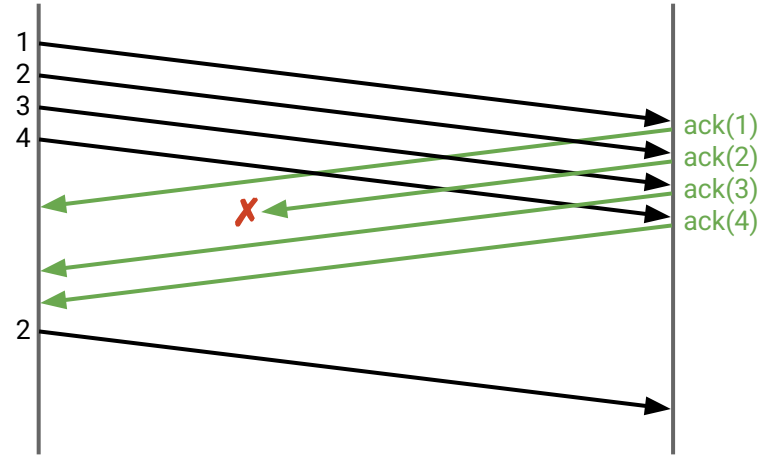




Lecture 11 (Transport 1)

The Transport Layer: Reliability

CS 168, Fall 2026 @ UC Berkeley

Transport Layer Design Goals

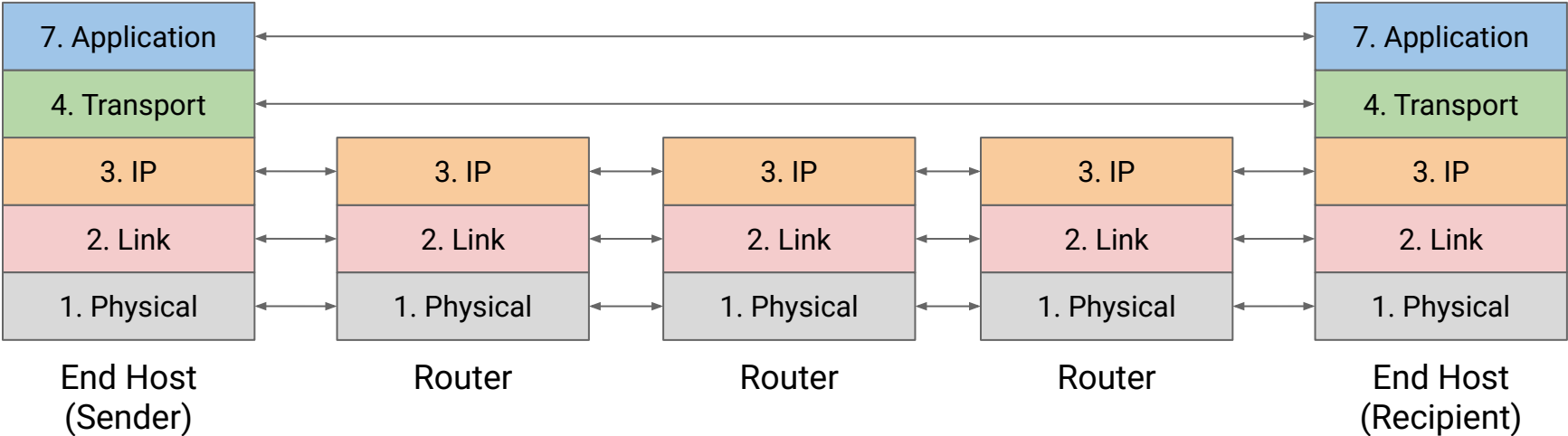
Lecture 11, CS 168, Fall 2026

Transport Layer Design Goals

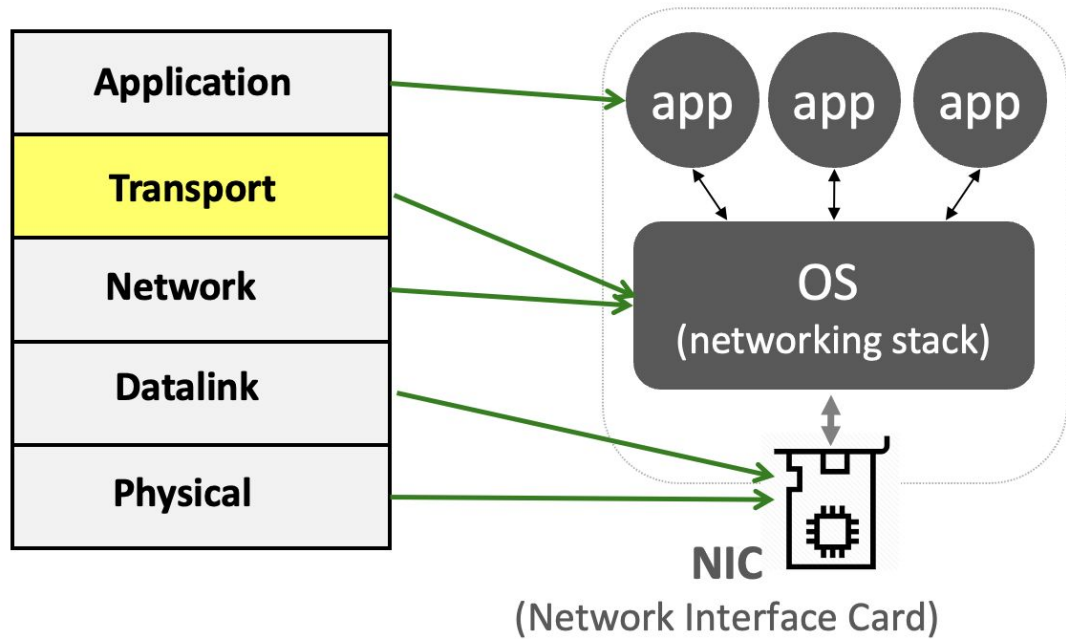
Implementing Reliability

- Single Packet
- Multiple Packets (Windows)
- Avoiding Overload
(Flow and Congestion Control)
- Smarter Acknowledgments
- Detecting Loss Early
- Alternate Designs

Recall: Transport Layer is Implemented at End Hosts



Recall: Transport Layer is Typically Implemented in the host OS



- Bridge the gap between:
 - The abstraction application developers want
 - The abstraction the network can easily provide
- Having a common implementation simplifies application development

- Bridge the gap between:
 - The abstraction application developers want
 - The abstraction the network can easily provide
- Having a common implementation simplifies application development
- Abstraction at the network layer (IP): best-effort delivery of individual packets
 - Packets can be lost, corrupted, reordered, delayed, or duplicated
- Problem: Applications don't want to think about packets and best-effort.
 - Programmers want to think in terms of a more convenient abstraction

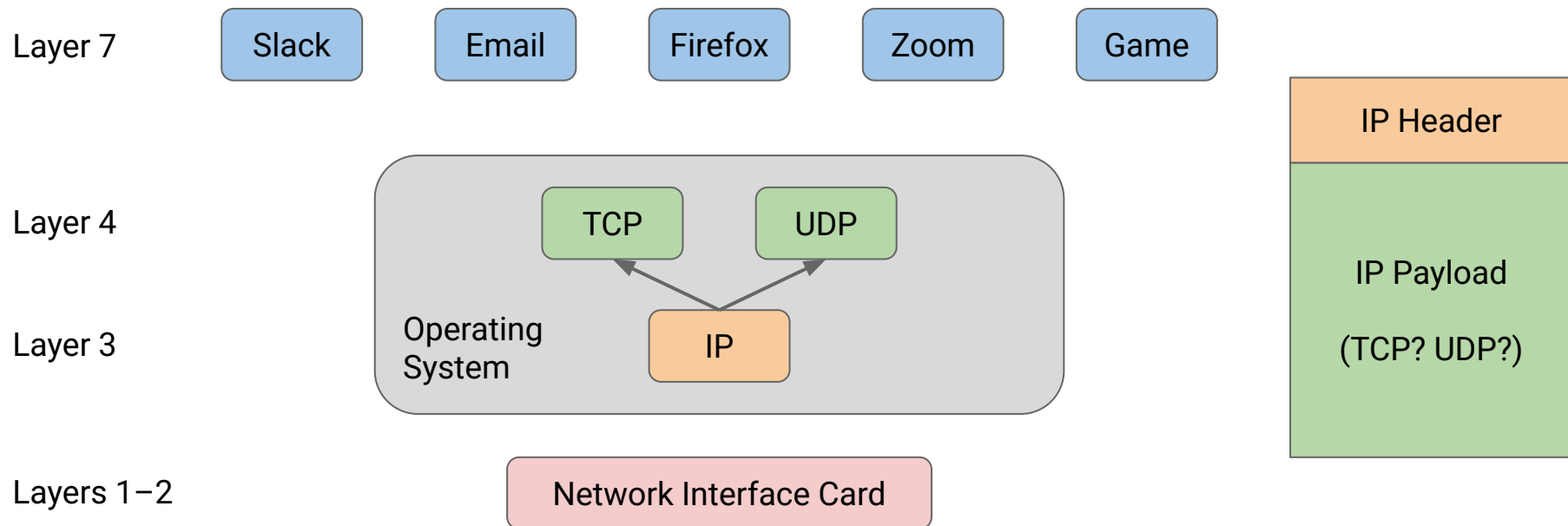
Transport Layer Protocols: Functions

1. Demultiplexing: between processes/applications
 2. Translate to/from packets and application-level abstraction
 3. Reliability
 4. Flow control: avoid overloading the receiver
 5. Congestion control: avoid overloading the network (lecture#13, 14)
- A transport protocol doesn't have to provide all of the above functions!
 - Many transport protocols exist but the most common are TCP and UDP
 - **TCP** (Transmission Control Protocol): Implements 1-5 above
 - **UDP** (User Datagram Protocol): Only implements 1 & 2

Transport Layer Feature: De-Multiplexing with Ports

Recall: Protocol field in IP Header (L3) supports de-multiplexing between L4 protocols.

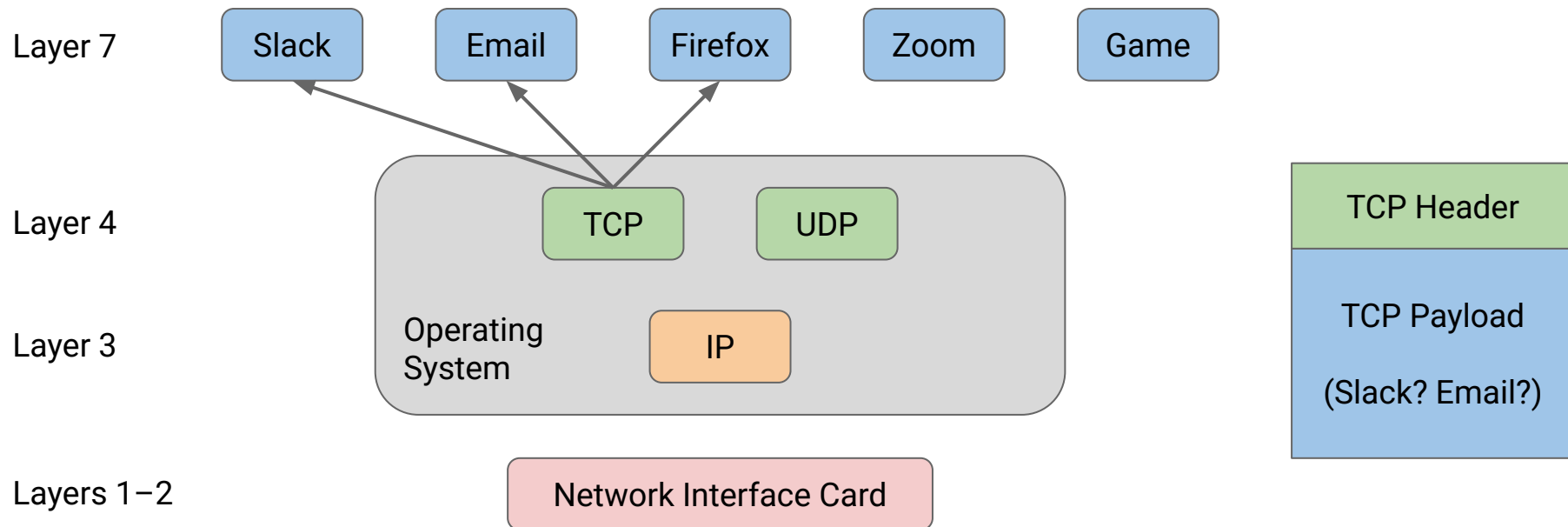
- Should I pass this packet to the TCP code or the UDP code?



Transport Layer Feature: De-Multiplexing with Ports

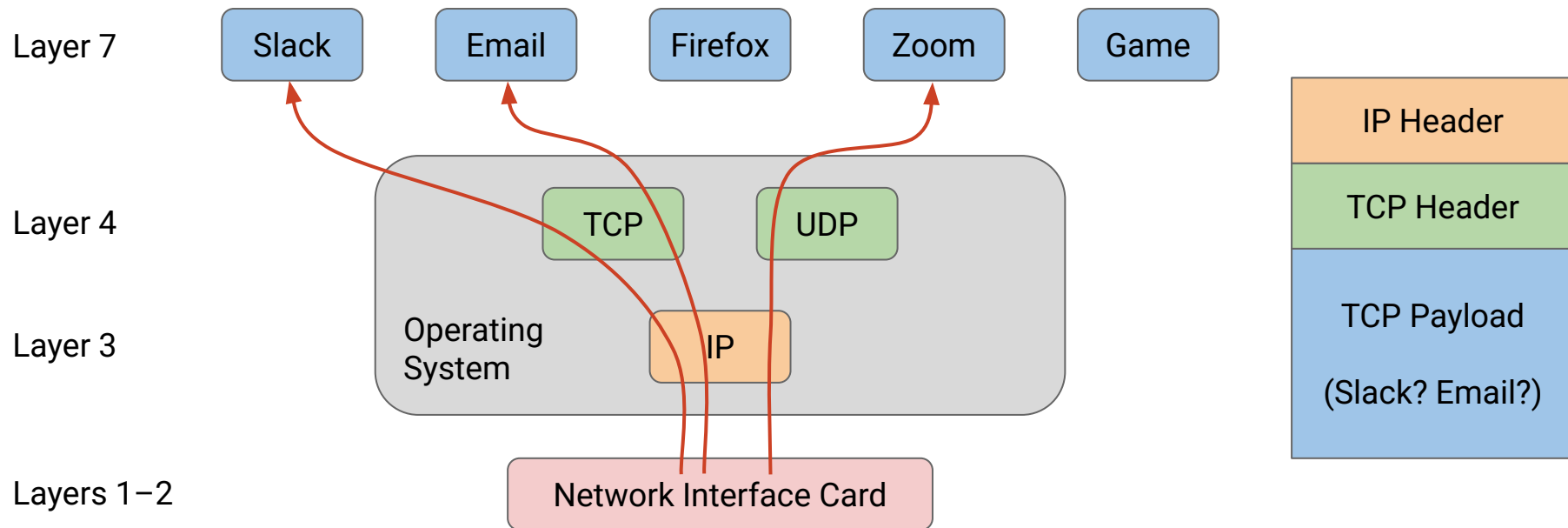
Port field in TCP header (L4) supports de-multiplexing between L7 applications.

- Which application should I pass this packet to?



Transport Layer Feature: De-Multiplexing with Ports

Protocol (L3) and port (L4) together allow us to de-multiplex arriving packets.



Ports identify applications on a specific machine

- Port numbers are 16 bits long
- Both TCP and UDP implement port numbers
- Consider a client (e.g., browser on your laptop) accessing a public server (e.g., [cnn.com](https://www.cnn.com))
 - The server's port number needs to be well-known and public
 - Examples: SSH (22), HTTP (80).
 - Services can listen for requests on a well-known port.
 - Client's port number can be anything
 - "Ephemeral" port numbers: 1024–65535.
 - Applications can pick a random port number

TCP implementing reliability gives the application a new abstraction.

- Without reliability (L3): best-effort packet abstraction.
 - Send individual packets, which might be dropped/corrupted/re-ordered.
- With reliability (L4): Reliable **bytestream abstraction**.
 - Sender application sends an arbitrary-length string of bytes.
 - The **same** string of bytes appear, **in order**, at the recipient application.

Transport Layer Feature: New Abstractions: Bytestream

TCP implementing reliability gives the application a new abstraction.

- Without reliability (L3): best-effort packet abstraction.
 - Send individual packets, which might be dropped/corrupted/re-ordered.
- With reliability (L4): Reliable **bytestream abstraction**.
 - Sender application sends an arbitrary-length string of bytes.
 - The **same** string of bytes appear, **in order**, at the recipient application.



UDP offers a **best-effort datagram abstraction**.

- Basically the same as the Layer 3 packet abstraction
- Application sends UDP a chunk of bytes which is sent as an IP packet
- The application is responsible for splitting data into individual packets
- No guarantee of delivery, ordering, etc.

Implementing Reliability: Single Packet

Lecture 11, CS 168, Fall 2026

Transport Layer Design Goals

Implementing Reliability

- Single Packet
- Multiple Packets (Windows)
- Avoiding Overload
(Flow and Congestion Control)
- Smarter Acknowledgments
- Detecting Loss Early
- Alternate Designs

Challenge: Building a reliable service (L4) on top of unreliable packet delivery (L3).

Semantics for correct delivery

- At the network layer: best-effort delivery
- At the transport layer: *at-least-once* delivery
 - Every packet arrives, but some might arrive more than once (duplicates)
- At the application layer: *exactly-once* delivery
 - Every packet arrives exactly once

Challenge: Building a reliable service (L4) on top of unreliable packet delivery (L3).

Semantics for correct delivery (bridging the gap across layers)

- At the network layer: best-effort delivery
- At the transport layer: *at-least-once* delivery
 - Every packet arrives, but some might arrive more than once (duplicates)
→ Implemented on top of best-effort delivery
- At the application layer: *exactly-once* delivery
 - Every packet arrives exactly once
→ By getting rid of duplicates before handing data to the application

Implementing Reliability: Goals

- Correctness: The destination receives every packet, uncorrupted, in order.
- Timeliness: Minimize time until data is transferred.
- Efficiency: Minimize use of bandwidth.
 - Avoid sending packets unnecessarily.
 - Example of inefficient protocol: Send 1000 copies of every packet.

Note!

A reliability protocol is allowed to give up, but must announce it to the application.

- Example: Computer not connected to Internet.
- TCP is not magic. Can't deliver packets if the computer is not connected.
- **TCP must never falsely claim to deliver a packet.**

Implementing Reliability: Single Packet

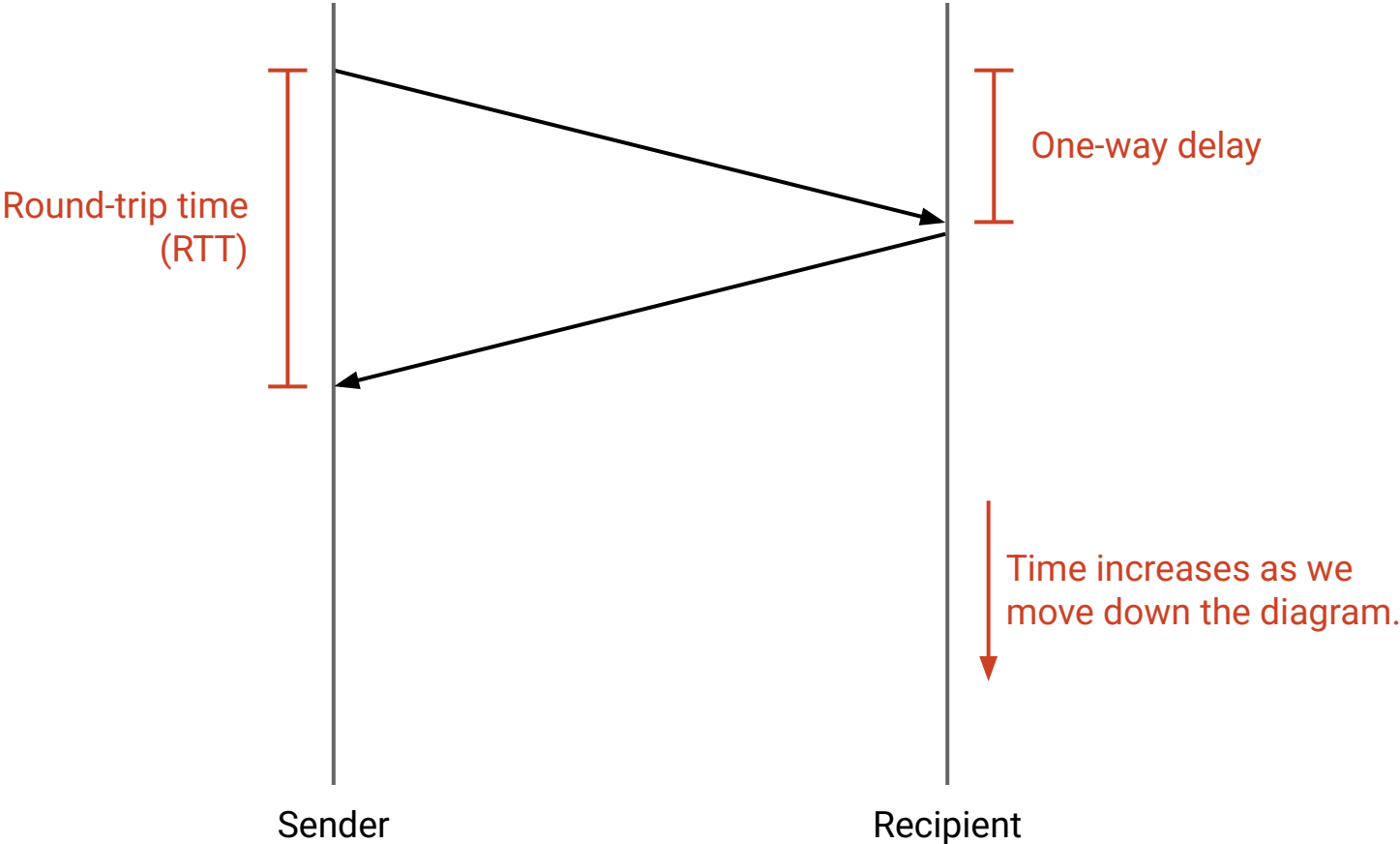
Lecture 11, CS 168, Fall 2026

Transport Layer Design Goals

Implementing Reliability

- **Single Packet**
- Multiple Packets (Windows)
- Avoiding Overload
(Flow and Congestion Control)
- Smarter Acknowledgments
- Detecting Loss Early
- Alternate Designs

Reminder: Timing Diagrams



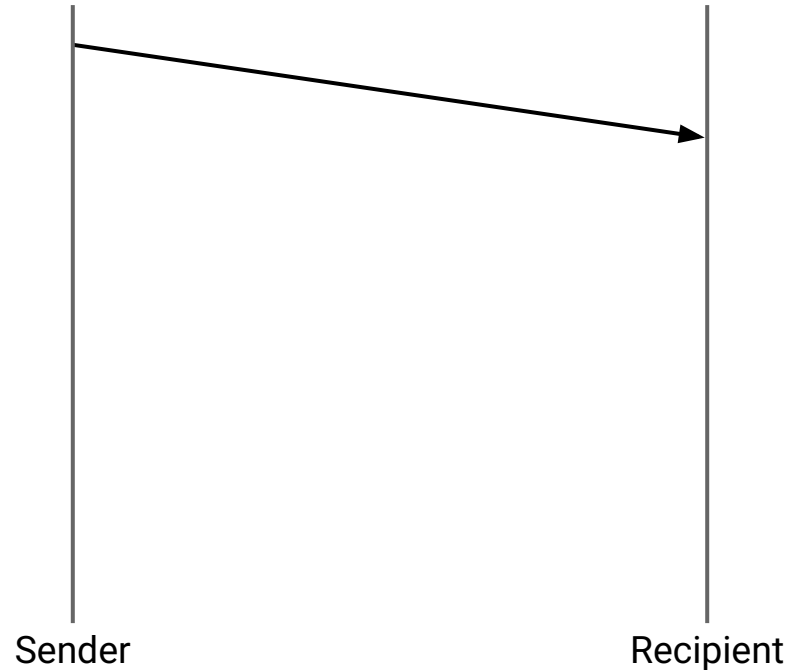
Let's start with providing *at-least-once* delivery for a single packet.

We have to deal with 5 problems from the best-effort service model:

1. Packets can be dropped.
2. Packets can be corrupted.
3. Packets can be delayed.
4. Packets can be duplicated.
5. Packets can be reordered.

Designing Reliability for a Single Packet

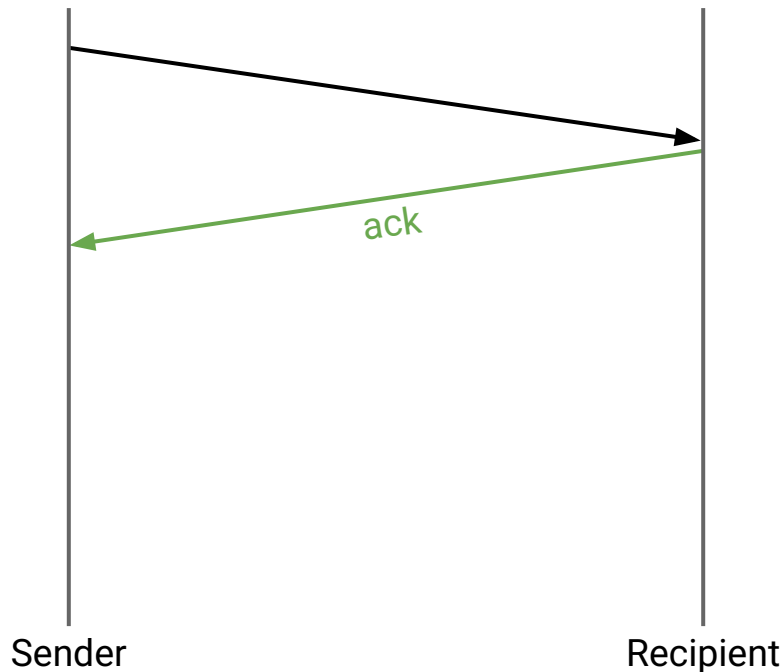
The packet is sent. How does the sender know if it was received successfully?



Designing Reliability for a Single Packet

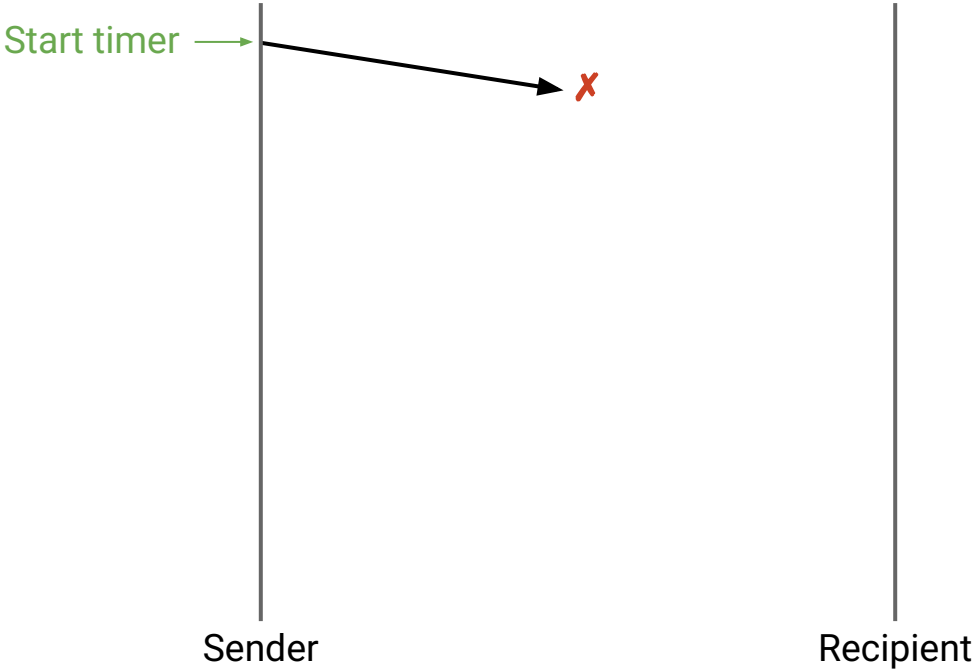
The packet is sent. How does the sender know if it was received successfully?

Solution: The recipient replies with an **acknowledgment** (ack).



Problem 1/5: Dropped Packets

Problem 1/5: Dropped Packets

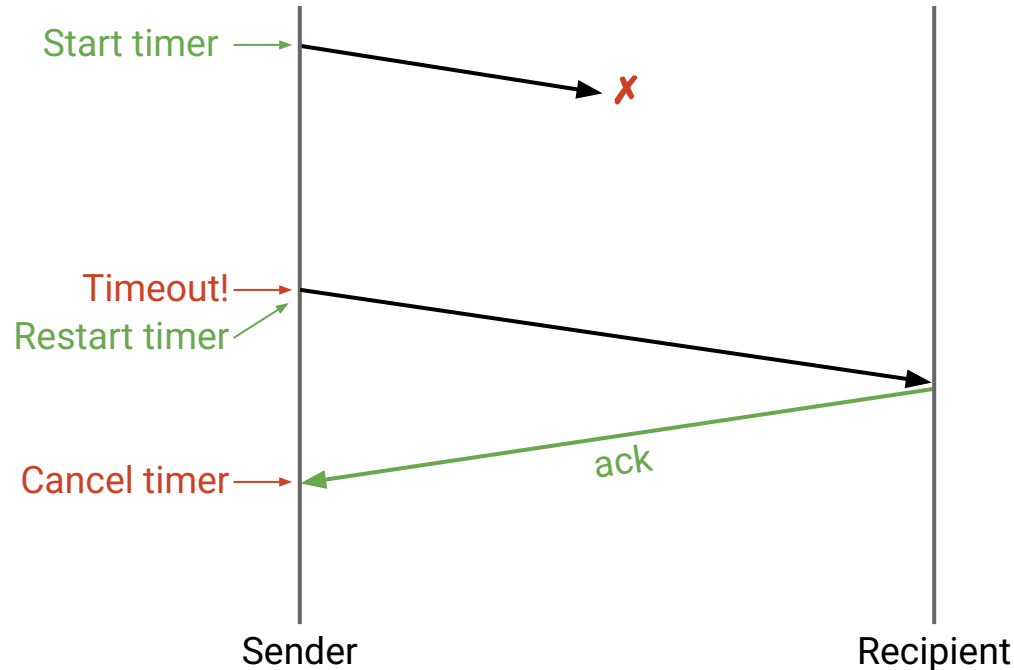


Problem 1/5: Dropped Packets

Solution: Set a timer when we send the packet.

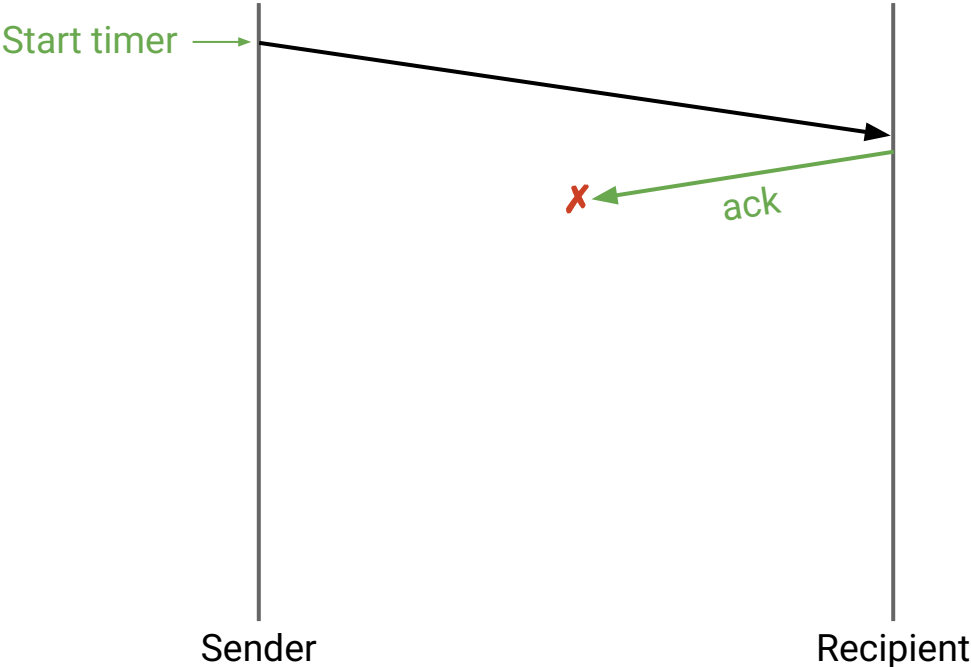
If the timer expires and we don't get the ack, resend the packet.

When we get the ack, cancel the timer.



Problem 1/5: Dropped Packets

What if the ack gets dropped?

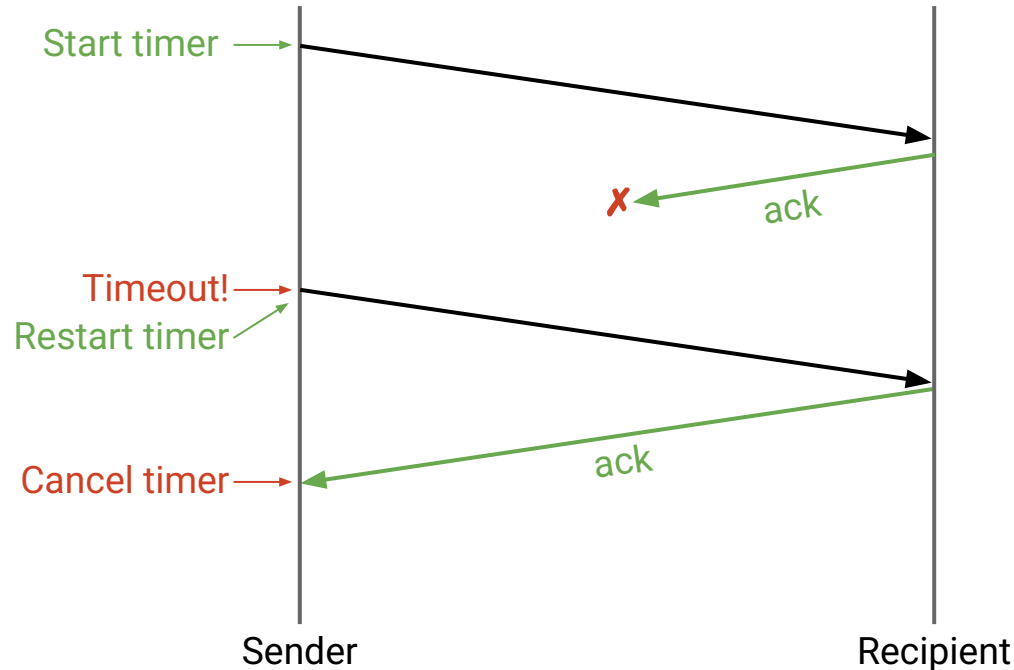


Problem 1/5: Dropped Packets

What if the ack gets dropped?

The timer expires, and the packet gets resent. Repeat until the ack arrives.

The destination received the packet twice, but that's okay (*at-least-once* delivery).



Problem 1/5: Dropped Packets

How do we set the timer length?

- Too long: Delays delivery.
- Too short: Causes unnecessary retransmission.

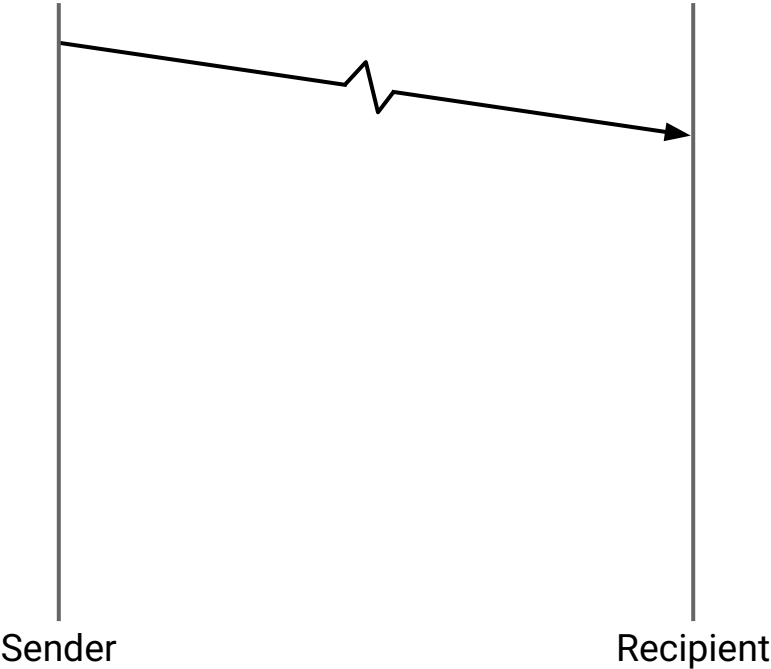
Ideally: Timer is proportional to RTT (round-trip time).

- Intuition: RTT is when you expected to see the ack.

In practice: Measuring the RTT is not trivial.

- RTT varies depending on what path the packet takes.
- RTT varies along a fixed path: packet could get stuck in queues at links.

Problem 2/5: Corrupted Packets

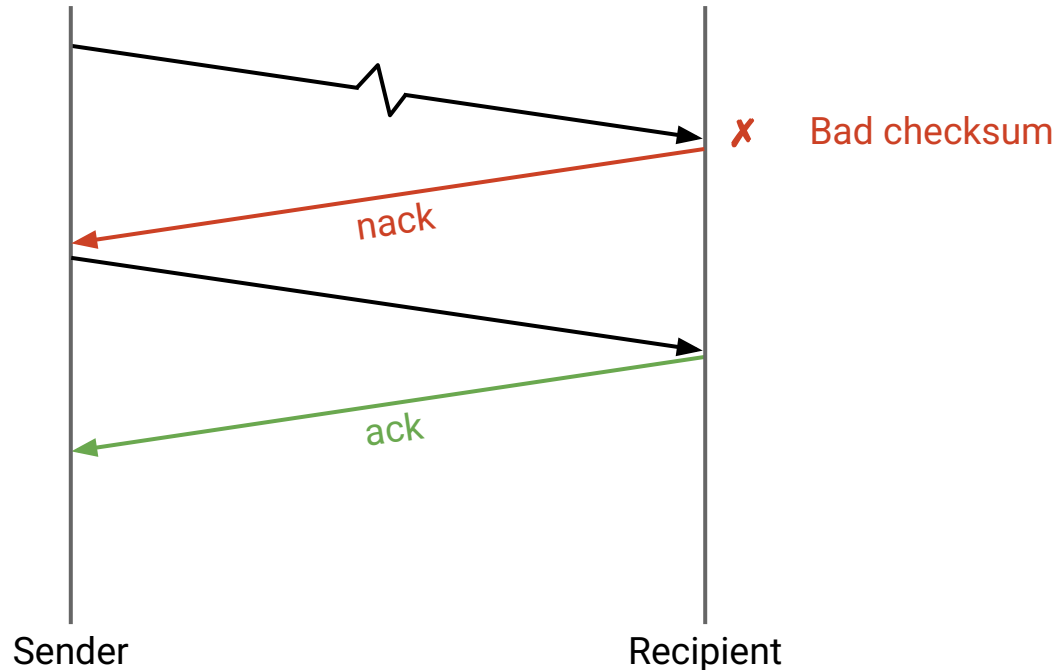


Problem 2/5: Corrupted Packets

Add a checksum to detect corruption. 2 approaches to dealing with a bad checksum:

Solution 1: When you get a corrupt packet, send a **negative acknowledgement** (nack).

Sender sees the nack and resends.

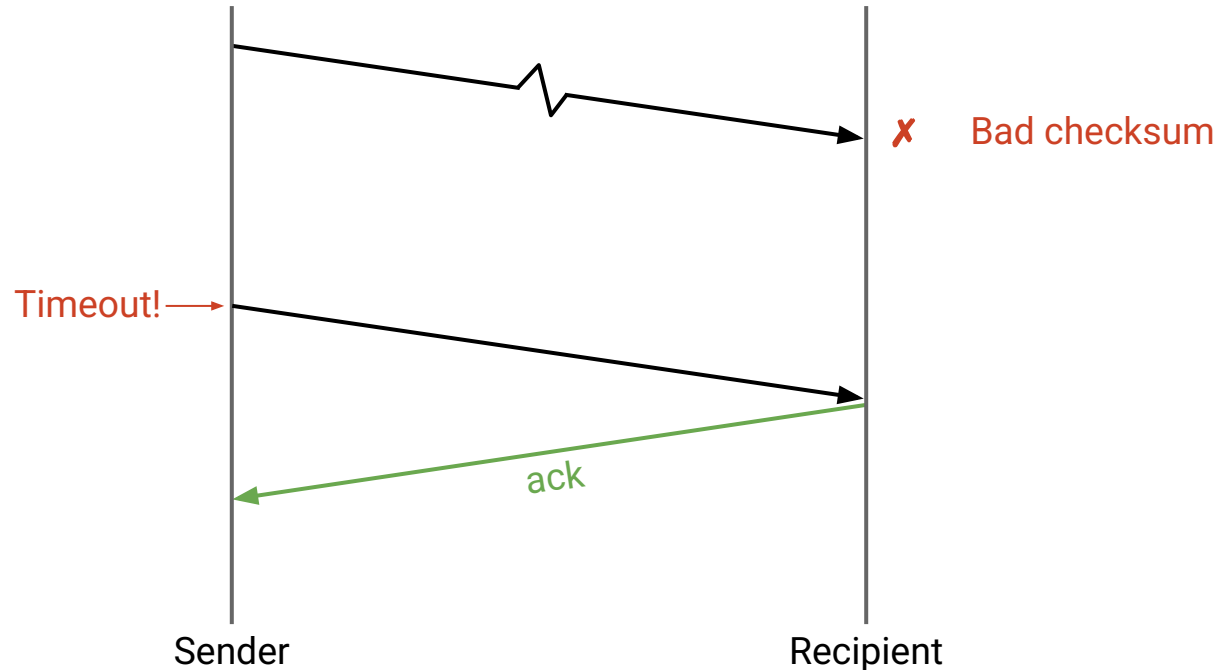


Problem 2/5: Corrupted Packets

Solution 2: When you get a corrupt packet, ignore it (don't send the ack).

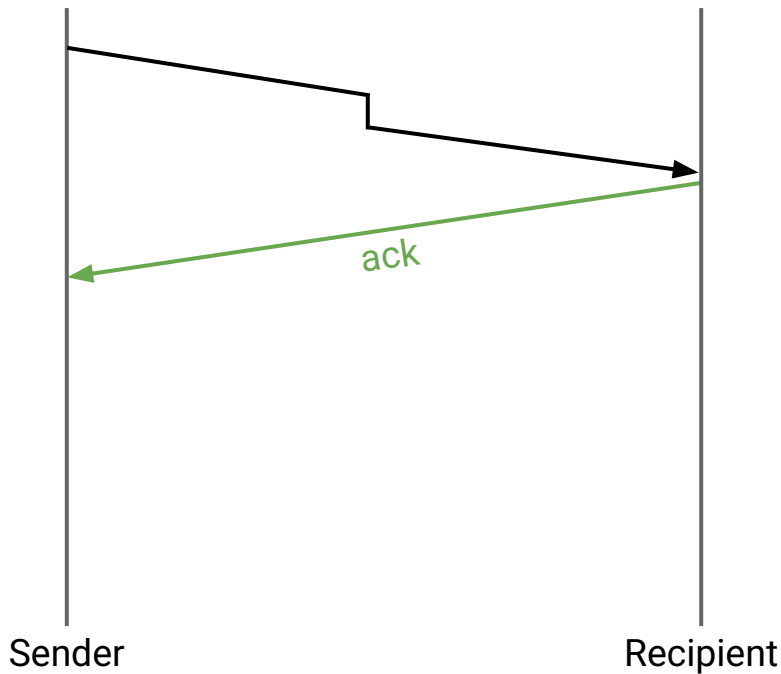
Sender times out and resends.

Both solutions work. TCP uses this one (no nacks).



Problem 3/5: Delayed Packets

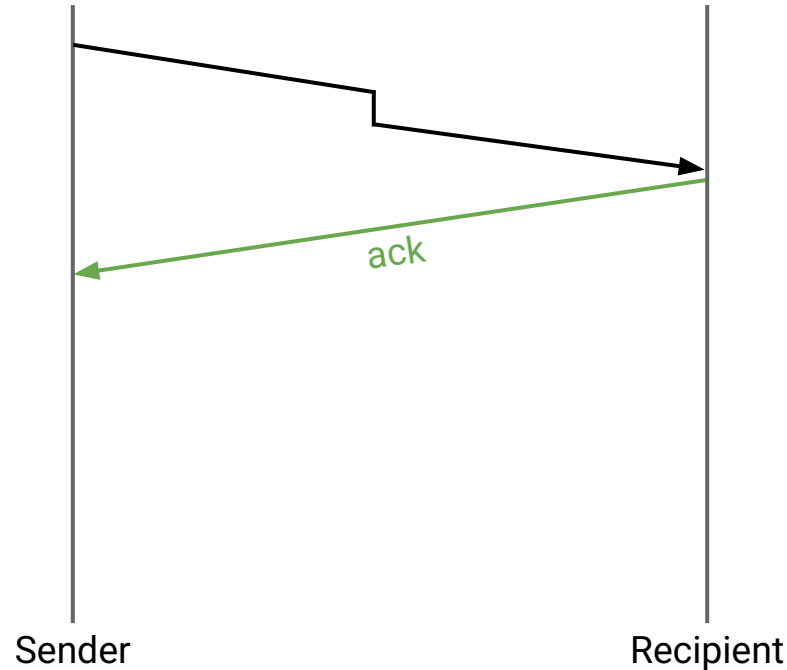
Minor delay?



Problem 3/5: Delayed Packets

Minor delay: No problem. The ack still arrives before the timeout.

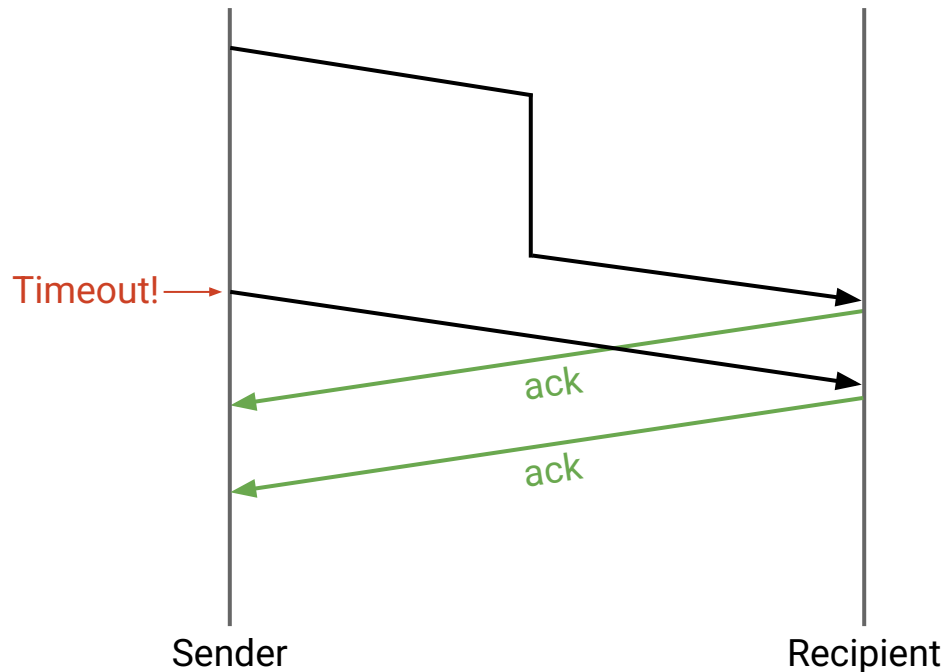
What happens if the packet is delayed such that it reaches after the timeout?



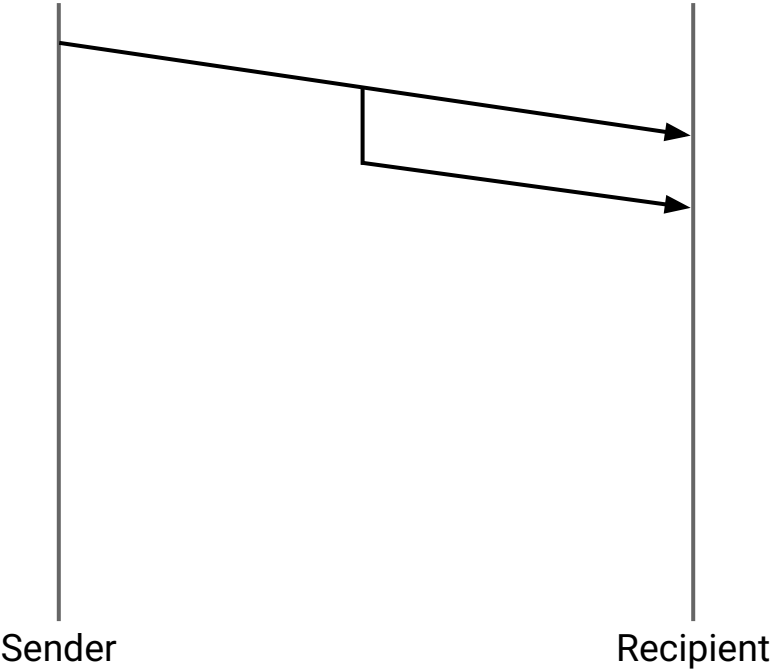
Problem 3/5: Delayed Packets

Longer delay: Sender times out and resends.

The sender receives two acks. That's fine.



Problem 4/5: Duplicated Packets

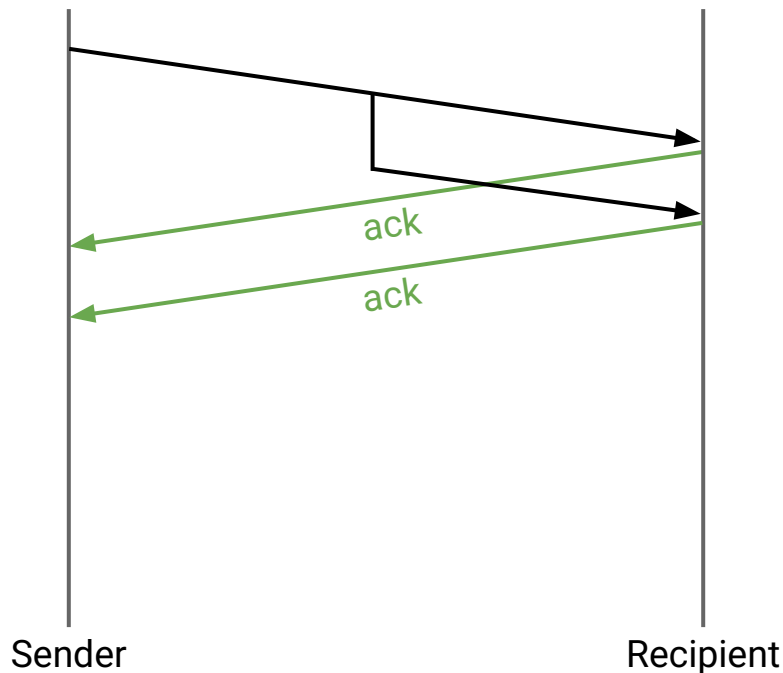


Problem 4/5: Duplicated Packets

The recipient gets the packet twice, and the sender receives two acks. That's fine.

Why would the network even duplicate a packet?

Usually, because of link-level reliability gone wrong (very rare).



We Have Solved Reliability for a Single Packet!

Sender:

- Send packet, and start a timer.
- If the ack doesn't arrive before the timer goes off, resend packet and reset timer.
- If ack arrives, cancel timer.

Recipient:

- When you receive the packet, send the ack.

We Have Solved Reliability for a Single Packet!

Sender:

- Send packet, and start a timer.
- If the ack doesn't arrive before the timer goes off, resend packet and reset timer.
- If ack arrives, cancel timer.

Recipient:

- When you receive the packet, send the ack.

We solved all 5 problems from the best-effort service model:

1. Packets can be dropped. (timeout and resend)
2. Packets can be corrupted. (nack, or timeout and resend)
3. Packets can be delayed. (data received multiple times)
4. Packets can be duplicated. (data received multiple times)
5. Packets can be reordered. (not relevant in single-packet case)

What did we learn from this protocol?

- *Checksums* help detect corruption.
- The recipient should send *feedback*: ack (positive), possibly also nack (negative).
- The sender *retransmits* lost packets.
- Use a *timeout* to decide when to resend a packet.

At-least-once delivery simplifies our protocol.

- Recipient can receive the same packet more than once.
- Sender can see the same ack/nack more than once.
- To achieve *exactly-once* delivery, recipient simply discards duplicates.

Multiple Packets, Windows

Lecture 11, CS 168, Fall 2026

Transport Layer Design Goals

Implementing Reliability

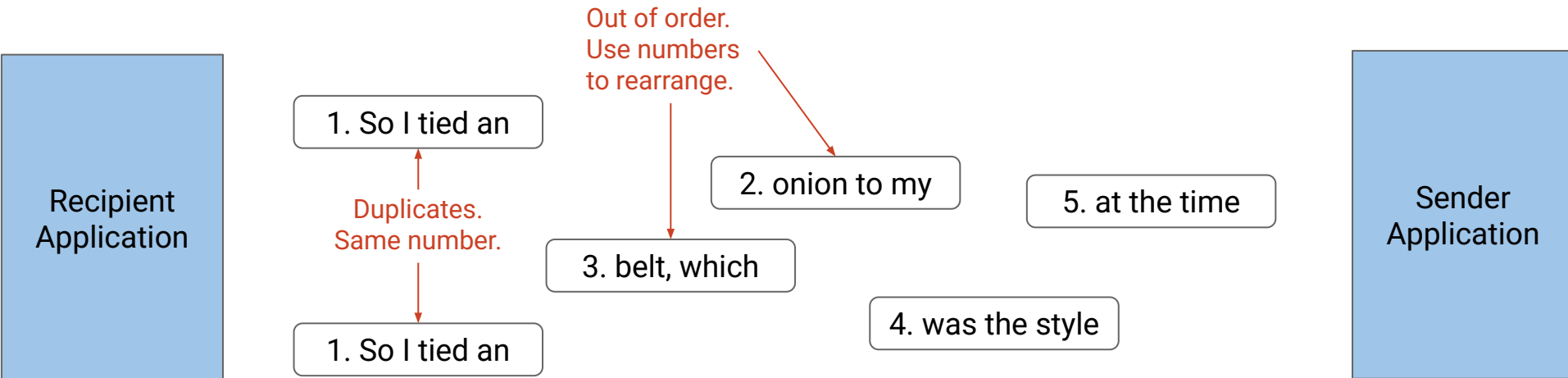
- Single Packet
- **Multiple Packets (Windows)**
- Avoiding Overload
(Flow and Congestion Control)
- Smarter Acknowledgments
- Detecting Loss Early
- Alternate Designs

Designing Reliability for Multiple Packets

We can use the single-packet solution repeatedly to support sending multiple packets.

One extra design component: **sequence numbers**.

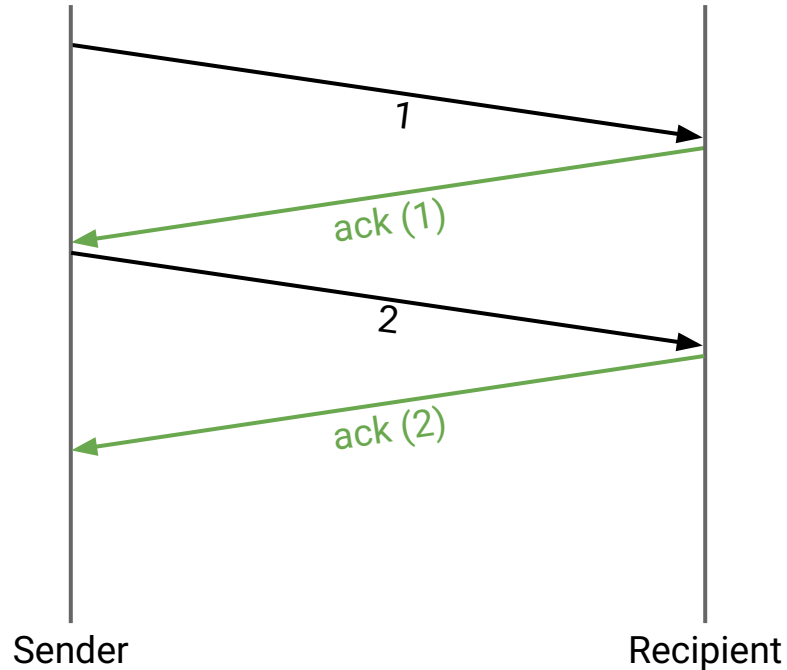
- Label each packet with a unique, increasing number.
- Label each ack with which numbered packet is being acked.
- Solves reordering – use numbers to reorder packets.



Sequence Numbers

Each packet has a unique, increasing sequence number.

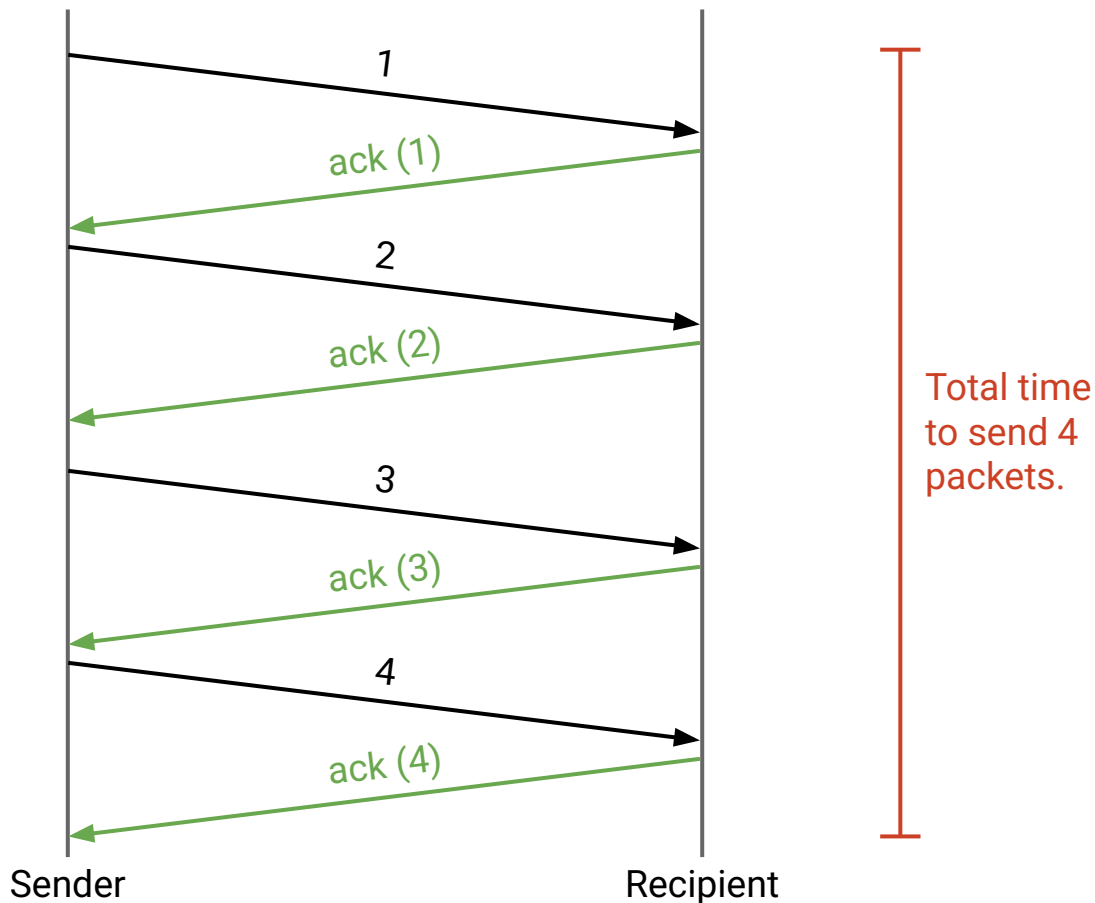
Acks indicate which packet is being acked.



Stop-and-Wait Protocol

Naive approach: Wait for packet i to be acked before sending packet $i+1$.

- This is correct, but really slow.
- Throughput: One packet per RTT.

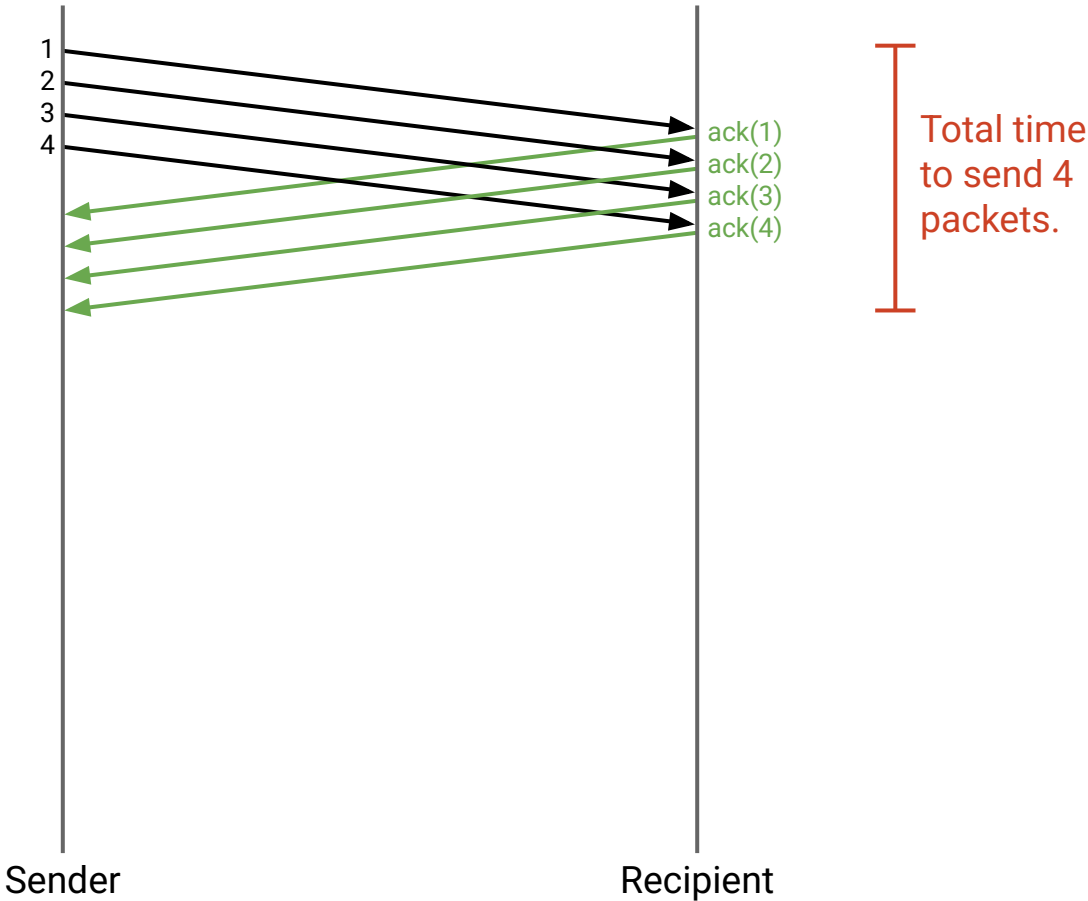


Window-Based Algorithms

Better approach: Send more packets while waiting for acks.

Idea: Have multiple packets **in flight** simultaneously.

In flight: Packet has been sent, but has not been acked yet.



Window-Based Algorithms

Idea: Have multiple packets in flight simultaneously.

Could we send all the packets at once?

- No, limited by bandwidth (at sender, router, and destination).

Window-based algorithms: Limit the amount of packets in flight.

- Maximum W packets can be in-flight (sent, but not acked) at any time.
- W is the size of the **window**.

To stay inside the window limit:

- Start: Send W packets.
- Each time a packet gets acked, send the next packet in line.

Window isn't necessary for correctness, but is added for efficiency (performance).

Window-Based Algorithms

Start: Send $W = 4$ packets.

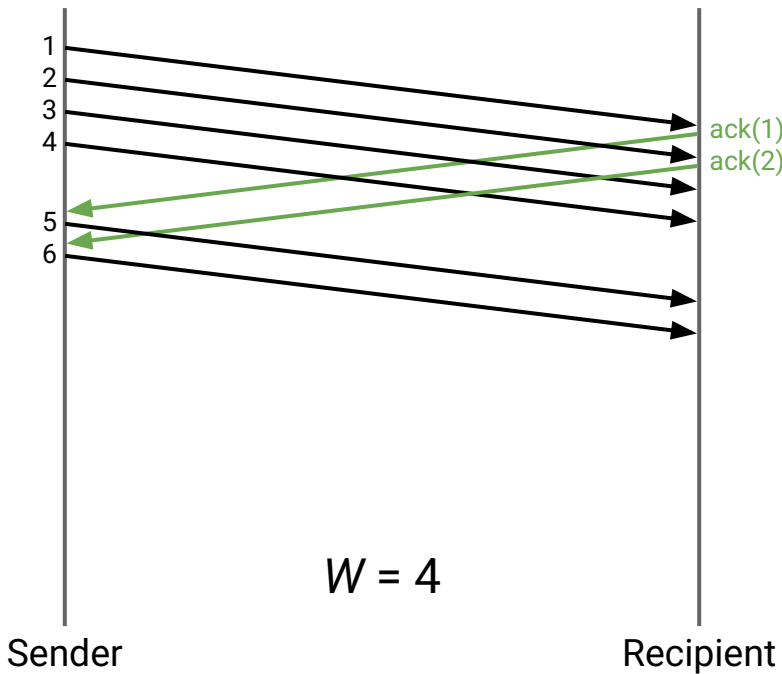
In-flight packets: {1, 2, 3, 4}.

When ack(1) arrives: Send 5.

In-flight packets: {2, 3, 4, 5}.

When ack(2) arrives: Send 6.

In-flight packets: {3, 4, 5, 6}.



Setting Window Size

How big should the window be?

Pick window size W to balance three goals:

1. Take advantage of network capacity ("fill the pipe")...
2. ...but don't overload the recipient ...
3. ...and don't overload the network

Setting Window Size (1/3): Filling the Pipe

First goal: Take advantage of network capacity ("fill the pipe").

- The sender should never be sitting idle.
- Set W large enough to keep the sender constantly busy.

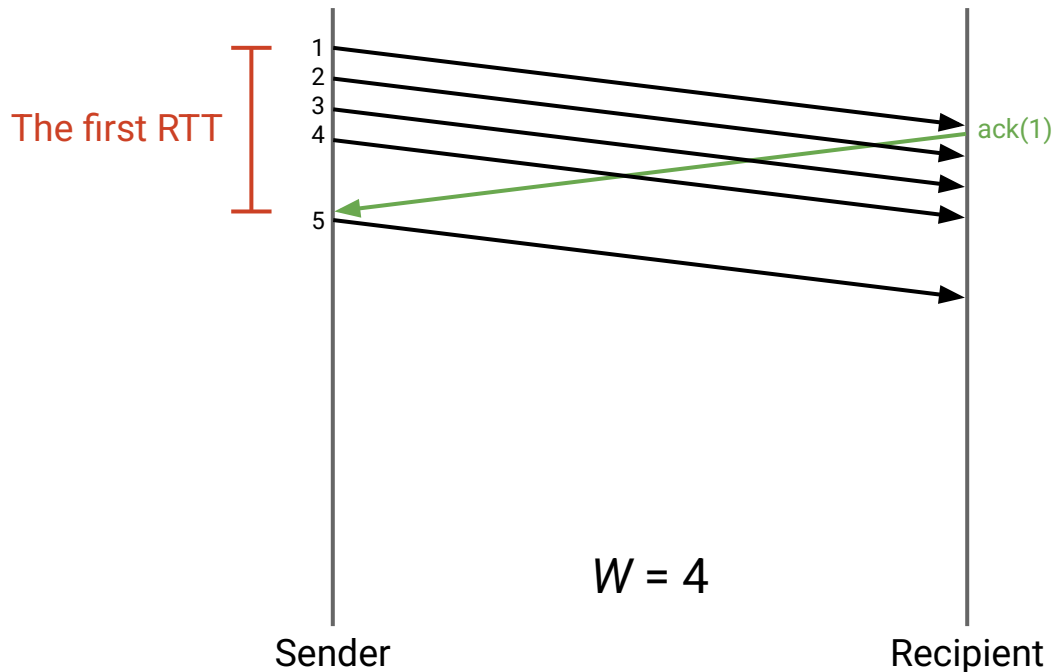
How to compute this W ?

- Suppose $RTT = 5$ seconds, and bandwidth = 10 packets per second.
- Let's focus on the very first RTT .
- For the first 5 seconds, no acks arrive.
- We want the sender to be constantly busy for all 5 seconds.
- How many packets can be sent in 5 seconds?
- $(5 \text{ seconds}) \times (10 \text{ packets per second}) = 50 \text{ packets}$.
- Thus, $W = 50$ packets keeps the sender busy.
- $W < 50$ means the sender reaches the limit before any acks, and has to sit idle.

More generally, to fill the pipe: $W = RTT \times \text{bandwidth}$.

Setting Window Size (1/3): Filling the Pipe

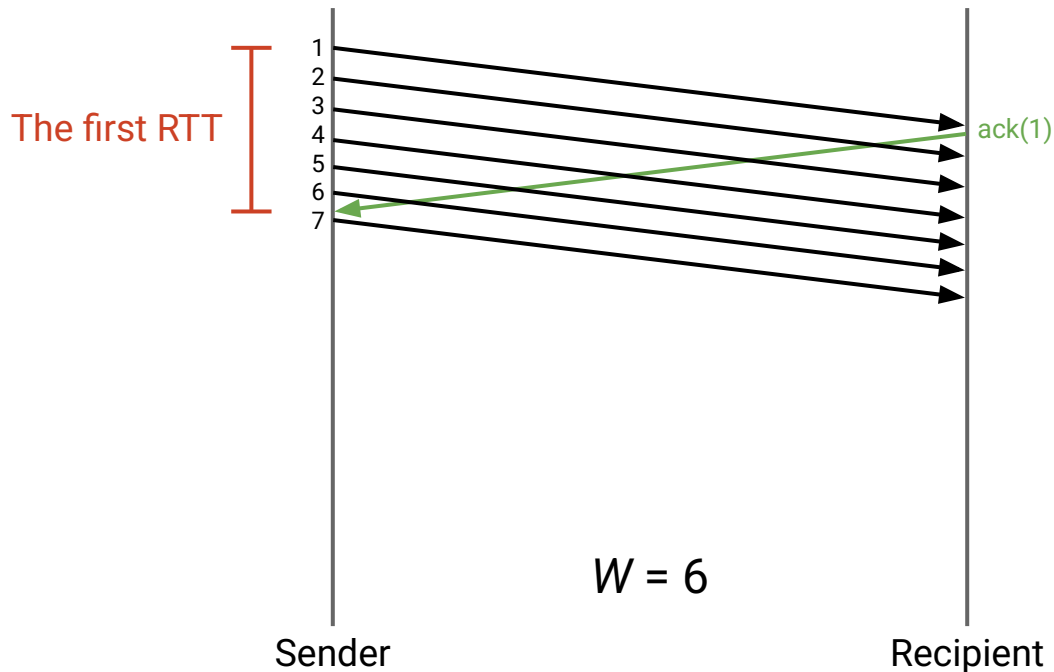
After sending the maximum $W = 4$ packets, the sender sits idle, waiting for `ack(1)`.
This window size is too small to fill the pipe.



Setting Window Size (1/3): Filling the Pipe

The sender is busy the entire RTT sending $W = 6$ packets.

ack(1) arrives just as the last of the 6 packets leaves.



Setting Window Size (1/3): Filling the Pipe

To fill the pipe: $W = \text{RTT} \times \text{bandwidth}$.

What exactly is the bandwidth?

- The minimum ("bottleneck") link bandwidth along the path.
- This tells us how fast packets can be sent (without clogging the network).



Setting Window Size (1/3): Filling the Pipe

To fill the pipe: $W = \text{RTT} \times \text{bandwidth}$.

What exactly is the bandwidth?

- The minimum ("bottleneck") link bandwidth along the path.
- This tells us how fast packets can be sent (without clogging the network)

This gives us an *upper bound* on the desired size of W .

- The other two conditions (don't overload recipient and network) might impose stricter limits and decrease W .

Setting Window Size (1/3): Filling the Pipe

To fill the pipe: $W = \text{RTT} \times \text{bandwidth}$.

What exactly is the bandwidth?

- The minimum ("bottleneck") link bandwidth along the path.
- This tells us how fast packets can be sent (without clogging the network)

This gives us an *upper bound* on the desired size of W .

- The other two conditions (don't overload recipient and network) might impose stricter limits and decrease W .

We could rewrite this in terms of bytes, not packets.

- $W \times (\text{packet size}) = \text{RTT} \times \text{bandwidth}$
- W = window size, in packets, and $W \times (\text{packet size})$ = window size, in bytes.
- Bandwidth now measured in bits/second, not packets/second.

Avoiding Overload: Flow Control and Congestion Control

Lecture 11, CS 168, Fall 2026

Transport Layer Design Goals

Implementing Reliability

- Single Packet
- Multiple Packets (Windows)
- **Avoiding Overload
(Flow and Congestion Control)**
- Smarter Acknowledgments
- Detecting Loss Early
- Alternate Designs

Setting Window Size

How big should the window be?

Pick window size W to balance three goals:

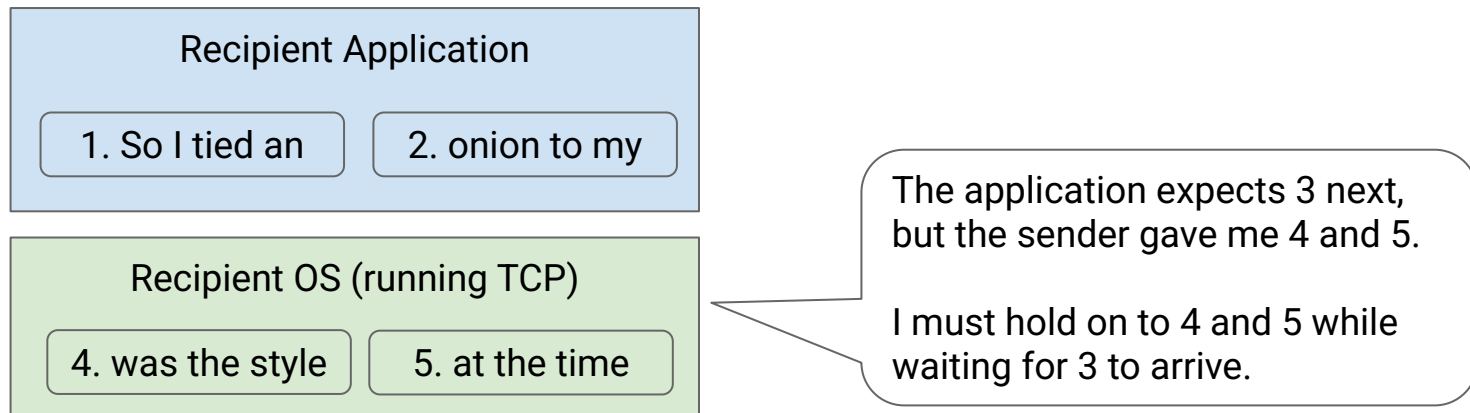
1. Take advantage of network capacity ("fill the pipe")... (set $W = \text{RTT} \times \text{bandwidth}$)
2. ...but don't overload the recipient (flow control)...
3. ...and don't overload the network (congestion control).

Setting Window Size (2/3): Don't Overload Recipient (Flow Control)

Consider the recipient:

- TCP might receive packets out-of-order, but can only deliver packets to the application in order.
- The recipient must *buffer* incoming packets that are out of order.
- Packets stay in the buffer until all "missing" packets arrive.

We have to make sure the recipient doesn't run out of buffer space.



Setting Window Size (2/3): Don't Overload Recipient (Flow Control)

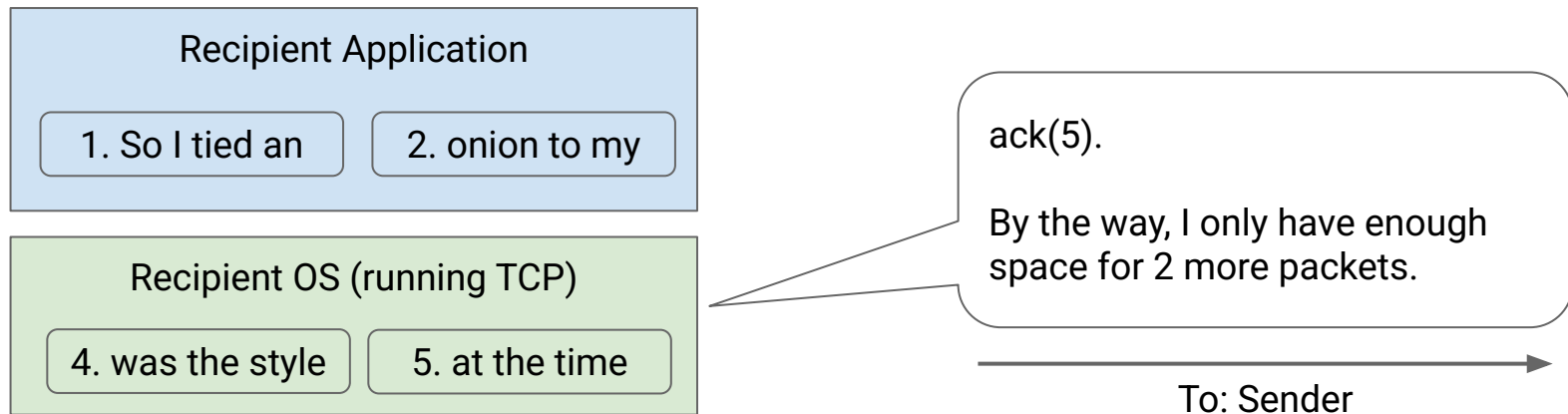
Flow control ensures the recipient has enough buffer space for out-of-order packets.

Recipient tells the sender how much space it has left (remember, feedback is key!)

- The size of remaining space is called the **advertised window**.
- This value is carried in ack messages.

Sender adjusts its window accordingly.

- Number of packets in flight must be less than the recipient's advertised window.



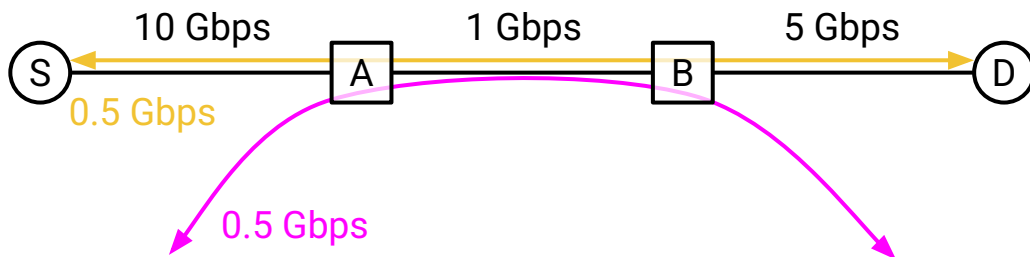
Setting Window Size (3/3): Don't Overload Network (Congestion Control)

Previously, we set W to fully consume the bottleneck link bandwidth.

- Example: S–D would set W to send data at 1 Gbps.

In practice, the bottleneck is shared with other flows.

- Example: With the red flow, S–D should only send at 0.5 Gbps.
- Each sender should only consume *its share* of the bandwidth.
- But, how do we compute this share?



Recall: $W = \text{RTT} \times \text{bandwidth}$.

So W and bandwidth are proportional (roughly speaking).

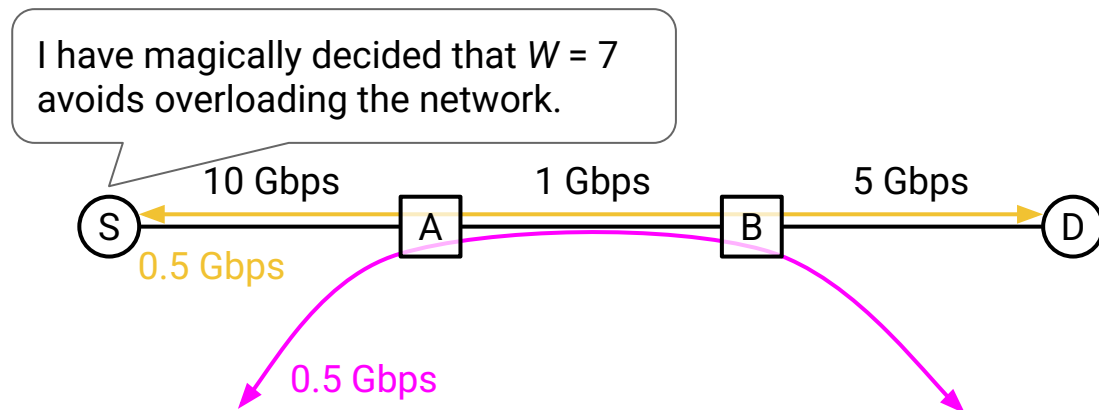
Setting Window Size (3/3): Don't Overload Network (Congestion Control)

The sender's TCP code implements a **congestion control algorithm** that dynamically computes the sender's share of bandwidth.

- The output of the algorithm is a **congestion window** (cwnd).
- The algorithm balances a few goals (e.g. performance, avoid overload, fairness).

More about the algorithms later.

- For today: The sender runs magic algorithm and outputs a congestion window.



Recall: $W = \text{RTT} \times \text{bandwidth}$.

So W and bandwidth are proportional (roughly speaking).

Setting Window Size

How big should the window be?

Pick window size W to balance three goals:

1. Take advantage of network capacity ("fill the pipe")... (set $W = \text{RTT} \times \text{bandwidth}$)
2. ...but don't overload the recipient (flow control)... (set $W = \text{recipient's advertised window}$)
3. ...and don't overload links (congestion control). (set $W = \text{sender's congestion window}$)

Setting Window Size

How big should the window be?

Pick window size W to balance three goals:

1. Take advantage of network capacity ("fill the pipe")... (set $W = \text{RTT} \times \text{bandwidth}$)
2. ...but don't overload the recipient (flow control)... (set $W = \text{recipient's advertised window}$)
3. ...and don't overload links (congestion control). (set $W = \text{sender's congestion window}$)

Ideally: W is set to the minimum of the 3 values.

In practice: W is the minimum of advertised window (2) and congestion window (3).

- Congestion window $\leq$ pipe-filling window.
Equal if we're the only connection, less than if sharing bandwidth with others.

Smarter Acknowledgments

Lecture 11, CS 168, Fall 2026

Transport Layer Design Goals

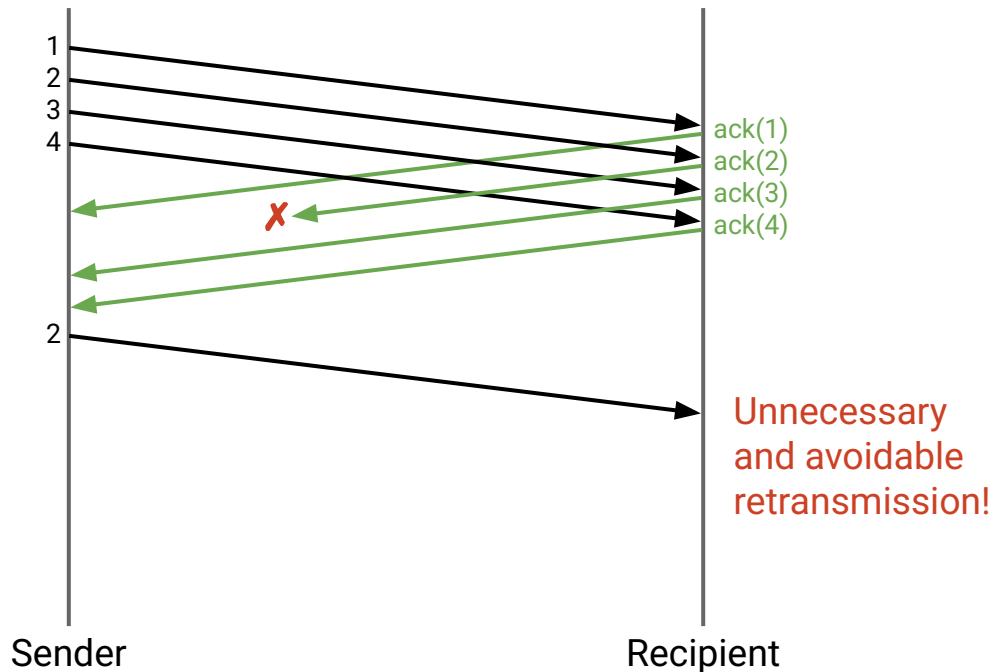
Implementing Reliability

- Single Packet
- Multiple Packets (Windows)
- Avoiding Overload
(Flow and Congestion Control)
- **Smarter Acknowledgments**
- Detecting Loss Early
- Alternate Designs

Ack Strategy (1/3): Individual Packet Acks

Our design so far: **Individual packet acks.**

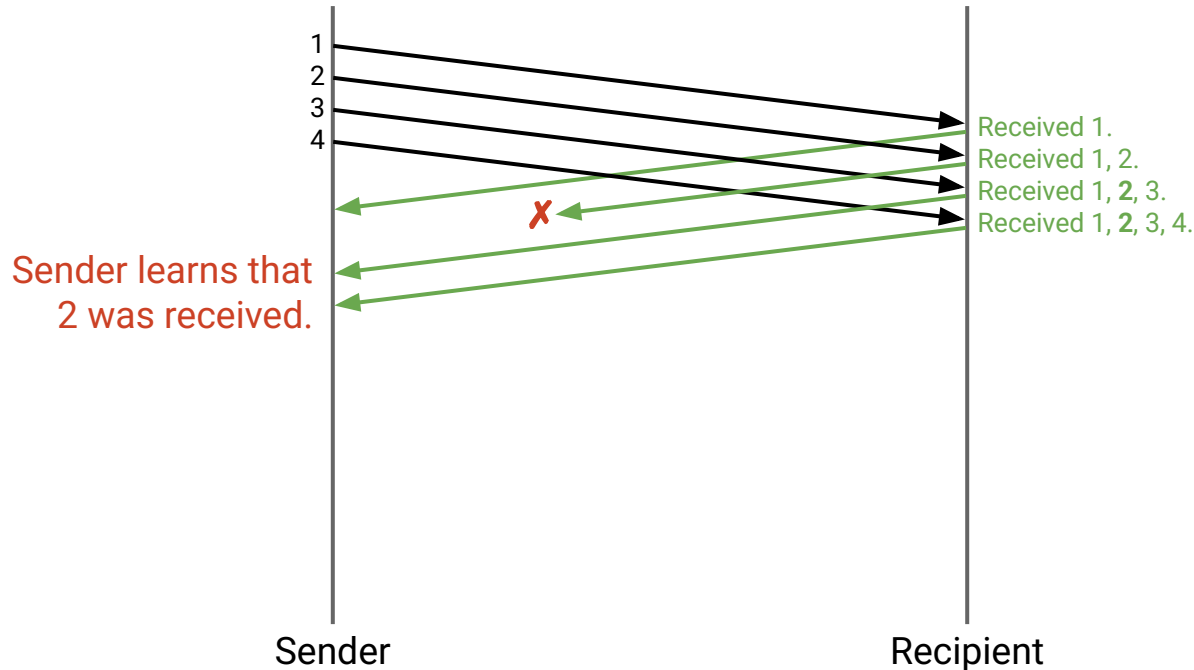
When you receive packet i , send $\text{ack}(i)$.



Ack Strategy (2/3): Full Information Acks

Smarter idea: **Full information acks.**

When you receive a packet, send an ack listing *every packet received*.



Ack Strategy (2/3): Full Information Acks

Full information ack format: "All packets 1~ n , plus *[list out all other packets]*."

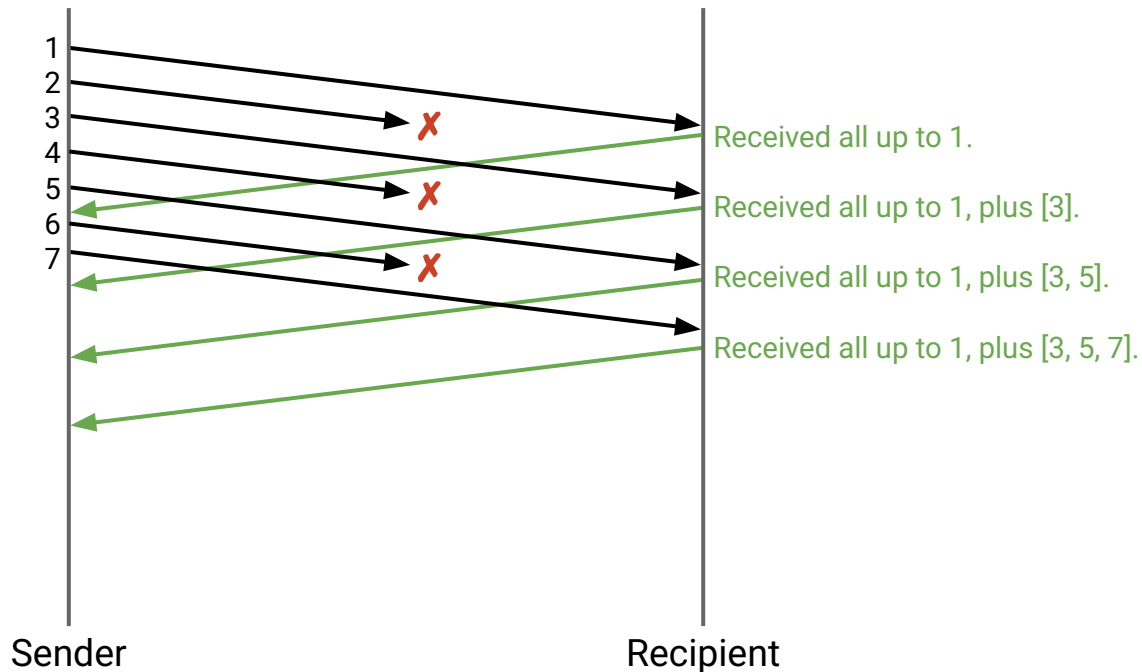
Example: "I received all packets up to 12, plus 14, 15, 16."

Problem: The list can get long.

If every even-numbered packet is dropped:

"Received all up to 1,
plus [3, 5, 7, ..., 999]."

Long.



Ack Strategy (3/3): Cumulative Acks

What TCP actually uses: **Cumulative acks.**

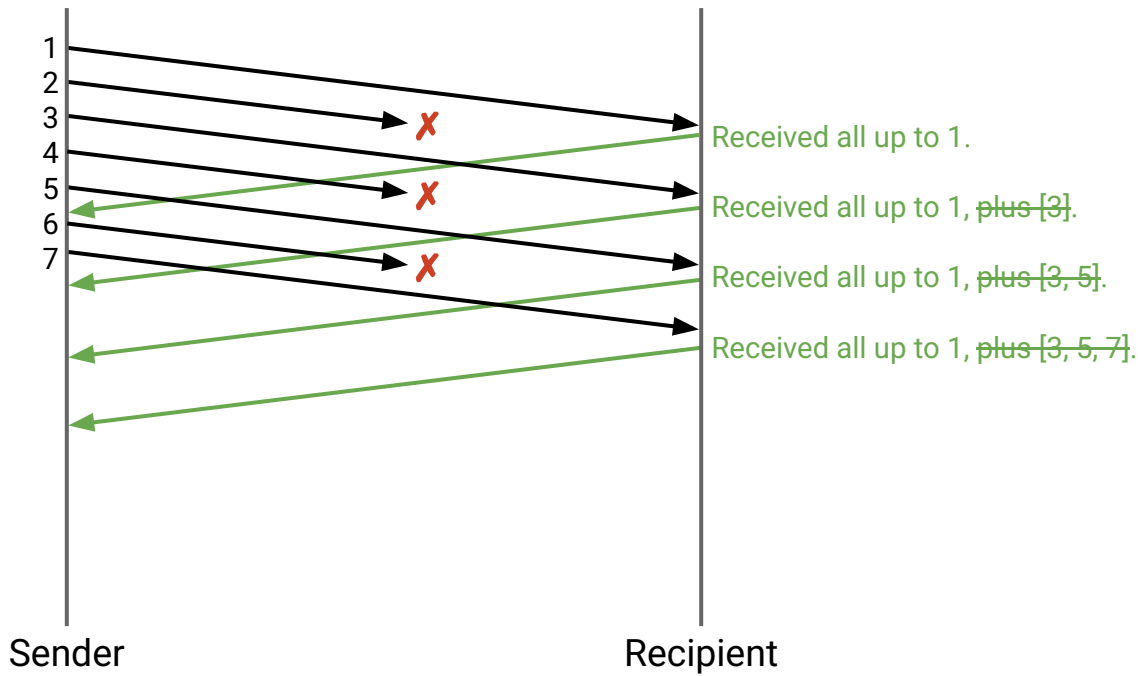
Example: "I received all packets up to 12, ~~plus 14, 15, 16.~~" (Full info, minus the list.)

Ack the highest sequence number for which all previous packets have been received.

If every even-numbered packet is dropped:

"Received all up to 1, ~~plus [3, 5, 7, ..., 999].~~"

Shorter, but more ambiguous.



Individual packet acks: Each ack corresponds to one packet.

- Pros: Compact, simple.
- Cons: Loss of ack packet always requires retransmission.

Full information acks: Each ack lists all packets received.

- Pros: Complete information on data packets. More resilient to ack loss.
- Cons: Could require sizable overhead in bad cases.

Cumulative acks: Each ack says, "all packets up to n received."

- Pros: Compact. More resilient to ack loss (than individual acks).
- Cons: Incomplete information on which packets arrived.

TCP uses cumulative acks.

Detecting Loss Early

Lecture 11, CS 168, Fall 2026

Transport Layer Design Goals

Implementing Reliability

- Single Packet
- Multiple Packets (Windows)
- Avoiding Overload
(Flow and Congestion Control)
- Smarter Acknowledgments
- **Detecting Loss Early**
- Alternate Designs

So far, we detect loss by waiting for a timeout.

- Timeout length is roughly RTT (a few milliseconds).

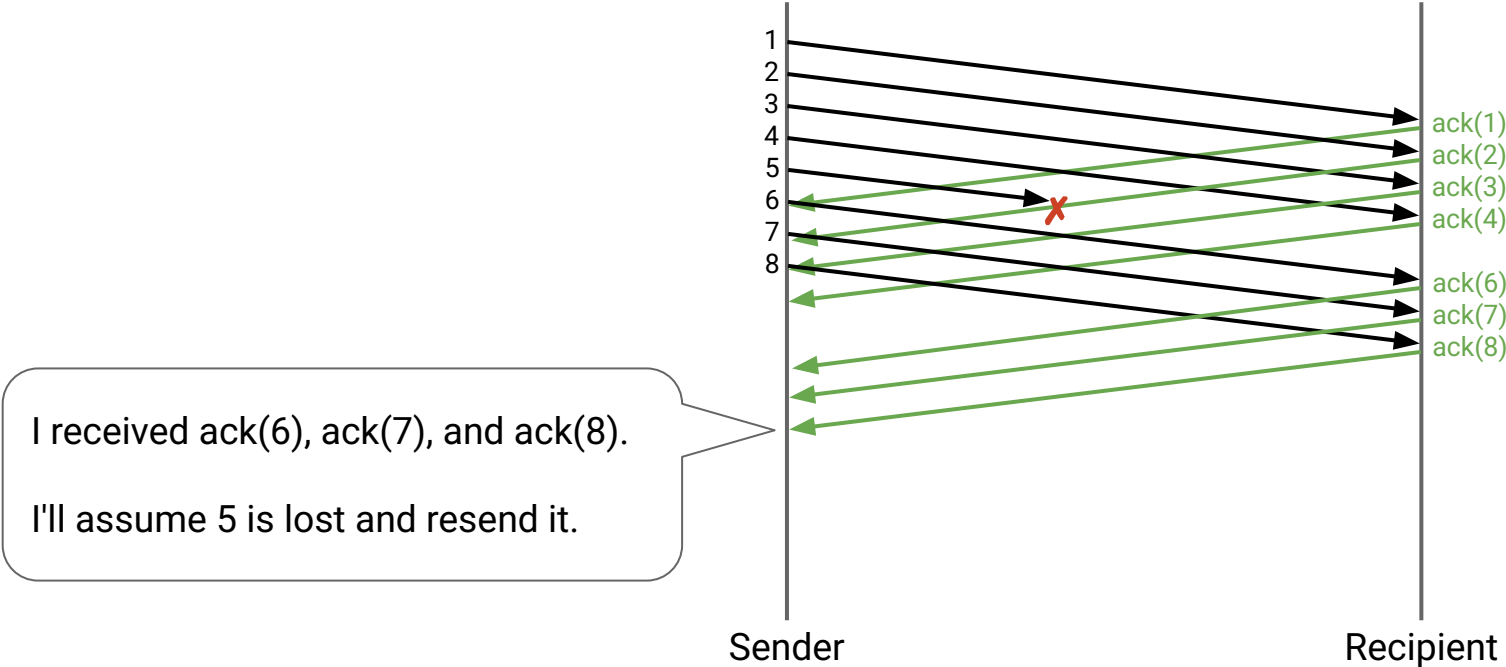
Another approach: Detect loss when acks for subsequent packets arrive.

- If we receive acks for 1, 2, 3, 4, 6, 7, 8, 9, we probably lost 5.
- Declare a packet lost if k subsequent packets are acked.
 - $k = 3$ is common.
- Packets arrive every few nanoseconds, so this approach usually detects loss way earlier.

Implementing ack-based loss detection depends on the type of ack we're using.

Ack-Based Loss Detection: Individual Acks

Individual acks: Declare a packet lost if k subsequent packets are acked.

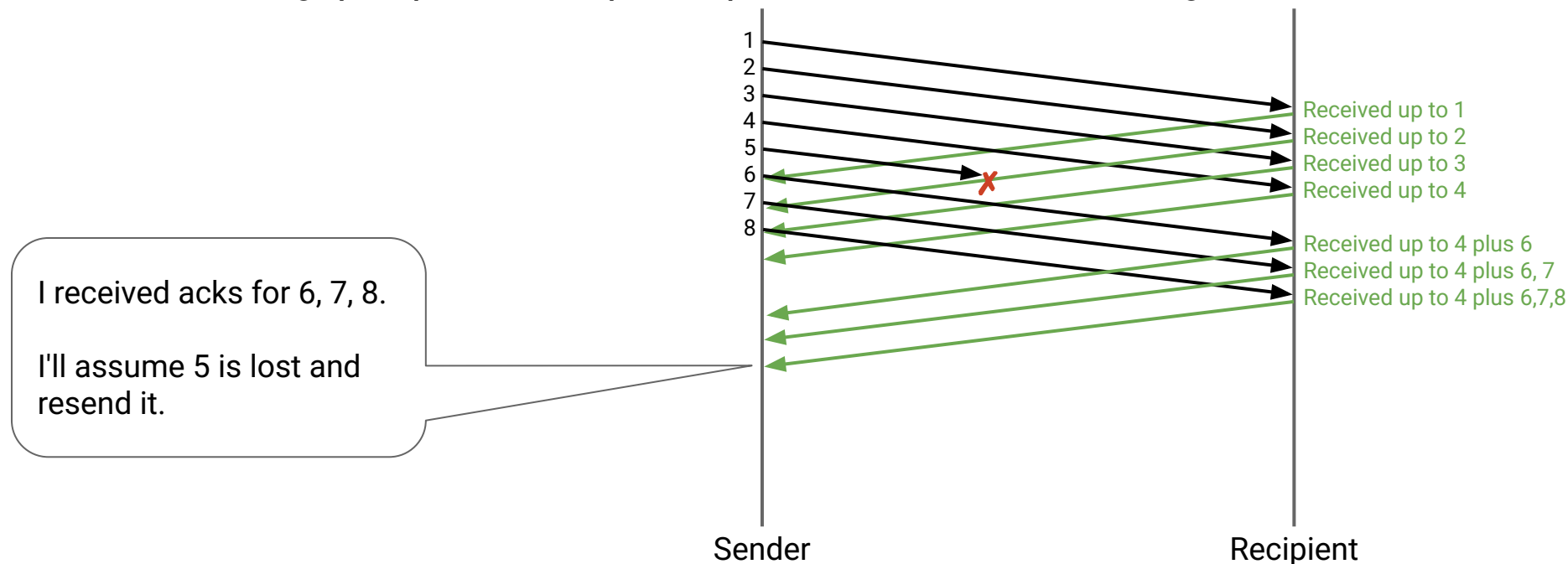


(Focus only on loss detection in this example; we'll ignore windows and the transmission of new packets 9, 10, ...)

Ack-Based Loss Detection: Full-Information Acks

Full-information acks: Declare a packet lost if k subsequent packets are acked.

We can see the gap in packets: "up to 4, plus 6, 7, 8" $\rightarrow$ 5 is missing.

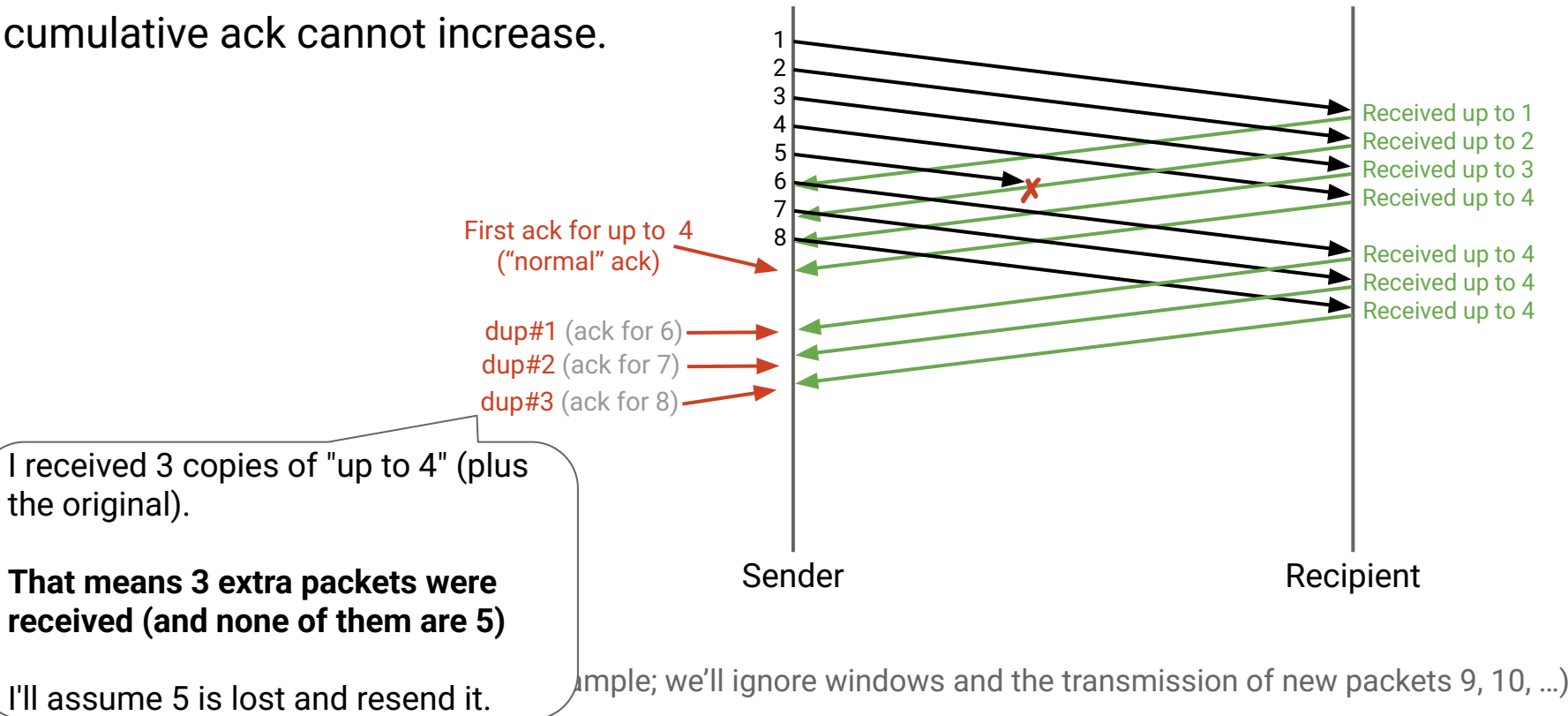


(Focus only on loss detection in this example; we'll ignore windows and the transmission of new packets 9, 10, ...)

Ack-Based Loss Detection: Cumulative Acks

Cumulative acks: Declare loss if k **duplicate acks** are received.

Duplicate acks: Later packets (6, 7, 8) received, but there's a gap, so the "up to 4" cumulative ack cannot increase.



Timeouts:

- We keep a separate timer for each packet.
- If a timer expires, resend the corresponding packet.

Subsequent acks: (This strategy is called "fast retransmit.")

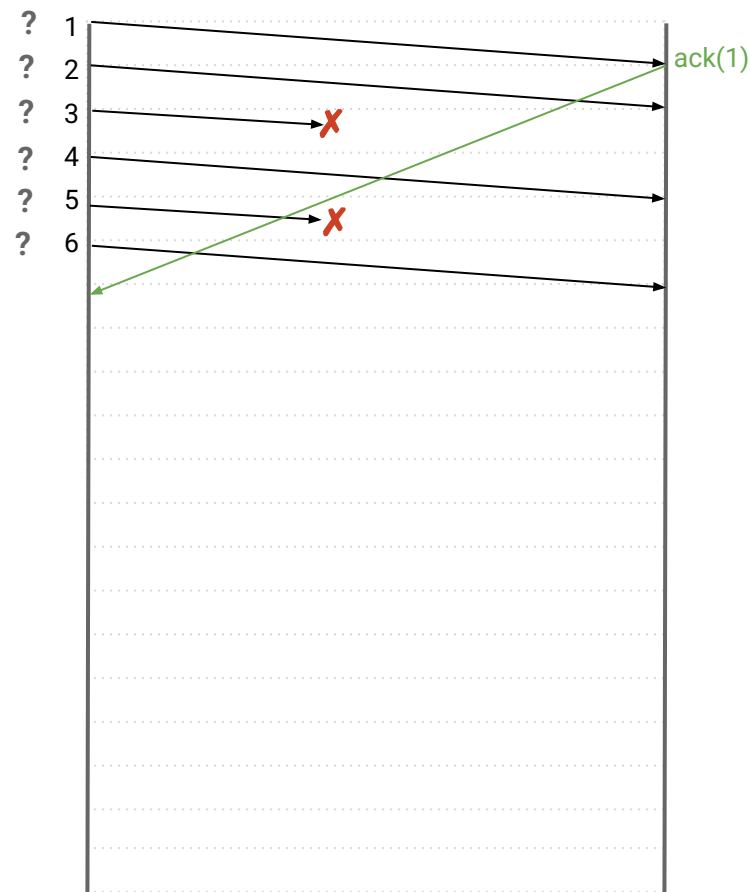
- Declare loss when k subsequent packets are acked

Resend the un-acked packet. But which packet is that?

- No ambiguity with individual and full-information acks.
- No ambiguity with cumulative acks and a single packet loss.
- Possibly ambiguous with *cumulative acks* and *multiple losses*.

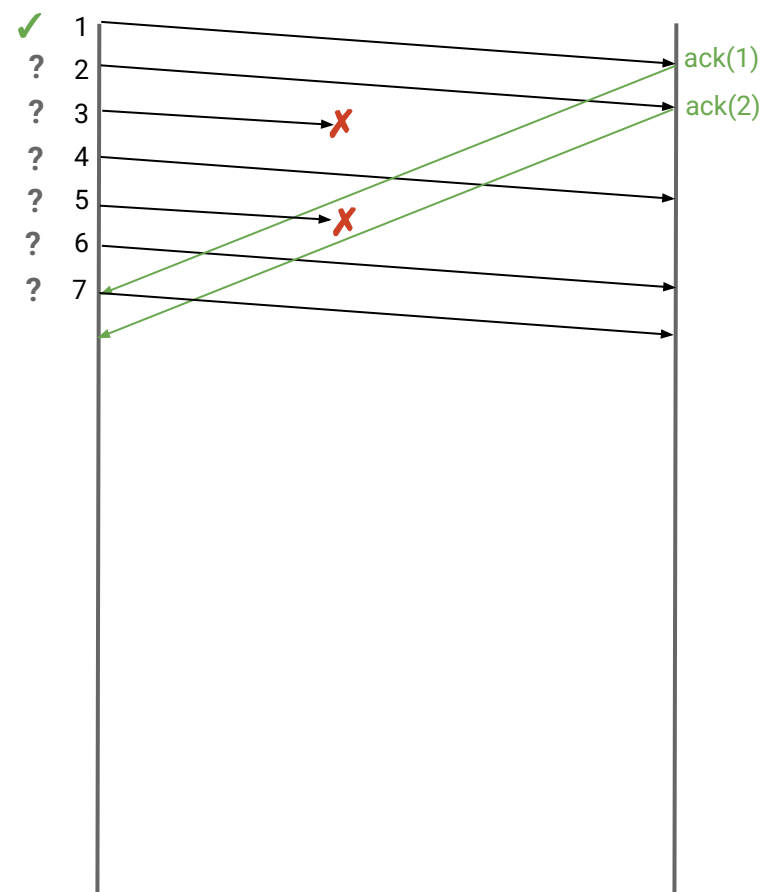
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 later acks*) ; $W = 6$ (*6 packets in flight*)



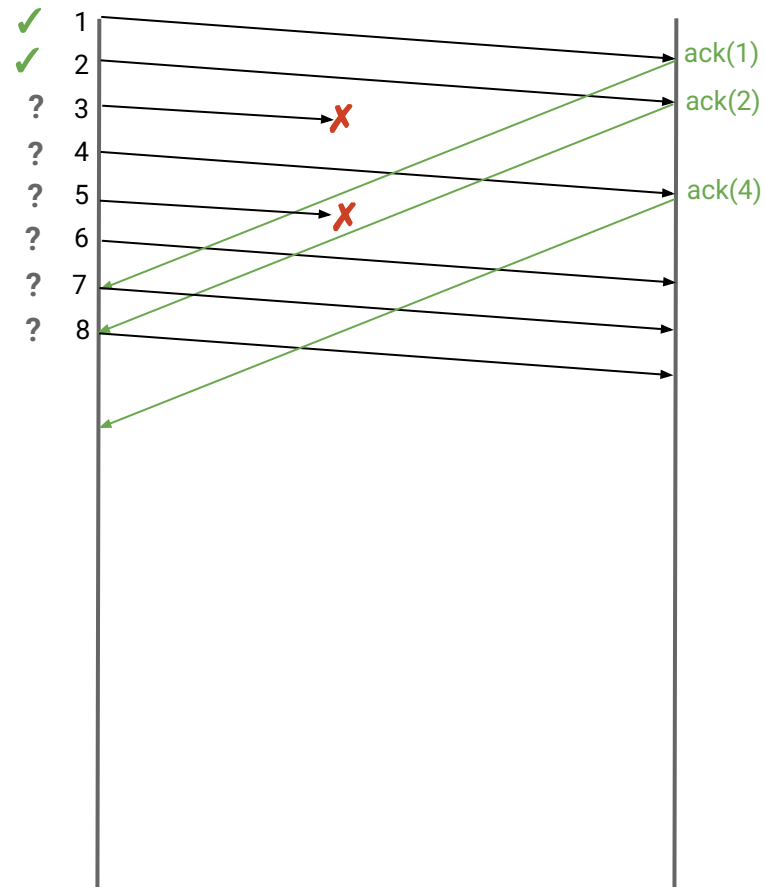
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 later acks*) ; $W = 6$ (*6 packets in flight*)



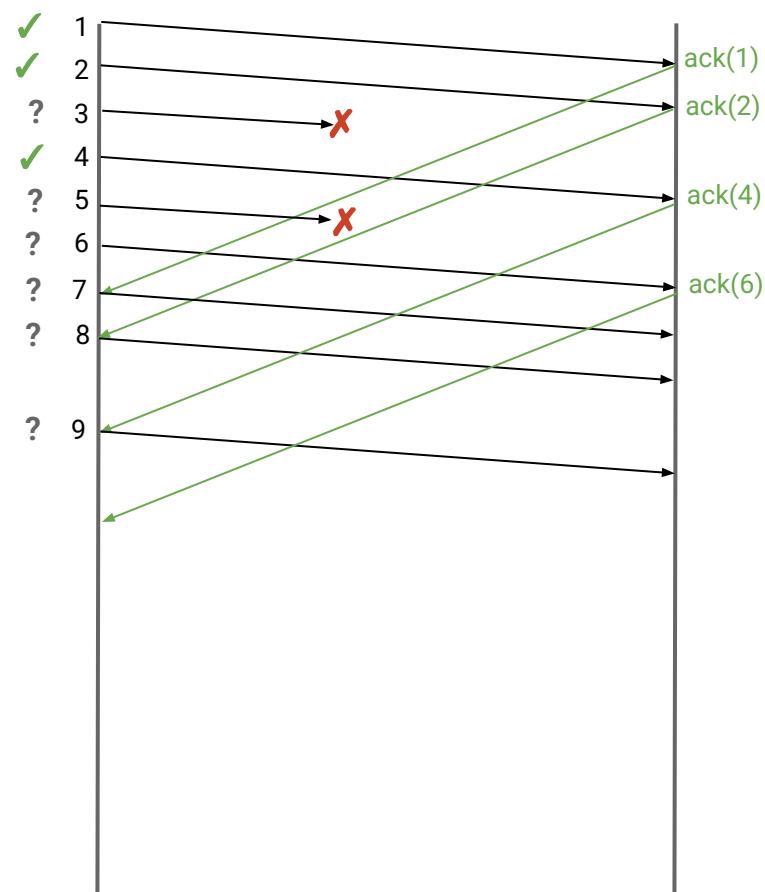
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 later acks*) ; $W = 6$ (*6 packets in flight*)



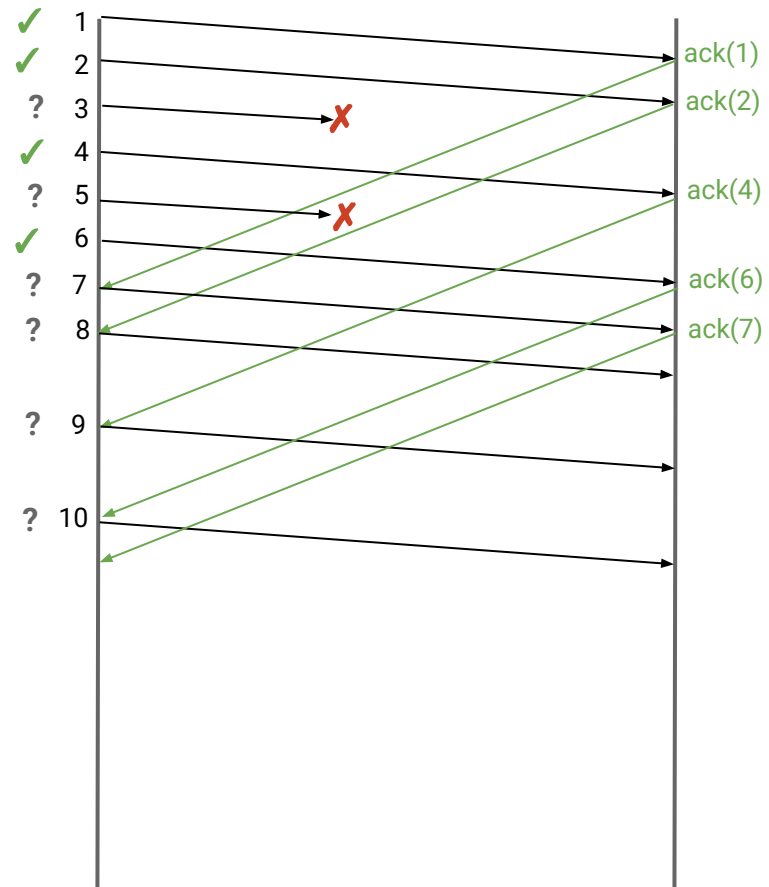
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 later acks*) ; $W = 6$ (*6 packets in flight*)



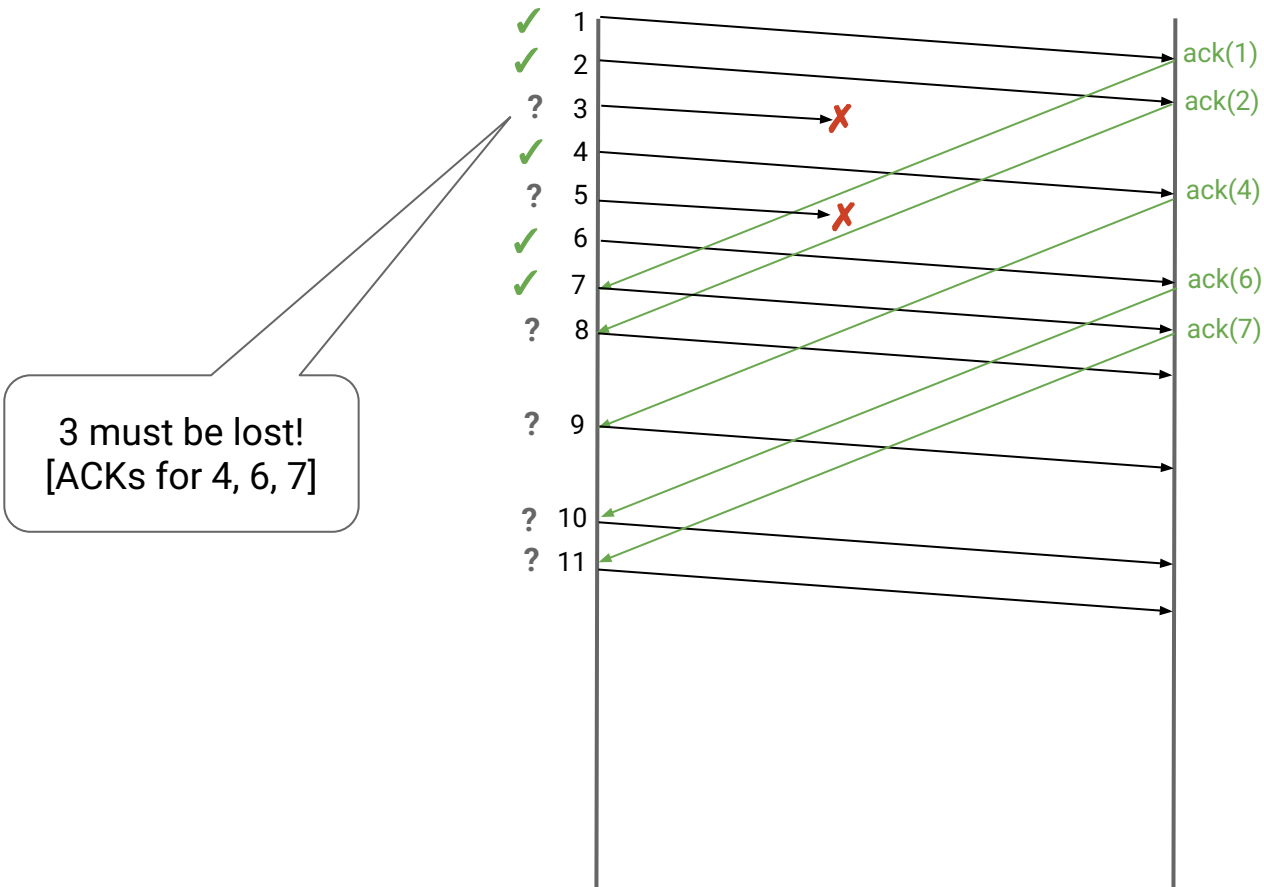
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 later acks*) ; $W = 6$ (*6 packets in flight*)



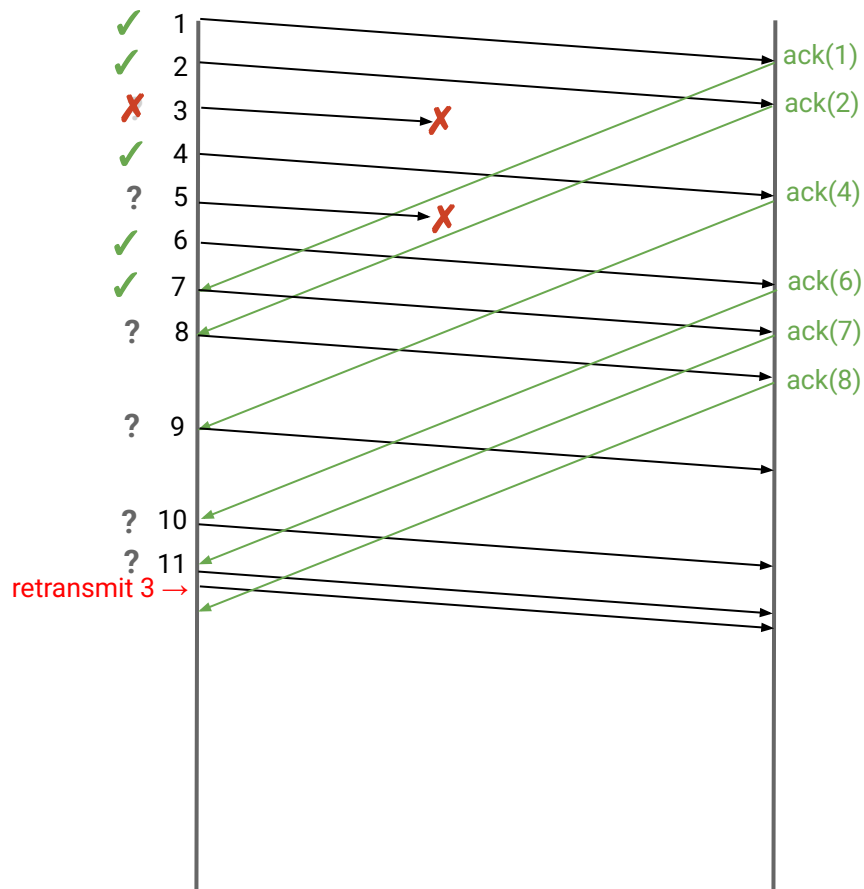
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (lost after 3 later acks) ; $W = 6$ (6 packets in flight)



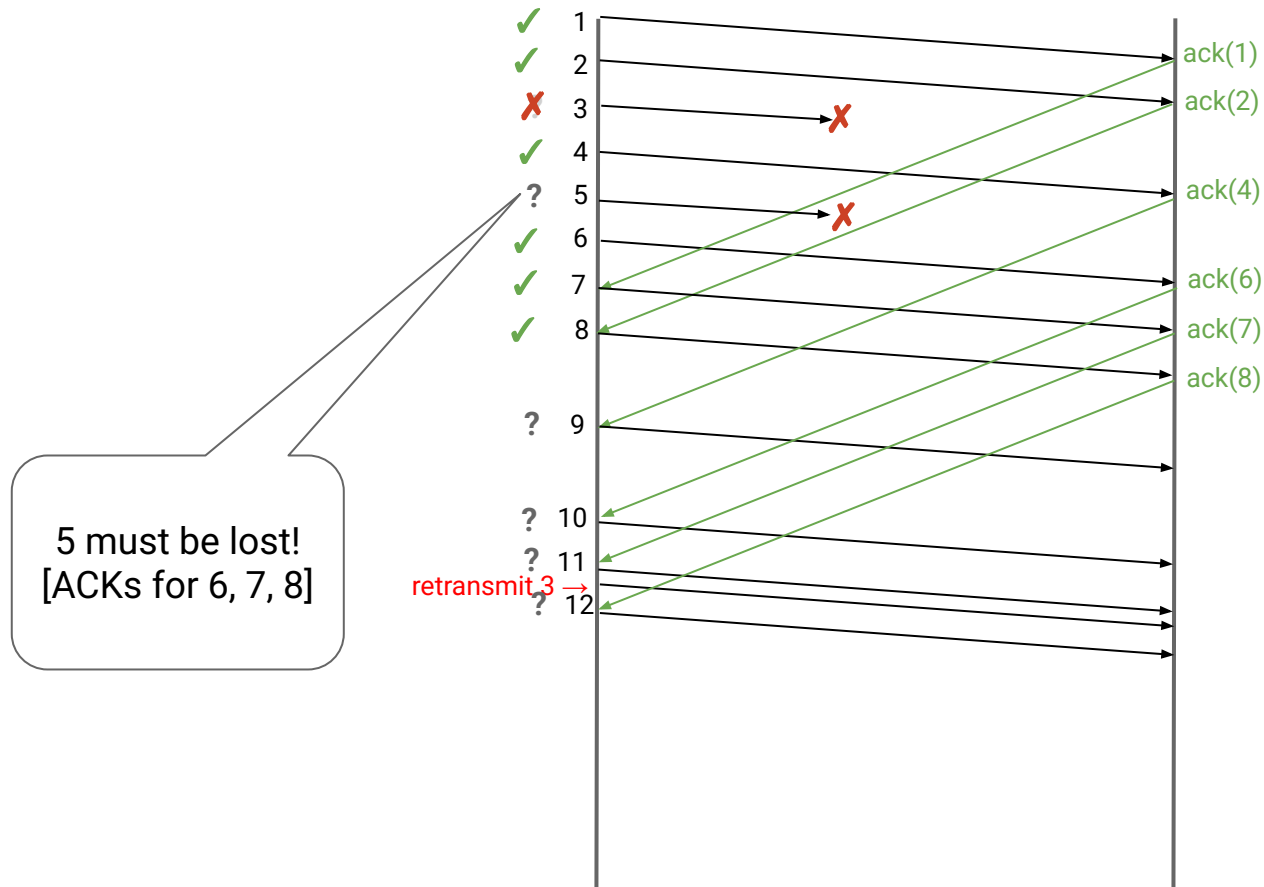
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (lost after 3 later acks) ; $W = 6$ (6 packets in flight)



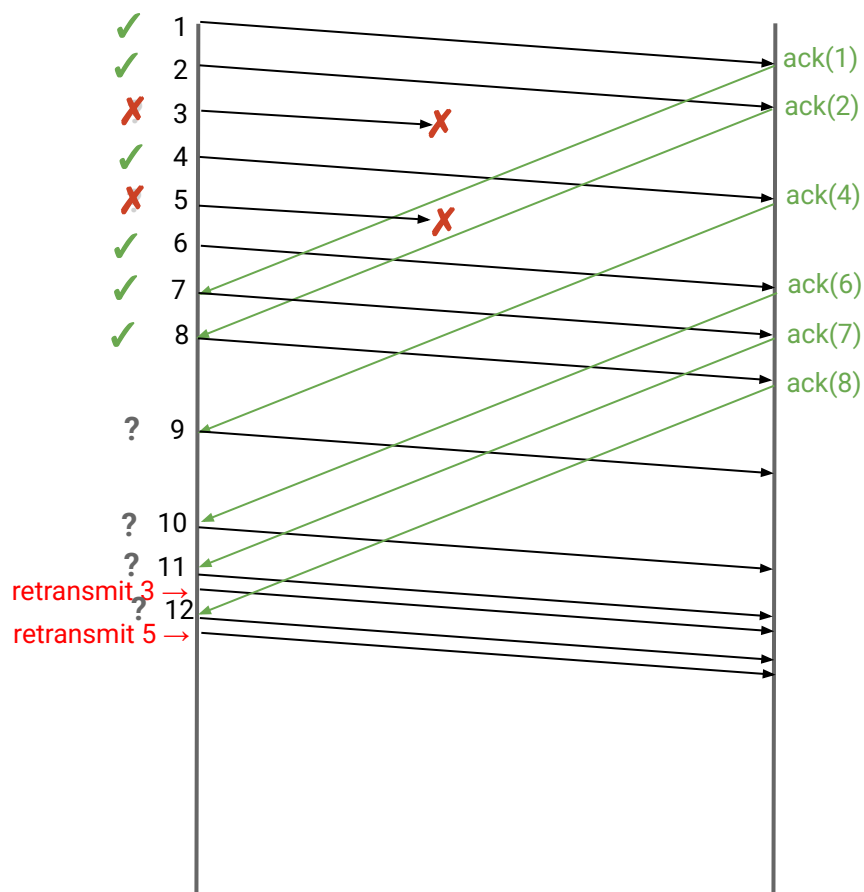
Responding to Loss: Individual Acks, Multiple Losses

Assumptions: $k = 3$ (lost after 3 later acks) ; $W = 6$ (6 packets in flight)



Responding to Loss: Individual Acks, Multiple Losses

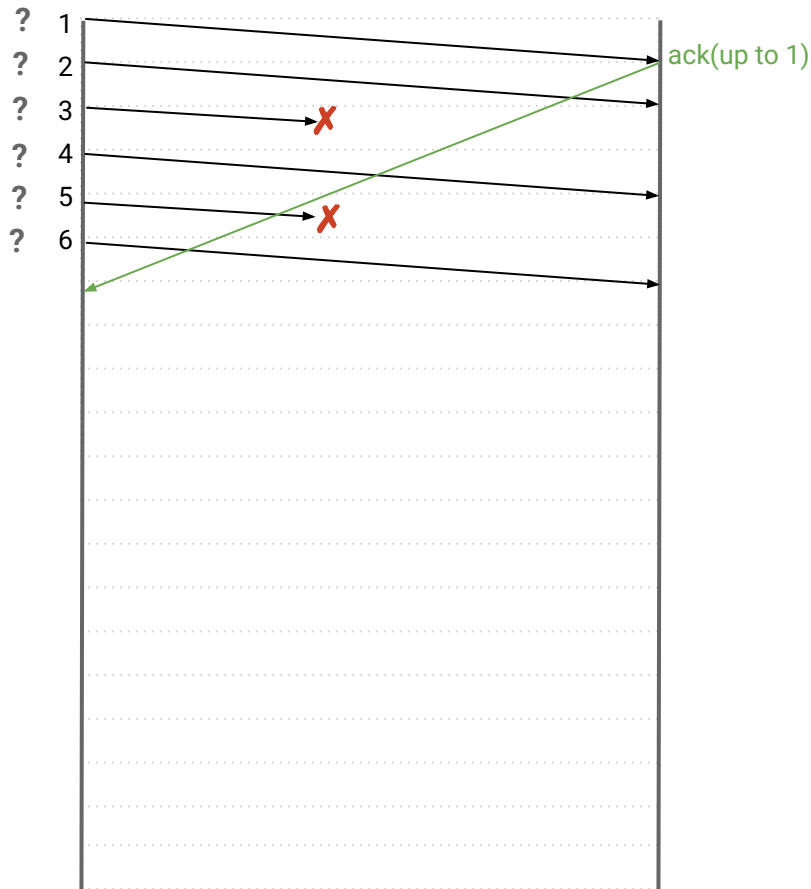
Assumptions: $k = 3$ (lost after 3 later acks) ; $W = 6$ (6 packets in flight)



Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 duplicate acks*) ; $W = 6$ (*6 packets in flight*)

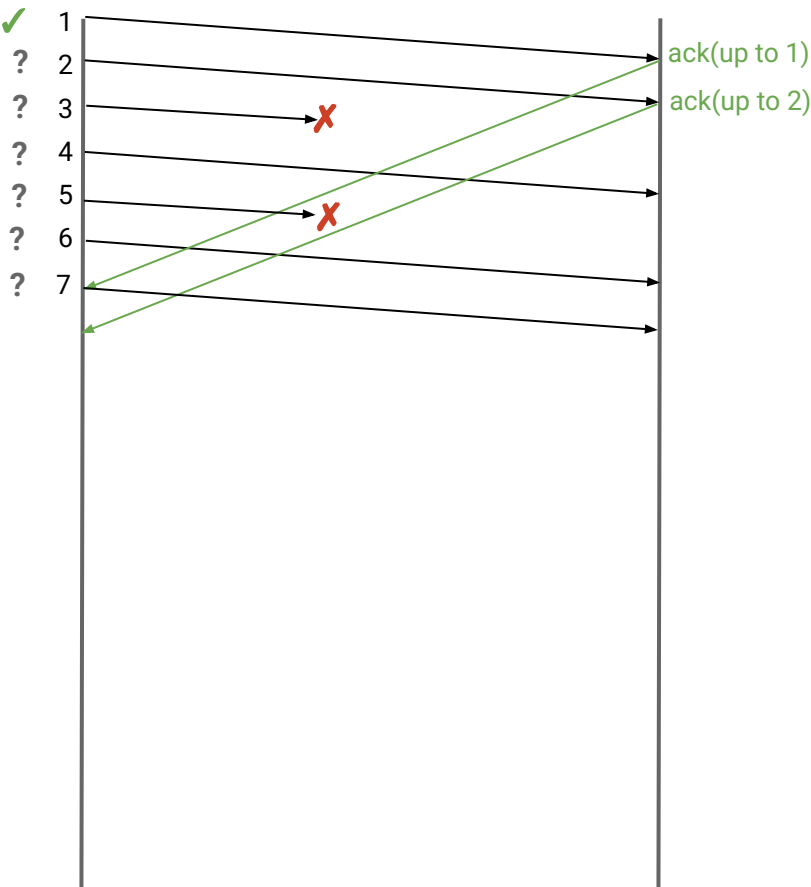
#duplicate ACKs = 0



Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 duplicate acks*) ; $W = 6$ (*6 packets in flight*)

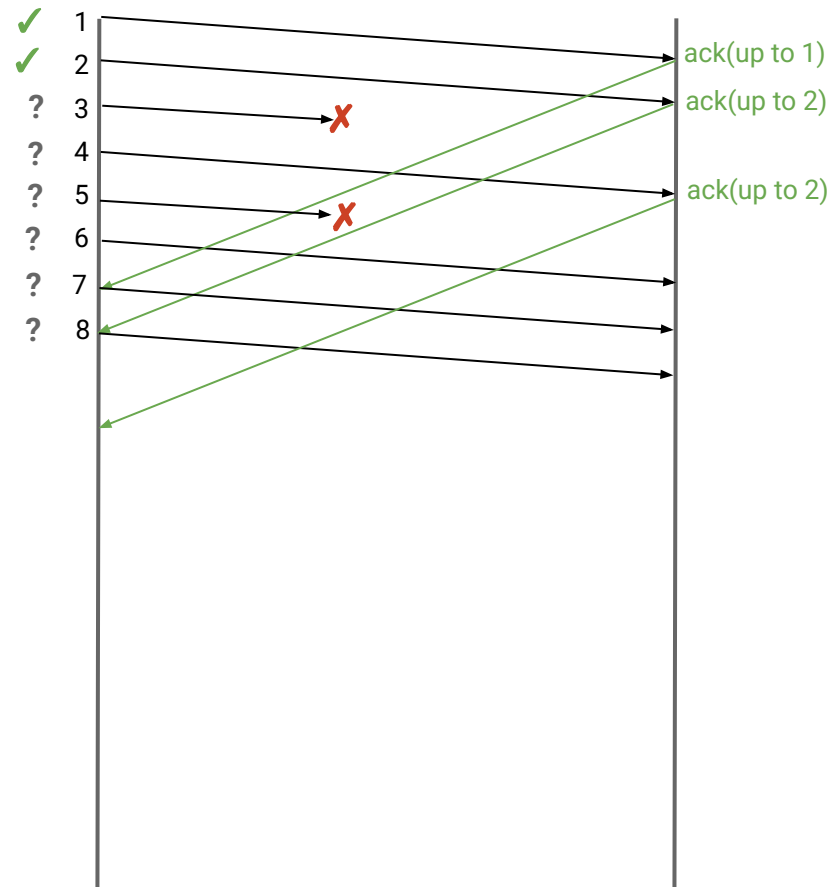
#duplicate ACKs = 0



Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 duplicate acks*) ; $W = 6$ (*6 packets in flight*)

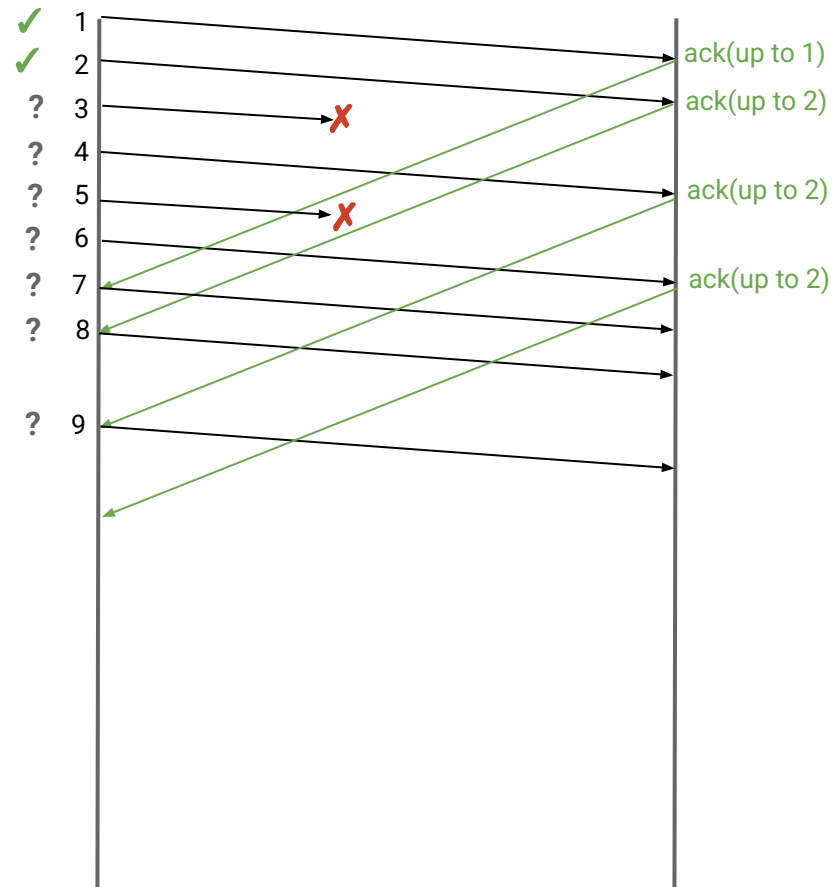
#duplicate ACKs = 0



Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 duplicate acks*) ; $W = 6$ (*6 packets in flight*)

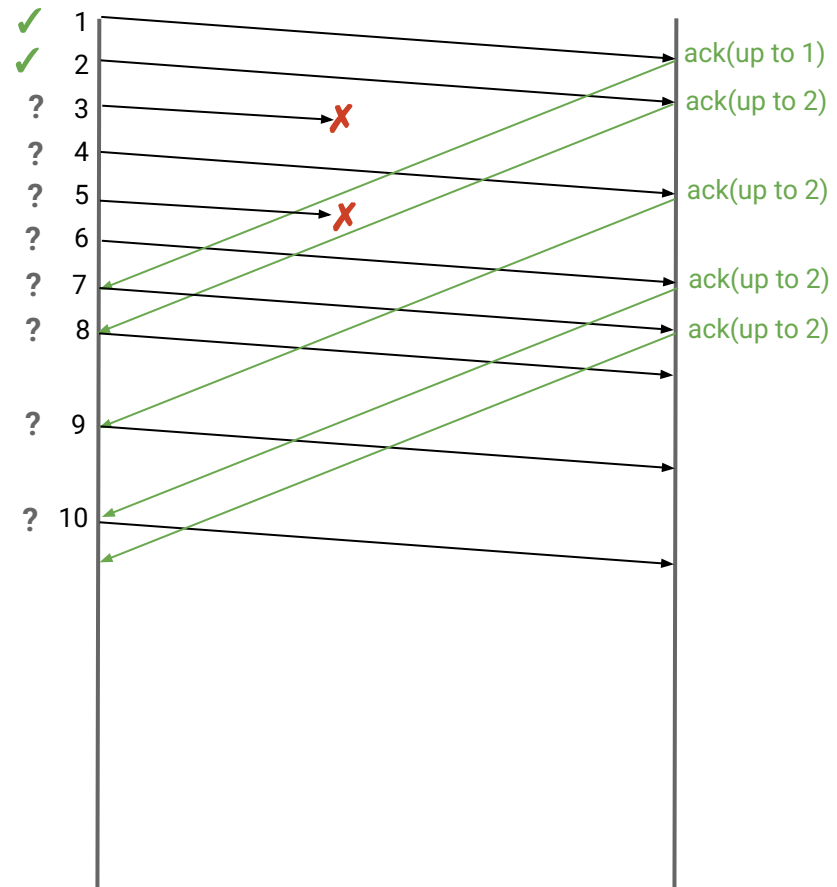
#duplicate ACKs = 1



Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (*lost after 3 duplicate acks*) ; $W = 6$ (*6 packets in flight*)

#duplicate ACKs = 2

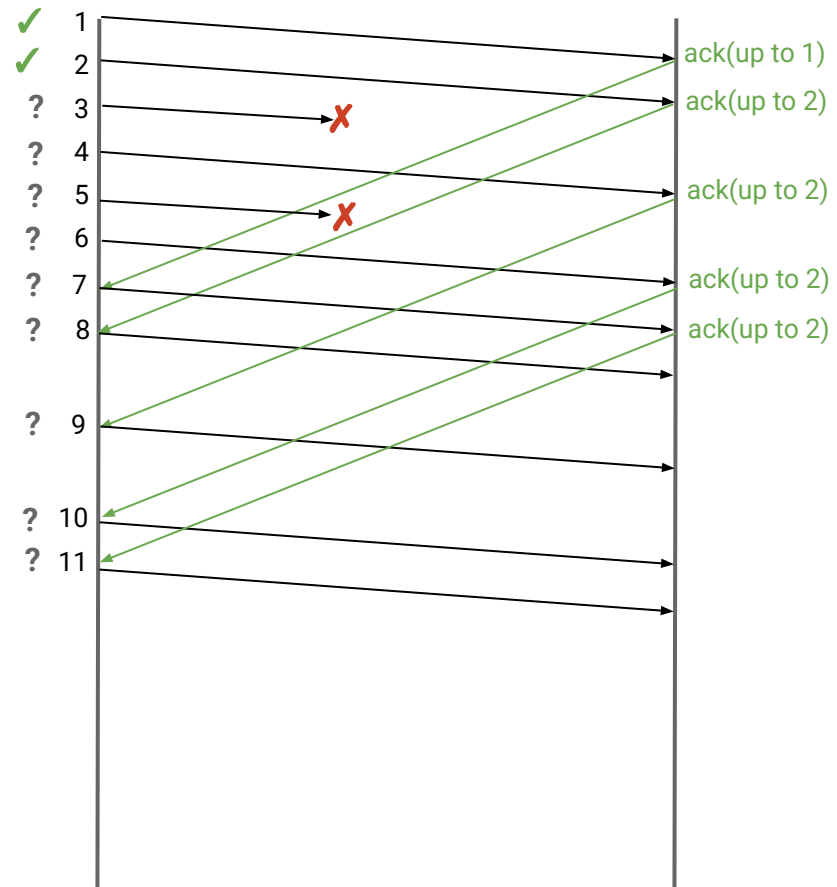


Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (lost after 3 duplicate acks) ; $W = 6$ (6 packets in flight)

#duplicate ACKs = 3

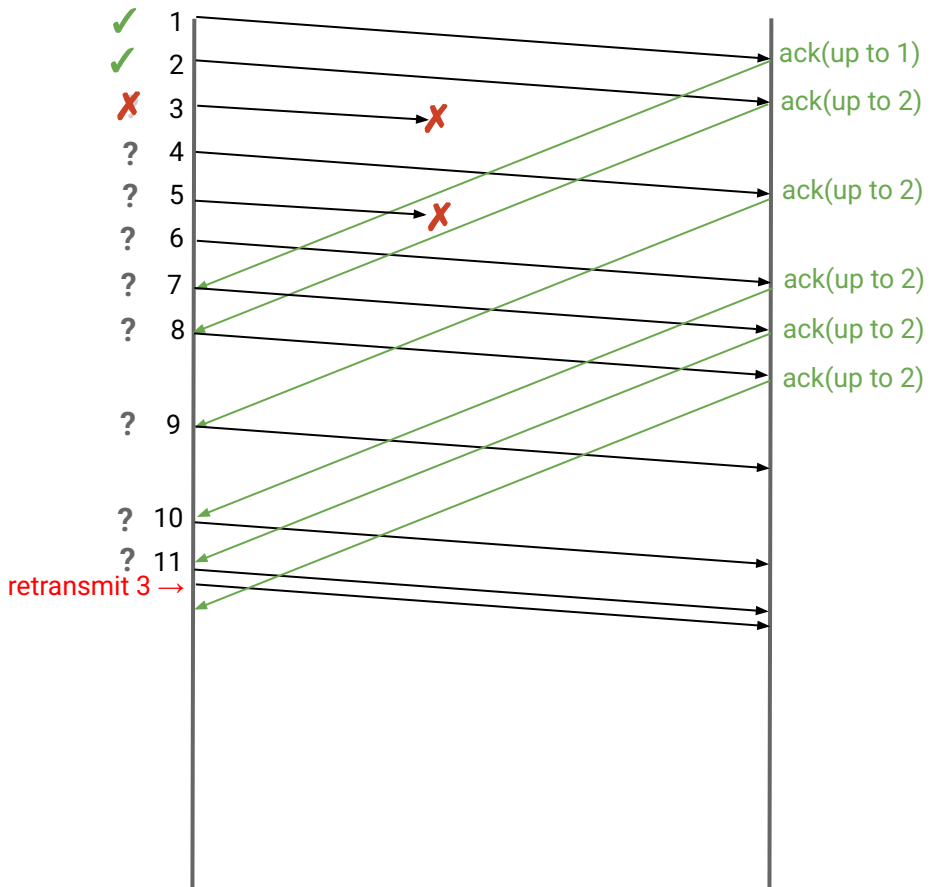
3 must be lost!



Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (lost after 3 duplicate acks) ; $W = 6$ (6 packets in flight)

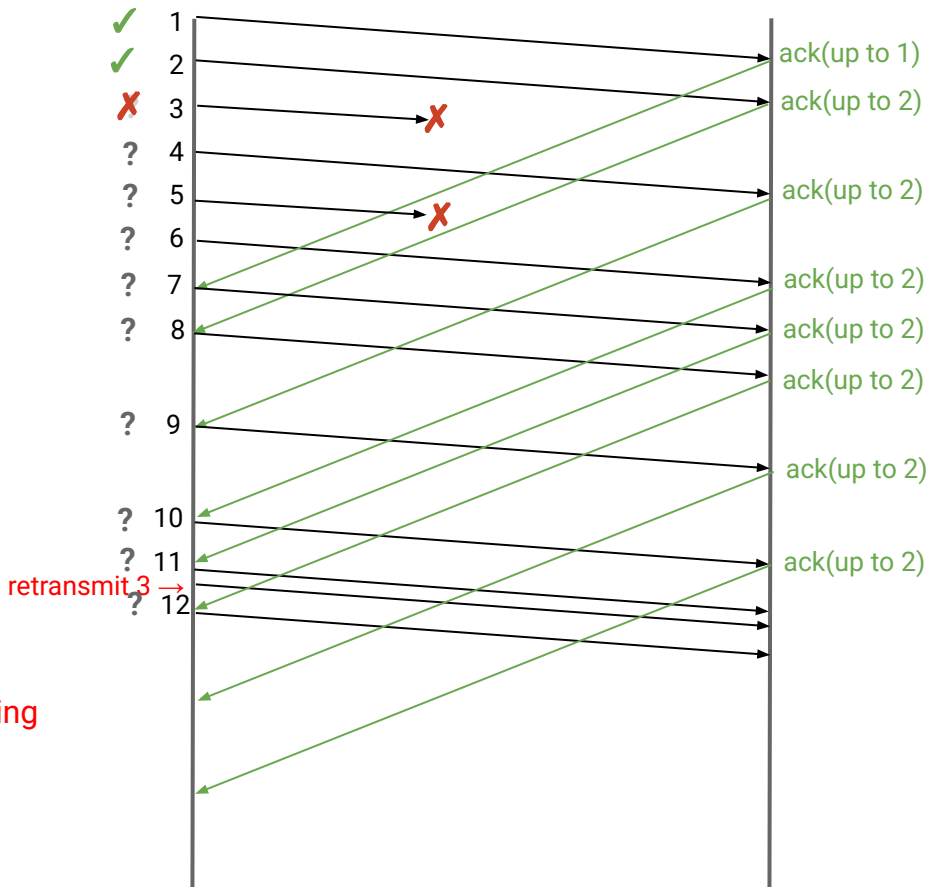
#duplicate ACKs = 3



Responding to Loss: Cumulative Acks, Multiple Losses

Assumptions: $k = 3$ (lost after 3 duplicate acks) ; $W = 6$ (6 packets in flight)

#duplicate ACKs = 4
#duplicate ACKs = 6



Sender has no way of knowing
that packet 5 is also lost!

Responding to Loss: Cumulative Acks, Multiple Losses

Problem: Cumulative acks don't tell the sender exactly which packets were received.

- Can tell *how many* packets to send.
 - Because #duplicate ACKs tells us how many packets were *received*
- But not *which* packets to (re)send

Problem: Cumulative acks don't tell the sender exactly which packets were received.

- Can tell *how many* packets to send.
 - Because #duplicate ACKs tells us how many packets were *received*
- But not *which* packets to (re)send
- Different TCP implementations handle this differently
 - E.g., look up selective acknowledgements (SACK) if interested
- The version we consider:
 - Sender does not reset #duplicate counter until a new ACK is received
 - I.e., ACK for new data
 - Sender does not try and guess whether/which additional packets are lost
 - Does adjust the window size to reflect #duplicates (future lecture)

We've covered all the basic building blocks of TCP.

- Checksums
- Acknowledgments
- Timeouts
- Retransmissions
- Sequence numbers
- Windows

Along the way, we sometimes had to pick one design out of several alternatives.

- Do we send a nack if the packet is corrupted? (*TCP: No.*)
- Do we use individual, full-information, or cumulative acks? (*TCP: Cumulative.*)
- Do we detect loss with timeouts, subsequent acks, or both? (*TCP: Both.*)

Other Design Choices

TCP makes specific design choices (e.g. cumulative acks).

If you make different choices, you'll get different reliability protocols.

- Stop-and-wait: Set $W = 1$.
- Go-Back-N is another reliability protocol.

Understanding how to design and evaluate a reliability protocol is more important than memorizing the details of any specific implementation!