Data Communication and Computer Networks

4. Transport Layer

PART-A

Dr. Aiman Hanna

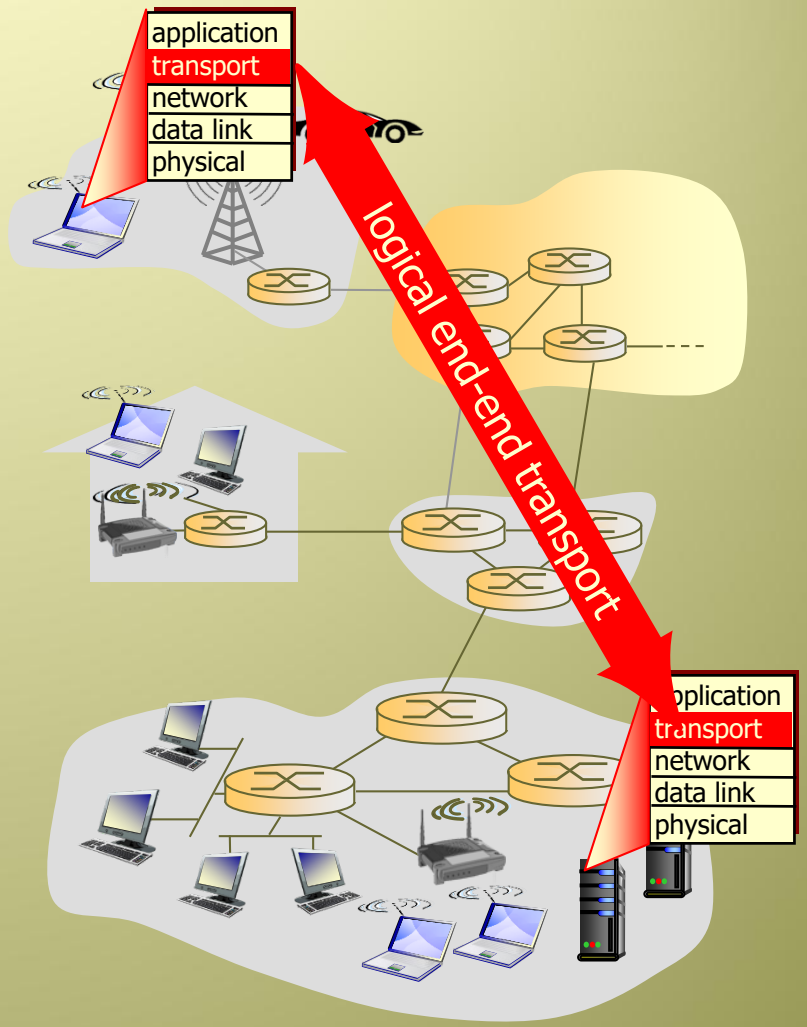
Department of Computer Science & Software Engineering
Concordia University, Montreal, Canada

These slides has mainly been extracted, modified and updated from original slides of :
Computer Networking: A Top Down Approach, 6th edition Jim Kurose, Keith Ross
Addison-Wesley, 2013

Additional materials have been extracted, modified and updated from:
Understanding Communications and Networking, 3e by William A. Shay 2005

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network 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
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

household analogy:

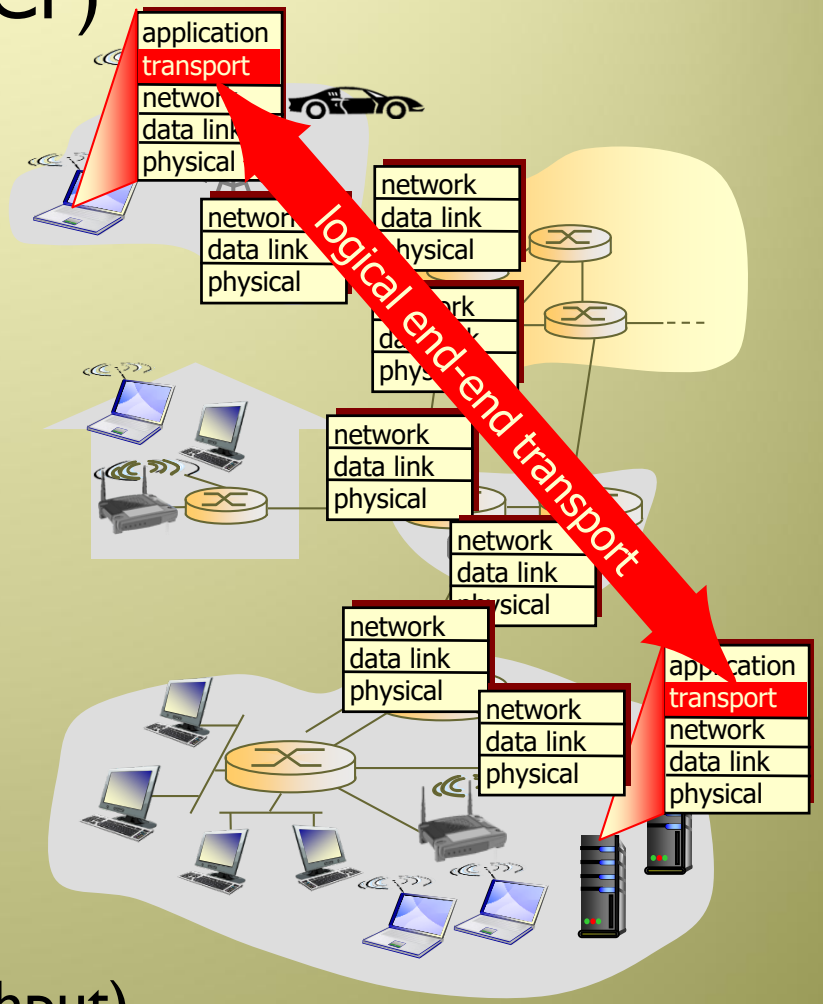
12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings (responsible for collection and distribution)
- network-layer protocol = postal service (move letters from house to house, not from person to person)

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees (timing)
  - bandwidth guarantees (throughput)

➔ Why there are no such guarantees?



UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - delivered out-of-order to app
  - lost
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP is used by:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - RIP

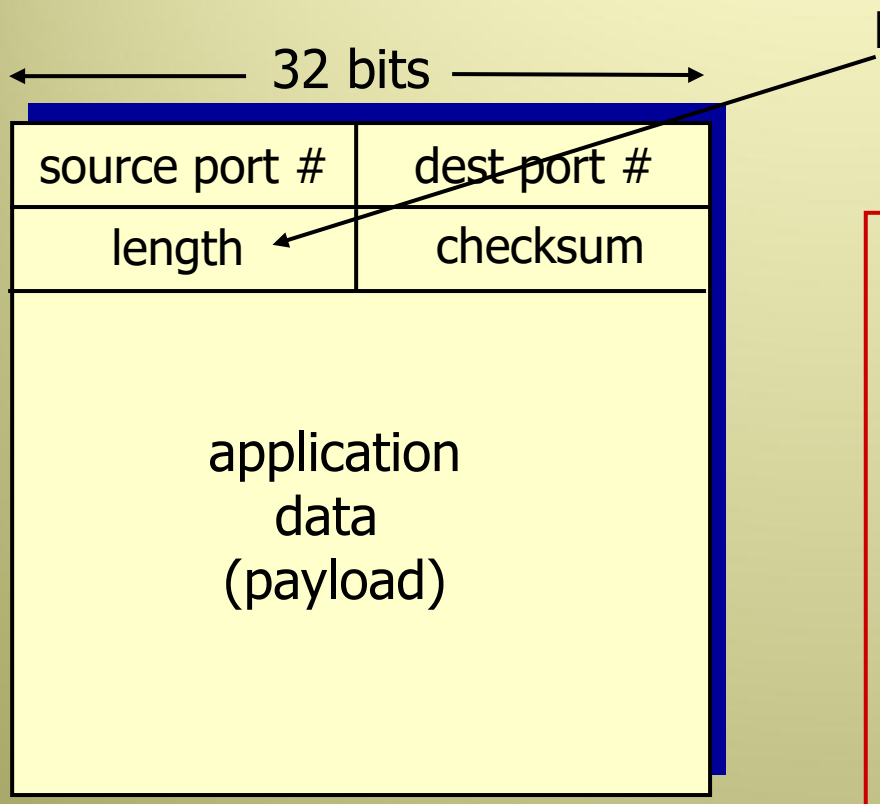
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP

why is there a UDP?

- finer application-level control over what is sent and when
- no connection establishment (which can add delay)
- no congestion control: UDP can blast away as fast as desired
- no need for retransmission to achieve reliability (not needed in many cases)
- simple: no connection state at sender, receiver
- small header size: 8 bytes of header overhead compared to 20 bytes with TCP

UDP: segment header



UDP segment format

- data field may contain for instance a DNS request or query, or streaming data
- length indicates the amount of attached data since it varies
- Checksum is used to perform basic error detection

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one' s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute sum of received segment
- check if sum and checksum add to all 1's:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: example

example: add two 16-bit integers

	0	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
sum	0	1	1	1	1	0	1	1	1	0	1	1	1	0	1	1
checksum	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0

Internet checksum: example

- at the receiver, add both numbers to the checksum.
- result should be all 1s; otherwise something has been altered

1st number	0	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0
2nd number	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
checksum	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0
result	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

UDP checksum

why UDP provides error-detection in the first place?

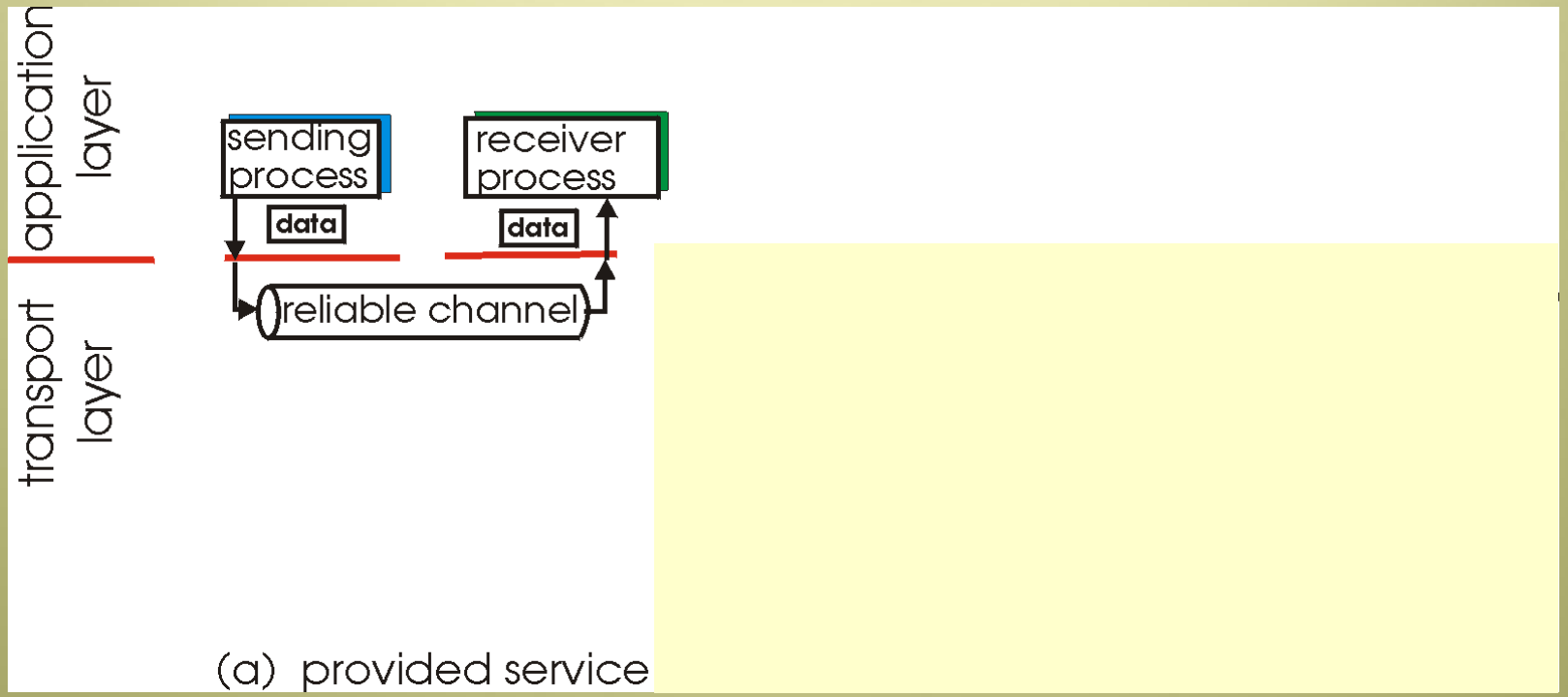
- error-checking is done at the data-link layer, however:
  - at one, or more, of the links in between, the utilized protocol may not provide error checking, which in essence nullifies all the rest of the checking
  - even if all links provide error checking, the packets may get altered while being stored in routers memory

- UDP provides error checking as an **end-end principle**
  - Since the data finally made it to the destination, at least warn the user in case it is altered
  - user may accept it with a warning, or reject it

- Notice that after all UDP does not do anything in case an error has occurred

Principles of reliable data transfer

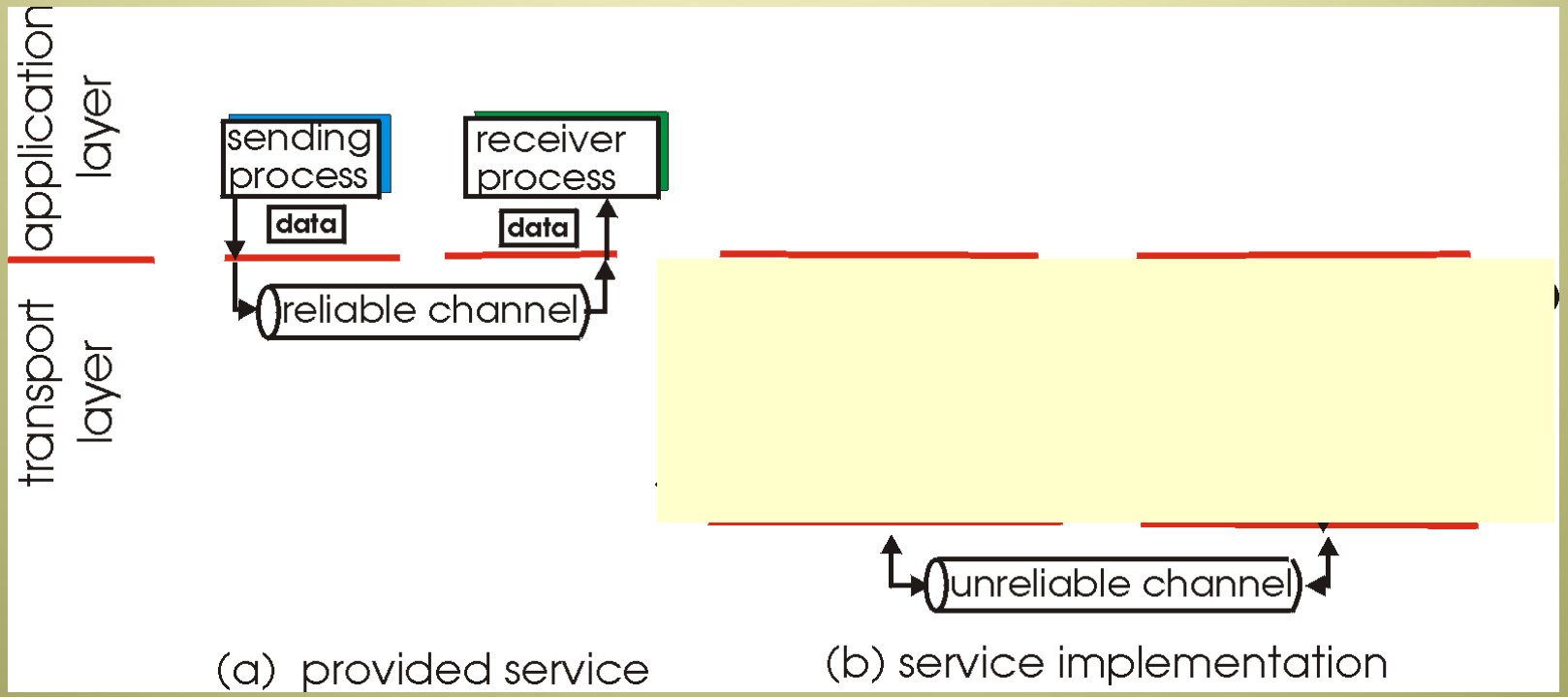
- important in Application, Transport, and Data-Link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

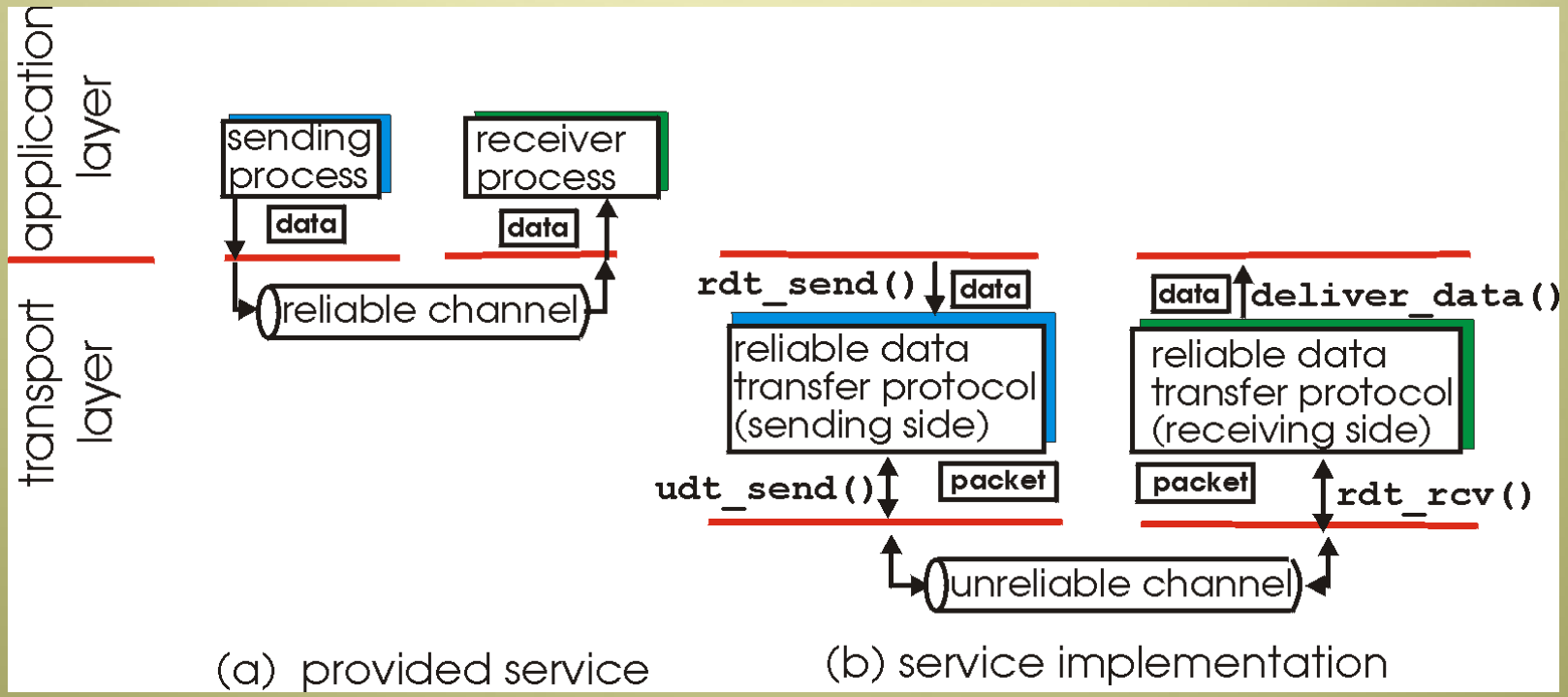
- important in application, transport, link layers
  - top-10 list of important networking topics!



