Institute of Telecommunications
Warsaw University of Technology
2019

Internet Technologies and Standards

- Piotr Gajowniczek
- Andrzej Bąk

Transport Layer

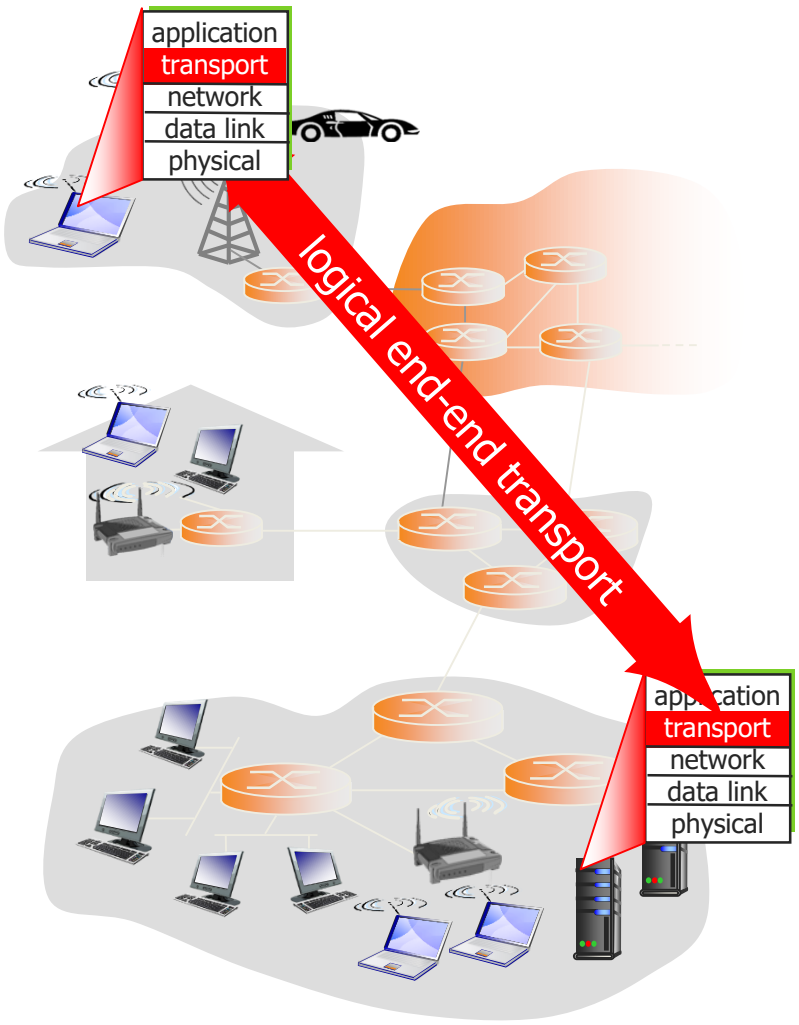
TCP/UDP protocols

outline

- *transport-layer*
  - services
  - multiplexing and demultiplexing
  - Internet sockets (API)
- *connectionless transport: UDP*
- *principles of reliable data transfer*
- *connection-oriented transport: TCP*
  - segment structure
  - connection management
  - reliable data transfer
  - flow control
- *principles of congestion control*
- *TCP congestion control*

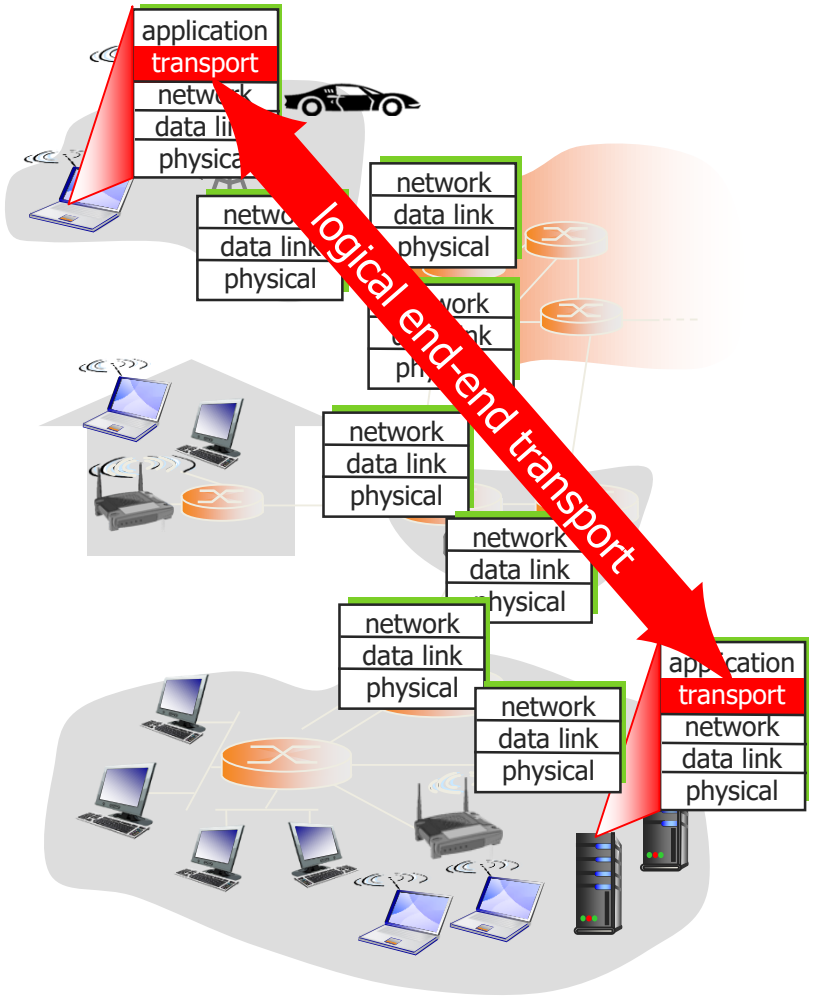
Internet transport layer services

- *Transport layer provides logical communication between app. processes running on different hosts*
  - Transport protocols run in end systems
- *Encapsulation*
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- *Multiplexing*
  - support for multiply application usage of transport layer at the same time



Internet transport-layer services

- *Reliable, in-order delivery: TCP*
  - connection management
  - flow control
  - congestion control
- *Unreliable, unordered delivery: UDP*
  - connectionless
  - extension of “best-effort” IP
- *Services not available:*
  - delay guarantees
  - bandwidth guarantees



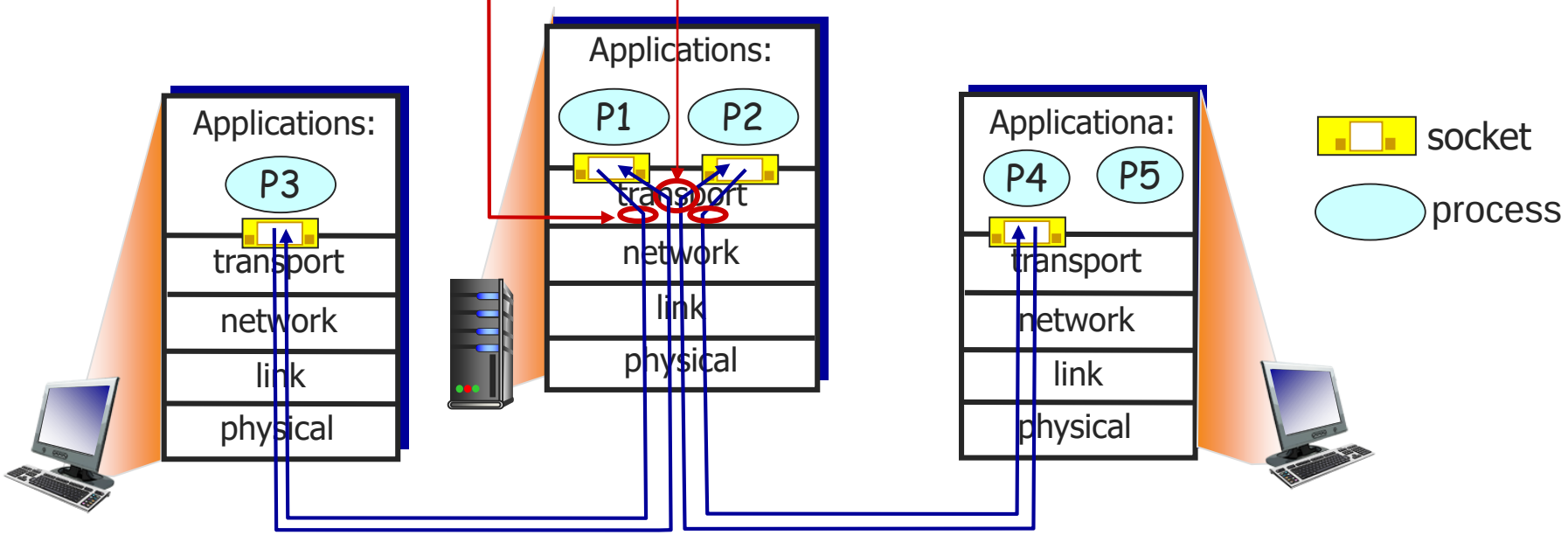
Multiplexing/Demultiplexing

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



Internet Sockets

- *Socket*
  - It is an endpoint abstraction through which an application may send and receive data over the network (Internet)
- *POSIX sockets (Berkeley sockets)*
  - Standard Unix API for Internet sockets used for inter-process communication (IPC), originated with the 4.2BSD Unix operating system in 1983
  - Berkeley sockets are now part of the POSIX standard
  - Implemented in C
  - Most of the other APIs follows the POSIX implementation (usually implemented as wrappers around C library)
- *WINSOCK – windows sockets (very similar)*

Internet sockets

- *Two types of (TCP/IP) sockets*
  - Stream sockets (e.g. uses TCP)
    - provide reliable byte-stream service
    - connection oriented
  - Datagram sockets (e.g. uses UDP)
    - provide best-effort datagram service
    - messages up to 65.500 bytes
- *Over the network socket is identified by*
  - Internet address
  - Port number
  - Protocol
- *In the local process the socket is identified by*
  - Socket descriptor – socket handle similar to the file handle

Socket API

- *socket() – creates socket, return socket descriptor*
- *bind() – assigns socket to IP address and port*
- *listen() – informs socket to listen for client connections (stream sockets)*
- *accept() – accepts connections and creates dedicated socket for client (stream sockets)*
- *connect() – attempts to establish connection with server (stream sockets)*
- *send(), recv() – data send and receive*
- *sendto(), recvfrom() – data send and receive (datagram sockets)*
- *close() – release the connection/free socket*

Socket API

- *Creates new socket:*

```
int sockid=socket(family, type, protocol)
```

  - sockid: socket descriptor, an integer (like a file-handle)
  - family: communication domain
    - PF_INET, IPv4 protocols, Internet addresses (typically used)
    - PF_UNIX, Local communication, File addresses
  - type: communication type
    - SOCK_STREAM -reliable, 2-way, connection-based service
    - SOCK_DGRAM -unreliable, connectionless, messages of maximum length
  - protocol: protocol type
    - IPPROTO_TCP IPPROTO_UDP

NOTE: socket call does not specify where data will be coming from, nor where it will be going to – it just creates the interface!

Socket API

- *Assigns an IP address and port for use by the socket*
- *Must be called if the application wants to receive data on the datagram sockets*
- *Must be called on the server in case of the stream sockets*

```
int status=bind(sockid, &addrport, size)
```

- sockid: socket descriptor
- addrport: struct sockaddr, the (IP) address and port of the machine
  - for TCP/IP server, internet address is usually set to INADDR_ANY, i.e., chooses any incoming interface
- size: the size (in bytes) of the addrport structure
- status: upon failure -1 is returned

Socket API

- *Instructs the stream socket to listen for connections*

```
int status=listen(sockid, queueLimit)
```

  - sockid: socket descriptor
  - queueLimit : # of active participants that can “wait” for a connection
  - status: 0 if listening, -1 if error

- *listen() is non-blocking: returns immediately*

Socket API

- *The client establishes a connection with the server by calling connect()*

```
int status=connect(sockid, &foreignAddr, size)
```

  - Sockid: socket to be used in connection
  - foreignAddr: struct sockaddr: address of the passive participant
  - size: integer, size of the foreignAddr structure
  - status: 0 if successful connect, -1 otherwise
- *connect() is blocking*
- *Client's port number is randomly picked*

Socket API

- *The server gets a socket for an incoming client connection by calling accept()*

```
int clisock=accept(sockid, &clientAddr, &size)
```

  - clisock: descriptor of the socket used for data transfer
  - sockid: the orig. socket (being listened on)
  - clientAddr: struct sockaddr, returns address of the active participant
  - size: returns size of clientAddr structure
- *accept() is blocking:*
  - waits for connection before returning
  - dequeues the next connection on the queue for socket sockid

Socket API

```
int count=send(sockid, msg, msgLen, flags);
```

- msg: const void[], message to send
- msgLen: message size
- flags: options
- count: number of bytes actually sent (-1 on failure)

```
int count=recv(sockid, buf, bufLen, flags);
```

- buf: void[], receiver buffer
- bufLen: buffer size
- flags: options
- count: number of bytes actually received, -1 error, 0 end of transmission

- *Calls to send() i recv() are blocking*
- *Used to send/receive data over stream sockets*

Socket API

```
int count=sendto(sockid, msg, msgLen, flags,
                 &foreignAddr, size);
```

