Chapter 3
Transport Layer

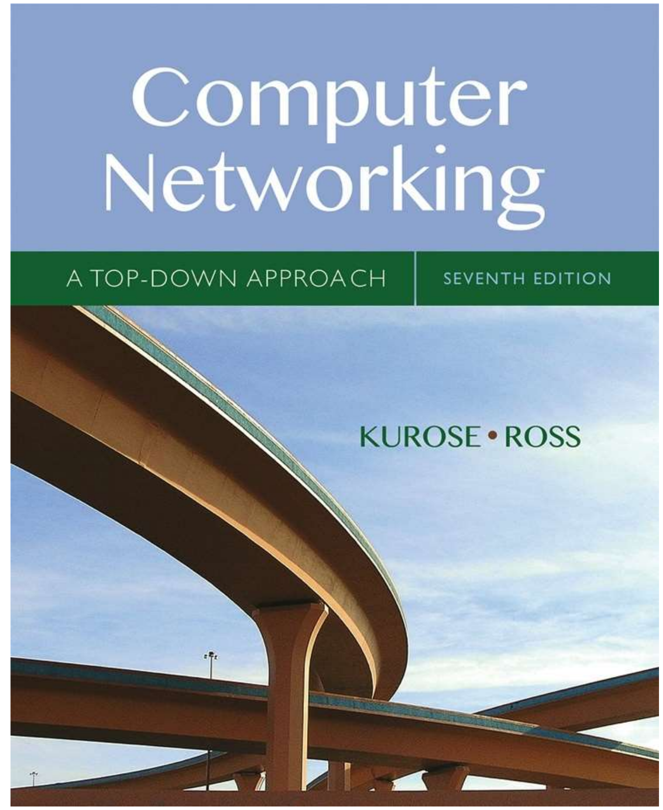
A note on the use of these Powerpoint slides:

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

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

Thanks and enjoy! JFK/KWR

© All material copyright 1996-2016
J.F Kurose and K.W. Ross, All Rights Reserved



Computer Networking: A Top Down Approach

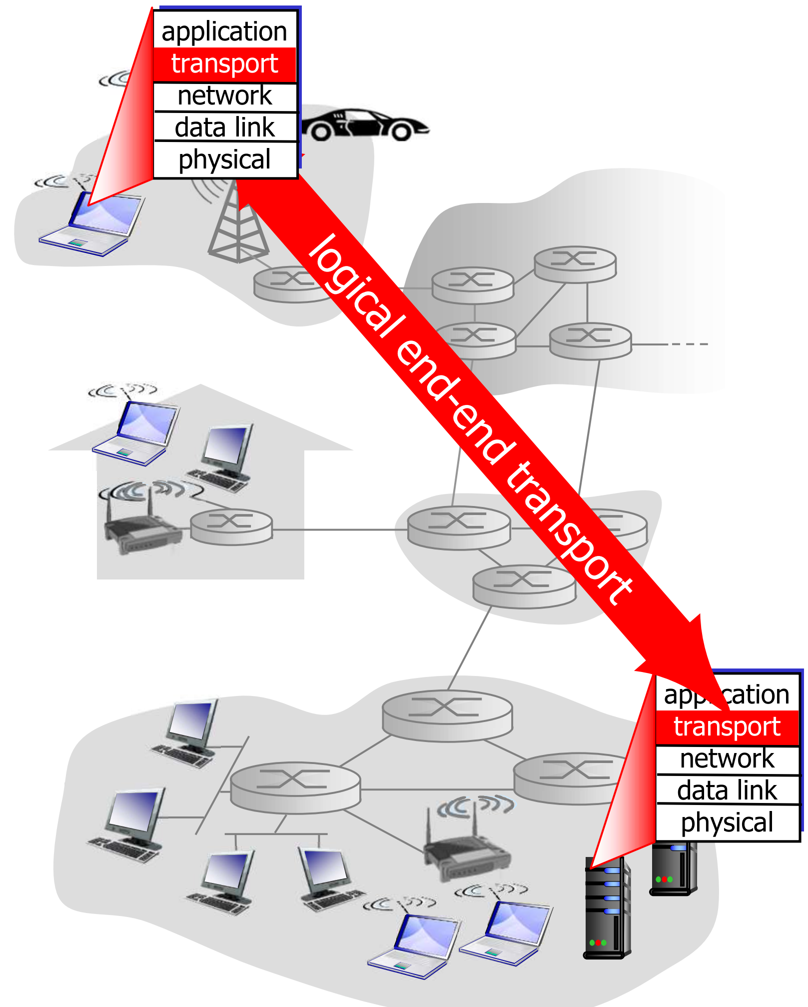
7^{th} edition

Jim Kurose, Keith Ross

Pearson/Addison Wesley
April 2016

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

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

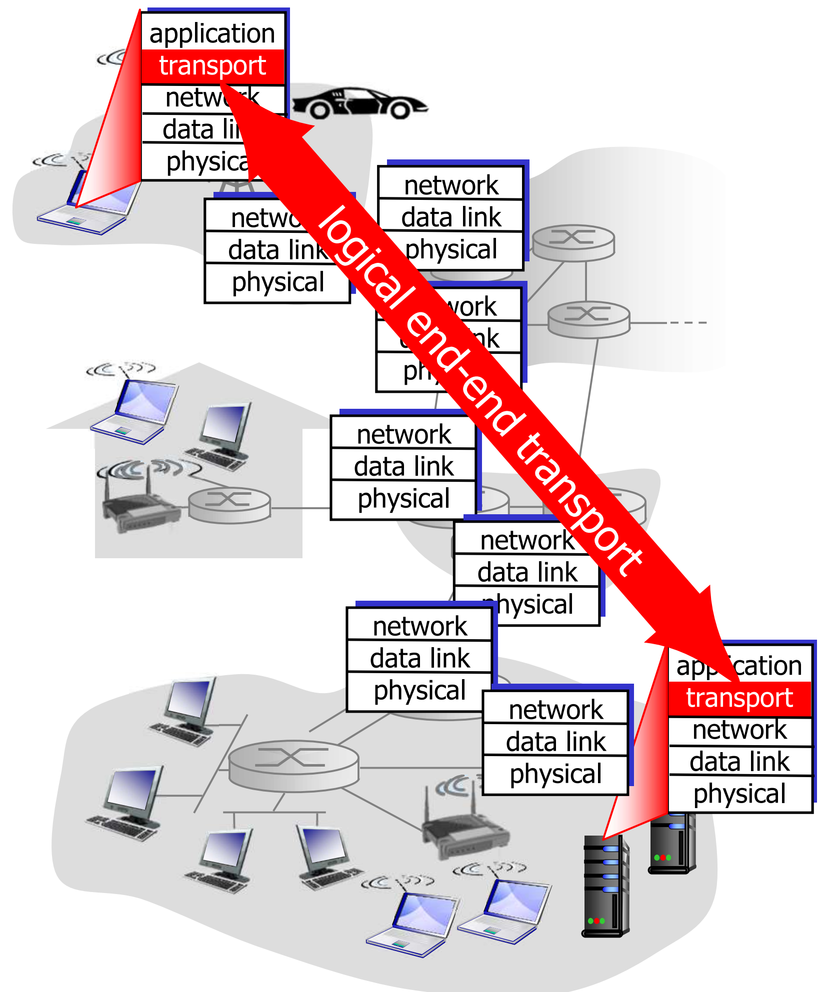
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

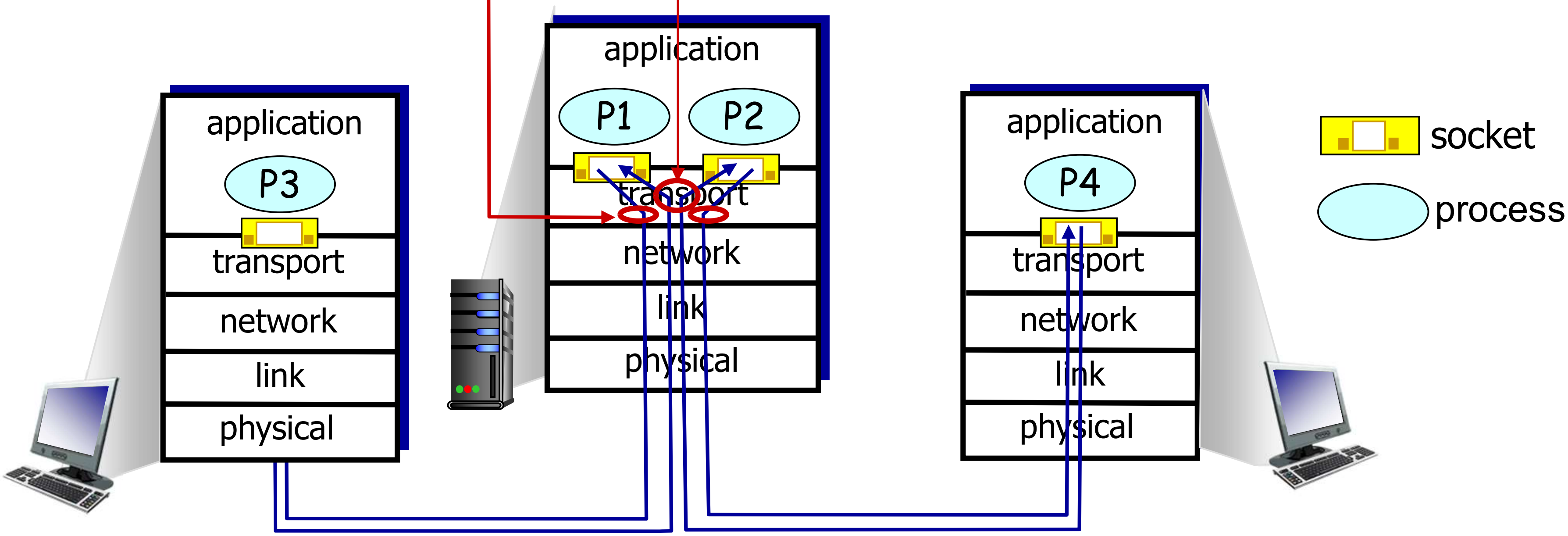
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

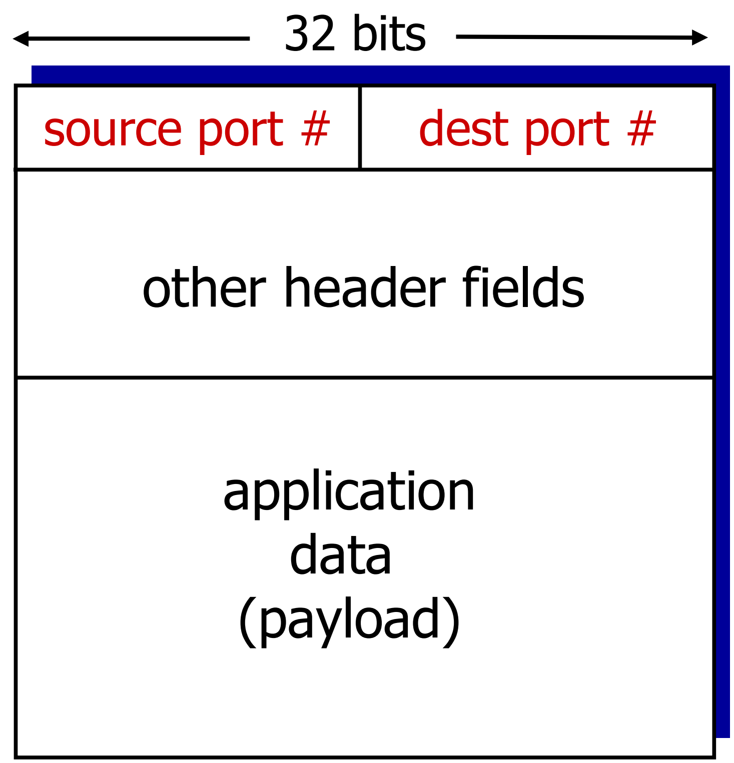
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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



