Transport Layer and Data Center TCP

Hakim Weatherspoon
Assistant Professor, Dept of Computer Science

CS 5413: High Performance Systems and Networking

September 5, 2014

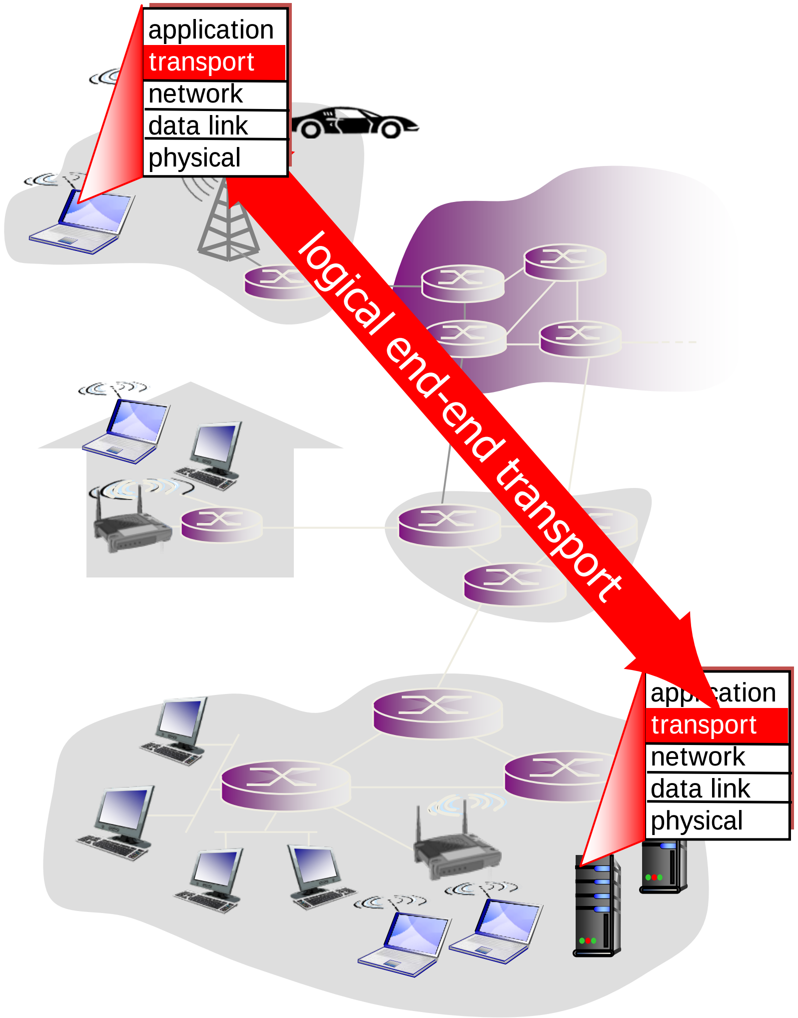
Slides used and adapted judiciously from Computer Networking, A Top-Down Approach

Goals for Today

- Transport Layer
  - Abstraction / services
  - Multiplexing/Demultiplexing
  - UDP: Connectionless Transport
  - TCP: Reliable Transport
    - Abstraction, Connection Management, Reliable Transport, Flow Control, timeouts
    - Congestion control
- Data Center TCP
  - Incast Problem

Transport Layer: Services/Protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport Layer: Services/Protocols

Transport vs Network Layer

- ***network layer:*** logical communication between hosts
- ***transport layer:*** logical communication between processes
  - relies on, enhances, network layer services

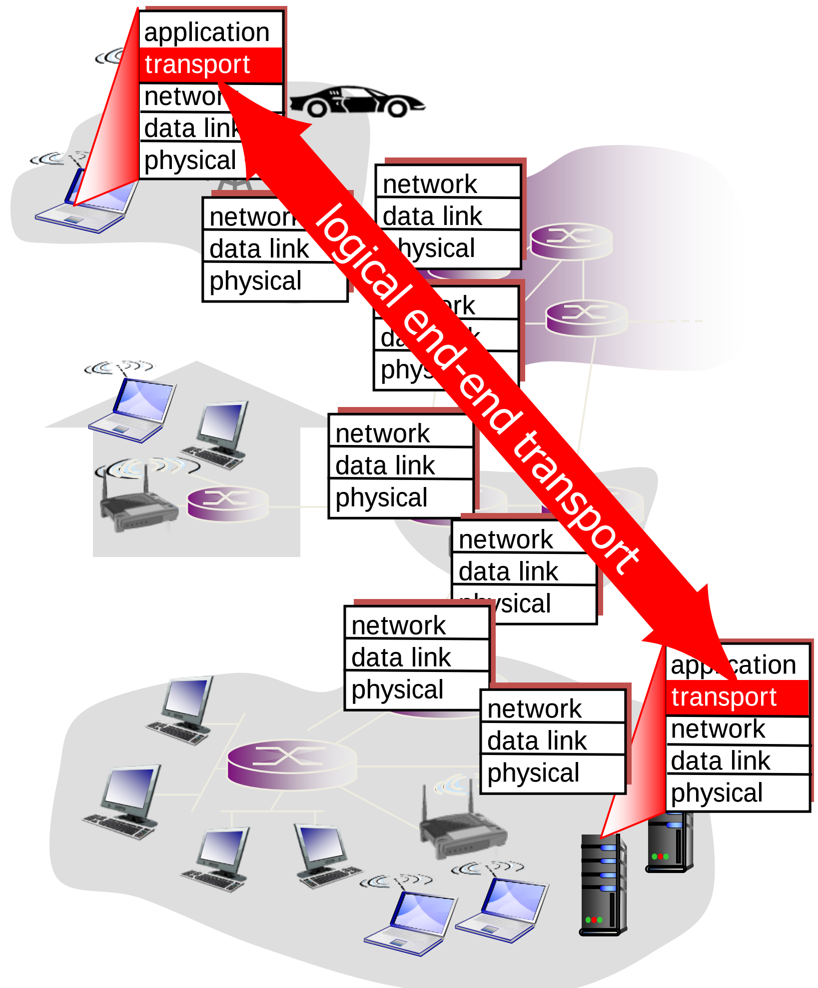
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Transport Layer: Services/Protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Transport Layer: Services/Protocols

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup,

Q: why bother? Why is there a UDP?

Goals for Today

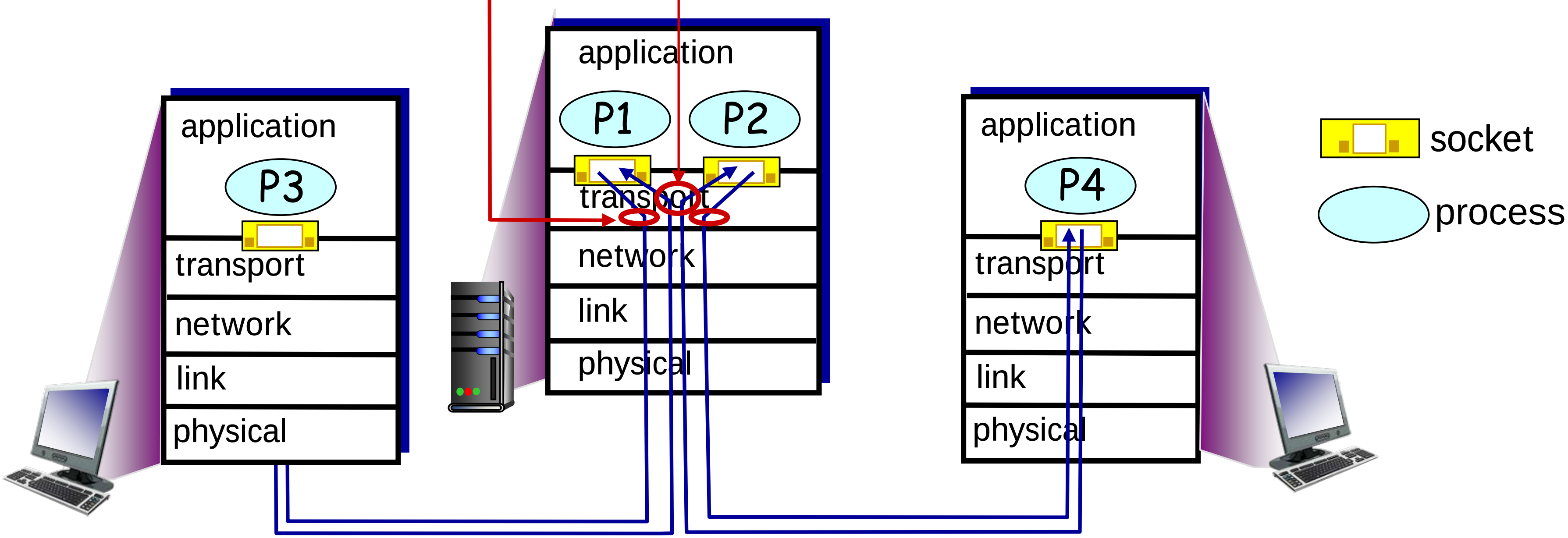
- Transport Layer
  - Abstraction / services
  - Multiplexing/Demultiplexing
  - UDP: Connectionless Transport
  - TCP: Reliable Transport
    - Abstraction, Connection Management, Reliable Transport, Flow Control, timeouts
    - Congestion control

- Data Center TCP
  - Incast Problem

Sockets: Multiplexing/Demultiplexing

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



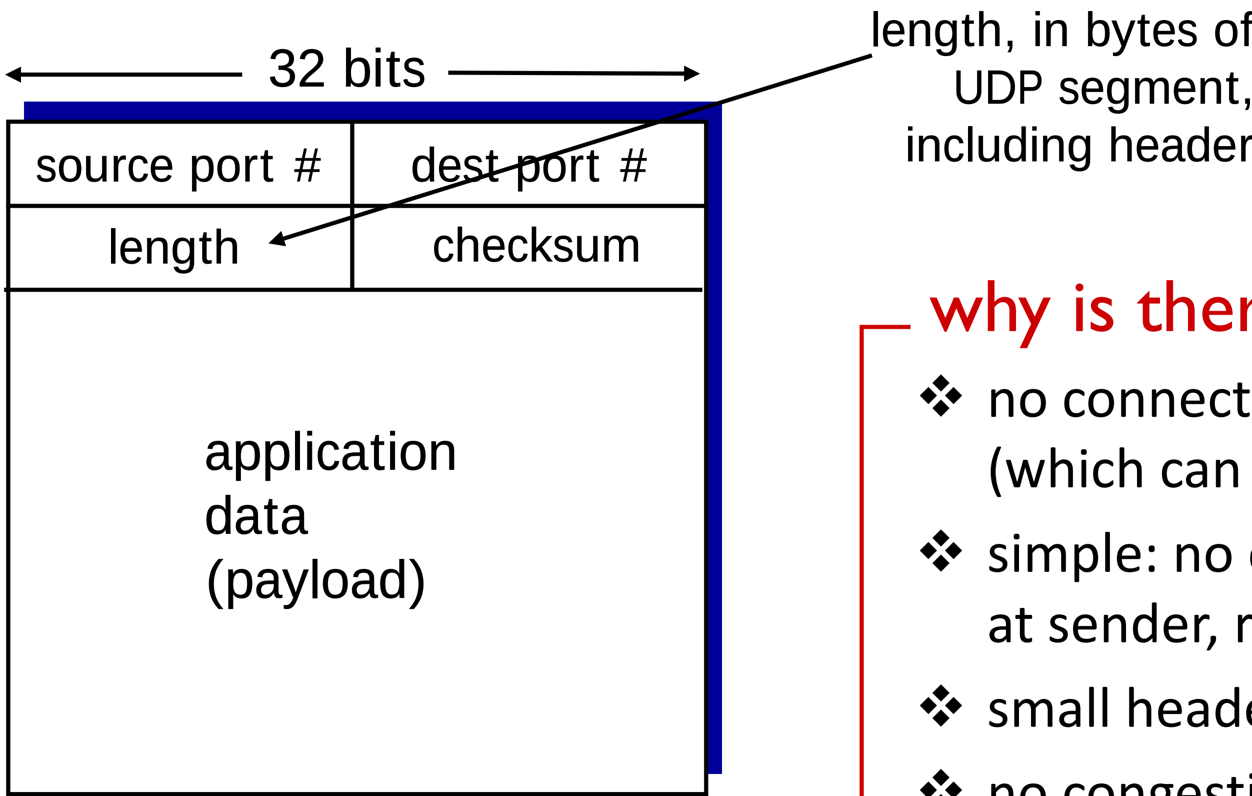
Goals for Today

- Transport Layer
  - Abstraction / services
  - Multiplexing/Demultiplexing
  - UDP: Connectionless Transport
  - TCP: Reliable Transport
    - Abstraction, Connection Management, Reliable Transport, Flow Control, timeouts
    - Congestion control

- Data Center TCP
  - Incast Problem

UDP: Connectionless Transport

UDP: Segment Header



UDP segment format

why is there a UDP?

