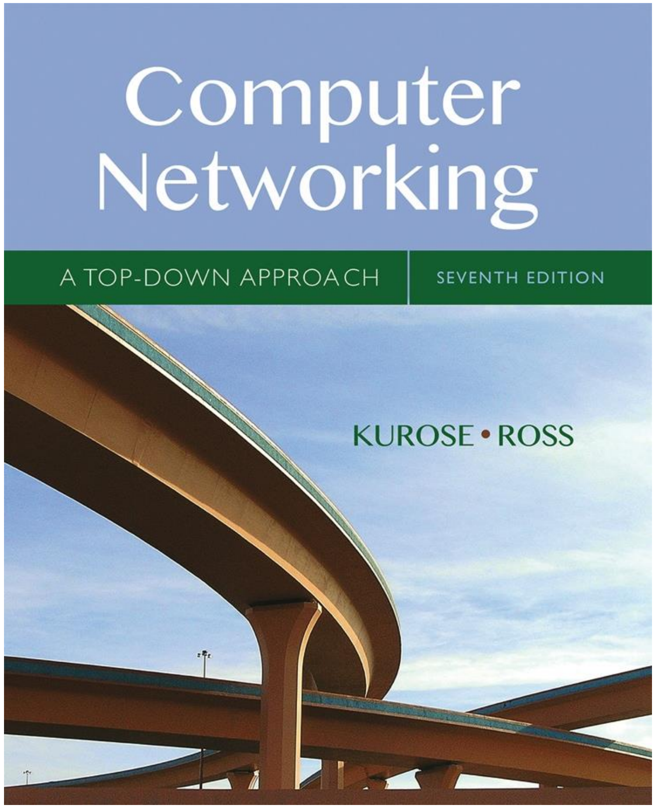
Chapter 3
Transport Layer



Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Transport Layer

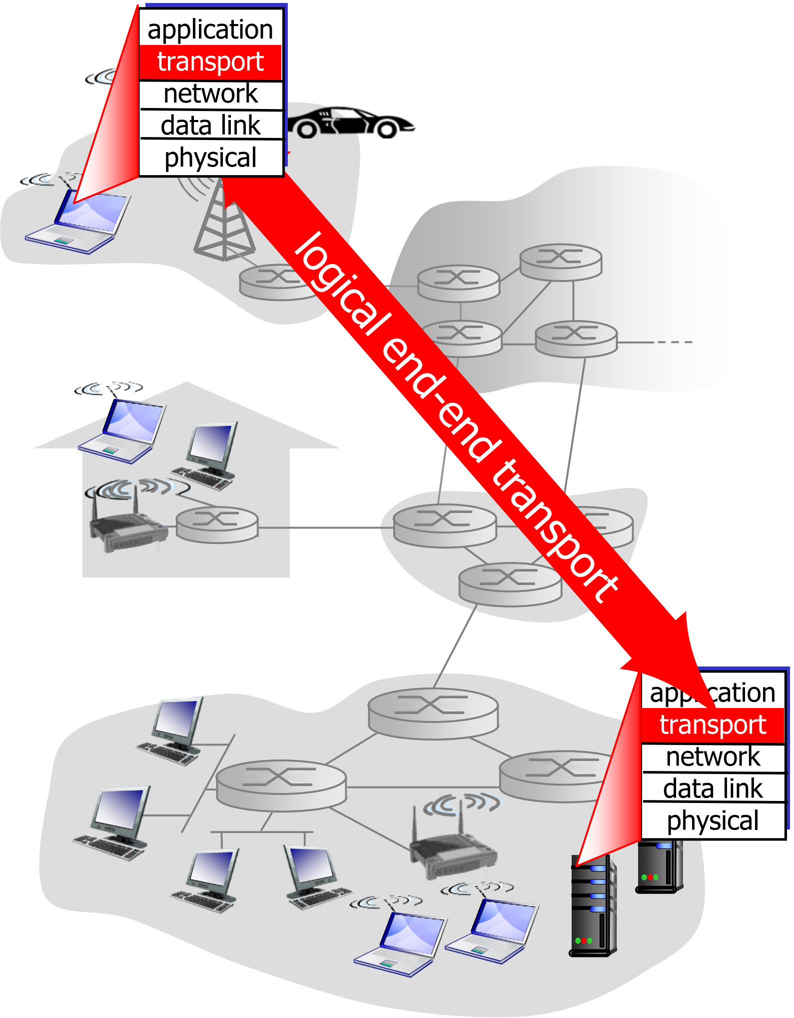
Chapter 3: Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Transport services and 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 vs. network layer

- *network layer:* logical communication between hosts (IP address to IP address)
- *transport layer:* logical communication between processes (port # to port #)
  - relies on network layer services

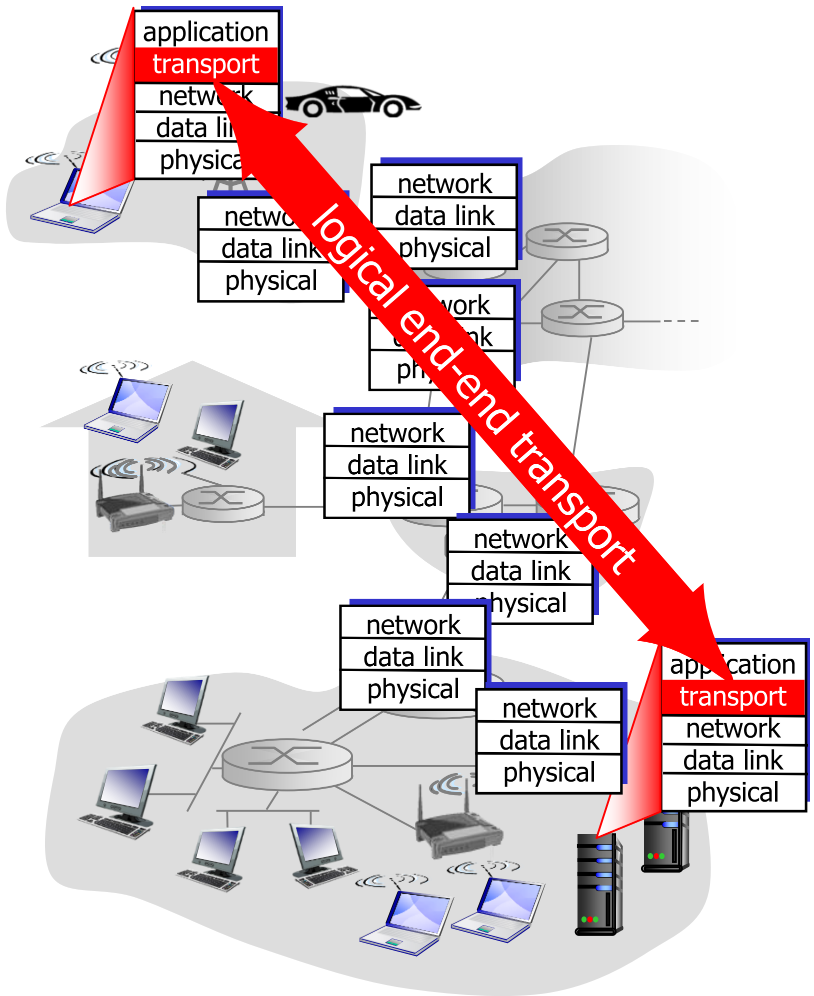
household analogy:

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

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

Internet transport-layer 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 guaranteed:
  - no delay guarantees
  - no bandwidth guarantees

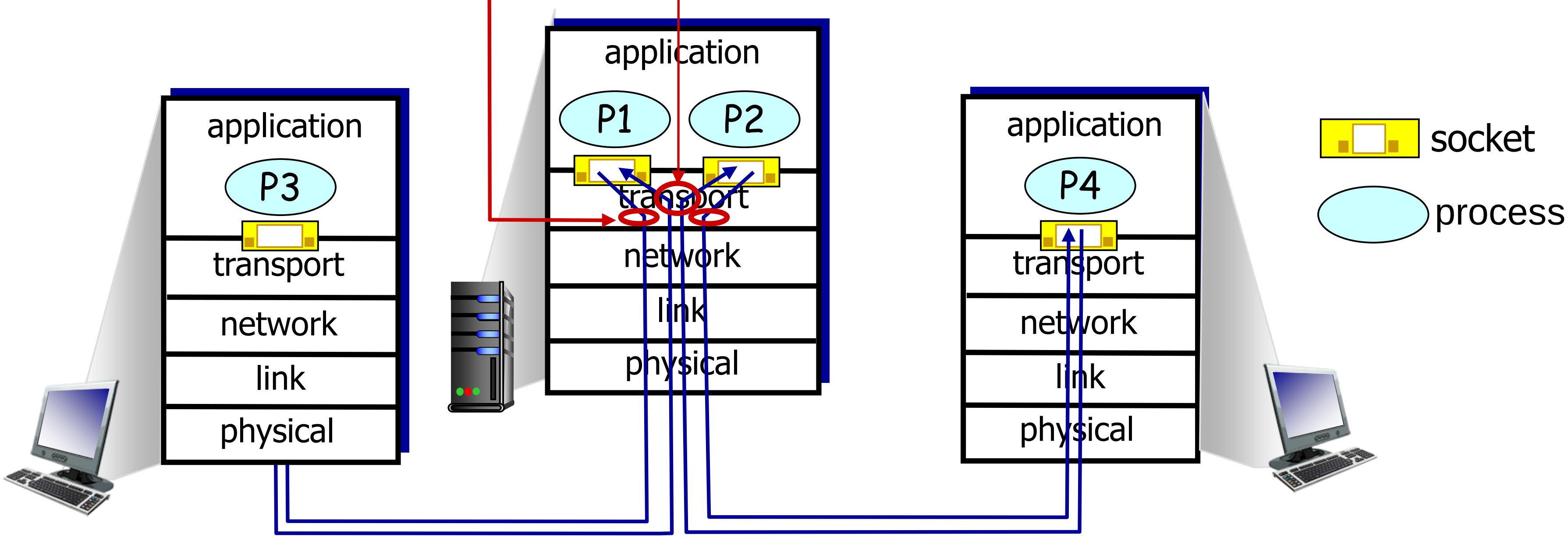


Protocol Multiplexing

Multiplexing/demultiplexing

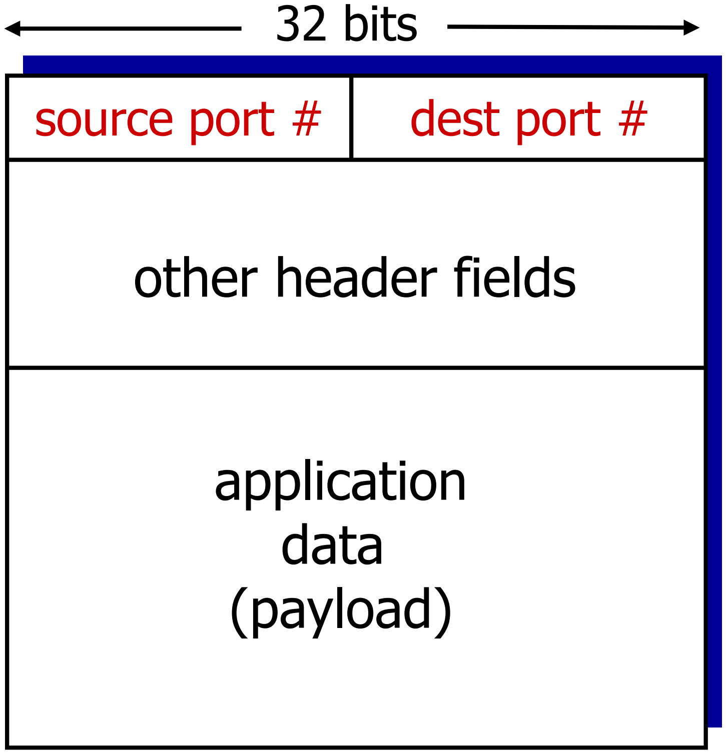
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source and destination port #
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

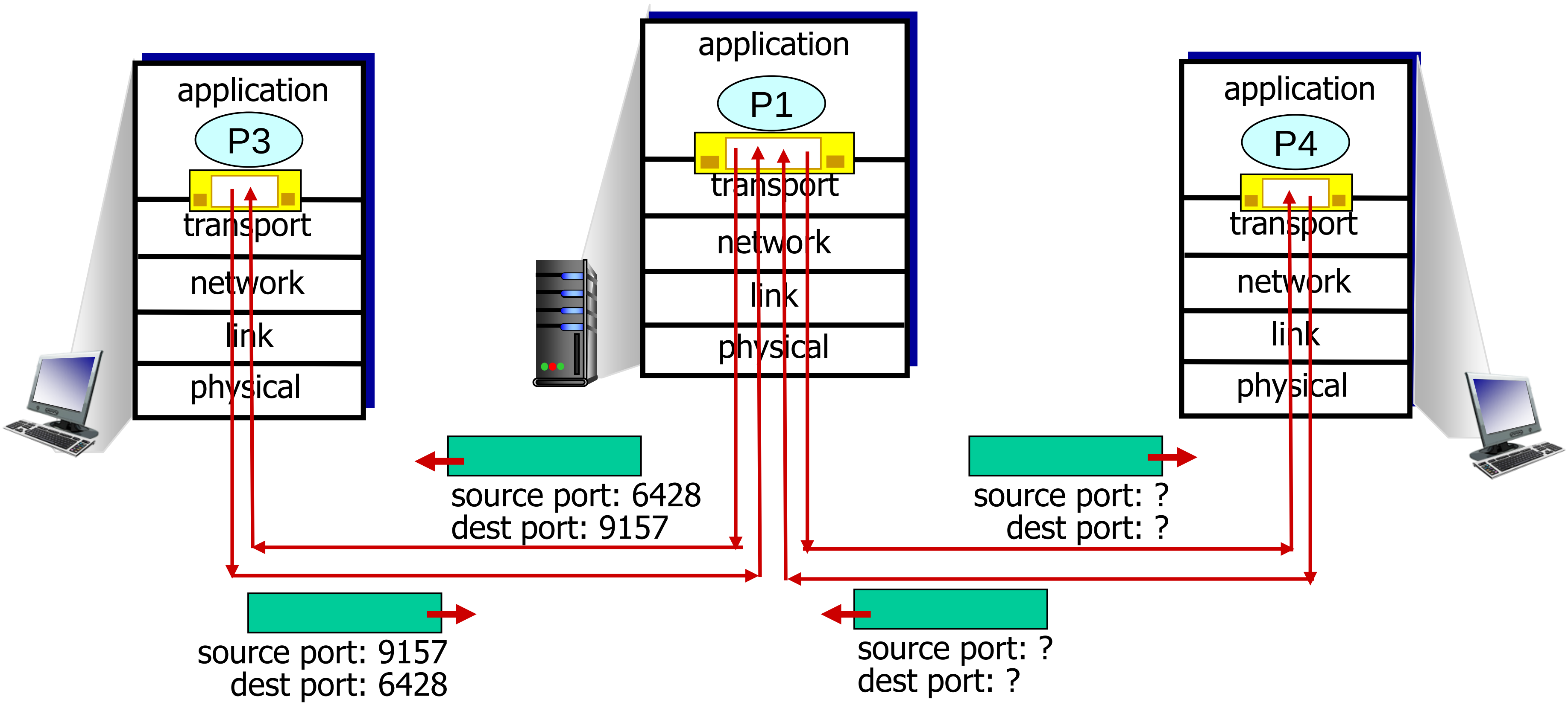
Connectionless demultiplexing

- *recall:* when sending datagrams into UDP socket, we must specify
  - destination IP address
  - destination port #

- when dest. host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

IP datagrams with *same dest. port #,* but different source IP addresses (and/or source port #) will be directed to *same socket* at destination host

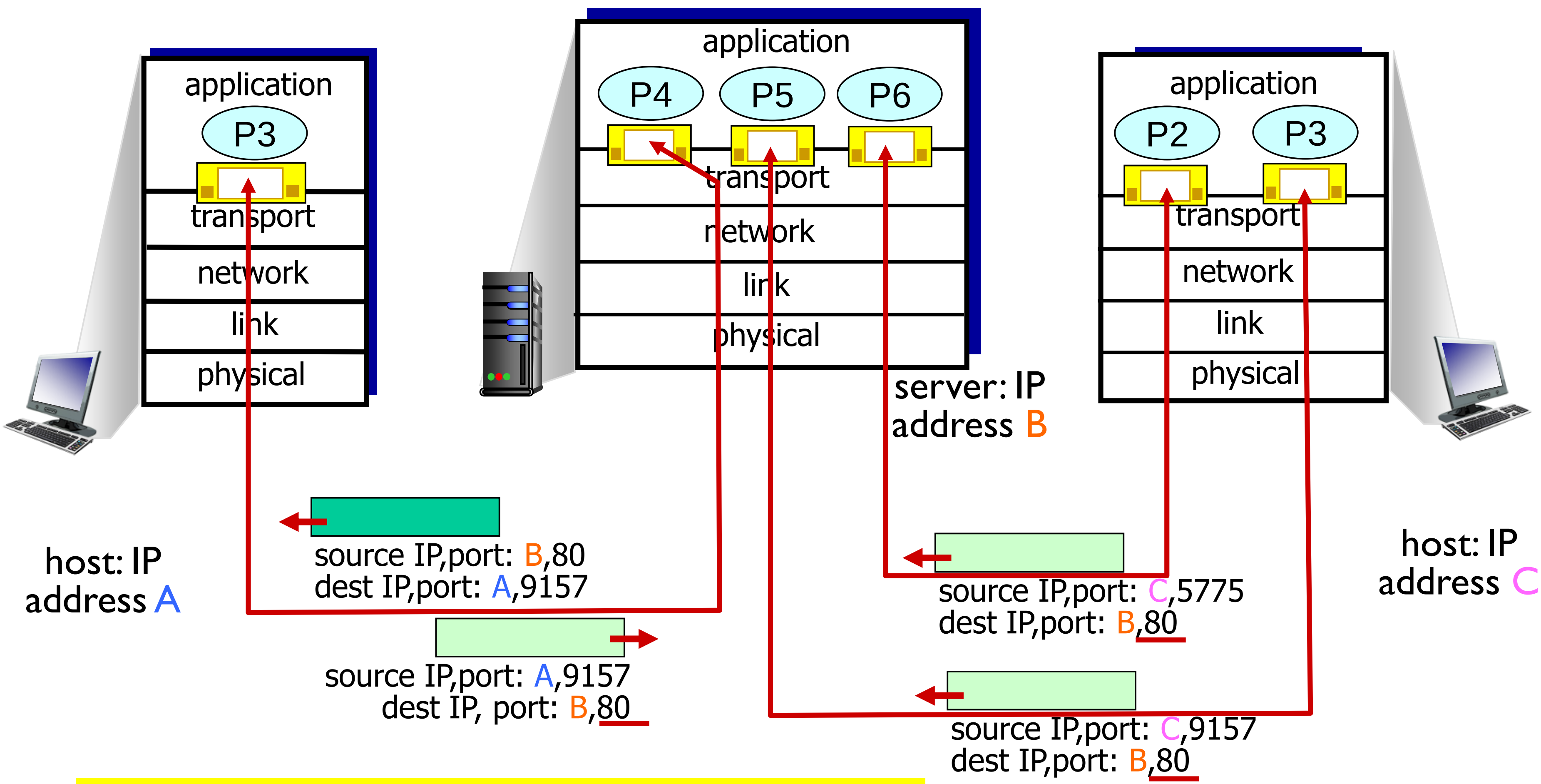
Connectionless demux: example



Connection-oriented demux

- TCP socket identified by 4-tuple (Why?):
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all **four** values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

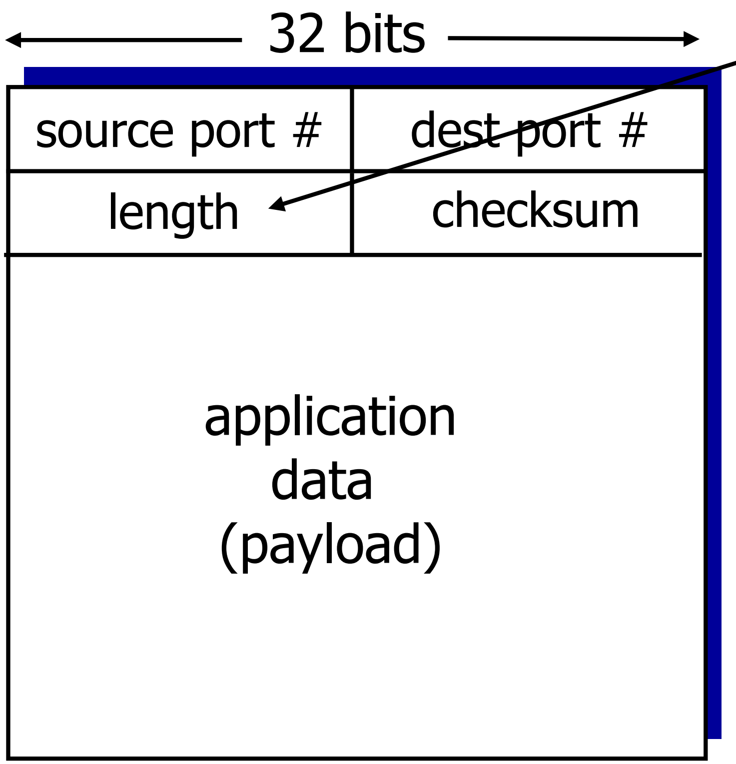
UDP

UDP: User Datagram Protocol [RFC 768]

- "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 is handled independently

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- *reliable* transfer over UDP (e.g., QUIC):
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

length, in bytes of UDP segment, including header

why is there a UDP?

