CS 305: Computer Networks

Fall 2024

Lecture 7:   Transport Layer

Ming Tang

Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

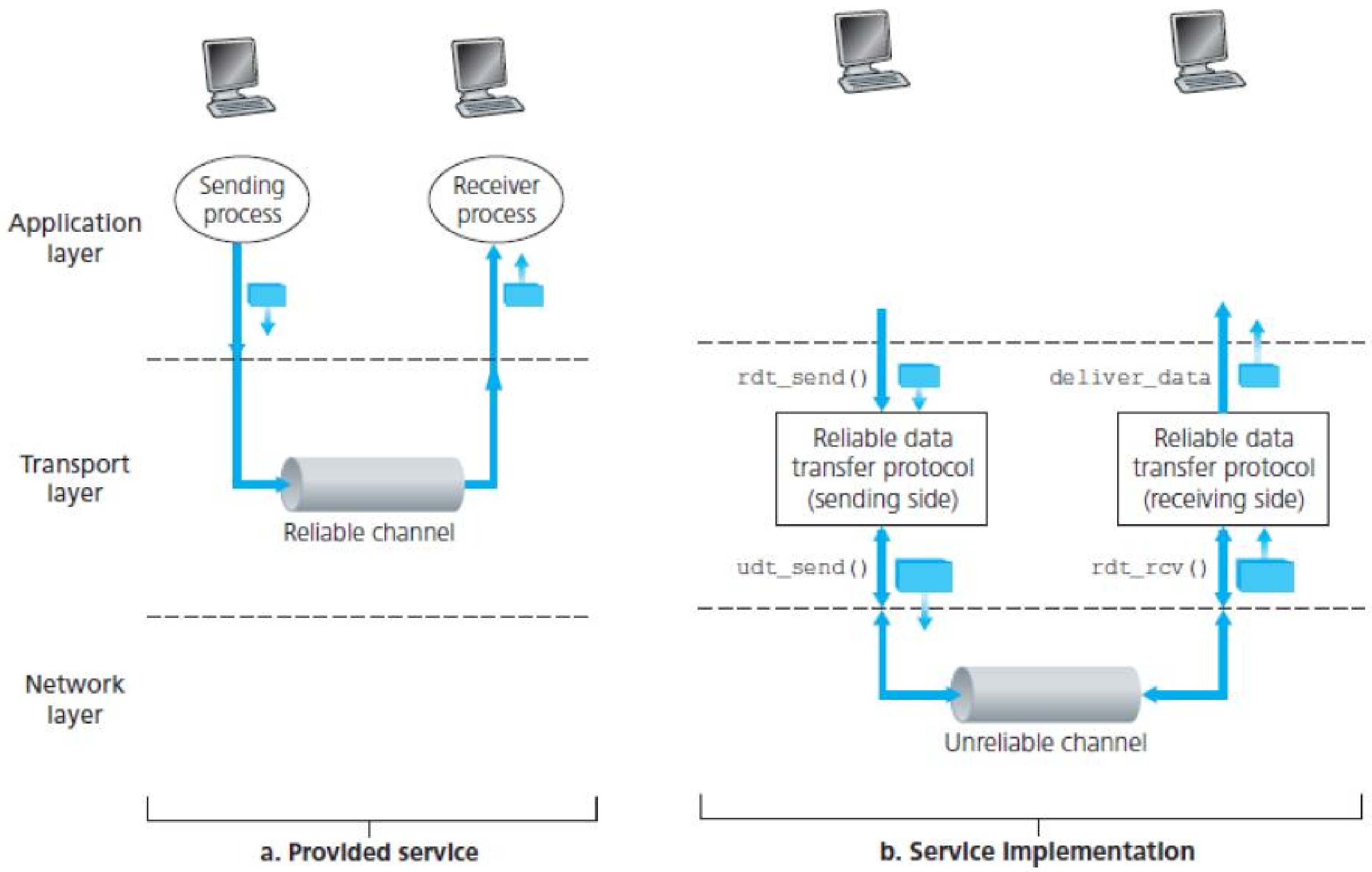
3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 TCP congestion control

Reliable Data Transfer (rdt)

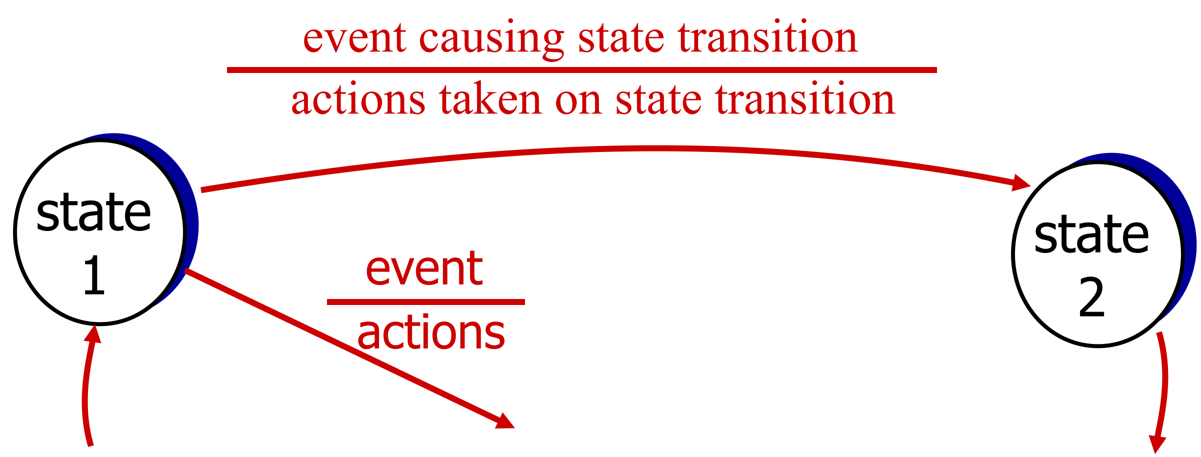


Reliable data transfer: getting started

We'll:

- **Incrementally** develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only **unidirectional data transfer**
  - but control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event



Overview

Roadmap:

- Perfectly reliable channel： rdt1.0
- Channel with bit error:
  - bit error in packet: rdt 2.0
  - bit error in ACK: 2.1
  - NAK-free: 2.2
- Lossy channel: rdt 3.0

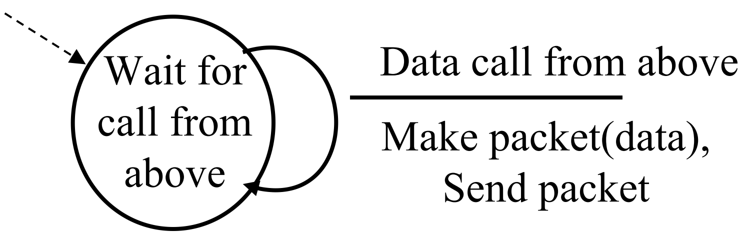
Summary of Techniques

- Checksum
- Sequence number
- ACK packets
- Retransmission
- Timeout

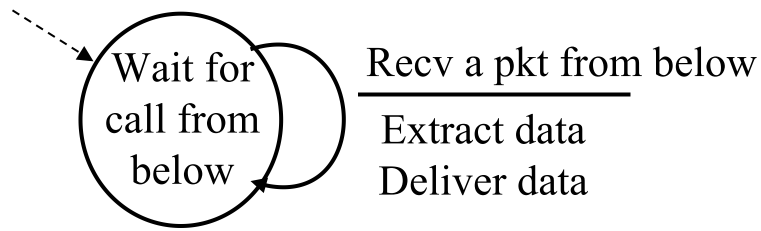
rdt1.0: reliable transfer over a reliable channel

- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

- **Rdt 1.0:**
  - sender sends data into underlying channel
  - receiver reads data from underlying channel
  - Reliable channel, no need for feedback （no control message）

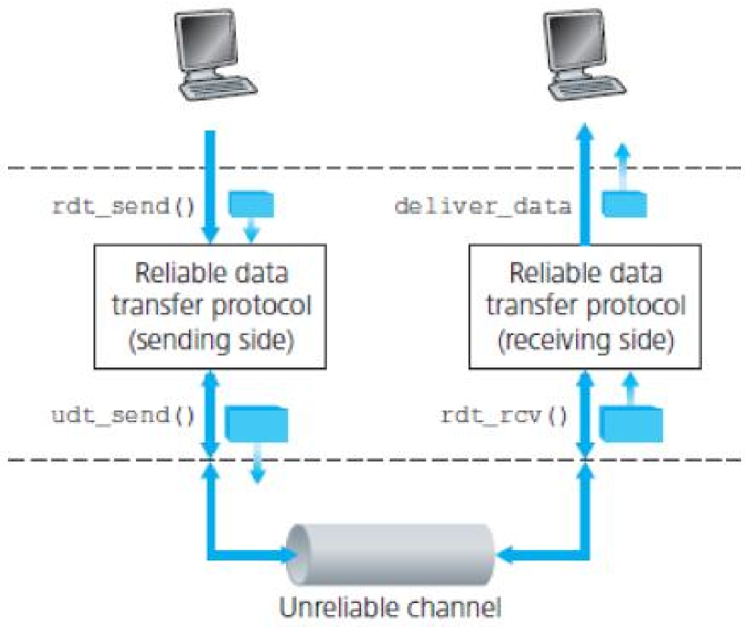


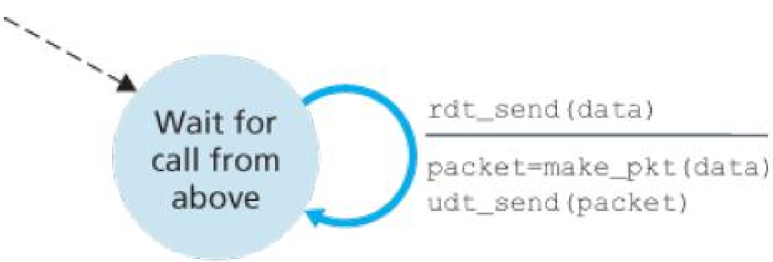
sender



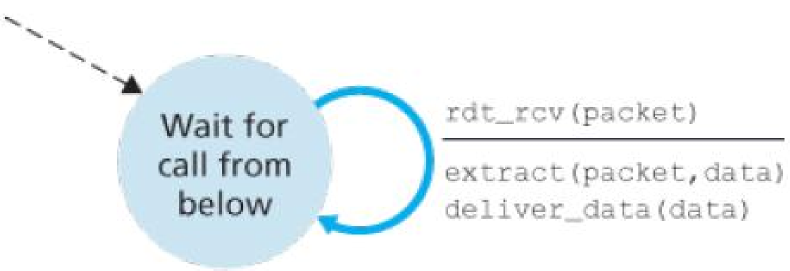
receiver

rdt1.0: reliable transfer over a reliable channel





sender

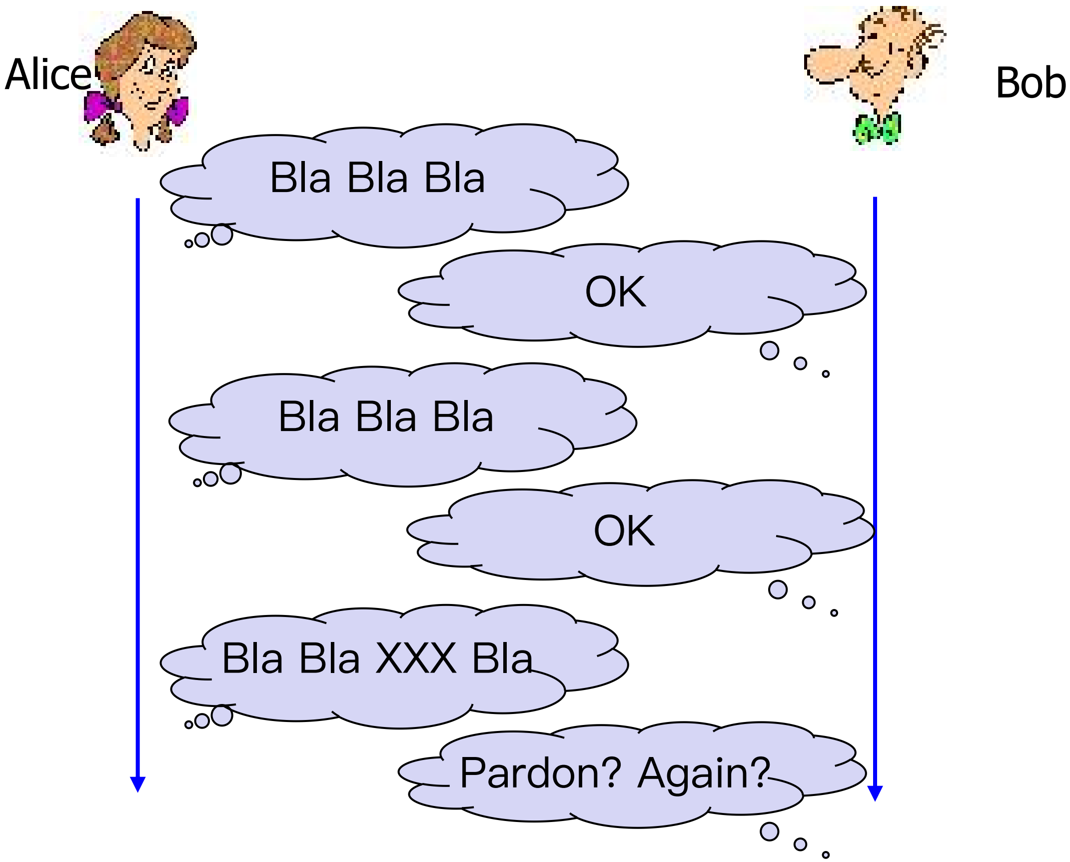


receiver

rdt2.0: channel with bit errors

- Underlying channel may flip bits (0 → 1) in packet

How do humans recover from "errors" during conversation?



rdt2.0: channel with bit errors

- Underlying channel may flip bits (0 → 1) in packet

How do humans recover from "errors" during conversation?

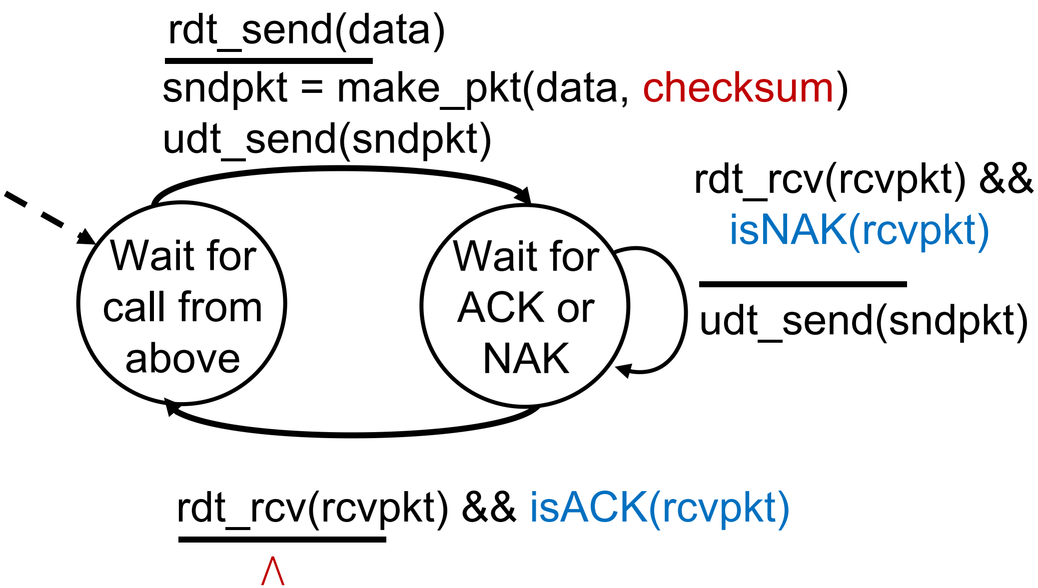
- The question: how to recover from errors?
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender
  - retransmission

rdt2.0: channel with bit errors

- Key mechanisms:
  - error detection
  - feedback: control msgs (ACK, NAK) from receiver to sender
  - retransmission

