UNIVERSITY OF WATERLOO

CS 456/656
Computer Networks

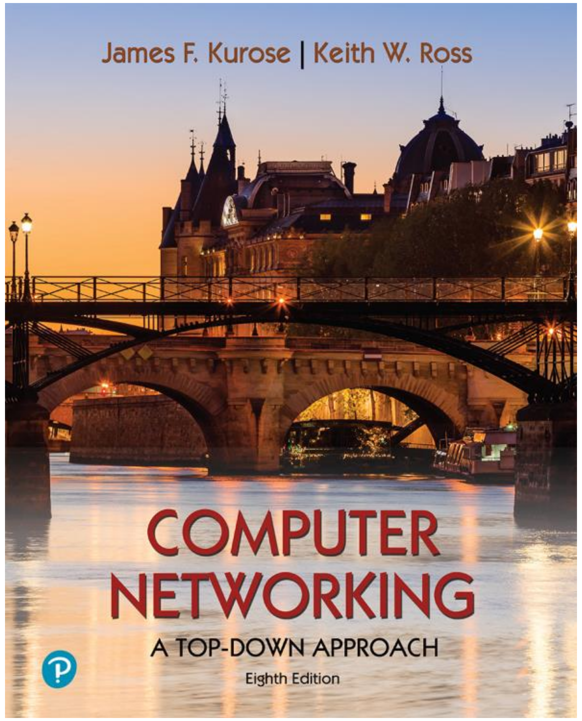
Lecture 6: Transport Layer – Part 2

Mina Tahmasbi Arashloo and Bo Sun

Fall 2024

A note on the slides

Adapted from the slides that accompany this book.



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
- One of the most important services a transport protocol can provide over an unreliable network layer

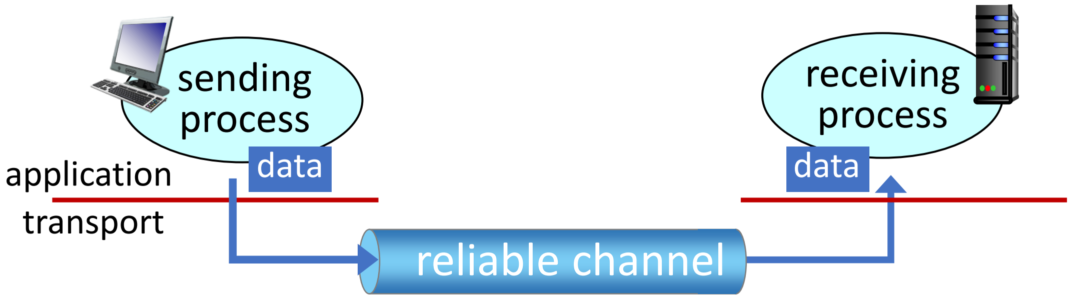
Principles of RDT - Agenda

- `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 (`rdt`)



sending process

receiving process

application

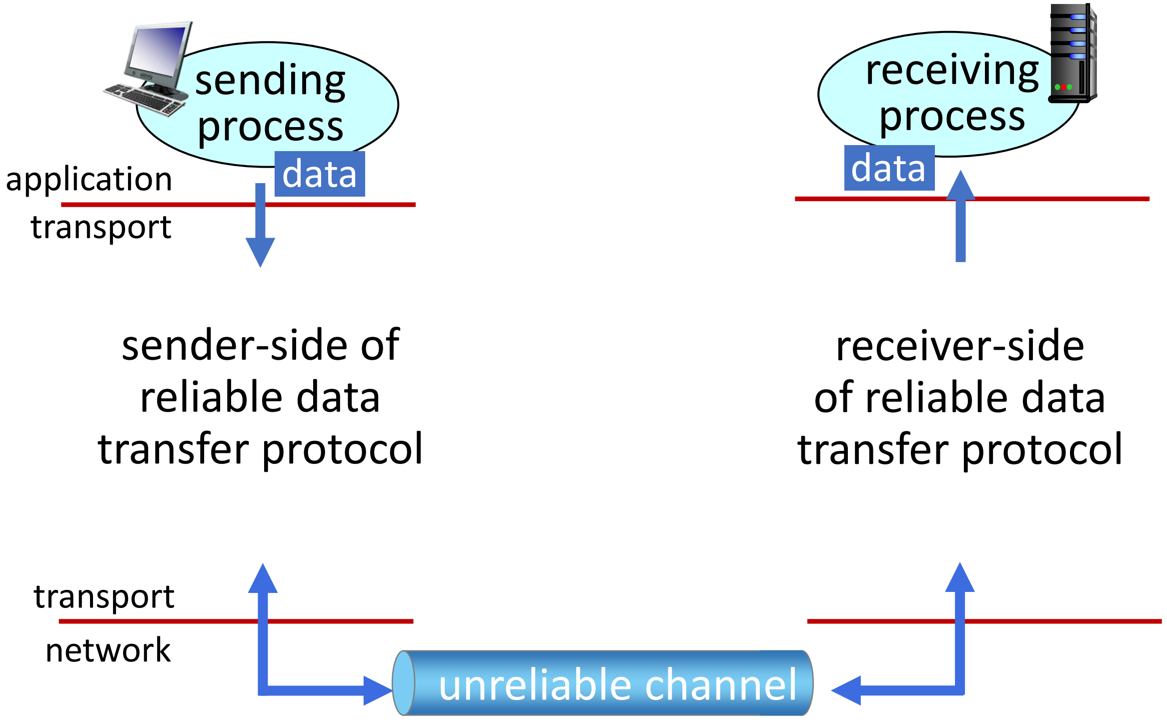
transport

data

data

reliable channel

reliable service *abstraction*



sending process

receiving process

application

transport

data

data

sender-side of reliable data transfer protocol

receiver-side of reliable data transfer protocol

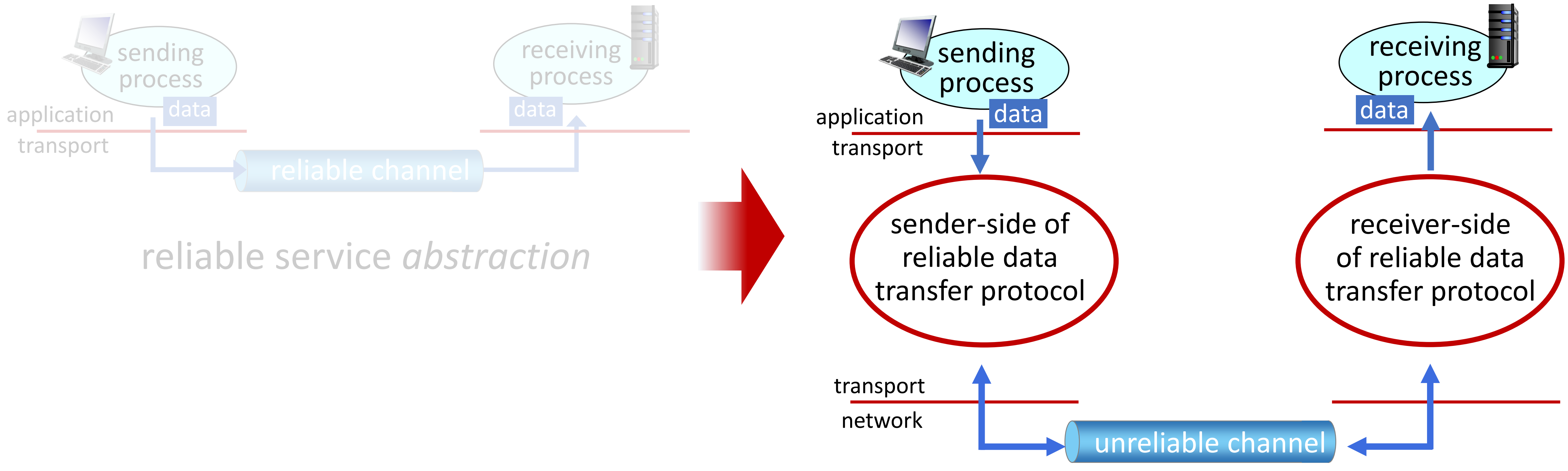
transport

network

unreliable channel

reliable service *implementation*

Reliable data transfer (`rdt`)



