DRONACHARYA
College of Engineering

## Computer Science & Engineering

# Data Communication and Computer Networks

( MTCSE-101-A )

# CONGESTION CONTROL

# Congestion Control

- When one part of the subnet (e.g. one or more routers in an area) becomes overloaded, congestion results.
- Because routers are receiving packets faster than they can forward them, one of two things must happen:
  - The subnet must prevent additional packets from entering the congested region until those already present can be processed.
  - The congested routers can discard queued packets to make room for those that are arriving.

# Factors that Cause Congestion

- Packet arrival rate exceeds the outgoing link capacity.
- Insufficient memory to store arriving packets
- Bursty traffic
- Slow processor

# Congestion Control vs Flow Control

- Congestion control is a global issue – involves every router and host within the subnet
- Flow control – scope is point-to-point; involves just sender and receiver.

# Congestion Control, cont.

- Congestion Control is concerned with efficiently using a network at high load.
- Several techniques can be employed. These include:
  - Warning bit
  - Choke packets
  - Load shedding
  - Random early discard
  - Traffic shaping
- The first 3 deal with congestion detection and recovery. The last 2 deal with congestion avoidance.

# Warning Bit

- A special bit in the packet header is set by the router to warn the source when congestion is detected.
- The bit is copied and piggy-backed on the ACK and sent to the sender.
- The sender monitors the number of ACK packets it receives with the warning bit set and adjusts its transmission rate accordingly.

# Choke Packets

- A more direct way of telling the source to slow down.
- A choke packet is a control packet generated at a congested node and transmitted to restrict traffic flow.
- The source, on receiving the choke packet must reduce its transmission rate by a certain percentage.
- An example of a choke packet is the ICMP Source Quench Packet.

# Hop-by-Hop Choke Packets

- Over long distances or at high speeds choke packets are not very effective.
- A more efficient method is to send to choke packets hop-by-hop.
- This requires each hop to reduce its transmission even before the choke packet arrive at the source.

# Load Shedding

- When buffers become full, routers simply discard packets.
- Which packet is chosen to be the victim depends on the application and on the error strategy used in the data link layer.
- For a file transfer, for, e.g. cannot discard older packets since this will cause a gap in the received data.
  - For real-time voice or video it is probably better to

throw away old data and keep new packets.

- Get the application to mark packets with discard priority.

# Random Early Discard (RED)

- This is a proactive approach in which the router discards one or more packets *before* the buffer becomes completely full.
- Each time  a packet arrives, the RED algorithm computes the average queue length, ***avg***.
- If *avg* is lower than some lower threshold, congestion is assumed to be minimal or non-existent and the packet is queued.

# RED, cont.

- If *avg* is greater than some upper threshold, congestion is assumed to be serious and the packet is discarded.
- If *avg* is between the two thresholds, this might indicate the onset of congestion. The probability of congestion is then calculated.

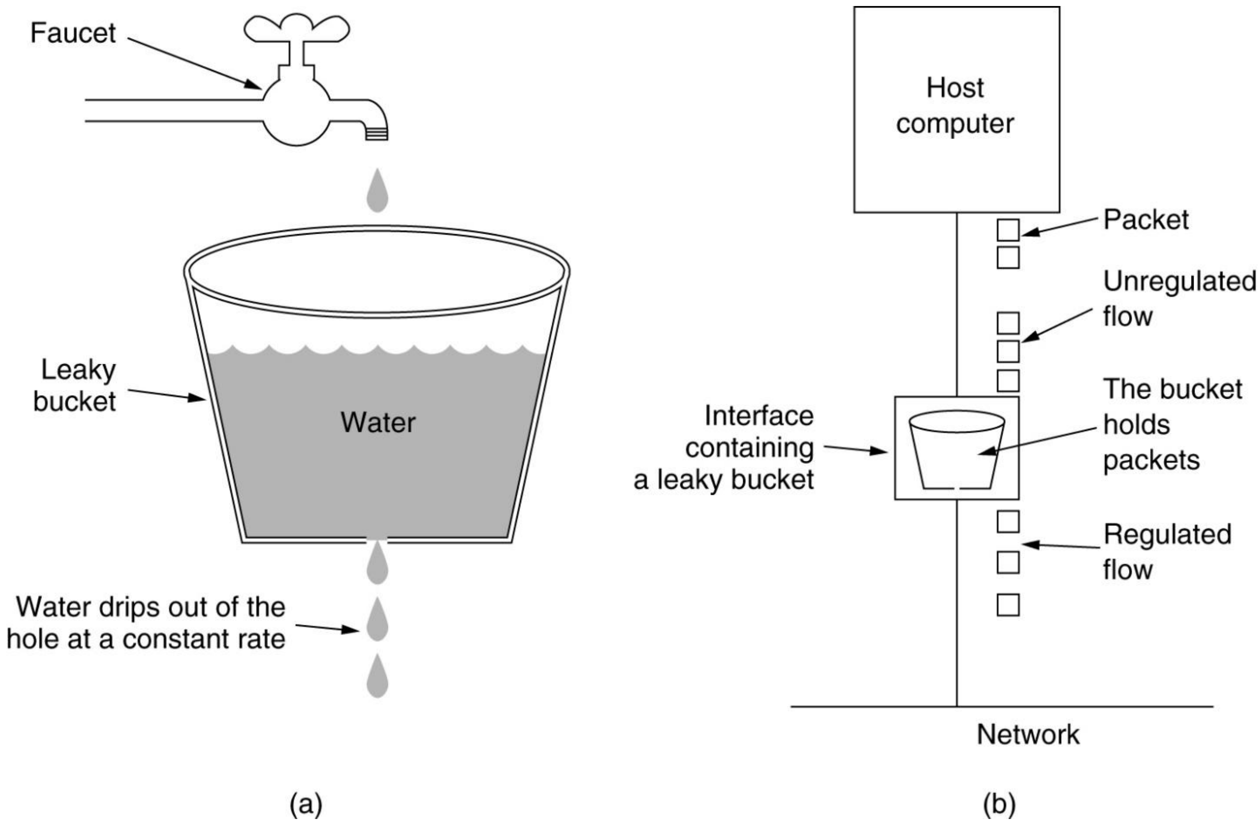
# Traffic Shaping

- Another method of congestion control is to "shape" the traffic before it enters the network.
- Traffic shaping controls the *rate* at which packets are sent (not just how many). Used in ATM and Integrated Services networks.
- At connection set-up time, the sender and carrier negotiate a traffic pattern (shape).
- Two traffic shaping algorithms are:
  - Leaky Bucket
  - Token Bucket

# The Leaky Bucket Algorithm

- The **Leaky Bucket Algorithm** used to control rate in a network. It is implemented as a single-server queue with constant service time. If the bucket (buffer) overflows then packets are discarded.

# The Leaky Bucket Algorithm




(a) A leaky bucket with water. (b) a leaky bucket with packets.

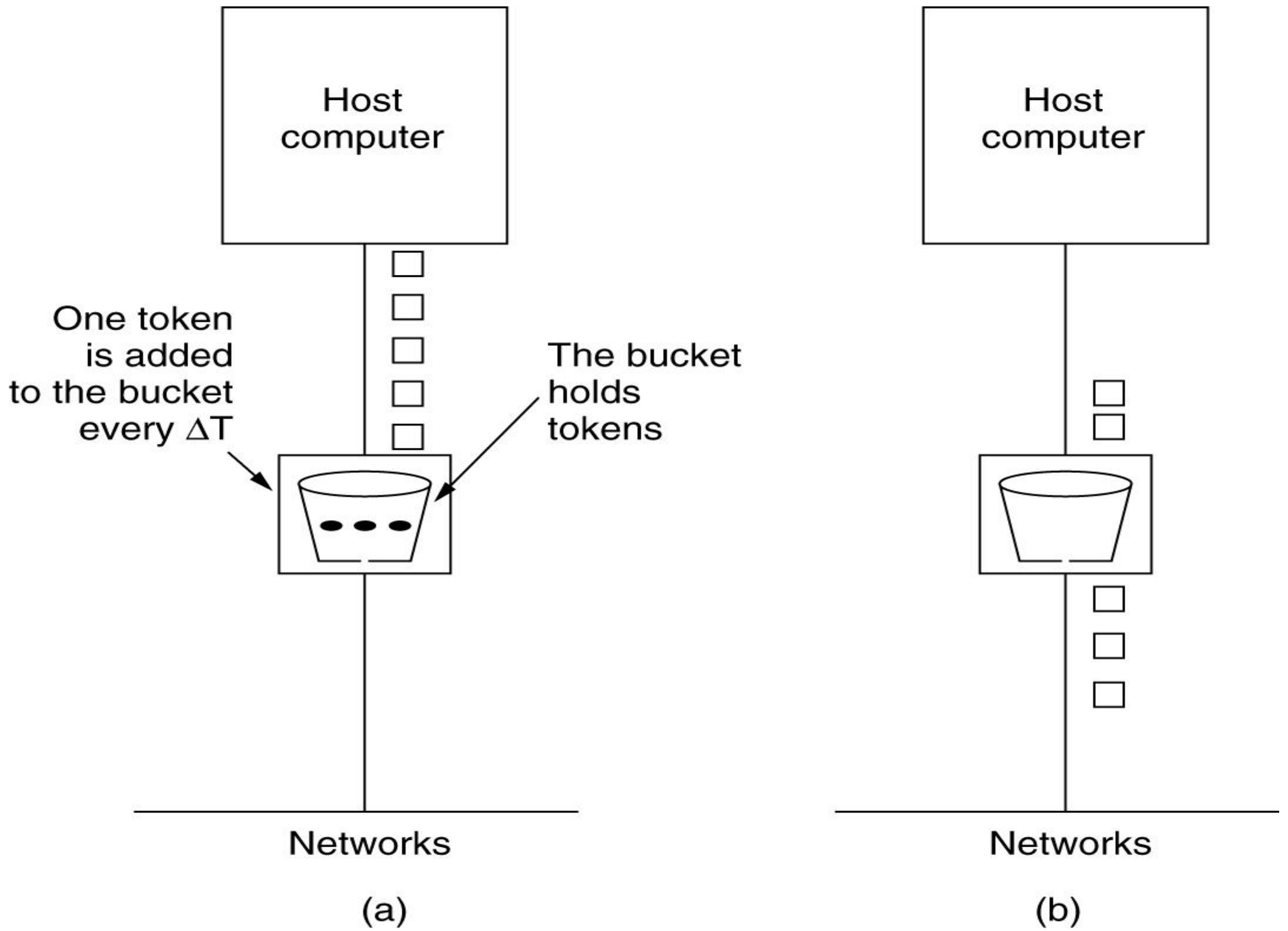
# Leaky Bucket Algorithm, cont.

- The leaky bucket enforces a constant output rate (average rate) regardless of the burstiness of the input. Does nothing when input is idle.
- The host injects one packet per clock tick onto the network. This results in a uniform flow of packets, smoothing out bursts and reducing congestion.
- When packets are the same size (as in ATM cells), the one packet per tick is okay. For variable length packets though, it is better to allow a fixed number of bytes per tick. E.g. 1024 bytes per tick will allow one 1024-byte packet or two 512-byte packets or four 256-byte packets on 1 tick.

# Token Bucket Algorithm

- In contrast to the LB, the Token Bucket Algorithm, allows the output rate to vary, depending on the size of the burst.
- In the TB algorithm, the bucket holds tokens. To transmit a packet, the host must capture and destroy one token.
- Tokens are generated by a clock at the rate of one token every $\Delta t$ sec.
- Idle hosts can capture and save up tokens (up to the max. size of the bucket) in order to send larger bursts later.

# The Token Bucket Algorithm




(a) Before. (b) After.