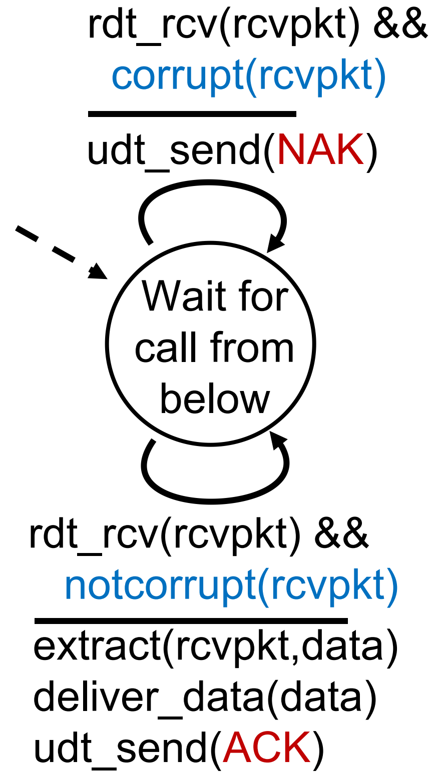
- Error detection: checksum
- Feedback messages:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK

rdt2.0: FSM specification



sender

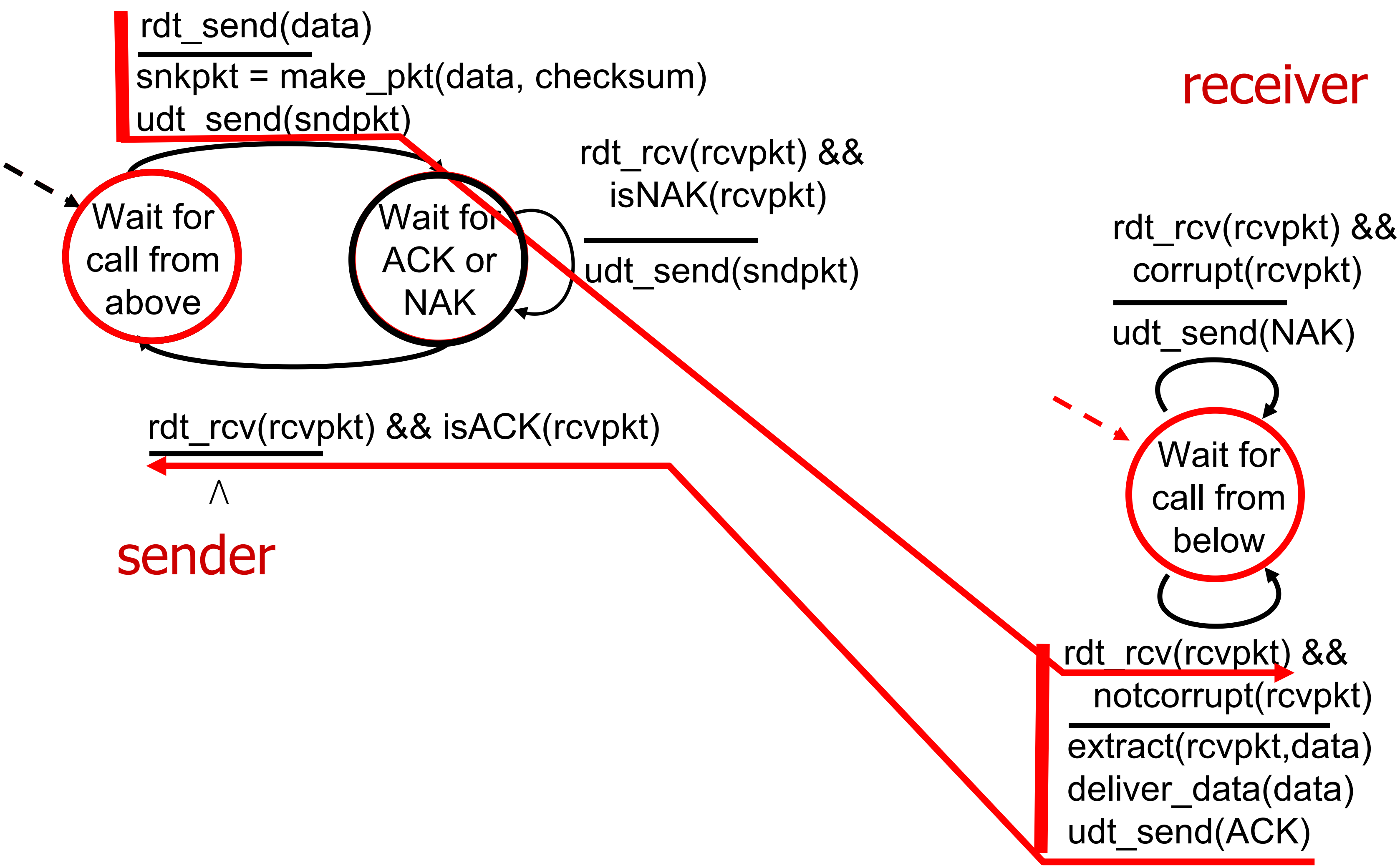
receiver



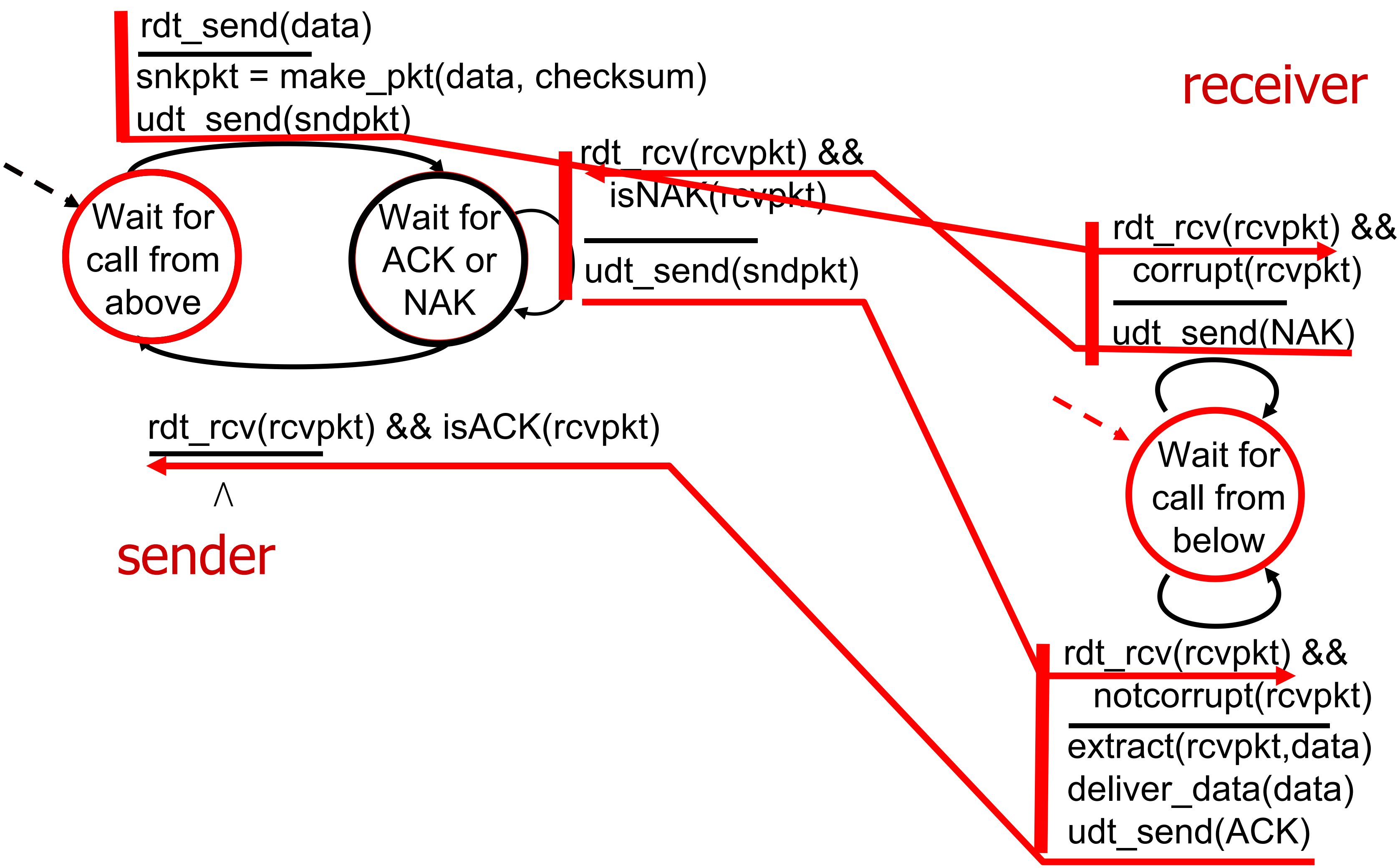
Stop and wait

Sender sends one packet, then waits for receiver response

rdt2.0: operation with no errors



rdt2.0: error scenario



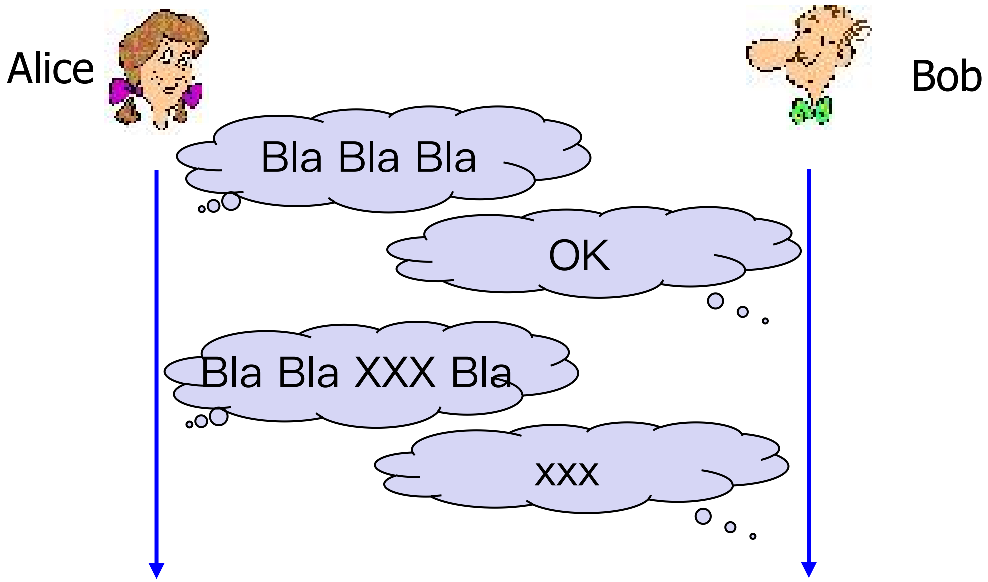
rdt2.0 has a fatal flaw!

The possibility that ACK or NAK packet could be corrupted:

- Checksum bits

Handling corrupted ACKs or NAKs:

- Option 1: “blabla…”, “OK”, “What did you say?”, “OK”
  - “What did you say?”, “What did you say?”, …
- Option 2: add enough checksum to recover
- Option 3: when garbled ACK or NAK, retransmit



rdt2.0 has a fatal flaw!

The possibility that ACK or NAK packet could be corrupted:

- Checksum bits

Handling corrupted ACKs or NAKs:

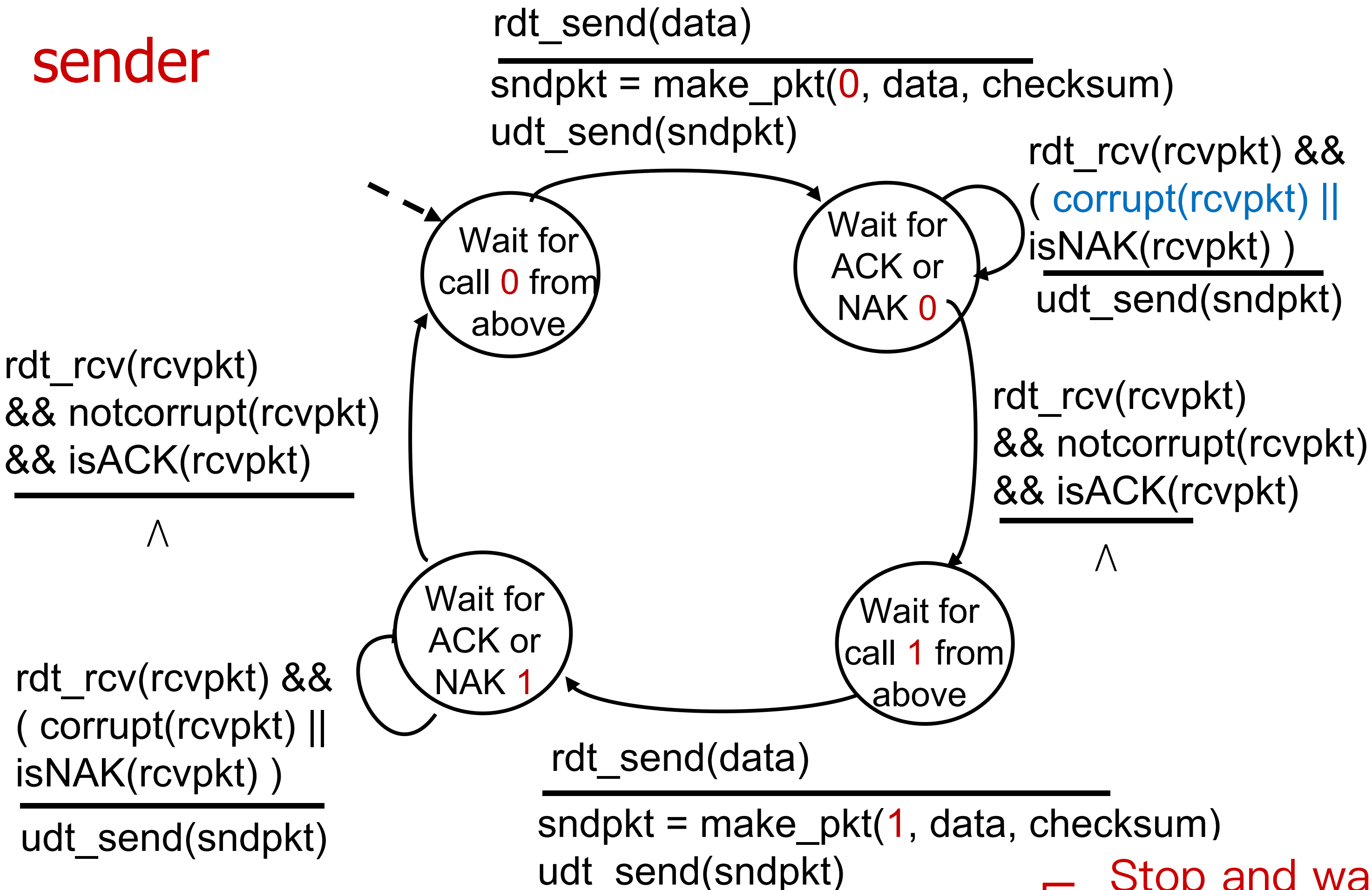
- Option 1: "blabla…", "OK", "What did you say?", "OK"
  - "What did you say?", "What did you say?", …
- Option 2: add enough checksum to recover
- **Option 3: when garbled ACK or NAK, retransmit**

Problem: can't just retransmit: new data or retransmission? possible duplicate

Handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

rdt2.1: sender, handles garbled ACK/NAKs



Stop and wait

Sender sends one packet, then waits for receiver response

Two sequence number would be sufficient!

rdt2.1: receiver, handles garbled ACK/NAKs

receiver

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

No sequence number

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

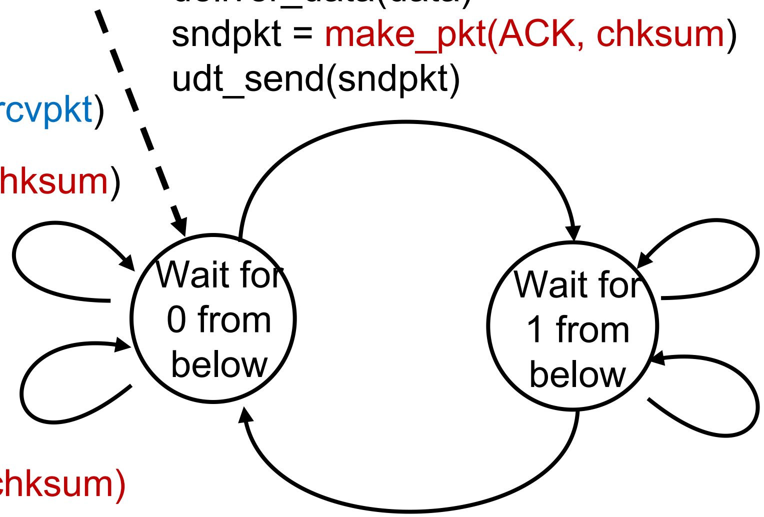
---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)



rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Saying seq1 has been received, please send the next one

rdt2.1: discussion

sender:

- seq # added to pkt
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

receiver:

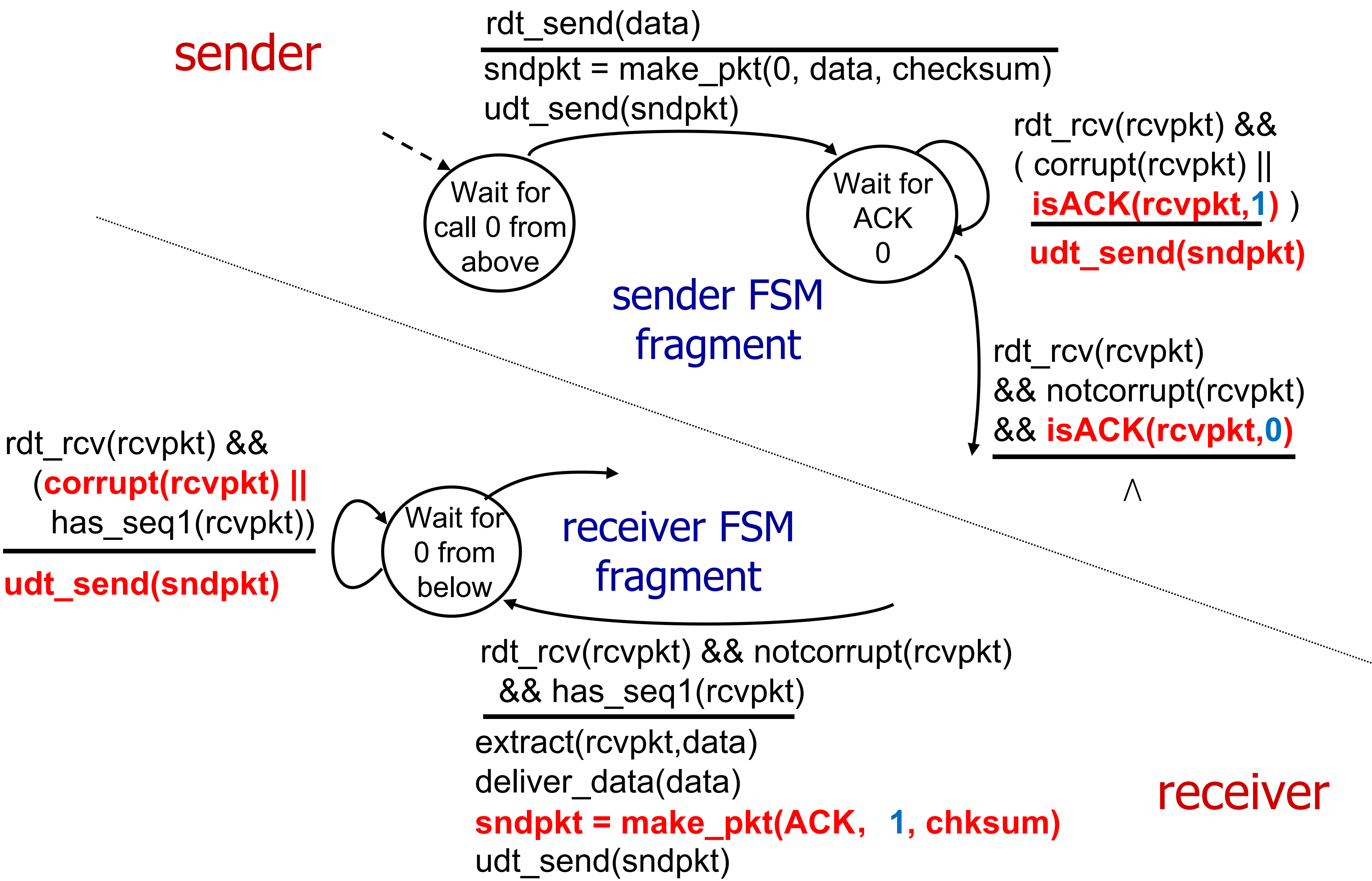
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

Two seq. #'s (0,1) will suffice. Why?

