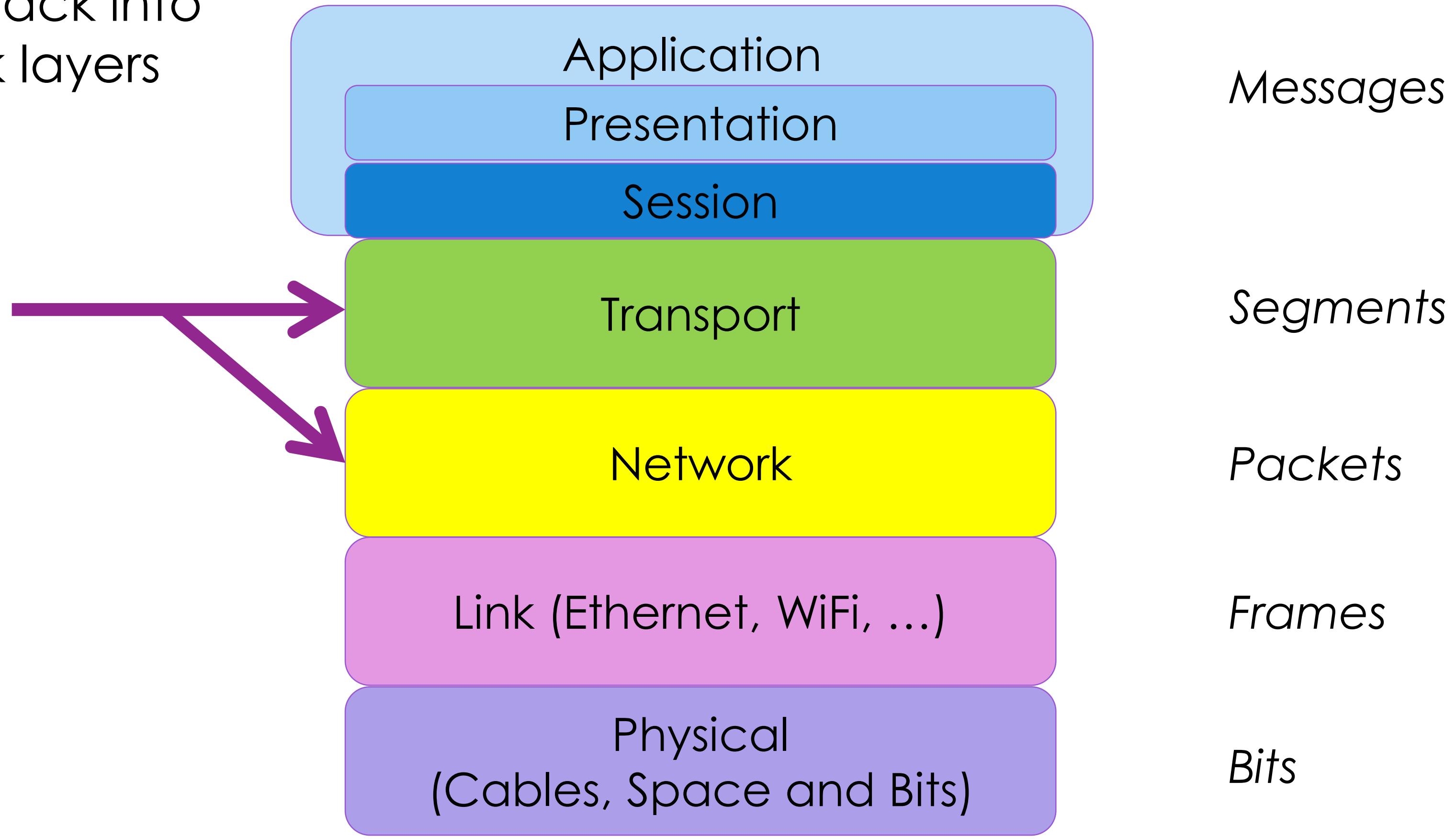
COMP3310/6331 – #19

Flow control and congestion

Dr Markus Buchhorn: *markus.buchhorn@anu.edu.au*

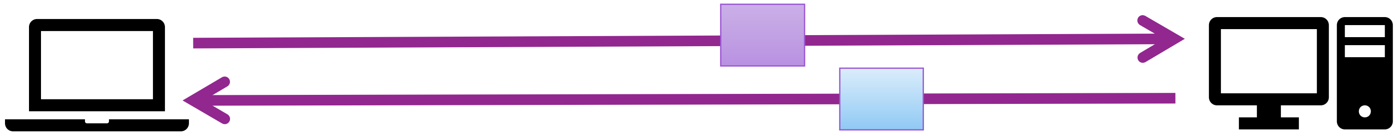
Where are we?

- Going back into the dark layers



Remember (TCP) Sliding Windows?

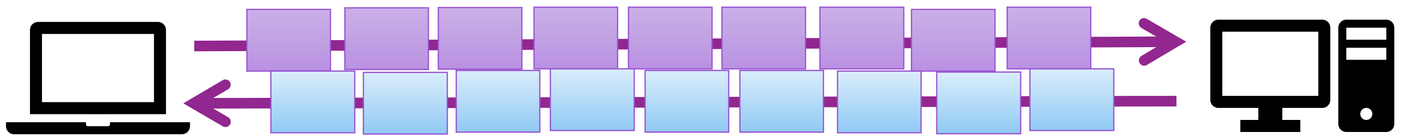
- Want reliability and throughput – and fill pipes!
- Start with ARQ – stop-and-wait
  - Single segment outstanding = problem on high bandwidth*delay networks



- Say one way delay=50ms so round-trip-time (RTT)=2d=100ms
- Single segment per RTT = 10 packets/s
  - Typical packet on Ethernet? Say 1000 bytes = ~10,000 bits -> 100kb/s or 10% of link
- Even if bandwidth goes up, throughput doesn't!

