

# CSE 321

# Computer Networks

July 2018 Term

## Class 17-19: Reliable Data Transfer

Khaled Mahmud Shahriar  
Assistant Professor, Dept. of CSE, BUET  
[khaledshahriar@cse.buet.ac.bd](mailto:khaledshahriar@cse.buet.ac.bd)

# Chapter 3

## Transport Layer

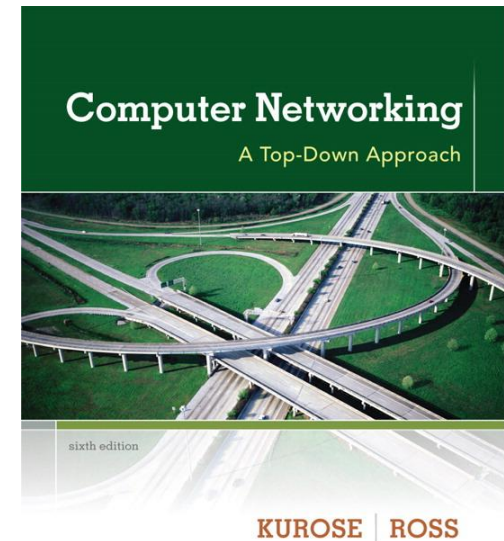
### A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- ❖ If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- ❖ If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

© All material copyright 1996-2012  
J.F Kurose and K.W. Ross, All Rights Reserved



*Computer  
Networking: A Top  
Down Approach*  
6<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Addison-Wesley  
March 2012

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

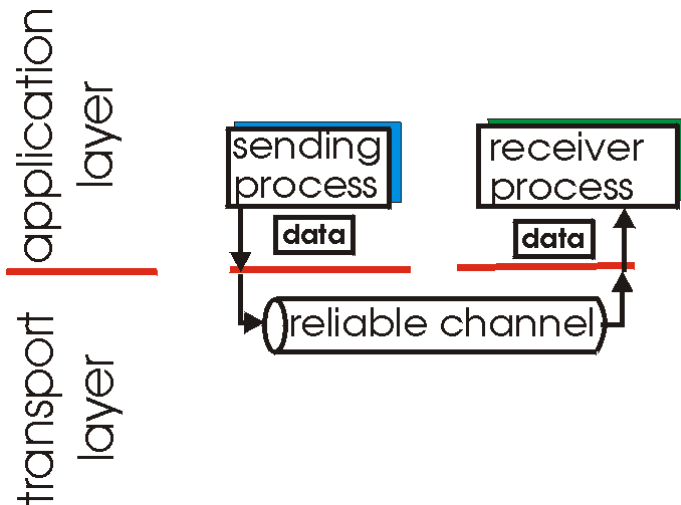
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Principles of reliable data transfer

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

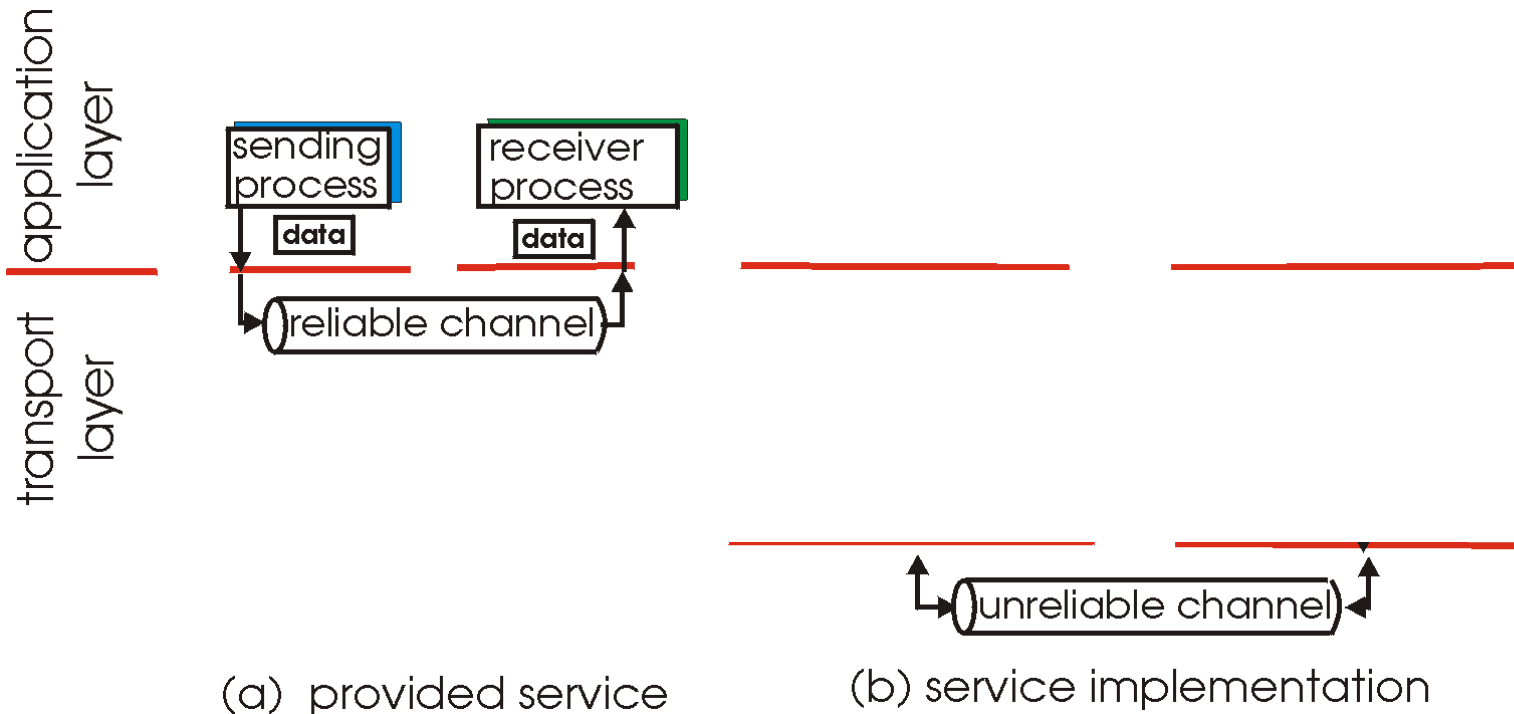


(a) provided service

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

# Principles of reliable data transfer

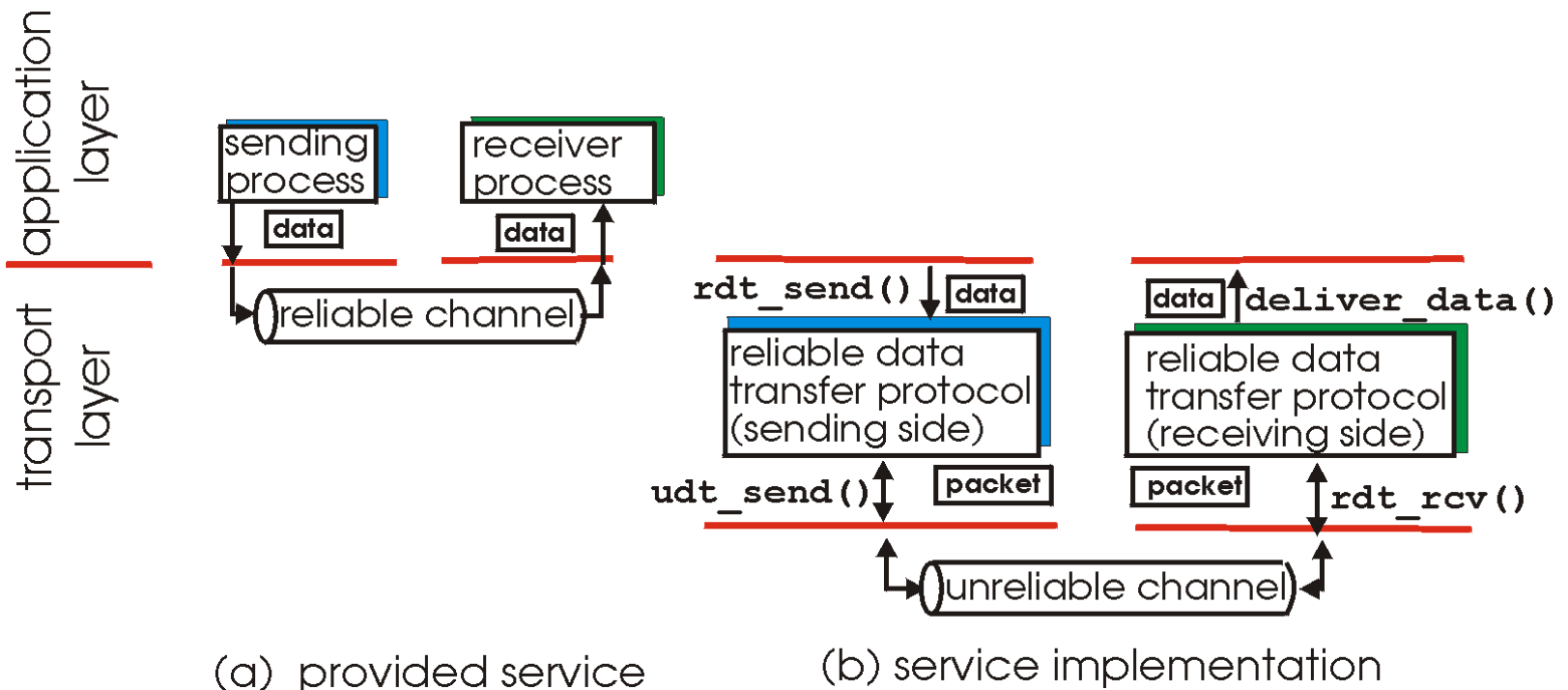
- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



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

# Principles of reliable data transfer

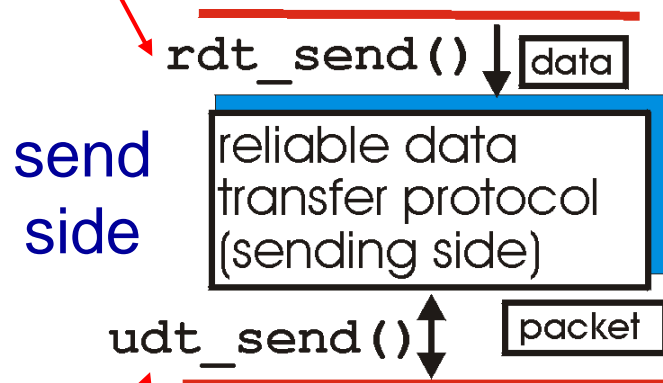
- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



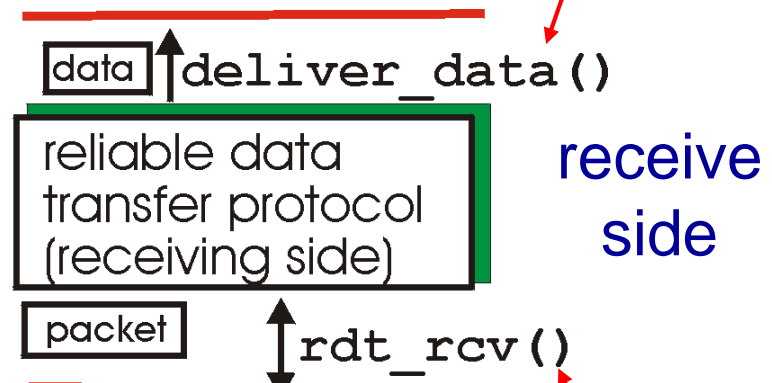
- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Reliable data transfer: getting started

**rdt\_send()** : called from above,  
(e.g., by app.). Passed data 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

# Characteristics of Channel

- ❖ Packet may get corrupted due to noise, device error etc. but always reaches the receiver
- ❖ Packets may get lost in transit
- ❖ Packets may be reordered
  - problem of delayed duplicate

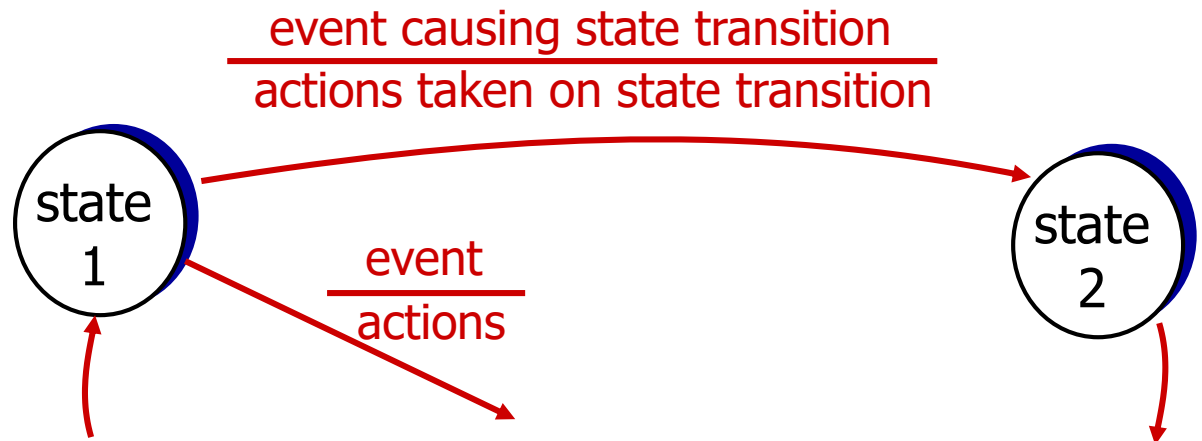


# Reliable data transfer: getting started

we'll:

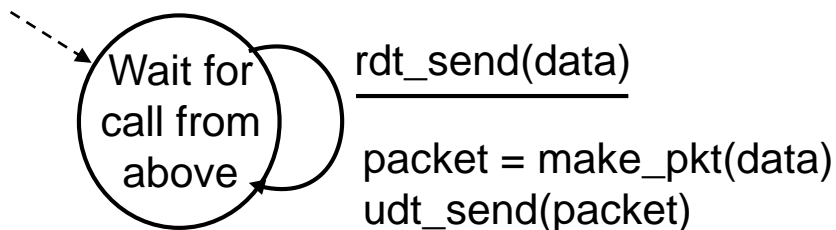
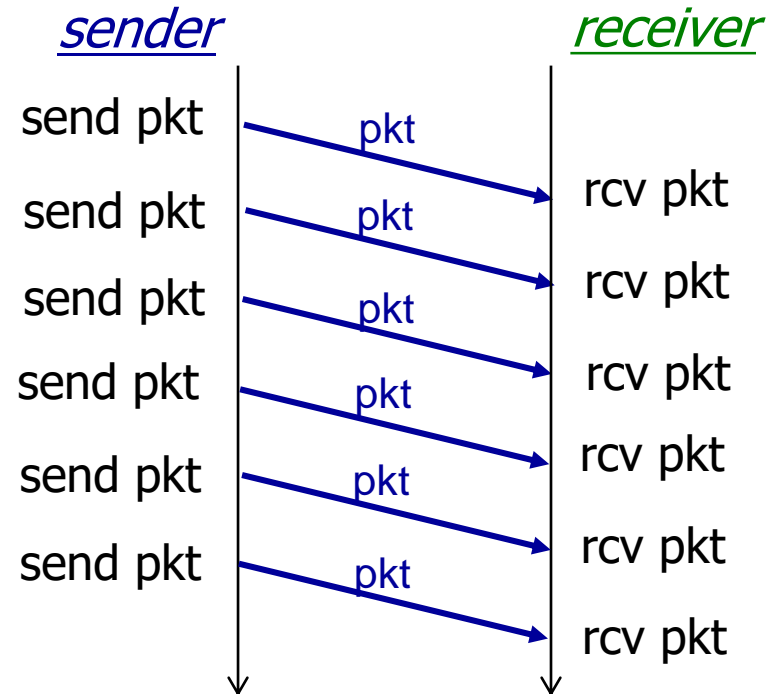
- ❖ incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- ❖ consider only unidirectional data transfer
  - but control info will flow on both directions!
- ❖ use finite state machines (FSM) to specify sender, receiver

**state:** when in this “state” next state uniquely determined by next event

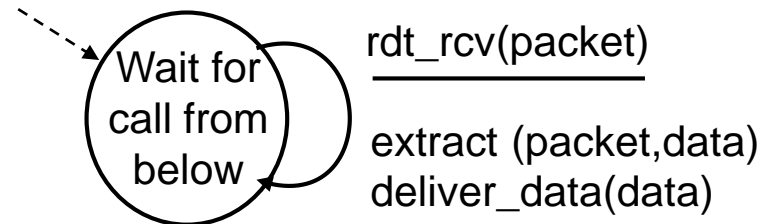


# rdt1.0: reliable transfer over a reliable channel

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



sender



receiver

# rdt2.0: channel with bit errors

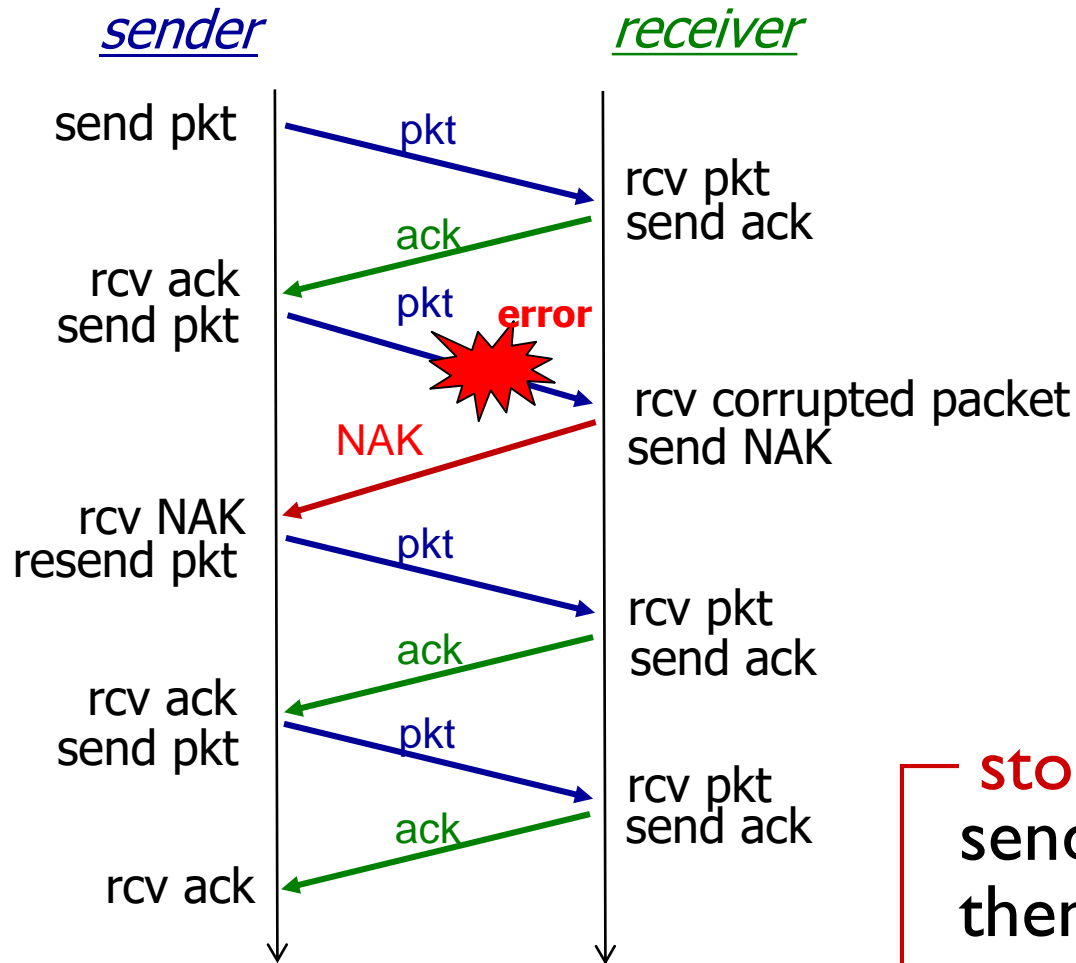
- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*

*How do humans recover from “errors”  
during conversation?*

# rdt2.0: channel with bit errors

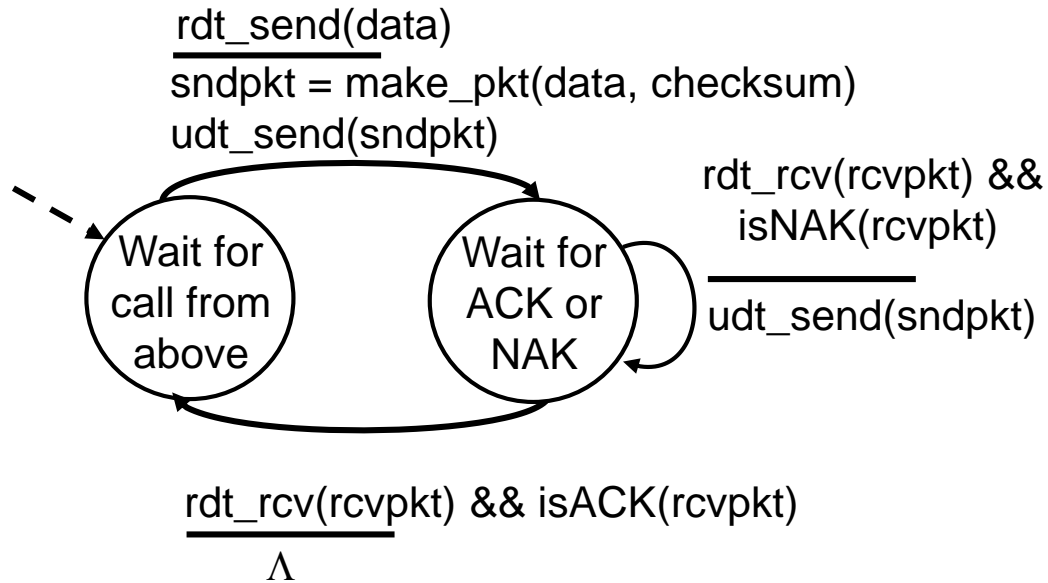
- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - 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

# rdt2.0 in action space-time diagram



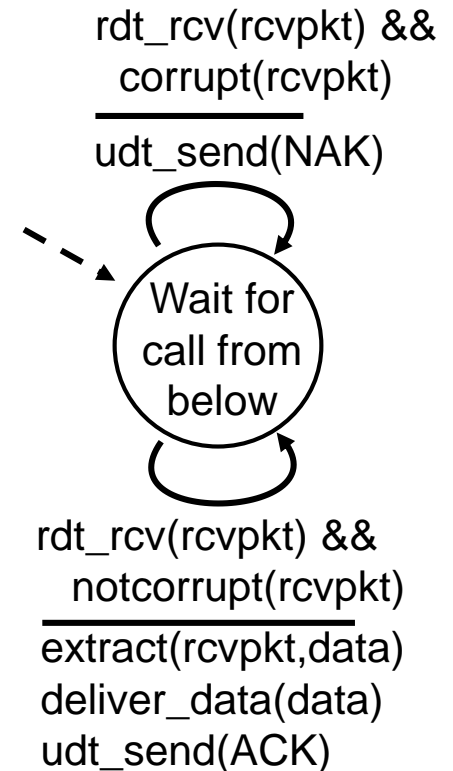
**stop and wait**  
sender sends one packet,  
then waits for receiver  
response

