Transport Layer: Reliable Data Transfer

October 8, 2024

Min Suk Kang
Associate Professor
School of Computing/Graduate School of Information Security

NetS&P Research Lab @ KAIST

KAIST

Chapter 3: roadmap

- 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
- Evolution of transport-layer functionality

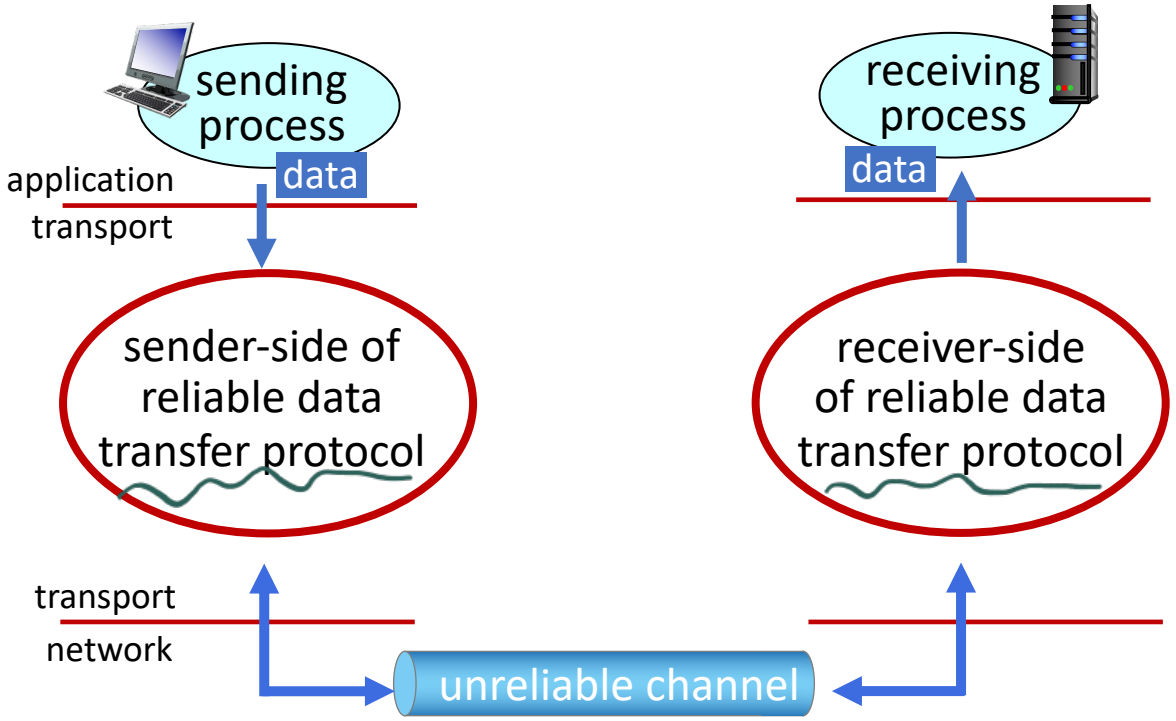
Principles of reliable data transfer

reliable service *abstraction*

추상화

Principles of reliable data transfer

reliable service *abstraction*

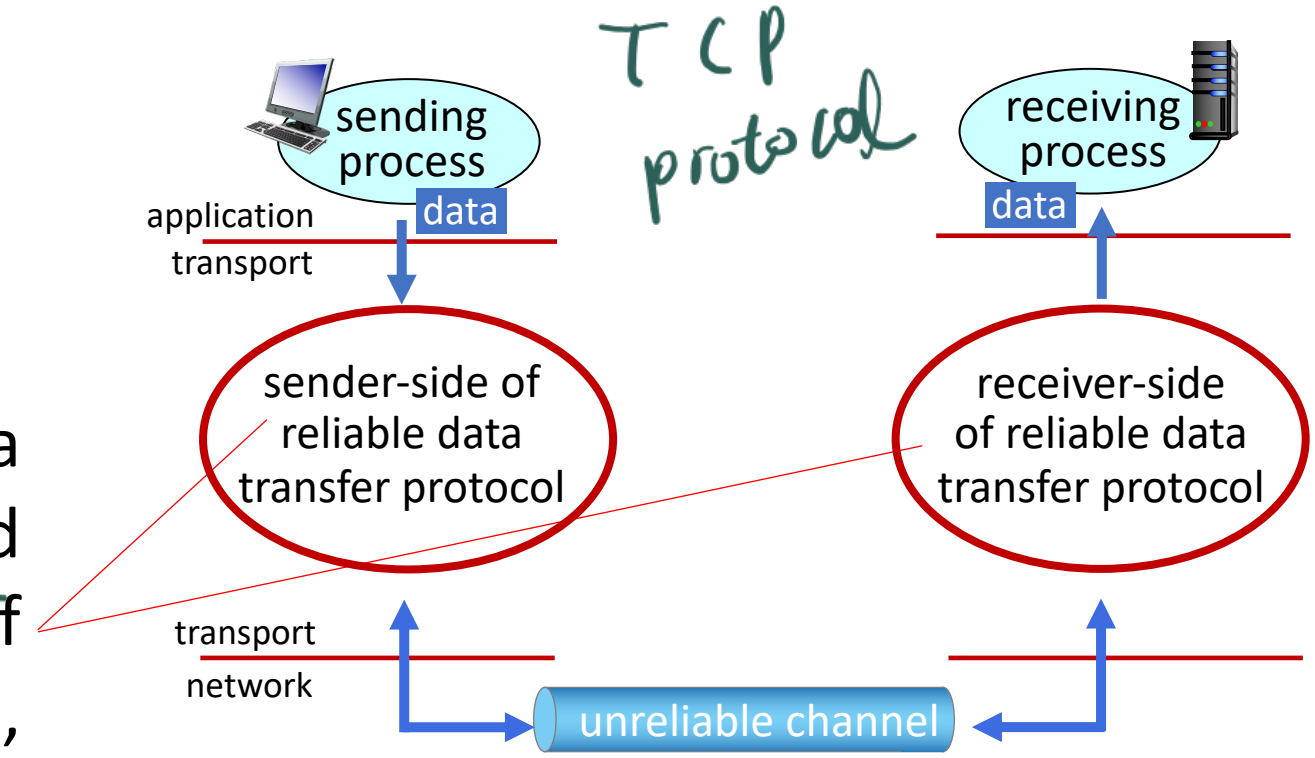


reliable service *implementation*

구현

Principles of reliable data transfer

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)

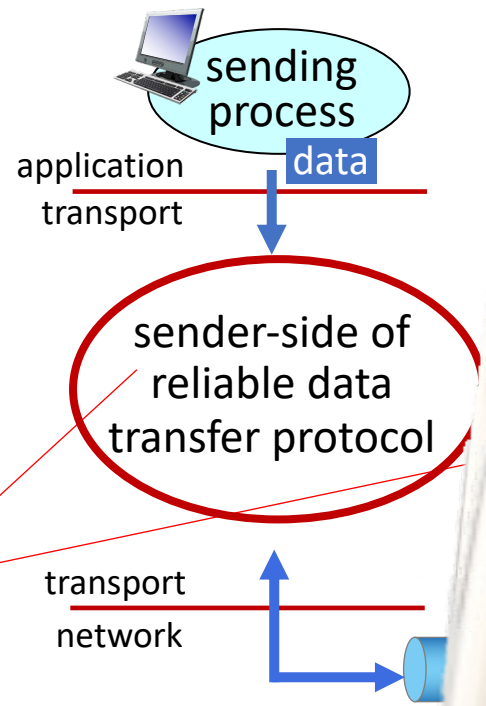


reliable service *implementation*

Principles of reliable data transfer

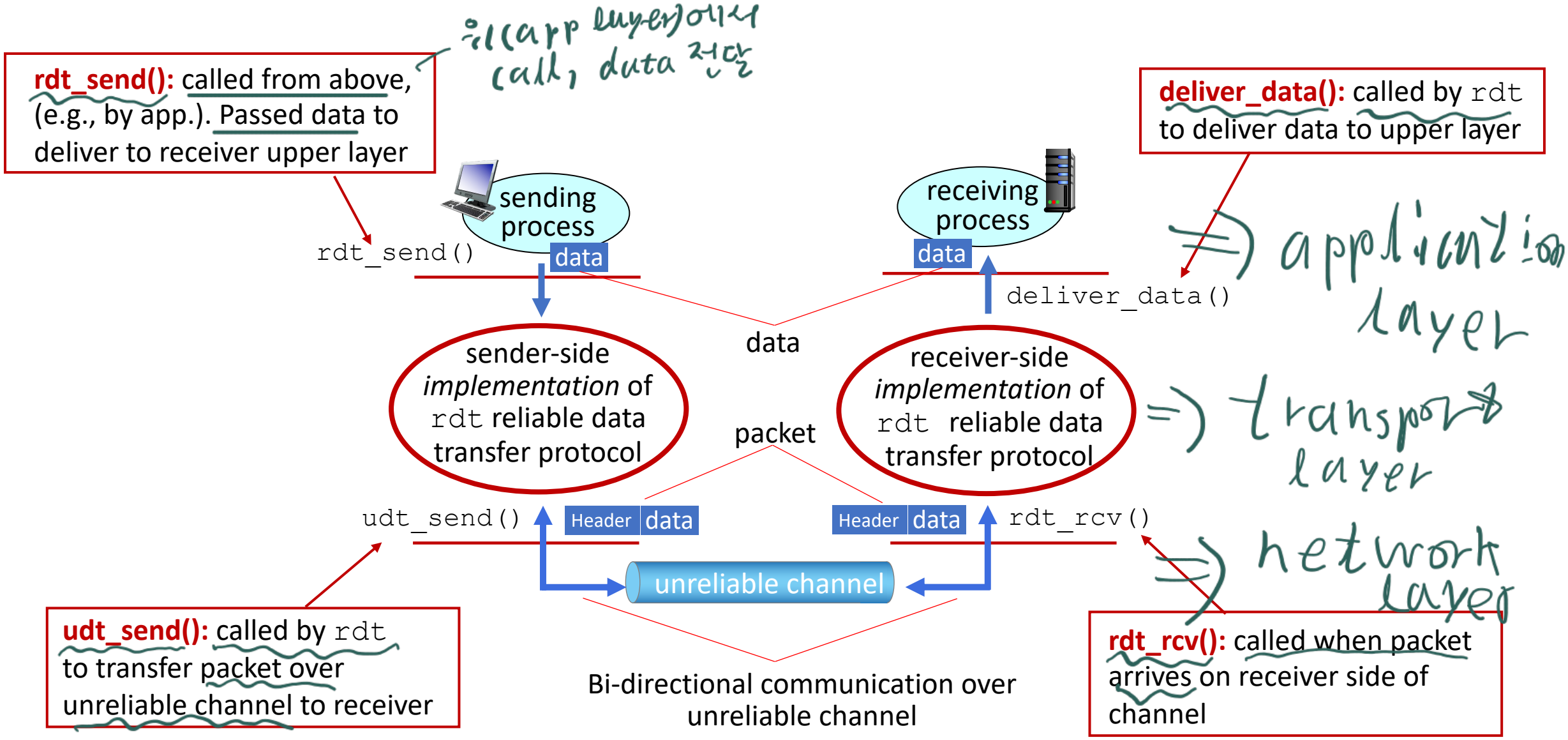
Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message



reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces



Reliable data transfer: getting started

reliable protocol in non-reliable channel

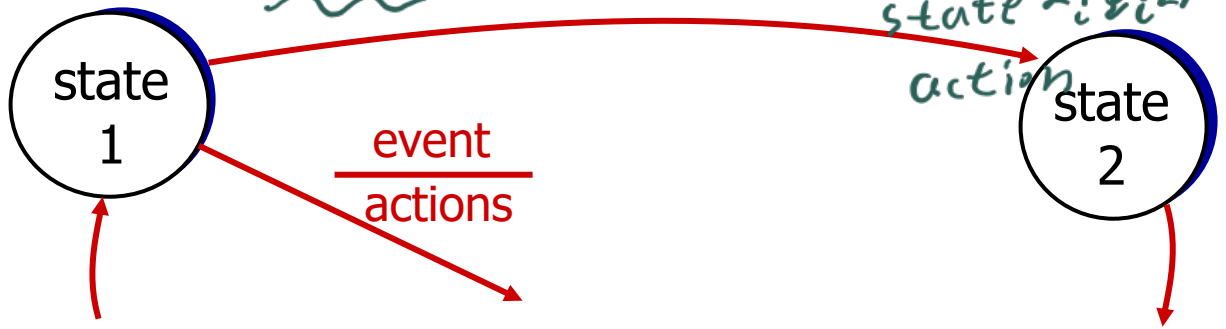
We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- consider only unidirectional data transfer ⇒ 단방향
  - but control info will flow in both directions!
- use finite state machines (FSM) to specify sender, receiver

event causing state transition → state 전환

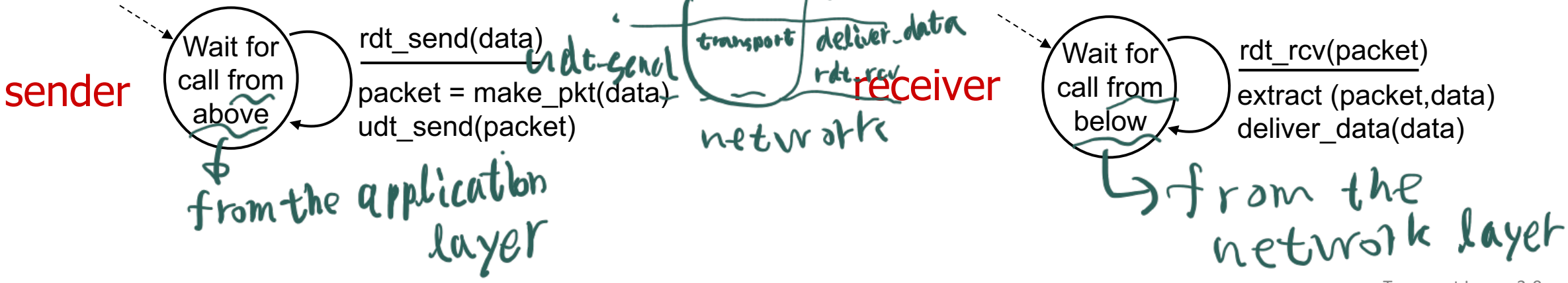
actions taken on state transition

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



rdt1.0: reliable transfer over a reliable channel

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



rdt2.0: channel with bit errors

↳ bit error 처리

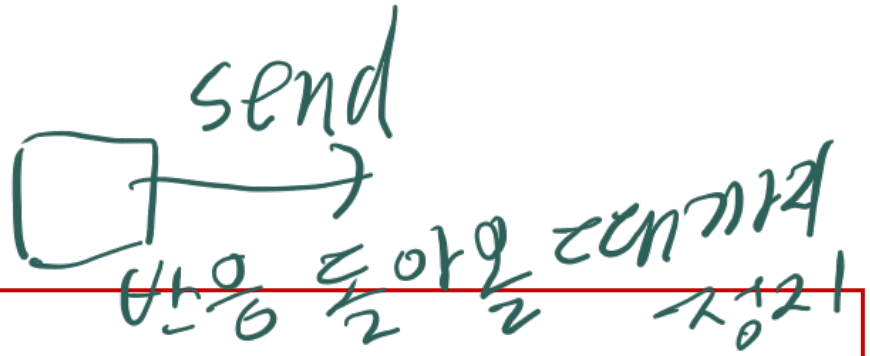
- assumption: underlying channel may flip bits in packet
  - checksum (e.g., Internet checksum) to detect bit errors
  - no loss of packets
- *the* question: how to recover from errors?

How do humans recover from "errors" during conversation?

↳ recall

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 → 성공 시 응답
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors → 실패 시 응답
  - sender *retransmits* pkt on receipt of NAK

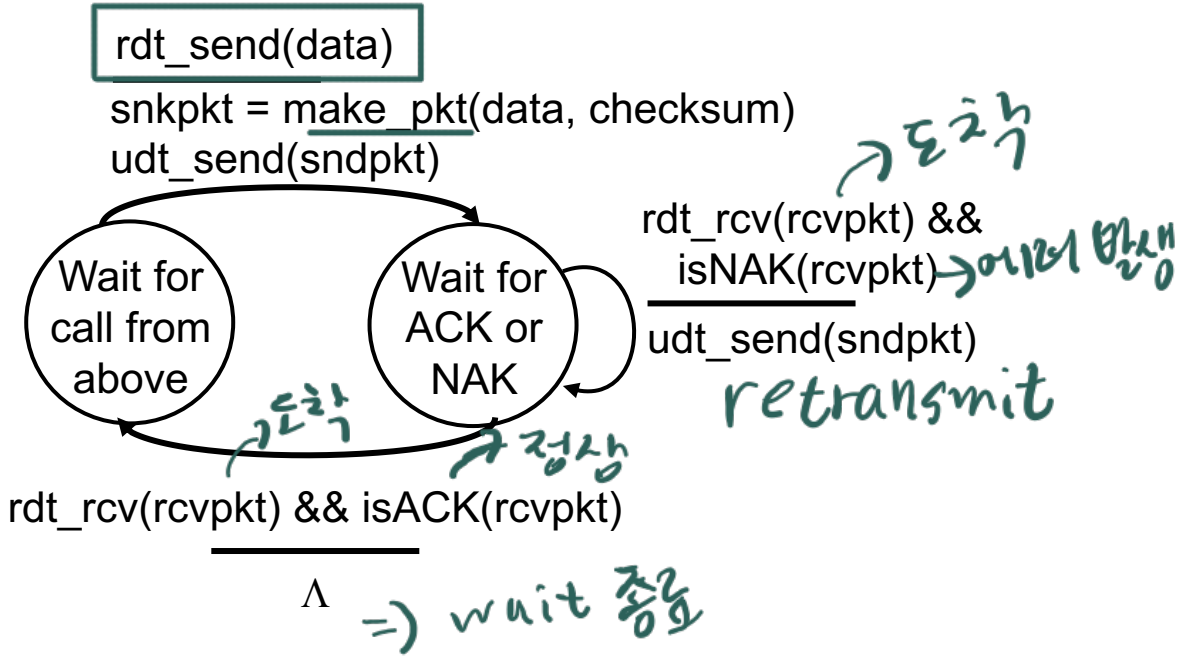


stop and wait

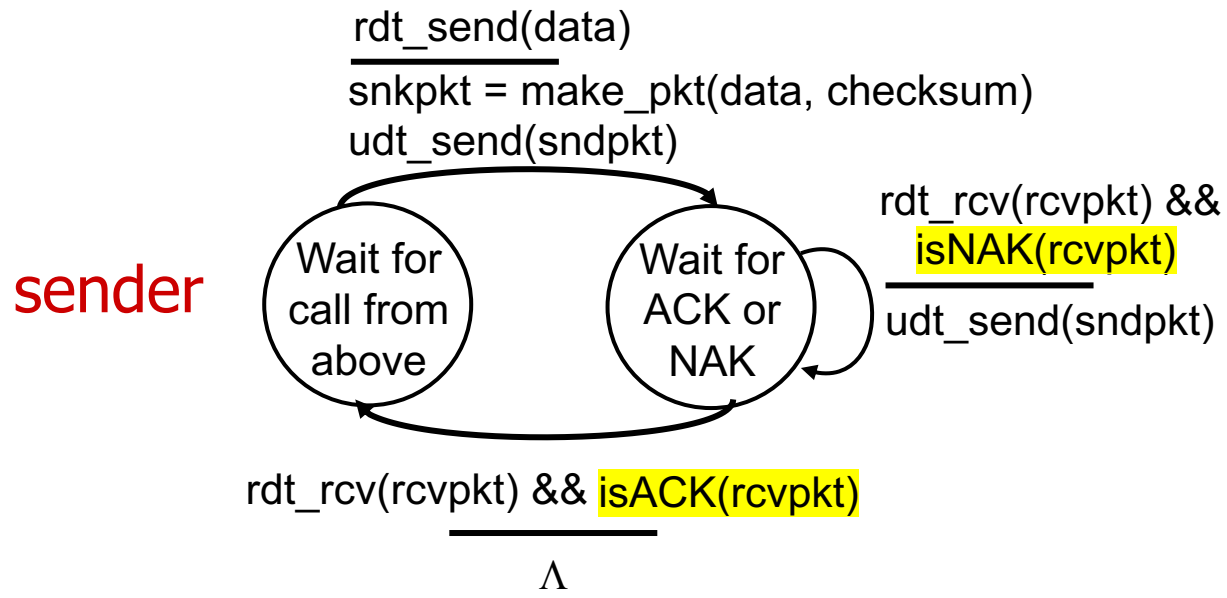
sender sends one packet, then waits for receiver response

