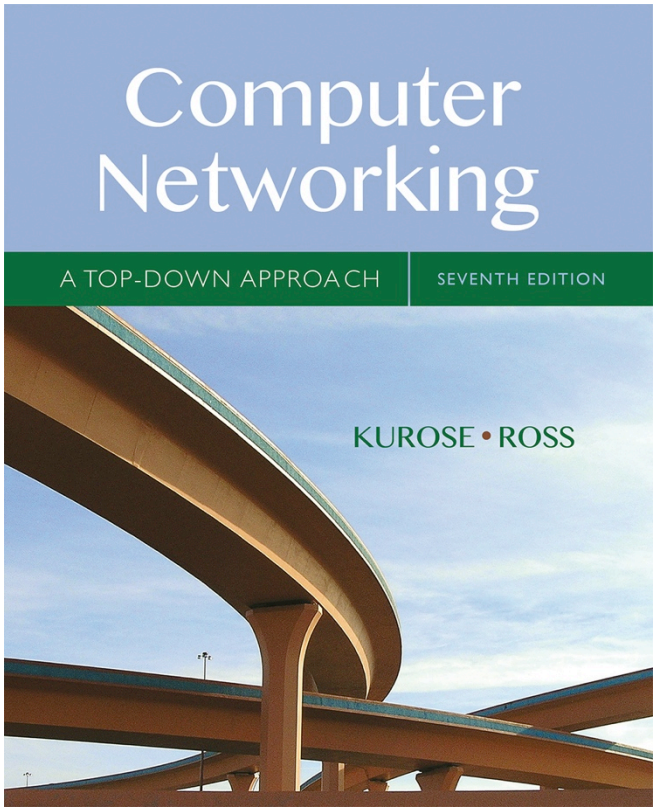
Chapter 3
Transport Layer



Computer Networking: A Top Down Approach

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

A note on the use of these Powerpoint slides:

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

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

Thanks and enjoy! JFK/KWR

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`**

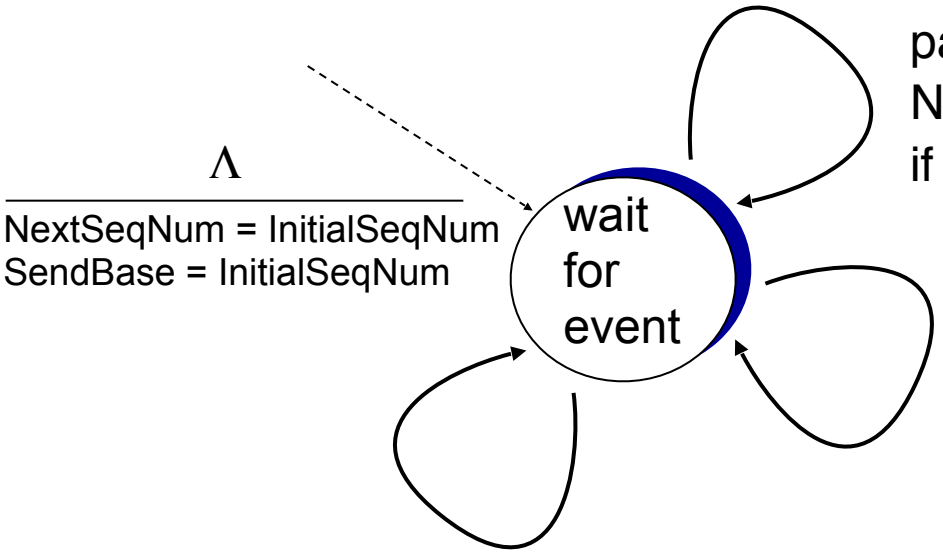
timeout:

- 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 sender (simplified)



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

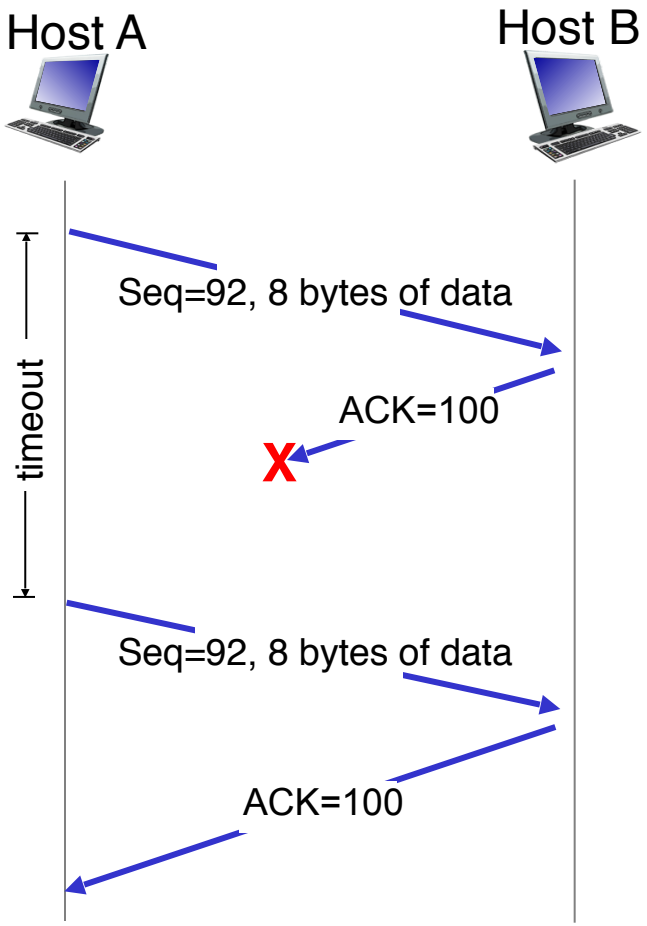
timeout

retransmit not-yet-acked segment with smallest seq. #
start timer

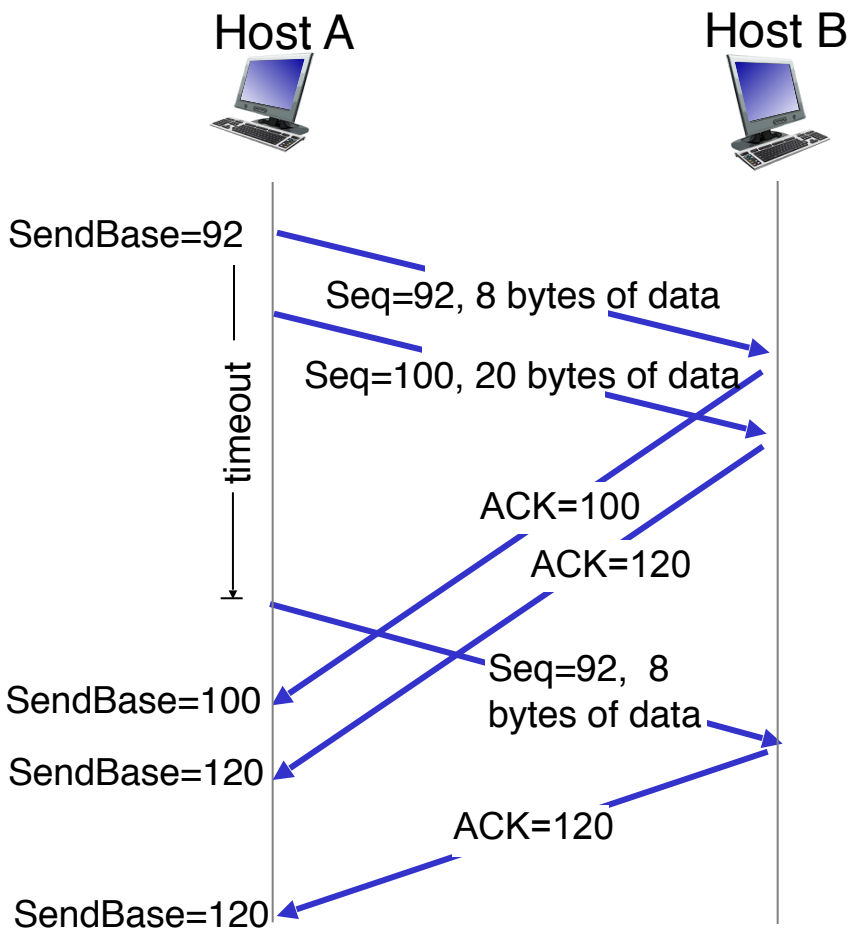
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
    else stop timer
    }
```

TCP: retransmission scenarios



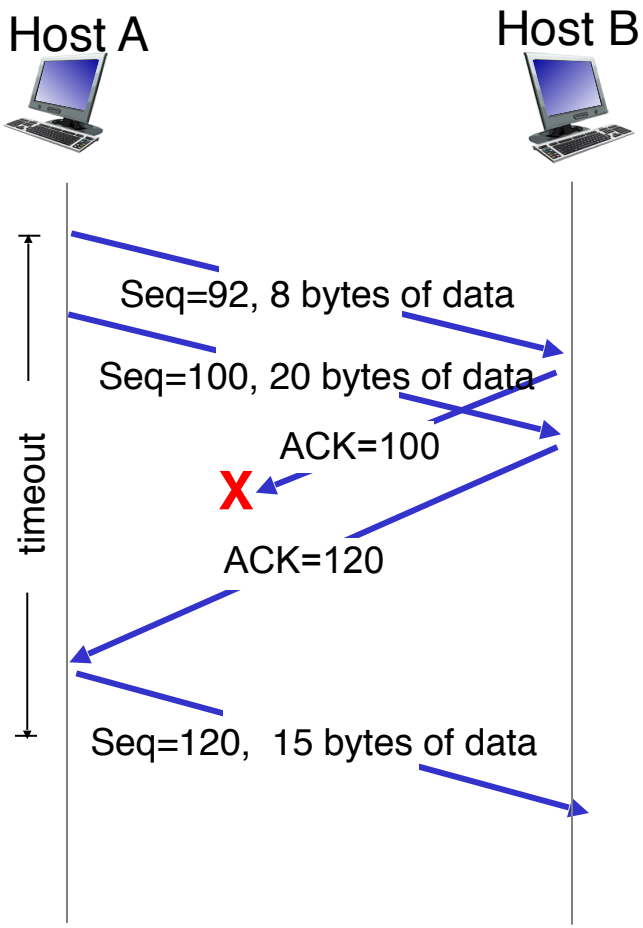