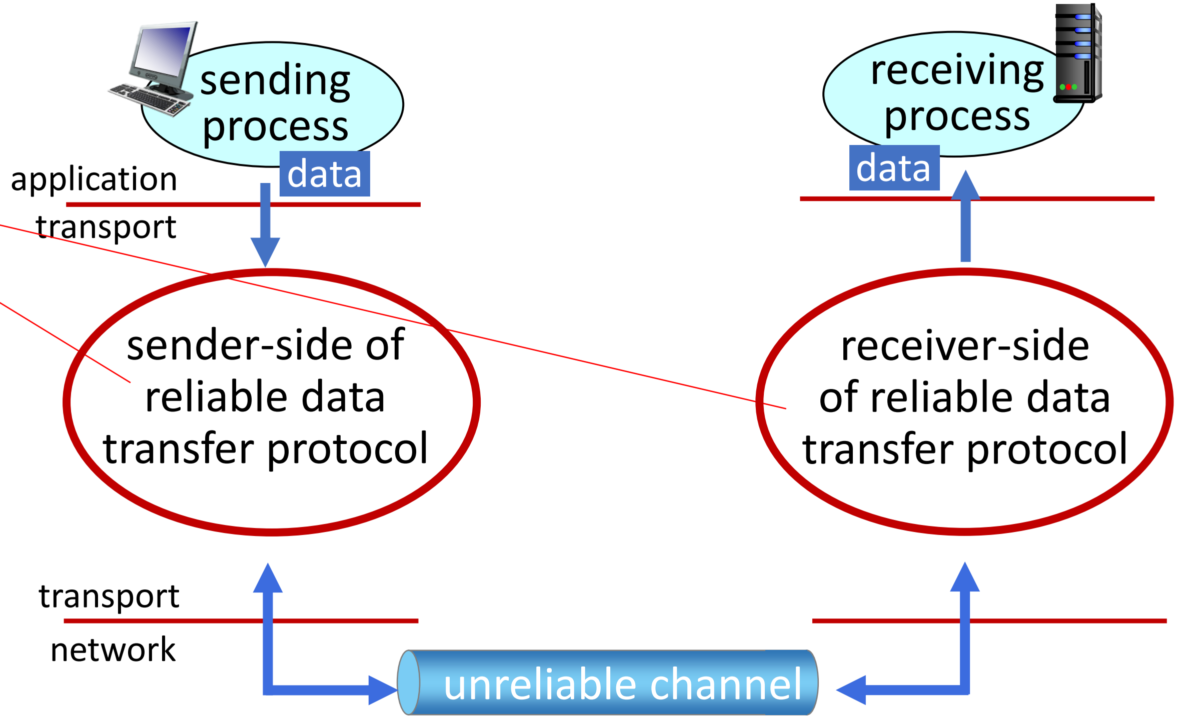
reliable service *abstraction*

reliable service *implementation*

Reliable data transfer (rdt)

- 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

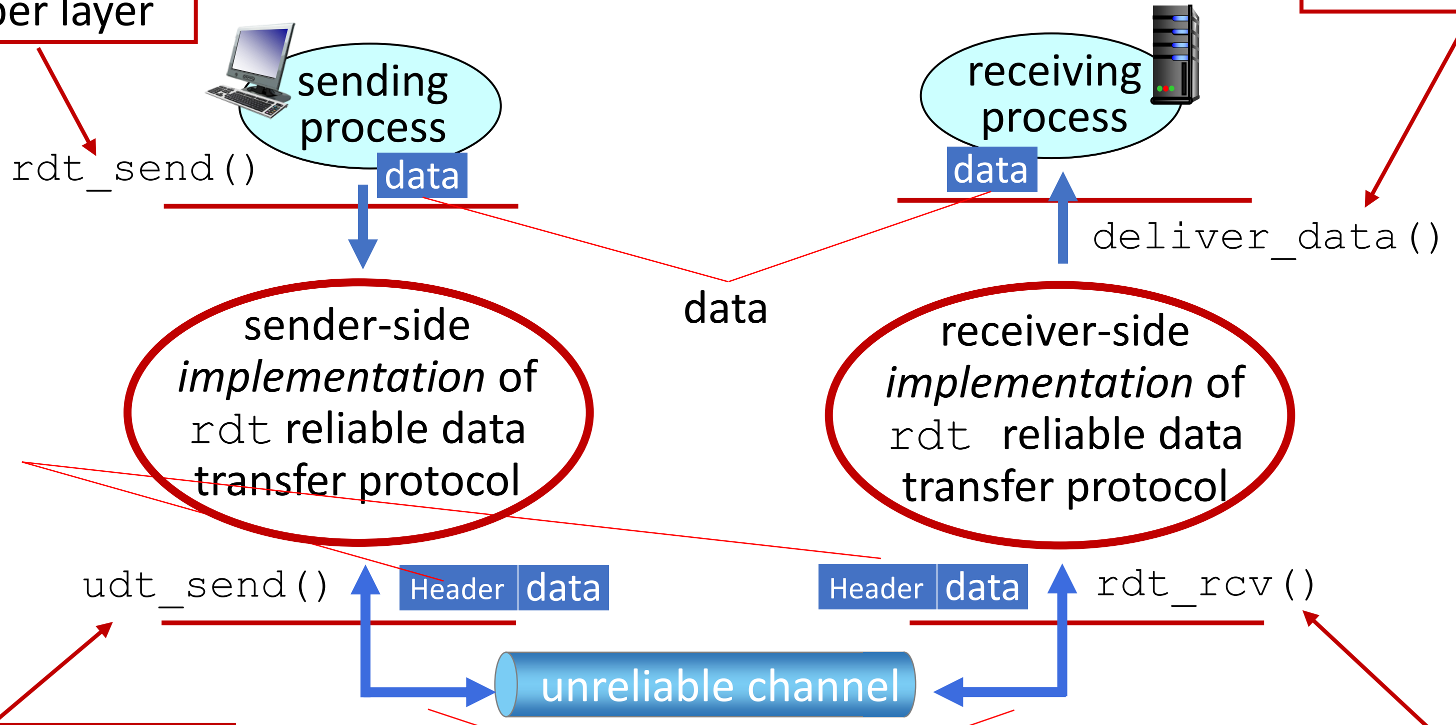

sending process
data
application
transport
sender-side of reliable data transfer protocol
transport
network
unreliable channel
receiving process
data
receiver-side of reliable data transfer protocol

reliable service *implementation*

Reliable data transfer protocol (`rdt`): interfaces

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 layer


sending process
rdt_send()
data
receiving process
data
deliver_data()
data
sender-side *implementation* of rdt reliable data transfer protocol
receiver-side *implementation* of rdt reliable data transfer protocol
udt_send()
Header data
Header data
rdt_rcv()
unreliable channel

One transport-layer segment (a.k.a a packet over the network)

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

Bi-directional communication over unreliable channel
(e.g., for reliable data transfer)

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

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
  - ARQ (Automatic Repeat request) protocols

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

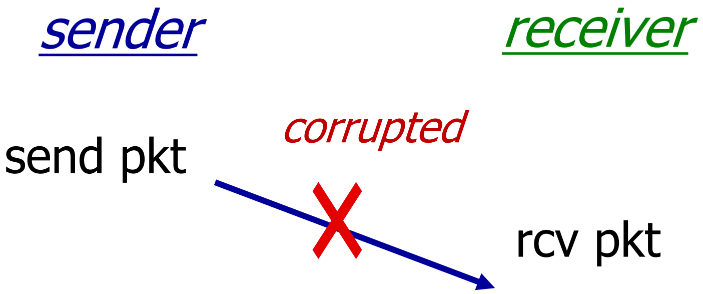
- Send a segment
- Wait to make sure it is delivered properly
- Then send the next one
- We will develop a “simple” stop-and-wait protocol in class as an example

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

Q: How do humans recover from "errors" during conversation?