- 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!

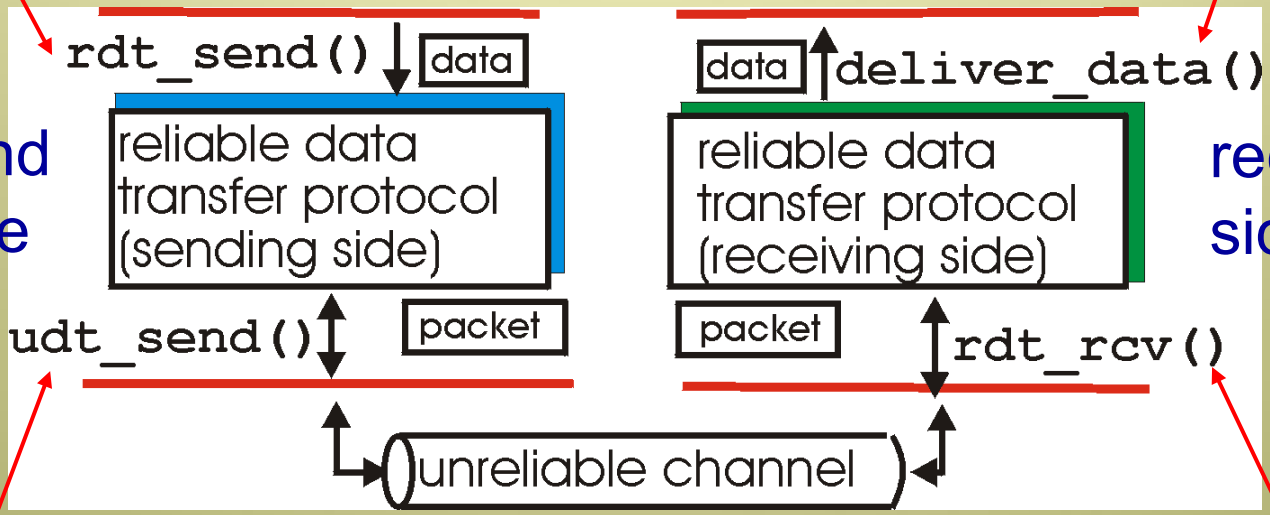


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

`rdt_send()`: called from above, (e.g., by app.). Passed data is to be delivered to receiver upper layer

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



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

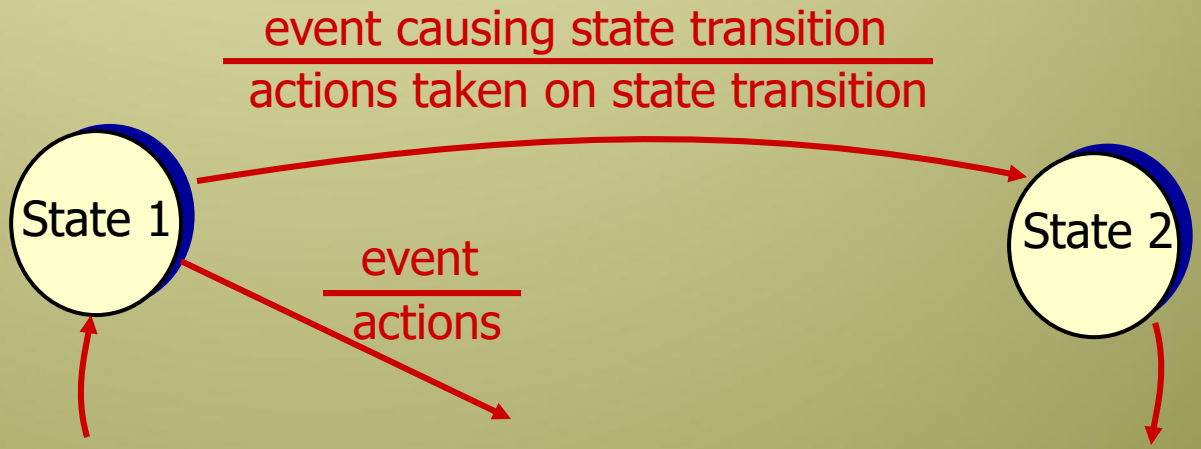
`rdt_rcv()`: called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

we'll:

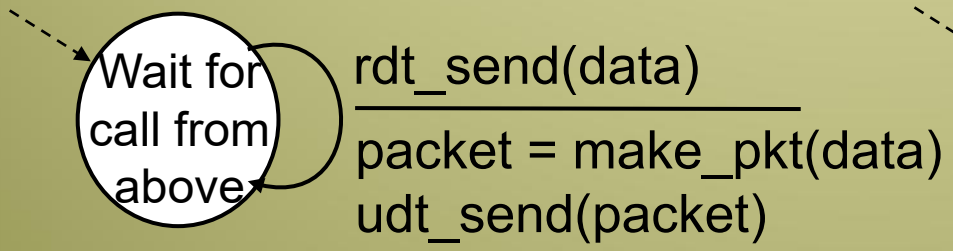
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

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

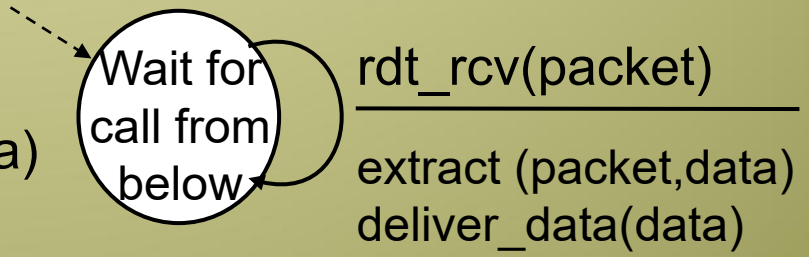


rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
  - no need for receiver to send feedback to the sender
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel
- such protocols are referred to as **Unrestricted Protocols**



rdt 1.0 sending side



rdt 1.0 receiving side

rdt2.0: channel with bit errors

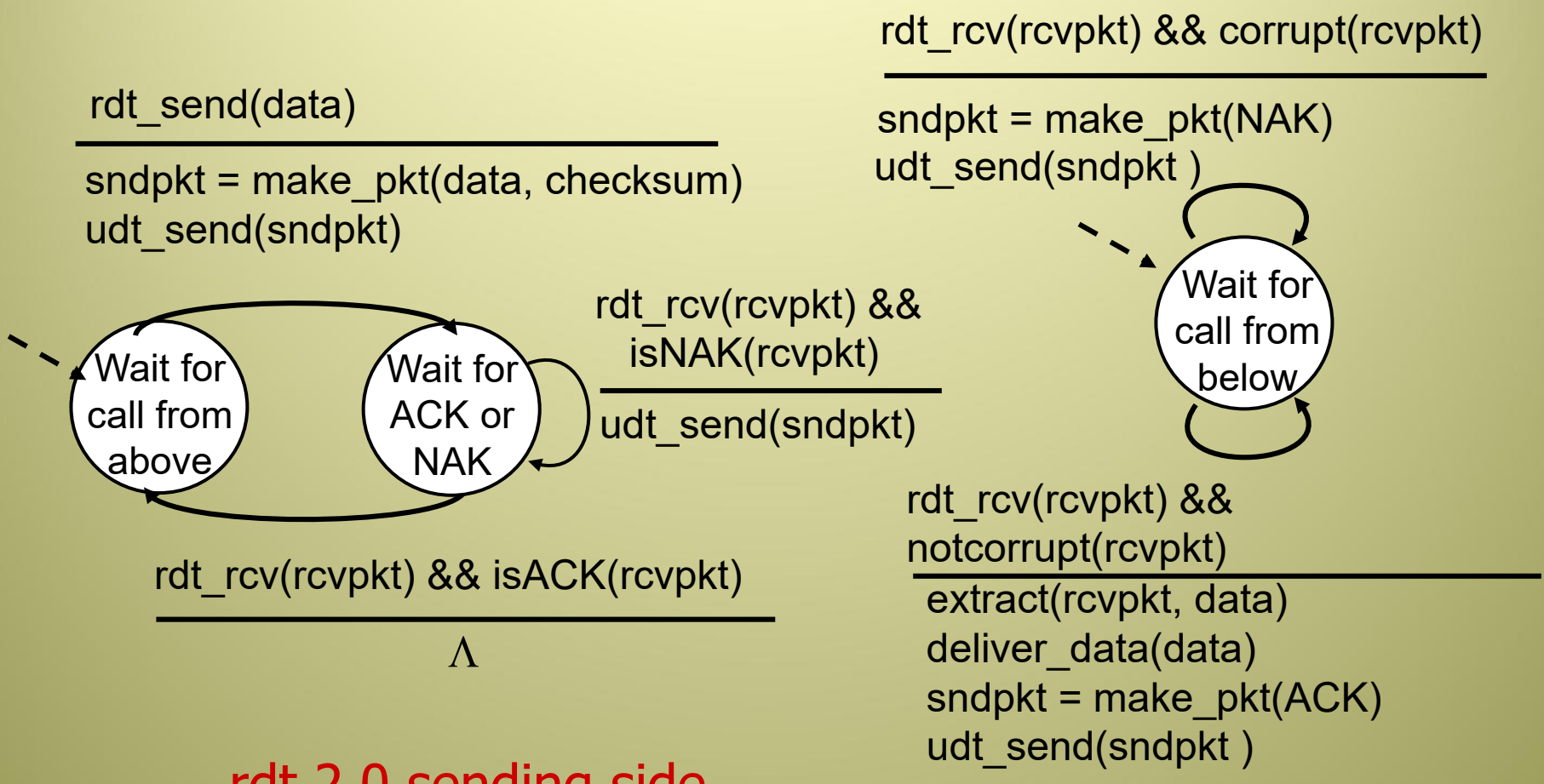
- underlying channel may flip bits in packet
  - checksum to ***detect*** bit errors
  - however, for now we assume **no loss**
- *the* question: how to ***recover*** from errors:

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
- protocols based on ACKs and NAKs retransmission are referred to as **ARQ** (**Automatic Repeat reQuest**) protocols

rdt2.0: FSM specification

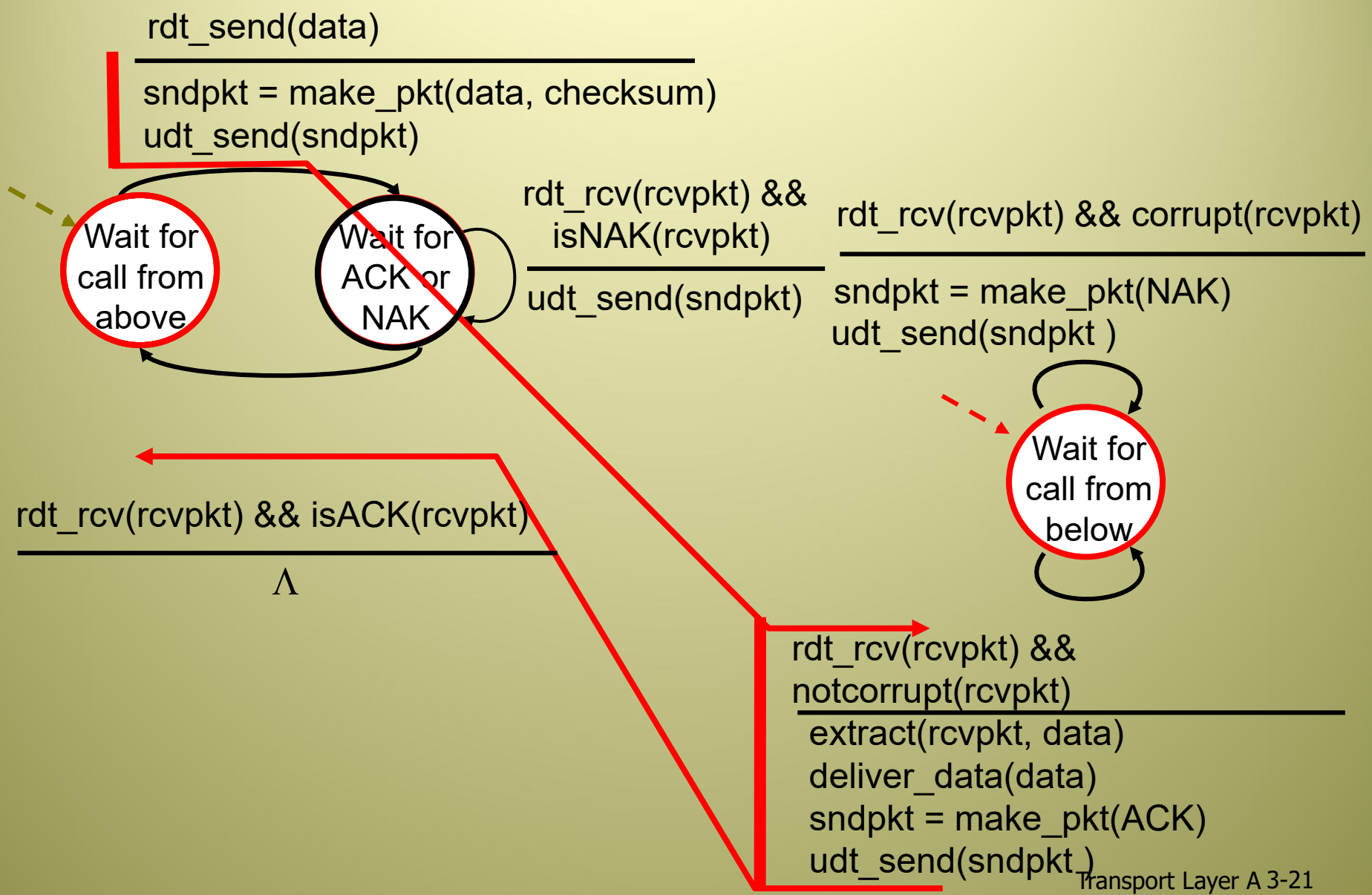


rdt 2.0 sending side

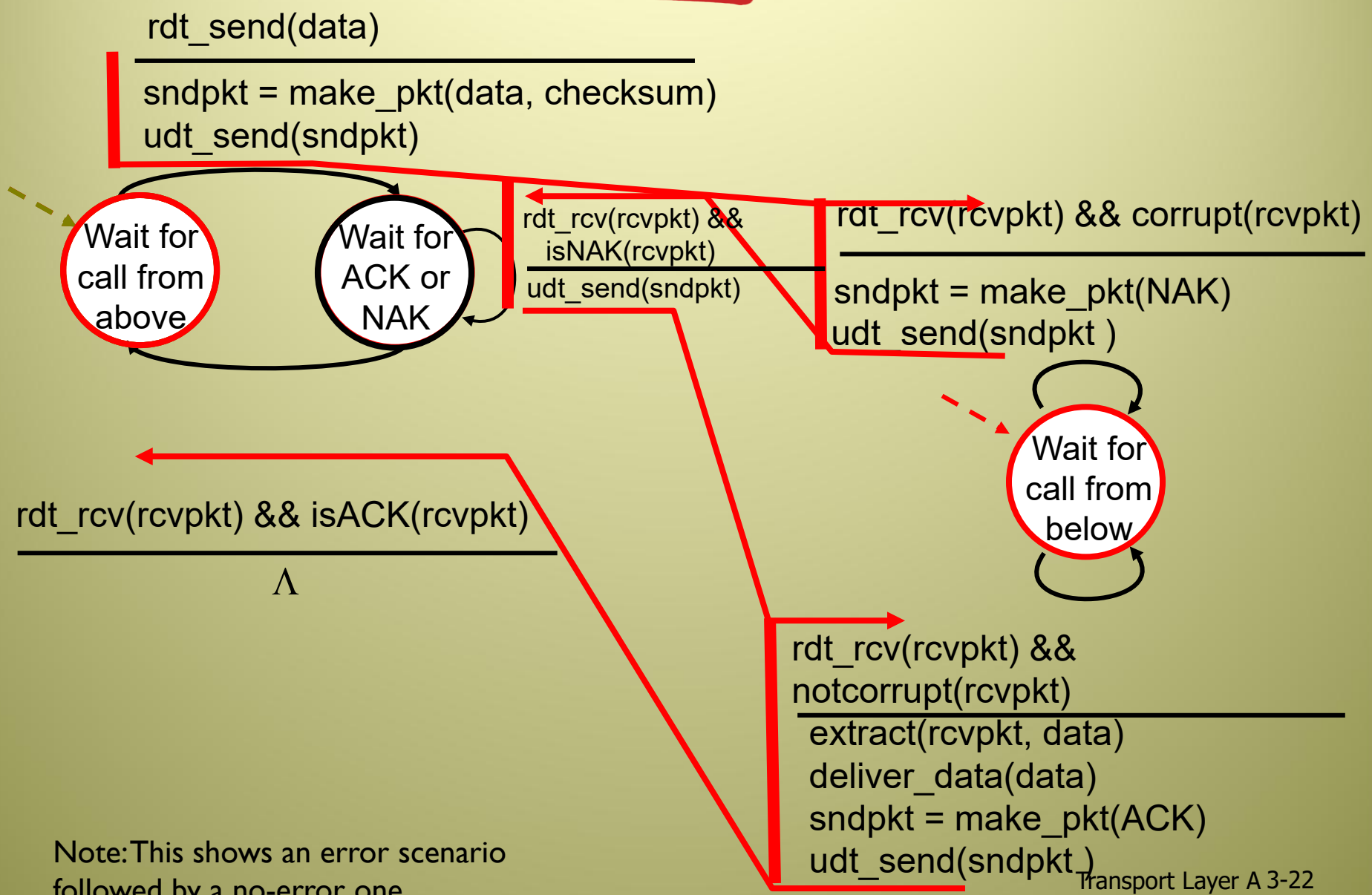
rdt 2.0 receiving side

Λ ≡ no action required

rdt2.0: operation with no errors



rdt2.0: error scenario



Note: This shows an error scenario followed by a no-error one

rdt2.0 has a fatal flaw!

