How do we achieve correctness and with what tradeoffs?

Design a *correct, timely, efficient* and *fair* transport mechanism

knowing that

packets can get lost → let's focus on these aspects first

corrupted

reordered

delayed

duplicated

How do we achieve correctness and with what tradeoffs?

Alice

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

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

Bob

```
receive_packet(p);
if check(p.payload) == p.checksum:
    send_ack();

    if word not delivered:
        deliver_word(word);
else:
    pass;
```

There is a clear tradeoff between timeliness and efficiency in the selection of the timeout value

Alice

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

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

Bob

```
receive_packet(p);
if check(p.payload) == p.checksum:
    send_ack();

    if word not delivered:
        deliver_word(word);
else:
    pass;
```

There is a clear tradeoff between timeliness and efficiency in the selection of the timeout value

Alice

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

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

Bob

```
receive_packet(p);
if check(p.payload) == p.checksum:
    send_ack();

    if word not delivered:
        deliver_word(word);
else:
    pass;
```

This algorithm is known as “stop and wait”

Stop and Wait demo

https://www2.tkn.tu-berlin.de/teaching/rn/animations/gbn_sr/

(for stop and wait, choose go back N and set the window size to 1)

Timeliness argues for small timers, efficiency for large timers

timeliness

small
timers

risk

unnecessary retransmissions

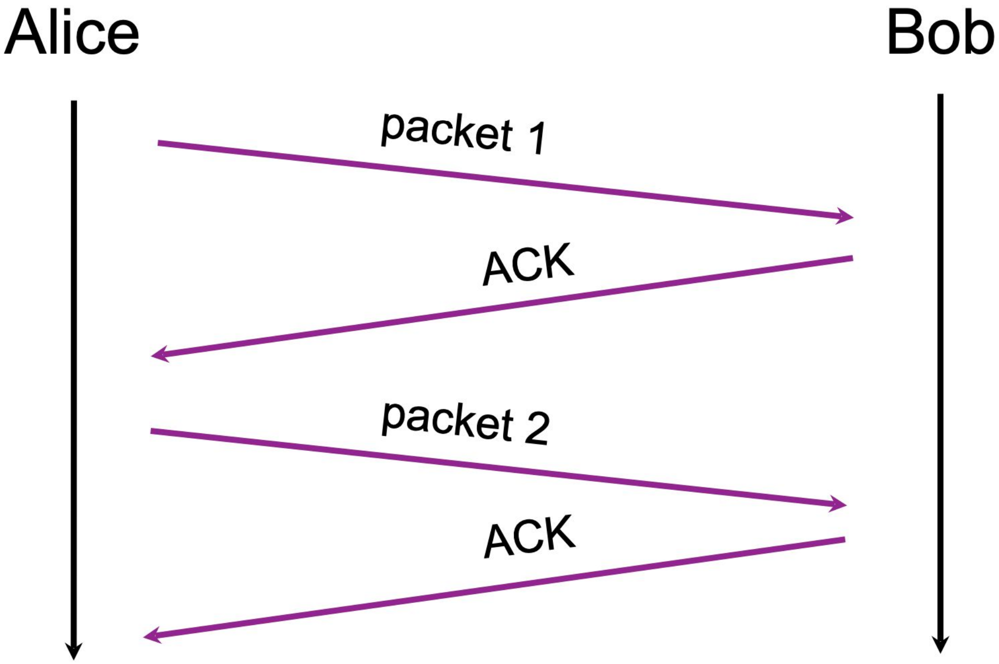
efficiency

large
timers

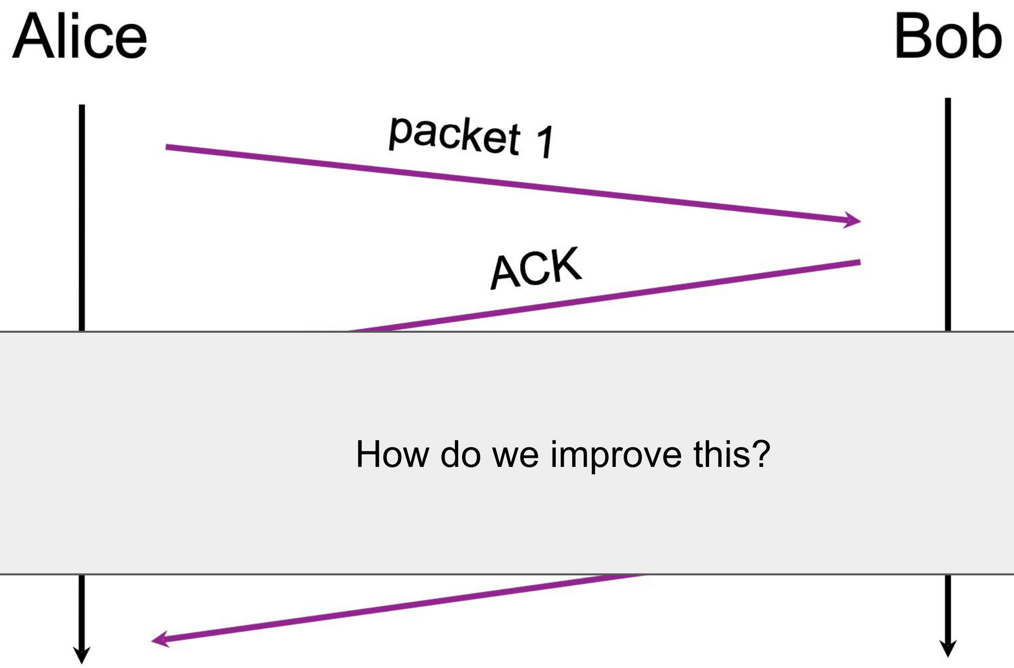
risk

slow transmission

Even with small timers, stop and wait has terrible timeliness - one packet per round trip time (RTT)



Even with small timers, stop and wait has terrible timeliness - one packet per round trip time (RTT)

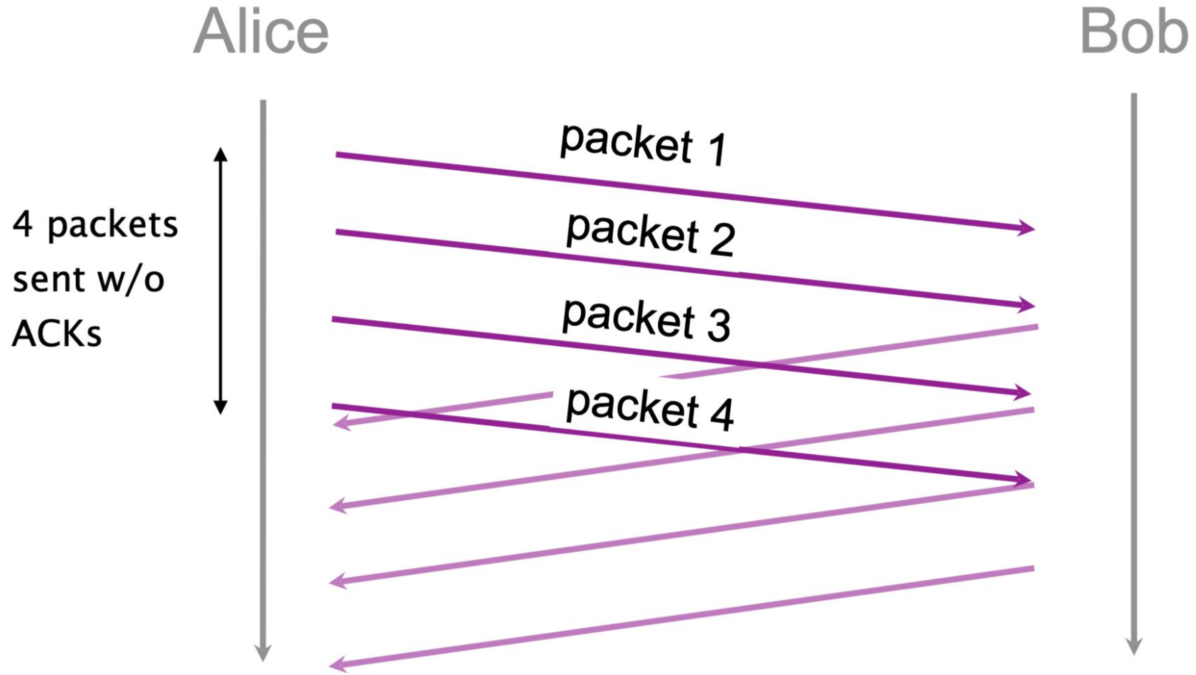


How do we improve this?

