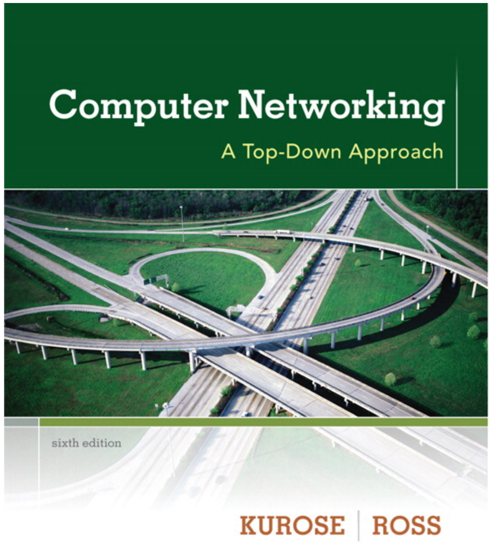
Chapter 4
Transport Layer



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

Chapter 4: 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

Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

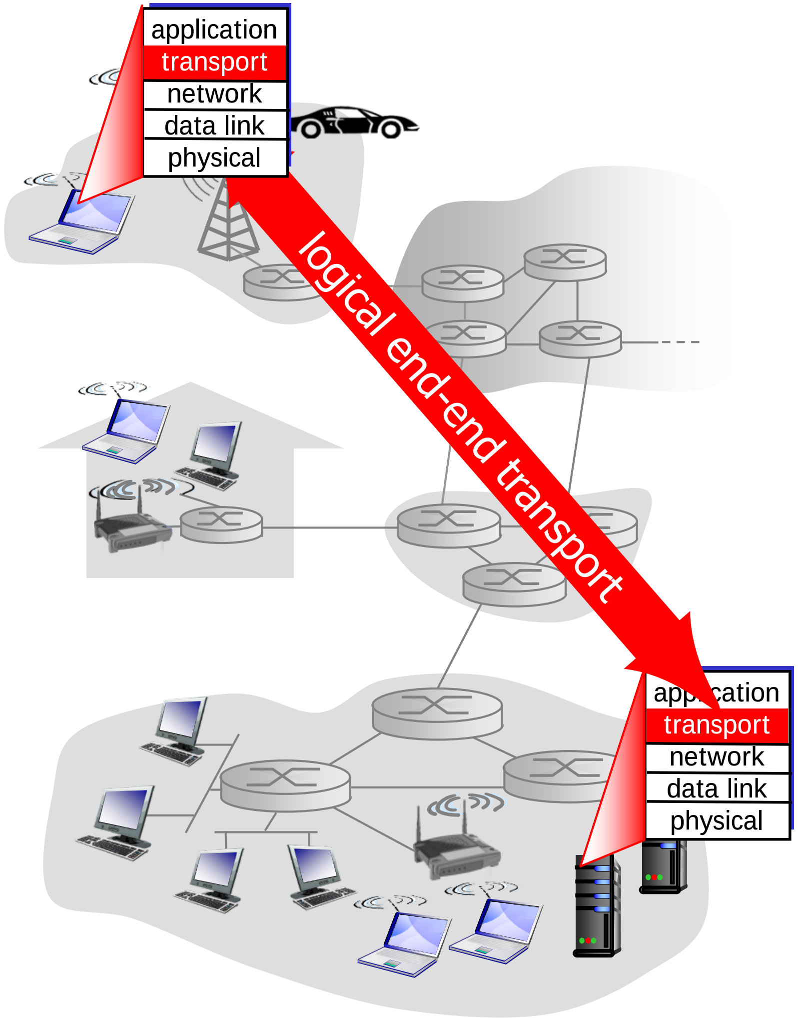
4.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

4.6 principles of congestion control

4.7 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
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

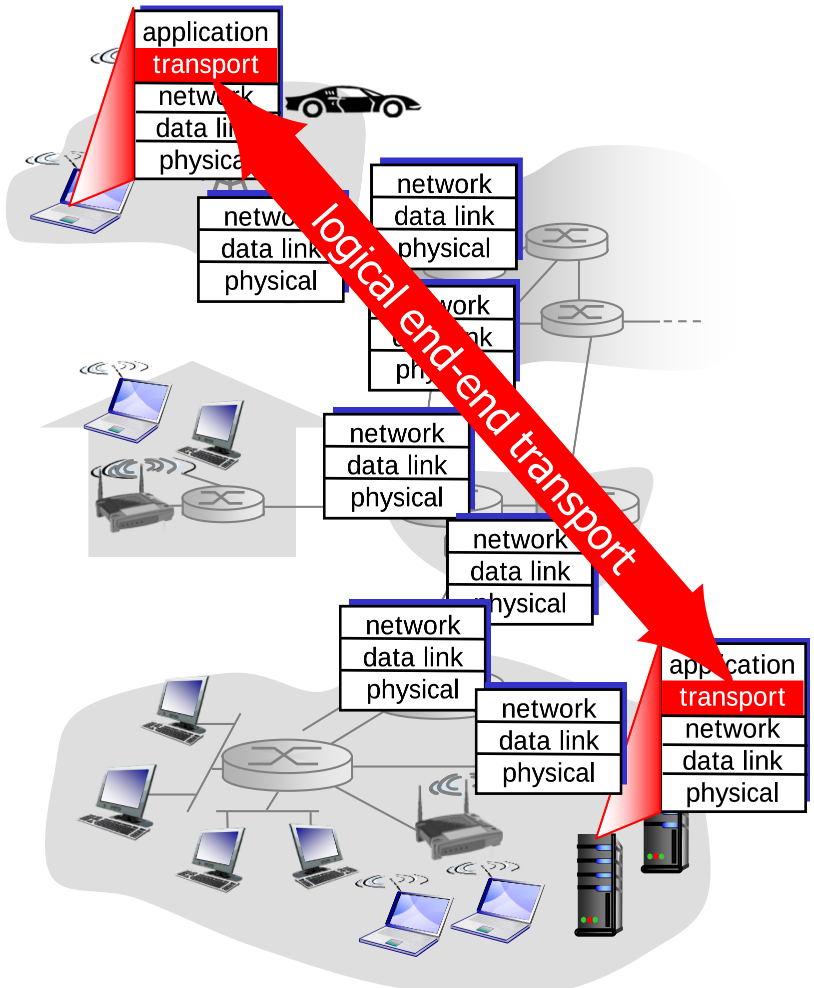
household analogy:

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

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 available:
  - delay guarantees
  - bandwidth guarantees



Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

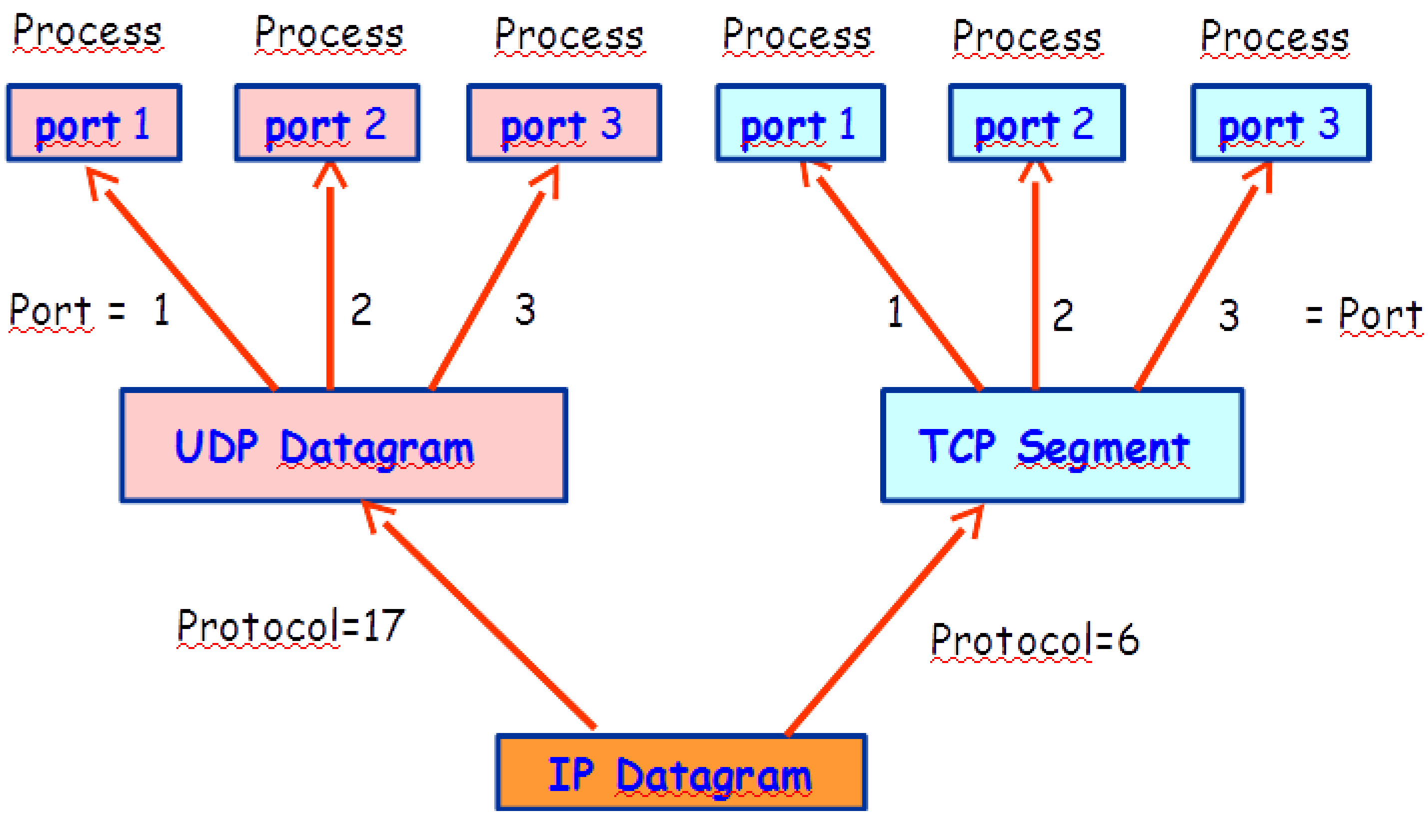
4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

4.6 principles of congestion control

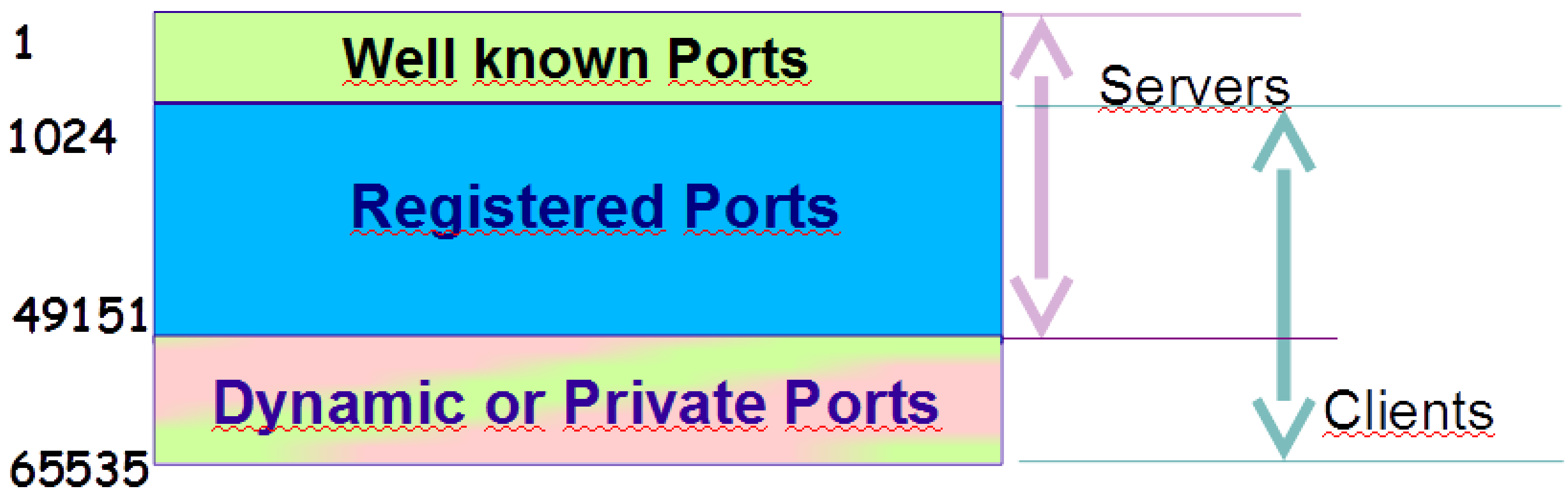
4.7 TCP congestion control

Addressing of Transport Layer



Addressing of Transport Layer

- Source/ Destination Port:
  - 16-bits number
    - There are 65,535 possible port numbers (2 to the power of 16 minus 1

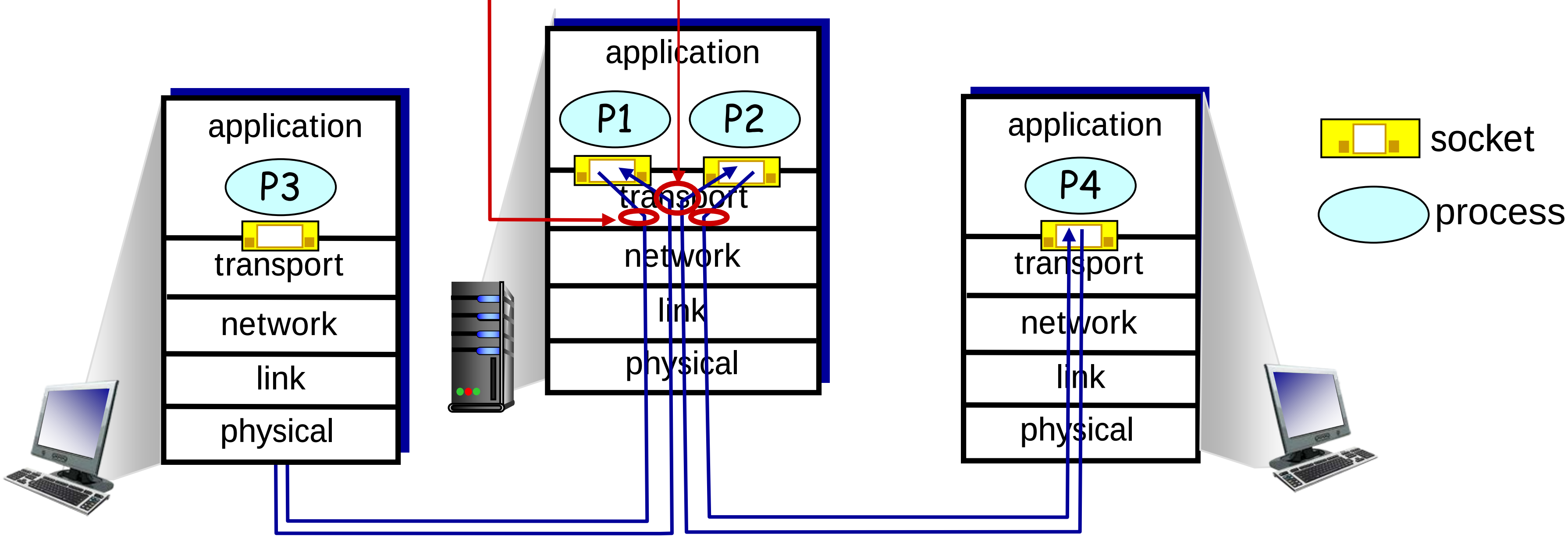


http://www.iana.org/assignments/port-numbers

Multiplexing/demultiplexing

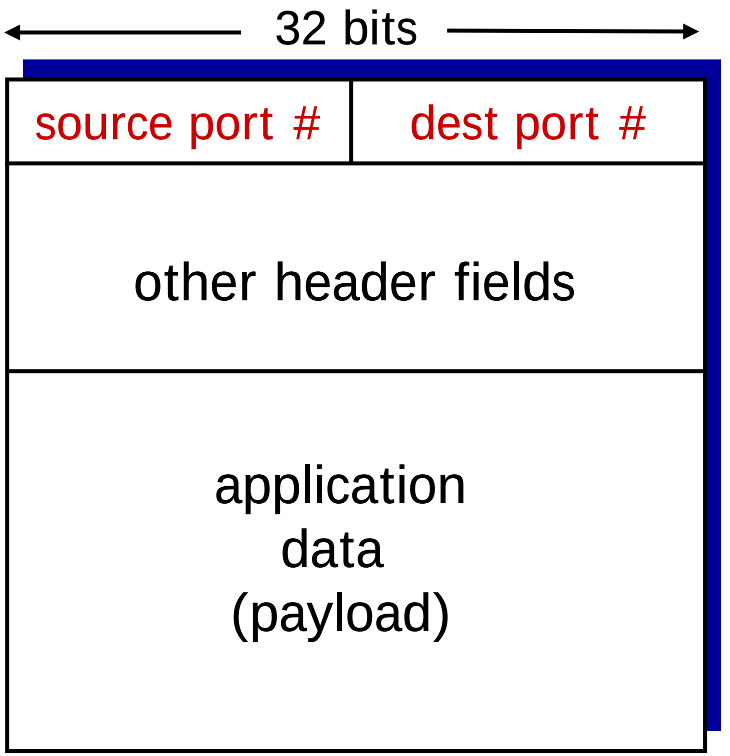
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

