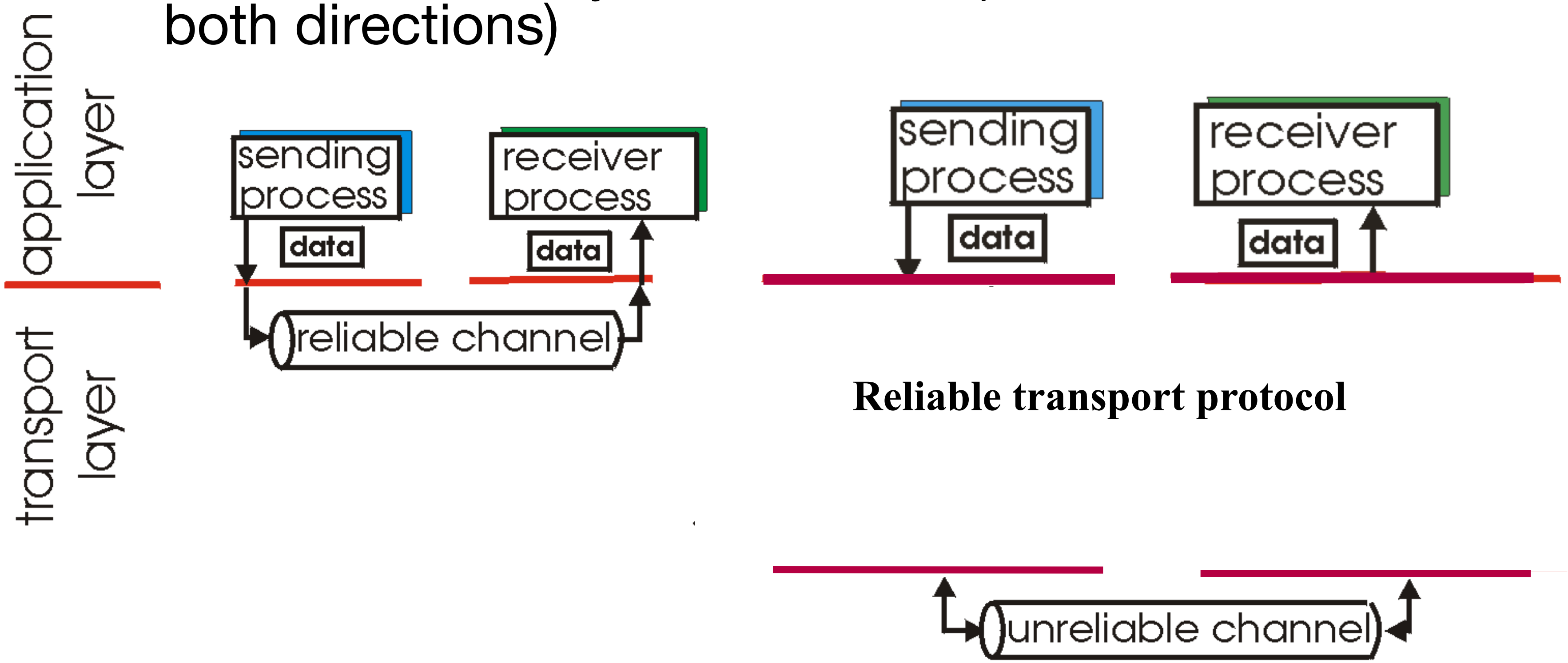
Principles of Reliable Data Transfer

- characteristics of unreliable channel determines complexity of a reliable data transfer protocol (rdt)
- incrementally develop sender, receiver sides of rdt
  - consider one-way data transfer (control info will flow in both directions)



(a) provided service
(b) service implementation

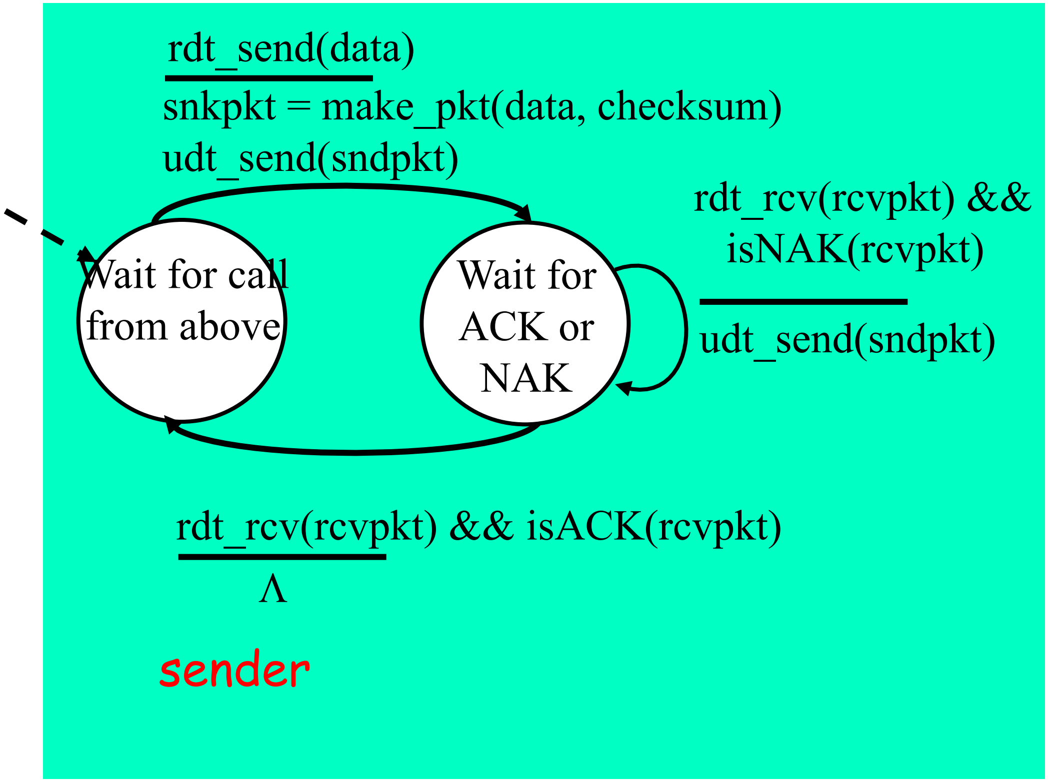
Three basic components in reliable data delivery by retransmission

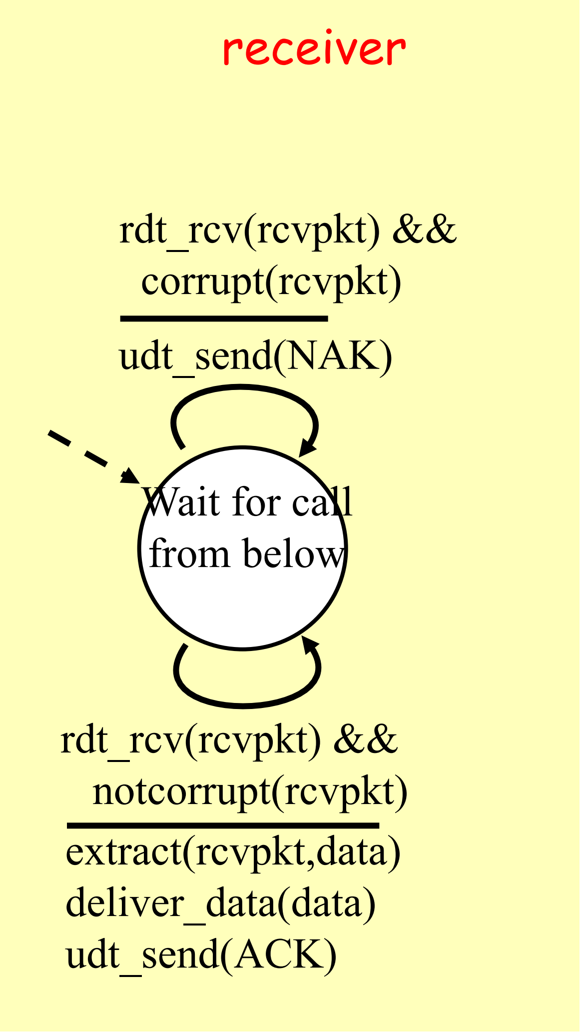
- sequence #: used to uniquely identify individual piece of data
- Acknowledgment (ACK): reception report sent by receiver to the sender
- Retransmission timer set by the sender for the already sent, but has not been acknowledged packet
  - Retransmit the packet when timer expires

Rdt2.0: channel with bit errors

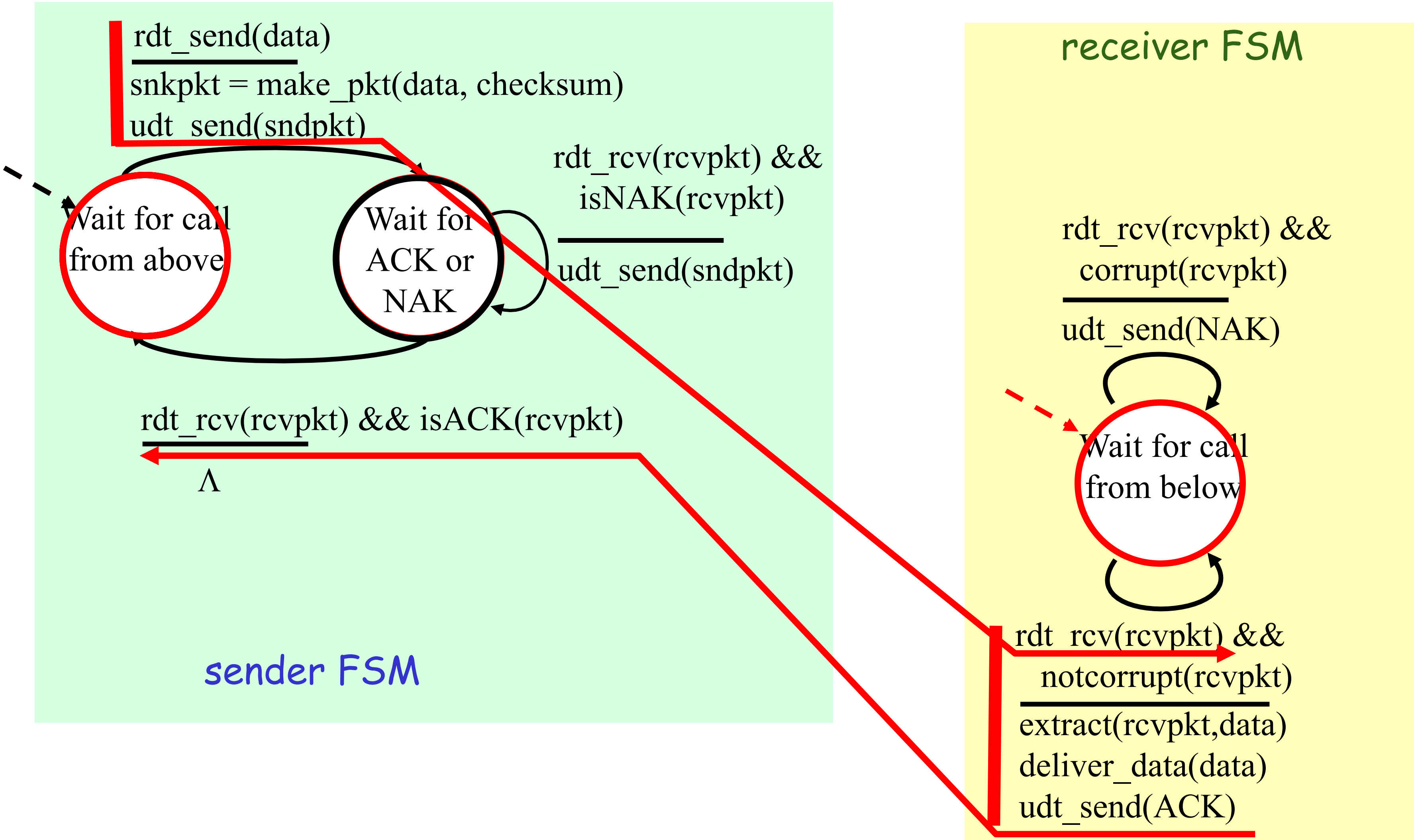
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr → sender

