Redes de Comunicação 2023/2024

T03 Transport layer

Jorge Granjal
University of Coimbra

UNIVERSIDADE D COIMBRA

T04
Transport Layer

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

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

3.5 TCP congestion control

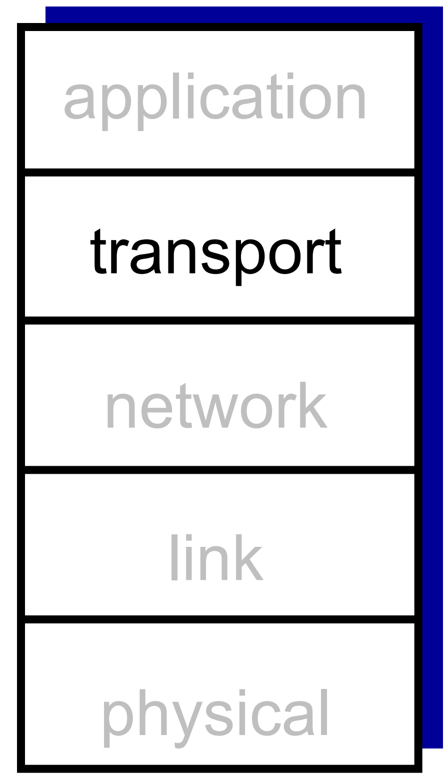
T04
Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

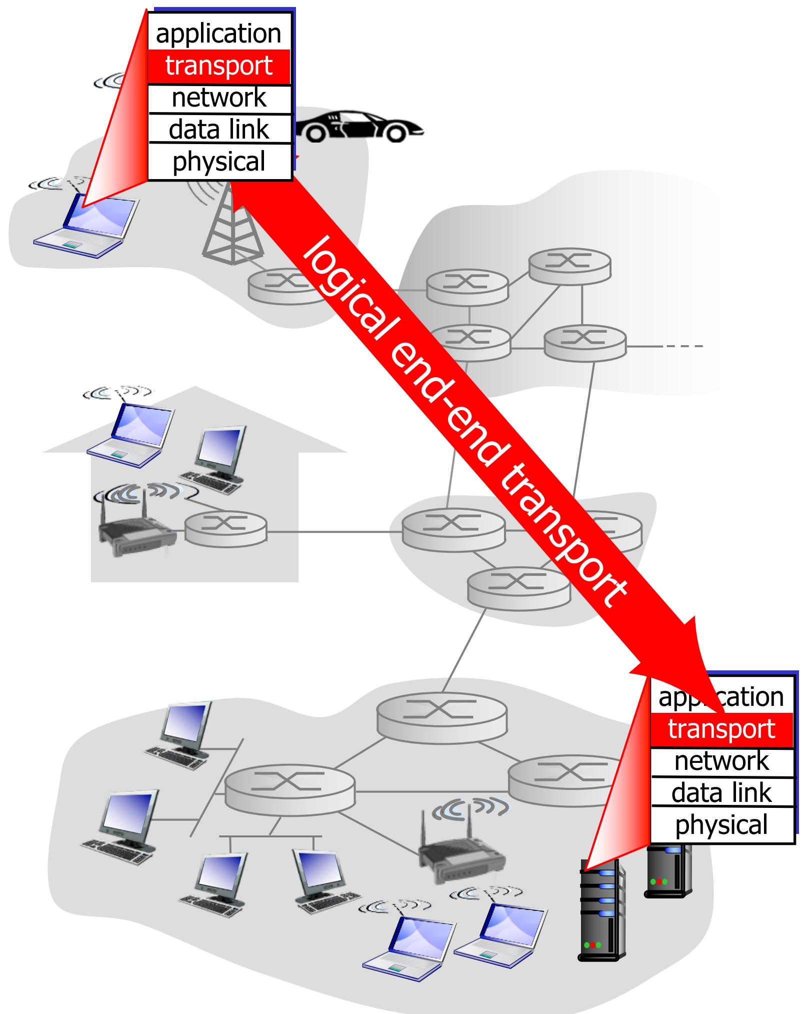
Internet (TCP/IP) protocol stack

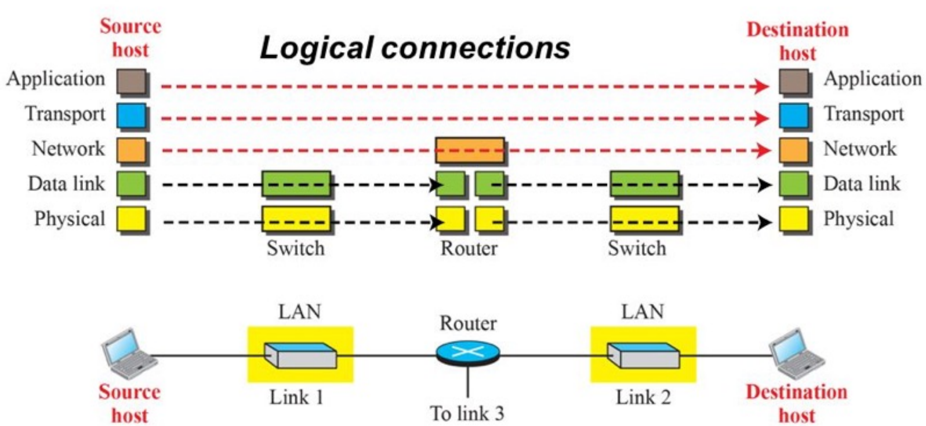
- *application:* supporting network applications
  - FTP, SMTP, HTTP, …
- *transport*: process-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- *physical:* bits "on the wire"



Transport services and protocols

- provide *logical (rather than physical) communication* between app processes running on different hosts
  - *From an application's perspective, it is as if the hosts running the processes were directly connected!*
- application processes use the logical communication provided by the transport layer to exchange messages
  - *Application processes do not have to worry with the details of the physical infrastructure used to carry the messages*

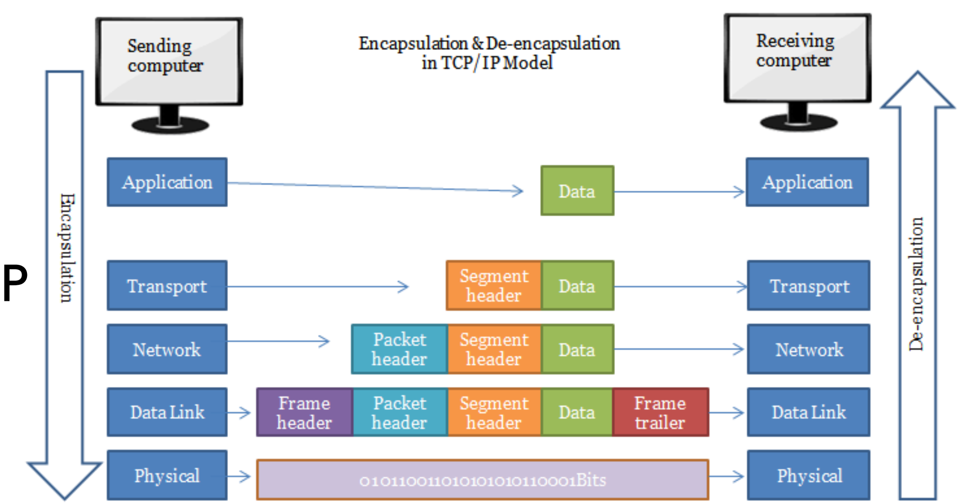




Transport services and protocols

- transport protocols run in end systems (not in network routers)
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
  - network routers act only on the network-layer fields of the datagram
- more than one transport protocol available to apps
  - Internet: TCP and UDP (different services to applications)

TCP, UDP

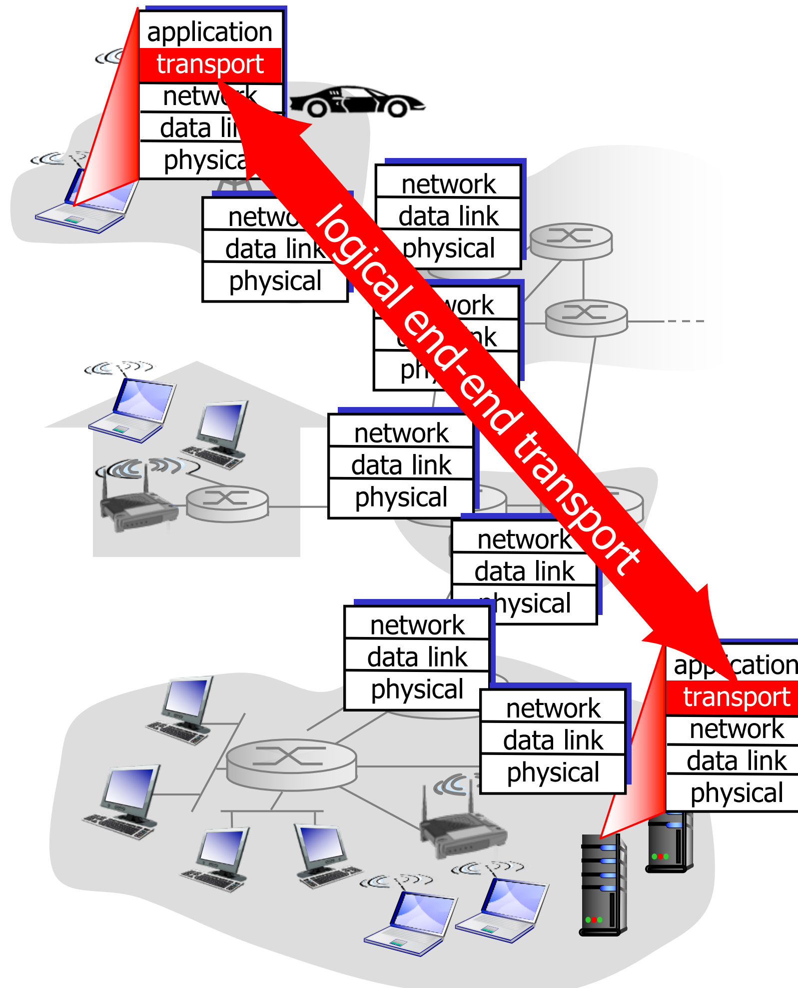


Transport vs. network layer

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

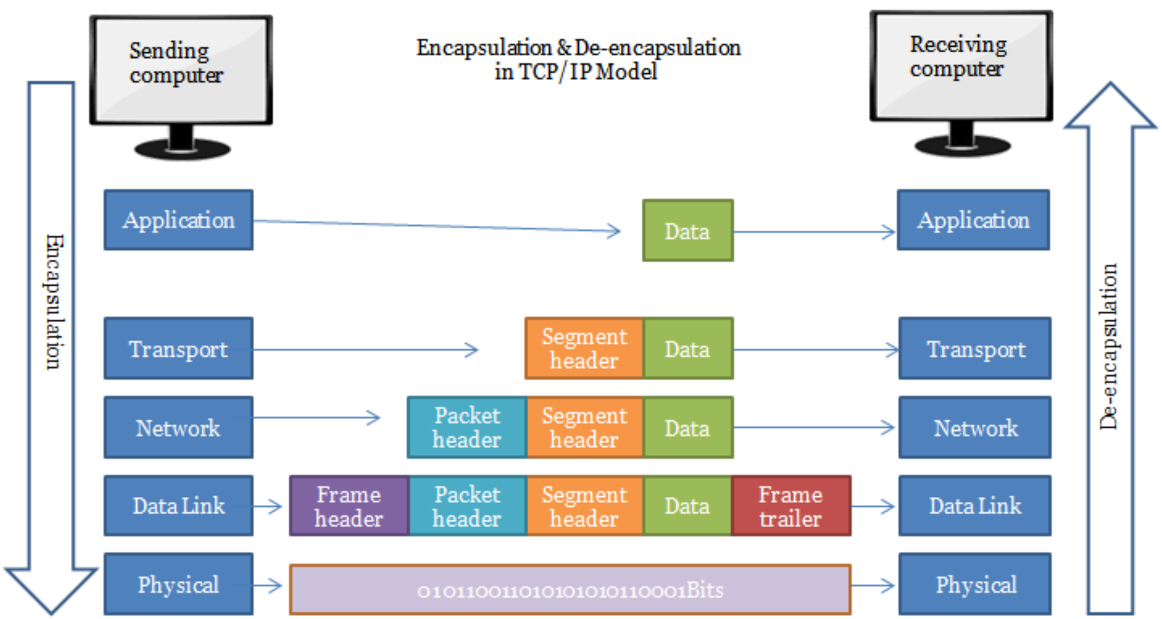
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection-oriented
  - connection setup
- unreliable, unordered delivery: UDP
  - connecionless
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

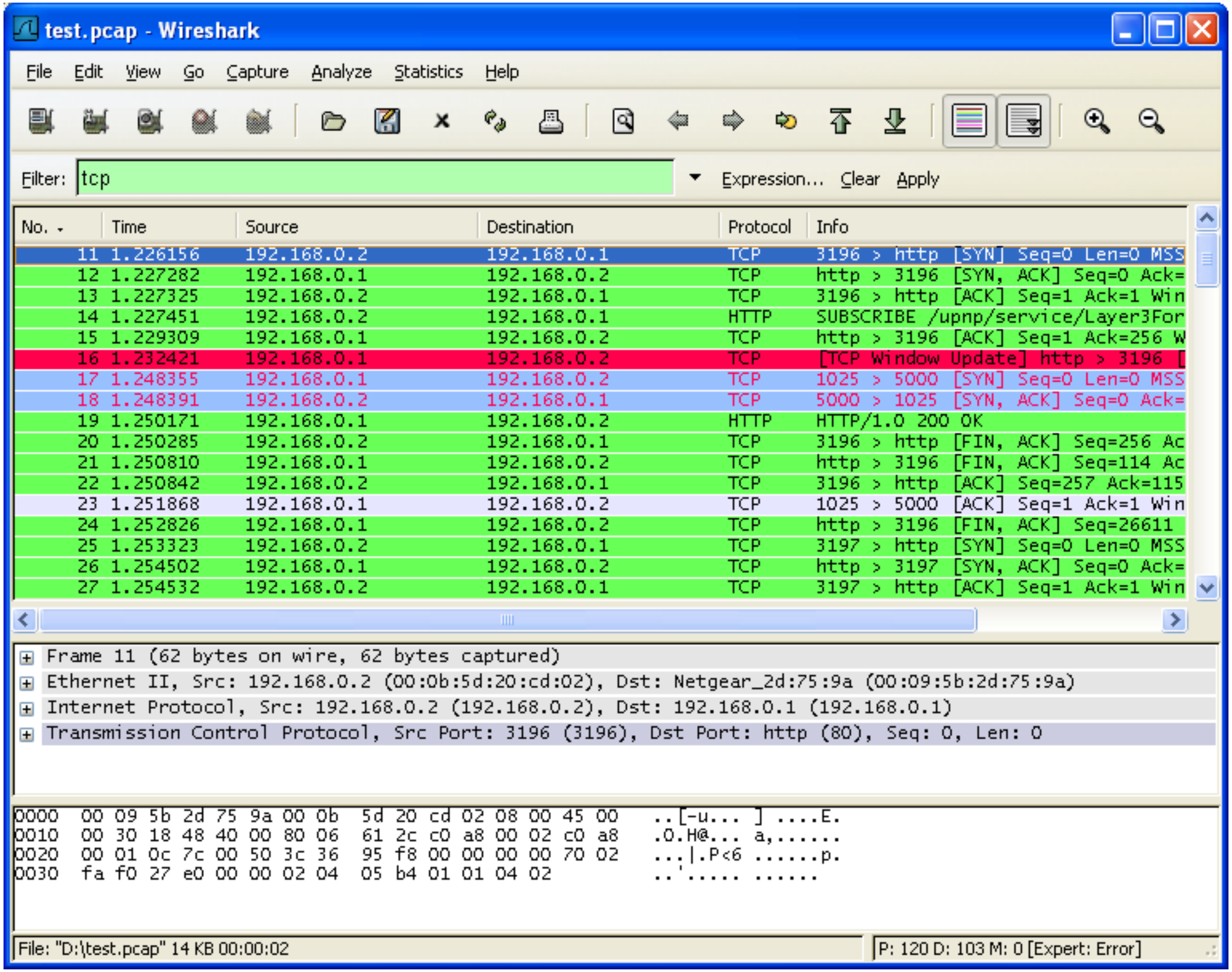


Do not get confused!