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

UDP: Connectionless Transport

UDP: Checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

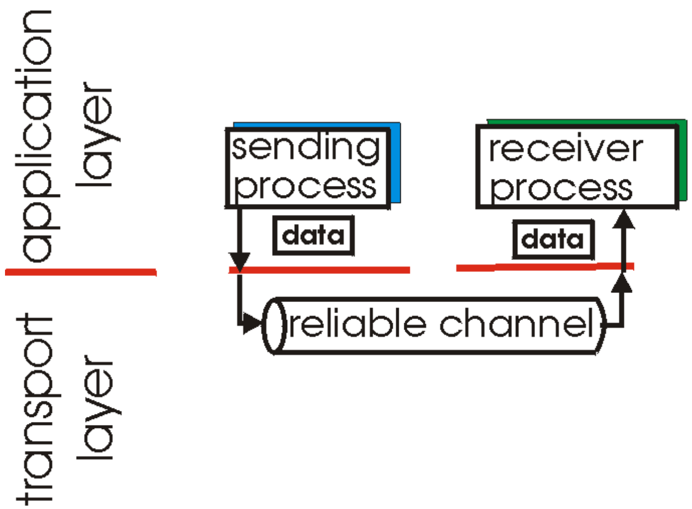
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ....

Goals for Today

- Transport Layer
  - Abstraction / services
  - Multiplexing/Demultiplexing
  - UDP: Connectionless Transport
  - TCP: Reliable Transport
    - Abstraction, Connection Management, Reliable Transport, Flow Control, timeouts
    - Congestion control
- Data Center TCP
  - Incast Problem

Principles of Reliable Transport

- important in application, transport, link layers
  - top-10 list of important networking topics!

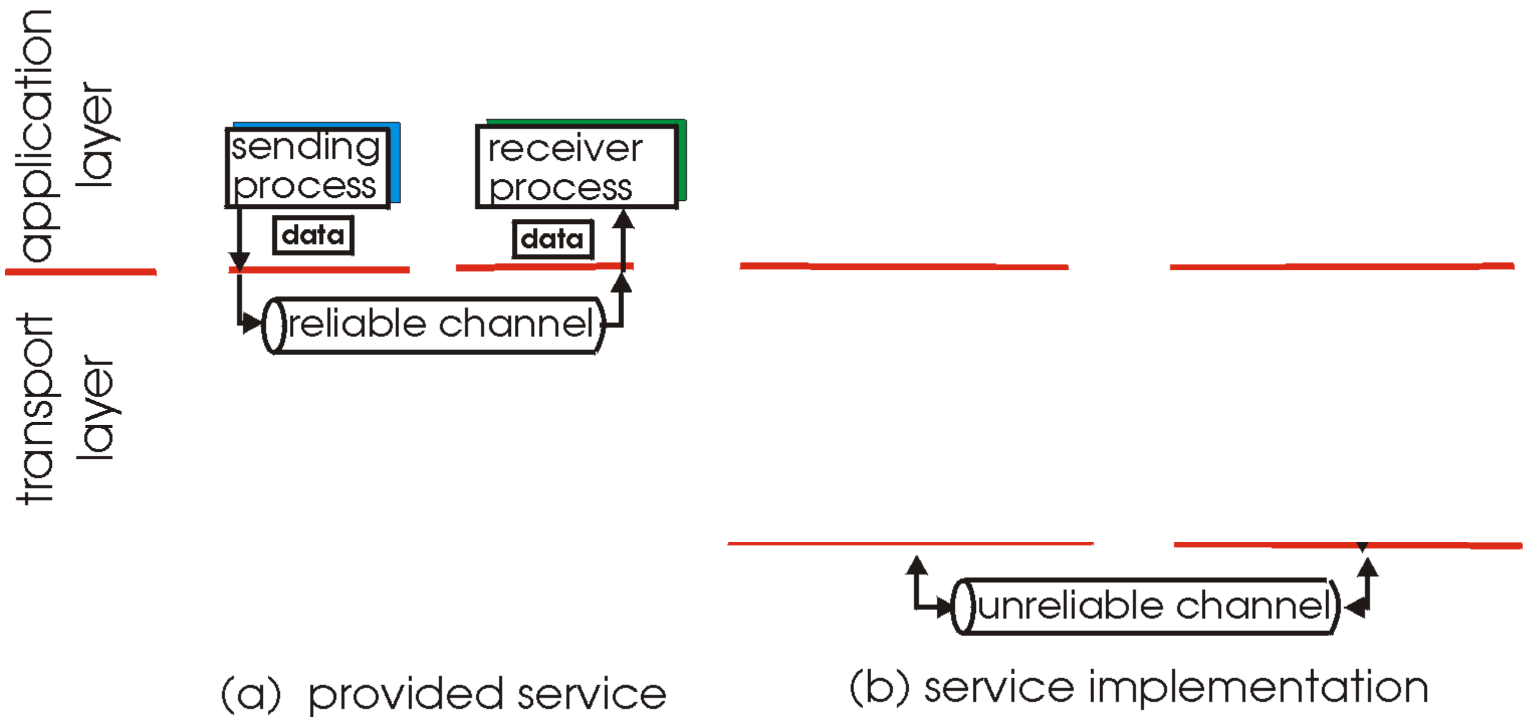


(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable Transport

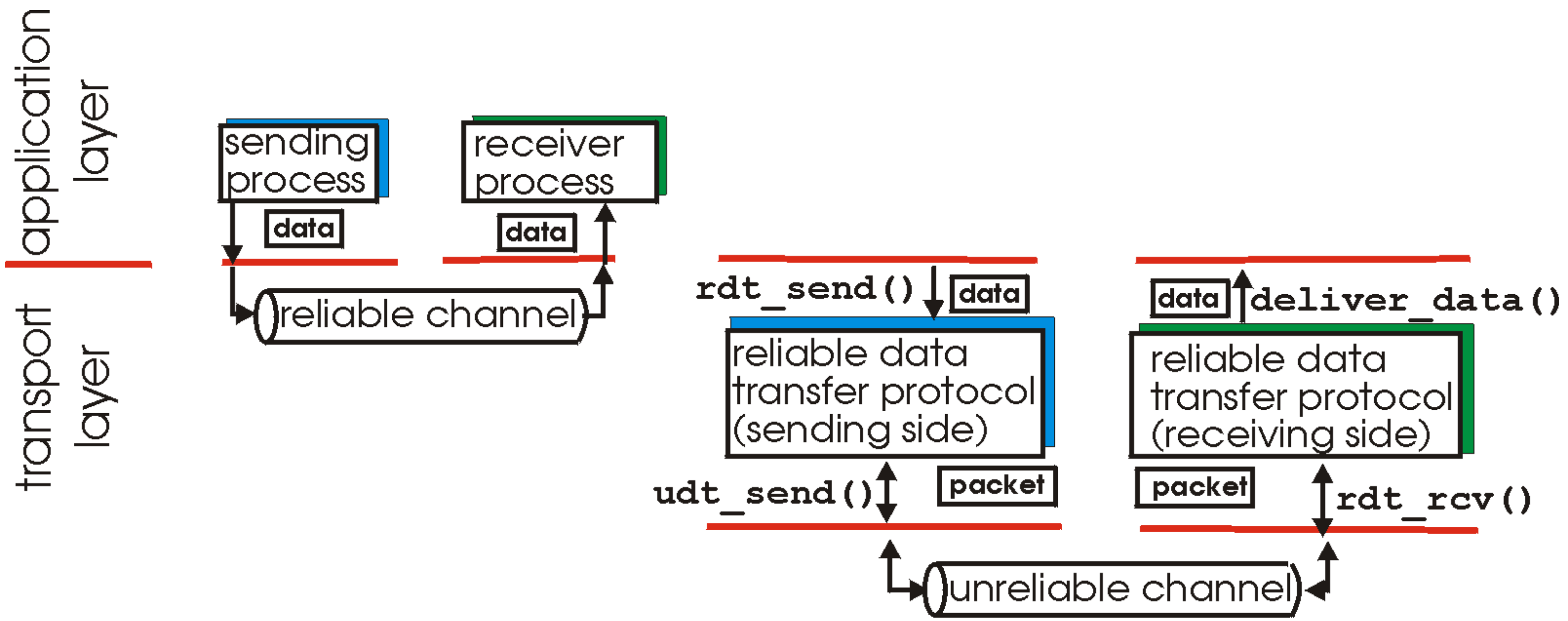
- important in application, transport, link layers
  - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable Transport

- important in application, transport, link layers
  - top-10 list of important networking topics!



(a) provided service

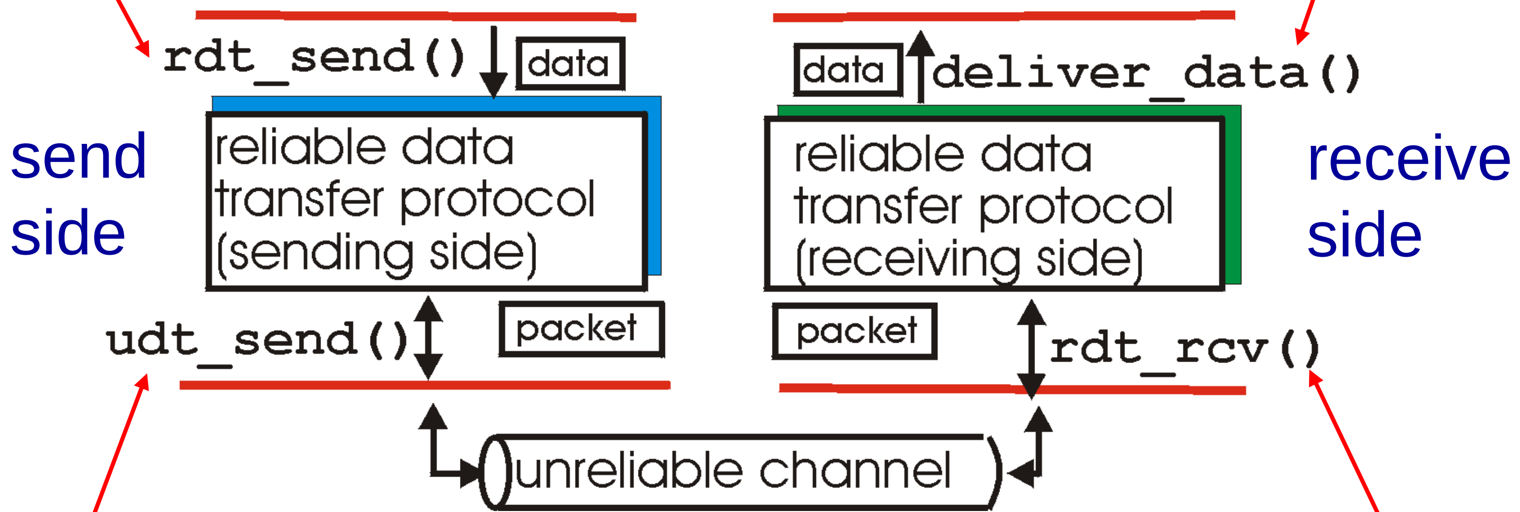
(b) service implementation

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable Transport

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

TCP: Reliable Transport

TCP: Transmission Control Protocol

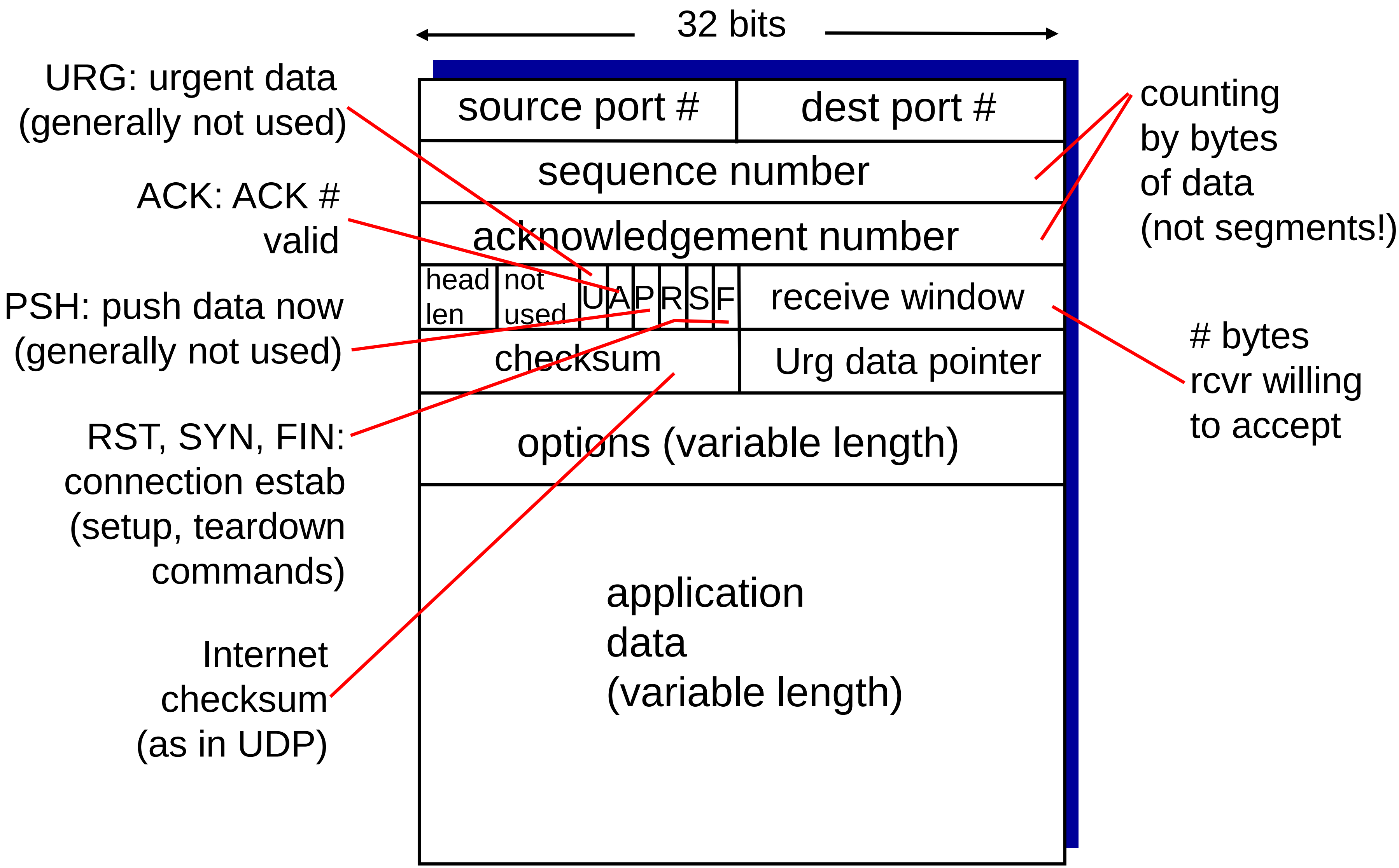
RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP: Reliable Transport

