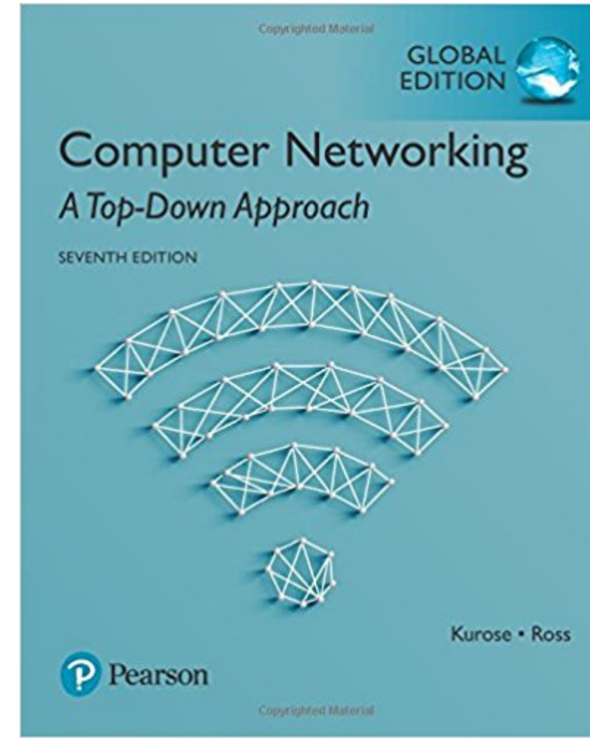
Chapter 3
Transport Layer part 2

School of Computing
Gachon Univ.
Joohyung Lee

Most of slides from J.F Kurose and K.W. Ross. And, some slides from Prof. Joon Yoo



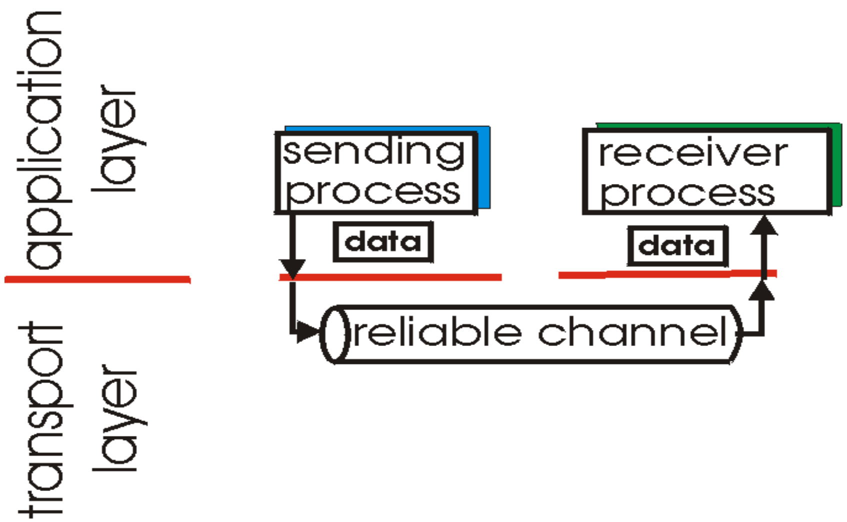
Computer Networking: A Top Down Approach

7th edition
Jim Kurose, Keith Ross
Pearson, 2017

Chapter 3 outline

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!



(a) provided service

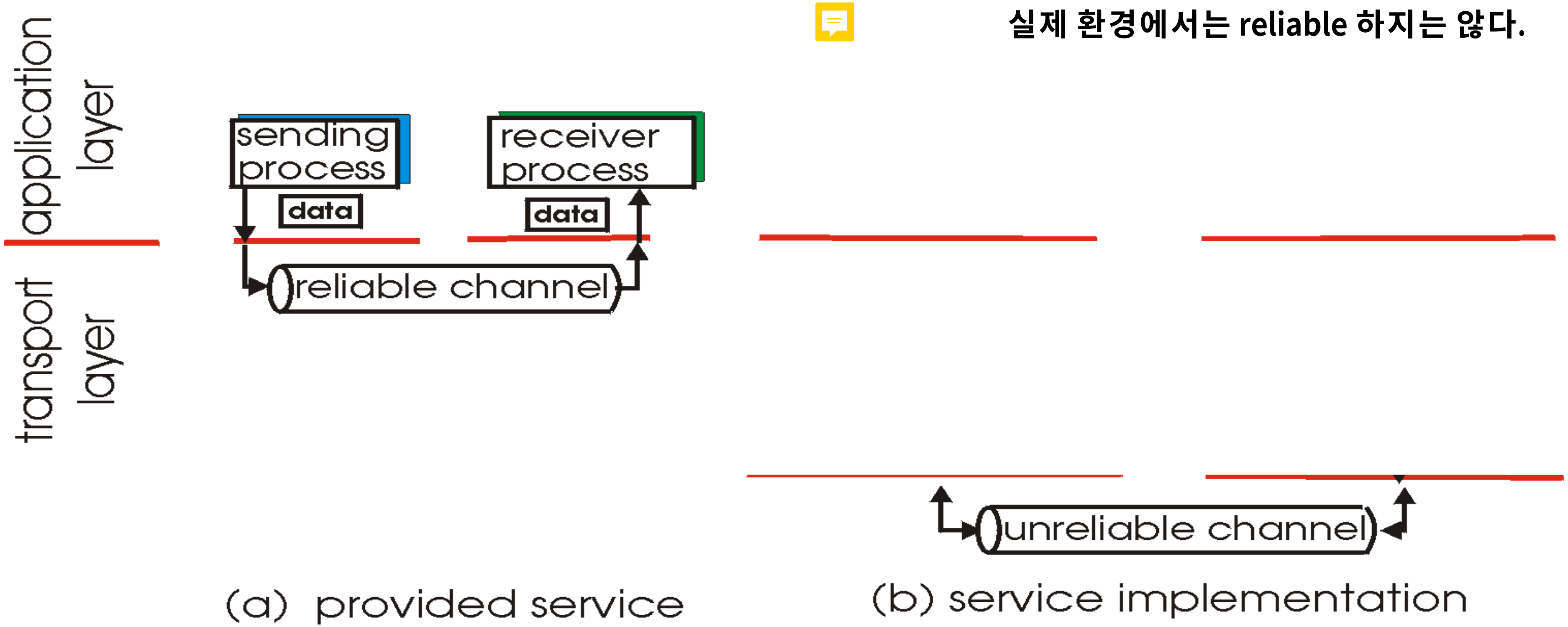
process와 socket은 1:1 대응은 아니지만 그렇게 이해하는게 쉬움

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

실제 환경에서는 reliable 하지는 않다.



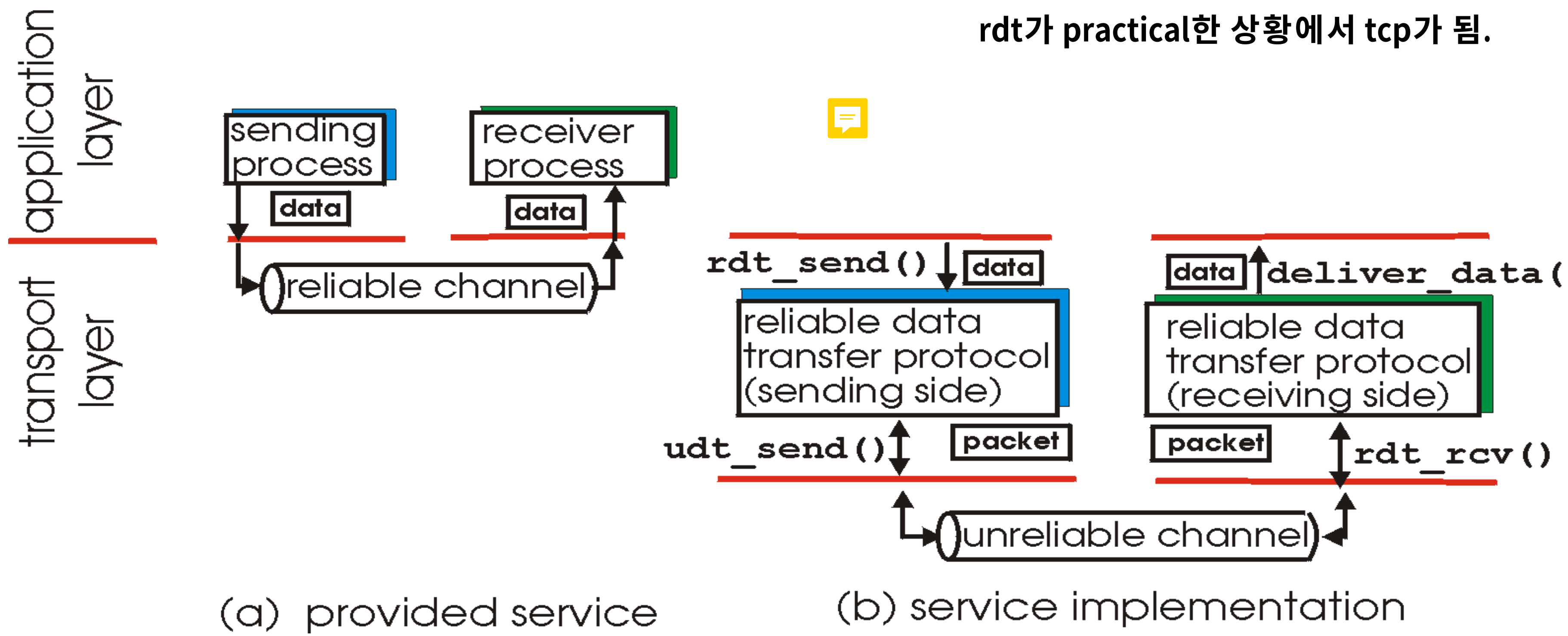
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

Reliable data transfer protocols

rdt가 practical한 상황에서 tcp가 됨.

