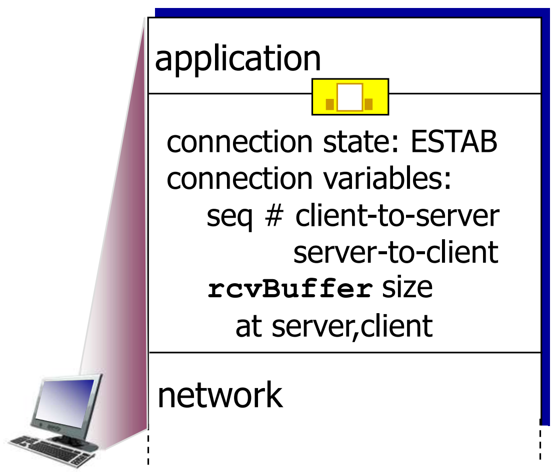
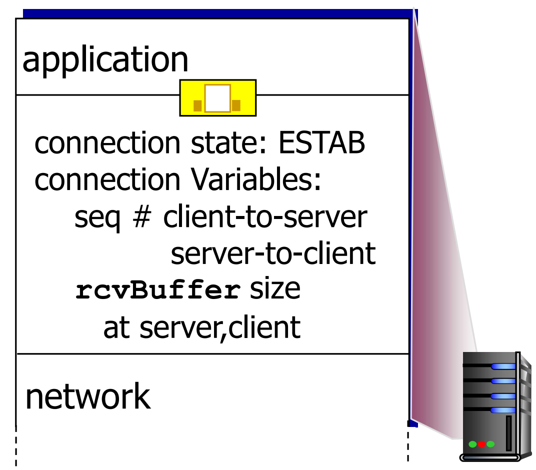
Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



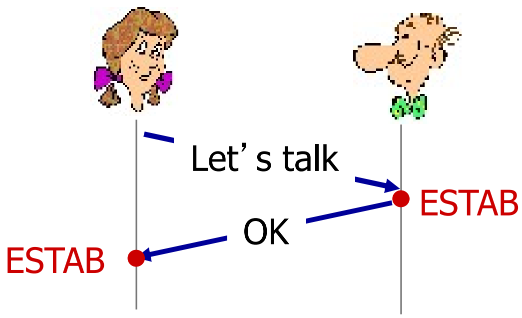

```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

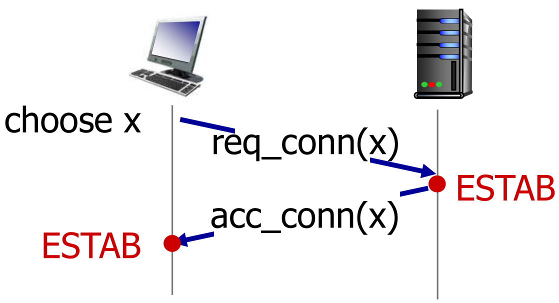



```
Socket connectionSocket =
  welcomeSocket.accept();
```

AGREEING TO ESTABLISH A CONNECTION (3 WAY HANDSHAKE)

2-way handshake:





Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- Can't "see" other side

Problems with 2-way handshake:

i) Early/Premature connection establishment -- false connections.

ii) Half-open connections – Duplicate packets from previous connection.

iii) Spoofing Problem – Bombard server with loads of unrelated data

TCP 3-WAY HANDSHAKE

client state

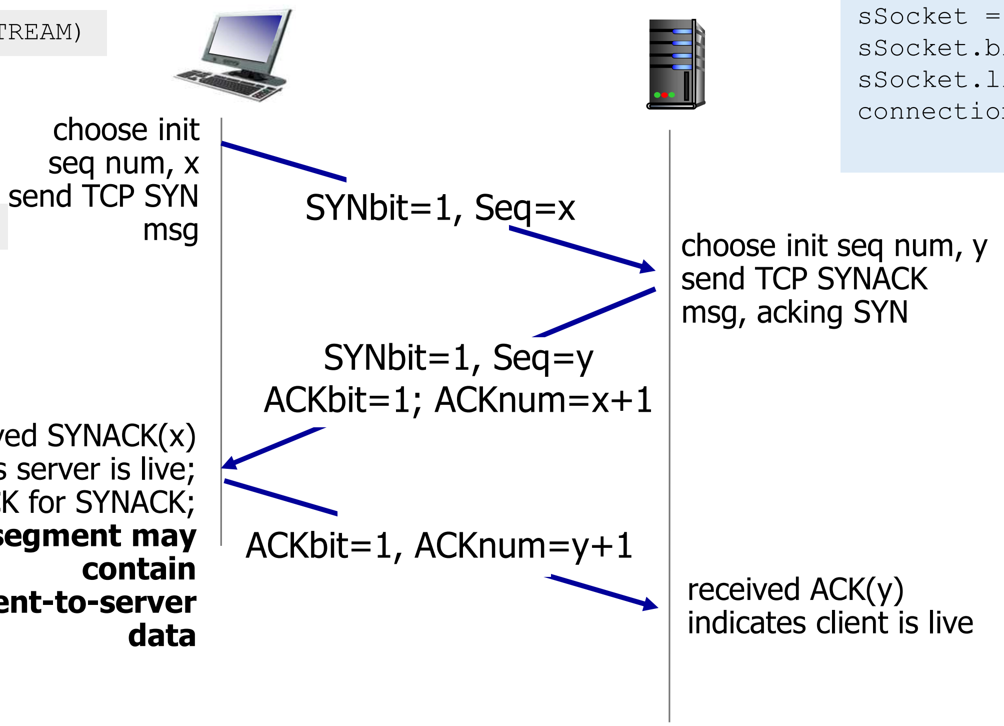
```
cSocket = socket(AF_INET,SOCK_STREAM)
```

LISTEN

```
cSocket.connect((sName,sPort))
```

SYNSENT

ESTAB



server state

```
sSocket = socket(AF_INET,SOCK_STREAM)
sSocket.bind(('',sPort))
sSocket.listen(1)
connectionSocket, Caddr = sSocket.accept()
```

LISTEN

SYN RCVD

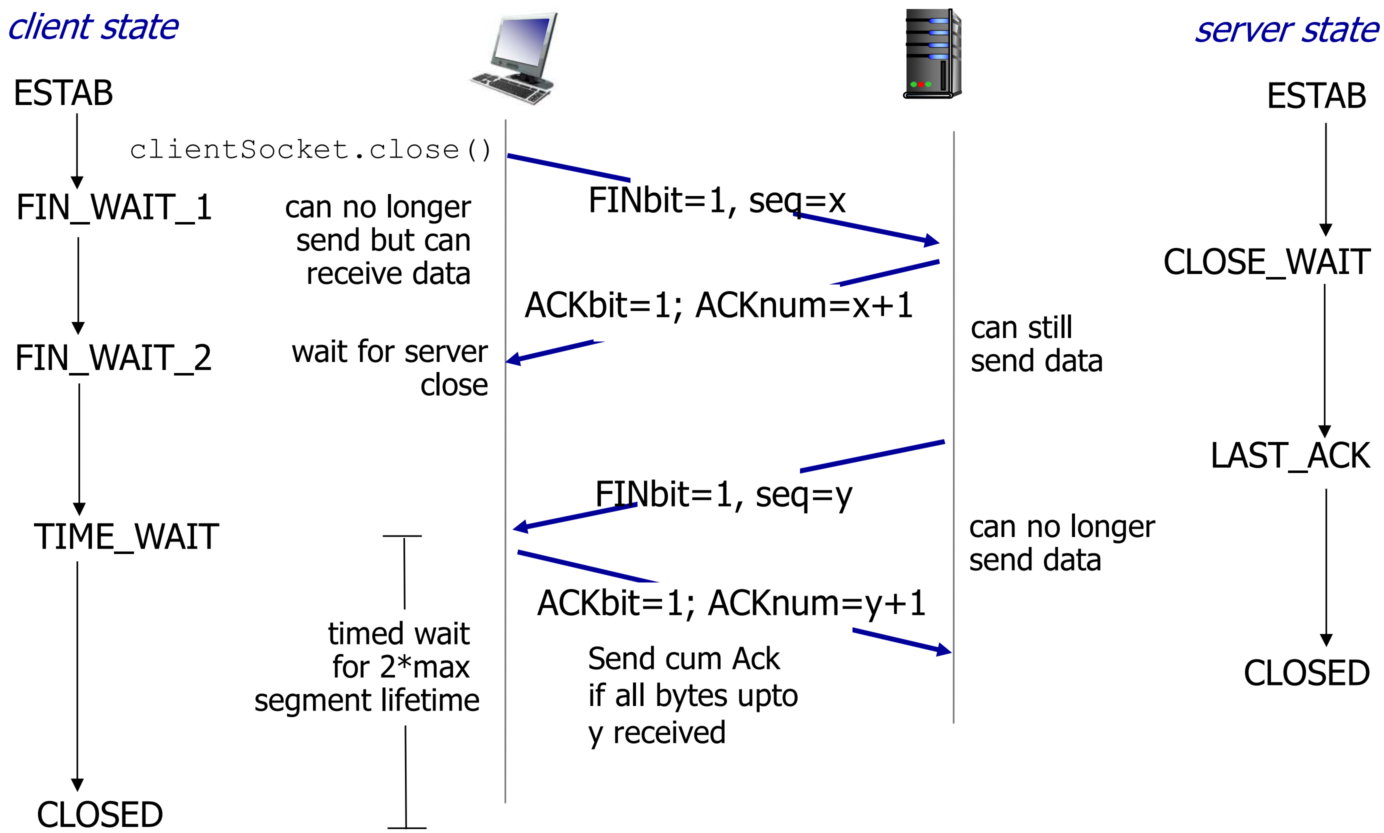
ESTAB

RFC 793: The principle reason for the three-way handshake to reduce the possibility of false connections i.e. to prevent old duplicate connection initiations from causing confusion.

+

Bi-Directional ISN Synchronization

TCP: CLOSING A CONNECTION



- Important to pick ISN appropriately; Choosing a fixed ISN is undesirable.
