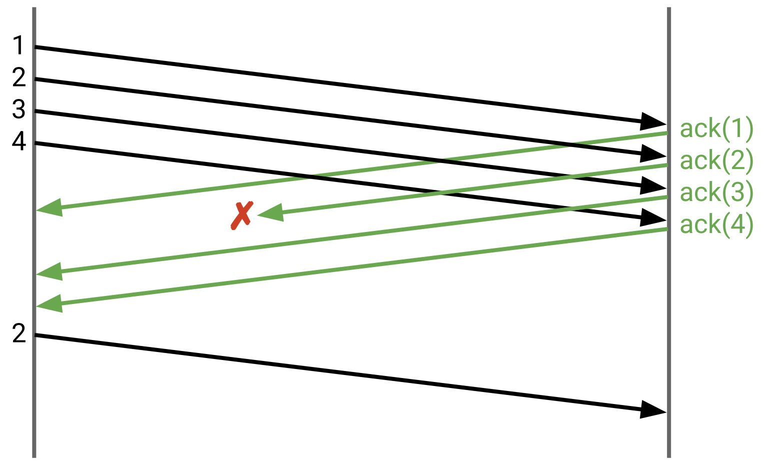


Lecture 11 (Transport 1)

Reliability

CS 168, Spring 2025 @ UC Berkeley

Slides credit: Sylvia Ratnasamy, Rob Shakir, Peyrin Kao

Transport Layer Design Goals

Lecture 11, CS 168, Spring 2025

Transport Layer Design Goals

Implementing Reliability

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

Transport Layer Protocols

Layer 3 (IP) gave us the ability to send packets anywhere in the Internet.

- Abstraction level: Packets individually sent through the network.
- Problem: IP offers best-effort delivery.
  - 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.

Layer 4 (Transport) builds extra features on top of Layer 3.

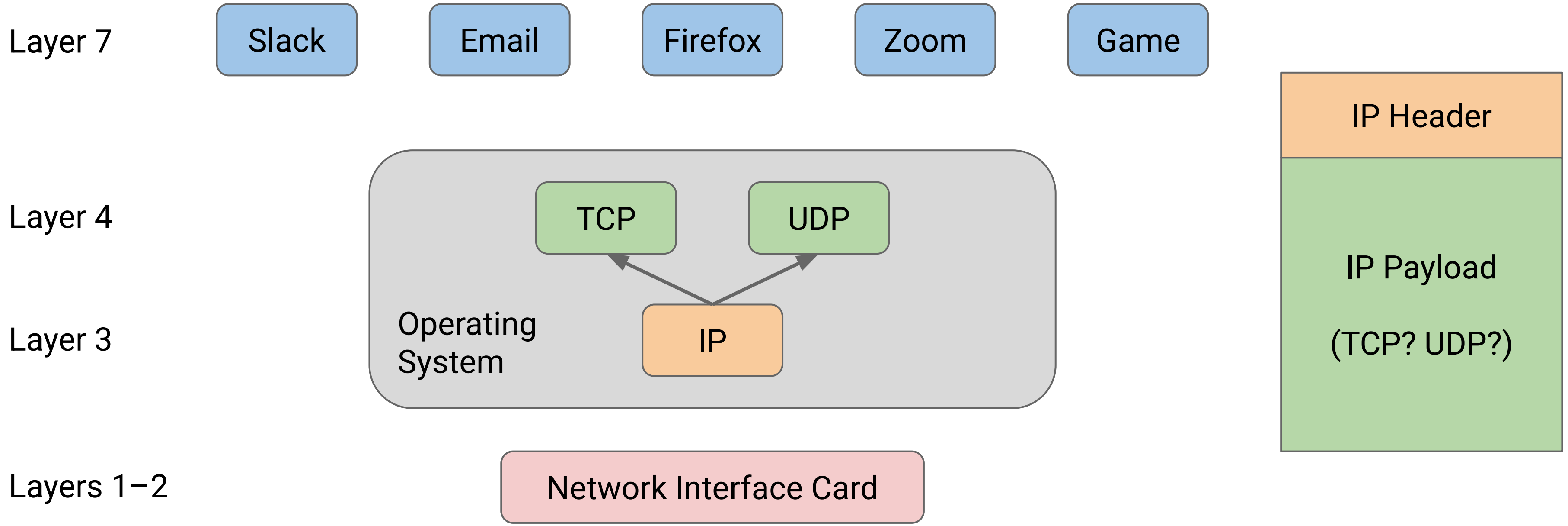
- **TCP** (Transmission Control Protocol): Adds *de-multiplexing* and *reliability*.
- **UDP** (User Datagram Protocol): Adds *de-multiplexing* only.
- Both protocols provide a more useful abstraction to programmers.

Let's check out each of these features.

Transport Layer Feature: De-Multiplexing with Ports

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

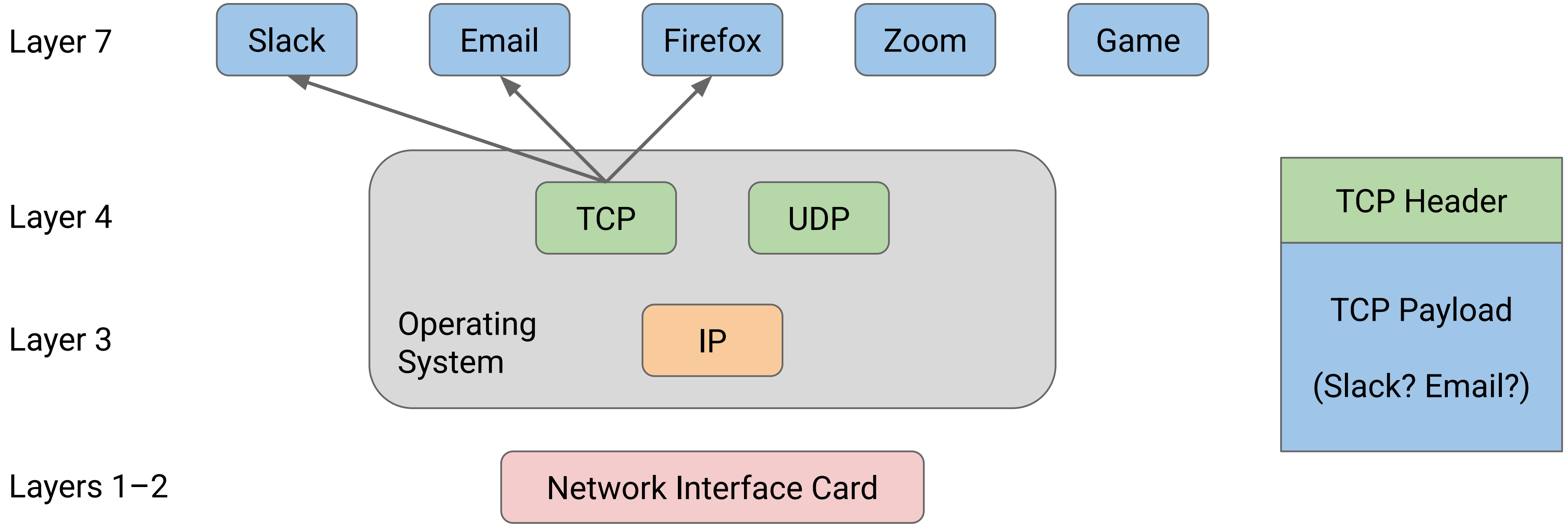
- Should I pass this packet to the TCP code or the UDP code?



Transport Layer Feature: De-Multiplexing with Ports

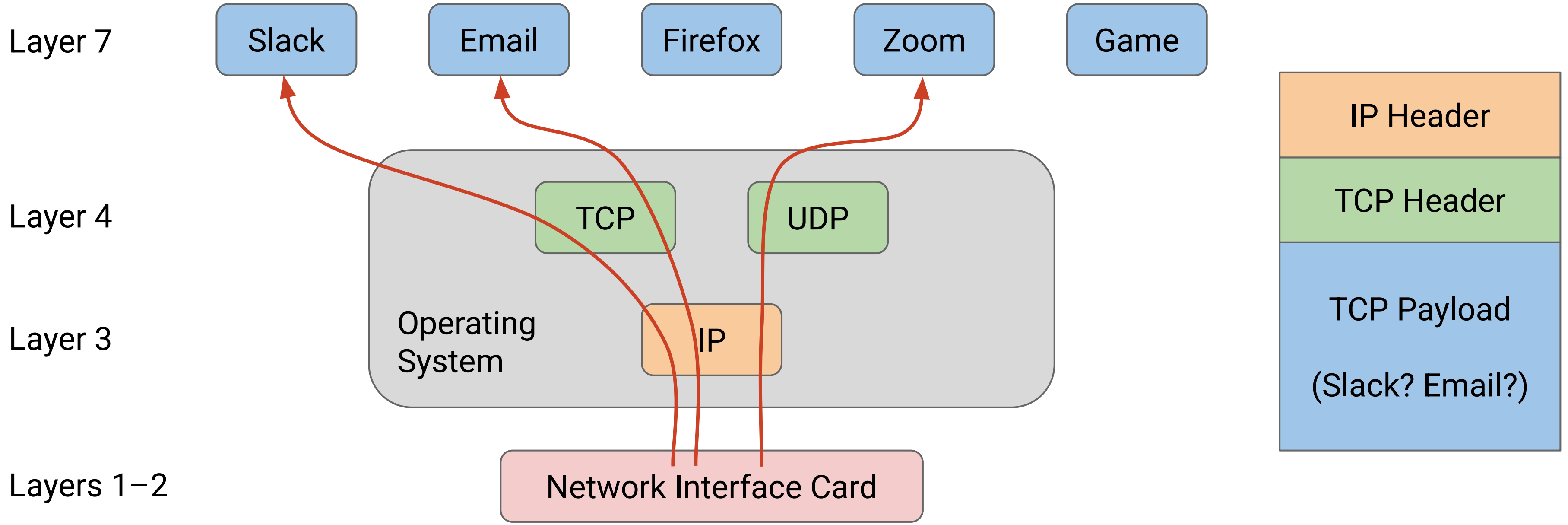
Port field in TCP header (L4) supports de-multiplexing between L7 applications.

- Which application should I pass this packet to?



Transport Layer Feature: De-Multiplexing with Ports

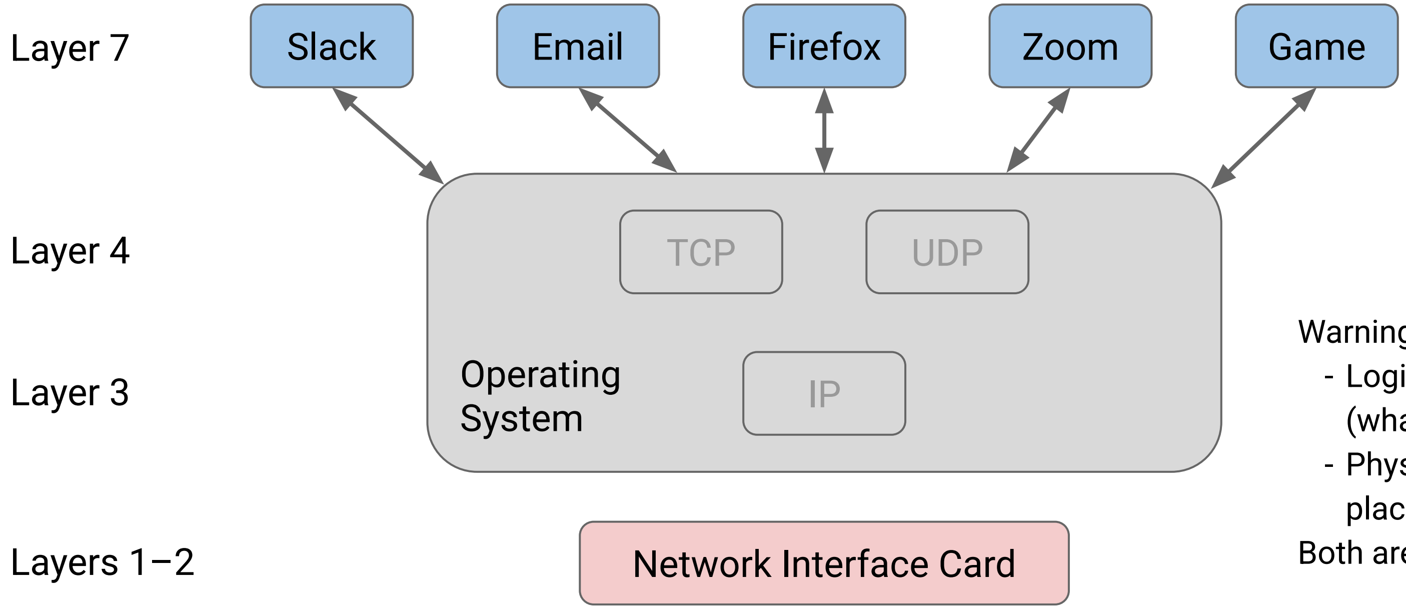
Protocol (L3) and port (L4) together allow us to de-multiplex arriving packets.



Transport Layer Feature: De-Multiplexing with Ports

Ports identify the attachment point between the application and the OS.

