Transport Layer

Transport

Network layer: communication between hosts

Transport layer: communication between processes

Transport

Network layer: communication between hosts

Transport layer: communication between processes

Muxing across many processes

Unit of data: segment

Transport

- Two principal transports: TCP and UDP
- TCP: Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- UDP: User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

Transport

routing

reliable
delivery

How do you ensure reliable transport
on top of best-effort delivery?

Transport

In the Internet, reliability is ensured by the end hosts, not by the network

Reliability is left to L4, the Transport Layer

Why?

Reliability is left to L4, the Transport 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

Network stack - reliability in L4

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

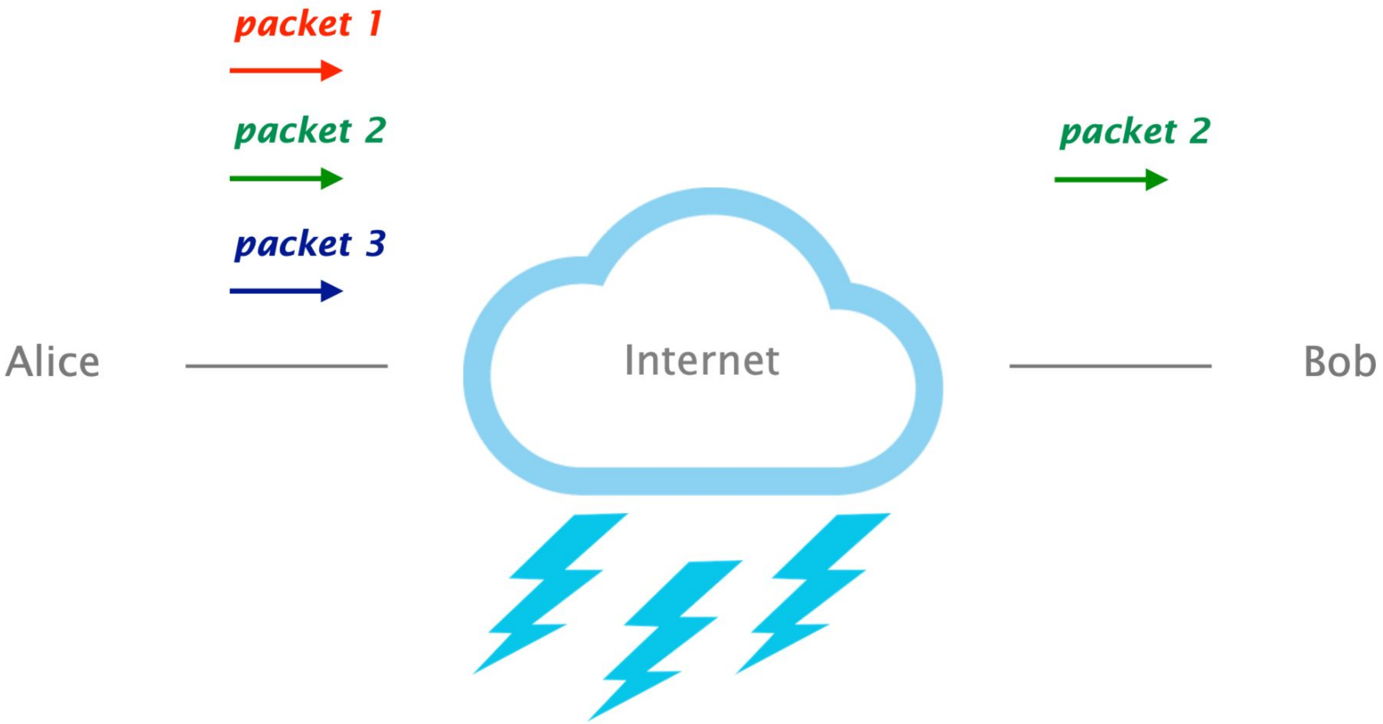
Network stack - reliability in L4

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

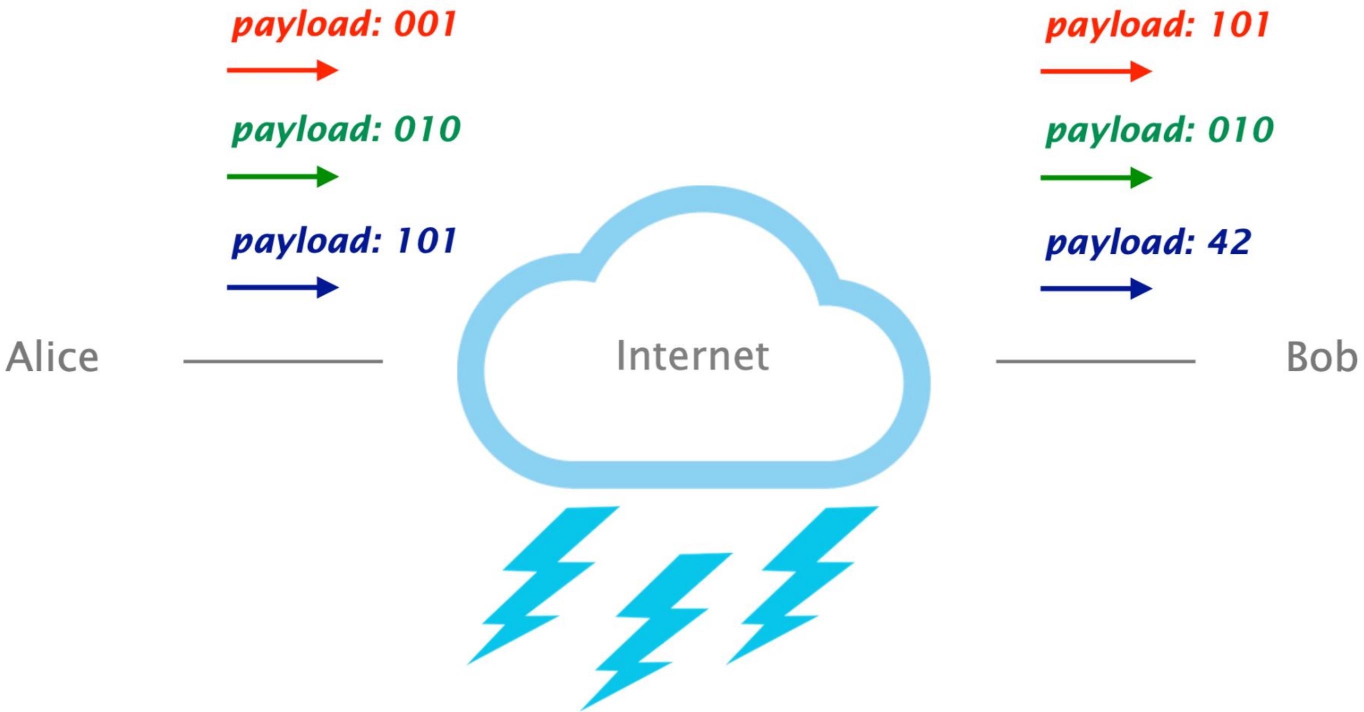
IP is “best-effort”

