Process-to-process Data Delivery

Acknowledgements
These Slides have been adapted from the originals made available by J. Kurose and K. Ross

Problem position

- *GOAL: Process-to-process delivery:*
  - logical communication between pairs processes on different hosts
- *Network layer provides* host-to-host delivery
- … but more processes typically run on the same host
- How to fill in the gap??
- *Transport layer*
  - relies on, enhances, network layer services

Goals

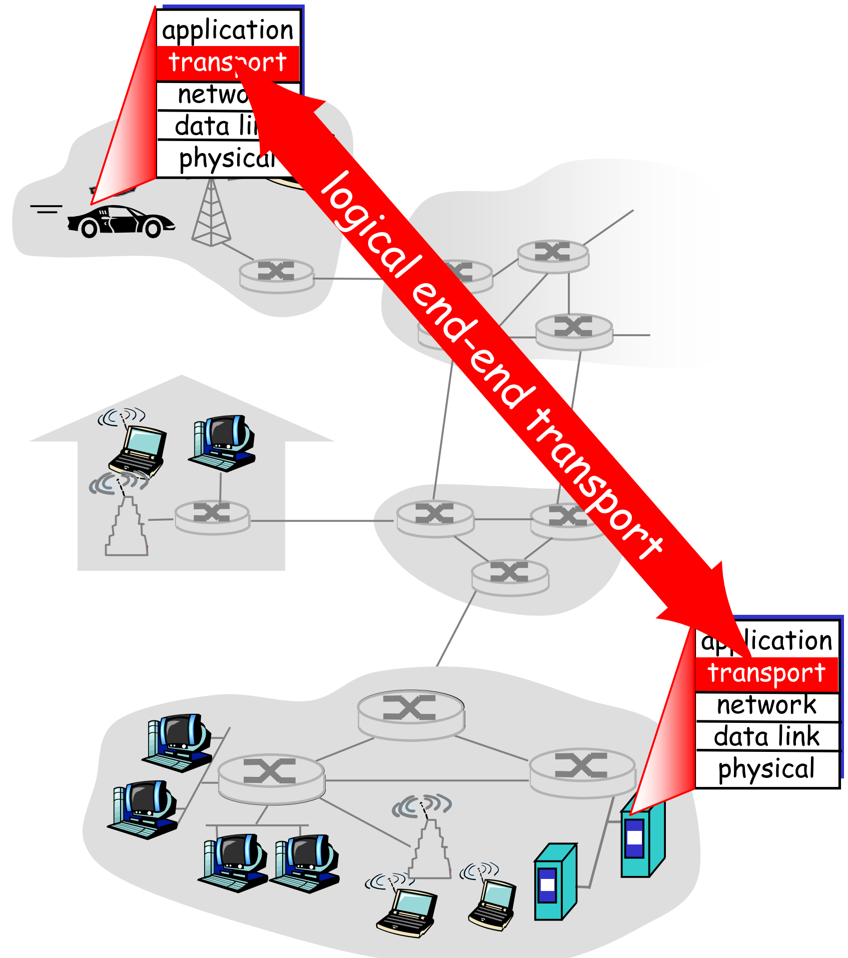
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport

Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - Segment Structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

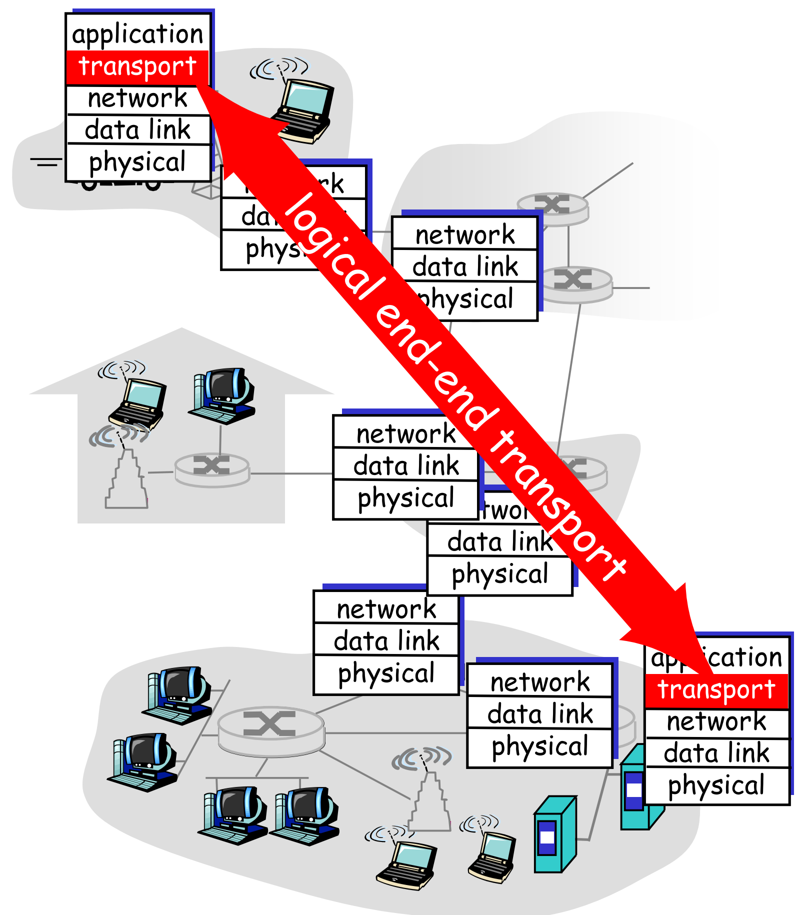
Transport services and protocols

- 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



Internet transport-layer protocols

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



Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - Segment structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

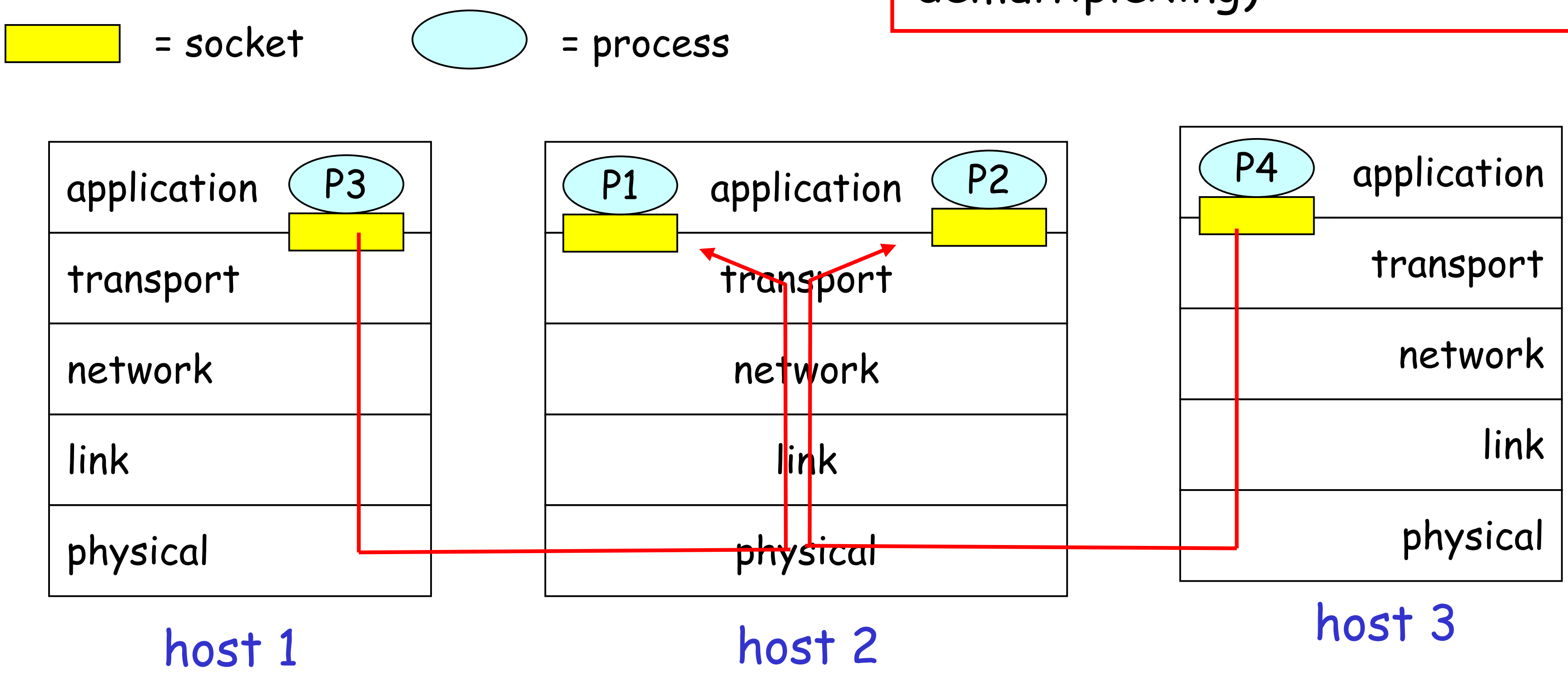
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