Ports allow data to pass between the application and the OS, without mixing up data between applications.



Warning: Naming conflict.

- Logical ports: L4 header field (what you're looking at now).
- Physical ports: On a router, the place you plug in a link.

Both are often called "ports."

Transport Layer Feature: De-Multiplexing with Ports

Analogy: Ports are like room numbers in a house/building.

- The address is the same, but the room number specifies one person in the house.
- Room numbers can be anything in private houses.
- Room numbers should be well-known and constant in public buildings.

Ports identify applications on a specific machine.

- Port numbers are 16 bits long.
- On private computers, ports can be anything.
    - "Ephemeral" port numbers: 1024–65535.
    - Applications can pick a random port number.
- On public services, ports should be well-known and public.
    - Examples: SSH (22), HTTP (80).
    - Services can listen for requests on a well-known port.
    - Users know the appropriate port number to contact.

Transport Layer Feature: Reliability

Many application semantics involve reliable delivery.

- Example: File transfer.

Implemented by TCP, but not UDP.

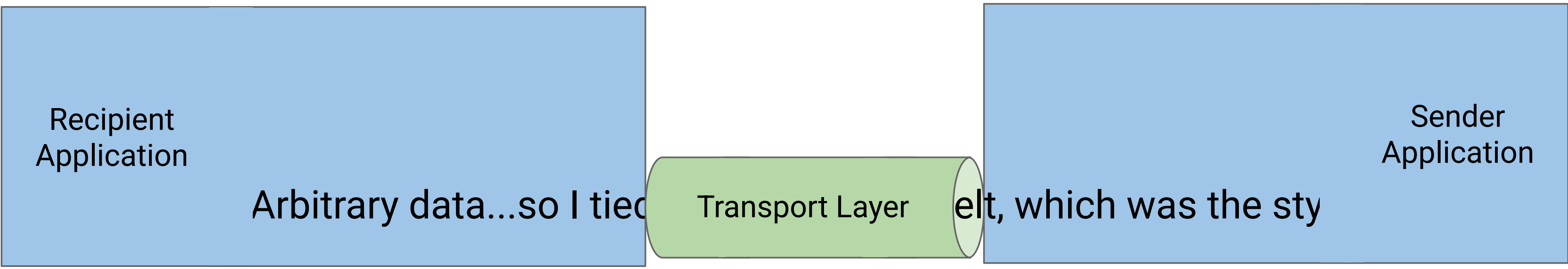
Challenge: Building a reliable service (L4) on top of unreliable packet delivery (L3).

- We'll spend most of today and next time building this.

Transport Layer Feature: New Abstractions: Bytestream

TCP implementing reliability gives the application a new abstraction.

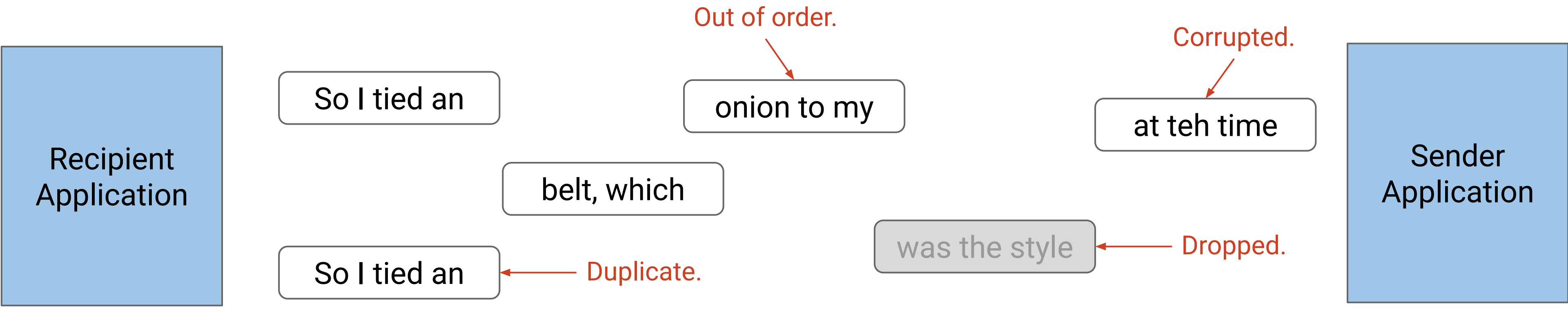
- Without reliability (L3): Unreliable packet abstraction.
  - Send individual packets, which might be dropped/corrupted/re-ordered.
- With reliability (L4): Reliable **bytestream abstraction**.
  - Sender application sends an arbitrary-length string of bytes.
  - The same bytes appear, in order, at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



Transport Layer Feature: New Abstractions: Datagram

UDP does not implement reliability, and it gives users the **datagram abstraction**.

- Basically the same as the Layer 3 packet abstraction.
- Still best-effort.
- The application is responsible for splitting data into limited-size packets.



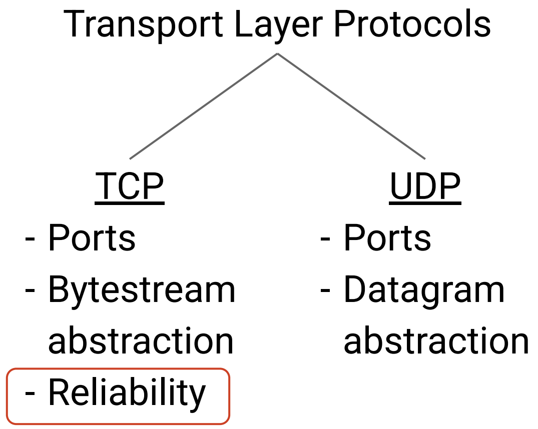
Transport Layer Protocols

Two choices of protocols at Layer 4:

- Both offer de-multiplexing with ports.
- Both offer a more useful abstraction to applications.
    - Each protocol offers a different abstraction.
- TCP additionally offers reliability.

The programmer can choose which one to use.

- If reliability needed: Choose TCP.
    - Example: File transfer.
- If reliability isn't needed: Choose UDP.
    - Example: Live streaming.

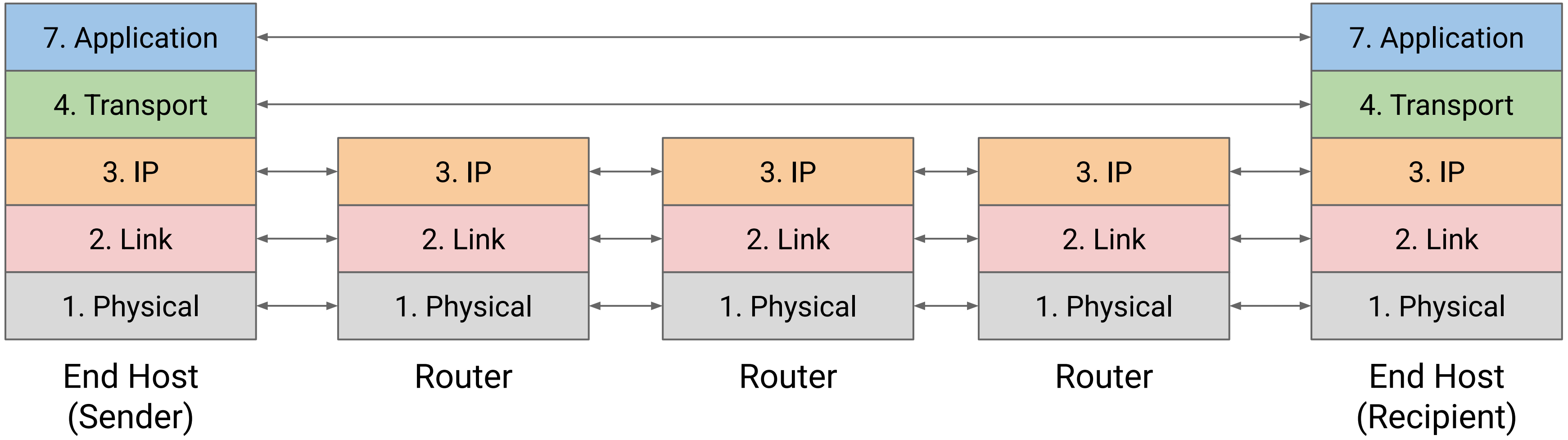


We're about to spend a lot of time here, but don't forget the bigger picture.

Transport Layer is Implemented at End Hosts

Both TCP and UDP are implemented at end hosts, not intermediate routers.

- De-multiplexing: Give the packet to the appropriate application...on the *end host*.
- Provide a better abstraction...for the *end hosts*.
- Reliability at end hosts: End-to-end principle.
  - Alternate design: Make every link between routers reliable.
    Not what we chose to do.



Implementing Reliability: Single Packet

Lecture 11, CS 168, Spring 2025

Implementing Reliability

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

- We'll spend most of today and next time building this.
- Our goal is to bridge the gap between the *packet abstraction* and the *bytestream abstraction*.

Goals for reliable transfer:

- 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.

Defining Reliability

3 different definitions of reliability:

- Best-effort delivery.
- *At-least-once* delivery: Every packet arrives, but some might arrive more than once (duplicates).
- *Exactly-once* delivery: Every packet arrives exactly once.

Our plan for bridging the abstraction gap:

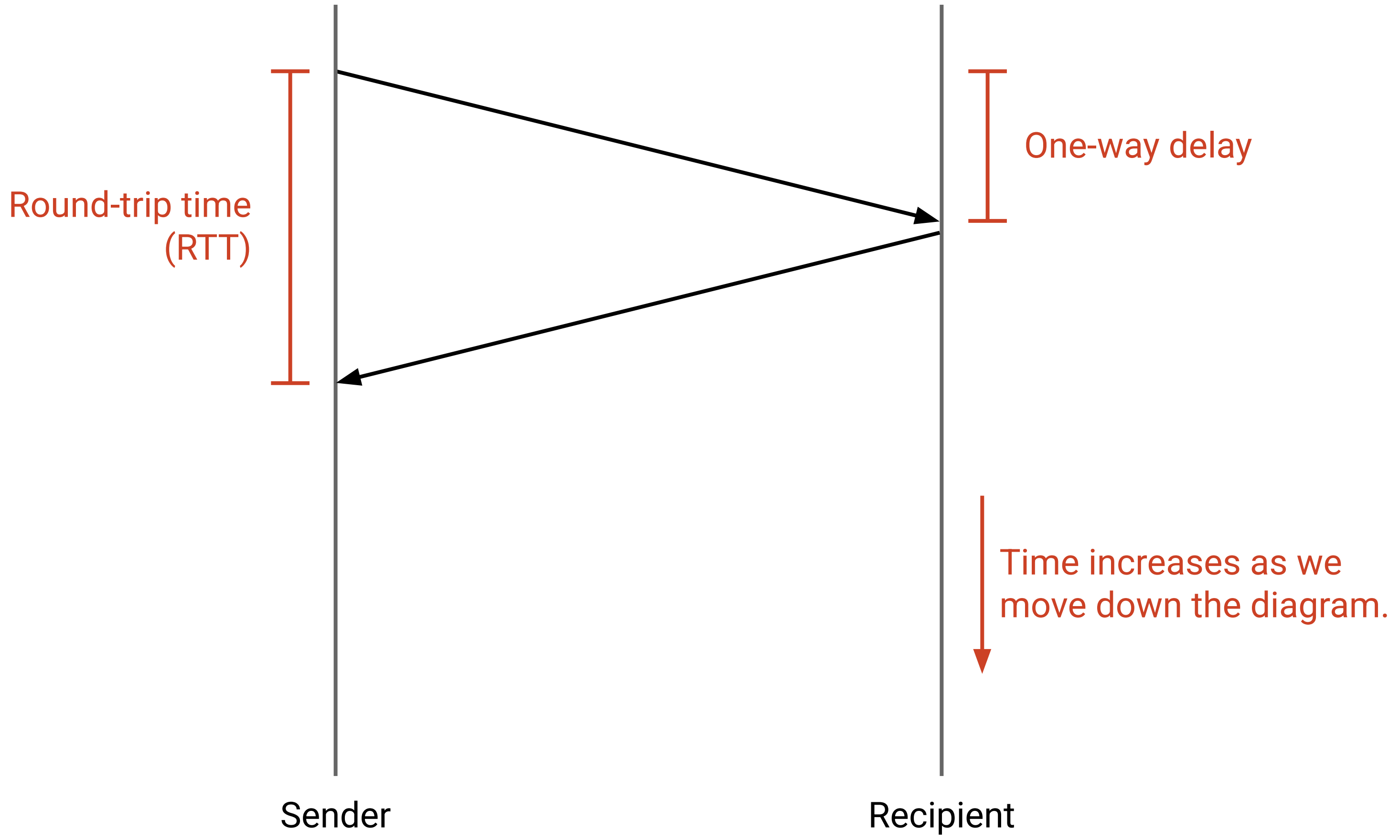
- At transport layer: Use IP (best-effort delivery) to build *at-least-once* delivery.
- Then, get rid of duplicates and pass the result to the application.
  This gives the application *exactly-once* delivery.

Defining Reliability

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 can never falsely claim to deliver a packet.