- Pick random value (intended to avoid others predicting ISN for a new connection)

TCP SEQ. NUMBERS, ACKS

sequence numbers:

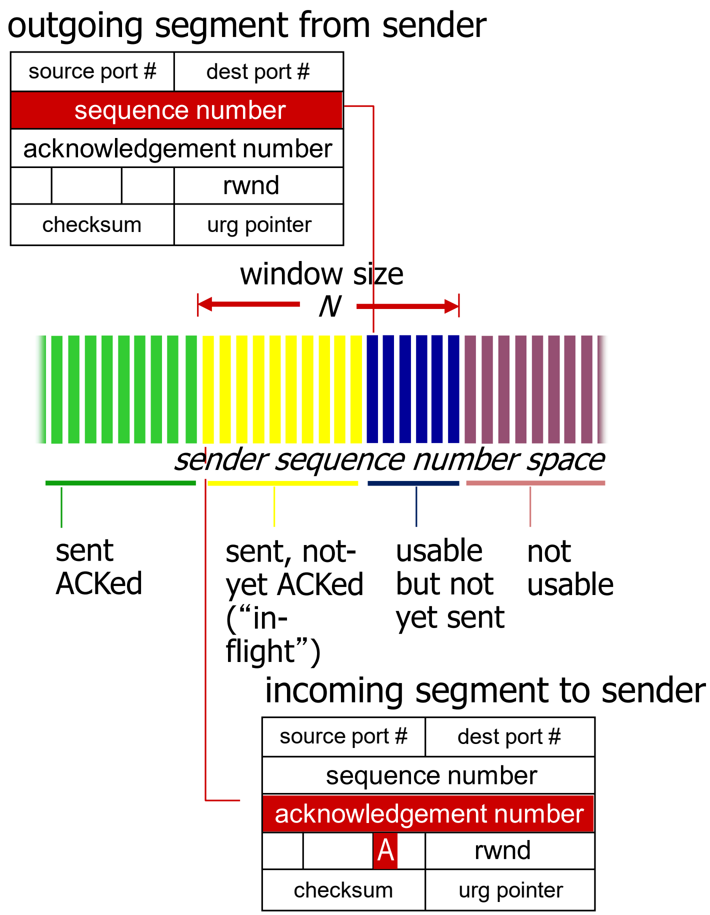
- byte stream "number" of the first byte in segment's data

acknowledgement number:

- seq # of **next** byte expected from other side
- Support cumulative ACK

Q: how should the receiver handle out-of-order segments?

- A: TCP spec doesn't say, - it is up to the implementor..

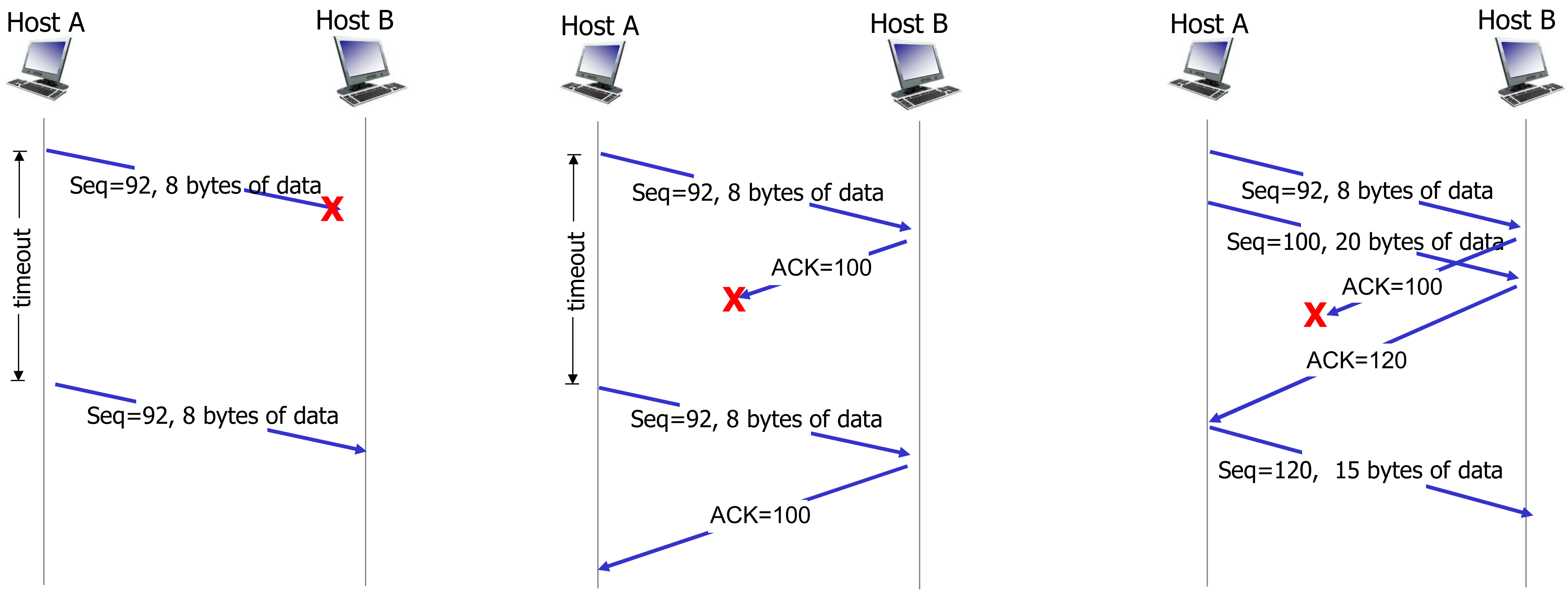


Q: How do you detect loss of packets ?

And

How do you recover from packet loss?

TCP: Packet Loss scenarios



Packet loss scenario

lost ACK scenario

cumulative packets with lost ACK

TCP round trip time (RTT) and Retransmission timeout (RTO)

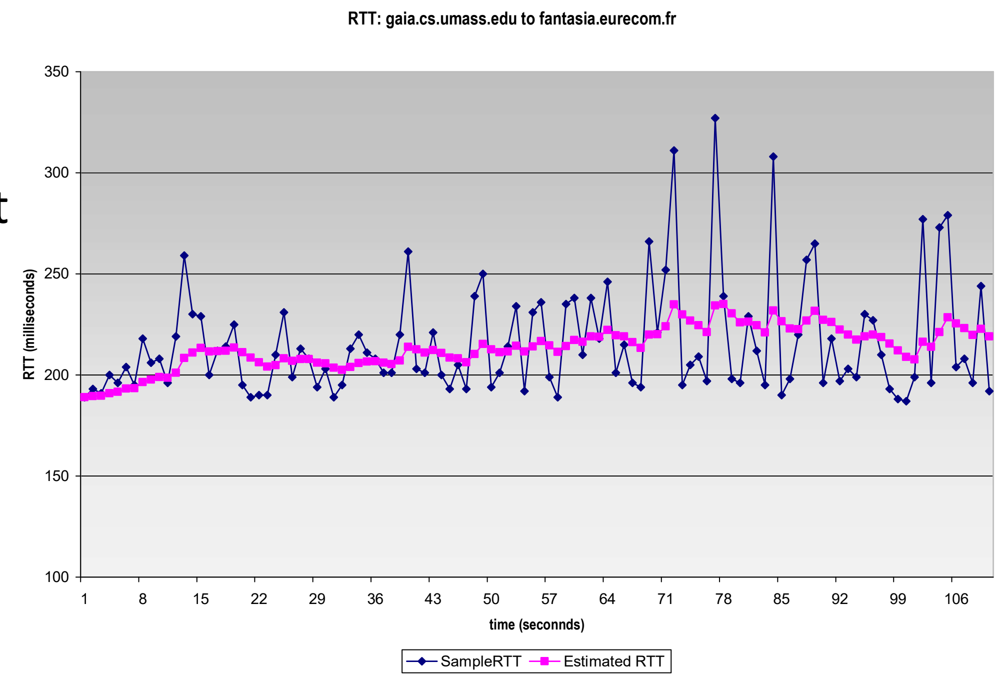
Q: how to set TCP timeout value?

RTO: Retransmission Timeout – duration to wait before retransmission in the absence of any ack for data.

- ❖longer than RTT - but RTT varies over time.
- ❖*too short:* premature timeout, unnecessary retransmissions.
- ❖*too long:* slow reaction to segment loss.

Q: how to estimate RTT?

- **SampleRTT (R)**: measured time from segment transmission until ACK receipt.
  - SampleRTT will vary, we want estimated RTT to be "smoother" using EWMA ($\alpha = 0.125$)
  - average several *recent* measurements, not just current **SampleRTT**
  - ignore retransmissions



SampleRTT = RTT$_i$; measured for every sample (#Seq <-> ACK pair)

TCP ROUND TRIP TIME, TIMEOUT [RFC 6298]

Initialization: Until SampleRTT is measured set *RTO =1s.*

When the first RTT measurement R_0 is made, the host MUST set

SRTT ← R_0

RTTVAR ← $R_0/2$

RTO ← SRTT + MAX (G, K*RTTVAR), where K = 4 and G is the clock granularity.