lost ACK scenario



premature timeout

Every time TCP transmits it normally doubles the interval time

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

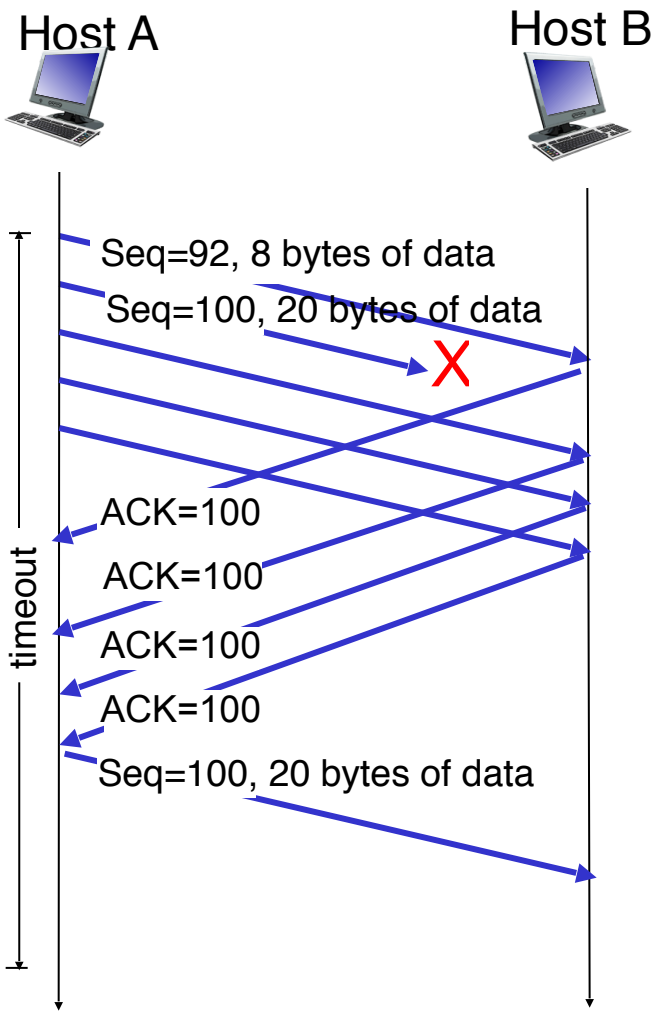
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

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

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

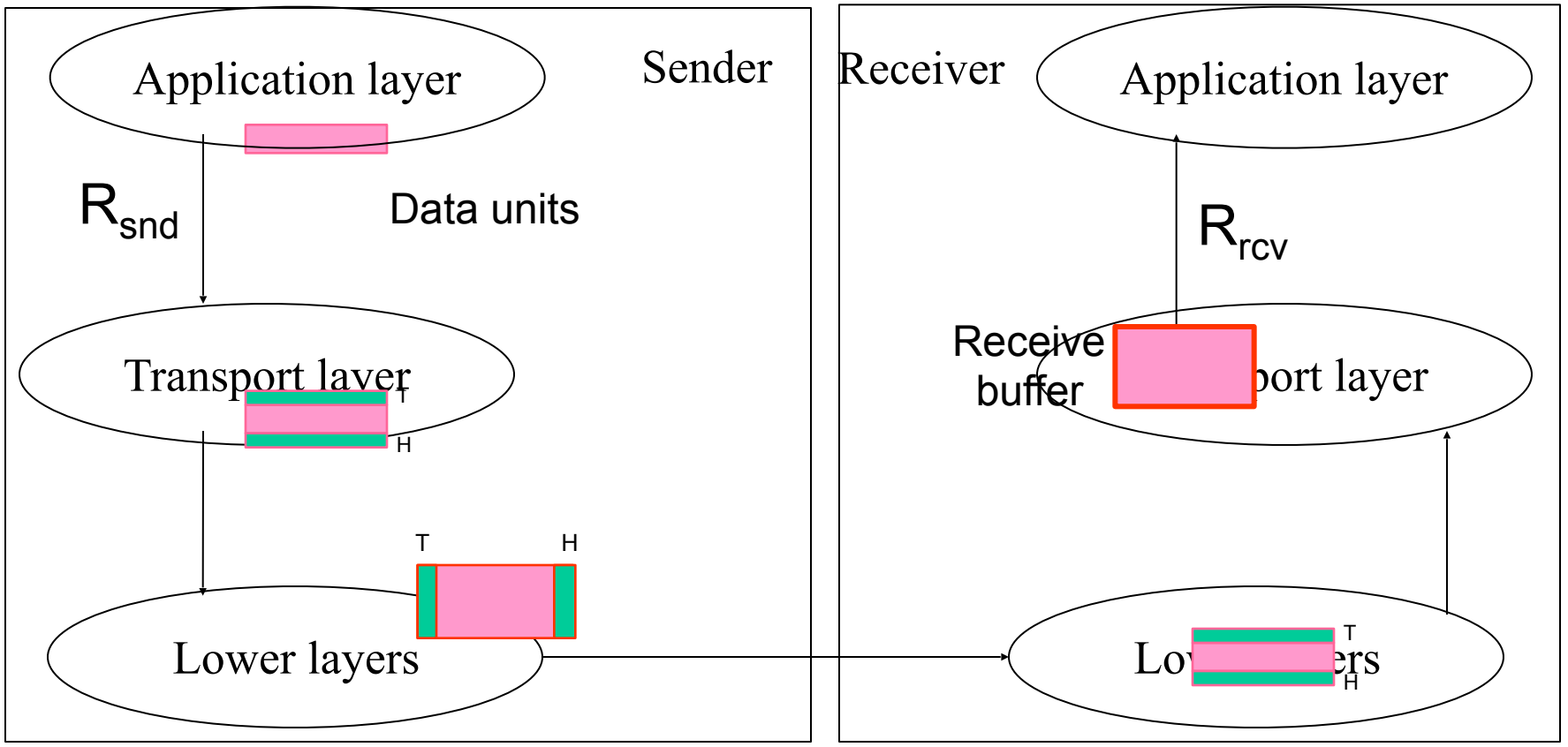
3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 TCP congestion control

Flow control problem

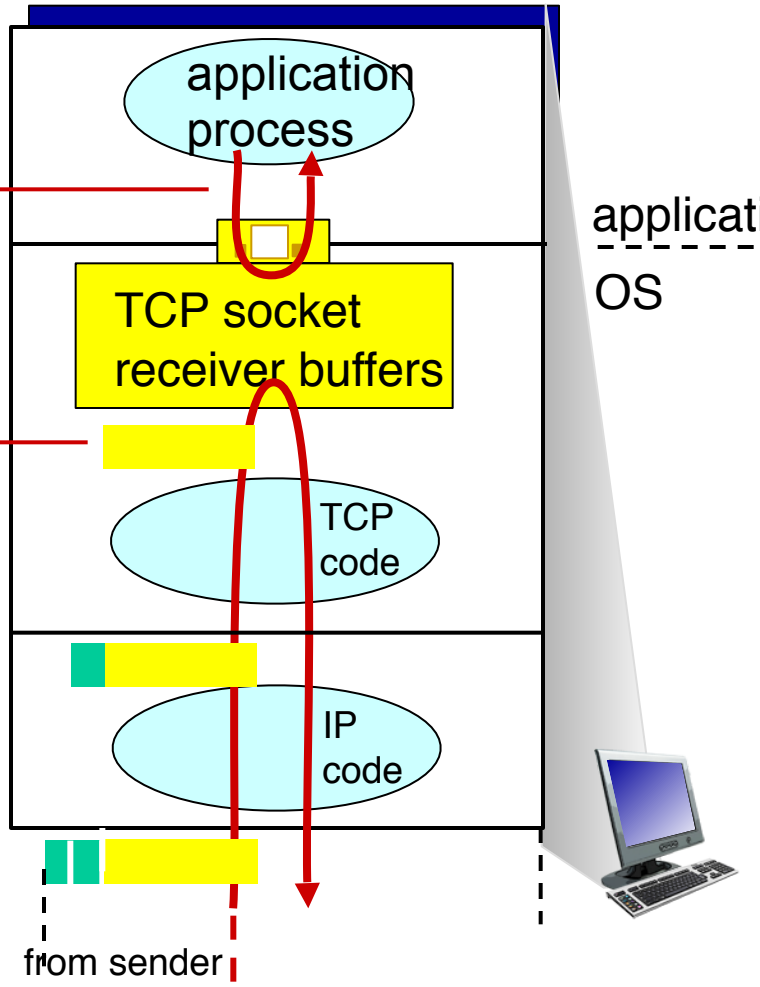


- The flow control problem arises if the application layer at the receiver does not deplete the buffer at the same rate at which data is being passed to the transport layer at the sender ($R_{snd} > R_{rcv}$)
- T: Trailer; H: Header; Each frame has a header, payload (also called data) and trailer

TCP flow control

application may remove data from TCP socket buffers ….