rdt2.0: FSM specifications

sender



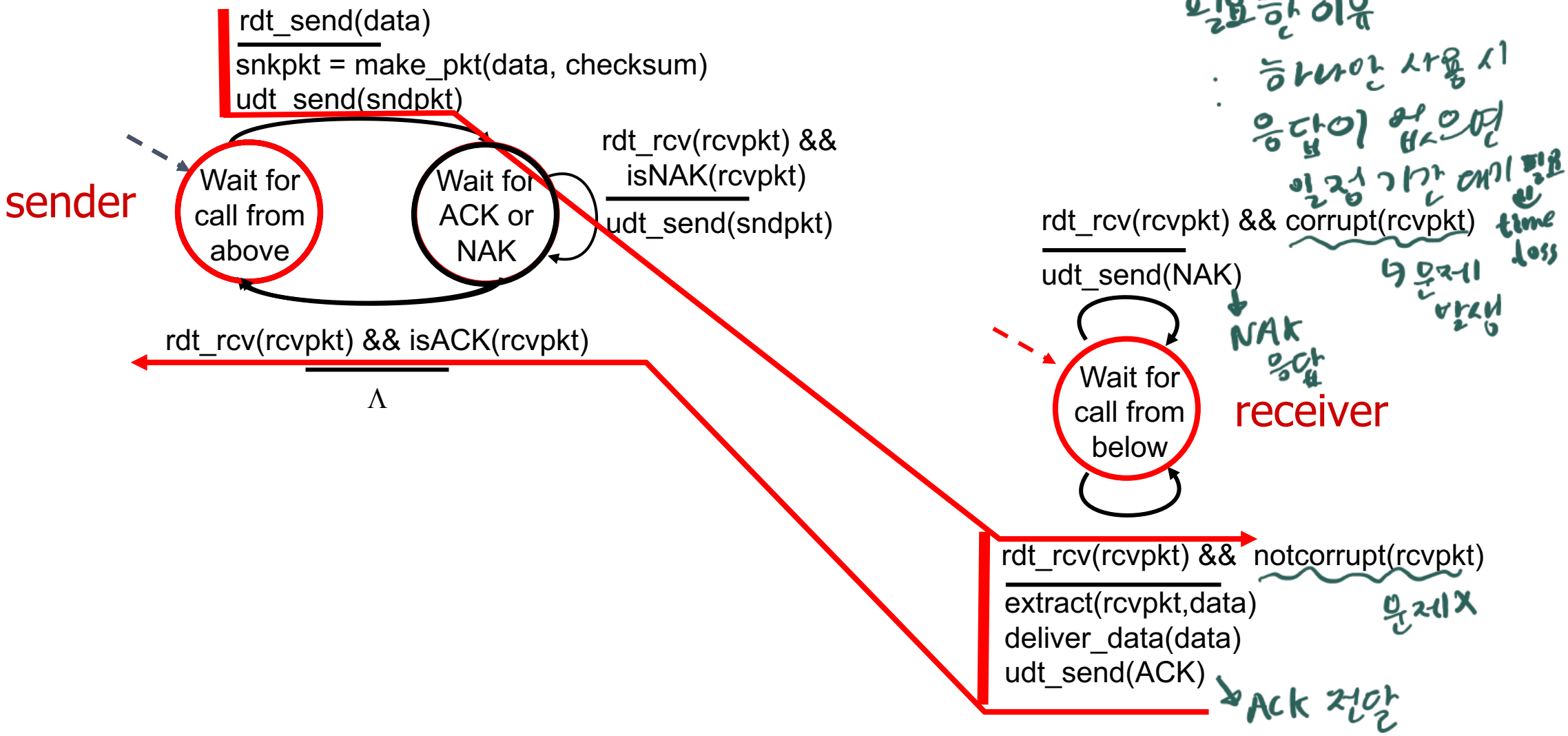
rdt2.0: FSM specification



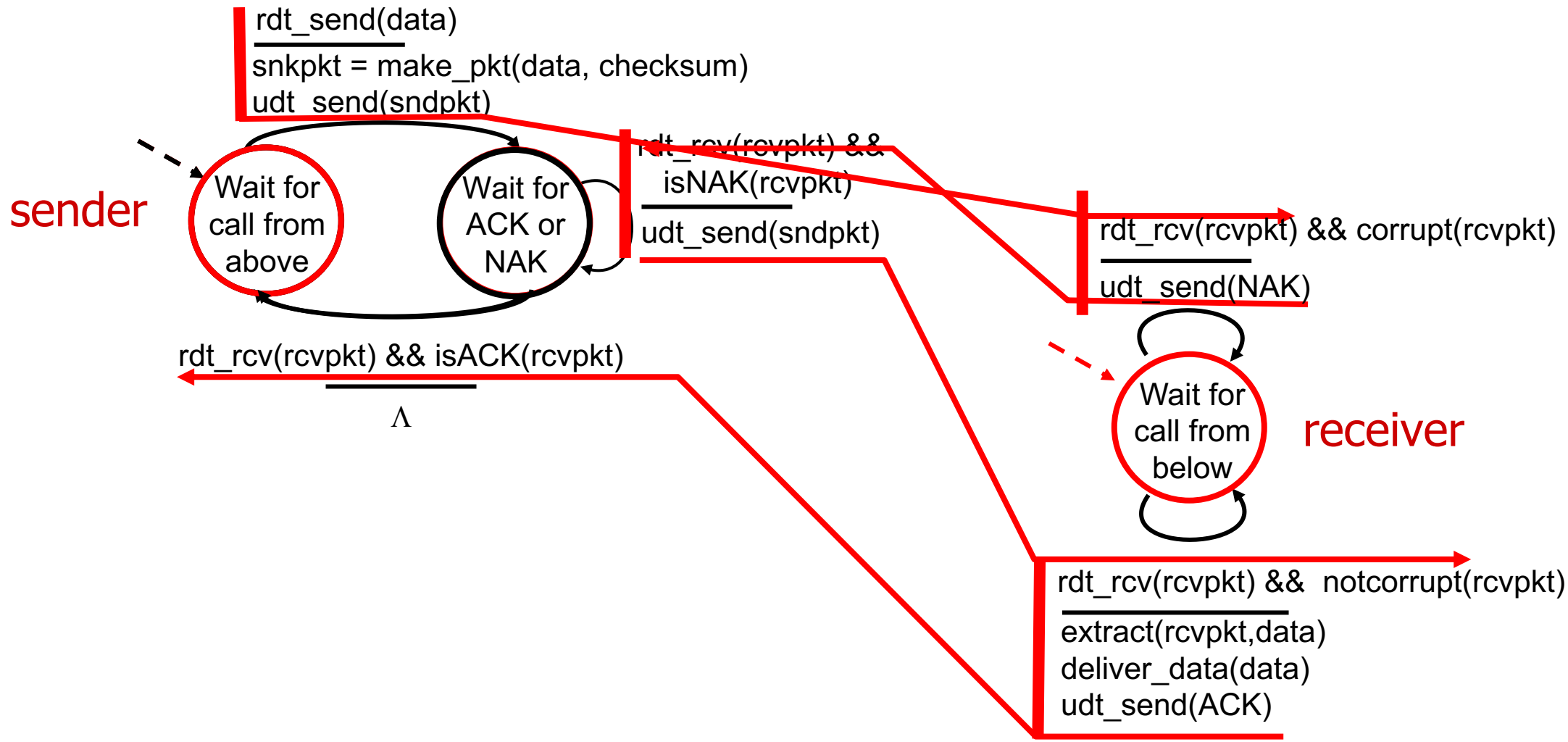
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



rdt2.0 has a fatal flaw!

치명적인 결함

retransmit 하면 중복 발생 가능

ACK/NAK corrupted?

⇒ sender는 state를 알지 못하므로 retransmit 한다면 duplication 발생 가능

handling duplicates:

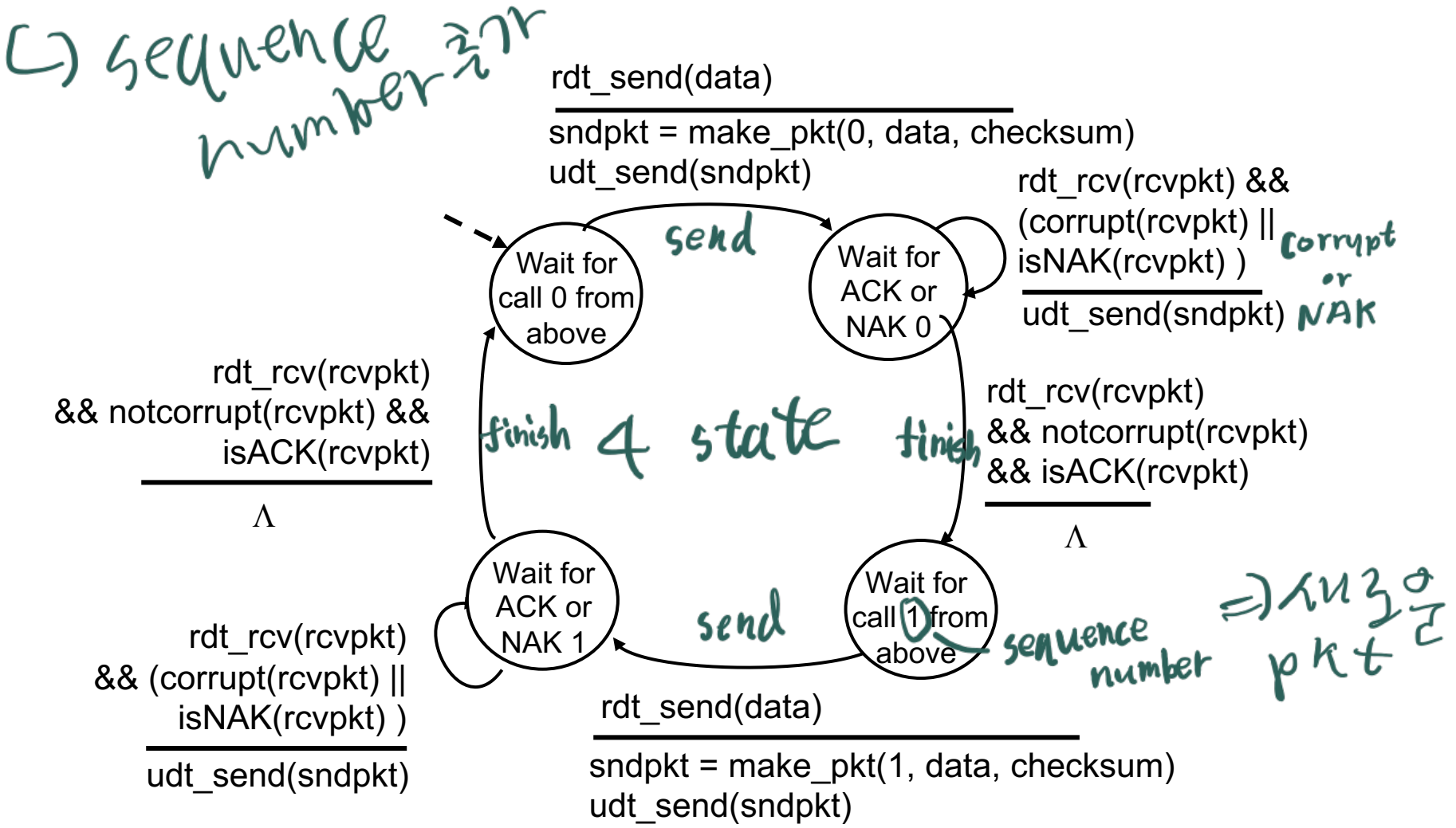
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

