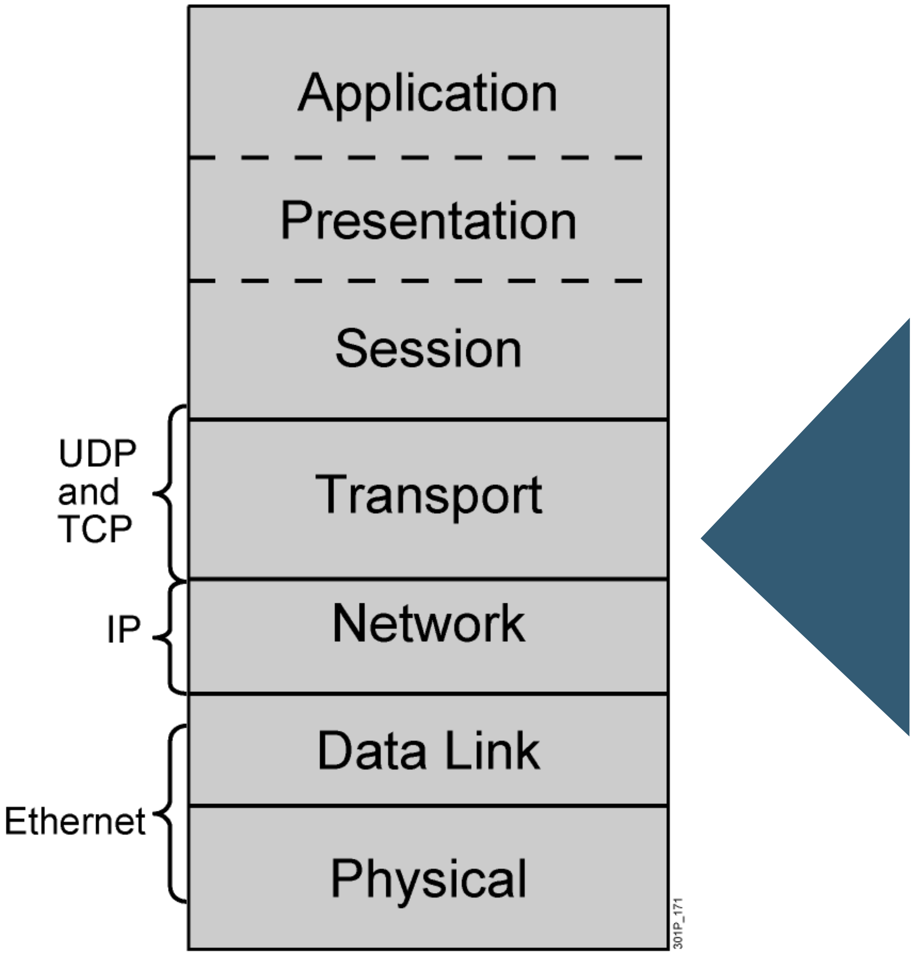
EPL606

Transport Layer

Outline

- Transport Layer Services
- TCP Overview
  - Segment structure
  - Sequence/Acknowledgement numbers
  - TCP connection management
  - RTT
  - acks, events, fast retransmit
- Flow Control
- Congestion Control
  - General causes
  - TCP cong control (slow start, AIMD)
- TCP Throughput
- TCP versions

Transport Layer

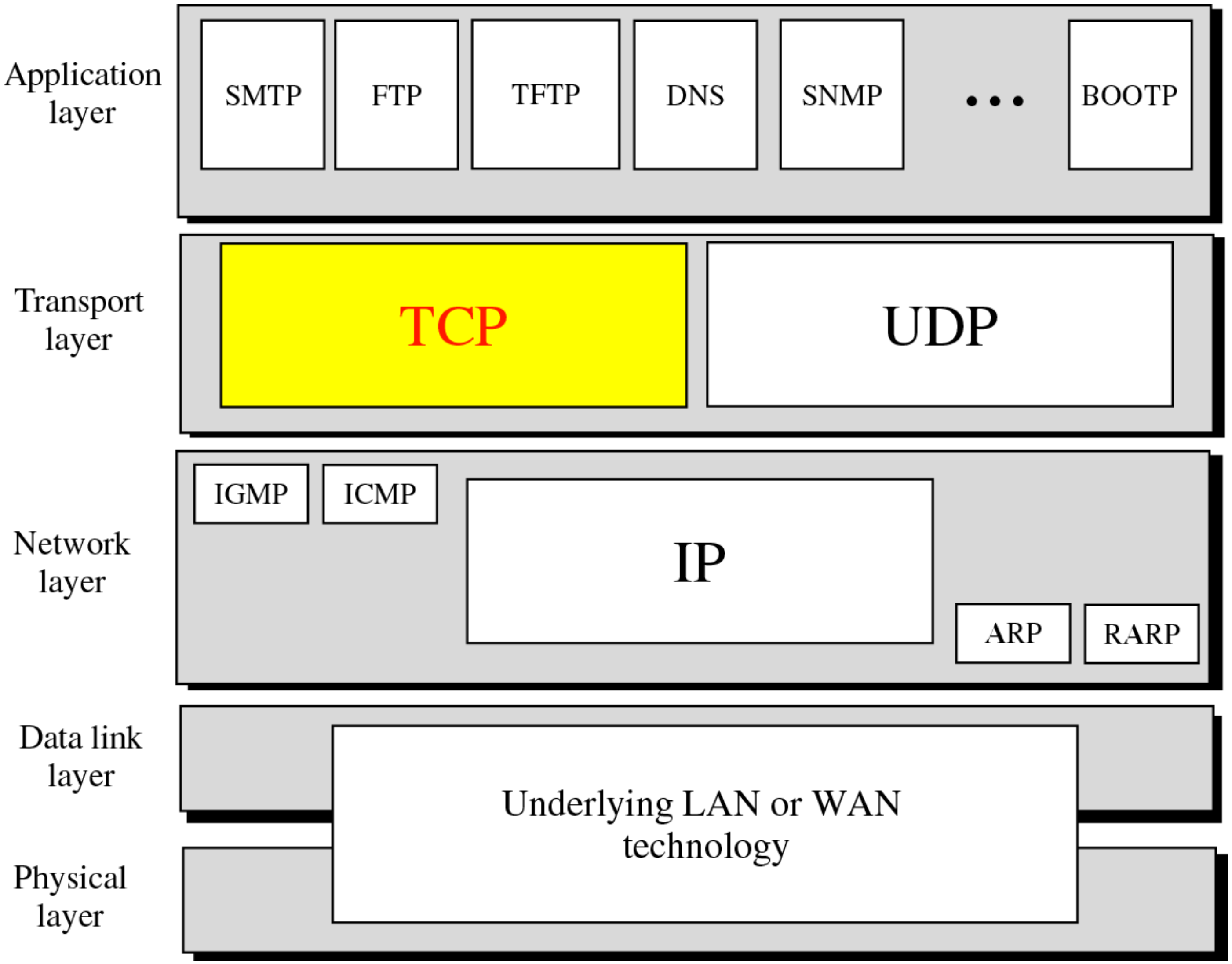


- Session multiplexing
- Segmentation
- Flow control (when required)
- Connection-oriented (when required)
- Reliability (when required)

End-to-End Protocols

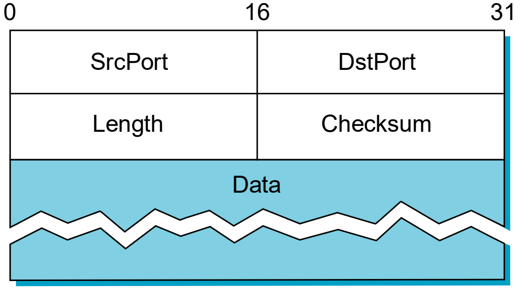
- Underlying best-effort network
  - drop messages
  - re-orders messages
  - delivers duplicate copies of a given message
  - limits messages to some finite size
  - delivers messages after an arbitrarily long delay
- Common end-to-end services
  - guarantee message delivery
  - deliver messages in the same order they are sent
  - deliver at most one copy of each message
  - support arbitrarily large messages
  - support synchronization
  - allow the receiver to flow control the sender
  - support multiple application processes on each host

Position of TCP and UDP in TCP/IP protocol suite



Simple Demultiplexor (UDP)

- Unreliable and unordered datagram service
- Adds multiplexing
- No flow control
- Endpoints identified by ports
  - servers have well-known ports
  - see /etc/services on Unix
- Header format



- Optional checksum
  - psuedo header + UDP header + data

Reliable vs. Best-Effort Comparison

	Reliable	Best-Effort
Connection Type	Connection-oriented	Connectionless
Protocol	TCP	UDP
Sequencing	Yes	No
Uses	▪ E-mail ▪ File sharing ▪ Downloading	▪ Voice streaming ▪ Video streaming

301P_957

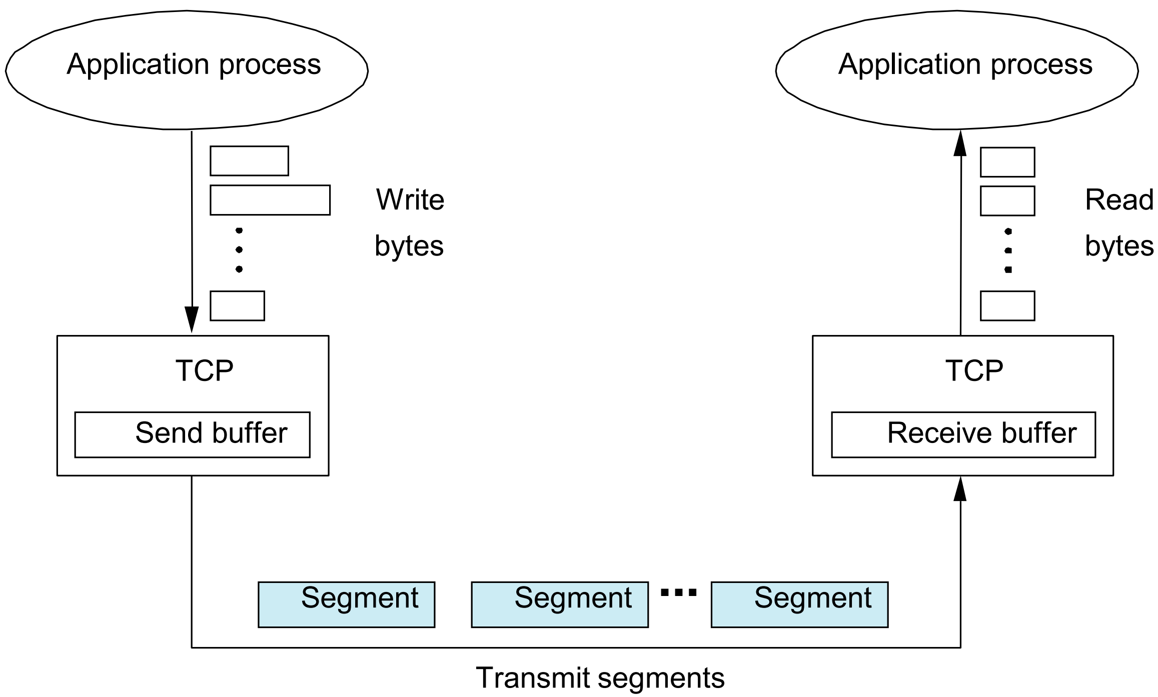
TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*

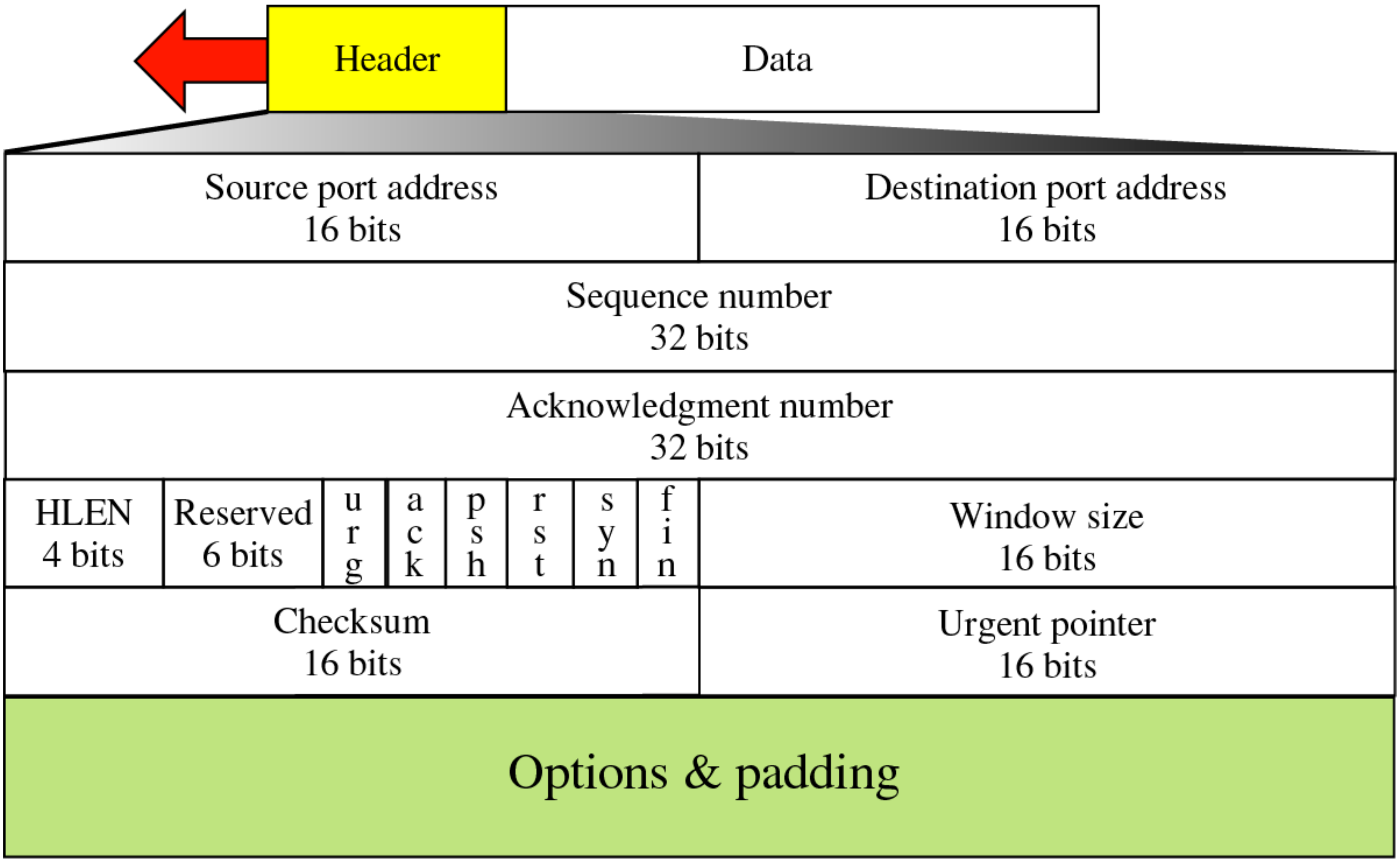
TCP Overview

- Byte-stream
  - app writes bytes
  - TCP sends *segments*
  - app reads bytes

- Flow control: keep sender from overrunning receiver
- Congestion control: keep sender from overrunning network



TCP segment structure



TCP segment structure - Control field

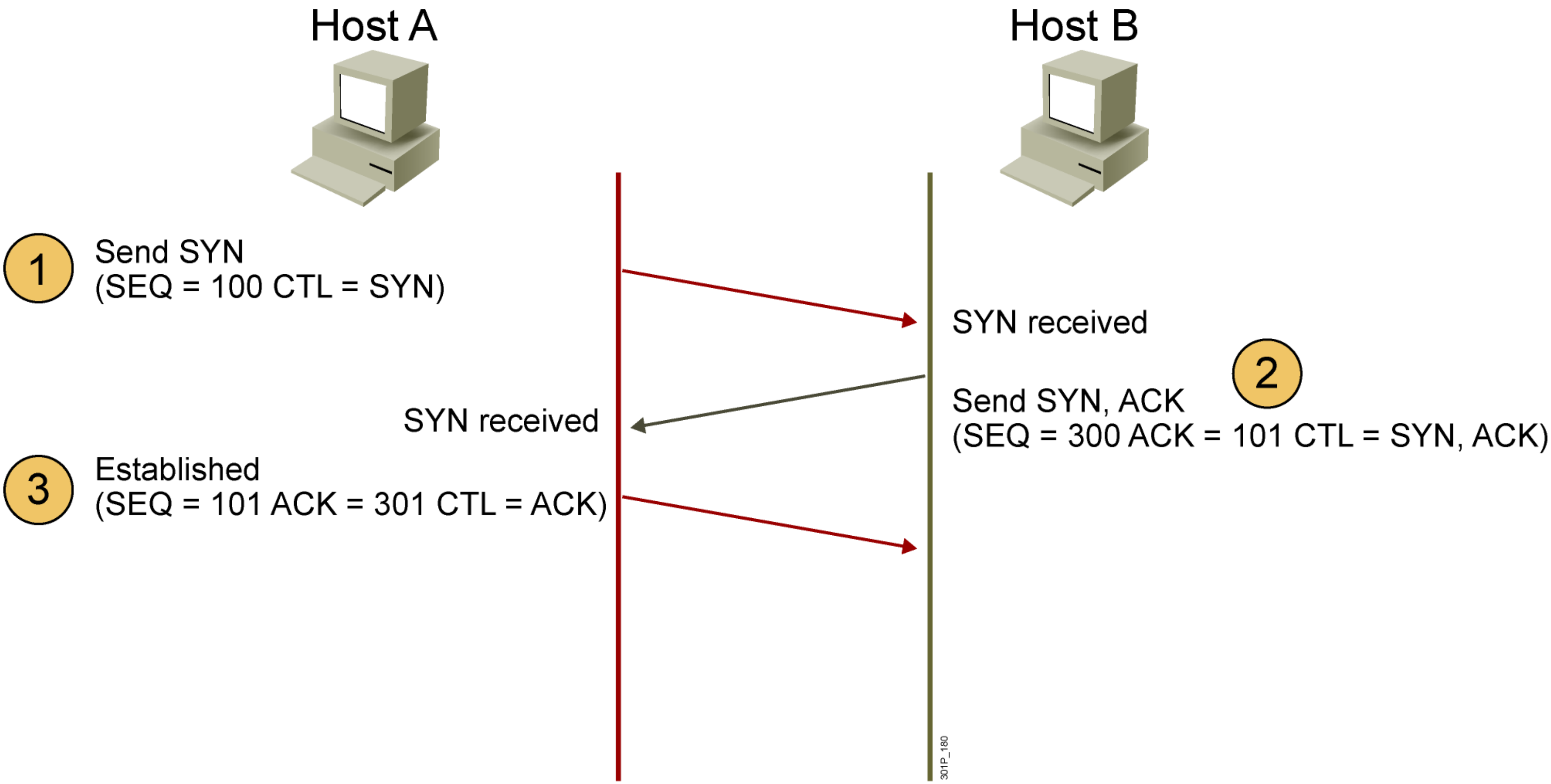
URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection

URG	ACK	PSH	RST	SYN	FIN

TCP Connection Management

- How do applications initiate a connection?
- One end (server) registers with the TCP layer instructing it to "accept" connections at a certain port
- The other end (client) initiates a "connect" request which is "accept"-ed by the server
- Recall: TCP sender, receiver establish "connection" before exchanging data segments
- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **RcvWindow**)

TCP Connection Management (cont.)



CTL = Which control bits in the TCP header are set to 1

TCP Connection Management (cont.)

Closing a connection:

client closes socket:
`clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

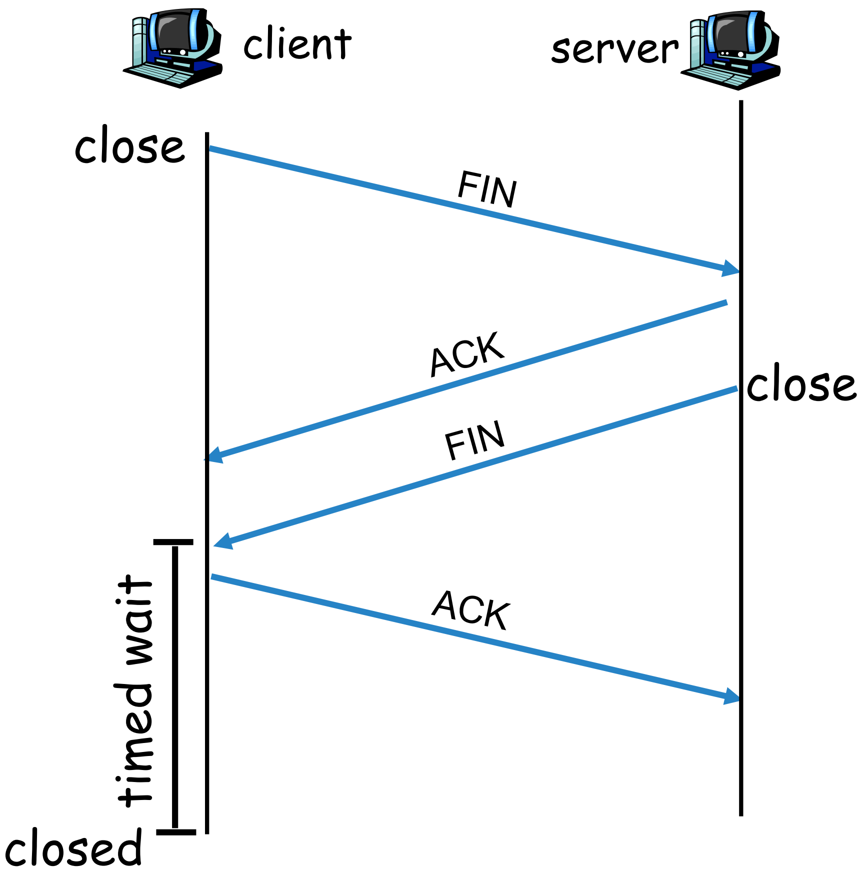
Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

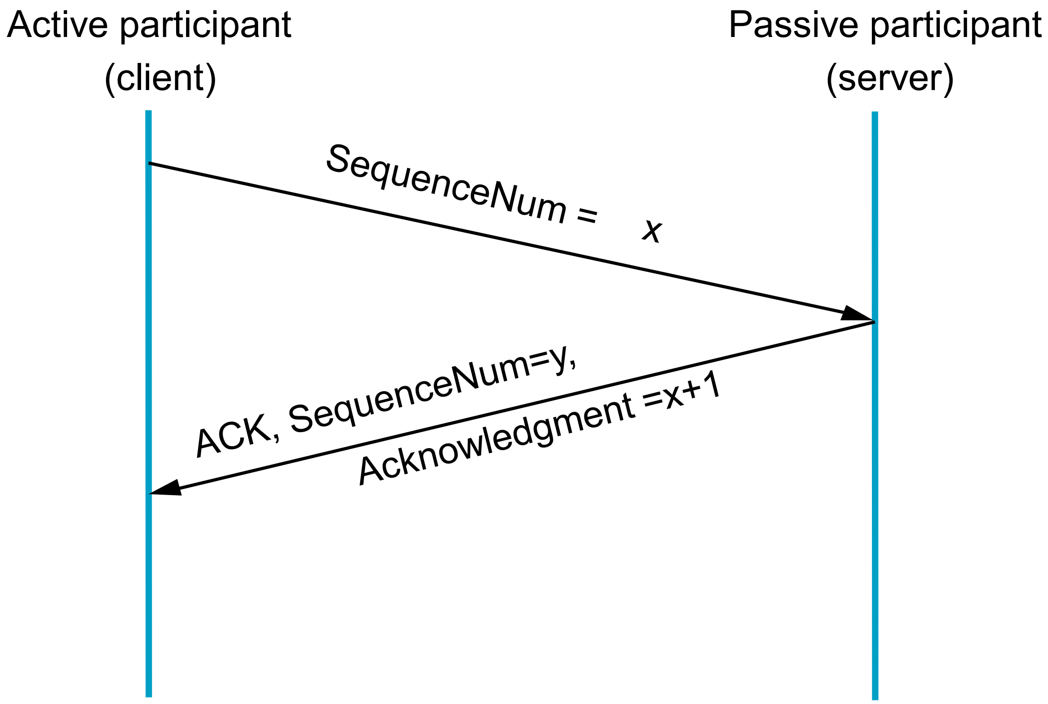
Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP seq. #'s and ACKs

- The bytes of data being transferred in each connection are numbered by TCP.
- The numbering starts with a randomly generated number.



TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

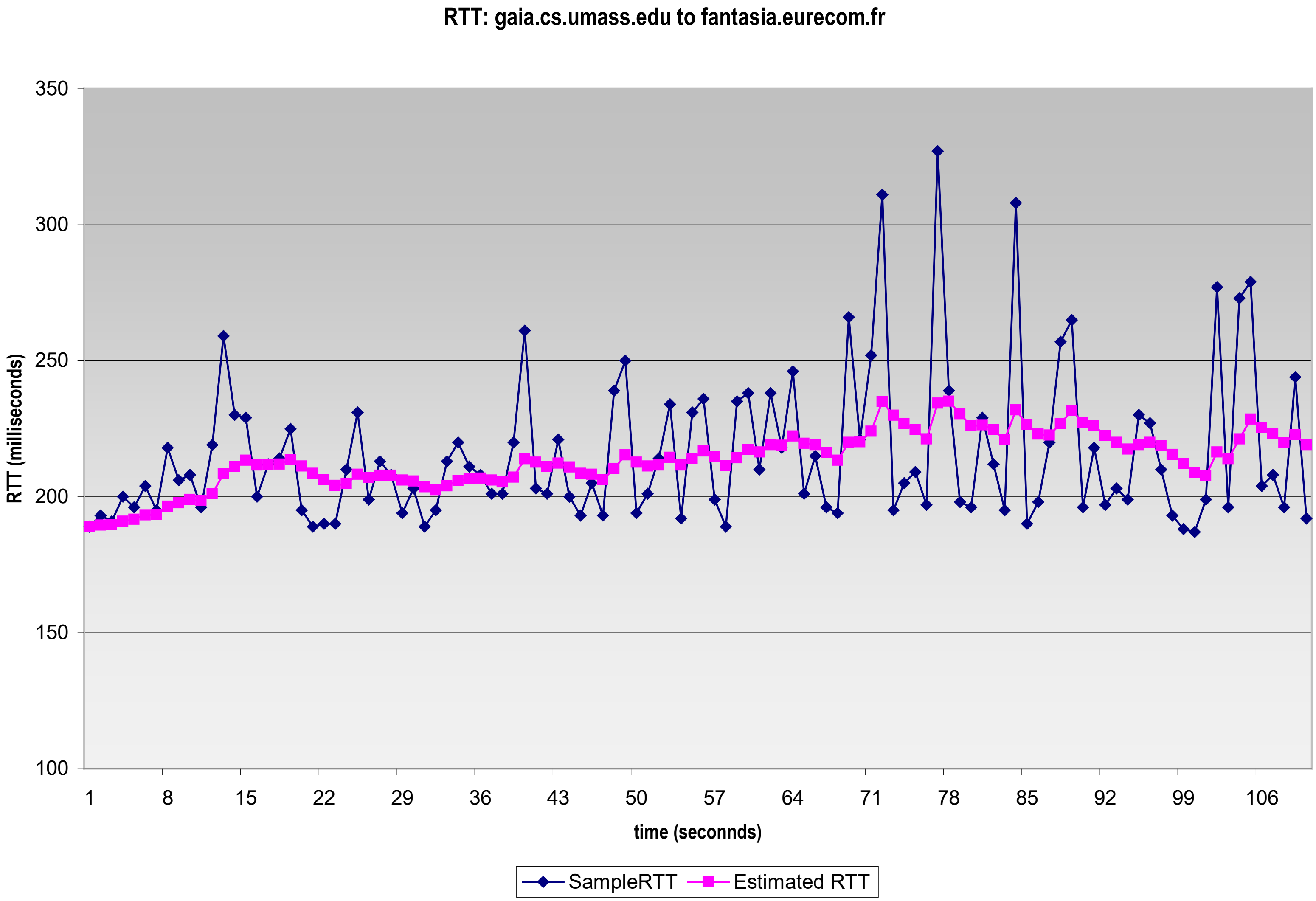
Q: how to estimate RTT?

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

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

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

Example RTT estimation:



TCP Round Trip Time and Timeout

<u>Setting the timeout</u>

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

- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, β = 0.25)
```

<u>Then set timeout interval:</u>

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

TCP reliable data transfer

- TCP creates reliable data transfer service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer

- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control

Segment Size

- Set to at most MSS (Maximum Segment Size)
  - MSS is the largest segment size that can be sent without IP fragmentation
- TCP supports push operation to allow application to explicitly send a segment

TCP sender events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
       create TCP segment with sequence number NextSeqNum
       if (timer currently not running)
            start timer
       pass segment to IP
       NextSeqNum = NextSeqNum + length(data)

   event: timer timeout
       retransmit not-yet-acknowledged segment with
            smallest sequence number
       start timer

   event: ACK received, with ACK field value of y
       if (y > SendBase) {
            SendBase = y
           if (there are currently not-yet-acknowledged segments)
                start timer
           }

 }  /* end of loop forever */
```

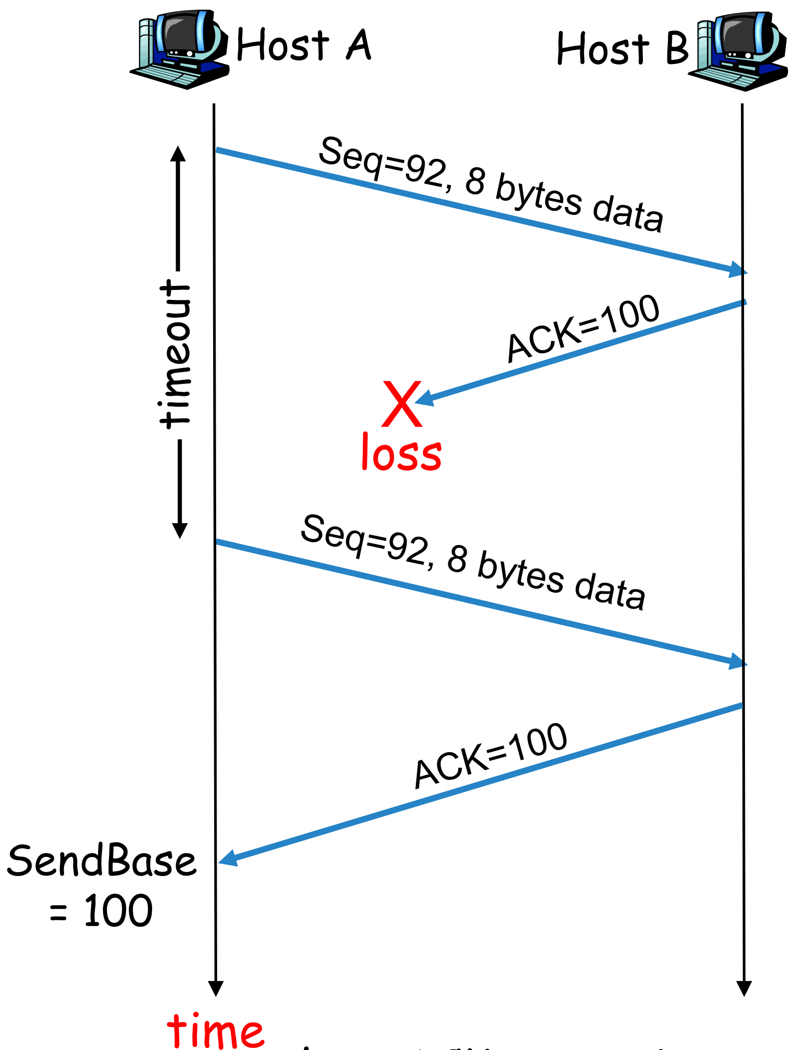
Comment:

- SendBase-1: last cumulatively ack'ed byte

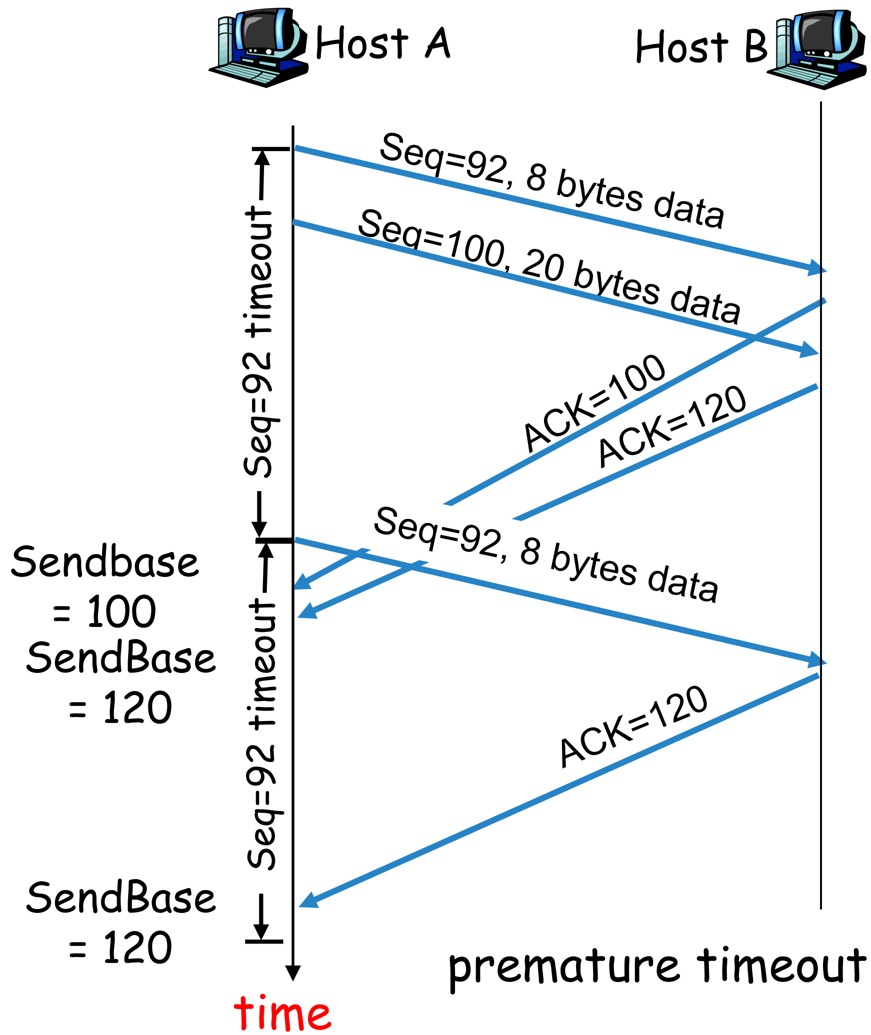
Example:

- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: retransmission scenarios

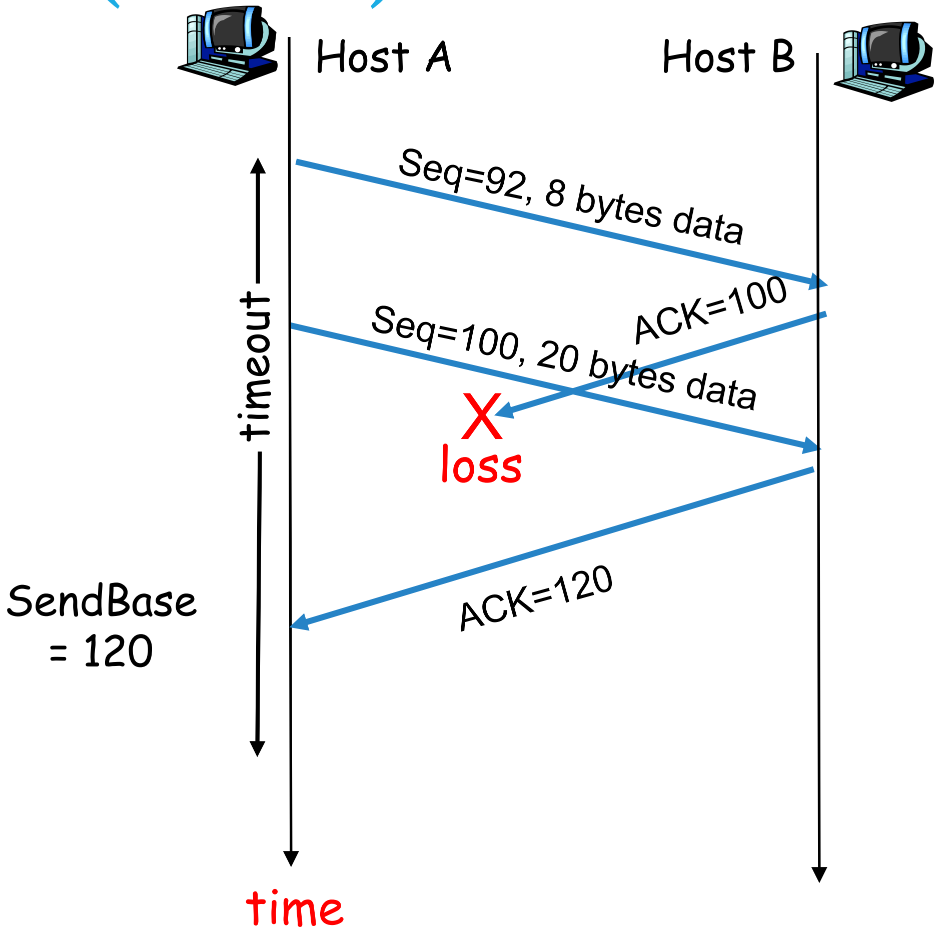


lost ACK scenario



premature timeout

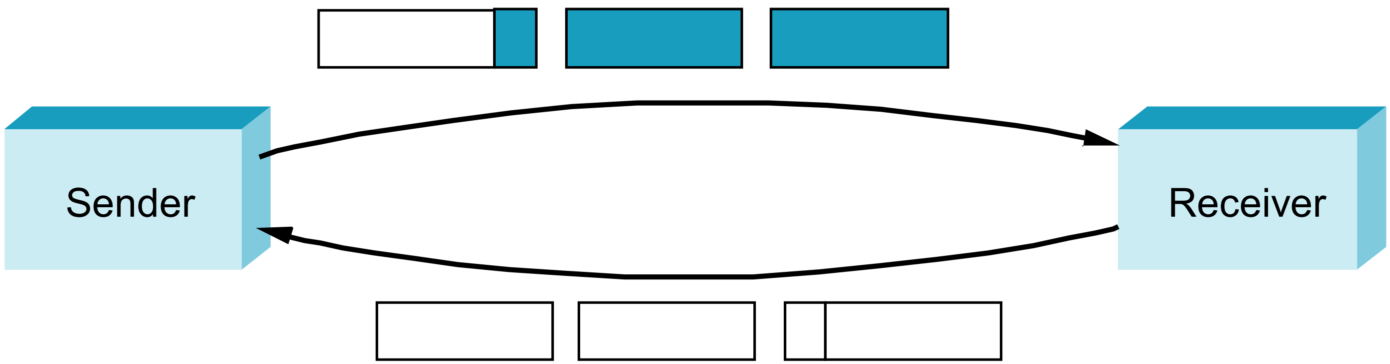
TCP retransmission scenarios (more)



Cumulative ACK scenario

Silly Window Syndrome

- How aggressively does sender exploit open window?



- Receiver-side solutions
  - after advertising zero window, wait for space equal to a maximum segment size (MSS)
  - delayed acknowledgements