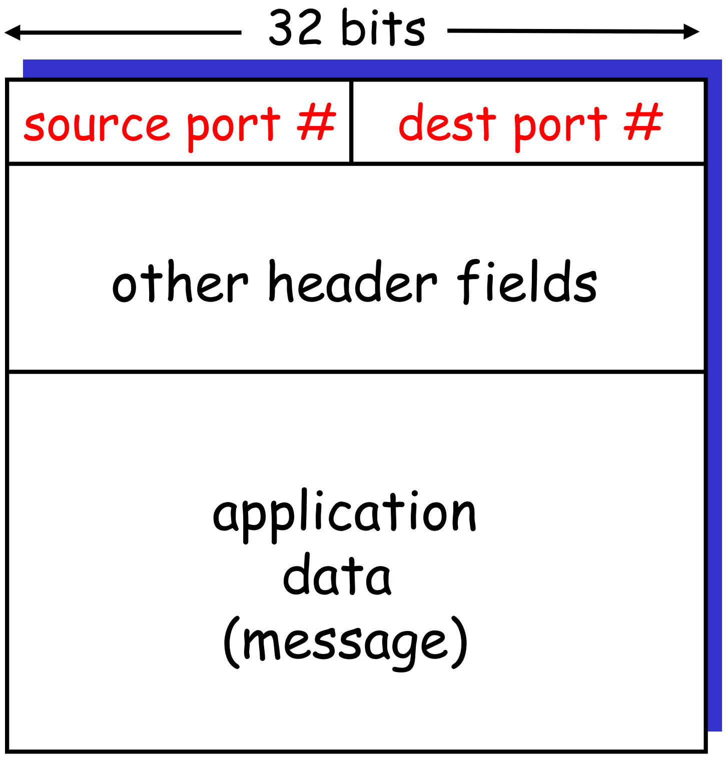
Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each segment has source, destination port number
  - each datagram carries 1 transport-layer segment
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- Datagrams with different source IP addresses and/or port numbers but with the same destination IP address and port number are directed to same socket
- UDP socket identified by a two-tuple:

  (dest IP address, dest port number)

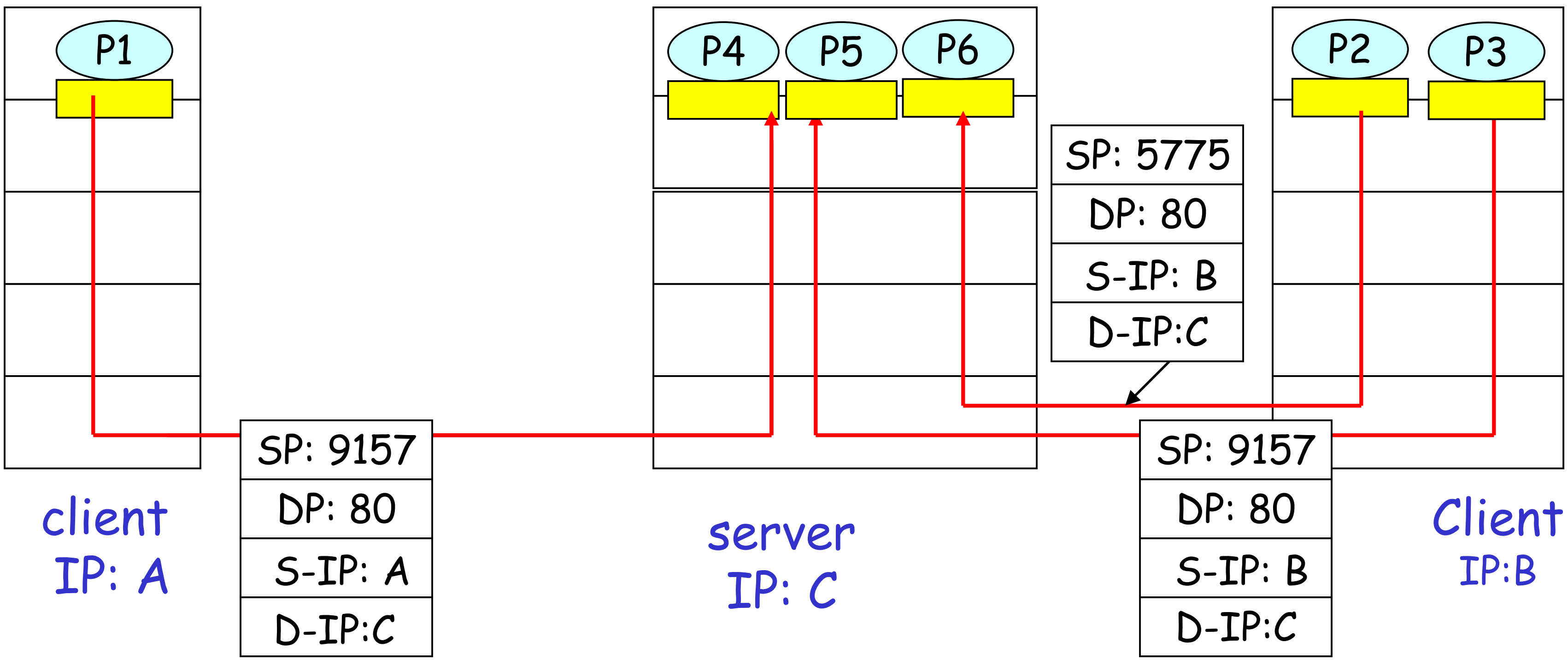
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address, source port number
  - dest IP address, dest port number
- receiving host 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

Multi-process server

```
#include <sys/types.h>
#include <unistd.h>
…
int sd, conn_sd;
struct sockaddr_in srv_addr, cl_addr;
pid_t child_pid;
…
  sd = socket(PF_INET, SOCK_STREAM,0);
  /* srv_addr initialization */
  bind(sd, &srv_addr, sizeof(srv_addr));
  listen(sd,QUEUE_SIZE);

  while(1){
     conn_sd = accept(sd, &cl_addr, sizeof(cl_addr));
     child_pid = fork();
     if(child_pid==0) { /* child process */
              …..
               …..
     }
     else  /* main process */
       close(conn_sd);
  }
```

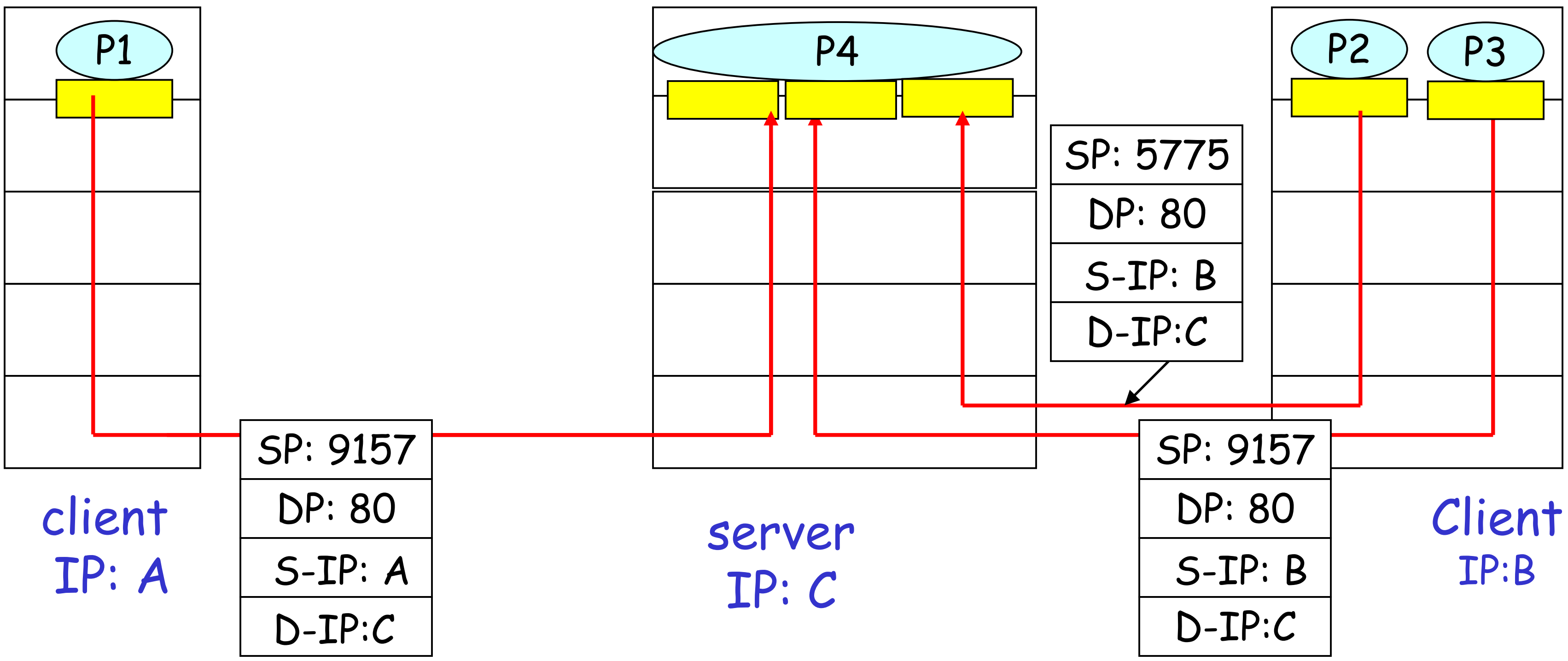
Connection-oriented demux (cont)



Multi-threaded Server

```
#include <sys/types.h>
#include <unistd.h>
…
…
int sd, conn_sd;
struct sockaddr_in srv_addr, cl_addr;
pthread_t tid;
…
 sock = socket(PF_INET, SOCK_STREAM,0);
 /* srv_addr initialization */
 bind(sd, &srv_addr, sizeof(srv_addr));
 listen(sd,QUEUE_SIZE);
 while(1){
     conn_sd = accept(sd, &cl_addr, sizeof(cl_addr));
     pthread_create( &tid, NULL, request_handler, (void*)conn_sd ) )
 }
```

Connection-oriented demux: Threaded Web Server



Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - Segment Structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

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

Why is there a UDP?

