

# Transport Layer

## Chapter goals:

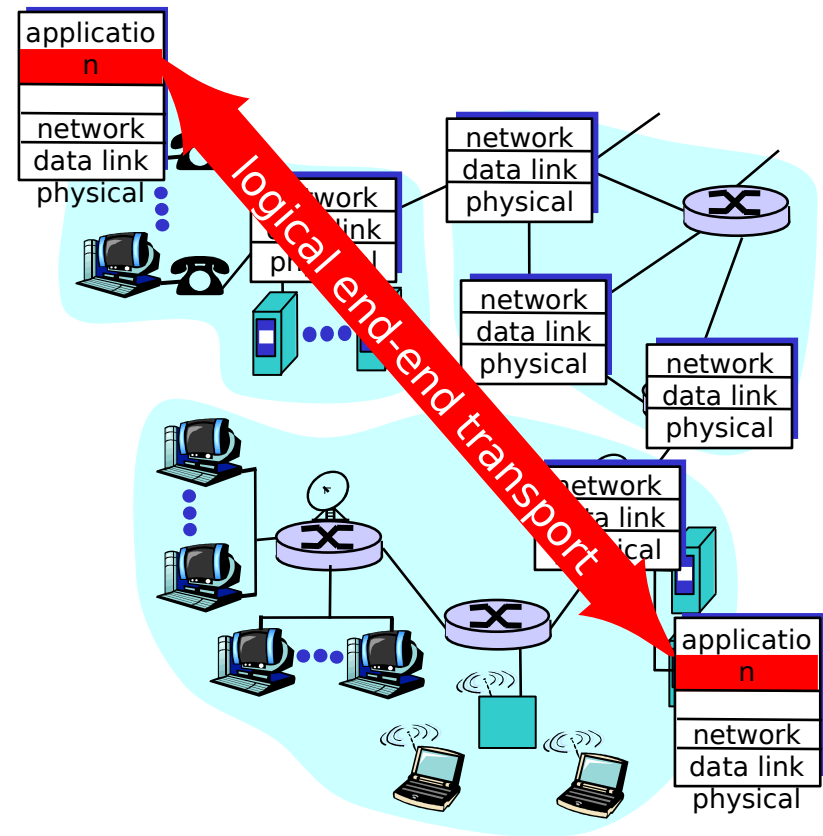
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
    - reliable data transfer
    - flow control
    - congestion control
- instantiation and implementation in the Internet

## Chapter Overview:

- transport layer services
- multiplexing/demultiplexing
- connectionless transport: UDP
- principles of reliable data transfer
- connection-oriented transport: TCP
  - reliable transfer
    - flow control
    - connection management
- principles of congestion control
- TCP congestion control

# Transport services and protocols

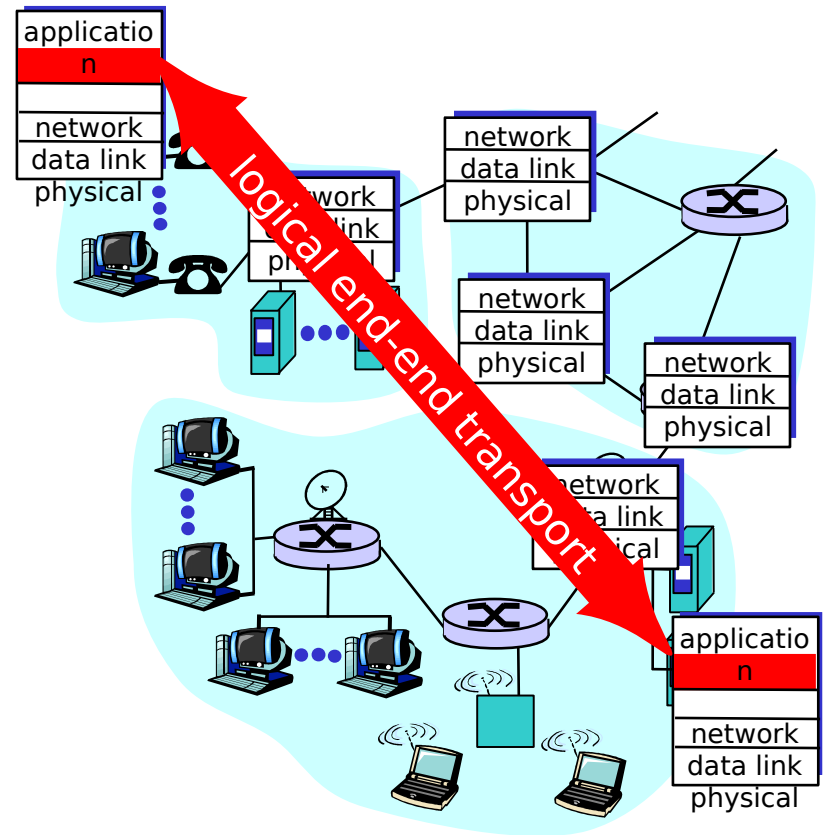
- provide *logical communication* between app' processes running on different hosts
- transport protocols run in end systems
- *transport vs network layer services:*
- *network layer:* data transfer between end systems
- *transport layer:* data transfer between processes
  - relies on, enhances, network layer services



# Transport-layer protocols

## Internet transport services:

- reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- unreliable (“best-effort”), unordered unicast or multicast delivery: UDP
- services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast

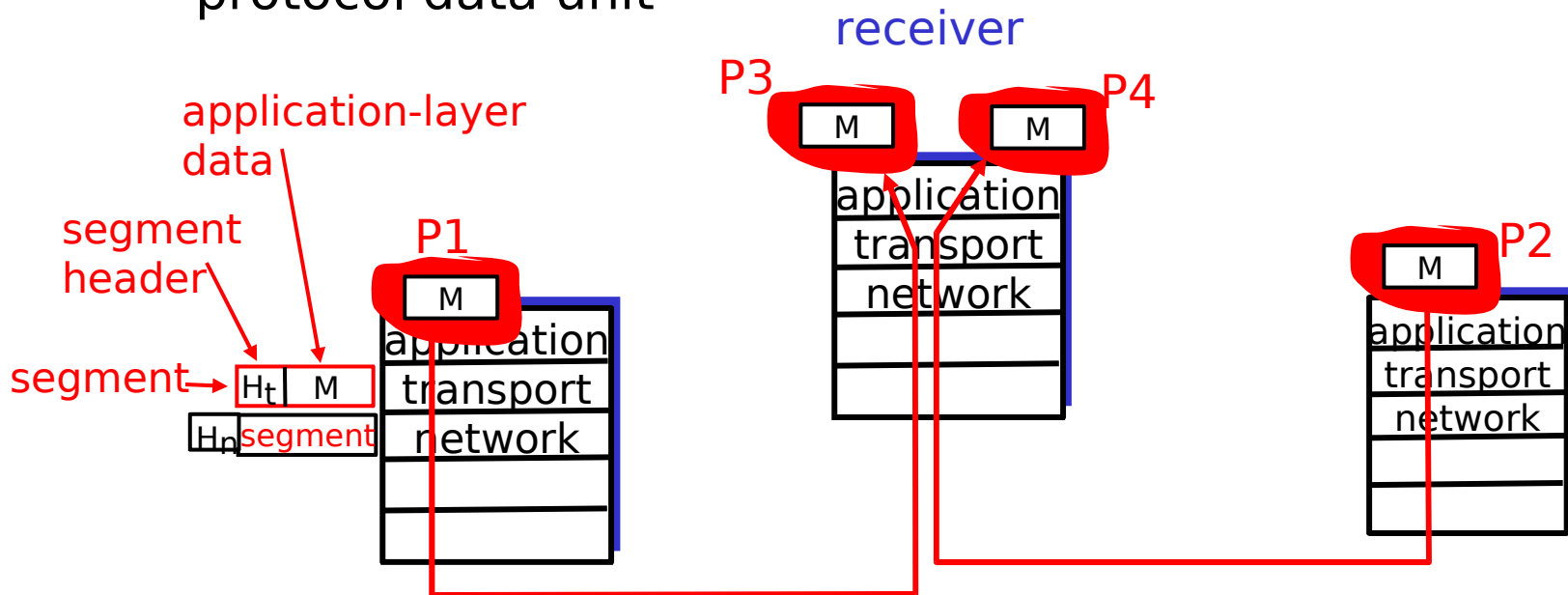


# Multiplexing/demultiplexing

Recall: *segment* - unit of data exchanged between transport layer entities

- aka TPDU: transport protocol data unit

**Demultiplexing:** delivering received segments to correct app layer processes



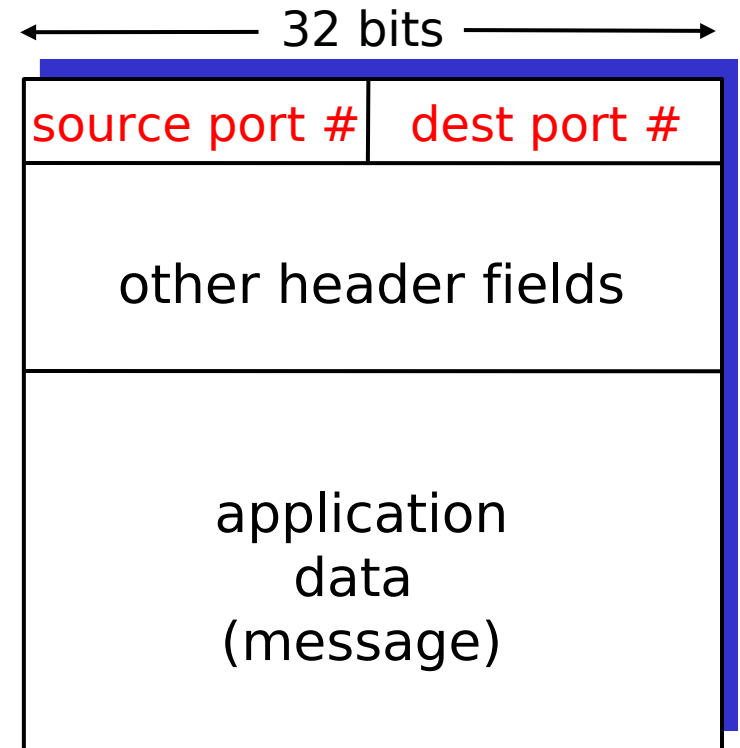
# Multiplexing/demultiplexing

**Multiplexing:**  
gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

multiplexing/demultiplexing

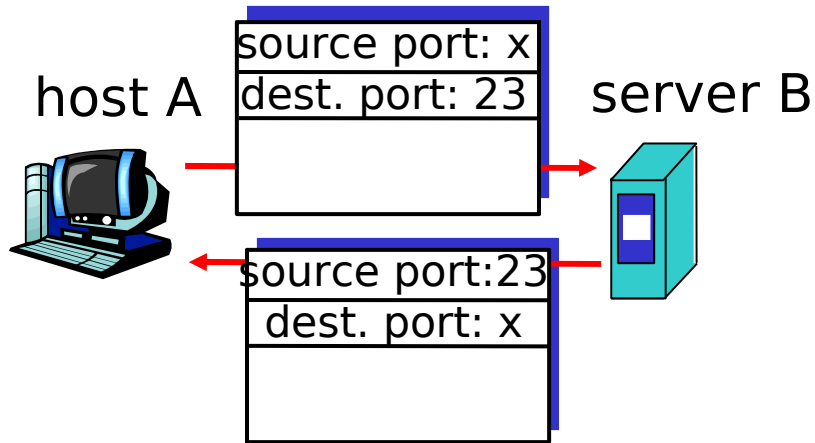
:

- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications

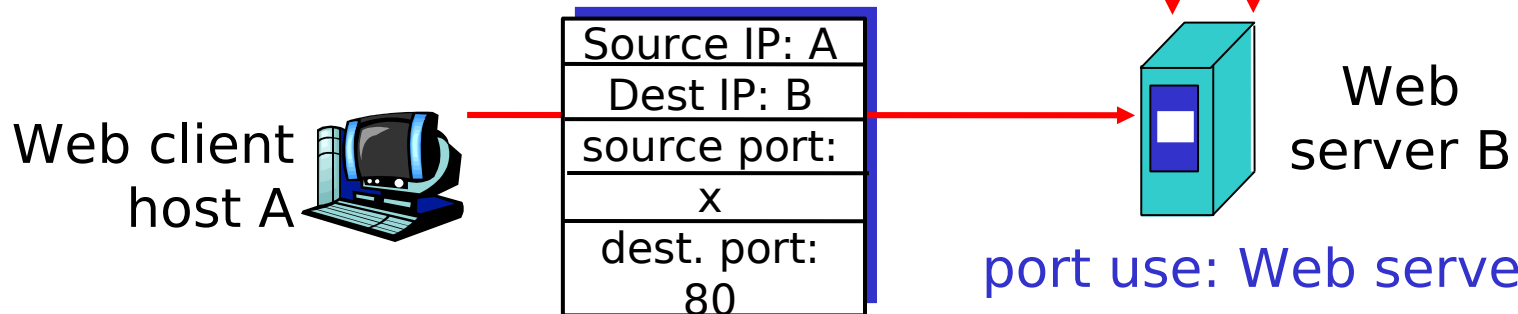
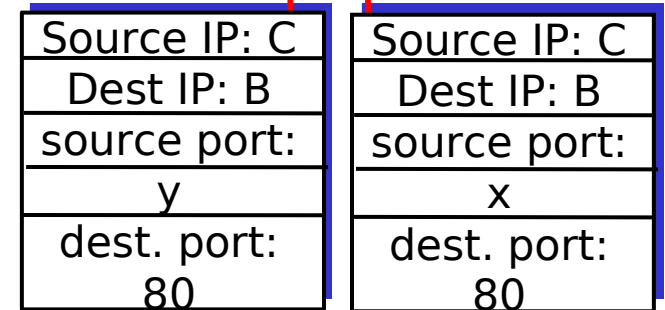


TCP/UDP segment format

# Multiplexing/demultiplexing: examples



port use: simple telnet app



port use: Web server

# UDP: User Datagram Protocol [RFC 768]

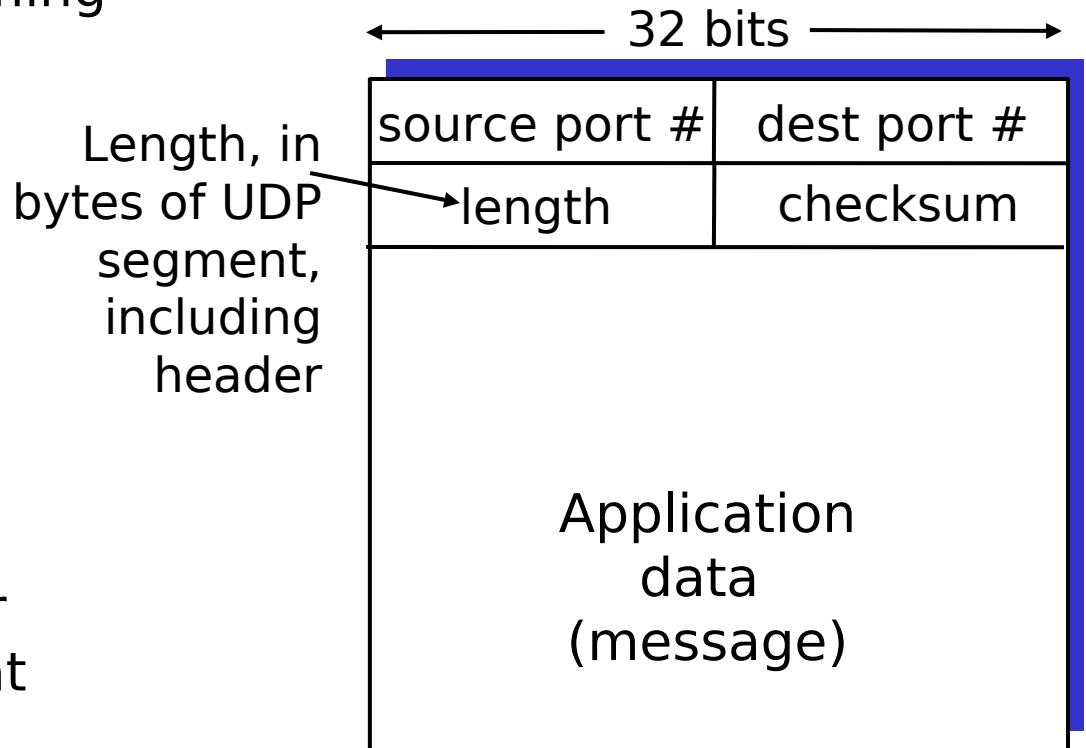
- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless*:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

## Why is there a UDP?

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

# UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses (why?):
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recover!