# Leaky Bucket vs Token Bucket

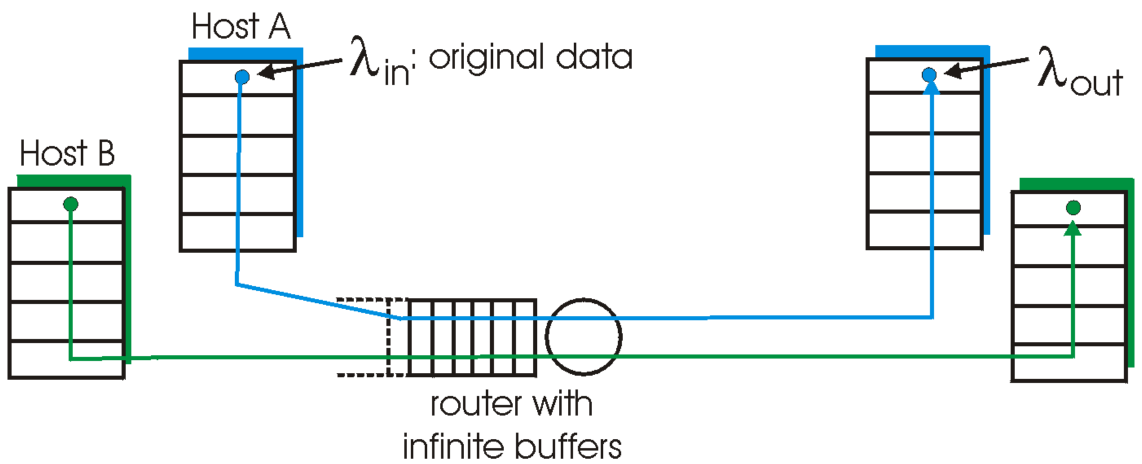
- LB discards packets; TB does not. TB discards tokens.
- With TB, a packet can only be transmitted if there are enough tokens to cover its length in bytes.
- LB sends packets at an average rate. TB allows for large bursts to be sent faster by speeding up the output.
- TB allows saving up tokens (permissions) to send large bursts. LB does not allow saving.

# Principles of Congestion Control

Congestion:

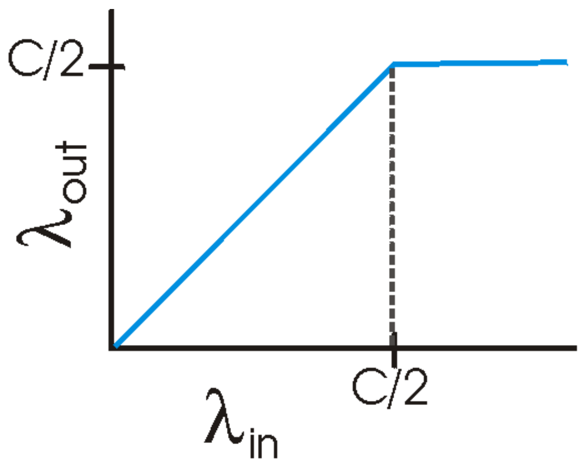
- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)
- a highly important problem!

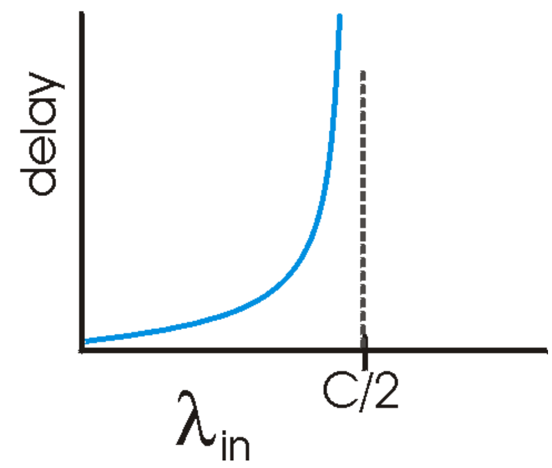
# Causes/costs of congestion: scenario 1




- two senders, two receivers
- one router,
- infinite buffers
- no retransmission

# Causes/costs of congestion: scenario 1

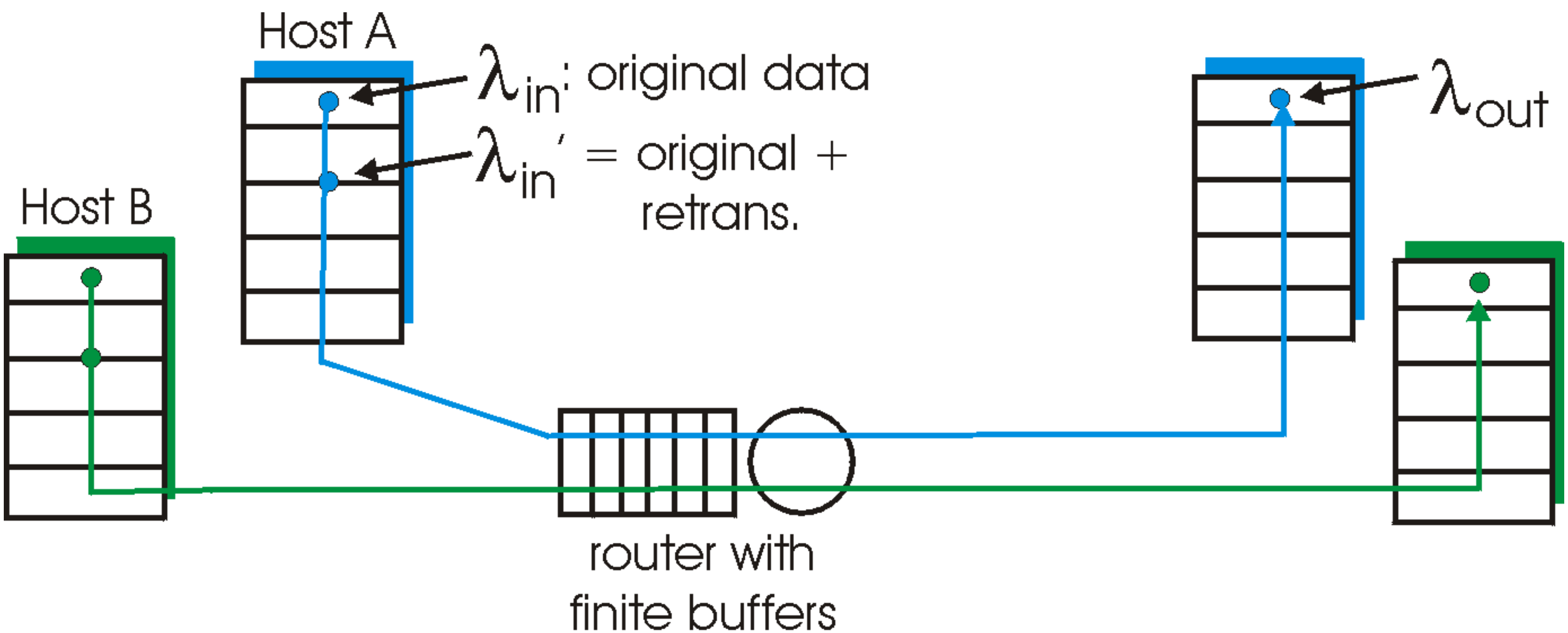



- Throughput increases with load
- Maximum total load C (Each session C/2)
- Large delays when congested
  - The load is stochastic

# Causes/costs of congestion: scenario 2

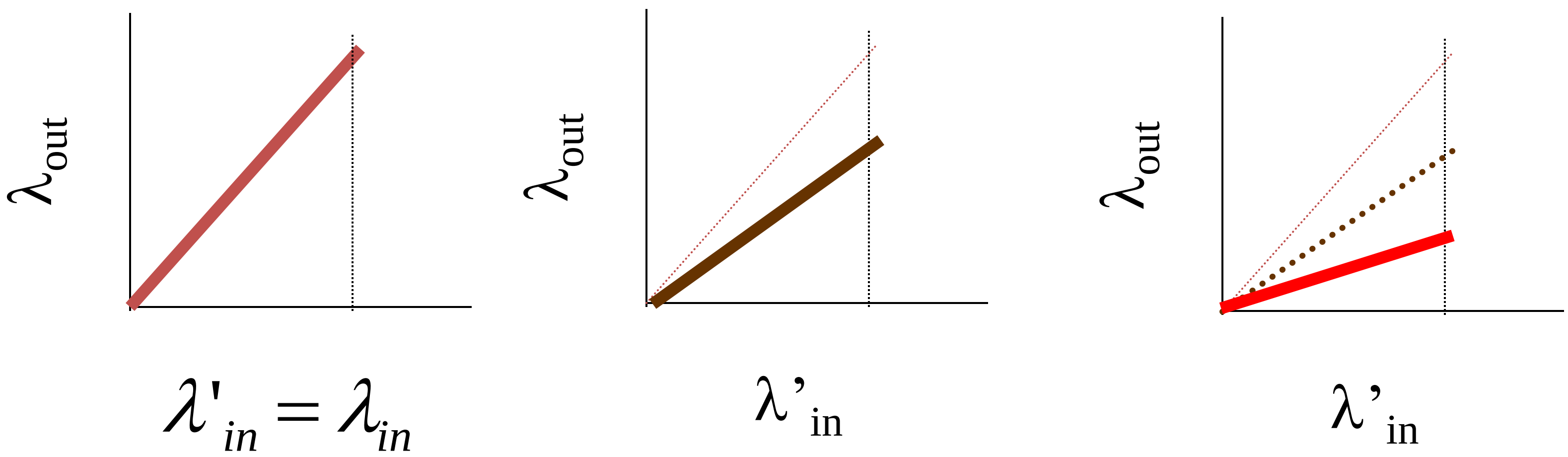
- one router, *finite* buffers
- sender retransmission of lost packet



# Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
  - Like to maximize goodput!
- "perfect" retransmission:
  - retransmit only when loss: $\lambda'_{in} > \lambda_{out}$
- Actual retransmission of delayed (not lost) packet
- makes $\lambda'_{in}$ larger (than perfect case) for same $\lambda_{out}$