- no connection establishment
  - which can add delay
- simple:
  - no connection state at sender, receiver
- finer application-layer control over data
  - no reliability/flow/congestion control
  - UDP can blast away as fast as desired
- small segment header

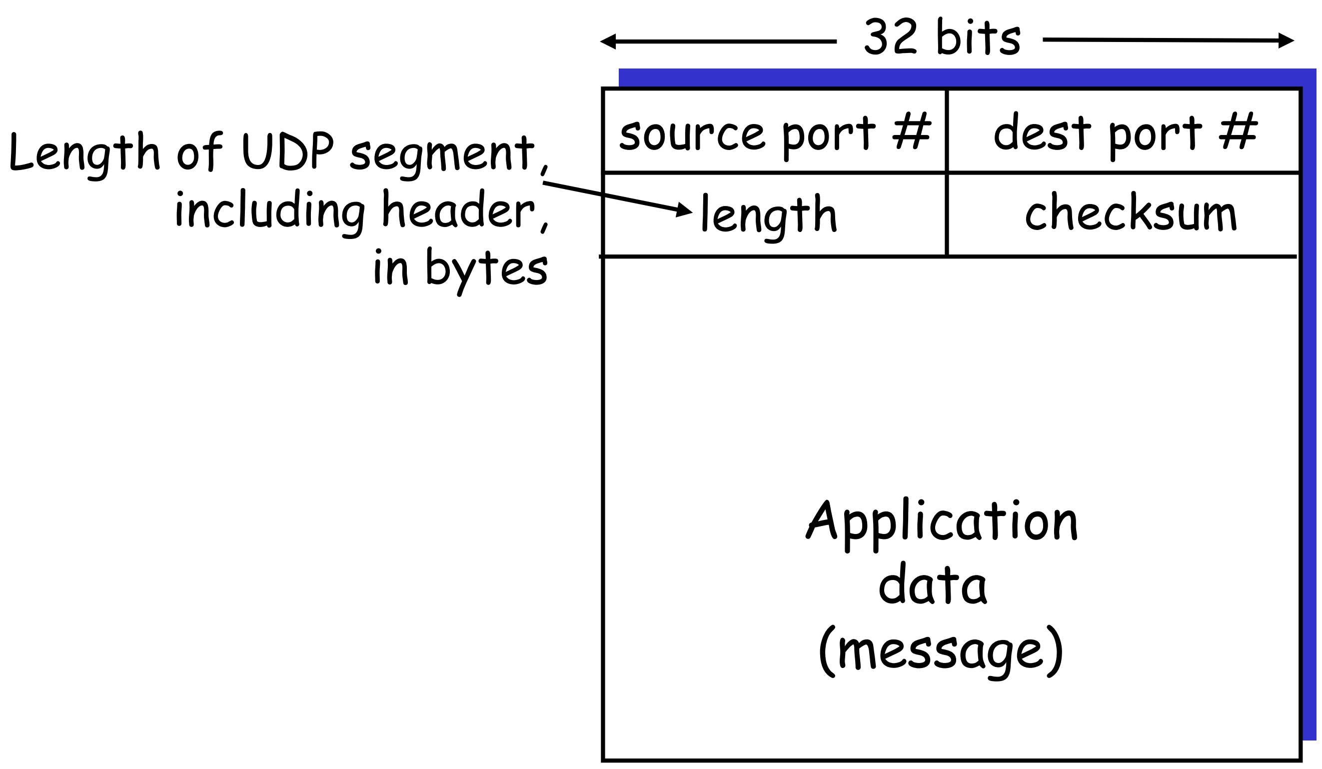
Why is there a UDP?

- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- Other UDP uses
  - DNS
  - NFS
  - SNMP (Simple Network Management Protocol)
  - RIP
- Reliable transfer over UDP
  - add reliability at application layer
  - application-specific error recovery!

Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - Segment structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

UDP Segment Format



UDP checksum

<u>Goal:</u> detect "errors" (e.g., flipped bits) in transmitted segment

<u>Sender:</u>

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

<u>Receiver:</u>

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

Internet Checksum Example

- Note
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

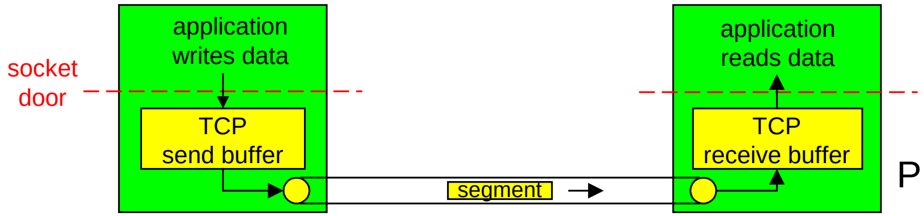
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - Segment structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

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

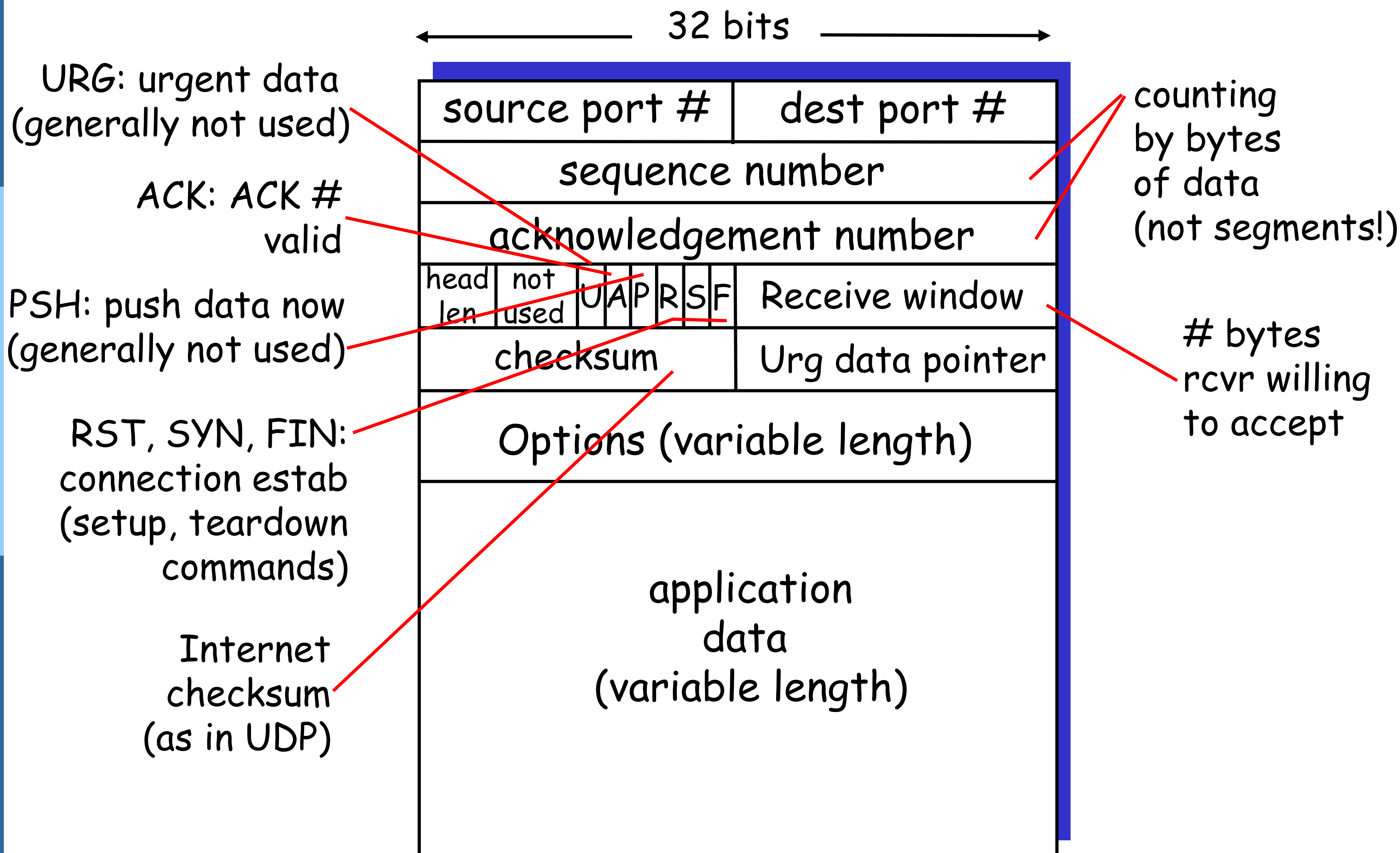
- **connection-oriented:**
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
  - Different from virtual circuit
- **point-to-point:**
  - one sender, one receiver
- **full duplex data:**
  - bi-directional data flow in same connection
- **reliable, in-order *byte stream:***
  - no "message boundaries"
- **Send & receive buffer**
  - MSS: max segment size
- **flow controlled:**
  - sender will not overwhelm receiver
- **pipelined:**
  - TCP congestion and flow control set window size



Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - Segment structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

