Internetværk og Web-programmering
TCP and Socket Programming

Lecture 12
Michele Albano

Distributed, Embedded, Intelligent Systems

DEIS

Agenda

3.5 connection-oriented transport: TCP

- **segment structure**
- reliable data transfer
- flow control

3.6 principles of congestion control

3.7 TCP congestion control

2.7 Socket programming / [DF] **16.9** Javascript sockets

Transmission Control Protocol:

- point-to-point:
  - one sender, one receiver
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- reliable, in-order byte stream:
  - no “message boundaries”

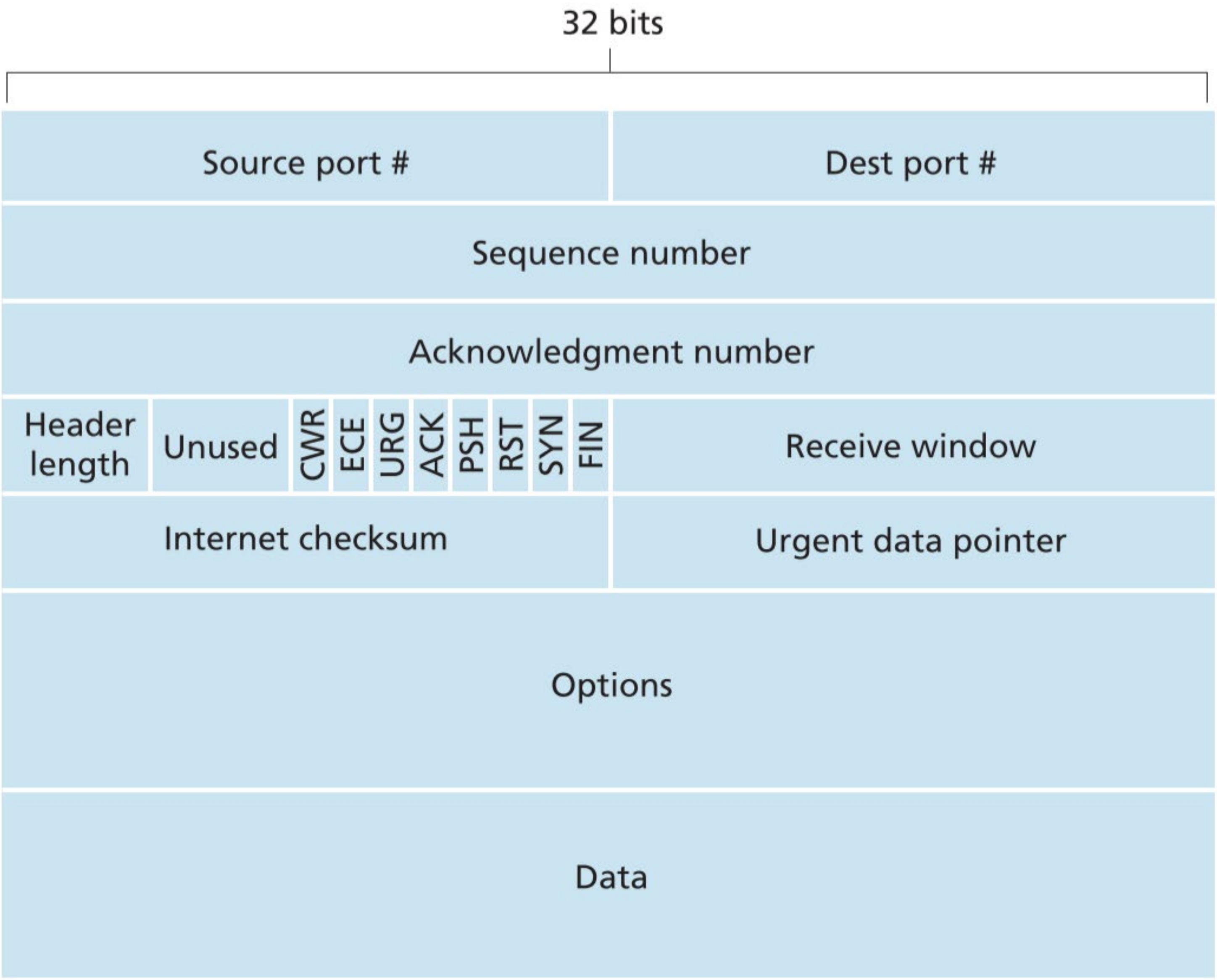
- connection-oriented:
  - handshaking (exchange of control msgs) initializes sender, receiver state before data exchange
- pipelined:
  - TCP congestion and flow control set window size
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure 1/2

- Src/dst ports
  - TCP and UDP have different namespaces
- Seq/ack numbers to the byte

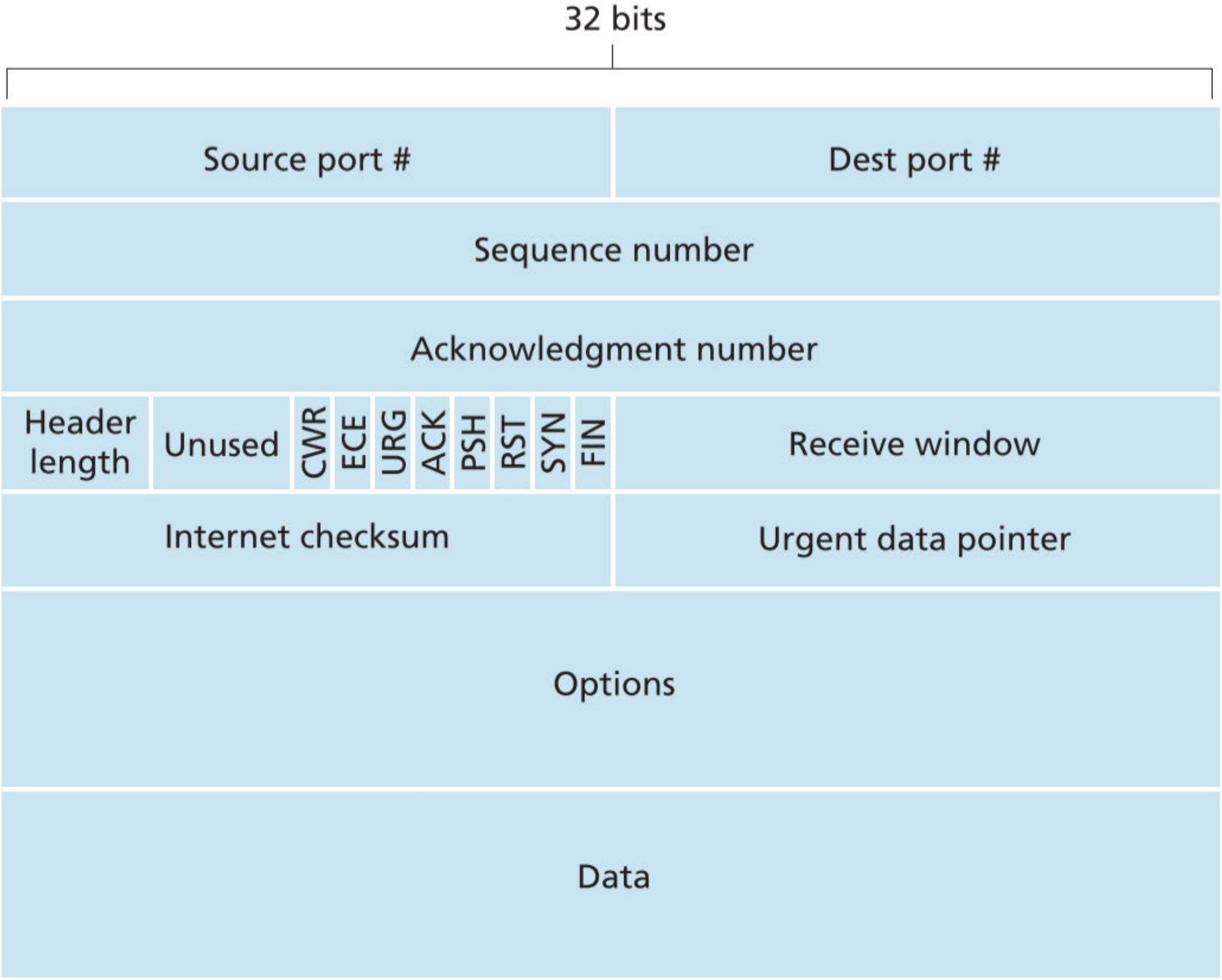
Flags:

- CWR and ECE – explicit congestion notification
- URG and PSH – urgent data
- SYN and FIN seen in lecture 11
- ACK is 1 if the ACK number is valid
- RST to kill the connection



TCP segment structure 2/2

- Header length – needed since “Options” has variable length
- Receive window: for flow control
- Internet checksum: 16-bit checksum for TCP header, payload, and some data from IP (routing layer)
- Urgent data pointer: used with the URG flag
- Options: for example to set the maximum size of the segments, or timestamp



32 bits

Source port #

Dest port #

Sequence number

Acknowledgment number

Header length

Unused

CWR

ECE

URG

ACK

PSH

RST

SYN

FIN

Receive window

Internet checksum

Urgent data pointer

Options

Data

TCP Sequence Numbers, ACKs 1/2

sequence numbers:

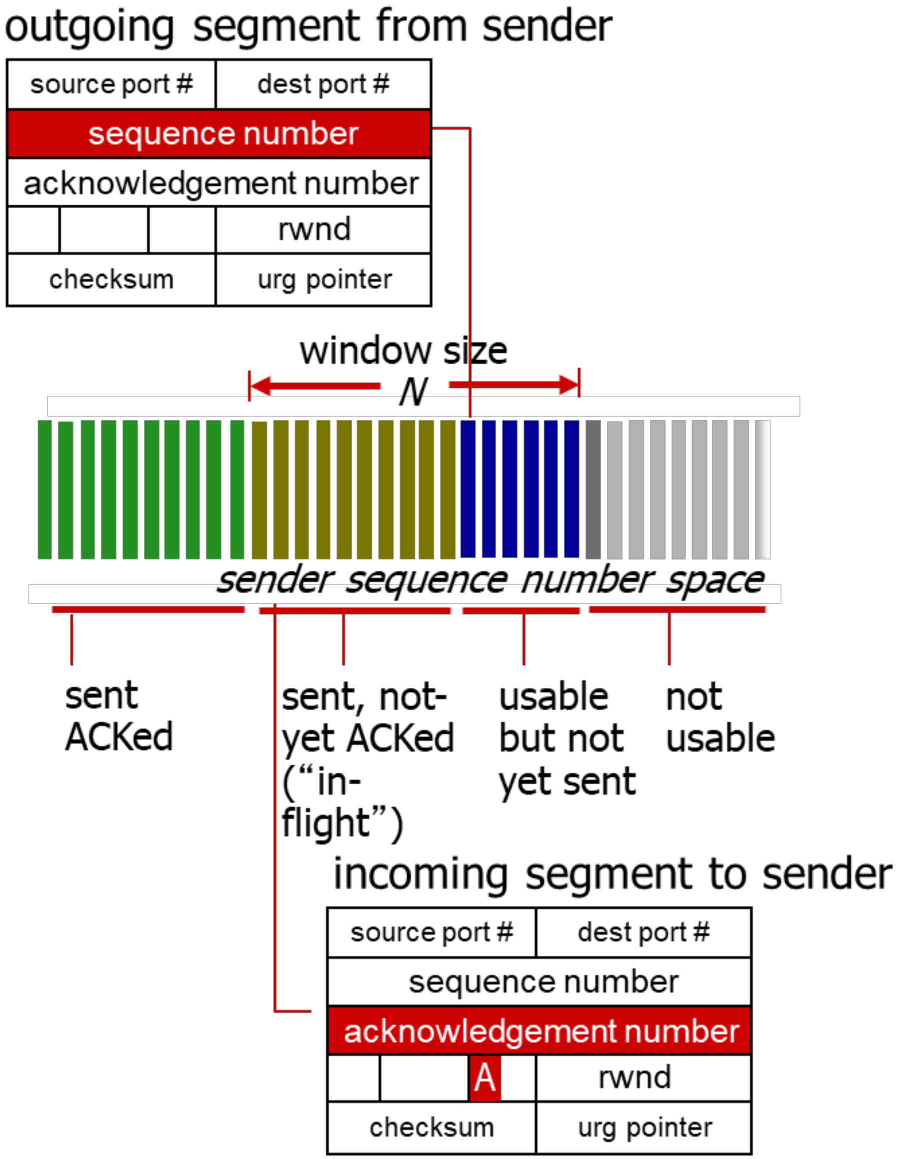
- byte stream "number" of first byte in segment's data

acknowledgements:

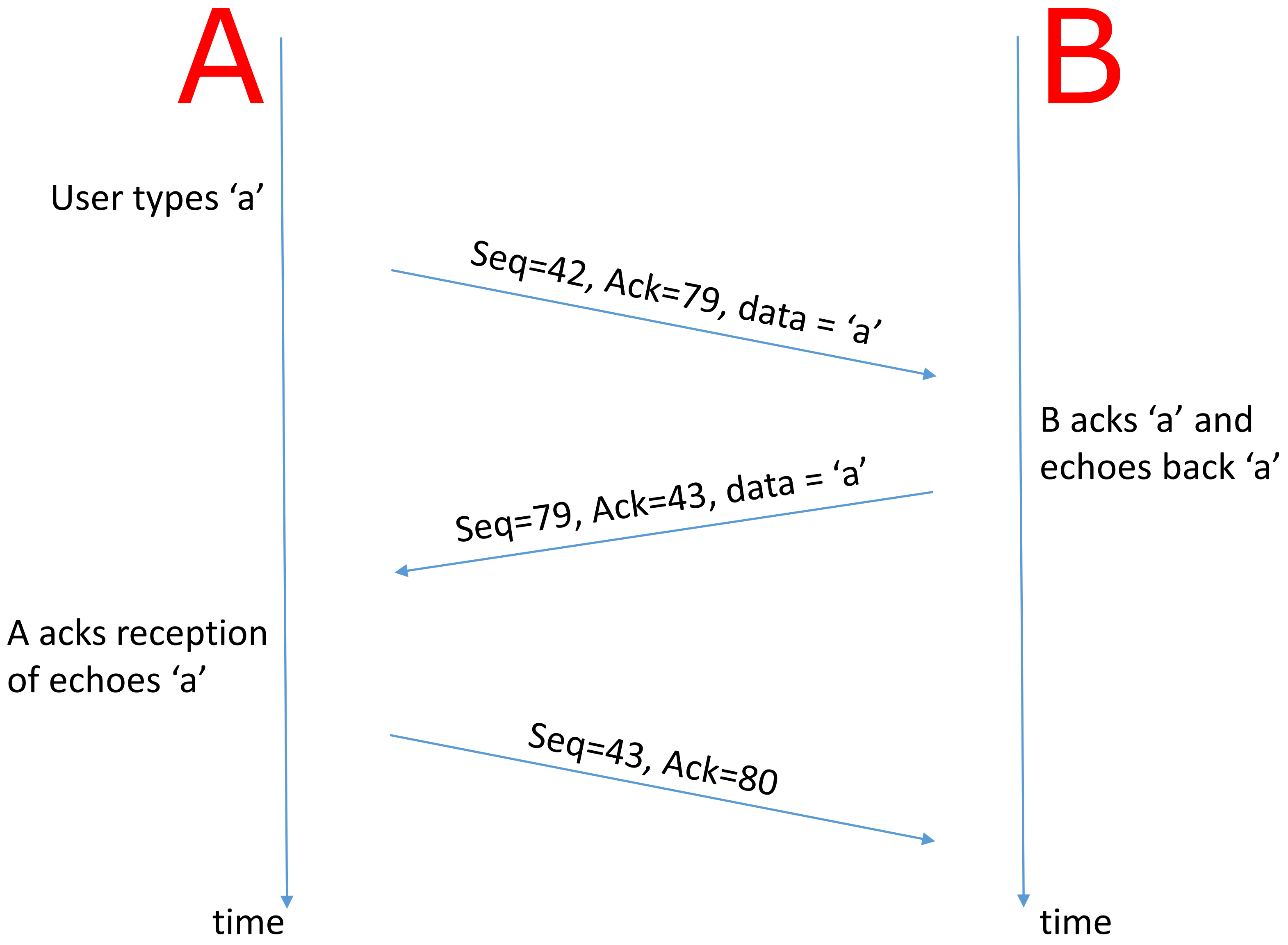
- seq # of next byte expected from other side
- cumulative ACK

How receiver handles out-of-order segments?

- TCP specs doesn't say
- implementation dependent



TCP Sequence Numbers, ACKs 2/2



A

B

User types ‘a’

Seq=42, Ack=79, data = ‘a’

B acks ‘a’ and
echoes back ‘a’

Seq=79, Ack=43, data = ‘a’

A acks reception
of echoes ‘a’

Seq=43, Ack=80

time

time

Agenda

3.5 connection-oriented transport: TCP

- segment structure
- **reliable data transfer**
- flow control

3.6 principles of congestion control

3.7 TCP congestion control

2.7 Socket programming / [DF] **16.9** Javascript sockets

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

Timeout: how long?

- longer than RTT (but RTT varies)
- **too short***:* premature timeout, unnecessary retransmissions
- **too long***:* slow reaction to segment loss

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

TCP Sender Events:

data received from app layer:

- 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`**

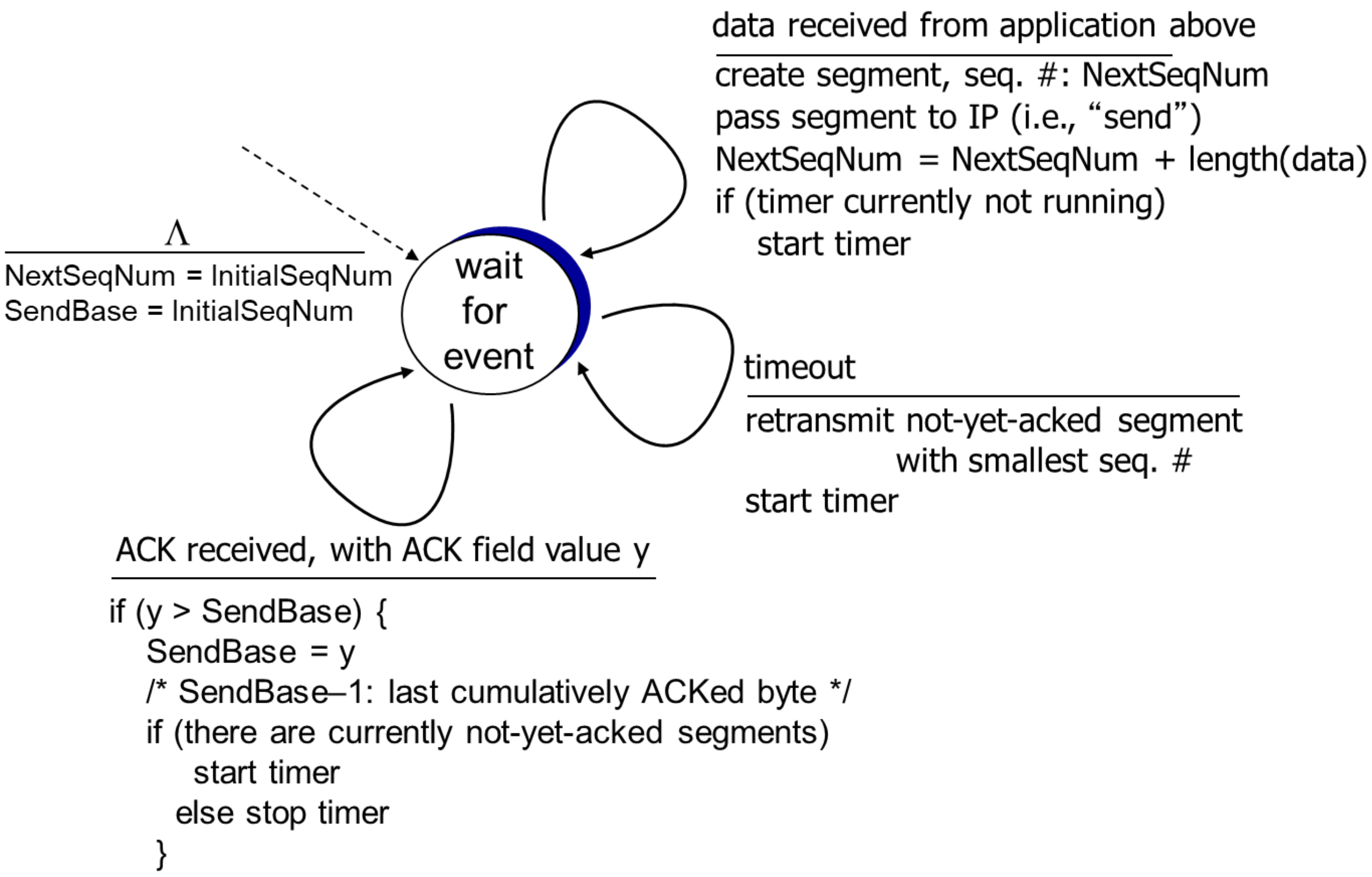
timeout:

