CS 330: Network Applications & Protocols

Transport Layer

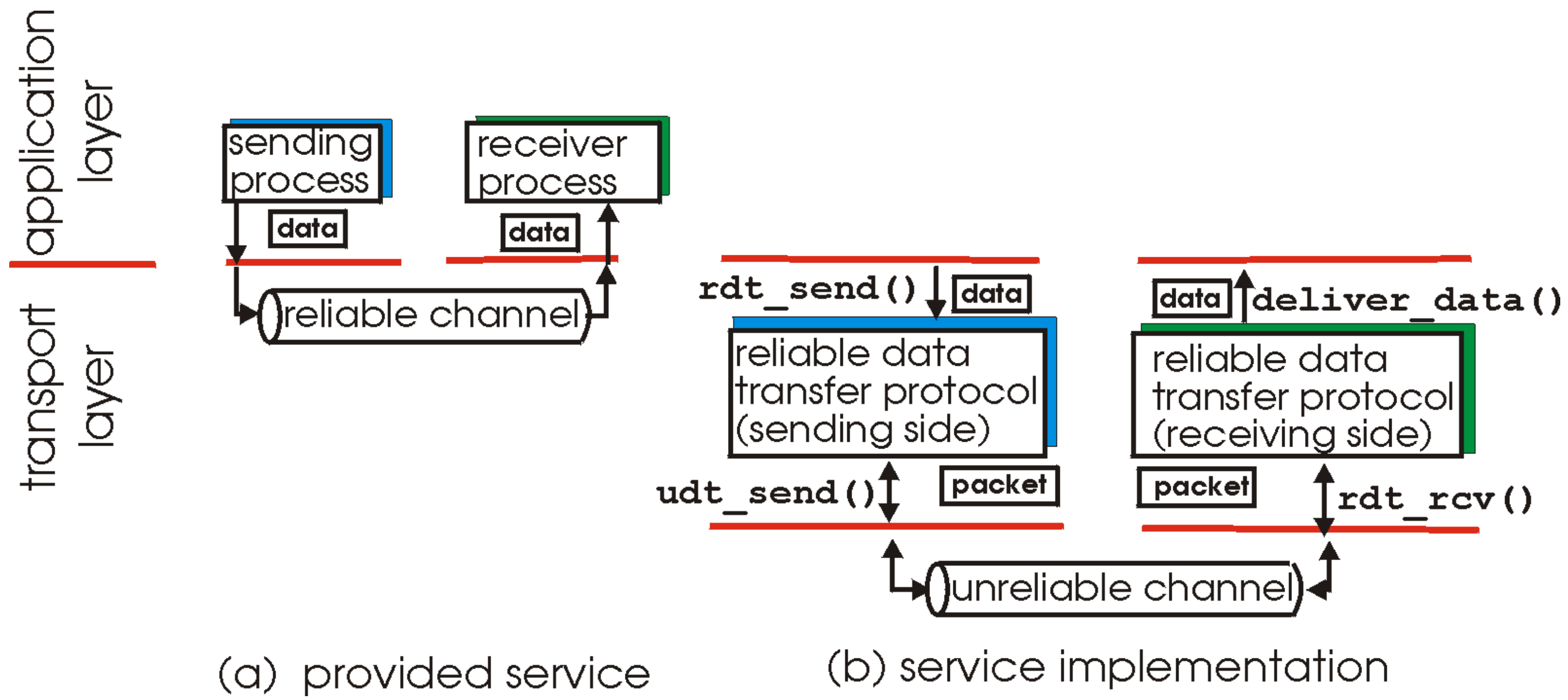
Galin Zhelezov
Department of Physical Sciences
York College of Pennsylvania

Overview of Transport Layer

- **Transport-layer Services**
- **Multiplexing and Demultiplexing**
- **Connectionless Transport: UDP**
- **Principles of Reliable Data Transfer**
  - Overview
  - Pipelined Protocols
    - Go-Back-N
    - Selective Repeat
- **Connection-oriented Transport: TCP**
- **Principles of Congestion Control**
- **TCP Congestion Control**

Principles of Reliable Data Transfer

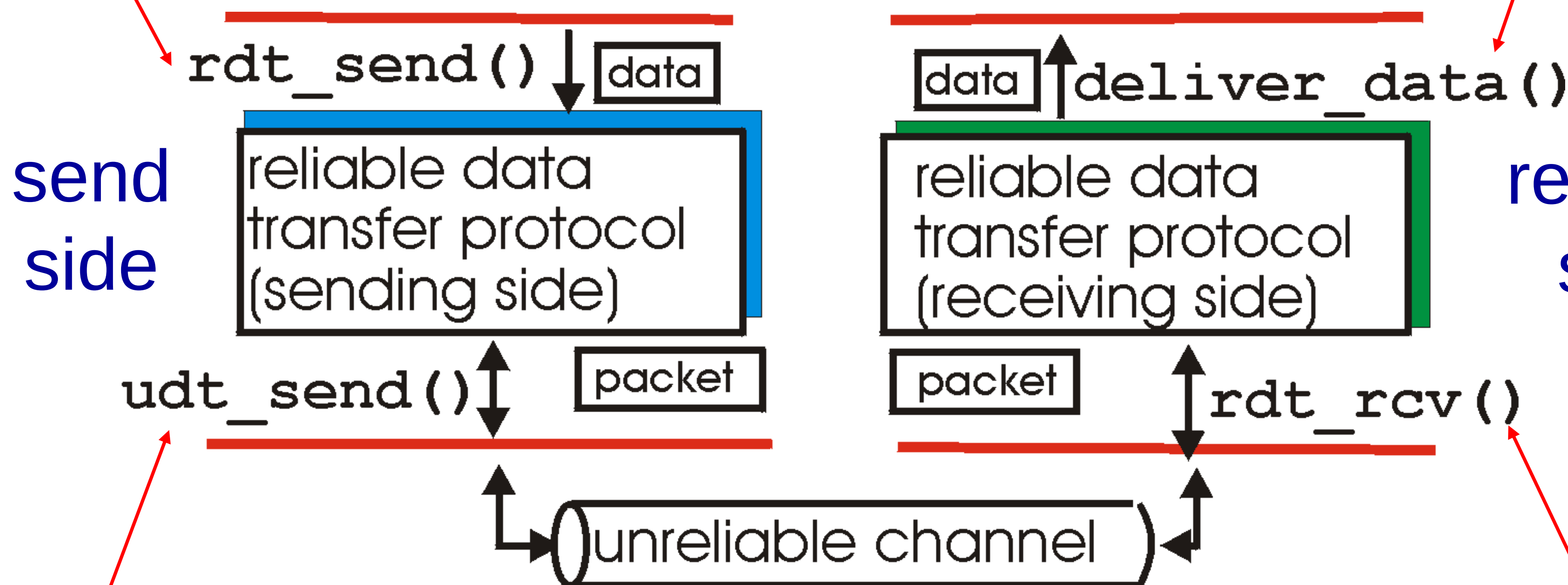
- **Reliable data transfer is very important application, transport, and link layers**
- **The characteristics of unreliable channel will determine the complexity of a Reliable Data Transfer (RDT) protocol**
- **To explore reliable data transfer, examine different types of loss and how to address them**