rdt2.0: FSM specification

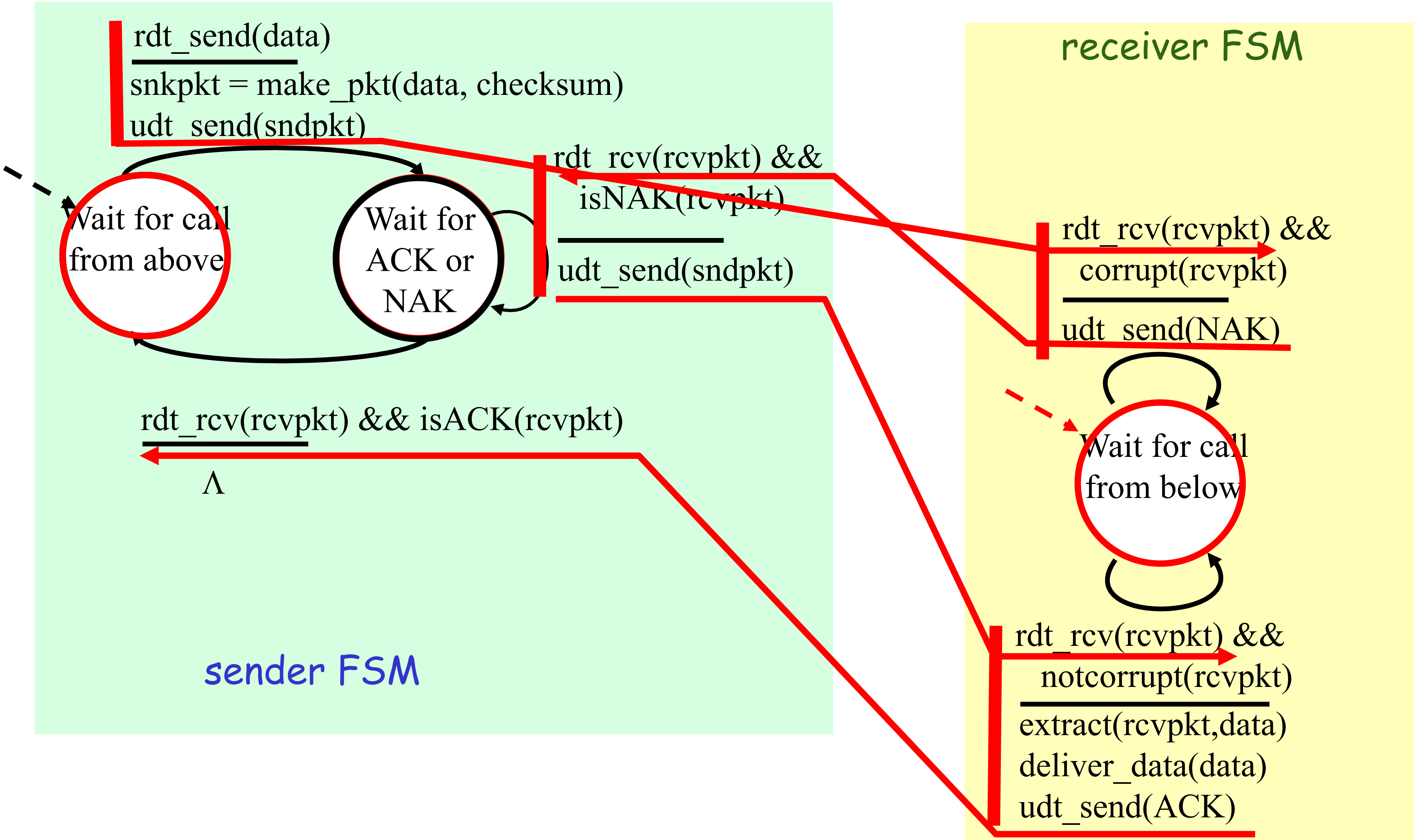




rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

→Need a way to detect duplicate

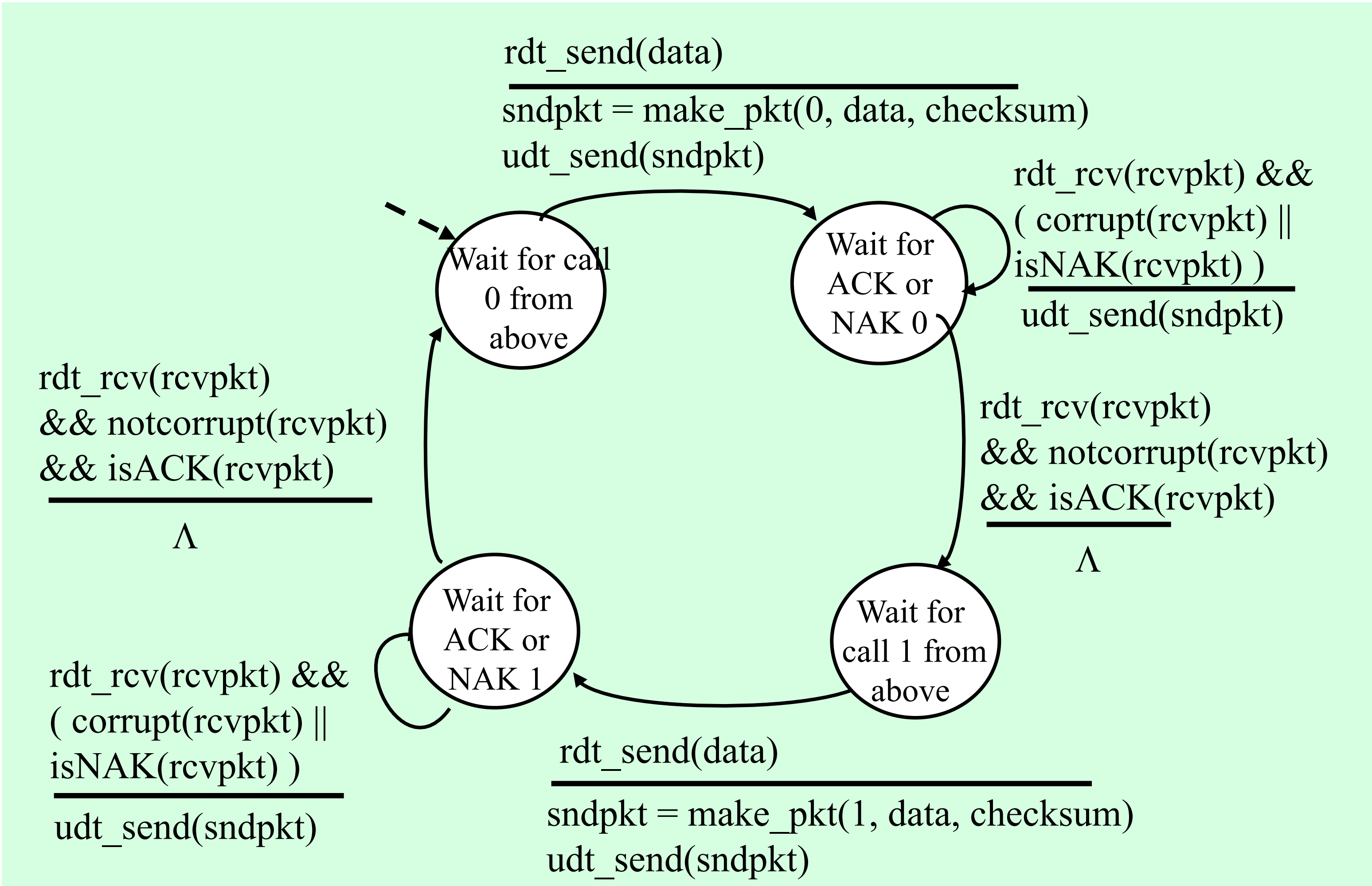
Handling duplicates:

- sender retransmits current pkt if ACK/NAK garbled
- sender adds *sequence number* to each pkt
- receiver discards duplicate pkt