Simple stop-and-wait protocol (v1)



Unreliable channel

- Channel with bit errors

rdt tools

- **?**

Channel with bit errors

- underlying channel may flip bits in pkts
  - checksum to detect bit errors
- *the* question: how to recover from errors?
  - *ACKs:* receiver explicitly tells sender that pkt received OK
  - *NAKs:* receiver explicitly tells sender that pkt had errors
  - sender *retransmits* pkt on receipt of NAK

Example stop-and-wait protocol (v1)

Sender

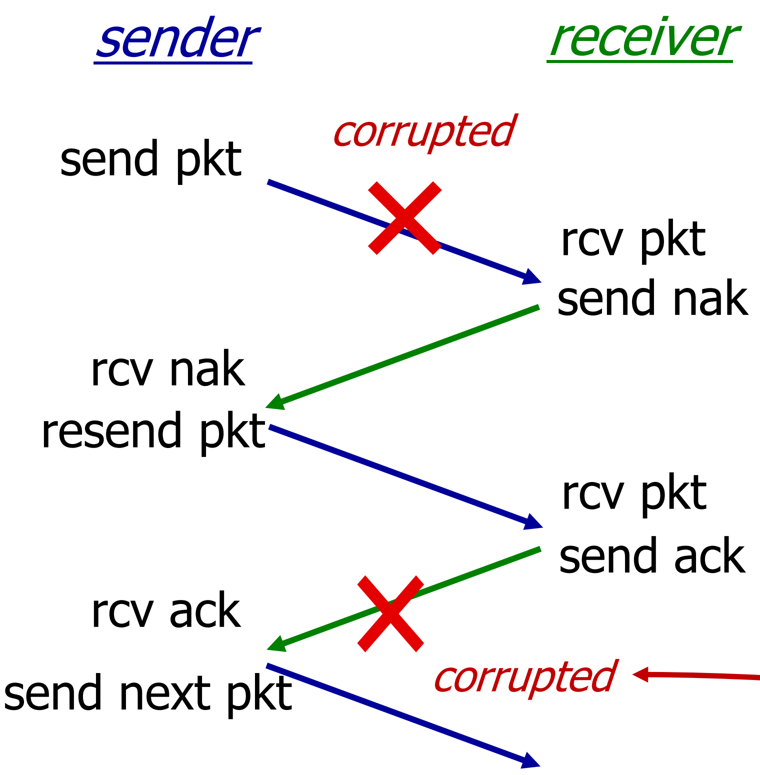
- Send a pkt
- Wait to get an ACK/NAK
  - If NAK, resend the pkt
    - go back to waiting
  - If ACK, proceed with sending next pkt

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	*rdt tools*
- Channel with bit errors • Corrupted data pkts • Corrupted feedback	- Checksum, ACK/NAK, retransmission + ?

what happens if ACK/NAK corrupted?

Corrupted feedback

what happens if ACK/NAK corrupted?

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

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

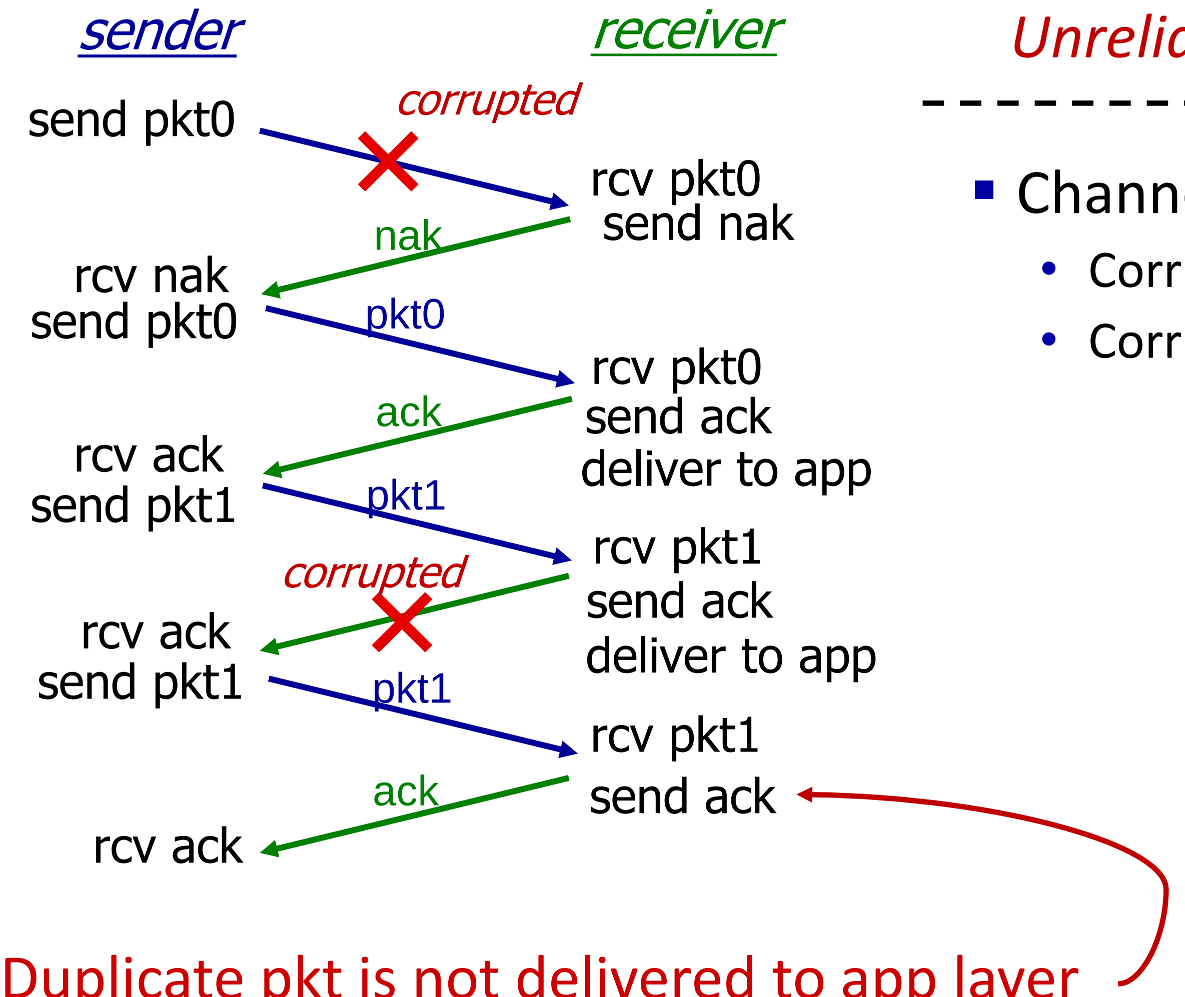
- Send a pkt
  - Seq # = 1 – last seq #
- Wait to get an ACK/NAK
  - If NAK or corrupted, resend
    - go back to waiting
  - If ACK, proceed with next pkt

Receiver

- When pkt is received
  - If correct pkt, send ACK
    - If Seq # ≠ 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)



Duplicate pkt is not delivered to app layer

Unreliable channel	*rdt tools*
Channel with bit errors • Corrupted data pkts • Corrupted feedback	Checksum, ACK/NAK, retransmission, sequence number