- For subsequent samples of RTT (R_i) where i>=1:
- First estimate SampleRTT *deviation (RTTVAR)* from current SampleRTT and past SmoothedRTT;
- Then estimate SmoothedRTT (SRTT)

$$\mathbf{RTTVAR}_{(i)} = (1-\beta)*\mathbf{RTTVAR}_{(i-1)} + \beta*|\mathbf{R}_{(i)}-\mathbf{SRTT}_{(i-1)}|$$
$$\mathbf{SRTT}_{(i)} = (1-\alpha)*\mathbf{SRTT}_{(i-1)} + \alpha*\mathbf{R}_{(i)}$$

(typically, β = 0.25 and α = 0.125)

- timeout interval: **SmoothedRTT** plus "safety margin"
  - large variation in **SmoothedRTT ->** larger safety margin

TimeoutInterval (RTO) = *SmoothedRTT* + 4*DevRTT

estimated RTT

"safety margin"

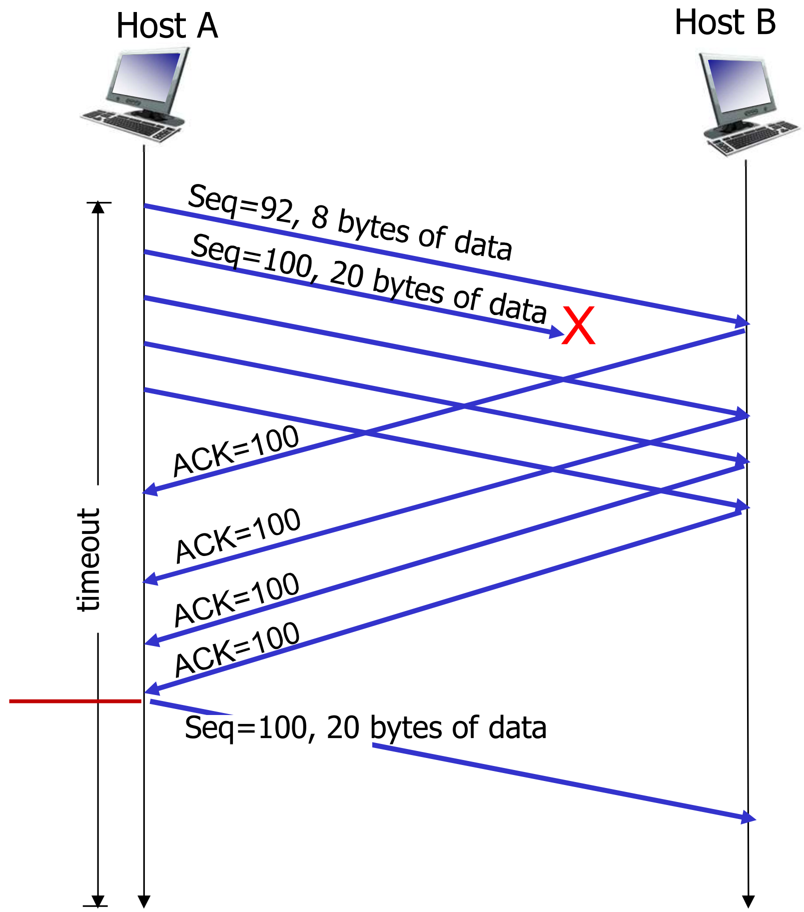
TCP FAST RETRANSMIT

TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

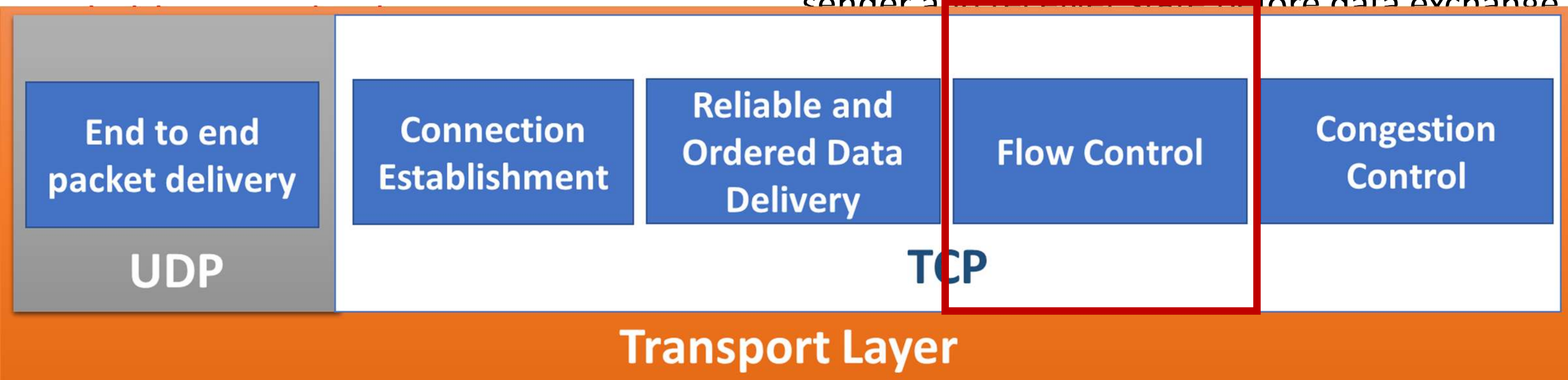
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



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

- point-to-point:
  - one sender, one receiver

- connection-oriented:
  - handshaking (exchange of control msgs) inits sender and receiver state before data exchange



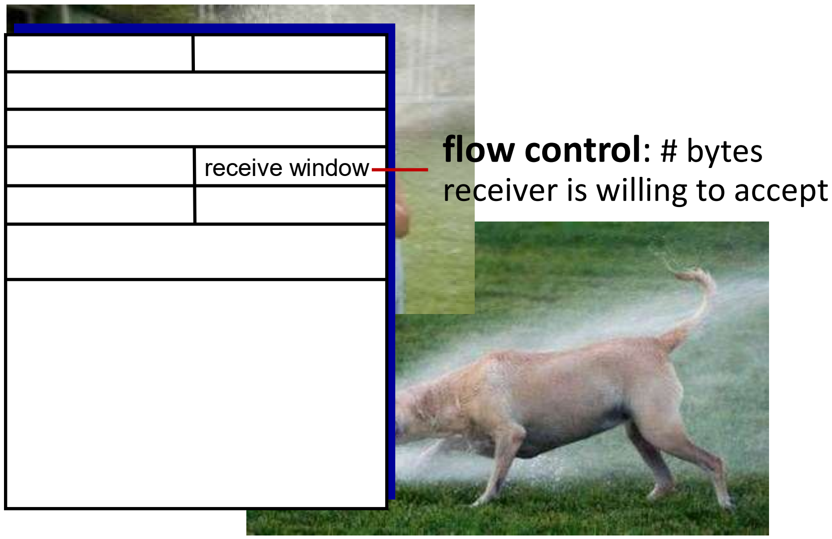
to the network capacity.

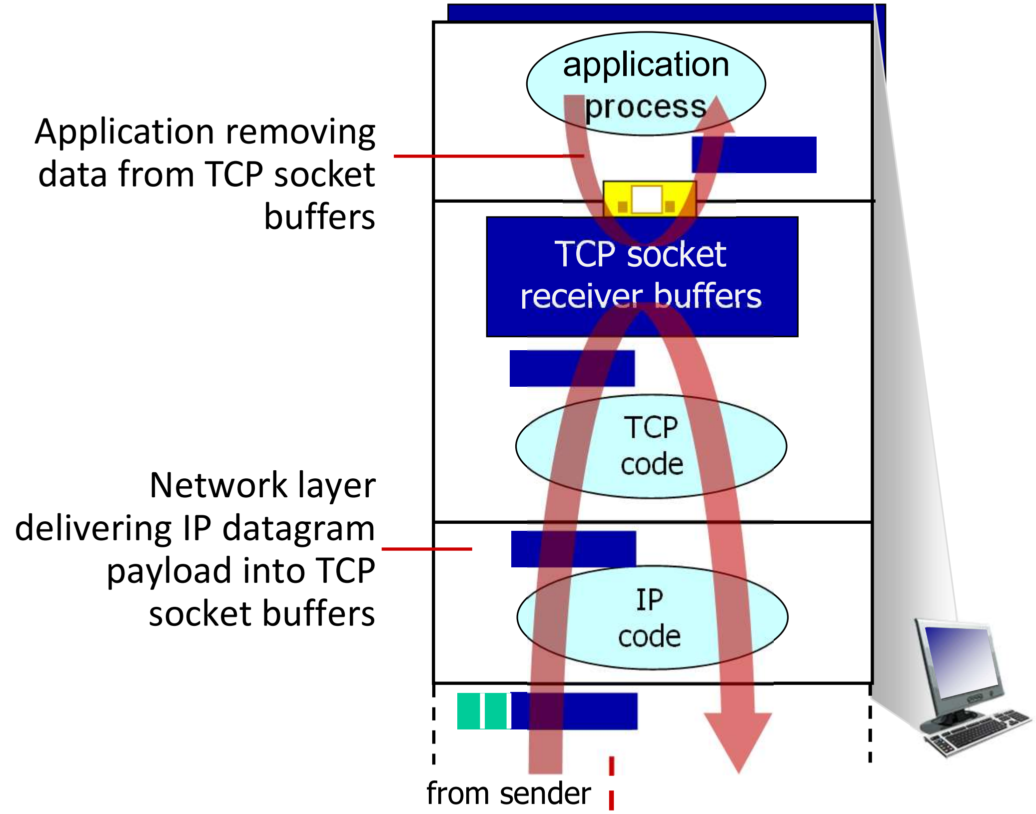
- TCP congestion and flow control set window size

Today's Focus:
Flow Control
Congestion Control

TCP FLOW CONTROL

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?





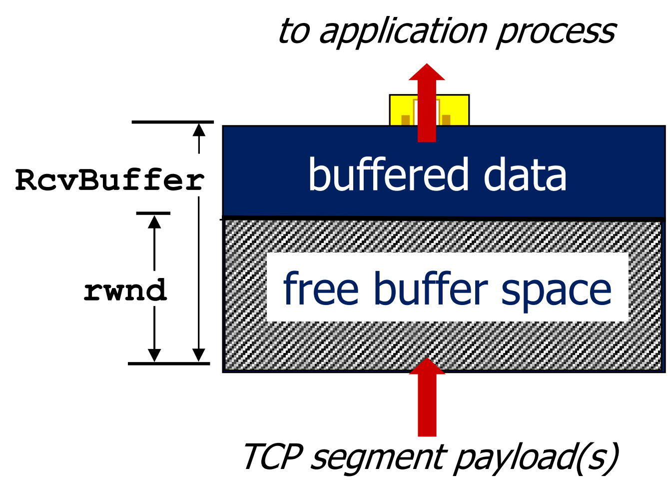
receiver protocol stack

flow control

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

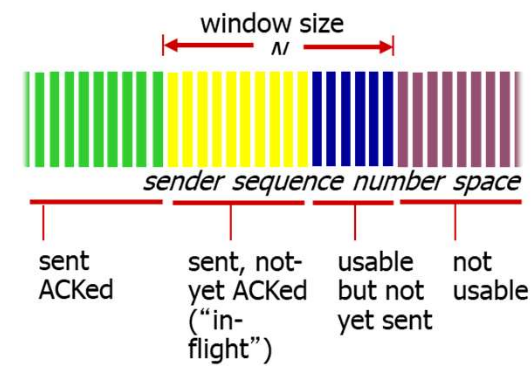
TCP FLOW CONTROL

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (default 64K bytes)
  - many operating systems auto adjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow



receiver-side buffering

Receiver:
rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]



Sender:
LastByteSent – LastByteAcked <= rwnd

Window Advancement

- Advancing a full window
  - When the receive window fills up, how to get started again?
    - Sender sends periodic probe to check availability while recv. window=0
- 'Silly Window Syndrome': Very small sized window ➔ small data transfer.
  - receiver requests sender to reduce the amount of data sent at a time.
  - Ack. with-holding at the receiver (delayed acks) helps, but not complete solution.
- 'Tinygram Syndrome': small data at the sender ➔ small data transfer.
  - Application at the sender has very small amount of data to be sent at a time.
  - Coalesce multiple chunks of small data into large segment.
  - Nagle's algorithm: RFC-896
    - If send.window < 1 MSS, delay sending the packet and aggregate/batch more data
    - Can be over-ridden by TCP-NODELAY option.

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

- point-to-point:
  - one sender, one receiver

- connection-oriented:
  - handshaking (exchange of control msgs) inits sender and receiver state before data exchange

to the network capacity.