# Causes/costs of congestion: scenario 2



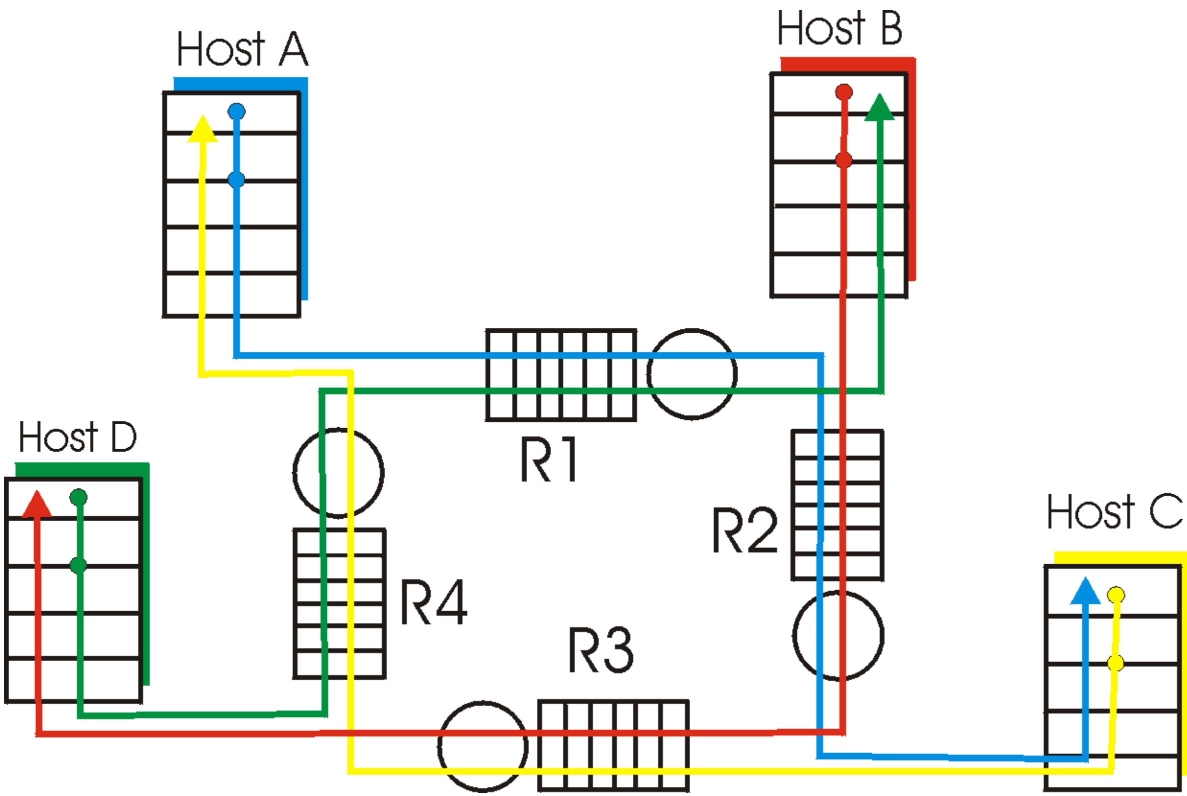

"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries (and delivers) multiple copies of pkt

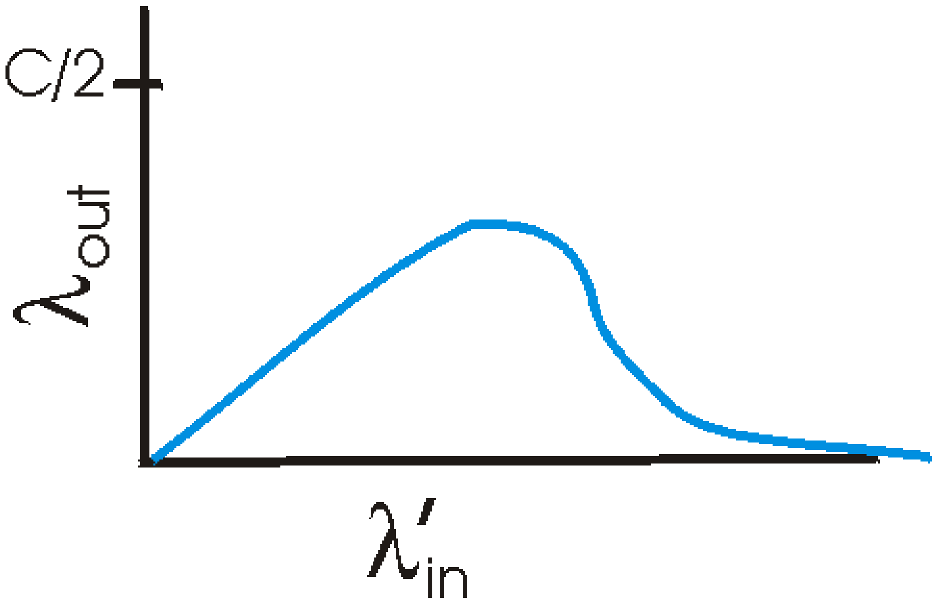
# Causes/costs of congestion: scenario 3

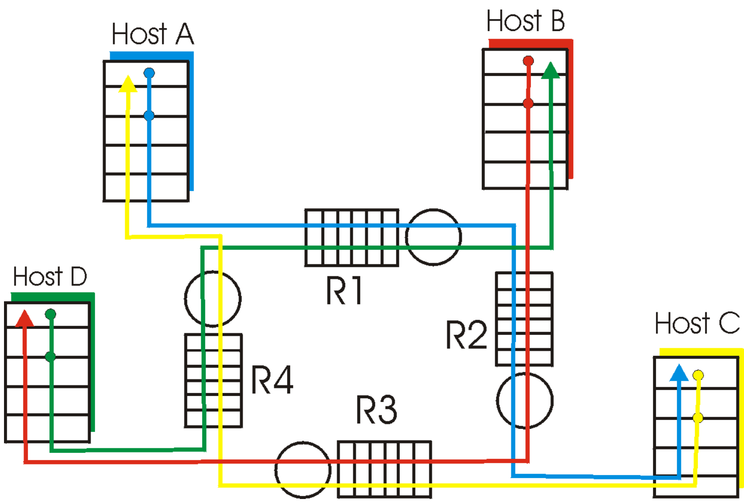
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{in}$ and $\lambda'_{in}$ increase ?



# Causes/costs of congestion: scenario 3




Another "cost" of congestion:

- when packet dropped, any "upstream" transmission capacity used for that packet was wasted!

# Approaches towards congestion control

## Two broad approaches towards congestion control:

### End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

### Network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at

# Goals of congestion control

- Throughput:
  - Maximize goodput
  - the total number of bits end-end
- Fairness:
  - Give different sessions “equal” share.
  - Max-min fairness
    - Maximize the minimum rate session.
  - Single link:
    - Capacity R
    - sessions m
    - Each sessions: R/m

# Max-min fairness

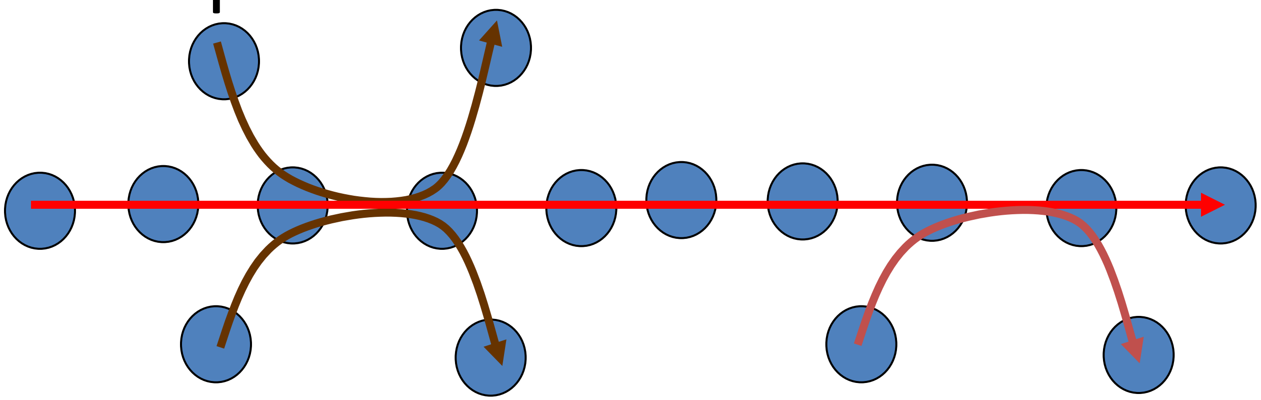
- Model: Graph G(V,e) and sessions $s_1 \dots s_m$
- For each session $s_i$ a rate $r_i$ is selected.
- The rates are a Max-Min fair allocation:
  - The allocation is maximal
    - No $r_i$ can be simply increased
  - Increasing allocation $r_i$ requires reducing
    - Some session j
    - $r_j \le r_i$
- Maximize minimum rate session.

# Max-min fairness: Algorithm

- Model: Graph G(V,e) and sessions $s_1 \ldots s_m$
- Algorithmic view:
  - For each link compute its fair share f(e).
    - Capacity / # session
  - select minimal fair share link.
  - Each session passing on it, allocate f(e).
  - Subtract the capacities and delete sessions
  - continue recessively.
- Fluid view.

# Max-min fairness

- Example



- Throughput versus fairness.

# Case study: ATM ABR congestion control

ABR: available bit rate:

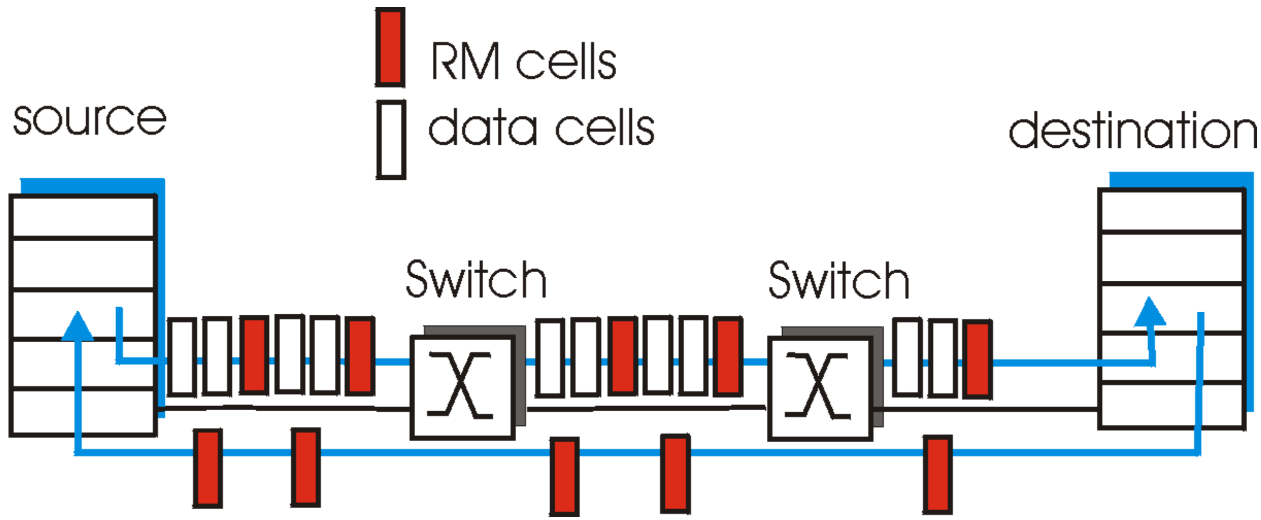
- "elastic service"
- if sender's path "underloaded":
  - sender can use available bandwidth
- if sender's path congested:
  - sender lowers rate
  - a minimum guaranteed rate
- Aim:
  - coordinate increase/decrease rate
  - avoid loss!

