Computer Networks

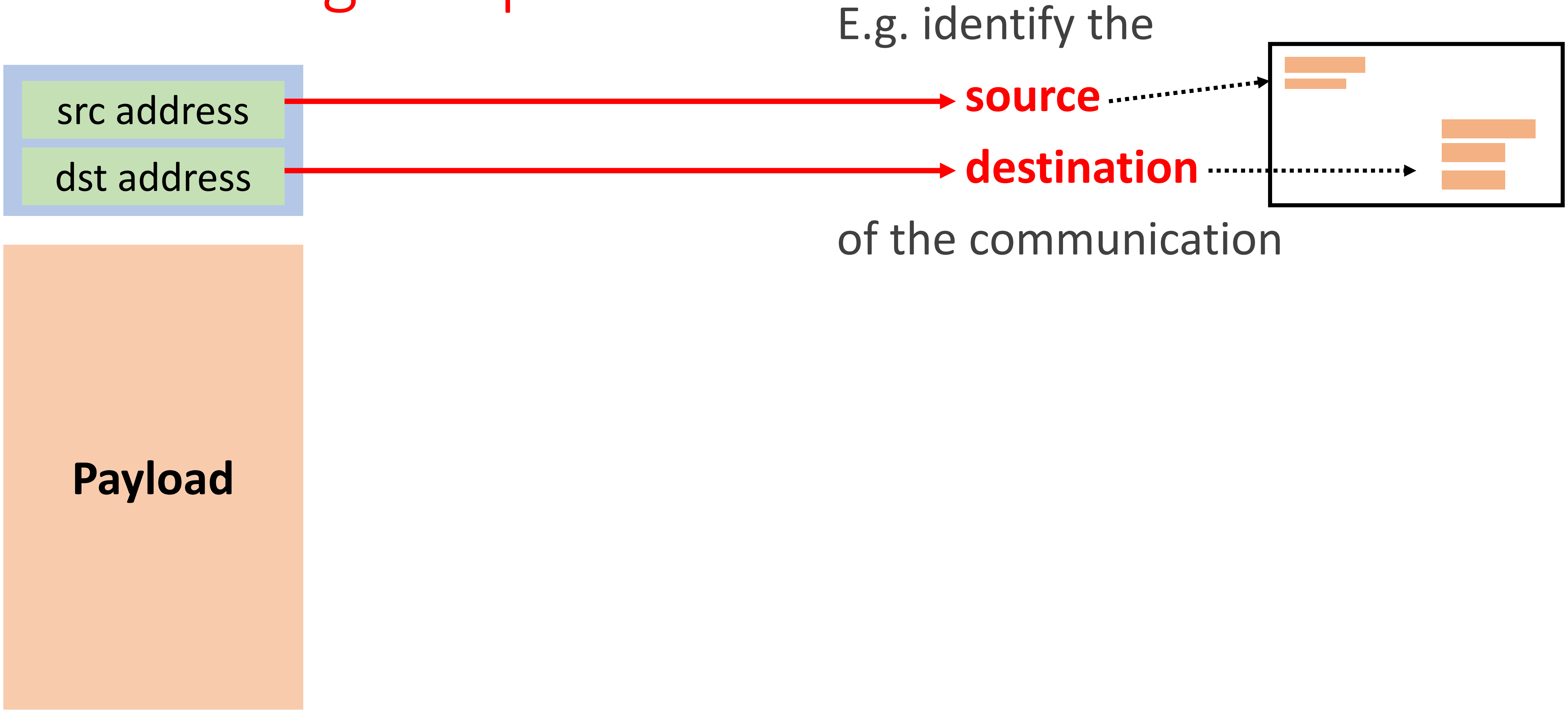
Sándor Laki
ELTE-Ericsson Communication Networks Laboratory
ELTE FI – Department Of Information Systems
lakis@elte.hu
http://lakis.web.elte.hu

Eötvös Loránd University

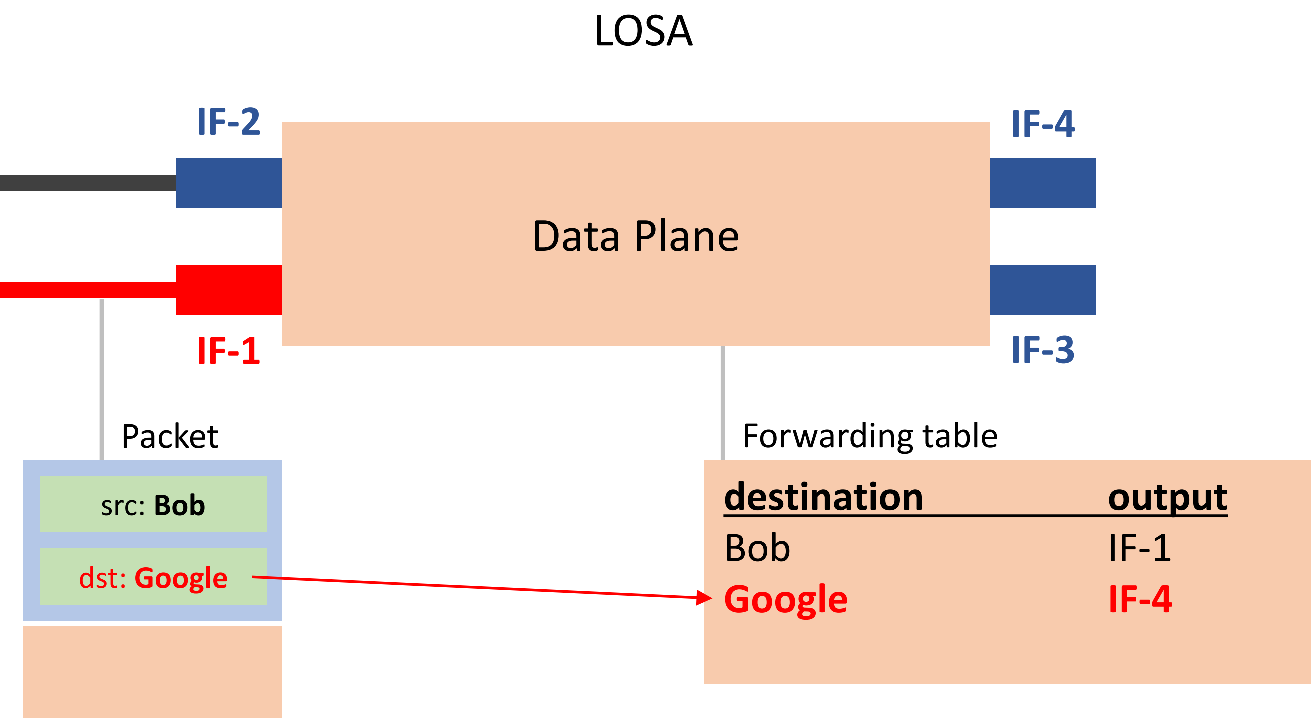
Based on the slides of Laurent Vanbever.
Further inspiration: Scott Shenker & Jennifer Rexford & Phillipa Gill

Last week on
Computer Networks

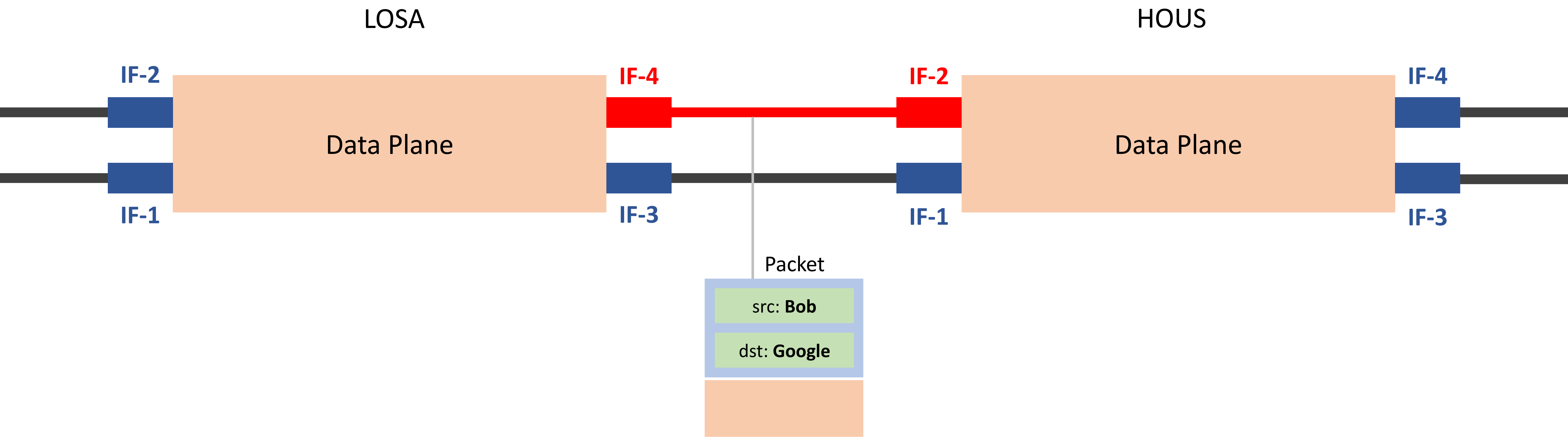
The header contains metadata needed for forwarding the packet



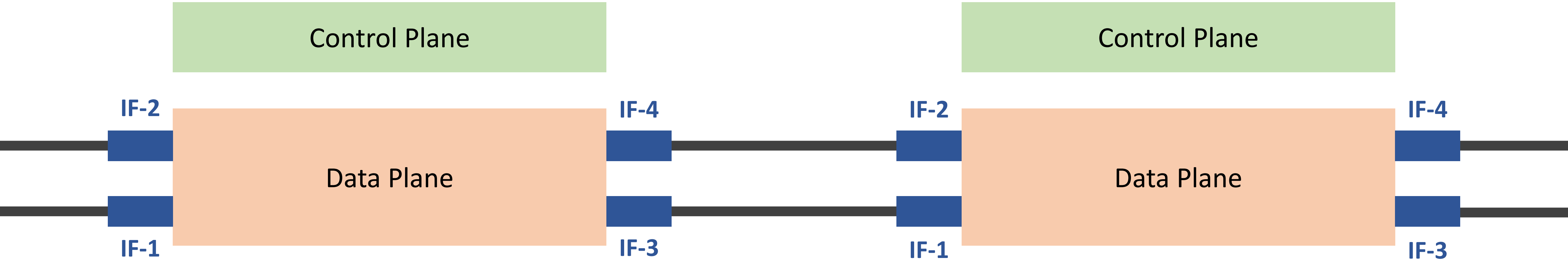
According to the fwd table, the packet should be directed to IF-4



According to the fwd table, the packet should be directed to IF-4



In addition to a data plane,
routers are also equipped with a **control plane**



While forwarding is a **local** process, routing is inherently a **global** process

A router should know how the network looks like
for directing the packet towards the destination.

Valid states

[Theorem]

A global forwarding state is valid iff (iff = <u>if and only if</u>)

A) there are no dead ends

dead end = i.e. no outgoing port defined in the table for a given dst

B) there are no loops

loop = i.e. packets going around the same set of nodes

Existing routing protocols differ in how they avoid loops

Essentially, there are three ways to compute valid routing state

Intuition	Example
1) Use tree-like topologies	*Spanning-tree*
2) Rely on global network view	*Link-state routing* *SDN*
3) Rely on distributed computation	*Distance vector routing* *BGP*

This week

Fundamental challenges – Part II

Reliable Transport

over an unreliable network…

Reliable Transport

In the Internet, reliability is ensured by the end hosts,

not by the network**!!!**

It is implemented in L4 (Transport Layer)!