- TCP congestion and flow control set window size

Today's Focus:
Congestion Control

congestion:

- informally:

  "***too many sources sending too much data too fast for network to handle***"

Deadlock and/or a Livelock

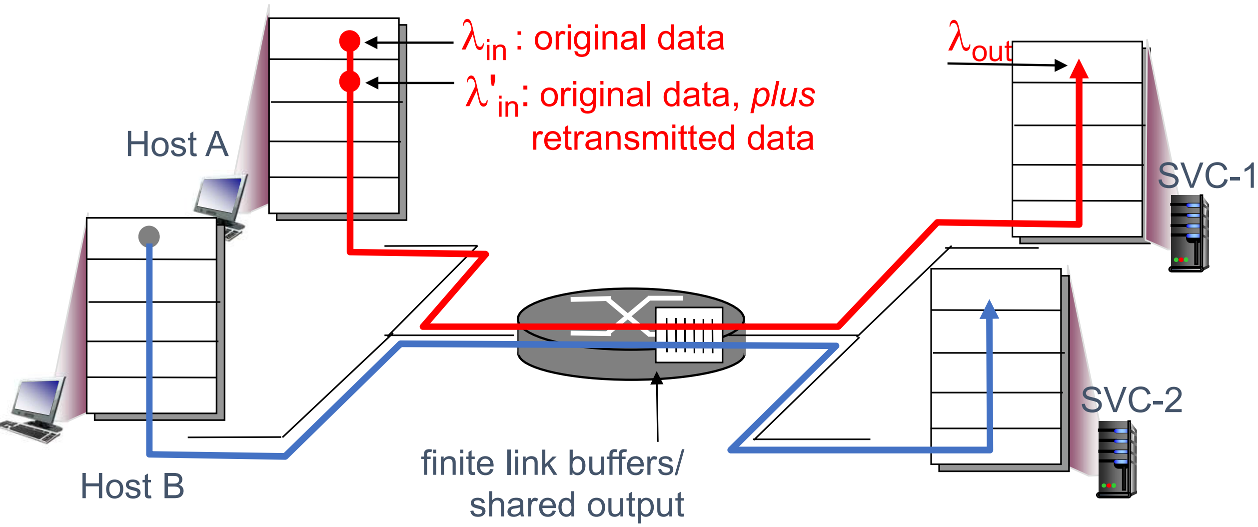
Congestion – we are familiar with…

- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 networking problem!

congestion: **<*Overload on Network*>**

CAUSES AND COSTS OF CONGESTION

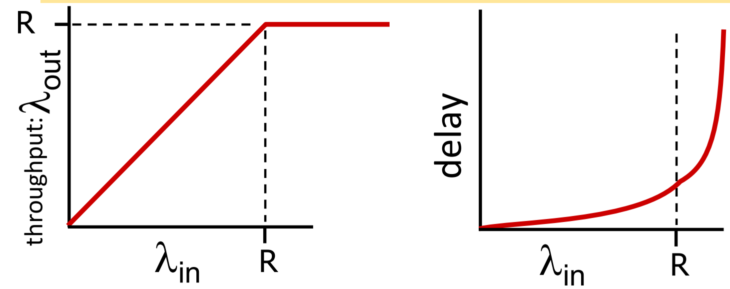
- application-layer input: λ_{in} = application-layer output λ_{out}
- Sender retransmits lost, timed-out packet(s).
- transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



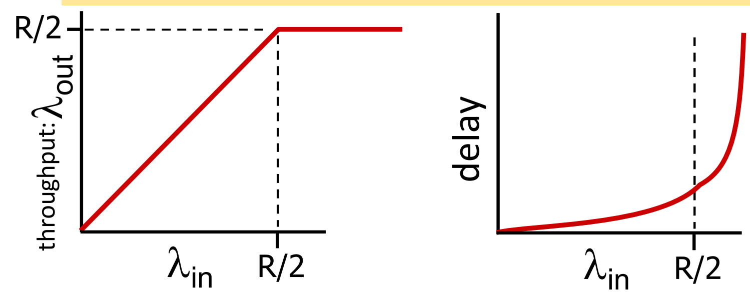
"costs" of congestion:

- Large delays (more queue ➔ more delay)
- more work (lost packets ➔ retransmissions)
- lower "goodput" or achieved receiver throughput.
- duplicates: link carries multiple copies of a packet.

Q1: What causes congestion?



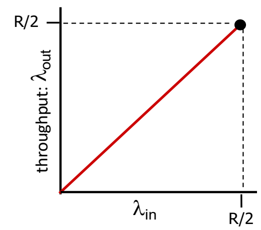
Q2: What happens as more hosts join?



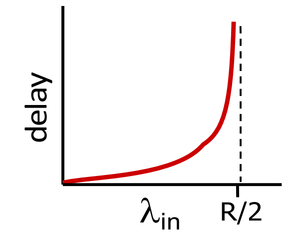
Q3: What is/are the cost(s) of congestion?

NETWORK CONGESTION: KEY INSIGHTS

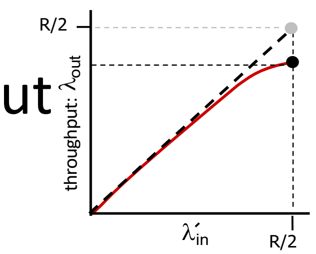
- throughput can never exceed capacity



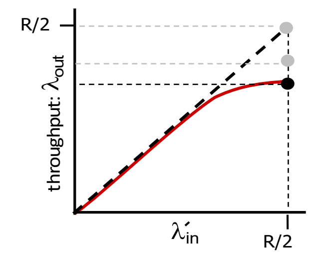
- delay increases as capacity approached



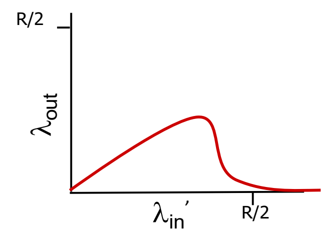
- loss/retransmission decreases effective throughput



- un-needed duplicates further decreases effective throughput



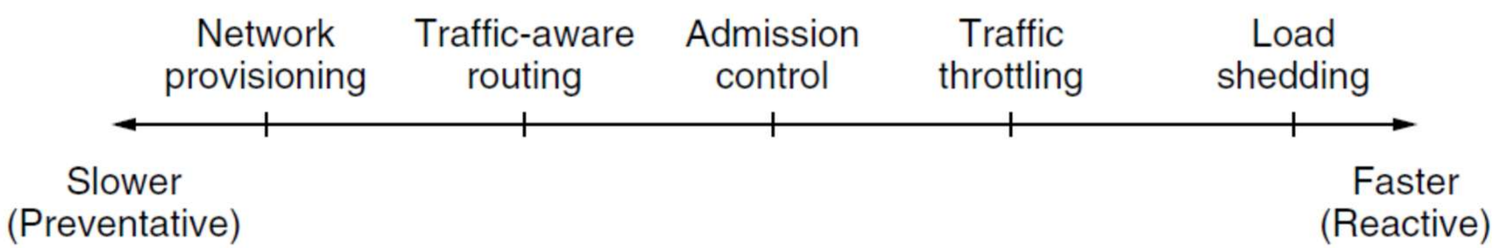
- upstream transmission capacity / buffering wasted for packets lost downstream



Typical Approaches to Congestion Control

Reduce the network load

Increase the network resources



APPROACHES TOWARDS CONGESTION CONTROL

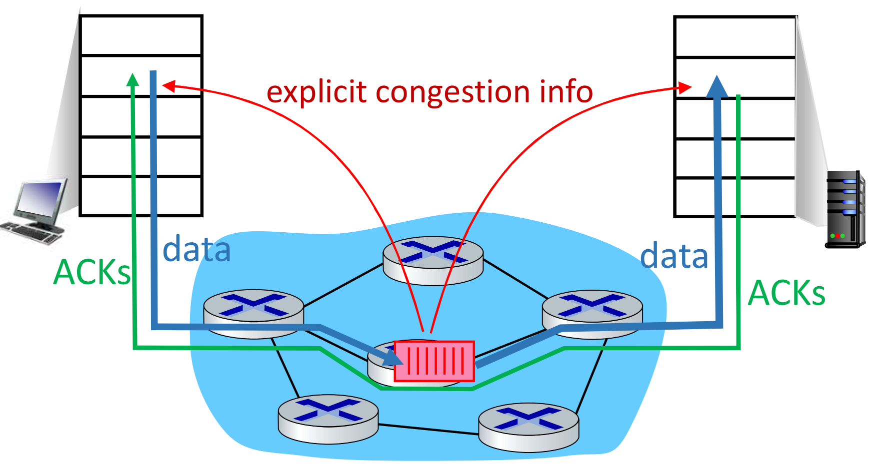
two broad approaches towards congestion control:

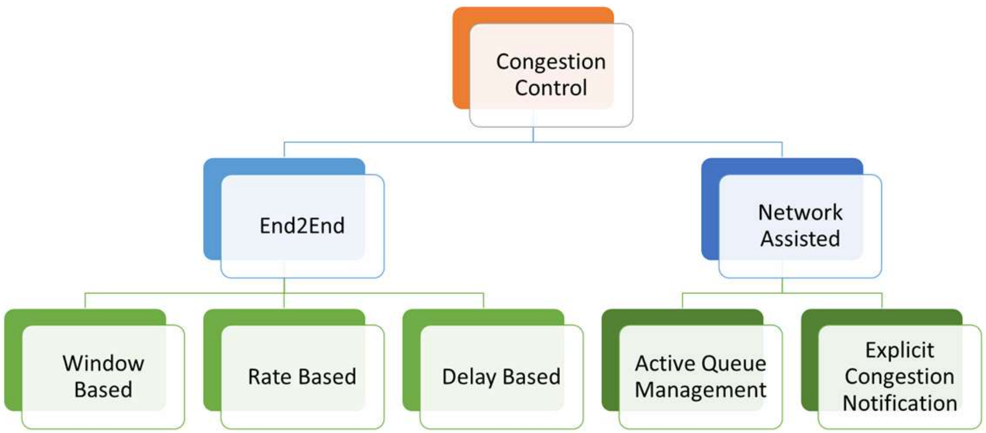
end-end congestion control:

- congestion inferred from end-system observed loss, delay.