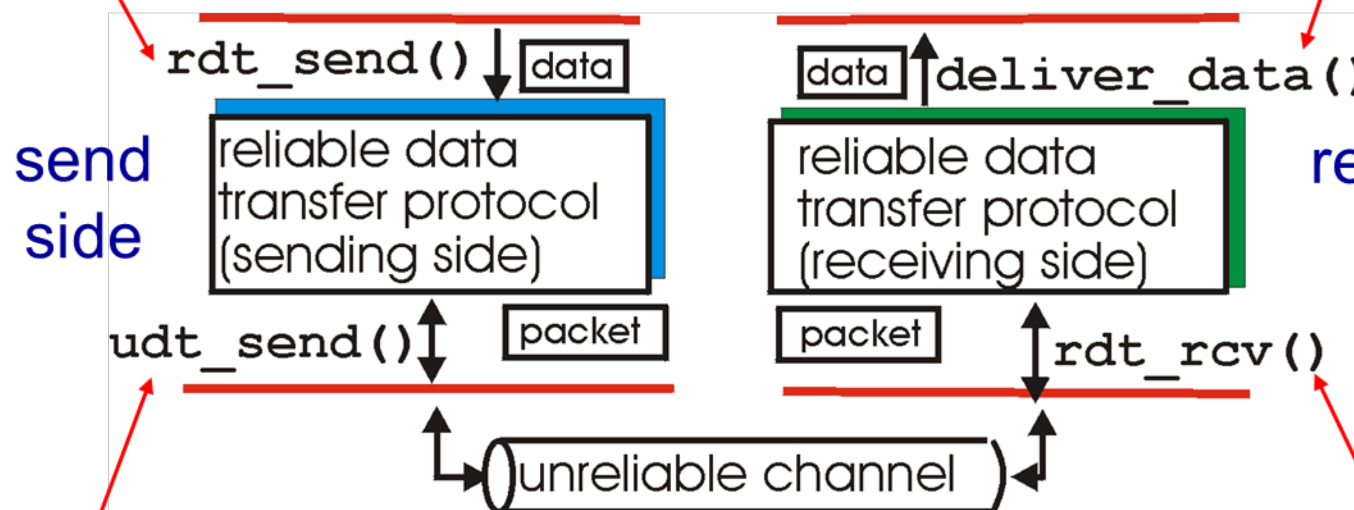


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

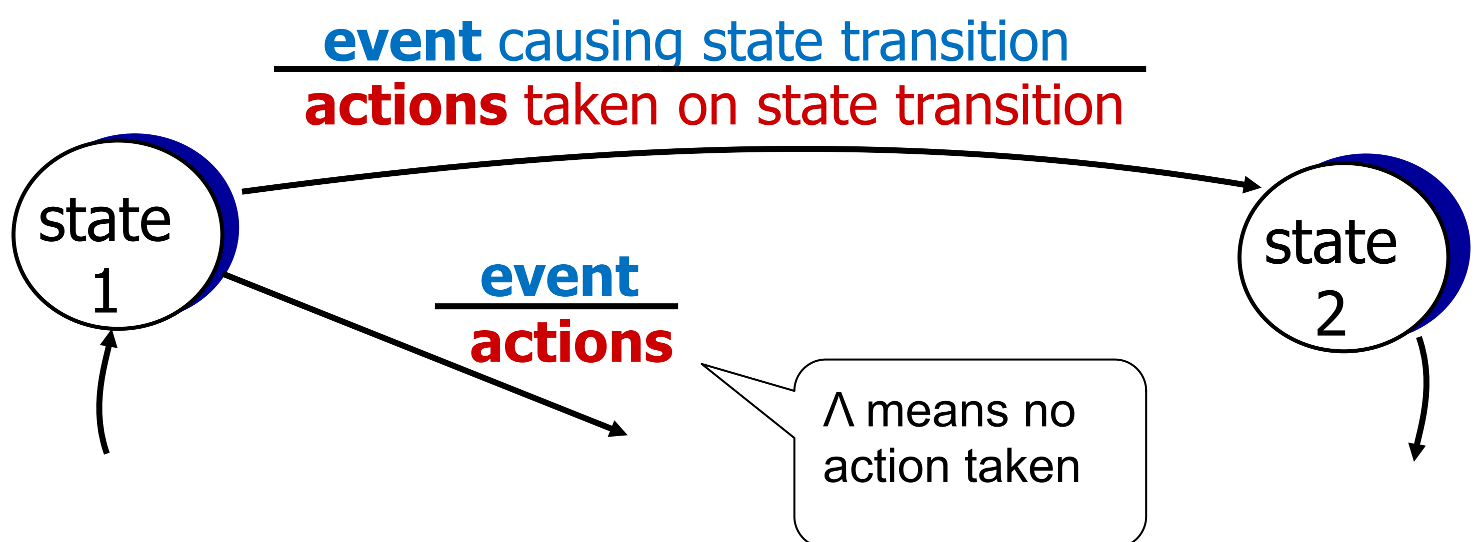
Reliable data transfer: getting started

state machine을 만듦. 네트워크를 구현할 때 많이 쓰는 기법.
동적으로 변화하는 시스템을 구현할 때 좋음. 코드일 때 pseudo code,
여기서는 FSM

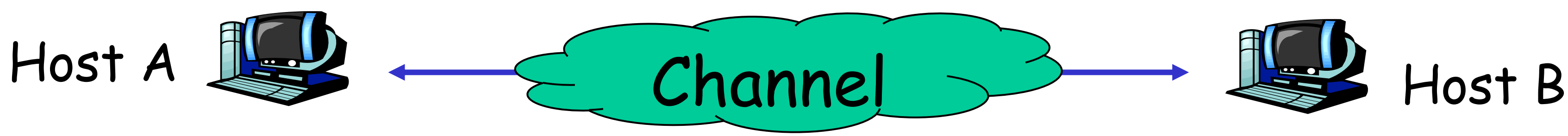
We'll:

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

state: when in this "state" next state is determined by some **event**



Unreliable channel?
Potential Channel Errors



- bit errors (but no packet losses)
  - How does receiver find bit errors?
- loss (drop) of packets
  - How does sender find packet losses?
- reordering or duplication
  - Arriving Pkts: 1, 2, 4, 5, 3, … or 1, 2, 3, 3, …

데이터가 왔지만 변형된 데이터가 옴

queuing loss 때문에 많이 발생. 변행된 데이터가 아예 오지 않음.

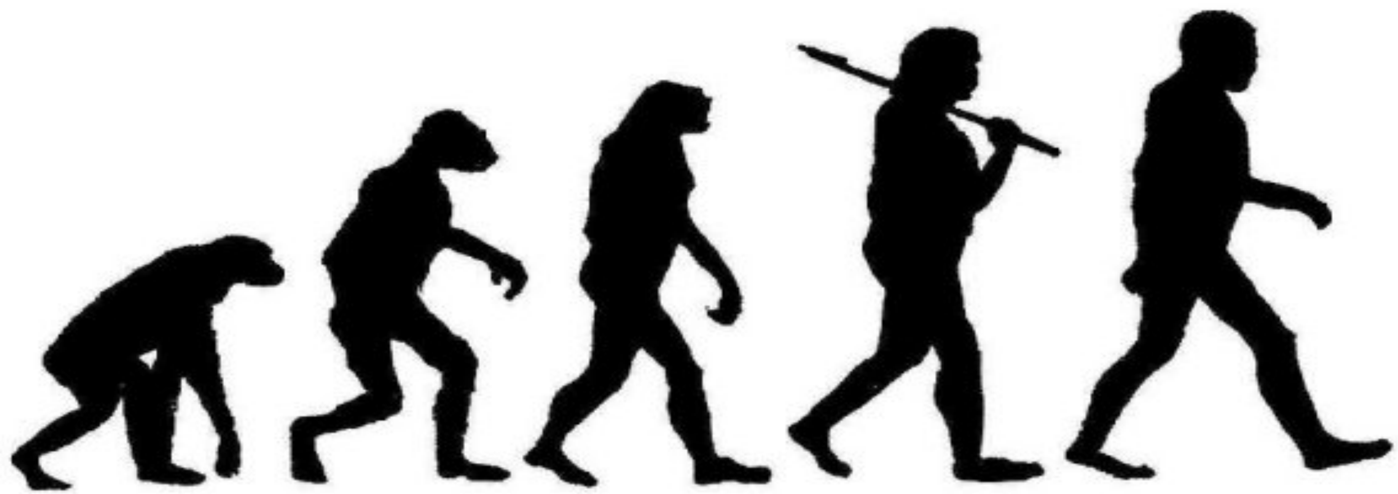
데이터가 순서가 바뀌어 오거나 중복하여 옴.
재전송을 하다보면 같은 데이터가 또 올 수 있음.

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt).

Different Conditions, Different Solutions

- The more difficult the condition,
  the smarter the solution to deal with the problem

임의로 지은 이름들이다.



reliable 하다고 가정함.

rdt (reliable data transfer) over <u>a reliable channel</u>

v1.0

rdt over <u>channels with bit errors</u>

v2.0

상대적으로 간단한 bit error가 발생한다는 가정을 함.

rdt over <u>channel with errors and losses</u>

v3.0

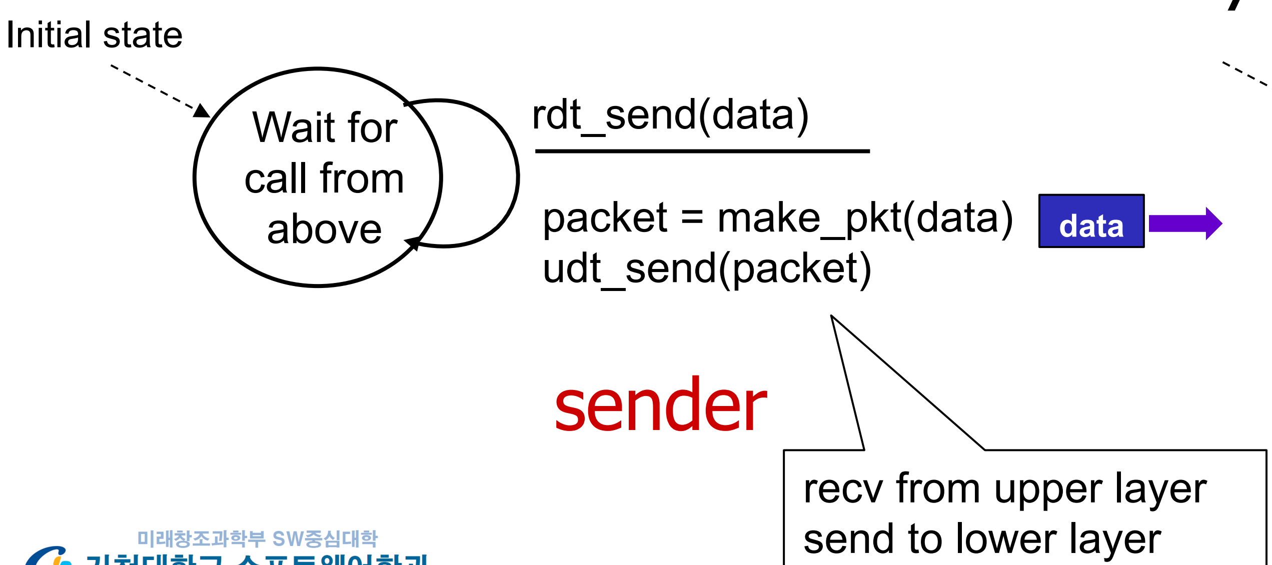
reliable 하게 하려면 어떻게 해야할까.