UDP segment format

# UDP checksum

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

## Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

## Receiver:

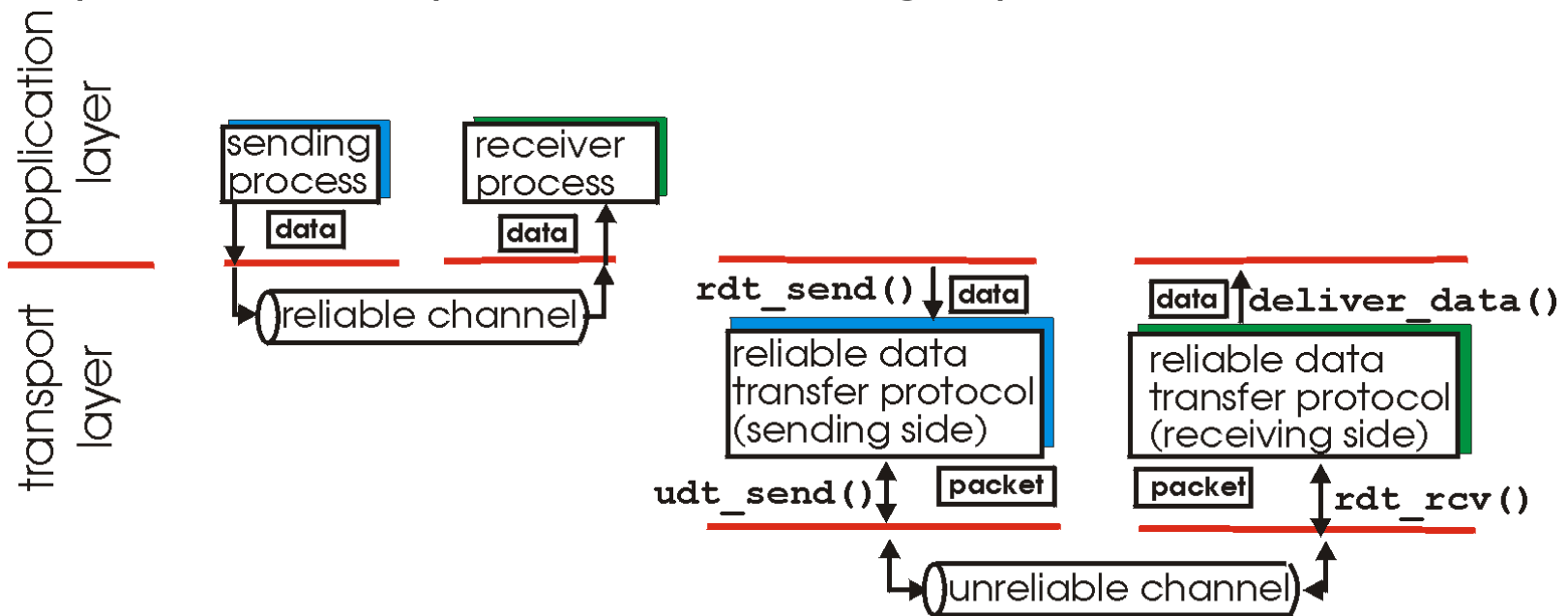
- compute checksum of received segment
  - check if computed checksum equals checksum field value:
    - NO - error detected
    - YES - no error detected. *But maybe errors nonetheless? More later*
- ....

# UDP checksum example:

- Three packets of 16 bits each
  - 0110011001100110
  - 0101010101010101
  - 0000111100001111
- adding the three, calling it 'r':
  - 1100101011001010
- Send the four packets, the original three and 1's complement of 'r' to destination
- The 1's complement of 'r' is:
  - 0011010100110101
- at destination the sum of four packets should be:
  - 1111111111111111
- If the packet is damaged:
  - 1111101111111111 (zeros!!)

# Principles of Reliable data transfer

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



(a) provided service

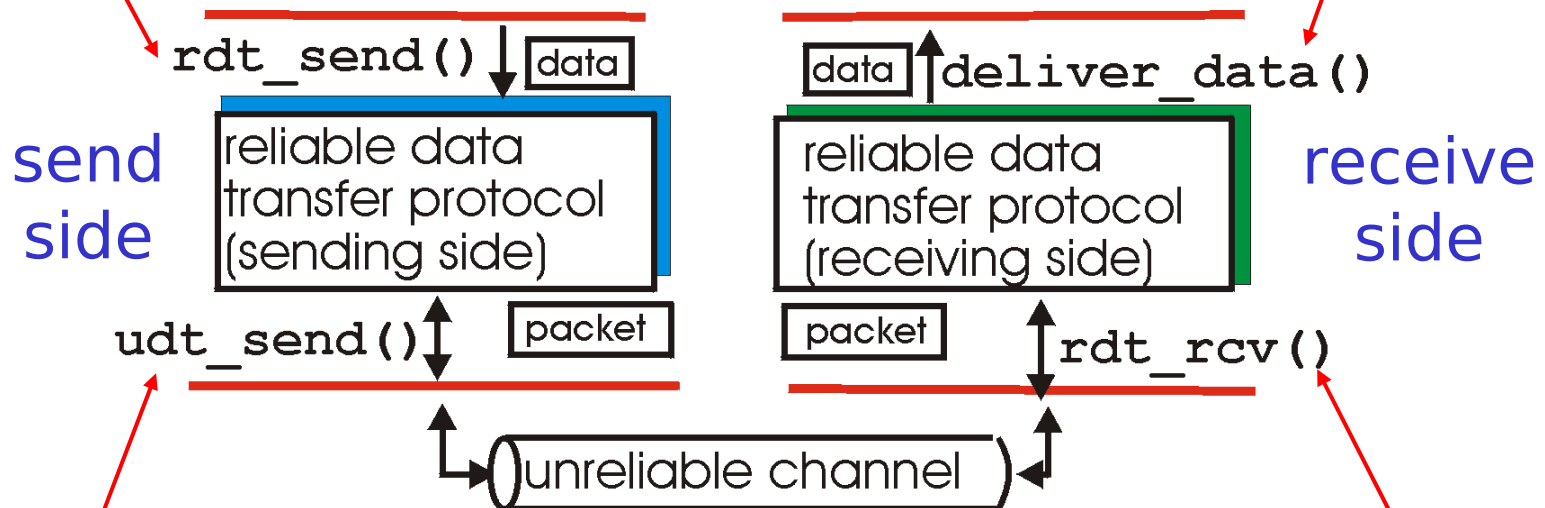
(b) service implementation

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

# Reliable data transfer: getting started

**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

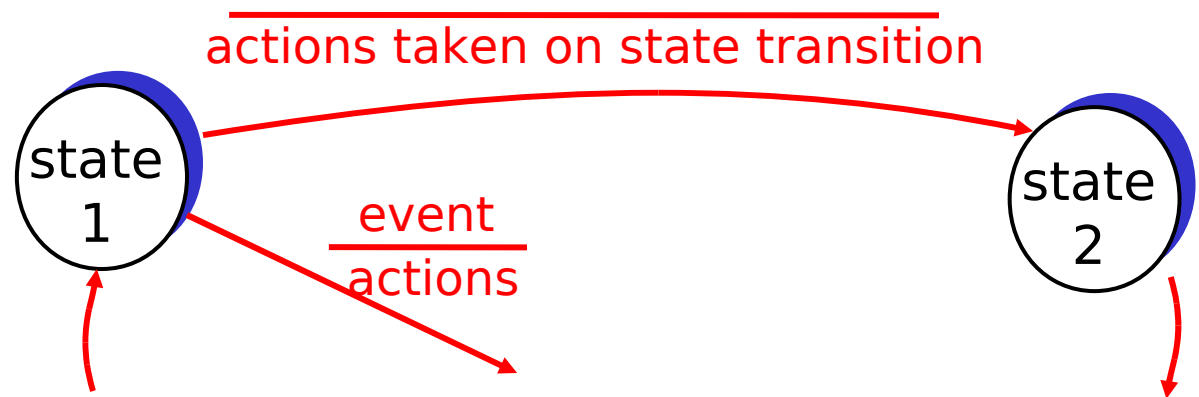
**rdt\_rcv()**: called when packet arrives on rcv-side of channel

# Reliable data transfer: getting started

We'll:

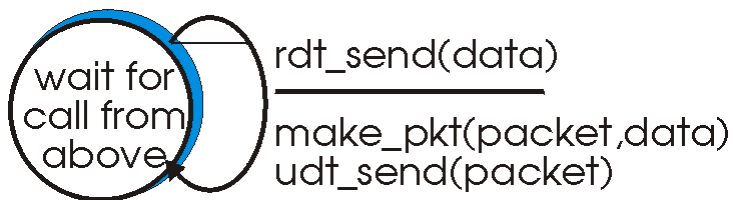
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
  - consider only unidirectional data transfer
    - but control info will flow on both directions!
  - use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event

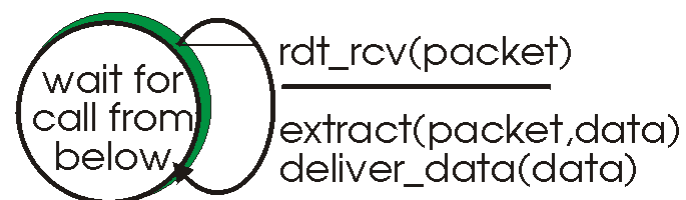


# Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



(a) rdt1.0: sending side

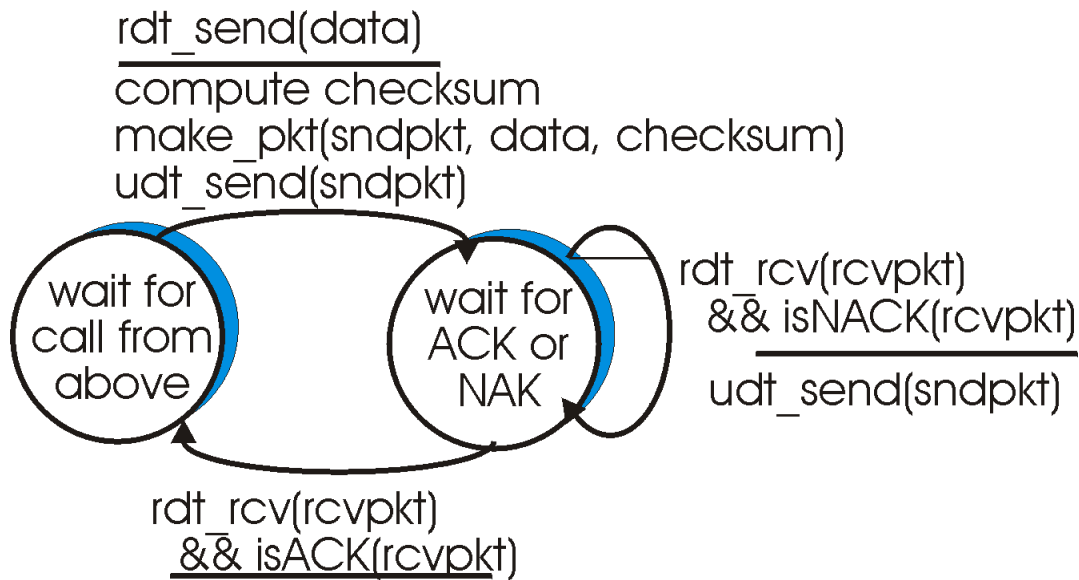


(b) rdt1.0: receiving side

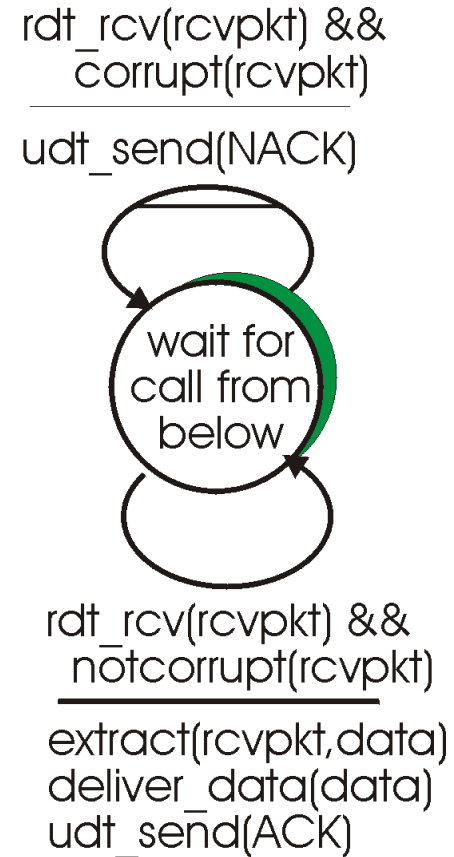
## Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP checksum to detect bit errors
- *the question*: how to recover from errors:
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
  - human scenarios using ACKs, NAKs?
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

# rdt2.0: FSM specification

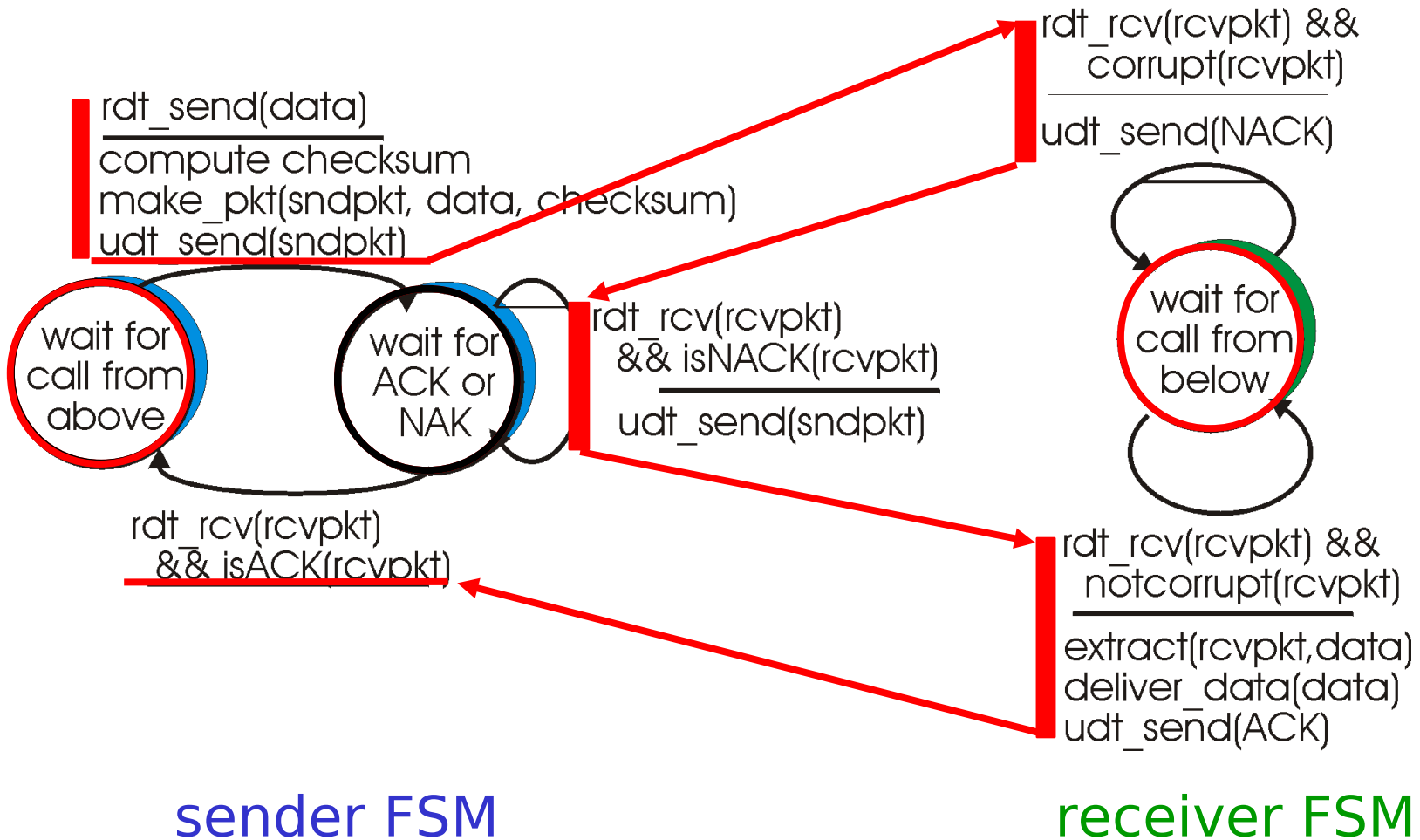


sender FSM



receiver FSM

# rdt2.0: in action (error scenario)



# rdt2.0 has a fatal flaw!

## What happens if ACK/NAK corrupted?

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

## What to do?

- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

## Handling duplicates:

- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

### stop and wait

Sender sends one packet, then waits for receiver response

# rdt3.0: channels with errors *and* loss

## New assumption:

underlying channel can also lose packets (data or ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