TCP: Segment Structure



TCP: Reliable Transport

TCP: Sequence numbers and Acks

<u>sequence numbers:</u>

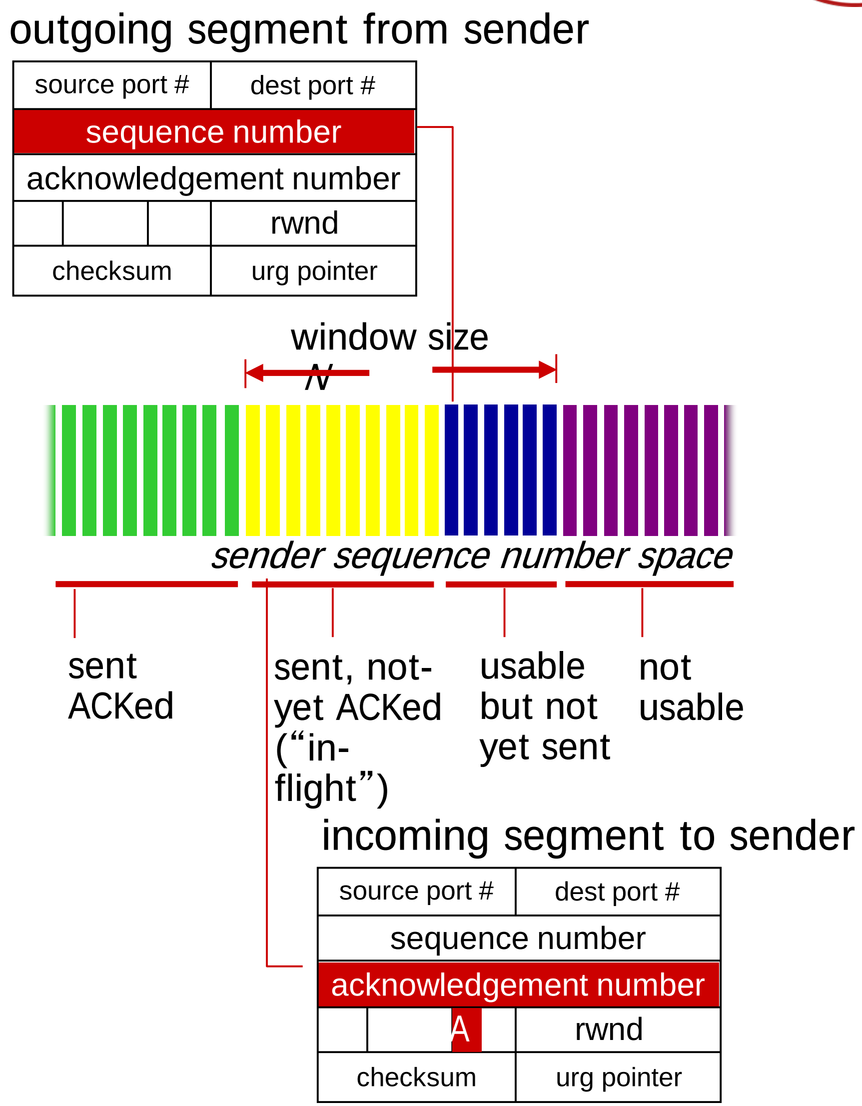
- byte stream "number" of first byte in segment's data

<u>acknowledgements:</u>

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

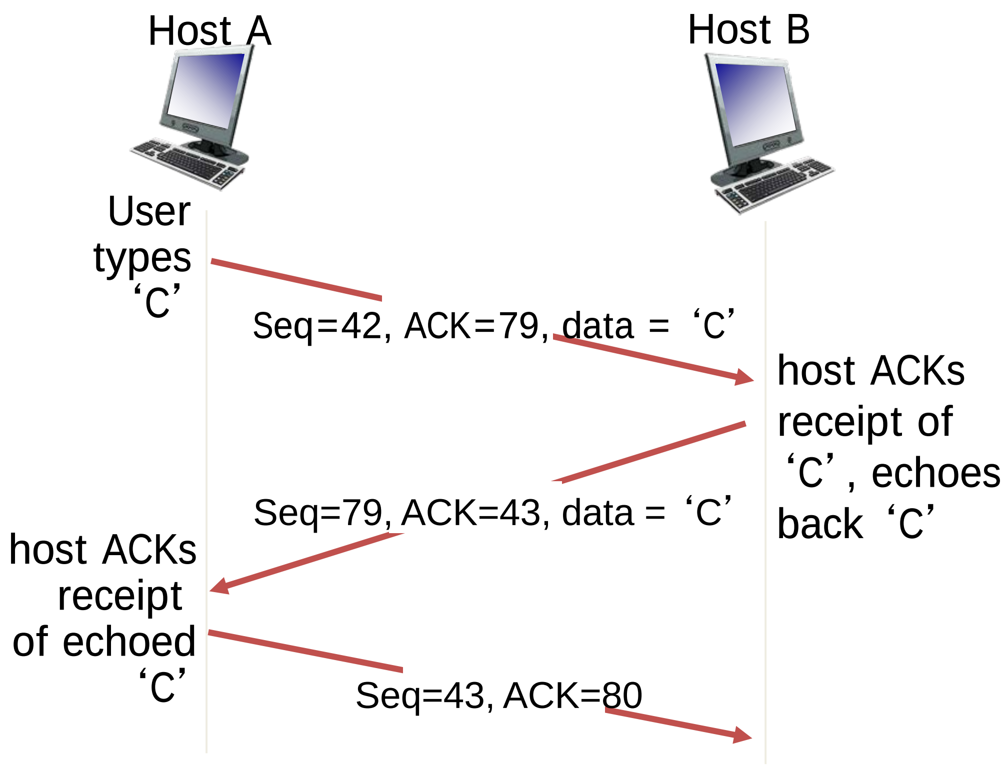
Q: how receiver handles out-of-order segments

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



TCP: Reliable Transport

TCP: Sequence numbers and Acks



simple telnet scenario

TCP: Reliable Transport

TCP: Transmission Control Protocol

RFCs: 793,1122,1323, 2018, 2581

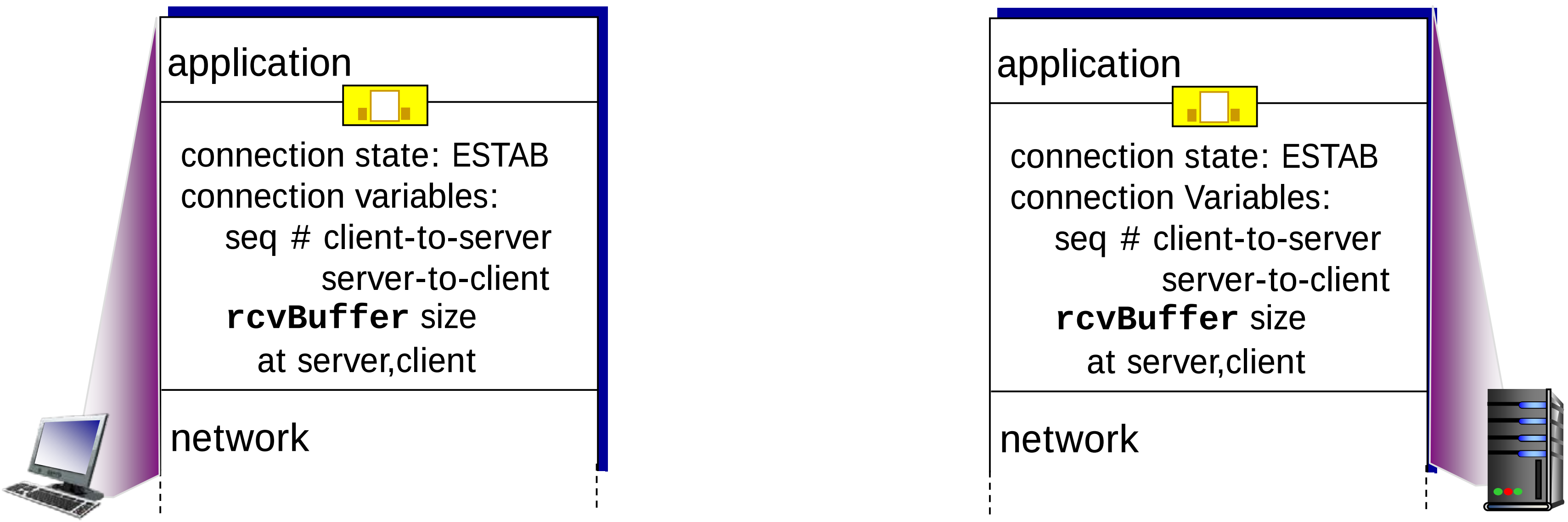
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

Connection Management: TCP 3-way handshake

before exchanging data, sender/receiver “handshake”:

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

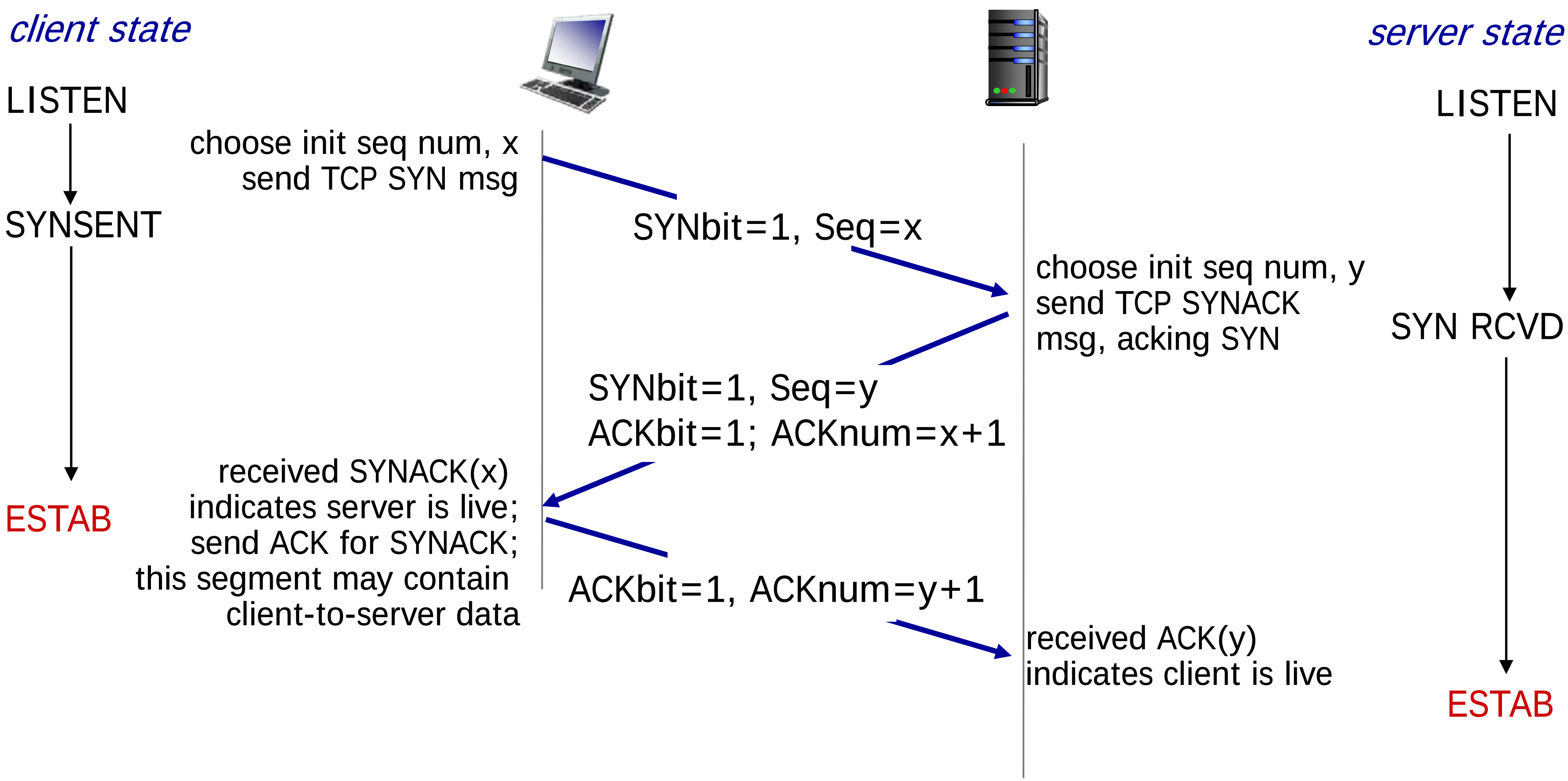



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

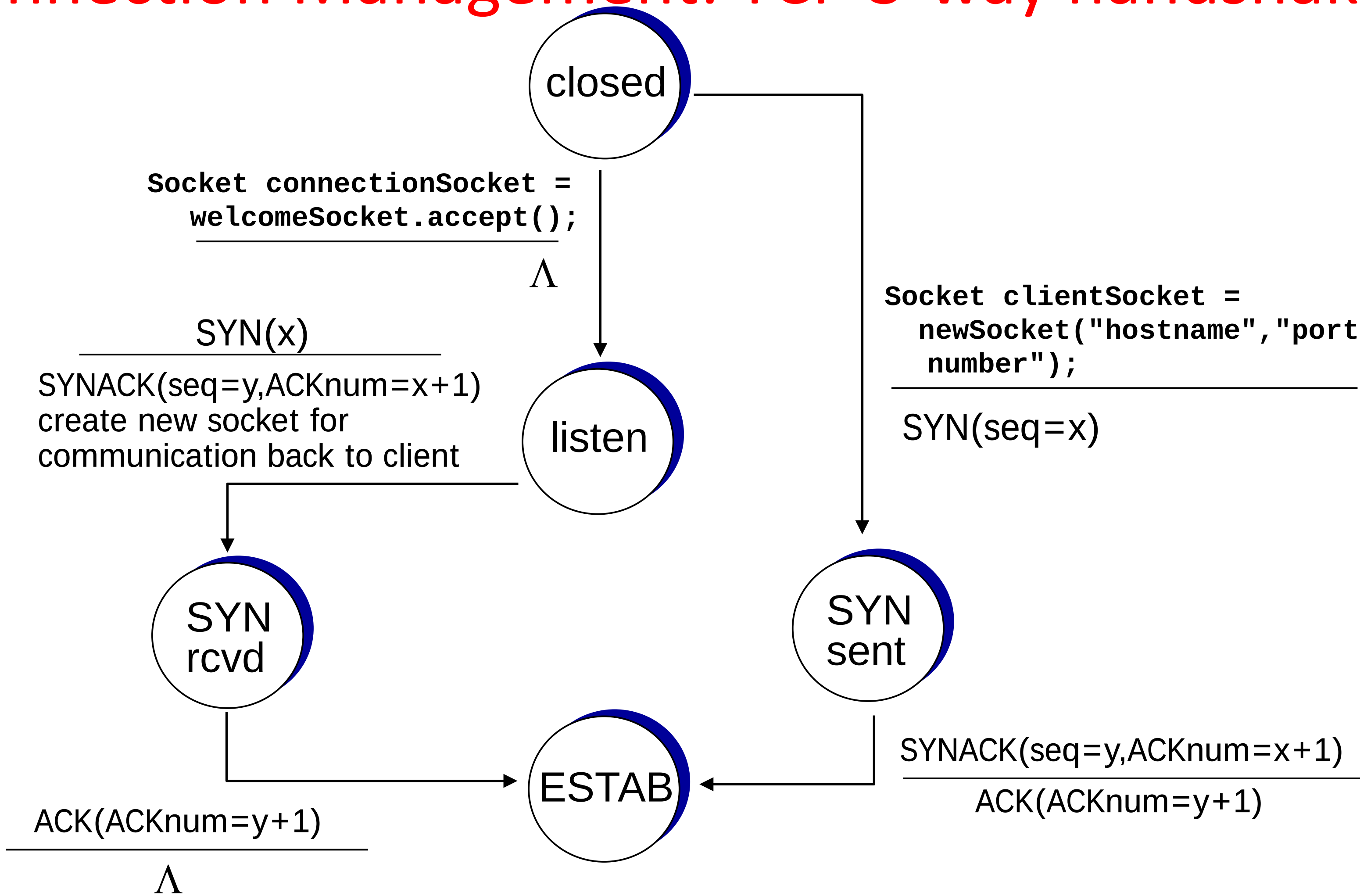
```
Socket connectionSocket =
    welcomeSocket.accept();
```

TCP: Reliable Transport

Connection Management: TCP 3-way handshake



Connection Management: TCP 3-way handshake



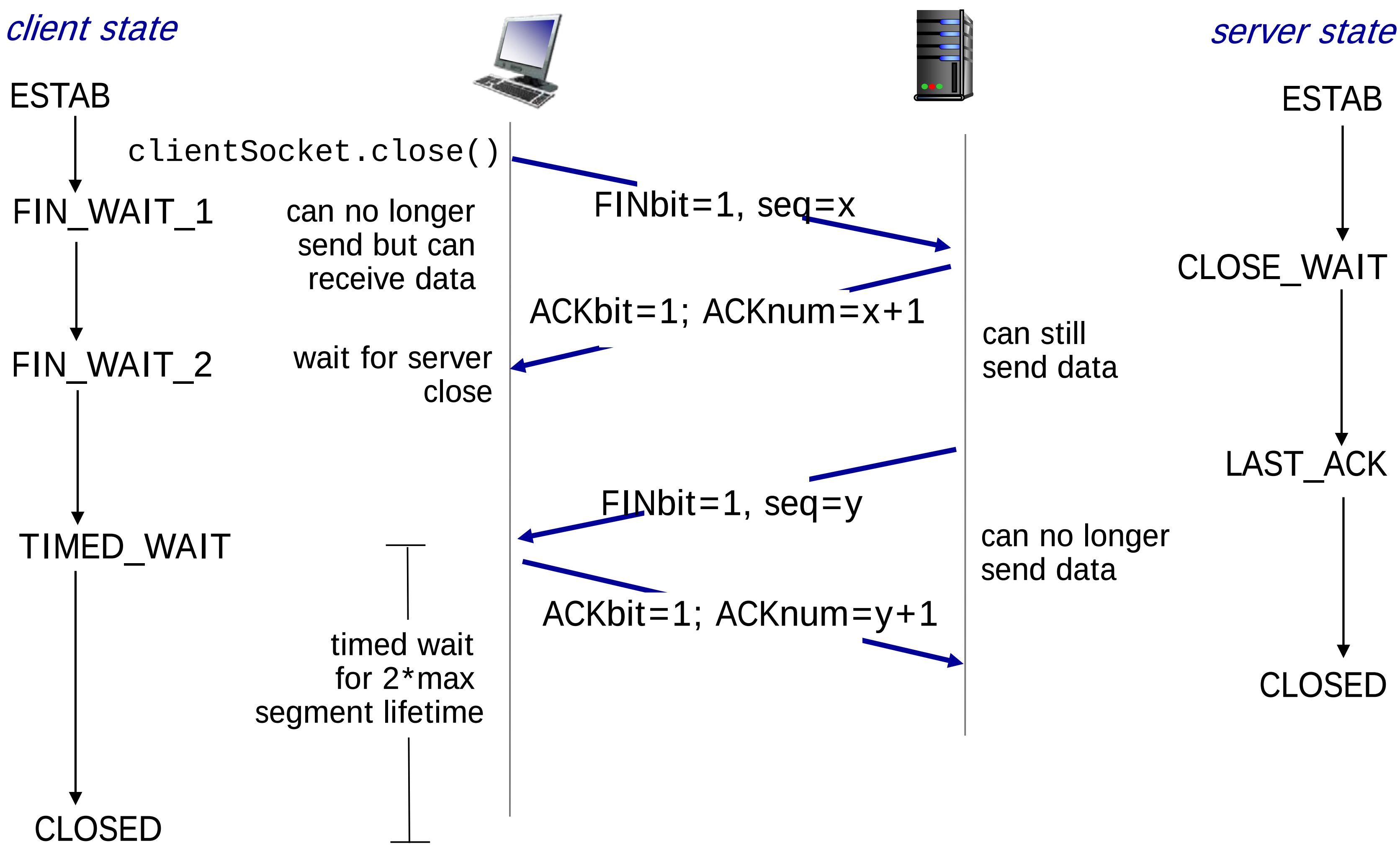
TCP: Reliable Transport

Connection Management: Closing connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: Reliable Transport

Connection Management: Closing connection



TCP: Reliable Transport

TCP: Transmission Control Protocol

RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP: Reliable Transport

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:

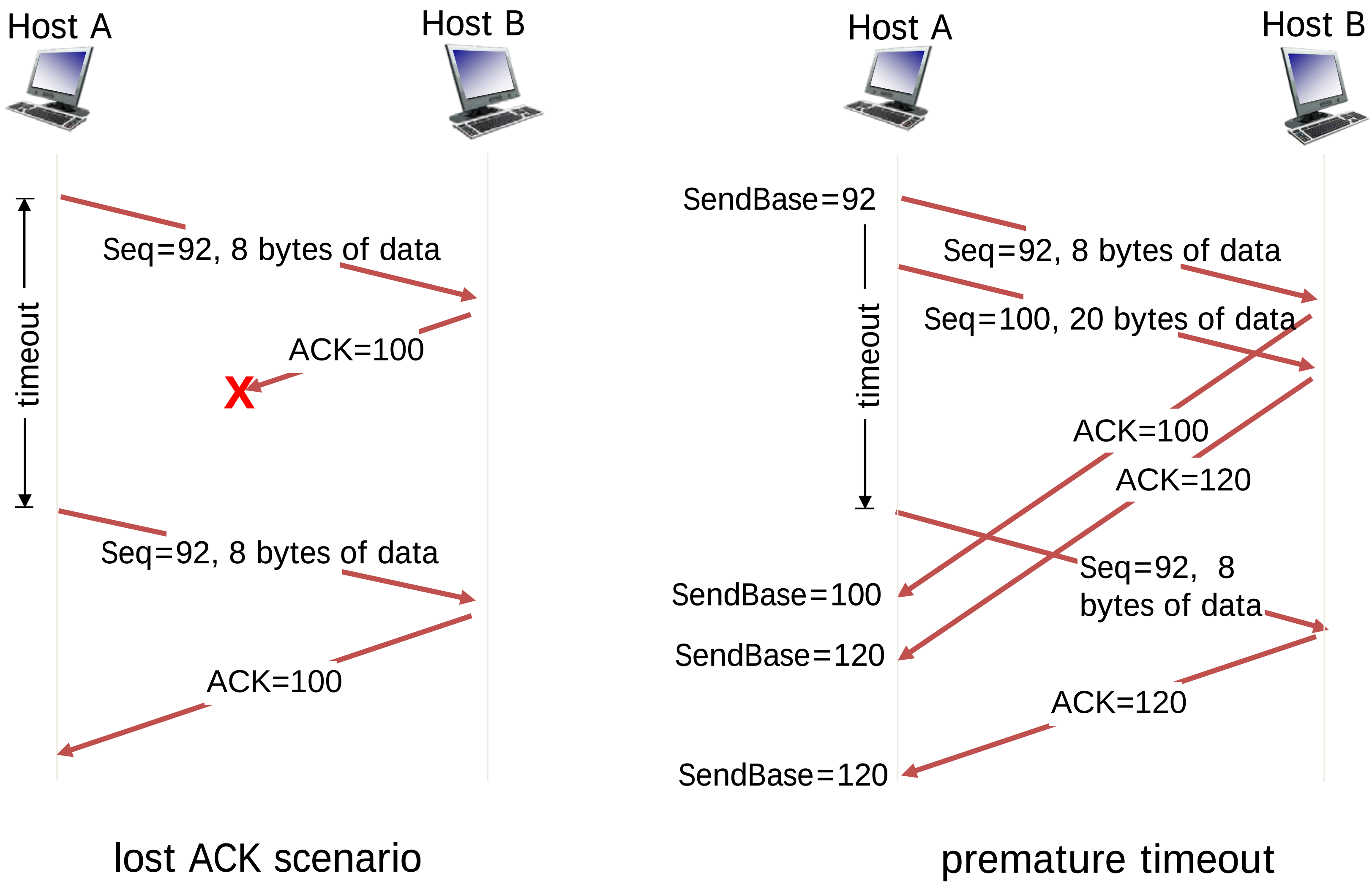
- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

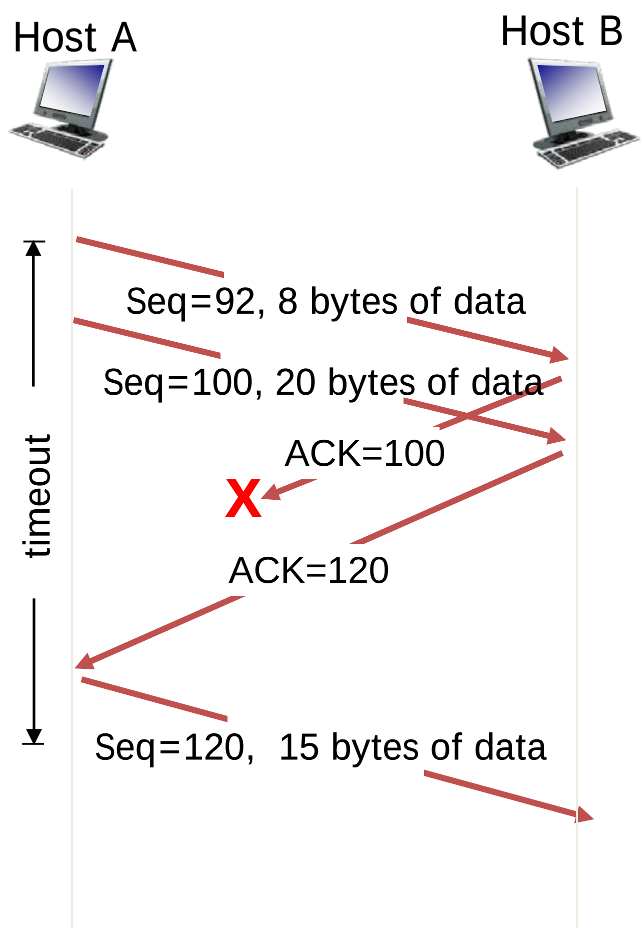
TCP: Reliable Transport

TCP: Retransmission Scenerios



TCP: Reliable Transport

TCP: Retransmission Scenerios



cumulative ACK

TCP ACK generation [RFC 1122, 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 starts at lower end of gap

TCP: Reliable Transport

TCP 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.