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: channels with errors *and* loss

New assumption: underlying channel can also lose packets (data, ACKs)

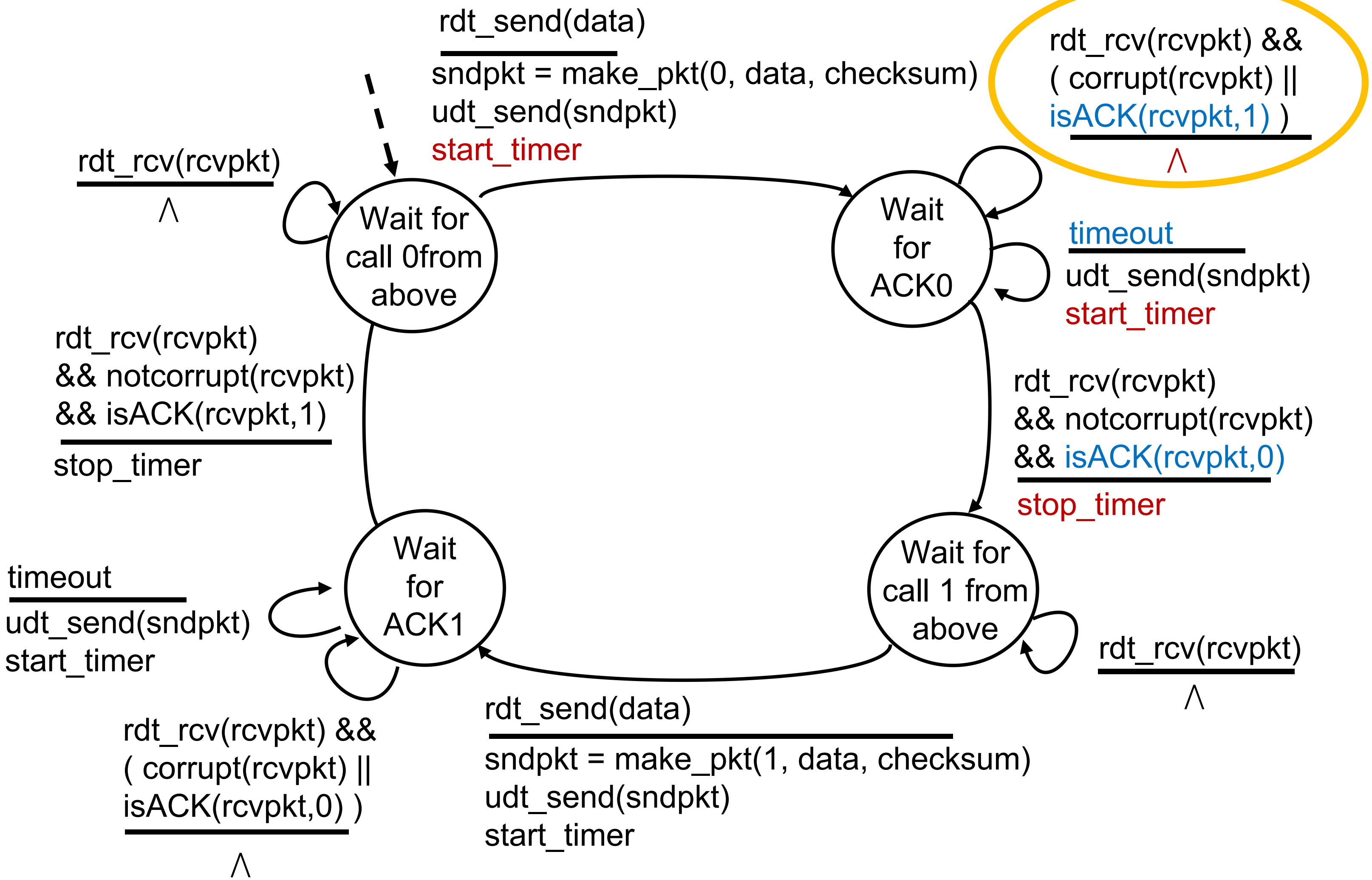
- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

Approach: sender waits "reasonable" amount of time for ACK

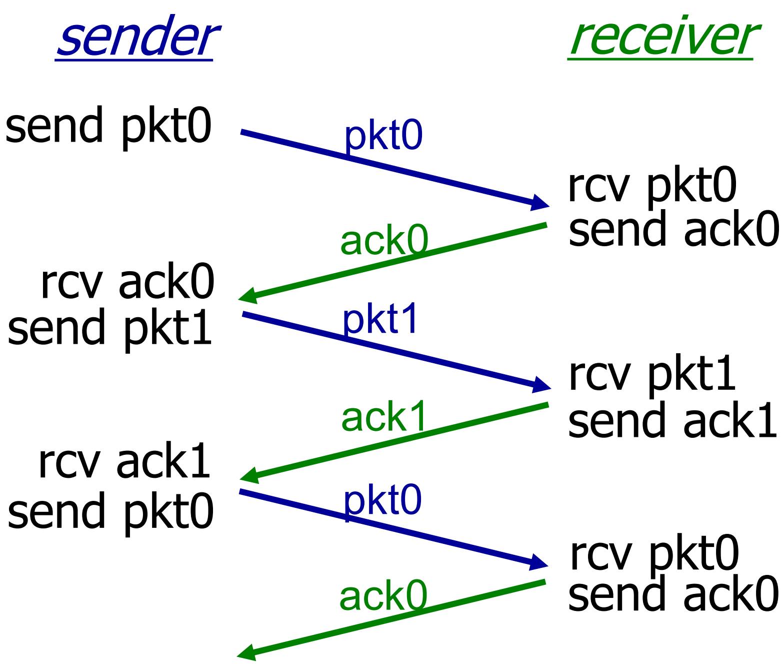
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer
  - start timer, timer interrupt, stop timer

How long should the sender wait?

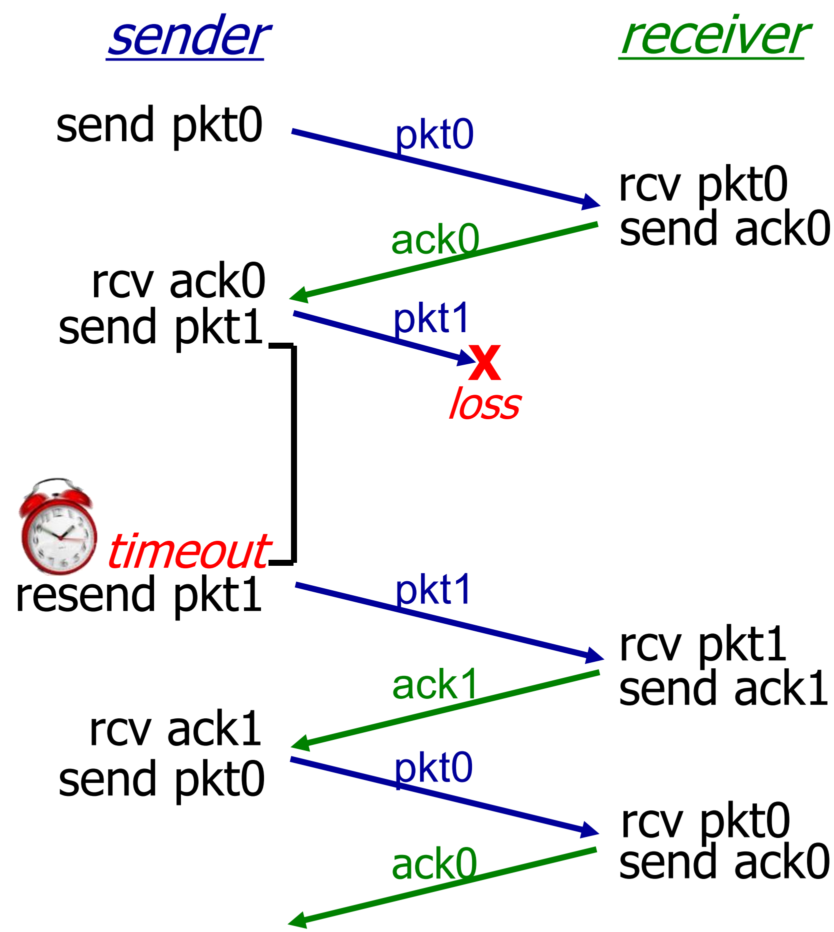
rdt3.0 sender



rdt3.0 in action

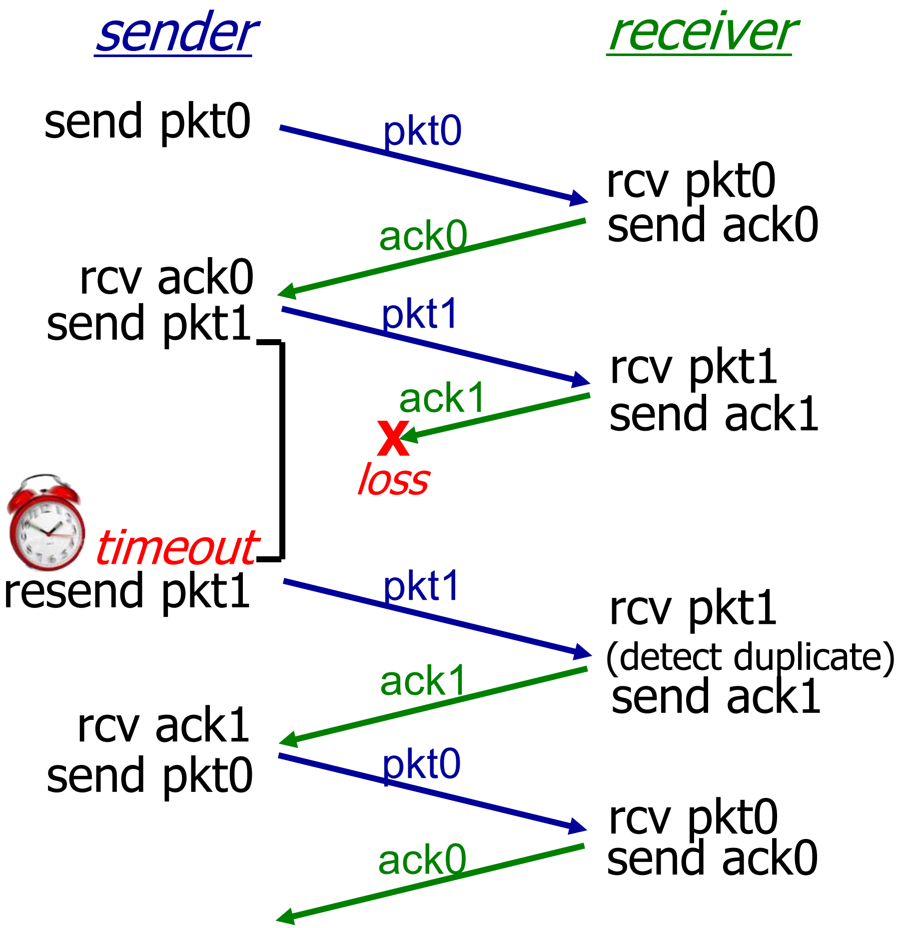


(a) no loss

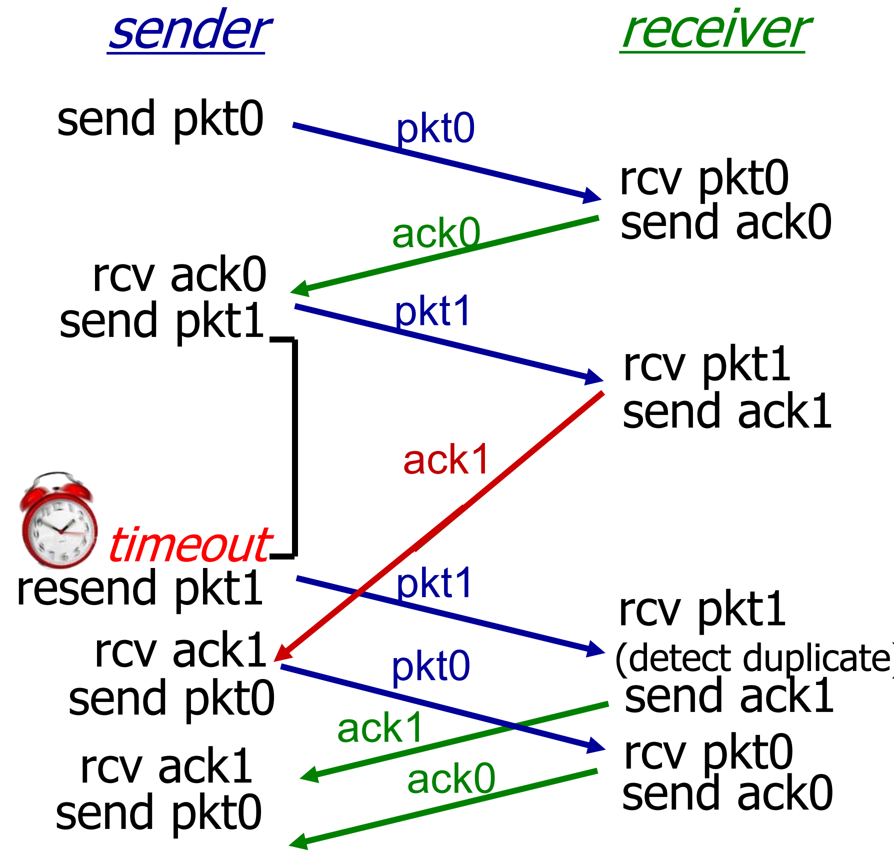


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Summary

Roadmap:

- Perfectly reliable channel： rdt1.0
- Channel with bit error:
  - bit error in packet: rdt 2.0
  - bit error in ACK: 2.1
  - NAK-free: 2.2
- Lossy channel: rdt 3.0

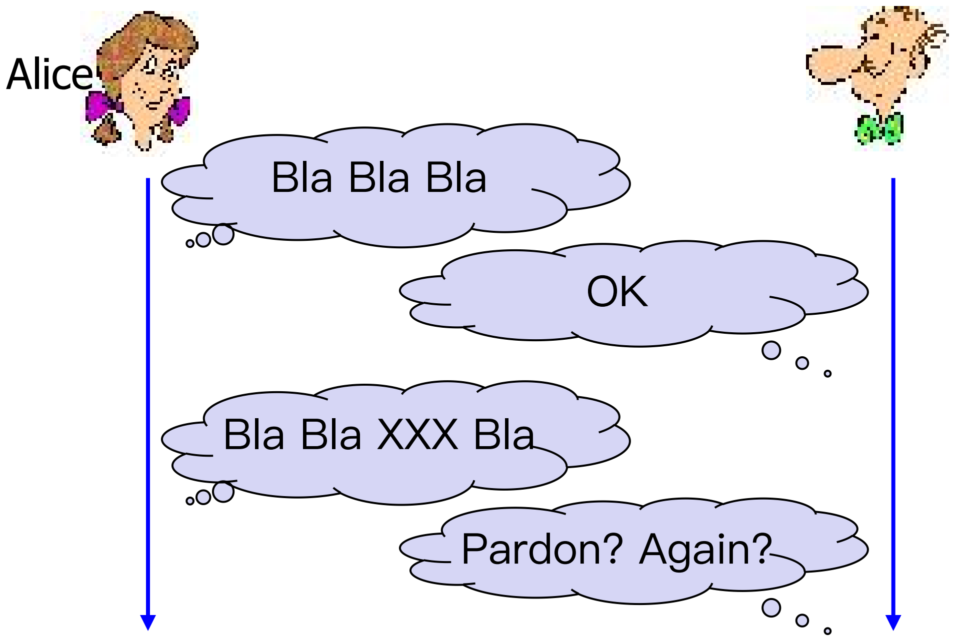
Summary of Techniques

- Checksum
- Sequence number
- ACK packets
- Retransmission
- Timeout

Stop and wait

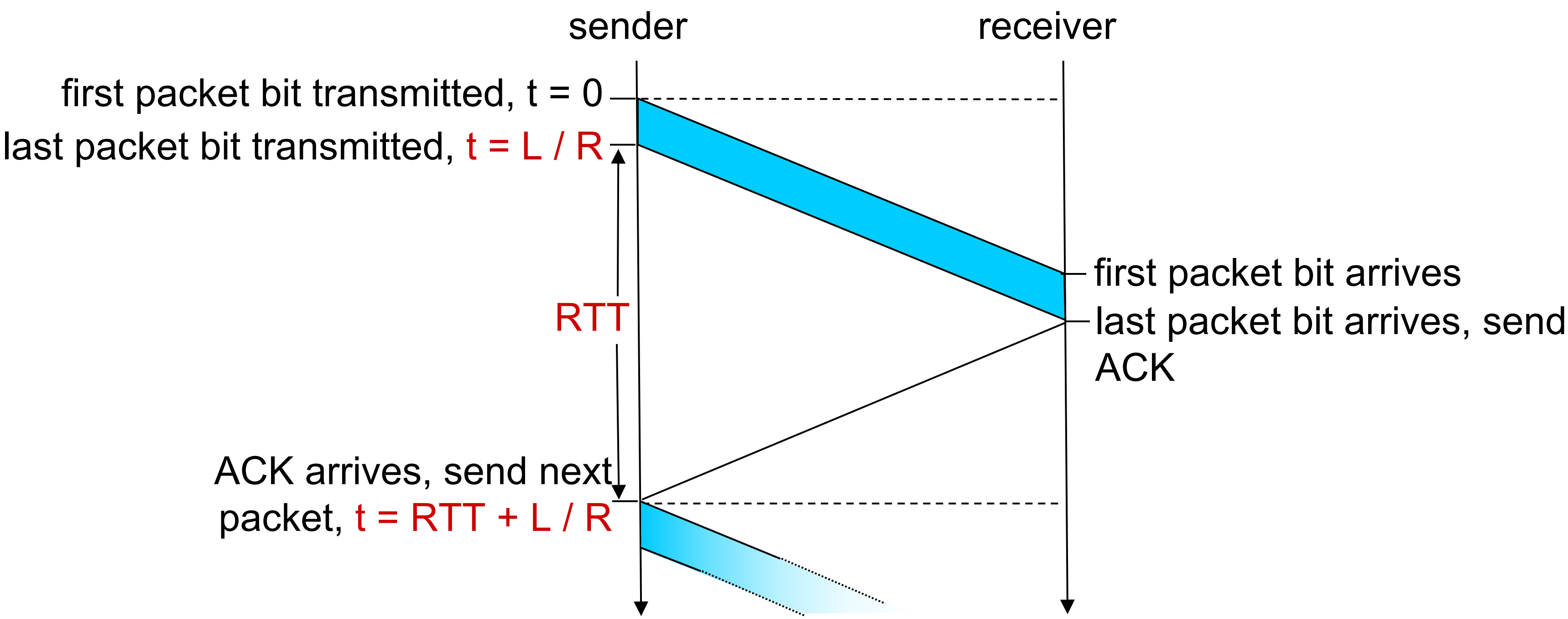
Sender sends one packet, then waits for receiver response

Limitations?