- remember, we still assume no loss is possible, however:

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit to solve the problem!
  - possible duplicates

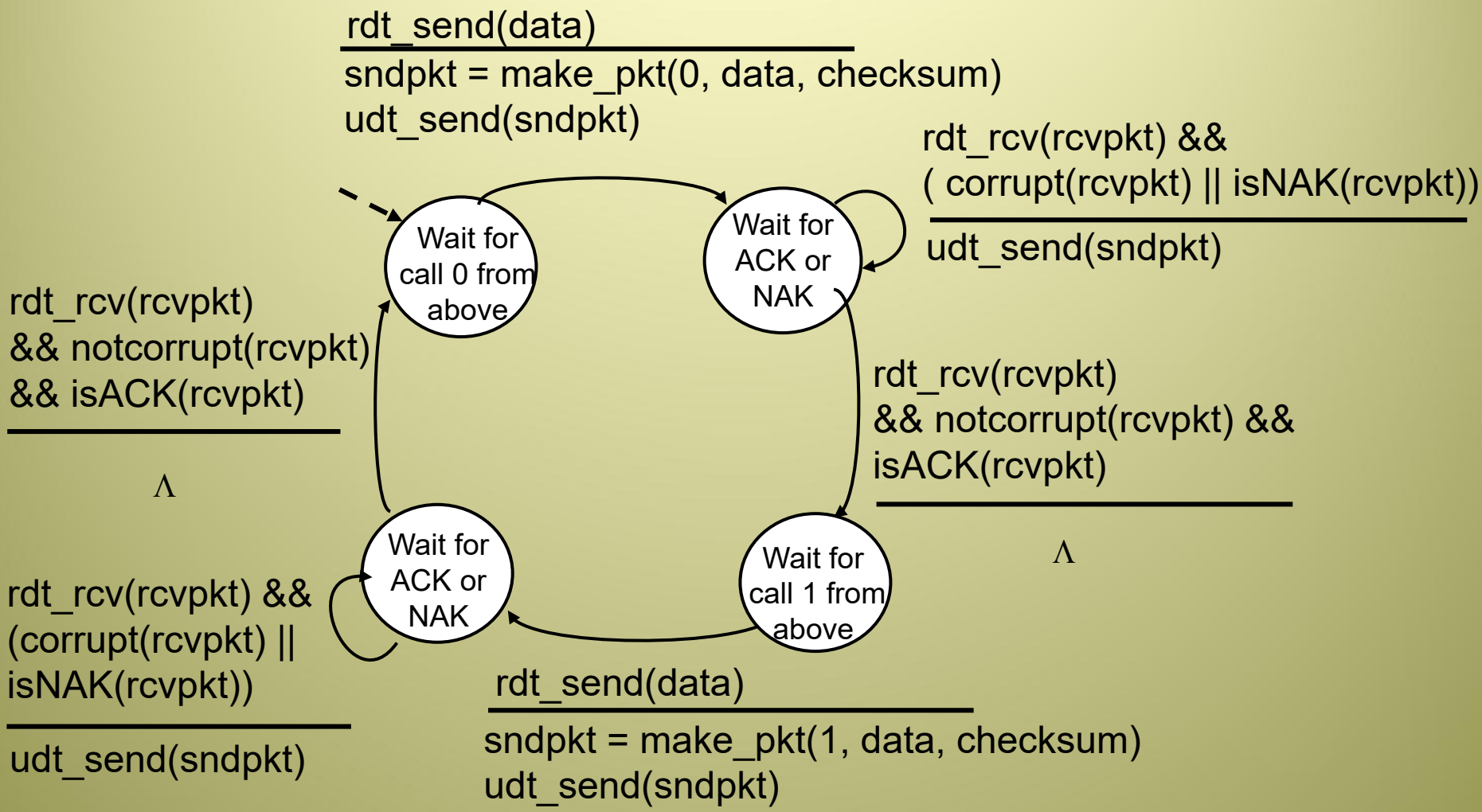
handling duplicates:

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

stop and wait

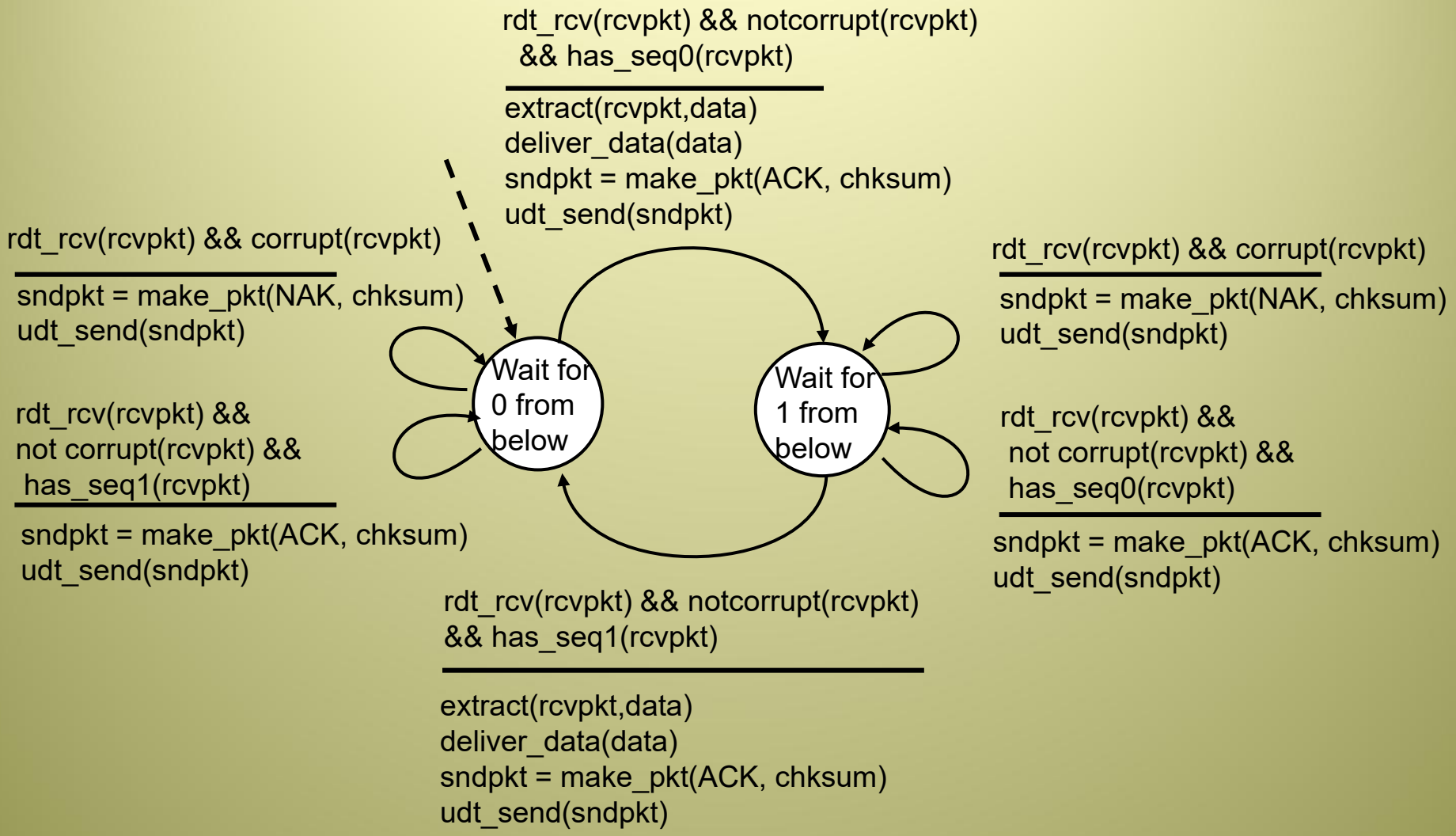
sender sends one packet, then waits for receiver response

rdt2.1: sender, handles garbled ACK/NAKs



rdt 2.1 sending side

rdt2.1: receiver, handles garbled ACK/NAKs



rdt 2.1 receiving side

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

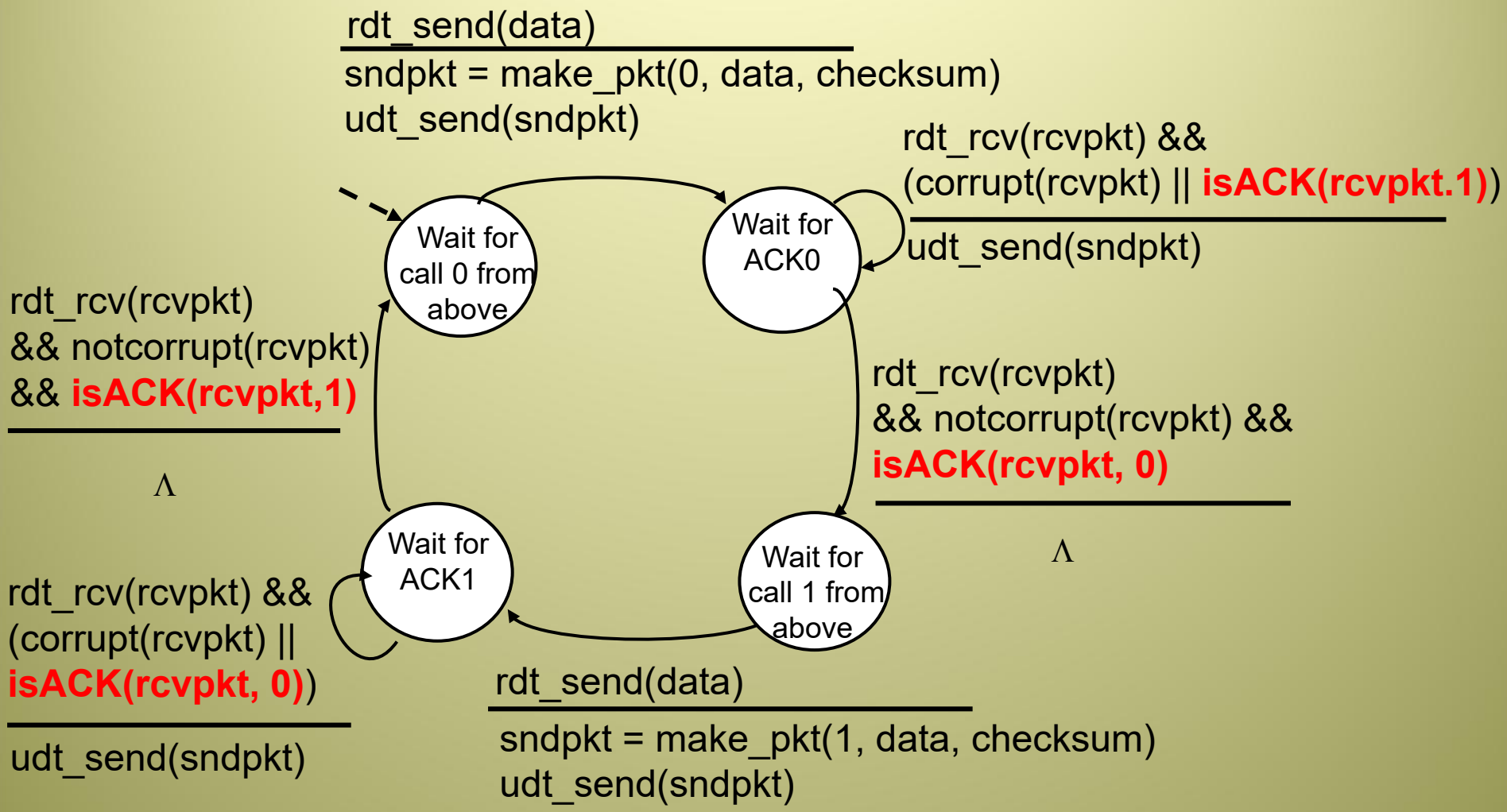
receiver:

- 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

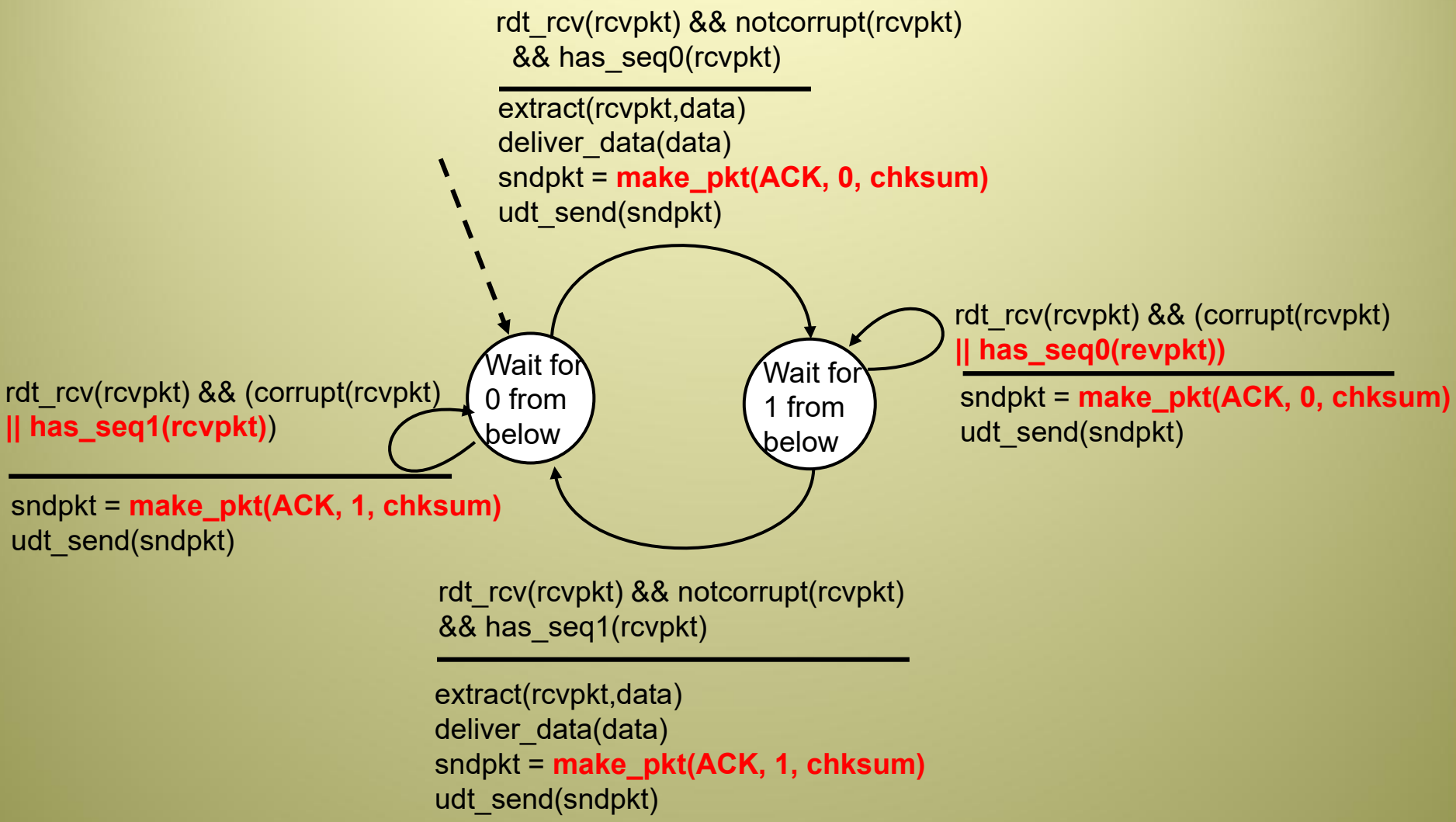
- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: sender



rdt 2.2 sending side

rdt2.2: receiver



rdt 2.2 receiving side

rdt3.0: channels with errors *and* loss

- Previous versions assumed no-loss channels
  - Packets may have errors, but they will always arrive

