# Reliable Data Delivery

Lecture 13

http://www.cs.rutgers.edu/~sn624/352-F24

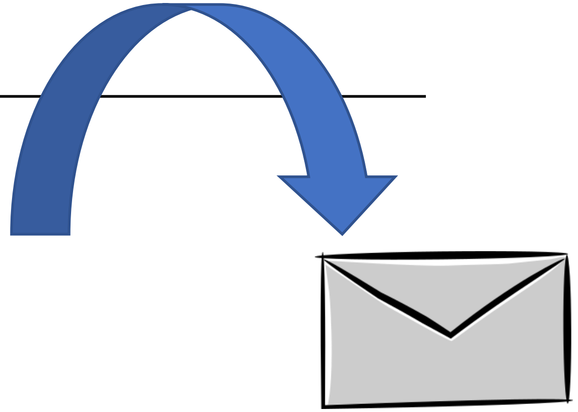
Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

# Review

- UDP: best-effort delivery + demultiplexing + error detection
- Checksum function: 1s complement of the 1s complement sum
- Sender: compute checksum & write.
  - Receiver: compute checksum, compare to 0

| | | |
|---|---|---|
| | | 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 |
| | | 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 |
| wraparound | 1 | 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 |
| sum | | 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 |
| checksum | | 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1 |



https://datatracker.ietf.org/doc/html/rfc1071

Not on the exam

# Checksum, why you being weird?

- Need a function that is fast to compute, catches likely errors, and easy to verify. Some design considerations:
- Basic bit-wise: AND, OR: many inputs map to the same output
- XOR: can catch single bit-flips, but not an even number of 1s/0s flipping
  - Some sort of addition is preferable to this (carries will show errors)
- Addition is commutative, associative, has an identity element (0), is efficient to calculate
  - Checksum can appear anywhere in the packet
  - Compute checksum by placing a 0 in place originally
  - Use operations at the natural bit-width of the machine (16 bits was common)
- (Regular) two's complement addition: errors in higher order bit positions can be missed (the final carry-out bit isn't part of the checksum)
  - One's complement: adding the final carry-out to the result helps ☺
- Why complement? Why not compare the checksum rather than to 0?
  - CPUs have ways of detecting if the last result was 0

# Some observations on checksums

- Checksums don't detect all bit errors
  - Consider (x, y) vs. (x – 1, y + 1) as adjacent 16-bit values in packet
  - Analogy: you can't assume the package hasn't been meddled with if its weight matches the one on the stamp. More smarts needed for that. ☺
  - But it's a lightweight method that works well in many cases

- Checksums are part of the packet; they can get corrupted too
  - The receiver will just declare an error if it finds an error

# Some observations on checksums

- Checksums are insufficient for reliable data delivery
  - If a packet is lost, so is its checksum
- UDP and TCP use the same checksum function
  - TCP also uses the lightweight error detection capability
  - However, TCP has more mature mechanisms for reliable data delivery (up next!)
- Checksum is a mechanism to detect errors, not correct them
  - Even when they detect errors, checksums don’t tell you where they lie

# Playing with checksums

- Let’s craft some UDP packets (again)!

- `sudo tcpdump -i lo udp –XAvvv # observe packets`
- `sudo scapy # tool used to send crafted packets`
- `send(IP(dst="127.0.0.1")/UDP(sport=1024, dport=2048)/"hello world”, iface="lo")`

- Now can you craft two UDP packets with an identical checksum?