Reliability in L4, just above the Network layer

goals

Keep the network simple, dumb

make it relatively "easy" to build and operate a network

Keep applications as network "unaware" as possible

a developer should focus on its app, not on the network

design

Implement reliability in-between, in the networking stack

relieve the burden from both the app and the network

Reliability in L4

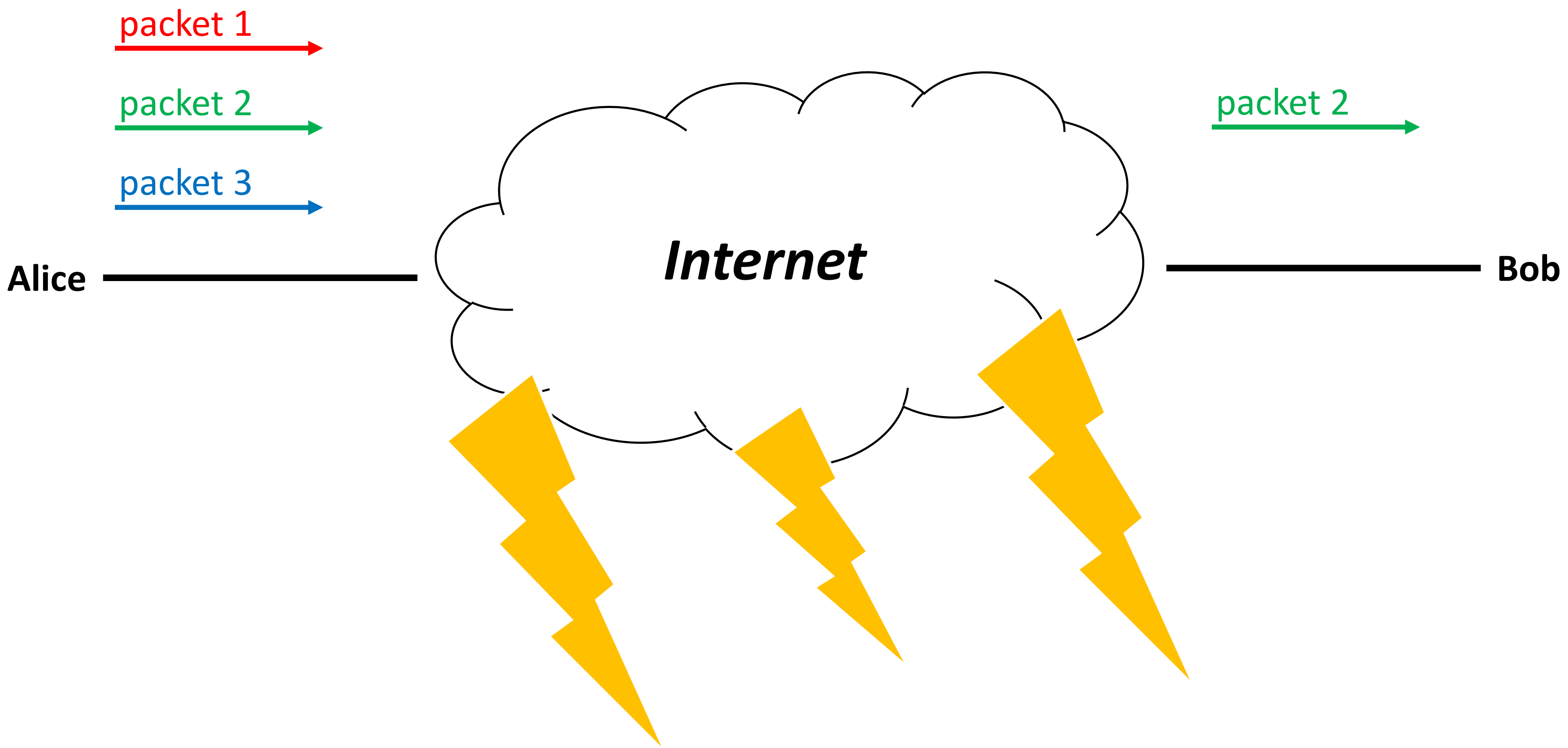
	layer	
	Application	
L4	**Transport**	**reliable end-to-end delivery**
L3	**Network**	global best-effort delivery
	Link	
	Physical	

On top of a best-effort delivery with quite poor guarantees

	layer	
	Application	
L4	**Transport**	**reliable end-to-end delivery**
L3	**Network**	**global best-effort delivery**
	Link	
	Physical	

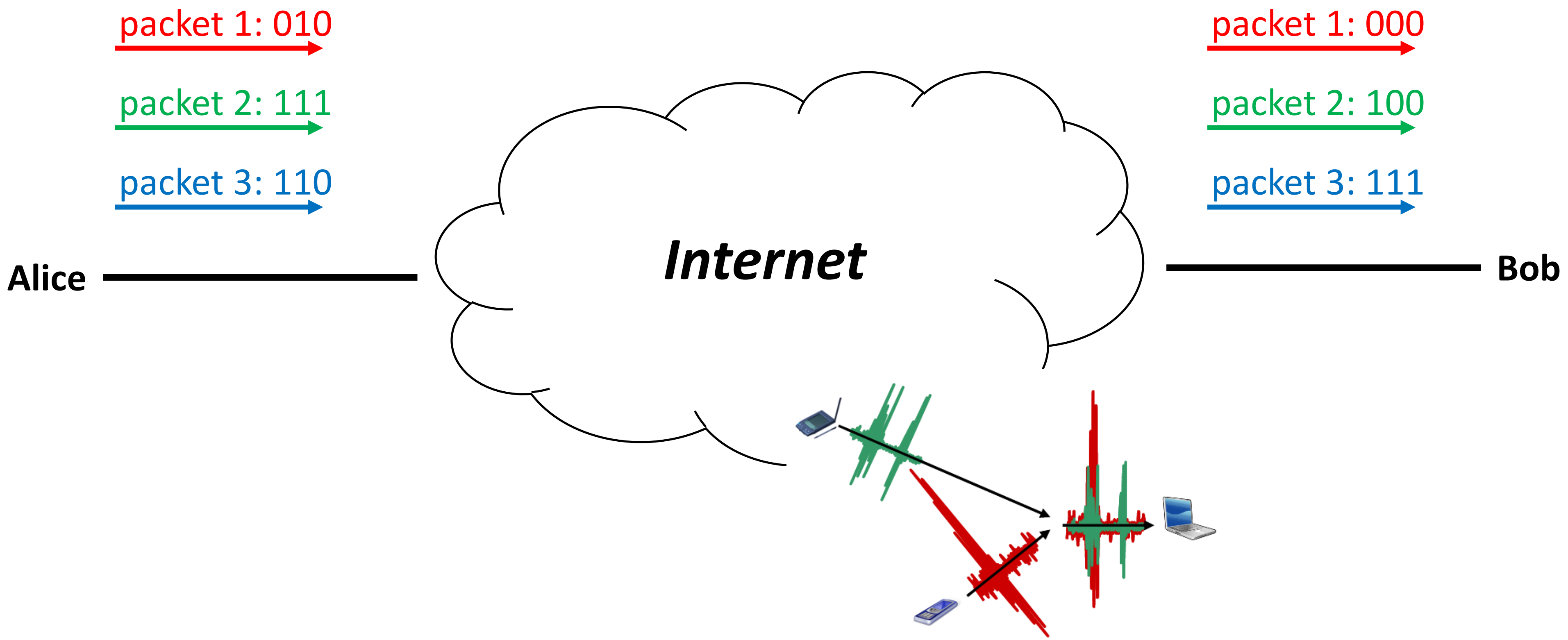
IP packets can get

lost or delayed



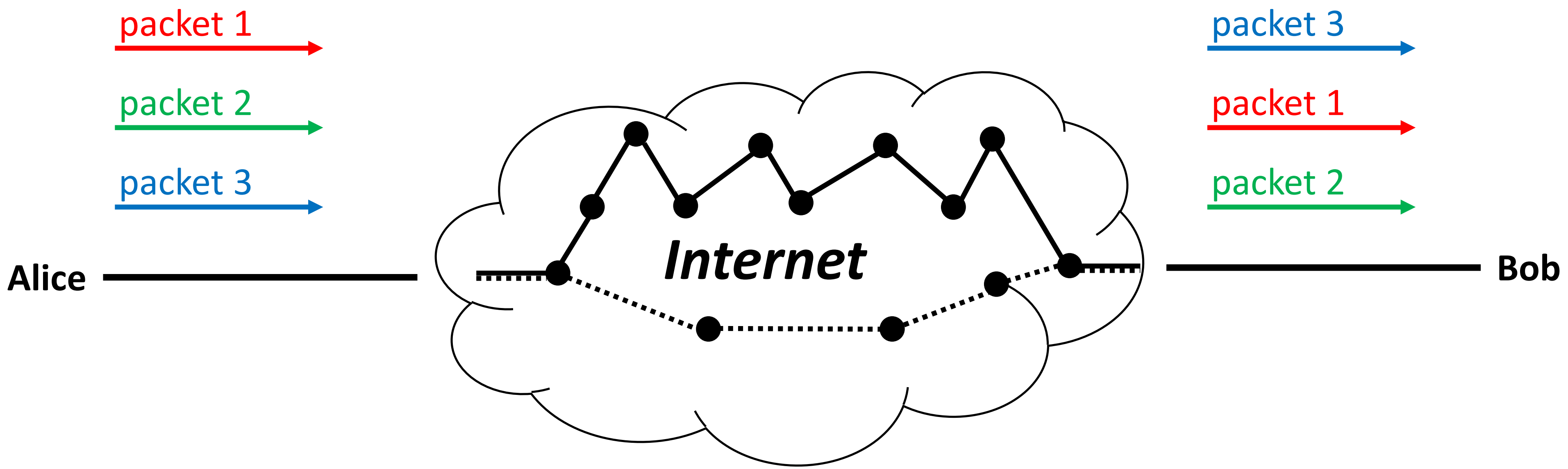
IP packets can get

corrupted



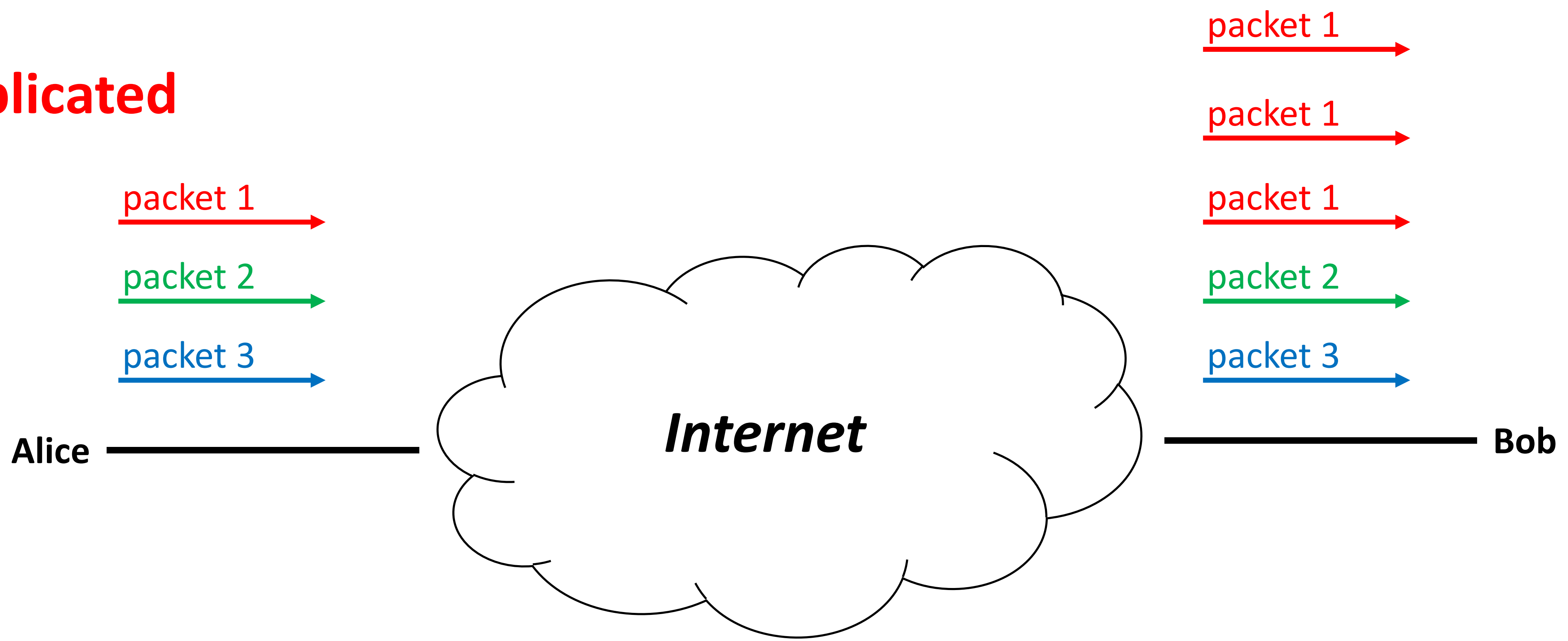
IP packets can get

reordered



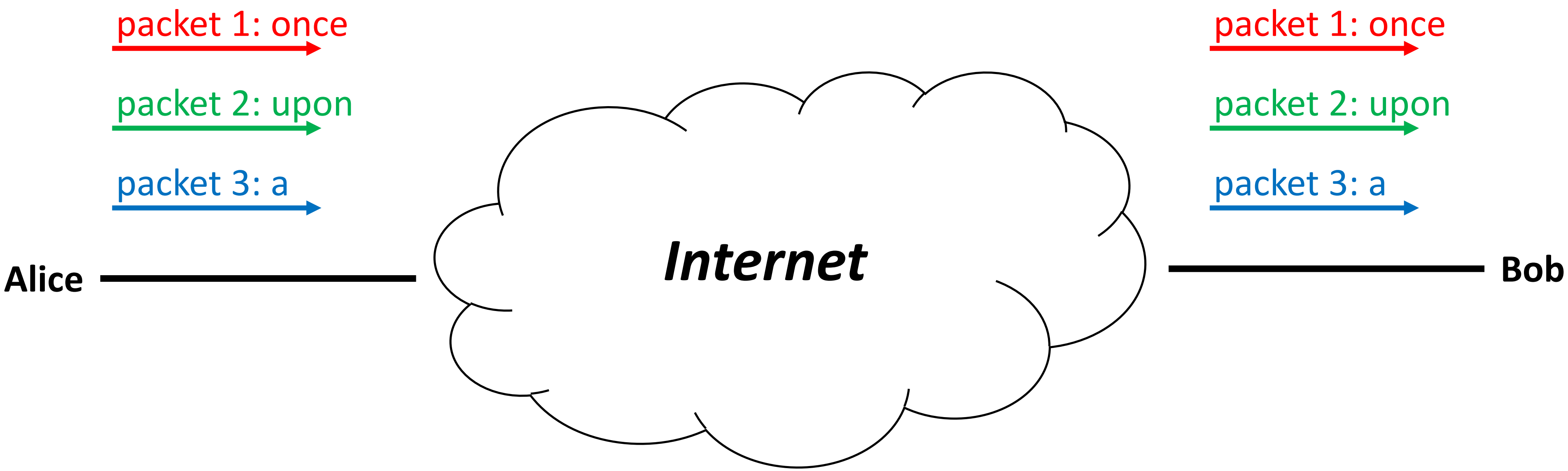
IP packets can get

duplicated



How to design a reliable transport protocol?

