KHOA CÔNG NGHỆ THÔNG TIN-ĐHKHTN
CSC11004 - MẠNG MÁY TÍNH NÂNG CAO

TCP CONGESTION CONTROL

Lê Ngọc Sơn

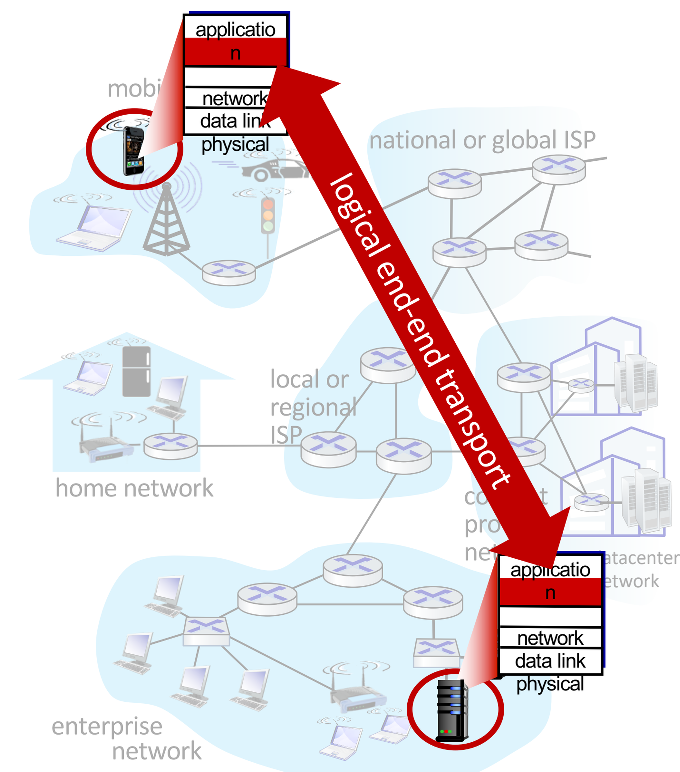
TPHCM, 9-2021

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

fit@hcmus

Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Agenda

- ☐Transport-layer services
- ☐UDP and TCP
- ☐TCP Flow control
- ☐Principles of congestion control
- ☐TCP congestion control

Agenda

- ☐Transport-layer services
- ☐UDP and TCP
- ☐TCP Flow control
- ☐Principles of congestion control
- ☐TCP congestion control

Agenda

- ☐Transport-layer services
- ☐UDP and TCP
- ☐TCP Flow control
- ☐Principles of congestion control
- ☐TCP congestion control

UDP: User Datagram Protocol

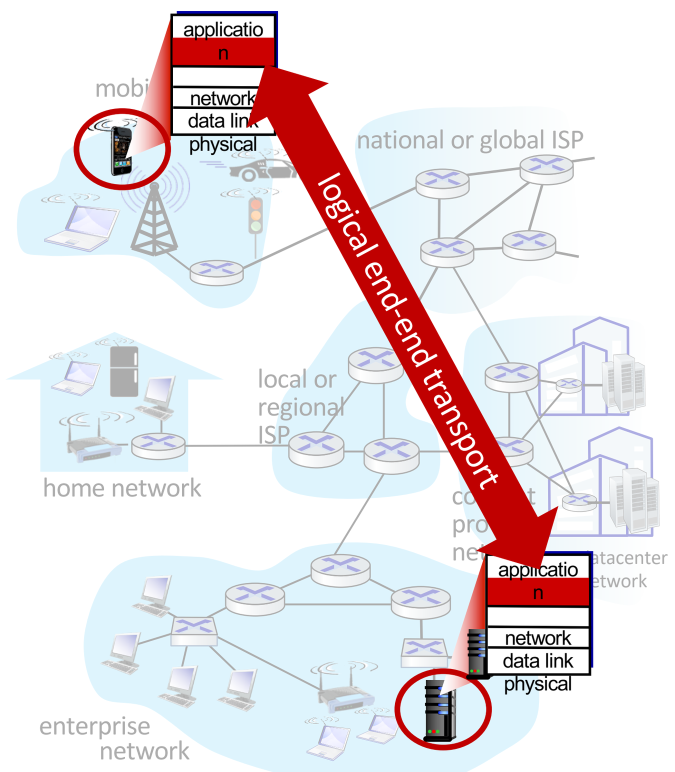
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

Transport services and protocols

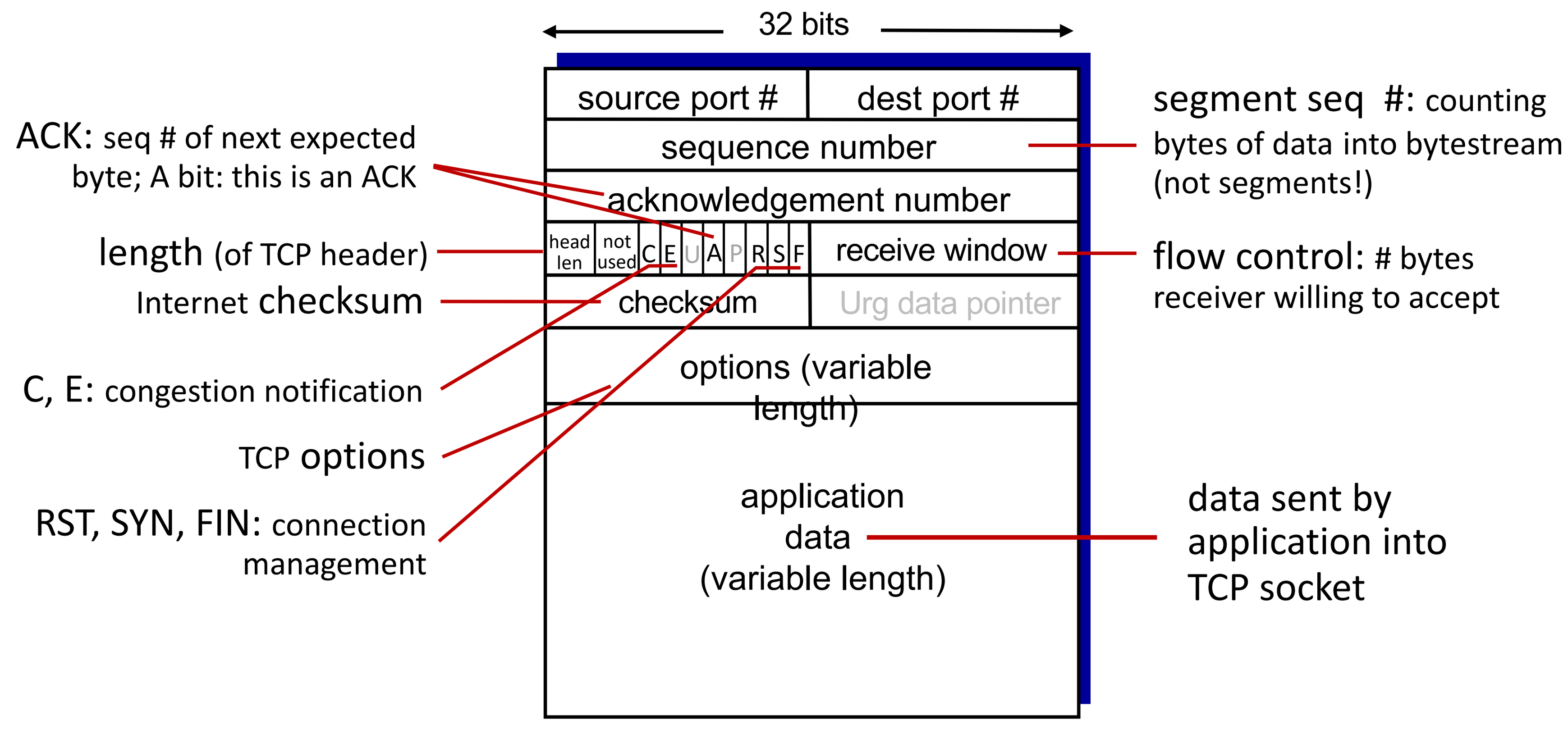
- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP sequence numbers, ACKs