TCP segment structure



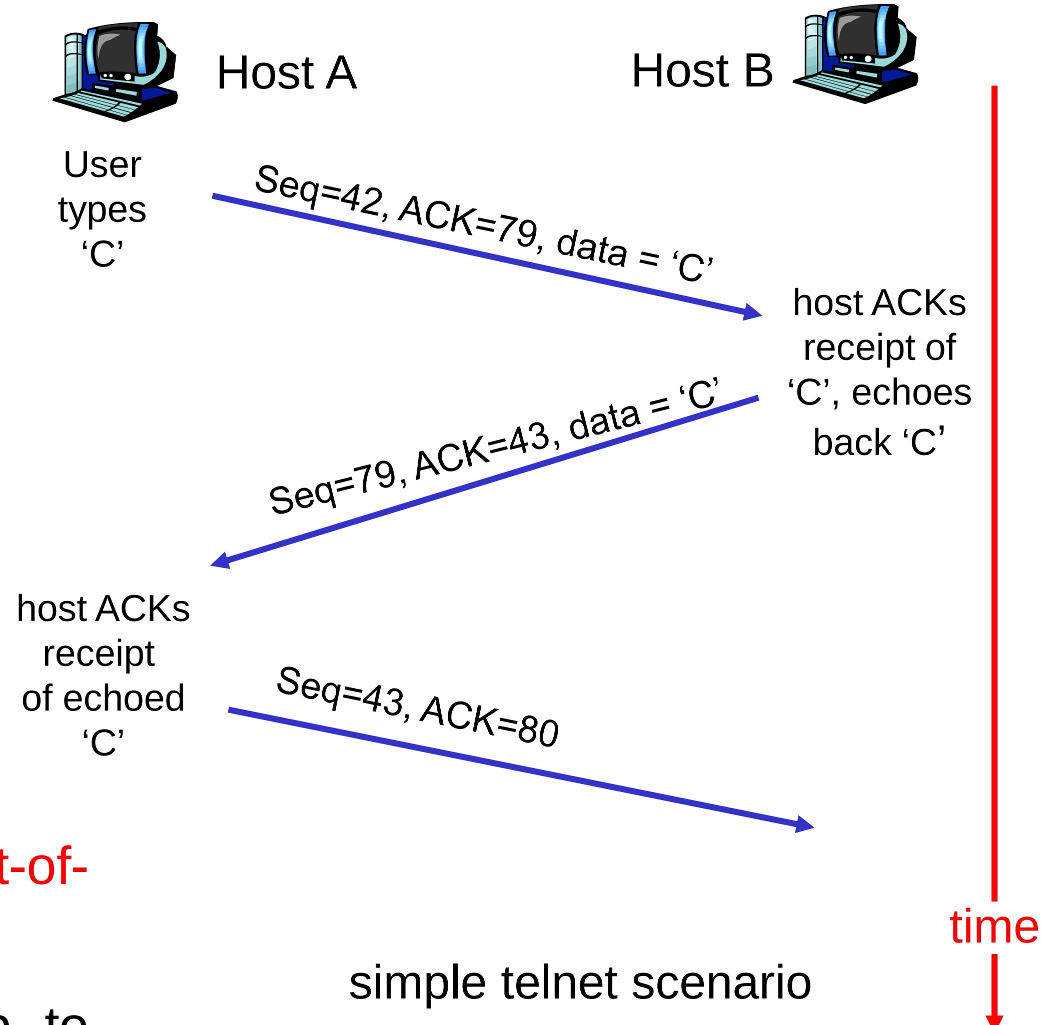
TCP sequence numbers and ACKs

Seq. #'s:

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

ACKs:

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



How receiver handles out-of-order segments?

TCP spec doesn't say, - up to implementer

simple telnet scenario

Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - segment structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

TCP Connection Management

- TCP sender, receiver establish "connection" before exchanging data segments
- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
  - …
- *client:* connection initiator

```
res=connect(sd, …)
```

- *server:* contacted by client

```
conn_sd=accept(sd, …)
```

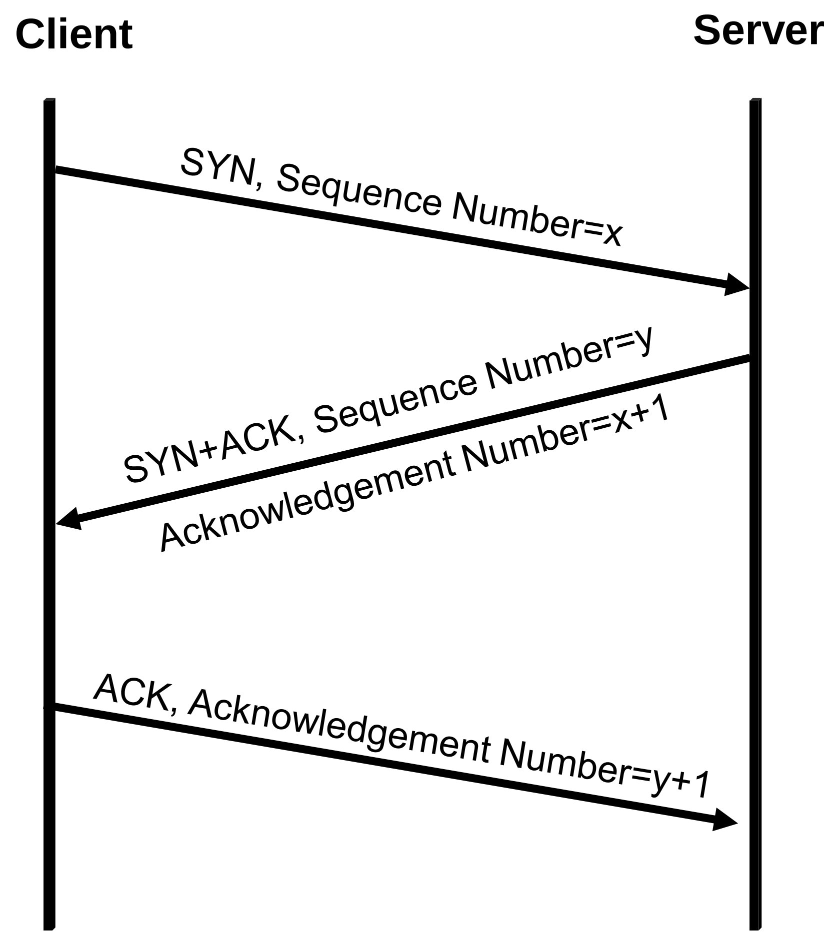
Connection Setup

Three way handskake

1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

2: server host receives SYN, replies with SYN-ACK segment
- server allocates buffers
- specifies server initial seq. #

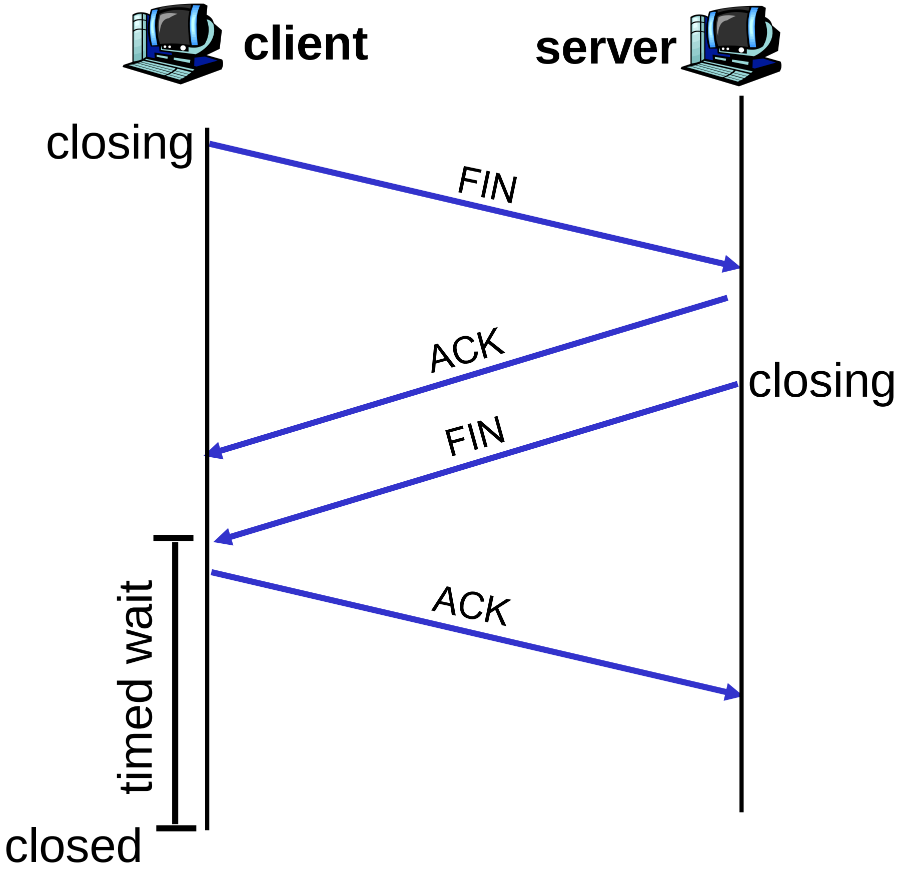
3: client receives SYN-ACK, replies with ACK segment
- may contain data



Connection tear-down

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

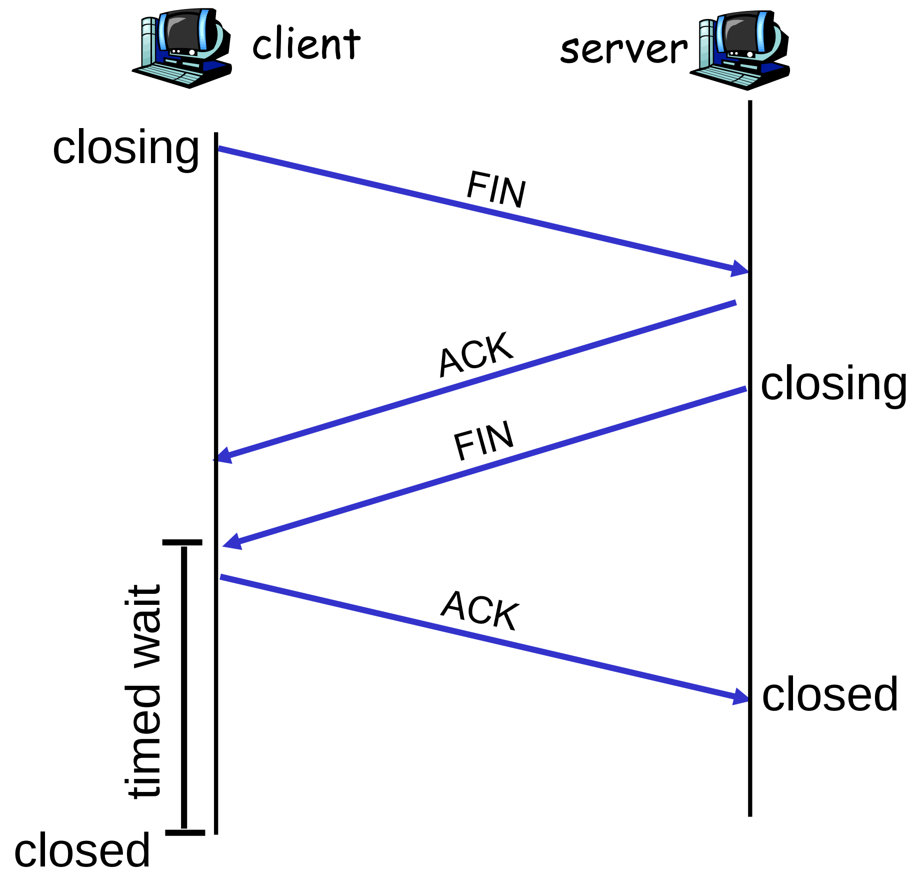


Connection tear-down (cont.)

