Chapter 3
Transport Layer

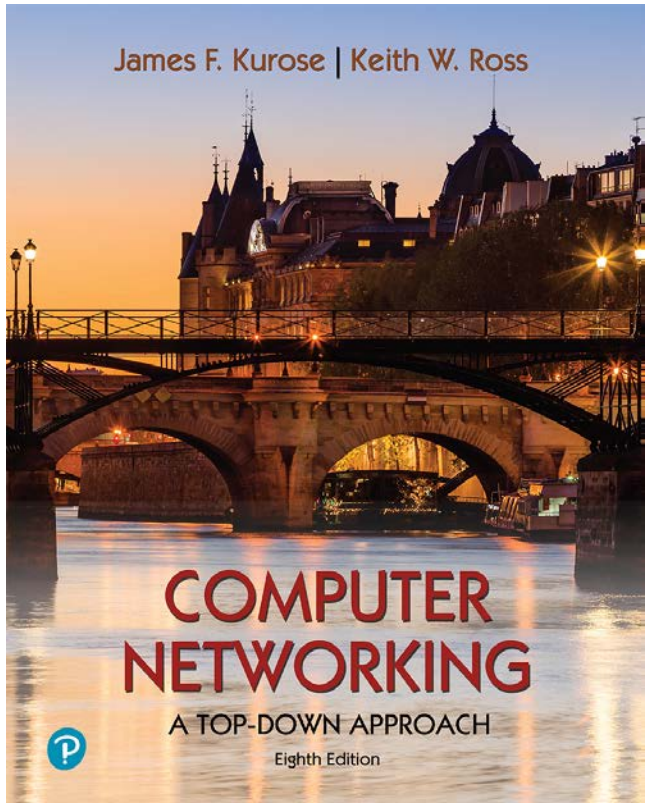
A note on the use of these PowerPoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

For a revision history, see the slide note for this page.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top-Down Approach

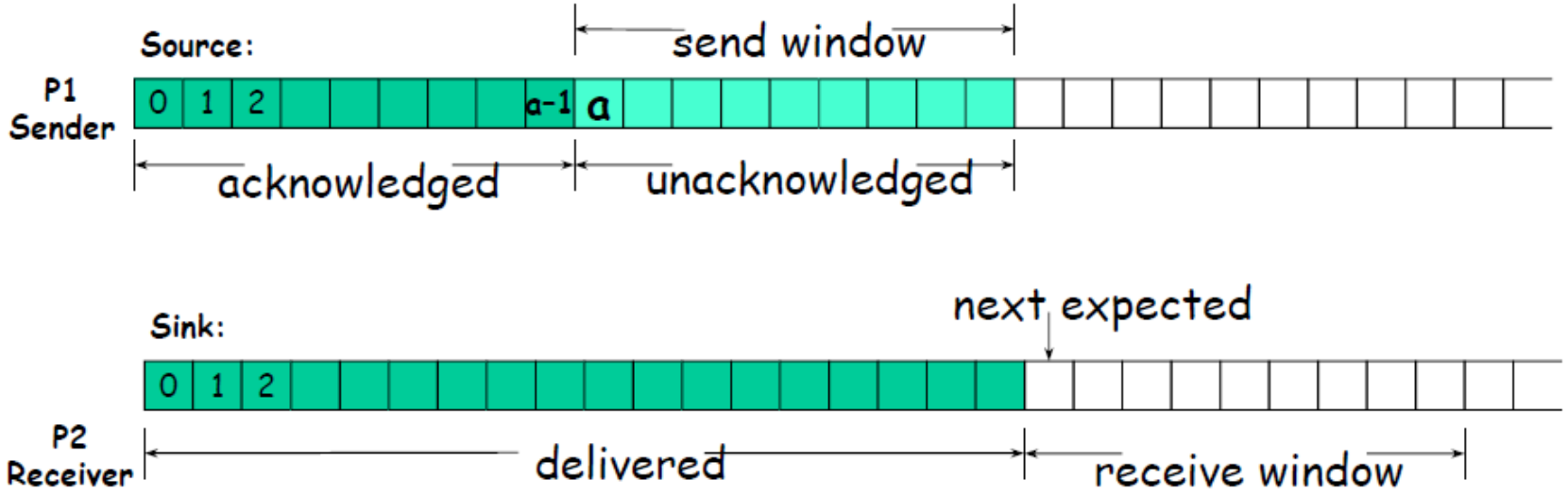
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Chapter 3: roadmap

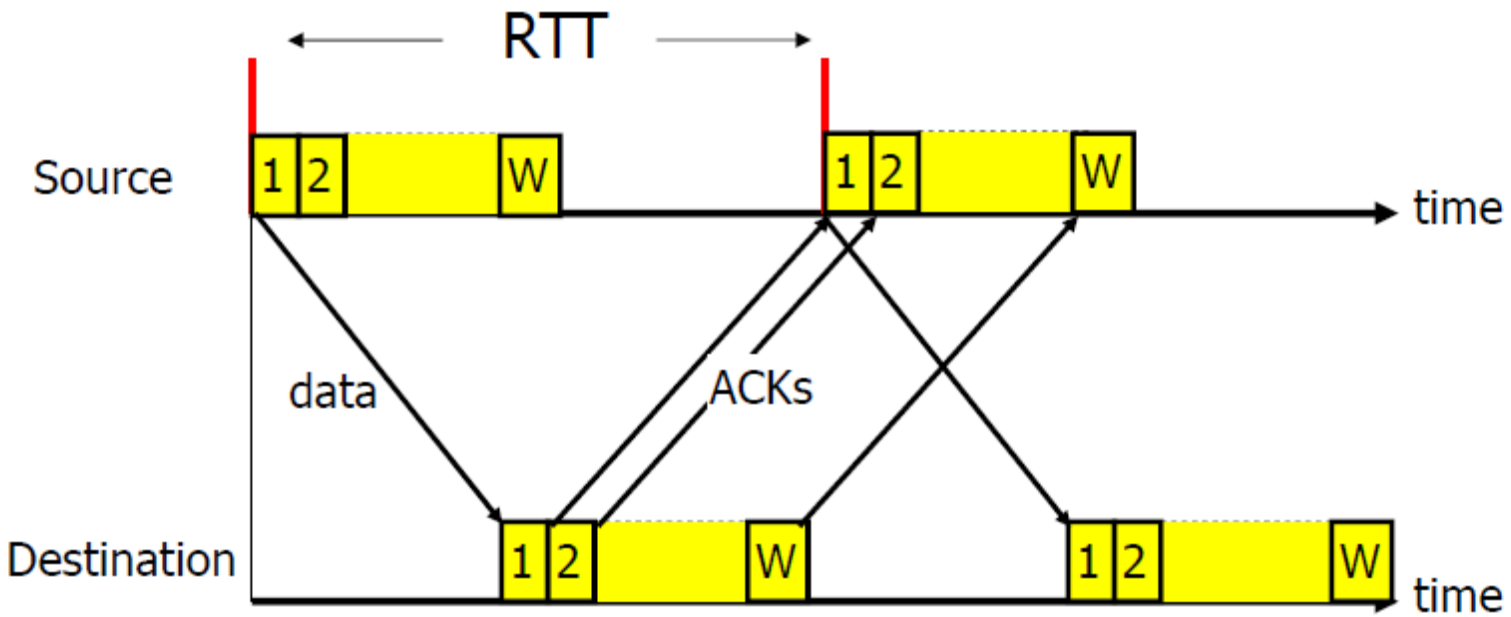
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control
- Evolution of transport-layer functionality

Sliding Window protocol

- Functions provided
  - reliable delivery (error and loss control)
  - in-order delivery
  - flow and congestion control
    - by varying send window size



Window Size Controls Sending Rate



- ~ W packets per RTT when no loss

Throughput

Max. throughput = W / RTT bytes/sec

- This is an upper bound
- Actual throughput is smaller
  - Average number in the send buffer is less than W
  - Retransmissions
- The throughput of a host's TCP send buffer is the host's send rate into the network (including original transmissions and retransmissions)

TCP Send Window Size

- TCP flow control
  - Avoid overloading receiver
  - Receiver calculates flow control window size (**rwnd**) based on the available receiver buffer space
  - Receiver sends flow control window size to sender in TCP segment heather
  - Sender keeps Send Window size less than most recently received **rwnd** value
- TCP Congestion Control
  - Avoid overloading network
  - Sender estimates network congestion from "loss indications"
  - Sender calculates congestion window size (**cwnd**)
  - Sender keeps Send Window size less than a maximum **cwnd** value
- **Sender sets W = min (cwnd, rwnd)**

TCP Congestion Control

- end-to-end control (no network assistance)
- Sender limits transmission

LastByteSent-LastByteAcked ≤ **cwnd**

Throughput ≤ **cwnd**/RTT bytes/sec

Note: For now consider **rwnd** to be very large such that the send window size is always set equal to **cwnd**

TCP Congestion Control

- How does sender estimate network congestion?
  - Packet loss is considered as an indication of network congestion
    - Time Out
    - Duplicate Acks
  - TCP sender reduces cwnd after a loss event

- How does sender determine **cwnd** size?
  - Sender adjusts existing cwnd according to the loss events
    - AIMD (Additive Increase Multiplicative Decrease)

TCP congestion control: AIMD

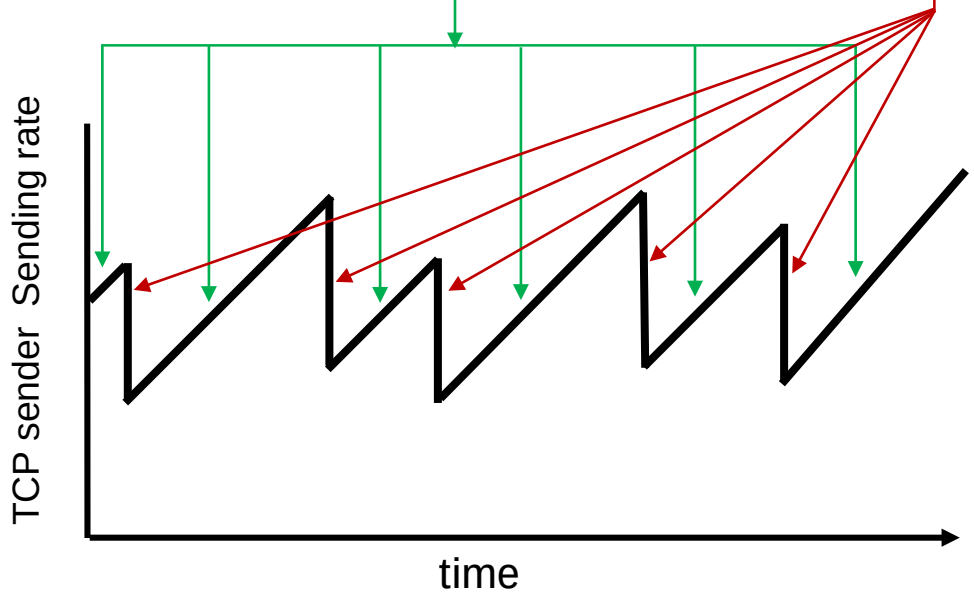
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

