Error Detection

Lecture 12

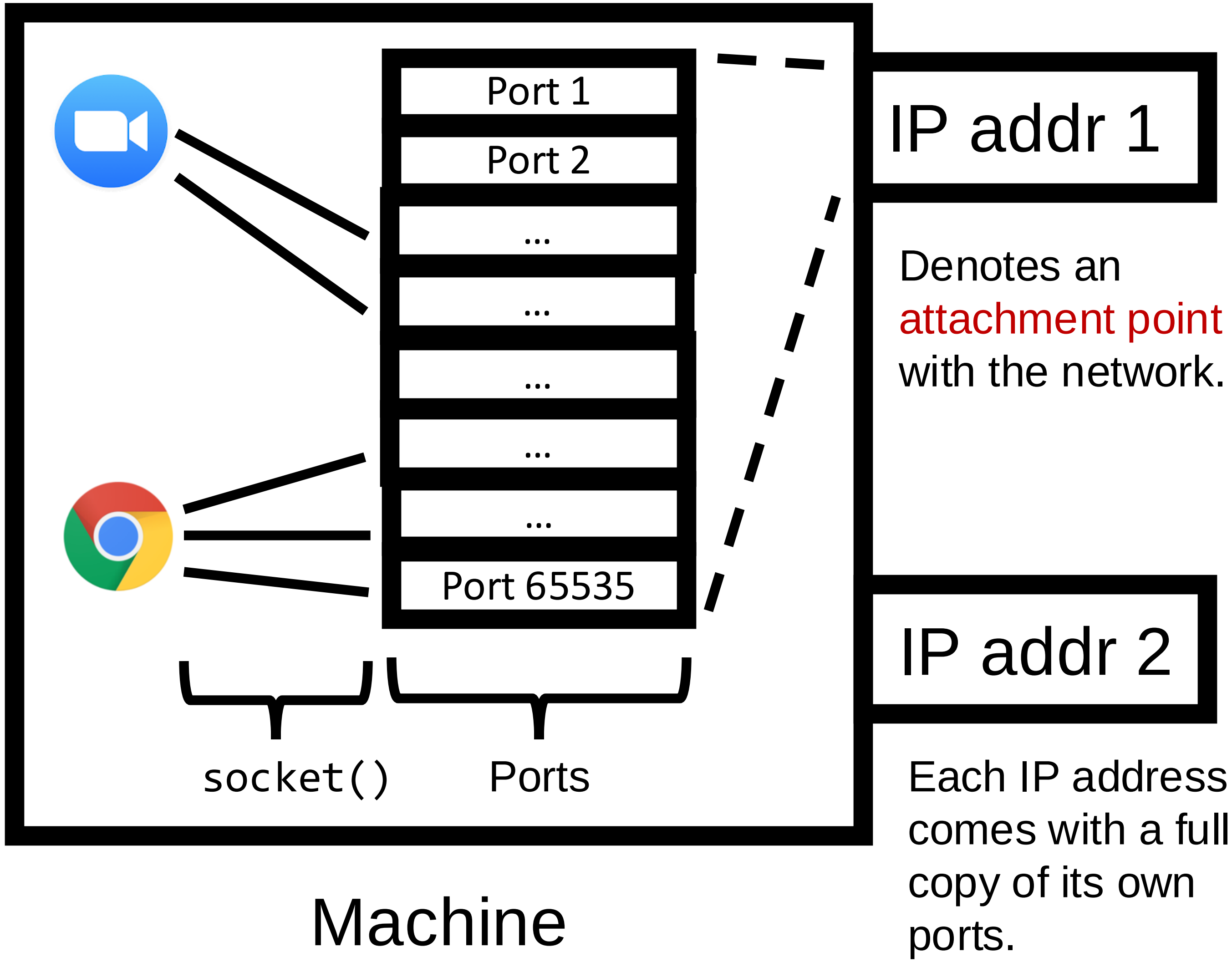
http://www.cs.rutgers.edu/~sn624/352-F24

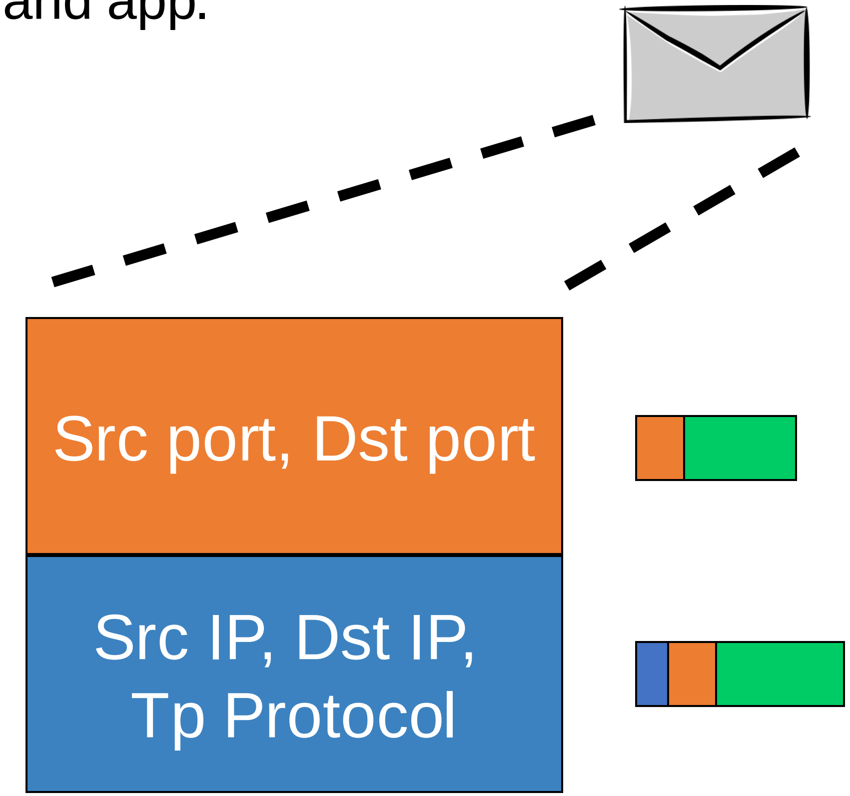
Srinivas Narayana

RUTGERS
UNIVERSITY | NEW BRUNSWICK

Review: Demultiplexing

Connection lookup: The operating system does a lookup using these data to determine the right socket and app.