# Case study: ATM ABR congestion control

RM (resource management) cells:

- sent by sender, in between data cells
  - one out of every 32 cells.
- RM cells returned to sender by receiver
- Each router modifies the RM cell
- Info in RM cell set by switches
  - *"network-assisted"*
- 2 bit info.
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication (lower rate)

# Case study: ATM ABR congestion control




- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

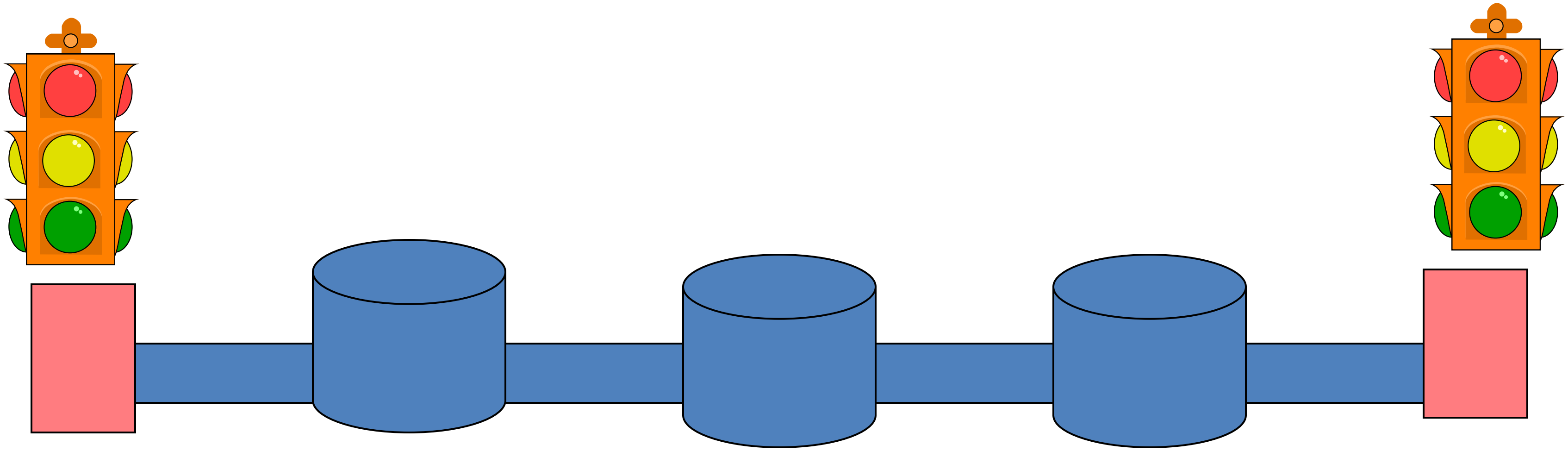
# Case study: ATM ABR congestion control

- How does the router selects its action:
  - selects a rate
  - Set congestion bits
  - Vendor dependent functionality
- Advantages:
  - fast response
  - accurate response
- Disadvantages:
  - network level design
  - Increase router tasks (load).
  - Interoperability issues.

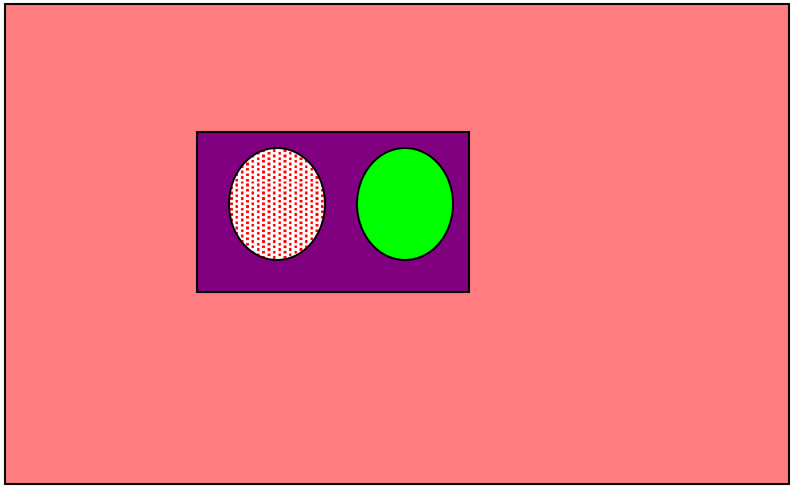
# End to end control

# End to end feedback

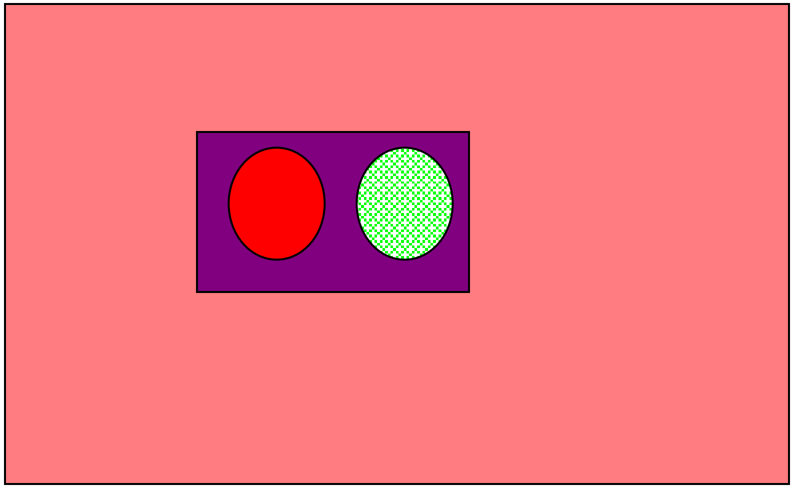
- Abstraction:
  - Alarm flag.
  - observable at the end stations



# Simple Abstraction



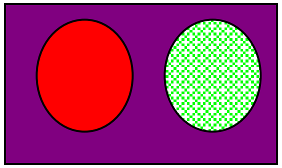
# Simple Abstraction



# Simple feedback model

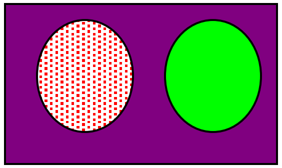
- Every RTT receive feedback
  - High Congestion

    Decrease rate



  - Low congestion

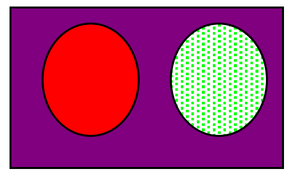
    Increase rate



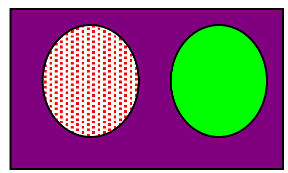
- Variable rate controls the sending rate.

# Multiplicative Update

- Congestion:
  - Rate = Rate/2

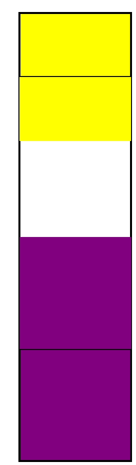


- No Congestion:
  - Rate= Rate *2



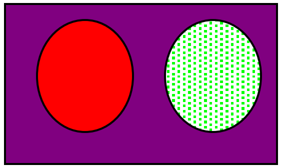
- Performance
  - Fast response
  - Un-fair:

  Ratios unchanged

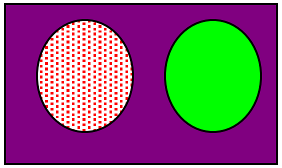


# Additive Update

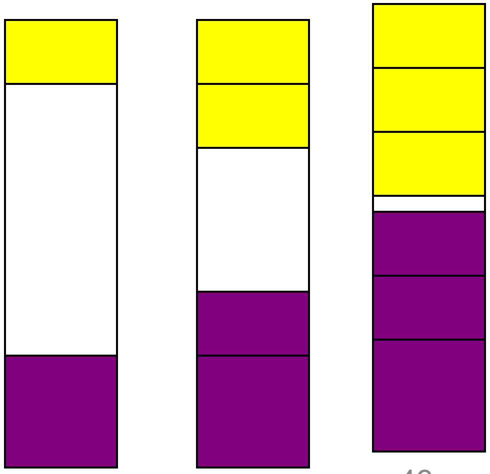
- Congestion:



  – Rate = Rate -1
- No Congestion:



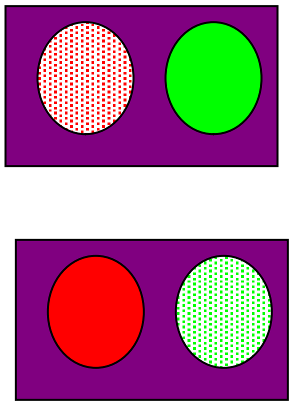
  – Rate= Rate +1
- Performance
  – Slow response
- Fairness:
  – Divides spare BW equally
  – Difference remains unchanged

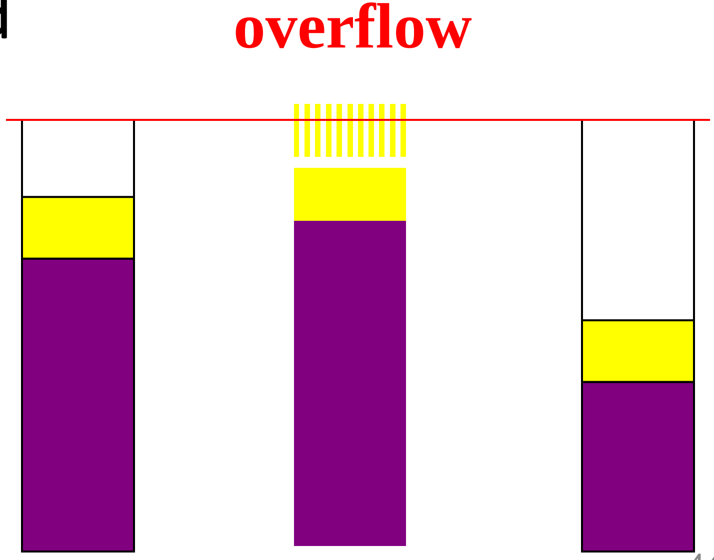


# AIMD Scheme

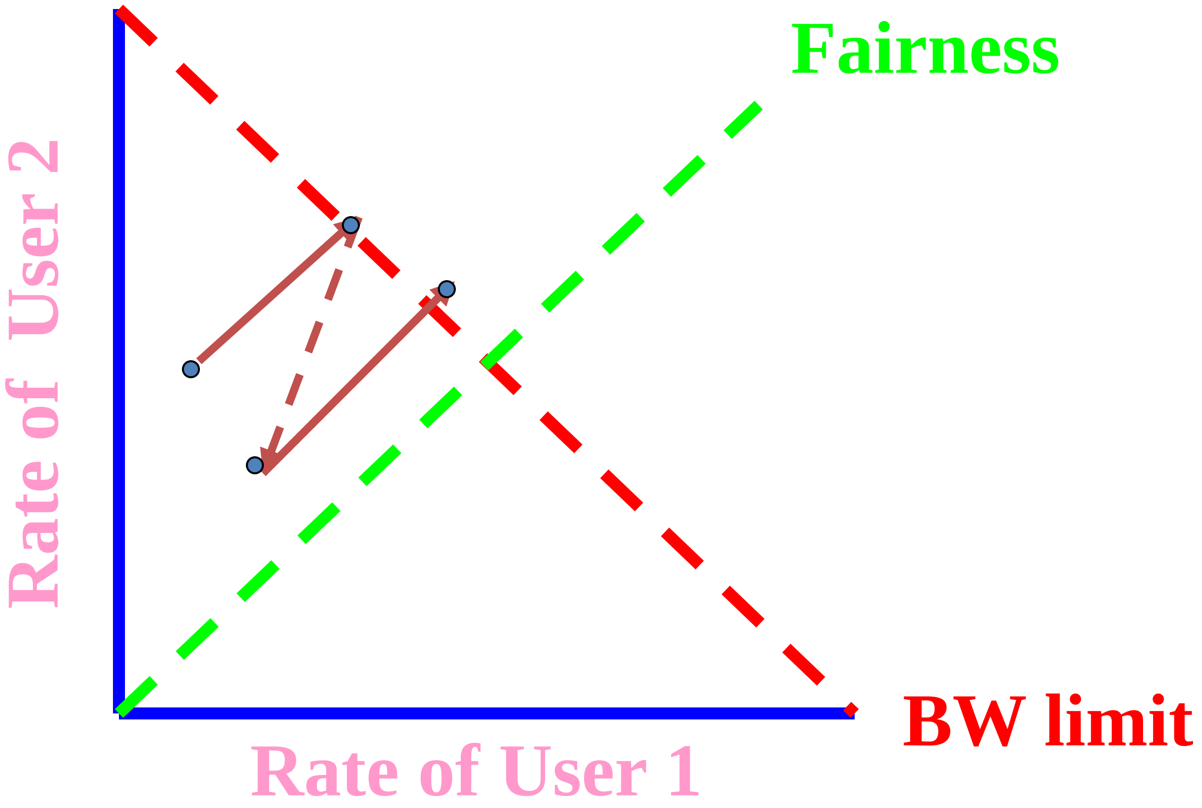
- Additive Increase
  - Fairness: ratios improves
- Multiplicative Decrease
  - Fairness: ratio unchanged
  - Fast response
- Performance:
  - Congestion -