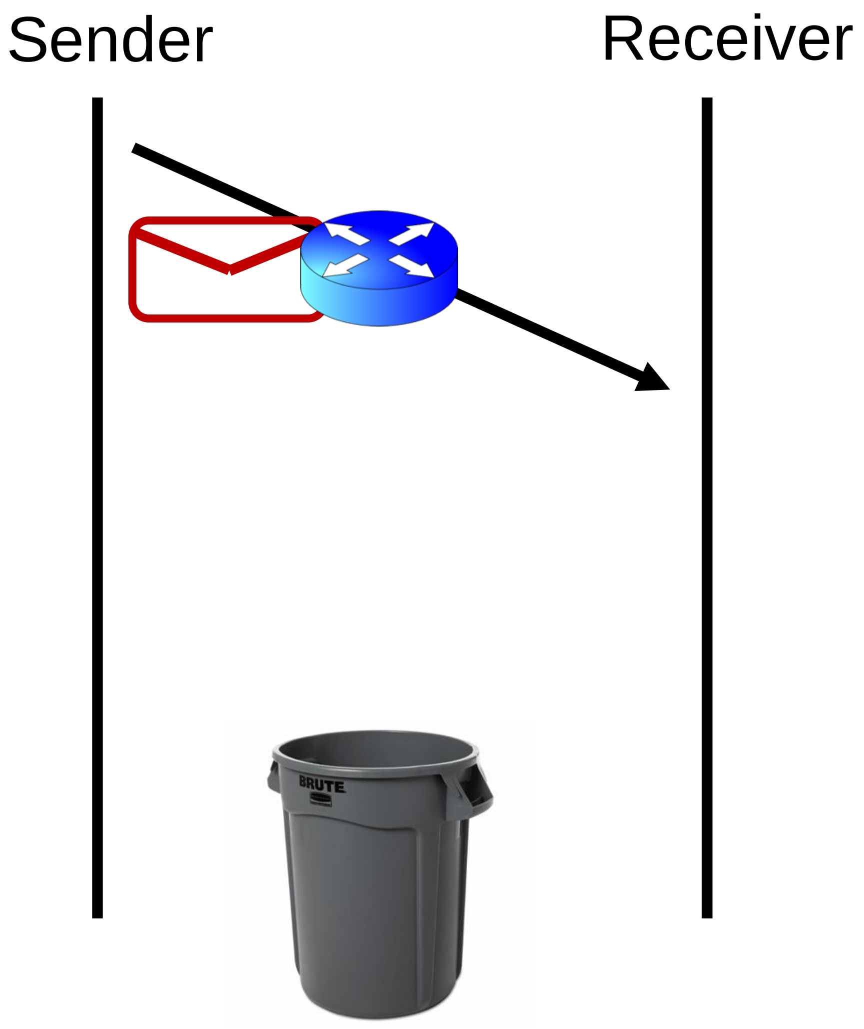
# Summary of UDP

- A simple transport: Send or receive a single packet from/to the correct application process. That's it
  - Just a thin shim around network layer's best-effort delivery
  - No connection building, no latency
  - Suitable for one-off request/response messages
  - Sometimes suitable for loss-tolerant but delay-sensitive applications

- No reliability, performance, or ordering guarantees
- Can do basic error detection (bit flips) using checksums
  - Error detection is necessary to deliver data reliably, but it is insufficient

# Reliable data delivery

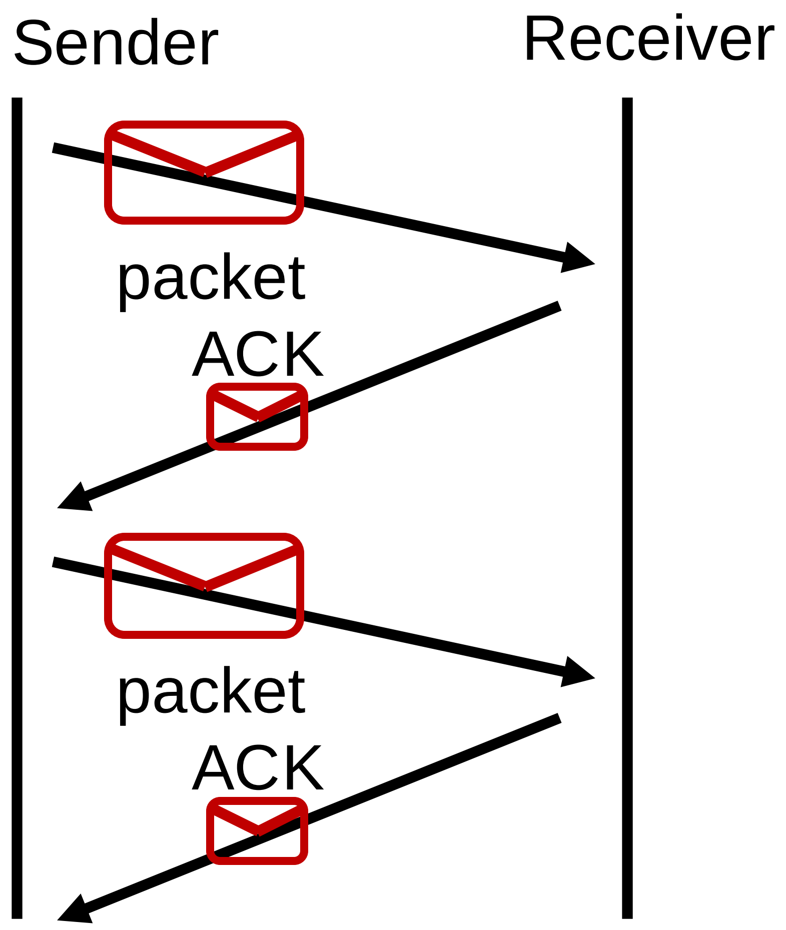
# Packet loss




- How might a sender and receiver ensure that data is delivered reliably (despite some packets being lost)?
- TCP uses three mechanisms