- retransmit segment that caused timeout
- restart timer

ack received:

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

TCP Sender (Simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

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
```

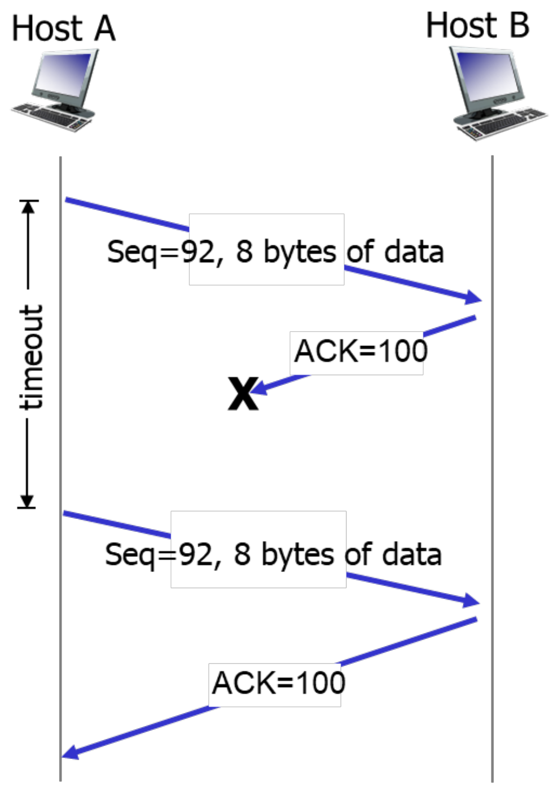
timeout

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

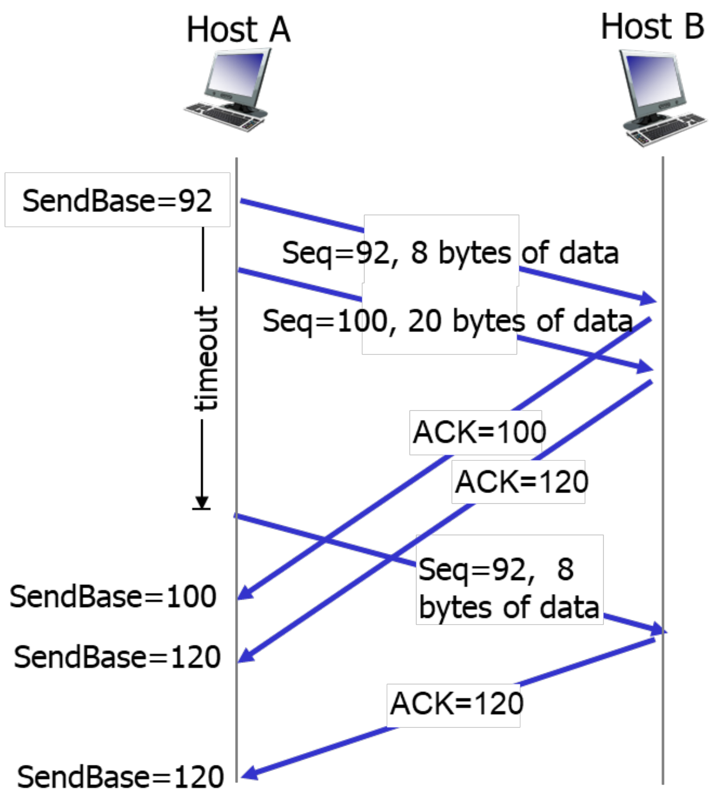
ACK received, with ACK field value y

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

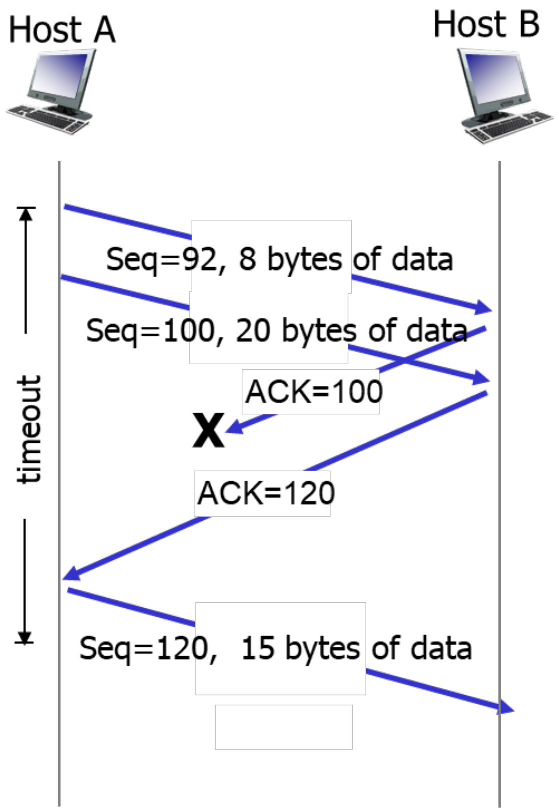
TCP Retransmission Scenarios



lost ACK scenario



premature timeout



cumulative ACK

TCP ACK Generation [RFC 1122, RFC2581]

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 Sequence # . Gap detected	immediately send **duplicate ACK,** indicating Sequence # 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

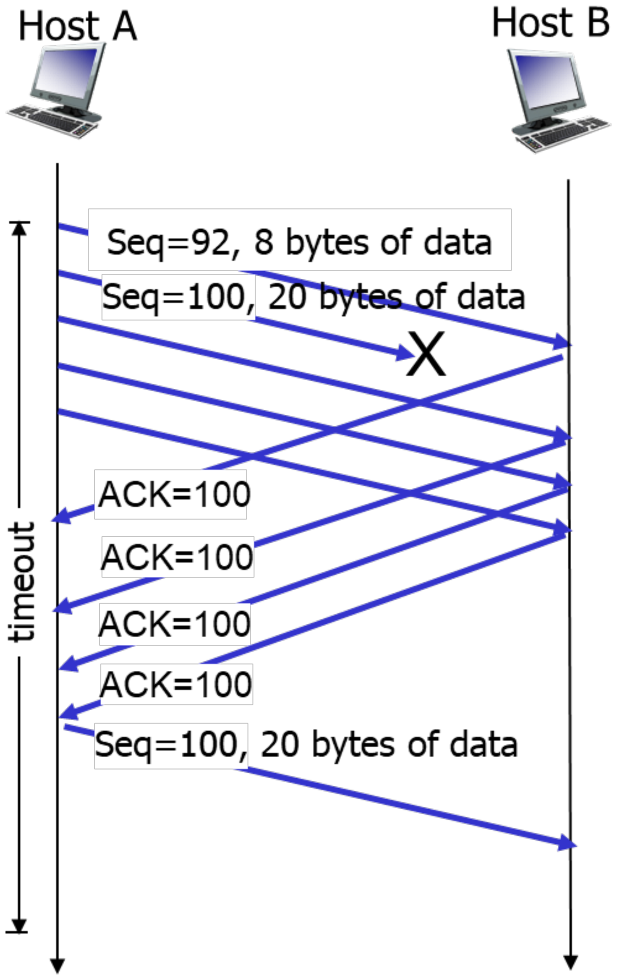
TCP Fast Retransmit

- 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



fast retransmit after sender receipt of triple duplicate ACK

Agenda

3.5 **connection-oriented transport: TCP**

- segment structure
- reliable data transfer
- **flow control**

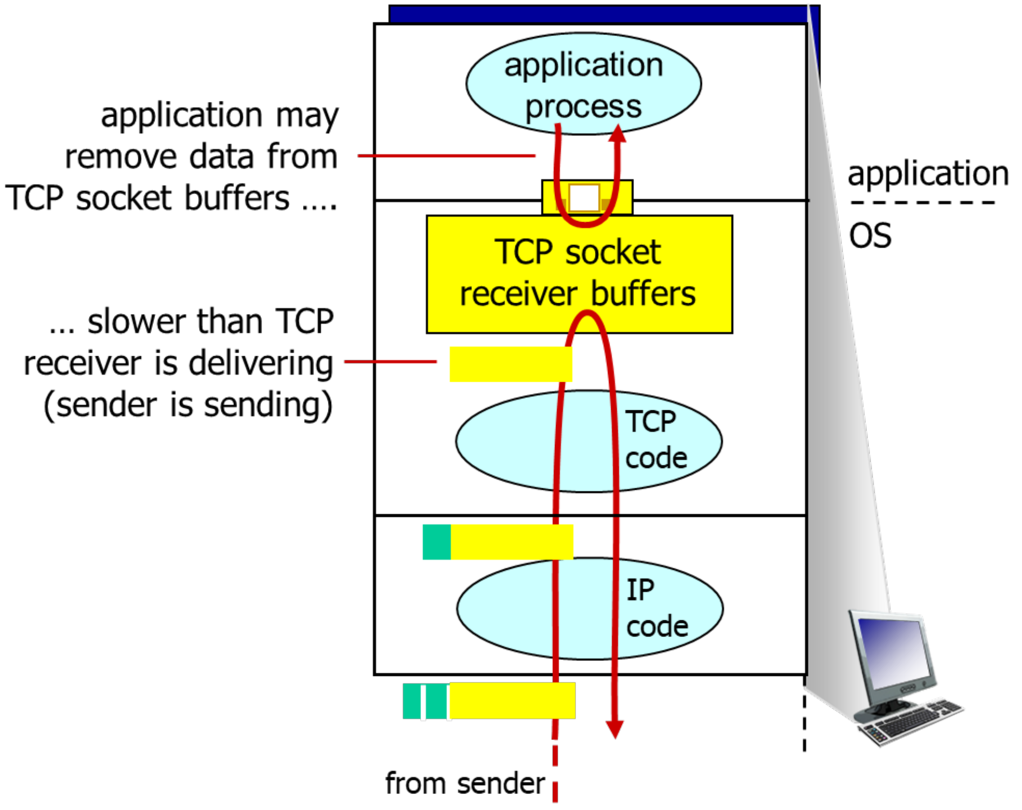
3.6 principles of congestion control

3.7 TCP congestion control

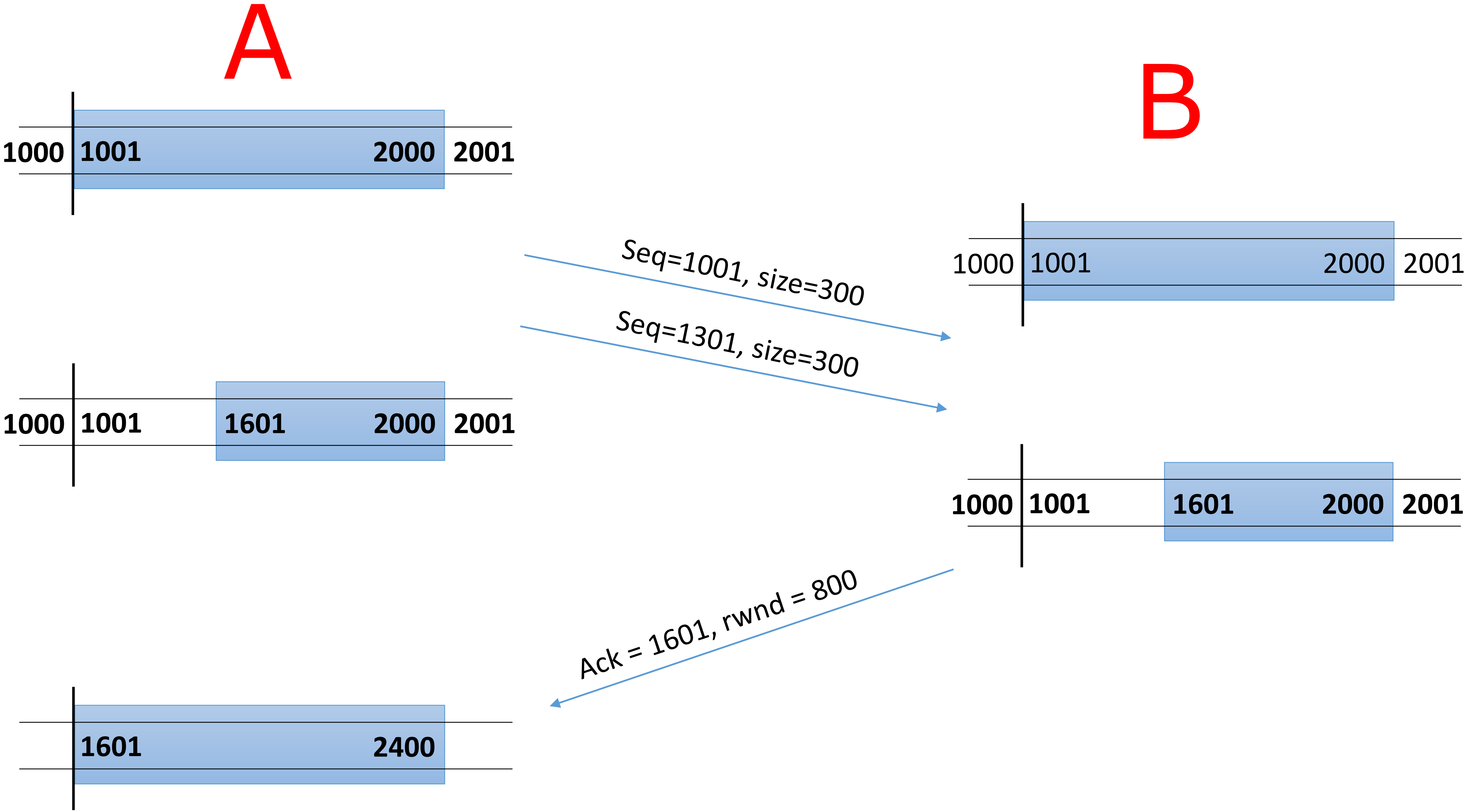
2.7 Socket programming / [DF] 16.9 Javascript sockets

TCP Flow Control

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


application process
application may remove data from TCP socket buffers ….
TCP socket receiver buffers
… slower than TCP receiver is delivering (sender is sending)
TCP code
IP code
application
OS
from sender

receiver protocol stack



A

1000 1001 2000 2001

1000 1001 1601 2000 2001

1601 2400

B

1000 1001 2000 2001

1000 1001 1601 2000 2001

Seq=1001, size=300

Seq=1301, size=300

Ack = 1601, rwnd = 800

Agenda

