Chapter 3: Transport Layer

- Principles of Transport Layer
- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol)

Transport Layer

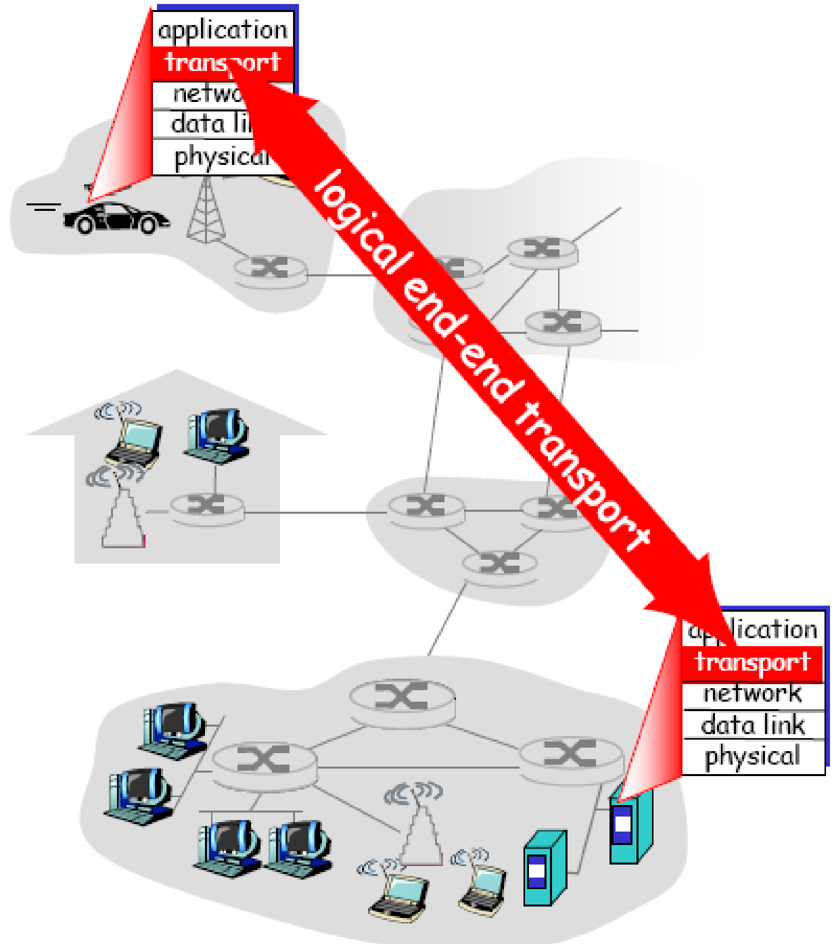
- Understand principles behind transport layer services
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Transport layer protocol
  - TCP
  - UDP

Chap 3

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

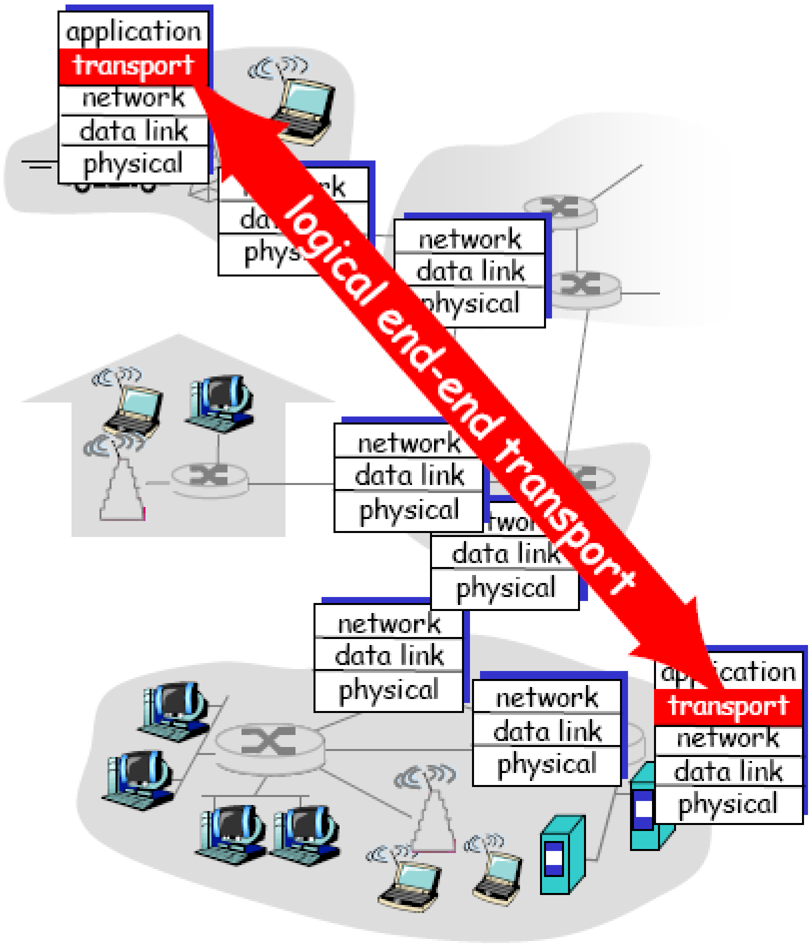
Transport Services and Protocols

- Provide *logical communication* between app processes
- Transport protocols in end system
  - send side: breaks app msg into segments, passes to IP 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
  - IP (Internet Protocol)
- Transport layer
  - logical communication between processes
  - relies on, enhances, network layer services
  - TCP and UDP



Chap 3

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

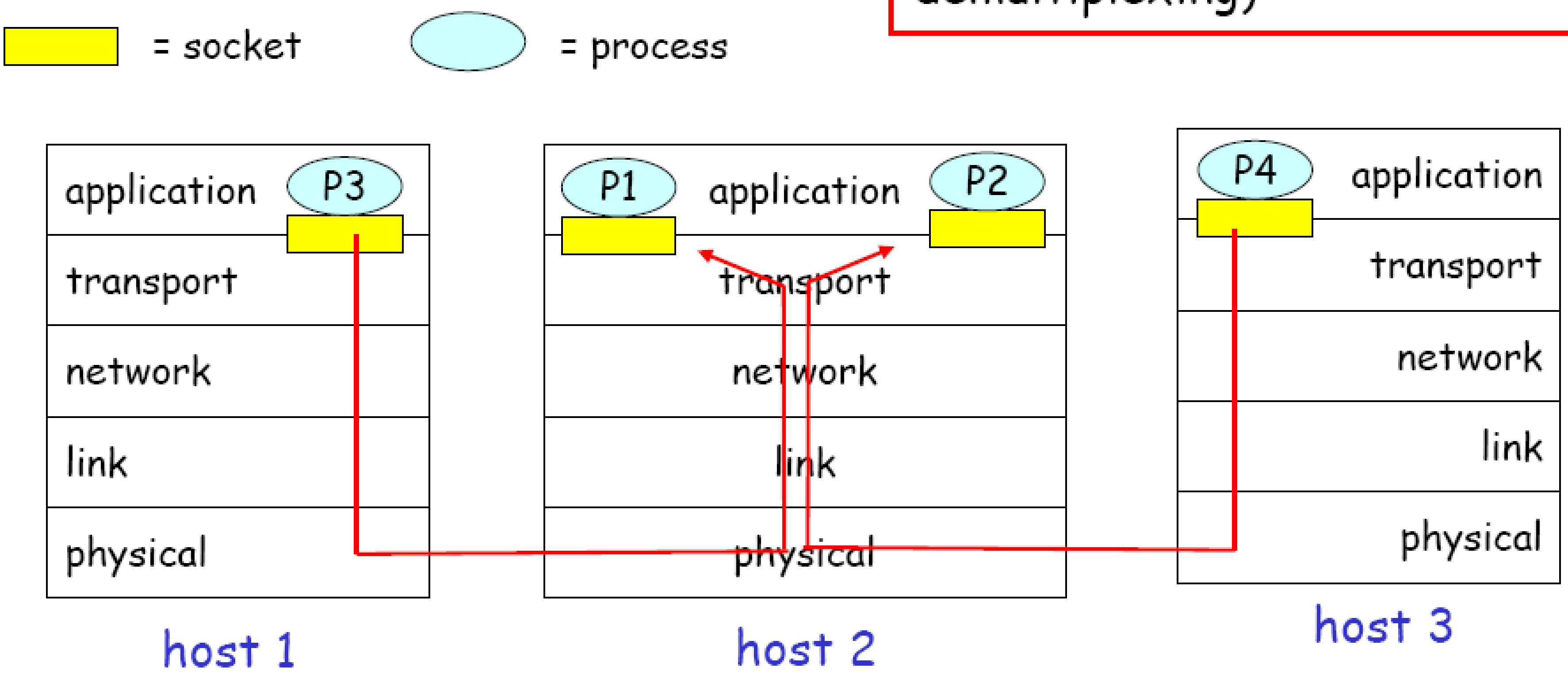
Multiplexing and Demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

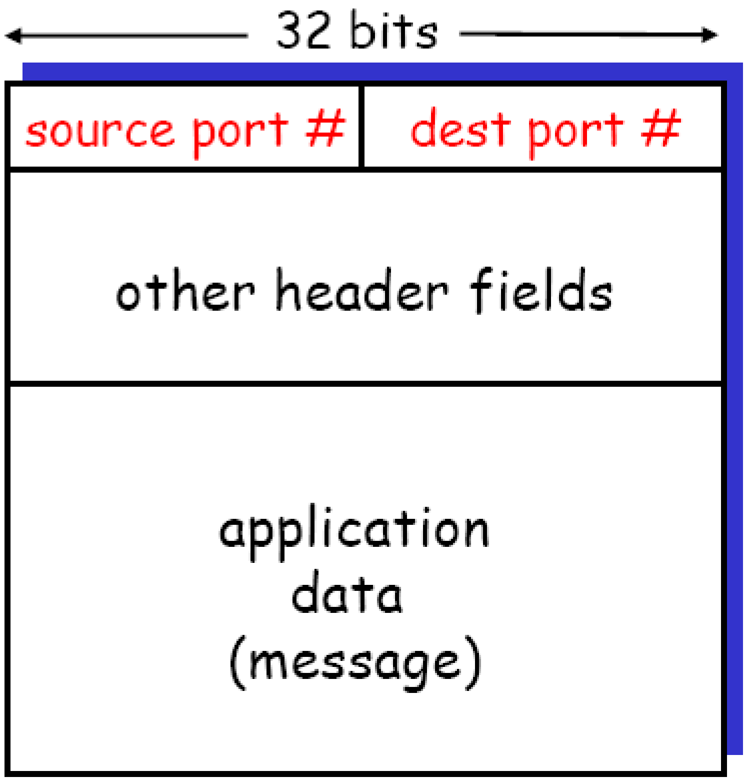
Multiplexing at send host:

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



Multiplexing and Demultiplexing

- host receives IP datagrams
  - each datagram has source IP address, destination IP address, and
  - each datagram has source, destination port number
- host uses (IP addr, port number) to direct segment to appropriate socket



Connectionless Demultiplexing

- UDP socket identified by two-tuple:

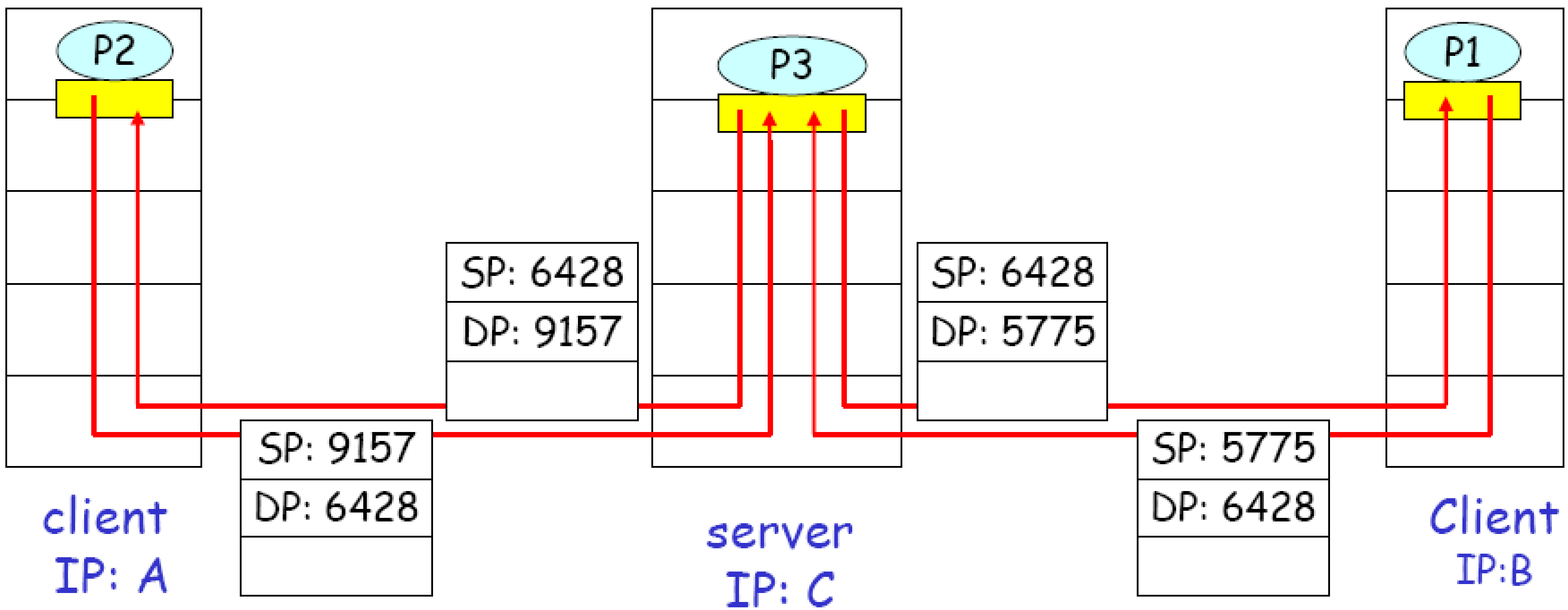
  (local_IP, local_Port, *, *)

- UDP datagram delivery: (SrcIP, DstIP, SrcPort, DstPort)
  - checks destination (IP, port number) in segment:
    (DstIP==local_IP) && (DstPort==local_Port)
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP and/or source port numbers directed to a UDP socket

Connectionless Demultiplexing

- UDP socket

DatagramSocket svrSocket = new DatagramSocket(6428);

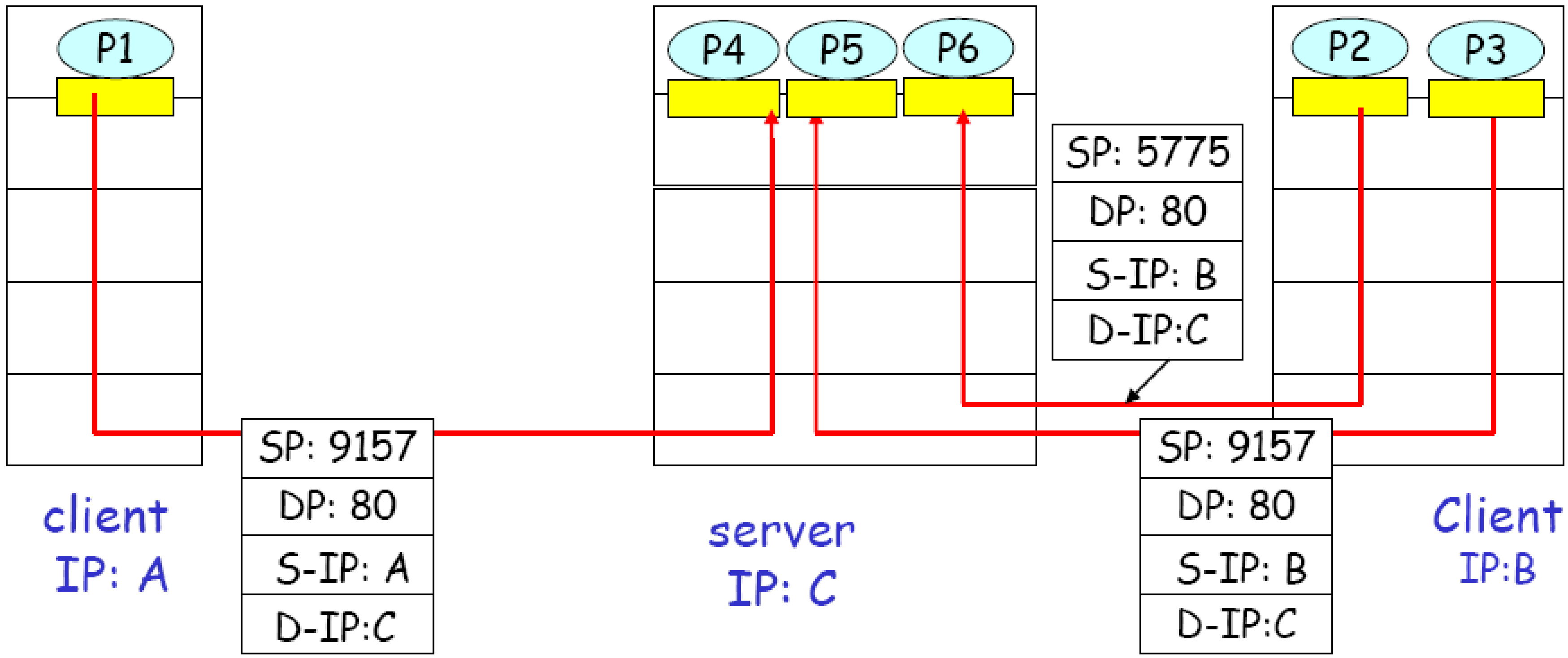


Connection-oriented Demultiplexing

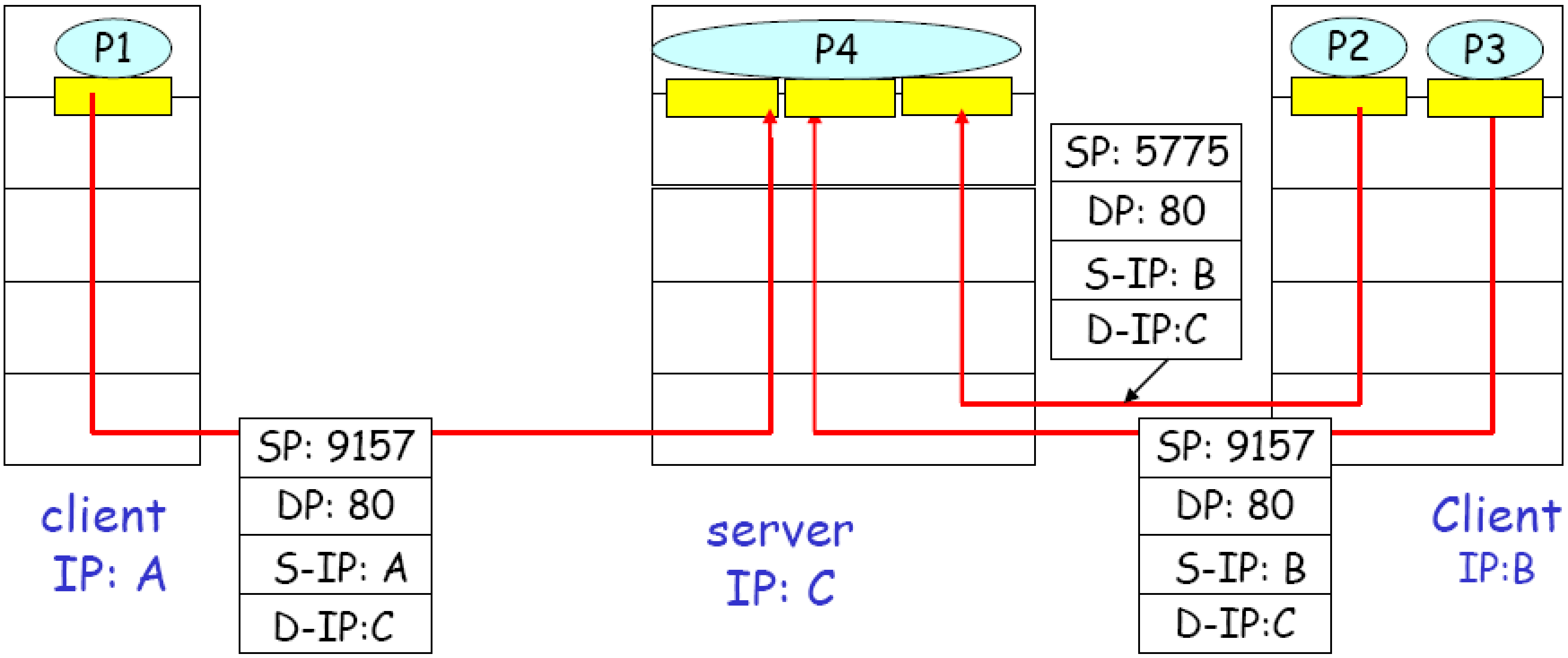
- TCP socket identified by 4-tuple:
  - (local_IP, local_Port, remote_IP, remote_Port)
- TCP segment delivery: (SrcIP, DstIP, SrcPort, DstPort)
  - checks source and destination (IP, port number) in segment: (DstIP==local_IP) && (DstPort==local_Port) && (SrcIP==remote_IP) && (SrcPort==remote_Port)
- Server support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - Listening socket: (local_IP, local_Port, *, *)
  - Connected socket: (local_IP, local_Port, remote_IP, remote_Port)

Connection-oriented Demultiplexing

- TCP socket



Connection-oriented Demux: Threaded Web Server



Chap 3

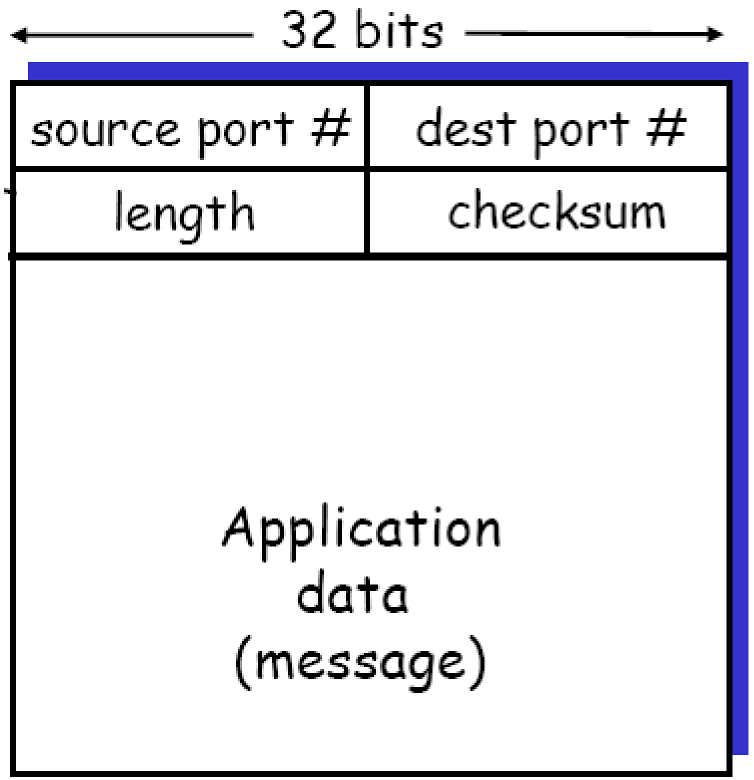
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

UDP: User Datagram Protocol

- UDP [RFC 768]
  - "best effort" service
  - UDP segments may be lost and delivered out of order to appl
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- UDP has smaller protocol overhead than TCP

UDP

- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP
  - need to implement reliability at application layer



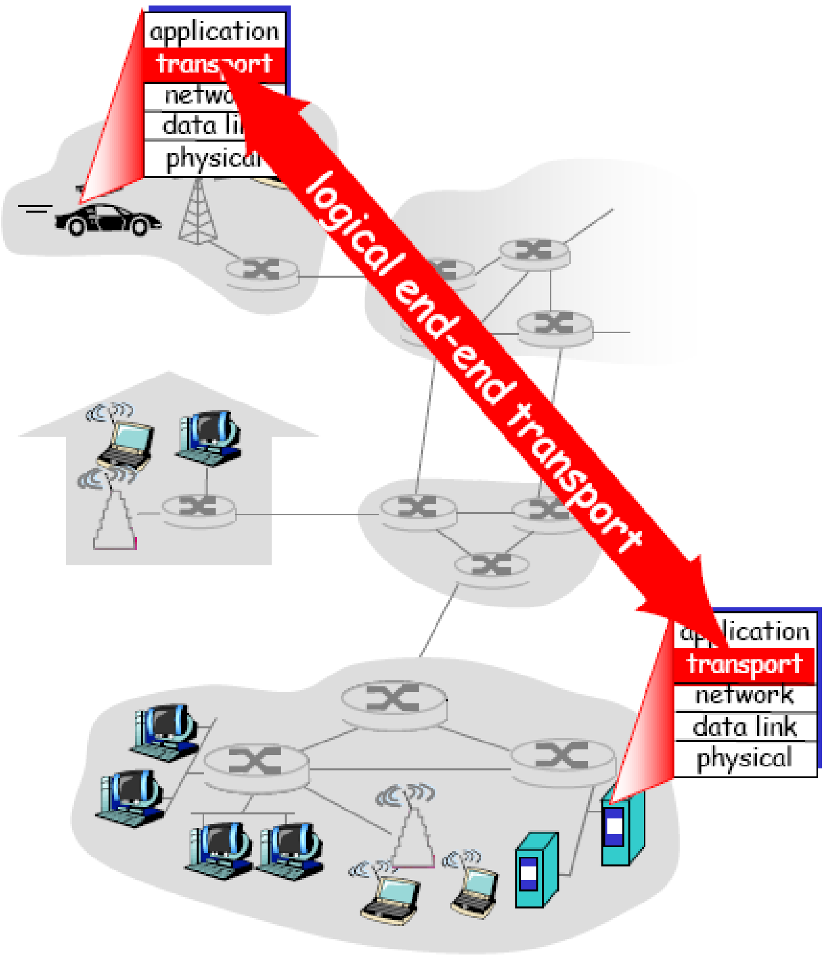
UDP datagram format

Chap 3

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

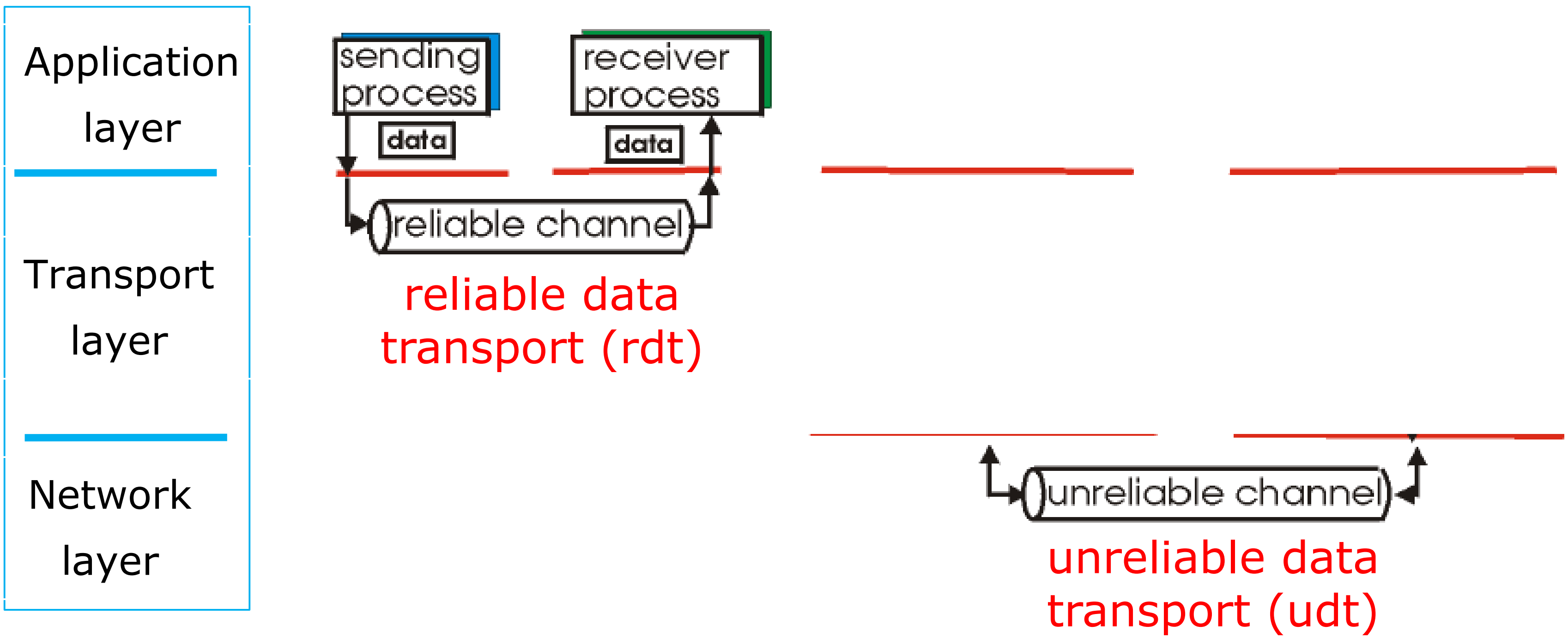
Principles of Reliable data transfer

- Applications needs reliable channel but network provides unreliable channel
- Network layer
  - unreliable communication between hosts
- Transport layer
  - end-to-end reliable communication between processes