# Coping with packet loss: (1) ACK

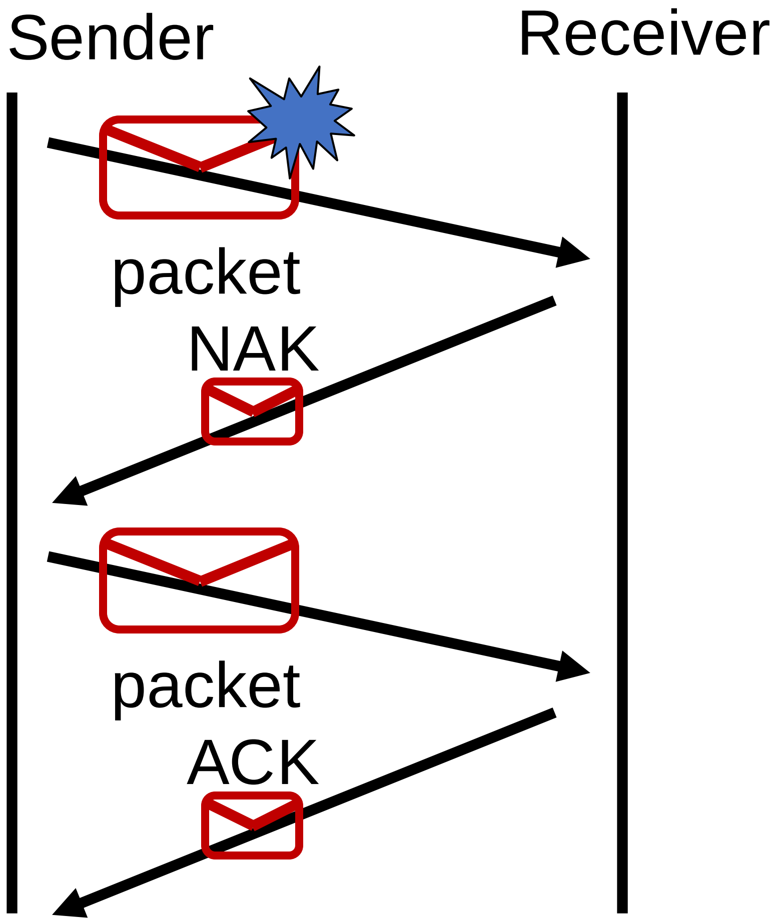
- Key idea: Receiver returns an acknowledgment (ACK) per packet sent
- If sender receives an ACK, it knows that the receiver got the packet.



# Coping with packet corruption: (1) ACK

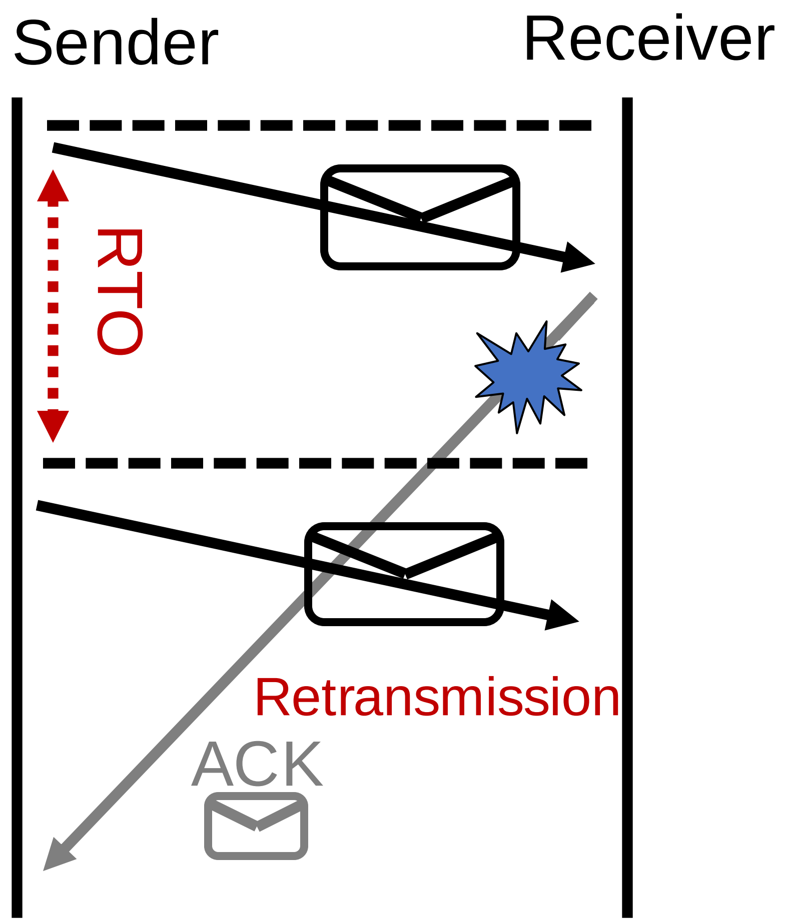
- ACKs also work to detect packet corruption on the way to the receiver
  - One possibility: A receiver could send a negative acknowledgment, or a NAK, if it receives a corrupted packet
  - Q: How to detect corrupted packet?
    - One method: Checksum!