↳ 중복 처리 (retransmit 여부 확인 가능)

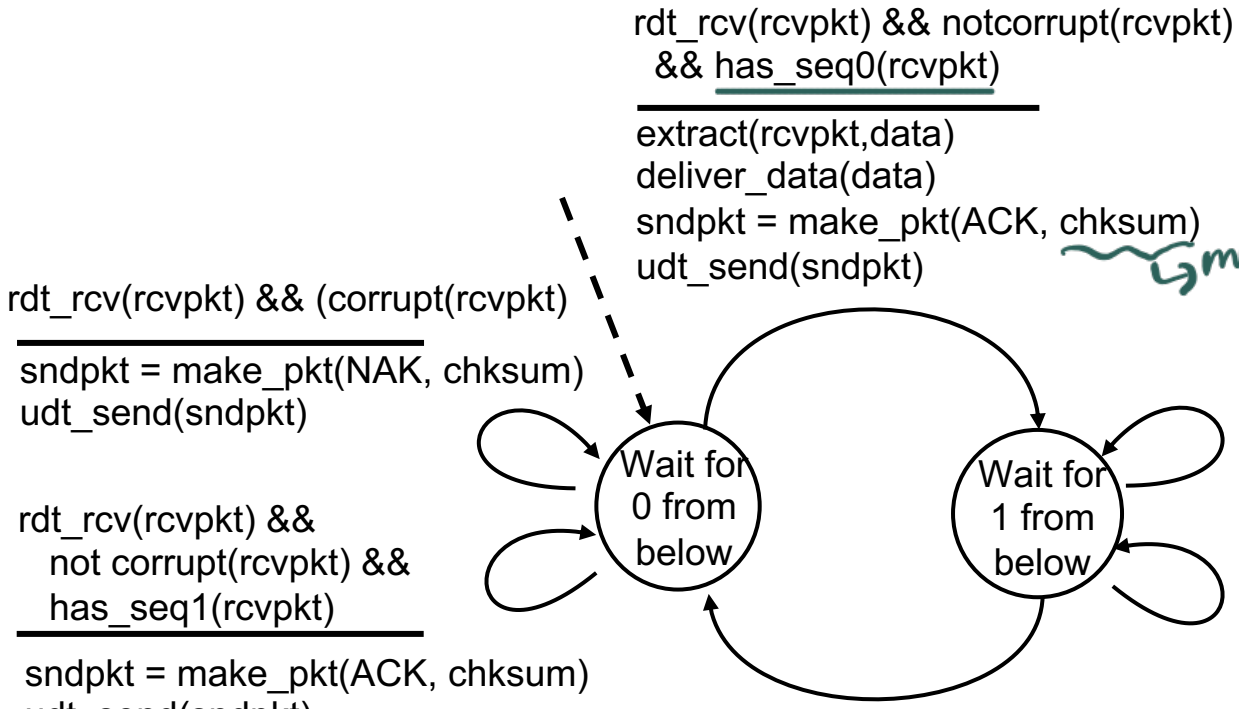
stop and wait

sender sends one packet, then waits for receiver response

rdt2.1: sender, handling garbled ACK/NAKs



rdt2.1: receiver, handling garbled ACK/NAKs



may get corrupt (error 여부판별)

corrupt ⇒ 재전송 요청

error x에서 중복 (이전 pkt) ⇒ 이미 deliver 한 data 는 있으니, ACK만 전송 (state 유지)

⇒ deliver data & ACK 전송

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 "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

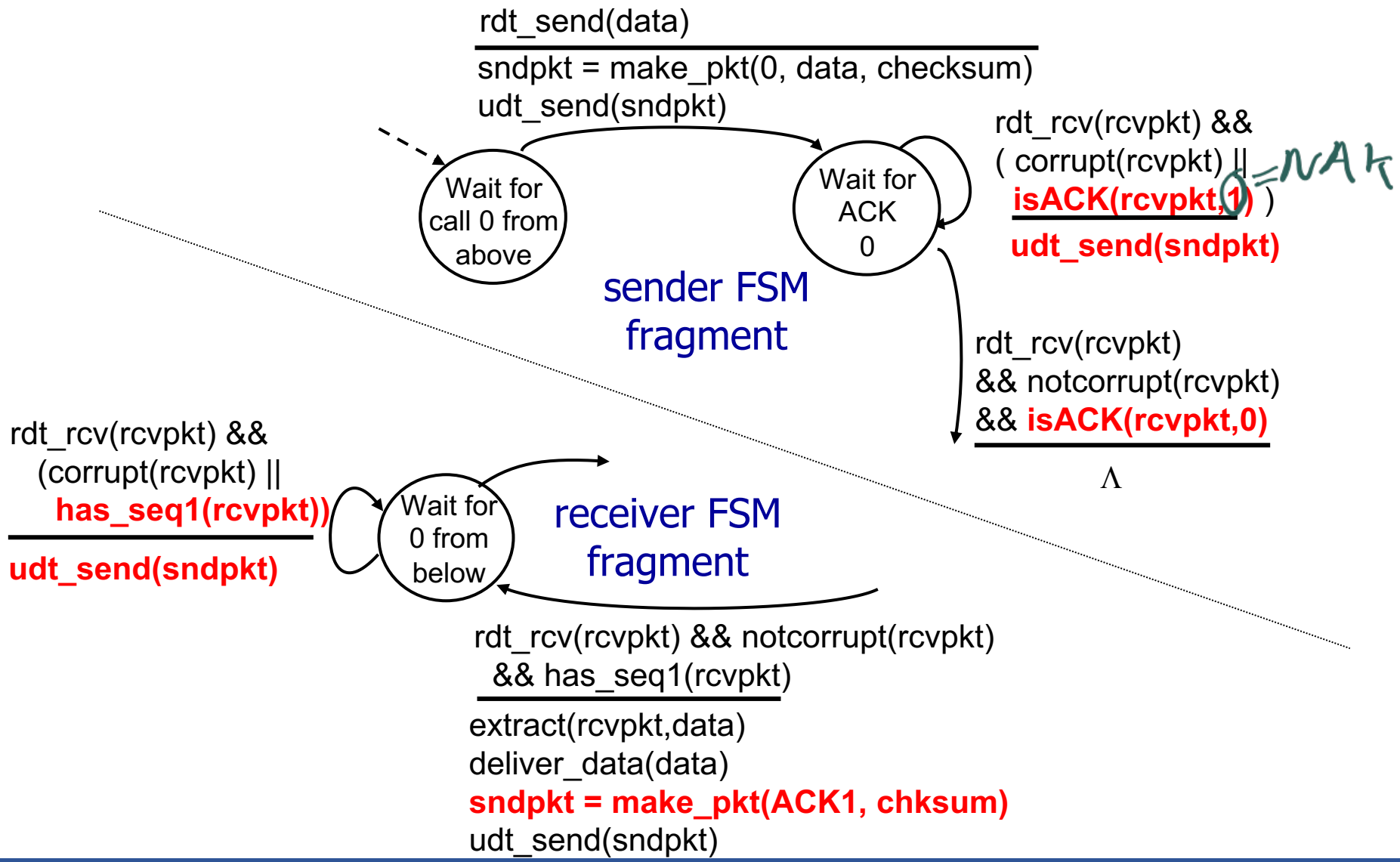
rdt2.2: a NAK-free protocol

- 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*

As we will see, TCP uses this approach to be NAK-free

→ 마지막으로 수신된 pkt의 seq # 전달 ⇒ 수신 여부 확인 가능

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss → loss 추가

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

- checksum, sequence #s, ACKs, retransmissions will be of help … but not quite enough

Q: How do *humans* handle lost sender-to-receiver words in conversation?

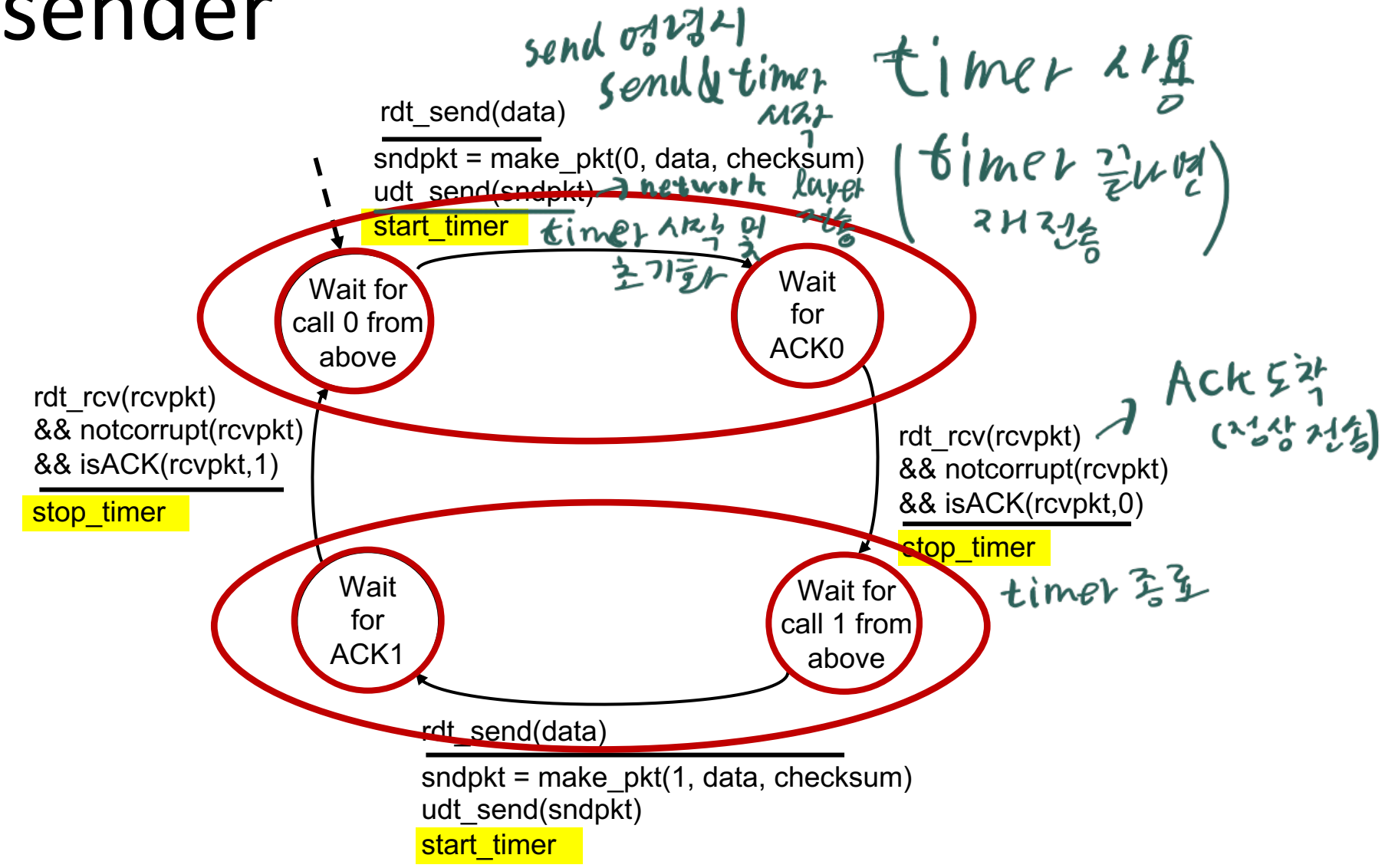
loss 시) 충분한 시간 혹 긴 반복 (repeat)