Let’s consider that Alice wants to transmit a text to Bob, word-by-word, via the Internet



How to design a reliable transport protocol running on Alice's and Bob's computer???

What properties are needed?

How to design a reliable transport protocol running on Alice's and Bob's computer???

correctness	**Bob should read exactly what Alice has typed**
	in the same order, without any gap

How to design a reliable transport protocol running on Alice's and Bob's computer???

correctness	**Bob should read exactly what Alice has typed** in the same order, without any gap
timeliness	**Bob should receive the complete text as fast as possible** minimize time until data is transferred

How to design a reliable transport protocol running on Alice's and Bob's computer???

correctness	**Bob should read exactly what Alice has typed** in the same order, without any gap
timeliness	**Bob should receive the complete text as fast as possible** minimize time until data is transferred
efficiency	**Minimize the use of bandwidth** don't send too many packets

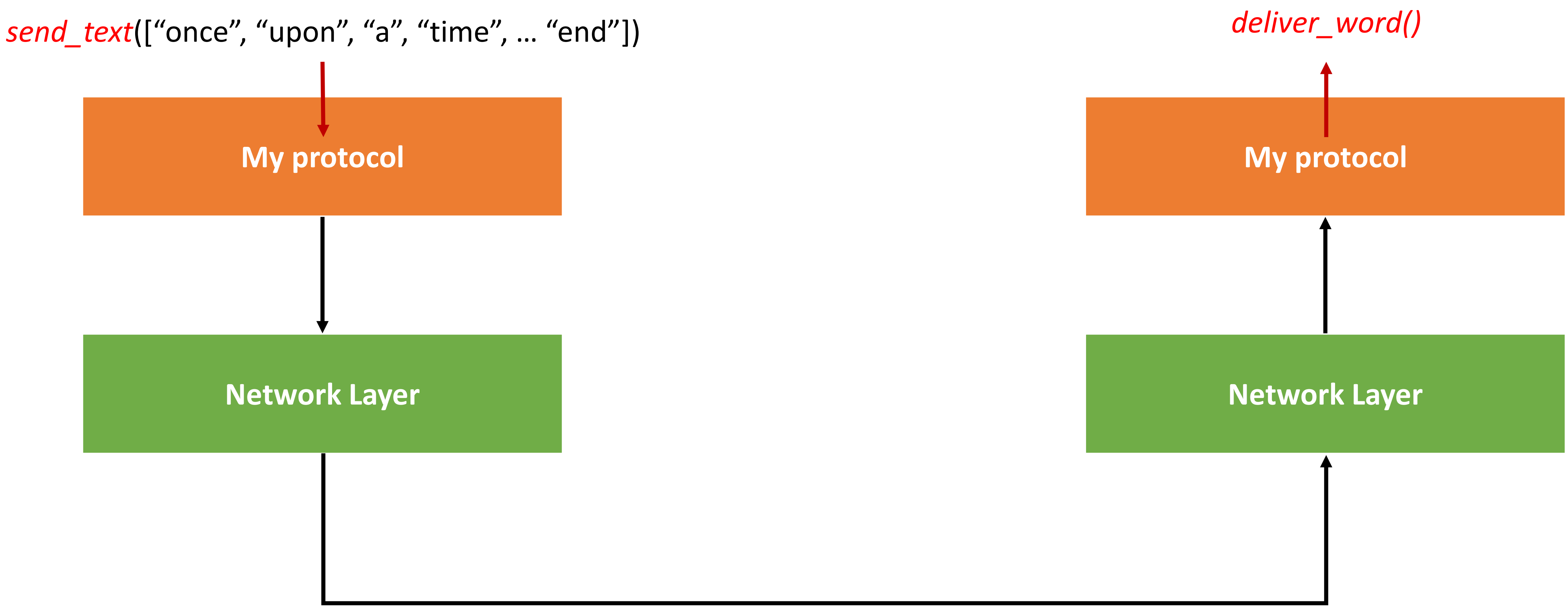
How to design a protocol that can deal with packet loss, corruption, reordering and duplication

The design includes

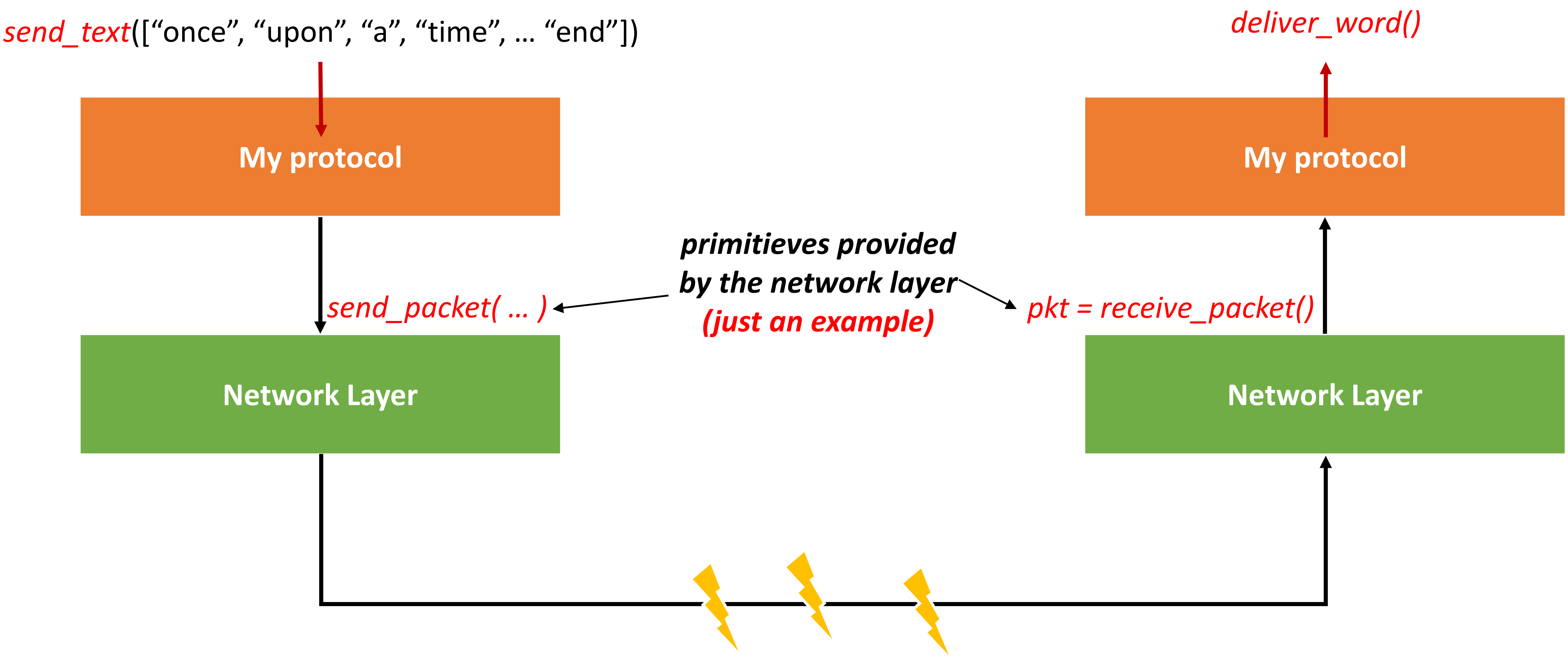
what **fields** do you add to the packets?

what **code** do you run on the end-points?

Interfaces like an API



Interfaces like an API



First attempt

Send at most 1 word/packet at a time.

Each packet can be lost, corrupted or duplicated…

First attempt – an example implementation

Python-like pseudo code of a sender (Alice)

```
def send_word(word_list):
    for word in word_list:
            send_packet(word)
            set_timer()

            upon timer going off:
                if no ACK received:
                        send_packet(word)
                        reset_timer()
            upon ACK:
                pass
```

Python-like pseudo code of a receiver (Bob)

```
receive_packet(p)

if p is not corrupted:
        send_packet(ACK)

        if p is not a duplicate:
                word = p.payload
                deliver_word( word )

else: # packet was corrupted
    pass
```

First attempt

Send at most 1 word/packet at a time.

Each packet can be lost, corrupted or duplicated…

Is it correct? **…timeliness?** **…efficient?**

First attempt

Send at most 1 word/packet at a time.

Each packet can be lost, corrupted or duplicated…

Is it correct? **…timeliness?** **…efficient?**

How can we extend this protocol by allowing the transmission of multiple words/packet at a time?

Protocol design…
we need to

define the header(s) needed to add to the packets

describe the procedure to be run on the sender and receiver

think of efficiency...

The four goals of reliable transfer

correctness	ensure data is delivered, in order, and untouched
timeliness	minimize time until data is transferred
efficiency	optimal use of bandwidth
fairness	play well with concurrent communications

How to define correctness?

A reliable transport design is correct if…

attempt #1

packets are delivered to the receiver

Wrong **Consider that the network is partitioned**

The design is not incorrect if it doesn't work in a partitionaed network

A reliable transport design is correct if…

attempt #2

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**

The design is not incorrect if it doesn't know the unknowable

A reliable transport design is correct if…

attempt #3

**a packet is resent if and only if
the previous transmission was lost or corrupted**

Wrong

1) packet delivered to the receiver and
all packets from the receiver were dropped

2) packet was dropped on the way and
all packets from the receiver were dropped

The sender has no feedback at all...
How to make a decision on what packet it should retransmit?

A reliable transport design is correct if…

attempt #3

a packet is resent if and only if
the previous transmission was lost or corrupted

~~Wrong~~

1) packet delivered to the receiver and
all packets from the receiver were dropped

2) packet was dropped on the way and

all packets from the receiver were dropped

The sender has no feedback at all...
How to make a decision on what packet it should retransmit?

A reliable transport design is correct if…

1) a packet is always resent if
the previous transmission was lost or corrupted

2) a packet may be resent at other times

A transport mechanism is correct
if and only if it resends all dropped or corrupted packets

definition

sufficient - if	**algorithm will always keep trying to deliver undelivered packets**
necessary - only if	**if it ever let a packet go undelivered without resending it, it isn't reliable**

it is ok to give up after a while but
must announce it to the application

How to ensure correctness?
And what about the tradeoffs…

Goal

design a correct, timely, efficient and fair transport mechanism

Reality

packets can be lost, corrupted, reordered, delayed, duplicated…

Timeliness vs efficiency

Timeliness argues for small timers

small timers may cause unnecessary retransmissions

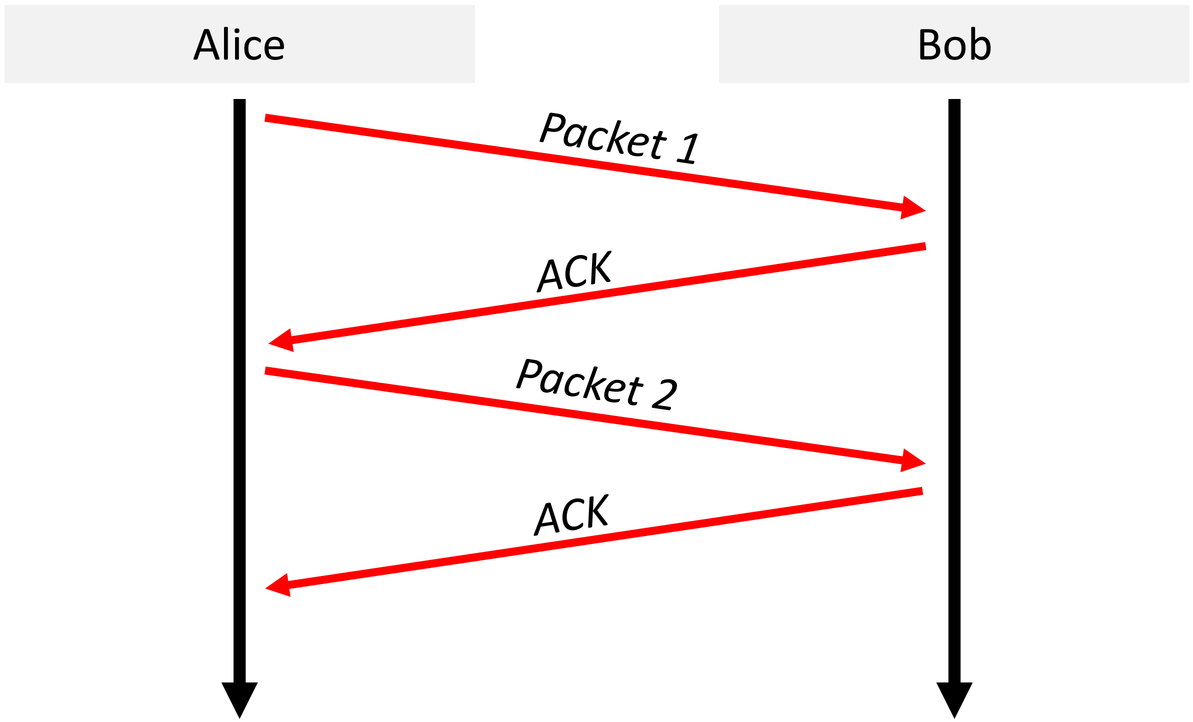
wasting resources

Efficiency for large ones

large timers may result in slow retransmission

delaying the continous data transmission e.g. in a lossy channel

Even with short timers, the timeliness of our protocol is extremely poor: one packet per Round-Trip Time (RTT)



Improvement

send multiple packets at the same time

idea

add sequence number inside each packet

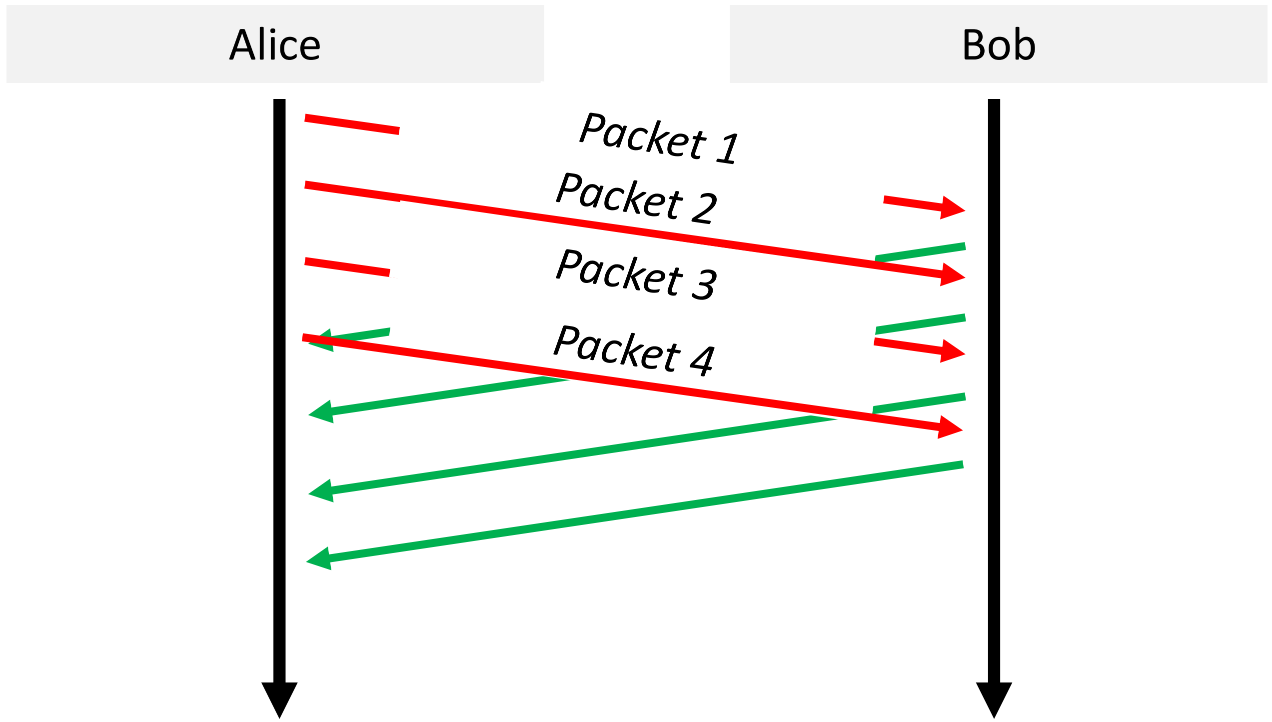
enables to distinguish individual packets

add buffers to the sender and receiver

at sender side store packets sent & not ACKed

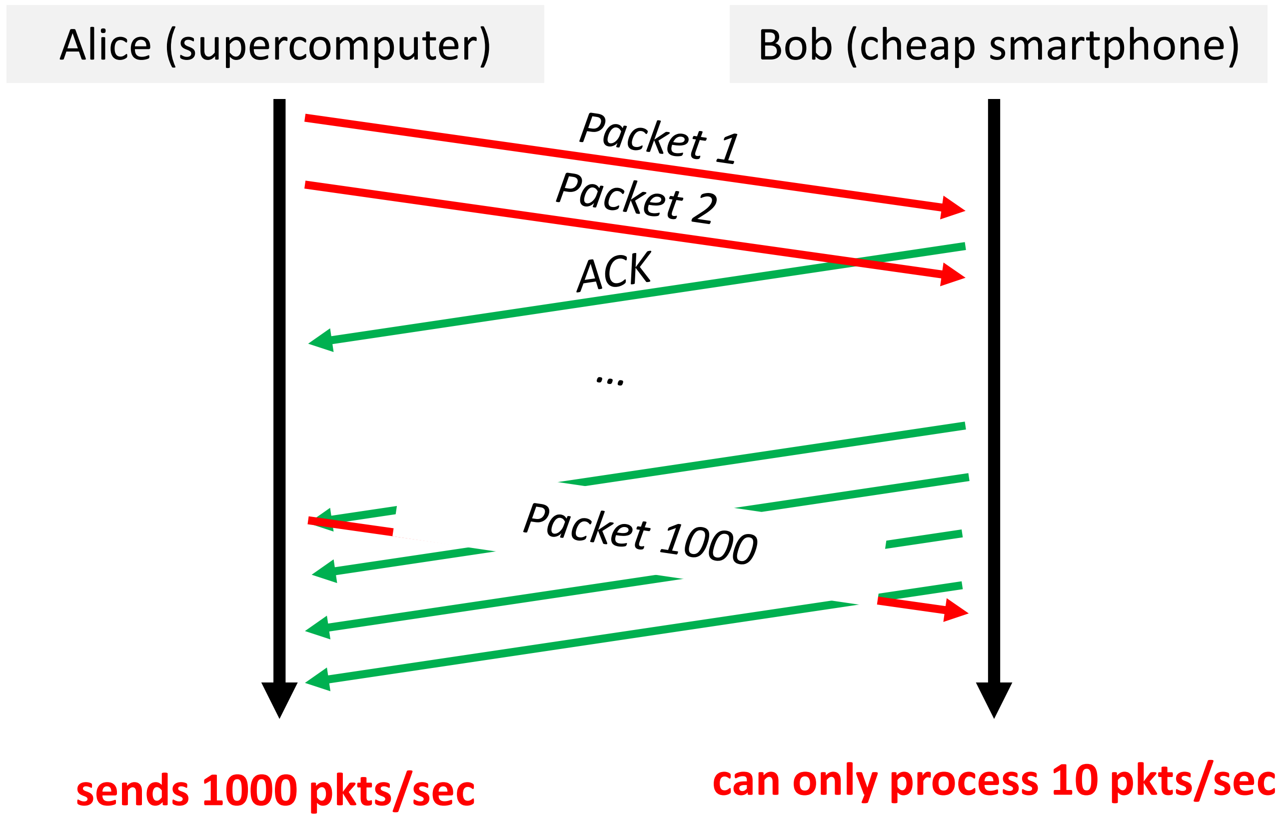
at receiver side store out-of-order packets received

Pipeline technique

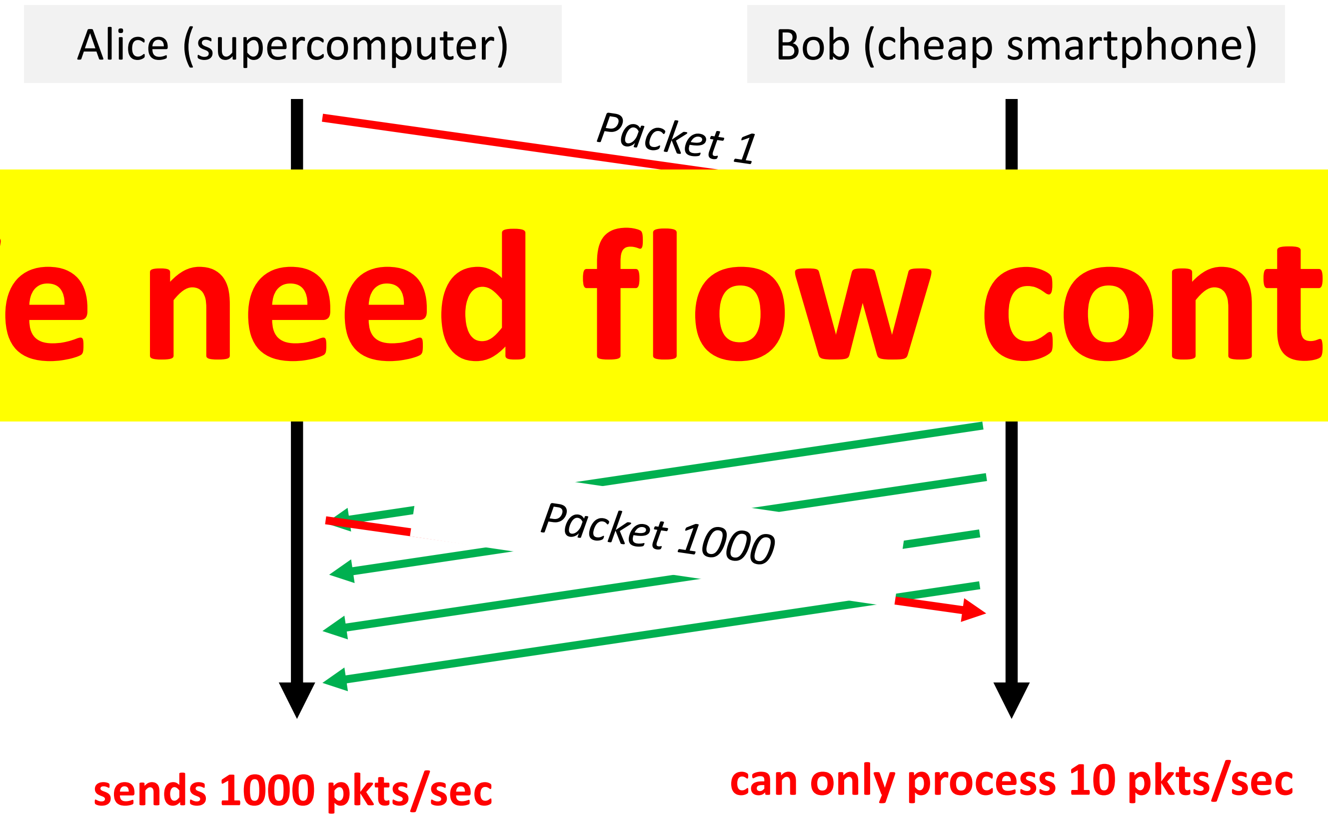


4 packets sent
w/o ACKs

Sending multiple packets improves timeliness, but it can overload a slow receiver



Sending multiple packets improves timeliness, but it can overload a slow receiver



We need flow control

A solution – sliding window

Sender keeps a list of the sequence numbers it can send

known as the maximum sending window,
and also keeps a list of sequence numbers that have already sent but not ACKed