Performance of rdt3.0

- rdt3.0 is correct, but performance is bad
- e.g.: link rate R=1 Gbps, prop. delay T_{pd}=15 ms, packet length L=8000 bit



- Calculate utilization U_{sender}: fraction of time sender busy sending

Performance of rdt3.0

- link rate R=1 Gbps, prop. delay T_{pd}=15 ms, packet length L=8000 bit

$$D_{trans} = t = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - utilization U_{sender}: fraction of time sender busy sending

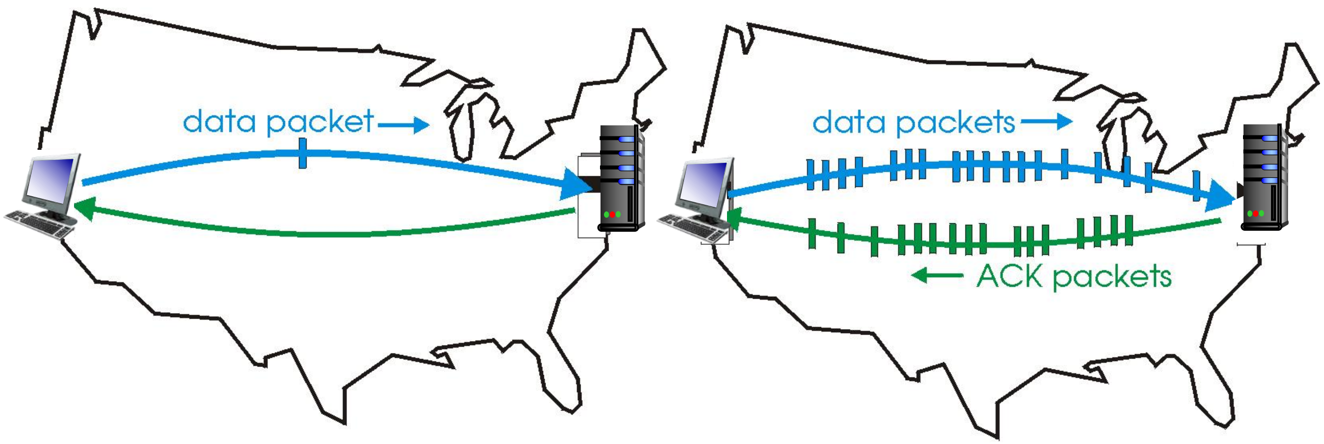
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

  - RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

Pipelined protocols

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

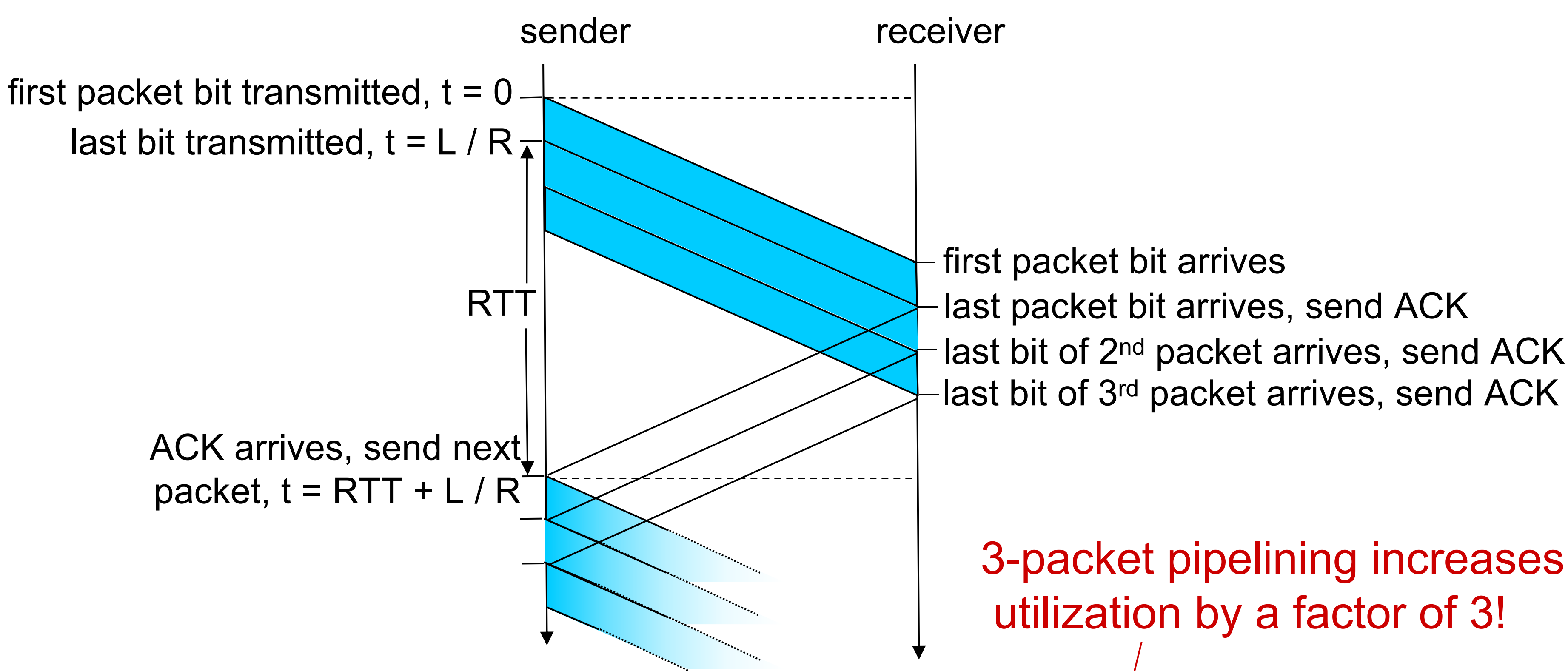
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

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

Pipelining: increased utilization



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

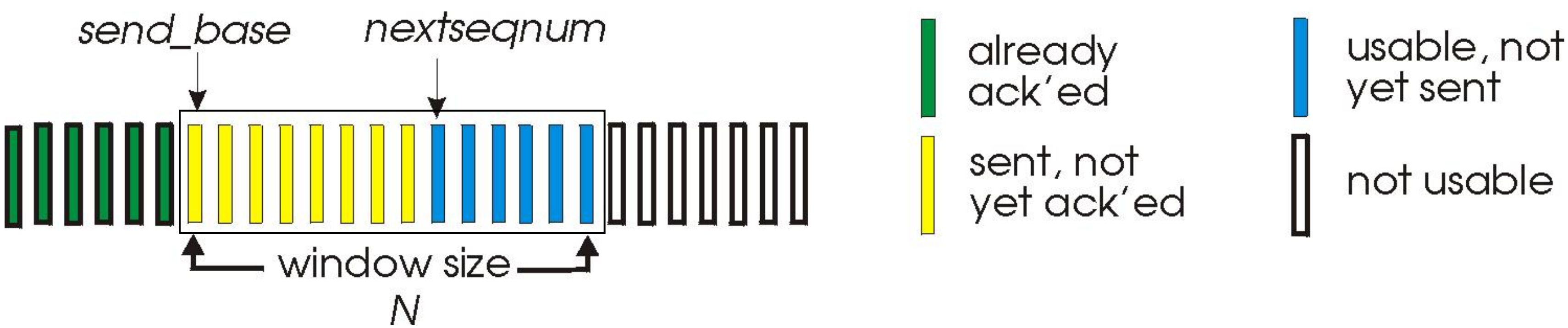
$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.00081$$

Pipelined Protocols

- Go-Back-N
  - Timer for the oldest unACKed packet
  - Cumulative ACK
  - Retransmit all packets in the window
- Selective repeat
  - Timer for each packet in window
  - Individual ACK for each correctly received packets
  - Retransmit only those packets that might be lost or corrupted

Go-Back-N: Sender

- k-bit seq # in pkt header (not 0 or 1): $[0, 2^k - 1]$
- At most N pkts in flight: window size = N, (N consecutive unacked pkts allowed



Sender: When rdt_send() is called from above,

- window is not full: a packet is sent, variables are updated.
- window is full: simply returns the data back to the upper layer

A timer for the oldest transmitted but not yet ACKed packet

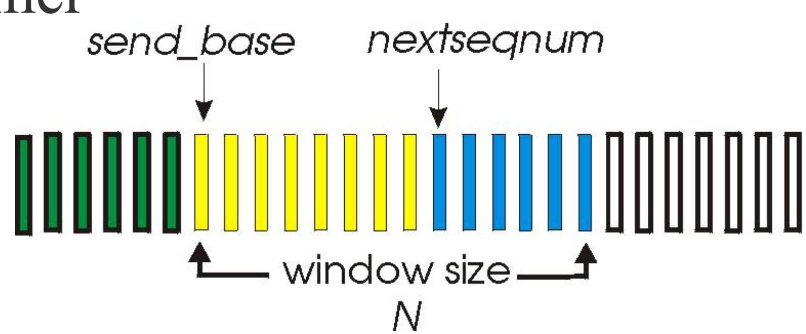
Go-Back-N: Receiver and Timeout

Receiver: Receipt of an ACK.

- Cumulative acknowledgment (ACK)
- ACK(*n*): all packets with a sequence # up to and including *n* have been correctly received at the receiver
- Expect *n* and receive *n*: ACK(*n*)
- Expect *n* and receive others: previous ACK; discard packet

Sender:

- **timeout** occurs: resends all packets in the window;
- ACK(n): slide window; restart timer



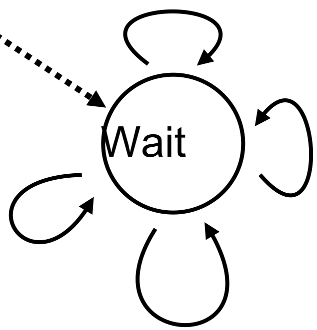
GBN: sender extended FSM

rdt_send(data)

```
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
      start_timer
    nextseqnum++
    }
else
  refuse_data(data)
```