- TCP only uses positive ACKs.



# Coping with packet loss: (2) RTO

- What if a packet is dropped?
- Key idea: Wait for a duration of time (called retransmission timeout or RTO) before re-sending the packet
- In TCP, the onus is on the sender to retransmit lost data when ACKs are not received
- Note that retransmission works also if ACKs are lost or delayed

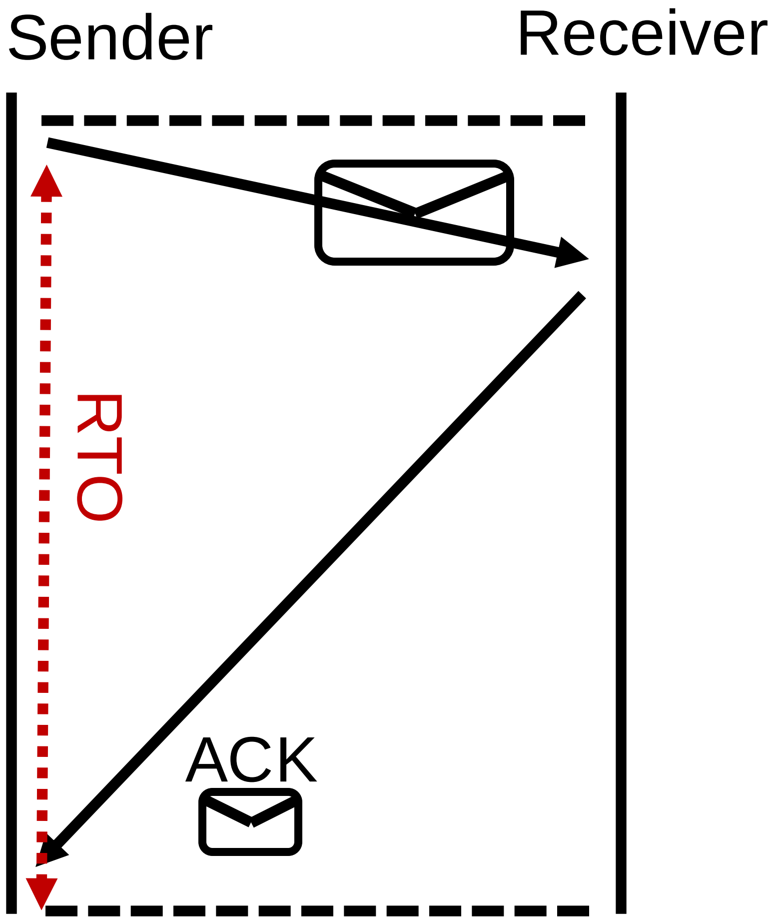


# How should the RTO be set?

- A good RTO must predict the round-trip time (RTT) between the sender and receiver
  - RTT: the time to send a single packet and receive a (corresponding) single ACK at the sender

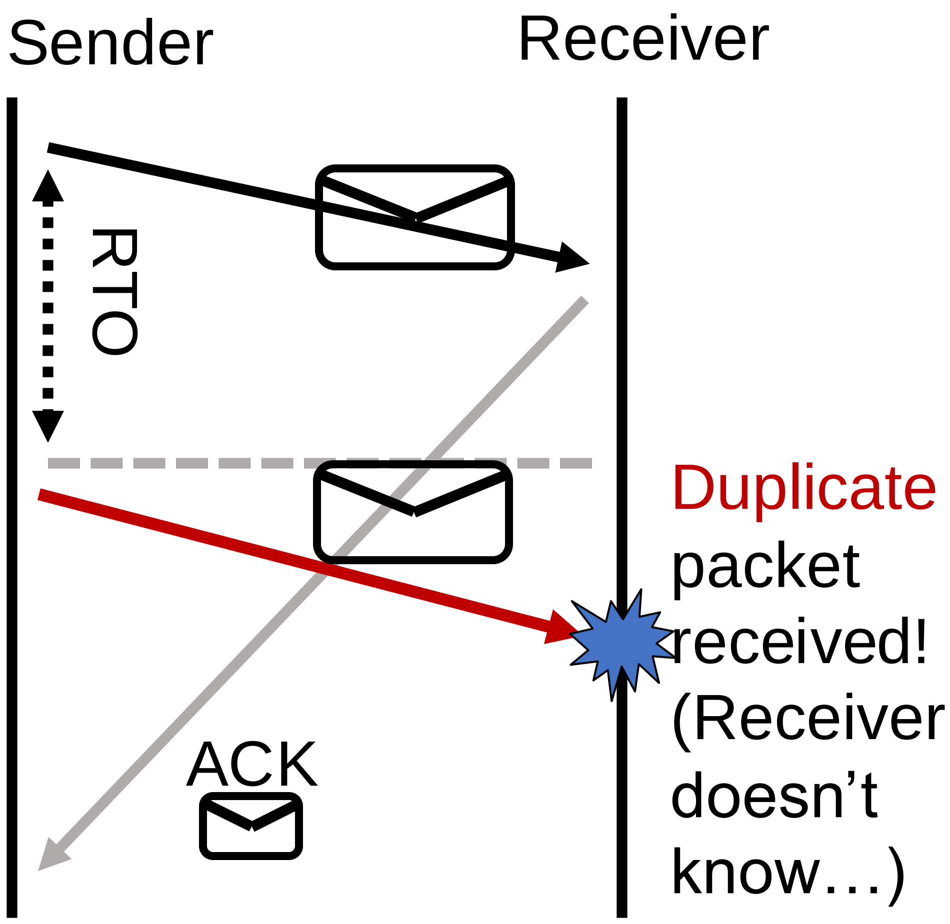
- Intuition: If an ACK hasn't returned, and our (best estimate of) RTT has elapsed, the packet was likely dropped.