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control

3.6 principles of congestion control

3.7 TCP congestion control

2.7 Socket programming / [DF] **16.9** Javascript sockets

Principles of Congestion Control

congestion:

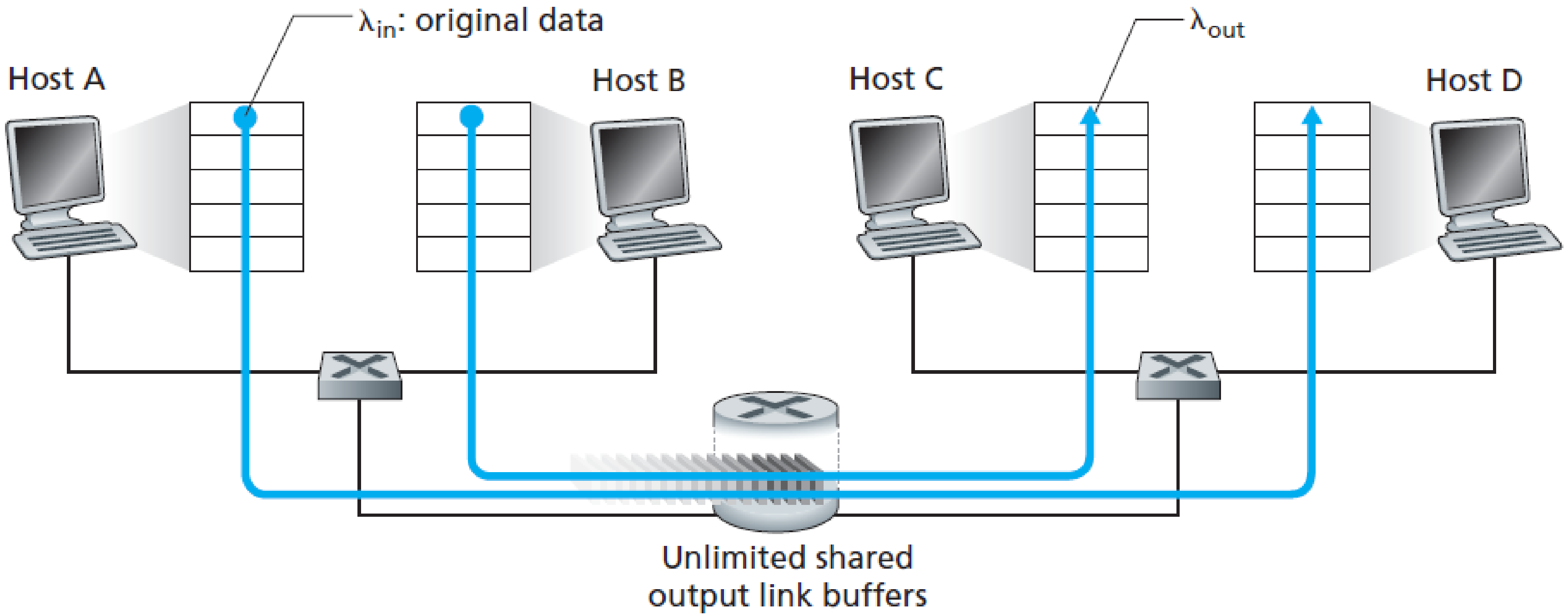
- informally: “too many sources sending too much data too fast for **network** to handle”
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)

- flow control vs congestion control
  - buffers of the receiver vs network resources
  - knowing the state of the receiving buffers vs estimating the state of the network

Causes/Costs of Congestion: Scenario 1

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

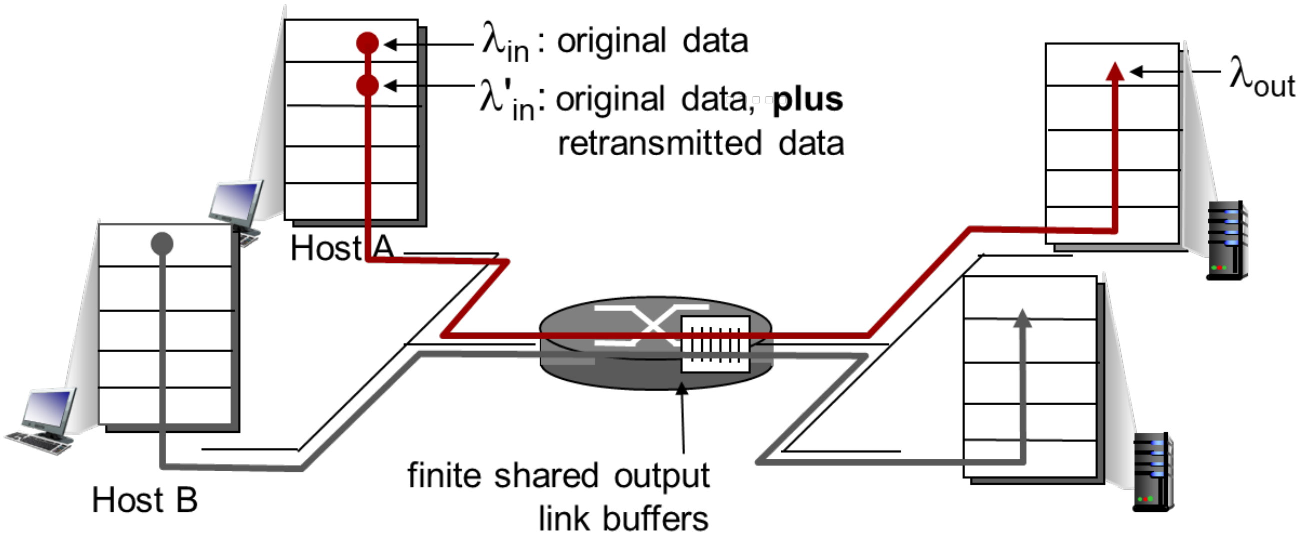
The capacity of the link must be split to all the data flows



λ_{in}: original data
λ_{out}
Host A
Host B
Host C
Host D
Unlimited shared output link buffers

Causes/Costs of Congestion: Scenario 2 (1/2)

- one router, **finite** buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes **retransmissions** : $\lambda'_{in} \geq \lambda_{in}$


λ_{in} : original data
λ'_{in}: original data, **plus** retransmitted data
λ_{out}
Host A
Host B
finite shared output link buffers

Idealizations:

- **perfect knowledge**
  - sender sends only when router buffers available
- **known loss** packets can be lost, dropped at router due to full buffers
  - sender only resends if packet **known** to be lost

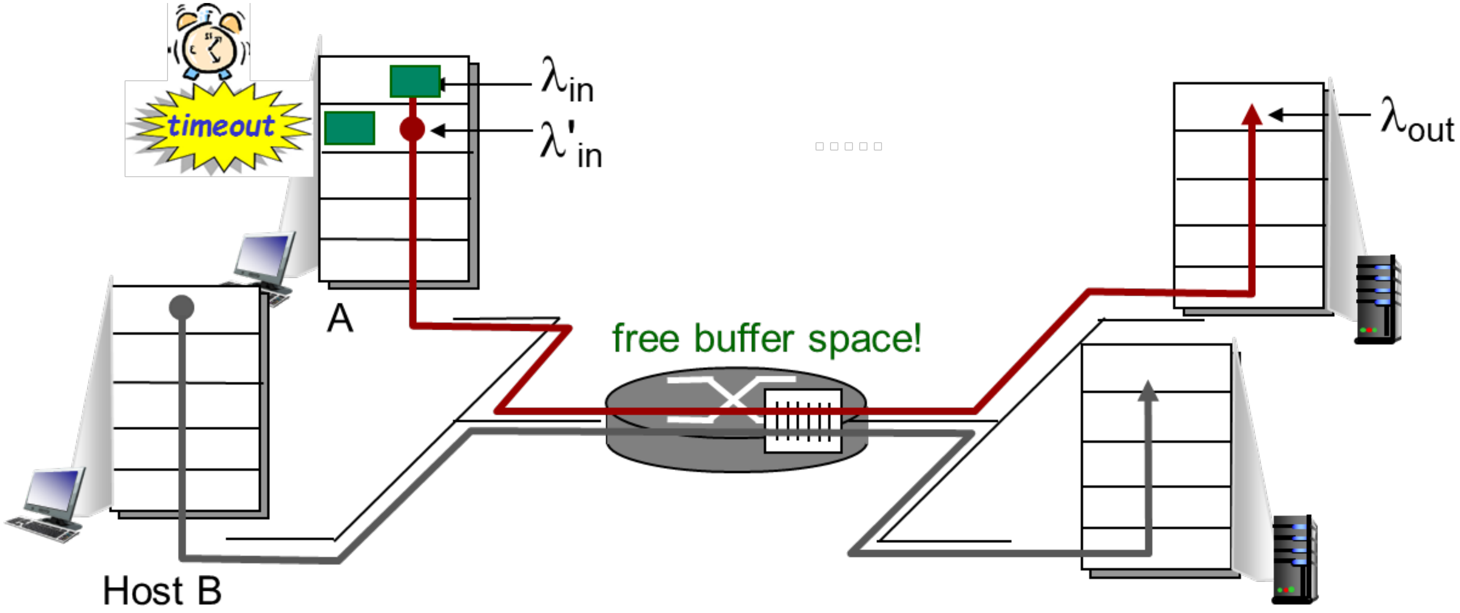
Causes/Costs of Congestion: Scenario 2 (2/2)

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending **two** copies, both of which are delivered

"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
    - decreasing goodput



Causes/Costs of Congestion: Scenario 3

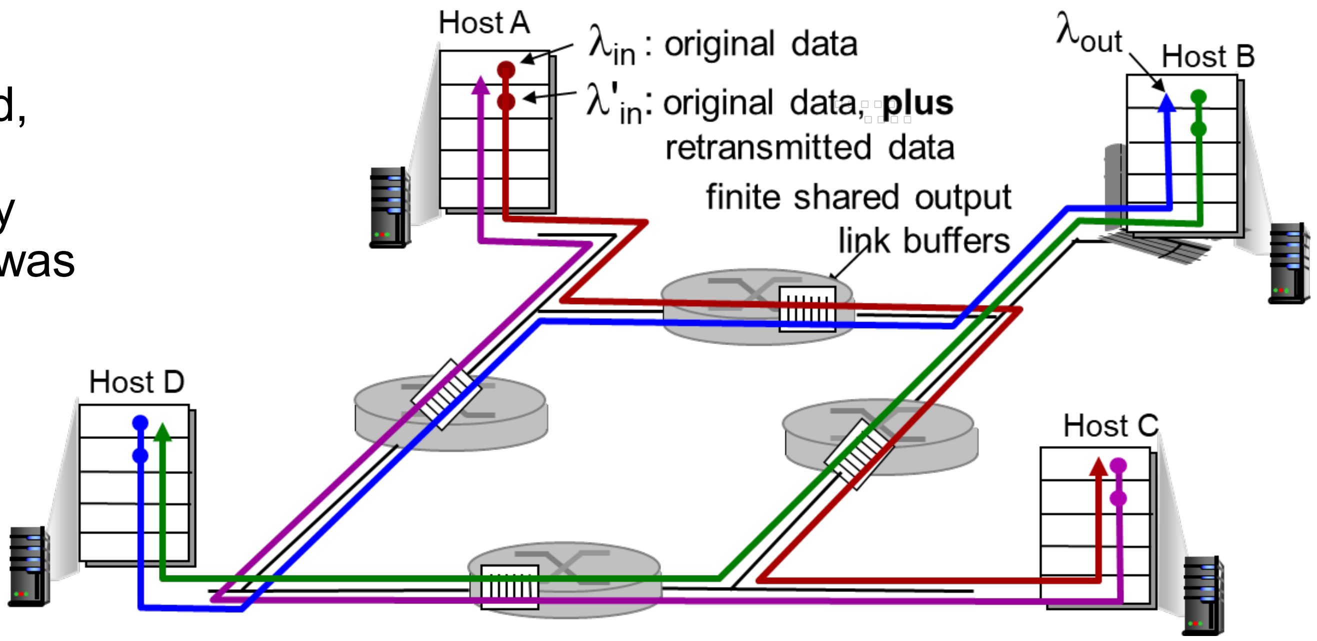
- four senders
- multihop paths
- timeout/retransmit

- **Q:** what happens as λ_{in} and λ_{in}' increase **?**