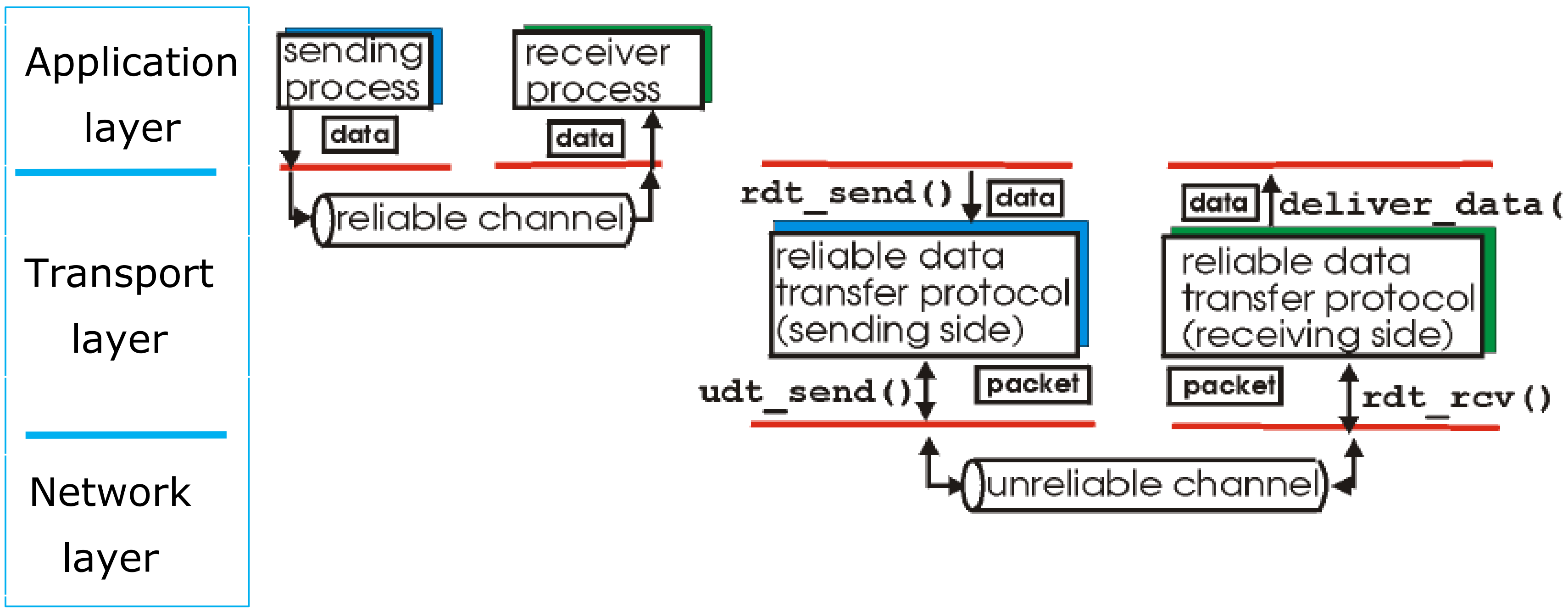
Principles of Reliable data transfer

- Reliability needs in transport layer

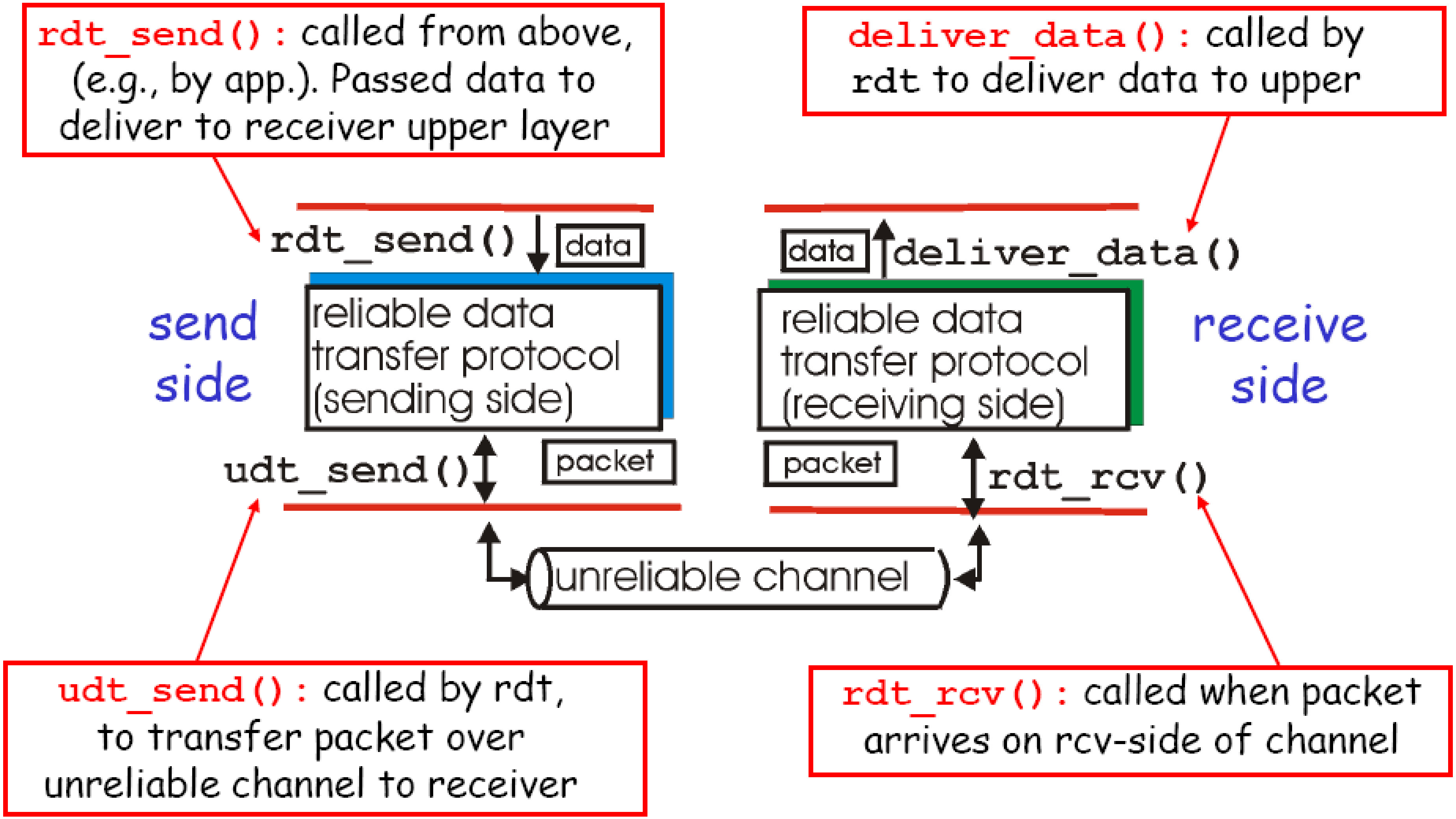


Principles of Reliable data transfer

- Reliability in transport layer

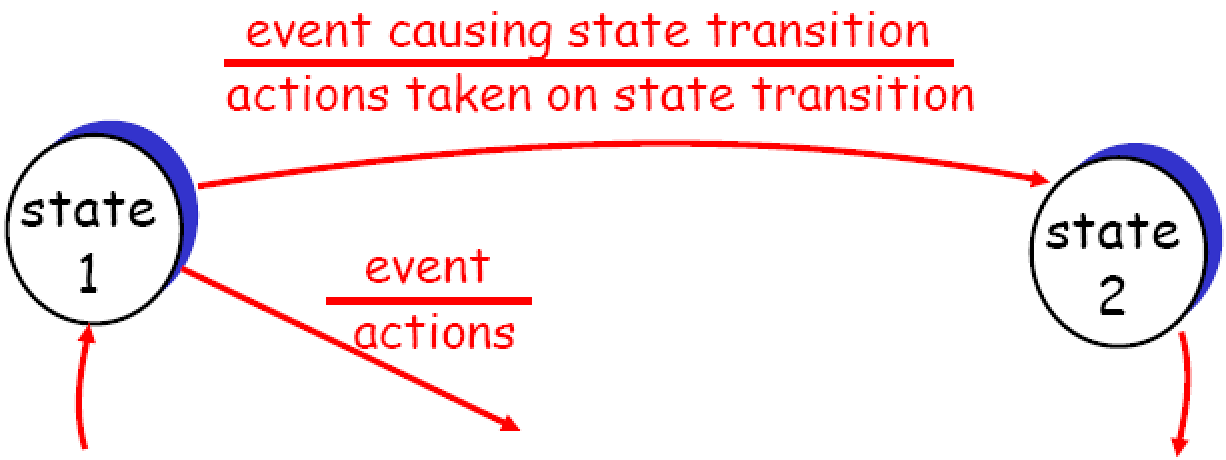


Reliable Data Transfer: getting started



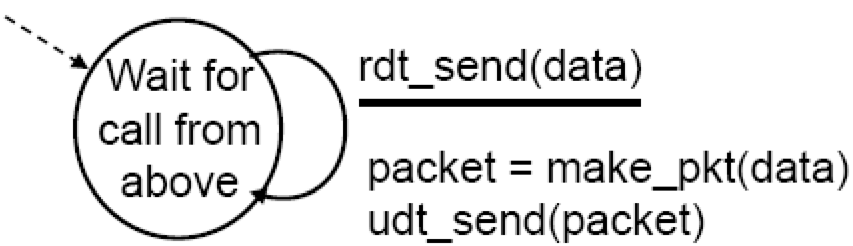
Reliable Data Transfer: getting started

- incrementally develop sender, receiver sides of reliable data transfer protocol
- use finite state machines (FSM) to specify sender, receiver

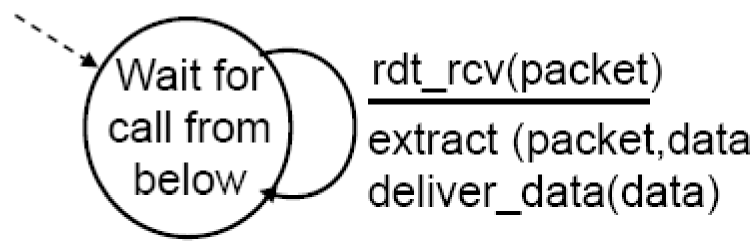


Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel

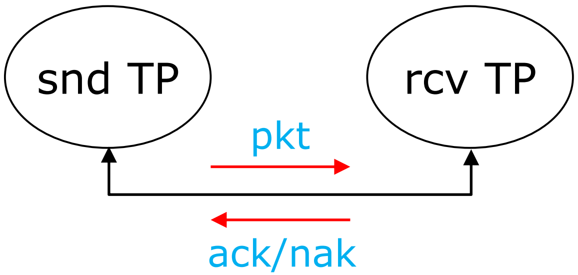


sender



receiver

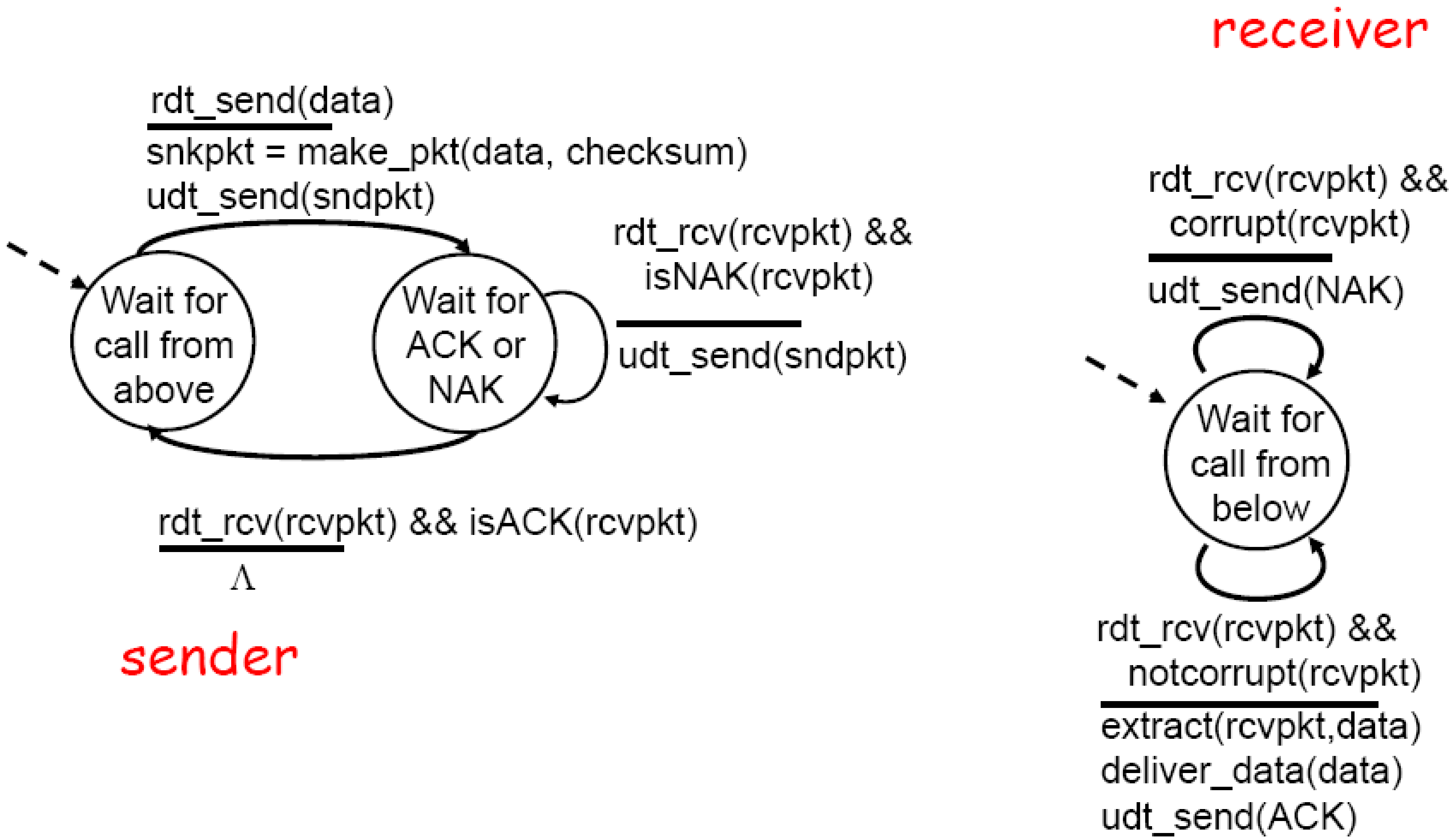
Rdt2.0: channel with bit errors, but no packet loss



- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *Q*: 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`**
  - error detection
  - receiver feedback: rcv sends control msgs (ACK,NAK) to sender

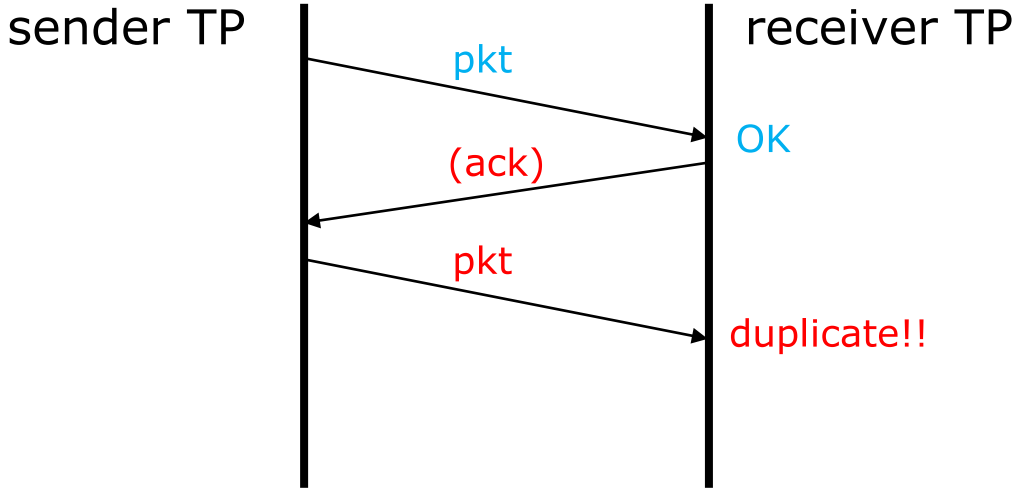
Rdt2.0: channel with bit errors, but no packet loss

- RDT2.0 FSM: Sender and receiver



Rdt2.0: problems

- What happens if ACK/NAK corrupted?
  - needs checksum in ACK or NAK packet
  - if corrupted, sender doesn’t know what happened at receiver
  - can’t just retransmit: possible duplicate

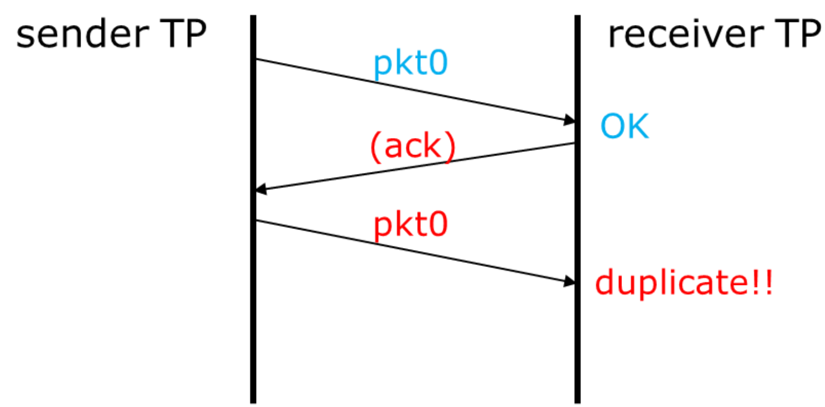


Rdt2.0: problems

- **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 response
  - Needs 1-bit sequence number

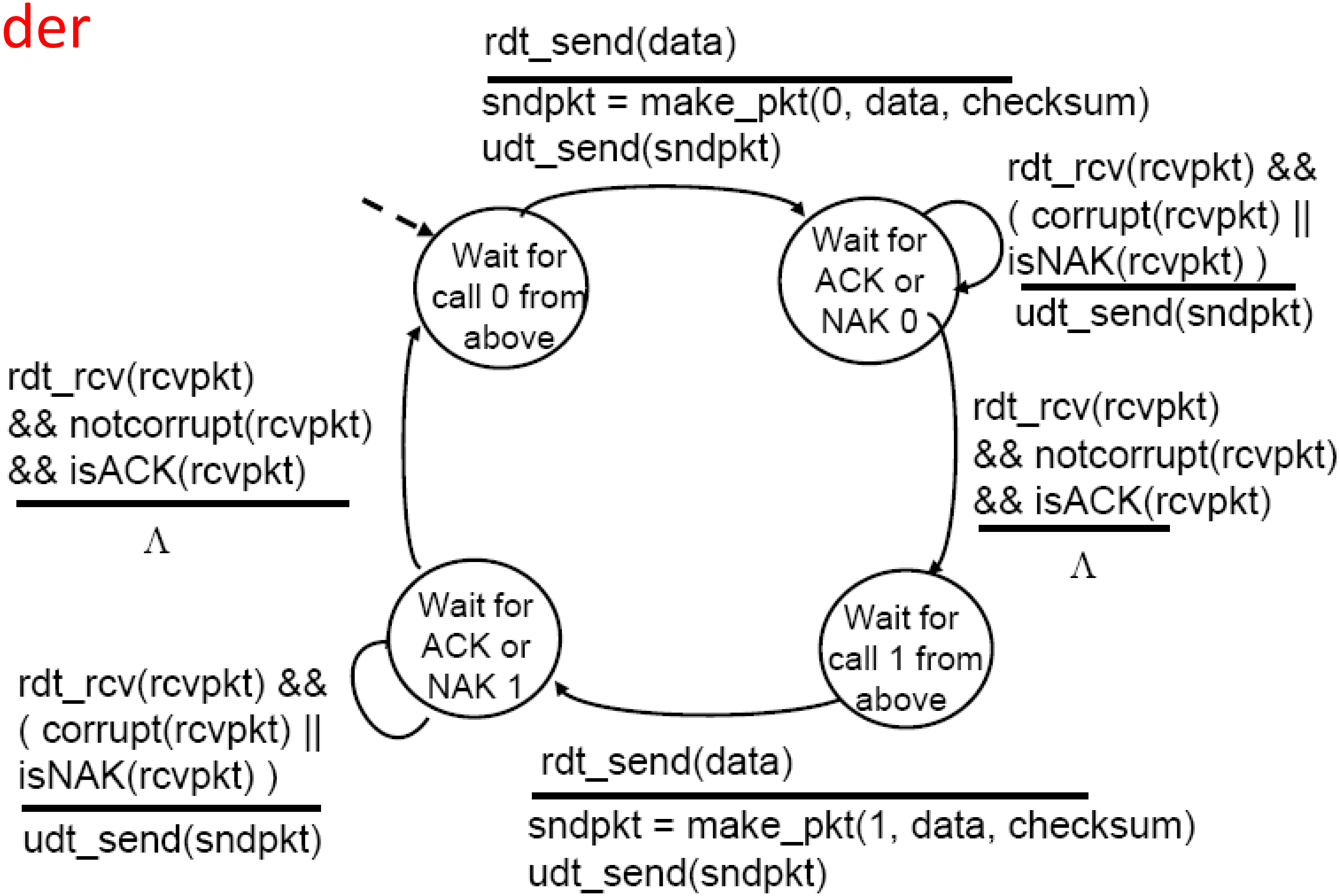
sequence number:

0	1	0	1	…



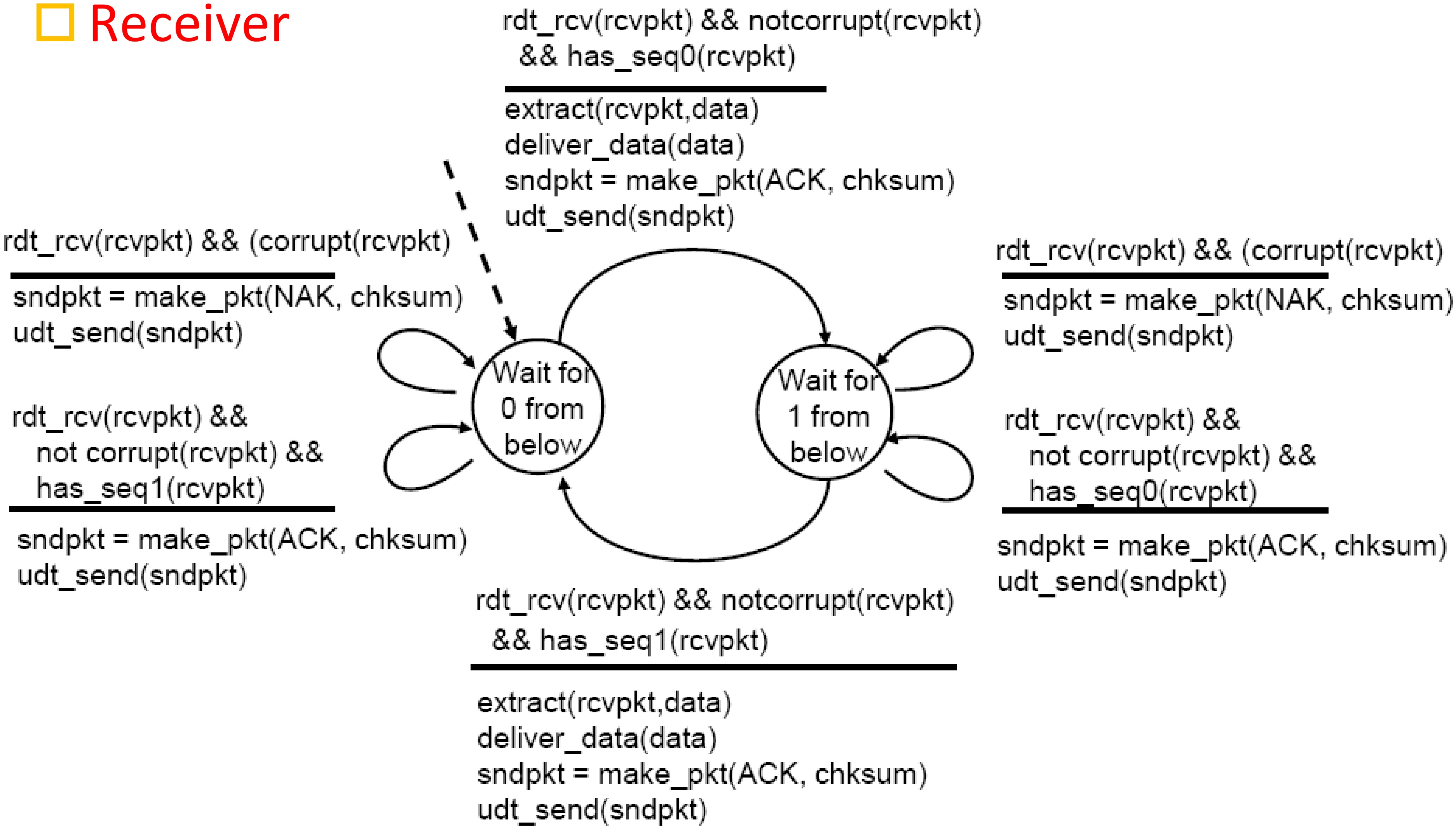
Rdt2.1: handles garbled ACK/NAKs

- Sender



Rdt2.1: handles garbled ACK/NAKs

- Receiver

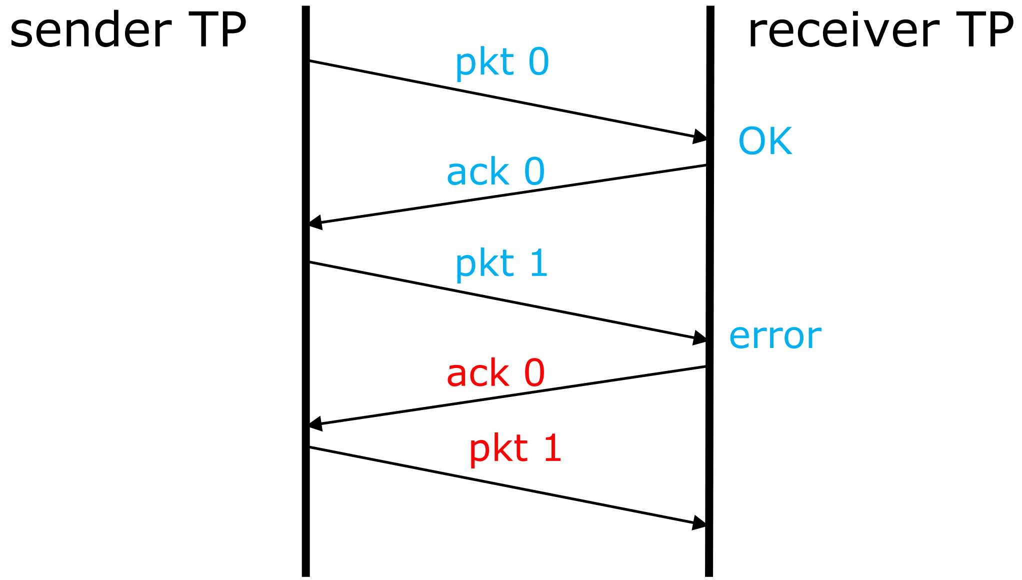


Rdt2.1: discussion

- Sender:
  - seq # added to pkt: needs two seq. #'s (0,1) in stop-and-wait
  - must check if received ACK/NAK packet corrupted
  - needs 4 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