rdt1.0: reliable transfer over a reliable channel

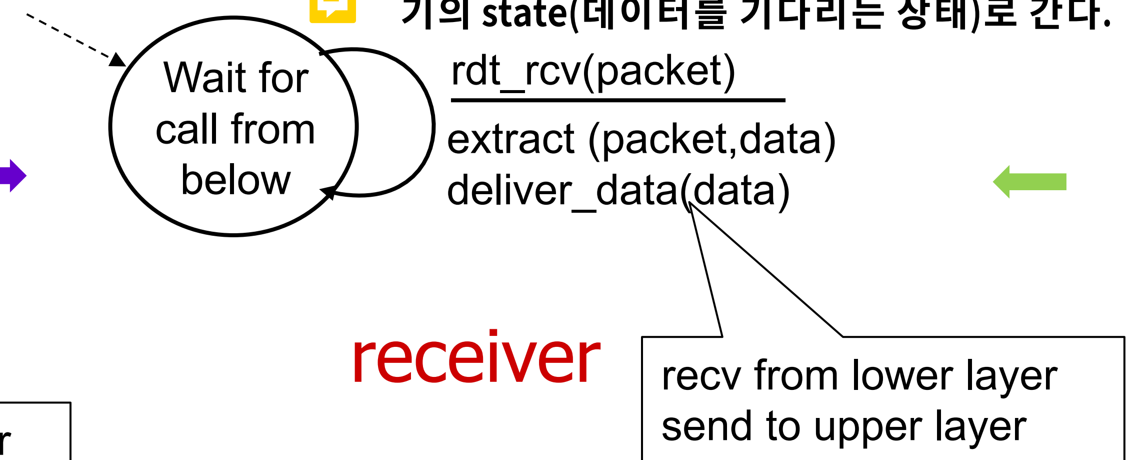
- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

rdt 1.0은 기능이 없다. 채널 자체가 완벽히 reliable 하다는 가정하에 했기 때문.
sender : 밑에 채널로 전달하는 역할
receiver : 받아서 올려주는 역할

application data를 가지고 packet을 만든다. unreliable한 데이터를 던진다. ...?
receiver : high layer로부터 데이터(packet)을 받는다. packet으로부터 데이터를 추출한다. 그 후 상위 계층으로 보내준다. 이 후에 자기의 state(데이터를 기다리는 상태)로 간다.



sender



receiver

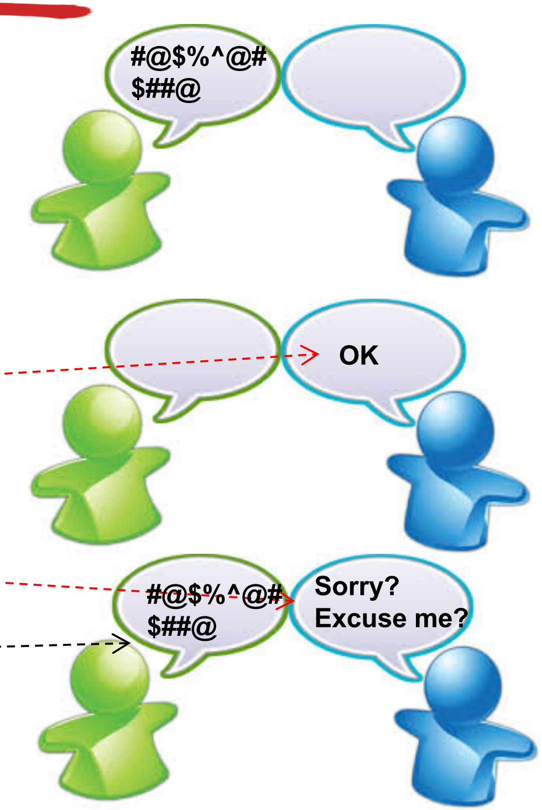
rdt2.0: channel with bit errors

- underlying channel may flip bits (0→1 or 1 → 0) in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

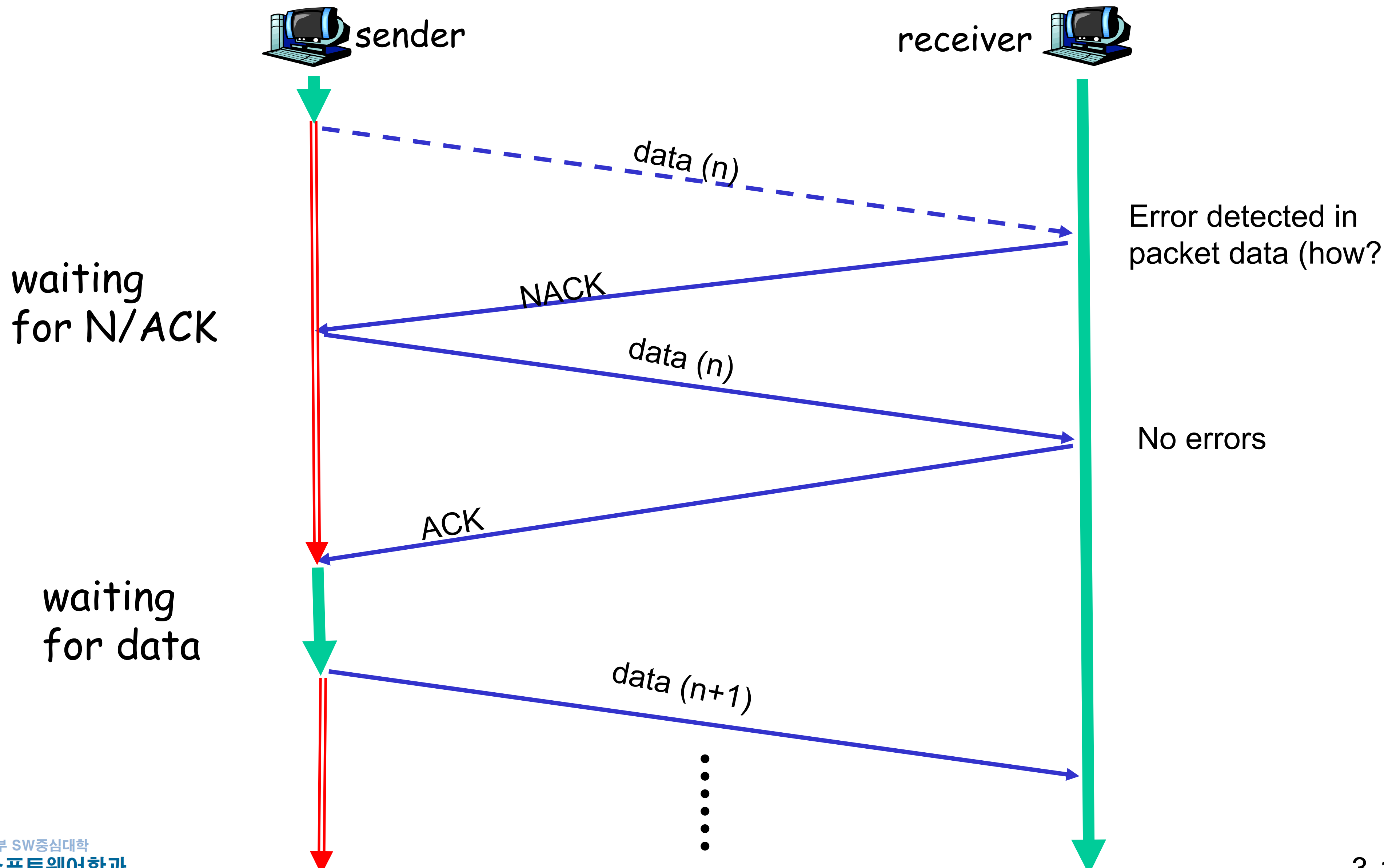
How do humans recover from "errors" during conversation?