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

- *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, 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 #

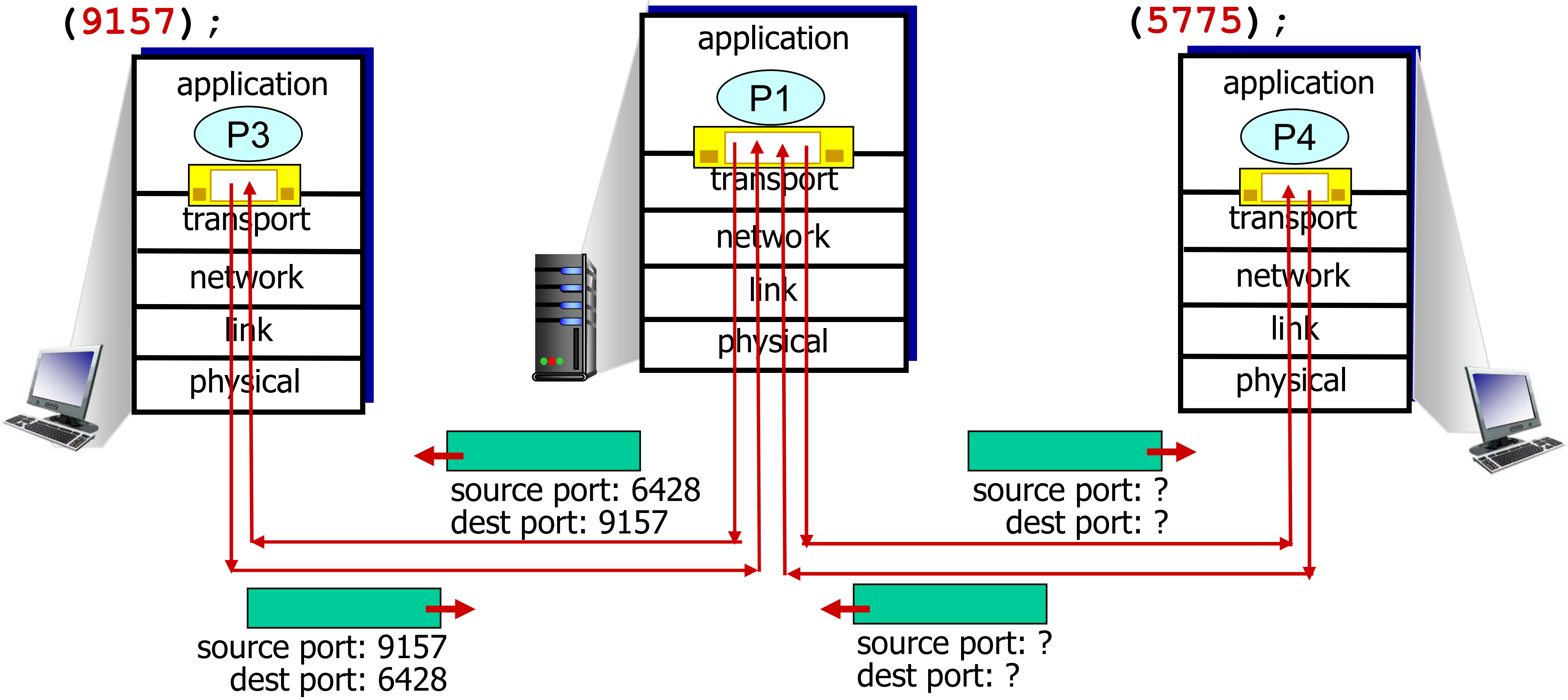
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