  Fast response
  - Fairness





# AIMD: Two users, One link



# TCP: Congestion Control

# TCP Congestion Control

- Closed-loop, end-to-end, window-based congestion control
- Designed by Van Jacobson in late 1980s, based on the AIMD alg. of Dah-Ming Chu and Raj Jain
- Works well so far: the bandwidth of the Internet has increased by more than 200,000 times
- Many versions
  - TCP/Tahoe: this is a less optimized version
  - TCP/Reno: many OSs today implement Reno type congestion control
  - TCP/Vegas: not currently used

For more details: see TCP/IP illustrated; or read http://lxr.linux.no/source/net/ipv4/tcp_input.c for linux implementation



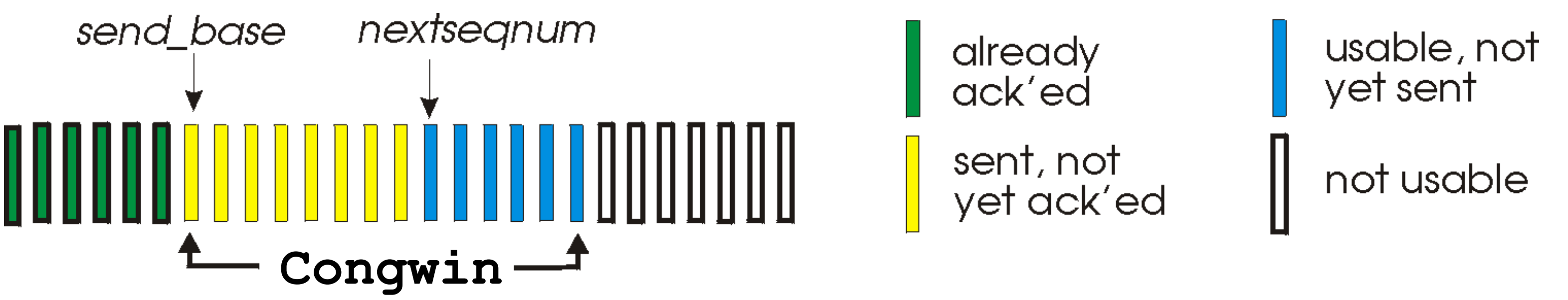
47

# TCP & AIMD: congestion

- Dynamic window size [Van Jacobson]
  - Initialization: MI
    - Slow start
  - Steady state: AIMD
    - Congestion Avoidance
- Congestion = timeout
  - TCP Taheo
- Congestion = timeout || 3 duplicate ACK
  - TCP Reno & TCP new Reno
- Congestion = higher latency

# TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, **`Congwin`**, over segments:




- w segments, each with MSS bytes sent in one RTT:

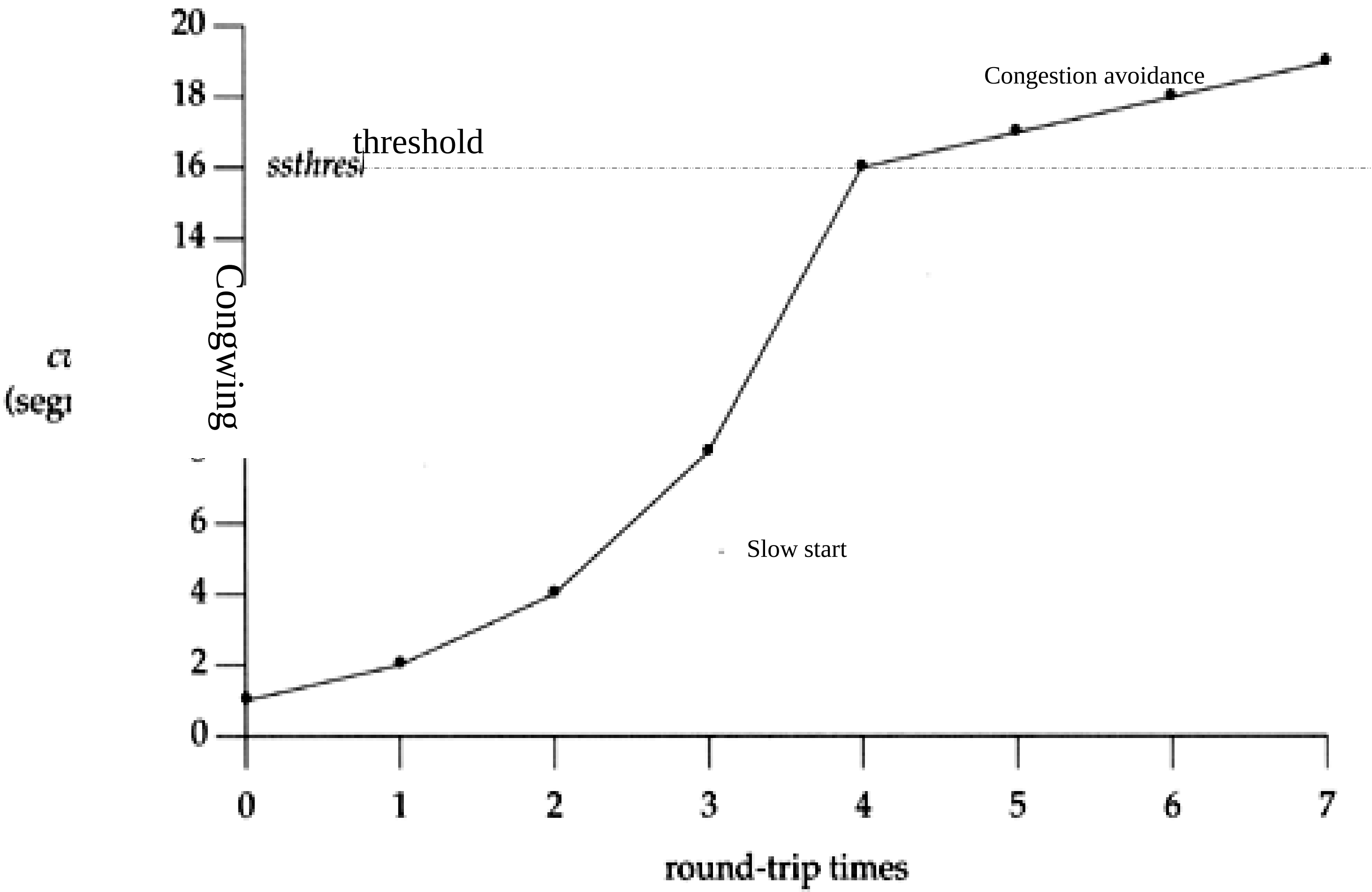
$$\boxed{\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}}$$

# TCP congestion control:

- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (**`Congwin`** as large as possible) without loss
  - *increase* **`Congwin`** until congestion (loss)
  - Congestion: *decrease* **`Congwin`**, then begin probing (increasing) again

- Basic structure:
- two "phases"
  - slow start - MI
  - congestion avoidance- AIMD
- important variables:
  - **`Congwin`**: window size
  - **`threshold`**: defines threshold between the slow start phase and the congestion avoidance phase

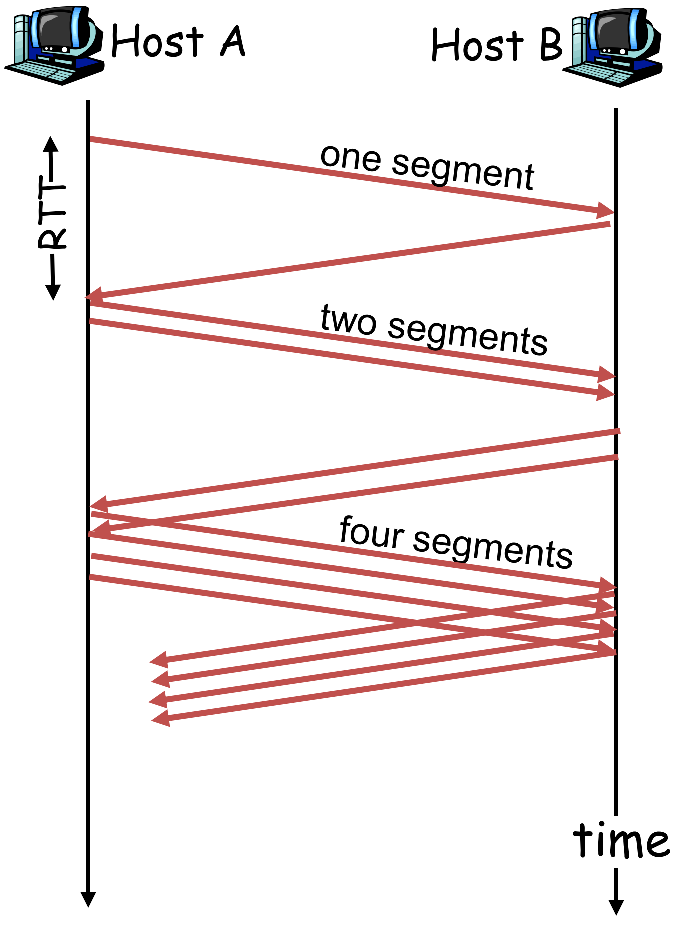
# Visualization of the Two Phases



# TCP Slowstart: MI

**Slowstart algorithm**

initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (congestion event OR
    CongWin > threshold)

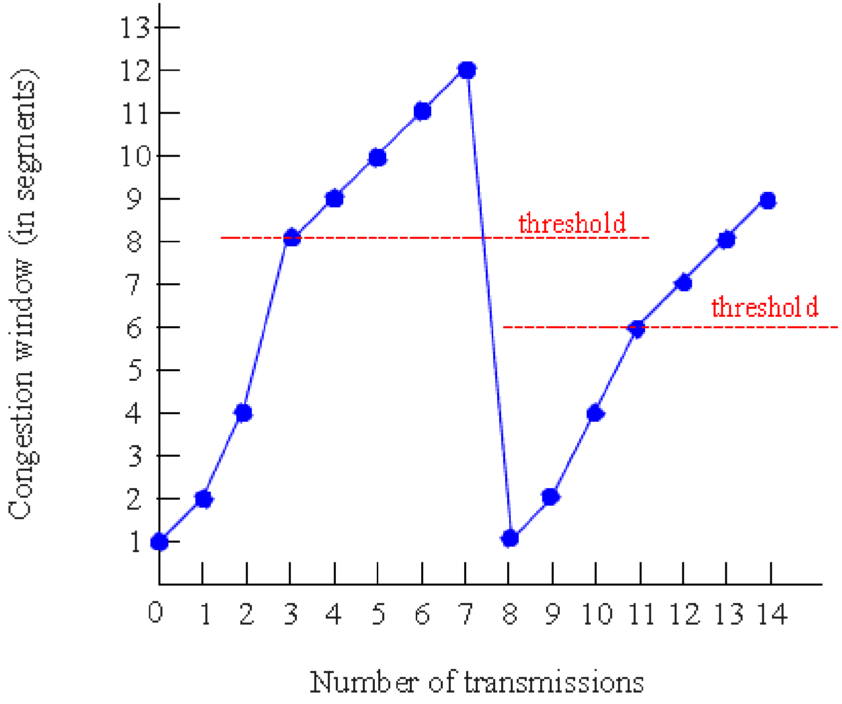



- exponential increase (per RTT) in window size (not so slow!)
- In case of timeout:
  - Threshold=CongWin/2

# TCP Taheo Congestion Avoidance

**Congestion avoidance**

```
/* slowstart is over        */
/* Congwin > threshold */
Until (timeout) { /* loss event */
  every ACK:
      Congwin += 1/Congwin
  }
threshold = Congwin/2
Congwin = 1
perform slowstart
```



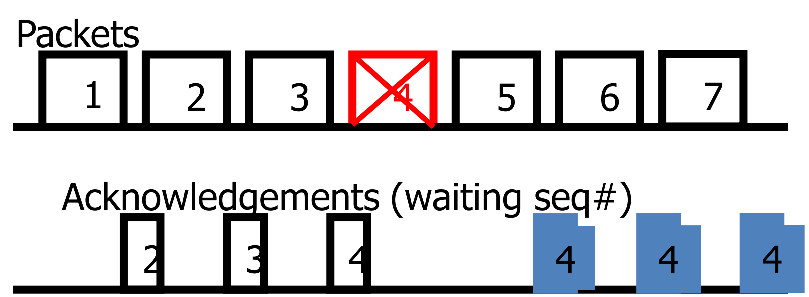

TCP Taheo

# TCP Reno

- Fast retransmit:
  - Try to avoid waiting for timeout
- Fast recovery:
  - Try to avoid slowstart.
- Single packet drop: great!

