresh`**
  - Leave Fast Recovery

Leaving Fast Recovery

- Else (in **Congestion Avoidance**)
  - **`cwnd`** = **`cwnd`** + 1/**`cwnd`**

"Congestion Avoidance" phase (additive increase)

- Reset DupACKcount

Event: dupACK

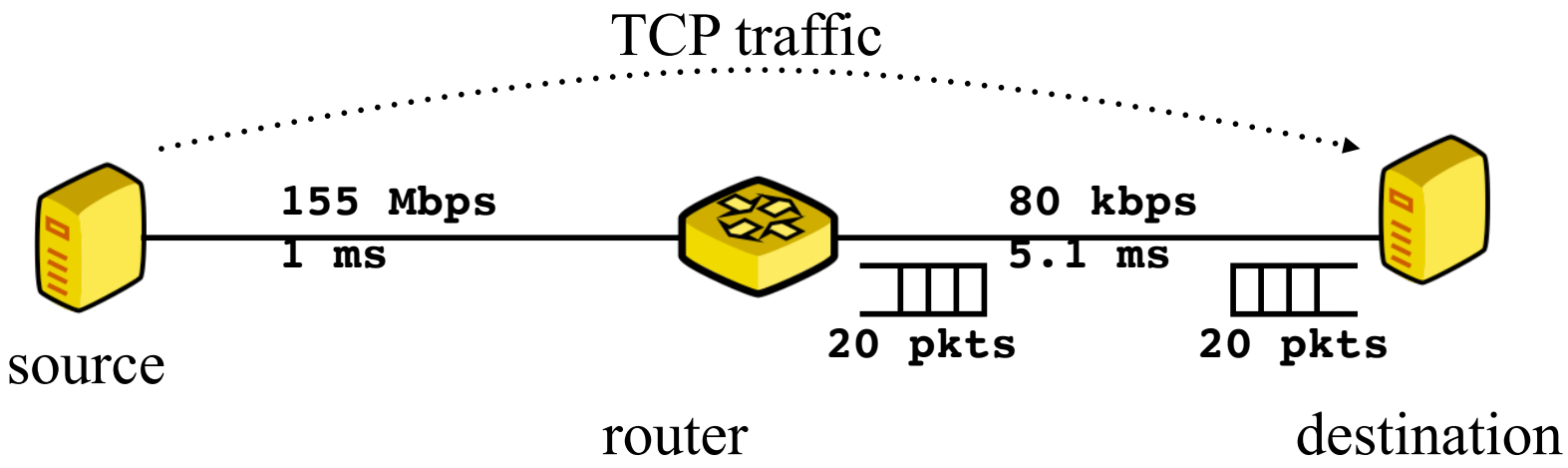
- dupACKcount ++

- If dupACKcount = 3 /* Fast Retransmit */
  - **`ssthresh`** = **`cwnd`**/2
  - **`cwnd`** = **`ssthresh`** + 3
  - **and retransmit packet!**

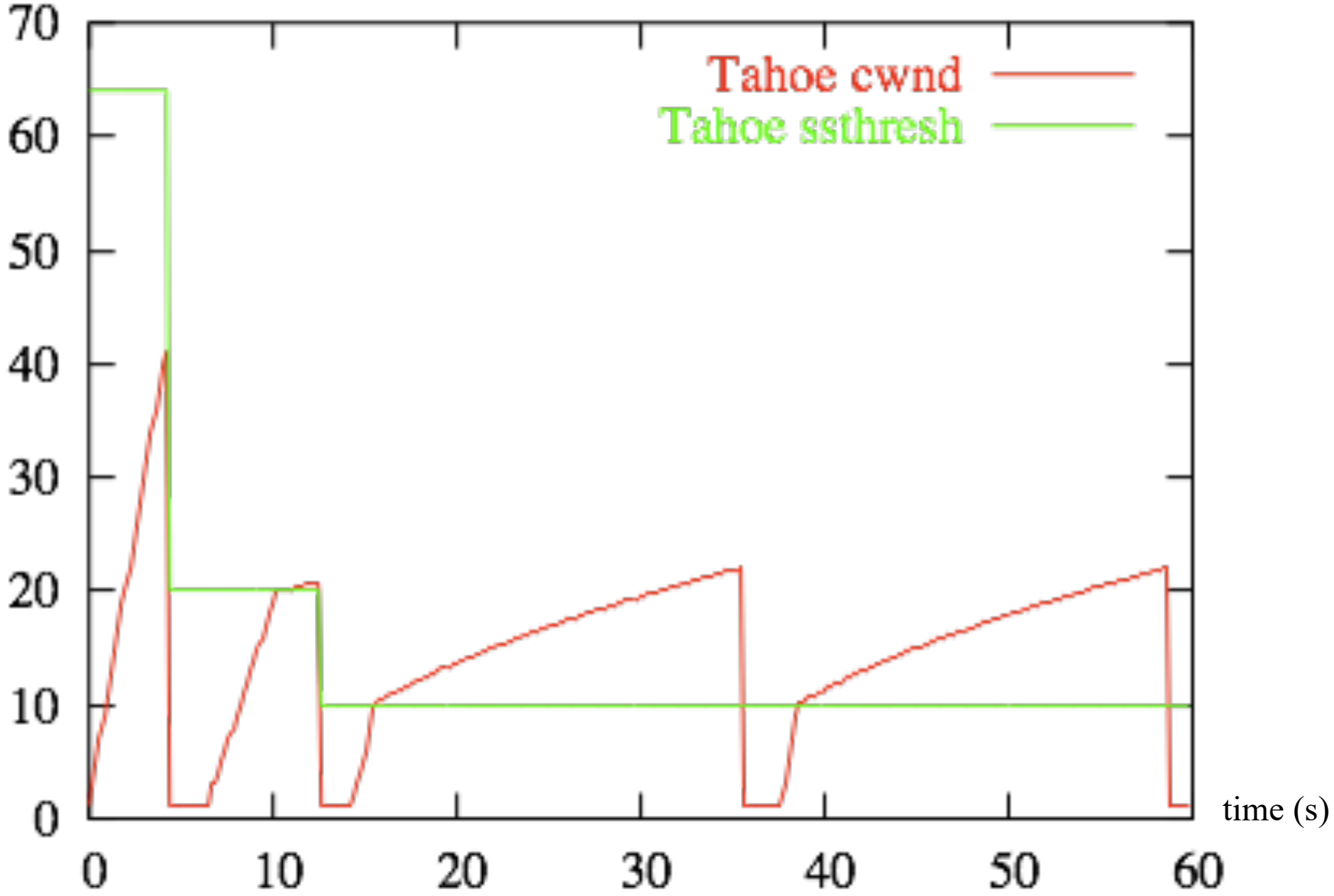
- If dupACKcount > 3 /* Fast Recovery */
  - **`cwnd`** = **`cwnd`** + 1 (MSS)

Simulations

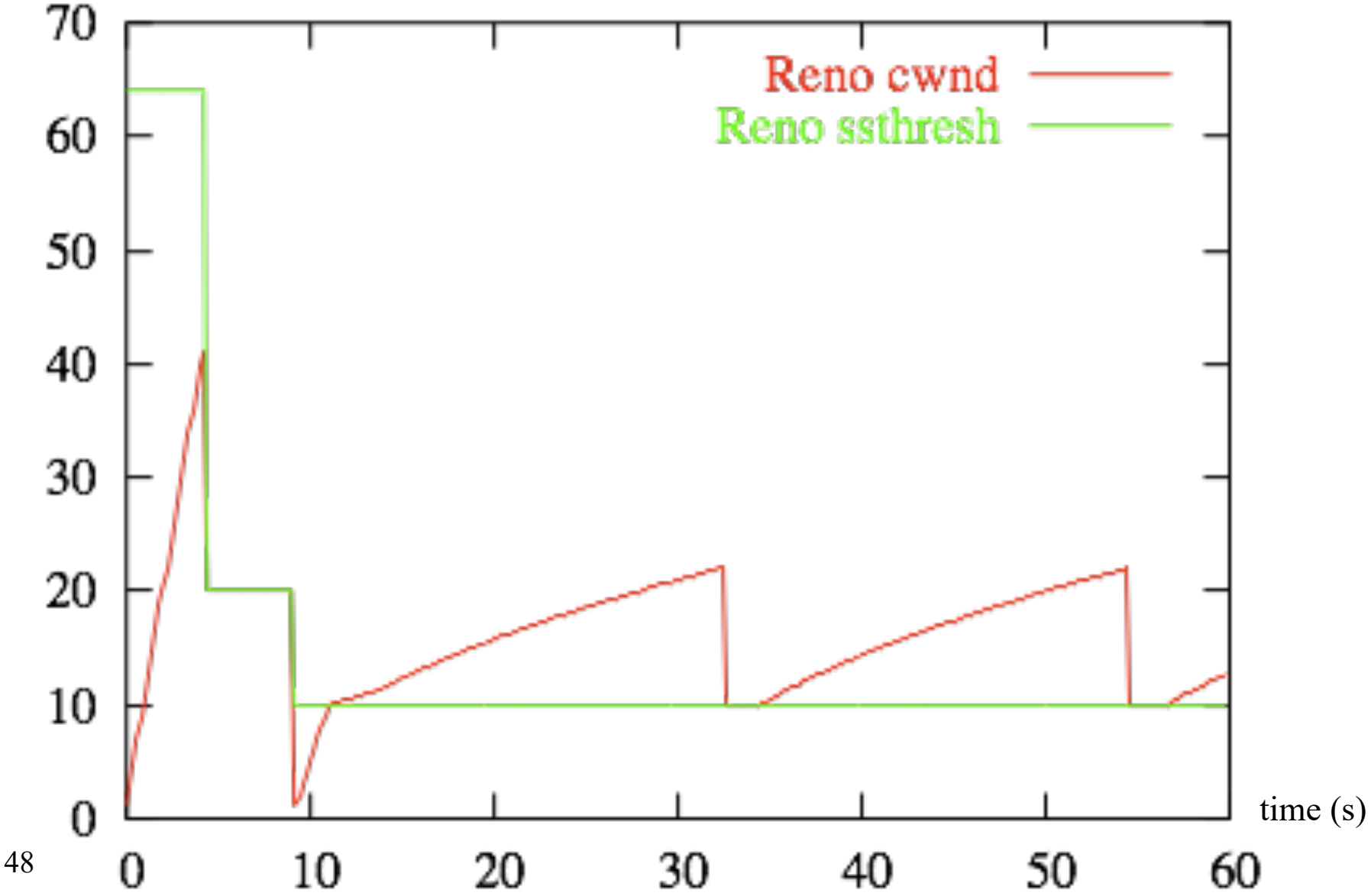
- Using ns-2 network simulator
- Simulated network



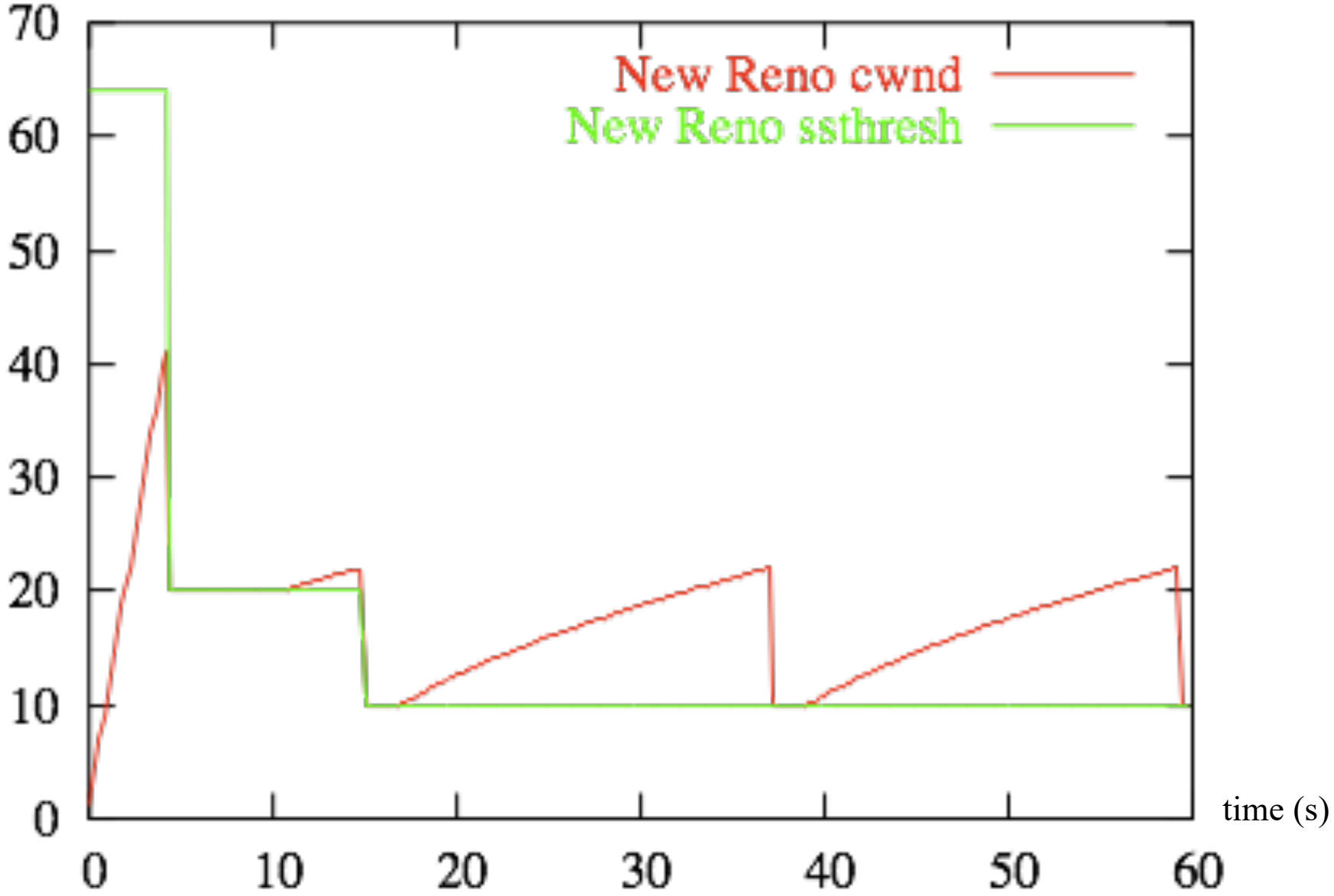
TCP Tahoe


Tahoe cwnd
Tahoe ssthresh
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
time (s)

TCP Reno


70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
Reno cwnd
Reno ssthresh
time (s)

TCP New Reno


New Reno cwnd
New Reno ssthresh
70
60
50
40
30
20
10
0
0
10
20
30
40
50
60
time (s)

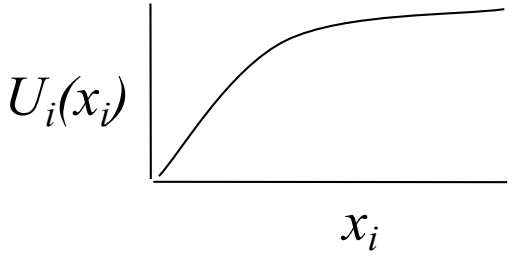
Fairness of TCP Reno

Utility Function

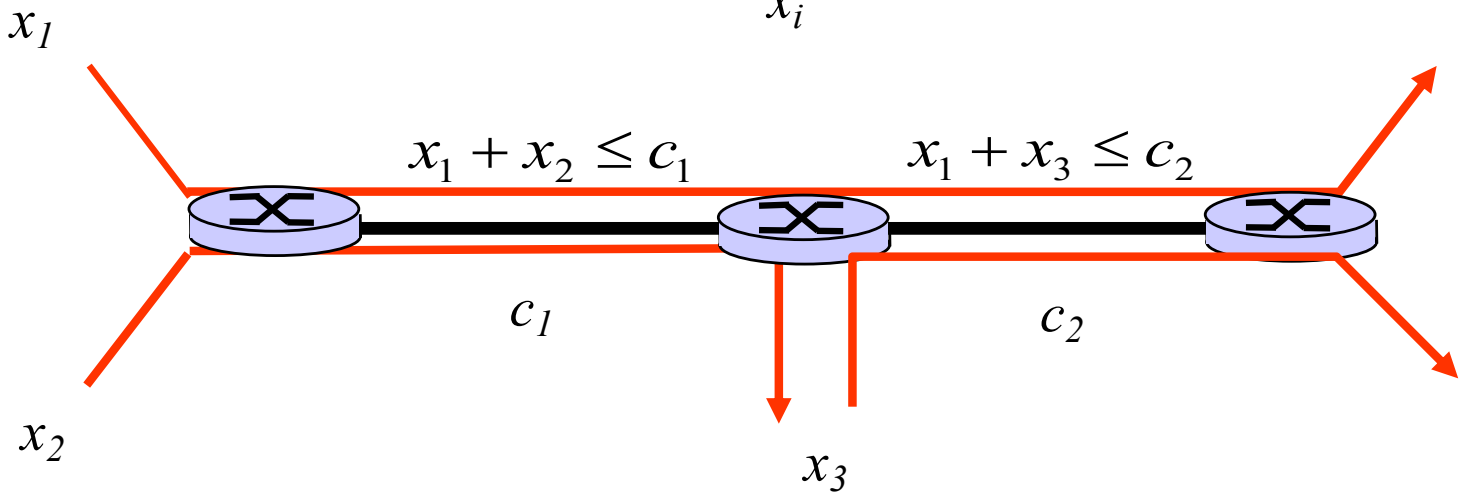
- Utility Function: maps from a given quality of service to a level of satisfaction
- Network: Links l each of capacity c_l
- Sources i: $L(i)$, $U_i(x_i)$

$L(i)$ - links used by source i

$U_i(x_i)$ - utility if source rate = x_i



example utility function for elastic application



Utility Fairness - proportional fairness

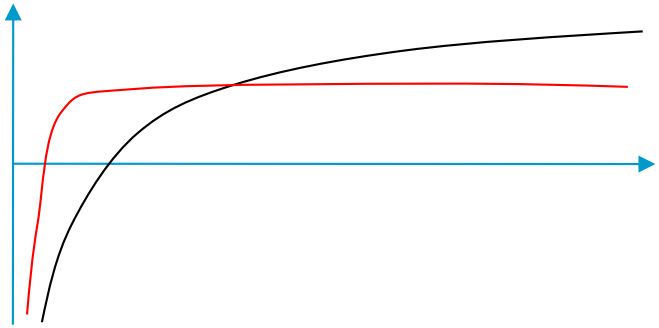
- One can interpret proportional fairness as the allocation that maximizes a global utility

$$\sum_i U_i(x_i) \text{ with } U_i(x_i) = \log x_i$$

Utility Fairness – max-min

- It can be shown that max-min fairness is the limit of utility fairness when the utility function converges to a step function but max-min fairness cannot be expressed exactly as a utility fairness

- $U(x) = 1 - x^{-m},\ m \rightarrow \infty$



$U(x) = \log(x)$: proportional fairness

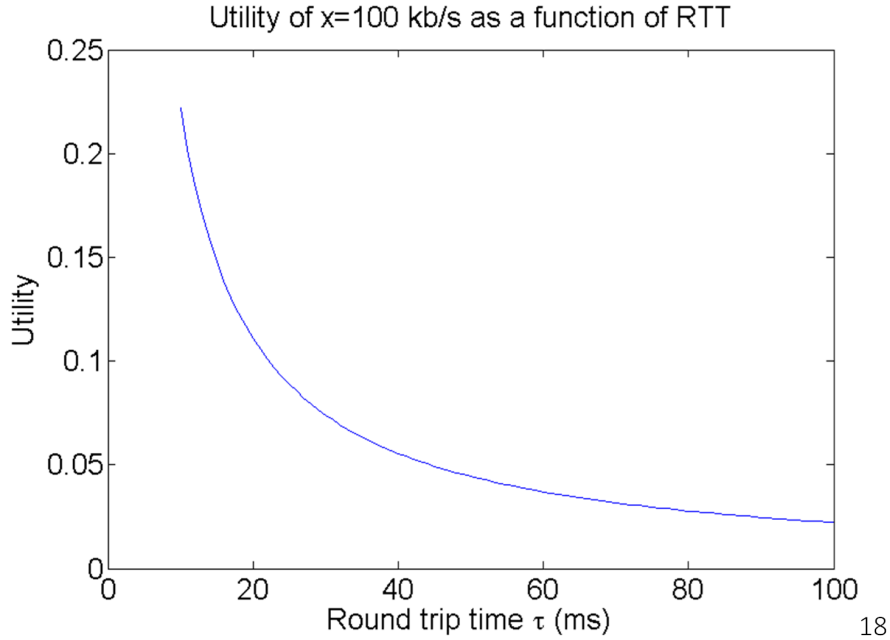
$U(x) = 1 - x^{-m}$: m large => ~ max min fairness

Fairness of TCP Reno

- For long lived flows, the rates obtained with TCP Reno are as if they were distributed according to utility fairness, with utility of flow given by

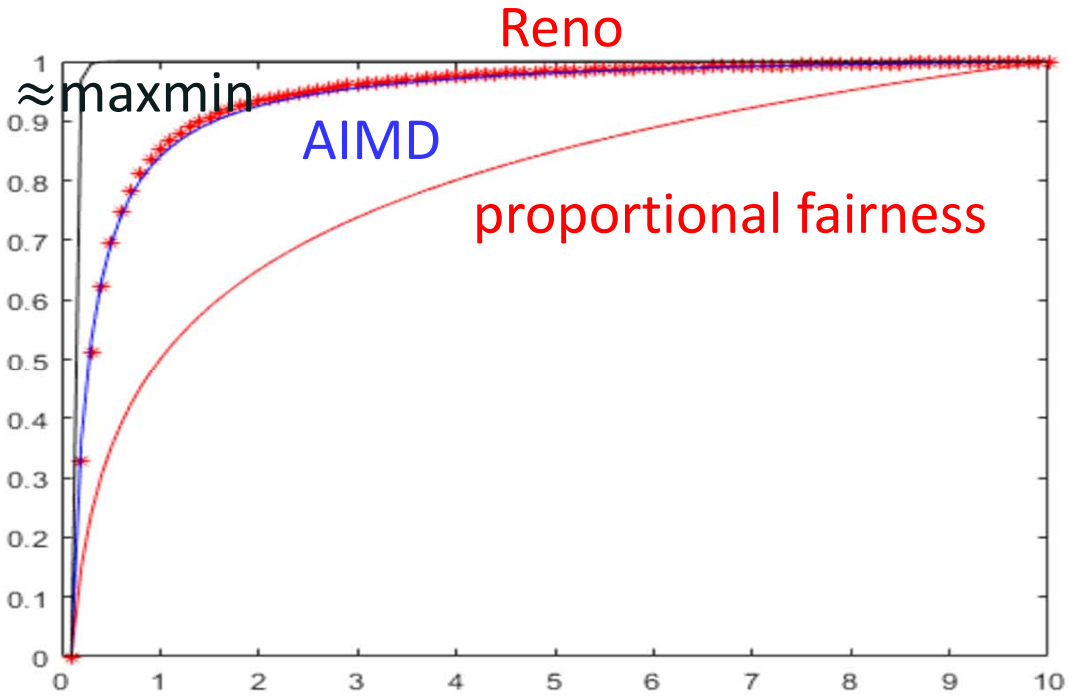
$$\frac{\sqrt{2}}{\tau_i} \arctan \frac{x_i \tau_i}{\sqrt{2}}$$