Λ

base=0
nextseqnum=0



timeout

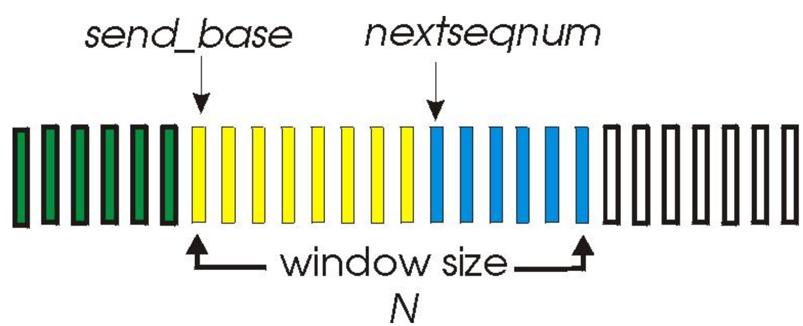
```
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

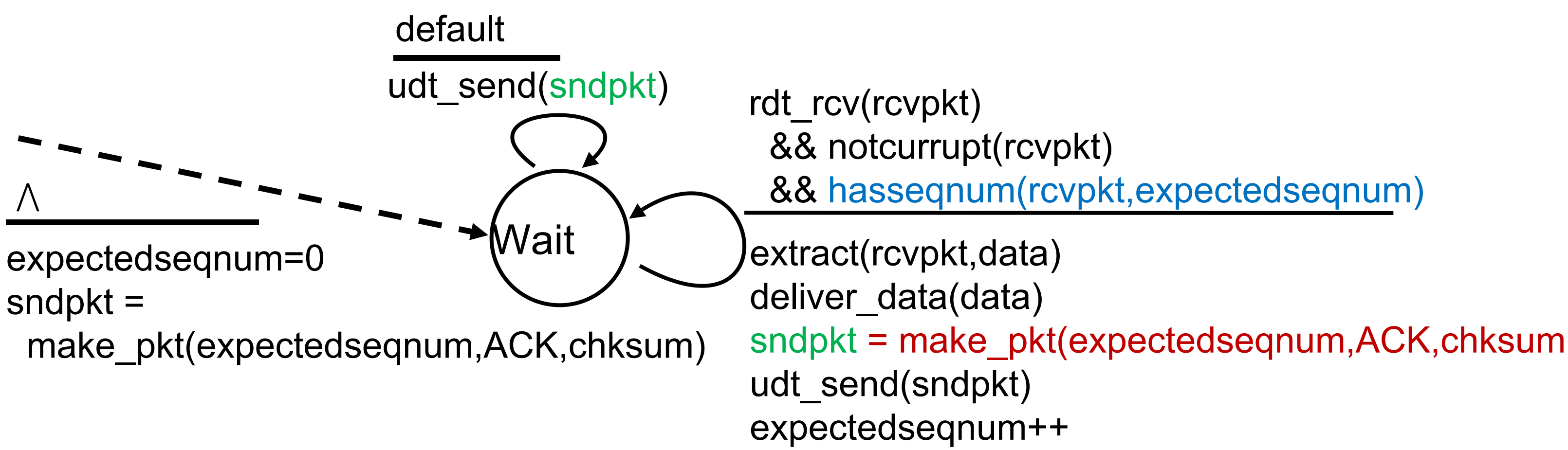
Λ

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```



GBN: receiver extended FSM



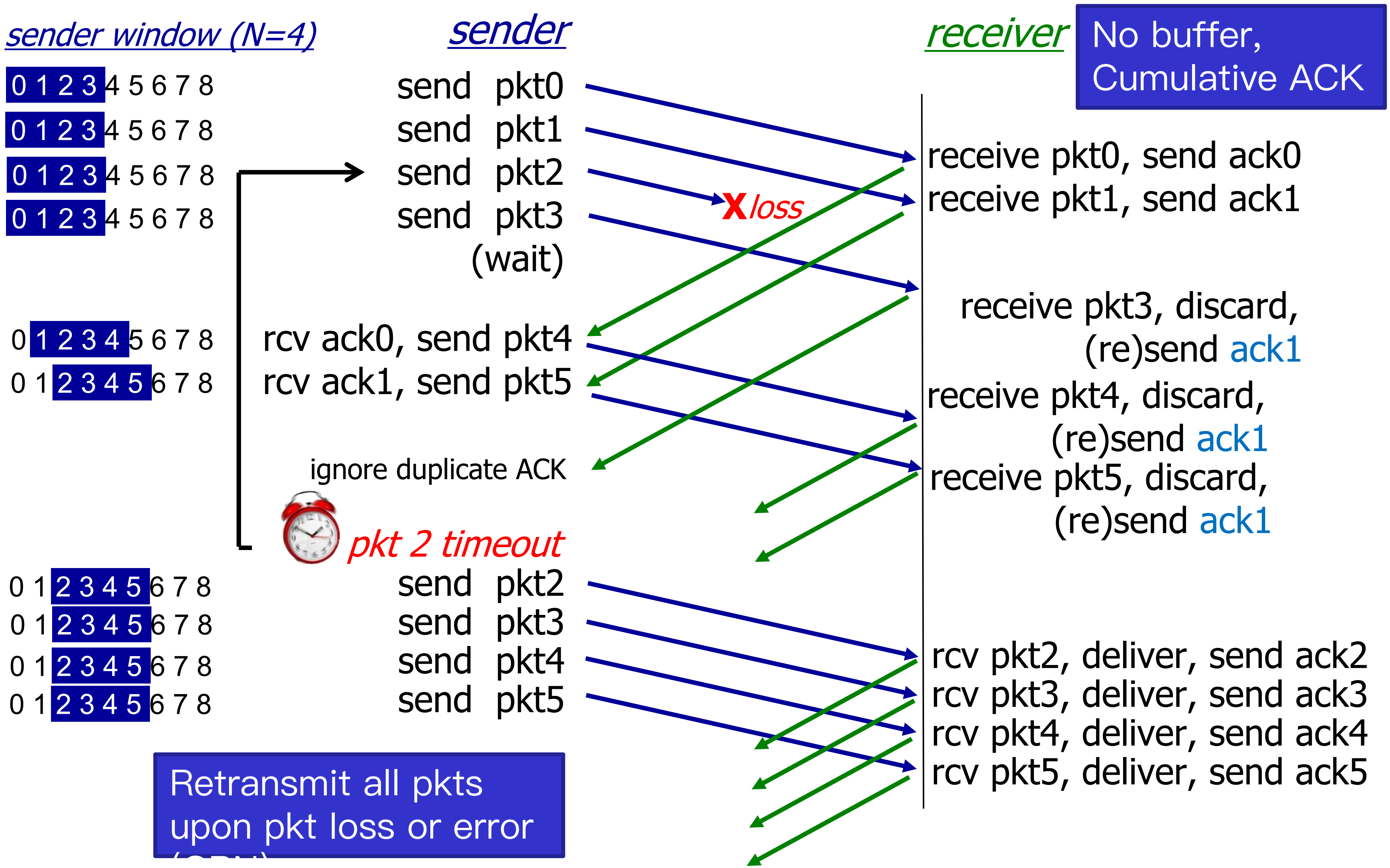
Cumulative ACK: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs

Out-of-order pkt:

- discard (don't buffer): *no receiver buffering!*
- re-ACK pkt with highest in-order seq #

Go-Back-N Recall



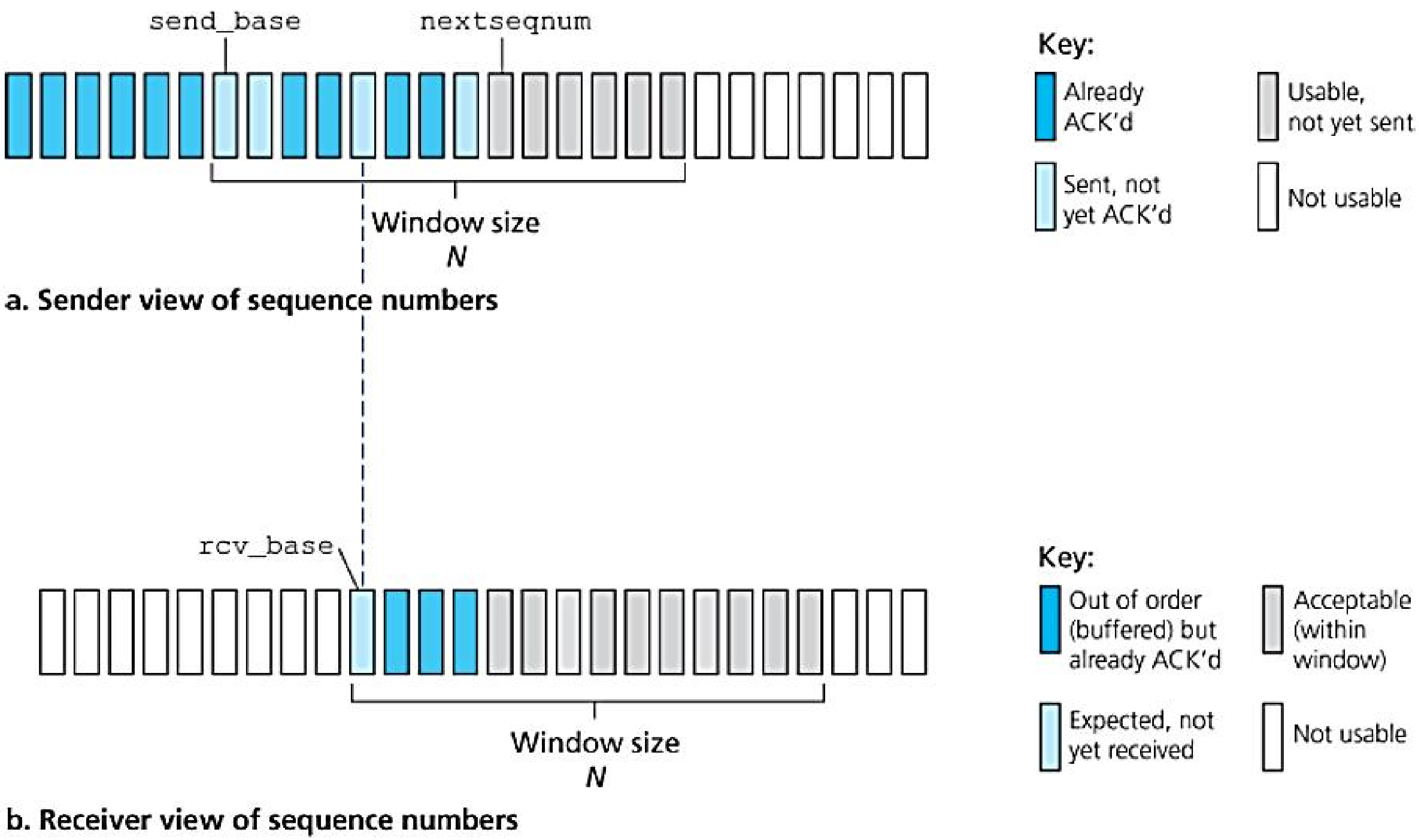
Pipelined Protocols

- Go-Back-N
  - Timer for the oldest unACKed packet
  - Cumulative ACK
  - Retransmit all packets in the window
- Selective repeat
  - Timer for each packet in window
  - Individual ACK for each correctly received packets
  - Retransmit only those packets that might be lost or corrupted

Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
  - Receiver needs to keep track of the out-of-packets
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

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

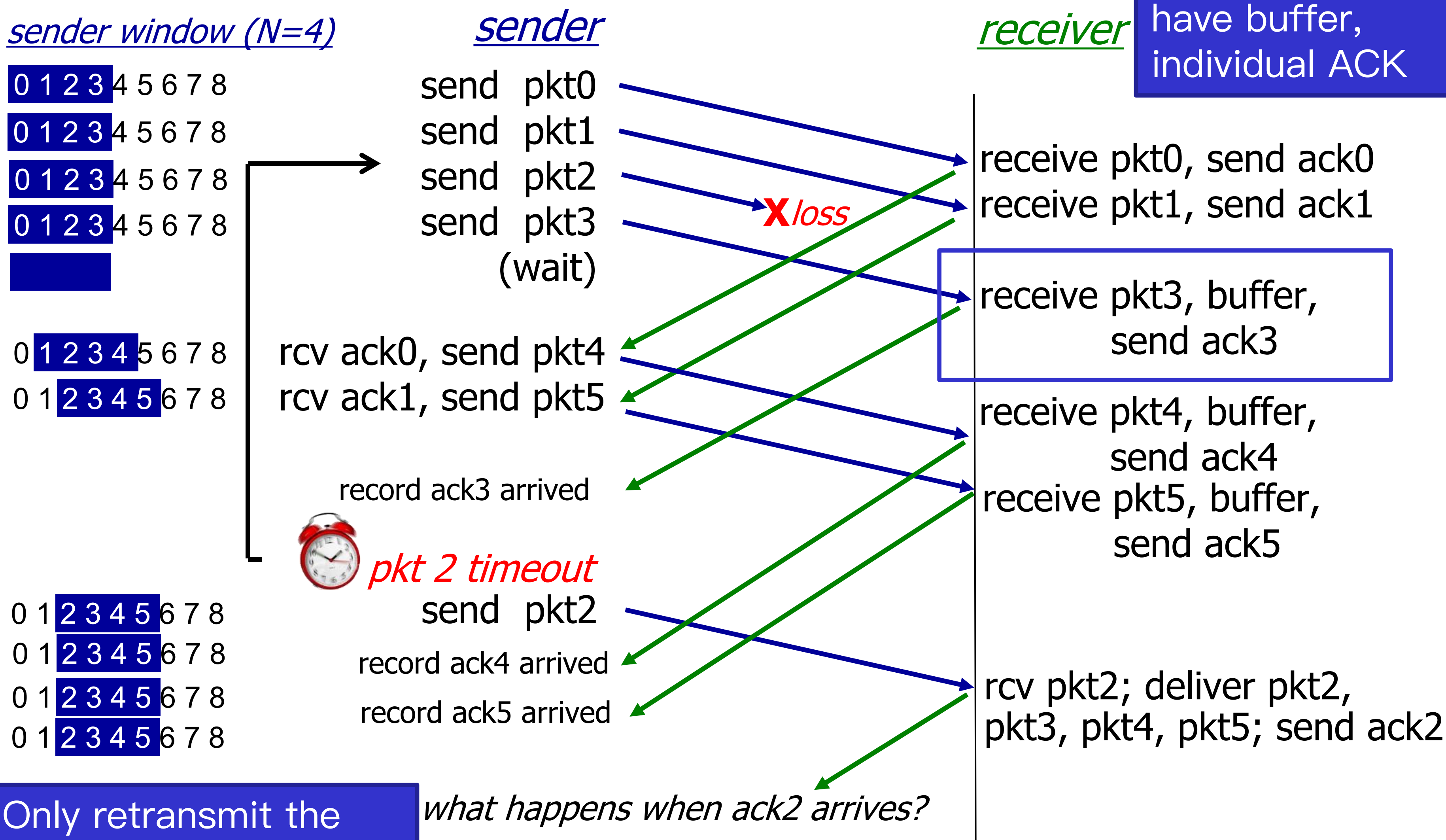
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat



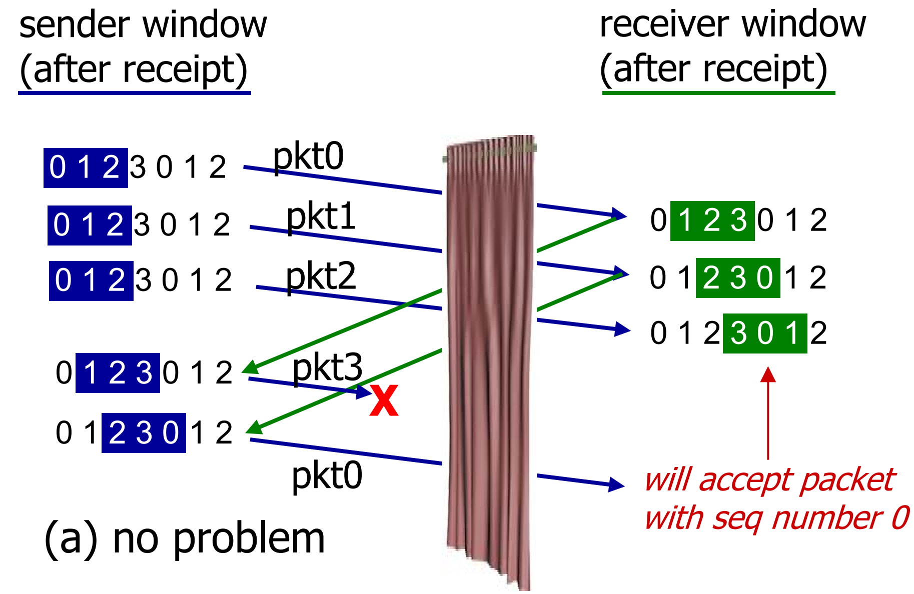
have buffer, individual ACK

Only retransmit the unacked pkt (SR)

Selective repeat: dilemma

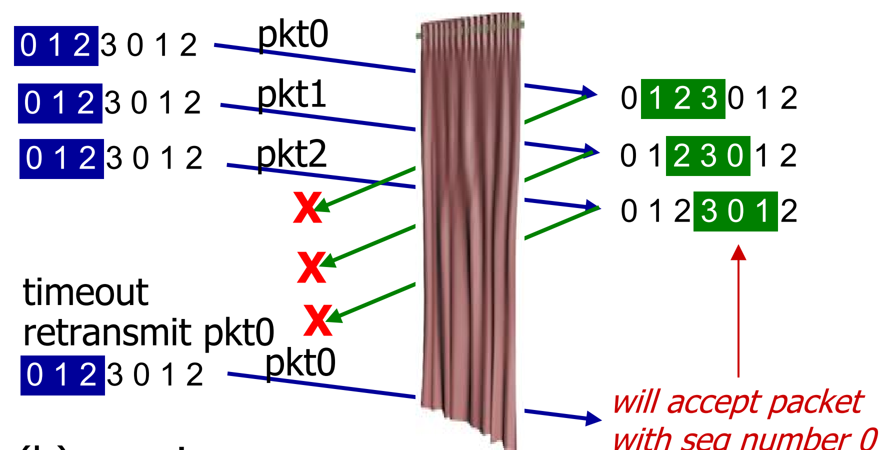
Example:

- seq #' s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)



(a) no problem

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



(b) oops!

Q: what relationship between seq # size and window size to avoid problem in (b)?

The window size must be less than or equal to half the size of the sequence number space for SR protocols.

GBN and SR comparison

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 unack'ed 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

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure, RTT measurement
- reliable data transfer
- flow control
- connection management

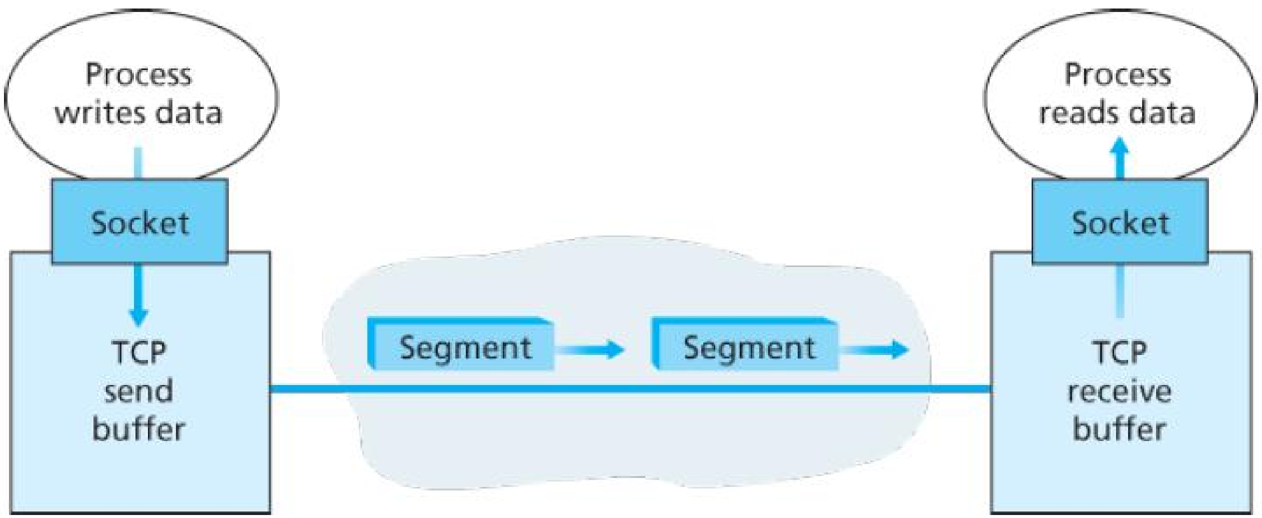
3.6 principles of congestion control

3.7 TCP congestion control

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

- point-to-point:
  - one sender, one receiver
  - No buffers or variables are allocated to network elements between hosts
- reliable, in-order *byte stream*:
  - no "message boundaries"
  - Seq # and Ack # are in unit of byte, rather than pkt
- 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) initiates sender and receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

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

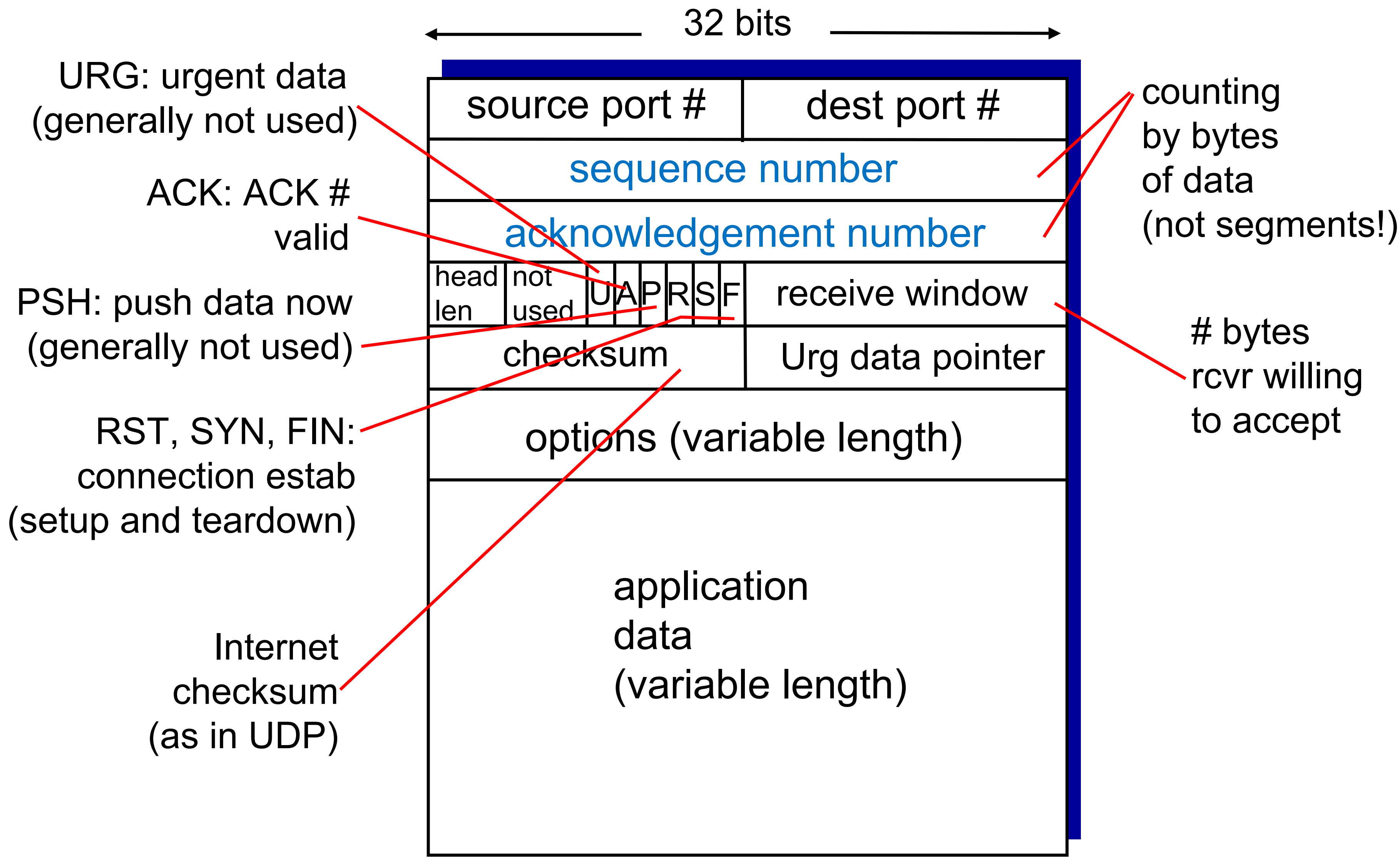


- TCP connection
- TCP grab chunks of data from the sender buffer
  - MSS: maximum segment size, typically 1460 bytes
  - MTU: maximum transmission unit (link-layer frame), typically 1500 bytes
    - Application data + TCP/IP header (typically 40 bytes)
- TCP receives a segment at the other end, place it in receiver buffer
- application reads the stream from the receive buffer

TCP Reliable Data Transfer

- Segment structure
  - Segment format
  - Seq. number and ACKs
  - An example
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

TCP segment structure



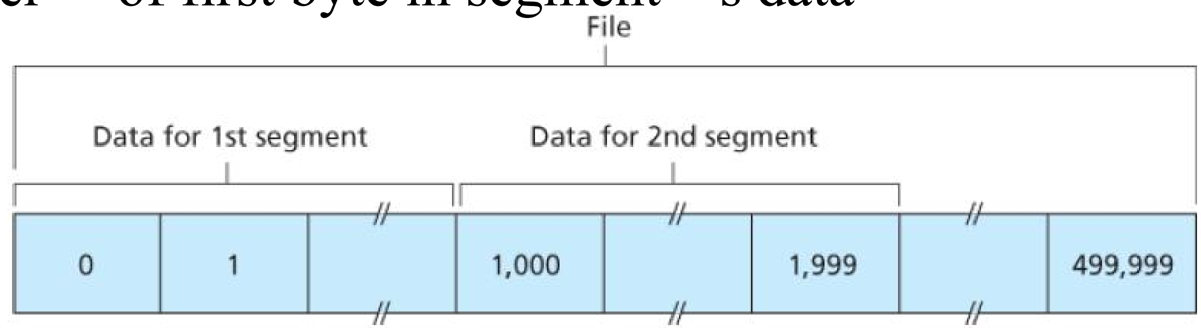
TCP seq. numbers, ACKs

TCP views data as an unstructured, but ordered, stream of bytes.

- Sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments

sequence numbers:

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



acknowledgements:

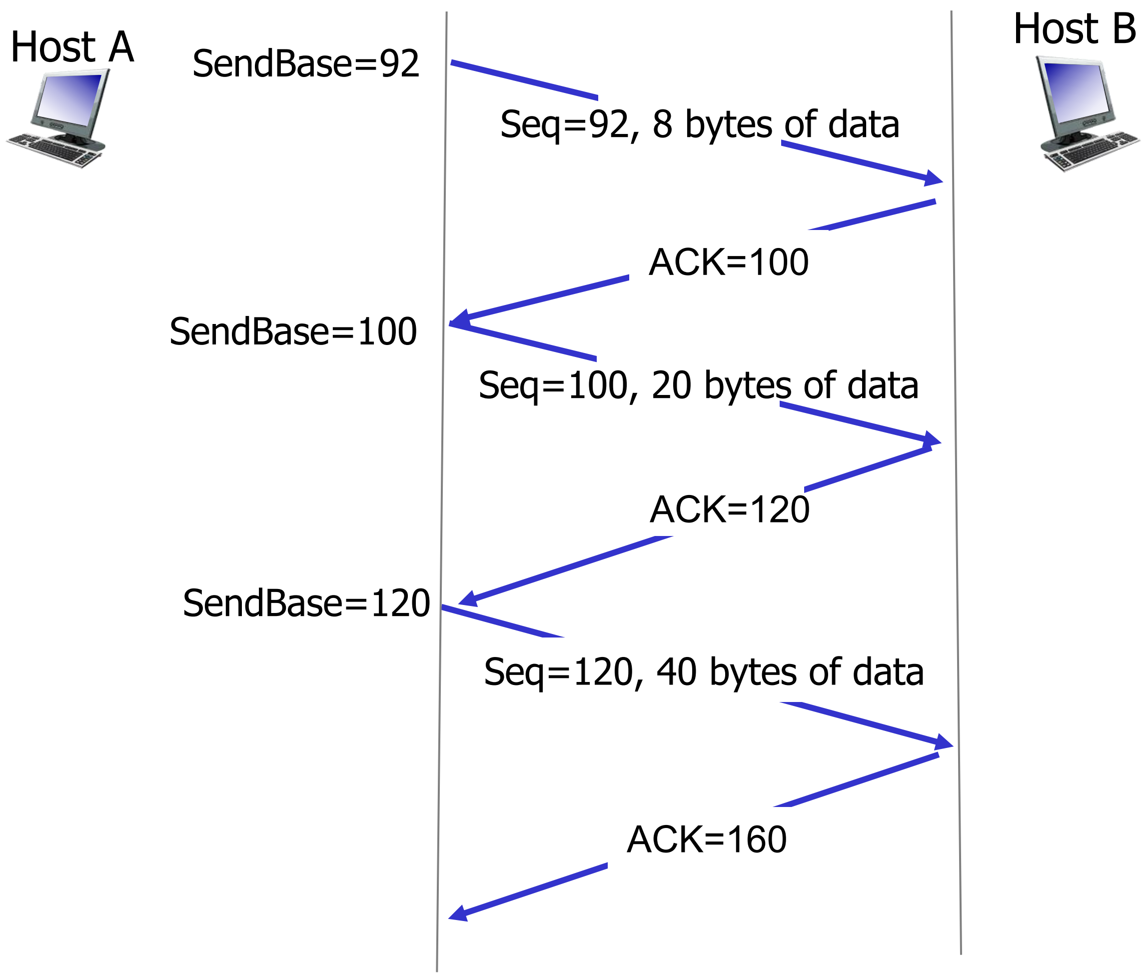
- seq # of next byte expected from other side
  - E.g., receiver has received bytes numbered 0 through 535 and 900 through 1000; then, acknowledgement number is 536.
- cumulative ACK

Q: how receiver handles out-of-order segments

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