rdt2.0: channel with bit errors

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



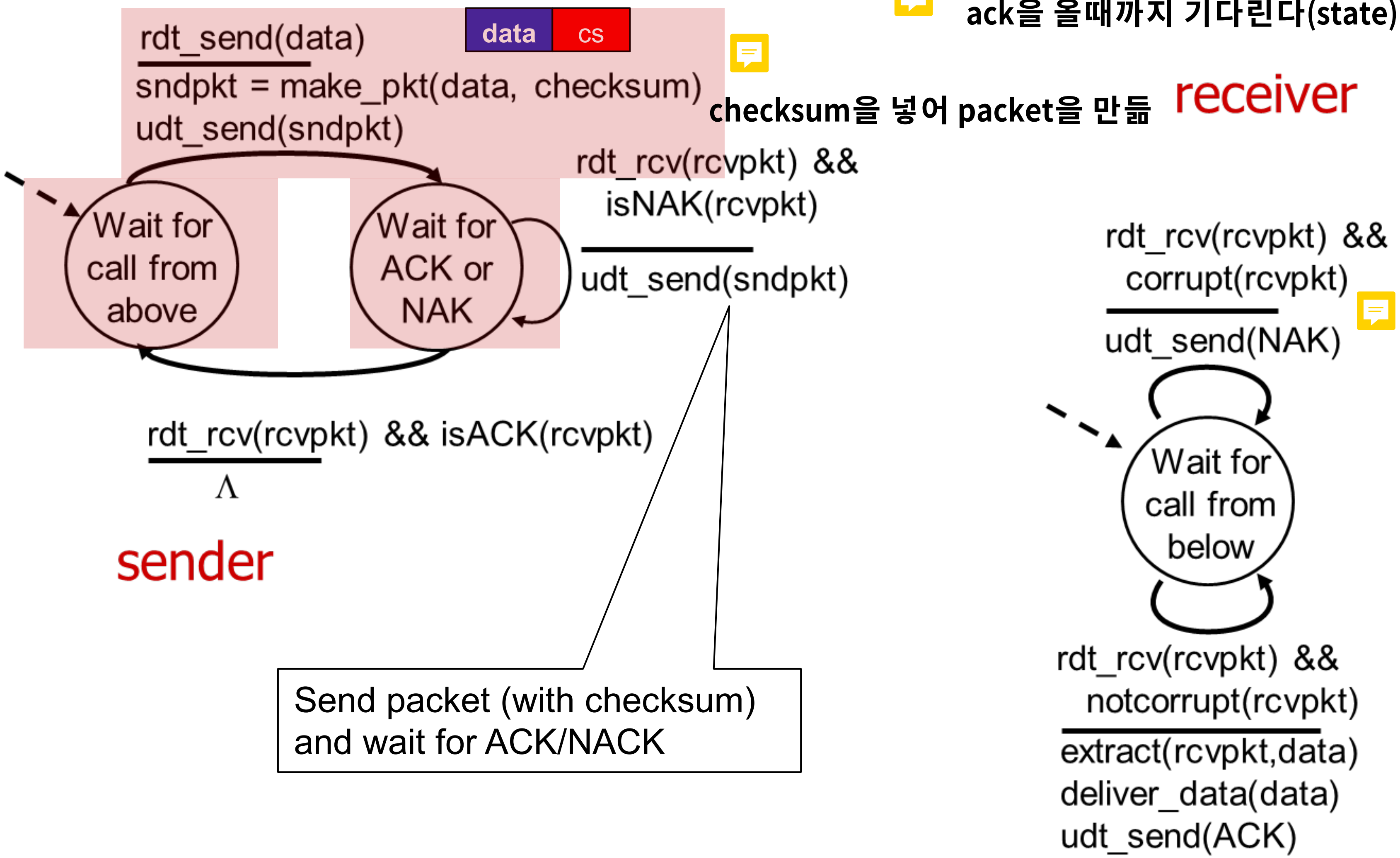
Big Picture of rdt2.0



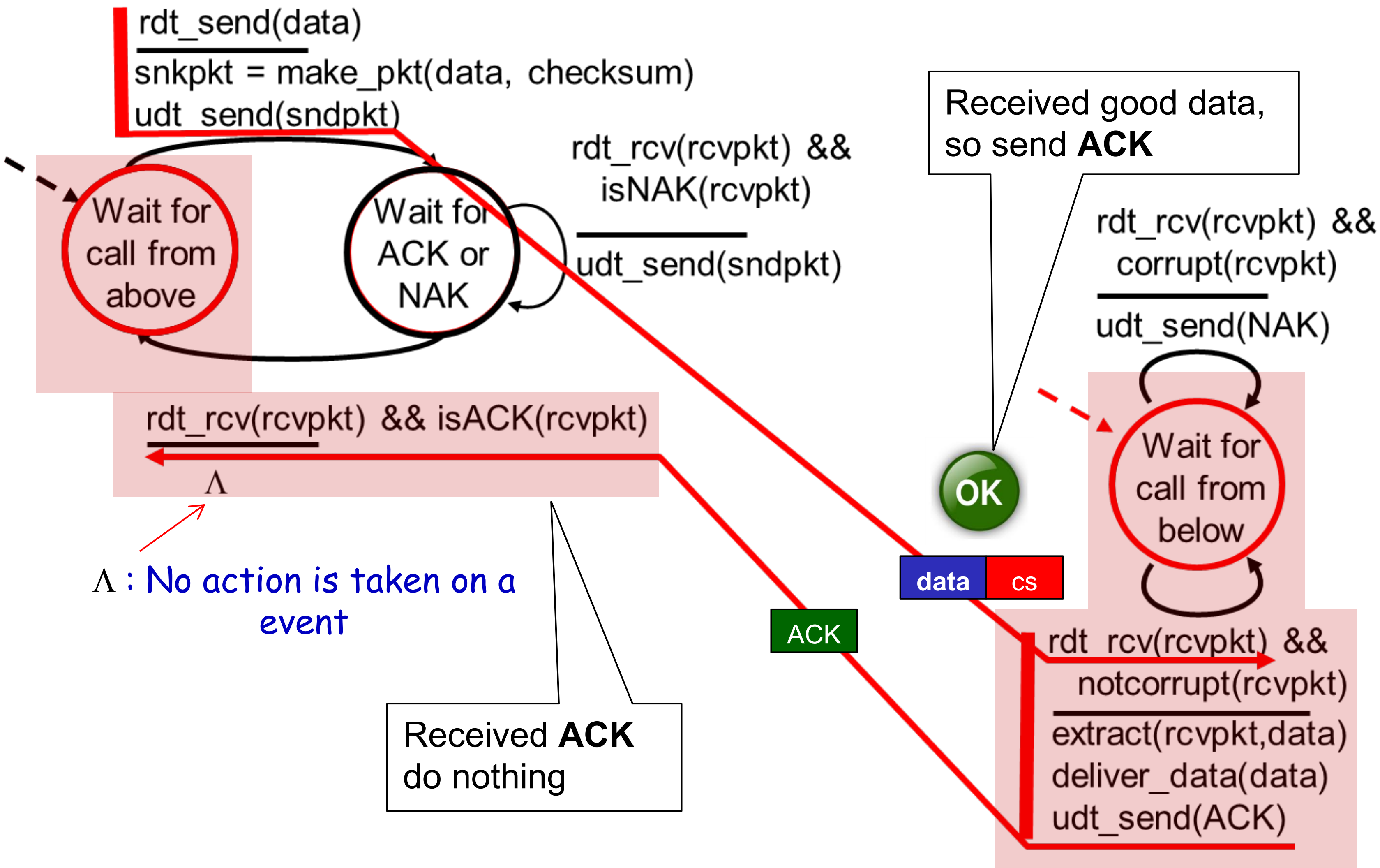
rdt2.0: FSM specification

1.0 버전과는 달리 받으면 state가 바뀐다.
데이터를 보냈으면 상대방이 잘 받았는지 nack이나
ack을 올때까지 기다린다(state)

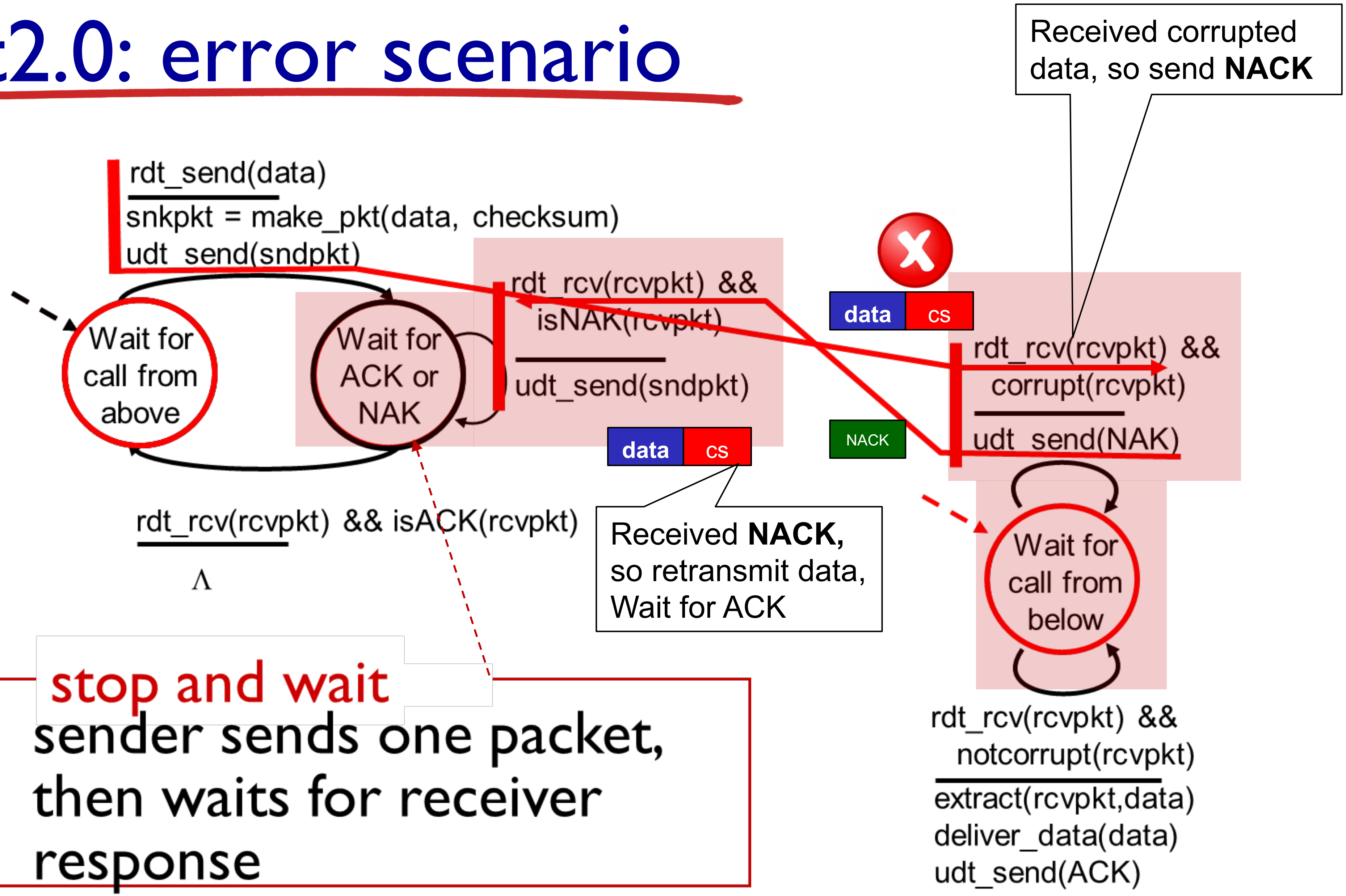
checksum을 넣어 packet을 만듦



rdt2.0: operation with no errors



rdt2.0: error scenario



stop and wait

sender sends one packet, then waits for receiver response

rdt3.0: channels with errors *and* loss

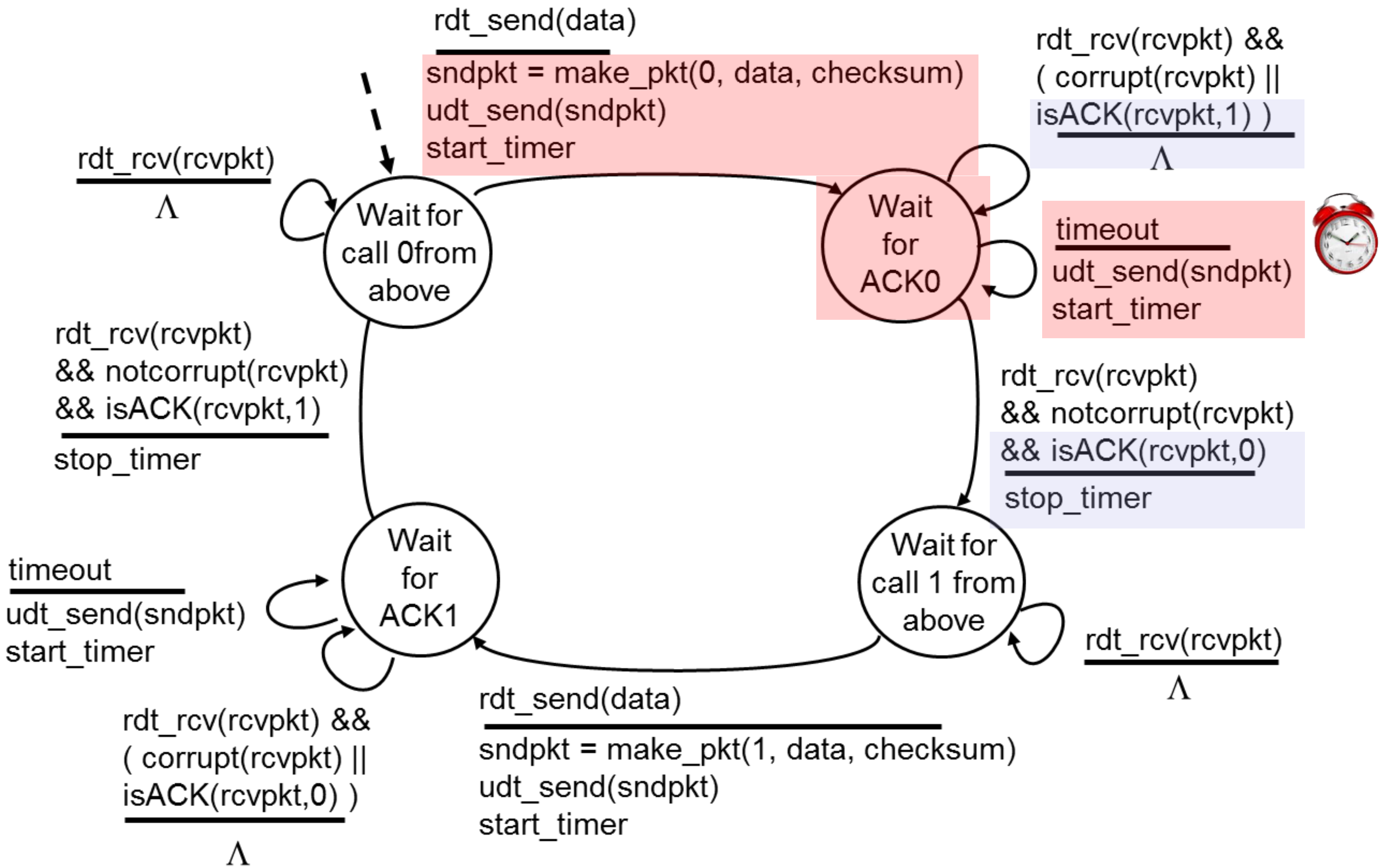
new assumption:
underlying channel can also ***lose*** packets (data, ACKs)