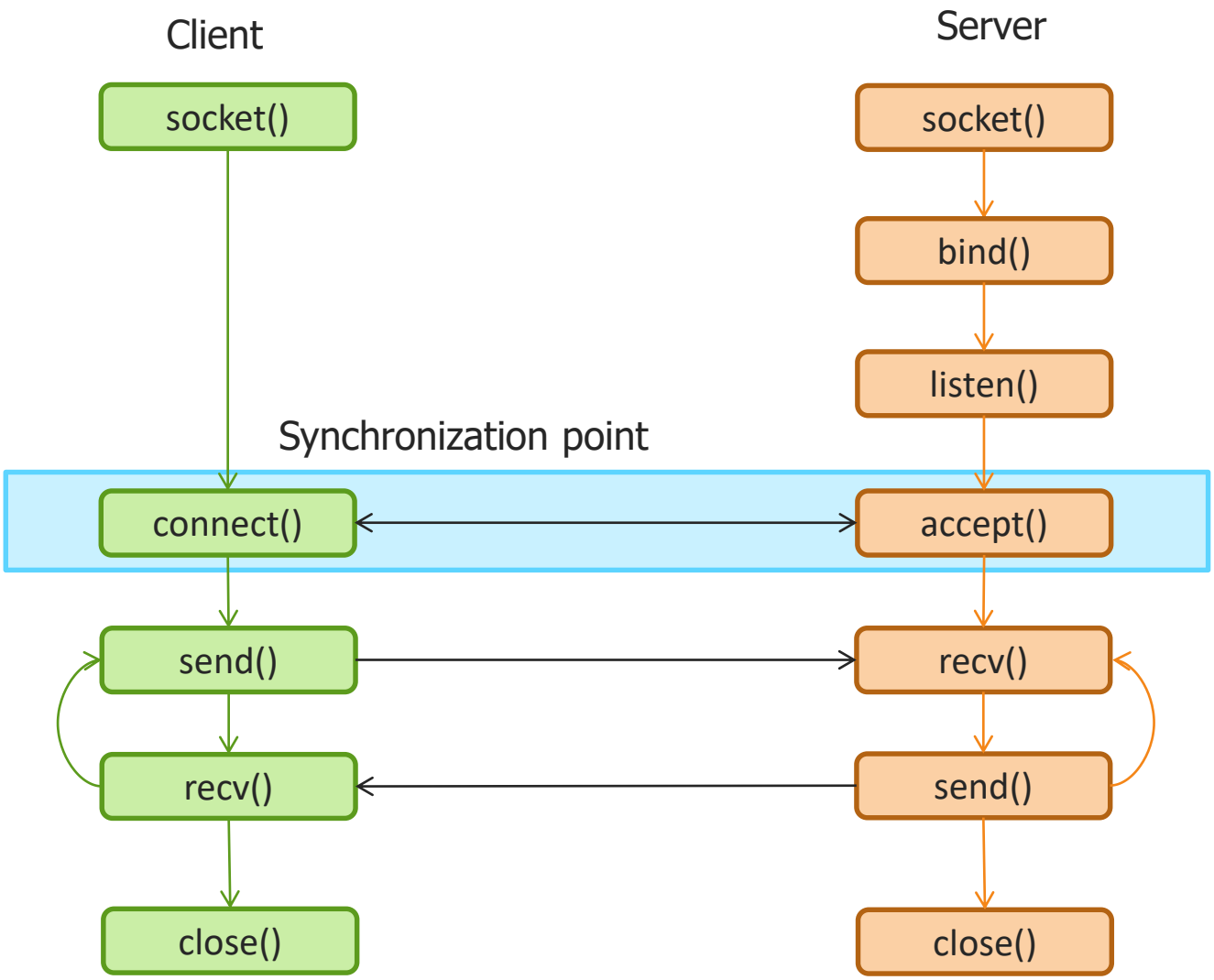
- msg, msgLen, flags – same as in send()
- foreignAddr – address of the remote socket
- size: size of the foreignAddr structure
- count: number of bytes actually sent (-1 on failure)

```
int count=recvfrom(sockid, buf, bufLen,
                   flags,&clientAddr, size);
```

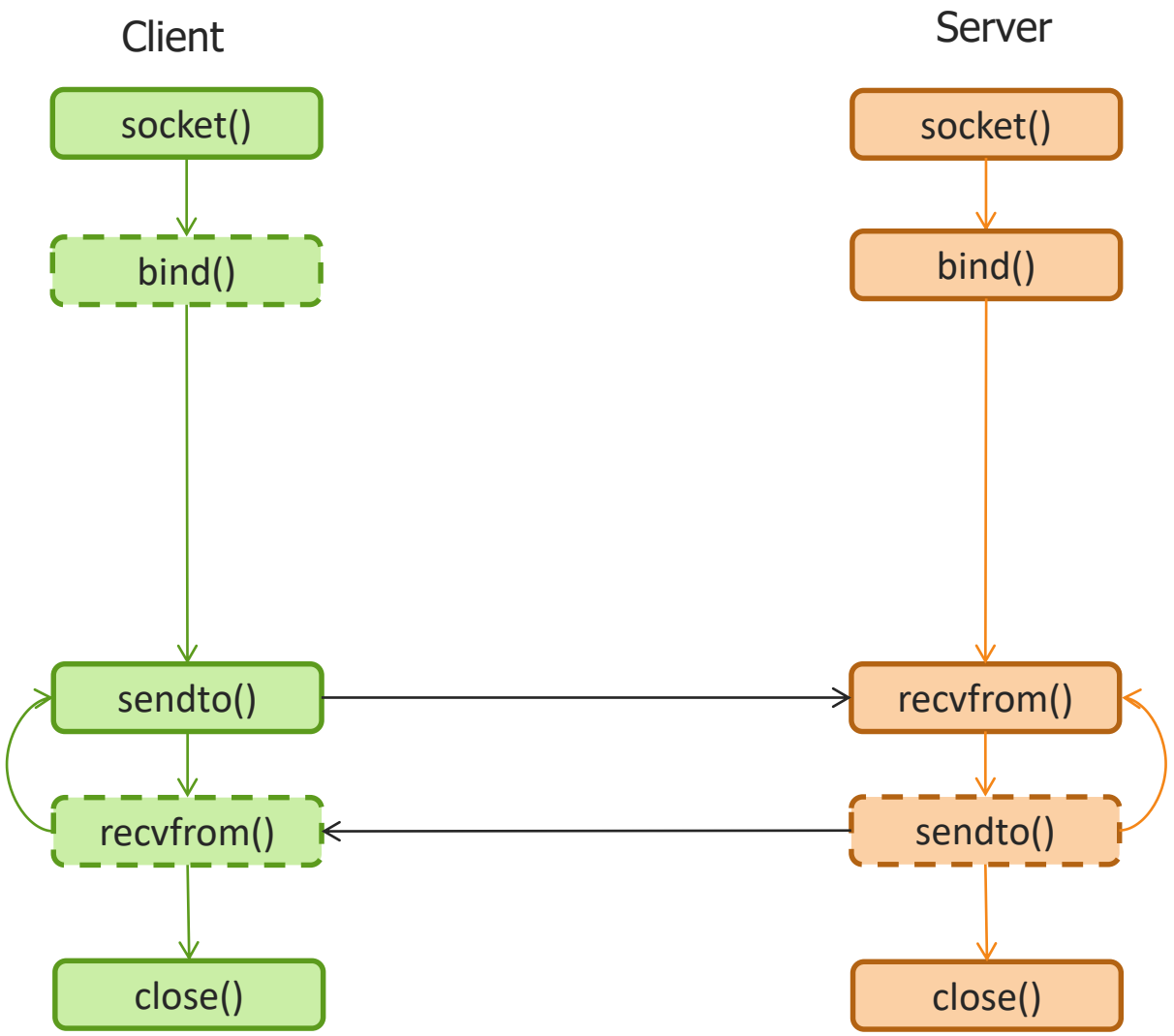
- buf, bufLen, flags – same as in recv()
- clientAddr – address of the client
- size: size of the clientAddr structure
- count: liczba bajtów odebranych, -1 błąd, 0 koniec transmisji

- *Calls to sendto() i recvfrom() are blocking*
- *Used to send/receive data over datagram sockets*

Stream (TCP) sockets



Datagram (UDP) sockets



Socket API

- *I/O multiplexing - handle multiple operation at the same time (reading and writing from/to multiple sources)*
- *I/O multiplexing is typically used in networking applications in the following scenarios:*
  - When a client is handling multiple inputs (interactive input and a network socket)
  - When a client is handling multiple sockets at the same time
  - If a TCP server handles both a listening socket and its connected sockets
  - If a server handles both TCP and UDP
  - If a server handles multiple services and perhaps multiple protocols

Socket API

- *I/O models*
  - blocking I/O
    - application blocks on single input until data available
  - nonblocking I/O
    - polling inputs by application
    - wastes CPU
  - I/O multiplexing (select and poll functions)
    - application waits for data on multiple inputs
    - application blocks
  - signal driven I/O
    - kernel signals the application when data is available
    - application does not block
  - asynchronous I/O (the POSIX aio_ functions)
    - kernel signals the application when the input operation is completed
    - application does not block

Socket API

- *Waits until one of the file descriptors (sockets) becomes ready:*

```
int status=select(nfds, readset, writeset,
                  exceptset, timeout)
```

  - nfds: highest-numbered descriptor + 1
  - ***readset, writeset, excepset:*** sets of descriptors (fd_set*)
  - timeout: timeout for waiting for descriptor to be ready for I/O operation
  - status: 0 timeout, the number of sockets ready or -1 on error
- *The descriptor sets are modified on return to indicate sockets ready for I/O operation*
- *Four macros are defined to manipulate the sets:*
  - FD_ZERO() clears a set
  - FD_SET() and FD_CLR() adds/removes a descriptor to/from a set
  - FD_ISSET() tests to see if a descriptor is part of the set

Socket API

- *Waits until one of the file descriptors (sockets) becomes ready:*

  int status=poll(fds, nfds, timeout)

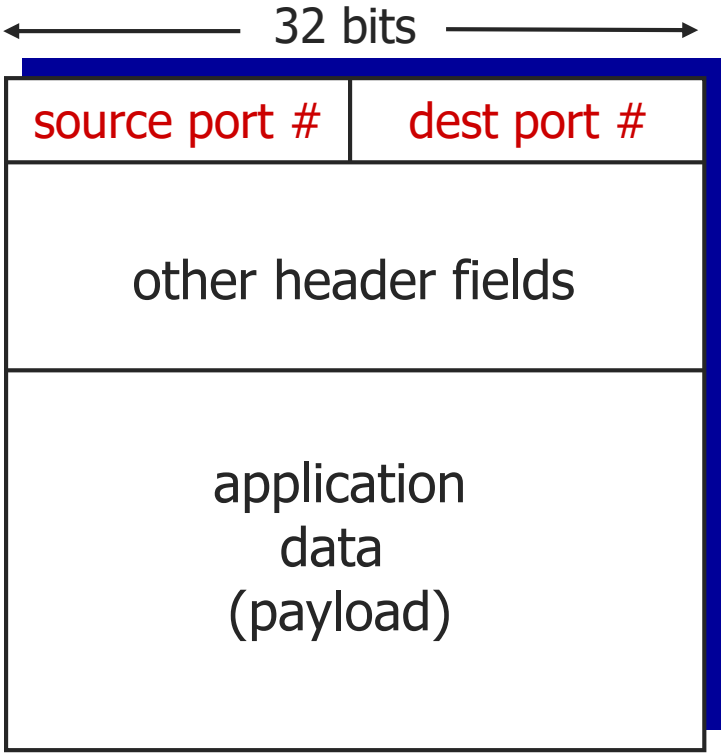
  - fds: array of structures:

```
struct pollfd { int fd; /*file descriptor*/
                short events; /*requested events*/
                short revents; /*returned events*/}
```

  - ***nfds:*** size of the array
  - timeout: timeout for waiting for descriptor to be ready for I/O operation
  - status: 0 timeout, the number of sockets ready or -1 on error
- *On return the revents fields are modified to indicate the occurred events on the descriptors*

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*

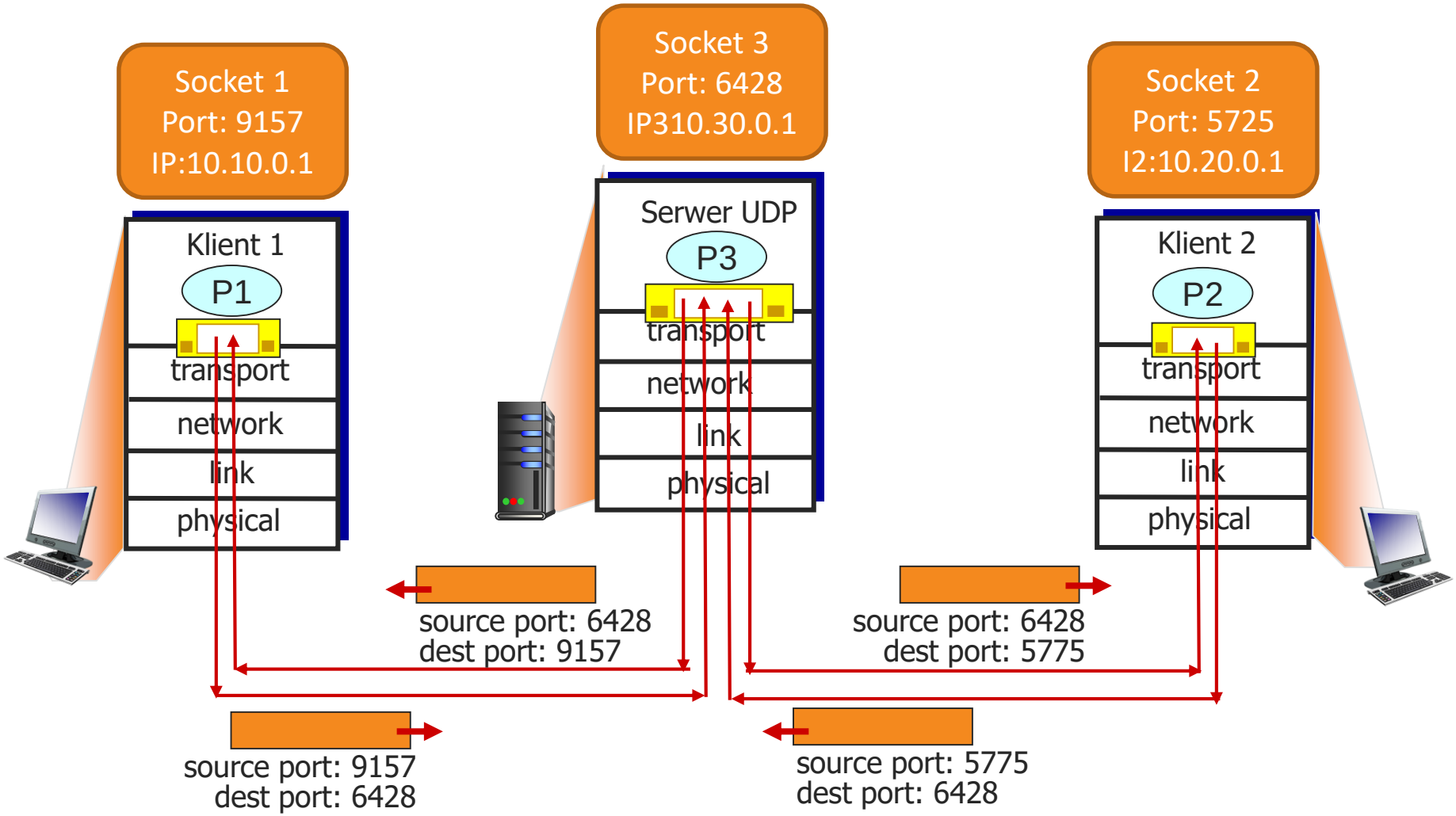


TCP/UDP segment format

connectionless demultiplexing

- *When creating datagram to send into UDP socket, host must specify*
  - destination IP address
  - destination port #
- *When host receives UDP segment:*
  - checks destination port # in segment
  - directs UDP segment 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 destination host*