- with x_i = rate = W/τ_i, τ_i = RTT
- The utility is a decreasing function of RTT



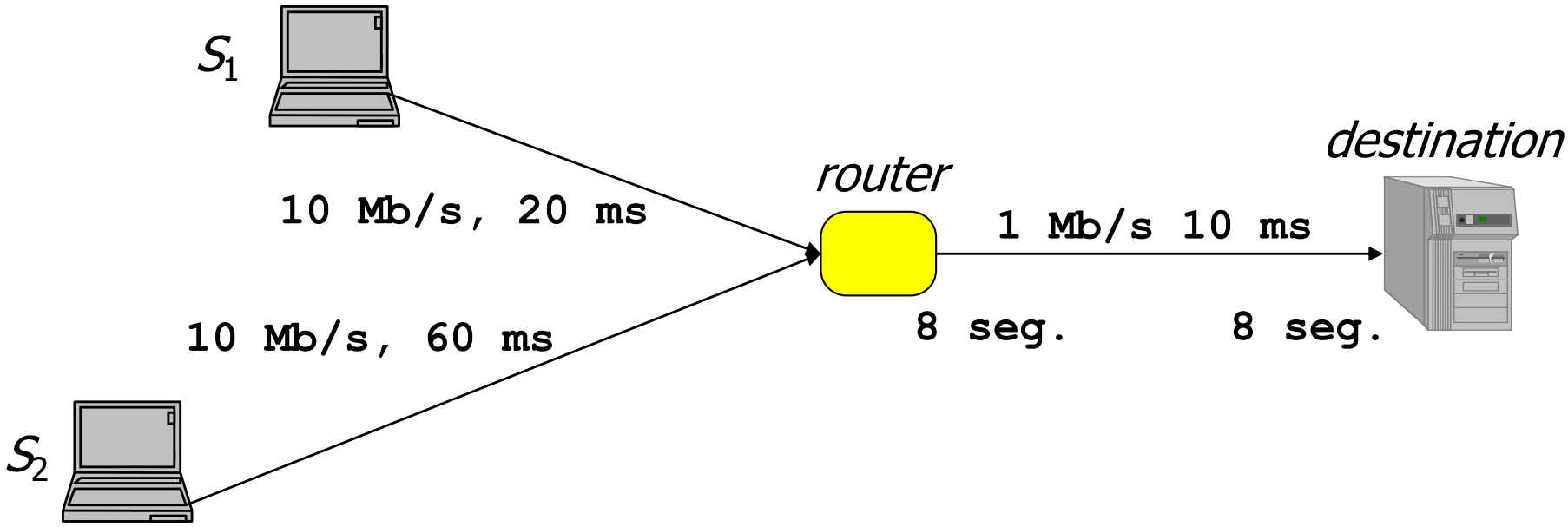
Fairness of TCP Reno

- For sources that have same RTT, the fairness of TCP is between max-min and proportional fairness, closer to proportional fairness



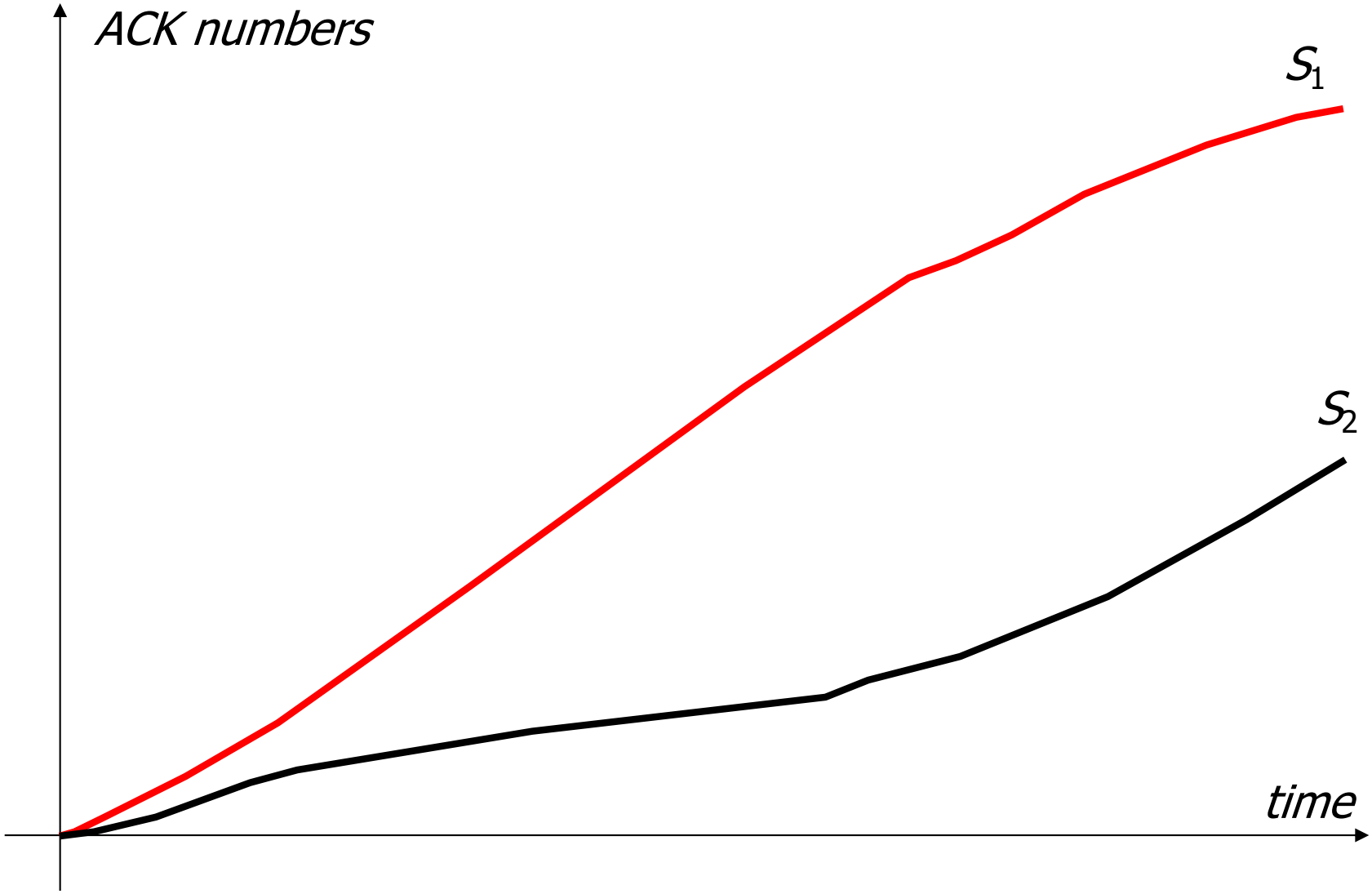
rescaled utility functions;
RTT = 100 ms
maxmin approx. is $U(x) = 1 - x^{-5}$

Fairness of the TCP



- Example network with two TCP sources
  - link capacity, delay
  - limited queues on the link (8 segments)
- NS simulation

Throughput in time



Fairness of the TCP

- S_1 has a smaller RTT than S_2
- Utility is less when RTT is large, therefore TCP tries less hard to give a high rate to sources with large RTT
- S_2 gets less

RTT Bias of TCP

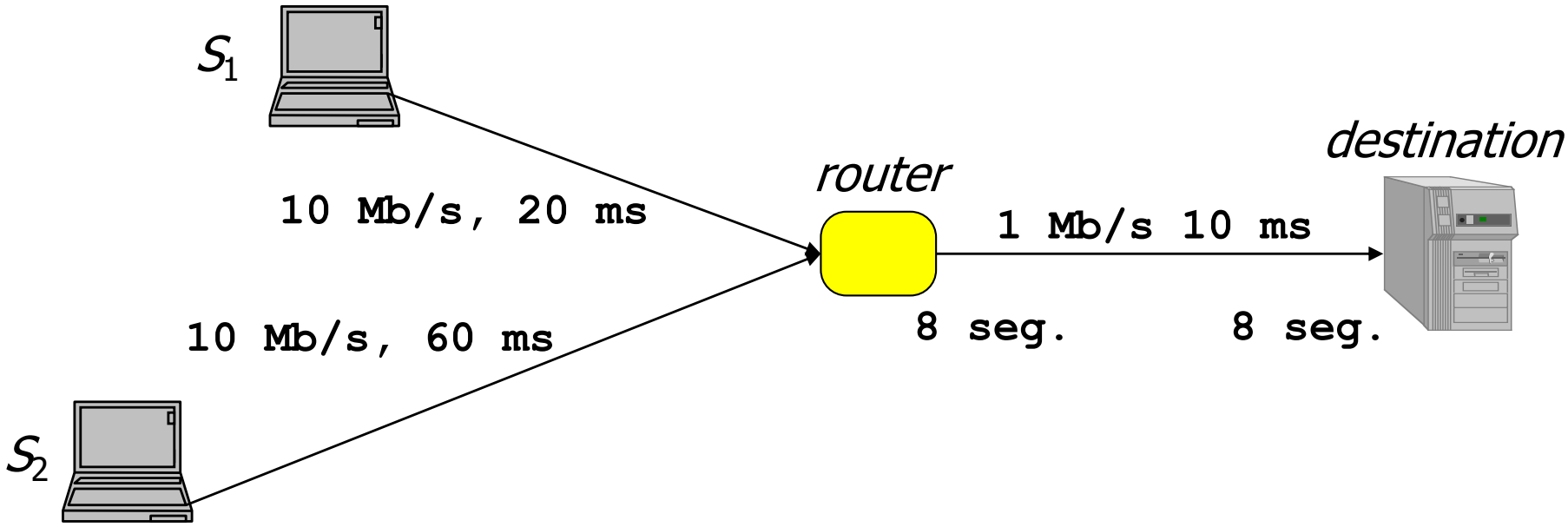
- A source that uses many hops obtains less rate because of two combined factors:
  1. this source uses more resources. The mechanic of proportional fairness leads to this source having less rate – this is desirable in view of the theory of fairness (improved throughput).
  2. this source has a larger RTT. The mechanics of additive increase leads to this source having less rate – this is an undesired bias in the design of TCP Reno - additive increase is one packet per RTT (instead of one packet per constant time interval).

TCP Loss - Throughput formulae

$$\theta = \frac{L}{T}\frac{C}{\sqrt{q}}$$

- TCP connection with
  - RTT T
  - segment size L
  - average packet loss ratio q
  - constant $C = 1.22$
- Transmission time negligible compared to RTT, losses are rare, time spent in Slow Start and Fast Recovery negligible

Fairness of the TCP



- What ratio between throughputs of S_1 and S_2?
  - $\vartheta_1 = 3/7\ \vartheta_2$
  - $\vartheta_1 = \vartheta_2$
  - $\vartheta_1 = 7/3\ \vartheta_2$
  - $\vartheta_1 = 1/3\ \vartheta_2$

TCP Friendly Applications

- All TCP/IP applications that generate long lived flows should mimics the behavior of a TCP source
  - RTP/UDP flow sending video/audio data
- Adaptive algorithm
  - application determines the sending rate
  - feedback - amount of lost packets, loss ratio
  - sending rate = rate of a TCP flow experiencing the same loss ratio

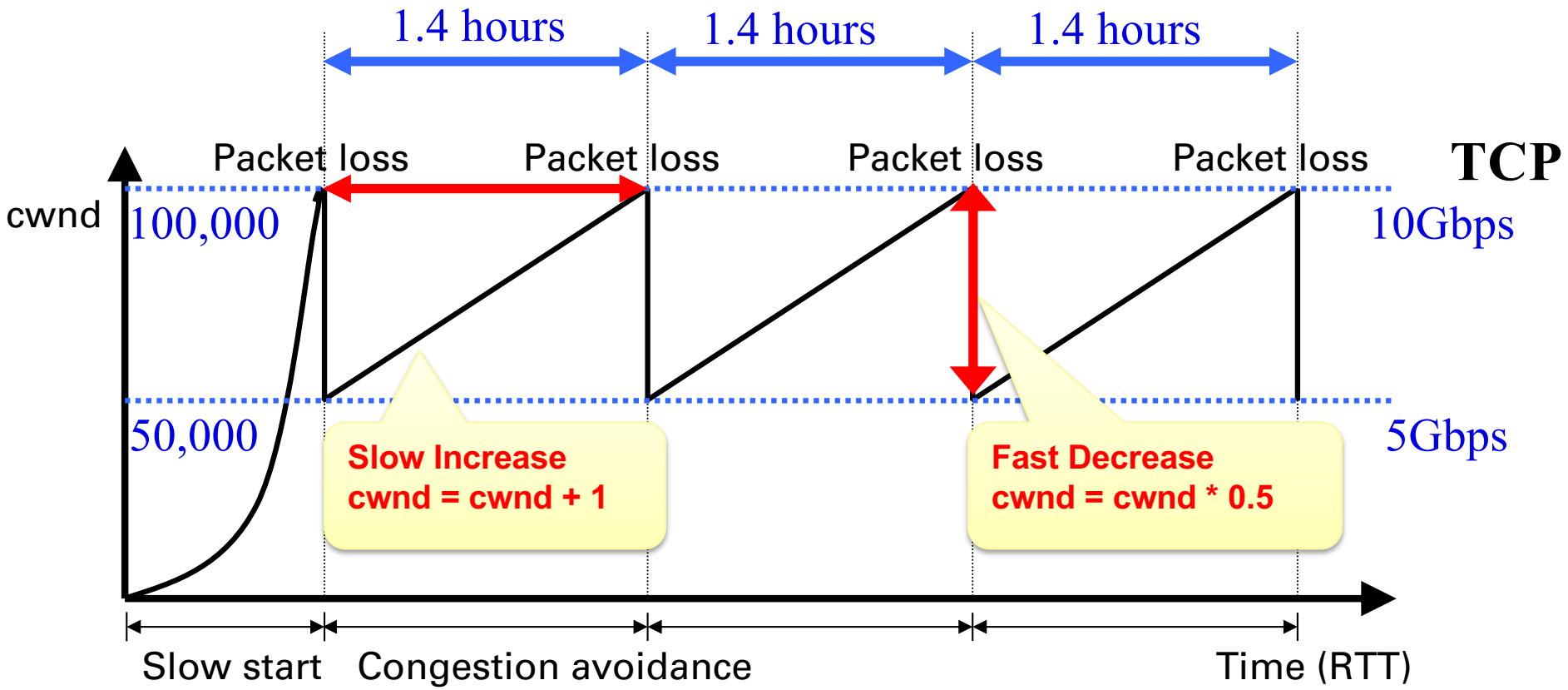
TCP Reno drawbacks

- RTT bias – not nice for users in New Zealand
- Periodic losses must occur, not nice for video streaming
- TCP controls the window, not the rate:
  - Large bursts typically occur when packets are released by host following e.g., a window increase – not nice for queues in the Internet, makes non smooth behavior
- Self inflicted delay:
  - if network buffers (in routers and switches) are large, TCP first fills buffers before adapting the rate - RTT is increased unnecessarily
  - Buffers are constantly full, which reduces their usefulness (bufferbloat) and increases delay for all users. Interactive, short flows see large latency when buffers are large and full

TCP Cubic

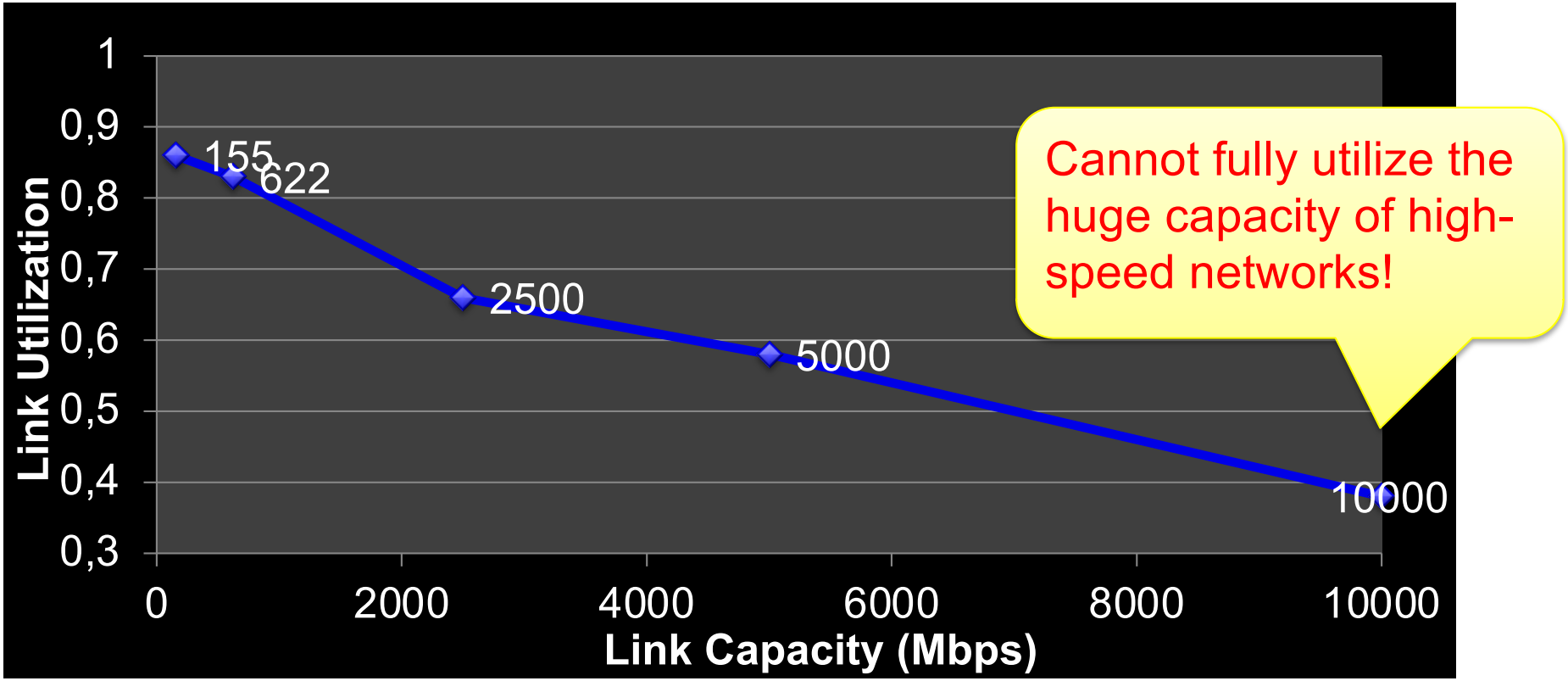
Long Fat Networks (LFNs)

- In an LFN, additive increase is too slow (MSS = 1250 B, RTT = 100 ms)



Standard TCP

- Underutilization of the bandwidth in **High-Speed Network**



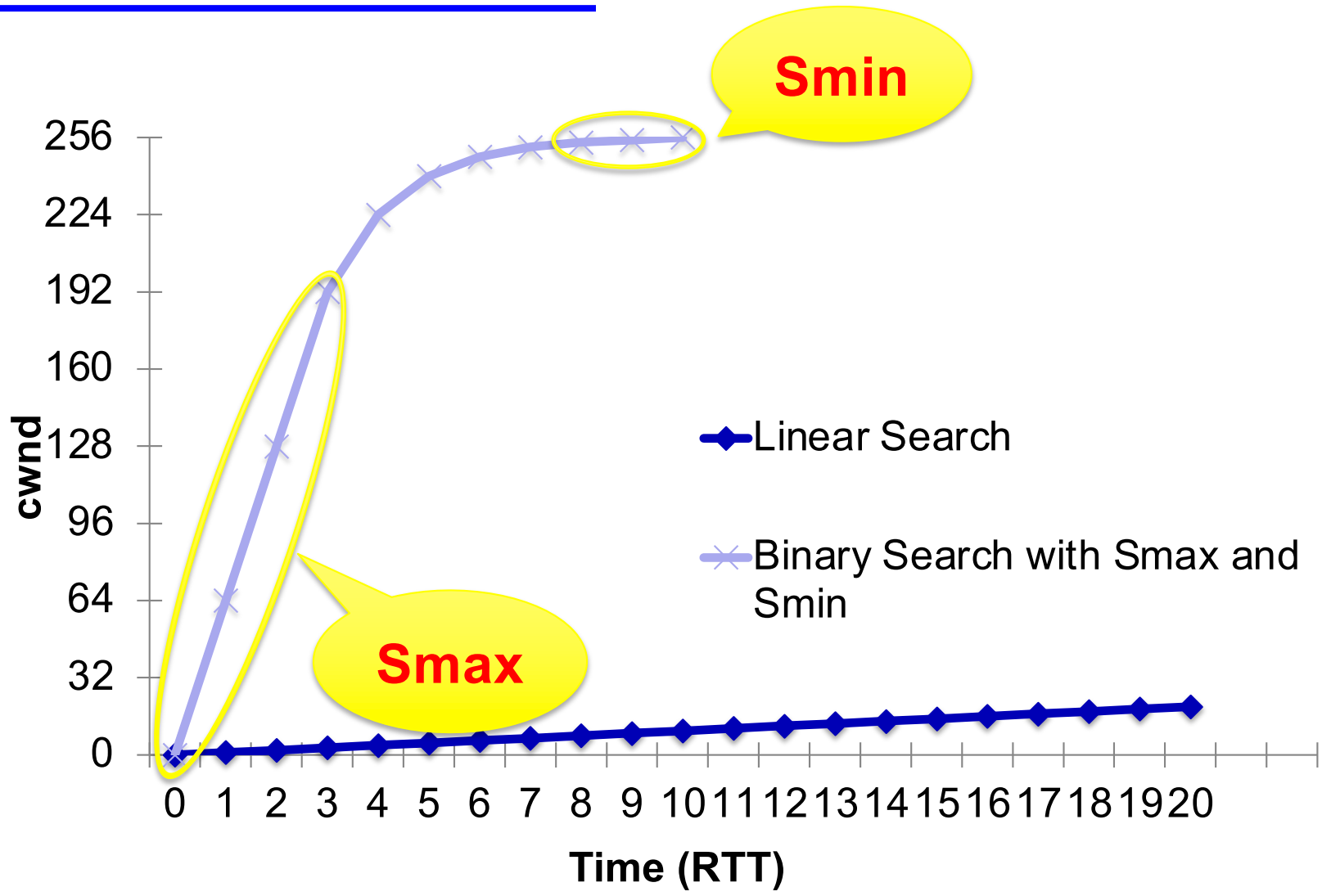
NS-2 Simulation (100 sec)

- Link Capacity = 155Mbps, 622Mbps, 2.5Gbps, 5Gbps, 10Gbps,
- Drop-Tail Routers, 0.1BDP Buffer
- 5 TCP Connections, 100ms RTT, 1000-Byte Packet Size