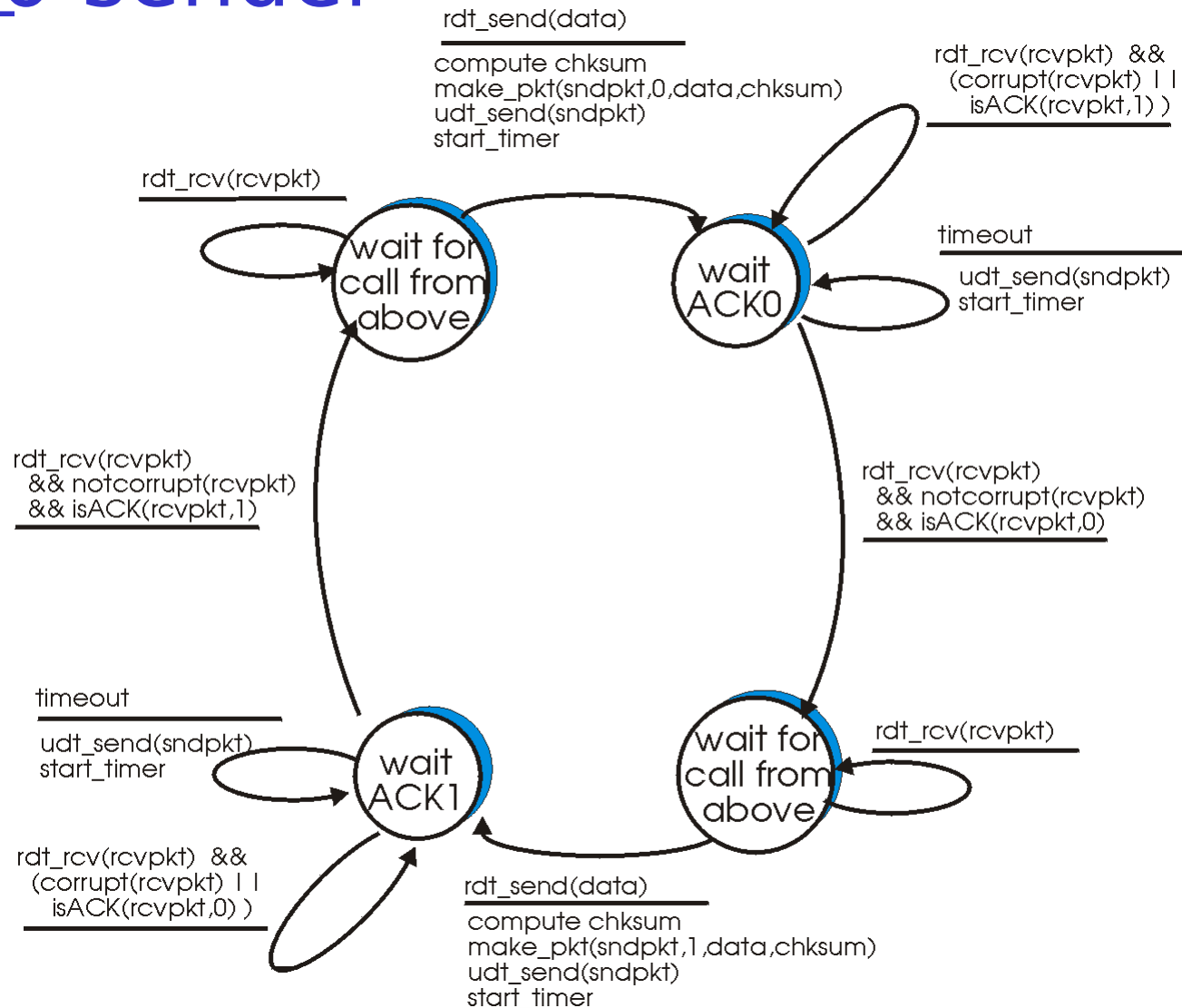
## Q: how to deal with loss?

- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

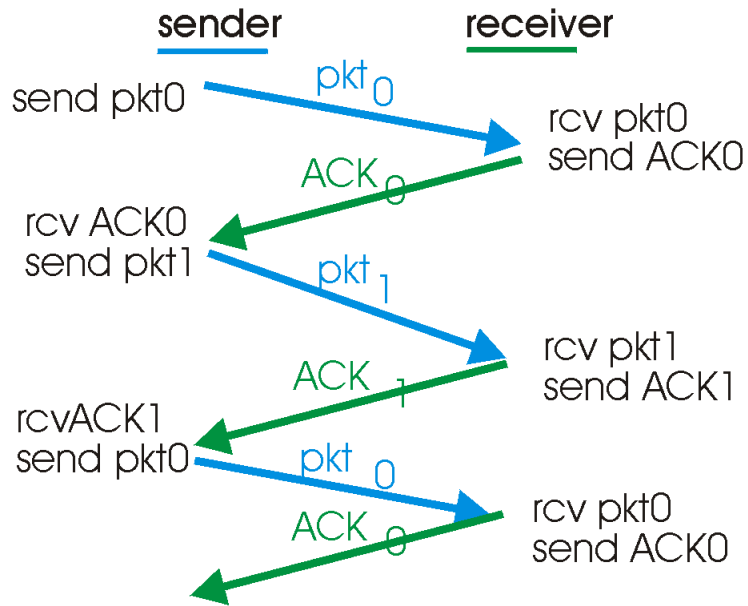
Approach: sender waits “reasonable” amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

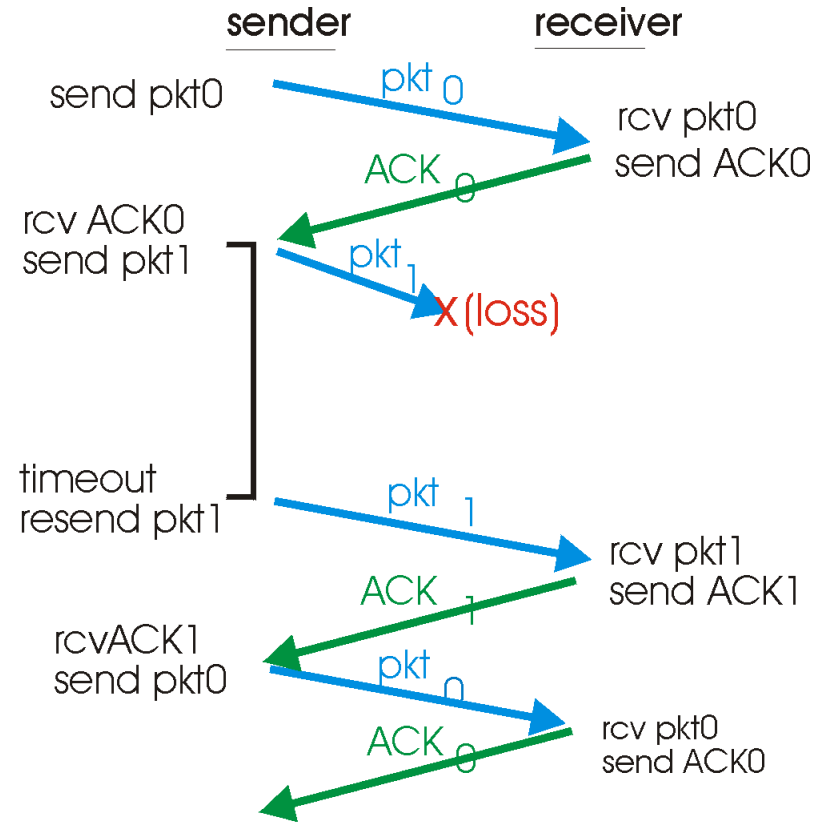
# rdt3.0 sender



# rdt3.0 (hypothetical protocol) in action

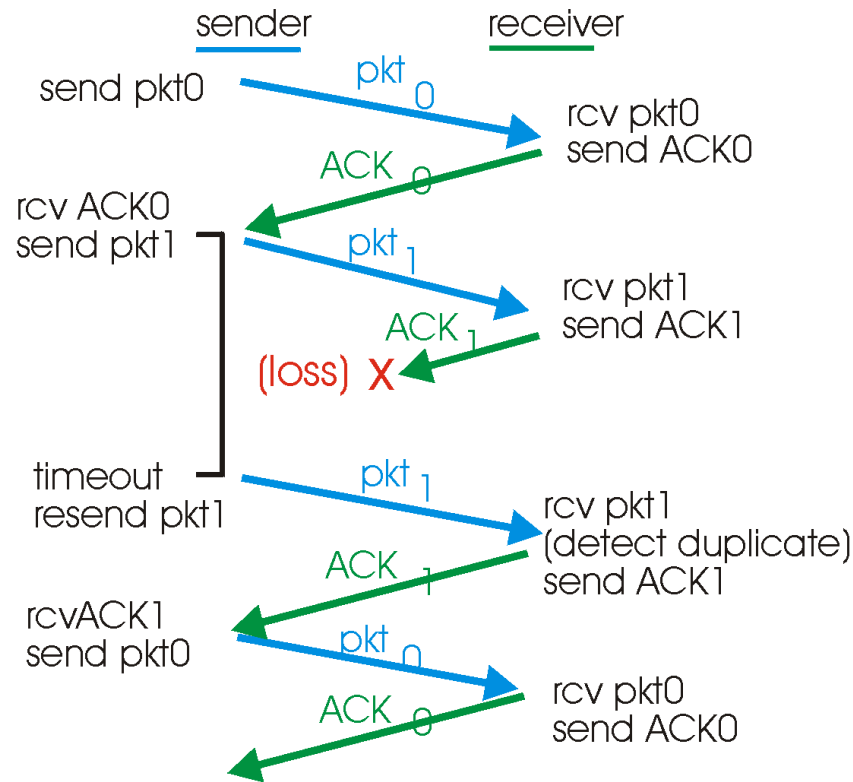


(a) operation with no loss

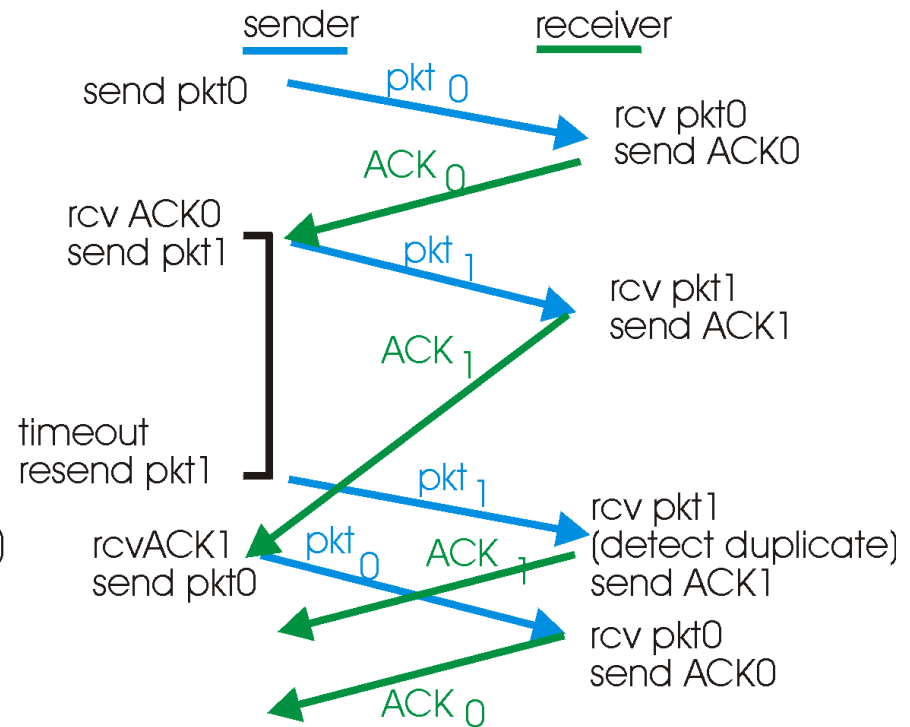


(b) lost packet

# rdt3.0 in action



(c) lost ACK



(d) premature timeout

# Performance of rdt3.0

- rdt3.0 works, but performance is not acceptable
- example: 1 Gbps link, 15 ms end to end propagation delay, 1KB packet (1KByte = 8Kbit)
- Utilization of sender (time busy sending)

$$T_{\text{transmit}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

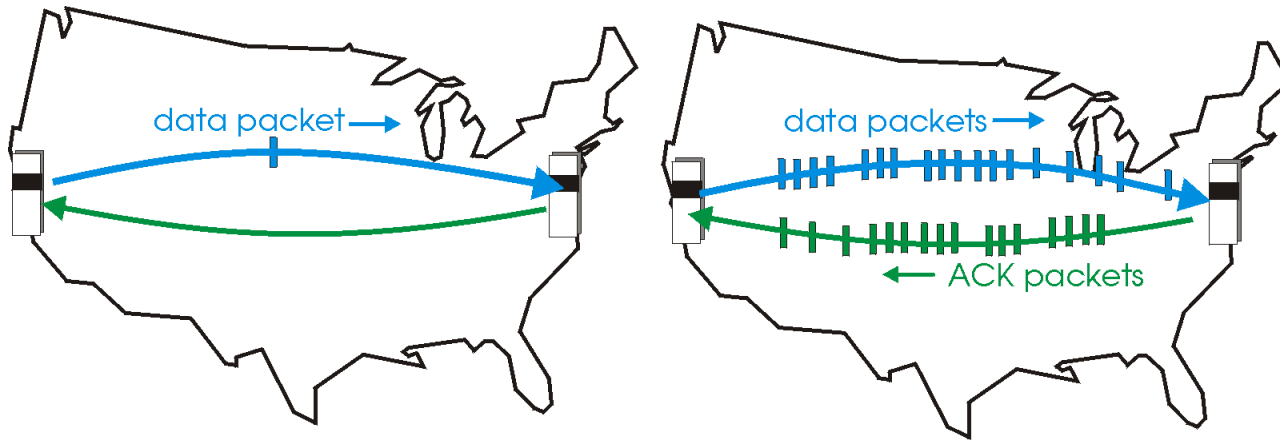
$$\text{Utilization} = U = \frac{\text{fraction of time sender busy sending}}{30.008 \text{ msec}} = \frac{0.008 \text{ msec}}{30.008 \text{ msec}} = 0.00027$$

- 1KB packet every 30 msec -> 267kb/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