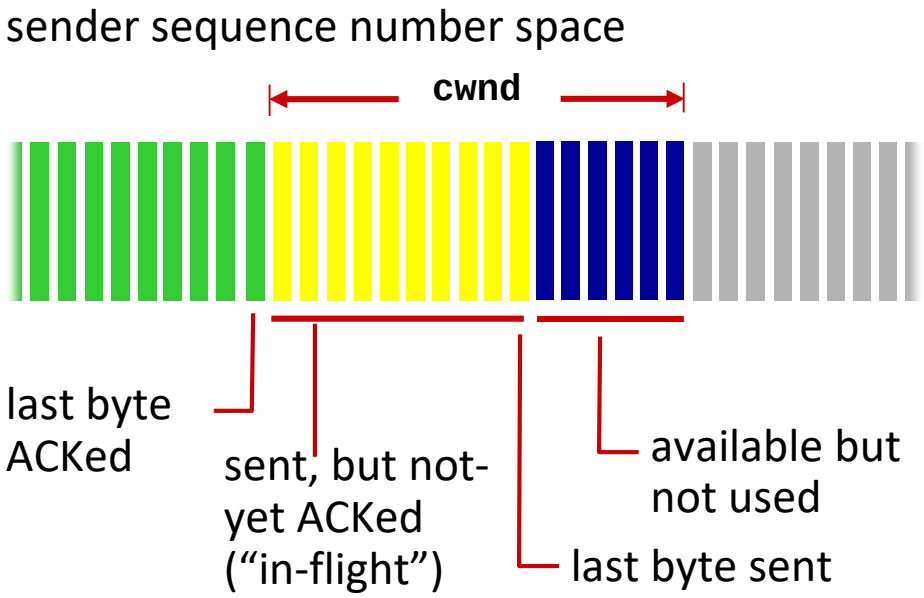
Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP congestion control: details



TCP sending behavior:

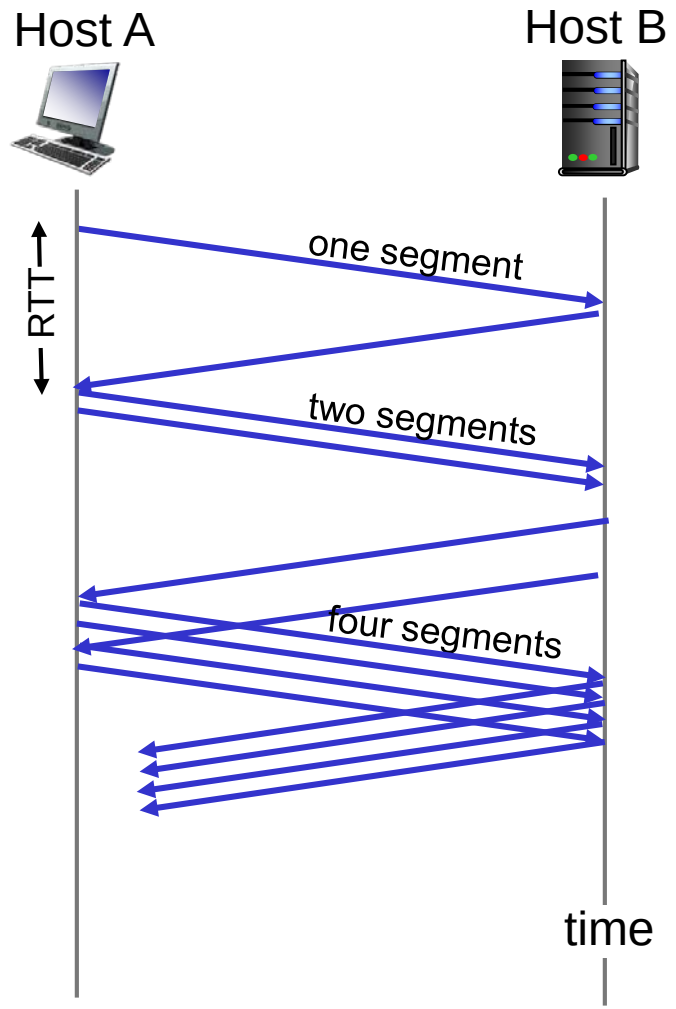
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission: `LastByteSent- LastByteAcked ≤ cwnd`
- cwnd is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



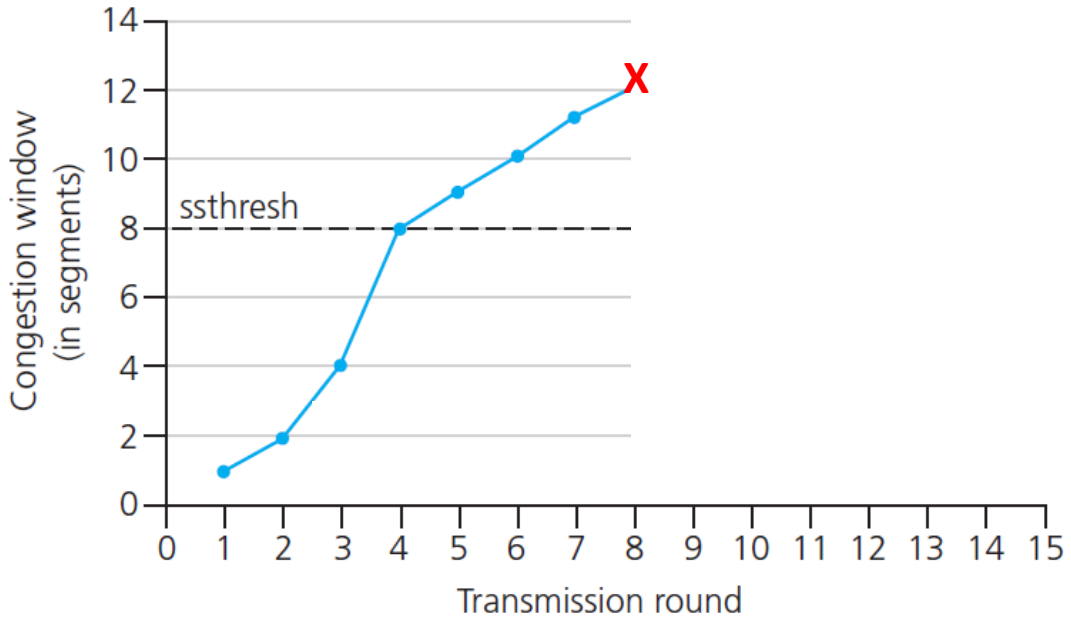
TCP: from slow start to congestion avoidance

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.

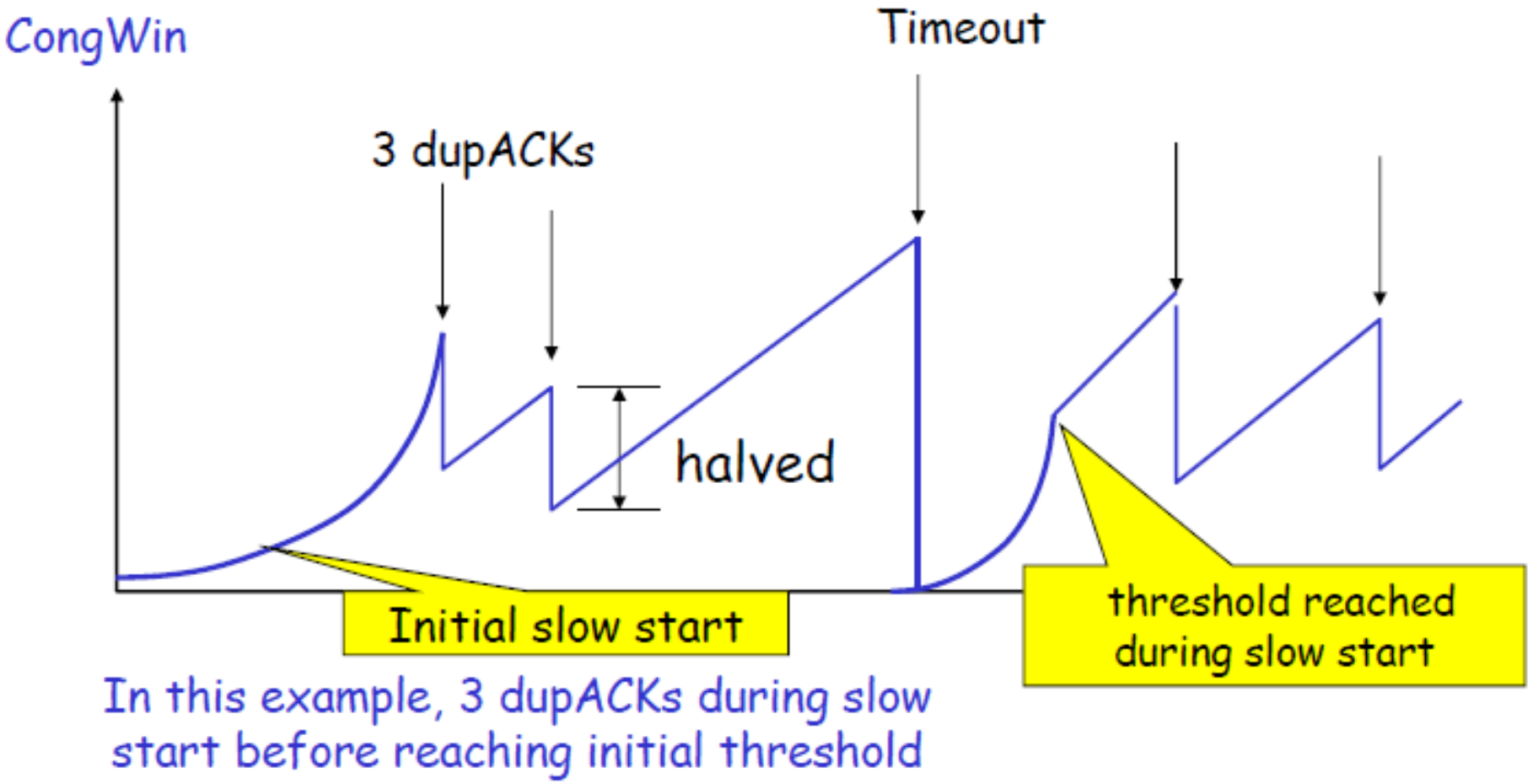
Implementation:

- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **cwnd** just before loss event



* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

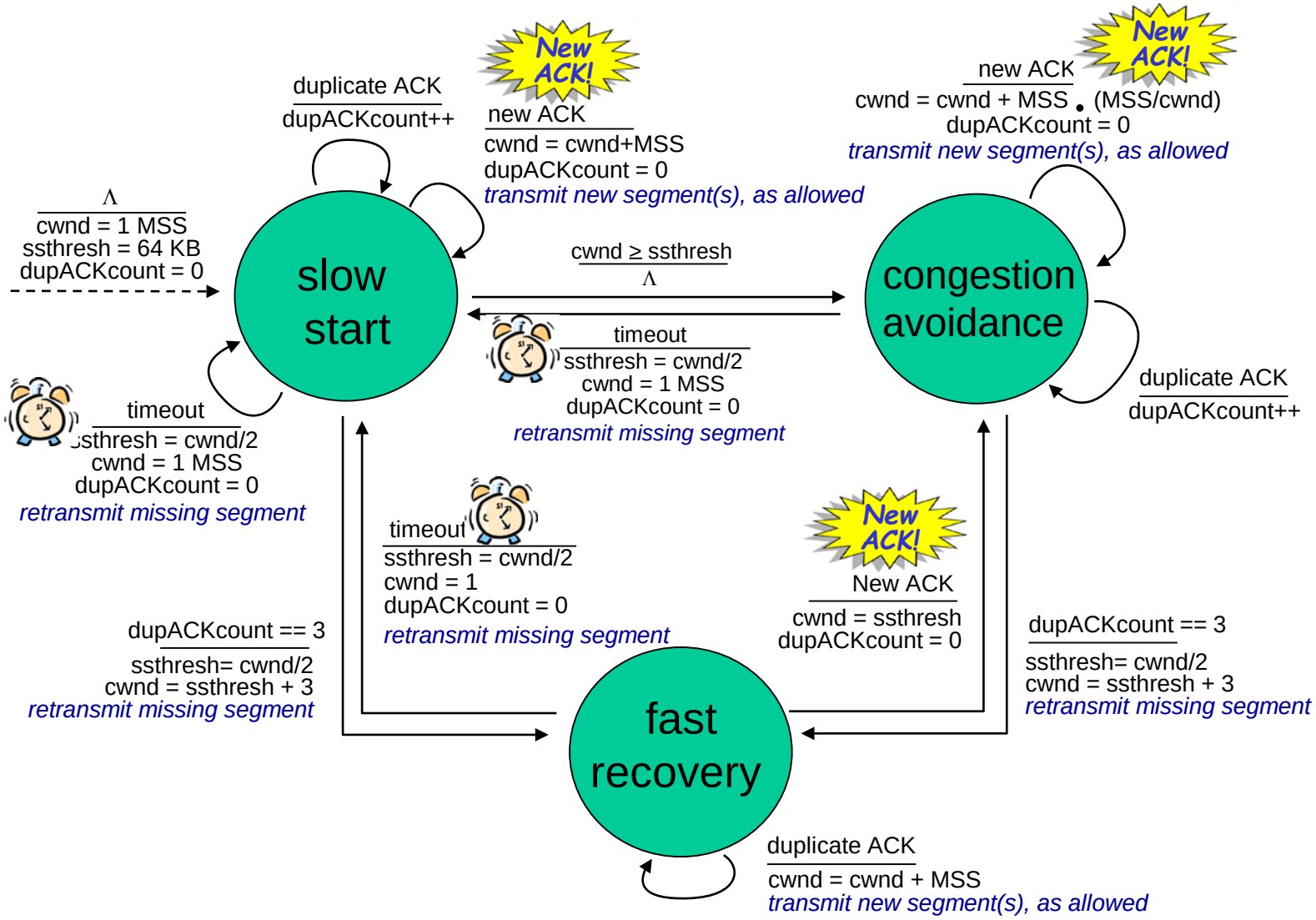
TCP Reno (example scenario)