Example: Alice and Bob

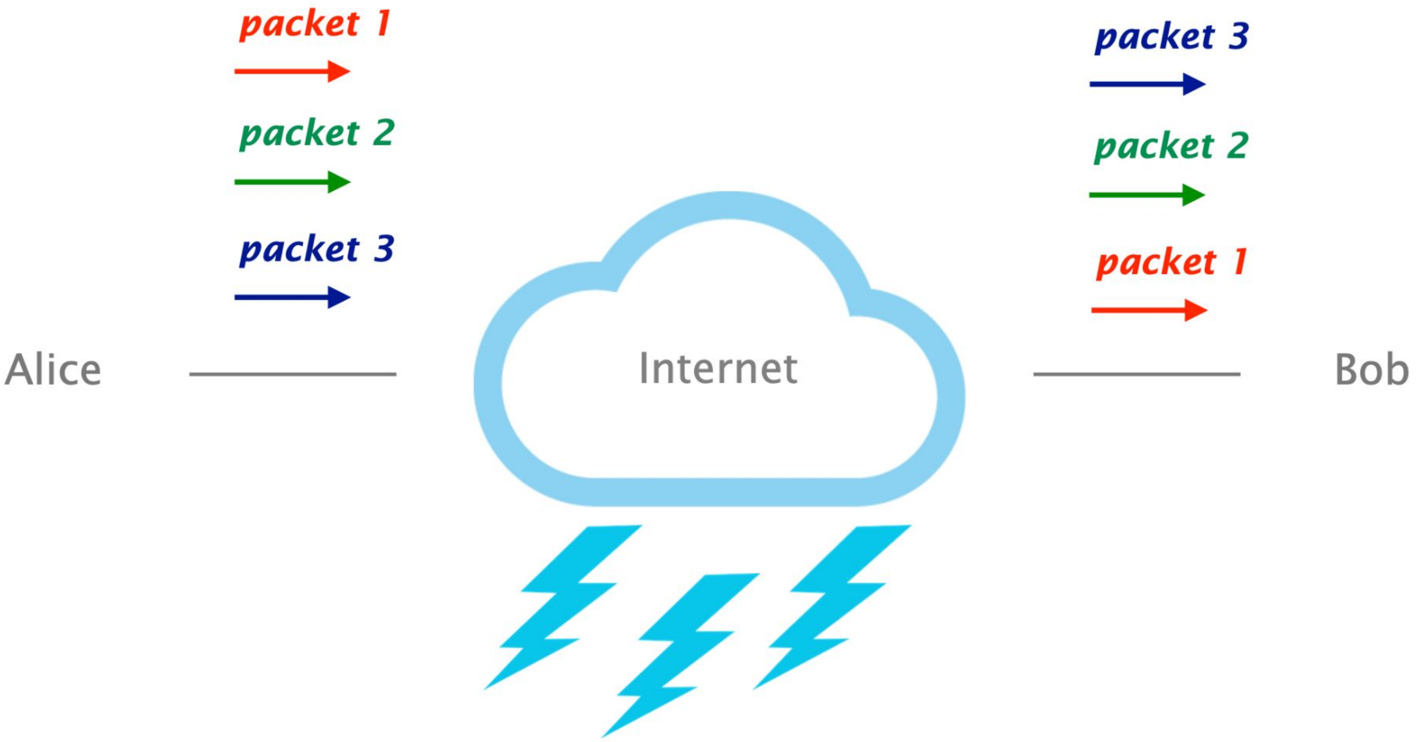
IP packets can be lost or delayed



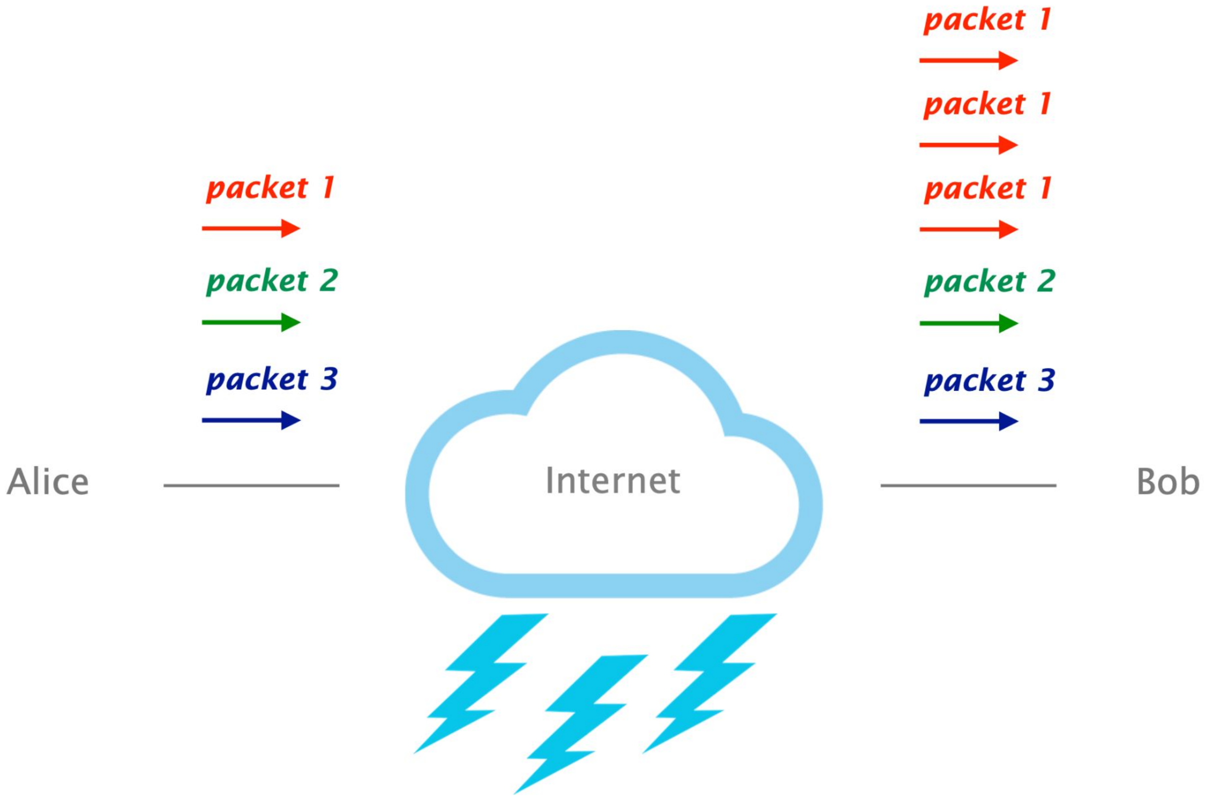
IP packets can get corrupted



IP packets can get reordered



IP packets can be duplicated



The four goals of reliable transport

goals	
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

The four goals of reliable transport

goals	
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

Correctness is clean / easy with routing

sufficient and necessary condition

Theorem a global forwarding state is valid if and only if

- there are no dead ends

  no outgoing port defined in the table

- there are no loops

  packets going around the same set of nodes

Correctness is clean / easy with routing

sufficient and necessary condition

Theorem

How can we come up with a similarly clean definition for transport?

- there are no loops

  packets going around the same set of nodes

Correctness in reliable transport

A reliable transport design is correct if…

attempt #1 packets are delivered to the receiver

Wrong Consider that the network is partitioned

We cannot say a transport design is *incorrect* if it doesn't work in a partitioned network…