- no explicit feedback from network

network-assisted congestion control:

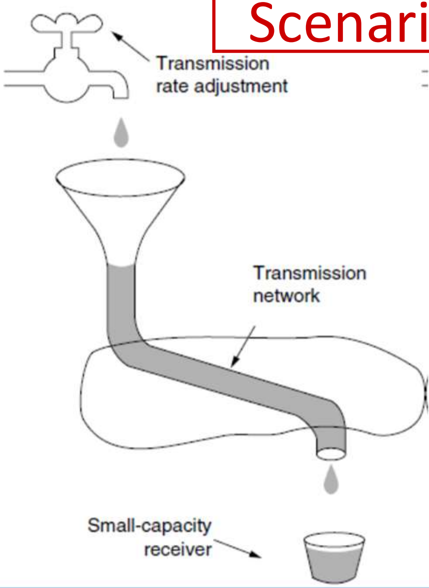
- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Convey explicit rate for sender
- Sender and receiver react to the feedback





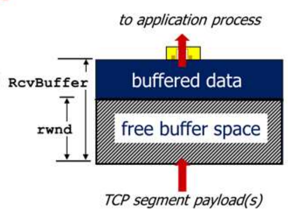
Congestion Control – A few more Basics

Scenario 1: A familiar one



TCP flow control

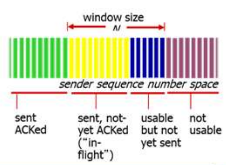
- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (default 64K bytes)
  - many operating systems auto adjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



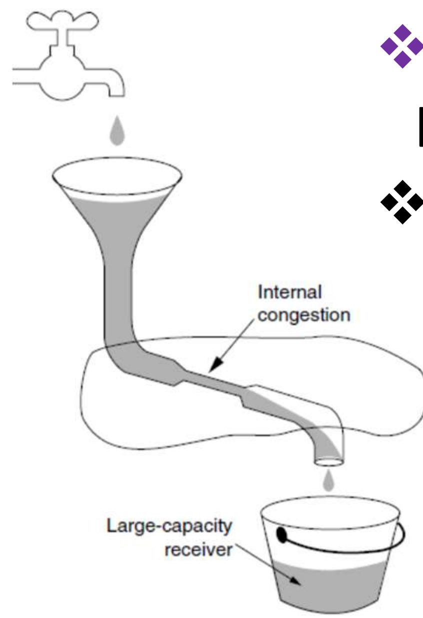
receiver-side buffering

Receiver:
rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]

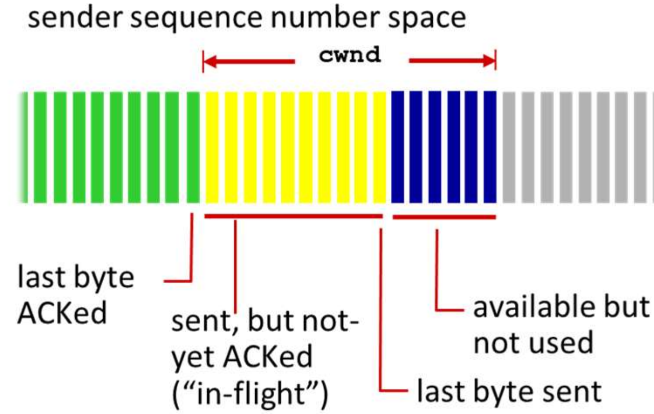
Sender:
LastByteSent – LastByteAcked <= rwnd



Scenario 2: Network Congestion



- **cwnd** is dynamic, function of perceived network congestion
- sender limits transmission:



LastByteSent-LastByteAcked ≤ Min(Rwnd, cwnd)

- *TCP sending rate:* cwnd bytes every RTT

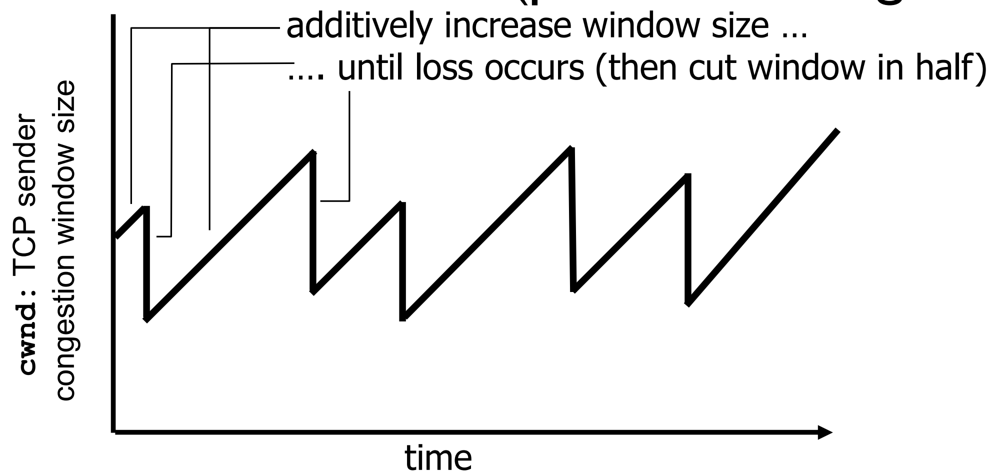
$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

On average, over a RTT time period.

TCP CC: Additive Increase and Multiplicative Decrease (AIMD)

- *Approach: Feedback control loop:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs.
  - *additive increase:* increase **cwnd** by **1** MSS every RTT until loss is detected.
  - *multiplicative decrease:* cut **cwnd** in half after loss (perceived congestion).

AIMD saw tooth behavior: probing for bandwidth



- Why AIMD?
  - AI: Gradual capacity/bandwidth probing with low potential for loss; improved fairness.
  - MD: Since increased congestion is more catastrophic, reduce it more aggressively.

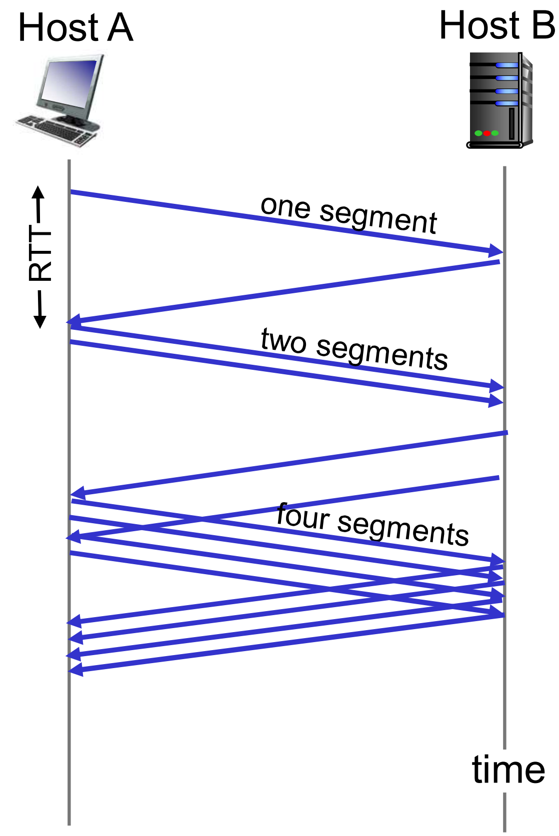
TCP CONGESTION CONTROL: DEFINING AND ADAPTING THE CWND, RFC 5681

Q1: What should a congestion window (cwnd) start with?

Initially we can set **cwnd** = 1 MSS (minimum/safe value)

```
CWND = (MSS > 2190)? (2*MSS): (MSS > 1095)? (3*MSS): (4*MSS)
```

Q2: How exactly to grow/shrink the CWND?



❖ When connection begins, increase the rate exponentially /linearly until a ***certain threshold*** or the ***first loss event***:

- Exponentially: double **cwnd** every RTT
- Linearly: increment **cwnd** by $1/\text{cwnd}$ for every ACK received.

```
If cwnd < ssthresh then
        Each time an Ack is received:
        cwnd = cwnd + 1
else: /* cwnd >= ssthresh */
        Each time an Ack is received:
        cwnd = cwnd + 1 / [ cwnd ]
endif
```

summary: **initial rate is small (slow) but ramps up exponentially fast**

TCP: Key Phases of Congestion Control: SS & CA

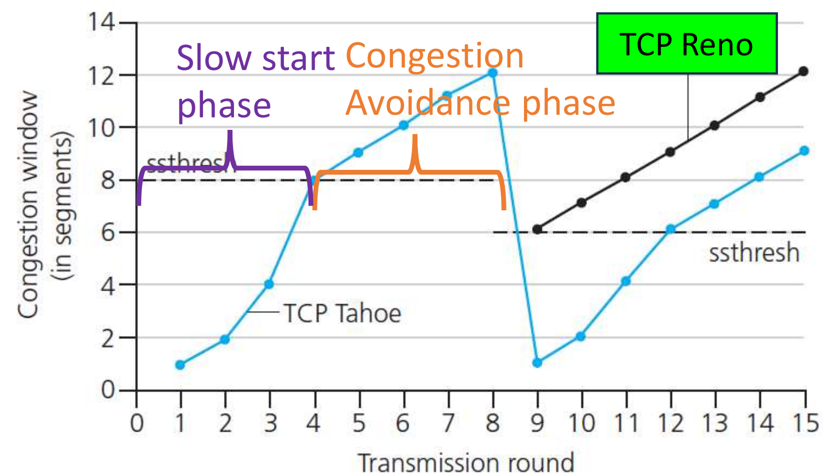
Q3: When should the exponential increase switch to linear increase?

- When we expect to be reaching near to the network capacity.