Sequence numbers:

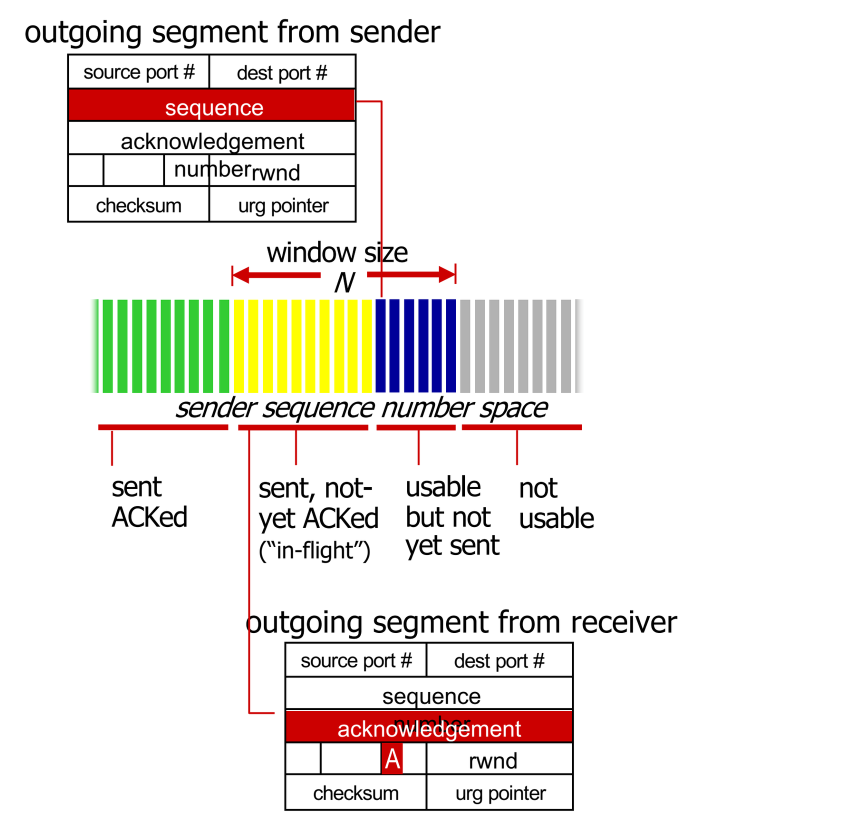
- byte stream "number" of first byte in segment's data

Acknowledgements:

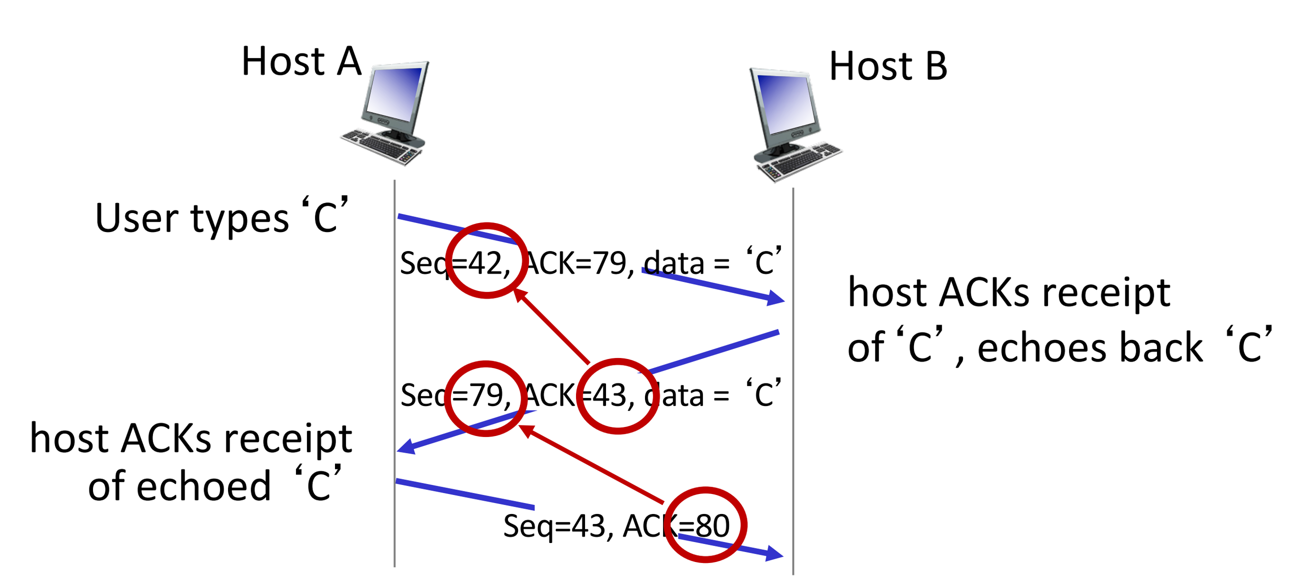
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor



TCP sequence numbers, ACKs



simple telnet scenario

TCP round trip time, 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

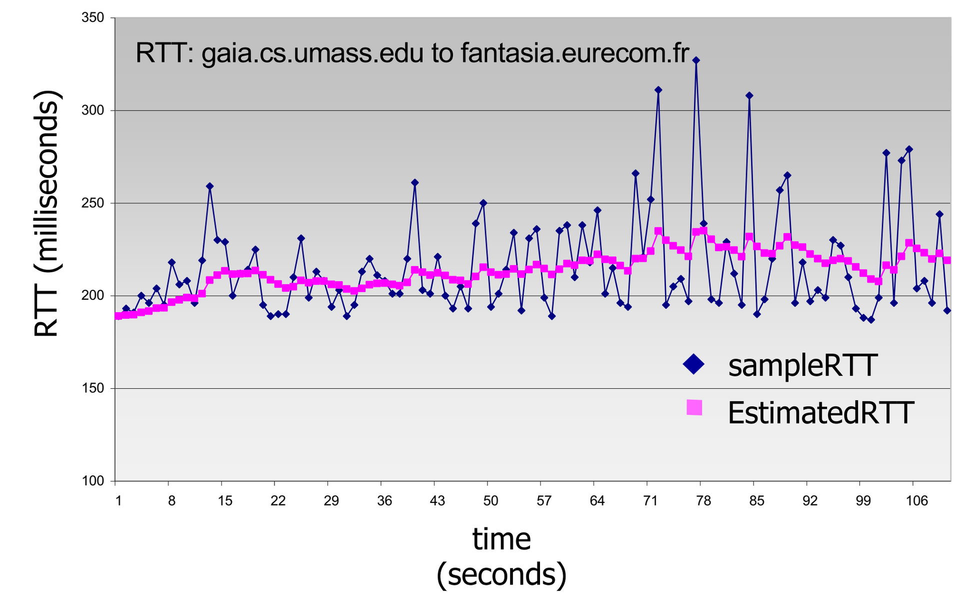
Q: how to estimate RTT?

- `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`

TCP round trip time, timeout

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT:** want a larger safety margin

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

estimated RTT "safety margin"

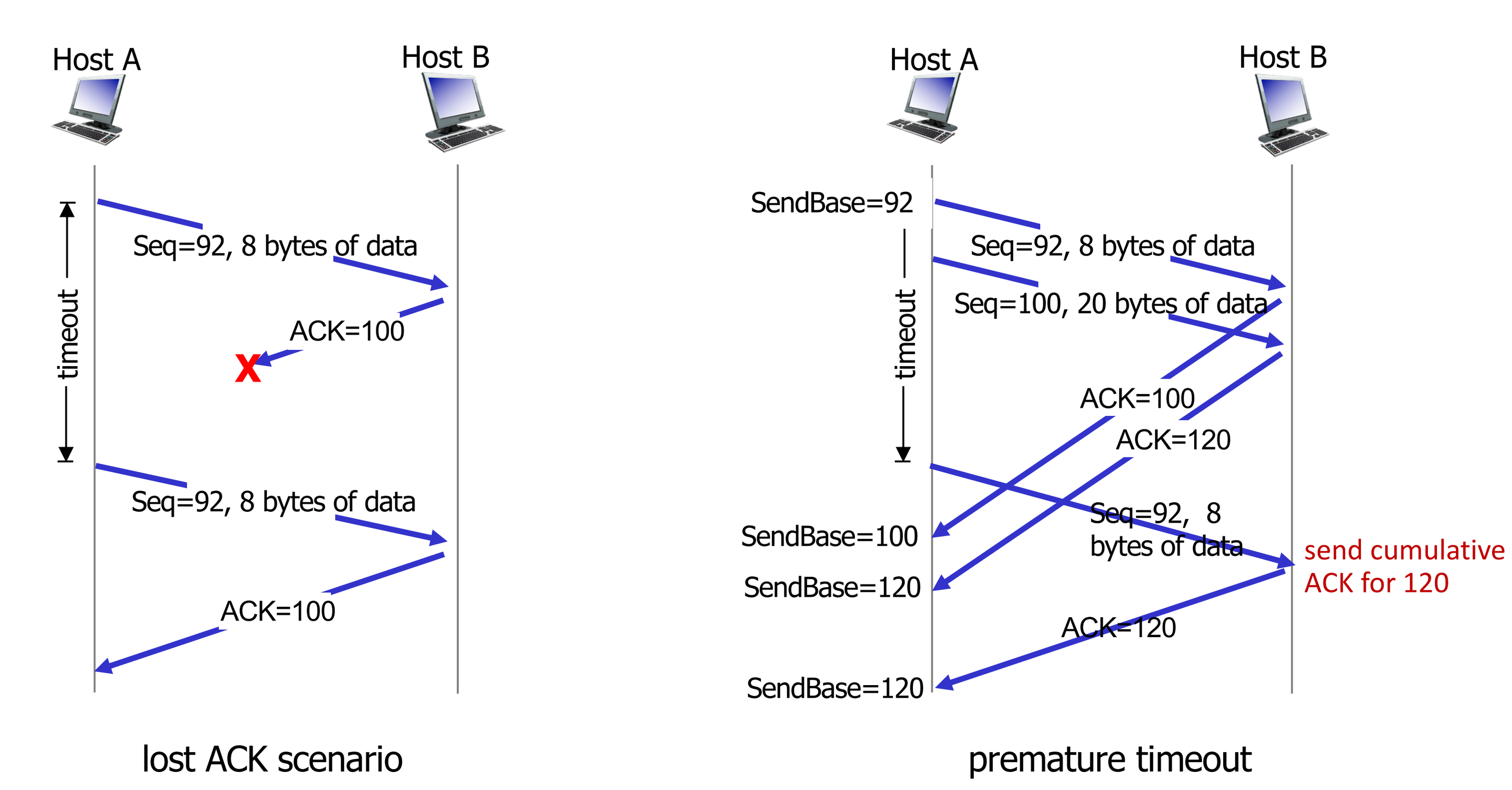
- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

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

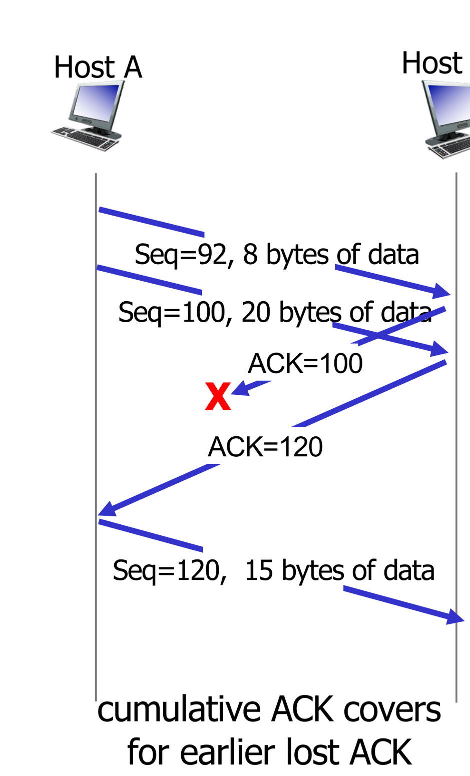
(typically, β = 0.25)