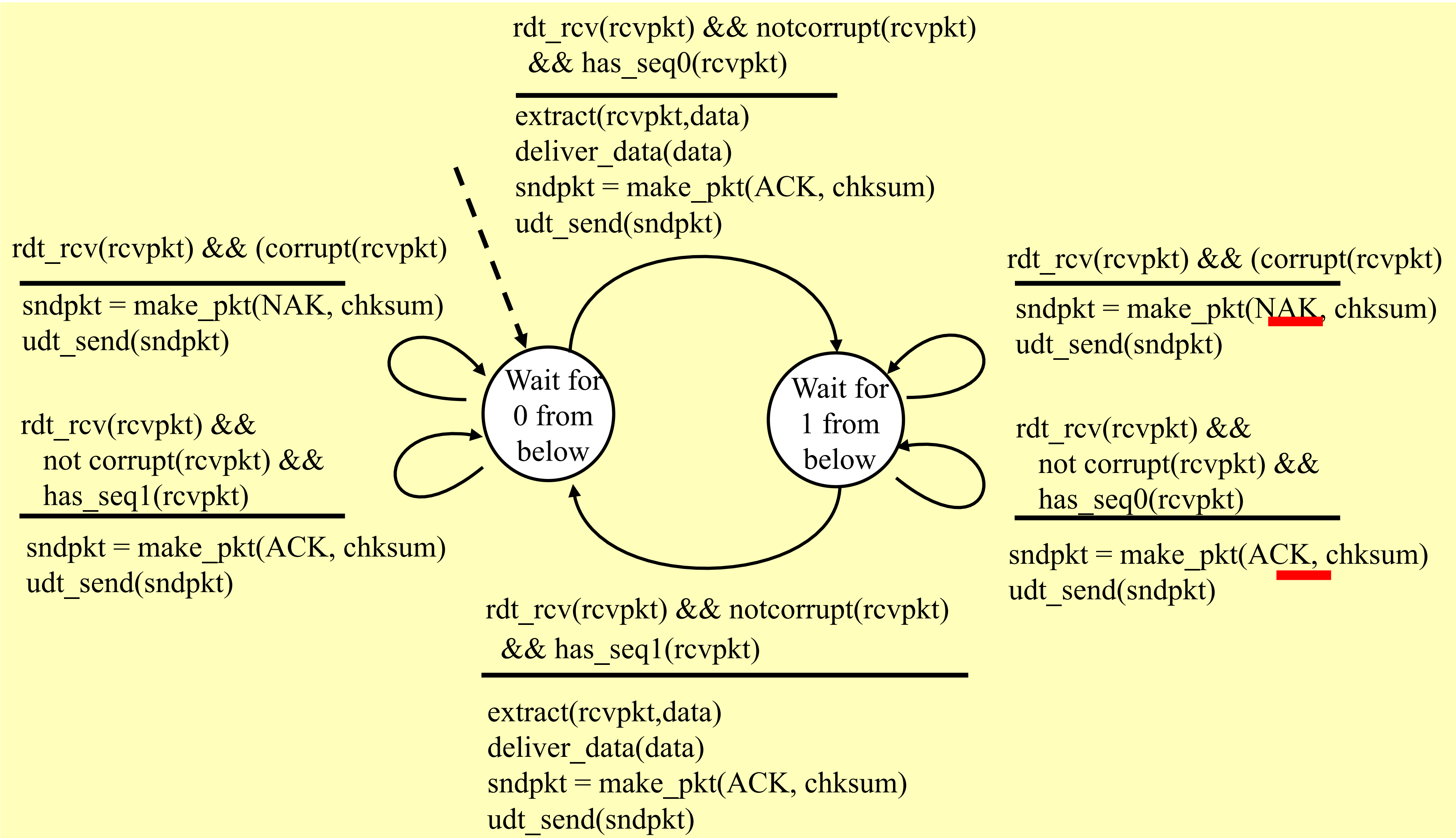
stop and wait

Sender sends one packet, then waits for receiver's response

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

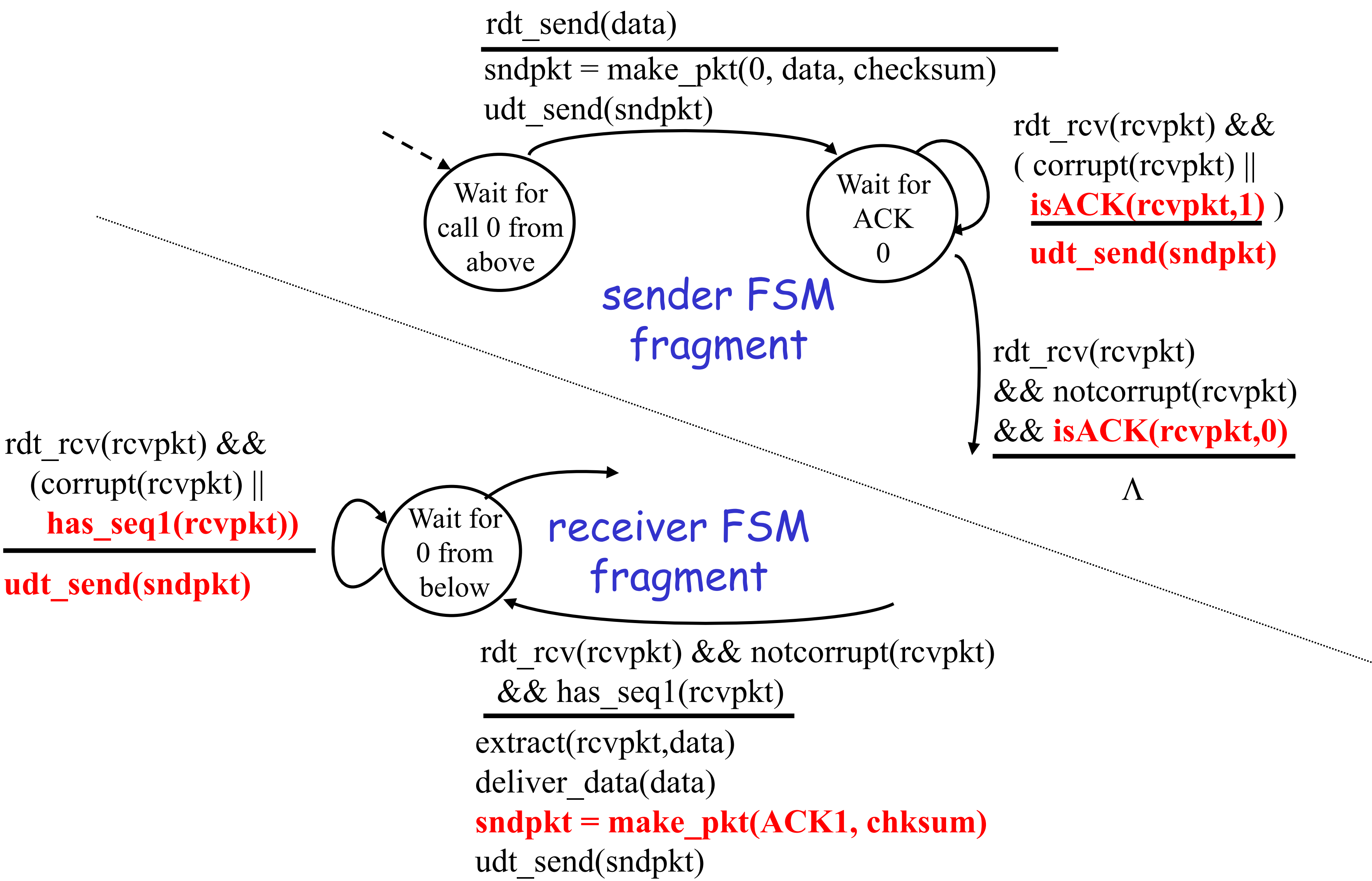
Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can*not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

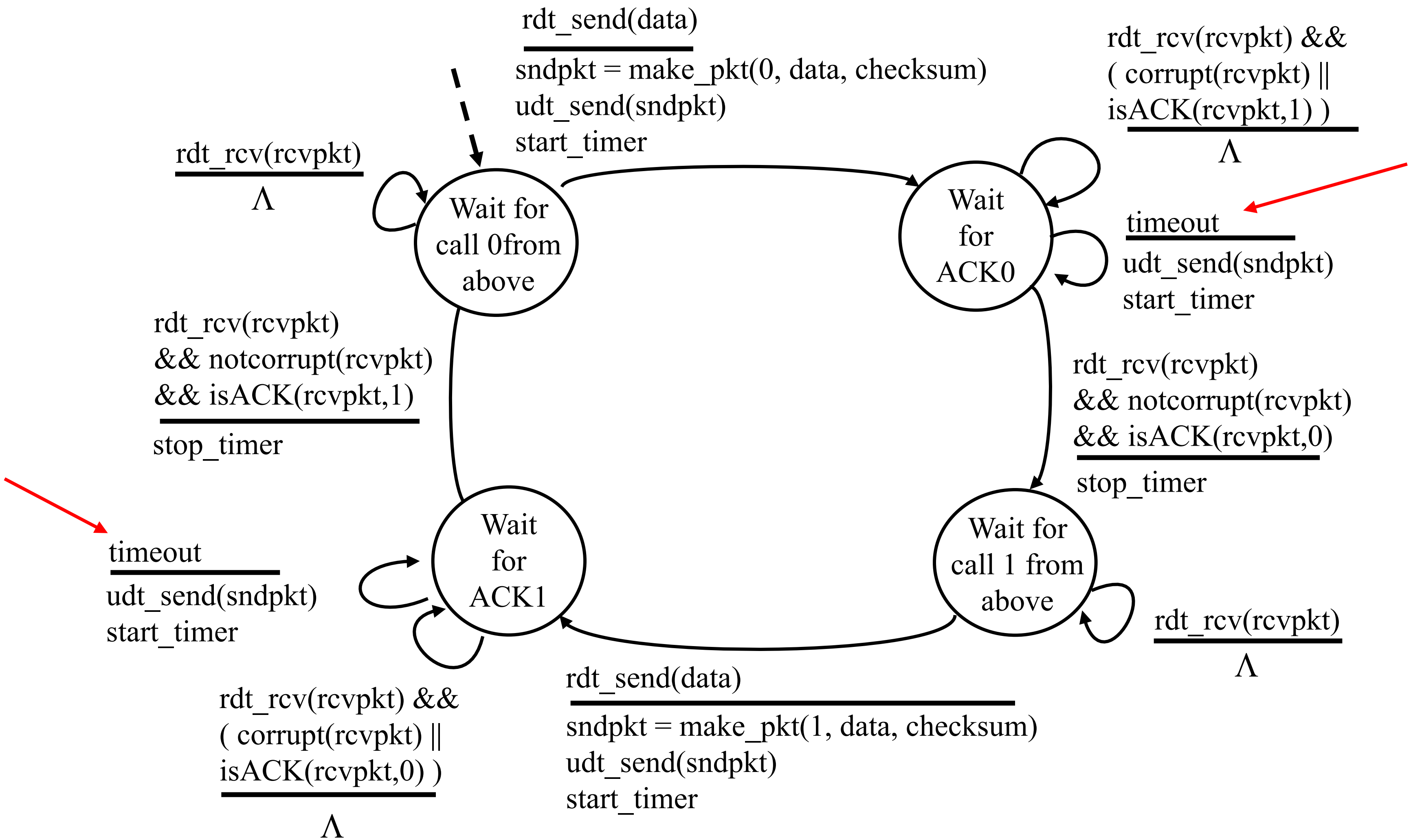
rdt2.2: sender, receiver fragments



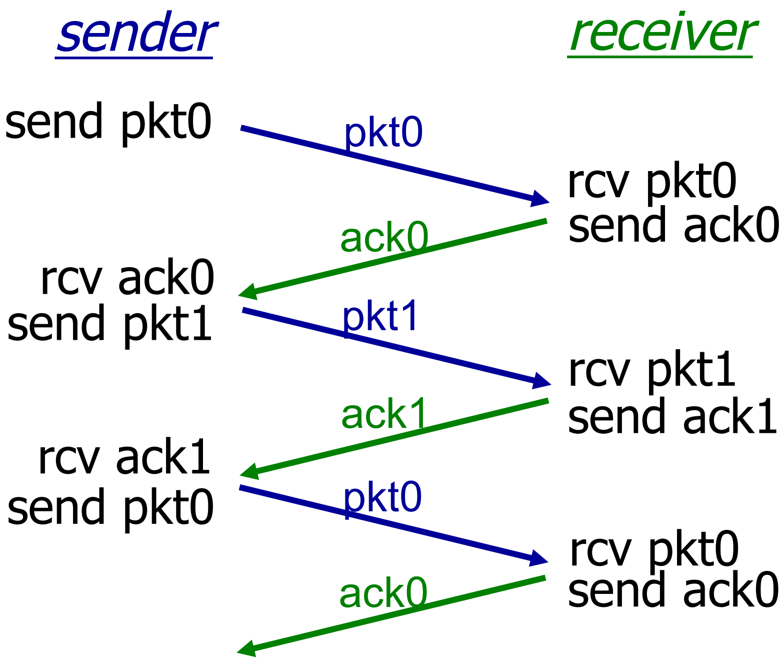
rdt3.0: channel with bit errors & packet loss

- After sending out a packet, the sender waits for ACK from receiver
  - Set up a retransmission timer
- When the timer expires: retransmits the packet
- In case the packet (or ACK) just delayed but not lost
  - Retransmitted packet will be a duplicate
  - Sequence number can detect this