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

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 ….

Internet checksum: example

example: add two 16-bit integers

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

Note: when adding numbers, an overflow bit from the most significant bit needs to be added to the result

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

Internet checksum: generation

1. initial cs = 0;
2. sum = sum all 16-bit words in segment including optional padding and initial cs;
3. cs = one's complement of sum
4. store cs in the checksum field before sending

- Note: cs is **NEVER** 0; if cs = 0 then store FFFF

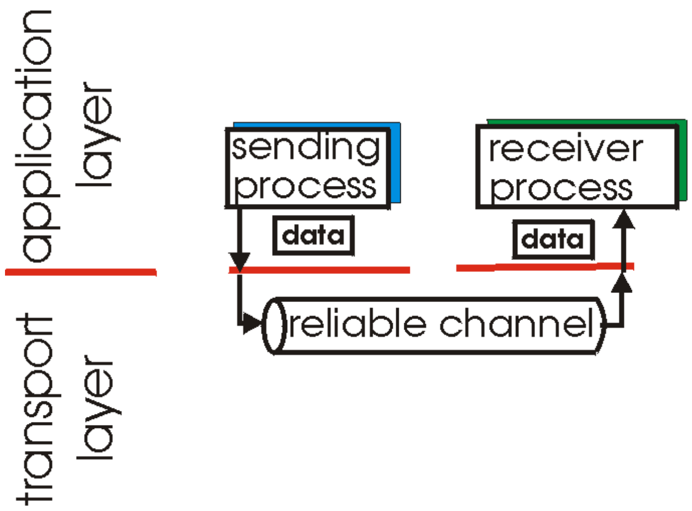
Internet checksum: validation

- sum all 16-bit words including optional padding and checksum (which is NEVER 0).
- let sum’ = one’s complement of sum
- Note:
  sum + ~sum = FFFF
  FFFF + FFFF = FFFF (add overflow bit)
  0000 = ~FFFF.
- if sum’ = 0000
  then checksum accepted
  otherwise error

Reliable Data Transfer

Principles of reliable data transfer

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

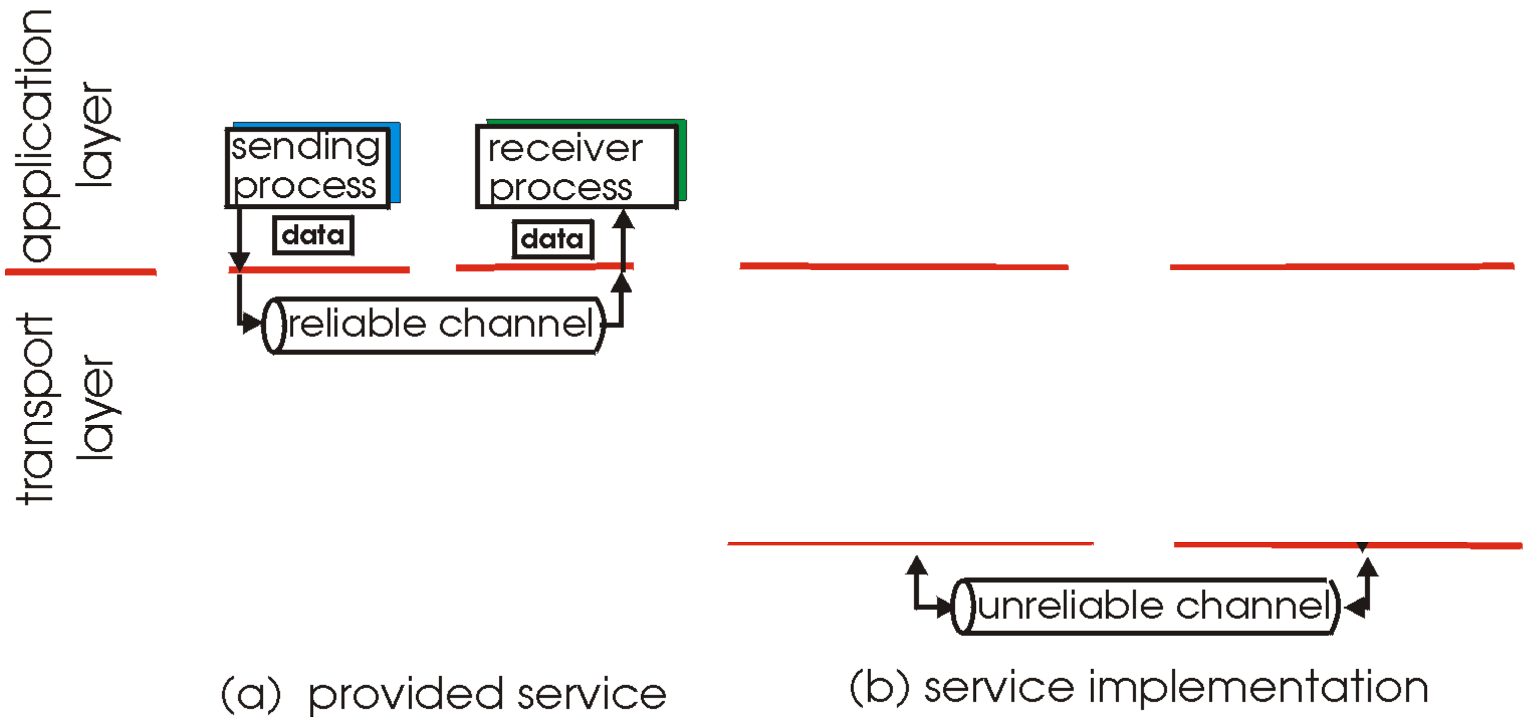


(a) provided service

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

Principles of reliable data transfer

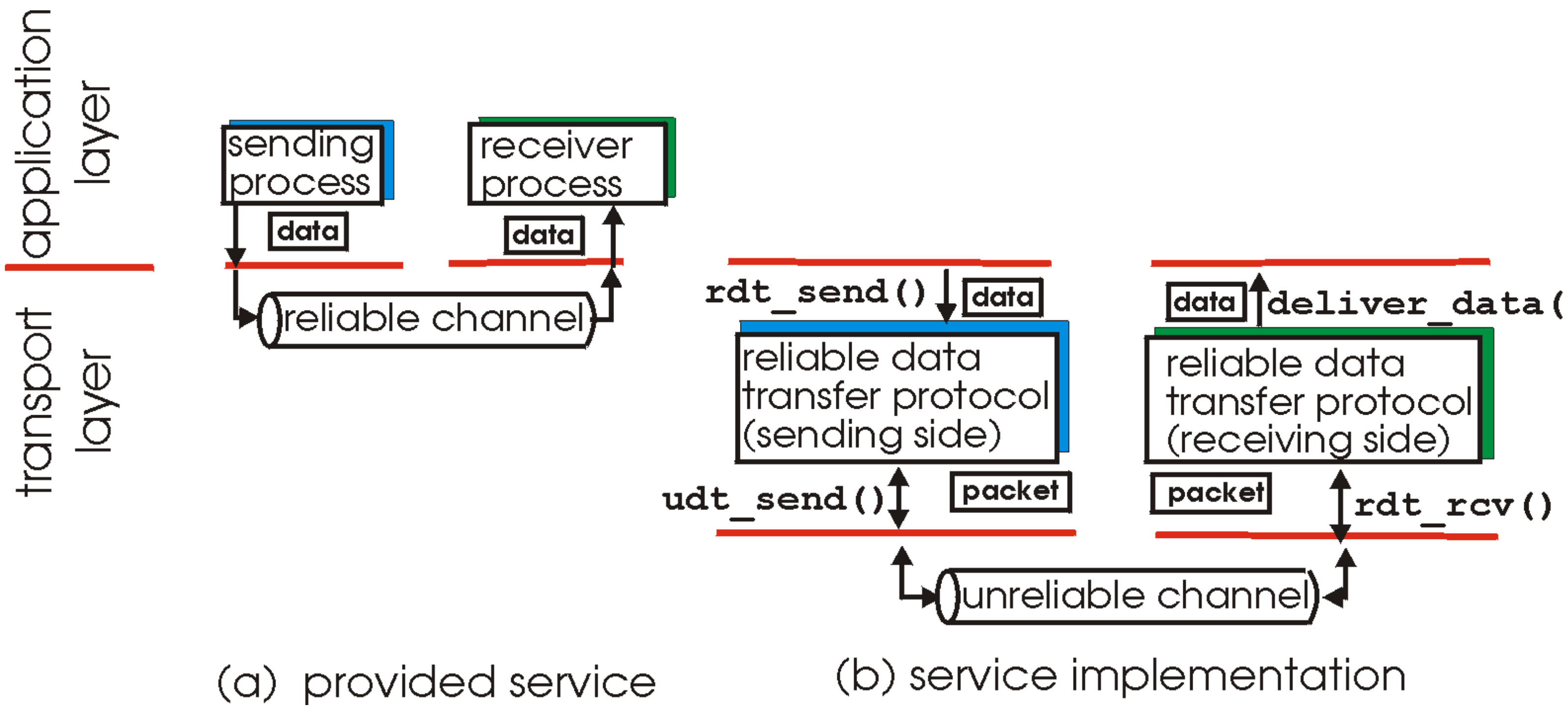
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

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

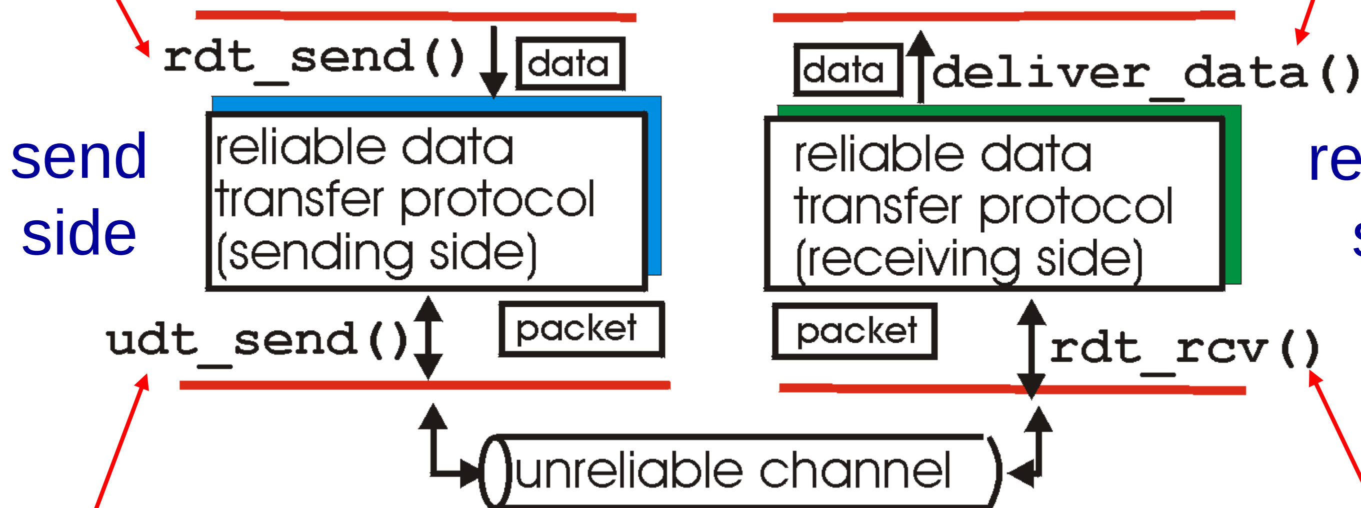


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: events

data: Passed data to deliver to receiver upper layer

deliver(data): deliver data to upper

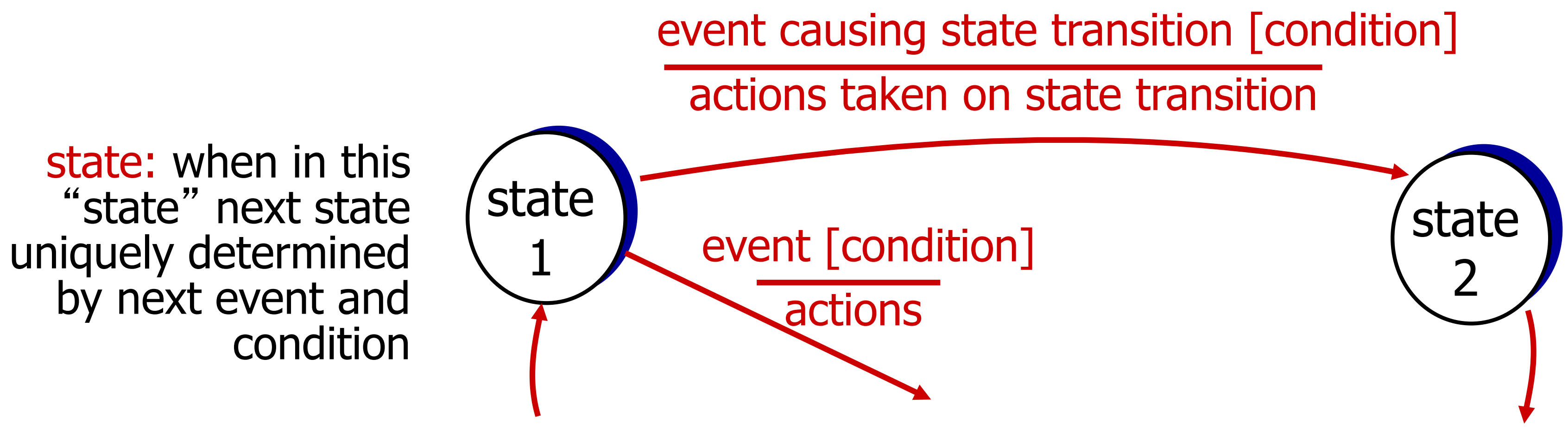


send(data/ack,#): transfer packet with # over unreliable channel

recv(data/ack,#): packet with # arrives

Reliable data transfer: getting started

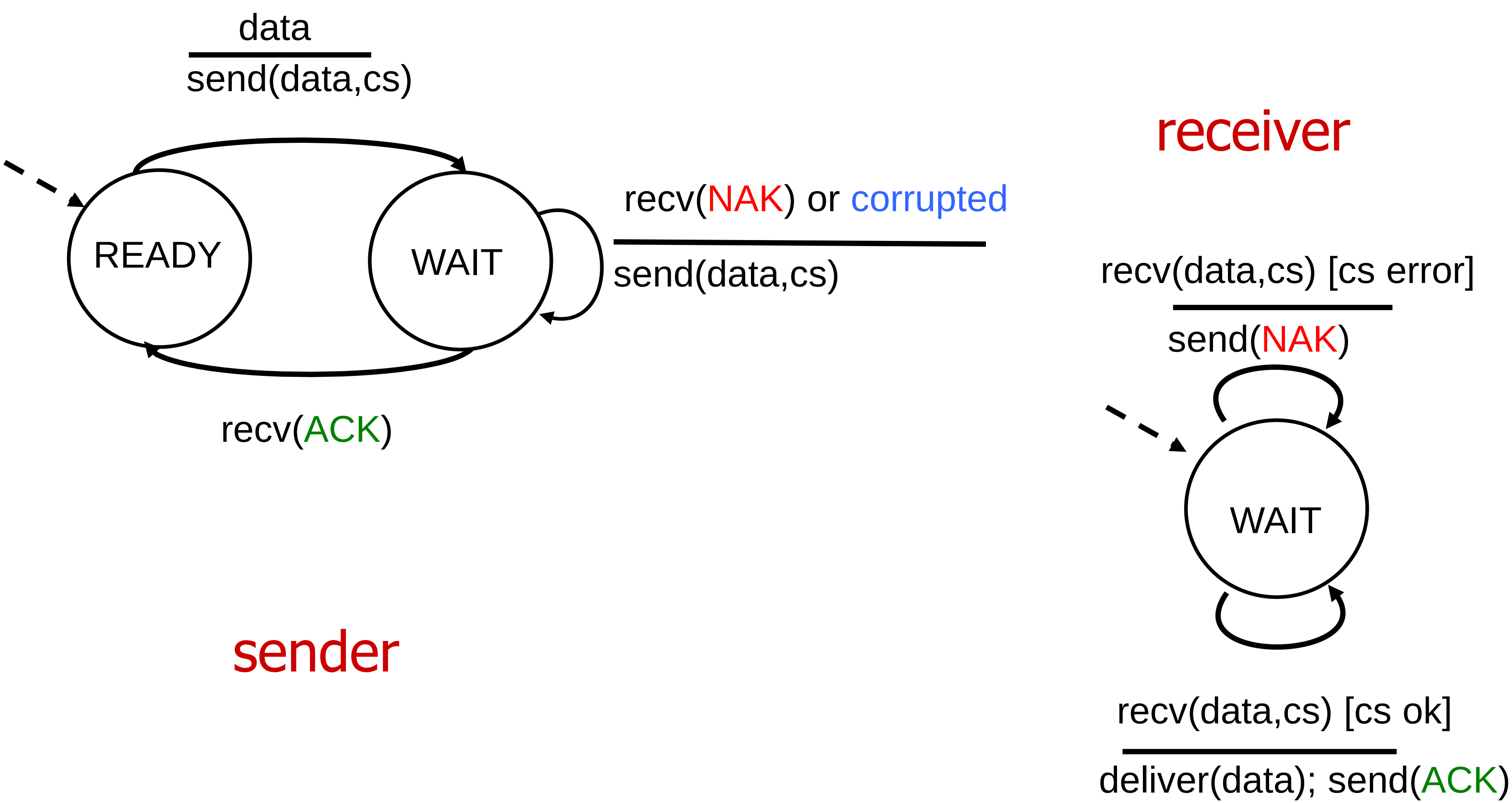
- consider only unidirectional data transfer
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver



rdt: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that packet received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - sender retransmits packet on receipt of *NAKs*
- new mechanisms:
  - error detection
  - feedback: control messages (*ACKs*, *NAKs*) from receiver to sender

rdt (1): specification



rdt (1) has a fatal flaw!