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



receiver protocol stack

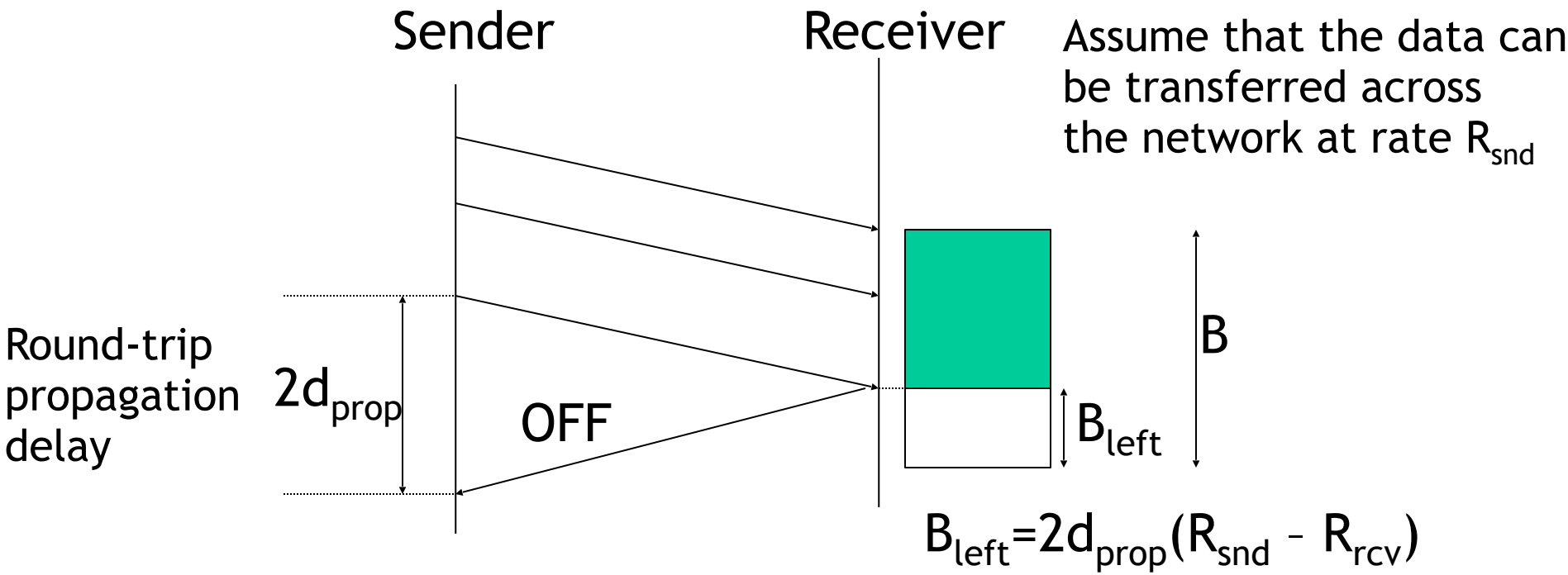
flow control

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

Stop-and-wait flow control

- Sender has to stop after sending each frame and wait for an ACK
- ACK is sent back from the receiver after the higher layer depletes the receive buffer holding the single frame's payload
- Therefore, the sender cannot overrun the receive buffer

ON /OFF flow control



When should the OFF signal be se

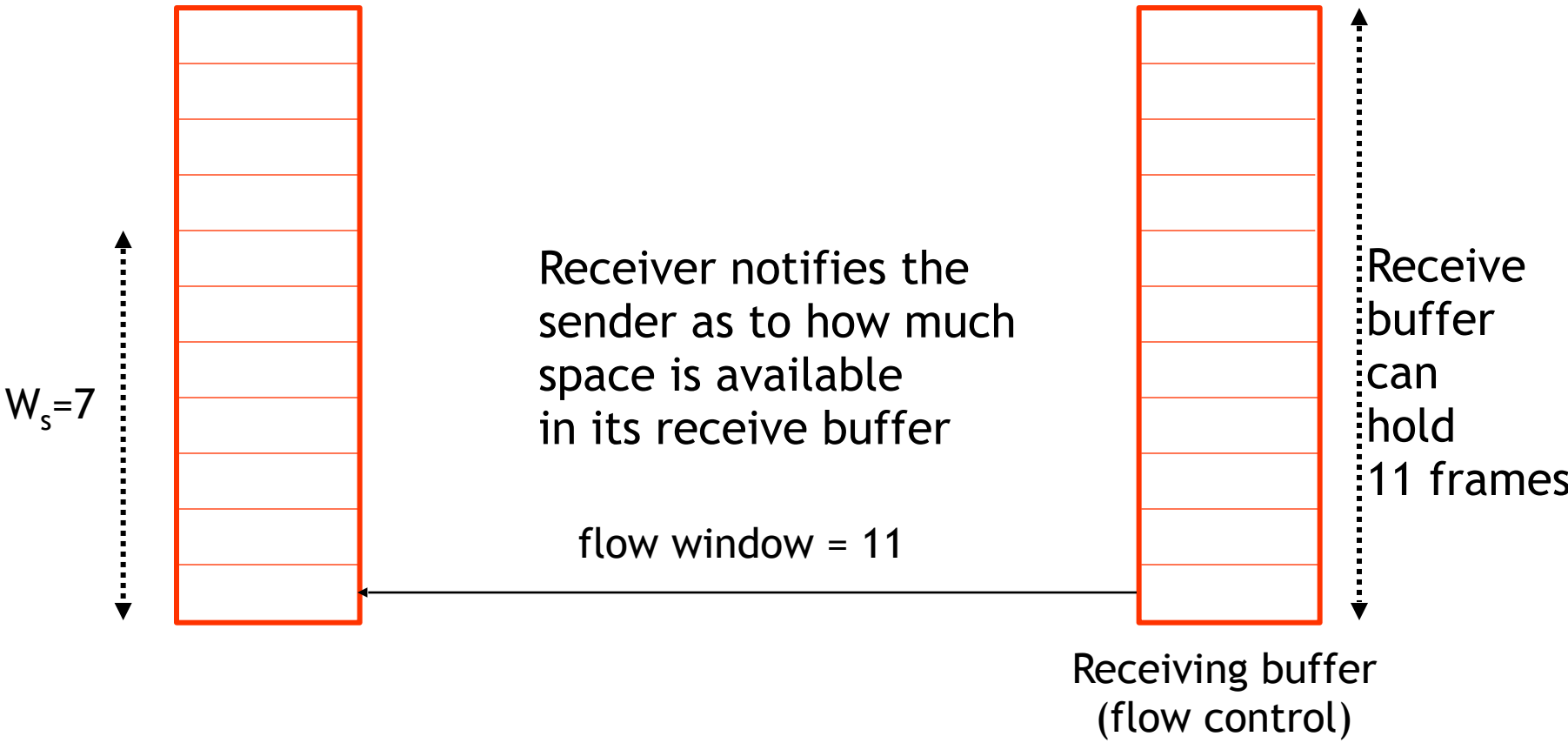
R_{snd}: rate at which the higher layer passes frames to the lower layer at the sender

R_{rcv}: rate at which the higher layer accepts frames from the lower layer at the receiver

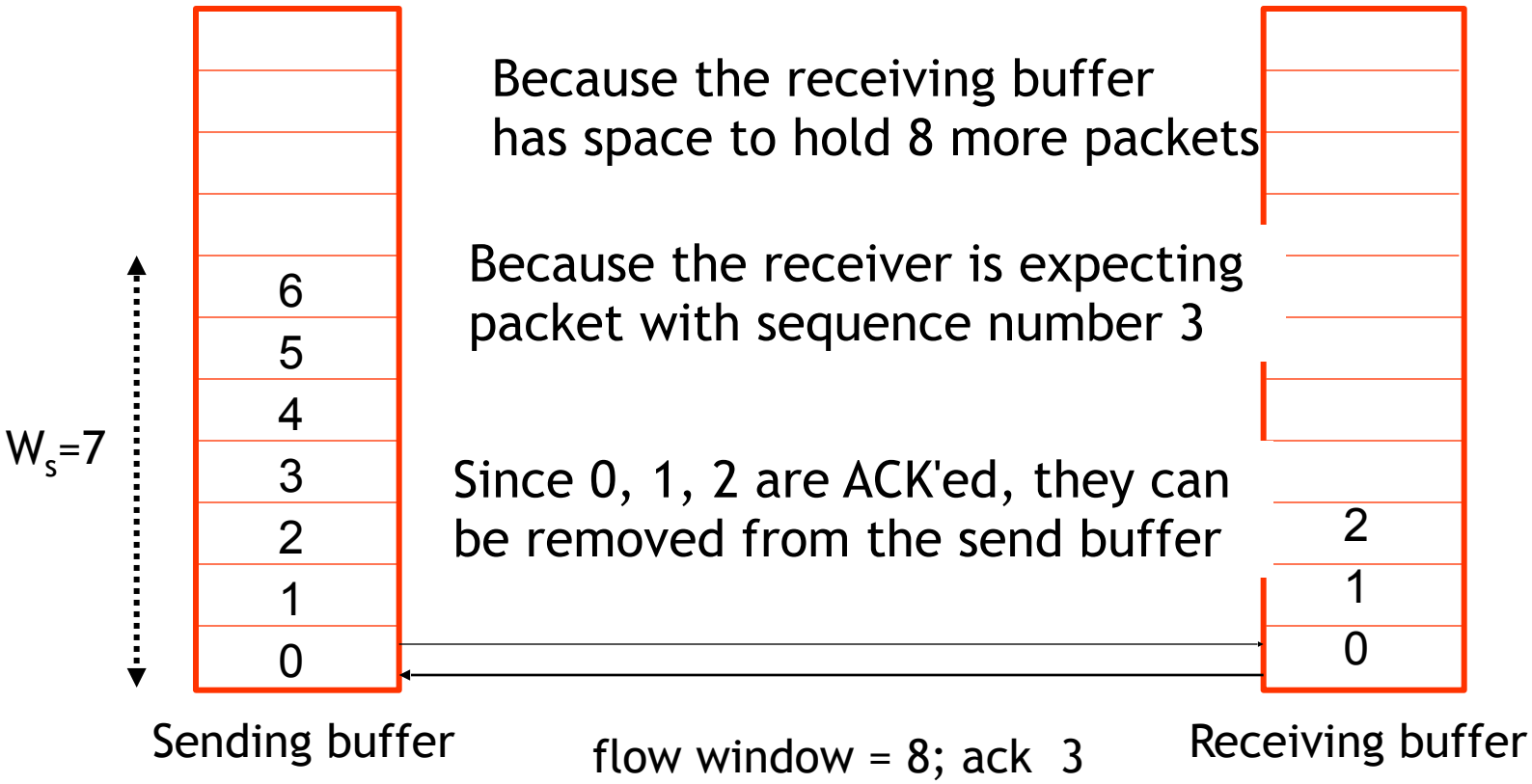
Sliding Window Flow Control

- **Sliding Window Flow Control**