Reliable data transfer: getting started

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

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



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

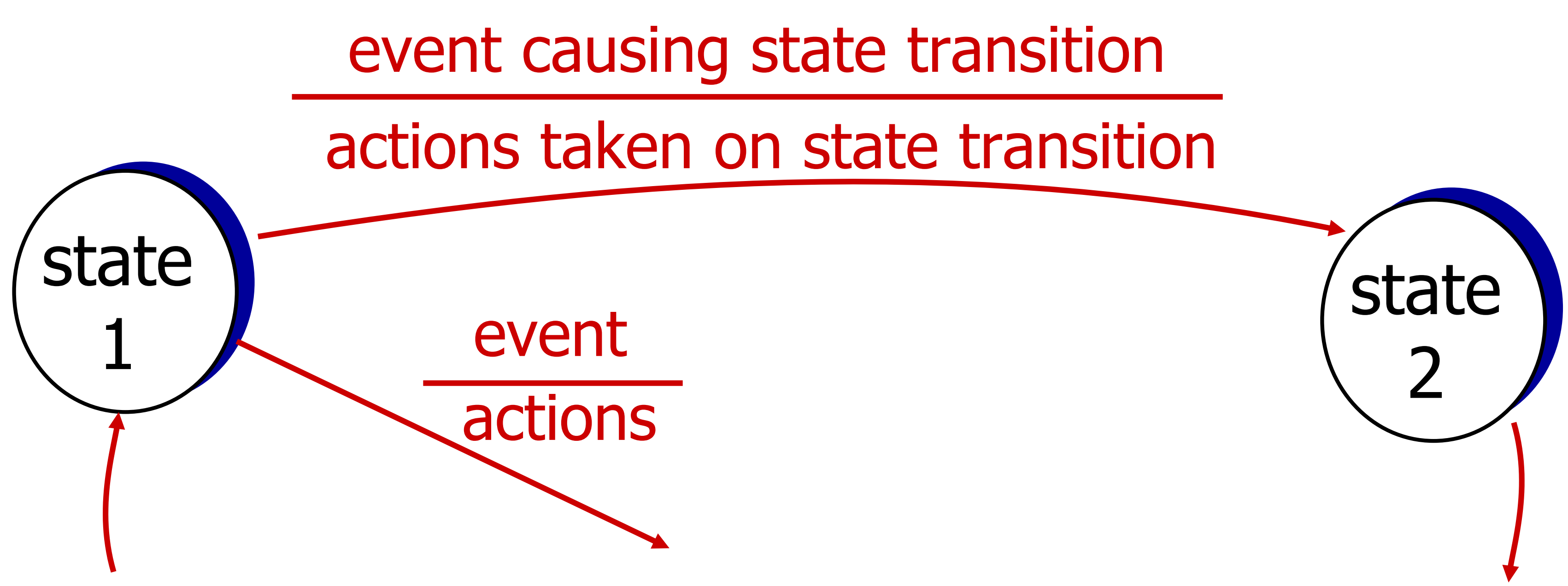
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

we'll:

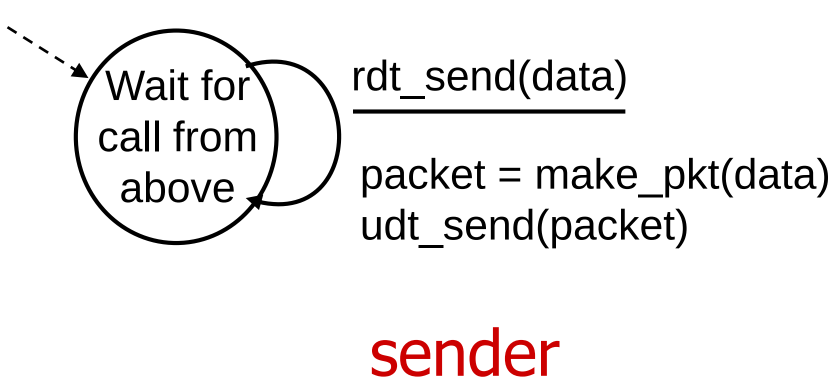
- **incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)**
- **consider only unidirectional data transfer**
  - but control info will flow on both directions!
- **use finite state machines (FSM) to specify sender, receiver**

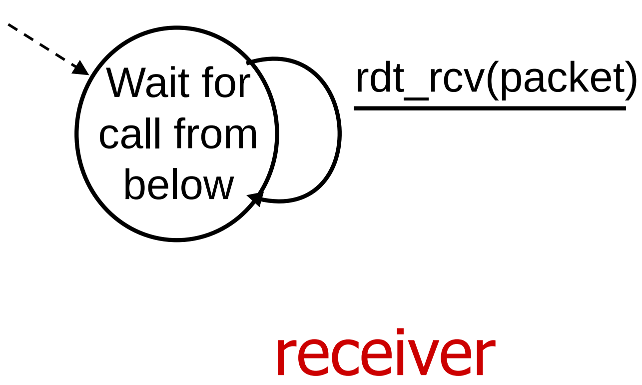
state: when in this "state" next state uniquely determined by next event



RDT 1.0: Reliable Transfer Over a Reliable Channel

- **Underlying data transmission channel is perfectly reliable**
  - No bit errors
  - No loss of packets
- **We don't have networks like this, but it's a good place to start**
- **separate FSMs for sender, receiver:**
  - sender sends data into underlying channel
  - receiver reads data from underlying channel





RDT 2.0: Data 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?

RDT 2.0: Data Channel With Bit Errors

- **Underlying data channel may flip bits in packet**
  - Use a checksum to detect bit errors
- **Question: How should system recover from these errors?**
  - Acknowledgements (ACKs): receiver explicitly tells sender that a packet is received OK
  - Negative acknowledgements (NAKs): receiver explicitly tells sender that a packet had errors when it was received
    - Sender retransmits packet on receipt of a NAK
- **New mechanisms in RDT 2.0 (beyond RDT 1.0):**
  - Error detection
  - Feedback: send control messages (ACK, NAK) from receiver to sender

A Fatal Flaw in RDT 2.0, on to RDT 2.1

- **What happens if ACK/NAK messages are corrupted?**
  - Sender doesn't know if receiver correctly received the data
    - Data may have been corrupt on the way to receiver
    - ACK/NAK may have been corrupt on the way back to sender
  - Can't just retransmit since receiver may receive duplicate data!
- **Handling duplicates:**
  - Sender retransmits current packet if ACK/NAK is corrupted
  - Sender adds a sequence number to each packet
  - Receiver discards duplicate packets at Transport Layer
    - Those packets are not delivered up to the Application Layer