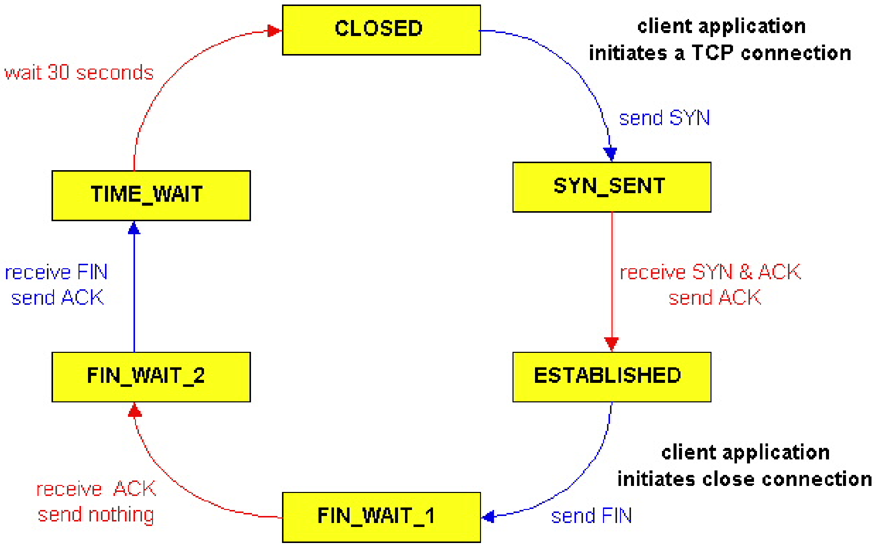
Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

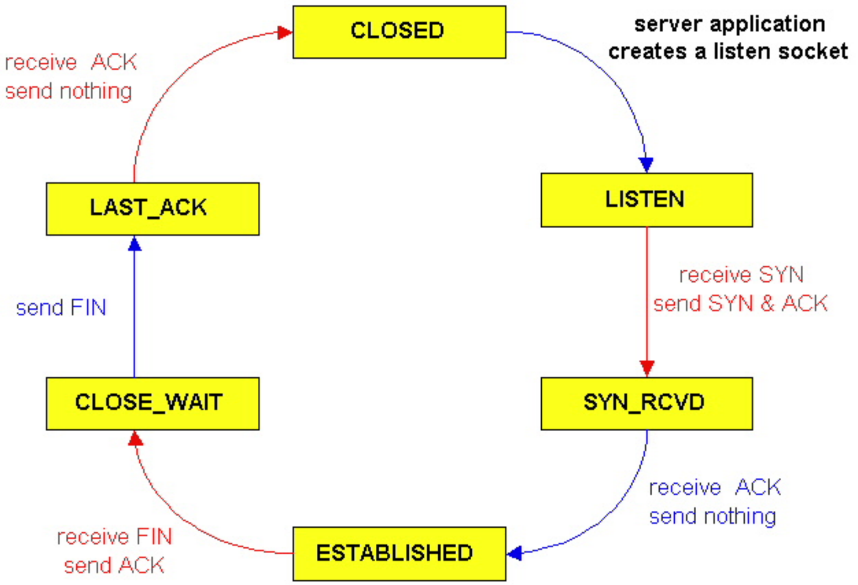


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



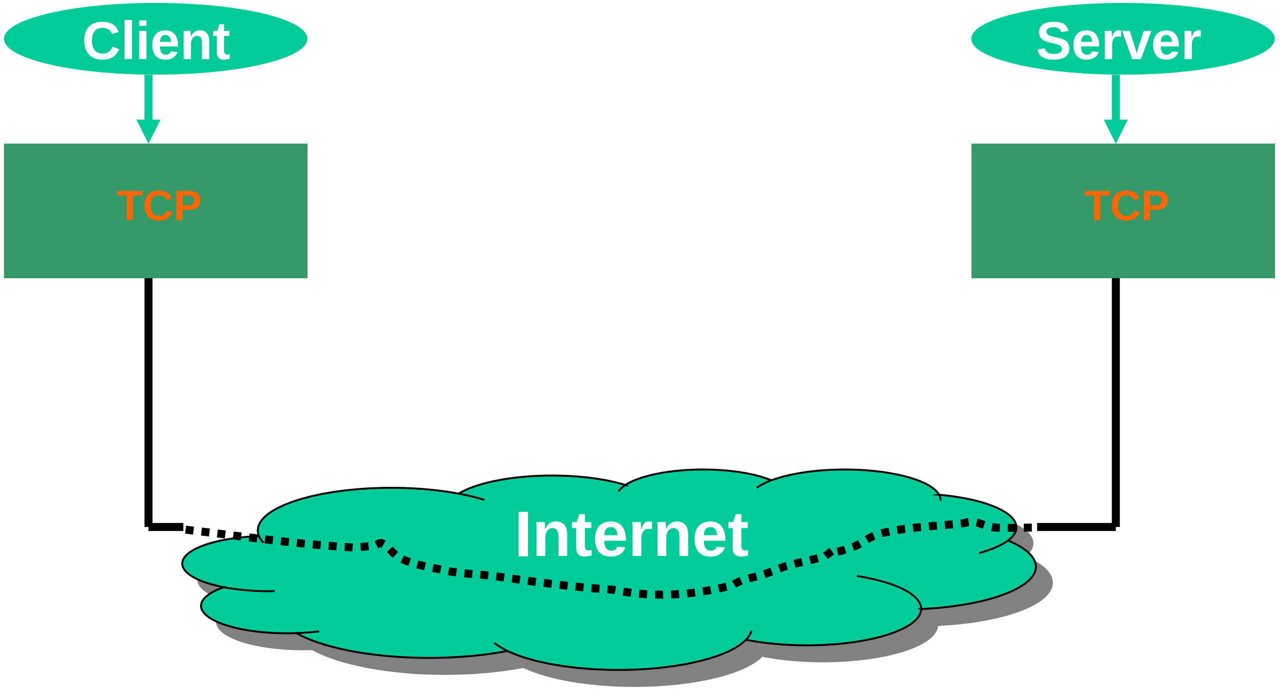
Roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - segment structure
  - connection management
  - reliable data transfer
  - flow control
  - congestion control

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Window-based ARQ scheme (pipeline)
  - Acknowledgements
  - Timeouts and Retransmissions
- How is the Timeout Interval chosen?

TCP Connection



There is a (virtual) connection between the TCP source and destination

TCP Round Trip Time and Timeout

How to set TCP timeout value?

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

How to estimate RTT?

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

RTT Estimate

$$SampleRTT \coloneqq \quad RTT$$
$$EstimatedRTT \coloneqq \quad ERTT$$

$$\alpha < 1$$

$$ERTT_1 = RTT_0$$
$$ERTT_2 = \alpha \cdot RTT_1 + (1 - \alpha) \cdot RTT_0$$
$$ERTT_3 = \alpha \cdot RTT_2 + \alpha(1 - \alpha) \cdot RTT_1 + (1 - \alpha)^2 \cdot RTT_0$$
.....
$$ERTT_{n+1} = \alpha \cdot RTT_n + \alpha(1 - \alpha) \cdot RTT_{n-1} + \alpha(1 - \alpha)^2 \cdot RTT_{n-2} + \cdots + (1 - \alpha)^n \cdot RTT_0$$

$$\Downarrow$$

$$ERTT_{n+1} = \alpha \cdot RTT_n + (1 - \alpha) \cdot [\alpha \cdot RTT_{n-1} + \alpha(1 - \alpha) \cdot RTT_{n-2} + \cdots + (1 - \alpha)^{n-1} \cdot RTT_0]$$