- Receiver will *not* receive any segments
- How will sender know that data packet (or ACK) has been lost?

approach: sender waits ***reasonable*** amount of time for ACK

- retransmits if no ACK received in this time
- requires **countdown timer** 

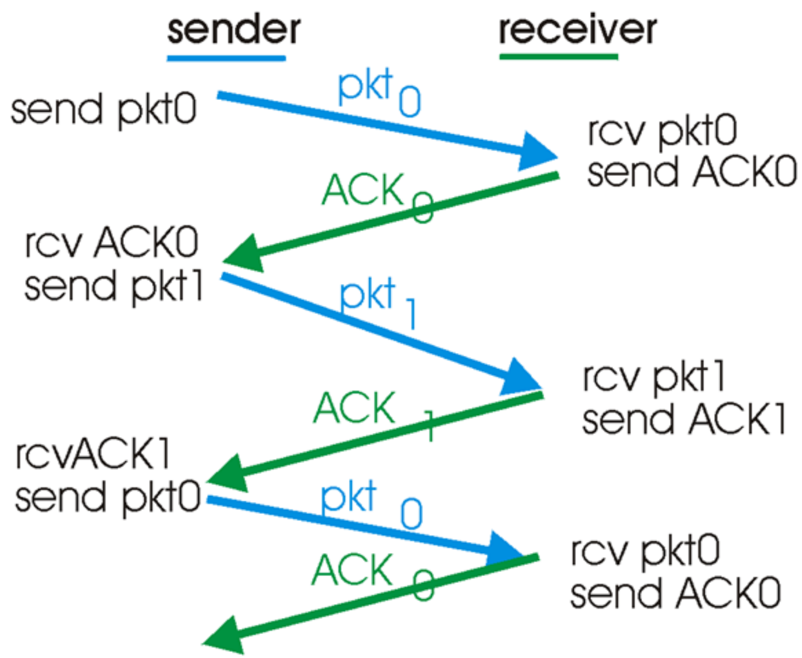
rdt3.0 sender



rdt3.0 in action

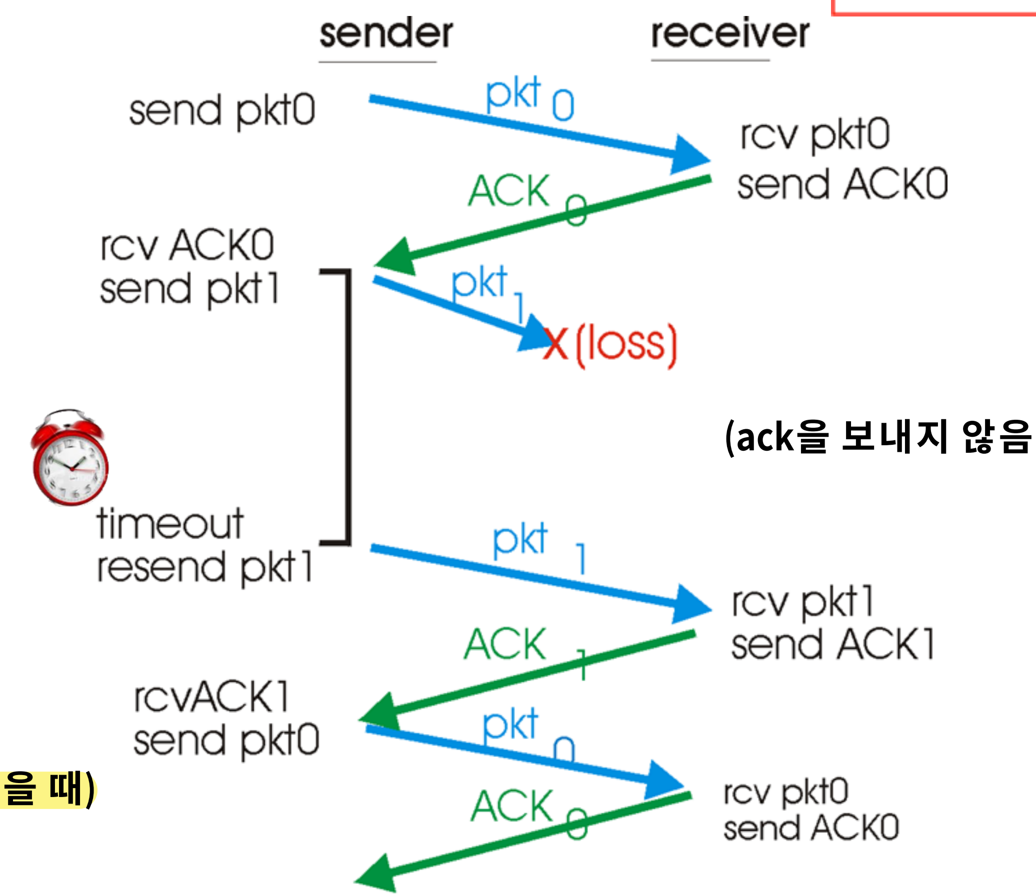
How many seq#?

Two sequence numbers are enough!



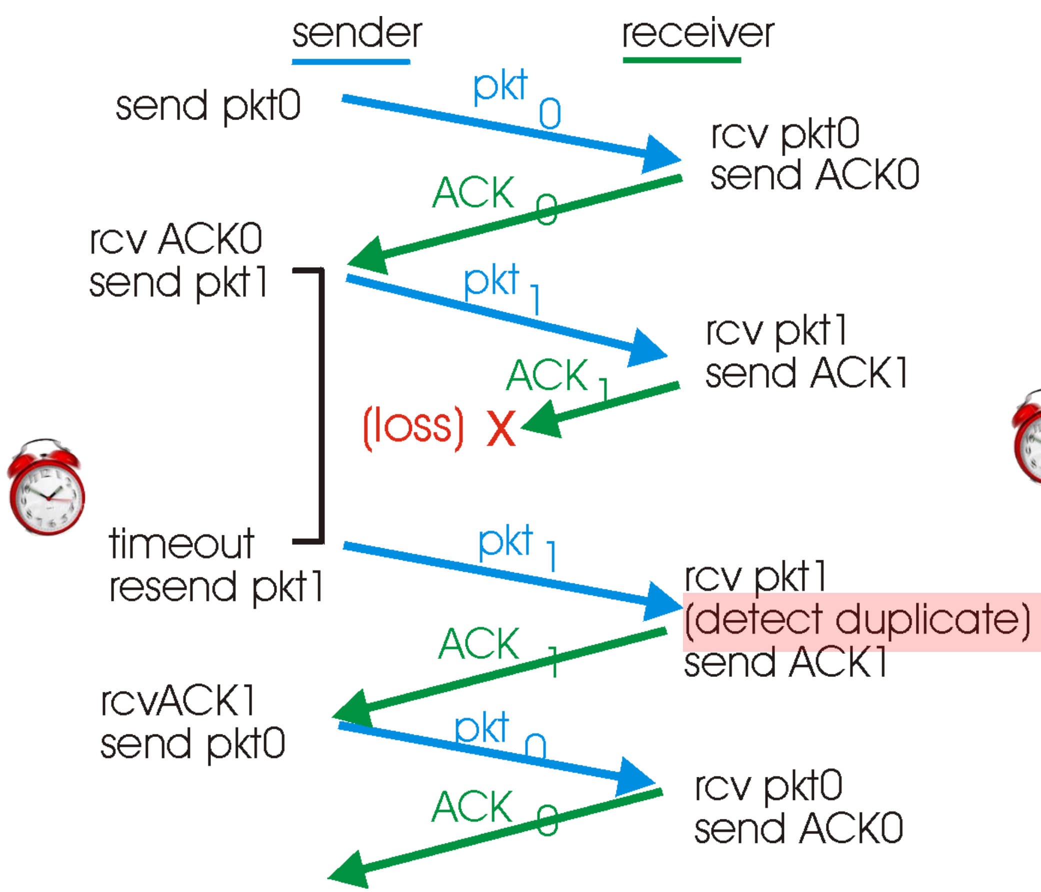
(a) operation with no loss (loss가 없을 때)

이전 패킷과 구분하기 위해 0과 1을 번갈아가며 보냄.

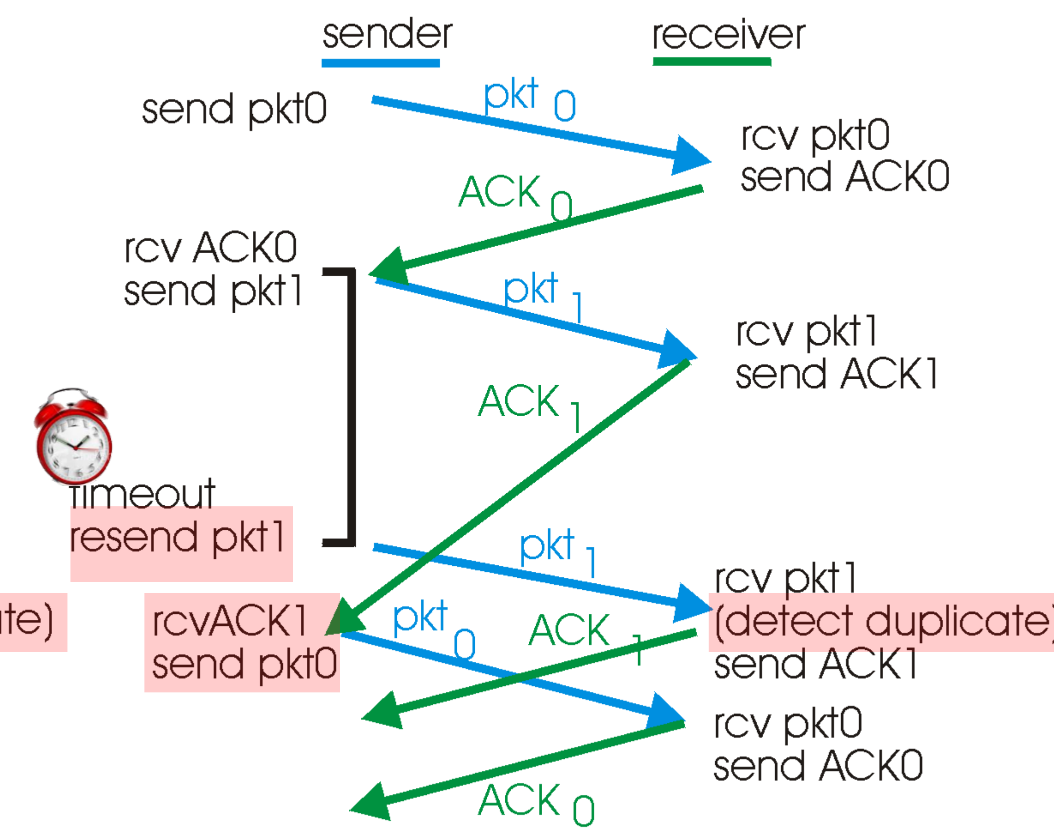


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

duplicate된 패킷을 받음. sequence number(0과 1)로 재전송 패킷인지 알 수 있다. 이미 리시버는 pkt1을 받은 상태니 중복되어 무시하고 재전송 한 뒤 ...

timeout 나서 loss로 가정한다. 중복된 것은 무시한다. ack1 받았으니 pkt0 보낸다.

Sequence Numbers

what happens if ACK is lost or arrives late?