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

Sender

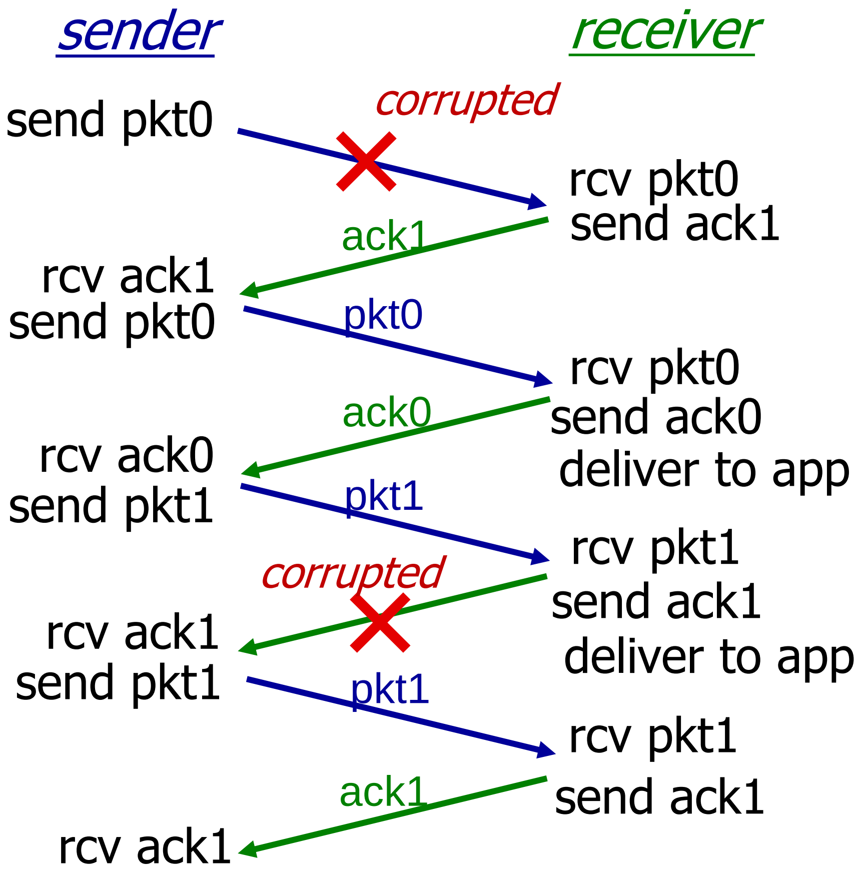
- Send a pkt
  - Seq # = 1 – last seq #
- Wait to get an ACK
  - If ACK (& last Seq #) or corrupted, resend
    - go back to waiting
  - If ACK (& Seq #), proceed with next pkt

Receiver

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



Unreliable channel

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

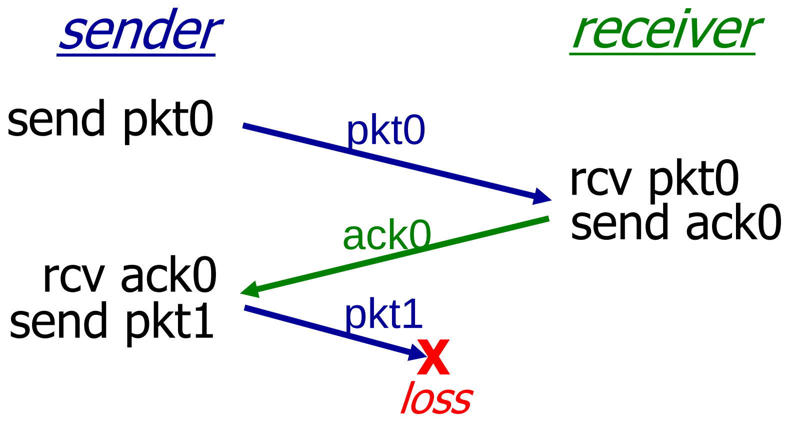
rdt tools

- Checksum, ACK, retransmission, sequence number

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?

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

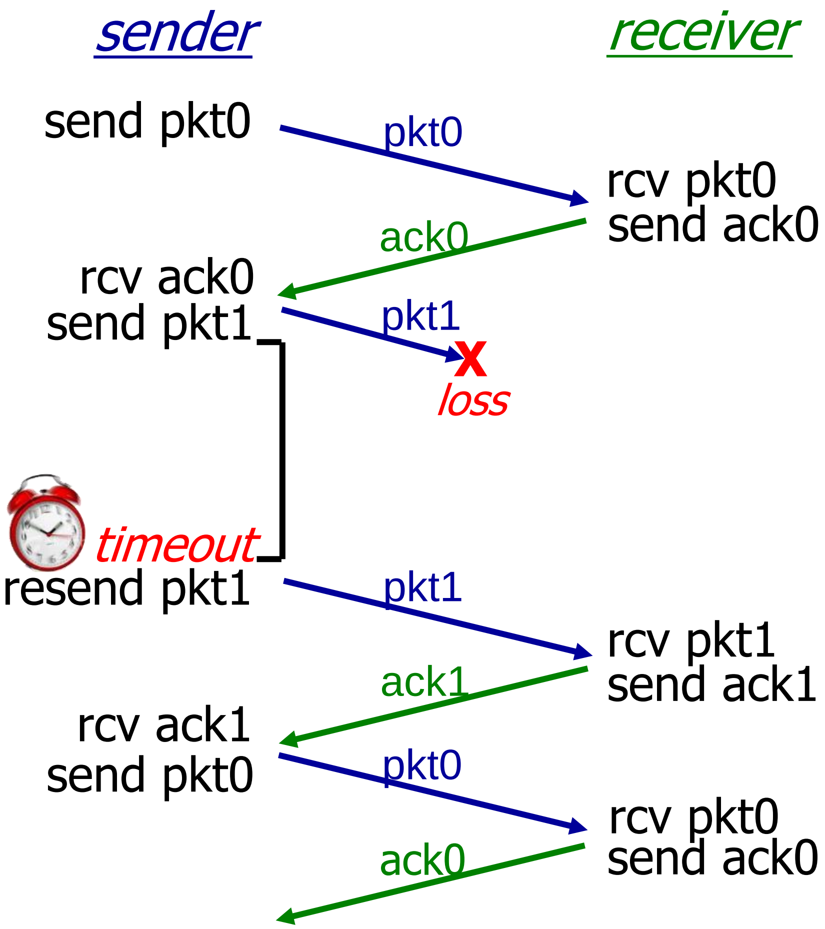
- Send a pkt
  - Seq # = 1 – last Seq #
  - Set timer
- Wait to get an ACK
  - If ACK (& last Seq #) or corrupted, resend pkt and reset timer
    - go back to waiting
  - If ACK (& Seq #), remove timer and proceed with next pkt
  - If timer goes off, resent pkt and reset timer

Receiver

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

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

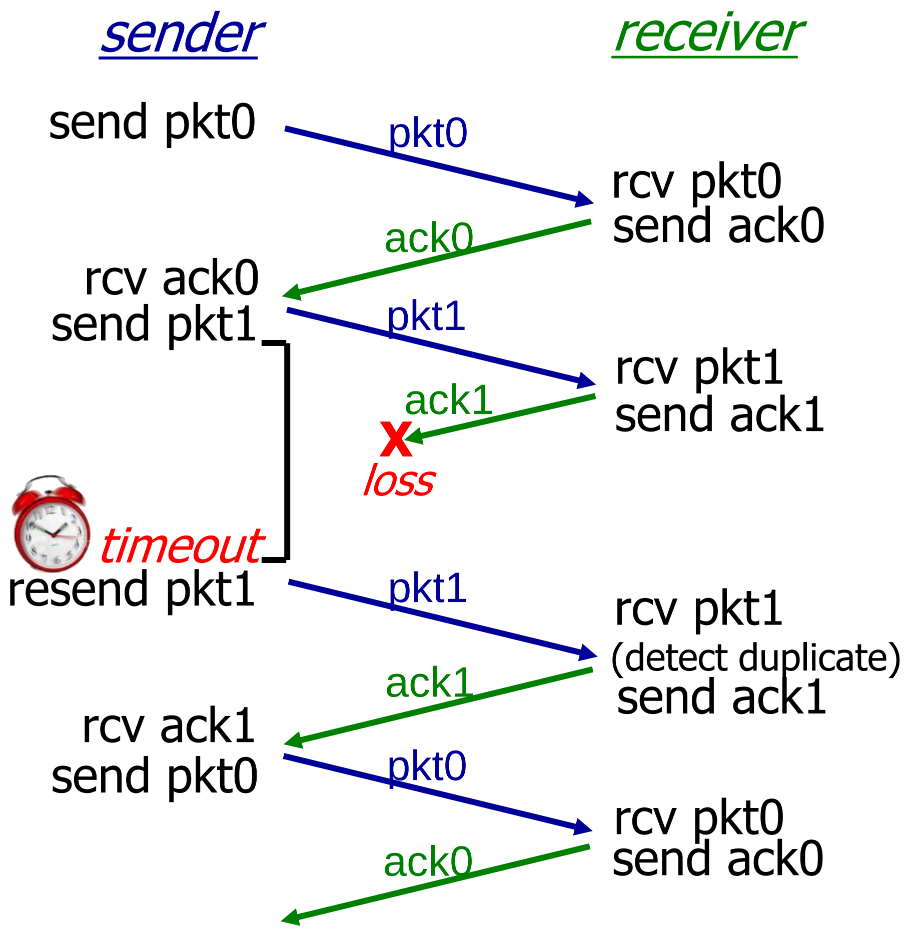
Example stop-and-wait protocol (v3)