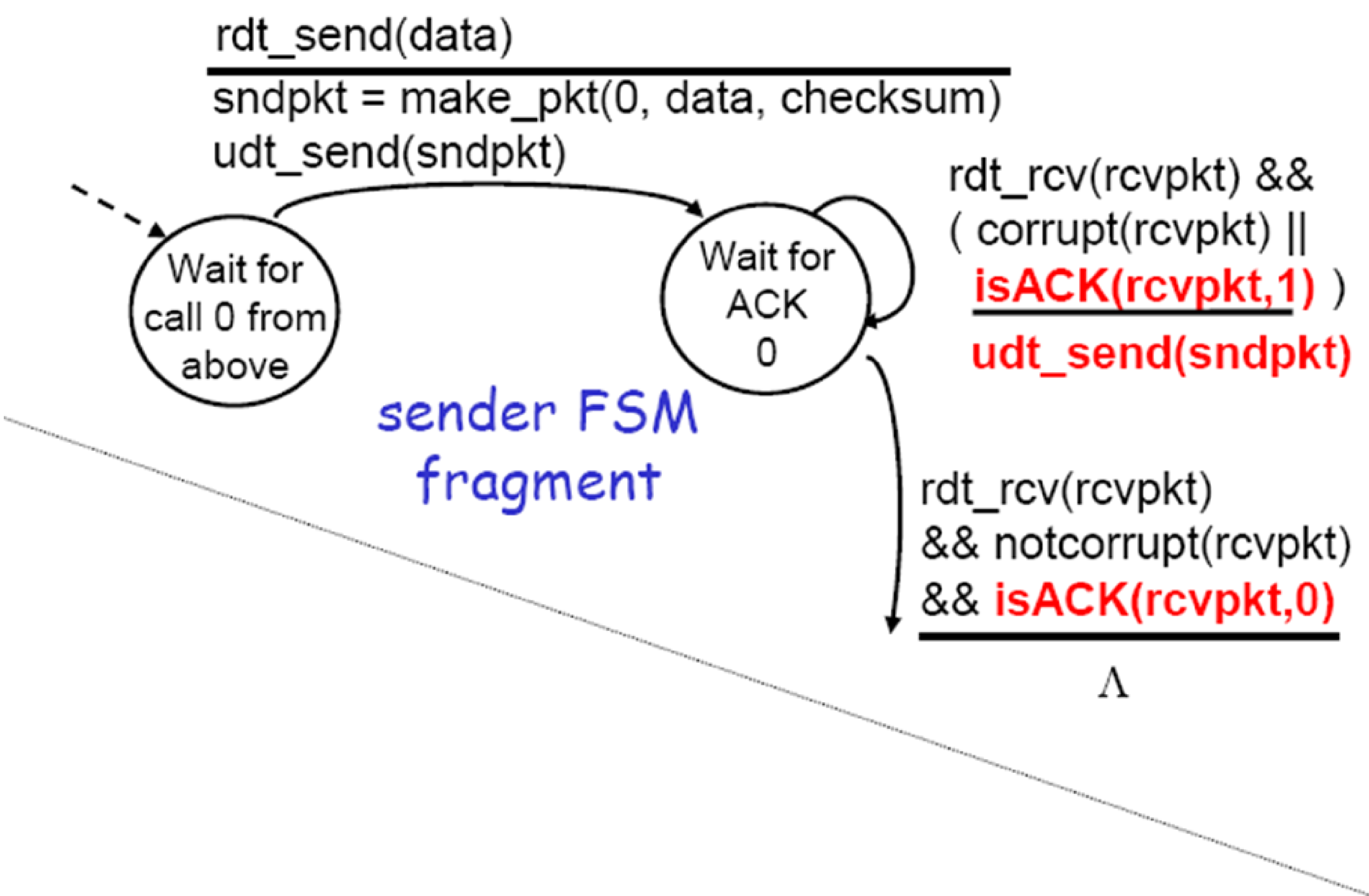
Rdt2.2: a NAK-free protocol

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



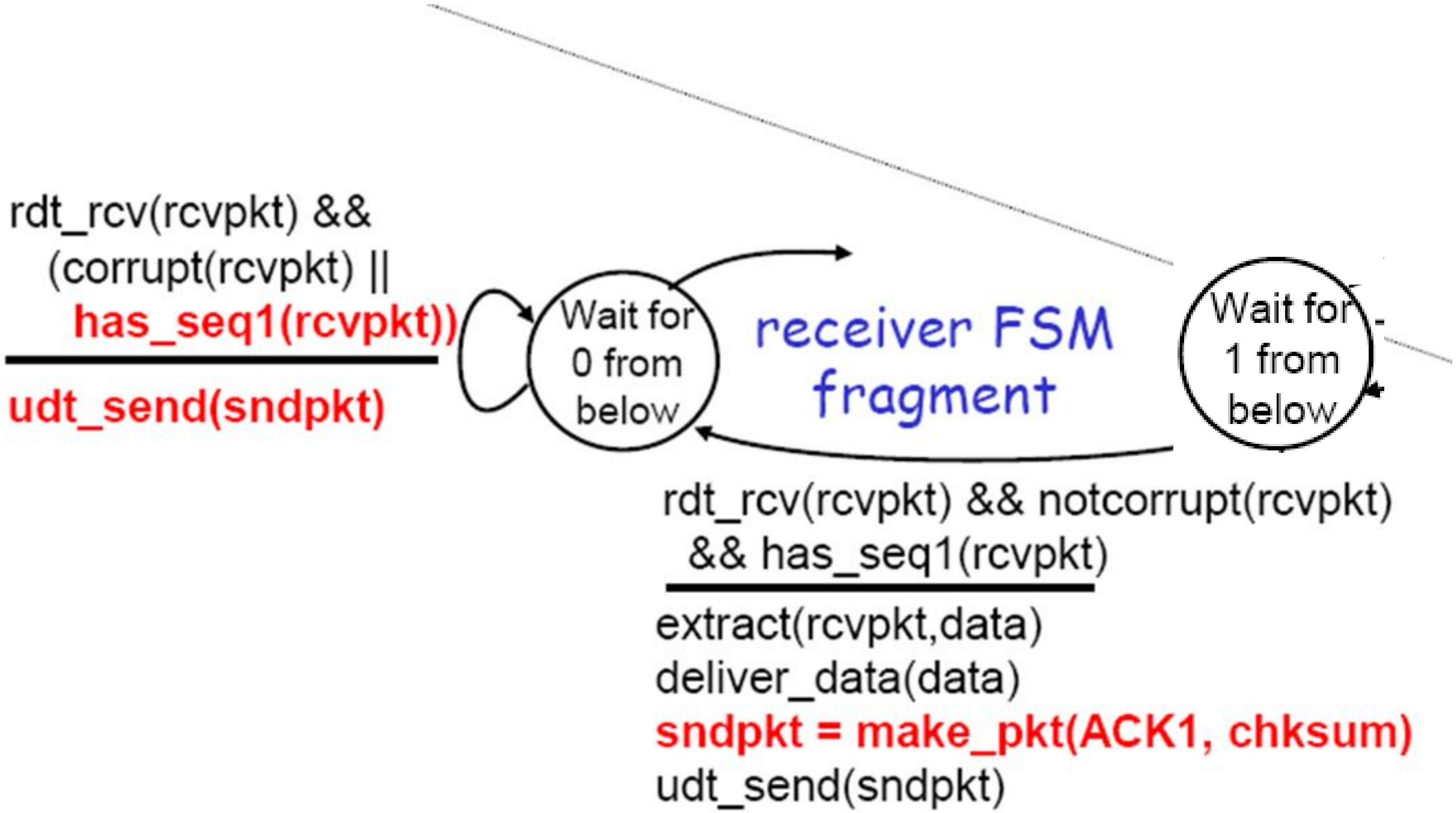
Rdt2.2: sender and receiver

- Sender:



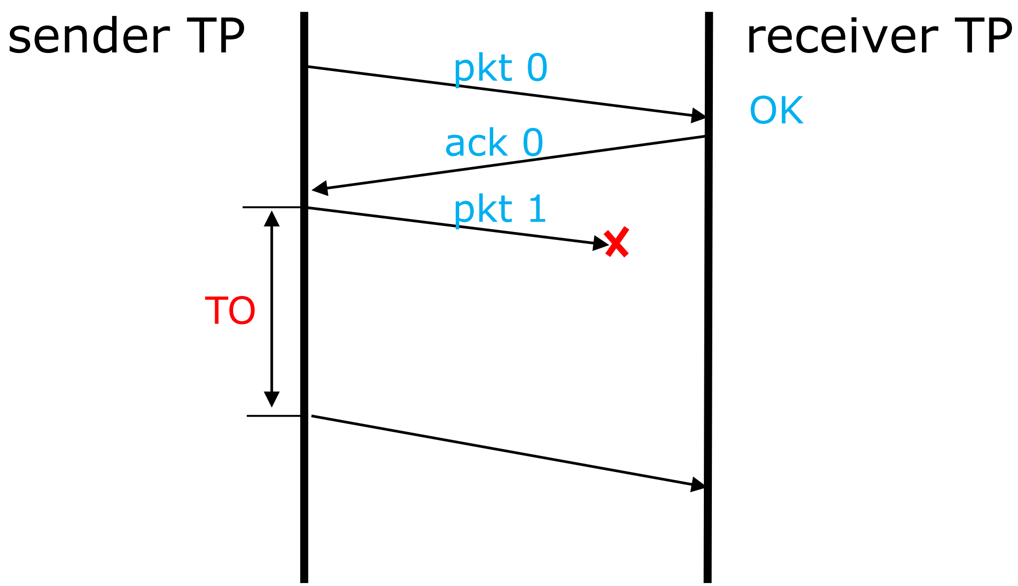
Rdt2.2: sender and receiver

- Receiver:



Rdt3.0: channels with errors *and* loss

- underlying channel can lose packets (data or ACKs)
  - checksum, seq. #, ACKs, retransmissions will be of help, but not enough

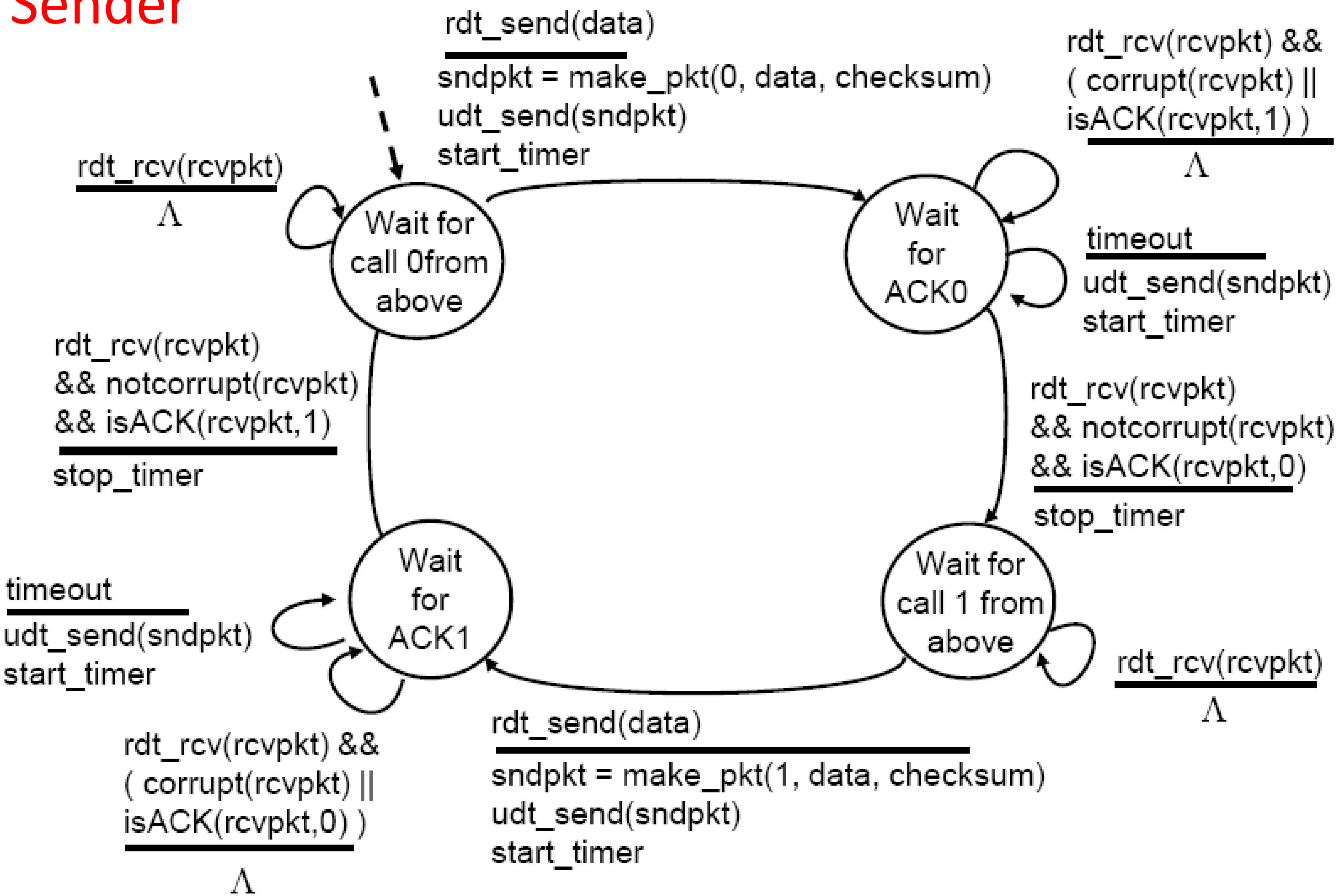


Rdt3.0: channels with errors *and* loss

- Approach:
  - sender waits “reasonable” amount of time for ACK → needs timeout timer (TO) for waiting ACK packet
  - retransmits if no ACK received in this time
  - what happens if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but use of seq. #’s already handles this
    - receiver must specify seq # of pkt being ACKed

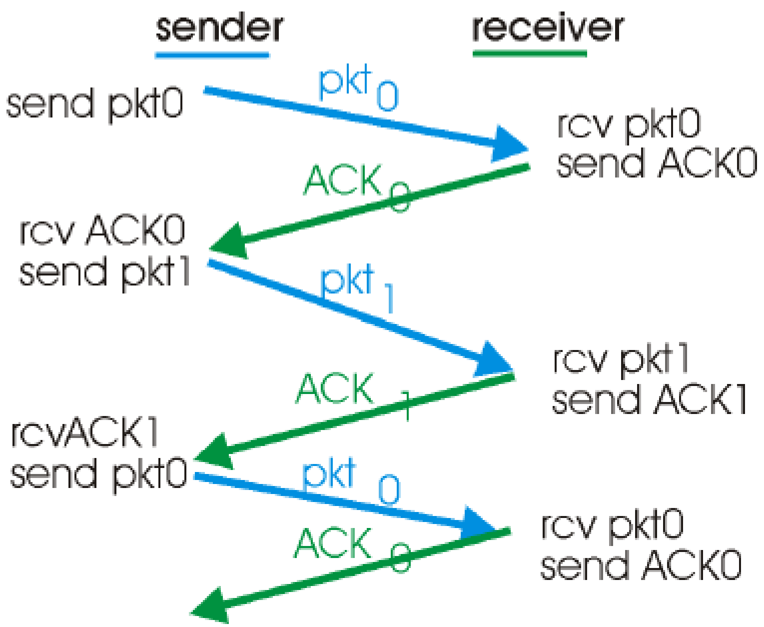
Rdt3.0:

- Sender

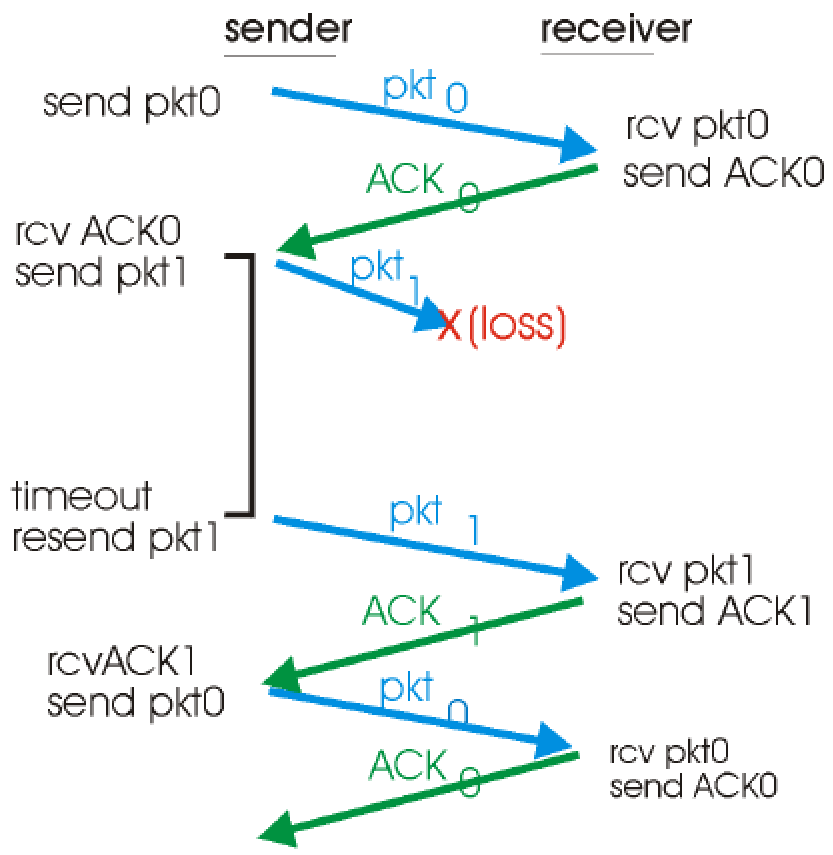


Rdt3.0 Operation

- No loss



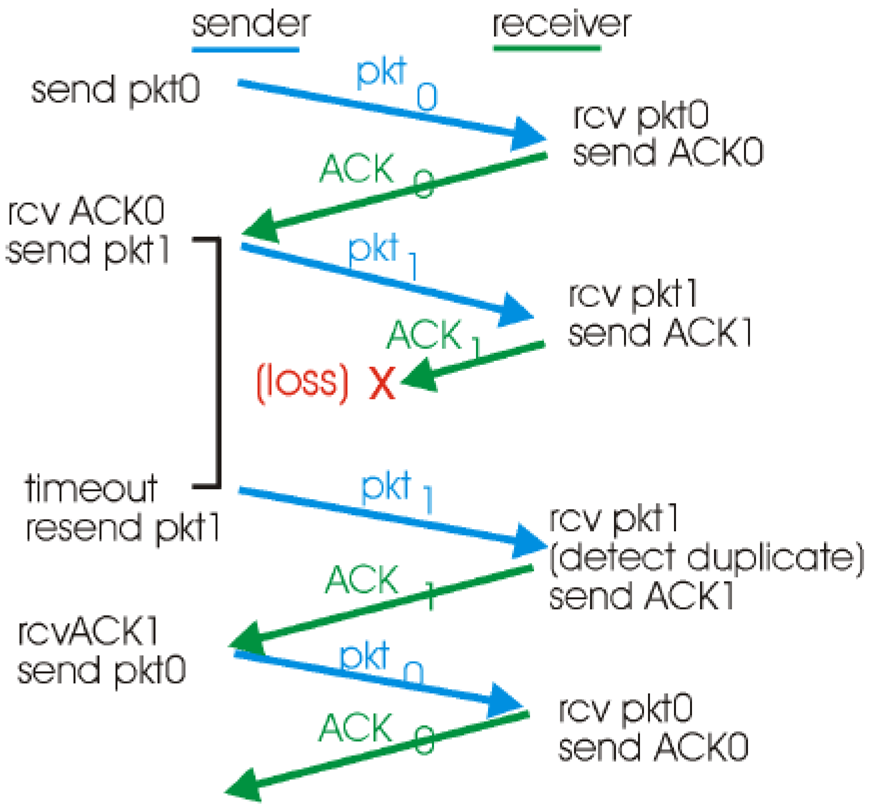
- Lost packet



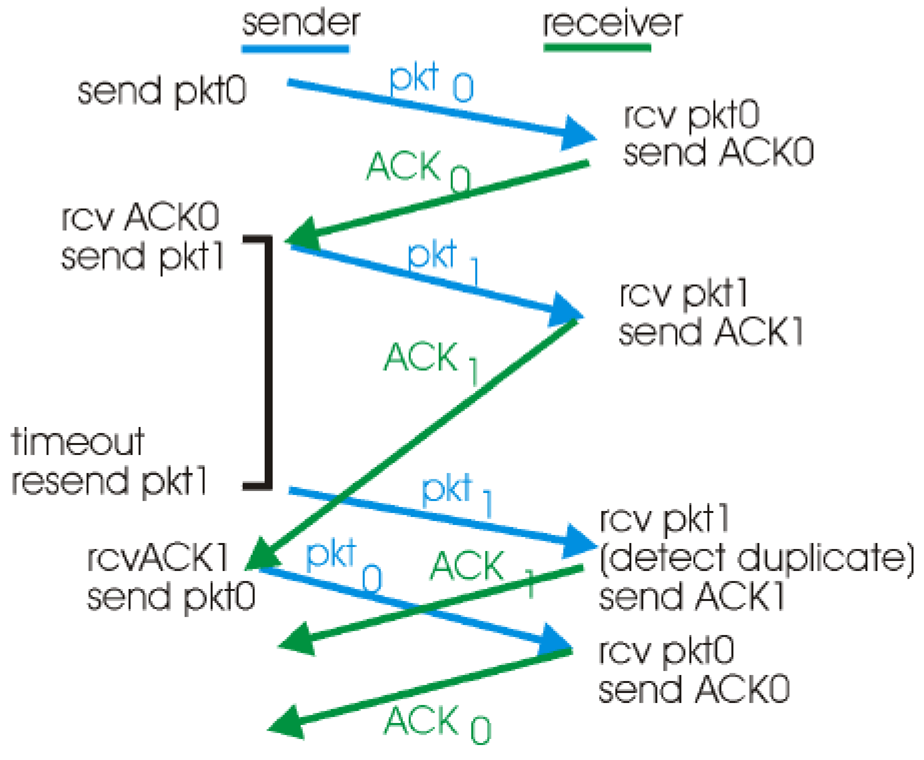
(b) lost packet

Rdt3.0 Operation

- No ACK



- Premature timeout



Performance of Rdt3.0

- Stop-and-wait protocol: rdt3.0 works, but very low performance
  - Ex) 1 Gbps link, 15 ms prop. delay, 8000 bit packet

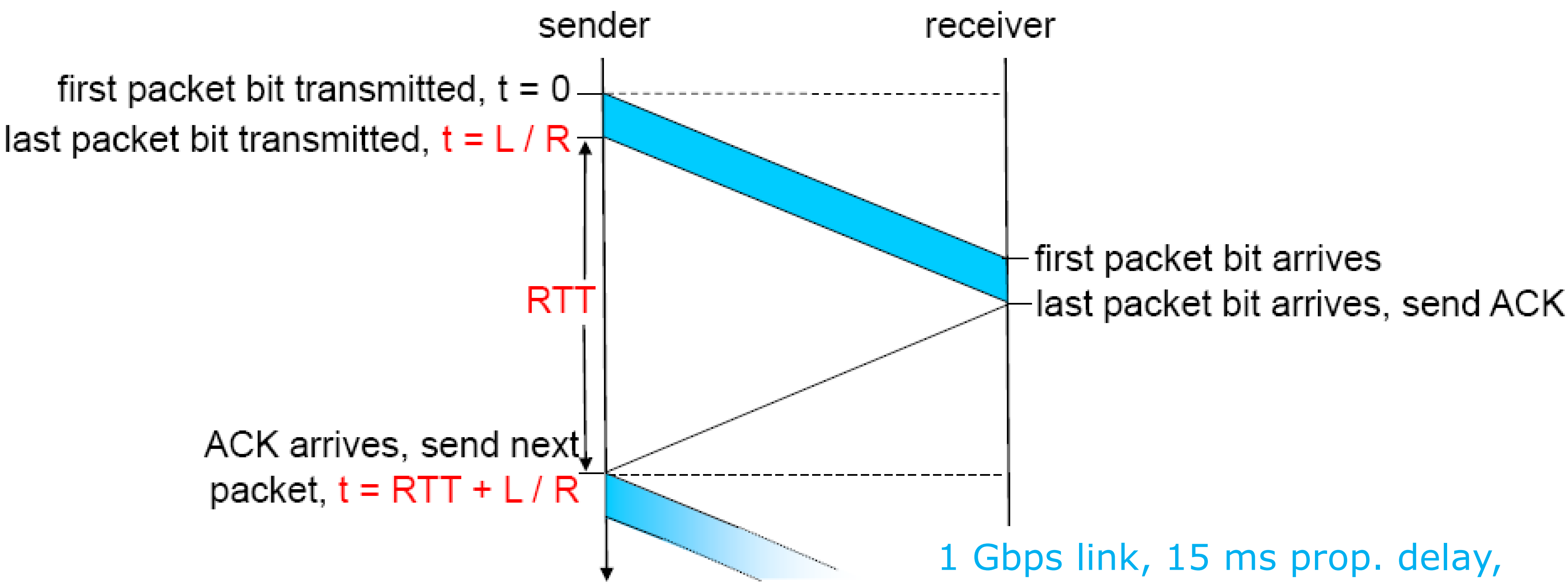
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{ bps}} = 8\text{ microseconds}$$