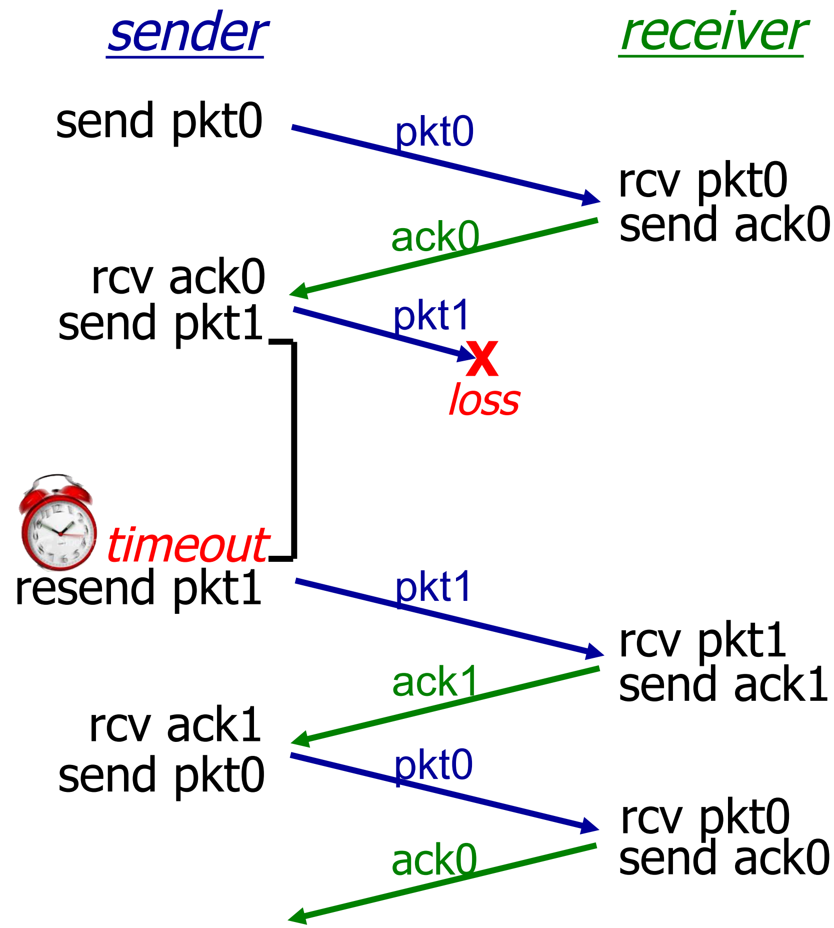
rdt3.0 sender



rdt3.0 in action

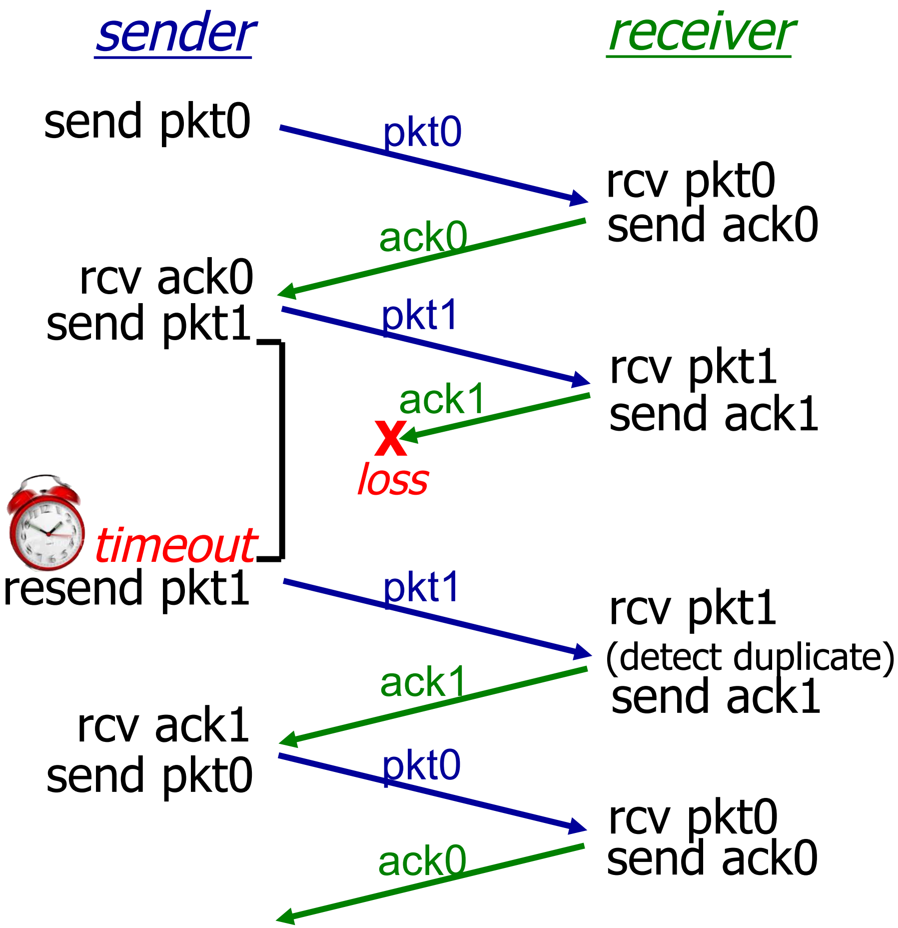


(a) no loss

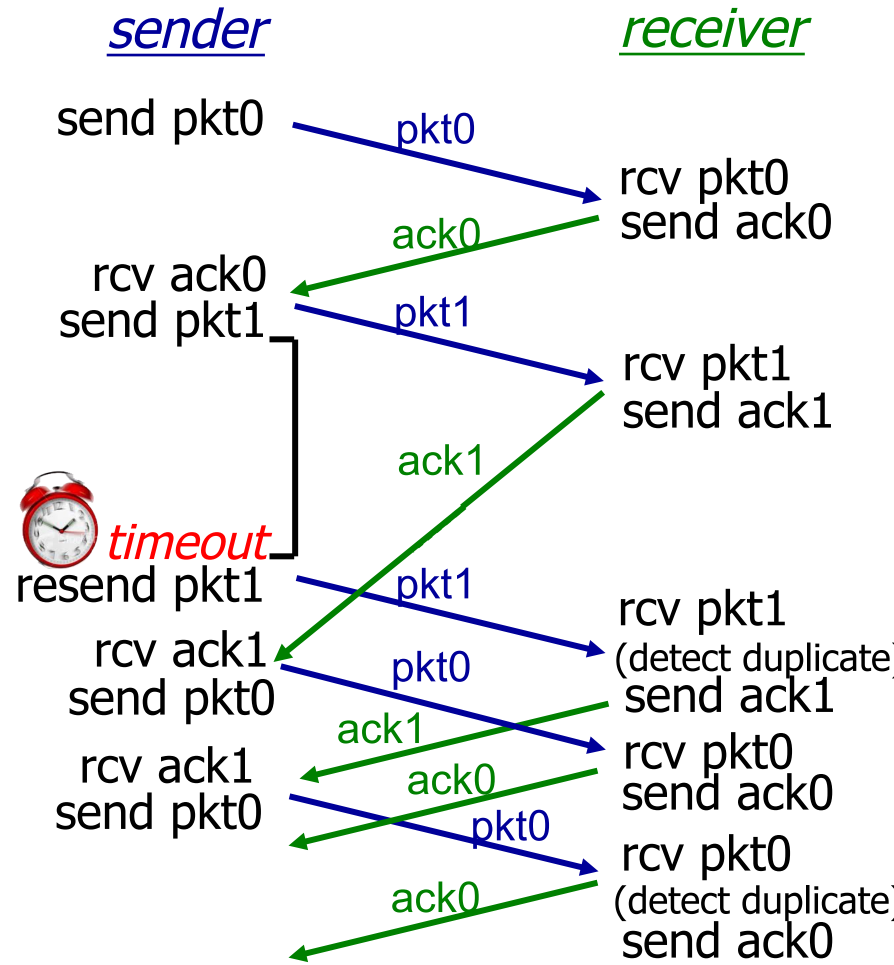


(b) packet loss

rdt3.0 in action

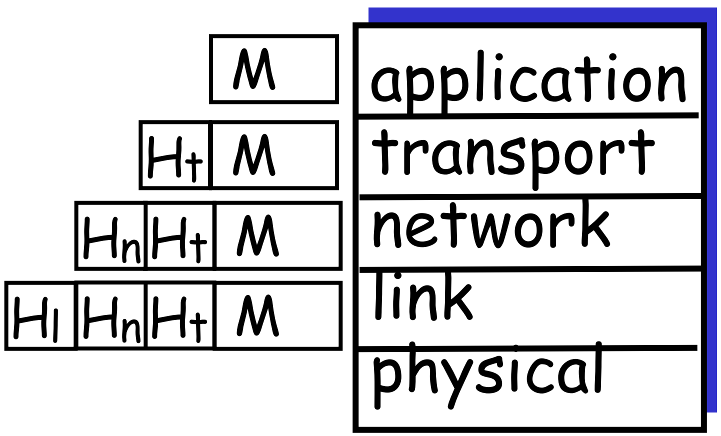


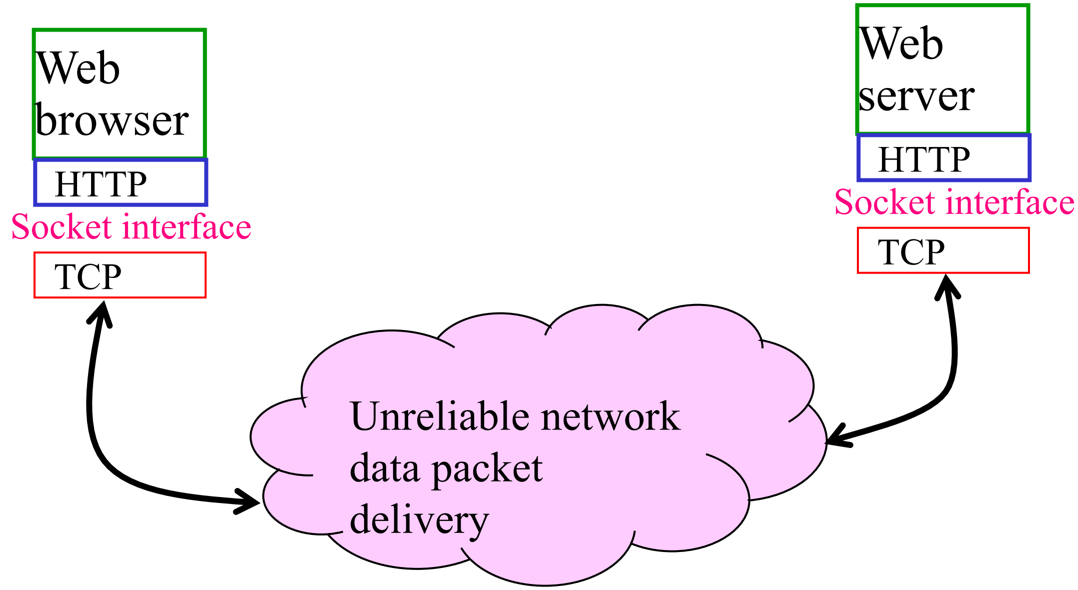
(c) ACK loss

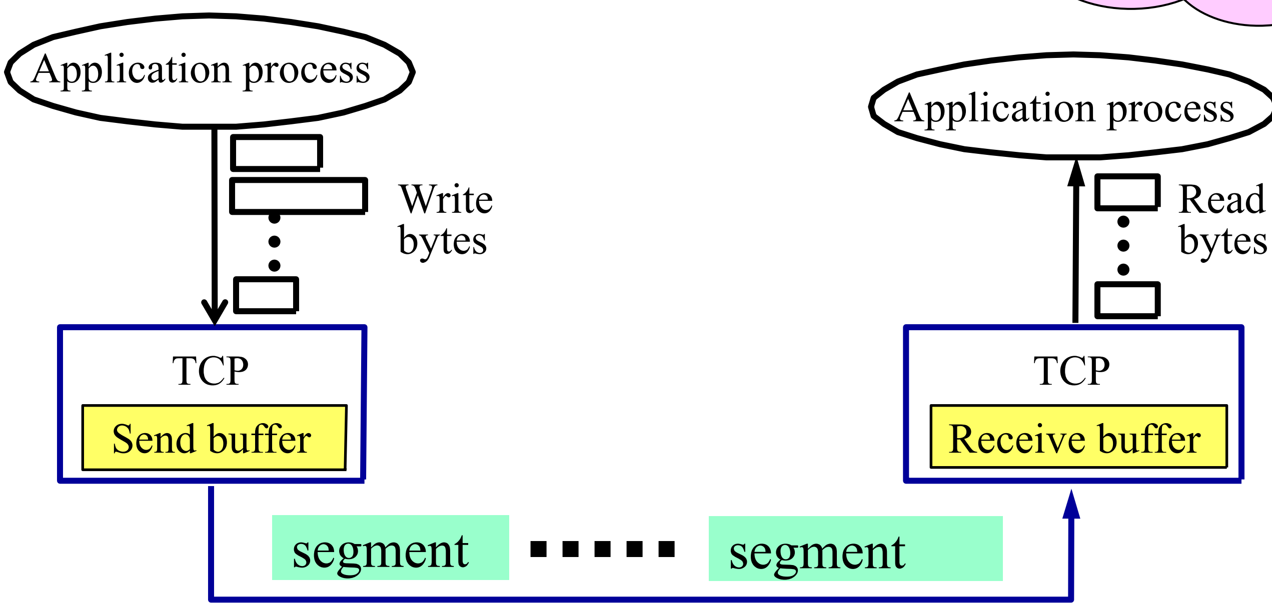


(d) premature timeout due to delayed ACK: duplicate transmissions

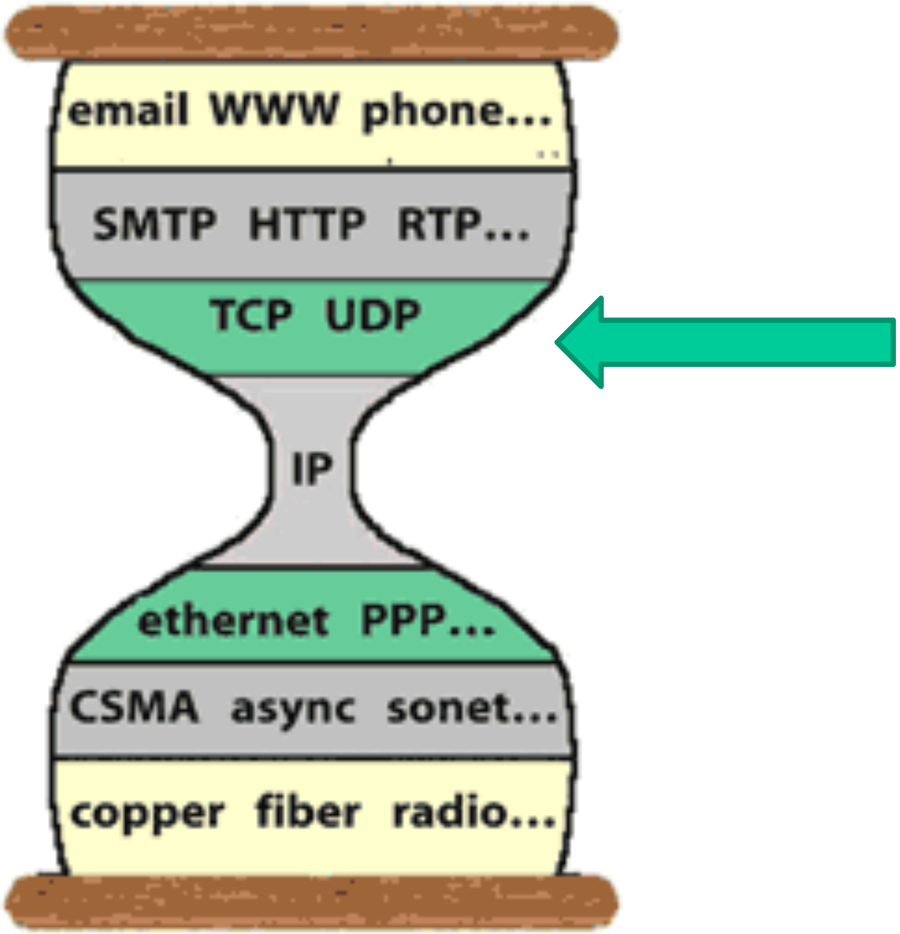
Keeps the Big Picture in Mind







Chapter 3 outline

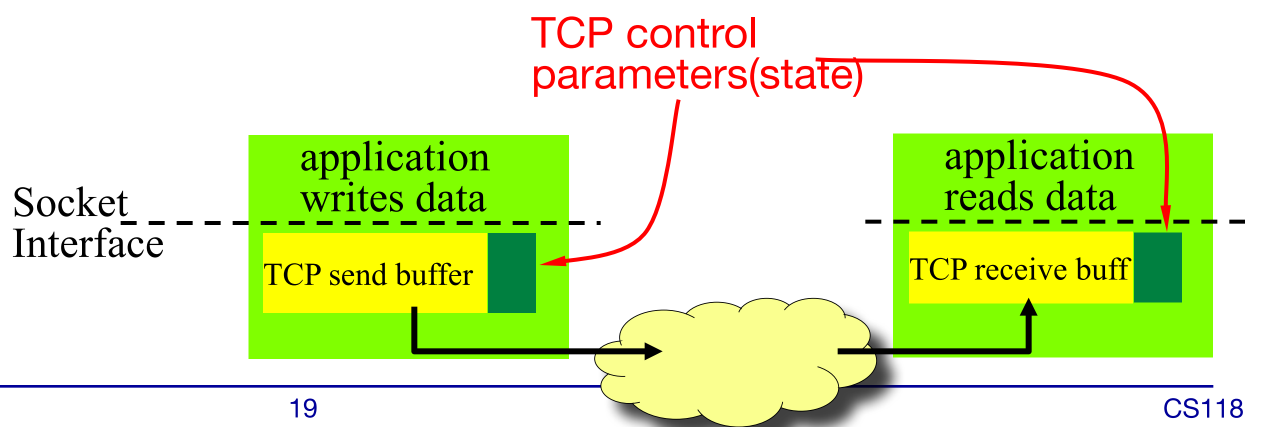


3.5 Connection-oriented transport: TCP

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

TCP: Overview

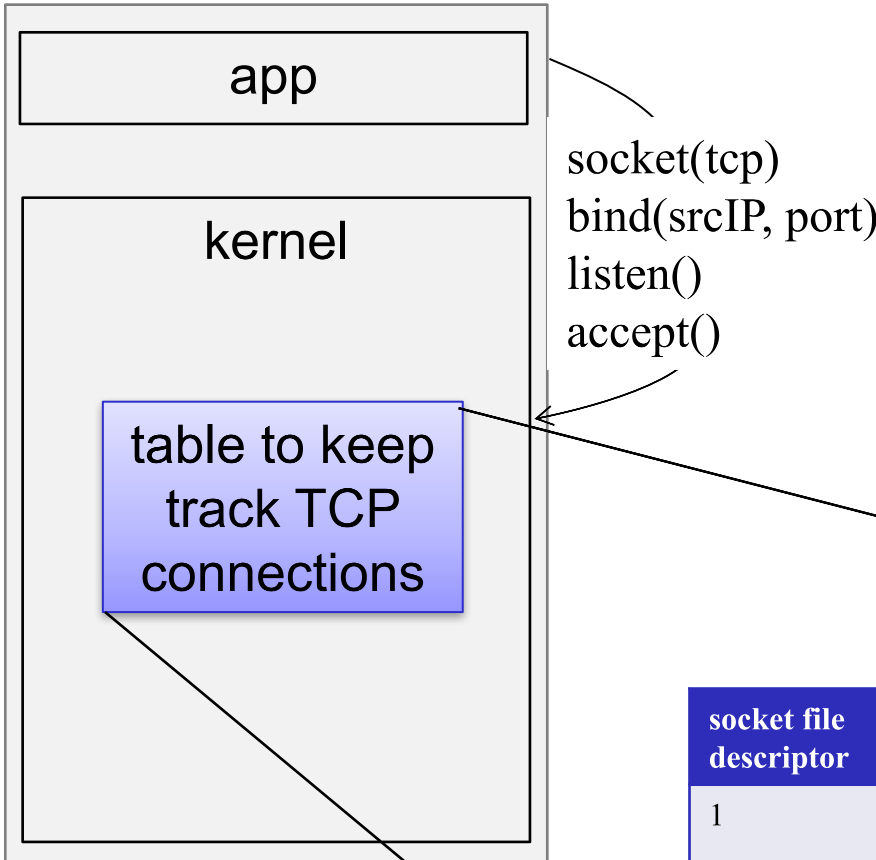
- point-to-point: creating a virtual connection between 2 processes
- connection-oriented:
  - exchange control msgs ***first*** to initialize connection state
- full duplex data delivery:
  - bi-directional data flow over the same connection
- reliable, in-order *byte steam* delivery
  - no “message boundaries”
- flow controlled
  - Receiver sets flow control window size to prevent sender from flooding
- congestion controlled
  - Reduce traffic overload in the network



Connection-oriented de-multiplexing

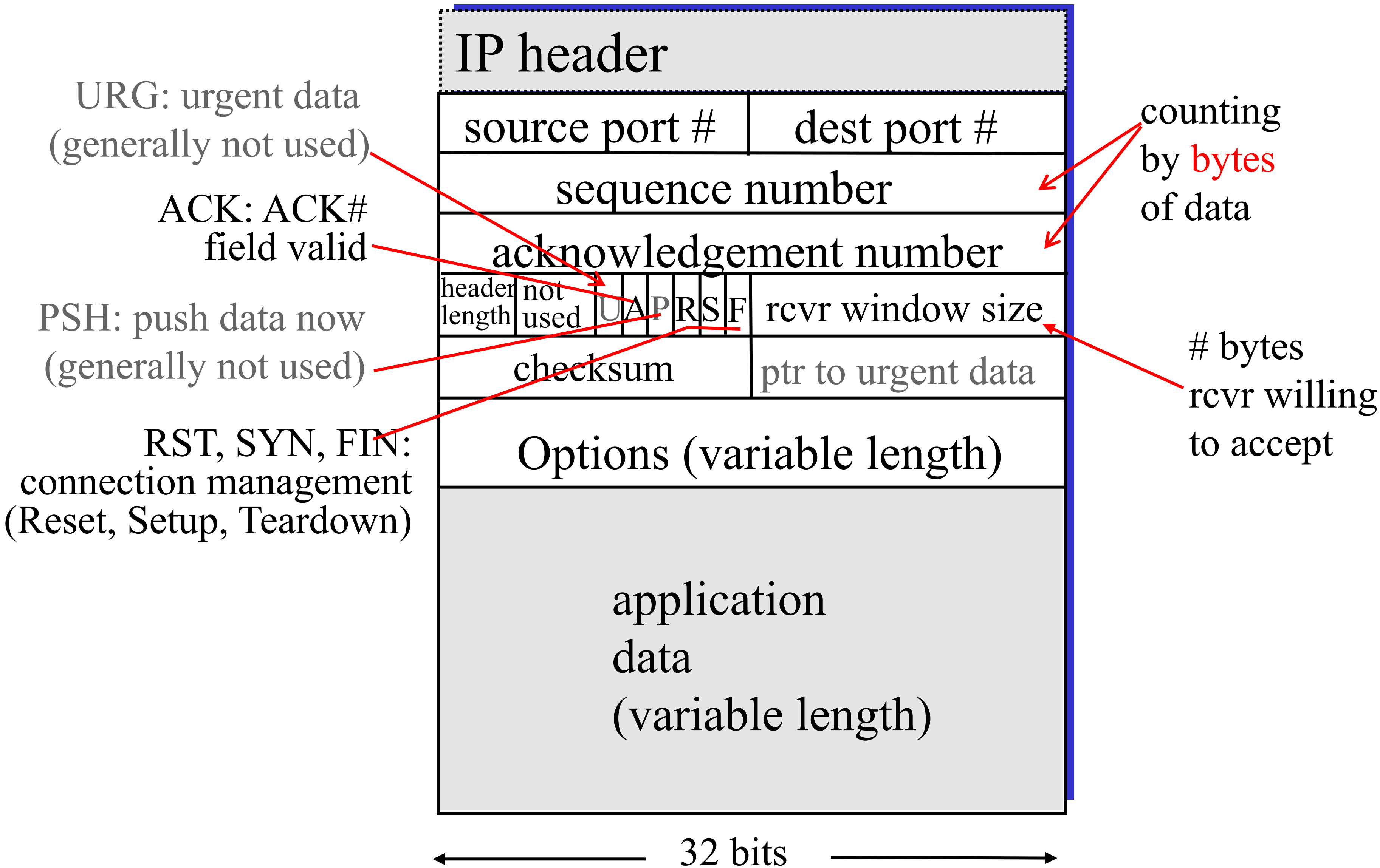
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - destination IP address
  - destination port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by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

Multiplexing/De-multiplexing



socket file descriptor	src IP, src port, dst IP, dst port
1	0.0.0.0, 4000, *, *
…	
1024	10.0.0.1, 4000, 131.179.196.45, 1054

TCP packet (segment) format



TCP's seq. #s and ACK #s

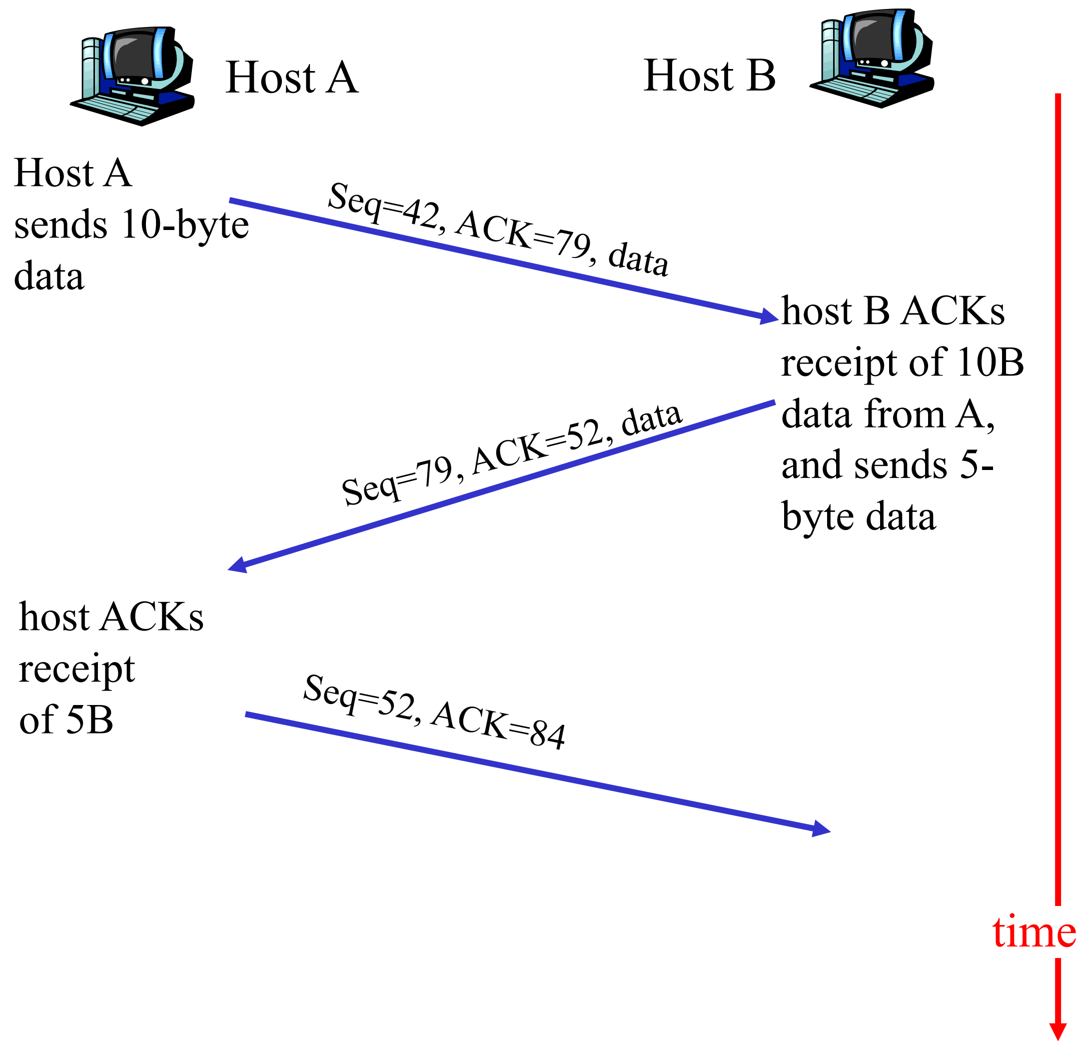
Seq. #: the seq# of the first byte in segment's data