Fast Retransmit

- Time-out 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:
  - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
            }
          else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                 }
```

a duplicate ACK for already ACKed segment

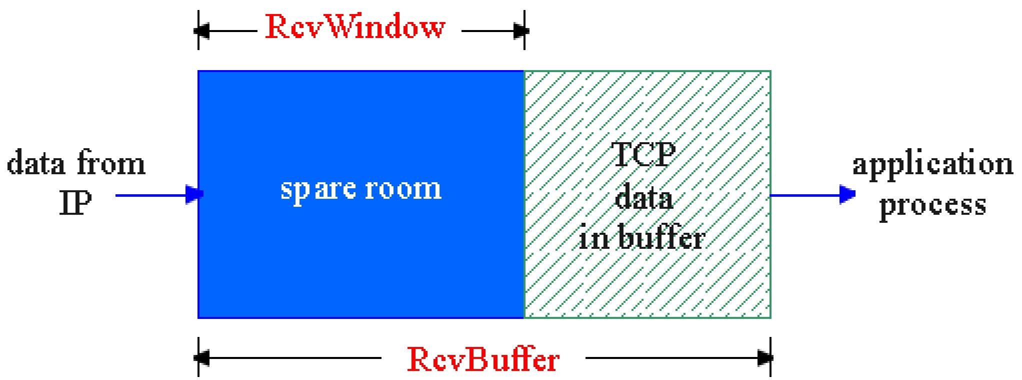
fast retransmit

Outline

- Transport Layer Services
- TCP Overview
  - Segment structure
  - Seq nums
  - Tcp connection management
  - RTT
  - Rtd: acks, events, fast retransmit
- Flow Control
- Congestion Control
  - General causes
  - Tcp cong control (slow start, AIMD)
- TCP Throughput
- TCP versions

TCP Flow Control

- receive side of TCP connection has a receive buffer:



- speed-matching service: matching the send rate to the receiving app's drain rate
- app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

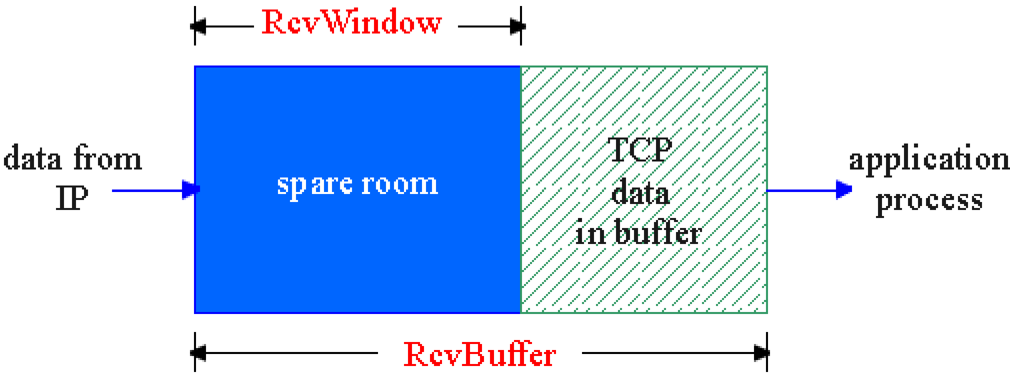
TCP Flow control: how it works

- spare room in buffer

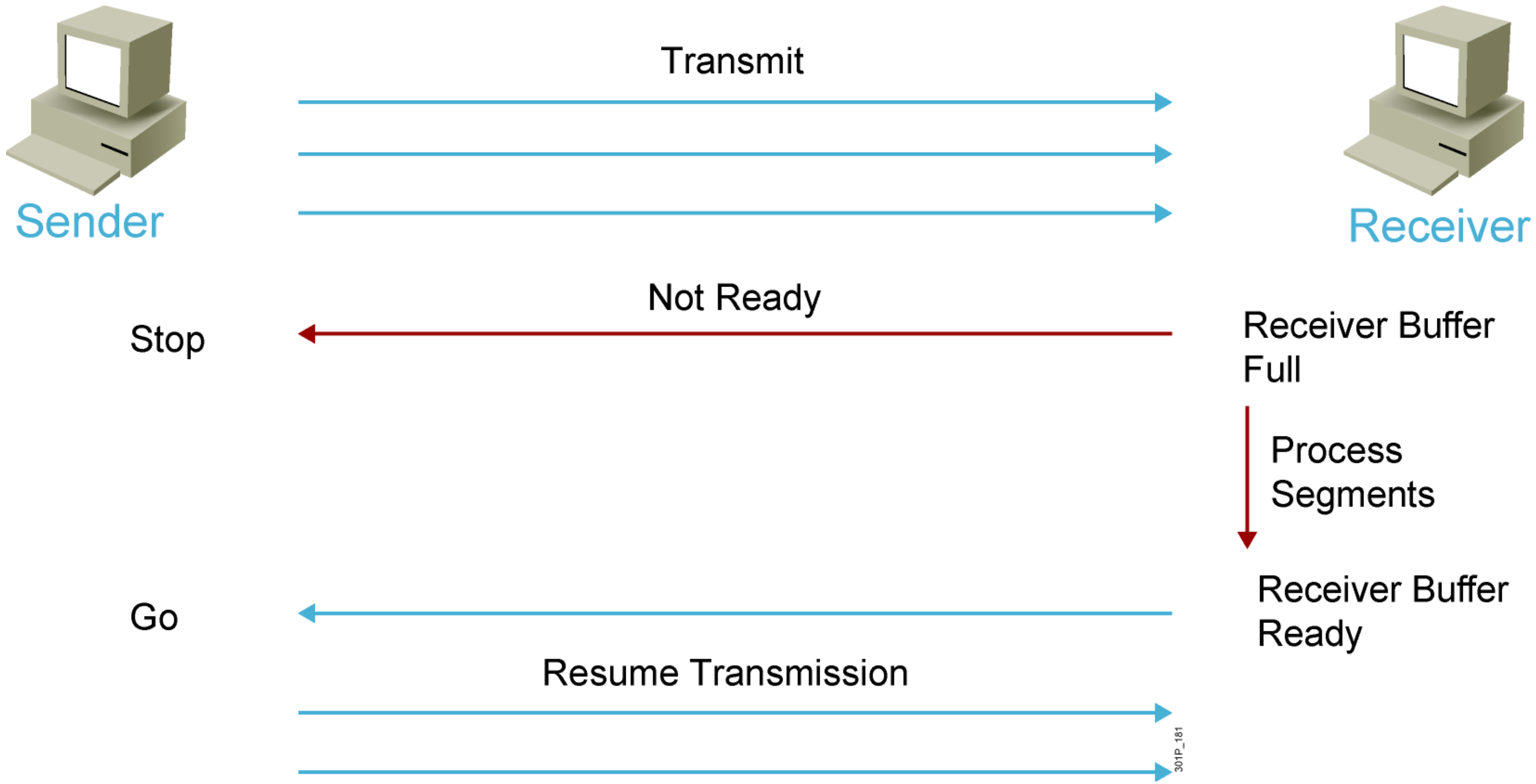
= RcvWindow

= RcvBuffer-[LastByteRcvd - LastByteRead]

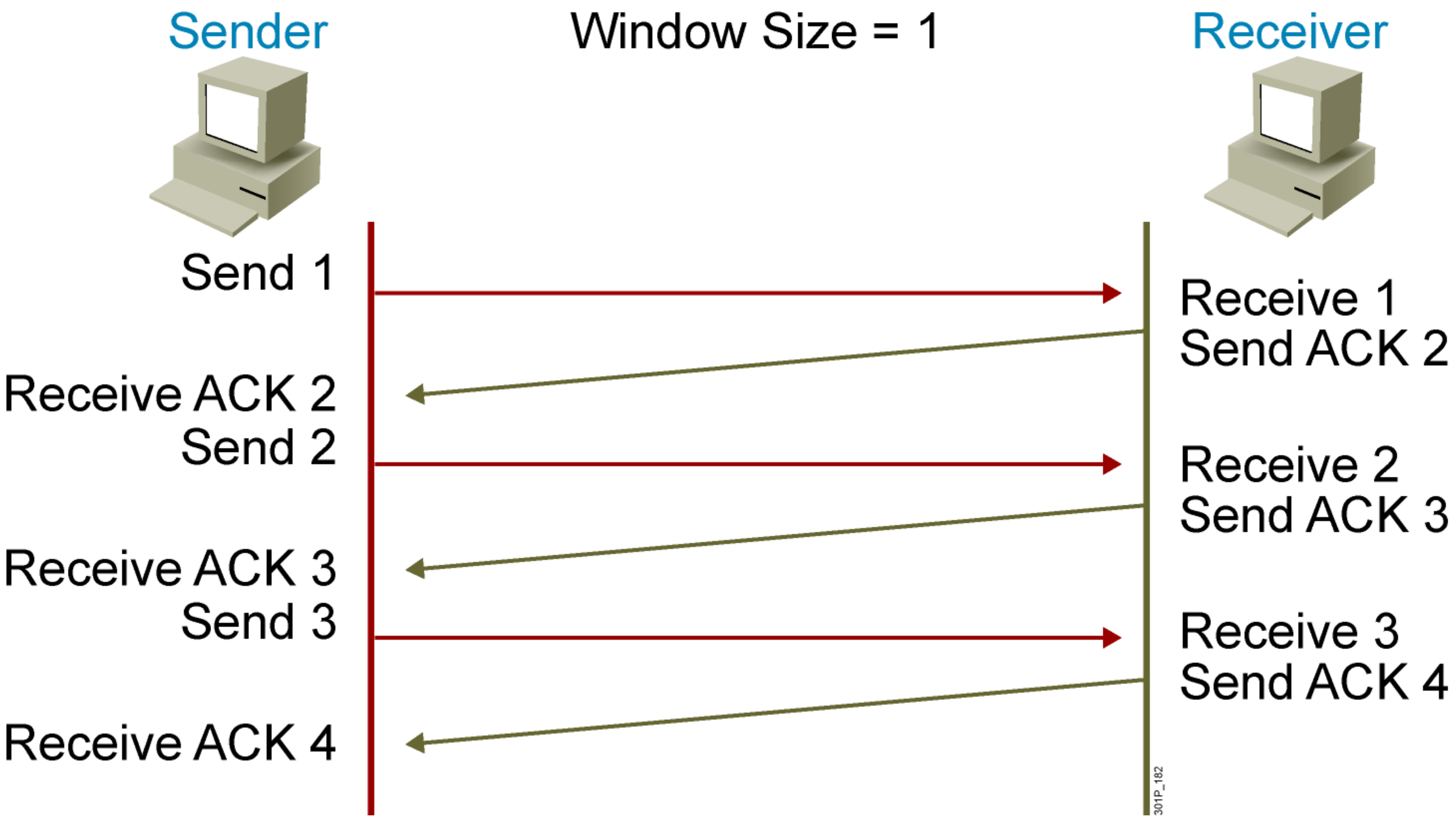
- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow



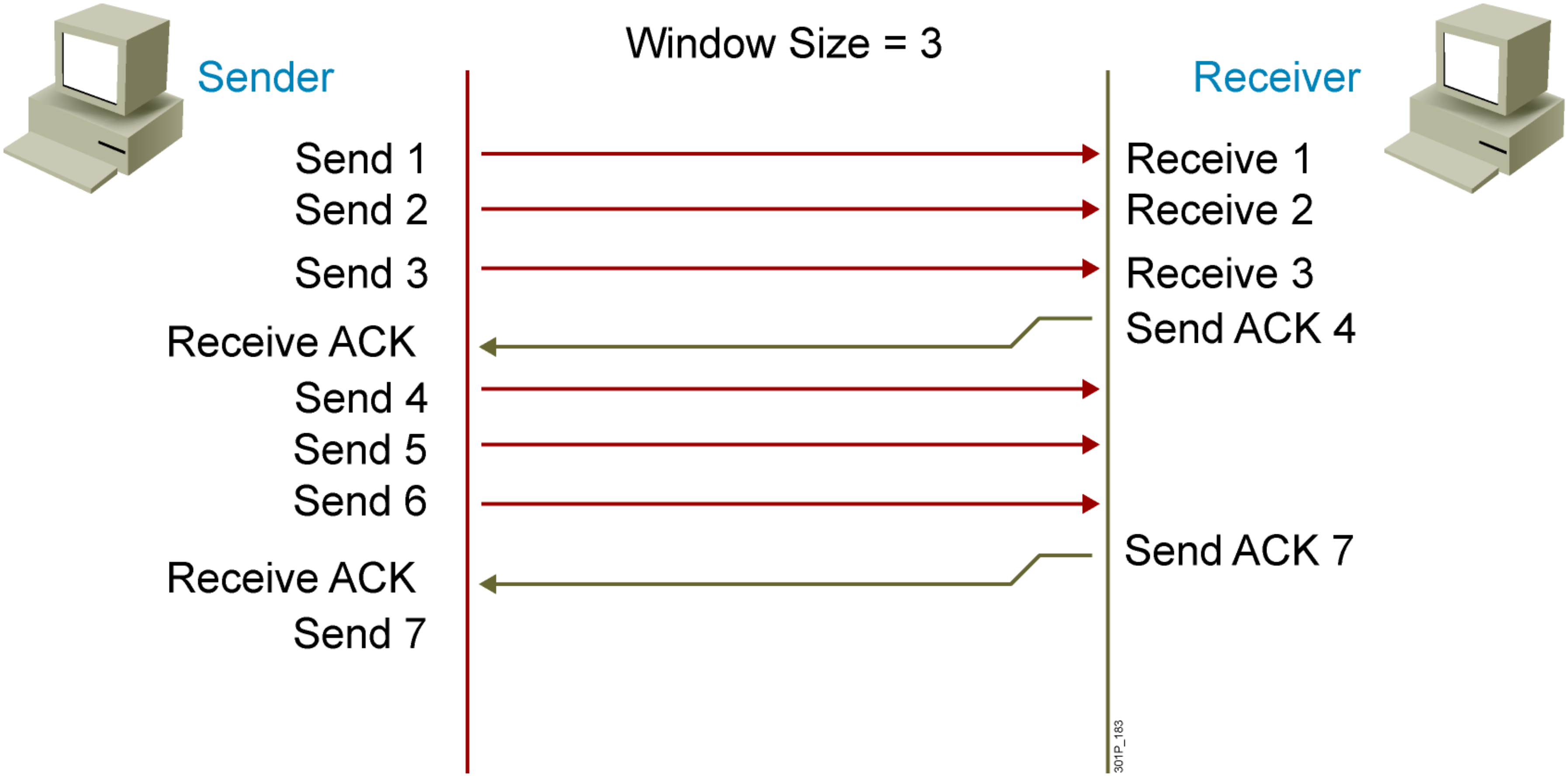
Flow Control



TCP Acknowledgment

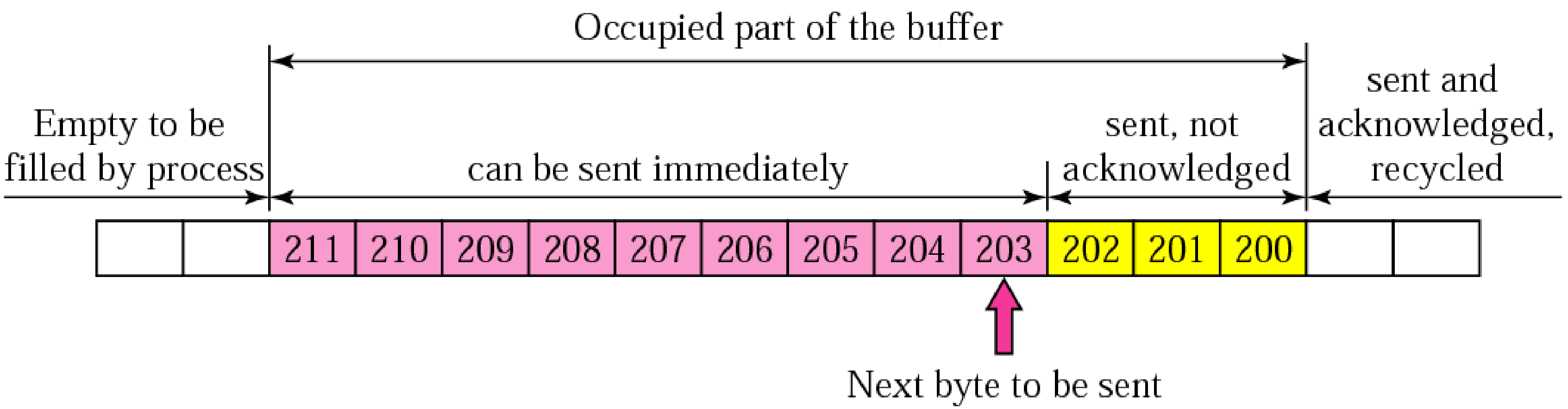


Fixed Windowing

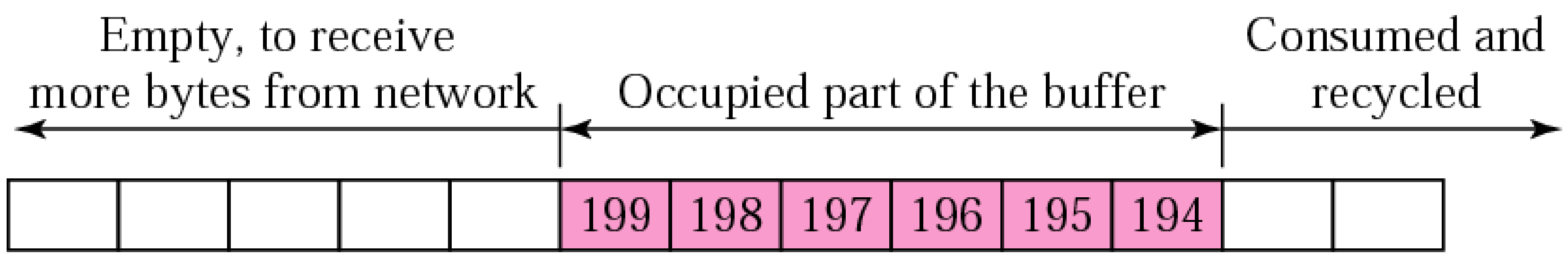


TCP Flow control: Example

Sender buffer

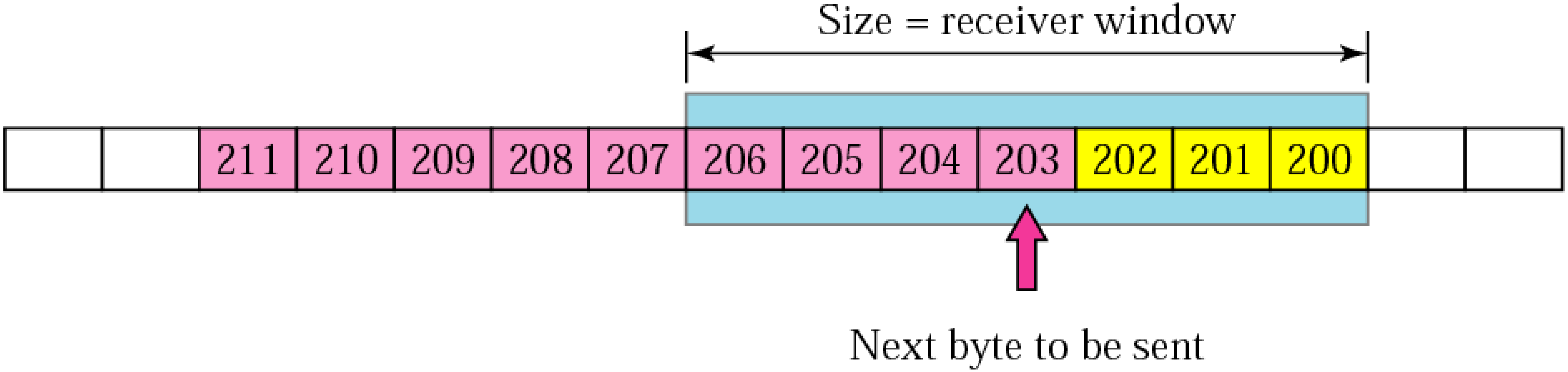


Receiver buffer



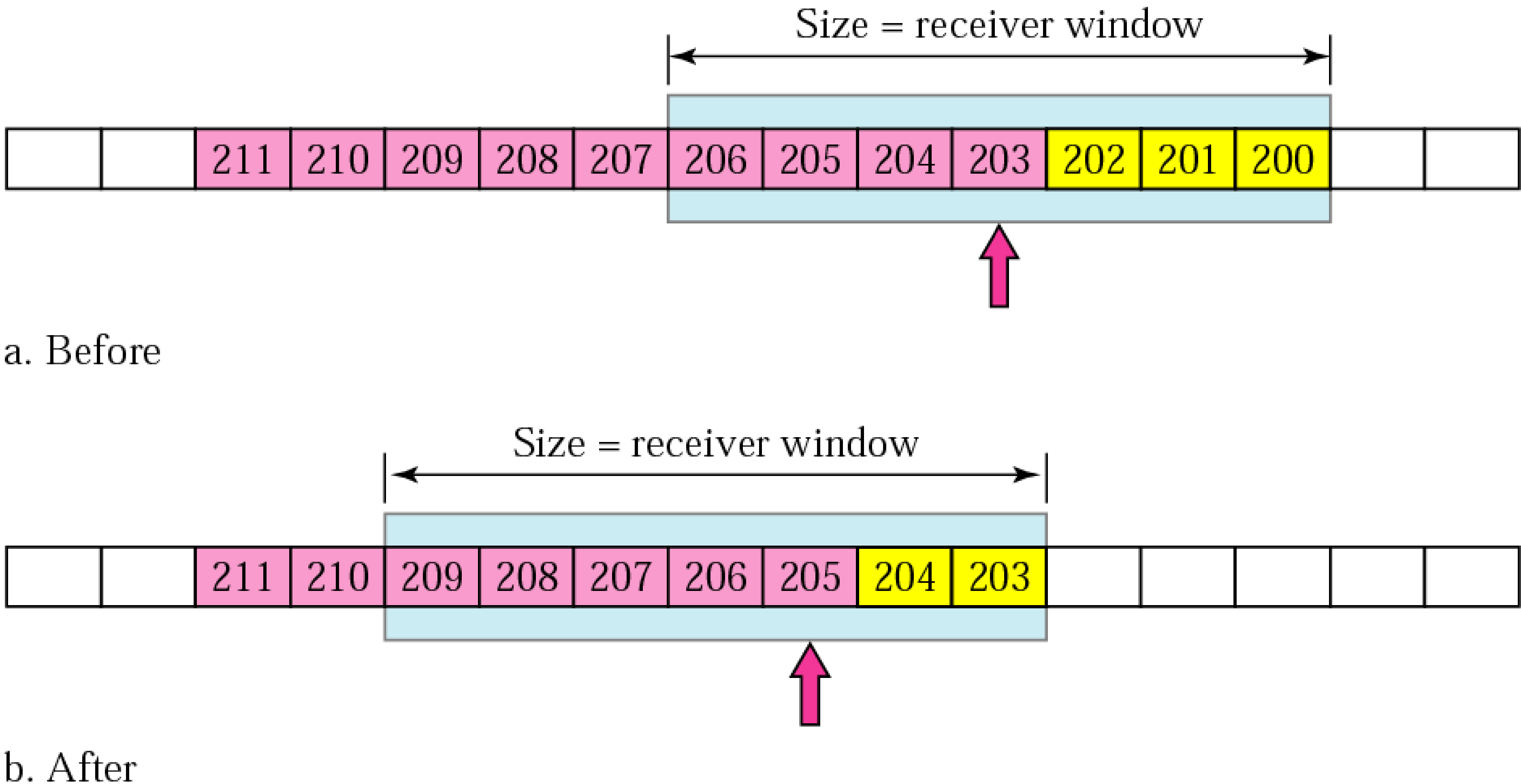
TCP Flow control: Example

Sender buffer and sender window



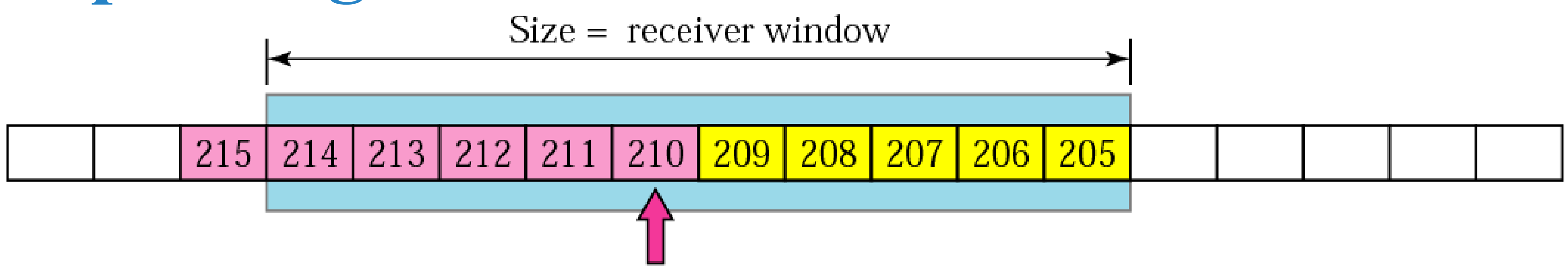
TCP Flow control: Example

Sliding the sender window

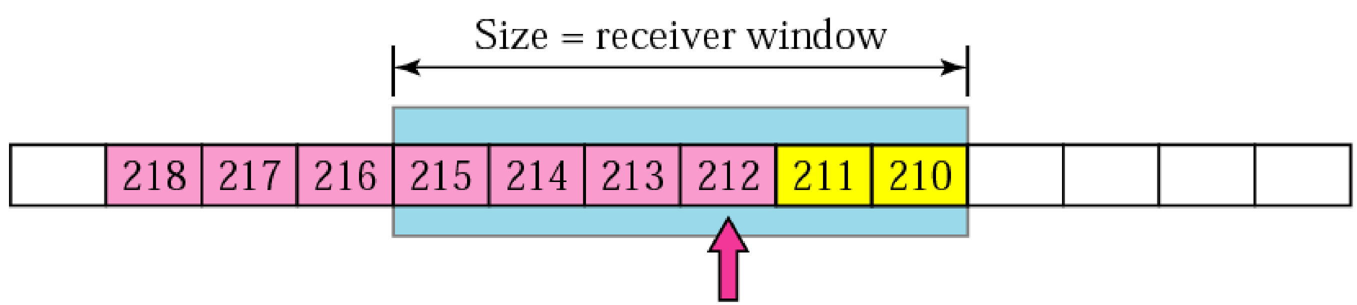


TCP Flow control: Example

Expanding the sender window



Shrinking the sender window



TCP Flow control: Example

- In TCP, the sender window size is totally controlled by the receiver window value.
- However, the actual window size can be smaller if there is congestion in the network.
- Some more points about TCP's Sliding Windows:
  - 1. The source does not have to send a full window's worth of data.
  - 2. The size of the window can be increased or decreased by the destination.
  - 3. The destination can send an acknowledgment at any time.

Keeping the Pipe Full

- D×B dictates how big the Advertised Window should be.
- Window should be opened enough to allow D×B data to be transmitted.
- Bandwidth & Time Until Wrap Around
- Wrap Around: 32-bit SequenceNum

Bandwidth	Time Until Wrap Around
T1 (1.5Mbps)	6.4 hours
Ethernet (10Mbps)	57 minutes
T3 (45Mbps)	13 minutes
FDDI (100Mbps)	6 minutes
STS-3 (155Mbps)	4 minutes
STS-12 (622Mbps)	55 seconds
STS-24 (1.2Gbps)	28 seconds

Delay-Bandwidth product

- Bytes in Transit: 16-bit AdvertisedWindow 64kB max)
- Bandwidth & Delay x Bandwidth Product for 100ms RTT

Bandwidth	Delay x Bandwidth Product
T1 (1.5Mbps)	18KB
Ethernet (10Mbps)	122KB
T3 (45Mbps)	549KB
FDDI (100Mbps)	1.2MB
STS-3 (155Mbps)	1.8MB
STS-12 (622Mbps)	7.4MB
STS-24 (1.2Gbps)	14.8MB

Nagle's Algorithm

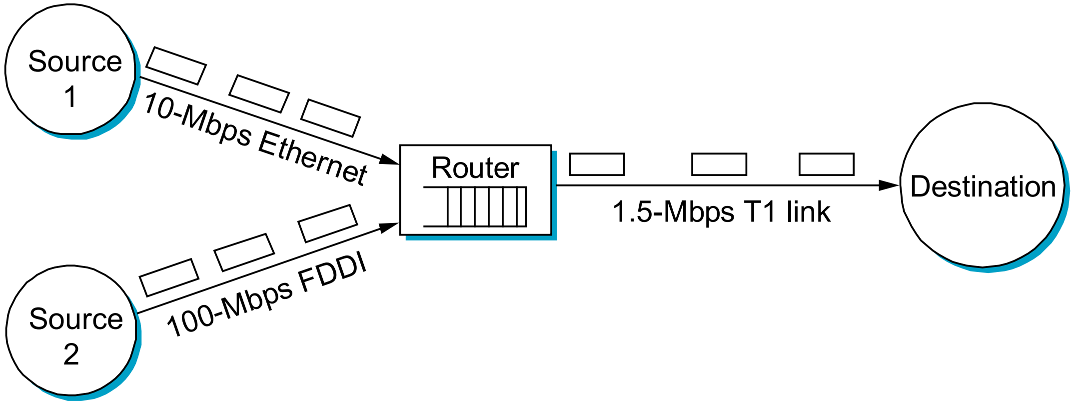
- How long does sender delay sending data?
  - too long: hurts interactive applications
  - too short: poor network utilization
  - strategies: timer-based vs self-clocking
- When application generates additional data
  - if fills a max segment (and window open): send it
  - else
    - if there is unack'ed data in transit: buffer it until ACK arrives
    - else: send it

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment startsat lower end of gap

Congestion Control Issues

- Two sides of the same coin
  - pre-allocate resources so at to avoid congestion
  - control congestion if (and when) is occurs

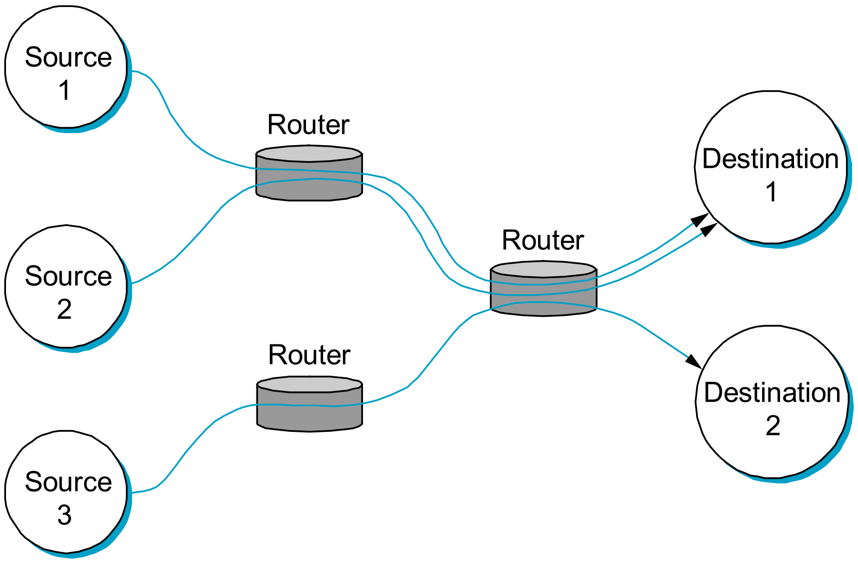


- Two points of implementation
  - hosts at the edges of the network (transport protocol)
  - routers inside the network (queuing discipline)

- Underlying service model
  - best-effort (assume for now)
  - multiple qualities of service (later)

Framework

- Connectionless flows
  - sequence of packets sent between source/destination pair
  - maintain soft state at the routers



- Taxonomy
  - router-centric versus host-centric
  - reservation-based versus feedback-based
  - window-based versus rate-based

Principles of Congestion Control

- **Congestion:**
- informally: “too many sources sending too much data too fast for network to handle”
- Formally: “Congestion occurs when number of packets transmitted approaches network capacity”
- Objective of congestion control:
  - keep number of packets below level at which performance drops off dramatically
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

Principles of Congestion Control

- Data network is a network of queues
- If arrival rate > transmission rate
  - then queue size grows without bound and packet delay goes to infinity
- Discard any incoming packet if no buffer available
- Saturated node exercises flow control over neighbors
  - May cause congestion to propagate throughout network

Ideal Performance

- Infinite buffers, no overhead for packet transmission or congestion control
- Throughput increases with offered load until full capacity
- Packet delay increases with offered load approaching infinity at full capacity
- Power = throughput / delay
- Higher throughput results in higher delay

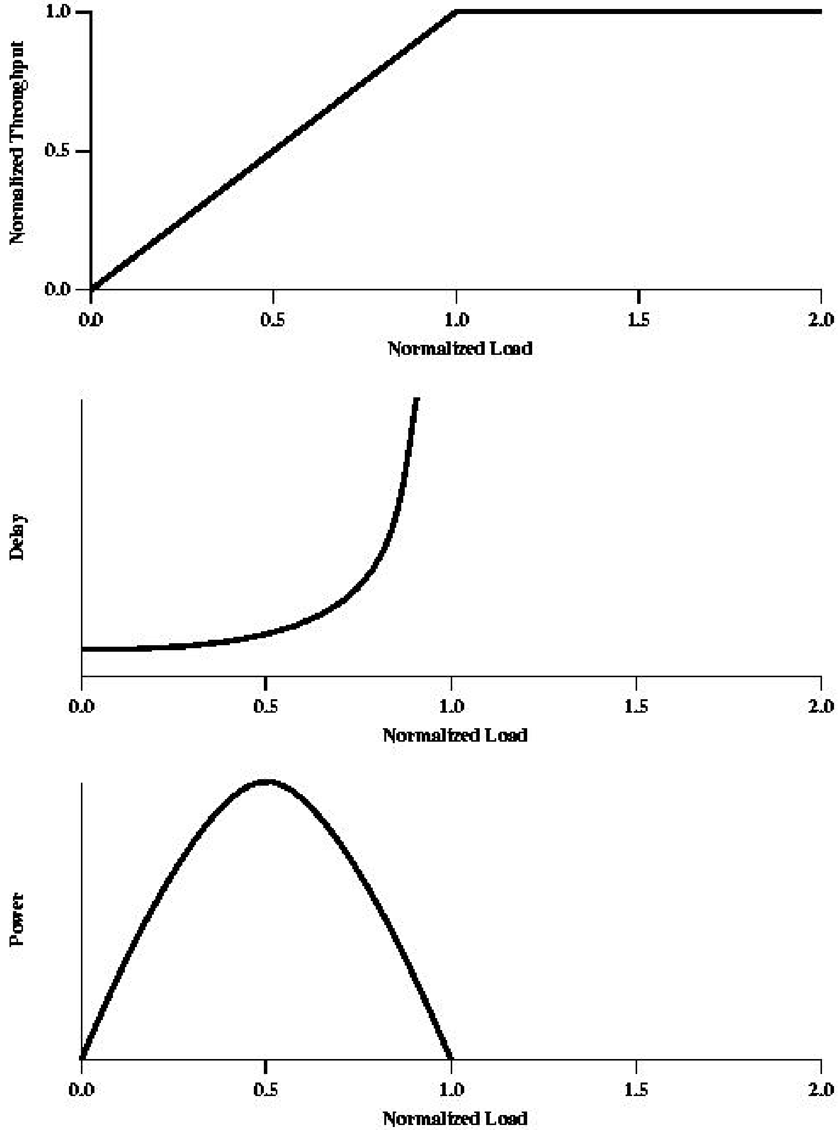


Figure 10.3 Ideal Network Utilization

Practical Performance

- Finite buffers, non-zero packet processing overhead
- With no congestion control, increased load eventually causes moderate congestion: throughput increases at slower rate than load
- Further increased load causes packet delays to increase and eventually throughput to drop to zero

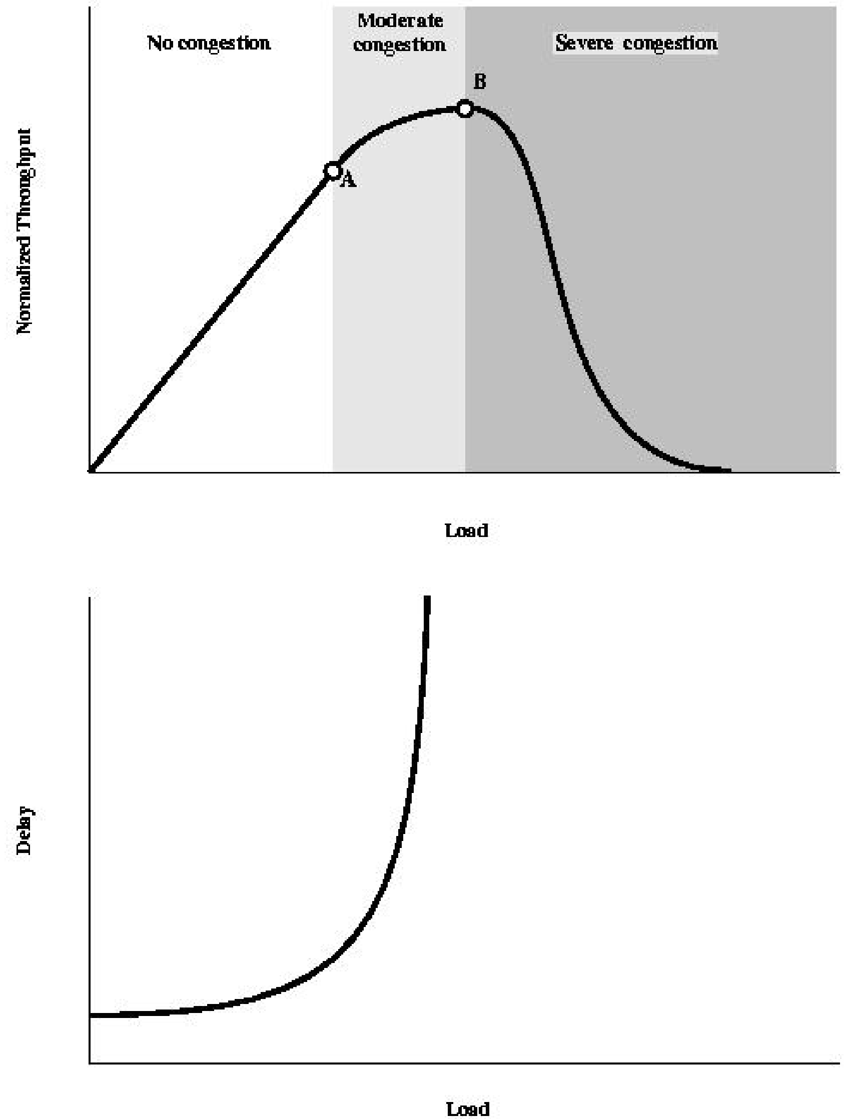
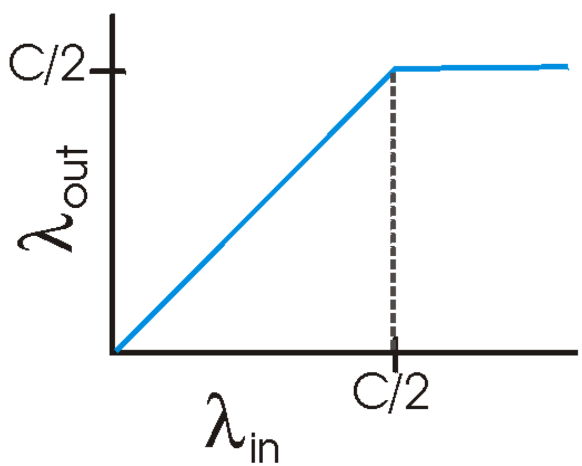
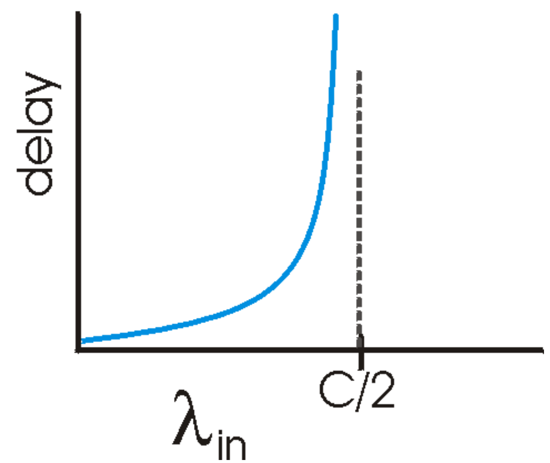


Figure 10.4 The Effects of Congestion

Causes/costs of congestion: scenario 1

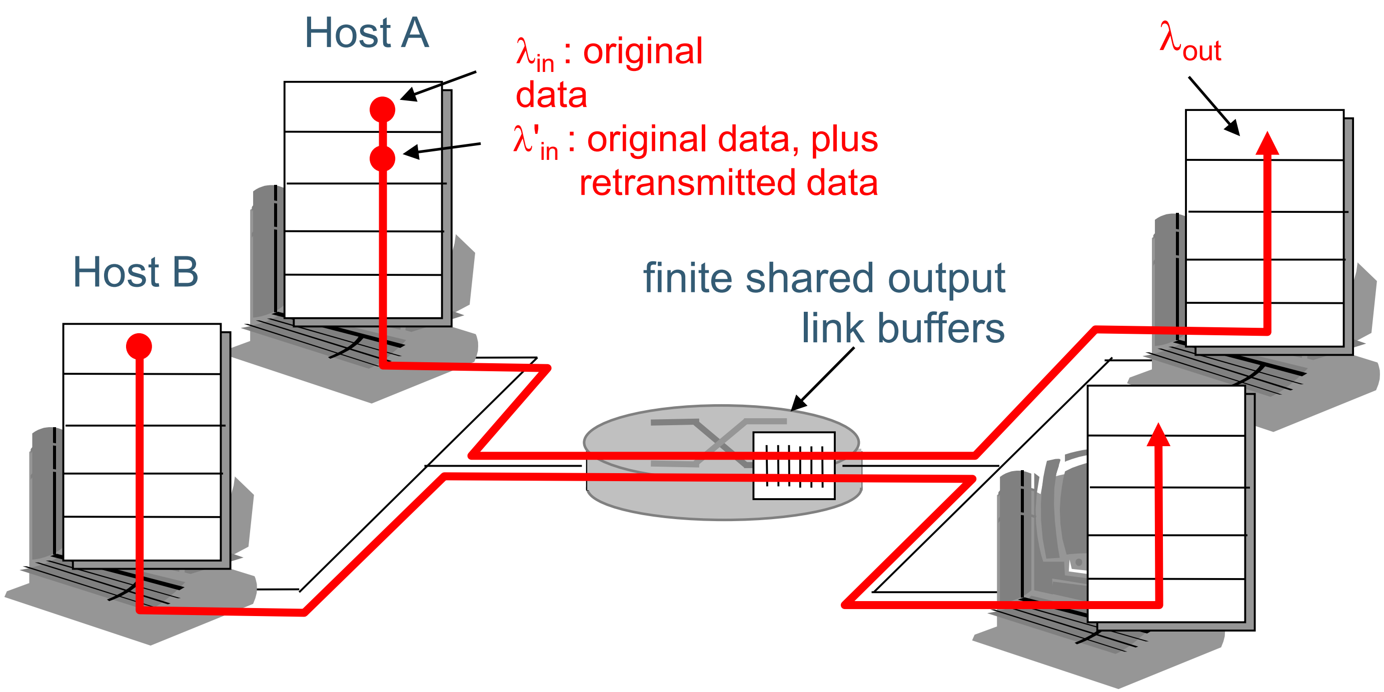
- two senders, two receivers
- one router, infinite buffers
- no retransmission
- large delays when congested
- maximum achievable throughput





Causes/costs of congestion: scenario 2

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

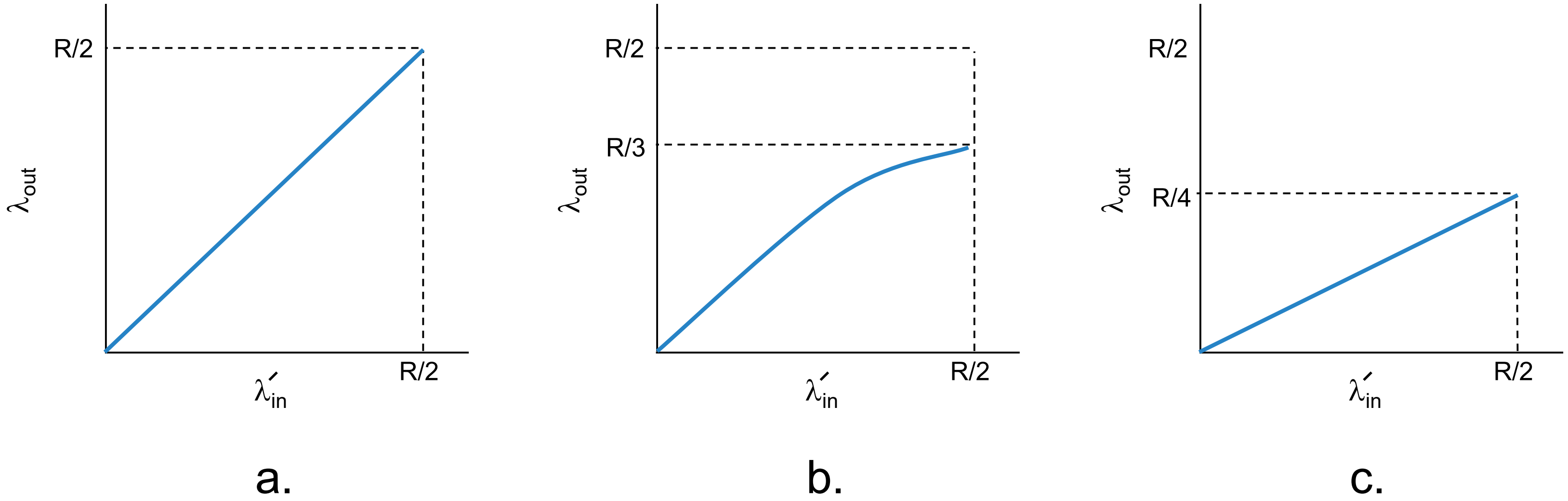


Causes/costs of congestion: scenario 2

a. always: $\lambda_{in} = \lambda_{out}$ (goodput)

b. "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$

c. retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



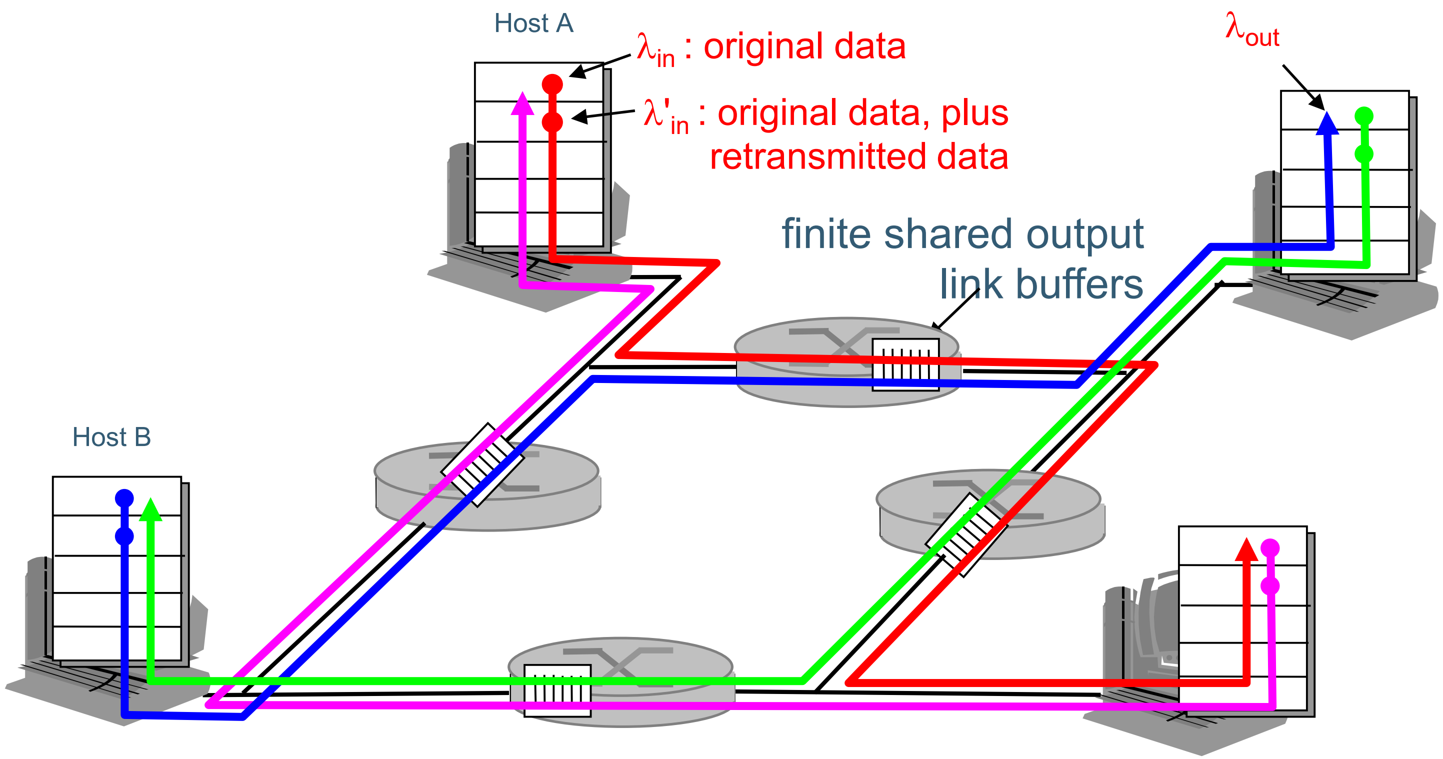
"costs" of congestion:

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

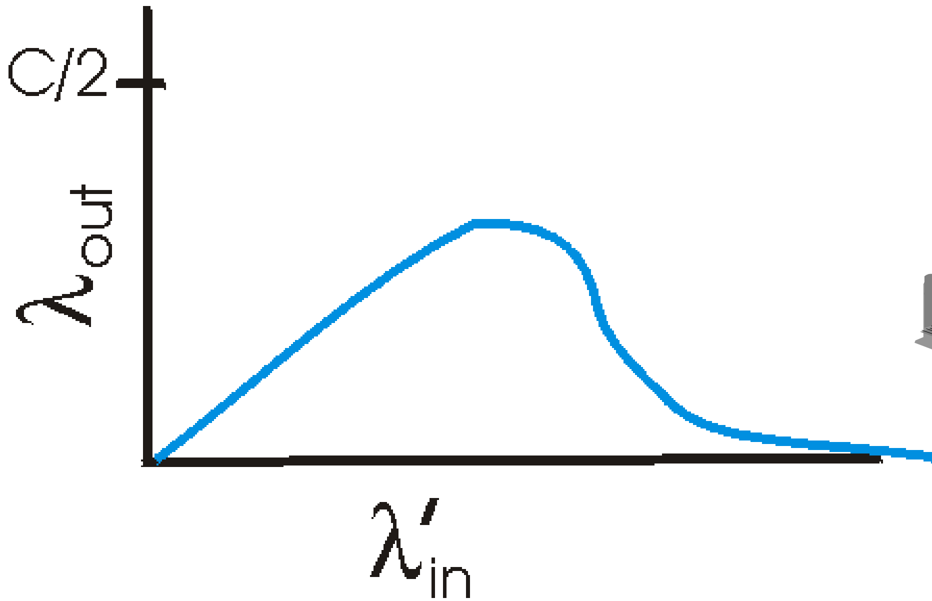
Causes/costs of congestion: scenario 3

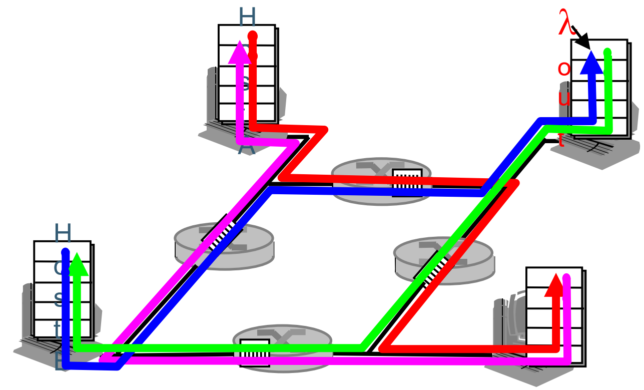
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{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

Implicit 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
  - “backpressure”

Explicit congestion signaling

- Direction
  - Backward
  - Forward
- Categories
  - Binary
  - Credit-based
  - rate-based

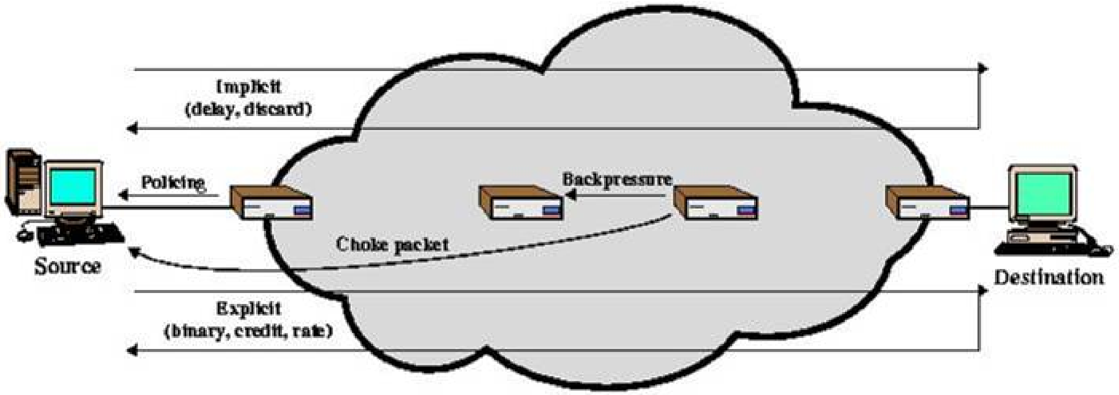


Figure 10.5 Mechanisms for Congestion Control

Congestion Avoidance with Explicit Signaling

- 2 strategies
- Congestion always occurred slowly, almost always at egress nodes
  - forward explicit congestion avoidance
- Congestion grew very quickly in internal nodes and required quick action
  - backward explicit congestion avoidance

2 Bits for Explicit Signaling

- Forward Explicit Congestion Notification
  - For traffic in same direction as received frame
  - This frame has encountered congestion
- Backward Explicit Congestion Notification
  - For traffic in opposite direction of received frame
  - Frames transmitted may encounter congestion

Congestion Control strategies

- Two strategies
  - pre-allocate resources so at to avoid congestion
  - send data and control congestion if (and when) it occurs
- Two points of implementation
  - hosts at the edges of the network (transport protocol)
  - routers inside the network (queuing discipline)

Taxonomy

- router-centric versus host-centric
  - Attempt to simplify routers
- reservation-based versus Feedback-based
  - RSVP requires API and application changes
- window-based versus rate-based
  - ATM has rate based algorithms to specify acceptable rates for each flow. Alternatives include congestion indication where hosts shrink their window.

Outline

- Transport layer Services
- TCP Overview
  - Segment structure
  - Seq nums
  - Tcp connection management
  - RTT
  - Rtd: acks, events, fast retransmit
- Flow Control
- Congestion Control
  - General causes
  - Tcp cong control (slow start, AIMD)
- TCP Throughput
- TCP versions

TCP Congestion Control

- Idea
  - assumes best-effort network (FIFO or FQ routers) each source determines network capacity for itself
  - uses implicit feedback
  - ACKs pace transmission (*self-clocking*)
- Challenge
  - determining the available capacity in the first place
  - adjusting to changes in the available capacity

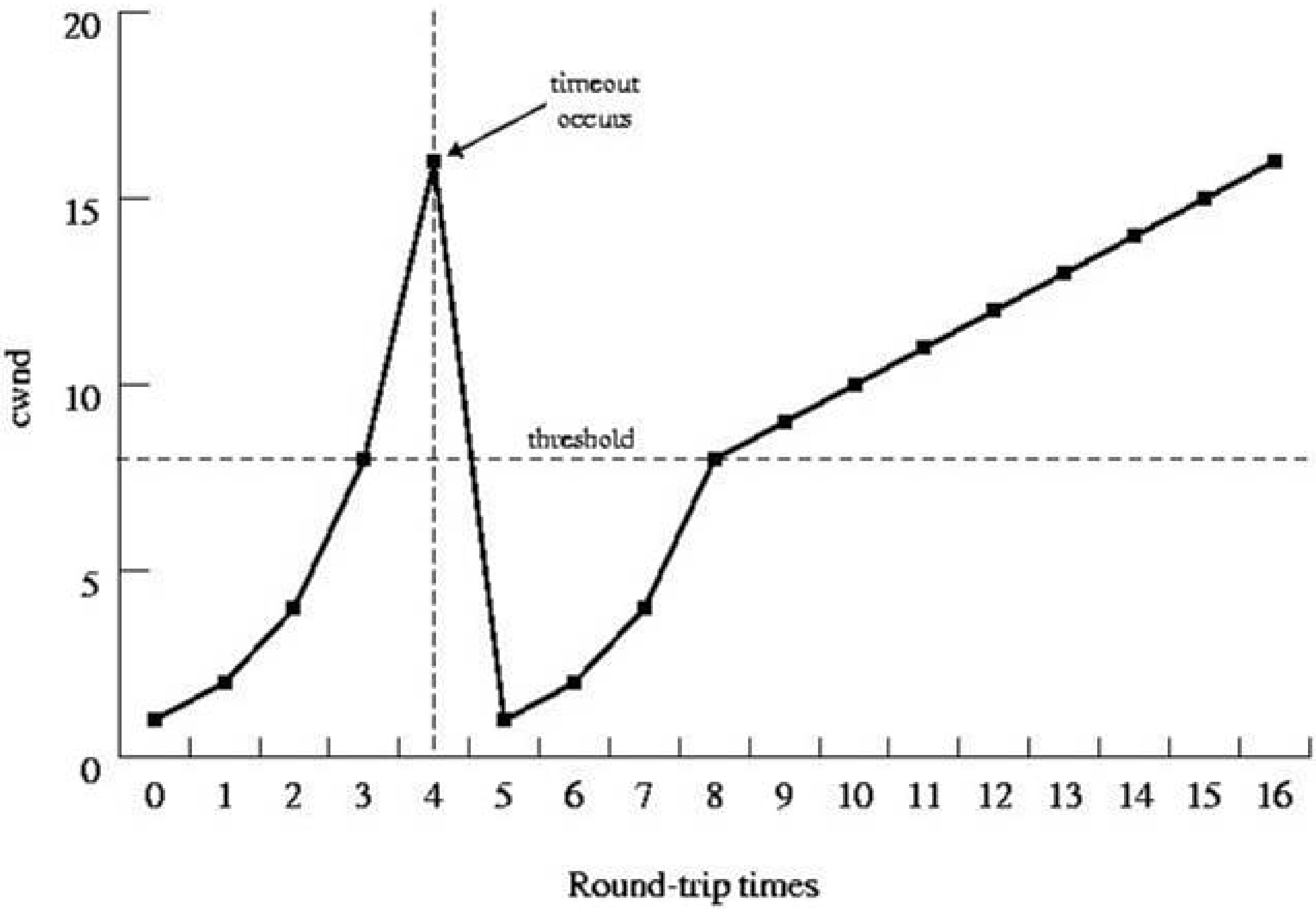


Figure 12.11 Illustration of Slow Start and Congestion Avoidance

Additive Increase/Multiplicative Decrease

- Objective: adjust to changes in the available capacity
- New state variable per connection: **`CongestionWindow`**
  - limits how much data source has in transit