Sliding Windows

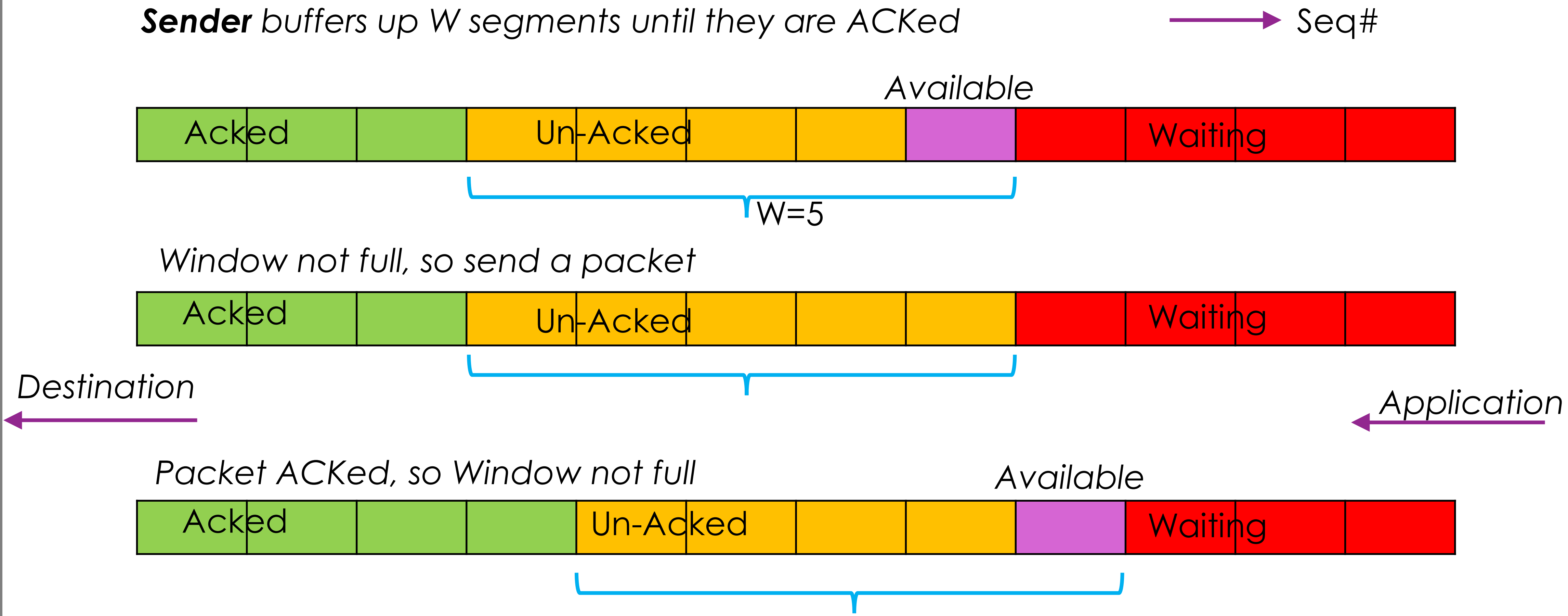
- Allow W segments to be 'outstanding' (unACKed) per RTT
  - Fill a pipeline with segments



- Set up a 'window' of W segments – **per connection**
- W=2*Bandwidth*delay
- At 100Mb/s, delay=50ms means W=10Mb
  - and assuming same 10kb segments, W=1000 segments
  - 500 are on their way out there!

Sliding Window approach

Sender *buffers up W segments until they are ACKed*



If(lost) then: ARQ – “Go Back N”

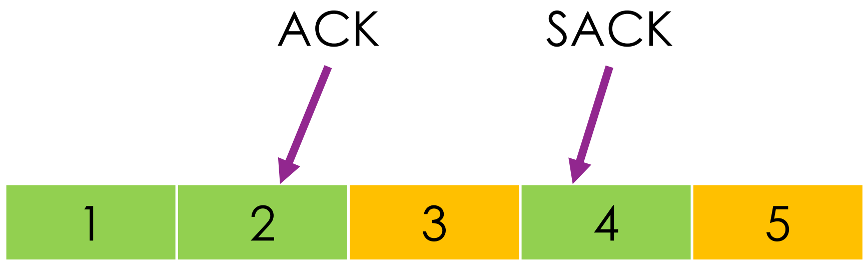
- **Receiver** buffers just a single segment
- If it’s the next one in sequence, ACK it, everyone happy
- If it’s not, drop it, *I just don’t care*

- Let sender retransmit what I’m actually waiting for

- **Sender** has a single timer. After timeout, resend
- Really simple, but somewhat inefficient

ARQ – "Selective Repeat"

- **Receiver** buffers <u>many</u> segments
  - Reduce retransmissions
- ACK what has been received in order
- <u>And</u> also ACK segments that haven't
  - Any gaps indicates missing segment!
  - *<u>Selective ACK (SACK)</u>*



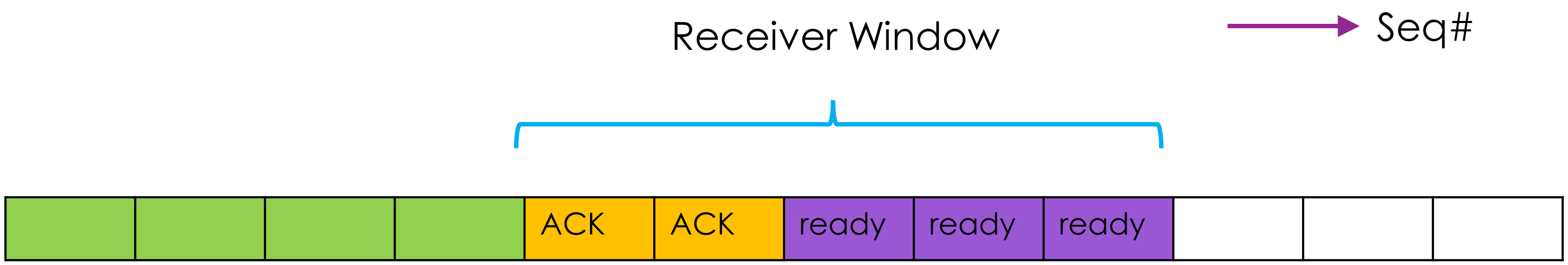
- **Sender** has a timer <u>per unACKed-segment</u>
  - As each timer expires, resend that segment
- Way more efficient, now widespread

Very sender/network oriented

- Sender manages the transmission
  - UDP – send-and-forget, no control
  - TCP - Slows down waiting for ACKs
  - Optimised to keep **network** full
  - What about the **receiver application**?
- Consider Receiver being swamped
  - HD video streaming to small device – it needs to **control** the **flow**