RDT 2.1: Discussion

- **Sender:**
  - Sequence number added to packets
  - Two sequence numbers (0,1) will suffice
  - Must check if received ACK/NAK corrupted
  - Sender must “remember” if it should be expecting a sequence number of 0 or 1
- **Receiver:**
  - Must check if received packet is duplicate
    - Receiver must “remember” if it should be expecting a sequence number of 0 or 1
  - Note: the receiver does not know if its last ACK/NAK message was received OK at sender

RDT 2.2: Eliminating the NAK Messages

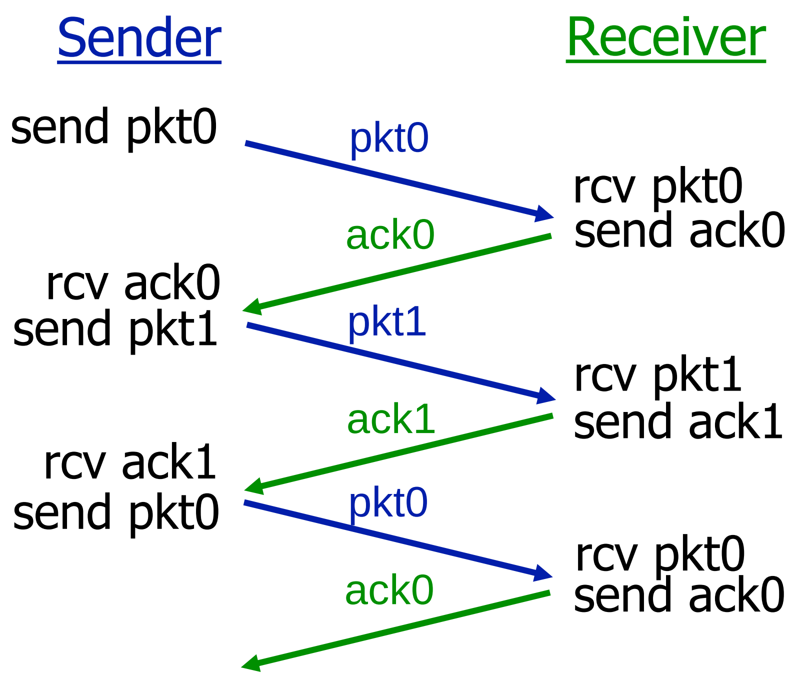
- **Possible to achieve same functionality as RDT 2.1 using ACKs messages only**
- **Receiver sends ACK messages for last packet that was received correctly**
  - No message is sent for a packet this is received with errors
- **Duplicate ACK messages received at sender results in same action as NAK from RDT 2.1 -- retransmit packet**
  - Duplicate ACK message would be detected at sender by receiving two consecutive ACK_0 messages or two consecutive ACK_0 messages

RDT 3.0: Channels with Bit Errors & Packet Loss

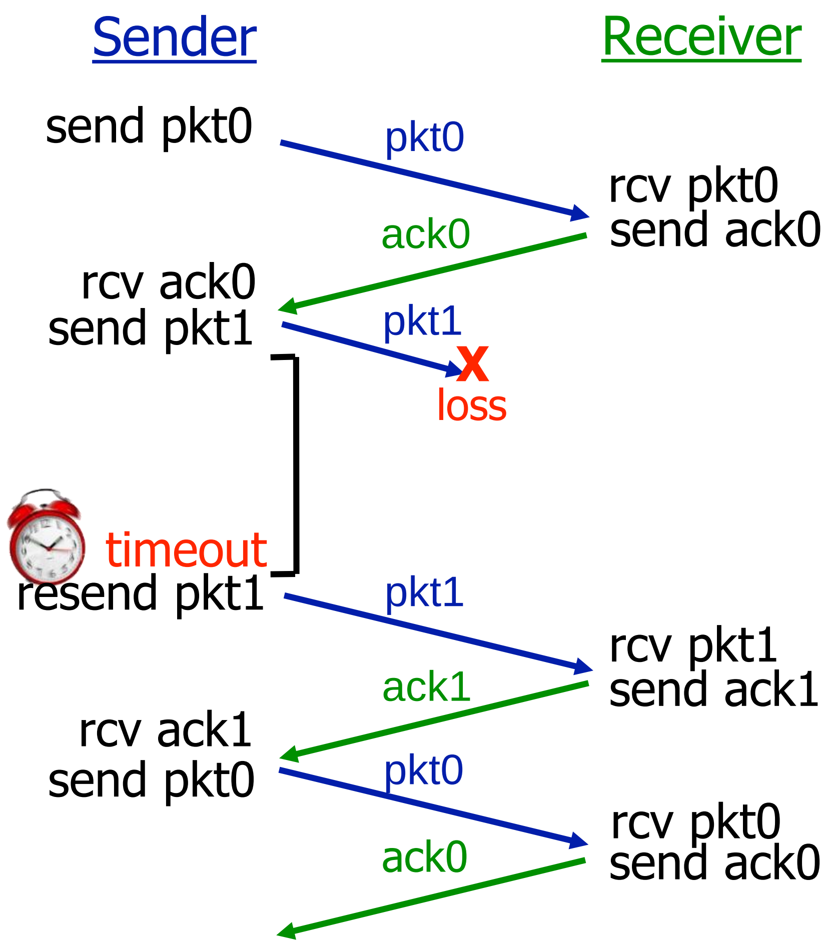
- **New assumption: underlying channel can also lose packets**
  - Can lose data packets or ACK messages
  - Checksum, sequence number, ACKs, retransmissions will be of help … but not enough

- **Approach: sender waits a "reasonable" amount of time for an ACK**
  - Retransmits packet if no ACK is received in this time
  - If packet (or ACK) is just delayed and not lost:
    - Retransmission will result in duplicate data, but sequence numbers from RDT 2.2. already handle this issue
    - Receiver must specify sequence number of packet being ACKed
  - Requires countdown timer