ACK #: seq # of next byte expected from the other side

- cumulative ACK

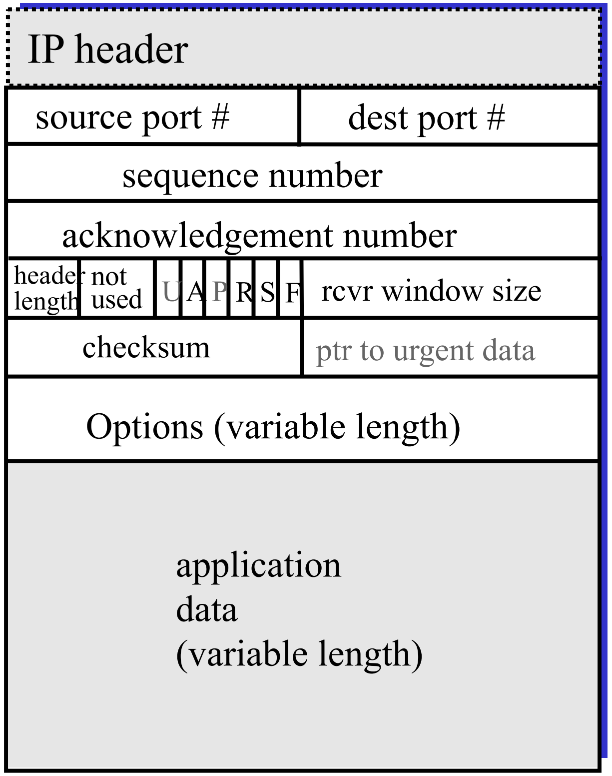
Q: how does receiver handle out-of-order segments?

A: TCP spec doesn't say, up to implementation



TCP Connection Management

- Connection setup
  - why?
- Connection teardown
  - why?



TCP Connection Setup

Must initialize TCP control variables before sending data

- Initial seq. # used in each direction
- Buffer size (rcvWindow)

Three way handshake

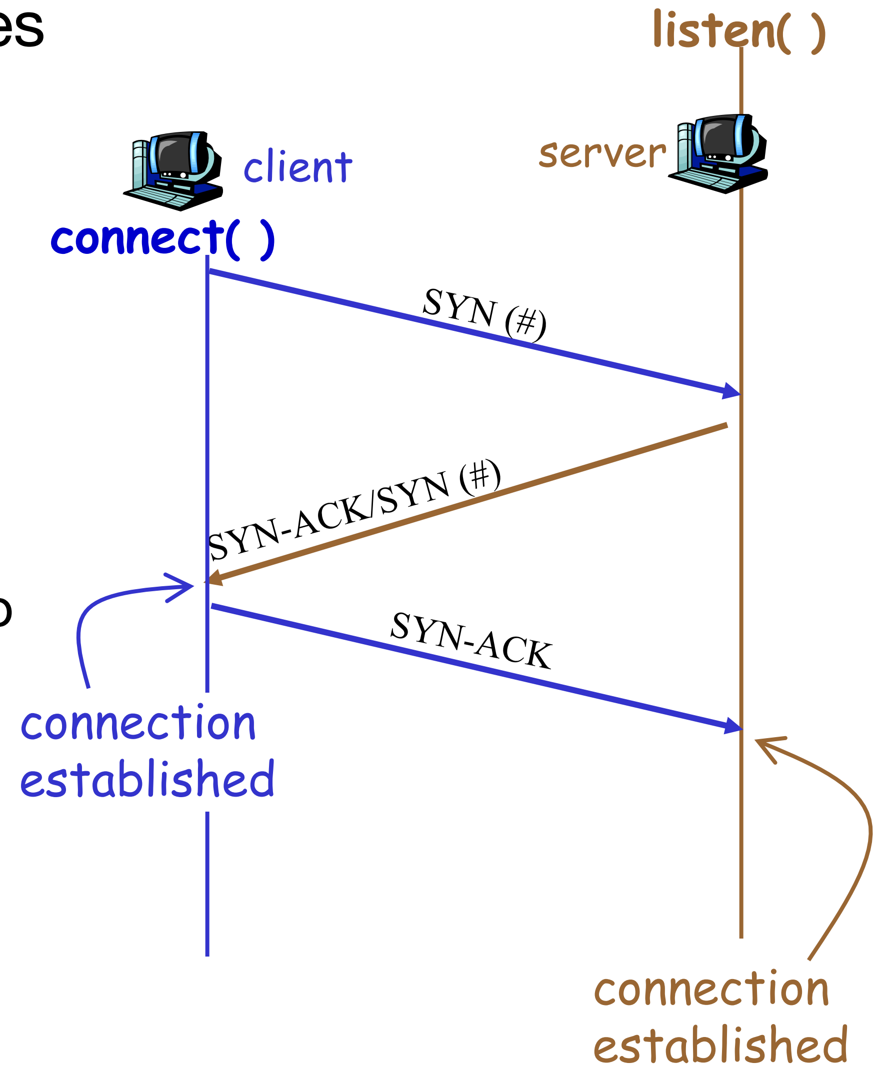
1: client host sends TCP SYN segment to server

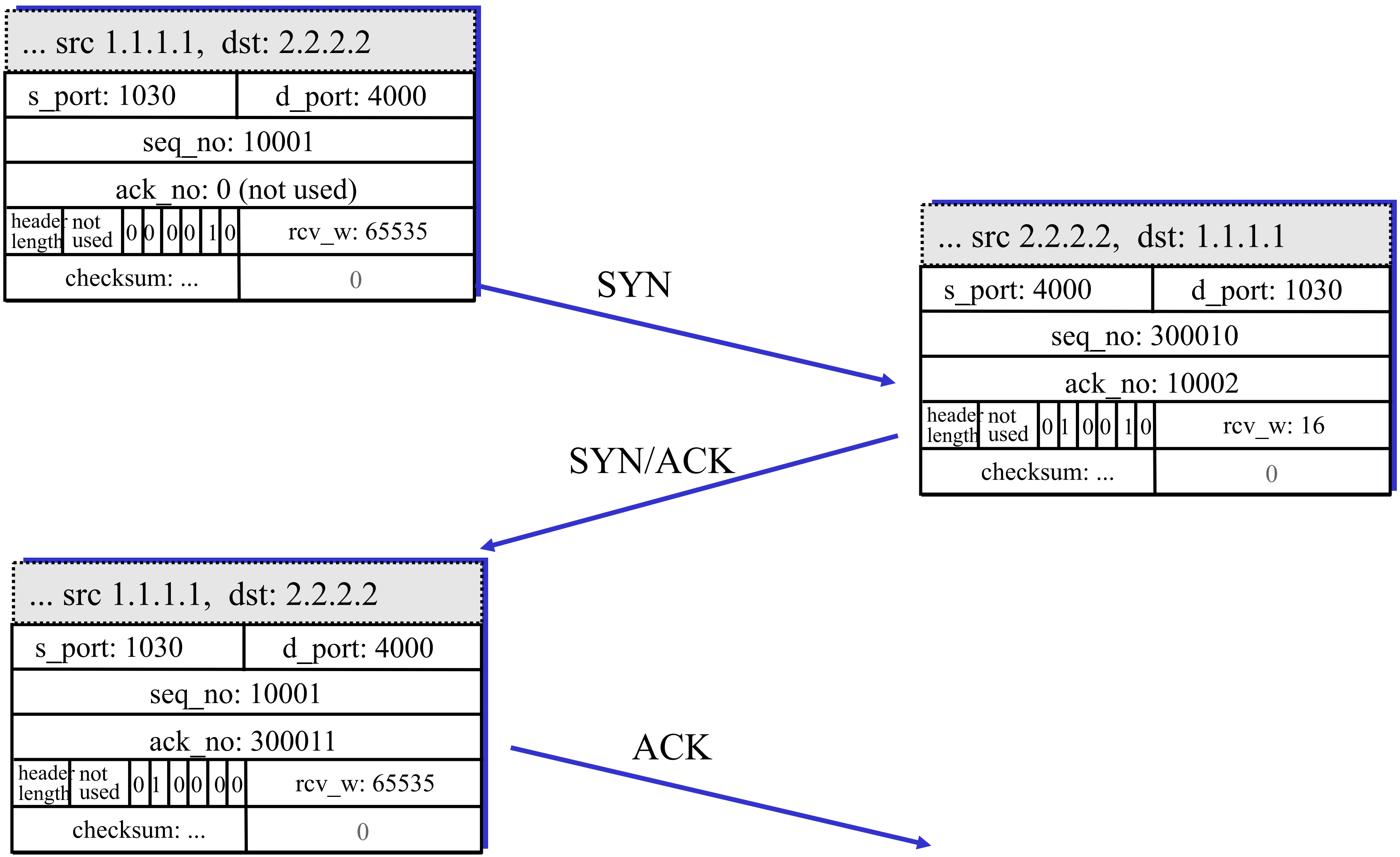
- specifies initial seq #
- Does *not* carry data

2: server receives SYN, replies with **SYN_ACK** and **SYN** control segment

3: client host sends SYN_ACK

- May carry data




... src 1.1.1.1, dst: 2.2.2.2
s_port: 1030
d_port: 4000
seq_no: 10001
ack_no: 0 (not used)
header length
not used
0 0 0 0 1 0
rcv_w: 65535
checksum: ...
0
SYN
... src 2.2.2.2, dst: 1.1.1.1
s_port: 4000
d_port: 1030
seq_no: 300010
ack_no: 10002
header length
not used
0 1 0 0 1 0
rcv_w: 16
checksum: ...
0
SYN/ACK
... src 1.1.1.1, dst: 2.2.2.2
s_port: 1030
d_port: 4000
seq_no: 10001
ack_no: 300011
header length
not used
0 1 0 0 0 0
rcv_w: 65535
checksum: ...
0
ACK

TCP Connection Close

- Either end can initiate the close of ***its end*** of the connection at any time

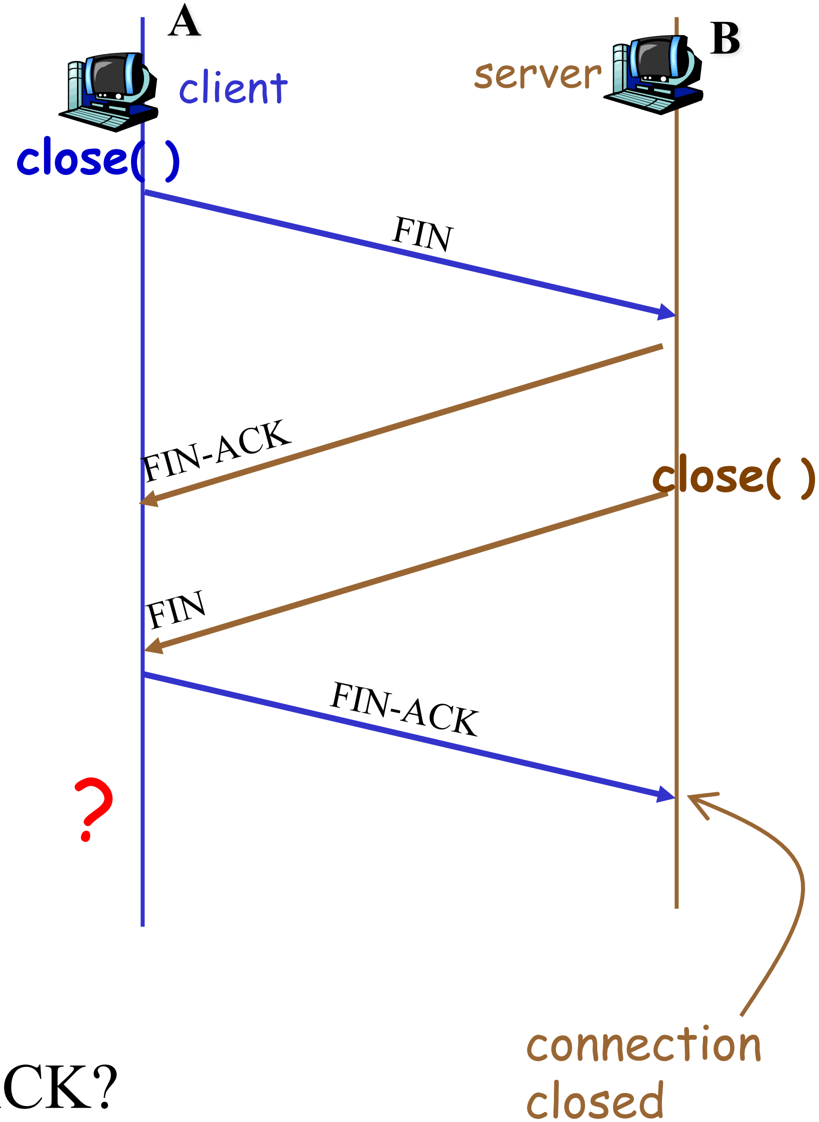
1: one end (A) sends TCP FIN control segment to the other

  - ***No data***

2: the other end (B) receives FIN, replies with FIN_ACK; when it's ready to close too, send FIN

3: A receives FIN, replies with **FIN-ACK**.