(a) packet loss

Unreliable channel	*rdt tools*
▪ Channel with bit errors • Corrupted data pkts • Corrupted feedback	▪ Checksum, ACK, retransmission, sequence number
▪ Channel with errors and lost • lost data pkts	▪ Checksum, ACK, retransmission, sequence number, timer

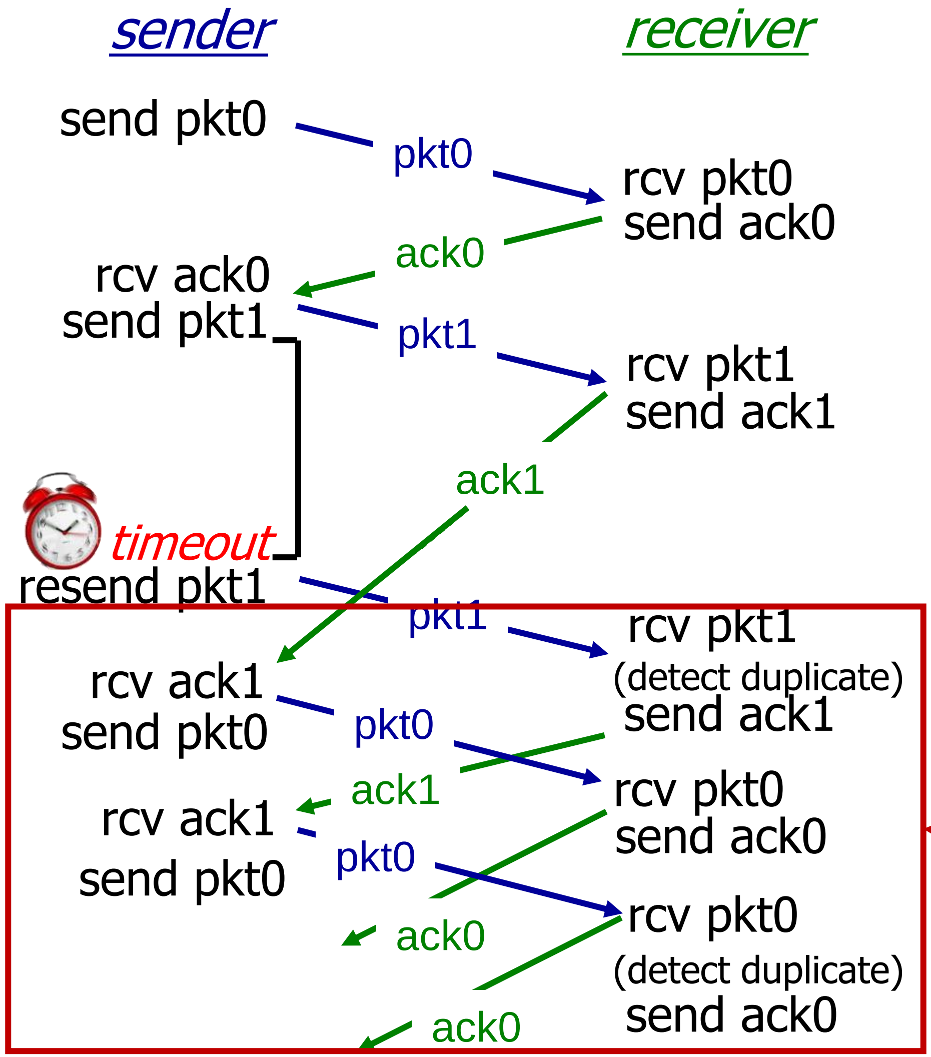
Example stop-and-wait protocol (v3)



(b) ACK loss

Unreliable channel	*rdt tools*
▪ Channel with bit errors • Corrupted data pkts • Corrupted feedback	▪ Checksum, ACK, retransmission, sequence number
▪ Channel with errors and lost • lost data pkts • lost feedback	▪ Checksum, ACK, retransmission, sequence number, timer

Example stop-and-wait protocol (v3)



(c) premature timeout/ delayed ACK

Duplicate pkt continues

Unreliable channel	*rdt tools*
▪ Channel with bit errors • Corrupted data pkts • Corrupted feedback	▪ Checksum, ACK, retransmission, sequence number
▪ Channel with errors and lost • lost data pkts • lost feedback	▪ Checksum, ACK, retransmission, sequence number, timer

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

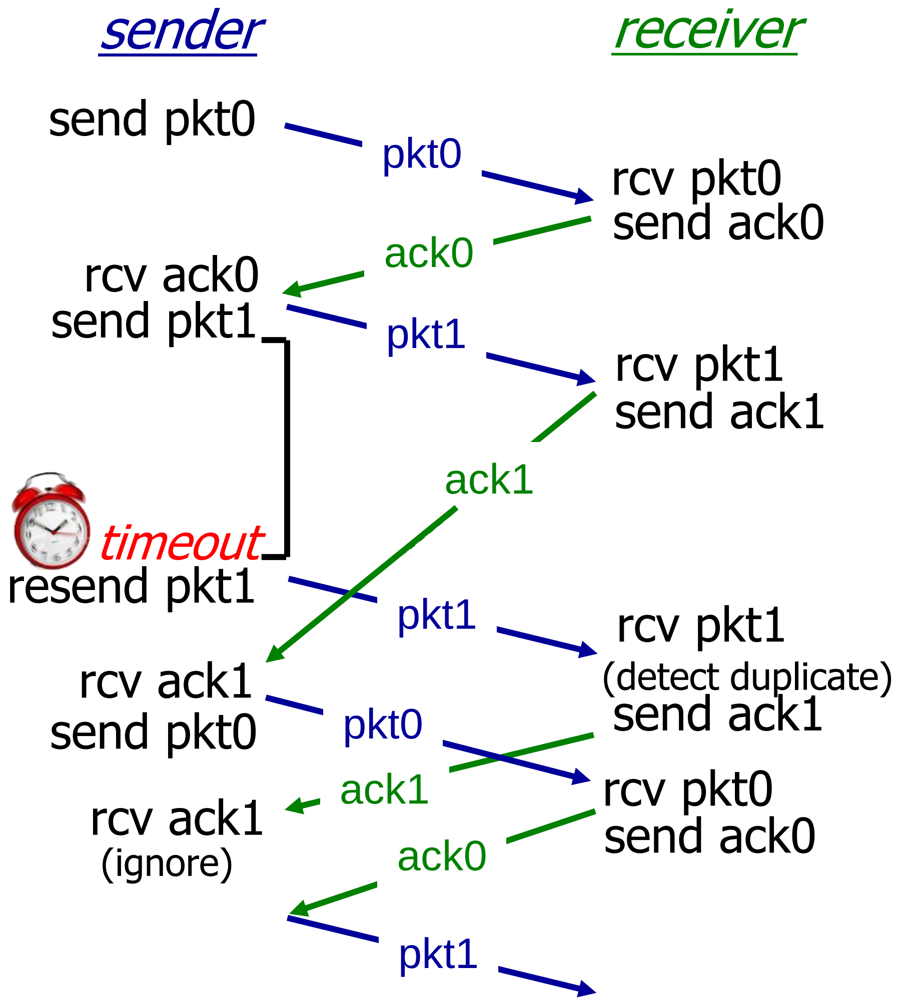
Receiver

- When pkt is received
  - If correct pkt, send ACK (& Seq #)
    - If Seq # ≠ 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	*rdt tools*
Channel with bit errors • Corrupted data pkts • Corrupted feedback	Checksum, ACK, retransmission, sequence number
Channel with errors and lost • lost data pkts • lost feedback	Checksum, ACK, retransmission, sequence number, timer