```
MaxWin = MIN(CongestionWindow,
             AdvertisedWindow)
EffWin = MaxWin - (LastByteSent -
                   LastByteAcked)
```

- Idea:
  - increase **`CongestionWindow`** when congestion goes down
  - decrease **`CongestionWindow`** when congestion goes up

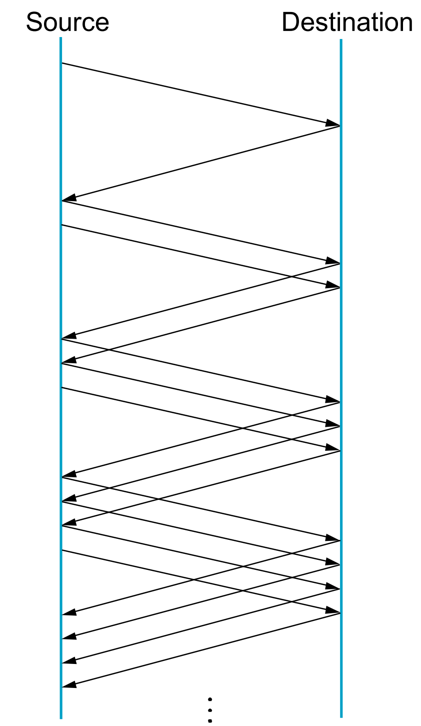
AIMD (cont)

- Question: how does the source determine whether or not the network is congested?

- Answer: a timeout occurs
  - timeout signals that a packet was lost
  - packets are seldom lost due to transmission error
  - lost packet implies congestion

AIMD (cont)

- Algorithm
  - increment **`CongestionWindow`** by one packet per RTT (*linear increase*)
  - divide **`CongestionWindow`** by two whenever a timeout occurs (*multiplicative decrease*)



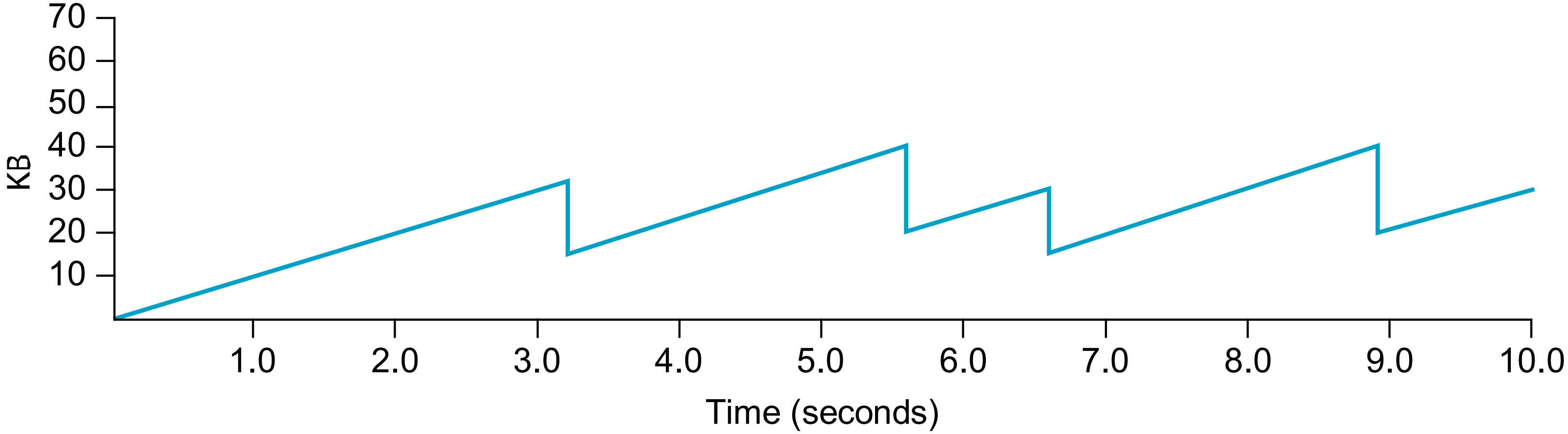
- In practice: increment a little for each ACK