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

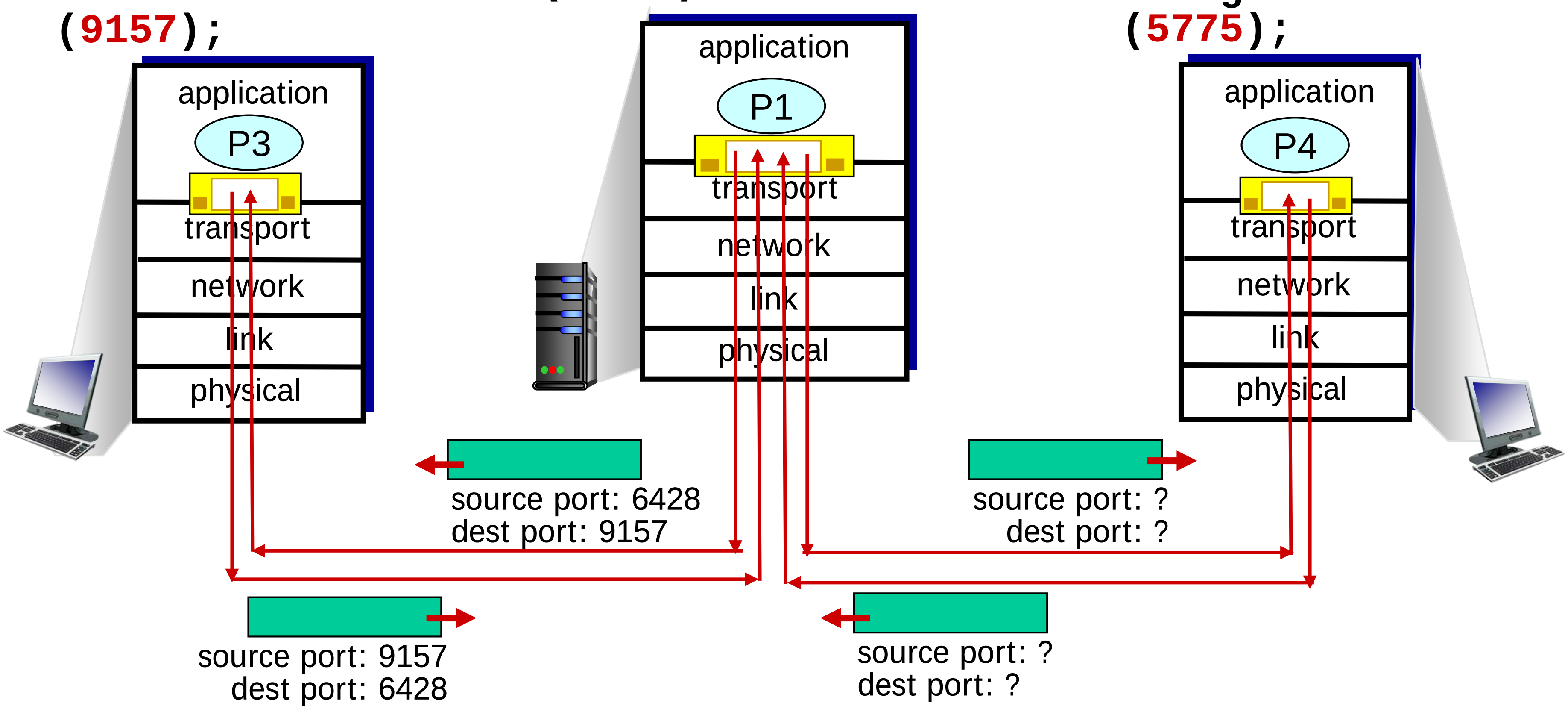
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
 serverSocket = new
 DatagramSocket
 (6428);
```

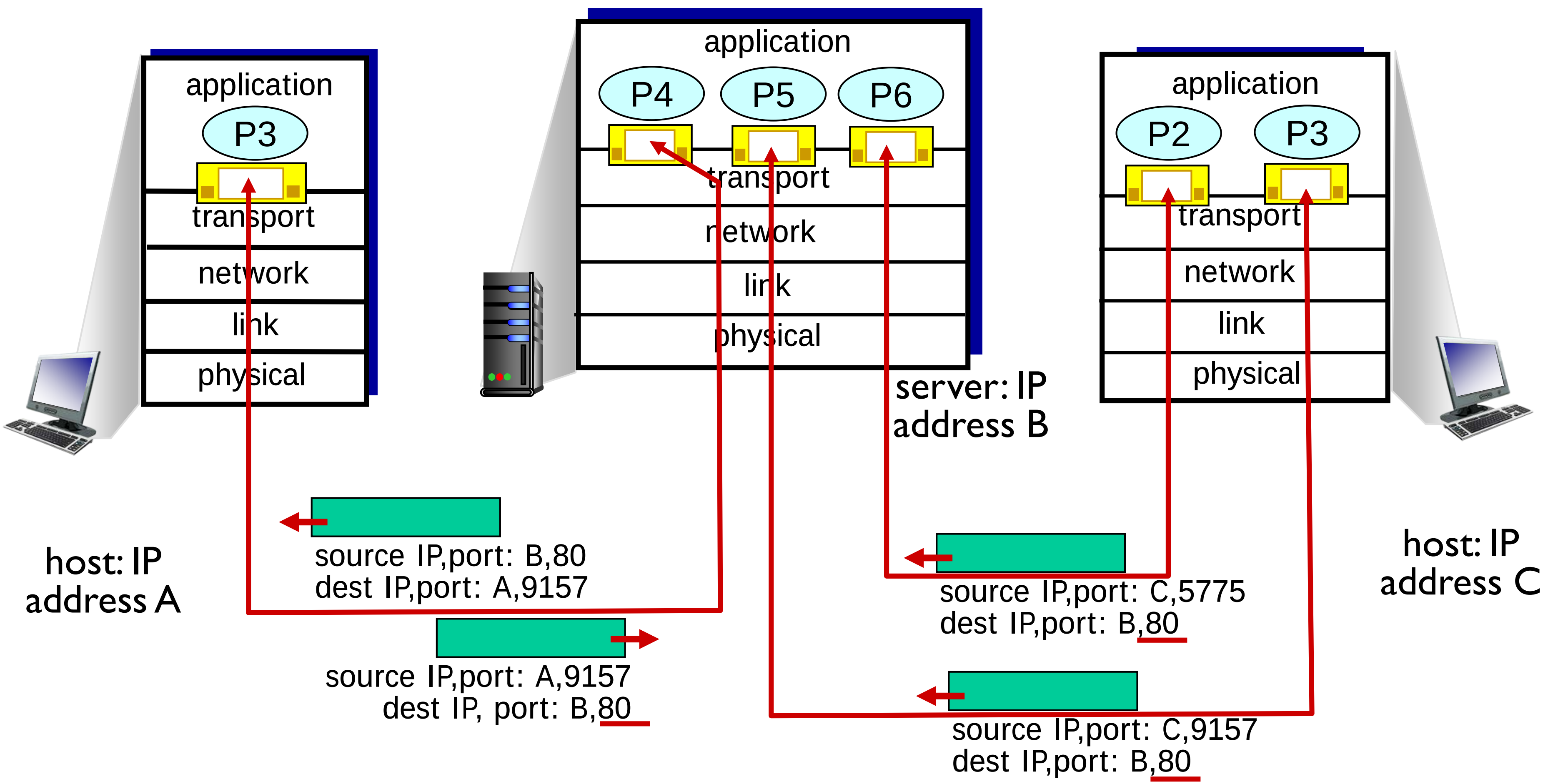
```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



Connection-oriented demux

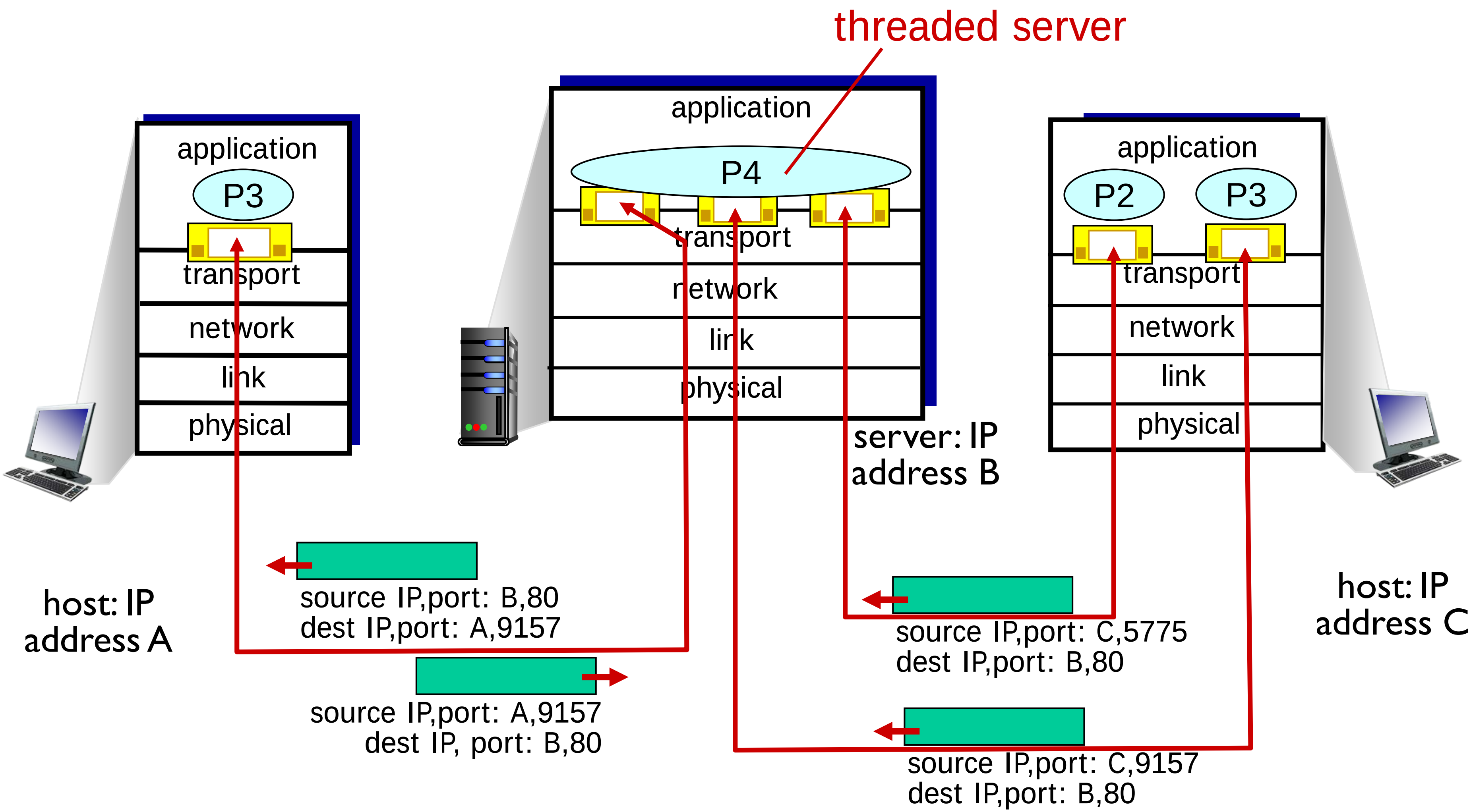
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- 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

Connection-oriented demux: example



Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

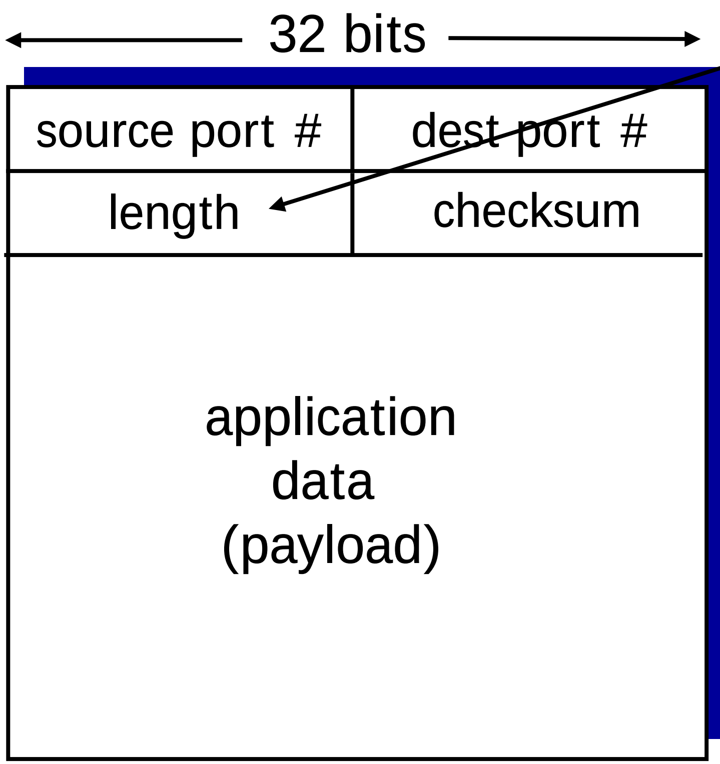
4.6 principles of congestion control

4.7 TCP congestion control

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP datagram may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP datagram handled independently of others
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: datagram header



UDP datagram format

length, in bytes of UDP datagram, including header (16 bits field->Max. 64KB)

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 checksum

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

sender:

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

receiver:

- compute checksum of received datagram
- 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

one's complement sum

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

Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

4.6 principles of congestion control

4.7 TCP congestion control

Principles of reliable data transfer

- Reliable data transfer s a problem that appears in application, transport, link layers
  - PROBLEM:
    - How can we achieve a reliable data transfer over an unreliable networks?
      - underlying layer can loss packets or may flip bits in transmitted packet
  - SOLUTION:
    - Detection
      - checksum to detect bit errors
    - Retransmission

Perfect Channel and Real Channel

- **Perfect Channel**:
  - underlying channel perfectly reliable
    - no bit errors
    - no loss of packets
- **Real Channel**:
  - Transmission error, congestion, routing errors, etc,
  - Receiver:
    - Is the received packet correct?
    - What can the receiver do if the packet isn't correct?
  - Sender:
    - was the packet correctly received?

Solution: ARQ (Automatic Repeat reQuest)

- error detection
- feedback: control msgs (ACK (Acknowledgment)) from receiver to sender
- Retransmission in case of failure
  - If ACK is not received before RTO (*retransmission timeout)* the packet is *retransmited*

ACK

sender:

what happens if ACK doesn't arrive?

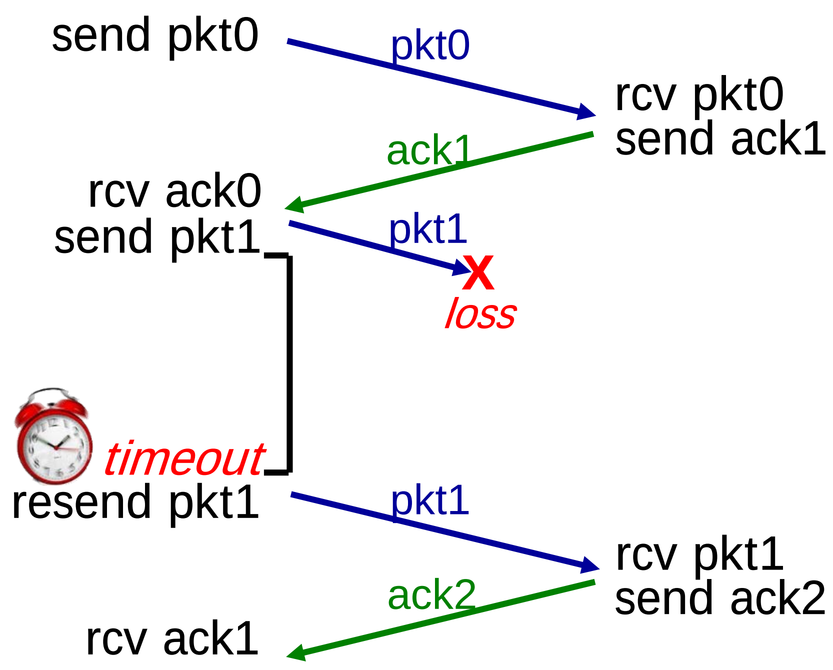
- packet loss?
- ack loss?
- sender doesn't know what happened at receiver!

receiver:

- must check if received packet is corrupted
  - Error detection (Checksum)

Packet loss

- sender waits "reasonable" amount of time for ACK
- retransmits if no ACK received in this time