4: B receives FIN_ACK, close connection



what should A do after sending FIN_ACK?

the well-known “two-army problem”

Q: how can the 2 red armies agree on an attack time?

- Fact: the last one who send a message does not know whether the msg is delivered
- one cannot send an ACK to acknowledge an ACK

TCP Connection Close

1: A sends TCP FIN control segment to the other

2: B receives FIN, replies with FIN_ACK; when it's ready to close too, send FIN

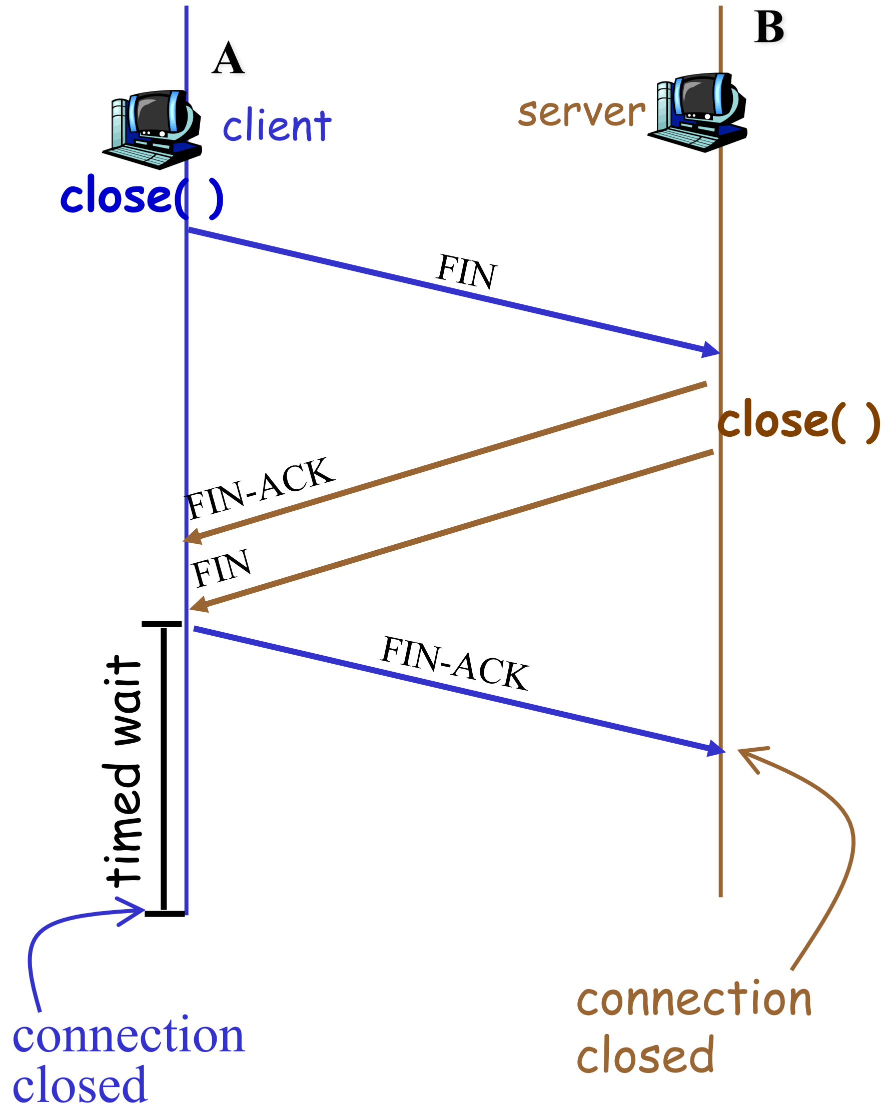
3: A receives FIN, sends FIN_ACK

- A Enters "timed wait", waits for 2 MSL before deleting the connection state

4: B receives FIN_ACK, close connection

5: A closes the connection after waiting for "long enough" time w/0 receiving retransmitted FIN