- RTT can be measured directly at the sender. No receiver or router help needed.



# Coping with packet duplication

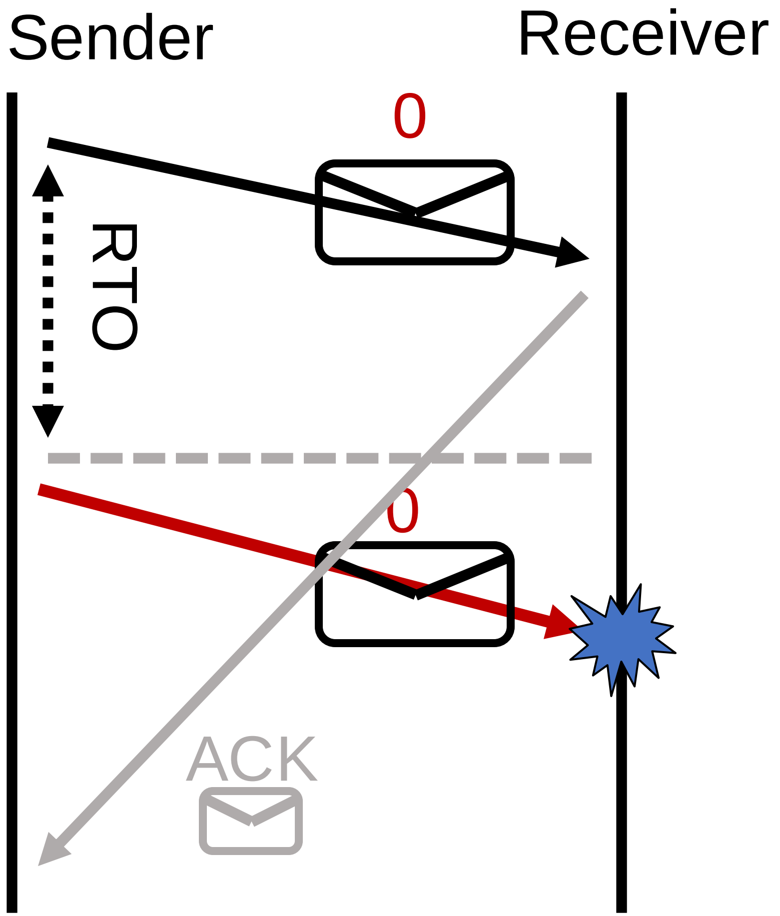
- If ACKs delayed beyond the RTO, sender may retransmit the same data
  - Receiver wouldn't know that it just received duplicate data from this retransmitted packet
- Add some identification to each packet to help distinguish between adjacent transmissions
  - This is known as the sequence number