rdt3.0: channels with errors *and* loss

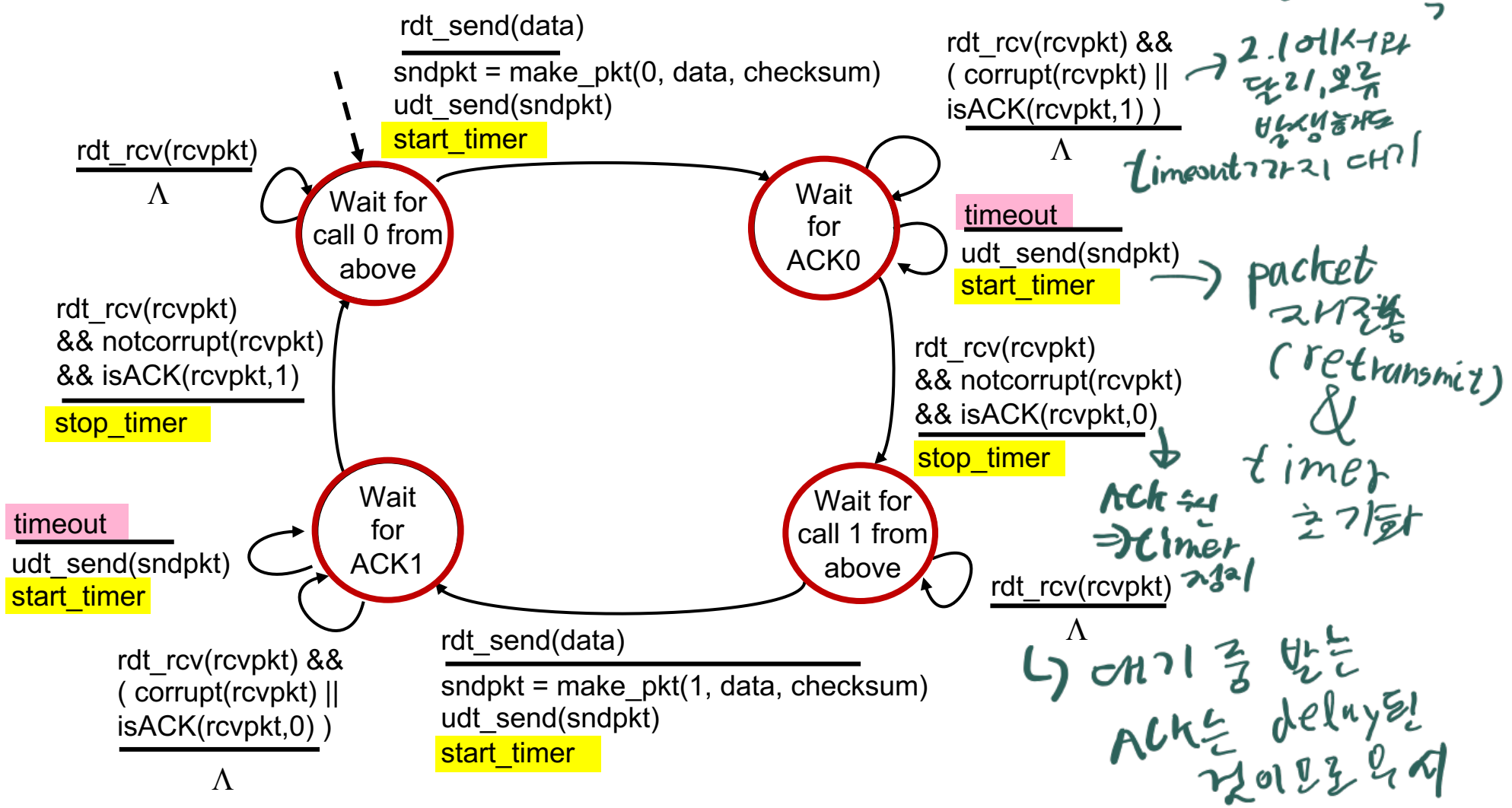
↷ ACK를 받을 때까지 대기

- 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 packet being ACKed
- use countdown timer to interrupt after "reasonable" amount of time