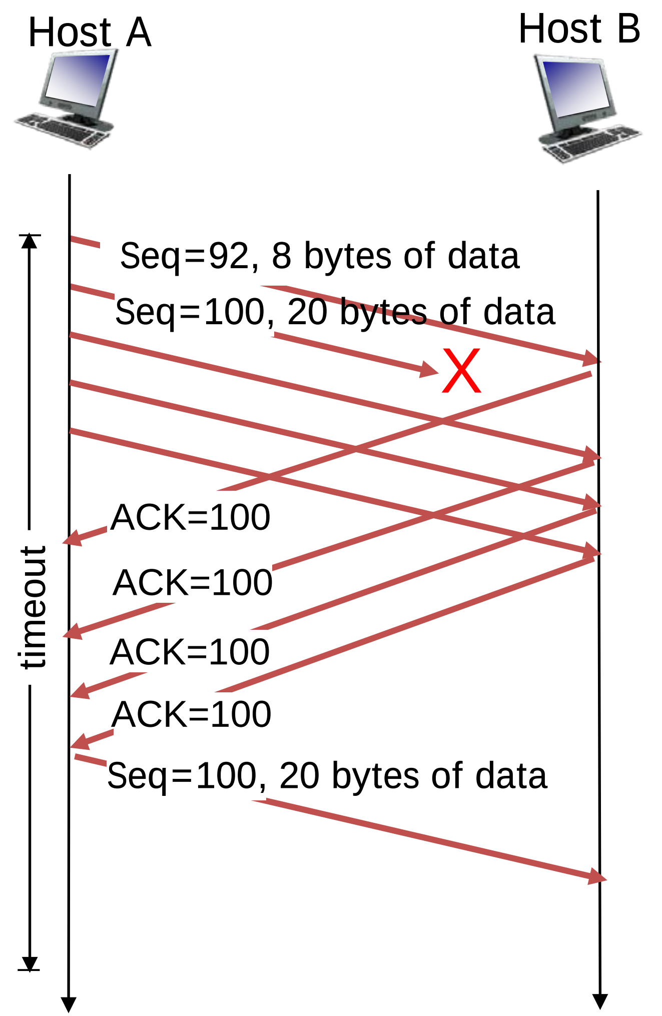
TCP fast retransmit

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

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

TCP: Reliable Transport

TCP Fast Retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP: Roundtrip time and timeouts

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

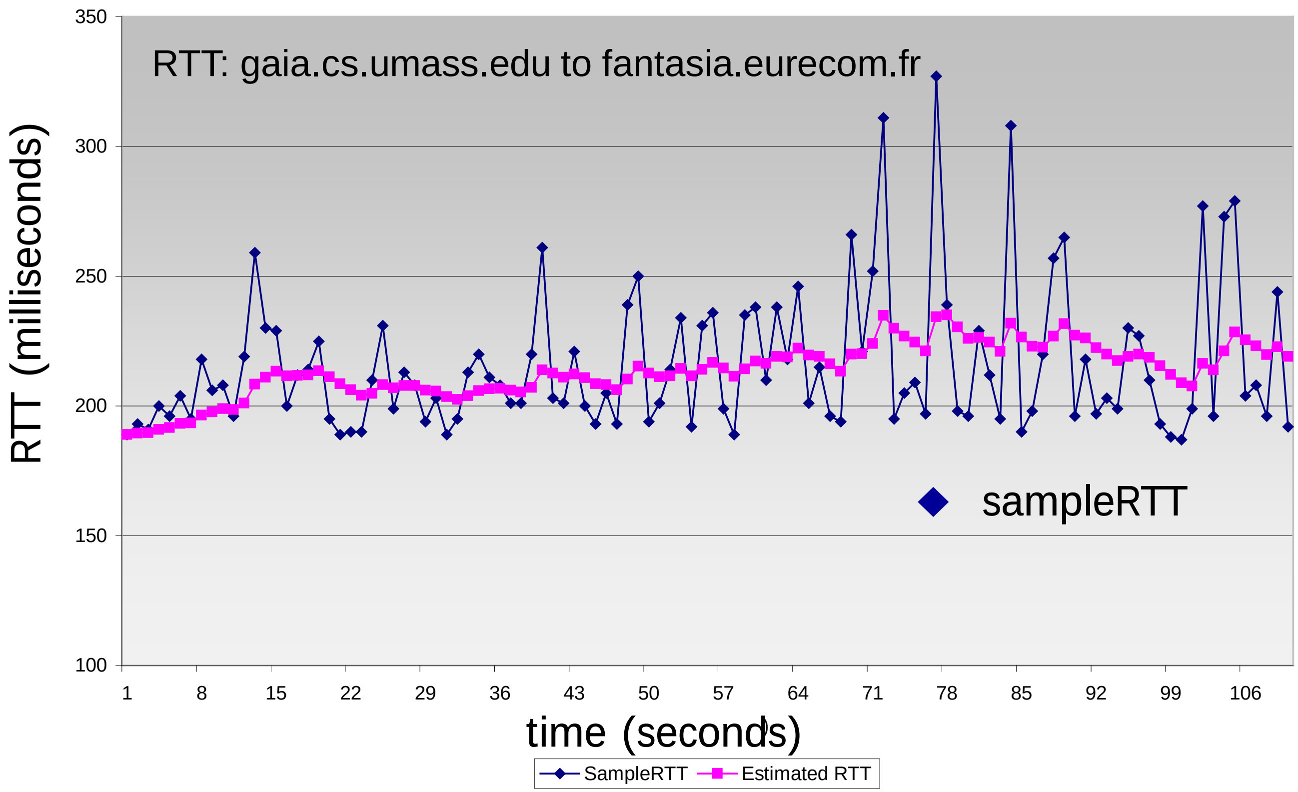
Q: how to estimate RTT?

- **`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: Roundtrip time and timeouts

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

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



TCP: Roundtrip time and timeouts

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT ->`** larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