# Coping with packet loss: (3) Sequence #s

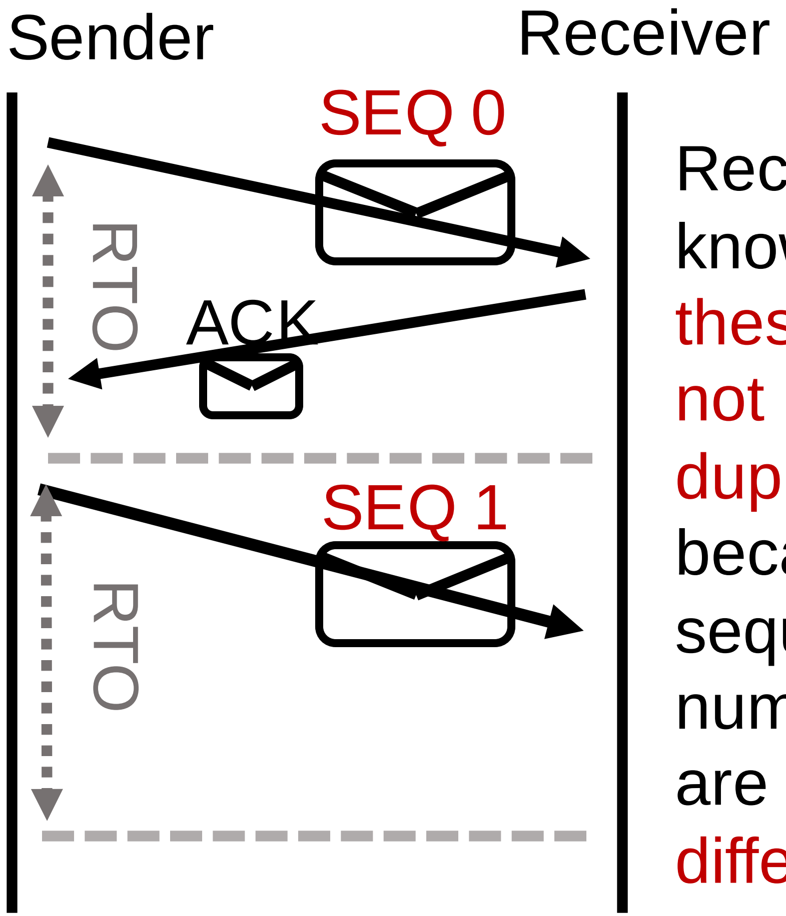
- A bad scenario: Suppose an ACK was delayed beyond the RTO; sender ended up retransmitting the packet.

- At the receiver: sequence number helps disambiguate a fresh transmission from a retransmission
    - Sequence number same as earlier: retransmission
    - Fresh sequence number: fresh data



# Coping with packet loss: (3) Sequence #s

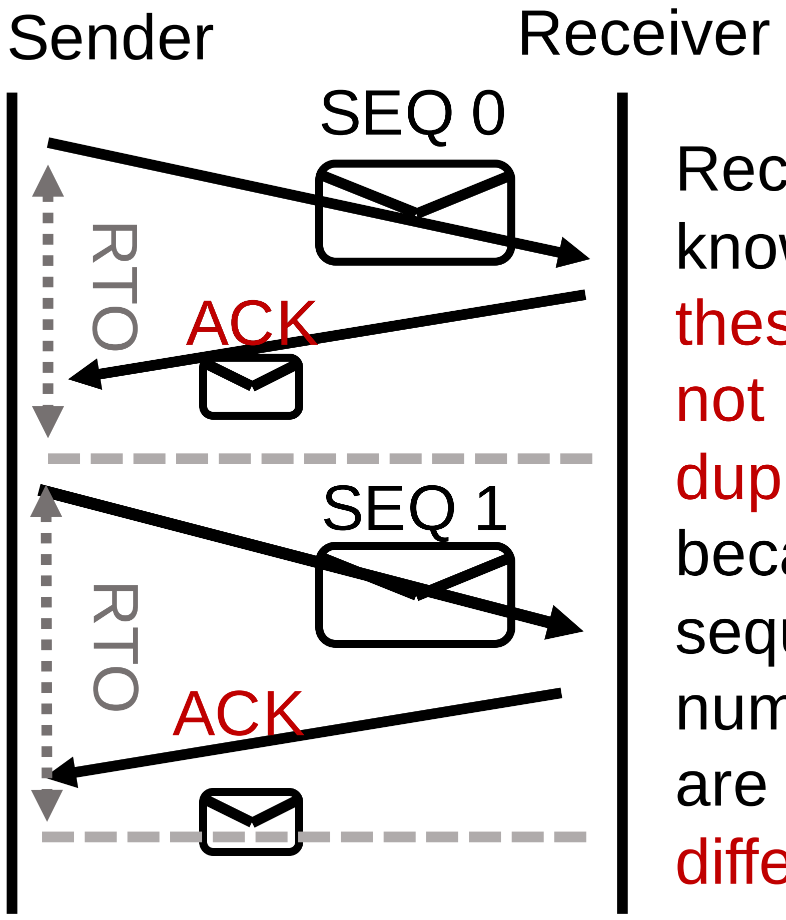
- A good scenario: packet successfully received and ACK returned within RTO
- Sequence numbers of successively transmitted packets are different