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

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

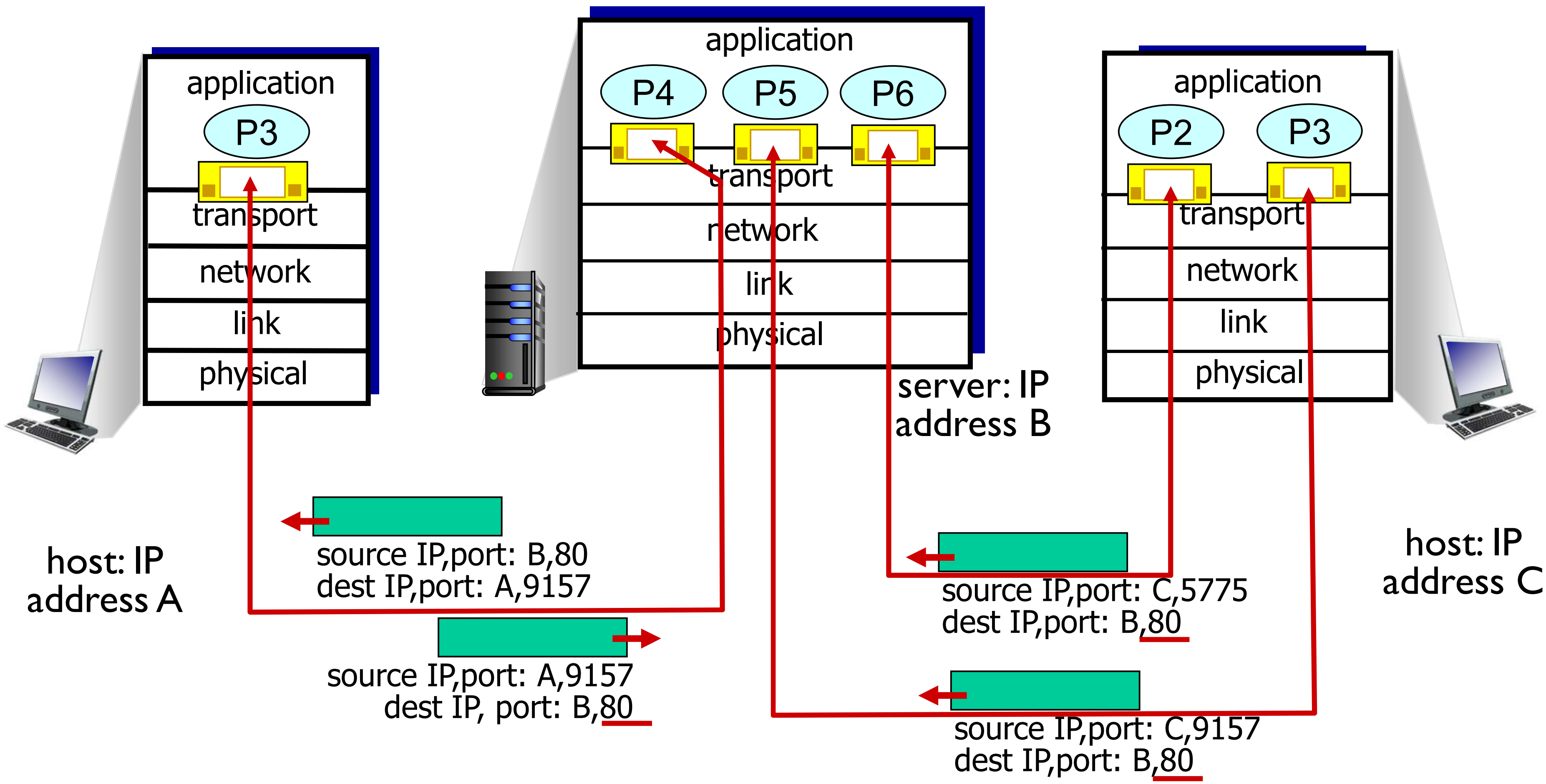
```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



Connection-oriented demux

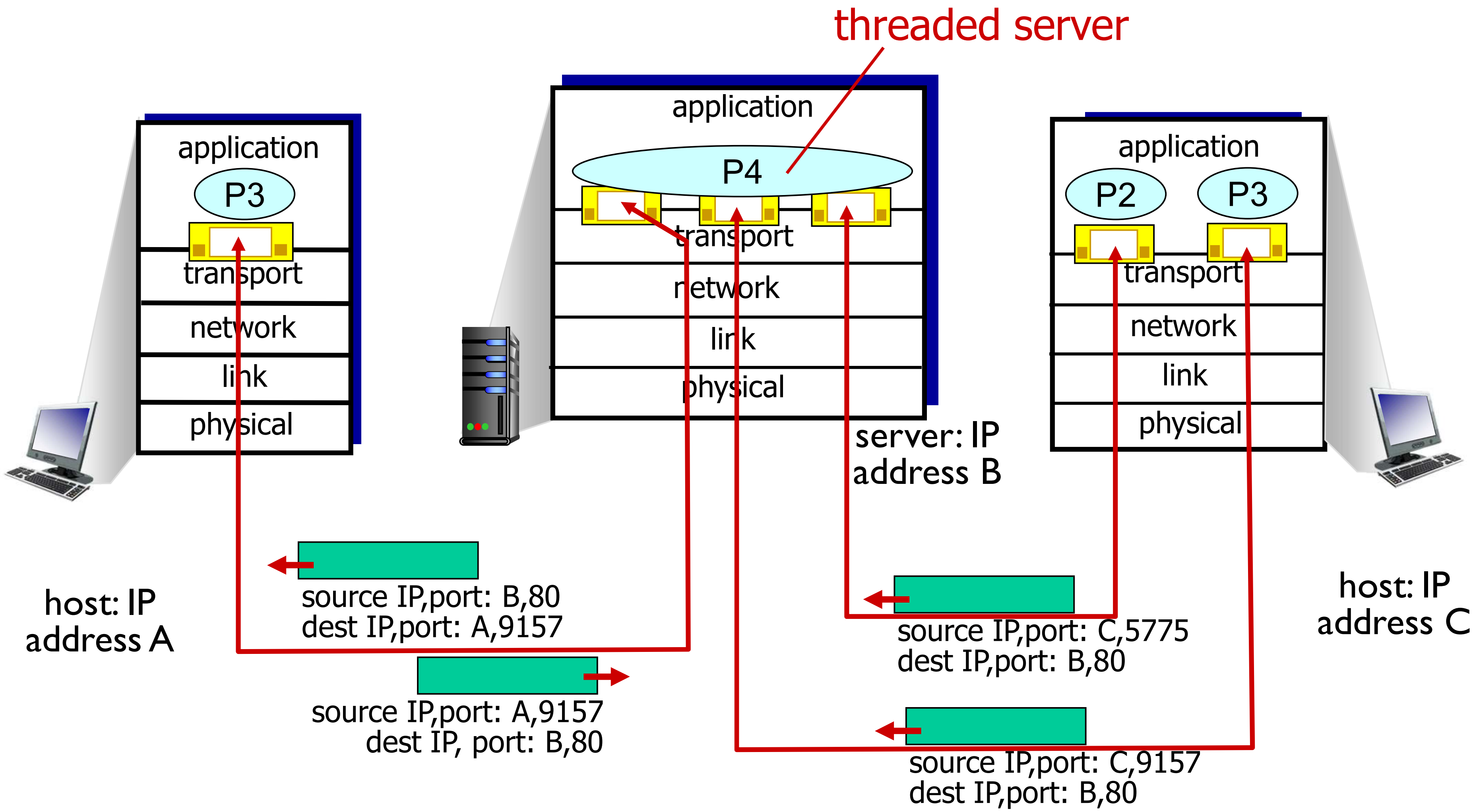
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - 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

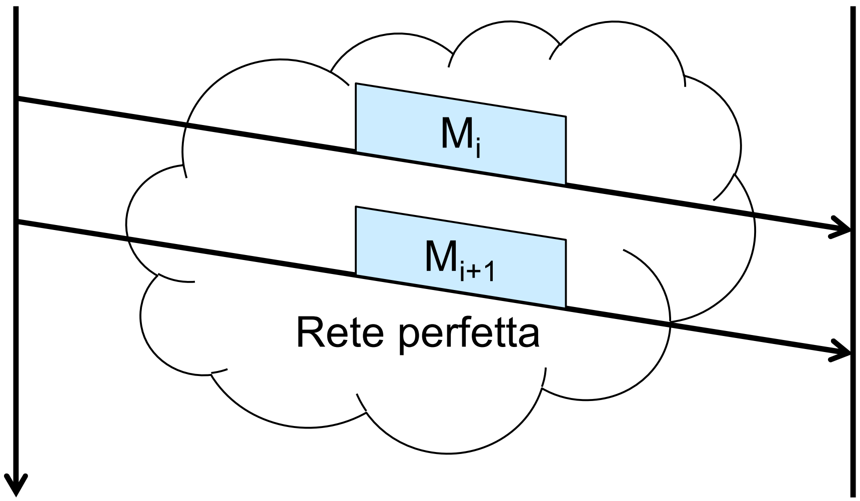


UDP: User Datagram Protocol [RFC 768]

- "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

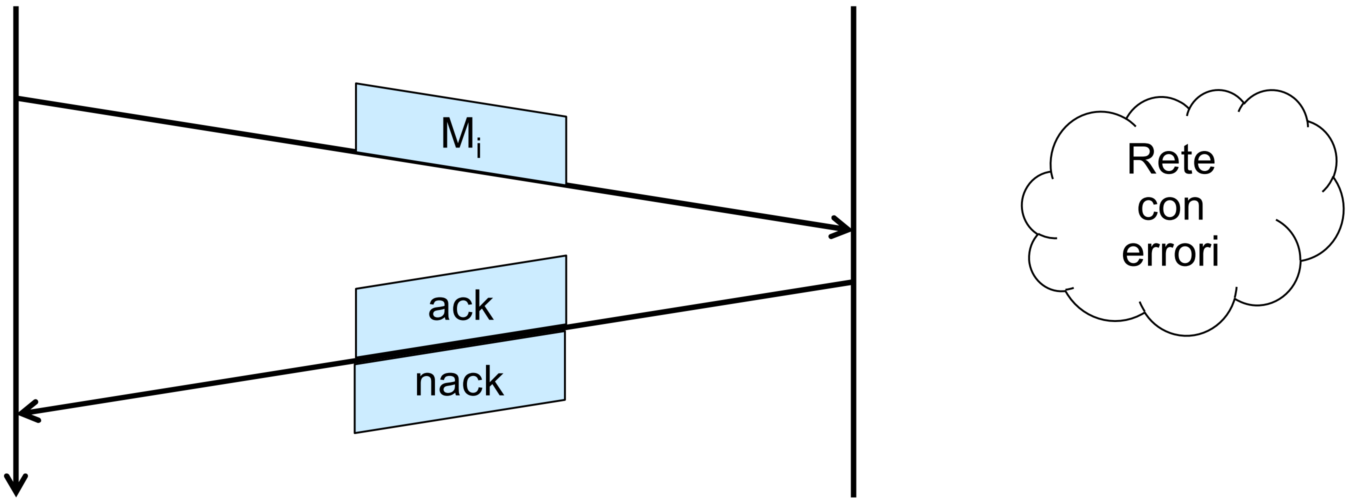
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

Trasporto affidabile (1/10)



- Se la rete è «perfetta», ossia non introduce
  - errori sui bit
  - scarti
  - fuori sequenza
- Lo strato di trasporto non ha nulla da correggere e il protocollo è banale: il sender invia i messaggi uno dopo l’altro e il receiver li riceve tutti senza necessità di controlli

Trasporto affidabile (2/10)

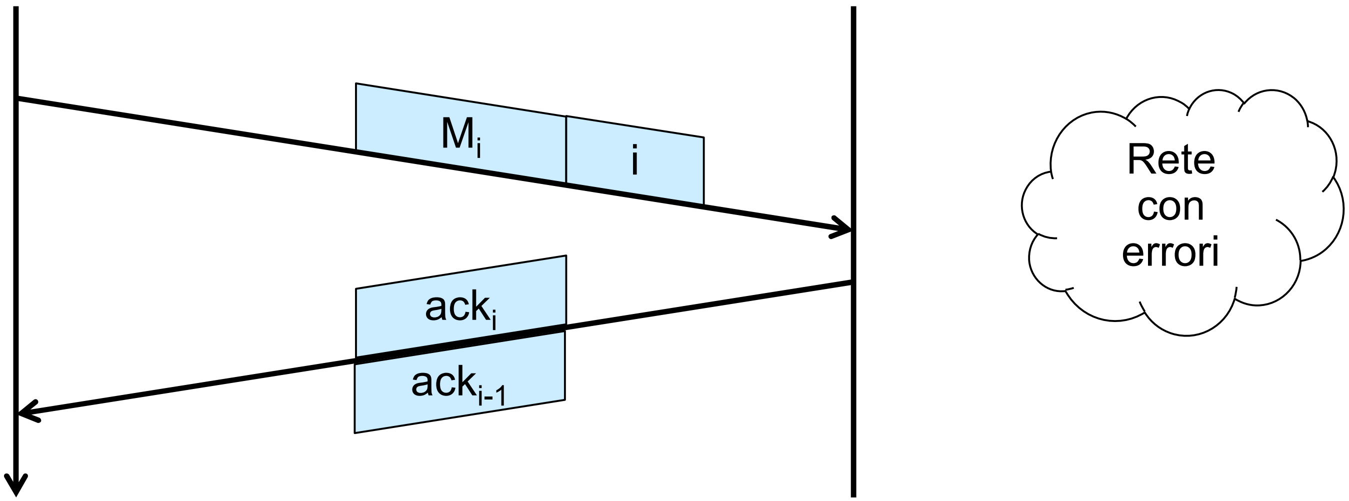


- In una rete con errori posso pensare di introdurre ack positivi e negativi. Ma il semplice algoritmo del sender:

```
IF ack
THEN Mi+1
ELSE IF nack
     THEN Mi
     ELSE ?
```

Non funziona! Perché anche ack/nack possono avere errori!

Trasporto affidabile (3/10)



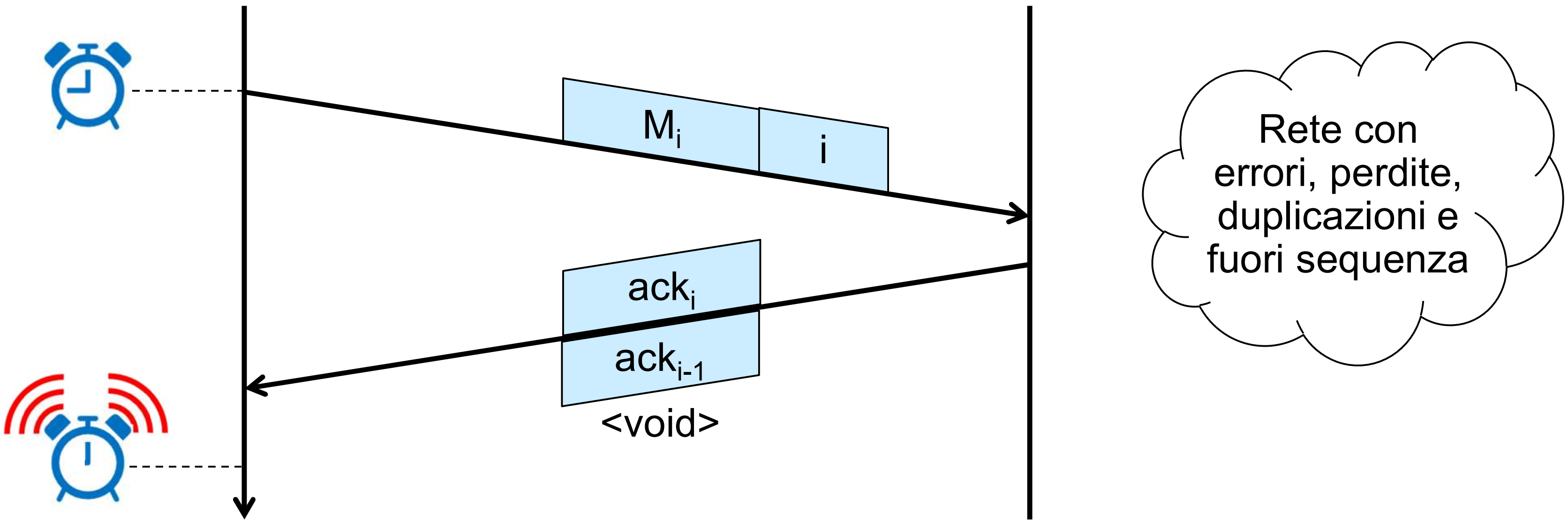
- Se numero i pacchetti non corro rischi anche se invio duplicati:

```
IF ack_i
THEN M_i+1
ELSE M_i
```

Inoltre numerando gli ack posso eliminare la necessità dei nack grazie alla regola: un secondo ack_{i-1} equivale ad un $nack_i$

- Purtroppo le reti con errori e senza perdite non esistono

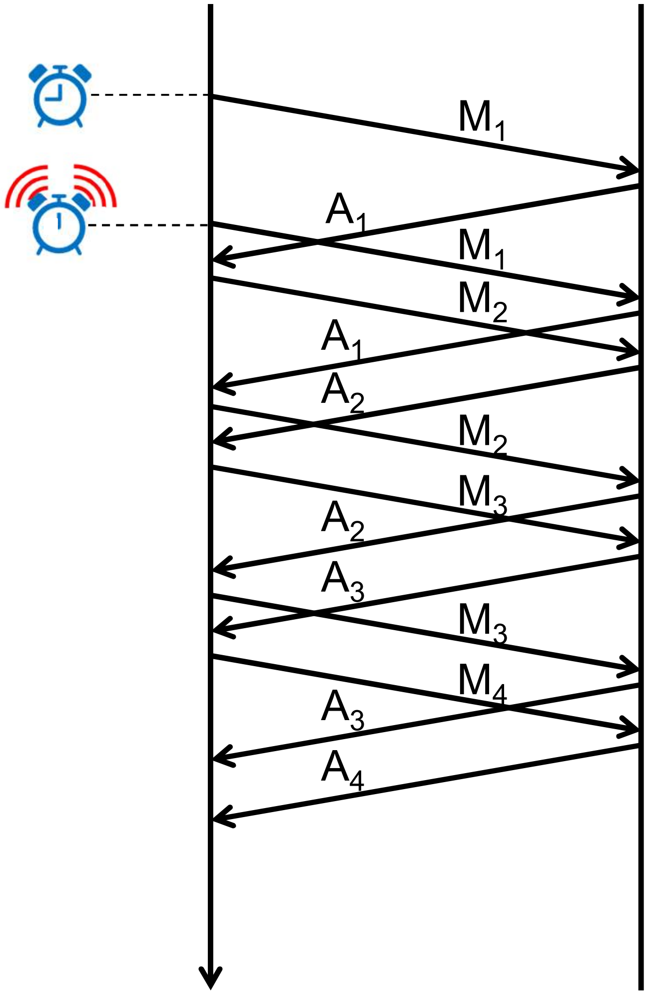
Trasporto affidabile (4/10)



- Se inseriamo un timer possiamo gestire le perdite (del pacchetto o degli ack)

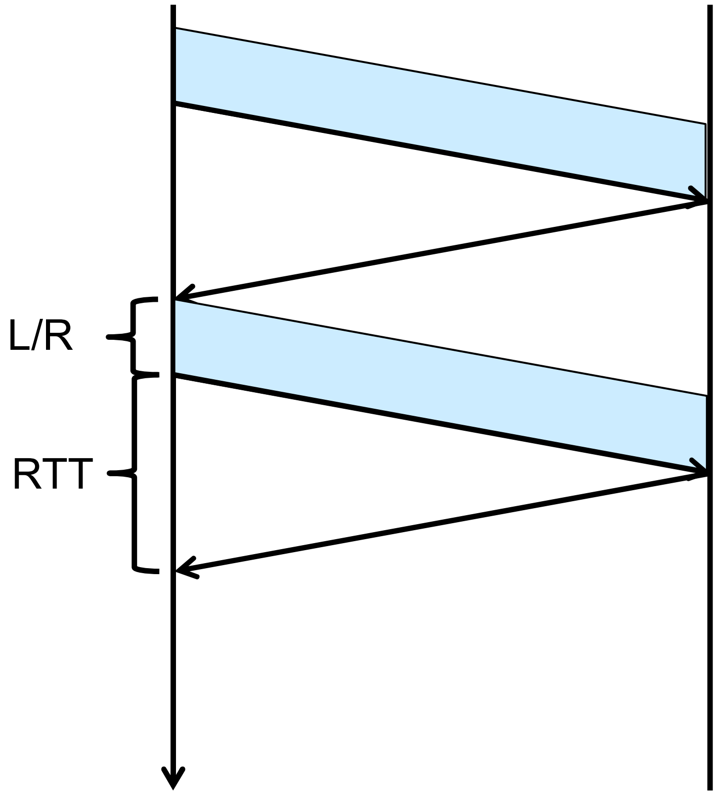
```
IF T-off
THEN Mi,set T
ELSE IF acki
    THEN Mi+1,set T
    ELSE Mi,set T
```

Trasporto affidabile (5/10)



- Fissare la deadline di un timer è un problema complesso
- Un timer troppo corto può generare ritrasmissioni inutili
- Nel caso visto finora (trasmissione e riscontro di un pacchetto per volta) si possono addirittura generare sequenze molto lunghe di ritrasmissioni inutili
- Un timer troppo lungo ferma la trasmissione per troppo tempo nel caso di una perdita

Trasporto affidabile (6/10)



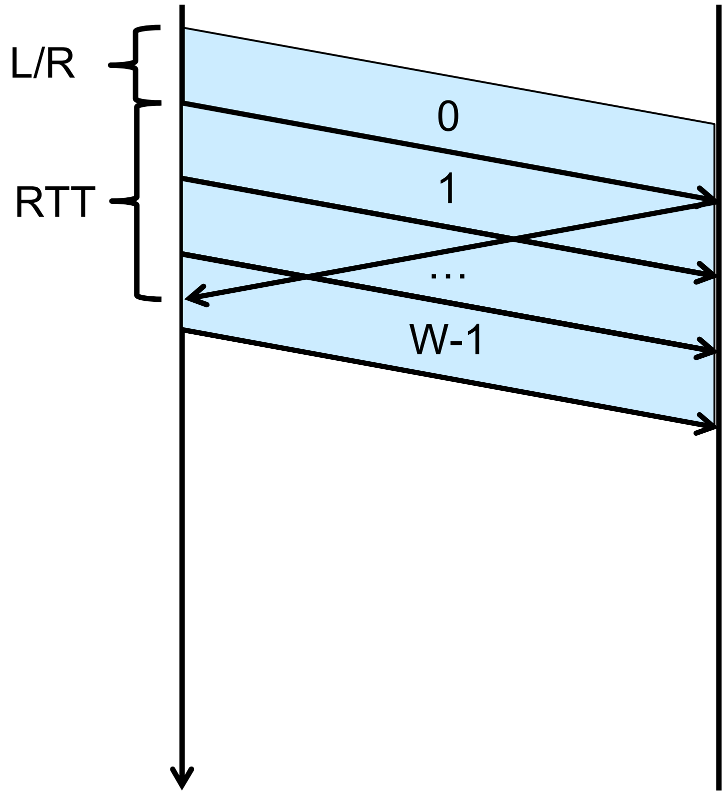
- Trasmettere un pacchetto per volta ed aspettare il riscontro prima del successivo porta a limitare in maniera significativa le prestazioni

$$U = \frac{L/R}{RTT + L/R}$$