Reminder: Timing Diagrams



Designing Reliability for a Single Packet

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

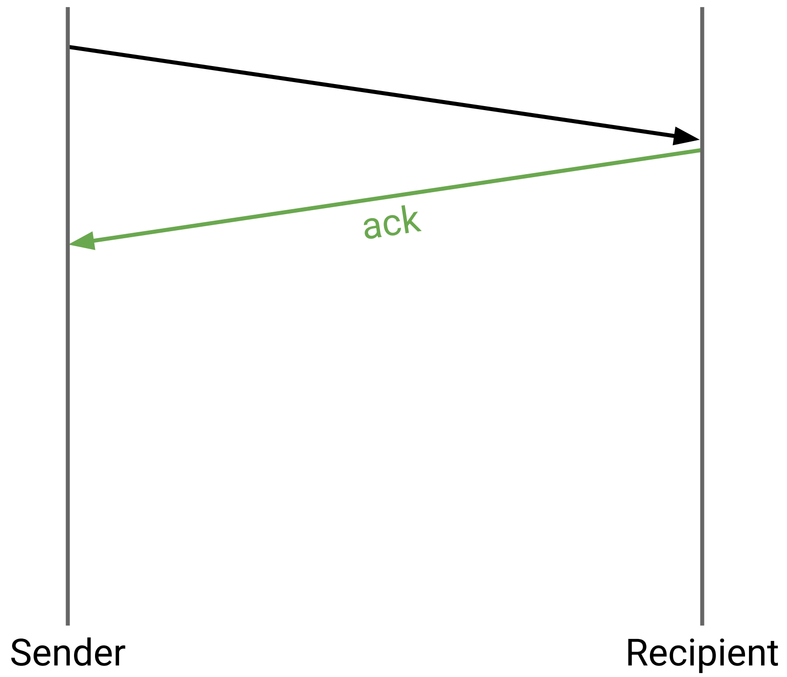
We have to deal with 5 problems from the best-effort service model:

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?

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

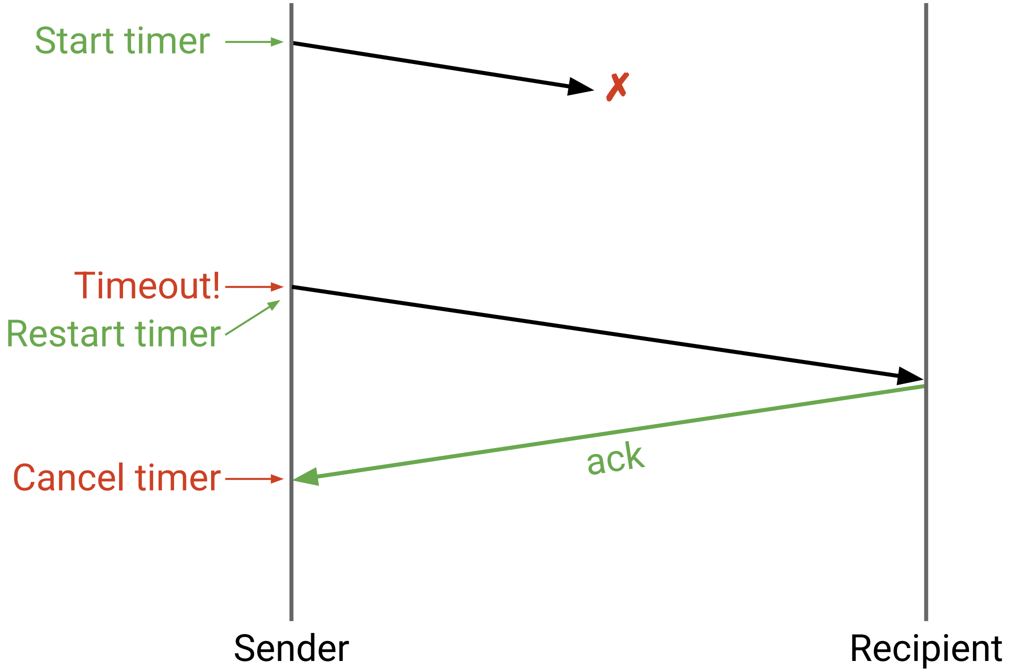


Problem 1/5: Dropped Packets

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.

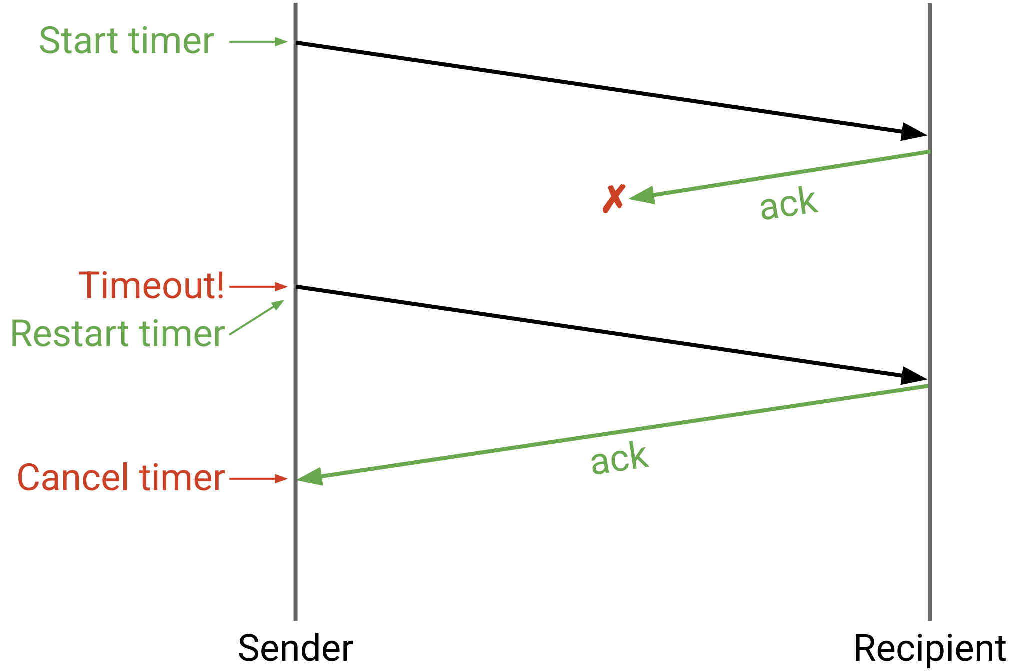


Problem 1/5: Dropped Packets

What if the ack gets dropped?

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 non-trivial.

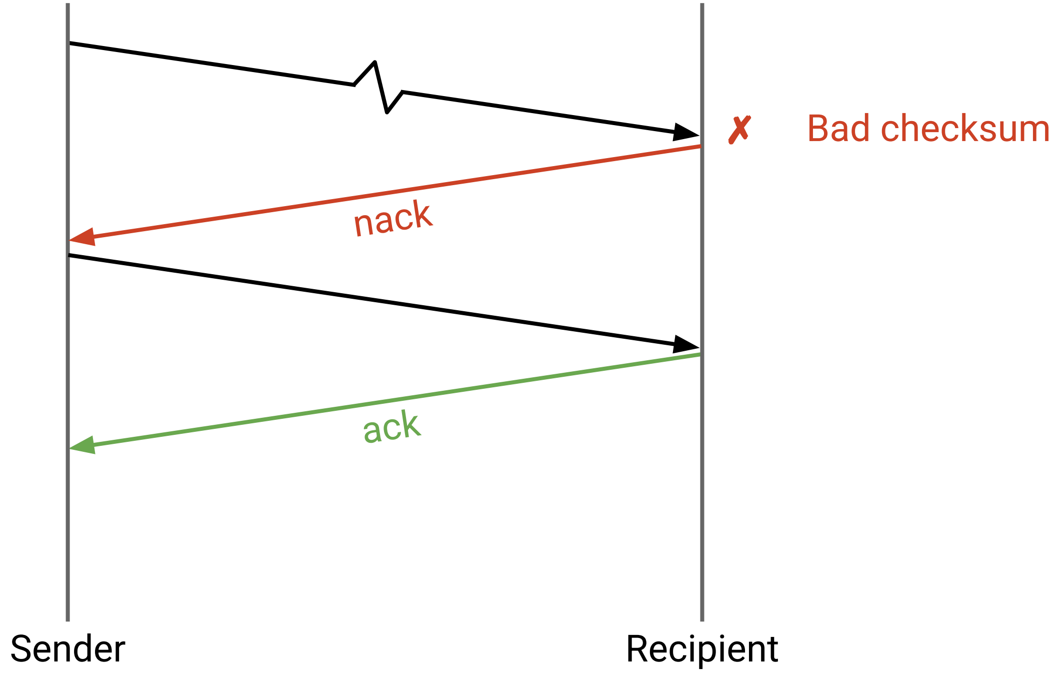
- RTT varies depending on what path the packet takes.
- RTT varies along a fixed path: packet could get stuck in queues at links.

Problem 2/5: Corrupted Packets

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.

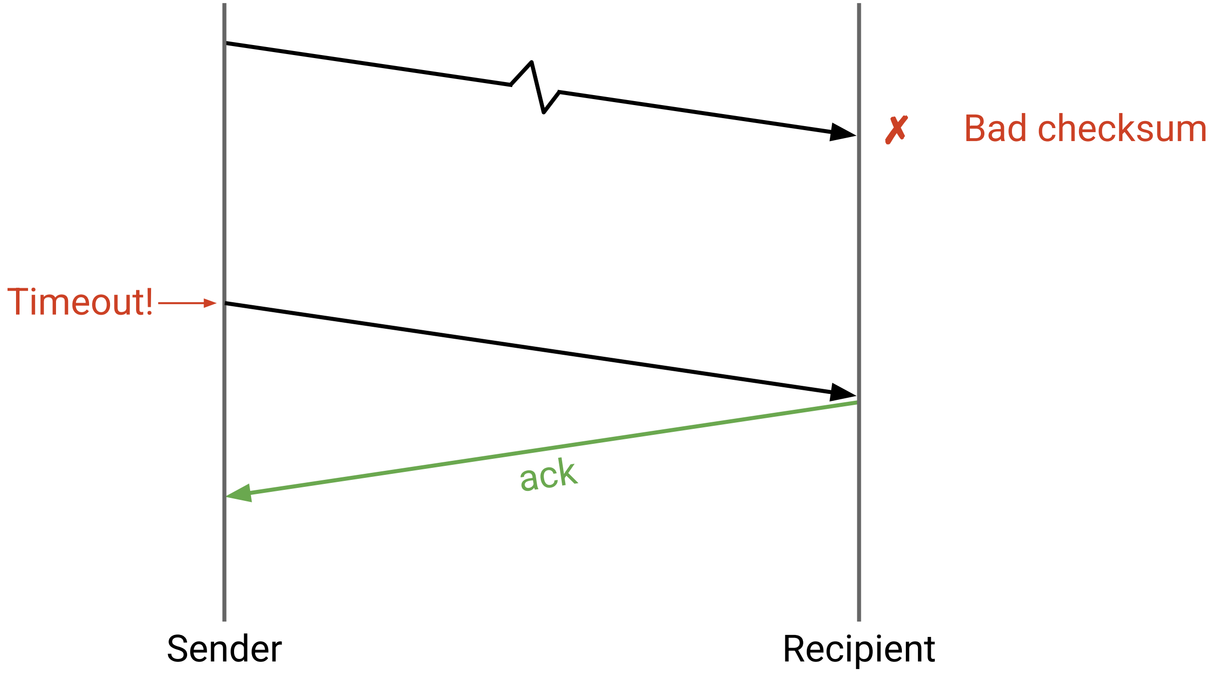


Problem 2/5: Corrupted Packets

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

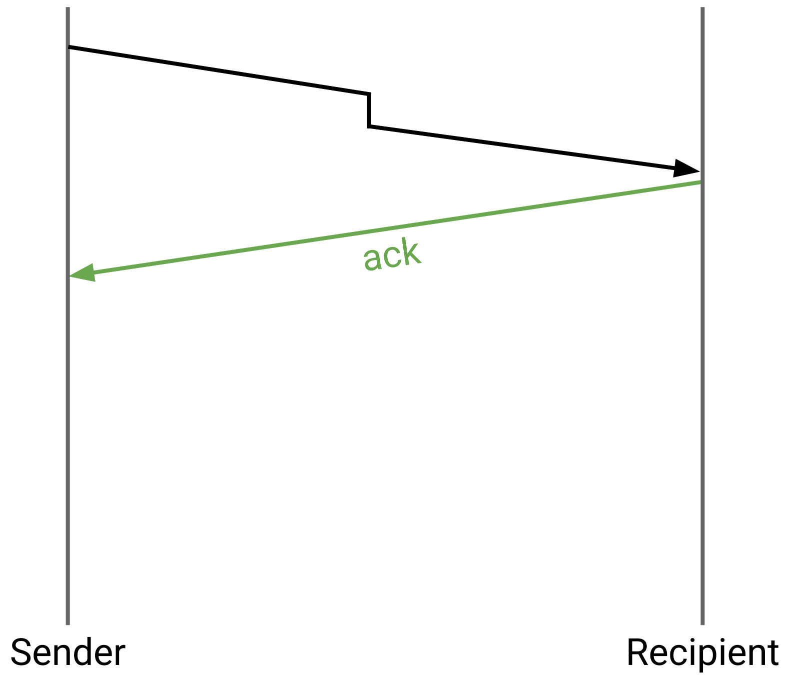
Sender times out and resends.

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



Problem 3/5: Delayed Packets

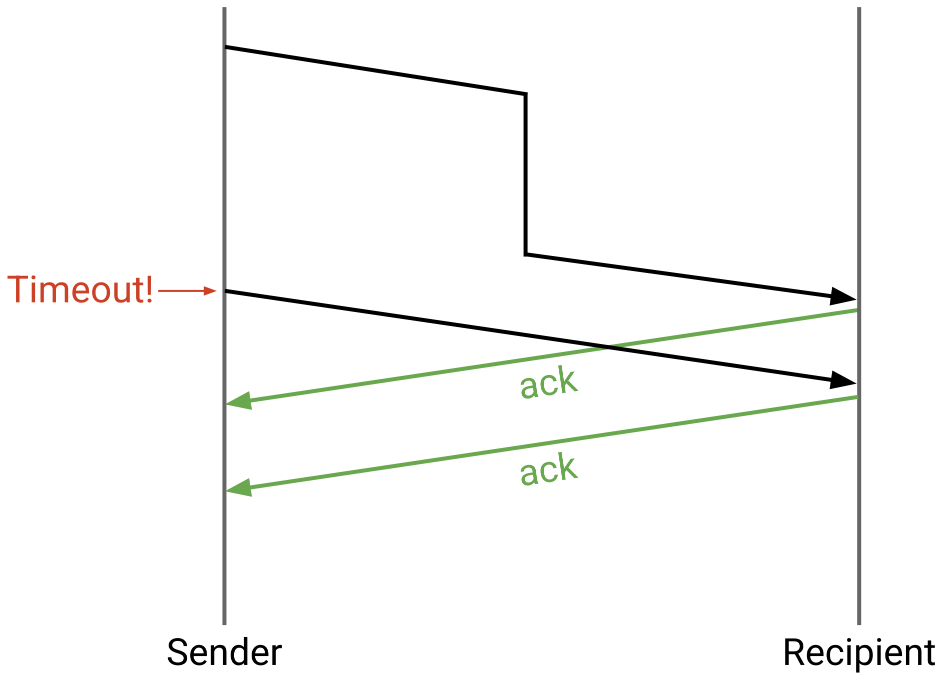
Minor delay: No problem. The ack still arrives before the timeout.



Problem 3/5: Delayed Packets

Longer delay: Sender times out and resends.

The sender receives two acks. That's fine.

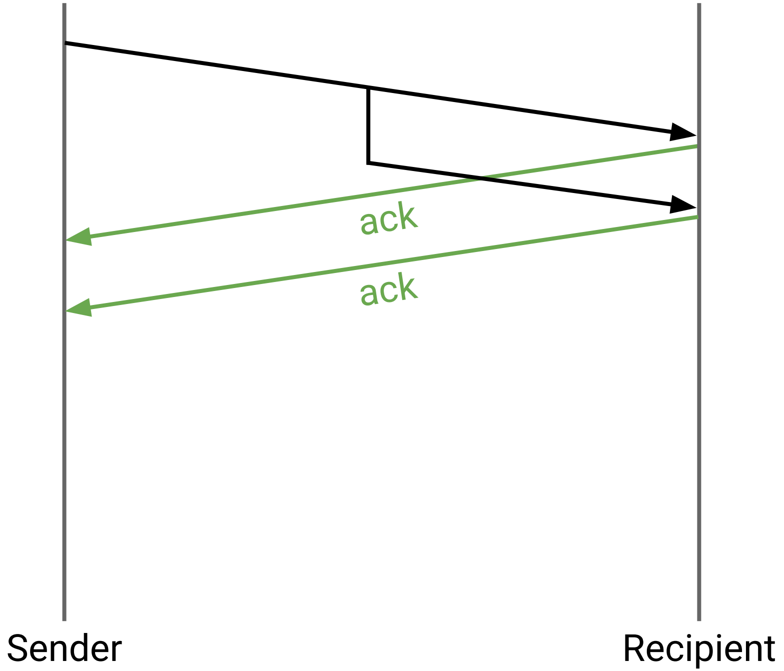


Problem 4/5: Duplicated Packets

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).



Designing 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, reset 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)

Designing Reliability for a Single Packet

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, Spring 2025

Transport Layer Design Goals

Implementing Reliability

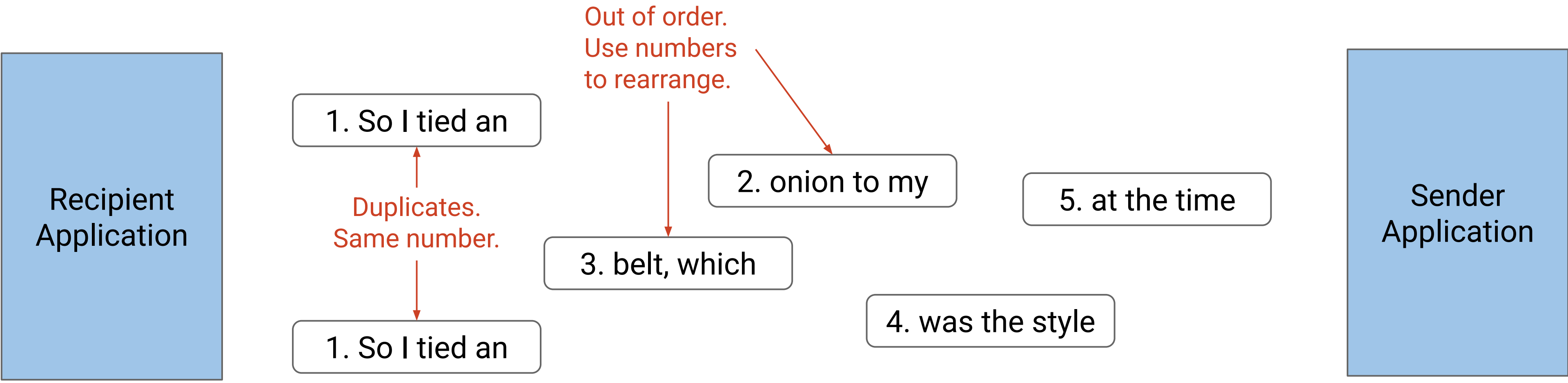
- Single Packet
- **Multiple Packets (Windows)**
- Avoiding Overload
  (Flow and Congestion Control)
- Smarter Acknowledgments
- Detecting Loss Early
- Alternate Designs

Designing Reliability for Multiple Packets

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

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

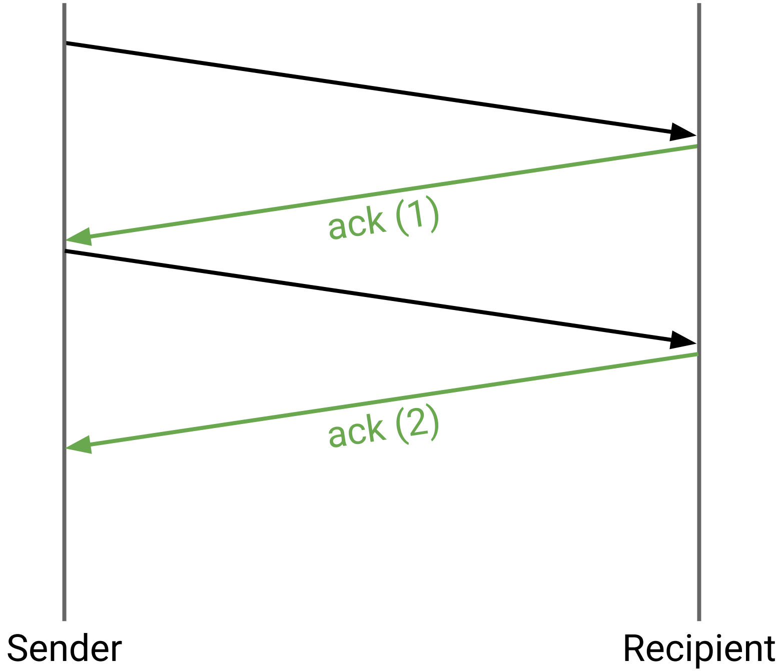
- Label each packet with a unique, increasing number.
- Label each ack with which numbered packet is being acked.
- Solves reordering – use numbers to reorder packets.



Sequence Numbers

Each packet has a unique, increasing sequence number.

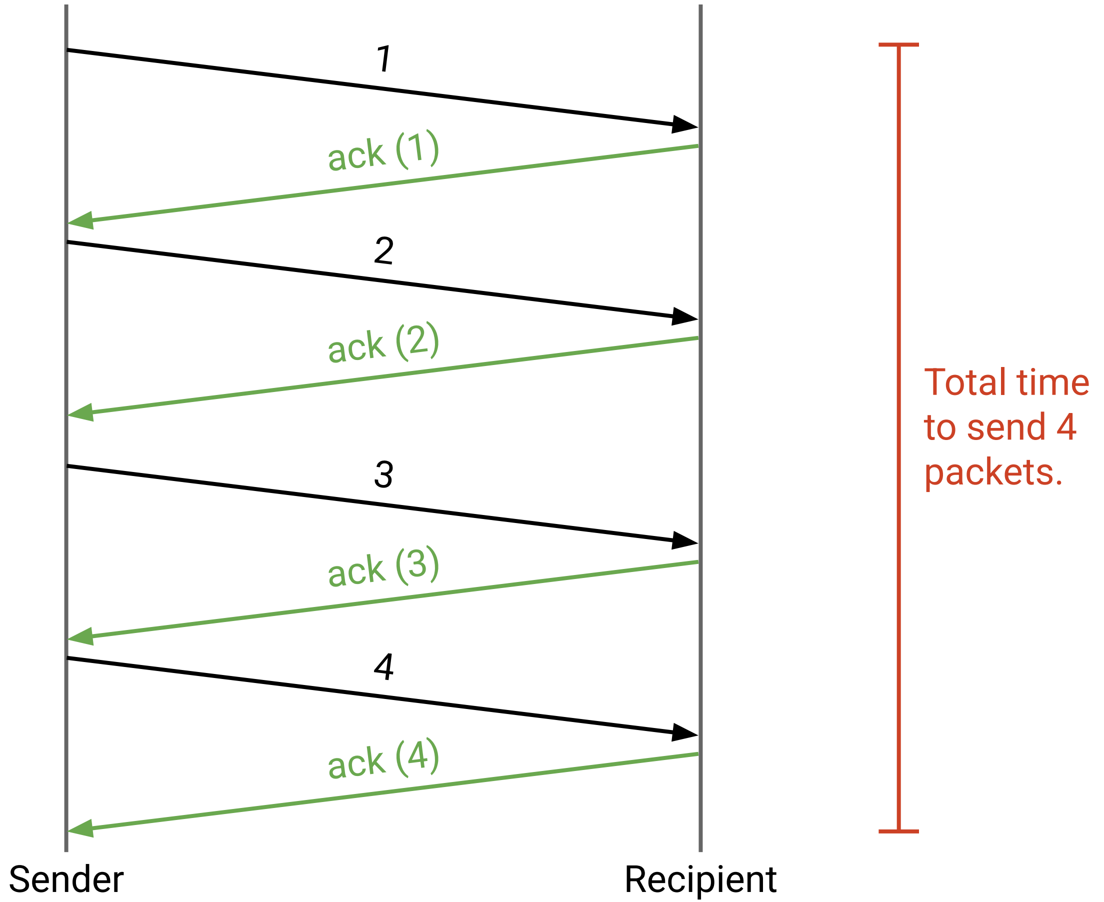
Acks indicate which packet is being acked.



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.