- **A:** as red λ_{in}' increases, all arriving blue pkts

at upper queue are dropped, blue throughput → 0

“cost” of congestion:

- when packet dropped, any upstream transmission capacity used for that packet was wasted



Host A
λ_{in} : original data
λ'_{in}: original data, **plus** retransmitted data
finite shared output link buffers
λ_{out}
Host B
Host D
Host C

Agenda

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control

3.6 principles of congestion control

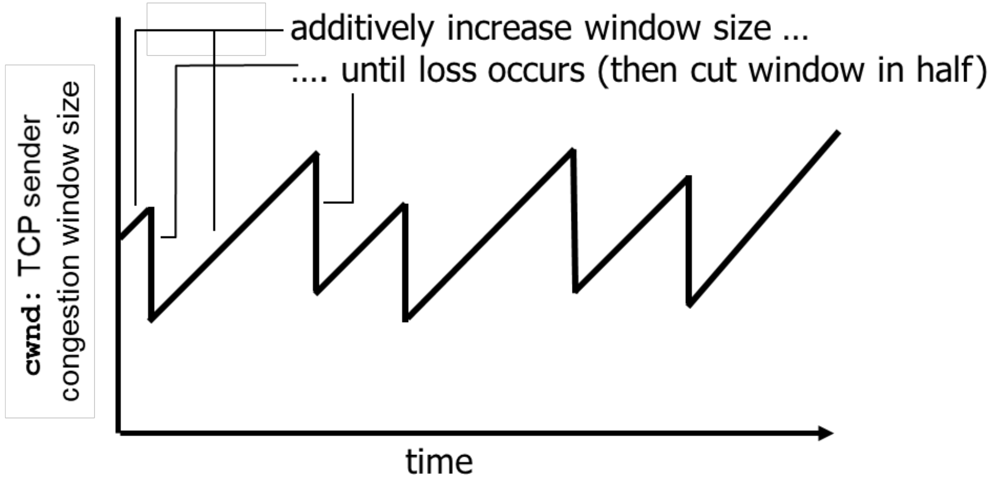
3.7 TCP congestion control

2.7 Socket programming / [DF] **16.9** Javascript sockets

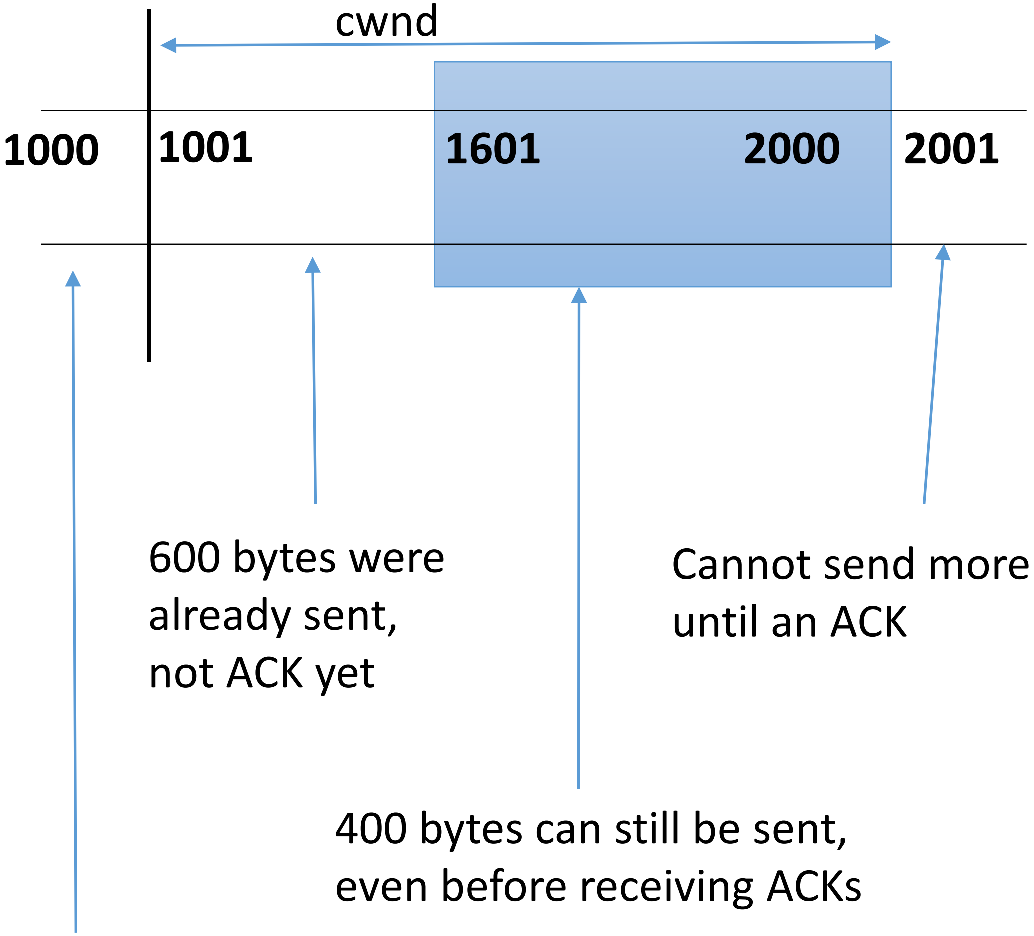
TCP Congestion Control: Additive Increase Multiplicative Decrease

- **Mechanism:** changing `cwnd`
- sender *increases* the window size, probing for usable bandwidth, until loss occurs
  - **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: the sender



600 bytes were already sent, not ACK yet

Cannot send more bytes until an ACK

400 bytes can still be sent, even before receiving ACKs

1000 bytes were already acknowledged

- sender limits transmission
- **cwnd** is dynamic, function of perceived network congestion

TCP sending rate:

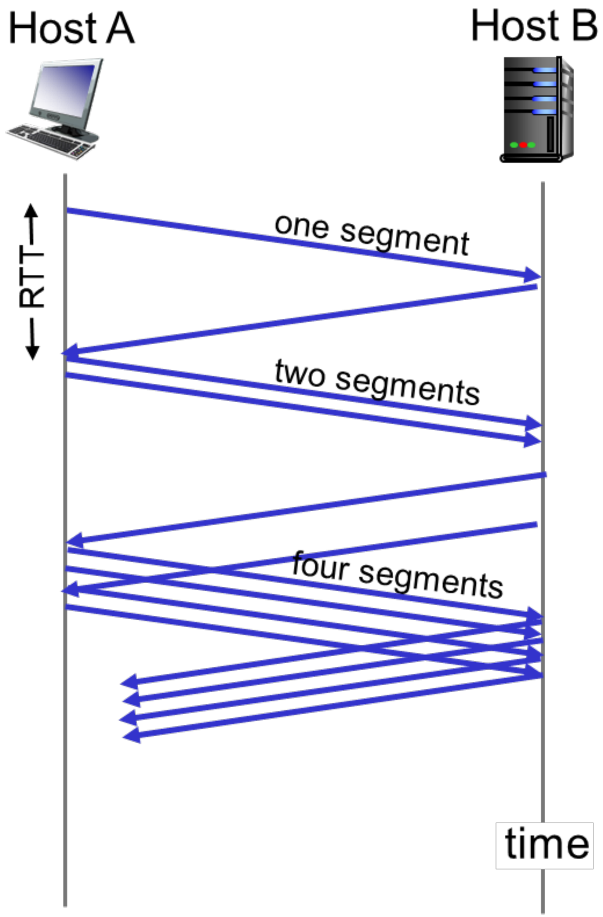
- **roughly:** send cwnd bytes, wait RTT for ACKS, then send more bytes

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

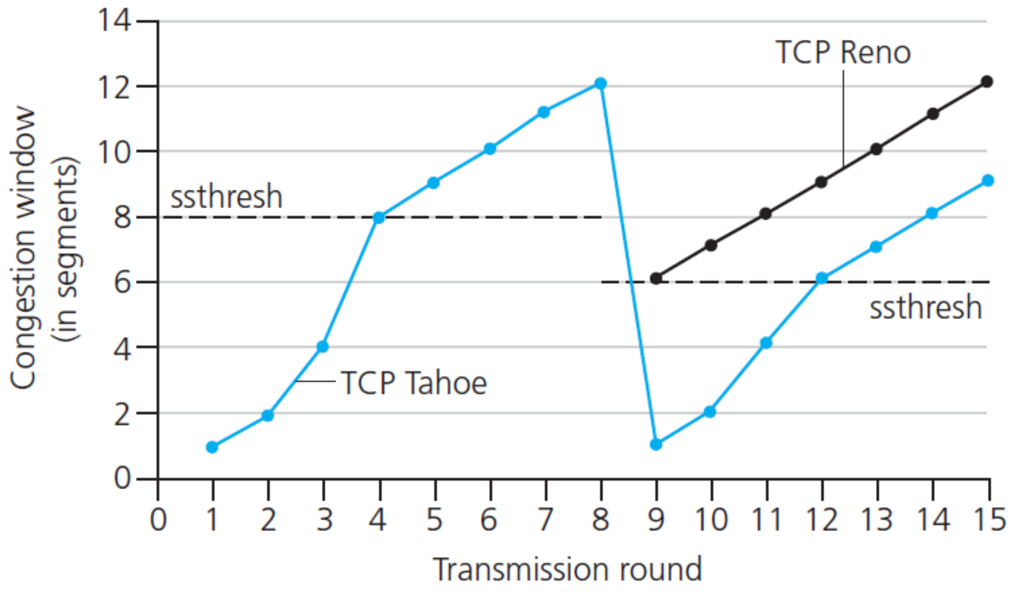
$$\texttt{LastByteSent-} \atop \texttt{LastByteAcked} \leq \texttt{cwnd}$$

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
- initial rate is slow but ramps up exponentially fast



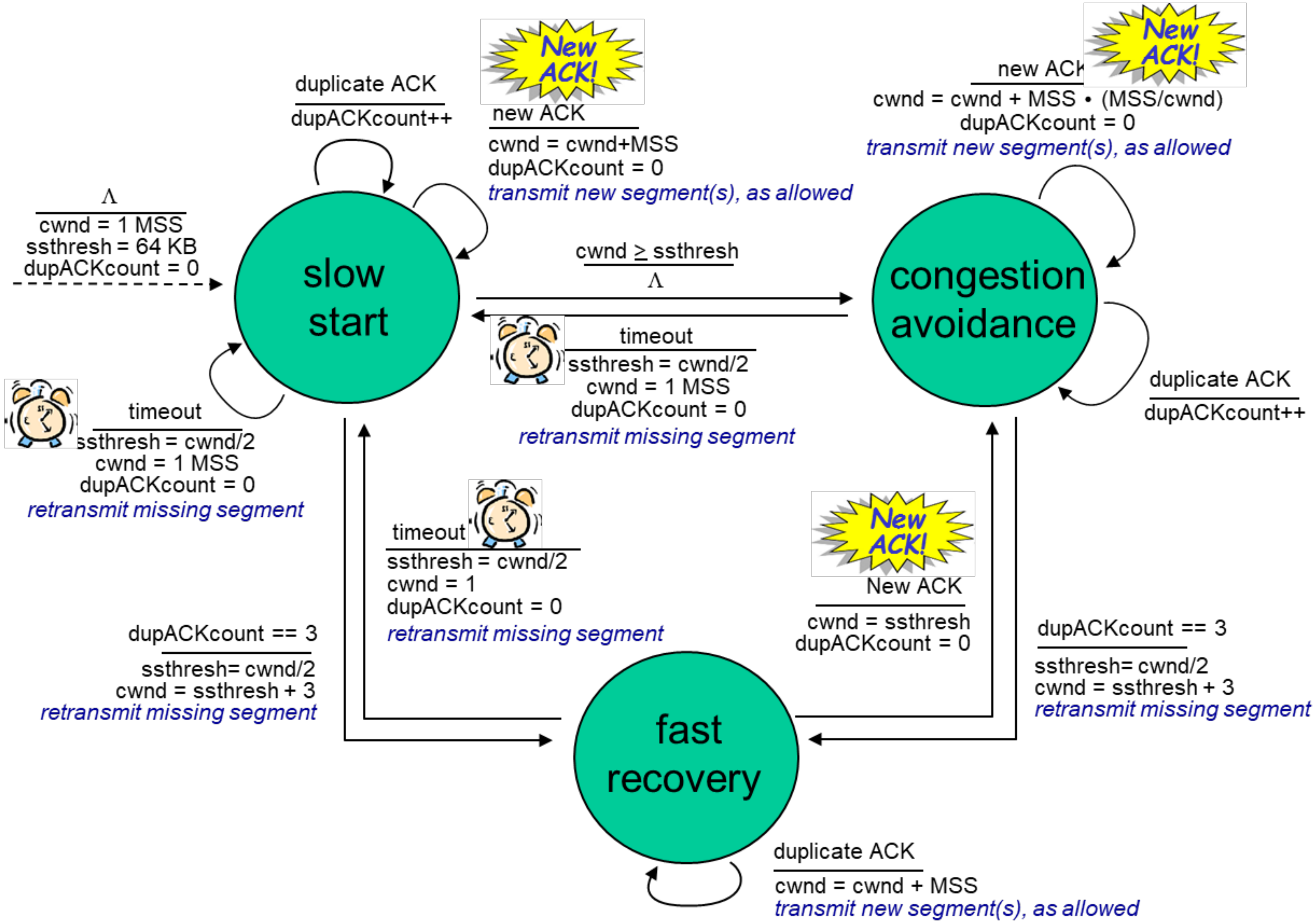
TCP: Detecting, Reacting to Loss



- loss indicated by timeout:
  - **cwnd** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold (a configuration parameter, controlled by the sender*), 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)