  - Receiver has a receive buffer
  - Receiver sends reports of available space in this buffer to the sender
    - This is called flow window or advertised window
  - Sender uses this number to determine the maximum number of frames it can have outstanding (i.e., unacknowledged)

Sliding window FC illustrated



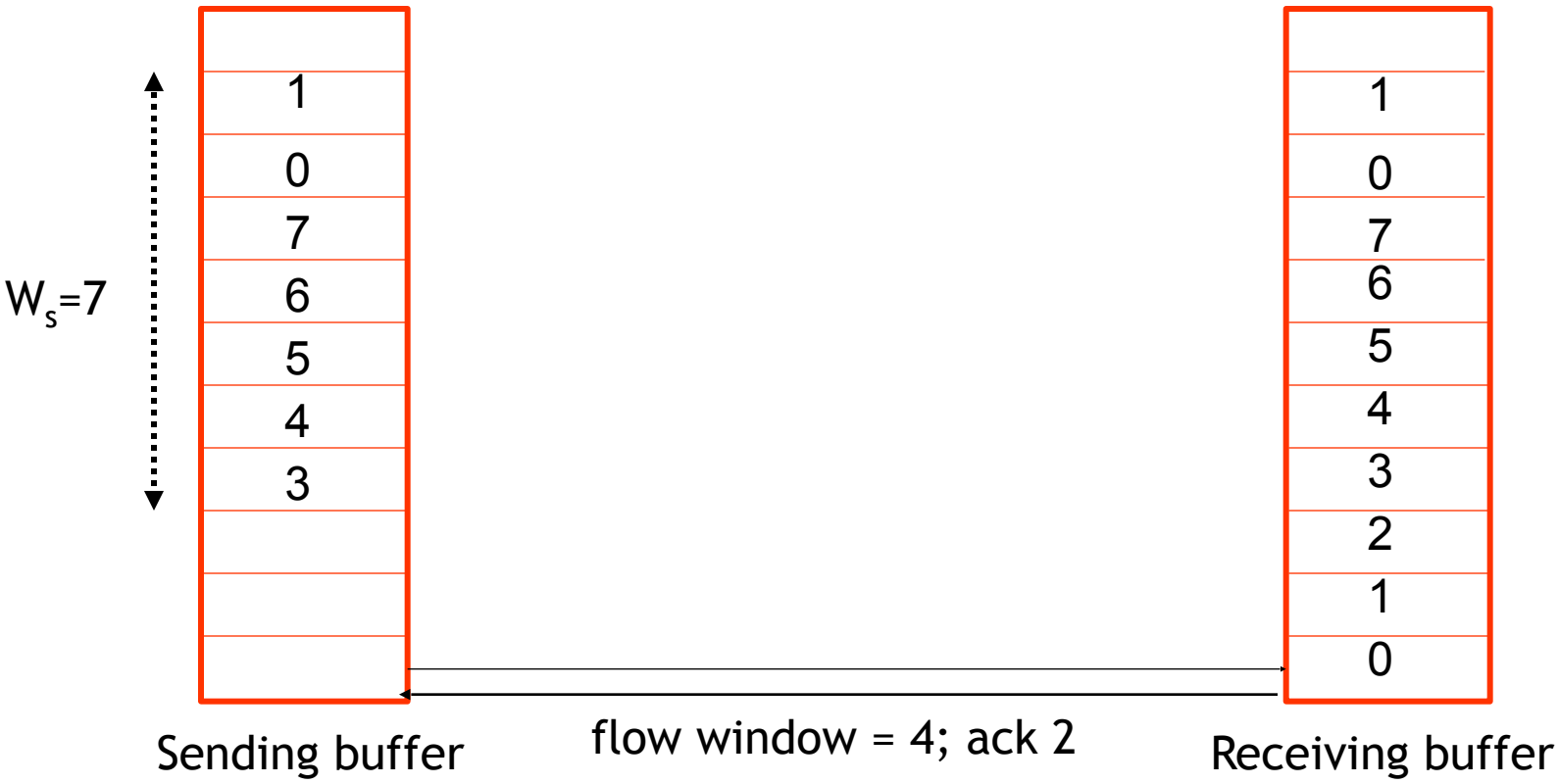
Sliding window FC illustrated



Flow window

- Flow window is the:
  - Left-over space in the receiver's buffer
  - In previous slide, flow window of 8 is a report of how much space is left in the receiver's buffer (enough to hold 8 frames)
- Flow window is also called "advertised window" or "receiver's window"

Sliding window FC illustrated



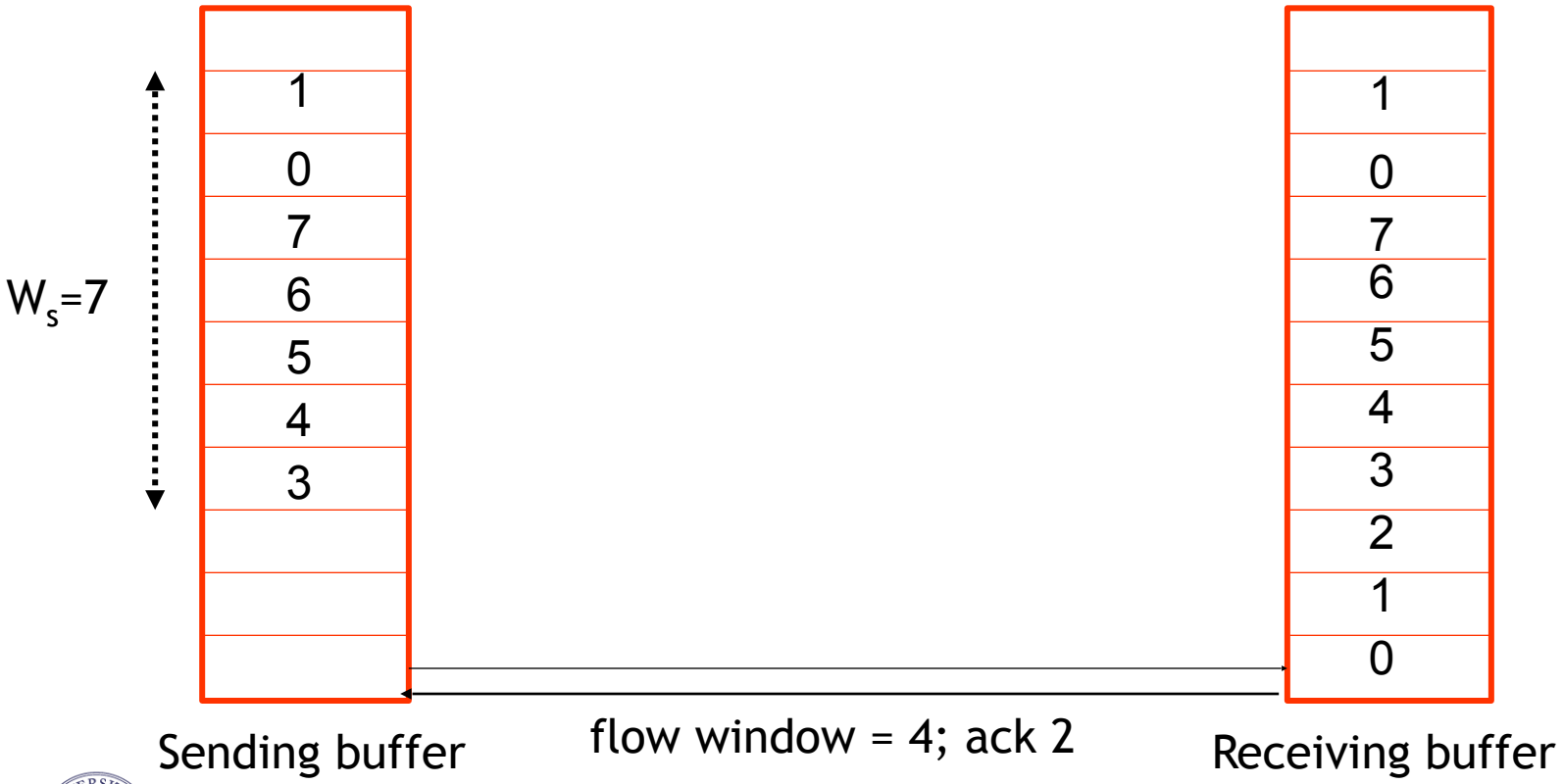
Watch with animation; frames 0, 1, 2 are read out of the receive buffer by the higher-layer, which is why the flow window indication is 4