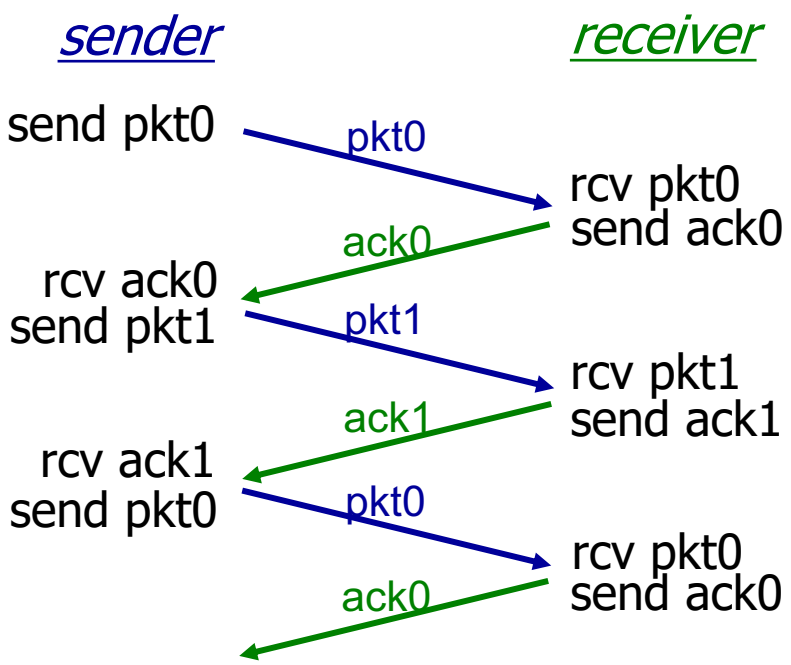
rdt3.0 sender



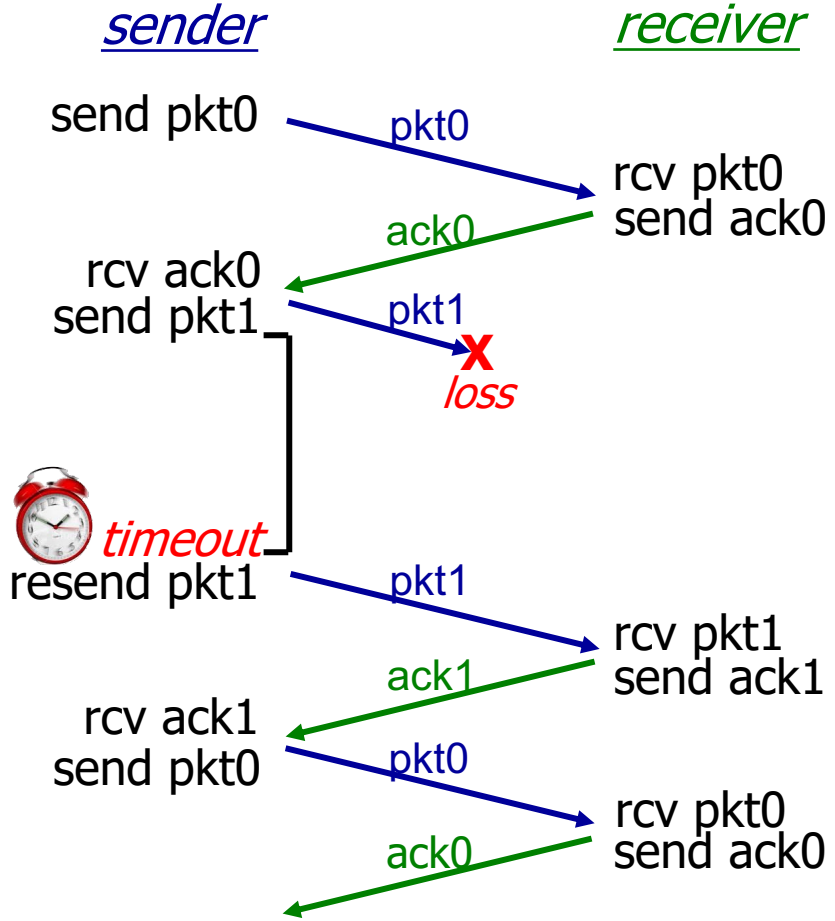
rdt3.0 sender



rdt3.0 in action

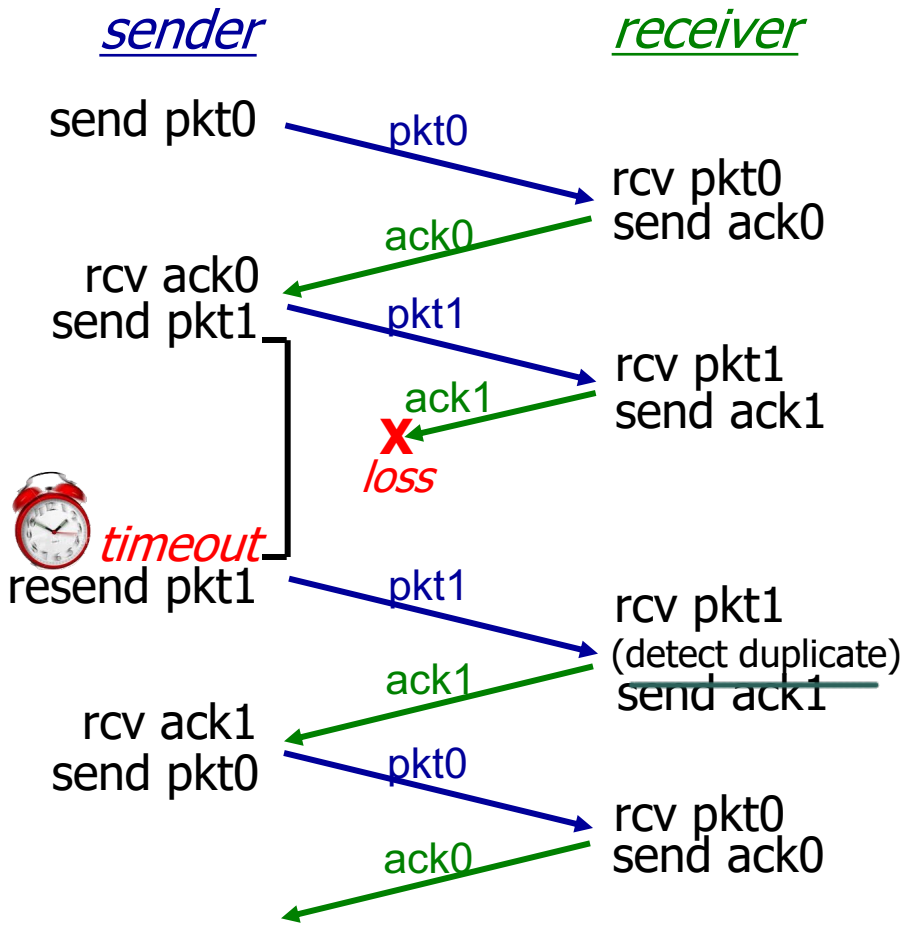


(a) no loss

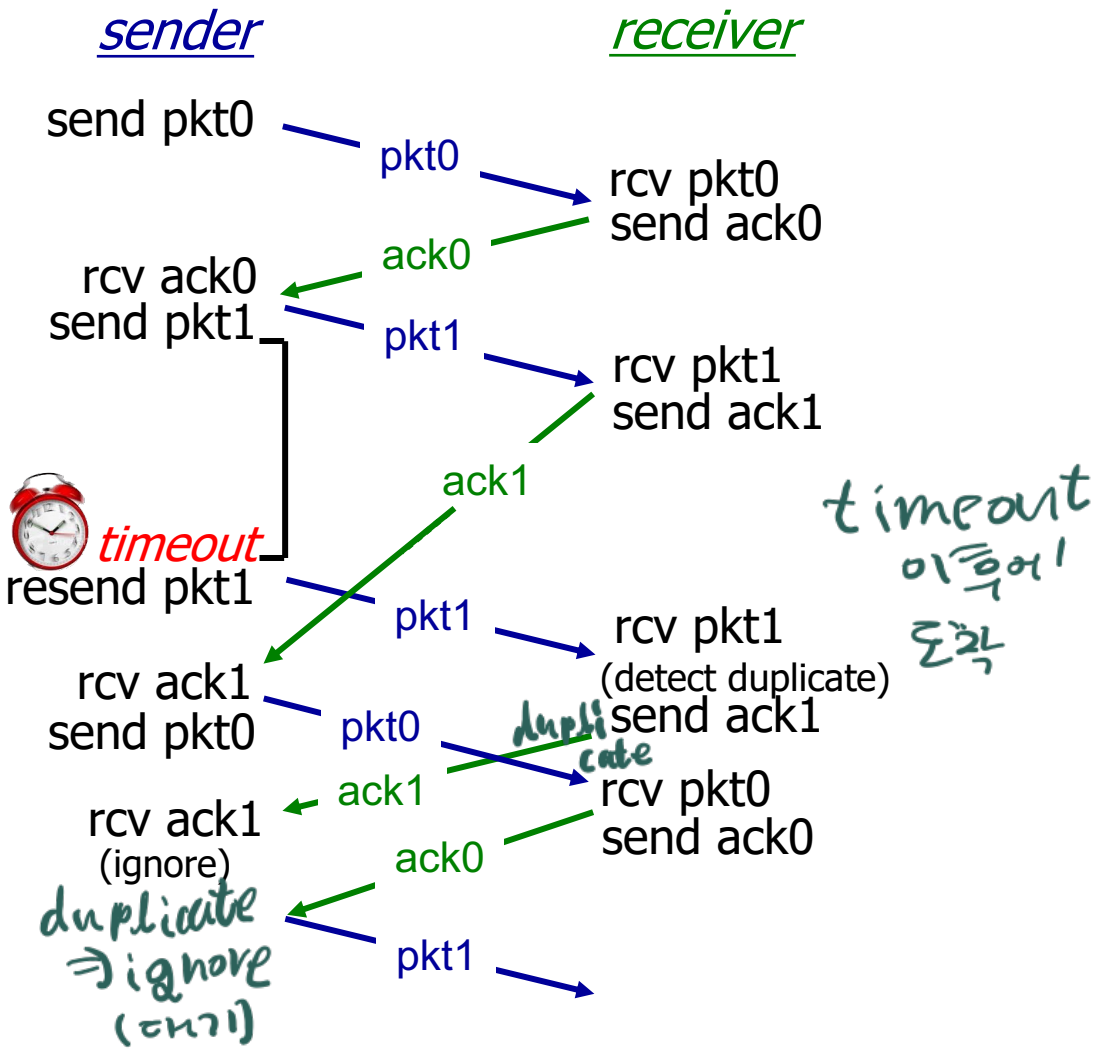


(b) packet loss

rdt3.0 in action



(c) ACK loss



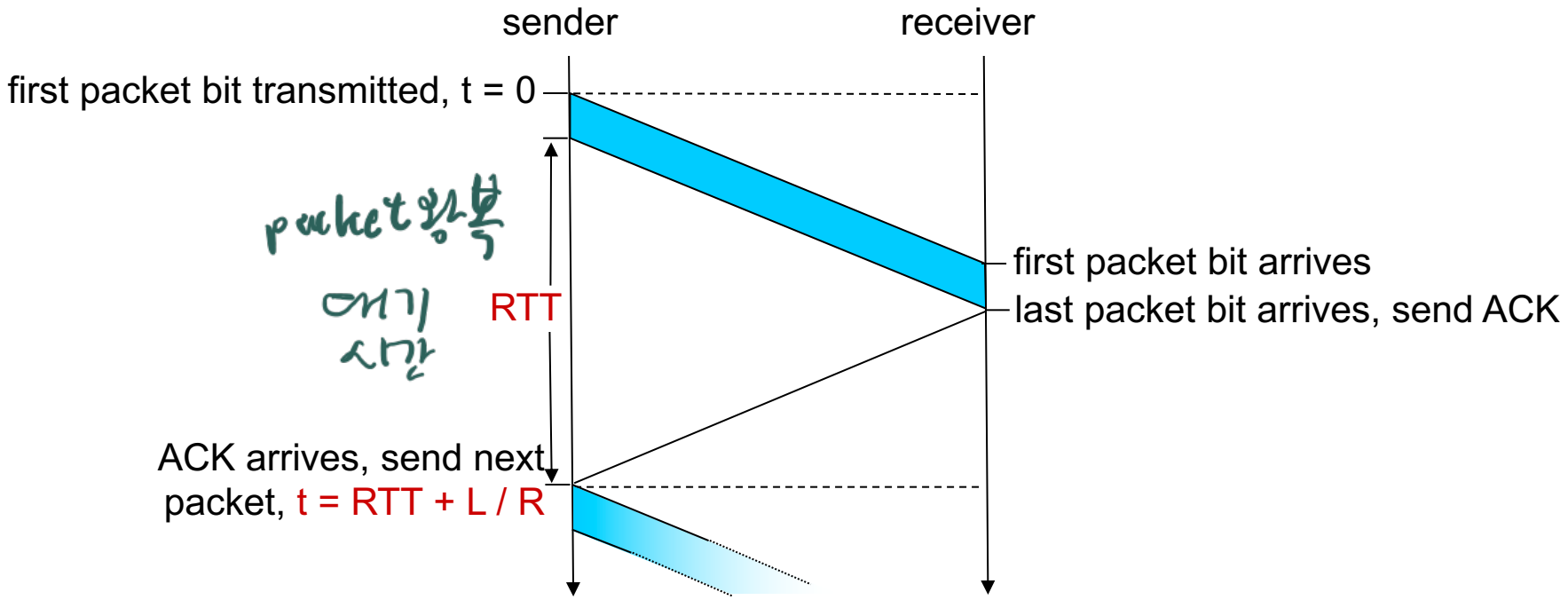
(d) premature timeout/ delayed ACK

Performance of rdt3.0 (stop-and-wait)

- U_{sender}: *utilization* – fraction of time sender busy sending
- example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet
  - time to transmit packet into channel:

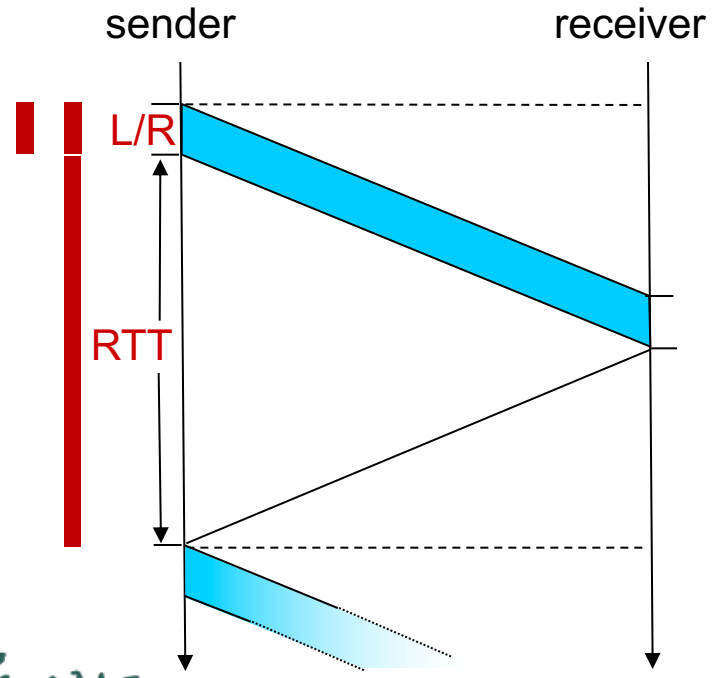
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