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

TCP: retransmission scenarios



TCP: retransmission scenarios



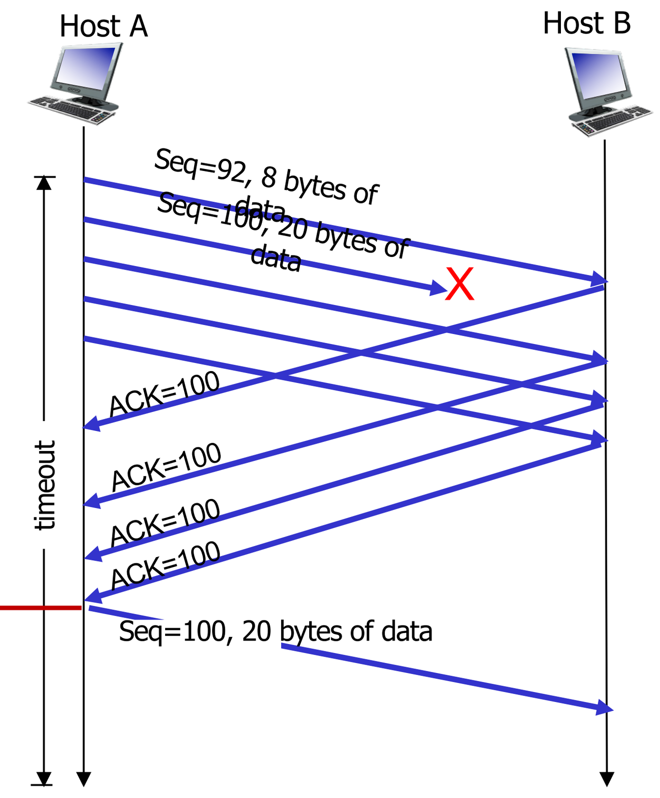
TCP fast retransmit

TCP fast retransmit

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

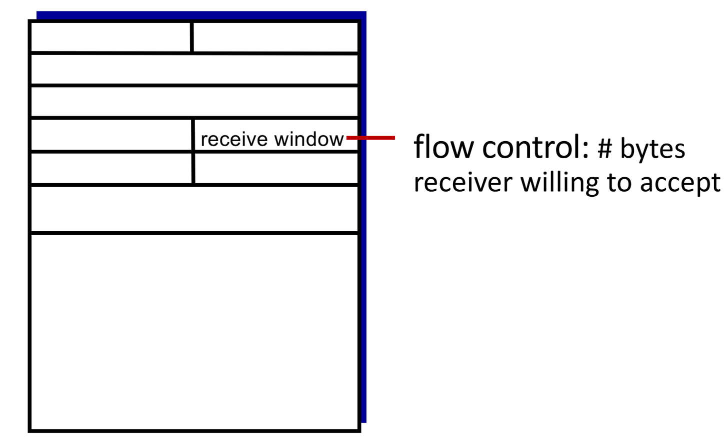
- likely that unACKed segment lost, so don't wait for timeout

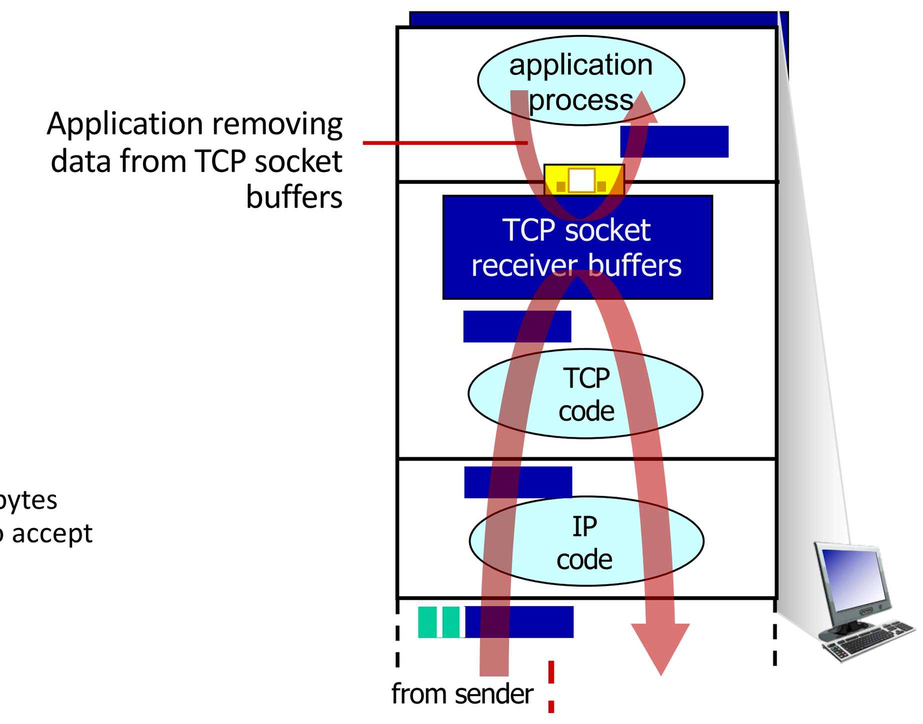
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



TCP flow control

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





receiver protocol stack

Agenda

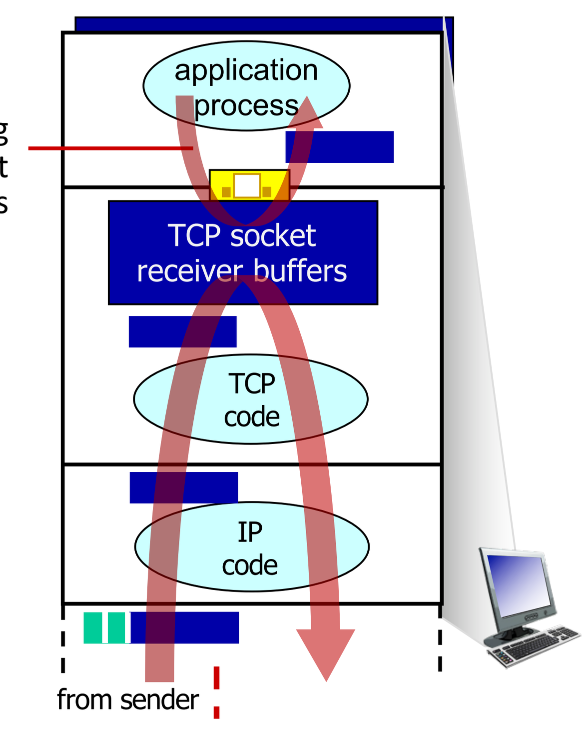
- Transport-layer services
- UDP and TCP
- TCP Flow control
- Principles of congestion control
- TCP congestion control