Implementation:

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

ssthresh = max(cwnd/2, 2*MSS)

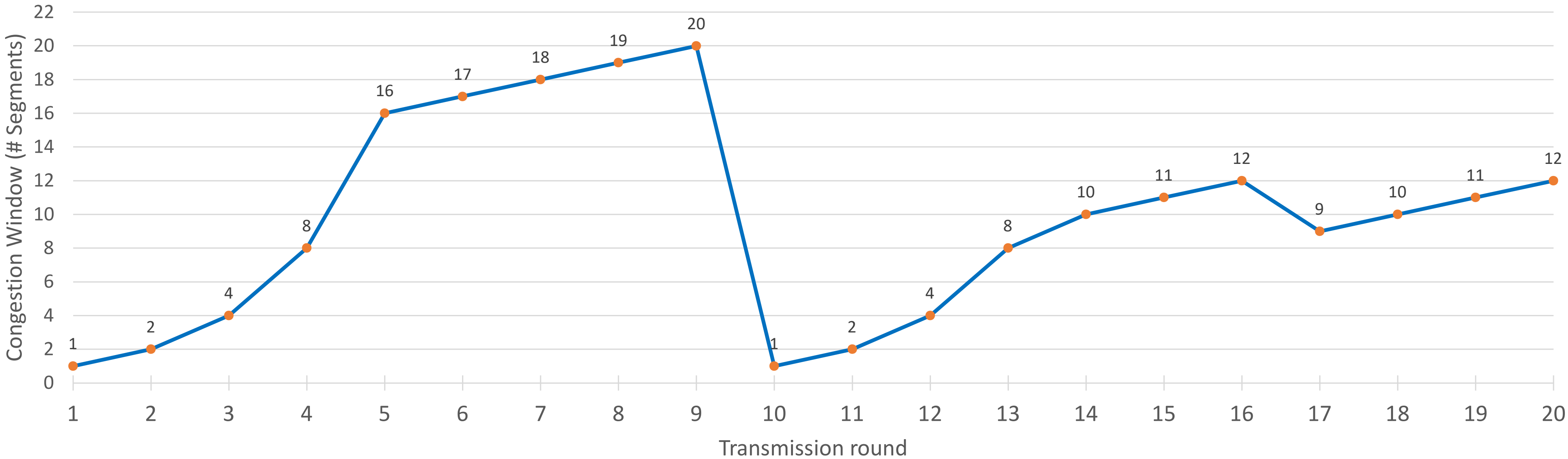



```
If Timeout then: //or 3D-ACK (Tahoe) then:
//Indicates packet loss: (MD)
       ssthresh = max(cwnd /2, 2*MSS)
       cwnd = 1 or cwnd = ssthresh + 3
Else: //regular ACK (@every ack)
       If cwnd < ssthresh then:
              cwnd = cwnd + 1
       else: /* cwnd >= ssthresh */
              cwnd = cwnd + 1 / [ cwnd ]
       endif
endif
```

*Why should we modify the **ssthresh** on loss?*

TCP CONGESTION WINDOW

Consider: RTT=1s; MSS=1000 bytes



***Q1**: What is the initial ssthresh value if any?*

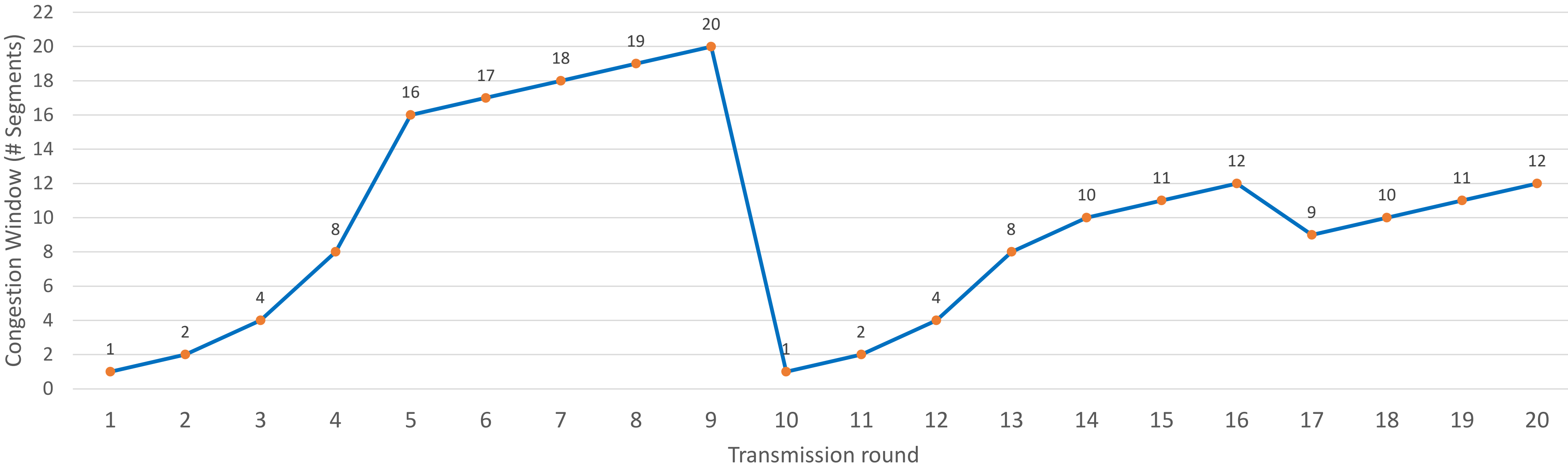
***Q2**: How many congestion events occurred?*

***Q3**: Identify the slowstart and CA phases?*

***Q4**: ssthresh value at Transmission round 12 and 18 respectively?*

TCP Congestion Window

Consider: RTT=1s; MSS=1000 bytes



***Q5**: Name the congestion control mode used?*

***Q6**: How many segments were delivered until first congestion event?*

***Q7**: Average throughput in first slow-start phase?*

***Q8**: Average throughput in first congestion avoidance phase?*

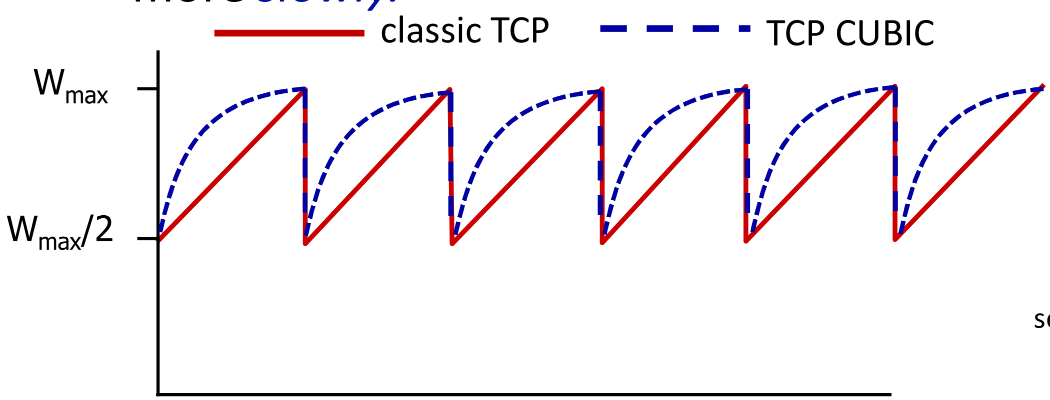
TCP Variants: TCP Tahoe to TCP BBR

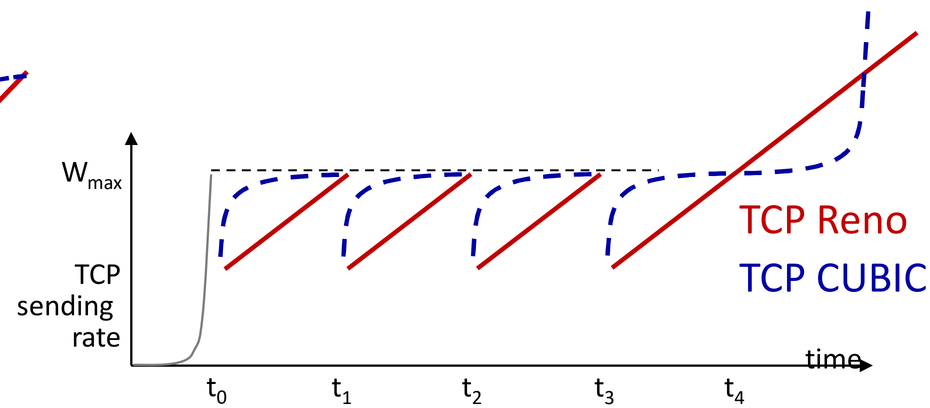
- ***TCP Tahoe:*** Introduced the congestion control mechanisms: Slowstart, Congestion Avoidance.
- ***TCP Reno***: Fast retransmit and fast recovery are added in addition to previous congestion control mechanisms.
  - Has other features -- header compression (if ACKs are being received regularly, omit some fields of TCP header).
  - Delayed ACKs -- ACK only every other segment.
- ***TCP NewReno***: Modified and Advanced Fast recovery mode.
  - Can handle multiple packet losses -- Extension likewise to TCP SACK.
- ***TCP Vegas***: Rate/Delay based Congestion Detection.
  - New Congestion Avoidance and Retransmission mechanism.
- ***TCP CUBIC***: Provides Faster Congestion response in high speed networks.
  - Default TCP congestion control setting in Linux.
- ***TCP BBR***: Recent Extension by Google (Also rate/delay based).
  - Supported in Linux v 4.9 onwards.
- Several other variants like ***TCP Fast, Hybla, Illinois, Veno, WestWood, etc.*** are not discussed.

https://en.wikipedia.org/wiki/TCP_congestion_control

TCP CUBIC [PAPER]

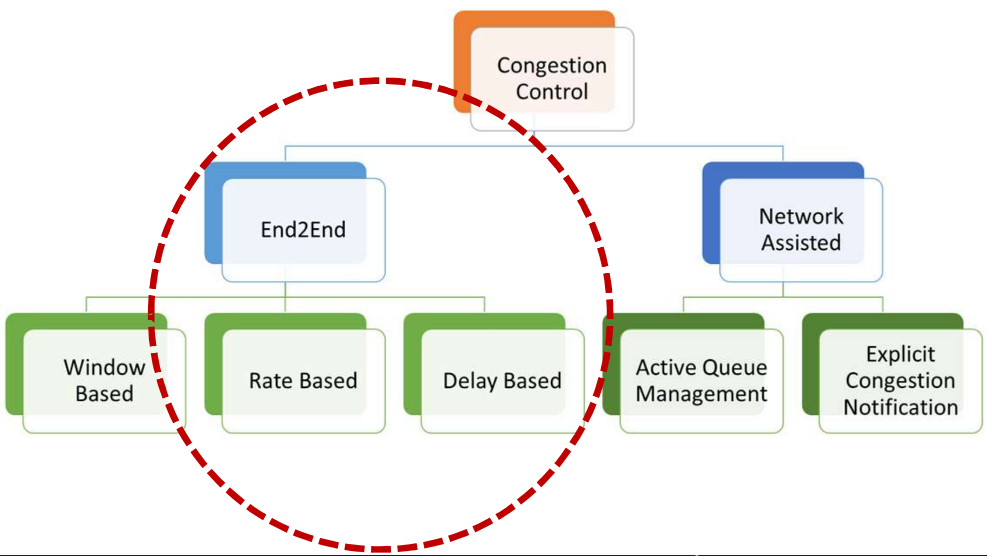
- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected.
  - congestion state of bottleneck link ***probably (?)*** hasn't changed much.