```
Increment = (MSS * MSS)/CongestionWindow

CongestionWindow += Increment
```

AIMD (cont)

- Trace: sawtooth behavior

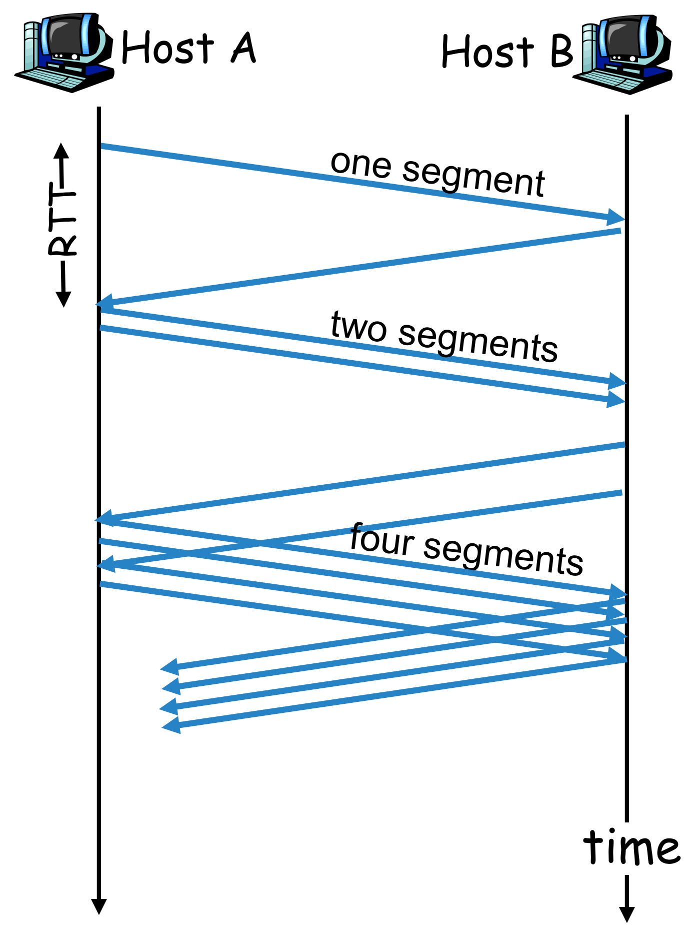


TCP Slow Start

- Objective: determine the available capacity in the first place
- When connection begins, **`CongWin`** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

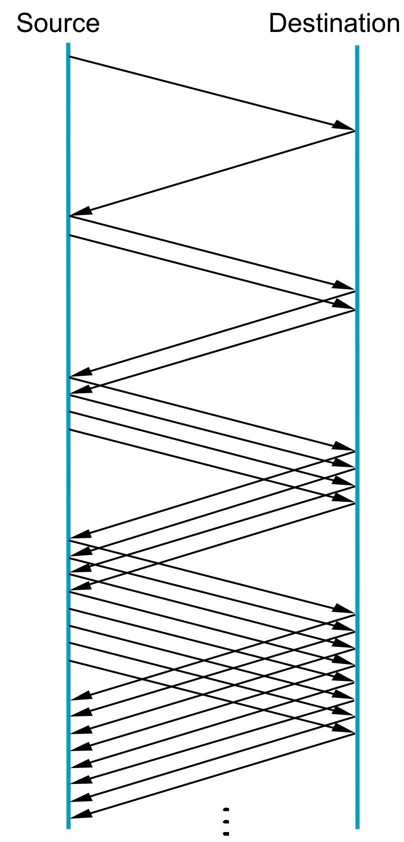
TCP Slow Start (more)

- Available Window = MIN[window, cwnd]
- Start connection with cwnd=1
- Double CongWin every RTT = =
- Increment cwnd at each ACK, to some max
- ➔ cwnd= cwnd+1



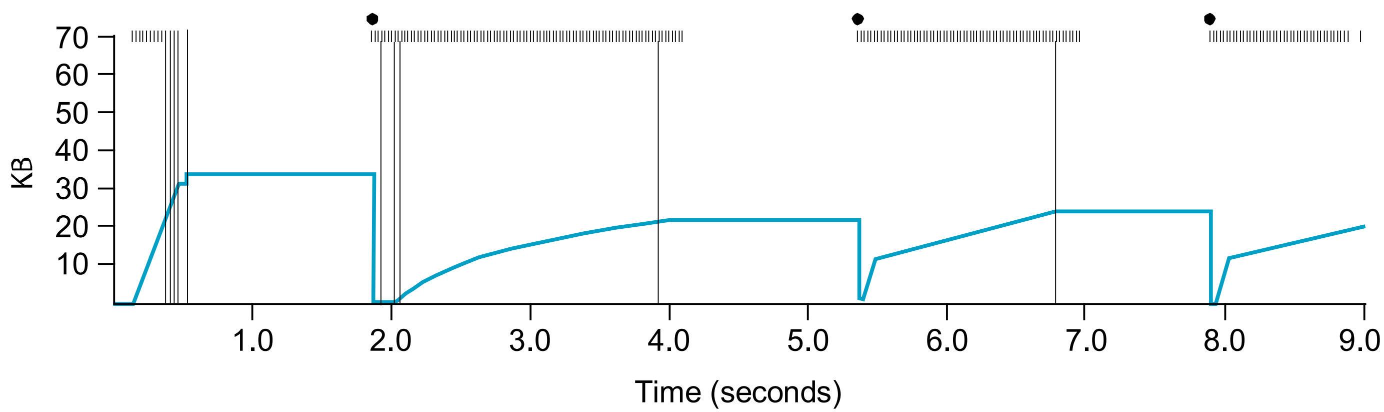
Slow Start

- Objective: determine the available capacity in the first
- Idea:
    - begin with **`CongestionWindow`** = 1 packet
    - double **`CongestionWindow`** each RTT (increment by 1 packet for each ACK)



Slow Start (cont)

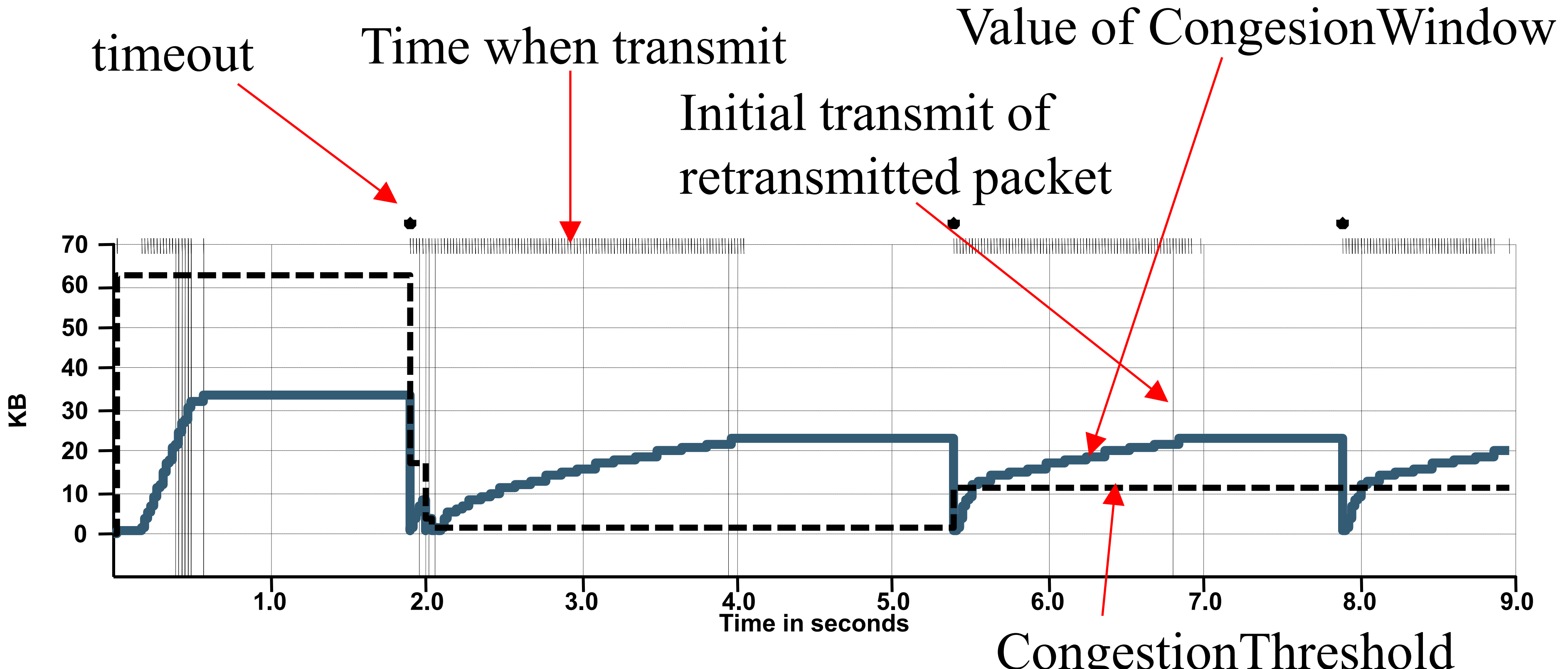
- Exponential growth, but slower than all at once
- Used…
  - when first starting connection
  - when connection goes dead waiting for timeout
- Trace



- Problem: lose up to half a **`CongestionWindow`**'s worth of data

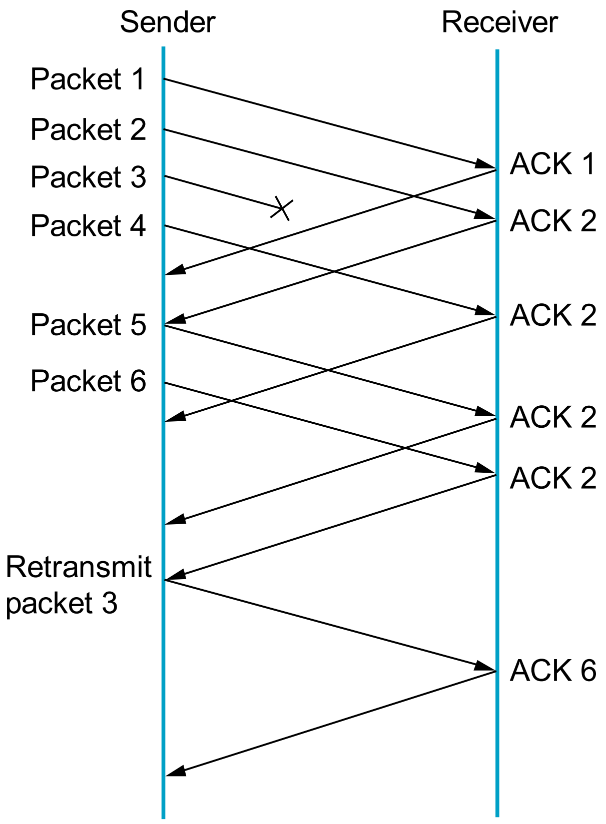
Example trace

- Loss event detected only using timeouts.
- Problem: course grain TCP timeouts lead to idle periods



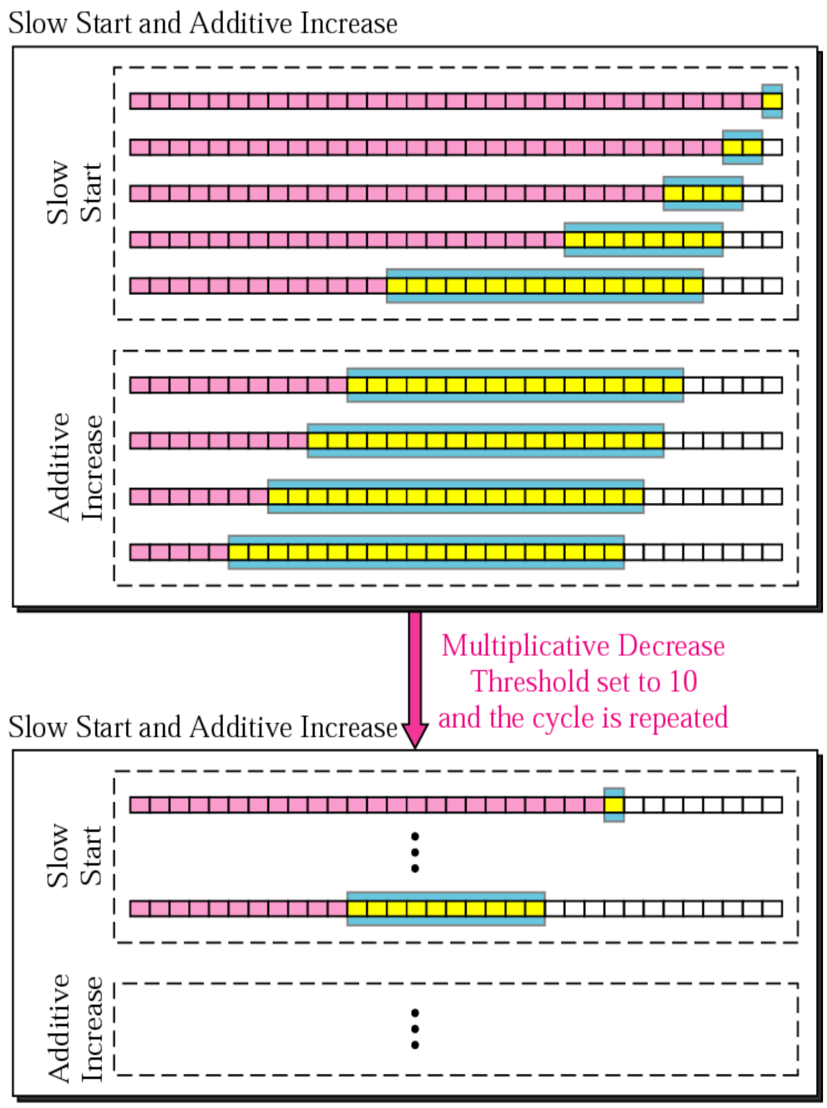
Fast Retransmit and Fast Recovery

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use duplicate ACKs to trigger retransmission



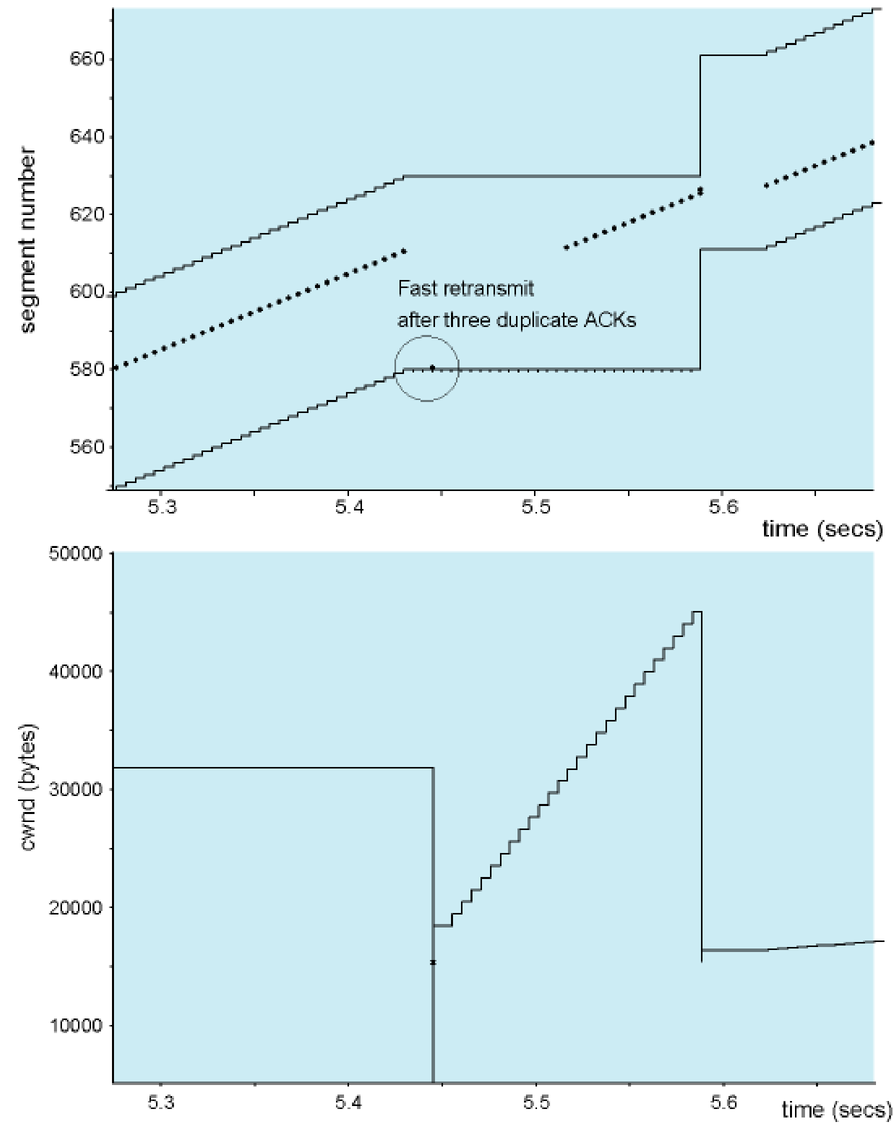
Fast Retransmit and Fast Recovery

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use duplicate ACKs to trigger retransmission

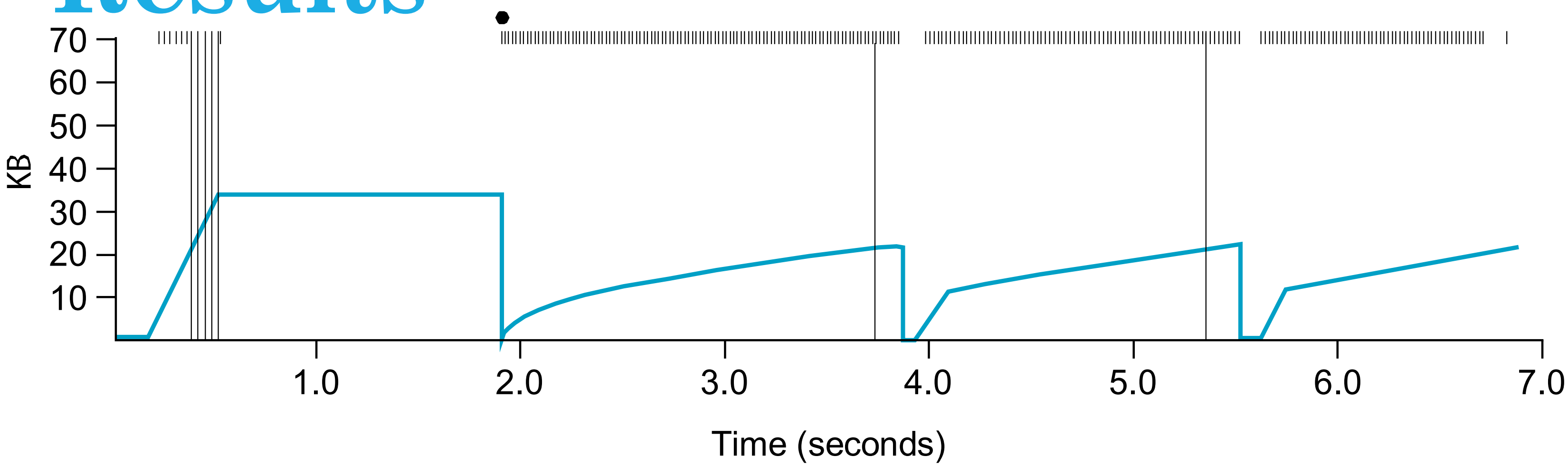


Fast Retransmit and Fast Recovery

- Problem: coarse-grain TCP timeouts lead to idle periods
- Fast retransmit: use duplicate ACKs to trigger retransmission



Results



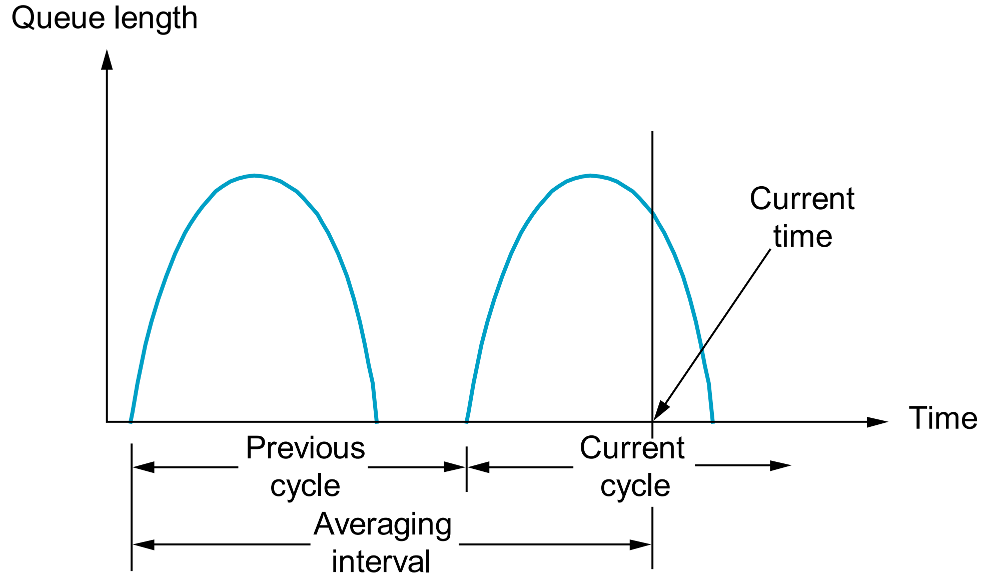
- Fast recovery
  - skip the slow start phase
  - go directly to half the last successful **`CongestionWindow (ssthresh)`**

Congestion Avoidance

- TCP's strategy
  - control congestion once it happens
  - repeatedly increase load in an effort to find the point at which congestion occurs, and then back off
- Alternative strategy
  - predict when congestion is about to happen
  - reduce rate before packets start being discarded
  - call this congestion avoidance, instead of congestion control
- Two possibilities
  - router-centric: DECbit and RED Gateways
  - host-centric: TCP Vegas

DECbit

- Add binary congestion bit to each packet header
- Router
  - monitors average queue length over last busy+idle cycle



  - set congestion bit if average queue length > 1
  - attempts to balance throughout against delay

End Hosts

- Destination echoes bit back to source
- Source records how many packets resulted in set bit
- If less than 50% of last window's worth had bit set
  - increase **`CongestionWindow`** by 1 packet
- If 50% or more of last window's worth had bit set
  - decrease **`CongestionWindow`** by 0.875 times

Random Early Detection (RED)

- Notification is implicit
  - just drop the packet (TCP will timeout)
  - could make explicit by marking the packet
- Early random drop
  - rather than wait for queue to become full, drop each arriving packet with some *drop probability* whenever the queue length exceeds some *drop level*

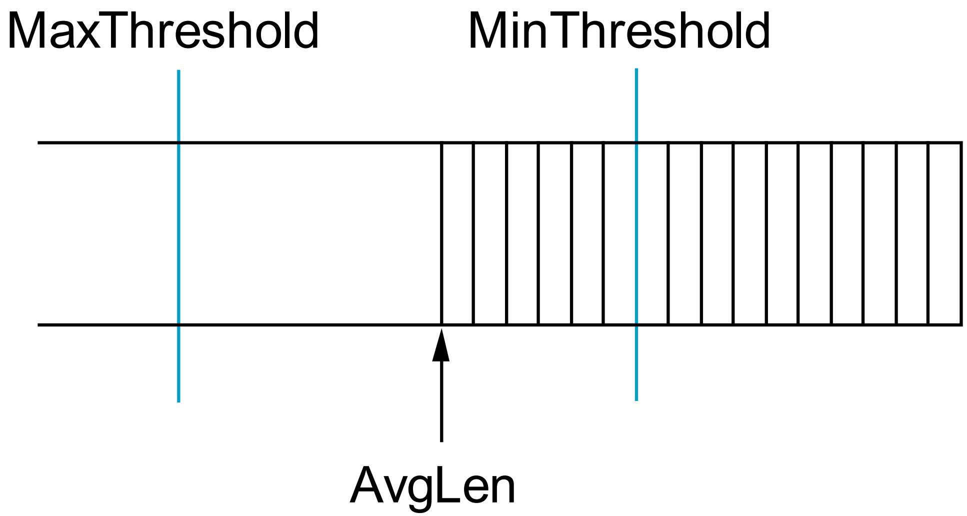
RED Details

- Compute average queue length