RDT 3.0 in Action with Packet Loss

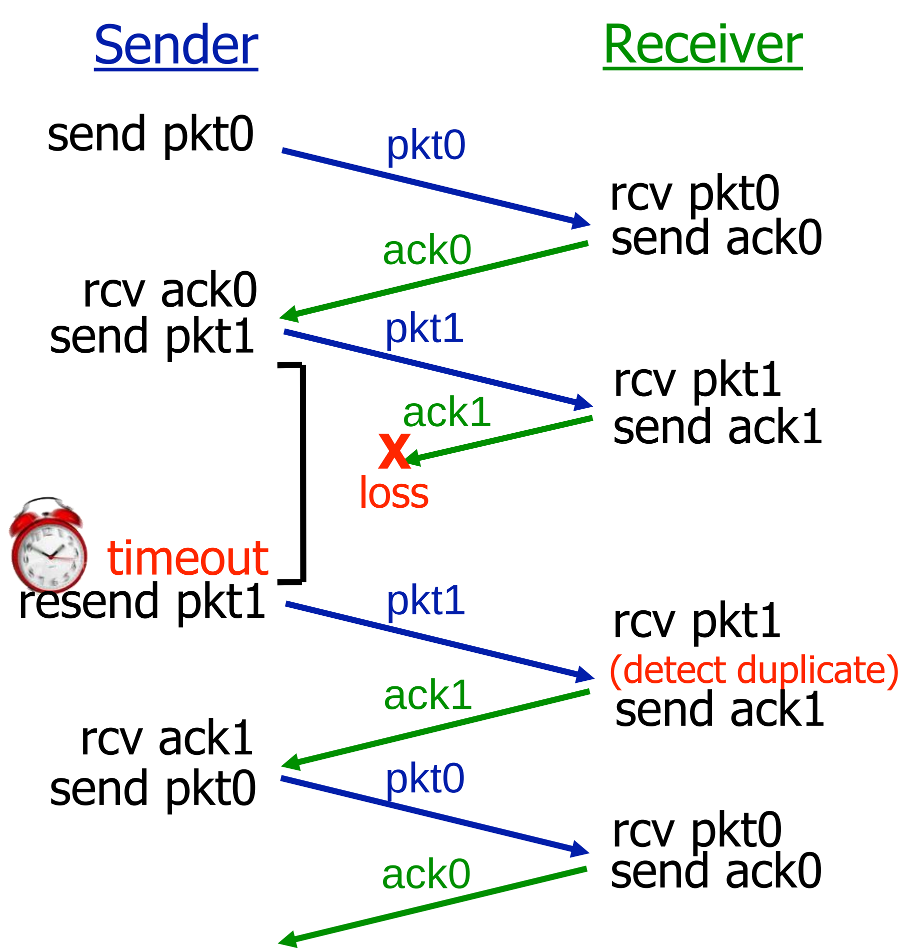


No Packet Loss

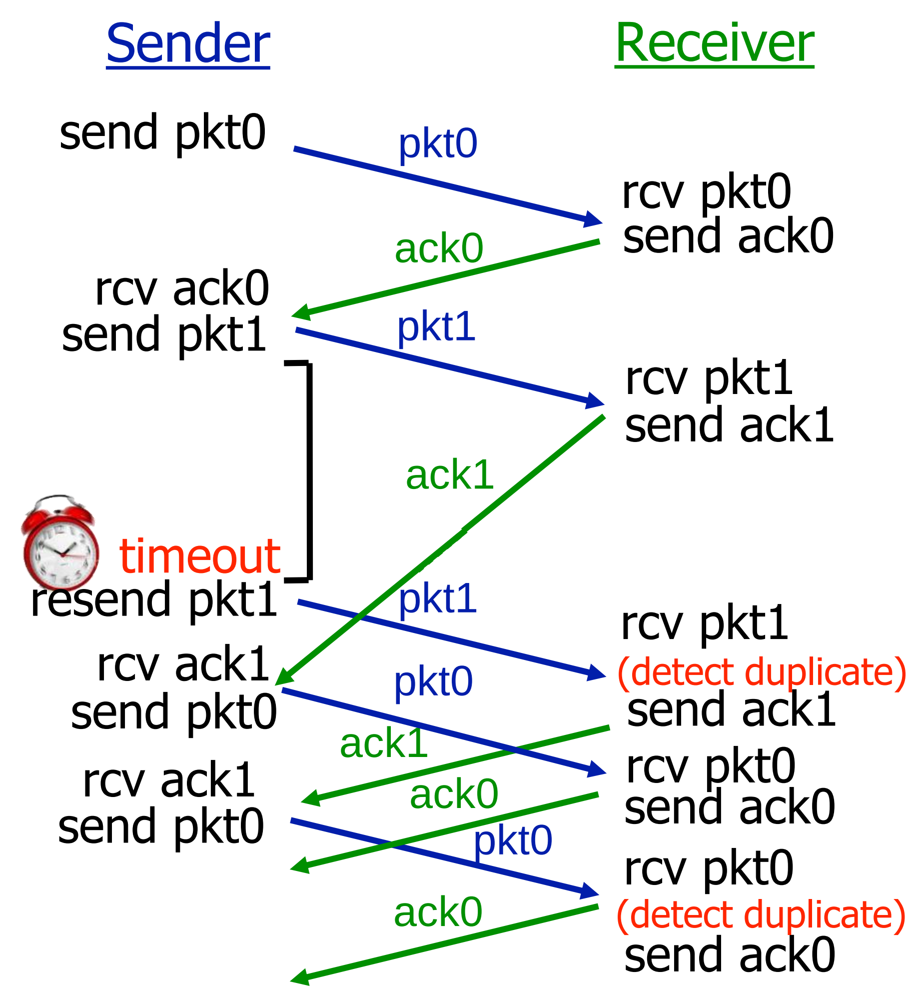


With Packet Loss

RDT 3.0 in Action with Lost/Delayed ACK



ACK Loss



Premature Timeout/Delayed ACK

Performance of RDT 3.0

- **RDT 3.0 will work reliably, but would have terrible performance**
  - RDT 3.0 utilizes a Stop-and-Wait protocol
    - Sender sends one packet, then waits for receiver response before sending the next packet

- **Example: Assume a system sending 8000 bit packets on a 1 Gbps link, where there is a 15 ms propagation delay between the sender and the receiver**

Time for sender to transmit data

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

Performance of RDT 3.0 (Cont.)