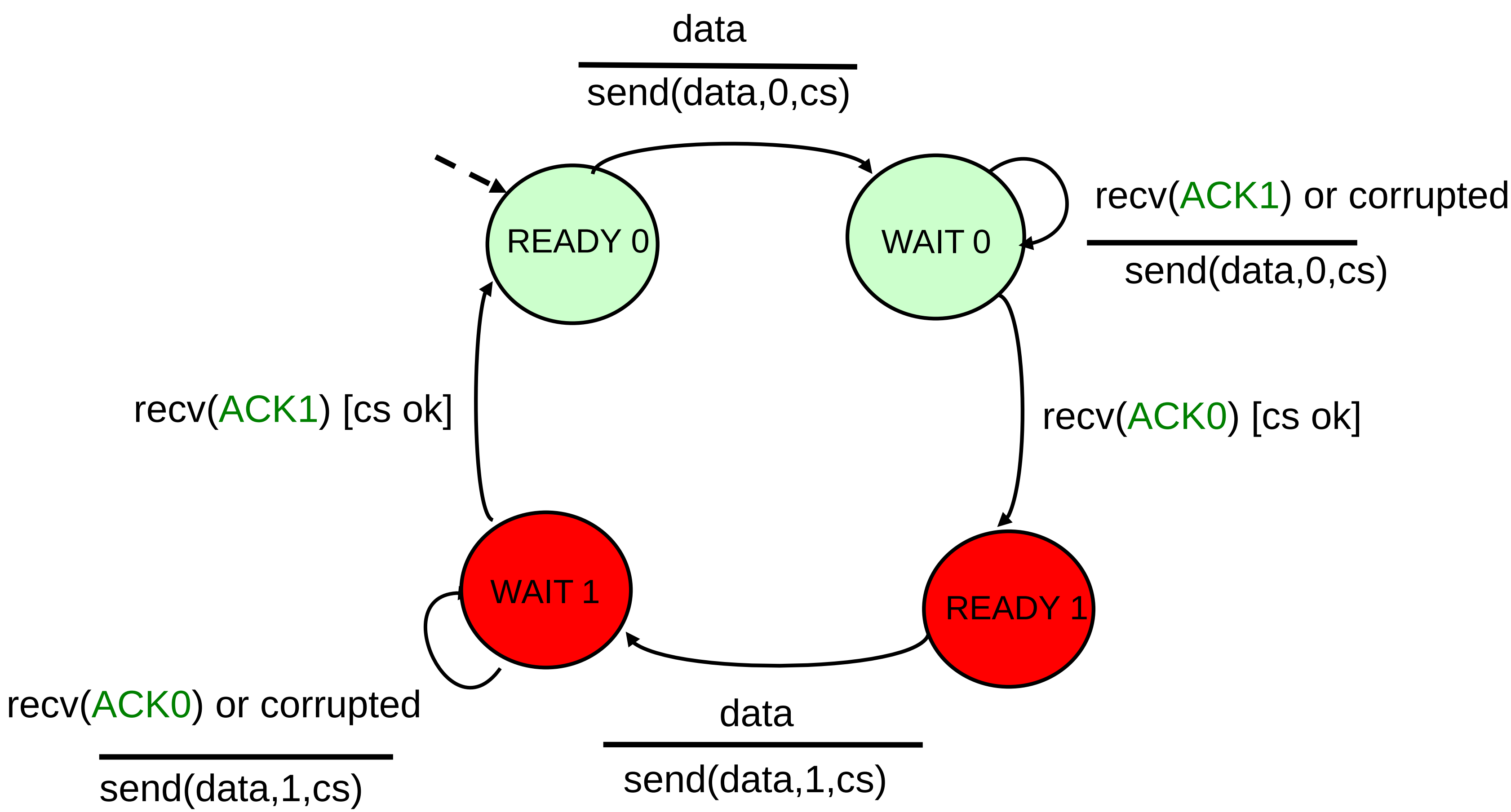
what happens if ACK/NAK corrupted or lost?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

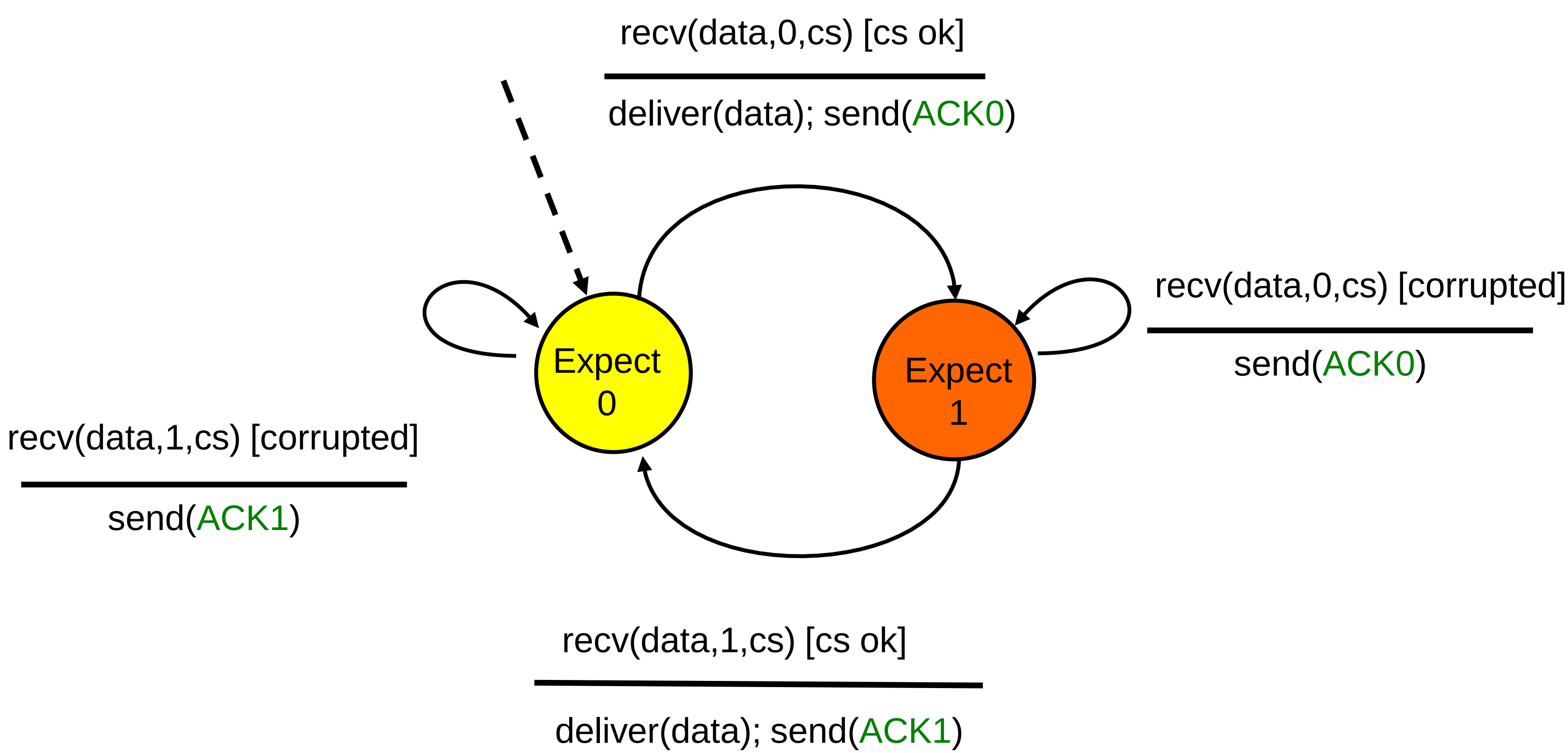
handling duplicates:

- sender retransmits current packet if ACK/NAK corrupted
- sender adds *sequence number* to each packet
- receiver discards (doesn't deliver up) duplicates

rdt (2): ACK with sequence # (sender)



rdt (2): ACK with sequence # (receiver)

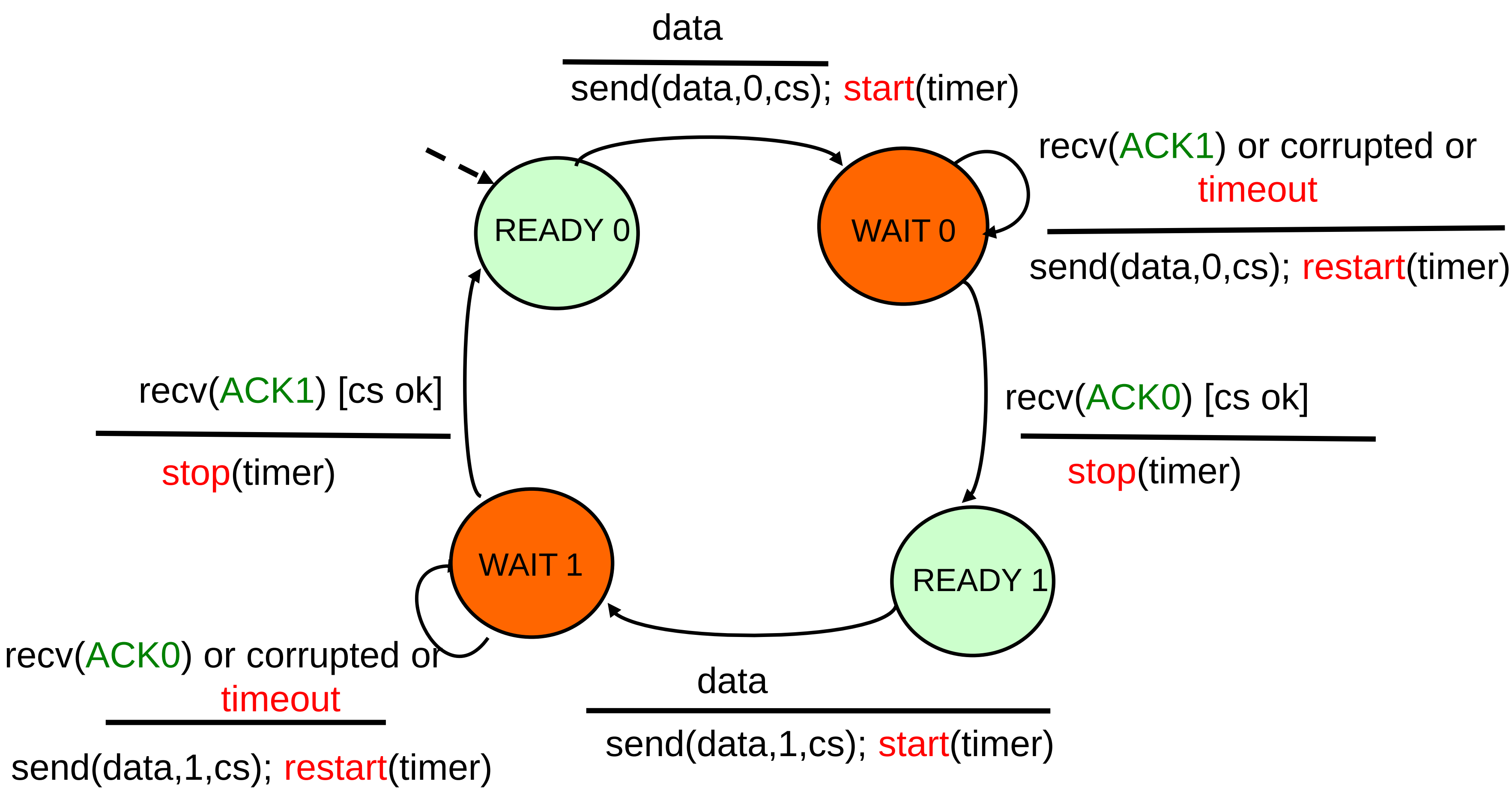


rdt (3): error, duplicate, and loss

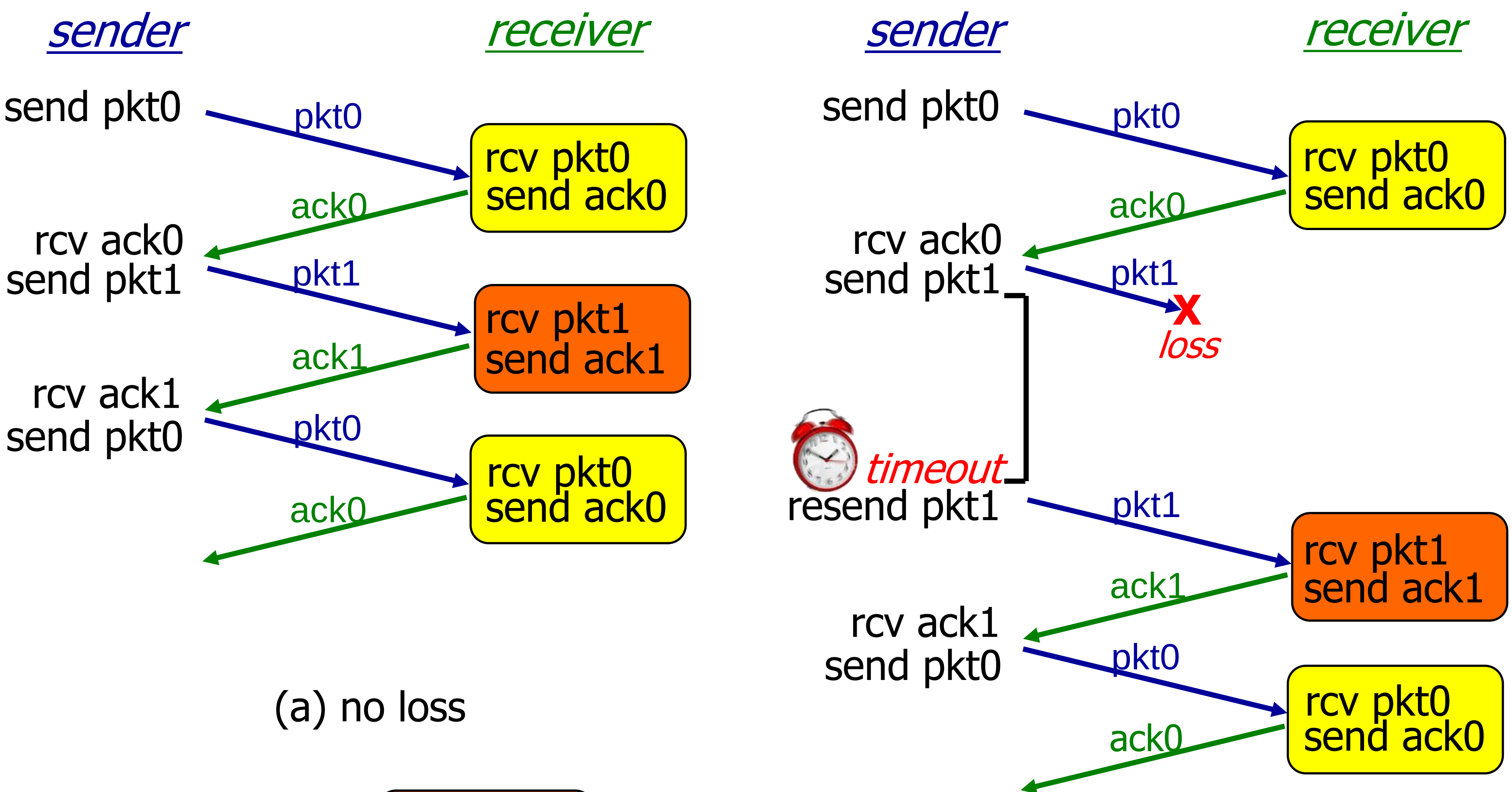
sender waits "reasonable" amount of time for ACK:

- retransmits if **timeout**
- retransmission will be **duplicated** if packet is delayed instead of lost
- the received ACK# must **match**, what is **expected** (i.e., ACK# means ``receiver seen packet #"); retransmit if not
- known as "**Alternating Bit Protocol**" (ABP).

rdt (3): ACK, sequence #, timer (sender)