- Sender will retransmit
- But, receiver has received the data packet
  - ***possible duplicate*** at receiver!
  - Handling duplicate at receiver
    - Is it a **new data packet** or **retransmission?**
- sender adds 1-bit ***sequence number*** to each pkt
- receiver discards (doesn't deliver up) duplicate pkts

handling duplicates:

- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate: data seq. #'s handles this
- Also, ACK may be duplicate
  - pkt (or ACK) just delayed (not lost)
  - receiver must also specify seq # of pkt being ACKed

duplicated 해도 상관 없다.

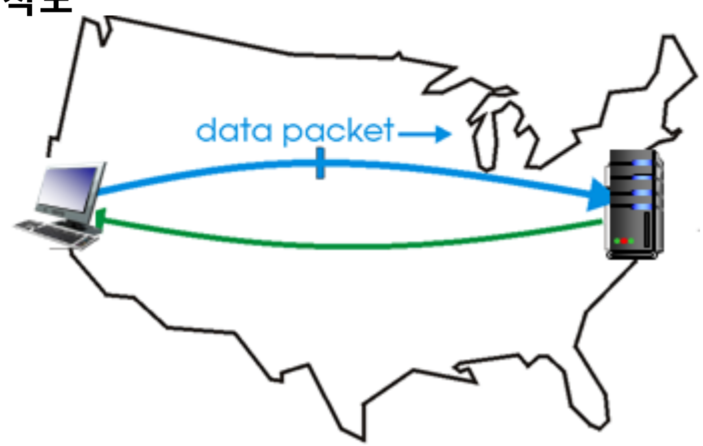
Performance of rdt3.0

- rdt3.0 (stop-and-wait) is correct, but performance not good
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

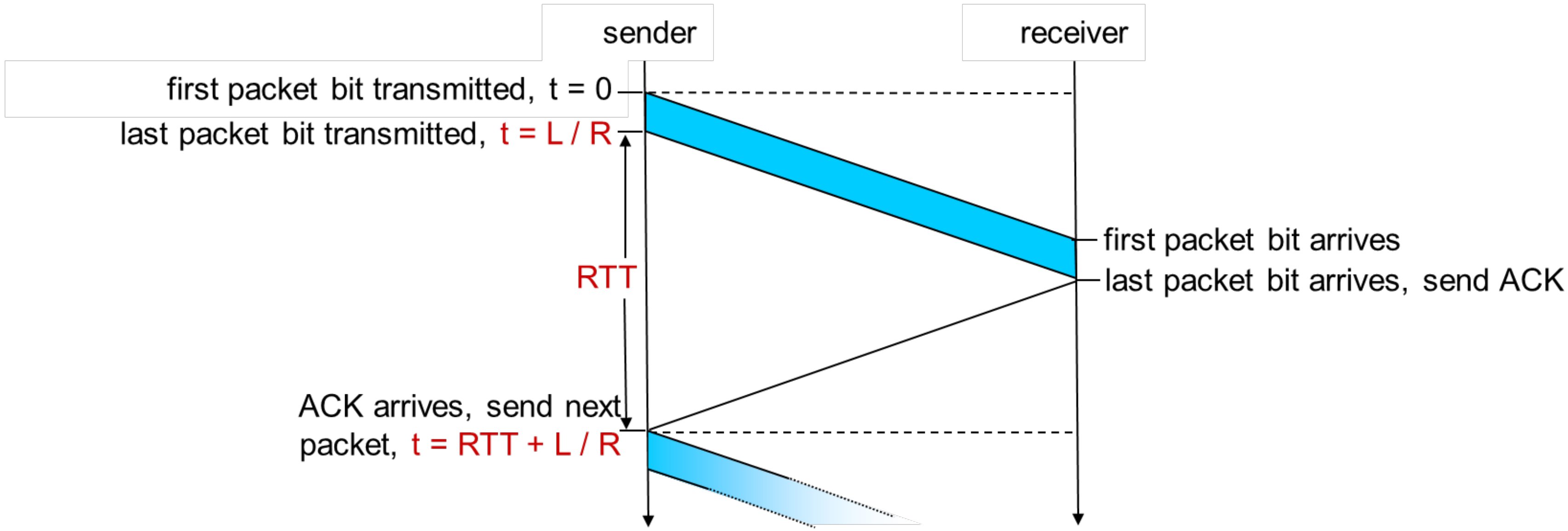
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ us } (= 0.008 \text{ ms})$$

- U $_{sender}$: *utilization* – fraction of time sender busy sending

링크가 얼마나 바쁜지 나타내는 척도



rdt3.0: stop-and-wait operation



- U $_{sender}$: *utilization* : 전체에서 데이터가 보내지는 시간

물리 자원을 잘 못 쓰고 있다

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$ (or just 0.027%)

- if RTT=30 msec, 8000-bit (=1kB) pkt every 30 msec: 267kbps throughput over 1 Gbps link

❖ network protocol (rdt3.0) limits use of physical resources (1Gbps) !

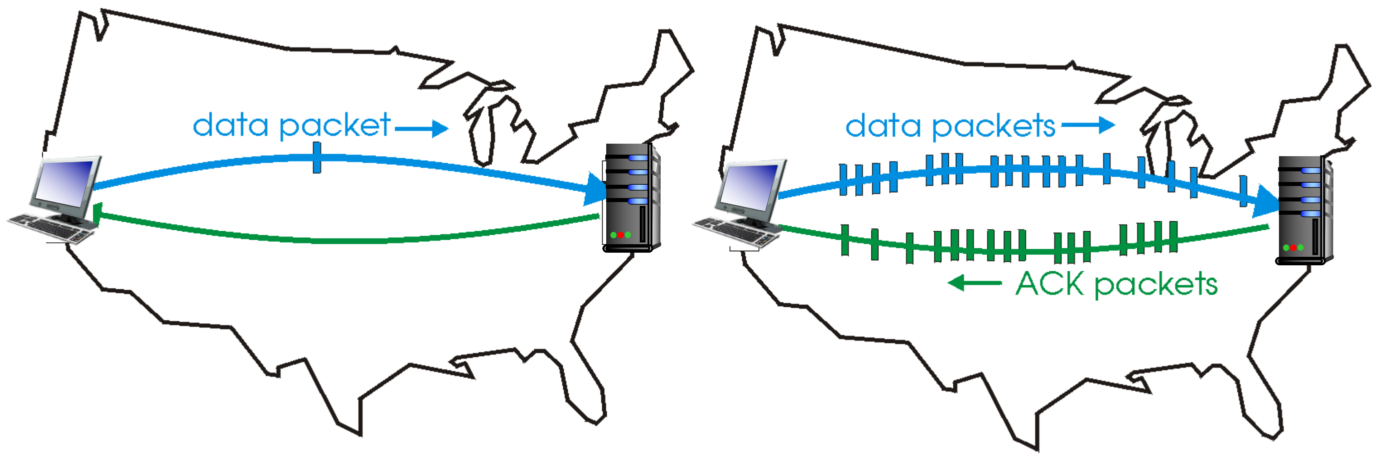
Pipelined protocols

ack을 안 받은 상태에서 다른 것들을 전송하자.

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

- range of sequence numbers must be increased 0과 1이 아닌 더 넓은 range.
- buffering at sender and/or receiver

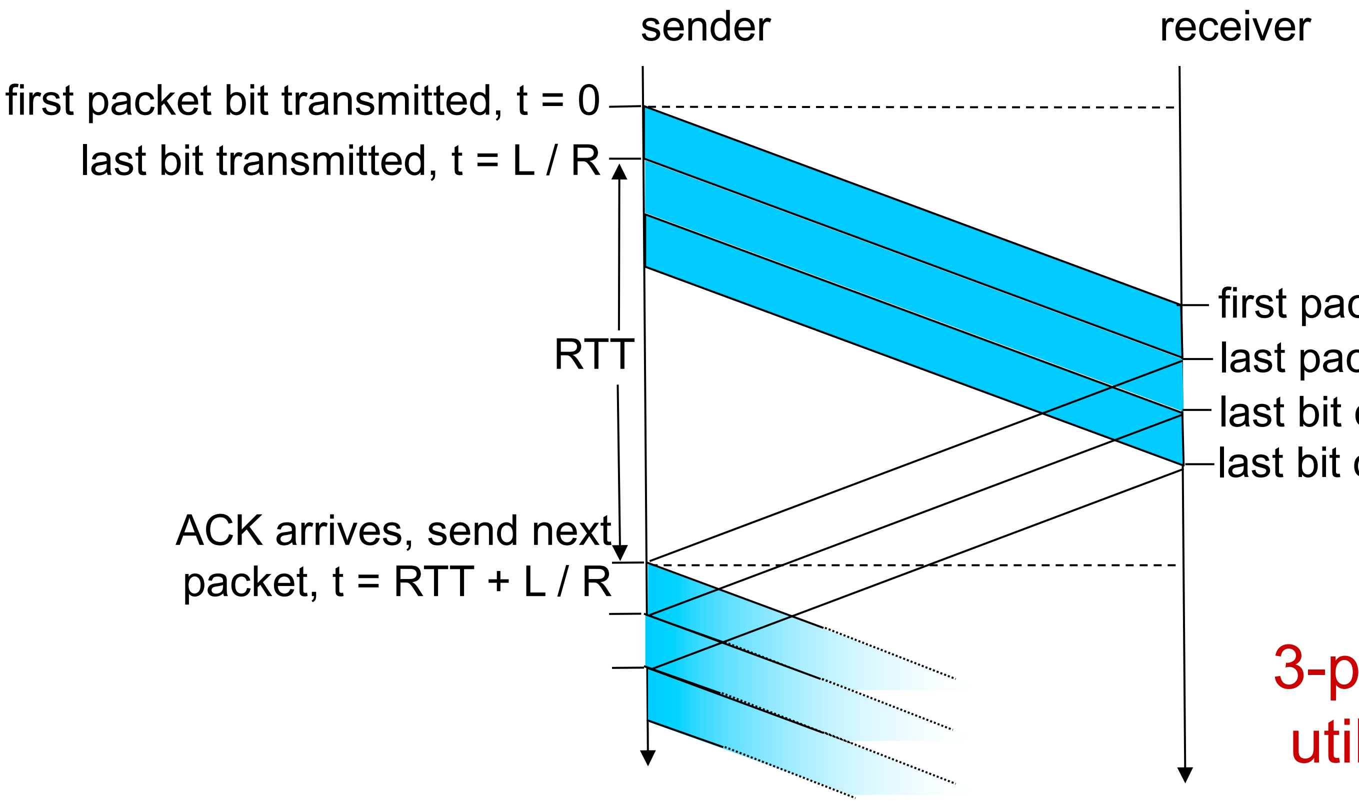
재전송을 할 수 있으니 날리면 안됨. buffering 해야한다.