```
AvgLen = (1 - Weight) * AvgLen +
             Weight * SampleLen
```

0 < **`Weight`** < 1 (usually 0.002)

`SampleLen` is queue length each time a packet arrives



RED Details (cont)

- Two queue length thresholds

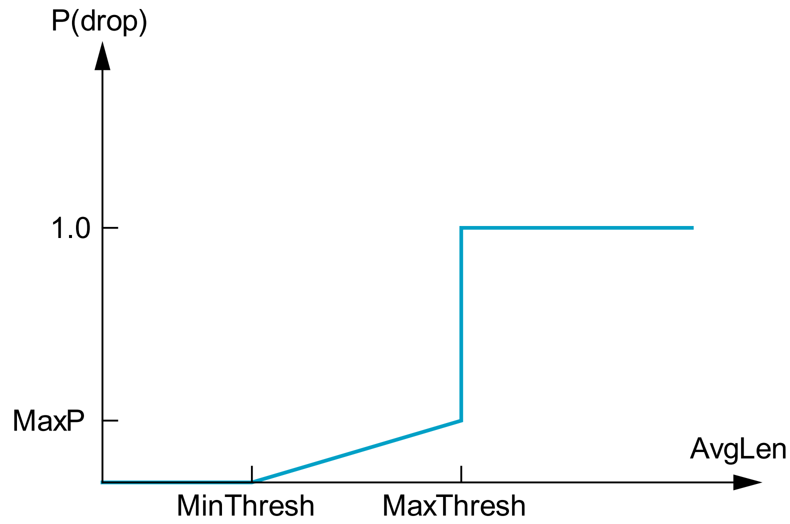
```
if AvgLen <= MinThreshold then
     enqueue the packet
if MinThreshold < AvgLen < MaxThreshold then
     calculate probability P
     drop arriving packet with probability P
if ManThreshold <= AvgLen then
     drop arriving packet
```

RED Details (cont)

- Computing probability P