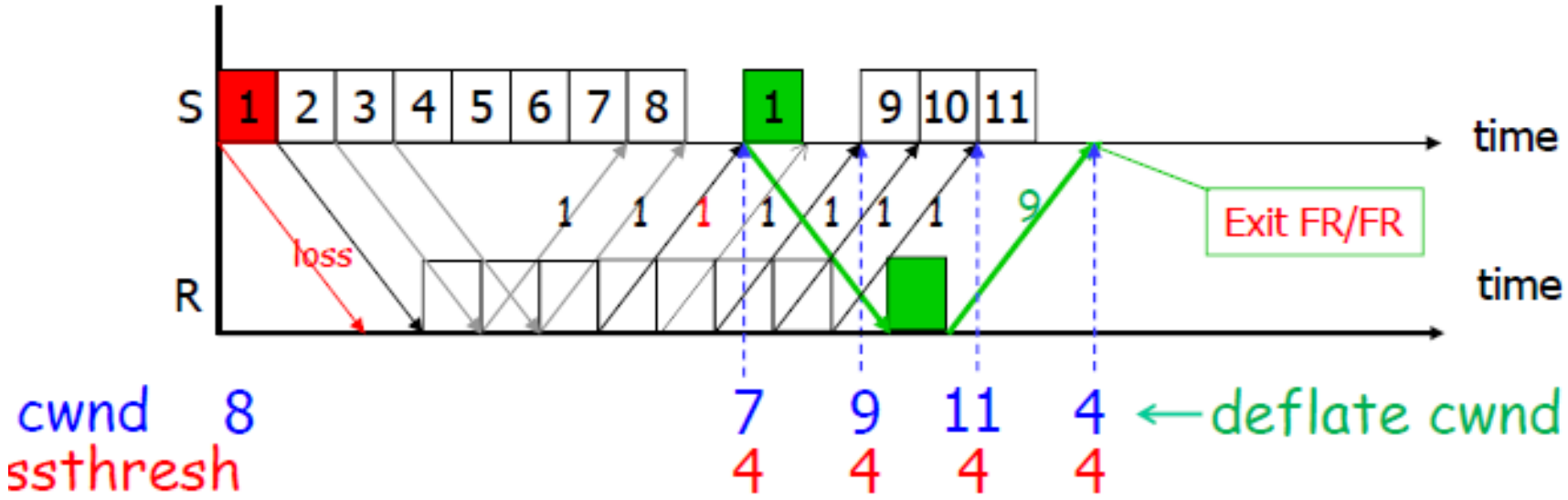
Summary (TCP Reno)

- When cwnd is below Threshold, sender in slow-start phase, window grows exponentially (until loss event or exceeding threshold).
- When cwnd is above Threshold, sender is in congestion-avoidance phase, window grows linearly.
- When timeout occurs, Threshold set to cwnd/2 and cwnd is set to 1 MSS.
- When a triple duplicate ACK occurs, Threshold set to cwnd/2 and cwnd set to Threshold (also fast retransmit happens).

Summary: TCP congestion control



Fast Recovery entry and exit

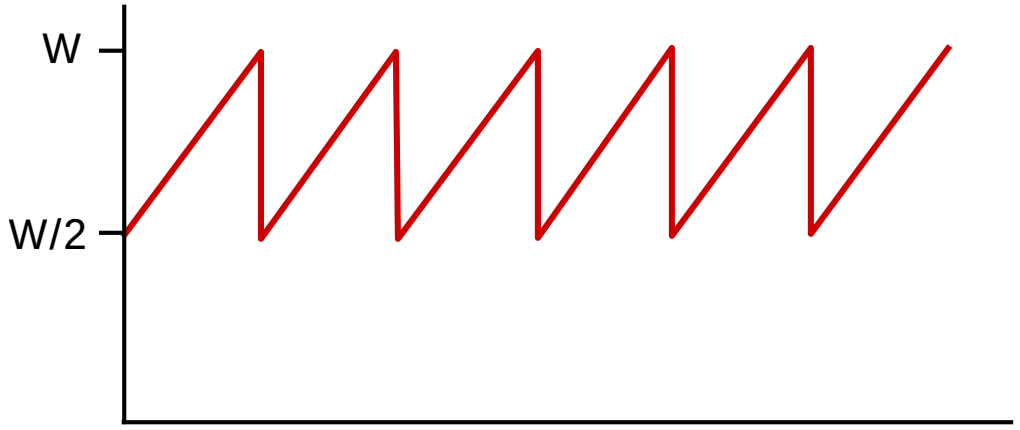


- Above scenario: Packet 1 is lost, packets 2, 3, and 4 are received; 3 dupACKs with seq. no. 1 returned
- Fast retransmit
  - Retransmit packet 1 upon 3 dupACKs
- Fast recovery (in steps)
  - Inflate cwnd with #dupACKs such that new packets 9, 10, and 11 can be sent while repairing loss

TCP throughput

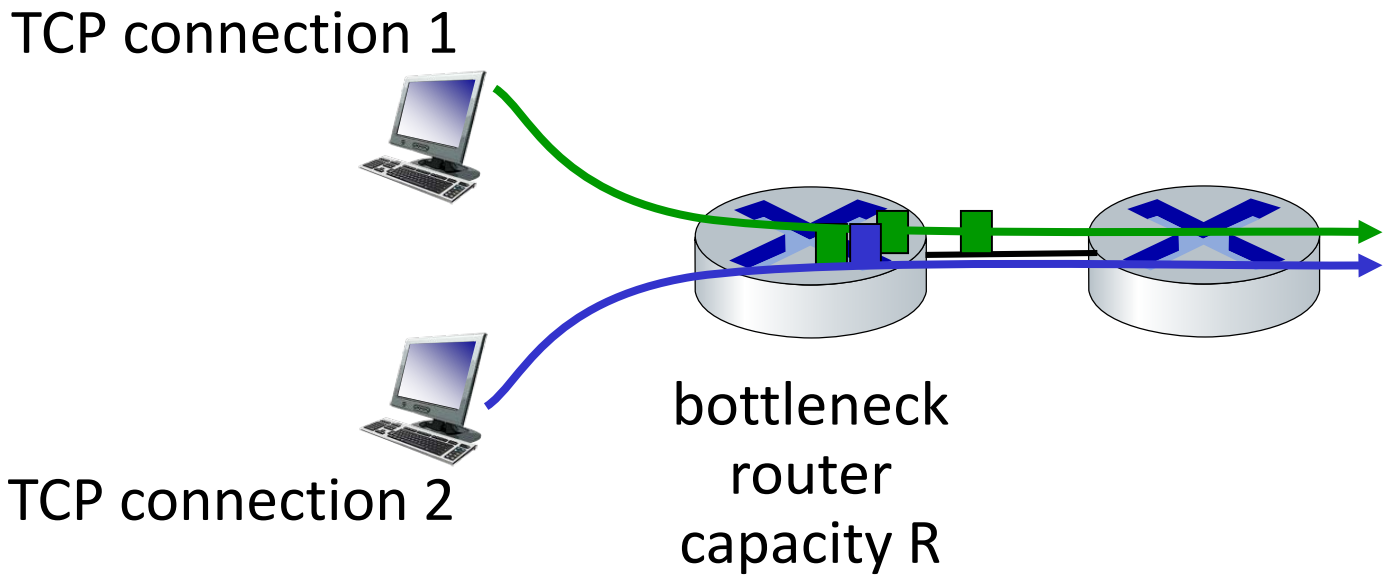
- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume there is always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP fairness

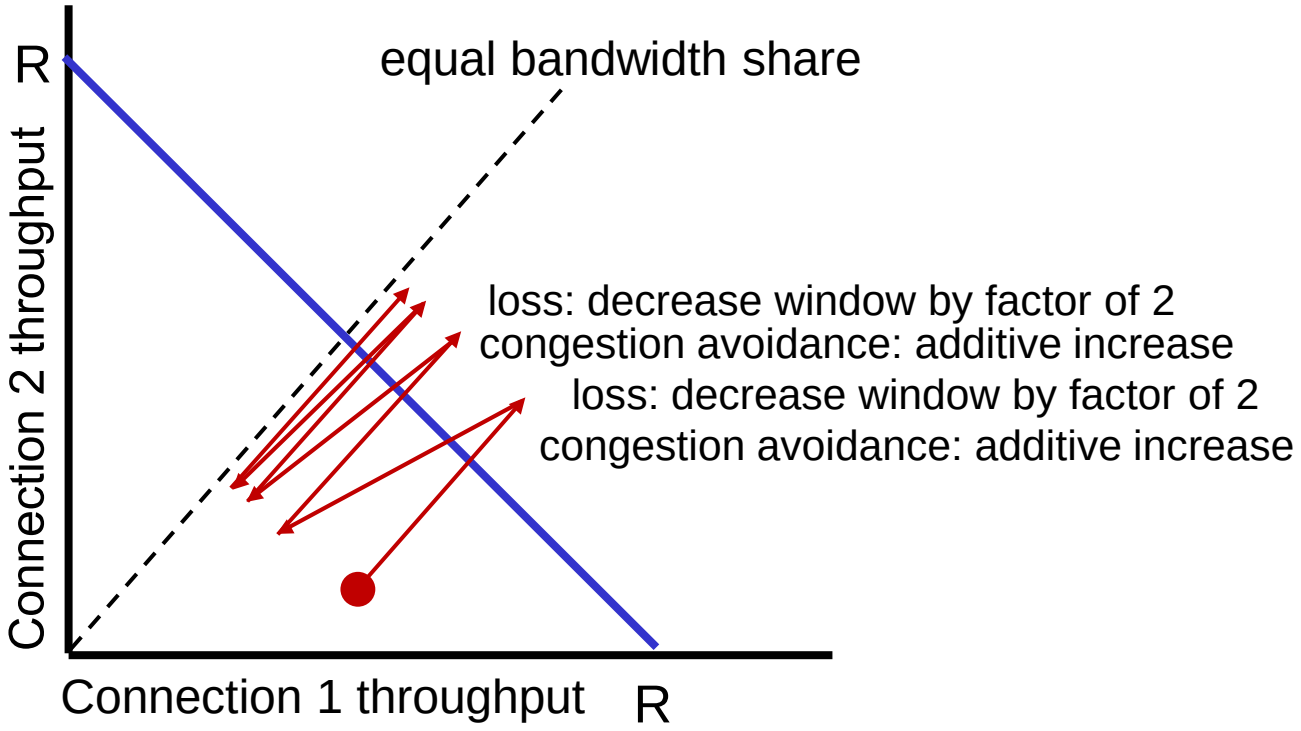
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be "fair"?