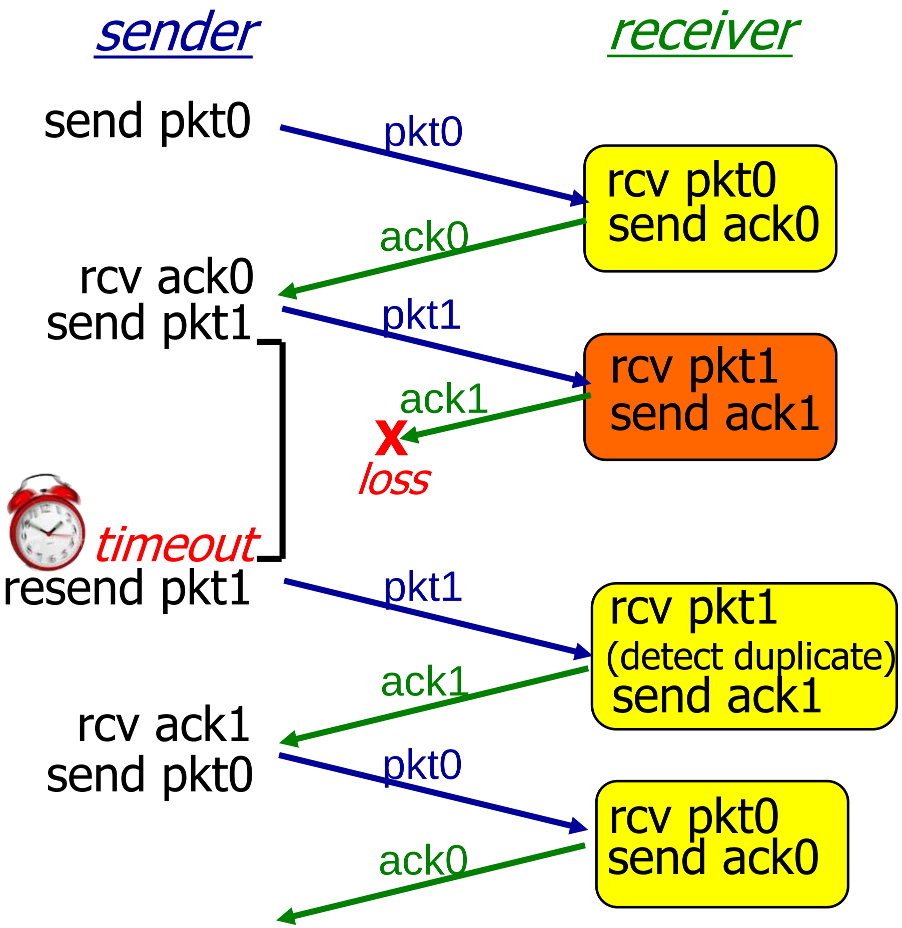
rdt (3) in action



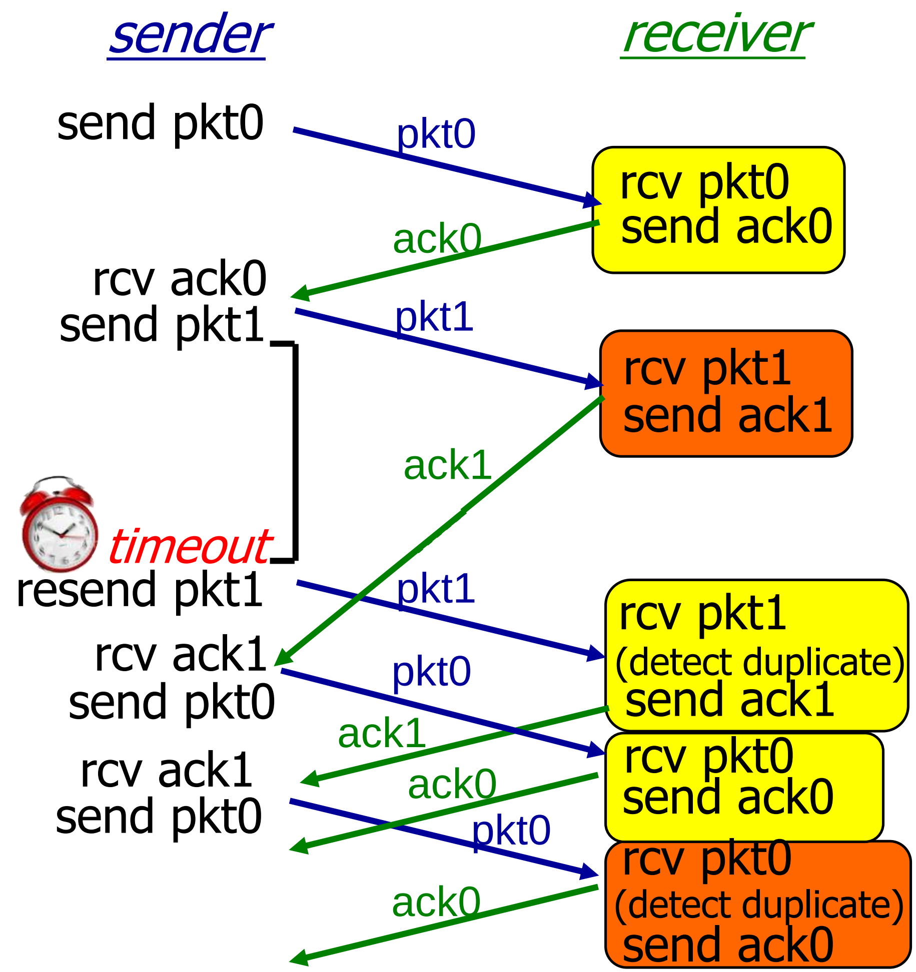
(a) no loss

(b) packet loss

rdt (3) in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt (3)

- rdt (3) is **correct**, but **performance stinks!**
- e.g.: 1 Gbps link, 30 ms RTT delay, 8000 bit packet:

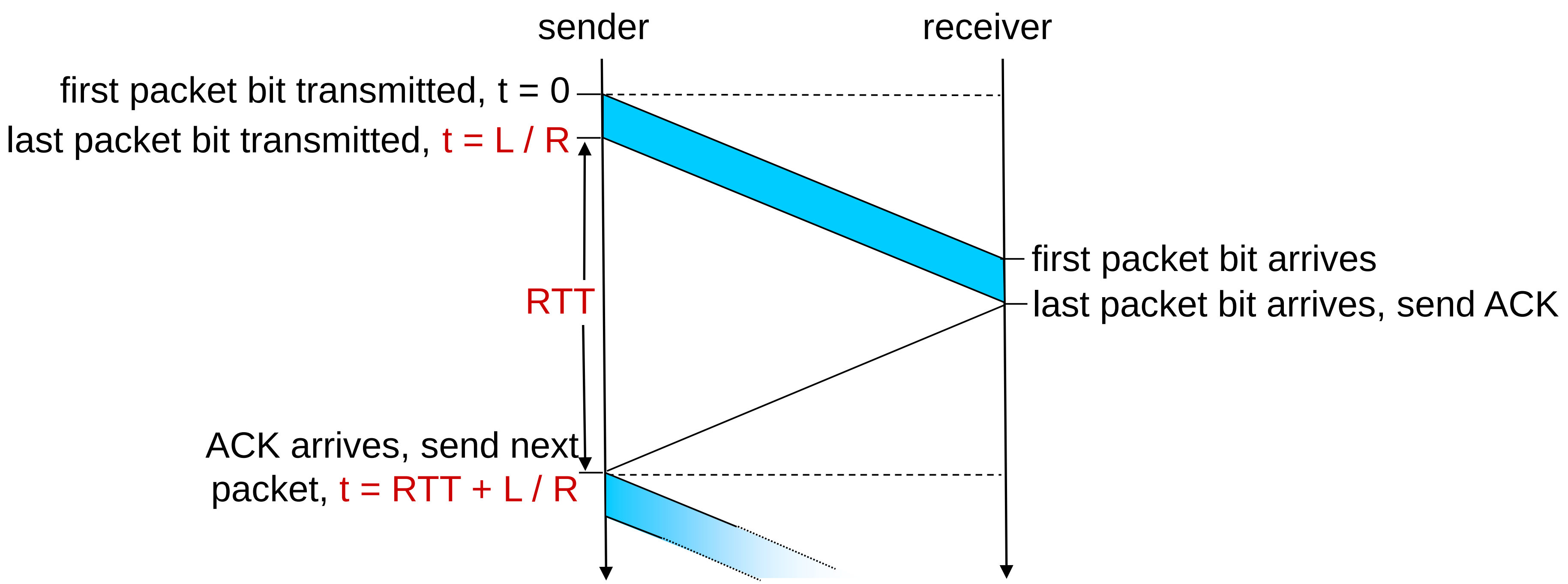
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U_{sender}: *utilization* – fraction of time sender busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

  - if RTT=30 msec, 1KB packet every 30 msec: 33kB/sec throughtput over 1 Gbps link
- network protocol limits use of physical resources!

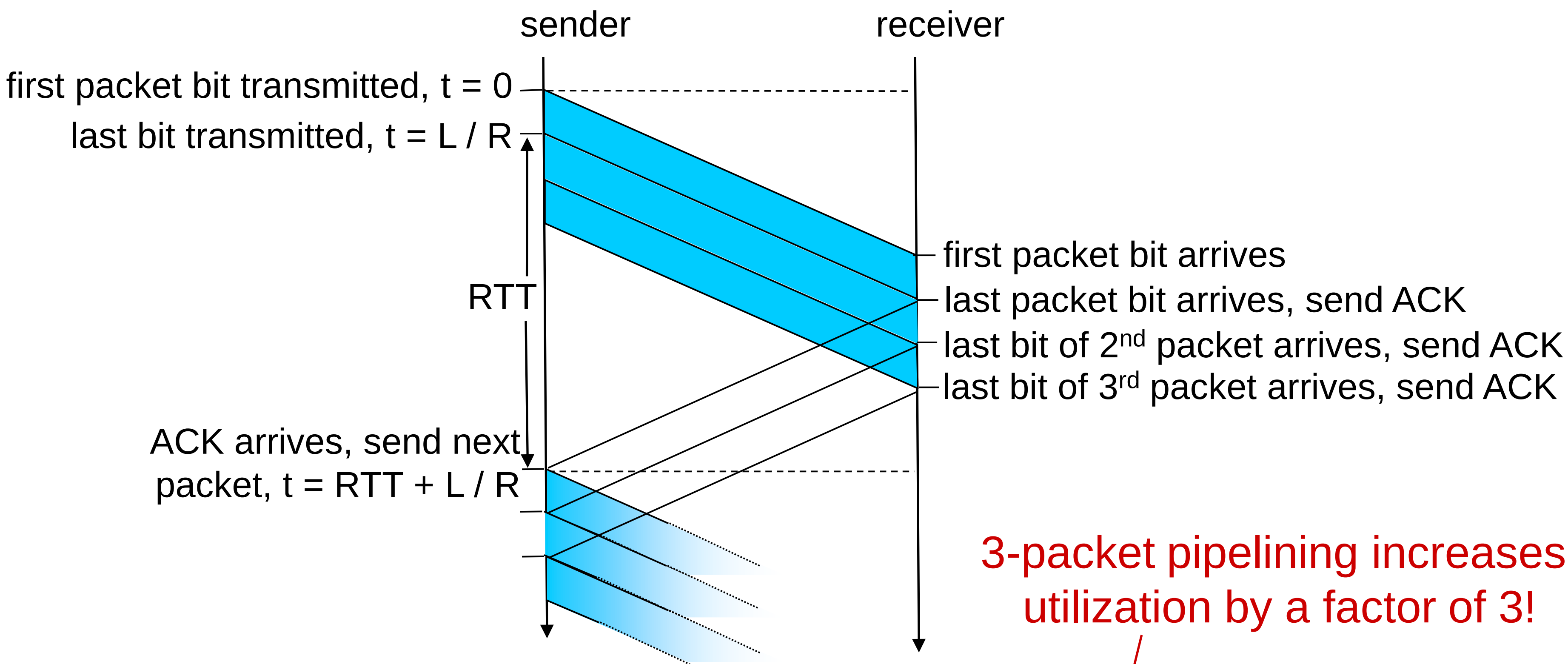
rdt (3): stop-and-wait operation



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Sliding Window Protocols

Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

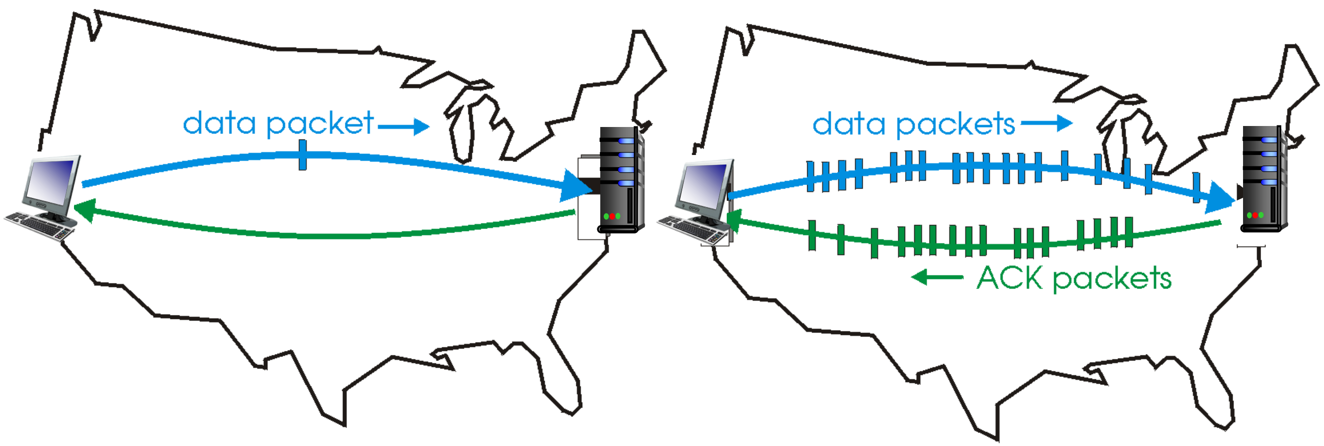
Pipeline capacity

- It is defined as **bandwidth-delay** product.
- For example, bandwidth = 1 Mbps (1 million bits/sec), and RTT delay = 1 second.
- Then, the pipeline **capacity** is 1 Mbps x 1 second = 1 million bits.
- The **higher** the bandwidth-delay product; the **larger** the pipeline capacity.
- However, network "capacity" varies over time, due to congestion and packet loss.

Pipelined protocols

pipelining: sender allows **multiple**, "**in-flight**", **yet-to-be-acknowledged** packets

- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Sliding window protocols

Go-back-N:

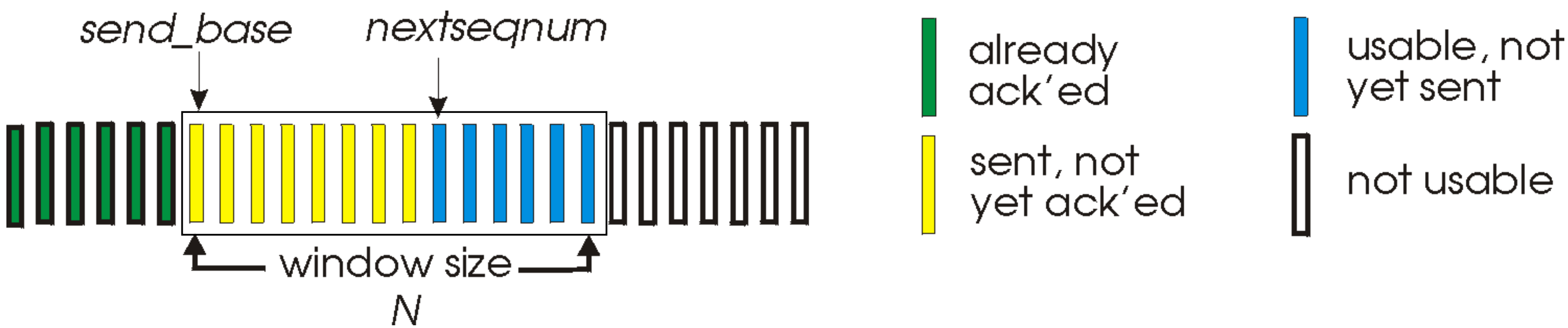
- sender can have up to N unack'ed packets in pipeline
- receiver only sends *cumulative ack*, what it has seen so far
- sender has timer for **oldest** unack'ed packet
  - when timer expires, retransmit ***all*** unacked packets

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- receiver sends *individual ack* for each packet
- sender maintains a timer for **each** unack'ed packet
  - when a timer expires, retransmit **only** that unack'ed packet

Go-Back-N: sender

- assign a k-bit seq # in packet header ($2^k > N$)
- "window" of up to N, consecutive unack'ed packets allowed

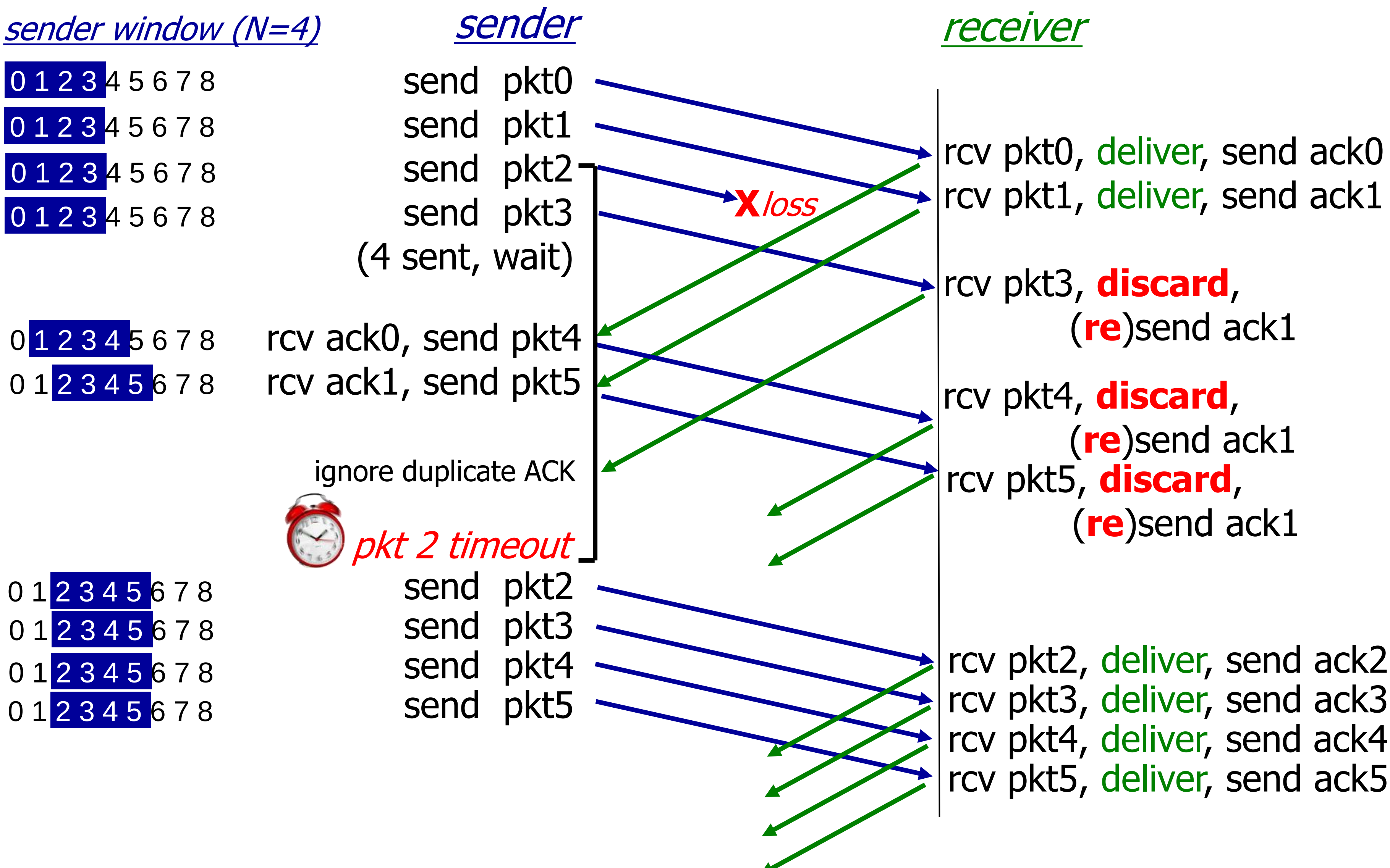


- ACK(n): all packets up to and including seq #n received *"cumulative ACK"*
  - sender may receive **duplicate** ACKs (see receiver)
- a timer for the **oldest** (earliest) unack'ed packet
- *timeout(n)*: retransmit **all** unack'ed packets in window

Go-Back-N: receiver

- Use the same sequence #s as the sender
- But, it has a window of **size 1**
- The window specifies the next **expected** packet sequence #.
- **Cumulative ACK**s: ACK(n) if all packets up to and including sequence #n have been received.
- Out of order packets are discarded!