- Applications send **messages**
- Transport layer packets are referred to as **segments**
- Network layer (IP) packets are referred to as **datagrams** or **packets**
- Data link packets are referred to as **frames**
- *Although, in the Internet literature (e.g. in RFCs) the term datagram may refer to IP packets but also to UDP packets!*



Details in Wireshark

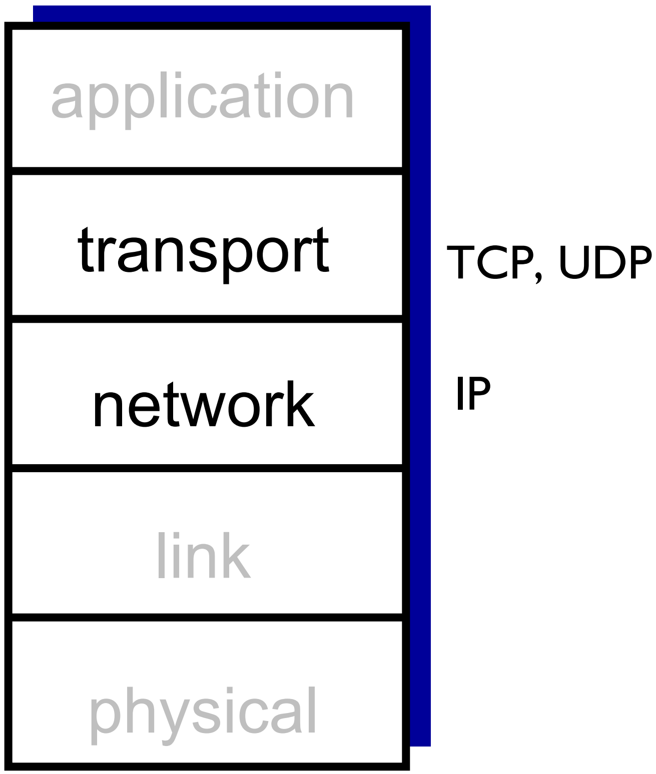


A few words about the network layer

- Network layer has a name: IP (Internet Protocol) layer

- IP service model is "best effort", no guarantees of:
  - segment delivery
  - orderly delivery of segments
  - integrity of the data in segments

- UDP and TCP extend host-to-host delivery of IP to process-to-process delivery of the transport layer:
  - UDP only provides process-to-process data delivery and error checking
  - TCP provides a reliable data transport service between processes



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 connection-oriented transport: TCP

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

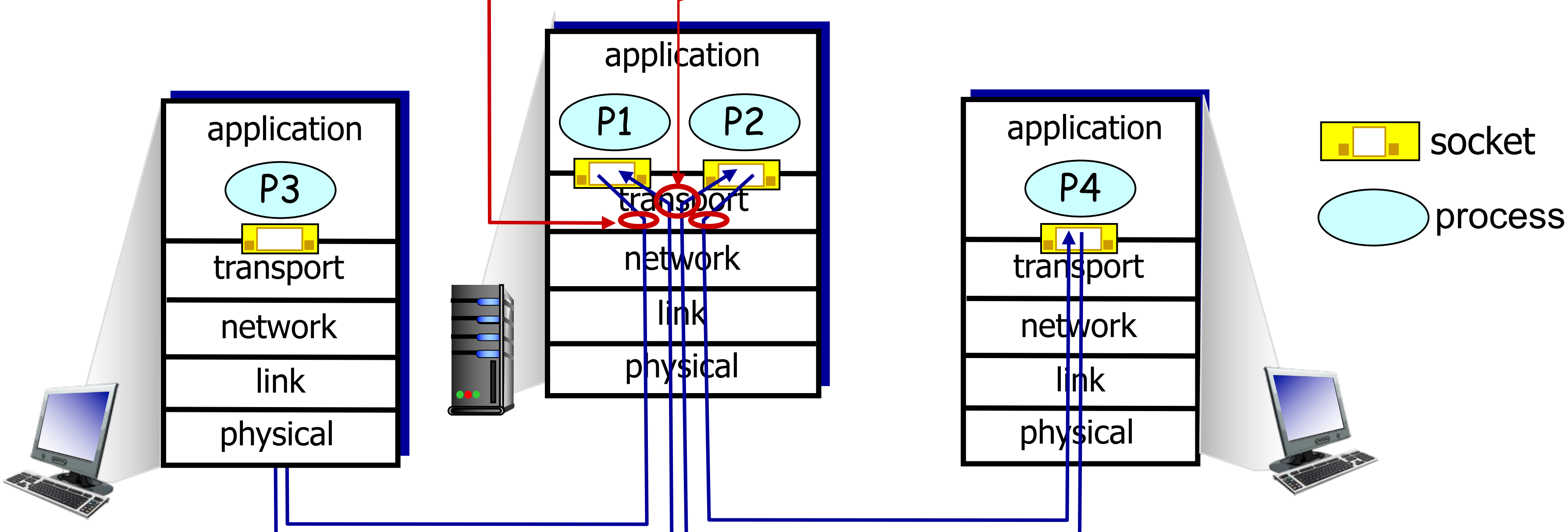
3.5 TCP congestion control

Multiplexing/demultiplexing

A process (which is part of an application) can have one or more sockets, “doors” though which they exchange data with the network

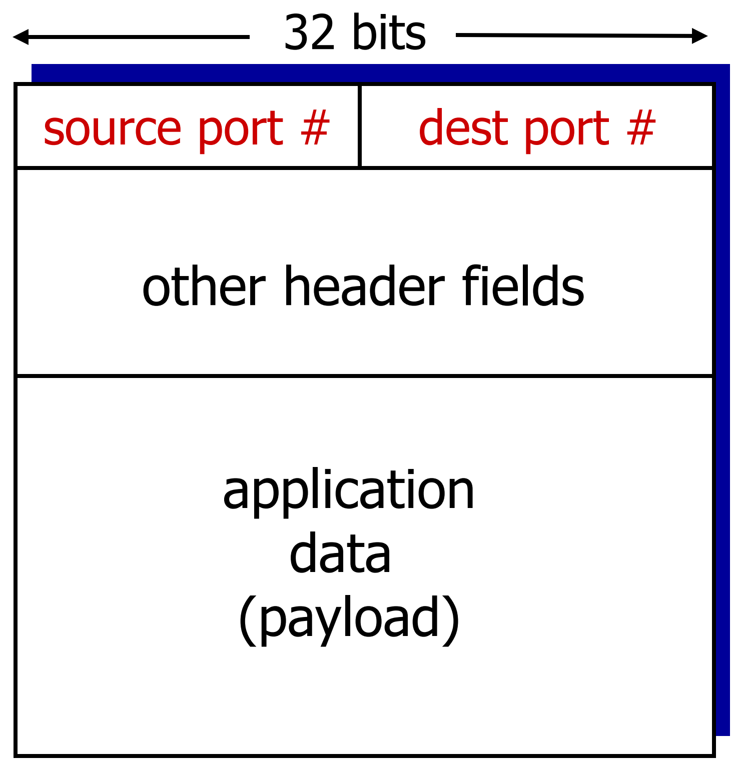
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket (and process)



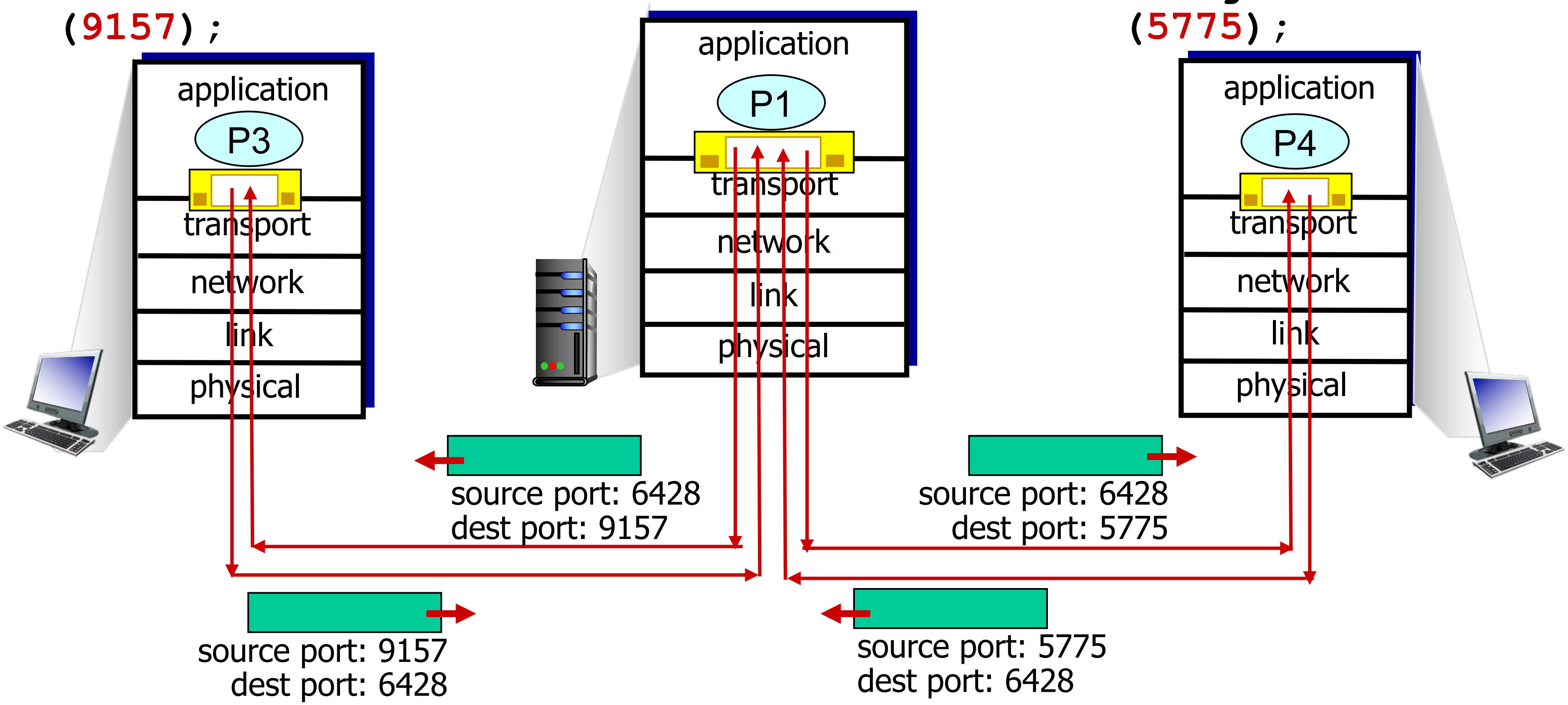
TCP/UDP segment format
(common structure)

Connectionless demux: example

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

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

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

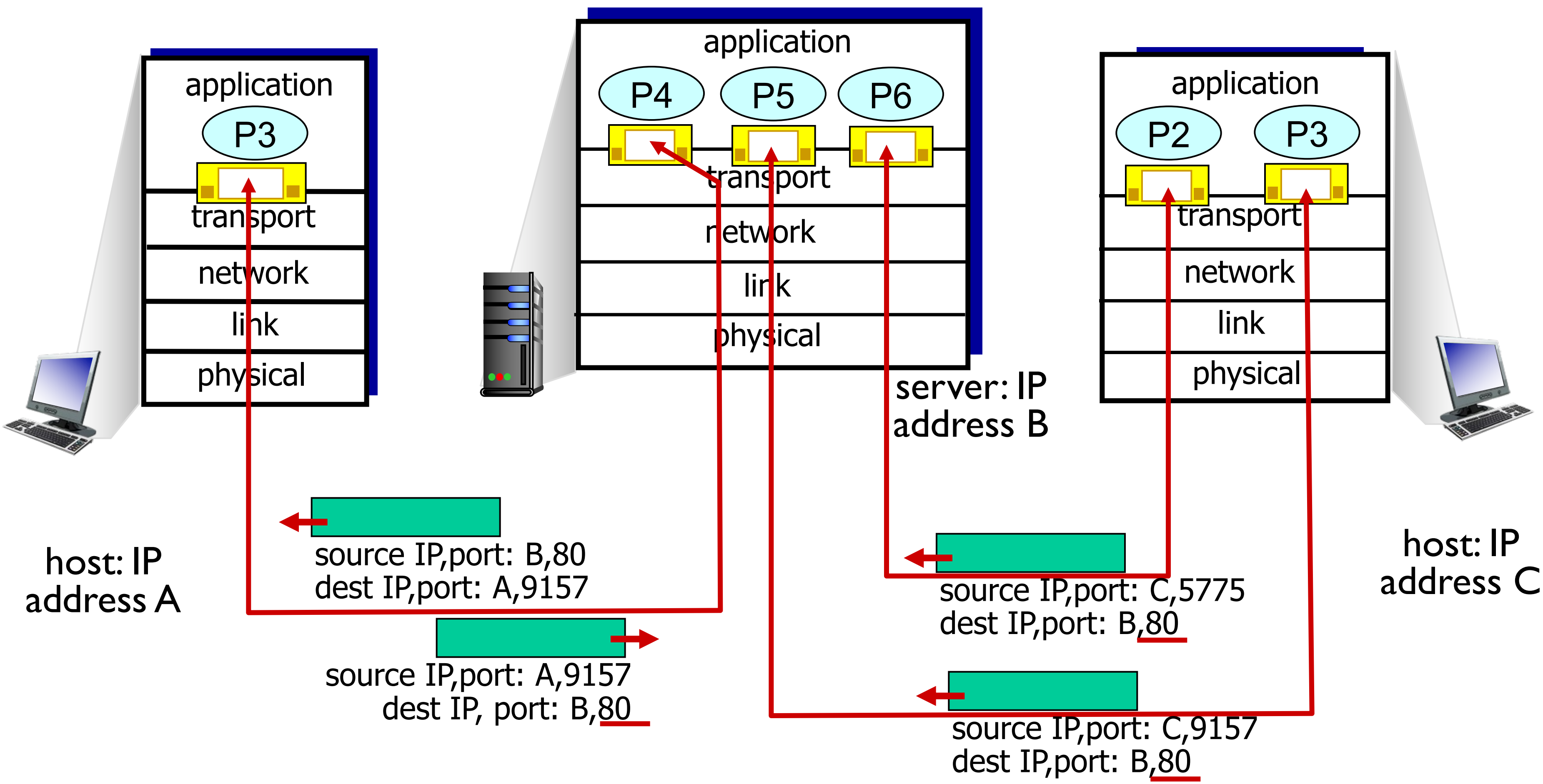


Connection-oriented demux

- UDP socket identified by 2-tupple:
  - dest IP address
  - dest port number
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

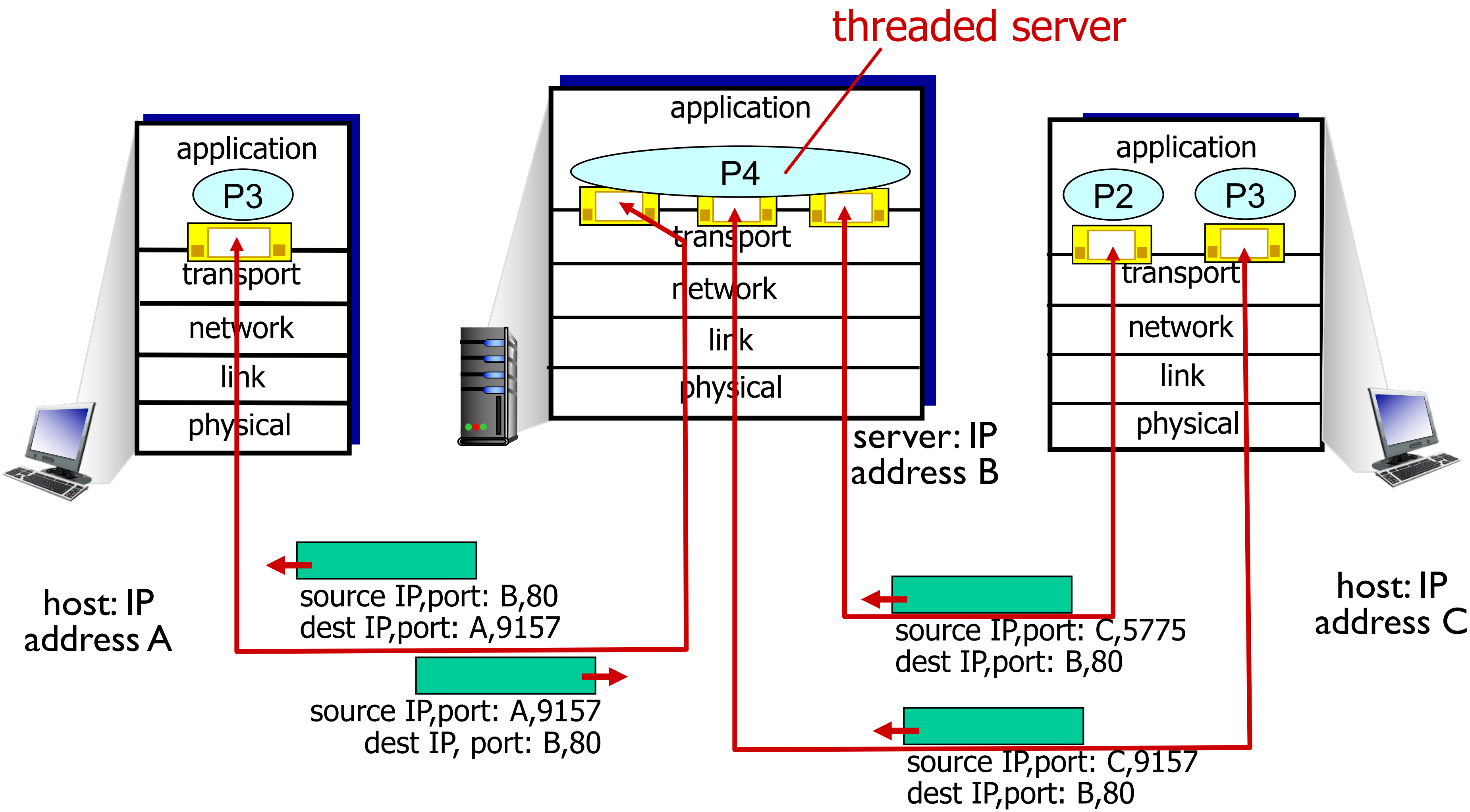
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - example: non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

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

3.5 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!