rdt3.0: stop-and-wait operation



rdt3.0: stop-and-wait operation

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

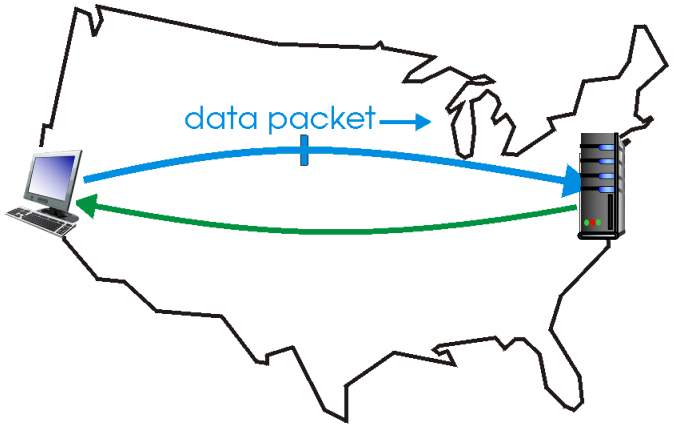


- rdt 3.0 protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

rdt3.0: pipelined protocols operation

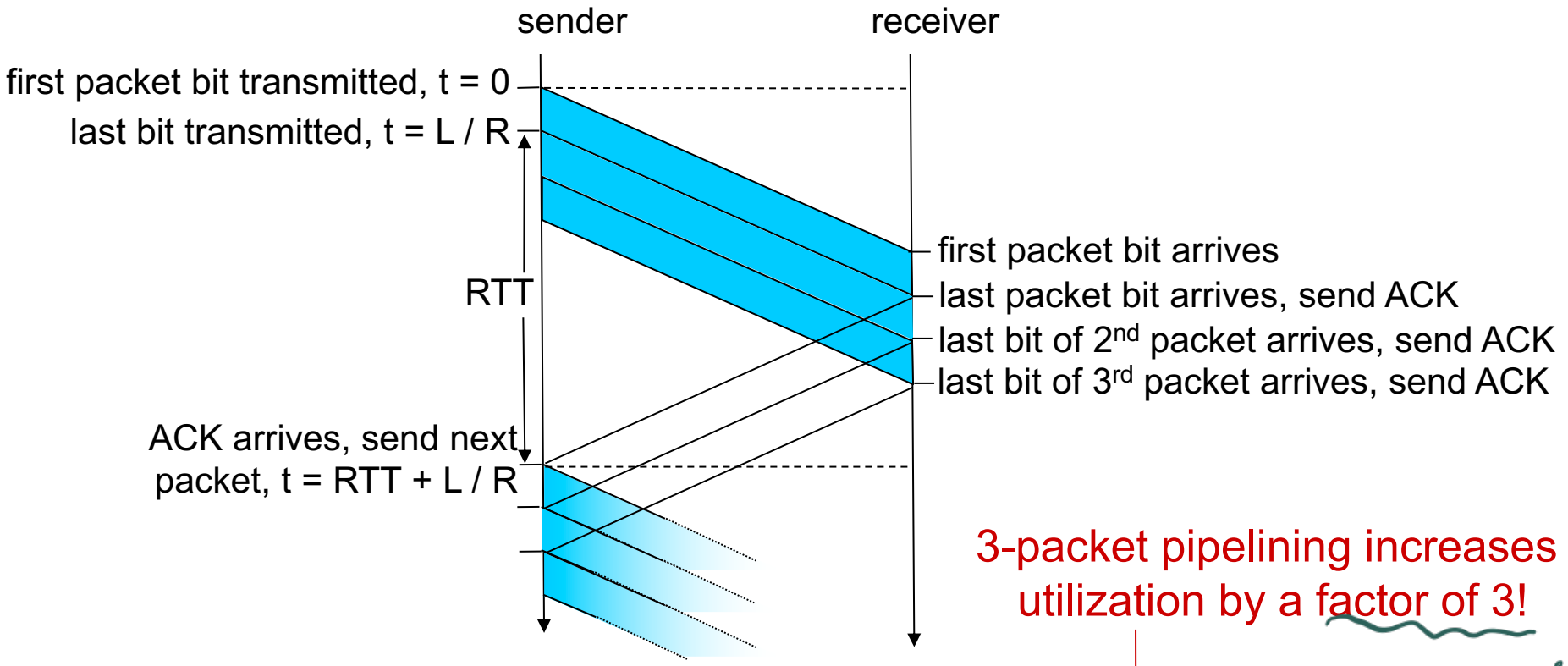
pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets

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



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

Pipelining: increased utilization



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

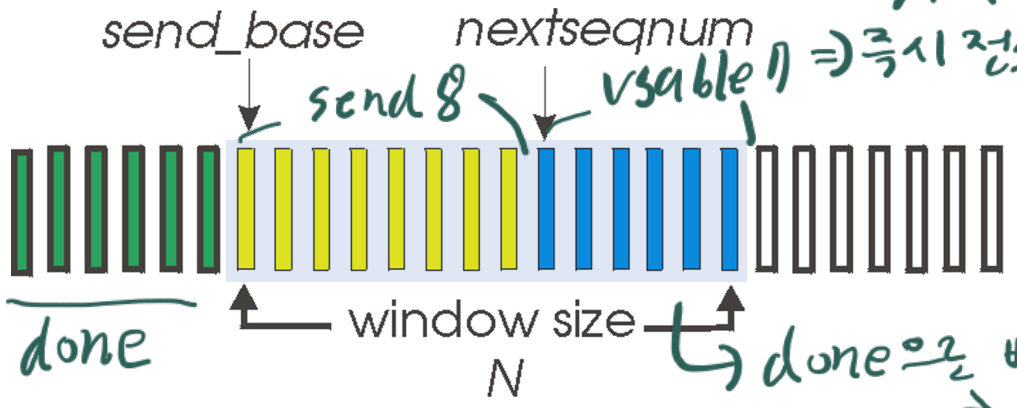
$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

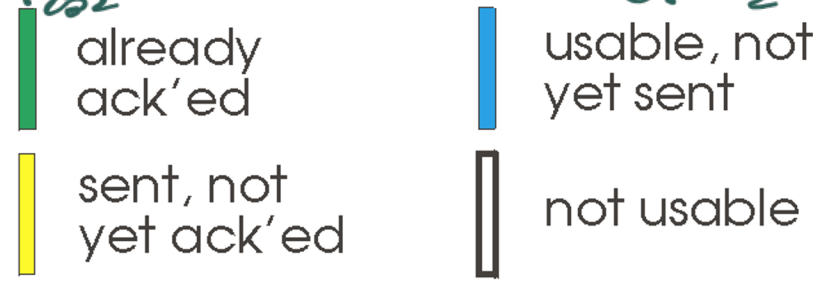
pipelining을 사용하여 utility↑

1. sender는 ACK를 받지 않은 상태로 동시에 최대로 보낼 수 있는 패킷의 수인 window size를 설정
2. sender는 send_base, nextseqnum 설정
→ send_base: 송신자가 이미 보냈지만 아직 ACK를 받지 못한 첫번째 패킷의 시퀀스 번호
→ nextseqnum: 송신자가 현재 전송할 수 있는 다음 패킷의 시퀀스 번호
3. 수신자는 패킷들을 연속해서 받고, 각각의 패킷을 받을 때마다 sequence number를 확인하고 ACK를 전송
→ N번까지 잘받았다 → Culmulative Ack
→ 패킷이 유실되어 sequence number에 빈 틈이 생기면, 정상적으로 받은 가장 마지막 패킷 번호를 ACK에 담아 보낸다.
→ 수신자는 정상적으로 받은 마지막 패킷 sequence number를 rcv_base에 기록한다.
4. 송신자는 보낸 패킷의 ACK가 정상적으로 왔다면, 패킷 처리 후 window를 오른쪽으로 한 칸 sliding 한다.