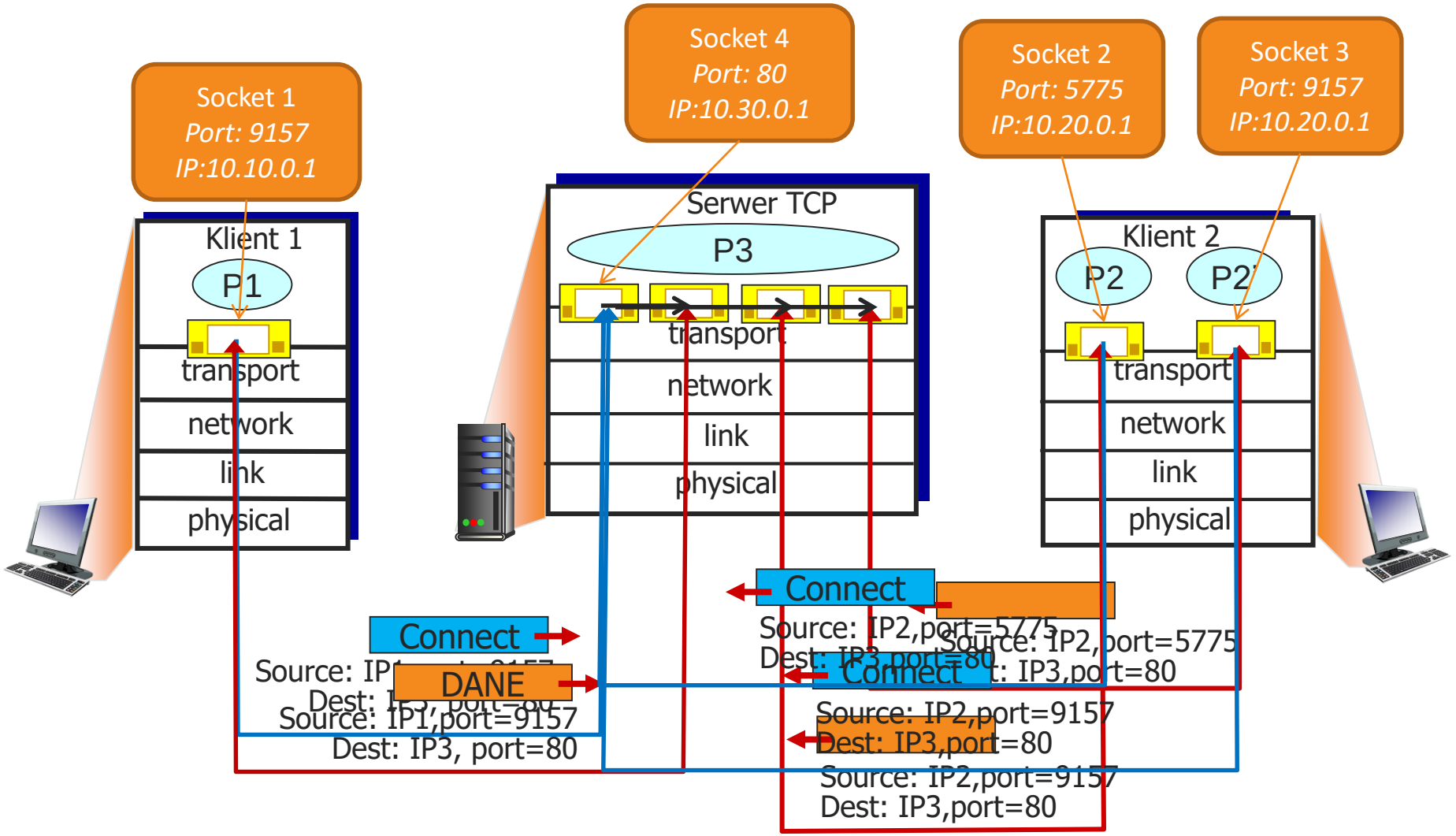
connectionless demux: example



connection-oriented demux

- *Server host may support many simultaneous TCP connections:*
  - Listen socket is created to receive connection requests
    - It is bind to given IP address and port
    - The conection requests are demux to this socket
  - The connection sockets are created on demand as connection requests arrive
    - Each socket is bind to single TCP connection automatically by connect()/accept() API functions
- *TCP connection 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*

connection-oriented demux: example



outline

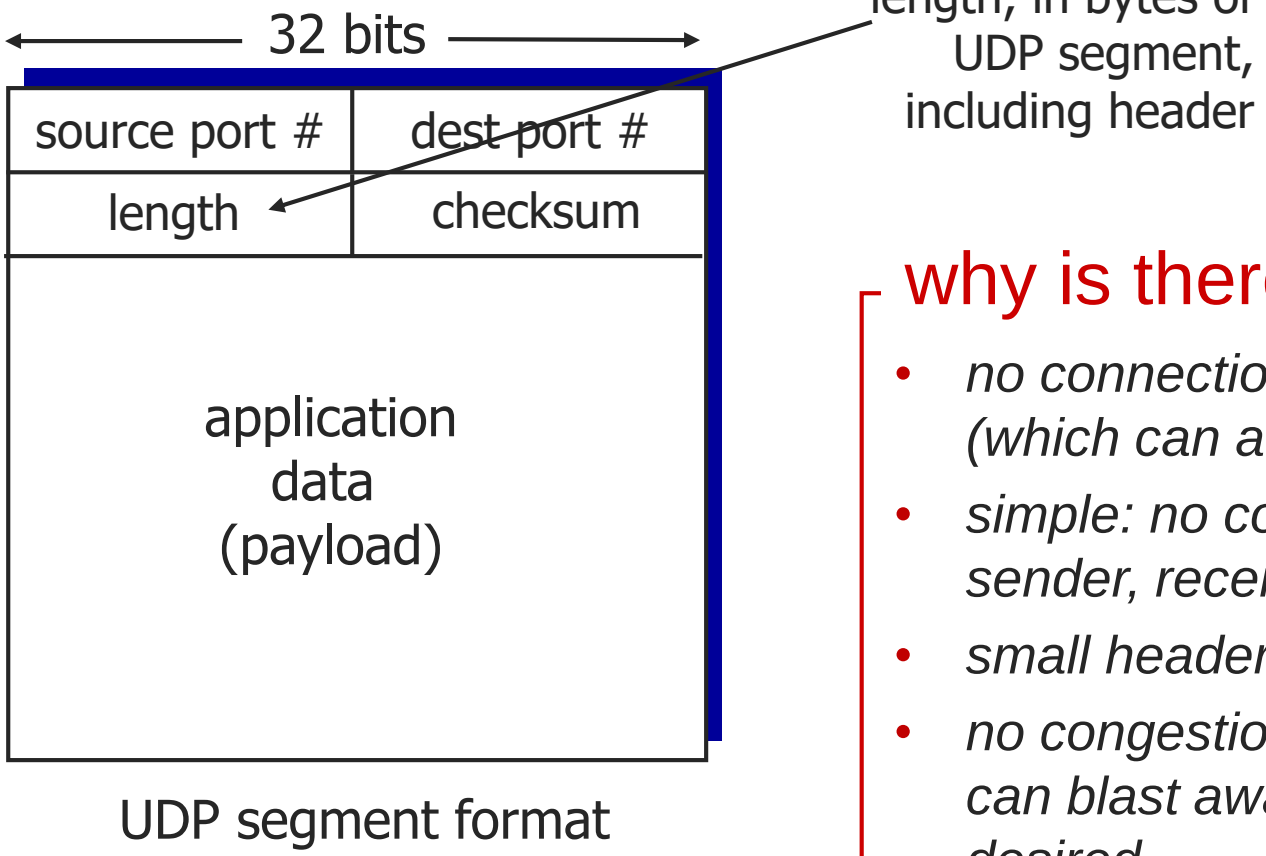
- *transport-layer*
  - services
  - multiplexing and demultiplexing
  - Internet sockets (API)
- *connectionless transport: UDP*
- *principles of reliable data transfer*
- *connection-oriented transport: TCP*
  - segment structure
  - connection management
  - reliable data transfer
  - flow control
- *principles of congestion control*
- *TCP congestion control*

UDP: User Datagram Protocol [RFC 768]

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

- *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: segment header



UDP segment format

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 segment

sender:

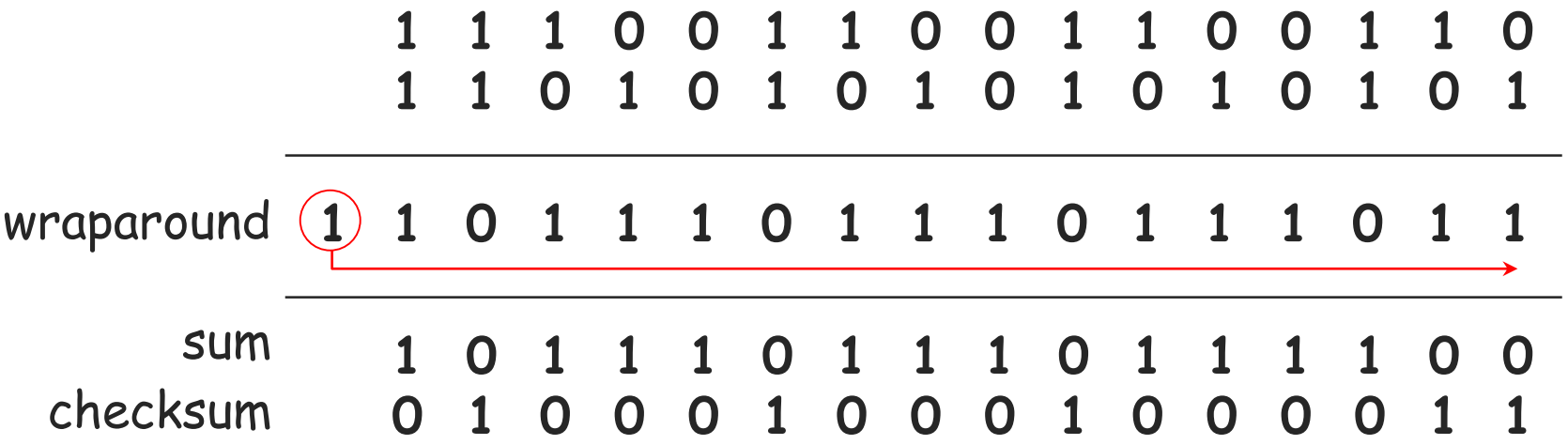
- *treat segment contents, including header fields, as sequence of 16-bit integers*
  - adds pseudo header
- *checksum: addition of segment contents (flip bits after calculation)*
- *sender puts checksum value into UDP checksum field*

receiver:

- *compute the sum of received segment content (including checksum field)*
- *should get all ones:*
  - NO - error detected
  - YES - no error detected. But errors may still be present.
    - detects 1-bit errors

Internet checksum: example

- *example: add two 16-bit integers*

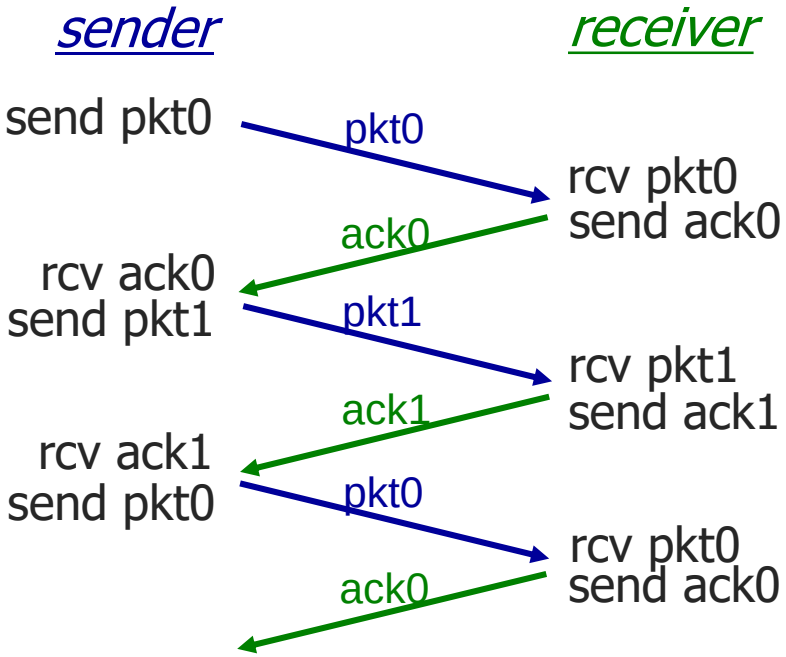


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

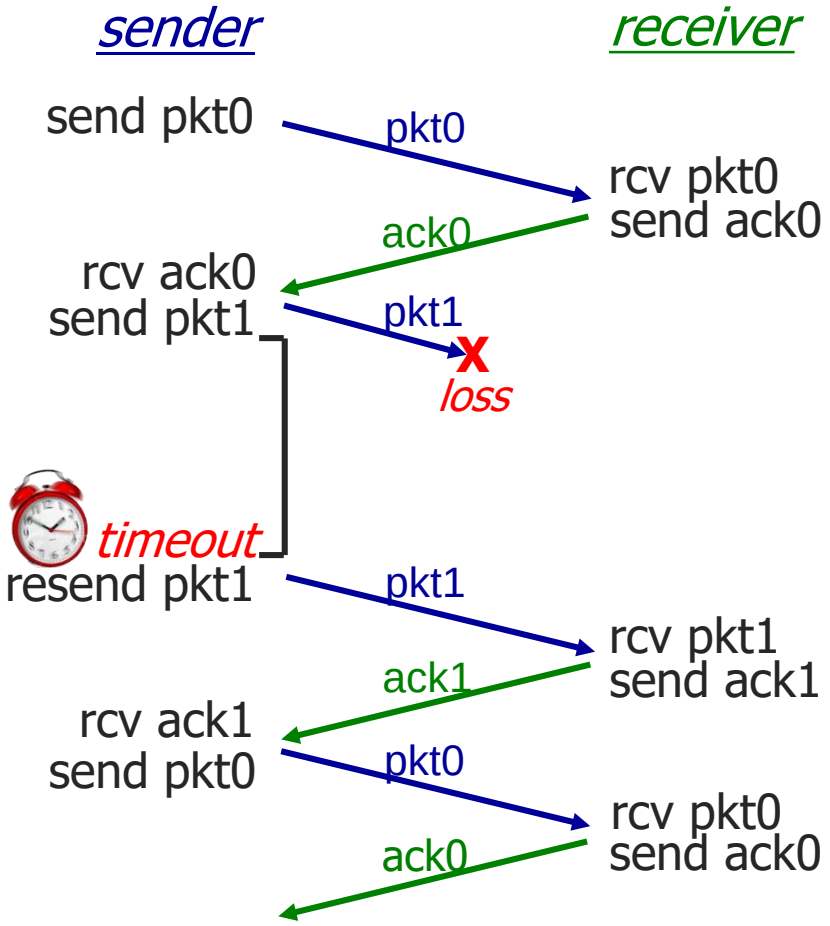
outline

- *transport-layer*
  - services
  - multiplexing and demultiplexing
  - Internet sockets (API)
- *connectionless transport: UDP*
- *principles of reliable data transfer*
- *connection-oriented transport: TCP*
  - segment structure
  - connection management
  - reliable data transfer
  - flow control
- *principles of congestion control*
- *TCP congestion control*