Listing sockets and connections

- `ss`
- `iperf –s` and `iperf –s -u`

User Datagram Protocol

UDP: User Datagram Protocol [RFC 768]

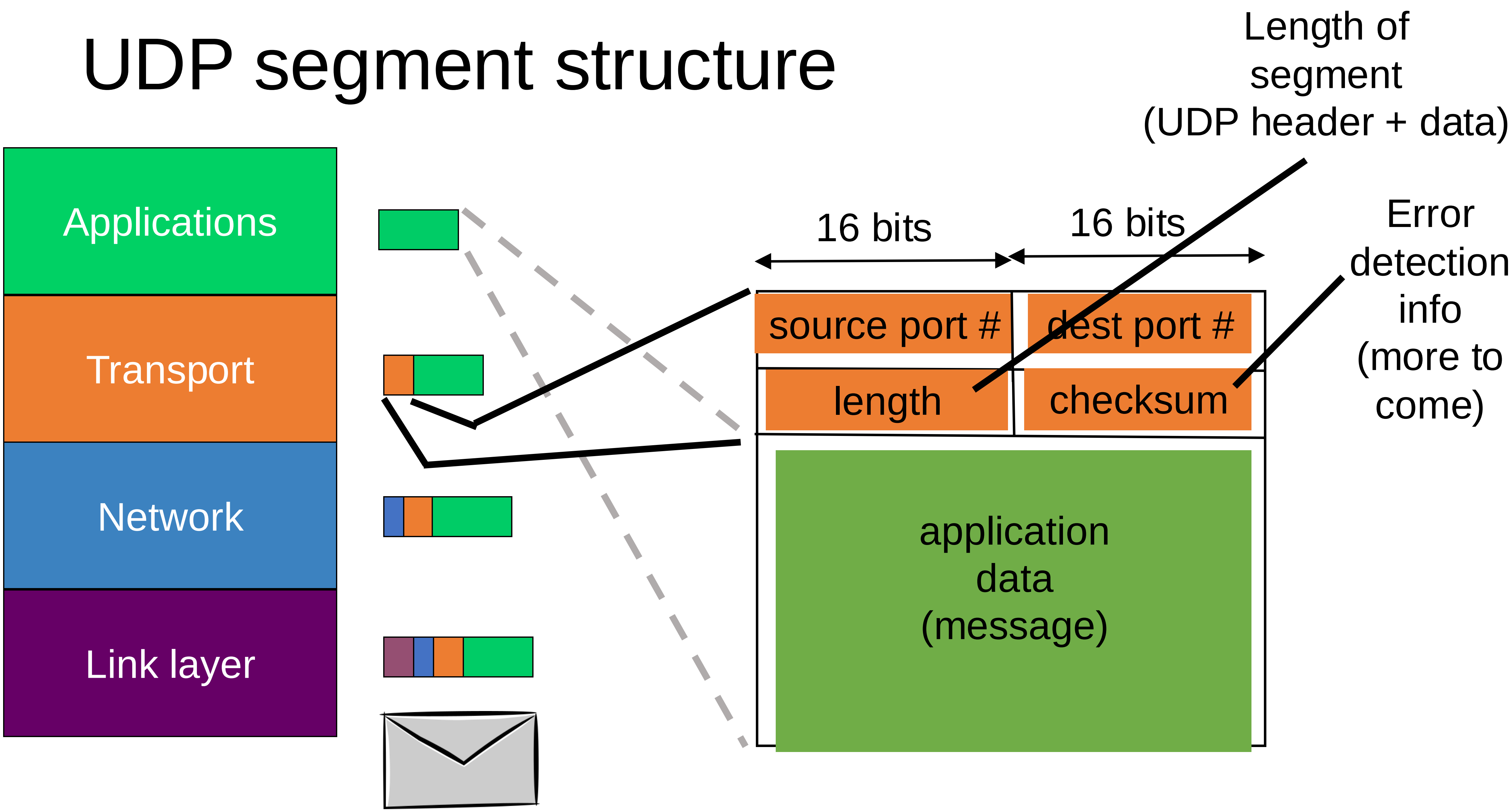
- Best effort service
  - UDP segments may be lost, corrupted, reordered
- UDP is connectionless
  - Each UDP segment handled independently of others (i.e. no “memory” across packets)
- Suitable for one-off req/resp
  - E.g., DNS uses UDP
- Early multimedia apps used UDP
  - Delay-sensitive but loss tolerant

Why are UDP’s guarantees even okay?