# 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

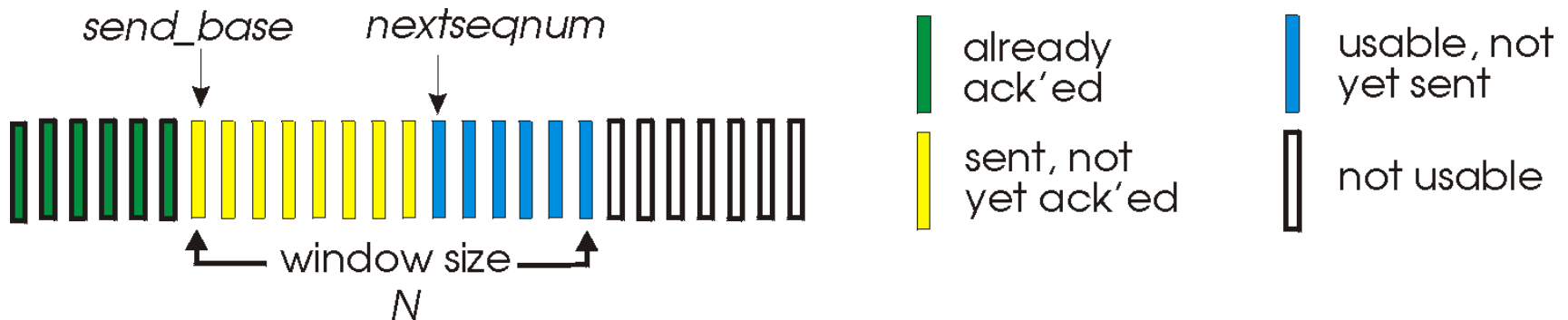
(b) a pipelined protocol in operation

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

# Go-Back-N

## Sender:

- k-bit sequence # in packet header
- “window” of up to N, consecutive unack’ed packets allowed



- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

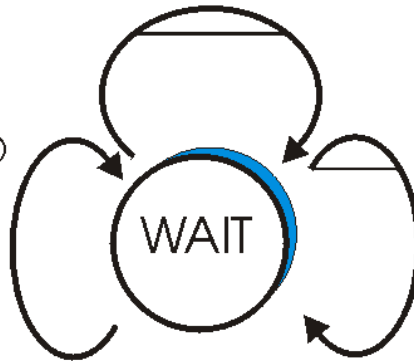
# GBN: sender extended FSM

rdt\_send(data)

```
if (nextseqnum < base+N) {  
    compute chksum  
    make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum)  
    udt_send(sndpkt(nextseqnum))  
    if (base == nextseqnum)  
        start_timer  
    nextseqnum = nextseqnum + 1  
}  
else  
    refuse_data(data)
```

rdt\_rcv(rcv\_pkt) && notcorrupt(rcvpkt)

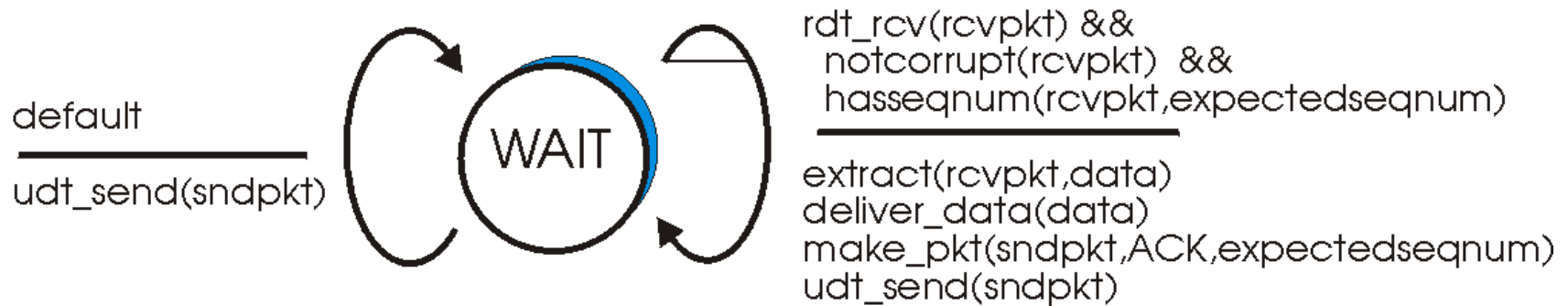
```
base = getacknum(rcvpkt)+1  
if (base == nextseqnum)  
    stop_timer  
else  
    start_timer
```



timeout

```
start_timer  
udt_send(sndpkt(base))  
udt_send(sndpkt(base+1))  
.....  
udt_send(sndpkt(nextseqnum-1))
```

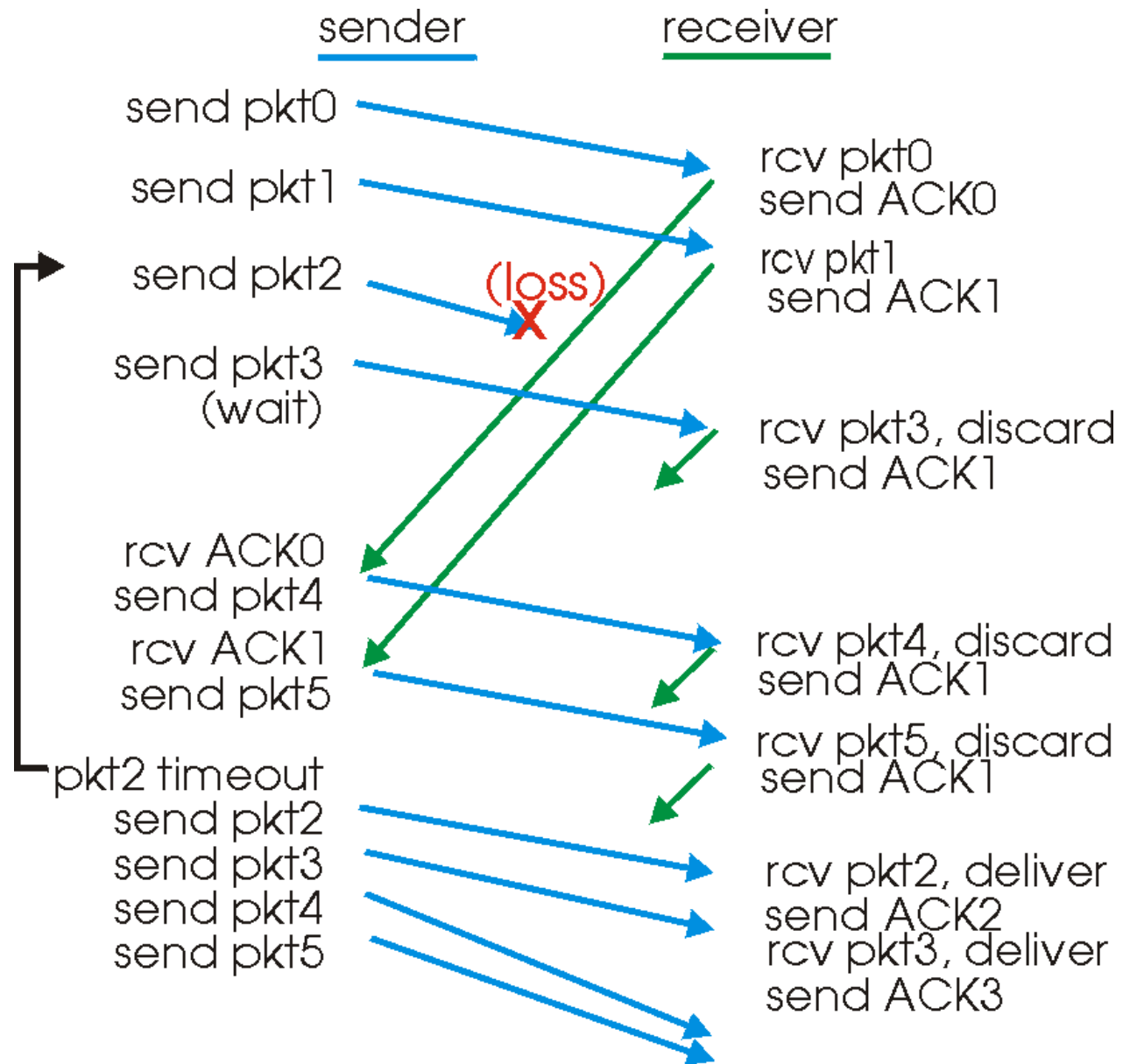
# GBN: receiver extended FSM



## receiver simple:

- ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember **expectedseqnum**
- out-of-order pkt:
  - discard (don't buffer) -> **no receiver buffering!**
  - ACK pkt with highest in-order seq #

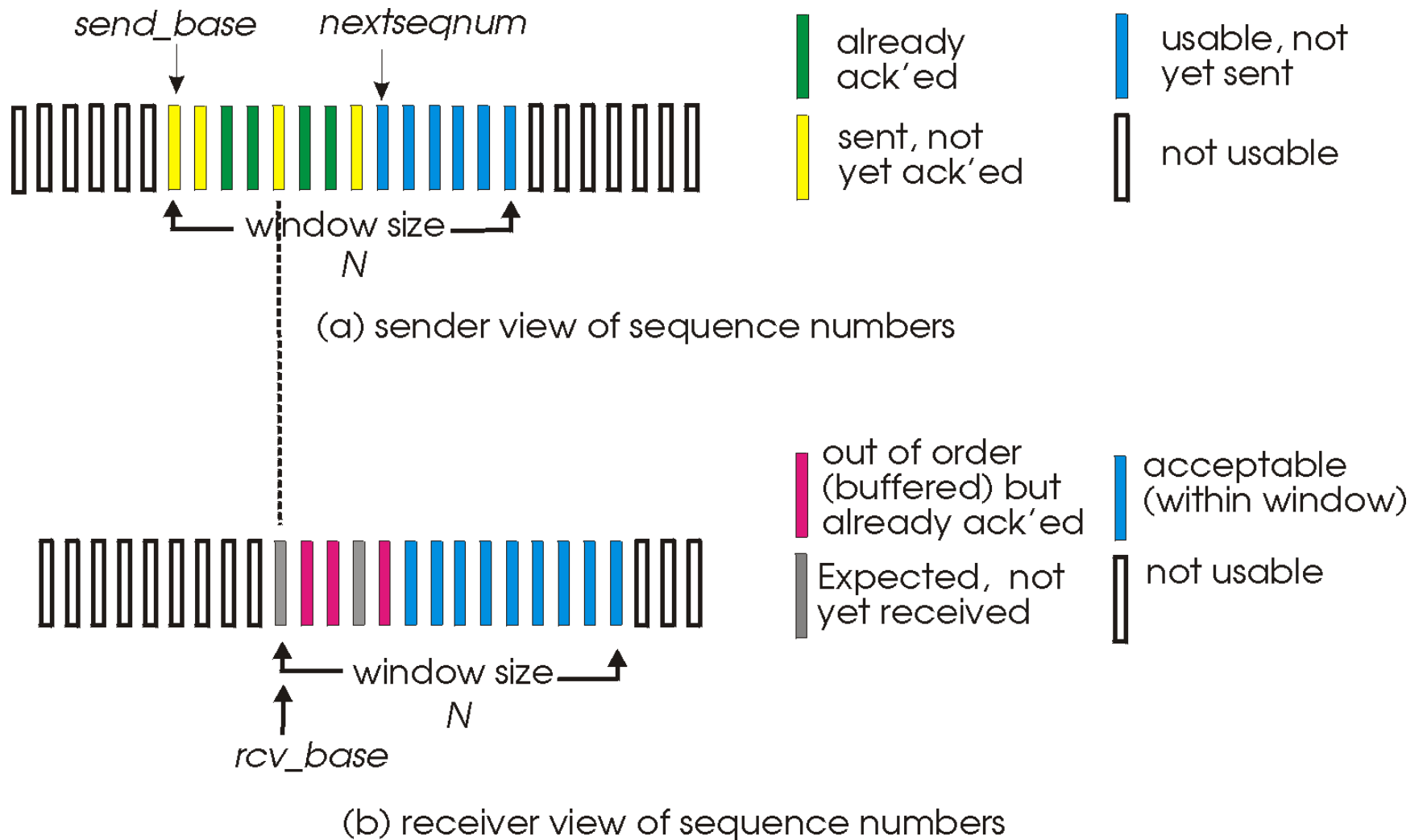
# GBN in action



# Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #'s of sent, unACKed pkts

# 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

[sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

## receiver

### pkt n in [rcvbase, rcvbase+N-1]

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

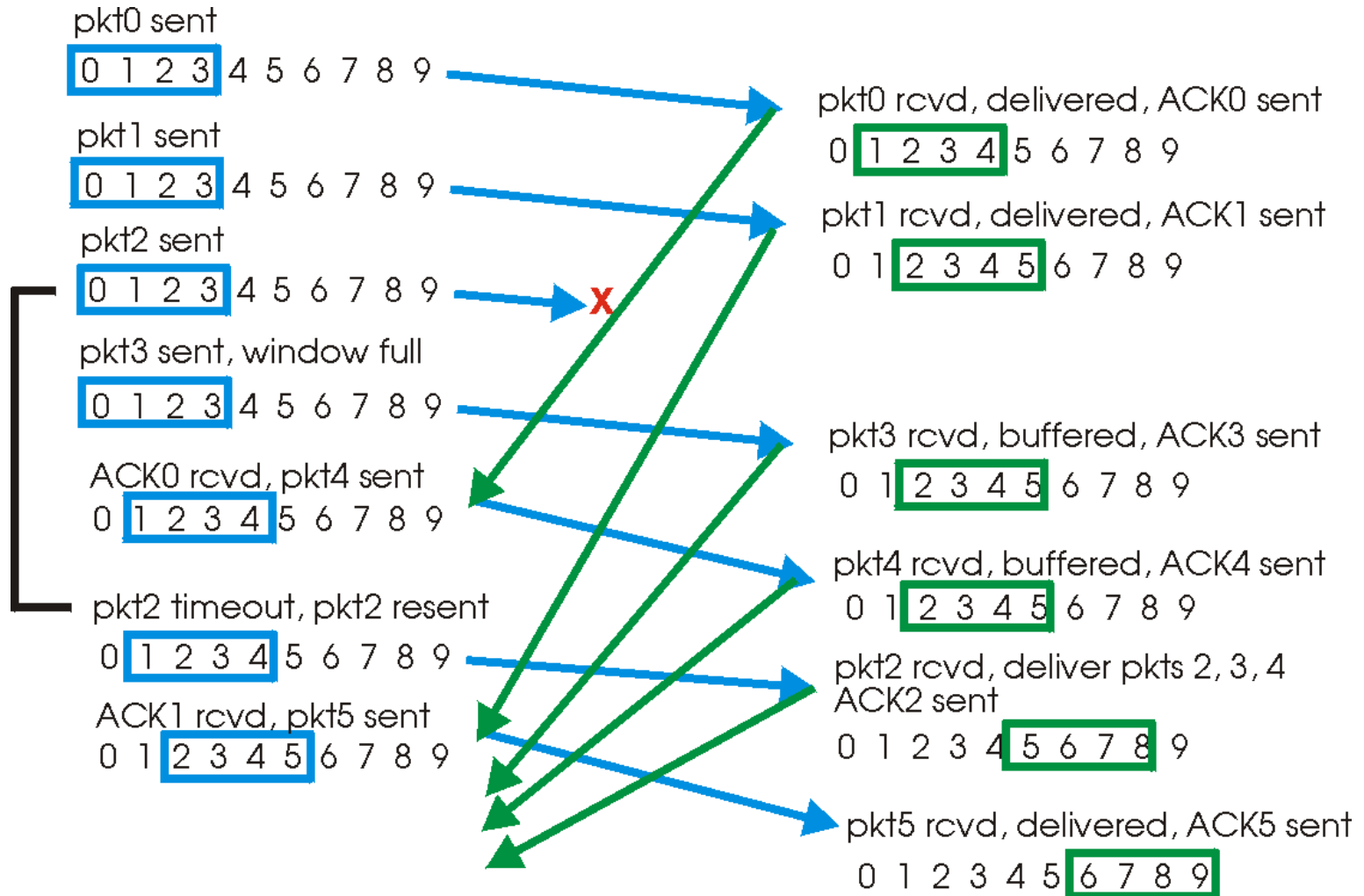
### pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

### otherwise:

- ignore

# Selective repeat in action



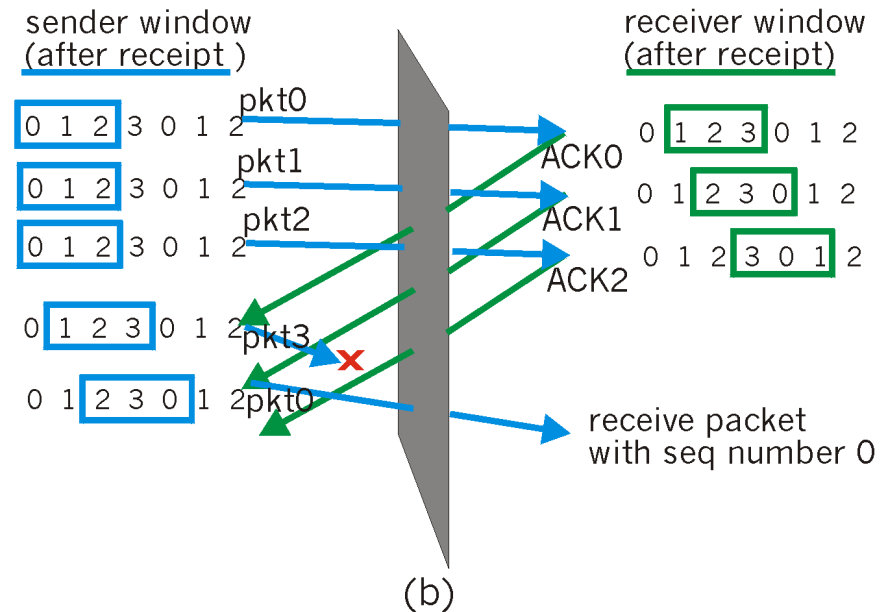
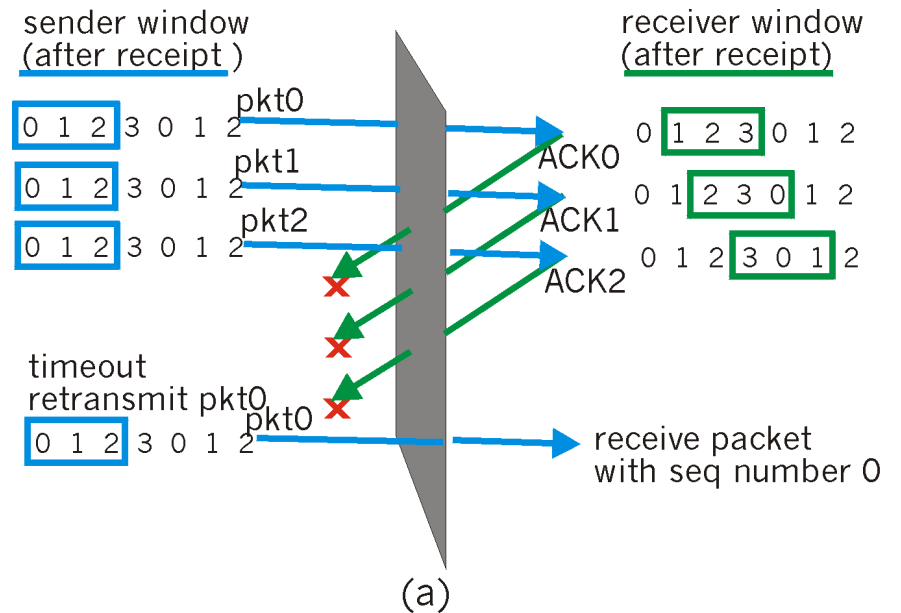
# Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

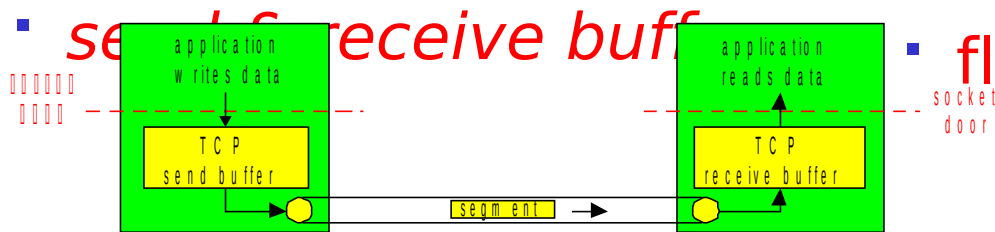
- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

**Q:** what relationship between seq # size and window size?

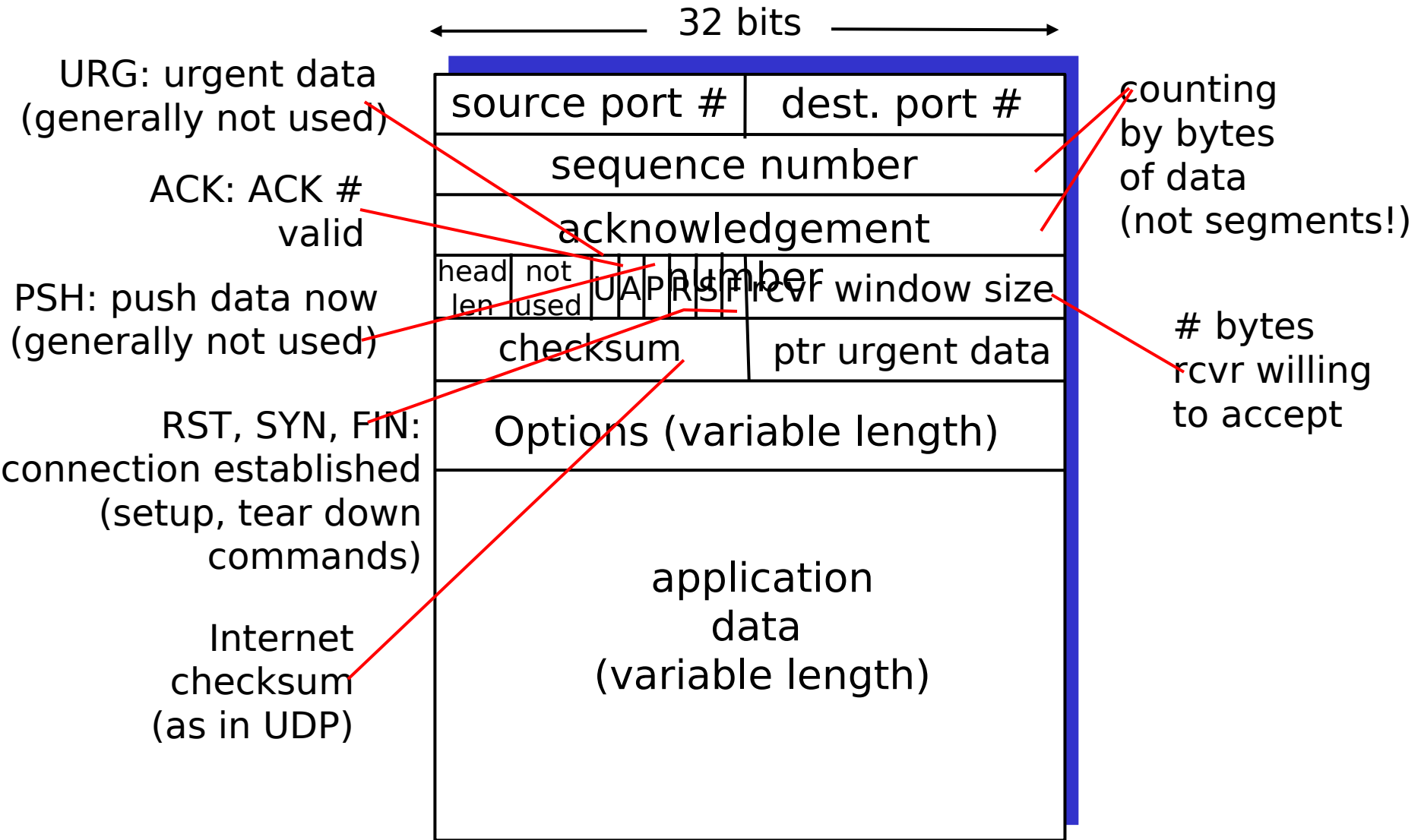


# TCP: Overview

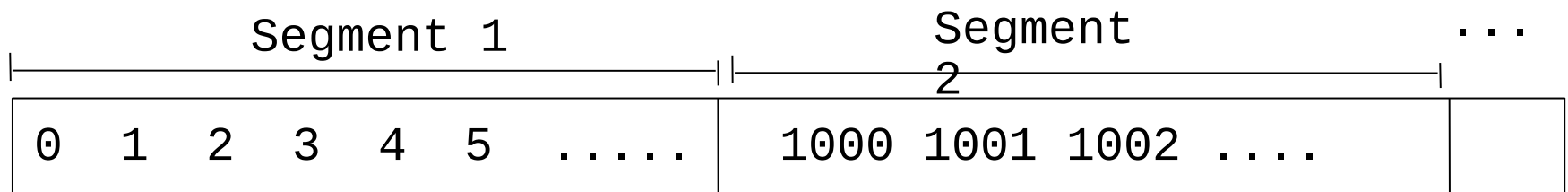
- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order *byte stream*:**
  - no message boundaries
- **pipelined:**
  - TCP congestion and flow control set window size
- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- **connection-oriented:**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



# TCP segment structure



# TCP sequence #'s and ACKs



## Sequence Numbers (#'s):

- byte stream 'number' of first byte in segment's data
- Not necessarily starts from 0, use random initial number R
  - Segment 1:  $0 + R$
  - Segment 2:  $1000 + R$  etc...

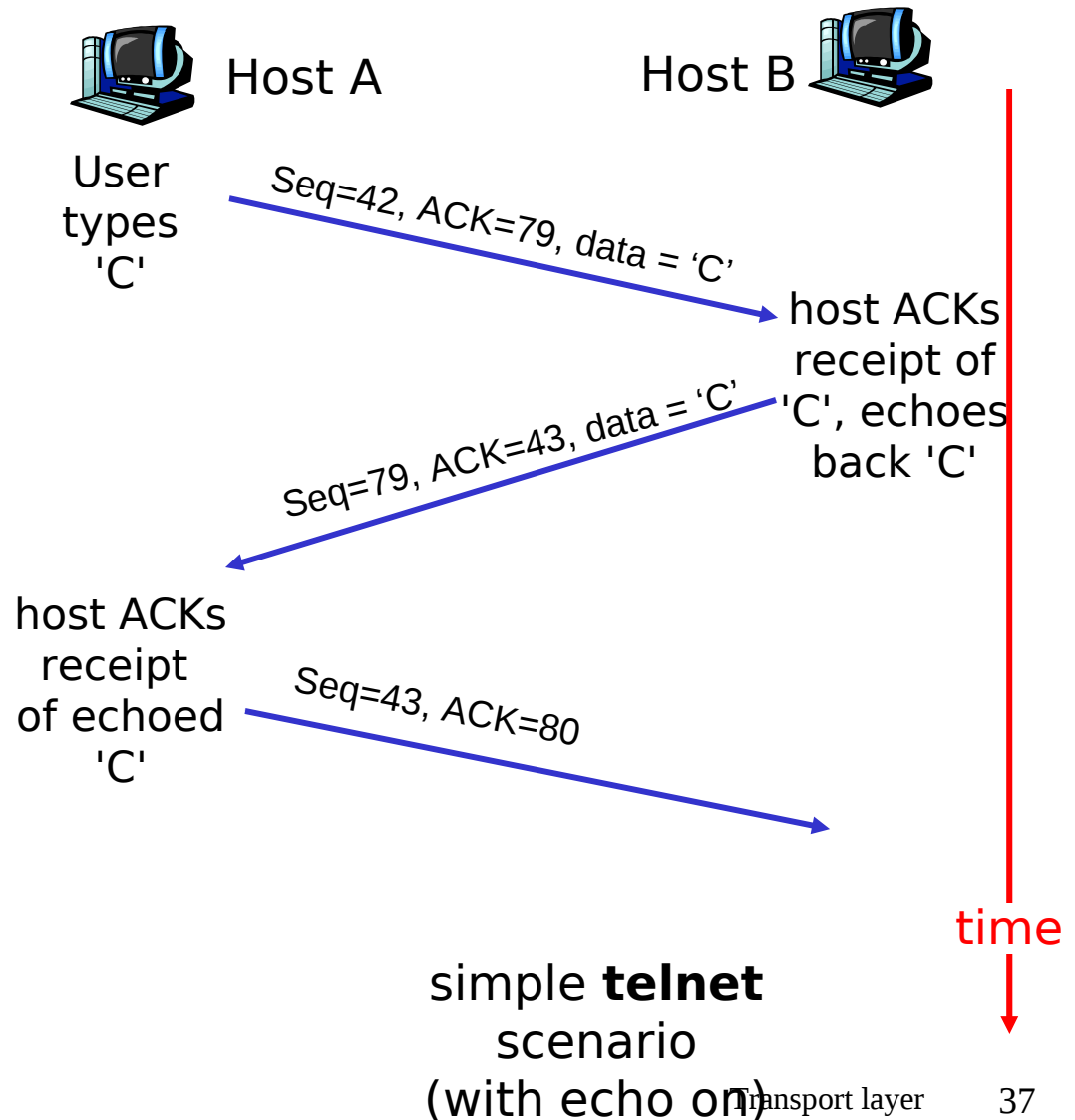
## ACKs (acknowledgment):

- seq # of next byte expected from other side (last byte +1)
- cumulative ACK
- If received segment 1, waits for segment 2
- Ack= $1000 + R$  (received up to 999<sup>th</sup> byte)