TCP flow control

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

flow control

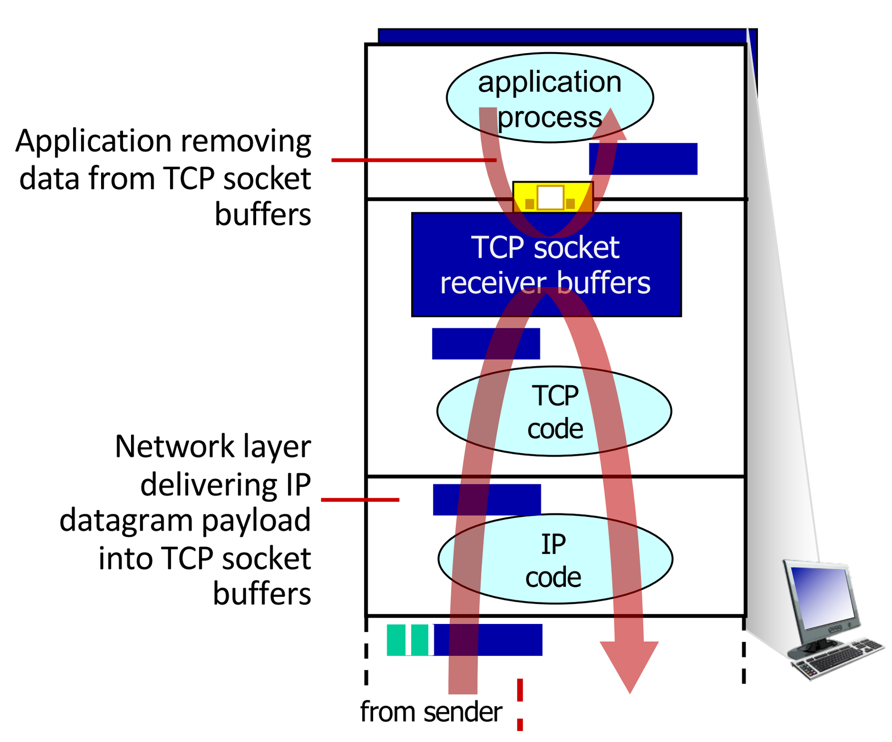
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



receiver protocol stack

TCP flow control

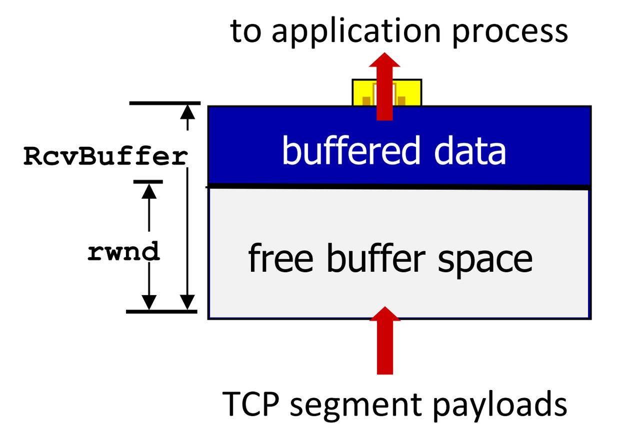
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

TCP flow control

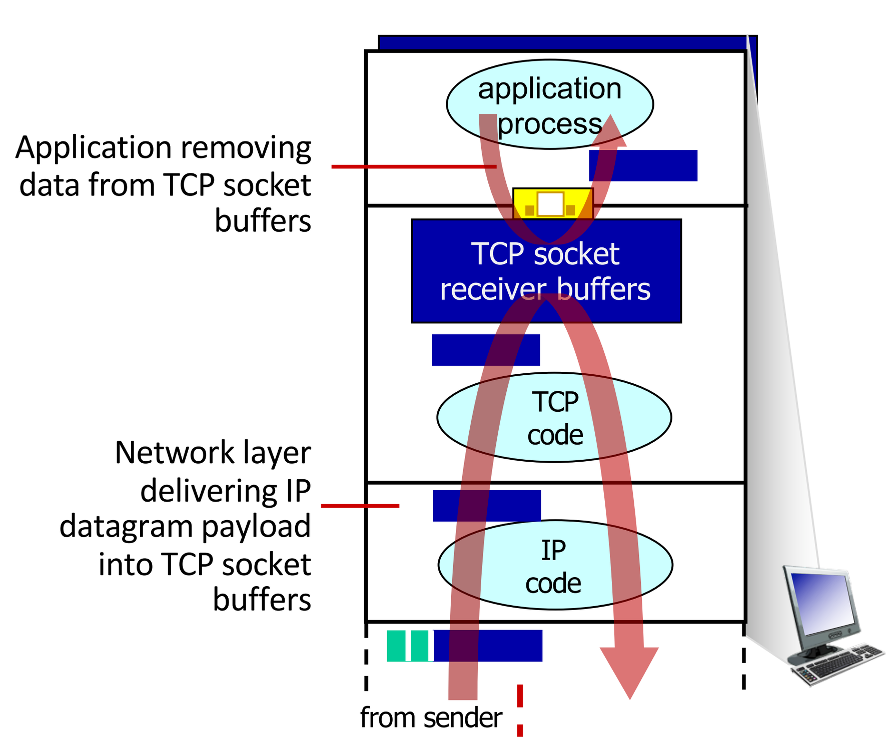
- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed ("in-flight") data to received **rwnd**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

TCP flow control

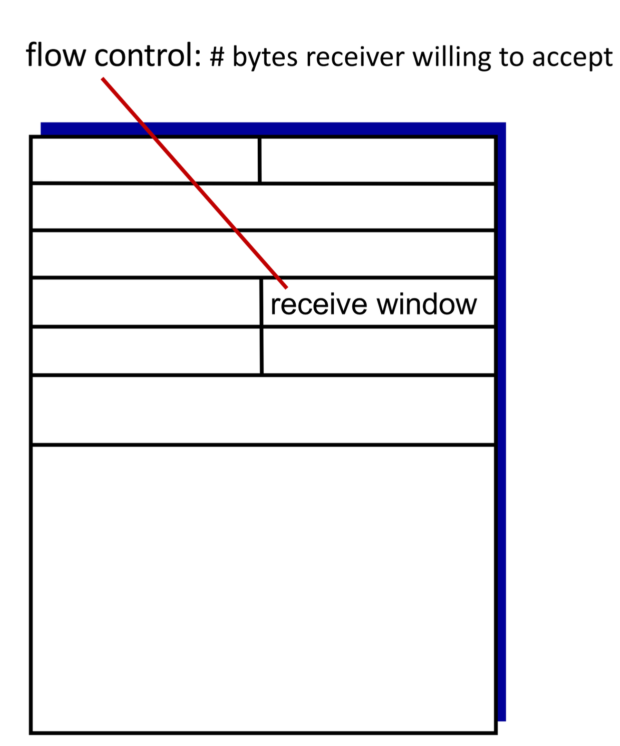
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

TCP flow control

- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unACKed ("in-flight") data to received **rwnd**
- guarantees receive buffer will not overflow



TCP segment format

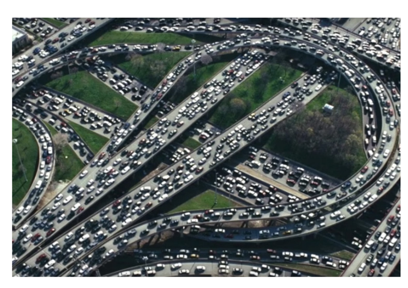
Agenda

- ☐Transport-layer services
- ☐UDP and TCP
- ☐TCP flow control
- ☐Principles of congestion control
- ☐TCP congestion control

Principles of congestion control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!