  - Long enough = 2 x Max. Seg. Lifetime

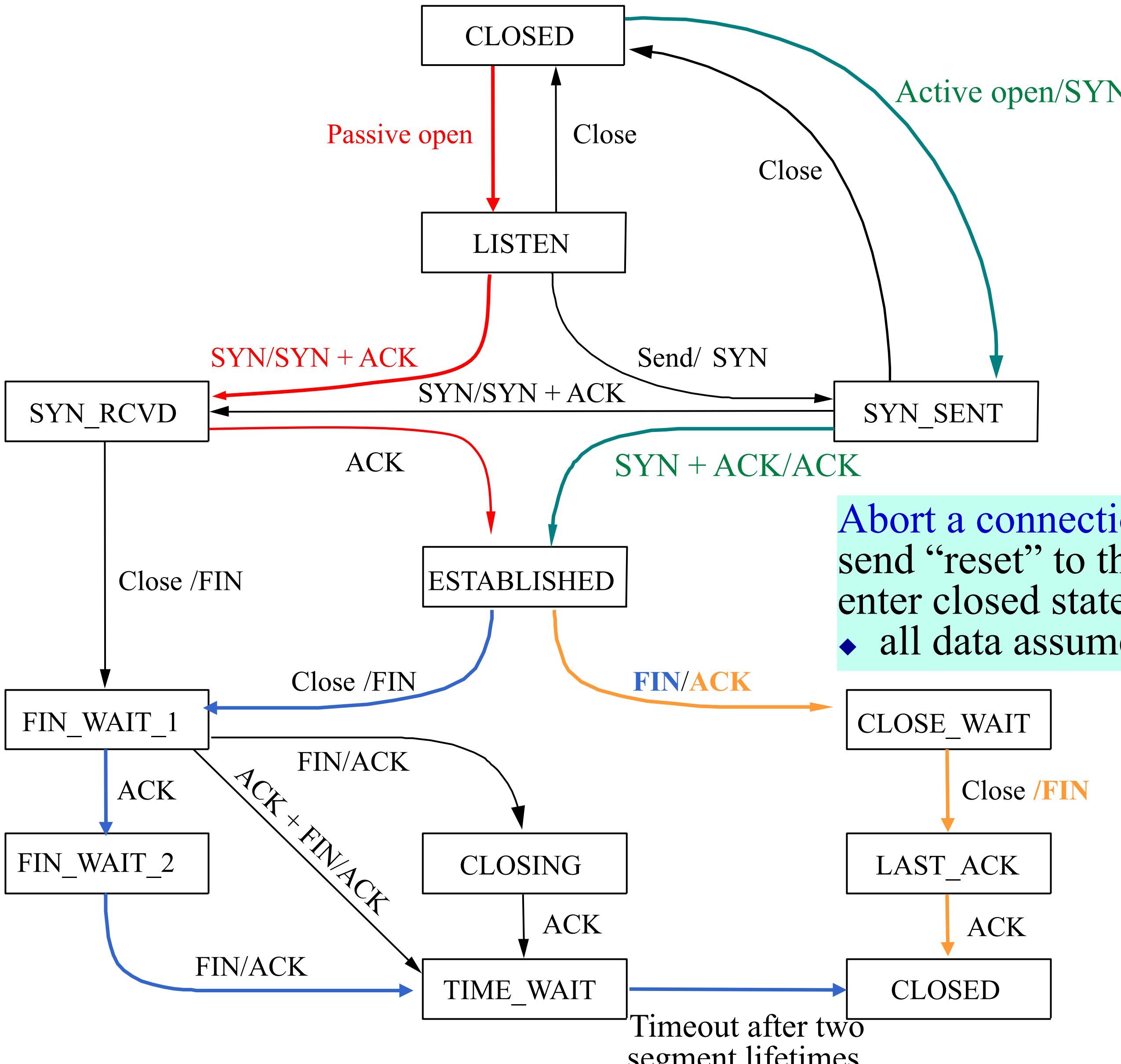


netstat -an -p tcp

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q  Local Address          Foreign Address        (state)
tcp6       0      0  ::1.587                *.*                    LISTEN
tcp4       0      0  127.0.0.1.587          *.*                    LISTEN
tcp6       0      0  ::1.25                 *.*                    LISTEN
tcp4       0      0  127.0.0.1.25           *.*                    LISTEN
tcp4       0      0  131.179.6.105.52914    188.174.252.22.80      SYN_SENT
tcp4       0      0  131.179.6.105.52911    52.201.115.248.443     CLOSE_WAIT
tcp4       0      0  *.5533                 *.*                    LISTEN
tcp4       0      0  *.*                    *.*                    CLOSED
tcp4       0      0  *.5352                 *.*                    LISTEN
tcp4       0      0  *.53                   *.*                    LISTEN
tcp4      31      0  131.179.6.105.52896    108.160.172.193.443    CLOSE_WAIT
tcp6       0      0  2607:f010:2e9:4:.52894 2607:f8b0:4007:8.443   ESTABLISHED
tcp4       0      4  131.179.6.105.52893    77.38.186.20.14307     ESTABLISHED
tcp4       0      4  131.179.6.105.52892    77.38.172.214.24731    ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52890 2607:f8b0:4007:8.443   ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52883 2001:668:108:9a4.80    ESTABLISHED
tcp4       0      0  131.179.6.105.52865    74.112.184.85.443      CLOSE_WAIT
tcp4       0      0  131.179.6.105.52835    198.189.255.163.80     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52828    198.189.255.163.80     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52827    198.189.255.163.80     CLOSE_WAIT
tcp6       0      0  2607:f010:2e9:4:.52798 2607:f8b0:4007:8.443   ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52797 2607:f8b0:4007:8.80    CLOSE_WAIT
tcp4       0      0  131.179.6.105.52794    74.112.185.182.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52793    74.112.185.182.443     CLOSE_WAIT
tcp6       0      0  2607:f010:2e9:4:.52603 2607:f8b0:4007:8.443   CLOSE_WAIT
tcp4       0      0  131.179.6.105.52565    17.110.241.16.993      ESTABLISHED
tcp4      31      0  131.179.6.105.52547    54.192.139.170.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52479    13.94.234.1.443        ESTABLISHED
tcp6       0      0  2607:f010:2e9:4:.52439 2607:f8b0:400e:c.5228  ESTABLISHED
tcp4       0      0  131.179.6.105.52437    74.112.184.86.443      ESTABLISHED
tcp4       0      0  131.179.6.105.52401    107.152.24.197.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52387    131.253.34.234.443     ESTABLISHED
tcp4       0      0  131.179.6.105.52382    216.58.217.205.443     CLOSE_WAIT
tcp4       0      0  131.179.6.105.52378    173.194.202.125.5222   ESTABLISHED
tcp4       0      0  131.179.6.105.52329    74.125.28.109.993      ESTABLISHED
tcp4       0      0  131.179.6.105.52305    17.110.226.165.5223    ESTABLISHED
tcp4       0      0  131.179.6.105.52303    65.52.108.74.443       ESTABLISHED
tcp4       0      0  131.179.6.105.52299    162.125.17.3.443       ESTABLISHED
tcp4       0      0  131.179.6.105.52297    17.110.241.16.993      ESTABLISHED
tcp4       0      0  131.179.6.105.52295    194.68.30.24.4070      ESTABLISHED
tcp4       0      0  131.179.6.105.52293    17.110.228.92.5223     ESTABLISHED
tcp4       0      0  131.179.6.105.52289    91.190.216.55.12350    ESTABLISHED
tcp4       0      0  131.179.6.105.52288    157.55.130.172.40008   ESTABLISHED
tcp4       0      0  131.179.6.105.7313     *.*                    LISTEN
tcp4       0      0  131.179.6.105.53       *.*                    LISTEN
tcp4      31      0  131.179.6.74.52156     199.47.217.97.443      CLOSE_WAIT
tcp4       0      0  131.179.6.74.51067     107.152.24.197.443     CLOSE_WAIT
tcp4       0      0  131.179.6.74.51065     107.152.24.197.443     CLOSE_WAIT
tcp4       0      0  131.179.6.74.51053     173.194.202.125.5222   ESTABLISHED
tcp6       0      0  2607:f010:3f9::1.65165 2607:f8b0:4007:8.443   CLOSE_WAIT
tcp4       0      0  131.179.196.220.64254  74.112.185.87.443      CLOSE_WAIT
tcp6       0      0  2605:e000:1521:5.63475 2607:f8b0:400e:c.5222  ESTABLISHED
tcp6       0      0  2605:e000:1521:5.63473 2607:f8b0:400e:c.5222  ESTABLISHED
tcp6       0      0  2605:e000:1521:5.54836 2607:f8b0:4007:8.443   CLOSE_WAIT
tcp4       0      0  131.179.196.220.62360  131.179.196.228.17500  ESTABLISHED
....
```

TCP state-transition diagram

rfc793_state_machine.pptx

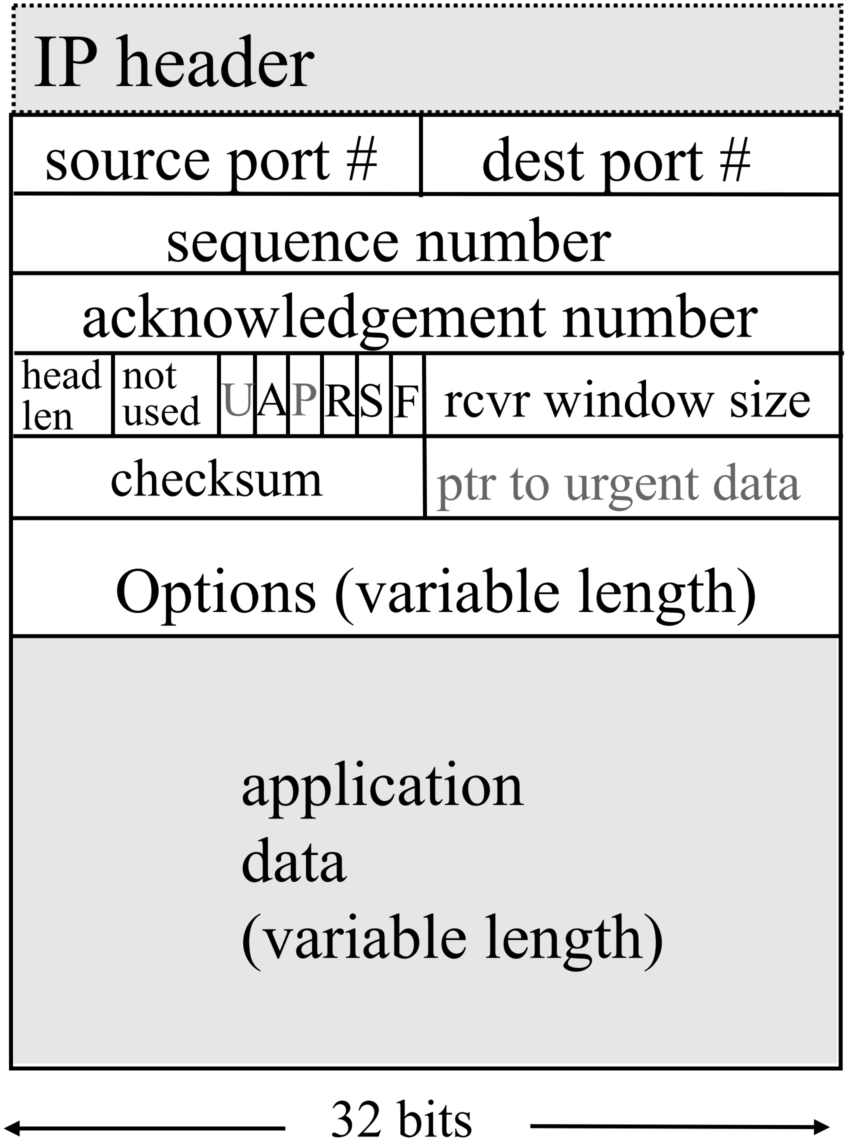


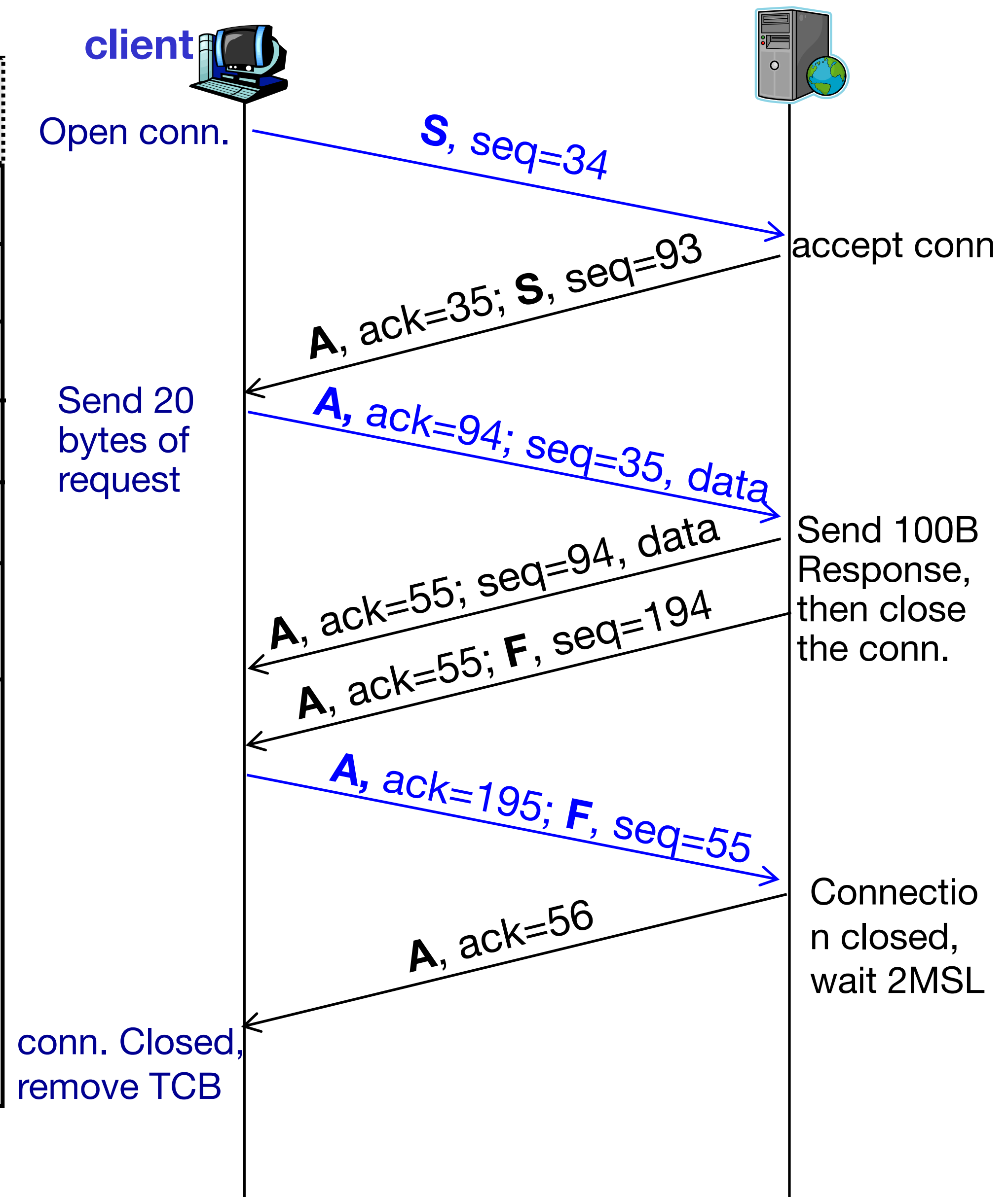
Abort a connection: either end may send “reset” to the other end, then enter closed state immediately

- all data assumed lost

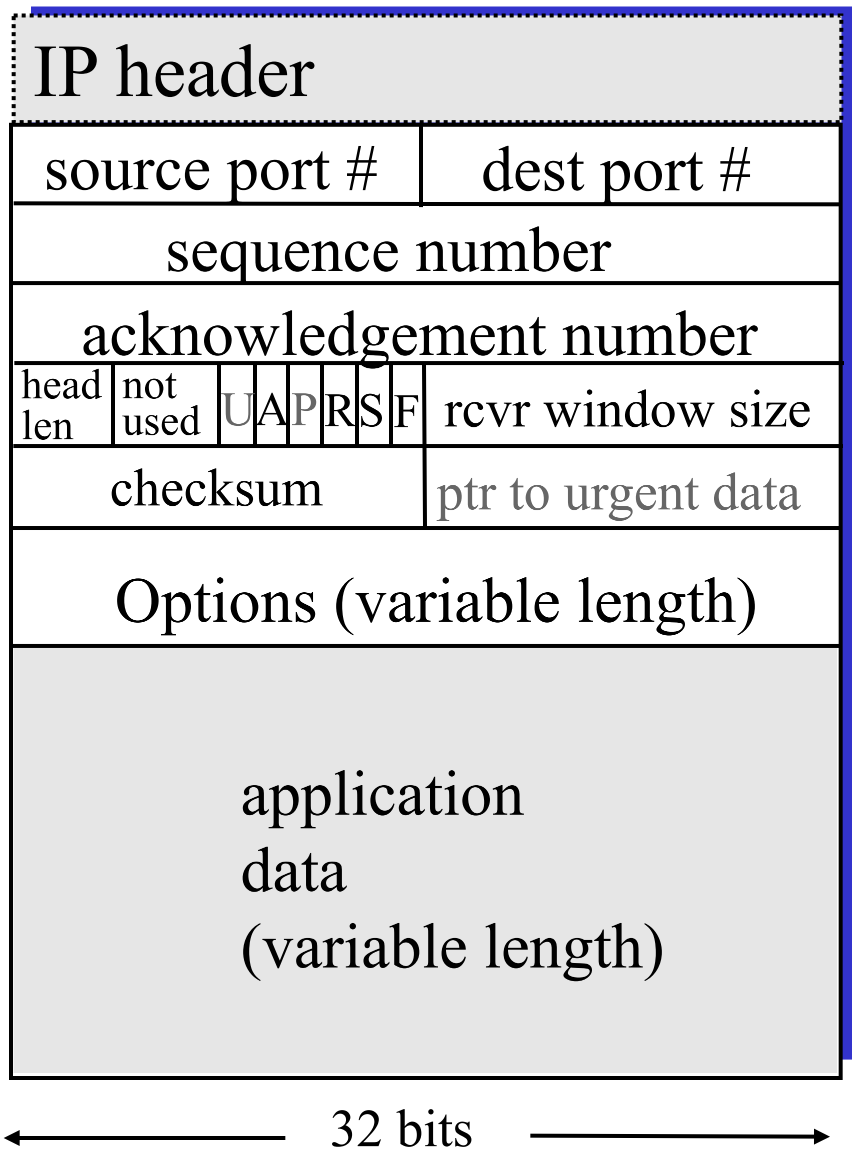
An HTTP 1.0 connection example

important





TCP segment format



TCP Functions

- ✧ Connection set up
- ✧ Connection tear down
- ✧ Reliable delivery
  - For both data and control (SYN, FIN msgs)
  - Need Seq & Ack numbers
  - Need retransmission timer
- ✧ Flow control
- ✧ Congestion control

TCP Flow Control

Flow control: Prevent sender from overrunning receiver by transmitting too much too fast

receiver: informs sender of (dynamically changeable) amount of free buffer space (**`RcvWindow`** field in TCP header)

sender: keeps the amount of transmitted, unACKed data no more than most recently received **`RcvWindow`** value