  - after cutting window in half on loss, initially ramp to to W_{max}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





- K – a tunable point in time when TCP window size will reach $W_{max.}$
- increase W as a function of the *cube* of the distance between current time & K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K

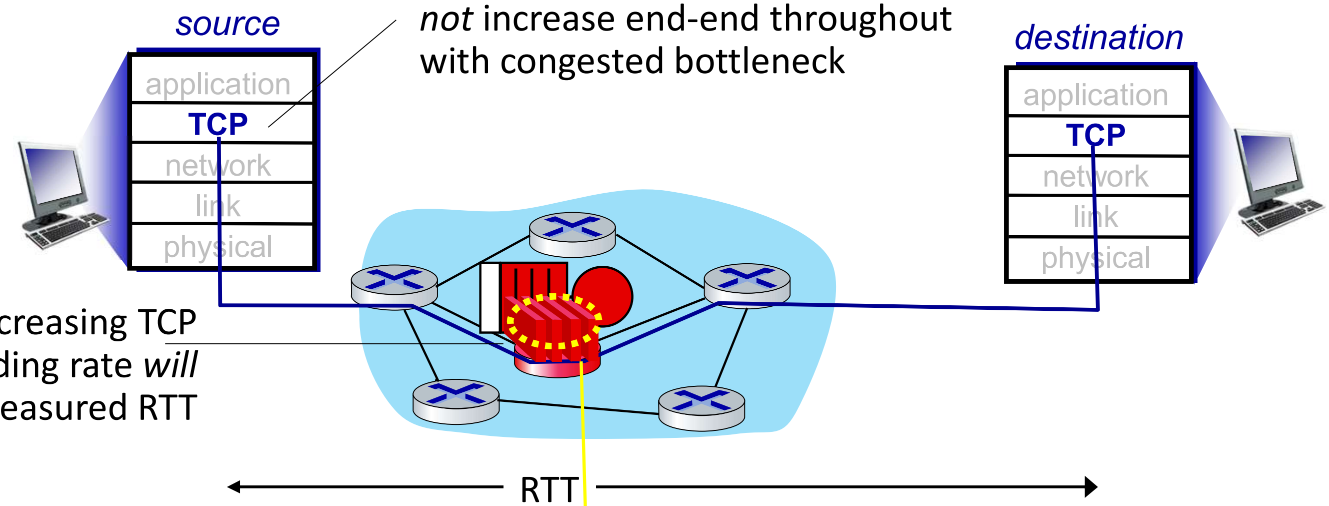
Delay/Rate based Congestion Control



TCP AND THE CONGESTED "BOTTLENECK LINK"

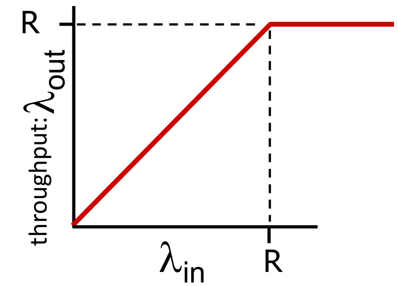
- understanding congestion: useful to focus on congested bottleneck link

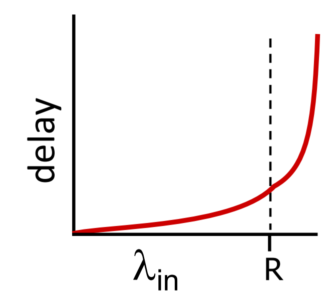
insight: increasing TCP sending rate will *not* increase end-end throughout with congested bottleneck



insight: increasing TCP sending rate *will* increase measured RTT

bottleneck link (almost always busy);
packet queue almost never empty, sometimes overflows packet (loss)

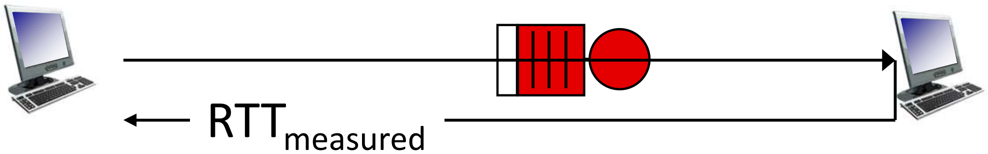




Goal: *"keep the end-end pipe just full, but not fuller"*

Rate-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



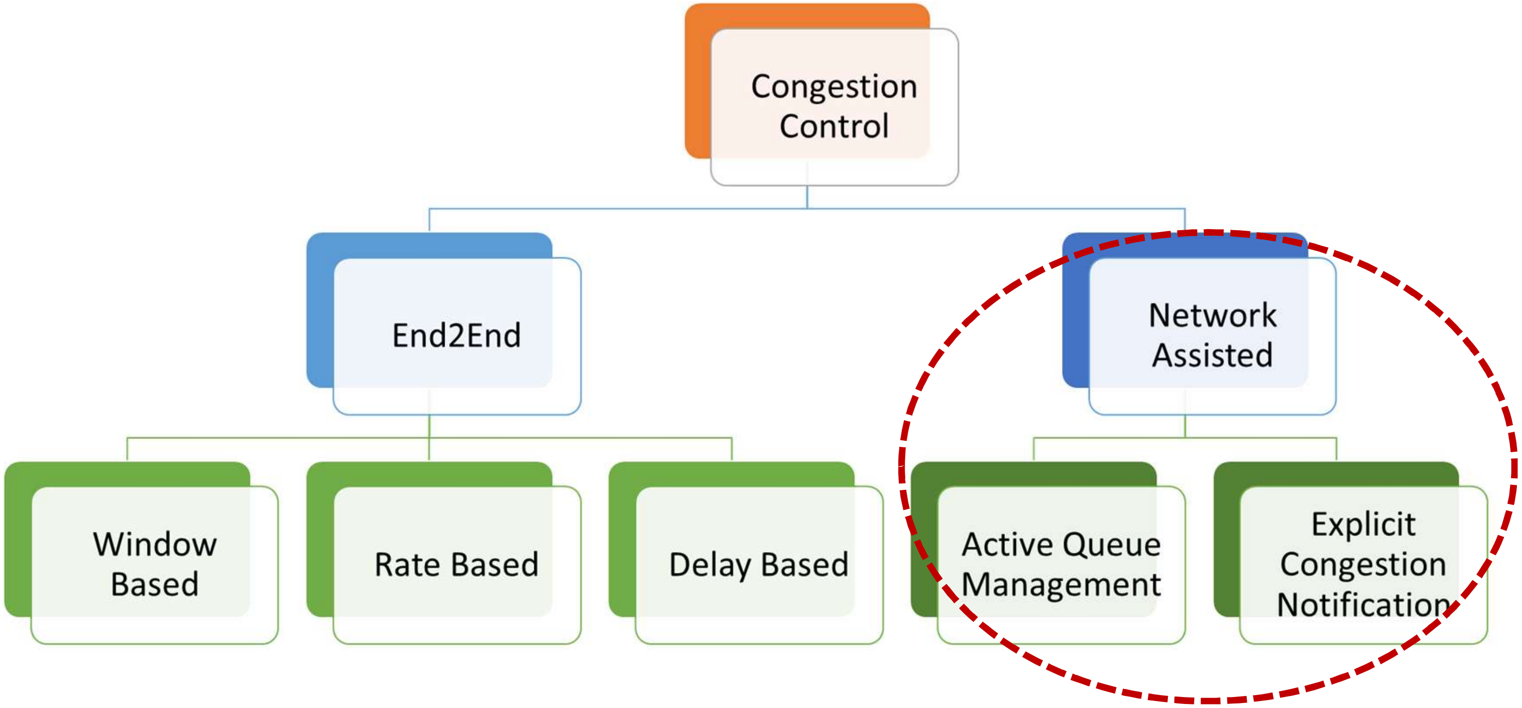
$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

- congestion control without inducing/forcing loss → Maximizes throughput
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network.
  - RTT_{min} : minimum observed RTT (i.e. RTT for 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 */

NETWORK-ASSISTANCE FOR CONGESTION CONTROL



- Split the responsibility of congestion control between end hosts and routers.

1. Router monitors congestion & explicitly notifies end-hosts when congestion is about to occur.
2. In response, the source adjusts transmit rate upon such notifications.

Random Early Drop (RED)

- Each router monitors its queue length.
- IF the ***queue length*** > ***threshold***, THEN:
  - Drop packets (implicit notification) to indicate network congestion.

Or

  - Explicitly notify source by marking "Explicit Congestion Notification" bit.

- Note that packets are dropped much earlier than usual -- before buffer resources are exhausted completely -- *so drops are fewer*
  - Bursty drops are also reduced.

Details of RED

- The principle is to drop the packet with some "**drop probability**" when the queue length exceeds a certain "***drop level***".

Q1: Should we use the Sample Queue Length or the average queue length?

- Instead of a sample queue length, average queue length (more accurately captures notion of congestion) is considered.

*AvgLen = (1-weight)*AvgLen + weight * SampleLen*

Q2: How to effect the packet drops?

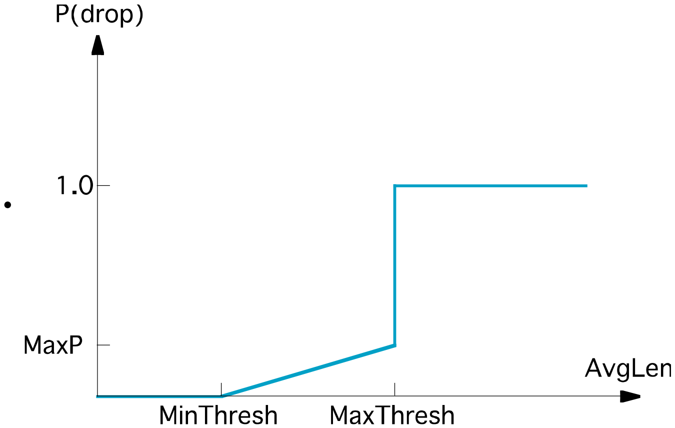
- Two thresholds : **MinThreshold** and **MaxThreshold**.

  If AvgLen <= MinThreshold queue the packet

  If AvgLen >= MaxThreshold drop the arriving packet.

  If MinThreshold <= AvgLen <= MaxThreshold, then,

  calculate a ***drop probabilty P*** and drop the arriving packet with the probability P.

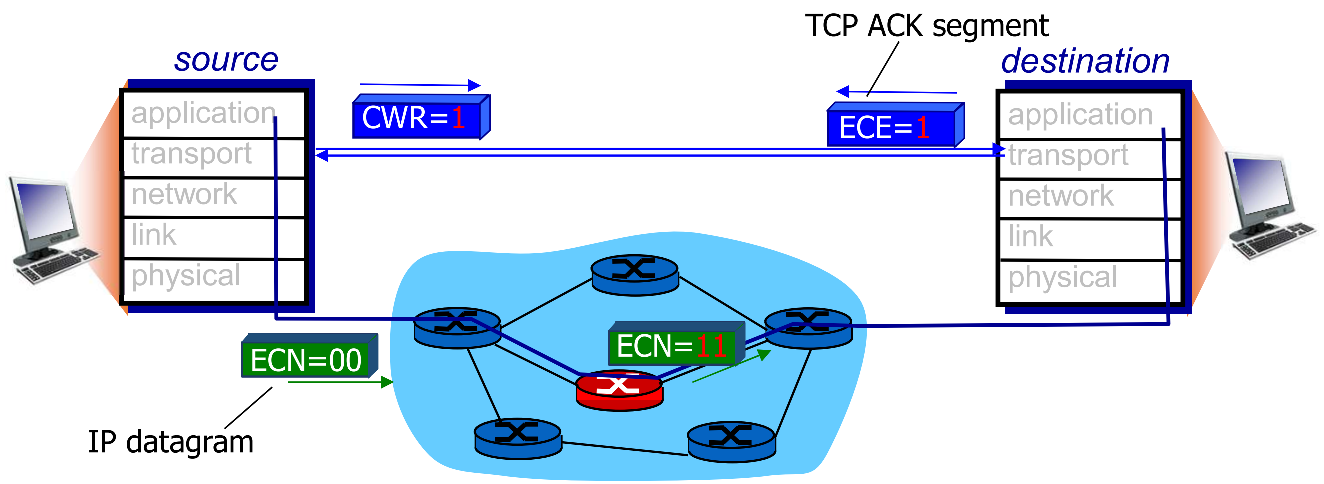


Reading more on RED: http://dl.acm.org/citation.cfm?id=169935

EXPLICIT CONGESTION NOTIFICATION (ECN)

Q3: Can we do better than just dropping the packets?