Duplicates

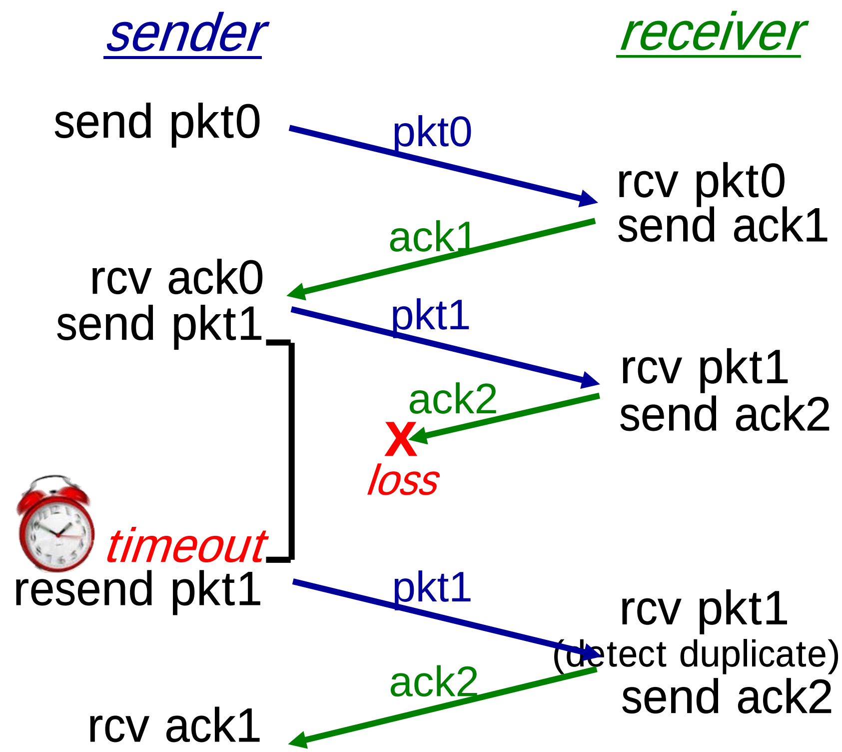
Handling duplicates:

sender:

- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate
  - sender adds sequence number to each pkt
  - receiver discards (doesn't deliver up) duplicate pkt

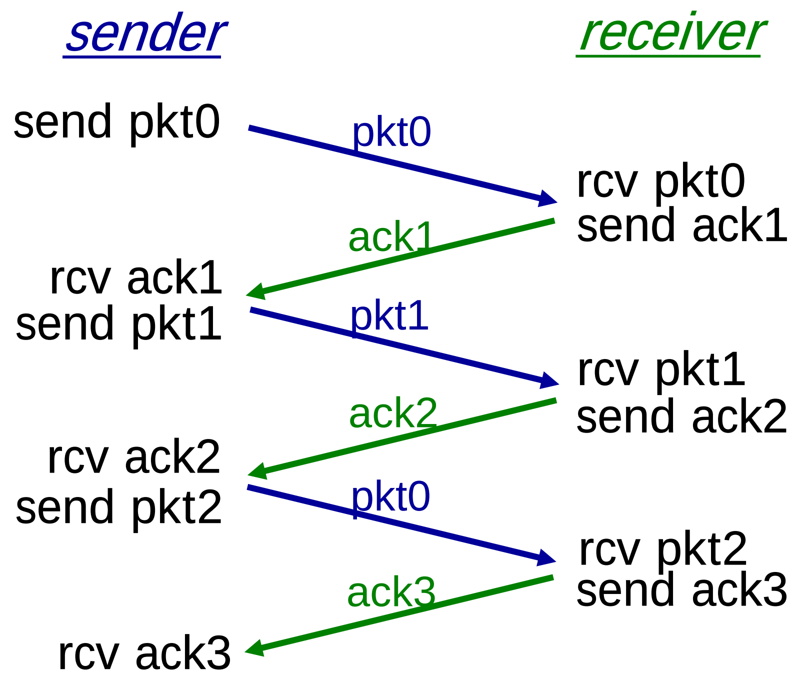
receiver:

  - receiver must specify seq # of pkt it is waiting



Stop and Wait

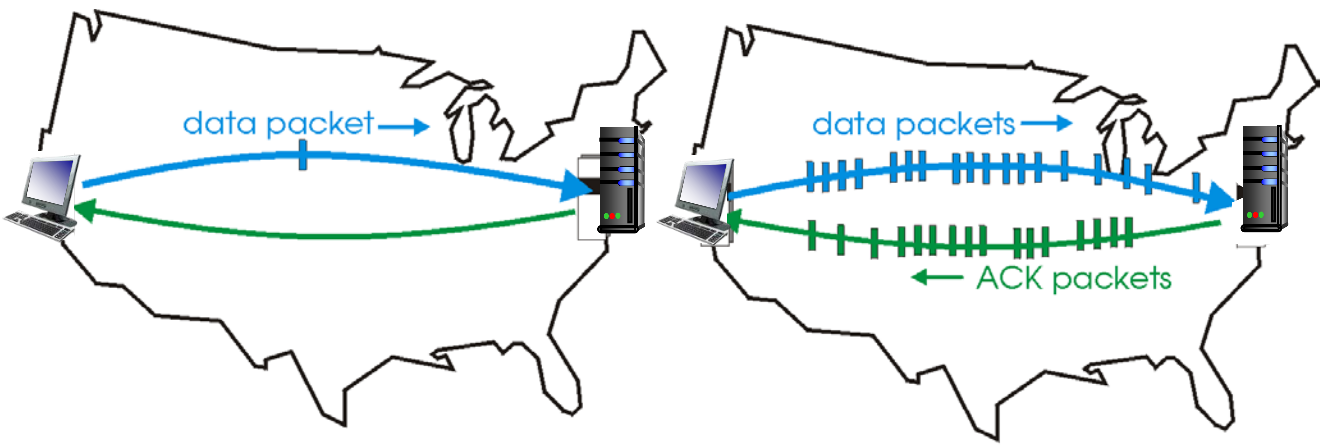
- sender sends one packet, then waits for receiver response
- Simple but inefficient!
- network protocol limits use of physical resources!



Pipelined protocols

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

- 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*

Pipelined protocols: overview

Go-back-N:

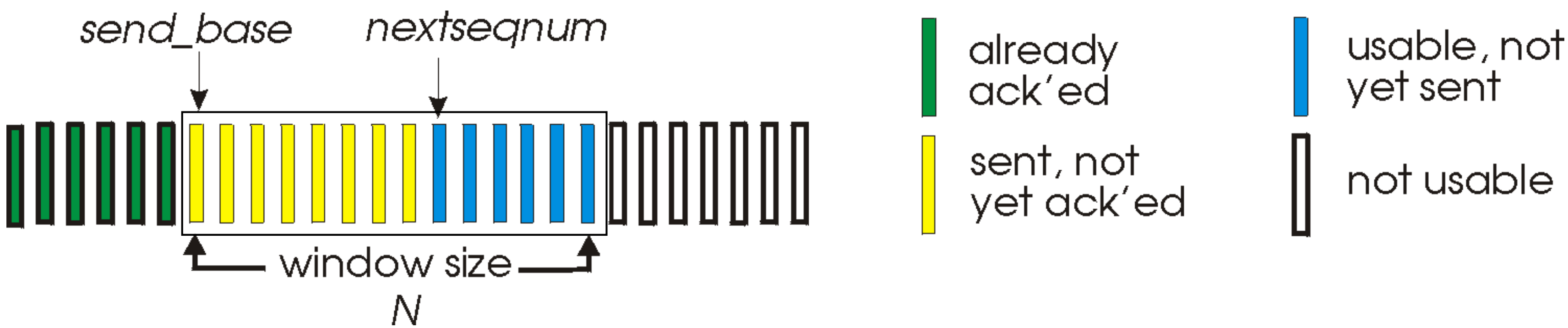
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unacked packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N: sender

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

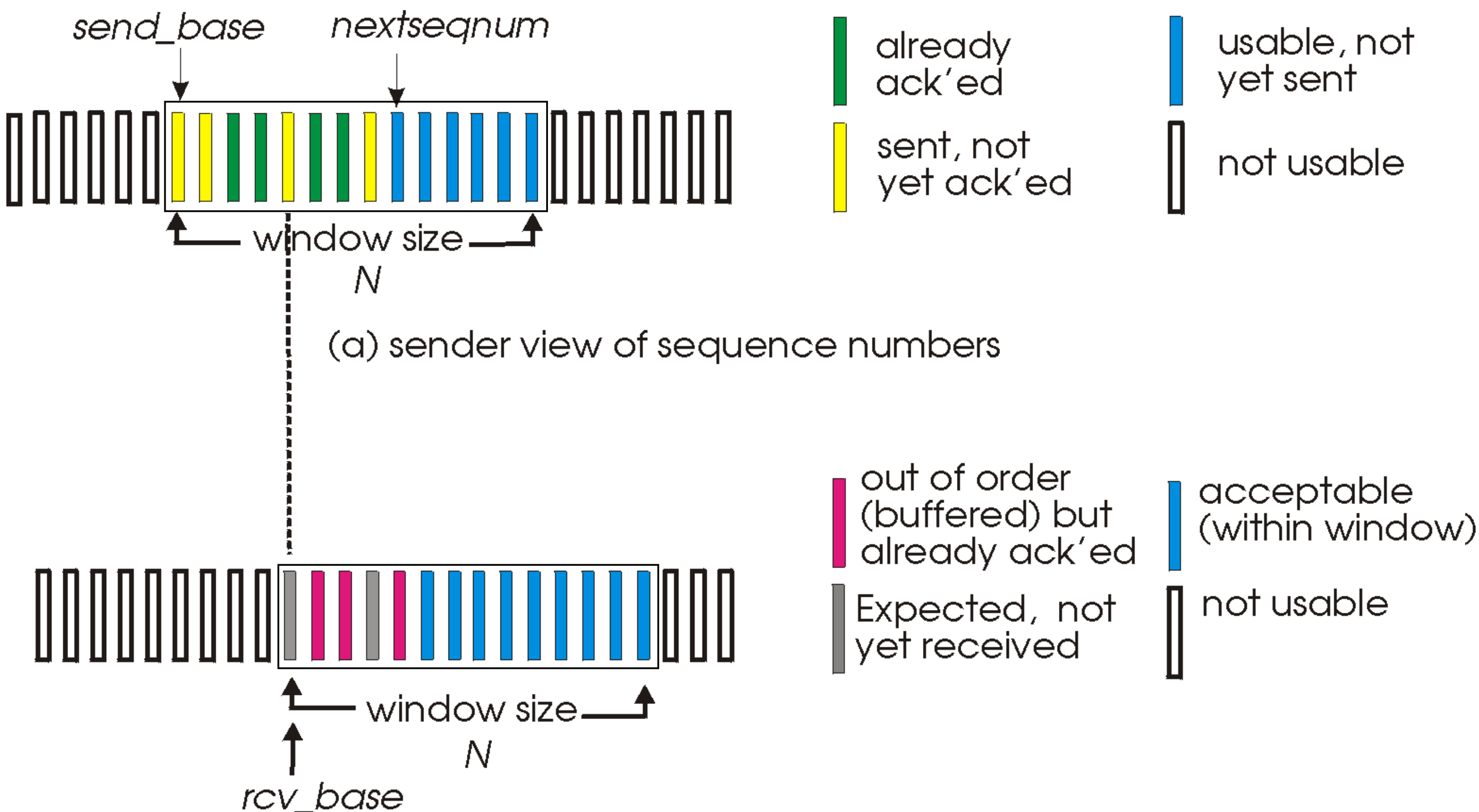


- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

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
  - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers

(b) receiver view of sequence numbers

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

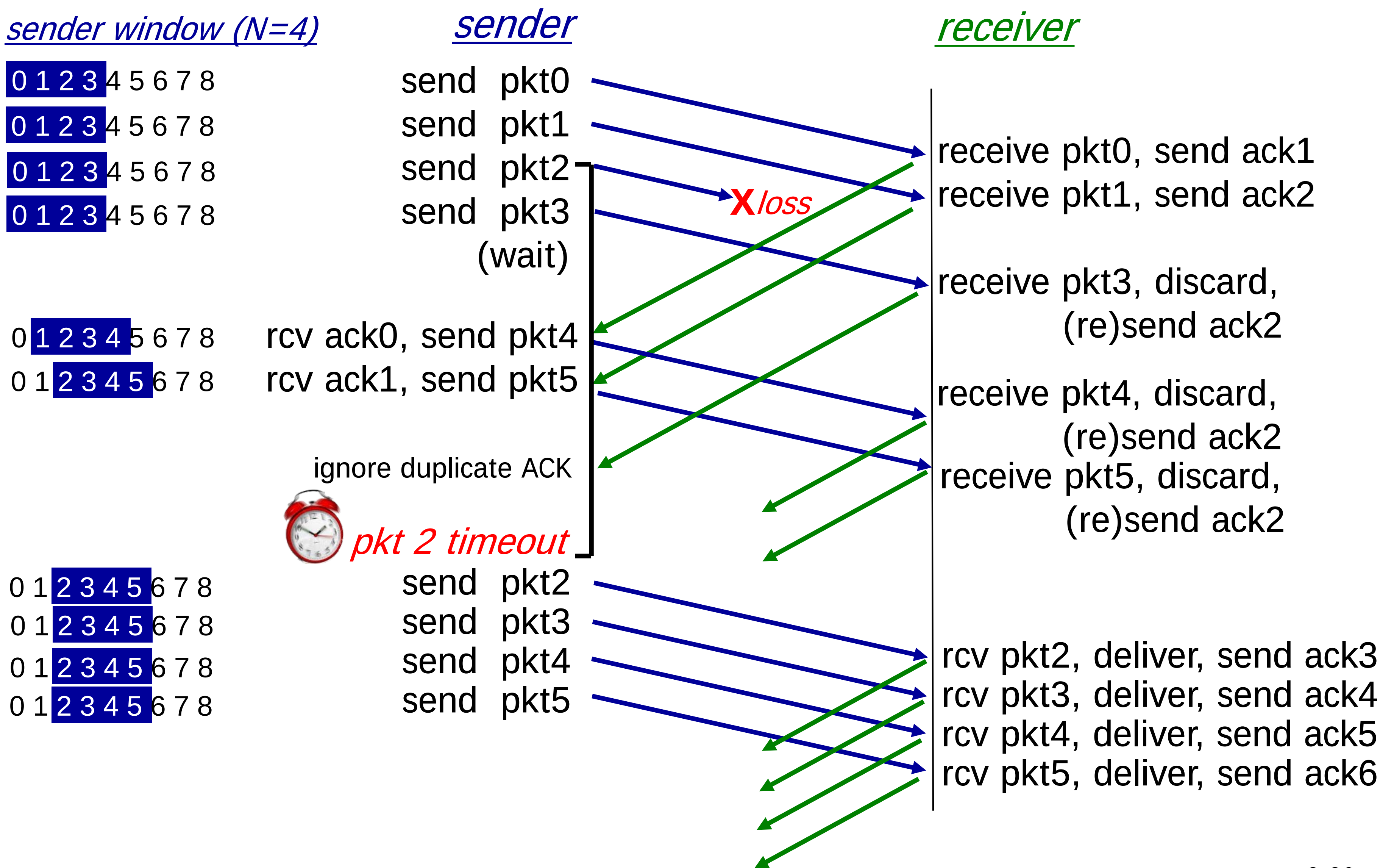
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

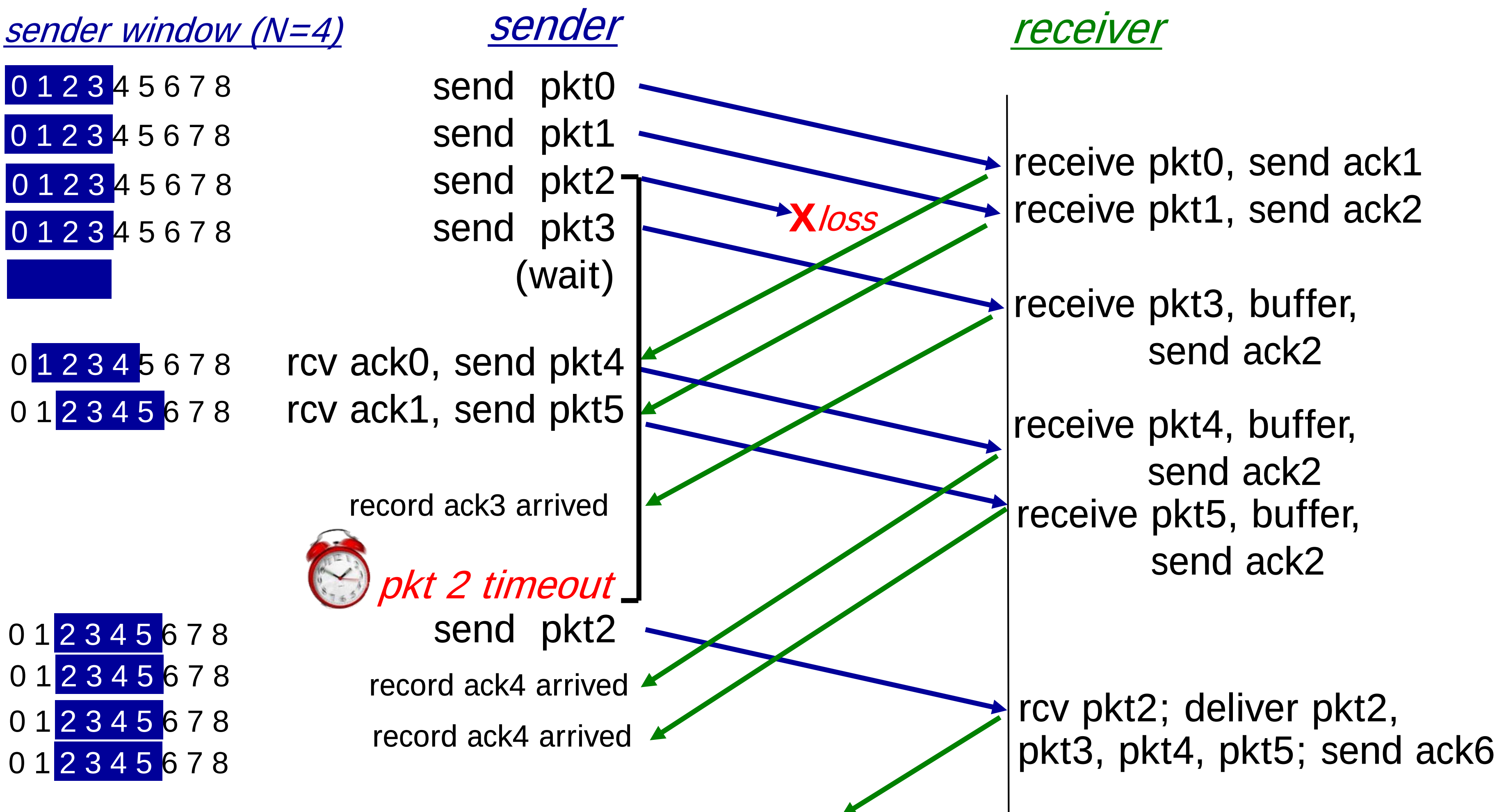
otherwise:

- ignore

GBN in action



Selective repeat in action



Q: what happens when ack2 arrives?

Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

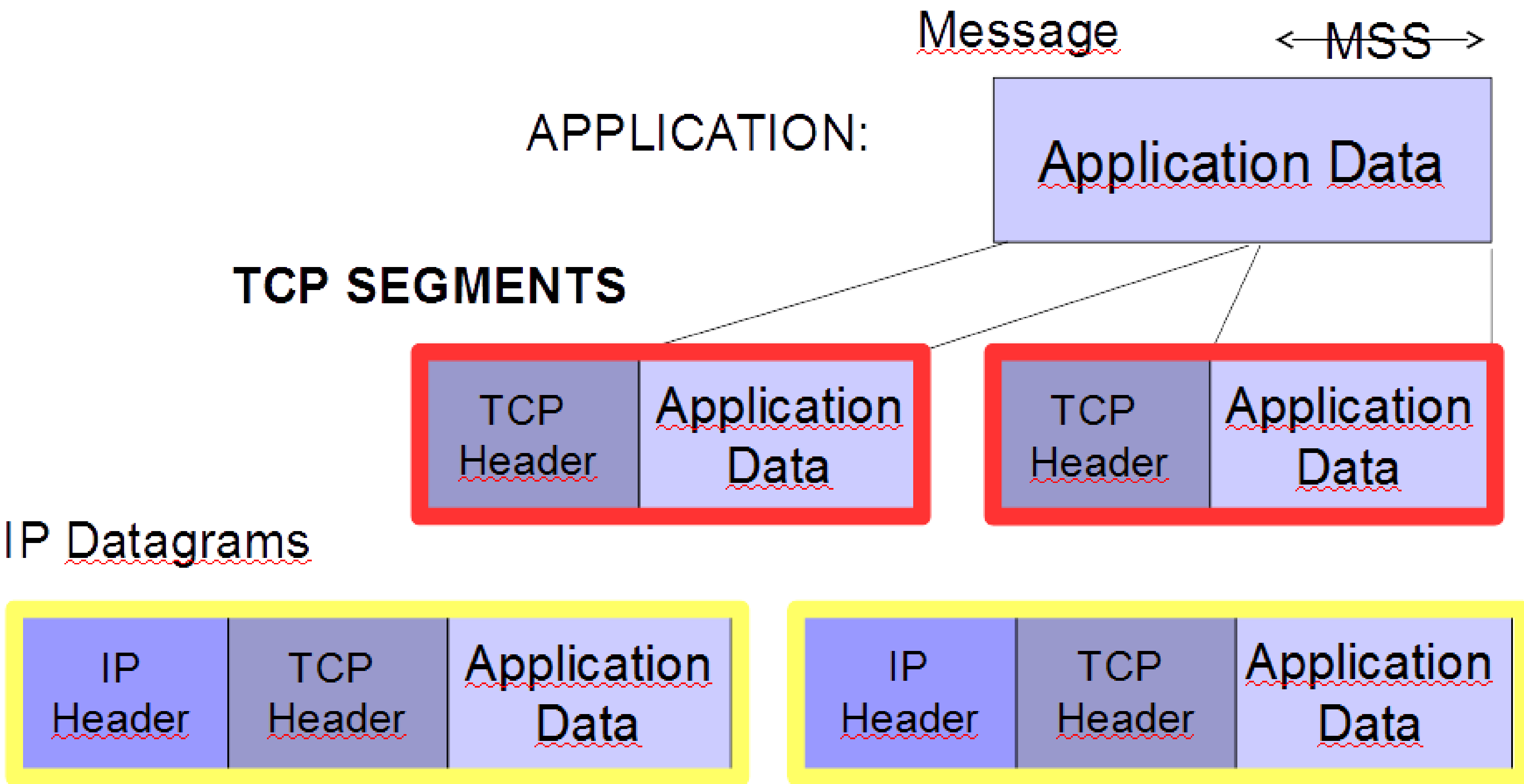
4.6 principles of congestion control

4.7 TCP congestion control

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

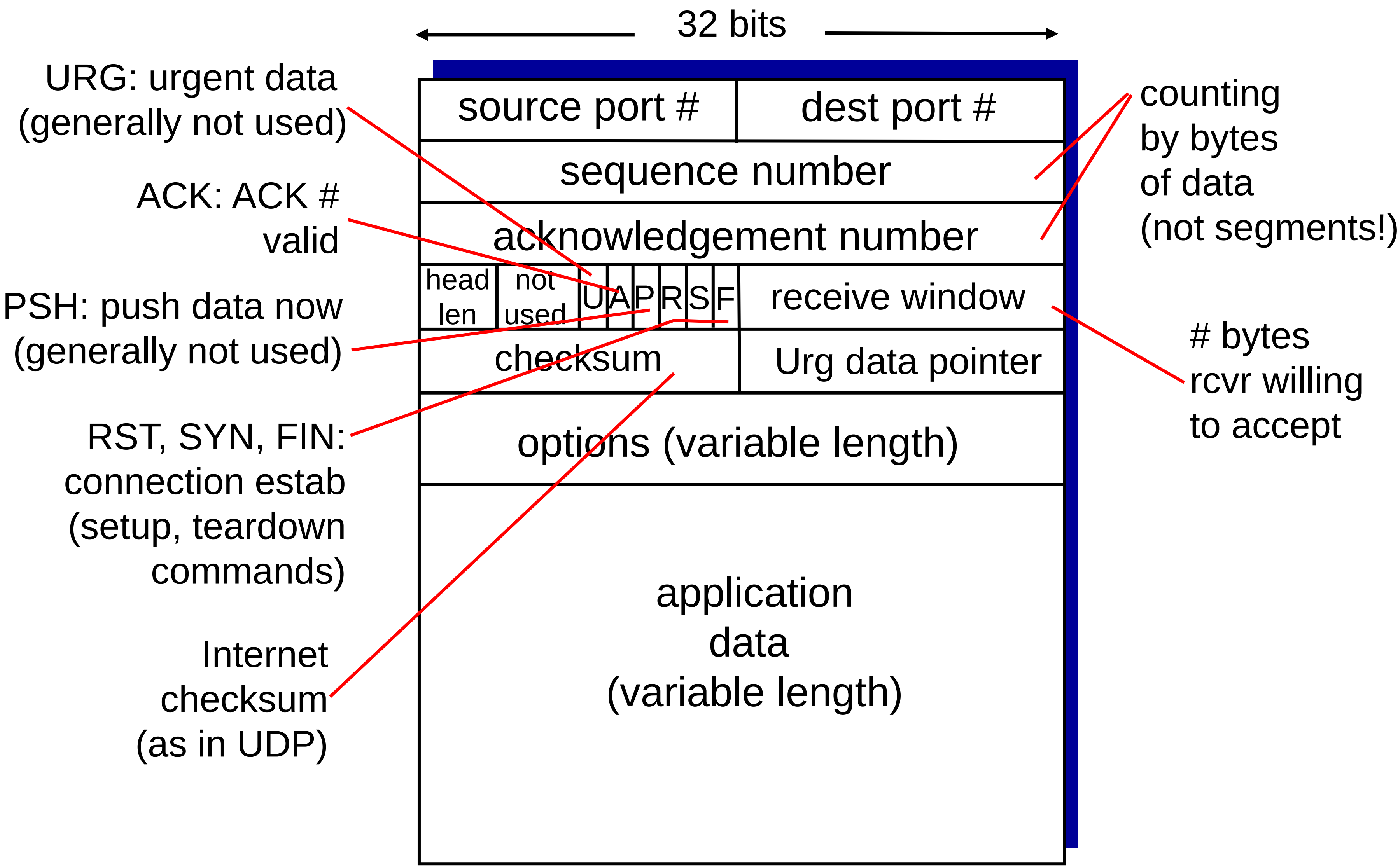
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam*
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP Segment Encapsulation



MSS=*Maximum Segment Size*

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

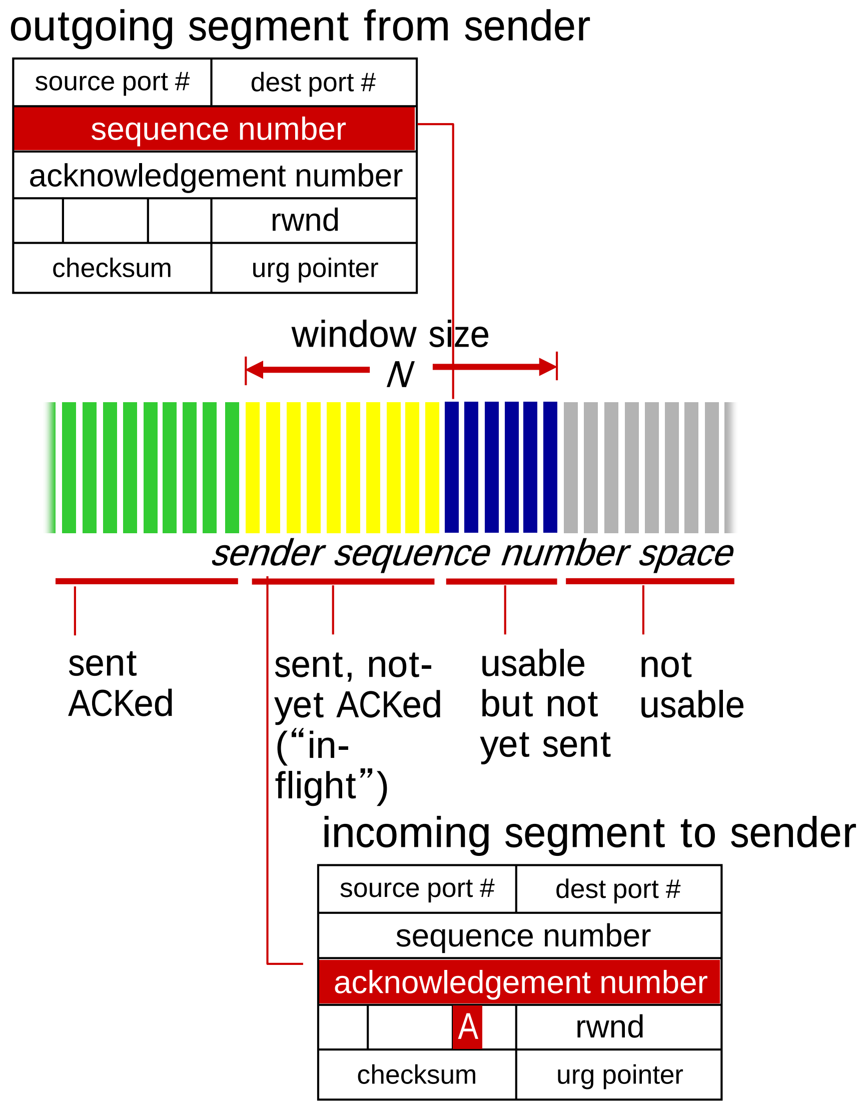
- byte stream "number" of first byte in segment's data

acknowledgements:

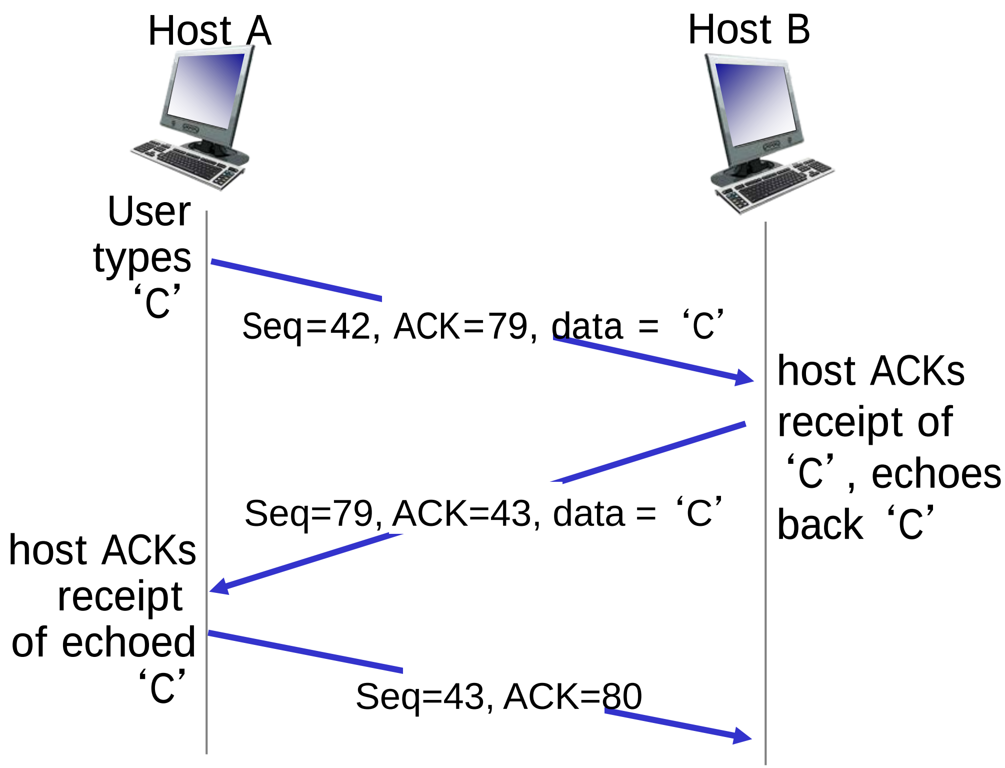
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACKs

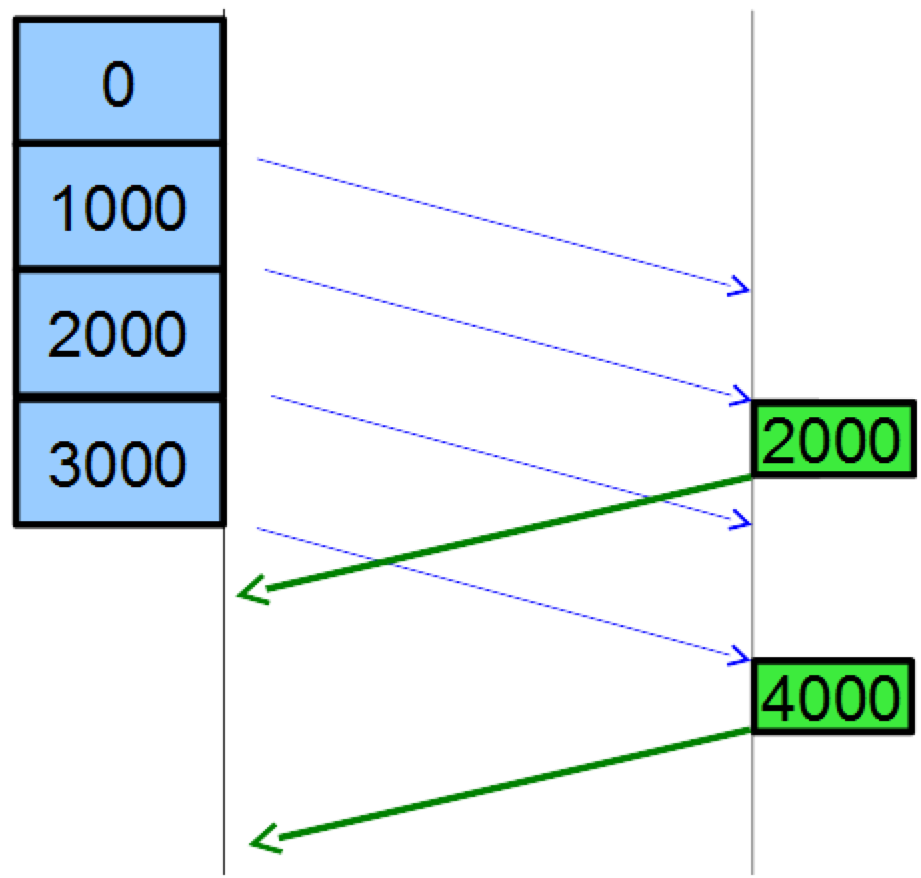


simple telnet scenario

Acknowledgements

- To reduce acknowledgements traffic, acknowledgements generating may be delayed until:
  - Received another segment
  - Send a segment in the opposite direction (piggybacking)
  - A timer (expires every 500 milliseconds)