CUBIC and BIC

- BIC: Binary Increase Congestion Control
  - Faster increase - binary search of a midpoint between Wmax (window size just before the last congestion event) and Wmin (window size just after the the last congestion event)
- CUBIC:
  - is a less aggressive and more systematic derivative of BIC, in which the window is a cubic function of time since the last congestion event, with the inflection point set to the window prior to the event

BIC with no loss



- Smin: minimum increment
- Smax: maximum increment

BIC Algorithm

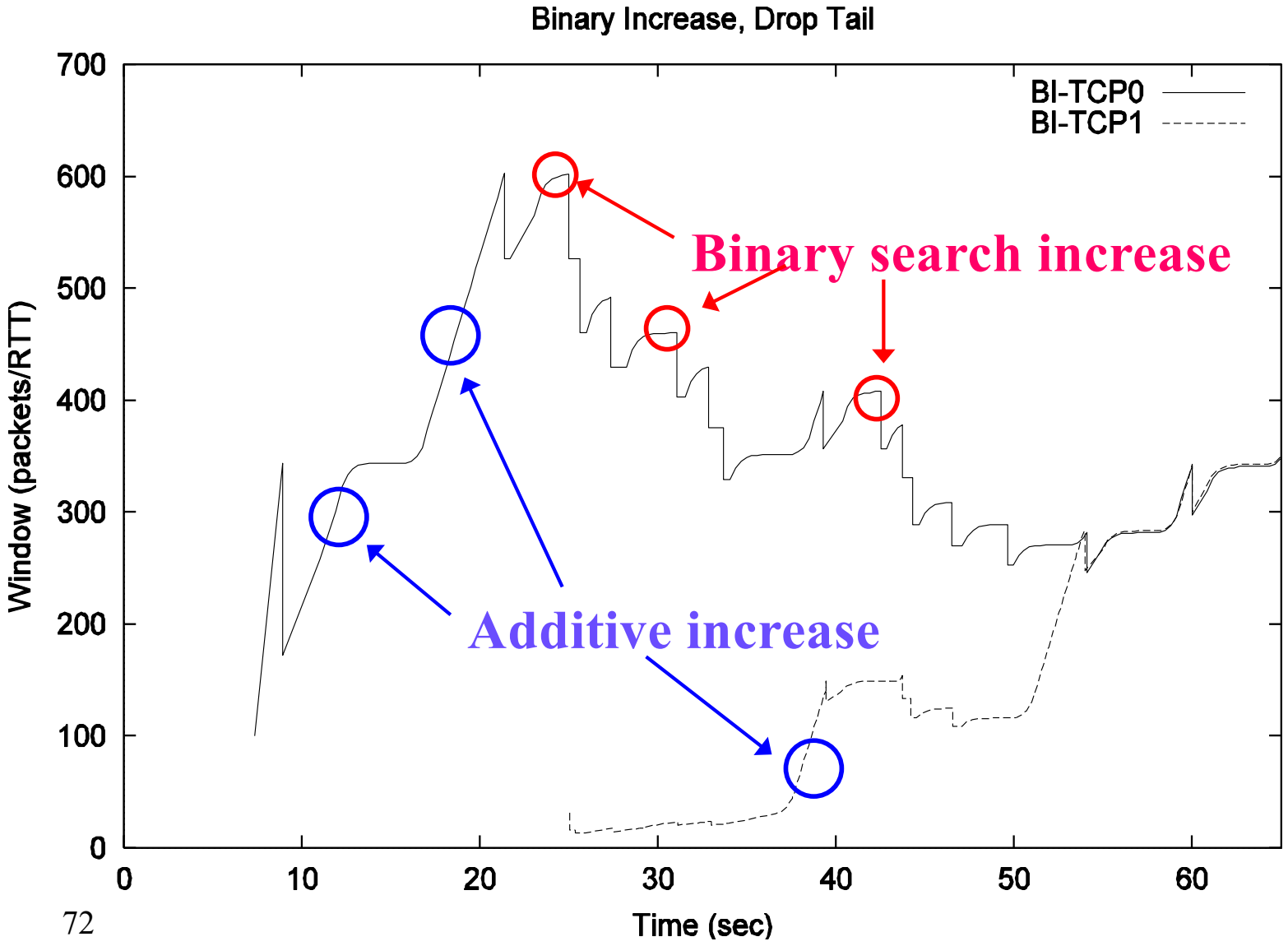
- If cwnd < low_window, normal TCP:
  - No loss, full window
    - cwnd = cwnd + 1
  - Enter recovery
    - cwnd = cwnd * 0.5
- Else, BIC

BIC Algorithm

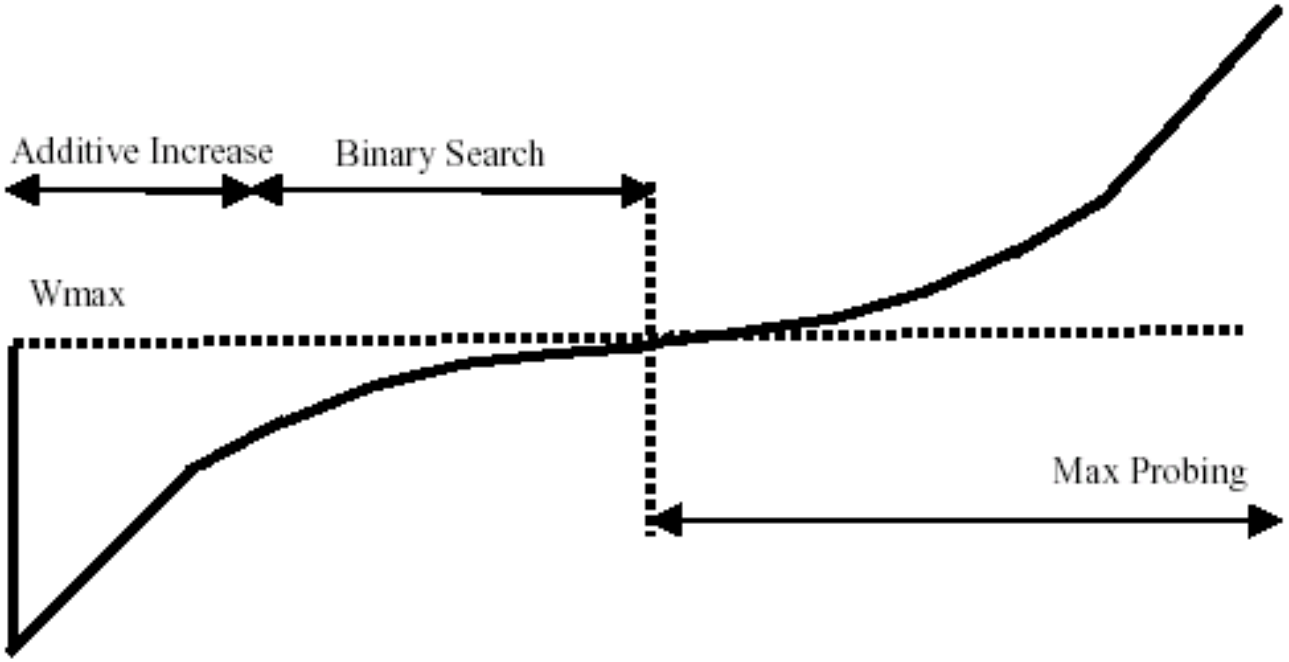
- No loss, full window
  - if cwnd < Wmax
    - bic_inc = (Wmax – cwnd) / 2
  - else
    - bic_inc = cwnd – Wmax
  - if (bic_inc > Smax)
    - bic_inc = Smax;
  - else if (bic_inc < Smin)
    - bic_inc = Smin;
  - cwnd = cwnd + bic_inc
- Recovery
  - If cwnd < Wmax
    - Wmax = cwnd * (2 – ß / 2)
  - Else
    - Wmax = cwnd
  - cwnd *= 1 - ß

- Wmax is the window size just before the last congestion event (loss)

Binary Increase Congestion Control (BIC)



In Summary BIC function



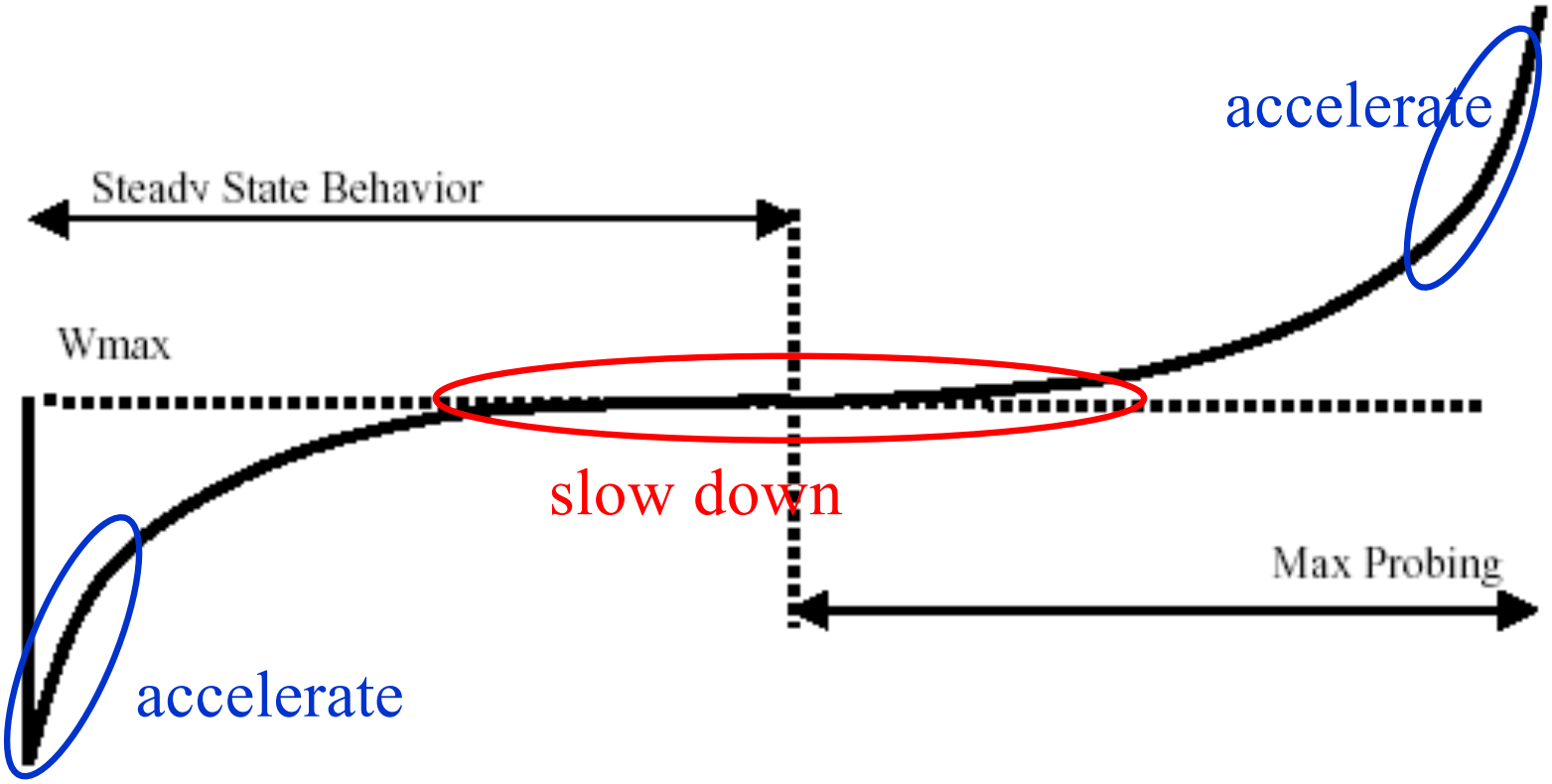
- BIC overall performs very well in evaluation of advanced TCP stacks on fast long-distance production.

- BIC growth function can be still aggressive for TCP especially under short RTTs or low speed networks.

Why CUBIC?

- Window control of BIC is so complex!
- BIC growth function can still be too aggressive for TCP, especially under short RTT or low speed networks.
- BIC still has room for improving TCP-friendliness and RTT-fairness!
- Cubic keeps the same slow start, congestion avoidance, fast recovery phases as TCP Reno, but:
  - Multiplicative Decrease is 0.7 (instead of 0.5)
  - During congestion avoidance, the increase is not additive but cubic

The CUBIC function



$$W_{cubic} = C(t-K)^3 + W_{\max} \qquad K = \sqrt[3]{W_{\max}\beta/C}$$

where C is a scaling factor, t is the elapsed time from the last window reduction, and β is a constant multiplication decrease factor

CUBIC Algorithm

- ACK received

$$cwnd = C \cdot (t - K)^3 + W_{\max}$$

  - C is a scaling factor
  - t is the elapsed time from the last window reduction
  - $Wmax$ is the window size just before the last window reduction
  - K is updated at the time of last lost event
- cwnd cannot be less than

$$cwnd = \beta \cdot W_{\max} + 3 \cdot \frac{1-\beta}{1+\beta} \cdot \frac{t}{RTT}$$

- as to keep the growth rate the same as standard TCP in **short RTT** networks.

- Recovery
  - Update K with:

$$K = \sqrt[3]{\beta \cdot W_{\max} / C}$$

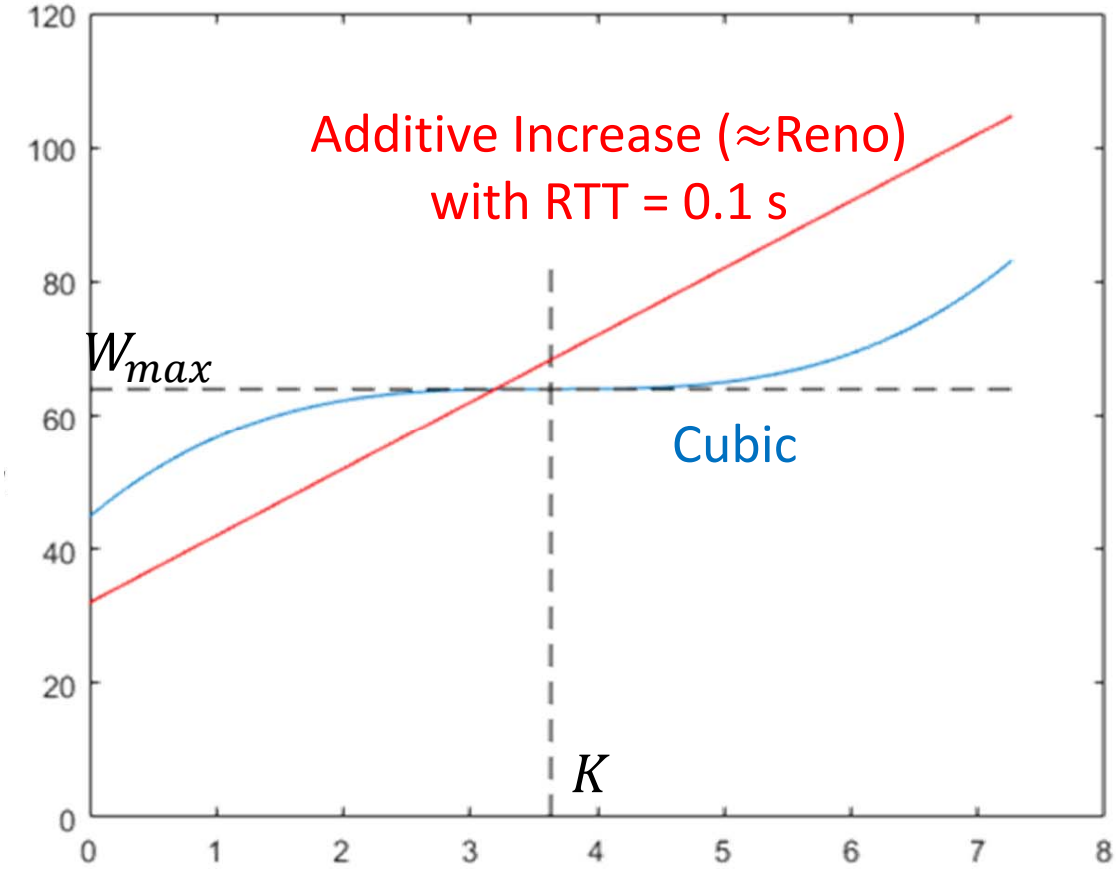
  - Update $Wmax$ with:

$$W_{\max} = \beta \cdot W_{\max}$$

  - β is a constant multiplication decrease factor

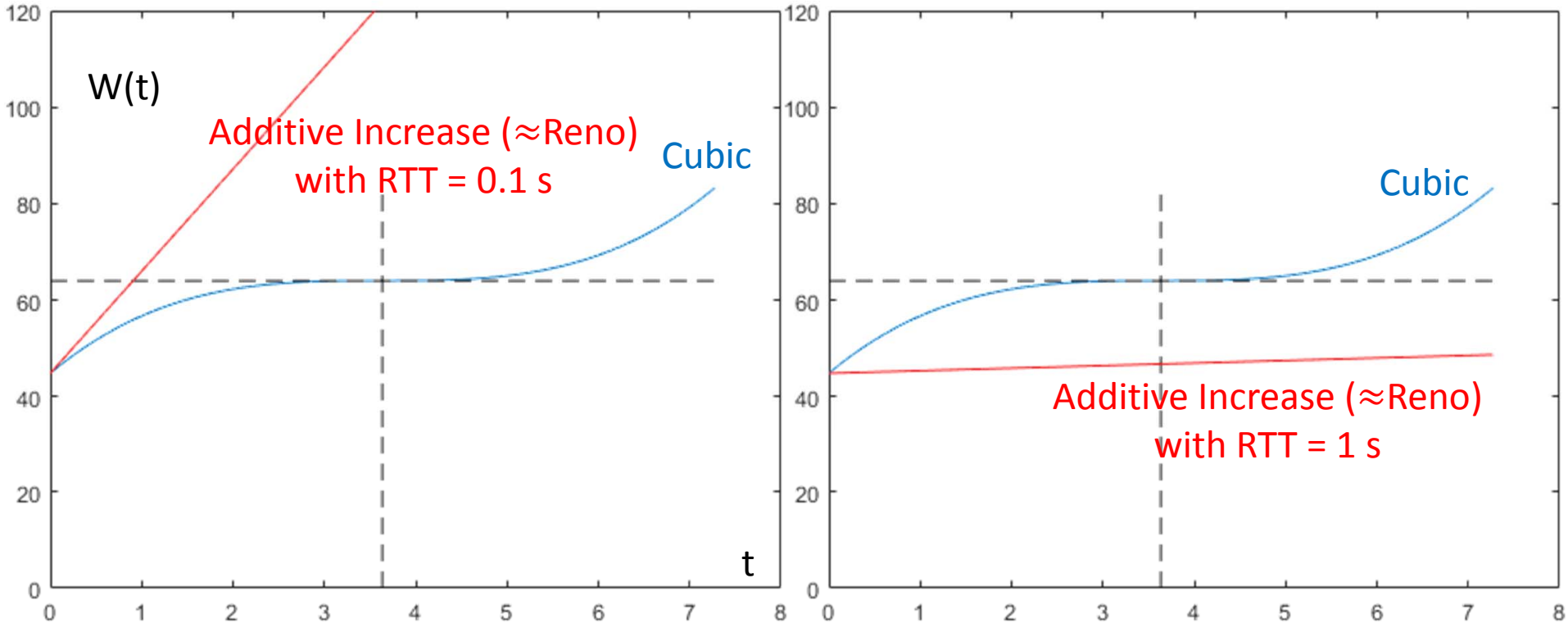
Cubic versus Reno

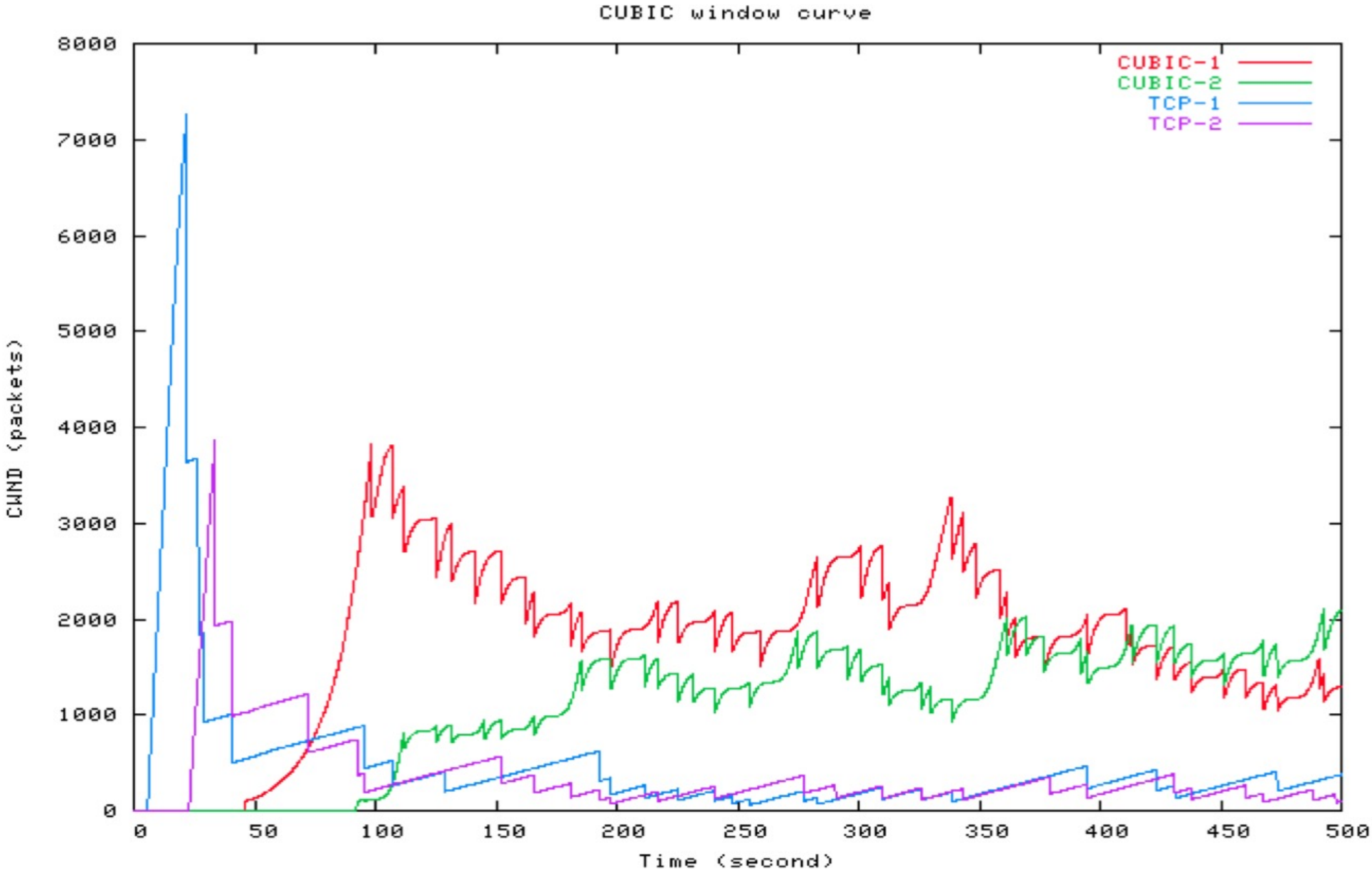
- Cubic increases window in concave way until reaches $Wmax$ then increases in a convex way
- Cubic window function is independent of RTT