At any instant in time, what is the maximum number of frames that the sender is permitted to send?

- Maximum number of outstanding frames at any instant in time is a minimum of sending window, W_s, and the flow window indicated by the receiver
- **Three steps:**
    1. **Find X = min (W_s, flow window)**
    2. **Find Y = number of outstanding frames (already sent but unacknowledged)**
    3. **Maximum number of frames that the sender is permitted to send = X - Y**
    - Result: if X-Y frames are sent, then the number of outstanding frames will become X (which is the max. limit for no. of outstanding frames)
-

Test your understanding.

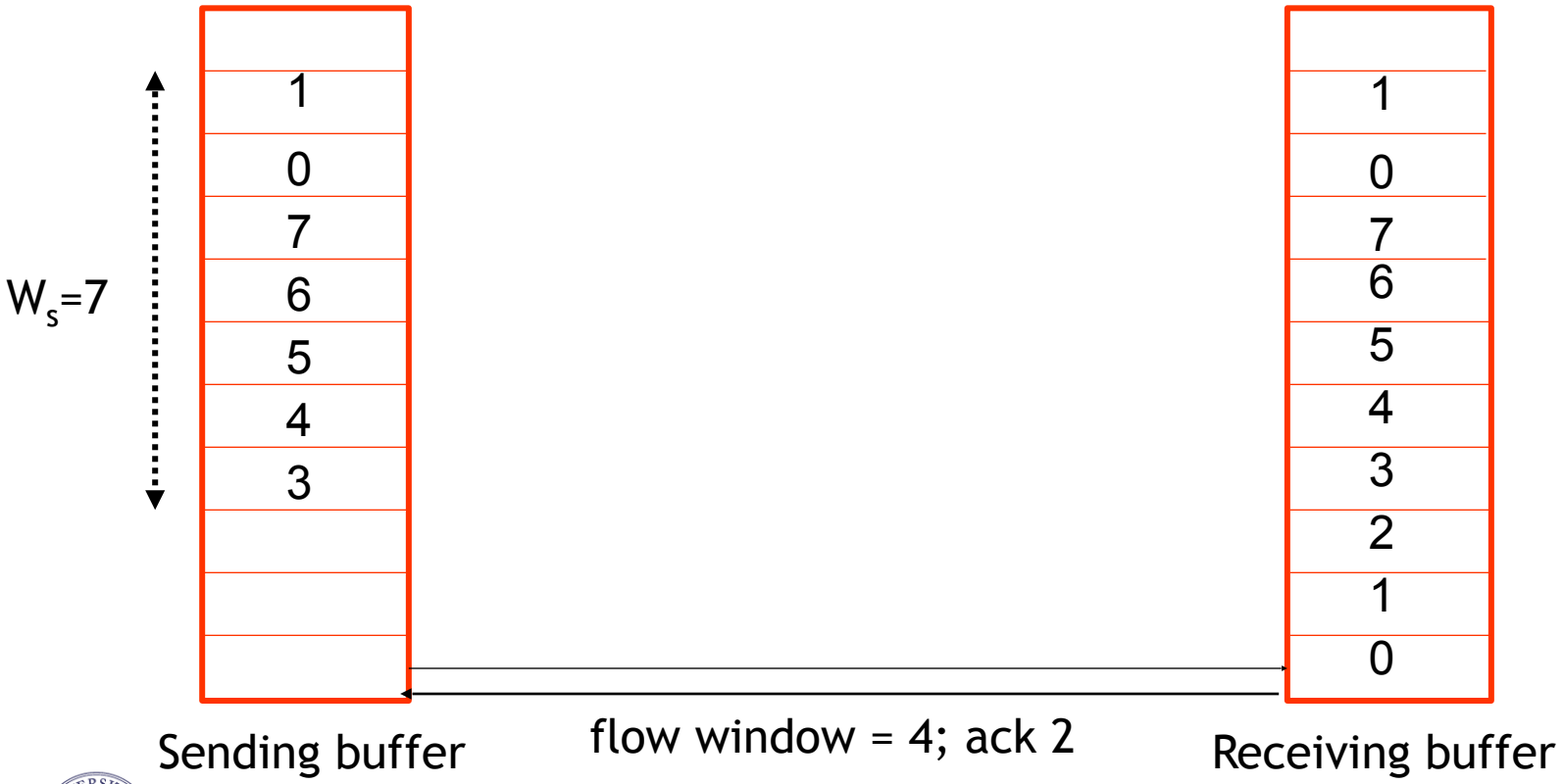
- After the sender receives the indication that the flow window = 4 and ACK = 2 how many frames can the sender send?