# rdt2.0: FSM specification

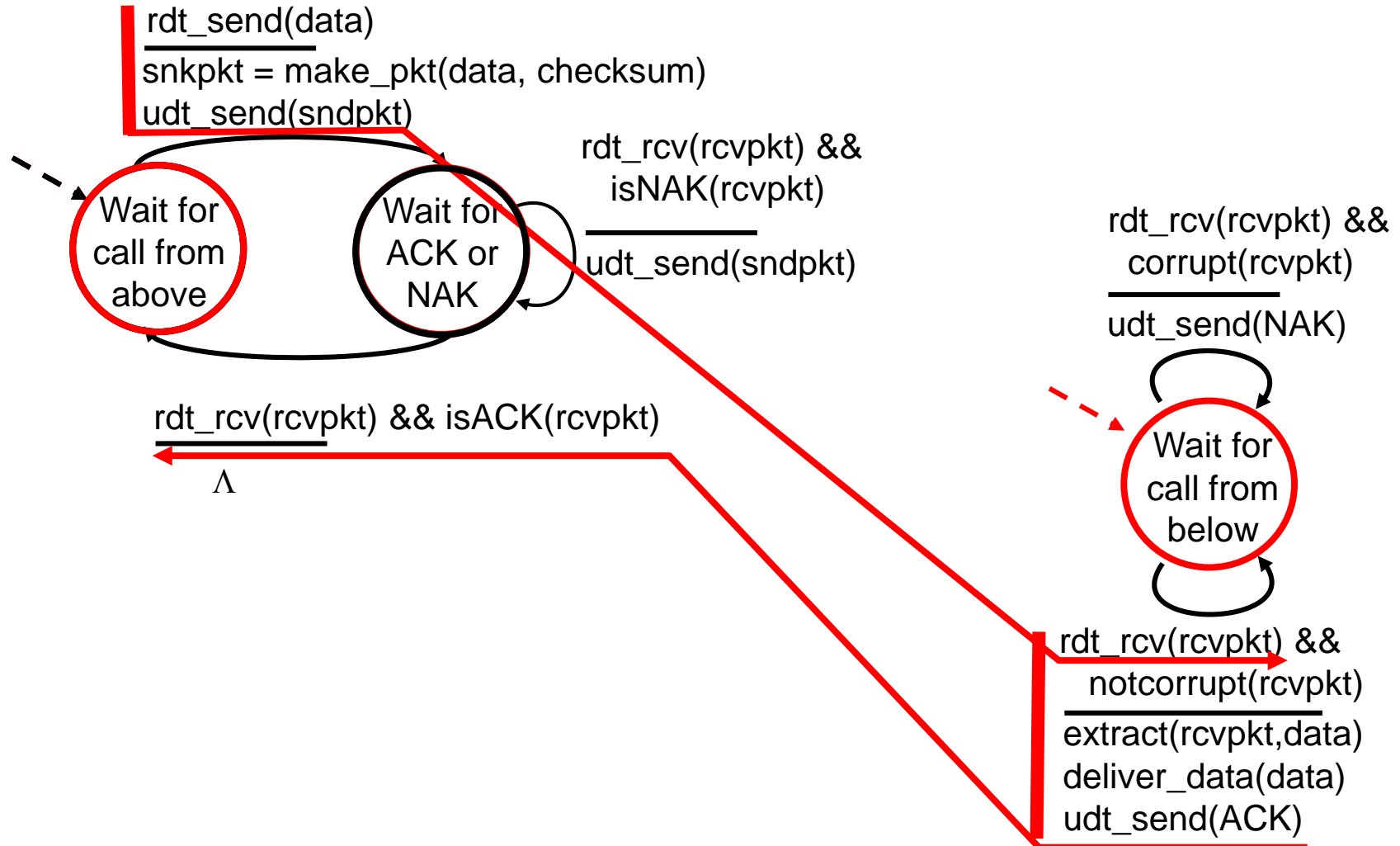


sender

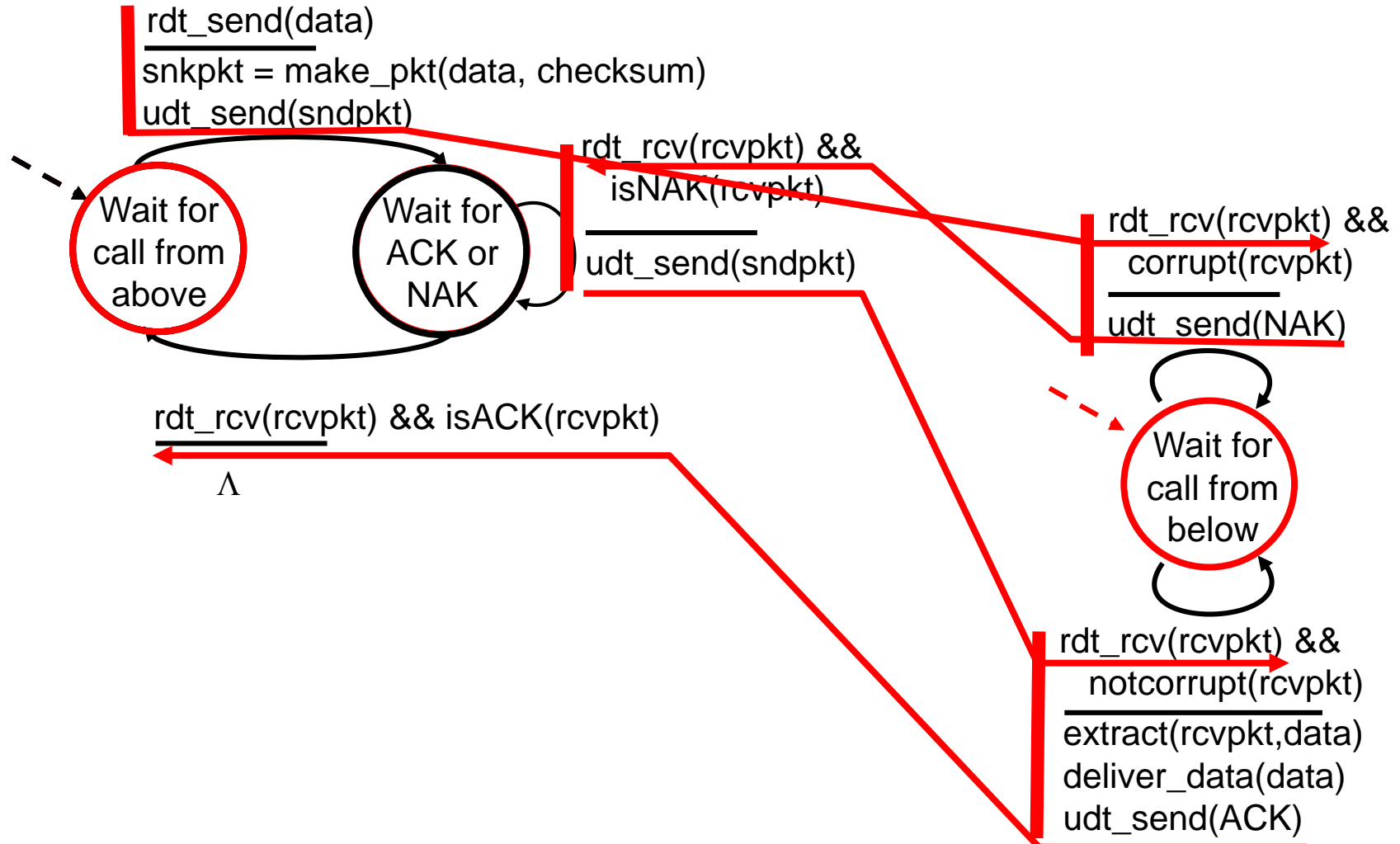
receiver



# rdt2.0: operation with no errors



# rdt2.0: error scenario

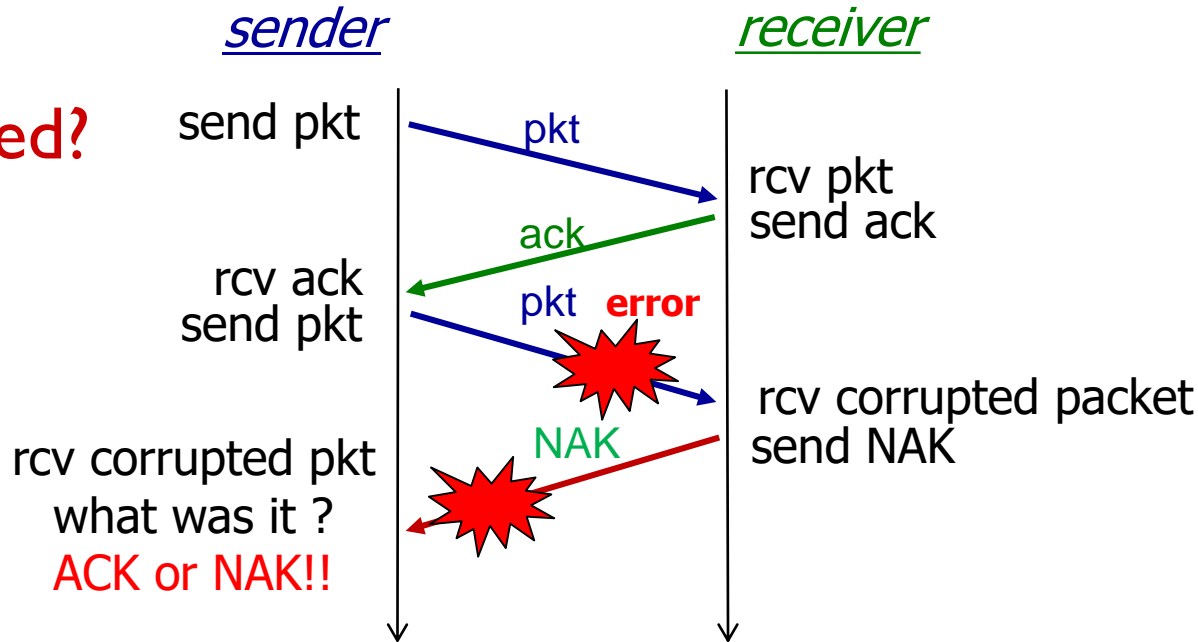




# rdt2.0 has a fatal flaw!

what happens if  
**ACK/NAK corrupted?**

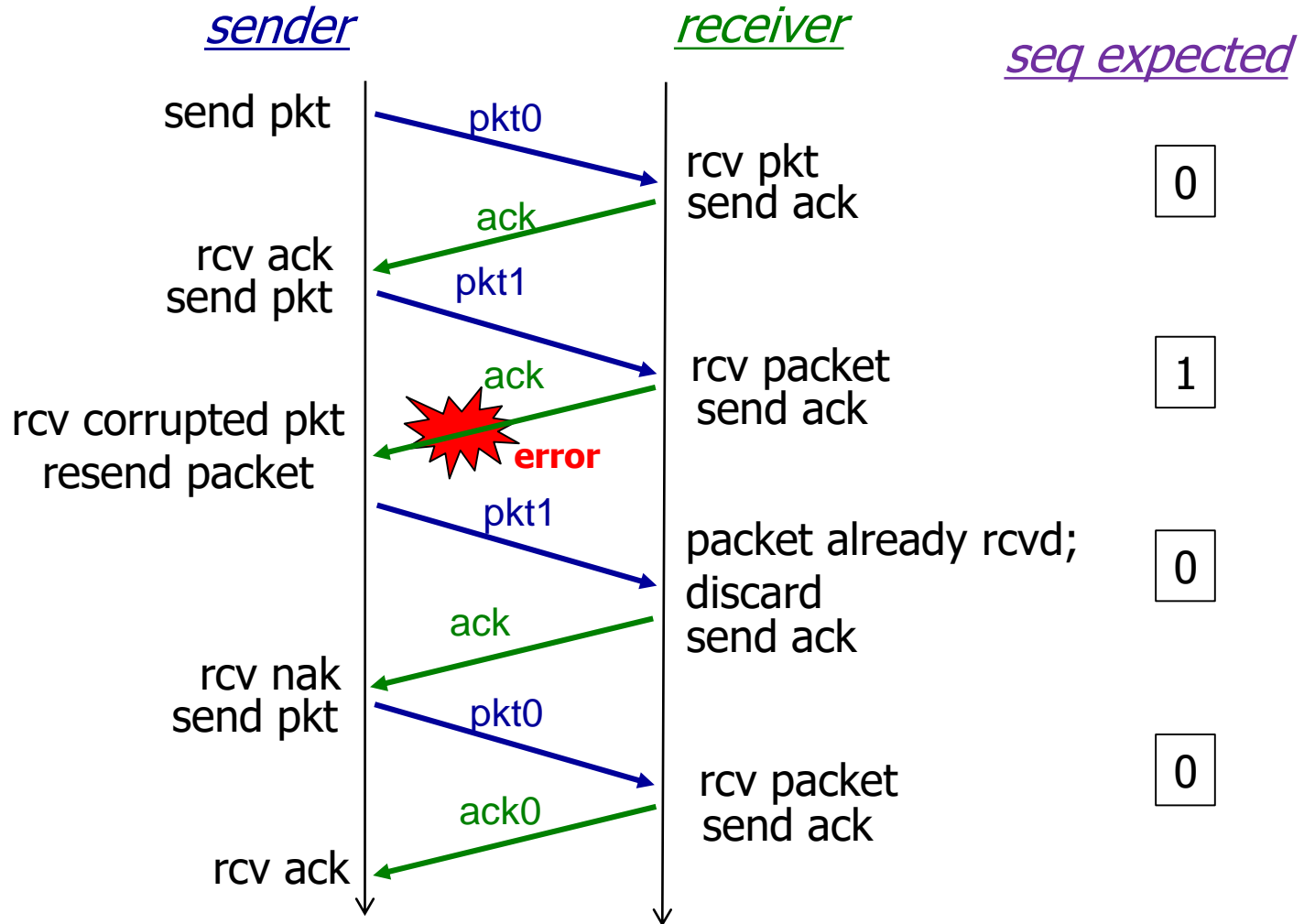
- ❖ sender doesn't know what happened at receiver!
- ❖ can't just retransmit: possible duplicate



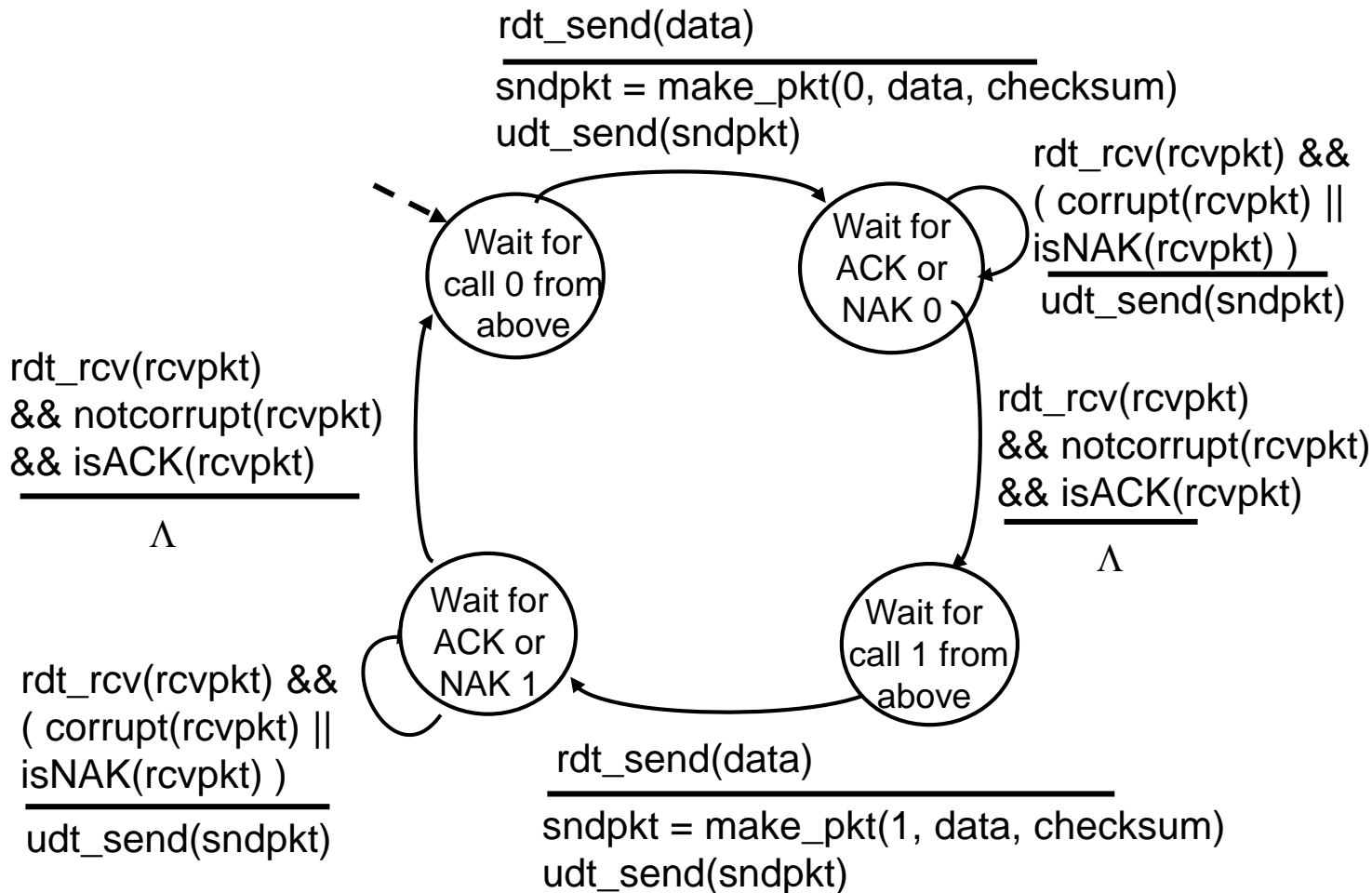
## handling duplicates:

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

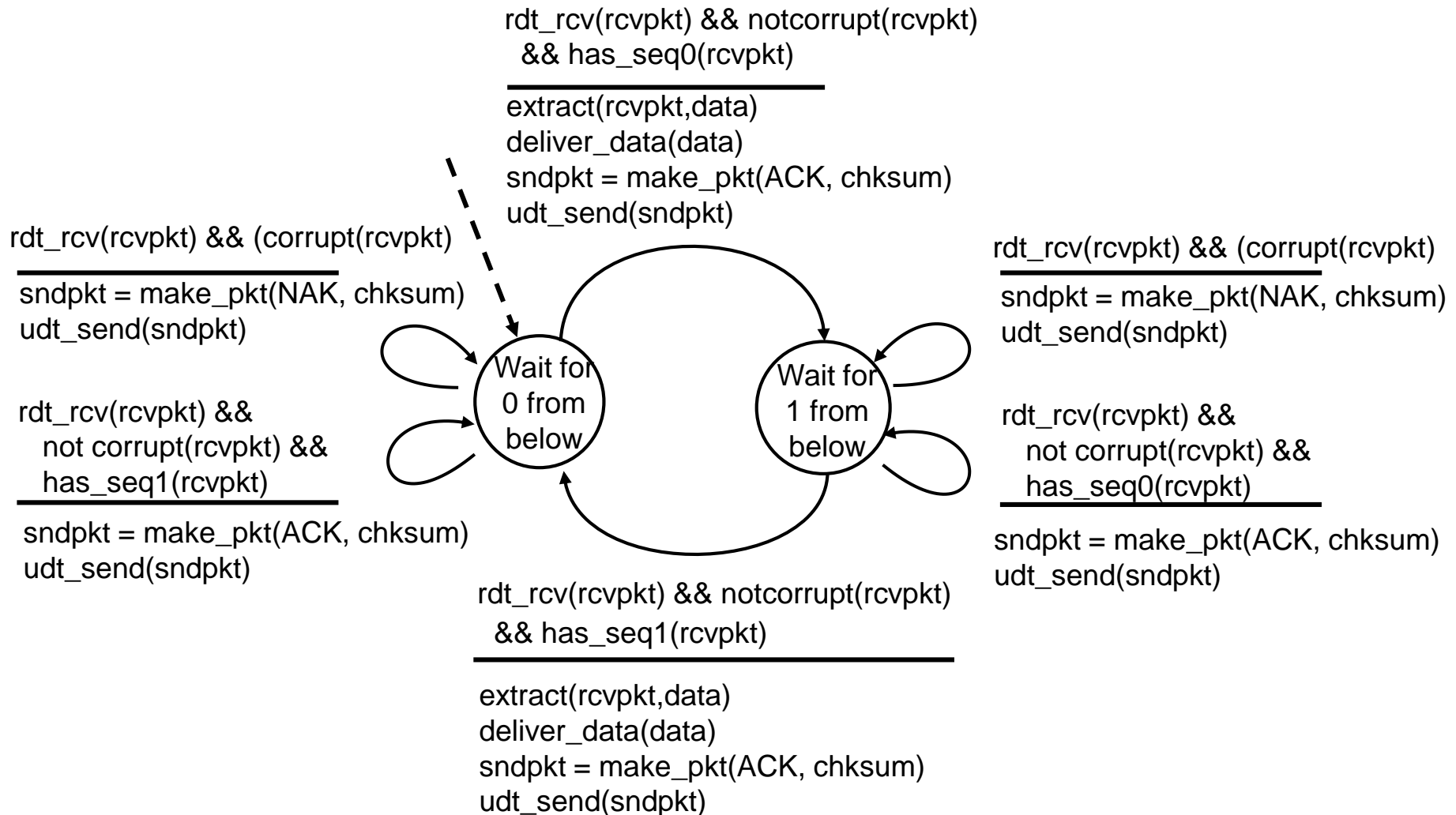
# rdt2.1: Frames with Seq No.



# rdt2.1: sender, handles garbled ACK/NAKs



# rdt2.1: receiver, handles garbled ACK/NAKs



# 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

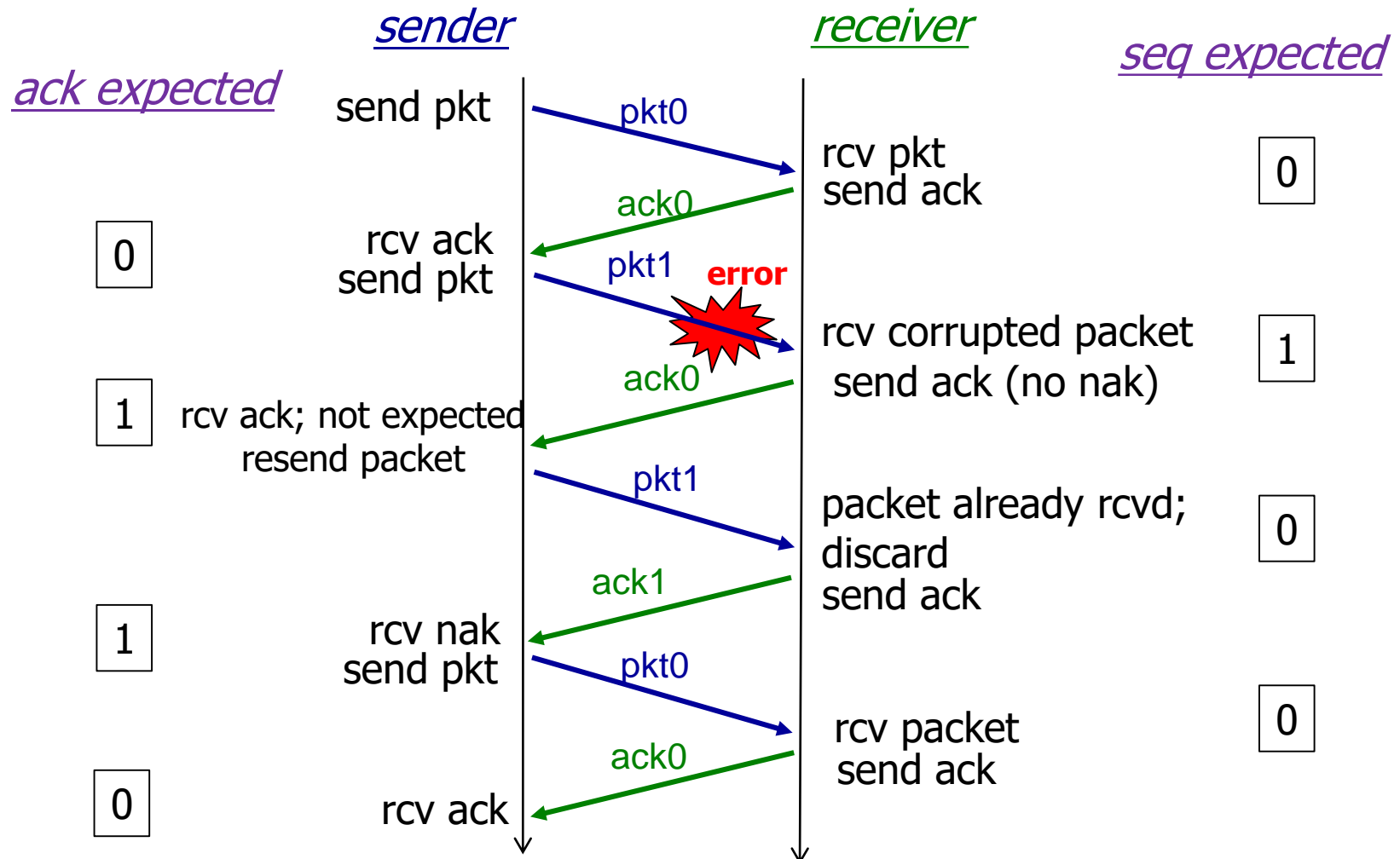
## receiver:

- ❖ 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

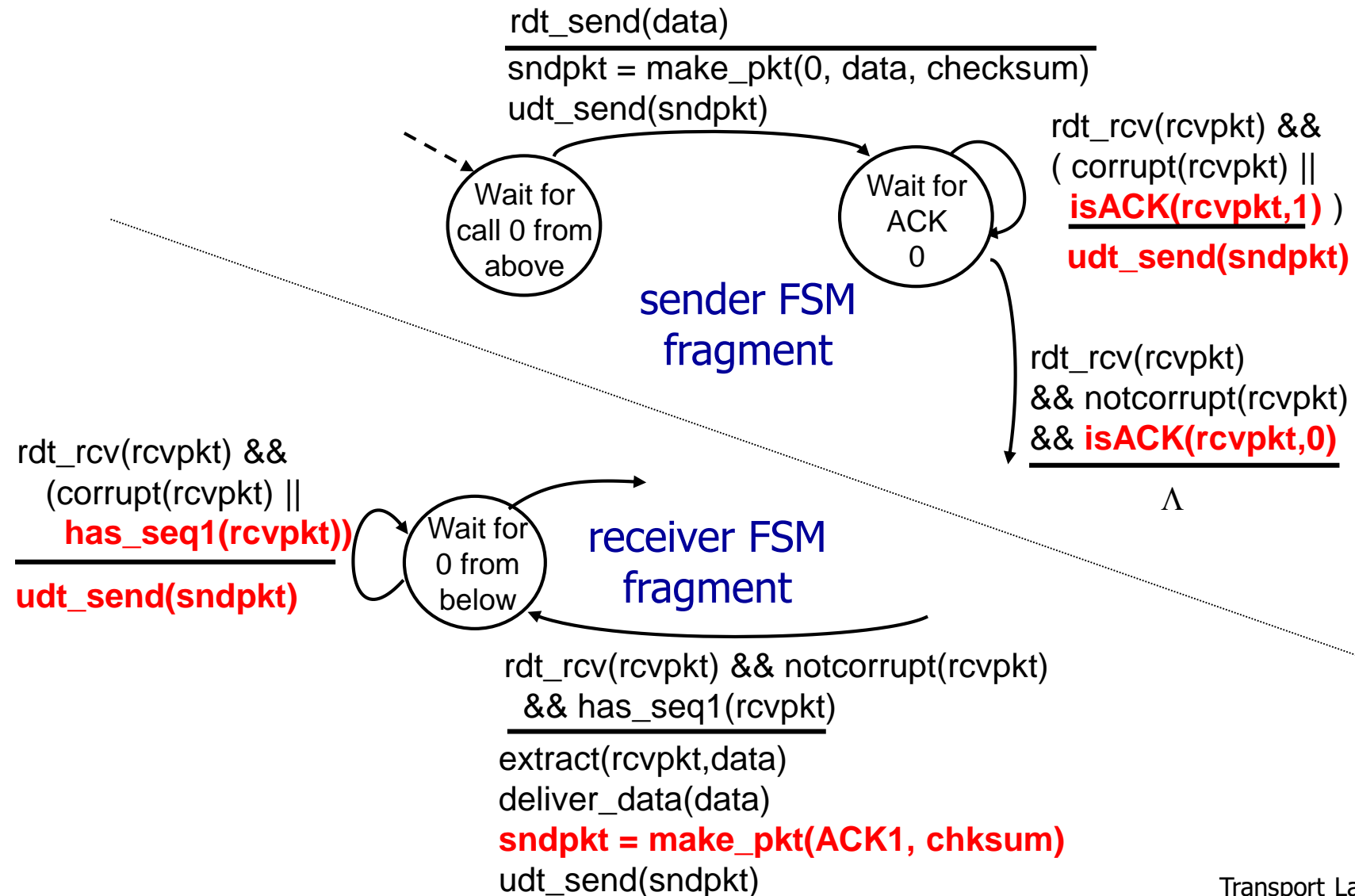
## rdt2.2: a NAK-free protocol

- ❖ 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*

# rdt2.1: No NAK



# rdt2.2: sender, receiver fragments





# rdt3.0: channels with errors and loss

## new assumption:

underlying channel can also lose packets (data, ACKs)

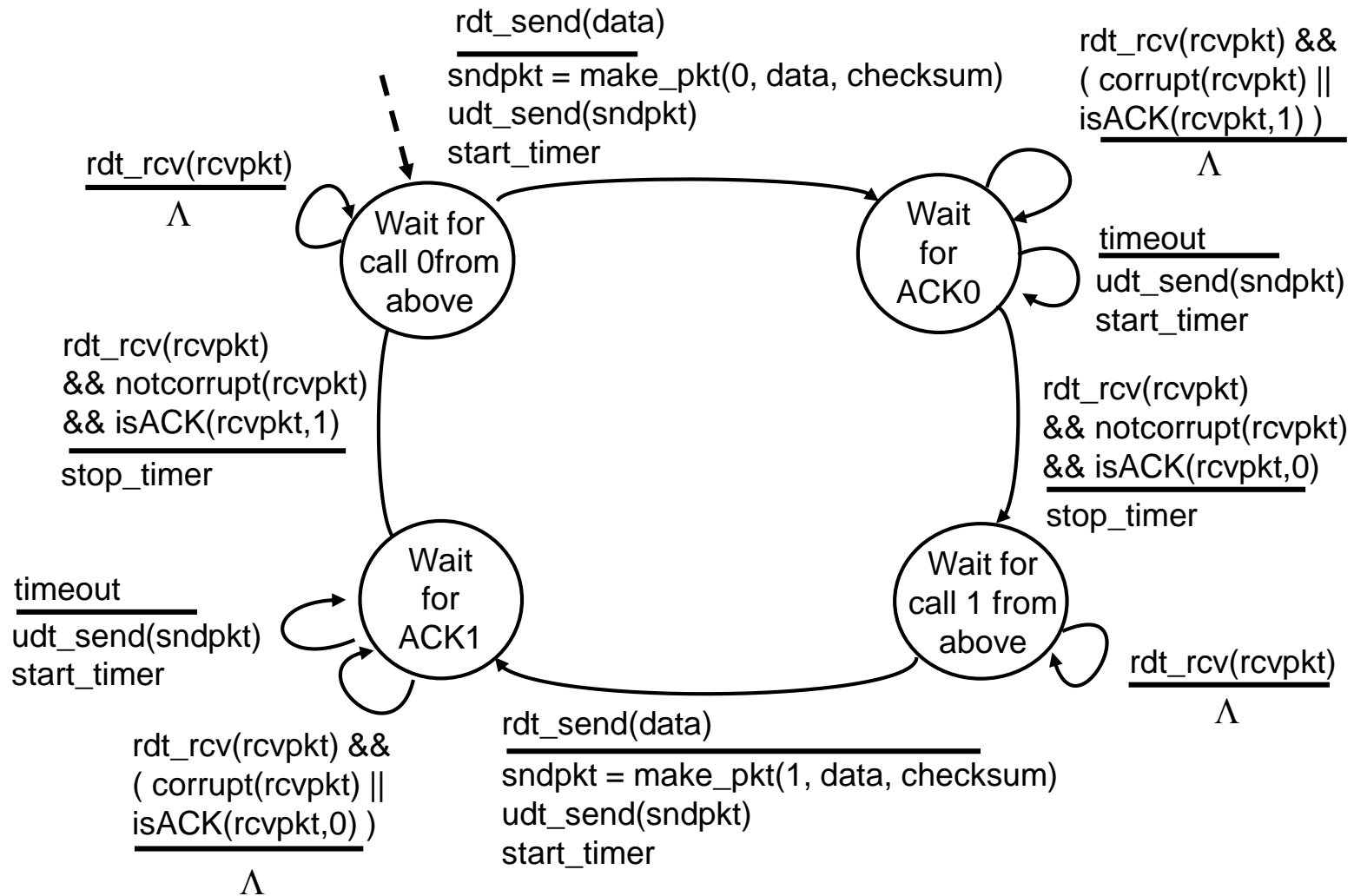
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

approach: sender waits “reasonable” amount of time for ACK

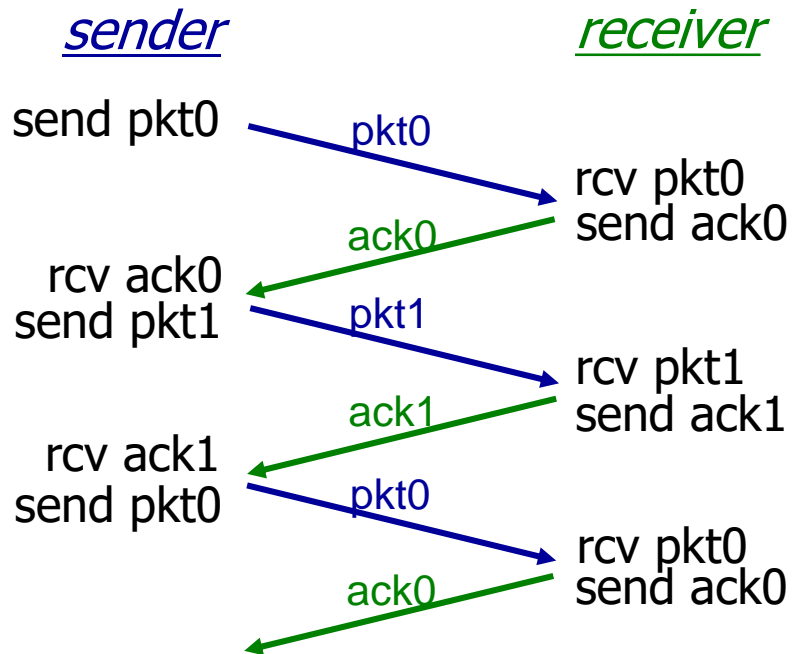
- ❖ retransmits if no ACK received in this time
- ❖ if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- ❖ requires countdown timer

Also known as alternating-bit protocol

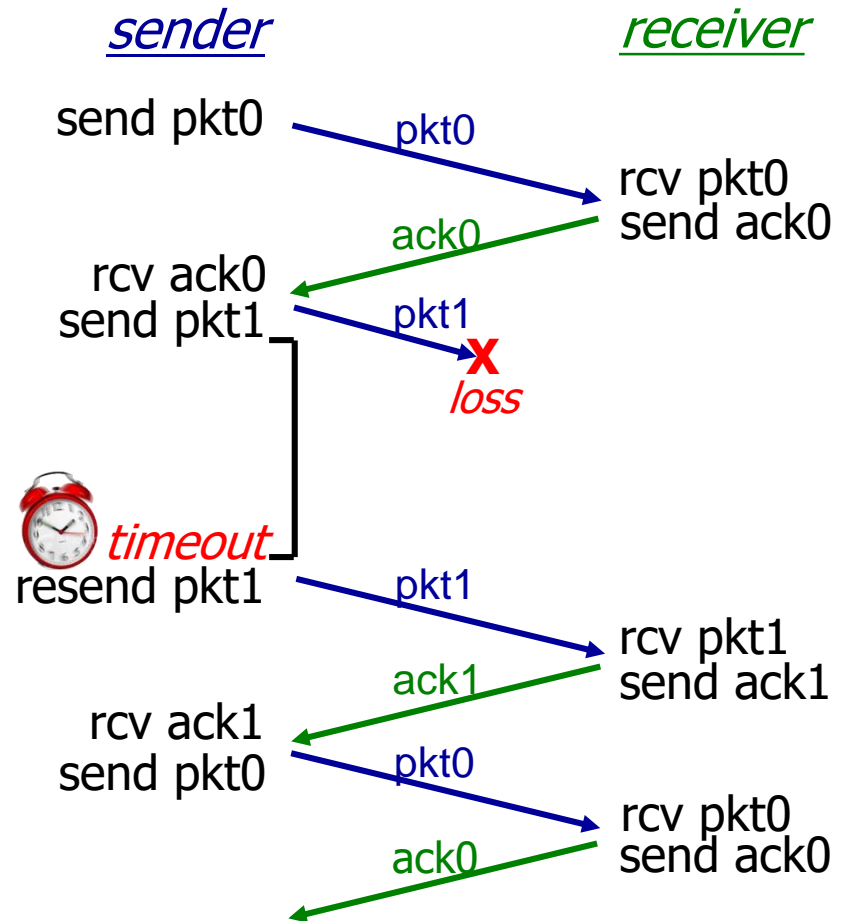
# rdt3.0 sender



# rdt3.0 in action

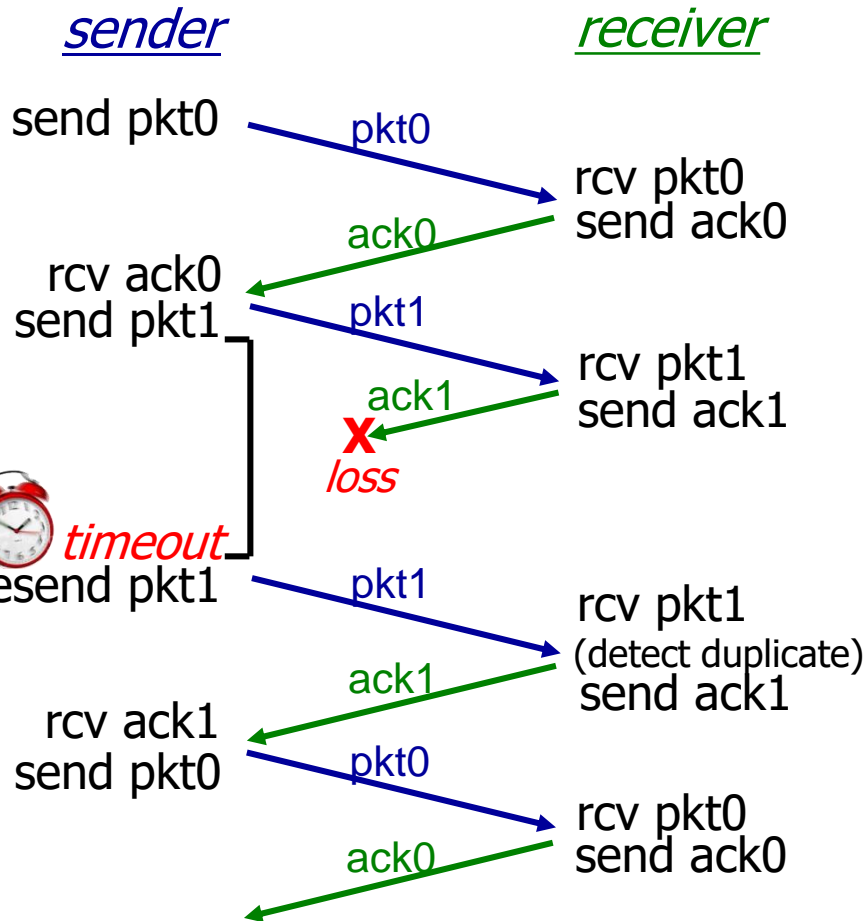


(a) no loss

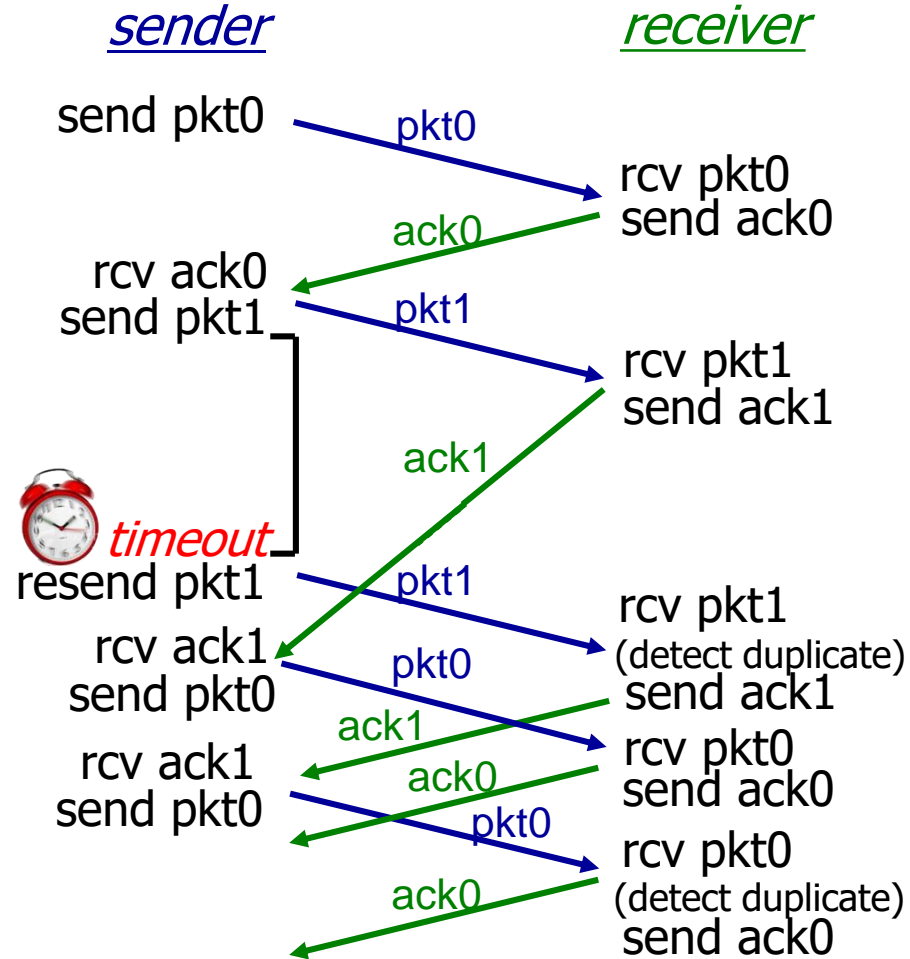


(b) packet loss

# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Determining the Value of Timeout

- ❖ Must be as long as the round-trip time (RTT) between the sender and receiver
- ❖ Easy for point to point link
- ❖ Difficult to compute end to end; need to consider
  - buffering at intermediate routers
  - time needed to process packet at each hop
- ❖ Waiting for worst case maximum delay would degrade performance
  - Determined dynamically as in TCP

# Performance of rdt3.0

- ❖ rdt3.0 is correct, but performance stinks
- ❖ e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

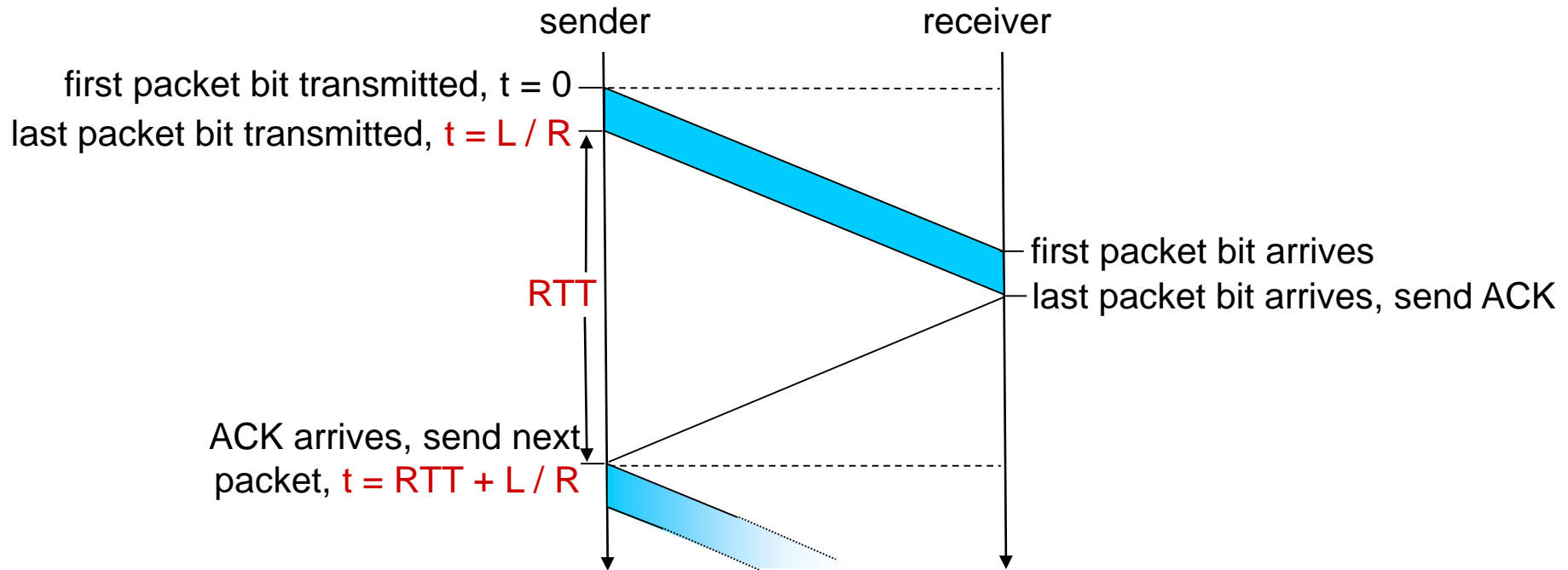
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microseconds}$$

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

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

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

# rdt3.0: stop-and-wait operation

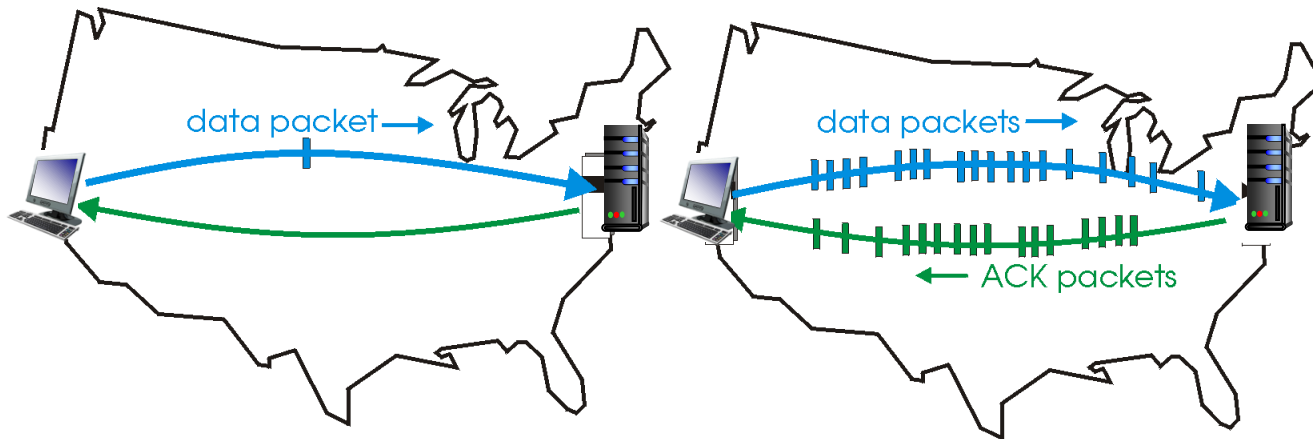


$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# Pipelined protocols

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

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

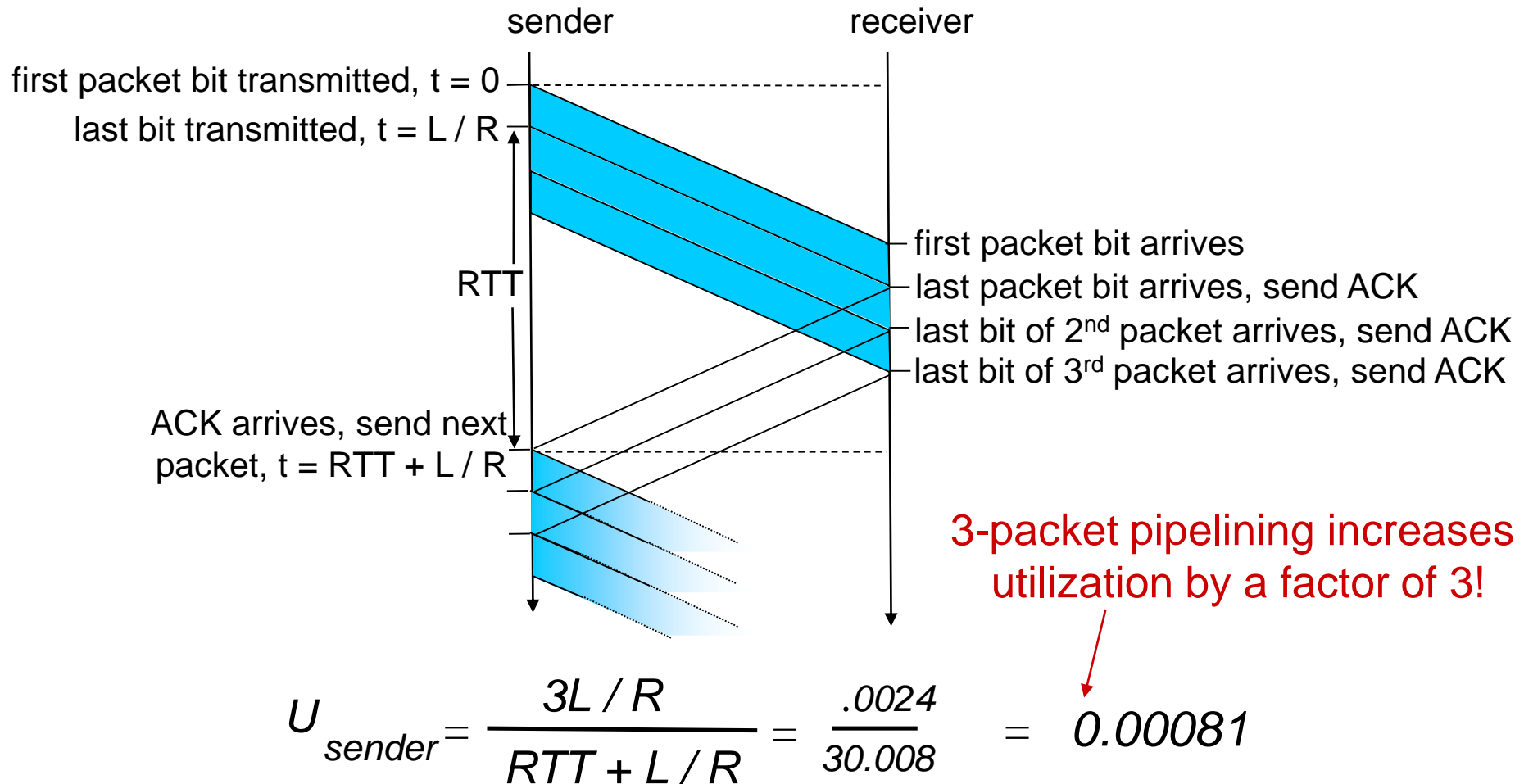


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation



# Pipelining: increased utilization



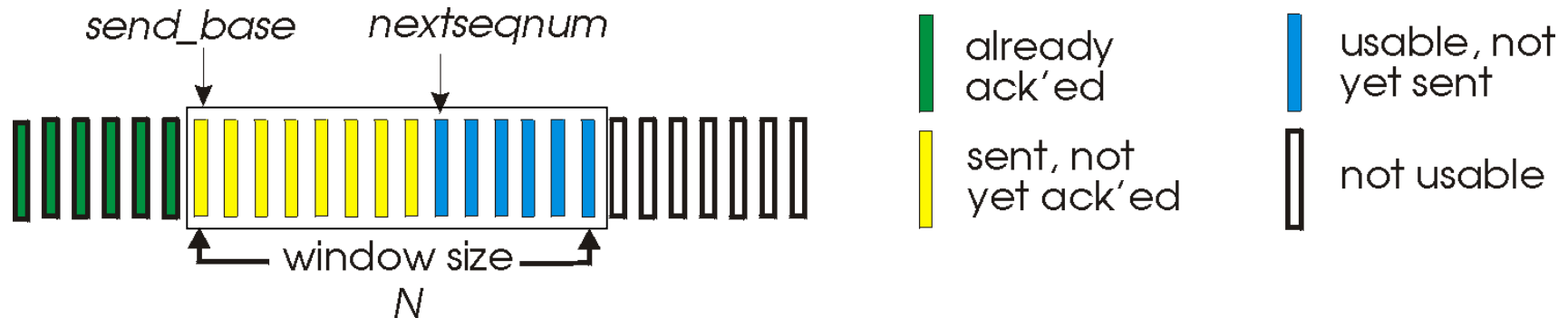
- ❖ two generic forms of pipelined protocols: *go-Back-N*, *selective repeat*

# Pipelined 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

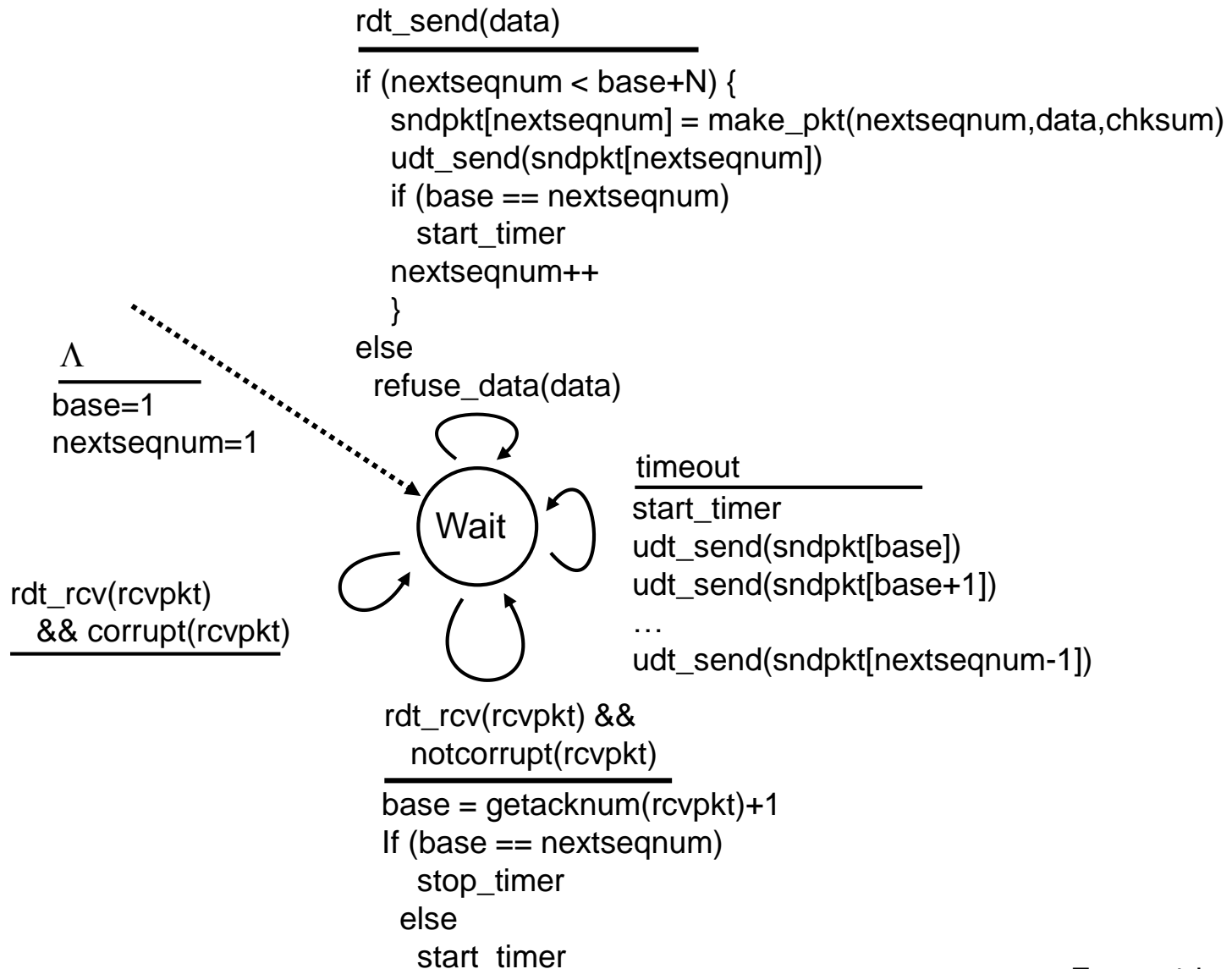
# Go-Back-N: sender

- ❖ k-bit seq # in pkt header
- ❖ “window” of up to N, consecutive unack’ed pkts allowed

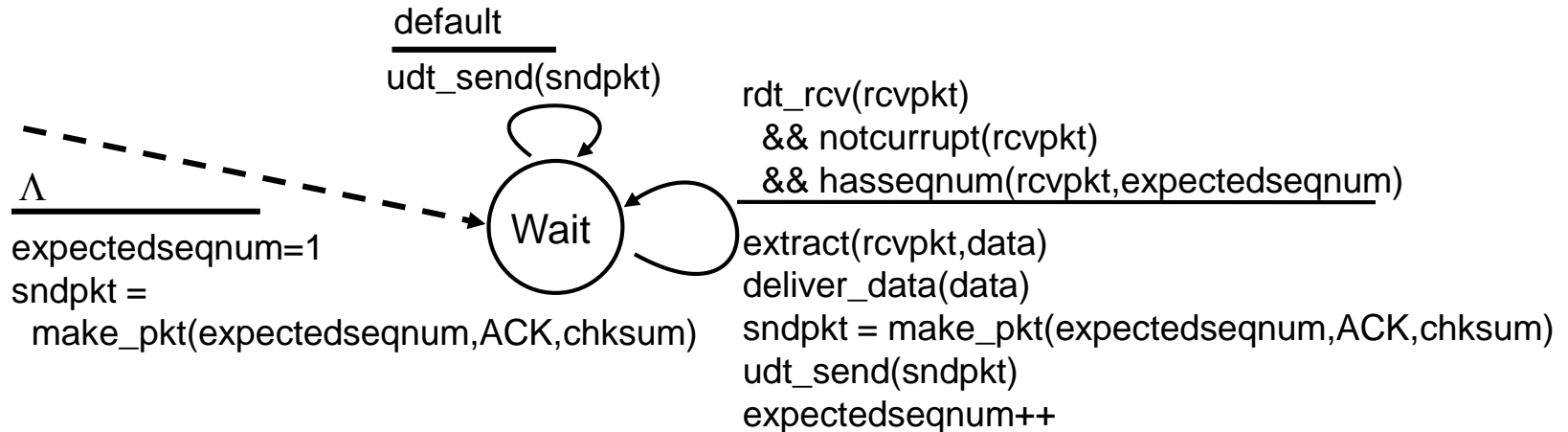


- ❖ ACK(n): ACKs all pkts up to, including seq # n - “*cumulative ACK*”
  - may receive duplicate ACKs (see receiver)
- ❖ timer for oldest in-flight pkt
- ❖ *timeout(n)*: retransmit packet n and all higher seq # pkts in window

# GBN: sender extended FSM



# GBN: receiver extended FSM



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 #

# GBN in action

sender window (N=4)

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8  
 0 1 2 3 4 5 6 7 8

sender

send pkt0  
 send pkt1  
 send pkt2  
 send pkt3  
 (wait)

rcv ack0, send pkt4  
 rcv ack1, send pkt5

ignore duplicate ACK



*pkt 2 timeout*

send pkt2  
 send pkt3  
 send pkt4  
 send pkt5

receiver

receive pkt0, send ack0  
 receive pkt1, send ack1

receive pkt3, discard,  
 (re)send ack1

receive pkt4, discard,  
 (re)send ack1

receive pkt5, discard,  
 (re)send ack1

rcv pkt2, deliver, send ack2  
 rcv pkt3, deliver, send ack3  
 rcv pkt4, deliver, send ack4  
 rcv pkt5, deliver, send ack5

**X loss**

# Problems of GBN

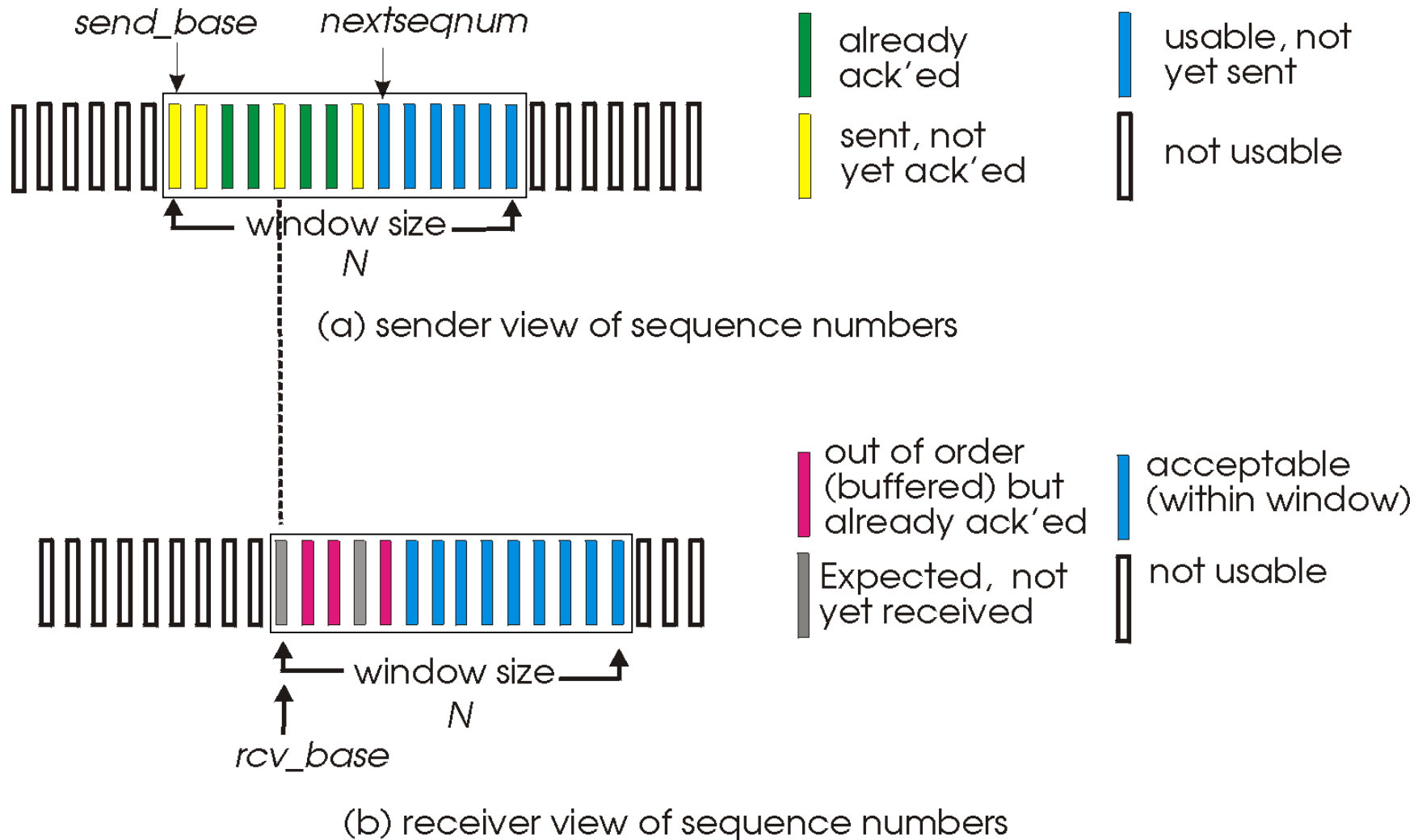
- ❖ For a large window size, a single packet error causes retransmission of a large number of packets
- ❖ The pipeline becomes filled with unnecessary retransmissions
- ❖ Can be solved by
  - accepting out of order packets by the receiver
  - selective retransmission
    - Selective Repeat (SR) protocol

# Pipelined protocols: Selective repeat

- ❖ sender can have up to  $N$  unack'ed packets in pipeline
- ❖ 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: sender, receiver windows



# Selective repeat

## sender

### data from above:

- ❖ if next available seq # in window, send pkt

### timeout(n):

- ❖ resend pkt n, restart timer

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

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

## receiver

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

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

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

- ❖ ACK(n)

### otherwise:

- ❖ ignore

# Selective repeat in action

sender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

record ack3 arrived



*pkt 2 timeout*

send pkt2

record ack4 arrived

record ack4 arrived

receiver

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,  
send ack3

receive pkt4, buffer,  
send ack4

receive pkt5, buffer,  
send ack5

rcv pkt2; deliver pkt2,  
pkt3, pkt4, pkt5; send ack2

*Q: what happens when ack2 arrives?*

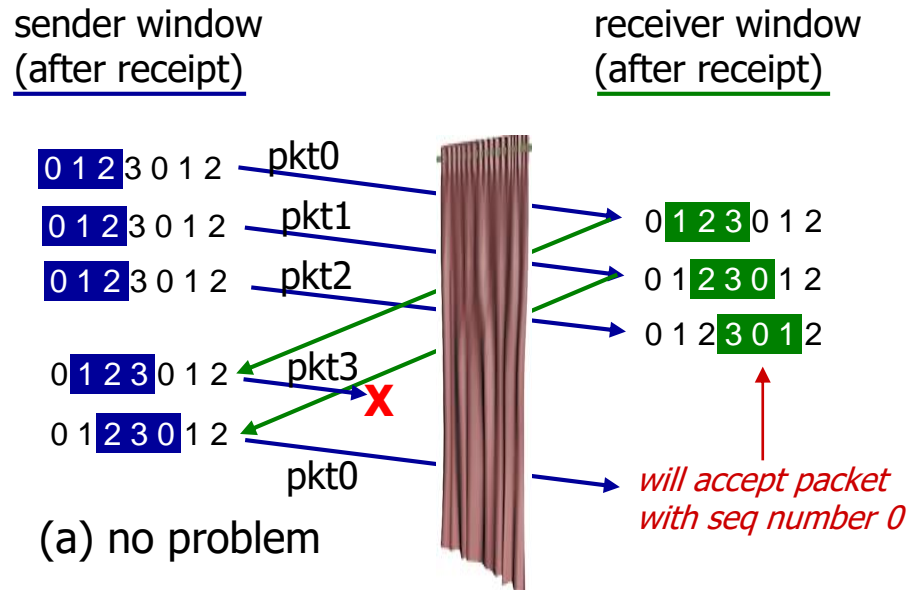
# Problems with Finite Sequence No.

example:

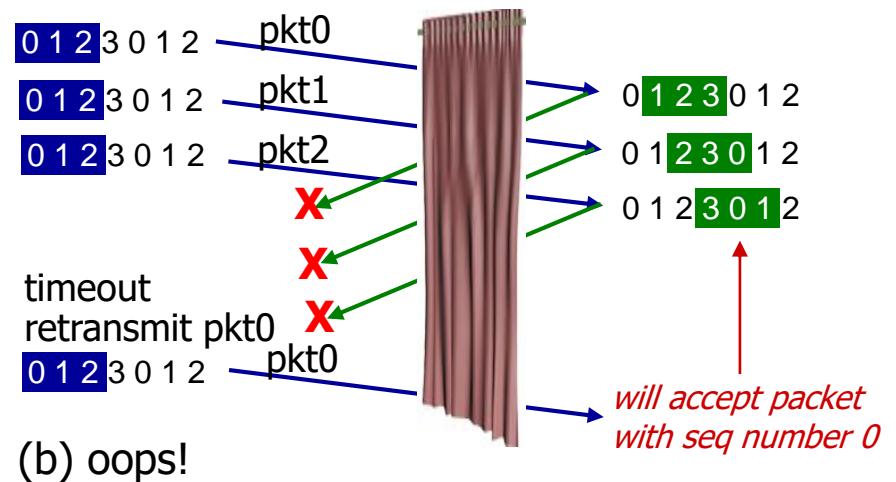
- ❖ seq #'s: 0, 1, 2, 3
- ❖ window size=3
- ❖ receiver sees no difference in two scenarios!
- ❖ duplicate data accepted as new in (b)

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

Q: What should the relation in GBN?

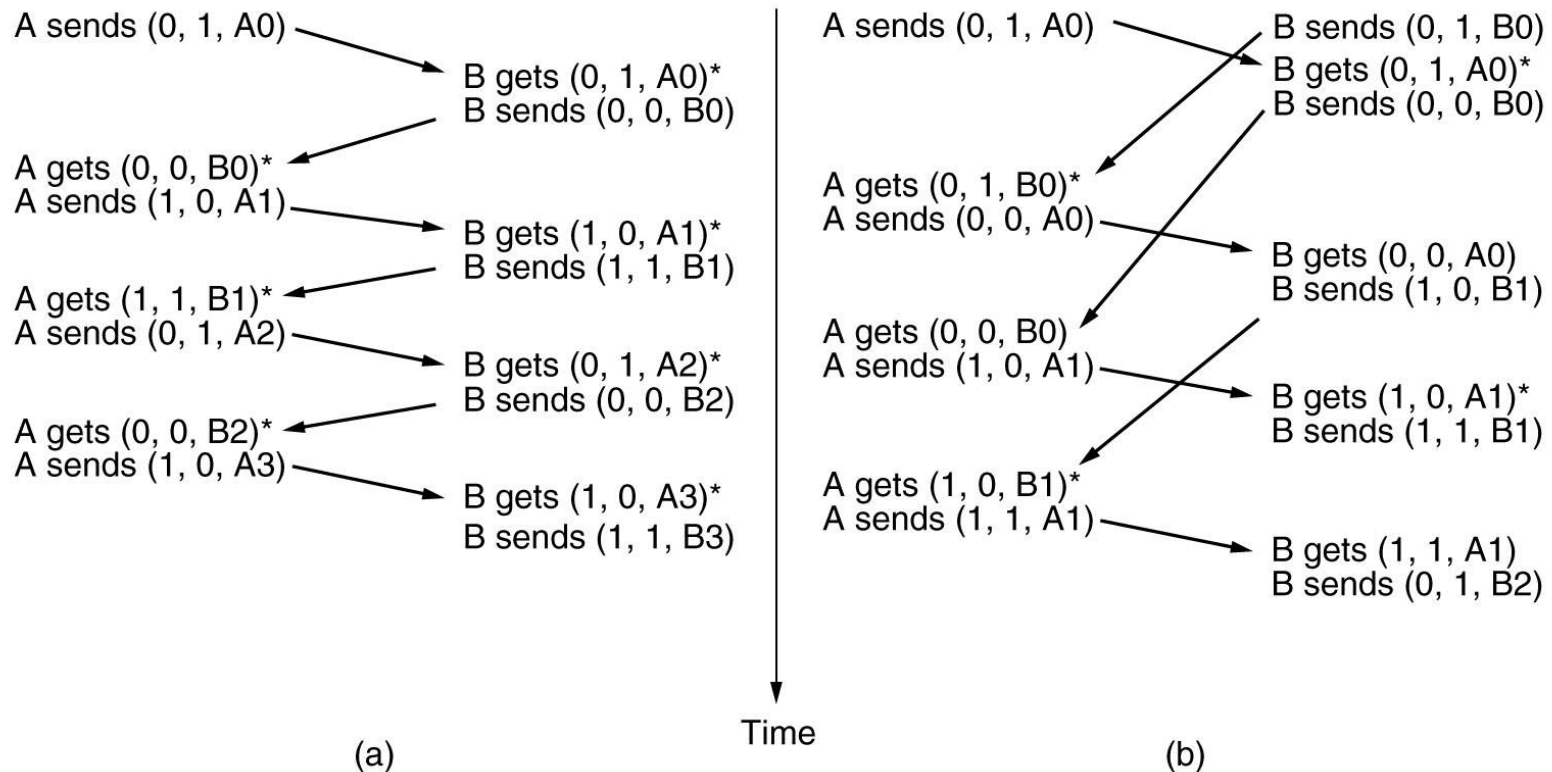


*receiver can't see sender side.  
receiver behavior identical in both cases!  
something's (very) wrong!*



# Bi-directional Transfer

- ❖ Send ack in the same frame for data
- ❖ Known as **Piggybacking**



# Delayed Duplicate Problem

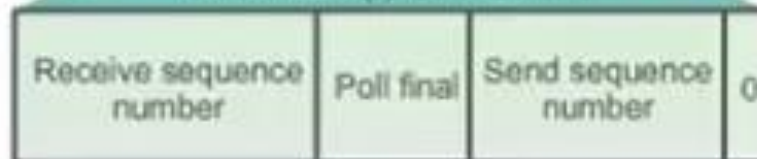
- ❖ Old copies of a packet may appear at the sender or receiver
- ❖ Since sequence numbers wrap around, care must be taken to guard against such duplicate packet
  - Do not use a sequence number, until all the earlier packets with the same sequence number have died out.
  - TCP extension for high speed network assumes maximum packet lifetime of approx. three minutes

# Sliding Window at Link Layer: HDLC

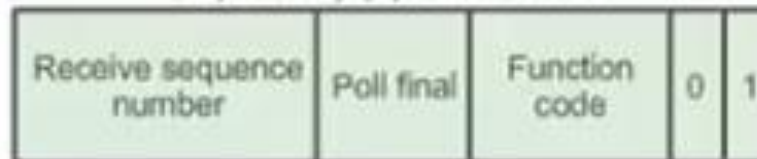
## HDLC Frame Structure



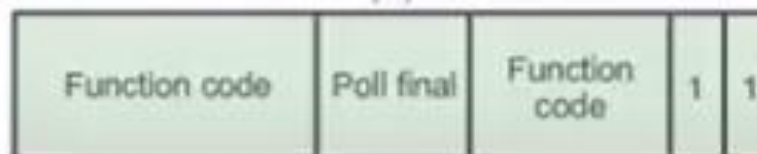
### Information (I) frame format



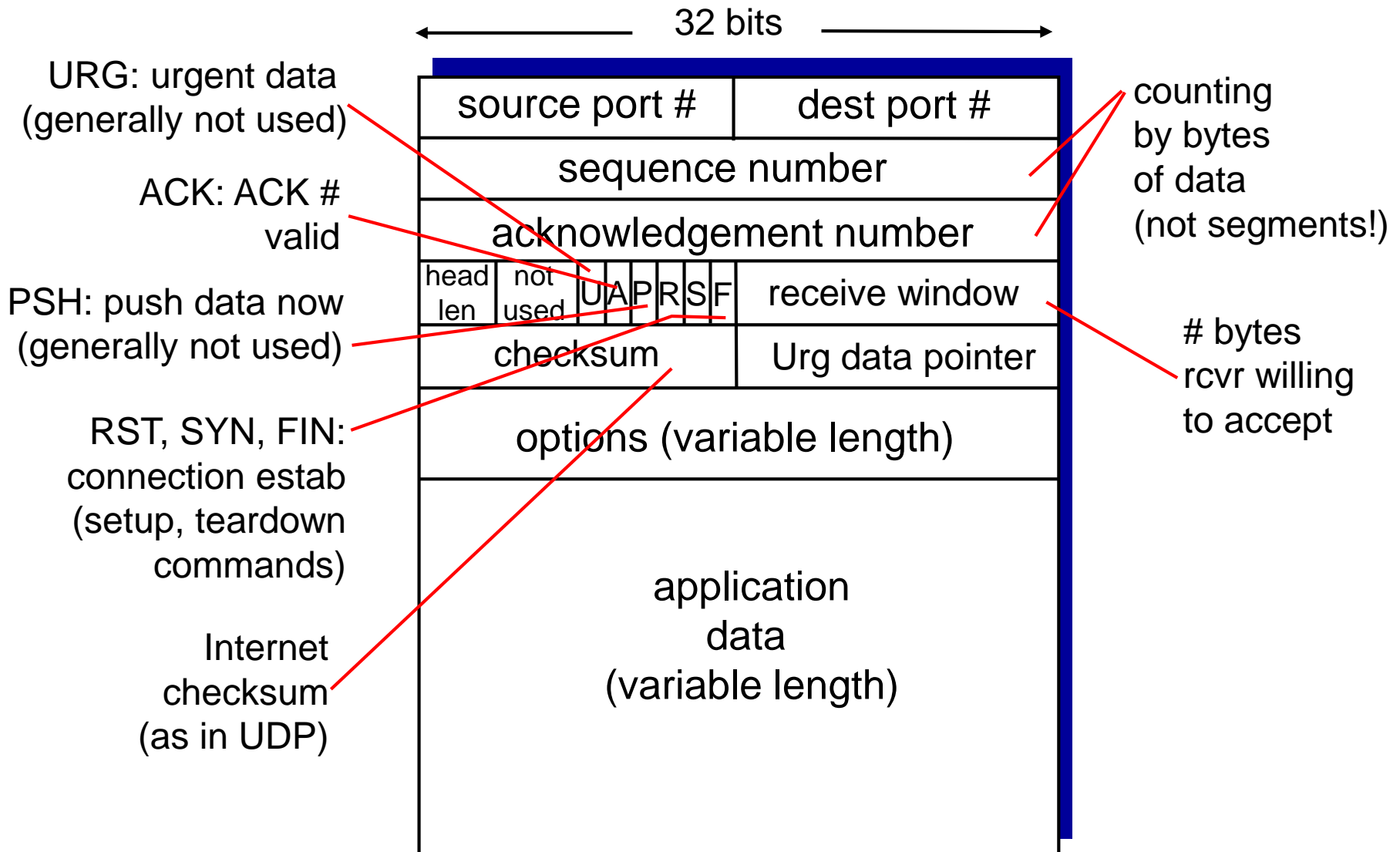
### Supervisory (S) frame format



### Unnumbered (U) frame format



# Sliding Window at Transport Layer: TCP



## TCP Header



# Sliding Window in TCP (2)

## sequence numbers:

- byte stream “number” of first byte in segment’s data

## acknowledgements:

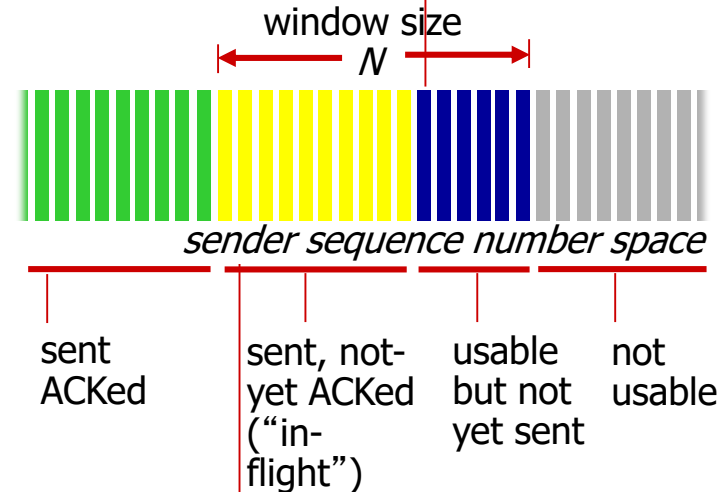
- seq # of next byte expected from other side
- cumulative ACK

**Q:** how receiver handles out-of-order segments

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

outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
	A
checksum	urg pointer

# TCP round trip time, timeout

- ❖ **timeout interval:** **EstimatedRTT** plus “safety margin”
  - large variation in **EstimatedRTT** -> larger safety margin
- ❖ estimate **SampleRTT** deviation from **EstimatedRTT**:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically,  $\beta = 0.25$ )

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑  
estimated RTT

↑  
“safety margin”

# TCP ACK generation [RFC 1122, RFC 2581]

<i>event at receiver</i>	<i>TCP receiver action</i>
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 <i>duplicate ACK</i> , 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 fast retransmit

- ❖ time-out period often relatively long:
  - long delay before resending lost packet
- ❖ detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

## *TCP fast retransmit*

if sender receives 3 ACKs for same data (“triple duplicate ACKs”), resend unacked segment with smallest seq #

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

# TCP fast retransmit