- Esempio: RTT=100ms, L=1kbyte, R=100Mbit/s

U=0,0008

Trasporto affidabile (7/10)



- I protocolli a finestra trasmettono fino a W pacchetti in attesa di ricevere il riscontro del primo
- La condizione per una trasmissione continua è che la finestra non si chiuda prima dell'arrivo del primo ack

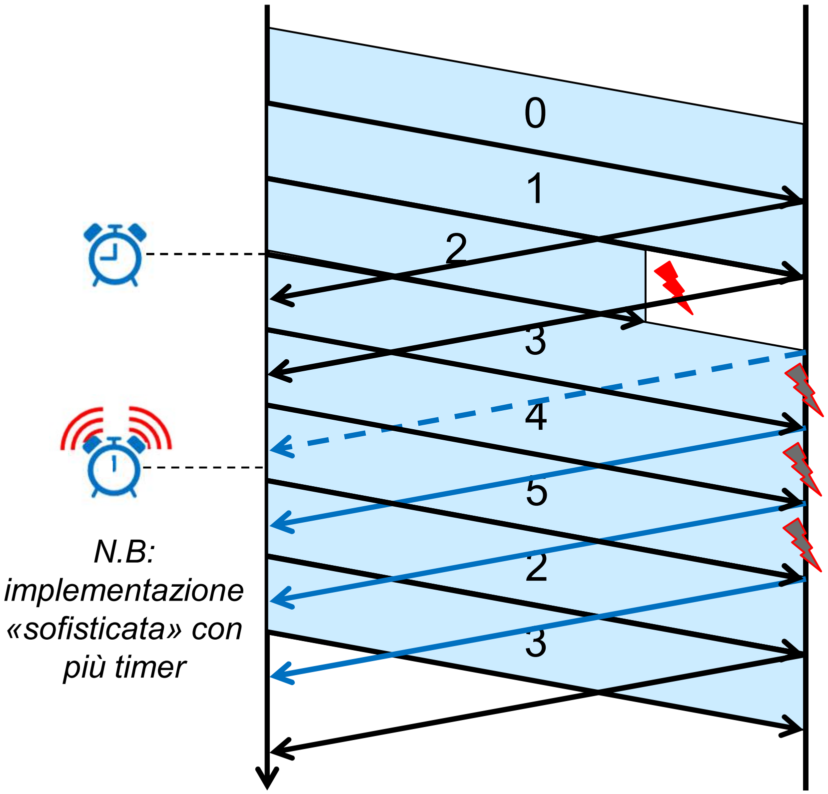
$$W \cdot {}^{L}/_{R} \geq RTT + {}^{L}/_{R}$$

$$W \geq \frac{RTT \cdot R}{L} + 1$$

- Nell'esempio precedente

W≥1251

Trasporto affidabile (8/10)

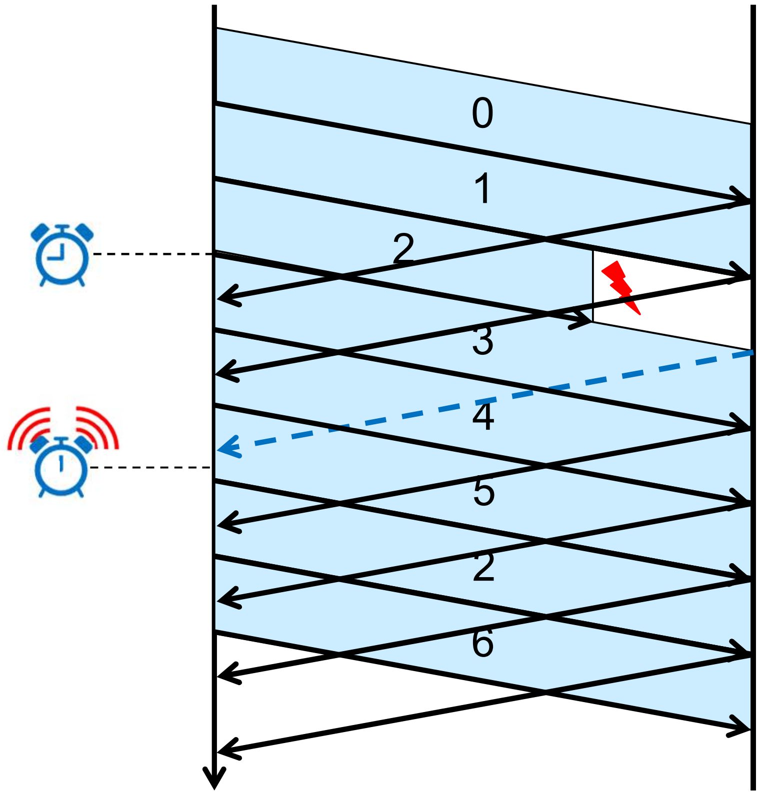


- Cosa ritrasmetto quando perdo un pacchetto?

Go-Back-N

- trasmetto ed avanzo la finestra ad ogni ack
- se T-off o ack ripetuto ritrasmetto tutto dall'ultimo pacchetto riscontrato
- nessun buffer sul ricevitore: i pacchetti ricevuti dopo una perdita vengono scartati fino a quando il pacchetto in sequenza viene ritrasmesso
- gli ack sono cumulativi (si recuperano molti eventi di perdita di ack)

Trasporto affidabile (9/10)



Selective Repeat

- gli ack sono individuali
- ogni pacchetto ha un suo timer
- se T-off ritrasmetto solo il pacchetto perso
- il ricevitore mantiene un buffer in cui i pacchetti vengono risequenziati e inviati in ordine all’applicazione