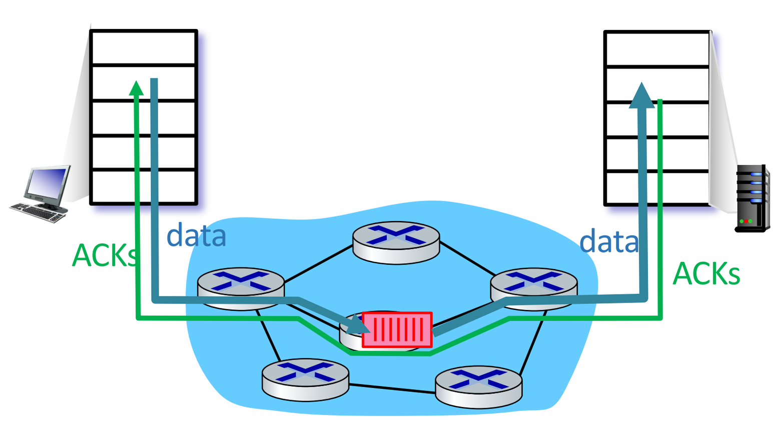
congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

Approaches towards congestion control

End-end congestion control:

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

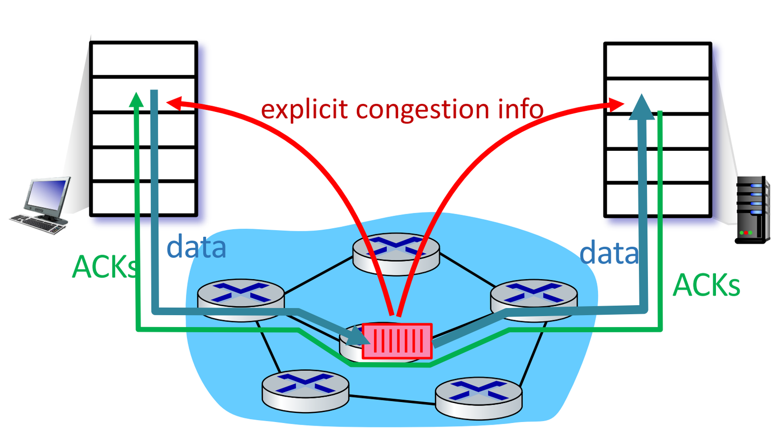


Approaches towards congestion control

Network-assisted congestion control:

- routers provide *direct* feedback to sending/receiving hosts with flows passing through congested router
- may indicate congestion level or explicitly set sending rate

☐TCP ECN, ATM, DECbit protocols



Agenda

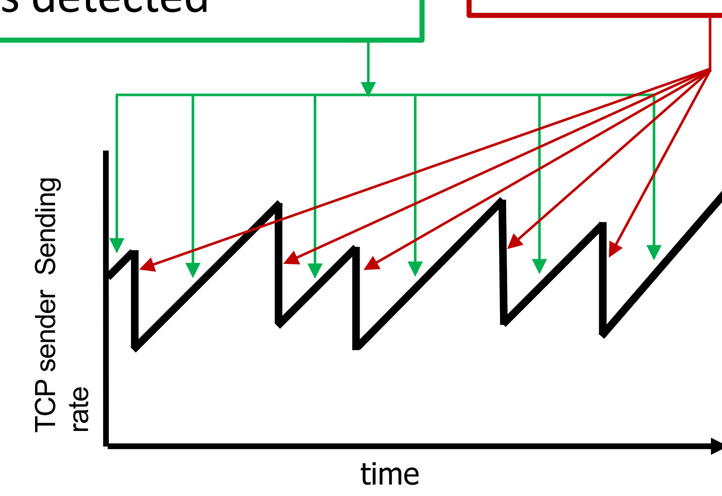
- ☐Transport-layer services
- ☐UDP and TCP
- ☐Principles of congestion control
- ☐TCP congestion control

TCP congestion control: AIMD

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

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

Multiplicative Decrease: cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

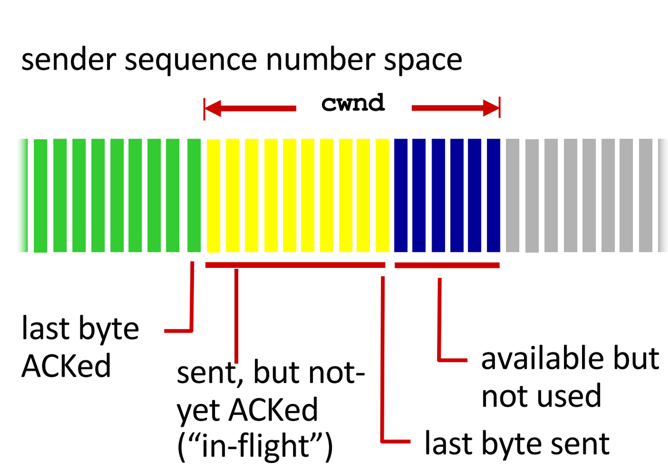
Multiplicative decrease detail: sending rate is

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

Why AIMD?