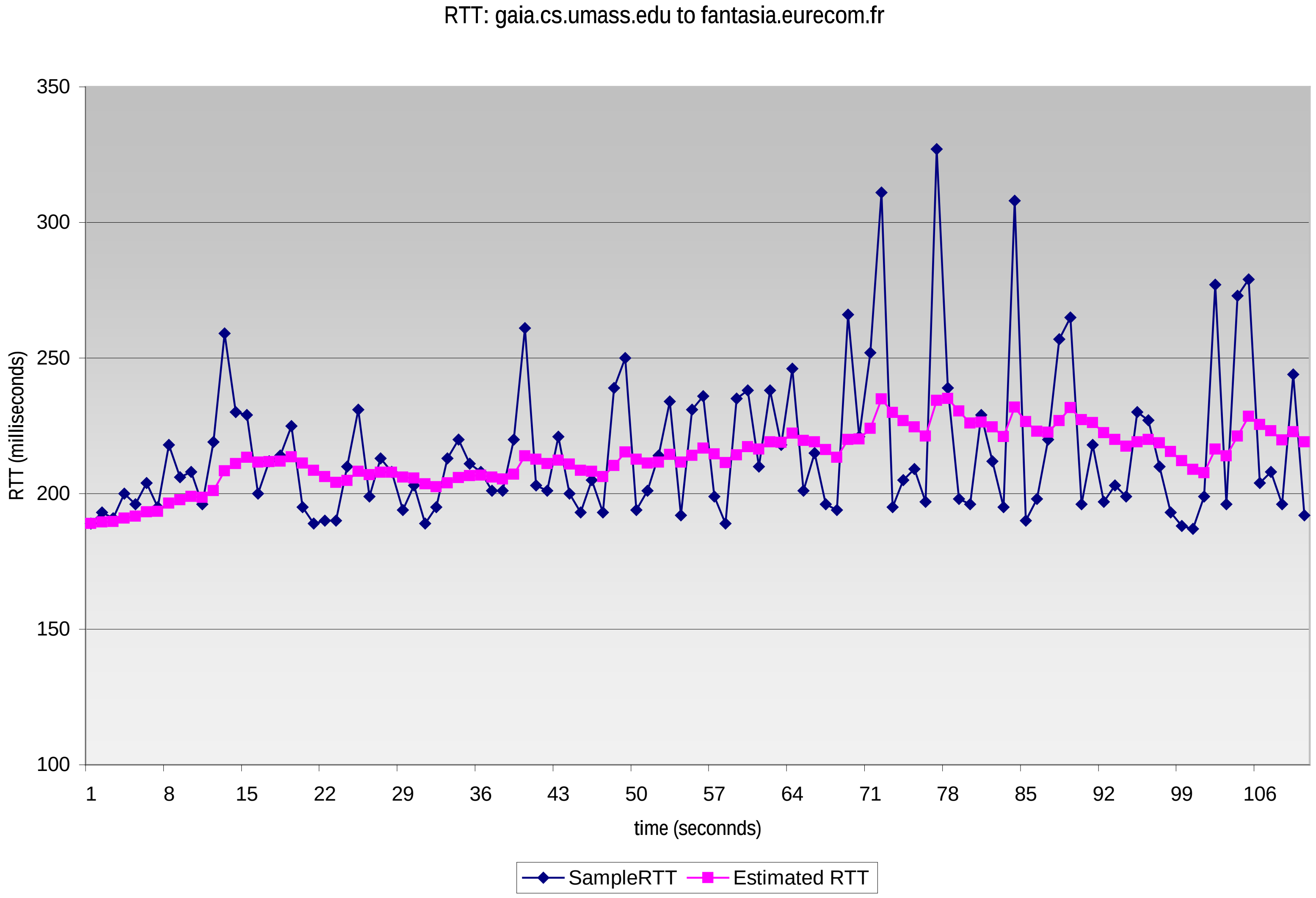
$$ERTT_{n+1} = \alpha \cdot RTT_n + (1 - \alpha) \cdot ERTT_n$$

RTT Estimate

$$\texttt{EstimatedRTT}_{n+1} = \alpha * \texttt{SampleRTT}_n + (1-\alpha) * \texttt{EstimatedRTT}_n$$

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

Example RTT estimation:



Setting the Timeout

Algoritmo di Karn-Partridge

- Re-transmitted segments are not considered in the RTT estimate
- The timeout value is set as

```
TimeoutInterval = 2*EstimatedRTT
```

Setting the Timeout

Algoritmo di Van Jacobson - Karel

- **EstimtedRTT** plus "safety margin"
  - large variation in **EstimatedRTT ->** larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, β = 0.25)
```

Then set timeout interval:

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

TCP reliable data transfer

- Window-based ARQ scheme (pipeline)
- cumulative ACKs
- TCP uses single retransmission timer
- retransmissions are triggered by:
  - timeout events
  - duplicate ACKs

- 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 acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are outstanding segments

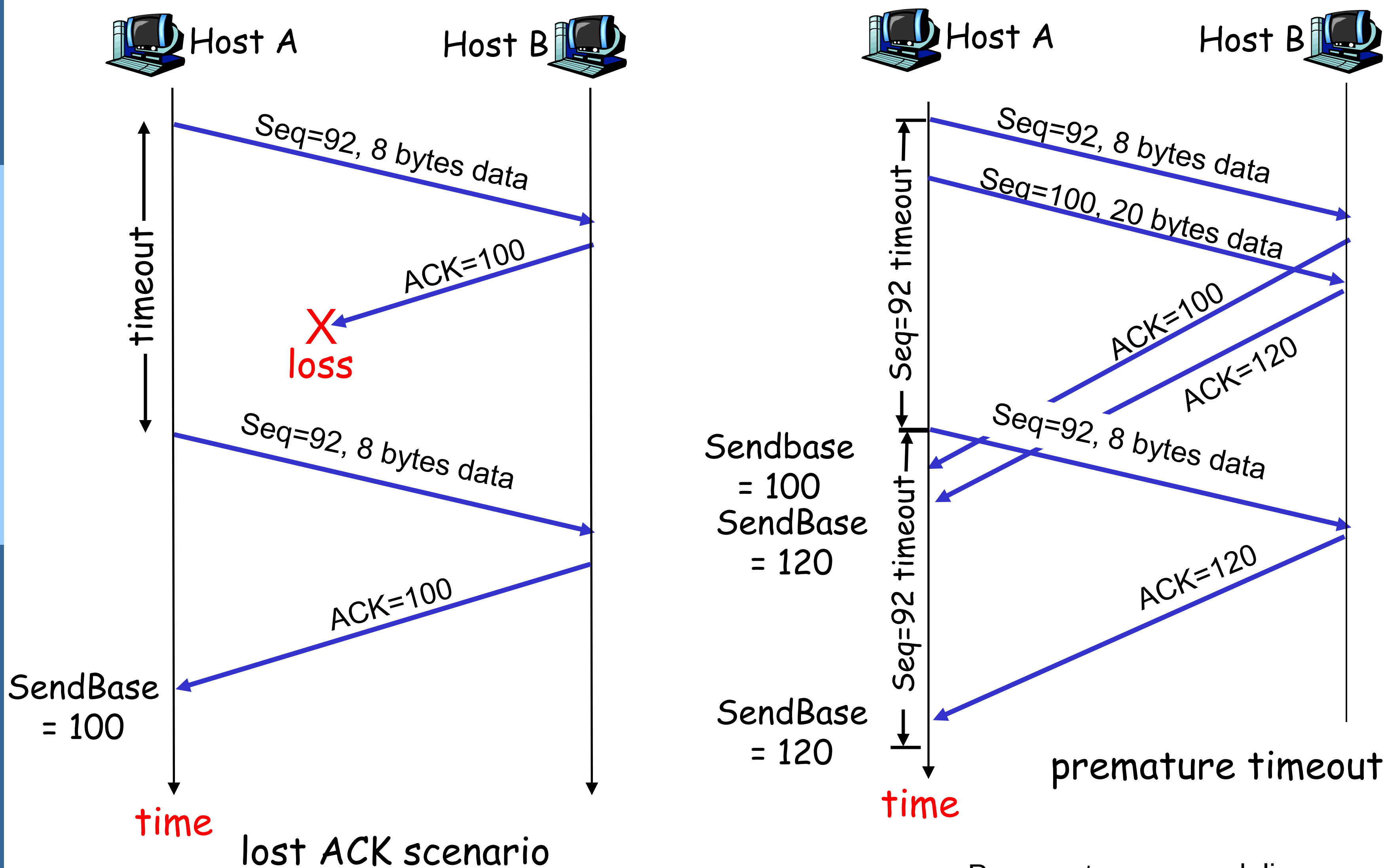
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
               start timer
          }

 }  /* end of loop forever */
```

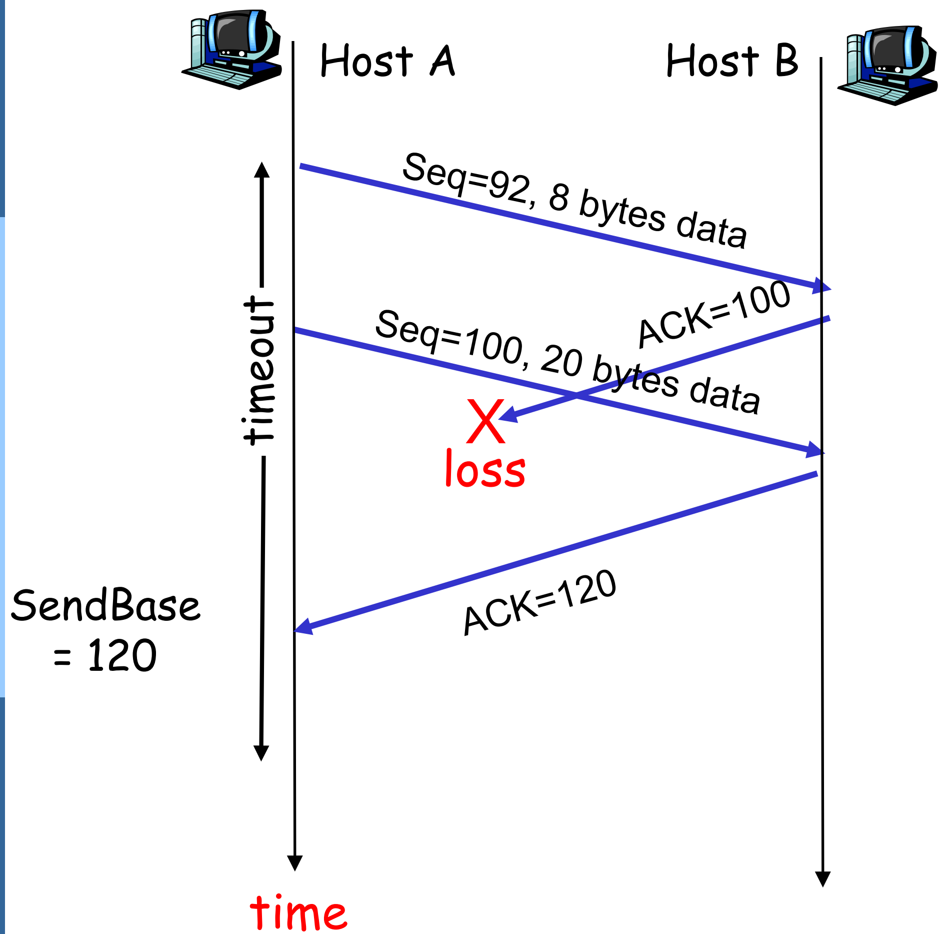
Comment:
- SendBase-1: last cumulatively ACKed byte