Cubic versus Reno

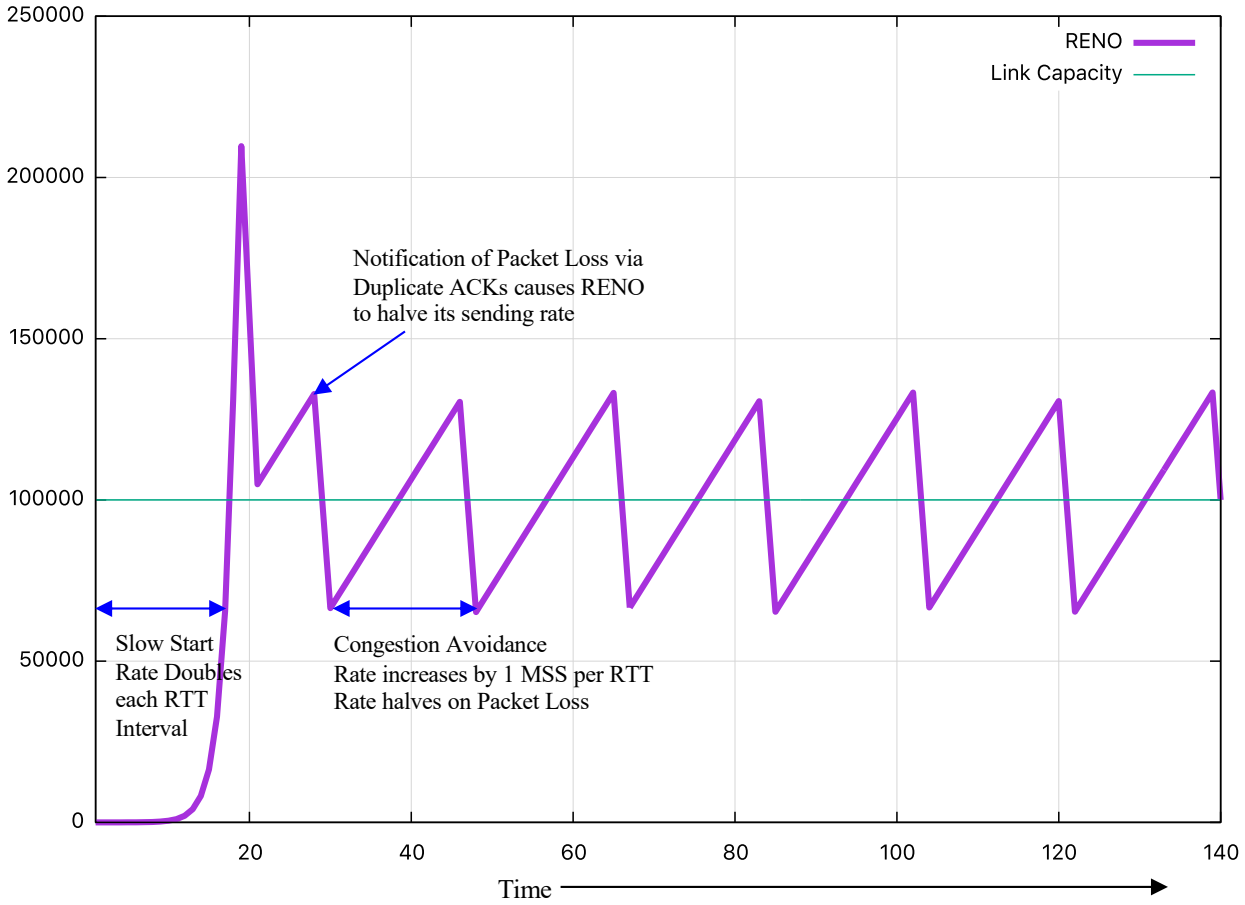
- Cubic window function is independent of RTT
- It is slower than Reno when RTT is small, larger when RTT is large



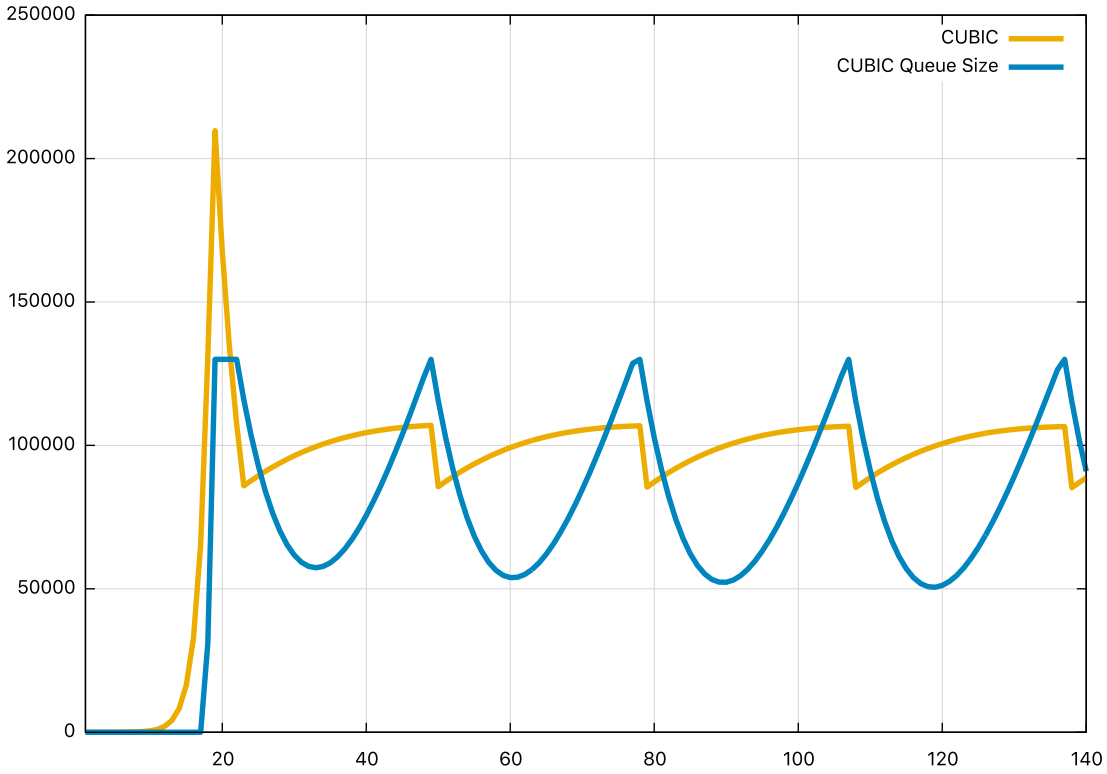


CUBIC window curves with competing flows (NS simulation in a network with 500Mbps and 100ms RTT), C = 0.4, β = 0.8.

Idealised TCP Reno



CUBIC and Queue formation



CUBIC characteristics

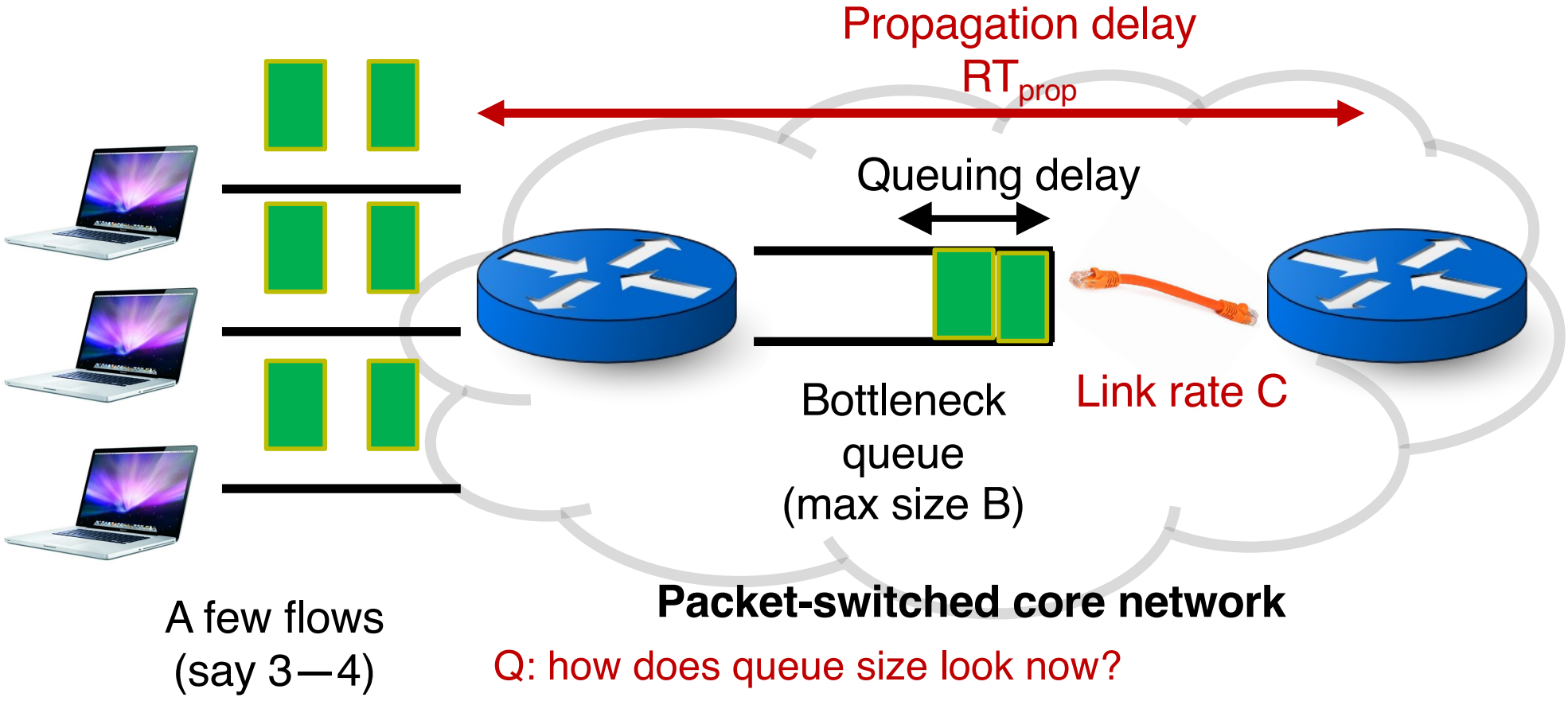
- Cubic throughput ≥ Reno throughput with equality when RTT or bandwidth-delay product is small
- More equitable BW allocations among flows with different RTTs
- Default TCP algorithm in Linux

TCP BBR

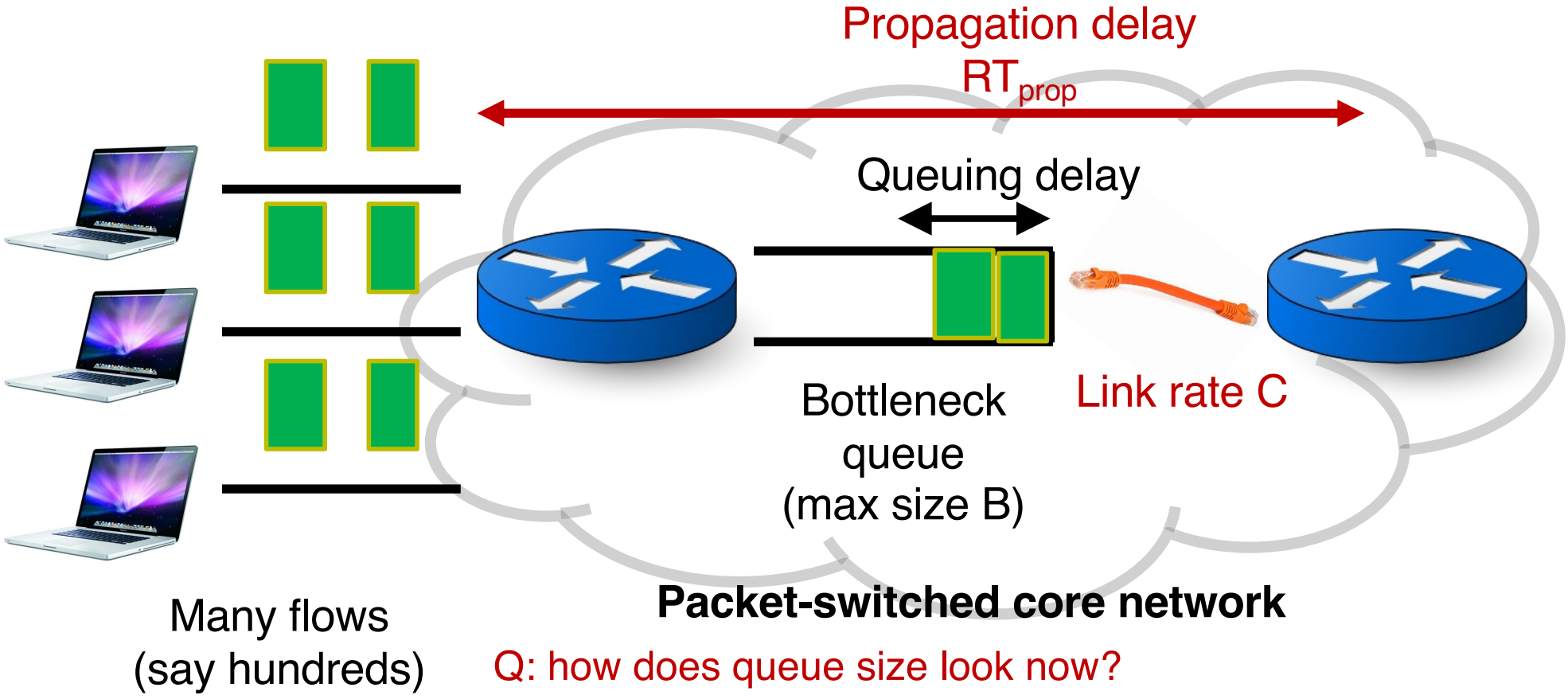
BBR - Bottleneck Bandwidth and RTT

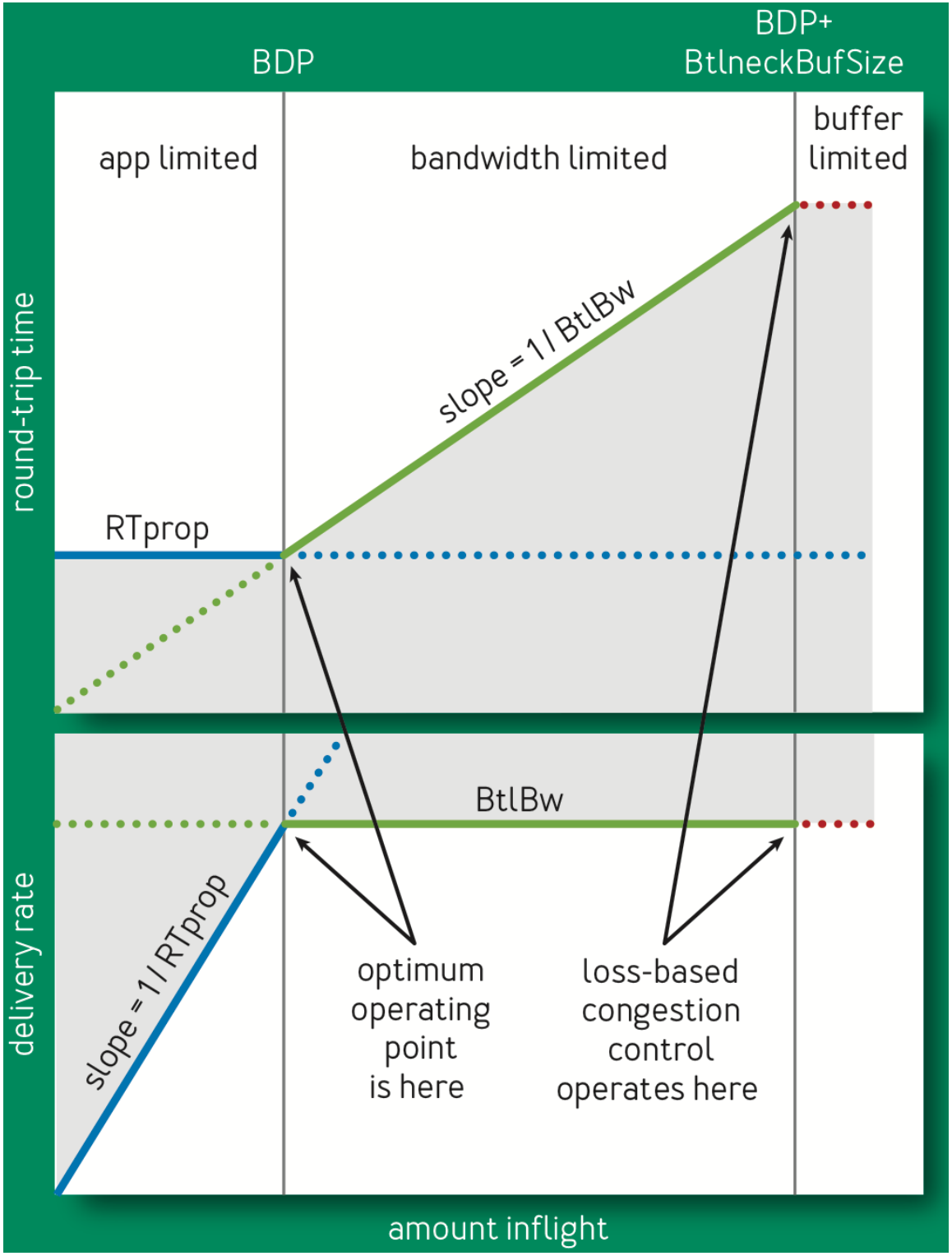
- Avoid bufferbloat
- How? TCP source controls rate (not window), estimates the rate and RTT by periodically overshooting/undershooting
- Losses are ignored

Network model

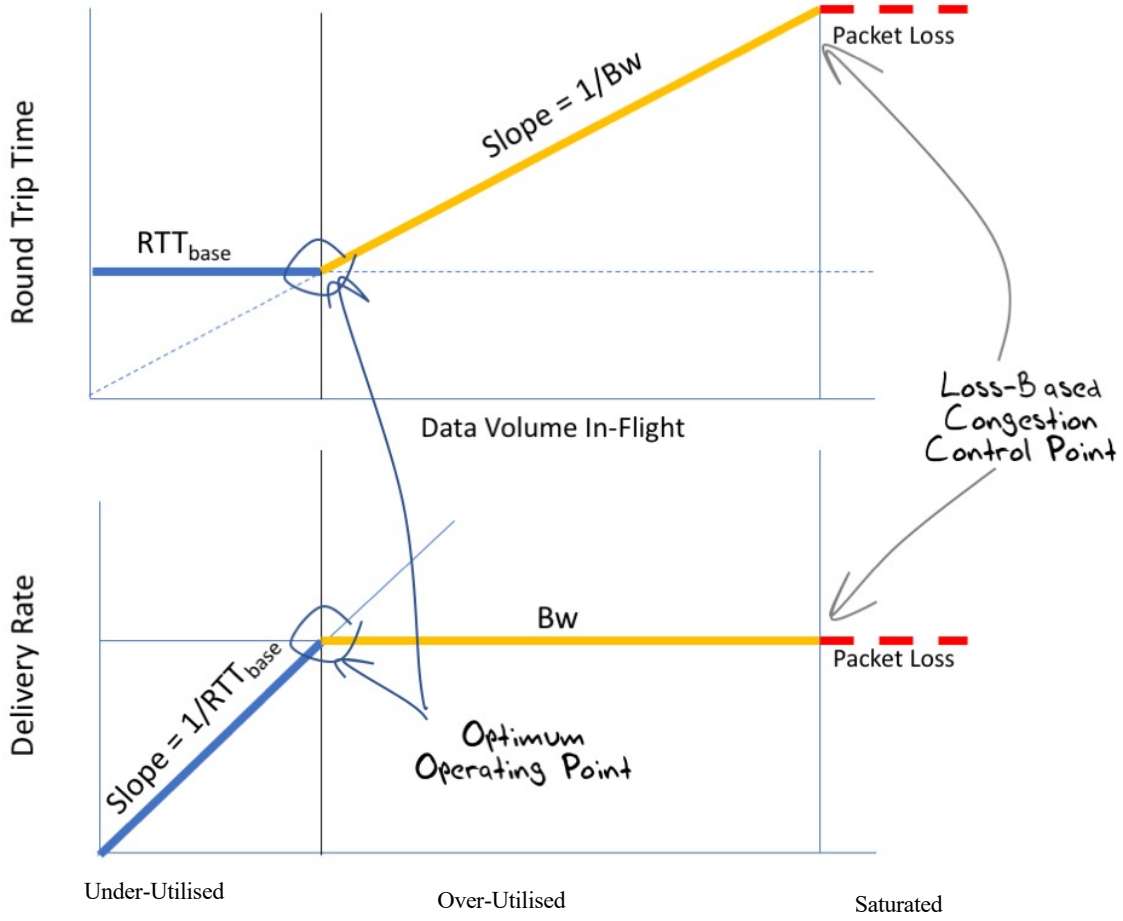


Network model

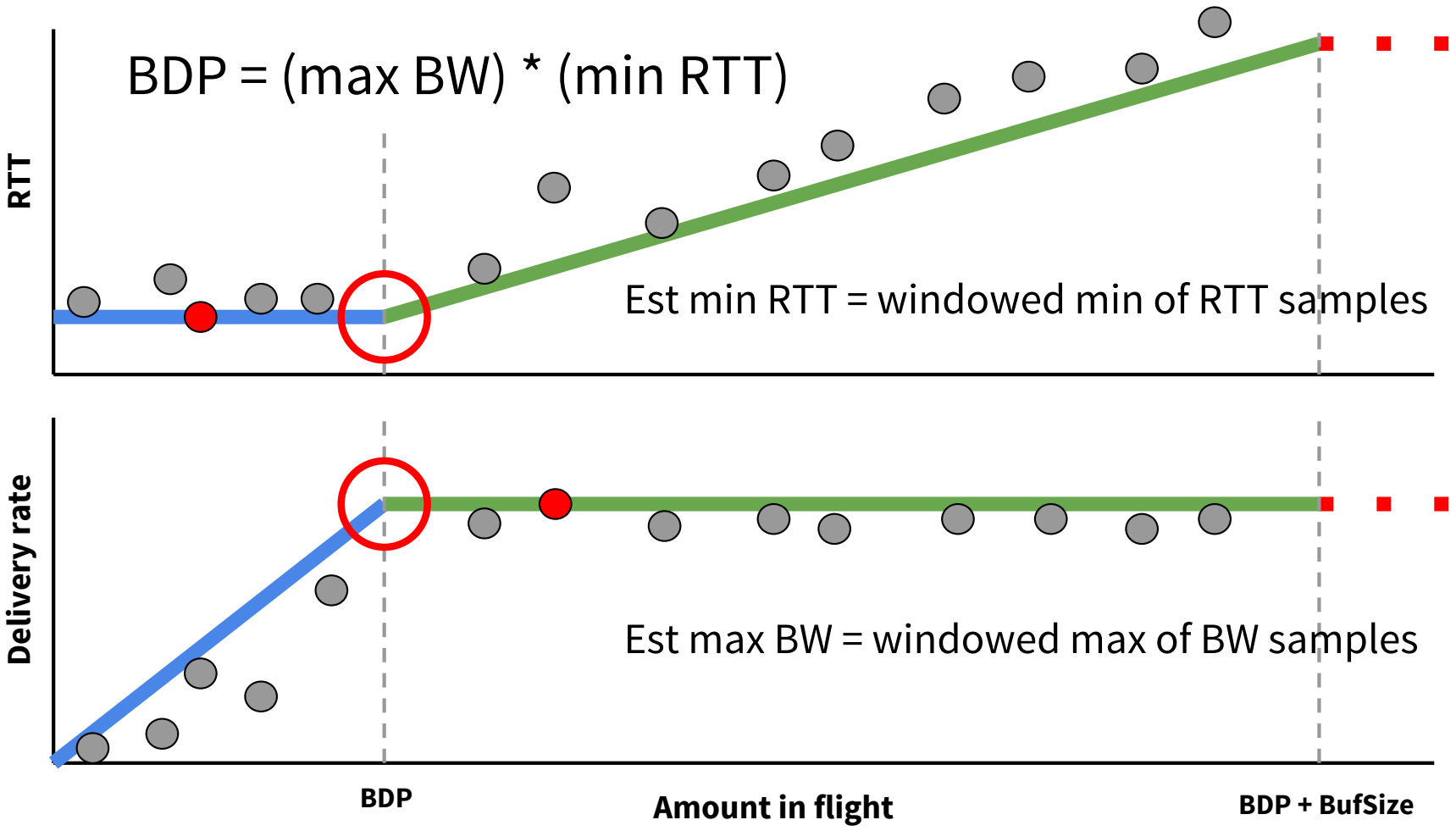



BDP
BDP+
BtlneckBufSize
app limited
bandwidth limited
buffer
limited
round-trip time
slope = 1 / BtlBw
RTprop
BtlBw
delivery rate
slope = 1 / RTprop
optimum
operating
point
is here
loss-based
congestion
control
operates here
amount inflight