Why UDP?

Isn't TCP always preferable, since TCP provides a reliable data transfer service, while UDP does not? No!

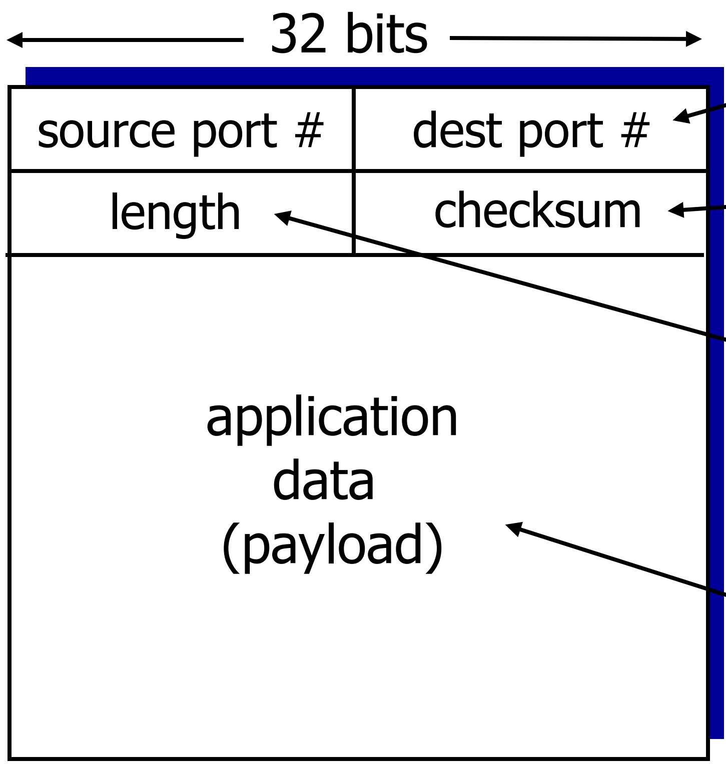
- Many applications are better suited for UDP:
  - Finer application-level control over what data is sent, and when: UDP sends data immediately, TCP implements congestion control
  - Some applications tolerate some loss, while TCP uses retransmission
  - Additional functionality may be implemented at the application
- No connection establishment:
  - No additional delay prior to start communicating
  - Example: DNS uses UDP
- No connection state:
  - Server using UDP may support many more clients than when using TCP
- Small packet header overhead:
  - Less overhead in communications

Why UDP?

- Examples of applications using UDP:
  - RIP updates: updates of routing tables are periodic, thus lost updates will be replaced by more recent information
  - Network management using SNMP: UDP is better in congested networks
  - DNS: IP and name resolutions are faster using UDP, without TCP's connection establishment delays

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

UDP: segment header



UDP segment format

used for demultiplexing

used by receiver to check whether segment has errors

length, in bytes of UDP segment, including header

e.g: DNS query or response, sample of audio stream, etc.

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment (e.g. errors due to noisy links, or while stored in a router)

sender:

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

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value
  - NO - error detected
  - YES - no error detected