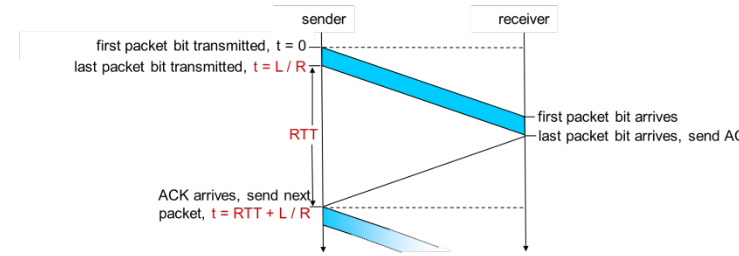


(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

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

Pipelining: increased utilization





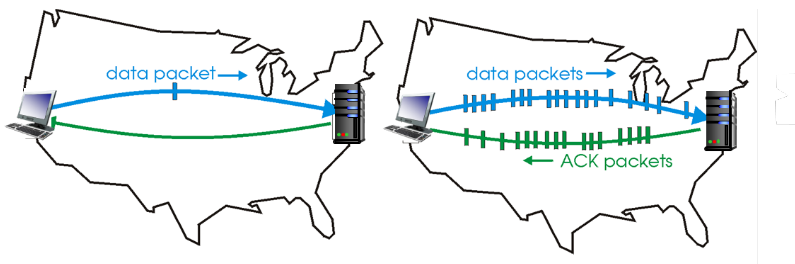
3-packet pipelining increases utilization by a factor of 3!

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

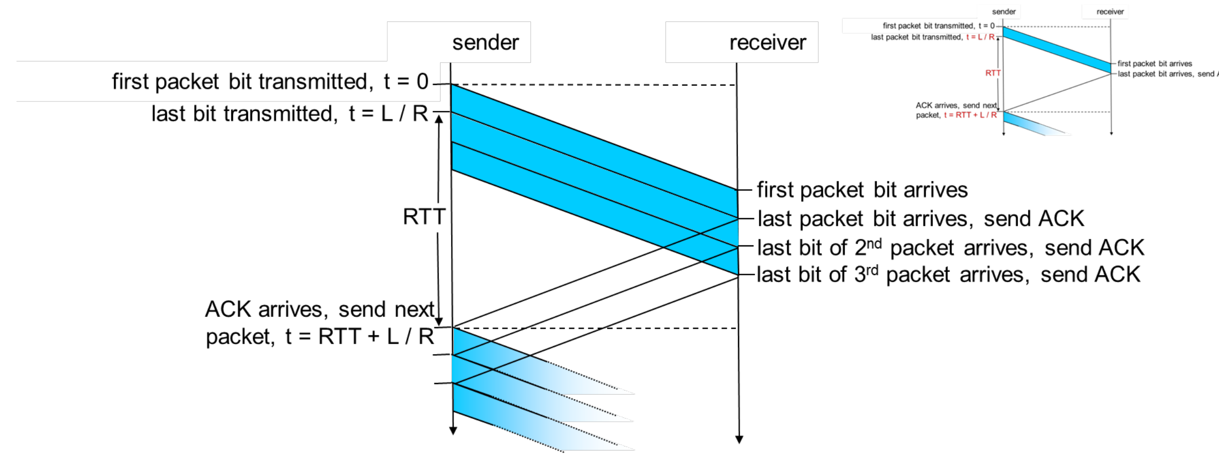
$$= 0.0027N$$

pipeline을 n개 하면 n배 utilization이 올라감. 무조건 늘릴 수 있나? x. 링크가 수용 못할 수 있음. buffer을 넘을 수 있음. receiver가 application layer에 올려야 되는데 network를 통해 들어오는게 많으면 tranport buffer에 쌓이게 됨. 그 이상이면 호스트까지 왔는데 loss가 발생. -> receiver가 소화할 수 없는 양.

Limit of N?



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



- Prev. example, **U = 0.0027N**
  - *Larger N means better utilization*

링크가 혼잡한지 아닌지 알기 어렵고 ...

- Can we increase **N** without any limit?
  - *Answer: No*
  - *Flow Control and Congestion control (Next few chapters)*

Pipelined protocols: Go-back-N (GBN)

Go-back-N:

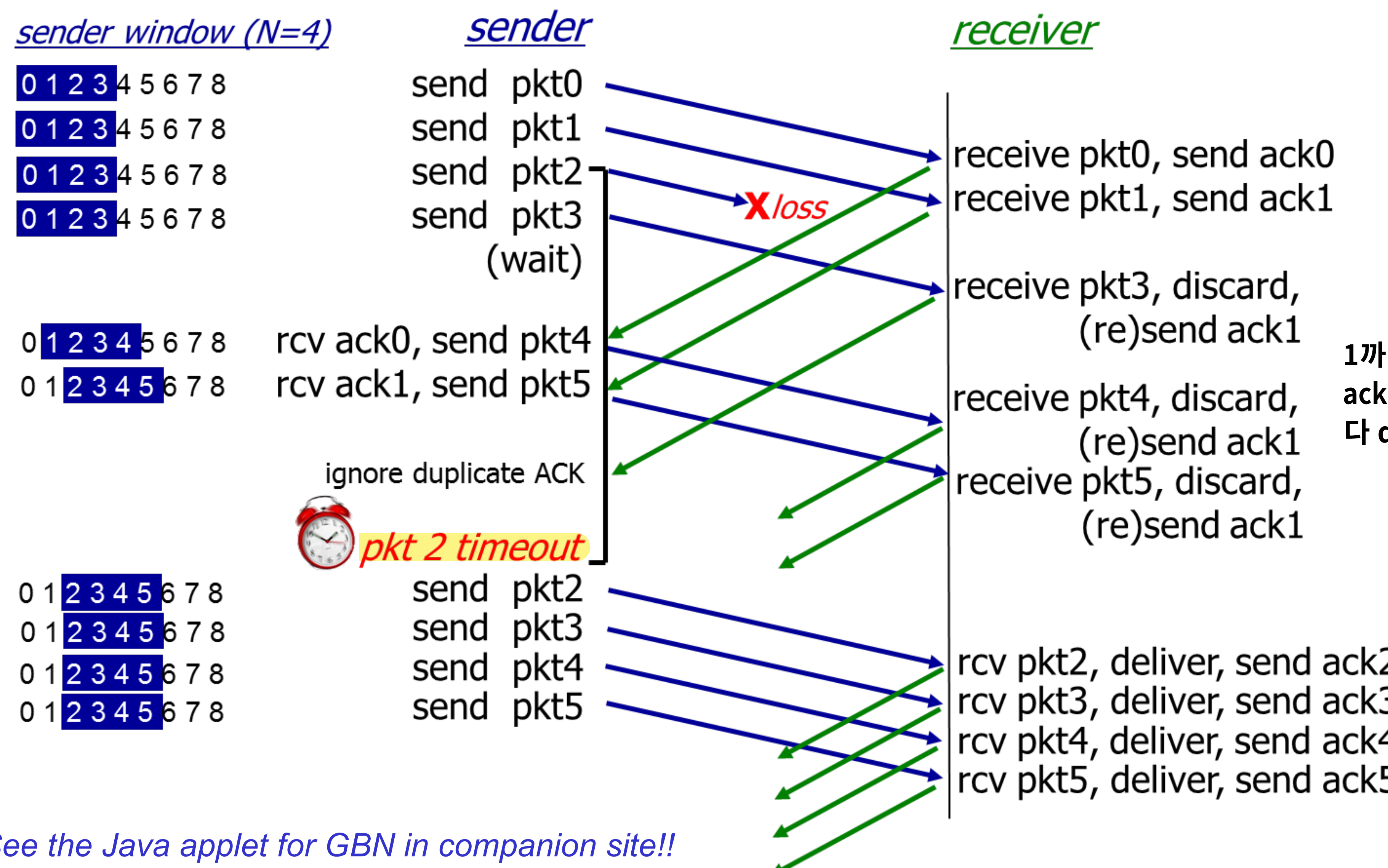
- Sender can send up to **N** (=window size) unacked packets
  - N: window size
  - *k*-bit seq # in pkt header (rtd3.0 had 1-bit)
- base: sender has timer for oldest unacked packet
- Sliding-window protocol

timer을 1개만 두는데 oldest unacked packet에 둔다.

가장 oldest unacked packet이 ack을 받게 되면...

GBN in action

base?

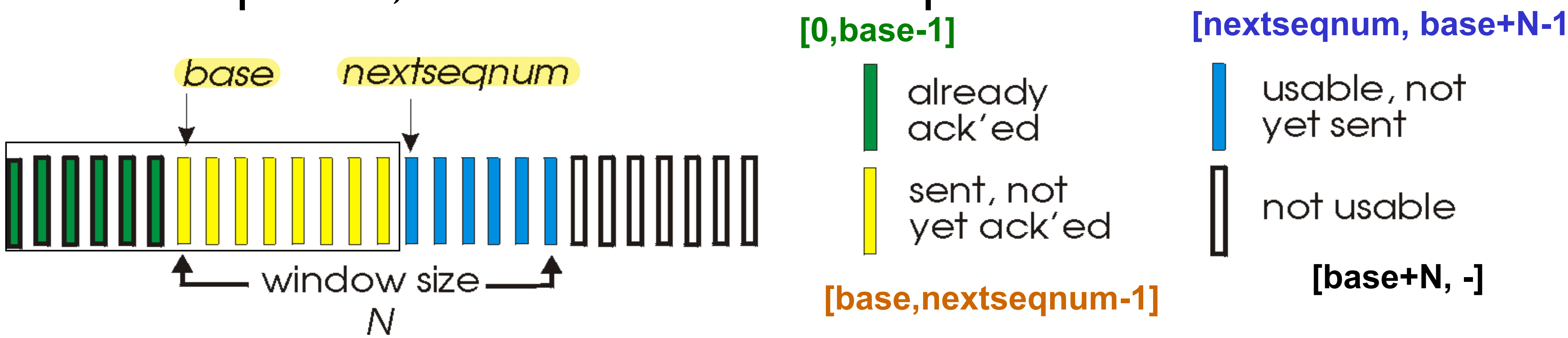


1까지 잘 받았으니
ack1을 보냄.
다 discard 한다.

Note: See the Java applet for GBN in companion site!!

