Internet Protocols EBU5403
The Transport Layer Part 2

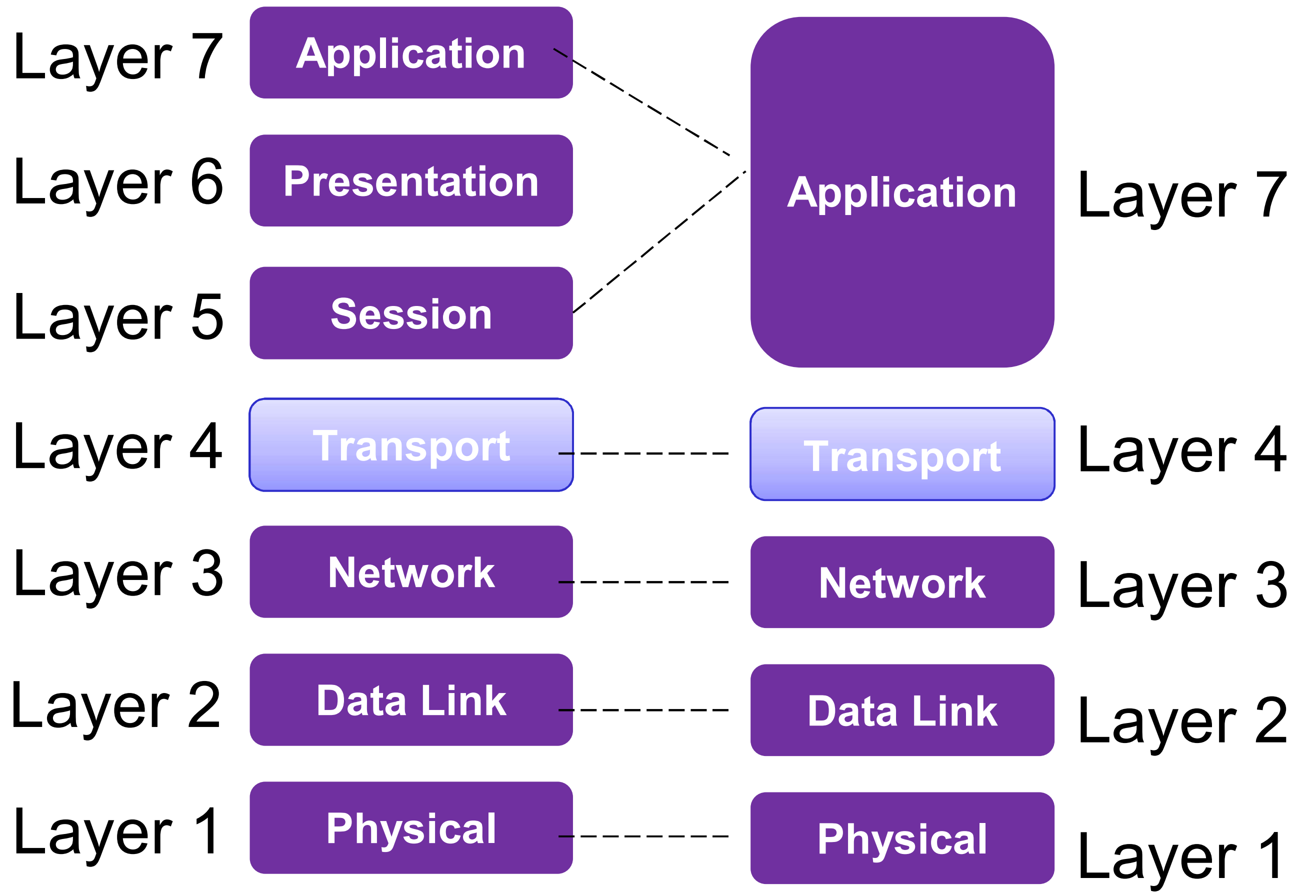
Module organiser: Richard Clegg
(r.clegg@qmul.ac.uk)
Michael Chai (michael.chai@qmul.ac.uk)
Cunhua Pan(c.pan@qmul.ac.uk)

	Part 1	Part 2	Part 3	Part 4
Ecommerce + Telecoms 1	Richard Clegg		Cunhua Pan	
Telecoms 2	Michael Chai			

Structure of course

- Part A
  - Introduction to IP Networks
  - The Transport layer (part 1)
- Part B
  - The Transport layer (part II)
  - The Network layer (part I)
  - Class test
- Part C
  - The Network layer (part II)
  - The Data link layer (part I)
  - Router lab tutorial (assessed lab work after this week)
- Part D
  - The Data link layer (part II)
  - Network management and security
  - Class test

Transport Layer



Transport layer outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

Recap of rdt 1.0 2.0, 2.1, 2.2, 3.0

- rdt 1.0 assumed no losses or errors
- rdt 2.0 introduced ACK (ACKnowledgment) and NACK to say a packet had been received or an error occurred.
- rdt 2.1 introduced the concept of the SEQuence number to deal with errors in ACKs.
- rdt 2.2 introduced the repeated ACK – ACK with repeated sequence number means same as NACK.
- rdt 3.0 introduced timeout in case packet lost completely not just corrupted.

Test your understanding

- Which version of rdt?
  - Assumes no losses or corruption
  - Works when packets can be corrupted but not lost.
  - Works with corruption and packets being lost or delayed.
  - Pause the video to write down your answer.

Test your understanding

- Which version of rdt?
  - Assumes no losses or corruption
    - rdt 1.0 – assumes a reliable channel
  - Works when packets can be corrupted but not lost.
    - rdt 2.2 – introduces ACK and sequence number to deal with corruption
  - Works with corruption and packets being lost or delayed.
    - rdt 3.0 – introduces timeout to cope with loss
  - If you got it wrong review rdt.

Performance of rdt3.0

- rdt3.0 is correct, but performance is very bad
- e.g.: 1 Gbps link, 15 ms delay, 8000 bit packet
- Define round-trip-time as time to propagate there and back -- (2 x end-to-end delay)

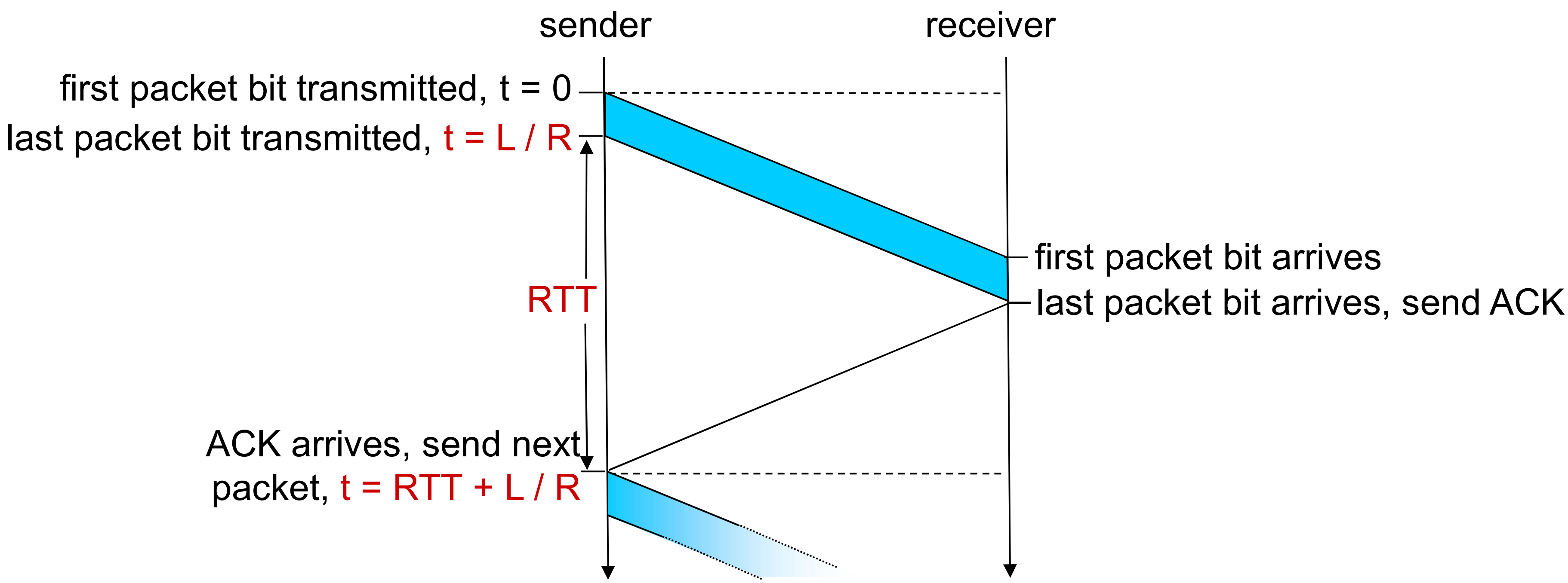
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - 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 pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
- Badly designed protocol limts how we use resource.

rdt3.0: stop-and-wait operation

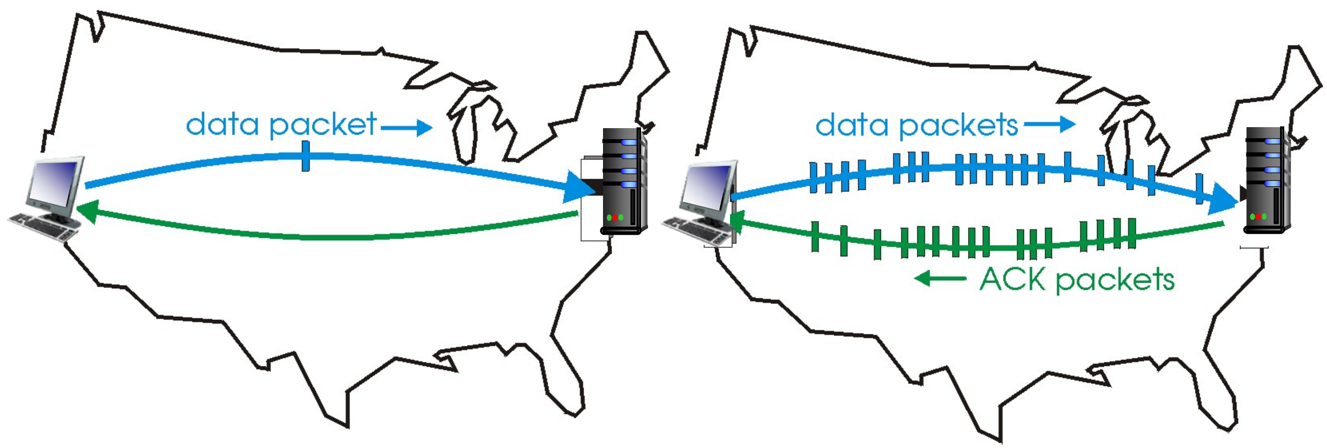


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets (packets with no ACK as yet)

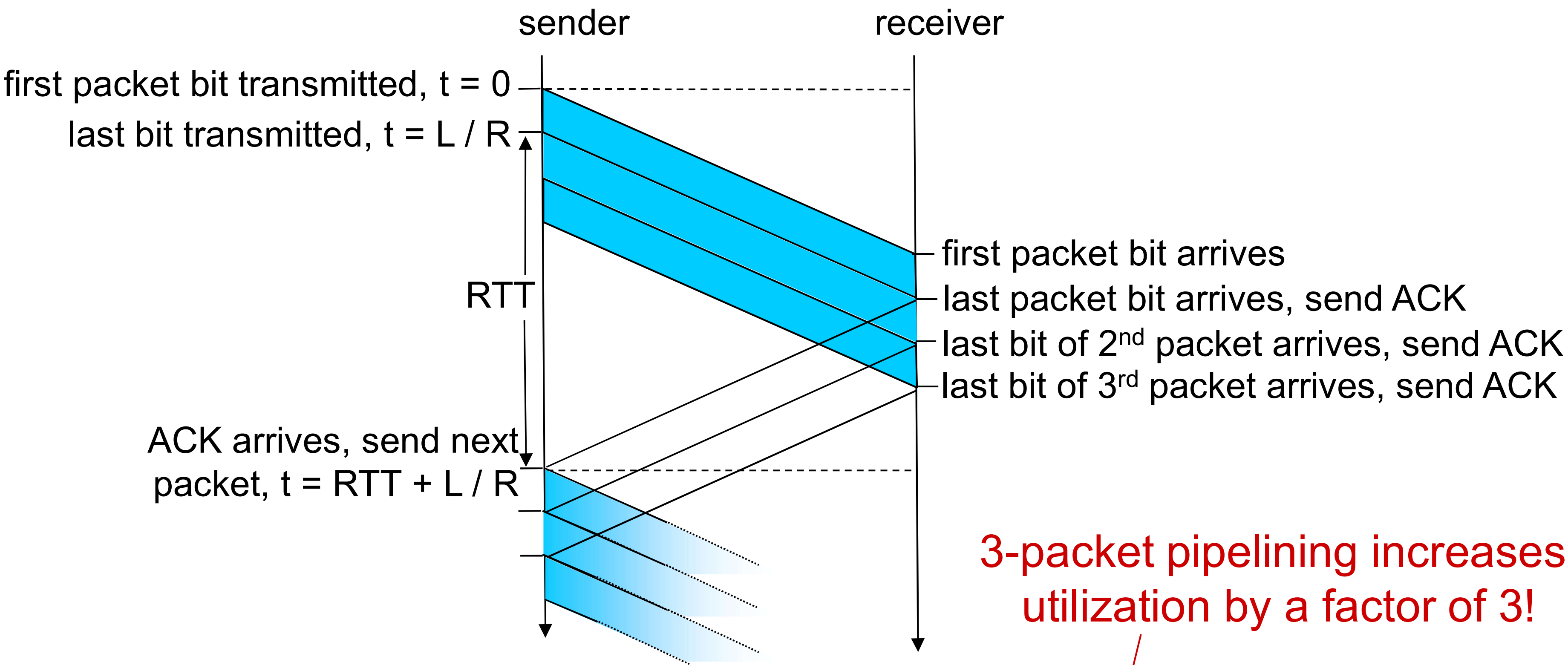
- range of sequence numbers must be increased
- buffering at sender and/or receiver



(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*

Pipelining: increased utilization



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

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

Test your understanding

- 1250 byte packet. 100 Mb/s link.
- End to end delay = 5ms

- What is the utilisation if we use stop and wait.