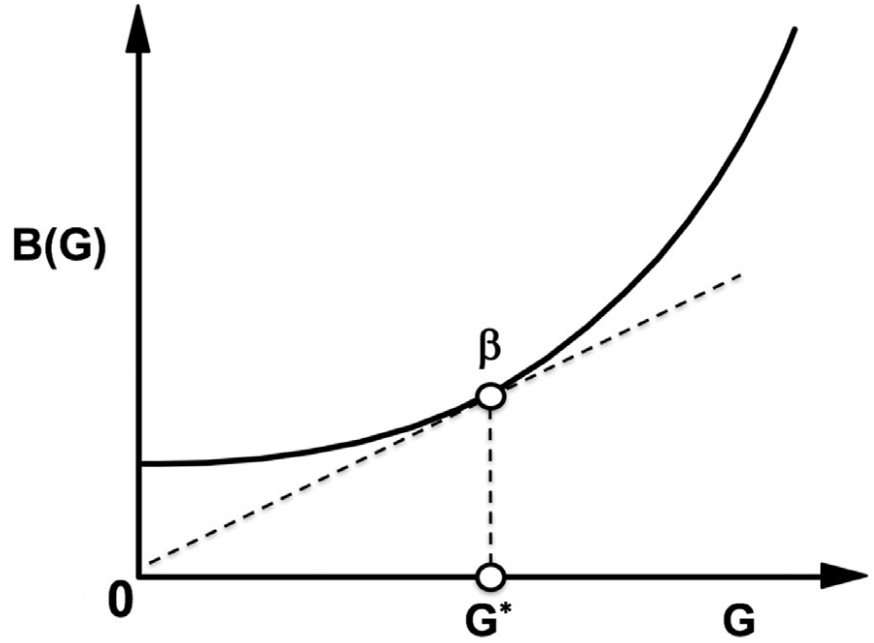
RTT and Delivery Rate with Queuing



RTT and Delivery Rate with Queuing

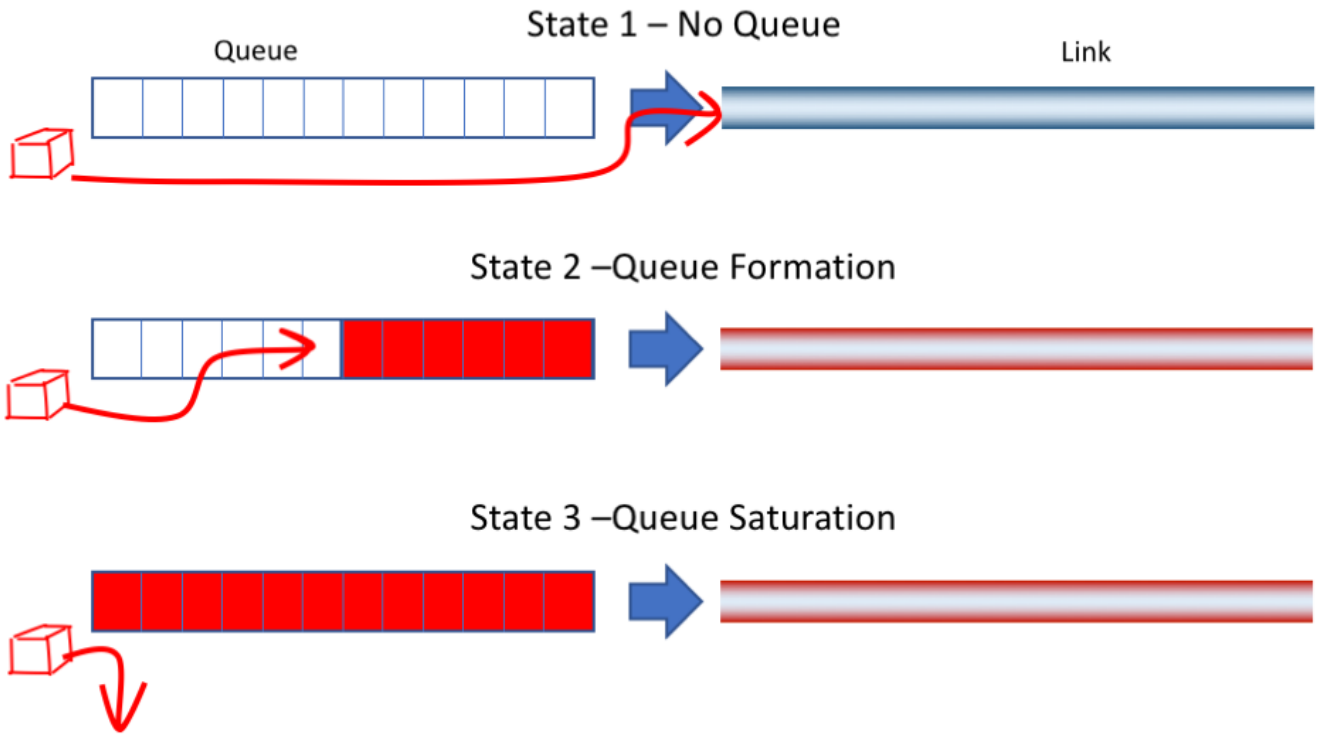


Optimal Power - Keep the Pipe Just Full, but No Fuller



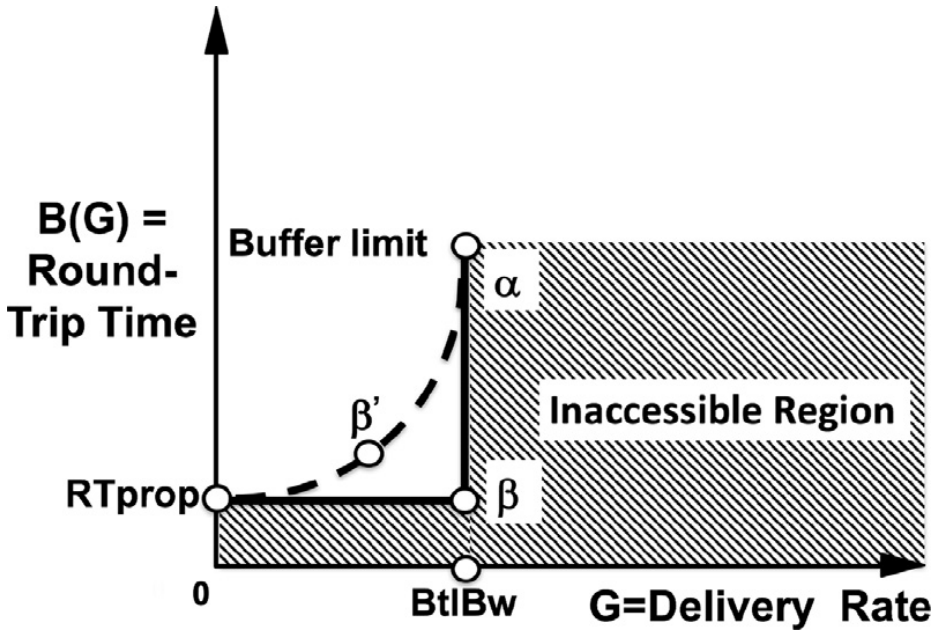
- B – Bad (Time), G – Good (Throughput)
- Power:
  - P(G) = G/B(G)
- G* which maximizes P(G)
  - line with minimal slope (slope – 1/P)
- Optimal number in system N* = 1

Optimal queues



- Three states of a queue
- Optimal point for a TCP session is the onset of buffering, or the state just prior to the transition from the first to the second state
- N* = 1

Optimal Power – TCP BBR



- G = Throughput and B(G) = Round-trip Time
- Deterministic case - optimal point β: G = BtlBW and B(G) = RTprop
- Stochastic case - optimal point β'
- At maximal Power point: minimal Round-trip Time with the maximum Throughput

BBR objectives

- Track the windowed maximum bandwidth and the minimum round-trip time on each ACK that gets returned to the source end of the link, to control the sending rate based on the model;
- sequentially probe for the maximum bandwidth and minimum round-trip times to feed the model samples;
- seek high throughput with small queues;
- approach the maximum achievable throughput for random losses less that 15%; and
- maintain small bounded queues in- dependent of the depth of the buffers

BBR Design Principles

- Probe the path capacity only intermittently
- Probe the path capacity by increasing the sending rate for a short interval and then drop the rate to drain the queue:
  - If the RTT of the probe equals the RTT of the previous state then there is available path bandwidth that could be utilised
  - If the RTT of the probe rises then the path is likely to be at the onset of queuing and no further path bandwidth is available
- Do not alter the path bandwidth estimate in response to packet loss
- Pace the sending packets to avoid the need for network buffer rate adaptation

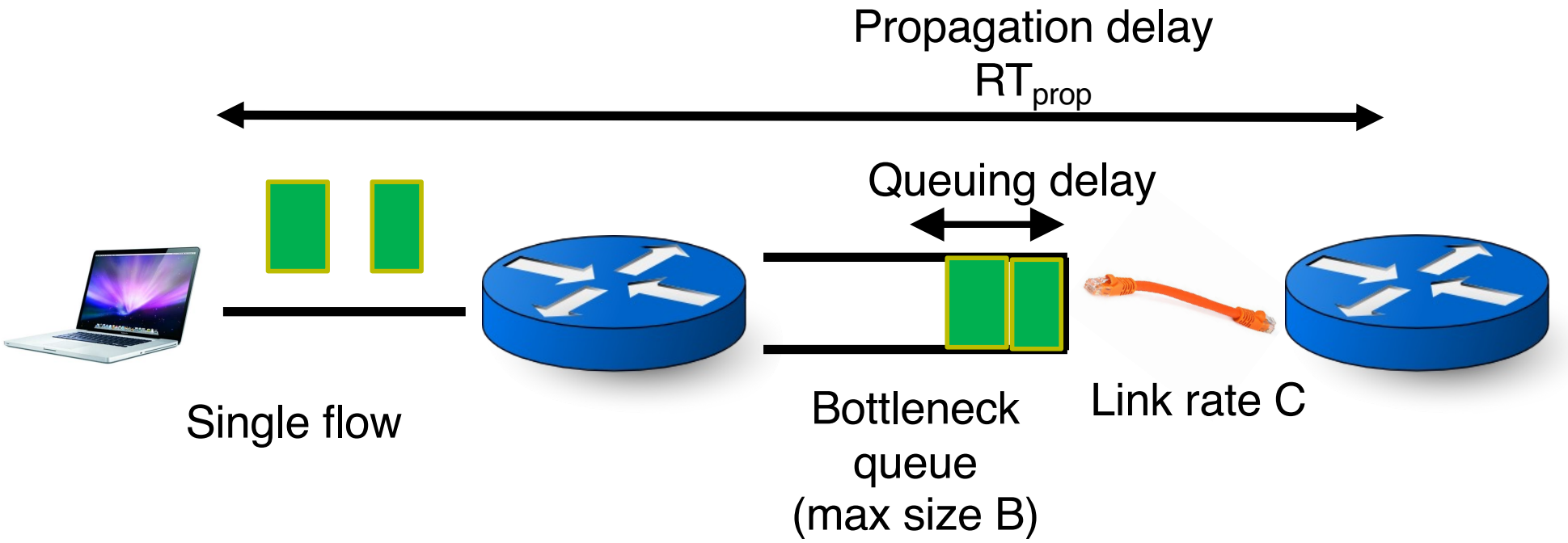
BBR Operation

- Source views network as a single bottleneck link
- Estimates RT_{prop} by taking min over the last 10 sec
- Estimates bottleneck rate (bandwidth) b_r = max of delivery rate over last 10 RTTs; delivery rate = amount of acked data per Δt
- Send data at rate

$$b_r \times c(t),$$

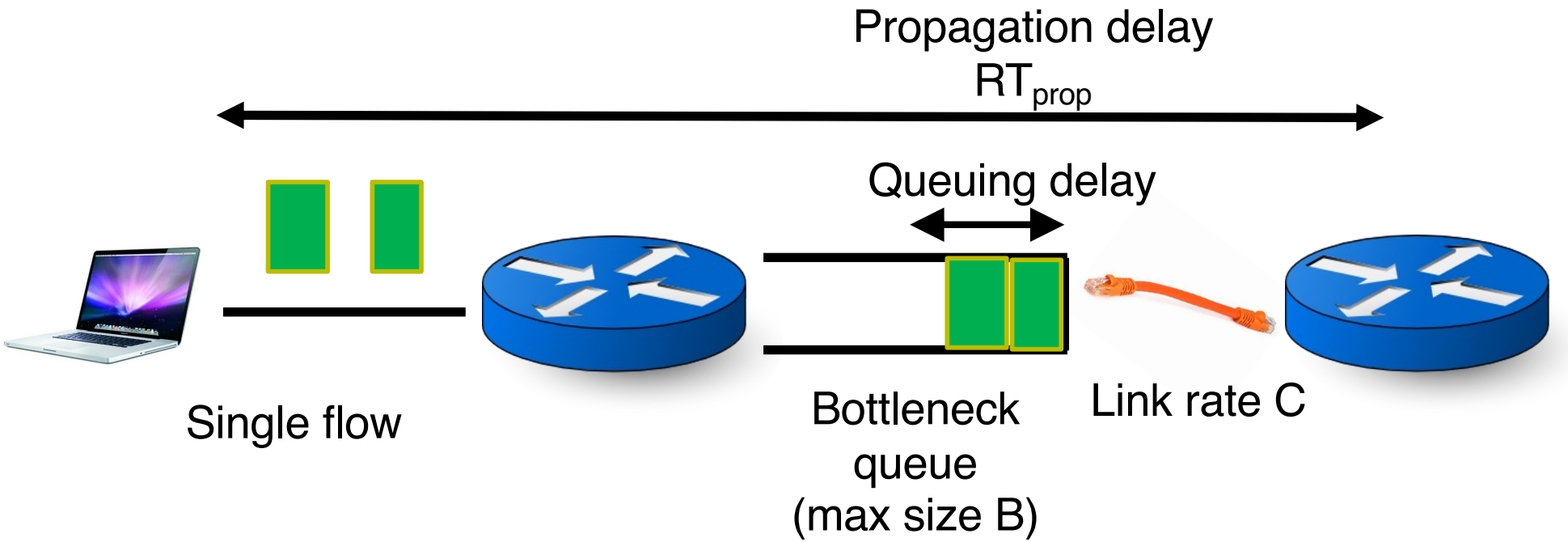
  - where i.e. c(t) is: 1.25, 0.75, 1, 1, 1… i.e., 1.25 during one RTT, then 0.75 during one RTT, then 1 during 6 RTTs ("probe bandwidth" followed by "drain excess" followed by steady state)
- Data is paced using a spacer at the source
- Max data in flight is limited to 2 x b_r x RT_{prop},
- There is also a special startup phase with exponential increase of rate

Key ideas



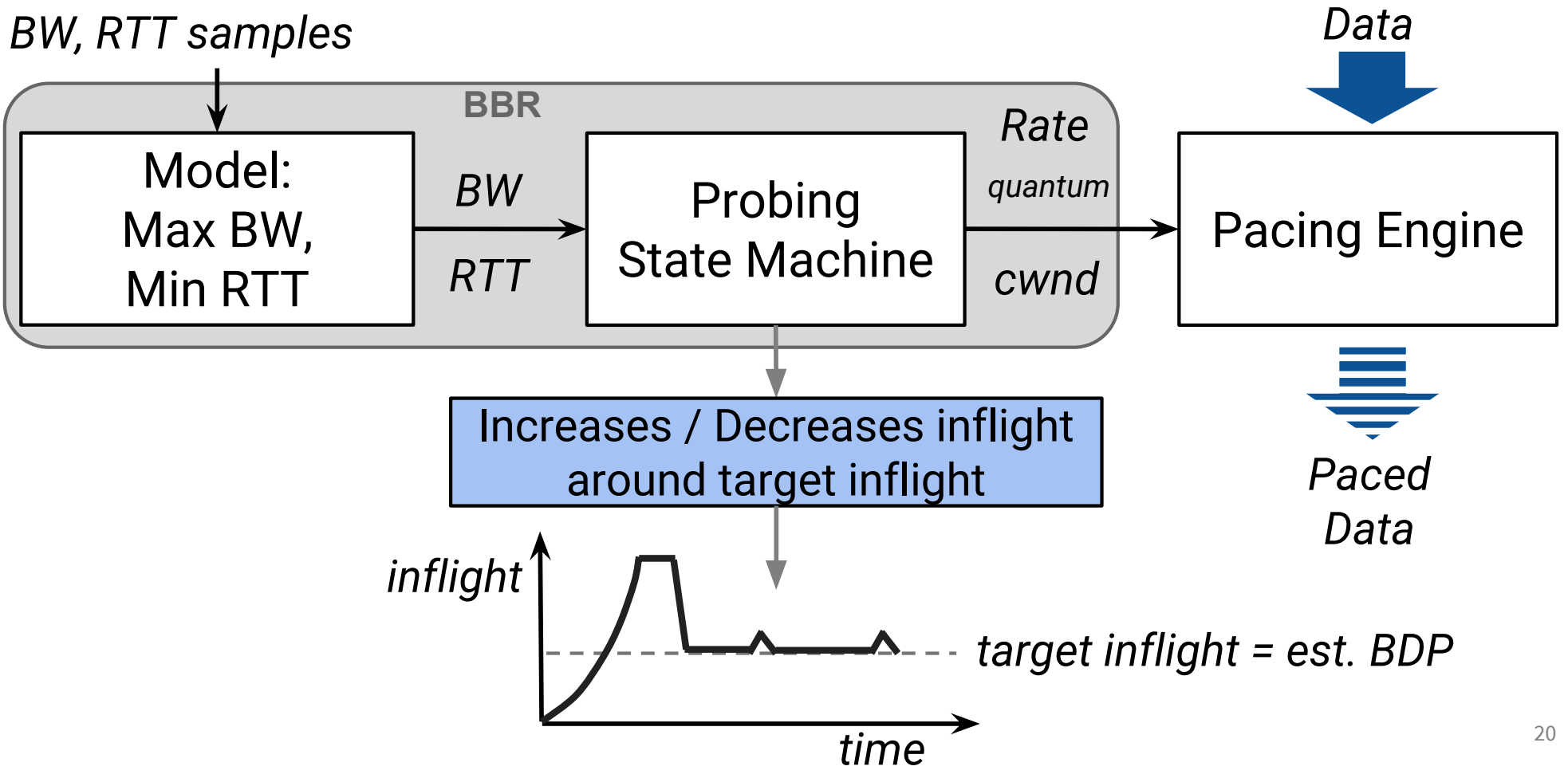
1. Estimate the bottleneck link rate C
2. Estimate the propagation delay RT_{prop}
3. Send at rate C with at most k * C * RT_{prop} packets in flight

Key ideas

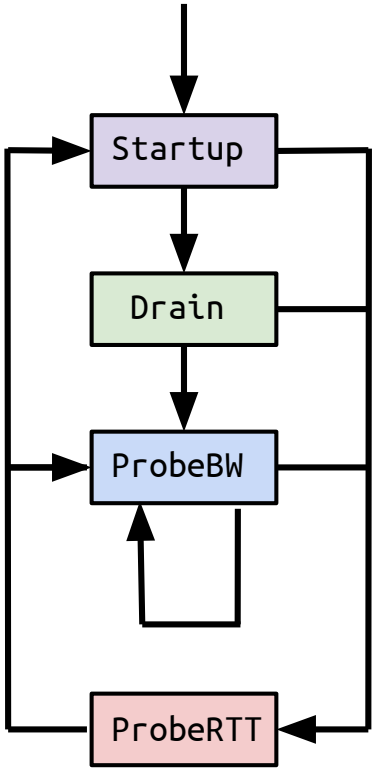


three states described above, the ‘optimal’ point for a TCP session is the onset of buffering, or the state just prior to the transition from the first to the second state.