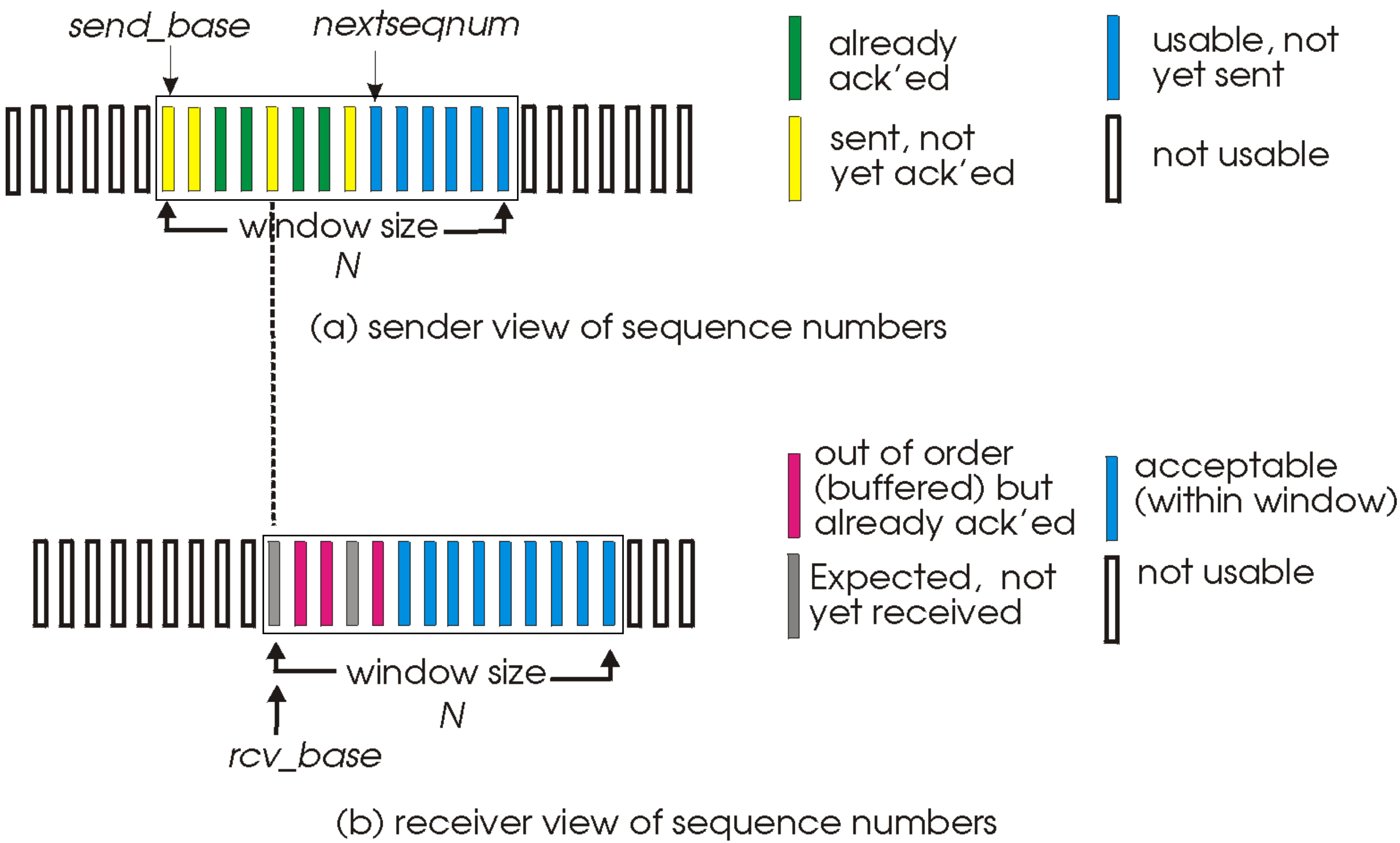
GBN in action



Selective repeat

- receiver ***individually*** acknowledges all ***correctly*** received packets within window
  - buffer packets, as needed, for eventual in-order delivery to upper layer
- sender only resends packets for which ACK ***not*** received (i.e., timeout)
  - sender set timer for each unACK'ed packet
- sender window
  - windows size *N* <= **half** of max. sequence #
  - limits seq #s of sent, unACK'ed packets
  - advance window when consecutive packets have been ACK'ed

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in window:

- mark pkt n as received
- If m smallest unACKed pkt #, advance window base to m

receiver

pkt n in window:

- send ACK(n)
- out-of-order: buffer
- in-order: deliver all in-order consecutive pkts, advance window to next not-yet-received pkt

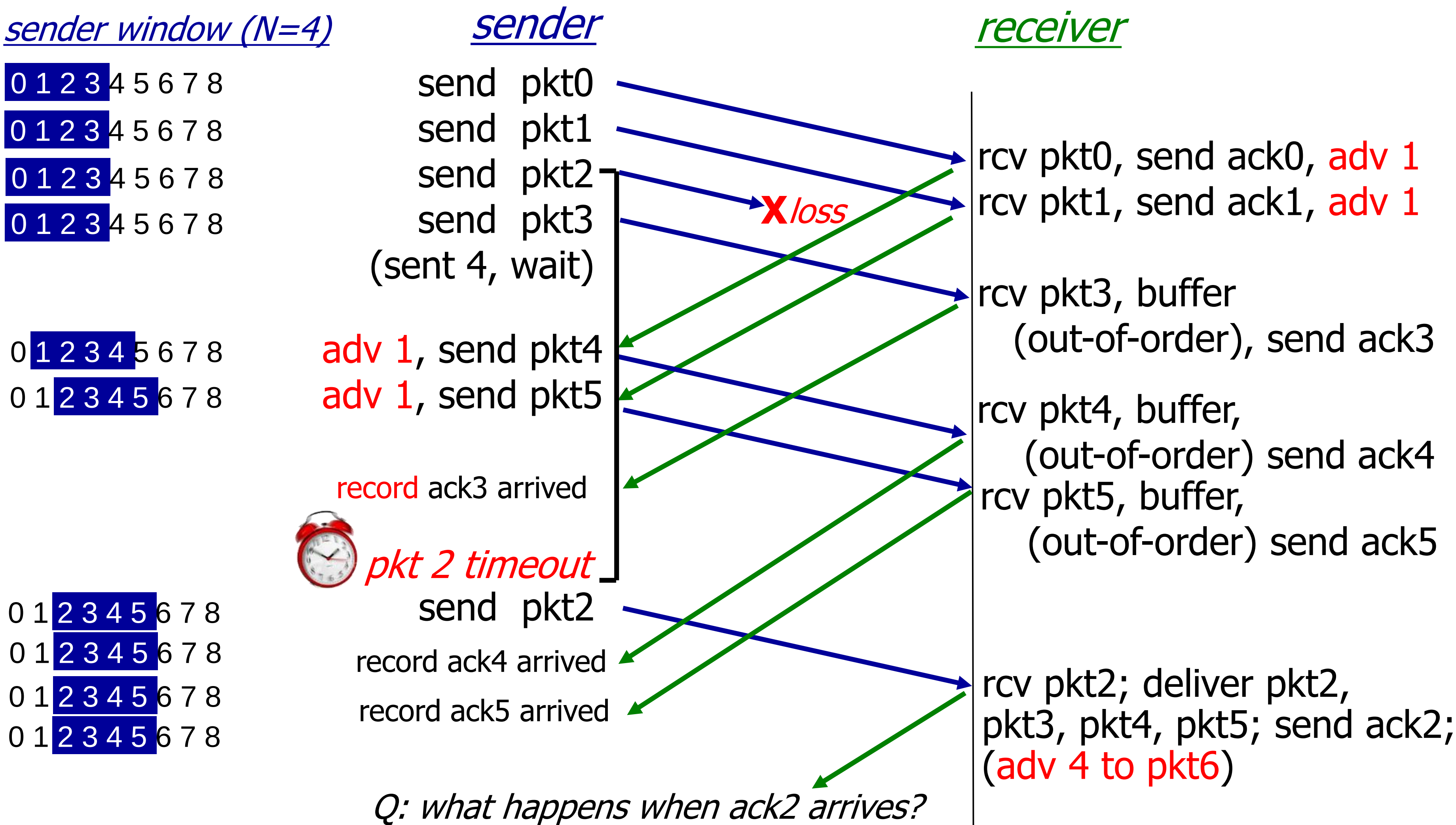
pkt n was seen:

- ACK(n)

otherwise:

- ignore

Selective repeat in action



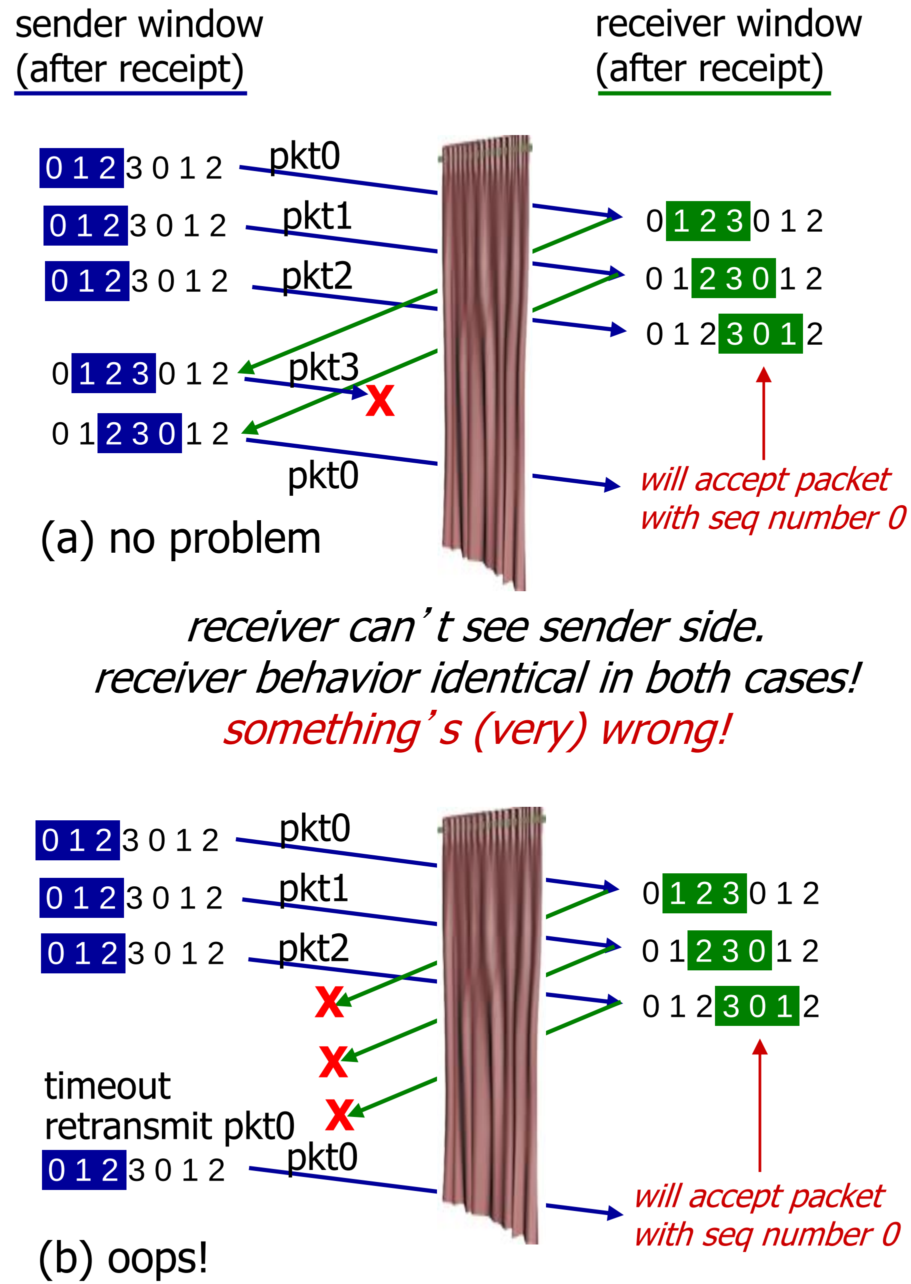
Selective repeat: dilemma

example:

- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size N to avoid problem in (b)?

N <= half of seq #

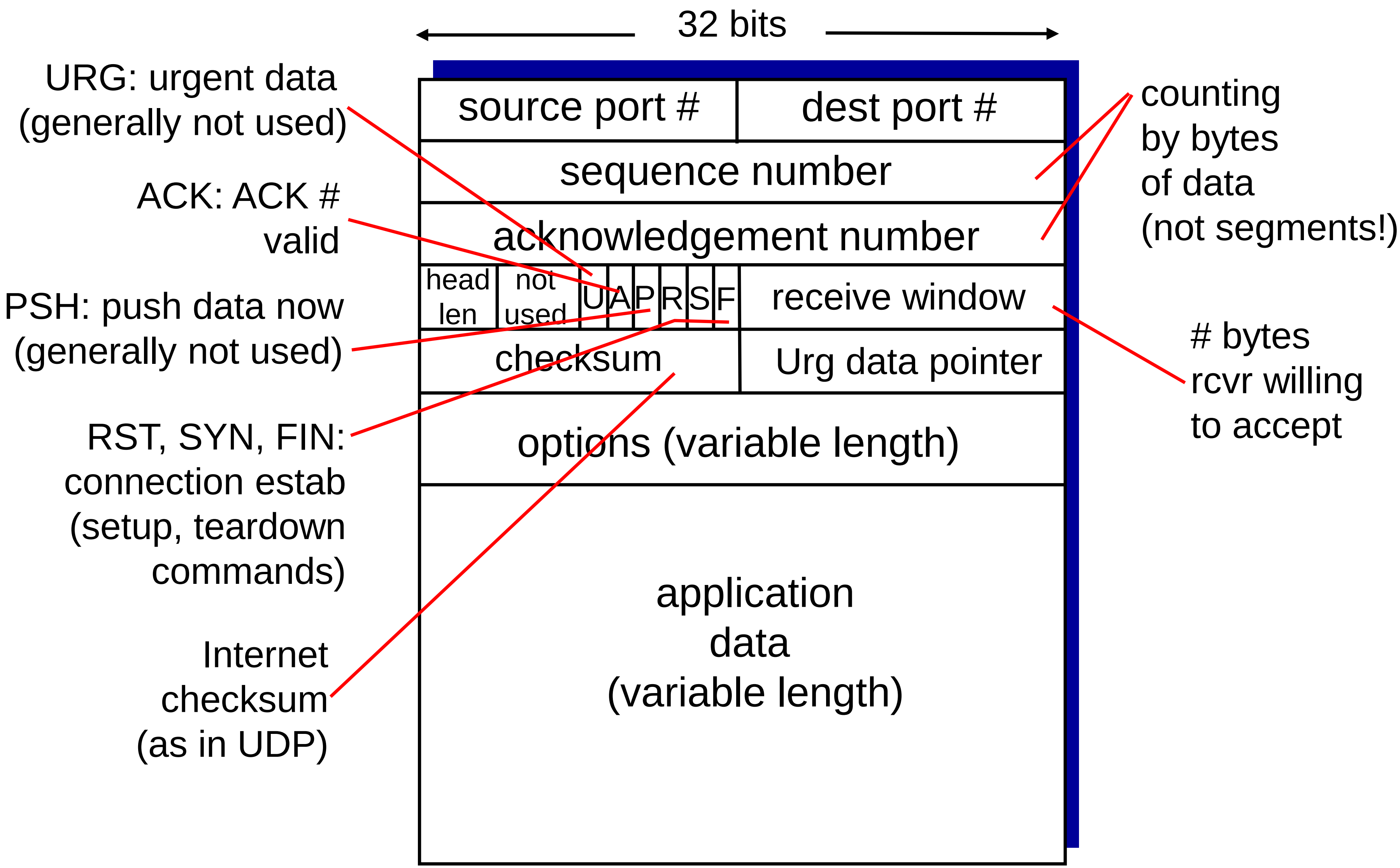


TCP

TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size automatically and dynamically
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size (< MTU)
- connection-oriented:
  - handshaking (3-way) agree on sender's, and receiver's initial sequence # and window sizes

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

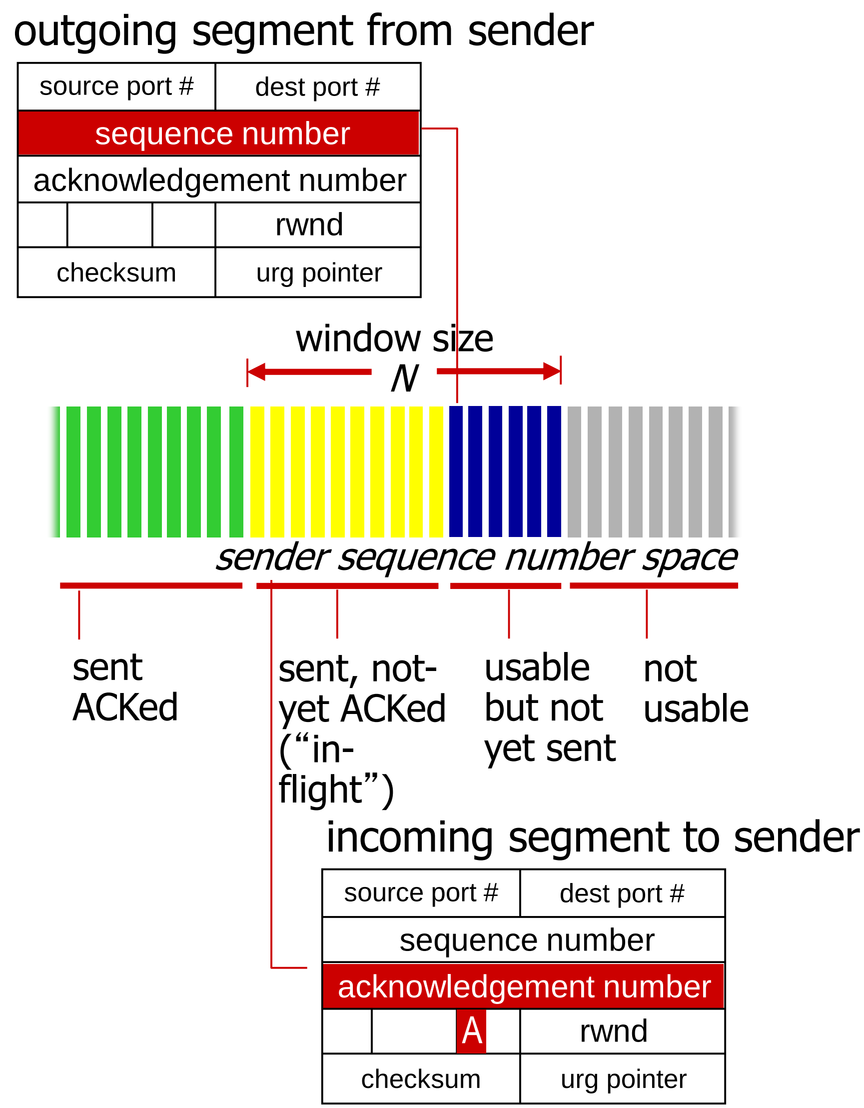
- byte stream "number" of first byte in segment's data

acknowledgements:

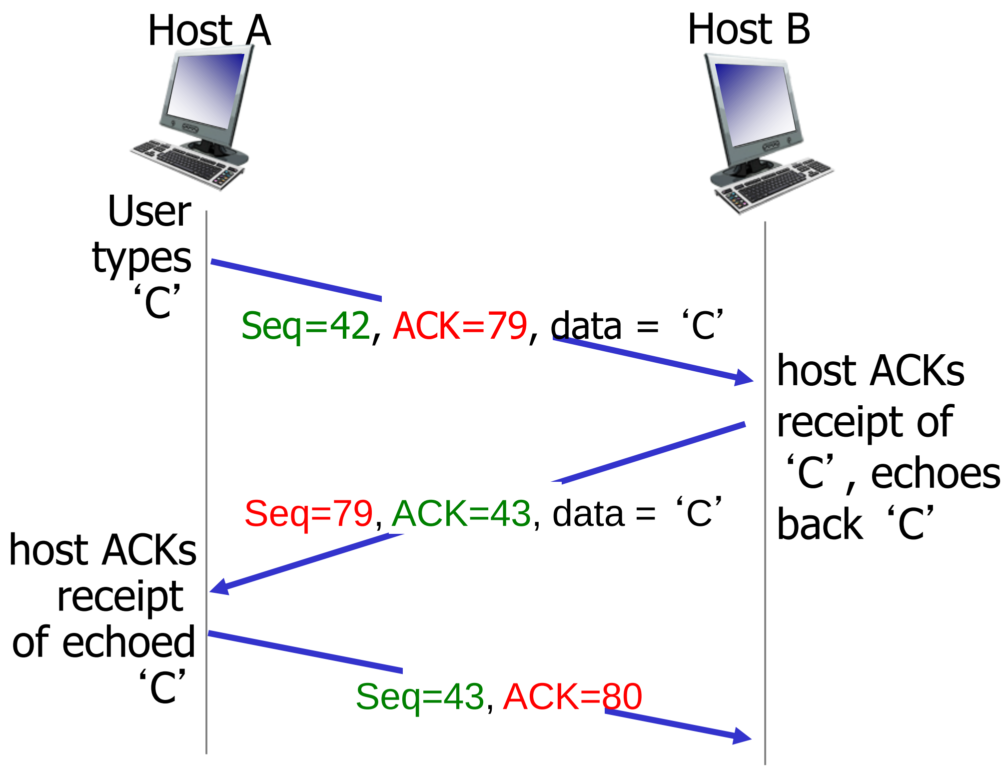
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP seq. 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

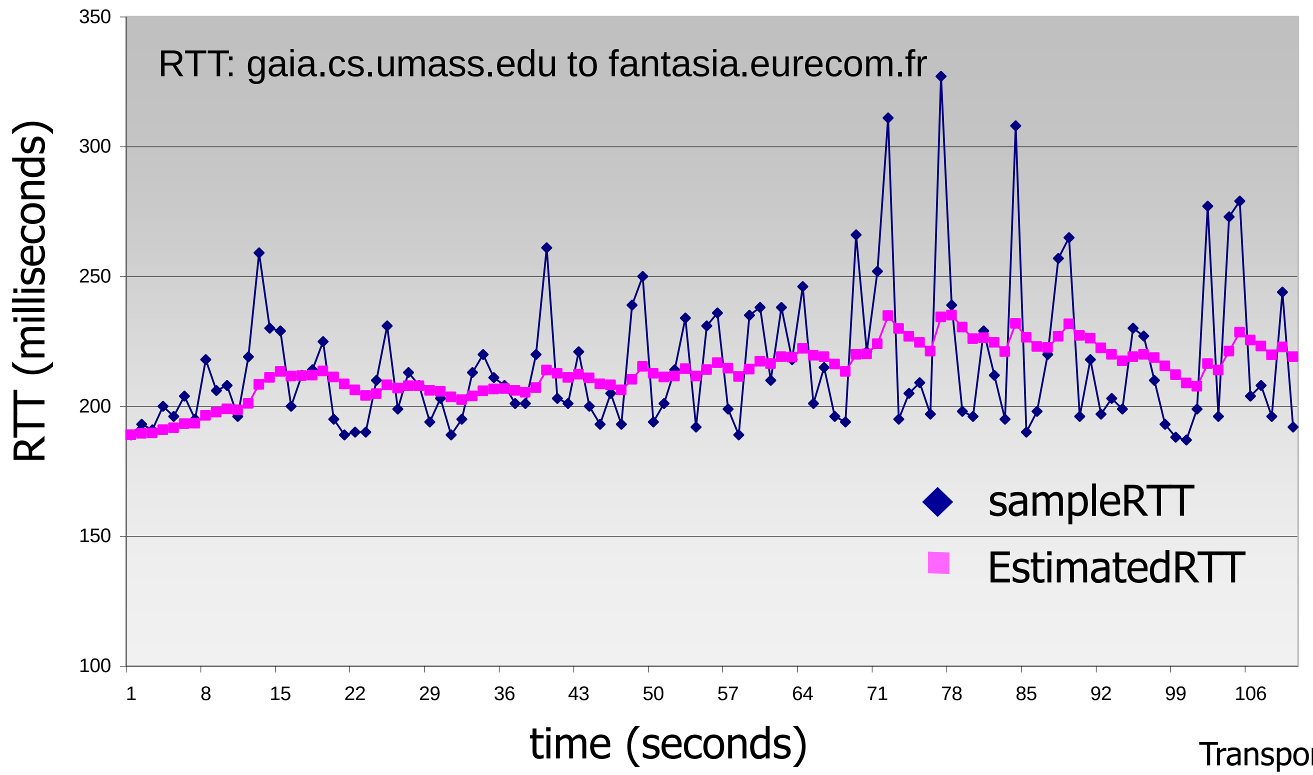
Q: how to estimate RTT?

- **`SampleRTT`**: *measured time from segment transmission until ACK receipt*
  - ignore retransmissions
- **`SampleRTT`** will vary, use *exponential moving average*
  - **`estimatedRTT`** : average several *recent* measurements, not just current **`SampleRTT`**

TCP round trip time, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$ (less sensitive to **SampleRTT**)



TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "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 data transfer

- TCP creates reliable data service on top of IP's unreliable service
  - pipelined (sliding window) segments
  - cumulative ACKs
  - single retransmission timer

- retransmissions triggered by:
  - timeout events
  - 3 duplicate ACKs

TCP sender events:

data (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 for **oldest** unACK'ed segment
  - expiration interval: **`TimeOutInterval`**

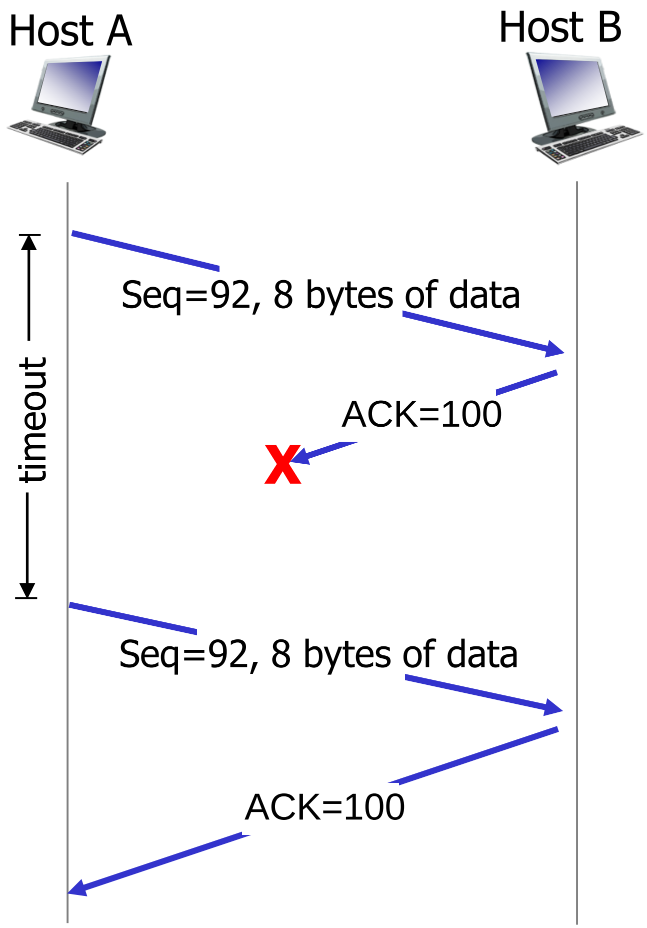
Timeout (internal):

- retransmit segment that caused timeout
- restart timer

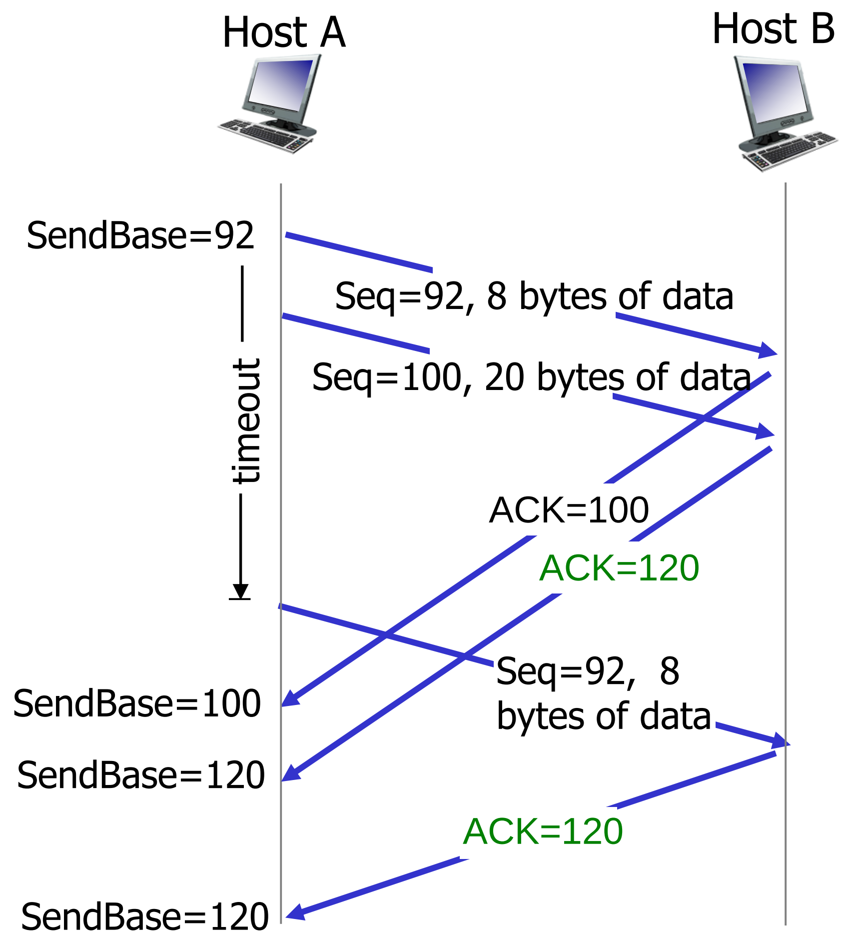
ack (from receiver):

- if ACK acknowledges previously unacked segments
  - update what is known to be ACK'ed
  - start timer if there are still unACK'ed segments

TCP: retransmission scenarios

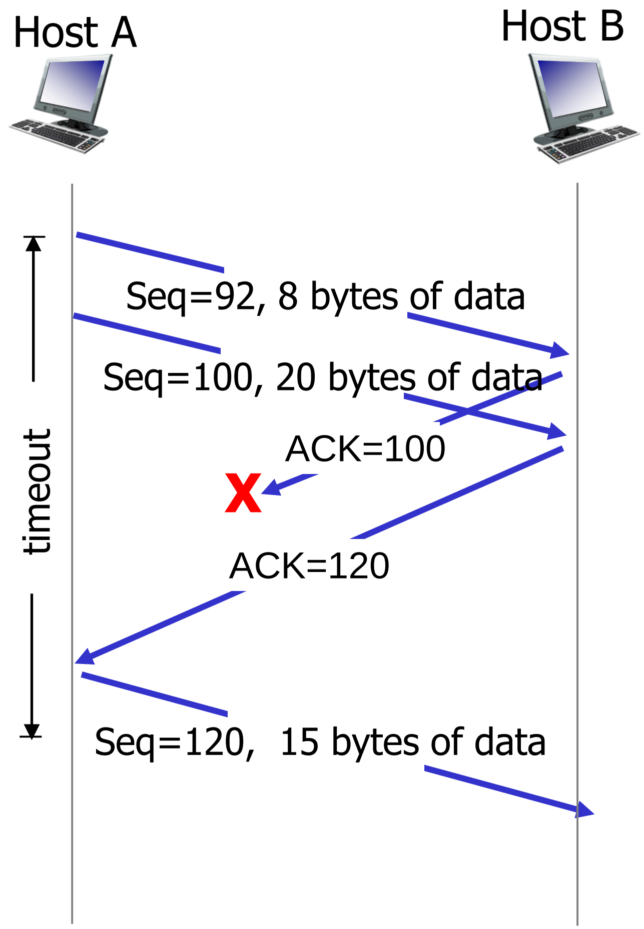


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP fast retransmit

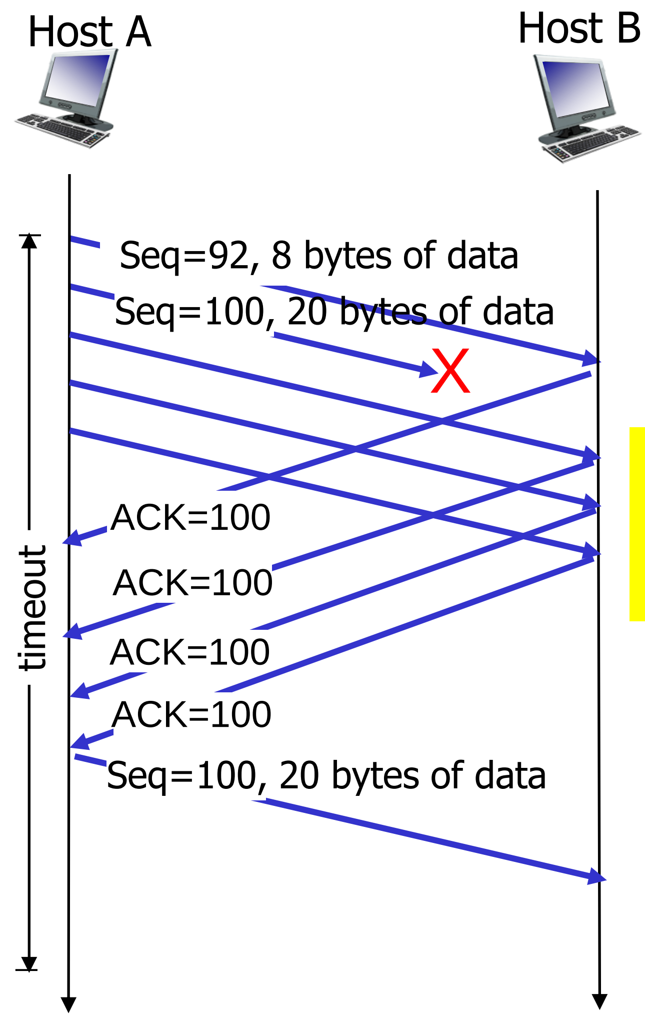
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data, resend unACK'ed segment with smallest seq #

- likely that unACK'ed segment is lost, so don't wait for timeout

TCP fast retransmit



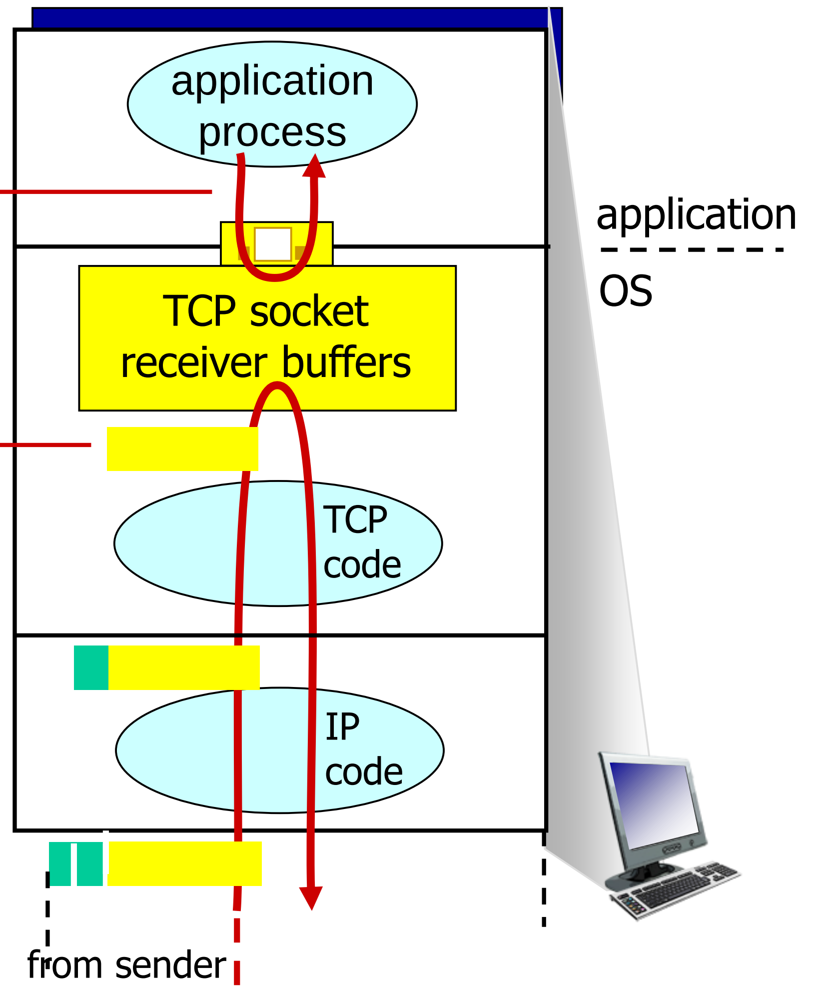
fast retransmit after sender receipt of triple duplicate ACKs (total 4)

TCP Flow Control

TCP flow control

application may remove data from TCP socket buffers ….