Go-Back-N: sender

- ***window*** of up to ***N***, consecutive unack'ed pkts allowed



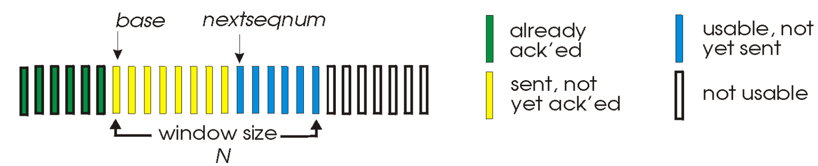
- base: seq# of oldest unacknowledged packet
- nextseqnum: smallest unused seq#
- Seq# uses k bits: seq# range is $[0, 2^k-1]$.
  - TCP uses 32-bit seq#

Go-Back-N: sender

- receiver sends - *"cumulative ACK"*
  - ACK(*n*): all pkts *up to and including n* have been correctly received at receiver – "n까지 잘 받았음" (example: ACK loss)
  - Loss happens: receiver sends duplicate ACKs – "n까지 잘 받았는데 다음은 못 받았음"

- Timeout event
  - When timeout occurs, retransmit *all* unacked packets
  - Single timer for oldest in-flight pkt (=base)

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)
```

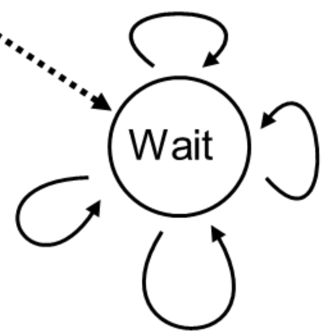
Send up to *N* unacked packets

Set timer for oldest (base) packet

Λ

---

base=1
nextseqnum=1



timeout

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

Retransmit all sent unacked packets

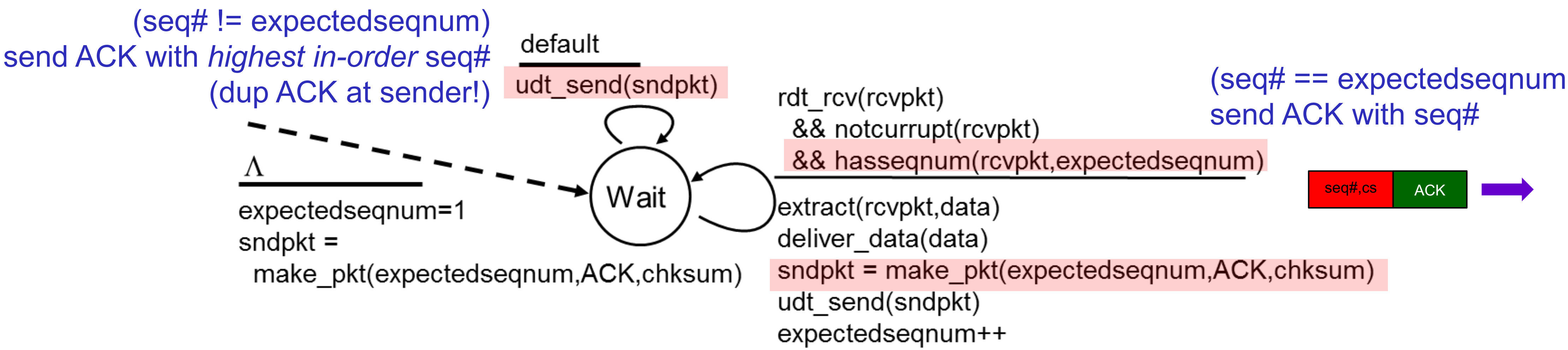
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

---

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

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

GBN: receiver extended FSM



- If data packet *n* is received correctly, and is *in-order* (=packet n-1 has also been received), then send ACK for packet *n*
- Otherwise (if *out-of-order*), resend an ACK for *most recently received in-order packet*
  - may generate duplicate ACKs
  - need only remember **`expectedseqnum`** (= seq# of next in-order packet)
  - *Out-of-order* packets: discard (don't buffer): *no receiver buffering!*
    - Why not just store out-of-order pkts? Simpler receiver; no need to buffer out-of-order pkts, they will be retransmitted by GBN sender anyway.

Pipelined protocols: Selective Repeat (SR)

불필요한 재전송이 늘어난다는 단점이 있어서 이 알고리즘을 사용 가능.

Selective Repeat:

- sender can have up to **N** unack'ed packets in pipeline (same as GBN)
- receiver sends *individual ack* for each packet
- sender maintains timer for *each* unacked packet
  - when timer expires, retransmit only that unacked packet

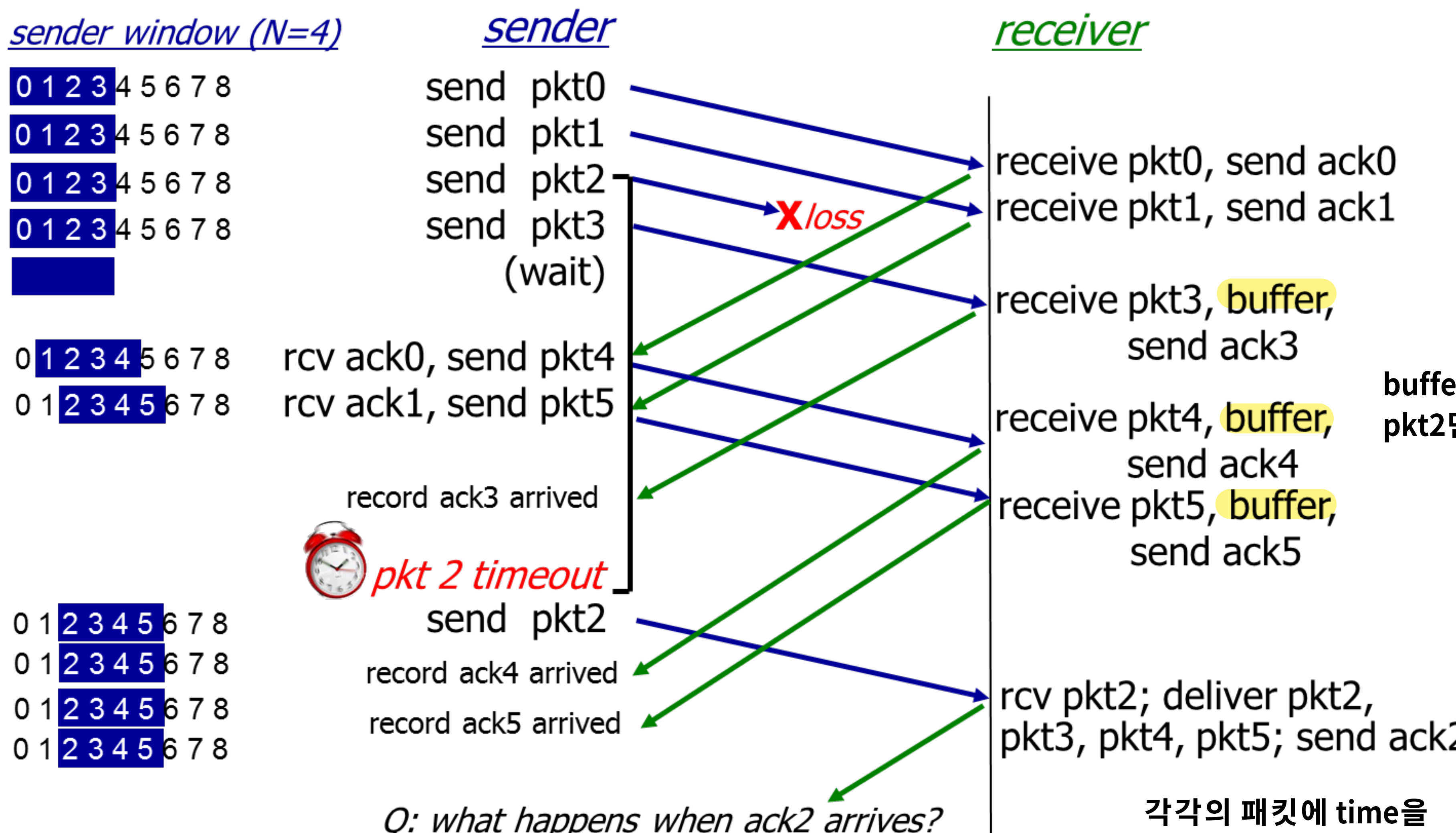
- cumulative ack이 아님.

Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - *N* consecutive seq #'s
  - limits seq #s of sent, unACKed pkts

Selective repeat in action

만약 loss가 발생되면 그것을 다시 보내고
다음 window로 간다.



buffering 걸어둔다
pkt2만 재전송한다.

각각의 패킷에 time을
재어야되고, 복잡성이 올라가
memory 높아짐.

Note: See the Java applet for GBN in companion site!!

Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend *only* pkt *n*, restart timer (compare GBN sender)

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

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

receiver

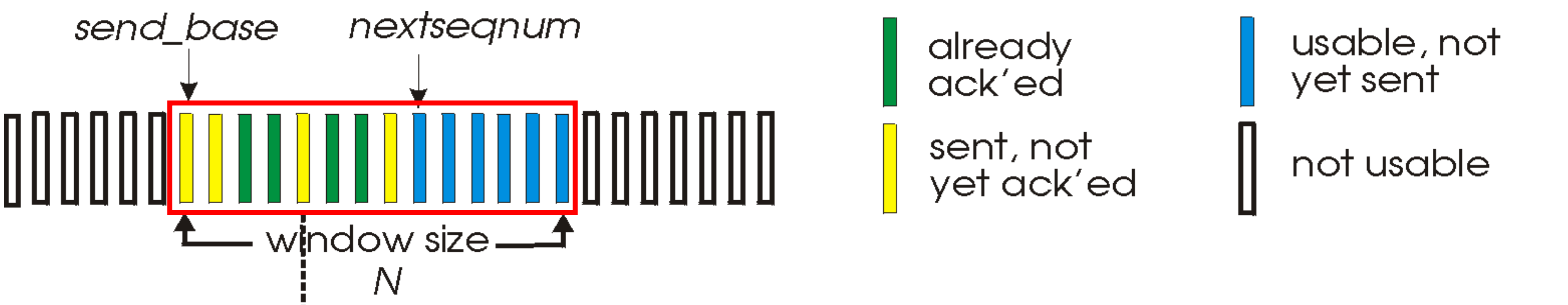
pkt n in [rcvbase, rcvbase+N-1]

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

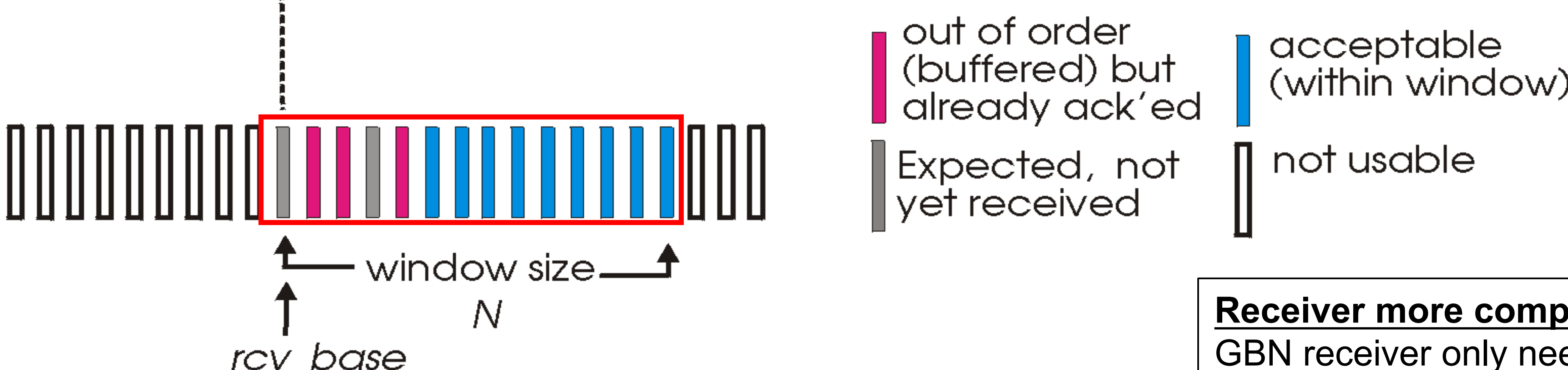
otherwise:

- ignore

Selective repeat: sender, receiver windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Receiver more complex!
GBN receiver only needed to remember **`expectedseqnum`**

Reliable Data Transfer: Summary

- Stop-and-wait
  - rdt 2.0: bit errors (but no packet losses):
  - rdt 3.0: loss (drop) of packets:
- Better utilization: GBN
- Efficient retransmission: Selective Repeat
- Next step: TCP!