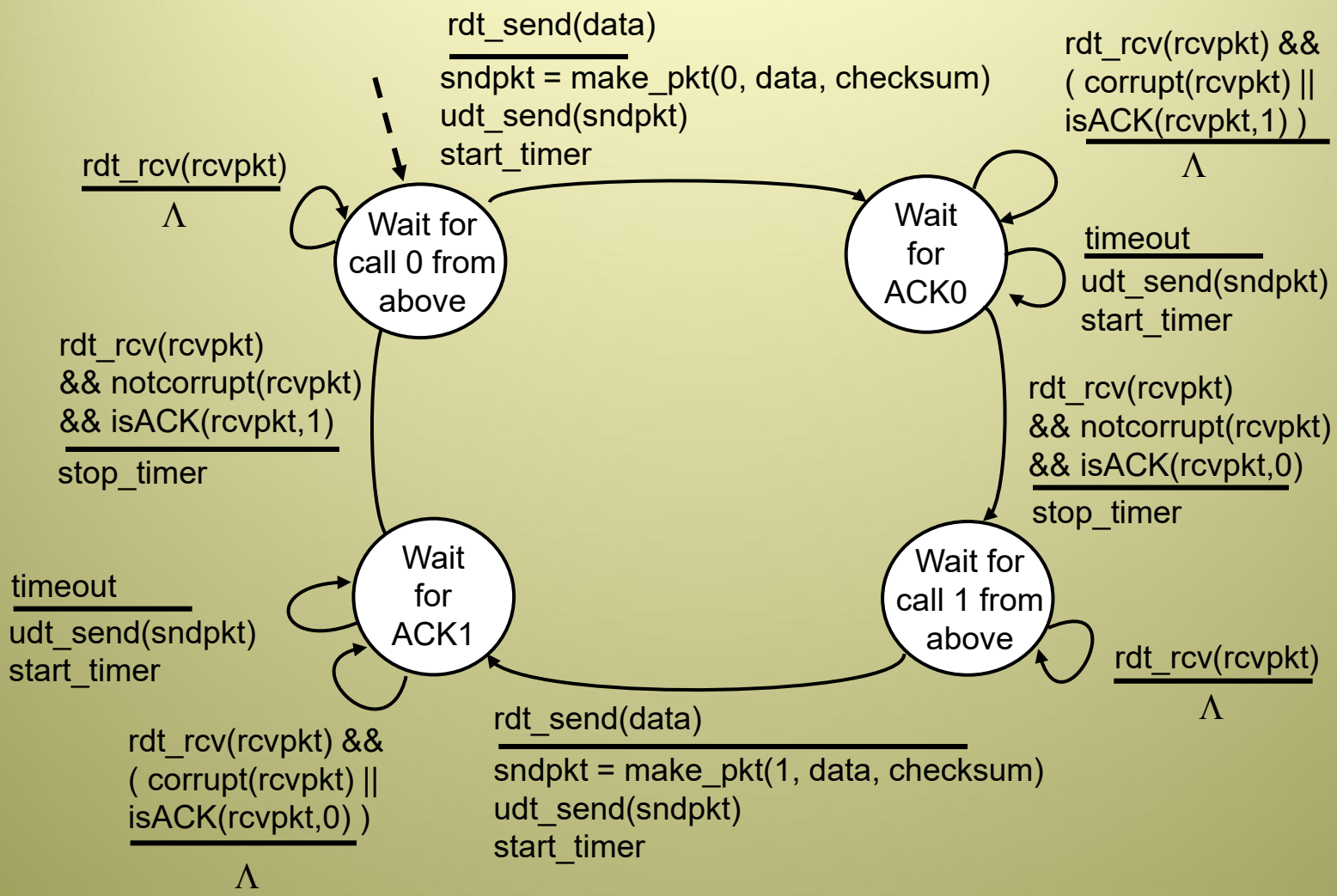
new realistic assumption: underlying channel can also lose packets (data, ACKs)

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

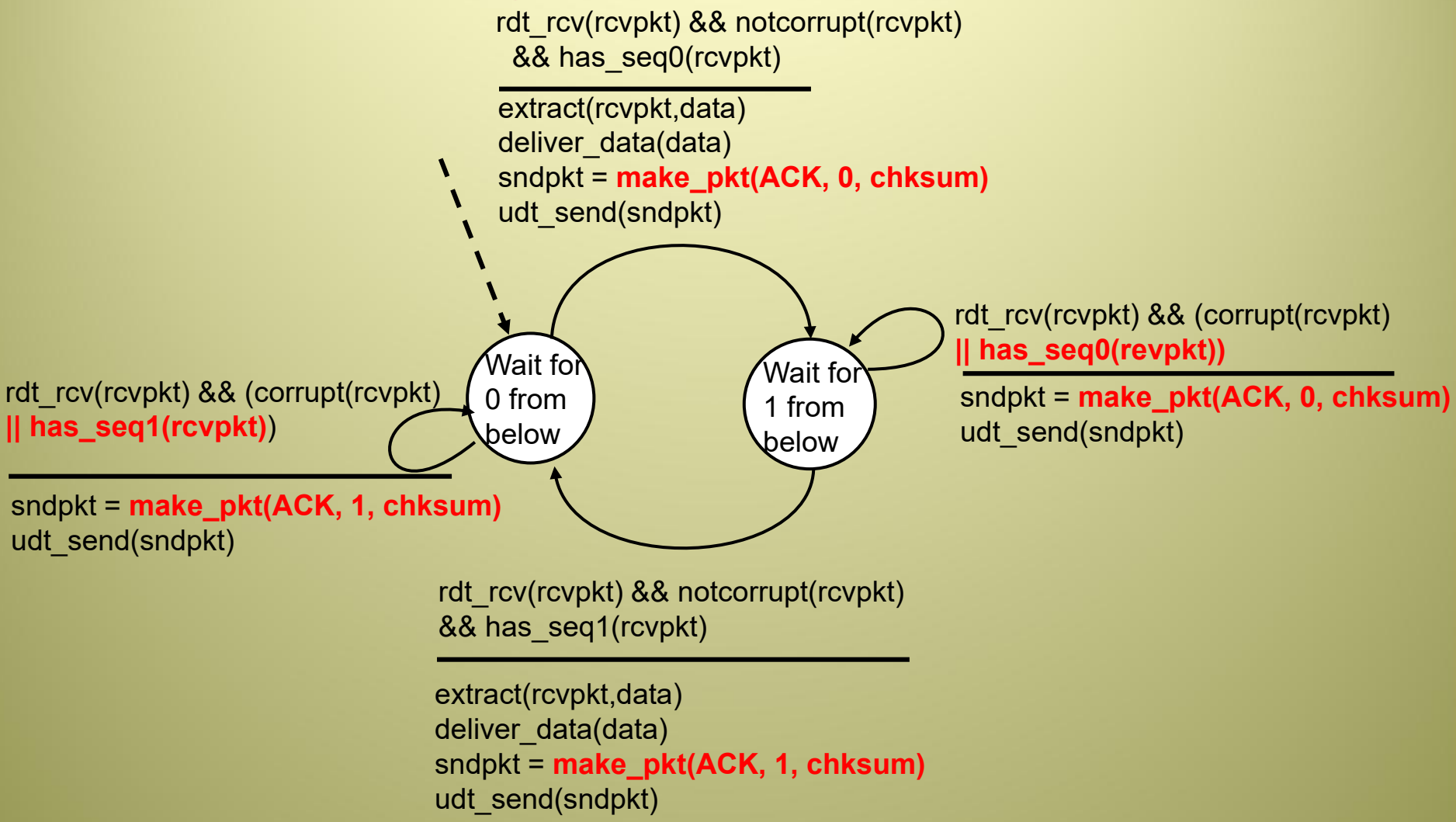
approach: sender waits "reasonable" amount of time for ACK

- 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

rdt3.0 sender

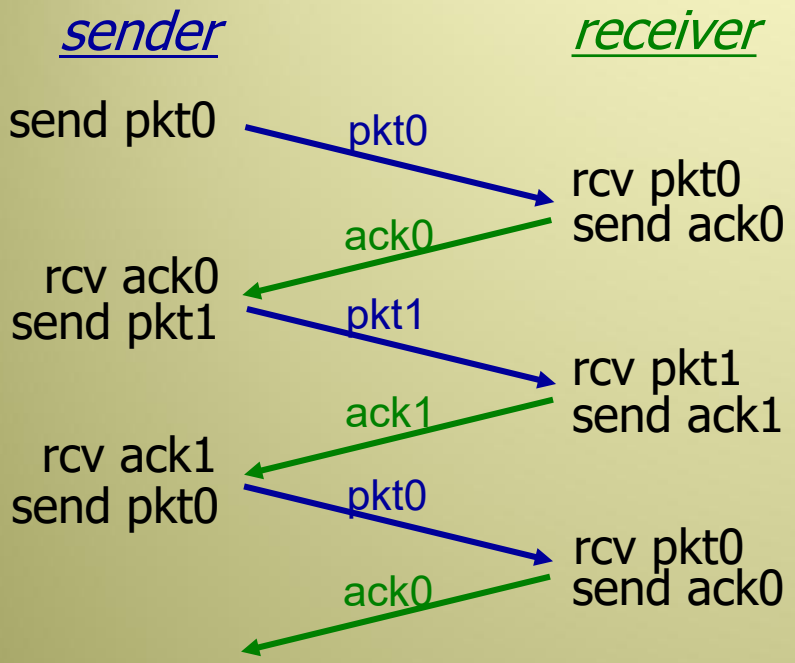


rdt3.0: receiver

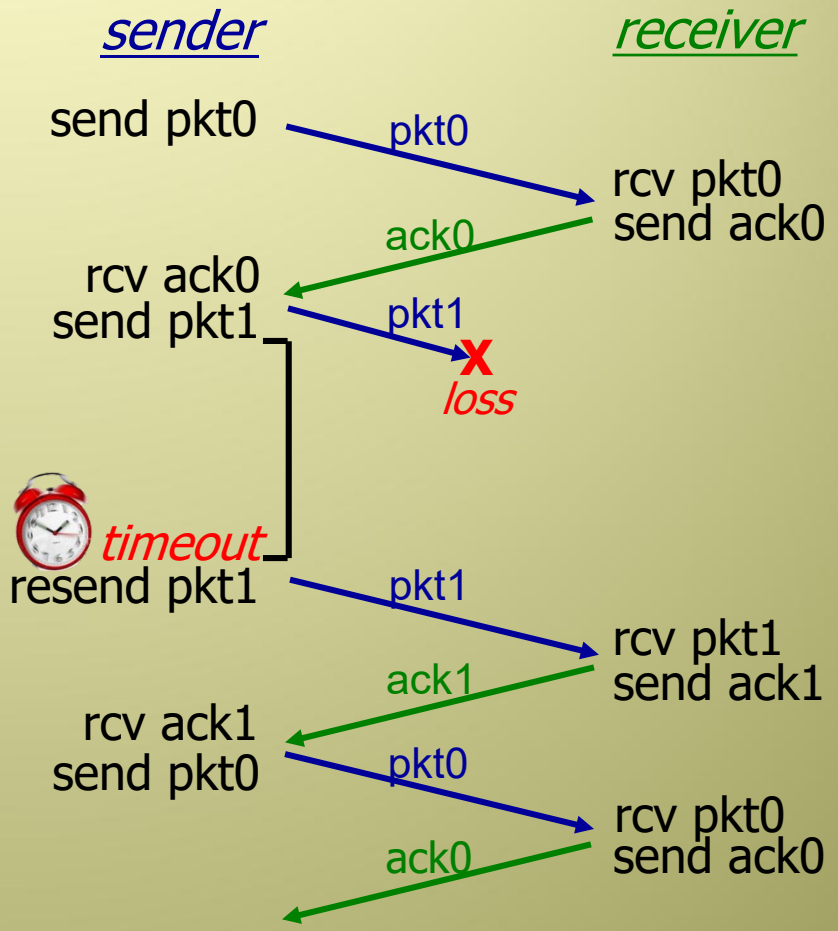


rdt 3.0 receiving side

rdt3.0 in action

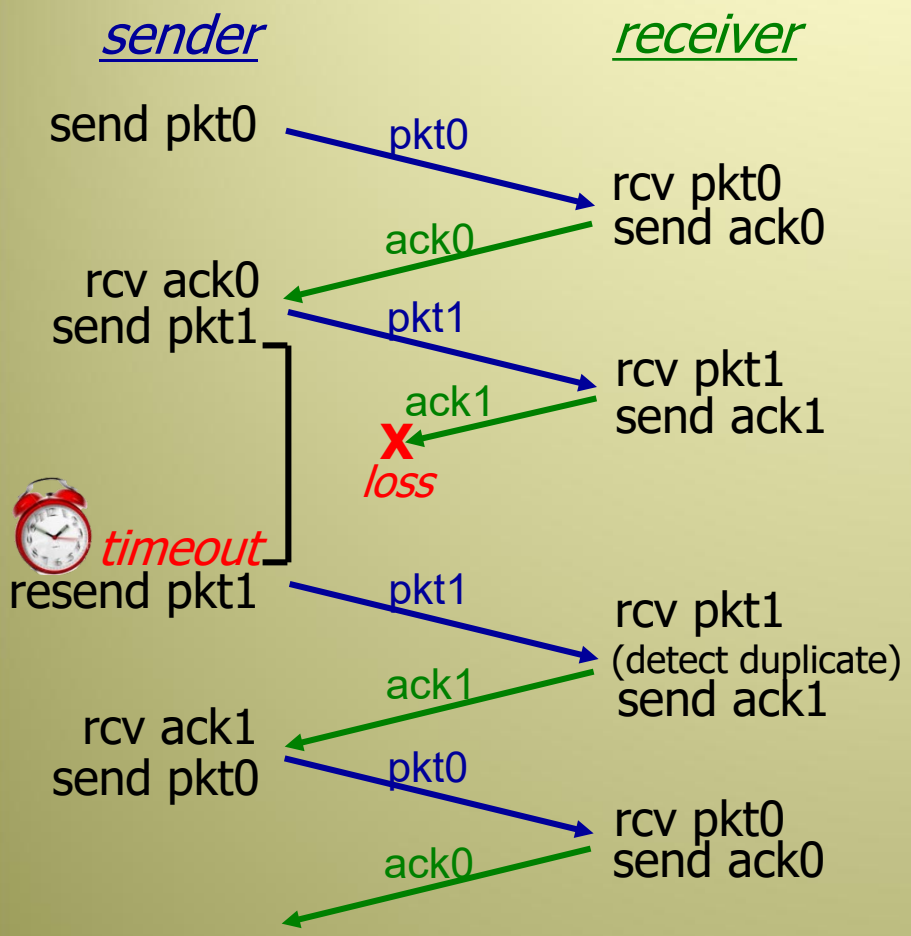


(a) no loss

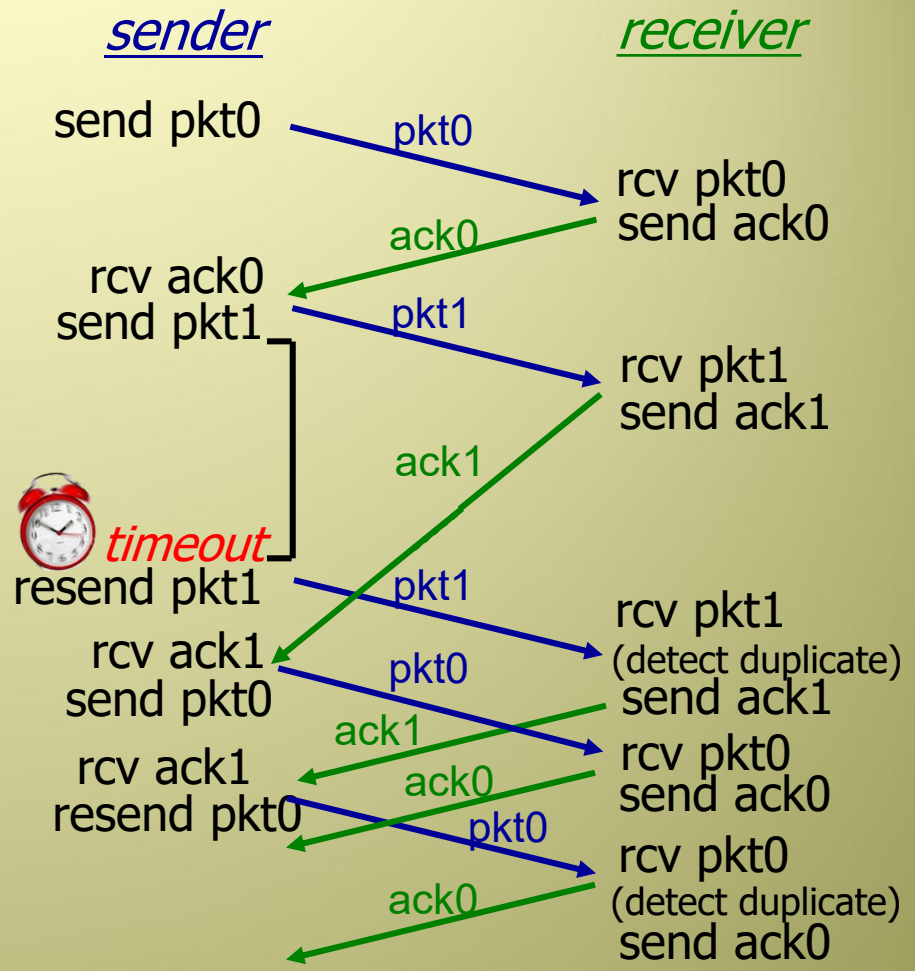


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

- since sequence # alternates between 1 & 0, such protocols are known as **Alternating-bit** protocols
- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