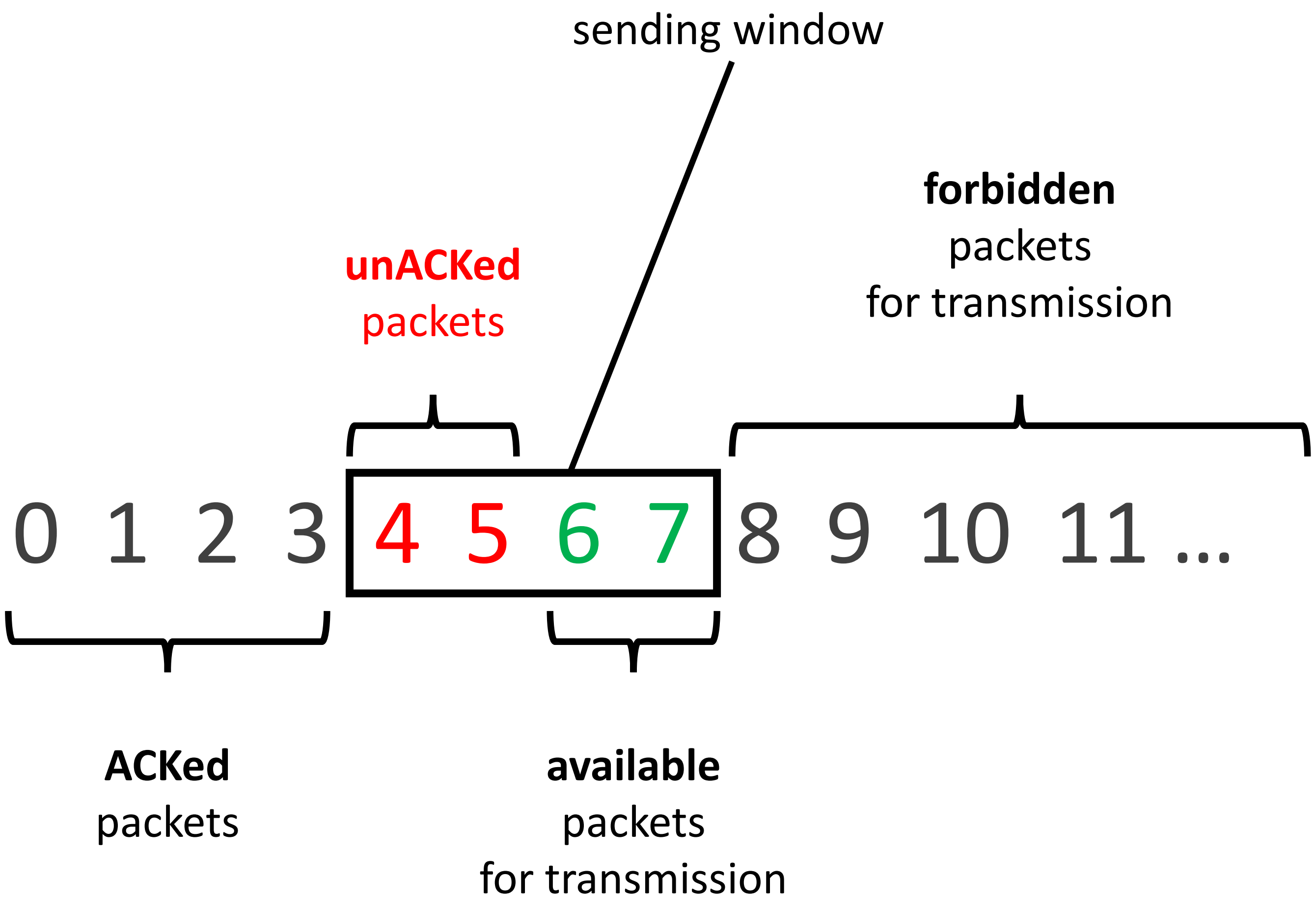
Receiver also keeps a list of the acceptable sequence numbers

known as the receiving window

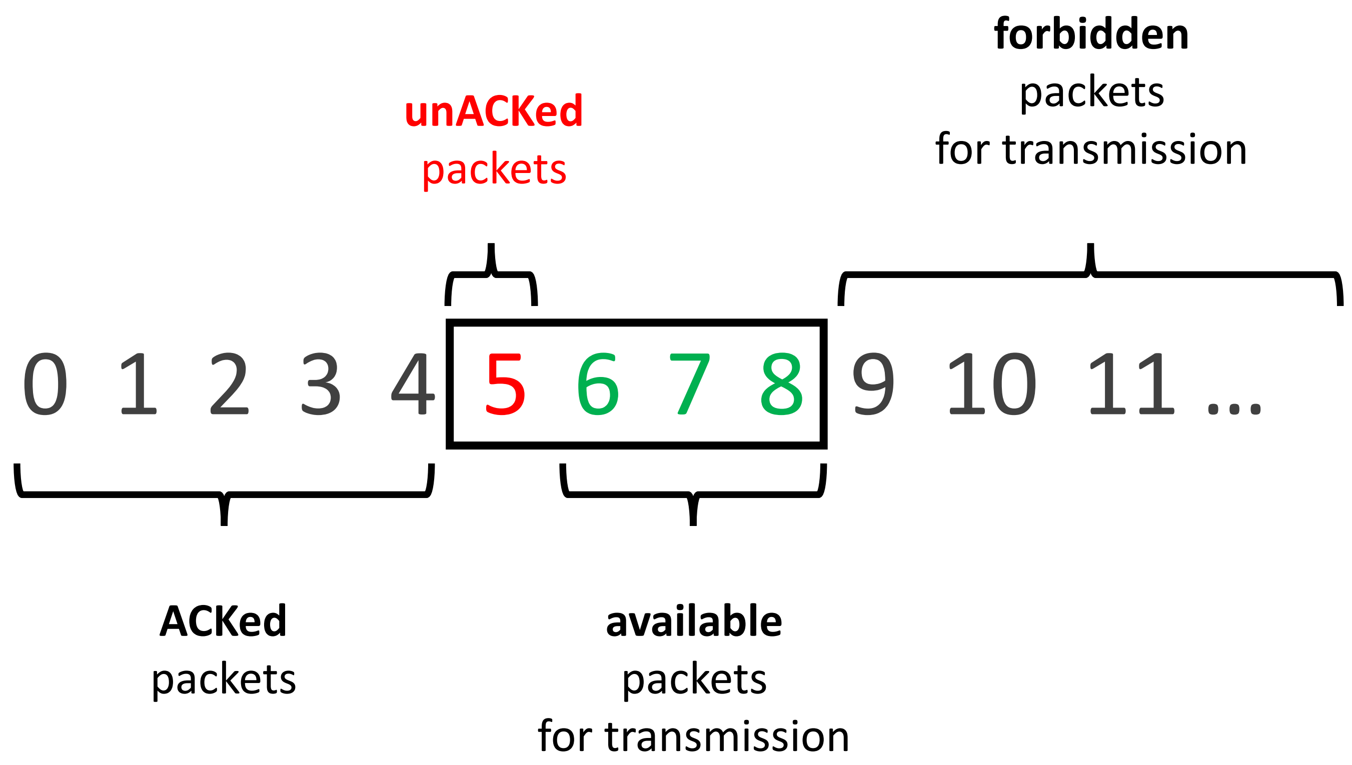
Sender and receiver negotiate the window size

sending window <= receiving window

Example with a window composed of 4 packets

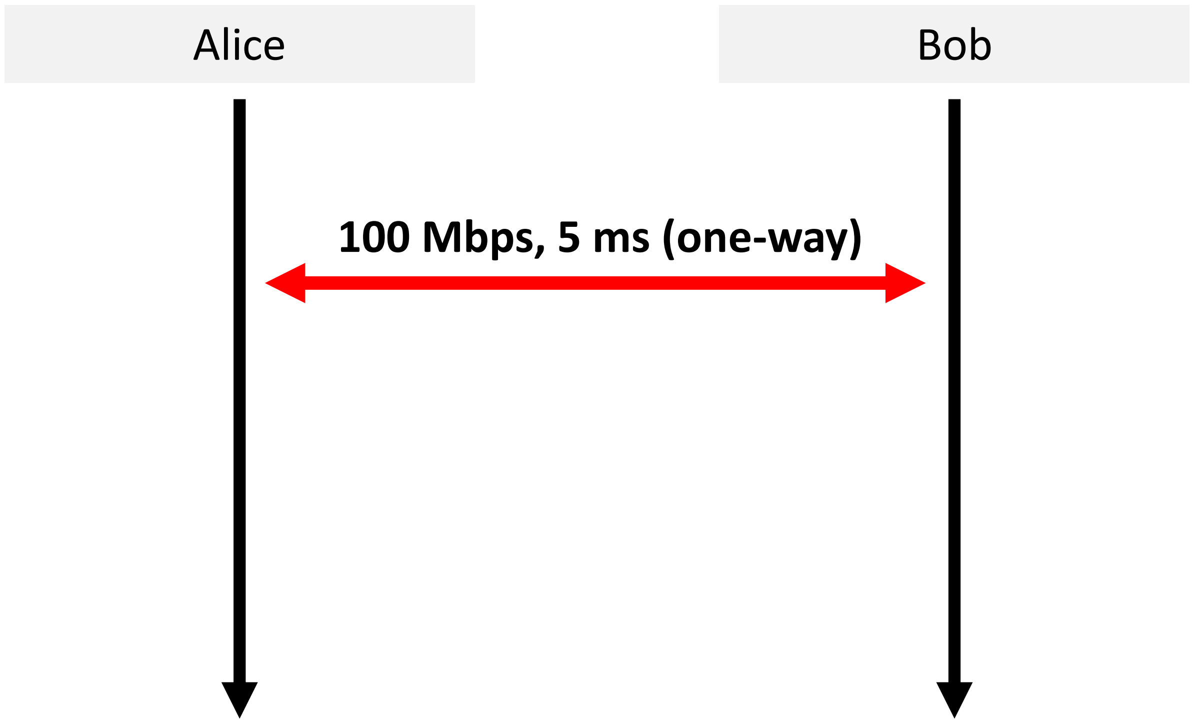


Sending window after sender receives ACK #4



Timeliness of the window protocol depends on the size of the sending window

Assuming infinite buffers, how big should the window be to maximize timeliness?



What should be the window size?
(in bytes)

The efficiency of our protocol essentially depends on two factors

receiver feedback

How much information does the sender get?

behavior upon losses

How does the sender detect and react to losses?

ACKing individual packets provides detailed feedback, but triggers unnecessary retransmission upon losses

advantages	disadvantages
information about each packet	**loss of an ACK requires retransmission**
simple window algorithm WND single-packet algorithm	**causes unnecessary retransm.**
not sensitive to reordering	

Cumulative ACKs instead of ACKing individual packets

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

Cumulative ACKs enables to recover from lost ACKs, but provides coarse-grained information to the sender

advantages

recover from lost ACKs

disadvantages

confused by reordering

incomplete information about which packets have arrived

causes unnecessary retransm.

Full Information Feedback

List all packets that have been received

Give the highest cumulative ACK,
plus any additional packets

Example:

up to 1
up to 2
up to 3
up to 4
up to 4, plus 6 ← **the „hole" is explicit**
up to 4, plus 6 and 7

Full Information Feedback prevents unnecessary retransmission, but can induce a sizable overhead

advantages	disadvantages
complete information	**overhead**
resilient form of individual ACKs	**lower efficiency**
simple loss detection	

How to detect loss?

As of now, we detect loss by using timers.

That’s only one way though

Losses can also be detected by relying on ACKs

With **individual ACKs**, missing packets (gaps) are implicit

Let’s assume packet 5 is lost, but no other

ACK stream

1

2

3

4

6

7

…

Sender can see that 5 is missing
and resend 5 after k subsequent packets

With **full information**, missing packets (gaps) are implicit

Let’s assume packet 5 is lost, but no other

ACK stream

up to 1

up to 2

up to 3

up to 4

up to 4, plus 6 → **Sender can see that 5 is missing** and resend 5 after k subsequent packets

up to 4, plus 6 and 7

…

With **cumulative ACKs**, missing packets are harder to know

Let's assume packet 5 is lost, but no other

ACK stream

up to 1

up to 2

up to 3

up to 4

up to 4 (to pkt 6) → **Duplicates indicate isolated losses**

up to 4 (to pkt 7)

…

Duplicated ACKs are a sign of isolated losses. Dealing with them is trickier though.

Lack of ACK progress means that 5 hasn't made it

Stream of ACKs means that (some) packets are delivered

Sender could trigger resend upon receiving k duplicates ACKs

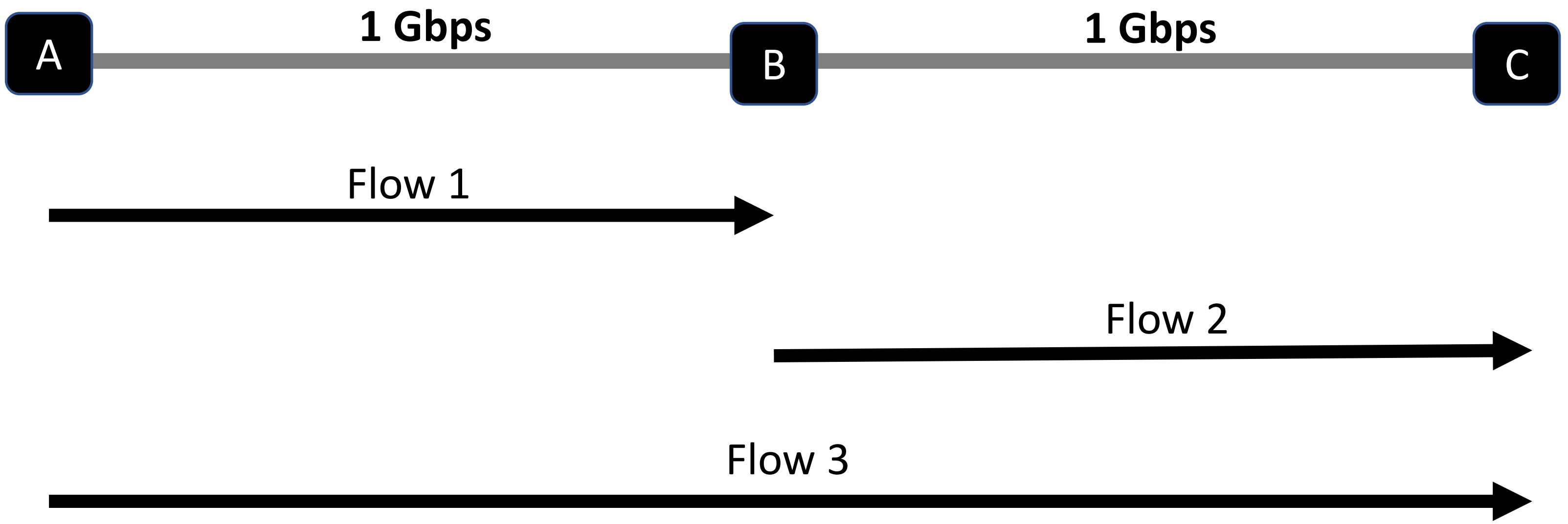
but which packet should be resent?

only 5 or 5 and everything after?

What about fairness?

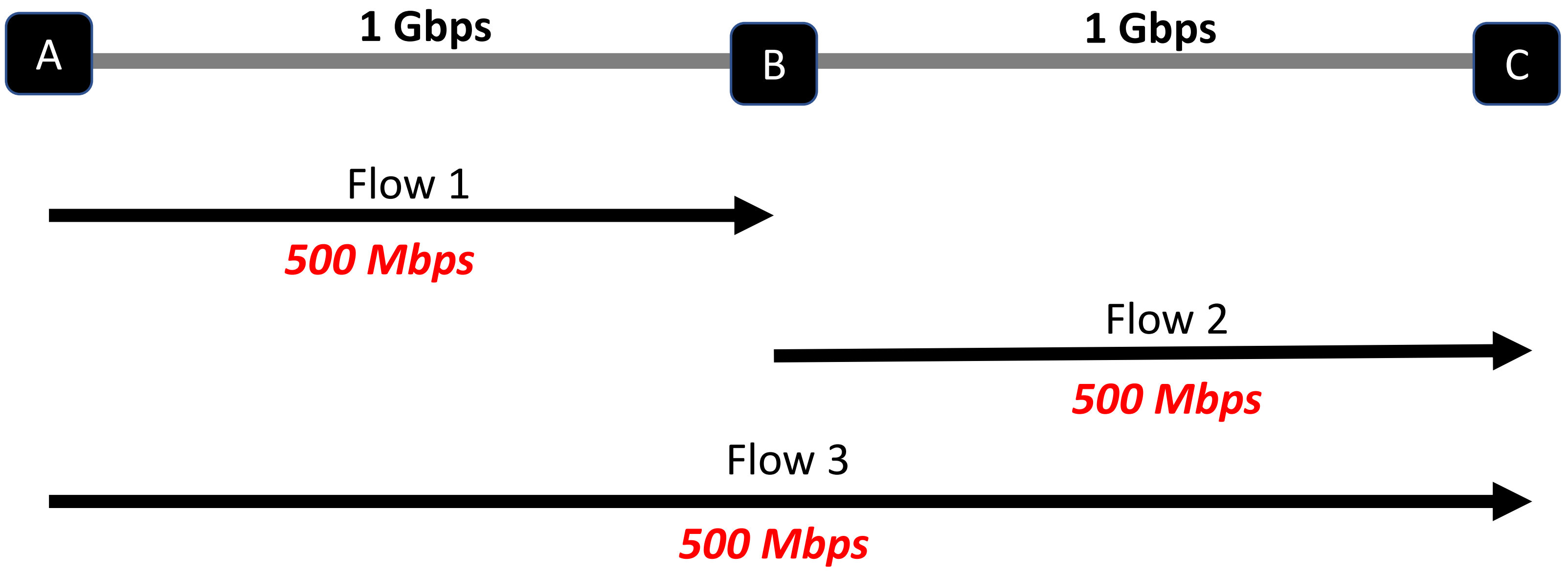
When n entities are using our transport mechanism, we want a fair allocation of the available bandwidth

Consider this simple network in which three hosts are sharing two links



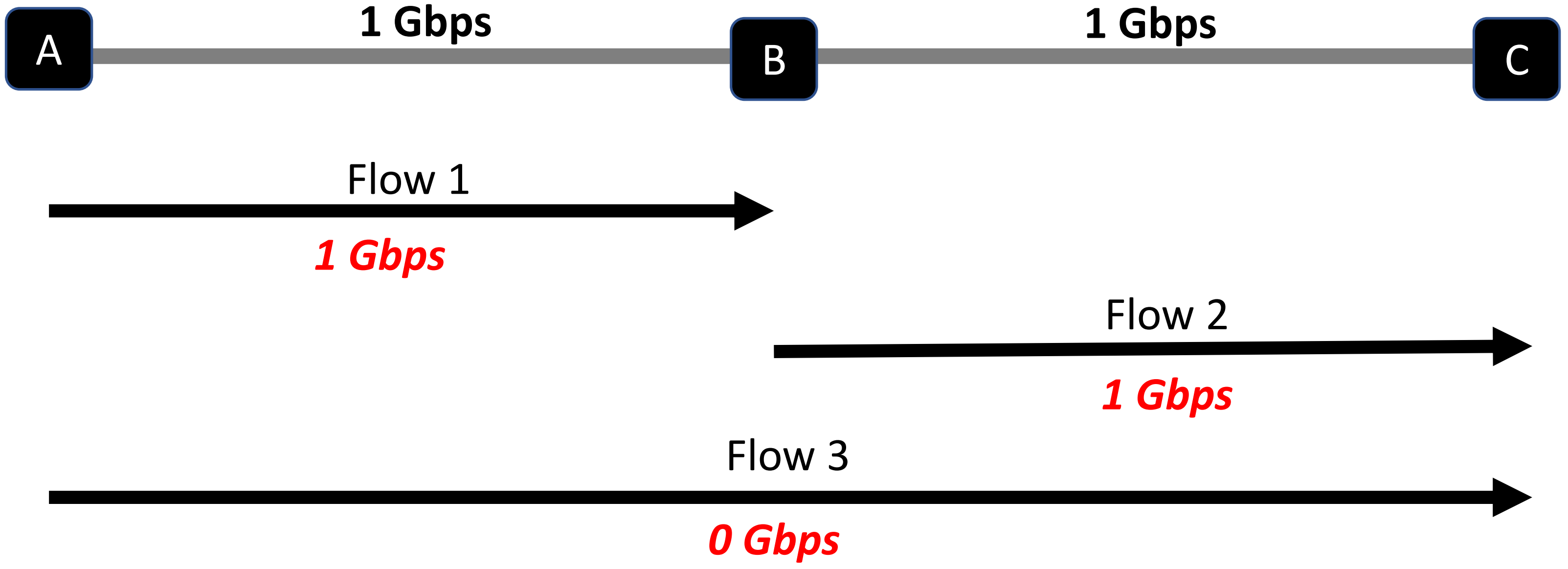
What is a fair allocation for the 3 flows?

An equal allocation is certainly “fair”, but what about the efficiency of the network?



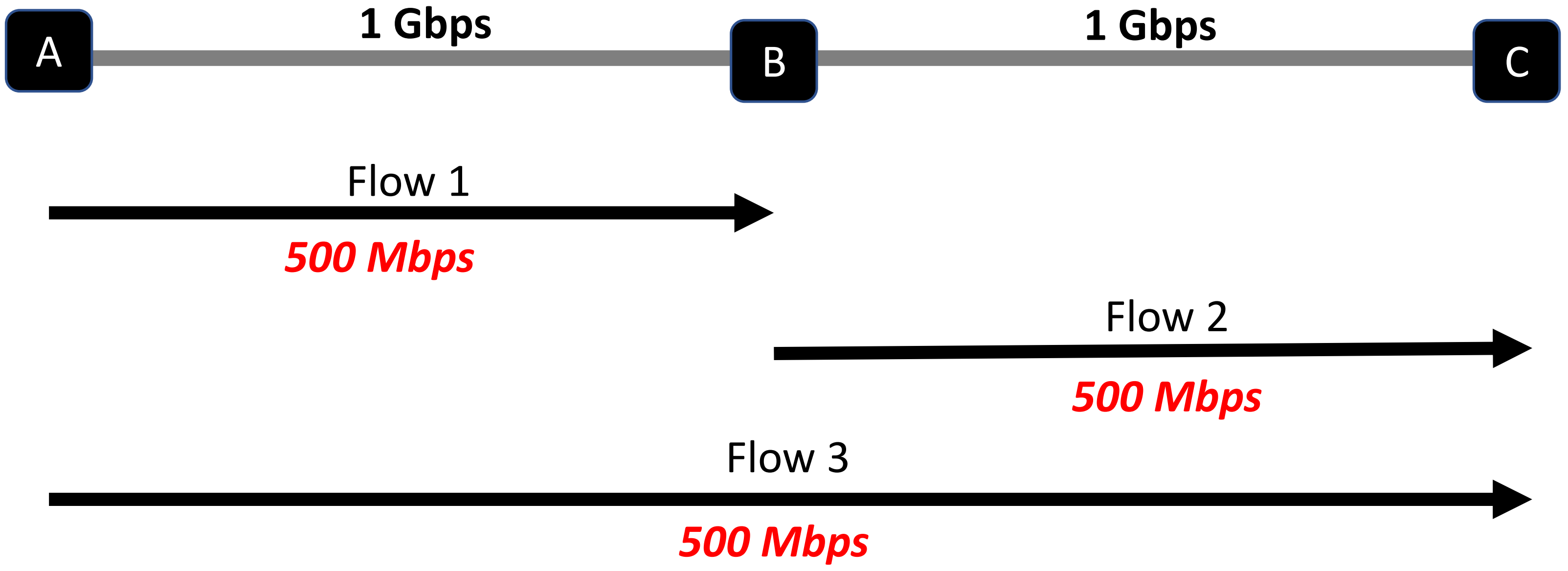
Total traffic is 1.5 Gbps

Fairness and efficiency don't always play along, here an unfair allocation ends up more efficient



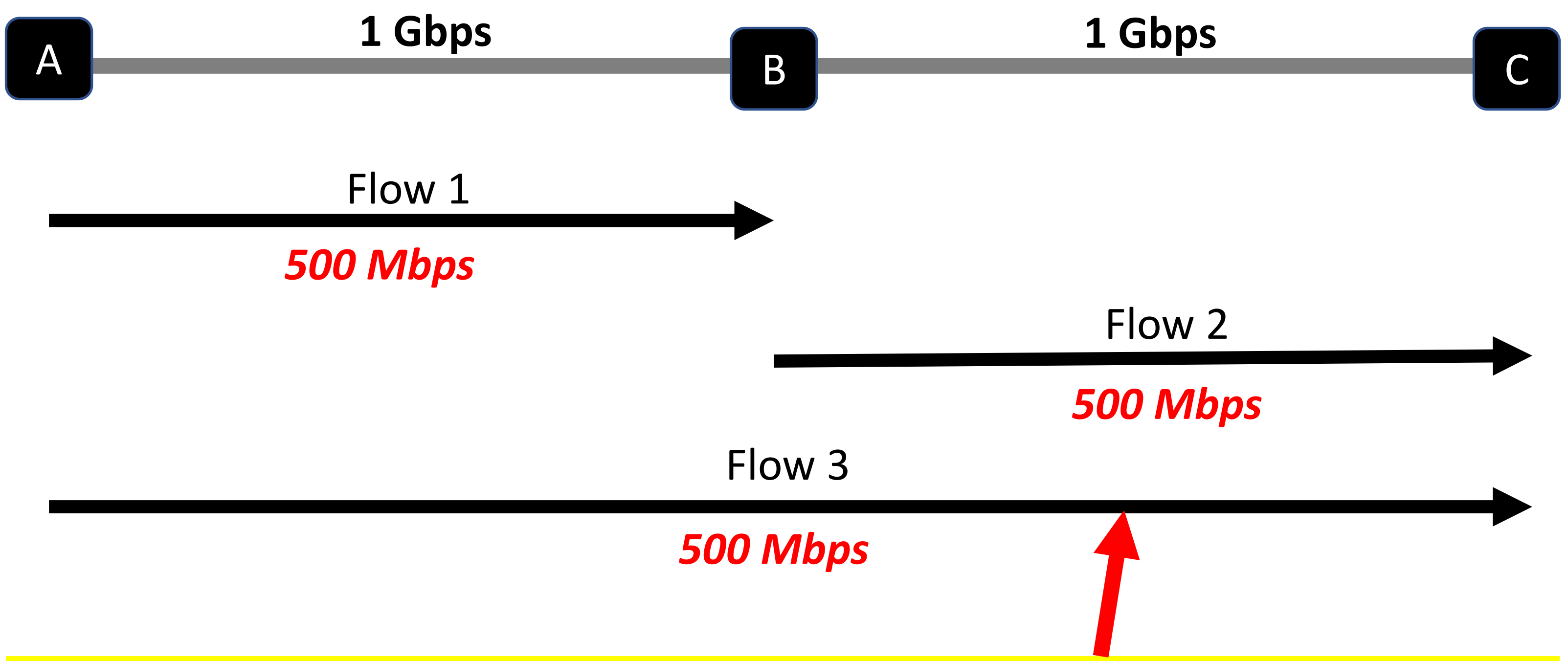
Total traffic is 2.0 Gbps

What is fair anyway?



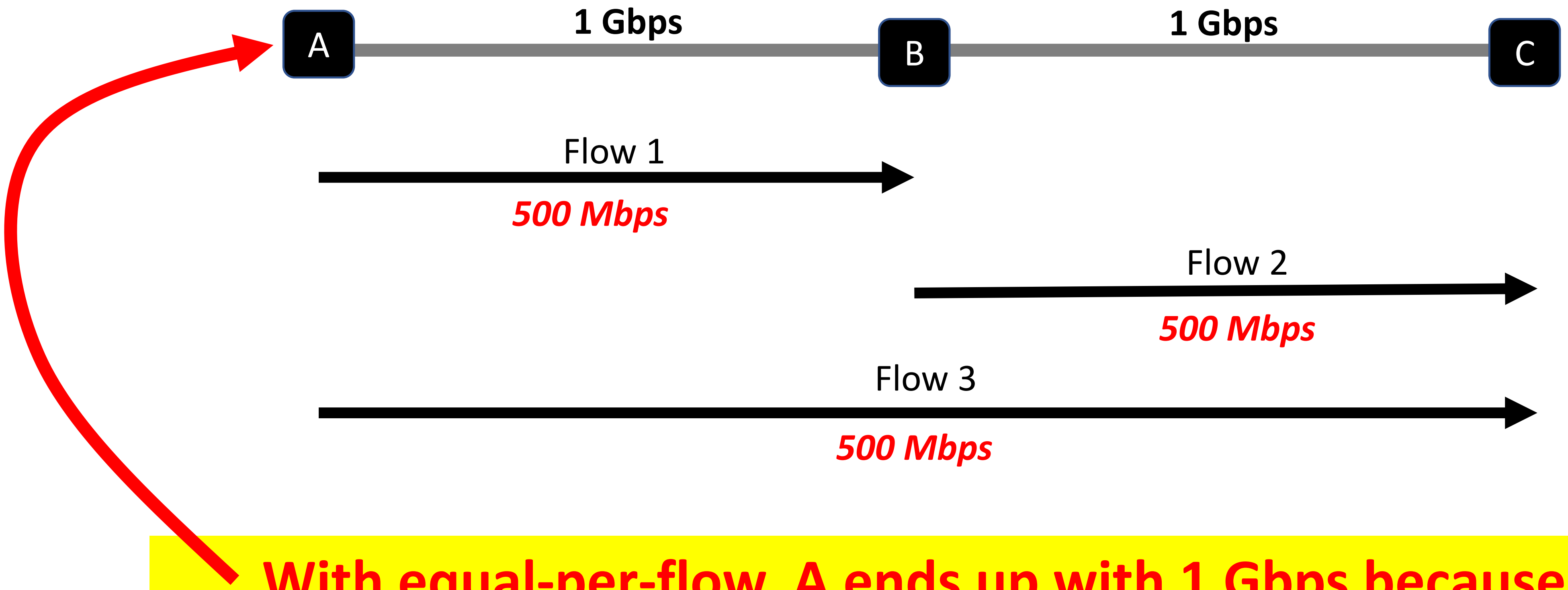
Total traffic is 1.5 Gbps

What is fair anyway?



Equal-per-flow isn't really fair as (A,C) crosses two links: it uses *more* resources

What is fair anyway?



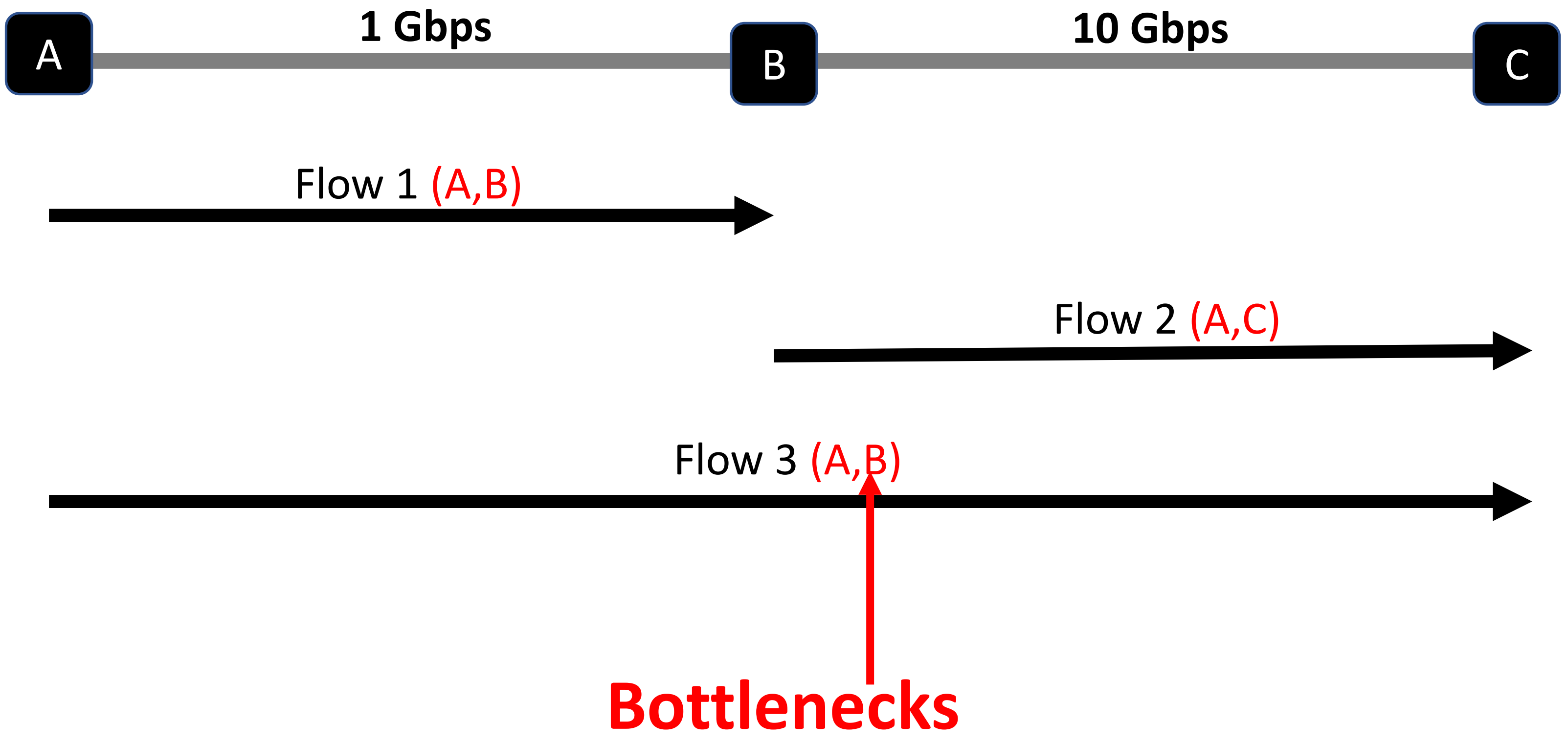
With equal-per-flow, A ends up with 1 Gbps because it sends 2 flows, while B ends up with 500 Mbps

Is it fair???

Seeking an exact notion of fairness is not productive.
What matters is to avoid starvation.

equal-per-flow is good enough for this

Simply dividing the available bandwidth doesn't work in practice since flows can see different bottleneck



Intuitively, we want to give users with "small" demands what they want, and evenly distribute the rest

Max-min fair allocation is such that

the lowest demand is maximized

after the lowest demand has been satisfied,
the second lowest demand is maximized

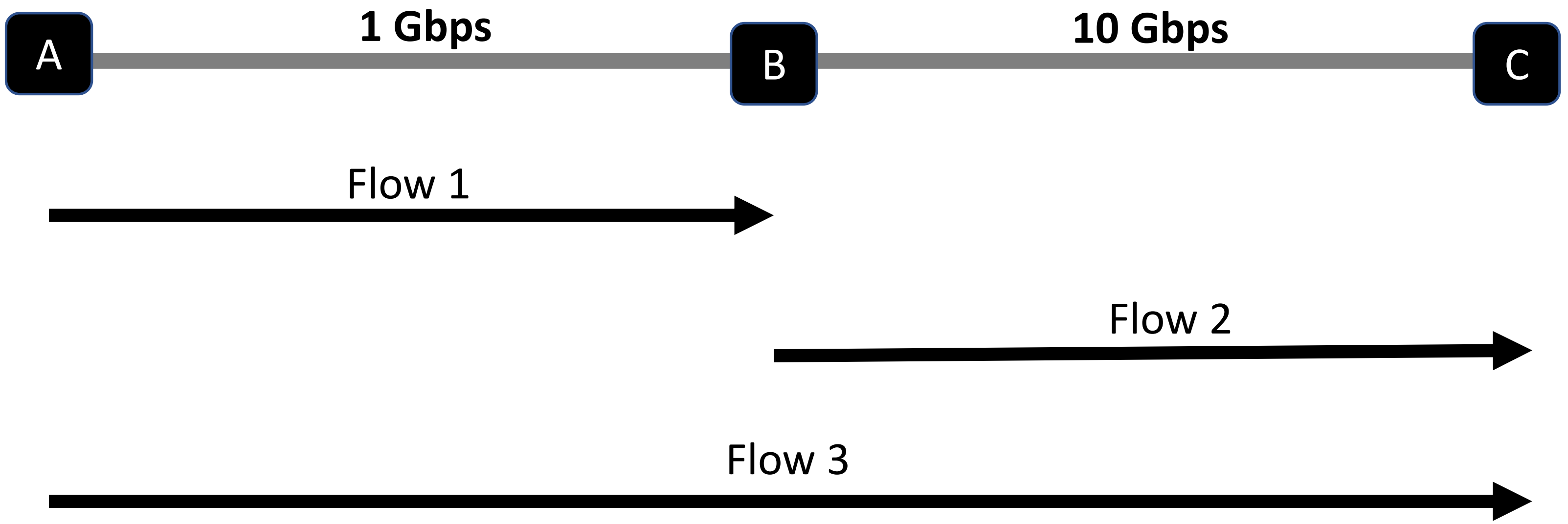
after the second lowest demand has been satisfied,
the third lowest demand is maximized

and so on…

Max-min fair allocation can easily be computed

step 1	**Start with all flows at rate 0**
step 2	**Increase the 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**

Example



What's the max-min fair allocation?

Max-min fair allocation can be approximated by slowly increasing WND until a loss is detected

Intuition

Progressively increase the sending window size — *max=recv. window*

Whenever a loss is detected, decrease the window size — *signal of congestion*

Repeat

Dealing with corruption is easy

Rely on a checksum,

treat corrupted packets as lost

The effect of reordering depends on the type of ACKing mechanism used

solution	
individual ACKs	no problem
full feedback	no problem
cumm. ACKs	create duplicate ACKs why is it a problem?

Long delays can create useless timeouts, for all designs

Packets duplicates can lead to duplicate ACKs whose effects will depend on the ACKing mechanism used

solution	
individual ACKs	no problem
full feedback	no problem
cumm. ACKs	problematic

Here is one correct, timely, efficient and fair transport mechanism

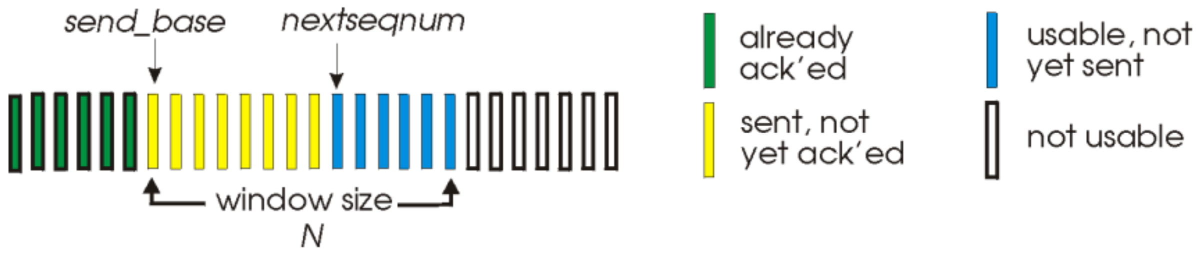
ACKing	**full information ACK**
retransmission	**after timeout** **after k subsequent ACKs**
window management	**additive increase upon successful delivery** **multiple decrease when timeouts**

More details later when we see TCP

Examples

Go-back-N and Selective Repeat

Go-Back-N (GBN)



a simple sliding window protocol using cumulative ACKs

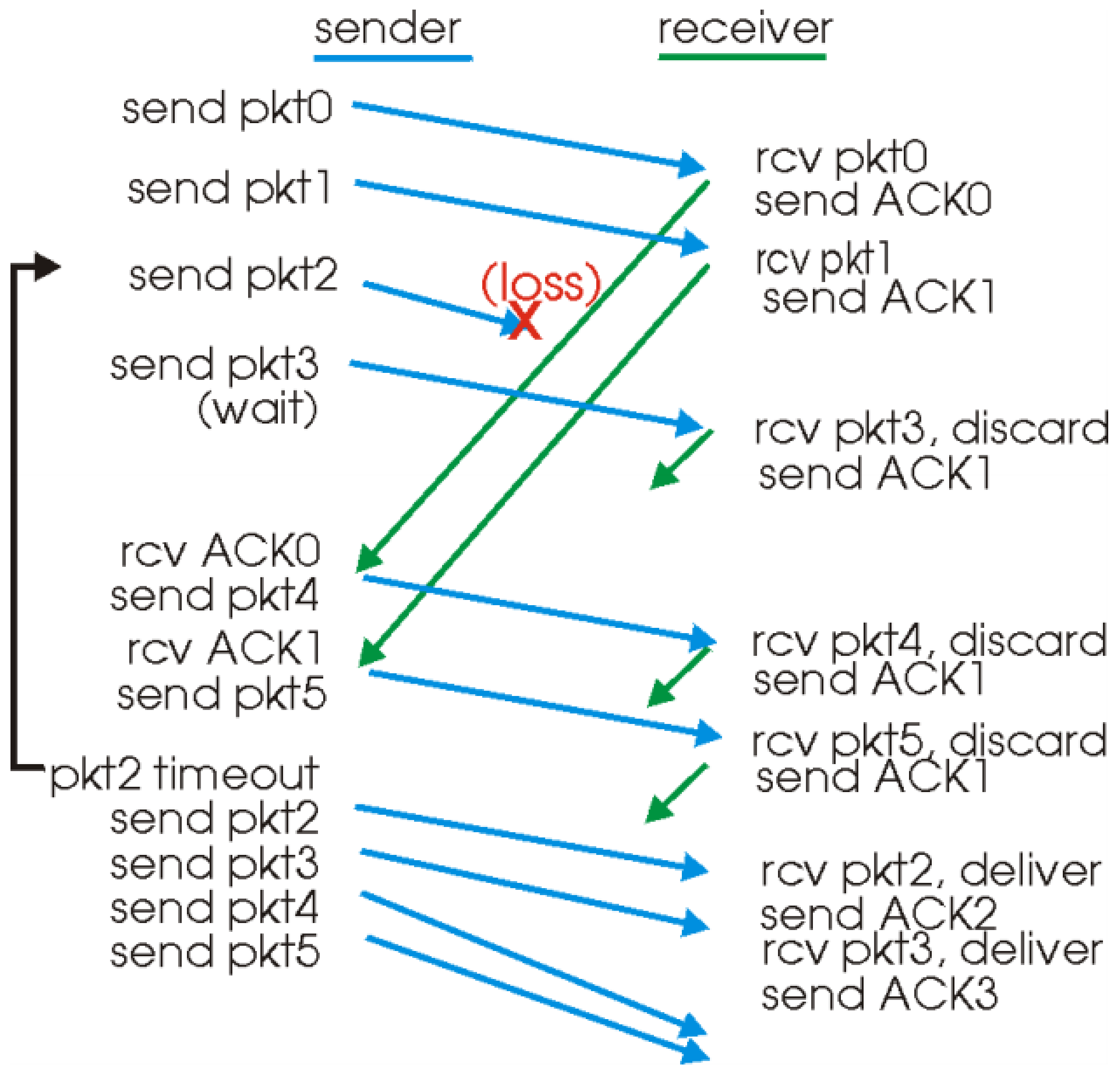
goal **receiver should be as simple as possible**

receiver **delivers packets in-order to the upper layer**
receiver wnd size is 1

sender **use a single timer to detect loss, reset at each new ACK**

upon timeout, resend all WND packets
starting with the lost one

GBN in action

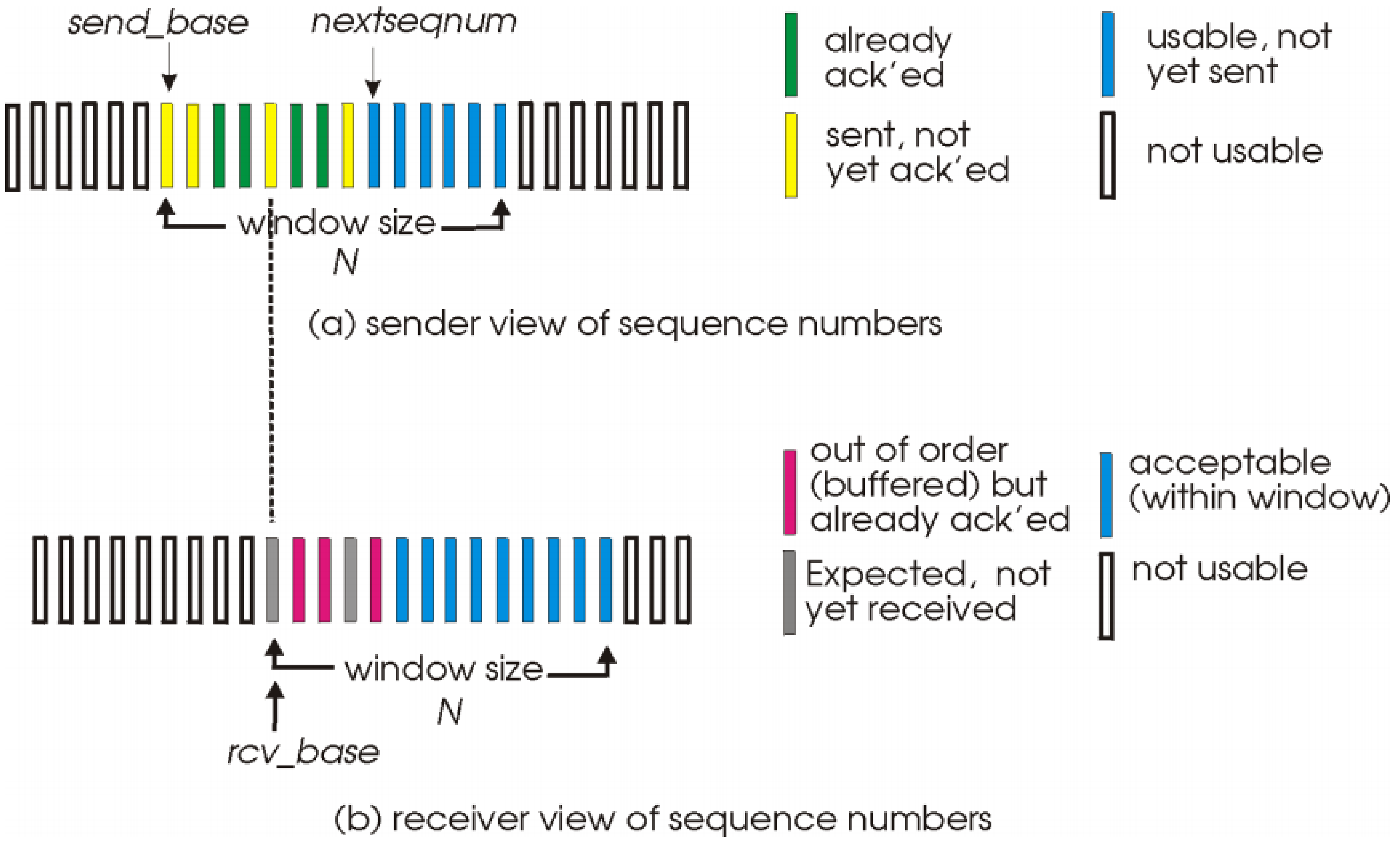


Selective Repeat (SR)

avoid unnecessary retransmissions by <u>using per-packet ACKs</u>

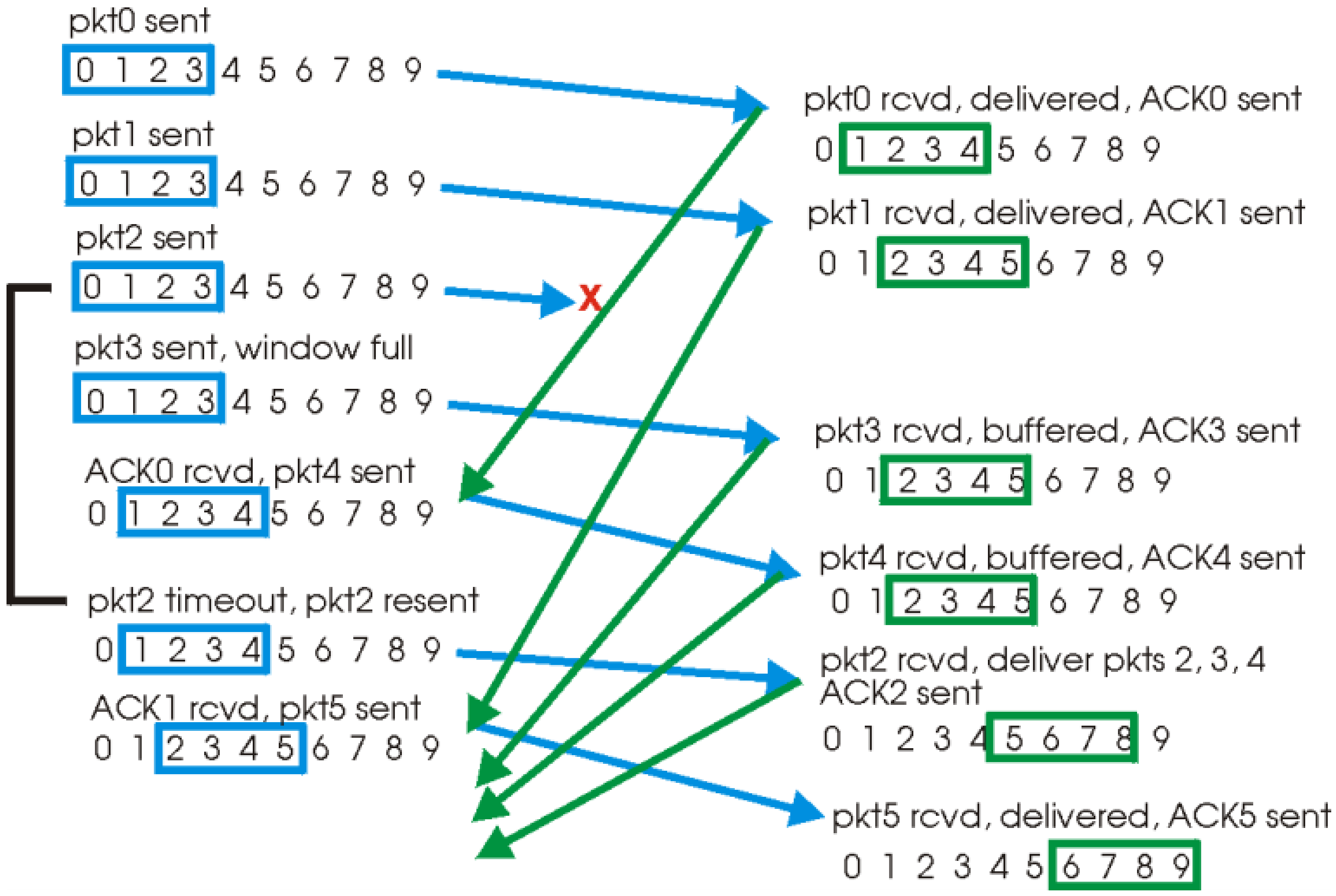
goal	**avoids unnecessary retransmissions**
receiver	**ACK each packet, in-order or not** **buffer out-of-order packets**
sender	**use per-packet timer to detect loss**
	upon loss, only the lost packet

SR - windows



send_base
nextseqnum
window size
N
(a) sender view of sequence numbers
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable
window size
N
rcv_base
(b) receiver view of sequence numbers
out of order (buffered) but already ack'ed
Expected, not yet received
acceptable (within window)
not usable

SR in action



pkt0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 sent
0 1 2 3 4 5 6 7 8 9
pkt2 sent
0 1 2 3 4 5 6 7 8 9
X
pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9
ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9
pkt2 timeout, pkt2 resent
0 1 2 3 4 5 6 7 8 9
ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9
pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9
pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9
pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9
pkt2 rcvd, deliver pkts 2, 3, 4
ACK2 sent
0 1 2 3 4 5 6 7 8 9
pkt5 rcvd, delivered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

To be continued…