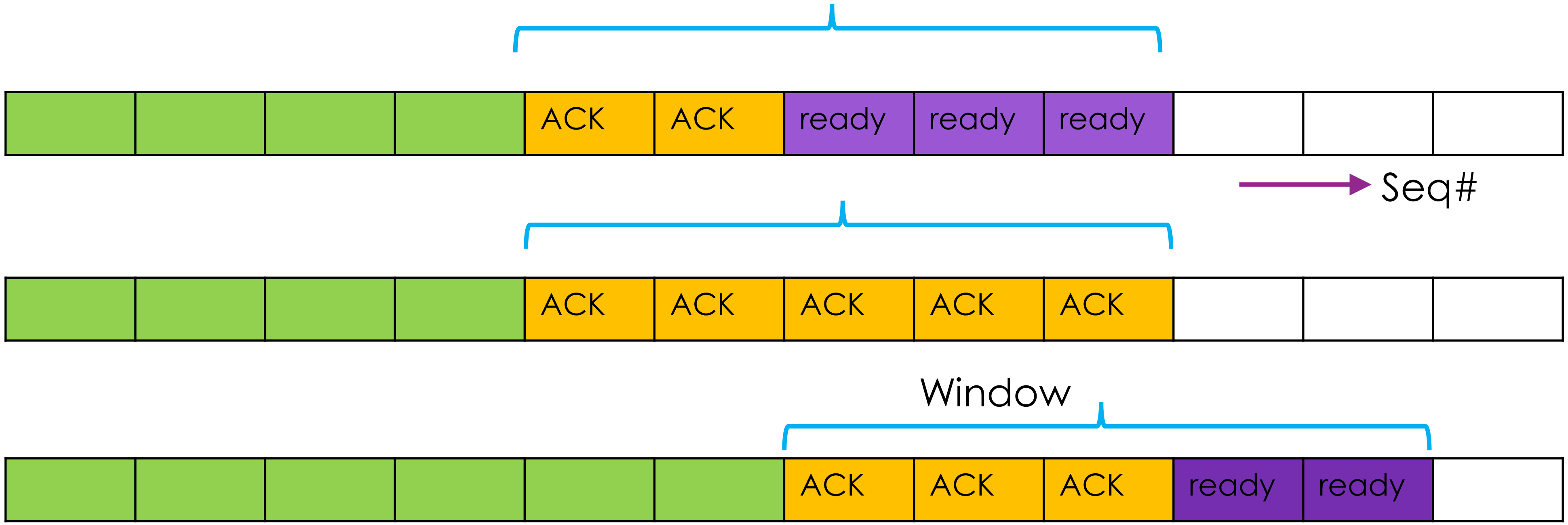
Flow Control:
Sliding Windows on the Receiver side

- **Transport** layer:
  – receives the segment from the network
  – and adds it to application buffer
- **Application** calls recv(*N-bytes*) to read from buffer
  – But what happens if the application is slow?



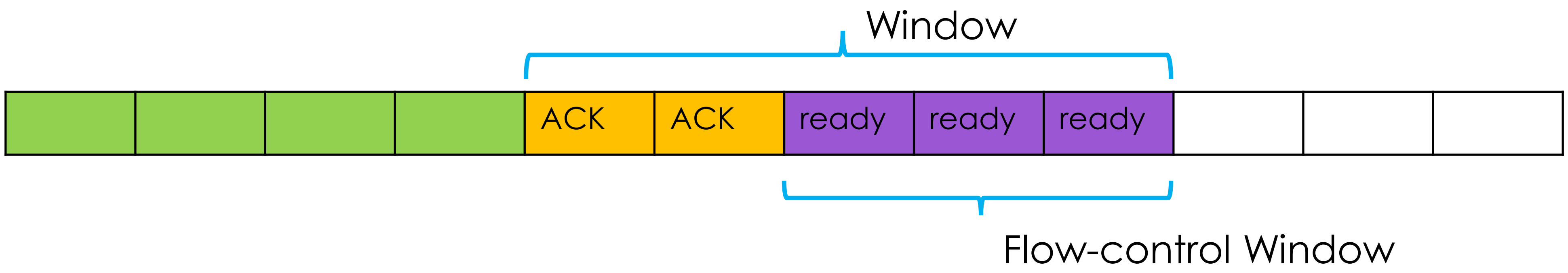
Sliding Windows on the Receiver side

- More segments arrive, fill (TCP) buffer – and eventually **application** recv()s



Two windows to get through?

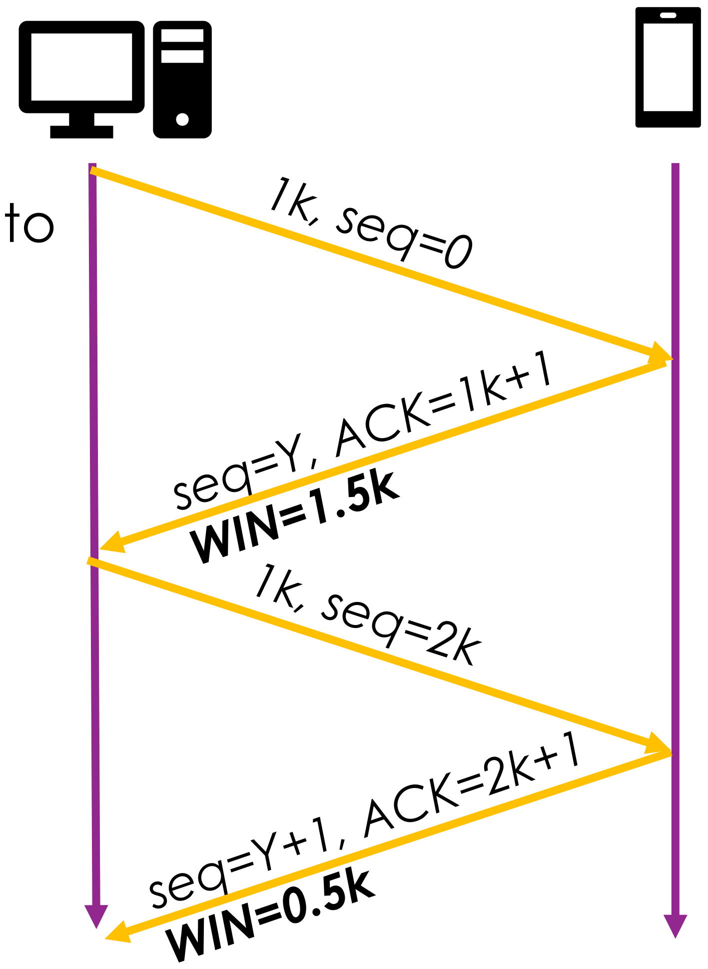
- TCP Sender Sliding Window (**W**)
  - Both sides know
- <u>Receiver</u> Sliding Window = <u>Flow Control</u> Window (**WIN**)
  - Number of "ACCEPTABLE" segments to be sent



- Sender gets told **WIN** and uses <u>lower</u> of **W** and **WIN** as the 'effective' window

A little simpler

- Sequence numbers identify where sender is up to
- Acknowledgements where receiver is up to
- But receiver can also report buffer available
- Simple Flow Control