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

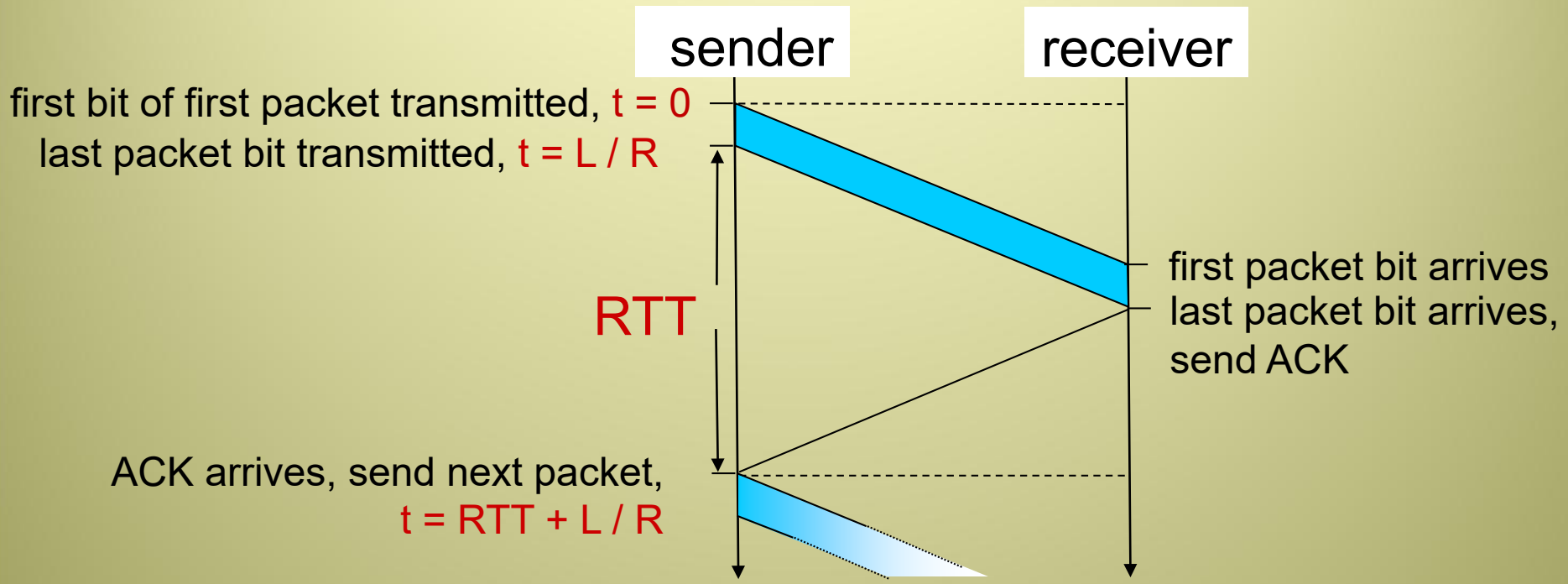
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

  - if RTT=30 msec, 8000 bit pkt every 0.030008 sec,

  ➔ throughput over 1 Gbps link is only: 266.6 kbps

- network protocol limits use of physical resources!

rdt3.0: **stop-and-wait** operation



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

Stop-and-Wait Protocols

Note: ***This is stop-and-wait in general; it is not related to rtd3.0***

- Sender sends one frame, then waits for an acknowledgment (ACK) from the receiver

- If the frame is damaged, this will also be indicated in the ACK

```
void send_data;
{
   damaged=0;
   while there are packets to send
   {
      if (!damaged)
            /* !0 is the same as true in C */
      {
         Get packet from the user;
         Put packet into a frame;
      }
      send(frame);
      Wait for acknowledgment to arrive;
      receive(ack);
      if ack.error
         damaged=1;
      else
         damaged=0;
   }
}
```

Sender code

```
void receive_data;
{
    while there are packets to receive
    {
        Wait for frame to arrive;
        receive(frame);
        Examine frame for transmission error;
        if no transmission error
        {
            ack.error=0;
            Extract packet from the frame;
            Give packet to the user;
        }
        else
            ack.error=1;
        send(ack);
    }
}
```

Receiver code

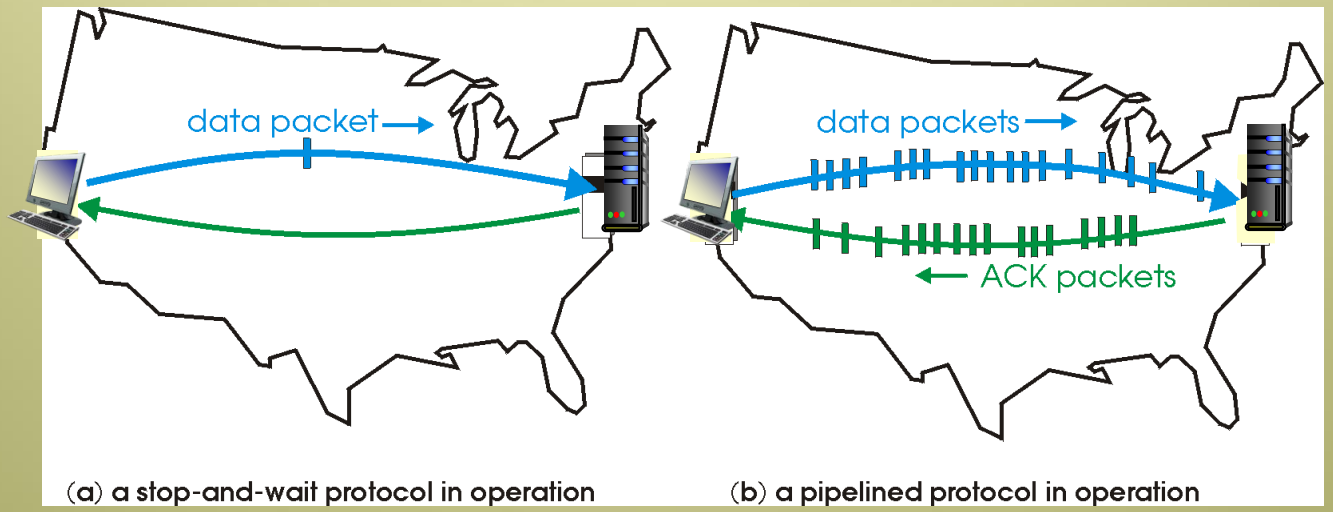
Stop-and-Wait

- ***Analysis of stop-and-wait***
  - **Notice that this is a general analysis of stop-and wait, and has nothing to do with rdt3.0 in particular.**
    - What if frame is lost?
    - What if ACK is lost?
    - What if ACK is received but it is damaged?
    - What about efficiency;
      - Is the protocol too slow and is there a lot of waiting at both sides?
      - What is good about that protocol?

Pipelined protocols

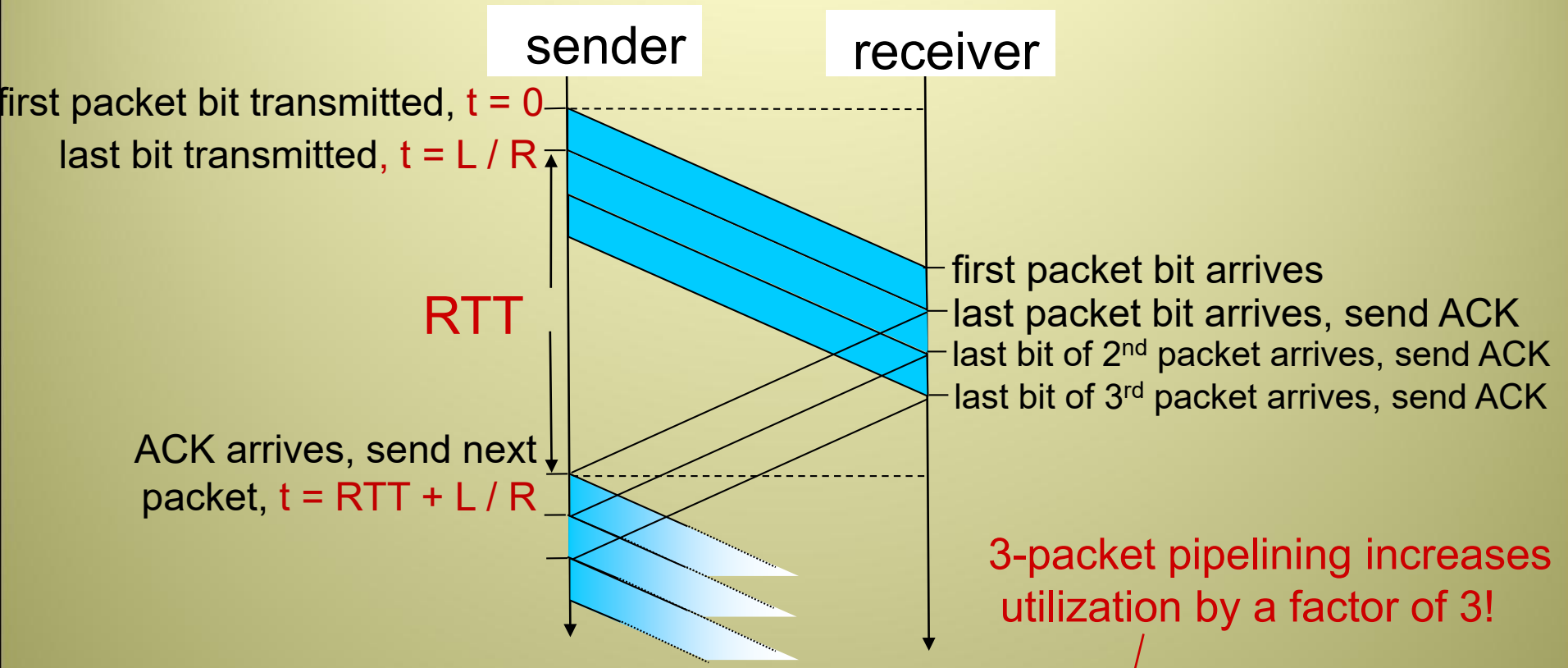
pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

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



- two generic forms of pipelined protocols:
  - ***Go-Back-N,***
  - ***Selective Repeat***

Pipelining: increased utilization



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

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

Another look at Protocol Efficiency

Protocol Efficiency

- There are different ways to measure efficiency
- One measure to efficiency is the ***Effective Data Rate***
- Effective data rate is number of transferred bits per unit of time
- It can be calculated as:

  Number of sent data bits / Elapsed time between sending of two consecutive frames

Another look at Protocol Efficiency

Protocol Efficiency

Assume:

R: bit rate (bps)

S: signal speed (meters/μsec)

D: distance between sender and receiver (meters)

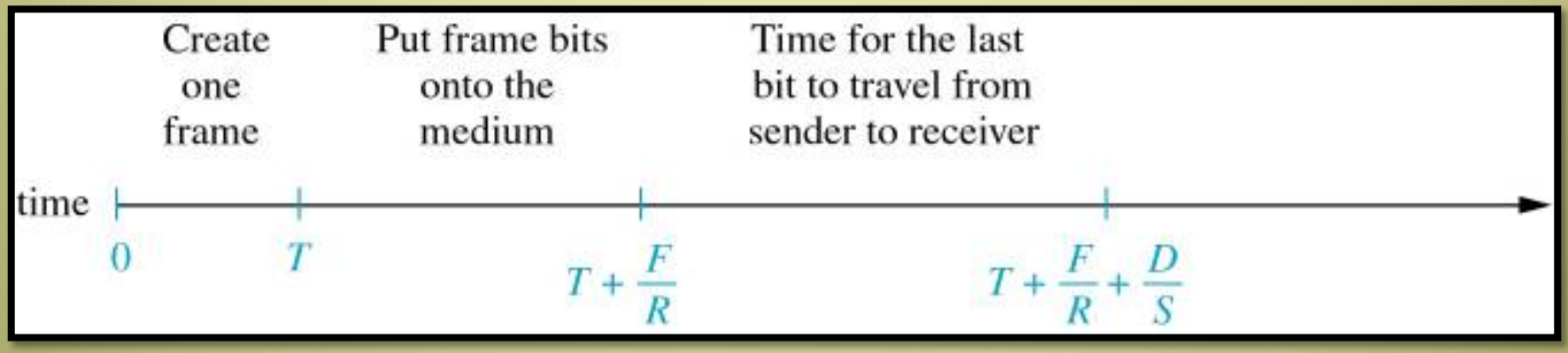
T: time to create a frame

F: frame bits (# of bits per frame including header)

N: data bits (# of data bits per frame)

A: number of bits in an ACK frame

Another look at Protocol Efficiency



Time Required to Send a Frame to Receiver

Another look at Protocol Efficiency

Protocol Efficiency

- For the **unrestricted** protocol, elapsed time between sending consecutive frames is:

  time = T + F/R

➔ EDR = N/(T + F/R)

- For the **stop-and-wait** protocol, it is:

  time to send a frame + time to get ACK back

  time = (T + F/R + D/S) + (T + A/R + D/S)

  Note: the time needed to create an ACK frame may actually differ from the time needed to create a full frame. We assume that the two are the same just for simplicity

➔ EDR = N/[2(T + D/S) + (F + A)/R]

Another look at Protocol Efficiency

Protocol Efficiency

- ***Example:*** Assume:
  - R = 10 Mbps (10 bits /μsec)
  - S = 200 meters/μsec
  - D = 200 meters
  - T = 1 μsec
  - F = 200 bits
  - N = 160 bits
  - A = 40 bits

- for the unrestricted protocol:

➔ EDR = N/(T + F/R) ≈ 7.6 Mbps

- for the stop-and-wait protocol,

  ➔ EDR = N/[2(T + D/S) + (F + A)/R] ≈ 5.7 Mbps

Protocol Efficiency

- the unrestricted protocol and the stop-and-wait protocol are two extremes
- in practice, we may need some other protocol that falls in between these extremes
- Sliding Window protocols provide such alternative
- two major implementation of sliding window protocols are:
  - **Go-back-*n***
  - **Selective Repeat**

Pipelined protocols: overview

- *Also referred to as **Sliding Windows** protocols*

Go-back-N:

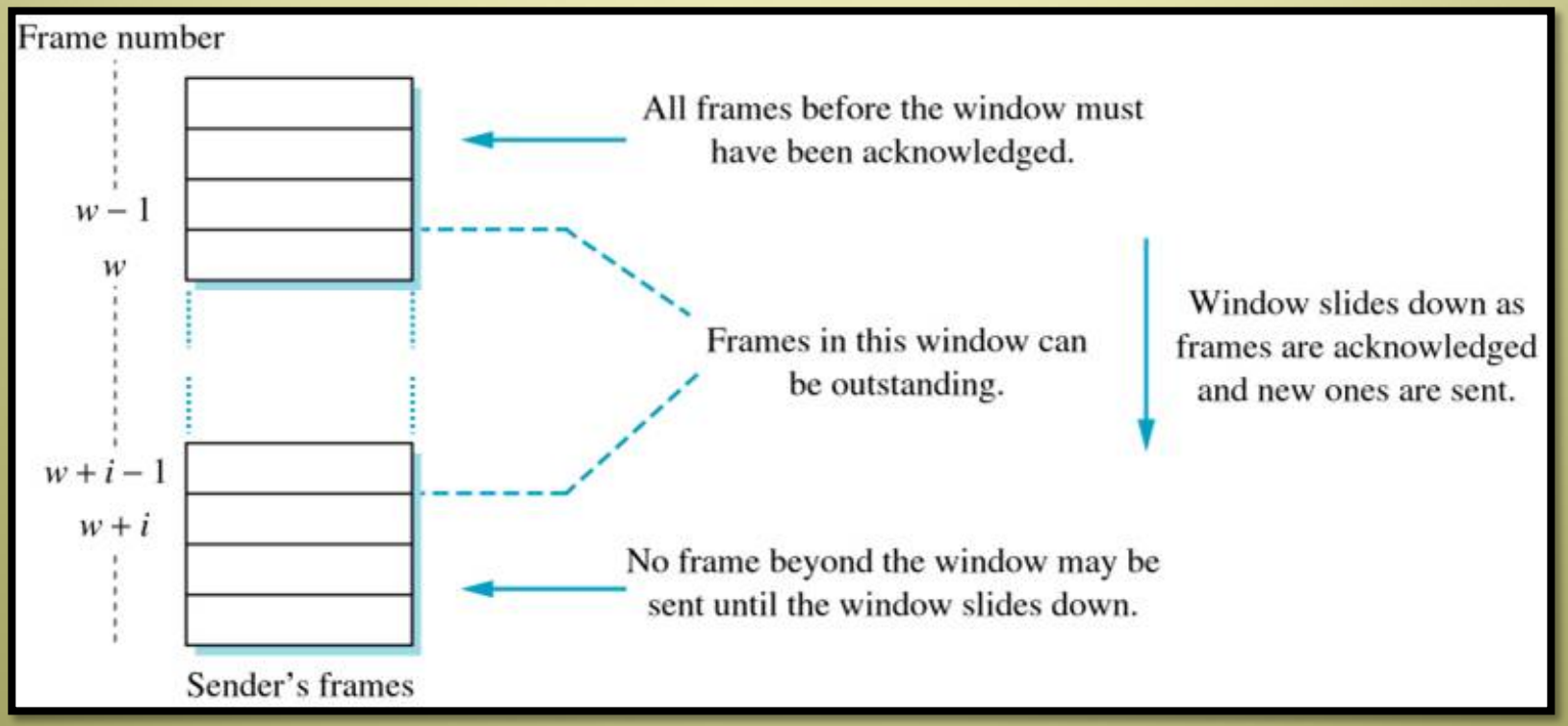
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