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

Summary: TCP Congestion Control



Λ

cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0

slow start

duplicate ACK

dupACKcount++

New ACK!

new ACK

cwnd = cwnd+MSS
dupACKcount = 0
transmit new segment(s), as allowed

timeout

ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment

cwnd ≥ ssthresh

Λ

timeout

ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment

congestion avoidance

New ACK!

new ACK

cwnd = cwnd + MSS • (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed

duplicate ACK

dupACKcount++

timeout

ssthresh = cwnd/2
cwnd = 1
dupACKcount = 0
retransmit missing segment

dupACKcount == 3

ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment

New ACK!

New ACK

cwnd = ssthresh
dupACKcount = 0

dupACKcount == 3

ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment

fast recovery

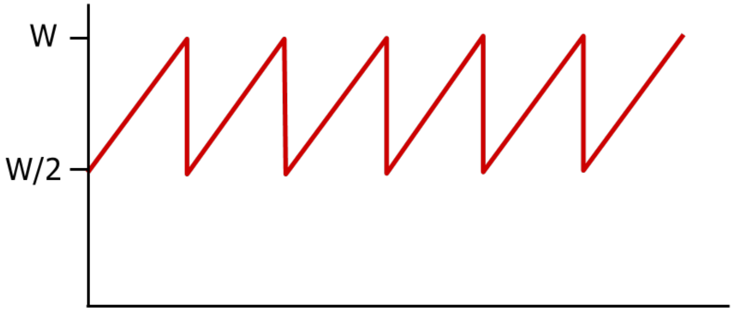
duplicate ACK

cwnd = cwnd + MSS
transmit new segment(s), as allowed

TCP Throughput

- average TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - average window size (# in-flight bytes) is
  - average thruput is $\frac{3}{4}W$ • per RTT $\frac{3}{4}W$

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



Fairness

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

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
- example, link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate $\frac{R}{10}$
  - new app asks for 11 TCPs, gets $\frac{R}{2}$

Agenda

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control

3.6 principles of congestion control

3.7 TCP congestion control

2.7 Socket programming / [DF] 16.9 Javascript sockets

The Socket

- Formed by the concatenation of a port value and an IP address
    - Unique throughout the Internet
- Used to define an API
    - Generic communication interface for writing programs that use TCP or UDP