MSS = 1000 bytes



Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP
- segment structure
- flow control
- reliable data transfer
- connection management

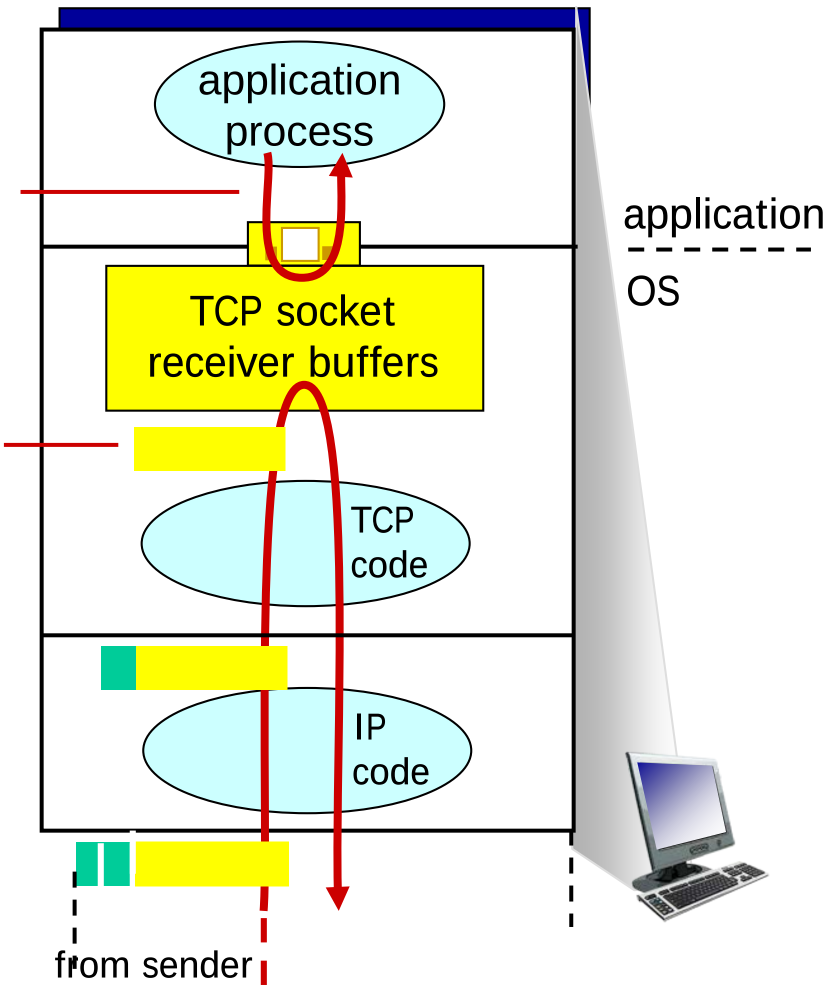
4.6 principles of congestion control

4.7 TCP congestion control

TCP flow control

application may remove data from TCP socket buffers ....

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



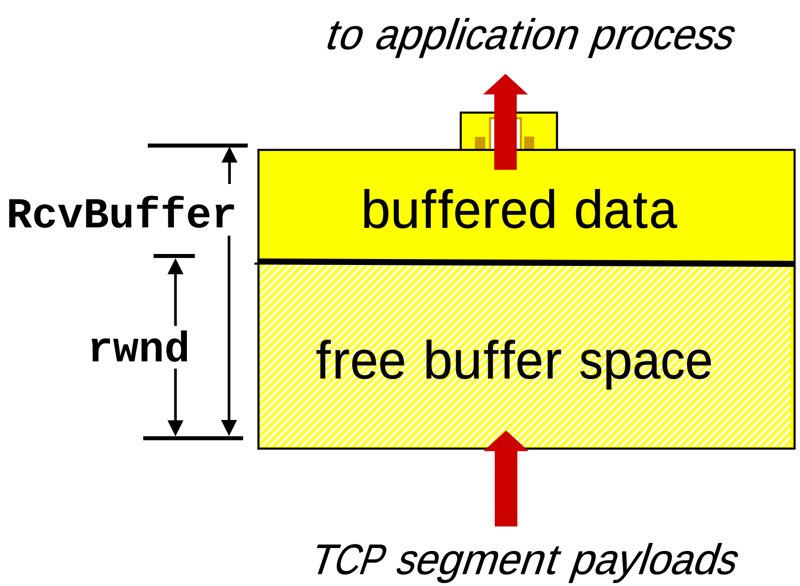
receiver protocol stack

flow control

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

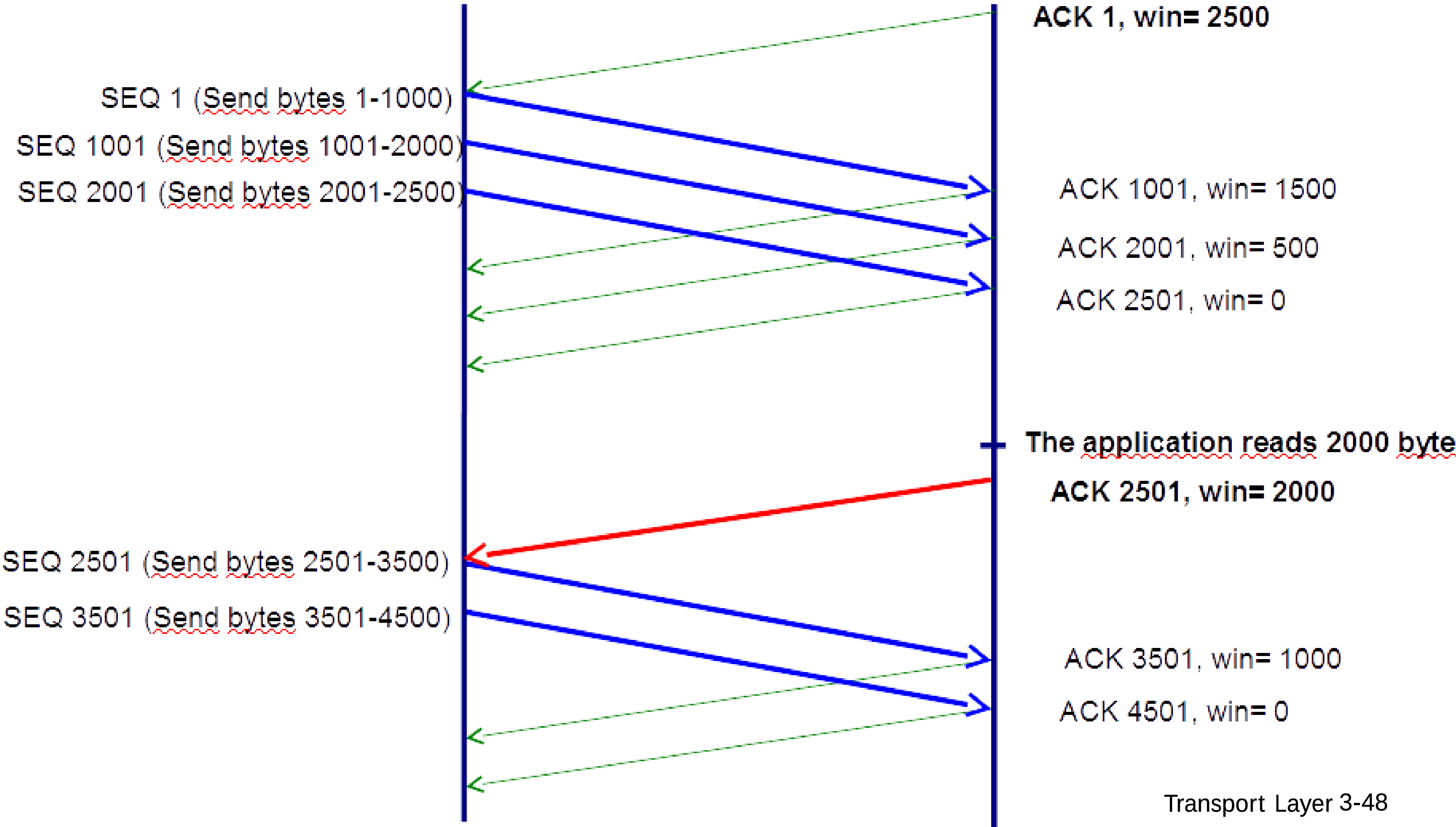
TCP flow control

- 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 unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow

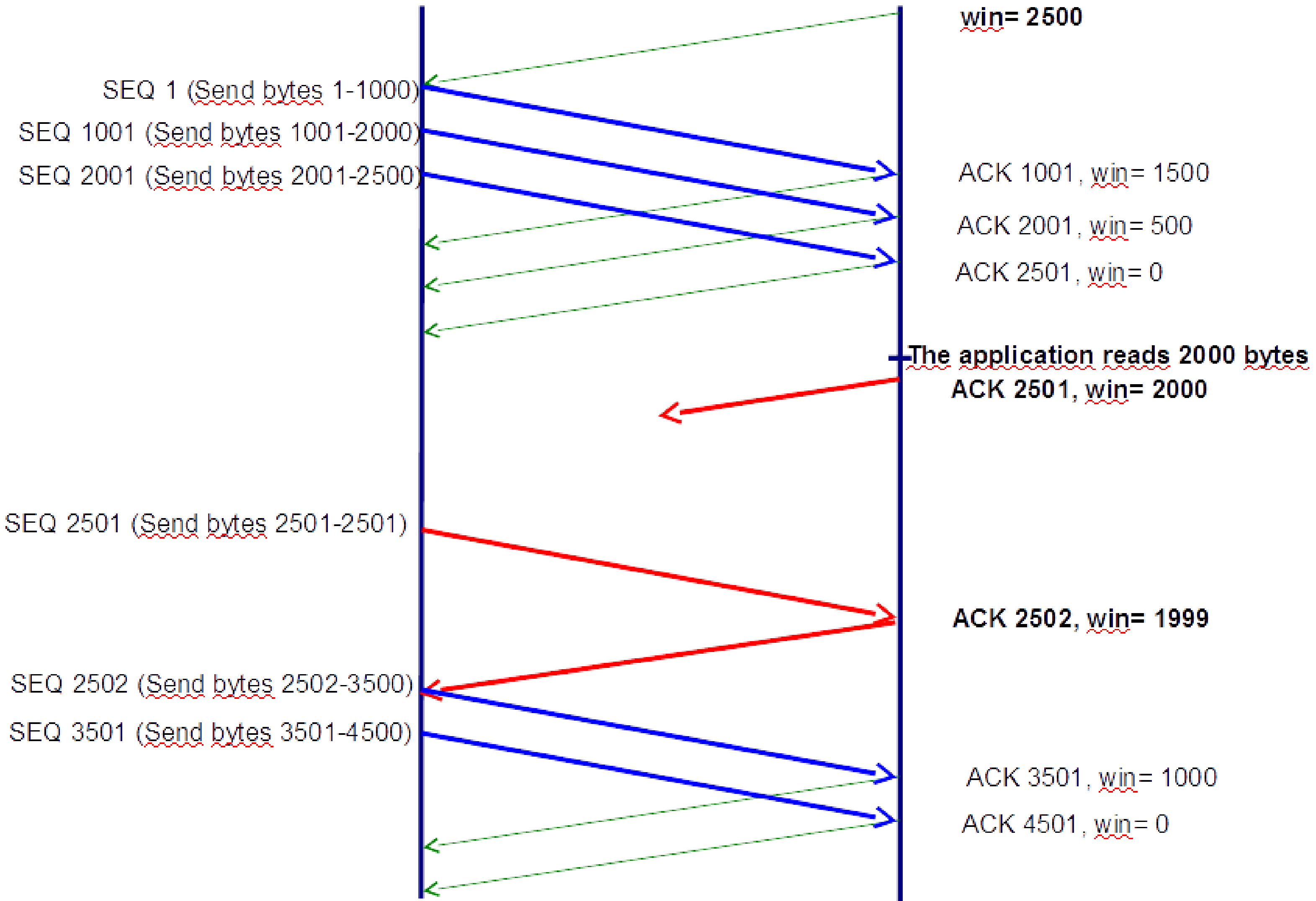


receiver-side buffering

Win Field Example



Win Field Example



Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP
- segment structure
- flow control
- reliable data transfer
- connection management

4.6 principles of congestion control

4.7 TCP congestion control

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

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`

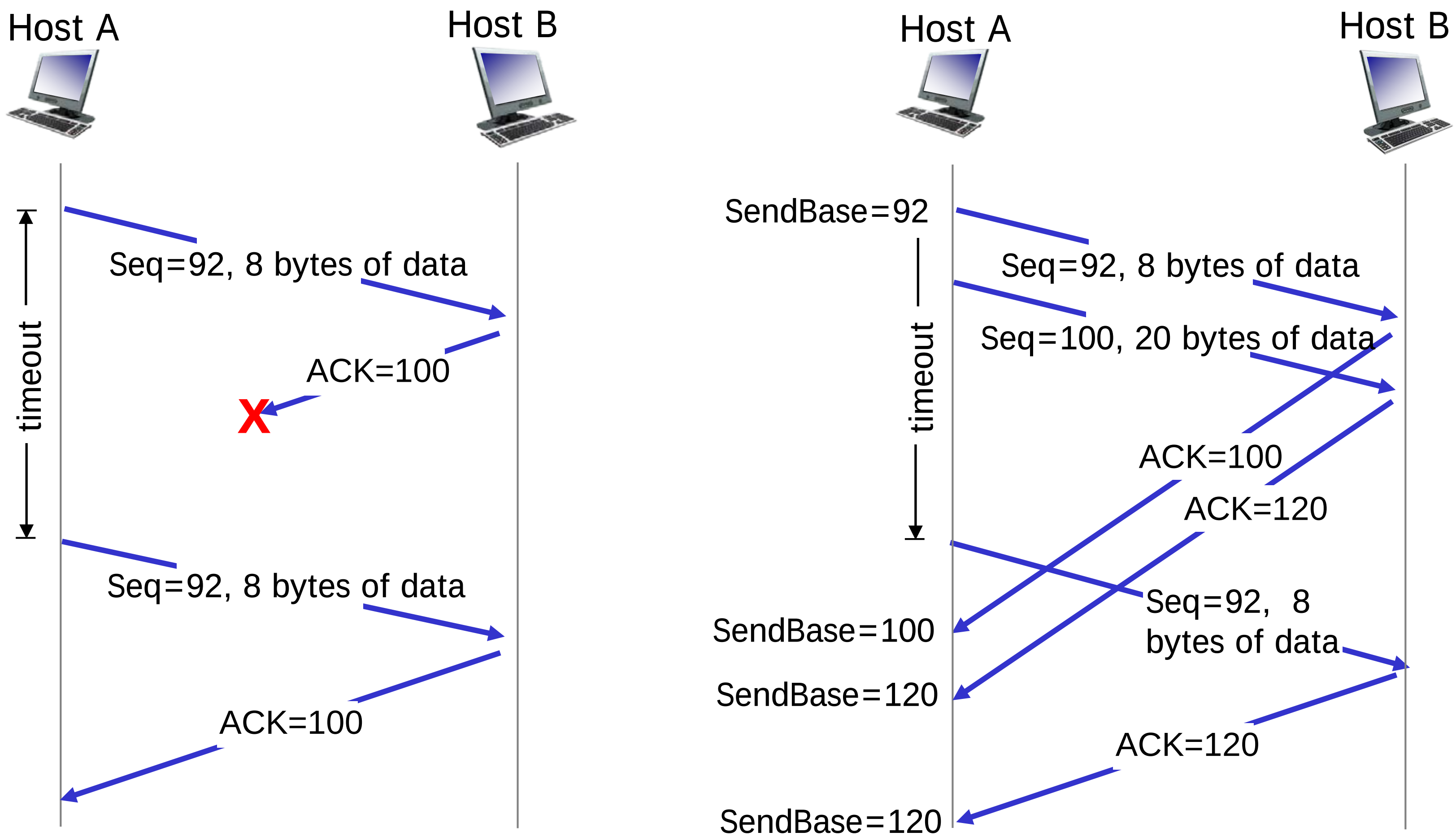
timeout:

- 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

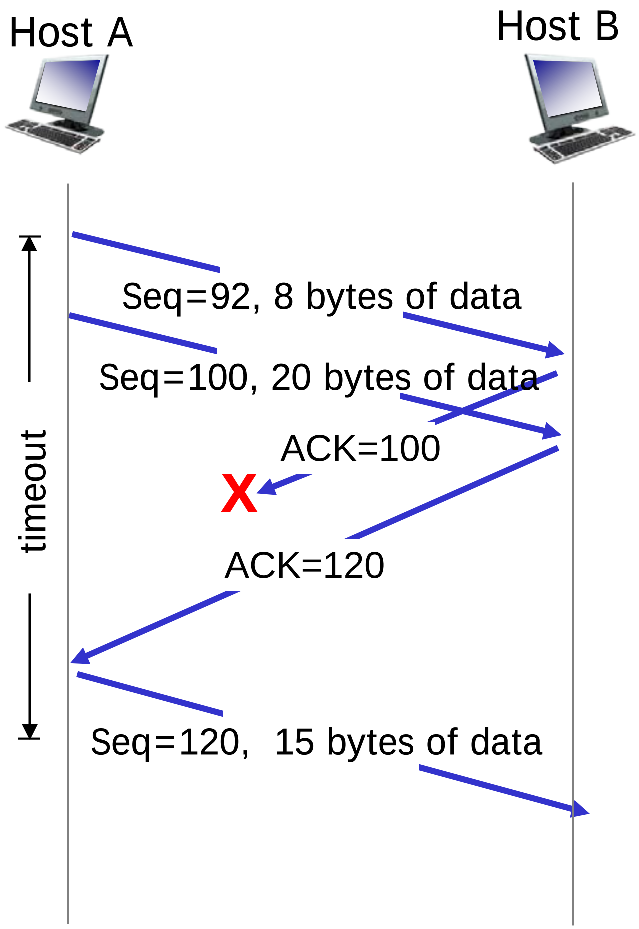
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP ACK generation [RFC 1122, RFC 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 fast retransmit