Receiver knows these are not duplicate, because sequence numbers are different

# Coping with packet loss: (3) Sequence #s

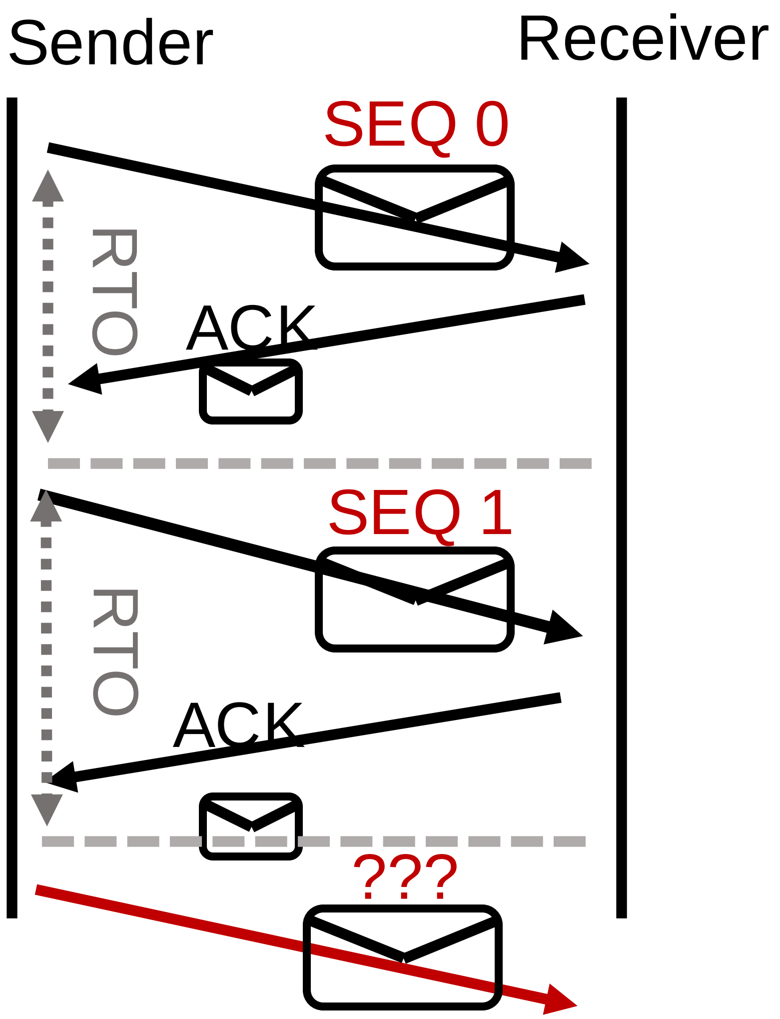
- A good scenario: packet successfully received and ACK returned within RTO
- Sequence numbers of successively transmitted packets are different




Receiver knows these are not duplicate, because sequence numbers are different

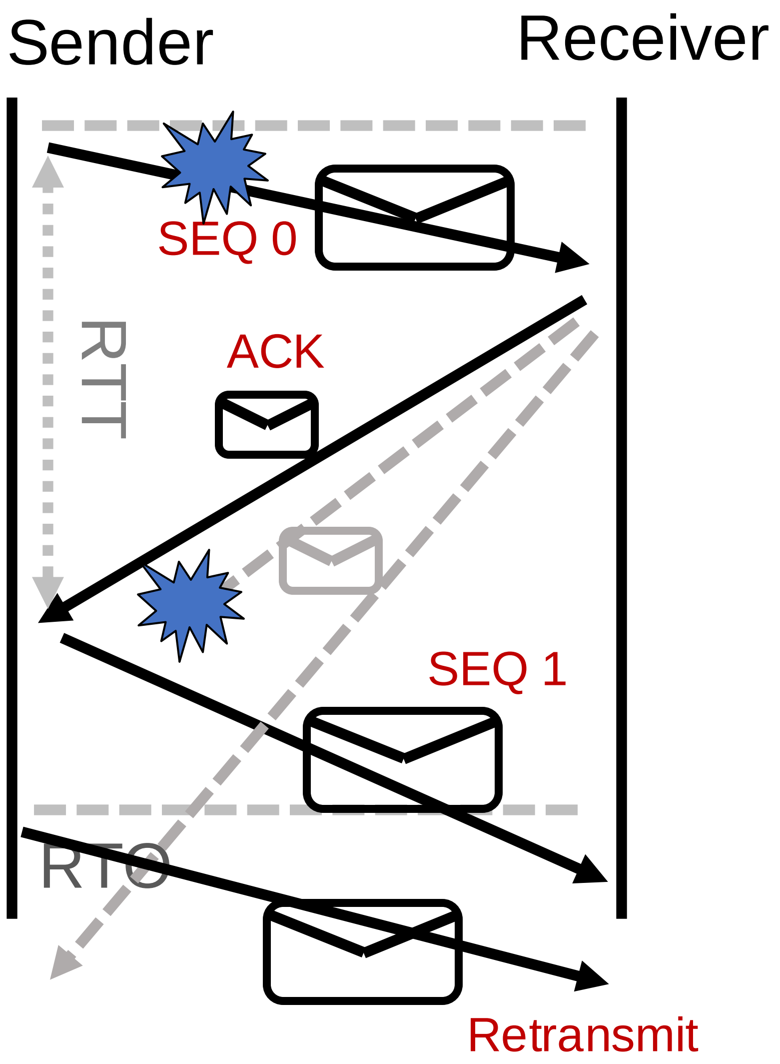
# Q: What is the seq# of third packet?