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

Sliding Window

- sliding window protocol numbers the frames to be sent and defines a window of these frames



A Sliding Window Protocol

Go-Back-N Protocol

- Go-back-n requires frames to be received in the same order they are sent
- the receiver hence rejects all out-of-order frames; in other words, it rejects all frames but the next expected one
- in reality, communication can be full-duplex, so data is sent in both directions
- instead of sending separate ACK frames, ACKs can be sent along with data

- this is called **piggybacking**

Typical Frame Format

Go-Back-N Protocol

- details of Go-back-n:
  - if the *Number* field in the frame has k bits, then frames can be numbered from 0 → 2^k - 1
  - if there is more than 2^k frames, then frame numbers are duplicated. For example, if k=3, then frames will be 0, 1, …, 6, 7, 0, 1, …
  - receiver expects frames in order. Any out-of-order frames are rejected and a NAK is sent for the expected frame
  - if the expected frame is received but damaged, then it is discarded and a NAK is sent
  - not every frame is explicitly ACKed

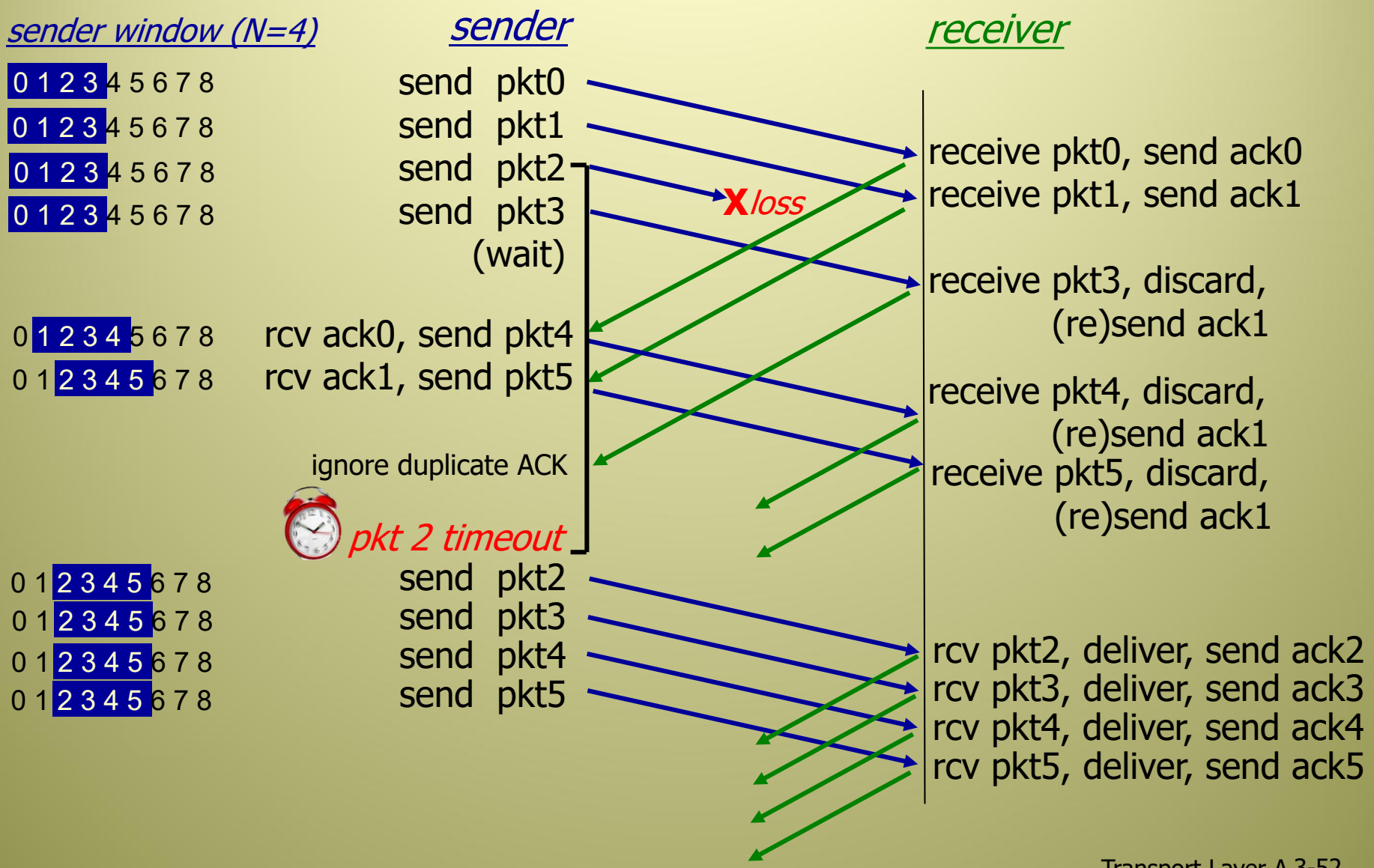
    For example, if sender sends frames 35, 36, 37, 38 & 39, ACKs for 36 & 39 may be sent. ACK for 36 implicitly indicates that 35 & 36 were accepted. ACK for 39 implicitly indicate that 37, 38 & 39 were accepted

Go-Back-N Protocol

- details of Go-back-n (continue…):
  - piggybacking is used whenever possible, however frames must be ACKed within a specific time period
  - an ***ACK timer*** is defined at the receiver; if this timer reaches its limit, the frame must be ACKed even without piggybacking
  - if the sender does not receive an ACK within a time period (a ***frame timer*** is used here), it assumes that something has gone wrong and so it resends every packet in its window

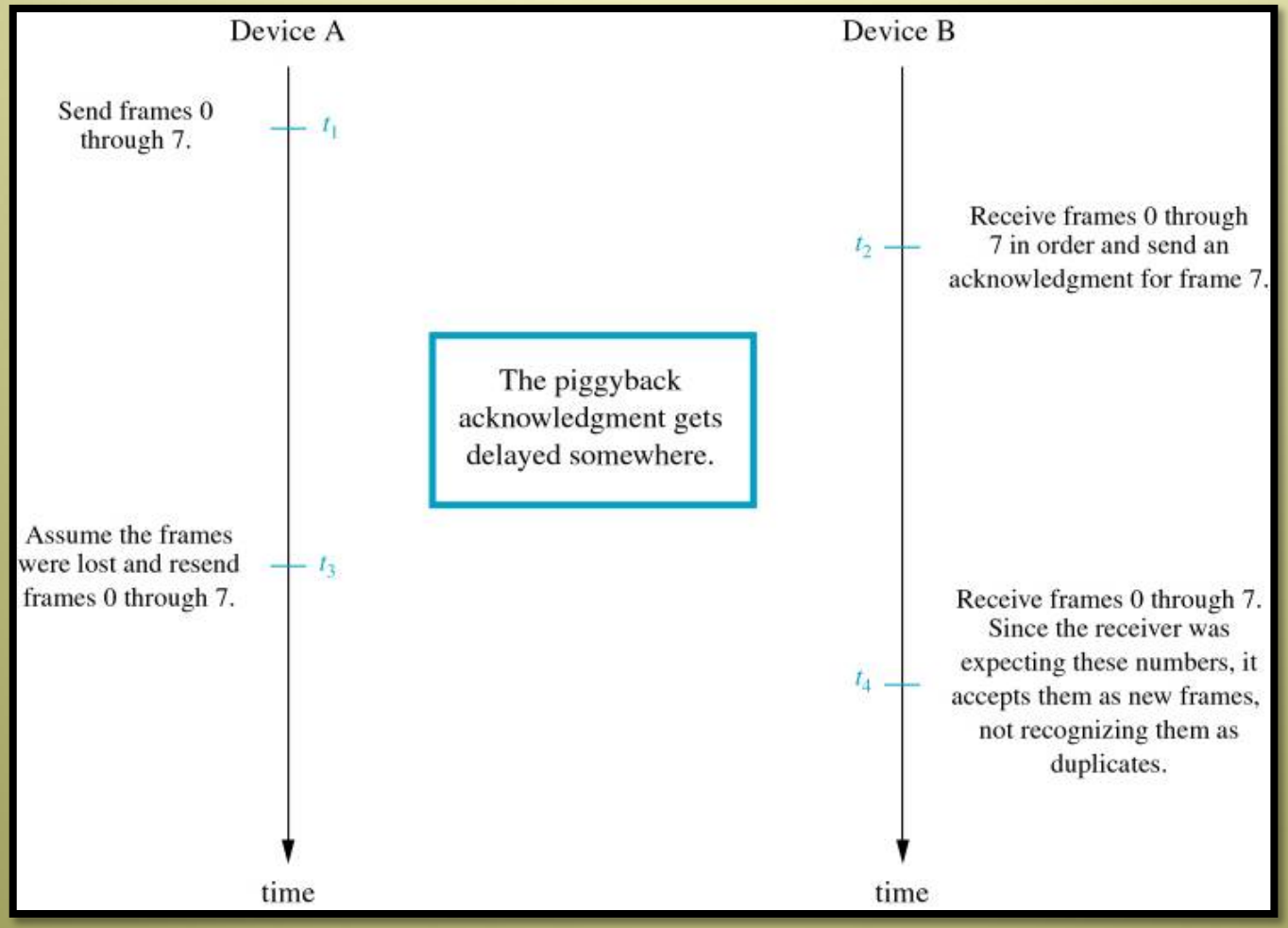
    in other words, the sender goes back to the beginning of the window (n) and resends all frames in the window; that is the reason behind the name: *go-back-n*

GBN in action



Go-Back-N Protocol

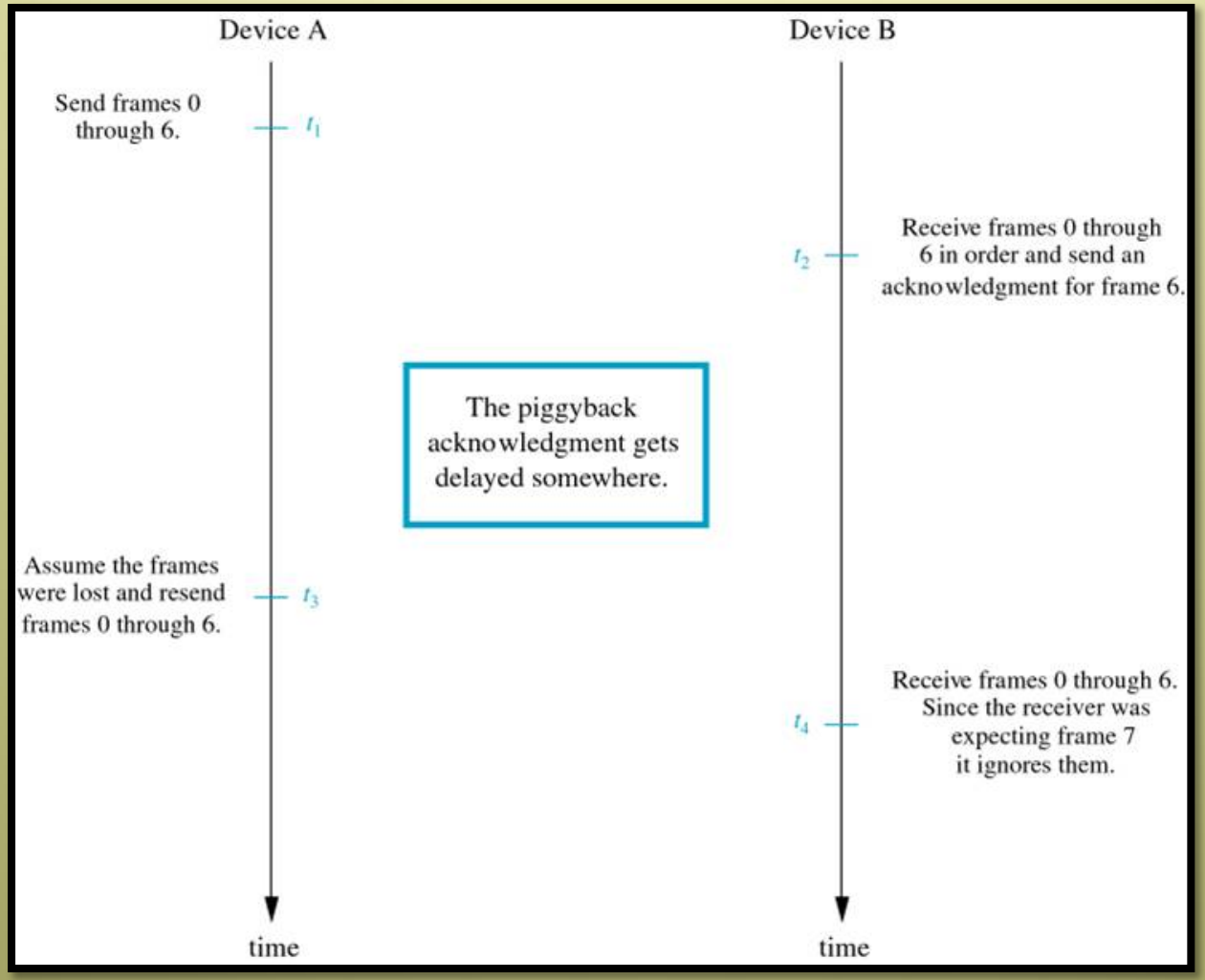
- what is the maximum window size?



Protocol Failure when Window Size = 2^k

Go-Back-N Protocol

- what is the maximum window size?



Protocol Success when Window Size = $2^k - 1$

Selective repeat

- Go-back-n works well only under certain conditions (what are these?)
- once these conditions change, the protocol becomes less efficient
- selective repeat provides an alternative
- with selective repeat, the receiver is required to accept (buffers) frames even they are out of order
- all frames must however be reordered before being delivered

Selective repeat

- Details of Selective Repeat:

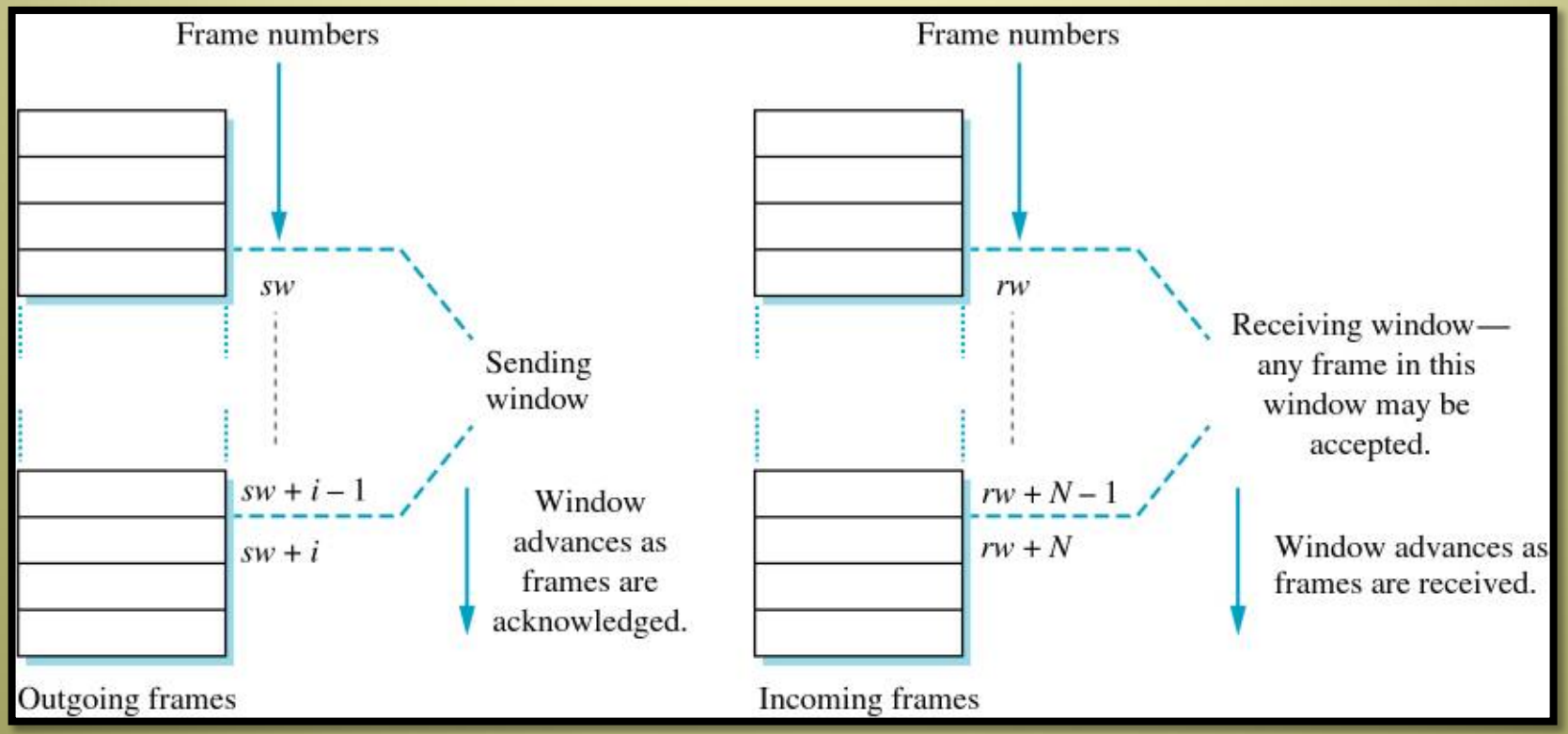
  Selective repeat is similar to go-back-n in the following:

  - Uses the same frame format
  - All frames are numbered using k-bit field
  - Sender has a window that defines the maximum number of outstanding frames
  - ACKs are piggybacked whenever possible
  - Not all frames are explicitly ACKed
  - Uses NACKs (or duplicate ACKs if protocol is NAK-free) for damaged and out-of-order frames
  - There is an ACK timer that is used to force the sending of an ACK frame if piggybacked is not possible

The similarities end here!