Fairness and UDP

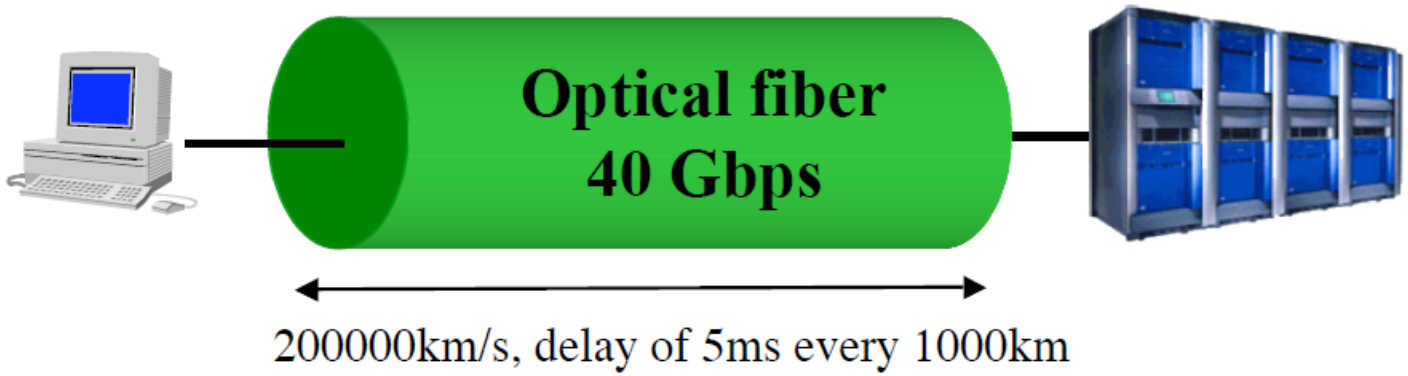
- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no "Internet police" policing use of congestion control

Fairness, parallel TCP connections

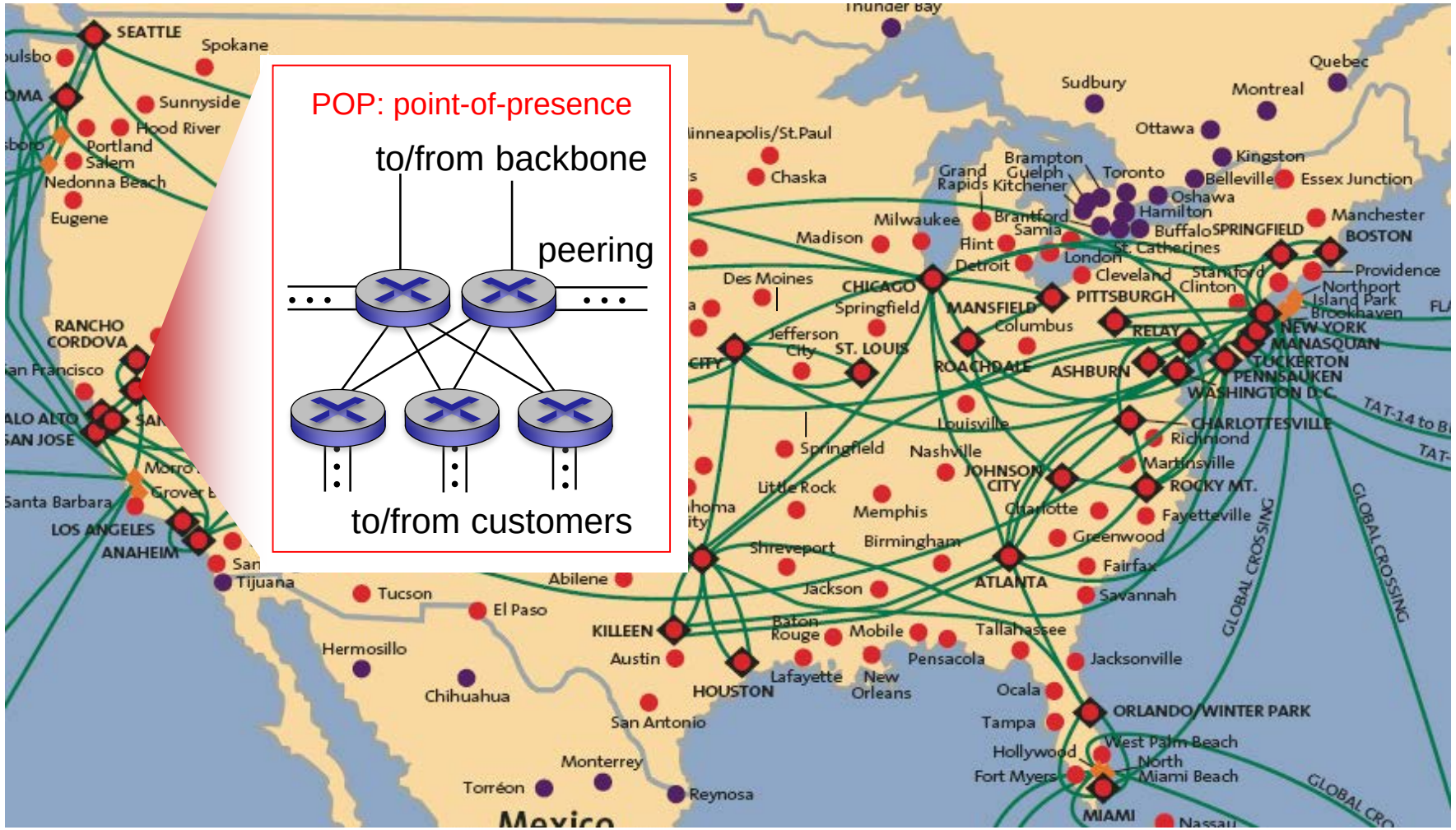
- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

TCP in high-speed links

- Today's backbone links are optical, DWDM–based,
- High bit rates (1~100 Gbps)
- Long distances (Thousands of Km)
- Transmission time <<< propagation time

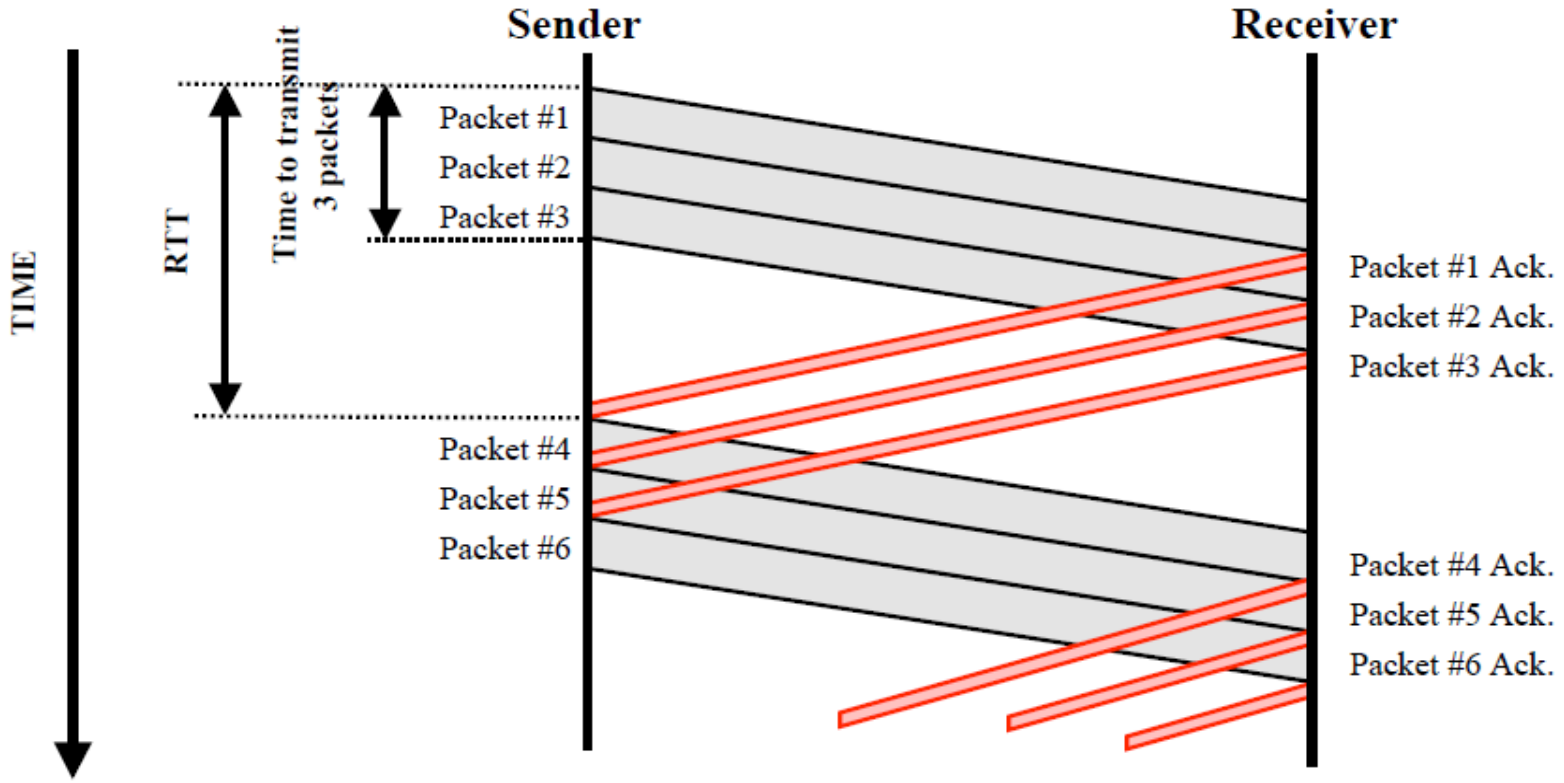


Tier-1 ISP: e.g., Sprint



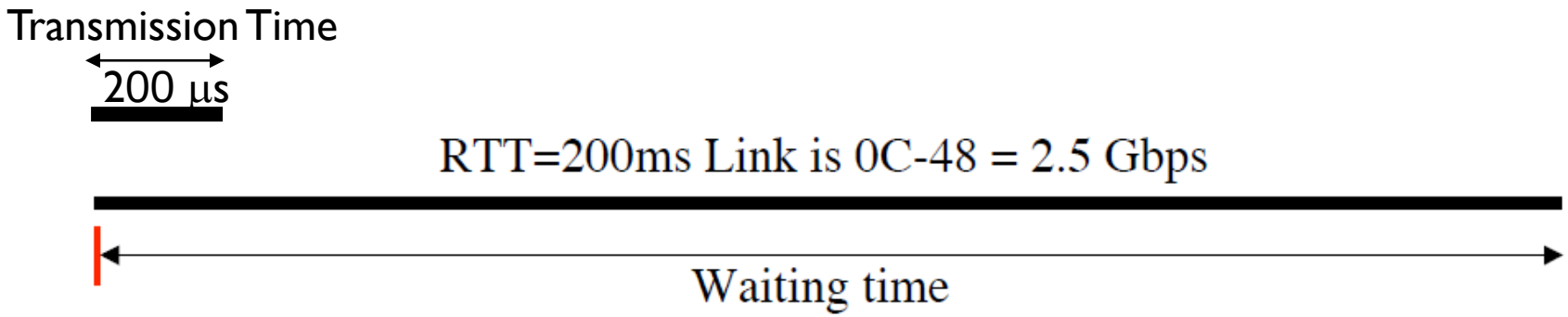
Problem: window size

- The default maximum window size is 64Kbytes.
- Then the sender has to wait for acks.