Go-Back-N: sender

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header



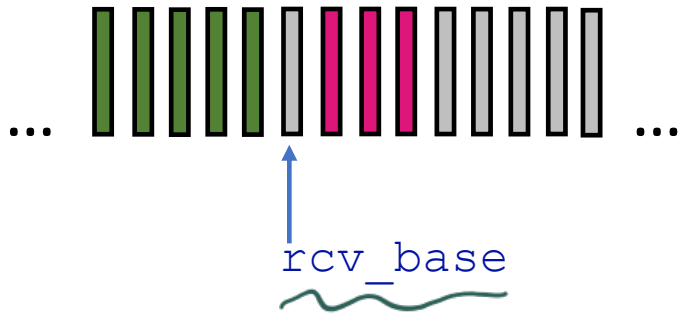


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

Go-Back-N: receiver

- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

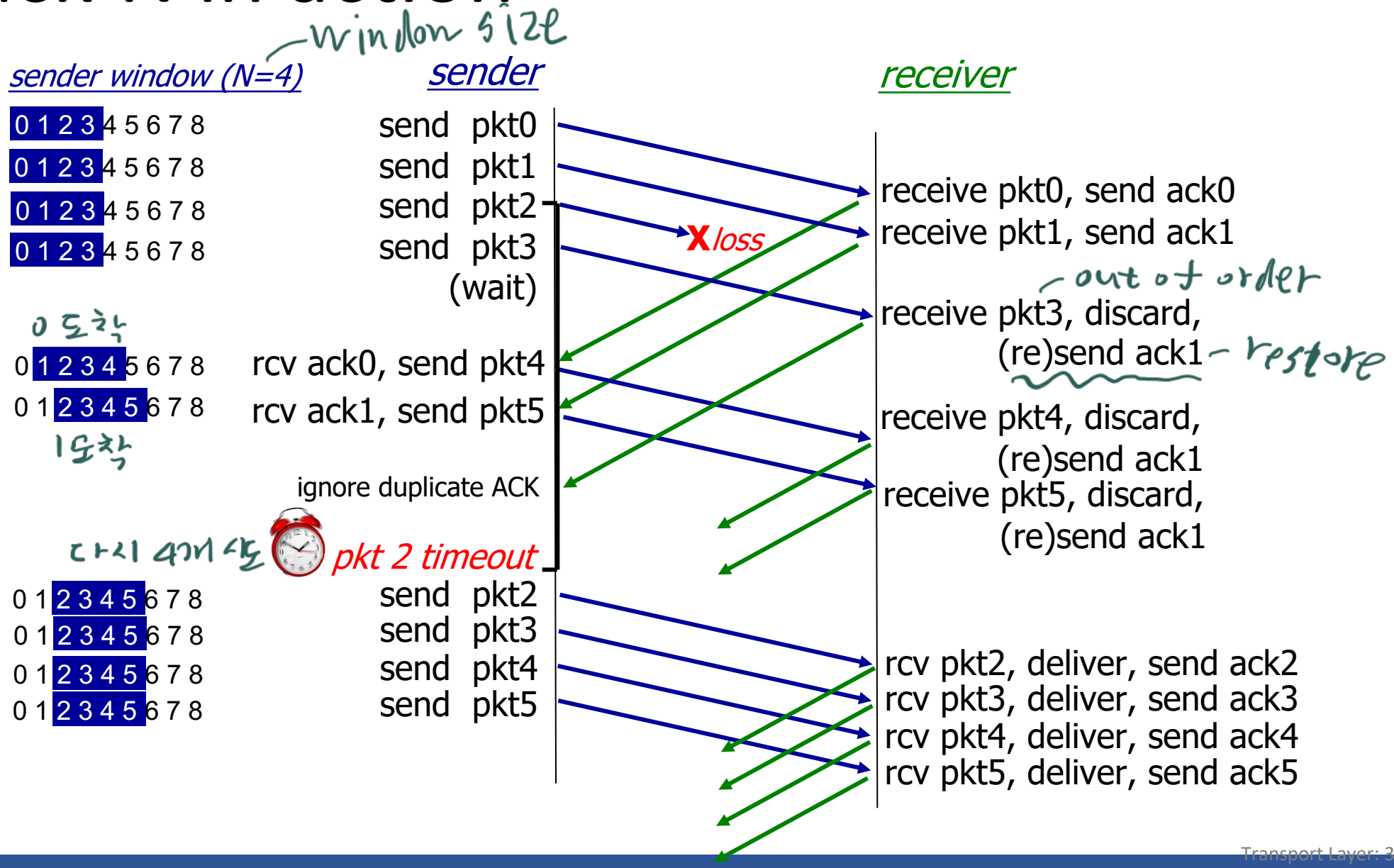


received and ACKed

Out-of-order: received but not ACKed

Not received

Go-Back-N in action



1. sender는 window size 설정
2. window size 내에 들어있는 패킷들을 전부 송신 후 ACK을 기다림
→ 각각의 패킷에 대해 타이머를 가지고 있어 time-out을 활용해 재전송 한다.
3. Selective repeat 방식에서는 수신자가 버퍼를 가지고 있다.
-수신 패킷 처리: 수신자는 패킷의 시퀀스 번호를 확인하여 rcv_base에 저장된 시퀀스 번호와 어긋난다면 패킷들을 임시로 버퍼에 저장해 둔다 (중간에 currupt 발생).
-순서에 맞는 패킷 도착 & ACK 전송: 이후에 순서에 맞는 패킷이 도착하면, 수신자는 버퍼에 저장된 패킷들을 모아 상위 계층으로 전달한다.
-윈도우 슬라이딩: ACK 전송 횟수만큼 송신자의 윈도우가 슬라이딩된다.
4. 특정 패킷에 대해 ACK가 들어오면 그 패킷을 완료 기록
→ 송신자가 보낸 패킷들 중 아직 ACK되지 않은 패킷들 중 가장 번호가 작은 패킷이 ACK되면 window를 sliding

Selective repeat – another way

각각의 packet에 대한 정보 저장

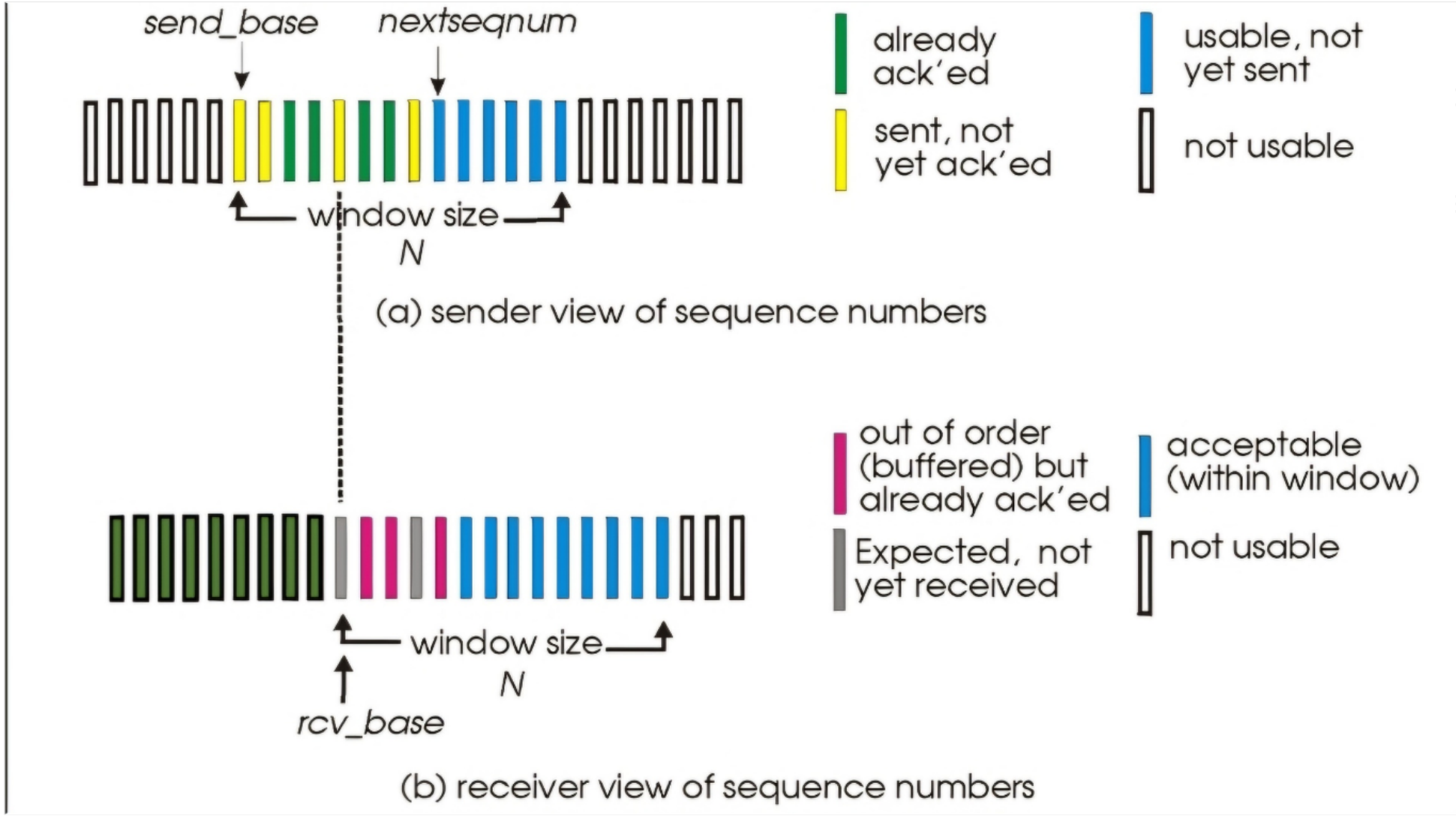
- receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

buffer 필요

실패해 이득을 전부 재전송하는 예신, packet만 재전송

ACK 전달 ⇒ 실패한

Selective repeat: sender, receiver windows



Selective repeat: sender and receiver

sender

data from above:

- if next available seq # in window, send packet

timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N]:

- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

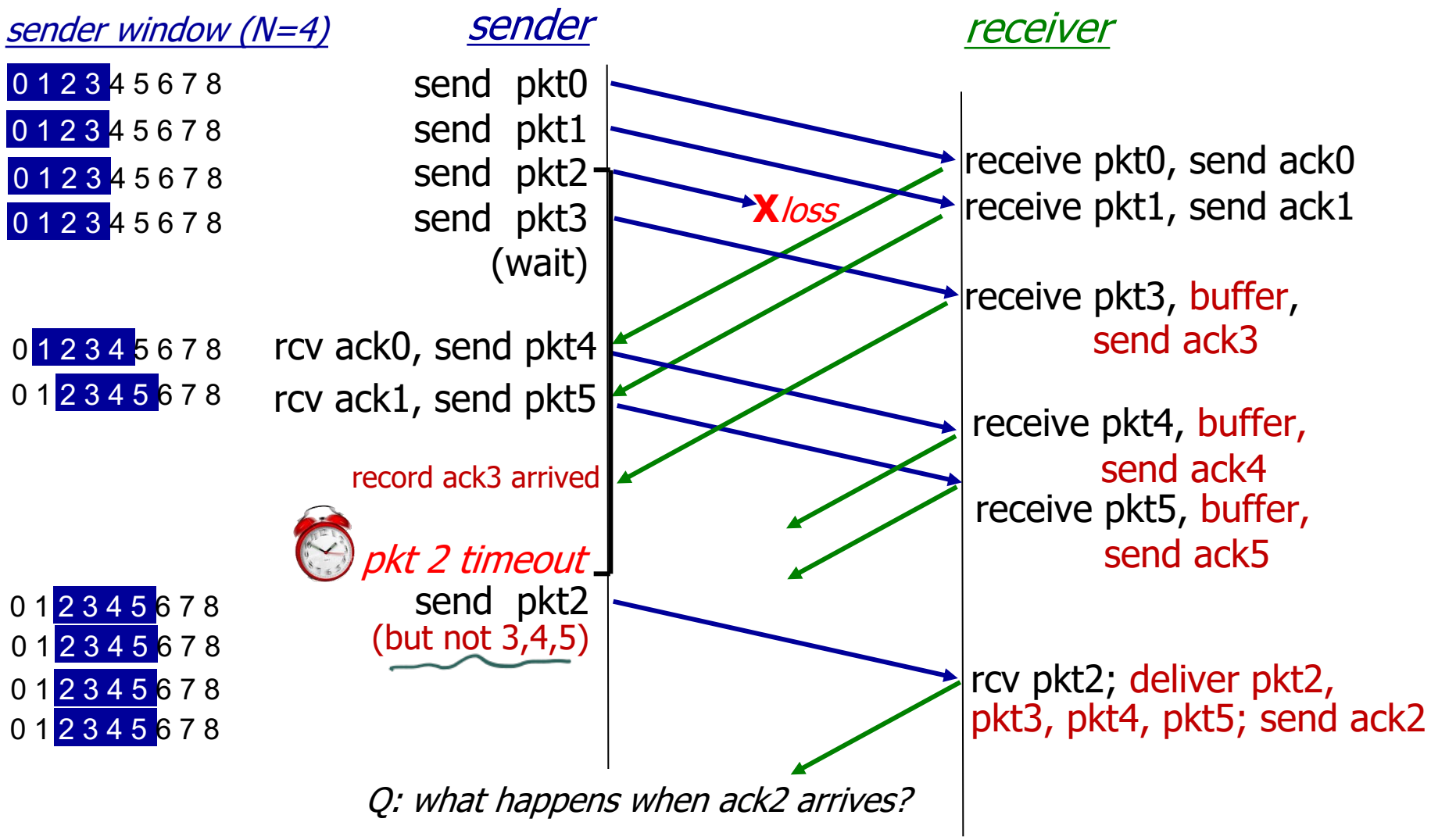
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action



Next…

- *Chapter 3.5 Connection-Oriented Transport: TCP*