-



Three steps:

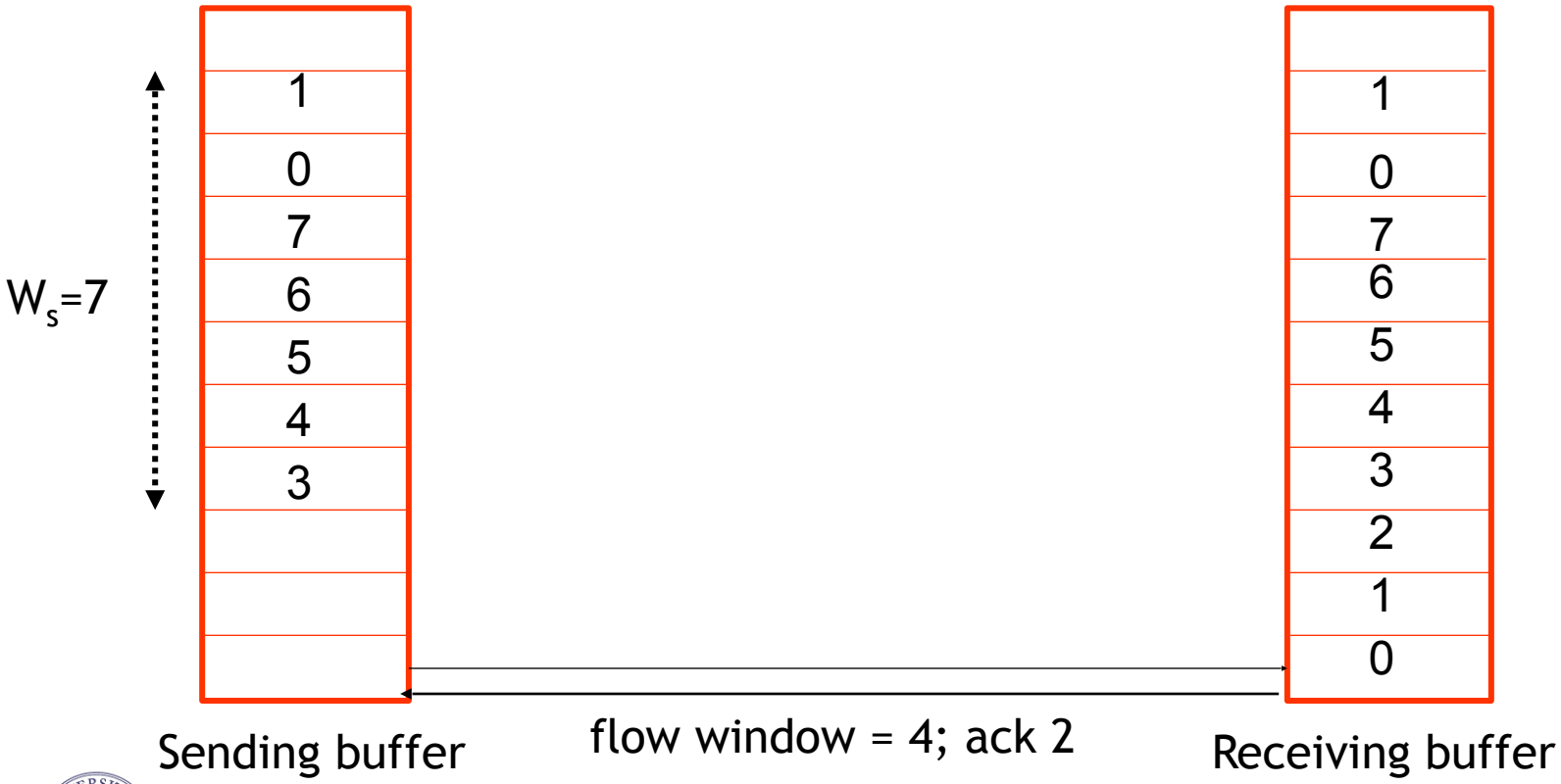
1. **Find X = min (W_s, flow window)**
2. **Find Y = number of outstanding frames (already sent but unacknowledged)**
3. **Maximum number of frames that the sender is permitted to send = X - Y**

- After the sender receives the indication that the flow window = 4 and ACK = 2 how many frames can the sender send?
-



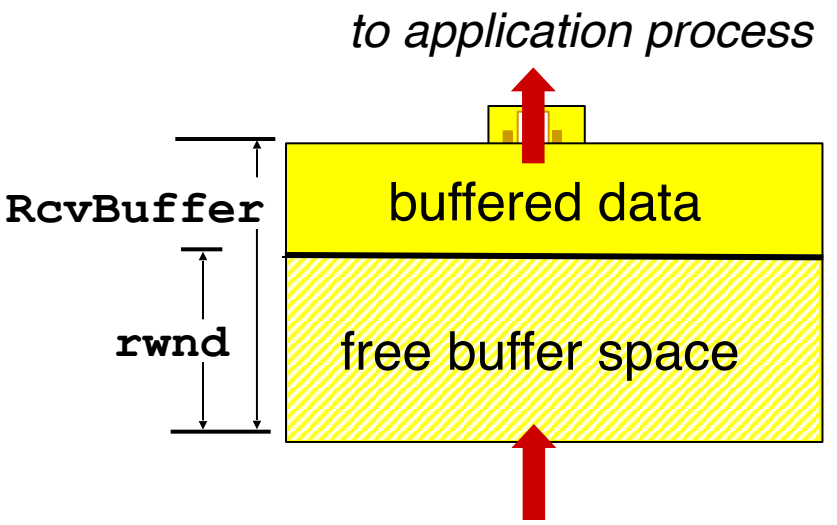
Test your understanding.

- What are the sequence numbers of the frames that it will send (assume that the application layer keeps passing frames down to the transport layer at the sender)?