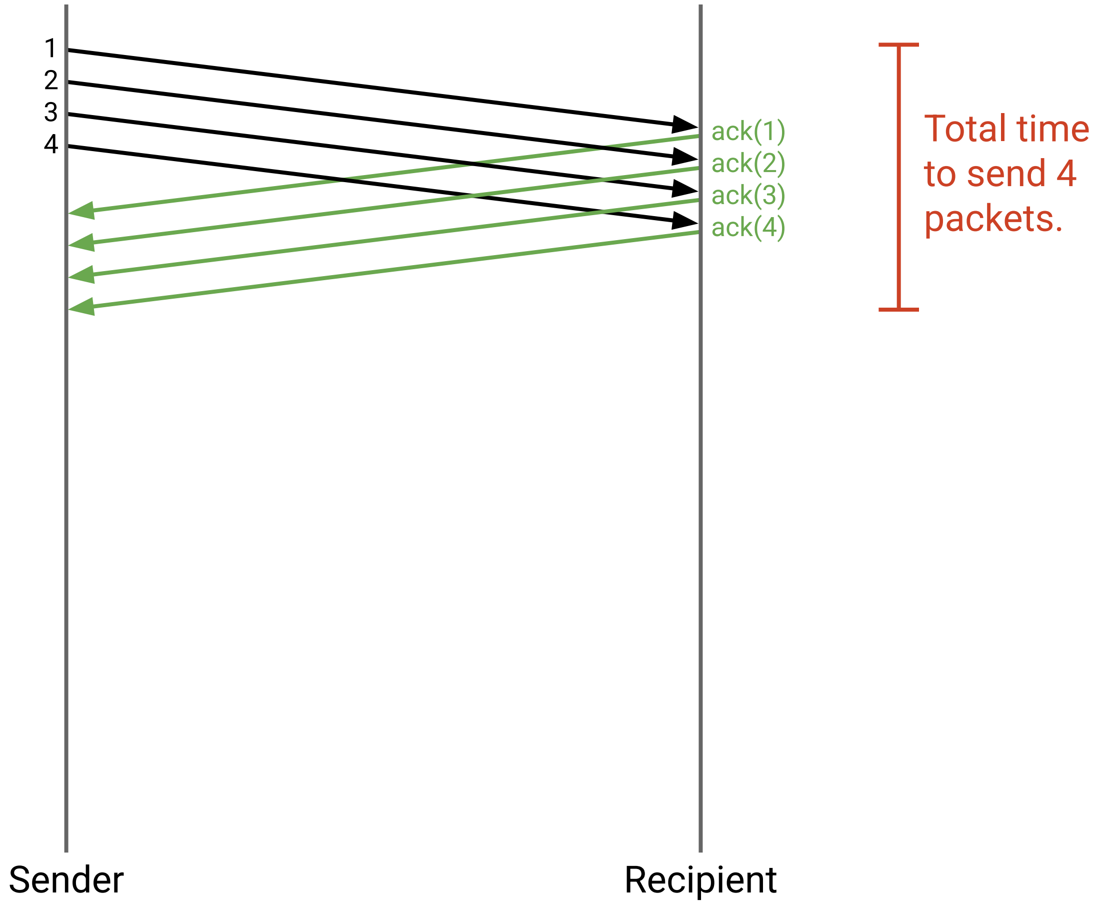


Window-Based Algorithms

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**.

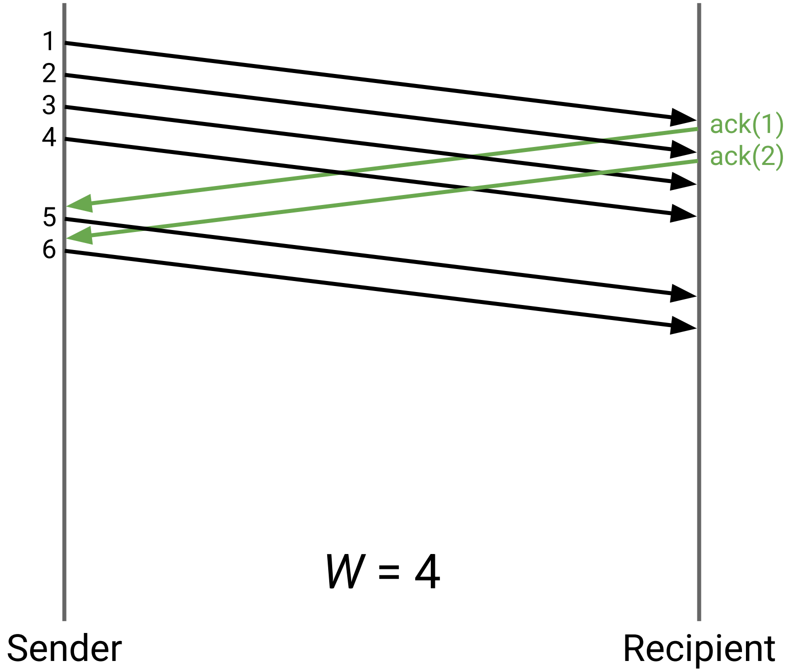
To stay inside the window limit:

- 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 (flow control)...
3. ...and don't overload links (congestion control).

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*?

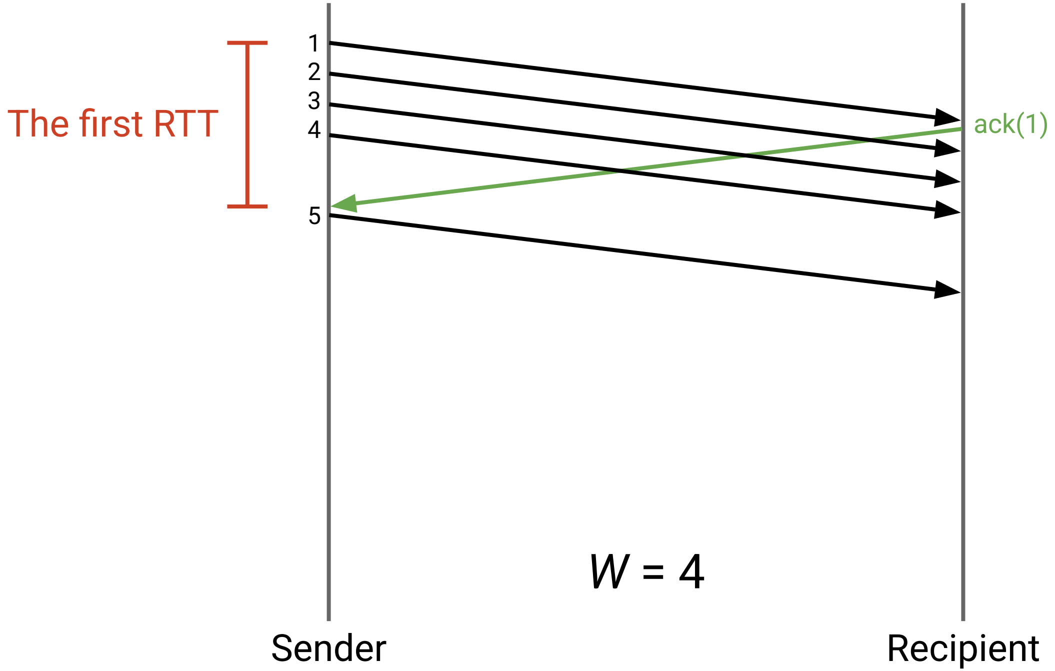
- Suppose RTT = 5 seconds, and bandwidth = 10 packets per second.
- Let's focus on the very first RTT.
- For the first 5 seconds, no acks arrive.
- We want the sender to be constantly busy for all 5 seconds.
- How many packets can be sent in 5 seconds?
- (5 seconds) × (10 packets per second) = 50 packets.
- 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: *W* = RTT × bandwidth.

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

After sending the maximum *W* = 4 packets, the sender sits idle, waiting for ack(1).

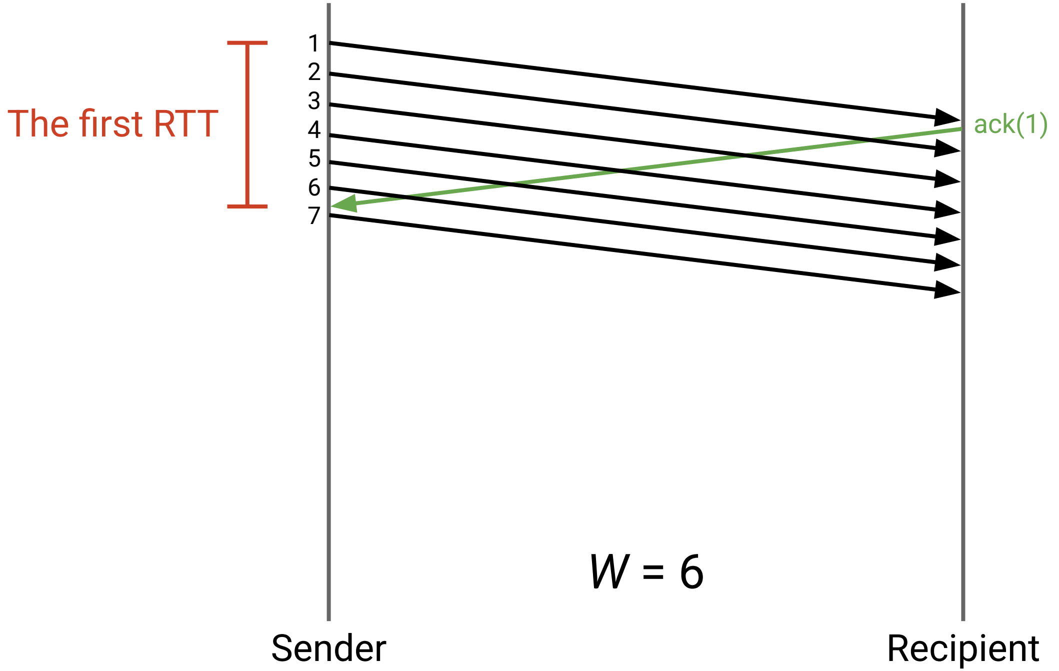
This window size is too small to fill the pipe.



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 = RTT × 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).

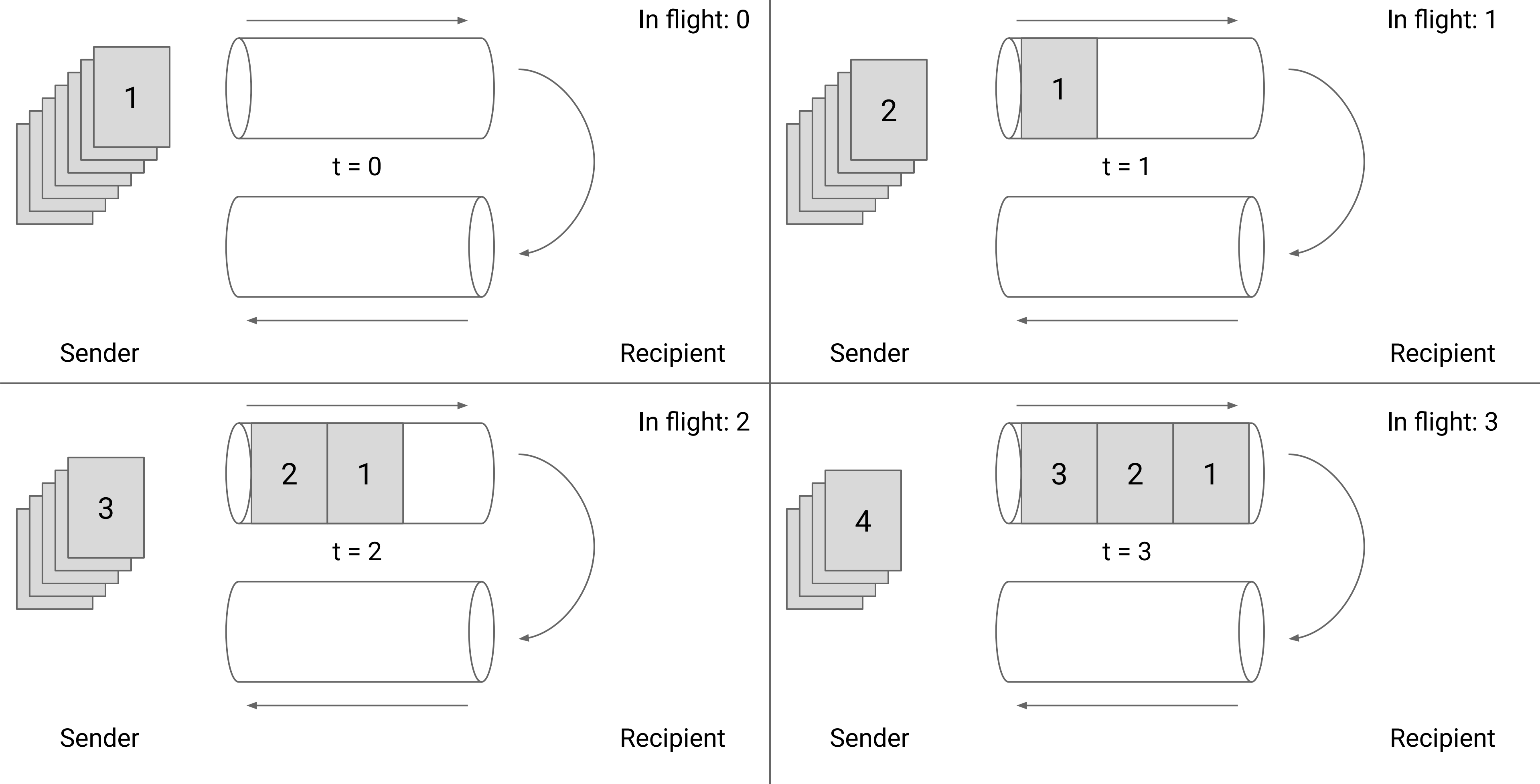
We could rewrite this in terms of bytes, not packets.

- W × (packet size) = RTT × bandwidth
- W = window size, in packets, and W × (packet size) = window size, in bytes.
- Bandwidth now measured in bits/second, not packets/second.

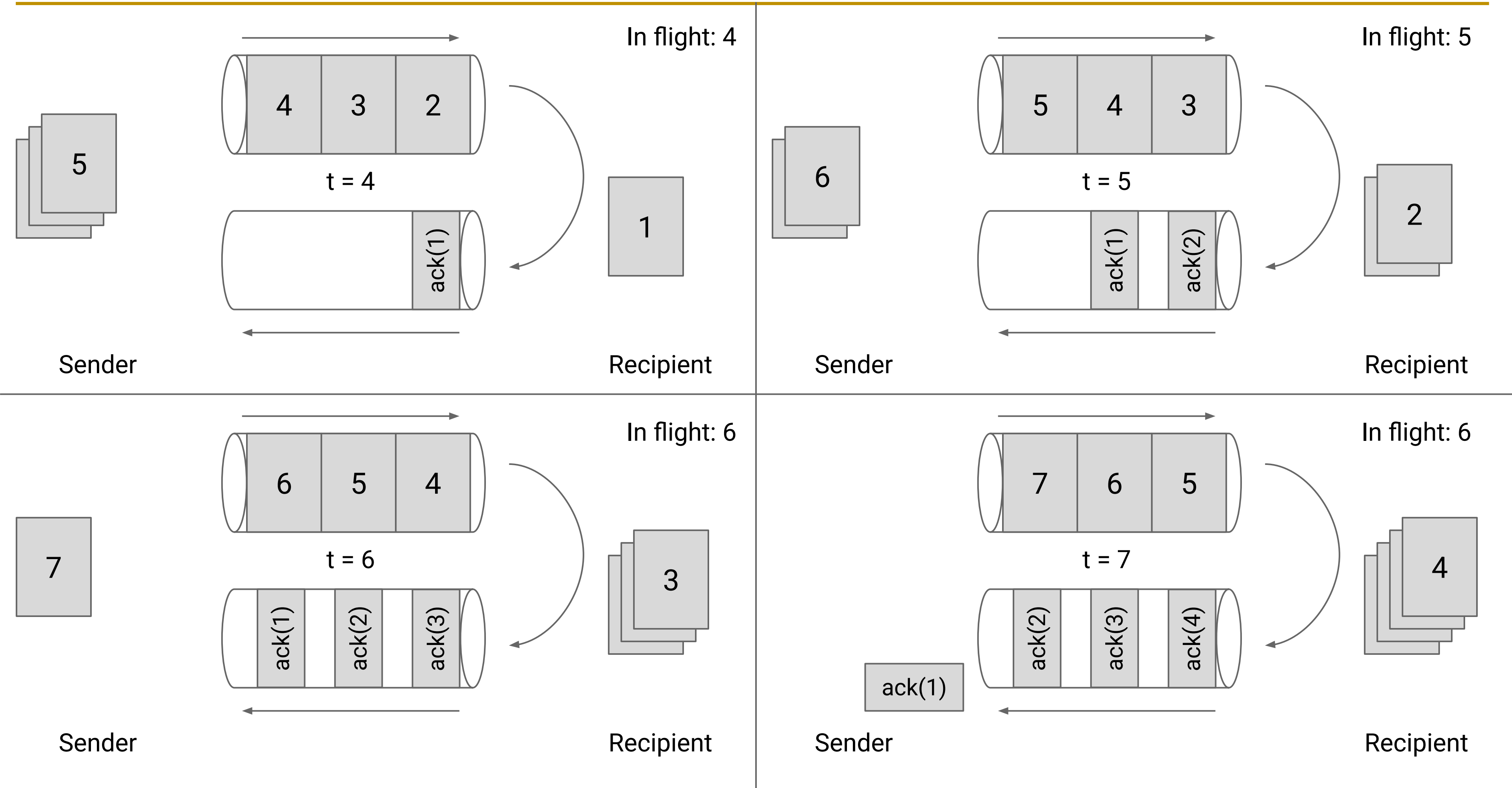
This gives us an *upper bound* on the desired size of W.

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

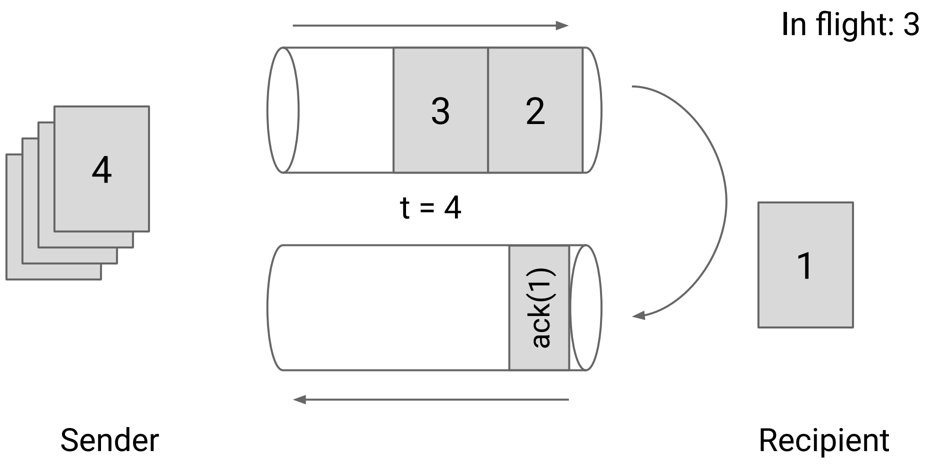
Setting Window Size (1/3): Filling the Pipe (W = 6)

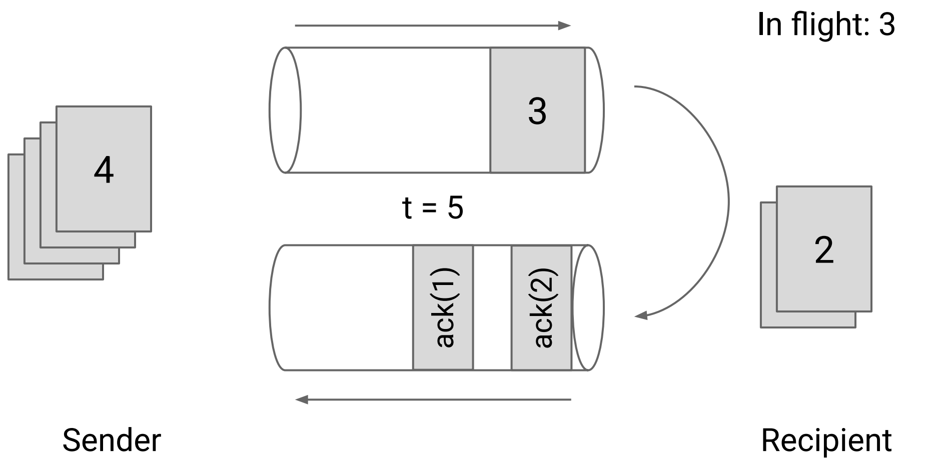


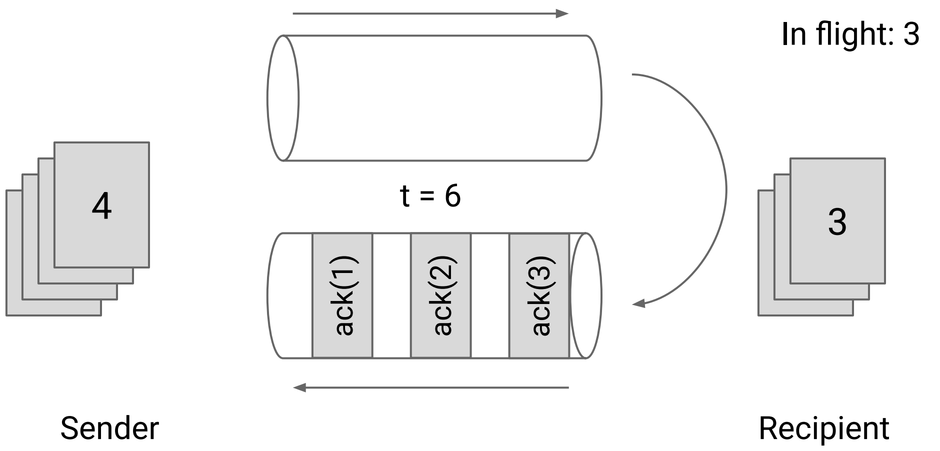
Setting Window Size (1/3): Filling the Pipe (W = 6)

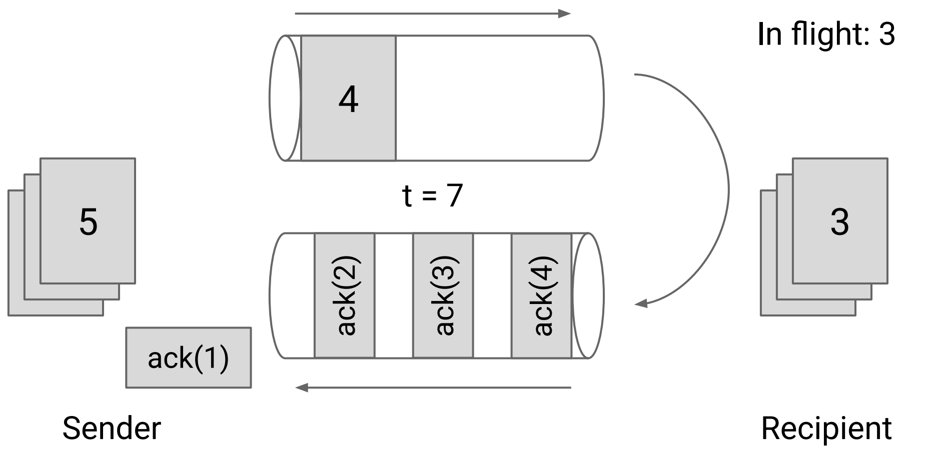


Setting Window Size (1/3): Filling the Pipe (*W* = 3)









Avoiding Overload: Flow Control and Congestion Control

Lecture 11, CS 168, Spring 2025

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:

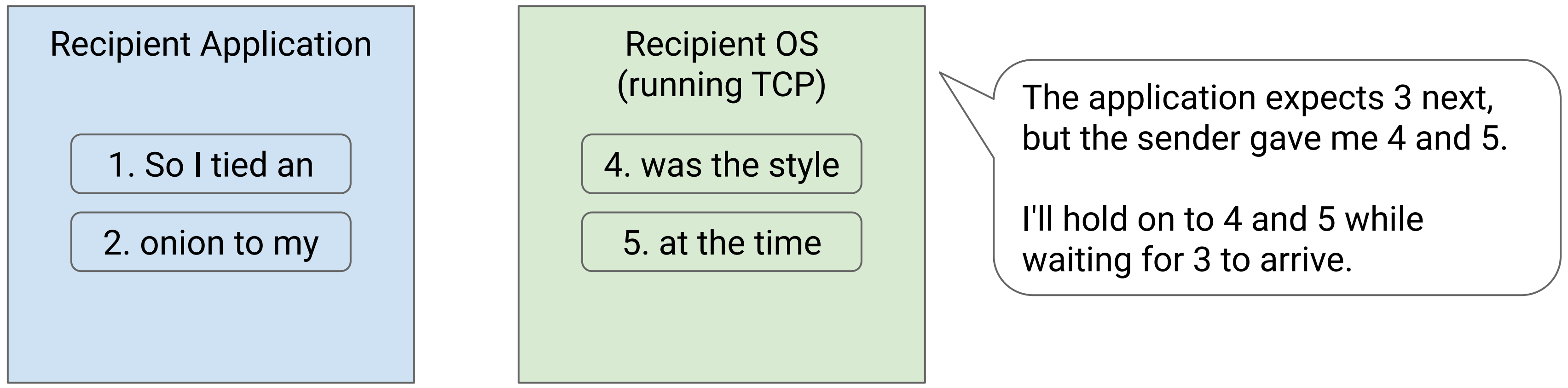
1. Take advantage of network capacity ("fill the pipe")... (set *W* = RTT × bandwidth)
2. ...but don't overload the recipient (flow control)...
3. ...and don't overload links (congestion control).

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.


Recipient Application
1. So I tied an
2. onion to my
Recipient OS (running TCP)
4. was the style
5. at the time
The application expects 3 next, but the sender gave me 4 and 5.
I'll hold on to 4 and 5 while waiting for 3 to arrive.

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

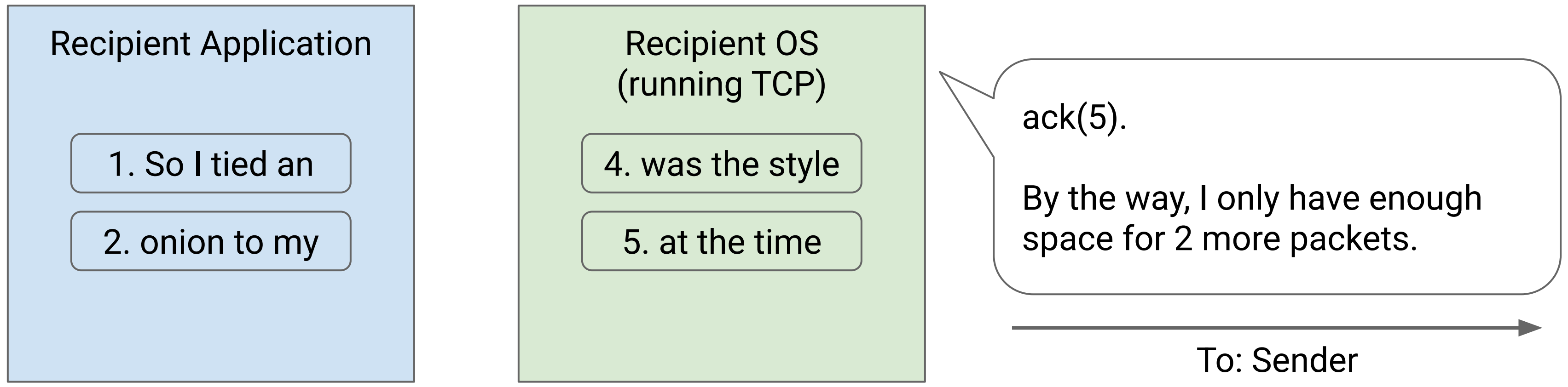
Flow control ensures the recipient has enough buffer space for out-of-order packets.