# Fast Retransmit

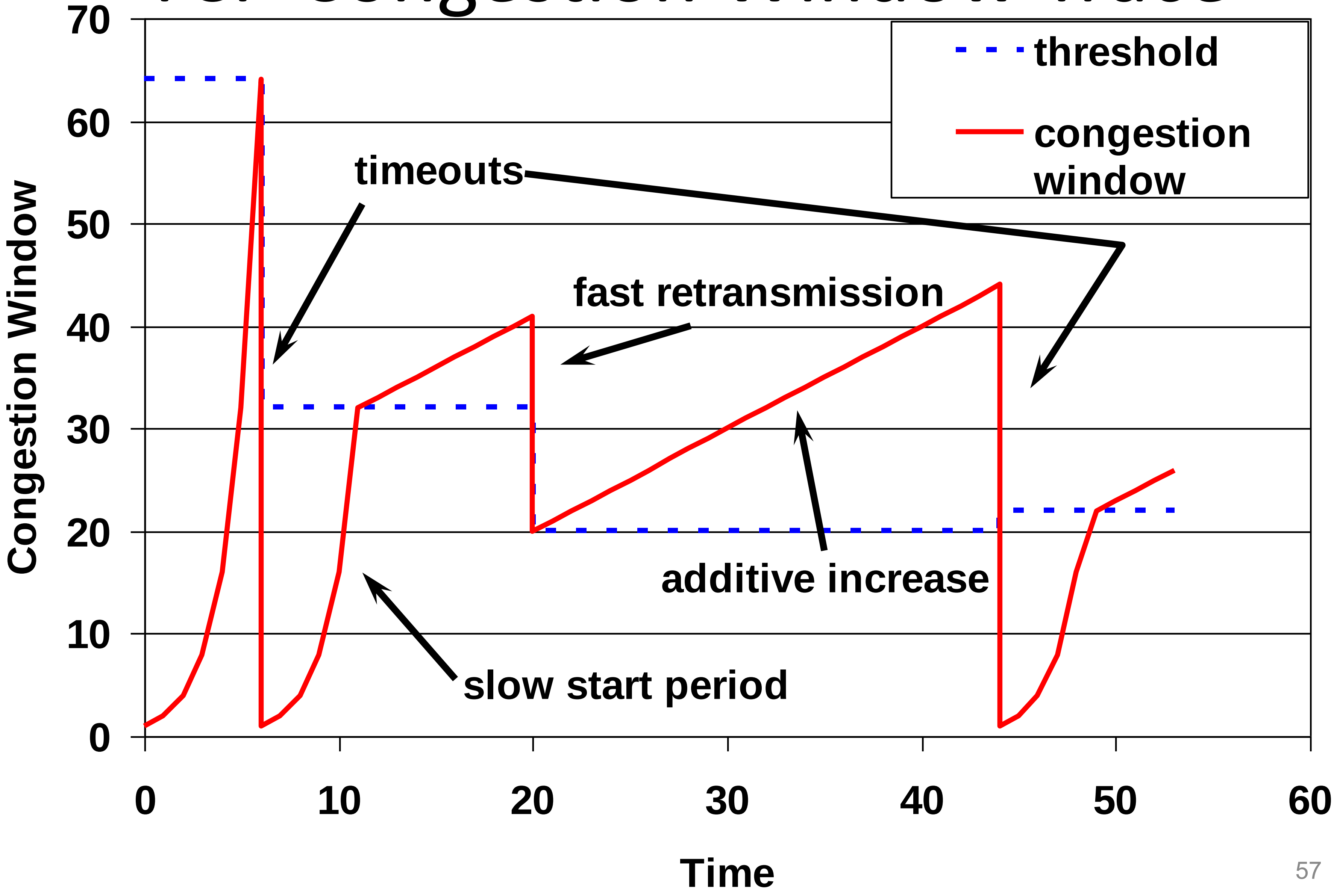
- Timeout period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - resend segment before timer expires



# Fast Recovery

- Fast recovery:
  - After retransmission do not enter slowstart.
  - Threshold = Congwin/2
  - Congwin = 3 + Congwin/2
  - Each duplicate ACK received Congwin++
  - After new ACK
    - Congwin = Threshold
    - return to congestion avoidance

# TCP Congestion Window Trace



# TCP Vegas:

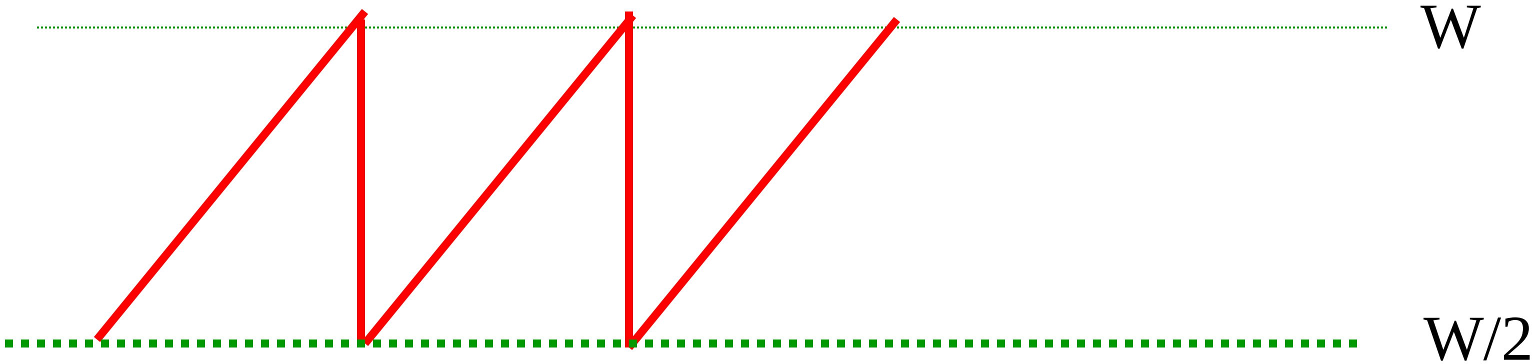
- Idea: track the RTT
  - Try to avoid packet loss
  - latency increases: lower rate
  - latency very low: increase rate
- Implementation:
  - sample_RTT: current RTT
  - Base_RTT: min. over sample_RTT
  - Expected = Congwin / Base_RTT
  - Actual = number of packets sent / sample_RTT
  - $\Delta$ =Expected - Actual

# TCP Vegas

- $\Delta$ = Expected - Actual
- Congestion Avoidance:
  - two parameters: $\alpha$ and $\beta$, $\alpha<\beta$
  - If ($\Delta < \alpha$) Congwin = Congwin +1
  - If ($\Delta > \beta$) Congwin = Congwin -1
  - Otherwise no change
  - Note: Once per RTT
- Slowstart
  - parameter $\gamma$
  - If ($\Delta > \gamma$) then move to congestion avoidance

# TCP Dynamics: Rate

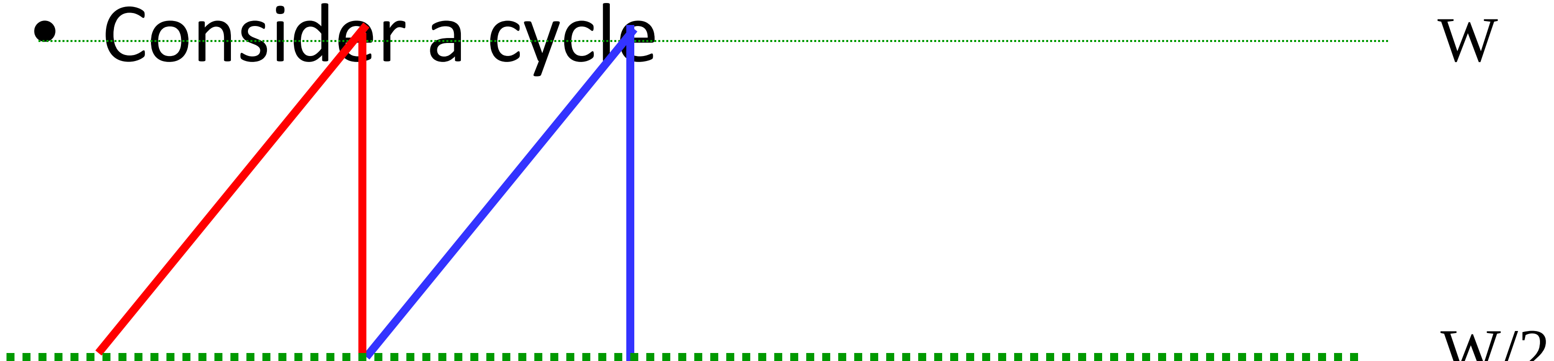
- TCP Reno with NO Fast Retransmit or Recovery
- Sending rate: Congwin*MSS / RTT
- Assume fixed RTT




- Actual Sending rate:
  - between W*MSS / RTT and (1/2) W*MSS / RTT
  - Average (3/4) W*MSS / RTT

# TCP Dynamics: Loss

- Loss rate (TCP Reno)
  - – No Fast Retransmit or Recovery
- Consider a cycle




- Total packet sent:
  - about (3/8) $W^2$ MSS/RTT = $O(W^2)$
  - One packet loss
- Loss Probability: $p=O(1/W^2)$ or $W=O(1/\sqrt{p})$

# TCP latency modeling

Q: How long does it take to receive an object from a Web server after sending a request?