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

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

  - 1KB pkt every 30 msec → 33kB/s throughput over 1 Gbps link

Rdt3.0: stop-and-wait operation

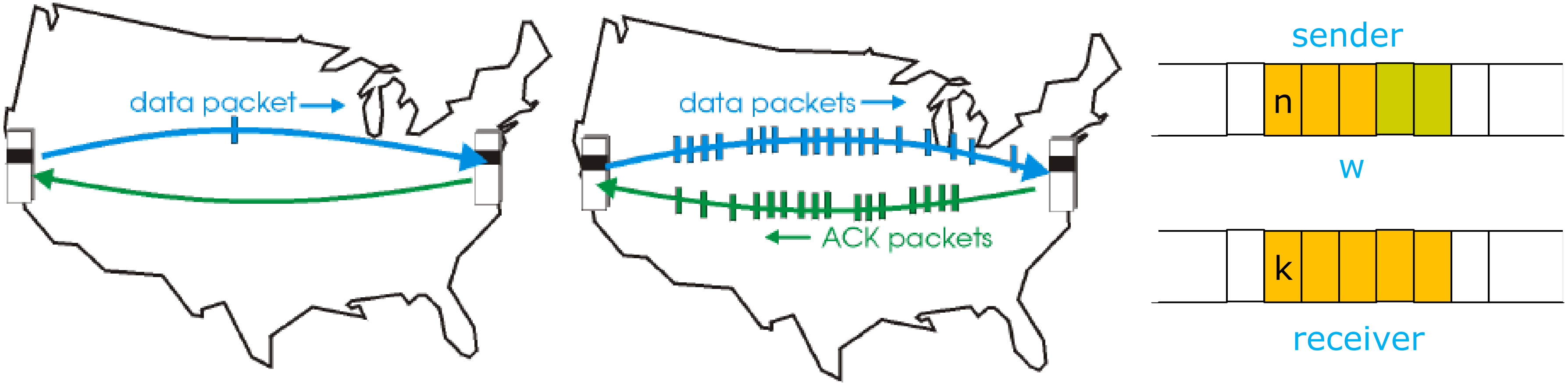


1 Gbps link, 15 ms prop. delay,
8000 bit packet

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

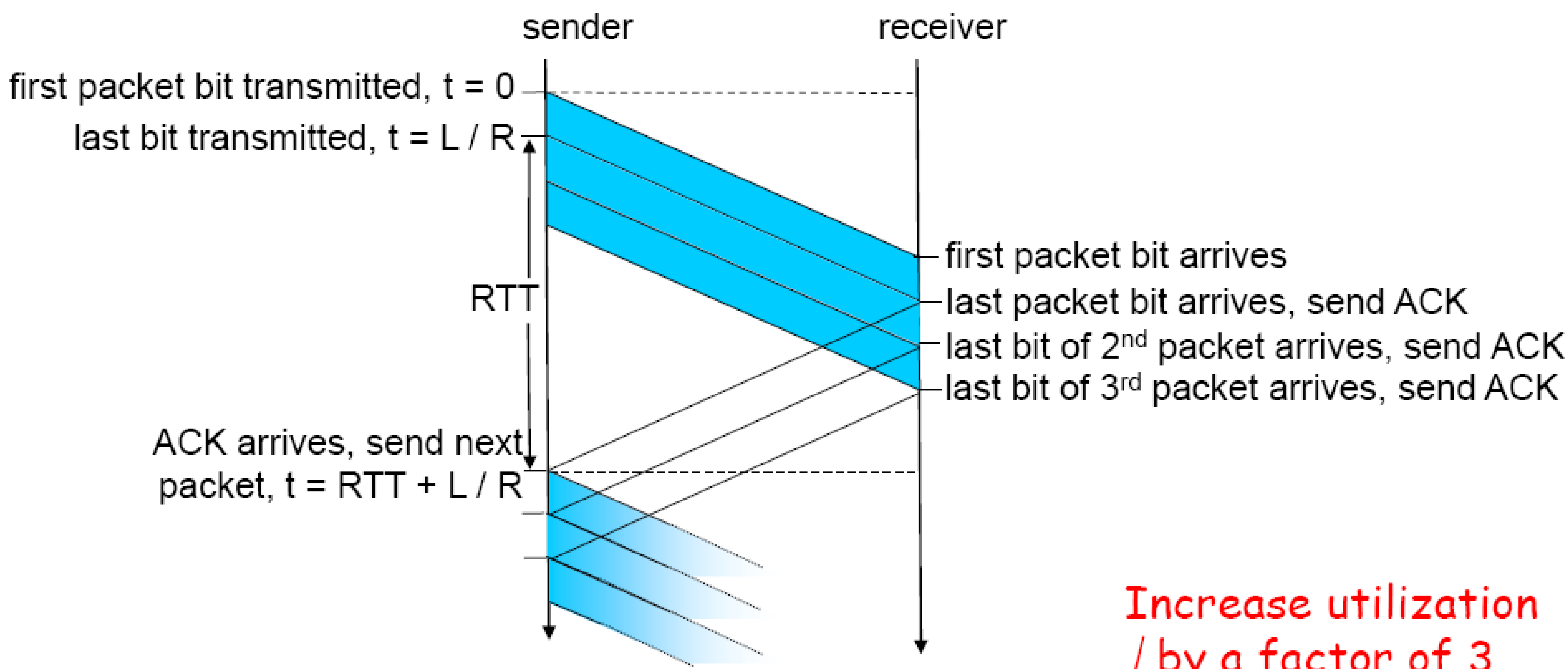
Pipelined Protocols

- Pipelining (sliding window) protocol :
  - sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
  - window of packets (“in-flight” and “can-transmit”) is managed
  - window can slide while transmitting or receiving packets
  - buffering of packets in window at sender and/or receiver



- Two pipelined protocols: *go-Back-N, selective repeat*

Pipelining: increased utilization



Increase utilization by a factor of 3

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Pipelining Protocols

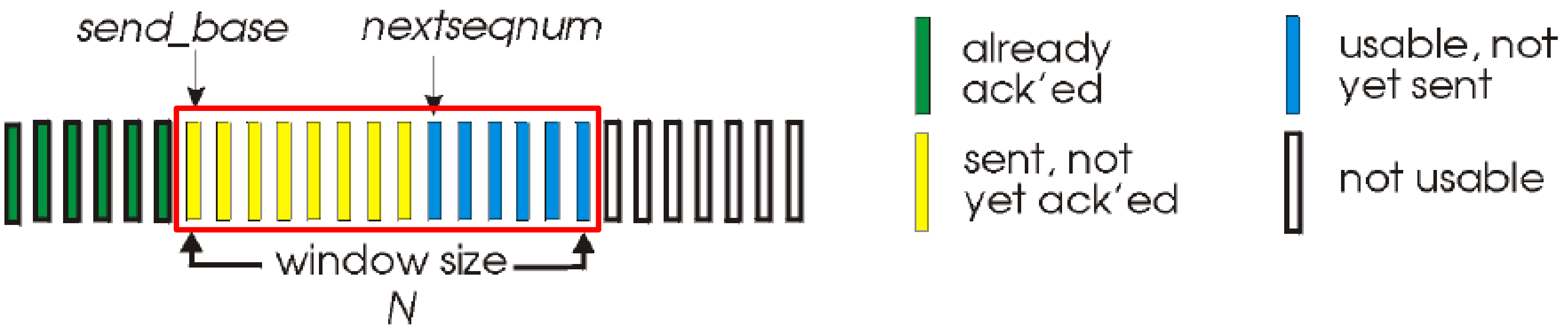
- Go-back-N:
  - Sender can have up to N un-acked packets in pipeline
  - Rcvr receives packets only in the order of the original sequence
  - Rcvr sends cumulative ACKs
  - Sender keeps timer for the oldest un-acked packet (the first packet in the window)
    - If timer expires, retransmit all un-acked packets

Pipelining Protocols

- Selective Repeat:
  - Sender can have up to N un-acked packets in pipeline
  - Rcvr can receive packets (in the window) in an arbitrary order
  - Rcvr acks individual packets
  - Sender maintains timer for each un-acked packet
    - When timer expires, retransmit only un-acked packet

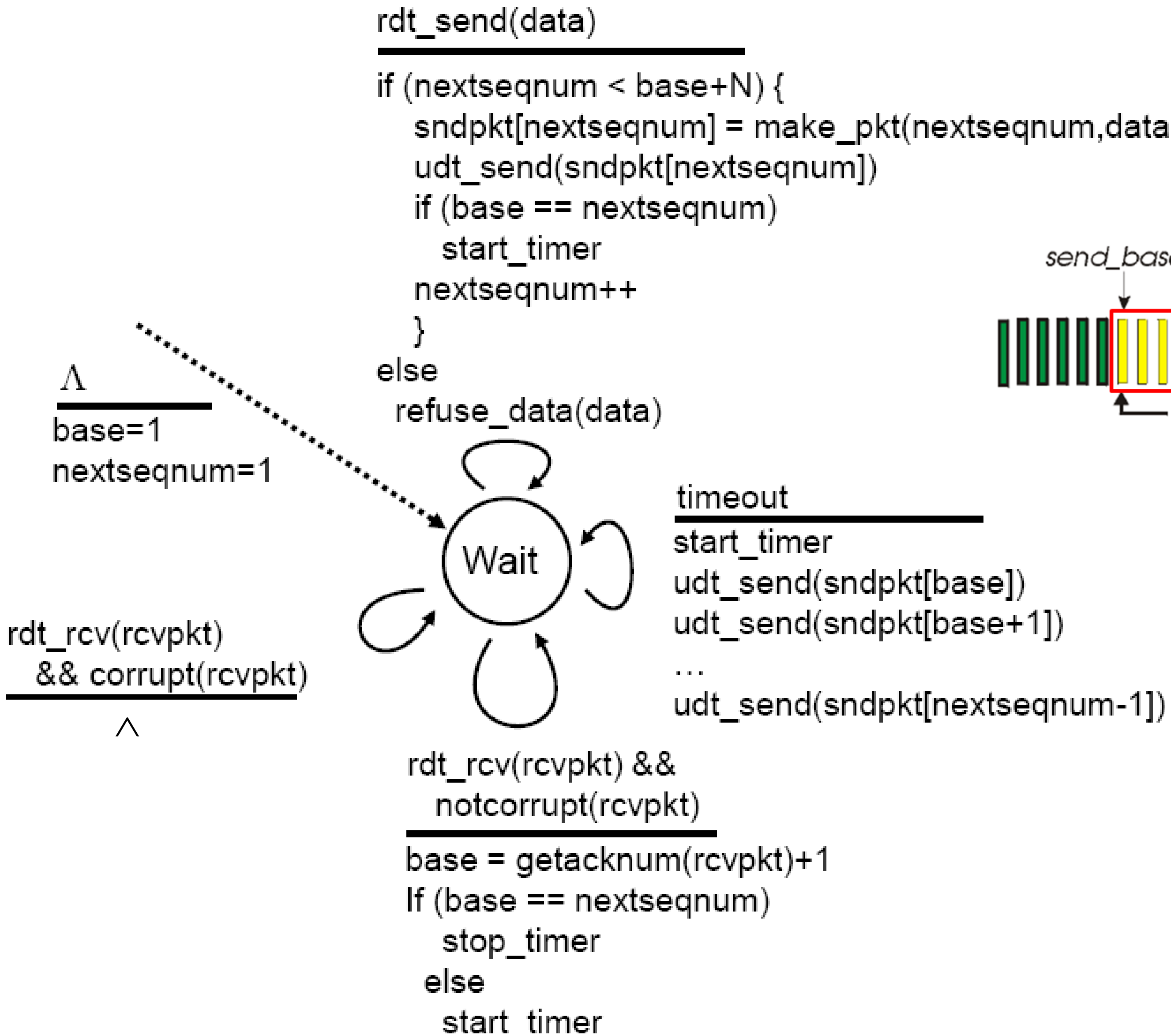
Go-Back-N

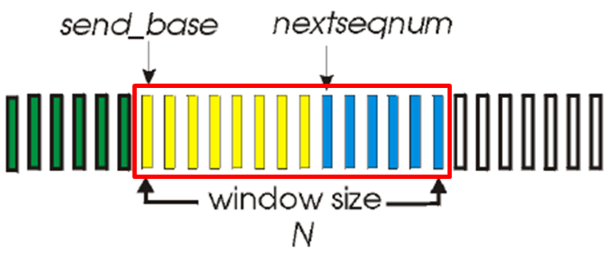
- Sender:
  - k-bit seq # in pkt header
  - "window" of up to N, consecutive un-ack'ed pkts allowed



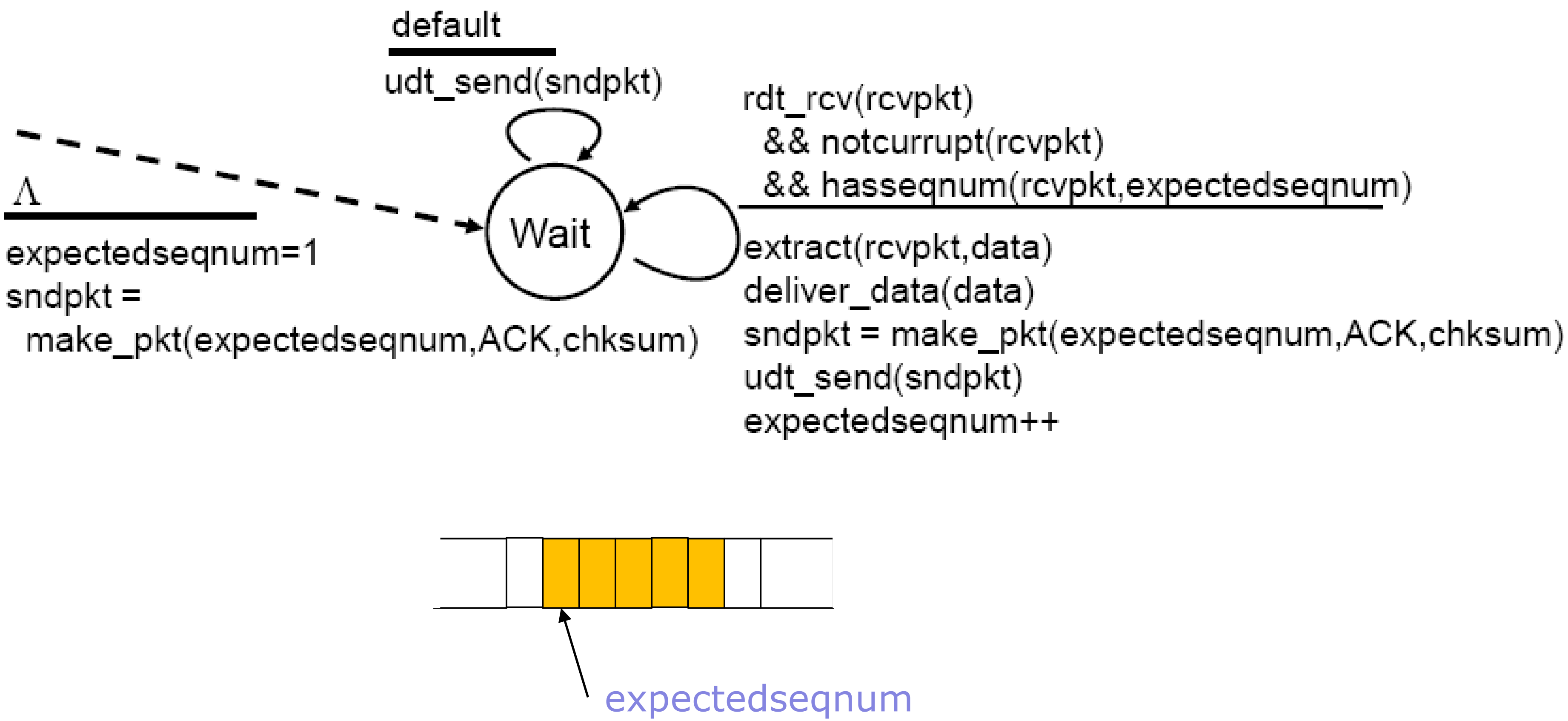
  - cumulative ACK : ACK(n): ACKs all pkts up to, including seq # n
  - timer for the first packet in the window
  - *timeout(n):* retransmit pkt *n* and all un-ack'ed pkts in window

GBN: sender extended FSM



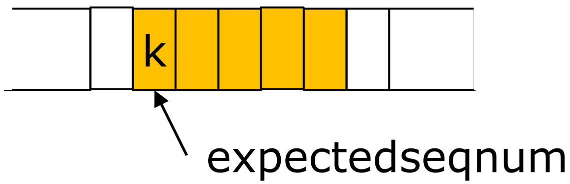


GBN: receiver extended FSM



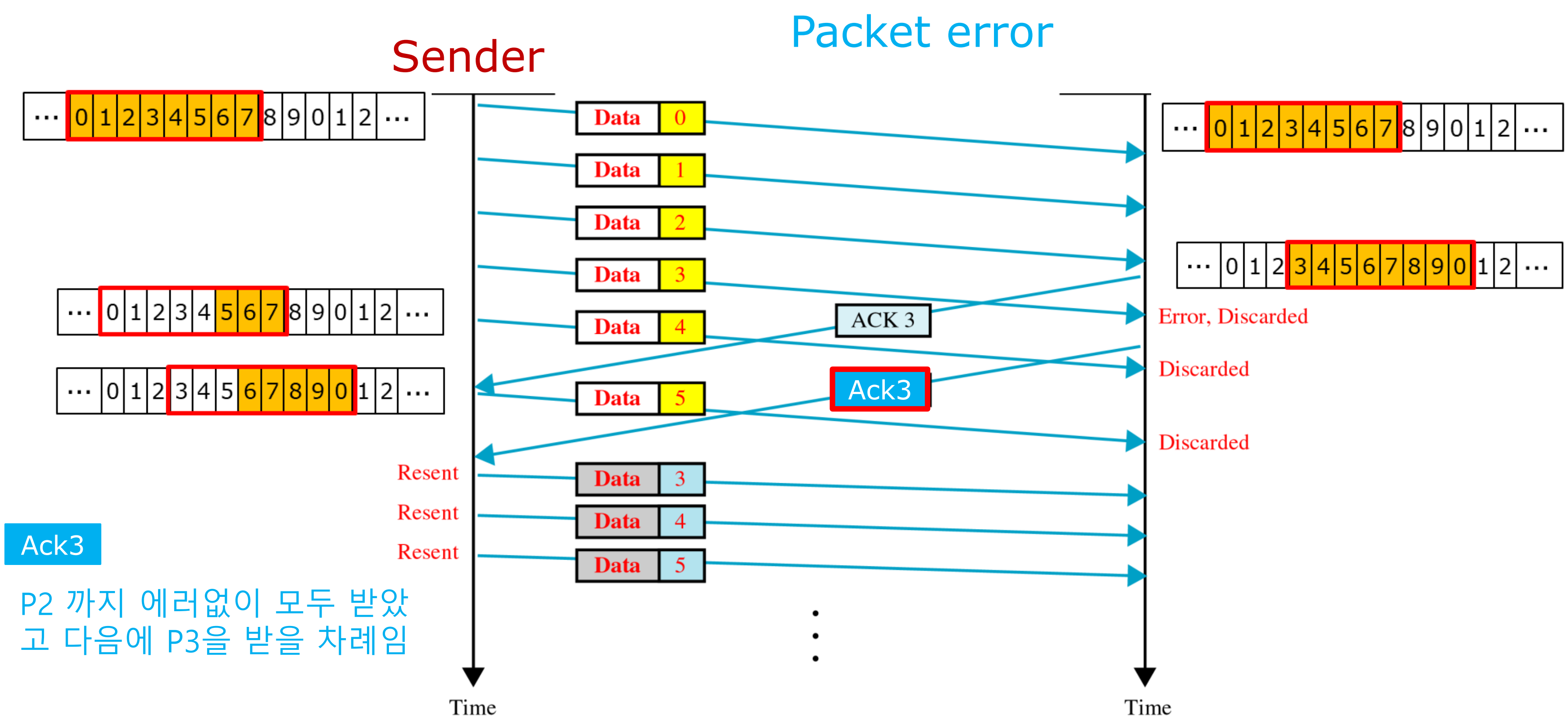
GBN: receiver extended FSM

- Receiver:
  - ACK-only: always send ACK for correctly-received pkt with highest in-order seq #
  - may generate duplicate ACKs
  - need only remember expectedseqnum
  - out-of-order pkt:
    - discard (don't buffer) -> no receiver buffering!
    - Re-ACK pkt with highest in-order seq # ← duplicate ACK



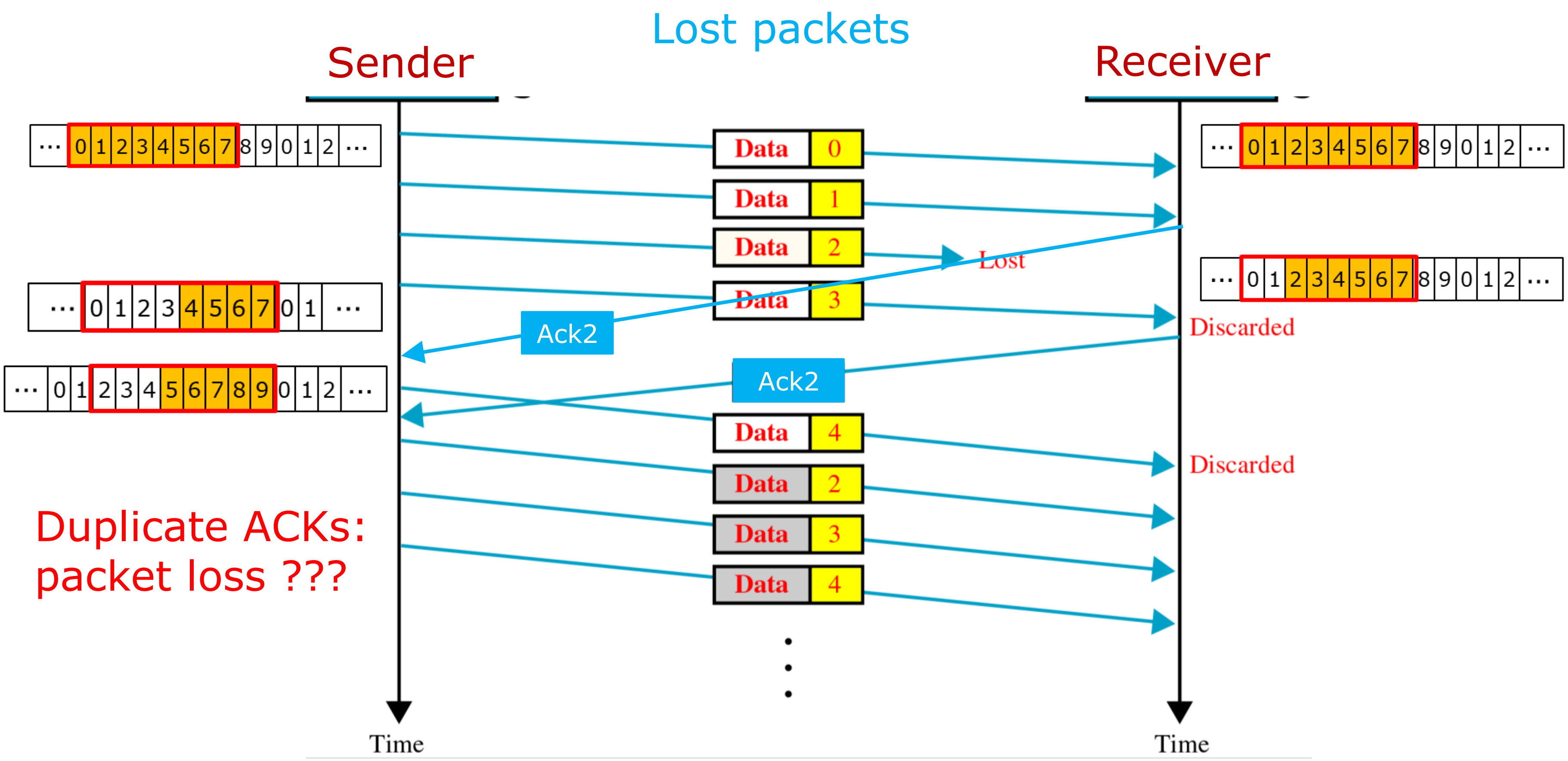
Pipelined Protocol: Go-back N

- Go-back-*n*
  - If one frame is lost or damaged, all frames after the frame are retransmitted



Pipelined Protocol: Go-back N

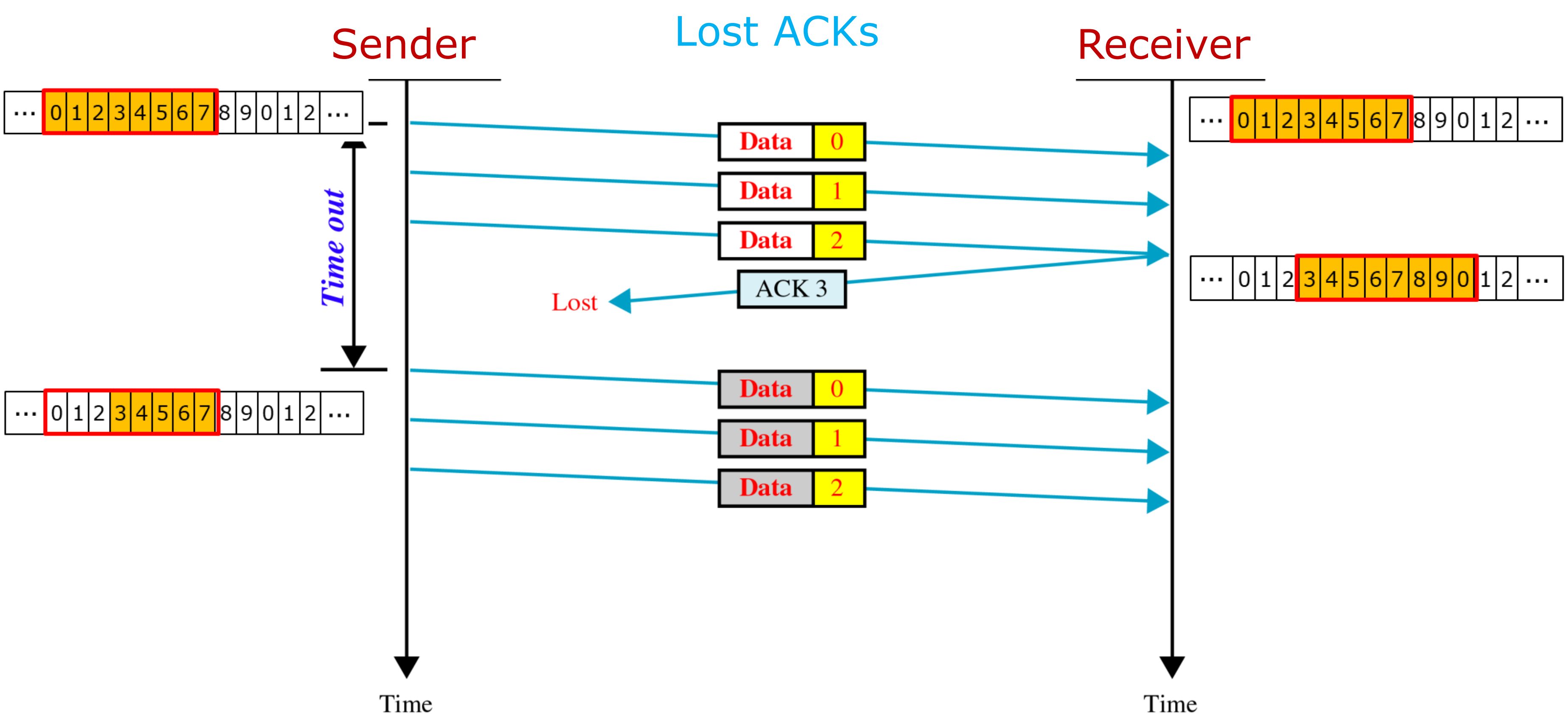
- Go-back-*n*



Duplicate ACKs: packet loss ???

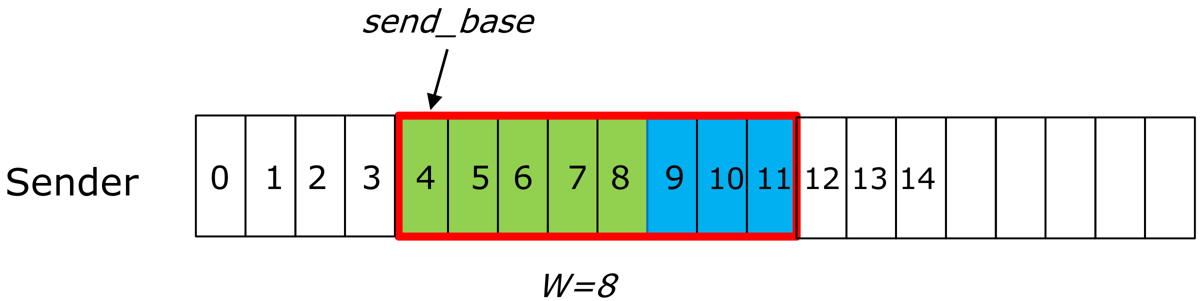
Pipelined Protocol: Go-back N

- Go-back-*n*
  - need packet timeout timer



Pipelined Protocol: Go-back N

- Sender



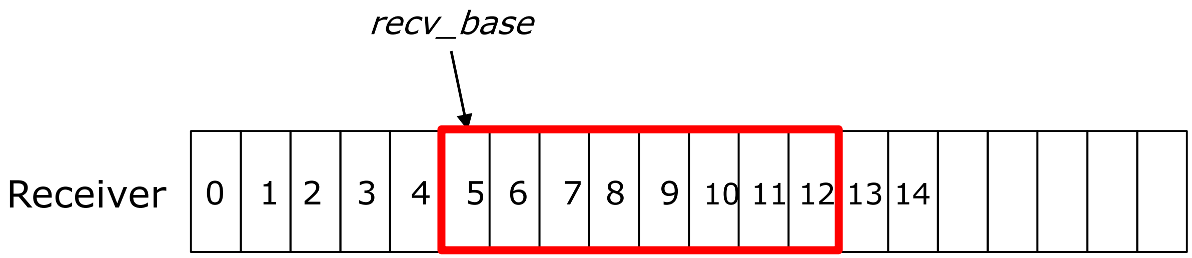
ACK 5: P4까지 모두 다 받았고, 다음에 P5를 받을 차례임

What happen if the following event occur?