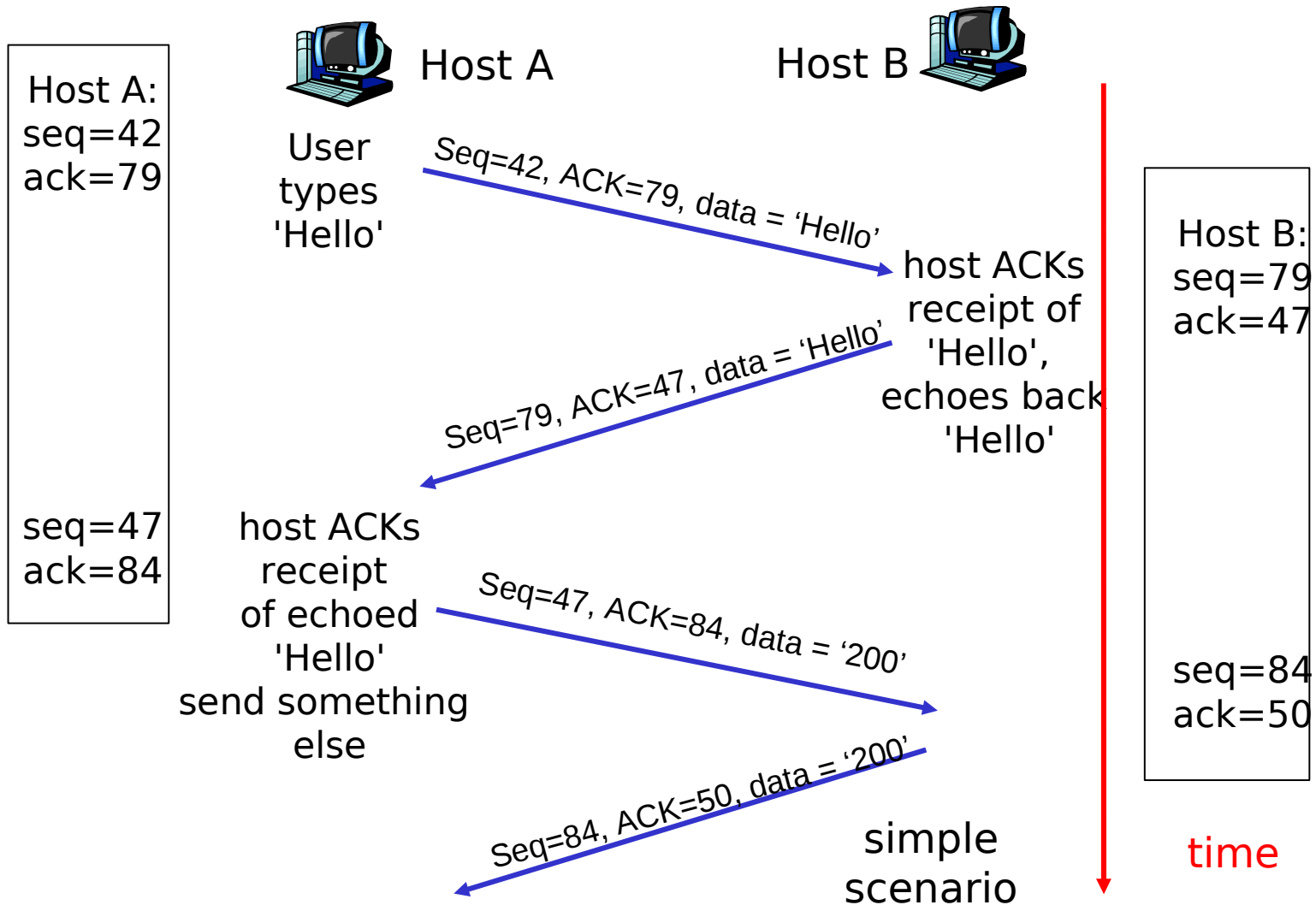
# TCP sequence #'s and ACKs

Q: how receiver handles out-of-order segments

□ A: TCP spec doesn't say, - decide when implementing



# Yet another example



# TCP: reliable data transfer

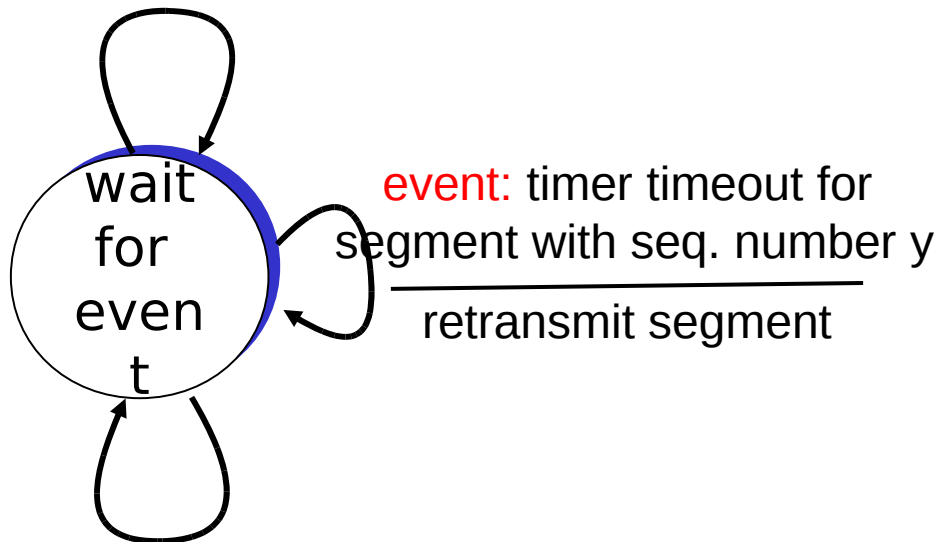
**event:** data received  
from application above  

---

create, send segment

simplified sender, assuming

- one way data transfer
- no flow, congestion control



**event:** ACK received,  
with ACK number y  

---

ACK processing

# TCP: reliable data transfer

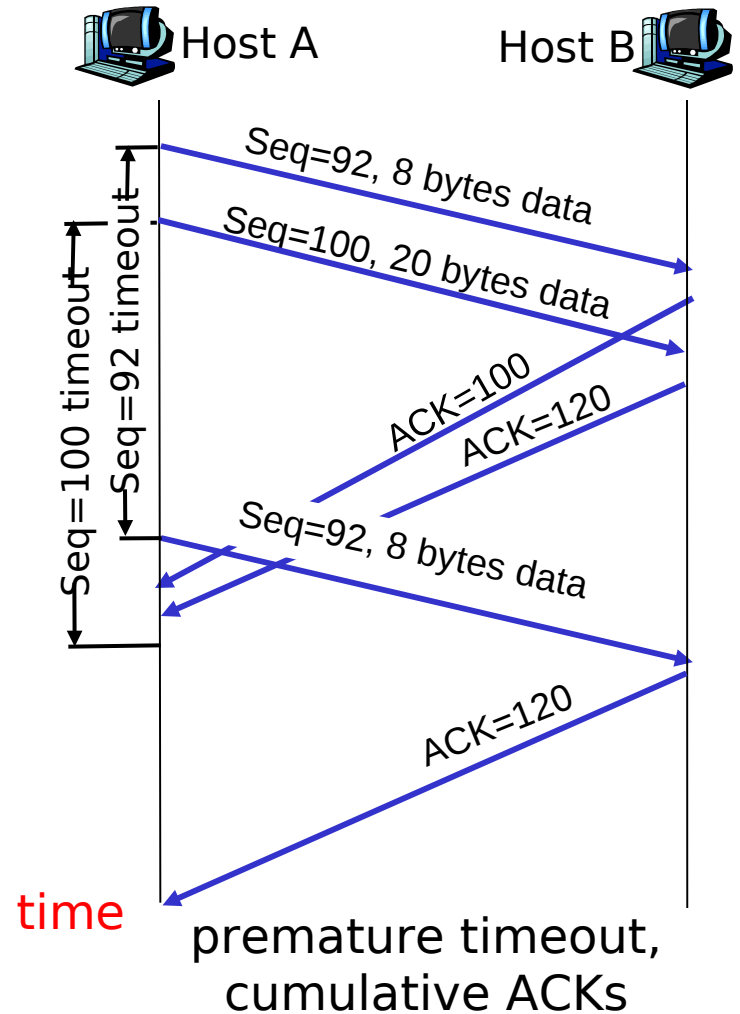
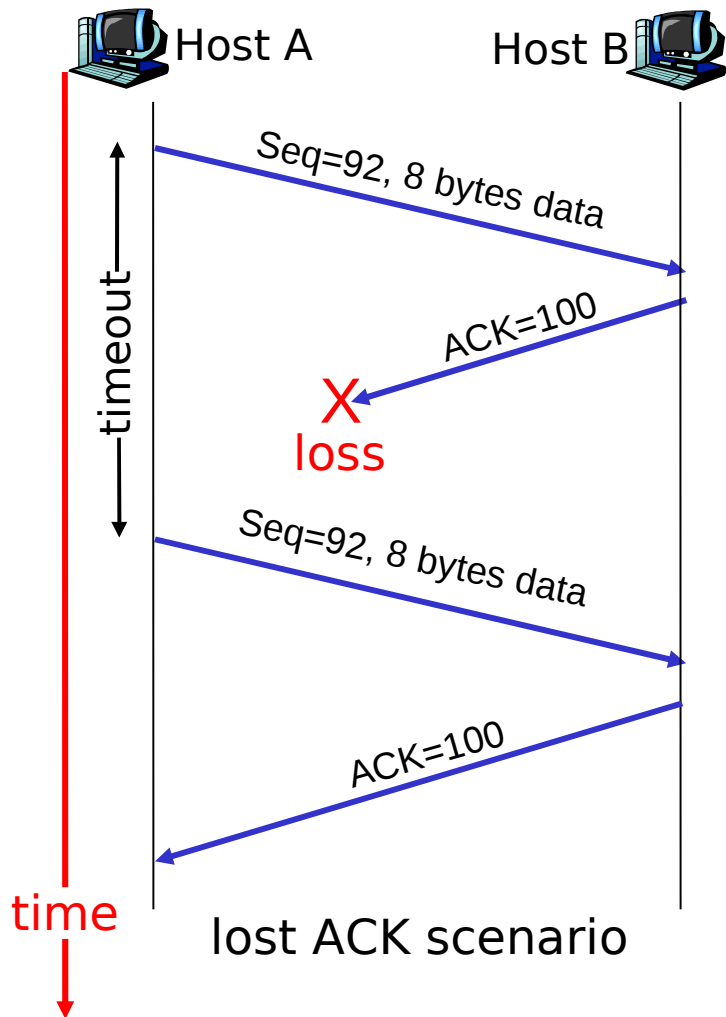
## Simplified TCP sender

```
00 sendbase = initial_sequence number
01 nextseqnum = initial_sequence number
02
03 loop (forever) {
04     switch(event)
05         event: data received from application above
06             create TCP segment with sequence number nextseqnum
07             start timer for segment nextseqnum
08             pass segment to IP
09             nextseqnum = nextseqnum + length(data)
10         event: timer timeout for segment with sequence number y
11             retransmit segment with sequence number y
12             compute new timeout interval for segment y
13             restart timer for sequence number y
14         event: ACK received, with ACK field value of y
15             if (y > sendbase) { /* cumulative ACK of all data up to y */
16                 cancel all timers for segments with sequence numbers < y
17                 sendbase = y
18             }
19             else { /* a duplicate ACK for already ACKed segment */
20                 increment number of duplicate ACKs received for y
21                 if (number of duplicate ACKS received for y == 3) {
22                     /* TCP fast retransmit */
23                     resend segment with sequence number y
24                     restart timer for segment y
25                 }
26             } /* end of loop forever */
```

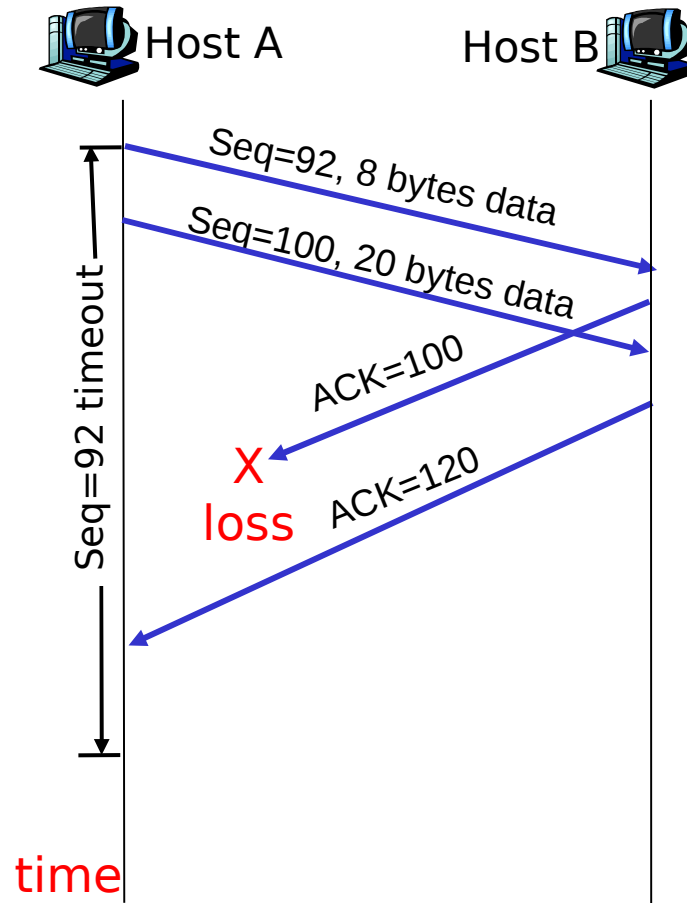
# TCP ACK generation [RFC 1122, RFC 2581]

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

# TCP: retransmission scenarios



# TCP: retransmission scenarios



cumulative ACKs,  
avoids retransmission of the first segment

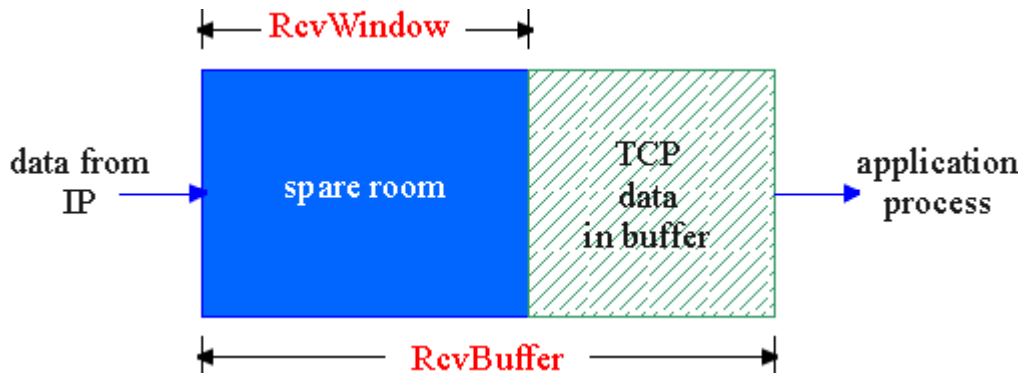
# TCP Flow Control

**flow control**  
sender won't overrun receiver's buffers by transmitting too much,

**too fast**

**RcvBuffer** = size of TCP Receive Buffer

**RcvWindow** = amount of spare room in Buffer



receiver buffering

**receiver:** explicitly informs sender of (dynamically changing) amount of free buffer space

- **RcvWindow field** in TCP segment

**sender:** keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

**question:** What happens when Rcv buffer is full?

Sender keeps sending 1 byte...

# TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT \*
  - note: RTT will vary
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

\* RTT = round trip time

Q: how to estimate RTT?

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

# TCP Round Trip Time and Timeout

$$\text{EstimatedRTT} = (1-x) * \text{EstimatedRTT} + x * \text{SampleRTT}$$

- Exponential weighted moving average
- influence of given sample decreases exponentially fast
- typical value of x: 0.125

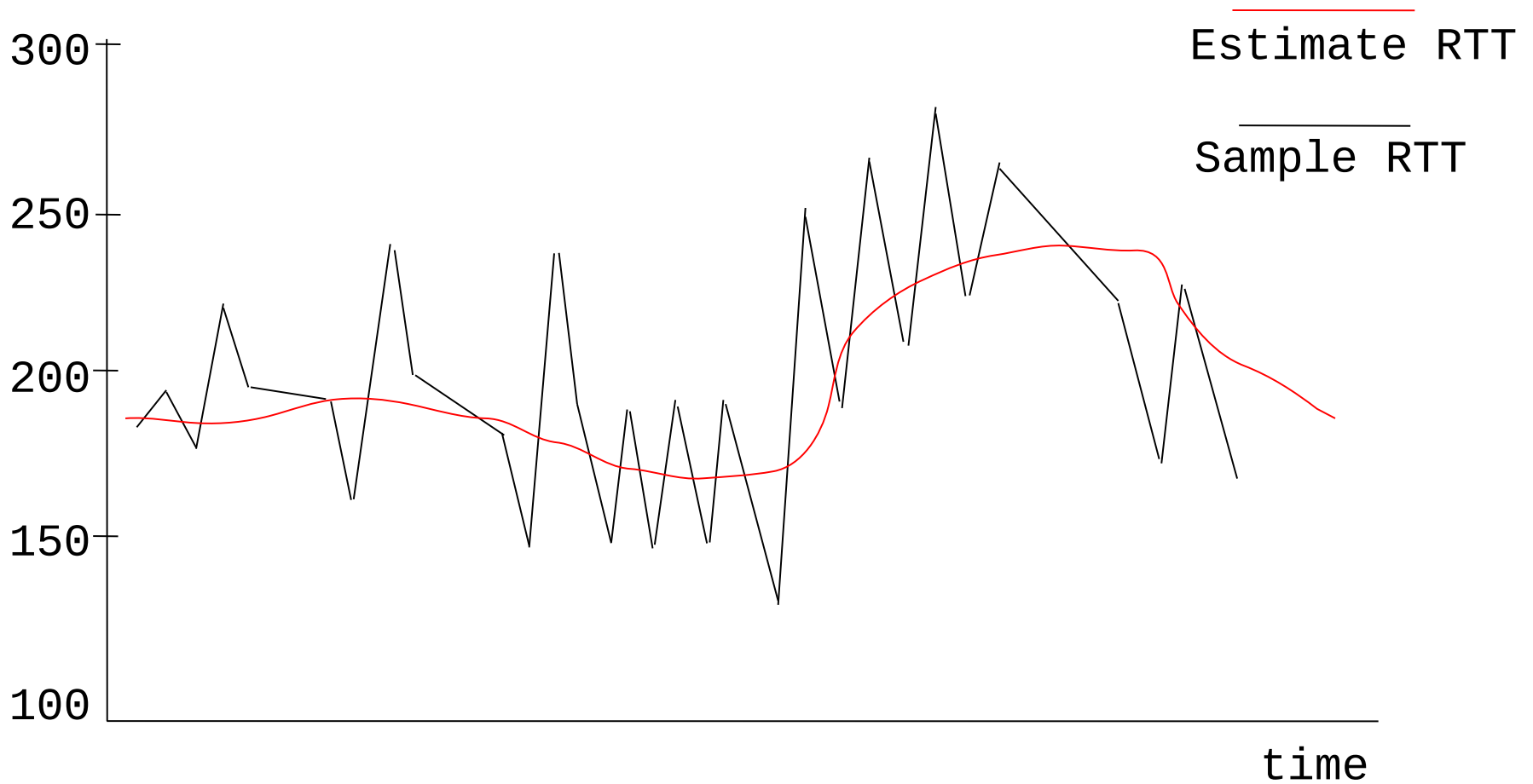
## Setting the timeout

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

$$\text{Timeout} = \text{EstimatedRTT} + 4 * \text{Deviation}$$

$$\text{Deviation} = (1-x) * \text{Deviation} + x * |\text{SampleRTT} - \text{EstimatedRTT}|$$