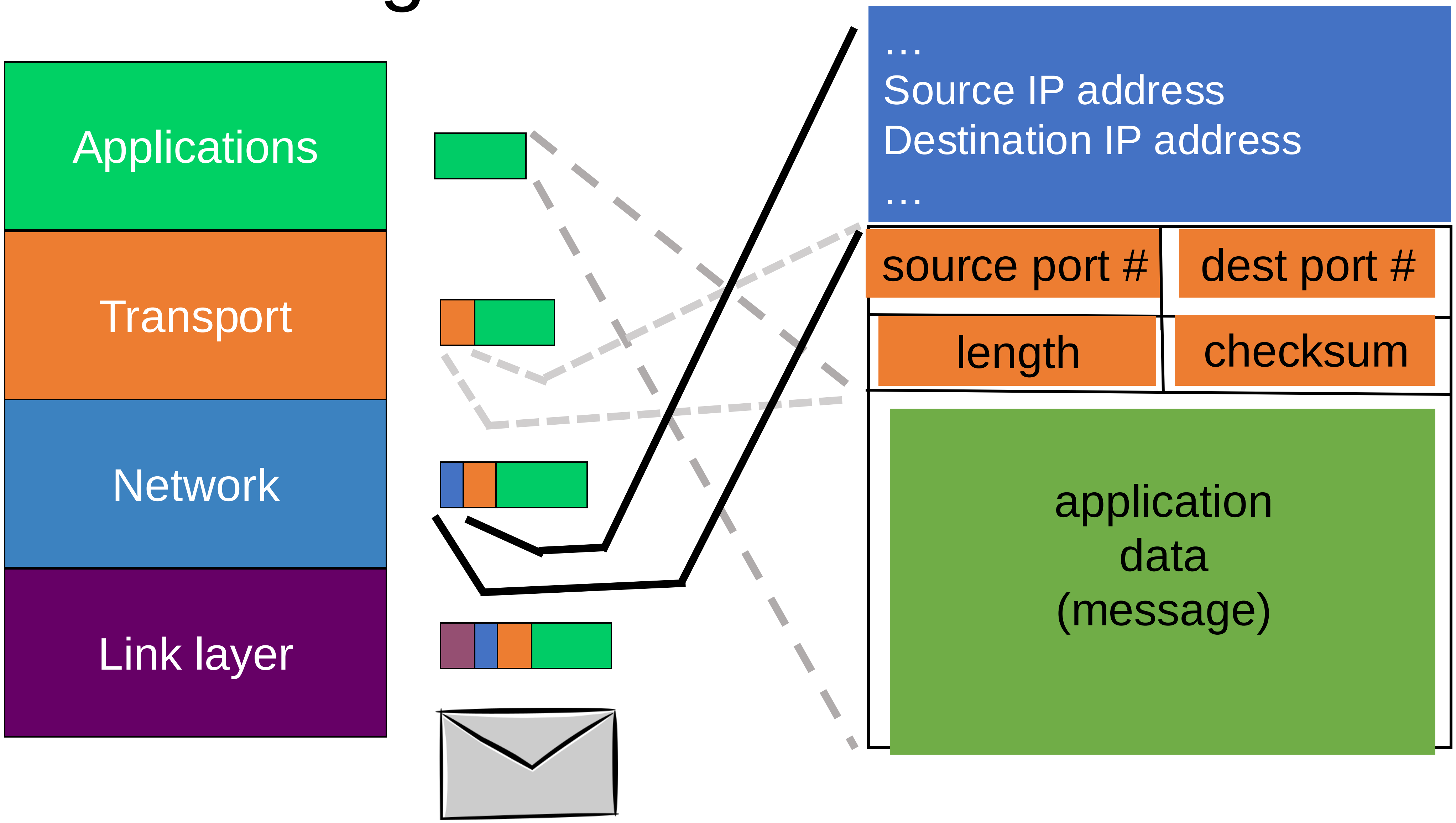
Simple & low overhead compared to TCP:

- No delays due to “connection establishment” (which TCP does)
  - UDP can send a packet immediately
- Small segment header (TCP’s is larger)
- UDP can blast data without control
  - TCP is more balanced and measured
- Less memory for connection “state” at sender & receiver relative to TCP

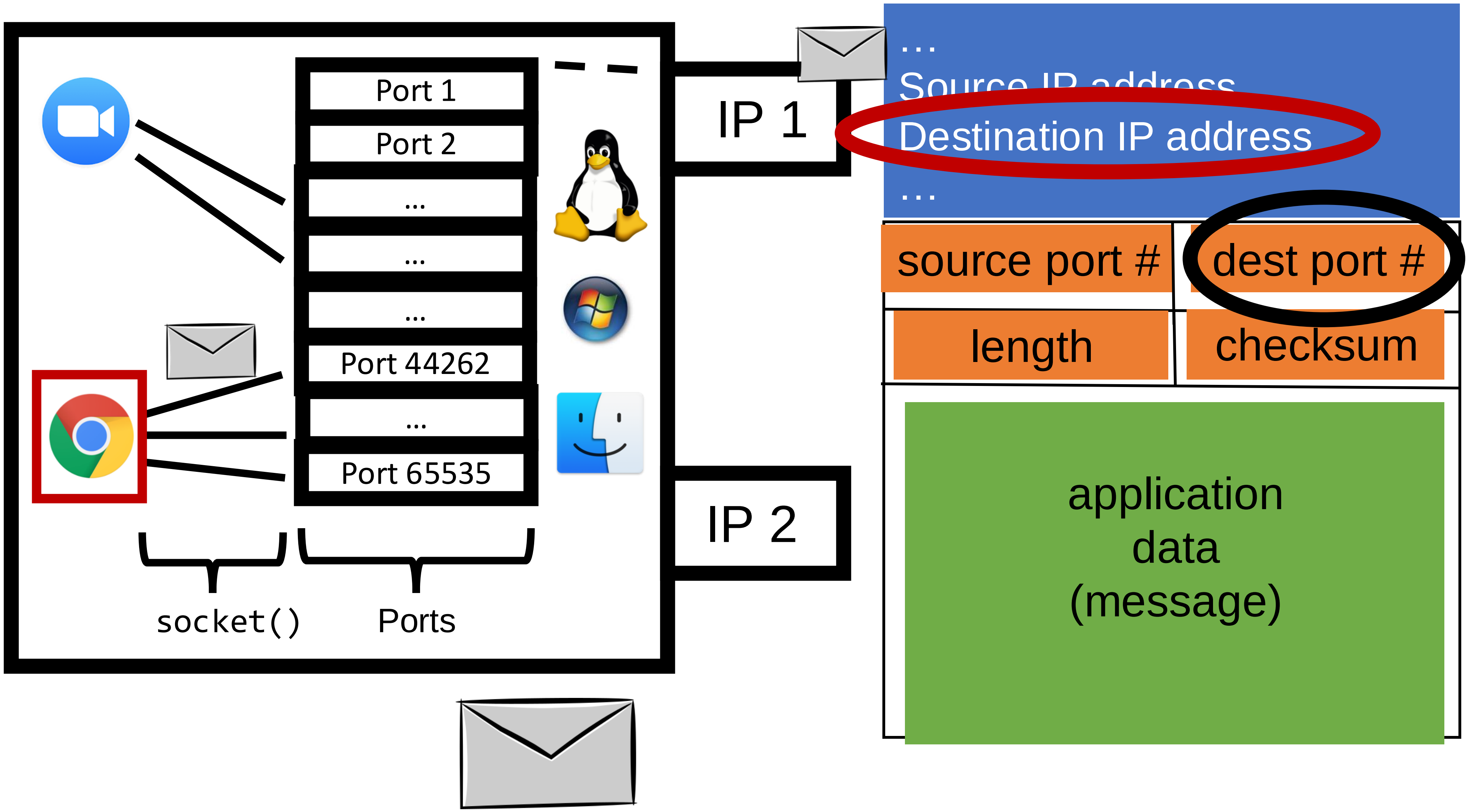
UDP segment structure



UDP segment structure



Review: UDP demultiplexing



Seeing UDP packets in action

- How to craft and send (UDP) packets?
  - It’s simpler than you think!

- `sudo tcpdump -i lo -XAvvv udp # observe packets`
- `sudo scapy # tool used to send crafted packets`
- Example:
  - `send(IP(dst="127.0.0.1")/UDP(sport=1024, dport=2048)/"hello world”, iface="lo")`
- See other fields of UDP using `UDP().fields_desc`
- Scapy can send and receive crafted packets!
  - However, it requires sudo (superuser privileges)

Error Detection in the Transport Layer

Why error detection?

- Network provides best effort service
- UDP is a simple and low overhead transport
  - Data may be corrupted along the way (e.g., 1 -> 0)
- However, simple error detection is possible!
  - Was the data I received the same data the remote machine sent?
- Error detection is a useful feature for all transport protocols including TCP

- Q: Suppose you’re sending a package to a friend. How would you detect tampering with that package?

Error Detection in UDP and TCP

- Key idea: have sender compute a function over the data
  - Store the result in the packet
  - Receiver can check the function's value in received packet

- An analogy: you're sending a package of goodies and want your recipient to know if goodies were leaked along the way

- Your idea: weigh the package; stamp the weight on the package
  - Have the recipient weigh the package and cross-check the weight with the stamped value

Requirements on error detection function

- Function must be easy to compute
- Function value must change if the packet changes
  - If the packet was modified through “likely” changes, the function value must change
- Function must be easy to verify

- UDP and TCP use a class of function called a checksum
  - Very common idea: used in multiple parts of networks and computer systems

UDP & TCP's Checksum function

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP/TCP checksum field

Receiver:

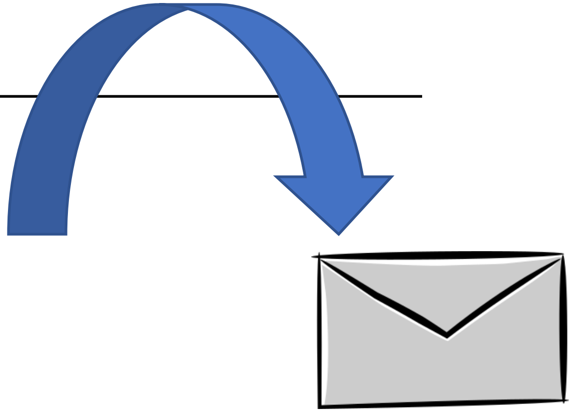
- compute a checksum of the received segment, including the checksum in packet itself
- check if the resulting (computed) checksum is 0
- NO – an error is detected
- YES – *assume* no error

Computing 1's complement sum

- Very similar to regular (unsigned) binary addition.
- However, when adding numbers, a carryout from the most significant bit needs to be added to the result

- Example: add two 16-bit integers

		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



From the UDP specification (RFC 768)

- Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.

- The pseudo header conceptually prefixed to the UDP header contains the source address, the destination address, the protocol, and the UDP length.

Warning: Technical language ahead

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
  - However, checksums don’t enable the receiver to detect where the error lies or correct the error(s)
  - Checksum is an error detection mechanism; not a correction mechanism.

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!)

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?

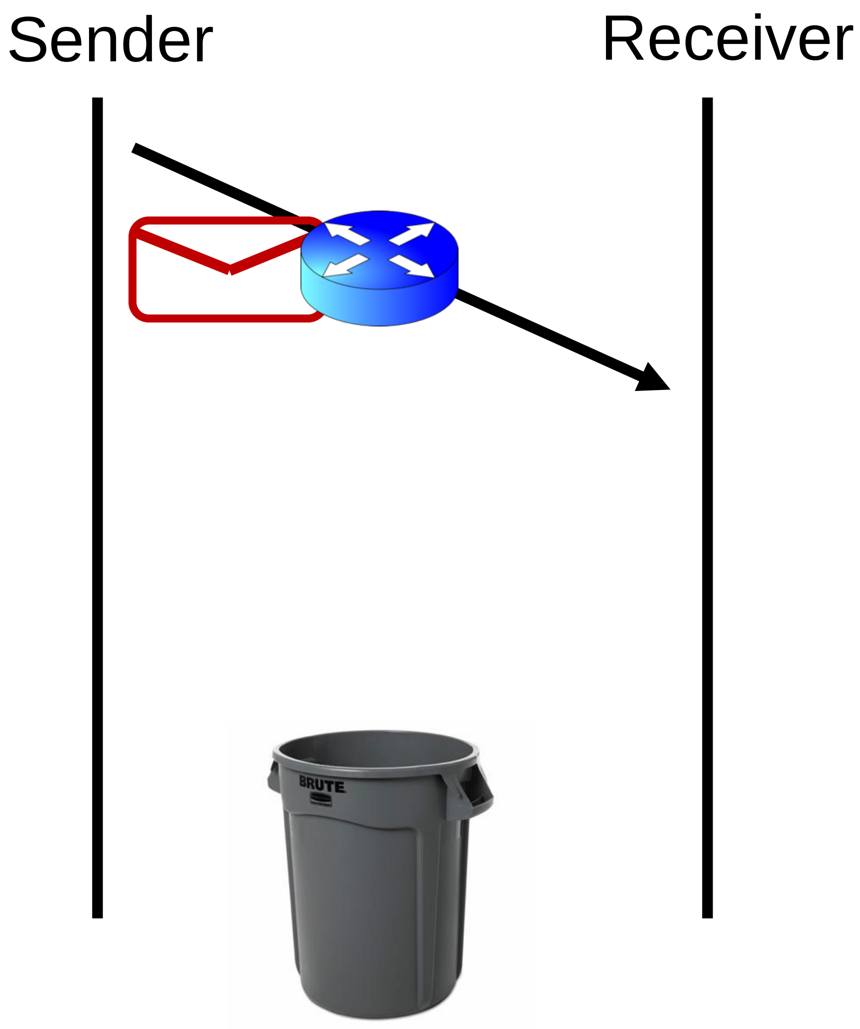
Summary of UDP

- 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

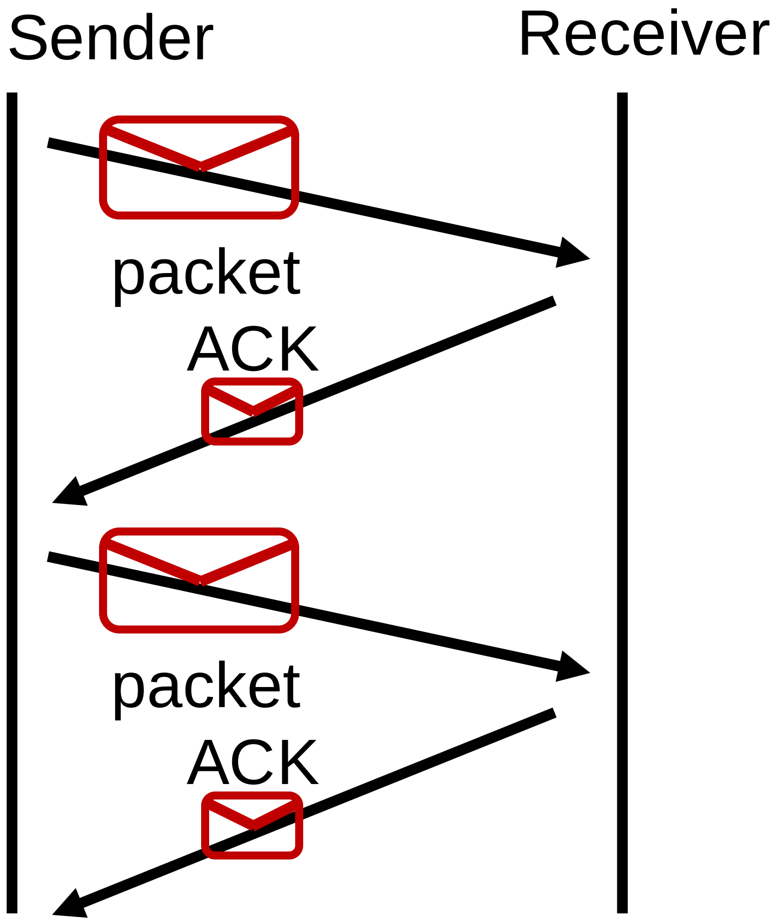


- How might a sender and receiver ensure that data is delivered reliably (despite some packets being lost)?
- TCP uses three mechanisms

Coping with packet loss: (1) ACK

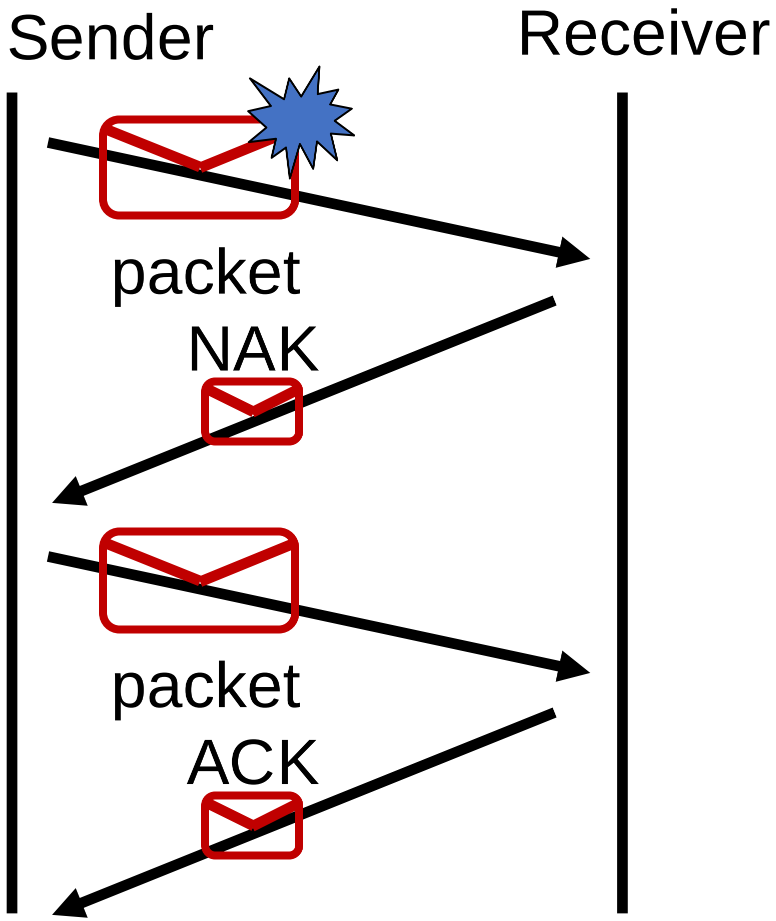
- Key idea: Receiver returns an acknowledgment (ACK) per packet sent

- If sender receives an ACK, it knows that the receiver got the packet.



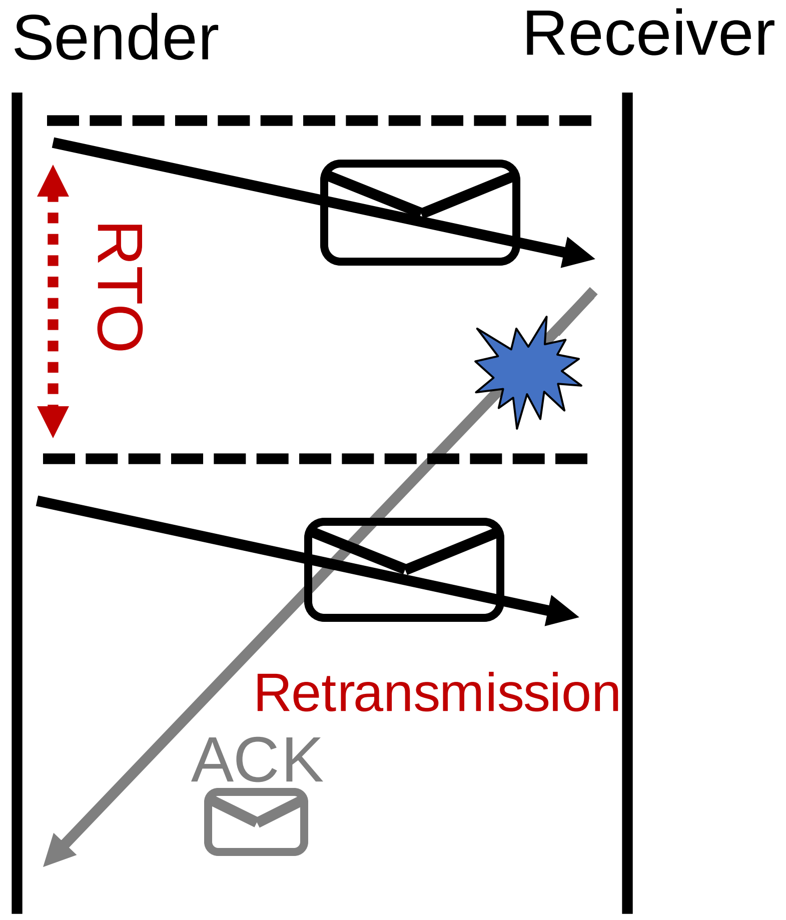
Coping with packet corruption: (1) ACK

- 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.



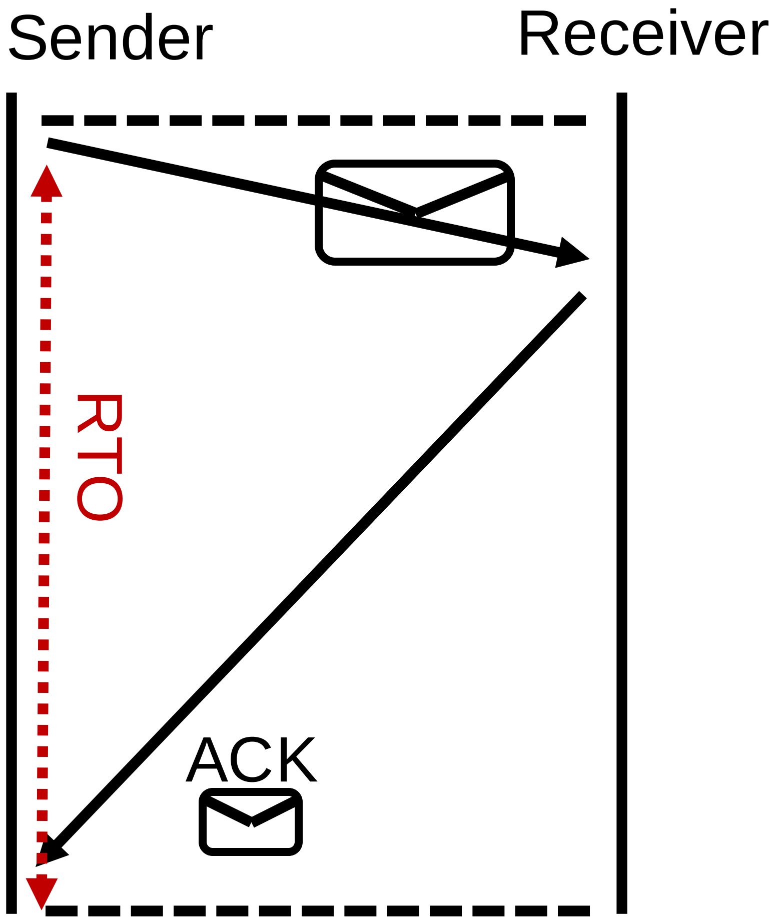
Coping with packet loss: (2) RTO

- 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



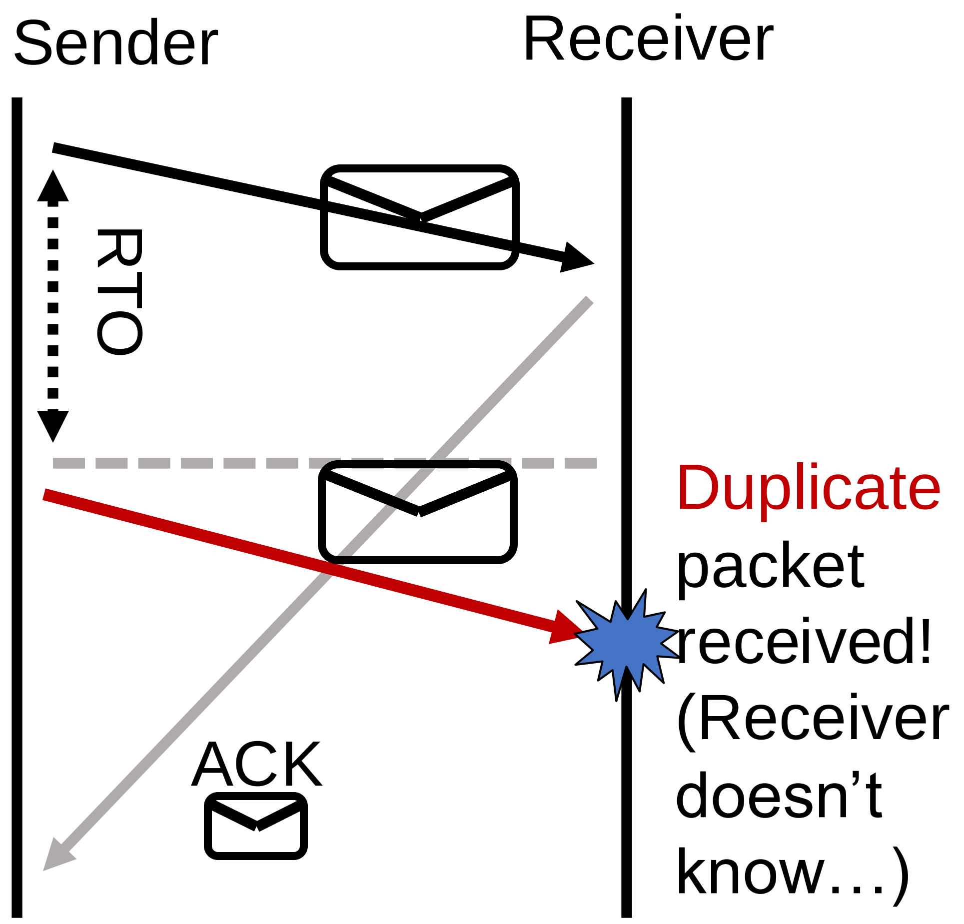
How should the RTO be set?

- 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

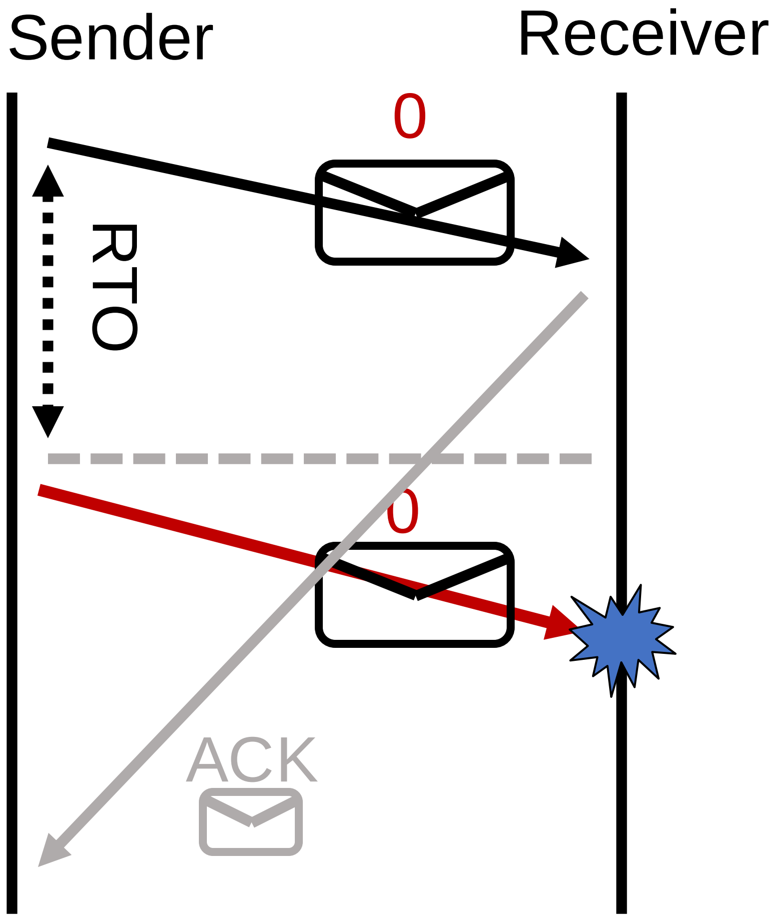
- If ACKs delayed beyond the RTO, sender may retransmit the same data
  - Receiver wouldn't know that it just received duplicate data from this retransmitted packet
- Add some identification to each packet to help distinguish between adjacent transmissions
  - This is known as the sequence number



Coping with packet loss: (3) Sequence #s

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

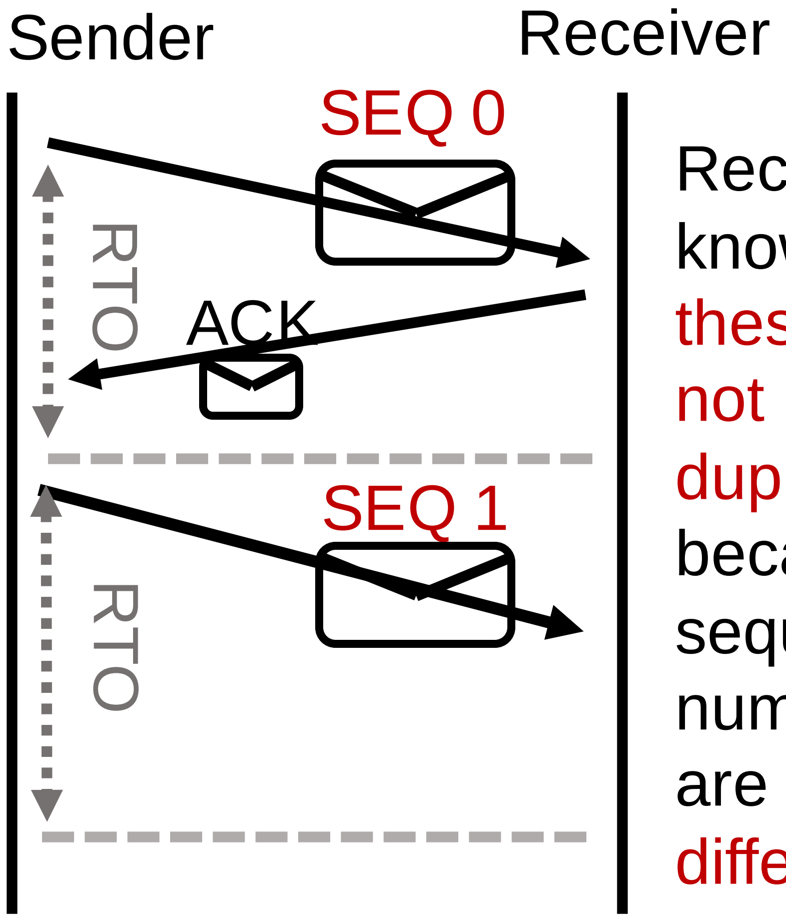
- At the receiver: sequence number helps disambiguate a fresh transmission from a retransmission
  - Sequence number same as earlier: retransmission
  - Fresh sequence number: fresh data



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

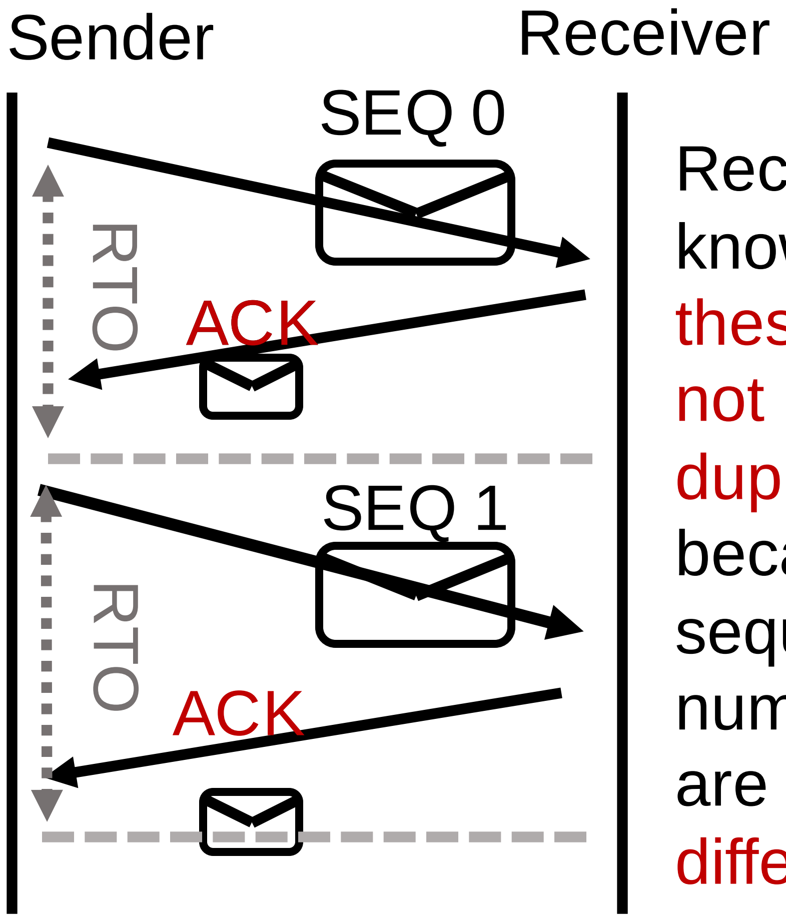


Receiver knows these are not duplicate, because sequence numbers are different

Coping with packet loss: (3) Sequence #s

- A good scenario: packet successfully received and ACK returned within RTO

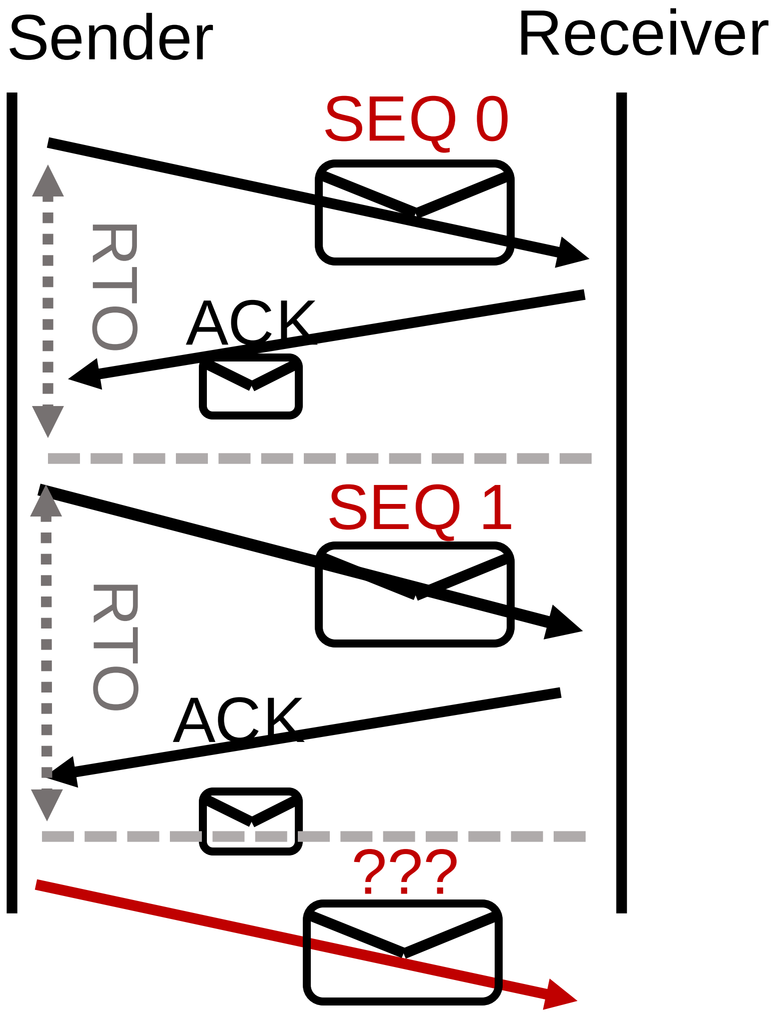
- Sequence numbers of successively transmitted packets are different



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

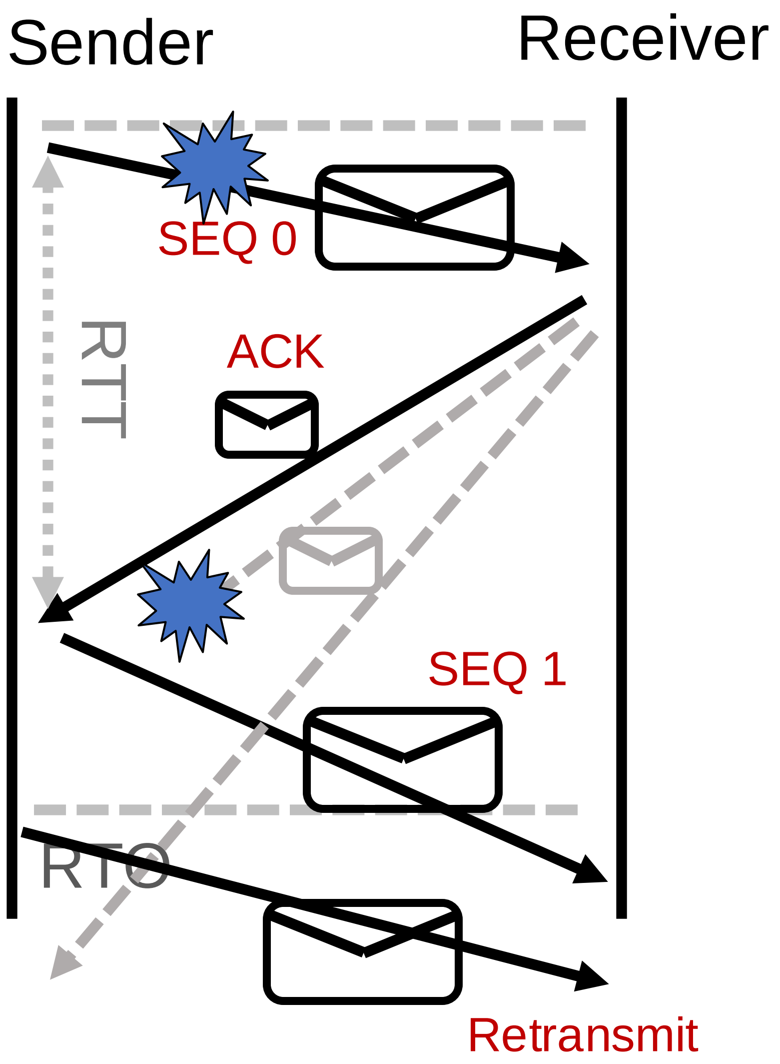
Q: What is the seq# of third packet?

- 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!