Stop-and-wait protocol

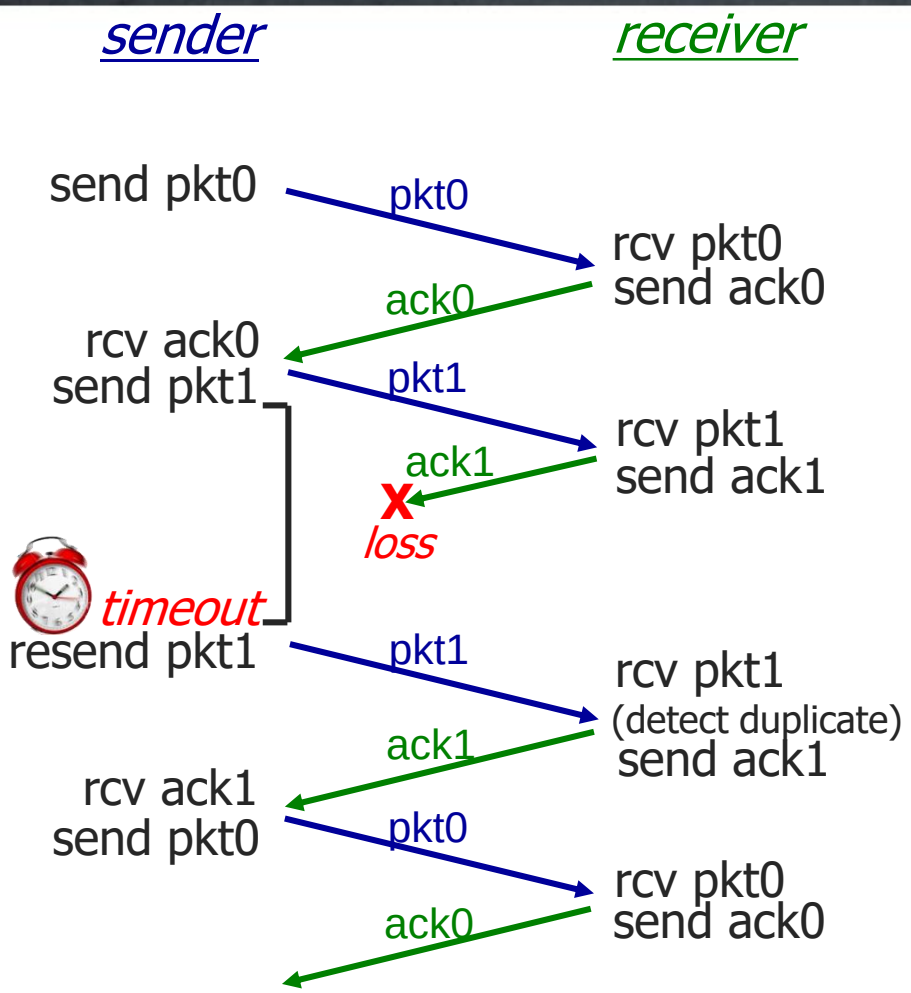


(a) no loss

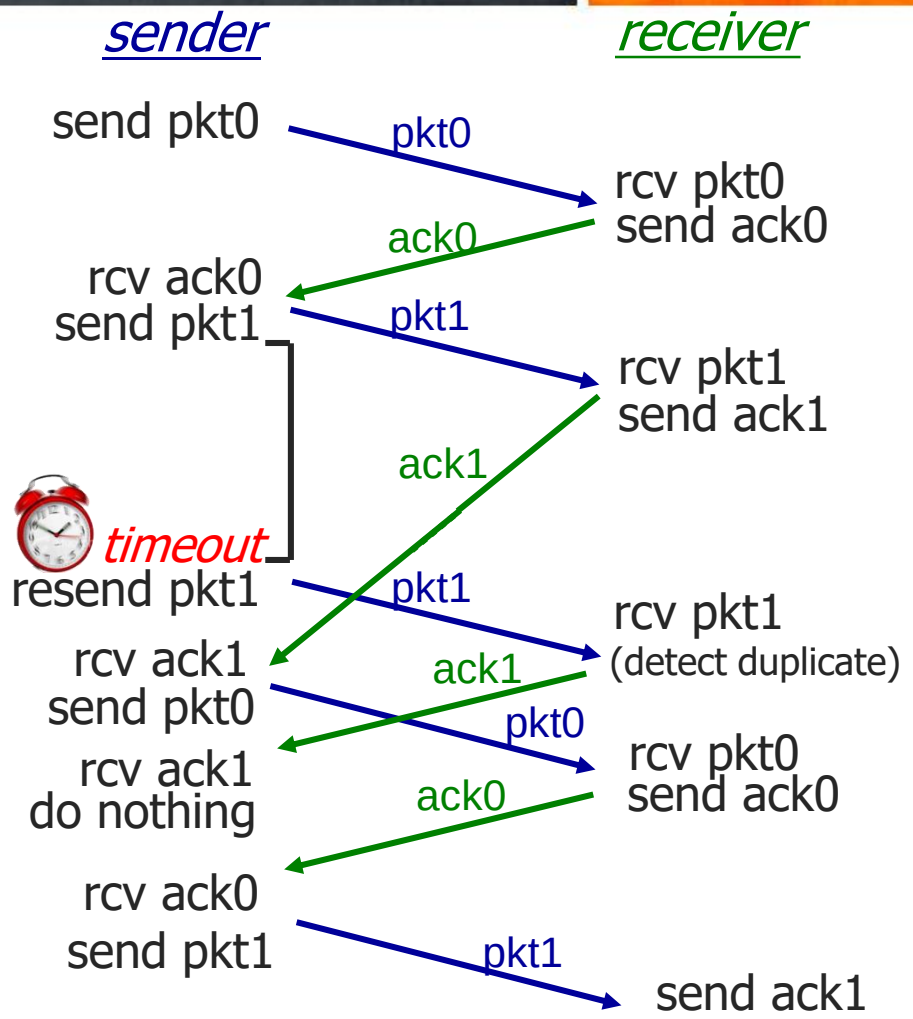


(b) packet loss

Stop-and-wait protocol



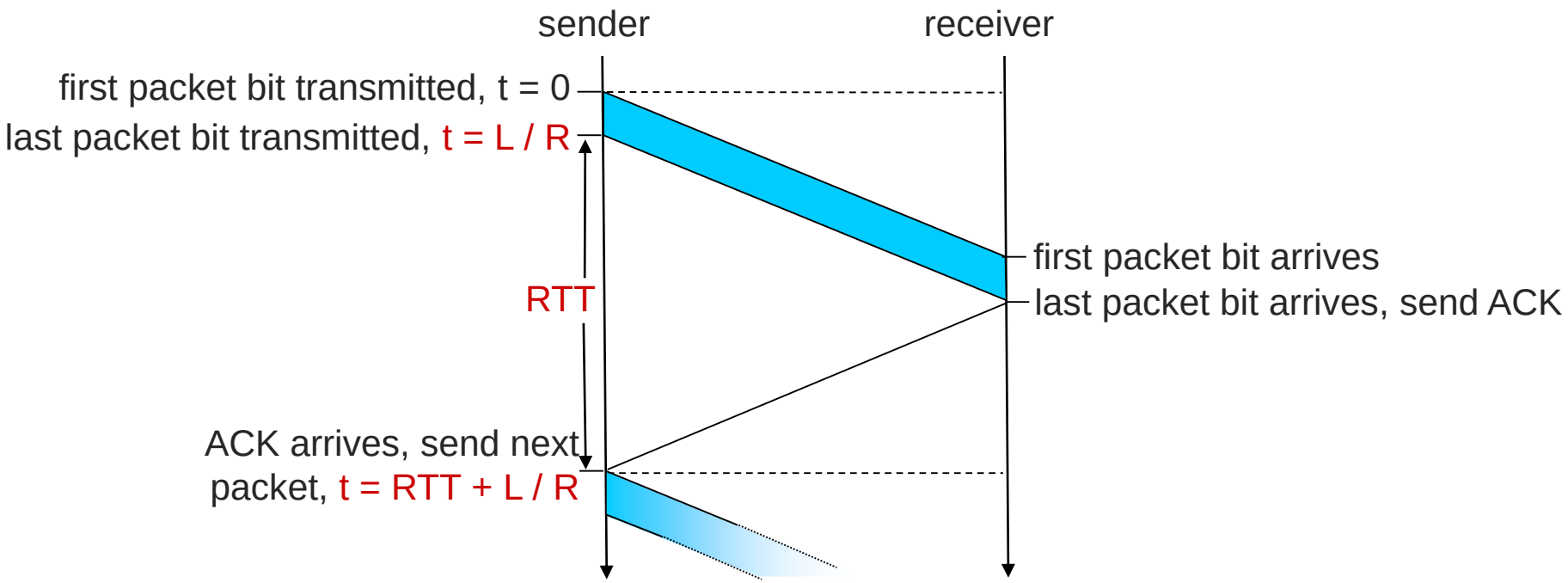
(c) ACK loss



(d) premature timeout/ delayed ACK

Stop-and-wait performance

- Transmission parameters: R=1 Mbps link, 15 ms prop. delay, L=8000 bit packet

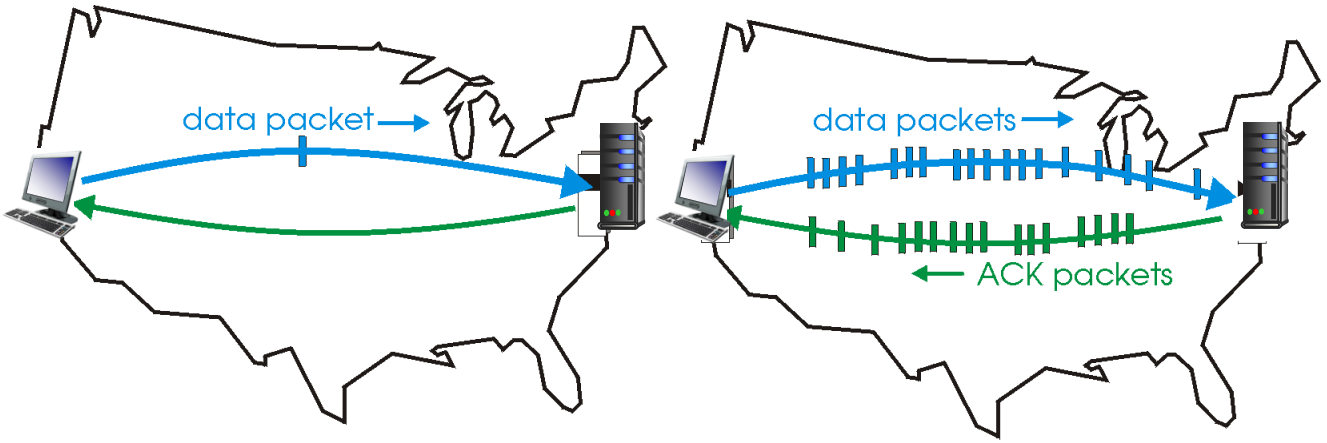


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{8\ [ms]}{30+8\ [ms]} = 0.21$$

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

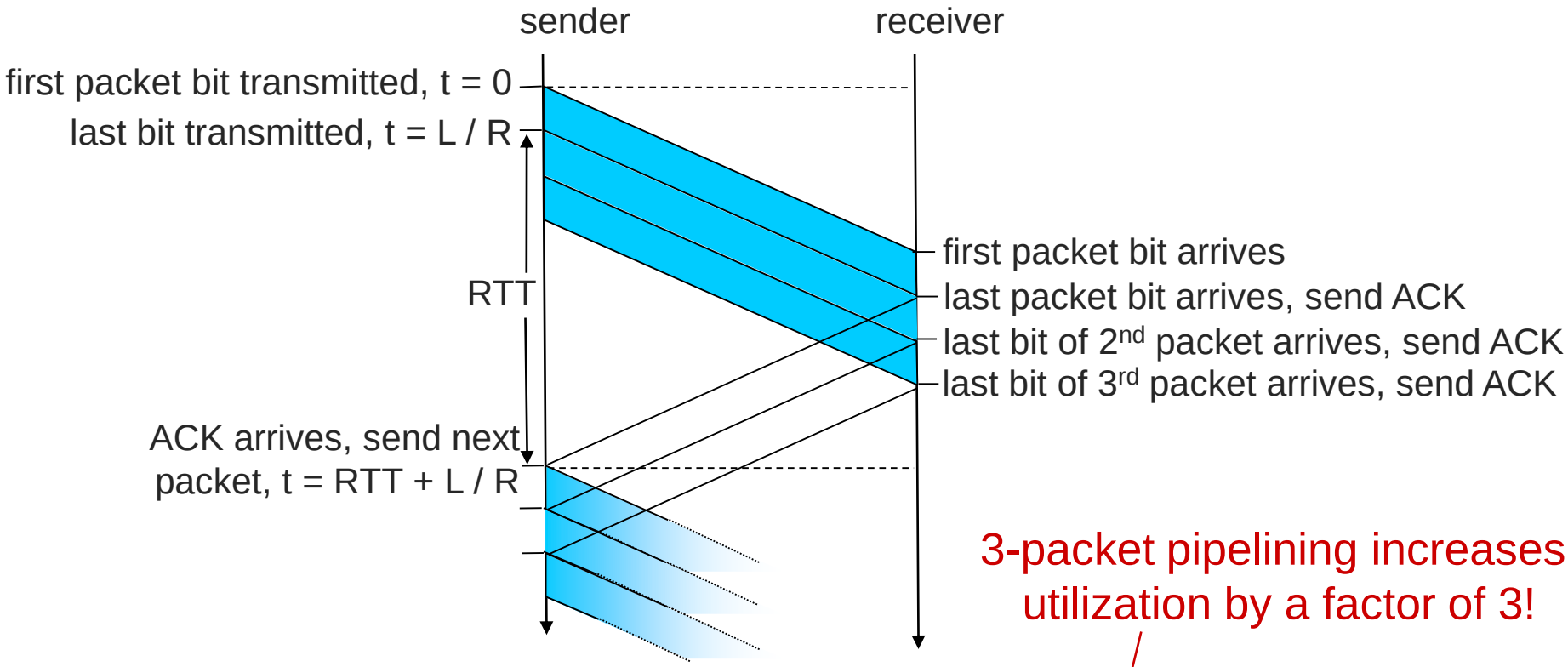


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

Pipelining: increased utilization



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

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{24\ [ms]}{30+8\ [ms]} = 0.63$$

Pipelined protocols: overview

Go-back-N:

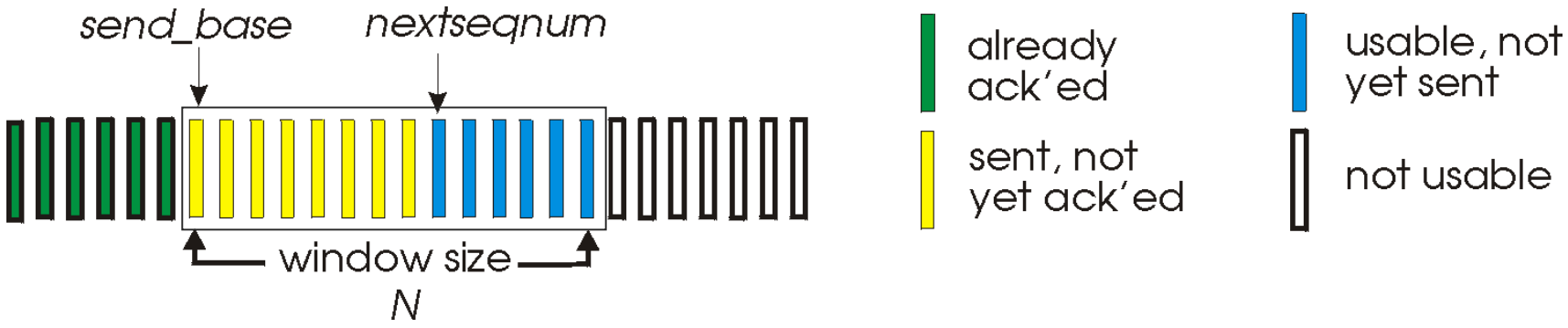
- *sender can have up to N unacked packets in pipeline (window size)*
- *receiver only sends cumulative ack*
  - doesn't ack packet if there's a gap
  - discards all packet received after missing one
- *sender has timer for oldest unacked packet*
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- *sender can have up to N unacked packets in pipeline (window size)*
- *receiver sends individual ack for each packet*
  - keeps in the buffer all packets received after missing one
- *sender maintains timer for each unacked packet*
  - when timer expires, retransmit only that unacked packet