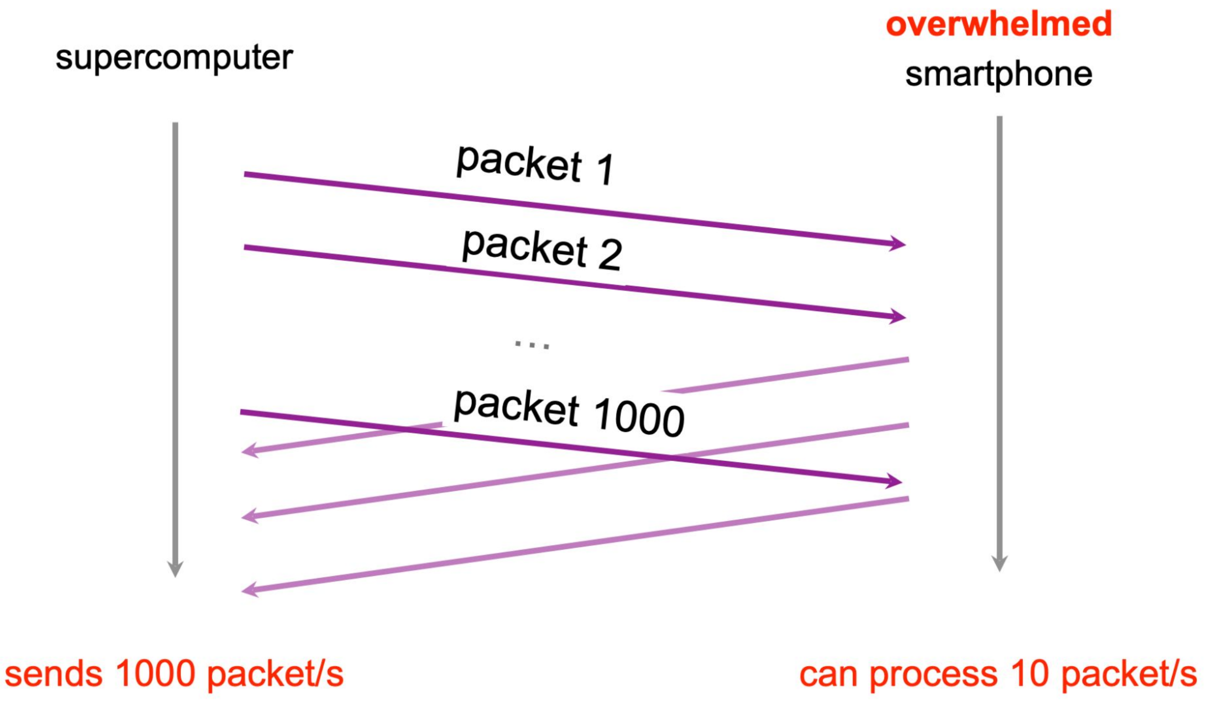
An obvious solution to improve timeliness is to send multiple packets at the same time

approach	add sequence number inside each packet	
	add buffers to the sender and receiver	
	sender	store packets sent & not acknowledged
	receiver	store out-of-sequence packets received

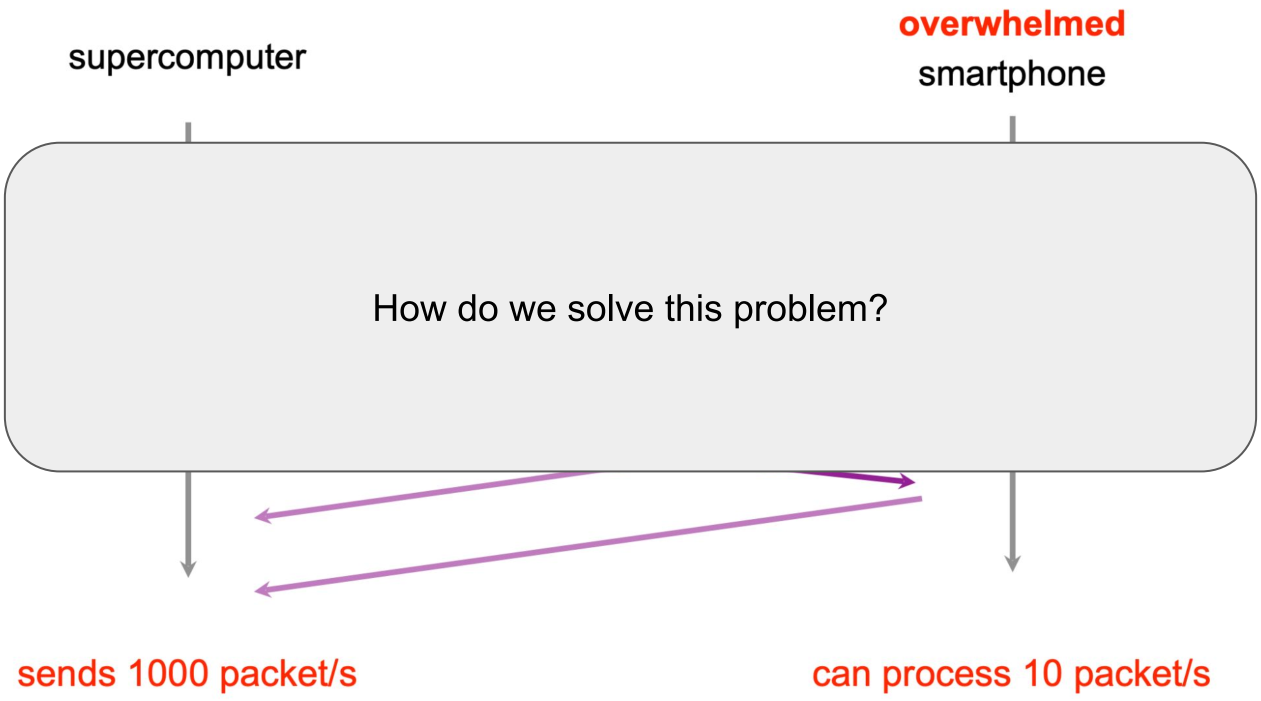
An obvious solution to improve timeliness is to send multiple packets at the same time