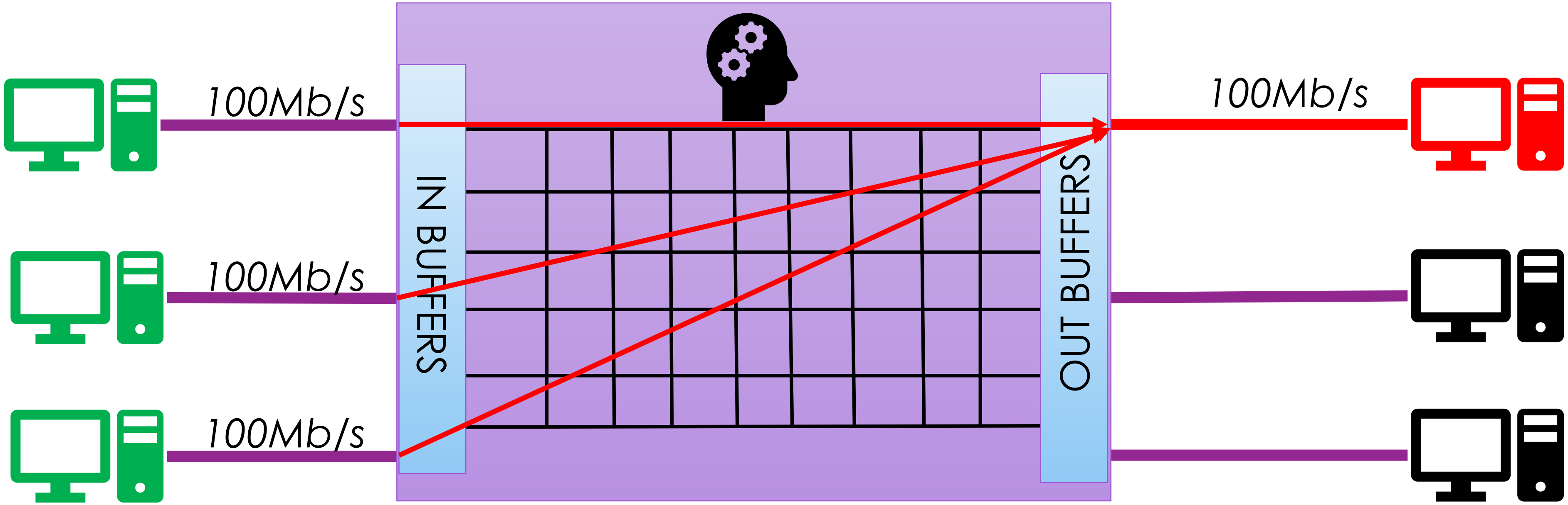
Congestion

- A traffic jam – something filled up and is holding up the rest
  - A dynamic condition
  - Somewhere (unknown) along the (unknown) path

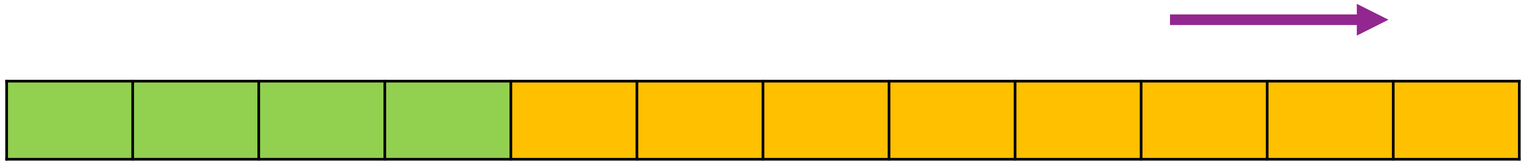
- Senders keep sending
  - Makes it worse, for themselves, and everybody else
  - Congestion -> loss
- It is <u>not</u> the **links** that “cause” loss (by being congested)
  - They run at a specific clock, bits in/bits out.
  - They set the limits