Principles of reliable data transfer (**`rdt`**)

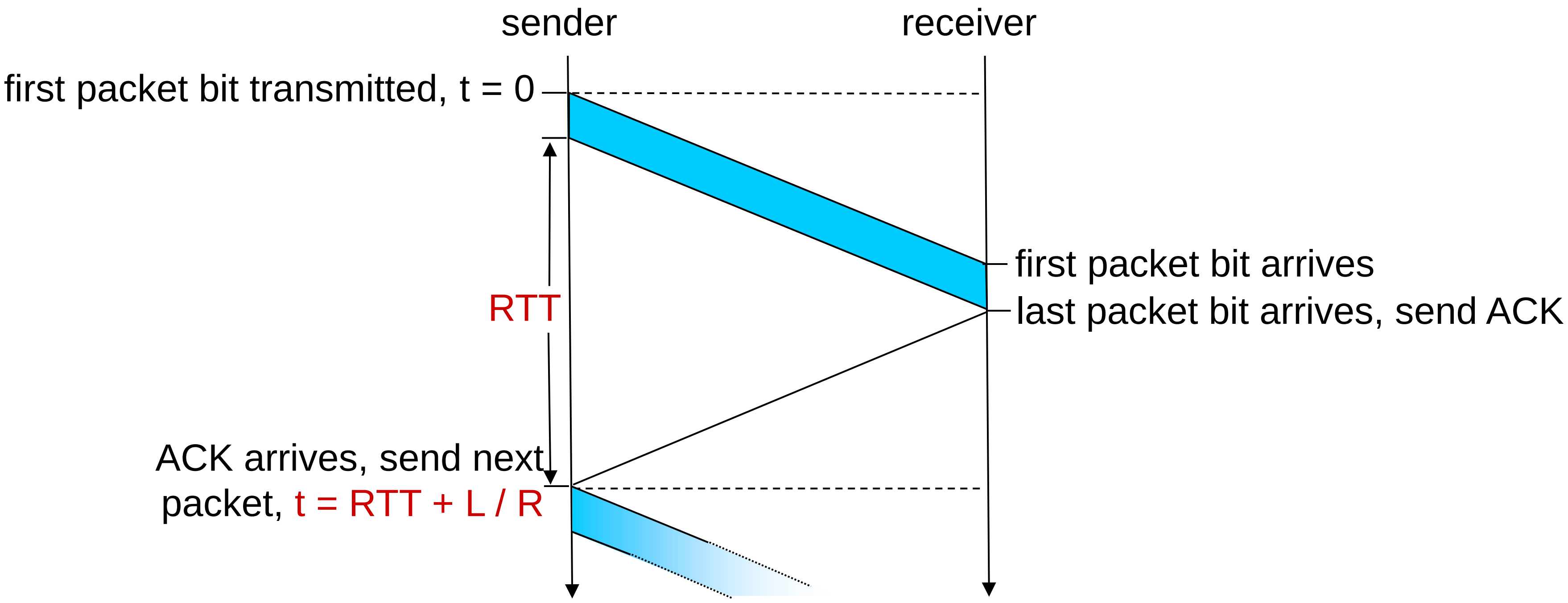
- `rdt` at a glance
- Stop-and-wait approach
  - sender sends one pkt, then waits for receiver's response
- Pipelined approach
  - Go-back-N (GBN)
  - Selective Repeat (SR)

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

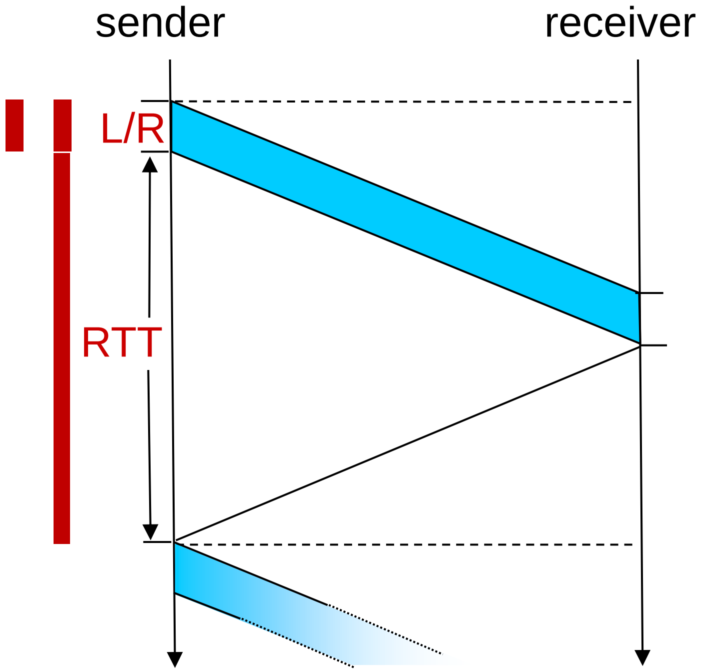

sender
receiver
first packet bit transmitted, t = 0
first packet bit arrives
last packet bit arrives, send ACK
RTT
ACK arrives, send next packet, t = RTT + L / R

Stop-and-wait protocol has a problem

$$U_{\text{sender}} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$

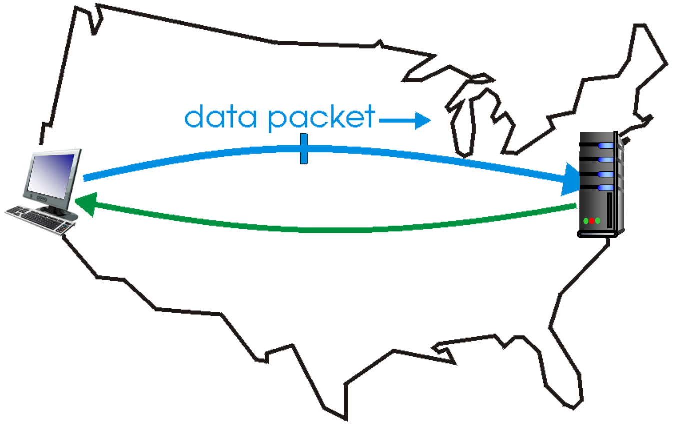


- Protocol performance stinks!
- Protocol limits performance of underlying infrastructure (channel)