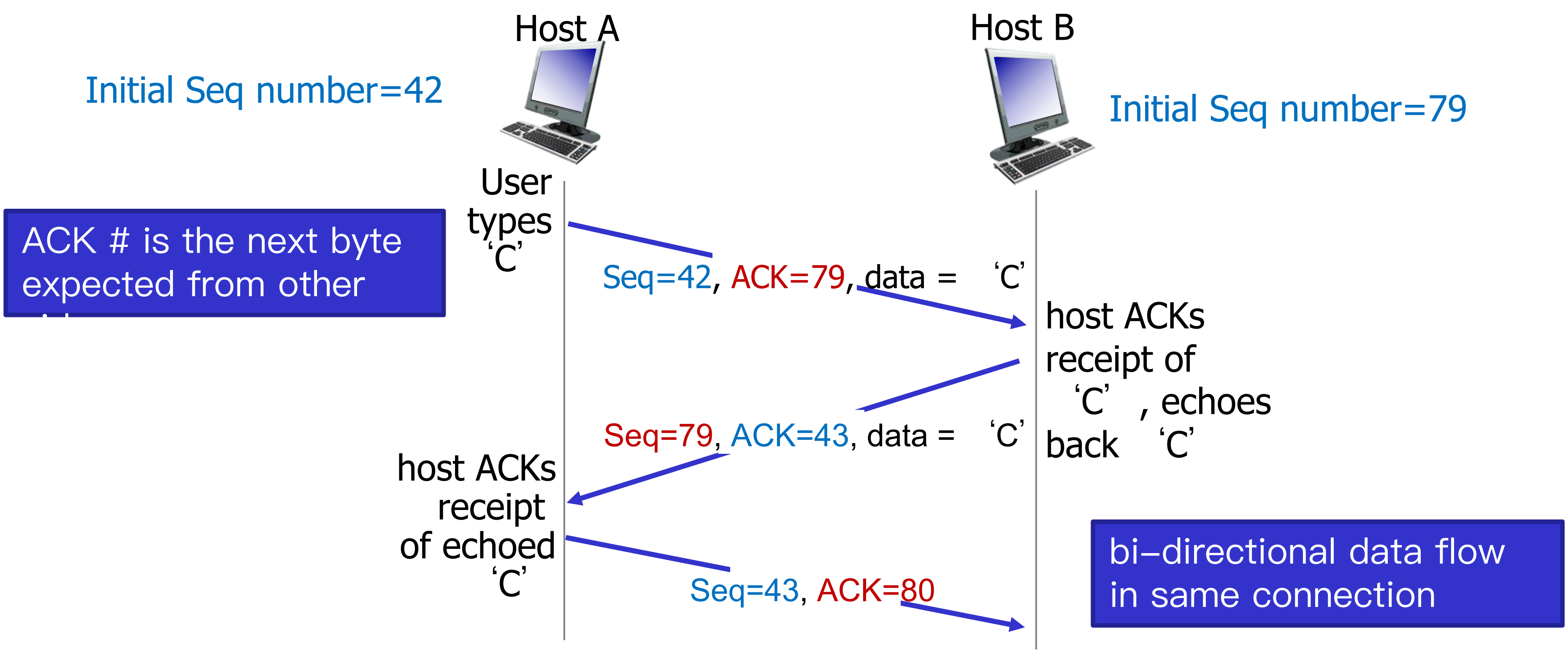
Initial sequence number is randomly chosen

TCP Example


Host A
Host B
SendBase=92
Seq=92, 8 bytes of data
ACK=100
SendBase=100
Seq=100, 20 bytes of data
ACK=120
SendBase=120
Seq=120, 40 bytes of data
ACK=160

Telnet Case Study

- User types a character at host A, and host A sends the character to host B
- Host B sends back a copy of the character
- Host A displays the character on user’s screen



TCP Reliable Data Transfer

- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

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

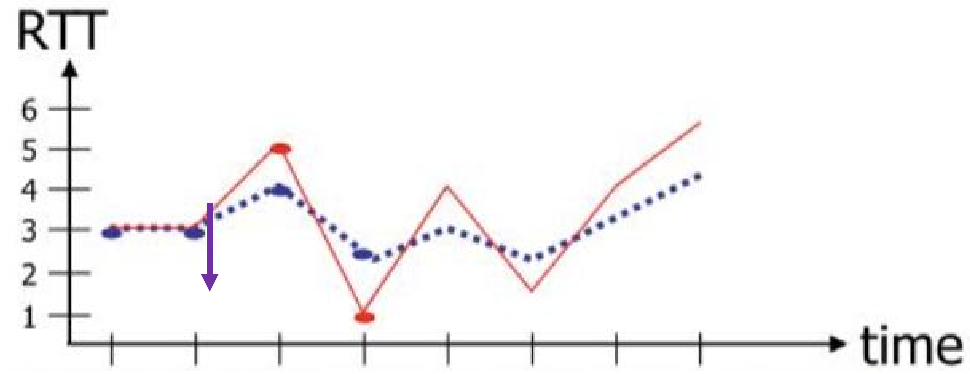
Example:

Suppose α = 0.5
EstimatedRTT = 3

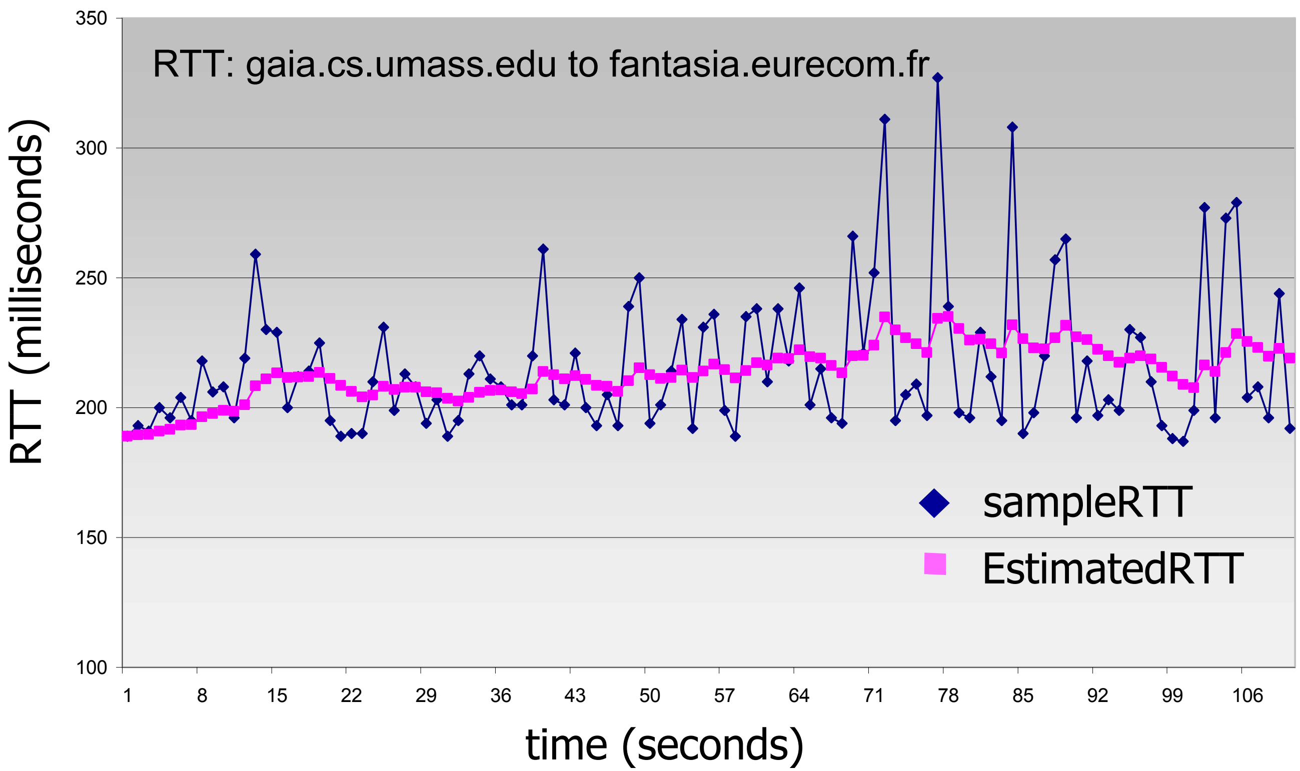
2) EstimatedRTT = .5 * 3 + .5 * 3 = 3

3) EstimatedRTT = .5 * 3 + .5 * 5 = 4

4) EstimatedRTT = .5 * 4 + .5 * 1 = 2.5



TCP round trip time, timeout



TCP round trip time, timeout

Variability of the RTT: how much **`SampleRTT`** deviates from **`EstimatedRTT`**:

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

TCP timeout interval: **`EstimatedRTT`** plus "safety margin"

large variation in **`EstimatedRTT`** -> larger safety margin

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

estimated RTT

"safety margin"

TCP Reliable Data Transfer

- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments: window size, SendBase
  - 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

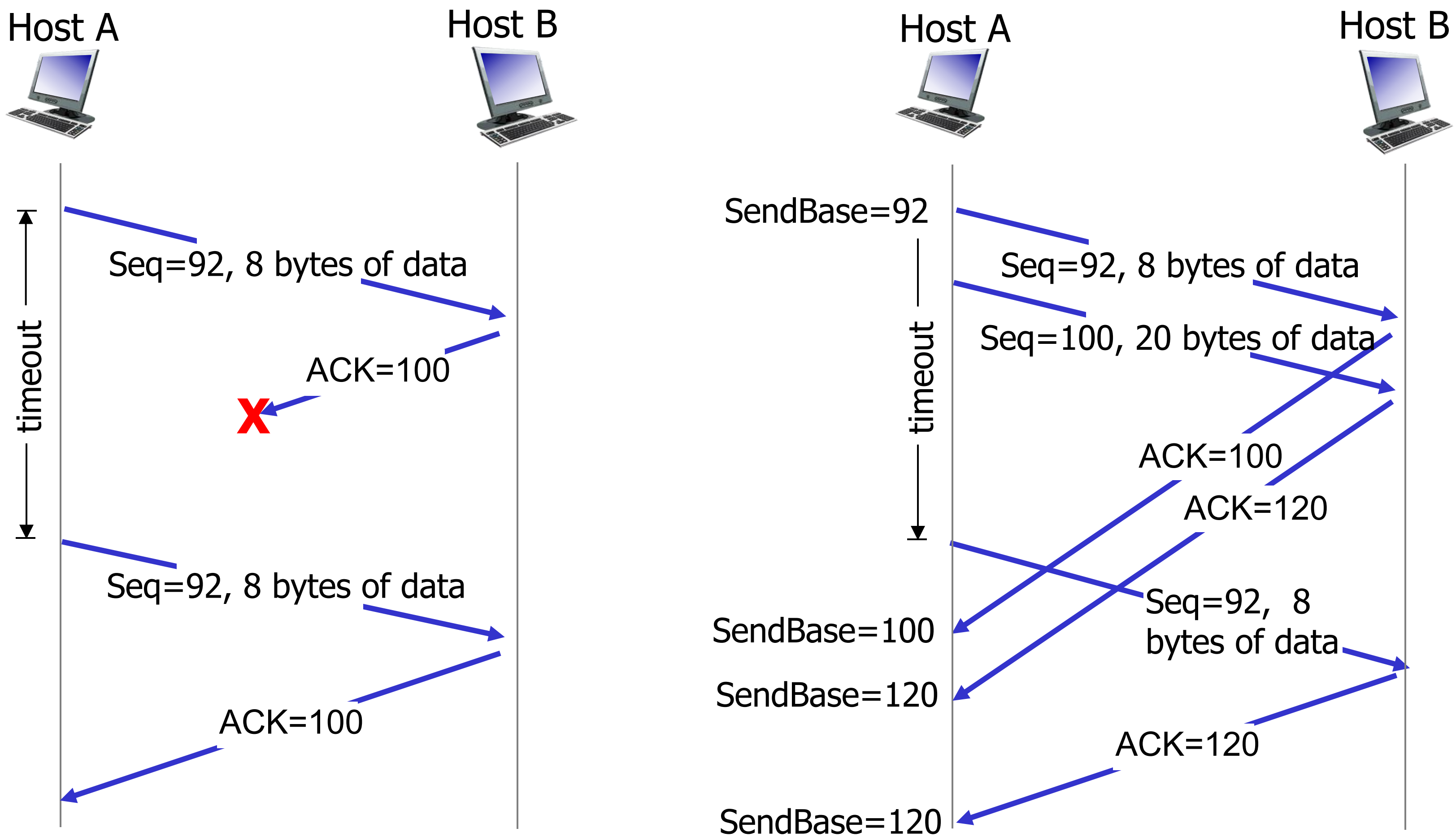
ack rcvd:

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

TCP sender events:

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

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)
            break;

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

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

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

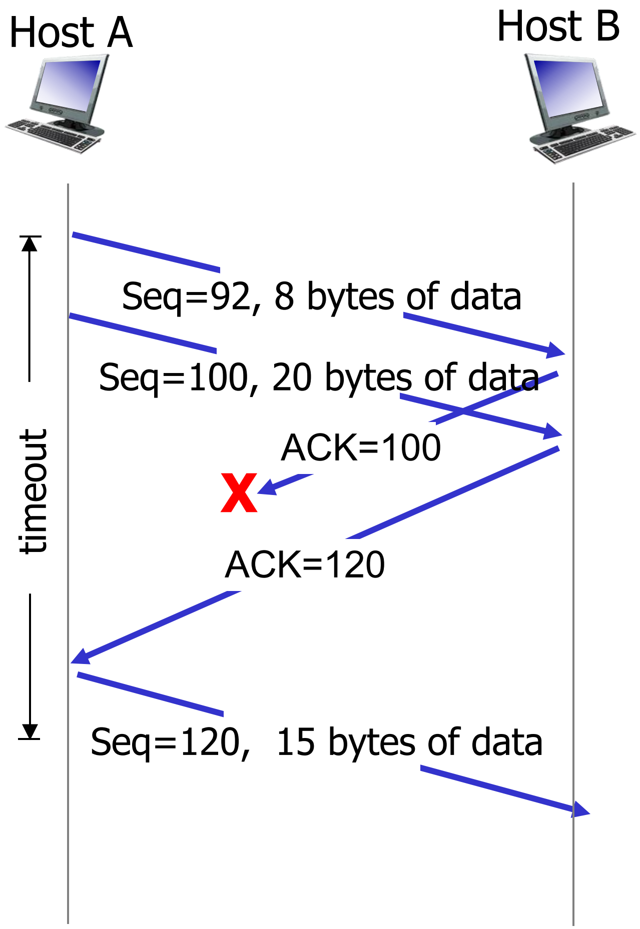
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP receiver [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

Double Timeout Interval

Each time TCP retransmits, it sets the next timeout interval to twice the previous value

- Suppose *TimeoutInterval* associated with the oldest not yet acknowledged segment is 0.75 sec.
- when the timer first expires, TCP will then retransmit this segment and set the new expiration time to 1.5 sec.
- If the timer expires again, TCP will again retransmit this segment, now setting the expiration time to 3.0 sec.

When the timer is started after either of the two other events (that is, data received from application above, and ACK received):

- the *TimeoutInterval* is derived from the most recent values of *EstimatedRTT* and *DevRTT* .

Part of congestion control: being polite

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 duplicate ACKs for same data

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

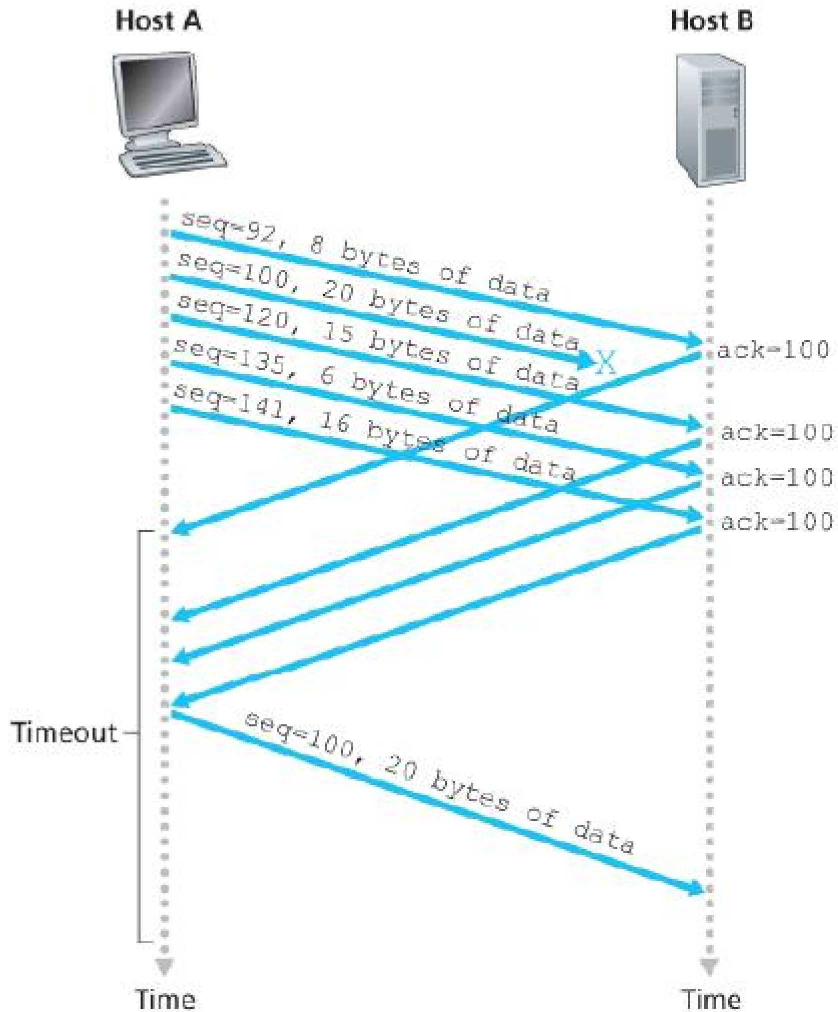
- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

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

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

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

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

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
            SendBase=y
            if (there are currently any not yet
                        acknowledged segments)
               start timer
            }
            else {/* a duplicate ACK for already ACKed
                  segment */
              increment number of duplicate ACKs
                  received for y
              if (number of duplicate ACKS received
                  for y==3)
                  /* TCP fast retransmit */
                  resend segment with sequence number y
              }
           break;
```