Routers/switches

- Too many inputs for the one output

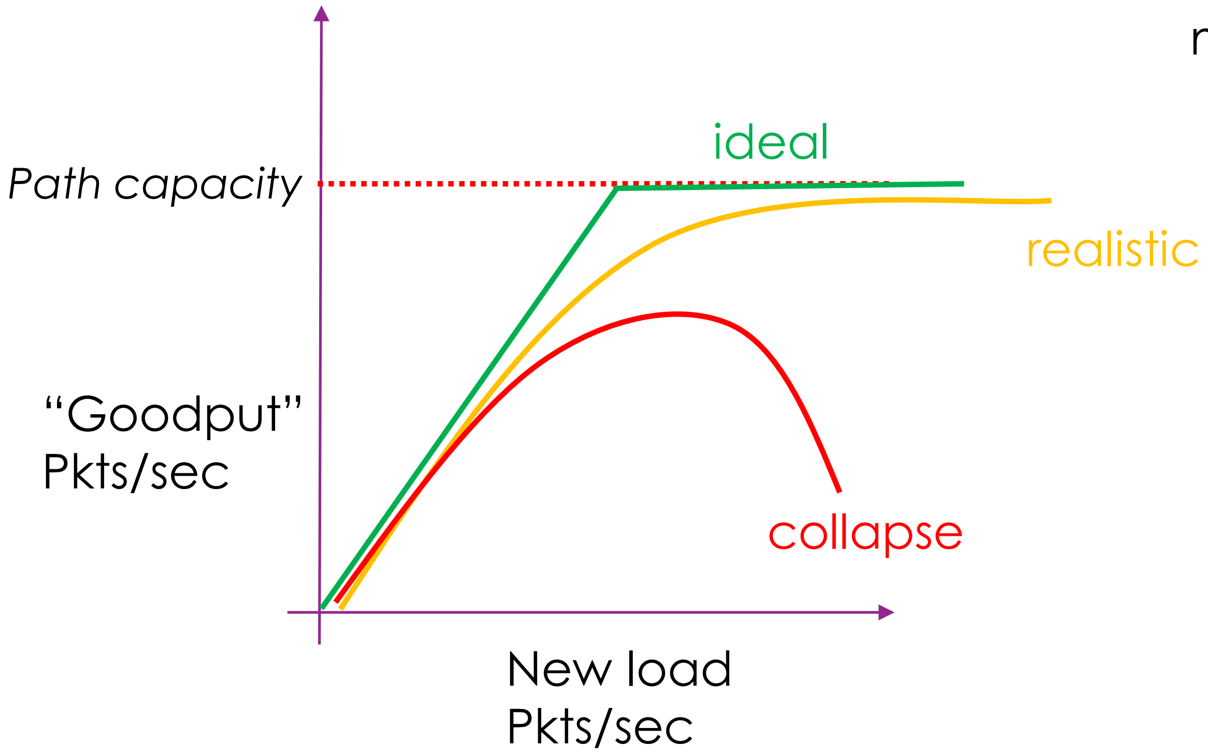


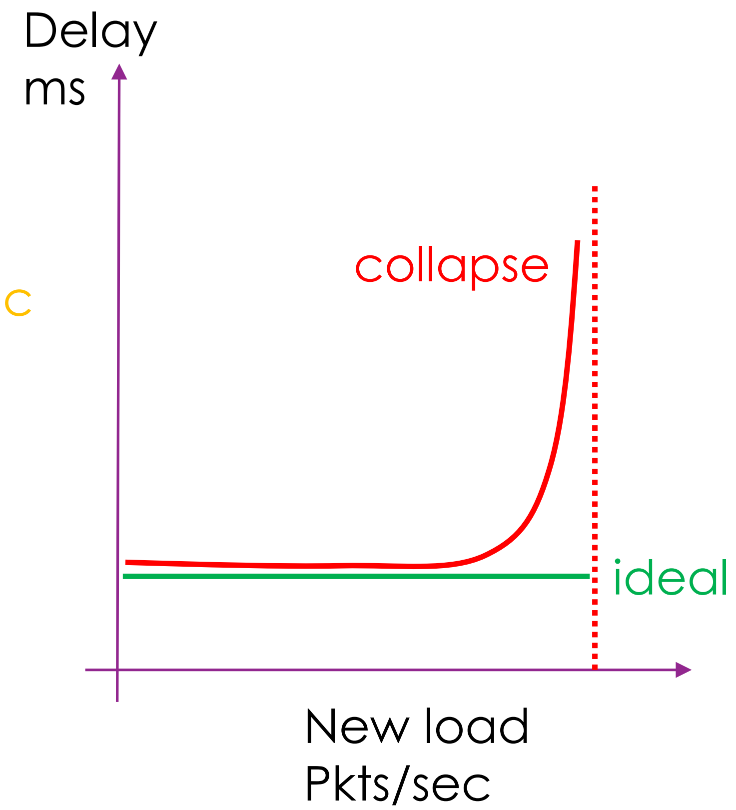
Router Buffers: Queues…



- FIFO (First in, First out) queues on every interface
- Great for absorbing (short) bursts of traffic
  - Data-rate in > data-rate out
- For a while… then queue overflows, and packets get dropped
- Largely driven by traffic patterns
  - Multiple conversations randomly sending to the same path at the same time
  - Assuming similar bandwidth links in/out

Congestion Effects





Why???

- Rising load fills buffers
  - Delays go up
- Overflowing buffers drop packets
  - Loss rises

- What do the receivers do?
  - ASK FOR RETRANSMISSION
- What do the senders to?
  - RETRANSMIT
- *Network fills with retransmitted packets, new packets are held back*
  - *Goodput goes to zero*

Managing capacity

- Want to operate (just) below congestion damage
  - Use the network to nearly “capacity”

- Need to allocate total capacity:
- Efficiently: get as much as I can, without causing congestion

and

- Fairly: everyone gets a reasonable share

Who handles that?

- To be effective, **both** Transport and Network layers have a role

- Network layer (IP) <u>sees</u> congestion
  - It's happening in the routers' buffers
  - And it could provide feedback

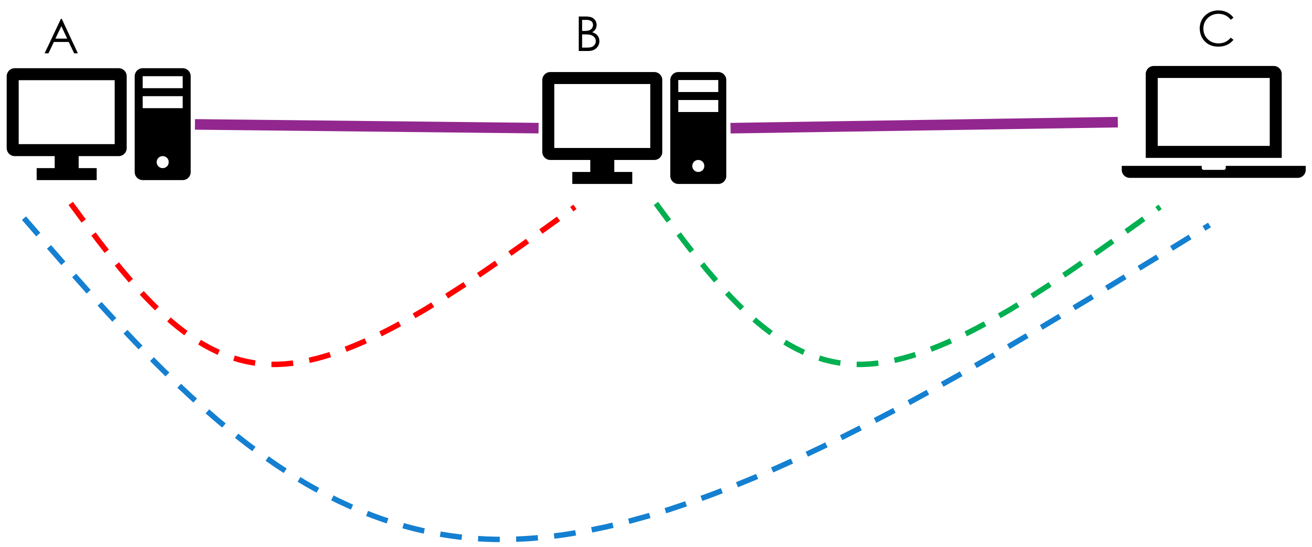
- Transport layer <u>causes</u> congestion!
  - But can't see where.
  - It can back off on transmissions

Isn't it statistical multiplexing?

- Could allow all senders just to fight it out – eventually it's even?
  - The very big and the very small
  - Problem: in congestion <u>everybody</u> loses.
- This is hard:
  - Different applications have different behaviours
    - cat video vs security sensor
  - Load is constantly changing (time)
  - Congestion may be happening in multiple, different, places (space)
  - There is no central view (everyone's blind)
- Need to find solution(s) where:
  - Senders adapt concurrently and continuously?
  - We can make it <u>efficient</u> and <u>fair?</u>