Recipient tells the sender how much space it has left.

- 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.



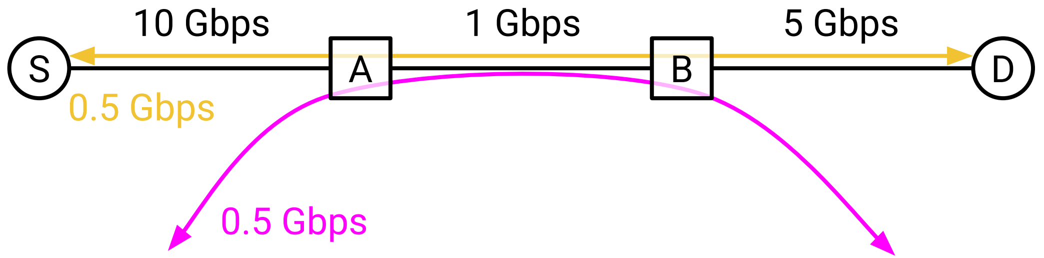
Setting Window Size (3/3): Don't Overload Network (Congestion Control)

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* = RTT × bandwidth.

So *W* and bandwidth are proportional (roughly speaking).

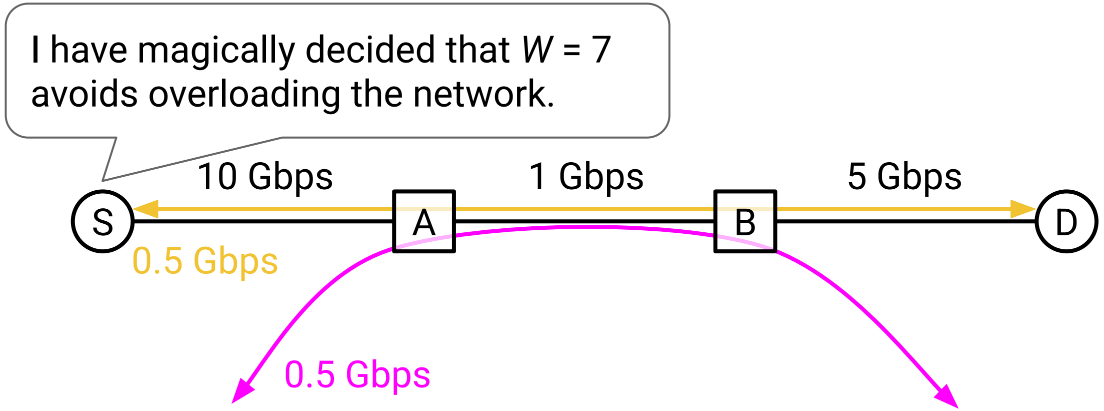
Setting Window Size (3/3): Don't Overload Network (Congestion Control)

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.


I have magically decided that *W* = 7 avoids overloading the network.
10 Gbps
1 Gbps
5 Gbps
S
A
B
D
0.5 Gbps
0.5 Gbps

Recall: W = RTT × 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* = RTT × bandwidth)
2. ...but don't overload the recipient (flow control)... (set *W* = recipient's advertised window)
3. ...and don't overload links (congestion control). (set *W* = 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 ≤ pipe-filling window.
  Equal if we're the only connection, less than if sharing bandwidth with others.
- It's hard for the sender to discover the bottleneck bandwidth and calculate (1).

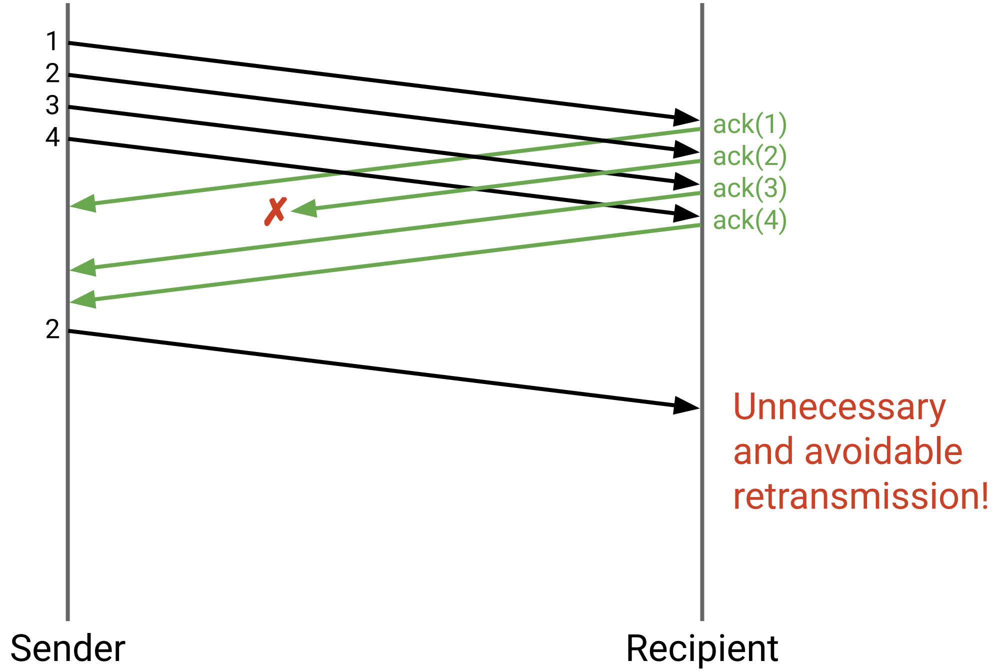
Smarter Acknowledgments

Lecture 11, CS 168, Spring 2025

Ack Strategy (1/3): Individual Packet Acks

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

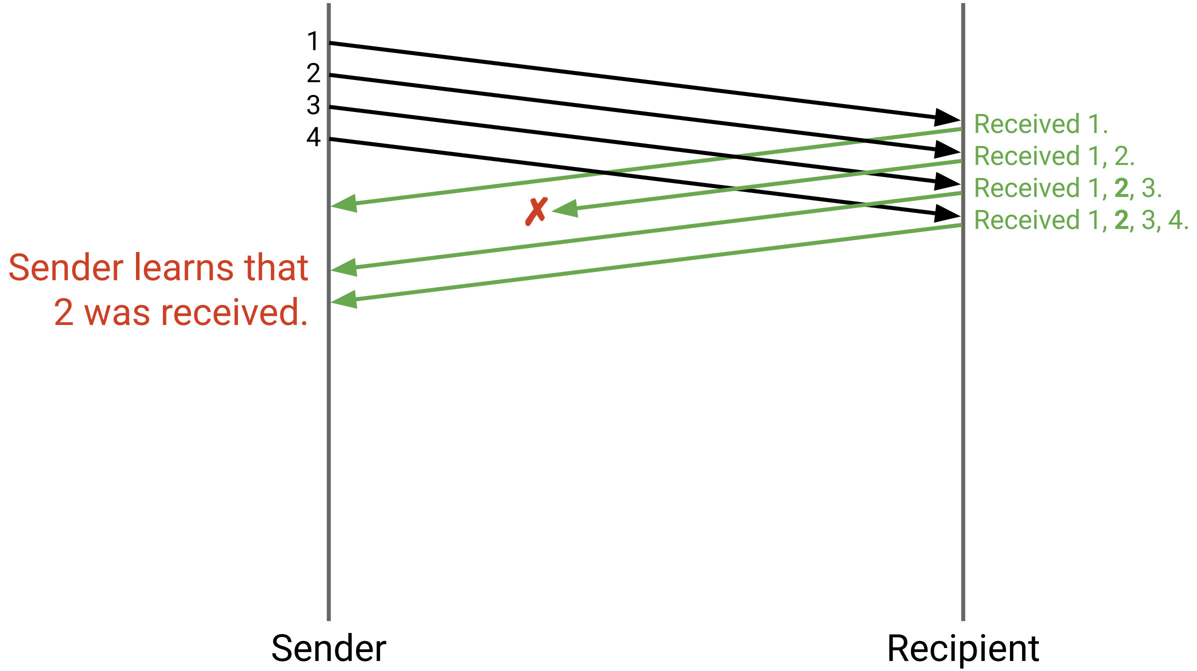
When you receive packet `i`, send ack(`i`).



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

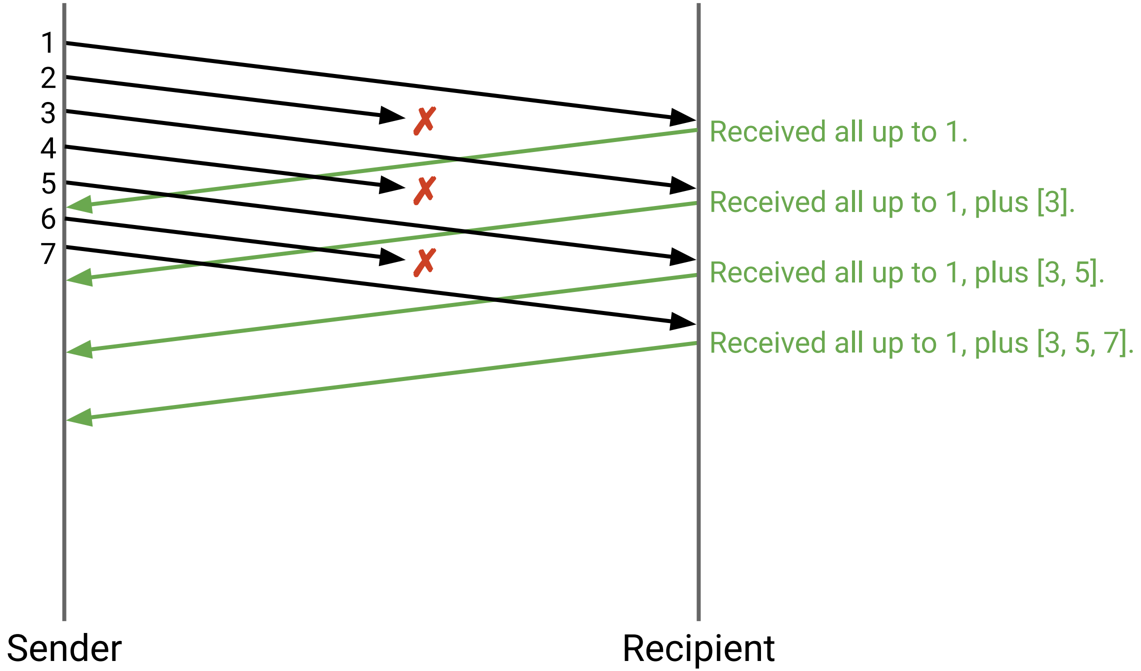
Full information ack format: "All packets 1~*n*, plus [*list out all other packets*]."

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:

$$\text{"Received all up to 1, plus } \underbrace{\text{[3, 5, 7, ..., 999]}}_{\text{Long.}}\text{"}$$



1
2
3
4
5
6
7
✗
✗
✗
Received all up to 1.
Received all up to 1, plus [3].
Received all up to 1, plus [3, 5].
Received all up to 1, plus [3, 5, 7].
Sender
Recipient

Ack Strategy (3/3): Cumulative Acks

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

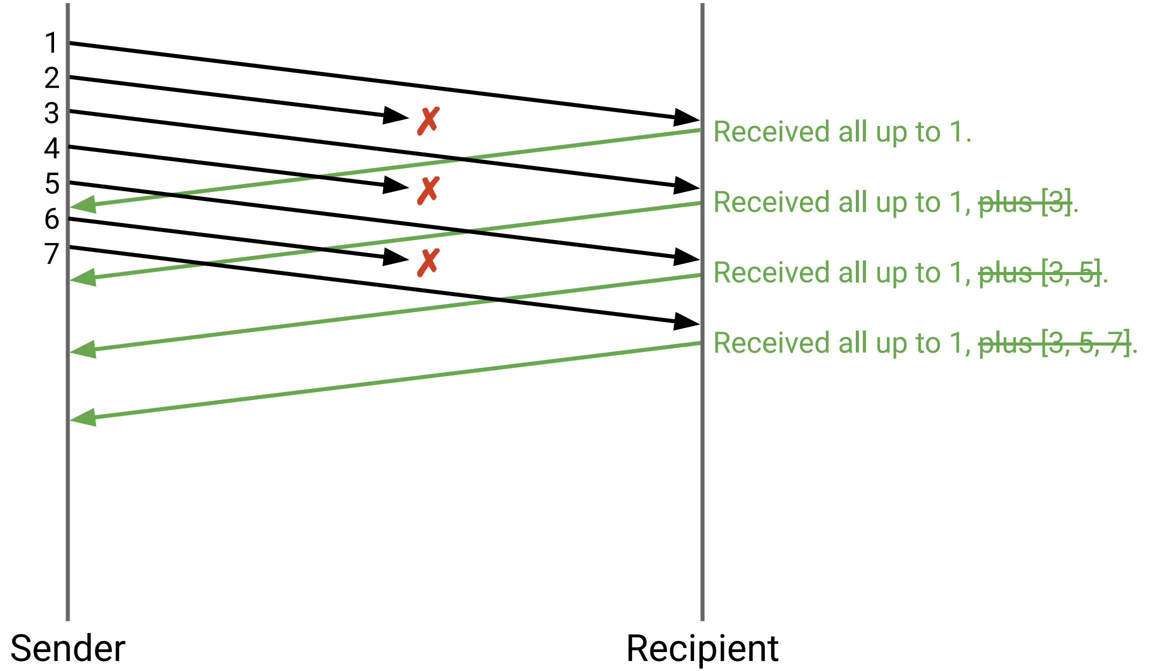
Example: "I received all packets up to 12, ~~plus 14, 15, 16~~." (Full info, minus the list.)