Selective repeat

- Details of Selective Repeat (continue…):



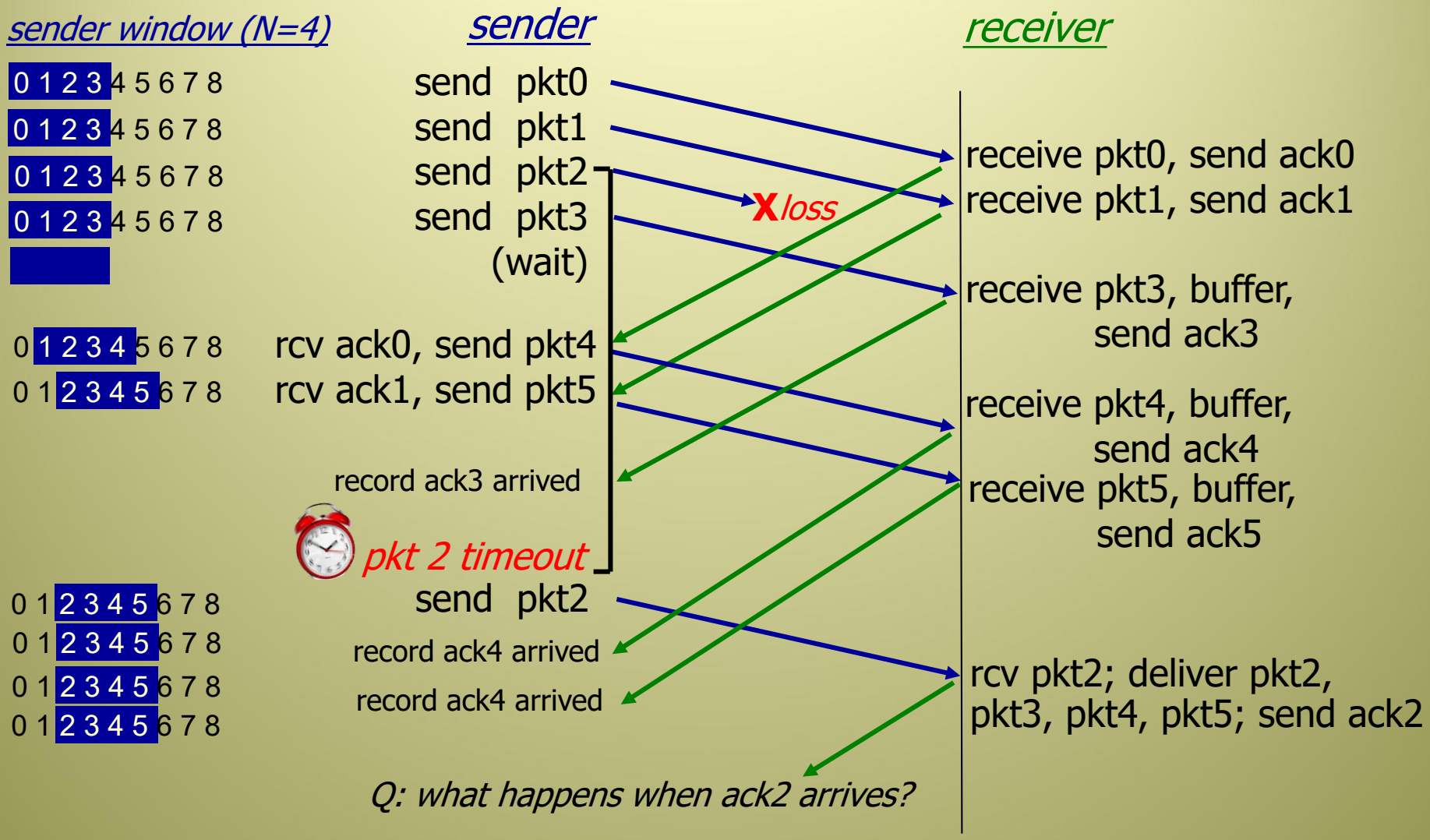
Sending & Receiving Windows for Selective Repeat Protocol

Selective repeat

- details of Selective Repeat (continue…):
  - out-of-order frames are buffered until all the ones that precede them arrive
  - once a receiver detects out-of-order arrival, it sends NAK for the expected frame
  - if the sender receive NAK, it resends only that frame
  - once all consecutive frames arrive, they are delivered and the receiving window is advanced
  - if a frame timer expires, only that frame is resent
  - piggybacking may not ACK the last received frame; rather it ACKs the highest one delivered to the user

Selective repeat in action

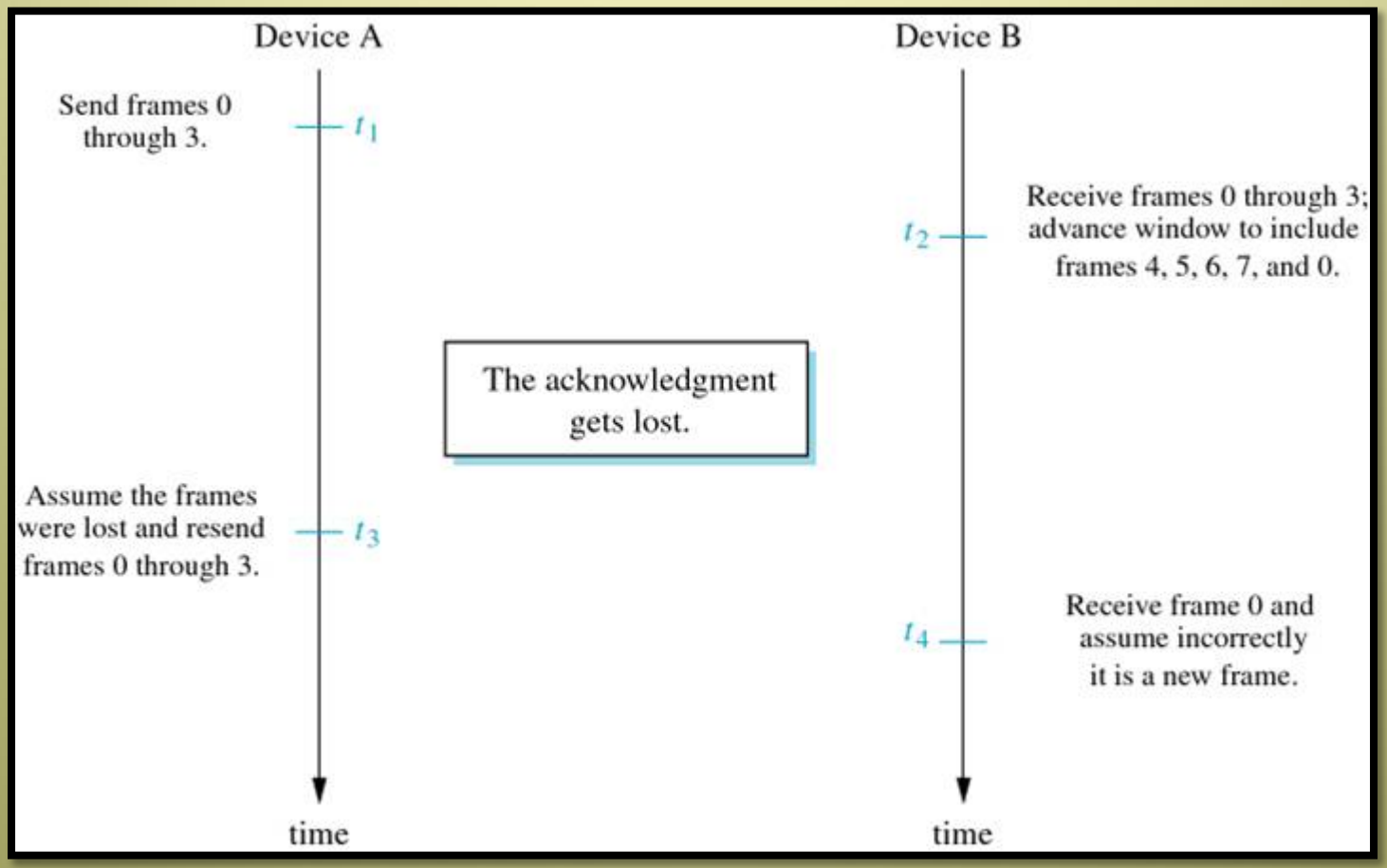
*Notice that this is a different variation of Selective Repeat , where each packet is **explicitly** ACKed?*



Selective repeat

- what is the maximum window size?

➔ constraint: window size must be <= ½ of 2^k; which is <= 2^{k-1}

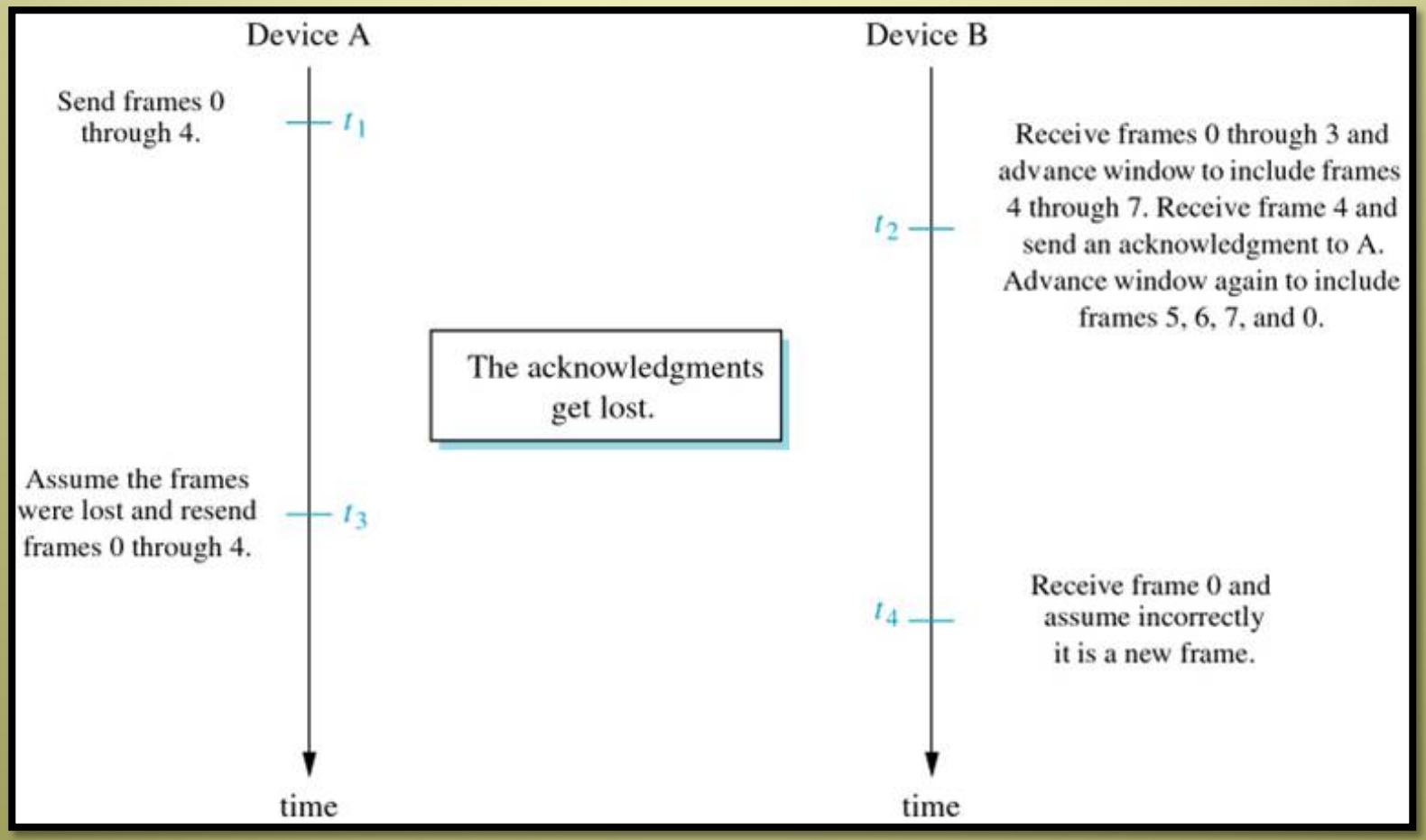


Protocol Failure: **Receiving** Window size is > 2^{K-1}

Selective repeat

- what is the maximum window size?

➔ constraint: window size must be <= ½ of 2^k; which is <= 2^{k-1}



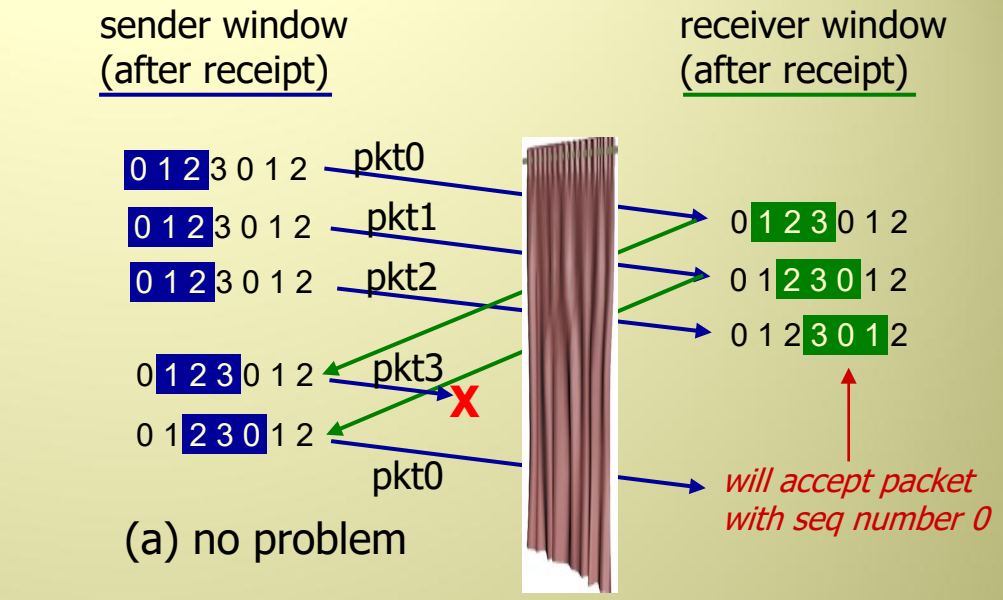
Protocol Failure: **Sending** Window size is > 2^{K-1}

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)

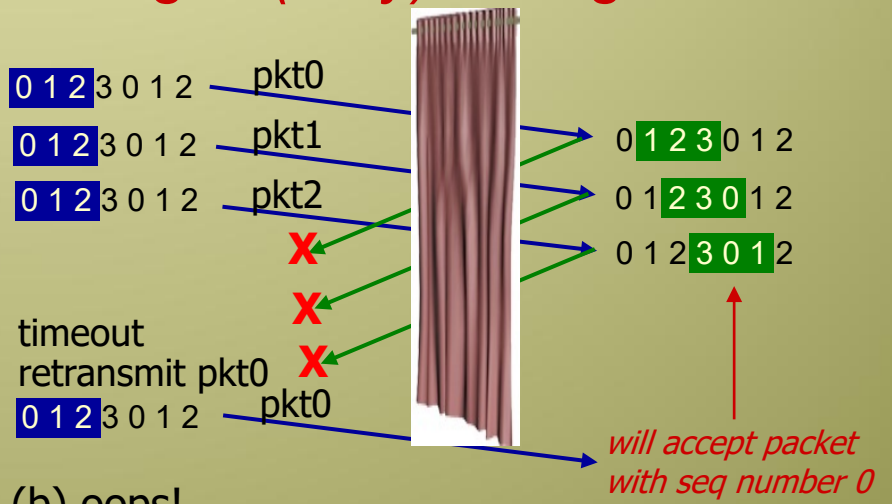
Q: what relationship between seq # size and window size to avoid problem in (b)?



(a) no problem

receiver can't see sender side.
receiver behavior identical in both cases!

something's (very) wrong!



(b) oops!