- What is the utilisation if we pileline 25 packets.

Test your understanding

- 1250 byte packet. 100 Mb/s link.
- End to end delay = 5ms

  L= 1250x8 bits = 10,000 bits

  Transmission delay = L/R = 10,000/100,000,000 = 0.0001 s = 0.1ms
- What is the utilisation if we use stop and wait.
- U = (L/R)/ (RTT+L/R) = 0.1/(10 + 0.1) ~ 0.01*
- What is the utilisation if we use stop and wait.
- U= 25* (L/R)/ (RTT+L/R) ~0.25
  If you did't get it review slides 8-11

Pipelined protocols: overview

Go-back-N:

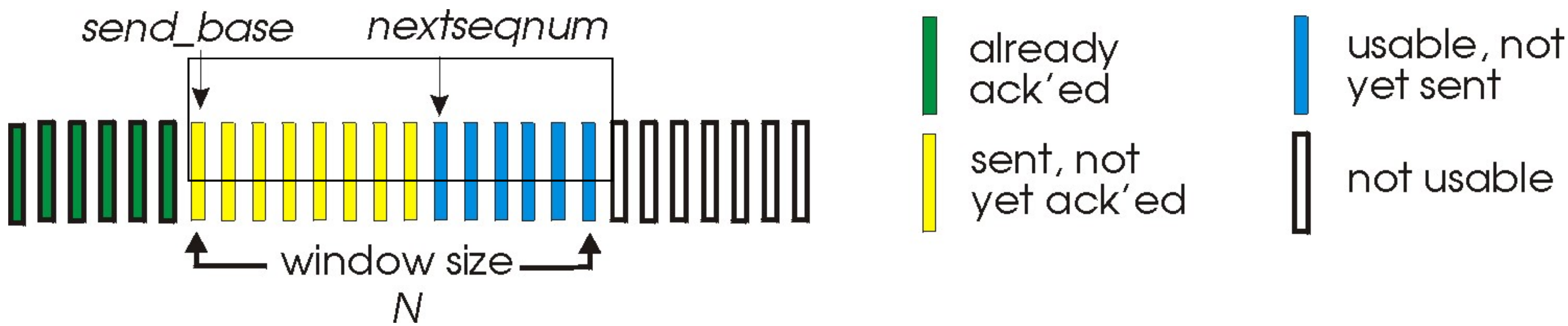
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ACK*
  - doesn't ACK packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N packets which it has not seen ACK for in "pipeline"
- Receiver sends *individual ACK* for each packet
- sender maintains timer for each packet with no ACK
  - when timer expires, retransmit only that unacked packet

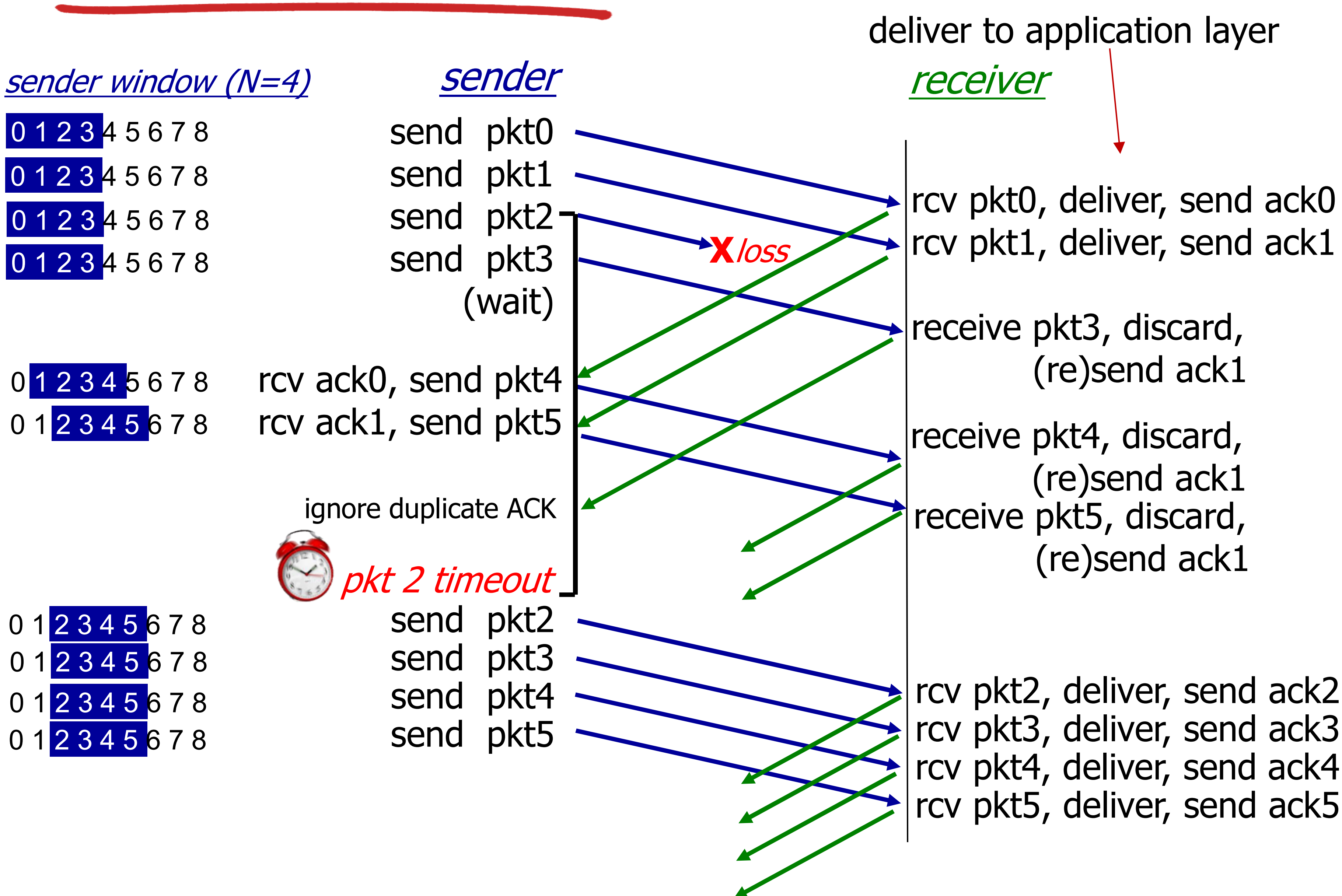
Go-Back-N: sender

- Now we need more sequence numbers k-bit sequence number in packet header
- "window" of up to N, consecutive packets with no ACK



- ACK(n): ACKs all packets up to, including sequence # 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

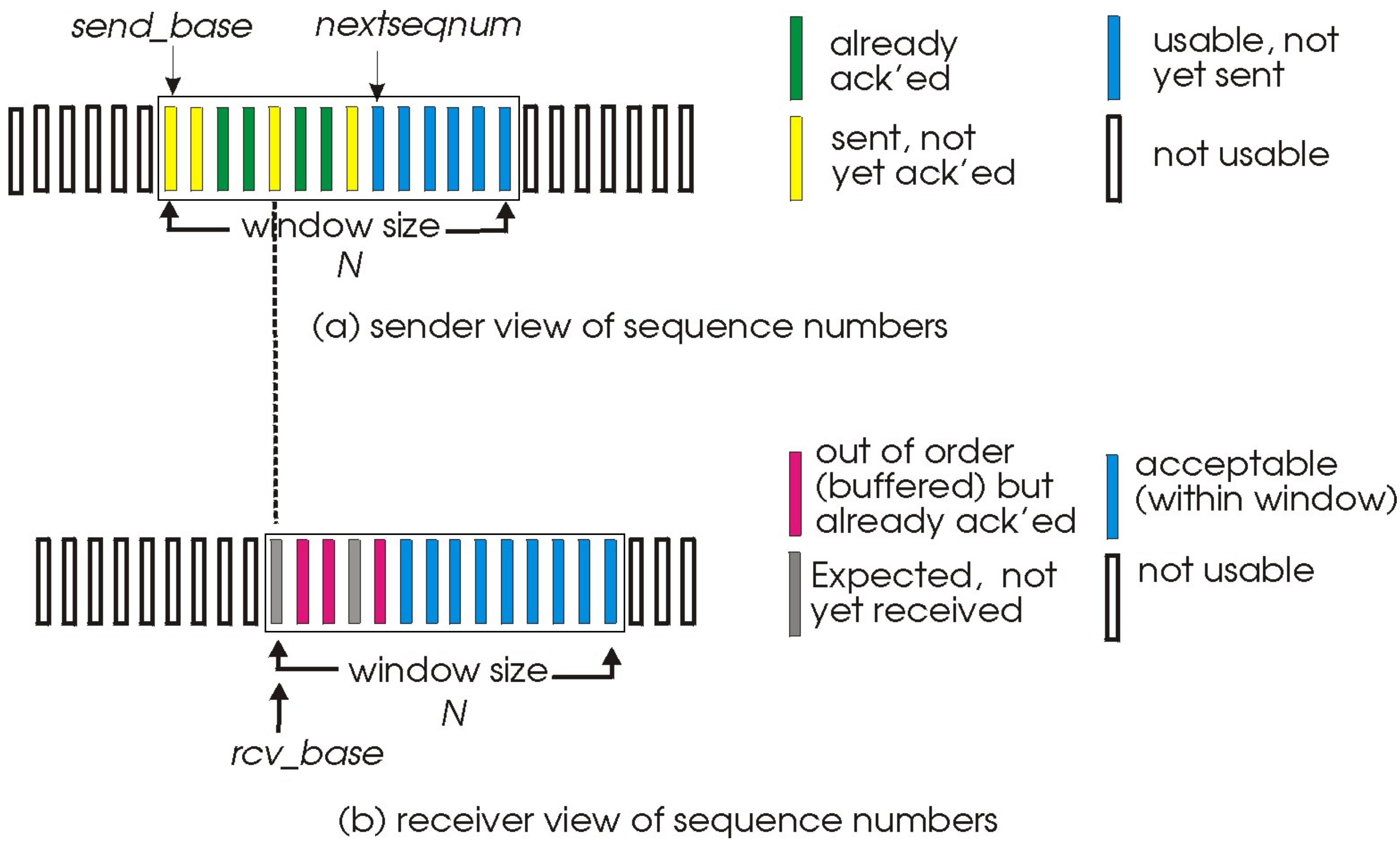
GBN in action



Selective repeat

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

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-1]:

- 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

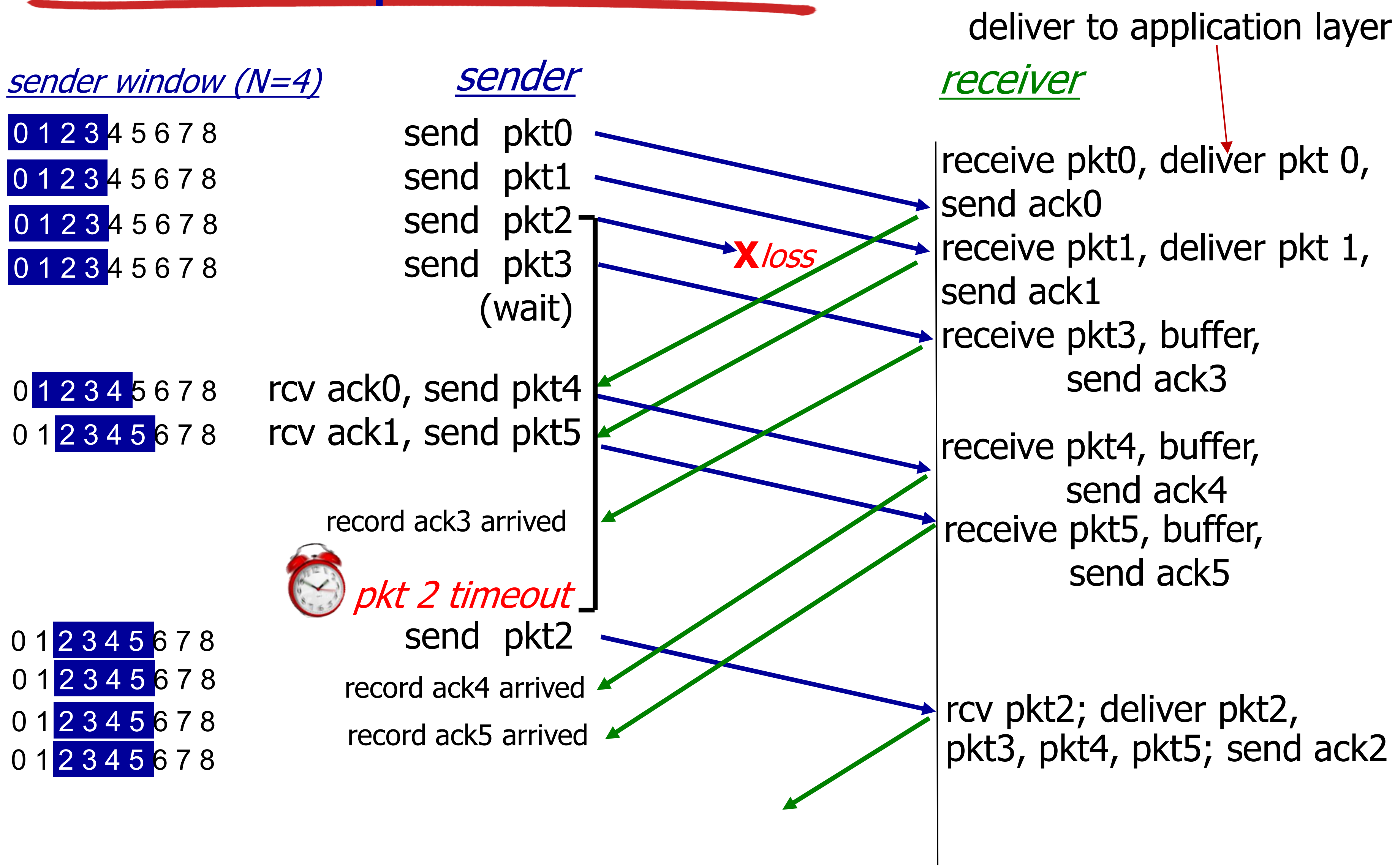
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