Fairness and Efficiency

- What's fair?
- Sometimes can't have both…

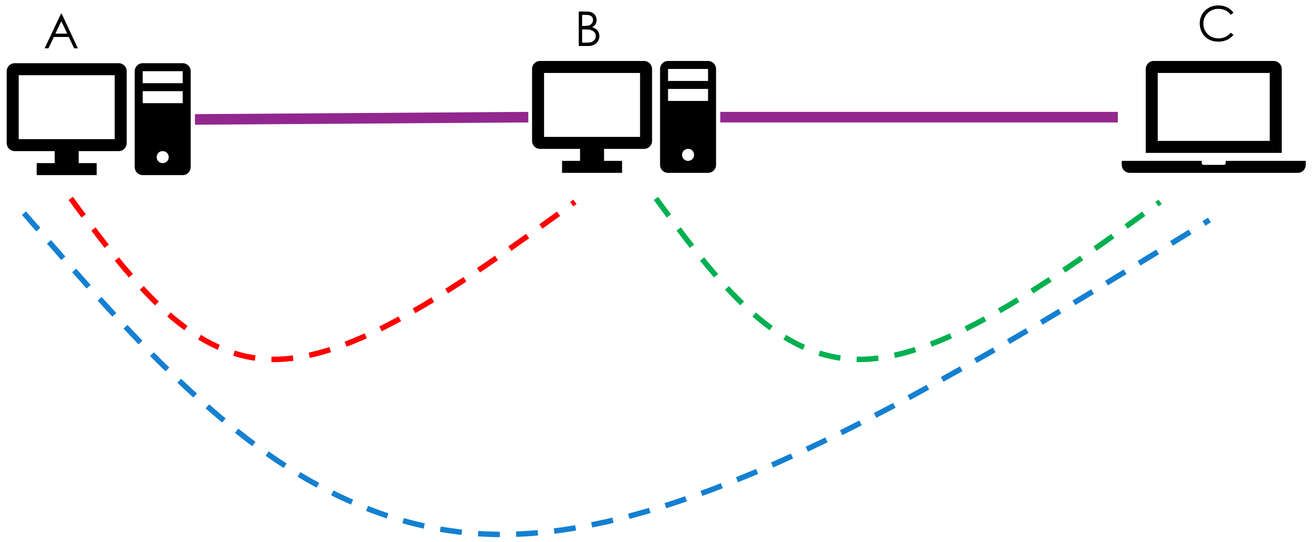


Fair…	
AB	0.5
BC	0.5
AC	0.5
Total: 1.5	

Efficient…	
AB	1
BC	1
AC	0
Total: 2.0	

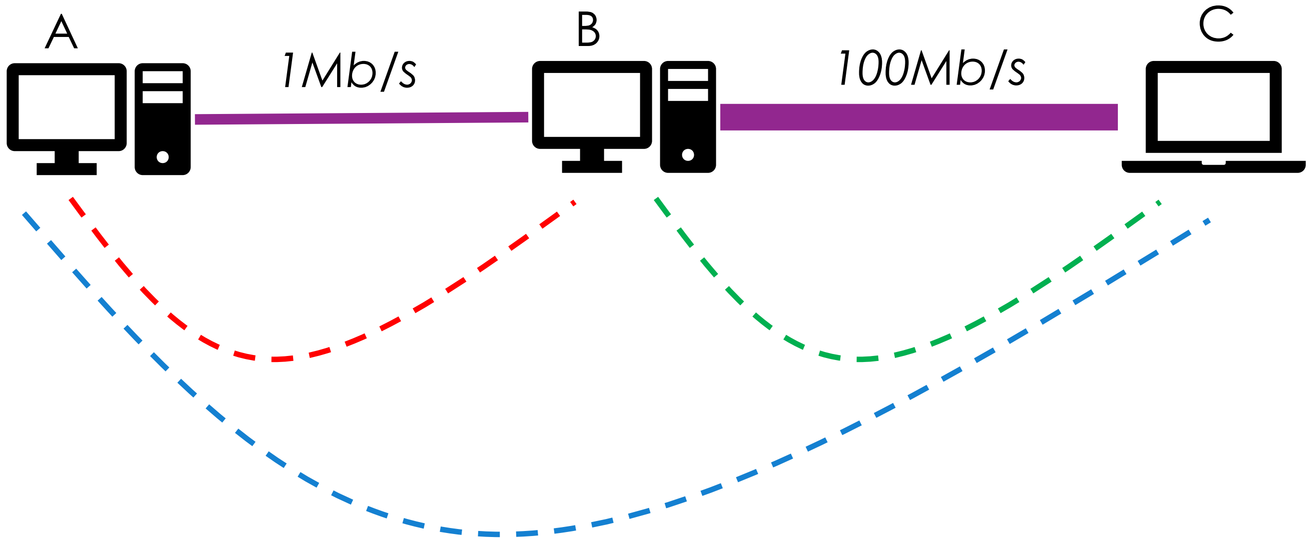
“Equal per flow” fairness?

- AC uses twice the network of AB, BC – is that fair?
- Exact fairness is hard. Avoiding full starvation (AC=0) is more important
  - Some starvation might be ok…



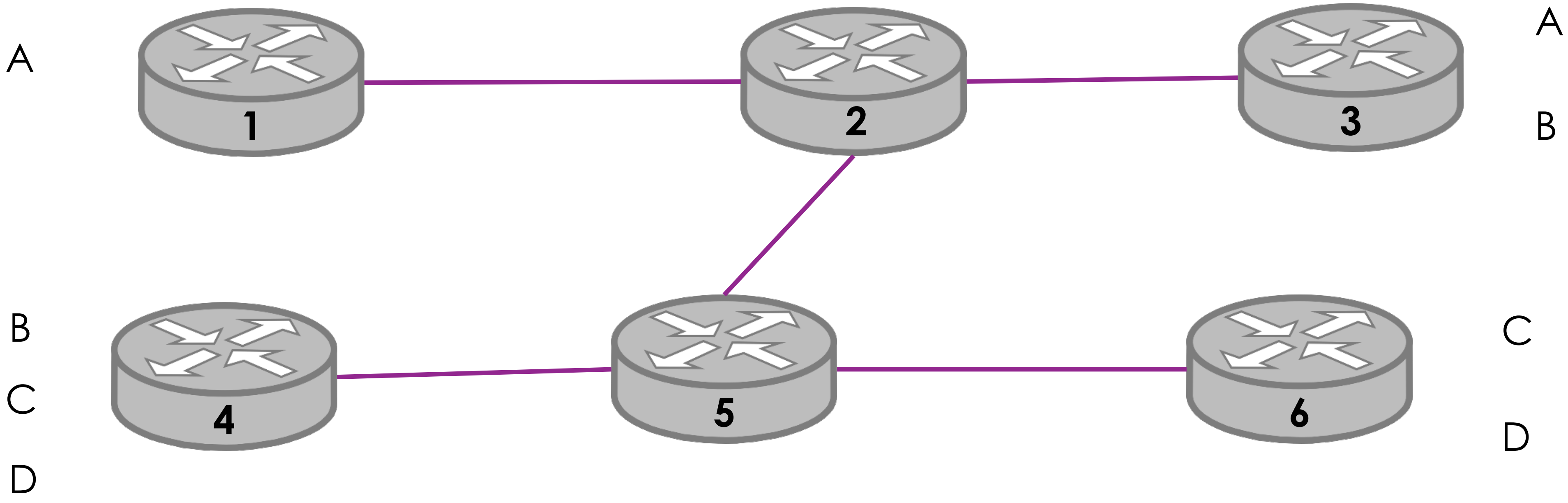
Network bottlenecks – unequal paths

- AC is choked by A-B link. BC is choked by B-C link
- So now what's fair?