Trasporto affidabile (10/10)

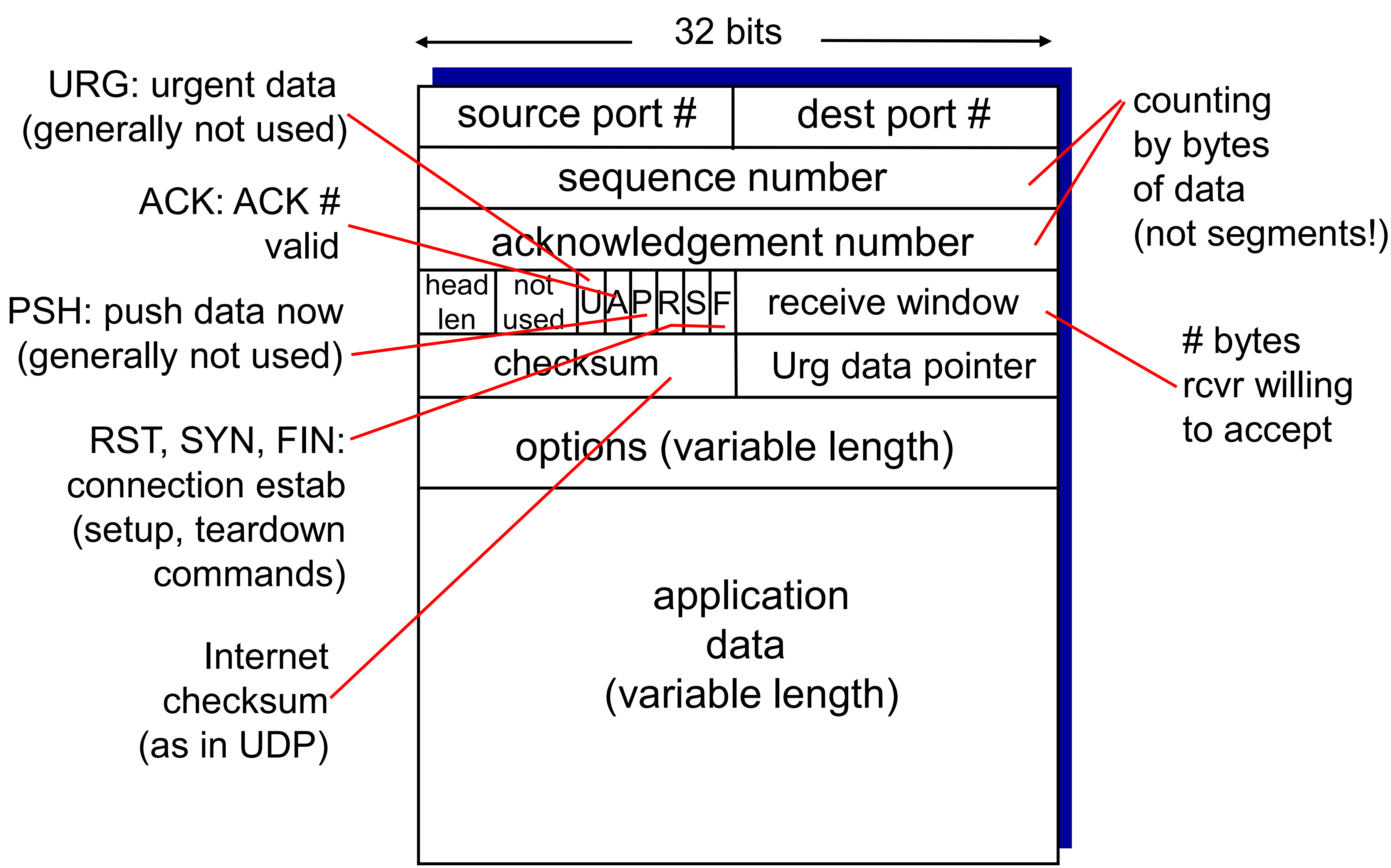
Osservazioni a margine

- Non esistono solo due possibilità (Go-Back-N puro o Selective Repeat puro): i protocolli reali (come TCP) sono spesso un ibrido di soluzioni
- I numeri di sequenza sono rappresentati con un numero finito di bit. Ci sono delle regole che legano il valore massimo del numero di sequenza alla dimensione della finestra e impediscono che dopo un ritorno a zero si creino confusioni
  - Go-Back-N: $N \geq W+1$
  - Selective Repeat: $N \geq 2W$

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

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

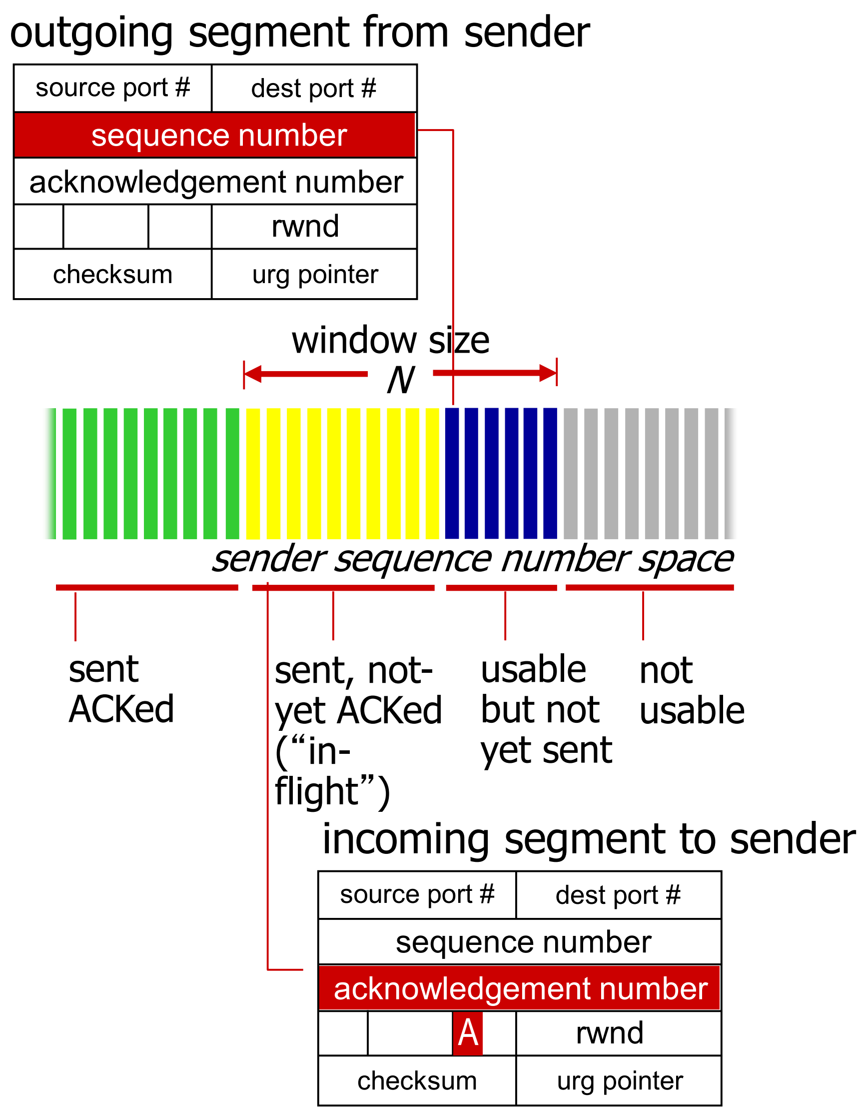
- byte stream "number" of first byte in segment's data

acknowledgements:

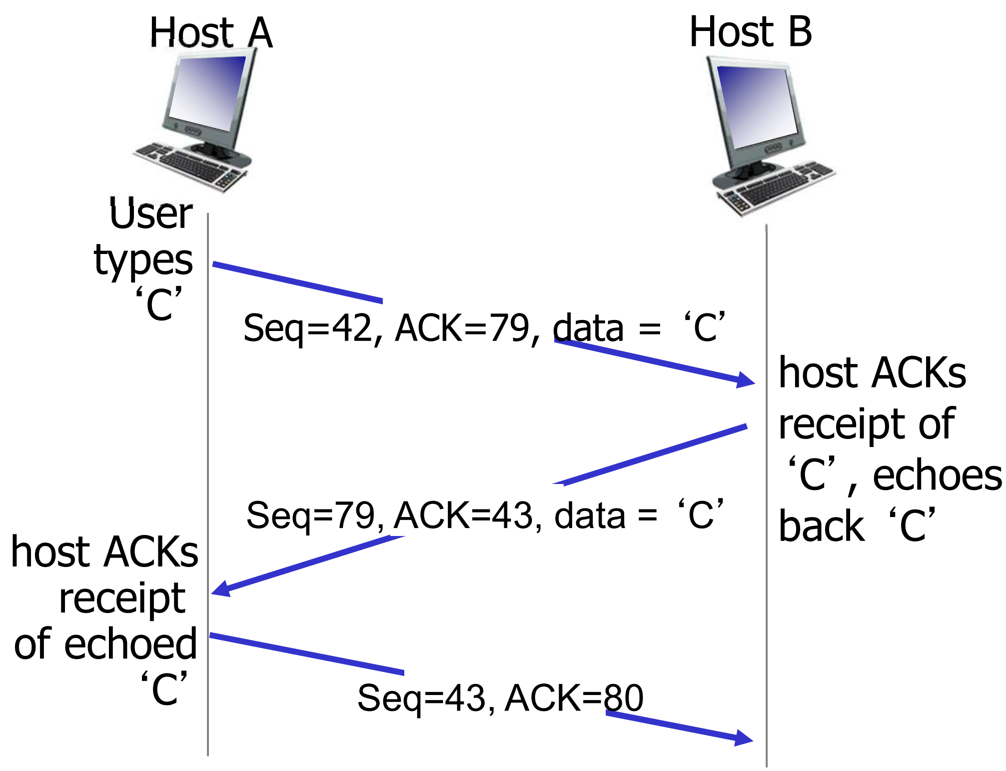
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACKs



simple telnet scenario

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short:* 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 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

let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