Correctness in reliable transport

A reliable transport design is correct if…

attempt #2

packets are delivered to 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 doesn't know the unknowable

Correctness in reliable transport

A reliable transport design is correct if…

attempt #3

It resends a packet if and only if
the previous 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

Correctness in reliable transport

A reliable transport design is correct if...

attempt #3 — It resends a packet if and only if the previous packet was lost or corrupted

In both cases the sender has no feedback. How can it know to re-send???

Wrong — Consider two cas

- 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

Correctness in reliable transport

A reliable transport design is correct if…

attempt #4

A packet is always resent if
the previous packet was lost or corrupted

A packet may be resent at other times

Correct!

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

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

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

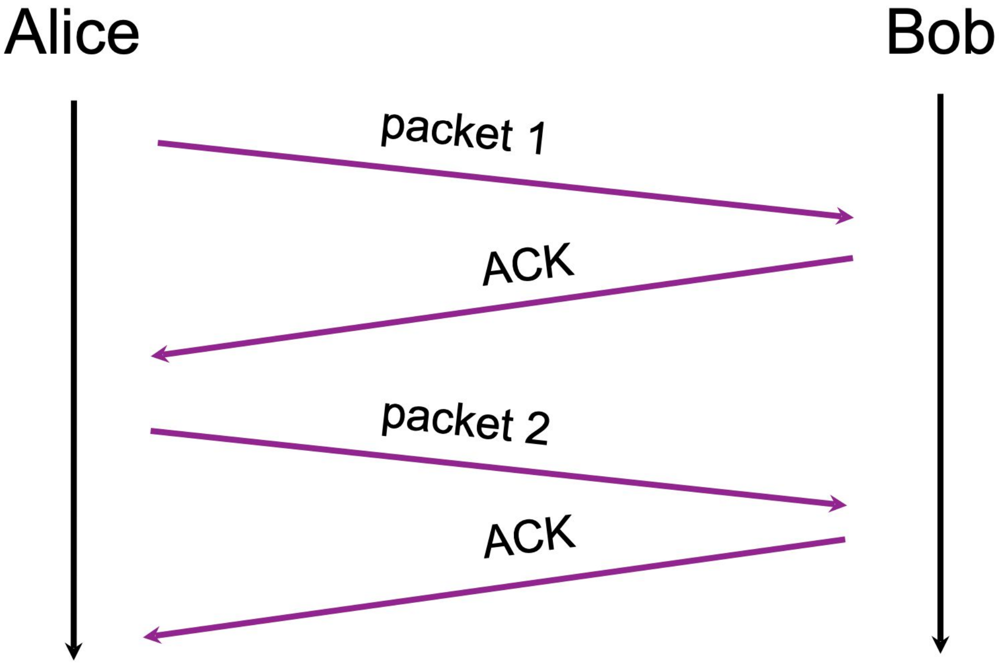
https://www2.tkn.tu-berlin.de/teaching/rn/animations/gbn_sr/

