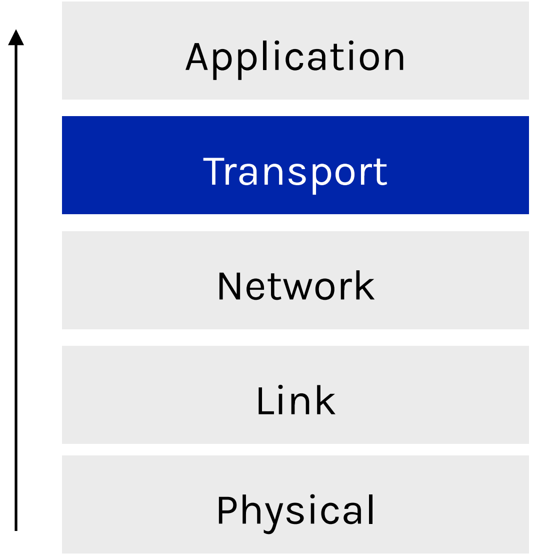
# Computer Networks (WS23/24)

## L7: The Transport Layer - Part 1

**Prof. Dr. Lin Wang**
Computer Networks Group (PBNet)
Department of Computer Science
Paderborn University

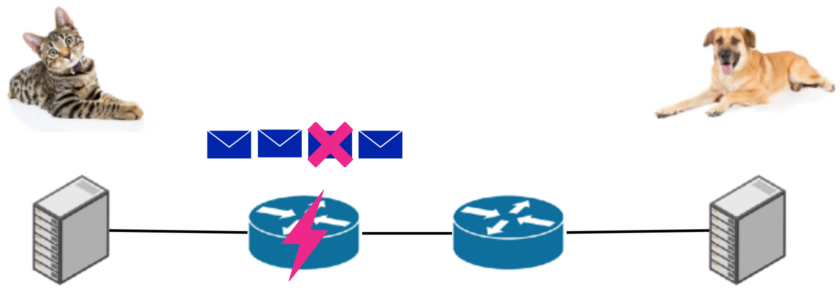
Materials inspired by Laurent Vanbever

# Learning objectives




**Part 1**

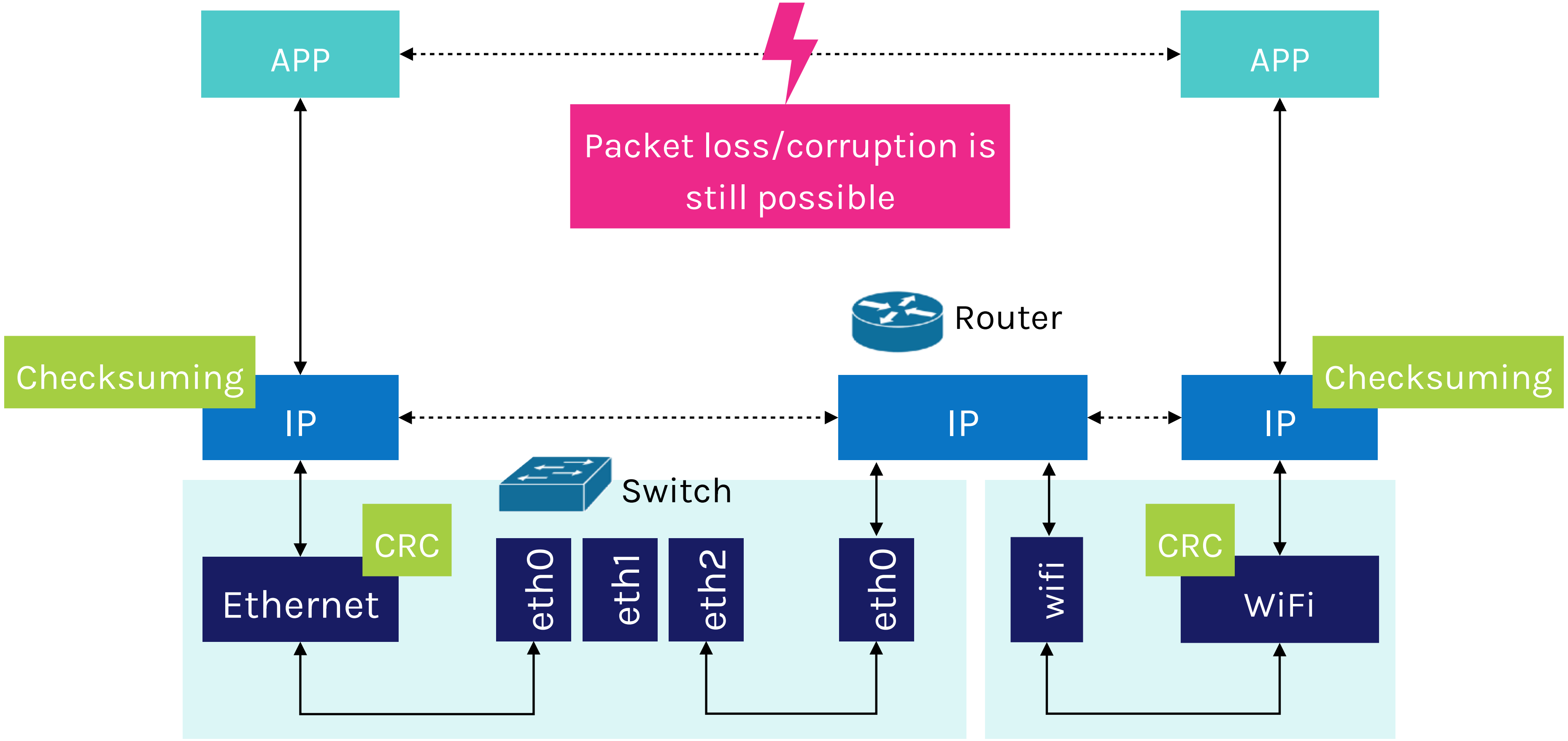
- Reliable delivery
- Correctness conditions
- Tradeoffs (timeliness, efficiency, fairness, etc.)
- Example transport mechanisms



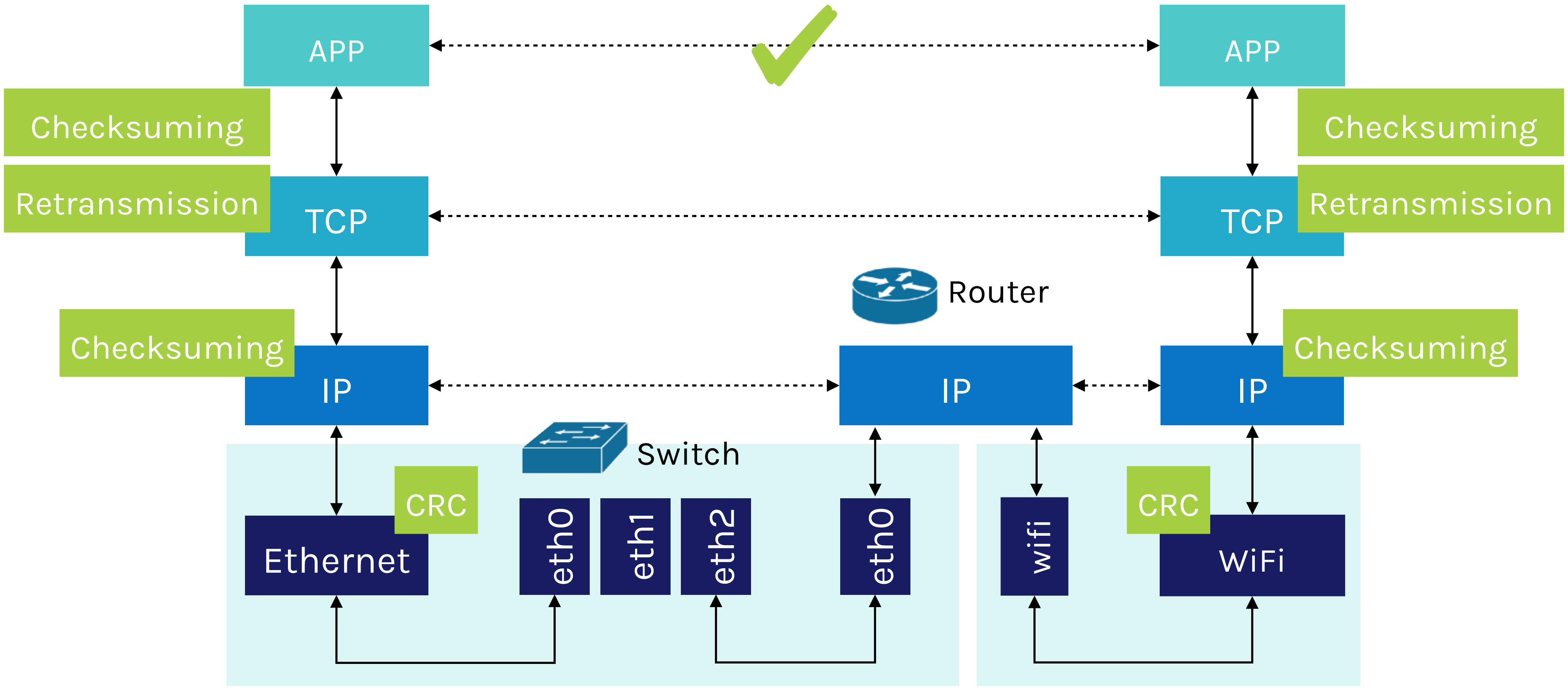
How to ensure what received is what sent?

# Reliable Delivery

# Network layer provides best-effort delivery



# Reliable delivery in the transport layer



# Reliablity delivery in the transport layer

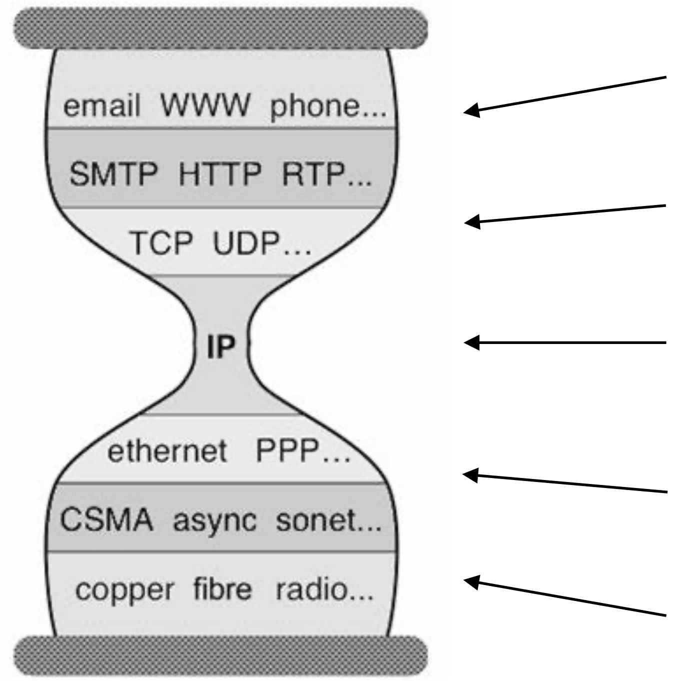
**Goals**

- Keep the network **simple, dumb**: make it relatively easy to build and operate a network
- Keep applications as **network agnostic** as possible: a developer should focus on the APP, not the specifics of the network the APP will run on

**Design**

- Implement reliability in between the network and the APP → the network layer
- Relieve the burden from both the APP and the network

# The Internet hourglass




Applications

...built on...