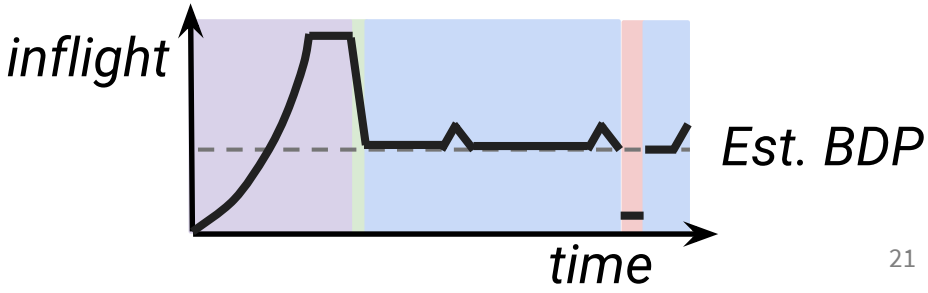
BBR – Big picture



Probing state machine



- State machine for 2-phase sequential probing:
  - 1: raise inflight to probe BtlBw, get high throughput
  - 2: lower inflight to probe RTprop, get low delay
  - At two different time scales: warm-up, steady state...
- Warm-up:
  - Startup: ramp up quickly until we estimate pipe is full
  - Drain: drain the estimated queue from the bottleneck
- Steady-state:
  - ProbeBW: cycle pacing rate to vary inflight, probe BW
  - ProbeRTT: if needed, a coordinated dip to probe RTT



21

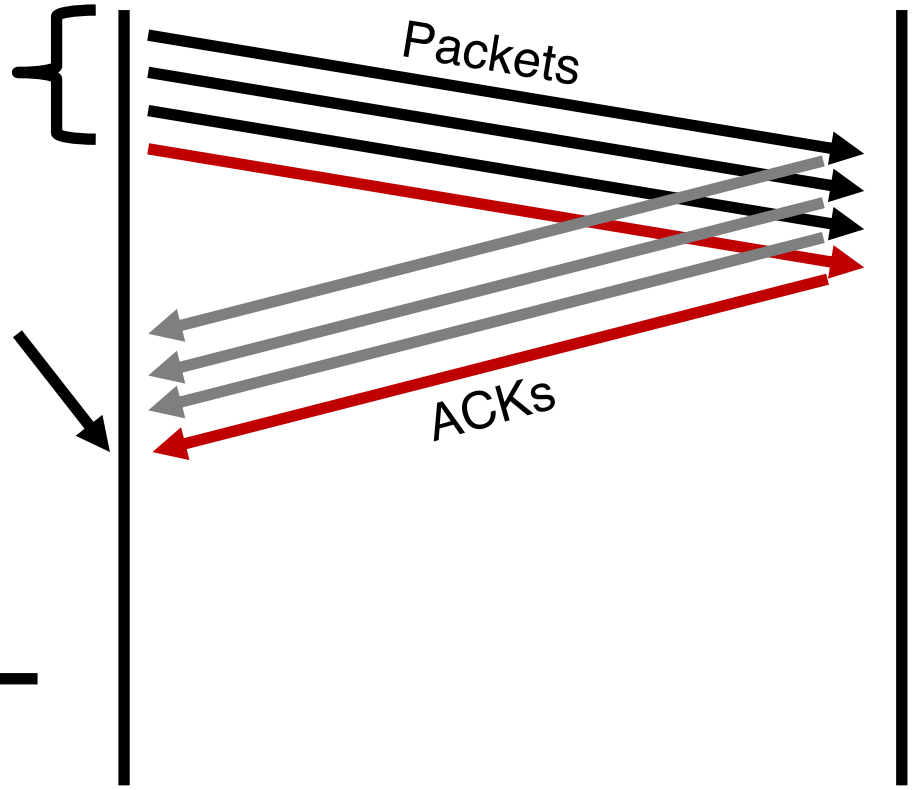
Estimating bottleneck link rate

- Data cannot be delivered to a receiver faster than the bottleneck link rate
- Measure the data delivery rate
  - And use the maximum value over the recent past
  - Important: measurements time out after a certain period
  - Occasionally send higher (PROBE_BW cycling) to see if changed
- Q: how would you measure delivery rate at the receiver?
- Q: how would you measure delivery rate at the sender?

Measuring delivery rate at sender

Data that is unACKed at the time of transmitting packet

Normal case: All that data (and only that data) is ACKed by this point



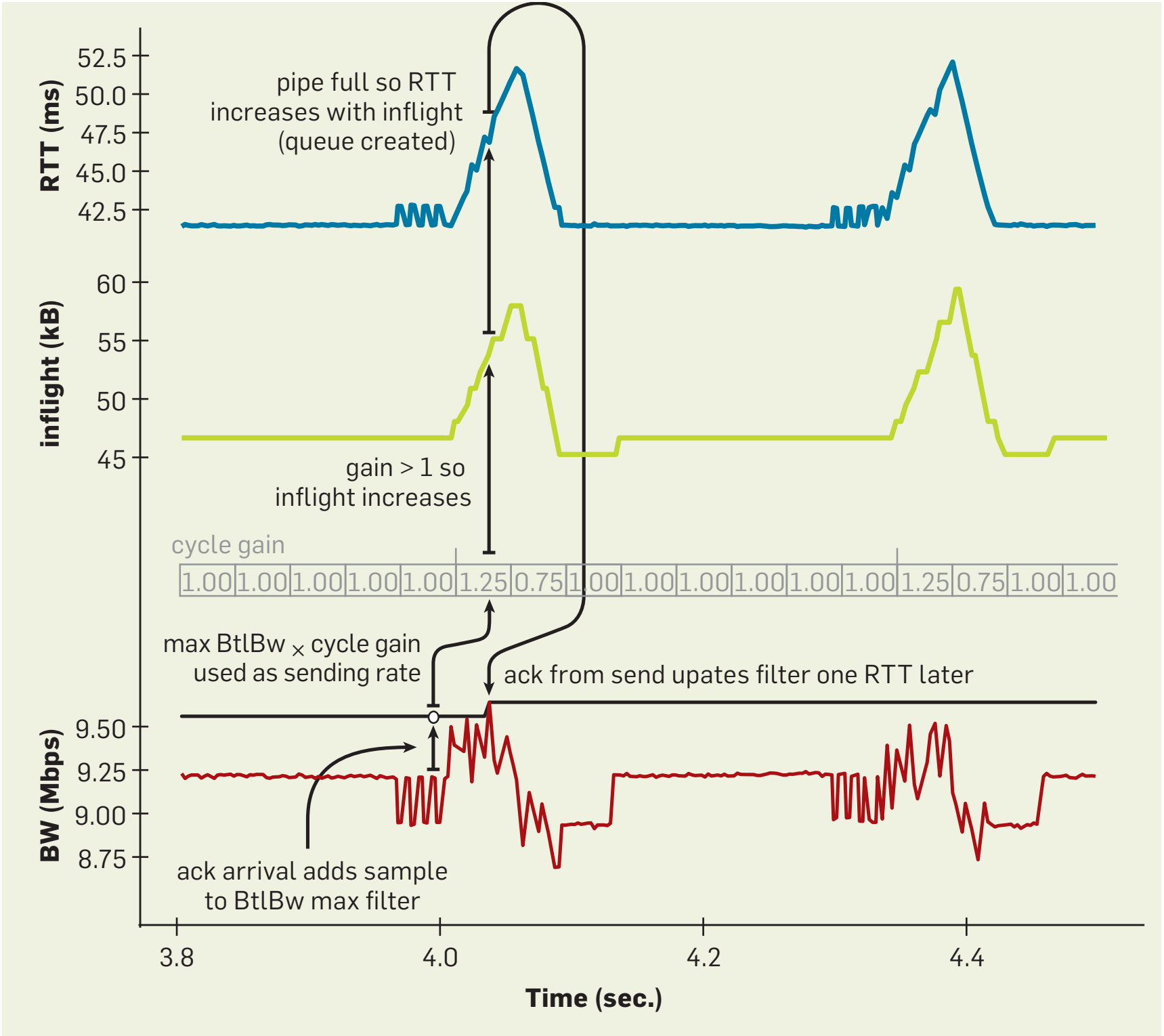
$$\frac{\text{unACKed data at pkt transmit time}}{\text{Round trip time between pkt-ACK}}$$

Estimating RT_{prop}

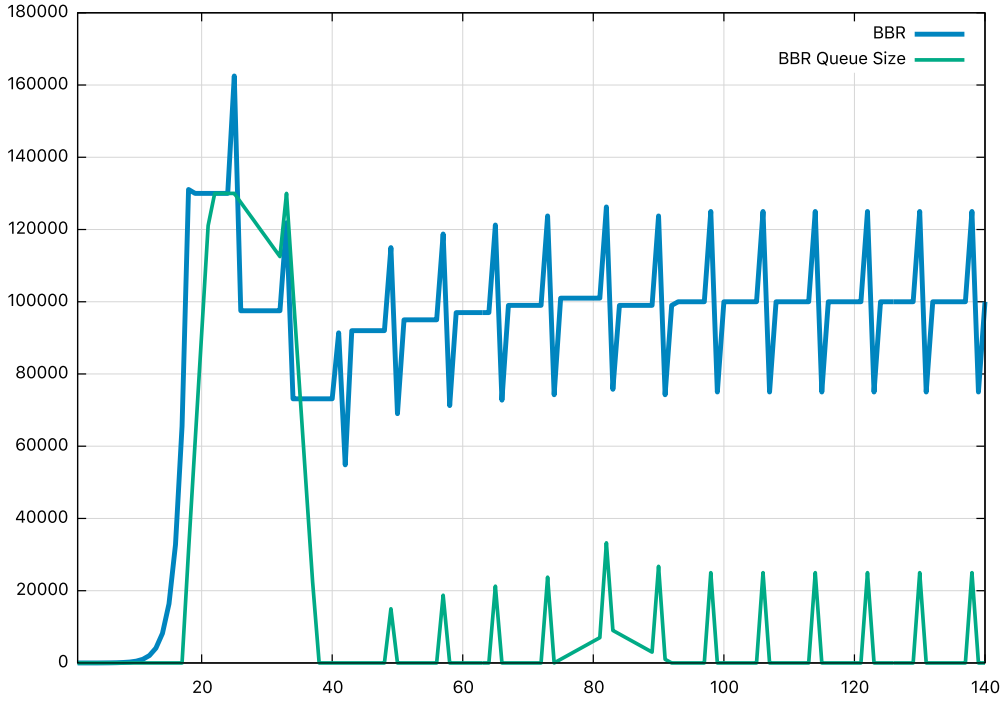
- Use the minimum of the RTT values experienced so far
- If you're sending at high rate, it is difficult to see the true RT_{prop} of the path
- Occasionally send just a few packets in an RTT to measure RT_{prop} (PROBE_RTT cycling)
- Also allows achieving fairness among BBR flows

BBR

- Estimate RTT min – min over a time window
- Estimate max BW:
- *deliveryRate* = *Δdelivered/Δt*
- max BW = max($deliveryRate_t$) over a time window


RTT (ms)
52.5
50.0
47.5
45.0
42.5
pipe full so RTT
increases with inflight
(queue created)
inflight (kB)
60
55
50
45
gain > 1 so
inflight increases
cycle gain
1.00 1.00 1.00 1.00 1.00 1.25 0.75 1.00 1.00 1.00 1.00 1.00 1.00 1.25 0.75 1.00 1.00
max BtlBw × cycle gain
used as sending rate
ack from send upates filter one RTT later
BW (Mbps)
9.50
9.25
9.00
8.75
ack arrival adds sample
to BtlBw max filter
3.8
4.0
4.2
4.4
Time (sec.)