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

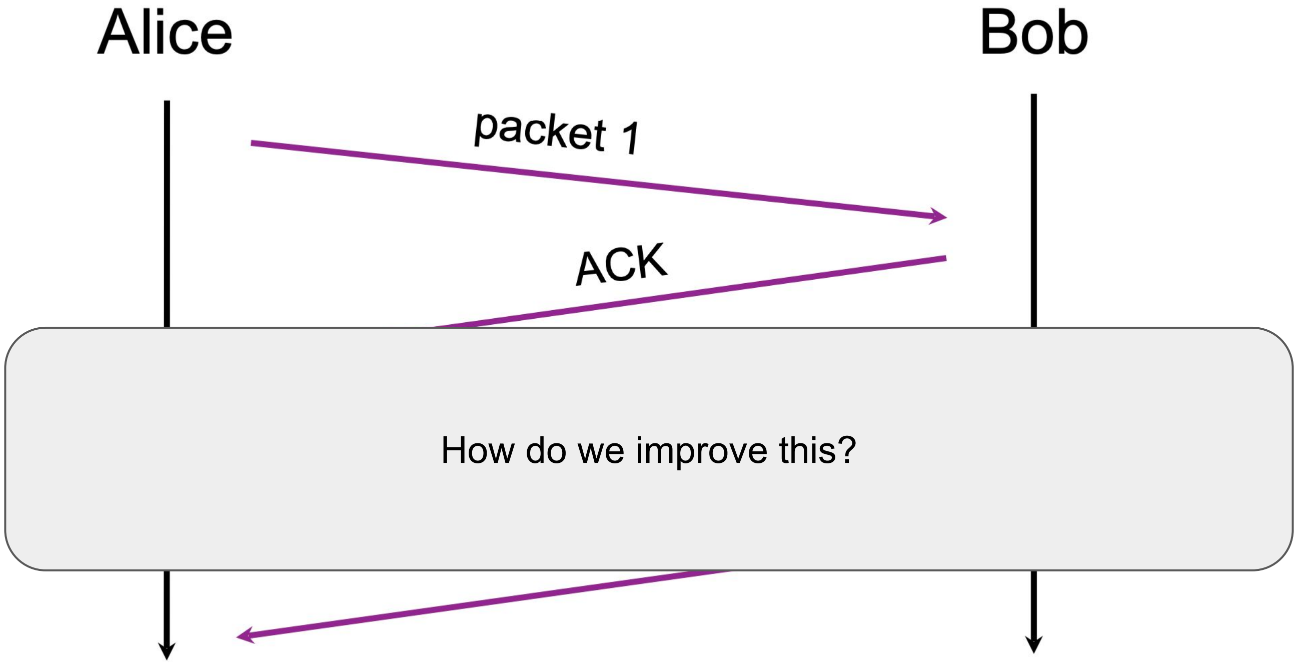
Timeliness argues for small timers, efficiency for large timers

timeliness	efficiency
small timers	large timers
risk	risk
unnecessary retransmissions	slow transmission

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



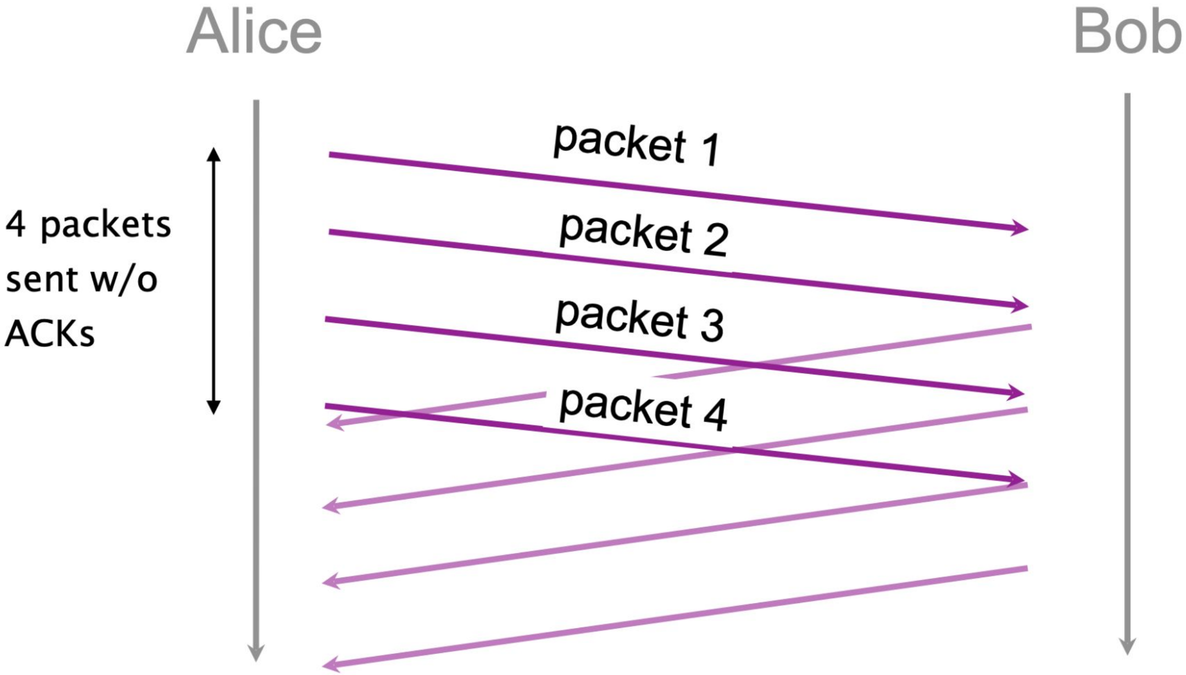
Even with small timers, stop and wait has terrible timeliness - one packet per round trip time (RTT)



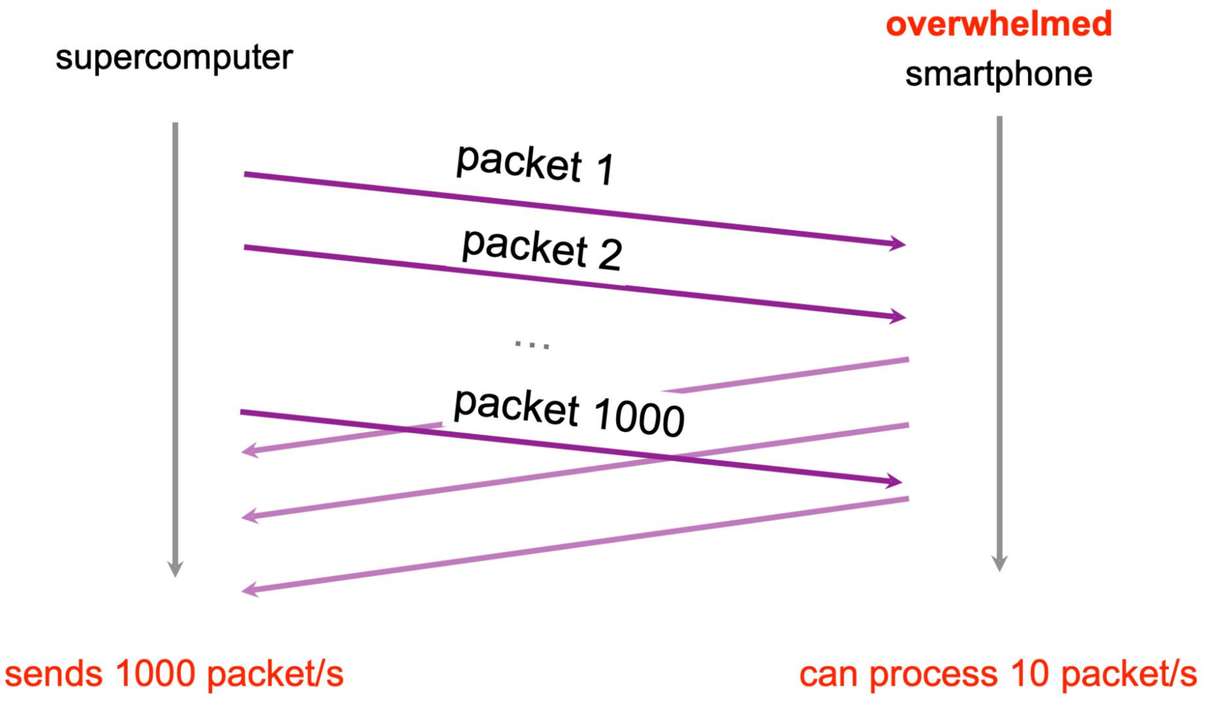
An obvious solution to improve timeliness is to send multiple packets at the same time