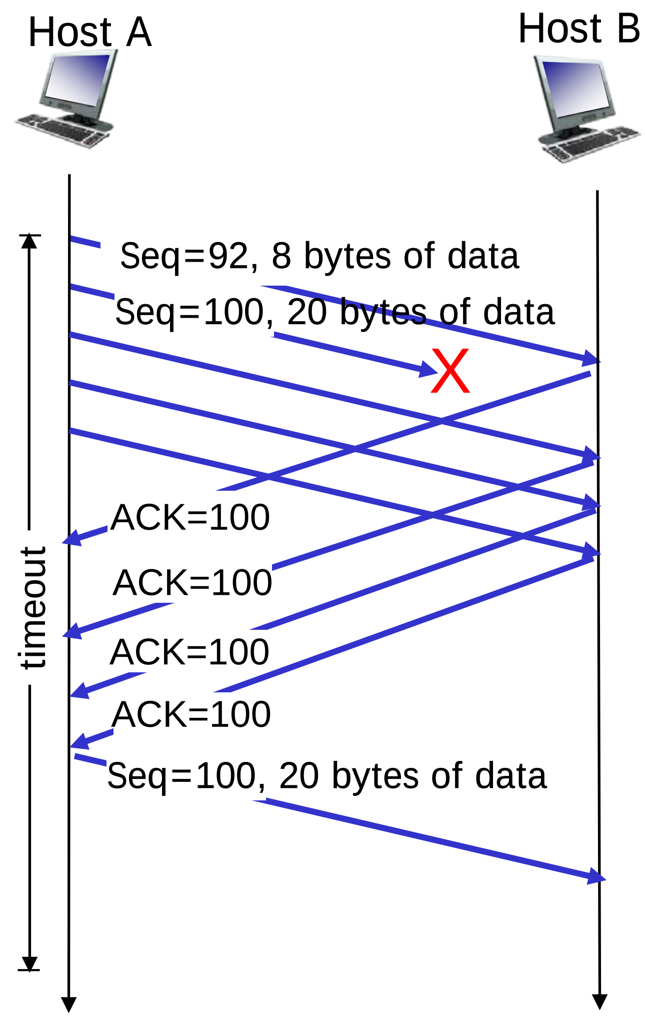
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

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 fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

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

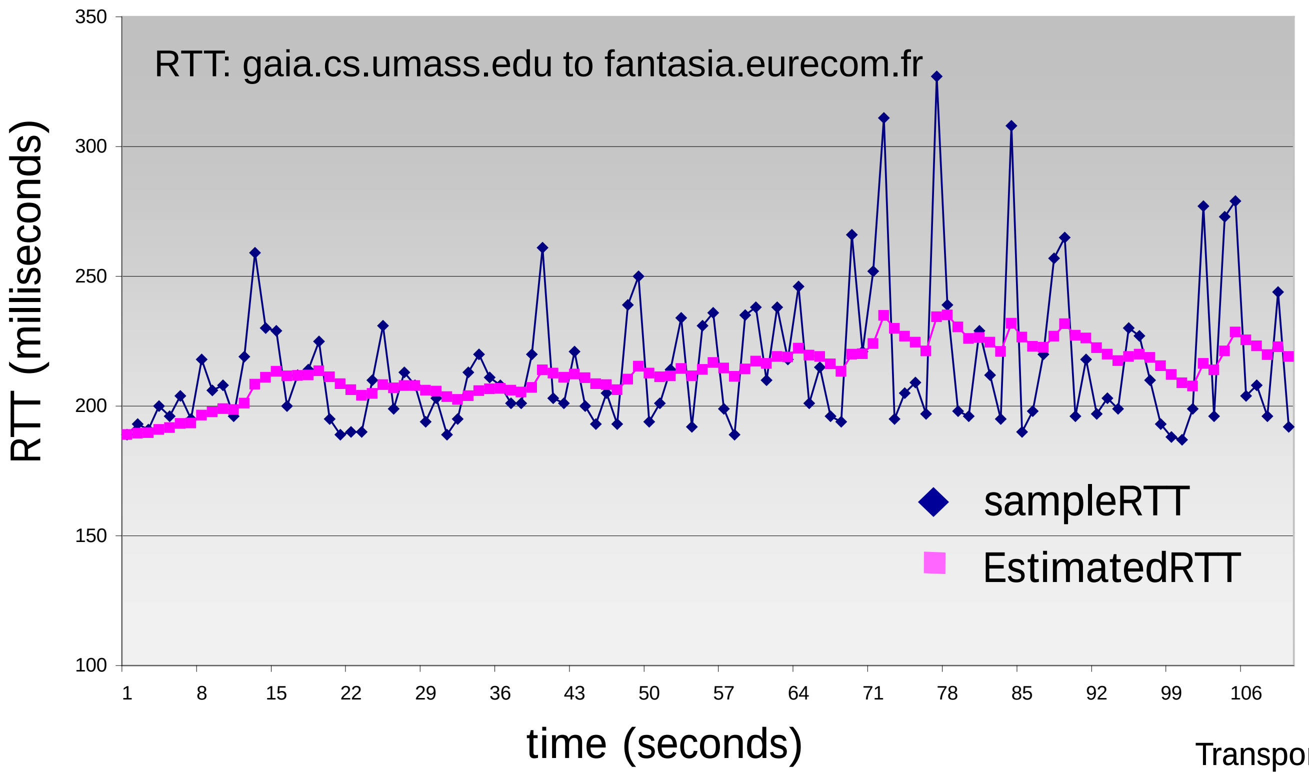
Q: how to estimate RTT?

- **`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
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$



TCP round trip time, timeout

- 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)
```

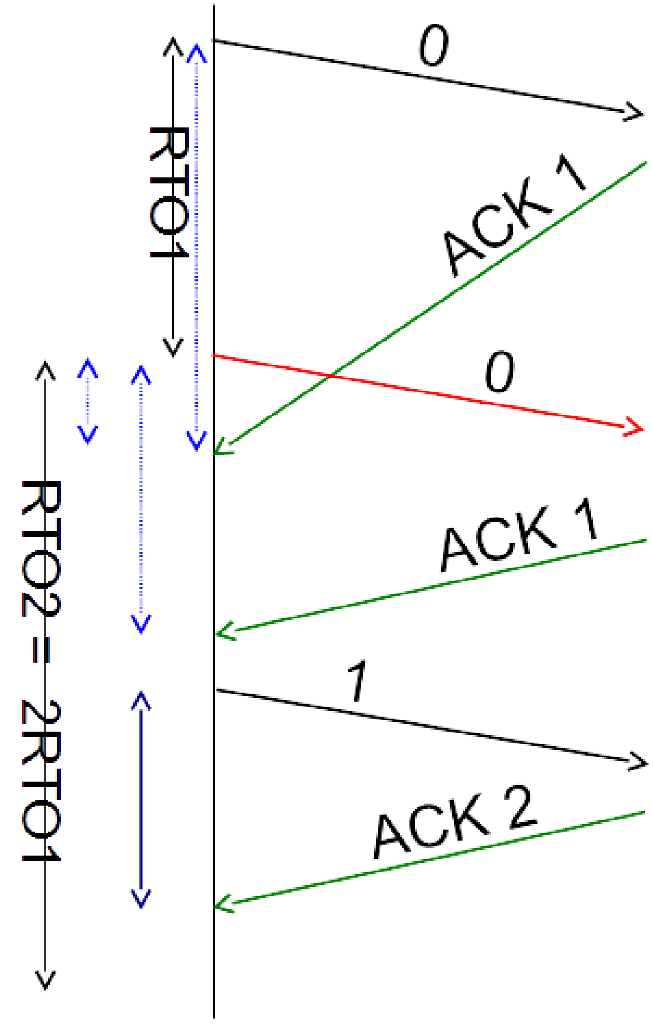
```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

TCP round trip time, timeout

- When a segment is retransmitted and an ACK is received is impossible to know at which copy corresponds (original or retransmitted segment)
  - Solution: Karn algorithm
    - Not taking into account the RTT measures of retransmitted segments
    - In retransmissions, RTO value doubles (exponential backoff)



Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

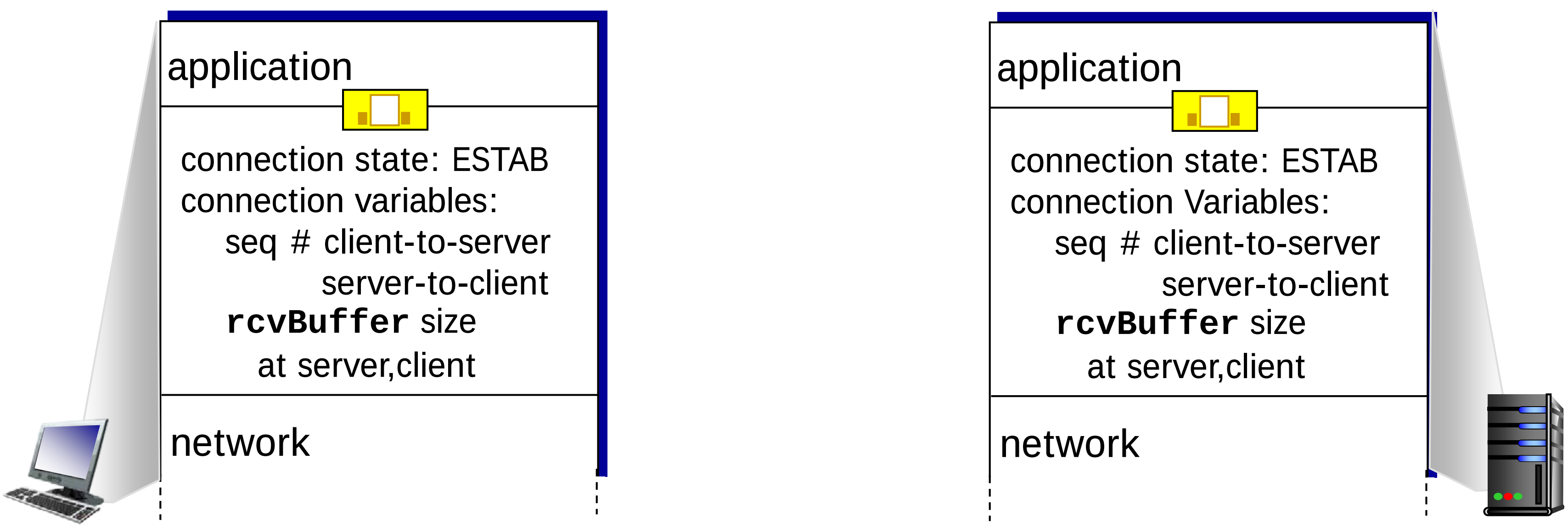
4.6 principles of congestion control

4.7 TCP congestion control

Connection Management

before exchanging data, sender/receiver "handshake":

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

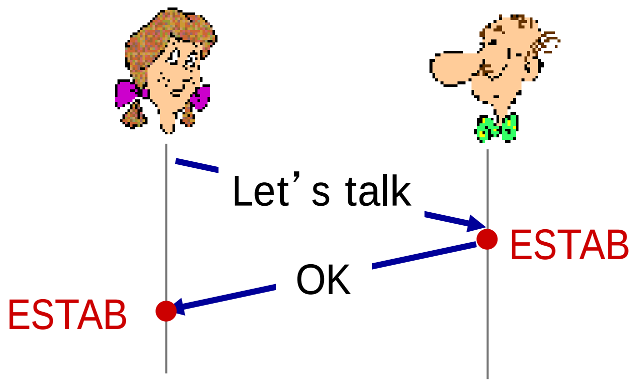


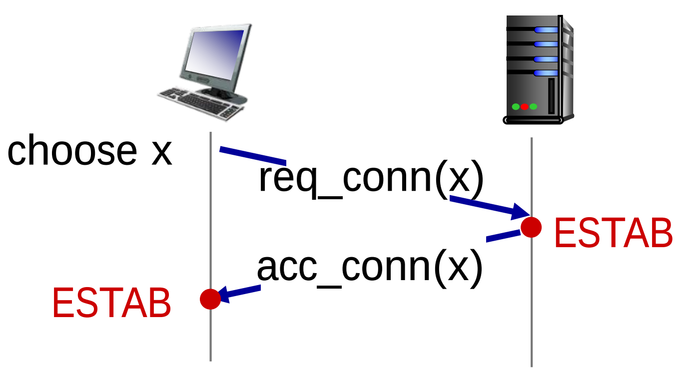

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to establish a connection

2-way handshake:



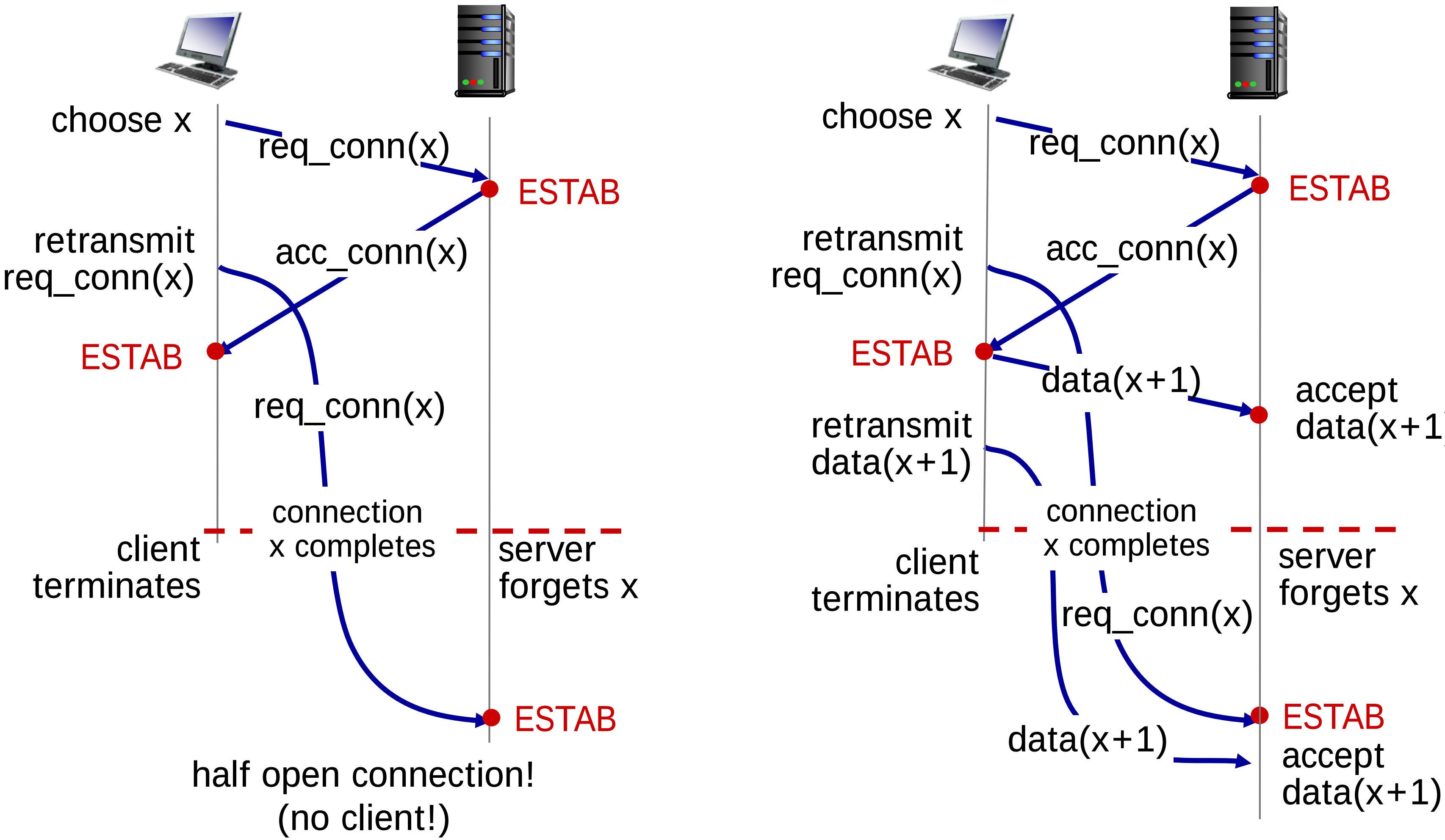


Q: will 2-way handshake always work in network?