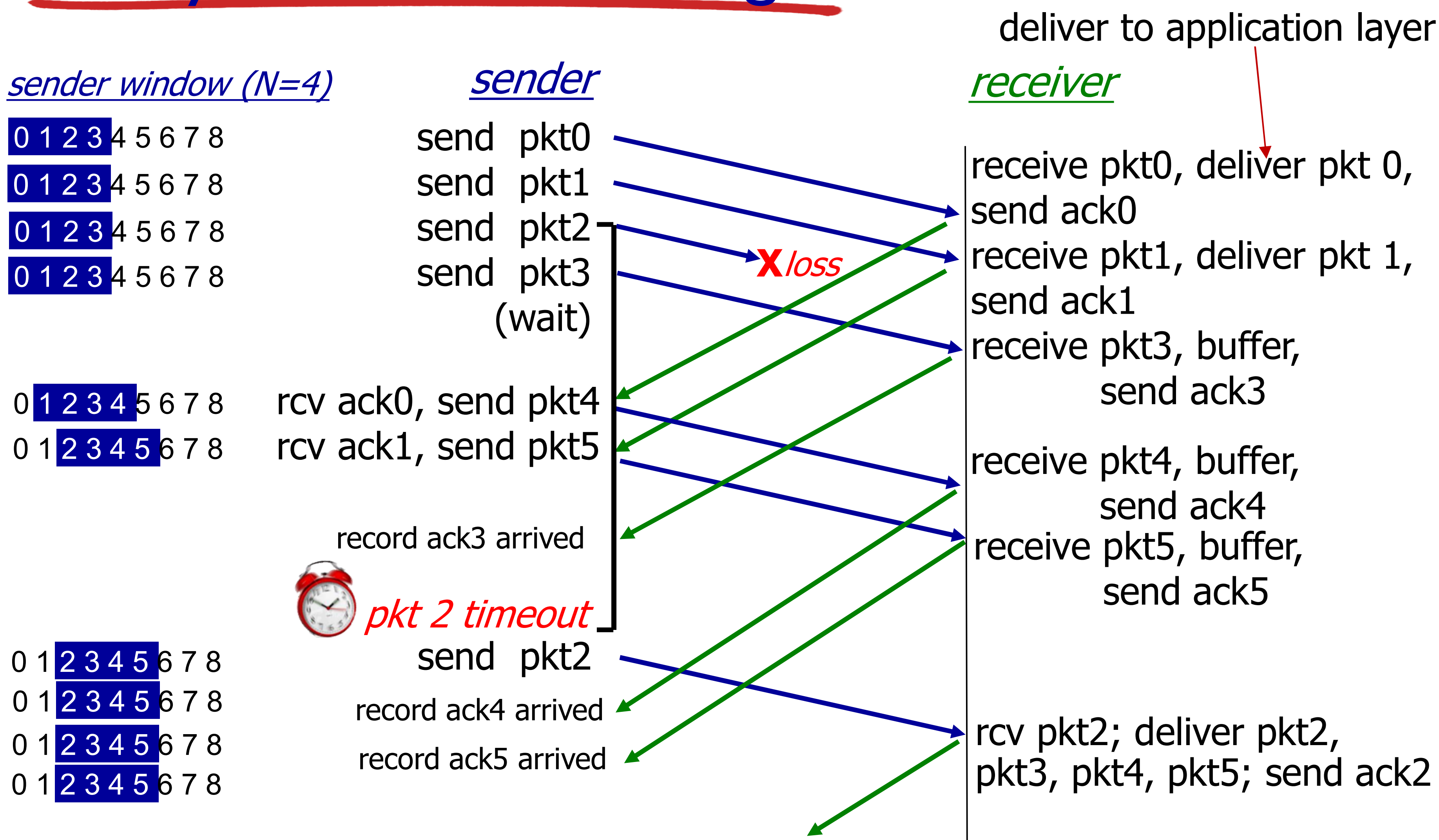
otherwise:

- ignore

Selective repeat in action

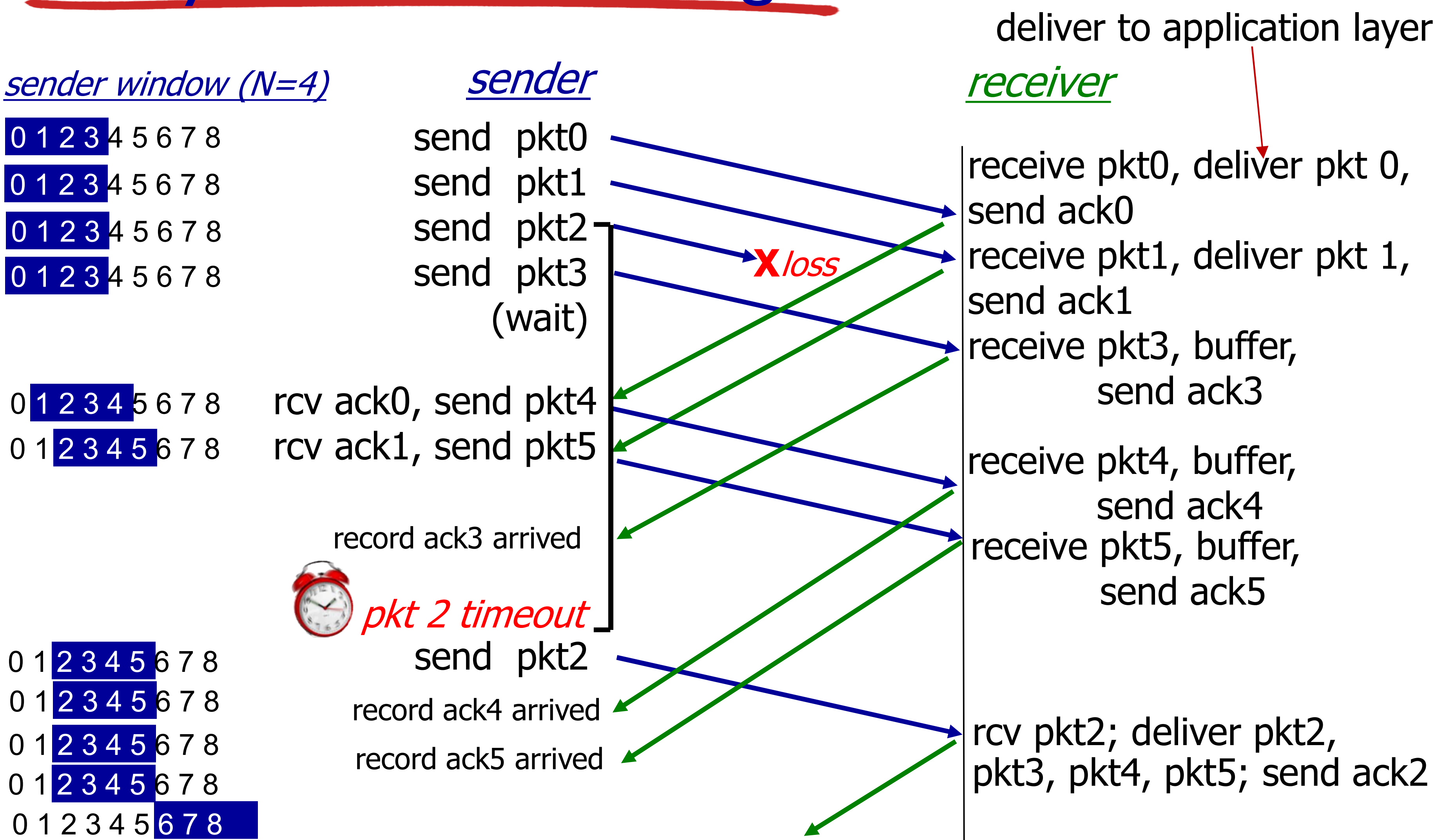


Test your understanding



What happens to the window when ACK 2 arrives?
Pause the video and write down your answer

Test your understanding



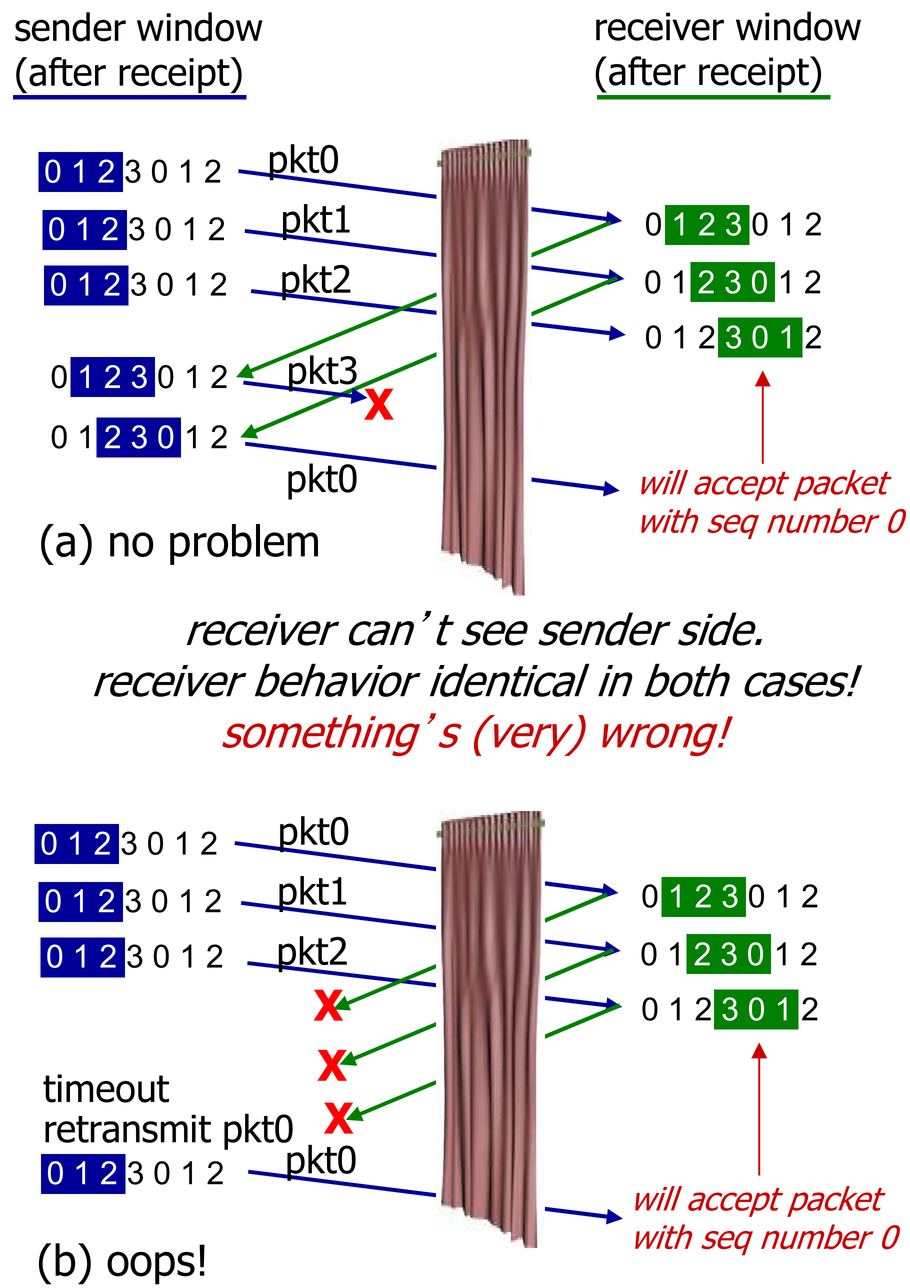
The window advances to 6 7 8 since ACKs 1-5 arrived
If you got it wrong review selective repeat

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 to avoid problem in (b)?



Transport layer: outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

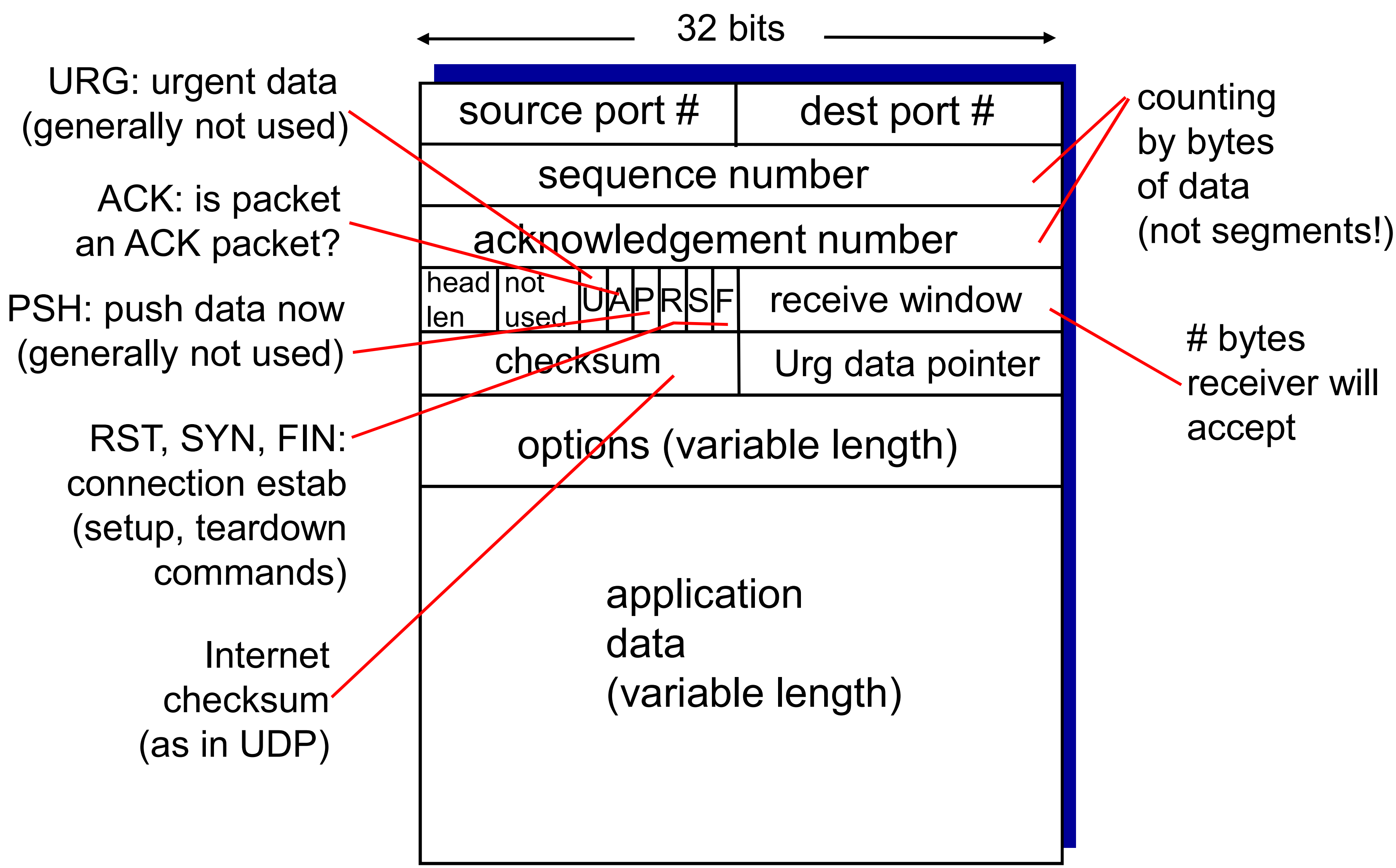
3.6 principles of congestion control

3.7 TCP congestion control

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

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

TCP header


32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
URG: urgent data (generally not used)
ACK: is packet an ACK packet?
PSH: push data now (generally not used)
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
counting by bytes of data (not segments!)
bytes receiver will accept