         (typically, β = 0.25)
```

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

estimated RTT — "safety margin"

TCP: Reliable Transport

TCP: Transmission Control Protocol

RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

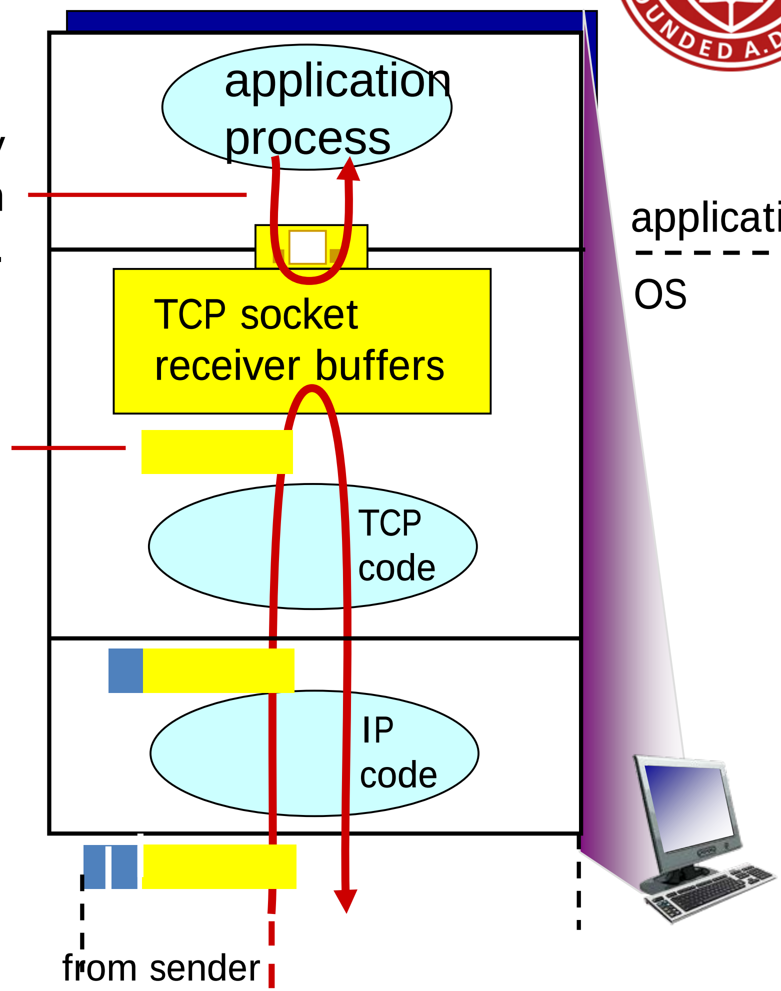
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP: Reliable Transport

Flow Control

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



receiver protocol stack

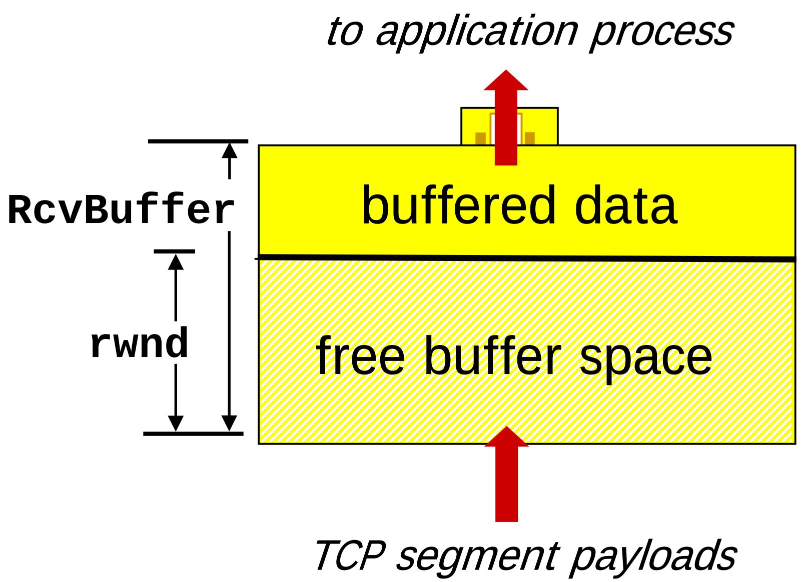
flow control

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

TCP: Reliable Transport

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 (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



receiver-side buffering

Goals for Today

- Transport Layer
  - Abstraction / services
  - Multiplexing/Demultiplexing
  - UDP: Connectionless Transport
  - TCP: Reliable Transport
    - Abstraction, Connection Management, Reliable Transport, Flow Control, timeouts
  - Congestion control

- Data Center TCP
  - Incast Problem

Principles of Congestion Control

congestion:

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

Principles of Congestion Control

two broad approaches towards congestion control:

end-end congestion control:

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

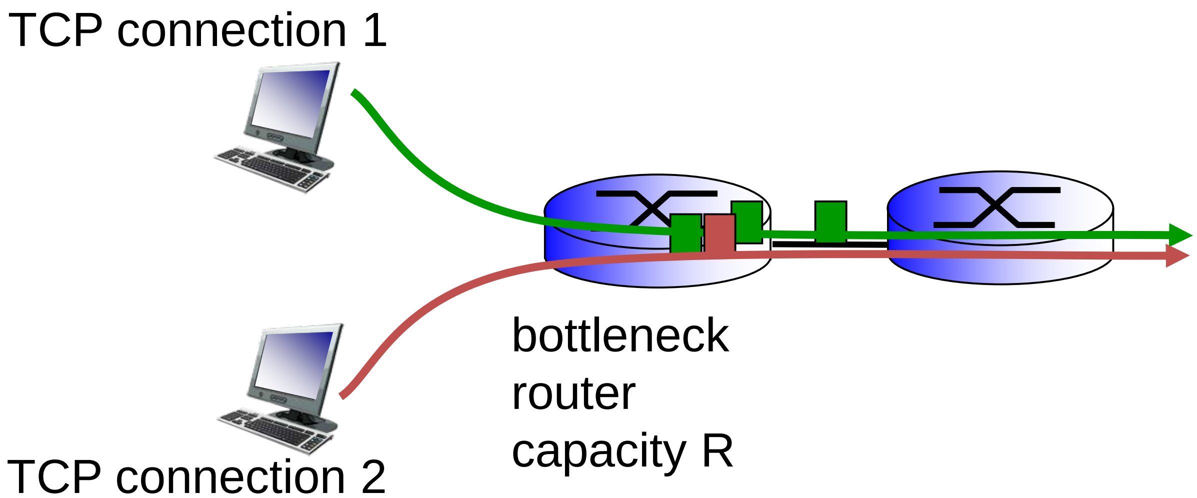
network-assisted congestion control:

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

TCP Congestion Control

TCP Fairness

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

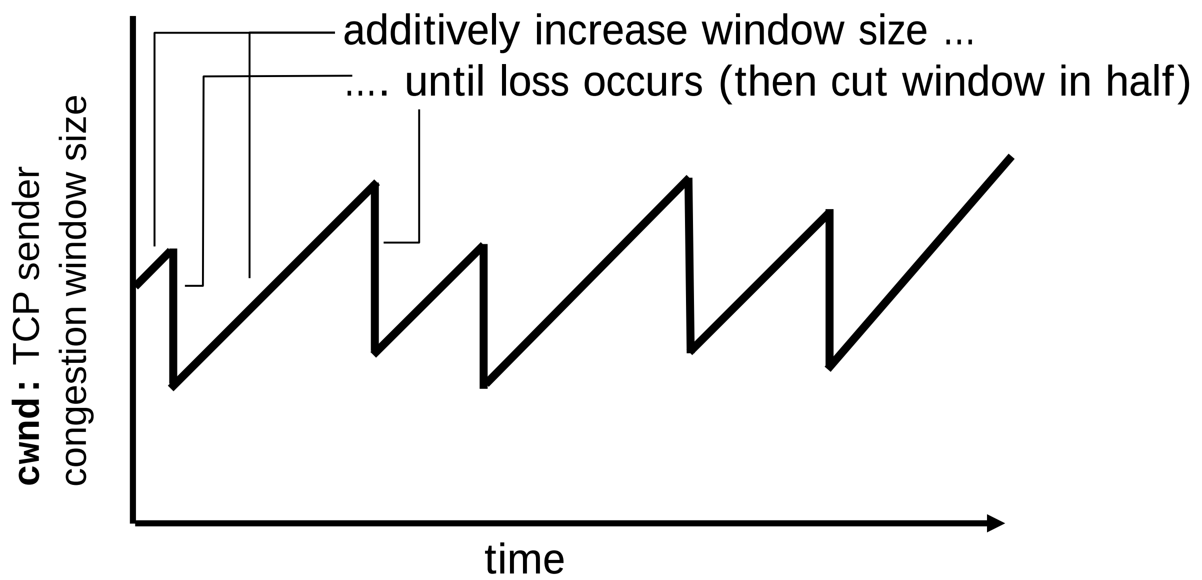


TCP Fairness: Why is TCP Fair?

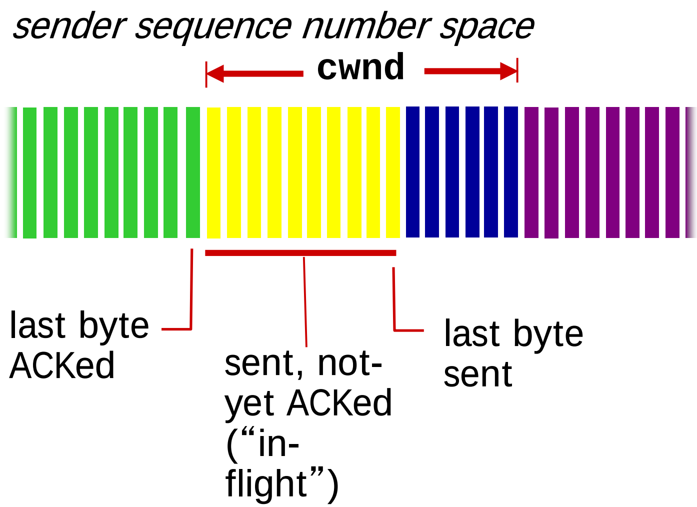
AIMD: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control



❖ sender limits transmission:

$$\boxed{\texttt{LastByteSent-LastByteAcked} \le \texttt{cwnd}}$$

❖ **cwnd** is dynamic, function of perceived network congestion

TCP sending rate:

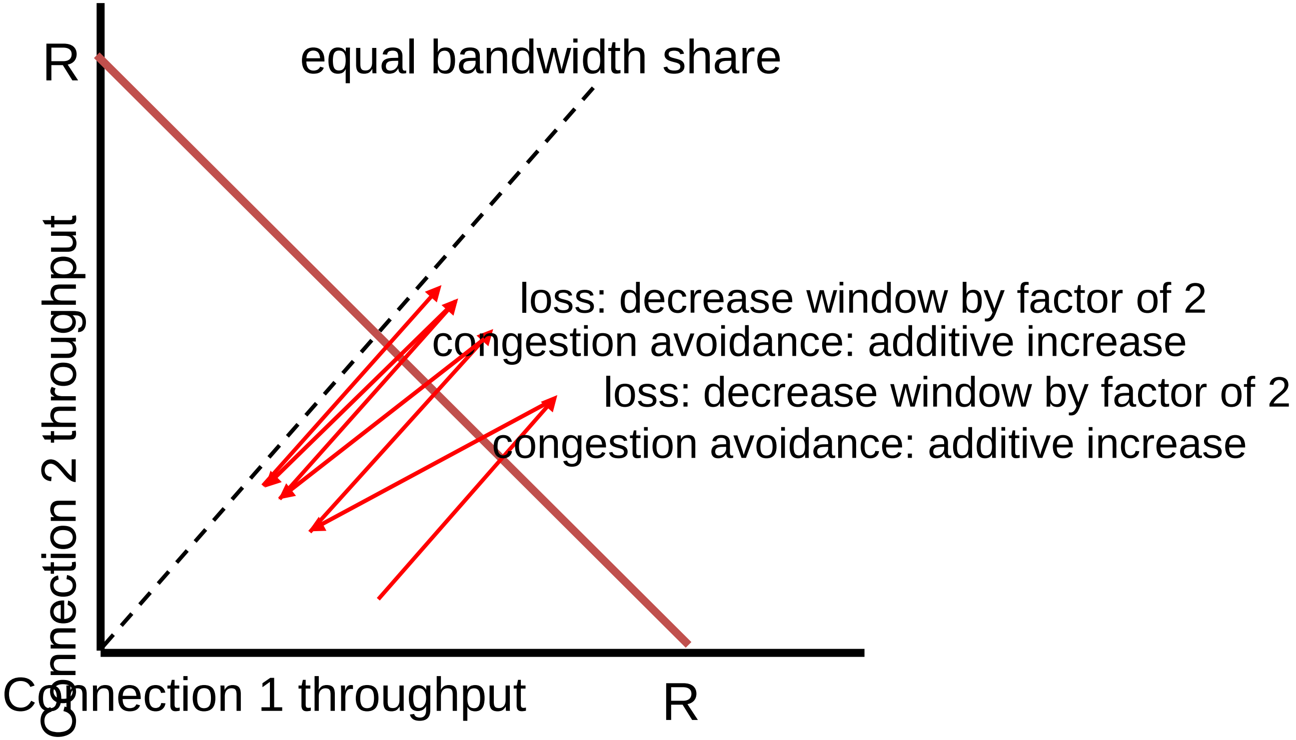
❖ *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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

TCP Fairness: Why is TCP Fair?