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



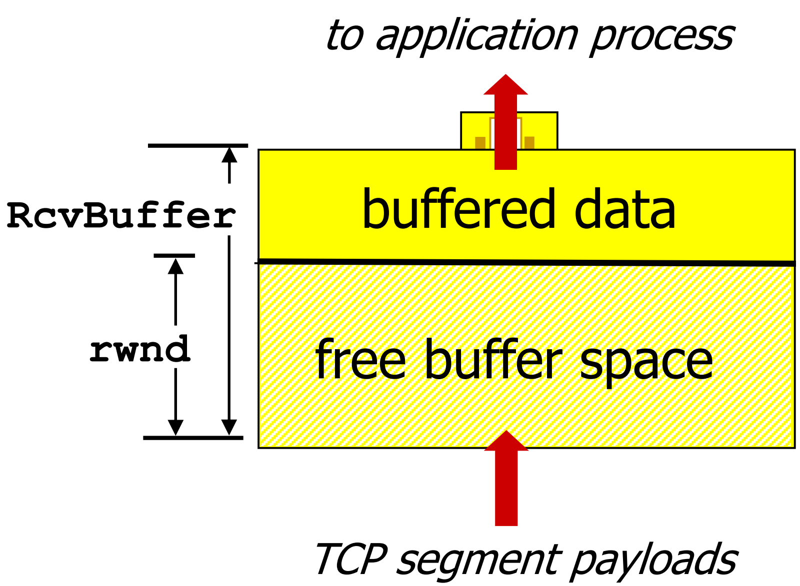
receiver protocol stack

flow control

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

TCP 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 unACK'ed data to receiver's **`rwnd`** value
- sender **stops** sending if **`rwnd`** == 0.

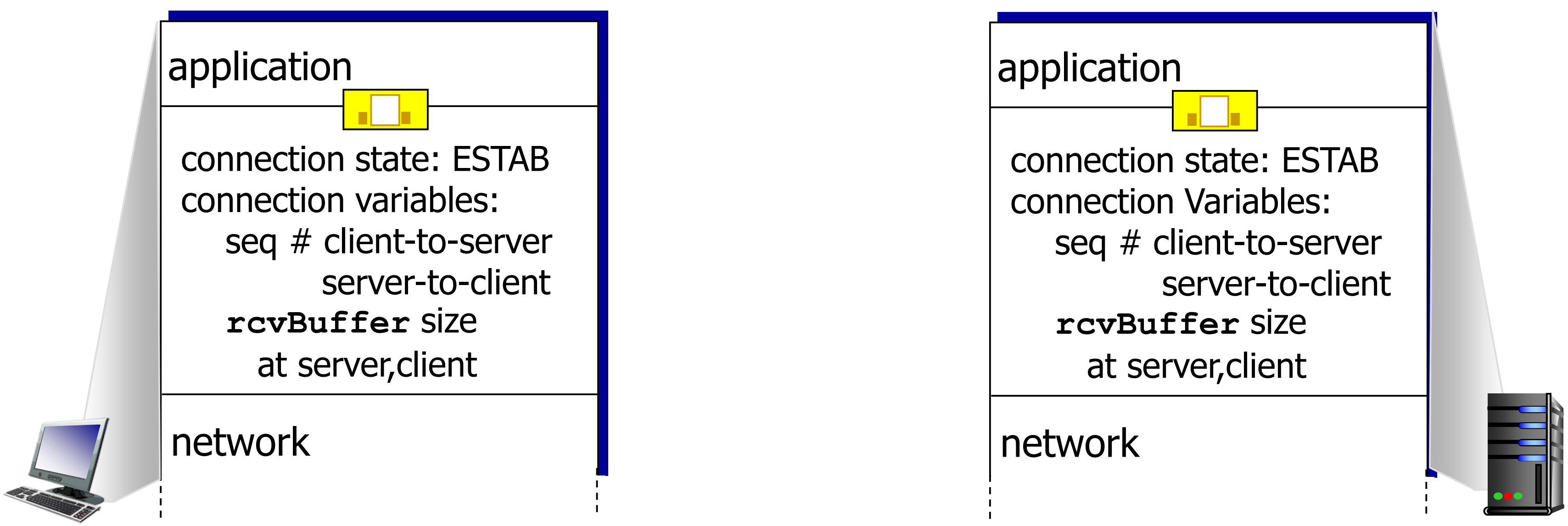


receiver-side buffering

Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on **connection parameters (initial seq. #s, rcvBuffer size)**

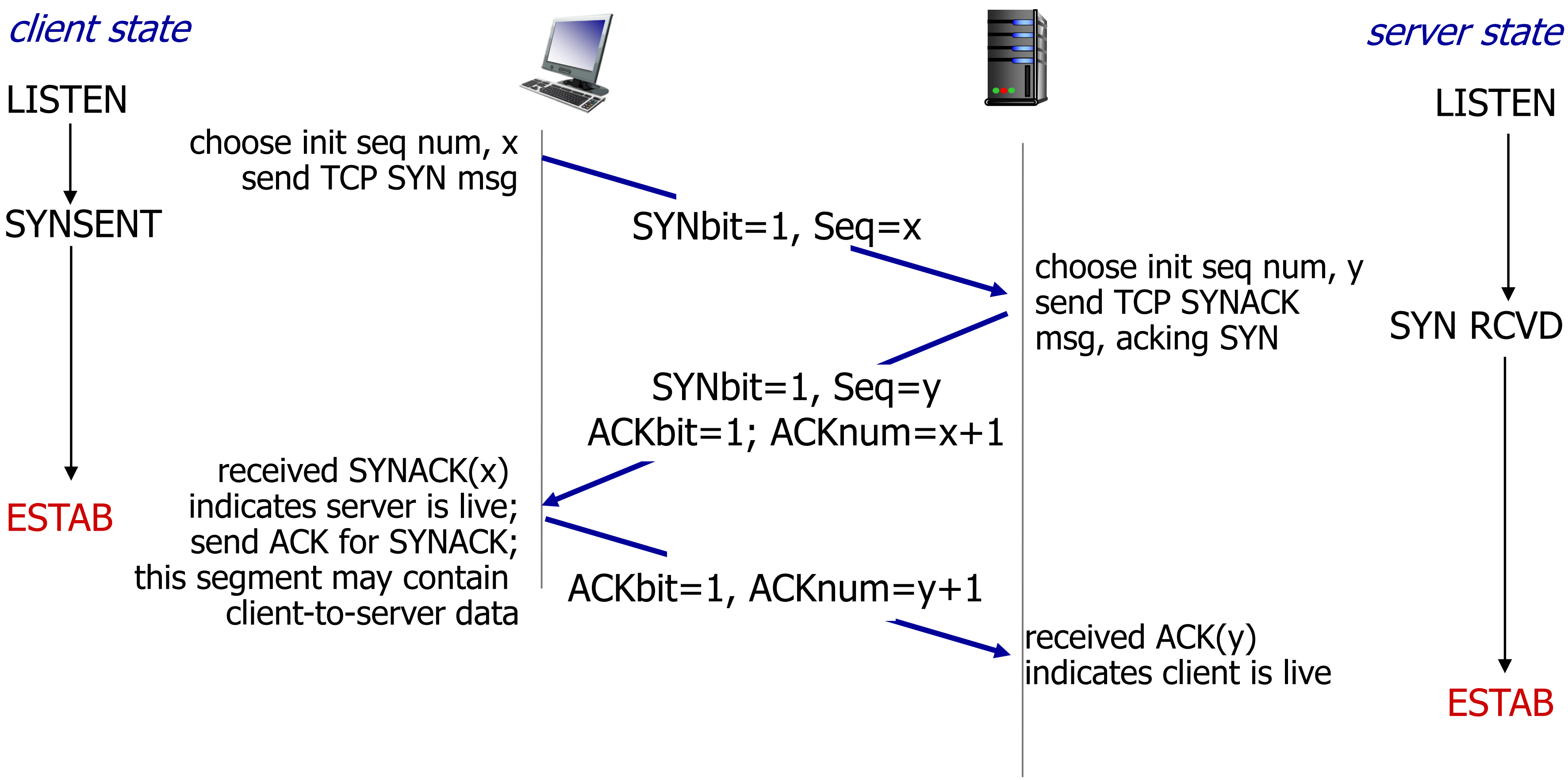



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

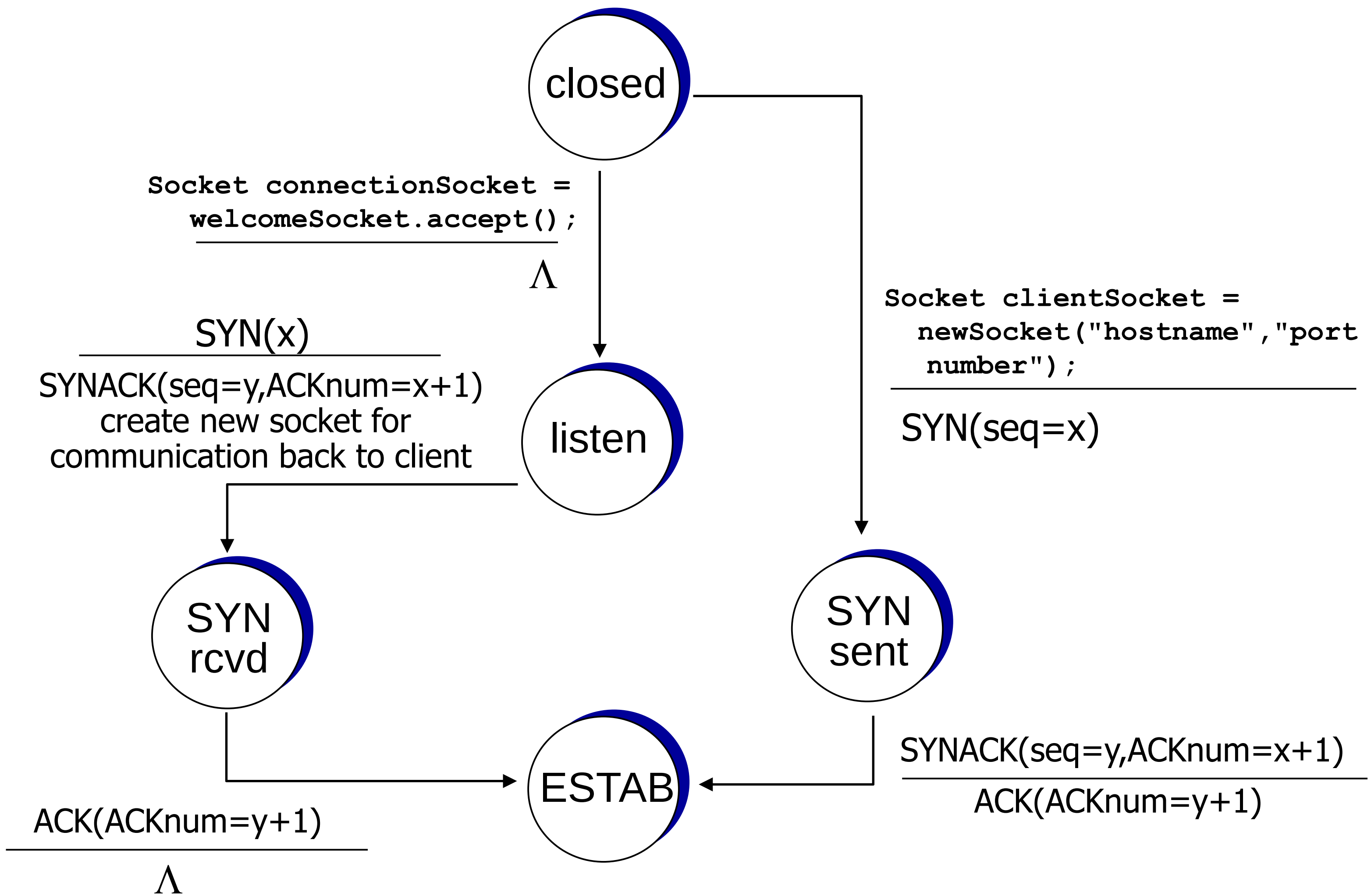
```
Socket connectionSocket =
   welcomeSocket.accept();
```

TCP Connection Setup

TCP 3-way handshake



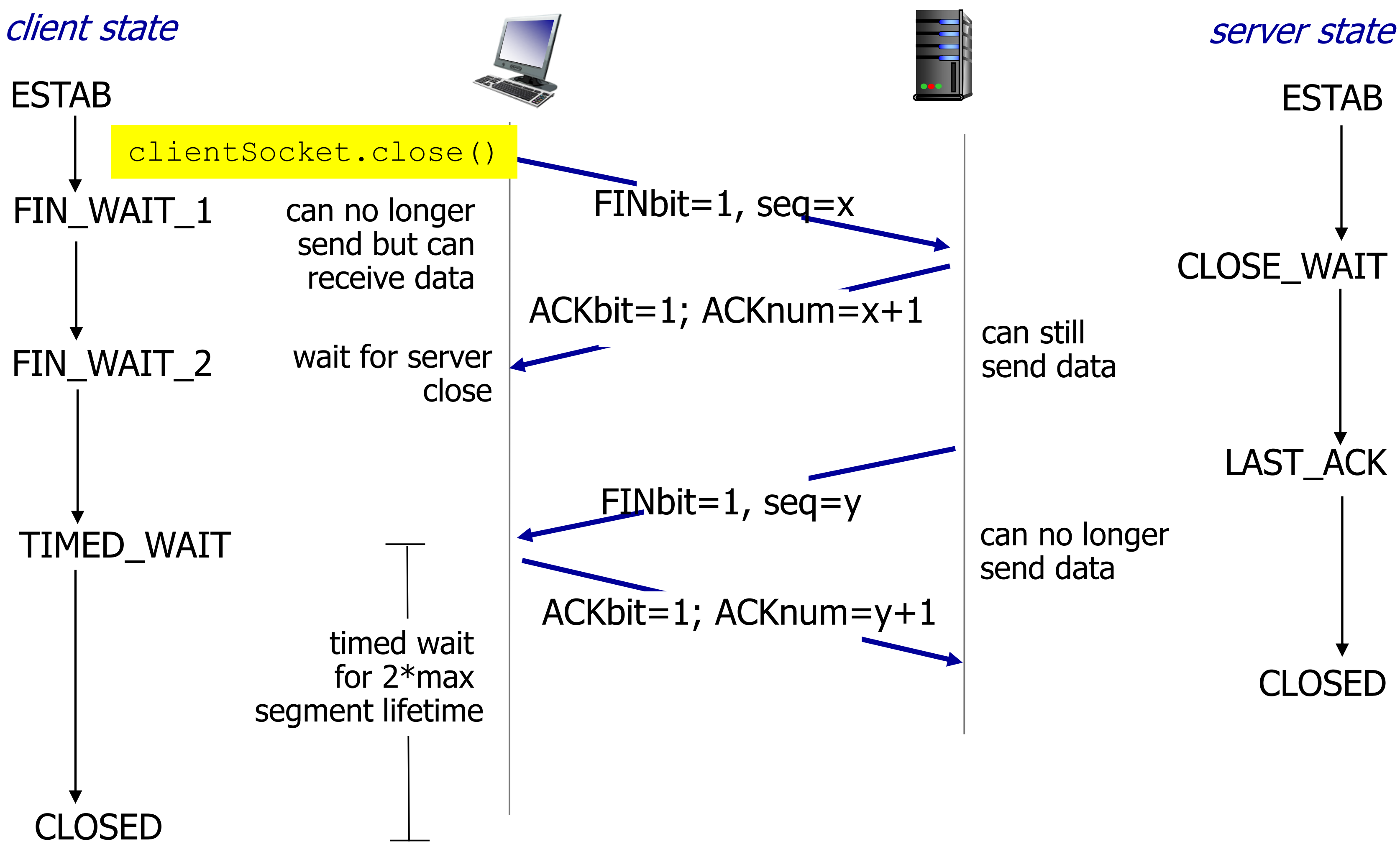
TCP 3-way handshake: FSM



TCP: closing a 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: closing a connection



TCP Congestion Control

Principles of congestion control

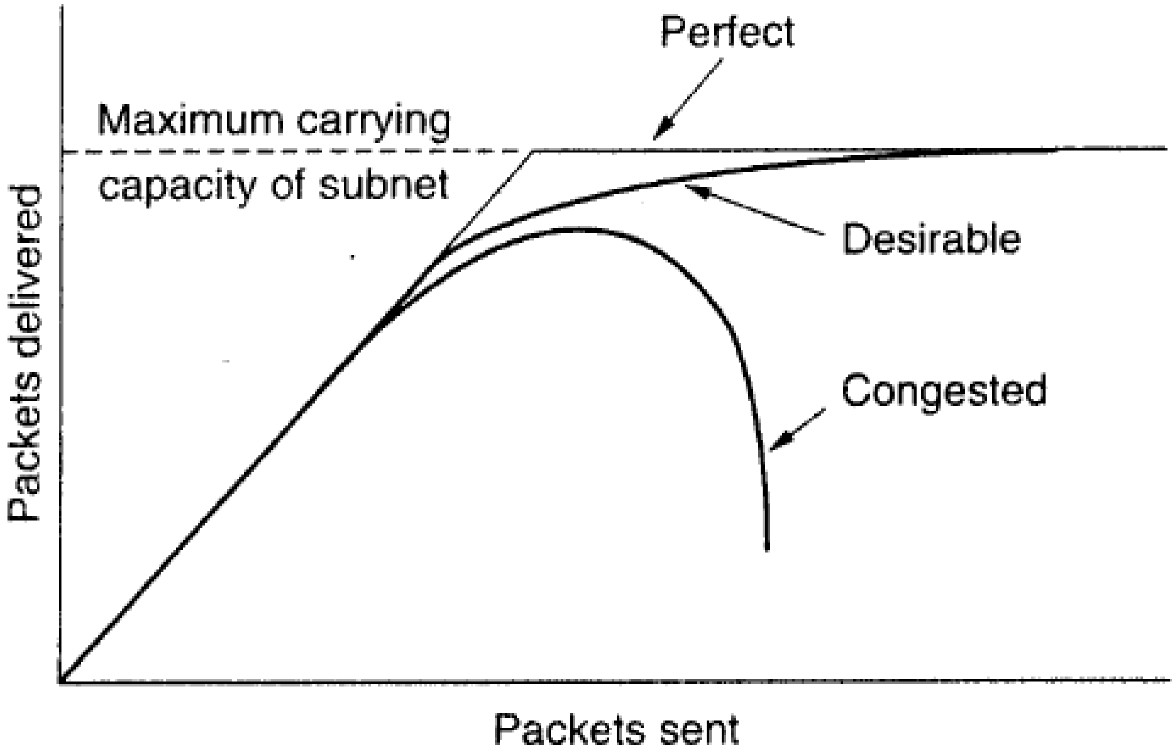
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control (receiver controlled)!
- congestion indicators:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

Flow vs congeston control

- (Flow control): the inability of the receiver accepting more data. Techniques addressing this problem are known as flow control, typically controlled by receiver.