Go-Back-N: sender

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

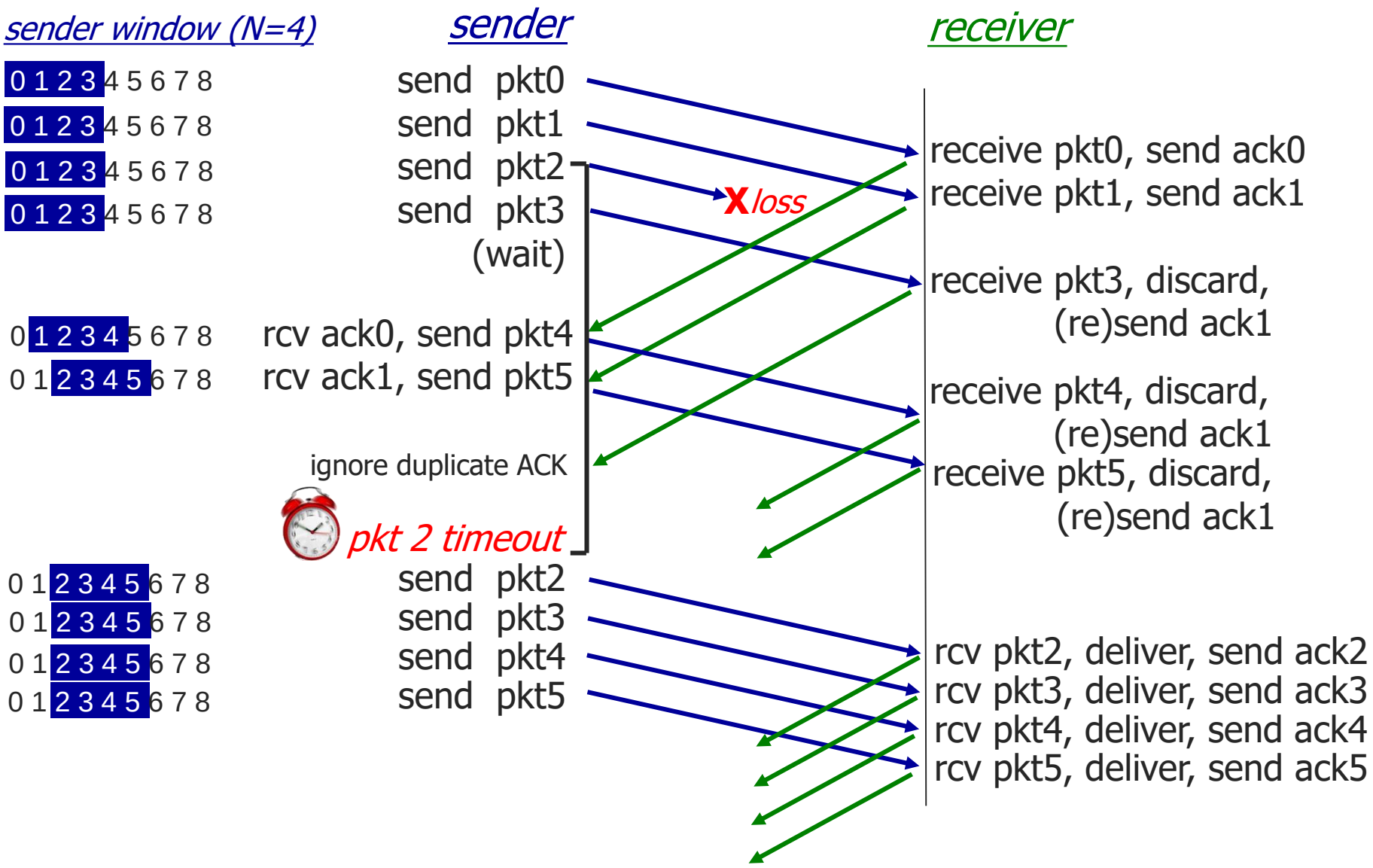


- *ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- *timer for oldest in-flight pkt*
- *timeout(n): retransmit packet n and all higher seq # pkts in window*

Go-Back-N: receiver

- *In-order pkt: send ACK for the packet seq #, determine the next expected packet*
  - need only remember expectedseqnum
- *out-of-order pkt:*
  - discard (don’t buffer): no receiver buffering!
  - re-ACK pkt with highest in-order seq #
  - may generate duplicate ACKs

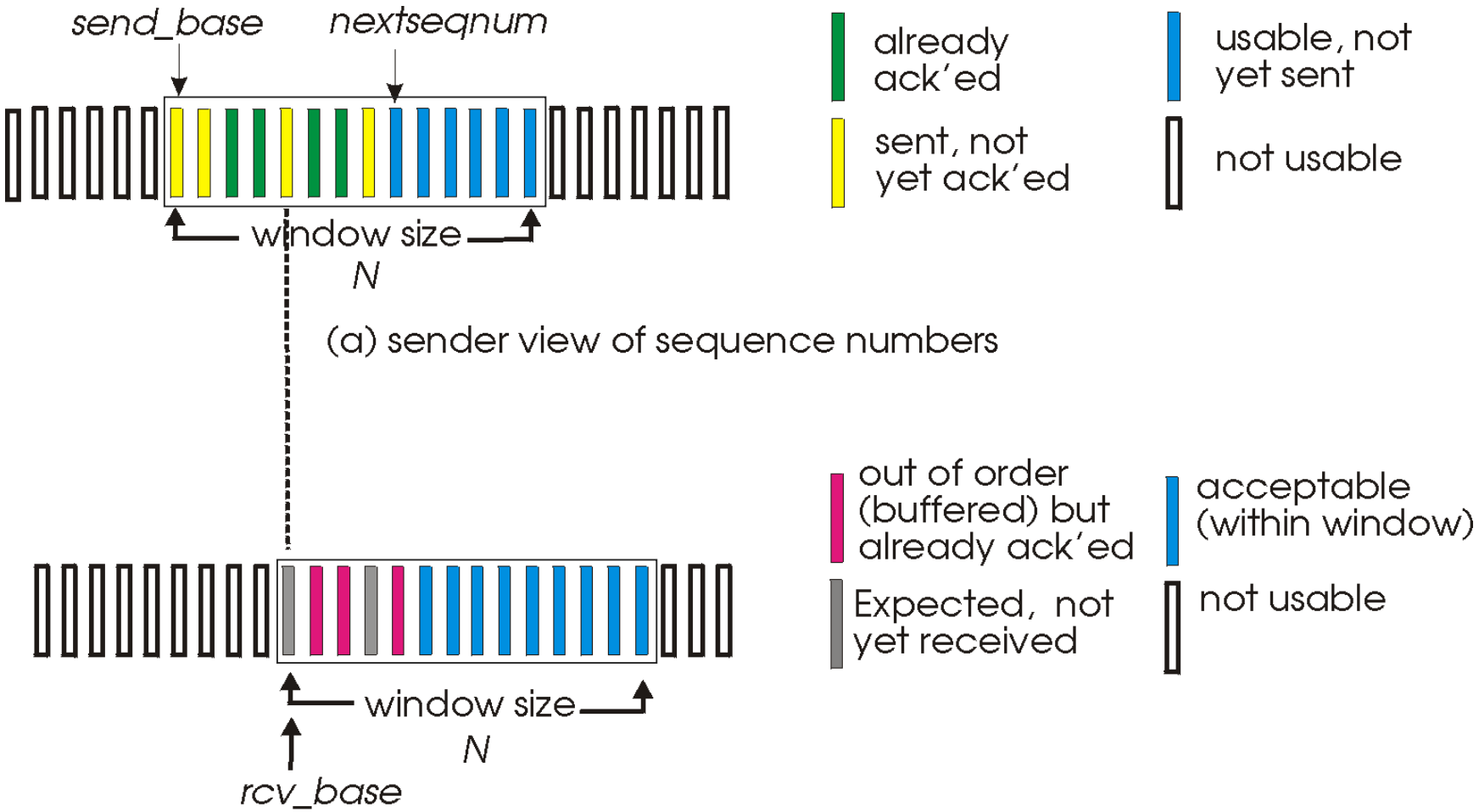
GBN in action



Selective repeat

- *„window" of up to N, consecutive unack'ed pkts allowed*
- *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 sets timer for each unACKed pkt
- *complicated implementation of sender and receiver*
  - multiply timers at sender
  - requires receiver buffering
  - packets reordering at receiver

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,nextseqnum-1]:

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

Selective repeat: receiver

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

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

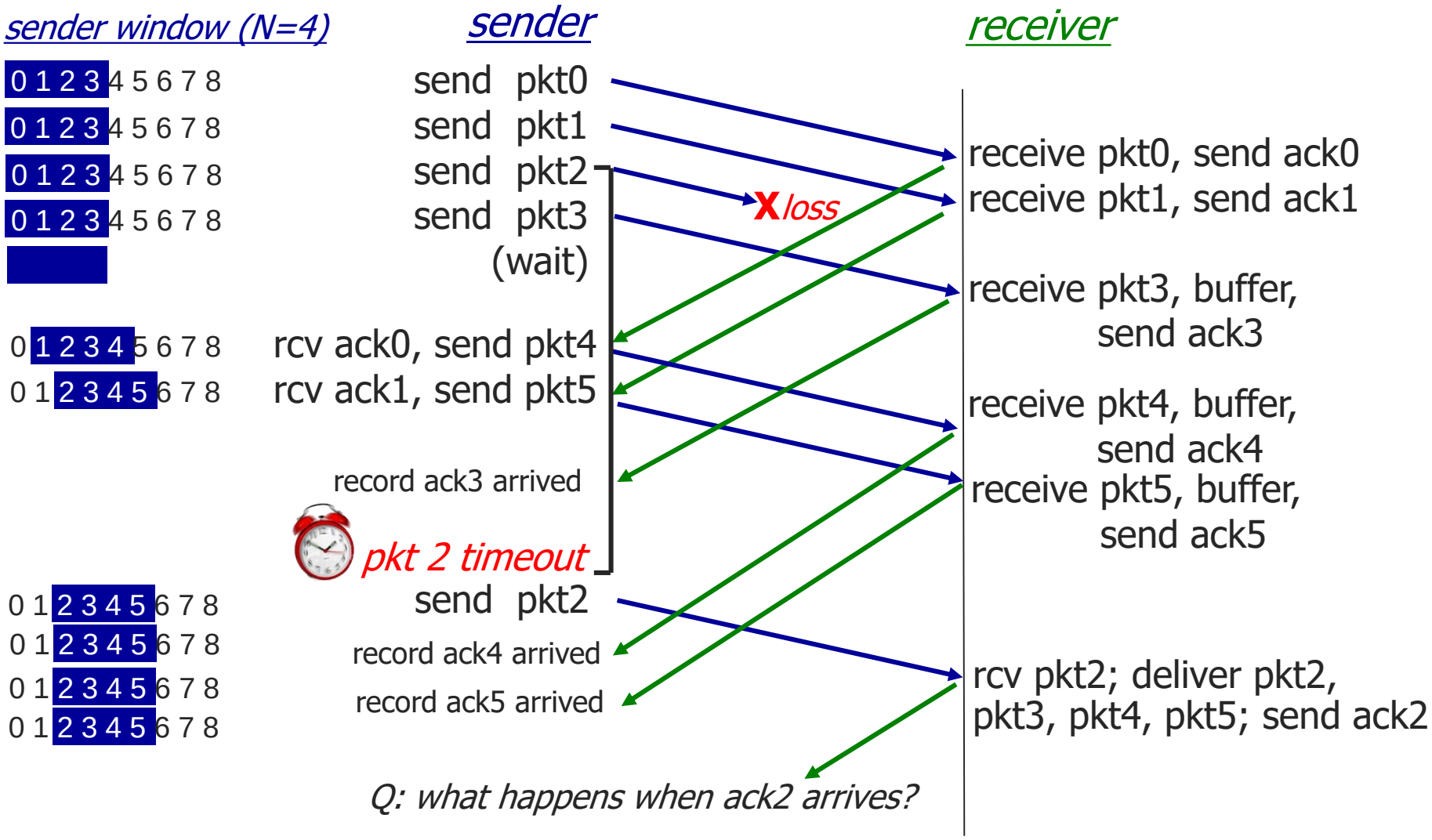
pkt n in [rcvbase-N,rcvbase-1]

- *ACK(n)*

otherwise:

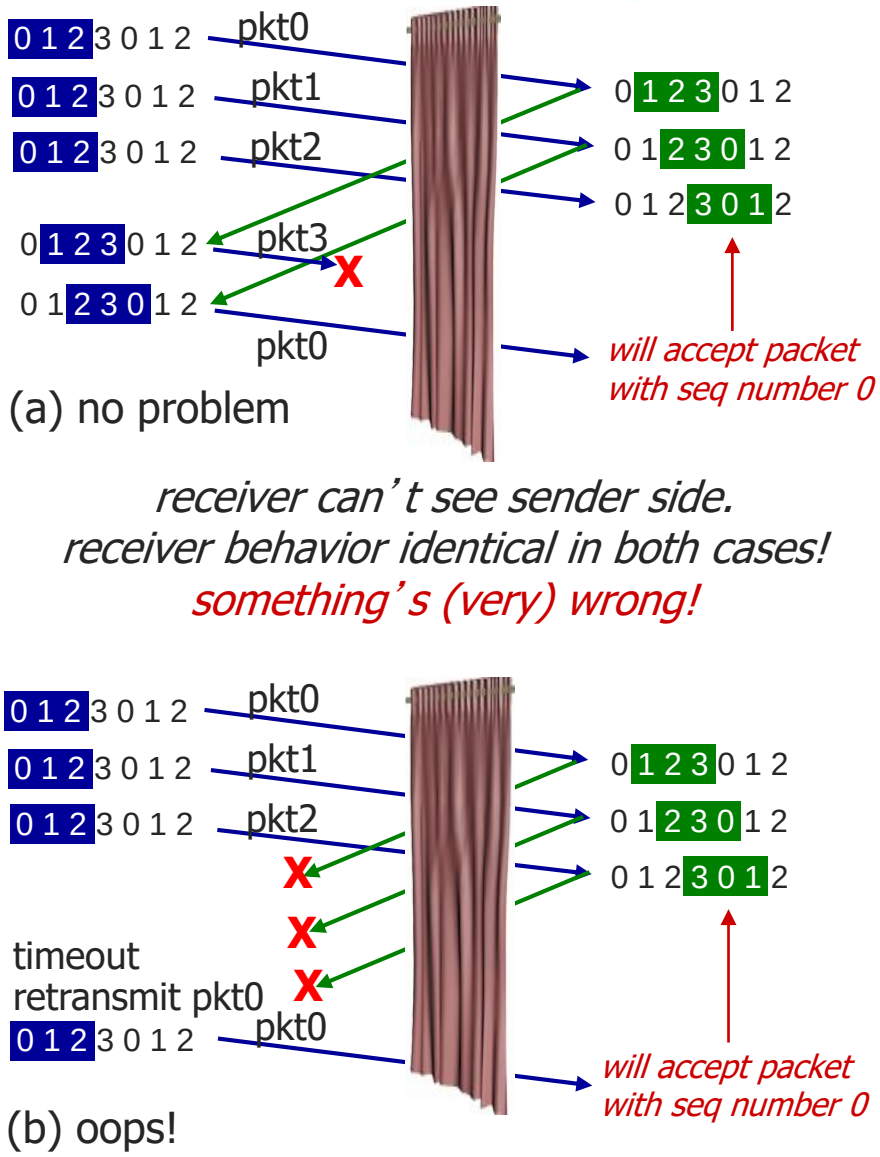
- *ignore*

Selective repeat in action



Selective repeat: dilemma

- *example:*
  - seq #'s: 0, 1, 2, 3
  - window size=3
  - receiver sees no difference in two scenarios!
  - duplicate data accepted as new in (b)
- *Q: what relationship between seq # size and window size to avoid problem in (b)?*



receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!