TCP seq. numbers, ACKs

sequence numbers:

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

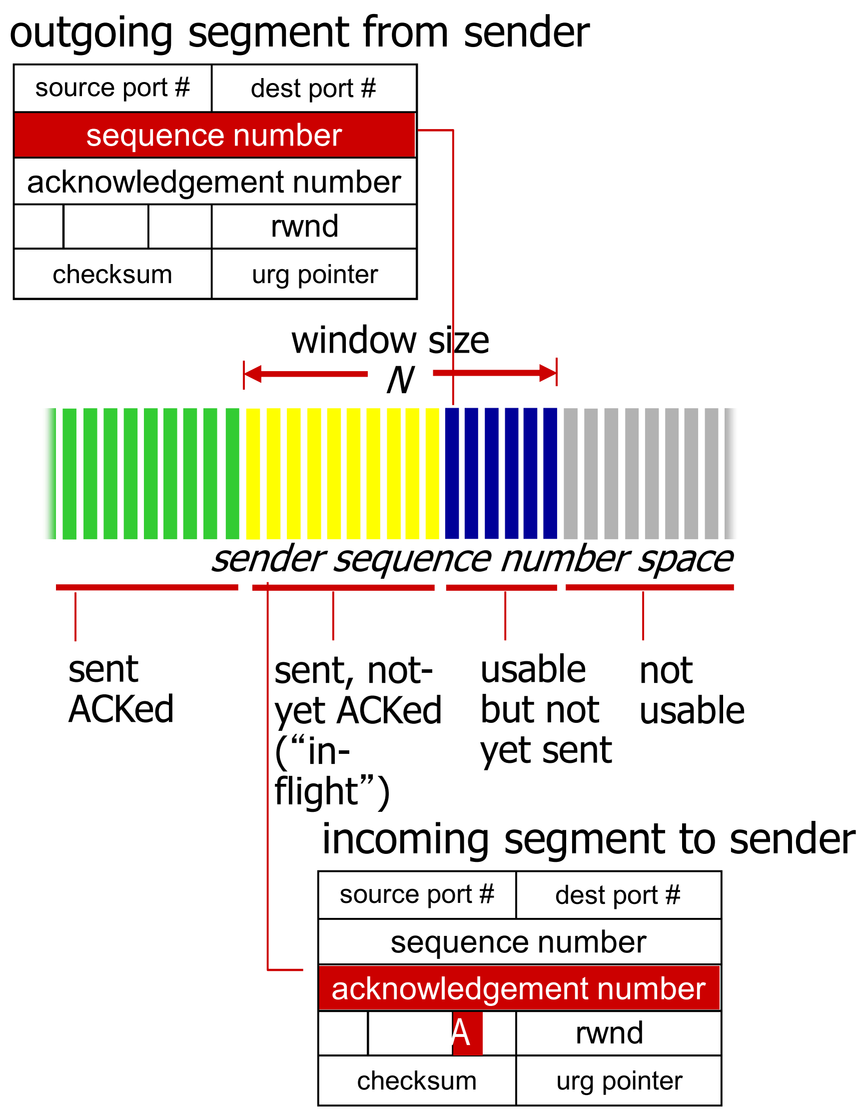
acknowledgements:

- seq # of **next byte expected from other side**

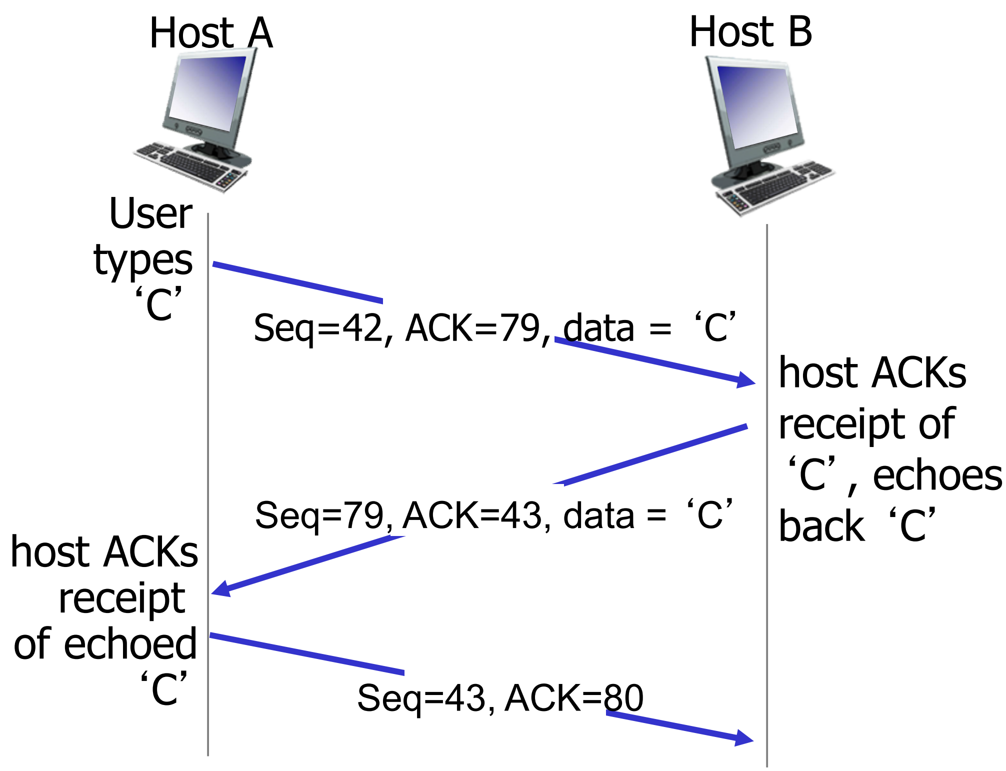
  (cumulative ACK)

Q: how receiver handles out-of-order segments

- A: TCP spec does not say, - up to those writing TCP code as long as data is delivered in order to application layer



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:* early timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

Q: how to estimate RTT?

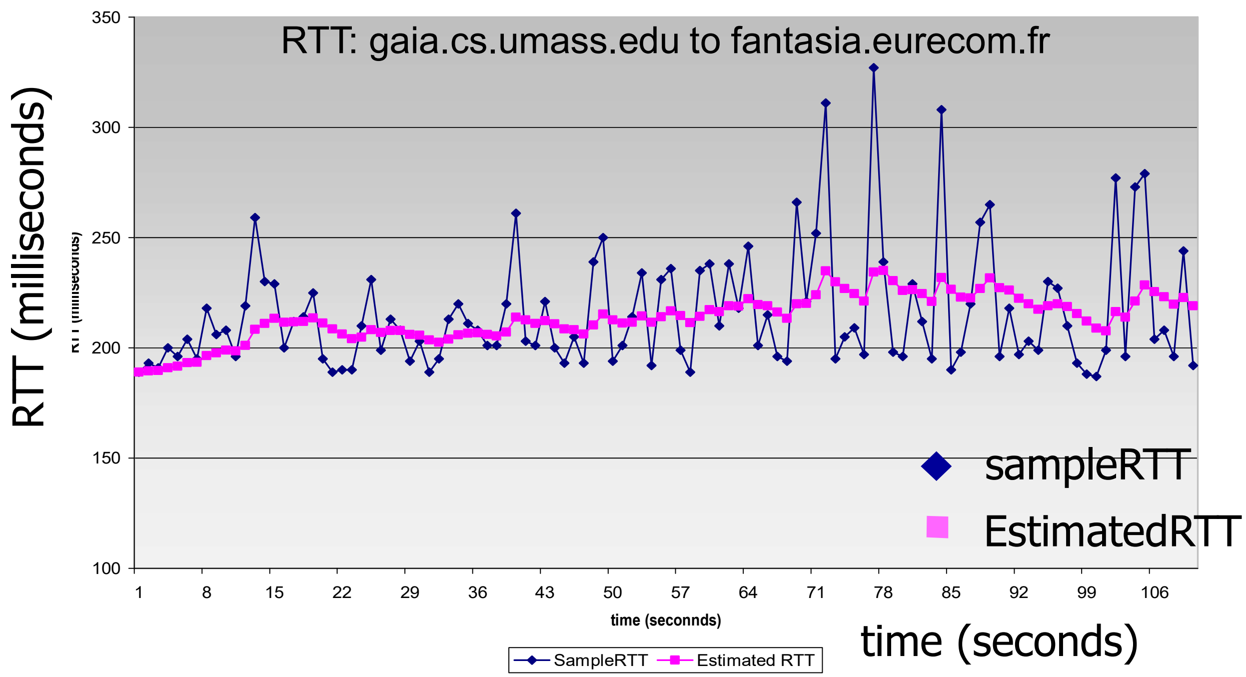
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP round trip time, timeout

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

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

EstimatedRTT_new = 0.875 · EstimatedRTT + 0.125 · SampleRTT



TCP round trip time, timeout

- Jacobsen/Karel's Algorithm
- 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)
```

Where |x| means the "absolute" value – x made positive.

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

$\uparrow$ estimated RTT $\uparrow$ "safety margin"

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

Test your understanding

- What is the problem if a TCP timeout is set too short?

- What is the problem if a TCP timeout is set too long?

- Estimated RTT is 12ms α = 0.125, Sample RTT is 8ms. Calculate a new Estimated RTT

Pause the video and write down your answer

Test your understanding

- What is the problem if a TCP timeout is set too short?
  - Packets are retransmitted needlessly (they were delayed not lost) wasting bandwidth
- What is the problem if a TCP timeout is set too long?
  - Packets take a long time to retransmit meaning your connection is slow.
- Estimated RTT is 12ms $\alpha = 0.125$, Sample RTT is 8ms. Calculate a new Estimated RTT
  - Estimated RTT= $(1- \alpha)\ 12 + \alpha\ 8$ ms = $(0.875).12 + 0.125\ .8$ ms = 10.5+1ms = 11.5 ms

If you did not get it right revise this section

Transport layer: outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 TCP congestion control

TCP sender events (simplified)

data received from application:

- 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`

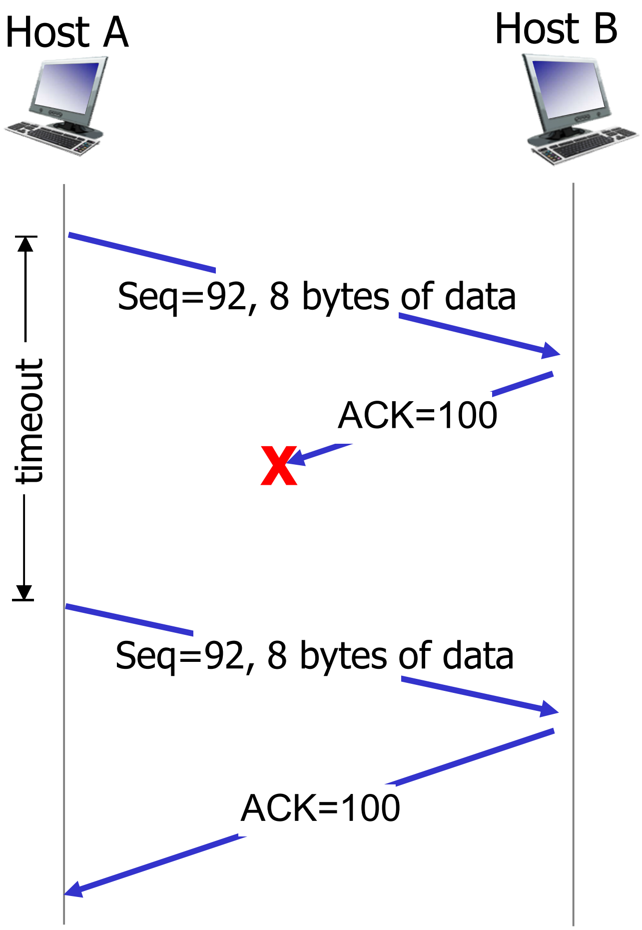
timeout:

- retransmit segment that caused timeout
- restart timer

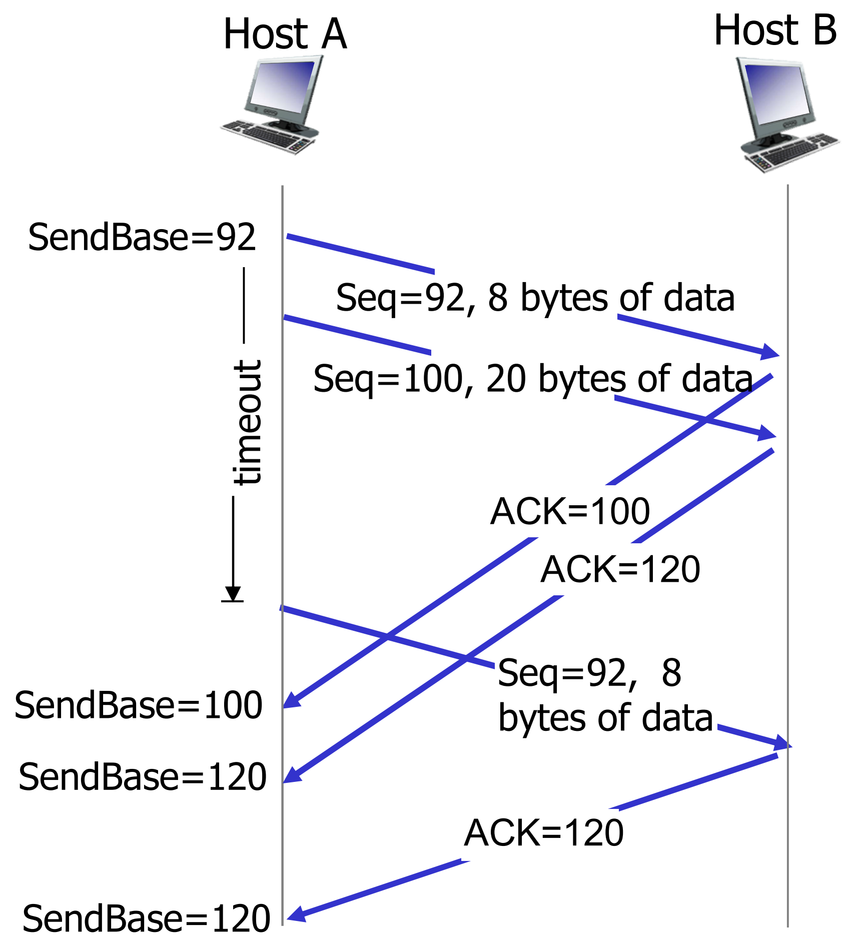
ack recieved:

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

TCP: retransmission scenarios

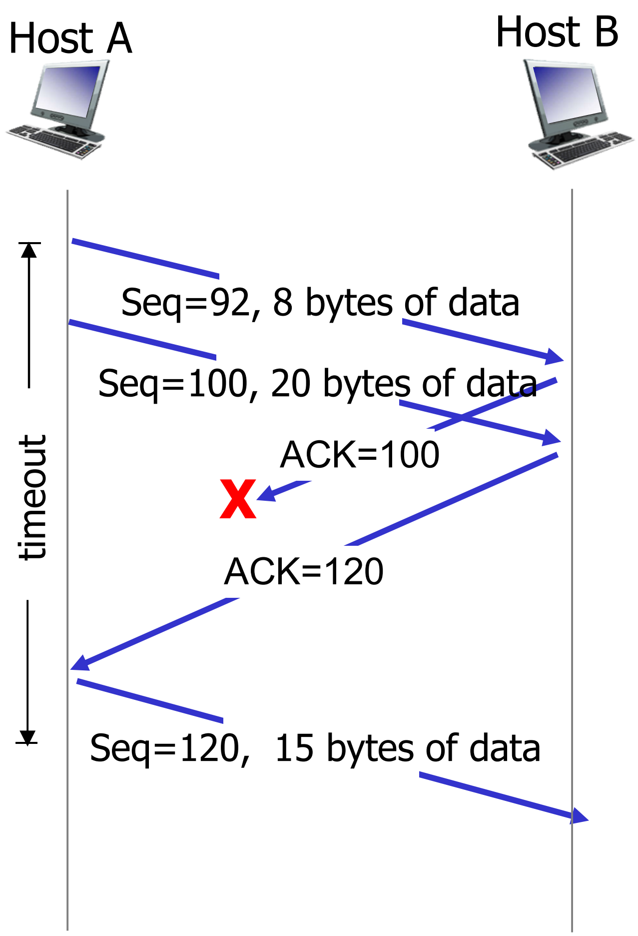


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP 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.

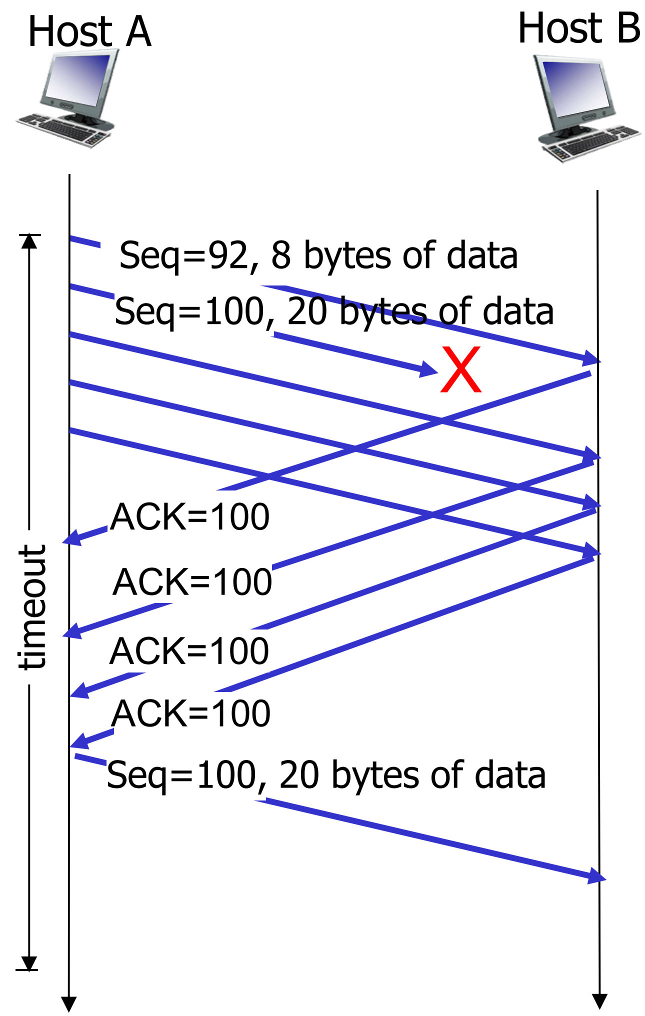
TCP fast retransmit

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

Test your understanding

- Simple TCP sender has a window size of 5. It has 10 packets to send. It sends packets 1,2,3,4,5.
- What should it do if it receives the following ACKS?
  - (What do you think happened?)
- Receives ACK for 1, 2, 3, 4, 5
- Receives ACK for 1, 3.
- Receives ACK for 1 four times.

Pause the video and write down your answer.

Test your understanding

- Simple TCP sender has a window size of 5. It has 10 packets to send. It sends packets 1,2,3,4,5.
- Receives ACK for 1, 2, 3, 4, 5
  - Send packets 6,7,8,9,10 (everything got there fine).
- Receives ACK for 1, 3.
  - Send packet 6, 7, 8. Wait for timeout for packet 4. (ACK 3 tells us packets 2 and 3 must have arrived – if packet 2 had not arrived ACK 1 would be generated.)
- Receives ACK for 1 four times.
  - Send packet 6. Send packet 2 (Triple duplicate). Packet 2 is lost, corrupted or delayed (hence receiver sends duplicate when it receives packets 3, 4, 5)

If you did not get the right answer review the last section.

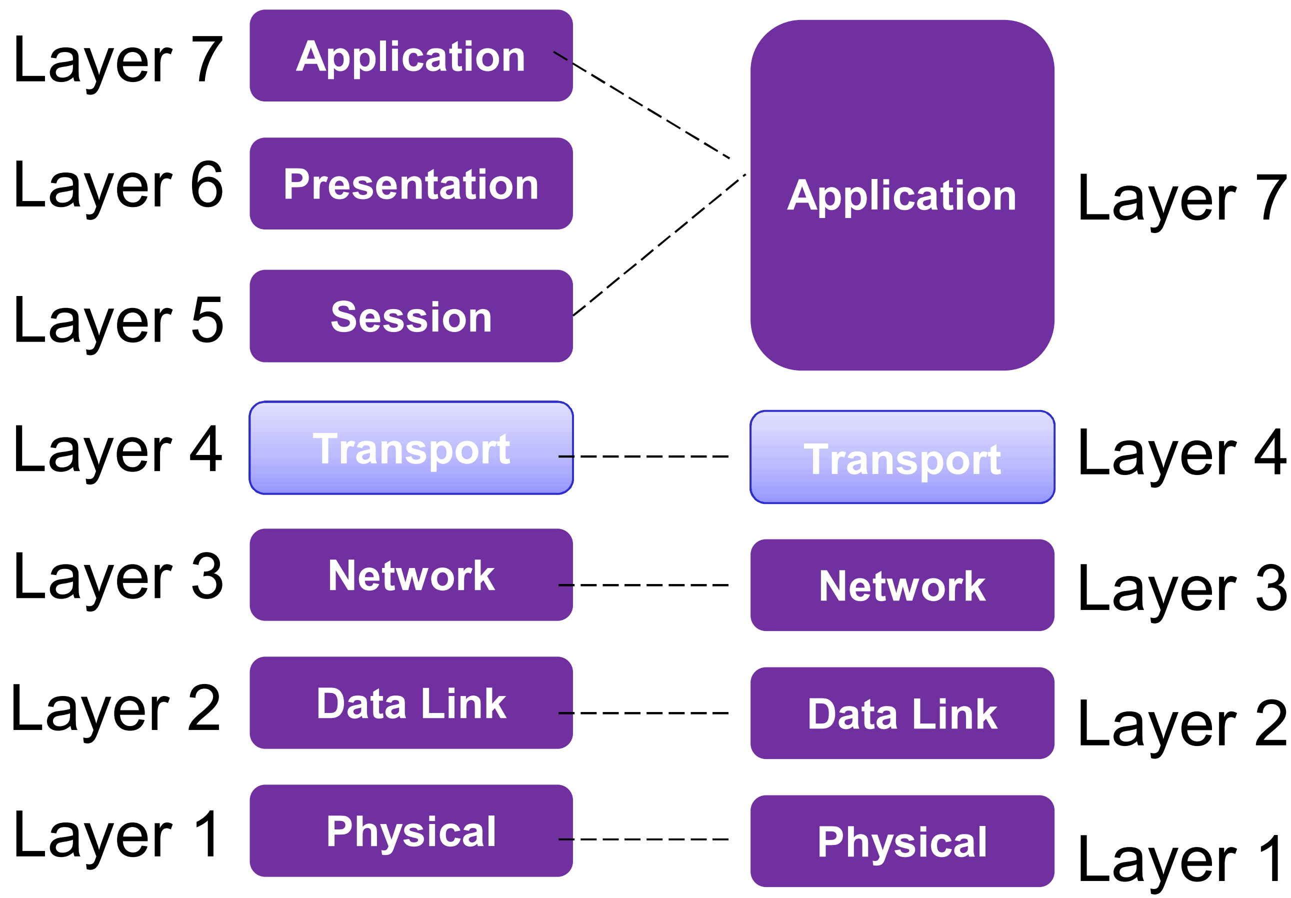
What have we learned?

- We learned about "pipelining" – sending more than one packet "in flight" at once. This makes transfer much more efficient.
- "Go back N" and "Selective repeat" – two ways to send several packets.
- We have seen how TCP uses SEQ and ACK to send the correct packets and retransmit missing ones.
- Triple duplicate ACK used to hint packet is "really lost" not just out of order – fast restransmit
- Jacobsen/Karel's Algorithm used to get "smooth" but "safe"estimate of RTT (and variance) and hence set timeout.

Structure of course

- Part A
  - Introduction to IP Networks
  - The Transport layer (part 1)
- Part B
  - The Transport layer (part II)
  - The Network layer (part I)
  - Class test
- Part C
  - The Network layer (part II)
  - The Data link layer (part I)
  - Router lab tutorial (assessed lab work after this week)
- Part D
  - The Data link layer (part II)
  - Network management and security
  - Class test

Transport Layer



Transport layer: outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

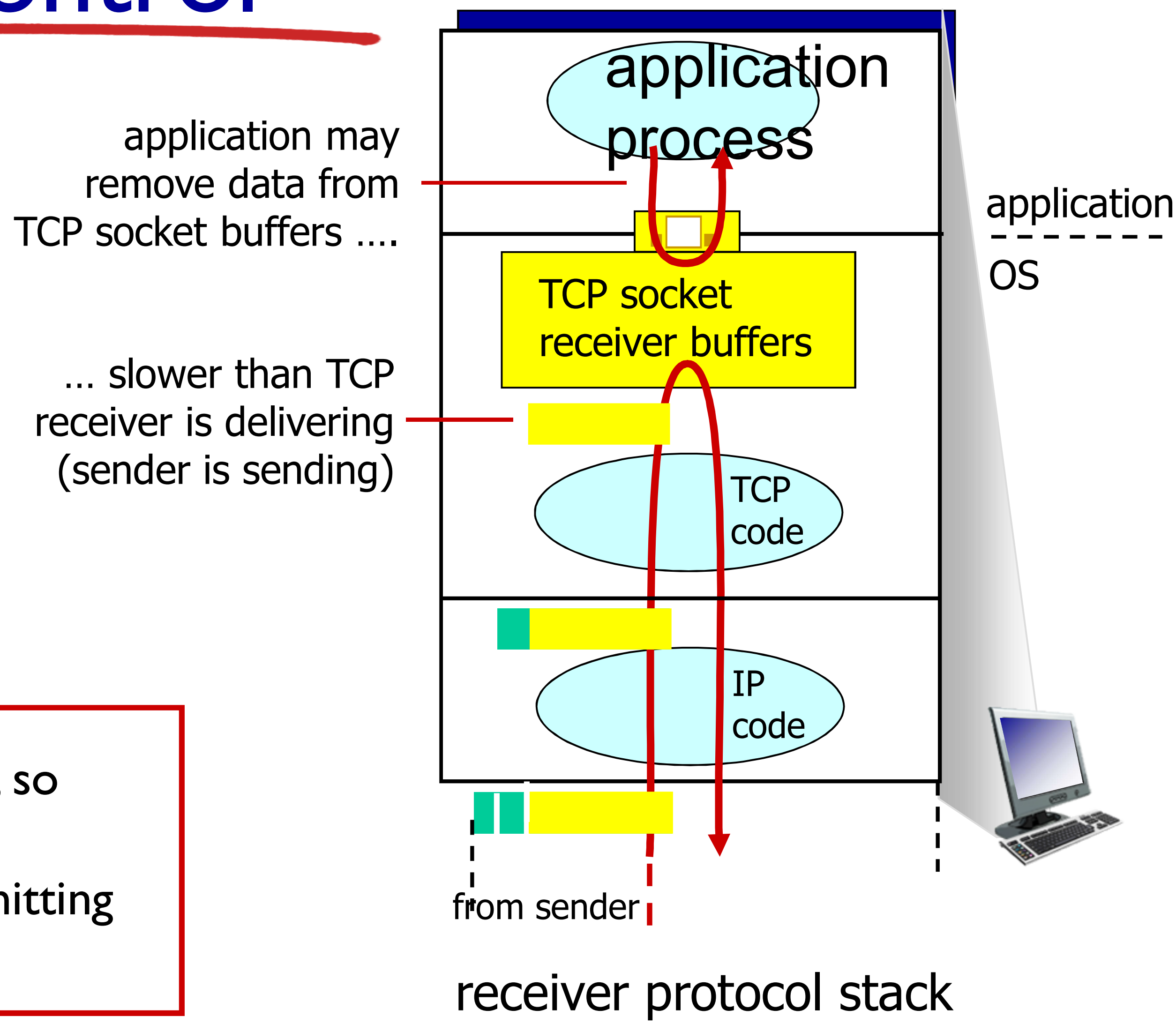
3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 TCP congestion control

TCP flow control

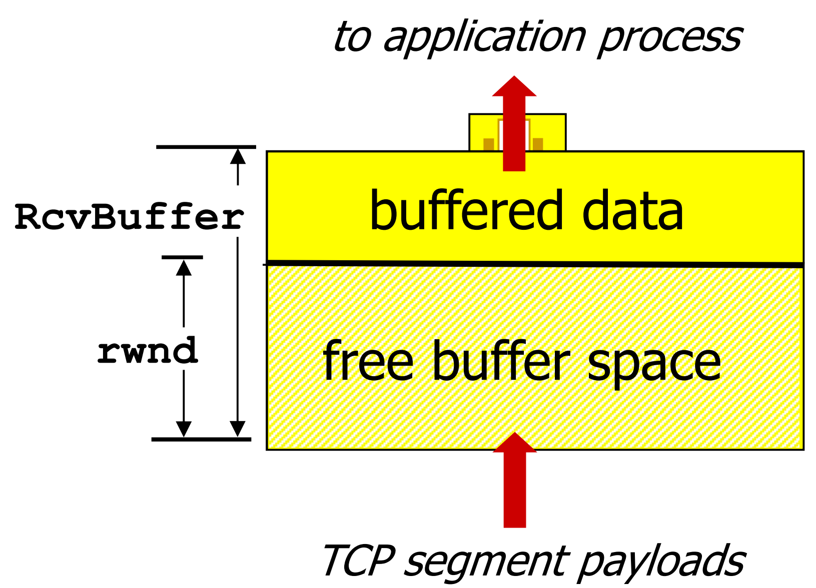


flow control

receiver controls sender, so sender will not overflow receiver 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



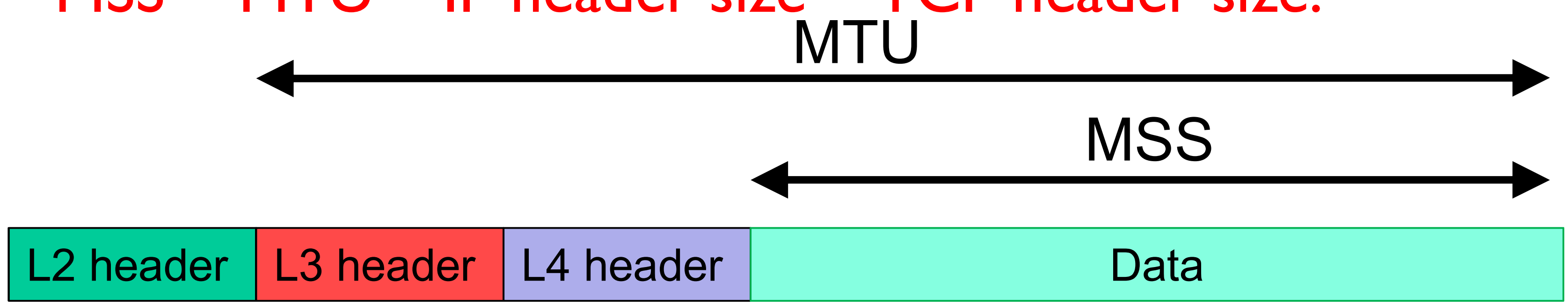
receiver-side buffering

`rwnd` = receive window

MSS and MTU

- MSS (mentioned in Nagle algorithm) is a parameter specifying the largest amount of data in a single IP datagram that should be sent by a remote host.
- MTU is a parameter specifying the largest amount of data that a communication protocol or system can pass onwards. For example, standards (e.g. Ethernet) can fix the size of an MTU, or systems (such as point-to-point serial links) may set MTU at connect time.
- MSS size is set according to MTU:

  MSS = MTU – IP header size – TCP header size.



Nagle's algorithm

- A problem can occur when an application generates data very slowly.
- Consider, ssh or telnet that generate data only when a user types.
- TCP will send the data as it arrives at the send buffer if there is space left in the send buffer.
- This means (for ssh/telnet) one packet sent every time user hits key.
- Overhead of this is huge (TCP header + IP header + frame header to send one byte).
- Cure is known as "Nagle's algorithm".

Nagle's Algorithm

1. The sending TCP sends the first piece of data it receives – no matter no small or large
2. Sending TCP accumulates data in the buffer and waits until one of the following before sending the segment:
   - The receiving TCP sends an acknowledgement
   - Data has accumulated to fill a maximum size segment
3. Repeat step 2

- Note: Sometimes Nagle's algorithm should be switched off – e.g. when fast interaction is vital and you want small packet sizes to be sent.

Silly Window Syndrome

- Silly Window Syndrome occurs when the TCP system is forced to send very small packets. Named because window size is "silly".
- This can happen in two separate ways:

1. Sender produces data very slowly.
   - Same problem as Nagle's algorithm.
2. Receiver processes data very slowly.
   - Single byte or small number removed from full receive buffer.
   - Sender is informed of opportunity to send small number of bytes and immediately sends filling buffer.
   - Process repeats.
   - Cure – receiver does not advertise windows that would cause sender to send small amounts of data.

Check your understanding

- What is the reason for flow control?
- Who sends rwnd?
- What problem might be caused by a very very small rwnd? (How might this happen?)
- Pause the video write down your answers.

Check your understanding

- What is the reason for flow control?

  Stop the sender sending too much for the receiver

- Who sends rwnd?

  The receiver sends it – it is the “receiver window”. It says how much buffer space the receiver has.

- What problem might be caused by a very very small rwnd? (How might this happen?)

  Sender might send small (inefficient) packets. This might happen if receiver is very slow – processing one or two bytes at a time.

If you are puzzled review the previous section.

Transport Layer outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

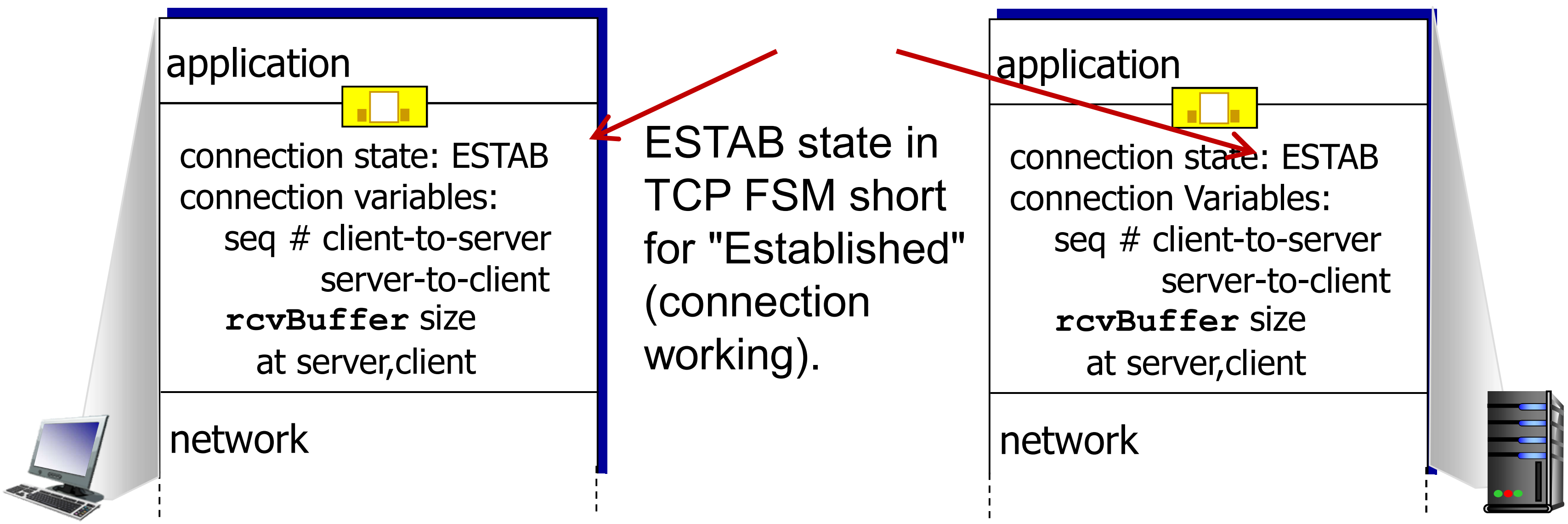
3.6 principles of congestion control

3.7 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

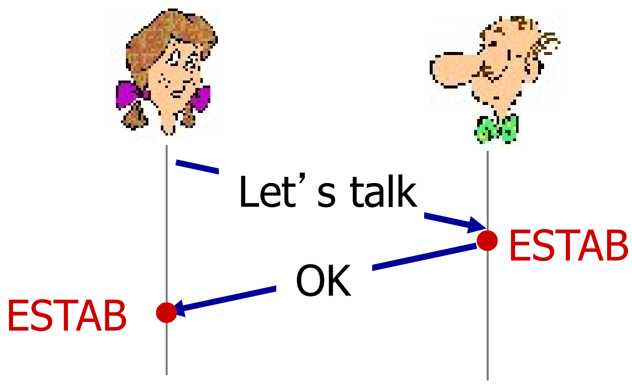


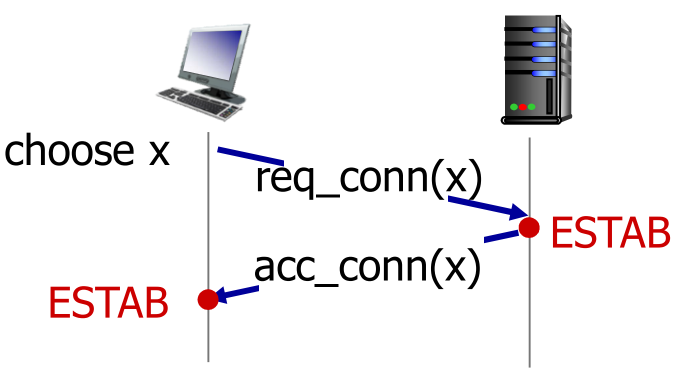

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to establish a connection

2-way handshake:



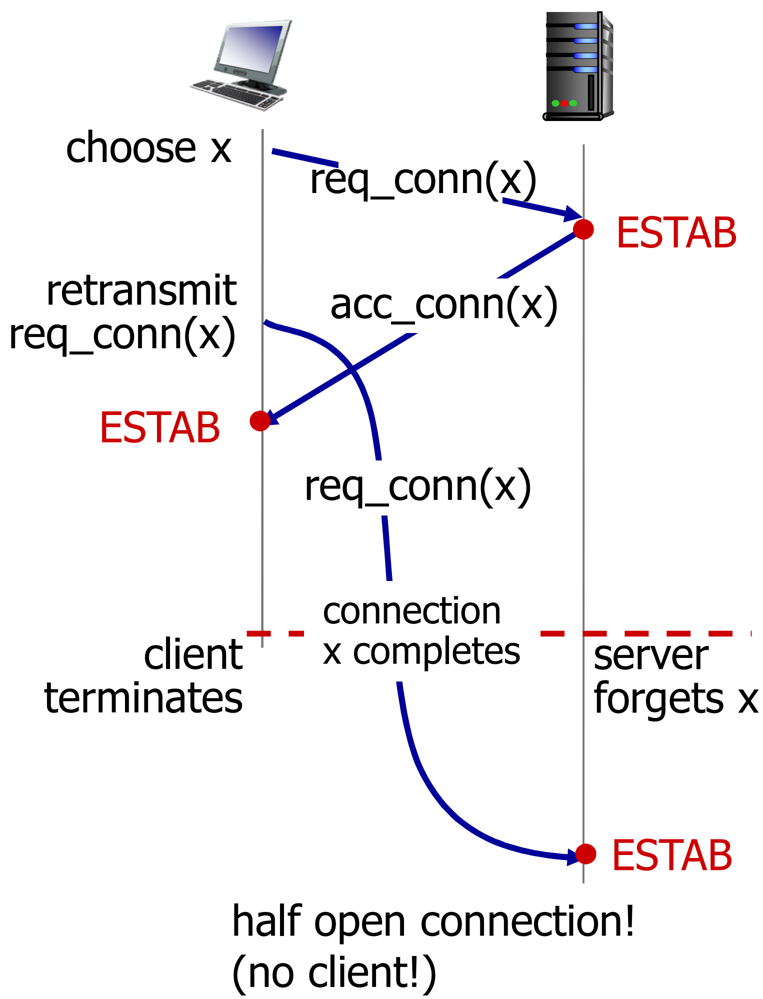


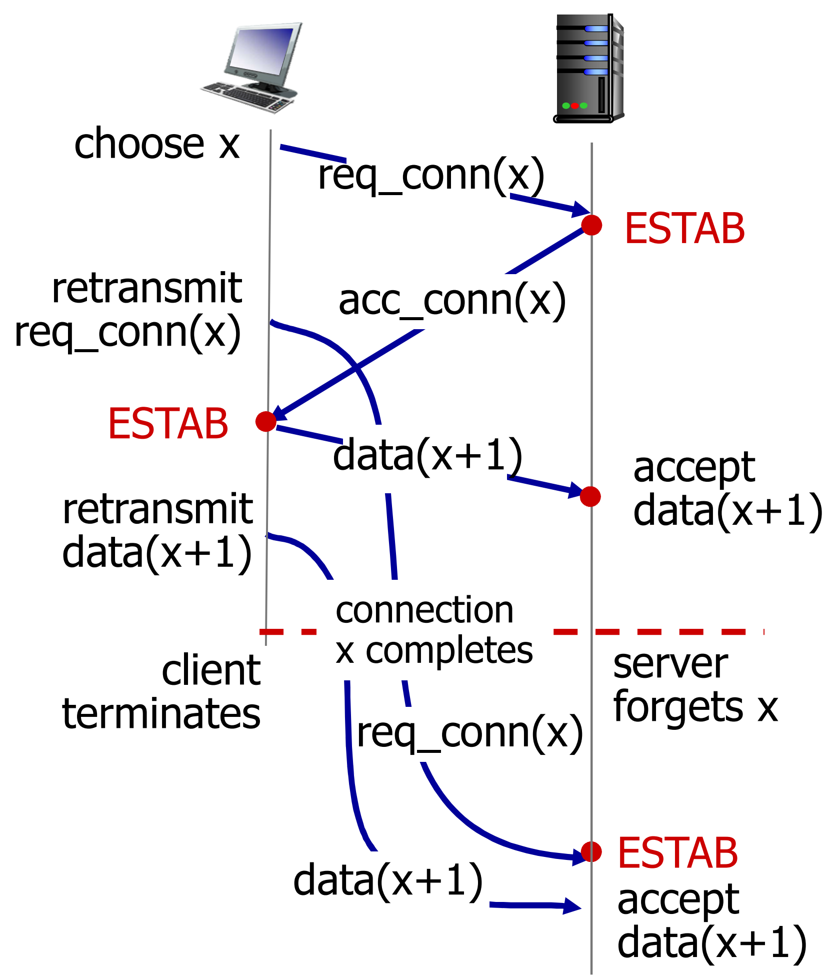
Q: will 2-way handshake always work in network?

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

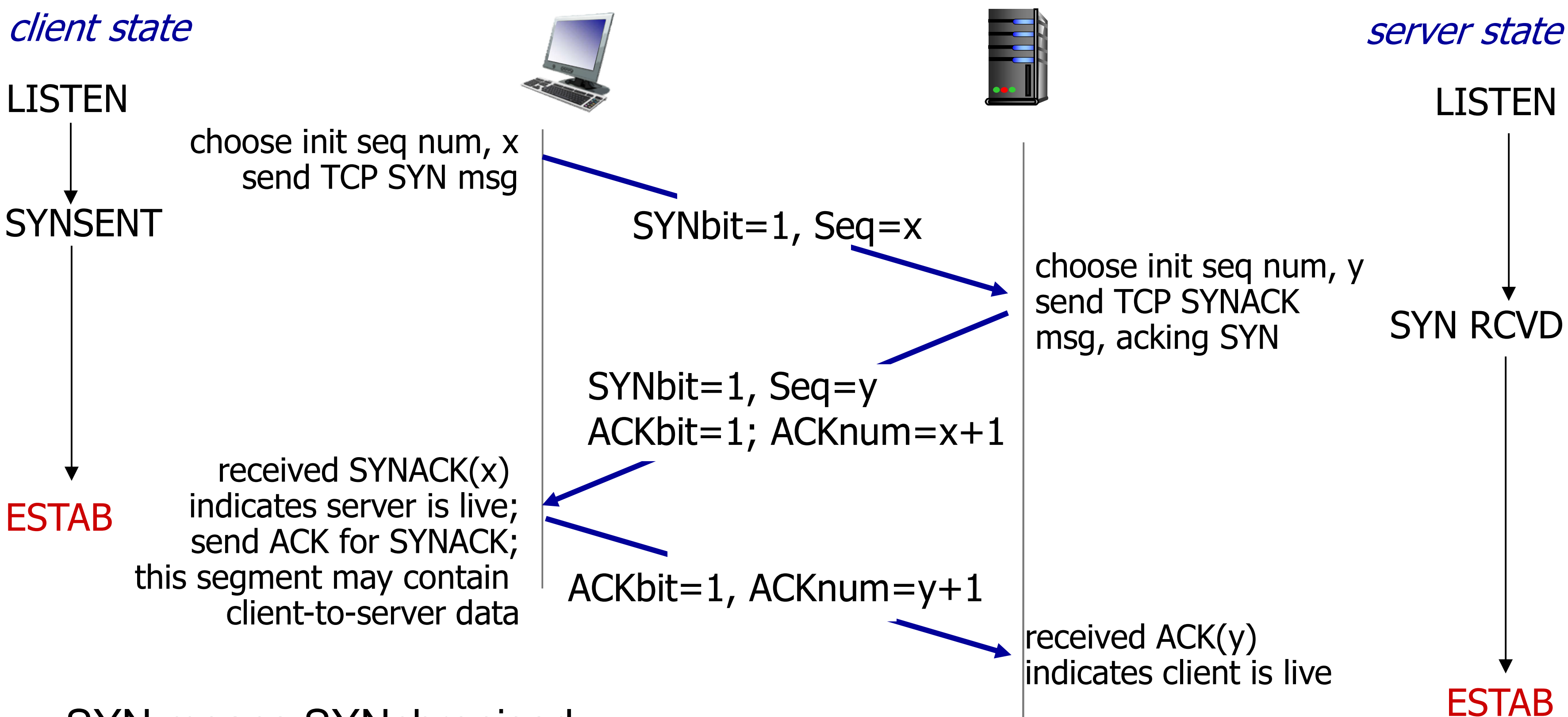
Agreeing to establish a connection

2-way handshake failure scenarios:



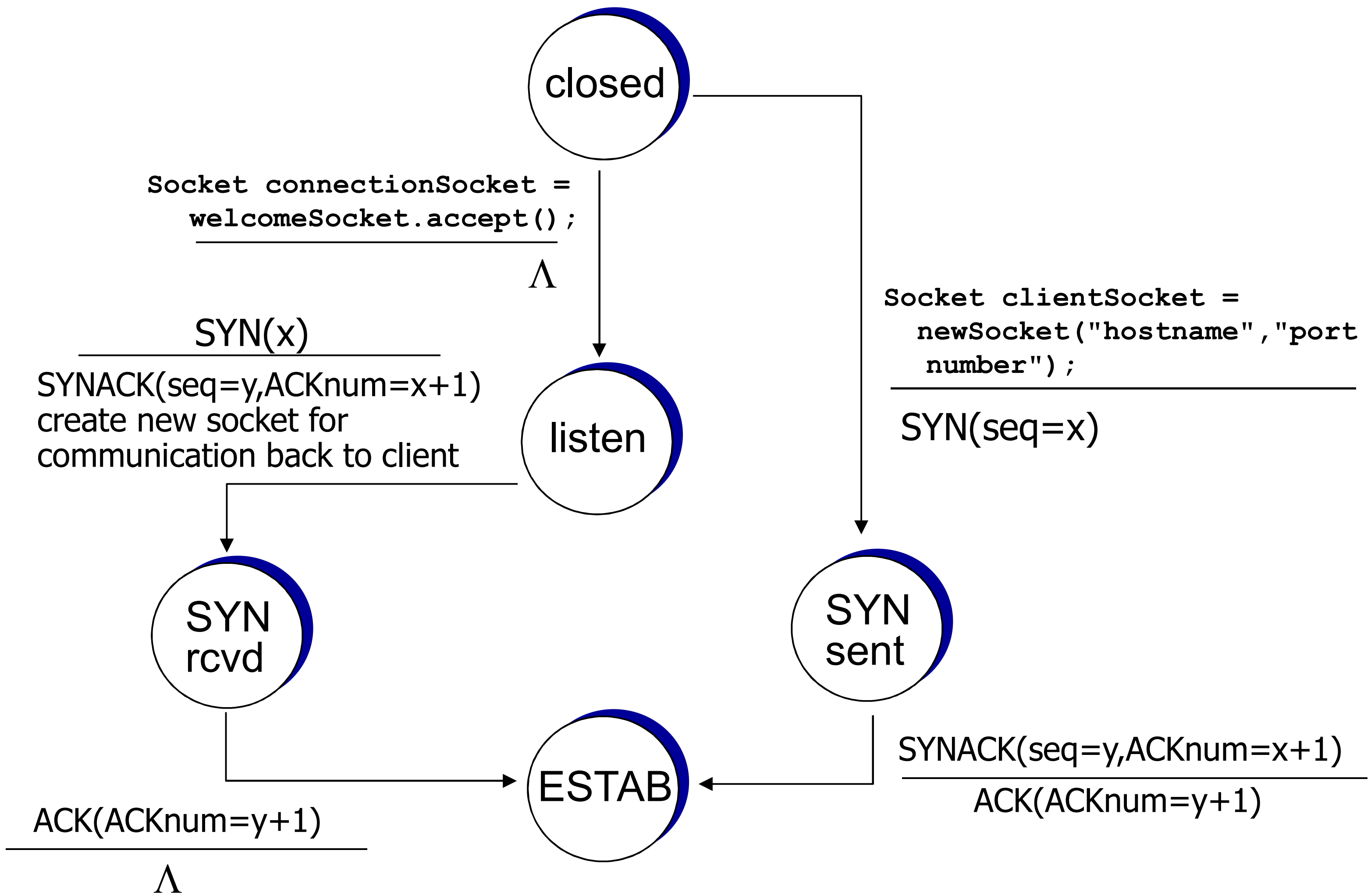


TCP 3-way handshake



SYN means SYNchronised
SYNbit = bit in TCP header saying this is SYN packet
ACKbit = bit in TCP header saying this is ACK packet

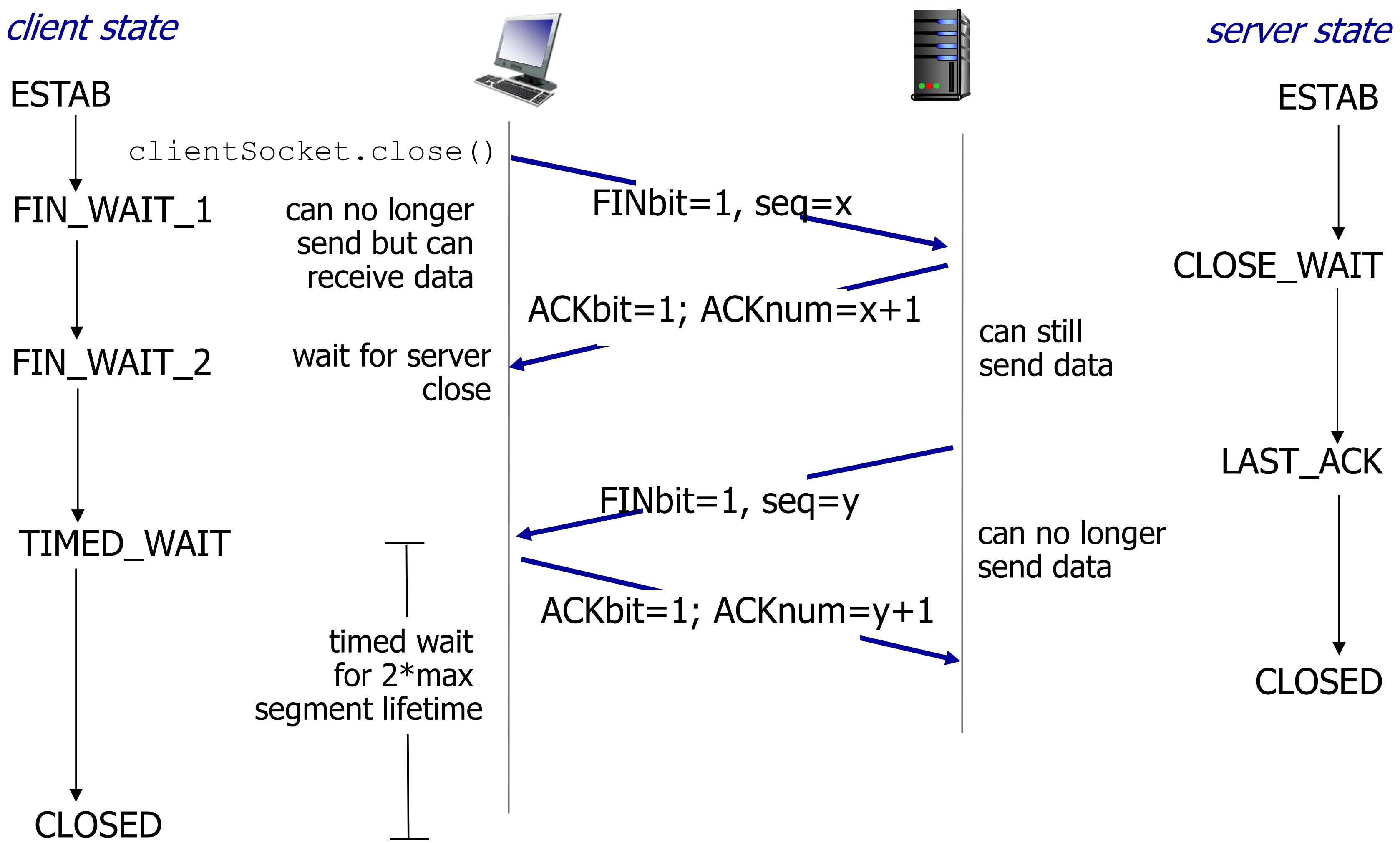
TCP 3-way handshake: Finite State Machine



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



Test your understanding

- What is the meaning of the following TCP flag combinations:

  1) SYN – ACK
  2) FIN
  3) FIN – ACK

- What are the flag combinations for the three packets in the three way handshake?
- Pause the video write down your answer

Test your understanding

1) SYN – ACK -- Response to SYN, will accept a new connection

2) FIN – I want to close the connection

3) FIN – ACK – I agree to close the connection

- What are the flag combinations for the three packets in the three way handshake?
- SYN SYN—ACK ACK
- If you had problems review the previous section

Transport layer: outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

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

3.6 principles of congestion control

3.7 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!
- how does this look?
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes and costs of congestion

- Too much traffic enters router – buffer fills up, this increases delay (and hence reduces throughput).
- Much too much traffic enters router – buffer overfills and causes loss. Packet needs to be retransmitted.
- If packet is lost after several "hops" then many resources are wasted. (e.g. Packet travels from A to B to C to D then lost at D – it has taken up space at A, B and C unnecessarily).
- Useful concept: goodput – this is the rate at which data reaches the application layer. Different from throughput because of:
  - loss
  - retransmission
  - corrupted packets

Transport layer: outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

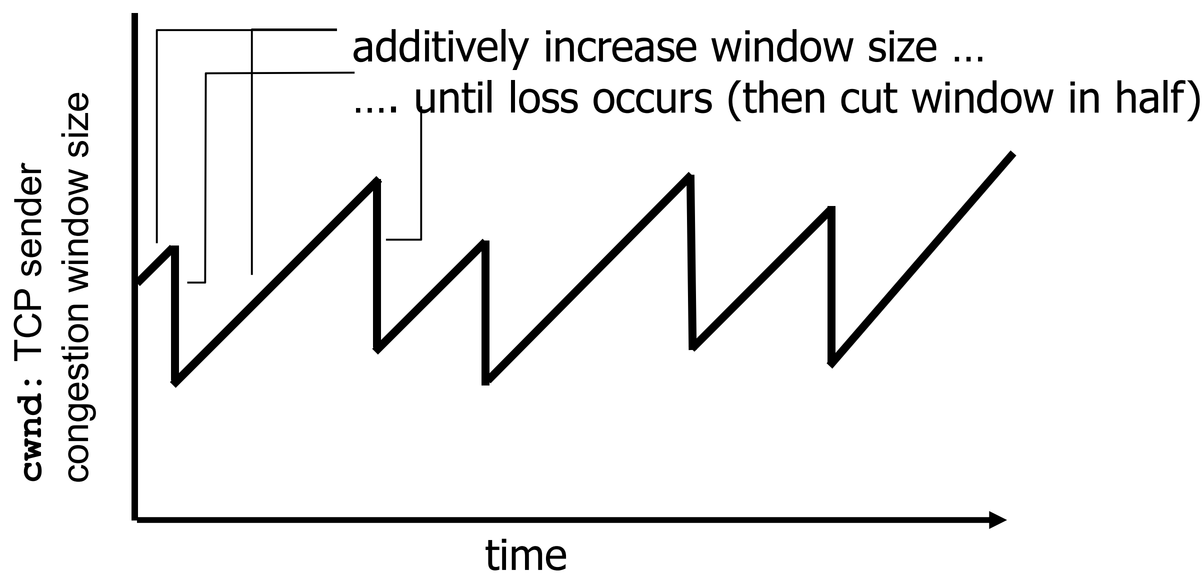
3.6 principles of congestion control

3.7 TCP congestion control

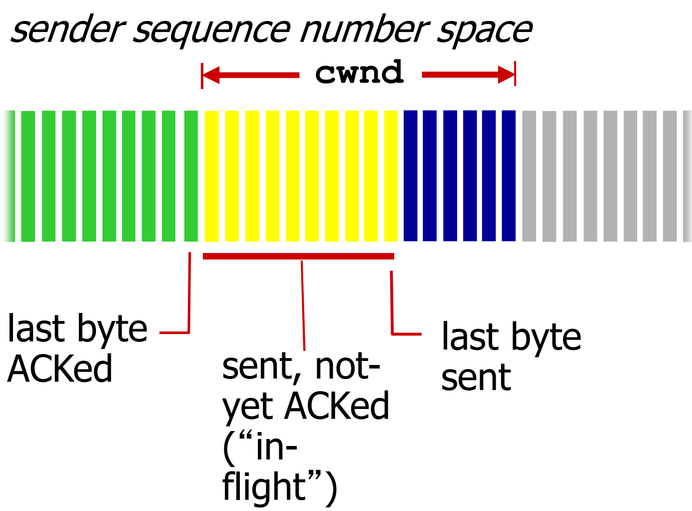
TCP congestion control: additive increase multiplicative decrease

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

AIMD saw tooth behavior: probing for bandwidth



TCP Congestion Control: details



- sender limits transmission:

```
LastByteSent-
  LastByteAcked  ≤  cwnd
```

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

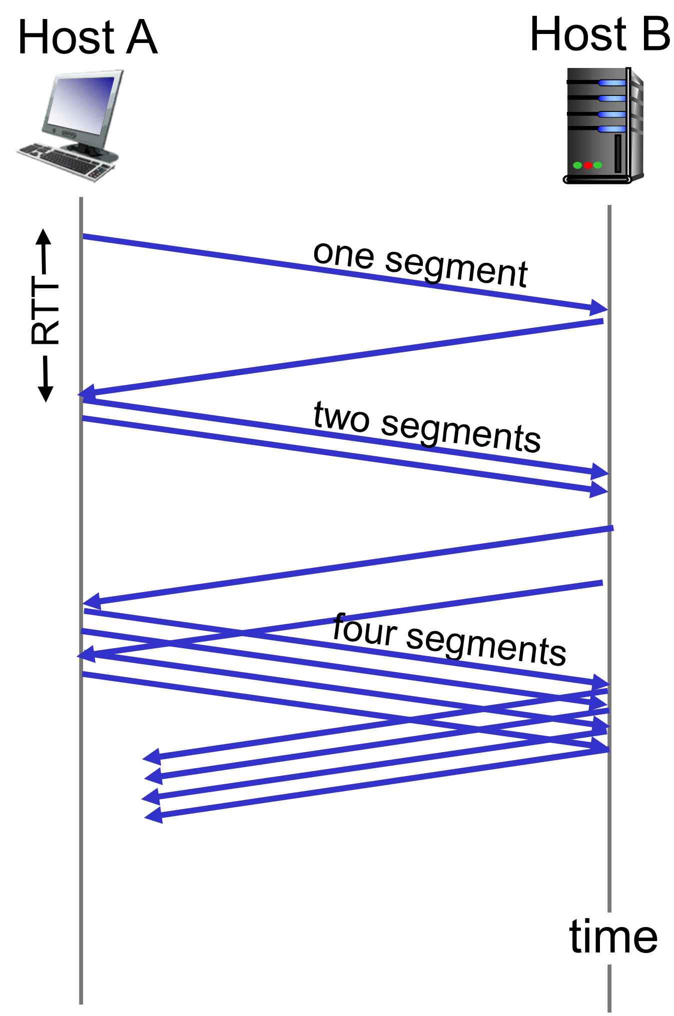
TCP sending rate:

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

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

TCP Slow Start

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



TCP flavours

- There are lots of implementations of TCP.
- The protocol specifies certain things but leaves others free.
- For example TCP protocols can choose how they want to react to duplicate ACKs or what their initial window sizes are.
- TCP protocols are sometimes named after places with casinos (gambling): Reno, Tahoe, New Reno

Test your understanding

- What are the purposes of windows rwnd and cwnd?
- In additive increase multiplicative decrease what triggers TCP to reduce cwnd?
- How does a receiver indicate is buffer is getting full and the sender must slow down?
- Pause the video write down your answer.

Test your understanding

- What are the purposes of windows rwnd and cwnd?

  rwnd = flow control

  (not too fast for receiver)

  cwnd = congestion control

  (not too fast for network)

- In additive increase multiplicative decrease what triggers TCP to reduce cwnd?

  loss (or timeout) of a packet

- How does a receiver indicate is buffer is getting full and the sender must slow down?

  Sends smaller rwnd

- If you had problem review previous section.

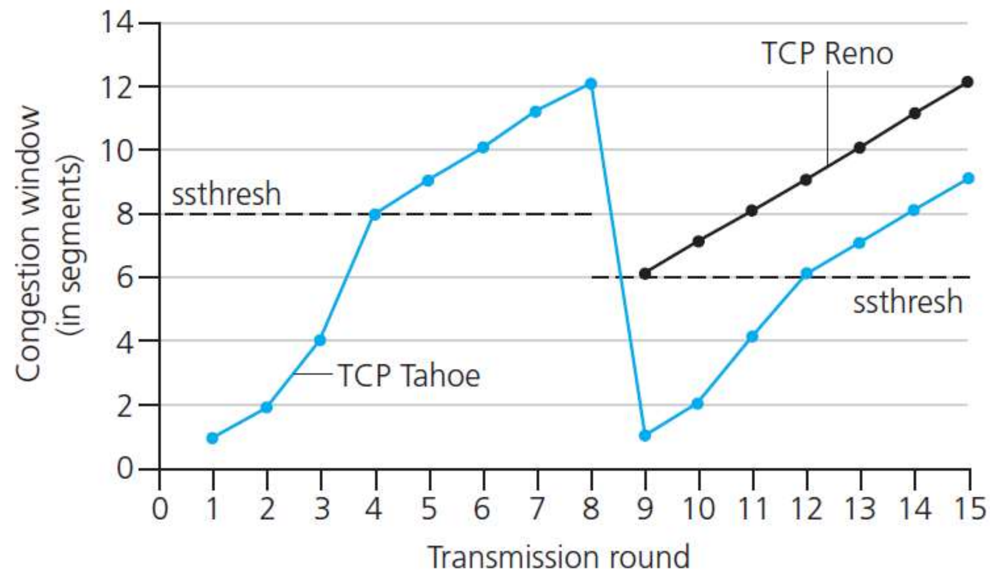
TCP: detecting, reacting to loss

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

TCP: switching from slow start to Congestion Avoidance

Q: when should the exponential increase switch to linear?

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



Implementation:

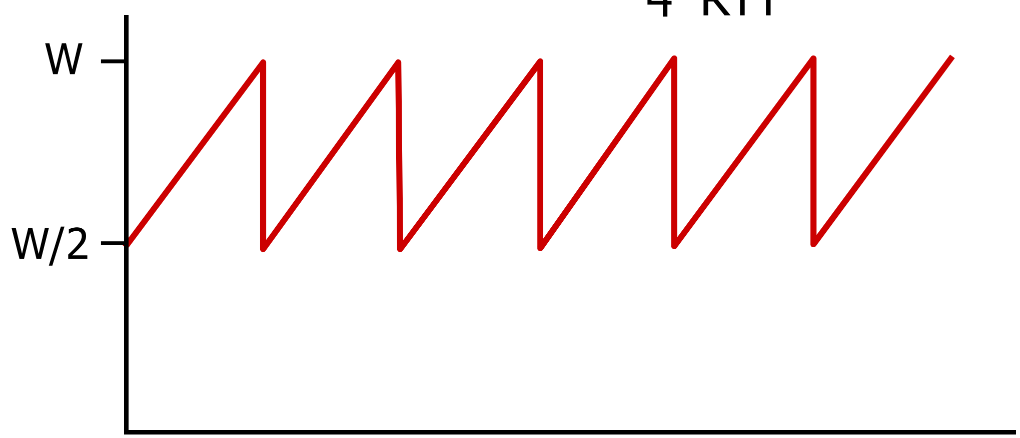
- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

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

TCP throughput

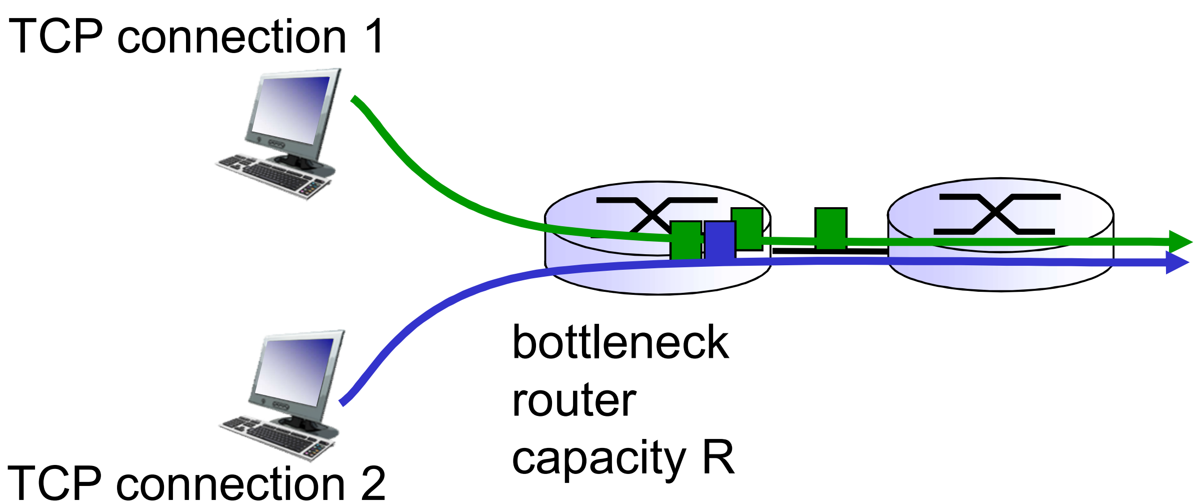
- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. throughput is 3/4W per RTT

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



TCP Fairness

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



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
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets just over R/2

Test your understanding

- Here are some “clever” TCP ideas – what is wrong with them?
- The receiver will only send a very small rwnd that way it can never get too much traffic.
- The sender will only reduce window size very slowly in case of congestion (AIAD).
- We can be very quick if we send another packet if we have only one duplicate ACK (not triple)
- Pause the video write down your answers.

Test your understanding

- The receiver will only send a very small rwnd that way it can never get too much traffic.

  This will be inefficient. The sender can only have a small window (like rdt 3.0)

- The sender will only reduce window size very slowly in case of congestion.

  We may lead to lots of lost packets.

  We may no longer respect fairness.

- We can be very quick if we send another packet if we have only one duplicate ACK (not triple)

  We will sends lots of unnecessary packets

- If you are confused review the previous section.

What have we learned?

- Completed our description of TCP
- Learned TCP mechanisms:
  - Set up of connection SYN – SYNACK – ACK
  - Closing a connection FIN – FINACK
  - Windows – used to control data "in flight" in the network.
  - Flow control – controlled by receiver limits window for connection. Stops receiver getting too much traffic.
  - Congestion control – reacts to network itself being overloaded. Stops network getting too much traffic.

Transport Layer: summary

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

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- Network layer:
  - data plane
  - control plane