Example:
- SendBase=72 → SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is ACKed

TCP: retransmission scenarios



TCP retransmission scenarios (more)



Cumulative ACK scenario

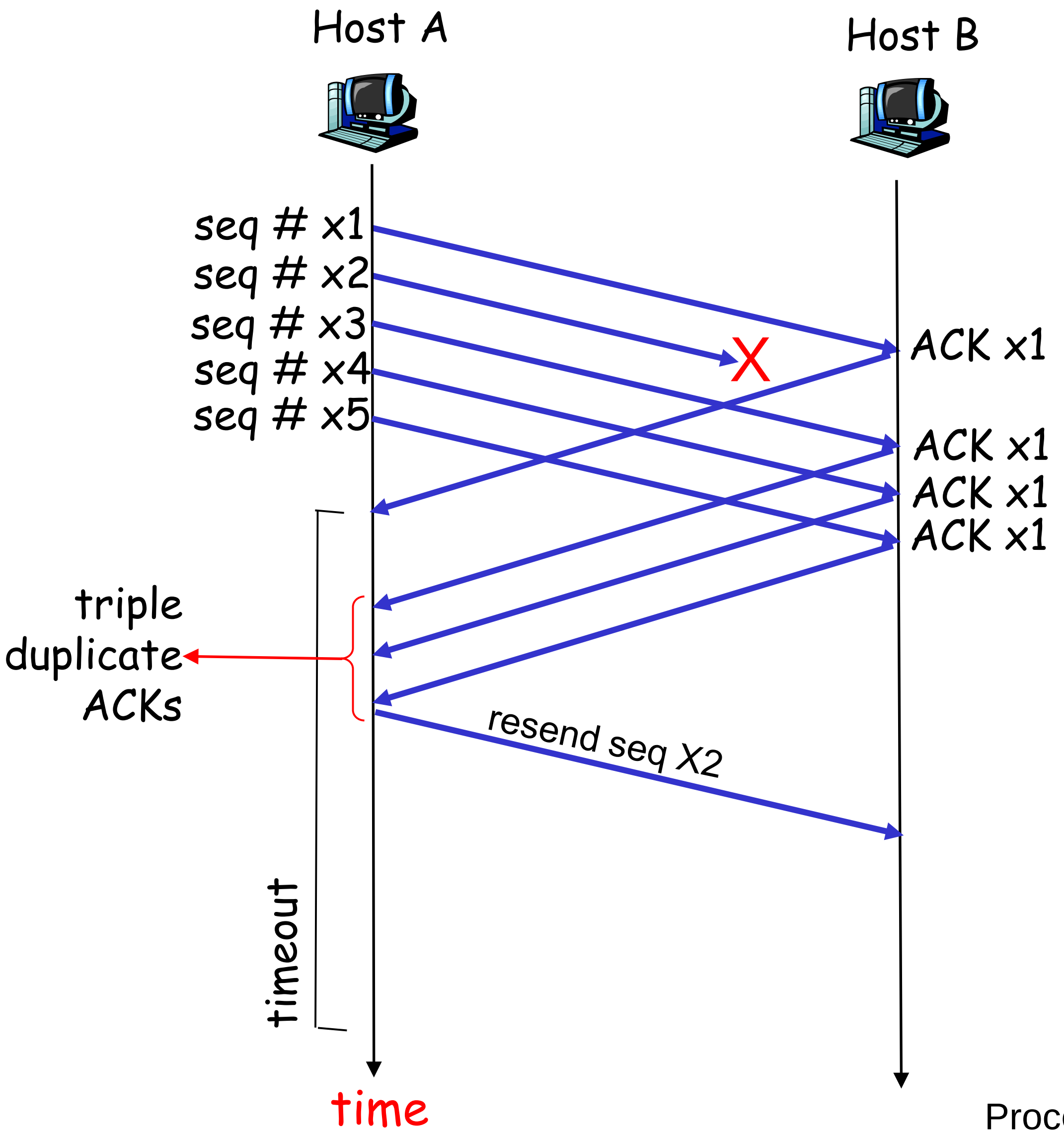
Doubling the Timeout Interval

- After each retransmissions the Timeout Interval is doubled
  - Exponential increase
- Simple form of congestion control
  - Similar to the backoff algorithm used in random-access MAC protocols (e.g. CSMA/CD, CSMA/CA, …)

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 for that segment
- If sender receives 3 duplicate ACKs (4 ACKS for the same data), it assumes that segment after ACKed data was lost.
- fast retransmit: resend segment before timer expires