- TCP connection establishment
- data transfer delay

## Notation, assumptions:

- Assume one link between client and server of rate R
- Assume: fixed congestion window, W segments
- S: MSS (bits)
- O: object size (bits)
- no retransmissions
  - no loss, no corruption

# TCP latency modeling

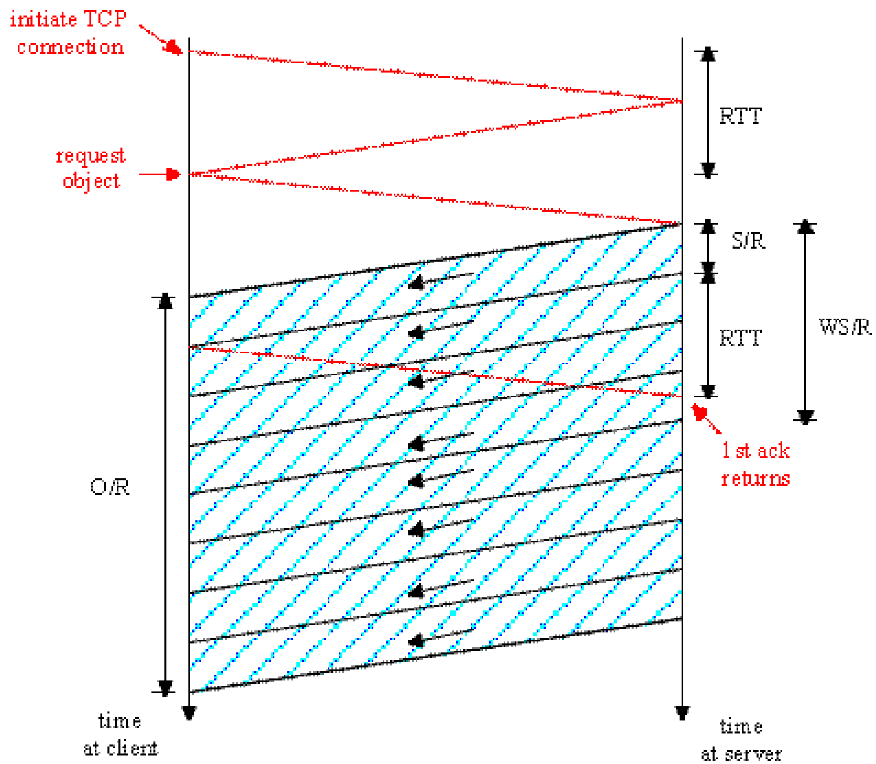
Optimal Setting: Time = O/R

Two cases to consider:

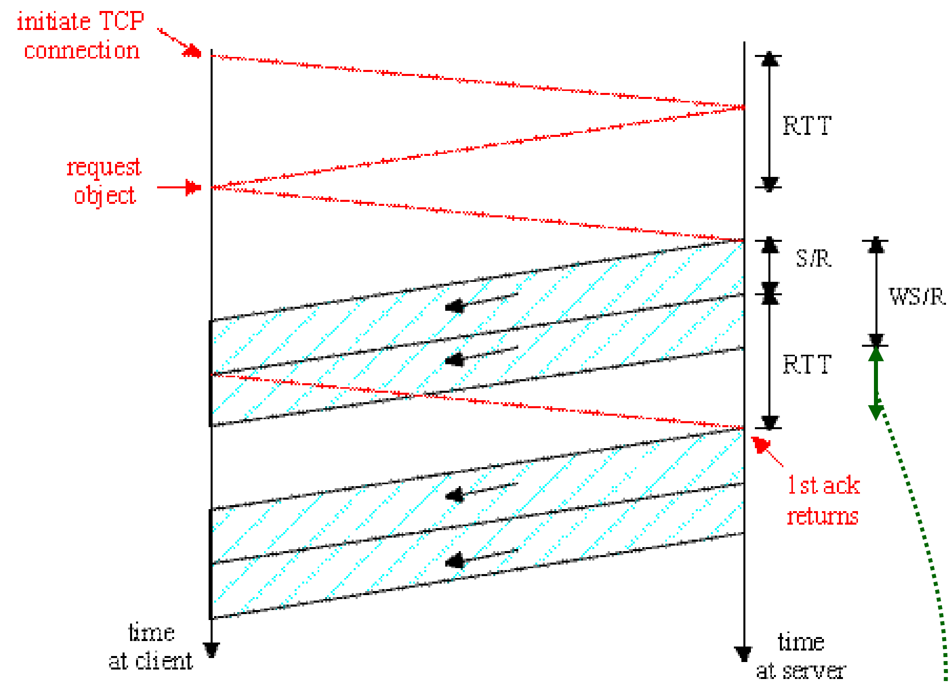
- WS/R > RTT + S/R:
  - ACK for first segment in window returns before window's worth of data sent
- WS/R < RTT + S/R:
  - wait for ACK after sending window's worth of data sent

# TCP latency Modeling

K:= O/WS




Case 1: latency = 2RTT + O/R




Case 2: latency = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

# TCP Latency Modeling: Slow Start

- Now suppose window grows according to slow start.
- Will show that the latency of one object of size O is:

$$Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

where $P$ is the number of times TCP stalls at server:

$$P = \min\{Q, K-1\}$$

- where Q is the number of times the server would stall if the object were of infinite size.

- and K is the number of windows that cover the object.

# TCP Latency Modeling: Slow Start (cont.)

Example:

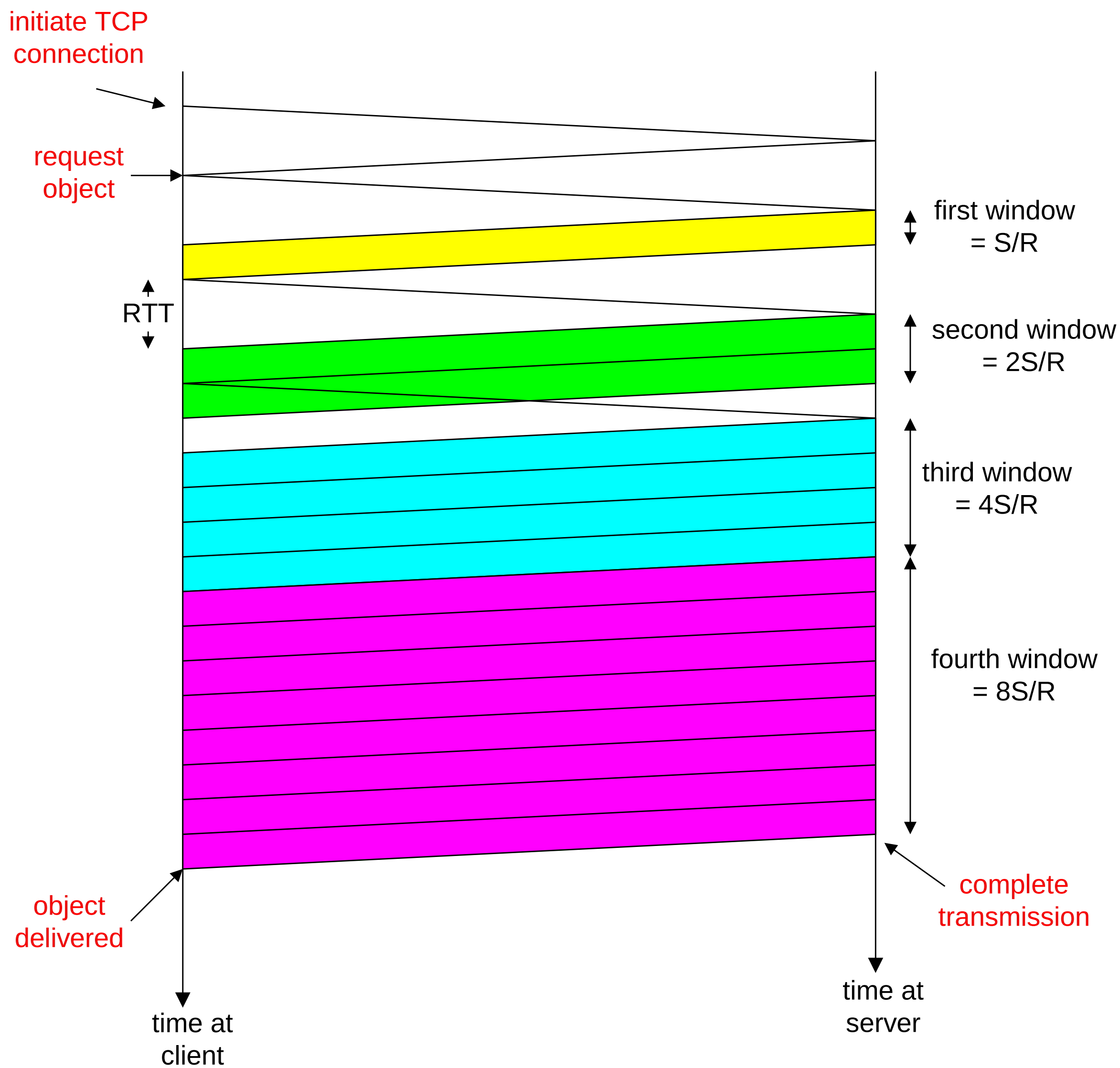
O/S = 15 segments

K = 4 windows

Q = 2

P = min{K-1,Q} = 2

Server stalls P=2 times.

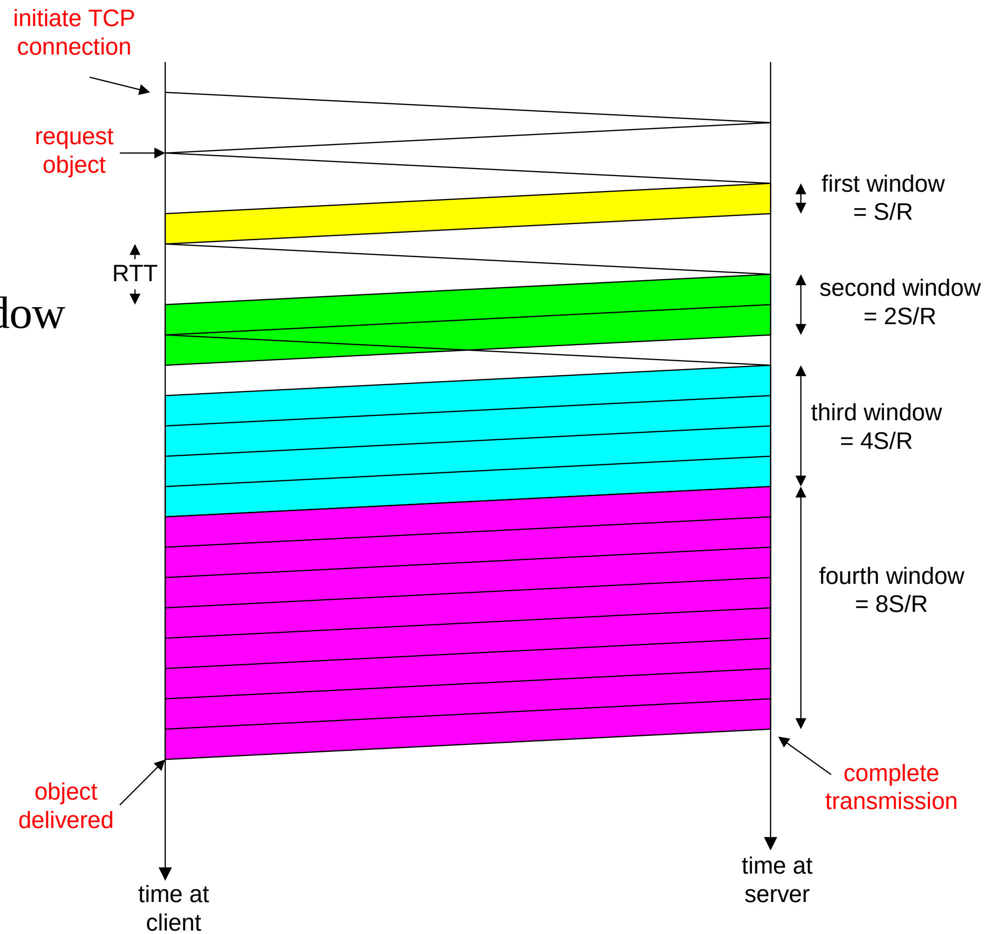


# TCP Latency Modeling: Slow Start (cont.)

$$\frac{S}{R} + RTT = \text{time from when server starts to send segment until server receives acknowledgement}$$

$$2^{k-1}\frac{S}{R} = \text{time to transmit the k}^{\text{th}} \text{ window}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} = \text{stall time after the } k^{\text{th}} \text{ window}$$




$$\begin{aligned} \text{latency} &= \frac{O}{R} + 2RTT + \sum_{p=1}^{P} stallTime_p \\ &= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right] \\ &= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R} \end{aligned}$$