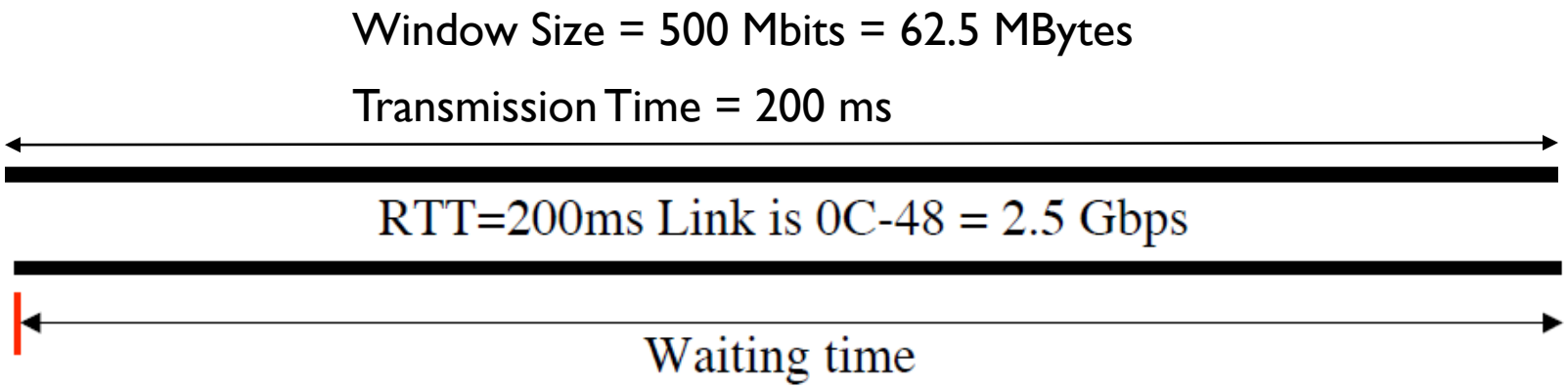
Problem: window size

- Less than 0.1% of the link bandwidth is utilized
- A big file transfer takes minutes



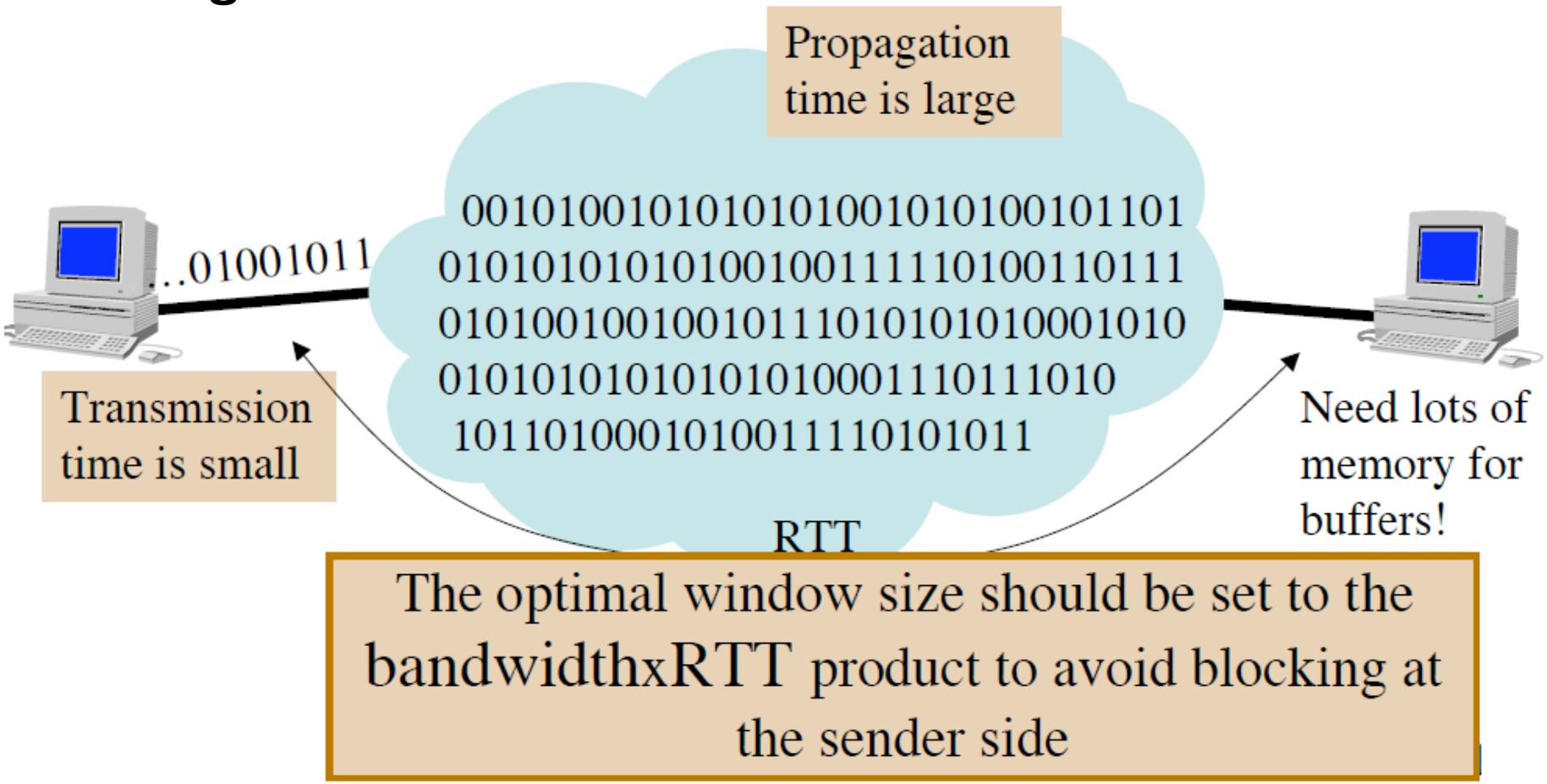
Problem: window size

- Much larger window size to utilize the link BW
- To keep sending till the first ACK
  - W = RTTx R bits