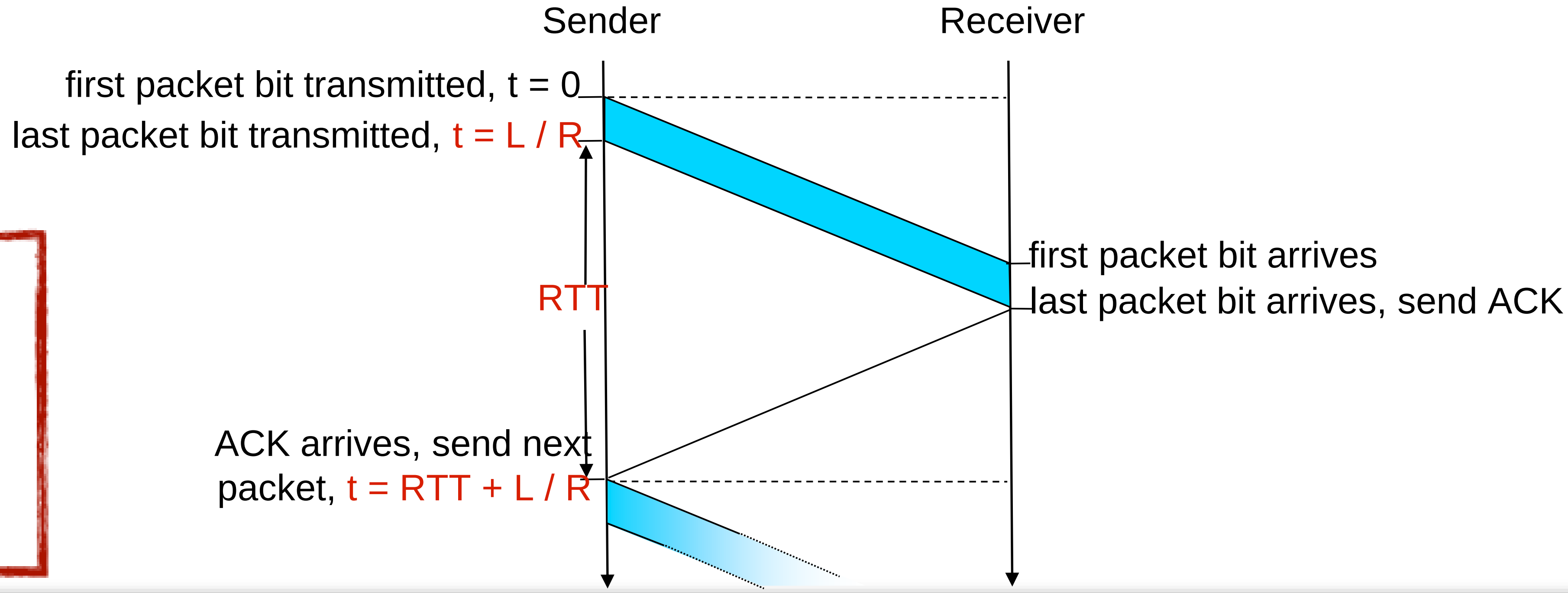
- **If Round-Trip Time (RTT) is 30 ms:**
  - Sender is only sending 8 microseconds
  - Can only send a new packet every 30.008 microseconds
  - Effectively makes 1 Gbps link run at ~270 Kbps!!!

Utilization: Fraction of time the sender is busy sending

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

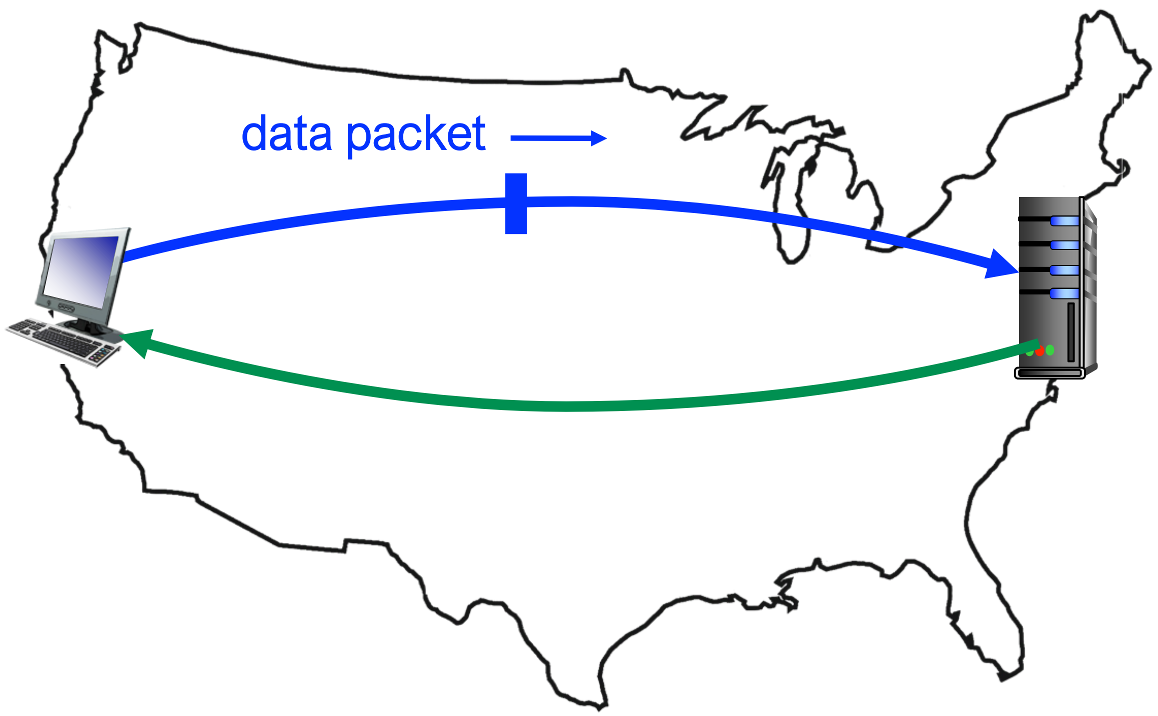
.027%

The network protocol limits use of physical resources!

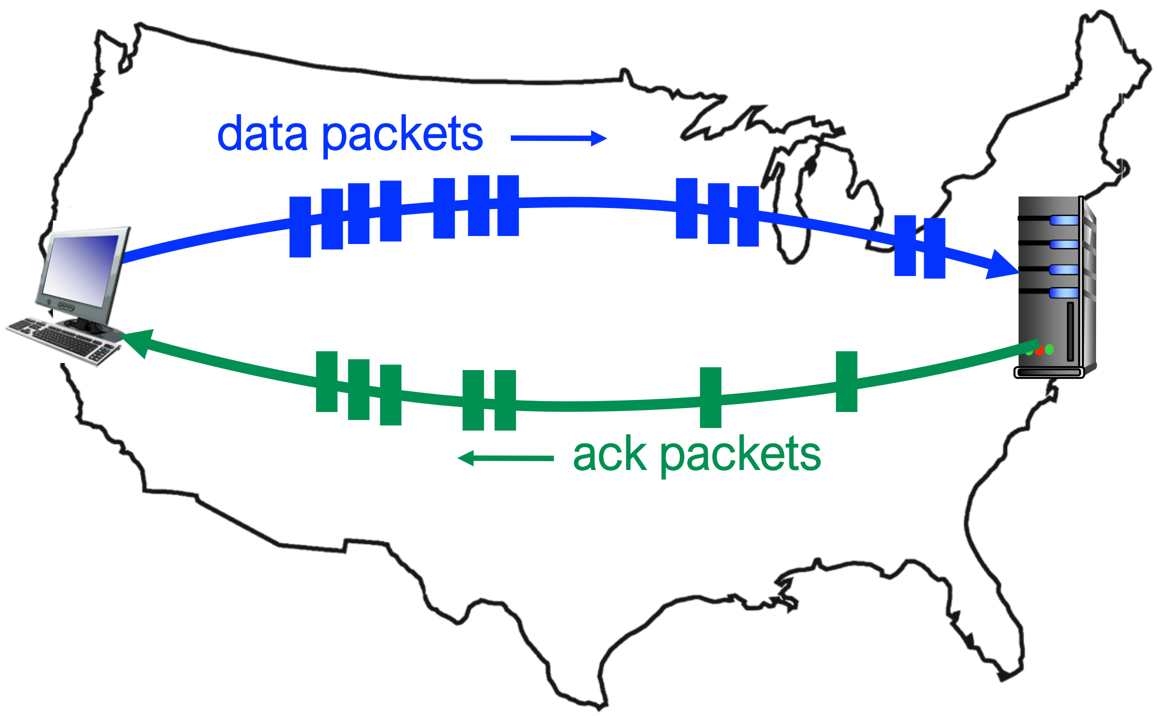


Pipelined Protocols

- **Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets**
  - Range of sequence numbers must be increased (0 and 1 will no longer suffice)
  - Buffering at sender and/or receiver is required
- **Two generic forms of pipelined protocols: Go-Back-N, and Selective Repeat**