Internet checksum: example

example: add two 16-bit integers

```
             1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
             1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
          ------------------------------------
wraparound 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
          ------------------------------------
       sum   1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
  checksum   0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

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

Is error checking in UDP useful?

- IP is supposed to run over just about any layer-2 protocol
- There is no guarantee that all the links between source and destination provide error checking
- Errors may also occur while segment is stored in a router's memory
- Neither link-by-link reliability nor in-memory error detection is guaranteed: UDP provides *error detection at the transport layer, on and end-to-end basis*
- UDP does <u>nothing</u> to recover from an error, segment may be discarded or passed to the application with a warning

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

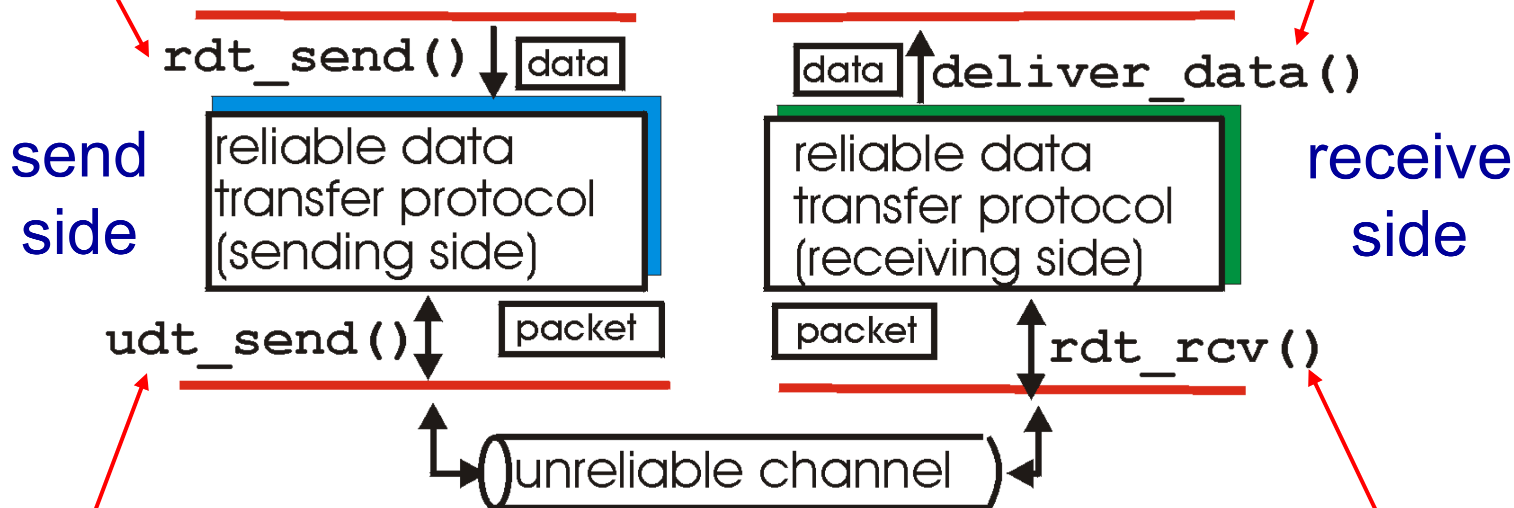
3.4 connection-oriented transport: TCP
- reliable data transfer
- segment structure
- flow control
- connection management

3.5 TCP congestion control

Reliable data transfer: an overview

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - 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
  - Sliding-window protocol

TCP segment structure

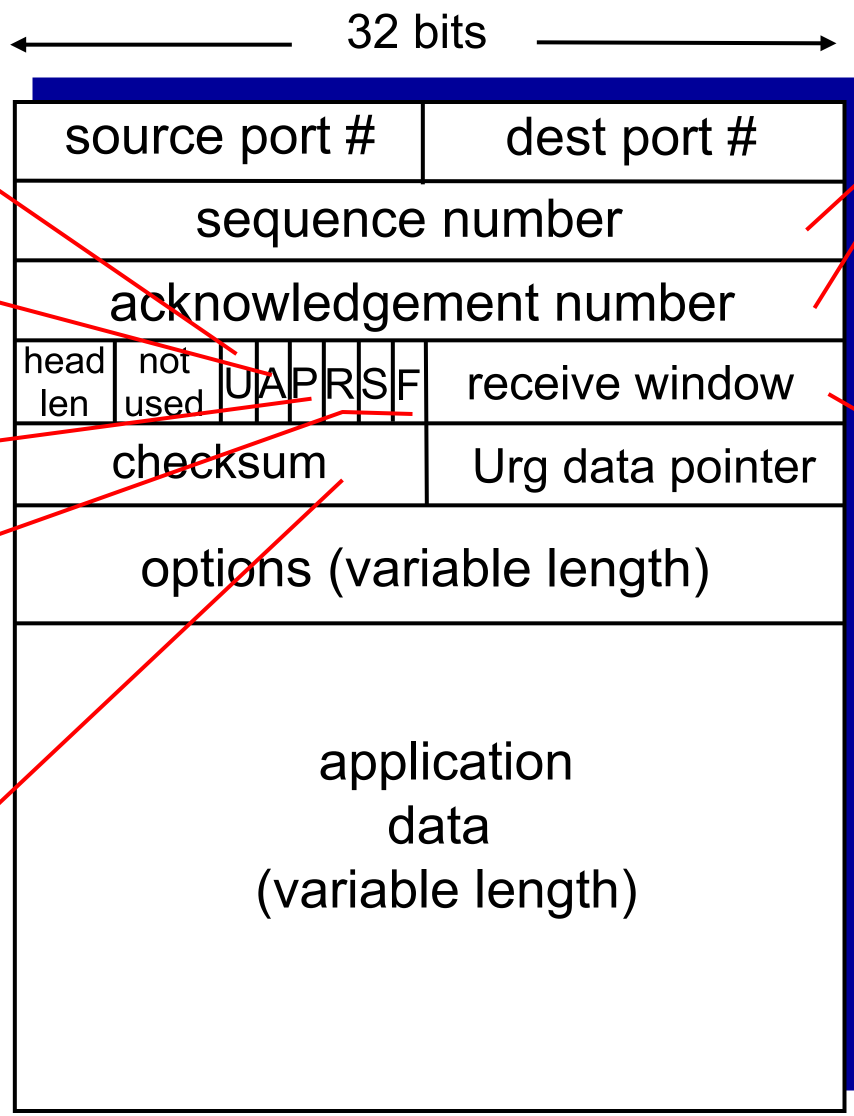
URG: urgent data
(generally not used)

ACK: ACK #
valid

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
rcvr willing
to accept
(flow control)

TCP seq. numbers, ACKs

sequence numbers:

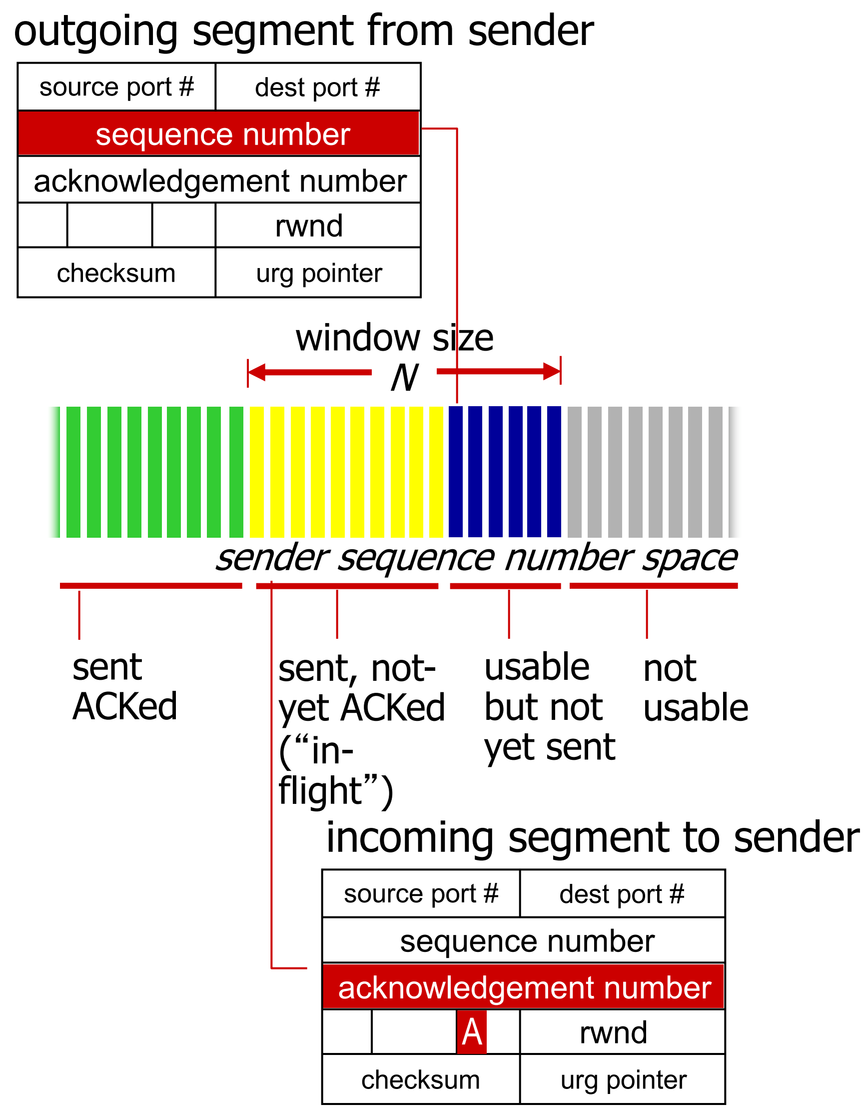
- byte stream "number" of first byte in segment's data

acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK (acknowledges bytes up to the first missing byte in the stream)

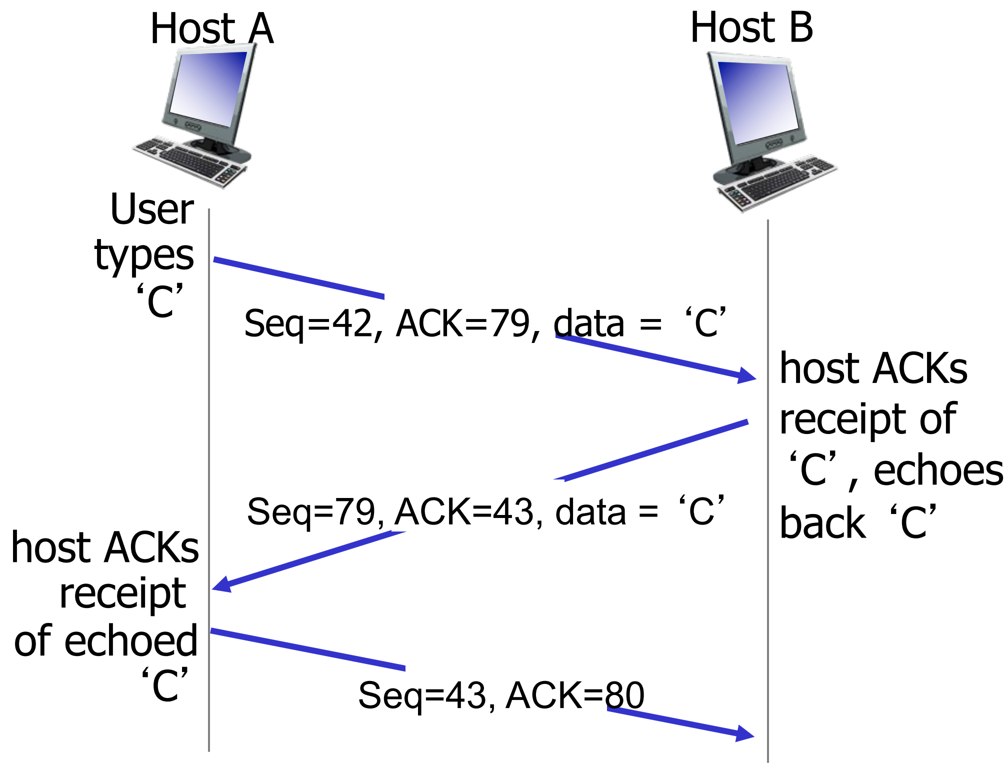
Q: how receiver handles out-of-order segments

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



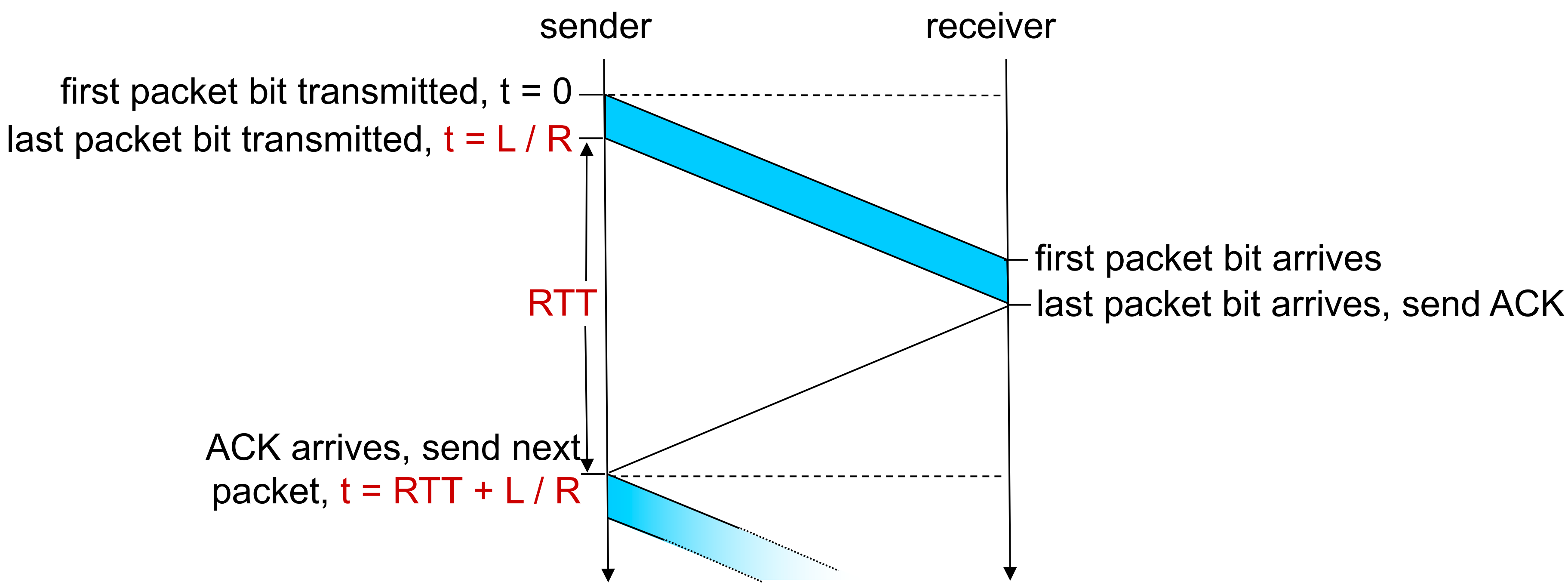
TCP seq. numbers, ACKs

- Acknowledgements contain the sequence number of next byte expected from other side
- Sequence number contain the byte stream “number” of first byte in segment’s data
- "Piggybacking": Using data segments to transport ACKs



simple telnet scenario

Stop-and-wait operation



Example: Considering R=1Gbps (10^9 bps), L=1000 bytes (packet size) and an RTT of 30 msec

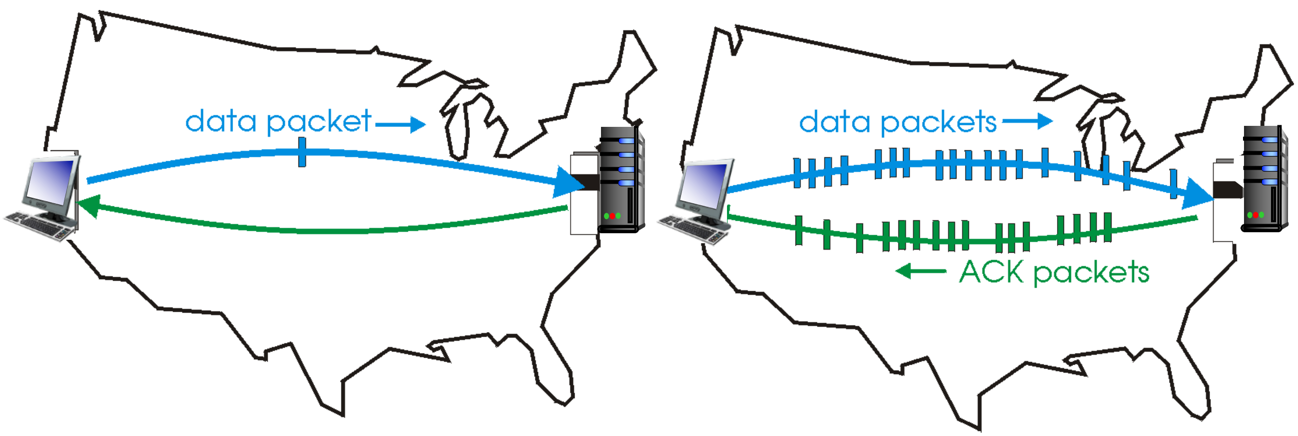
Sender is busy only 2.7 hundredths of one percent of the time!

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

Pipelined protocols

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

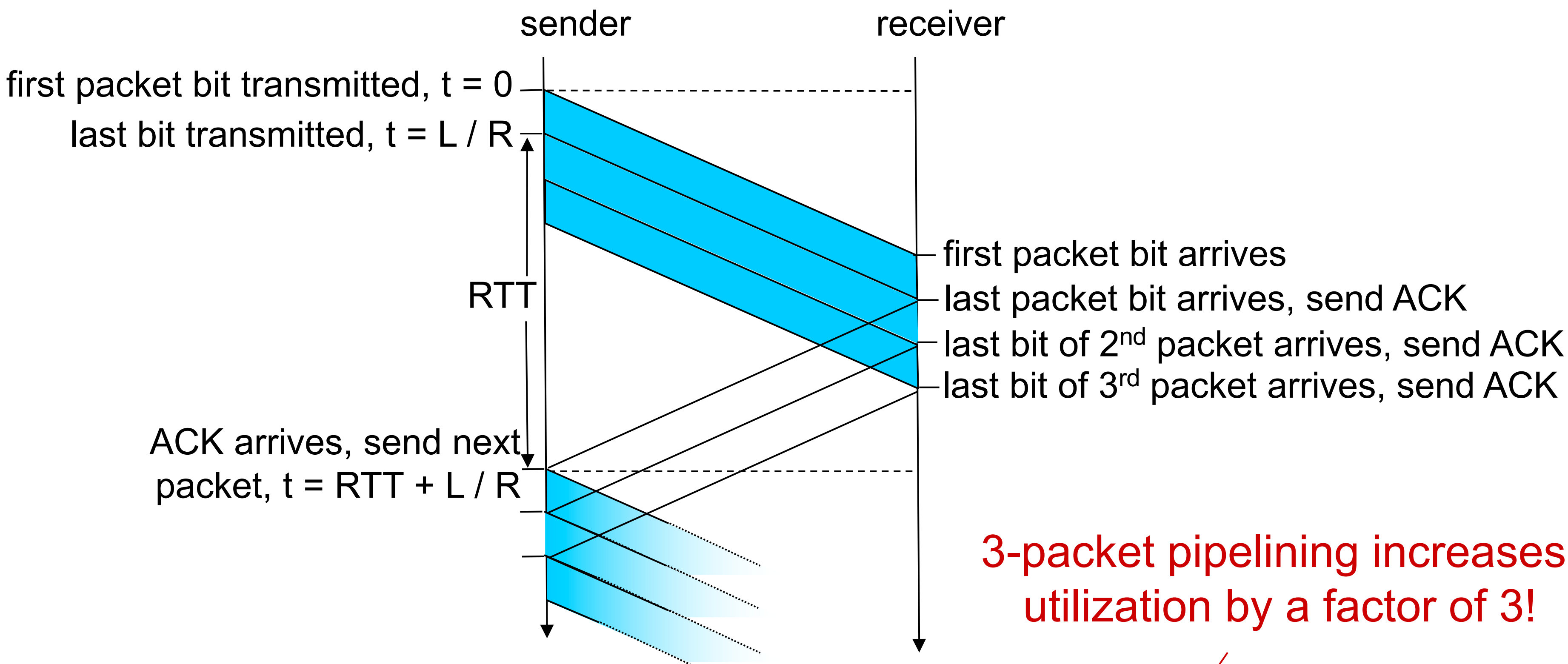
- An appropriate range of sequence numbers must be used
- buffering at sender and/or receiver is required



(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!

Considering again the previous example…

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

Pipelined protocols: overview

Go-back-N:

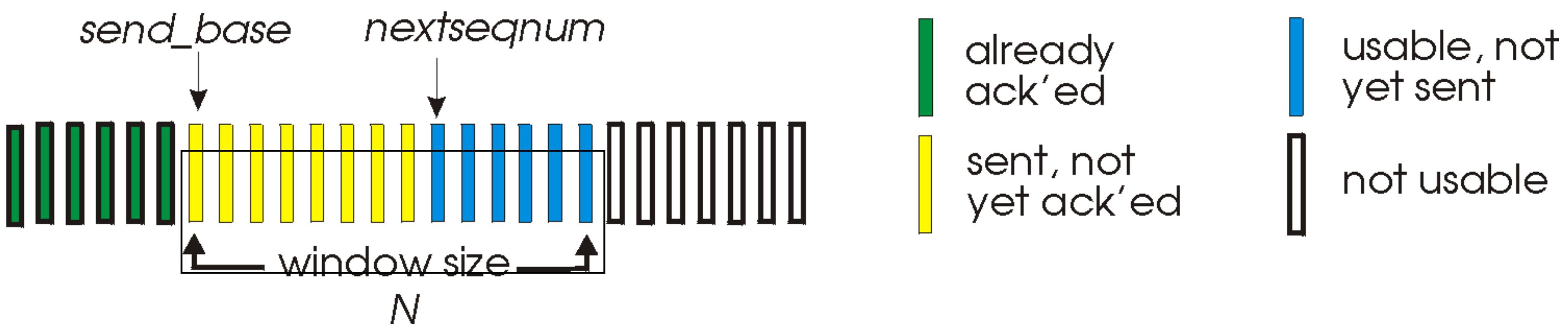
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

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

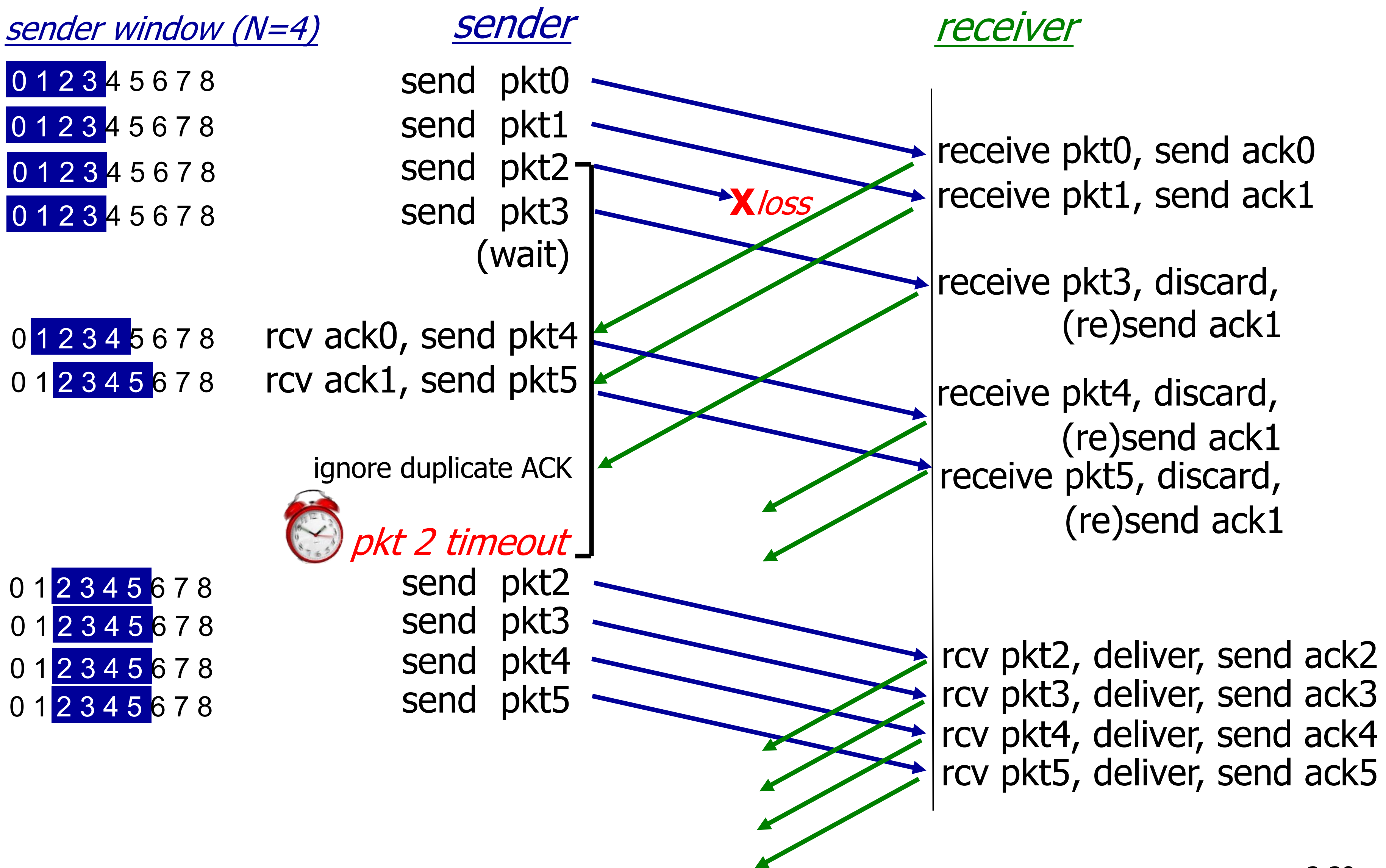
Go-Back-N: sender

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



- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window
- Simplicity: receiver need not buffer *any* out-of-order packets, just need to maintain seq number of next in-order packet

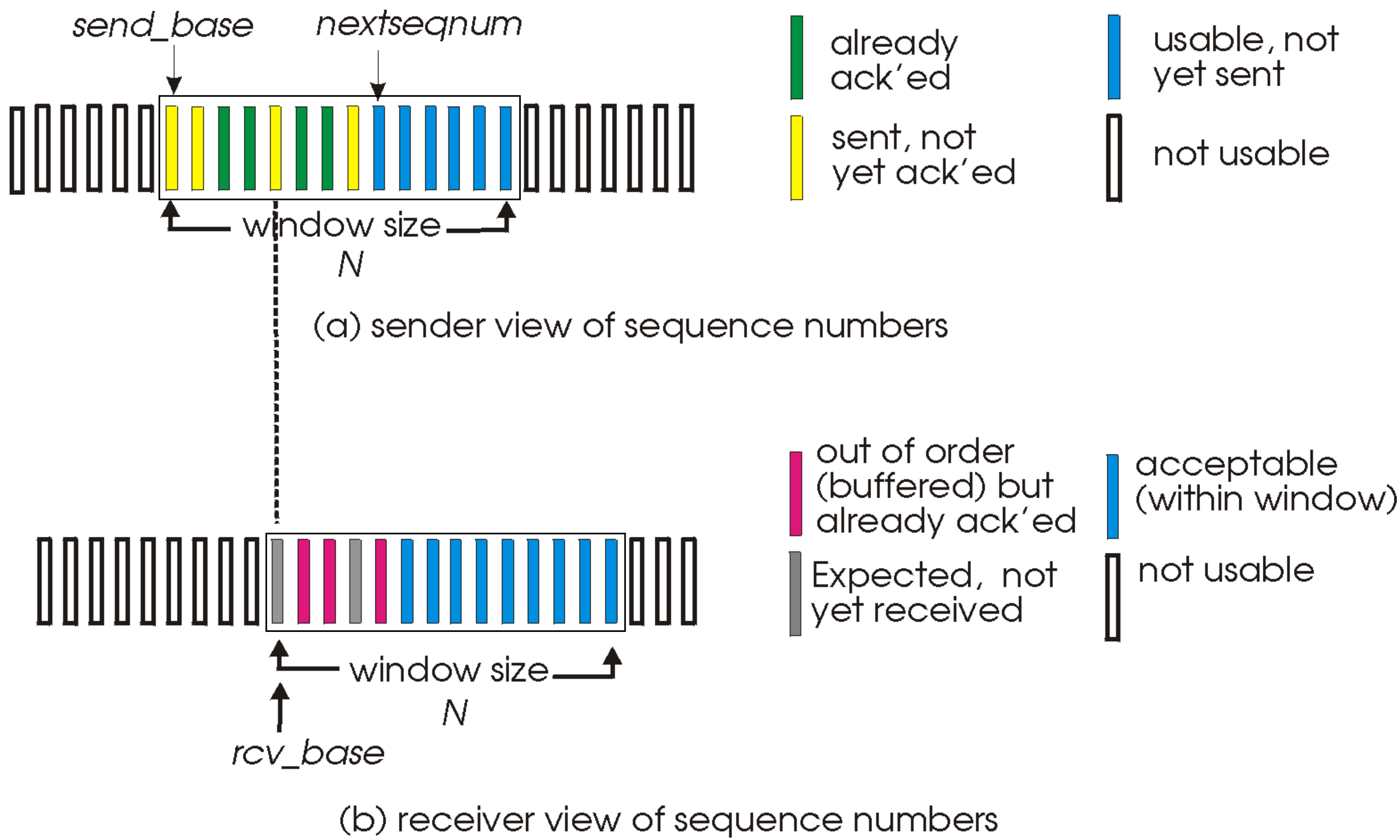
Go-Back-N in action



Selective repeat

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

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

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

receiver

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

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

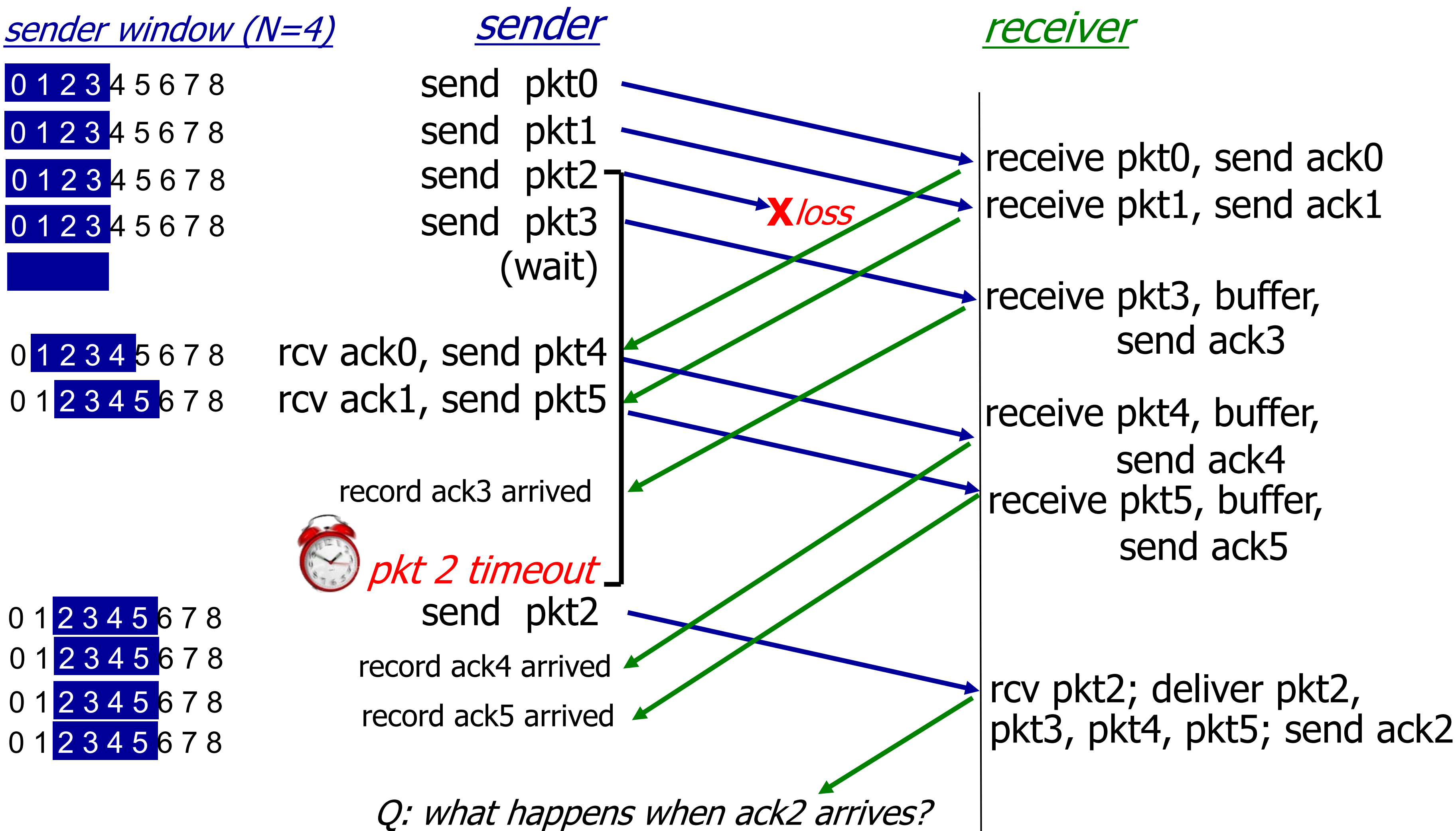
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



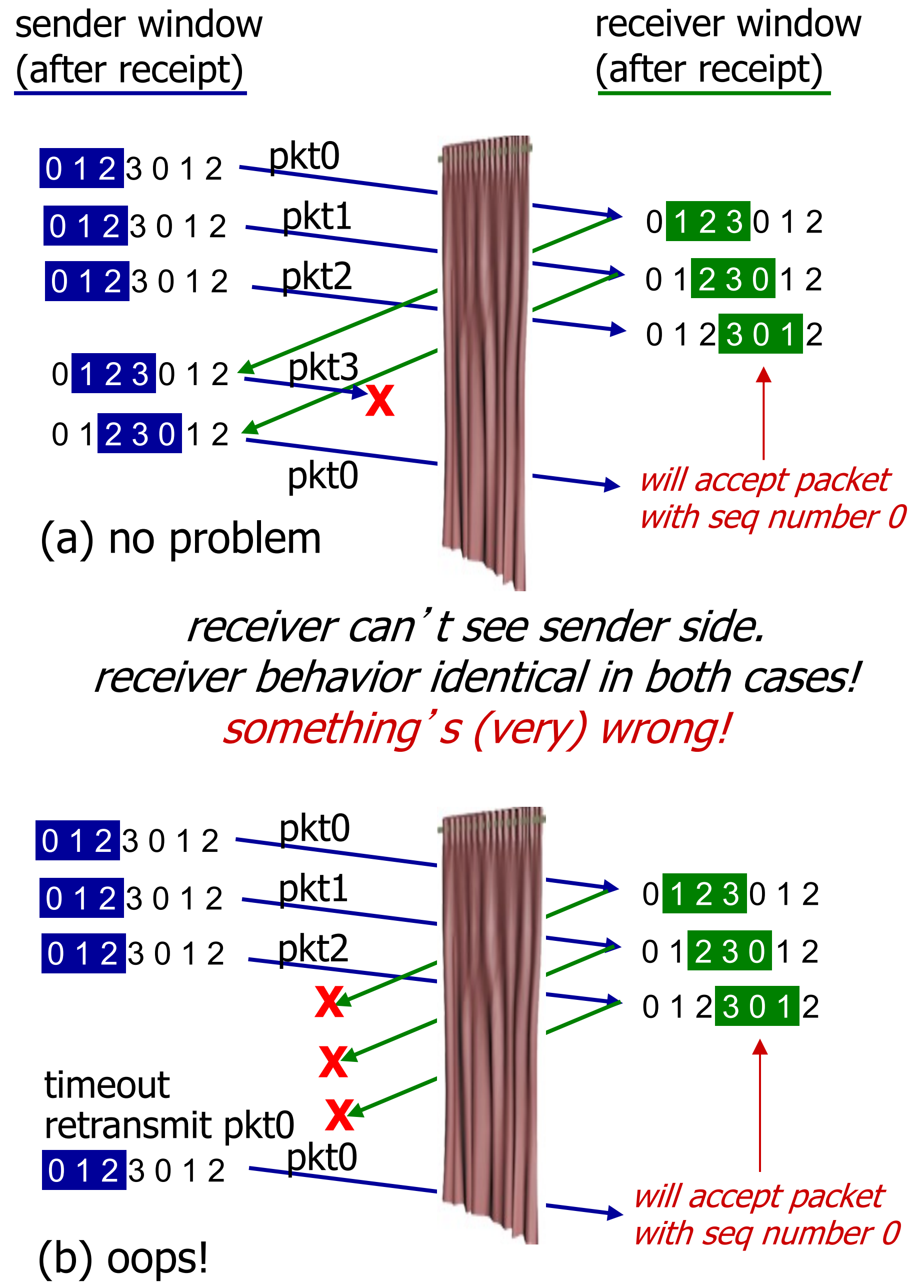
Selective repeat: dilemma

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)?

A: *window size must be less than or equal to half the seq #'s space*



Window size and performance

The window size relates directly with the performance of TCP connections. Considering:

W: Window size (in bytes)

C: Transmission Rate (in bps)

D: Propagation delay (in s)

S: Normalized throughput

Bits transmitted before a confirmation may arrive: 2CD

Bytes transmitted: 2CD/8 = CD/4

Window size and performance

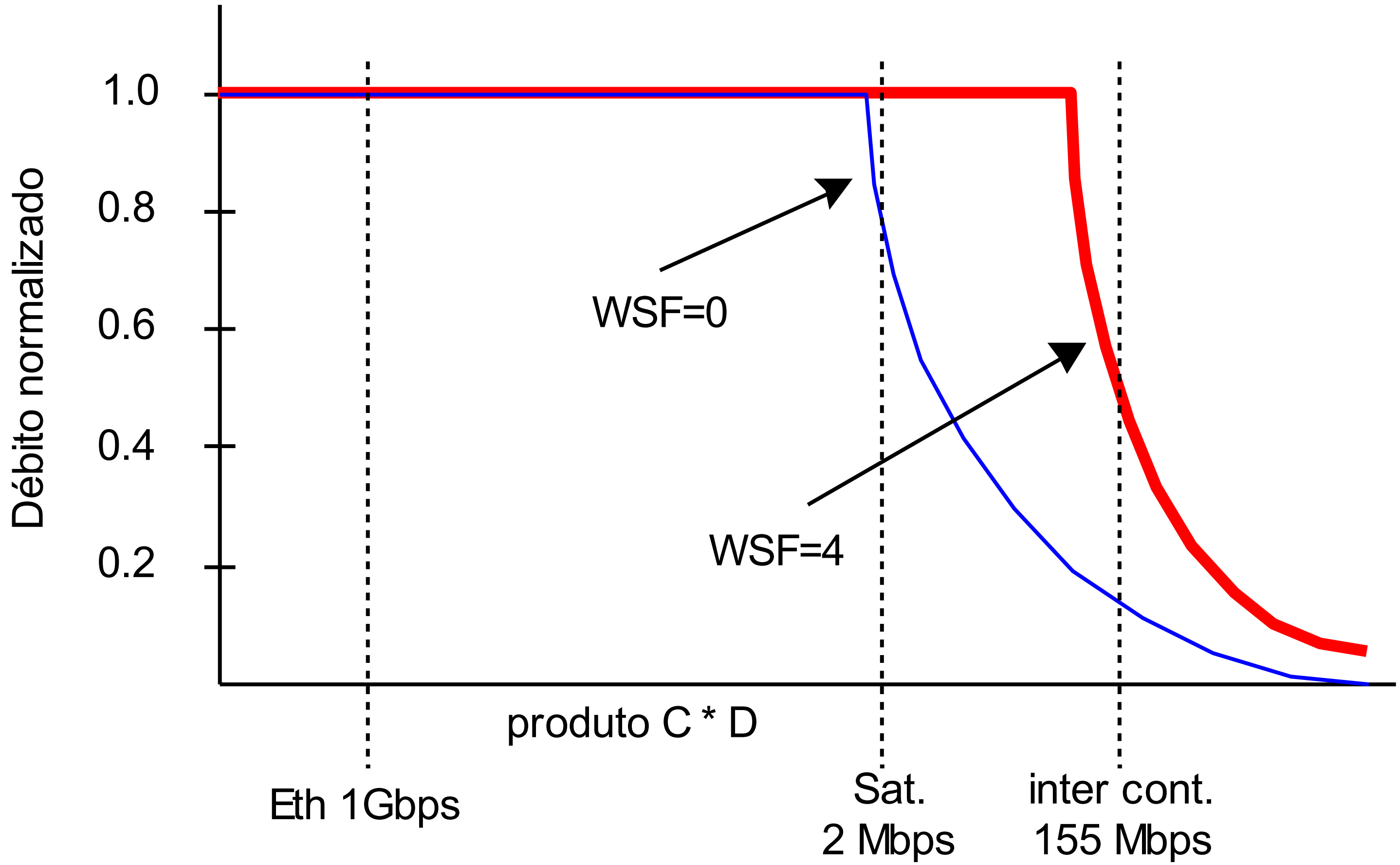
- Normalized throughput is 1 if window size is larger than number of bytes transmitted before a confirmation may be received by sender:

$$S = 1 \ , \quad \text{for } W > CD/4$$

- Otherwise normalized throughput is obtained by dividing windows size by the number of bytes that could be transmitted before confirmation arrives:

$$S = 4W/(CD) \ , \quad \text{for } W < CD/4$$

Window size and performance



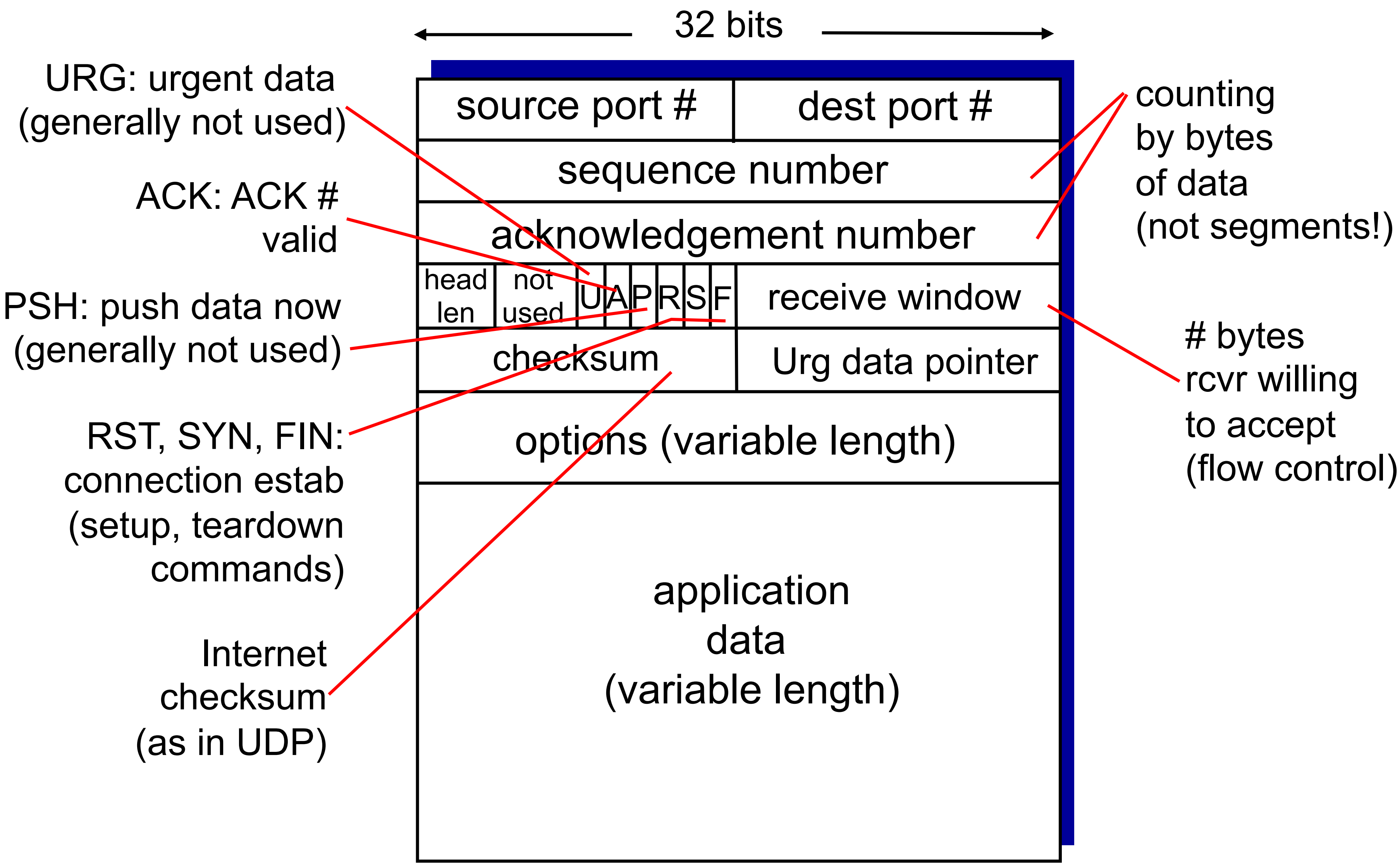
TCP Options

- **Maximum Segment Size (MSS)**: may be used during the connection establishment phase to negotiate the maximum size of the TCP segments that entity can receive (in bytes)

- **Window Scale Factor (WSF)**: may be used during the connection establishment phase to set larger window sizes (required for high bandwidth network links). If F is the value stored in this field ($F<15$), window size is multiplied by 2^F

- **Timestamp**: this option is set in data segments and copied to confirmation segments, and allows for continuous monitoring of the RTT between the client and server

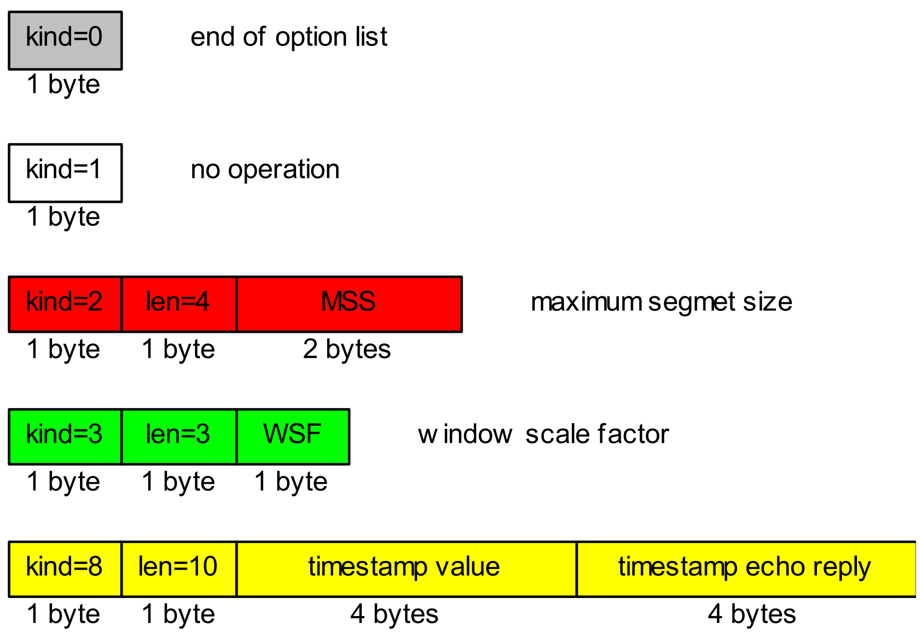
TCP segment structure



TCP Options (format)

TCP Option fields

- Kind: identifies the option.
- Length: total size of the option field
- Options are aligned in multiple 4-byte fields (the option “no operation” may be used for this purpose)



TCP round trip time, timeout

Q: how to set TCP timeout value?

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

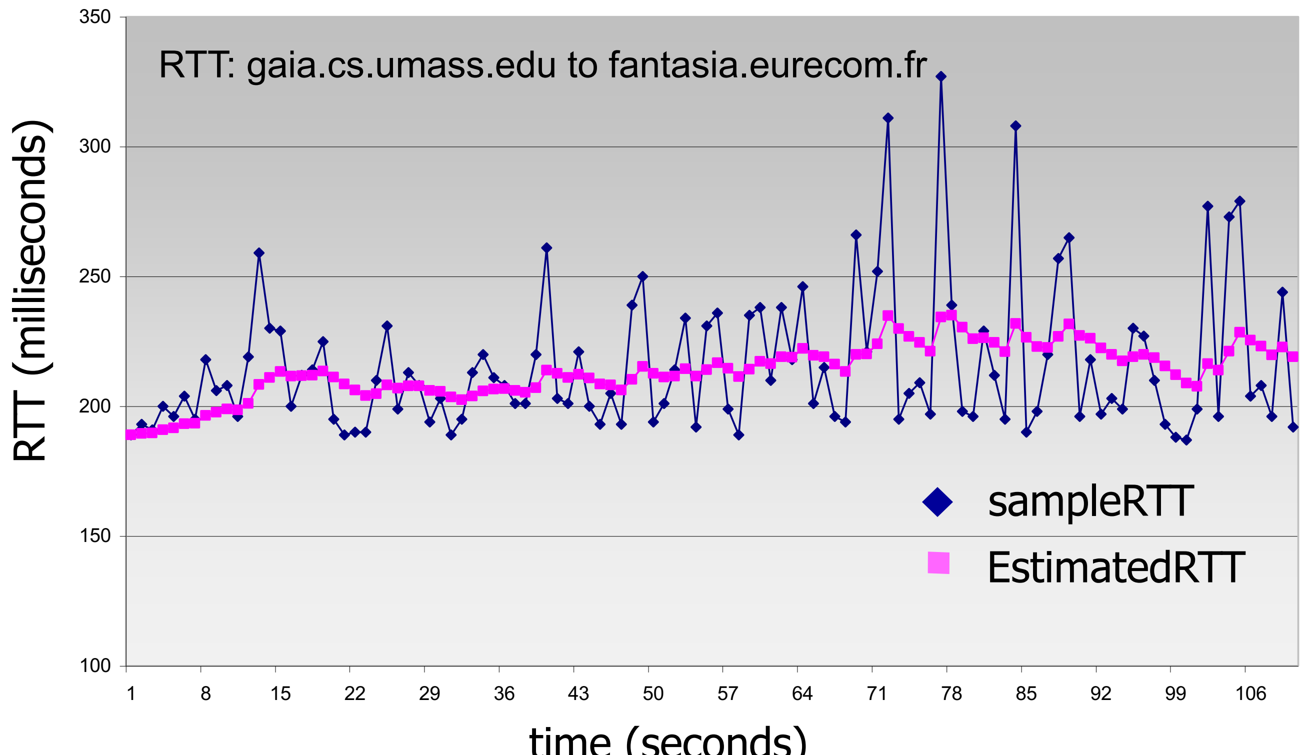
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125 (RFC 6298)
- Puts more weight on recent samples than on old samples



TCP round trip time, timeout

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

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|
              (typically, β = 0.25)
```