- Stream sockets
    - All blocks of data sent between a pair of sockets are guaranteed for delivery and arrive in the order that they were sent
- Datagram sockets
    - Delivery is not guaranteed, nor is order necessarily preserved
- Raw sockets
    - Allow direct access to lower-layer protocols

Defining a socket:

- Two main things to do
  - Addressing
    - Specifying a host and a service (IP + port)
    - It is a tuple (“www.cs.aau.dk”, 80)
  - Data transport
    - Mainly TCP (SOCK_STREAM) or UDP (SOCK_DGRAM)

Socket Programming with UDP

UDP: no “connection” between client & server

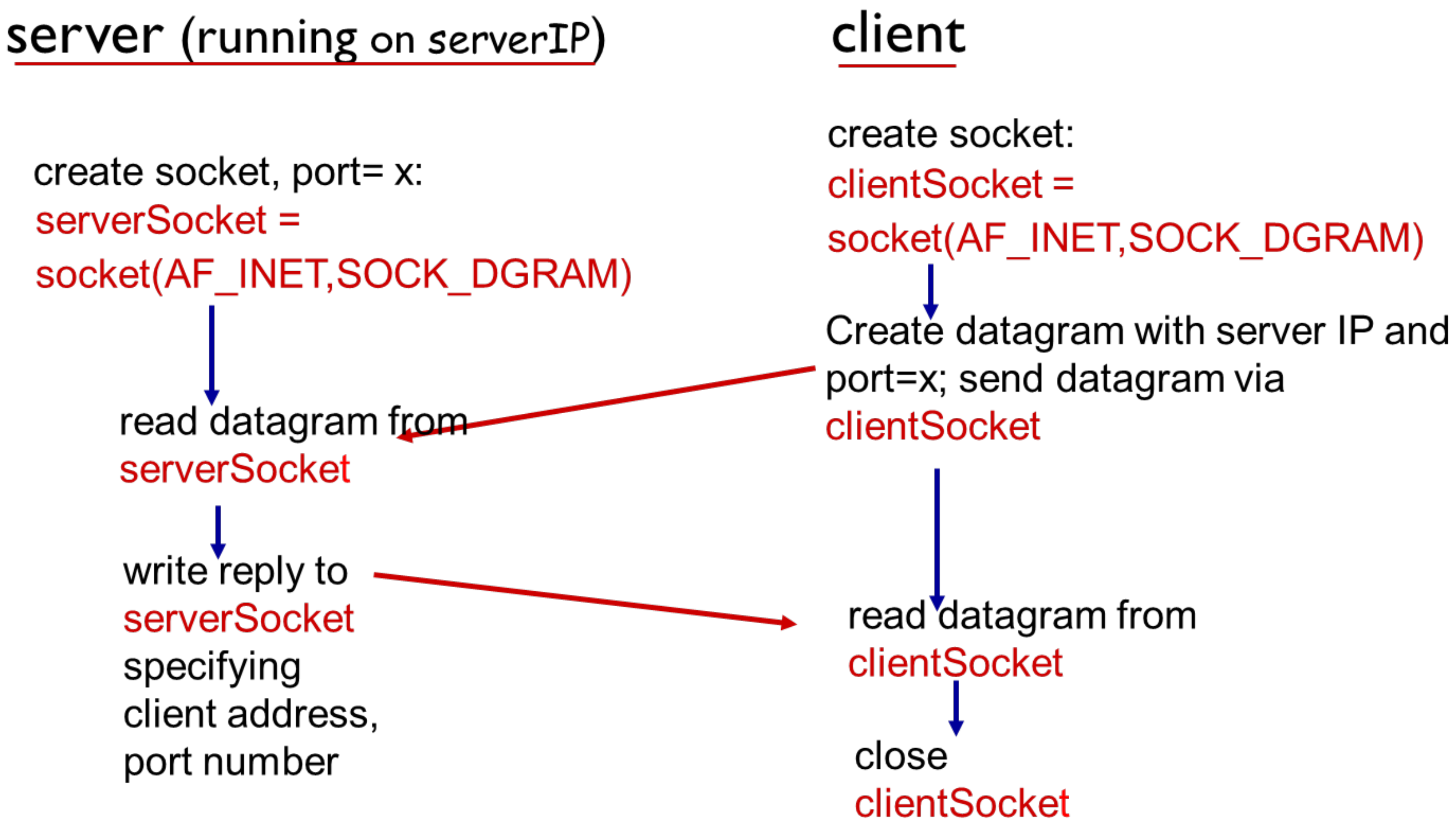
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides **unreliable** transfer of groups of bytes (“datagrams”) between client and server

Client/Server Socket Interaction: UDP



server (running on serverIP)

create socket, port= x:
serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:
clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Socket Programming with TCP

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

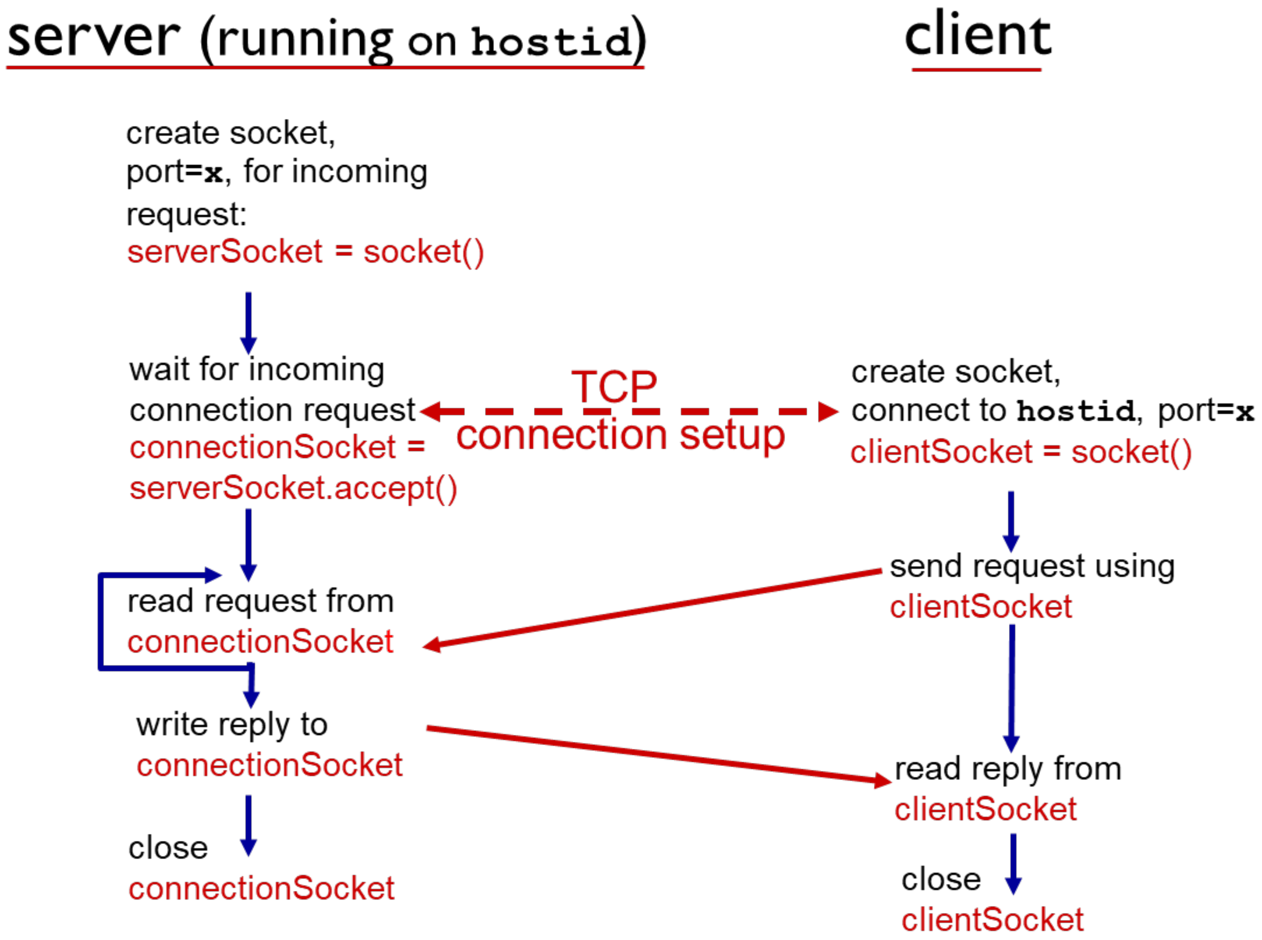
- Creating TCP socket, specifying IP address, port number of server process
- **when client creates socket:** client T CP establishes connection to server TCP

- when contacted by client, **server TCP creates new socket** for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/Server Socket Interaction: TCP



server (running on **hostid**)

client

create socket,
port=**x**, for incoming
request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **hostid**, port=**x**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

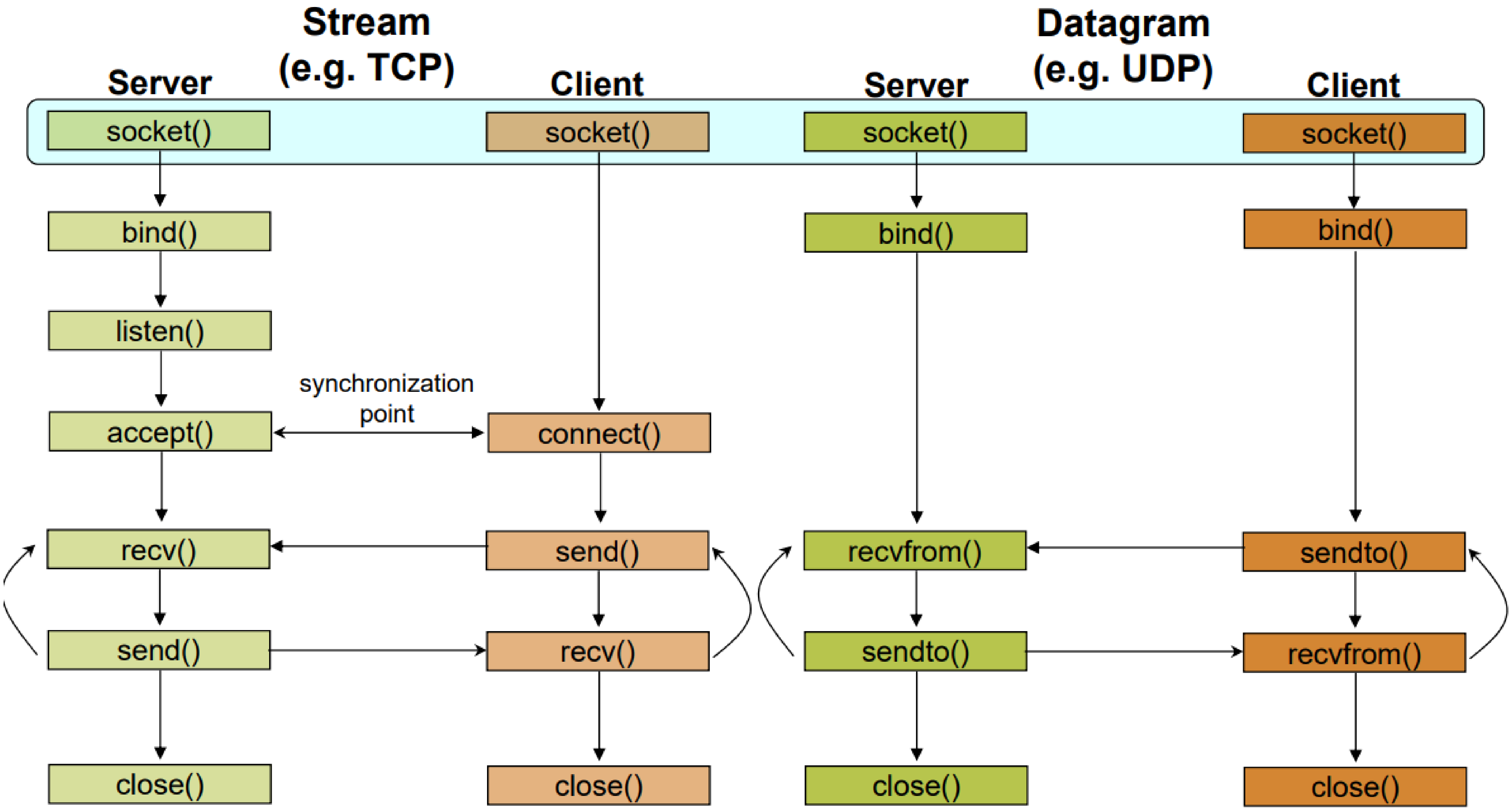
write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

The Socket API



Always check the return value of these calls !!!!

C Server

1) Create a socket, bind it to a port, make it listen:

```
s = socket(AF_INET, SOCK_STREAM)
s.bind(my_address, 9000)
listen(s, 5) # up to 5 pending connections
```

2) Accept an incoming connection to have a connected socket:

```
c = accept(s, (struct sockaddr*)&client_addr,
&addrlen);
print(“connected with %s“, client_addr)
```

3) Communicate:

```
send(c, “you are connected\n”, 19)
```

6) Close the Socket:

```
close(c);
```

7) Go to Step 2.

C Client

1) Create a socket, connect it:

```
s = socket(AF_INET, SOCK_STREAM)
connect(s, destination)
```

3) Communicate (receive data):

```
recv(d, data, 1024)
printf("Received %s", data)
```

4) Close the socket:

```
close(s);
```

Javascript Server

1) Import module “net”, create a socket, define what must be done all the time there is a connection:

```
const net = require('net');
const server = net.createServer((socket) => { … });
```

2) The function is in an implicit infinite loop. Extract the address of the client, log, and answer to the client:

```
net.createServer((socket) => {
  addr = socket.address();
  console.log("%s:%d connected", addr.address, addr.port);
  socket.end("you are " + addr.address +
      ":" + addr.port + "\n");
});
```

3) Bind and listen on the server socket:

```
server.listen(9999);
```

Javascript Client

1) Import module “net”, create a socket:

```
const net = require('net');
const client = new net.Socket();
```

2) Connect the socket to the host provided by the command line:

```
client.connect({ port: 9999, host: process.argv[2] });
```

3) Communicate (receive data) and log:

```
client.on('data', (data) => {
  console.log("Received:\n" + data.toString('utf-8'));
});
```

Windows programming

- Microsoft "version" of socket programming is called Winsock
- Same philosophy, implementation differences:
  - `WINSOCK_API_LINKAGE SOCKET WSAAPI socket( int af, int type, int protocol )`
    - It's not returning an integer!
- Need to initialize the Winsock subsystem:
  - `WSADATA wsaData;`
  - `iResult = WSAStartup(MAKEWORD(2,2), &wsaData);`
  - `if (iResult != 0) { exit(-1); }`

Issues:

- Reading/writing to a socket may involve partial data transfer
  - send() returns actual bytes sent
  - recv() length is only a maximum limit
- For TCP, the data stream is continuous---no concept of records, etc.
  - One recv() can return the strings sent in two send()
  - A send() can provide strings to two recv()s with a small maximum limit
- How to tell if there is no more data, in the sense that the connection has been (half-)closed?
  - recv() will return empty string
  - ret = s.recv(1000)
    - … and ret == “”

Issues with the recv()

- If a peer is doing a cycle like:

```
while (l = s.recv()):
      do_something(l)
```

- There is the problem of termination:
  - Until `recv()` provides data, it doesn’t end
  - When there is no more data, the program is stuck on the blocking `recv()` call
- Four solution:
  - Doing a `close()` on the other side
  - Send a *special* termination string
  - Use a timeout
  - Send the number of character that will be sent in the beginning

Drawbacks of each solution

- Doing a `close()` on the other side
  - The `read()` will return (with a “” string)
  - But the socket cannot be used anymore
- Send a *special* termination string
  - But if it was part of the normal communication flow, it could quit the application too early
- Use a timeout
  - The application gets slow since it needs to wait until the timeout
  - Should the network be slow, the timeout could end the application too early
- Send the number of character that will be sent in the beginning
  - In some applications, you don’t know the number of characters in advance

SLUT