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

Chapter 3 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

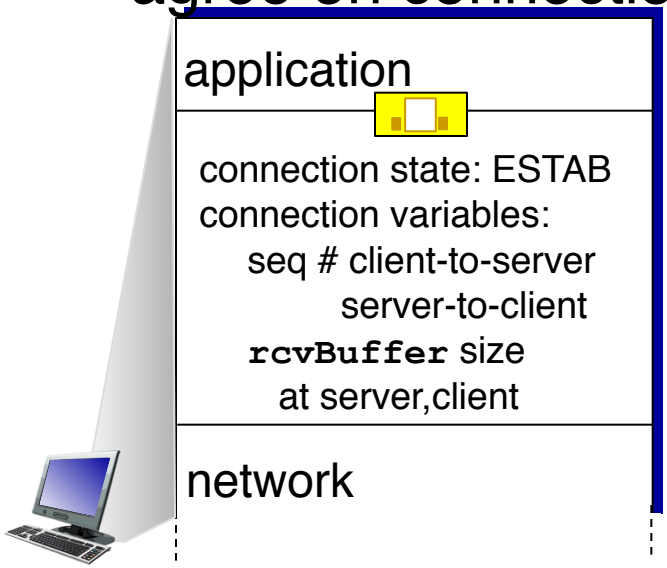
3.6 principles of congestion control

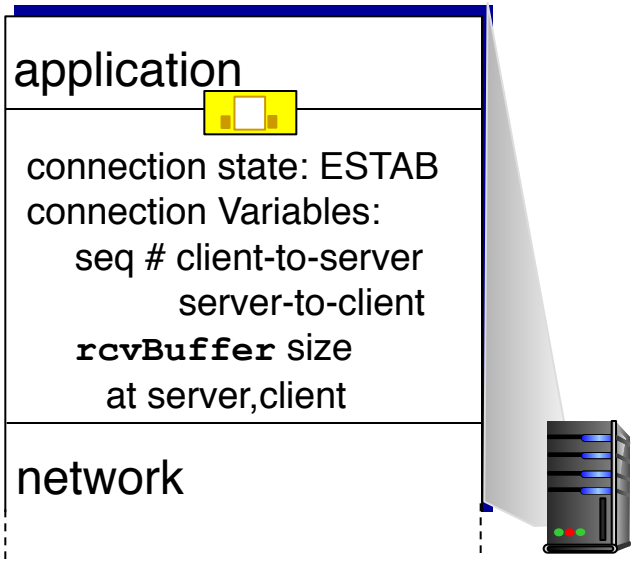
3.7 TCP congestion control

Connection Management

before exchanging data, sender/receiver "handshake":

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



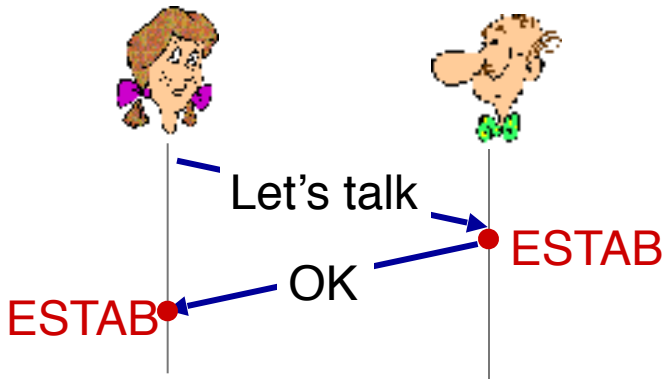


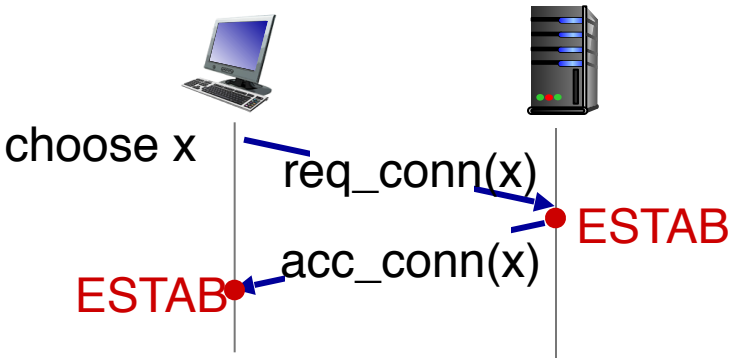

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to establish a connection

2-way handshake:



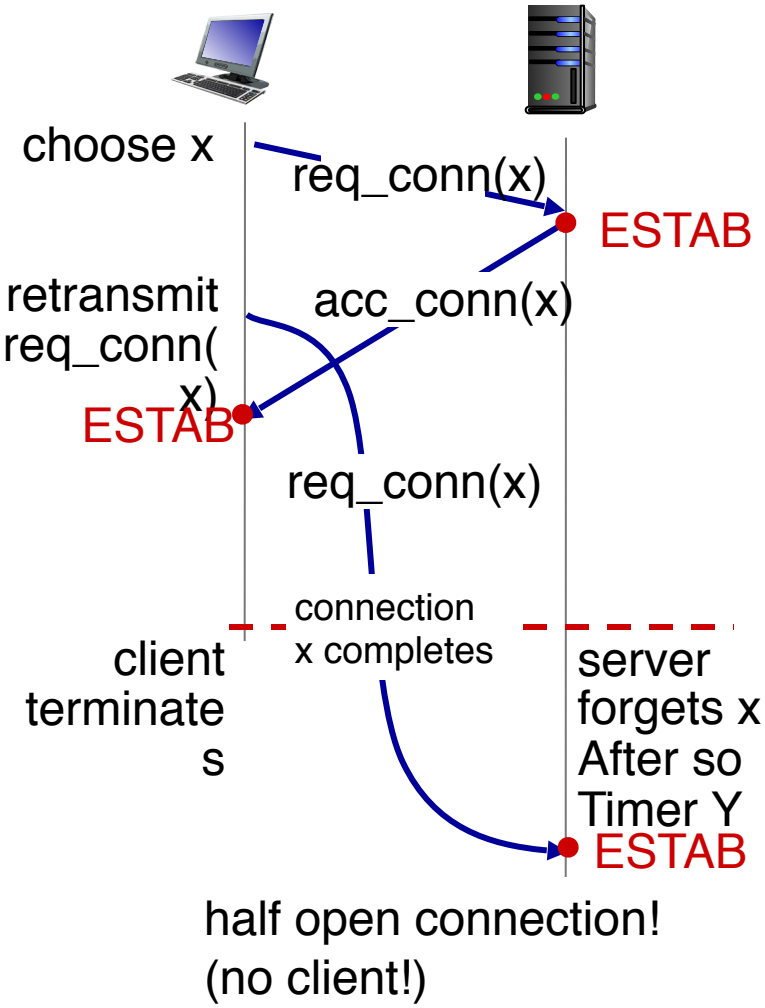