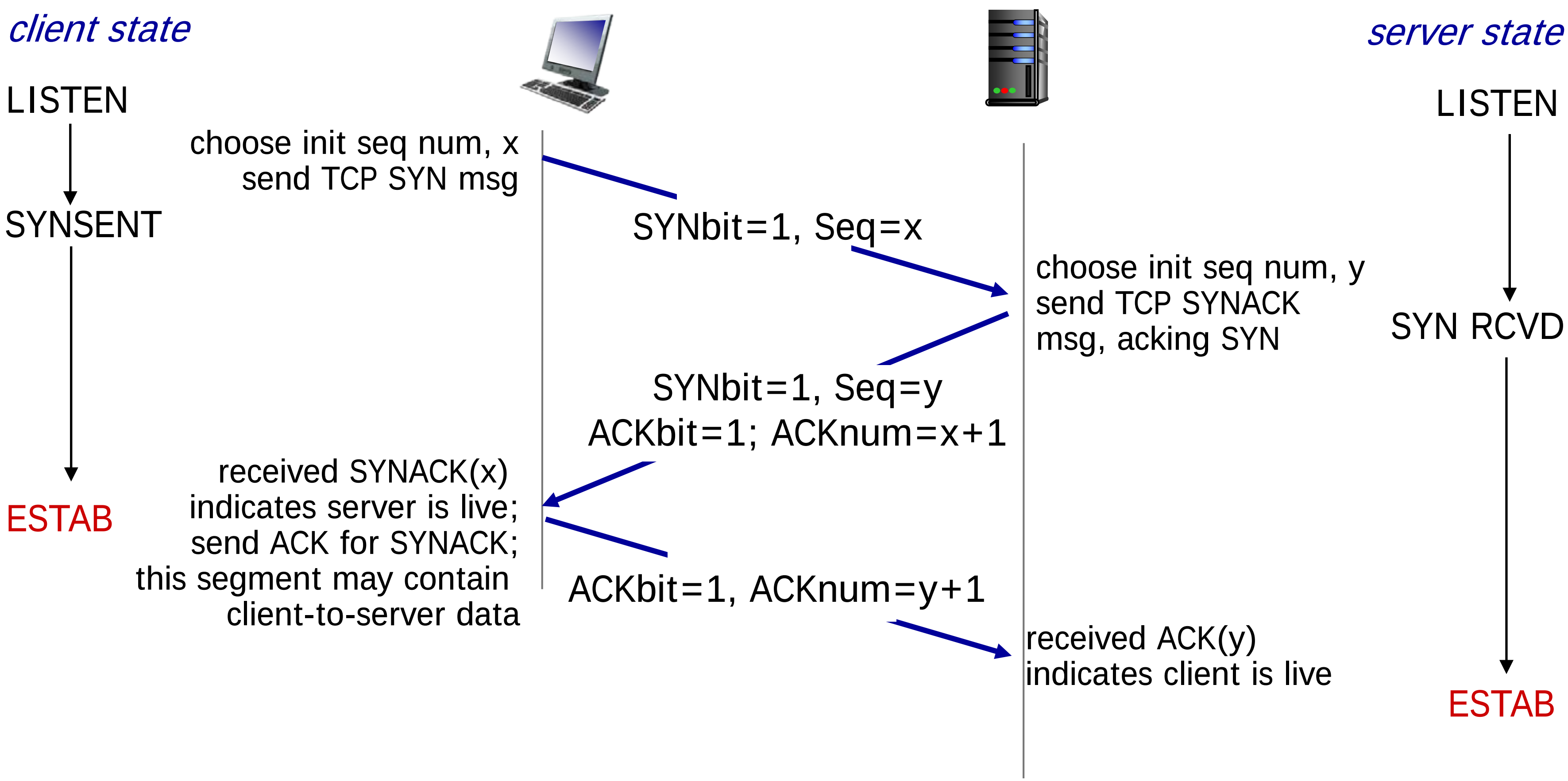
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

Agreeing to establish a connection

2-way handshake failure scenarios:

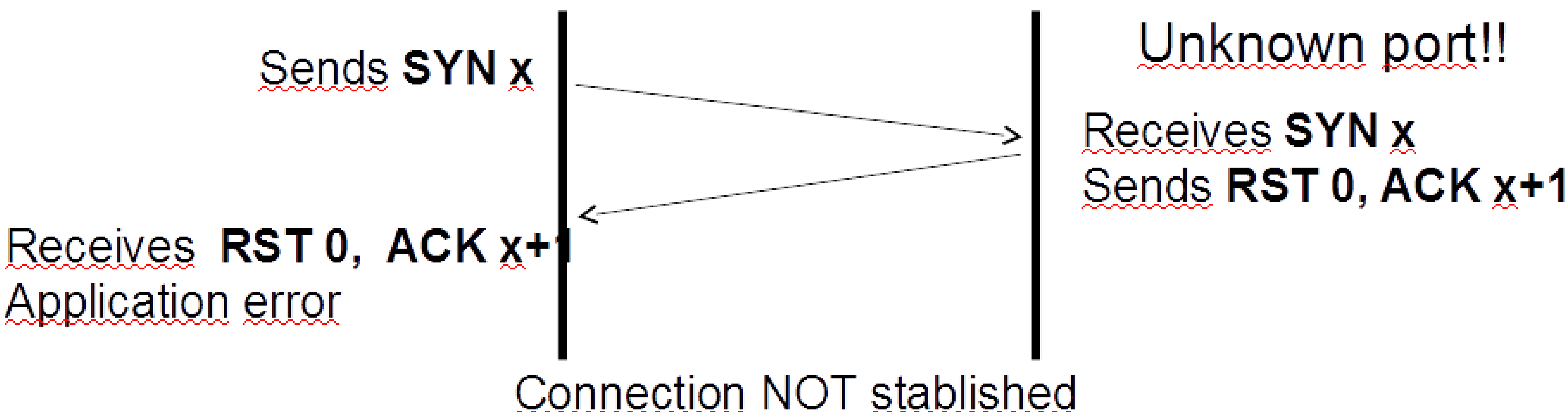


TCP 3-way handshake

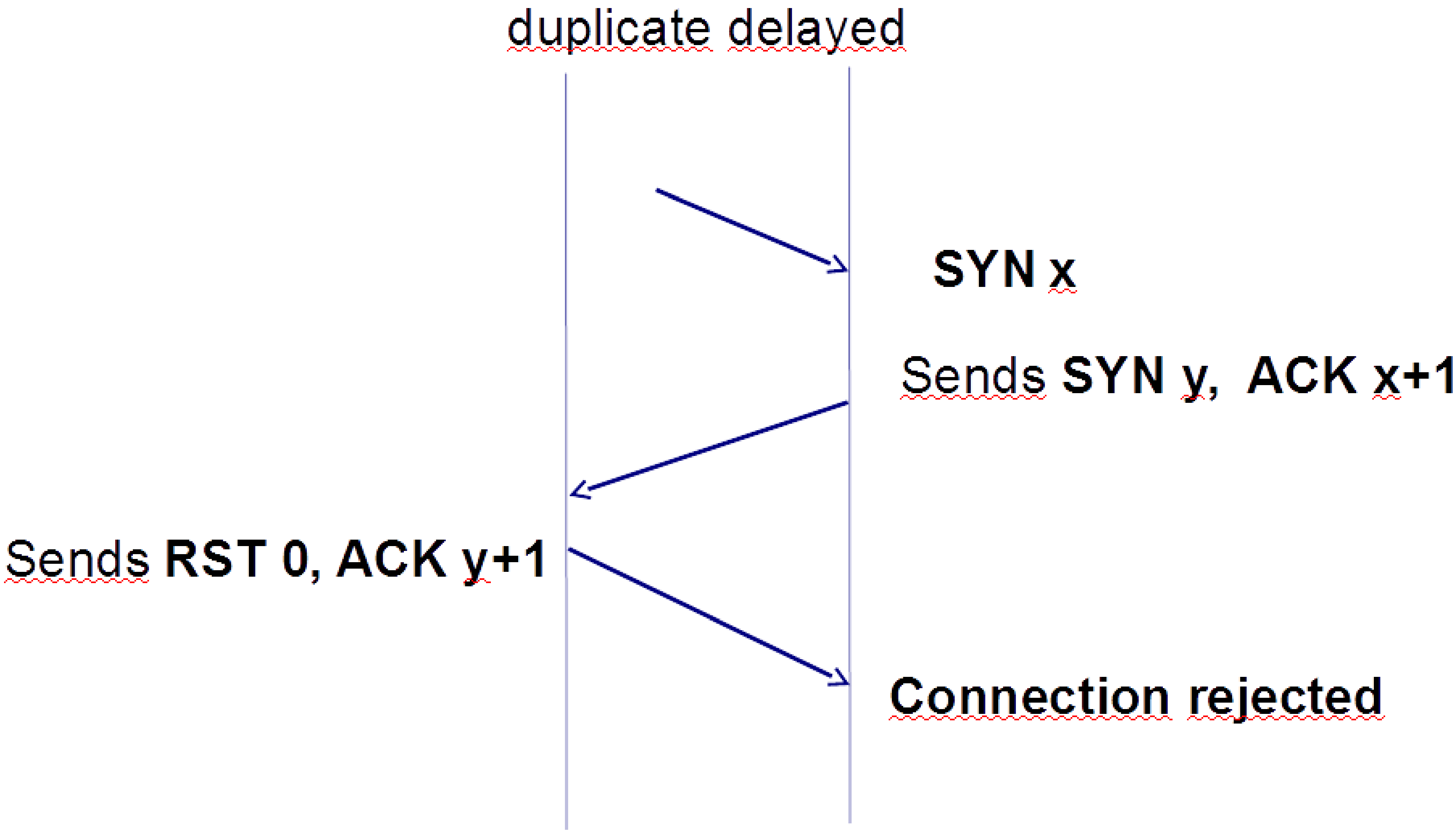


Reset Flag

- RESET: abortion of a TCP connection
  - causes:
    - Sequence numbers impossible
    - The destination port is not in use (not open)

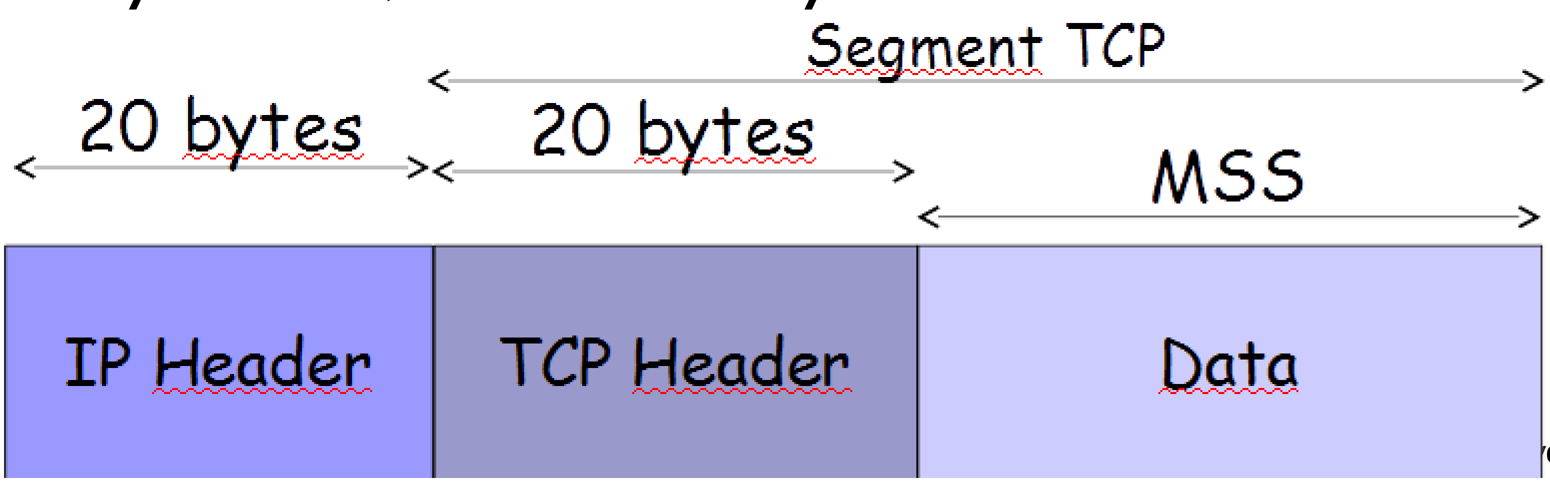


Duplicated delayed



TCP Options: MSS

- Each end of the connection announces its MSS (Maximum Segment Size) in the SYN segment
  - e.g: if host A announces MSS = 100 bytes, segments with more than MSS bytes can not be sent to it.
  - by default, MSS = 536 bytes



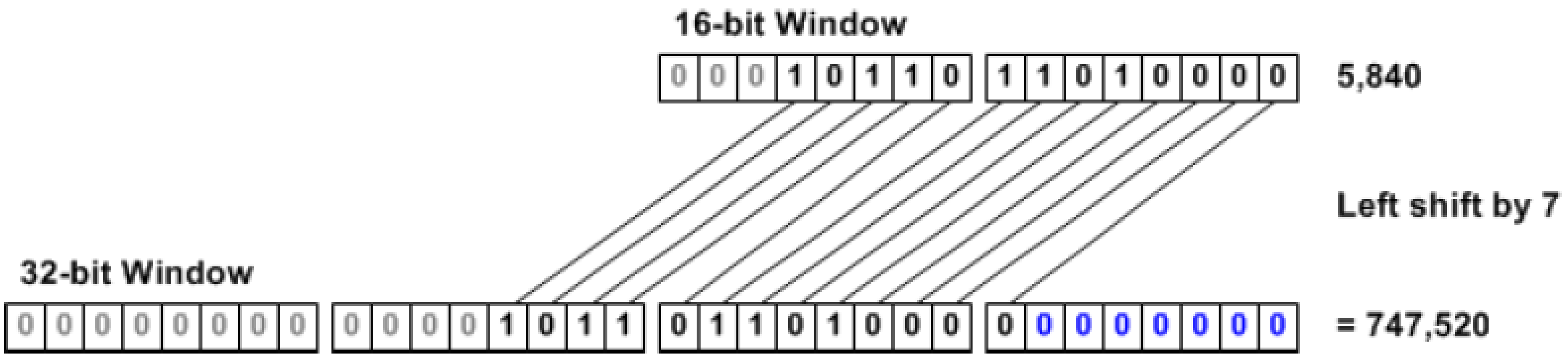
TCP Options: Window Scaling

- TCP hosts agree to limit the amount of unacknowledged data that can be in transit at any given time
  - This is referred to as the *window size*, and is communicated via a 16bit field in the TCP header
    - Maximum receive window is only 65,535 bytes
    - If RTT*vtrans> 65536? It wastes potential throughput

  - SOLUTION: TCP window scaling option (RFC 1323)
    - window scaling simply extends the 16bit window field to 32 bits in length
      - 2^n where n is the value of window scaling option
    - The window scaling option may be sent only once during a connection by each host, in its SYN packet
    - By using the window scale option, the receive window size may be increased up to a maximum value of 1,073,725,440 bytes
      - The maximum valid scale value is 14

Window Scaling Example

- Window Scaling = 7
  - multiplies the value by 128



Example from TCP Windows and Window Scaling PacketLife.net

TCP Options: Selective Acknowledgment

- Sack-Permitted Option
  - This option may be sent in a SYN by a TCP that has been extended to receive (and presumably process) the SACK option once the connection has opened.
  - It MUST NOT be sent on non-SYN segments
- The SACK option is to be used to convey extended acknowledgment information from the receiver to the sender over an established TCP connection.

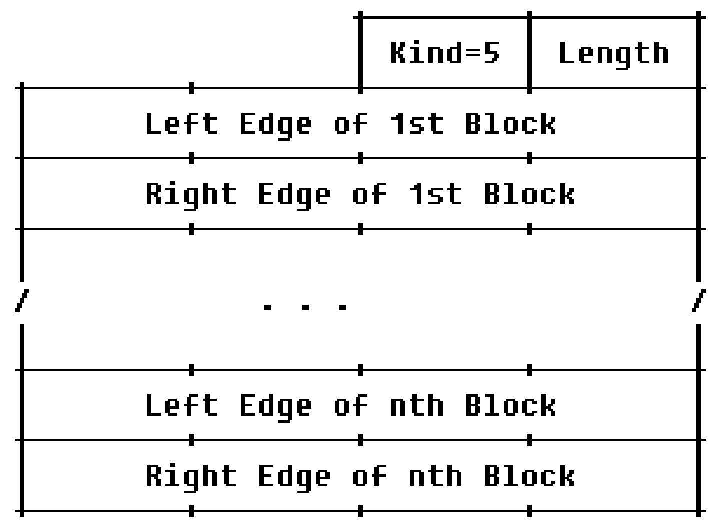
TCP Options: Selective Acknowledgment

- Cumulative ACKs can not confirm the reception of segments out of order
  - May cause unnecessary retransmissions
- The selective ACKs (SACK) permits the reception of out of order segment
  - Each block represents received bytes of data that are contiguous and isolated; that is, the bytes just below the block, (Left Edge of Block - 1), and just above the block, (Right Edge of Block), have not been received.

TCP SACK Option:

Kind: 5

Length: Variable



TCP Options: Timestamp

- Timestamp is used to calculate more accurately the RTT
- The Timestamps option carries two four-byte timestamp fields.
  - The TSval field contains the current value of the timestamp clock of the TCP sending the option
  - The TSecr field is valid if the ACK bit is set in the TCP header.

```
+--------+--------+---------------------+---------------------+
|Kind=8  |   10   |   TS Value (TSval)  |TS Echo Reply (TSecr)|
+--------+--------+---------------------+---------------------+
    1        1              4                     4
```

Timestamp Example

TCP A **TCP B**

<A,**TSval=1** ,TSecr=120> ------>

<---- <ACK(A),TSval=127,**TSecr=1** >

<B,**TSval=5**,TSecr=127> ------>

<---- <ACK(B),TSval=131,**TSecr=5**>

. . . .