1) receive ACK 5 2) receive ACK 6 3) receive ACK 4 4) TO happen

Pipelined Protocol: Go-back N

- Receiver



What happen if the following event occur?

1) receive Pkt 5 2) receive Pkt 6 3) receive Pkt 4

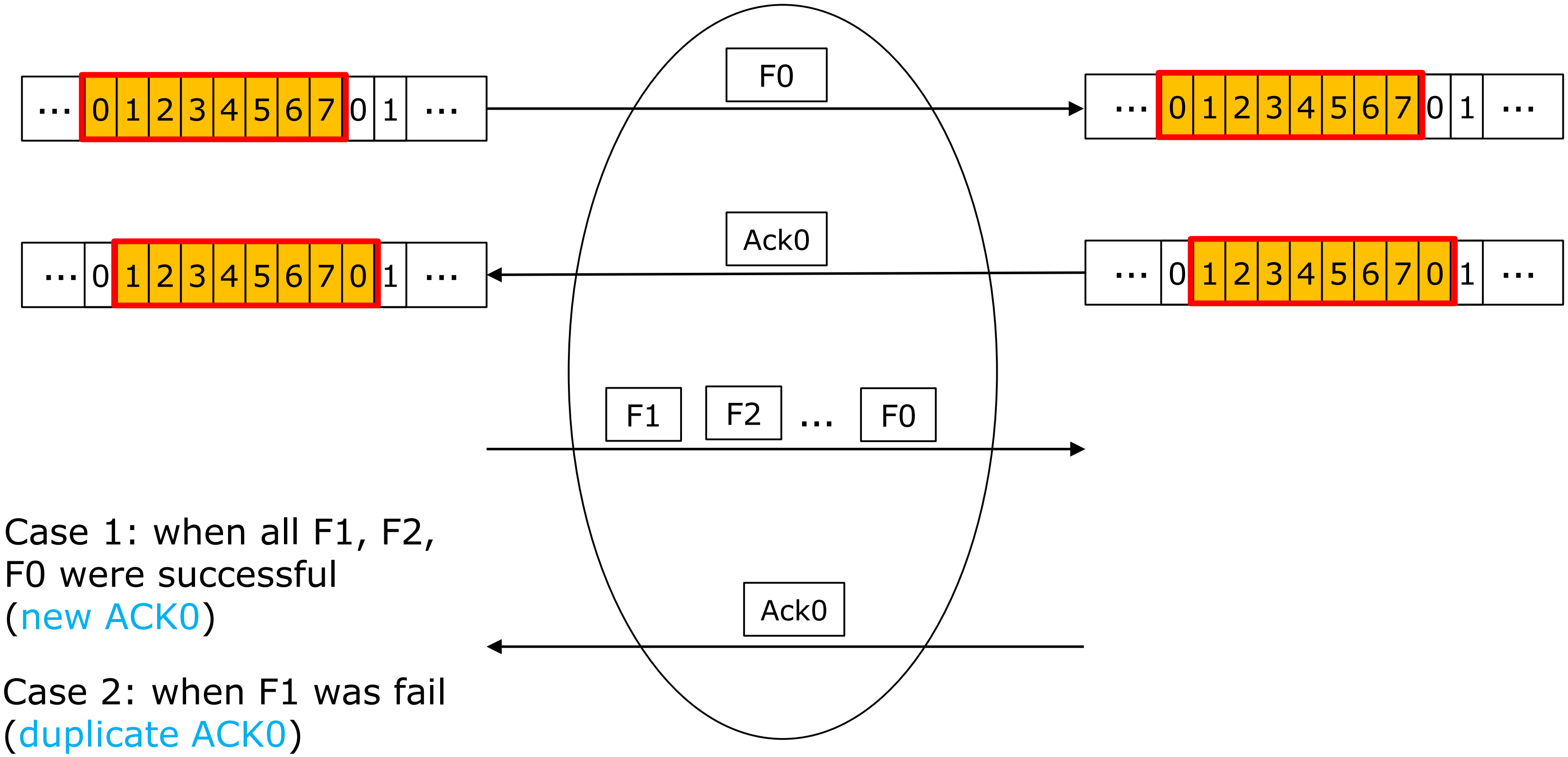
Pipelined Protocol: Go-back N

- Go-back-n : window size
  - Max window size: N – 1 (max number of seq num – 1)
  - When the window size becomes N
    - Sender: sends P0 and receives *ACK 0*
    - Sender: sends next P1, P2, …, P7, P0
    - Sender: receives *ACK0*
    - The sender can not determine whether the *ACK0* is the duplicate of the previous ACK0 or is a new one

ACK 0: P0까지 에러없이 받았음

Pipelined Protocol: Go-back N

- Go-back-n : window size
  - when W=8 in 3-bit SN



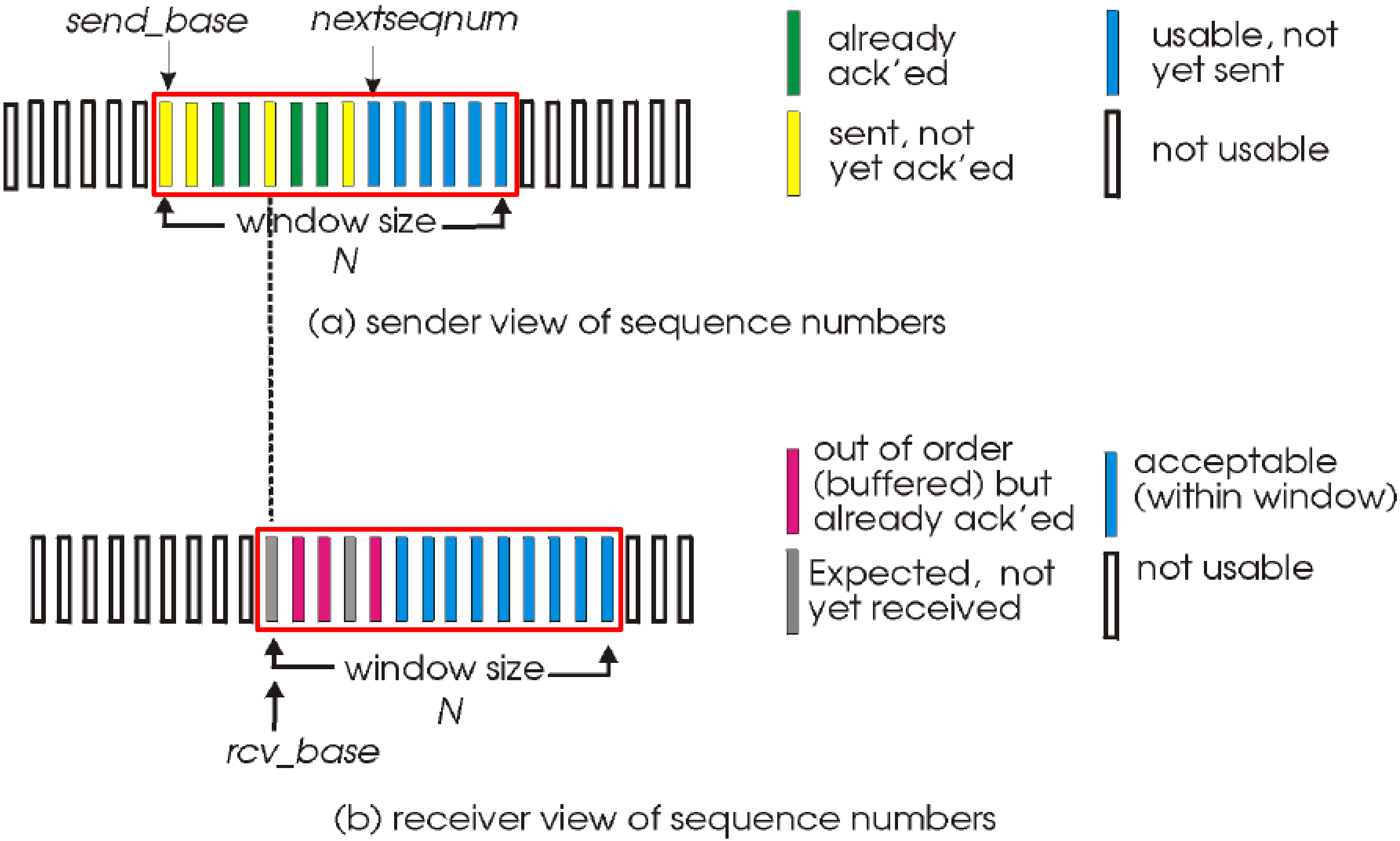
Case 1: when all F1, F2, F0 were successful (new ACK0)

Case 2: when F1 was fail (duplicate ACK0)

Pipelined Protocol: Selective-reject

- Selective-reject (Selective-repeat) protocol
  - Receiver
    - can receive packets out of sequence
    - Sends "ACK n" when successfully received "packet n"
    - sends "NAK n" when an error detected in "packet n"
  - Sender
    - can send packets out of sequence
    - keeps separate TOs for each packet transmitted
    - Re-transmits only the requested packet when it receives a NAK or TO happened

Selective repeat: sender, receiver

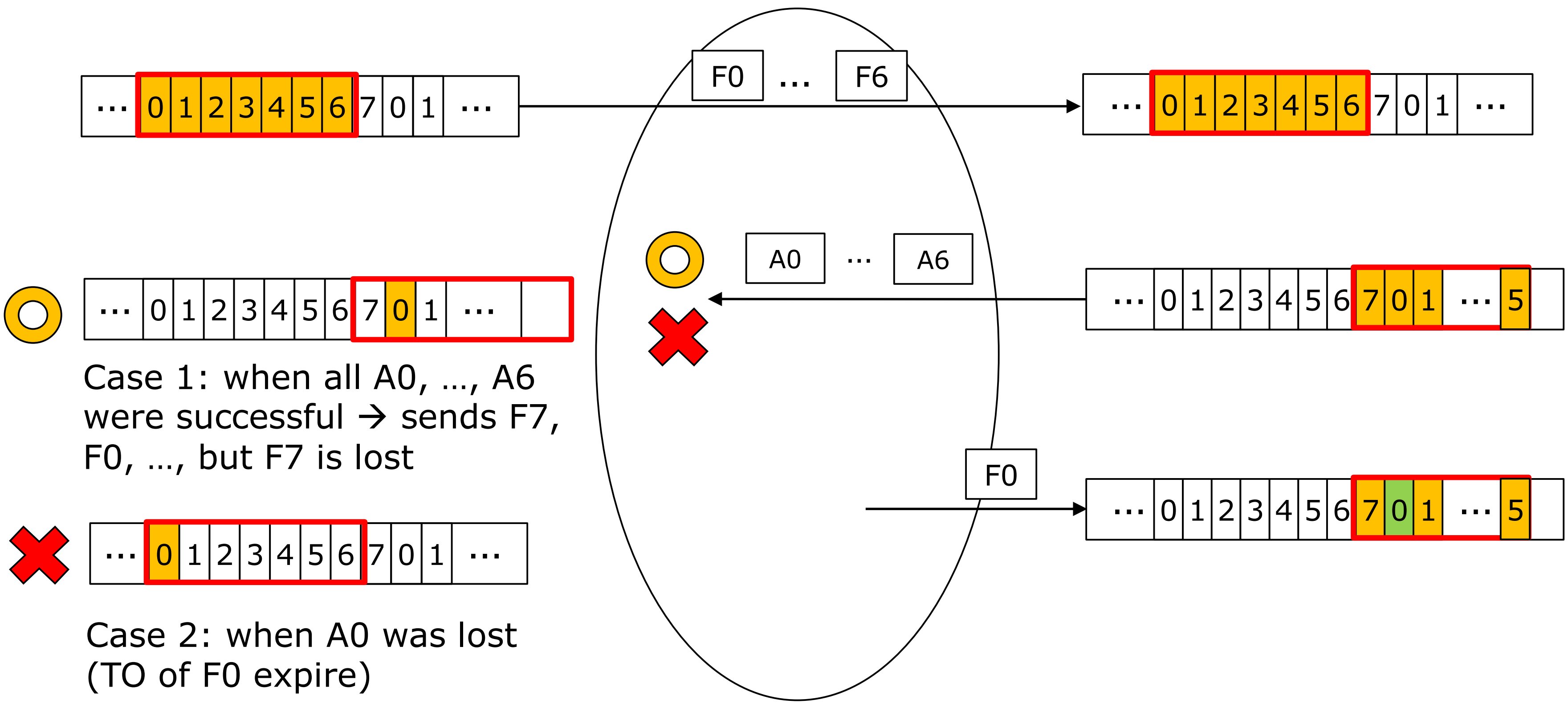


Pipelined Protocol: Selective-reject

- Selective-reject
  - Max window size: $\lfloor (N+1)/2 \rfloor$
  - When the window size = (N-1) (e.g. N = 8)
    - Station A: sends P0, P1, …, P6 to station B
    - Station B: sends ACK0, ACK1, …, ACK6 (but all lost) => expands the window to accept P7, P0, P1, …, P5
    - Station A: TO timer of P0 expires and retransmits P0
    - Station B thinks the received P0 as a new one and accepts it (wrong !!)

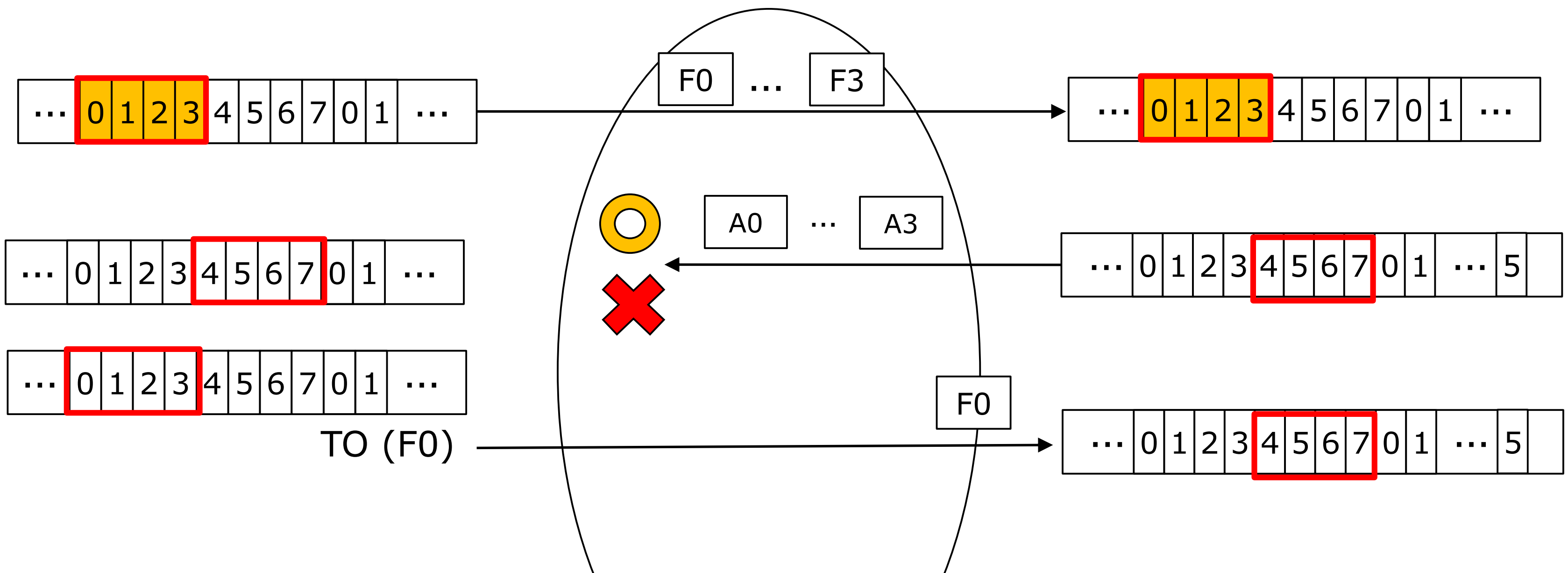
Pipelined Protocol: Selective-reject

- Selective-reject : window size
  - Problem: when W=7 in 3-bit SN



Pipelined Protocol: Selective-reject

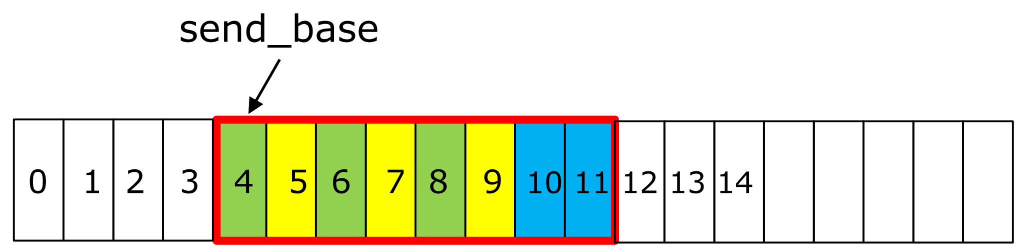
- Selective-reject : window size
  - when W=7 in 3-bit SN



There has to be no packet with the same ID between the current window and the window sliding after receiving all packets in the window successfully

Selective Repeat

- Sender



ACK 4: P4를 성공적으로 받았음

What happen if the following event occur?

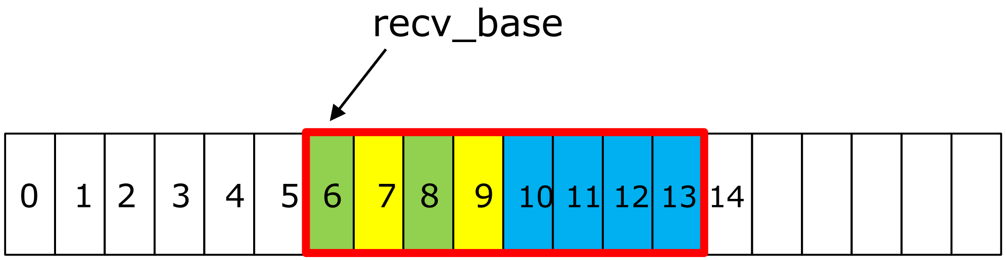
1) ACK4

2) ACK6

3) TO of P4

Selective Repeat

- Receiver



What happen if the following event occur?

1) receive Pkt 6
2) receive Pkt 8
3) receive Pkt 6 (error)
4) receive Pkt 6 after Pkt 8
5) receive Pkt 5

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:

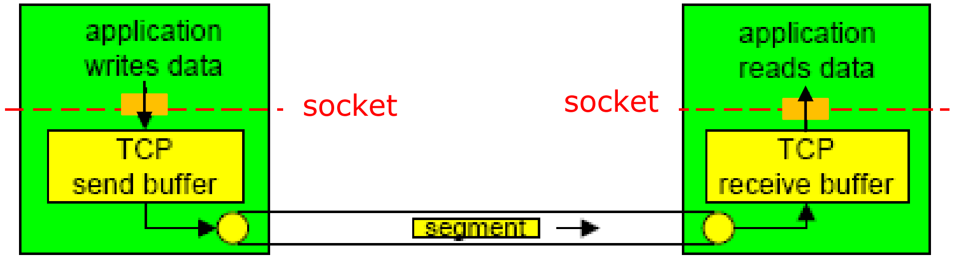
- ignore

Chap 3

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

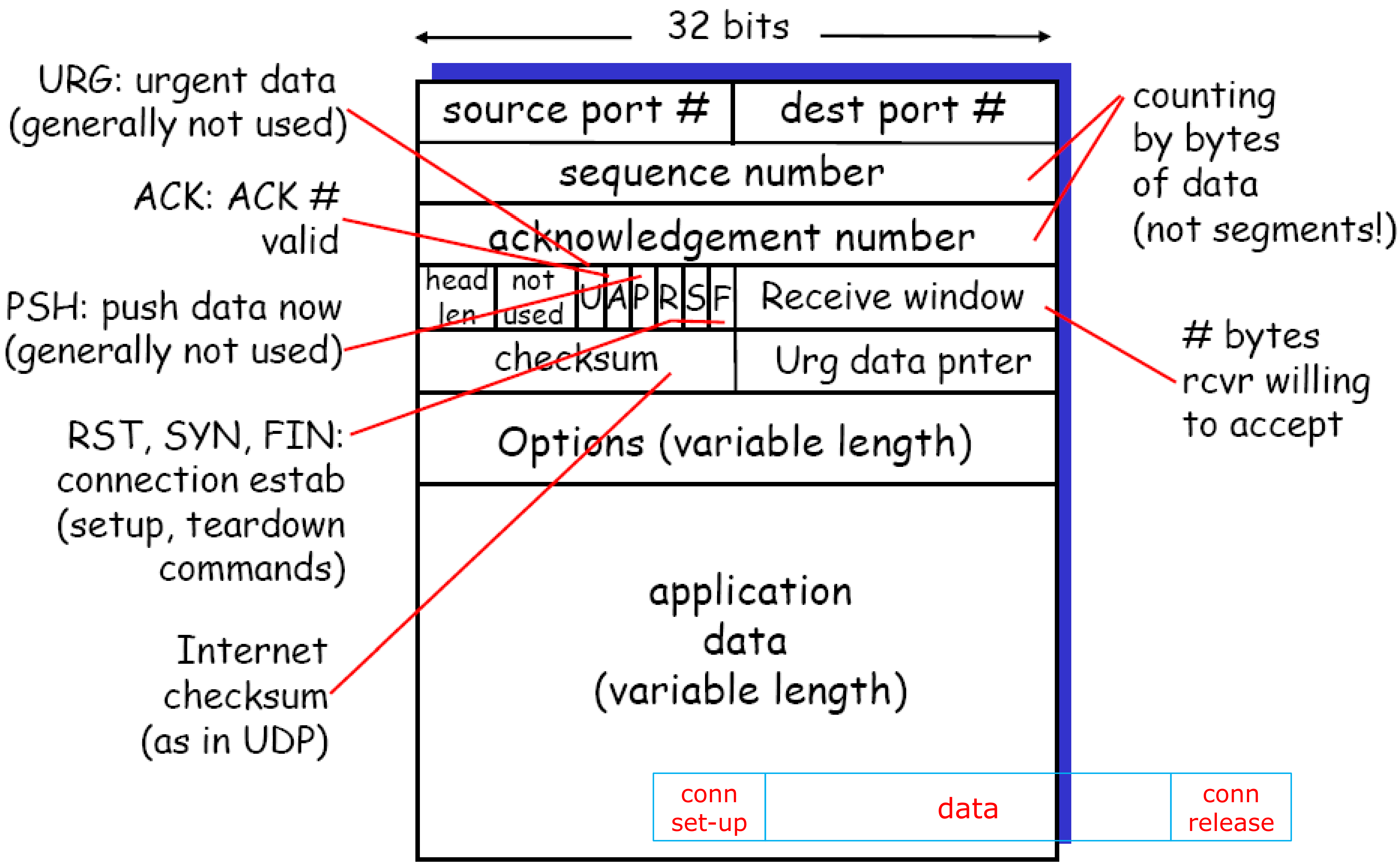
TCP: Overview
[RFCs: 793, 1122, 1323, 2018, 2581]

- point-to-point connection-oriented
  - one sender, one receiver
  - 3-way handshaking: initialize sender and receiver state before data exchange
- pipelined flow control:
  - TCP congestion and flow control set window size
- *sender & receiver side buffering*



- reliable, in-order byte steam:
  - no message boundaries
- full duplex data transmission:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- flow controlled :
  - sender will not overwhelm receiver

TCP Segment Format



TCP Seq. #'s and ACKs

Seq. #'s:

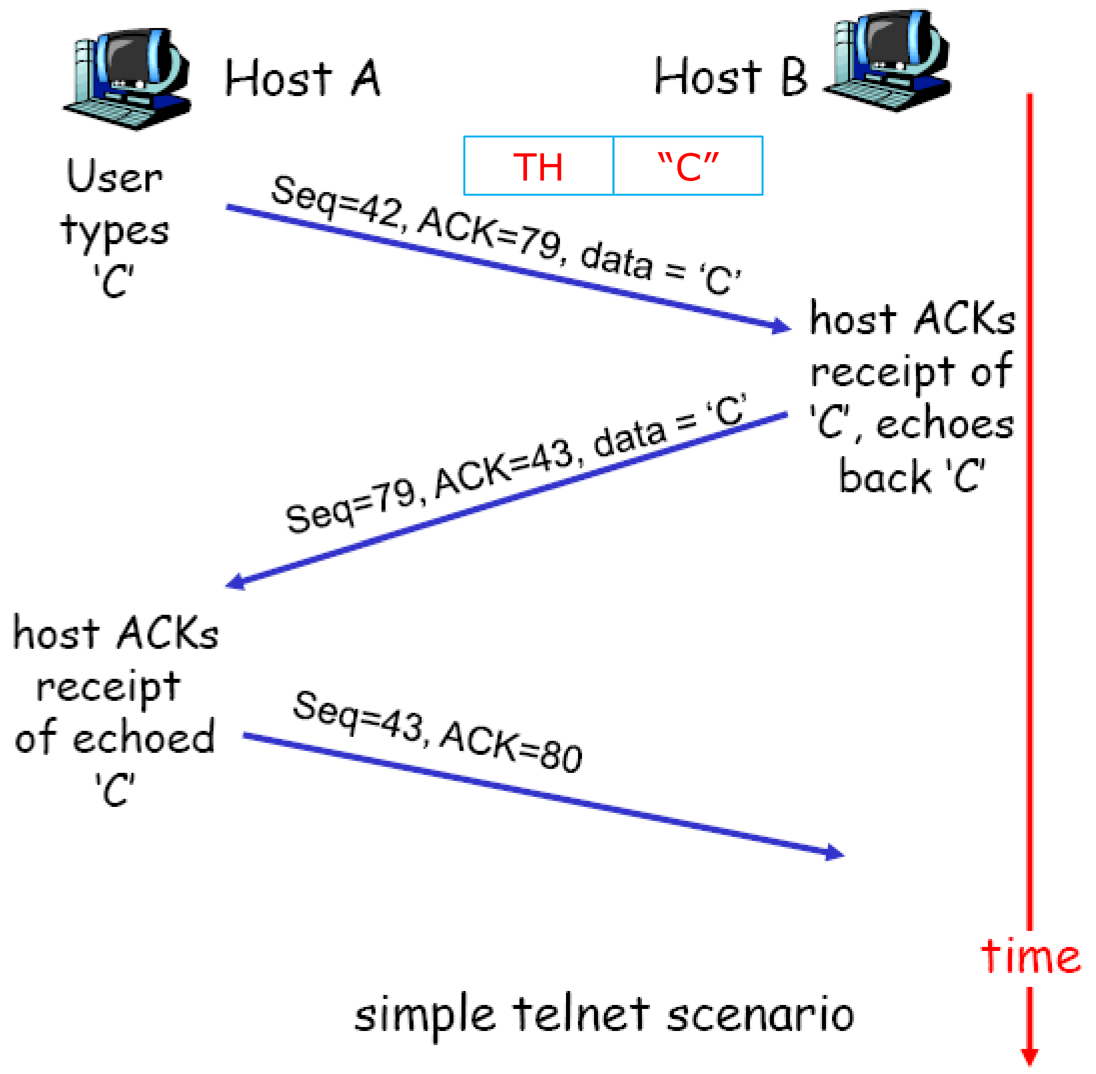
- byte stream "number" of first byte in segment's data

ACKs:

- 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

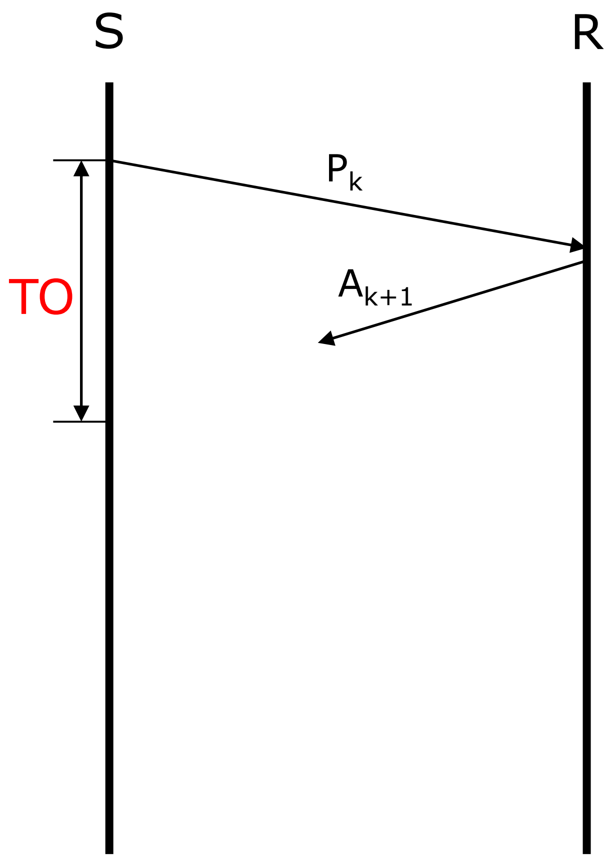


simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set timeout value?