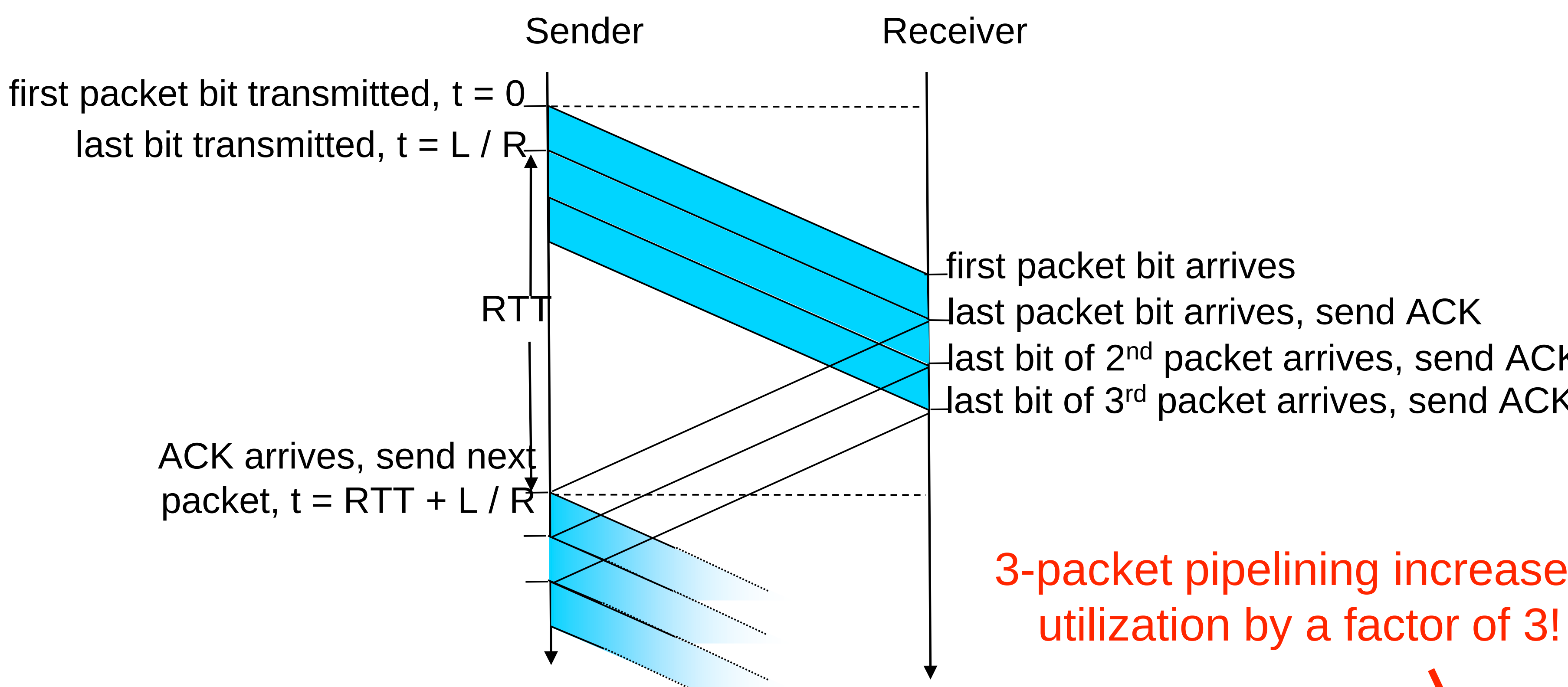


Stop-and-Wait Protocol



Go-Back-N Protocol

Pipelining: Increased Utilization



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

Utilization: Fraction of time the sender is busy sending

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.024}{30.008} = 0.0008$$

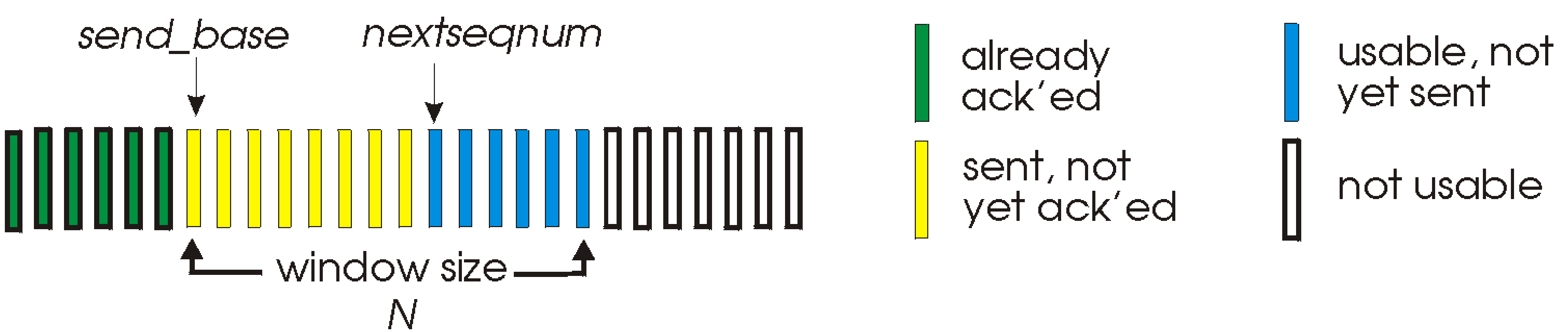
Pipelined Protocols: Overview

- **Go-back-N:**
  - Sender can have up to N unacked packets in pipeline
  - Receiver only sends cumulative acks
    - Doesn't ack a packet if there is a gap
  - Sender has a timer for the oldest unacked packet
    - When timer expires, retransmit ***all*** unacked packets

- **Selective Repeat:**
  - Sender can have up to N unacked packets in pipeline
  - Receiver sends individual acks for each packet
  - Sender maintains timer for each unacked packet
    - When timer expires, retransmit only that unacked packet