<C,**TSval=65**,TSecr=131> ------>

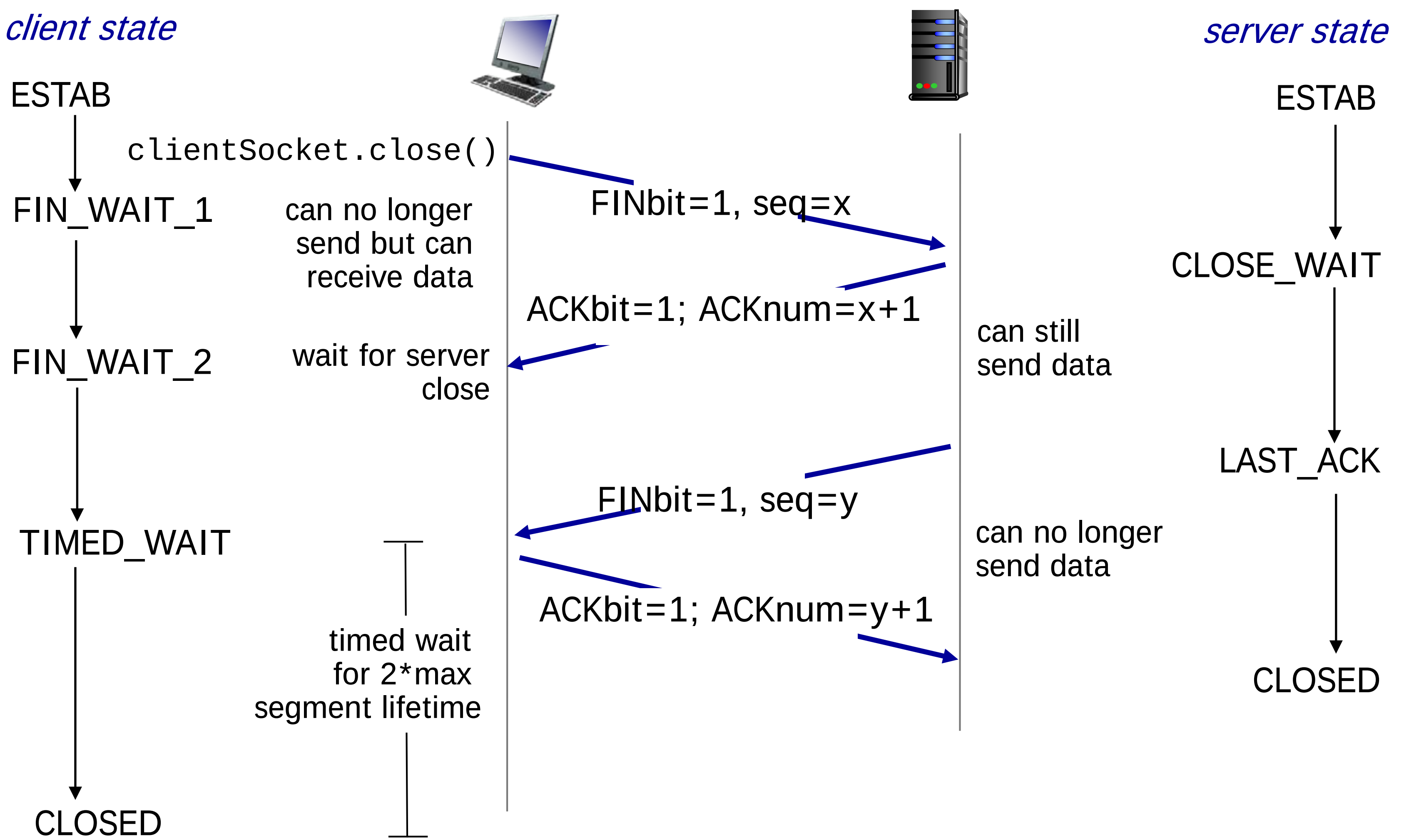
<---- <ACK(C),TSval=191,**TSecr=65**>

(etc)

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



Chapter 4 outline

4.1 transport-layer services

4.2 multiplexing and demultiplexing

4.3 connectionless transport: UDP

4.4 principles of reliable data transfer

4.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

4.6 TCP congestion control

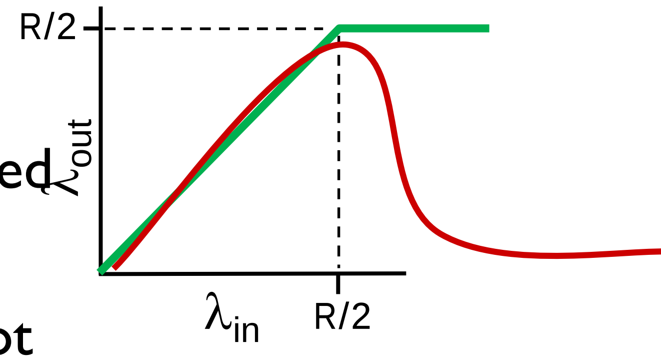
Principles of congestion control

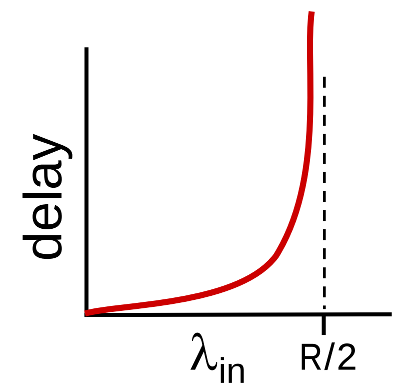
congestion:

- 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)
- a top-10 problem!

Goal of TCP Congestion Control

- Congestion is bad for the overall performance in the network.
  - Excessive delays can be caused.
  - Retransmissions may result due to dropped packets
    - Waste of capacity and resources.
  - In some cases (UDP) packet losses are not recovered from.
  - Note: Main reason for lost packets in the Internet is due to congestion -- errors are rare.
- Goal of TCP is to determine the available network capacity and prevent network overload.
  - Depends on other connections that share the resources.





TCP Congestion Control

- TCP sender must use two algorithms to control the amount of outstanding data being injected into the network.
  - **slow start algorithm**
  - **congestion avoidance algorithm**
- To implement these algorithms, two variables are added to the TCP per-connection state.
  - the **congestion window (cwnd)**
    - It is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgment (ACK),
  - **the slow start threshold (ssthresh)**
    - It is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission
    - when **cwnd < ssthresh**,
      - The **slow start** algorithm is used
    - when **cwnd > ssthresh**
      - the **congestion avoidance** algorithm is used

TCP Congestion Control

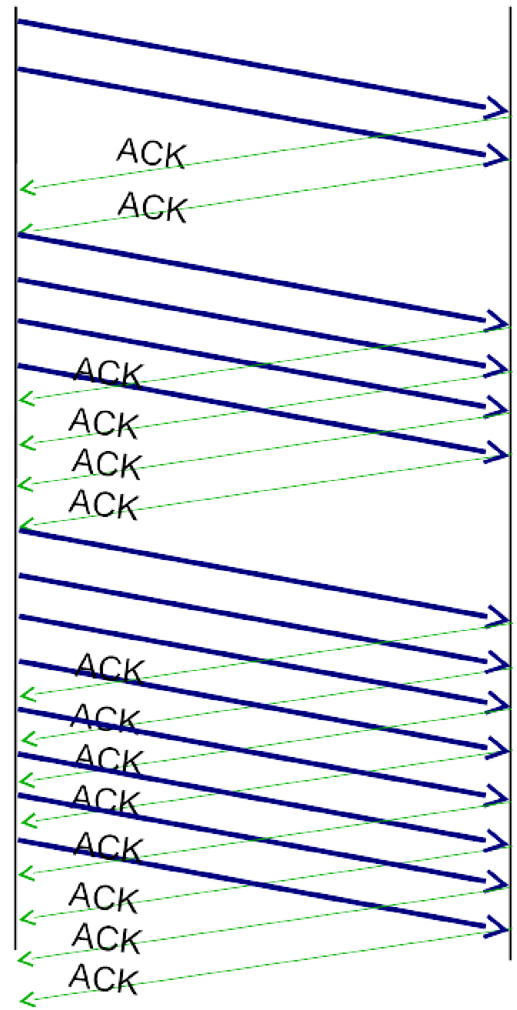
- The **minimum of cwnd and rwnd governs data transmission.**
  - **transmission window (twnd) = min(cwd,rwd)**
    - Remember that the receiver's advertised window (rwnd) is a receiver-side limit on the amount of outstanding data.

Slow Start Algorithm

- Beginning transmission into a network with unknown conditions requires TCP to slowly probe the network to determine the available capacity, in order to avoid congesting the network with an inappropriately large burst of data
- The **slow start** algorithm is **used** for this purpose
  - **at the beginning of a transfer**, or
  - **after repairing loss detected by the retransmission timer**.

Slow Start Algorithm

- **At the beginning of a transfer**
  - cwnd = 2 segments
  - When an ACK is received
    - cwnd += 1 segments
  - ssthresh = rwnd



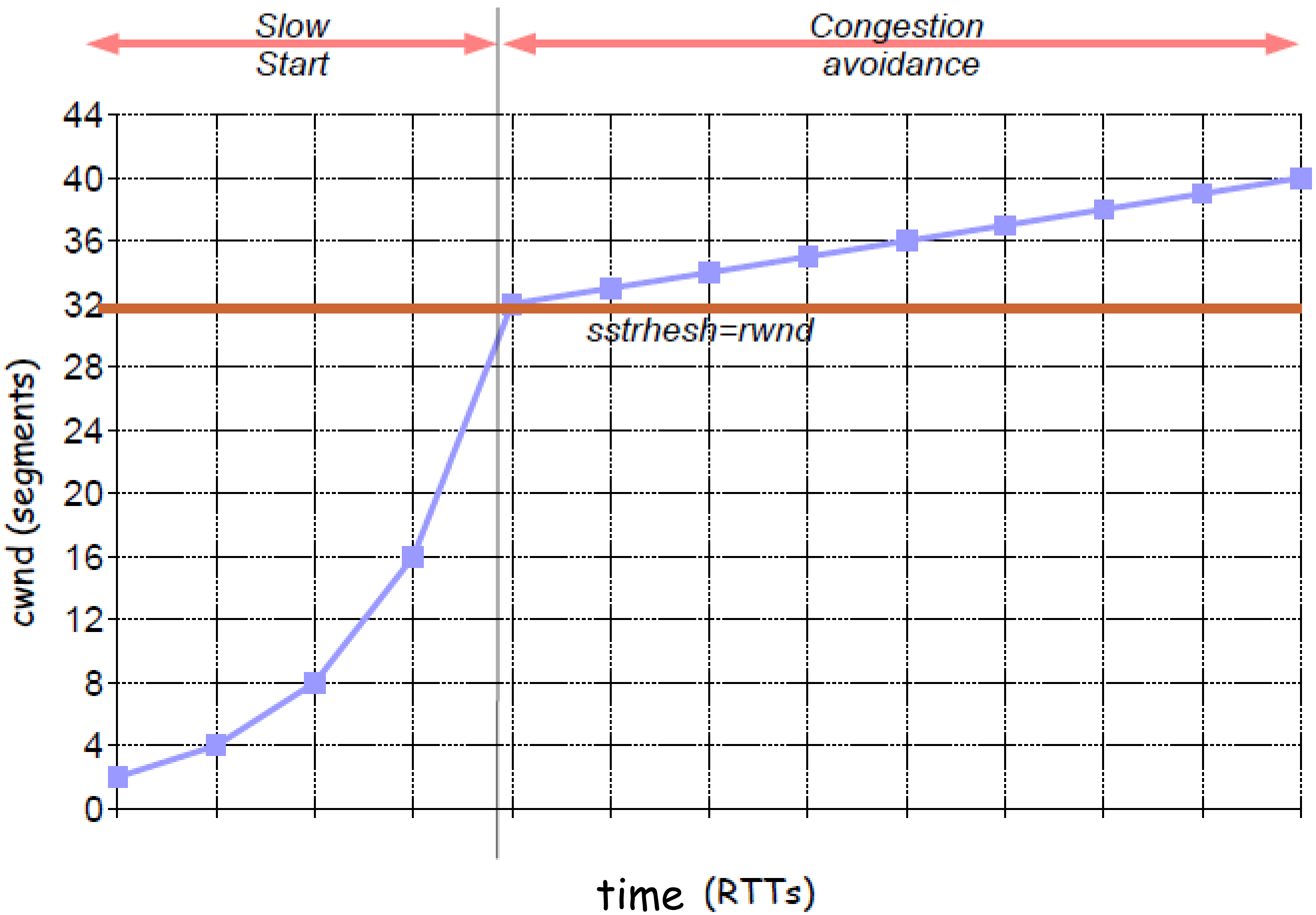
Congestion Avoidance Algorithm

- When the number of bytes acknowledged reaches cwnd, then cwnd can be incremented by up to 1 segment
  - cwnd + = 1/cwnd

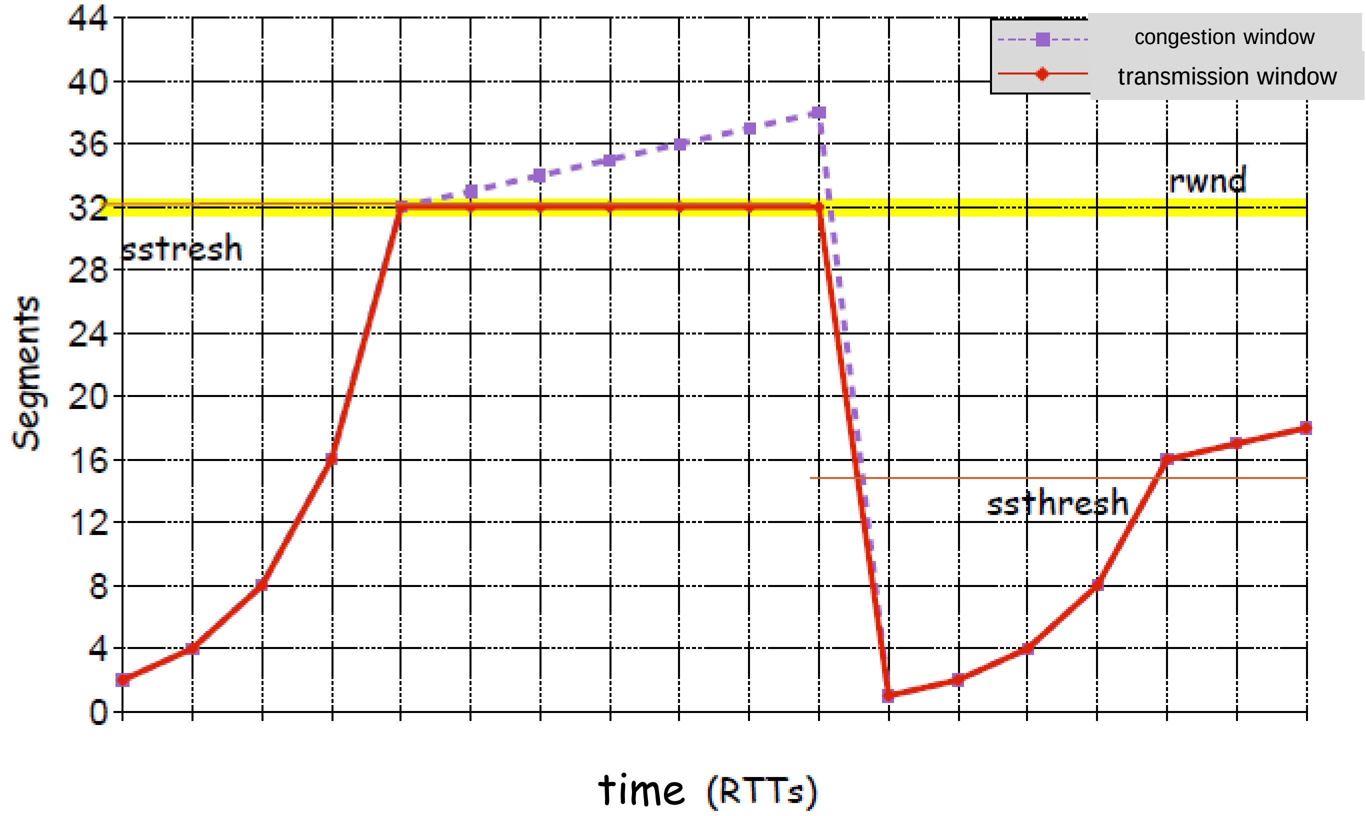
Congestion Detection

- Packet loss is a sign of congestion
- Two indicators of congestion:
  - A retransmission timer expires (timeout)
  - Three duplicate ACKs are received
- What does TCP do then?
  - ssthresh = max(twnd/2,2)
  - **After repairing loss detected by the retransmission timer**
    - cwnd = 1 segment
    - slow start
  - **When 3 ACKs are received**
    - cwnd = ssthresh
    - congestion avoidance

TCP Congestion Control



TCP Congestion Control



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this