two competing sessions:

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



TCP Congestion Control

TCP Fairness

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

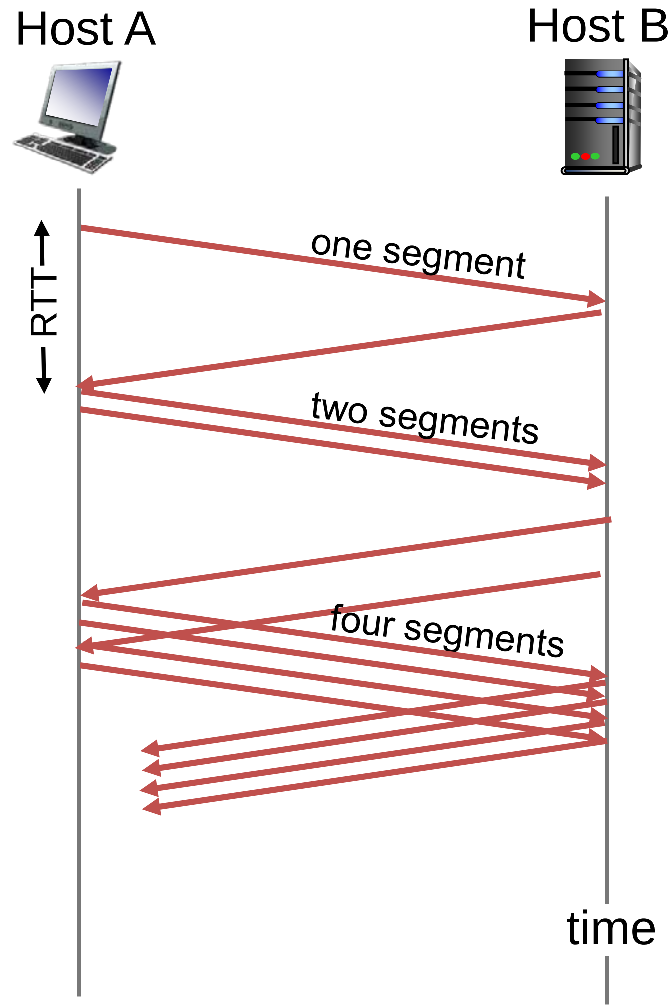
Fairness, parallel TCP connections

- 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 Congestion Control

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 Congestion Control

Detecting and Reacting to Loss

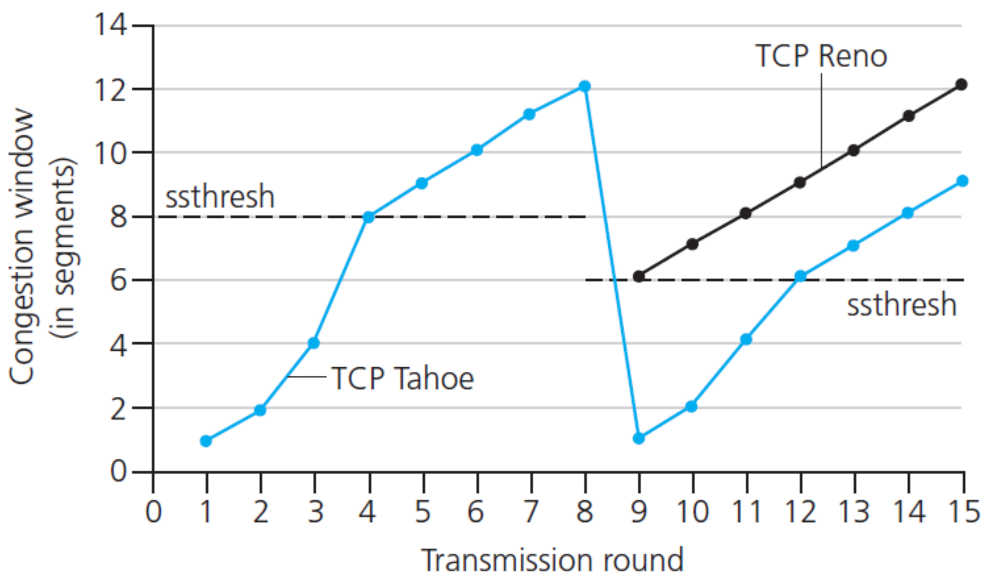
- loss indicated by timeout:
  - **cwnd** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - **cwnd** is cut in half window then grows linearly
- TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

TCP Congestion Control

Switching from Slow Start to Congestion Avoidance (CA)

Q: when should the exponential increase switch to linear?

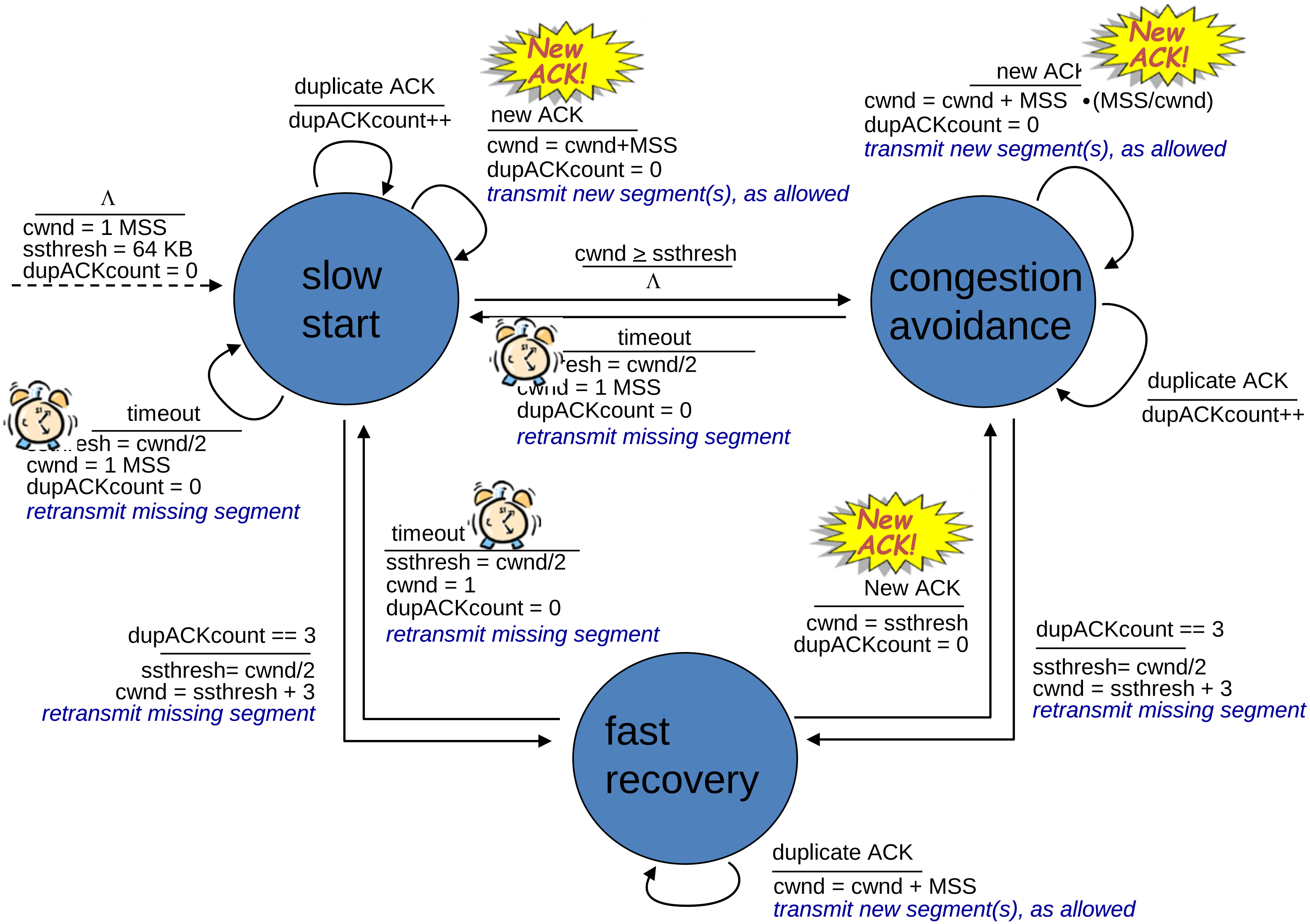
A: when **cwnd** gets to 1/2 of its value before timeout.



Implementation:

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

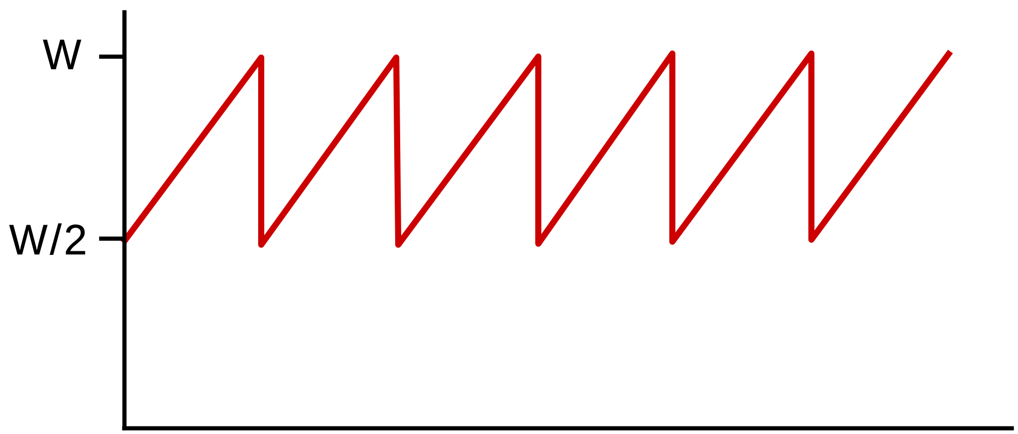
TCP Congestion Control



TCP Throughput

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

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



TCP over "long, fat pipes"

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ *– a very small loss rate!*

- new versions of TCP for high-speed

Goals for Today

- Transport Layer
  - Abstraction / services
  - Multiplexing/Demultiplexing
  - UDP: Connectionless Transport
  - TCP: Reliable Transport
    - Abstraction, Connection Management, Reliable Transport, Flow Control, timeouts
  - Congestion control

- Data Center TCP
  - Incast Problem

Slides used judiciously from "Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems", A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. *Proc. of USENIX File and Storage Technologies (FAST)*, February 2008.

TCP Throughput Collapse

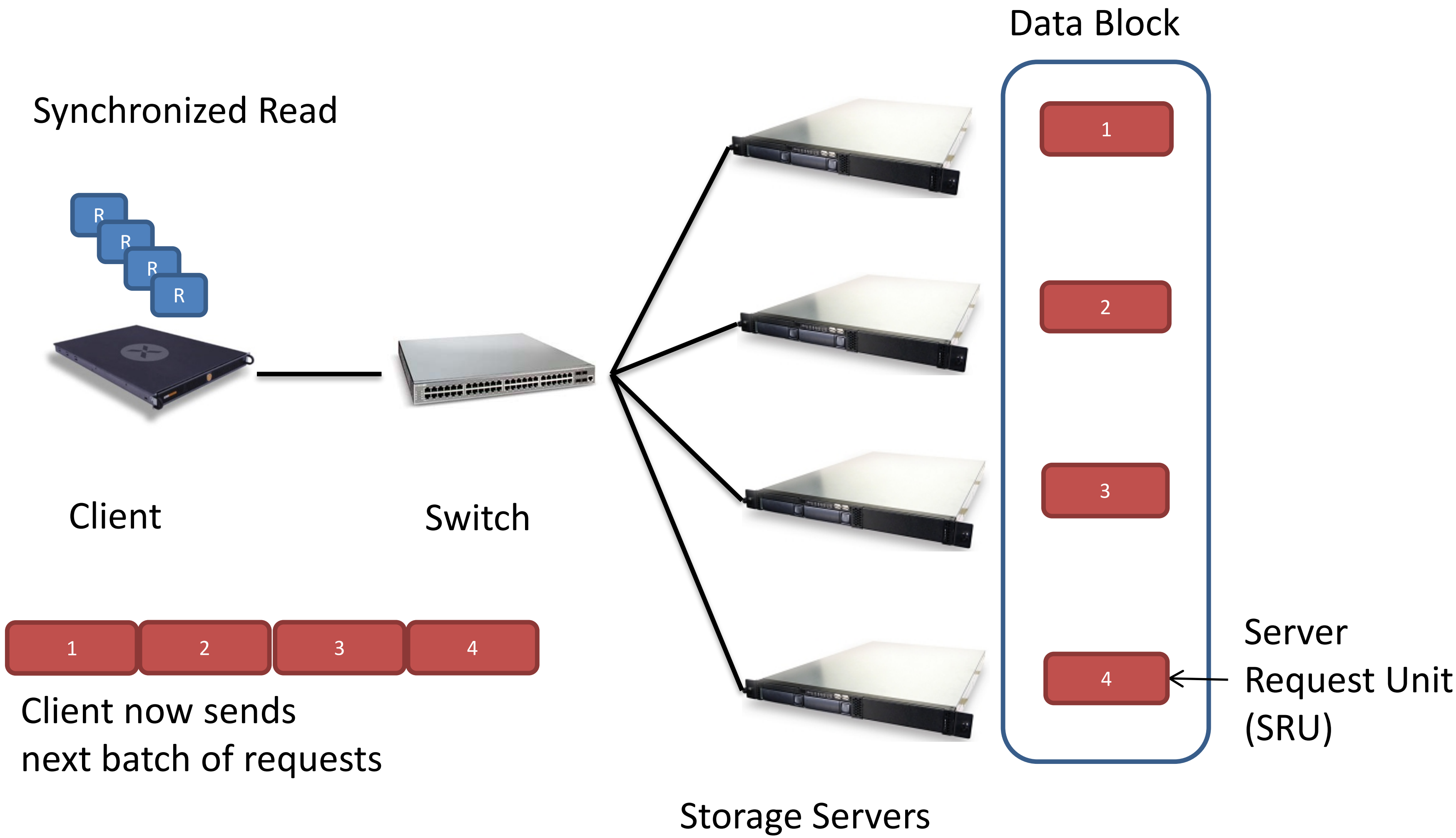
What happens when TCP is "too friendly"?

E.g.

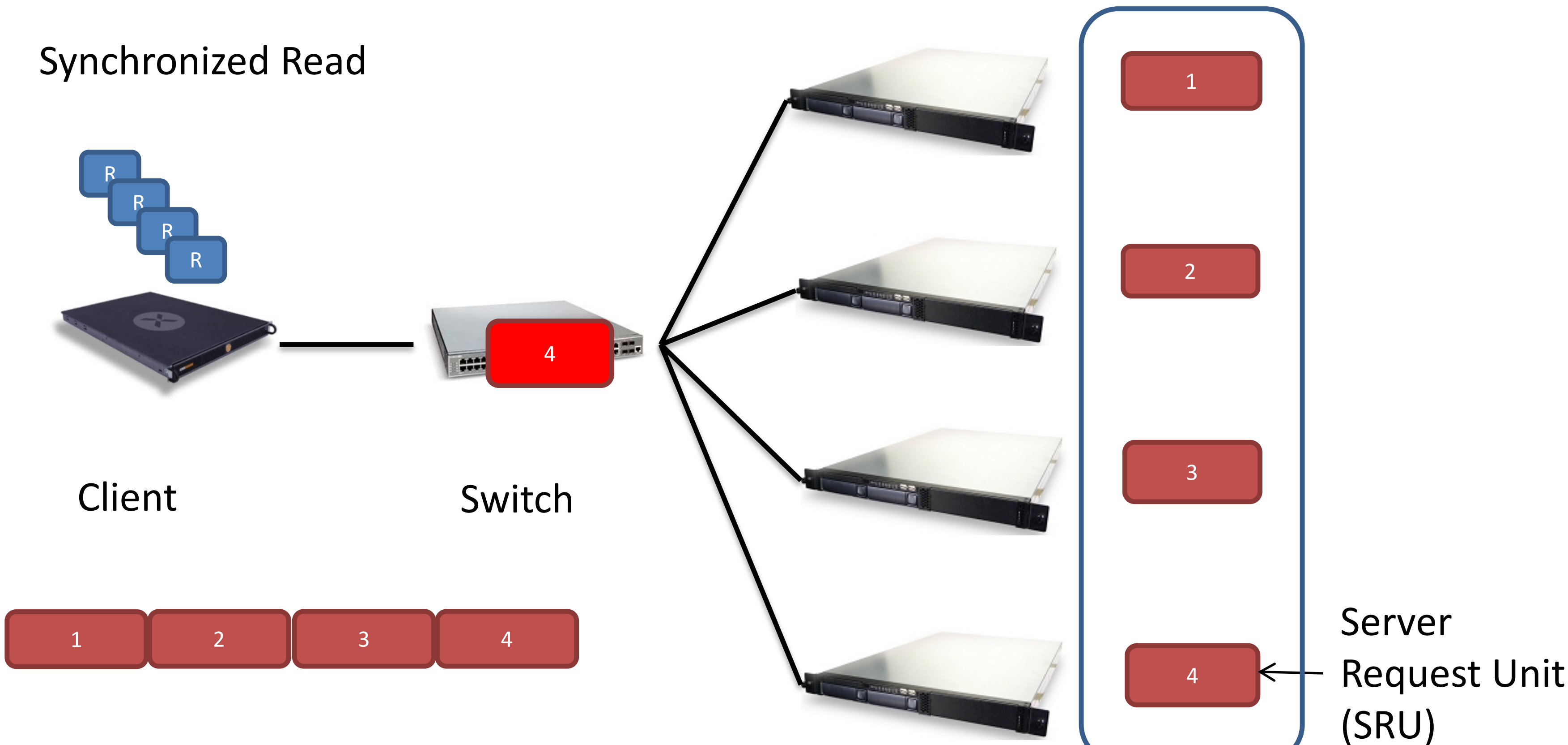
- Test on an Ethernet-based storage cluster
- Client performs synchronized reads