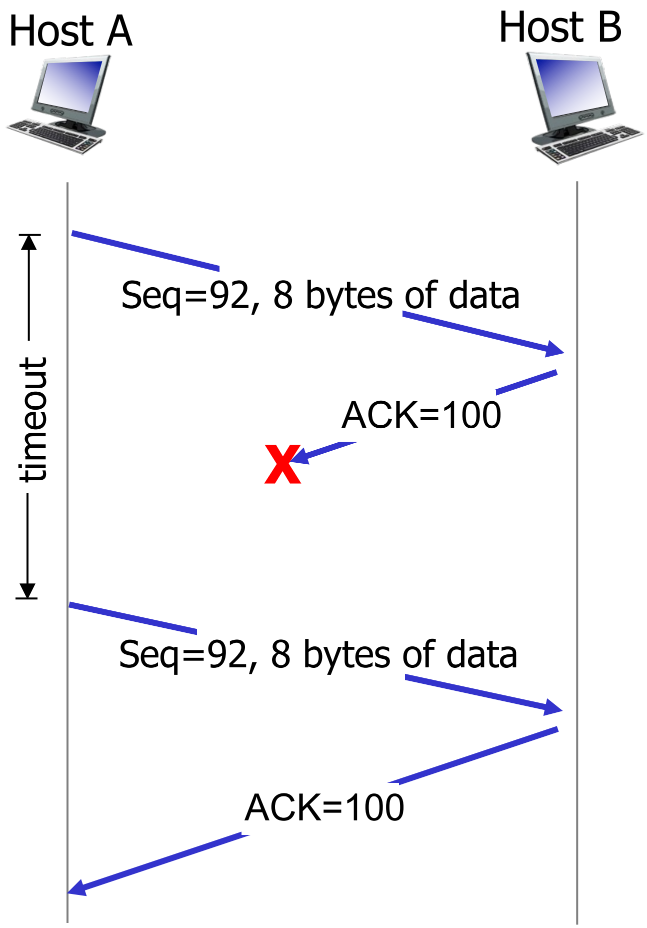
- DevRTT is small if SampleRTT values have little fluctuation, large otherwise