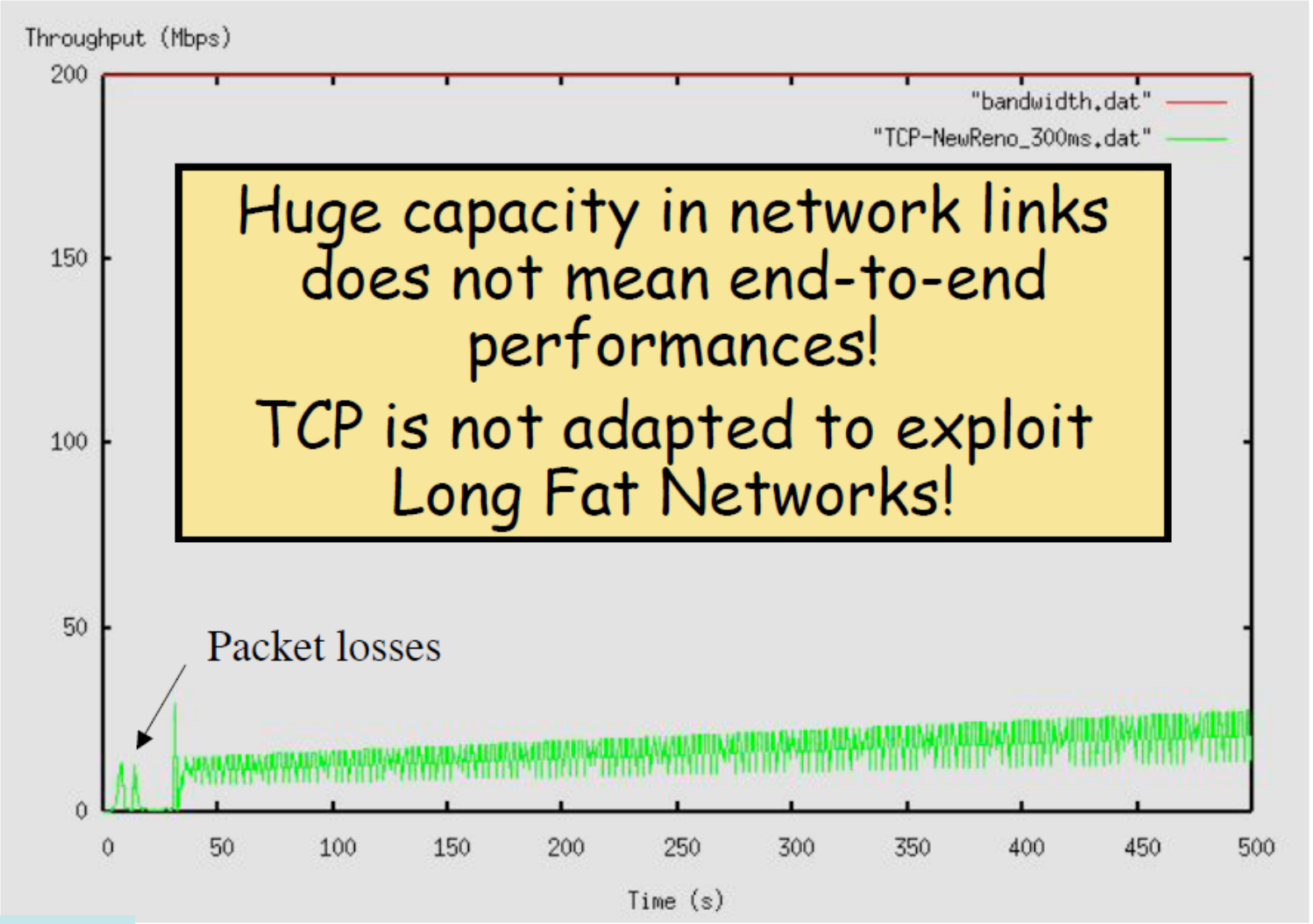


High capacity network

LFN: Long Fat Network

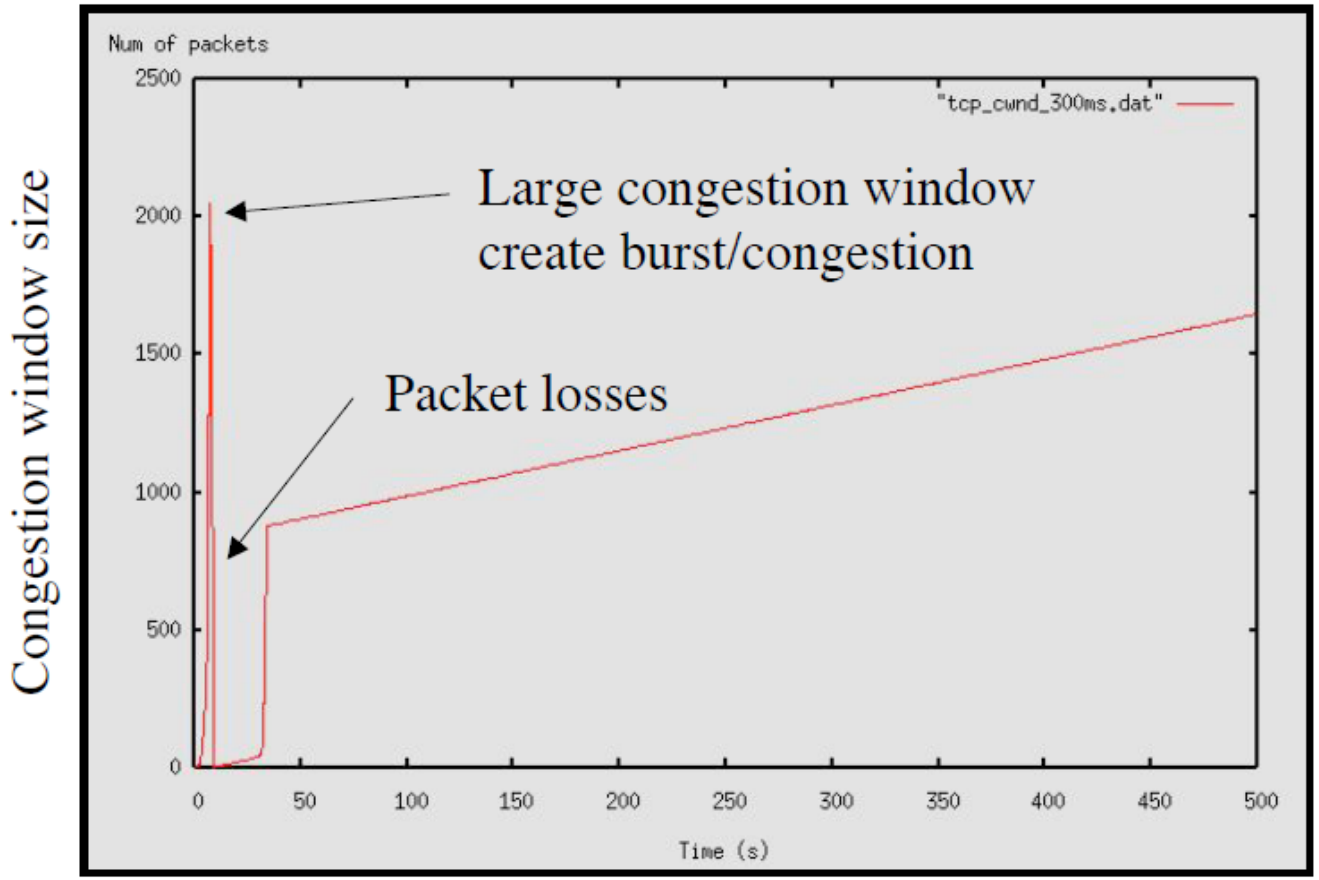


TCP throughput on a 200 Mbps link



Side effect of large windows

- TCP becomes very sensitive to packet losses in LFN



High capacity networks

- Sustaining high congestion windows:

A Standard TCP connection with:

– 1500-byte packets;

– a 100 ms round-trip time;

– a steady-state throughput of 10 Gbps;

would require:

– an average congestion window of 83,333 segments;

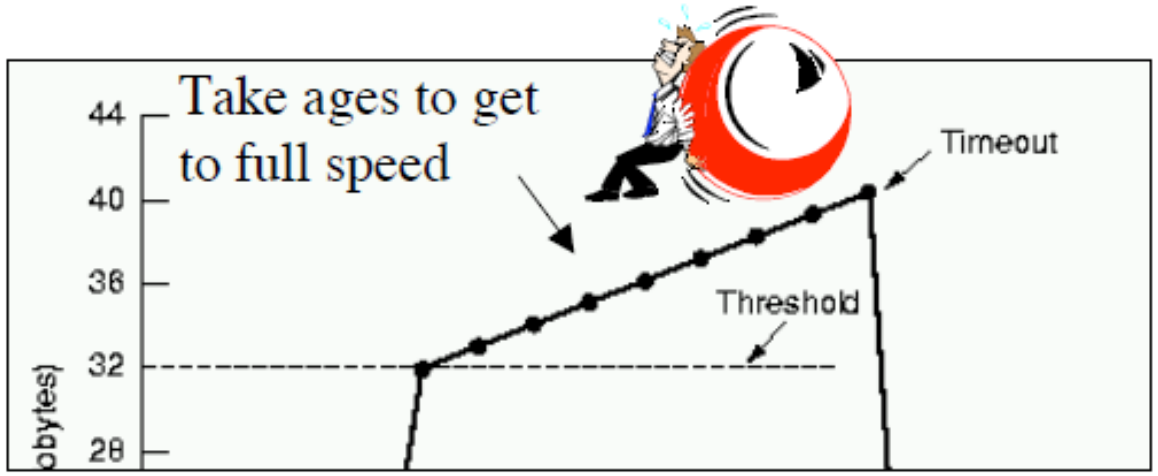
– and at most one drop (or mark) every 5,000,000,000 packets (or equivalently, at most one drop every 1 2/3 hours).

This is not realistic.

From S. Floyd

Additive increase is still too slow

- With 100ms of round trip time, a connection needs 203 minutes (3h23) to get 1Gbps starting from 1Mbps

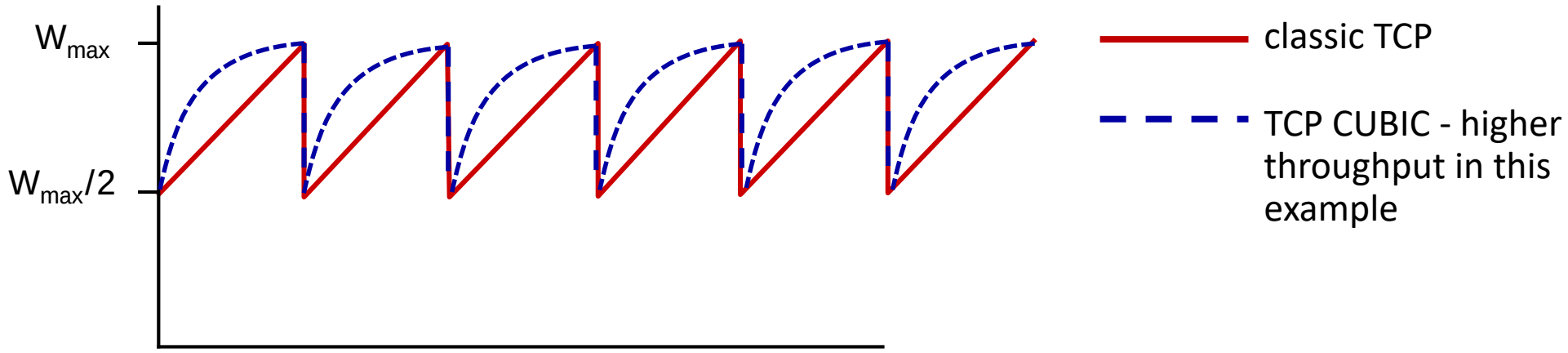


Solution?

- Several parallel TCP connections
  - Requires less configuration in the standard TCP
- New TCP Protocols for high-speed links
  - Research
    - Fast TCP
  - OS:
    - Cubic

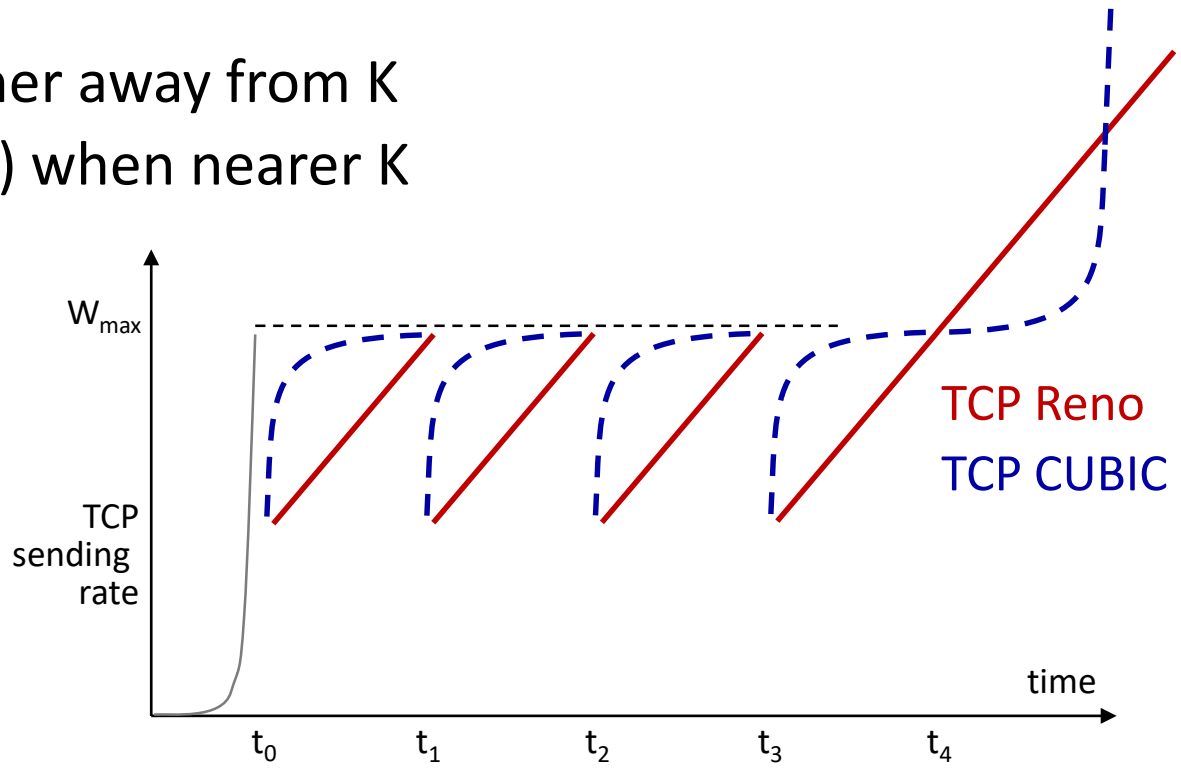
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