TCP fast retransmit

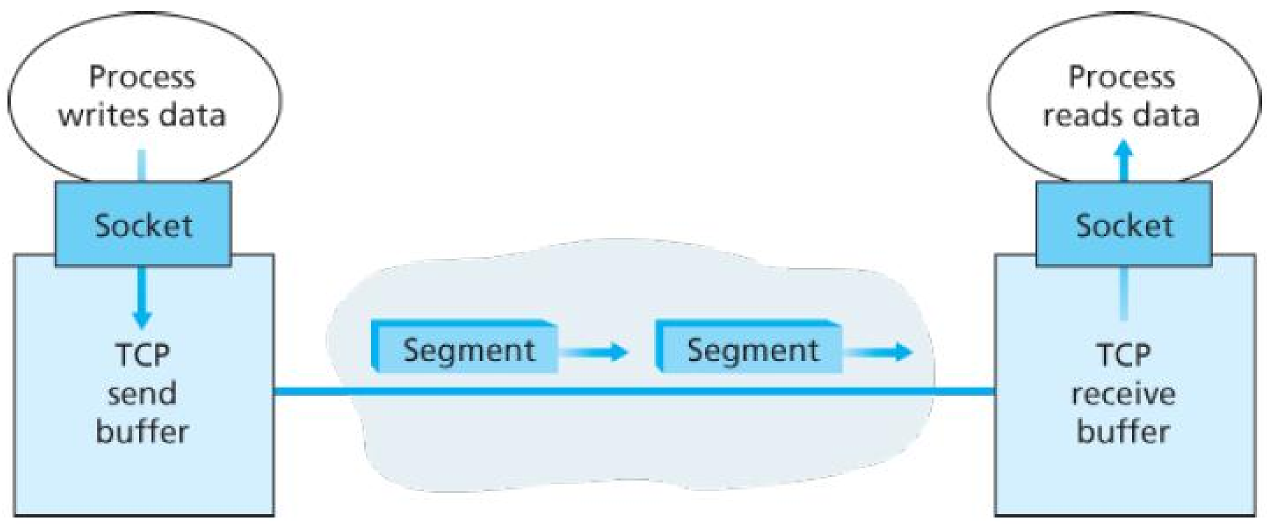


fast retransmit after sender receipt of triple duplicate ACK

TCP Reliable Data Transfer

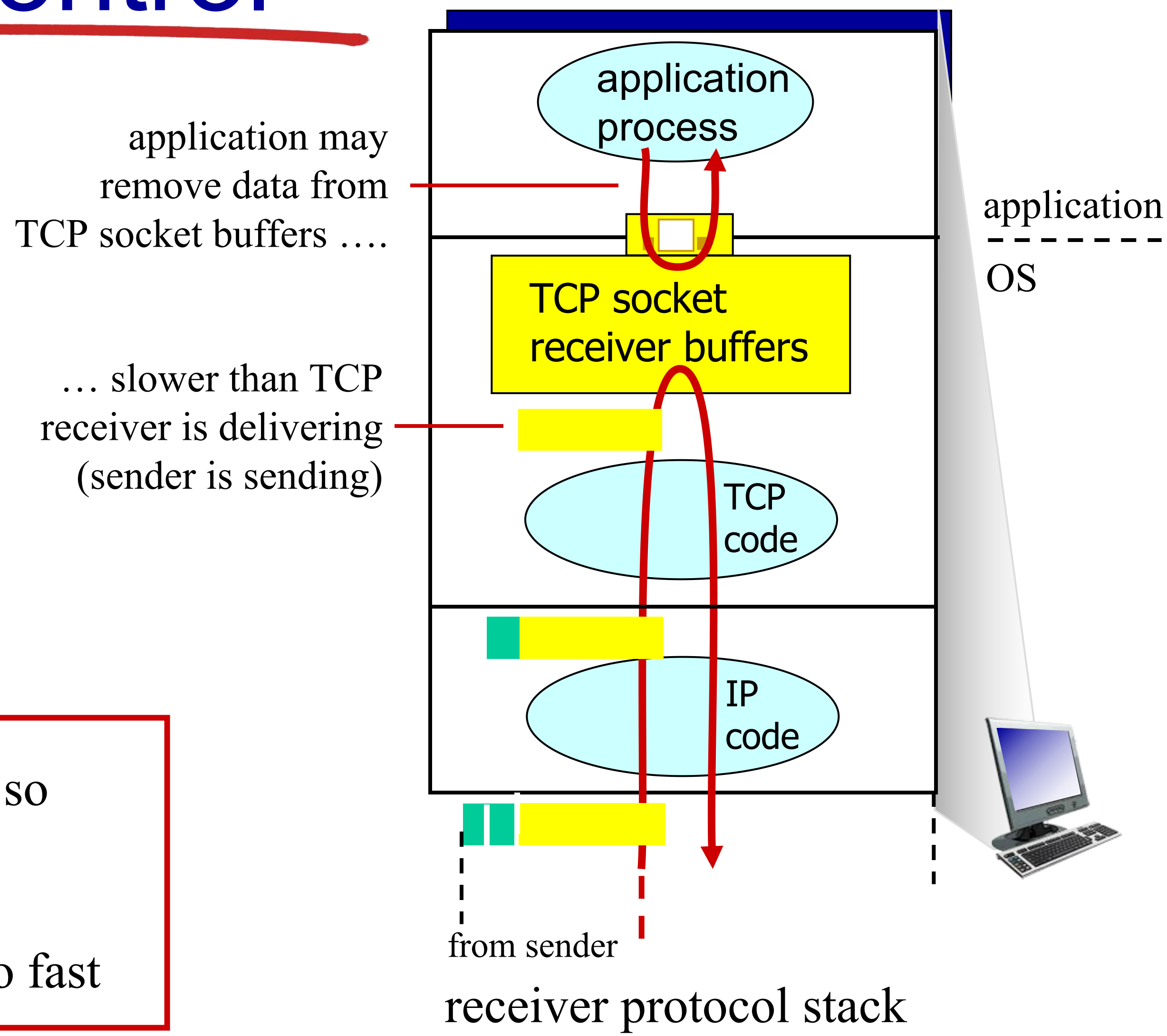
- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

TCP: Overview



- TCP connection
- TCP grab chunks of data from the sender buffer
- TCP receives a segment at the other end, place it in receiver buffer
- application reads the stream from the receive buffer

TCP flow control



flow control

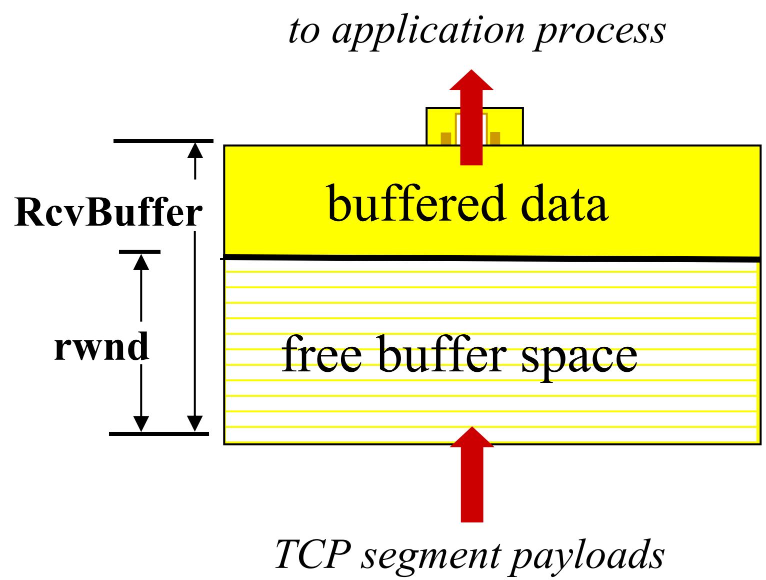
Receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

TCP flow control

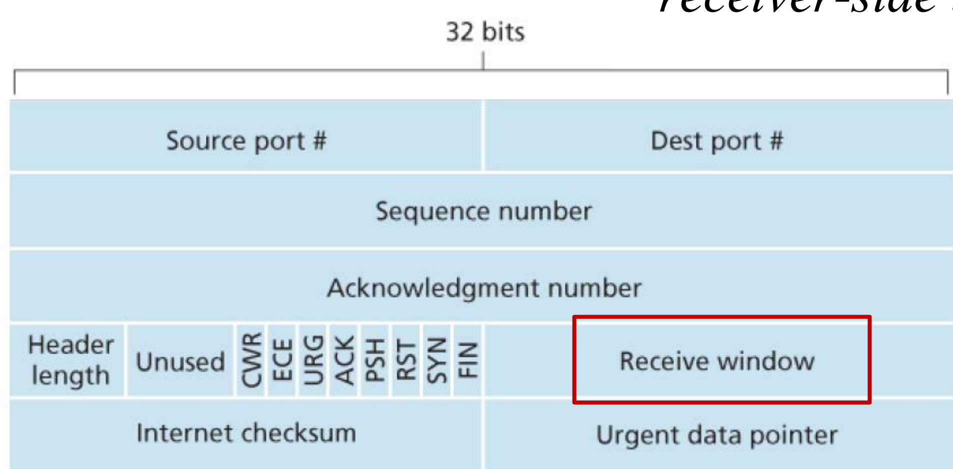
Receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments

- **RcvBuffer** size set via socket options (typical default is 4096 bytes)
- many operating systems autoadjust **RcvBuffer**

rwnd=RcvBuffer−[LastByteRcvd−LastByteRead]



receiver-side buffering

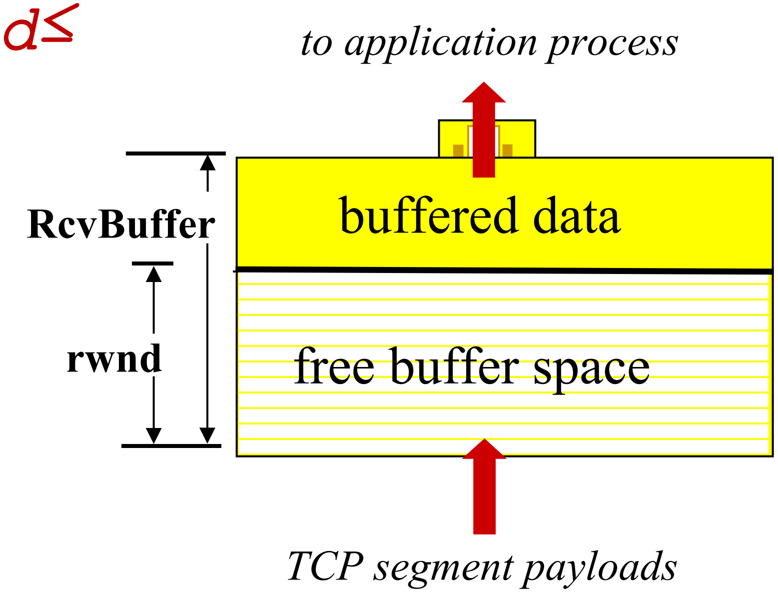


TCP flow control

Sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value

`LastByteSent-LastByteAcked≤ rwnd`

Guarantees receive buffer will not overflow



receiver-side buffering

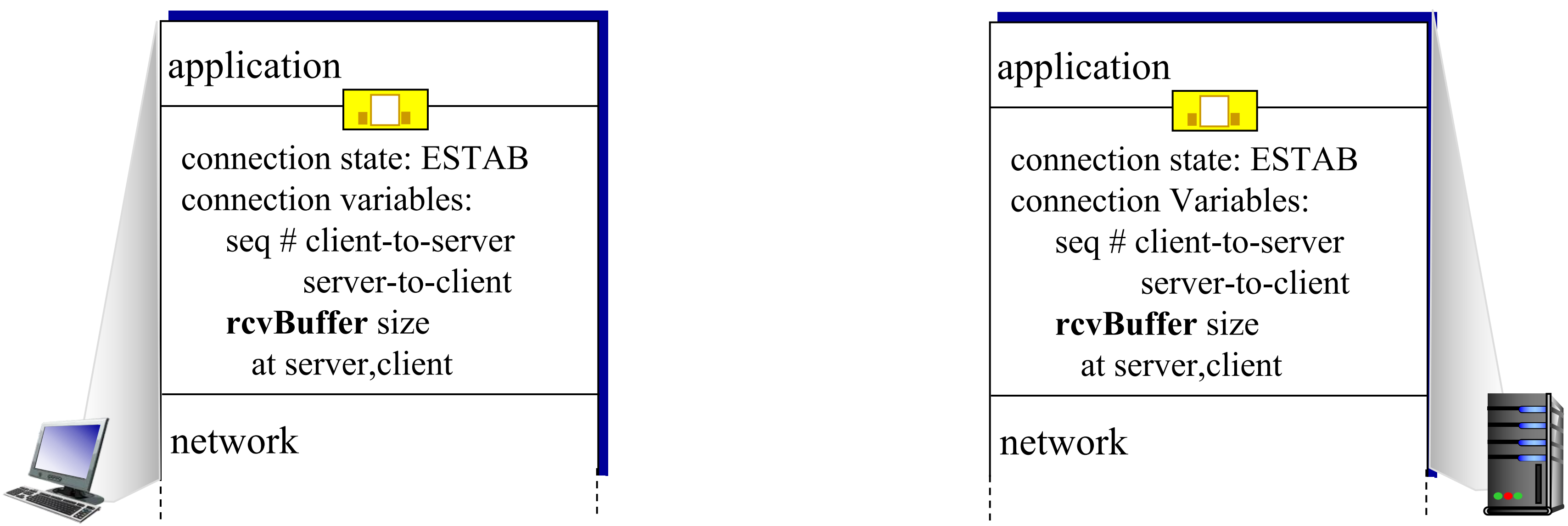
TCP Reliable Data Transfer

- Segment structure
- Round-trip time estimation
- Reliable data transfer
- Flow control
- Control management

Connection Management

before exchanging data, sender/receiver "handshake":