- Increase # of servers involved in transfer
  - SRU size is fixed
- TCP used as the data transfer protocol

Slides used judiciously from "Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems", A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. *Proc. of USENIX File and Storage Technologies (FAST)*, February 2008.

Cluster-based Storage Systems

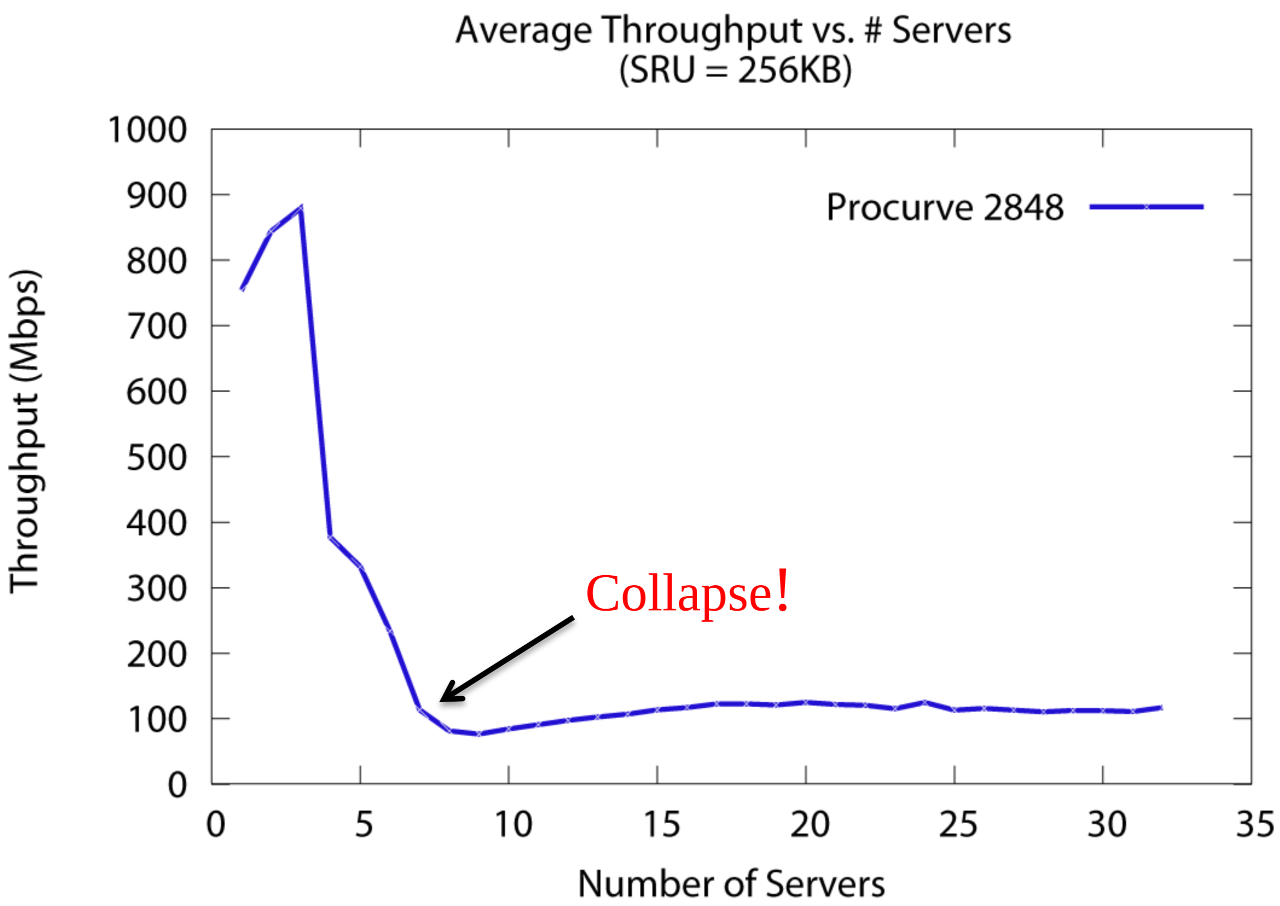


Link idle time due to timeouts



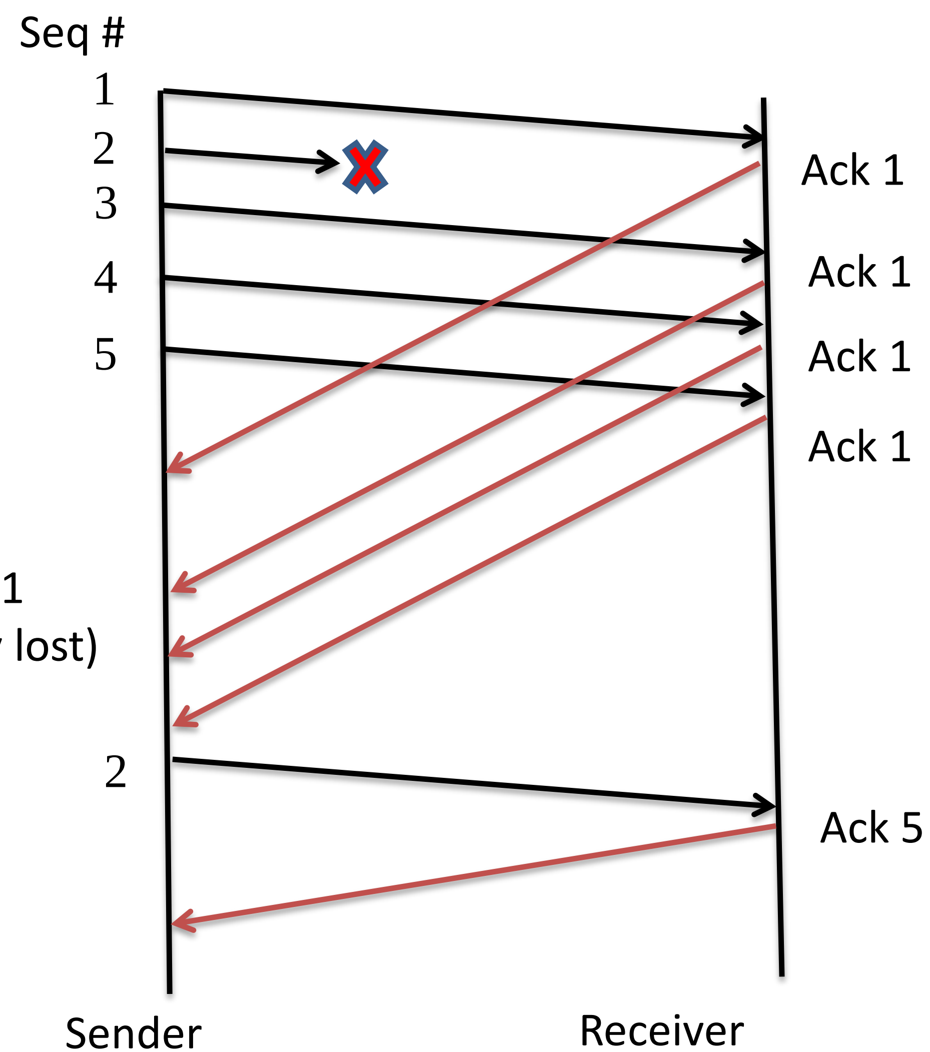
Link is idle until server experiences a timeout

TCP Throughput Collapse: Incast



- [Nagle04] called this *Incast*
- Cause of throughput collapse: TCP timeouts

TCP: data-driven loss recovery

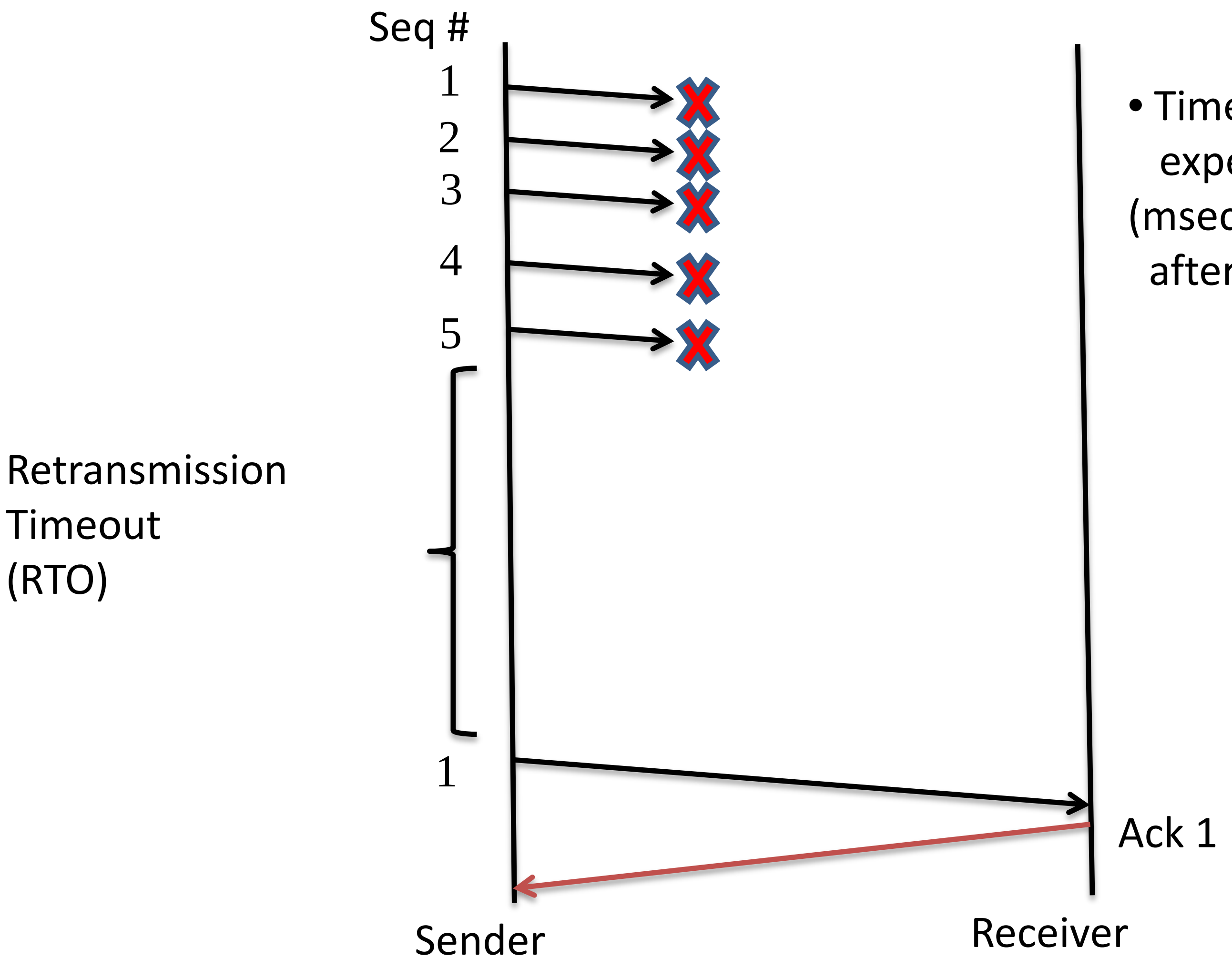


3 duplicate ACKs for 1
(packet 2 is probably lost)

Retransmit packet 2
immediately

In SANs
recovery in usecs
after loss.

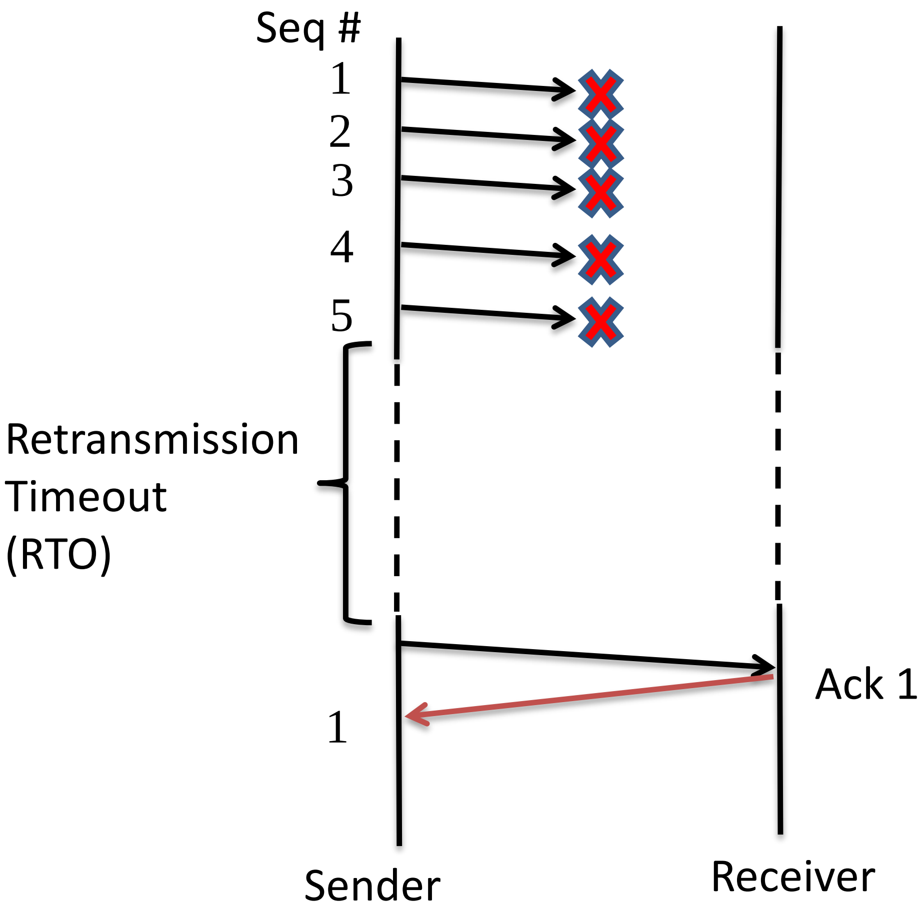
TCP: timeout-driven loss recovery



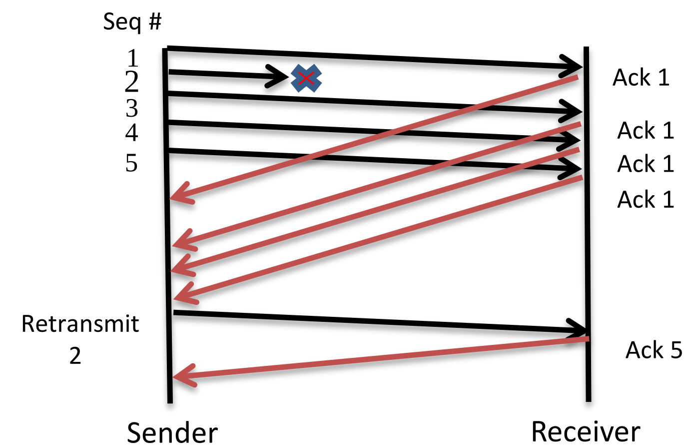
- Timeouts are expensive
(msecs to recover after loss)

TCP: Loss recovery comparison

Timeout driven recovery is
slow (ms)



Data-driven recovery is
super fast (us) in SANs



TCP Throughput Collapse Summary

- Synchronized Reads and TCP timeouts cause TCP Throughput Collapse

- Previously tried options
  - Increase buffer size (costly)
  - Reduce RTOmin (unsafe)
  - Use Ethernet Flow Control (limited applicability)
- DCTCP (Data Center TCP)
  - Limited in-network buffer (queue length) via both in-network signaling and end-to-end, TCP, modifications

Perspective

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

<u>Next time:</u>

- Network Layer
- leaving the network "edge" (application, transport layers)
- into the network "core"

Before Next time

- Project Proposal
  - due in one week
  - Meet with groups, TA, and professor
- Lab1
  - Single threaded TCP proxy
  - Due in one week, next Friday

- No required reading and review due
- But, review chapter 4 from the book, Network Layer
  - We will also briefly discuss data center topologies
  -
- Check website for updated schedule