# TCP Round Trip Time and Timeout



# TCP Connection Management

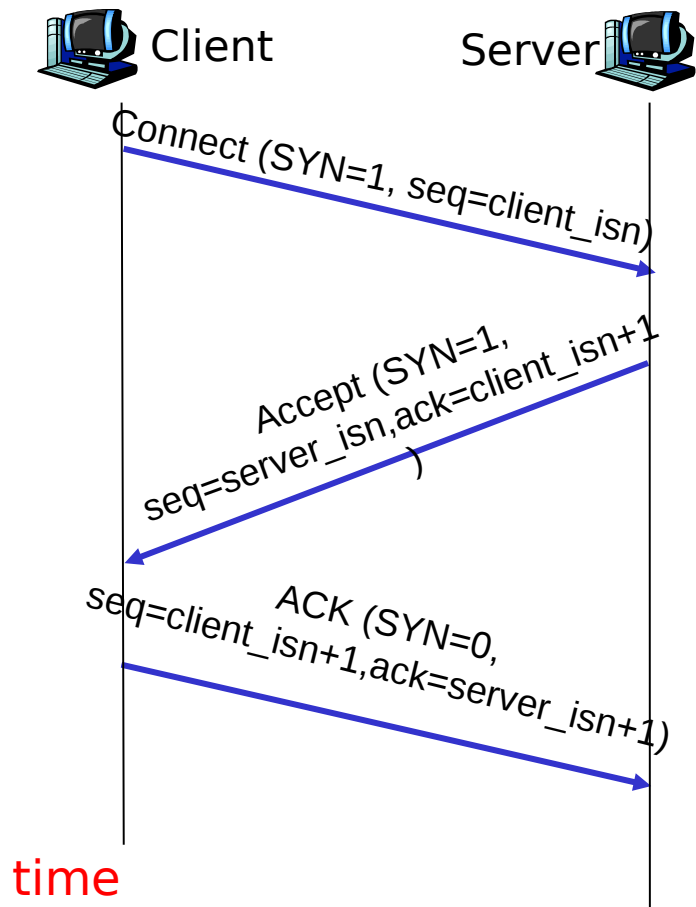
**Recall:** TCP sender, receiver establish “connection” before exchanging data segments

- initialize TCP variables:
  - sequence numbers
    - buffers, flow control info (e.g. **RcvWindow**)
- *client*: connection initiator

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

- *server*: contacted by client
- ```
Socket accept();
```

# TCP Connection Management



## Three way handshake:

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

- specifies initial seq number (isn)

Step 2: server end system receives SYN, replies with SYNACK control segment

- ACKs received SYN
- allocates buffers
- specifies server- $\rightarrow$  receiver initial seq. number

Step 3: client ACK the connection:

server\_isn + 1 and SYN=0

Connection established!

# TCP Connection Management (cont.)

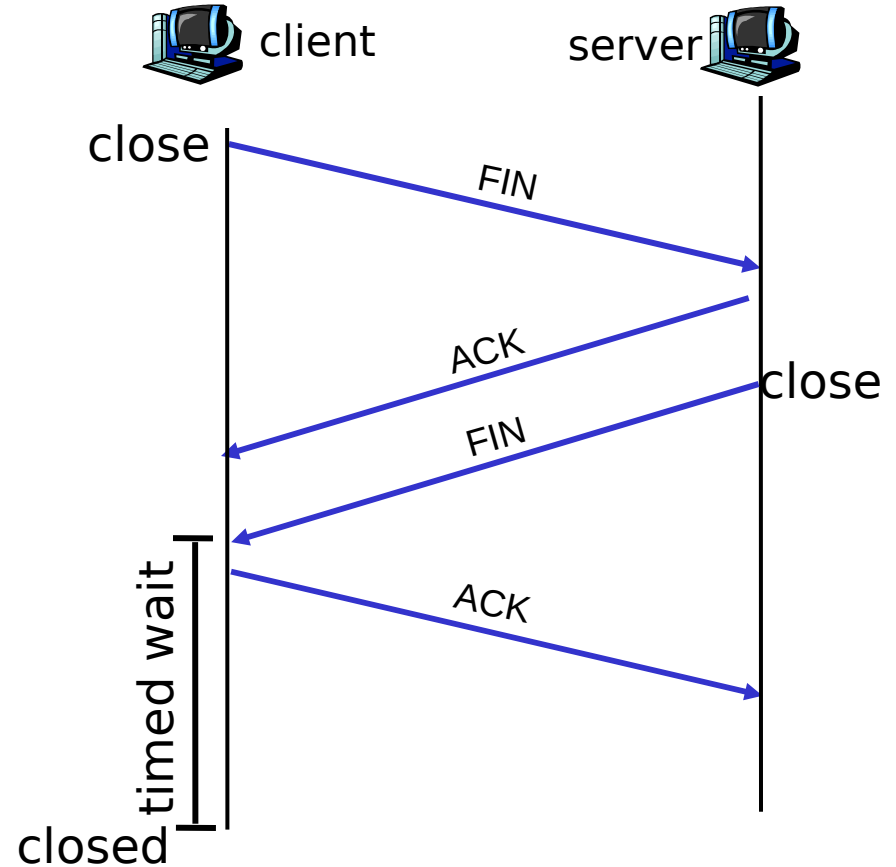
## Closing a connection:

client closes socket:

```
clientSocket.close();
```

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

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



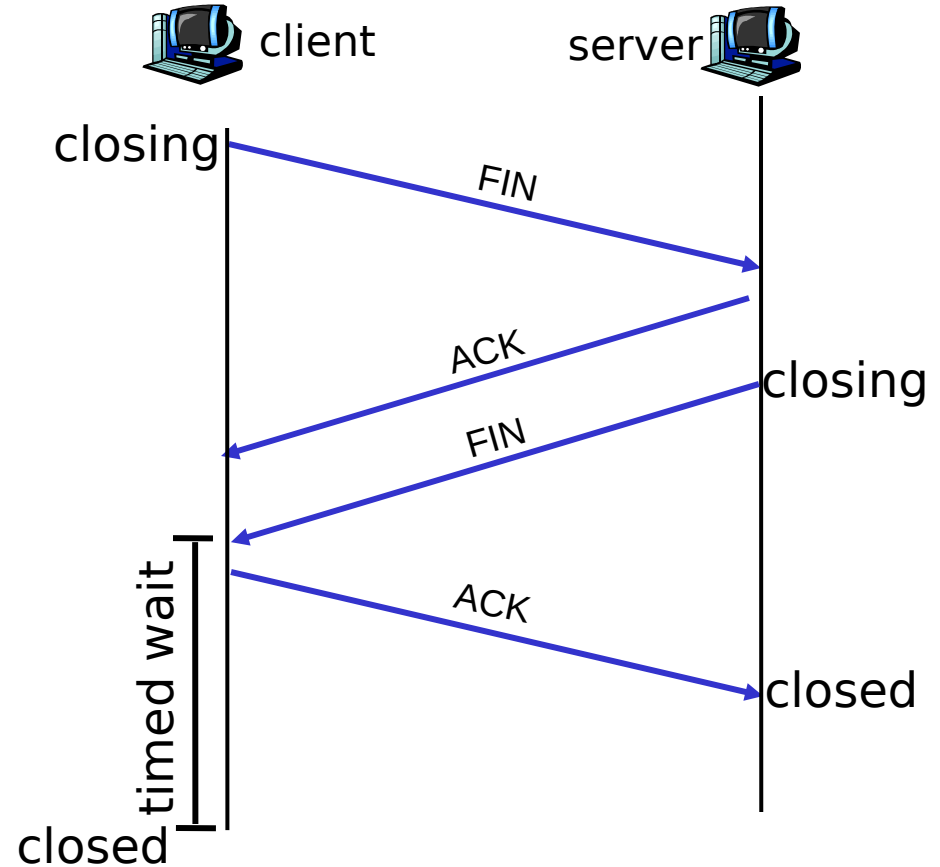
# TCP Connection Management (cont.)

**Step 3:** client receives FIN, replies with ACK.

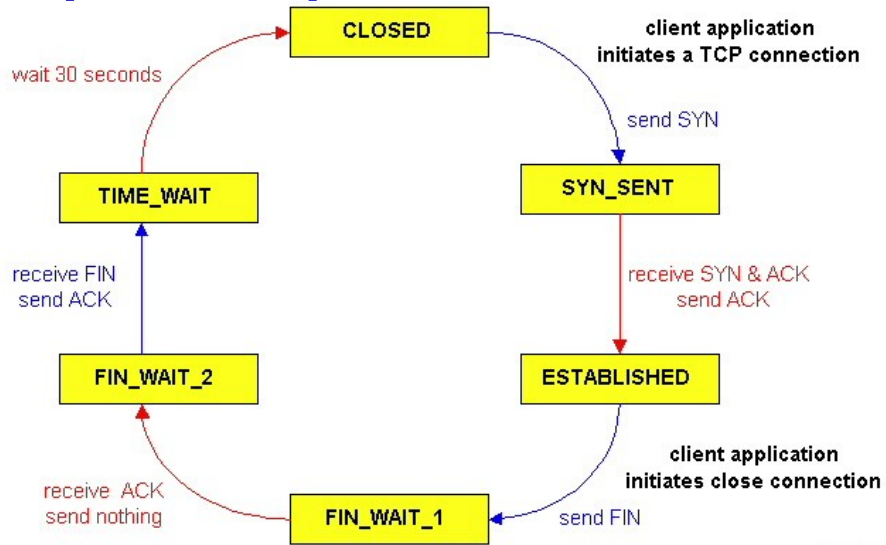
- ▣ Enters "timed wait" - will respond with ACK to received FINs

**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.

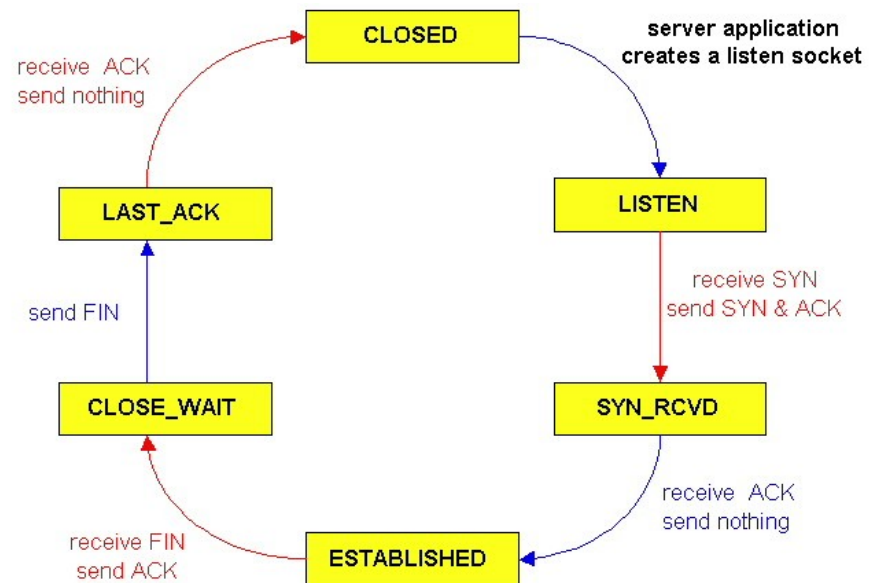


# TCP Connection Management (cont)



TCP **client** lifecycle

## TCP **server** lifecycle



# 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 (queuing in router buffers)
  - a top-10 problem!

# Approaches towards congestion control

Two broad approaches towards congestion control:

## End-end congestion control:

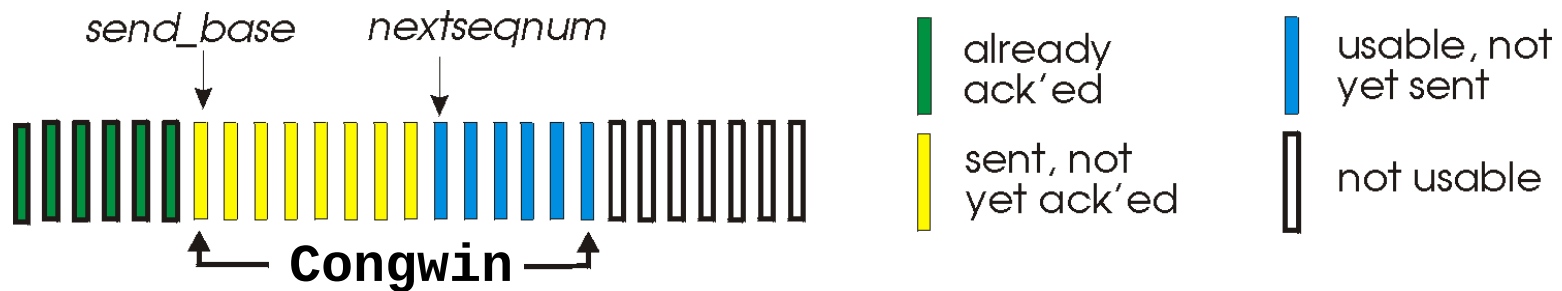
- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

## Network-assisted congestion control:

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

# TCP Congestion Control

- end-end control (no network assistance)
- transmission rate limited by congestion window size, **Congwin**, over segments:



- $w$  segments, each with MSS bytes sent in one RTT:

$$\text{throughput} = \frac{w * \text{MSS}}{\text{RTT}} \text{ Bytes/sec}$$

# TCP congestion control:

- "probing" for usable bandwidth:
  - *ideally*: transmit as fast as possible (**Congwin** as large as possible) without loss
  - *increase* **Congwin** until loss (congestion)
  - loss: *decrease* **Congwin**, then begin probing (increasing) again
- Two "phases"
  - *slow start*
  - *congestion avoidance*
- important variables:
  - **Congwin**
  - **threshold**: defines threshold between two slow start phase, congestion control phase

# TCP Slowstart

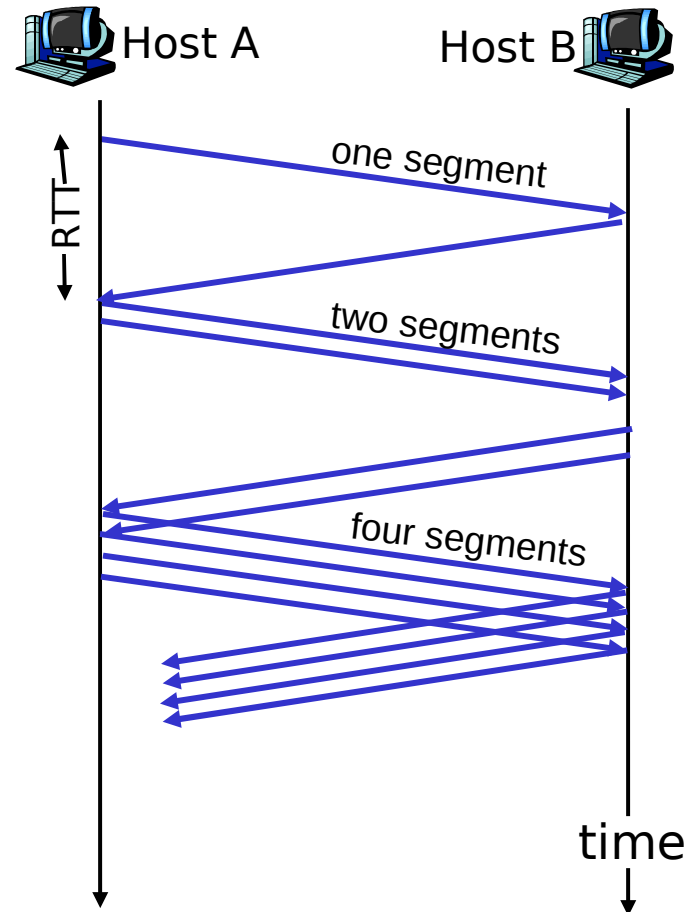
## Slowstart algorithm

initialize: Congwin = 1  
for (each transmission completed)