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

TCP congestion control: details



TCP sending behavior:

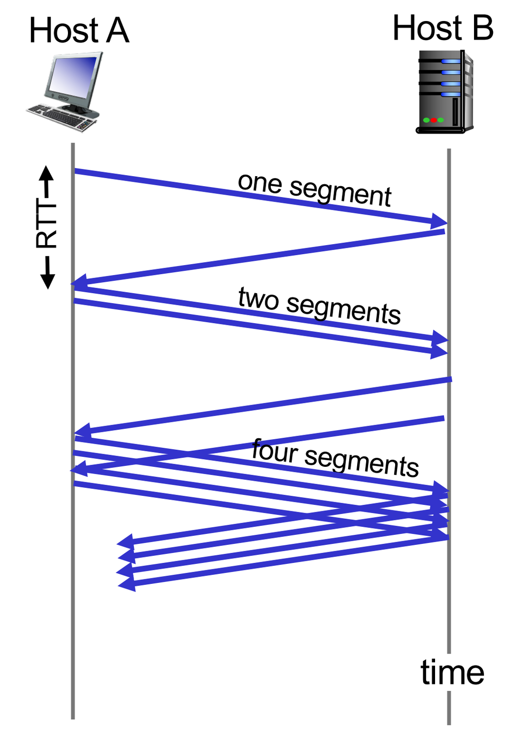
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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

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

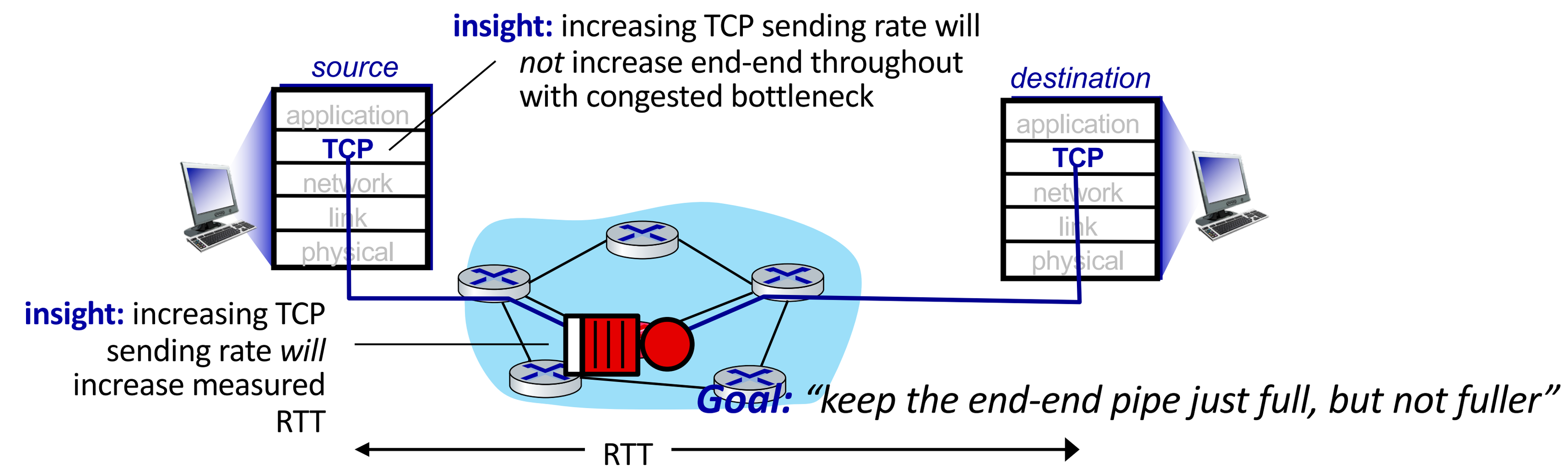
TCP slow start

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



TCP and the congested "bottleneck link"

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



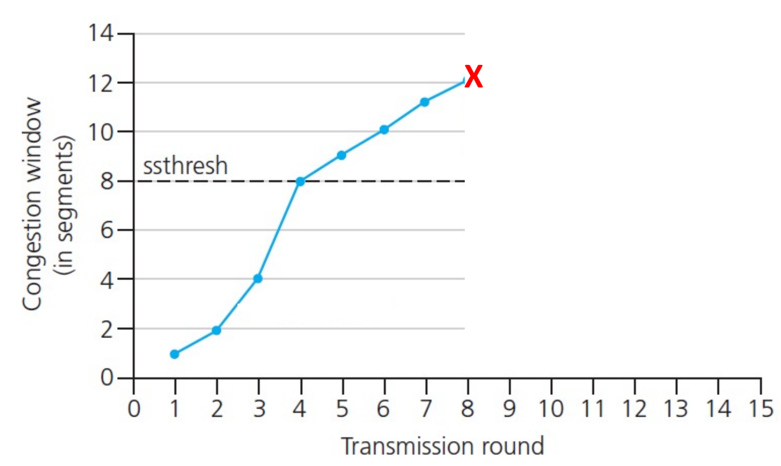
TCP: from slow start to congestion avoidanc

Q: when should the exponential increase switch to linear?

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

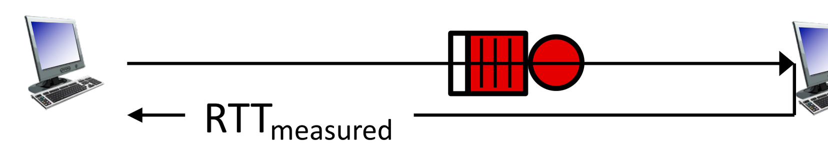
Implementation:

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



Delay-based TCP congestion control

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



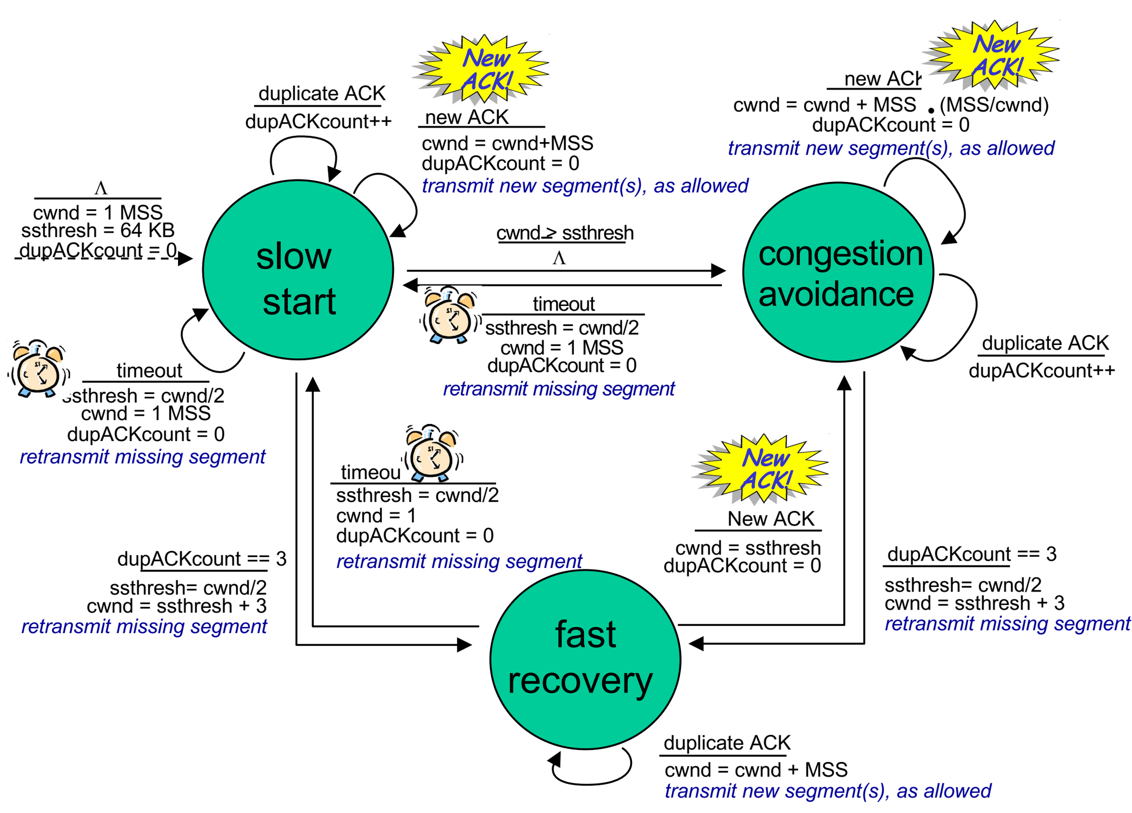
$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Delay-based approach:

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

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

Summary: TCP congestion control

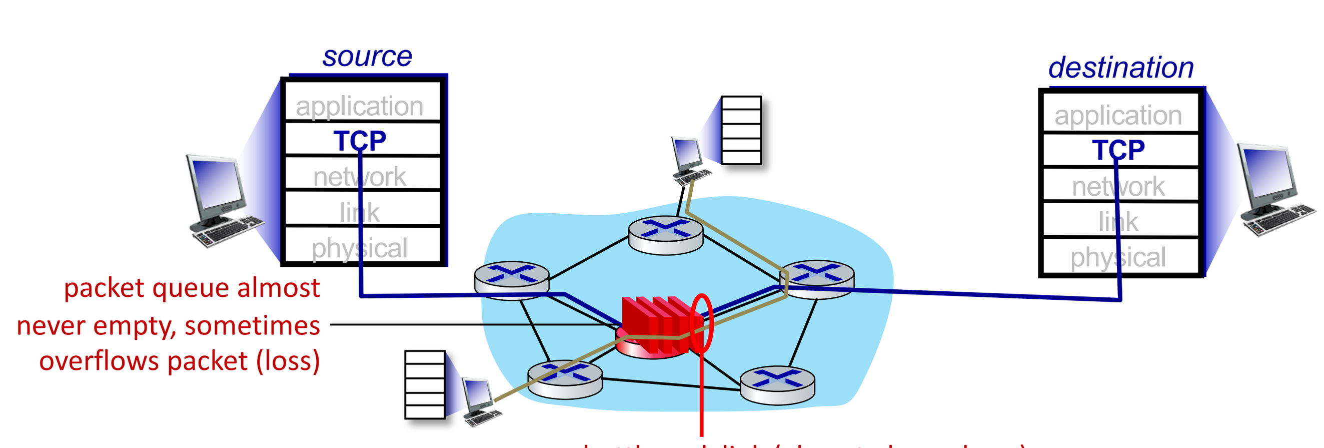


Delay-based TCP congestion control

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

TCP and the congested "bottleneck link"

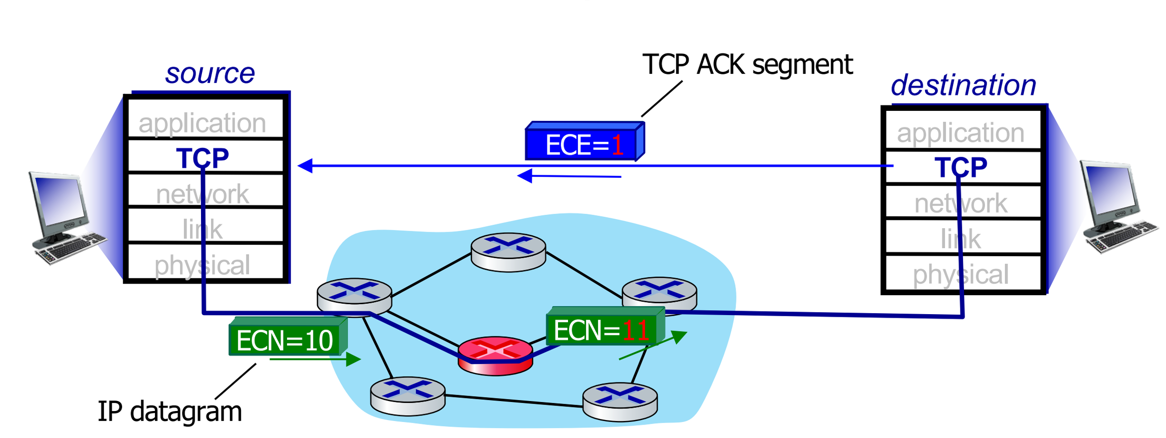
- TCP increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*



Explicit congestion notification (ECN)

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

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



fit@hcmus

TCP Congestion Control Today

TCP BIC	TCP CUBIC
• optimize long fat networks • Binary Increase Congestion control • used by default in Linux kernels 2.6.8 through 2.6.18	• Improvement of TCP BIC • used by default in Linux kernels between versions 2.6.19 and 3.2. • MacOS adopted CUBIC by at least the OS X Yosemite release in 2014 • Microsoft adopted it by default in Windows 10.1709 Fall Creators Update (2017), and Windows Server 2016 1709 update

fit@hcmus

Other variants

Variant	Feedback	Required changes	Benefits
(New) Reno	Loss	—	—
Vegas	Delay	Sender	Less loss
High Speed	Loss	Sender	High bandwidth
BIC	Loss	Sender	High bandwidth
CUBIC	Loss	Sender	High bandwidth
C2TCP[9][10]	Loss/Delay	Sender	Ultra-low latency and high bandwidth
NATCP[11]	Multi-bit signal	Sender	Near Optimal Performance
Elastic-TCP	Loss/Delay	Sender	High bandwidth/short & long-distance
Agile-TCP	Loss	Sender	High bandwidth/short-distance
H-TCP	Loss	Sender	High bandwidth
FAST	Delay	Sender	High bandwidth
Compound TCP	Loss/Delay	Sender	High bandwidth
Westwood	Loss/Delay	Sender	L
Jersey	Loss/Delay	Sender	L
BBR[12]	Delay	Sender	BLVC, Bufferbloat
CLAMP	Multi-bit signal	Receiver, Router	V
TFRC	Loss	Sender, Receiver	No Retransmission
XCP	Multi-bit signal	Sender, Receiver, Router	BLFC
VCP	2-bit signal	Sender, Receiver, Router	BLF
MaxNet	Multi-bit signal	Sender, Receiver, Router	BLFSC
JetMax	Multi-bit signal	Sender, Receiver, Router	High bandwidth
RED	Loss	Router	Reduced delay
ECN	Single-bit signal	Sender, Receiver, Router	Reduced loss

fit@hcmus

Question?