- (Congestion control): the inability of the network delivering more data due to router buffer overflow. Techniques addressing this problem are known as congestion control, which include "sender-based flow control" (transmission rate) or dynamic routing (alternative routes).

Congestion collapse



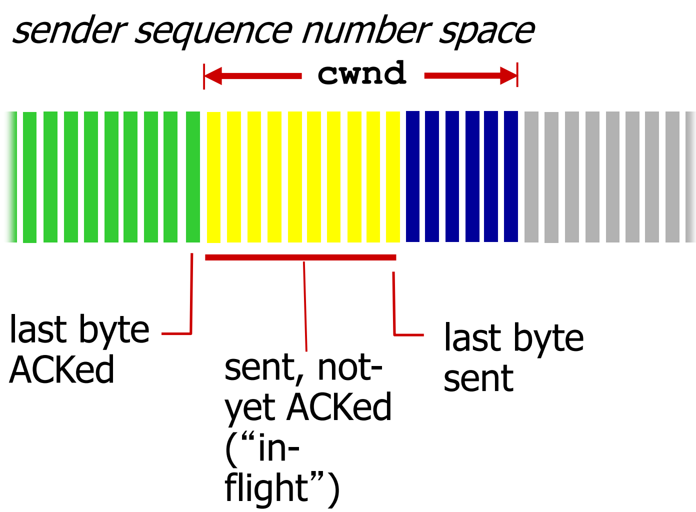
History of TCP congestion control

- 1973: 3-way handshake introduced
- 1978: TCP/IP split into TCP and IP
- 1986: Internet suffered congestion collapses
- 1988: TCP (Tahoe) introduced congestion control
- 1990: fast retransmit and recovery added by TCP (Reno), supported by most hosts today.

TCP before Tahoe

- Upon connection established, a sender transmits full receiver window size segments, as fast as possible!
- Upon timeout, retransmit all unACK'ed segments immediately!
- The network is full of window-size segments.
- Without control, congestion collapse occurs!

TCP Congestion Control



- sender limits transmission:

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}$$

- congestion window **`cwnd`** is dynamic, a kind of "sender-based flow control".

TCP sending rate:

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

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

TCP sender self-clocking

- A sender's congestion window (**cwnd**) is adjusted automatically via returned ACKs.
- If the network is not congested, then every new ACK will trigger **cwnd** adjustment.
- The faster the new ACK arrives (i.e., short RTT), the more frequent **cwnd** is adjusted.

TCP: congestion detection

- timeout:
  - `TimeOutInternval` is calibrated upon every ACK received.
  - When a timeout occurs, congestion is a likely cause.
- triple duplicate ACK(n)s:
  - Some data segments are lost but traffic is still getting through.
  - Duplicate ACK(n)s are most likely caused by out-of-order segments; multiple segments after "n" have been sent but not ACK'ed.

TCP congestion control algorithm

- TCP (Reno) has 3 phases:
  - **Slow Start** (**SS**)
  - **Congestion Avoidance** (**CA**)
  - **Fast Recovery** (**FR**)
- **SS** is about "bandwidth probing", find out what is max. available capacity of the connection.
- **CA** is about "slowing down" when congestion is about to occur.
- Finally, **FR** is about "restoring optimal" transmission rate after receiving too many duplicate ACKs.

TCP: state variables and events

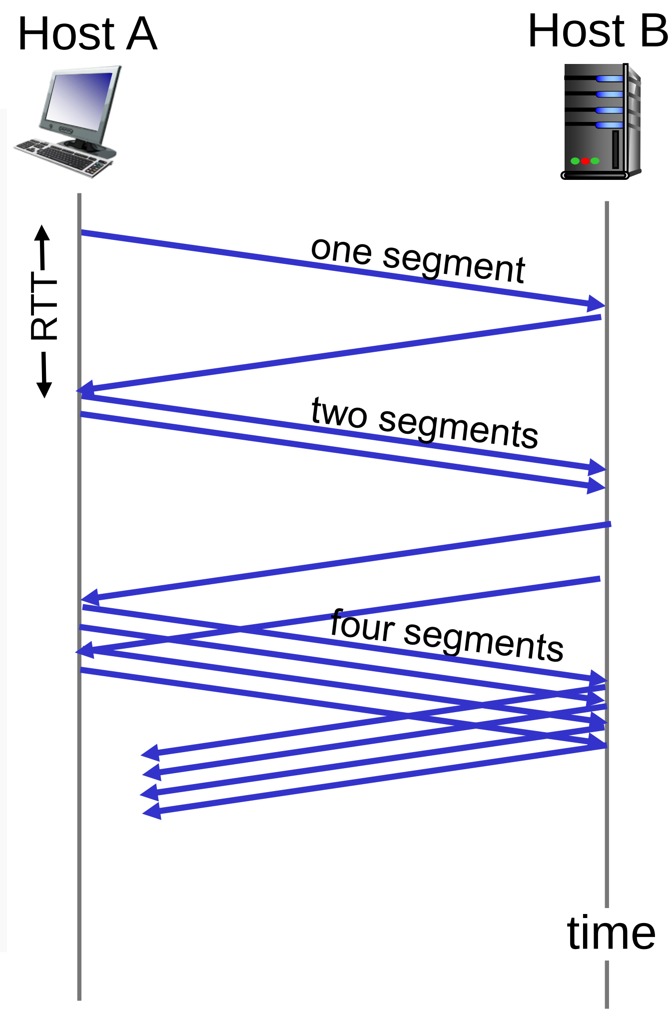
- In addition to **`cwnd`** (sender's congestion window), there is a **`ssthresh`** (threshold) triggering **SS** to **CA** phase change.
- **`cwnd`** and **`ssthresh`** are expressed in terms of MSS (max. segment size) bytes, typically 1460 bytes for MTU=1500 bytes.
- The following events trigger phase changes:
  - **newACK** : a new ACK just received
  - **3ACK(n)** : triple duplicate ACK(n)s
  - **timeout** : the oldest unACK'ed segment has just timeout.

TCP: initial conditions

- Upon connection setup, the receiver "advertises" its **`rwnd`** (max. free buffer space).
- Assume **`rwnd`** is always kept at its maximum (i.e., the receiver's host always picks up its data as fast as possible). Hence, we ignore flow control problems for now.
- To simplify our algorithm, we initialize **`cwnd`** to 2*64KB/MSS (will become clear later).
  (Note: TCP can send up to 64KB per IP datagram!)
- The algorithm starts in **SS** phase.

TCP Slow start

```
SS:  // probing bandwidth
ssthresh = cwnd/2;
cwnd = 1;
loop {
  event newACK:
    cwnd += 1;
    transmit new segments;
    if (cwnd==ssthresh) goto CA;
  event timeout:
    retransmit unACK'ed segments;
    goto SS;
  event 3ACK(n):
    retransmit segment @ n;
    goto FR;
}
```

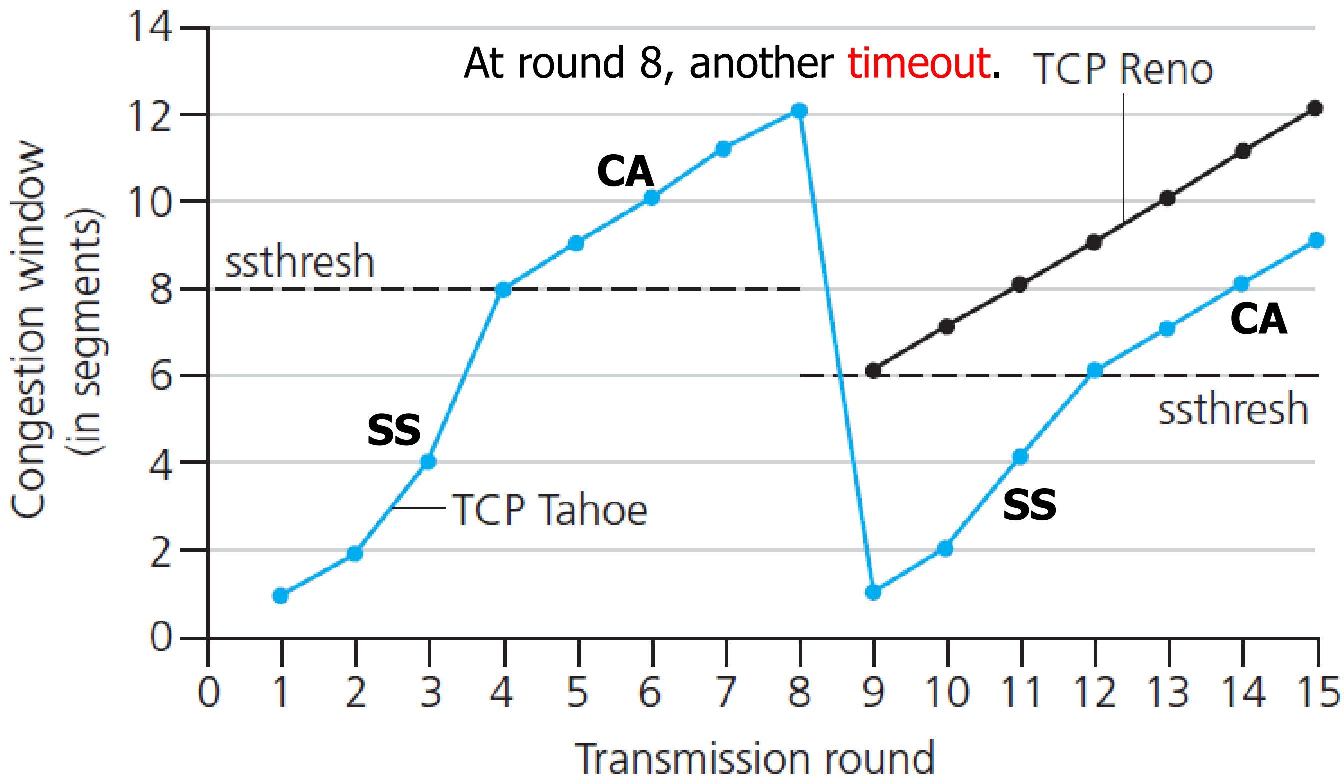


TCP Congestion avoidance

```
CA:  // slowing down
loop {
  event newACK:
    cwnd += 1/cwnd;
    transmit new segments;
  event timeout:
    goto SS;
  event 3ACK(n):
    retransmit segment @ n;
    goto FR;
}
```

TCP: **SS** to **CA (example)**

At round 0, a timeout occurs when cwnd=16;
hence, ssthresh=16/2=8, and cwnd=1 at round 1.



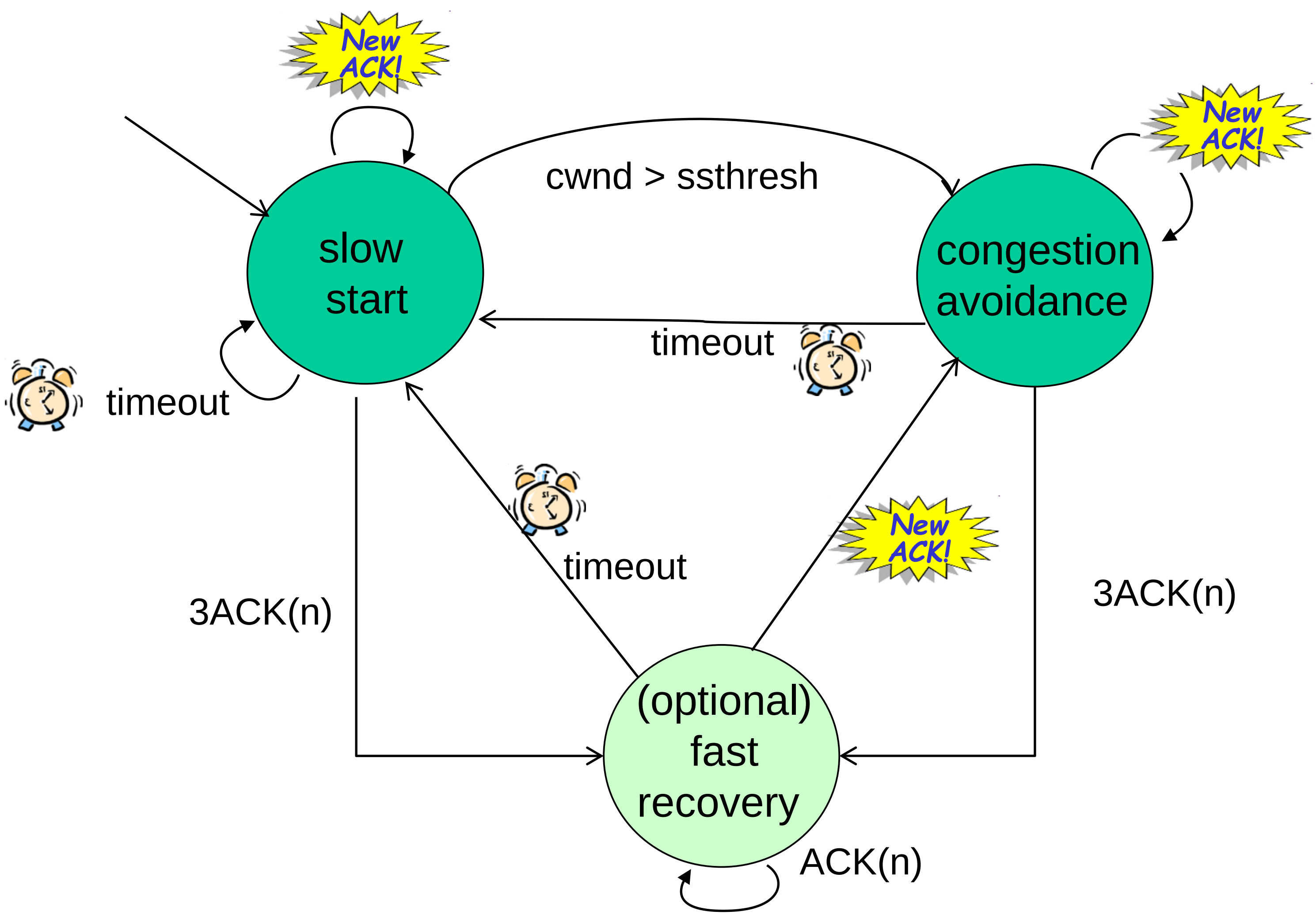
TCP (Reno) Fast recovery

```
FR:  // restoring rate
ssthresh = cwnd/2;
cwnd = ssthresh + 3;
loop{
  event newACK:
    goto CA;
  event timeout:
    goto SS;
  event ACK(n):
    cwnd += 1;
    retransmit segments @ n up
    to cwnd;
}
```

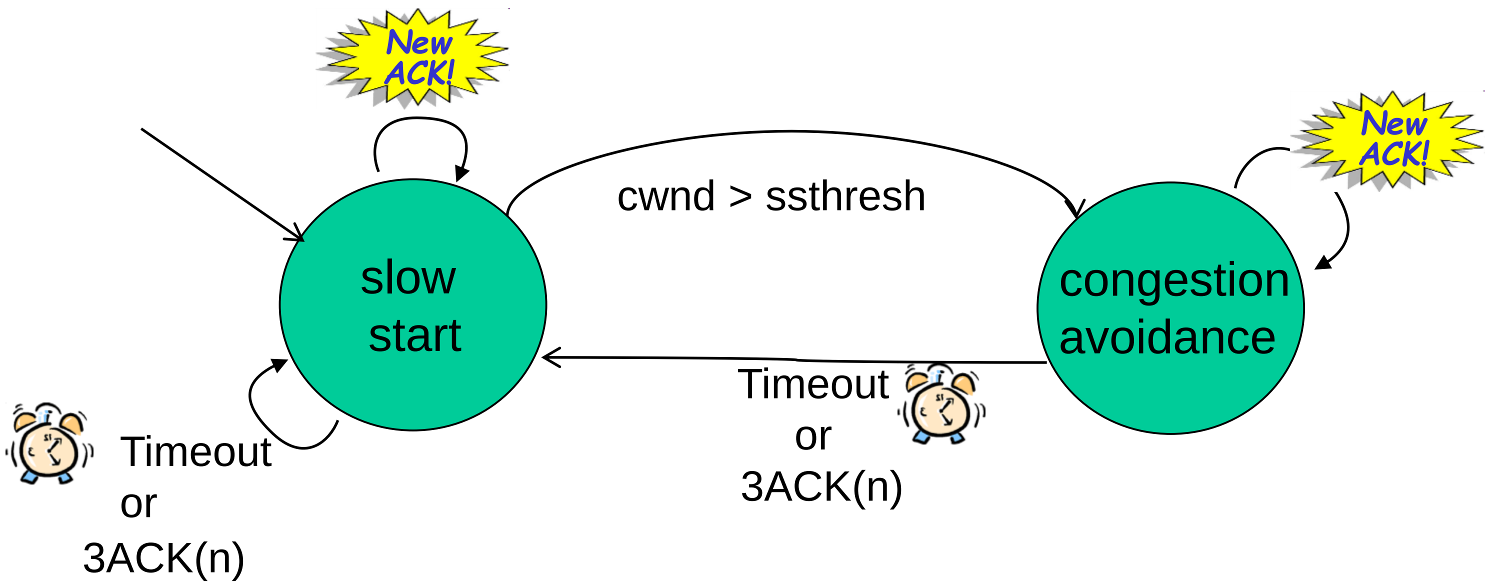
TCP (Tahoe)

- Fast recovery (**FR** optional) was introduced by TCP (Reno) around 1990, which is supported by most TCP installations today.
- However, TCP (Tahoe) doesn't support **FR**.
- Events timeout and 3ACK(n) are treated the same, as a segment loss; as a result, TCP (Tahoe) goes back to **SS** phase.

TCP (Reno): Congestion control



TCP (Tahoe): Congestion control



TCP: additive I multiplicative D

- transmission rate (**`cwnd`**) is increased until **loss** (timeout or 3ACKs) occurs
  - *additive increase:* increase **`cwnd`** by 1
  - *multiplicative decrease:* set **`ssthresh=cwnd`**/2, and set **`cwnd=1`**

AIMD sawtooth behavior: probing for bandwidth