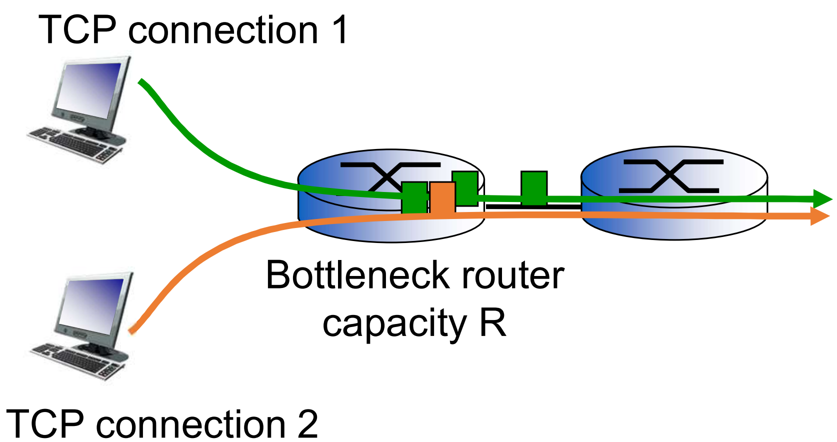
- Instead of packet drop, use the same packets to carry and notify the indication of congestion.
- two bits in IP header (ToS field) marked *by network router* to indicate congestion.
- receiver (seeing congestion indication in IP datagram)) sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion.

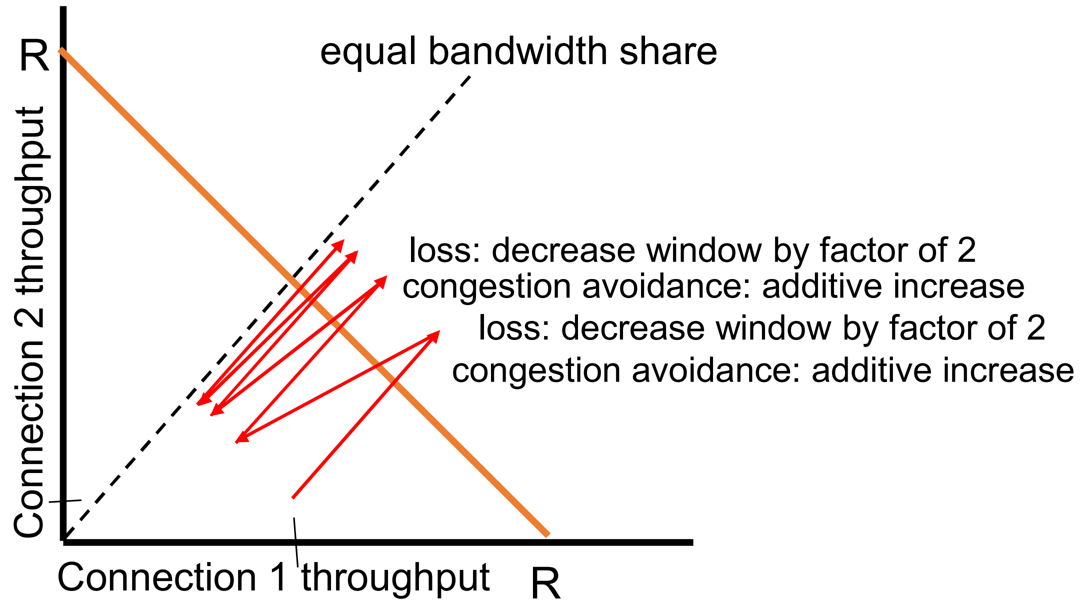


Reading more on ECN: https://dl.acm.org/doi/pdf/10.17487/RFC2481
https://www.cse.wustl.edu/~jain/papers/ftp/cr1.pdf

TCP Fairness

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





two competing sessions:

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

FAIRNESS (MORE)

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

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 SACK

A Fundamental Advancement in TCP!
<Not covered in the text books!>

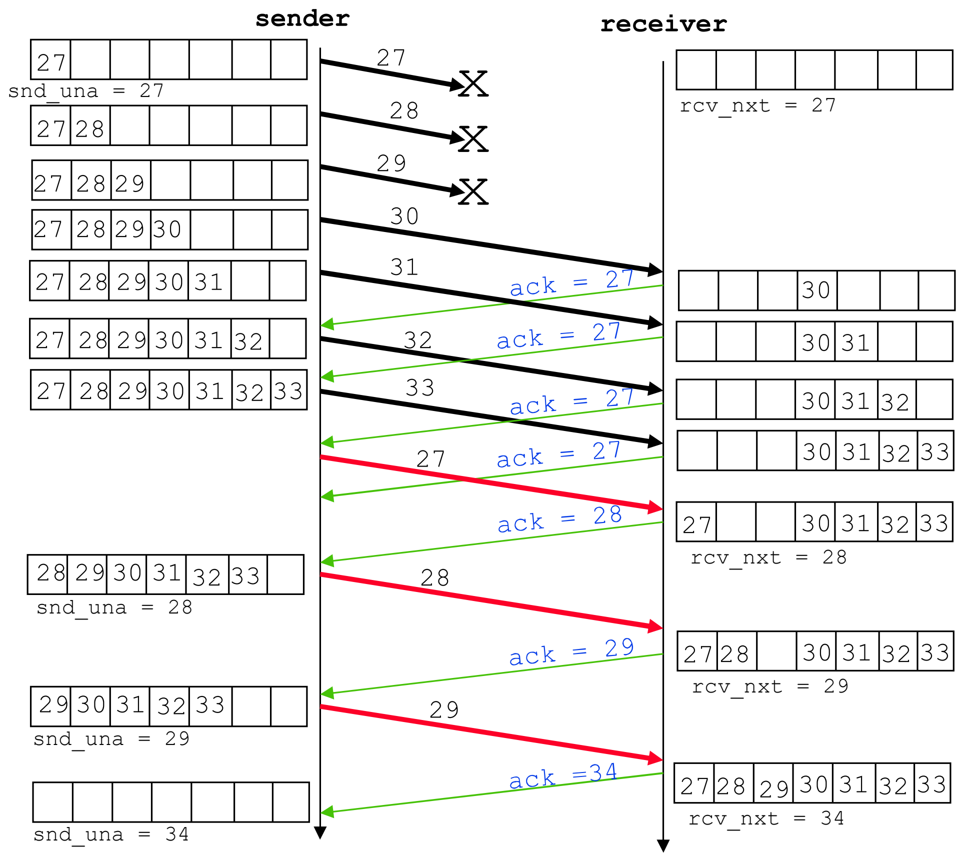
TCP SACK + Selective Repeat Retransmission Policy.

https://tools.ietf.org/html/rfc2018

SELECTIVE ACKNOWLEDGEMENT (RFC 2018)

- SACK = Selective Acknowledgement ('New' TCP option)
- Reasons for SACK
  - TCP may perform poorly when multiple packets are lost
  - Data may be retransmitted unnecessarily by go-back-N (TCP Tahoe)
  - Or one missing segment retransmitted per round trip
- Receiver's part of SACK
  - Acknowledges non-contiguous blocks of data with SACK blocks
  - SACK blocks: sequence number left/right edges of isolated data
  - May drop data that is "SACKed"
- Sender's part of SACK
  - SACKed blocks marked in retransmission buffer
  - Marked segments not retransmitted, "holes" retransmitted
  - Blocks unmarked after retransmission timeout

CUMULATIVE ACK EXAMPLE

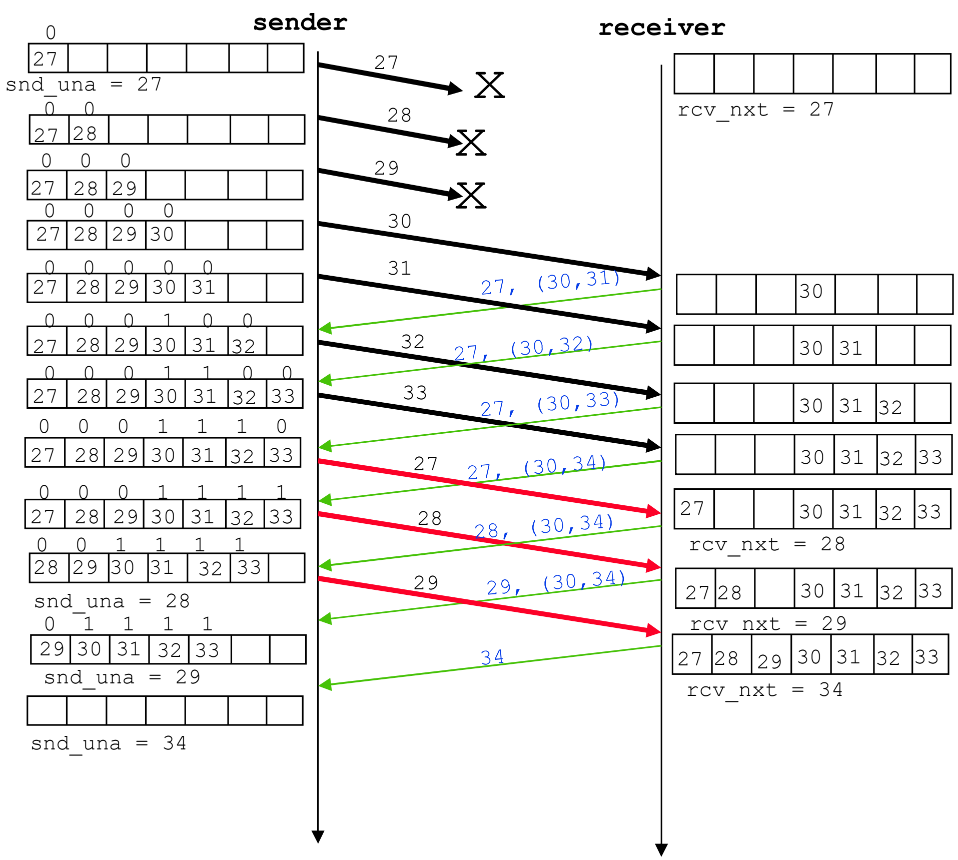


- Window size = 7
- Multiple packets lost in one RTT

- retransmit pkt 27 on receiving 2(3 total) dup acks.
- Pkt. 28 can be retransmitted only after receiving ack of pkt 27
  - 1 RTT wait

- receiver waits longer to deliver packets to application in sequence

SACK Example



- Still retransmit pkt 27 on receiving 2 (3 total) dup (s)acks.
- Pkt. 28 can be retransmitted right away, knowing that 3 packets have been received subsequently
  - only wait an inter-packet transmit time,
- ε reduction in latency

KEY TCP EXTENSIONS

- For High Bandwidth-Delay Connections
  - Accurate Round-Trip Time estimation
  - Overcome limit on window size
  - Dealing with Packet loss

- Use TCP Options
  - Timestamp Extension
  - Protecting from Sequence number wraparound (32 bits ~36 Gbits)
  - Extensions for Larger Windows (Window Scaling)