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

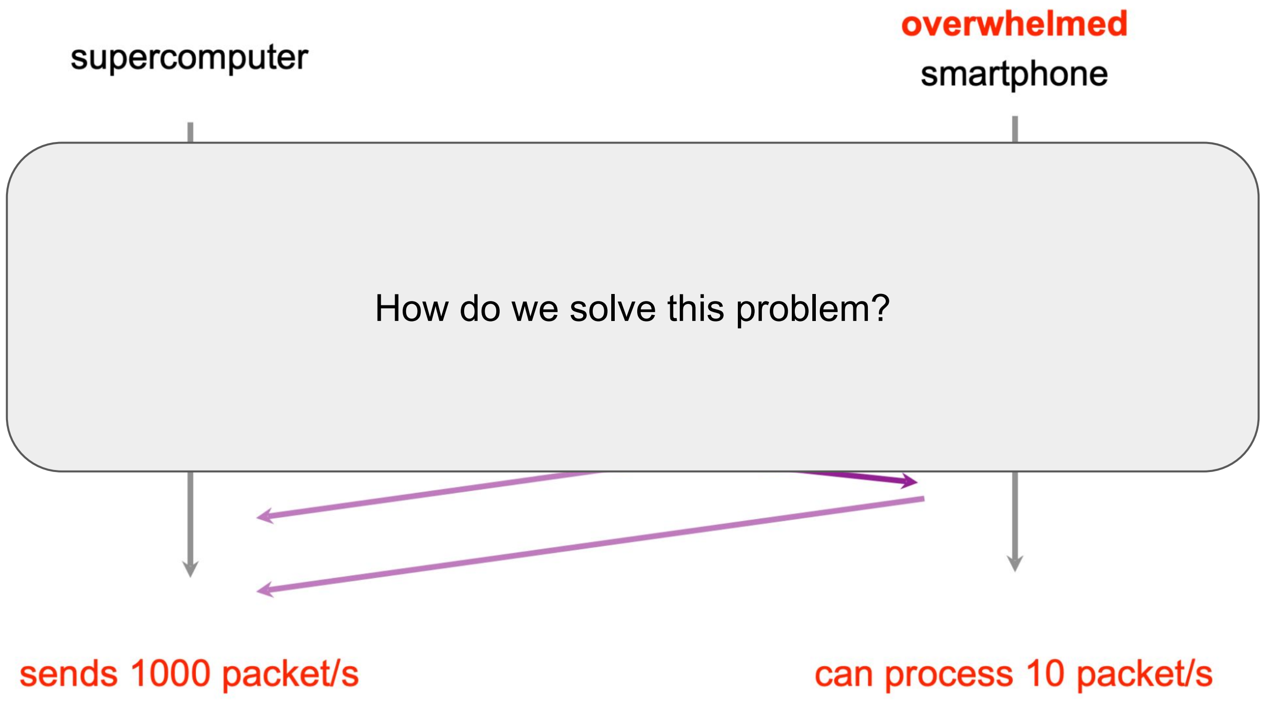
An obvious solution to improve timeliness is to send multiple packets at the same time



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