Go-Back-N: Sender Side

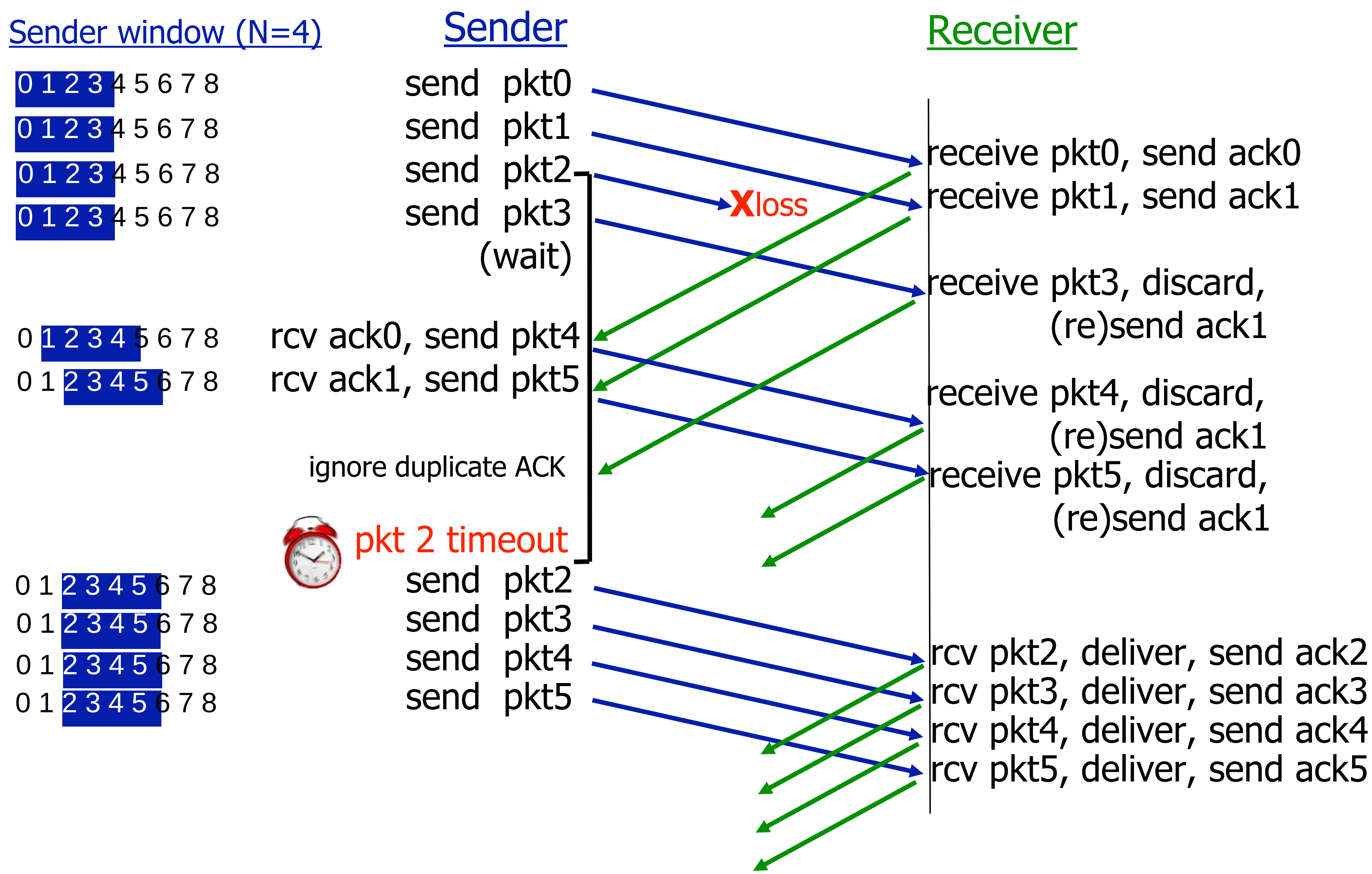
- **Sender can transmit multiple packets without waiting for an ack**
  - A "sliding window" of up to N consecutive unacked packets allowed
- **Include a k-bit sequence number in the packet header**
- **ACK(n): ACKs all packets up to and including sequence number n**
  - Cumulative ACK
  - May receive duplicate ACKs
- **Maintain timer for oldest in-flight packet**
- **Timeout(n): retransmit packet n and all higher sequence number packets in window**



Go-Back-N: Receiver

- **Send ACK message for correctly-received packet with highest in-order sequence number**
  - May generate duplicate ACKs
  - Need only remember expected sequence number
  - Must receive packets in-order to send ACK
- **Out-of-order packets:**
  - Discard (don't buffer) out-of-order packets (no receiver buffering)
    - Yes, even if they are correctly formatted and error-free
    - Will be retransmitted anyway based on sender's rules
  - Re-ACK packet with highest in-order sequence number