Pipelined protocols operation

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

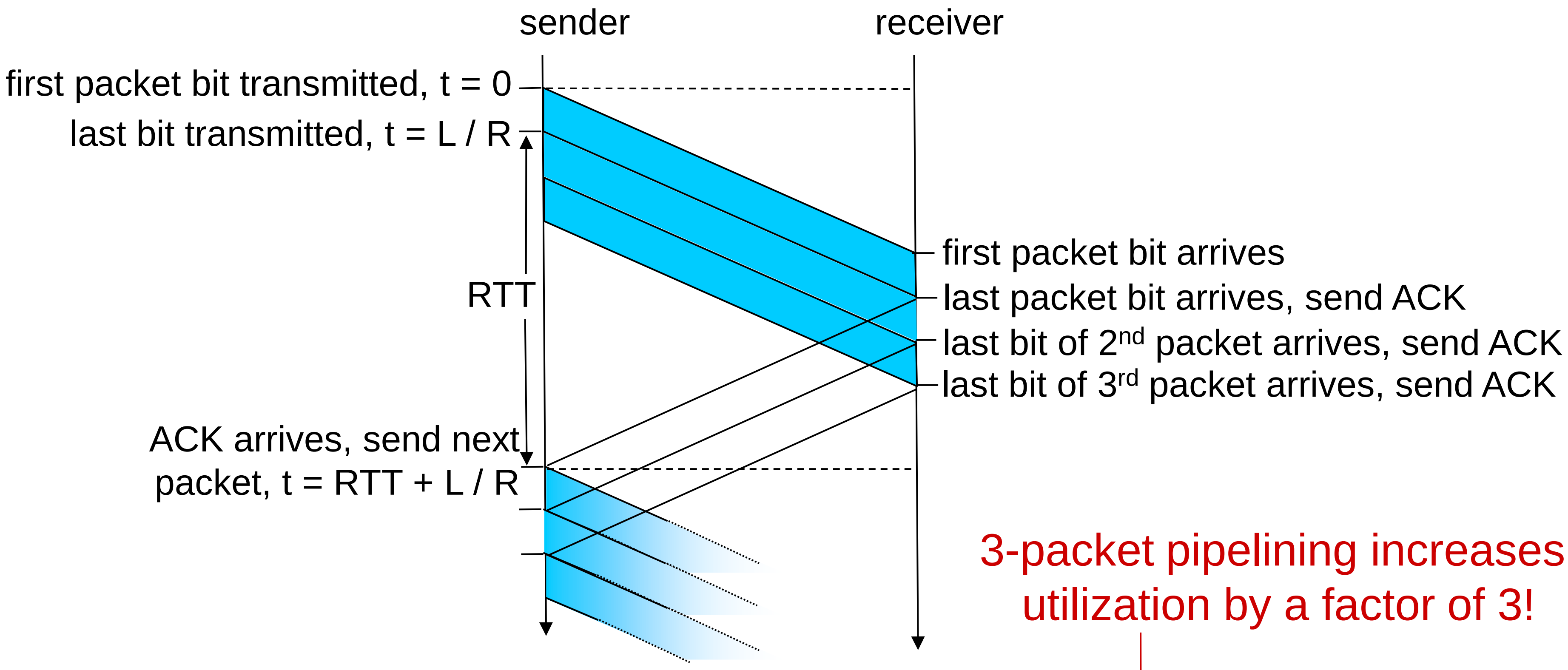
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

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

Pipelining: increased utilization

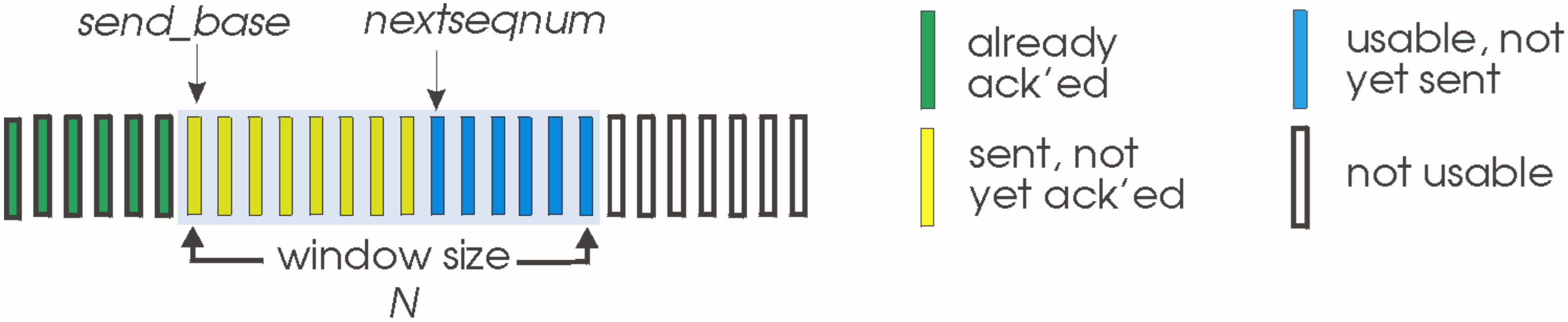

sender
receiver
first packet bit transmitted, t = 0
last bit transmitted, t = L / R
RTT
ACK arrives, send next packet, t = RTT + L / R
first packet bit arrives
last packet bit arrives, send ACK
last bit of 2nd packet arrives, send ACK
last bit of 3rd packet arrives, send ACK