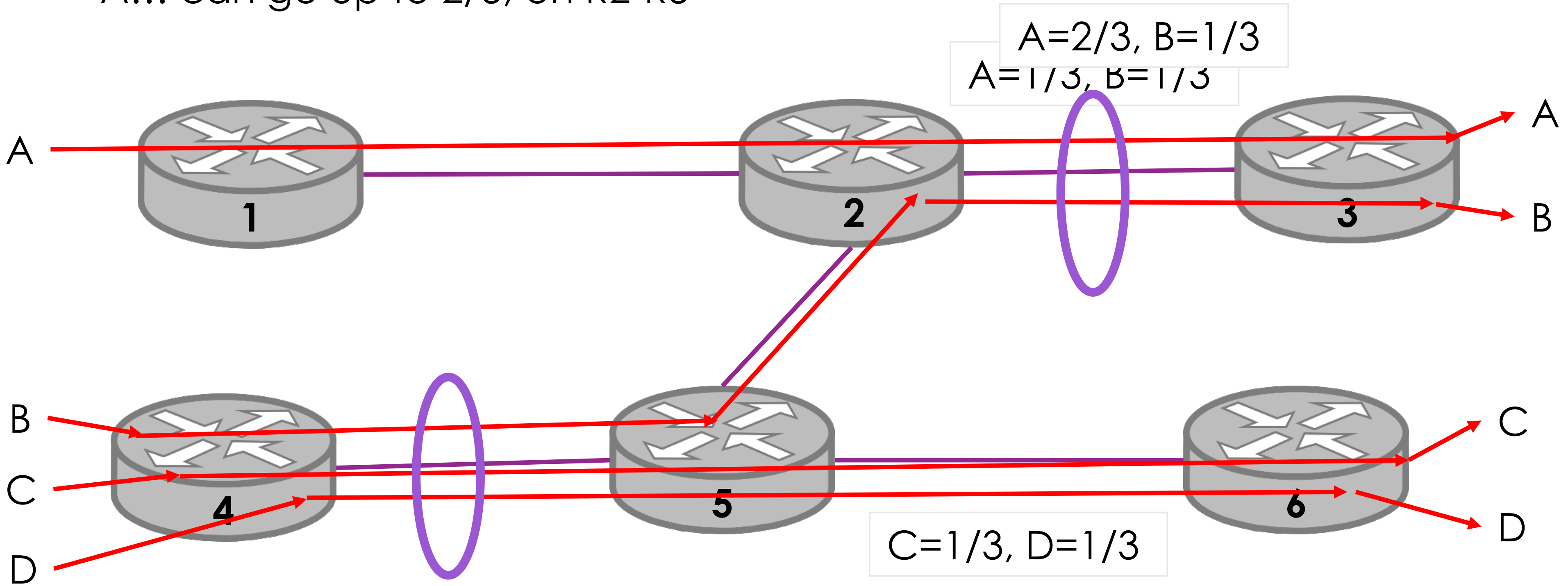
Max-Min Fairness

- Allocating bandwidth such that :
- *"increasing the rate of one flow will decrease the rate of a smaller flow"*
  - "Maximising the minimum" – keep adding, and sharing what's left.



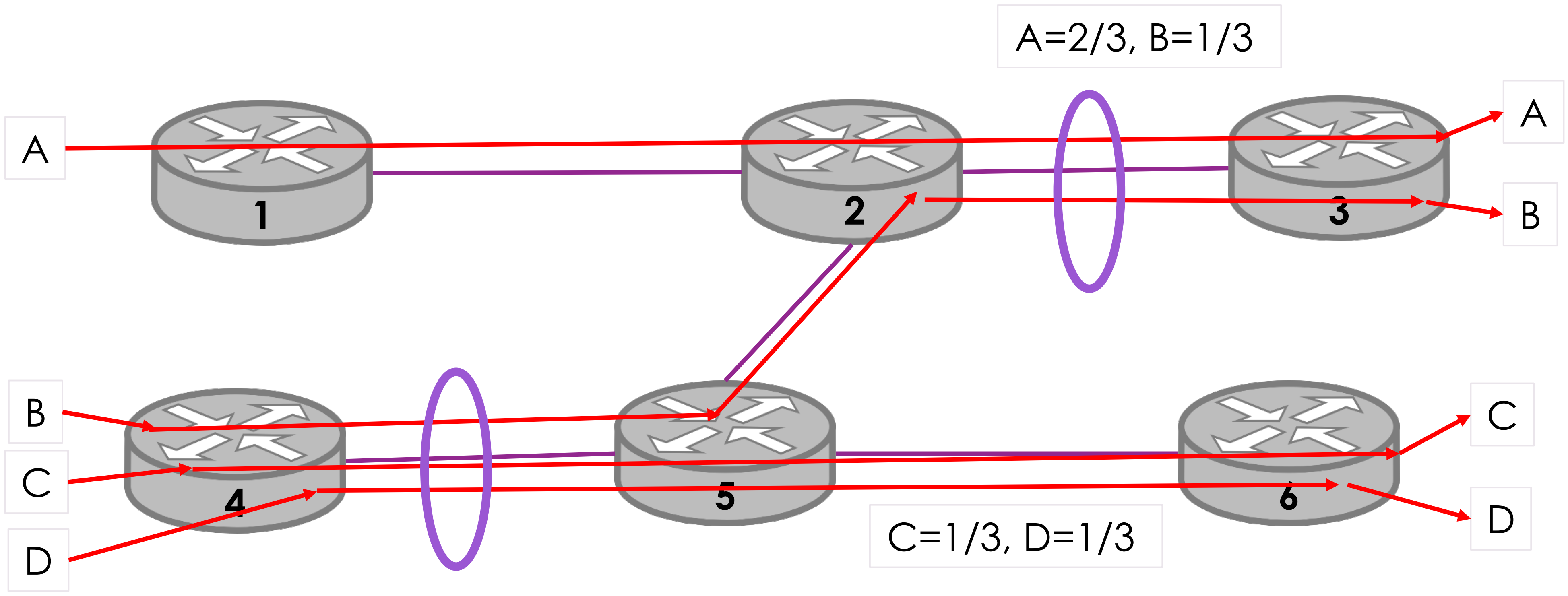
Max-Min Fairness

- Start from zero. Increase bandwidth of A,B,C,D till something bottlenecks
  - R4-R5 fills at 1/3 each for B,C,D. Hold them down. What's left to raise?
  - A… can go up to 2/3, on R2-R3