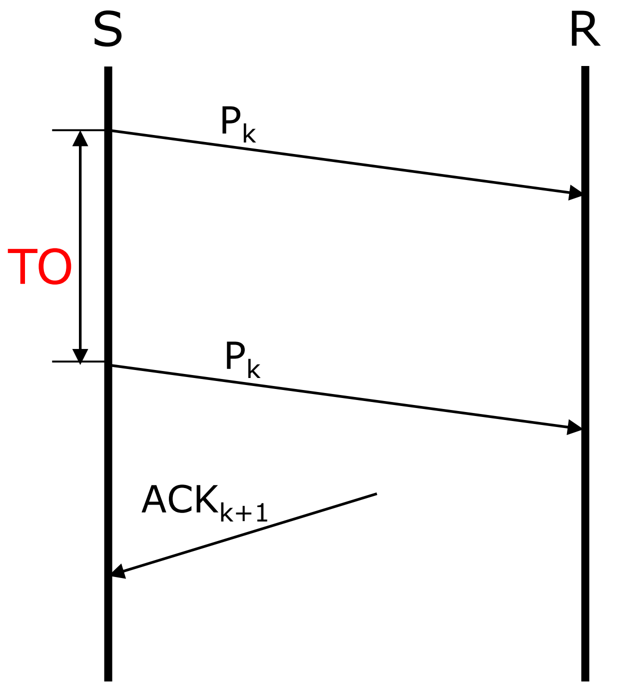
- must be TO > RTT
  - but RTT is variable
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

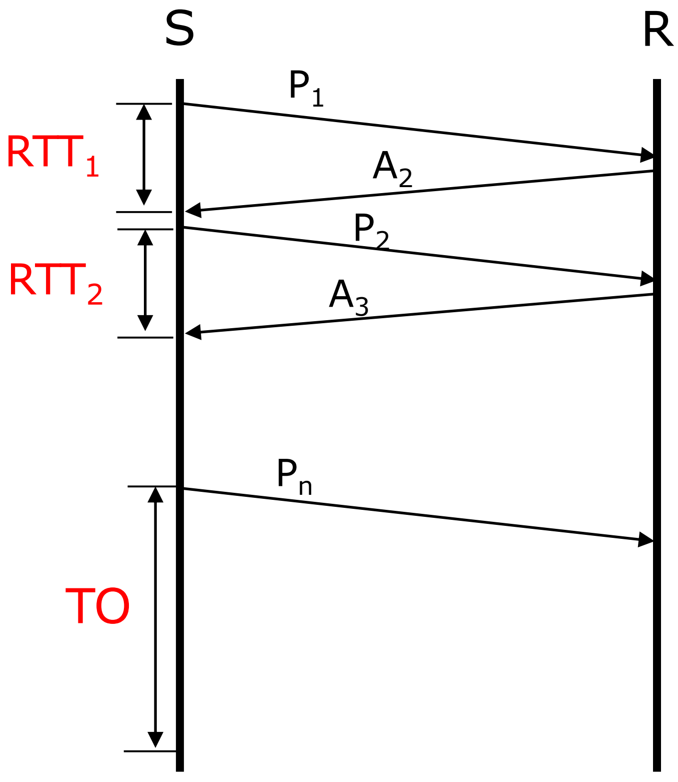


TCP Round Trip Time and Timeout

Q: how to estimate RTT?

- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions





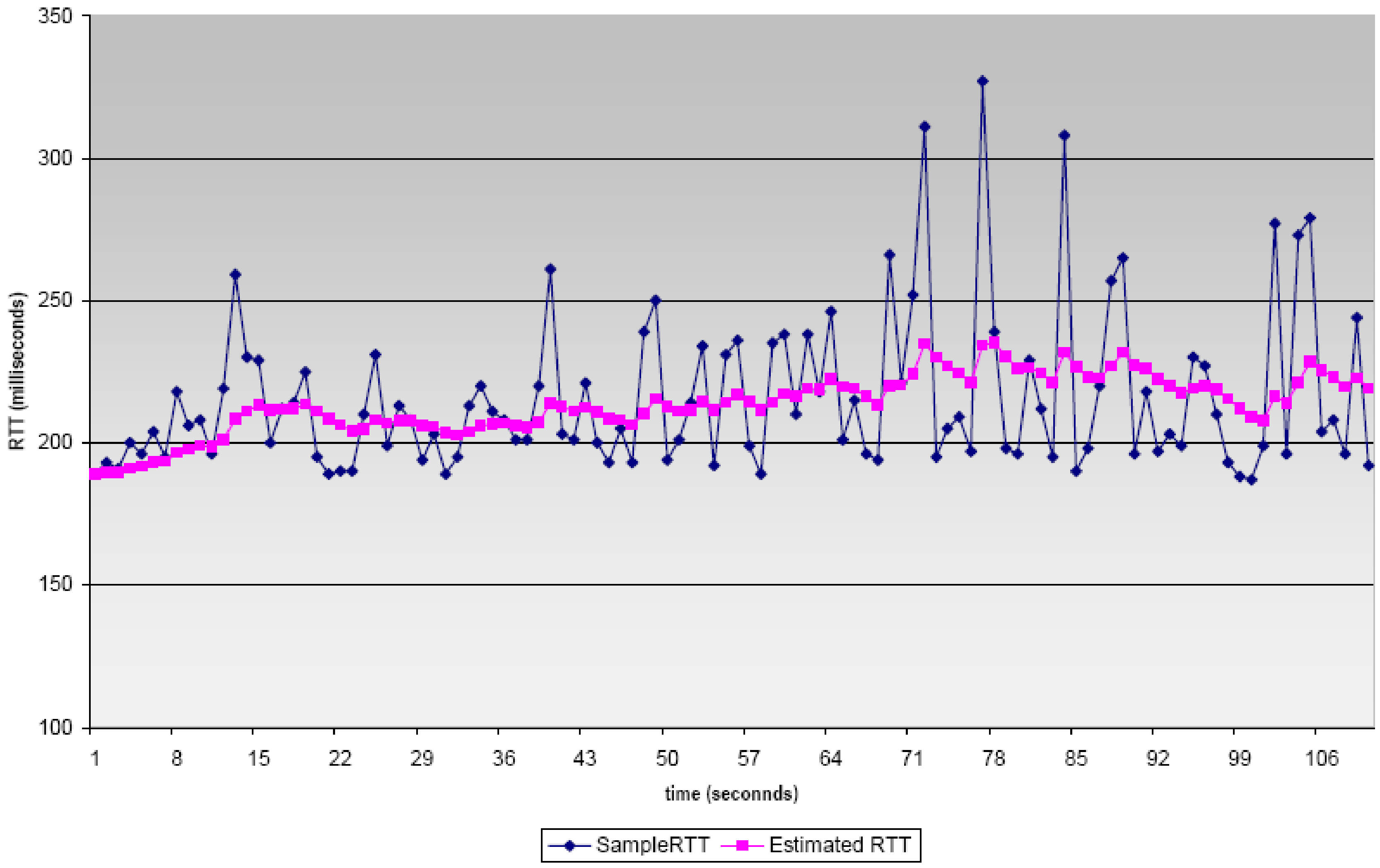
TCP Round Trip Time and Timeout

- SampleRTT will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current SampleRTT

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

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

Example RTT estimation



TCP Round Trip Time and Timeout

Setting the timeout

- **`TO = EstimtedRTT`** plus "safety margin"
  - larger variation in **`EstimatedRTT`** **->** larger safety margin
- DevRTT : weighted average of | SampleRTT-deviation |

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

(typically, β = 0.25)
```

- Setting timeout interval

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

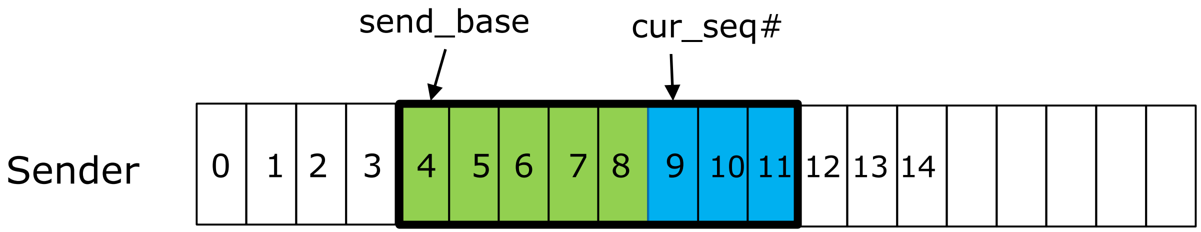
TCP reliable data transfer

- Pipelined segments: sliding window protocol
- Cumulative ACKs; ACK n means receiver received up to (n-1)
- Receiver receives out-of-order segments
- TCP uses single retransmission timer; the first segment in window
- Retransmissions are triggered by:
  - timeout events, duplicate ACKs
  - sender retransmits only the lost segment

- Initially consider simplified TCP sender:
  - ignore duplicate ACKs, flow control, congestion control

Simplified TCP sender events:

- data received from application
  - maintains a sliding window for pipelined transmission

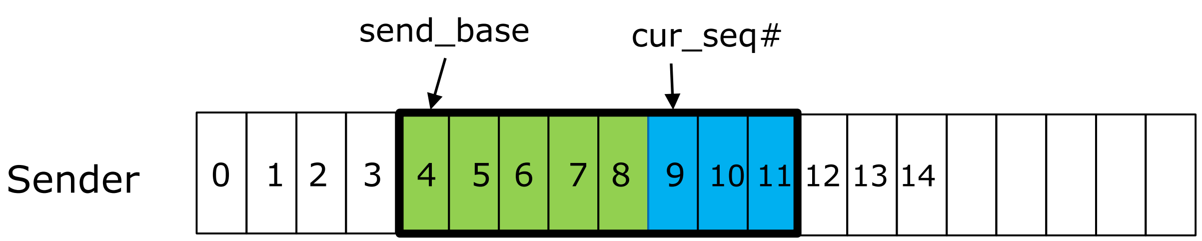


  - create segment with cur_seq #
  - cur_seq # is byte-stream num of first byte in segment
  - start timer if not already running
  - expiration interval: `TimeOutInterval`

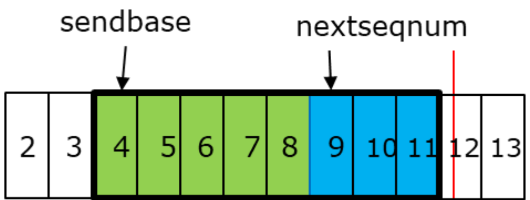
```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Simplified TCP sender events:

- timeout:
  - retransmit segment that caused timeout
  - restart timer
- ACK rcvd: ACK k
  - If acknowledges previously un-acked segments
    - Update send_base
    - start timer if there are outstanding segments

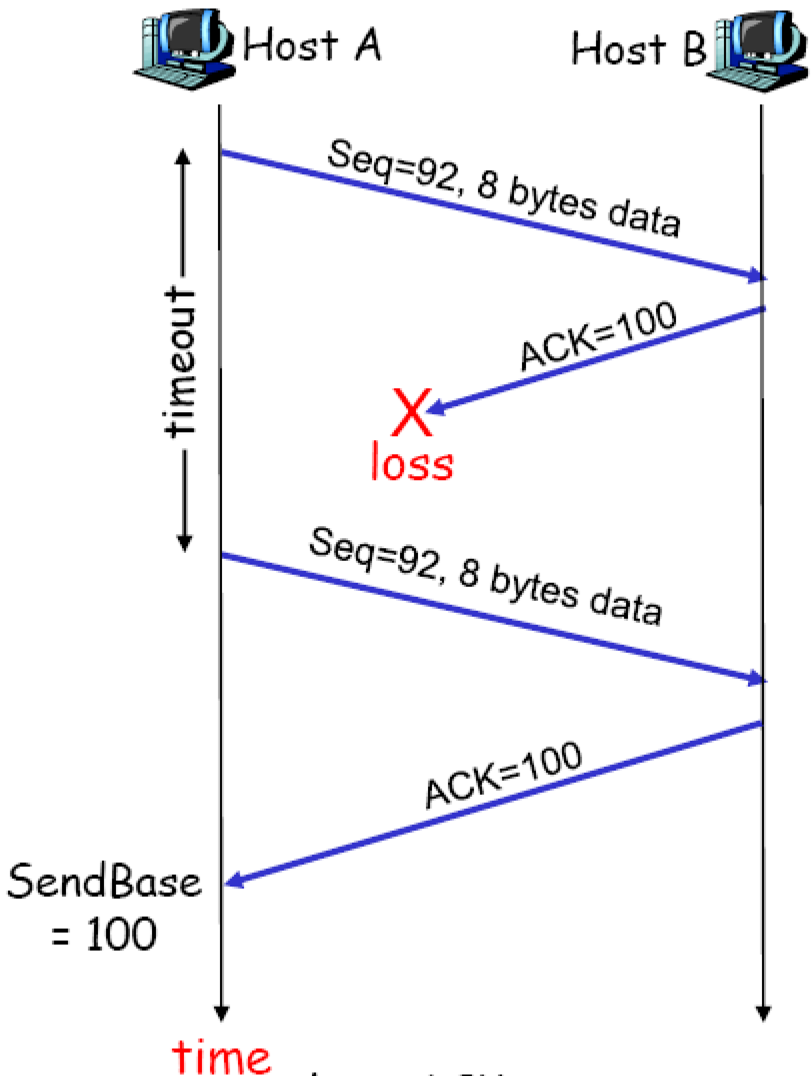


TCP Sender

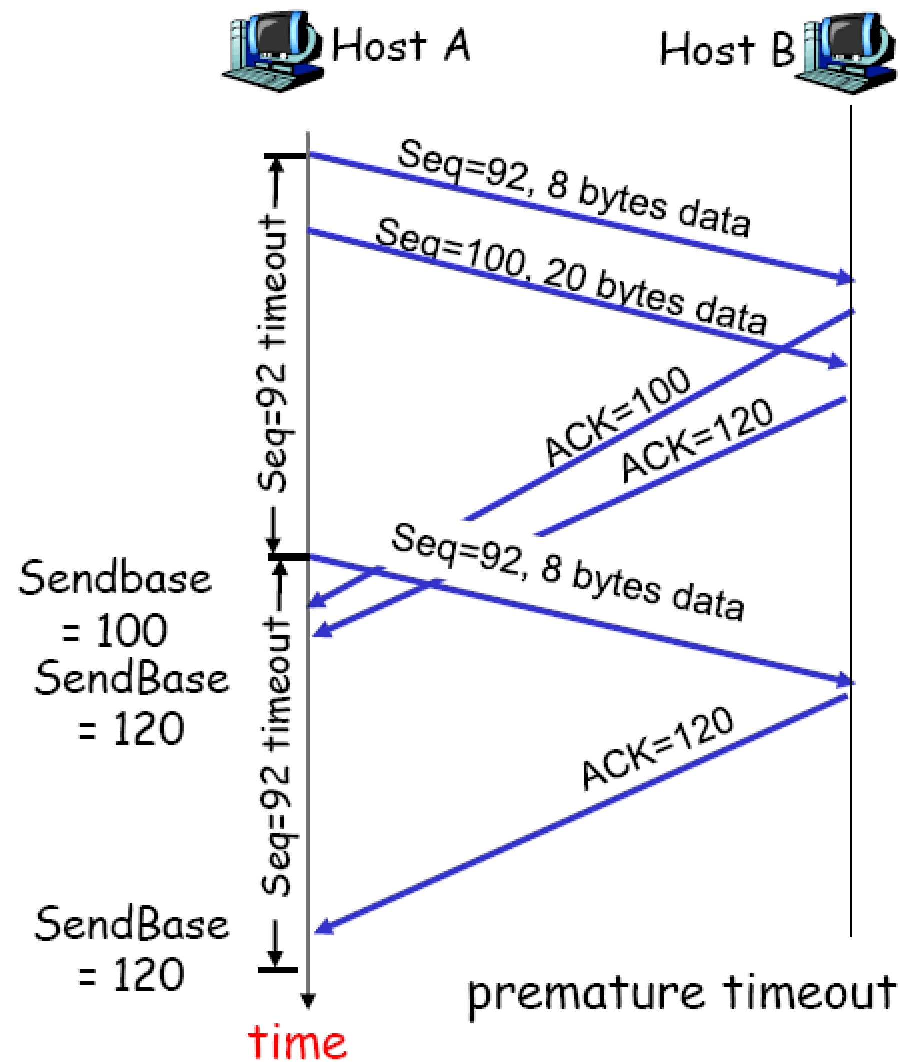



```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

 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 */
```

TCP: retransmission scenarios



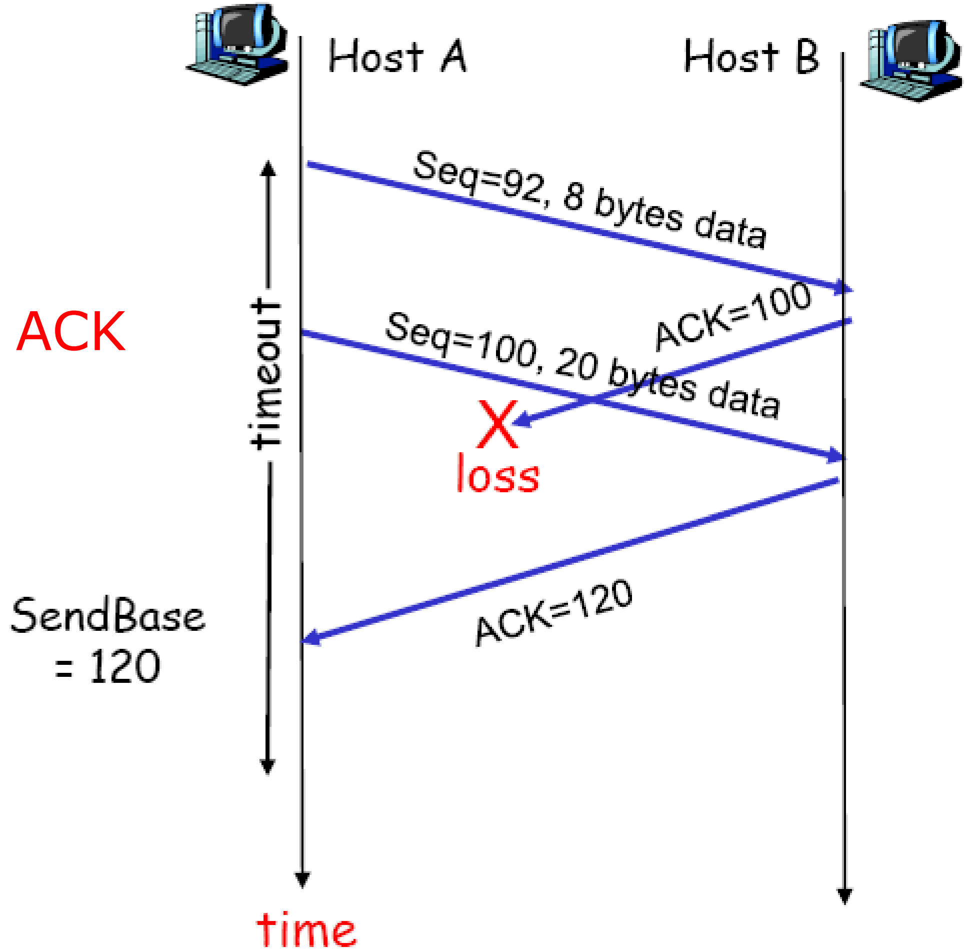
lost ACK scenario



premature timeout

TCP retransmission scenarios

Cumulative ACK



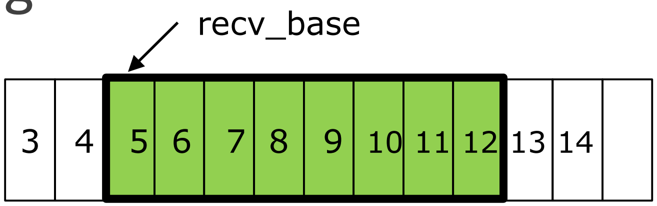
Cumulative ACK scenario

TCP Receiver: ACK Generation

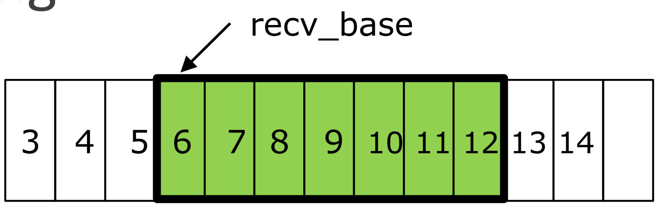
Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately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

TCP Receiver: ACK Generation

- In-order segment with no ACK pending
  - Receive Pkt 5
    - deliver Pkt 5 to appl
    - recv_base ← 6 and delay (ACK 6)

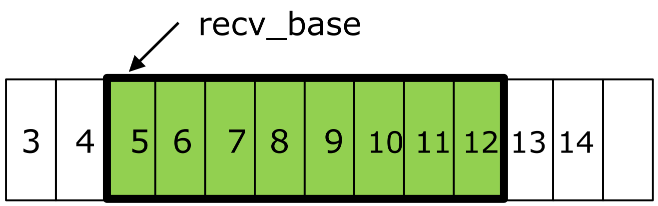


- In-order segment with 1-ACK pending
  - Receive Pkt 6
  - 1 delayed ACK: (ACK 6)
    - deliver Pkt 6 to appl
    - Send ACK 7 and recv_base ← 7

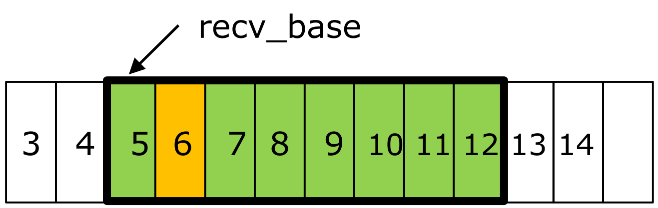


TCP Receiver: ACK Generation

- Out-order segment with higher seq-no than recv_base
  - Receive Pkt 6



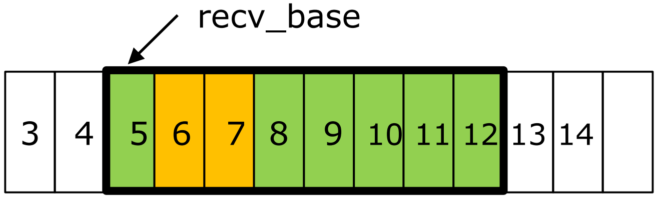
  - Buffer Pkt 6 (mark Pkt 6) and send ACK 5 (dup-ACK)
- Out-order segment with higher seq-no than recv_base
  - Receive Pkt 7



  - Buffer Pkt 7 (mark Pkt 7) and send ACK 5 (dup-ACK)

TCP Receiver: ACK Generation

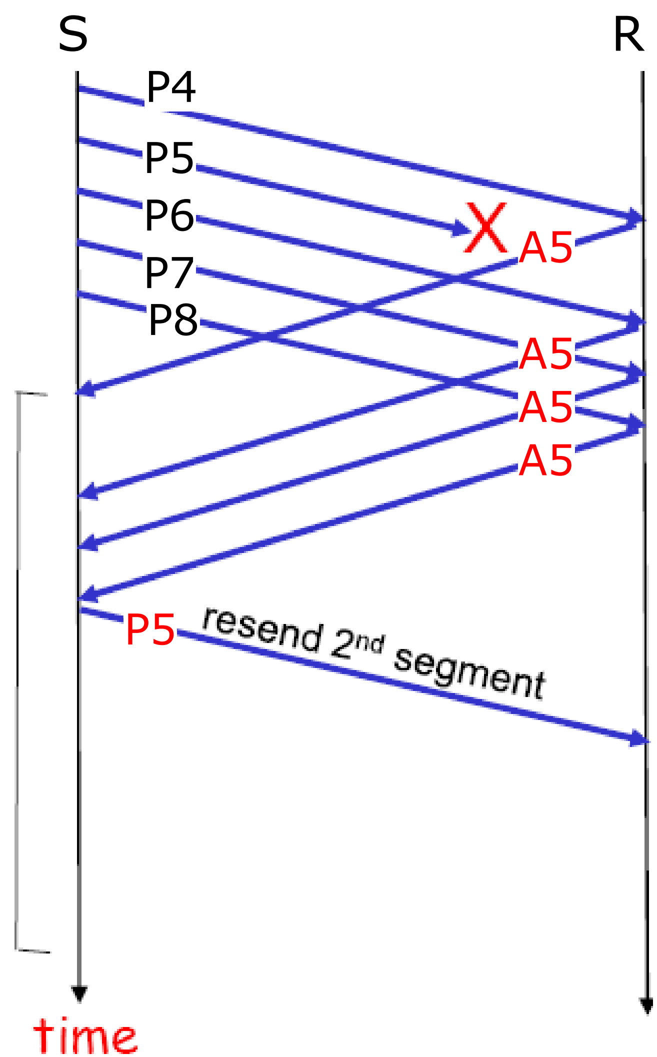
- In-order segment that fills gap
  - Receive Pkt 5



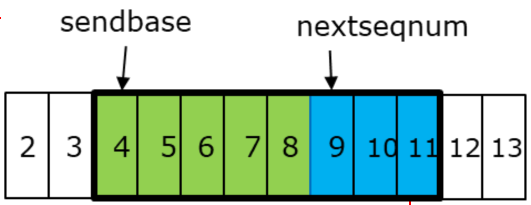
  - Deliver Pkt 5, Pkt 6, and pkt 7 to upper layers
  - send ACK 8 and recv_base ← 8

Fast Retransmit

- Timeout value relatively large: cause long delay before resending lost packet
- Detect segment loss via duplicate ACKs
  - Sender sends packets back-to-back
  - If segment is lost, there will likely be many duplicate ACKs
- If sender receives 3 duplicate ACKs, that segment may be lost with high probability
  - TCP Reno (fast retransmit): resend segment before timer expires