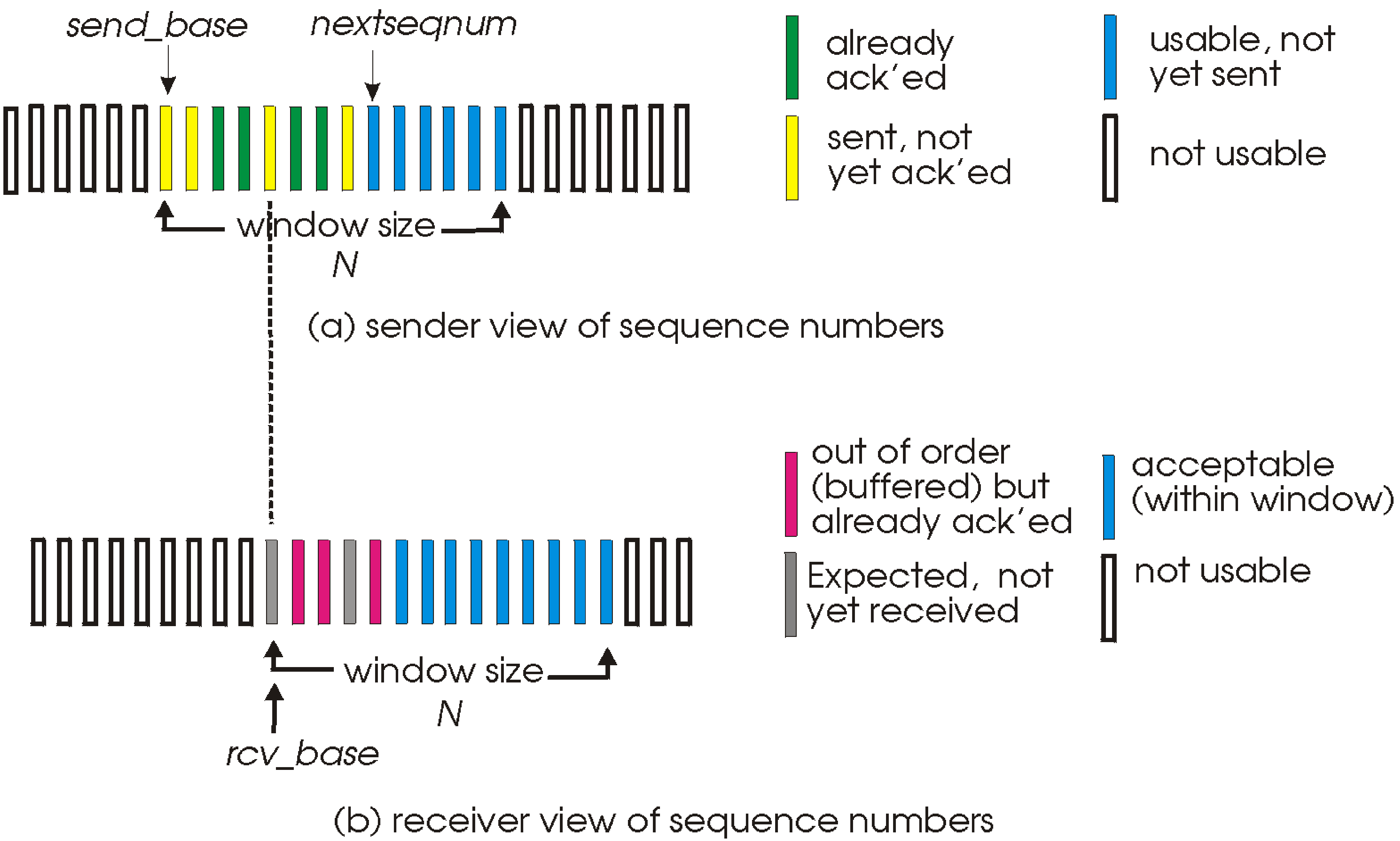
Go-Back-N in Action



Selective Repeat

- **Receiver individually acknowledges all correctly received packets**
  - Out-of-order packets are buffered at the receiver
  - Buffered packets are eventual delivered in-order to upper layer
- **Avoids unnecessary retransmission -- sender only resends packets for which an ACK was not received**
  - Sender has separate timer for each unACKed packet
- **Sender window**
  - N consecutive sequence numbers
  - Limits sequences numbers of sent, unACKed packets

Selective Repeat: Sender/Receiver Windows

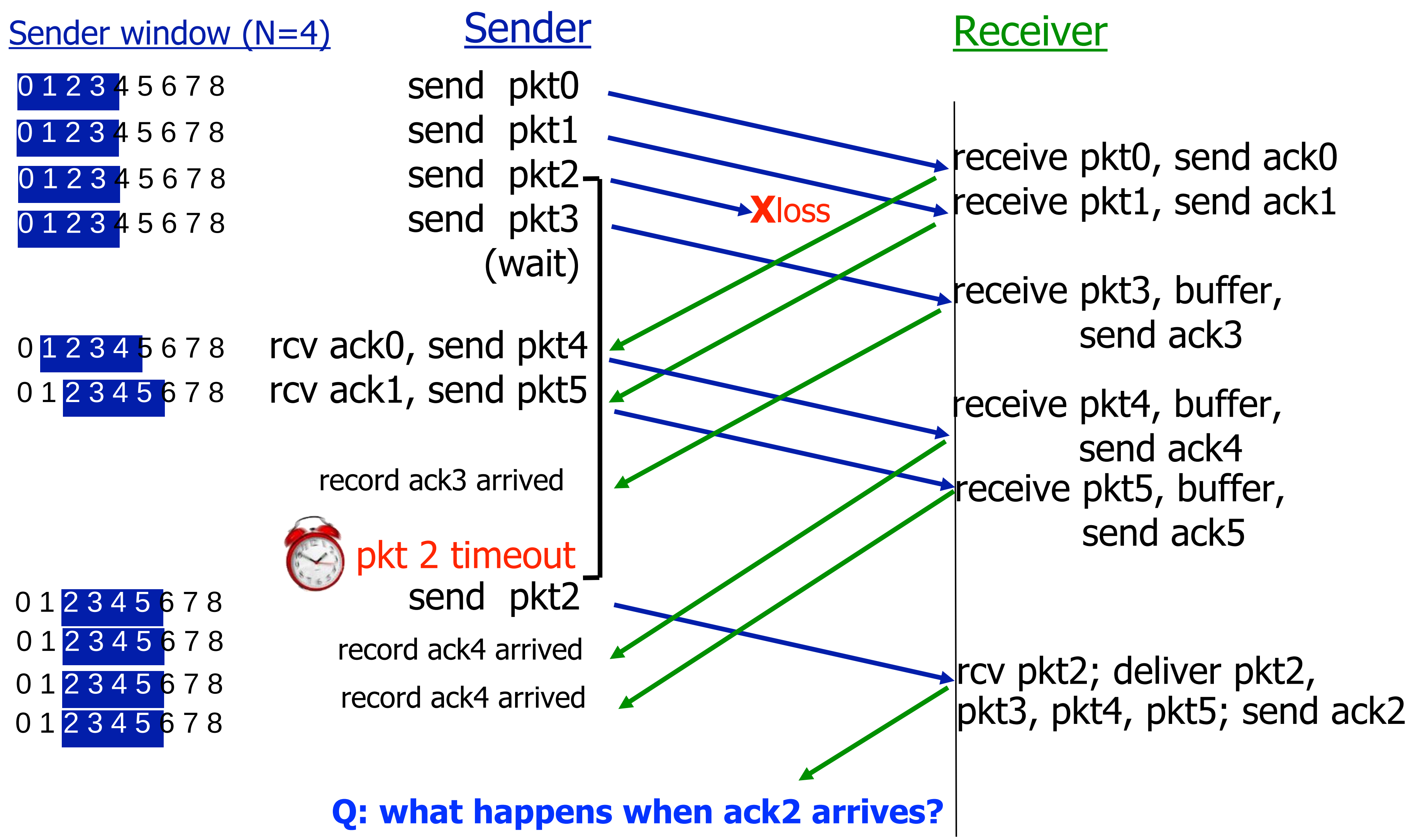


Selective Repeat

- **Sender**
  - Receives data from upper layer:
    - If next available sequence number is in the sliding window, send the packet
  - A timeout occurs for packet n:
    - Resend packet n, restart timer
  - ACK(n) received for packet in current window:
    - Mark packet n as received
    - If n is smallest unACKed packet in window, advance the window base to next unACKed sequence number

- **Receiver**
  - Receives packet n in receivers window
    - Send ACK(n)
    - If out-of-order, buffer
    - If in-order, deliver with other buffered, in-order packets to the upper layer. Also, advance window to next not-yet-received packet
  - Receives packet n that has already been seen and ACKed by the receiver
    - Send ACK(n) again
  - Otherwise:
    - Ignore the packet

Selective Repeat in Action



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)?