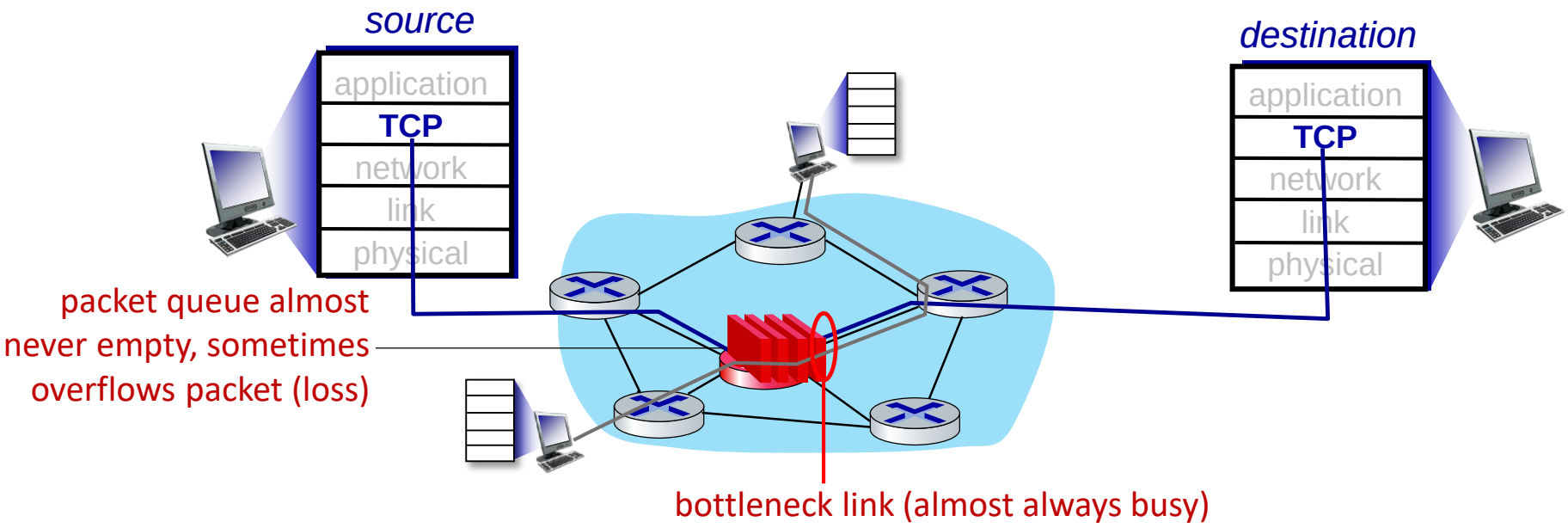
TCP CUBIC

- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



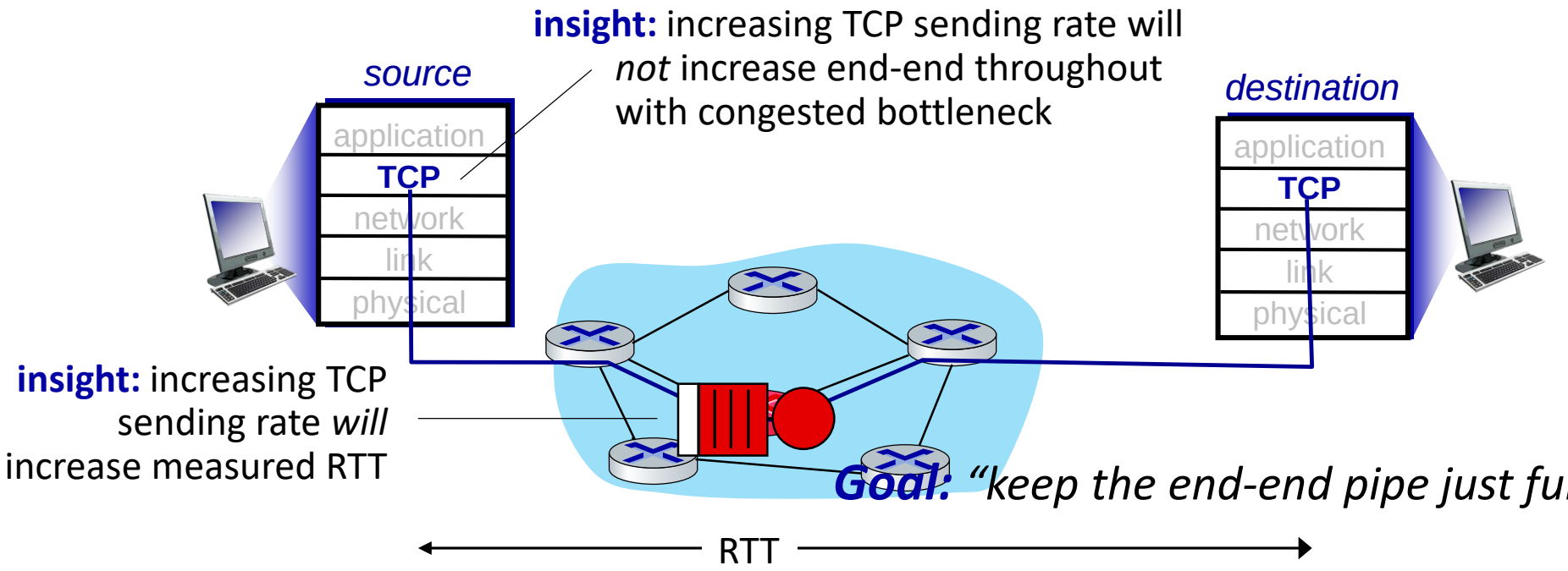
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



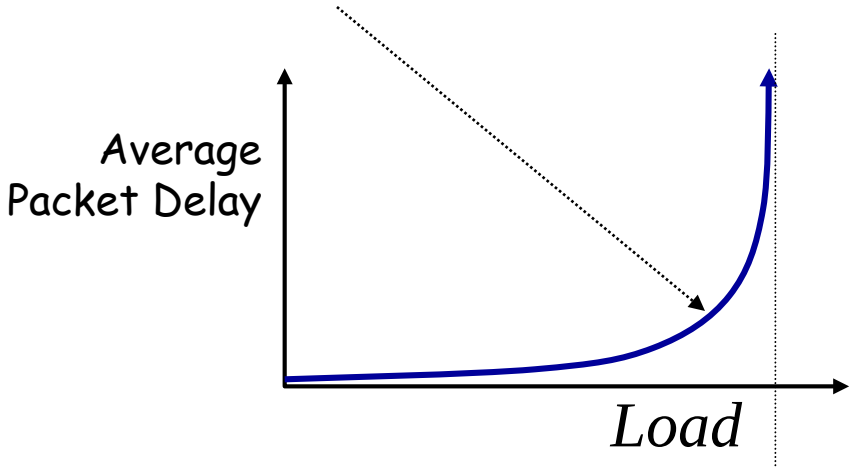
TCP and the congested "bottleneck link"

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link



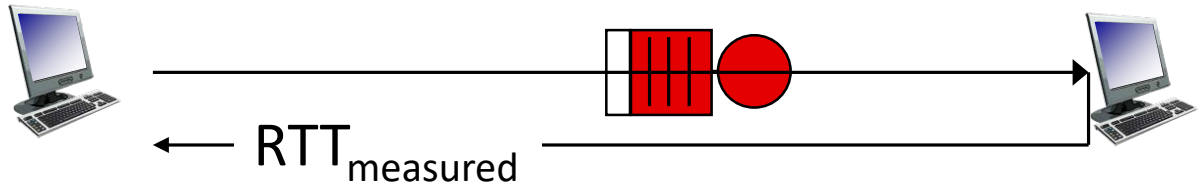
Congestion Avoidance

- TCP reacts to congestion *after* it takes place. The data rate changes rapidly and the system is barely stable (or is even unstable).
- Can we *predict* when congestion is about to happen and avoid it? E.g. by detecting the knee of the curve.



Delay-based TCP congestion control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window cwnd is cwnd/RTT_{min}

if measured throughput "very close" to uncongested throughput
    increase cwnd linearly /* since path not congested */
else if measured throughput "far below" uncongested throughout
    decrease cwnd linearly /* since path is congested */

Delay-based TCP congestion control

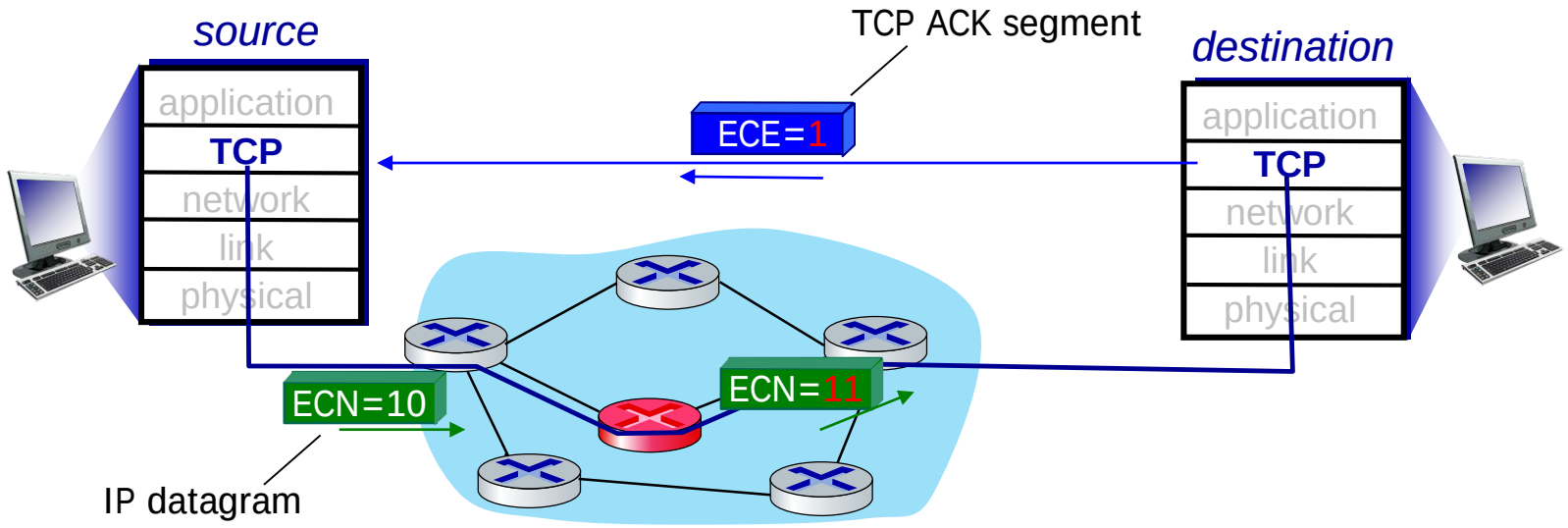
- congestion control without inducing/forcing loss
- maximizing throughout ("keeping the just pipe full… ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network

    (Bottleneck Bandwidth and Round-trip propagation time)

Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header ECE bit marking)



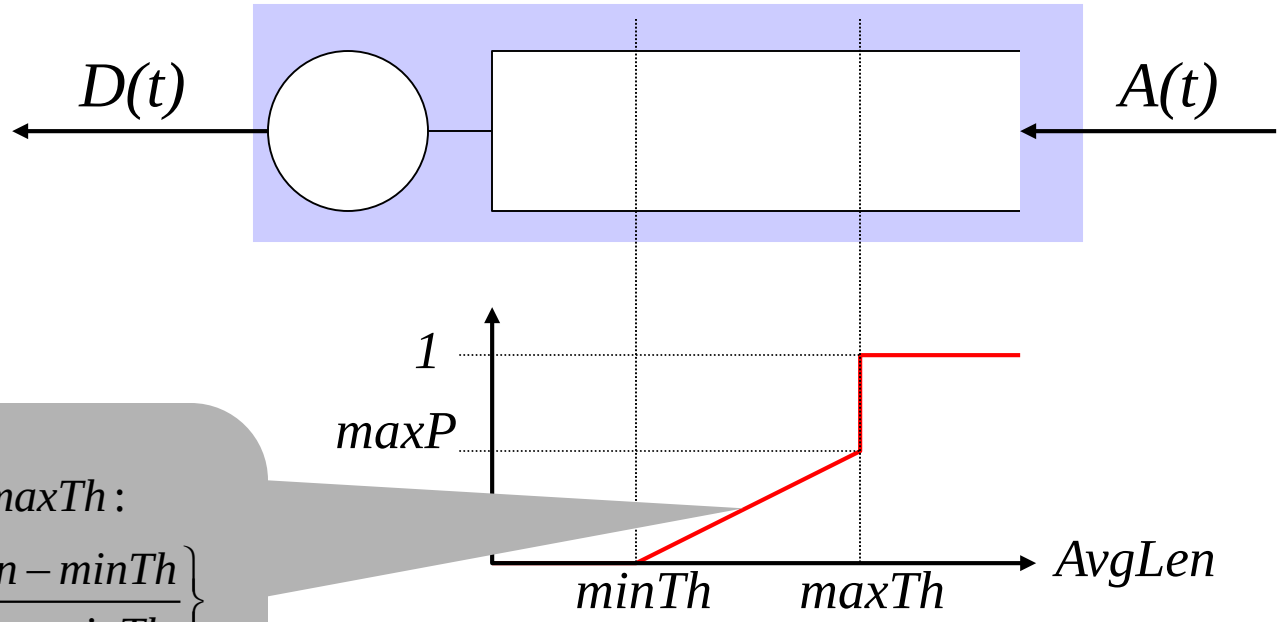
Random Early Detection (RED)

- RED is similar to DECbit, and was designed to work well with TCP.
- RED implicitly notifies sender by dropping packets.
- Drop probability is increased as the *average* queue length increases.
- (Geometric) moving average of the queue length is used so as to detect long term congestion, yet allow short term bursts to arrive.