Fast Retransmit Algorithm



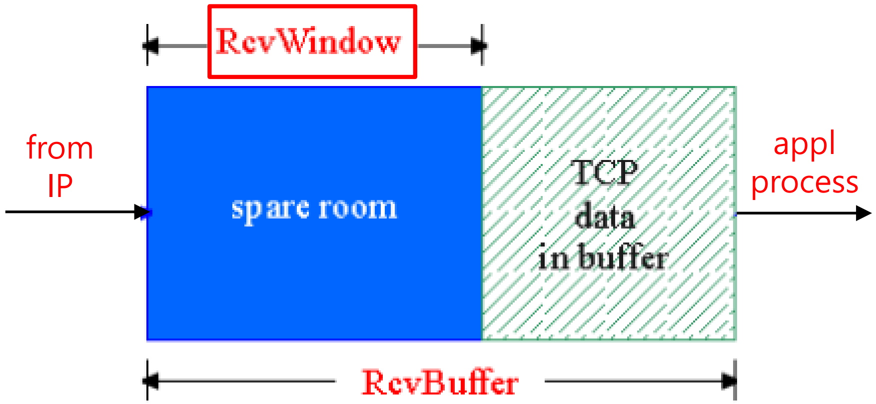

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
            }
          else {
               increment count of dup ACKs received for y
    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

- receiver side buffering: TCP receiver has a receive buffer
- appl process may be slow at reading from buffer

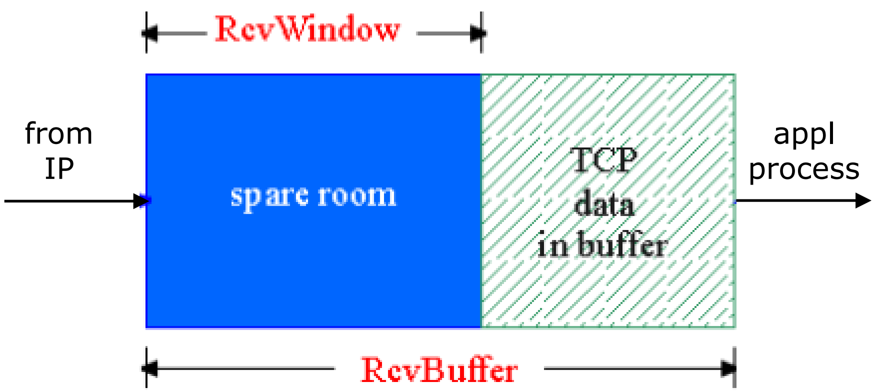


- need to match the speed b/w sending rate and the receiving appl's drain rate
- Flow control
  - sender won't overflow receiver's buffer by transmitting too much, too fast

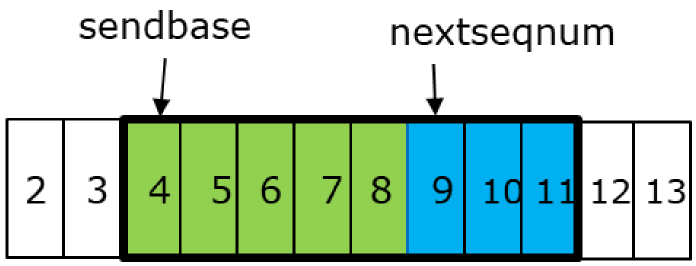
TCP Flow control

- spare room in buffer:

  = `RcvWindow`



- Receiver sends **`RcvWindow`** in segments (receiveWindow in TH): piggybacking



- Sender limits un-ACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow

TCP Connection Management

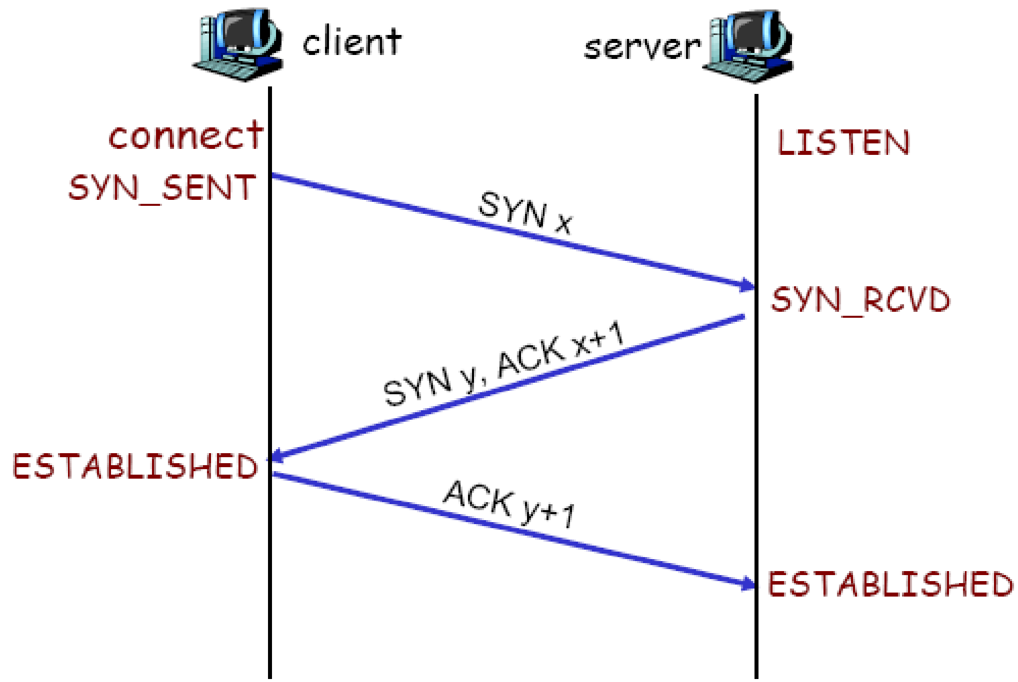
Connection set-up: 3-way handshaking

Step 1: client host sends TCP SYN segment to server

- specifies initial seq # (ISN)
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #



Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management

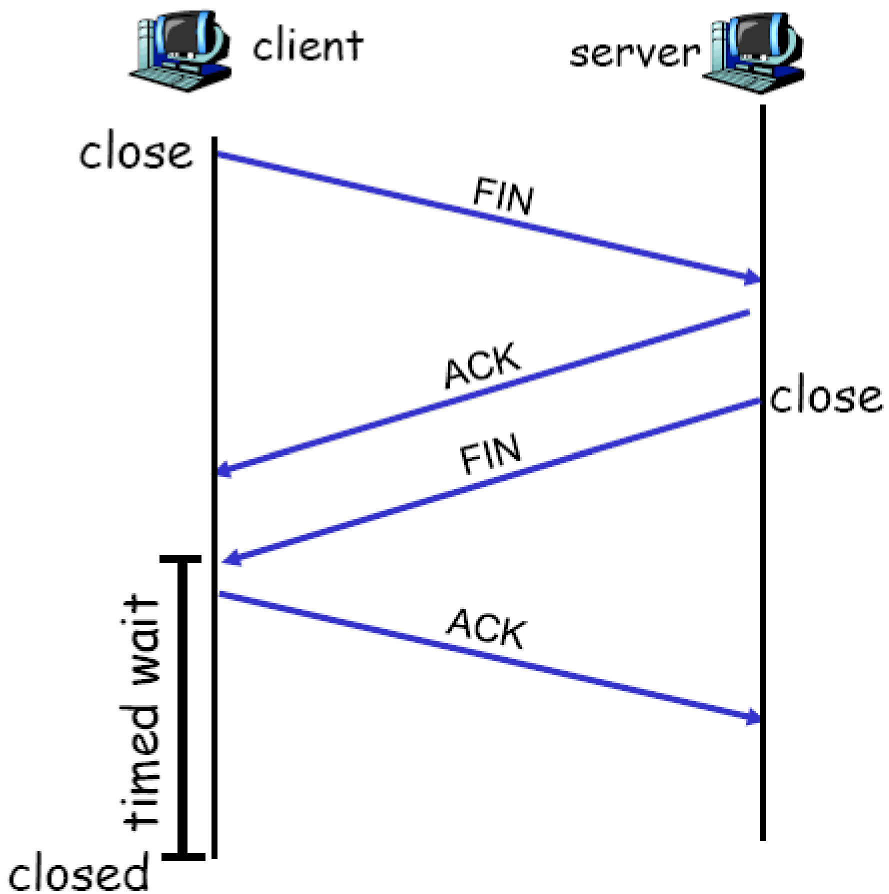
<u>Closing a connection:</u>

client closes socket:

`clientSocket.close();`

<u>Step 1:</u> client end system sends TCP FIN control segment to server

<u>Step 2:</u> server receives FIN, replies with ACK; Closes connection, sends FIN



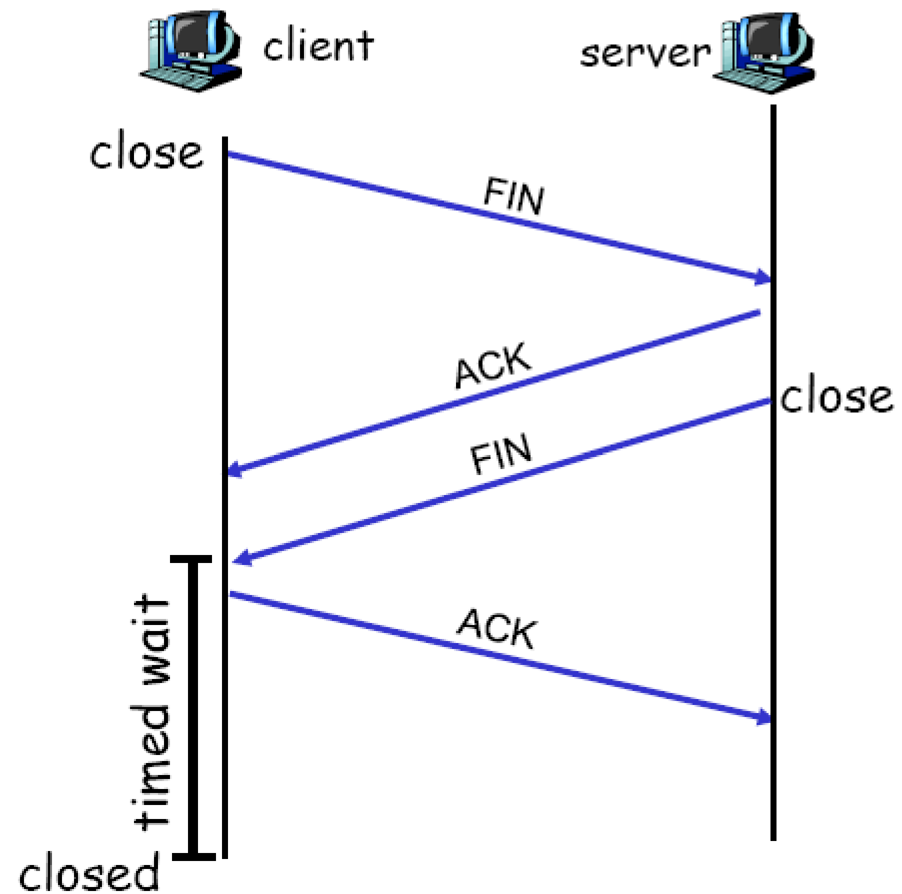
TCP Connection Management

Step 3: client receives FIN, replies with ACK

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

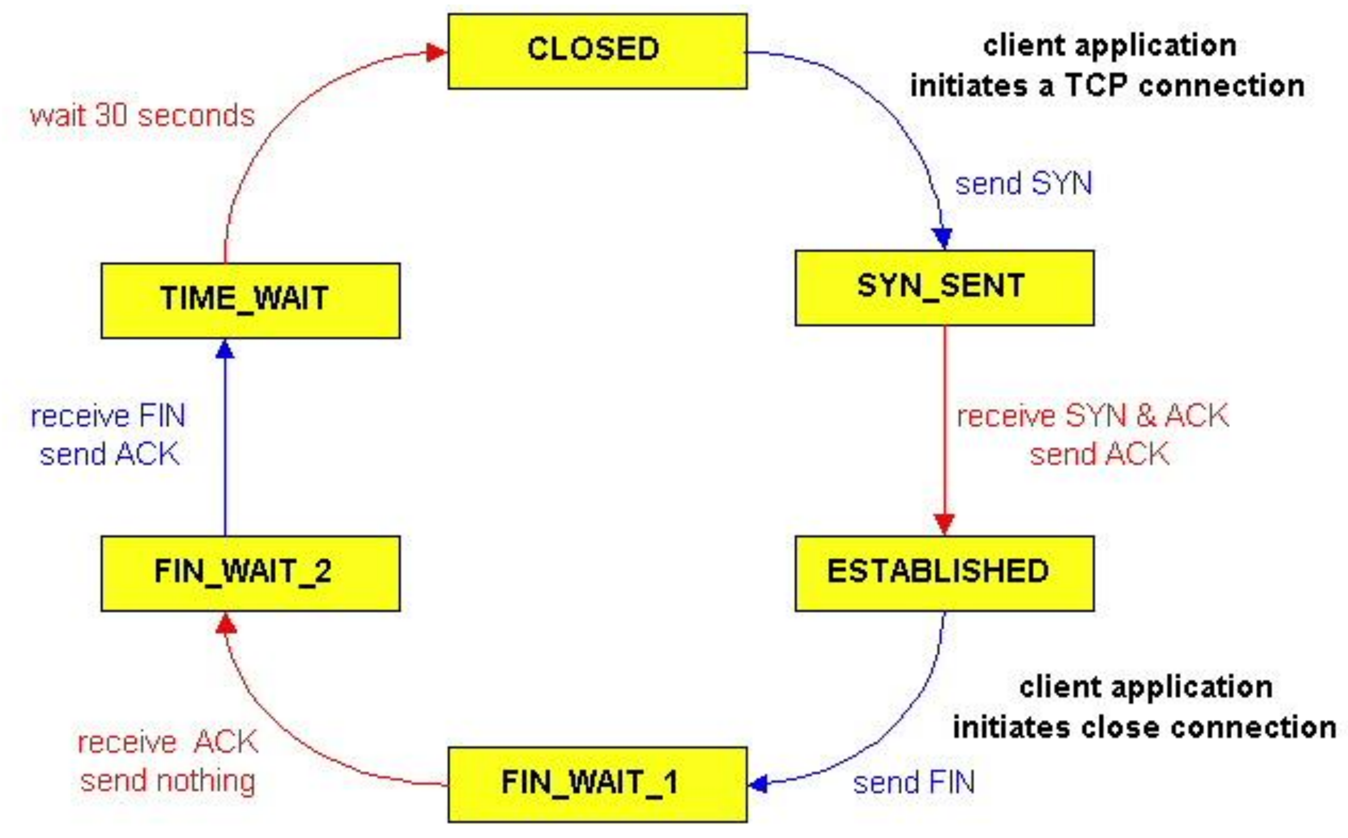
Step 4: server, receives ACK Connection closed

Note: with small modification, can handle simultaneous FINs



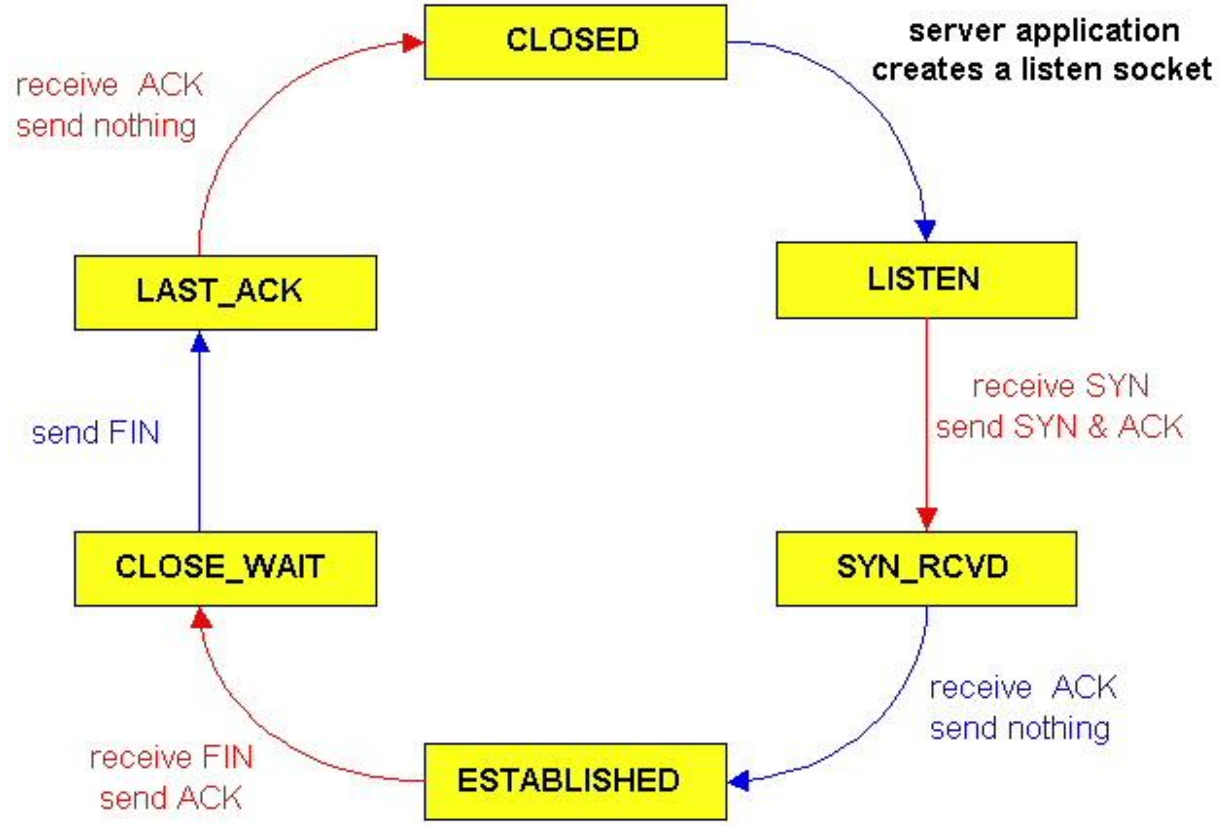
TCP Connection Management

- TCP client life cycle



TCP Connection Management

- TCP server life cycle



Chap 3

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

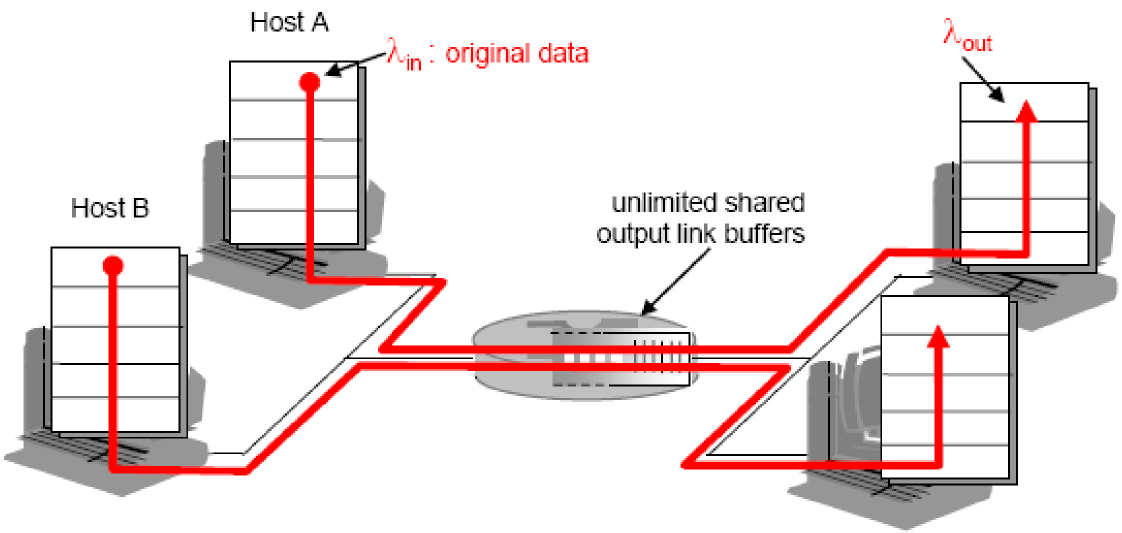
Principles of Congestion Control

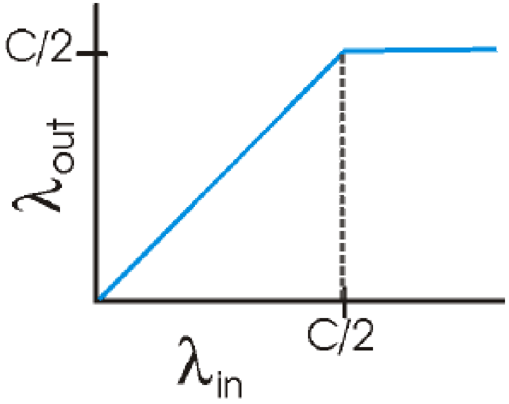
Congestion:

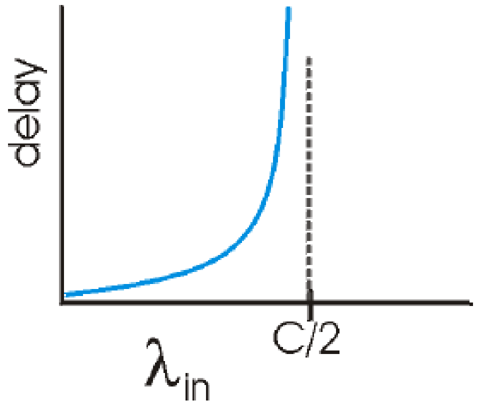
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - long delays (queueing in router buffers)
  - lost packets (buffer overflow at routers)

Congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers, output link capacity C
- no retransmission



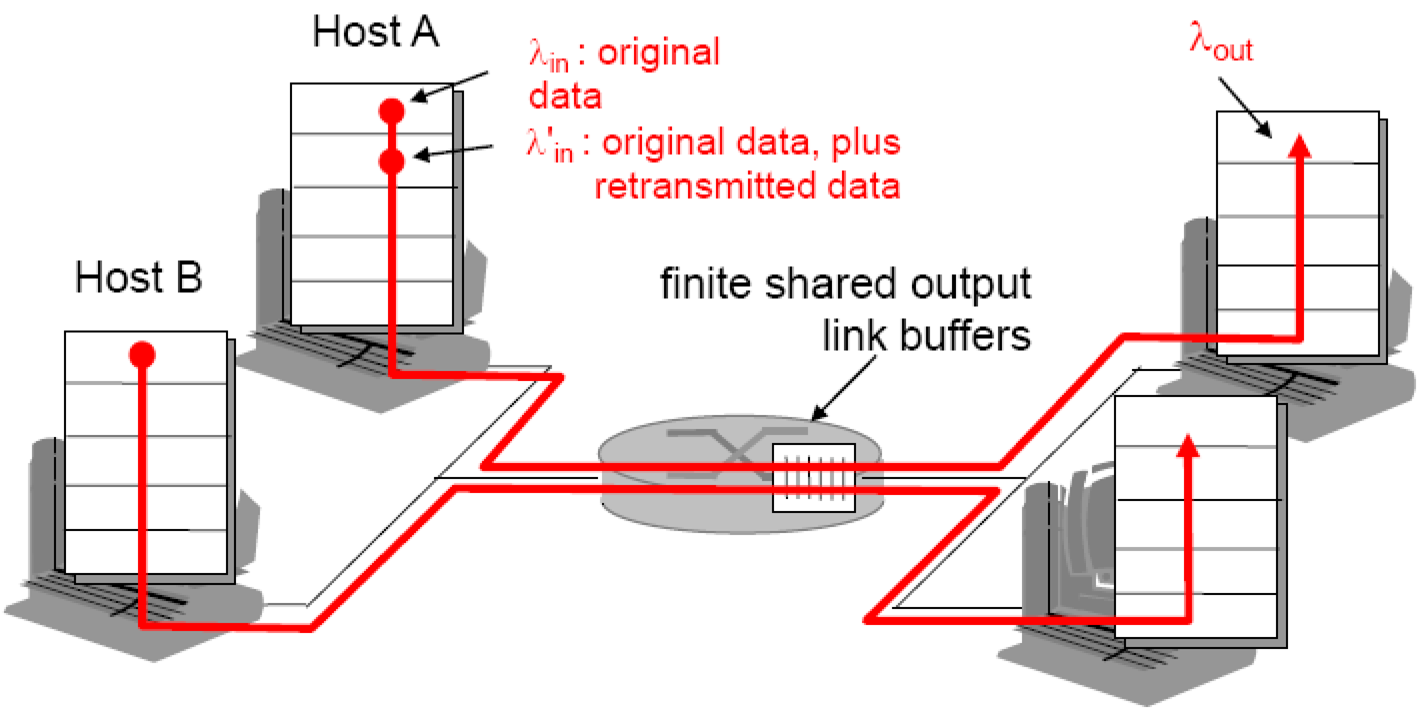




- large delays when congested
- maximum achievable throughput

Congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet

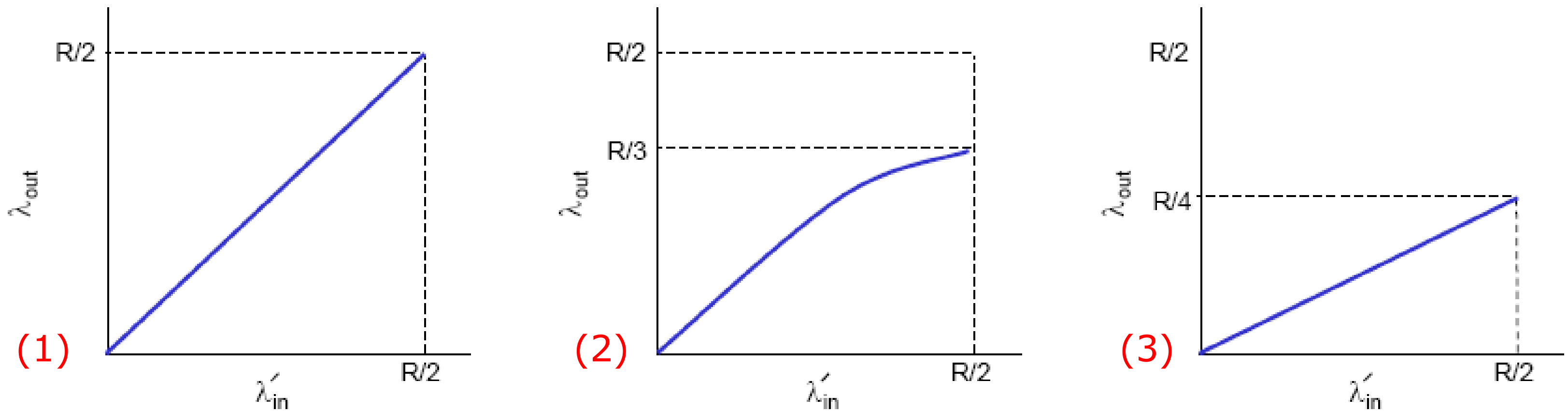


Congestion: scenario 2

(1) When A sends a packet only when a buffer is free: $\boxed{\lambda_{in} = \lambda_{out}}$ (goodput)

(2) "perfect" retransmission only when loss: $\boxed{\lambda'_{in} > \lambda_{out}}$