- Goal: Avoid ambiguity on which packet was received/ACK'ed from both the sender and receiver's perspective
- One option: increment seq#: 2, 3, …
- Alternative: since seq # 0 was successfully ACK'ed earlier, it is OK to reuse seq #0 for next transmission.
- Seq #s reusable if older packets with those seq #s known to be delivered



# Stop-and-Wait Reliability

- Sender sends a single packet, then waits for an ACK to know the packet was successfully received. Then the sender transmits the next packet.
- If ACK is not received until a timeout (RTO), sender retransmits the packet
- Disambiguate duplicate vs. fresh packets using sequence numbers that change on “adjacent” packets



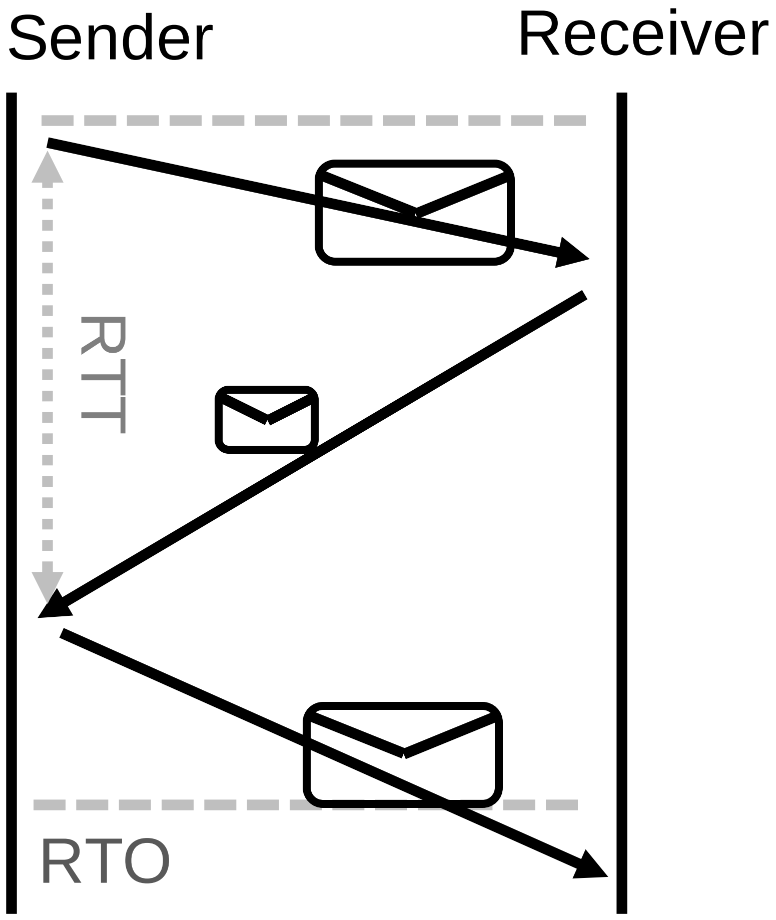
In principle, these three ideas are sufficient to implement reliable data delivery!

# Efficiency problem with stop-and-wait

- Sender sends one packet, waits for an ACK (or RTO) before transmitting next one
  - Unfortunately, too slow ☹

- Suppose RTO = RTT = 100 milliseconds
- Packet size (bytes in 1 packet) = 12,000 bits
- Bandwidth of links from sender to receiver = 12 Mbit/s (1 M = $10^6$)

- Rate of data transfer = data size / time

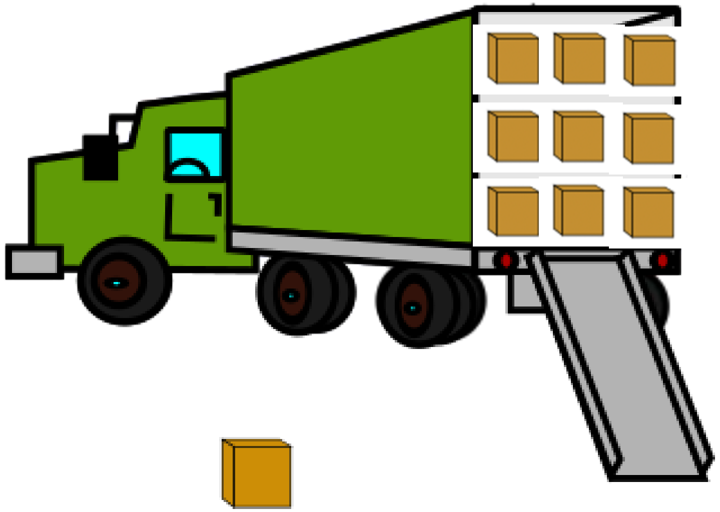



120 Kilobit/s == 1% of bw!

Sending one packet per RTT makes the data transfer rate limited by the time between the endpoints, rather than the bandwidth.

Ensure you got the (one) box safely; make N trips

Ensure you get N boxes safely; make just 1 trip!



Keep many packets in flight