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

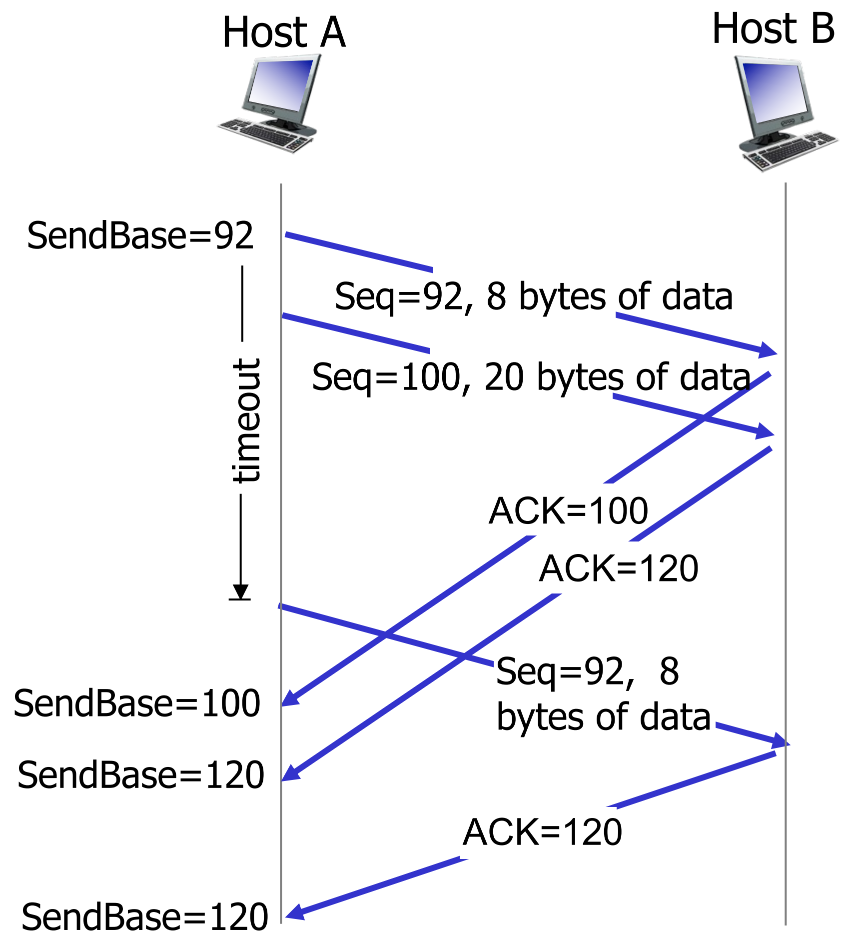
estimated RTT

"safety margin"

TCP: retransmission scenarios

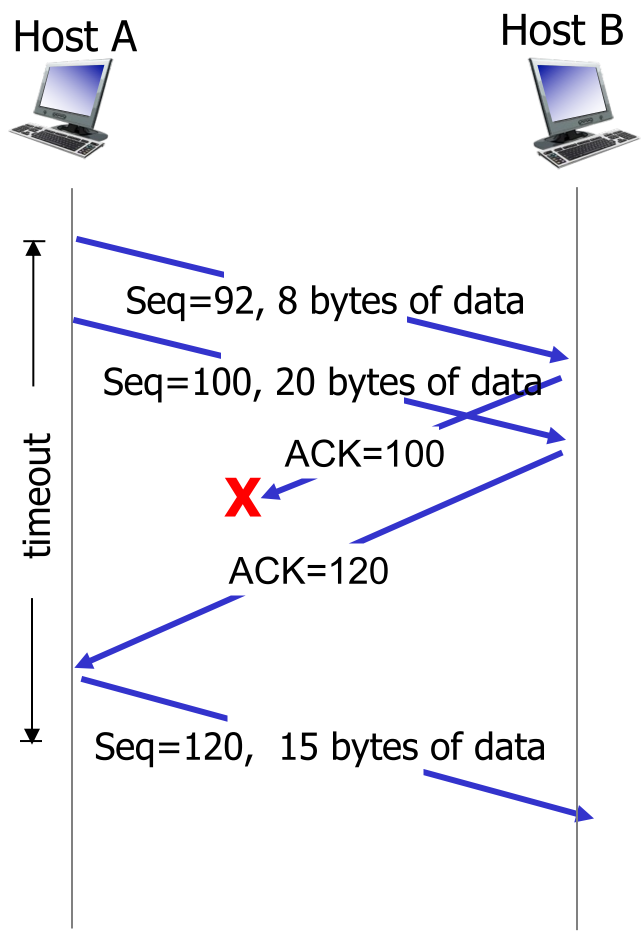


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP ACK generation [RFC 1122, RFC 2581]

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

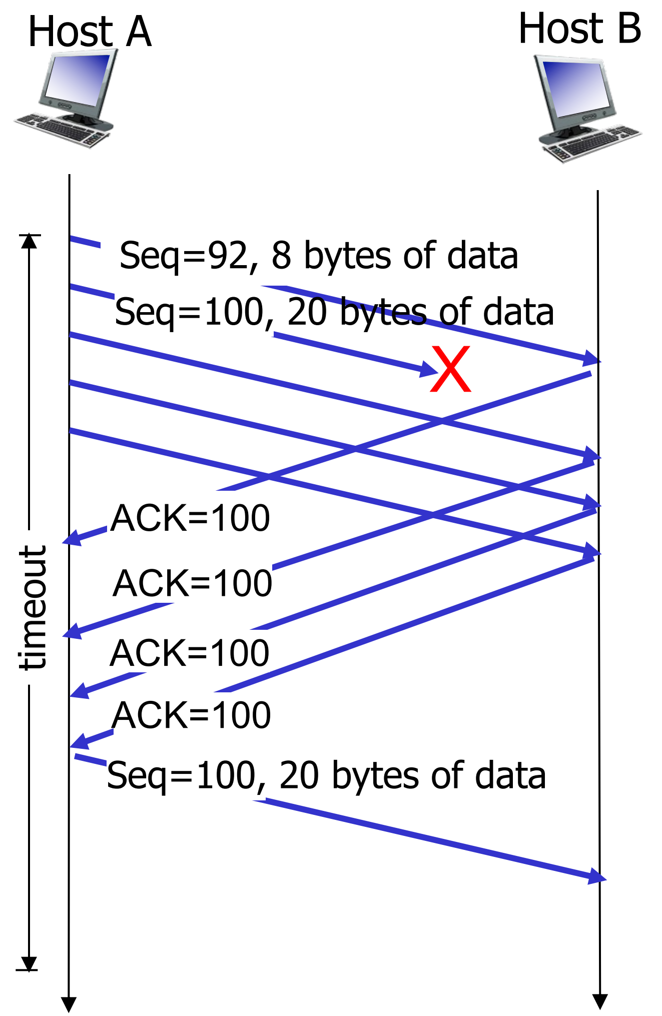
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs:
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so <u>don't bother wait for timeout</u>

TCP fast retransmit



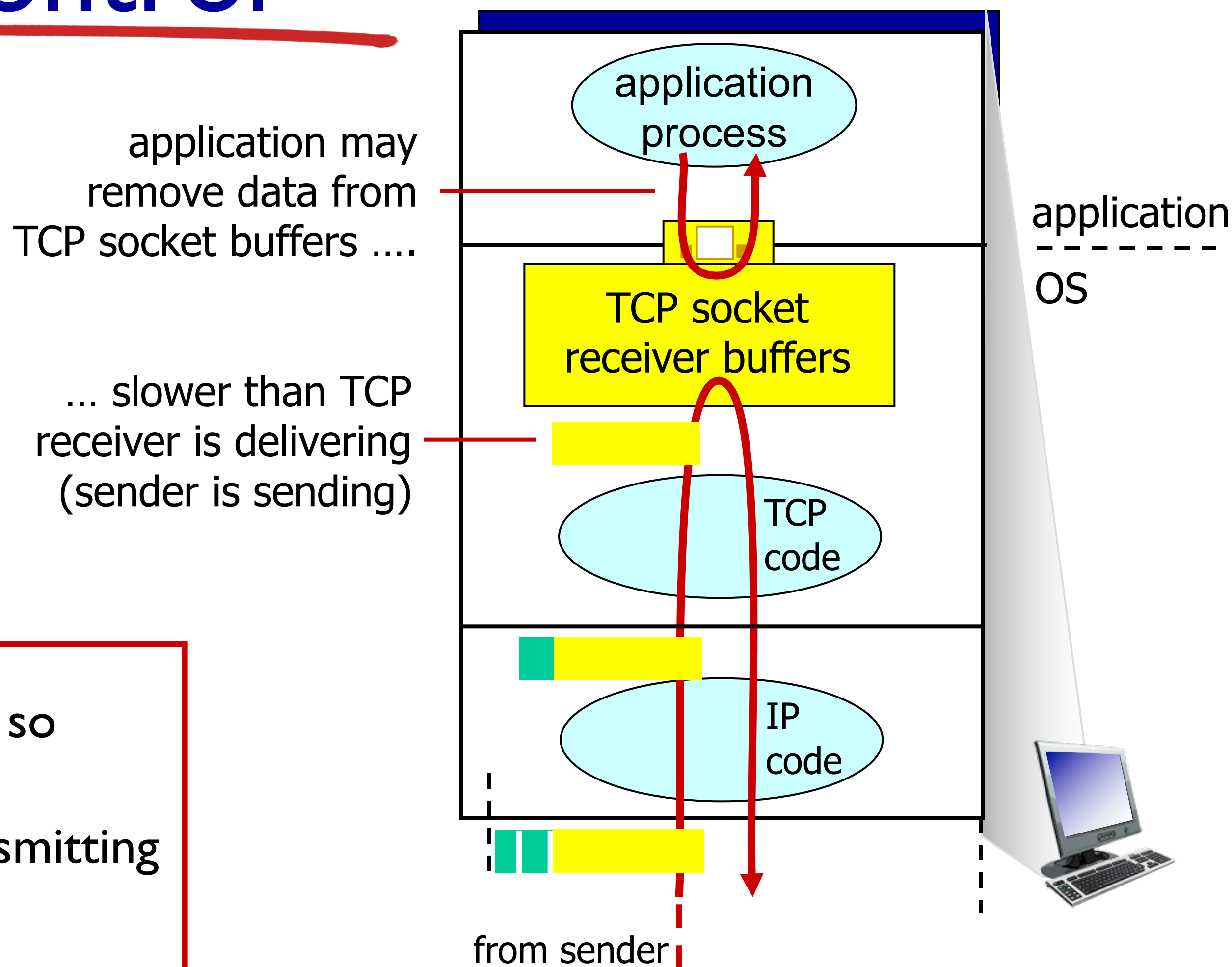
fast retransmit after sender receipt of triple duplicate ACK

Is TCP Go-Back-N or Selective Repeat?

- TCP ACKs are cumulative
- Correctly received but out-of-order segments are not individually ACKed by the receiver
- But many implementations buffer such segments
- Retransmissions may be of only a single lost segments if subsequent ACKs arrive before timeout

- TCP's error recovery is best categorized as a **hybrid** of GBN and SR!