**Reliable** end-to-end delivery

...built on...

**Best-effort** global packet delivery

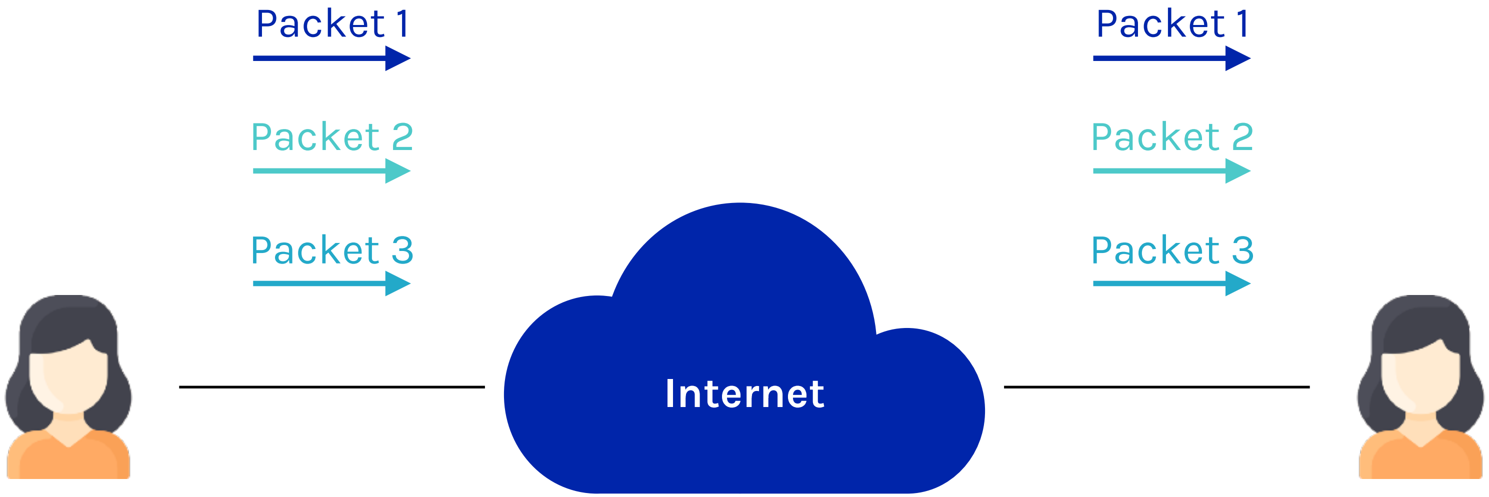
...built on...

Best effort local packet delivery

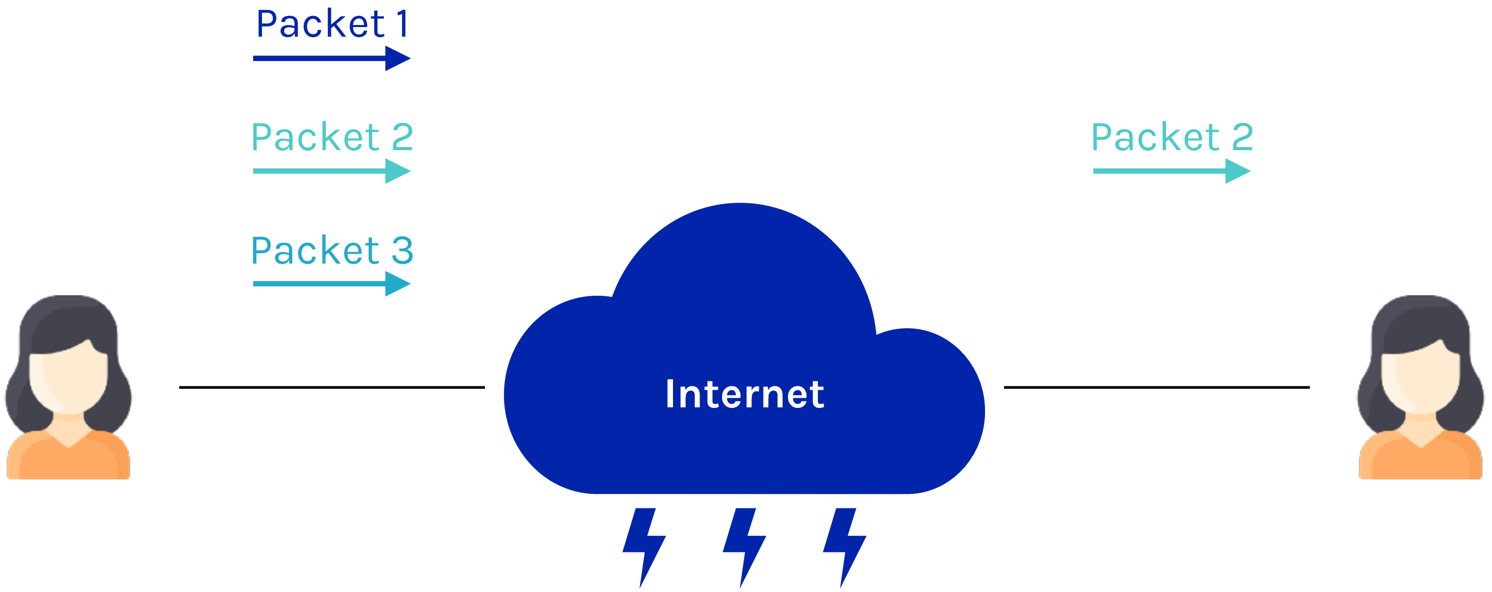
...built on...

Physical transfer of bits

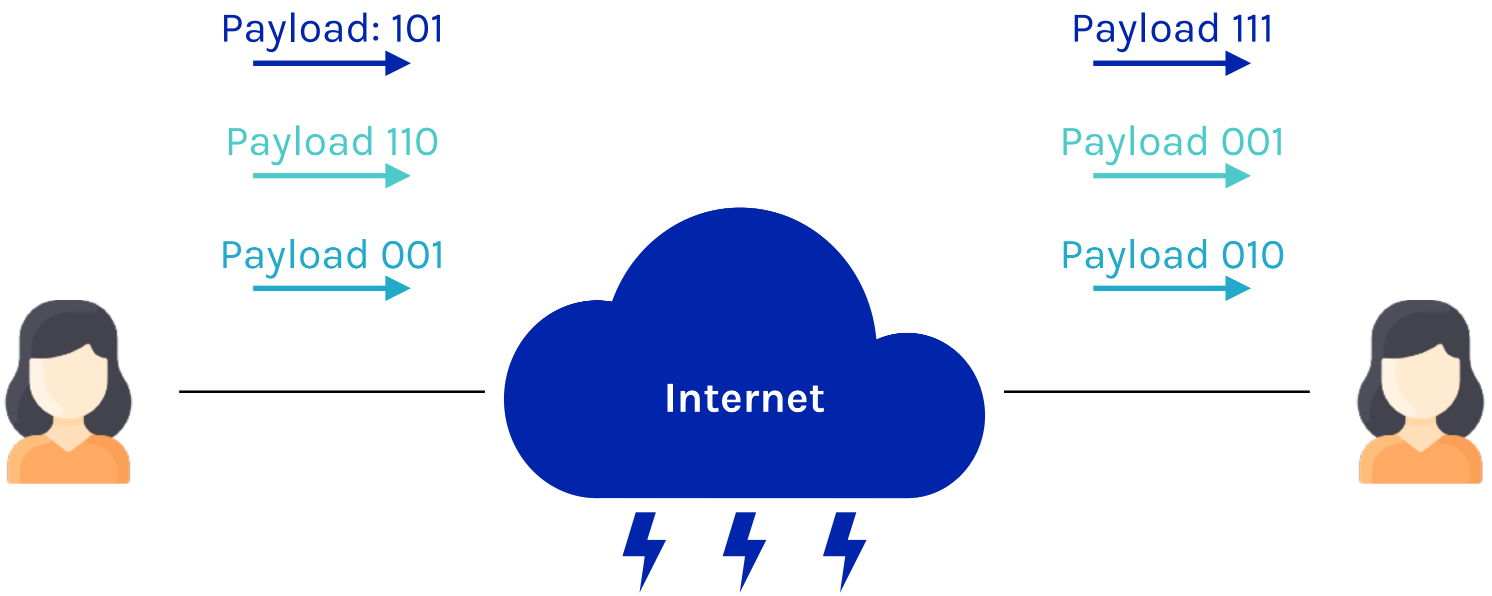
# Reliabile delivery: example



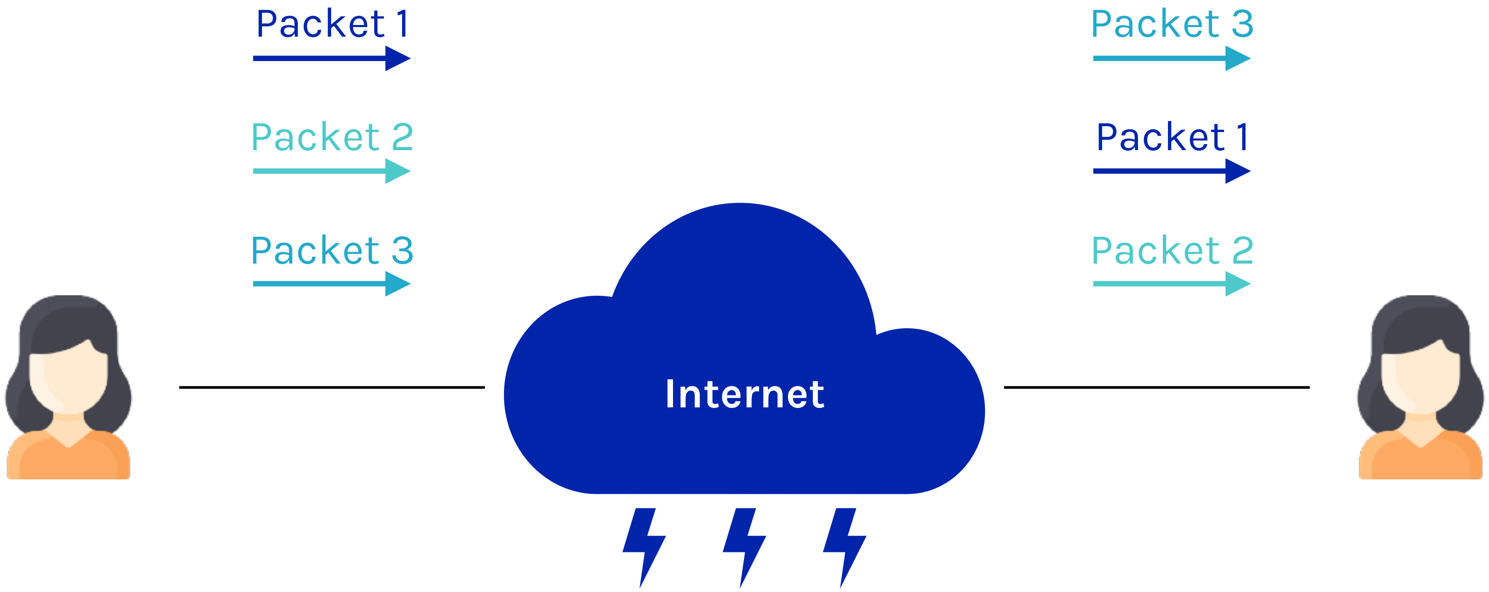
# Packet loss or delay



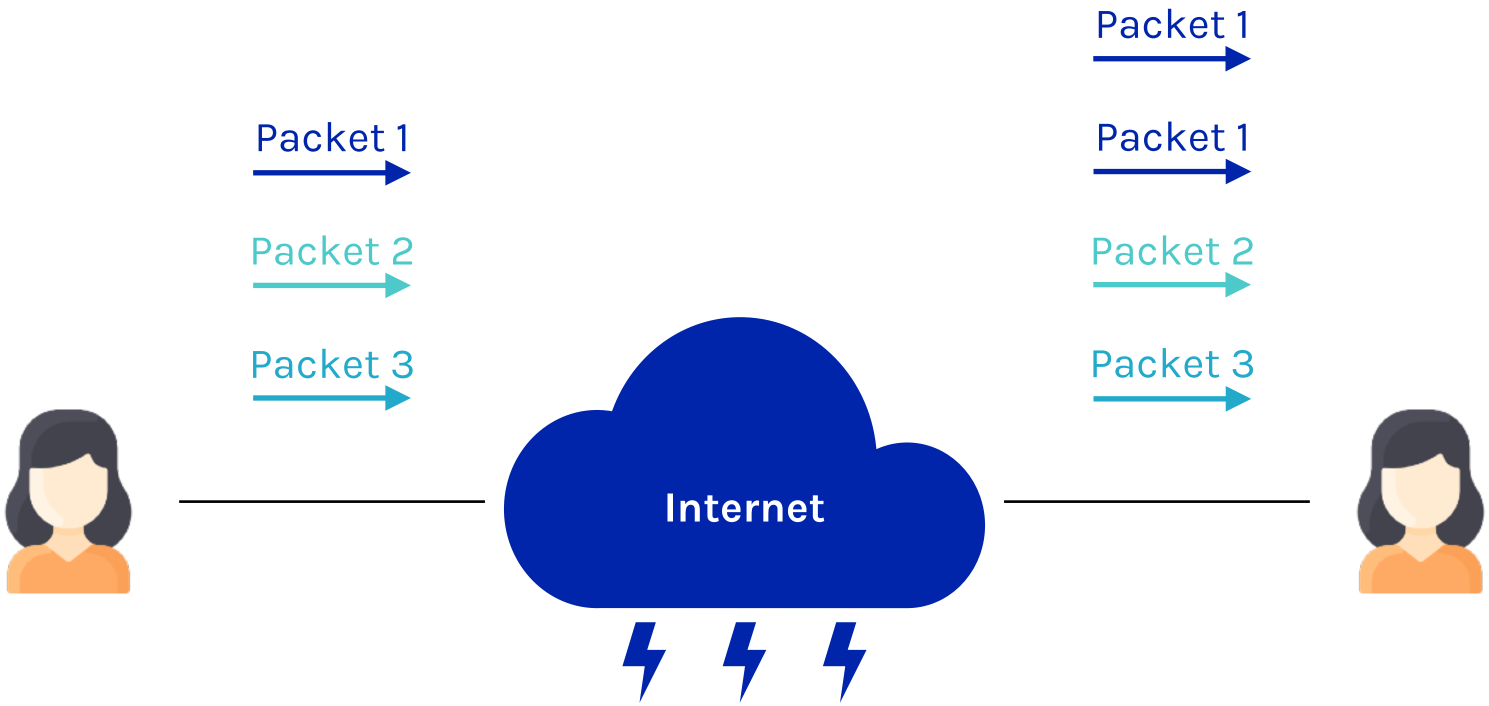
# Packet corruption



# Packet out-of-order



# Packet duplication



# Reliable transport

**Correctness**

If and only if…

**Tradeoffs**

Timeliness, efficiency…

**Mechanisms**

Go-Back-N…

# Correctness Conditions

# Correctness conditions for routing

A global forwarding state is valid **if and only if…**

Necessary and sufficient conditions

…there are **no dead ends** (no outgoing port undefined in the table)

…there are **no loops** (packets will never go around the same set of nodes)

# Correctness conditions for reliable transport

Attempt 1

A reliable transport design is valid if…

→ **…packets are delivered to the receiver**

Wrong

Consider a network is partitioned

We cannot say a transport design is incorrect if it does not work in a partitioned network

# Correctness conditions for reliable transport

Attempt 2

A reliable transport design is valid if…

**…packets are delivered to the receiver if and only if it was possible to deliver them**

Wrong

If the network is only available one instant in time, only an oracle would know when to send

We cannot say a transport design is incorrect if it does not know the unknowable

# Correctness conditions for reliable transport

Attempt 3

A reliable transport design is valid if…

**…it resends a packet if and only if it detects the previouse packet was lost or corrupted**

Wrong

Consider two cases:

- Packet made it to the receiver and all packets from receiver were dropped
- Packet is dropped on the way and all packets from receiver were dropped

In both cases, the sender has no feedback at all

**Does it resend or not?**

# Correctness conditions for reliable transport

Attempt 3

A reliable transport design is valid if…

**…it resends a packet if and only if it detects the previouse packet was lost or corrupted**

Wrong but better

It refers to what the design does (which it can control), not whether it always succeeds (which it cannot control)

# Correctness conditions for reliable transport

Attempt 4

A reliable transport design is valid if…

- a packet is always resent if it detects the previous packet was lost or corrupted
- a packet maybe resent at other times

Correct

# Correctness conditions for reliable transport

A reliable transport mechanism is correct **if and only if** it resents all dropped or corrupted packets

**Sufficient** The mechanism will always keep trying to deliver undelivered packets

**Necessary** If it ever lets a packet go undelivered without resending it, it is not reliable

It is okay to give up after a while but the sender must notify the application about it

# Tradeoffs

# Design goals of reliable transport

**Timeliness**
(minimize time until data is transferred)

**Efficiency**
(optimal use of available bandwidth)

**Fairness**
(play well with concurrent transfers)

**Correctness**
(ensure data is delivered in order and untouched)

# Example transport mechanism

```
for word in list:
    send_packet(word)
    set_timer()

    // time out, retransmit
    upon timer going off:
        if no ACK received:
            send_packet(word)
            reset timer()
    // success
    upon ACK:
        pass
```

**Sender**

```
receive_packet(p)
// received and intact
if check(p.payload) == p.checksum:
    // confirm to the sender
    send_ack()

    // deliver to the APP
    if p.payload not delivered:
        deliver_word(p.payload)
// ignore if corrupted
else:
    pass
```

**Receiver**

# Tradeoff between timeliness and efficiency

```
for word in list:
    send_packet(word)
    set_timer()

    // time out, retransmit
    upon timer going off:
        if no ACK received:
            send_packet(word)
            reset timer()
    // success
    upon ACK:
        pass
```

**Sender**

```
receive_packet(p)
// received and intact
if check(p.payload) == p.checksum:
    // confirm to the sender
    send_ack()

    // deliver to the APP
    if p.payload not delivered:
        deliver_word(p.payload)
// ignore if corrupted
else:
    pass
```

**Receiver**

# Tradeoff between timeliness and efficiency

Timeliness

**Small timers**

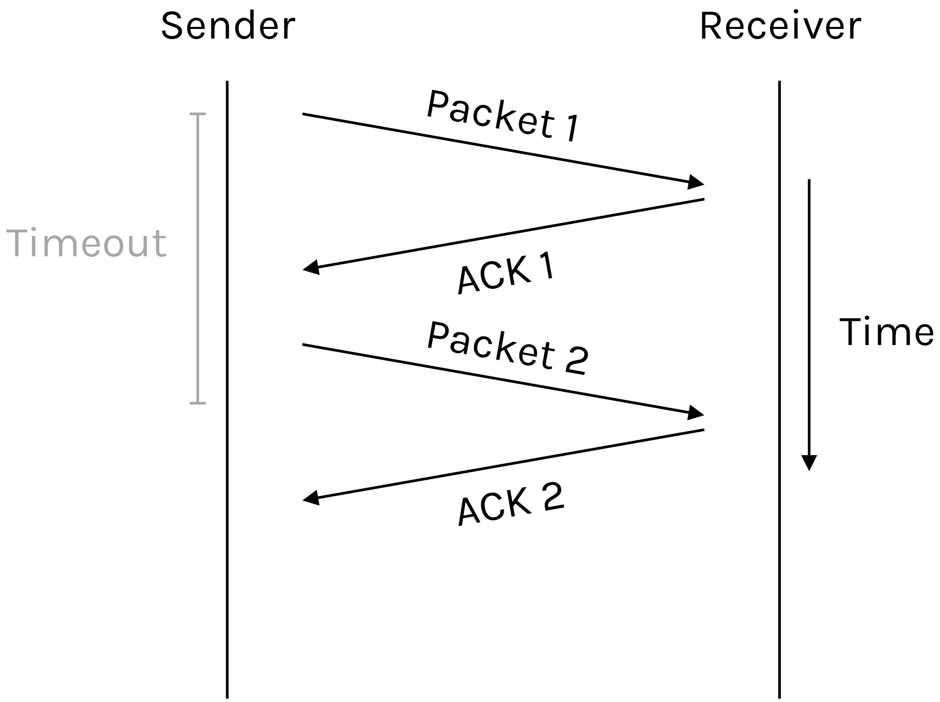
- Faster retransmission
- Might lead to unnecessary retransmissions