Sending multiple packets improves timeliness, but it can also overwhelm the receiver



Sending multiple packets improves timeliness, but it can also overwhelm the receiver



Flow control - sliding window

Sender keeps a list of the sequence # it can send

known as the *sending window*

Receiver also keeps a list of the acceptable sequence #

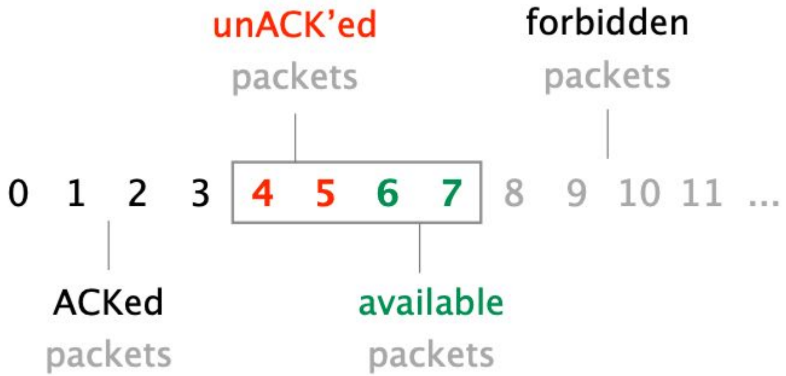
known as the *receiving window*

Sender and receiver negotiate the window size

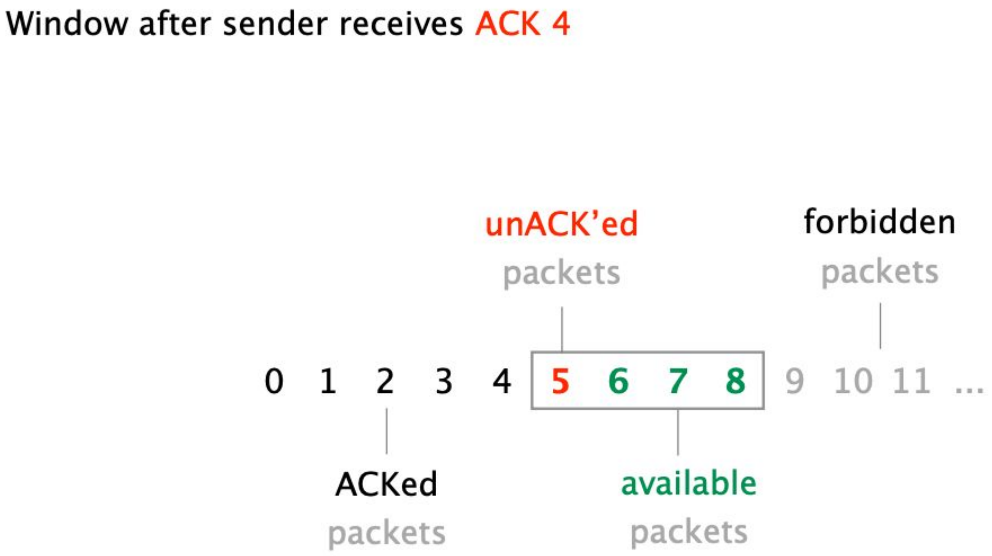
sending window <= receiving window

Flow control - sliding window

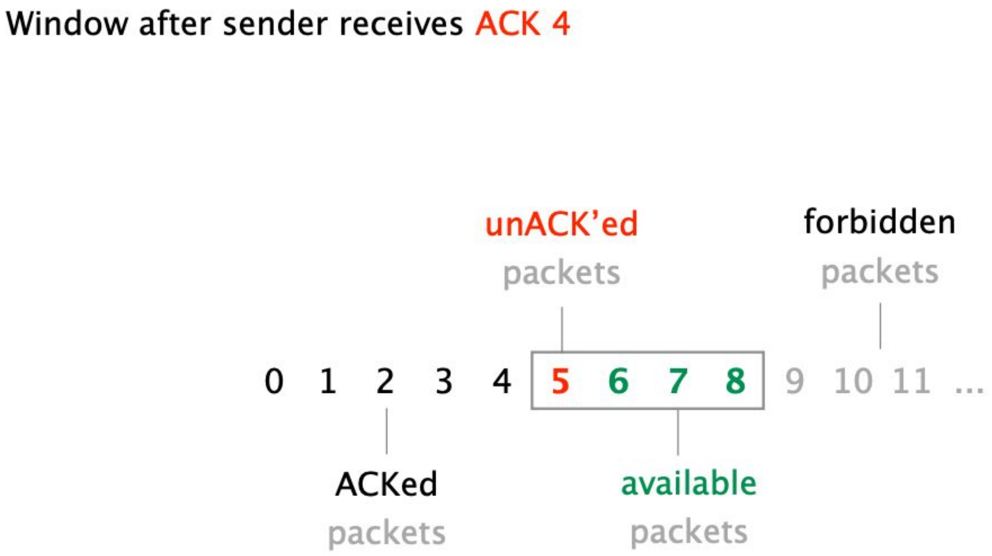
Example with a window composed of 4 packets



Flow control - sliding window



Flow control - sliding window



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

Efficiency of the protocol 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?

Efficiency of the protocol 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
know fate of each packet	loss of an ACK packet requires a retransmission
simple window algorithm W single-packet algorithms	causes unnecessary retransmission
not sensitive to reordering	

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

approach	ACK the highest sequence number for which all the previous packets have been received
advantages	recover from lost ACKs
disadvantages	confused by reordering incomplete information about which packets have arrived causes unnecessary retransmission

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

approach	List all packets that have been received highest cumulative ACK, plus any additional packets
advantages	complete information resilient form of individual ACKs
disadvantages	overhead (hence lowering efficiency) *e.g.*, when large gaps between received packets

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

approach — List all packets that have been received

highest cumulative ACK, plus any additional packets

Once again, Internet design is all about balancing tradeoffs.

adva

disadvantages — overhead (hence lowering efficiency)

e.g., when large gaps between received packets

Efficiency of the protocol 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?

We’ve been talking about detecting loss using timeouts. That’s not the only way

ACKS

With individual ACKs, missing packets (gaps) are implicit

Assume packet 5 is lost

but no other

ACK stream

1

2

3

4

6

7

...

sender can infer that 5 is missing

and resend 5 after *k* subsequent packets

With full information, missing packets (gaps) are explicit

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 learns that 5 is missing

up to 4, plus 6—7 retransmits after *k* packets

...

With cumulative ACKs, missing packets are harder to know

Assume packet 5 is lost

but no other

ACK stream

1

2

3

4

4 sent when 6 arrives

4 sent when 7 arrives

...

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

situation

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 *what* do you resend?

only 5 or 5 and everything after?

What about fairness?