```
TempP = MaxP * (AvgLen - MinThreshold)/
  (MaxThreshold - MinThreshold)
P = TempP/(1 - count * TempP)
```

- Drop Probability Curve



Tuning RED

- Probability of dropping a particular flow's packet(s) is roughly proportional to the share of the bandwidth that flow is currently getting
- MaxP is typically set to 0.02, meaning that when the average queue size is halfway between the two thresholds, the gateway drops roughly one out of 50 packets.
- If traffic id bursty, then MinThreshold should be sufficiently large to allow link utilization to be maintained at an acceptably high level
- Difference between two thresholds should be larger than the typical increase in the calculated average queue length in one RTT; setting MaxThreshold to twice MinThreshold is reasonable for traffic on today's Internet
- Penalty Box for Offenders

Summary: TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in slow-start phase, window grows exponentially.

- When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.

- When a triple duplicate ACK occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.

- When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS.

TCP sender congestion control

Event	State	TCP Sender Action	Commentary
ACK receipt for previously unacked data	Slow Start (SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
ACK receipt for previously unacked data	Congestion Avoidance (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
Timeout	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP throughput

- What's the average throughout ot TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT
- Average throughput as a function of drop probability:

$$B(p) = \sqrt{\frac{3}{2p}}$$

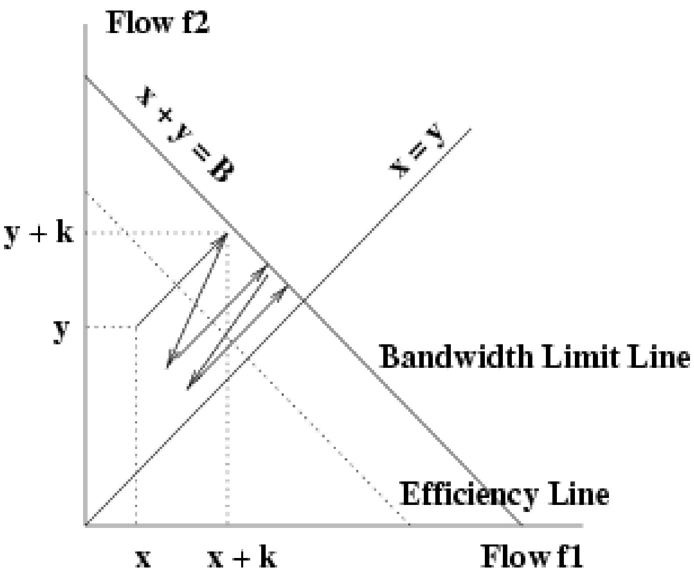
TCP Throughput

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ $L = 2 \cdot 10^{-10}$ *Wow*
- New versions of TCP for high-speed needed!

TCP Fairness



Incr: $w \leftarrow w + a$, $a = 1$
Decr: $w \leftarrow bw$, $b = 1/2$

f1(k+1)=f1(k)+a if f1(k)+f2(k) < B
f1(k+1)=bf1(k) if f1(k)+f2(k) >= B

f2(k+1)=f2(k)+a if f2(k)+f2(k) < B
f2(k+1)=bf2(k) if f1(k)+f2(k) >= B

f2(k+1)-f1(k+1)= f2(k)-f1(k) if f1(k)+f2(k) < B
f2(k+1)-f1(k+1)= b(f2(k)-f1(k)) if f1(k)+f2(k) >= B

TCP Flavors

- TCP-Tahoe
  - W=1 adaptation on congestion
- TCP-Reno
  - W=W/2 adaptation on fast retransmit, W=1 on timeout
- TCP-newReno
  - TCP-Reno + fast recovery
- TCP Vegas
  - Uses round-trip time as an early-congestion-feedback mechanism
  - Reduces losses
- TCP-SACK
  - Selective Acknowledgements

TCP Tahoe

- Slow-start
- Congestion control upon time-out.
- Congestion window reduced to 1 and slow-start performed again
- Simple
- Congestion control too aggressive
- It takes a complete timeout interval to detect a packet loss and this empties the pipeline

TCP Reno

- Tahoe + Fast re-transmit
- Packet loss detected both through timeouts, and through DUP-ACKs
- On receiving 3 DUP-ACKs retransmit packet and reduce the ssthresh to half of current window and set cwnd to this value. For each DUP-ACK received increase cwnd by one. If cwnd larger than number of packets in transit send new data else wait. In this way the pipe is not emptied.
- Window cut-down to 1 (and subsequent slow-start) performed only on time-out

TCP New-Reno

- TCP-Reno with more intelligence during fast recovery
- In TCP-Reno, the first partial ACK will bring the sender out of the fast recovery phase
- Results in multiple reductions of the cwnd for packets lost in one RTT.
- In TCP New-Reno, partial ACK is taken as an indication of another lost packet (which is immediately retransmitted).
- Sender comes out of fast recovery only after all outstanding packets (at the time of first loss) are ACKed.

TCP SACK

- TCP (Tahoe, Reno, and New-Reno) uses cumulative acknowledgements
- When there are multiple losses, TCP Reno and New-Reno can retransmit only one lost packet per round-trip time
- SACK enables receiver to give more information to sender about received packets allowing sender to recover from multiple-packet losses faster

TCP SACK (Example)

- Assume packets 5-25 are transmitted
- Let packets 5, 12, and 18 be lost
- Receiver sends back a CACK=5, and SACK=(6-11,13-17,19-25)
- Sender knows that packets 5, 12, and 18 are lost and retransmits them immediately

TCP Vegas

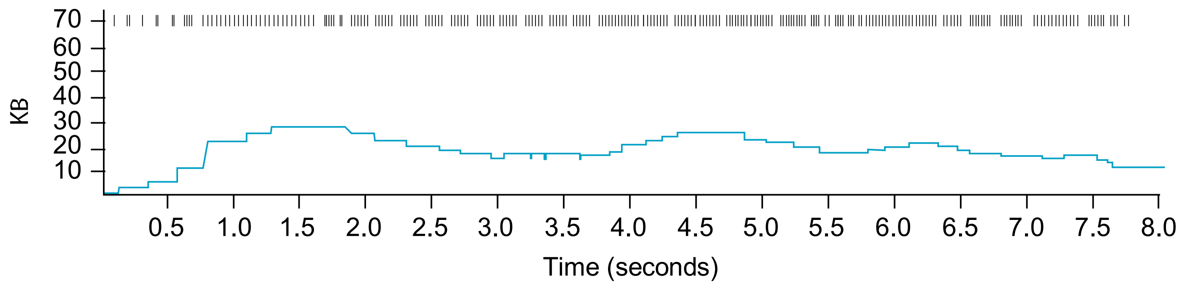
- Idea: source watches for some sign that some router's queue is building up and congestion will happen soon; e.g.,
  - RTT is growing
  - sending rate flattens

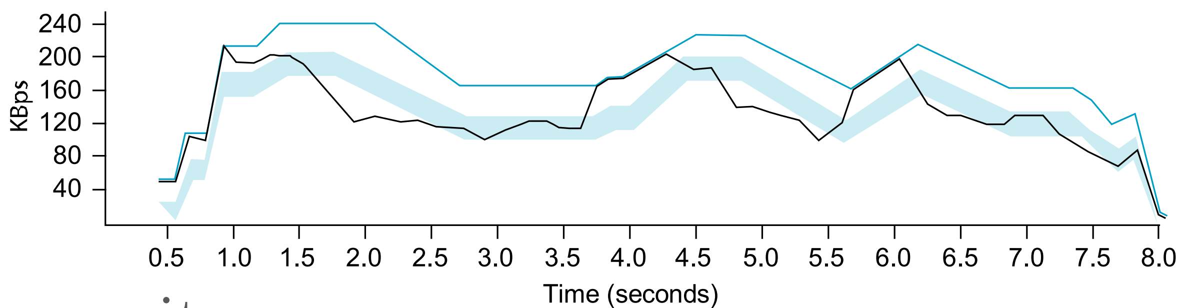
Algorithm

- Let BaseRTT be the minimum of all measured RTTs (commonly the RTT of the first packet)
- if not overflowing the connection, then
  - ExpectedRate = CongestionWindow / BaseRTT
- source calculates current sending rate (ActualRate) once per RTT
- source compares ActualRate with ExpectedRate
  - Diff = ExpectedRate – ActualRate
  - if Diff < α
    - -->increase CongestionWindow linearly