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

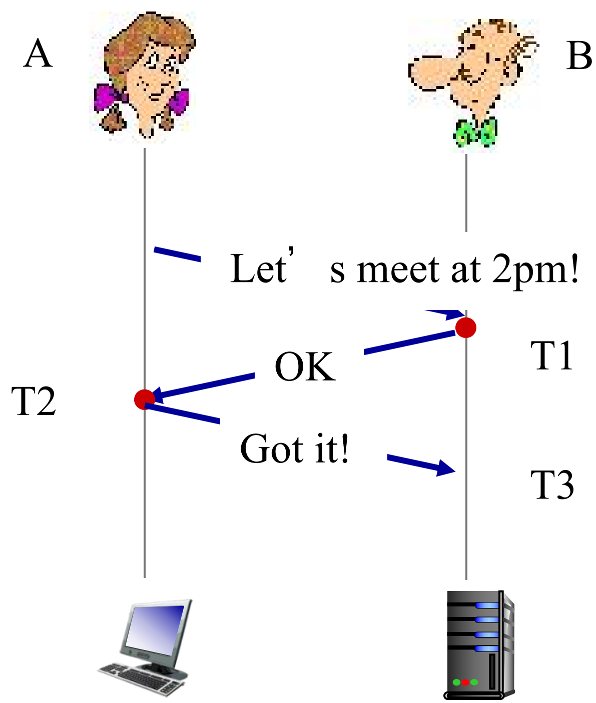



```
clientSocket = socket(AF_INET, SOCK_STREAM);
clientSocket.connect((hostname,port number));
```

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

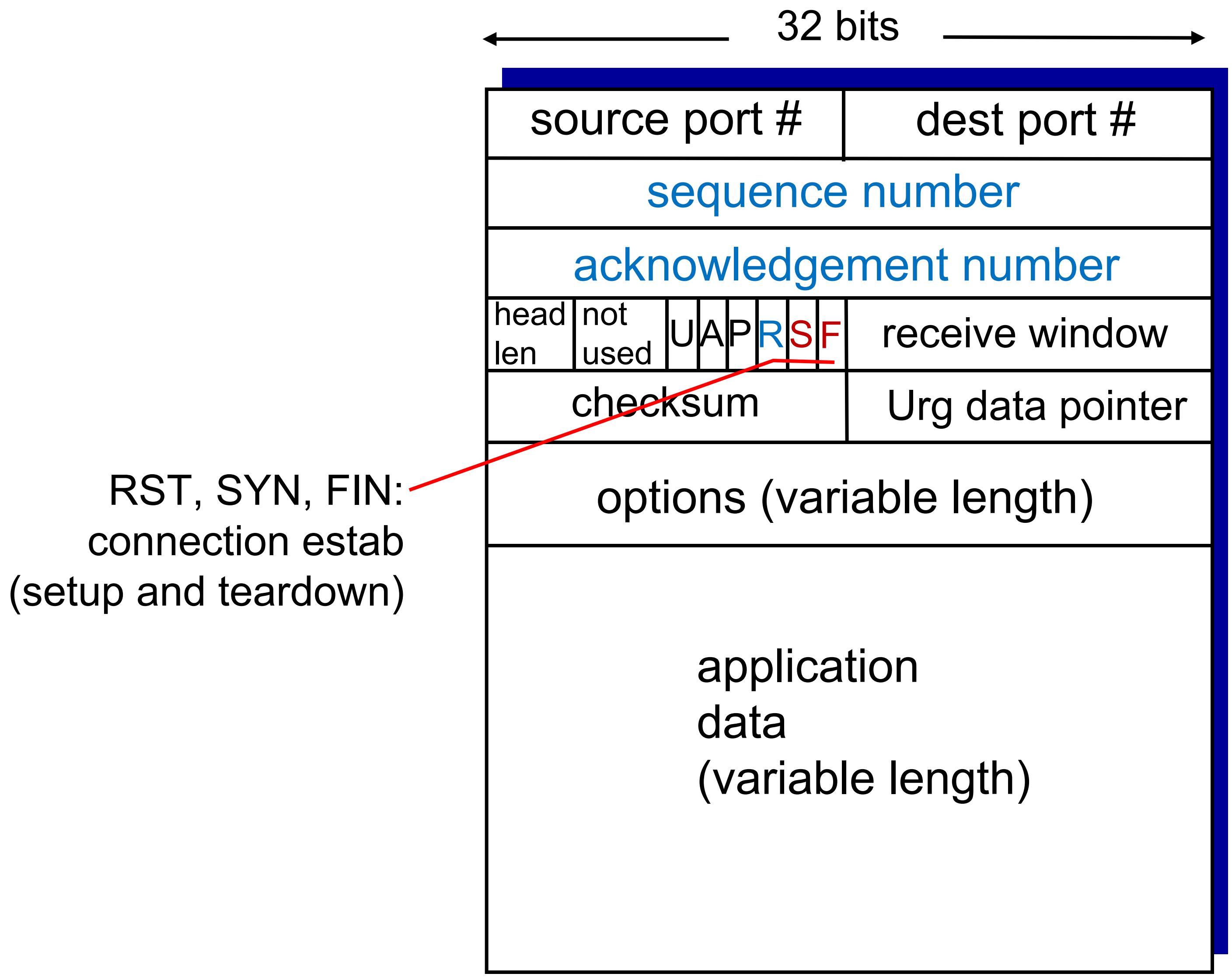
Agreeing to establish a connection

3-way handshake:

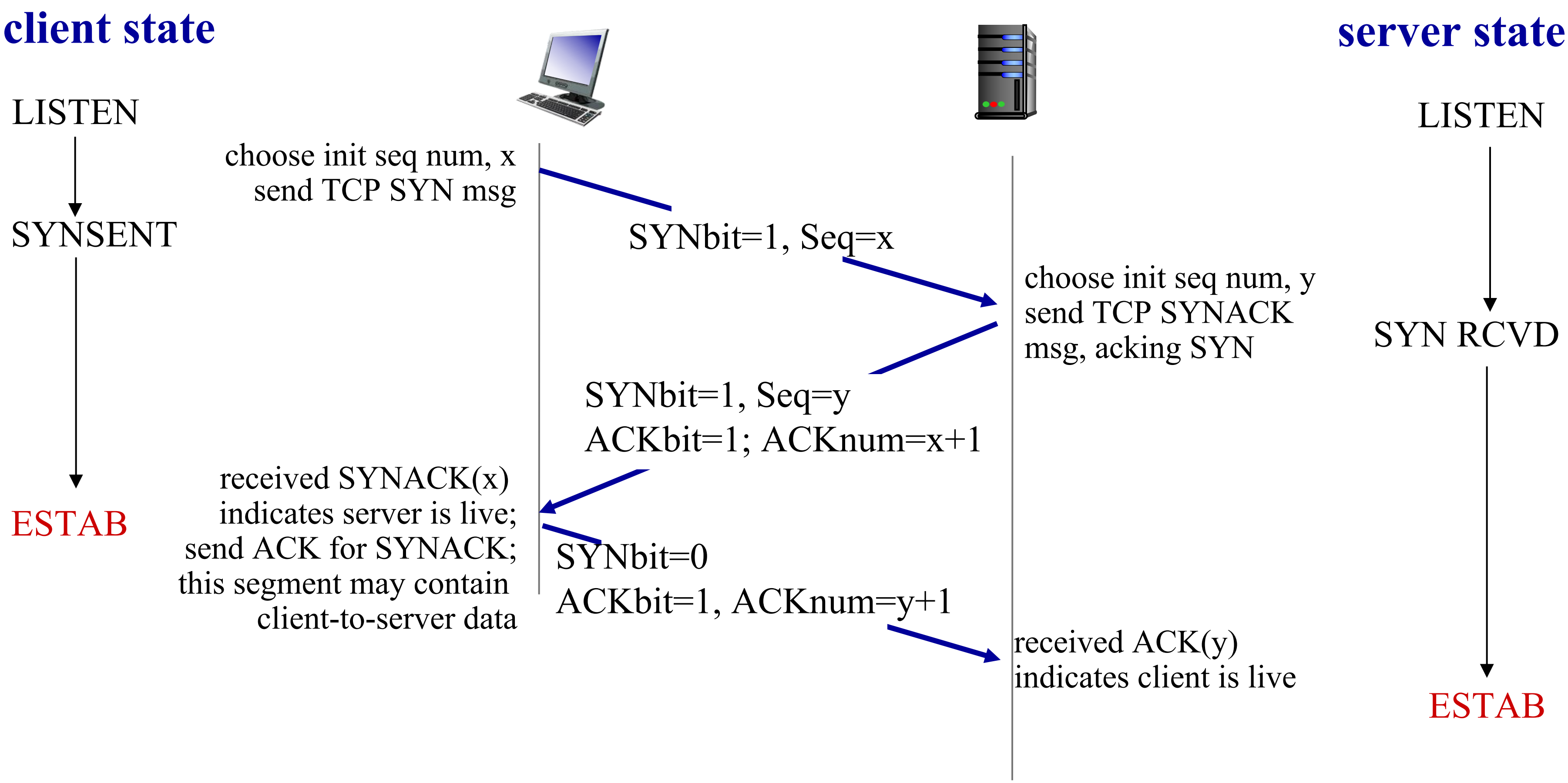


- T1： B knows A's transmitter and B's receiver is OK
- T2: A knows A's transceiver and B's transceiver is OK, B has no more information than T1
- T3: Both A and B know their transceiver are OK, they can start the communication!

TCP segment structure



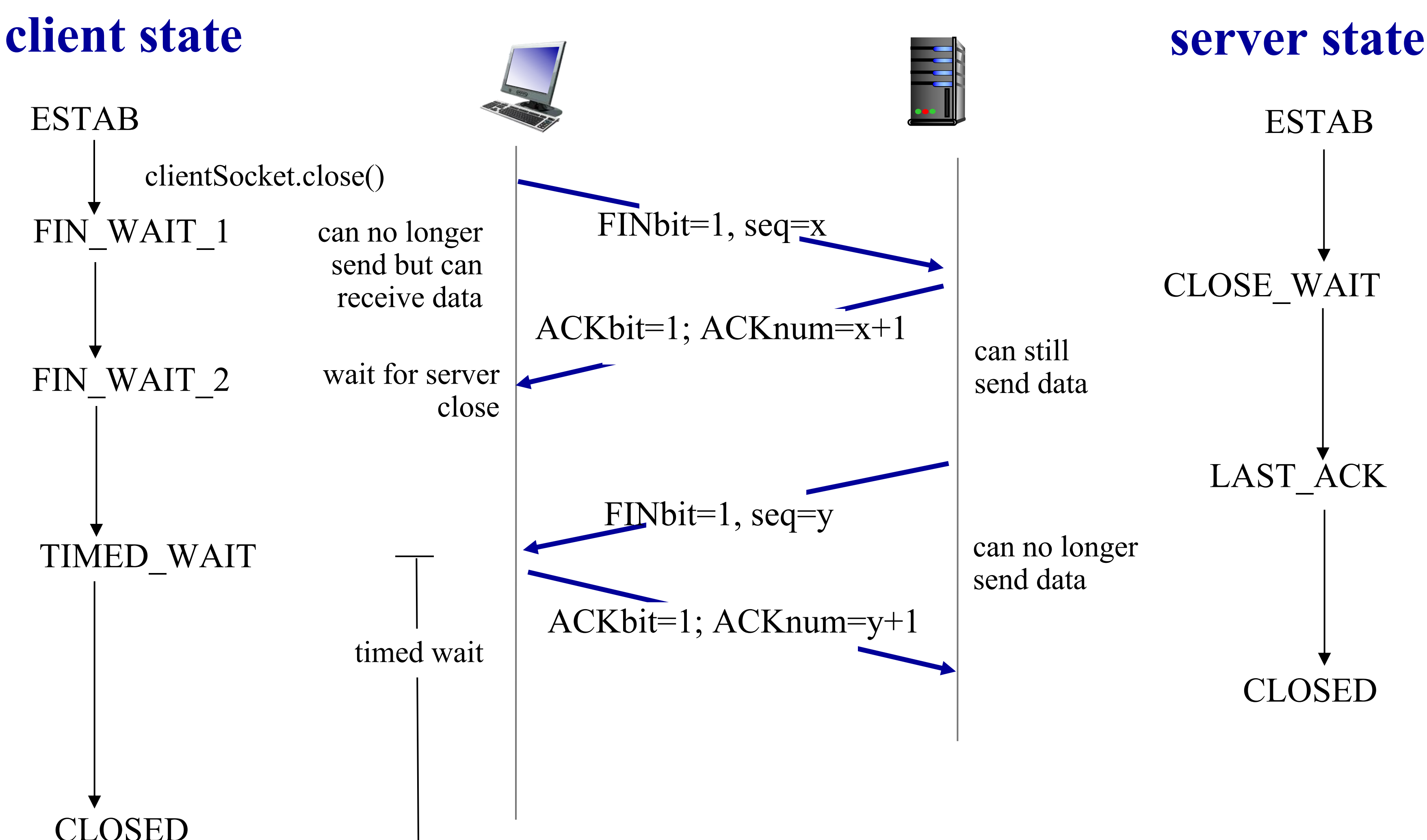
TCP 3-way handshake



Once these three steps have been completed, the client and server hosts can send segments containing data to each other.

- In each of these future segments, SYNbit=0

TCP: closing a connection

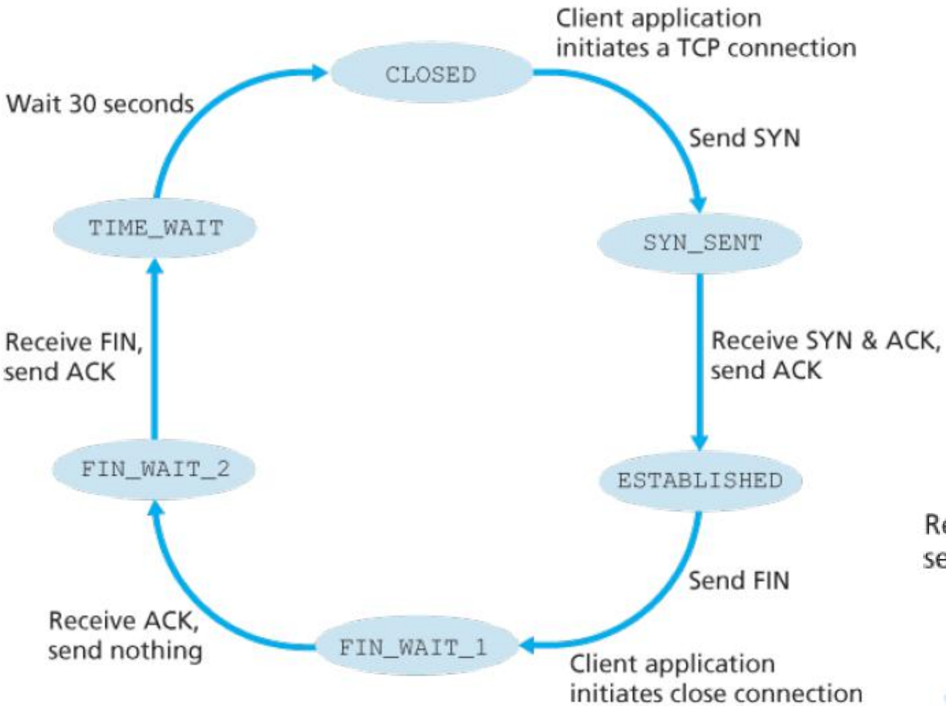


The TIME_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost.

TCP: closing a connection

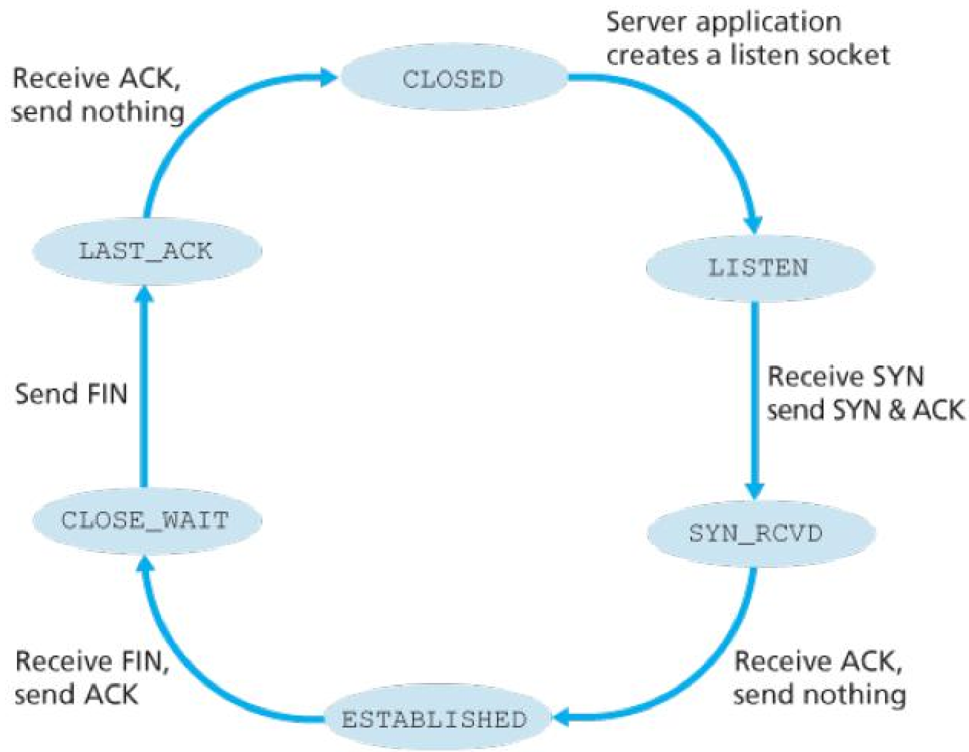
- Four-way handshaking
  - Either of the two processes participating in a TCP connection can end the connection.
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
- Why FIN and ACK can not be sent in one msg as SYNACK in connection establishment?
  - The other side may still have packets need to be sent. It can not send FIN until the transmission is finished.

TCP States



Client Side

Server Side



Reset Segment

When a host receives a TCP segment whose port numbers or source IP address do not match with any of the ongoing sockets.

- Then the host will send a special reset segment to the source. RST flag bit is set to 1.
- "I don't have a socket for that segment. Please do not resend the segment."

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 TCP congestion control