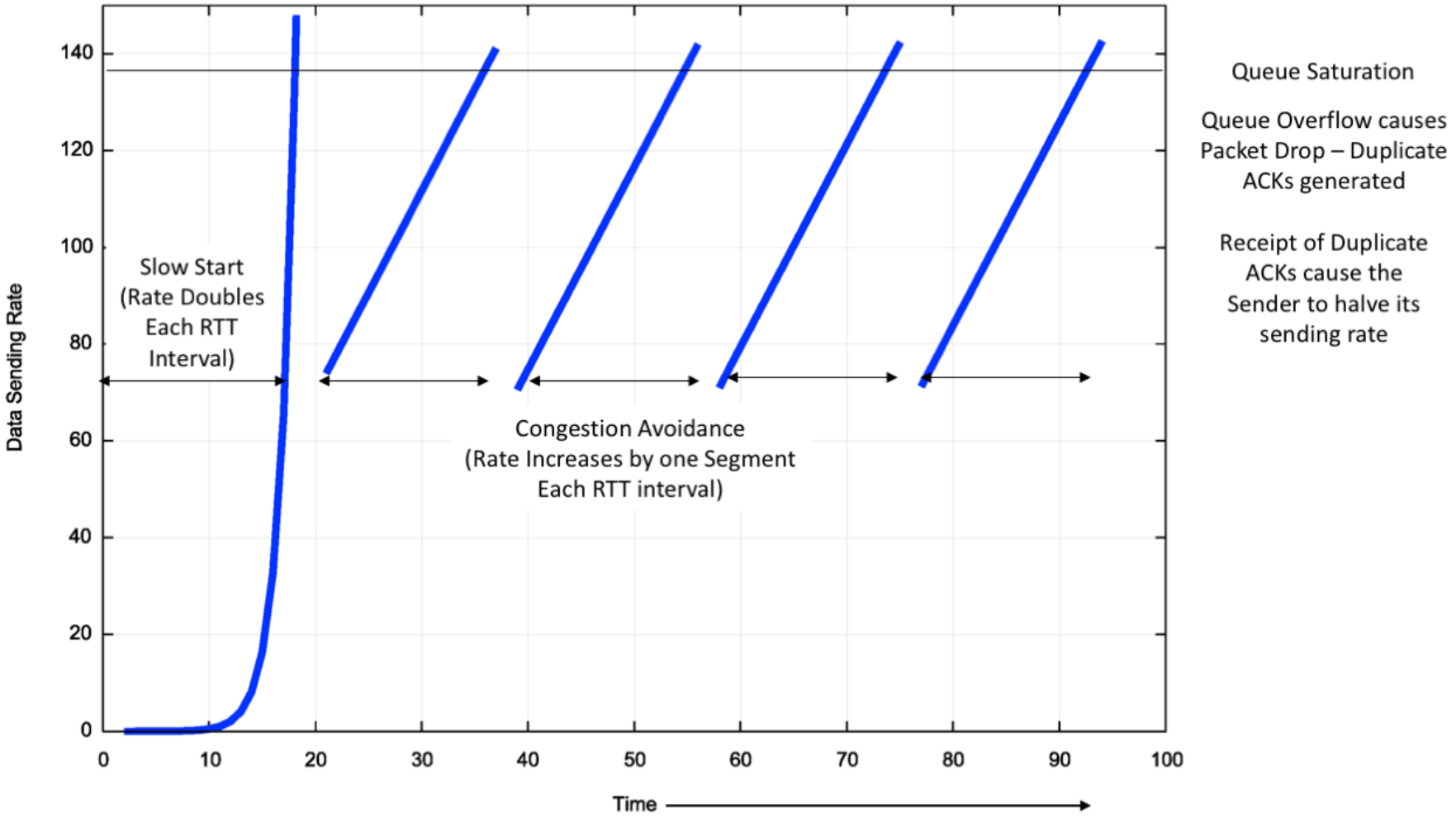
Idealised BBR profile



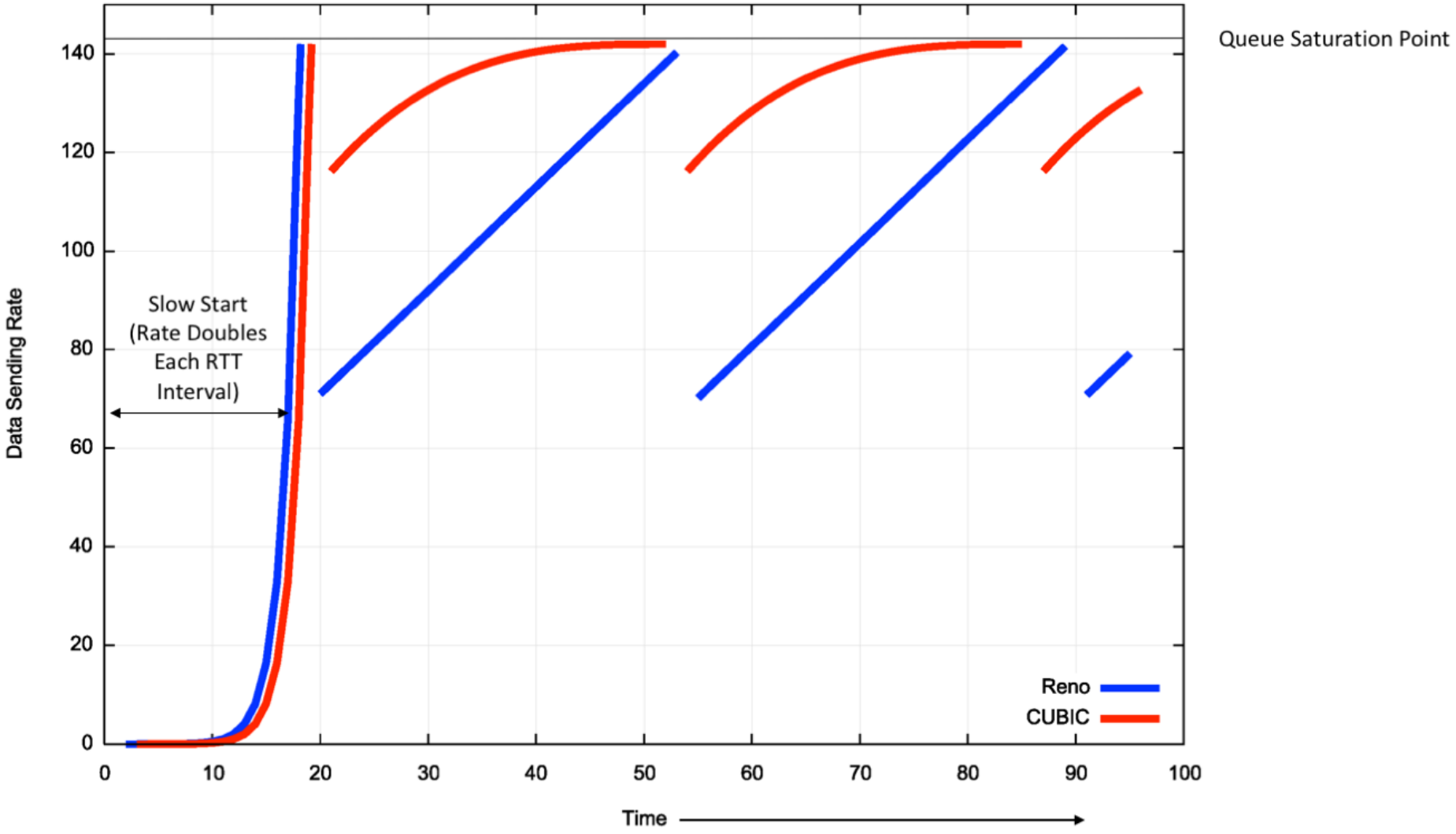
sending rate

network queues

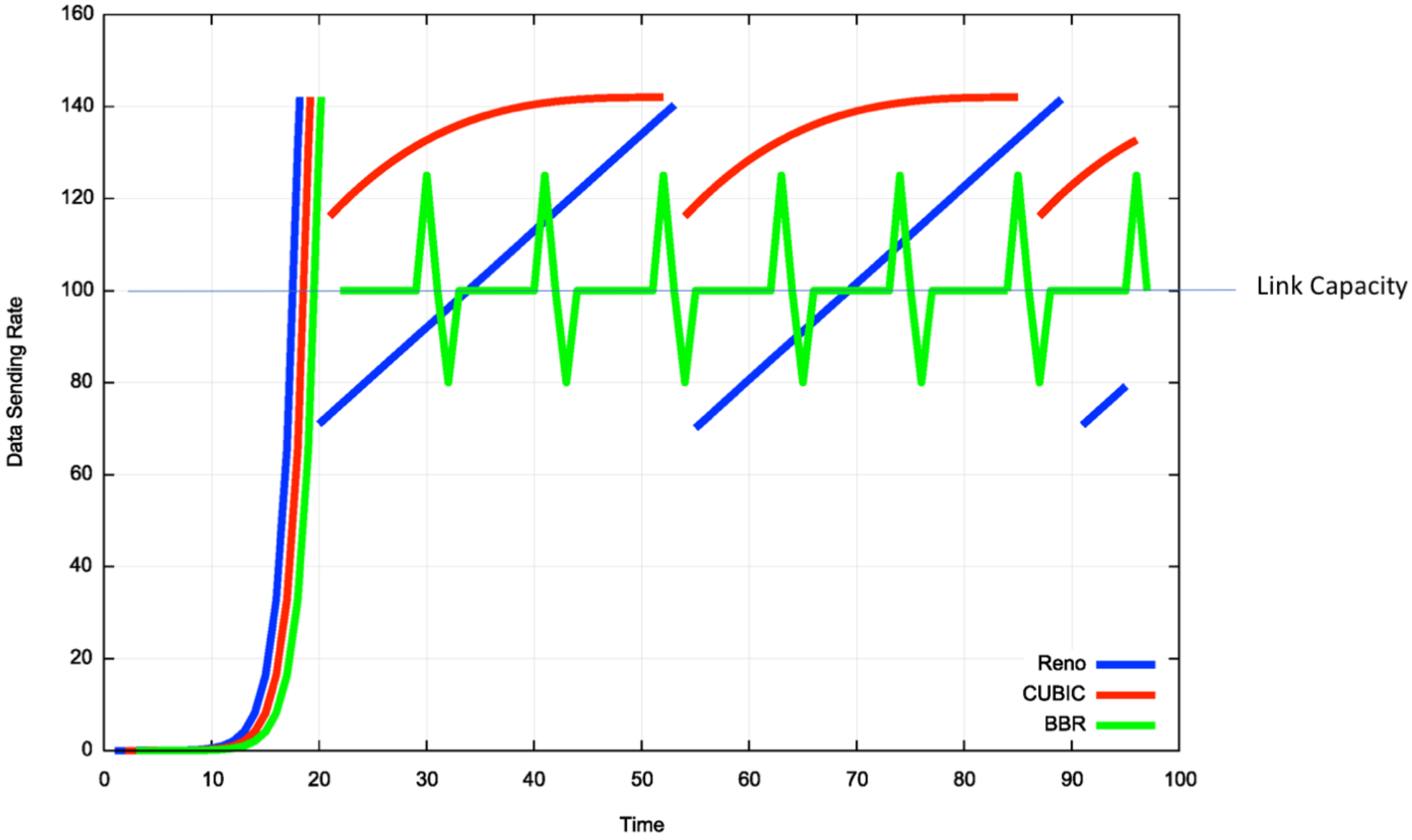
TCP Reno profile



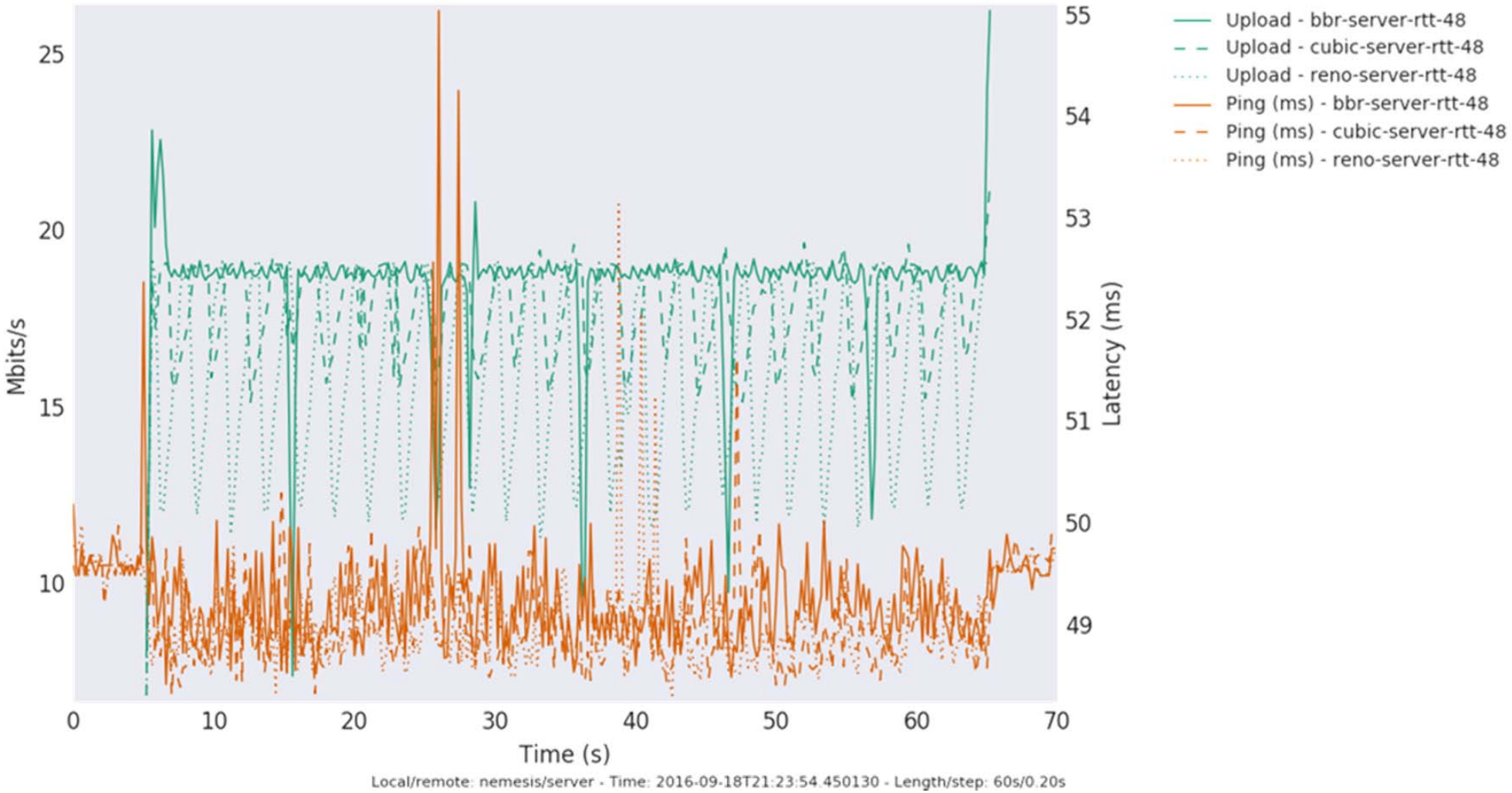
TCP Reno and Cubic profiles



TCP Reno, Cubic, and BBR profiles



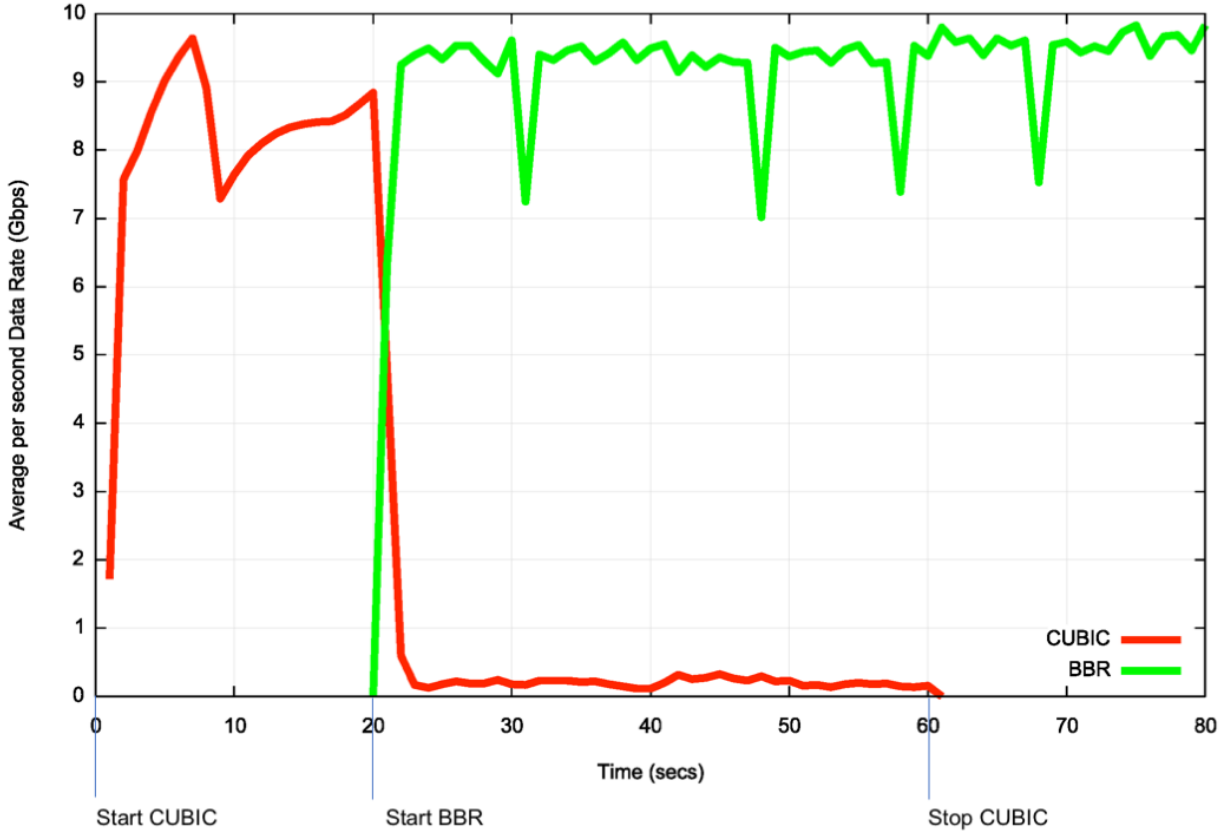
BBR performance



Properties of BBR

- Prevents unnecessary packet drops at routers → less retransmissions → improvement in the “GOODPUT”
- Avoids timeouts by getting faster notification to end hosts
- Less time to identify congestion
  - ✓ Non-ECN flows infer congestion from 3 duplicate ACKs or even worse from timeouts as opposed to ECN flows that get congestion notification in the first ACK
- Fewer retransmissions also means less traffic on the network

Cubic vs BBR over a 12ms RTT 10G circuit

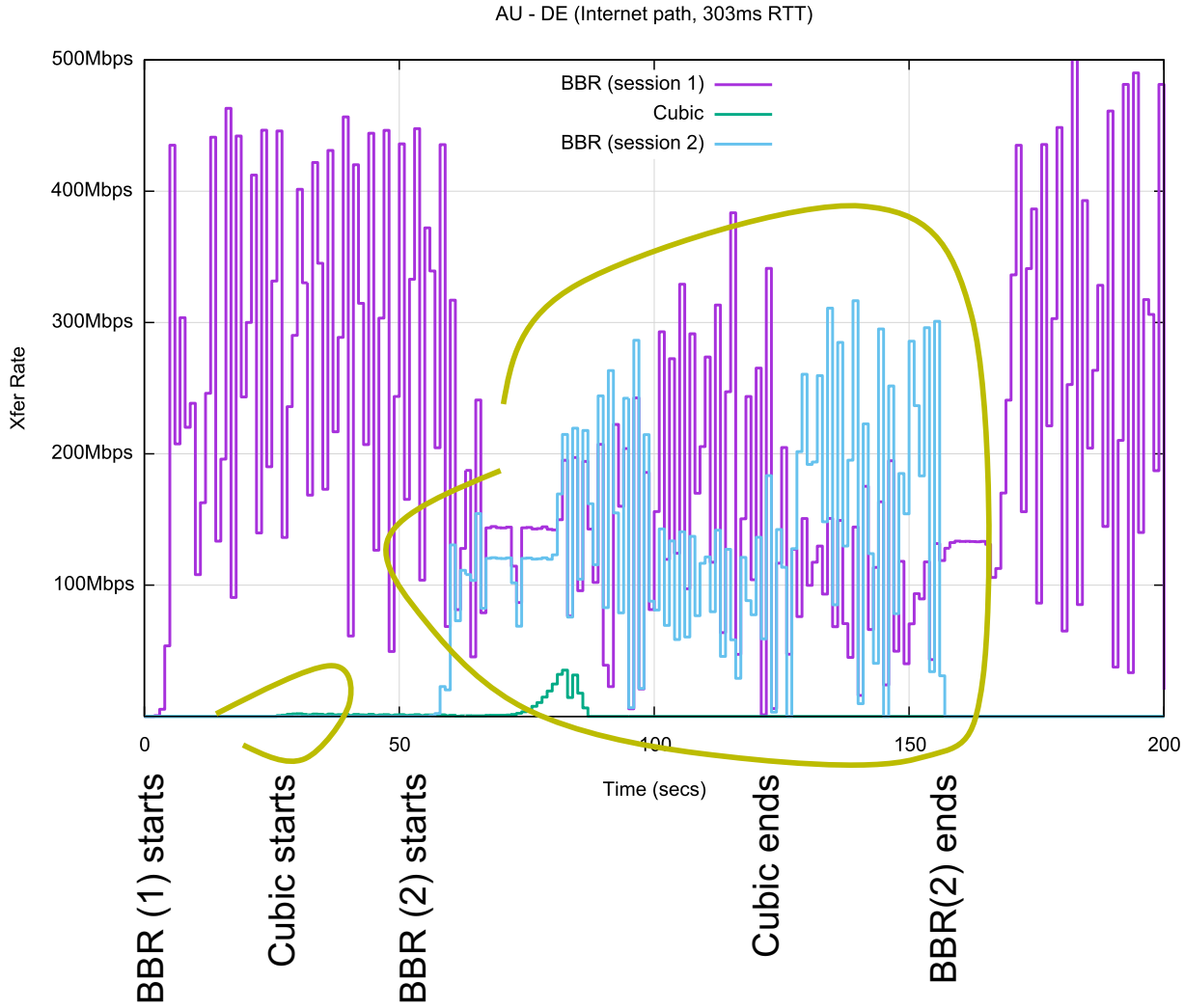


BBR vs Cubic

The Internet is capable of offering a 400Mbps capacity path on demand!

In this case BBR is apparently operating with filled queues, and this crowds out CUBIC

BBR does not compete well with itself, and the two sessions oscillate in getting the majority share of available path capacity



Why use BBR?

- Because it achieves
- Its incredibly efficient
- It makes minimal demands on network buffer capacity
- It is fast!

Why not use BBR?

- Because it **over achieves**!
- The classic question for many Internet technologies is scaling – “what if everyone does it?”
  - BBR is not a scalable approach
  - It works so well while it is used by just a few users, some of the time
  - But when it is active, BBR has the ability to slaughter concurrent loss-based flows
  - Which sends all the wrong signals to the TCP ecosystem
    - The loss-based flows convert to BBR to compete on equal terms
    - The network is then a BBR vs BBR environment, which is unstable

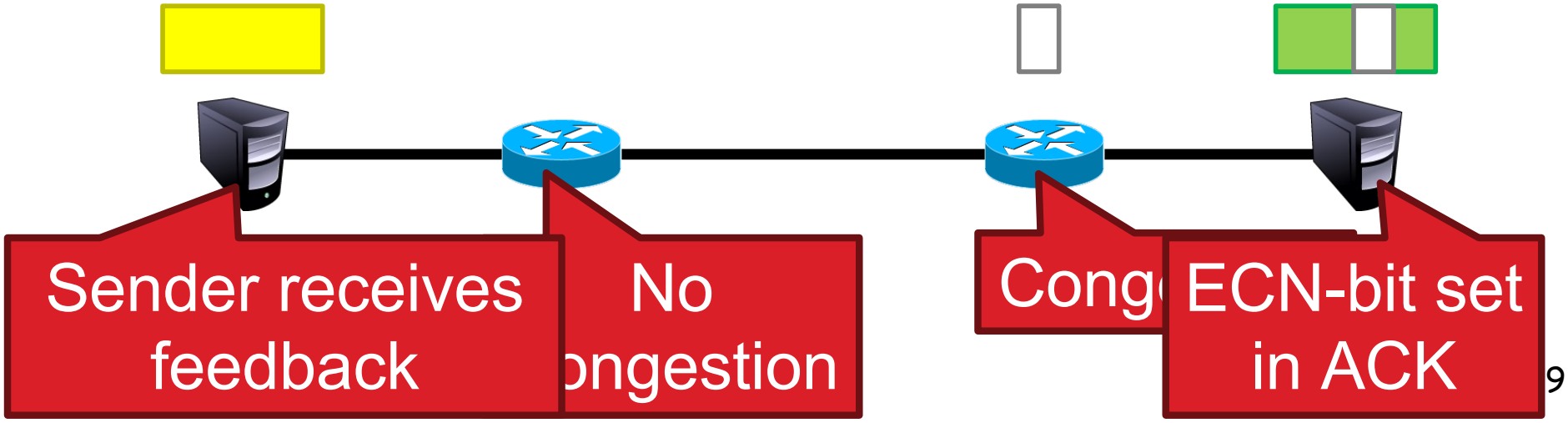
ECN

Explicit Congestion Notification

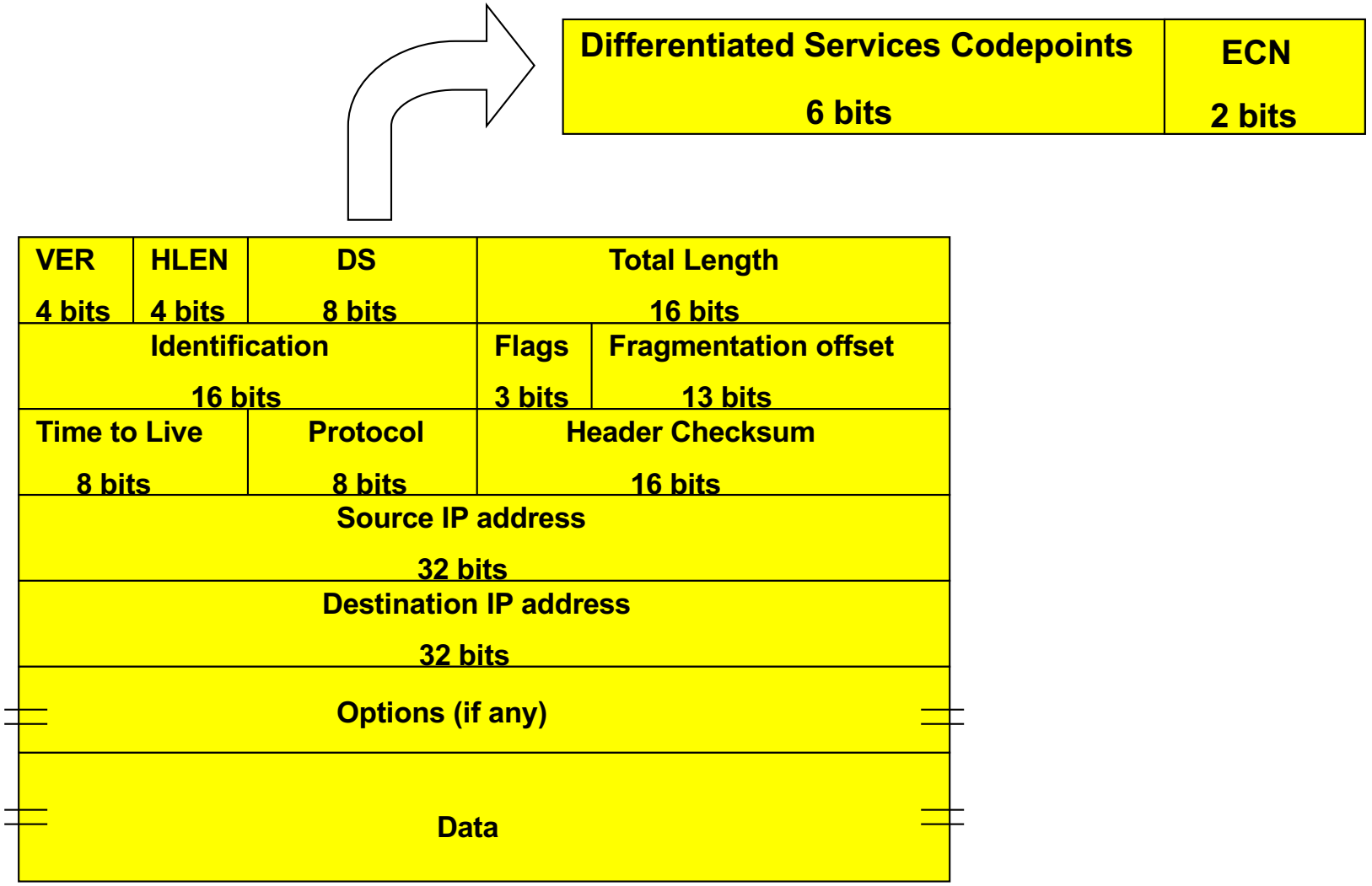
- Routers notify TCP senders/receivers about incipient congestion – without packet drops
- How?
  - ✓ Through IP and TCP headers
- TCP treats ECN signals exactly the same as when a single dropped packet is detected – but packets are not actually dropped

Explicit Congestion Notification

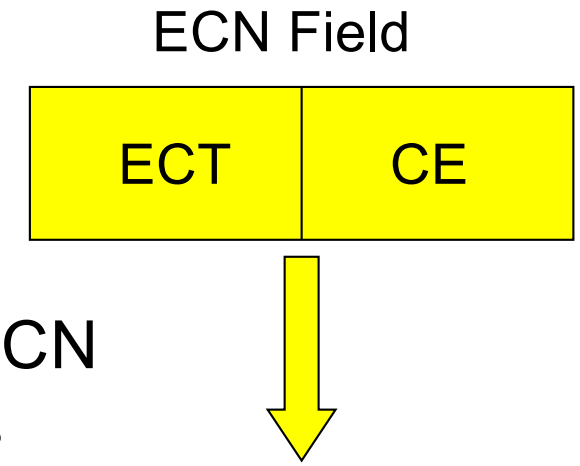
- ECN is an AQM mechanism
- Use TCP/IP headers to send ECN signals
  - Router sets ECN bit in header if there is congestion
  - Host TCP treats ECN marked packets the same as packet drops (i.e. congestion signal)
    - But no packets are dropped :)



ECN bits in IP header



ECN bits in IP header (cont'd)



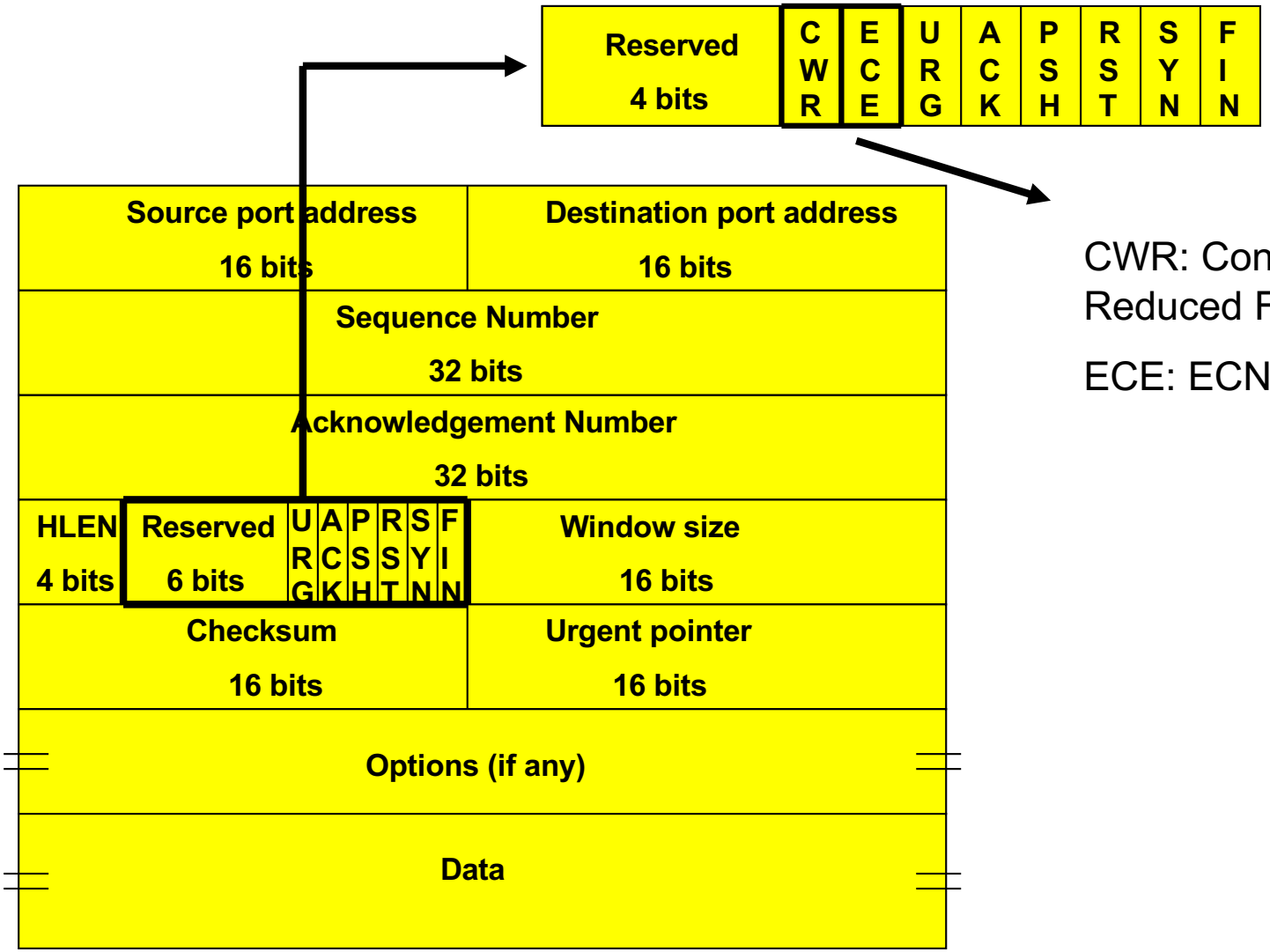
ECT: ECN Capable Transport

CE: Congestion Experienced

2 bits = 4 ECN Codepoints

ECT	CE	Names for the ECN bits
0	0	**Not-ECT** (Not ECN Capable Transport)
0	1	**ECT(1)** (ECN Capable Transport (1))
1	0	**ECT(0)** (ECN Capable Transport(0))
1	1	**CE** (Congestion Experienced)