TCP flow control



receiver protocol stack

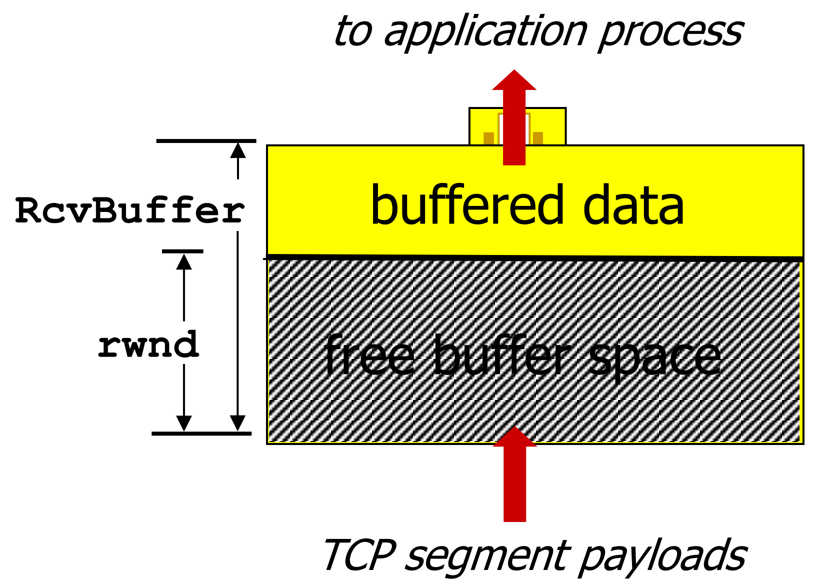
flow control

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

Flow control it's a speed-matching service

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

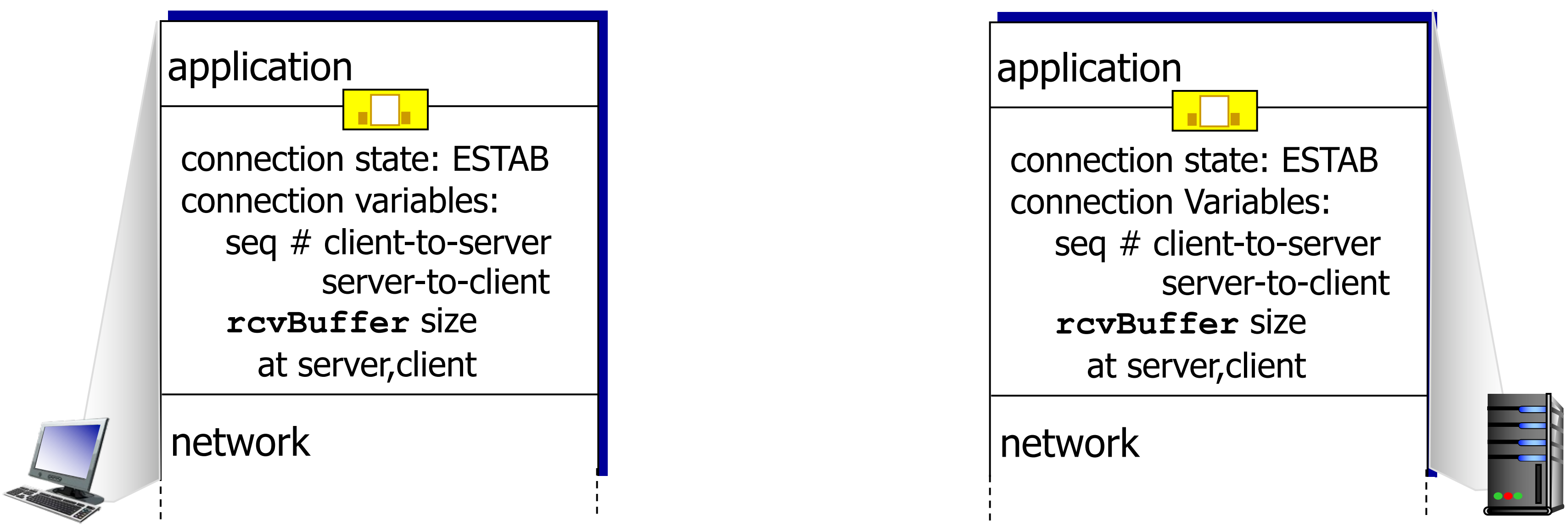


receiver-side buffering

Connection Management

before exchanging data, sender/receiver "handshake":

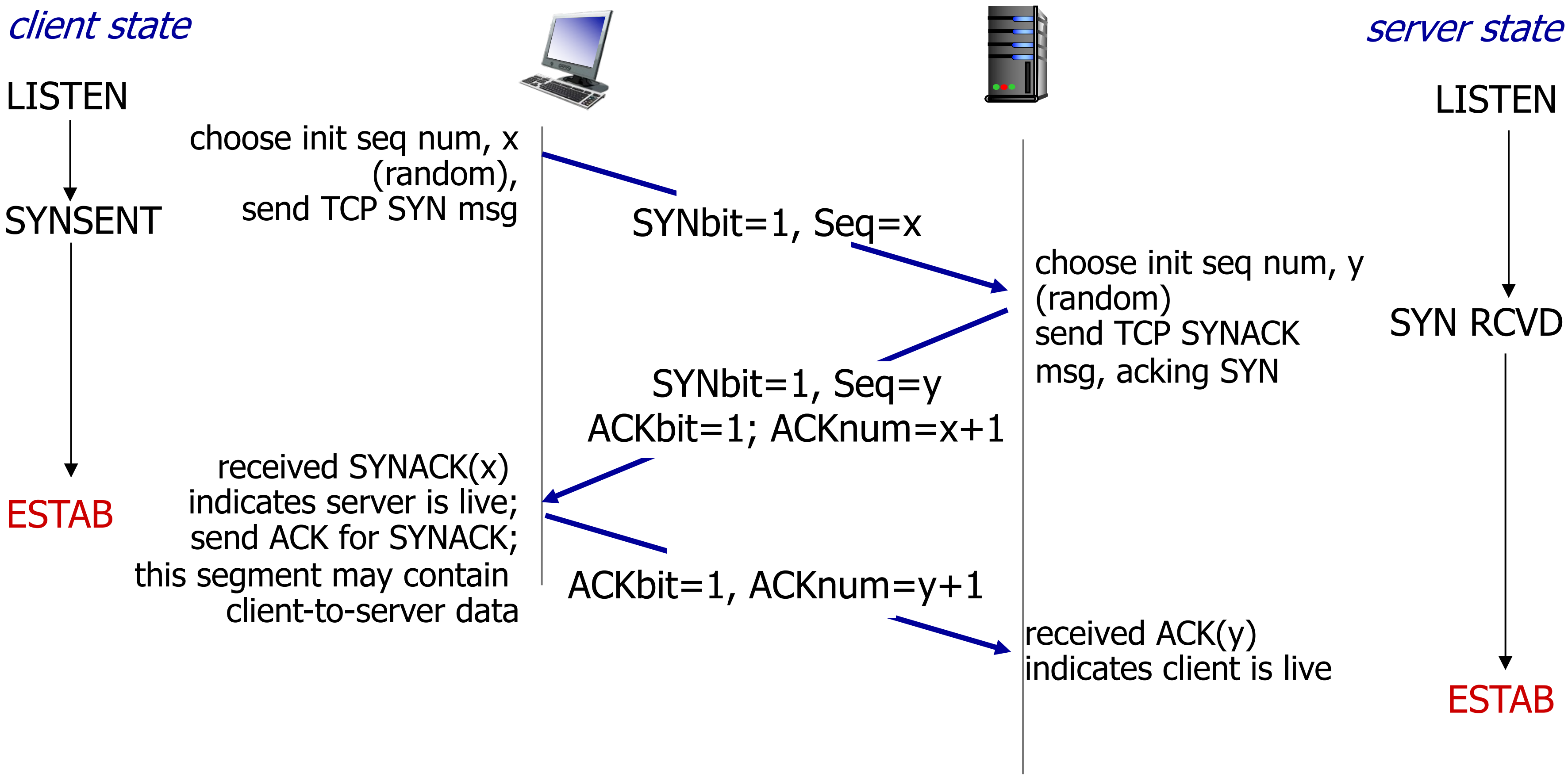
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters




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

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

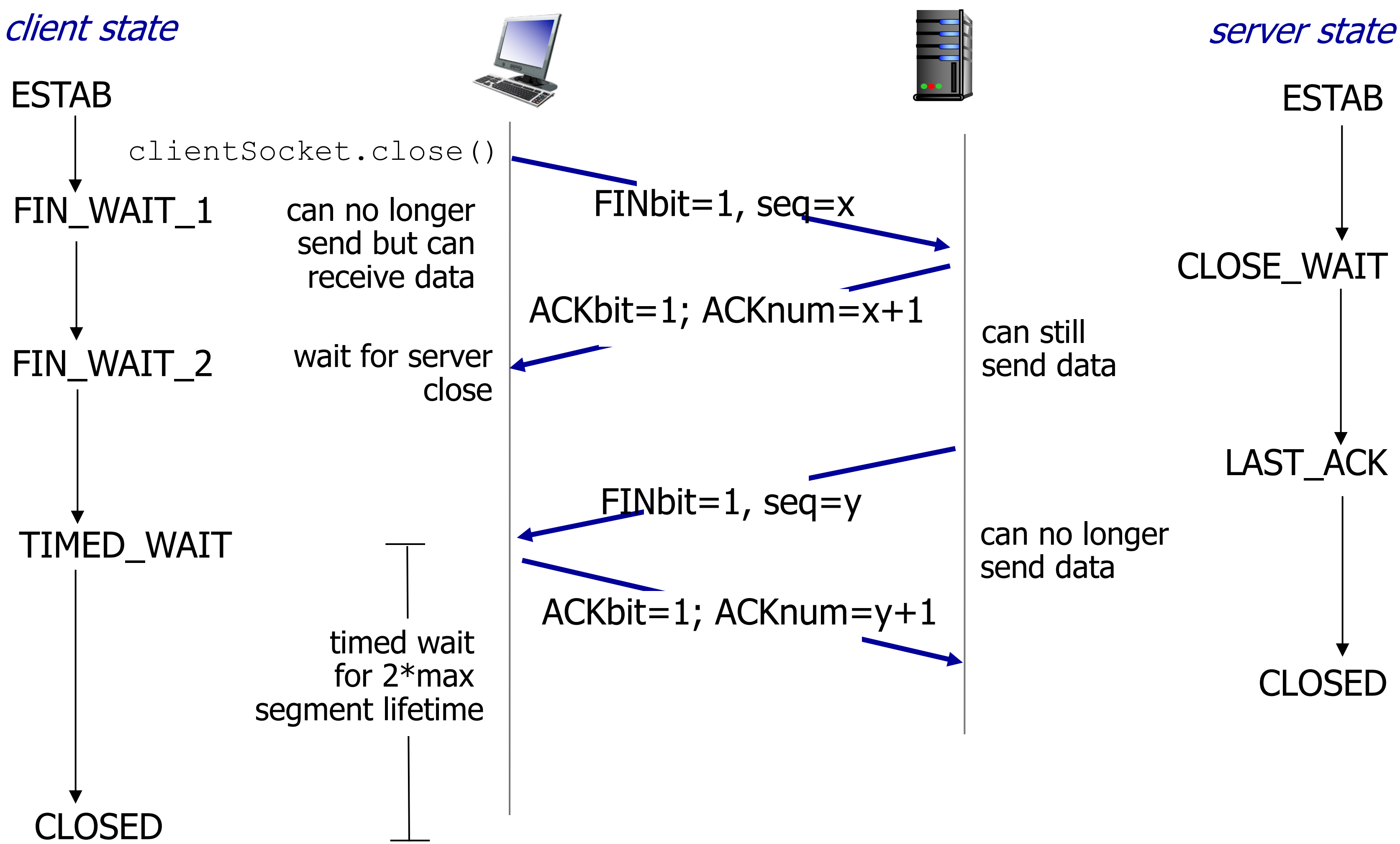
TCP 3-way handshake



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



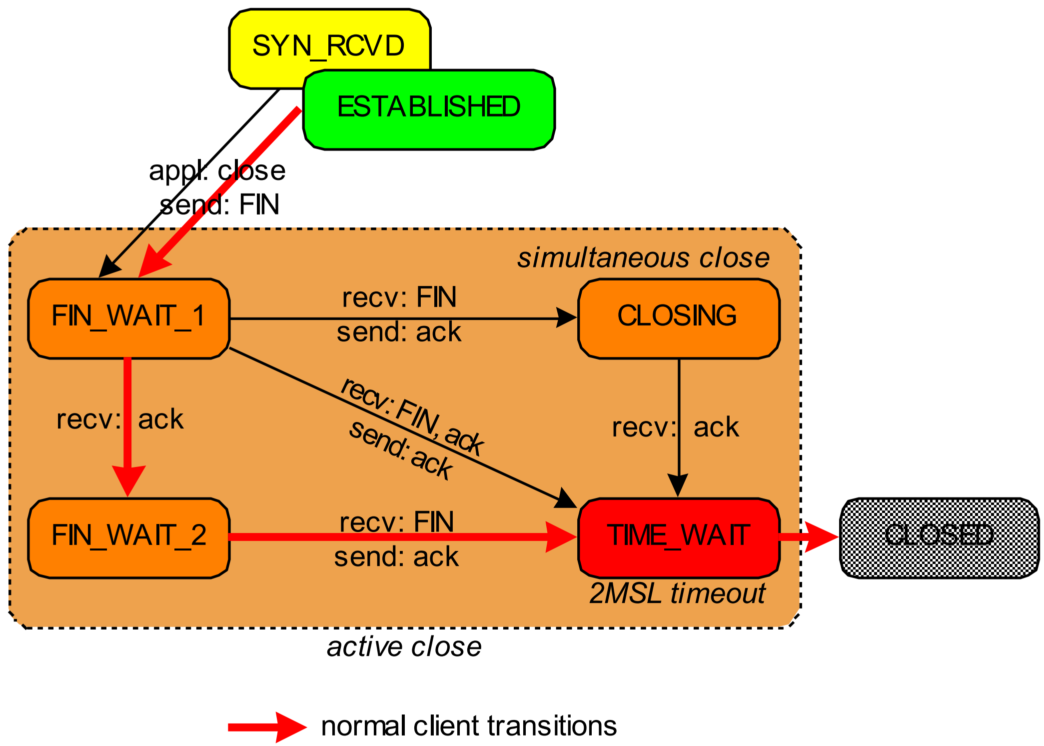
What is the 2MSL timer?

- Maximum segment lifetime (MSL) is the time a TCP segment can exist in the internetwork system

- The purpose of TIMED_WAIT is to prevent delayed packets from one connection being accepted by a later connection

- 2MSL depends on implementations, for example in MacOsX:

sysctl net.inet.tcp.msl = 15000
(2MSL = 30s)



Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

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

3.5 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!

Approaches to Congestion control

We can distinguish among congestion-control approaches by whether the network layer provides any explicit assistance :

- **End-to-end congestion control**: no explicit support from the network layer, congestion is inferred from network behavior
  - Used in TCP, sender limits the rate at which it sends traffic as a function of *perceived network congestion*
  - In case of loss (timeout) or triple ACKs, TCP decreases its (congestion) window size accordingly
  - May also consider increasing RTT as indicators of increased congestion
- **Network-assisted congestion control**
  - Routers provide explicit feedback
  - As used in IBM SNA, DEC DECnet, ATB ABR

Congestion and receiver window

- The TCP congestion-control mechanism operating at the sender keeps track of an additional variable: the congestion window (cwnd)

- The congestion window imposes a constraint on the rate at which TCP sender can send data into the network

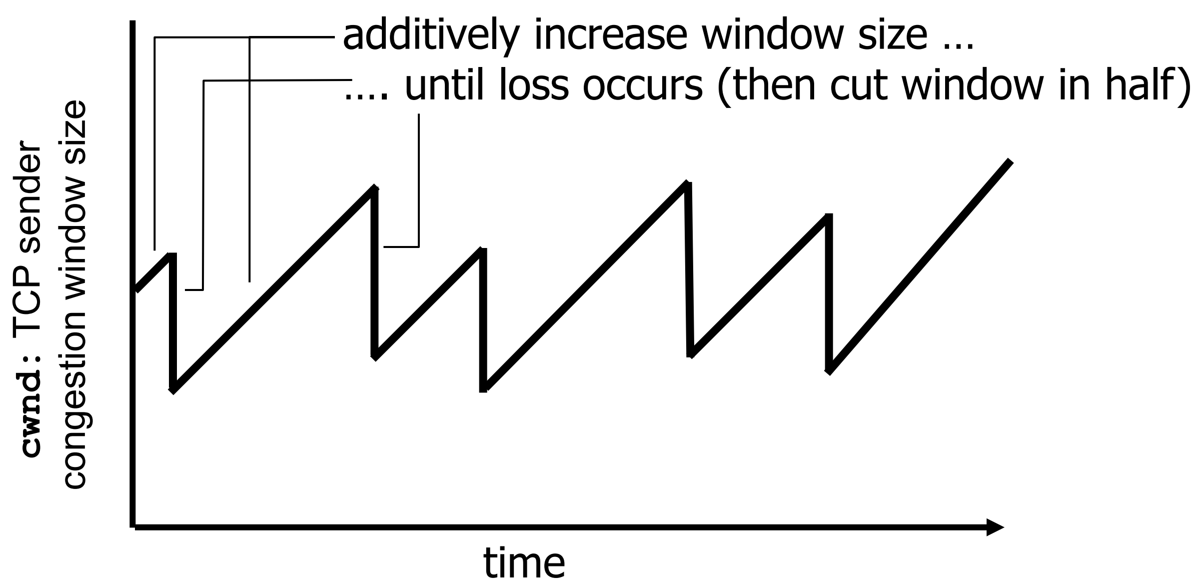
- At any given time, the amount of acknowledged data at sender may not exceed the minimum of *cwnd* and *rwnd*:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \min \{ \underbrace{\text{rwnd}}_{\text{Receiver window (flow control)}}, \underbrace{\text{cwnd}}_{\text{Congestion window (congestion control)}} \}$$

TCP congestion control: AIMD principle (additive increase multiplicative decrease)