Fast Retransmit



Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
            }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
               }
```

a duplicate ACK for already ACKed segment

fast retransmit

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

Is TCP a GBN or SR protocol?

- Cumulative acks
  - No specific ack for individual segments
- The sender only maintains `SendBase` and `NexSeqNum`
- But, at most one packet is retransmitted
- Hybrid protocol

- Selective ACK has been proposed [RFC 2018]
  - Selective ack for out-of-order segments

Roadmap

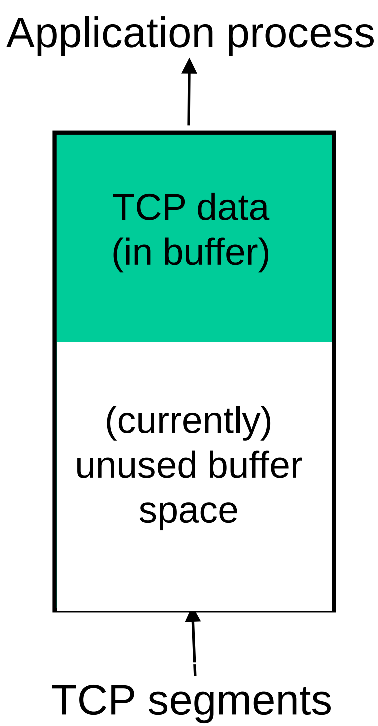
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
  - Segment structure
- Connection-oriented transport: TCP
  - Segment structure
  - connection management
  - reliable data transfer
  - flow control
  - Congestion control

TCP Flow Control

- receive side of TCP connection has a receive buffer.
  - app process may be slow at reading from buffer

flow control

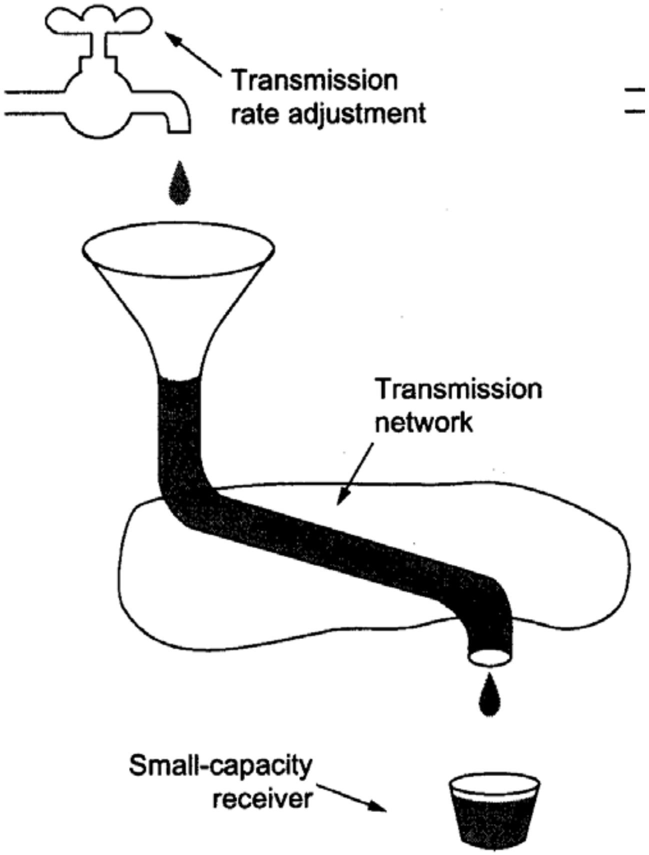
sender won't overflow receiver's buffer by transmitting too much, too fast



speed-matching service:

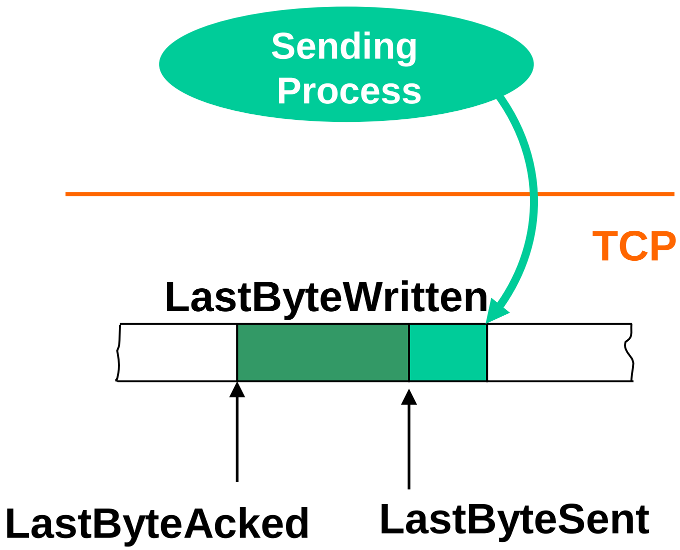
matching send rate to receiving application's drain rate

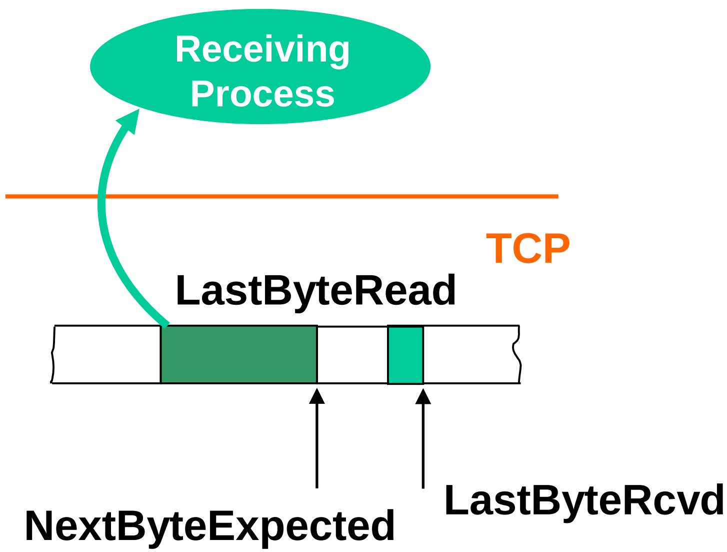
Flow Control



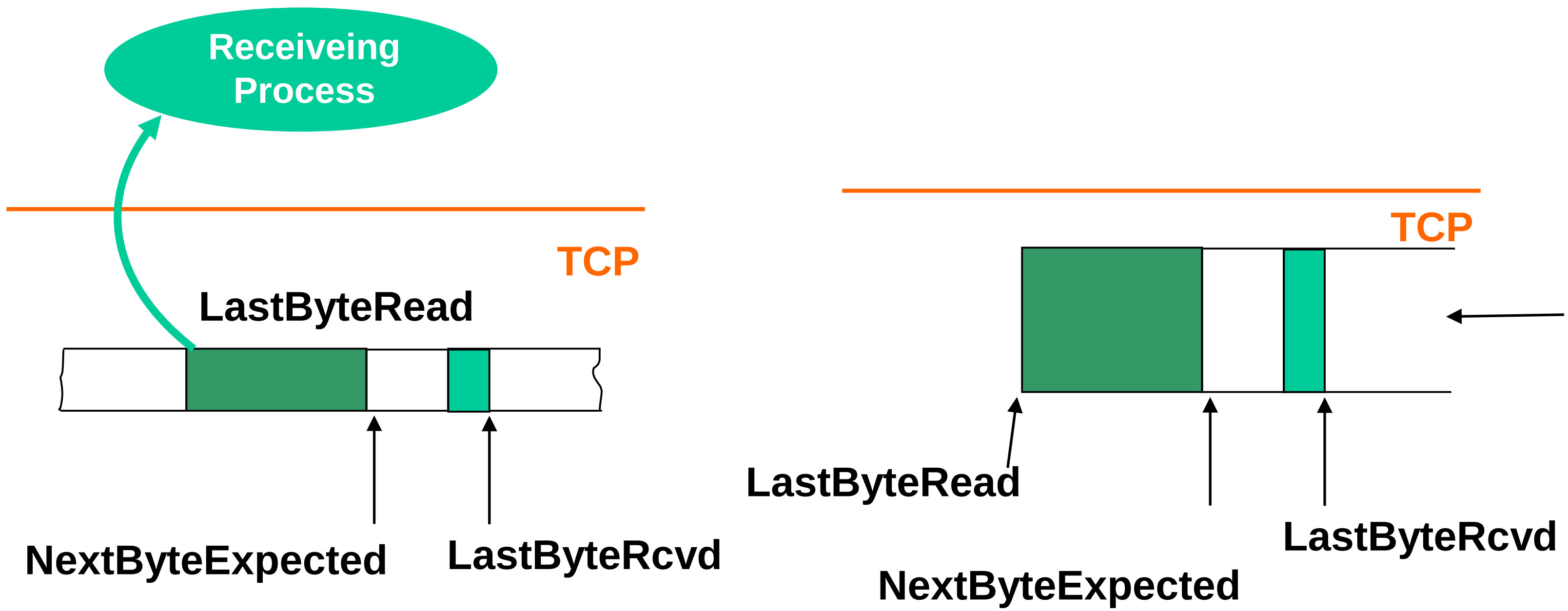
Receive/Transmit Buffers

- Transmit Buffer
  - Messages transmitted but not yet acked
  - Messages written by the application but not yet sent
- Receive Buffer
  - Out-of-order segments
  - In-order segments not yet read by the application





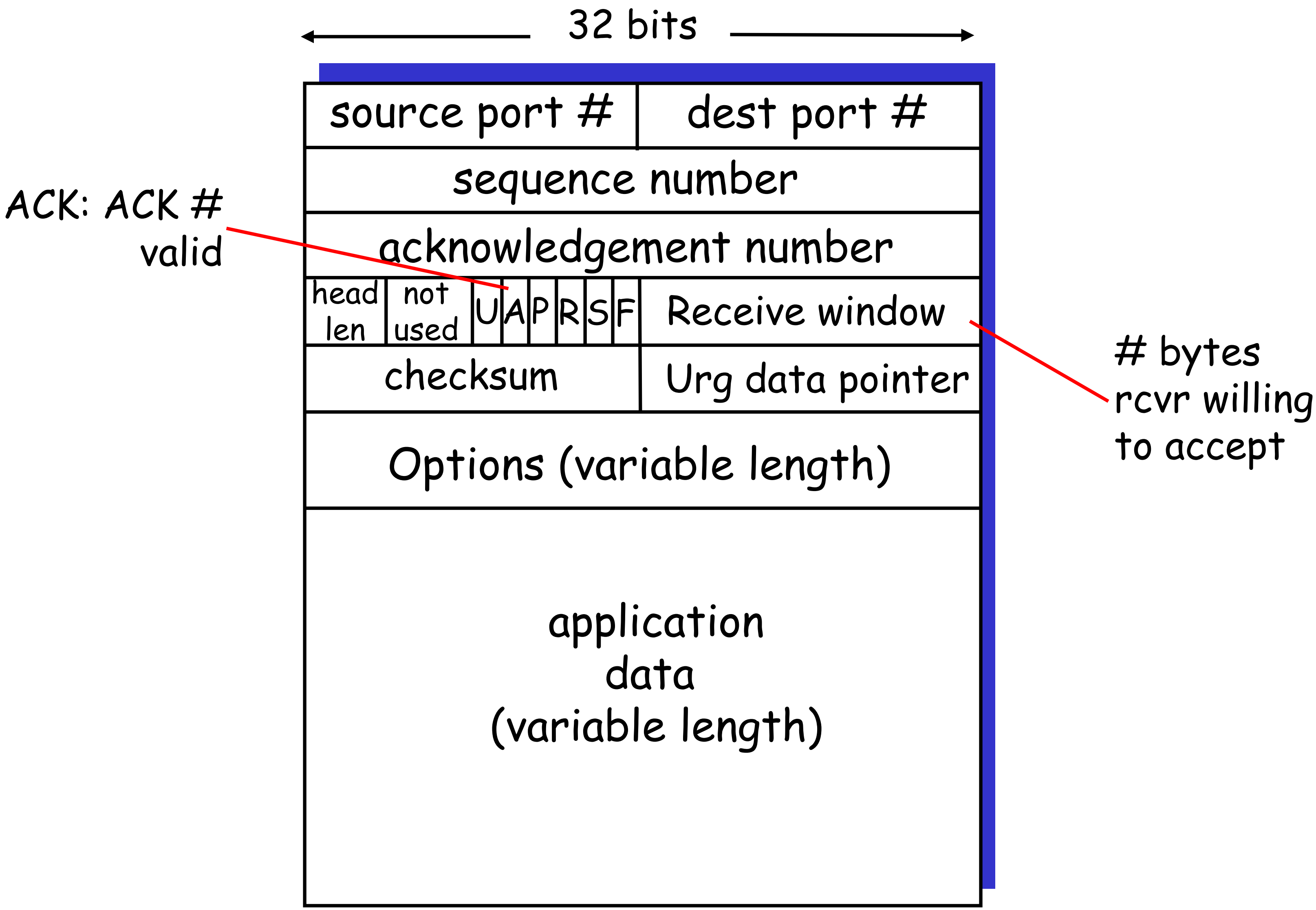
Receive Window size (receiver)



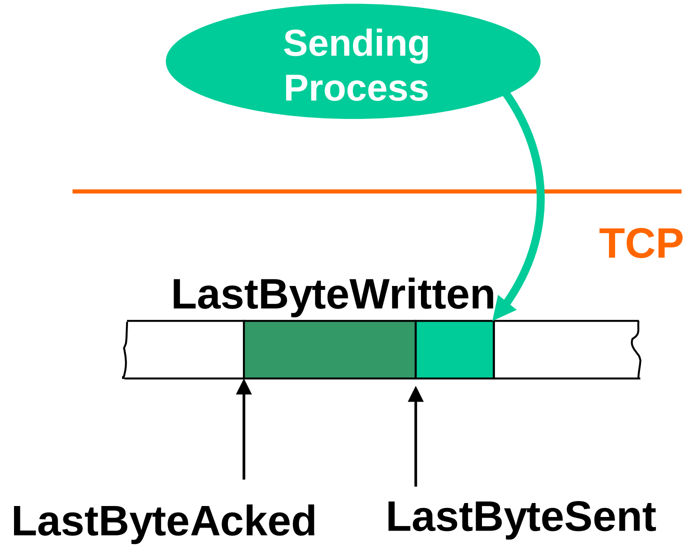
LastByteRcvd – LastByteRead ≤ RcvBuffer

AdvertisedRcvWindow = RcvBuffer – (LastByteRcvd – LastByteRead)

TCP segment structure



Receive Window size (sender)



LastByteSent – LastByteAcked $\leq$ AdvertisedWindow

RcvWindow = AdvertisedRcvWindow – (LastByteSent – LastByteAcked)

Question

- What happens if the available receive buffer reduces to 0?
  - Receiver: AdvertisedRcvWindow=0
  - Sender: RcvWindow=0 → the sender stops
  - The receiver cannot send acks → block
- TCP sender periodically sends a 1-byte segment to stimulate a reaction

Summary

- principles behind transport delivery services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control

- instantiation and implementation in the Internet
  - UDP
  - TCP