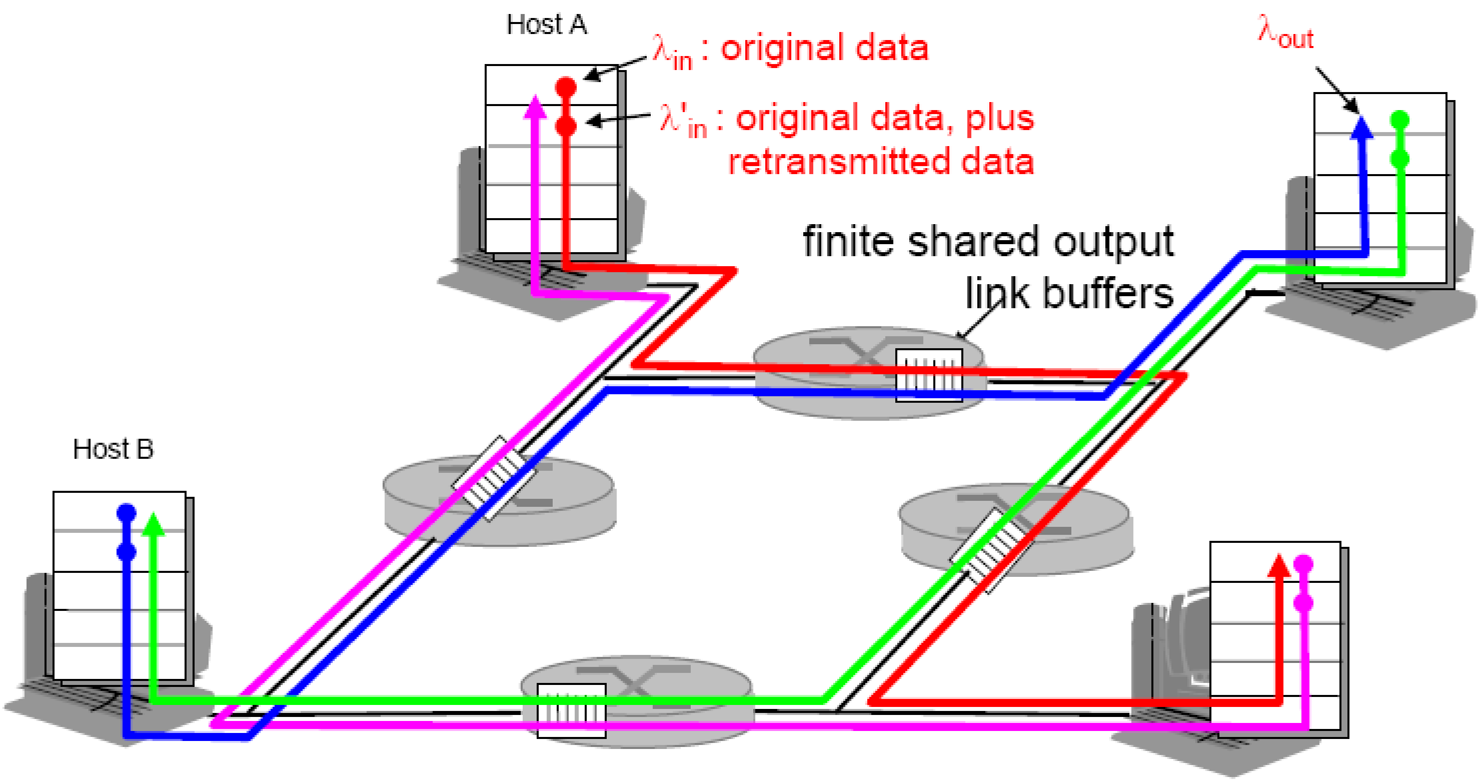
(3) retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



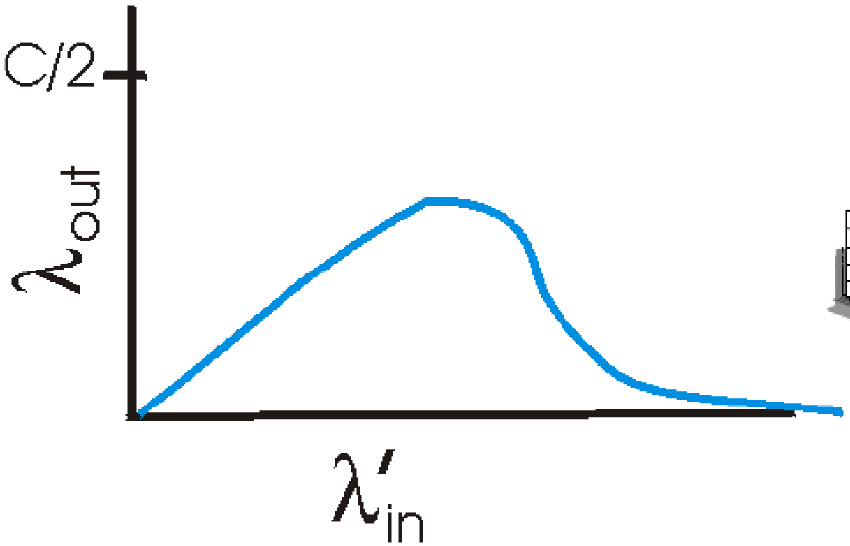
- costs of congestion:
  - more work (retransmission) for given "goodput"
  - unneeded retransmissions: link carries multiple copies of pkt

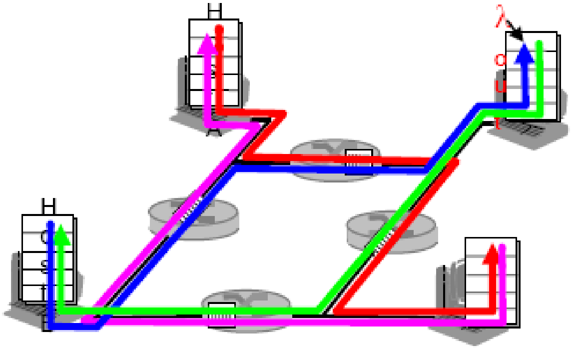
Congestion: scenario 3

- four senders, multihop paths
- timeout/retransmit
- Q: what happens as λ_{in} and λ'_{in} increase ?



Congestion: scenario 3





- Another "cost" of congestion:
  - when packet dropped, any "upstream transmission capacity used for that packet was wasted

Approaches towards congestion control

Network-assisted congestion control:

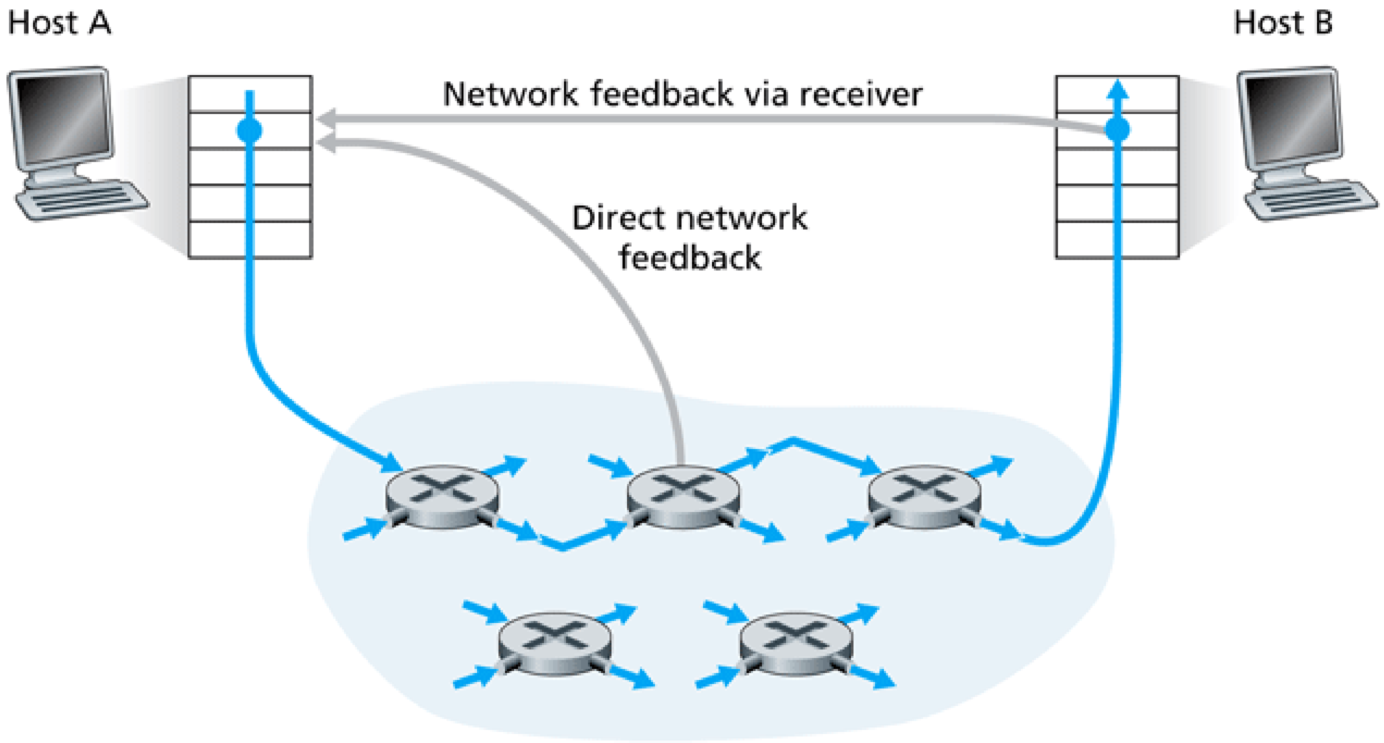
- routers provide feedback to end systems
  - single bit indicating congestion (TCP/IP ECN, ATM)
  - explicit rate sender should send at

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Approaches towards congestion control

Network-assisted congestion control:



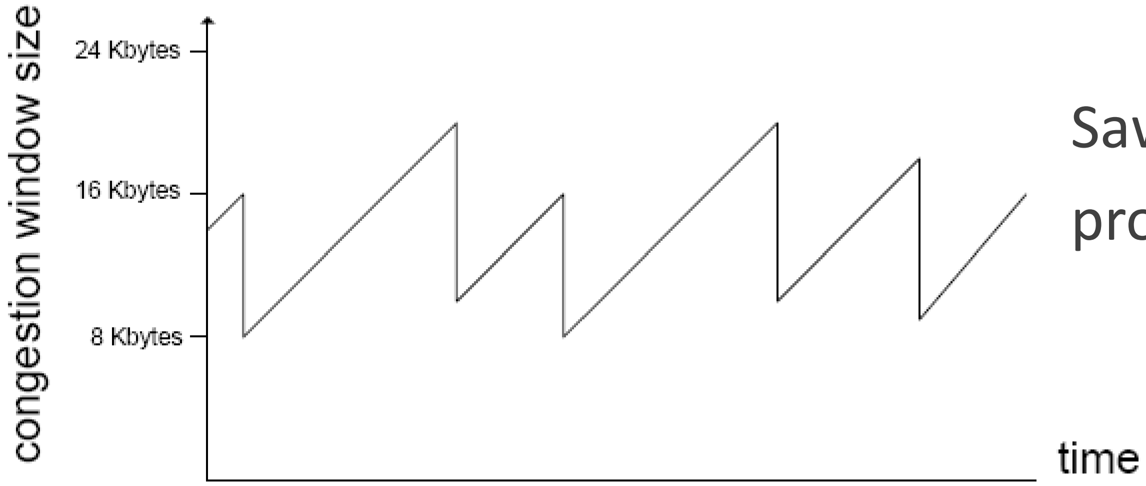
Chap 3

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

TCP Congestion Control

Additive increase, multiplicative decrease (AIMD)

- *Approach:* increase transmission rate (congestion window size: **cwnd**), 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



Saw tooth behavior: probing for bandwidth

TCP Congestion Control

- sender limits transmission:

  `(LastByteSent - LastByteAcked) ≤ cwnd`

- Roughly,

$$\boxed{\text{rate} = \frac{\text{cwnd}}{\text{RTT}} \text{ Bytes/sec}}$$

- **`cwnd`** is dynamic, indicates how much we can transmit before congestion

<u>How does sender perceive congestion?</u>

- loss event : timeout *or* 3 duplicate ACKs
- TCP sender reduces rate (**`cwnd`**) after loss event

<u>three mechanisms:</u> AIMD

  - slow start
  - congestion avoidance
  - congestion control

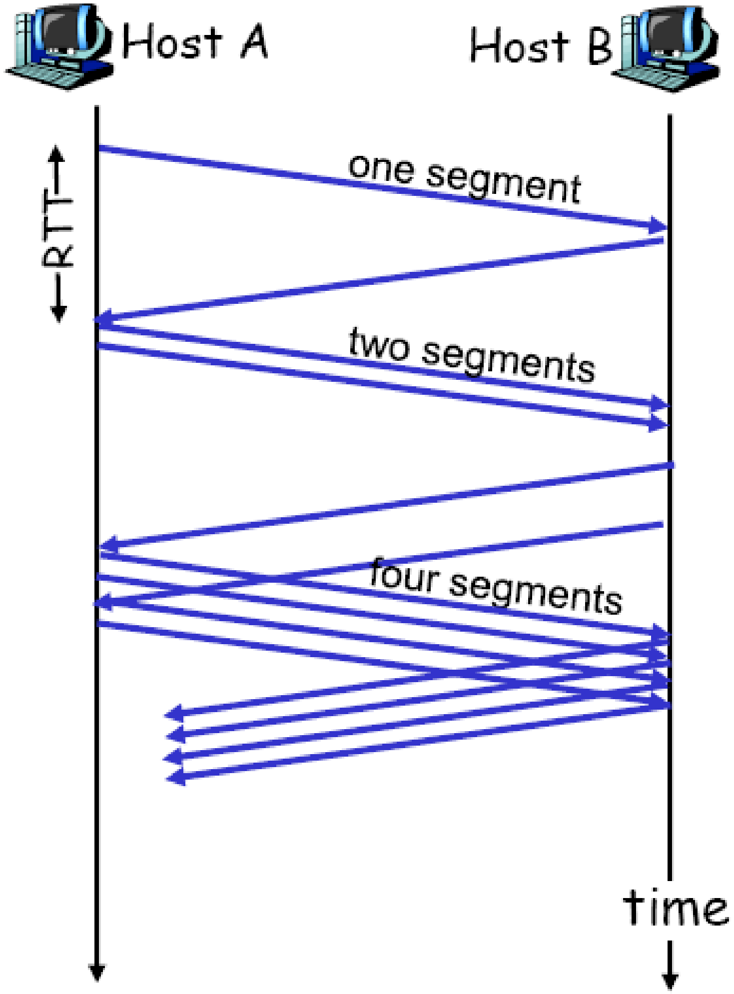
TCP Slow Start

- When connection begins, **`cwnd`** = 1*MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to available rate

- When connection begins, increase rate exponentially fast until first loss event

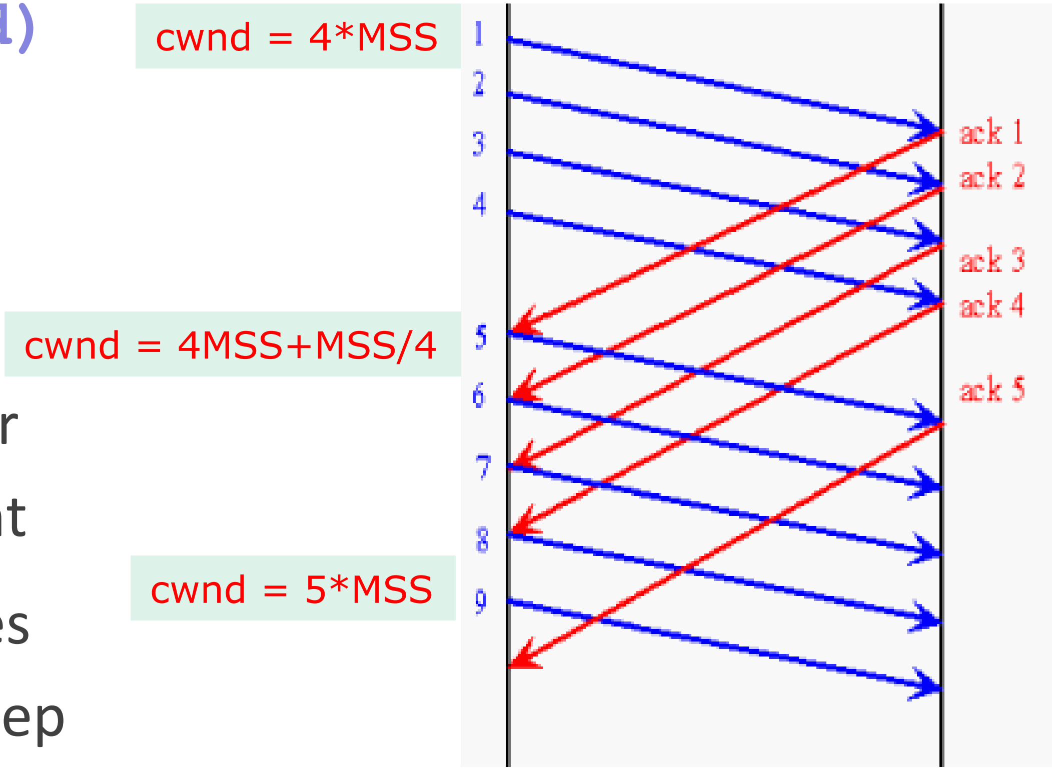
TCP Slow Start

- slow start: increase rate exponentially until loss event
  - increments **cwnd** by (1*MSS) for each ACK received until threshold (ssthreshold)
  - roughly double **cwnd** for each RTT
- initial rate is slow but ramps up exponentially fast



TCP Congestion Avoidance

- congestion avoidance: when (**cwnd** ≥ **ssthreshold**)
  - increments **cwnd** by MSS*(MSS/**cwnd**) for each ACK received
  - increase rate linearly for each RTT until loss event
  - when loss happens, goes to congestion control step



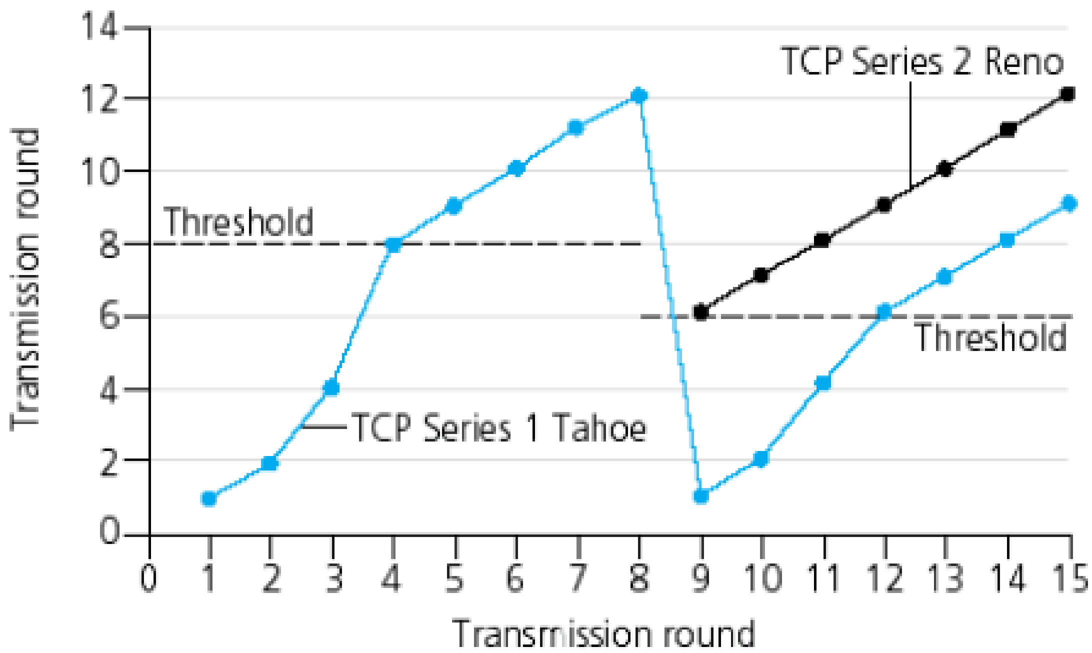
TCP Congestion Control

- loss: 3 dup ACKs
  - `ssthresold = cwnd/2`
  - `cwnd = ssthresold`
  - goes to congestion avoidance: window then grows linearly
- loss: timeout event
  - `ssthresold = cwnd/2`
  - `cwnd = 1*MSS`
  - goes to slow start

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

TCP Congestion Control



<u>Implementation:</u>

- Variable Threshold
- At loss event, Threshold is set to 1/2 of **cwnd** just before loss event

Summary: TCP Congestion Control

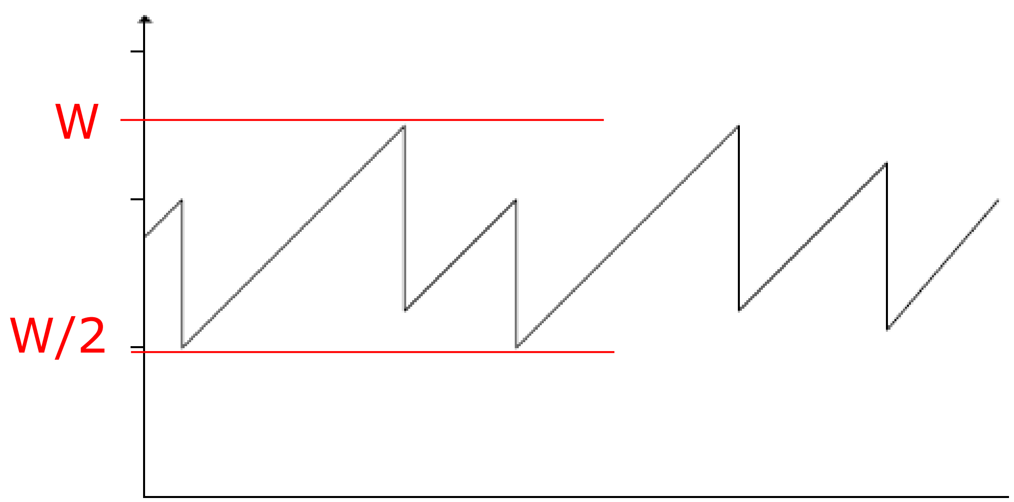
- When **cwnd** is below **Threshold**, sender in slow-start phase, window grows exponentially
- When **cwnd** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly
- When a 3 duplicate ACK occurs, **Threshold** set to **cwnd/2** and **cwnd** set to **Threshold**
- When timeout occurs, **Threshold** set to **cwnd/2** and **cwnd** is set to 1 MSS

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

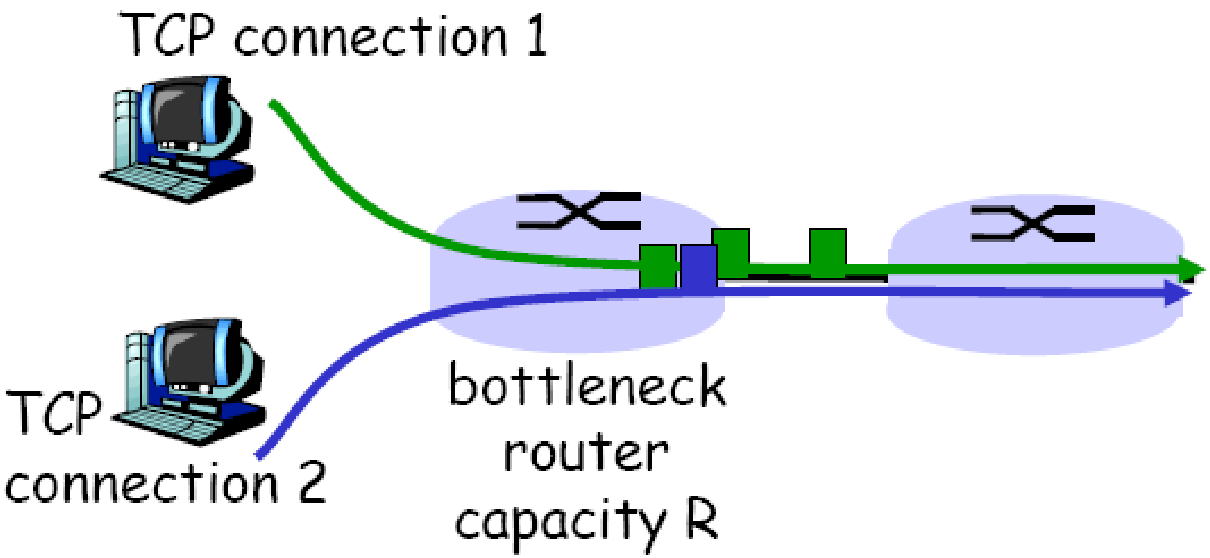
TCP Throughput

- average throughout of TCP as a function of window size and RTT?
  - Let W be the window size when loss occurs
  - When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT
- Average throughout: 0.75*(W/RTT)



TCP Fairness

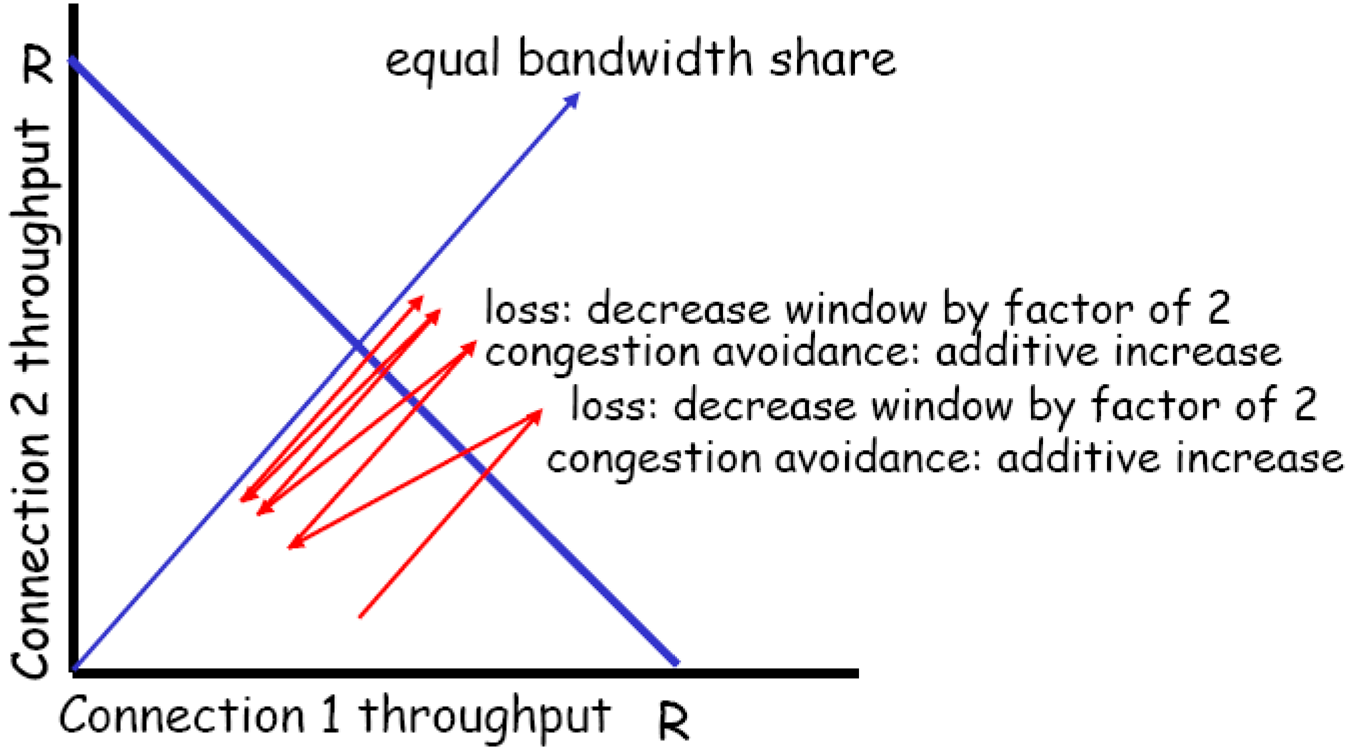
- **Fairness goal:** if K TCP sessions share a bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Fairness more

Fairness and parallel TCP connections

- nothing prevents app from opening parallel connections between 2 hosts
- Web browsers use parallel connections
- Example: link of rate C supporting 9 applications
  - 8 appl's ask for 1 TCP conn and 1 appl asks for 2 conn
  - 8 appl's with 1 TCP conn, gets rate C/10 for each appl
  - one appl with 2 TCP conns, gets C/5