# TCP:
# Flow Control

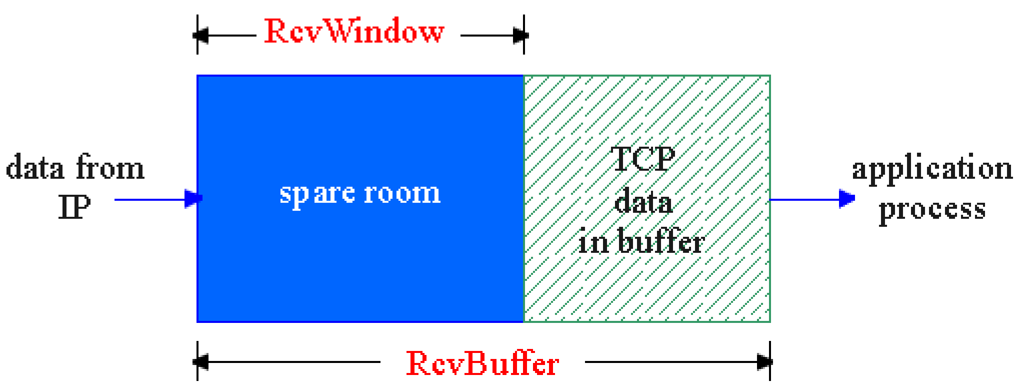
# TCP Flow Control

**flow control**

sender won't overrun receiver's buffers by transmitting too much, too fast

**RcvBuffer** = size or TCP Receive Buffer

**RcvWindow** = amount of spare room in Buffer



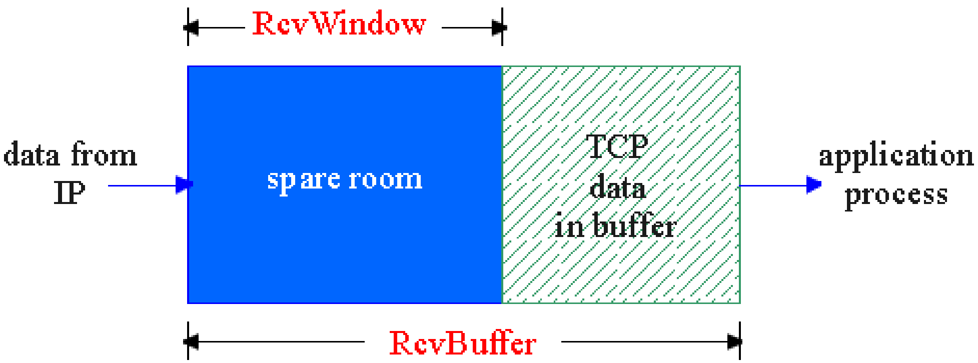

receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

– **RcvWindow field** in TCP segment

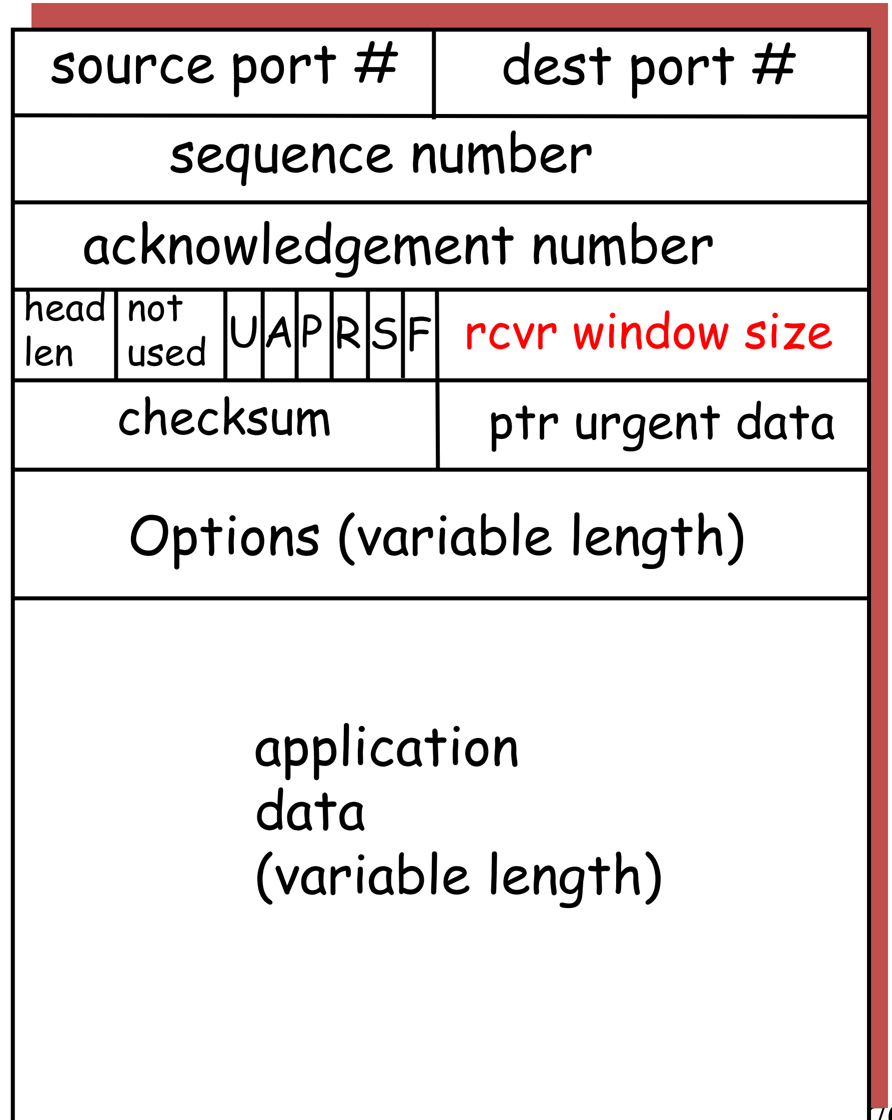
sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

# TCP Flow Control: How it Works




- spare room in buffer

= **RcvWindow**



# TCP: setting timeouts

# TCP Round Trip Time and Timeout

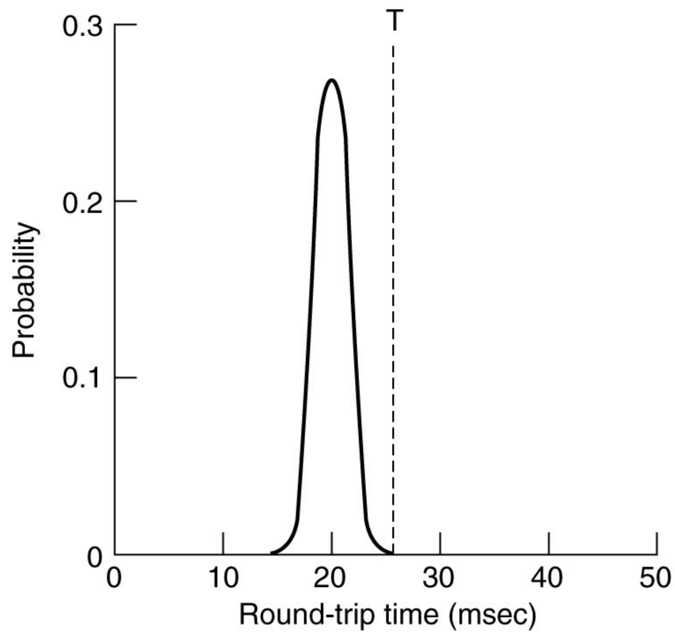
Q: how to set TCP timeout value?

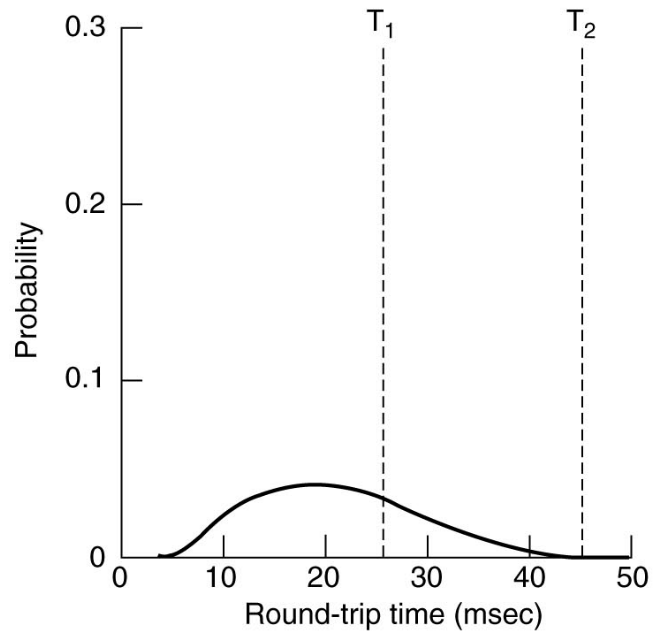
- longer than RTT
  - note: RTT will vary
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- `SampleRTT`: measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
- `SampleRTT` will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current `SampleRTT`

# High-level Idea

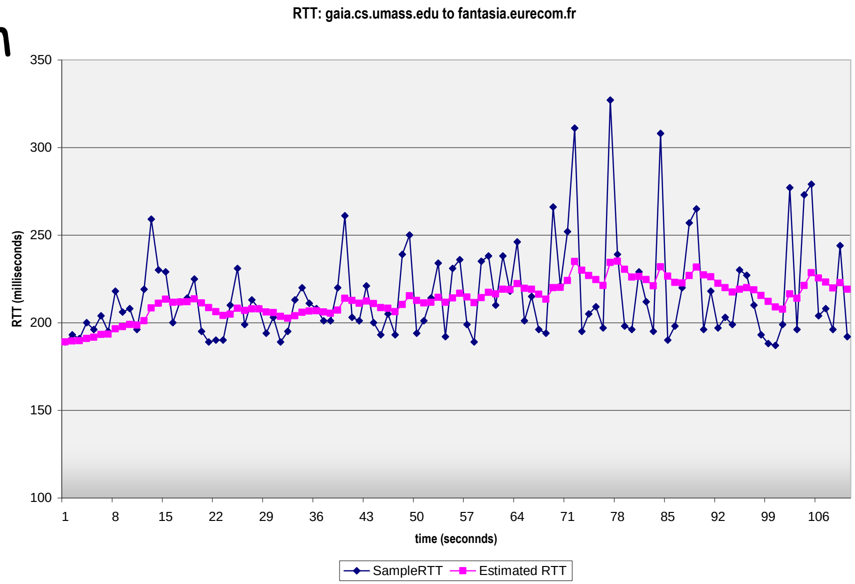



Set timeout = average + safe margin

# Estimating Round Trip Time

- **SampleRTT**: measured time from segment transmission until ACK receipt
- **SampleRTT** will vary, want a "smoother" estimated RTT
  - use several recent measurements, not just current **SampleRTT**



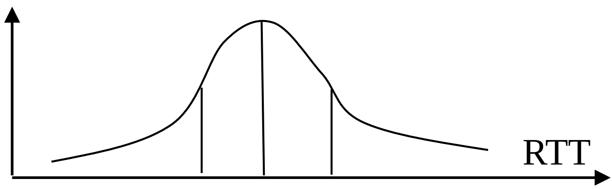

$$\texttt{EstimatedRTT} = (1-\alpha)*\texttt{EstimatedRTT} + \alpha*\texttt{SampleRTT}$$

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

# Setting Timeout

**Problem:**

- using the average of **`SampleRTT`** will generate many timeouts due to network variations




**Solution:**

- **`EstimtedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT ->`** larger safety margin

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

# An Example TCP Session