- *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

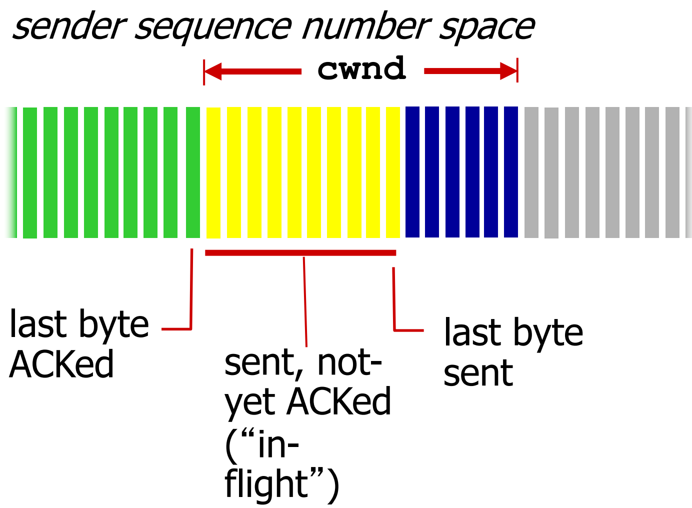
AIMD saw tooth behavior: probing for bandwidth



TCP congestion control algorithm

- Standardized in RFC 5681
- Is based on three major components:
  - **Slow start**: set initial transmission rate slow but ramp up exponentially fast (until loss is detected)
  - **Congestion avoidance**: on entering this state the value of cwnd is approximately half its value when congestion was last encountered
  - **Fast recovery**: when 3 ACKs are received sender performs fast retransmit of missing segment, proceeds in congestion avoidance mode (network is still capable of delivering segments)

TCP Congestion Control: details



- sender limits transmission:

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

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

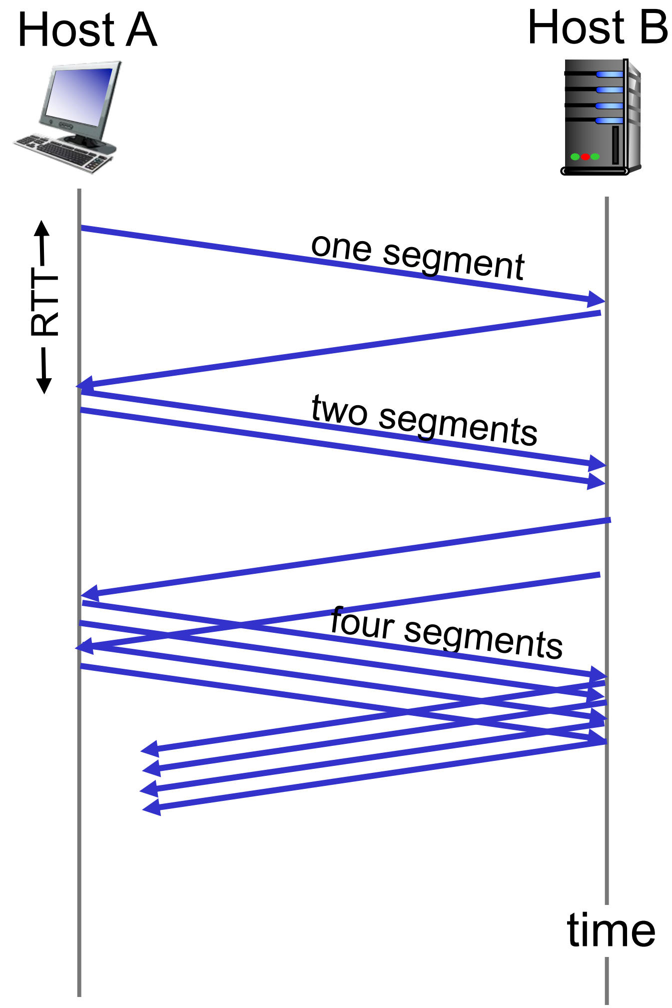
TCP sending rate:

- *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
  - double **`cwnd`** every RTT
  - done by incrementing **`cwnd`** for every ACK received
- *summary:* initial rate is slow but ramps up exponentially fast



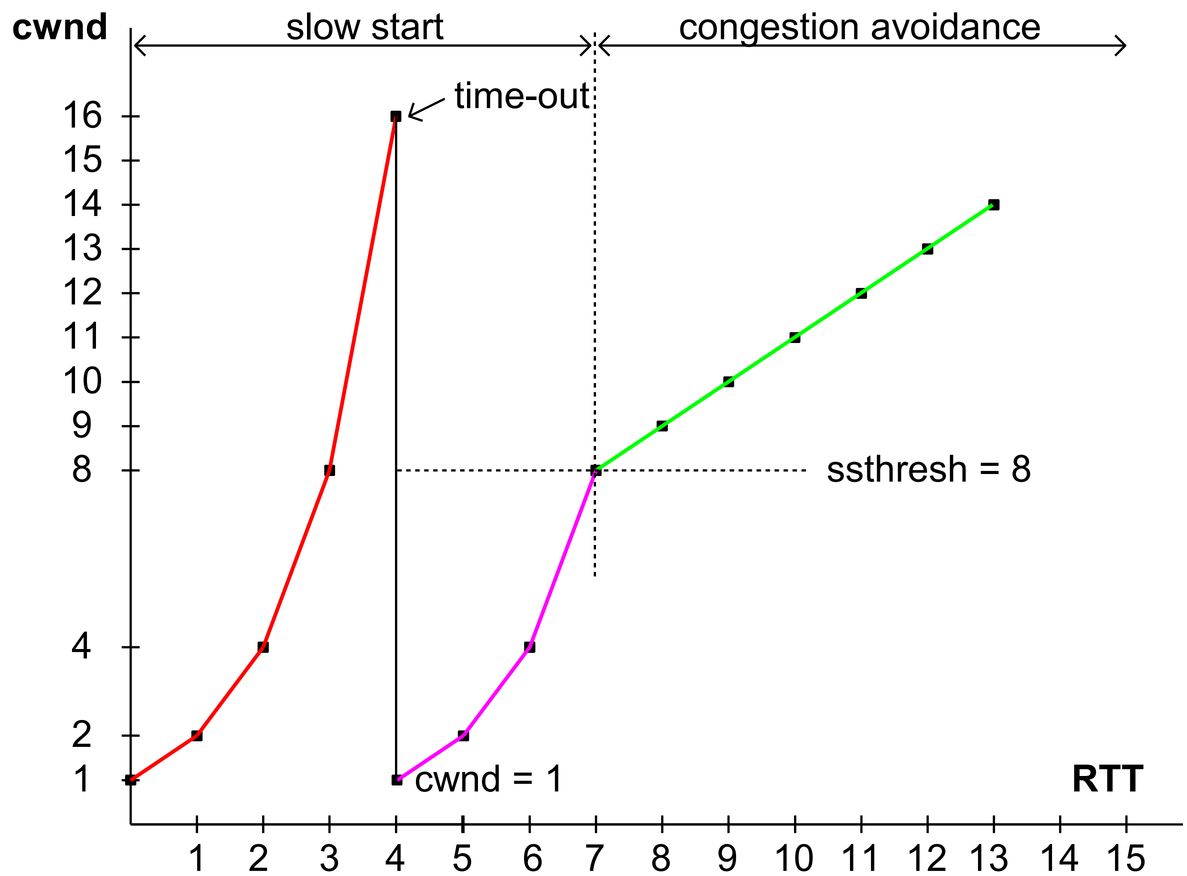
TCP: detecting, reacting to loss

- loss indicated by *timeout*:
  - **`cwnd`** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by *3 duplicate ACKs*:
  - dup ACKs indicate network capable of delivering some segments
  - Fast retransmit missing segment
  - **`cwnd`** is cut in half window then grows linearly

TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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



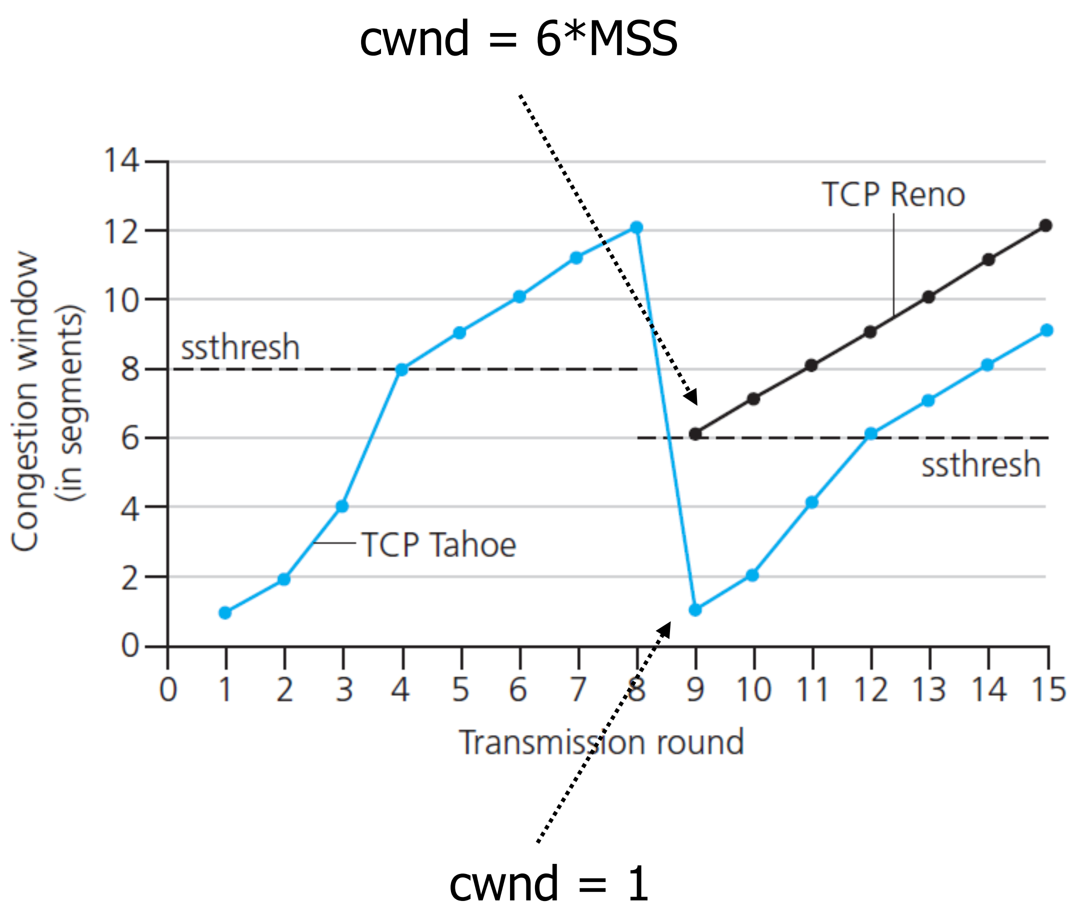
Implementation:

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

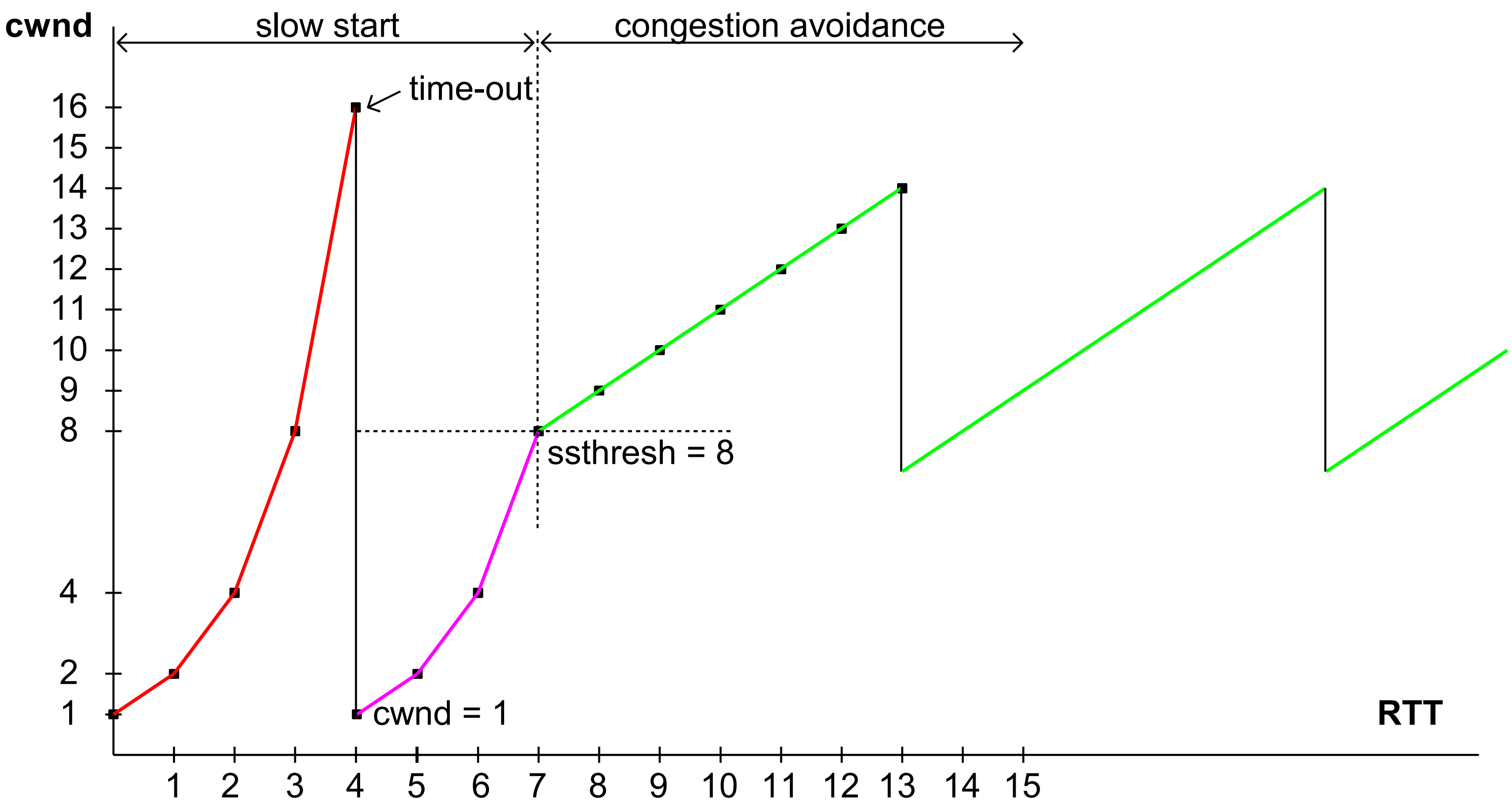
Fast recovery is recommended in TCP, but not required

- Early version of TCP (Tahoe) unconditionally cut cwnd to 1 MSS and enters slow-start after either a timeout of triple ACKs received

- TCP Reno (more recent) incorporates fast recovery



Typical behavior in TCP



TCP *flavours*

	RFC 793 Postel 81	Tahoe Jacobson 88	Reno Jacobson 90	Vegas Barkmo 94	SACK S. Floyd 2000
Slow start		√	√	√	√
Congestion avoidance		√	√	√	√
Fast retransmit		√	√	√	√
Fast recovery			√	√	√
Selective ACK					√

T04: summary

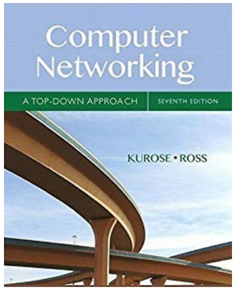
- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

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

T04: Bibliography

J. Kurose and K. Ross, “Computer Networking – a top-down approach”, Pearson. Chapter 3: Transport Layer



Redes de Comunicação 2023/2024

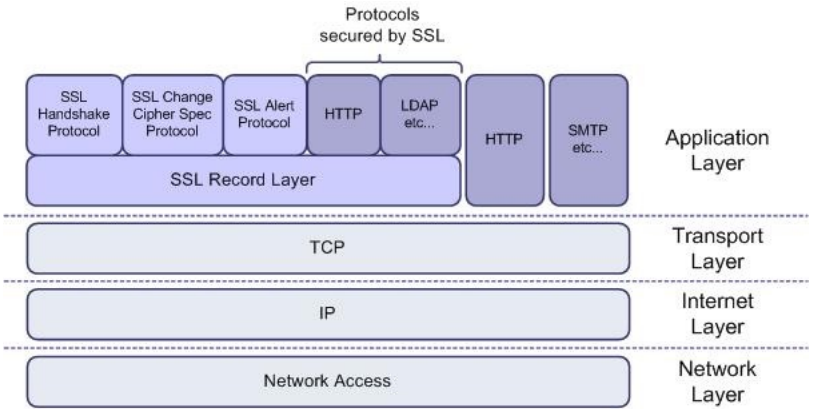
T03 Transport Layer Extra material

Jorge Granjal
University of Coimbra

T03: Transport Layer Security

- Transport Layer Security (TLS) or Security Services Layer (SSL) are security cryptographic protocols designed to provide communications security over a computer network.

[Encryption and its Applications]()
[SSL, TLS, HTTP, HTTPS Explained]()



T03: LFN (Long Fat Networks)

- The Window Size Option allows for larger window sizes (multiplication factor)
- A network with a large bandwidth-delay product is commonly known as a long fat network (LFN)

[The Bandwidth Delay Problem]

[How TCP Works - Window Scaling and Calculated Window Size]