Design a *correct*, *timely*, *efficient* and *fair* transport mechanism

knowing that

packets can get lost

corrupted

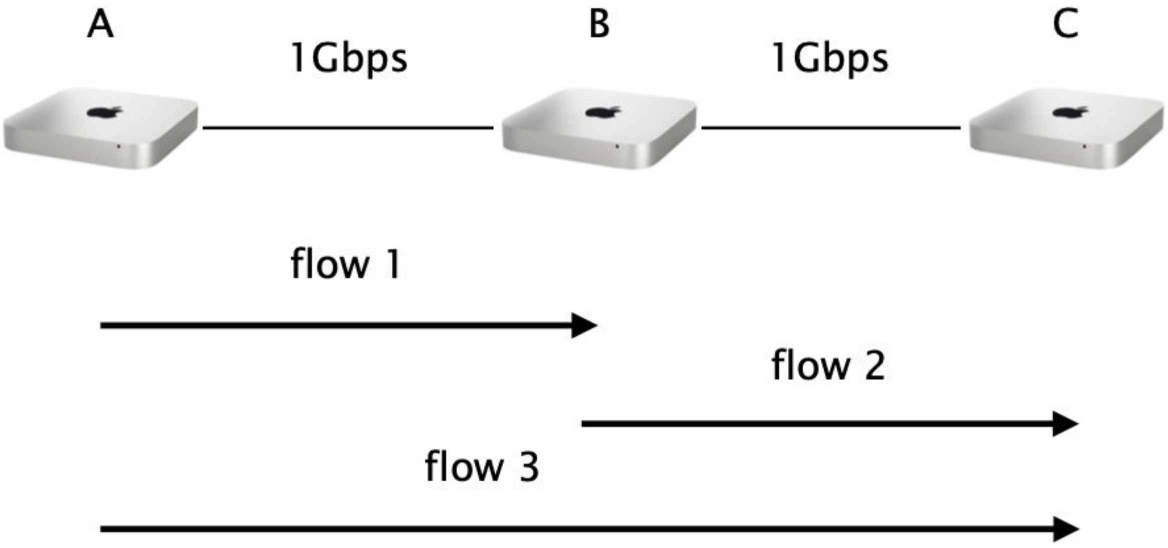
reordered

delayed

duplicated

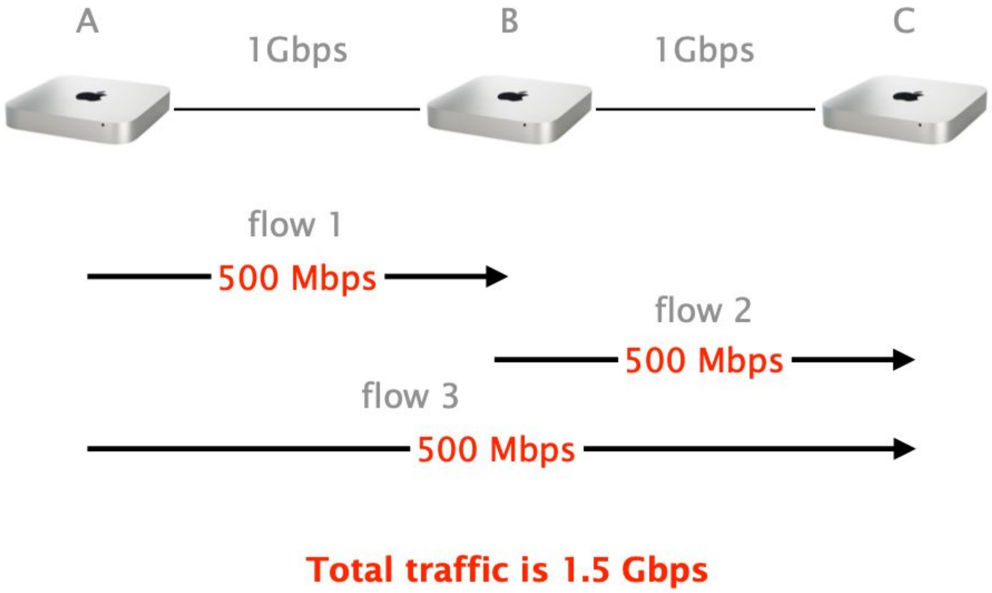
When *n* entities are using our transport mechanism, we want a fair allocation of the available bandwidth

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

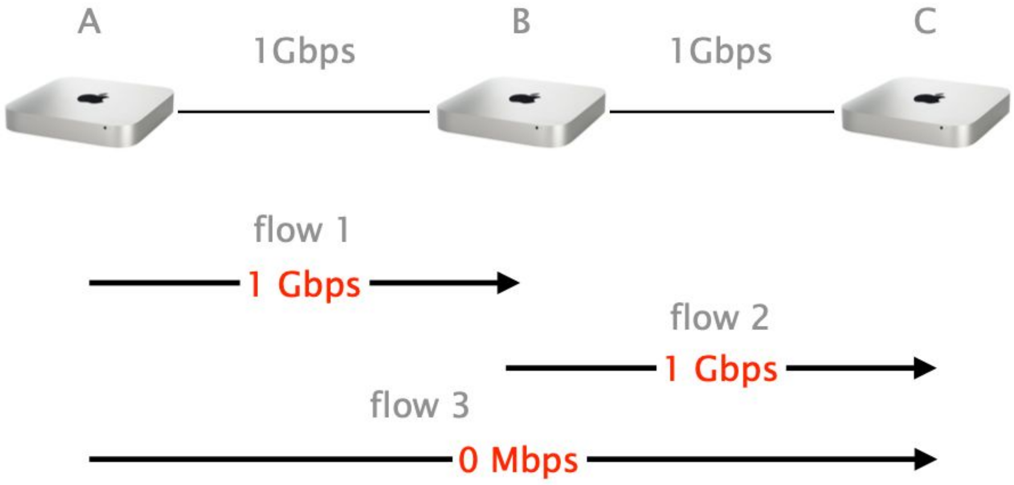


What is a fair allocation for the 3 flows?

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

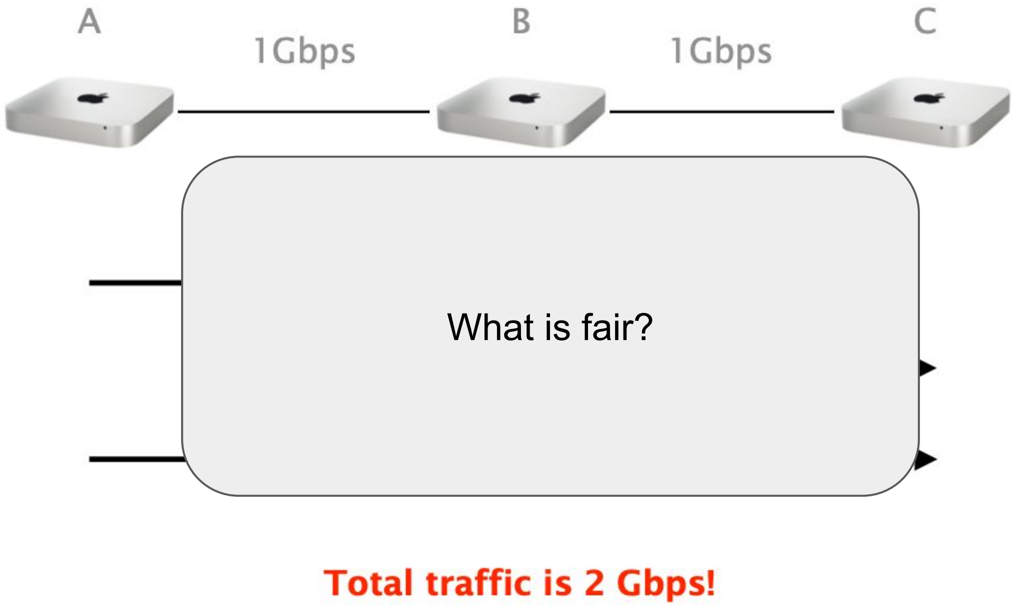


Fairness and efficiency don't always play along, here an unfair allocation ends up more *efficient*

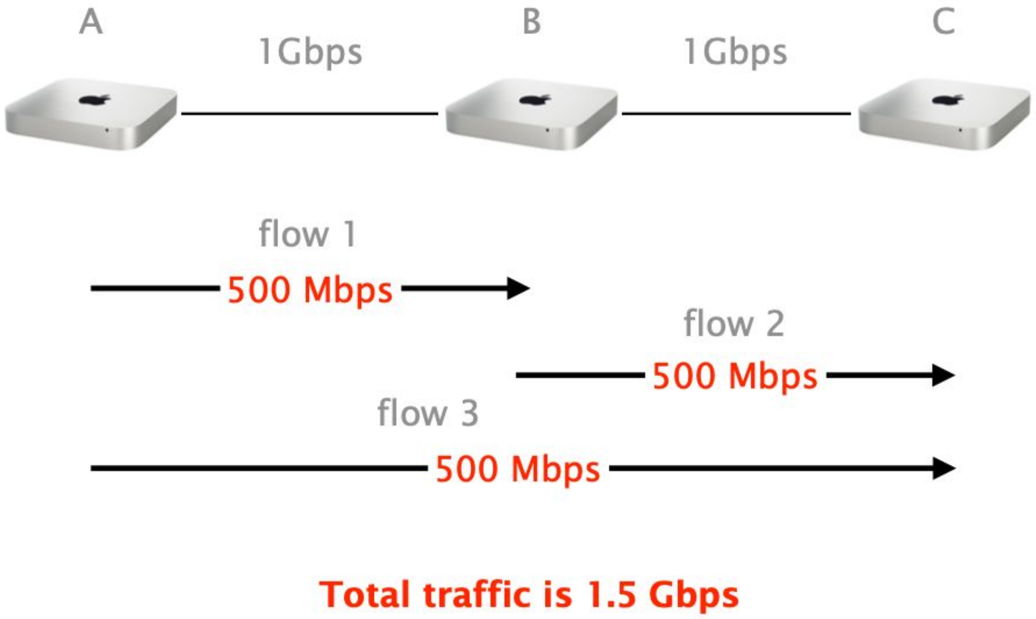


Total traffic is 2 Gbps!