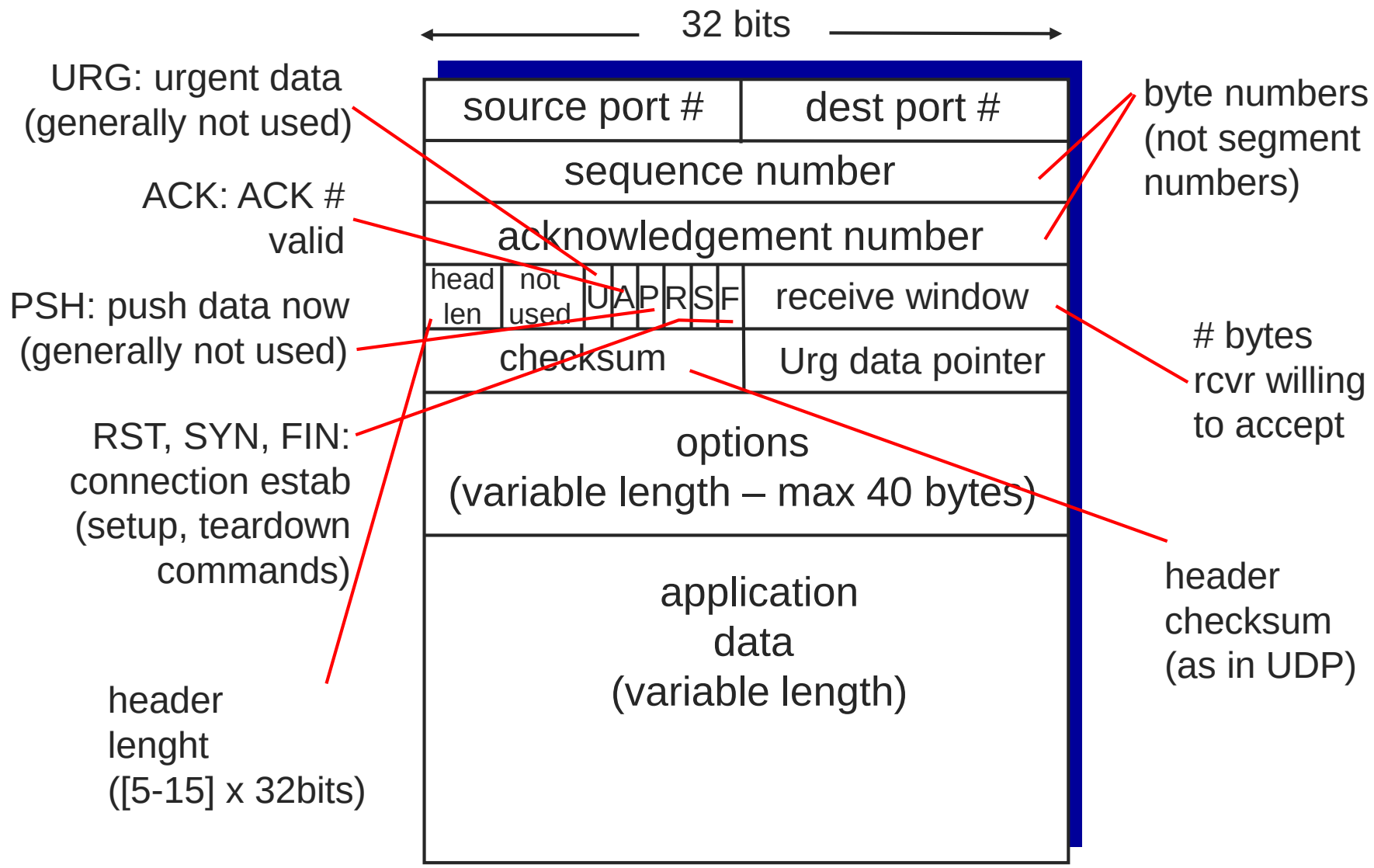
outline

- *transport-layer*
  - services
  - multiplexing and demultiplexing
  - Internet sockets (API)
- *connectionless transport: UDP*
- *principles of reliable data transfer*
- *connection-oriented transport: TCP*
  - segment structure
  - connection management
  - reliable data transfer
  - flow control
- *principles of congestion control*
- *TCP congestion control*

TCP protocol

- *connection-oriented:*
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- *point-to-point:*
  - one sender, one receiver
- *full duplex:*
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- *reliable, in-order data delivery:*
  - user data is treated as stream of bytes (no “message boundaries”)
- *flow control:*
  - sender will not overwhelm receiver
- *congestion control:*
  - TCP congestion control mechanism dynamically sets window size

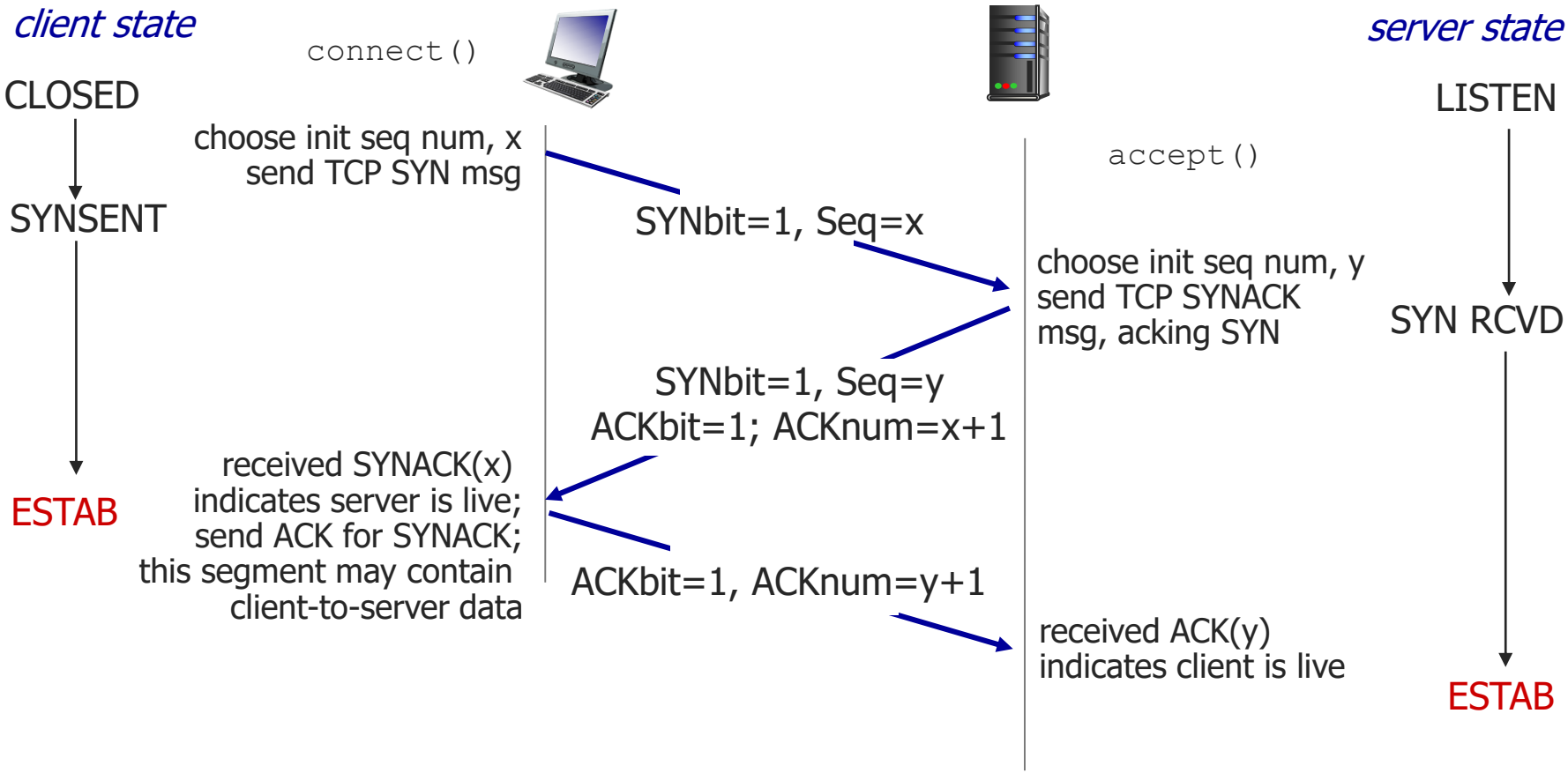
TCP segment structure



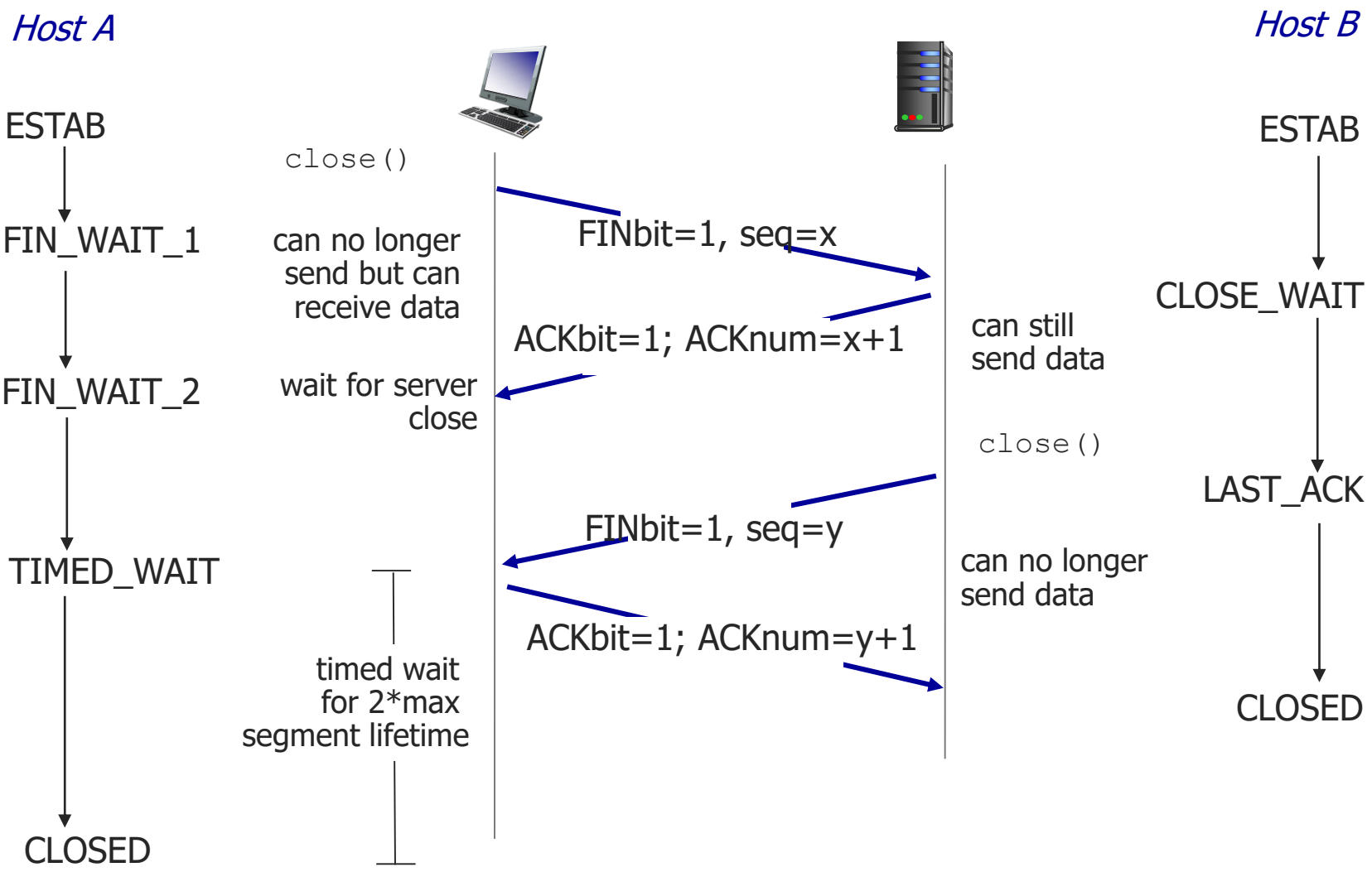
TCP segment

- *TCP flags*
  - URG – segment contains „urgent” data at specified position (sequence number+Urgent pointer), urgent data can be then processed immediately
  - ACK – segment contains valid acknowledgement number
  - PUSH – segment contains data that shuld be delivered to application immediately, all buffered data will be delivered to application
  - RST–connection request is rejected (no listener on specified port)
  - SYN – request to establish TCP connection and synchronize the sequence numbers
  - FIN – request to close the TCP connection

TCP: 3-way handshake

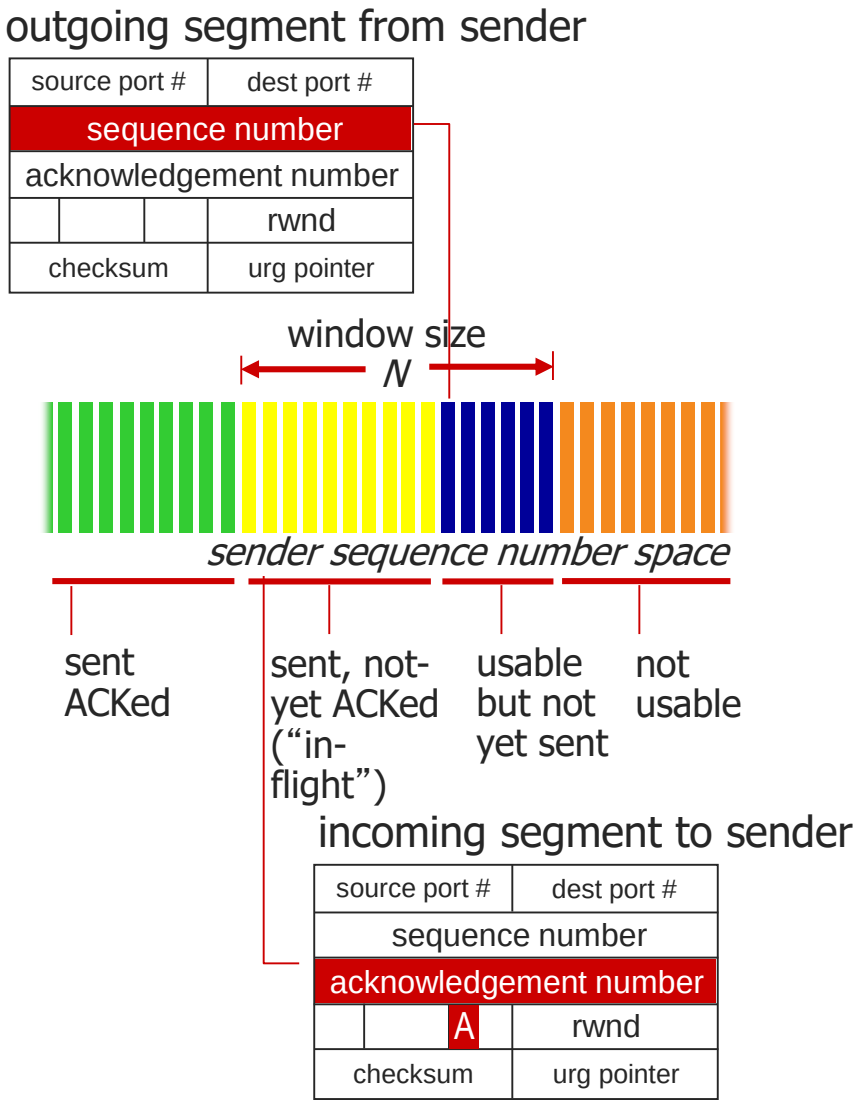


TCP: closing a connection



TCP window

- *TCP window cwnd:*
  - number of bytes that can be send without ACK
  - variable parameter
  - controlled by the congestion & flow control algorithms
- *sequence number:*
  - byte stream “number” of first byte in segment’s data
- *acknowledgement number:*
  - seq # of next byte expected by the receiver
  - cumulative ACK