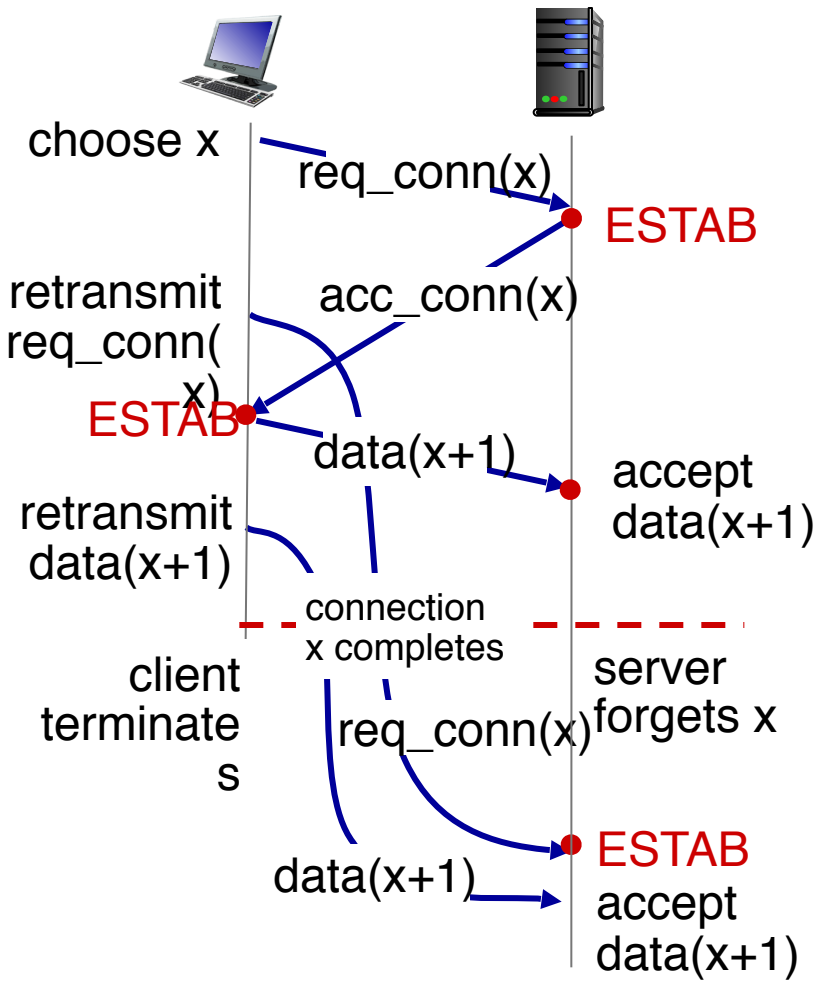
Q: will 2-way handshake always work in network?

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

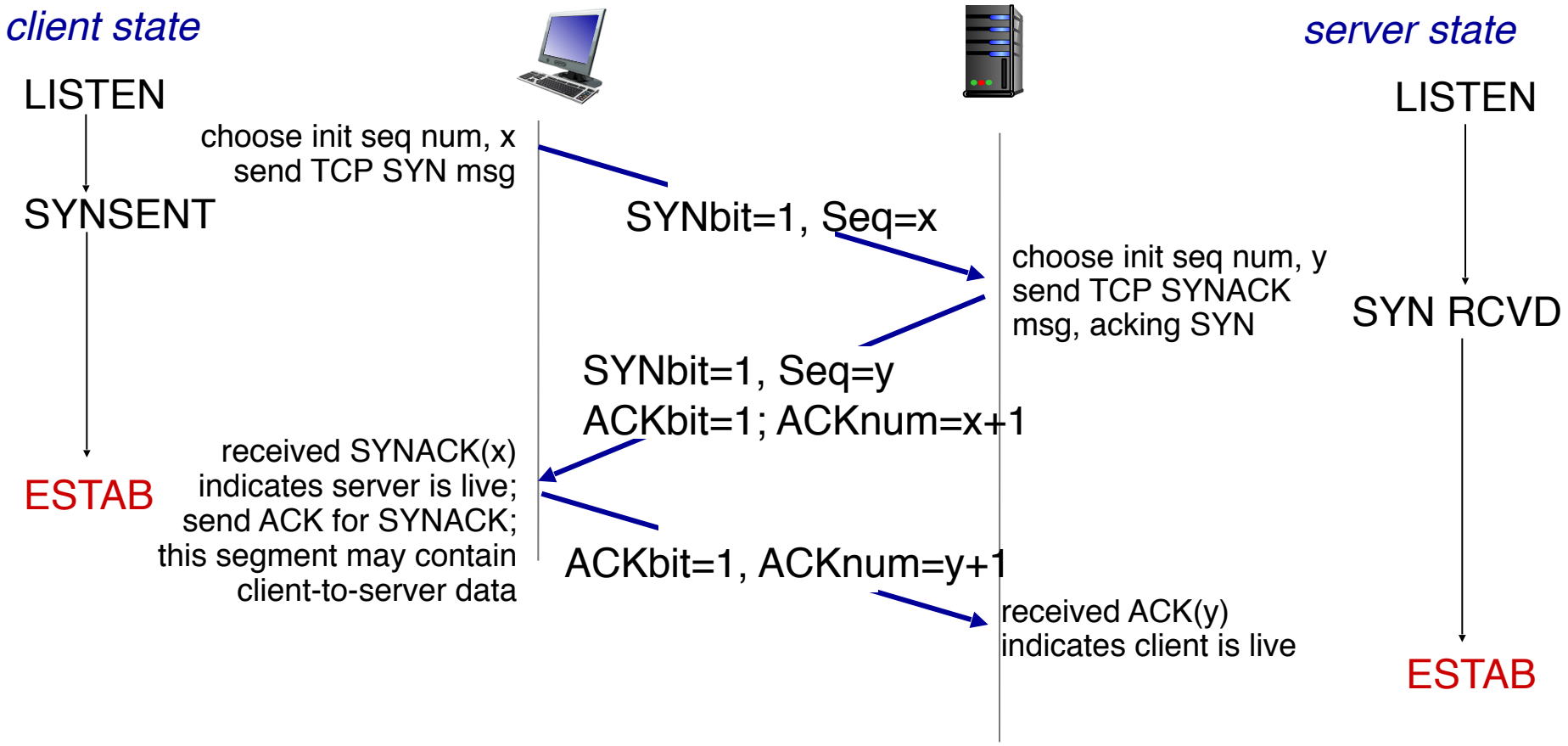
Agreeing to establish a connection

2-way handshake failure scenarios:

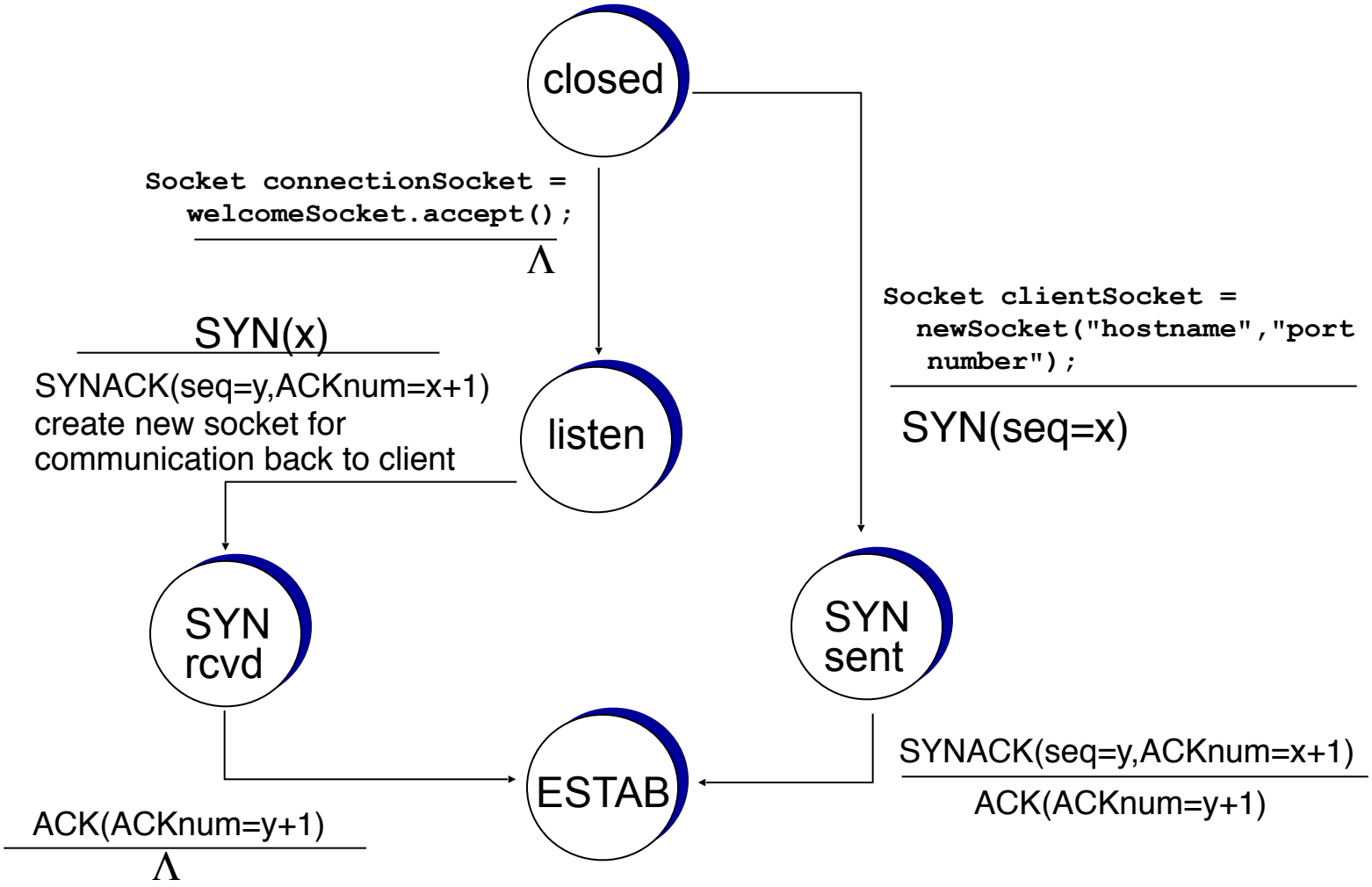




TCP 3-way handshake



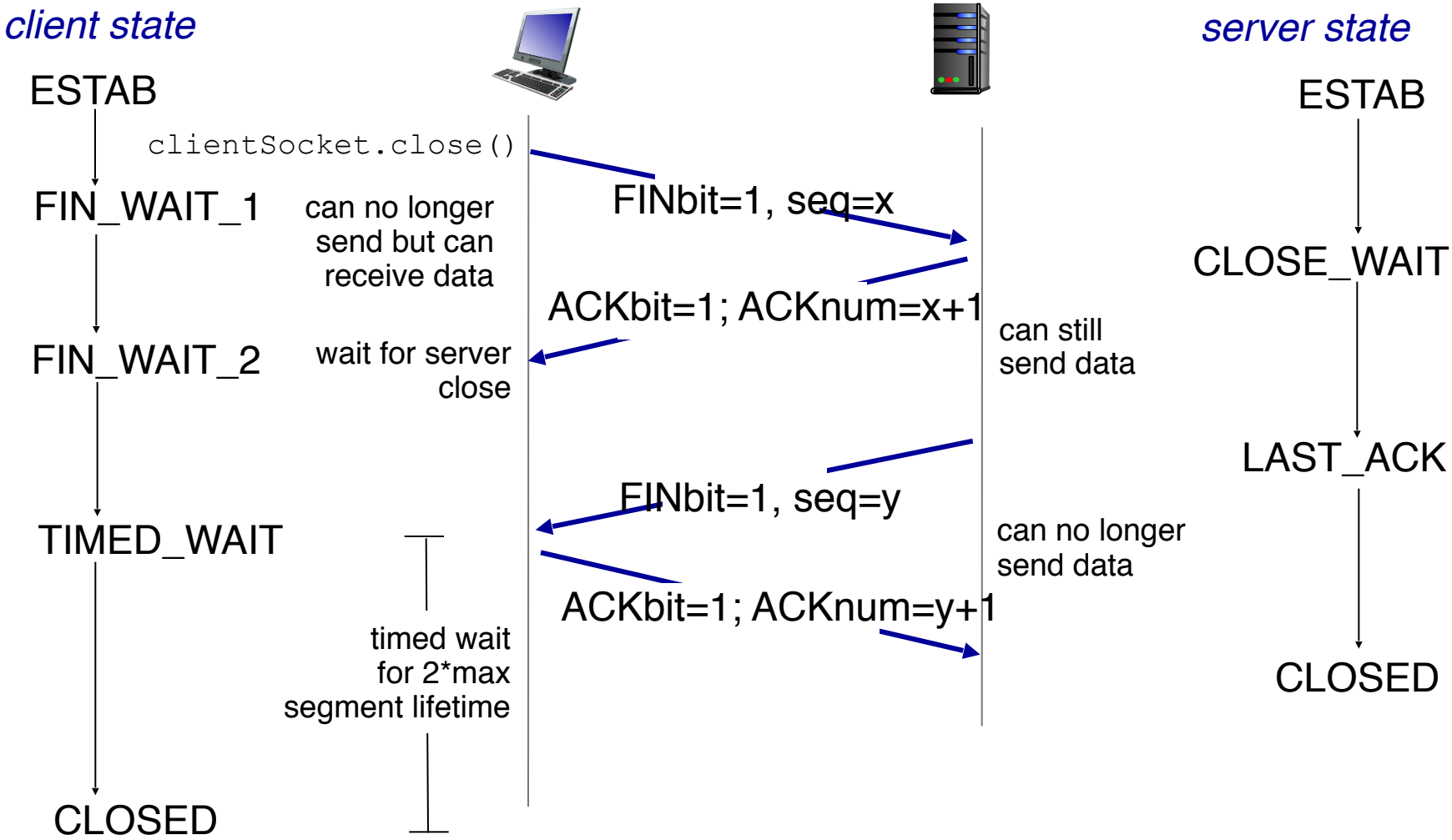
TCP 3-way handshake: FSM



TCP: closing a connection

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

TCP: closing a connection



Chapter 3 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!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission