



UNIVERSITY OF
WATERLOO

CS 456/656

Computer Networks

Lecture 6: Transport Layer – Part 2

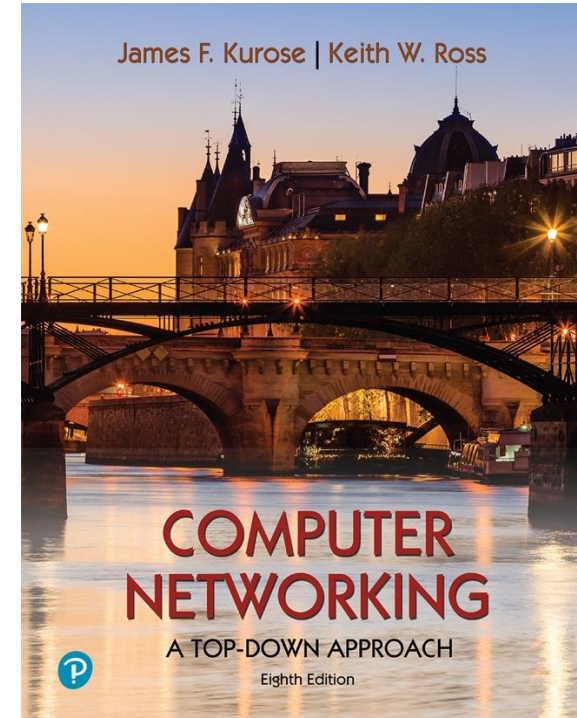
Mina Tahmasbi Arashloo and Uzma Maroof

Fall 2025

A note on the slides

Adapted from the slides that
accompany this book.

All material copyright 1996-2023
J.F Kurose and K.W. Ross, All Rights Reserved



Computer Networking: A Top-Down Approach

8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport layer: roadmap

- Transport-layer overview
- 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



Transport layer: roadmap

- Transport-layer overview
- 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



Reliable data transfer (rdt)

- How can I make sure all bytes are delivered reliably? Reliable data transfer (rdt)
- One of the most important services a transport protocol can provide over an unreliable network layer

Principles of RDT - Agenda

- `rdt` at a glance
- Stop-and-wait approach
 - sender sends one pkt, then waits for receiver's response
- Pipelined approach
 - Go-back-N (GBN)
 - Selective Repeat (SR)

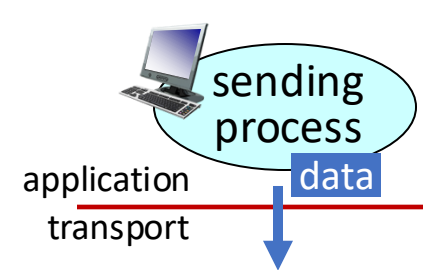
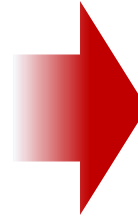
Principles of RDT - Agenda

- rdt at a glance
- Stop-and-wait approach
 - sender sends one pkt, then waits for receiver's response
- Pipelined approach
 - Go-back-N (GBN)
 - Selective Repeat (SR)

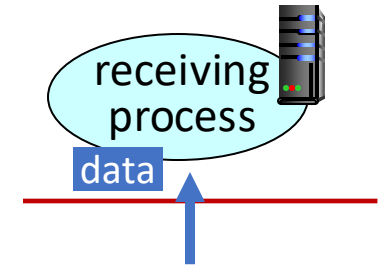
Reliable data transfer at a glance



reliable service *abstraction*



sender-side of
reliable data
transfer protocol



receiver-side
of reliable data
transfer protocol

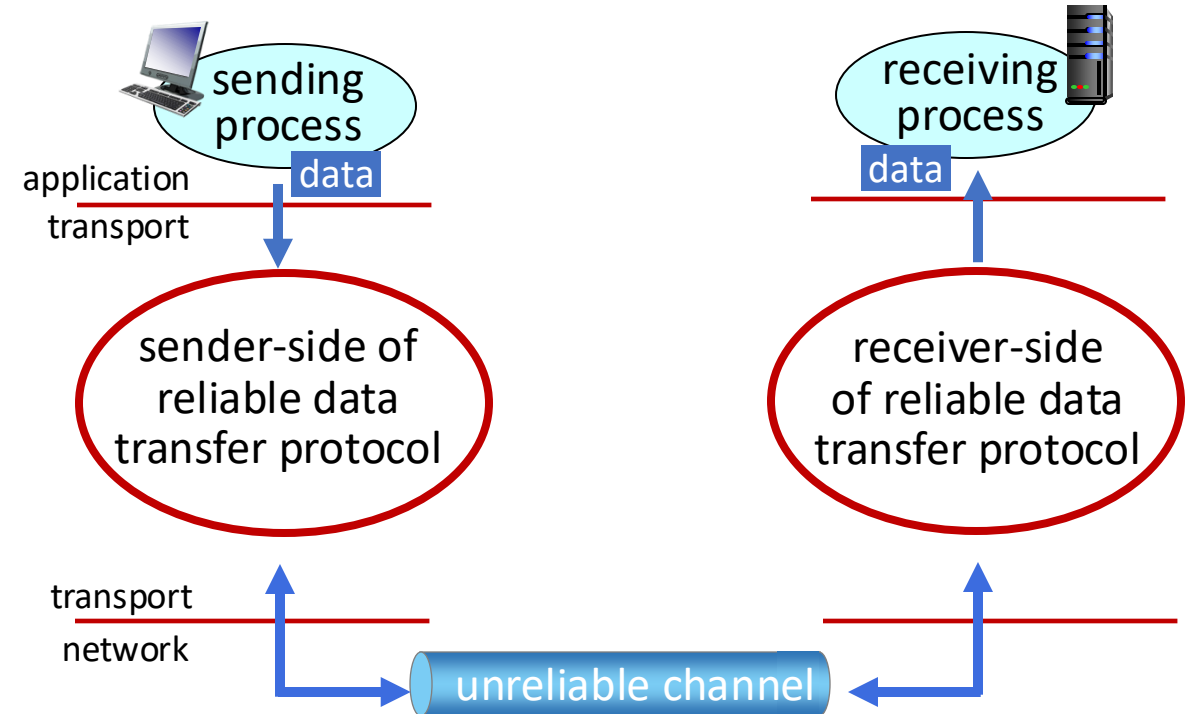
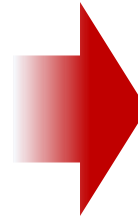


reliable service *implementation*

Reliable data transfer at a glance

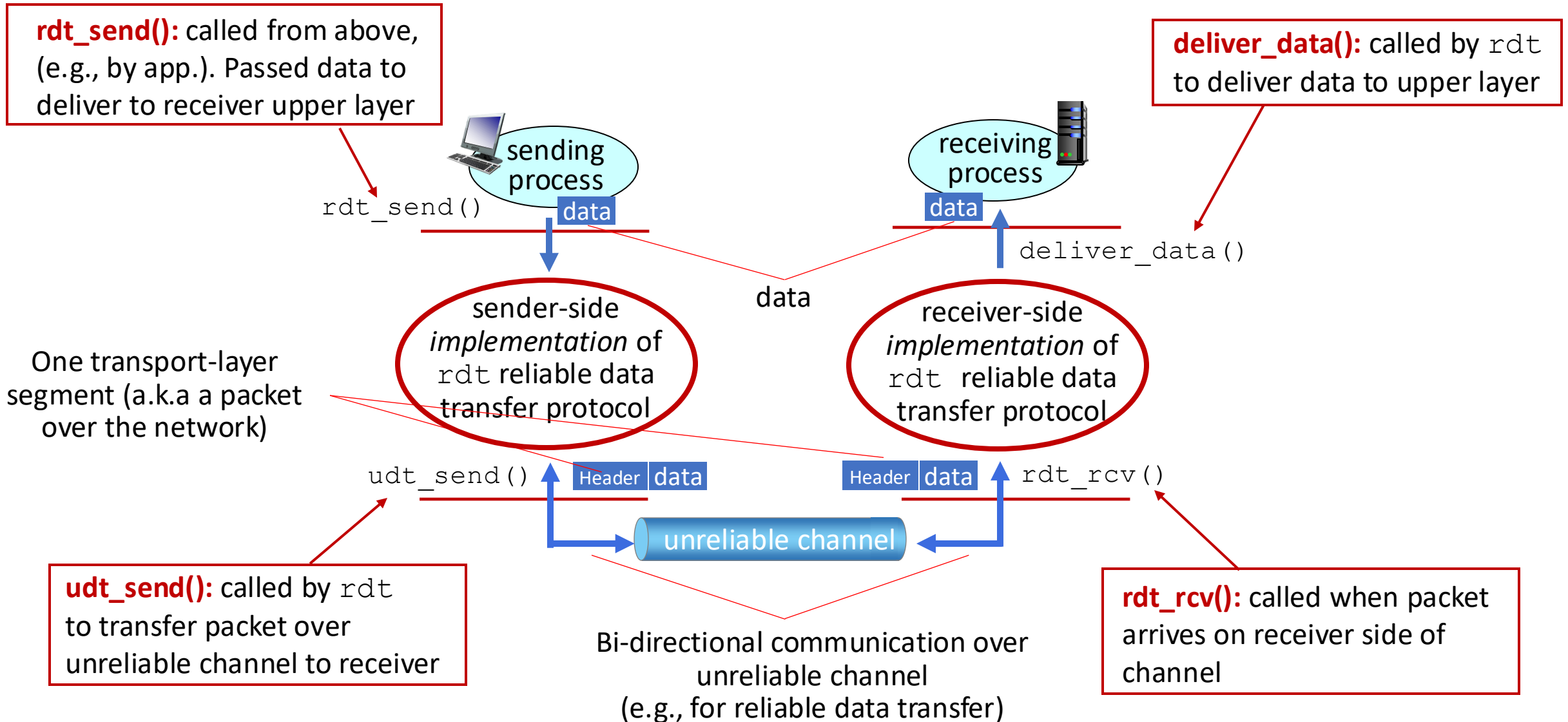


reliable service *abstraction*



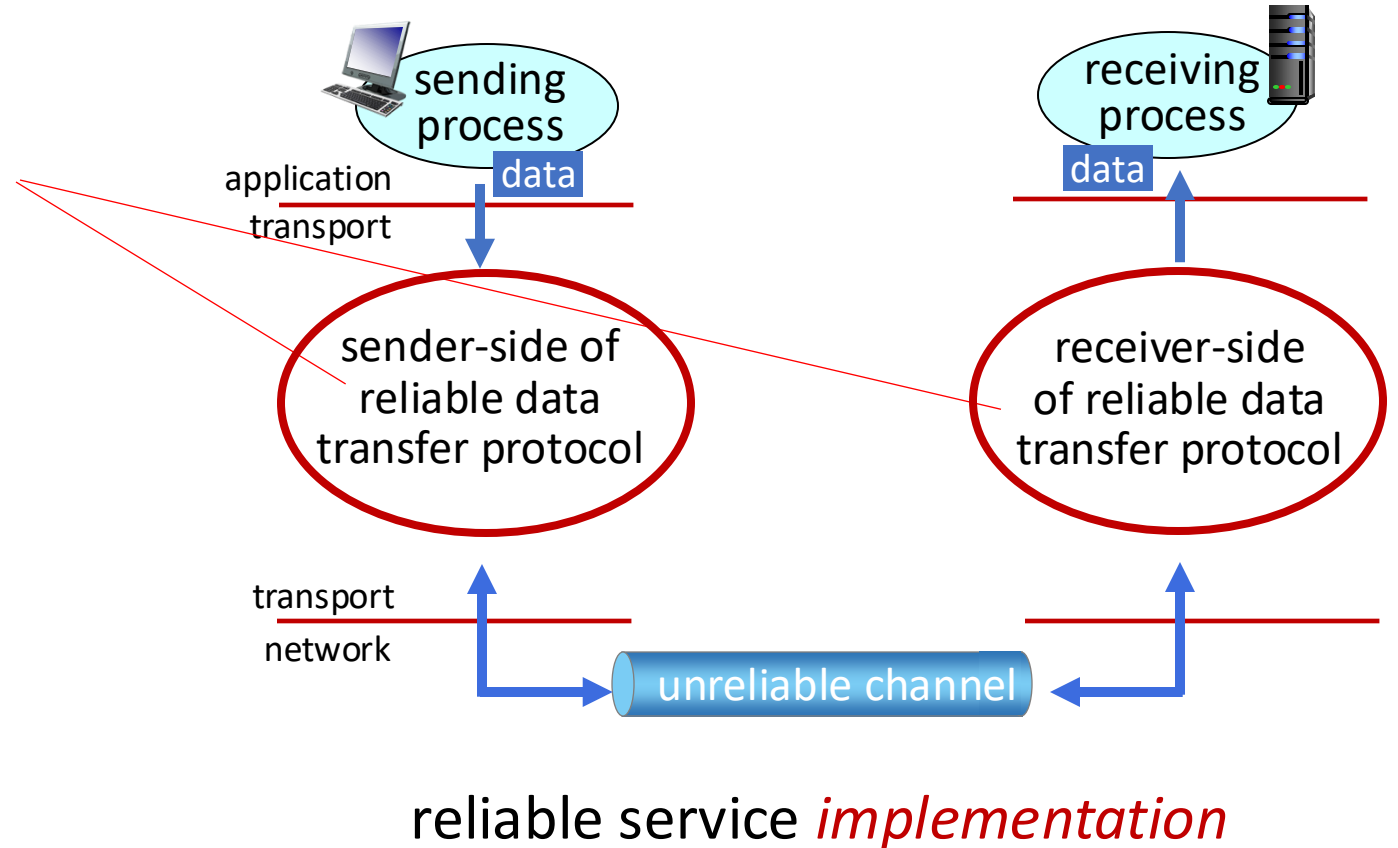
reliable service *implementation*

Reliable data transfer protocol (rdt): interfaces



Reliable data transfer at a glance

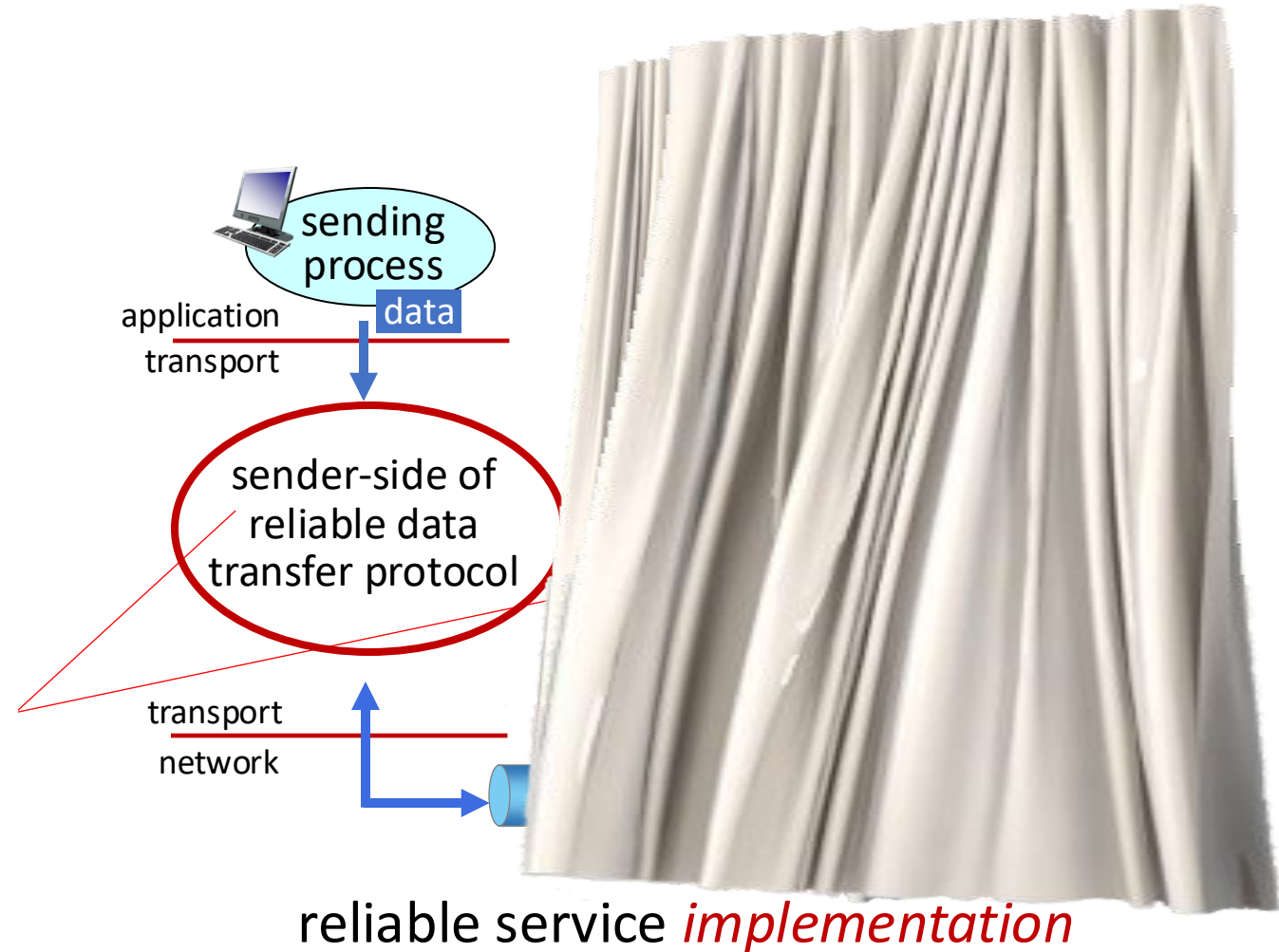
- Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel
 - Bit-errors
 - Pkt loss
 - Out-of-order delivery
- Requirements of `rdt`
 - No corrupted bits
 - All bits are delivered
 - No duplicates
 - Data is received in the order sent



Reliable data transfer at a glance

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

- unless communicated via a message



Reliable data transfer: getting started

We will:

- incrementally develop sender, receiver sides of reliable data transfer protocol (`rdt`)
- We will discuss a unidirectional data transfer
 - but remember, each end of the communication can act both as a sender and a receiver
 - Data and control packets can flow in both directions
- achieve `rdt` based on **error-detection + retransmission**
 - General approach to reliable data transfer in different layers

Tools for reliable data transfer (rdt)

Detecting “errors” – i.e., lost, out of order, or corrupt segments

- Sequence number
 - Identify data segments and their order
 - Avoid duplicate delivery
 - Maintain in-order delivery
- Receiver feedback
 - Positive acknowledge (ACK)
 - I have received these segments!
 - Negative acknowledge (NAK)
 - I have not received these segments!
- Timer expiration
 - Detect pkt lost in the absence of feedback
- Checksum
 - Detect bit errors
 - Used in many layers and protocols

How do we recover?

Sender retransmission

Principles of RDT - Agenda

- rdt at a glance
- Stop-and-wait approach
 - sender sends one pkt, then waits for receiver's response
- Sliding-window approach
 - Go-back-N (GBN)
 - Selective Repeat (SR)

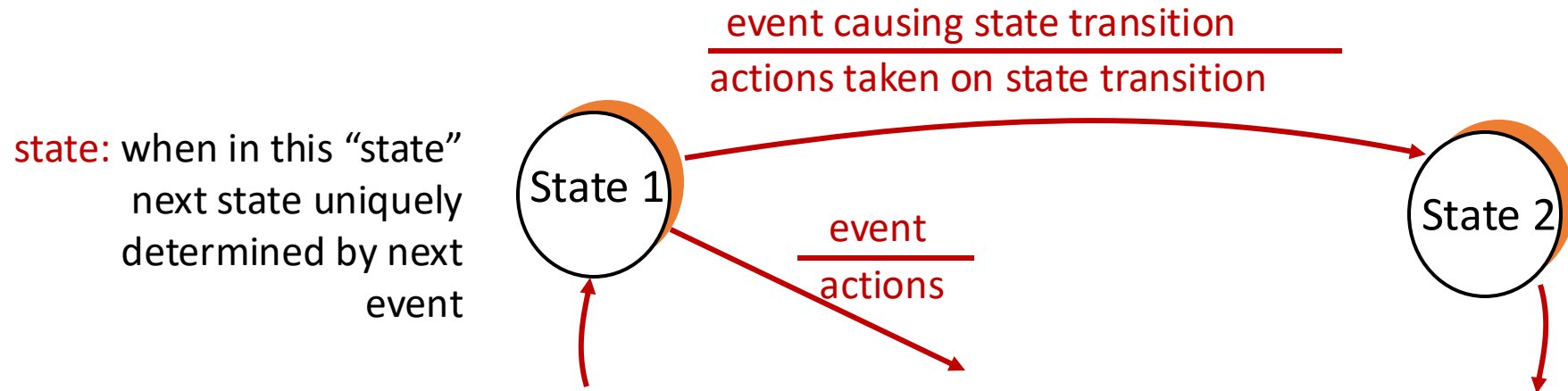
Stop and wait approach

- Send a segment
- Wait to make sure it is delivered properly
- Then send the next one

Reliable data transfer using FSMs

We will:

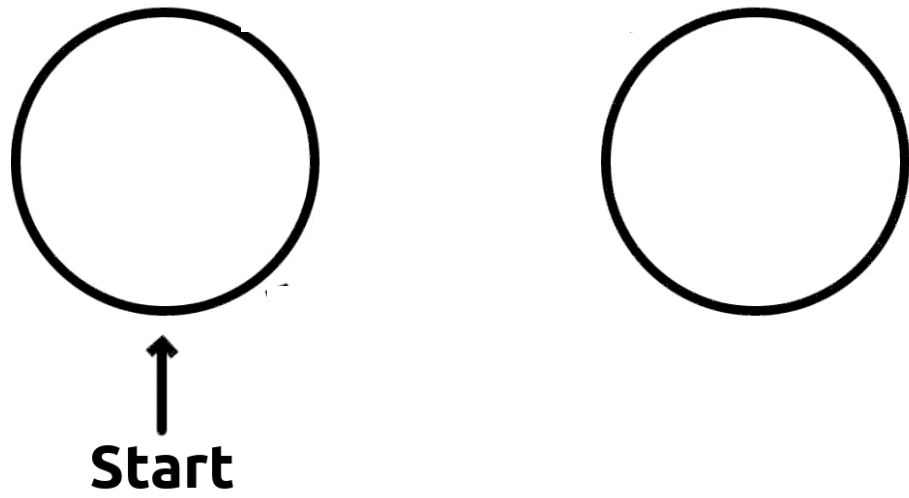
- use finite state machines (FSM) to specify sender, receiver



Sidenote: Finite State Machine (FSM) Refresher



Sidenote: Finite State Machine (FSM) Refresher



State	Event	Action	Next State

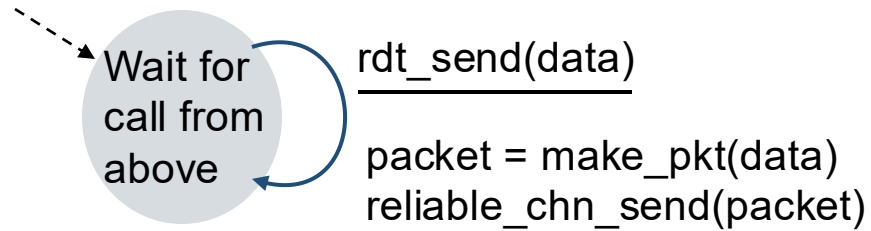


Warm-up: What if the underlying channel is reliable?

- underlying channel perfectly reliable
 - no bit errors
 - no loss of packets
- We use separate FSMs to show the logic of the sender & receiver side:
 - sender sends data into underlying channel
 - receiver read data from underlying channel
- Remember that a host can act simultaneously as a sender and receiver

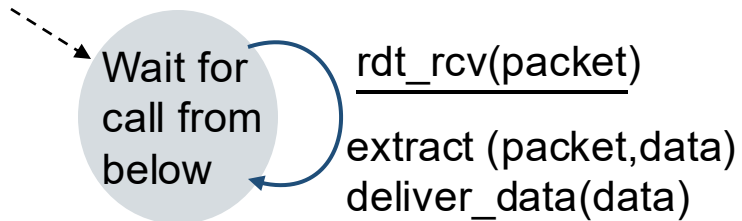
Warm-up: What if the underlying channel is reliable?

When will sender come out of this state???



a. sending side

State	Event	Action	Next State

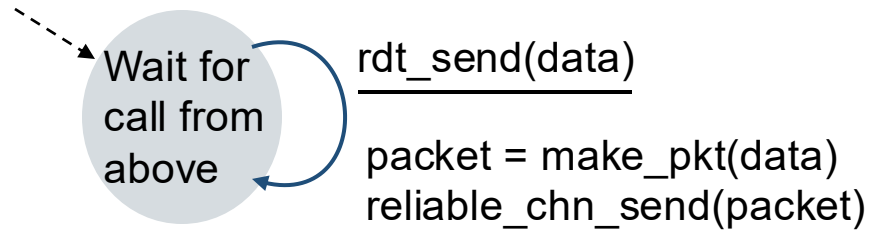


b. receiving side

State	Event	Action	Next State

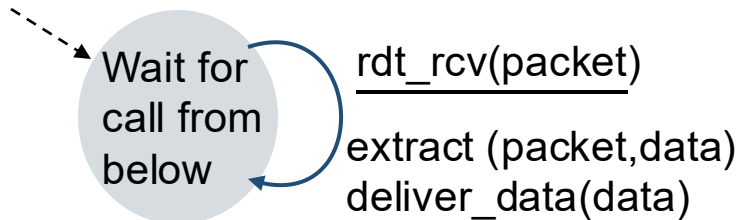
Warm-up: What if the underlying channel is reliable?

When will sender come out of this state???



a. sending side

State	Event	Action	Next State
Wait for call from above (data from application)	App-Layer says to send data	Send to network layer	Wait for call from above (data from application)



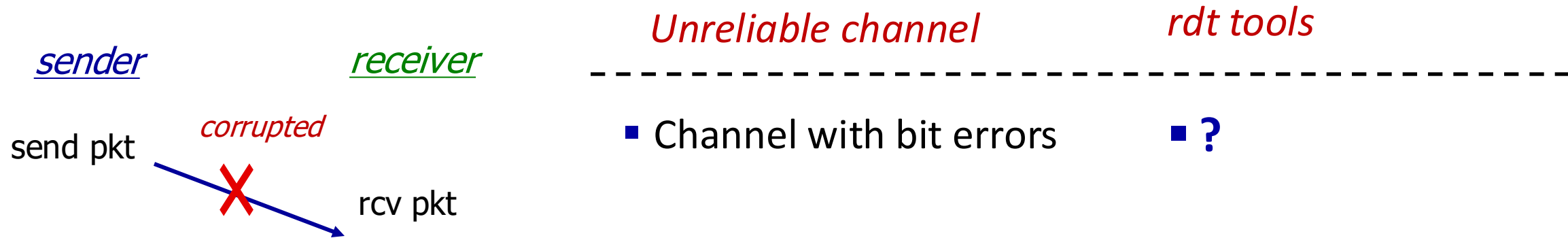
b. receiving side

State	Event	Action	Next State
Wait for call from below (data from network)	Network layer pkt arrives	Send to app layer (program)	Wait for call from below (data from network)

Unreliable channel v1: Channel with bit errors

- Remember: complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel
- For the example stop-and-wait protocol v1, we start with an underlying channel that may flip bits in packet

Simple stop-and-wait protocol (v1)



Unreliable channel v1: Channel with bit errors

- underlying channel may flip bits in packet
 - checksum (e.g., Internet checksum) to detect bit errors
- *the* question: how to recover from errors?

Q: How do humans recover from “errors” during conversation?

Unreliable channel v1: 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

Example stop-and-wait protocol (v1)

Sender

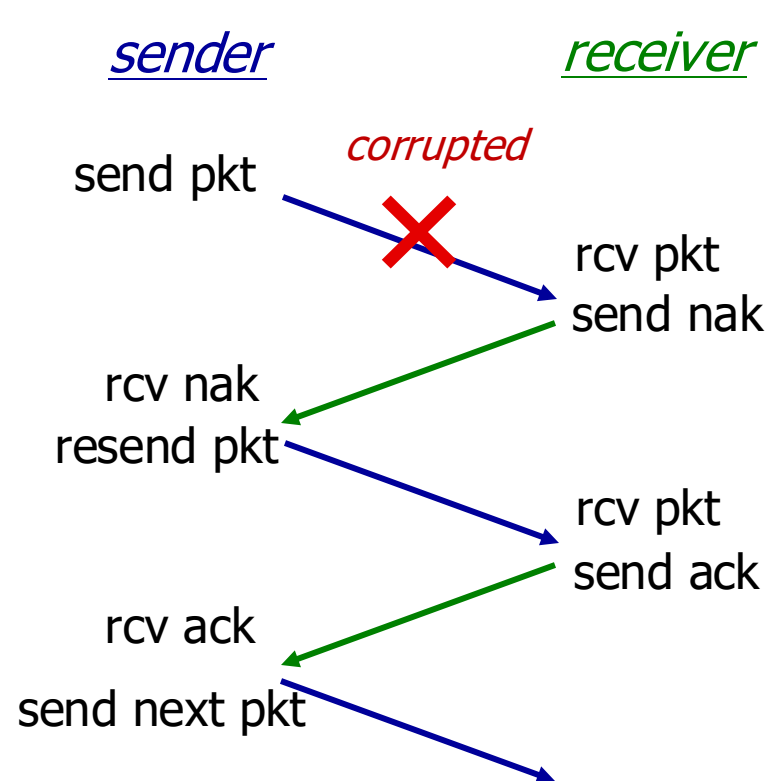
- Send a pkt
- Wait to get an ACK/NAK
 - If NAK, resend the pkt
 - go back to waiting
 - If ACK, proceed with sending next pkt

Receiver

- When pkt is received
 - examine checksum
 - If correct pkt, send ACK
 - deliver data to app layer
 - If corrupted pkt, send NAK

- Tools used: Checksum, ACK/NAK, retransmission

Example stop-and-wait protocol (v1)



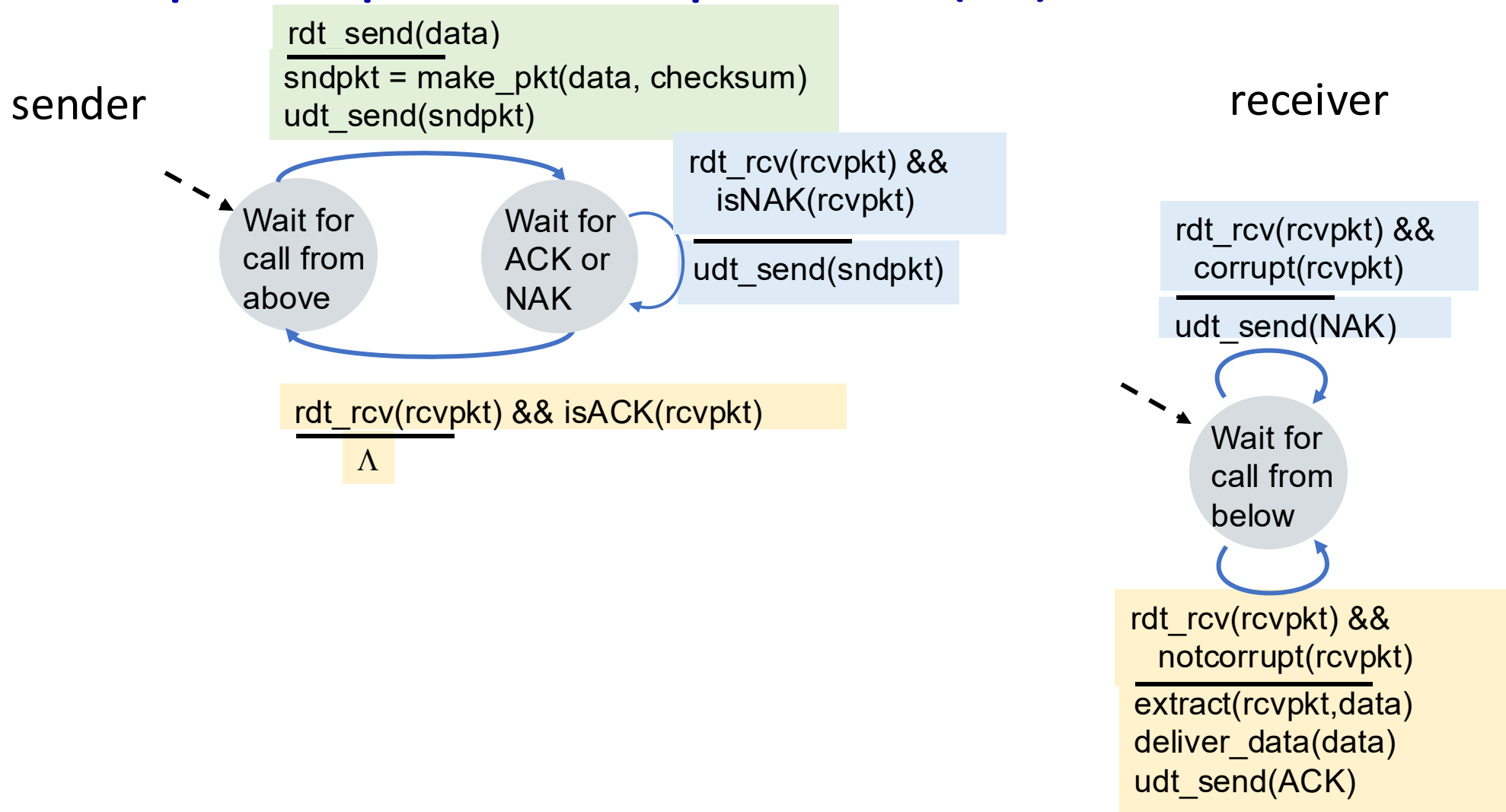
Unreliable channel

- Channel with bit errors
 - Corrupted data pkts

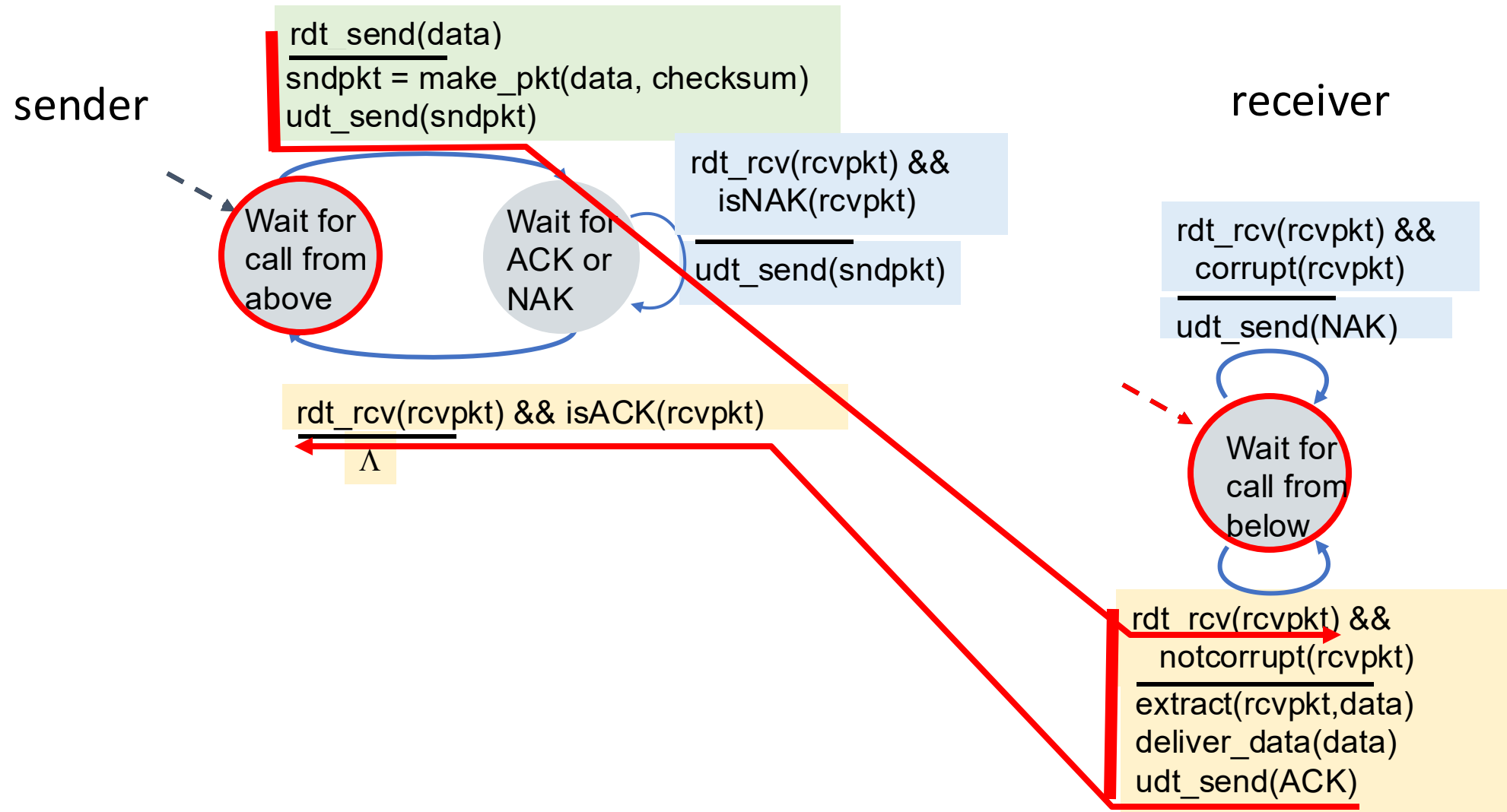
rdt tools

- Checksum, ACK/NAK, retransmission

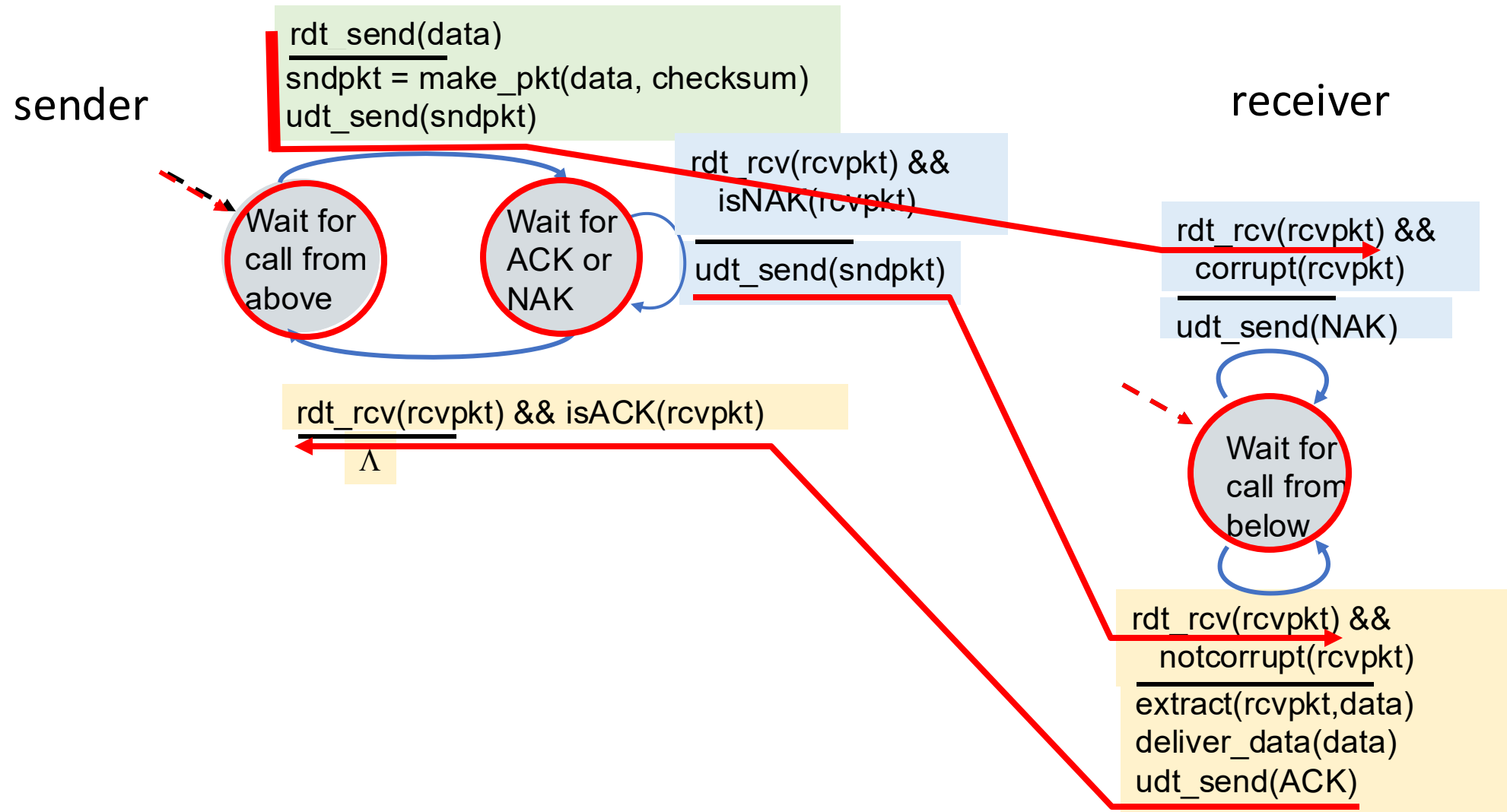
Example stop-and-wait protocol (v1) - FSM



No corruptions



Corruption scenario

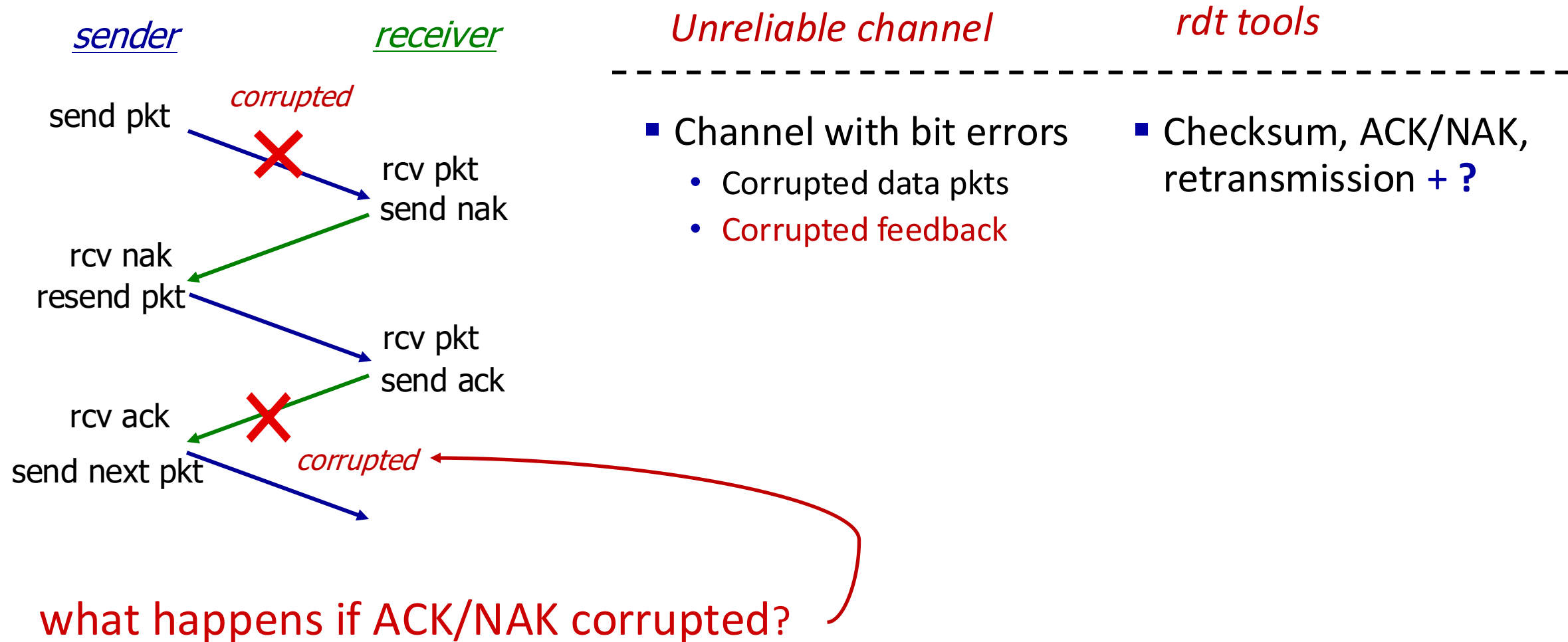


Example stop-and-wait protocol (v1) - FSM

State	Event	Action	State
Wait for call from above (data from application layer)	App-Layer says to send data	Send pkt to network layer (udt)	Wait for ACK/NAK (from network layer)
Wait for ACK/NAK	RECV NAK	Send pkt to network layer (udt)	Wait for ACK/NAK
Wait for ACK/NAK	RECV ACK	None	Wait for call from above

State	Event	Action	State
Wait for call from below (data from network layer)	pkt arrives (valid)	Send ACK, Send to app layer	Wait for call from below (data from network layer)
Wait for call from below	pkt arrives (corrupt)	Send NAK	Wait for call from below

Example stop-and-wait protocol (v1)



Corrupted feedback

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- So it should retransmit (to be on the safe side)
- But how will the rcvr distinguish between new and retransmitted data
- 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

Example stop-and-wait protocol (v2)

Sender

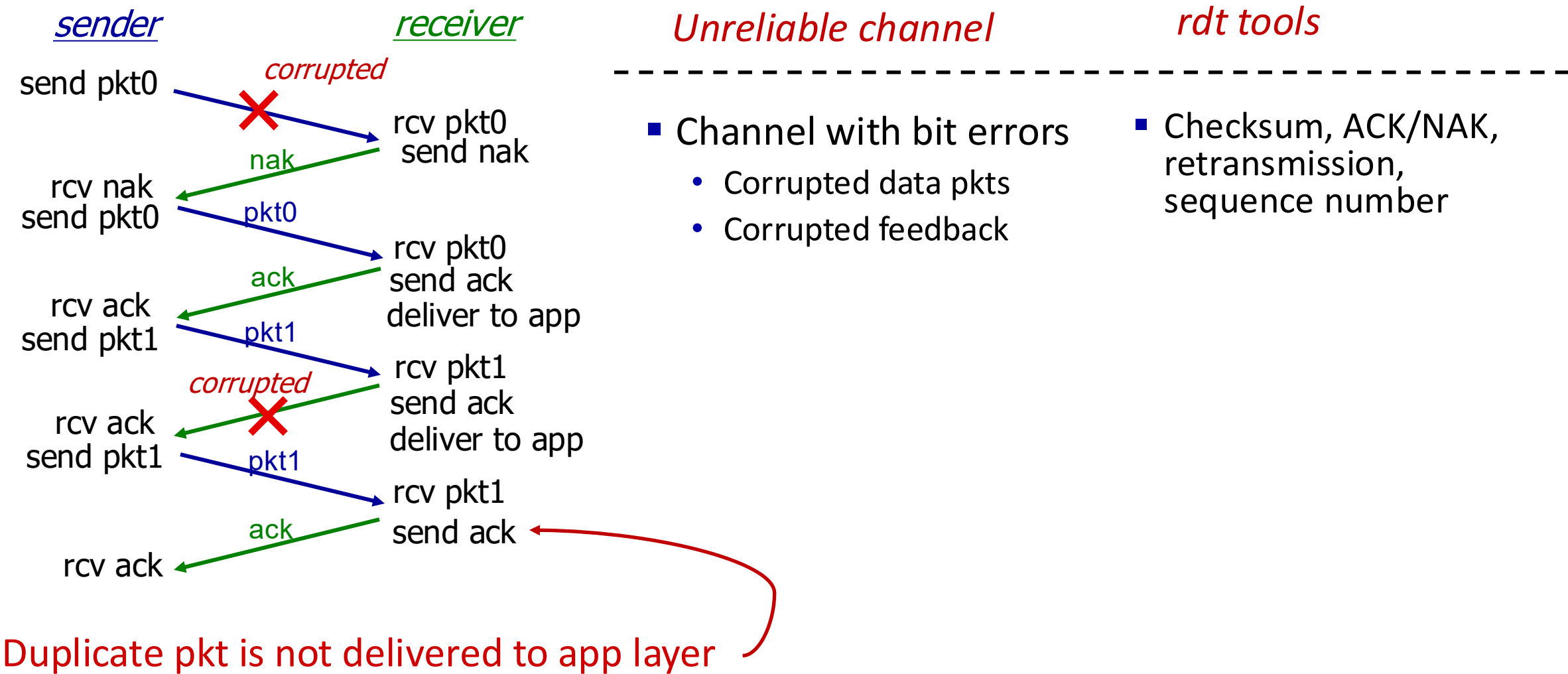
- Send a pkt
 - Seq # = 1 – last seq #
- Wait to get an ACK/NAK
 - If NAK or corrupted, resend
 - go back to waiting
 - If ACK, proceed with next pkt

Receiver

- When pkt is received
 - If correct pkt, send ACK
 - If Seq # \neq last Seq #, deliver data to app layer
 - If corrupted pkt, send NAK

- Tools used: Checksum, ACK/NAK, retransmission, 1-bit sequence number

Example stop-and-wait protocol (v2)



Example stop-and-wait protocol (v2)

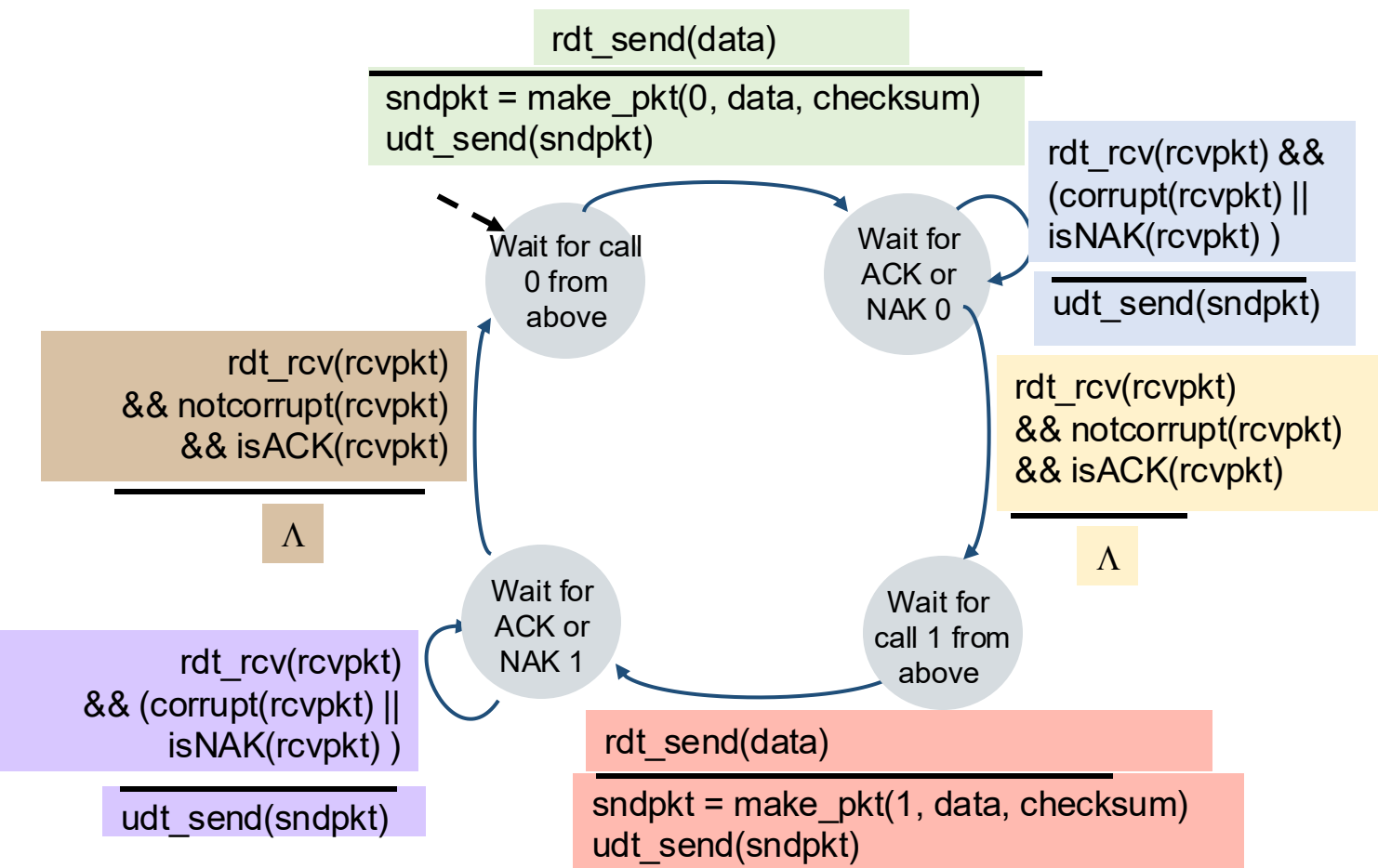
Sender:

- Seq # added to pkt
- How many sequence numbers are required?
- Two seq. #'s (0,1) will suffice.
- Must check if received ACK/NAK corrupted
- twice as many states
 - State must “remember” whether “current” pkt has 0 or 1 seq. #

Receiver:

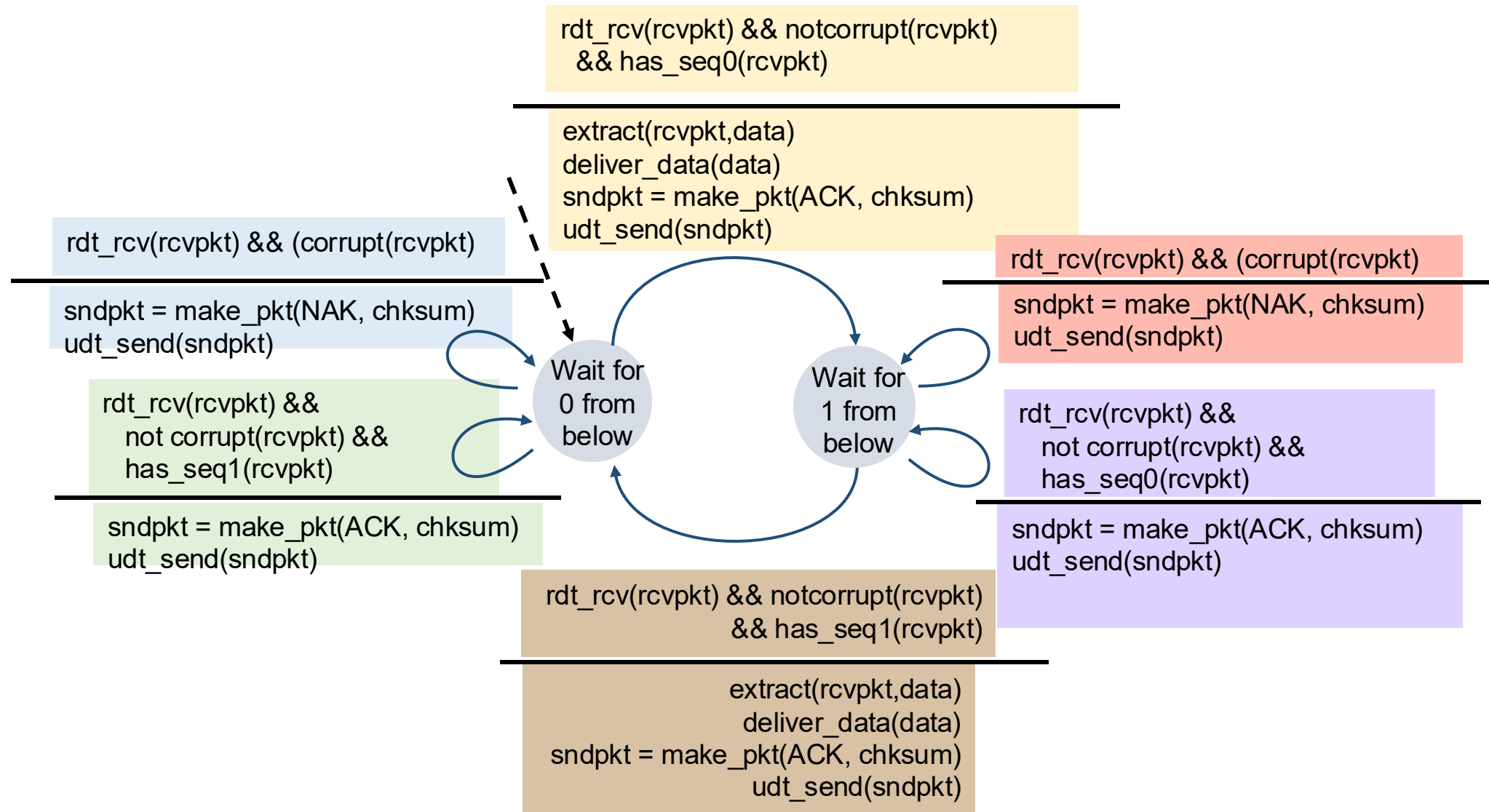
- Must check if received packet is duplicate
 - State indicates whether 0 or 1 is expected pkt seq #
- Note: receiver does not have a way of knowing if its last ACK/NAK received OK at sender

Sender FSM

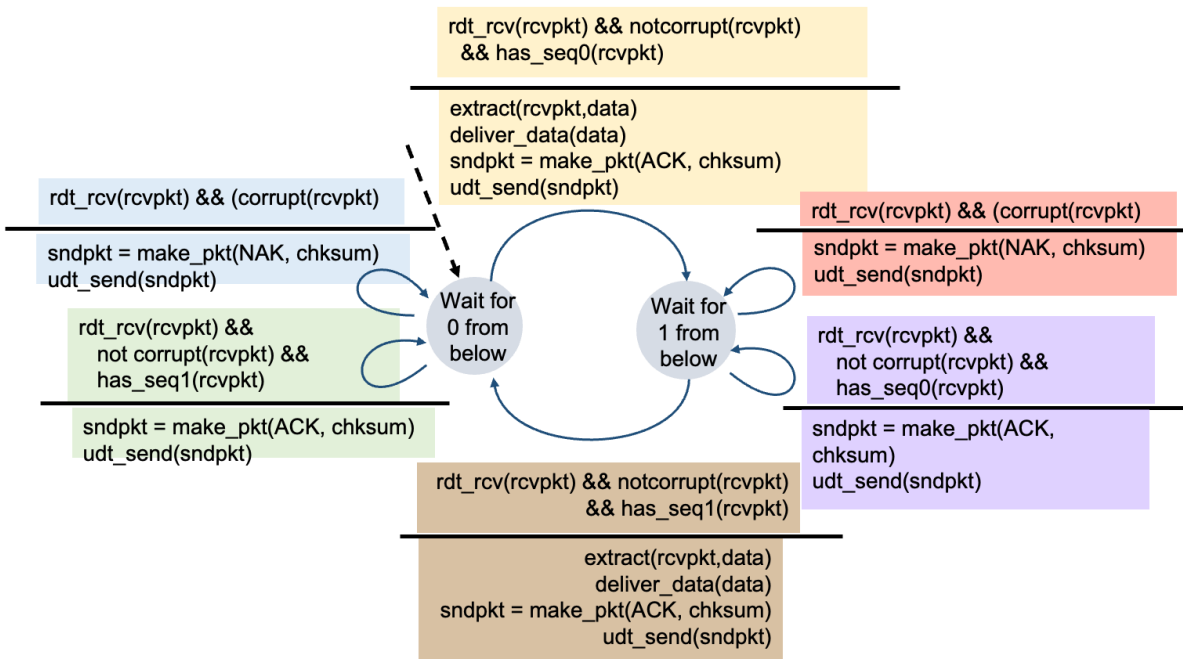


State	Event	Action	State
Wait for call from above (#0)	App-Layer says to send data	Send pkt #0 to network layer (udt)	Wait for ACK/NAK #0 (from network layer)
Wait for ACK/NAK #0	RECV NAK or RECV NAK/ACK (corrupt)	Send pkt #0 to network layer (udt)	Wait for ACK/NAK #0
Wait for ACK/NAK #0	RECV ACK (valid)	None	Wait for call from above (#1)
Wait for call from above (#1)	App-Layer says to send data	Send pkt #1 to network layer (udt)	Wait for ACK/NAK #1 (from network layer)
Wait for ACK/NAK #1	RECV NAK or RECV NAK/ACK (corrupt)	Send pkt #1 to network layer (udt)	Wait for ACK/NAK #1
Wait for ACK/NAK #1	RECV ACK (valid)	None	Wait for call from above (#0)

Receiver FSM



Receiver FSM



State	Event	Action	State
Wait for call from below (#0)	RECV pkt #0 (valid)	send ACK#0, Send to app	Wait for call from below (#1)
Wait for call from below (#0)	RECV pkt #0 (corrupt)	Send NAK#0 to udt	Wait for call from below (#0)
Wait for call from below (#0)	RECV pkt #1 (valid)	Send ACK #1 to udt	Wait for call from below (#0)
Wait for call from below (#1)	RECV pkt #1 (valid)	send ACK#1, send to app	Wait for call from below (#0)
Wait for call from below (#1)	RECV pkt #0 (valid)	Send ACK #0 to udt	Wait for call from below (#1)
Wait for call from below (#1)	RECV pkt #0 (corrupt)	Send NAK#1 to udt	Wait for call from below (#1)

Example stop-and-wait protocol (v2+): NAK-free

- Same functionality, using ACKs only
- Instead of NAK, receiver sends ACK for last pkt correctly received
 - Receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK:
retransmit current pkt

Example stop-and-wait protocol (v2+): NAK-free

Sender

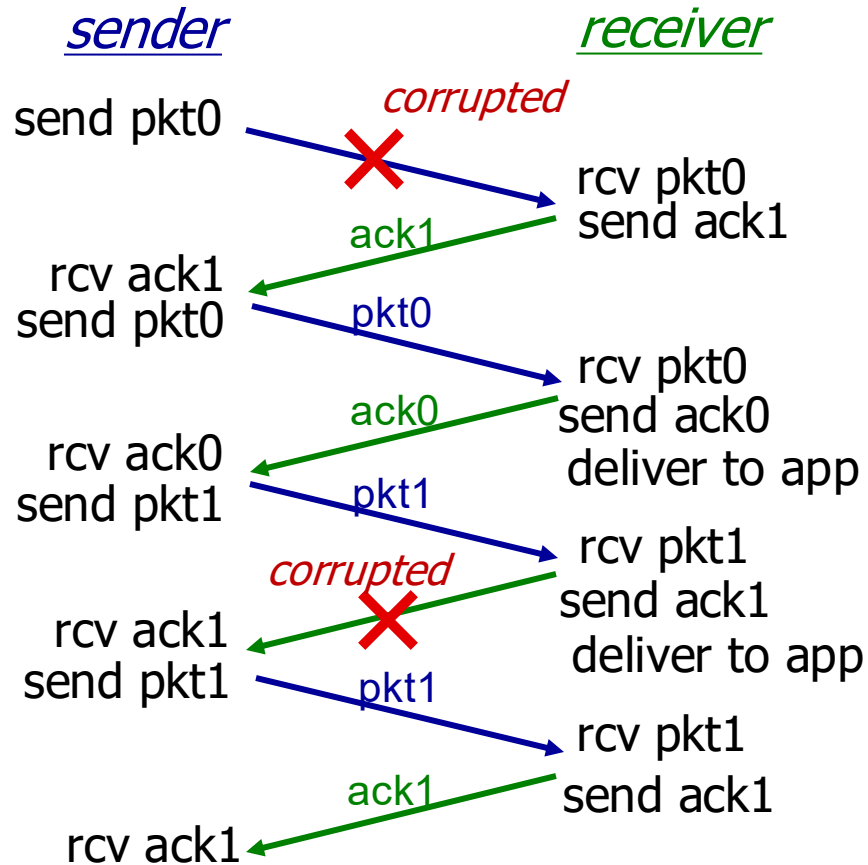
- Send a pkt
 - Seq # = 1 – last seq #
- Wait to get an ACK
 - If ACK (& last Seq #) or corrupted, resend
 - go back to waiting
 - If ACK (& Seq #), proceed with next pkt

Receiver

- When pkt is received
 - If correct pkt, send ACK (& Seq #)
 - If Seq # \neq last Seq #, deliver data to app layer
 - If corrupted pkt, send ACK (& last Seq #)

- instead of NAK, receiver sends ACK for last pkt correctly received
 - receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in the same action as NAK: retransmit current pkt

Example stop-and-wait protocol (v2+): NAK-free



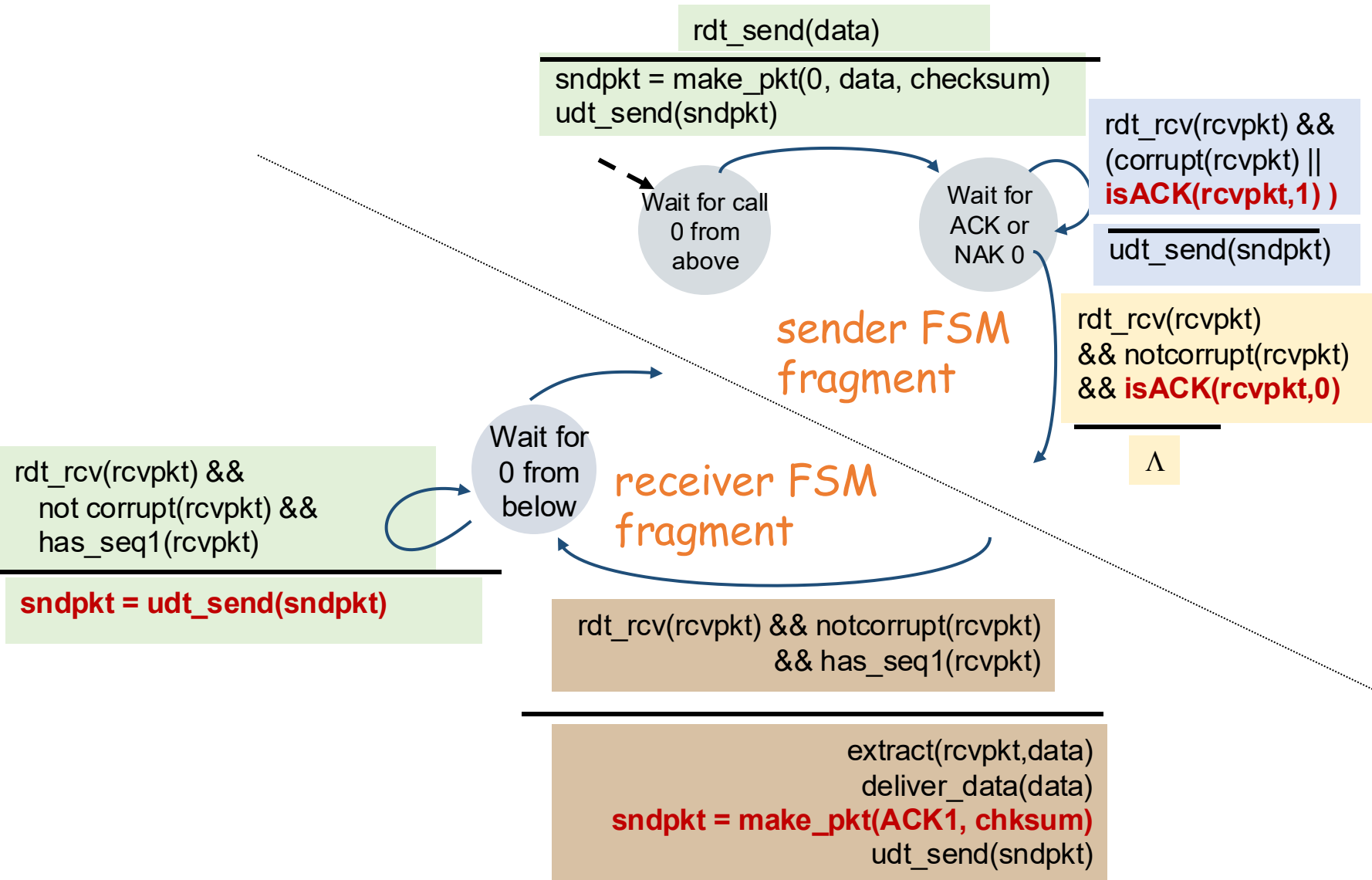
Unreliable channel

- Channel with bit errors
 - Corrupted data pkts
 - Corrupted feedback

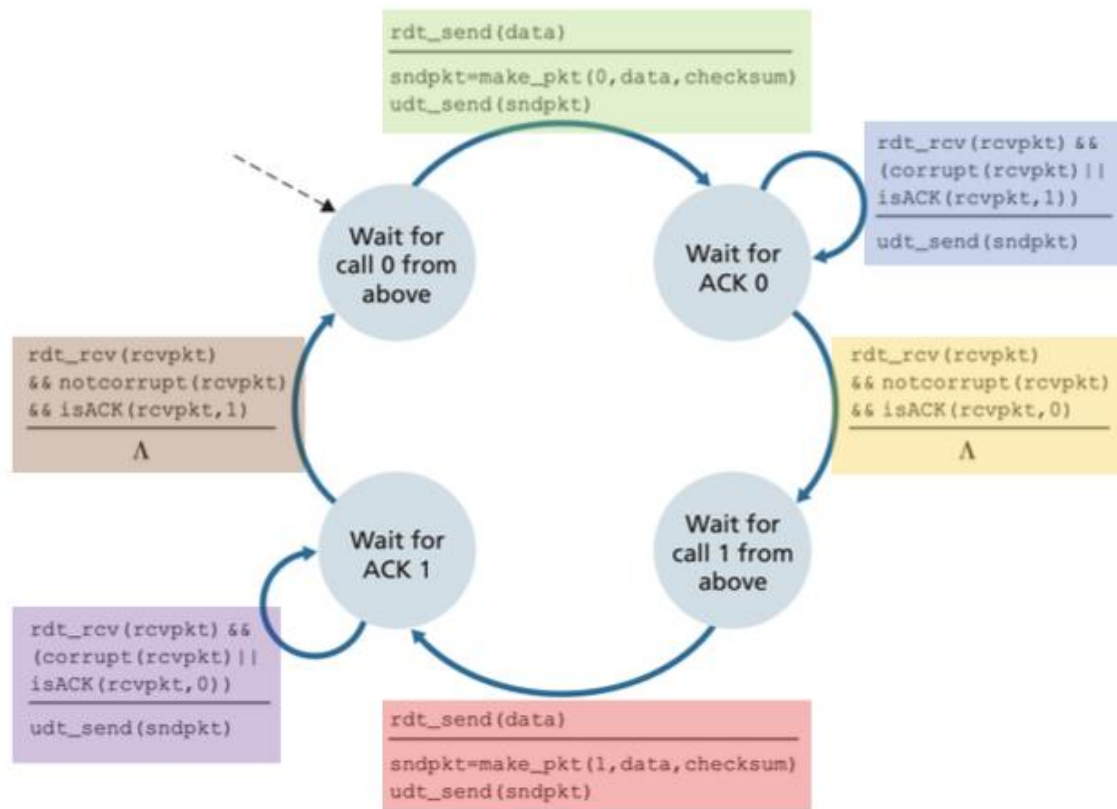
rdt tools

- Checksum, ACK, retransmission, sequence number

FSM sender, receiver fragments

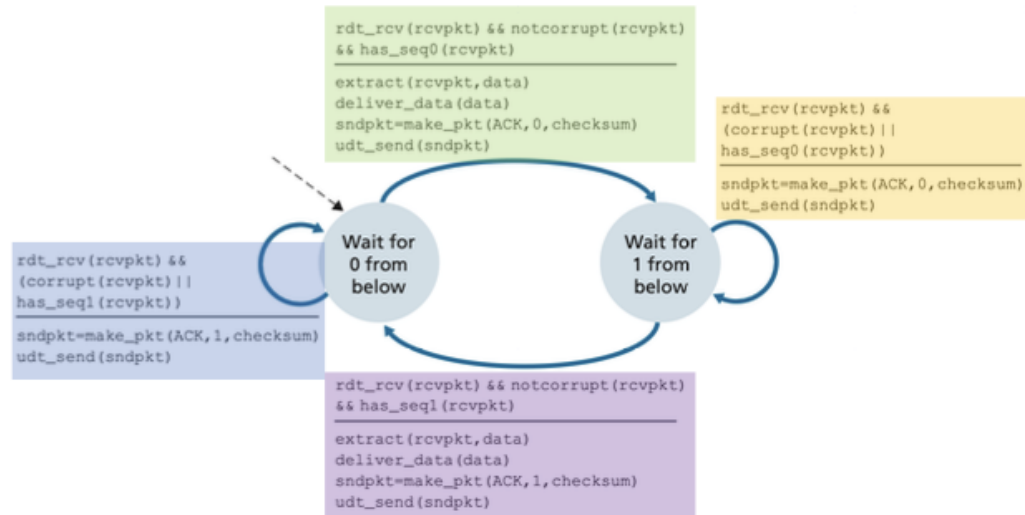


Sender FSM



State	Event	Action	State
Wait for call from above (#0)	App-Layer says to send data	Send pkt #0 to network layer (udt)	Wait for ACK #0 (from network layer)
Wait for ACK #0	RECV ACK #1 or RECV ACK #0 (corrupt)	Send pkt #0 to network layer (udt)	Wait for ACK #0
Wait for ACK #0	RECV ACK #0 (valid)	None	Wait for call from above (#1)
Wait for call from above (#1)	App-Layer says to send data	Send pkt #1 to network layer (udt)	Wait for ACK #1 (from network layer)
Wait for ACK #1	RECV ACK #0 or RECV ACK #1 (corrupt)	Send pkt #1 to network layer (udt)	Wait for ACK #1
Wait for ACK #1	RECV ACK #1 (valid)	None	Wait for call from above (#0)

Receiver FSM

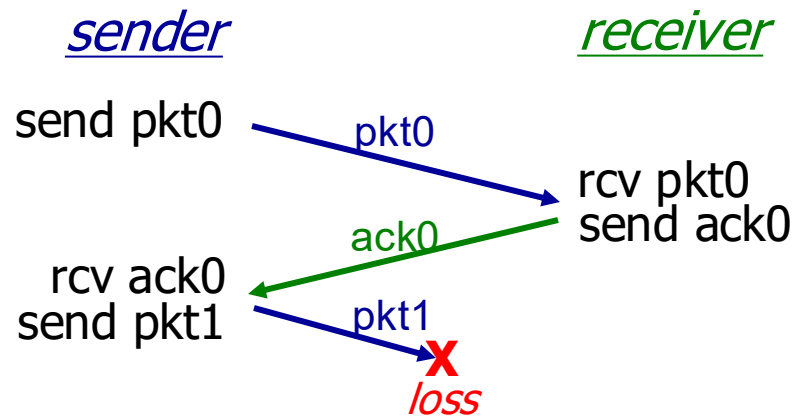


State	Event	Action	State
Wait for call from below (#0)	RECV pkt #0 (valid)	send ACK#0, Send to app	Wait for call from below (#1)
Wait for call from below (#0)	RECV pkt #0 (corrupt) or pkt #1	Send ACK#1 to udt	Wait for call from below (#0)
Wait for call from below (#1)	RECV pkt #1 (valid)	send ACK#1, send to app	Wait for call from below (#0)
Wait for call from below (#1)	RECV pkt #1 (corrupt) or pkt #0	send ACK#0 to udt	Wait for call from below (#1)

Unreliable channel v2: Channel with errors *and* loss

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

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



Q1: What is the difference between data corruption and data loss?

Q2: How do *humans* handle lost sender-to-receiver words in conversation?

Unreliable channel v2: Channel with errors *and* loss

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



timeout

Example stop-and-wait protocol (v3)

Sender

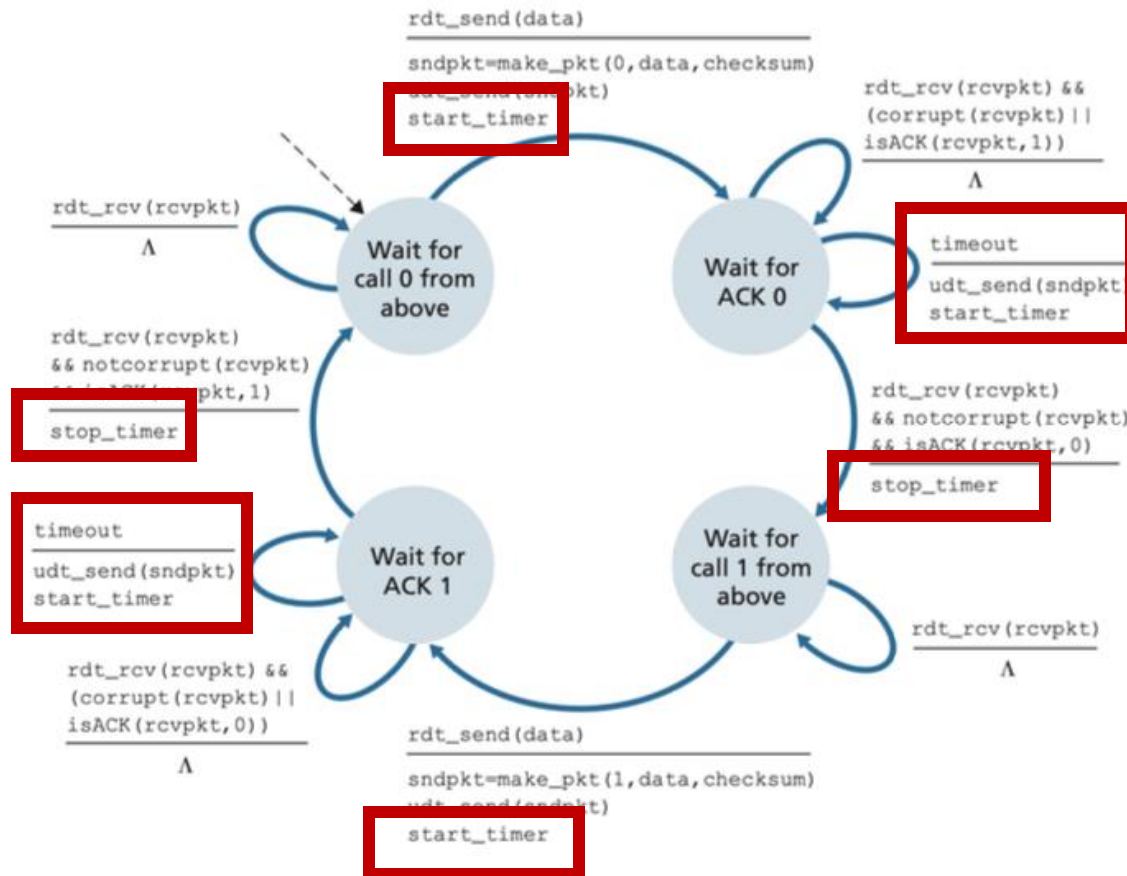
- Send a pkt
 - Seq # = 1 – last Seq #
 - Set timer
- Wait to get an ACK
 - If ACK (& last Seq #) or corrupted, resend pkt and reset timer
 - go back to waiting
 - If ACK (& Seq #), remove timer and proceed with next pkt
 - If timer goes off, resend pkt and reset timer

Receiver

- When pkt is received
 - If correct pkt, send ACK (& Seq #)
 - If Seq # \neq last Seq #, deliver data to app layer
 - If corrupted pkt, send ACK (& last Seq #)

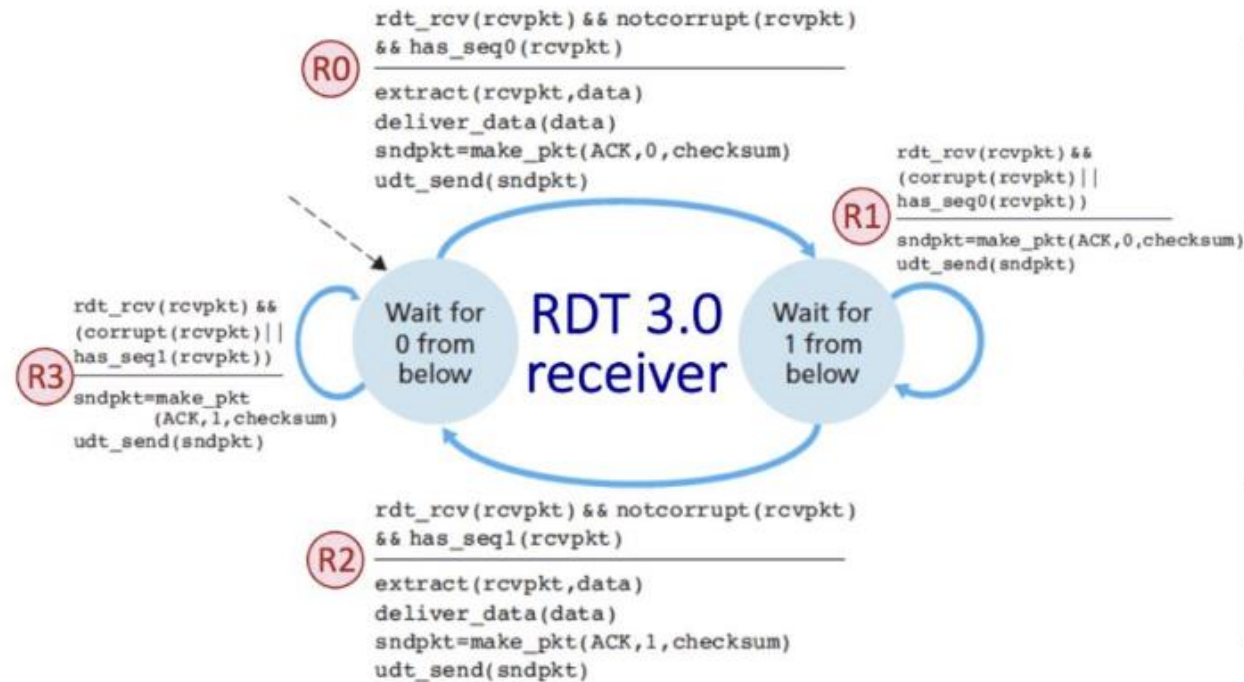
- Tools used: Checksum, ACK, retransmission, 1-bit sequence number, timer

Sender FSM



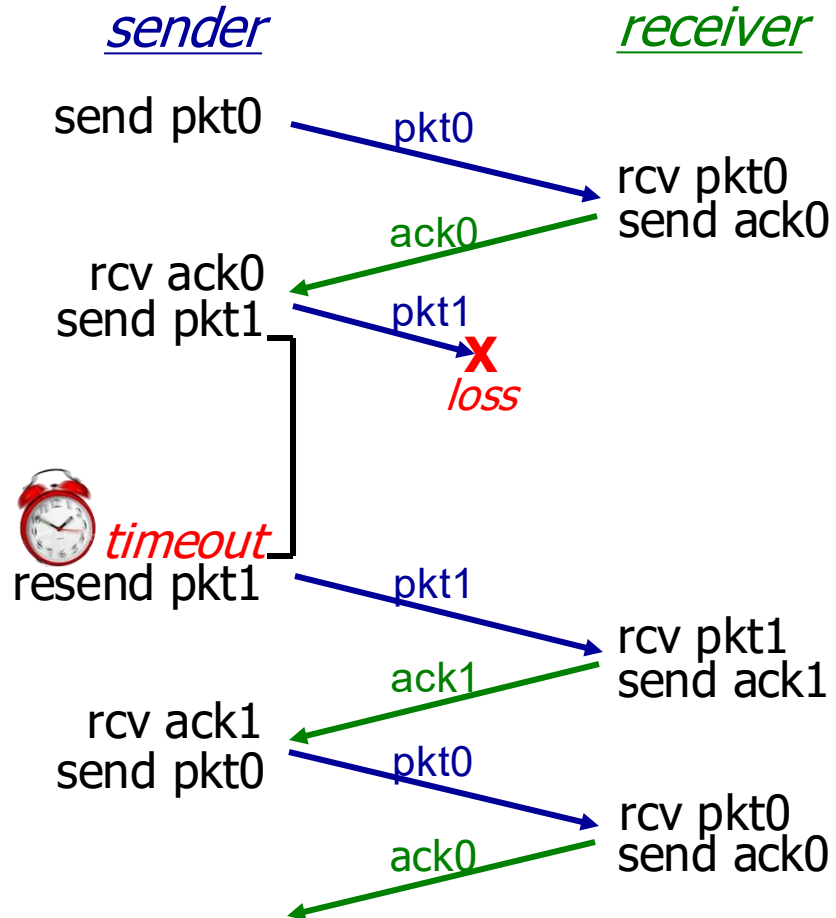
State	Event	Action	State
Wait for call from above (#0)	app says to send data	Send pkt #0 to network layer, start timer	Wait for ACK #0
Wait for call from above (#0)	RECV ACK	None	Wait for call from above (#0)
Wait for ACK #0	RECV ACK #1 or ACK #0 (corrupt)	None	Wait for ACK #0
Wait for ACK #0	timeout	Send pkt #0 to network layer	Wait for ACK #0
Wait for ACK #0	RECV ACK #0 (valid)	stop timer	Wait for call from above (#1)
Wait for call from above (#1)	app says to send data	Send pkt #1 to network layer, start timer	Wait for call from above (#1)
Wait for call from above (#1)	RECV ACK	None	Wait for call from above (#1)
Wait for ACK #1	RECV ACK #1 (valid)	Stop timer	Wait for call from above (#0)
Wait for ACK #1	RECV ACK #0 or ACK #1 (corrupt)	None	Wait for ACK #1
Wait for ACK #1	timeout	Send pkt #1 to network layer	Wait for ACK #1

Receiver FSM



State	Event	Action	State
Wait for call from below (#0)	RECV pkt #0 (valid)	Send ACK #0, send to app	Wait for call from below (#1)
Wait for call from below (#0)	RECV pkt #0 (corrupt) or pkt #1	Send ACK #1	Wait for call from below (#0)
Wait for call from below (#1)	RECV pkt #1 (valid)	Send ACK #1, send to app	Wait for call from below (#0)
Wait for call from below (#1)	RECV pkt #1 (corrupt) or pkt #0	Send ACK #0	Wait for call from below (#1)

Example stop-and-wait protocol (v3)



(a) packet loss

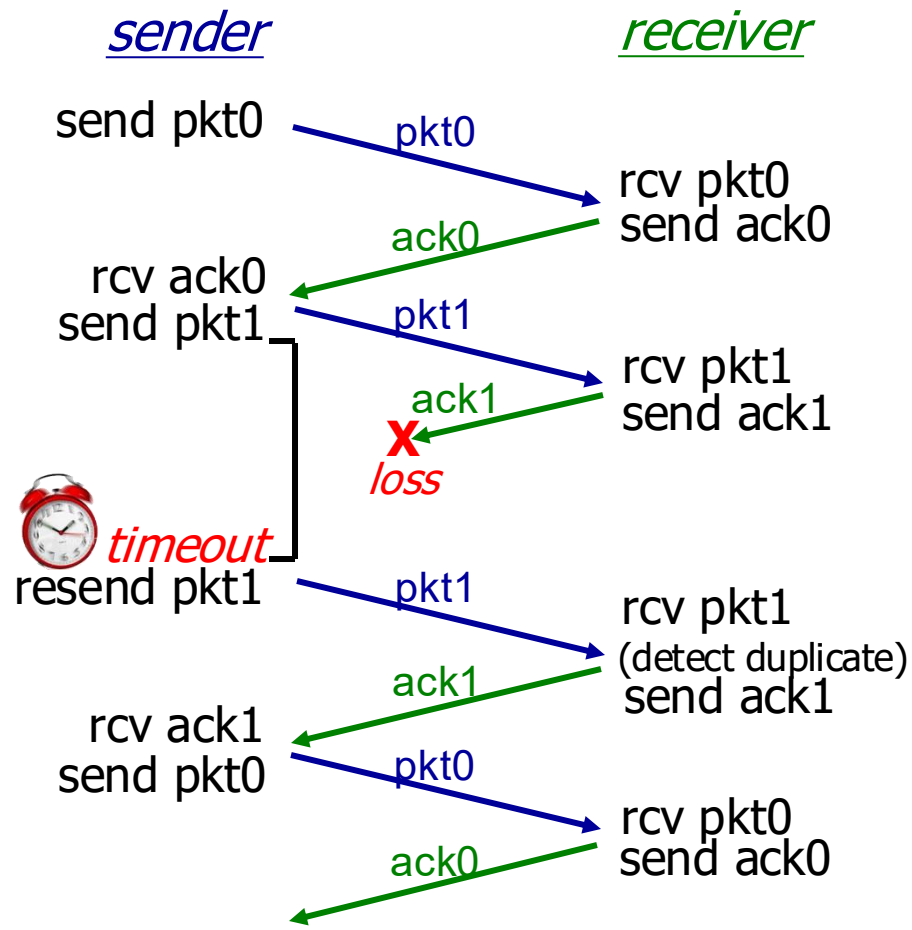
Unreliable channel

- Channel with bit errors
 - Corrupted data pkts
 - Corrupted feedback

rdt tools

- Channel with errors and lost
 - lost data pkts
- Checksum, ACK, retransmission, sequence number
- Checksum, ACK, retransmission, sequence number, timer

Example stop-and-wait protocol (v3)



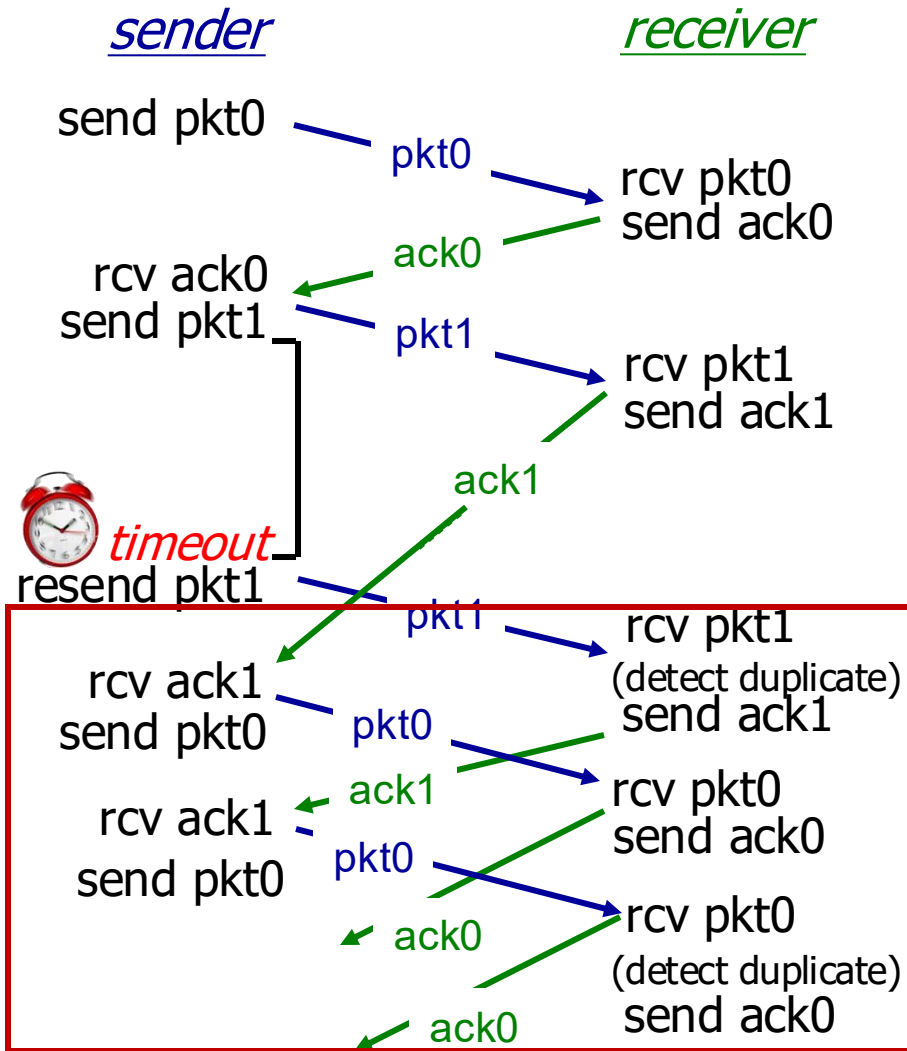
Unreliable channel

- Channel with bit errors
 - Corrupted data pkts
 - Corrupted feedback

rdt tools

- Channel with errors and lost
 - lost data pkts
 - lost feedback
- Checksum, ACK, retransmission, sequence number
- Checksum, ACK, retransmission, sequence number, timer

Example stop-and-wait protocol (v3)



(c) premature timeout/ delayed ACK

Unreliable channel

- Channel with bit errors
 - Corrupted data pkts
 - Corrupted feedback

rdt tools

- Checksum, ACK, retransmission, sequence number
-
- Checksum, ACK, retransmission, sequence number, timer

Duplicate pkt continues

Example stop-and-wait protocol (v3)

Sender

- Send a pkt
 - Seq # = 1 – last Seq #
 - Set timer
- Wait to get an ACK
 - If ACK (& last Seq #) or corrupted,
 - do nothing
 - If ACK (& Seq #), remove timer and proceed with next pkt
 - If timer goes off, resent pkt and reset timer

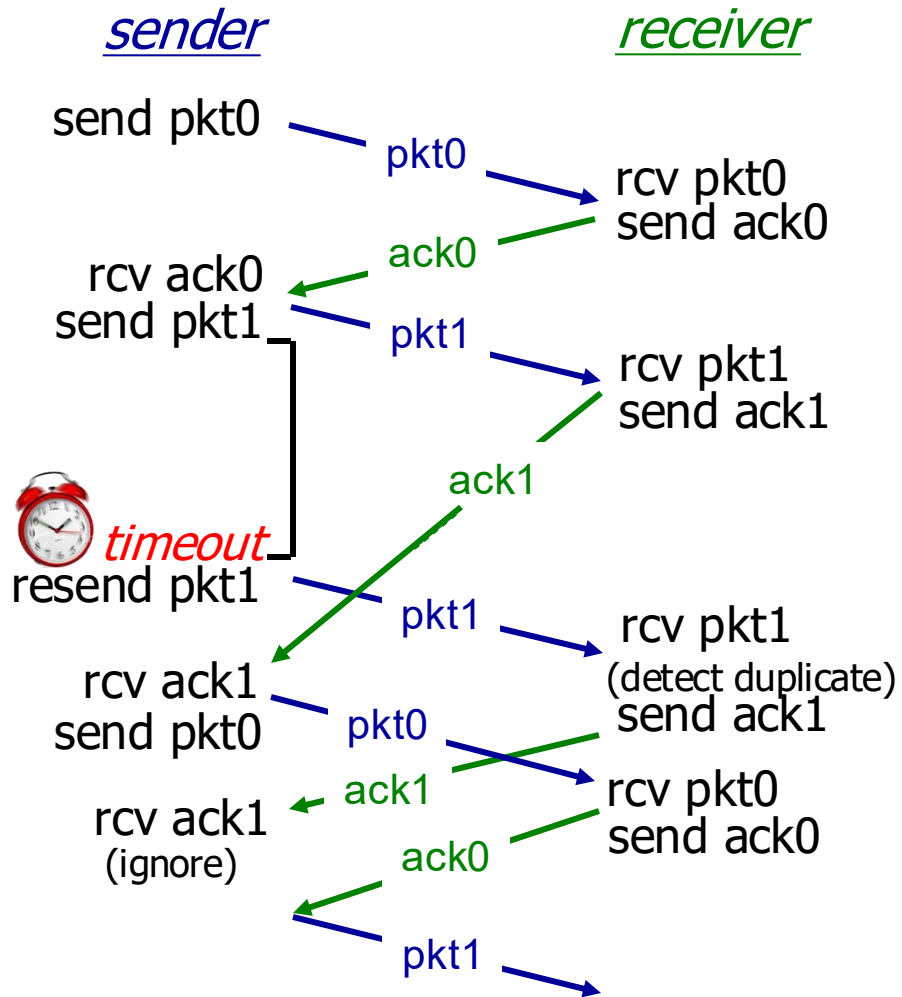
Receiver

- When pkt is received
 - If correct pkt, send ACK (& Seq #)
 - If Seq # \neq last Seq #, deliver data to app layer
 - If corrupted pkt, send (& last Seq #)

Timer can handle all retransmissions

- Tools used: Checksum, ACK, retransmission, 1-bit sequence number, timer

Example stop-and-wait protocol (v3)



(c) premature timeout/ delayed ACK

Unreliable channel

- Channel with bit errors
 - Corrupted data pkts
 - Corrupted feedback

rdt tools

- Channel with errors and lost
 - lost data pkts
 - lost feedback
- Checksum, ACK, retransmission, sequence number, timer

Principles of reliable data transfer (rdt)

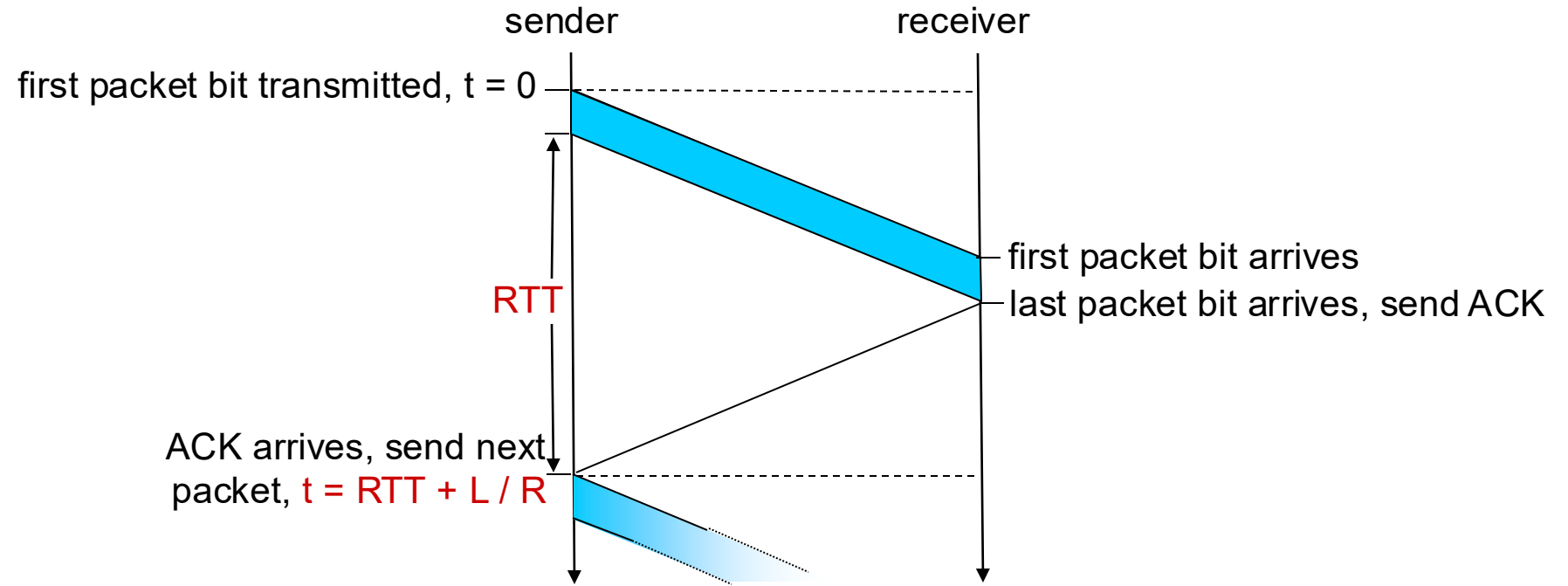
- rdt at a glance
- Stop-and-wait approach
 - sender sends one pkt, then waits for receiver's response
- Pipelined approach
 - Go-back-N (GBN)
 - Selective Repeat (SR)

Stop-and-wait protocol has a problem

- 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:

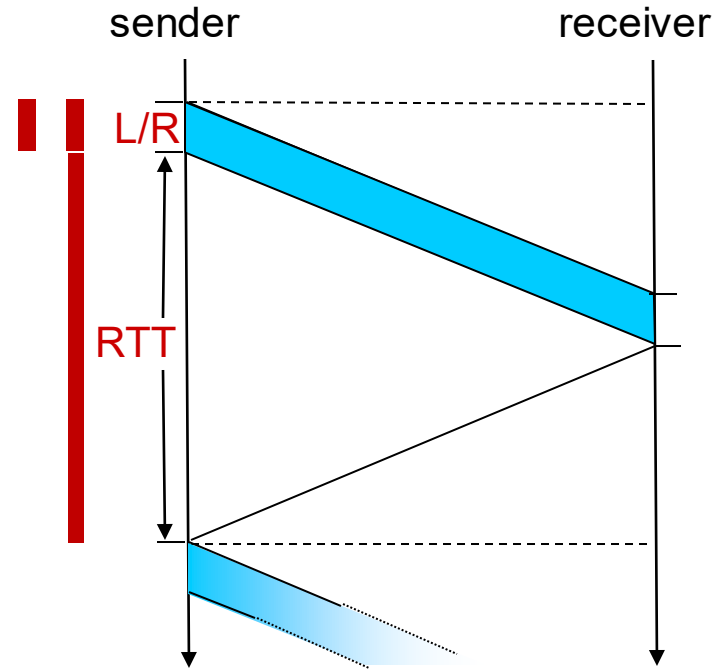
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

Stop-and-wait protocol has a problem



Stop-and-wait protocol has a problem

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

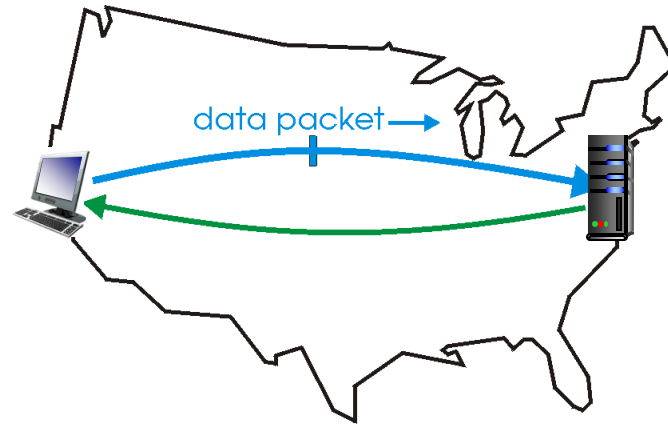


- Protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

Pipelined protocols operation

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

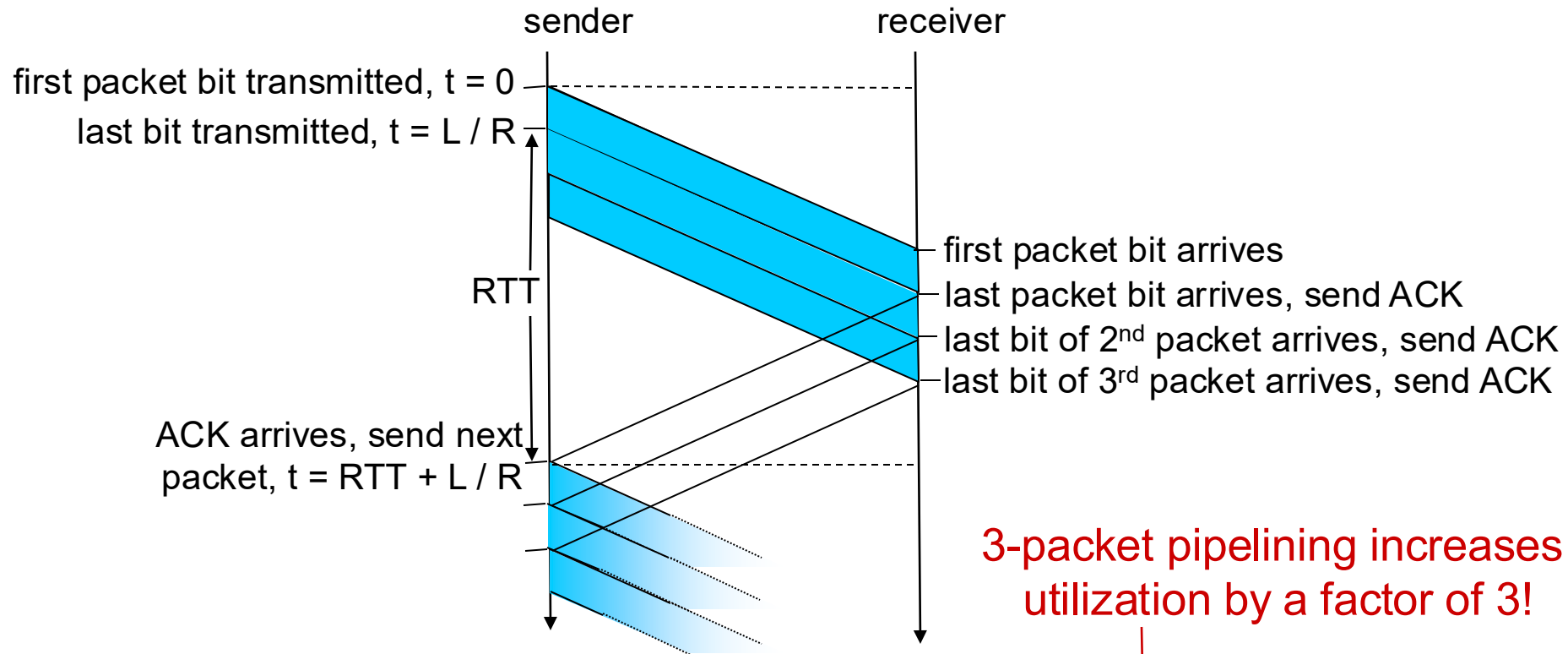
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

- two example forms of the pipelined approach: *go-Back-N, selective repeat*

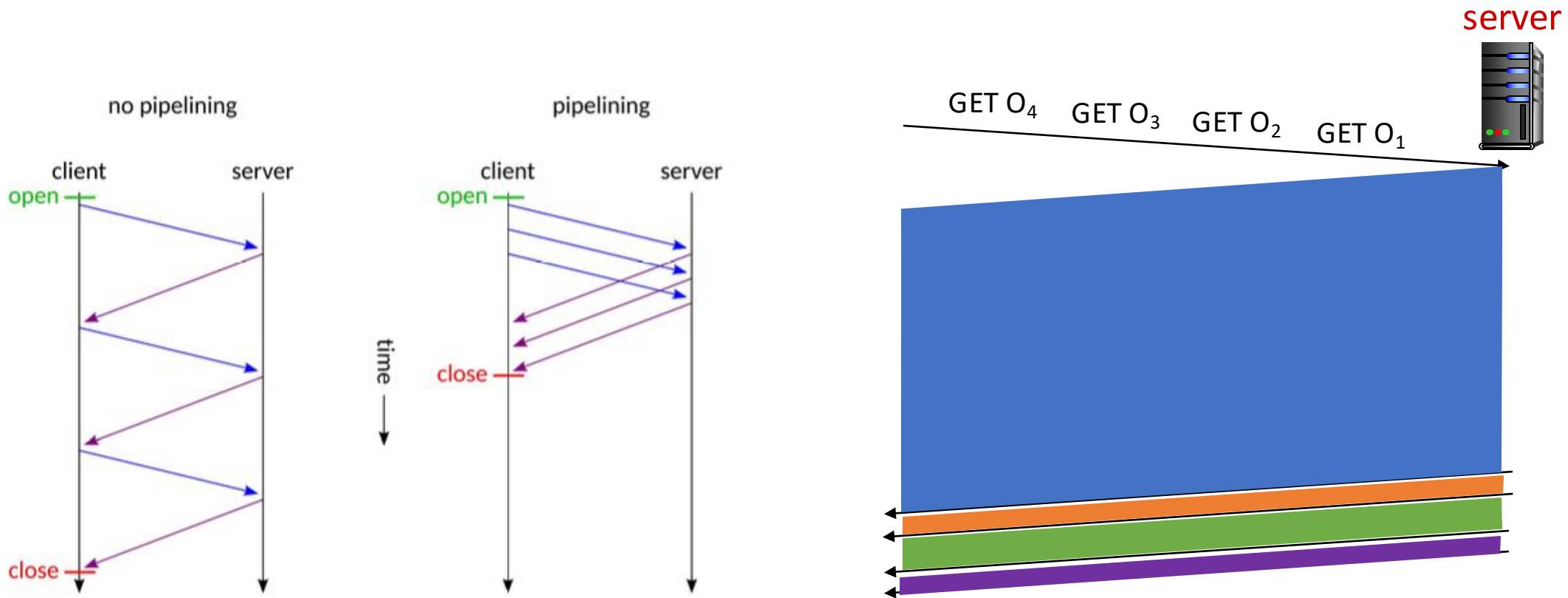
Pipelining: increased utilization



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

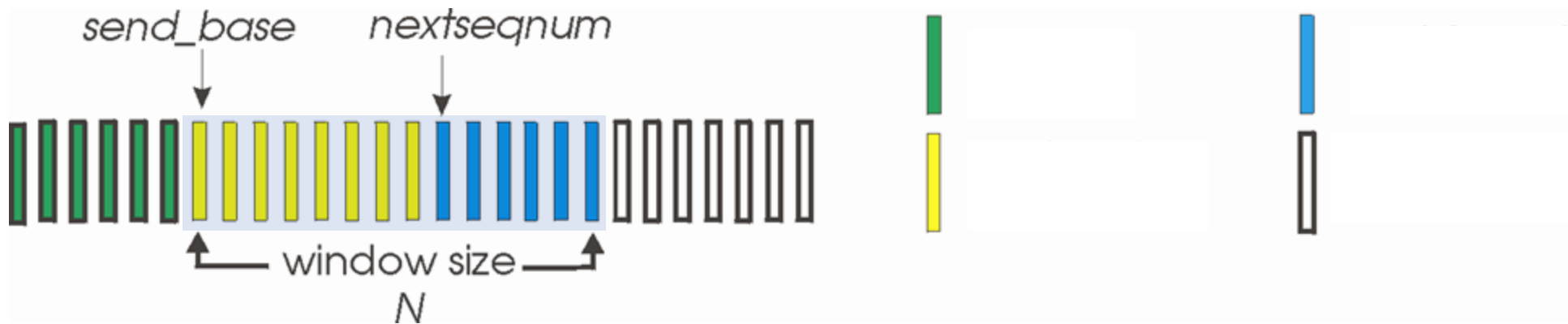
$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Recall: HTTP 1.0 (stop-and-wait) vs. HTTP 1.1 (Pipelining)



Go-Back-N: sender

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

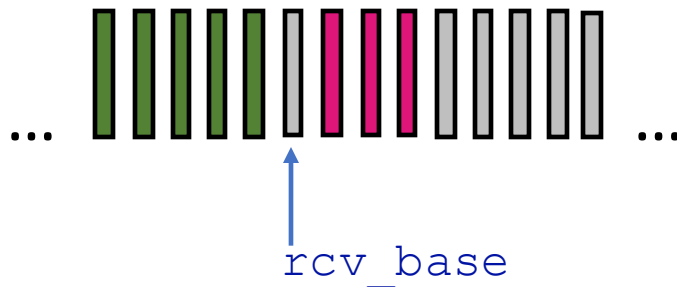


- ***cumulative ACK***: $ACK(n)$: ACKs all packets up to, including seq # n
 - on receiving $ACK(n)$: move window forward to begin at $n+1$
- Single timer for oldest in-flight packet
- *timeout(n)*: retransmit packet n and all higher seq # packets in window

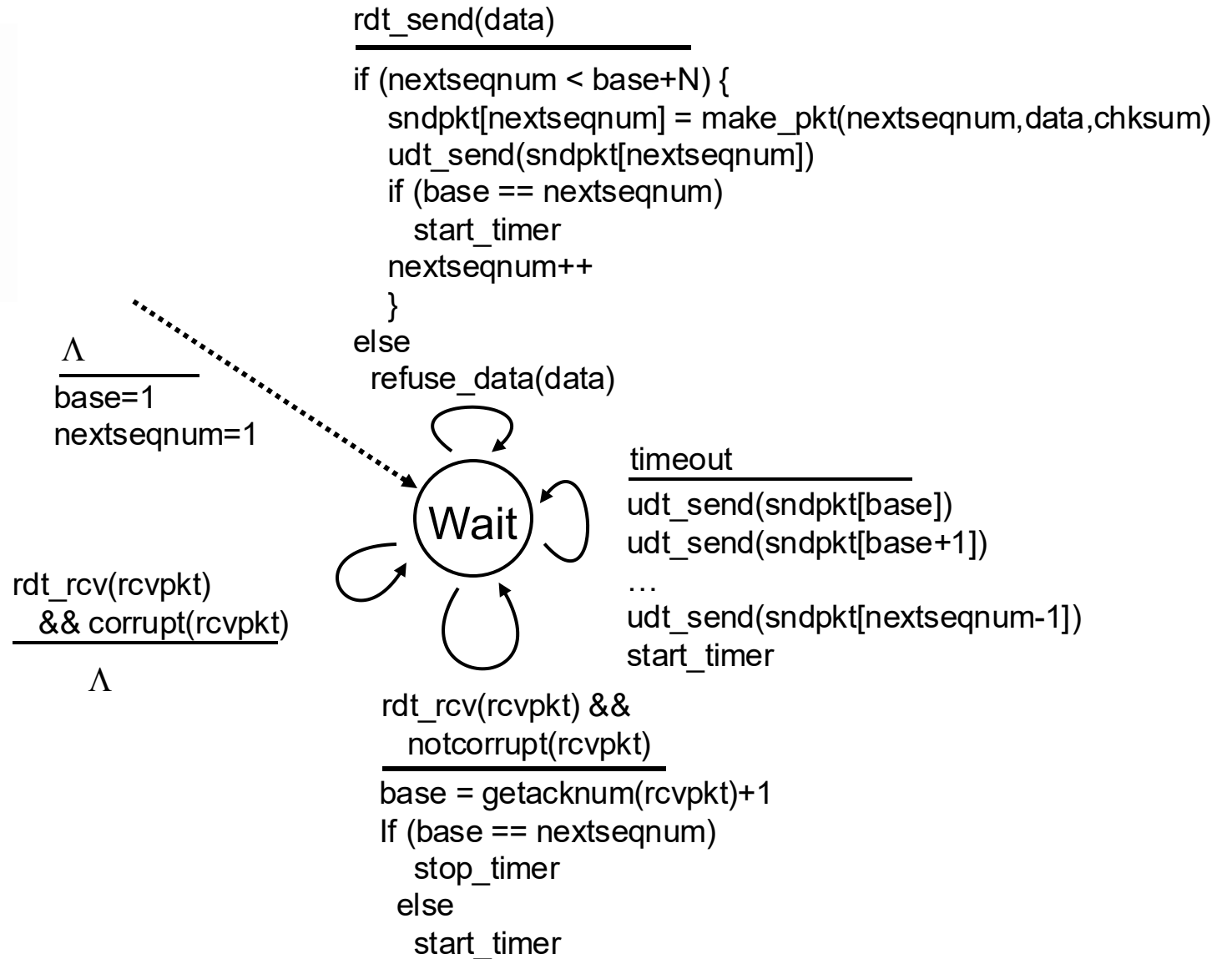
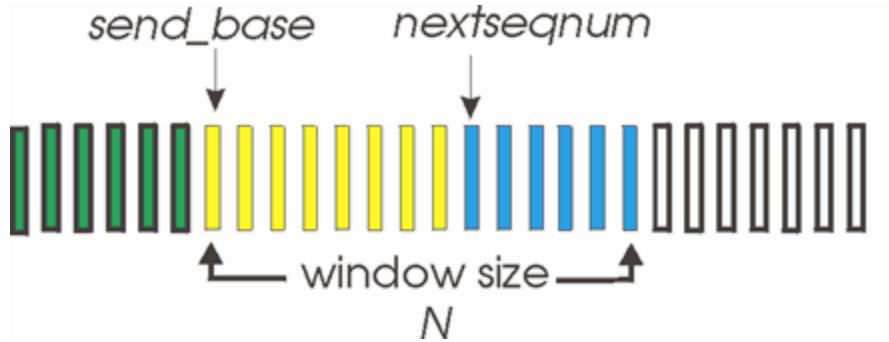
Go-Back-N: receiver

- 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 #

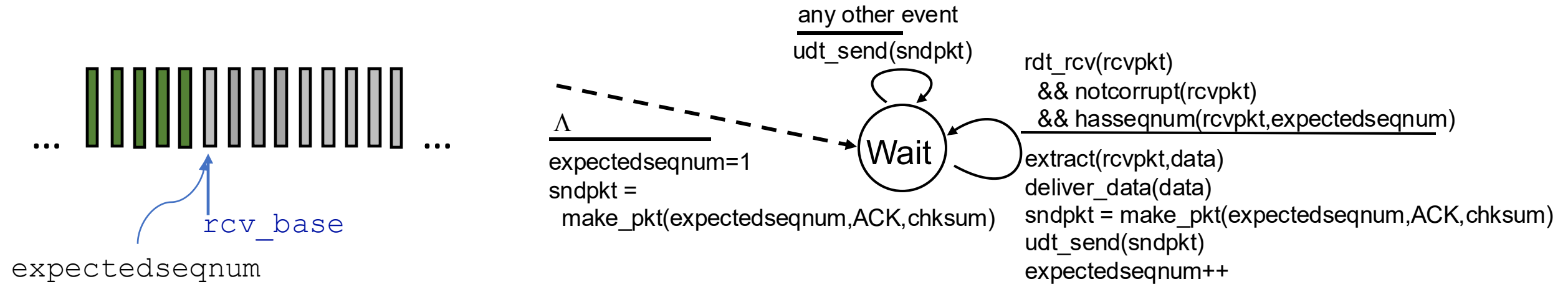
Receiver view of sequence number space:



Go-Back-N: sender extended FSM



Go-Back-N: receiver extended FSM



ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #

- may generate duplicate ACKs
 - need only remember **expectedseqnum**
- out-of-order packet:
- discard (don't buffer): *no receiver buffering!*
 - re-ACK pkt with highest in-order seq #

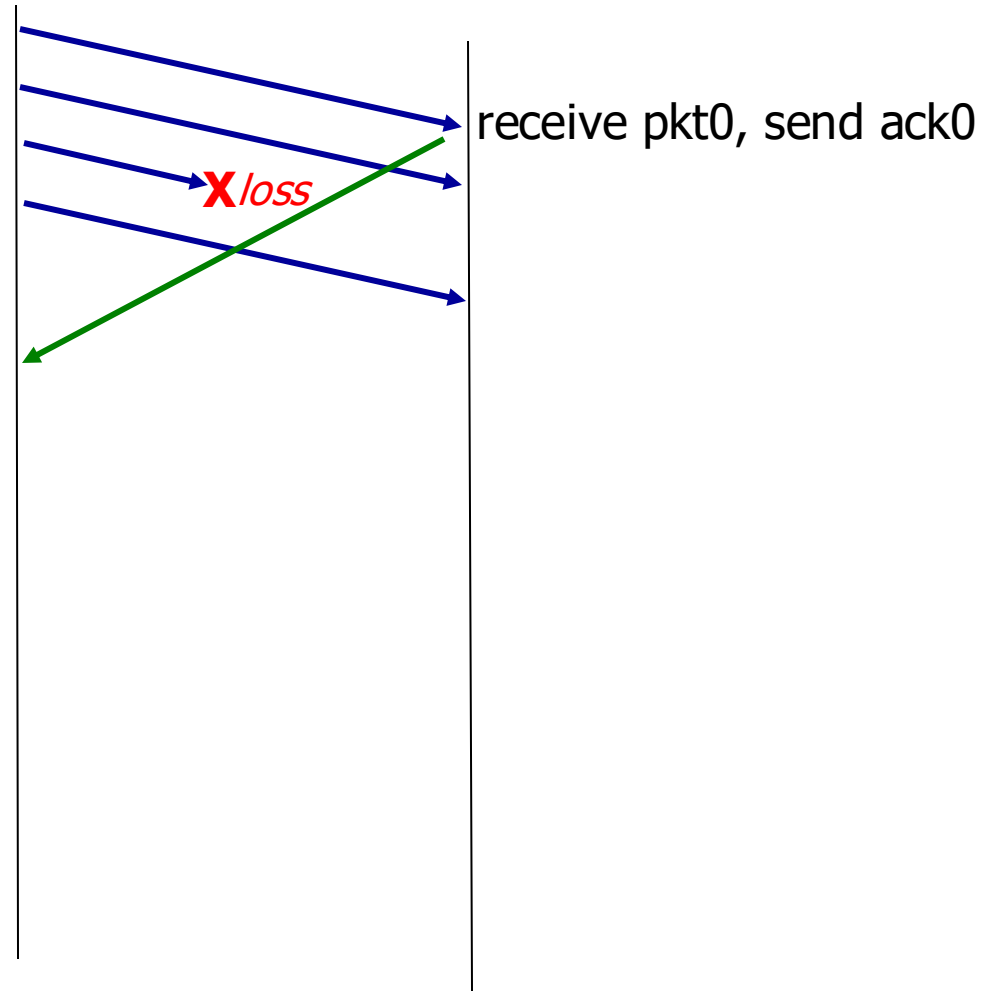
Go-Back-N in action

sender window (N=4)

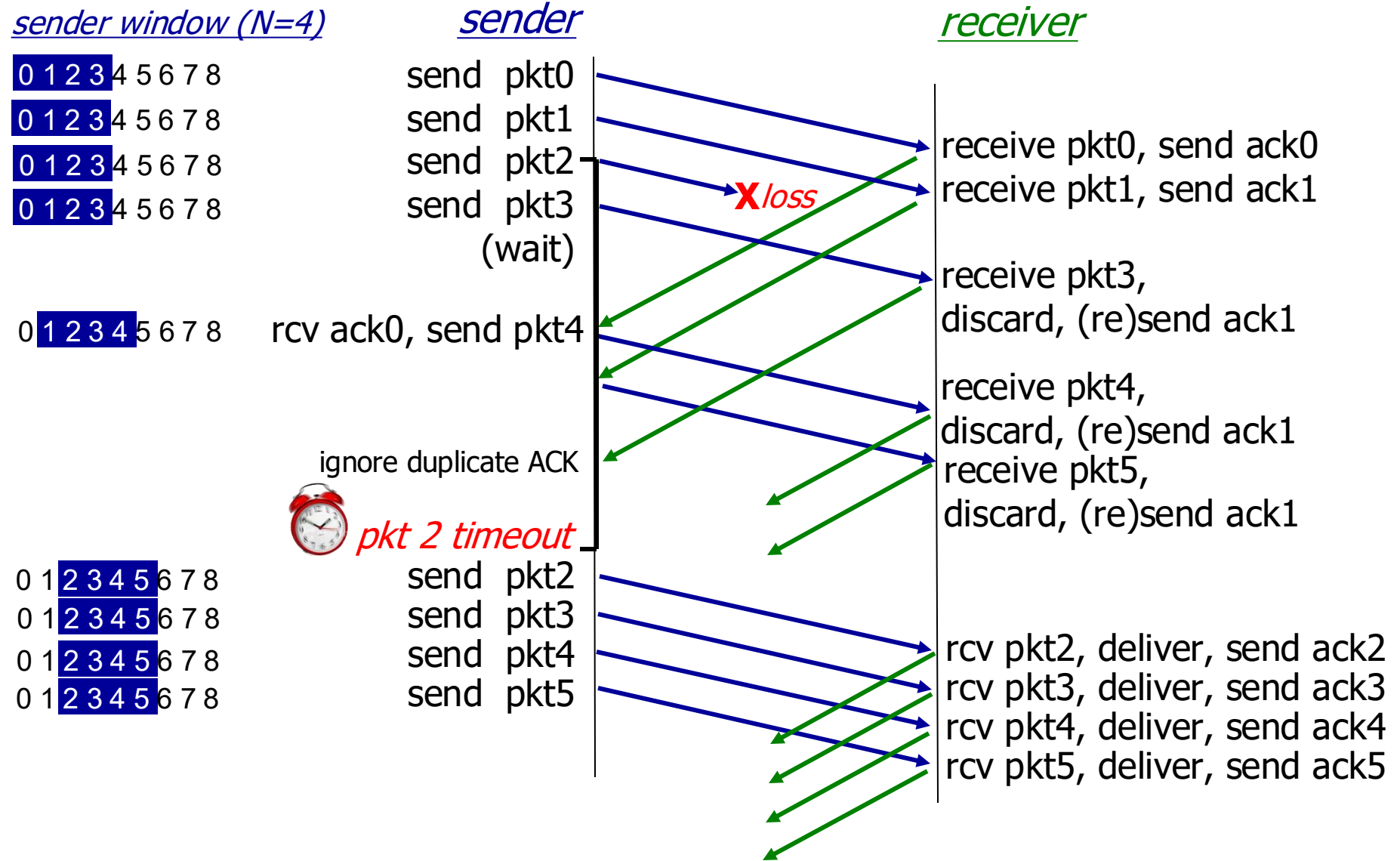
0 1 2 3 4 5 6 7 8

sender

receiver



Go-Back-N in action



Go-Back-N in action

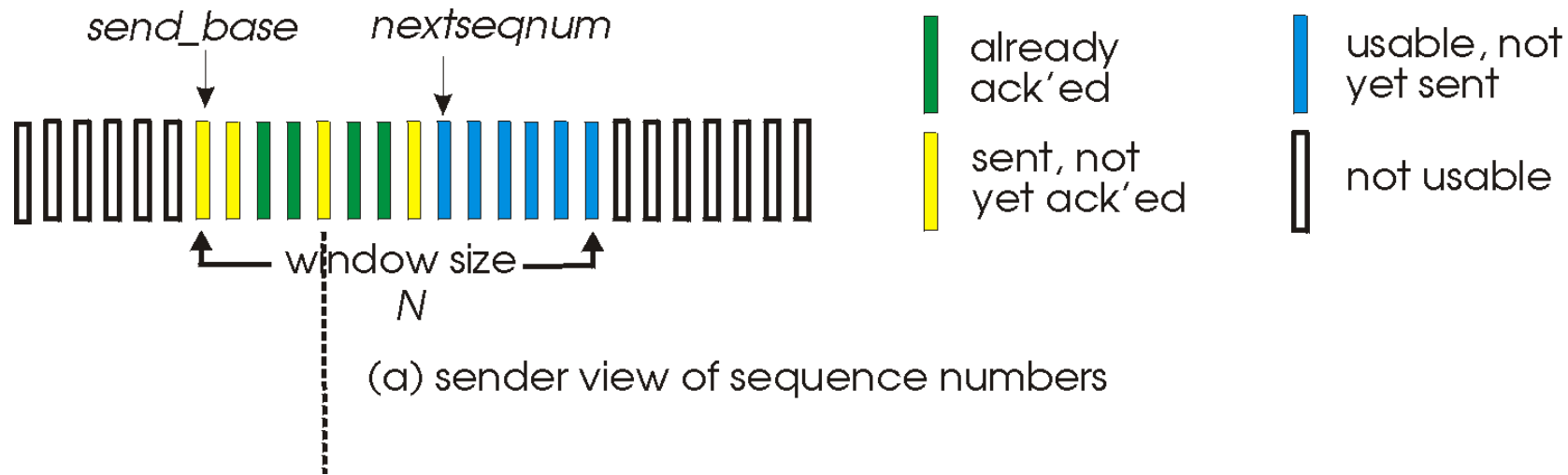
- Animation here:

https://media.pearsoncmg.com/ph/esm/ecs_kurose_com_pnetwork_8/cw/content/interactiveanimations/go-back-n-protocol/index.html

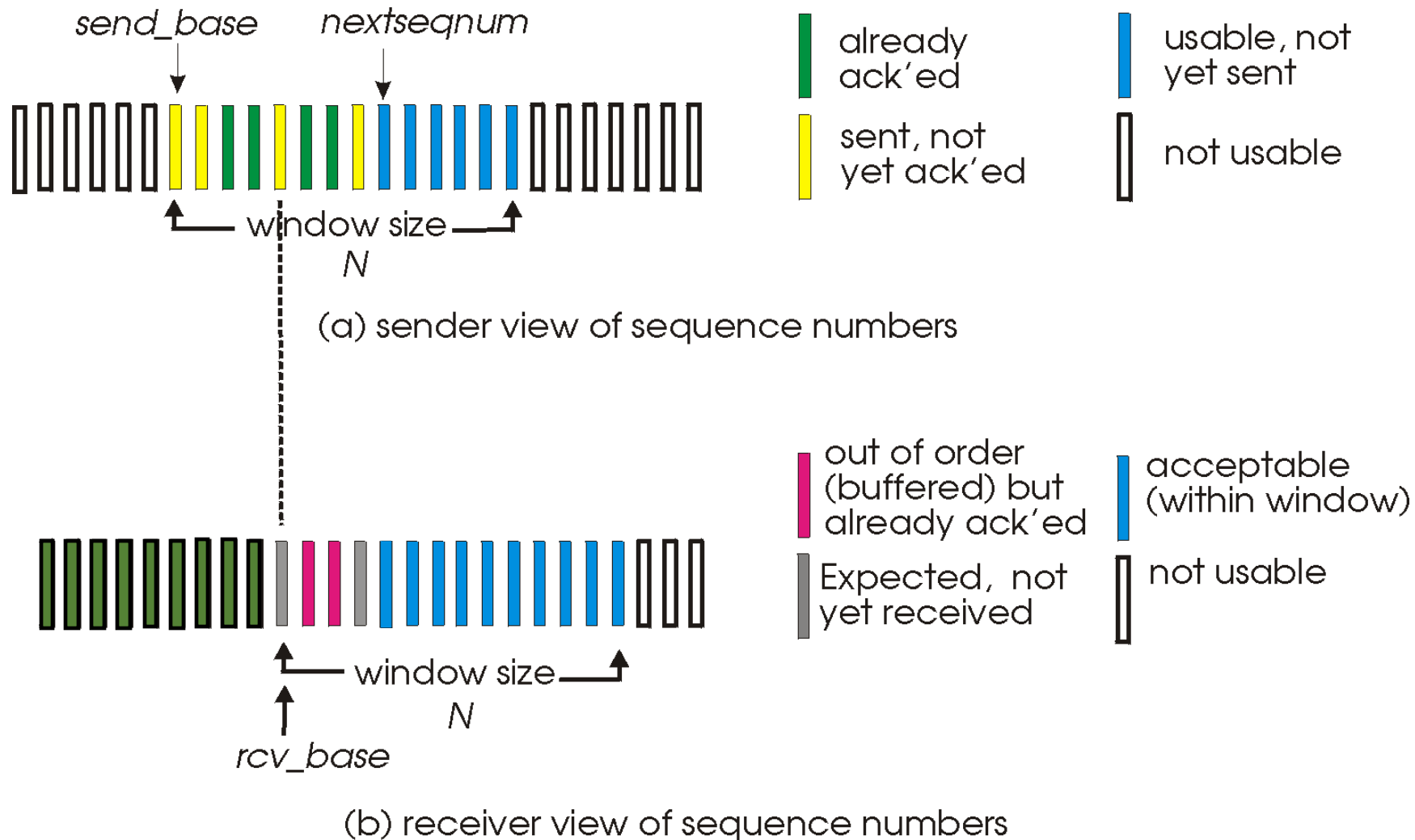
Selective repeat: the approach

- *pipelining*: multiple packets in flight
- *receiver individually ACKs* all correctly received packets
 - buffers packets, as needed, for in-order delivery to upper layer
- sender:
 - maintains (conceptually) a timer for each unACKed pkt
 - timeout: retransmits single unACKed packet associated with timeout
 - maintains (conceptually) “window” over *N* consecutive seq #s
 - limits pipelined, “in flight” packets to be within this window

Selective repeat: sender, receiver windows



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-1]:

- 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

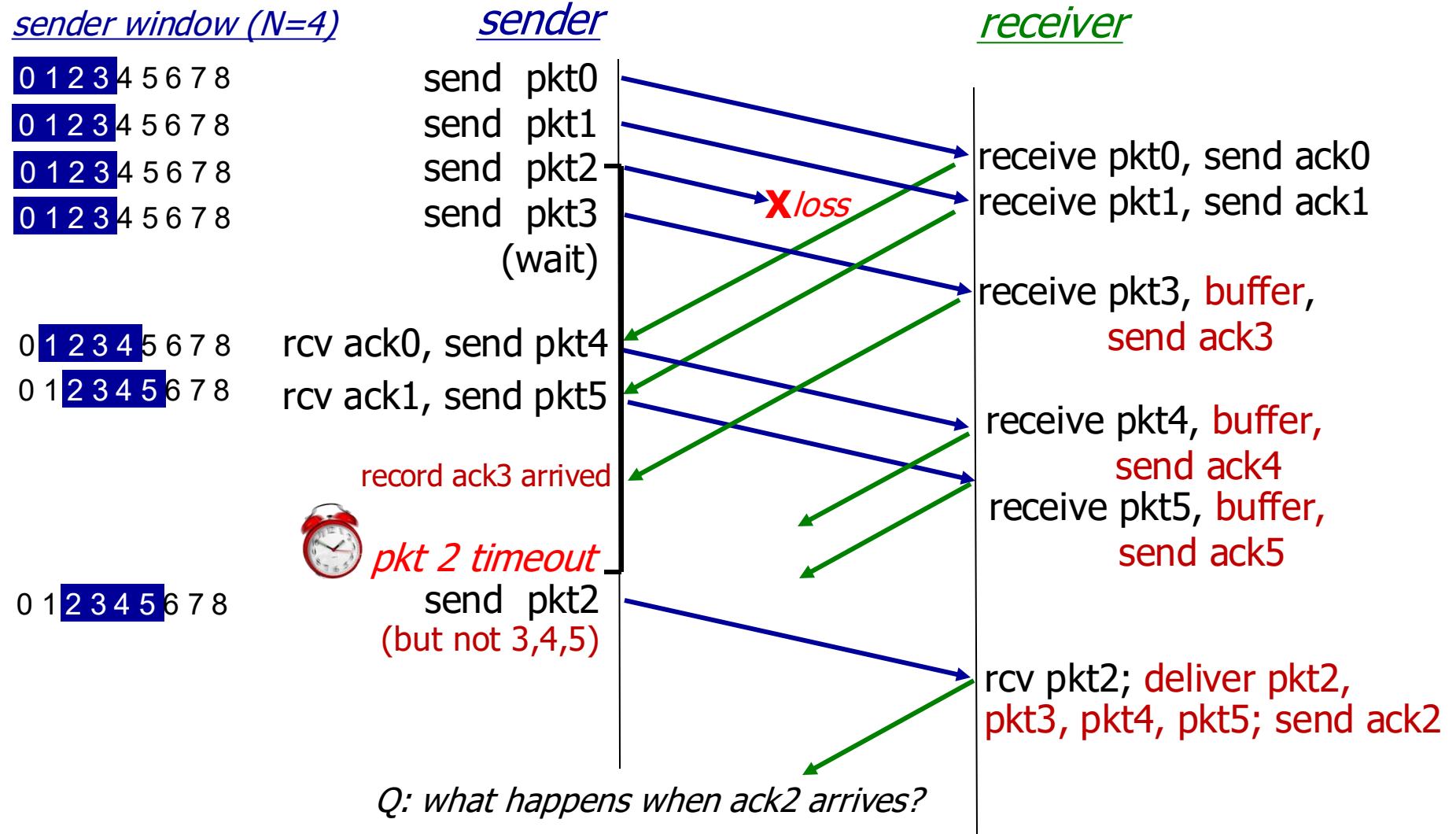
packet n in [rcvbase-N, rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective Repeat in action



Selective Repeat in action

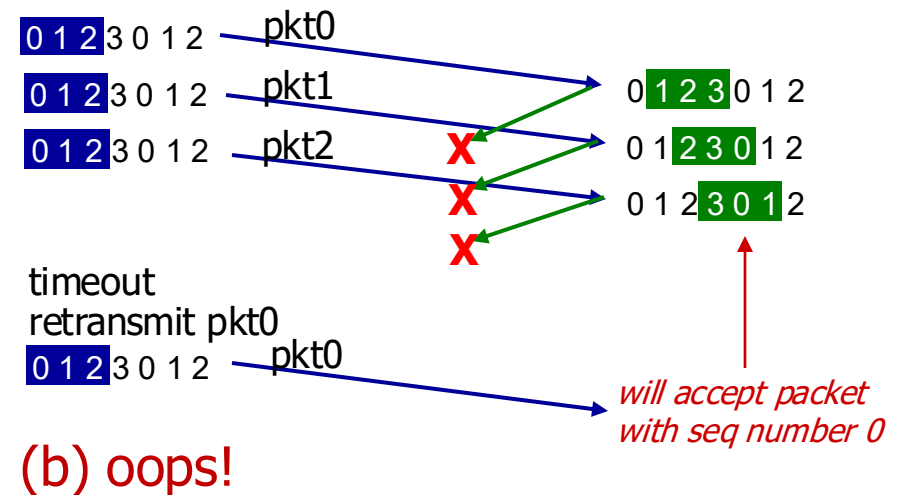
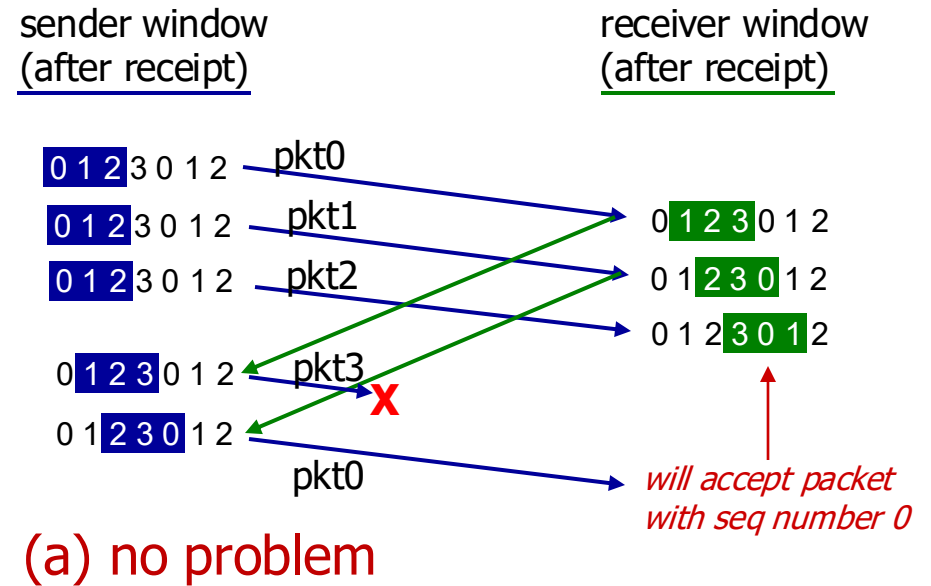
- Animation here:

https://media.pearsoncmg.com/ph/esm/ecs_kurose_com_pnetwork_8/cw/content/interactiveanimations/selective-repeat-protocol/index.html

Selective repeat: a dilemma!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



Selective repeat: a dilemma!

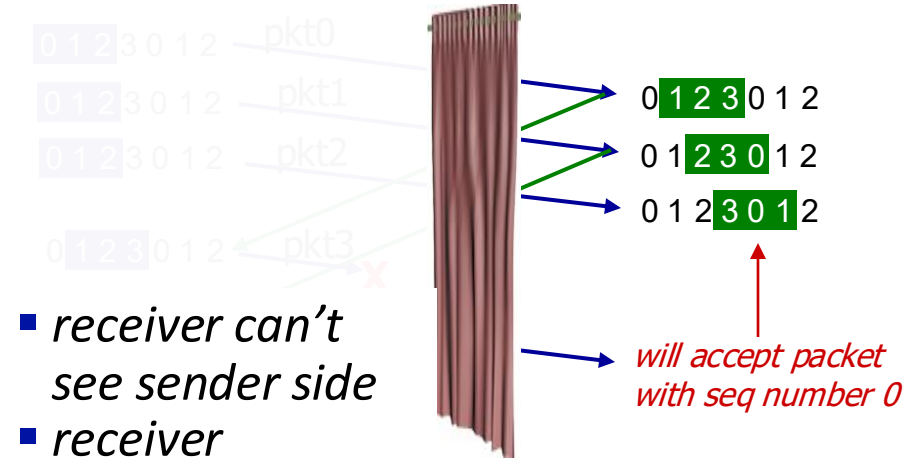
example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?

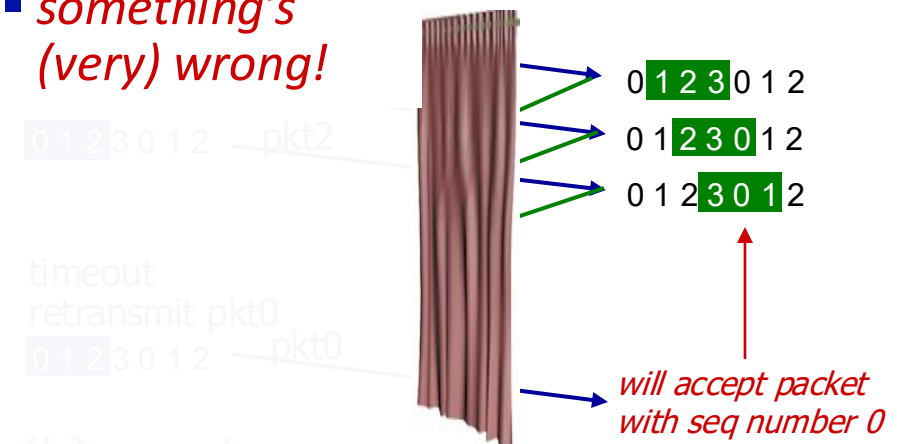
sender window
(after receipt)

receiver window
(after receipt)



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



■ *something's (very) wrong!*





(b) oops!

Summary for rdt tools

■ ACK/NAK

- provides receiver feedback 
- can also be corrupted or lost 


■ Timer

- detects pkt/feedback loss 
- may lead to duplicate pkts 

■ Sequence number

- detects duplicate pkts 
- Has to be a bounded number of bits 

■ Sliding window

- allows for pipelining pkt 
- reuses sequence number