Basic 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

ack rcvd:

- *if ack acknowledges previously unacked segments*
  - update what is known to be ACKed
  - restart timer if there are still unacked segments
- *duplicate ACK*

timeout:

- *retransmit segments*
- *restart timer*

Delayed ACK

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Start delayed ACK timer. Wait up to 500ms for next segment. If no next segment (timer expires) 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 , clear delayed ACK timer
DelayACK timer expiration	Send ACK for pending segment immidiatelly
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 reliable data transfer

- *TCP creates reliable service on top of IP's unreliable service*
  - cumulative acks
  - single retransmission timer
- *Retransmissions triggered by:*
  - timeout events
  - duplicate acks (fast retransmit)
    - single packet is retransmitted (we do not follow Go_Back_N scheme)

TCP 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*

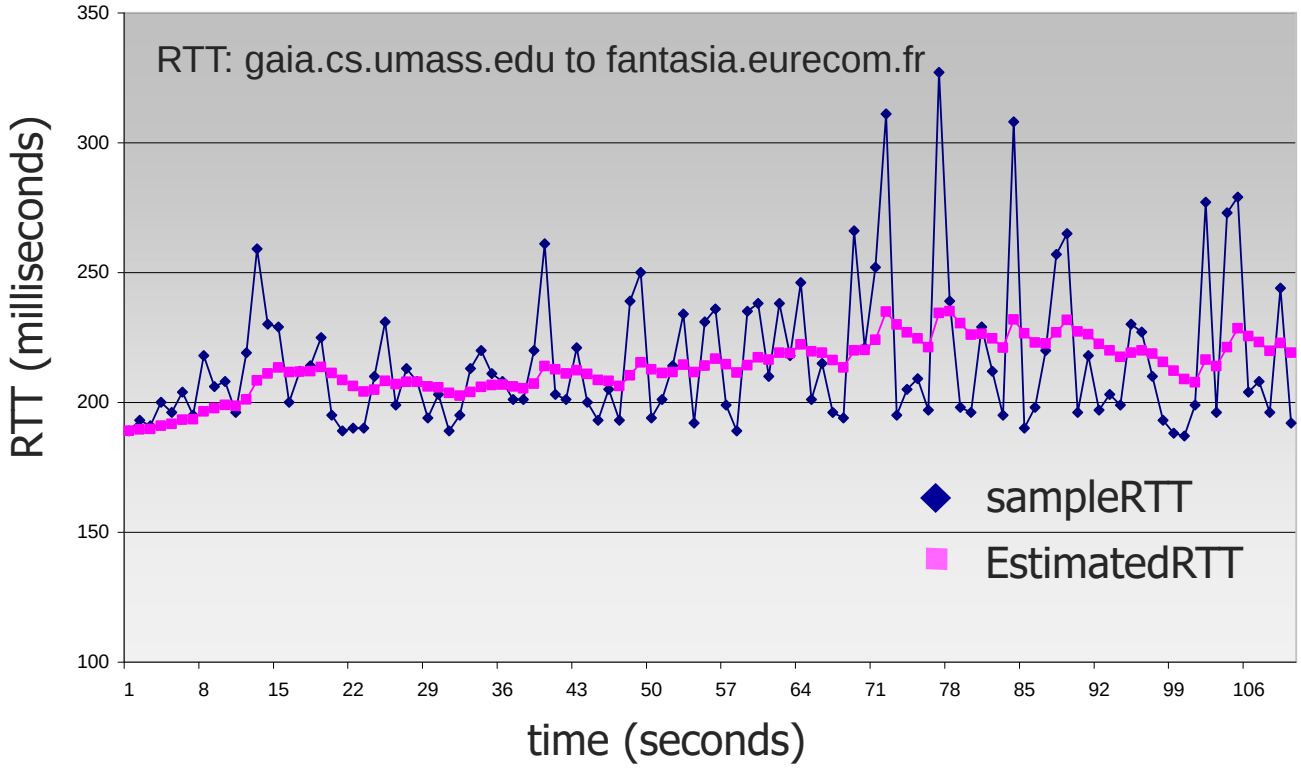
Q: how to estimate RTT?

- *SampleRTT: measured time from segment transmission until ACK receipt*
  - ignore retransmissions (Karn's algorithm)
- *SampleRTT will vary, want estimated RTT "smoother"*
  - average several recent measurements, not just use current SampleRTT

TCP timeout

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

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



TCP timeout

- *timeout interval:*
  - **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- *estimate SampleRTT deviation from EstimatedRTT:*

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

(typically, $\beta = 0.25$)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

estimated RTT

"safety margin"

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 be many duplicate ACKs.

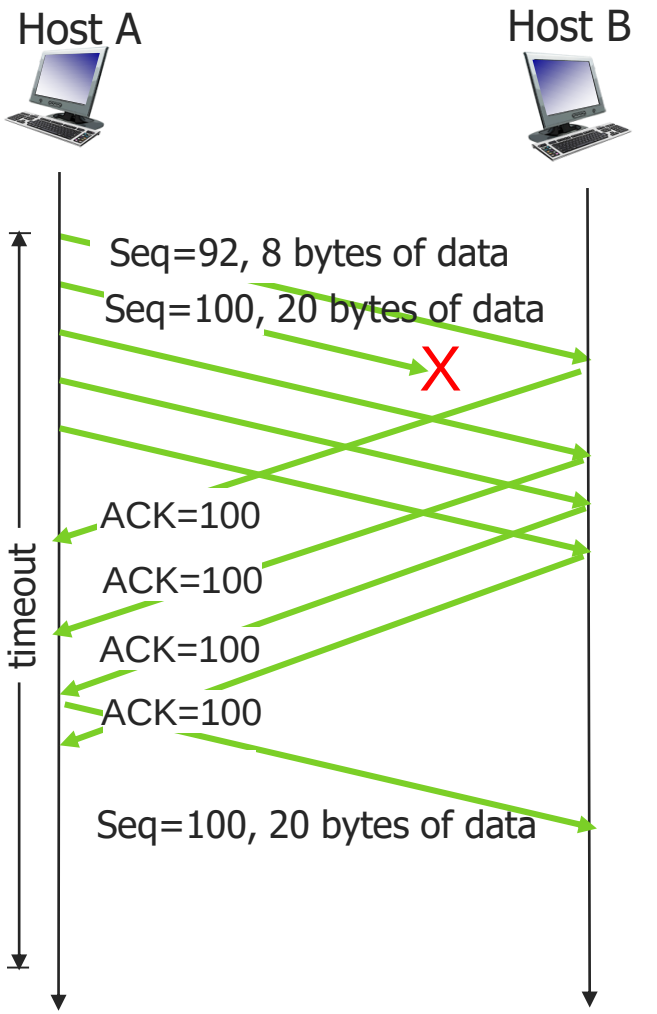
TCP fast retransmit

if sender receives 3 ACKs for same data

("triple duplicate ACKs"), resend unacked segment with smallest seq #

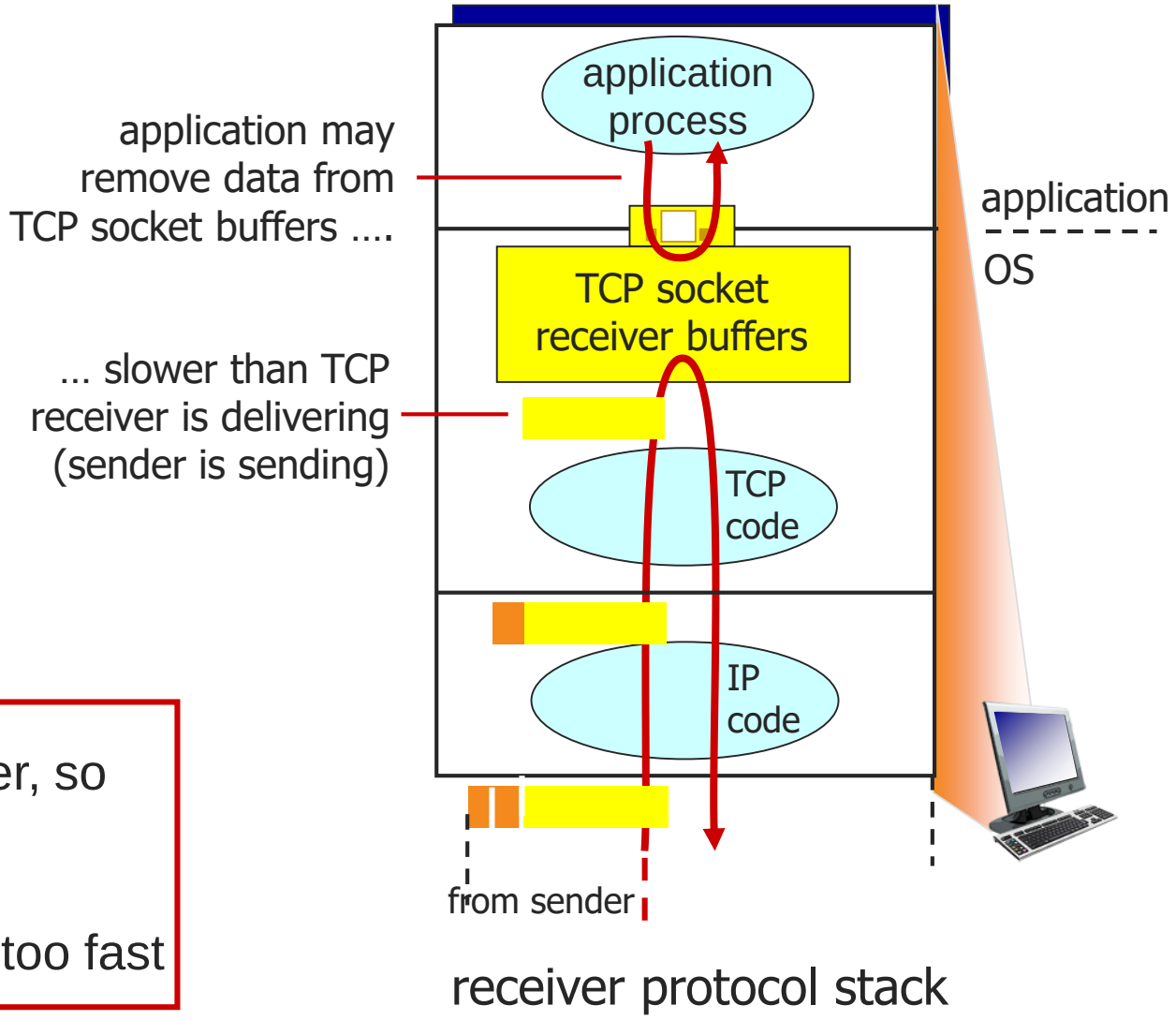
- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP flow control

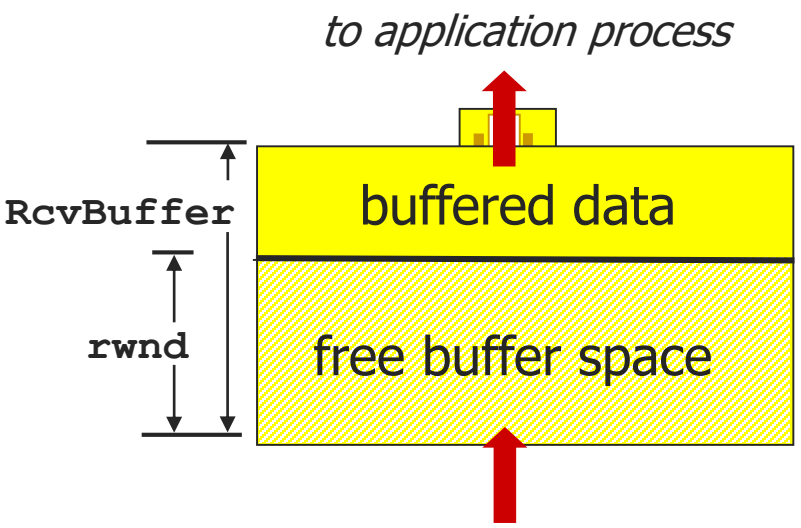


flow control

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
  - many operating systems autoadjust **RcvBuffer (DRS)**
- *sender limits amount of unacked ("in-flight") data to receiver's* ***rwnd*** *value*
- *guarantees receive buffer will not overflow*



receiver-side buffering

principles of congestion control

congestion:

- *informally: "too many sources sending too much data too fast for network to handle"*
- *different from flow control!*
- *manifestations:*
  - lost packets (buffer overflow at routers)
  - long delays (queuing in router buffers)

approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:

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

network-assisted congestion control:

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

case study: ATM ABR congestion control

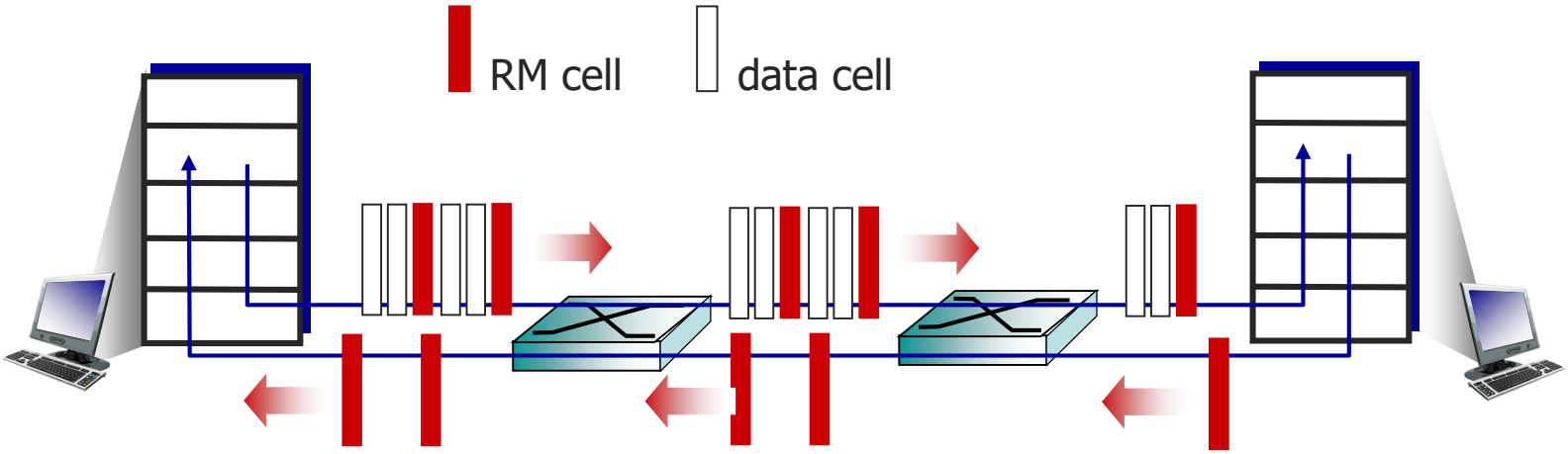
ABR: available bit rate:

- *"elastic service"*
- *if sender's path "underloaded":*
  - sender should use available bandwidth
- *if sender's path congested:*
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- *sent by sender, interspersed with data cells*
- *bits in RM cell set by switches ("network-assisted")*
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- *RM cells returned to sender by receiver, with bits intact*

case study: ATM ABR congestion control

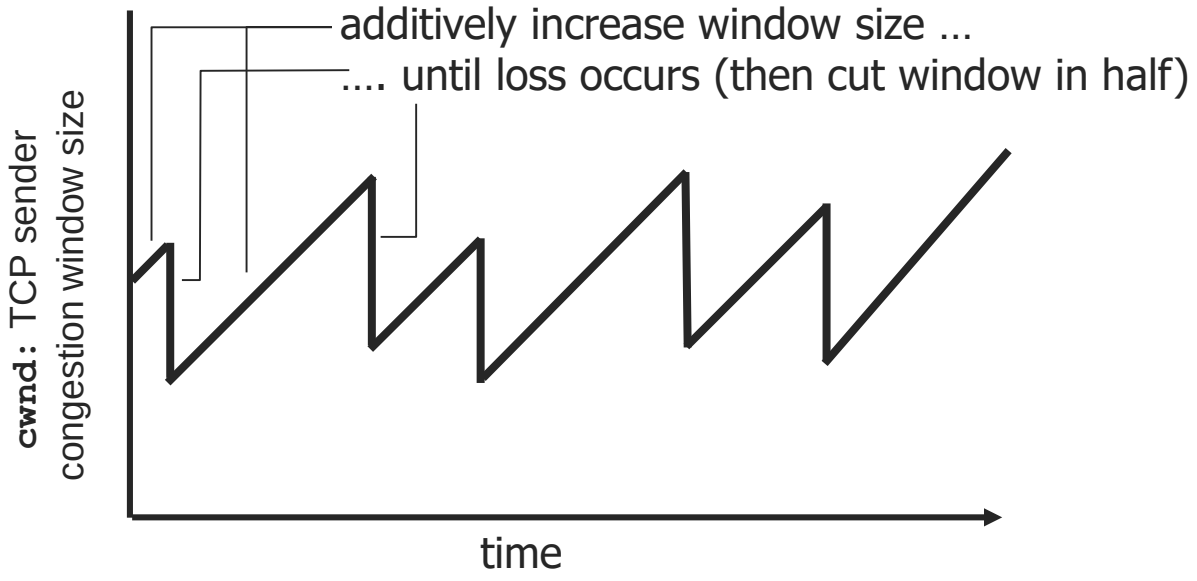


- *two-byte ER (explicit rate) field in RM cell*
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- *EFCI bit in data cells: set to 1 in congested switch*
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

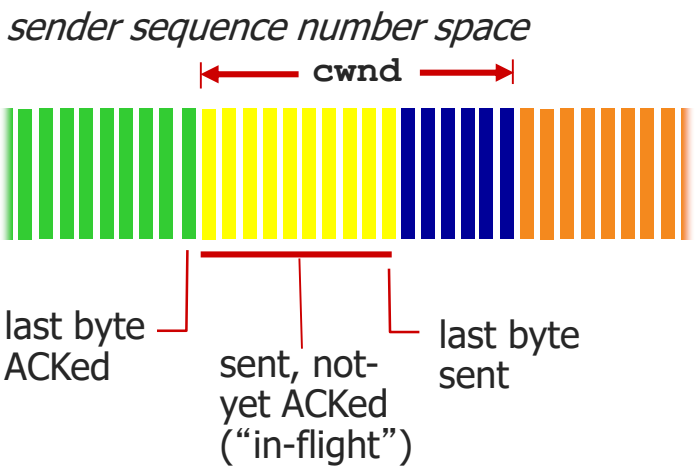
TCP congestion control: AIMD

- *approach: sender increases transmission rate (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: details



- *sender limits transmission:*

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

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

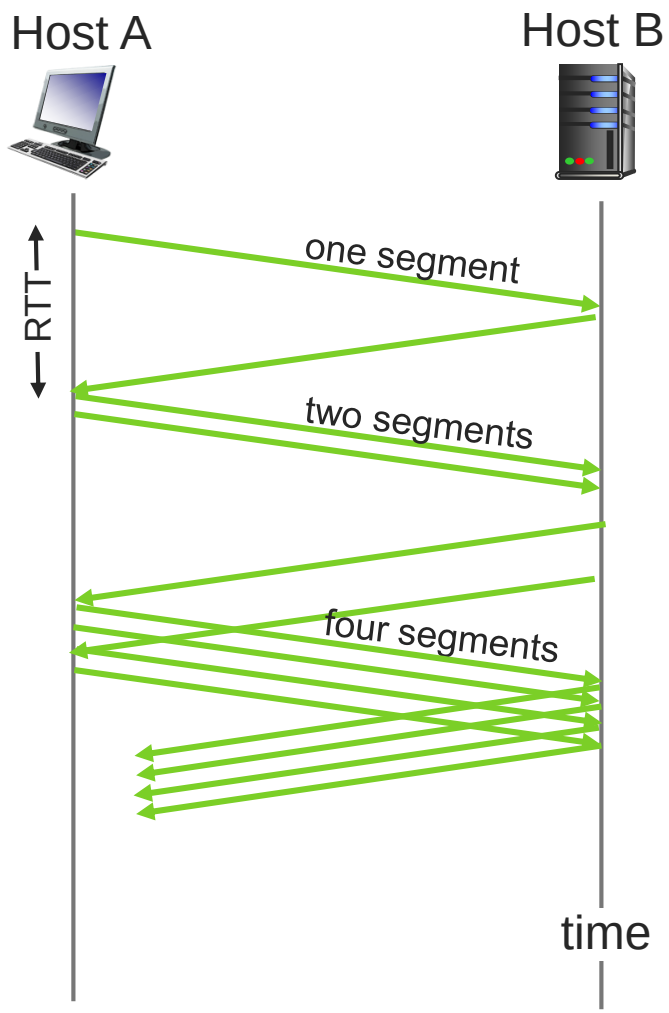
TCP sending rate:

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

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

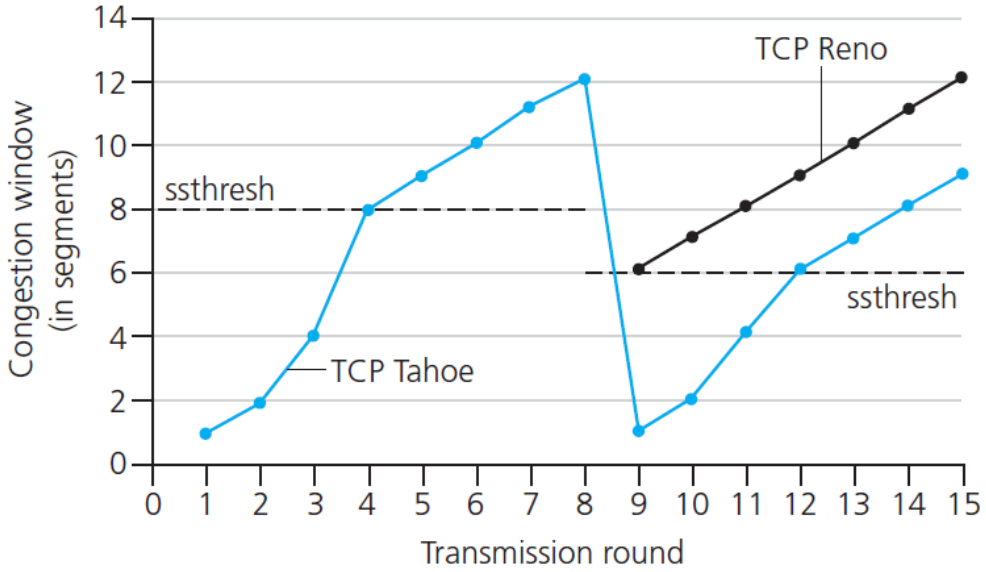
TCP Slow Start

- *when connection begins, increase rate exponentially until first loss event:*
  - initially cwnd = 1 MSS
    - in new TCP implementation the initail cwnd is set to larger value e.g. 10 MSS
  - double cwnd every RTT
  - done by incrementing cwnd for every ACK received
- *summary: initial rate is slow but ramps up exponentially fast*



Switching from slow start to congestion avoidance

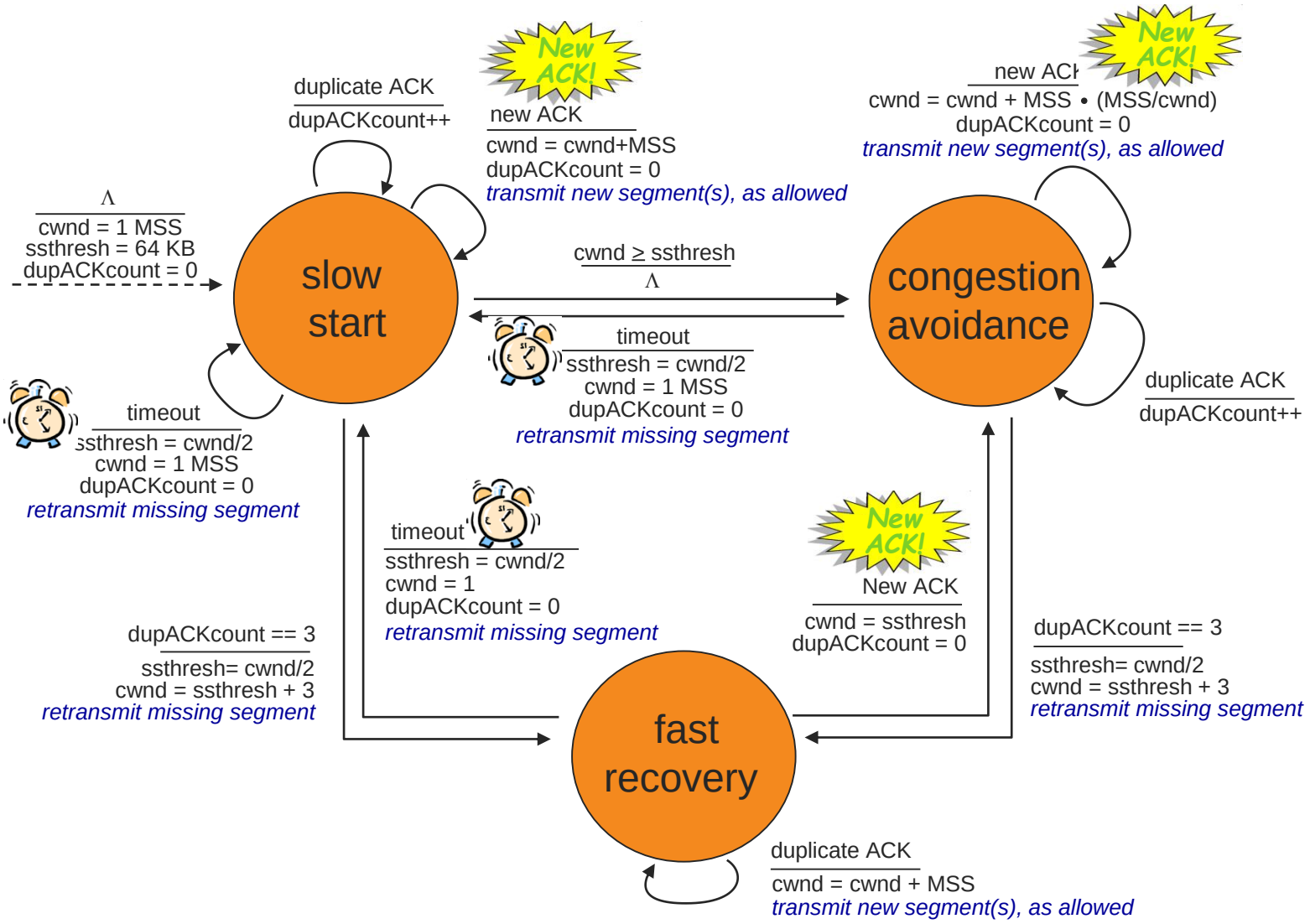
- *Congestion avoidance=Additive increase*
- *ssthresh variable controls the switching between slow start and congestion avoidance*
  - when cwnd reaches ssthresh TCP switches to congestion avoidance
- *On loss event, ssthresh is set to 1/2 of cwnd just before loss event*
- *On timeout TCP enters slow start*



TCP fast retransmit&fast recovery

- *after 3 dup-ack TCP retransmits lost segment (fast retransmit)*
- *reception of dup-ack segments informs sender that the network is still capable of delivering packets*
  - no need to go into the slow start phase - may limit TCP performance
- *sender enters the fast recovery phase*
  - sender sets the cwnd to ssthresh + 3 segments
  - Sender increases cwnd by 1 for each dup-ack received
  - When the retransmitted segments is acknowledged the sender set cwnd to ssthresh and enters CA

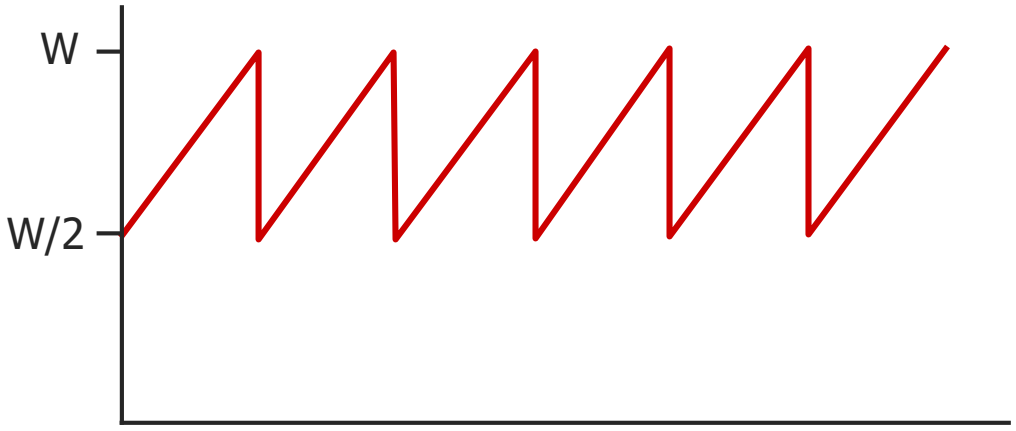
Summary: TCP Congestion Control



TCP throughput

- *avg. 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*
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

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



TCP Futures: TCP over "long, fat pipes"

example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput

- *requires W = 83,333 in-flight segments*
- *throughput in terms of segment loss probability, L [Mathis 1997]:*

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

  - ➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ – a very small loss rate!
- *new versions of TCP for high-speed*