Efficiency

**Large timers**

- Slow retransmission
- Avoid unnecessary retransmissions

# Poor timeliness, nonetheless




Only one packet per round-trip-time (RTT)

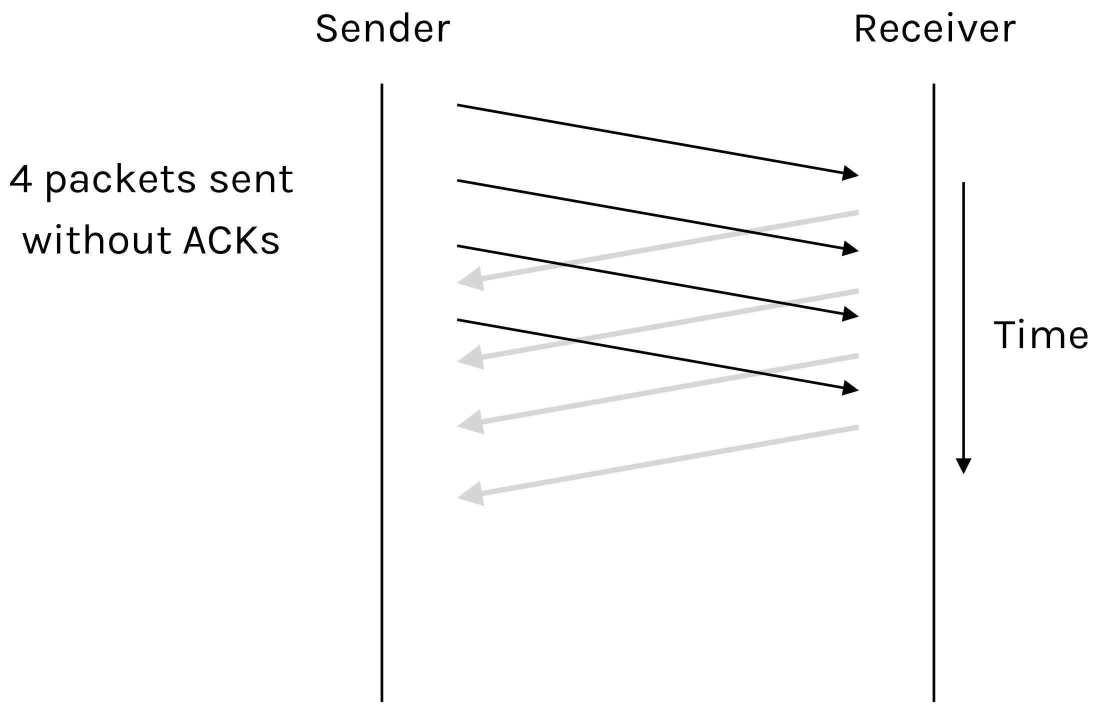
# Improvement idea: multiple packets simultaneously

1. **Add a sequence number inside each packet**

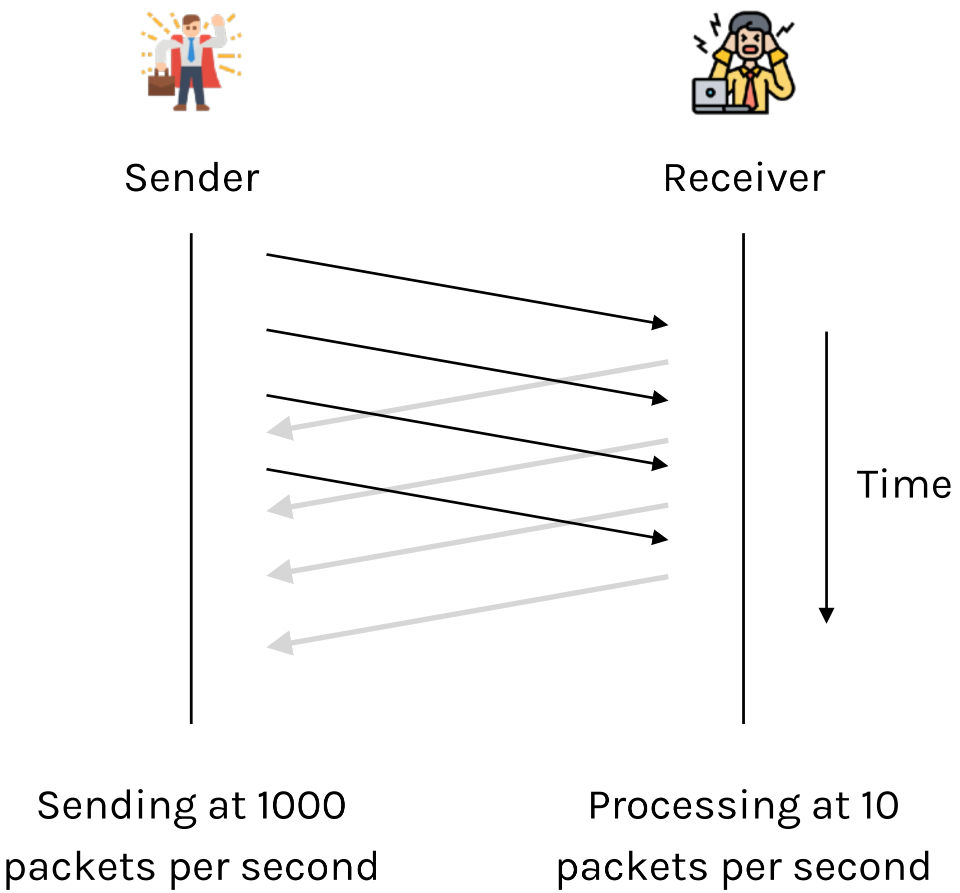
2. **Add buffers to the sender and receiver**

   Sender: store packets sent & not ACKed
   Receiver: store out of order packets received

# Improved timeliness



# Overwhelmed receiver



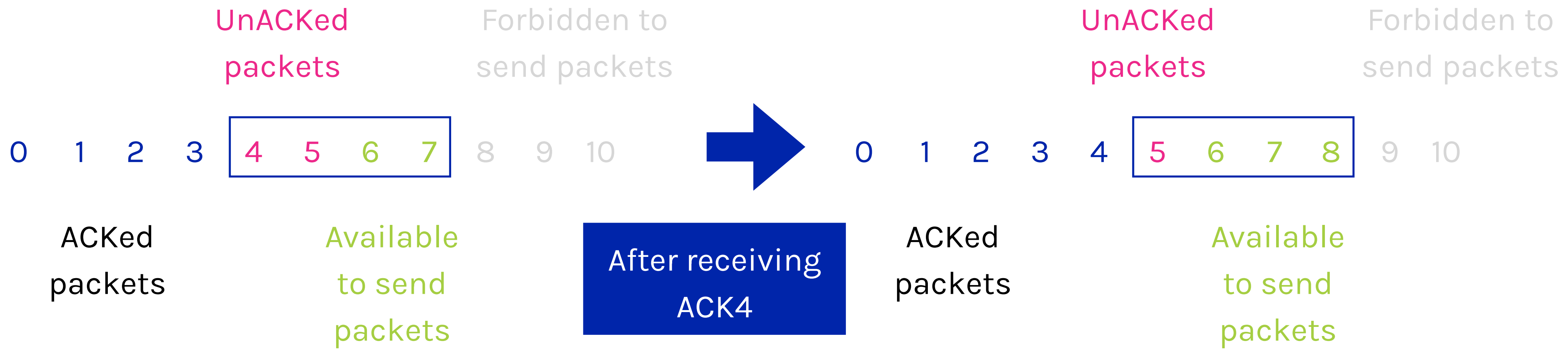
# Flow control

Sender keeps a list of the sequence numbers it can send
(known as the **sending window**)

Receiver also keeps a list of the acceptable sequence numbers
(known as the **receiving window**)

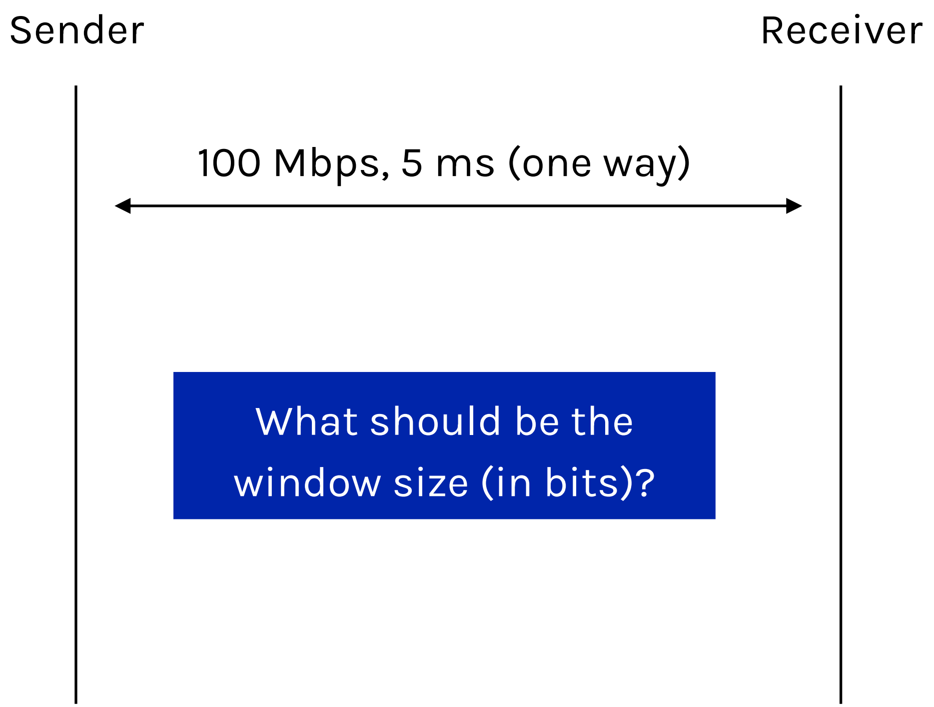
Sender and receiver negotiate the window size
(ensure **sending window <= receiving window**)

# Sending window example



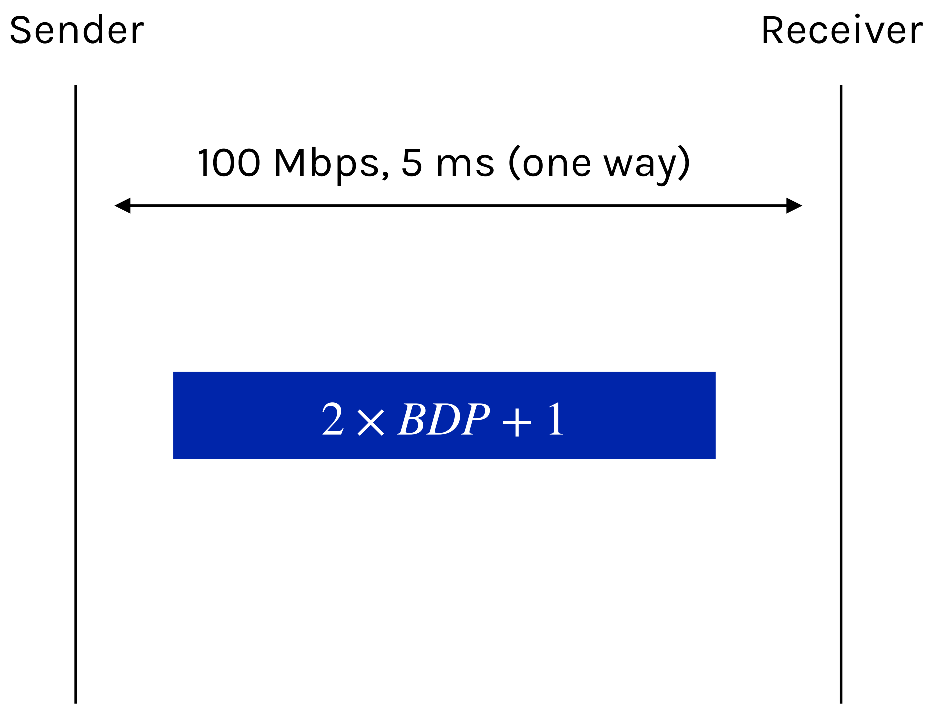
# Window sizing

**How big should the window be to maximize timeliness?**



# Window sizing

**How big should the window be to maximize timeliness?**



# Efficiency

**Receiver feedback**

(How much information does the sender get?)

**Behavior upon losses**

(How does the sender detect and react to losses?)

# Idea: ACKing individual packets

**Provides detailed feedback, but triggers unnecessary retransmission upon losses**

**Advantages**

- Know fate of each packet
- Simple window algorithm (multiple instances of the single-packet algorithm)
- Not sensitive to reordering

**Disadvantages**

- Loss of an ACK packet requires a retransmission

Causes unnecessary retransmission

# Idea: cumulative ACKs

| | |
|---|---|
| **Approach** | ACK the **highest sequence number** for which all the previous packets have been received |
| **Advantages** | Recover from lost ACKs |
| **Disadvantages** | Confused by reordering<br>Imcomplete information about which packets have arrived (which causes unnecessary retransmission) |

# Idea: cumulative ACKs improved

| | |
|---|---|
| **Approach** | List **all packets that have been received**<br>Highest cumulative ACK, plus any additional packets |
| **Advantages** | Complete information, resilient form of individual ACKs |
| **Disadvantages** | High overhead (hence lowering efficiency)<br>Especially when there are large gaps between received packets |

# Loss detection via ACK: individual ACKs

Assume packet 5 is lost, but no others

ACK stream

1

2

3

4

6

7

...

Sender can infer that 5 is missing, and resend 5 after $k$ subsequent packets

# Loss detection via ACK: cumulative ACKs

Assume packet 5 is lost, but no others

ACK stream

1

2

3

4

4 sent when 6 arrives

4 sent when 7 arrives

…

Duplicate ACKs

# Duplicate ACKs

ACK stream

**1**

**2**

**3**

**4**

**4 sent when 6 arrives**

**4 sent when 7 arrives**

**…**

Lack of ACK progress means that 5 has not made it

Stream of ACKs means that (some) packets are delivered

Sender could trigger resend upon receiving $k$ duplicate ACKs

Q: What does the sender resend? Only 5 or 5 and everything after?

# Loss detection via ACK: full information

Assume packet 5 is lost, but no others

ACK stream

**up to 1**

**up to 2**

**up to 3**

**up to 4**

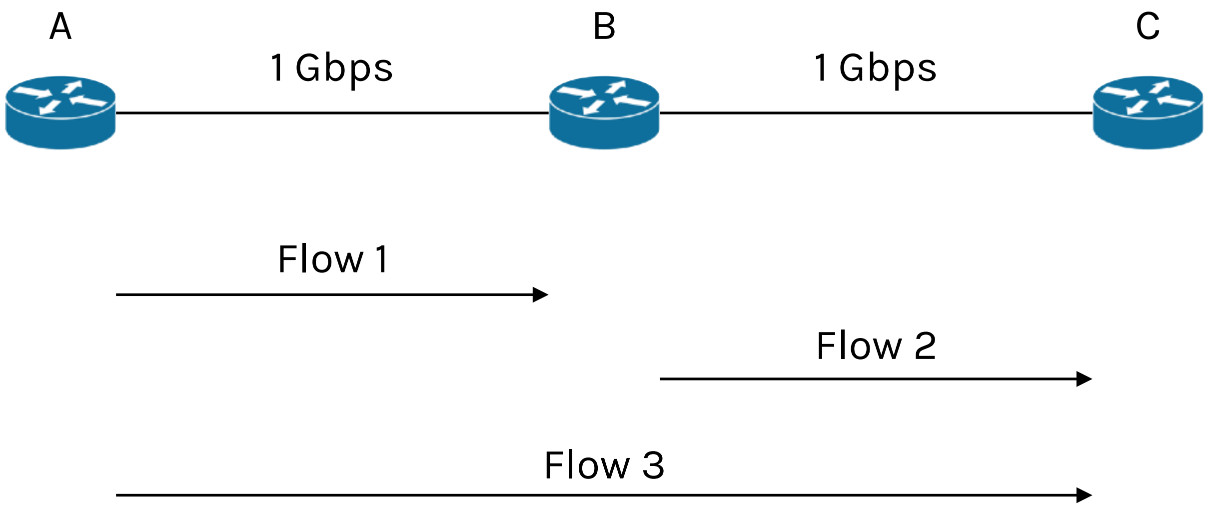
**up to 4, plus 6** → Sender learns that 5 is missing, and retransmits after $k$ packets

**up to 4, plus 6-7**

**...**

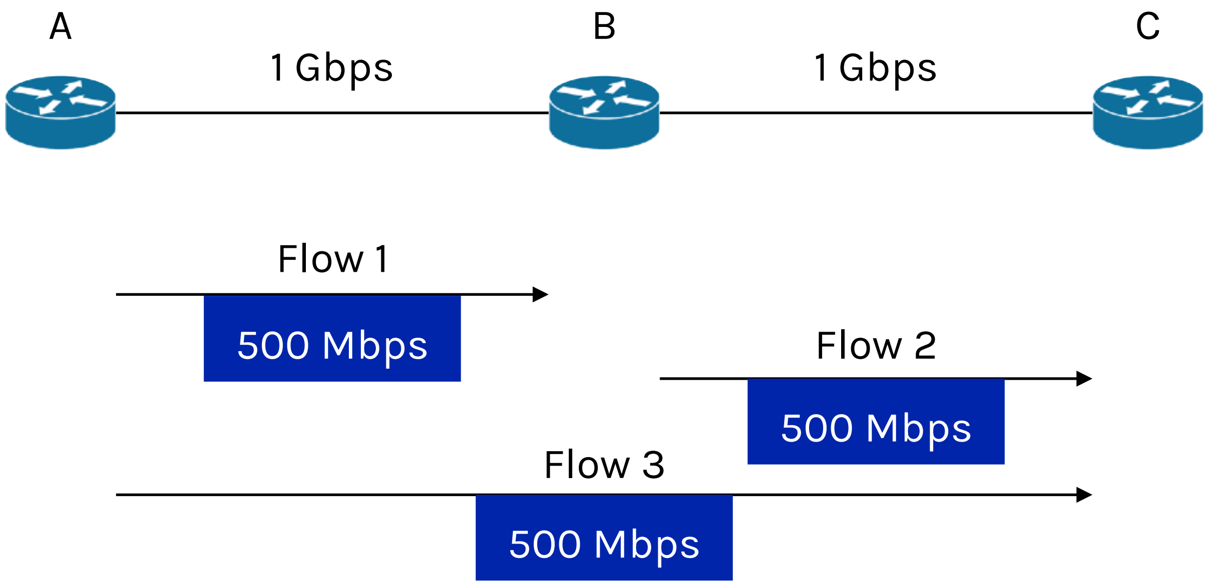
# Fairness

**Fair allocation of bandwidth among all entities using the transport mechanism**



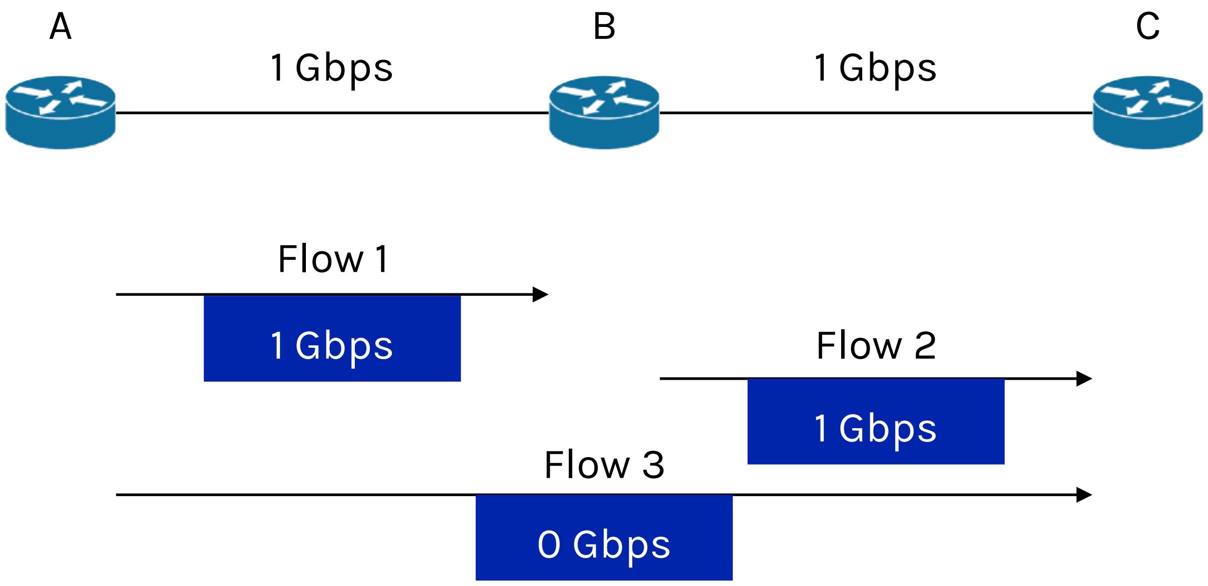

What is the fair allocation for the 3 flows?

# Equal allocation



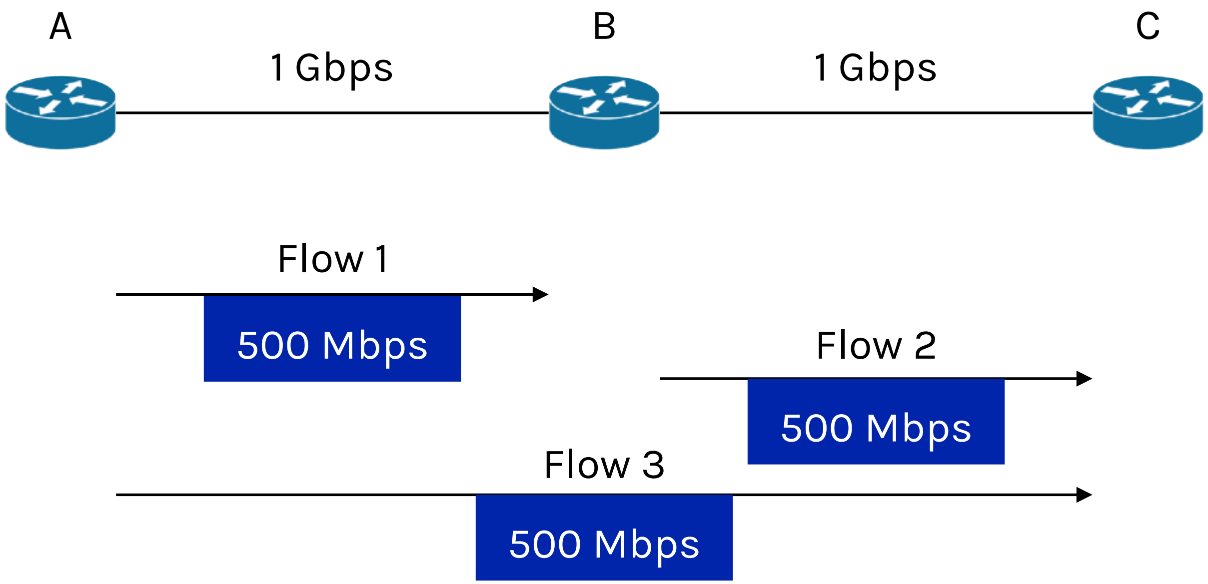

An equal allocation is certainly "fair", but the efficiency is not optimal: total traffic is 1.5 Gbps

# Unfair but more efficient allocation



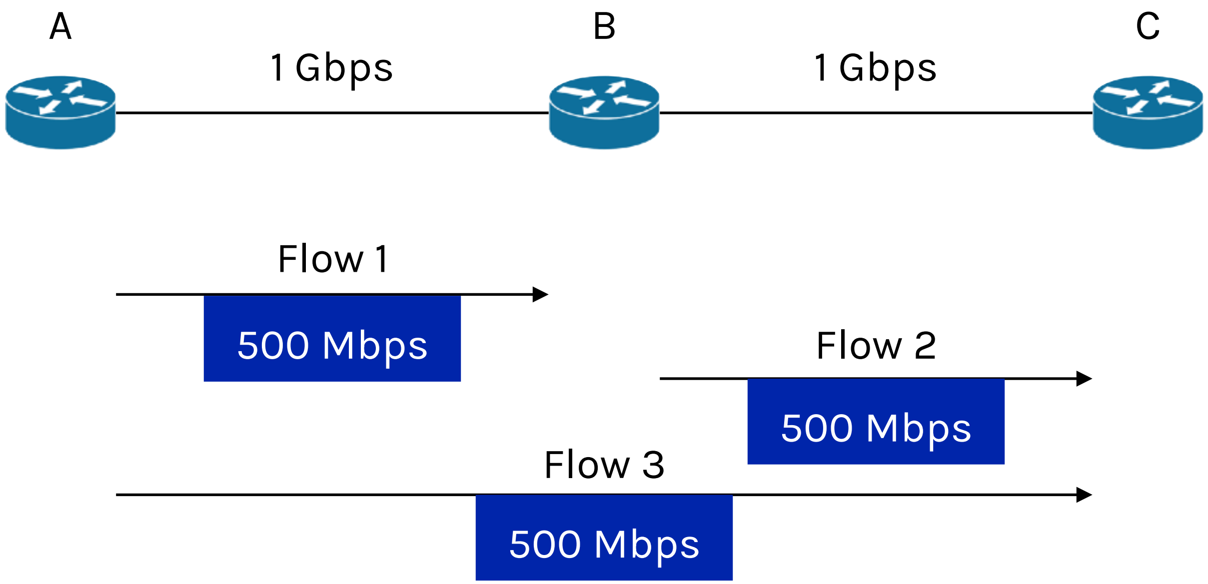

An unfair but more efficient allocation: total traffic is 2 Gbps

# What is fairness?




Equal-per-flow is not really fair as A-C crosses two links: it uses more resources

# What is fairness?




Equal-per-flow is fair as A gets 1 Gbps for 2 flows while B gets 500 Mbps for 1 flow

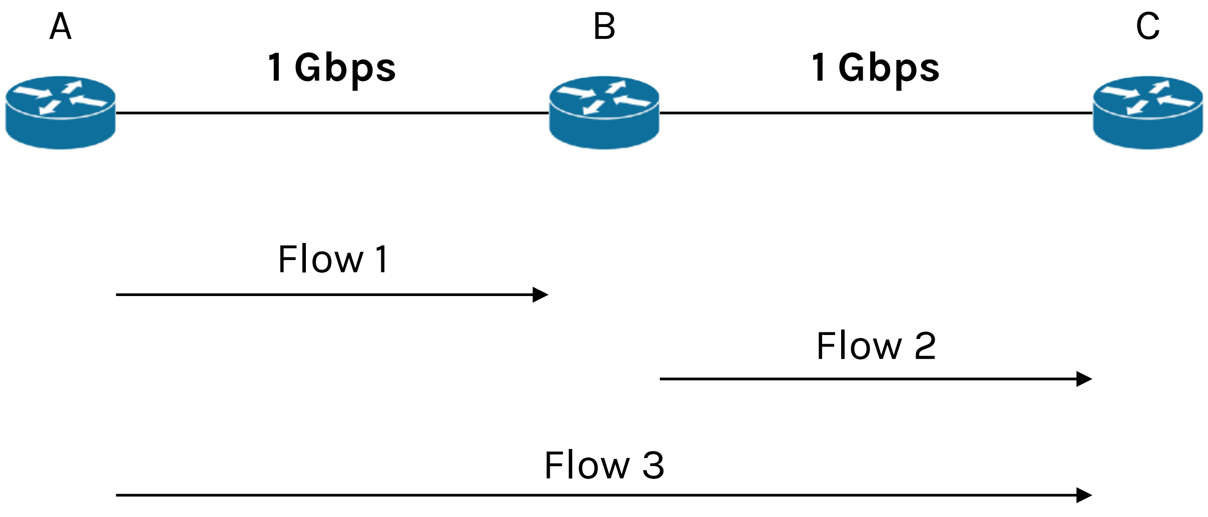
# Max-min fairness

**Intuitively, give users with small demands what they want, and evenly distribute the rest**

**Max-min fair allocation**

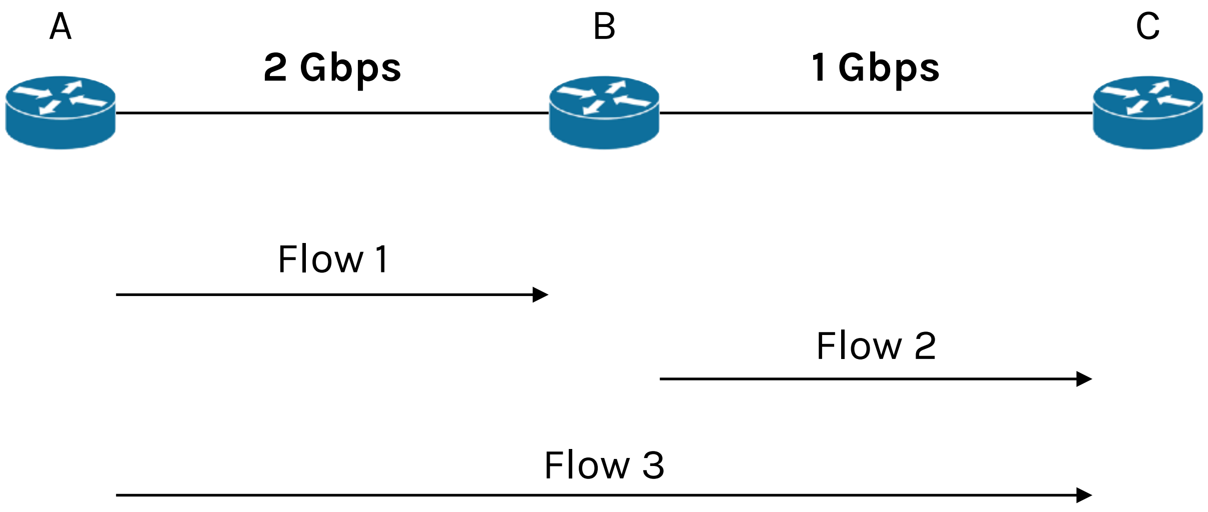
- Step 1: start with all flows at rate 0
- Step 2: increase the rate of flows until there is a new bottleneck in the network
- Step 3: hold the fixed rate of the flows that are bottlenecked
- Step 4: go to step 2 for the remaining flows
- Done!

# Max-min fair allocation




What is the max-min fair allocation?

# Max-min fair allocation




What is the max-min fair allocation?

# Approximating max-min fair allocation

Intuition

**Progressively increase the sending window size**

max = receiving window

**Whenever a loss is detected, decrease the window size**

Signal of congestion

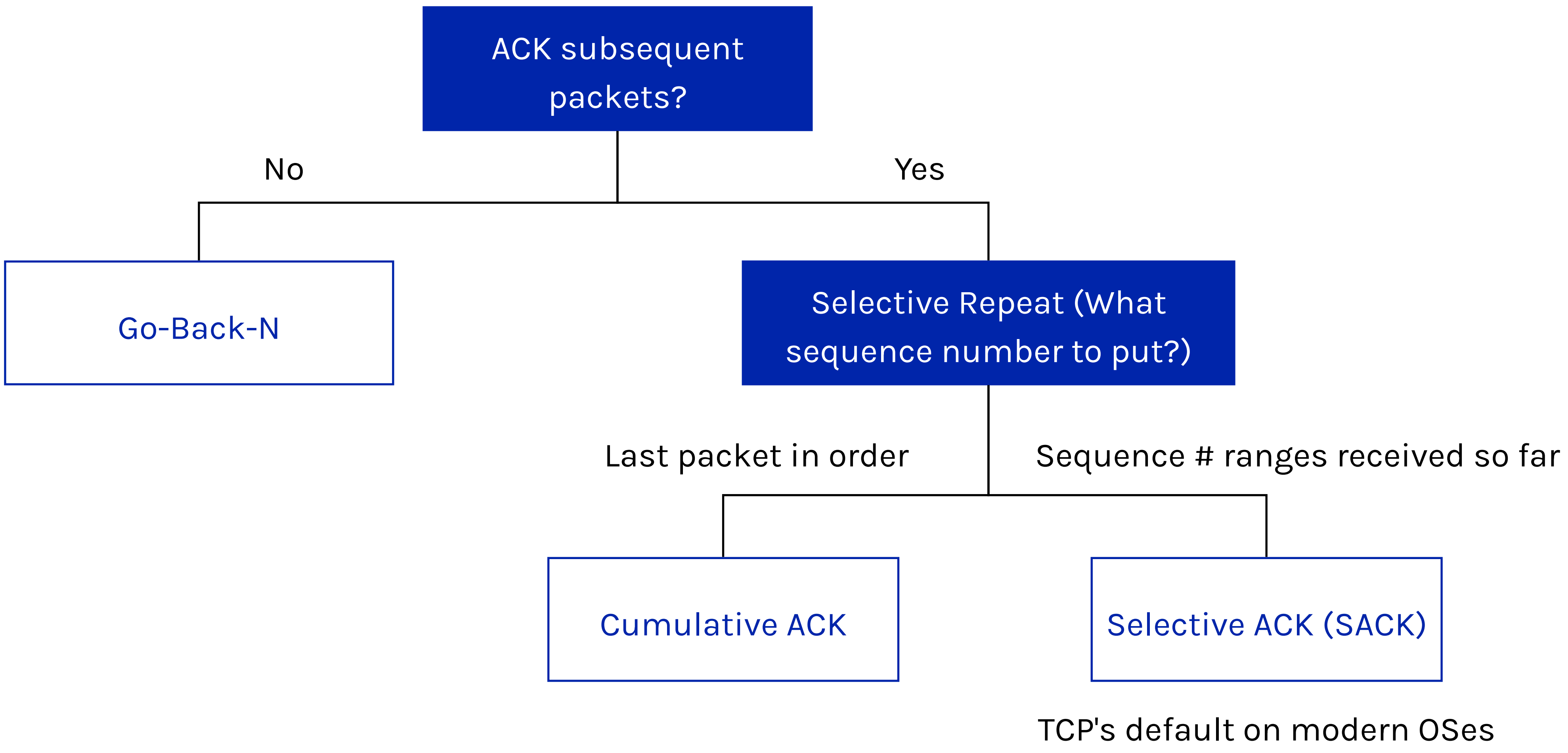
**Repeat**

# Different ACK schemes

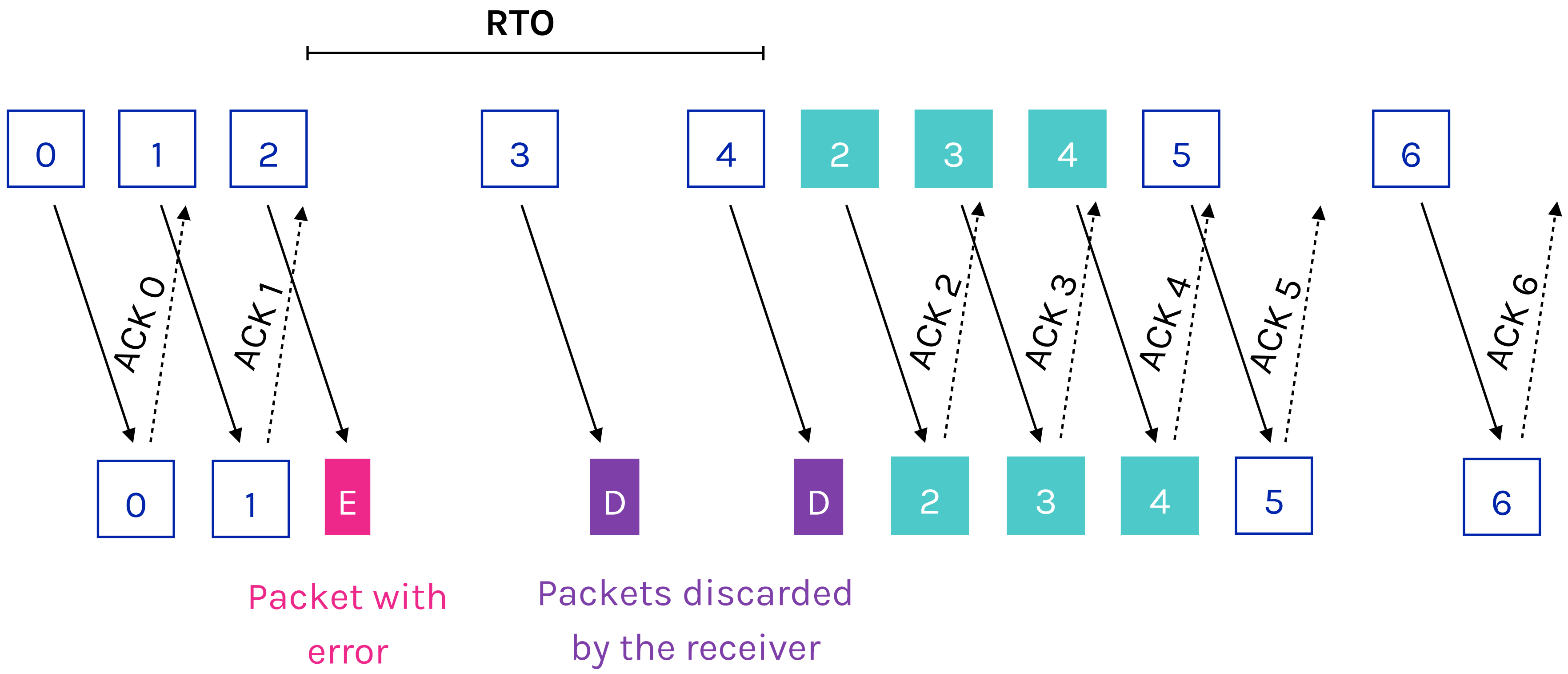
| | Unreliable network situation | | |
|---|---|---|---|
| | **Reordering** | **Long delays** | **Packet duplicates** |
| **Individual ACKs** | No problem | Useless timeouts | No problem |
| **Full feedback** | No problem | Useless timeouts | No problem |
| **Cummulative ACKs** | Create duplicate ACKs (maybe treated as packet loss) | Useless timeouts | Create duplicate ACKs (maybe treated as packet loss) |

# Some Transport Mechanisms

# Transport mechanisms categorization

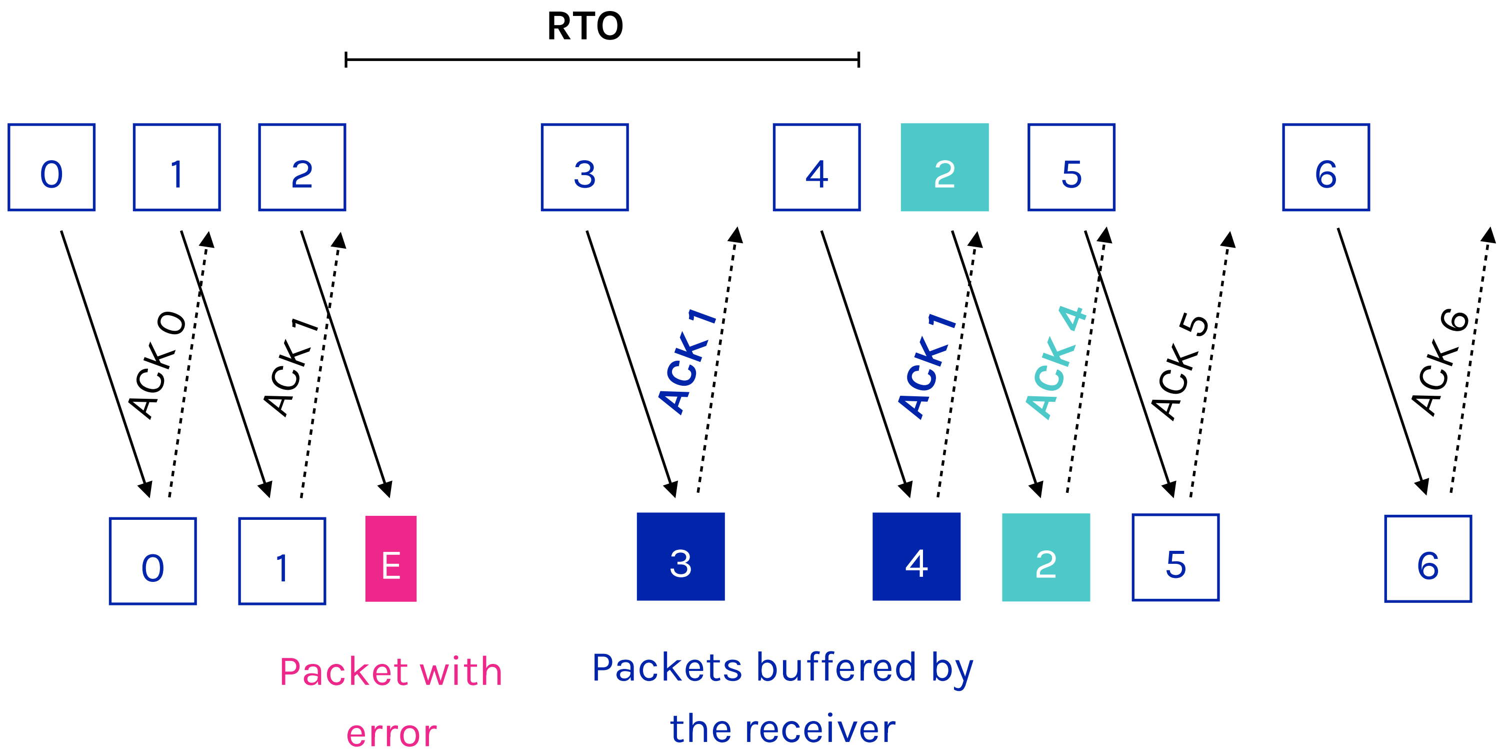


# Go-Back-N (GBN)



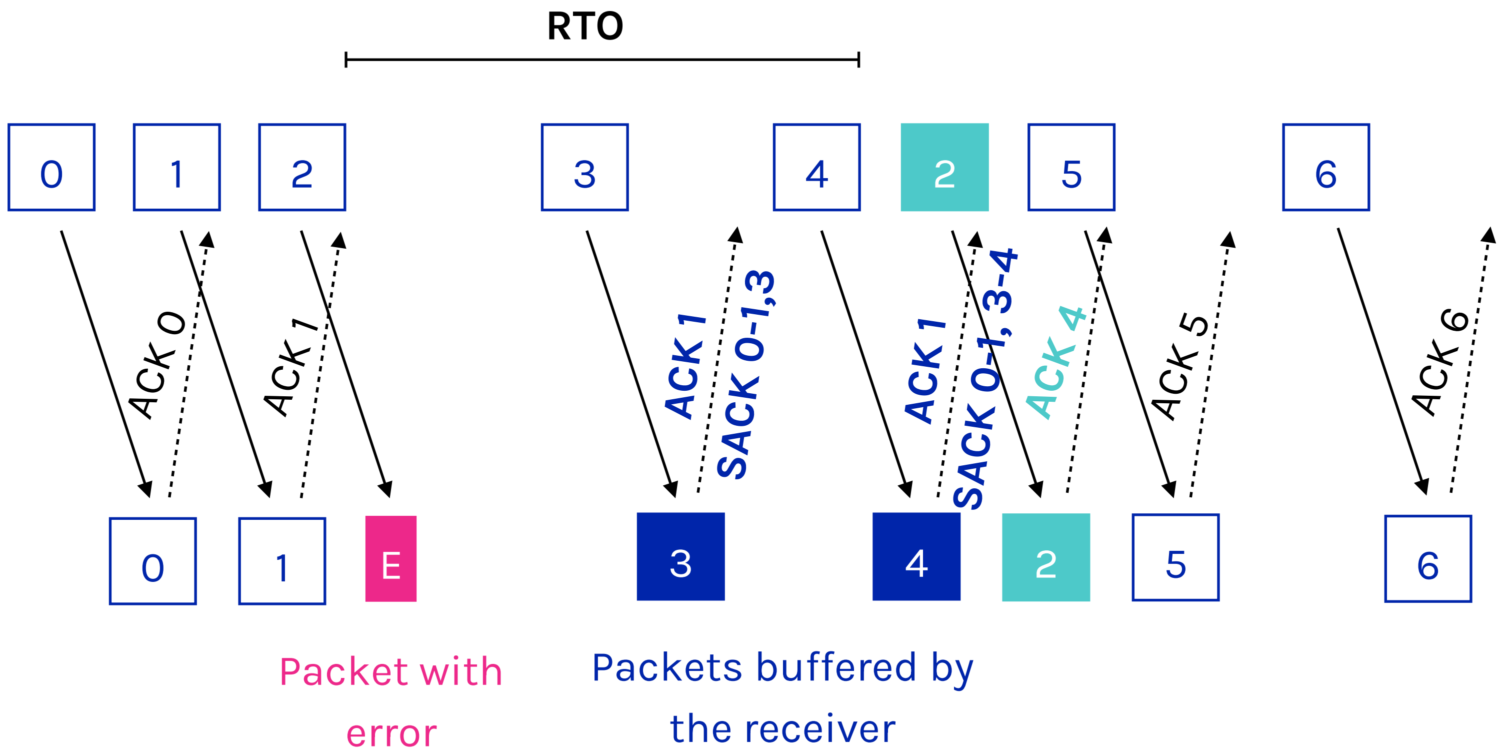

The sender spends time to retransmit data the receiver has already seen

# Selective repeat with cumulative ACK



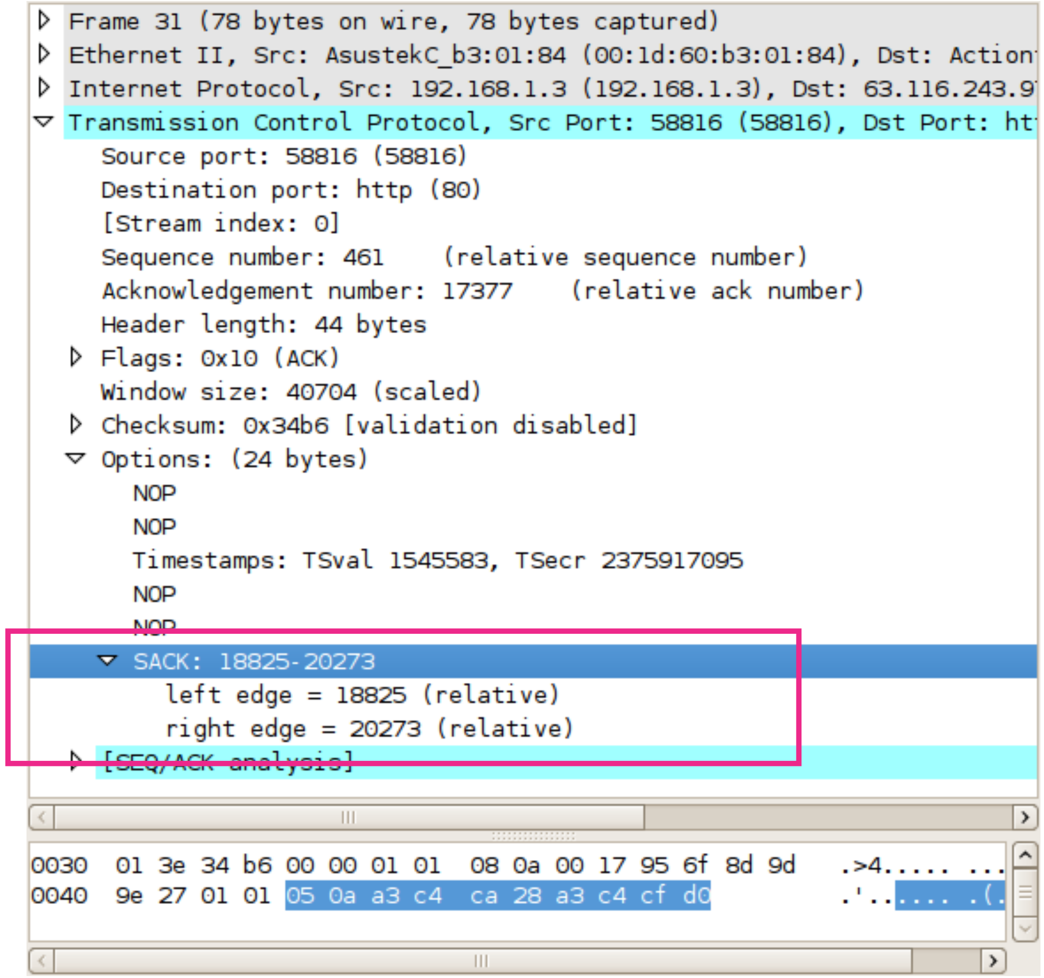

The sender only retransmits the first unACKed packet, not all its successors

# Selective repeat with selective ACK




The receiver keeps ACKing the first in-order sequence number, plus the packets that have been received after the missing packet

# SACK in action



# Summary

**Reliable delivery**

- Unreliable network situations

**Correctness conditions**

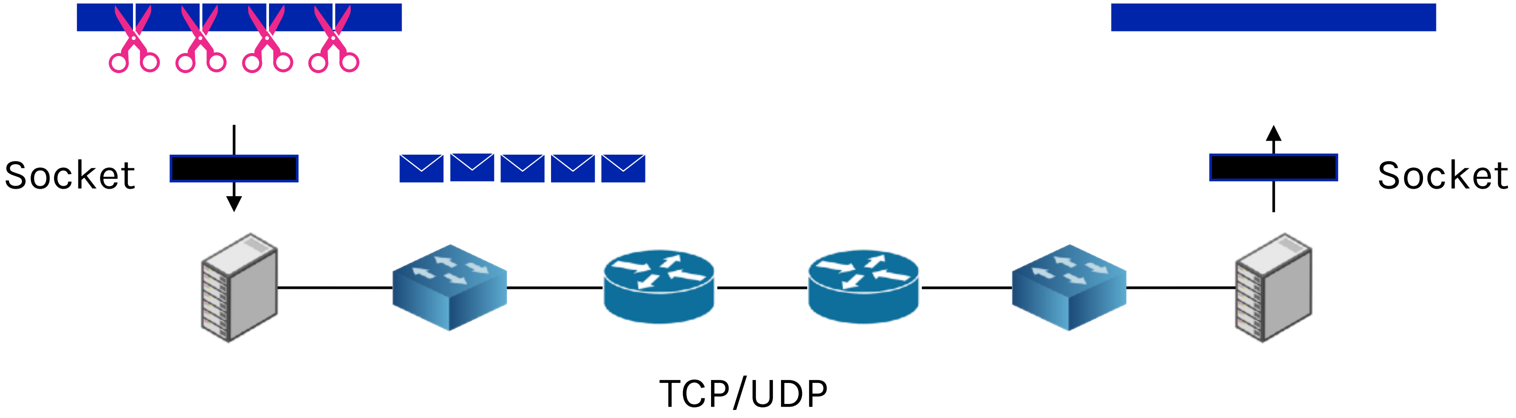
**Tradeoffs**

- Timeliness
- Efficiency
- Fairness

**Some transport mechanisms**

- Go-Back-N
- Selective repeat with cumulative ACK
- Selective repeat with selective ACK

# Next time: transport layer




What are the popular transport protocols?

# Further reading material

**James F. Kurose, Keith W. Ross. Computer Networking: A Top-Down Approach (5th edition).**

- Section 3.1: Introduction and Transport-Layer Services
- Section 3.4: Principles of Reliable Data Transfer

# Guest lecture (January 26, 2024, 13:00-15:00)

**Balakrishnan Chandrasekaran**

Assistant Professor

VU Amsterdam

Max-Planck-Institut für Informatik

TU Berlin

Duke University (PhD)

https://balakrishnanc.github.io/