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.


1
2
3
4
5
6
7
✗
✗
✗
Received all up to 1.
Received all up to 1, ~~plus [3]~~.
Received all up to 1, ~~plus [3, 5]~~.
Received all up to 1, ~~plus [3, 5, 7]~~.
Sender
Recipient

Ack Strategies

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, Spring 2025

Smarter Loss Detection

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.

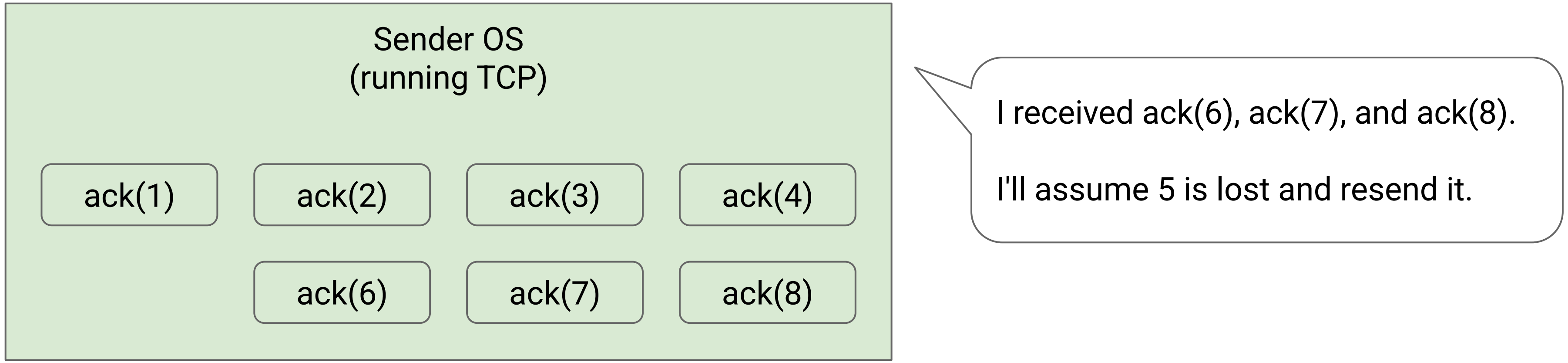
- If we receive acks for 1, 2, 3, 4, 6, 7, 8, 9, we probably lost 5.
- Declare a packet lost if *k* subsequent packets are acked.
    - *k* = 3 is common.
- Packets arrive every few nanoseconds, so this approach usually detects loss way earlier.

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

1 millisecond = 1,000,000 nanoseconds.

Ack-Based Loss Detection: Individual Acks

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



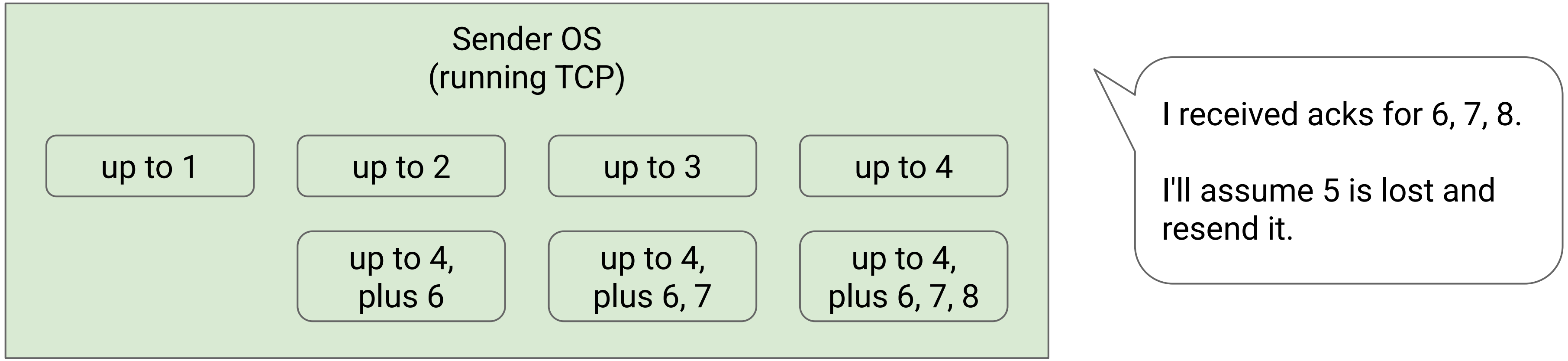
I received ack(6), ack(7), and ack(8).

I'll assume 5 is lost and resend it.

Ack-Based Loss Detection: Full-Information Acks

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

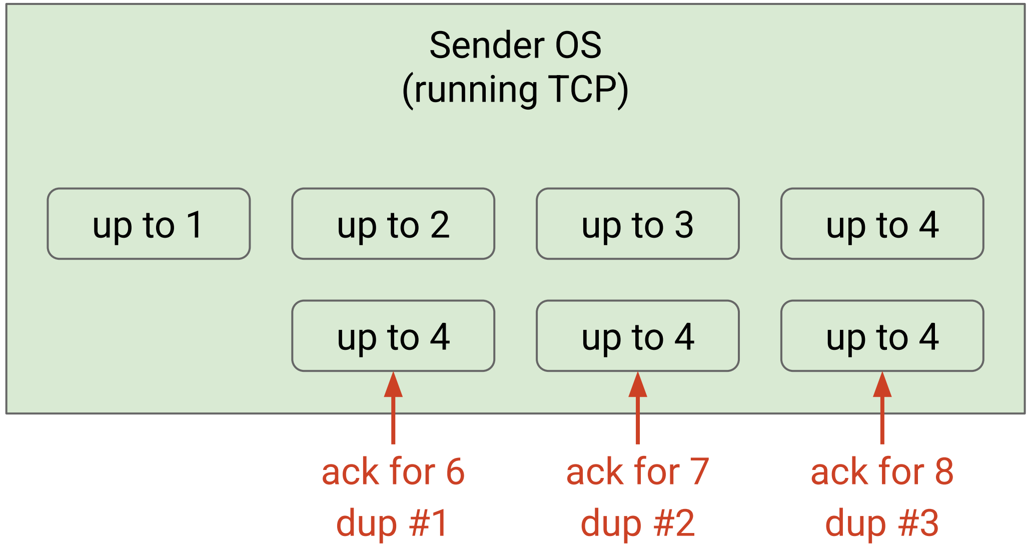
We can see the gap in packets: "up to 4, plus 6, 7, 8" → 5 is missing.



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.



I received 3 copies of "up to 4" (plus the original).

That means 3 extra packets were received (and none of them are 5).

I'll assume 5 is lost and resend it.

Responding to Loss

Timeouts:

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

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

- 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

1 2 3 4 5 6 7 8

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Individual Acks, Multiple Losses

1 2 3 4 5 6 7 8 9

- ACK 4 arrives → Send 9

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Individual Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10

- ACK 4 arrives → Send 9
- ACK 6 arrives → Send 10

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Individual Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10 11

- ACK 4 arrives → Send 9
- ACK 6 arrives → Send 10
- ACK 7 arrives → Send 11
    - [4, 6, 7] acked → Declare 3 lost. **Resend 3.**

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Individual Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10 11 12

- ACK 4 arrives → Send 9.
- ACK 6 arrives → Send 10.
- ACK 7 arrives → Send 11.
  - [4, 6, 7] acked → Declare 3 lost. **Resend 3.**
- ACK 8 arrives → Send 12.
  - [6, 7, 8] acked → Declare 5 lost. **Resend 5.**

Assumptions:

- $k = 3$
  (lost after 3 later acks)
- $W = 6$
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Individual Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10 11 12 13

- ACK 4 arrives → Send 9.
- ACK 6 arrives → Send 10.
- ACK 7 arrives → Send 11.
  - [4, 6, 7] acked → Declare 3 lost. **Resend 3.**
- ACK 8 arrives → Send 12.
  - [6, 7, 8] acked → Declare 5 lost. **Resend 5.**
- ACK 9 arrives → Send 13.
- (etc.)

Assumptions:

- k = 3
  (lost after 3 later acks)
- W = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Cumulative Acks, Multiple Losses

1 2 3 4 5 6 7 8

# duplicate acks = 0

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Cumulative Acks, Multiple Losses

1 2 3 4 5 6 7 8 9

# duplicate acks = 1

- (for packet 4) ACK 3 → Send 9

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Note: It might look like there's 7 packets in flight.
But the duplicate ACK 3 tells us one of them was received.
We just don't know which one, so we can't mark it.

Responding to Loss: Cumulative Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10

# duplicate acks = 2

- (for packet 4) ACK 3 → Send 9
- (for packet 6) ACK 3 → Send 10

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Cumulative Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10 11

# duplicate acks = 3

- (for packet 4) ACK 3 → Send 9
- (for packet 6) ACK 3 → Send 10
- (for packet 7) ACK 3 → Send 11
    - ACK 3 duplicated 3 times. **Resend 3.**

Assumptions:

- $k = 3$
  (lost after 3 later acks)
- $W = 6$
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Cumulative Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10 11 12

# duplicate acks = 4

- (for packet 4) ACK 3 → Send 9
- (for packet 6) ACK 3 → Send 10
- (for packet 7) ACK 3 → Send 11
  - ACK 3 duplicated 3 times. **Resend 3.**
- (for packet 8) ACK 3 → Send 12

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Cumulative Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10 11 12 13

# duplicate acks = 5

- (for packet 4) ACK 3 → Send 9
- (for packet 6) ACK 3 → Send 10
- (for packet 7) ACK 3 → Send 11
  - ACK 3 duplicated 3 times. **Resend 3.**
- (for packet 8) ACK 3 → Send 12
- (for packet 9) ACK 3 → Send 13

Assumptions:

- *k* = 3
  (lost after 3 later acks)
- *W* = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Cumulative Acks, Multiple Losses

1 2 3 4 5 6 7 8 9 10 11 12 13 14

# duplicate acks = 6

- (for packet 4) ACK 3 → Send 9
- (for packet 6) ACK 3 → Send 10
- (for packet 7) ACK 3 → Send 11
  - ACK 3 duplicated 3 times. **Resend 3.**
- (for packet 8) ACK 3 → Send 12
- (for packet 9) ACK 3 → Send 13
- (for packet 10) ACK 3 → Send 14
  - ACK 3 duplicated 6 times.
    Resend 3? Resend 4? Resend 5?

Assumptions:

- k = 3
  (lost after 3 later acks)
- W = 6
  (6 packets in flight)
- red = in flight
- green = acked
- Packets 3 and 5 are lost (but sender doesn't know).
- 1 and 2 already acked.

Responding to Loss: Cumulative Acks, Multiple Losses

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

- 3 copies of ACK 7 means:
  - I received everything up to (not including) 7.
  - I received 3 more packets after that (none are 7), but I didn't say which ones.
- This can tell us *how many* packets to send.
  - If we have 6 packets in-flight, 3 were received, so we can send 3 more.
- But this doesn't tell us *which* packets to resend.
  - Ambiguity leads to ad-hoc heuristics.

Unfortunately, TCP uses cumulative acks, so we have to deal with this ambiguity.

Alternate Designs

Lecture 11, CS 168, Spring 2025

Alternate Designs to Implementing Reliability

We implemented reliability by re-sending lost packets.

Another possible approach: Sender **encodes** the data to be resilient to loss.

- Basic idea: Add some redundancy to the data itself.

Example:

- We have k = 5 packets to send.
- We encode them into n = 20 packets.
- Original packets can be recovered if you receive any k' = 10 encoded packets.
- $n > k' > k$.
- Efficiency depends on k' / k. (How many encoded packets needed to recover.)

Vast literature exists on coding schemes.

- Examples: Fountain codes, raptor codes.

Historically, not used very much, but that could change... (e.g. video streaming).

Summary: Design Choices

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 made specific design choices (e.g. we chose 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!