Efficiency of Sliding Window Protocols

- unrestricted and stop-and-wait efficiency were affected by many factors such as distance, raw bit rate, …etc.
- here also, there are many factors that can affect the efficiency of a sliding window protocol; for example, the timer value to trigger resent, the number of frames carrying piggybacking ACKs, …etc.
- for simplicity, assume that there are no errors, and that there is enough data sent back so piggybacking is always used

Efficiency of Sliding Window Protocols

- Assume:

  R: bit rate (bps)

  S: signal speed (meters/μsec)

  D: distance between sender and receiver (meters)

  T: time to create a frame

  F: frame bits (# of bits per frame including header)

  N: data bits (# of data bits per frame)

  A: number of bits in an ACK frame

  W: window size

Efficiency of Sliding Window Protocols

- now, there are two possibilities:

Case 1) sender window never reaches its maximum size

- this is the case when ACKs come back fast enough so the sender just advances its window and sends again
- this case then resembles the unrestricted protocol

Case 2) the sender sends all frames in its window before getting any ACK back

- this is the case when the sender *W* frames at a time and then waits for an ACK
- this then resembles stop-and-wait, but with window size *W* instead of 1

- the two cases can then be compared by looking at the time needed to send W frames

Efficiency of Sliding Window Protocols

- to construct and send 1 frame, we need T + F/R
- ➔ to construct and send W frames, we need W(T + F/R)

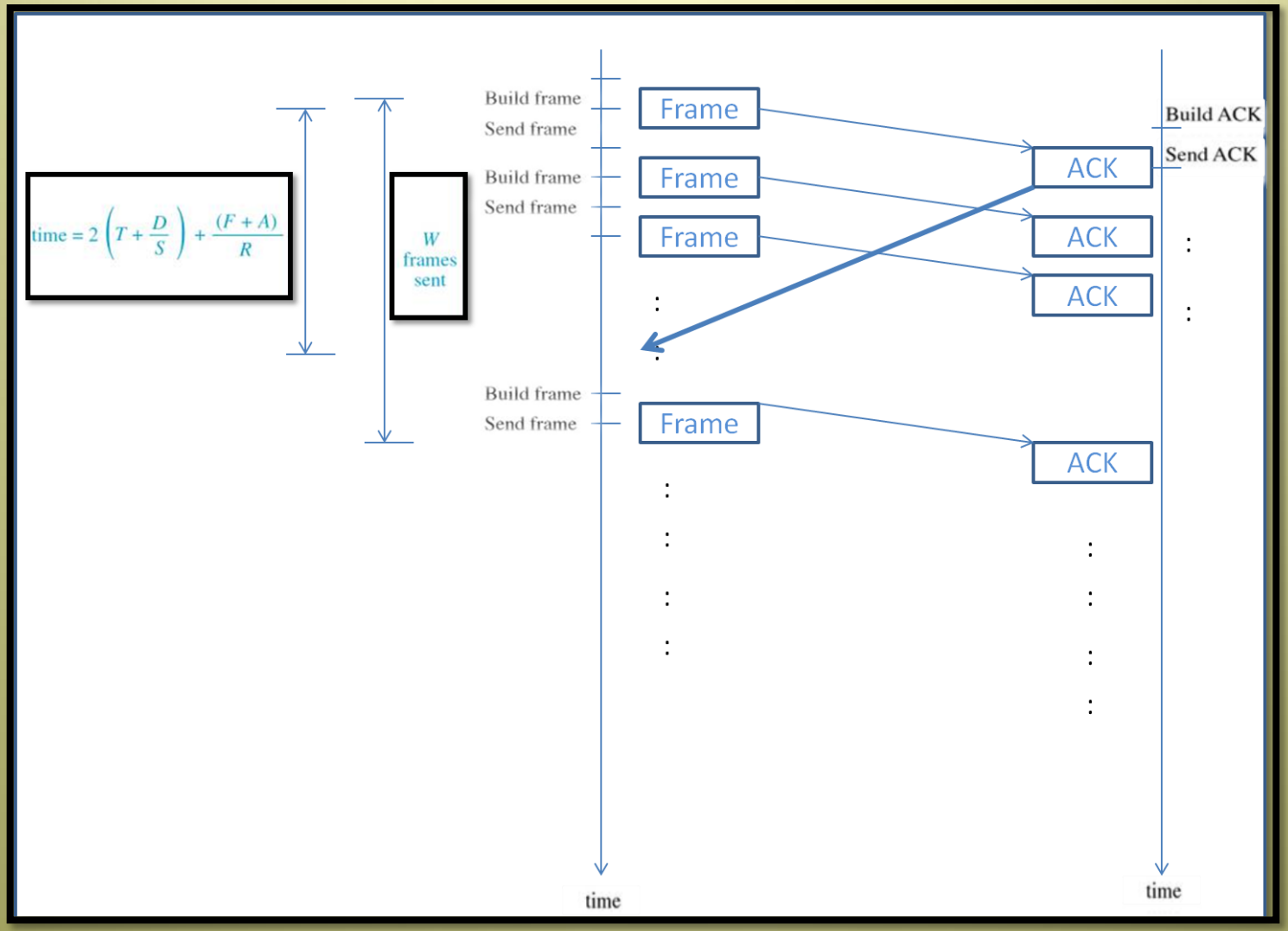
- to construct and send ACK back, we need T + A/R
- since ACKs are piggybacked, we need T + F/R

- we also need D/S for traveling time between sender and receiver
- ➔ total time to send 1 frame and get ACK back is:
  2 (T + F/R +D/S)

- for Case 1 (resembling unrestricted protocol):
  W(T + F/R) > 2 (T + F/R +D/S)

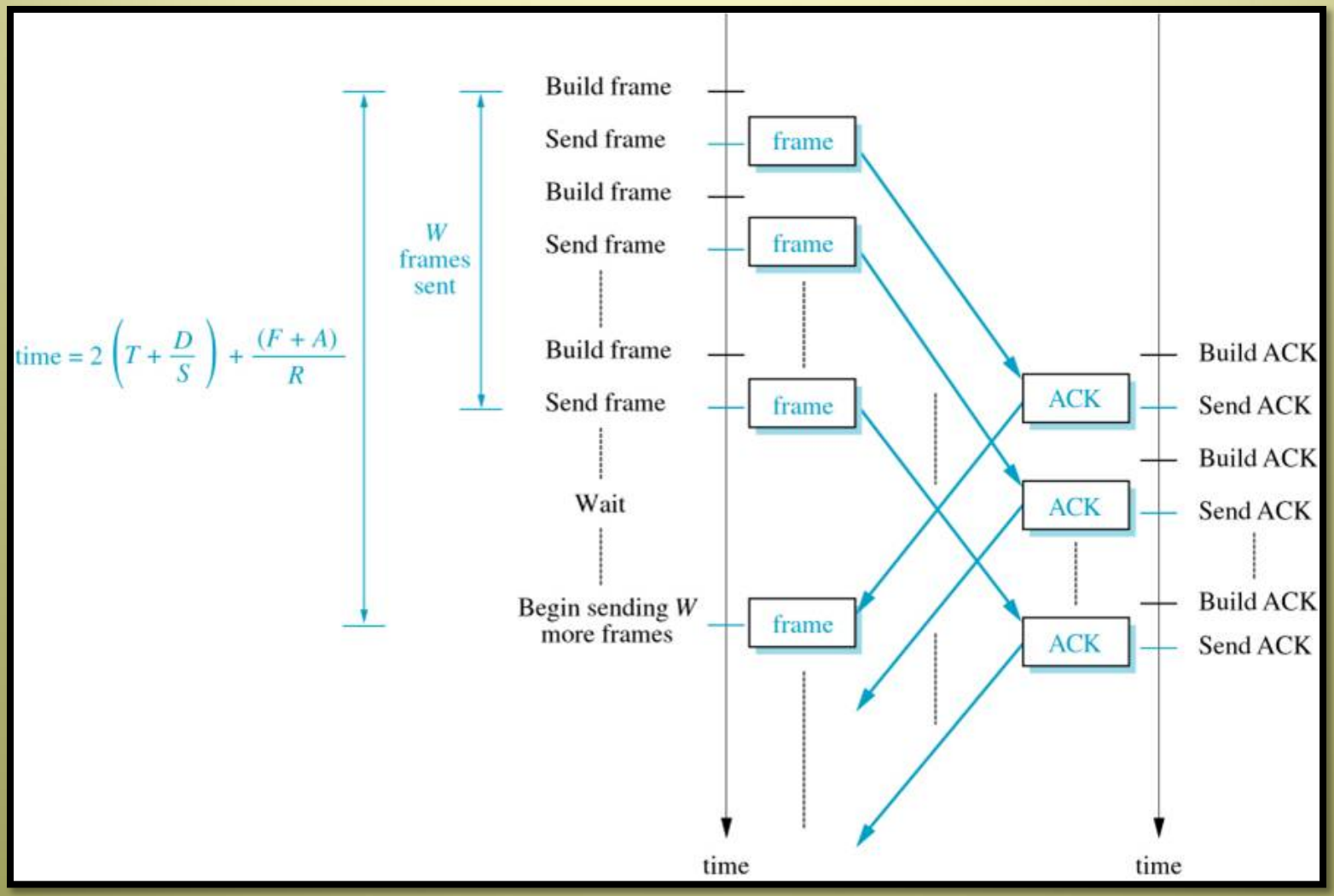
- for Case 2 (resembling W-Oriented stop-and-wait):
  W(T + F/R) < 2 (T + F/R +D/S)

Efficiency of Sliding Window Protocols



ACKs Received before Sending All Windowed Frames – **Case 1**

Efficiency of Sliding Window Protocols



Sending All Windowed Frames and Waiting – **Case 2**

Efficiency of Sliding Window Protocols

- what is the effective data rate then?
  - Case 1 is actually the unrestricted protocol, hence
    ➔ EDR = N/(T + F/R)
  - Case 2 is actually the W-Oriented stop-and-wait, hence
    ➔ EDR = (W * N)/[2(T + D/S) + (F + A)/R]

    here W * N replaced N in the original calculation. Why?

Efficiency of Sliding Window Protocols

- ***Example:*** Assume:
  - R = 10 Mbps (10 bits /μsec)
  - S = 200 meters/μsec
  - D = 200 meters
  - T = 1 μsec
  - F = 200 bits
  - N = 160 bits
  - A = 40 bits
  - W = 4 frames

➔ applying these values, Case 1 condition is the one satisfied; so it is unrestricted version and the efficiency can be calculated as:

➔ EDR = N/(T + F/R) ≈ 7.6 Mbps

Efficiency of Sliding Window Protocols

- ***Example:*** Assume:
  - R = 10 Mbps (10 bits /µsec)
  - S = 200 meters/µsec
  - D = 5000 meters
  - T = 1 µsec
  - F = 200 bits
  - N = 160 bits
  - A = 40 bits
  - W = 4 frames
- applying these values, Case 2 condition is the one satisfied; so

- EDR = (W * N)/[2(T + D/S) + (F + F)/R]
  = (4 * 160 bits)/[2(1 µsec + 5000 meters/ 200 meters/µsec) + 400 bits/ 10 bits /µsec]
  = 640 bits /[2(26 µsec) + 40 µsec]
  = 640 bits / 96 µsec
  = 6.96 Mbps

- notice that in that example all ACKs are piggybacked so F replaces A in the computation of EDR

Protocol Correctness

- does these protocols work? Will they fail in other cases that we do not know?
- in order to guarantee that a protocol works, formal proof is needed
- however, sometimes it is very difficult to derive formal proofs
- a reasonable alternative in such cases is verification
- two common verification tools are:
  - Finite State Machines (FSM)
  - Petri Net (similar to FSM in that it uses graphs to represent states and transitions, but does it in a different way)

Protocol Correctness

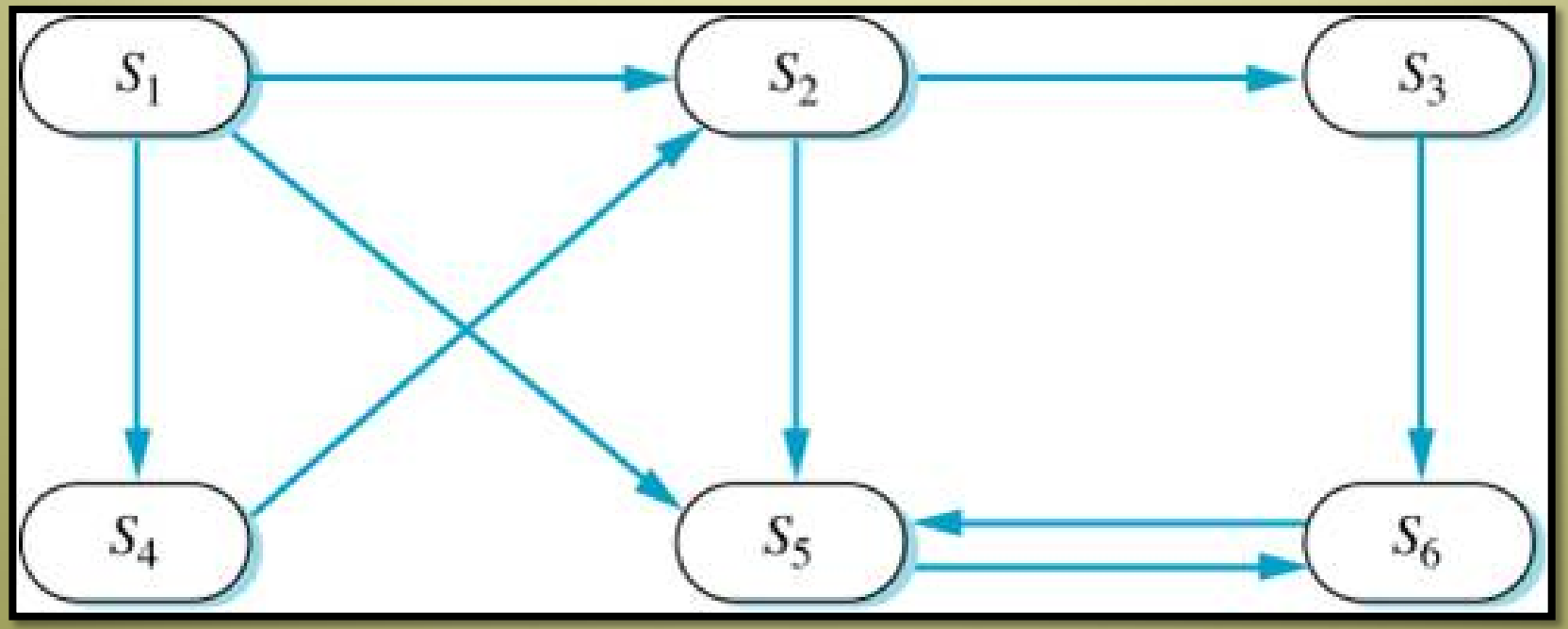
Finite State Machine

- indicates all possible states
- trace the different events that changes one state to another (or possibly back to the same state)
- an event that causes a change of state is called ***state transition***
- a State Transition Diagram (STD) can then be used to represent an algorithm

Protocol Correctness

Finite State Machine

what can you observe by looking at the following STD?



General State Transition Diagram

Protocol Correctness

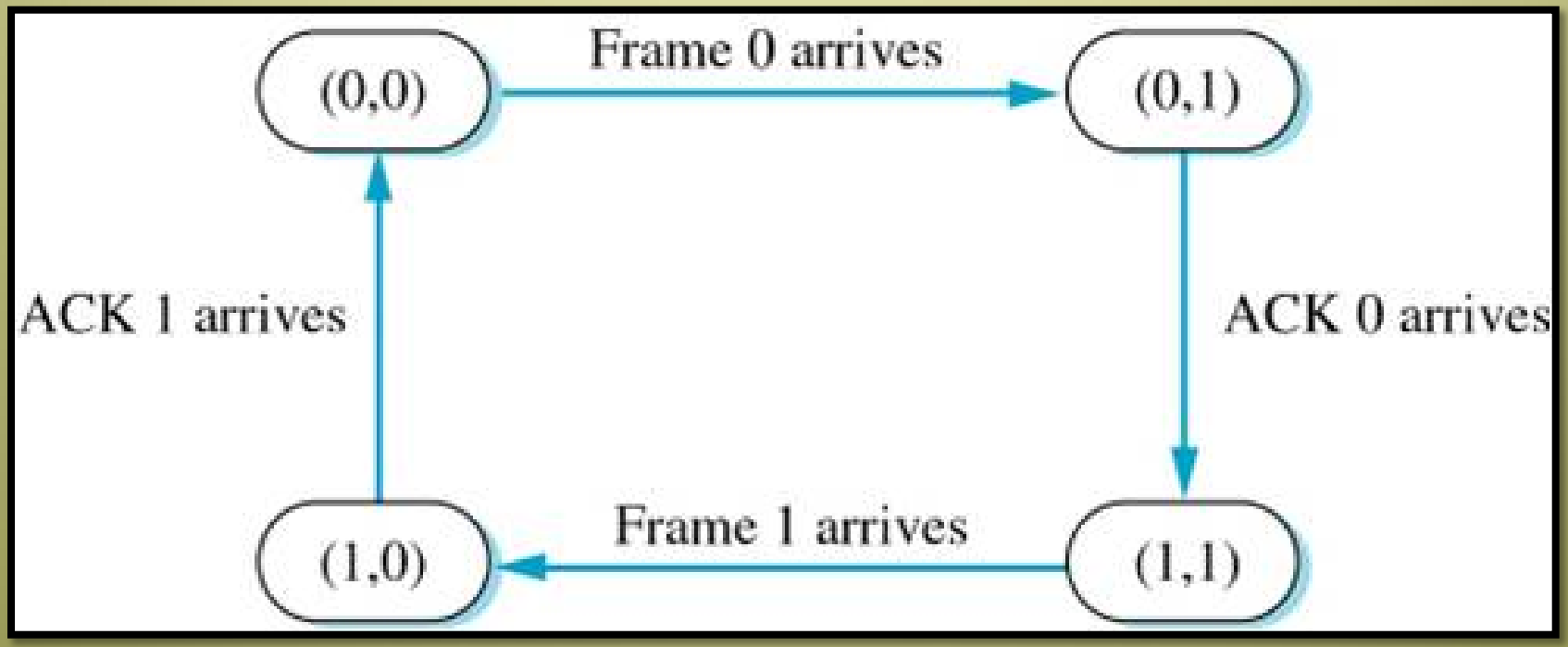
Finite State Machine

STD for a simplified go-back-n protocol

- assume: k=1, window size is 1, no time-out or retransmission and data go from sender to receiver

➔ this is like stop-and-wait protocol with frame numbers

- (x,y) ≡ x is the ACK number and y is the next expected frame number
  for example, (0,1) means there is an ACK for frame 0 and the next expected frame is 1



STD for a stop-and-wait Protocol with Frame Numbers