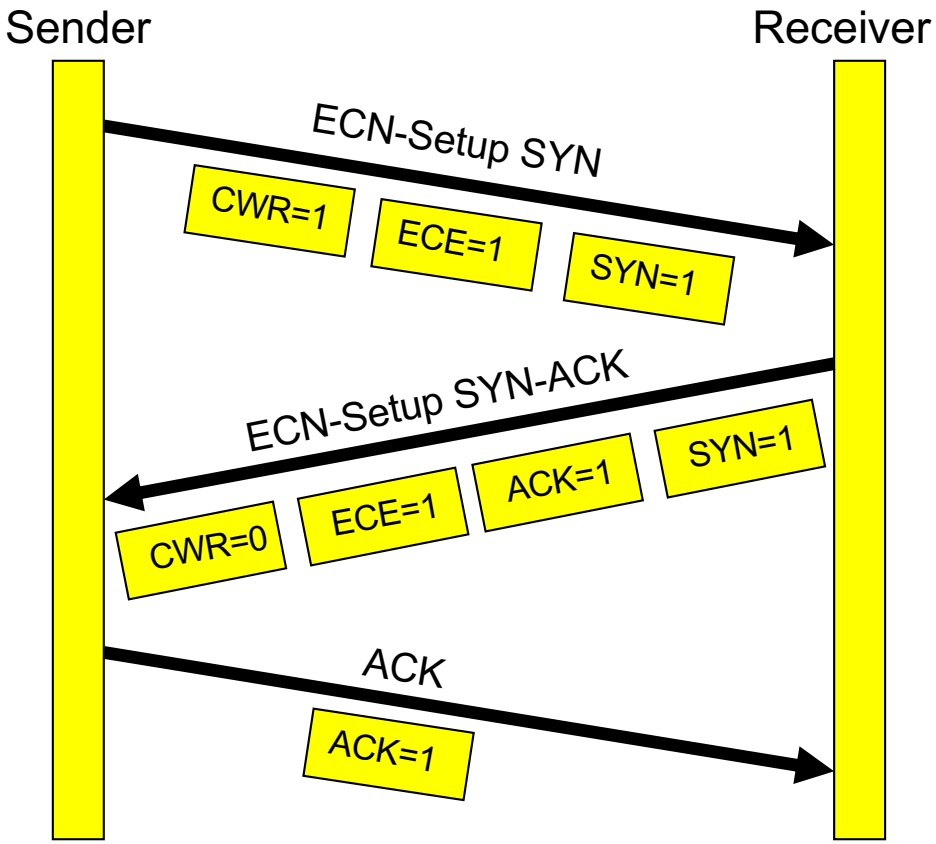
ECN bits in TCP header



CWR: Congestion Window Reduced Flag

ECE: ECN-Echo Flag

ECN negotiation between TCP end hosts



- A host must not set ECT in SYN or SYN-ACK

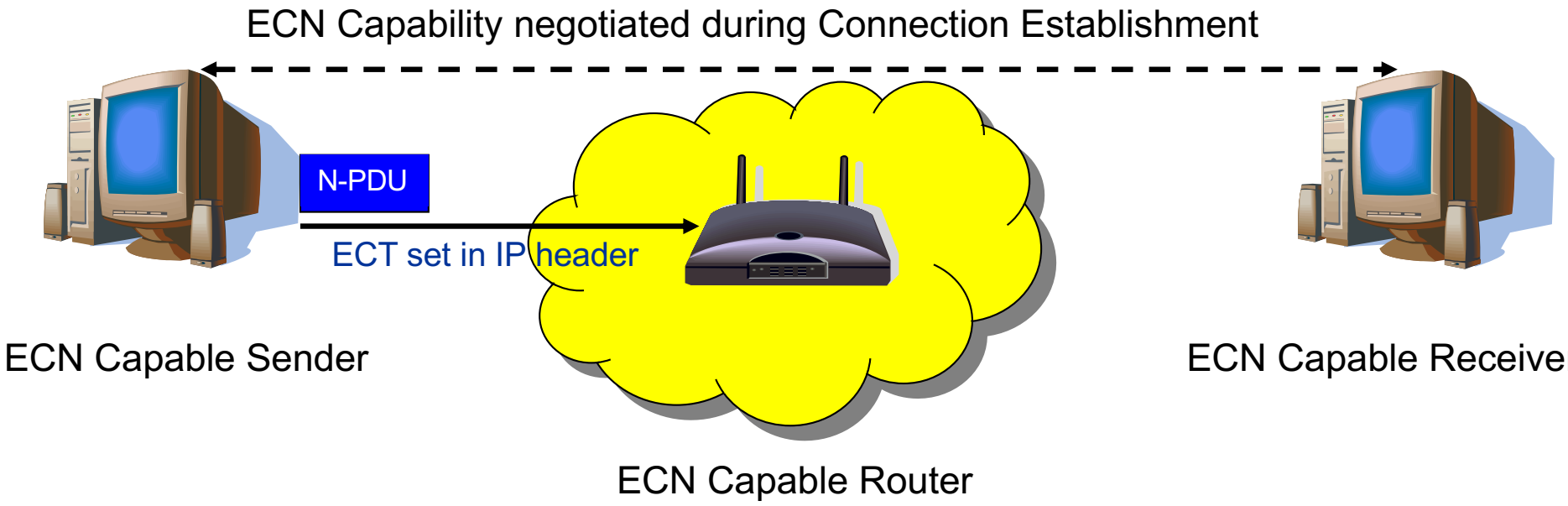
ECN Operation

- Source sends a packet using TCP
- Packet is received at congested router buffer; router marks the Congestion Experienced (CE) bit in IP header
- Receiver sees CE in received packet and set the ECN Echo (ECE) flag in the TCP header of packets sent in the reverse direction
- Packets with ECE is received by source.
- Source applies multiplicative decrease of the congestion window.

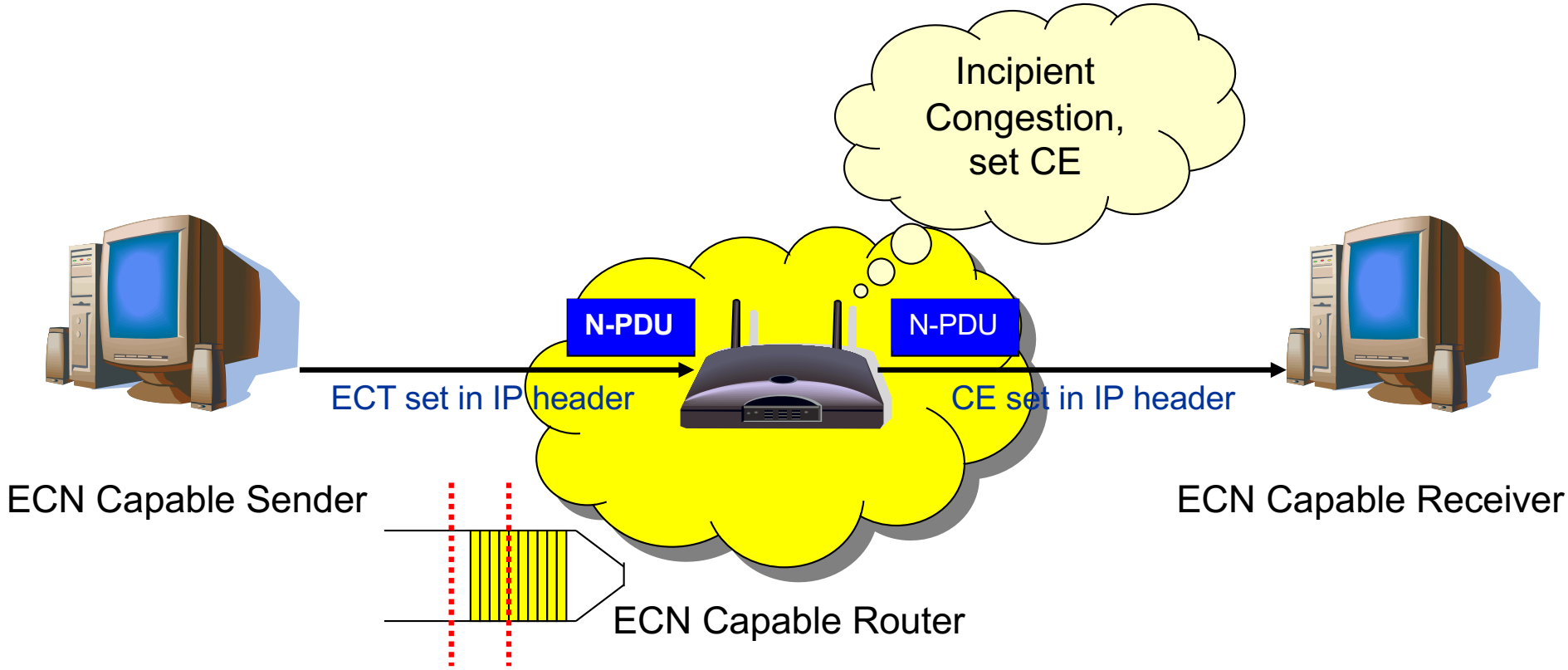
ECN

- Source sets the Congestion Window Reduced (CWR) flag in TCP header. The receiver continues to set the ECE flag until it receives a packet with CWR set.
- Multiplicative decrease is applied only once per window of data (typically, multiple packets are received with ECE set inside one window of data).

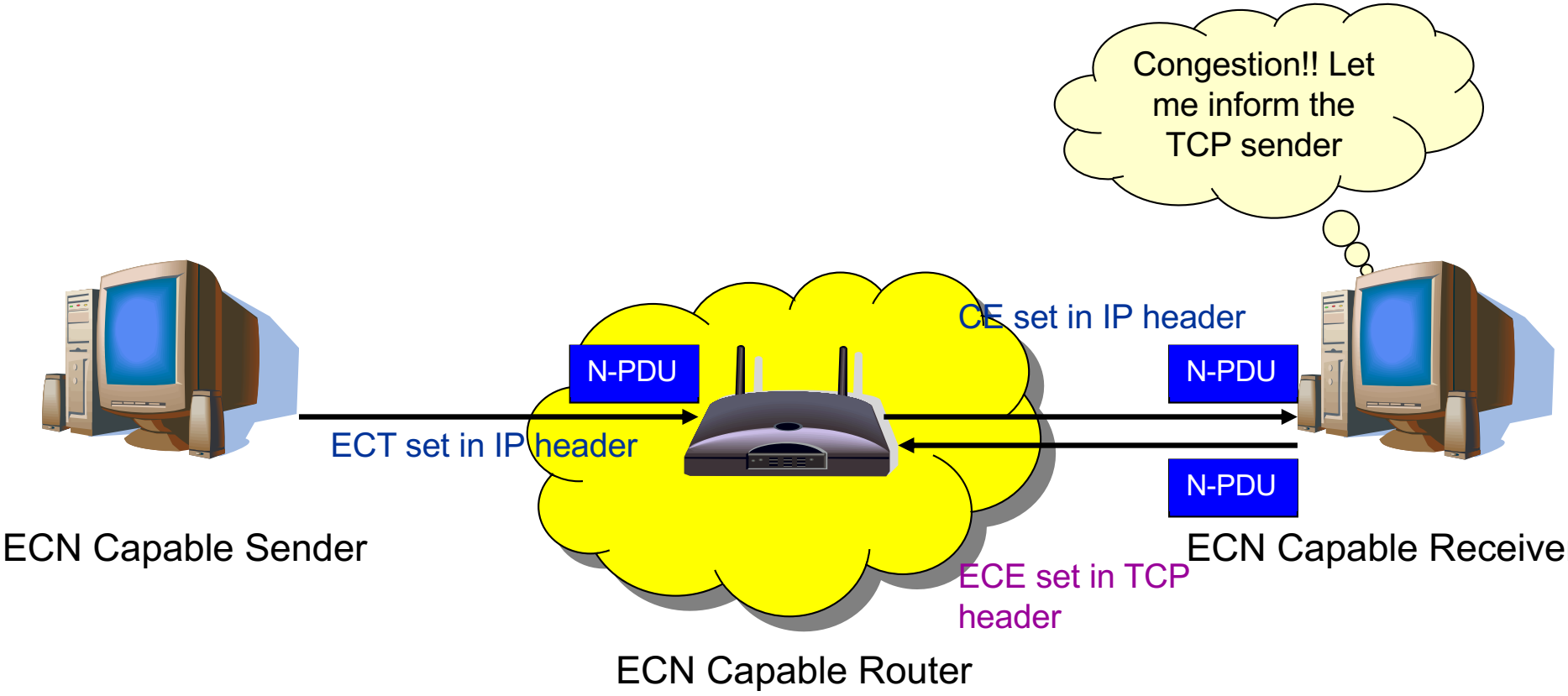
Typical sequence of events Event-1



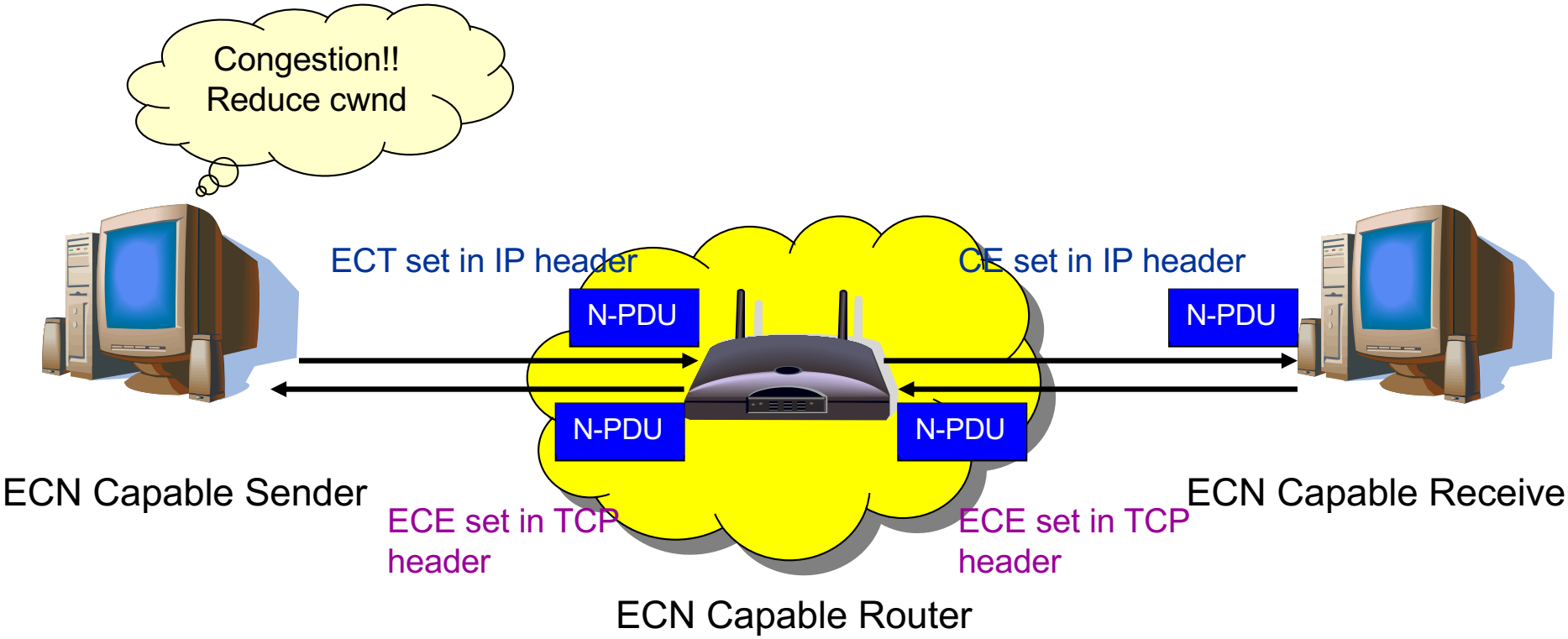
Typical sequence of events Event-2



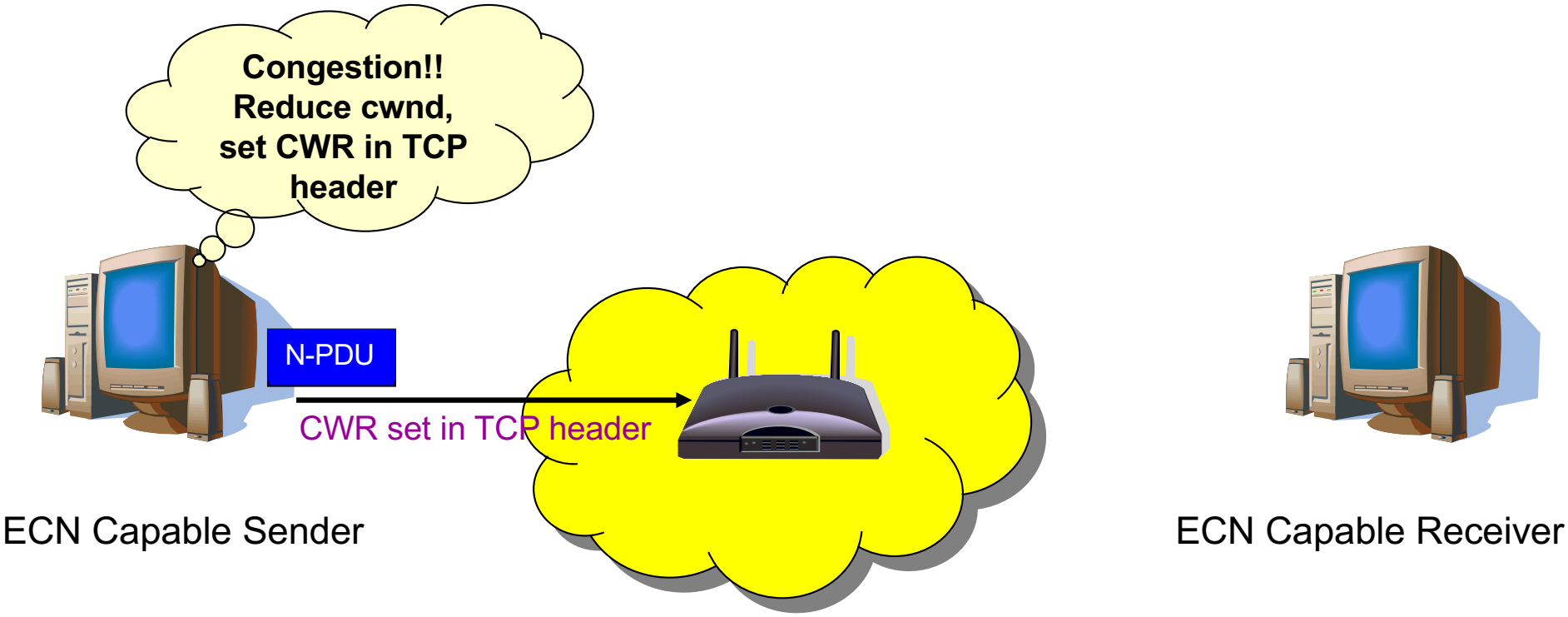
Typical sequence of events Event-3



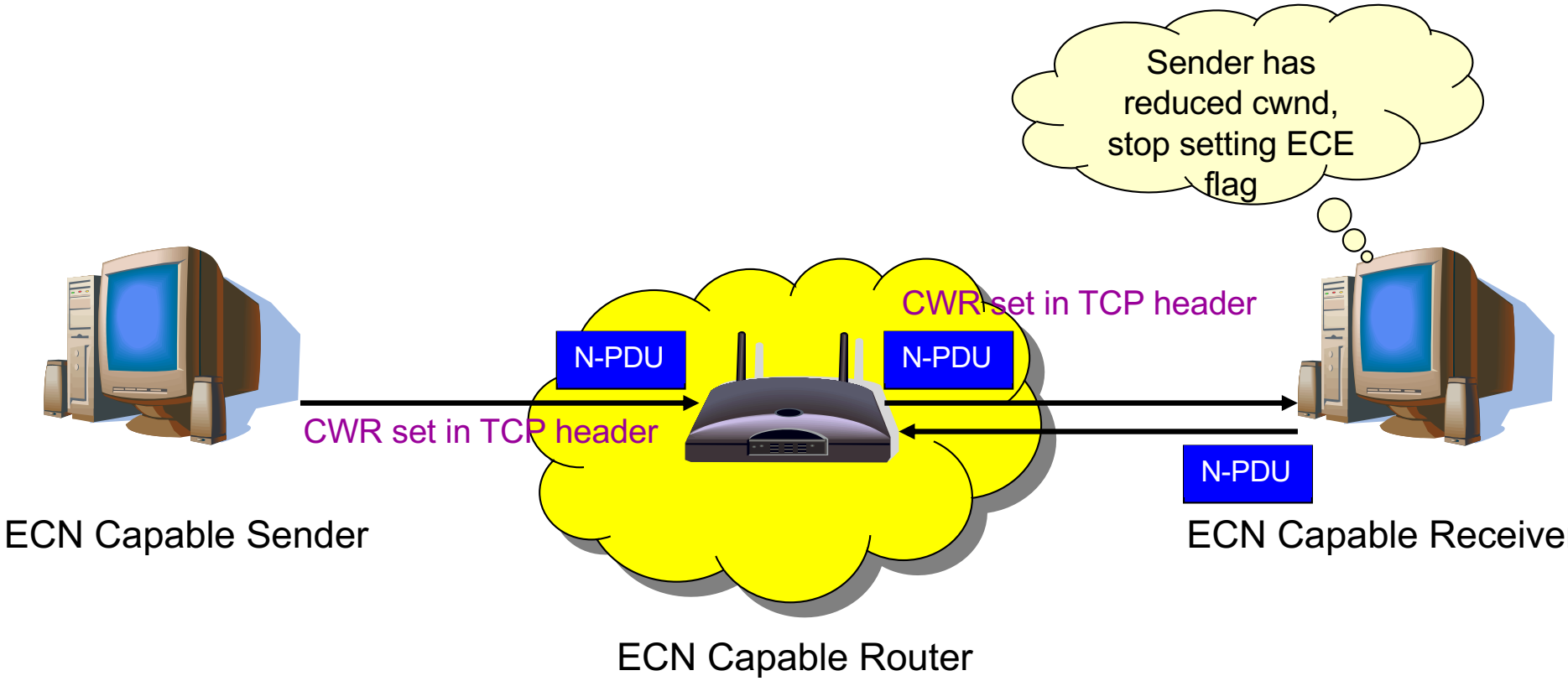
Typical sequence of events Event-4



Typical sequence of events Event-5



Typical sequence of events Event-6



Advantages of ECN

- Prevents unnecessary packet drops at routers → less retransmissions → improvement in the "GOODPUT"
- Avoids timeouts by getting faster notification to end hosts
- Less time to identify congestion
  - ✓ Non-ECN flows infer congestion from 3 duplicate ACKs or even worse from timeouts as opposed to ECN flows that get congestion notification in the first ACK
- Fewer retransmissions also means less traffic on the network

Facts to remember

- TCP performs congestion control in end-systems
  - sender increases its sending window until loss occurs, then decreases
    - additive increase (no loss)
    - multiplicative decrease (loss)
- TCP states
  - slow start, congestion avoidance, fast recovery
- Negative bias towards long round trip times
- UDP applications should behave like TCP with the same loss rate