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

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

Go-Back-N: sender

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

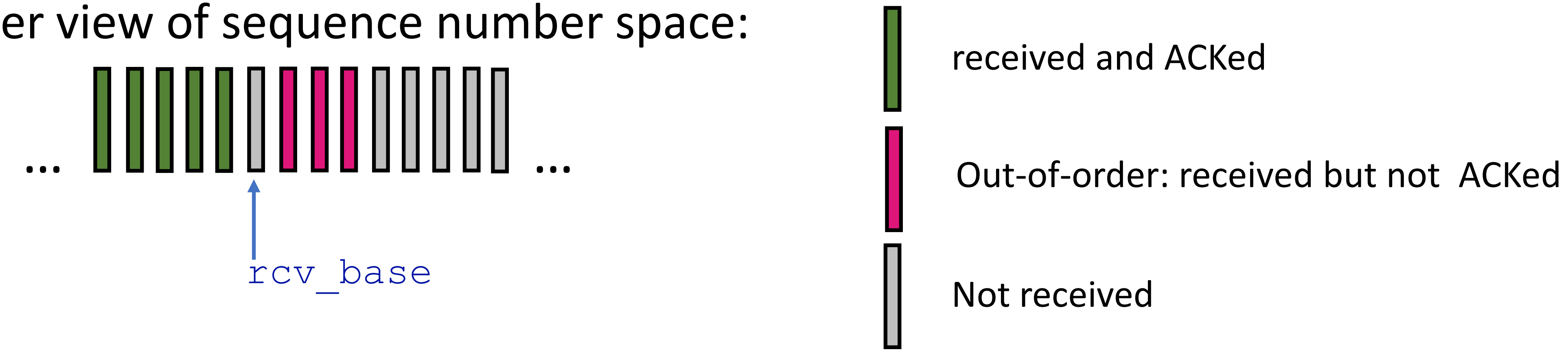


- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

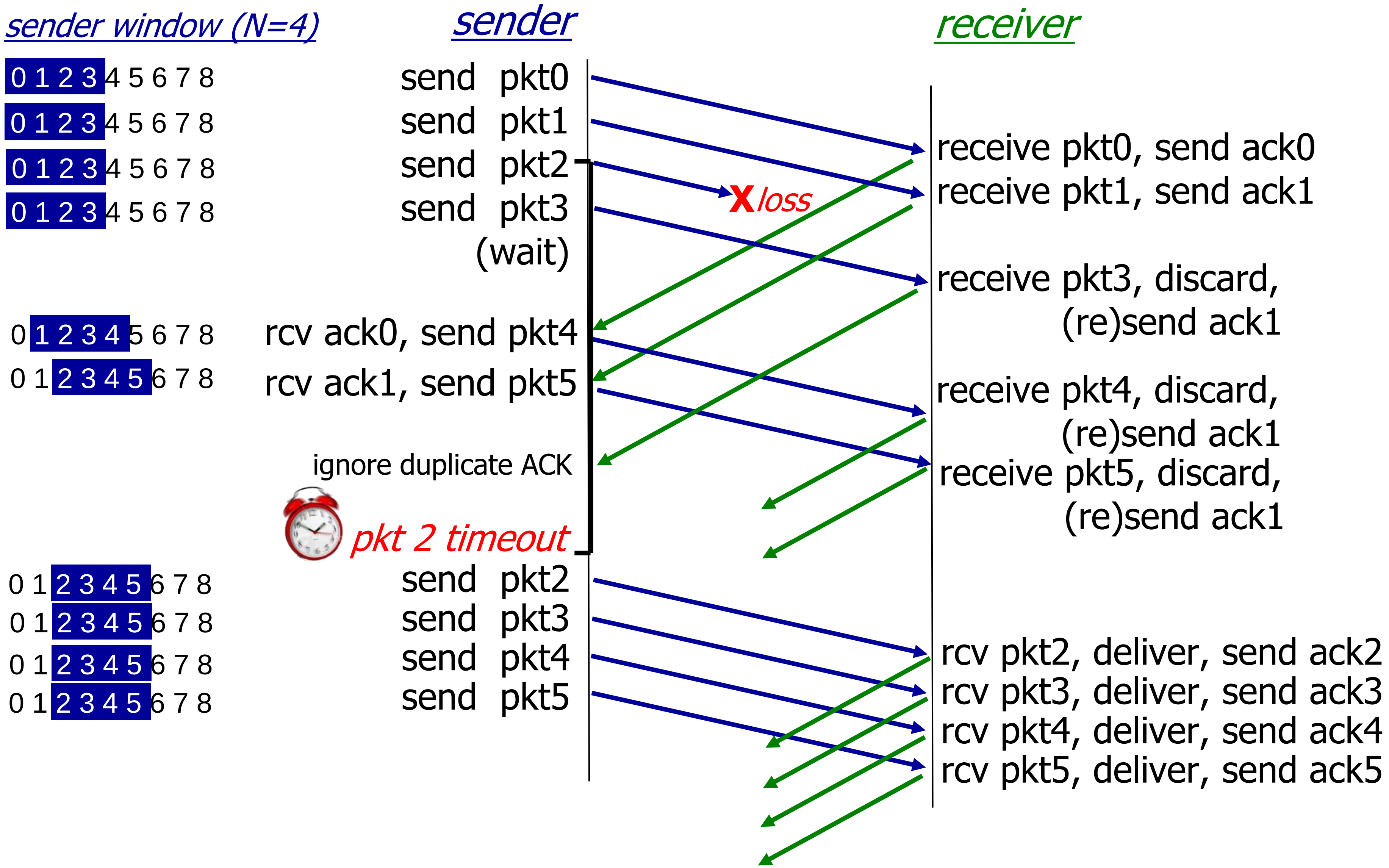
Go-Back-N: receiver

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

Receiver view of sequence number space:



Go-Back-N 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

Go-Back-N in action

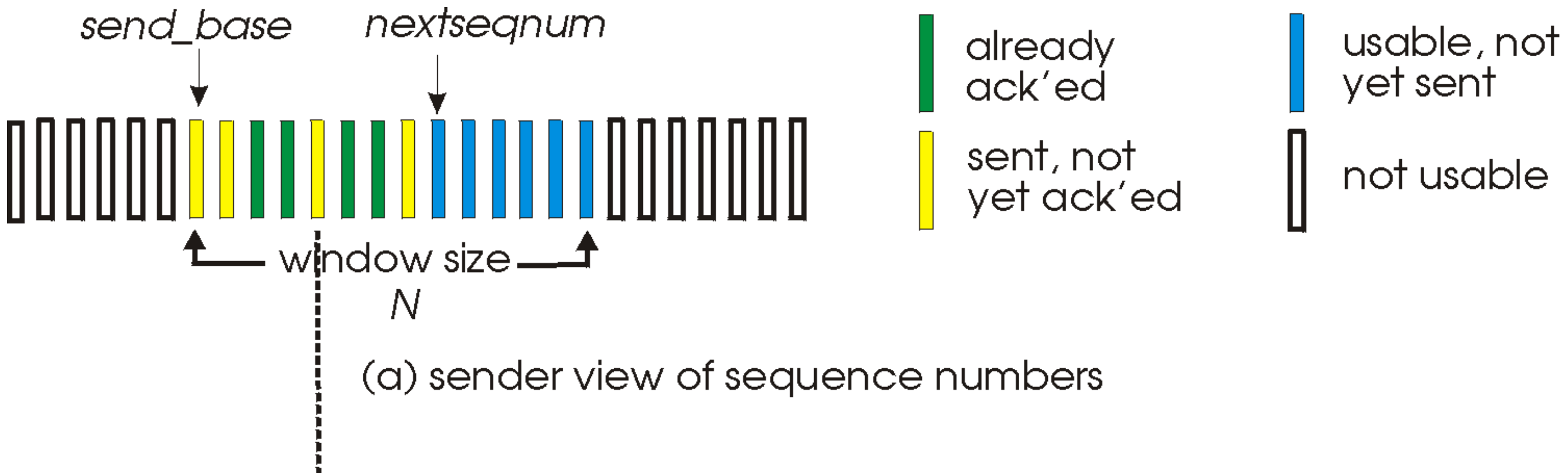
- Animation here:

https://media.pearsoncmg.com/ph/esm/ecs_kurose_compnetwork_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



(a) sender view of sequence numbers

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

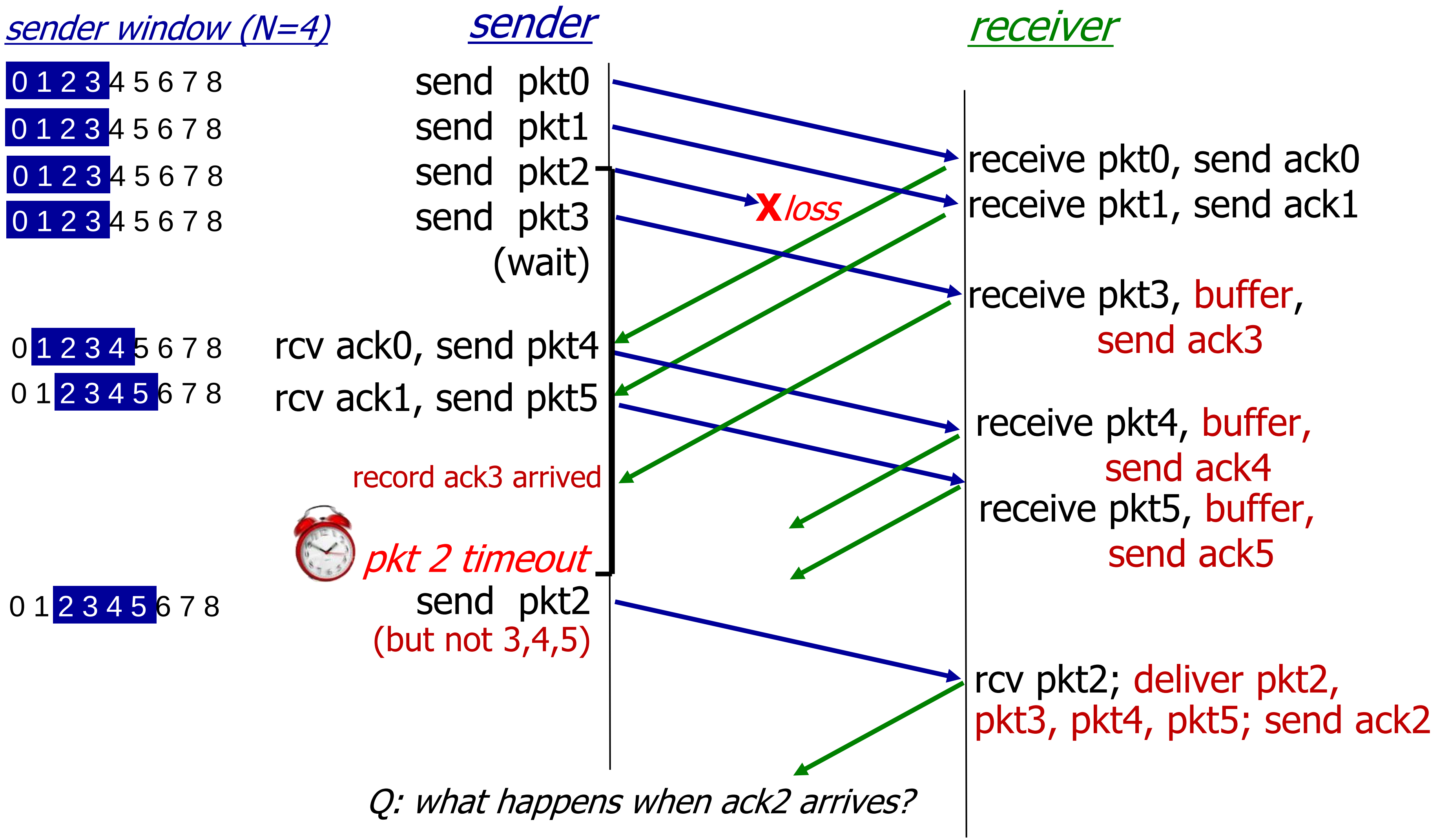
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

Selective Repeat in action



Q: what happens when ack2 arrives?

Selective Repeat in action

- Animation here:

https://media.pearsoncmg.com/ph/esm/ecs_kurose_compnetwork_8/cw/content/interactiveanimations/selective-repeat-protocol/index.html

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