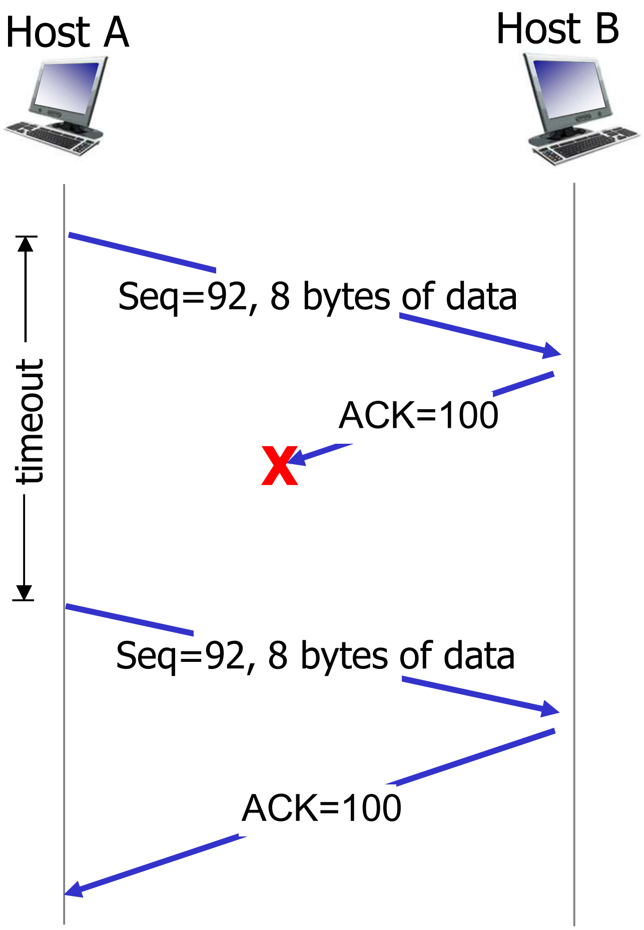
timeout:

- retransmit segment that caused timeout
- restart timer

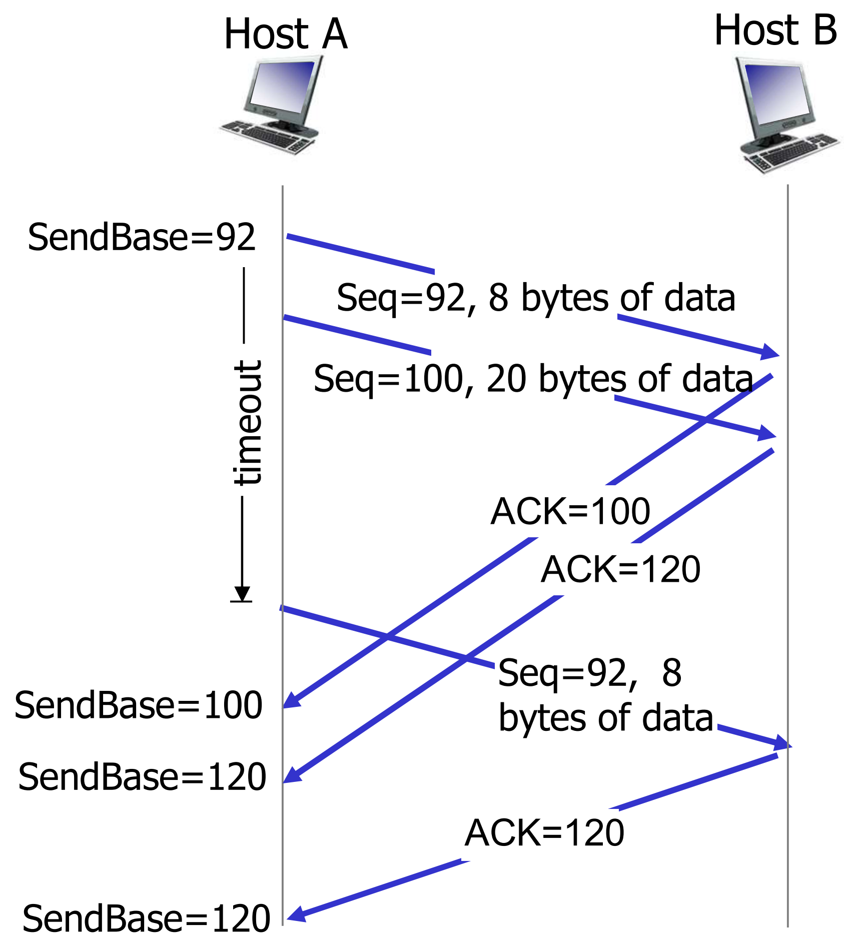
ack rcvd:

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

TCP: retransmission scenarios

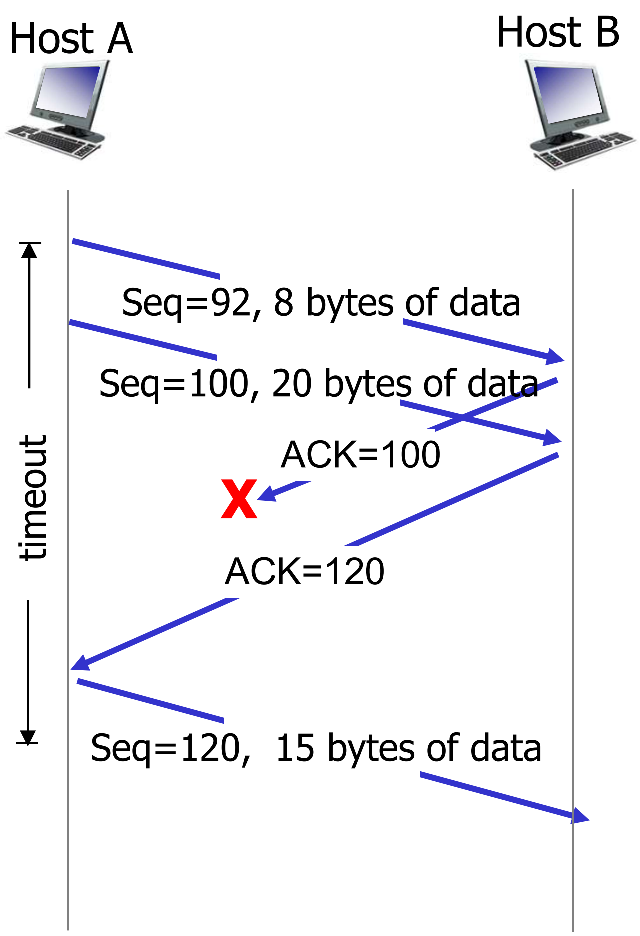


lost ACK scenario



premature timeout

TCP: retransmission scenarios

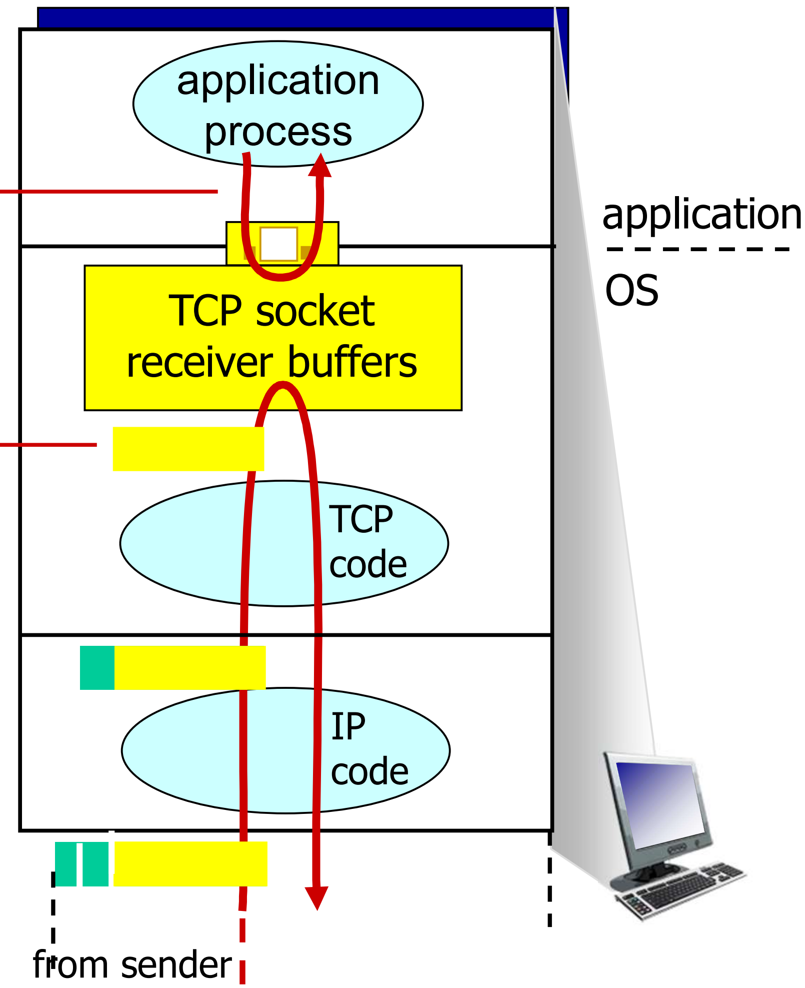


cumulative ACK

TCP flow control

application may remove data from TCP socket buffers ....

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



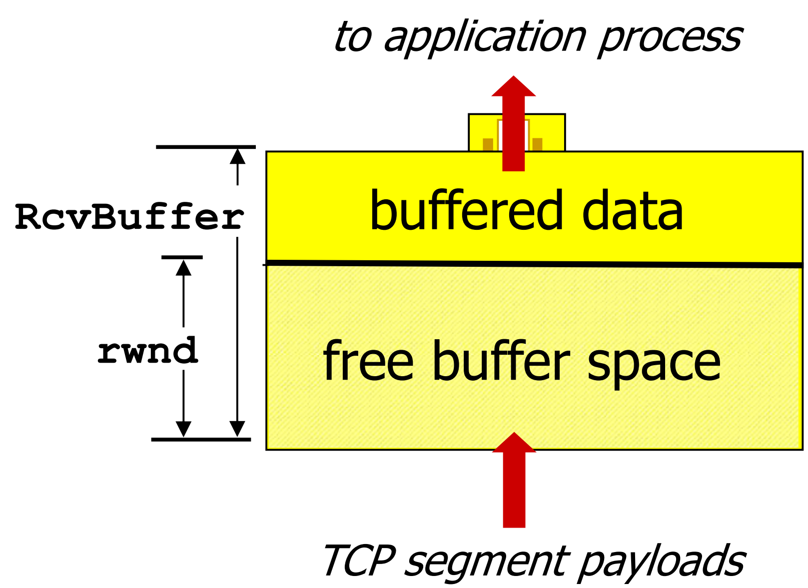
receiver protocol stack

flow control

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

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

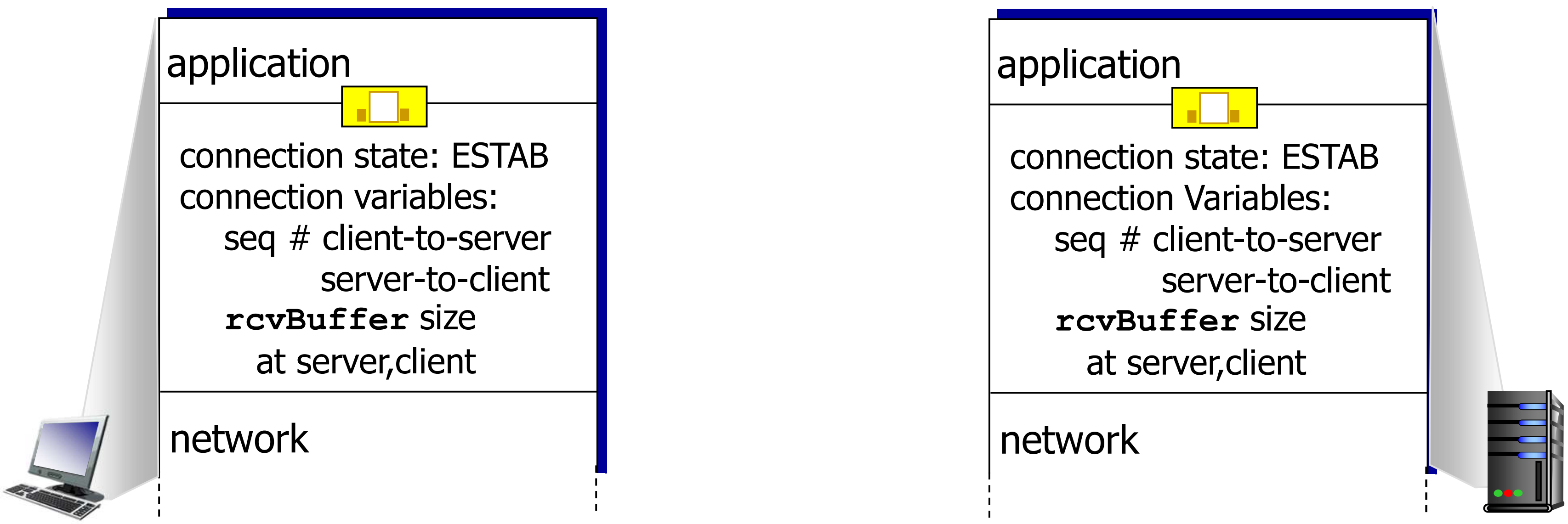


receiver-side buffering

Connection Management

before exchanging data, sender/receiver "handshake":

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

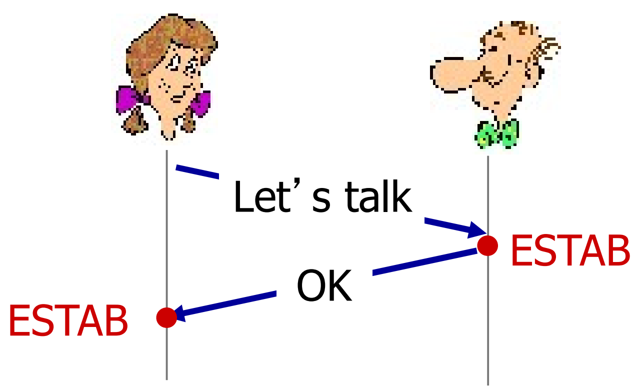


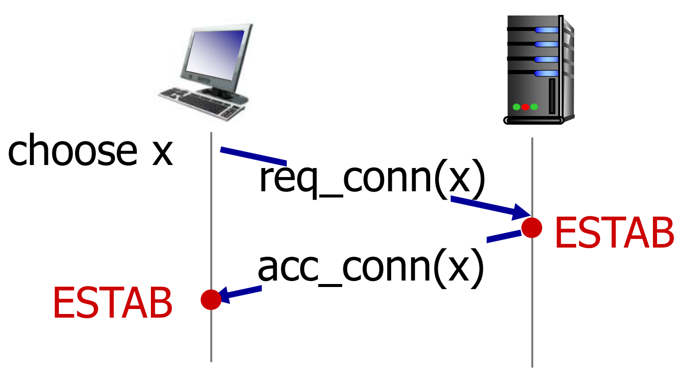

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to establish a connection

2-way handshake:



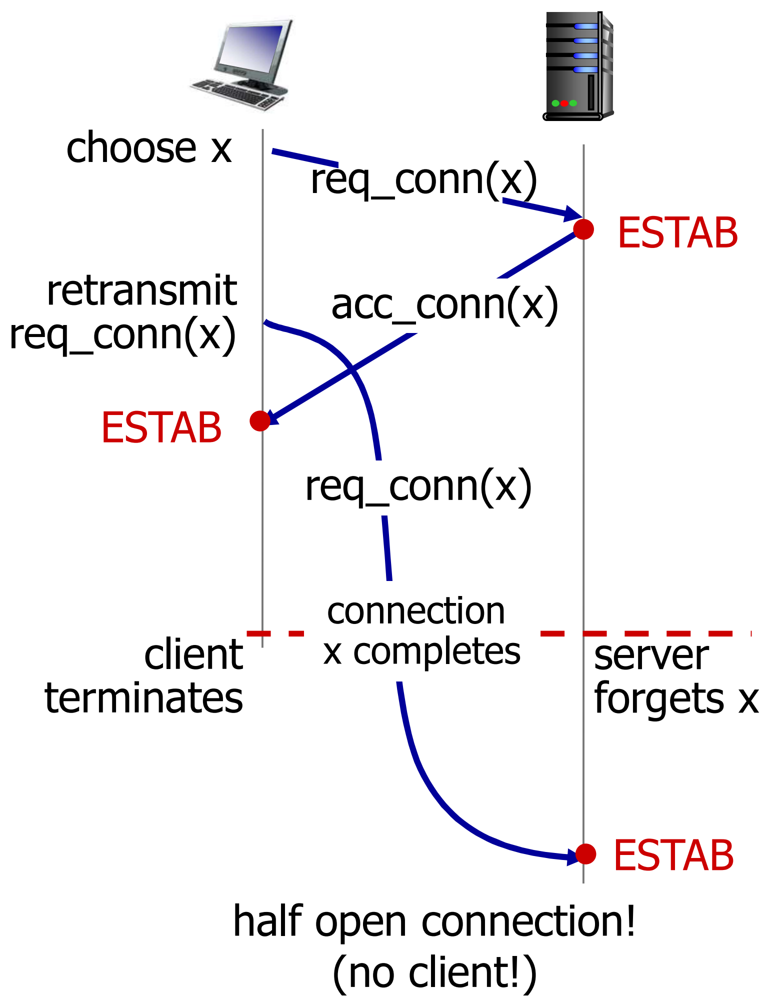


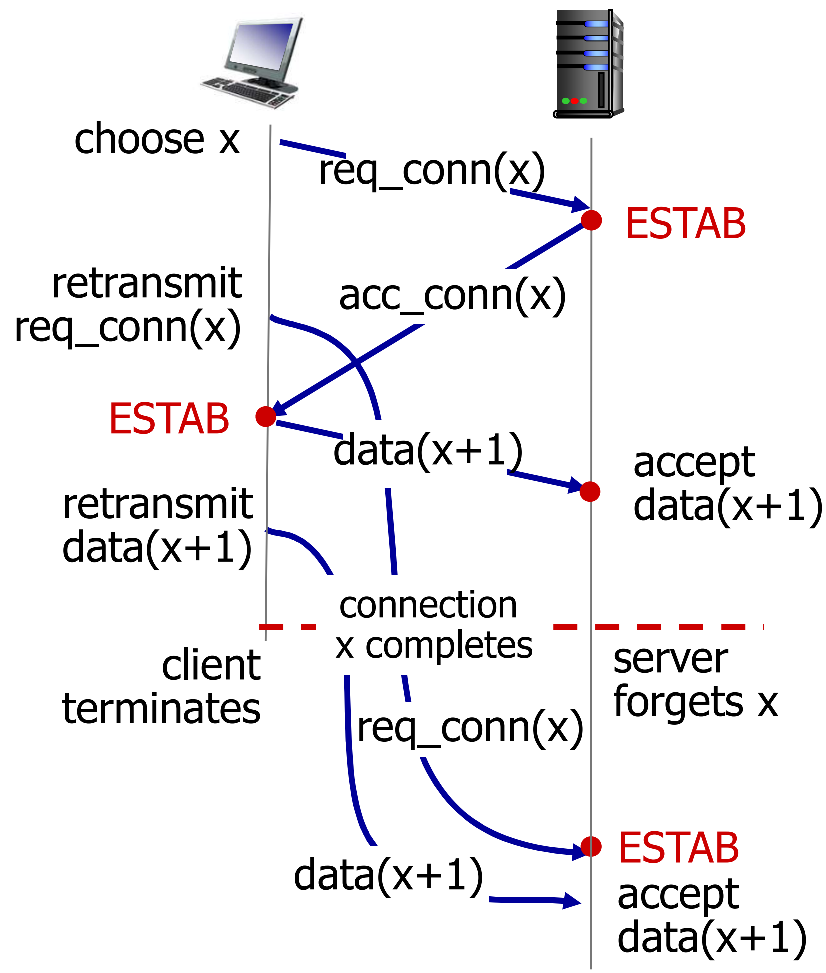
Q: will 2-way handshake always work in network?

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

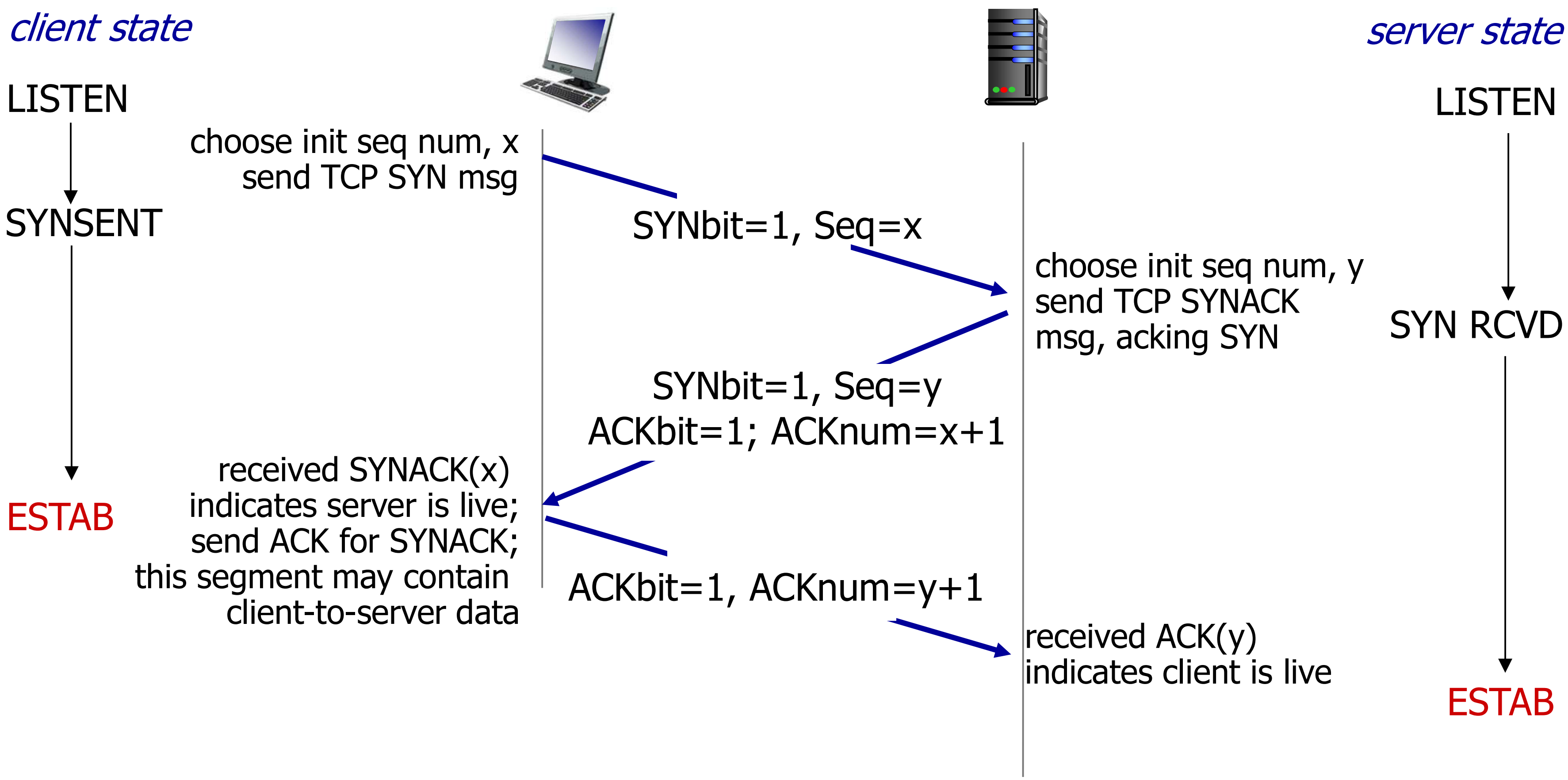
Agreeing to establish a connection

2-way handshake failure scenarios:

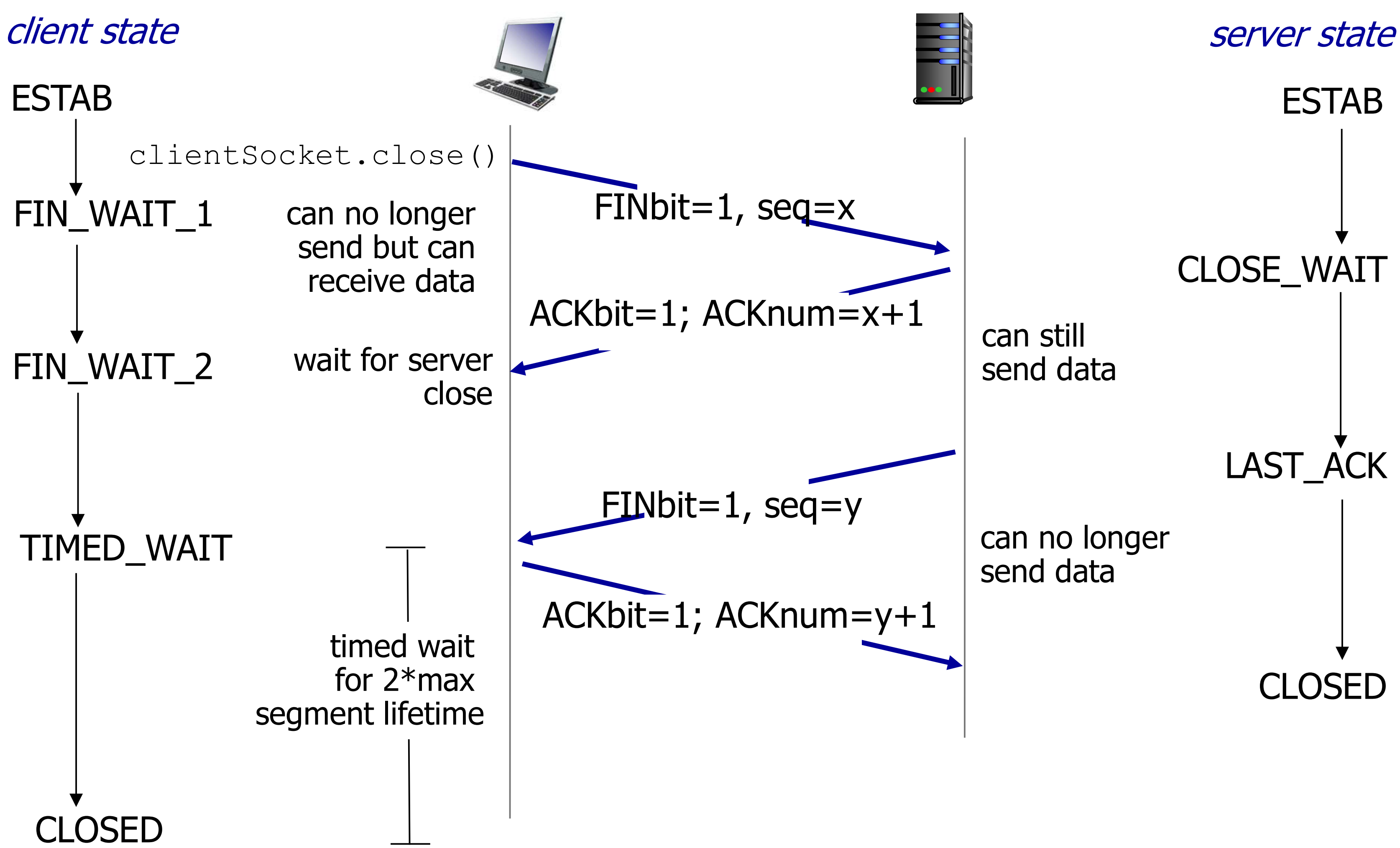




TCP 3-way handshake



TCP: closing a connection



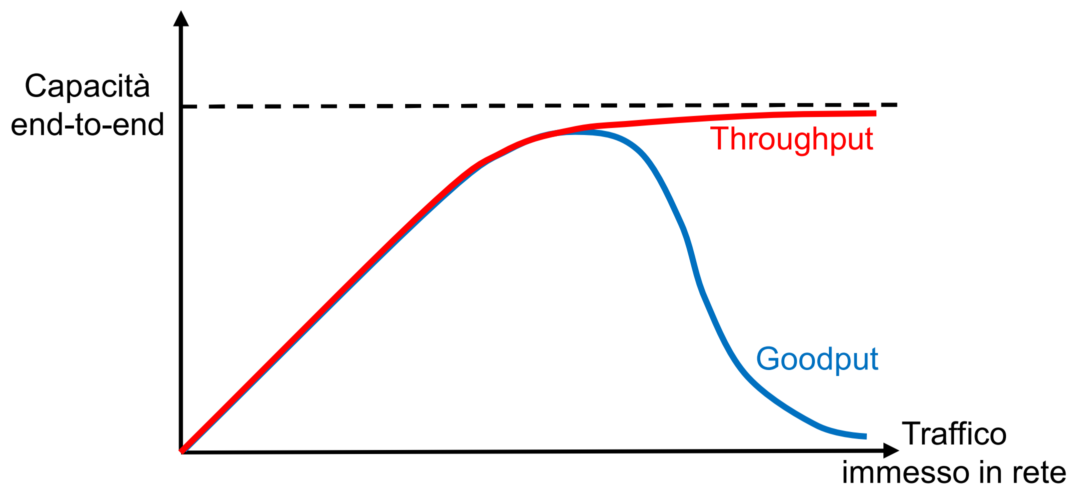
Principles of congestion control

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!

Congestione

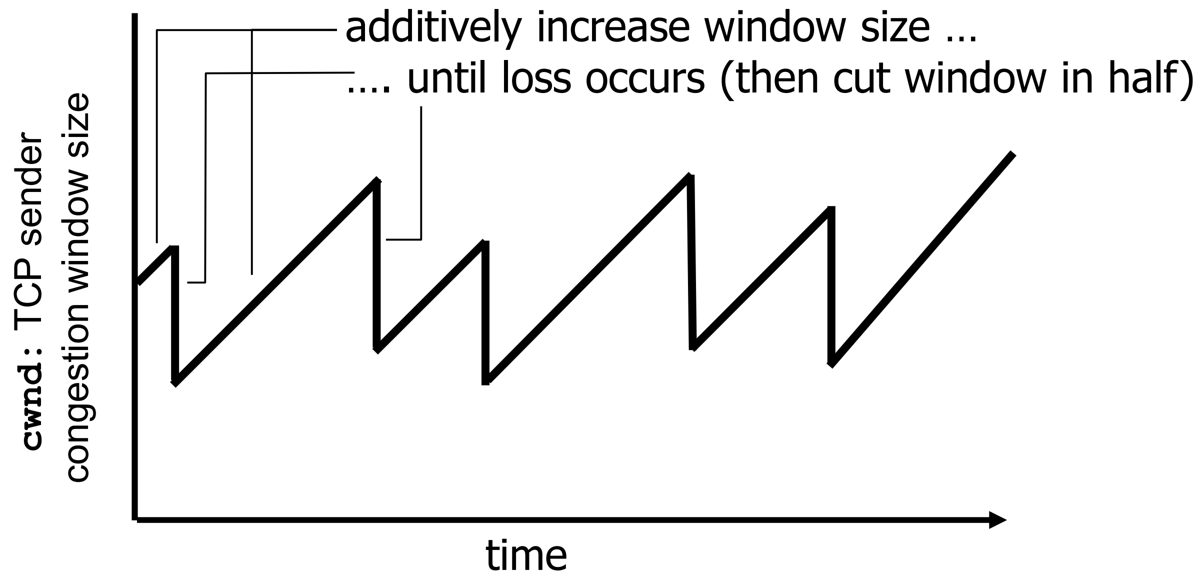
- Se la rete si avvicina alla saturazione delle risorse disponibili (capacità), il ritardo e la percentuale di perdite cresce
- Se il trasporto ritrasmette, aumenta il numero medio di ritrasmissioni di ogni pacchetto
- Mentre il throughput (pacchetti che attraversano la rete) si avvicina al 100% della capacità, il «goodput» visto dall'applicazione decresce !



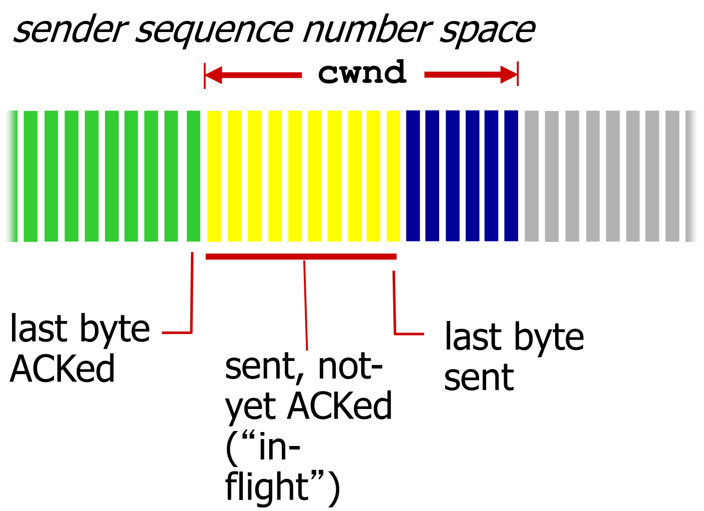
TCP congestion control: additive increase multiplicative decrease

- *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: 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