$$AvgLen_{n+1} = (1-\alpha) \times AvgLen_n + \alpha \times Length_n$$

i.e. $$AvgLen_{n+1} = \sum_{i=1}^{n} Length_i (\alpha)(1-\alpha)^{n-i}$$

RED Drop Probabilities



If $minTh < AvgLen < maxTh$:

$$\hat{p}_{AvgLen} = maxP\left\{\frac{AvgLen - minTh}{maxTh - minTh}\right\}$$

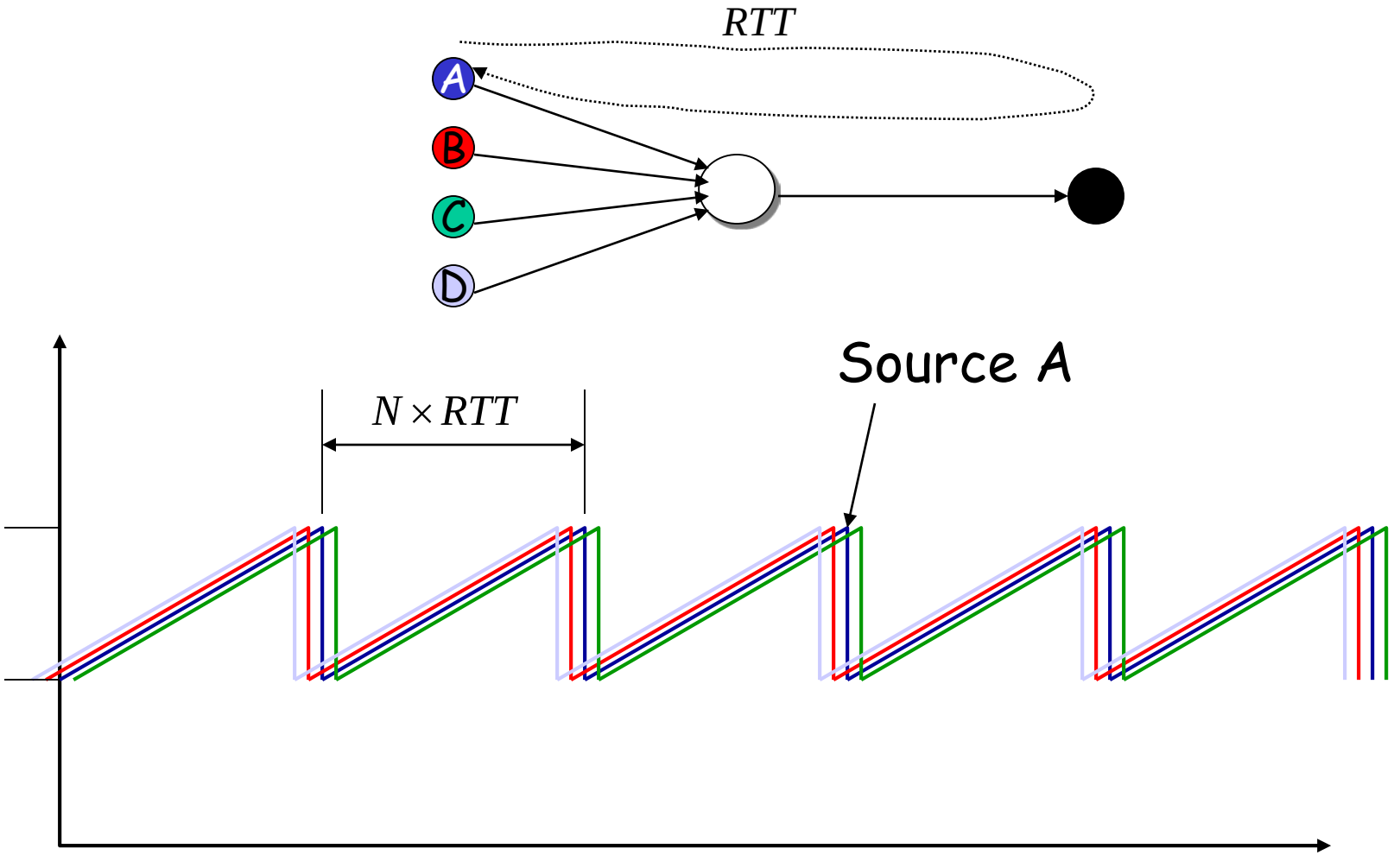
$$\Pr(\text{Drop Packet}) = \frac{\hat{p}_{AvgLen}}{1 - count \times \hat{p}_{AvgLen}}$$

count counts how long we've been in $minTh < AvgLen < maxTh$ since we last dropped a packet. i.e. drops are spaced out in time, reducing likelihood of re-entering slow-start.

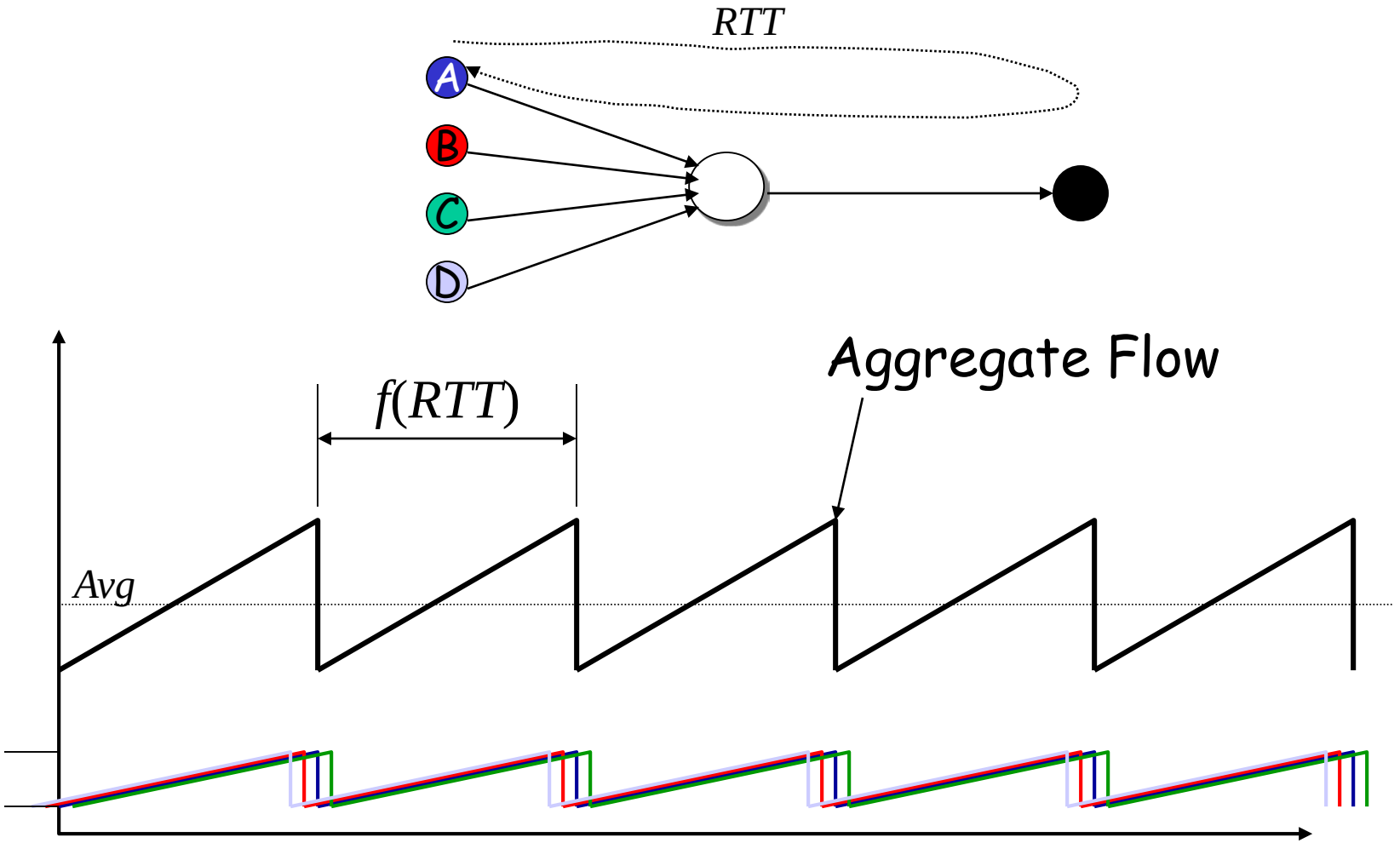
Properties of RED

- Drops packets before queue is full, in the hope of reducing the rates of some flows.
- Drops packet for each flow *roughly* in proportion to its rate.
- Drops are spaced out in time.
- Because it uses average queue length, RED is tolerant of bursts.
- Random drops hopefully desynchronize TCP sources.

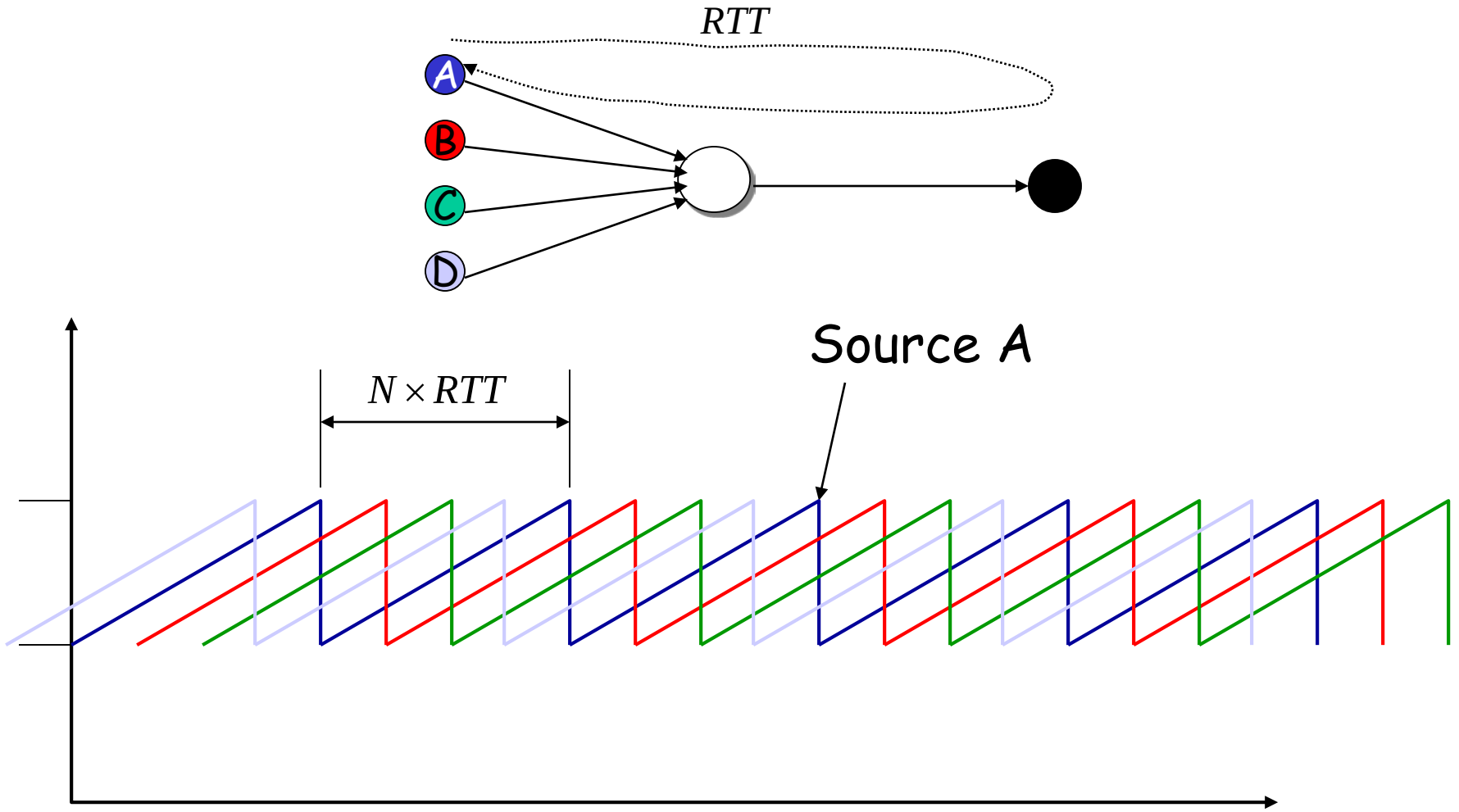
Synchronization of sources



Synchronization of sources



Desynchronized sources



Desynchronized sources