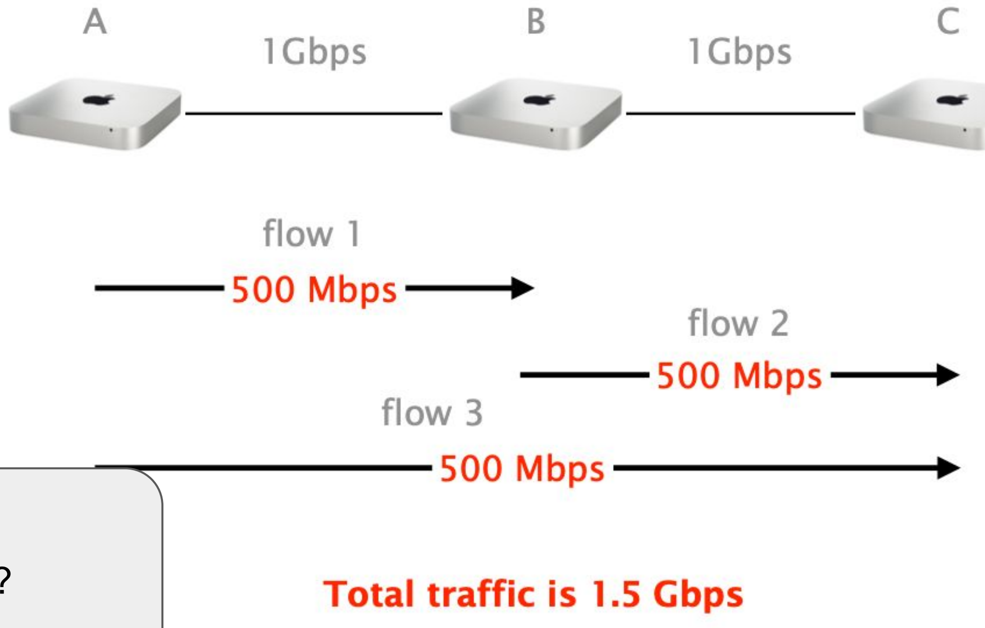
Fairness and efficiency don't always play along, here an unfair allocation ends up more *efficient*



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



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

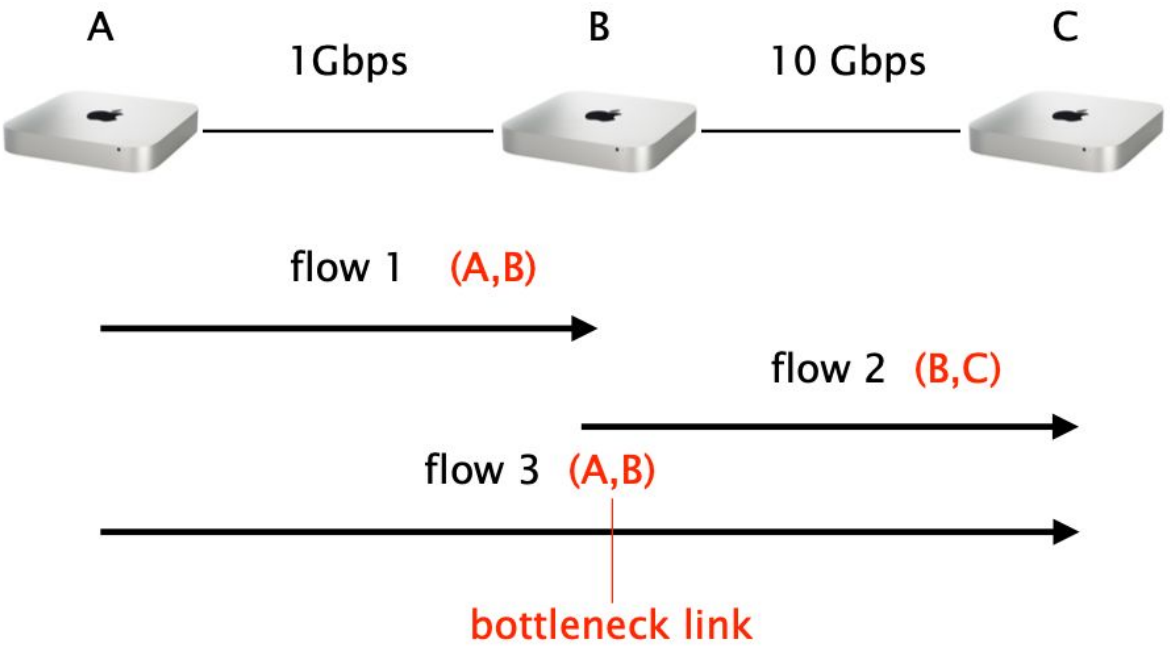


Is that fair?

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 bottlenecks



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
	Done!