  - else if Diff > β
    - -->decrease CongestionWindow linearly
  - else
    - -->leave CongestionWindow unchanged

Algorithm (cont)

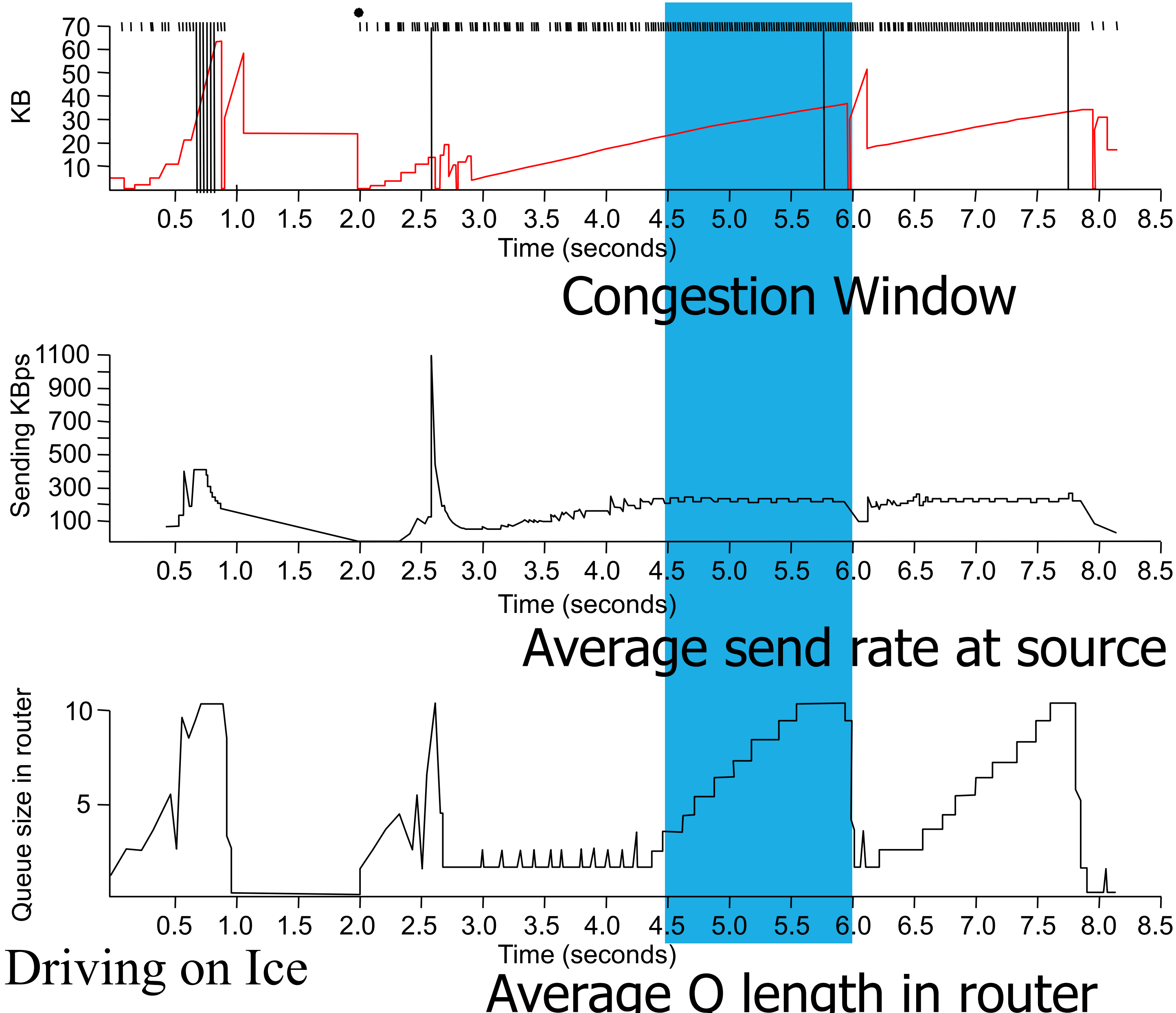
- Parameters
  - $\alpha = 1$ packet
  - $\beta = 3$ packets





- Even faster retransmit
  - keep fine-grained timestamps for each packet
  - check for timeout on first duplicate ACK

Intuition

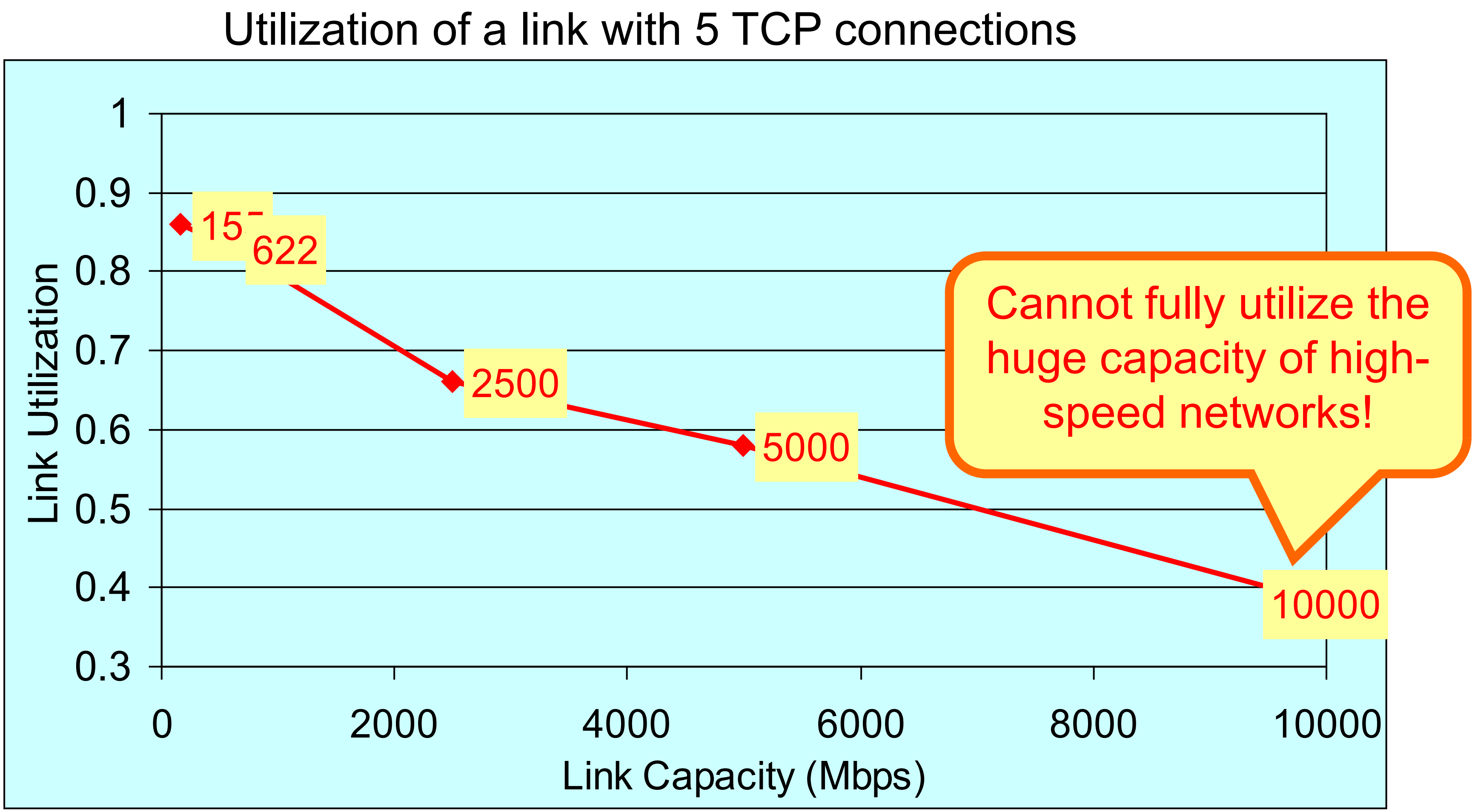


Driving on Ice

Vegas Details

- Value of throughput with no congestion is compared to current throughput
- If current difference is smaller, increase window size linearly
- If current difference is larger, decrease window size linearly
- The change in the Slow Start Mechanism consists of doubling the window every other RTT, rather than every RTT and of using a boundary in the difference between throughputs to exit the Slow Start phase, rather than a window size value.

TCP Performance

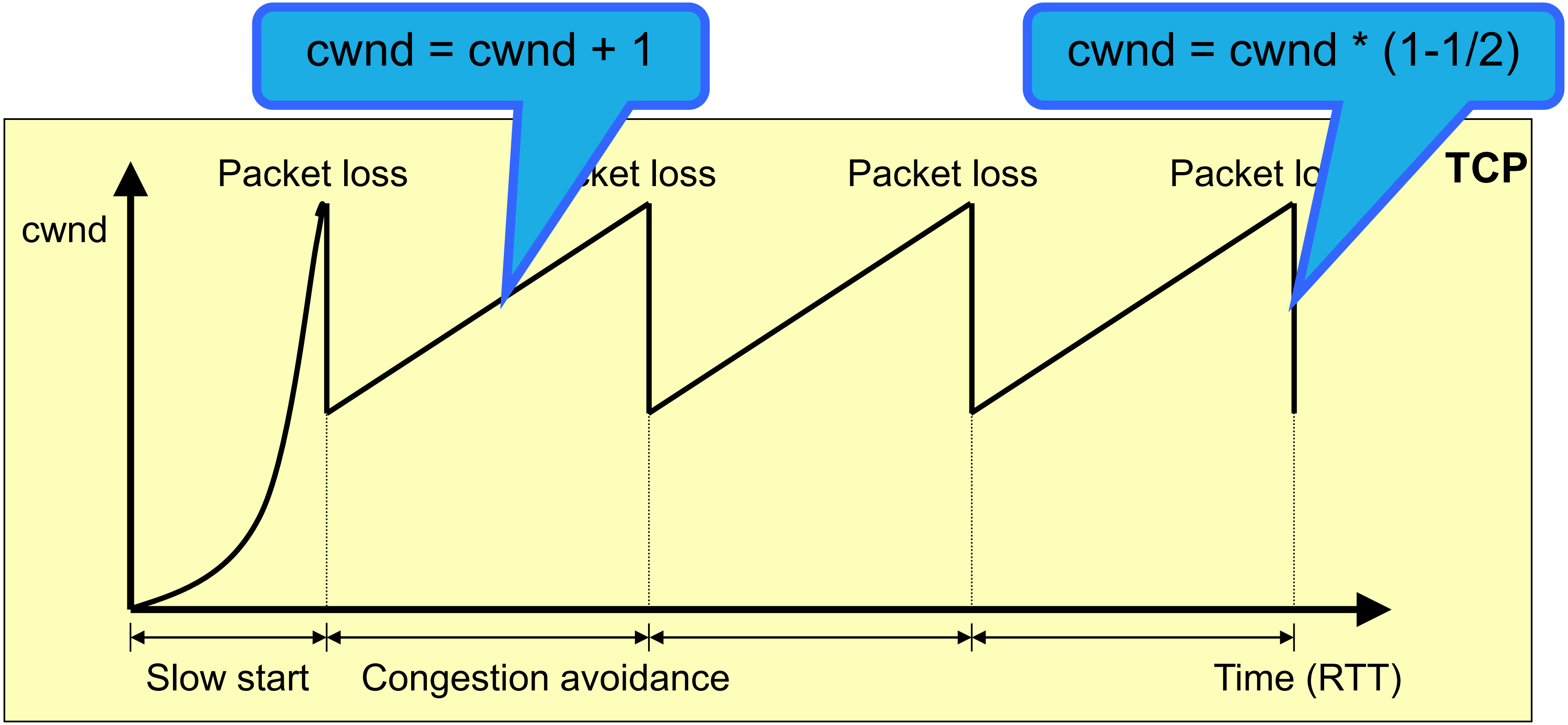


NS-2 Simulation (100 sec)

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

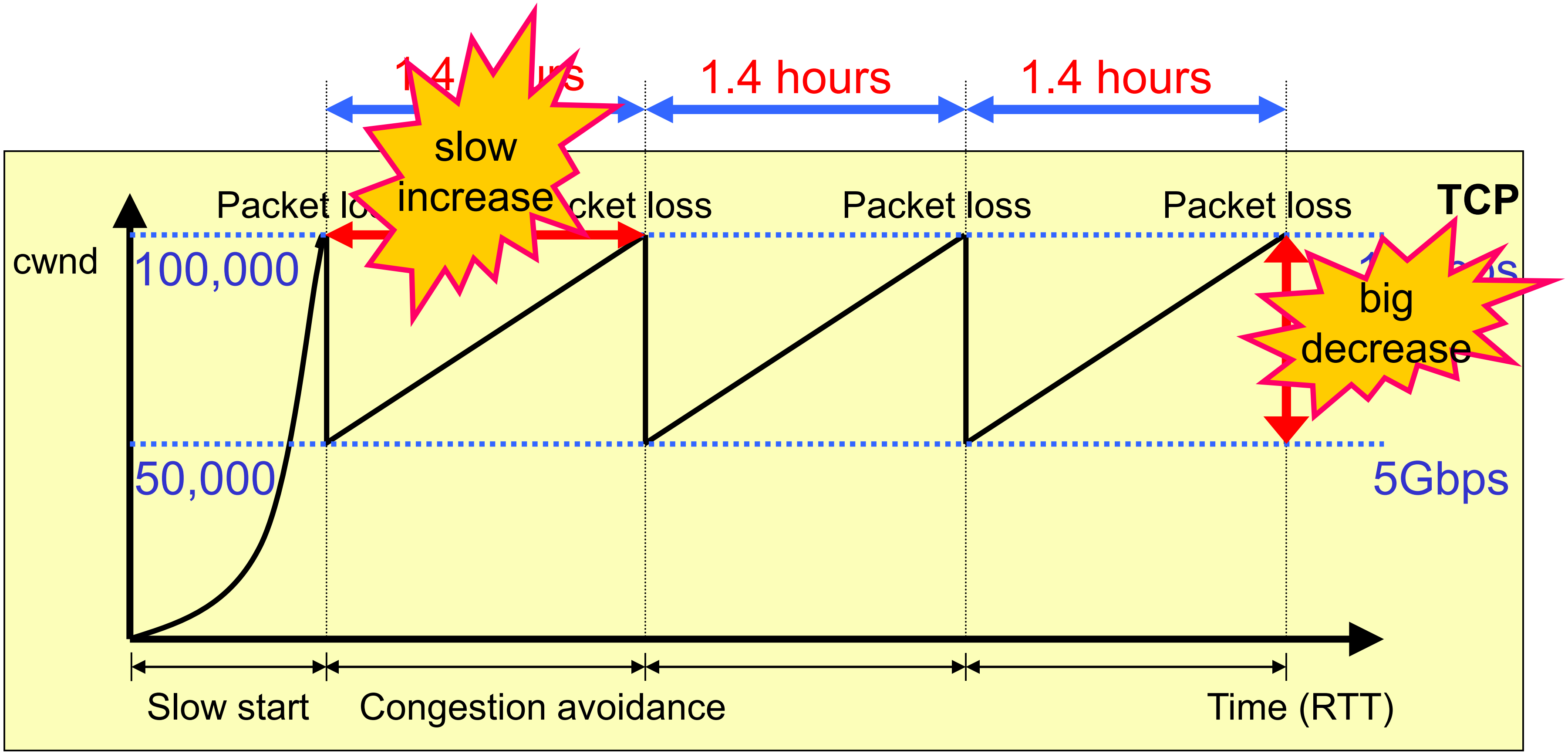
TCP Congestion Control

- The instantaneous throughput of TCP is controlled by a variable *cwnd*,
- TCP transmits approximately a *cwnd* number of packets per RTT (Round-Trip Time).



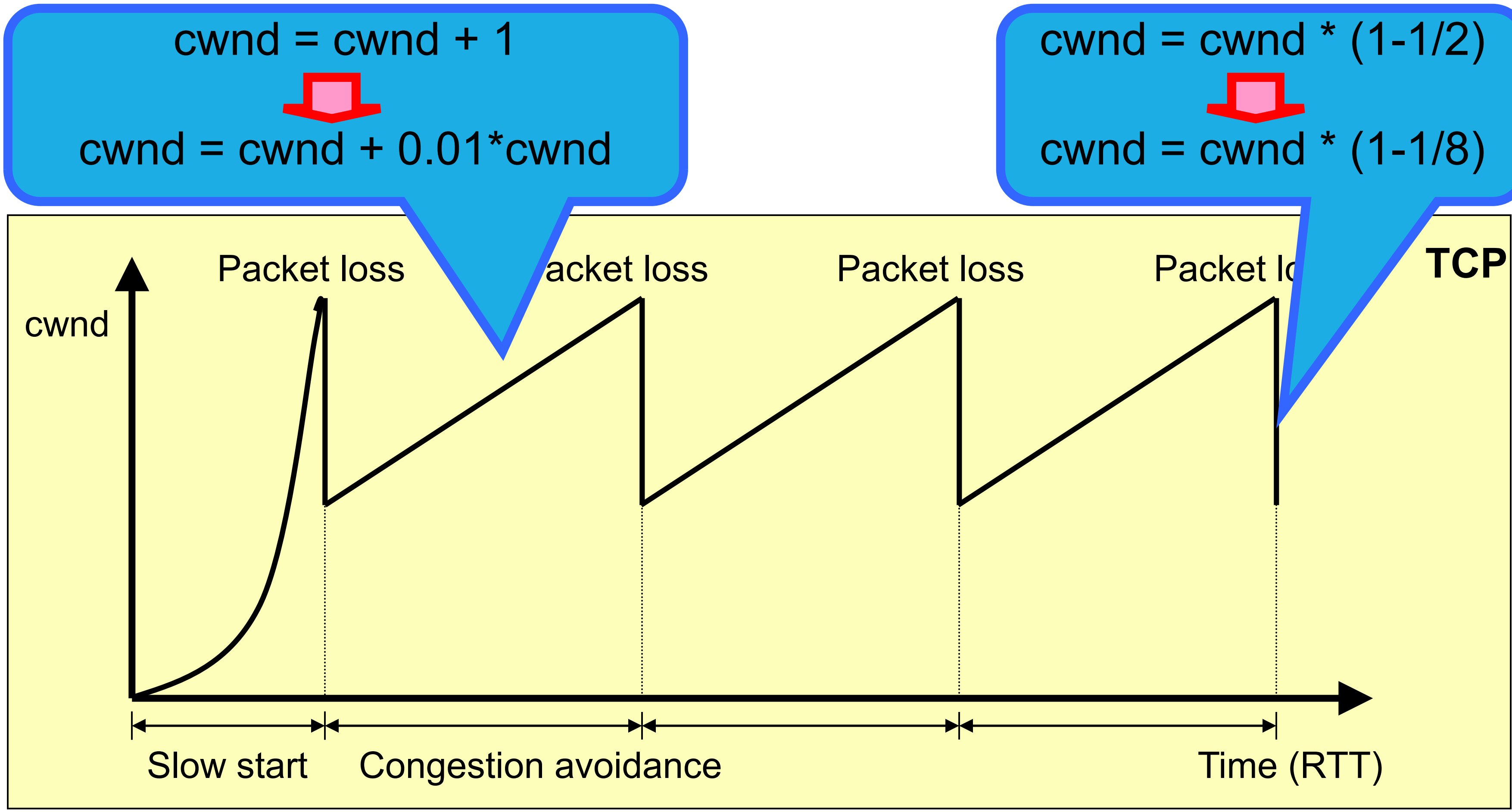
TCP over High-Speed Networks

- A TCP connection with 1250-Byte packet size and 100ms RTT is running over a 10Gbps link (assuming no other connections, and no buffers at routers)



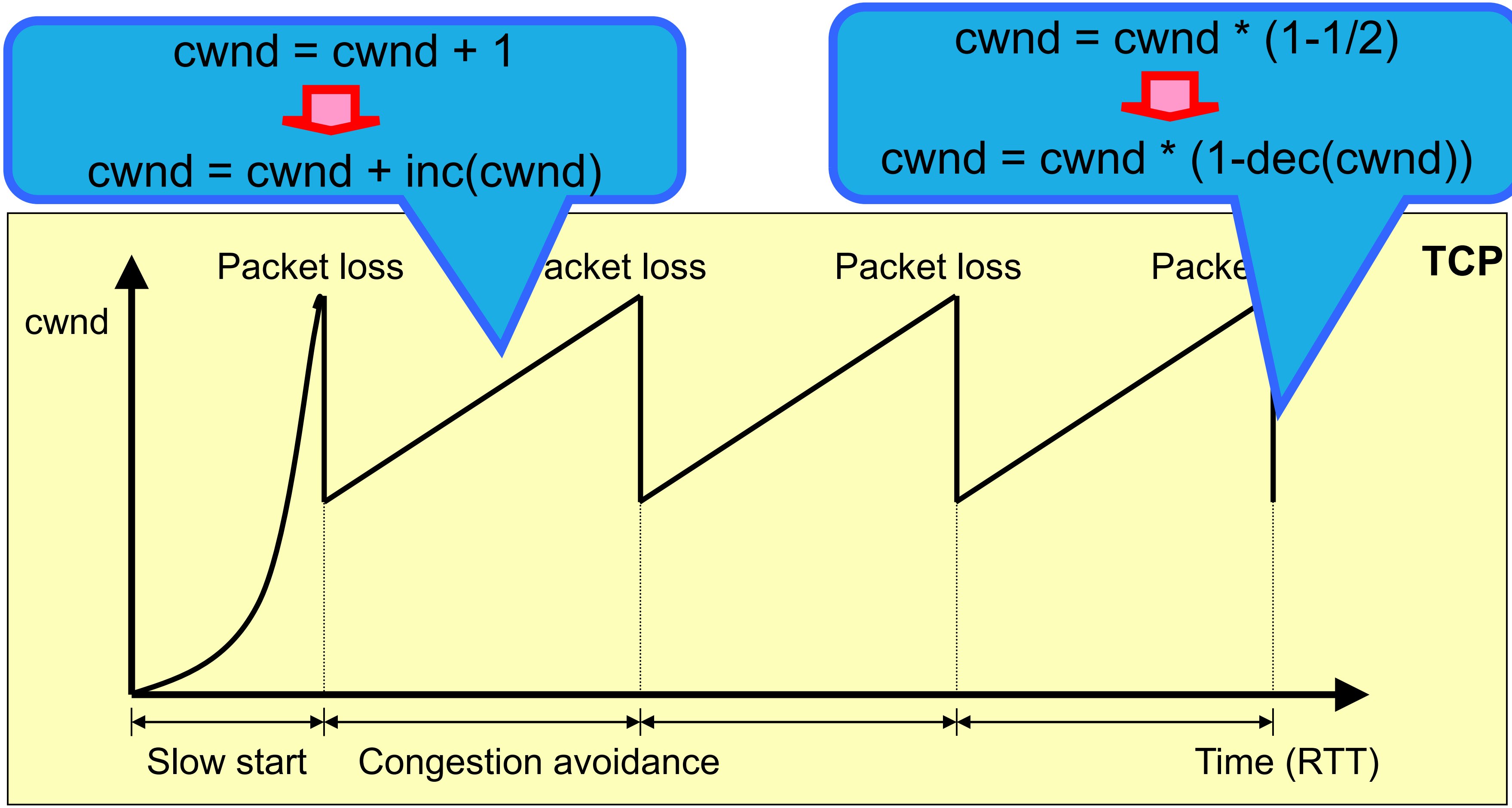
STCP (Scalable TCP)

- STCP adaptively increases *cwnd*, and decreases *cwnd* by 1/8.



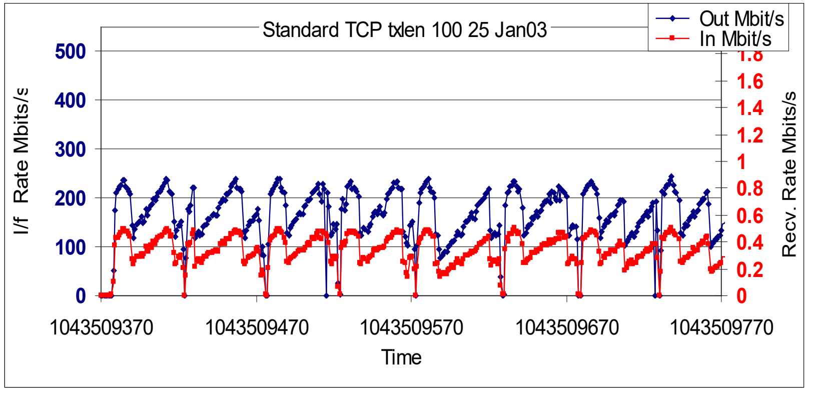
HSTCP (High Speed TCP)

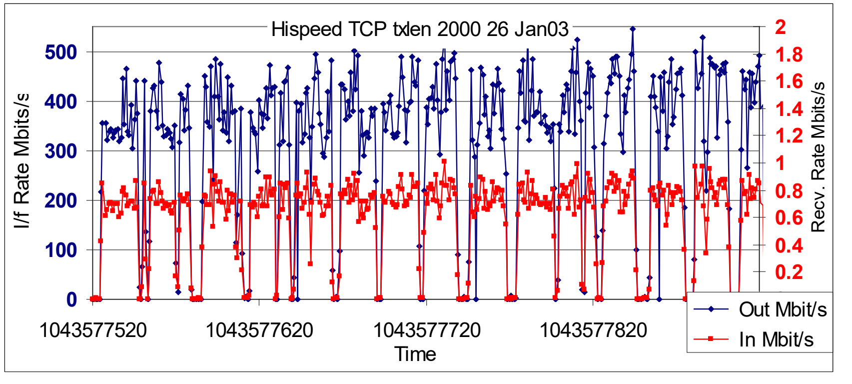
- HSTCP adaptively increases *cwnd*, and adaptively decreases *cwnd*.
- The larger the *cwnd*, the larger the increment, and the smaller the decrement.

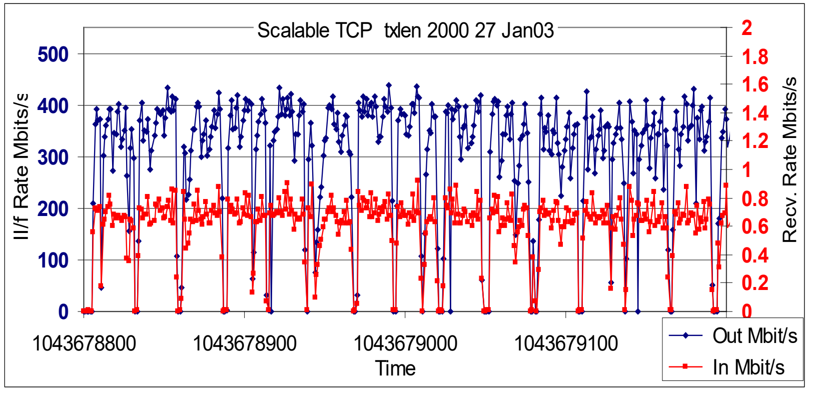


Some Measurements of Throughput CERN -SARA

- Using the GÉANT Backup Link
  - 1 GByte file transfers
    - Blue Data
    - Red TCP ACKs
  - Standard TCP
    - Average Throughput 167 Mbit/s
    - Users see 5 - 50 Mbit/s!
- High-Speed TCP
  - Average Throughput 345 Mbit/s
- Scalable TCP
  - Average Throughput 340 Mbit/s





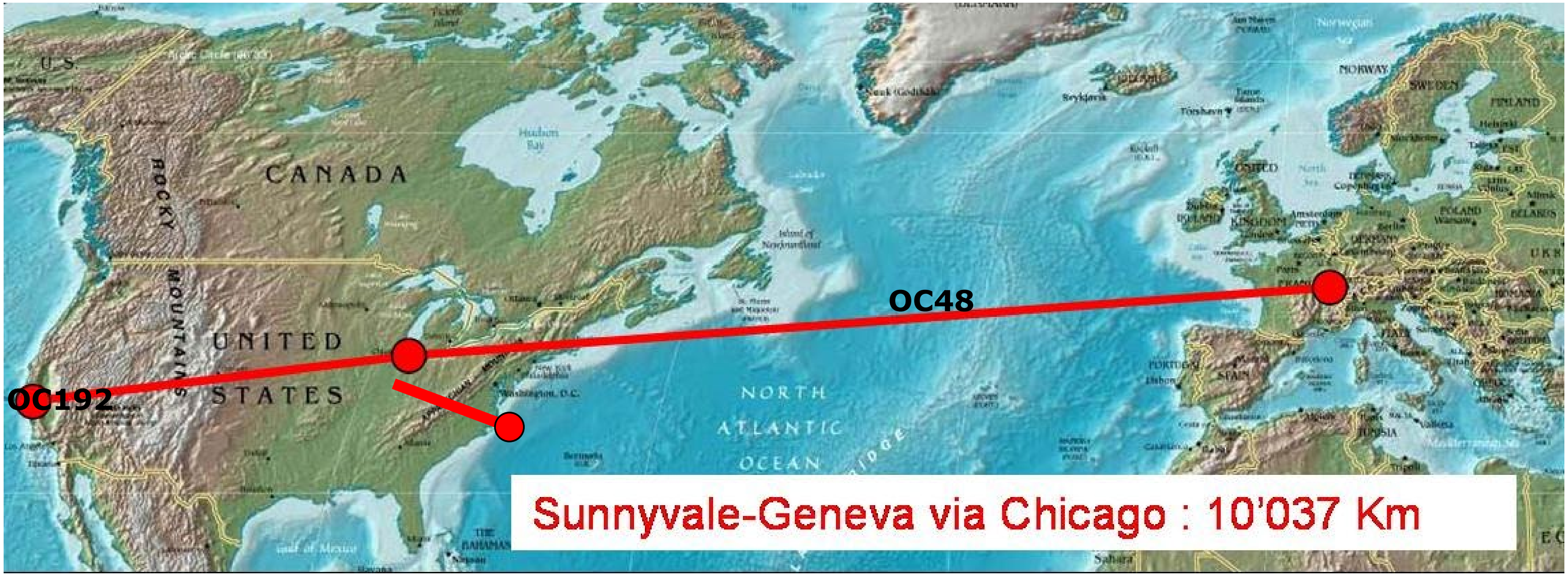


TCP FAST

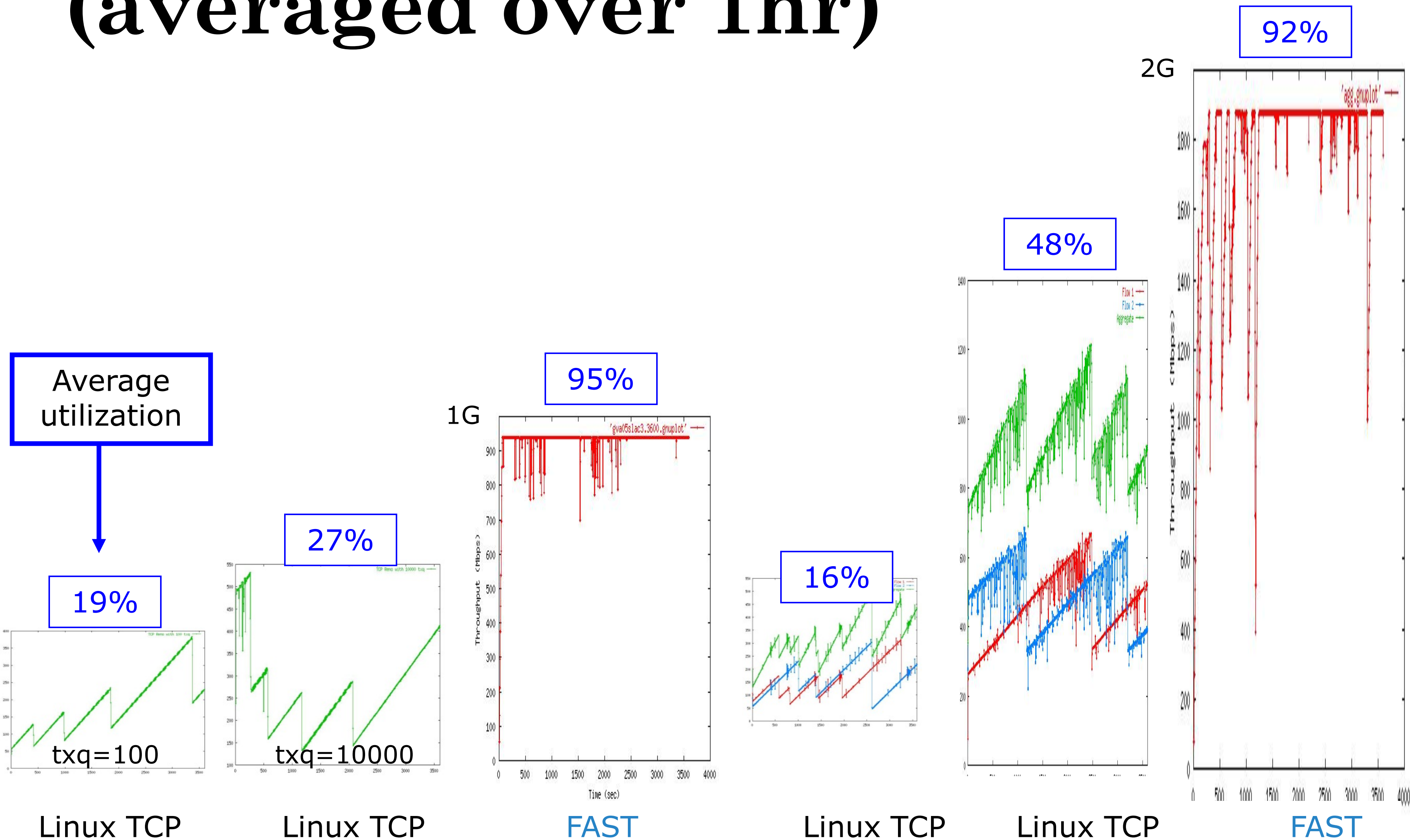
- Packet Losses give binary feedback to the end user .
- Binary feedback induces oscillations.
- Need multi-bit feedback to improve performance.
- Like TCP Vegas FAST TCP uses delays to infer congestion.
- The window is updated as follows.

$$w = w + \min[2w, (1-\gamma) + \gamma(\frac{baseRTT}{RTT} w + \alpha)]$$

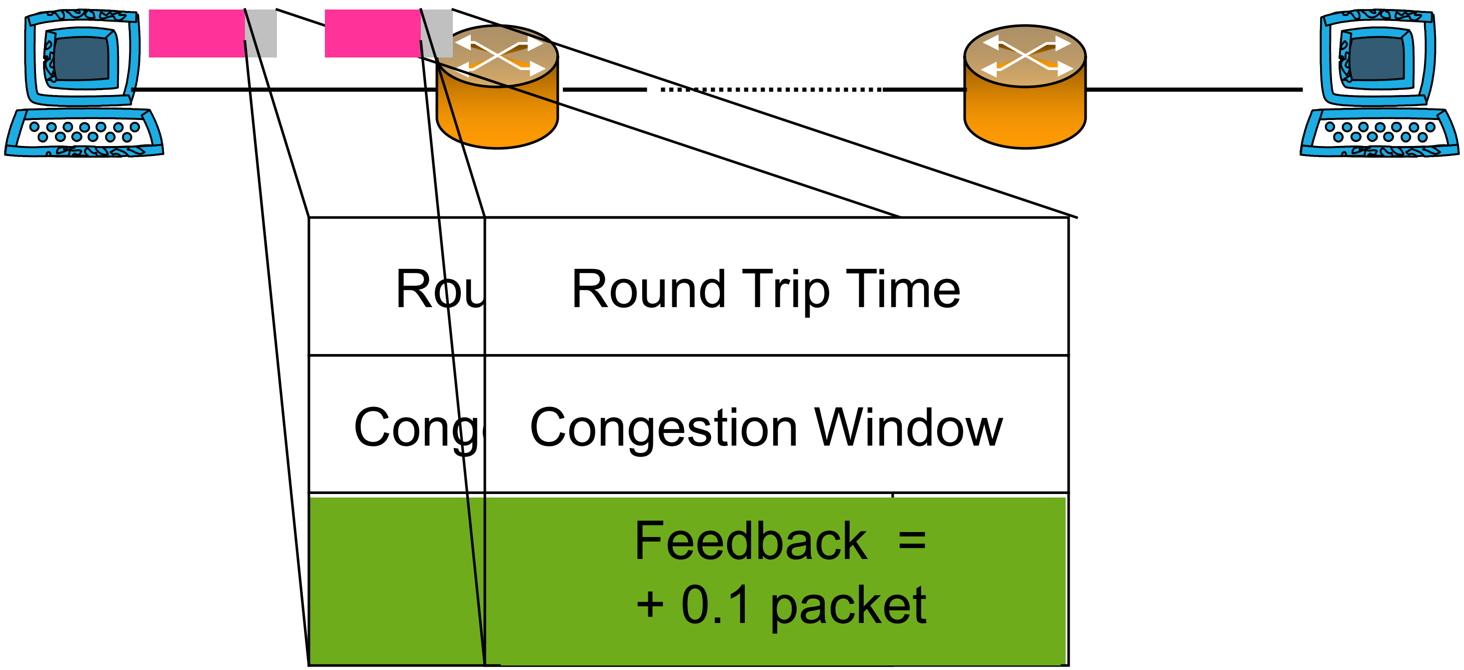
SC2002 Network



FAST throughput (averaged over 1hr)

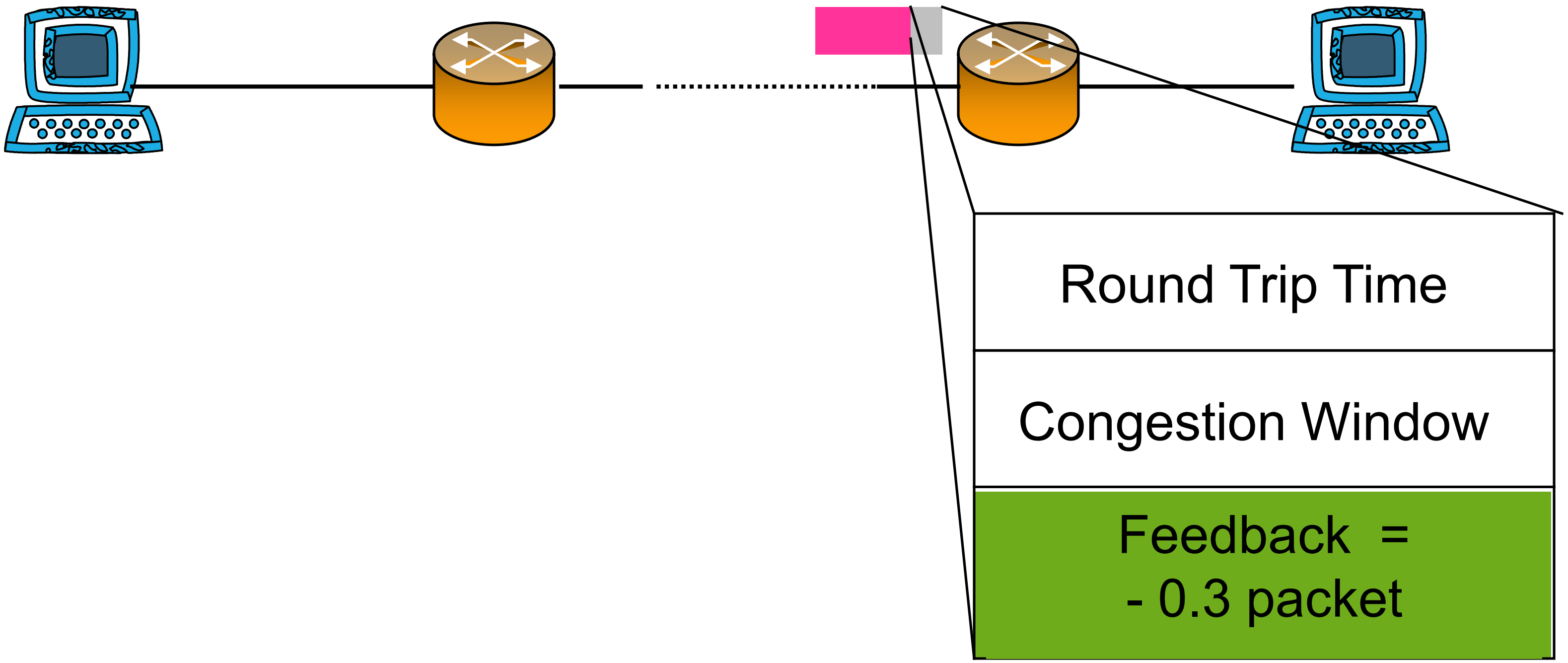


The XCP Protocol

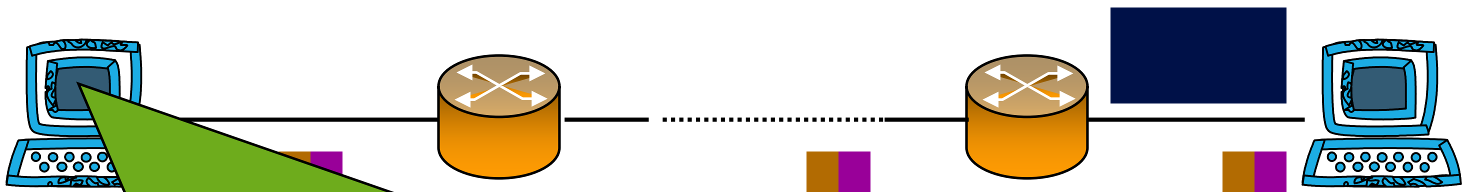


Congestion Header

How does XCP Work?



How does XCP Work?

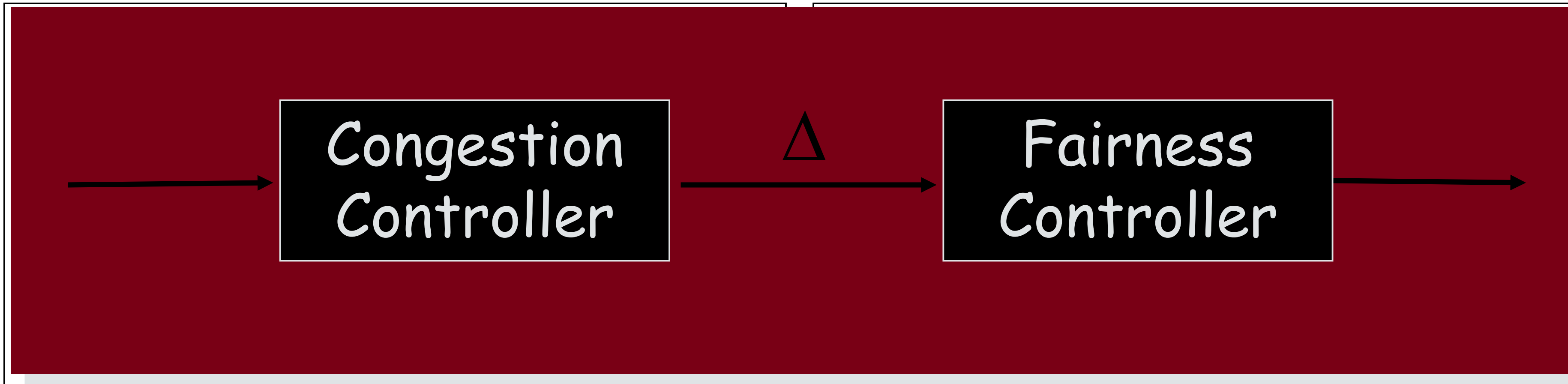


Congestion Window = Congestion Window + Feedback

XCP extends ECN and CSFQ

Routers compute feedback without any per-flow state

How Does an XCP Router Compute the Feedback?



queue

MIMD

Algorithm:

Aggregate traffic changes by Δ

Δ ~ Spare Bandwidth

Δ ~ - Queue Size

So, $\Delta = \alpha\, d_{avg}\, Spare - \beta\, Queue$

Congestio

AIMD

Algorithm:

If $\Delta > 0 \Rightarrow$ Divide Δ equally between flows

If $\Delta < 0 \Rightarrow$ Divide Δ between flows proportionally to their current rates

Getting the devil out of the details …

Congestion Controller

$\Delta = \alpha\, d_{avg}\, Spare - \beta\, Queue$

Theorem: System converges to optimal utilization (i.e., stable) for any link bandwidth, delay, number of sources if:

$$0 < \alpha < \frac{\pi}{4\sqrt{2}} \quad and \quad \beta = \alpha^2 \sqrt{2}$$

No Parameter Tuning

Fairness Controller

Algorithm:

If $\Delta > 0 \Rightarrow$ Divide Δ equally between flows

If $\Delta < 0 \Rightarrow$ Divide Δ between flows proportionally to their current rates

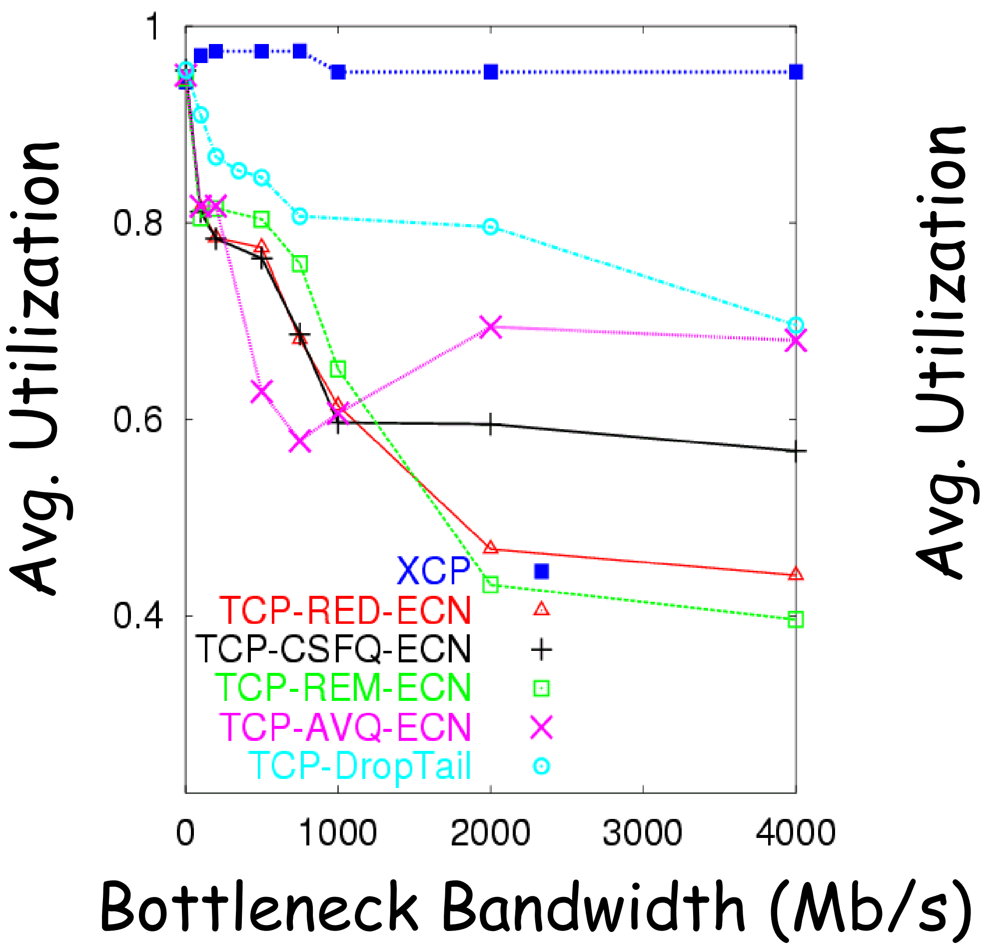
Need to estimate number of flows N

$$N = \sum_{pkts\ in\ T} \frac{1}{T \times (Cwnd_{pkt} / RTT_{pkt})}$$

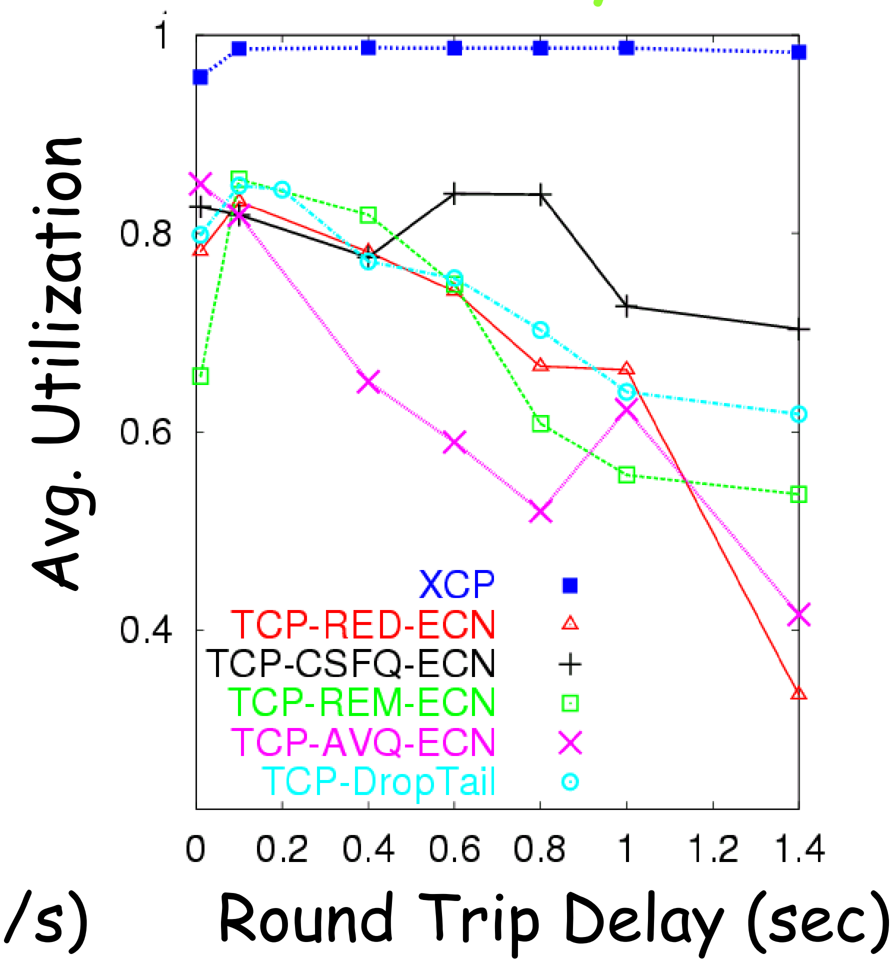
No Per-Flow State

XCP Remains Efficient as Bandwidth or Delay Increases

Utilization as a function of Bandwidth

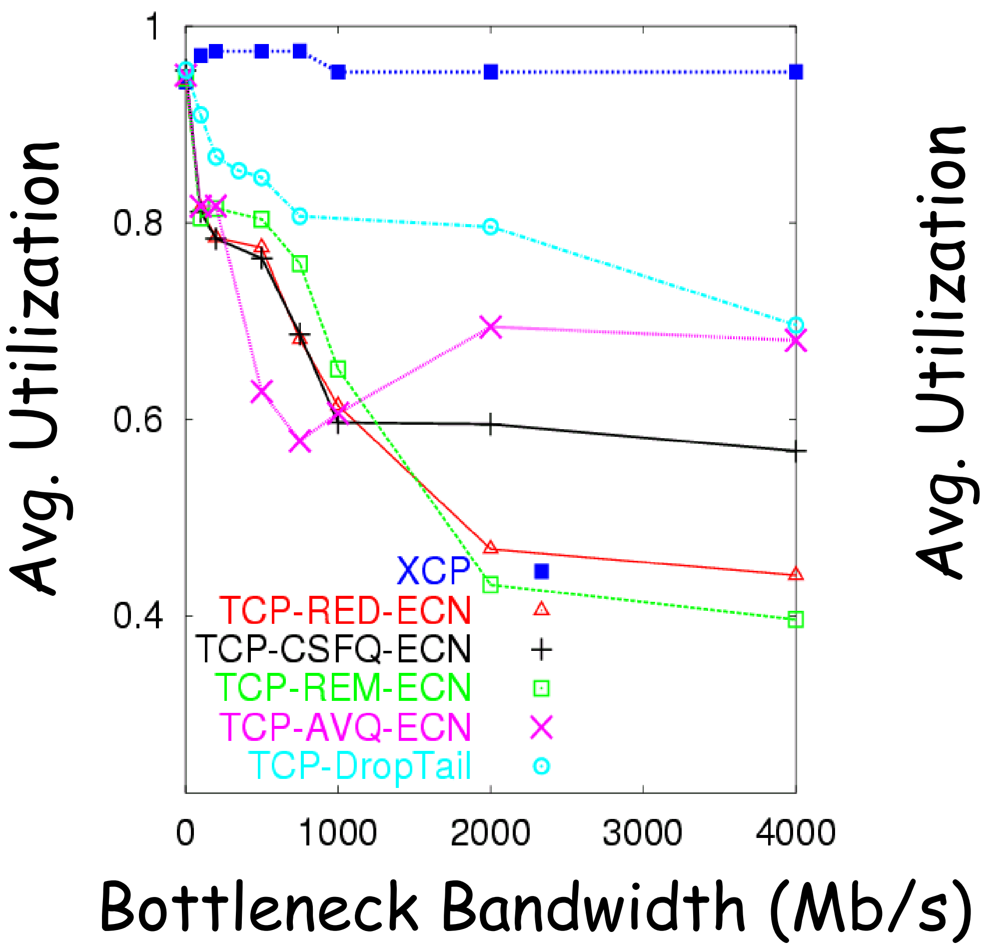


Utilization as a function of Delay

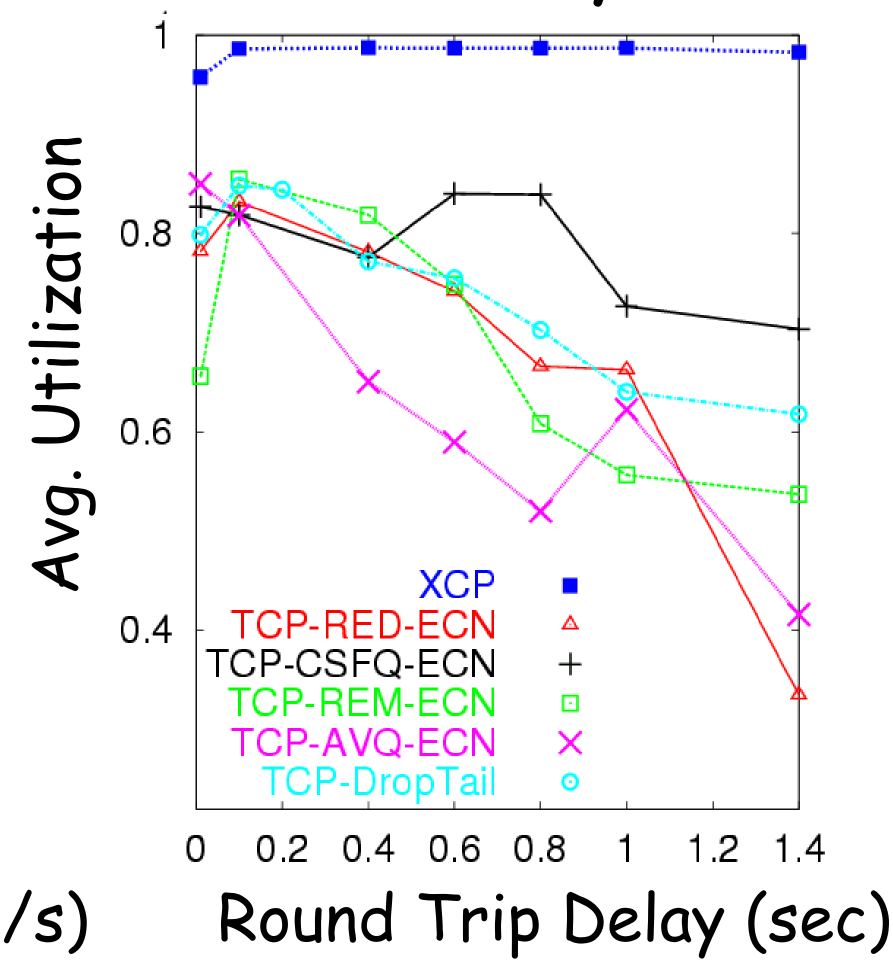


XCP Remains Efficient as Bandwidth or Delay Increases

Utilization as a function of Bandwidth



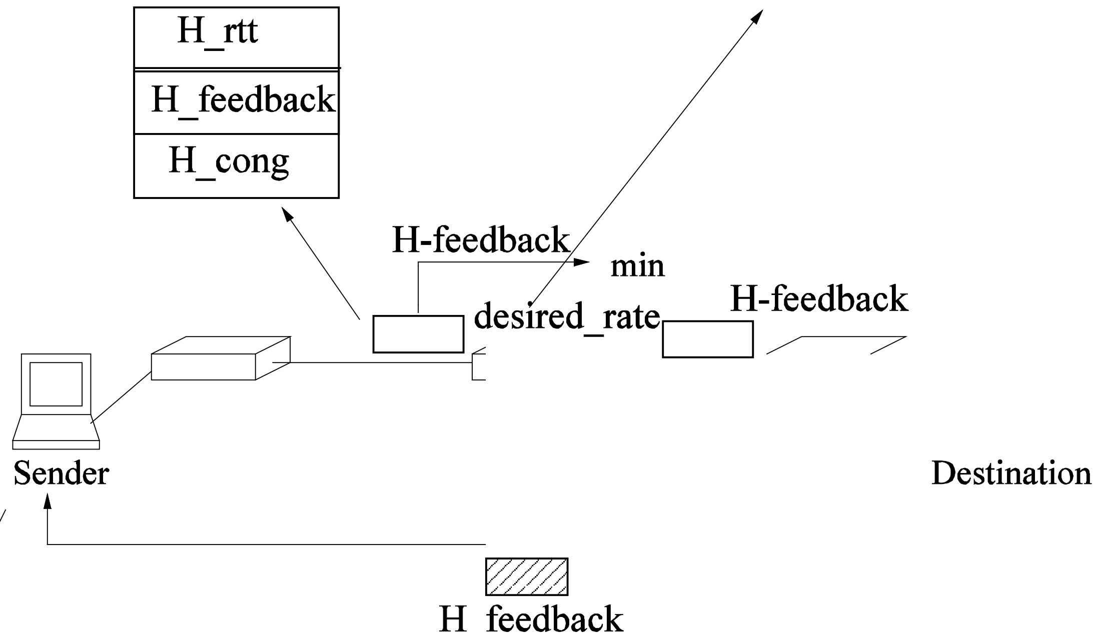
Utilization as a function of Delay



The ACP protocol

excess_capacity=0.98 link_capacity-incoming traffic

desired_rate=Pr[desired_rate+1/N_hat(k_i excess_capacity-k_q*queue_bytes)]



$$\text{desired\_window}=\frac{\text{H\_feedback*mrtt}}{\text{size}}$$

cwnd=Pr[cwnd+0.1/cwnd(desired_window-cwnd)] if H_cong=1

cwnd=Pr[cwnd+1/cwnd(desired_window-cwnd)]   otherwise

Responses generated by ACP