Congwin = Congwin \* 2

until (loss event OR  
CongWin > threshold)

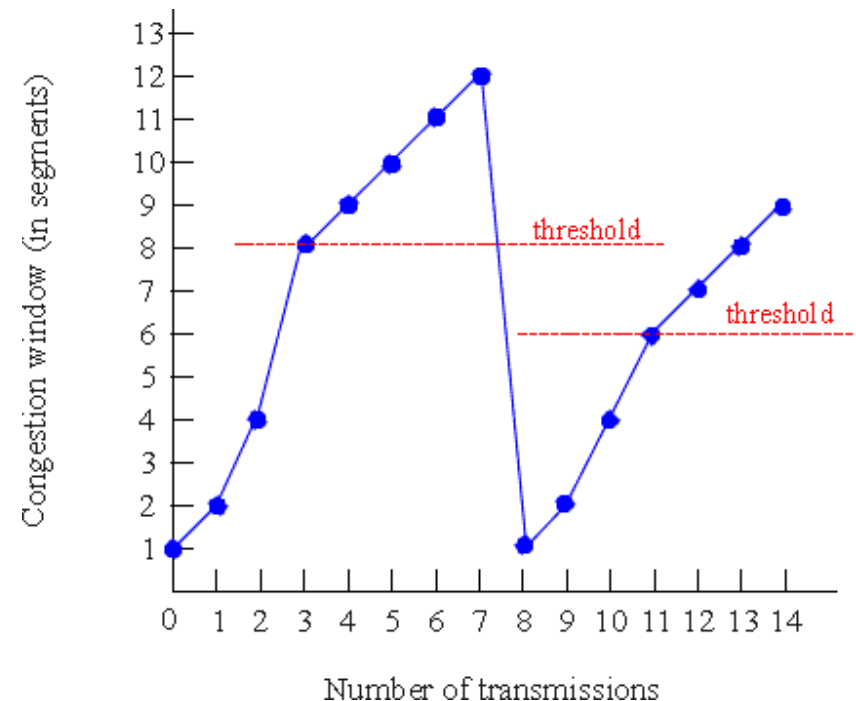
- exponential increase (per RTT) in window size (not so slow!)
- **loss event**: timeout and/or or three duplicate ACKs



# TCP Congestion Avoidance

## Congestion avoidance

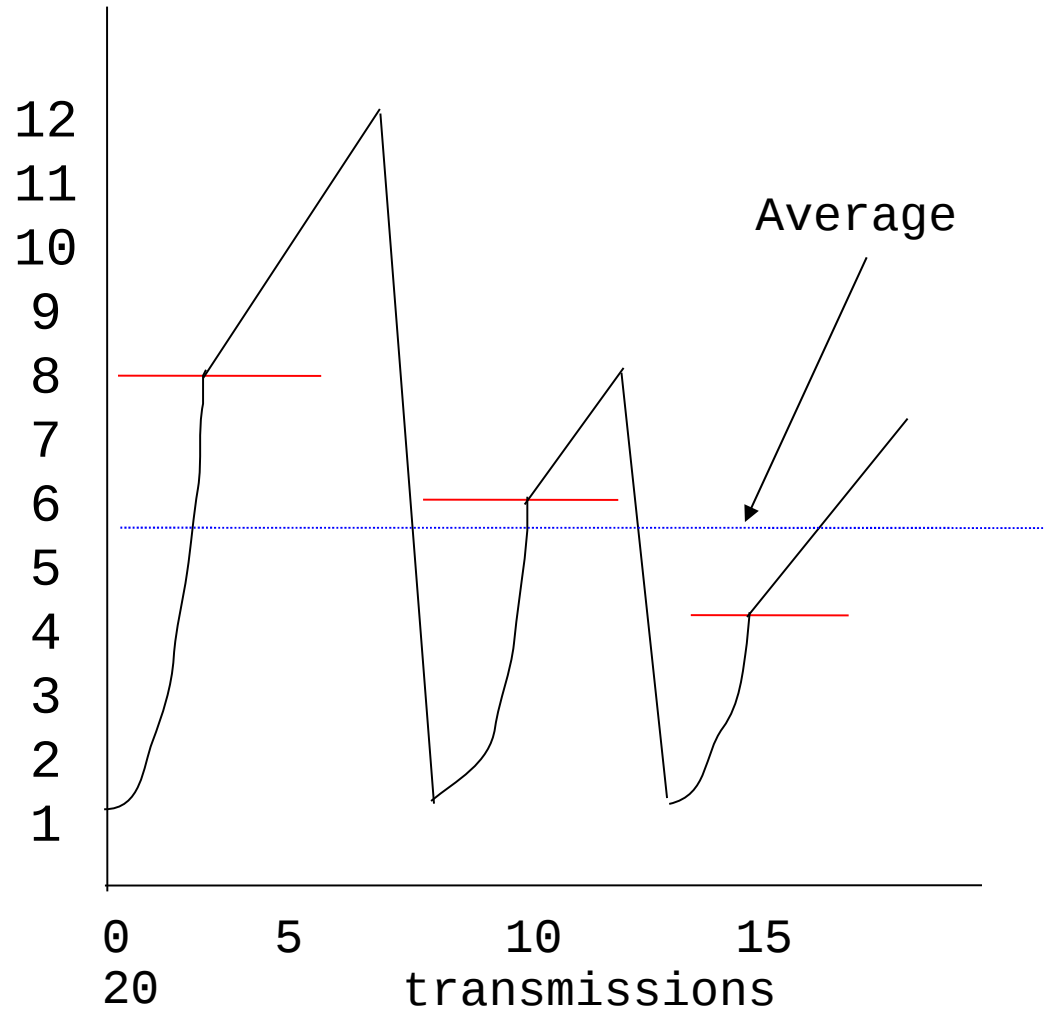
```
/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
    every w segments ACKed:
        Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart1
```



# AIMD

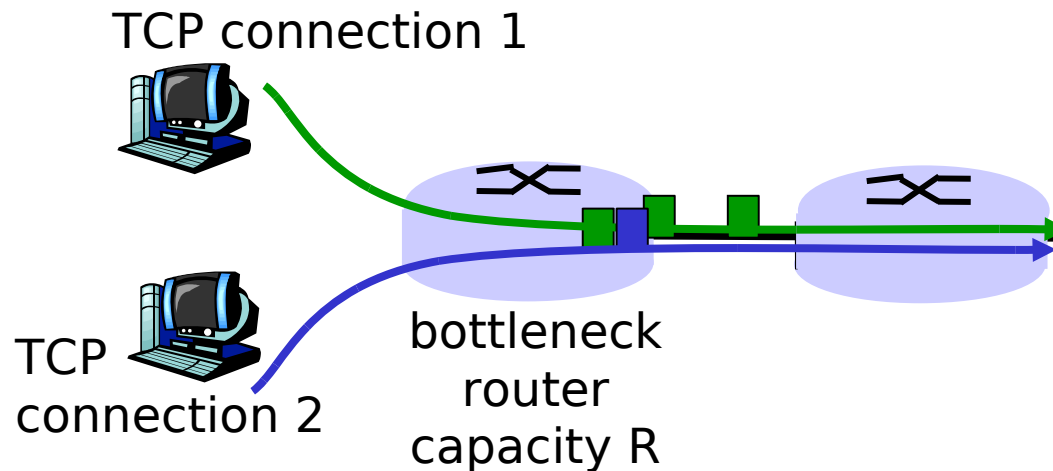
TCP congestion avoidance:

- **AIMD**: *additive increase, multiplicative decrease*
  - increase window by 1 per RTT
  - decrease window by factor of 2 on loss event



# TCP Fairness

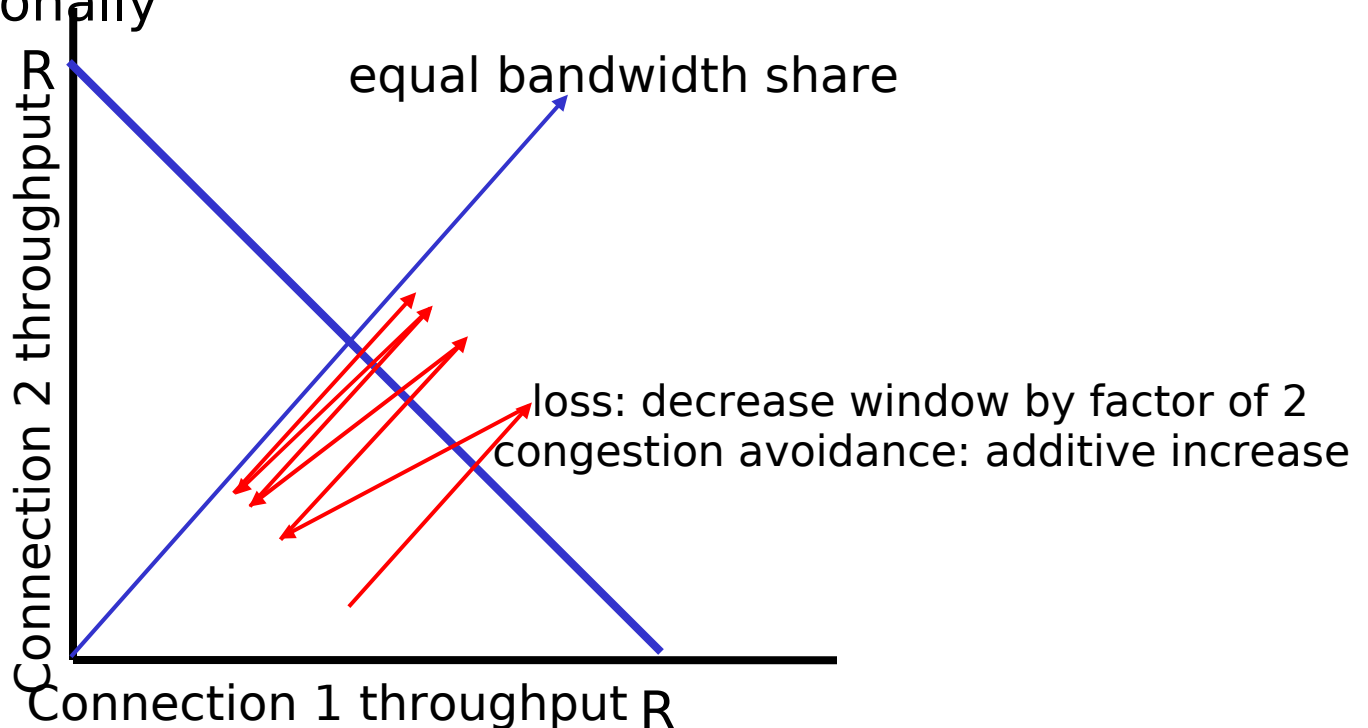
**Fairness goal:** if  $N$  TCP sessions share same bottleneck link, each should get  $1/N$  of link capacity



# Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- multiplicative decrease: decreases throughput proportionally



# TCP latency modeling

Q: How long does it take to receive an object from a Web server after sending a request?

- TCP connection establishment
- data transfer delay

## Notation, assumptions:

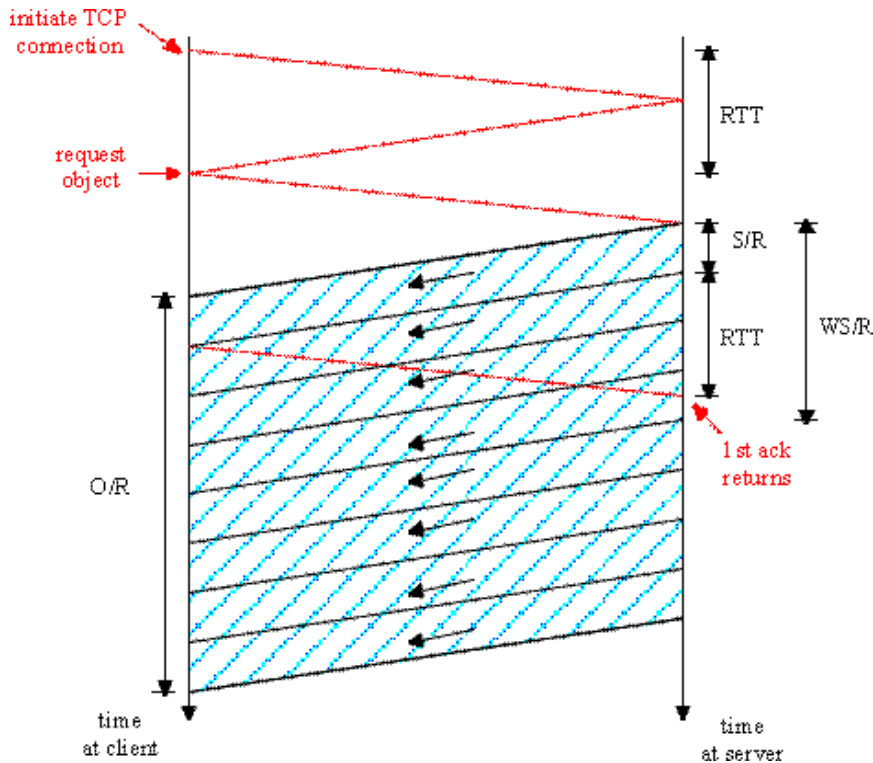
- Assume one link between client and server of rate  $R$
- Assume: fixed congestion window,  $W$  segments
- $S$ : MSS (bits)
- $O$ : object size (bits)
- no retransmissions (no loss, no corruption)

## Two cases to consider:

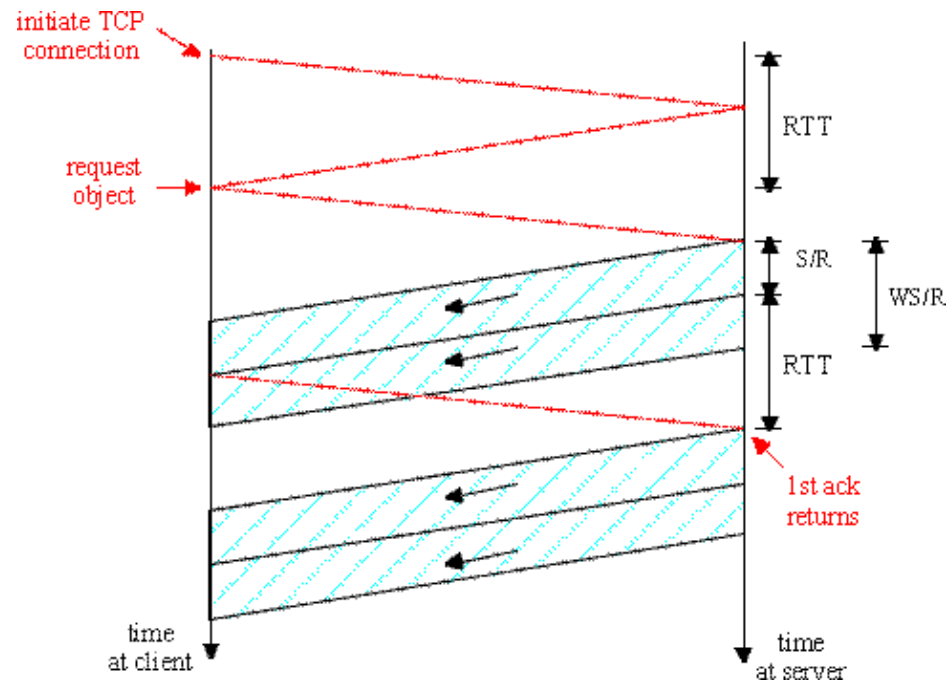
- $WS/R > RTT + S/R$ : ACK for first segment in window returns before window's worth of data sent
- $WS/R < RTT + S/R$ : wait for ACK after sending window's worth of data sent

# TCP latency Modeling

$$K := O/WS$$



Case 1: latency =  $2RTT + O/R$



Case 2: latency =  $2RTT + O/R + (K-1)[S/R + RTT - WS/R]$