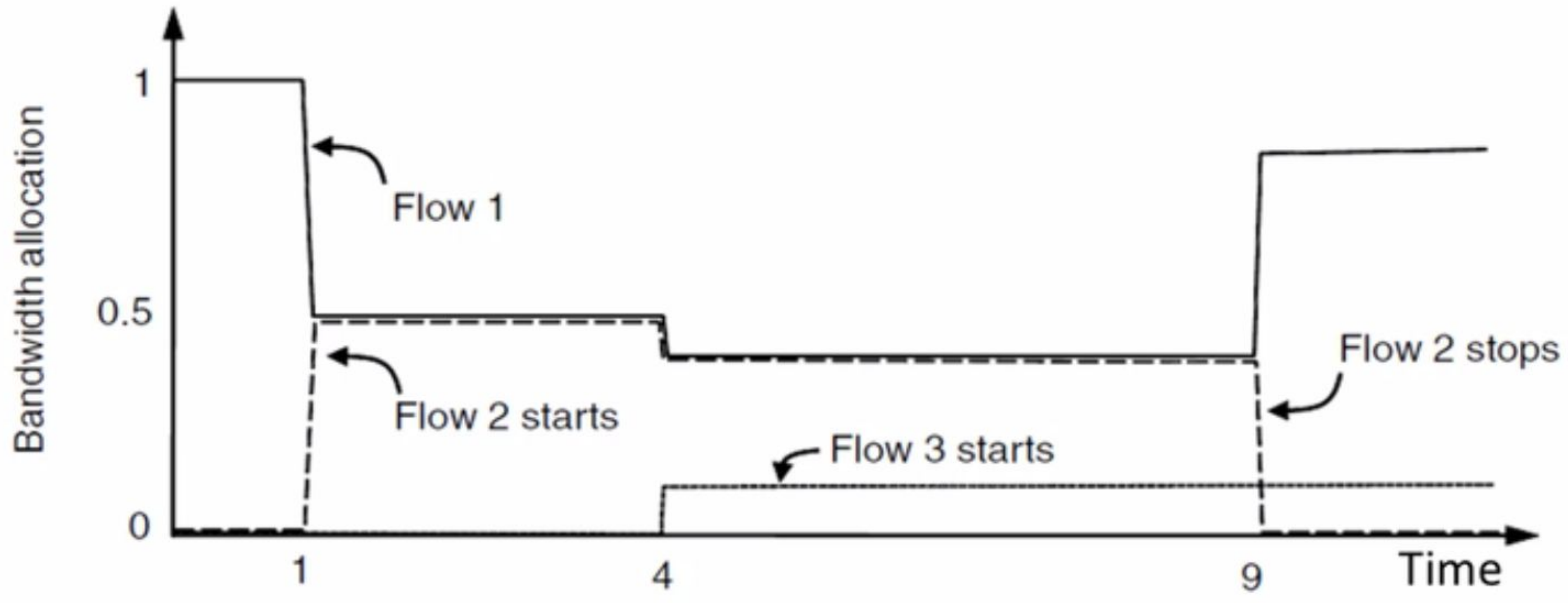


Max-Min Fairness

- So A = 2/3. B,C,D = 1/3
- R2-R3 and R4-R5 are full. Other 3 have unused capacity.



And adapt over time



So how to adapt?

1. Open/Closed loop
   - Open: reserve a circuit ahead of time
   - Closed: adjust on feedback
2. Host or Network driven
   - Host manages the allocation (use)
   - Network policing is strong, but inflexible
3. And “allocate” bandwidth: Rate based or Window based
   - Tell application to send at a specific rate, or
   - To watch window sizes

- *TCP is Closed-Loop, Host-Driven, Window-Based*

Two layers, working together

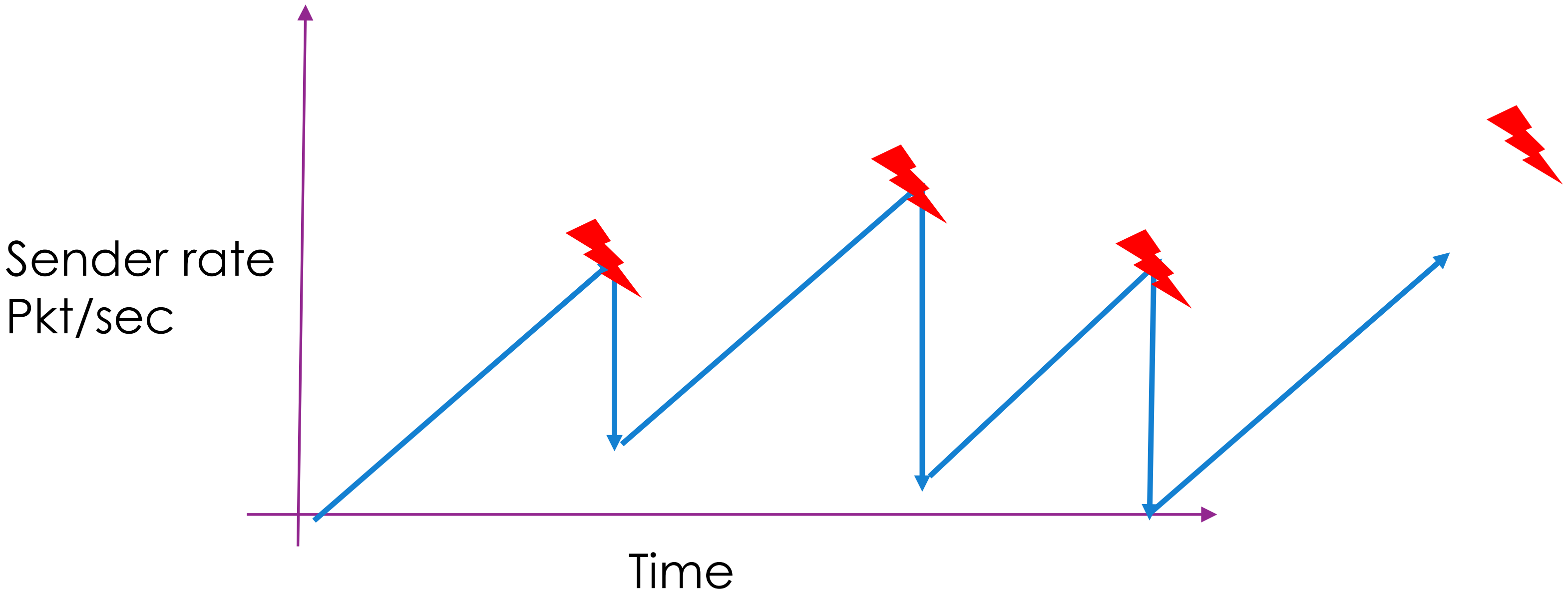
- Network layer (IP) provides feedback on allocation?
  - Actually, it indicates congestion

- Transport layer (TCP) modifies sender behaviour
  - TCP window sizes get adjusted
  - Dynamically, in response
  - This is a ‘control law’

- <u>Additive Increase, Multiplicative Decrease (AIMD)</u>
  - Senders additively increase rate, while no congestion (gently, gently)
  - Senders multiplicatively decrease rate when there is congestion (quickly, quickly!)

AIMD Sawtooth

- Slowly increase to probe the network
  - Multiple small steps that add to the rate
- Quickly decrease to avoid congestion collapse
  - Single(+) large percentage decrease



Nice features

- Converges to a fair and efficient allocation when all hosts run it
  - And everyone(*) does, with some parameter variations
  - Doesn't care about the topology
- Works effectively compared to other control laws
  - Slow decrease=bad, fast increase=bad, both slow=bad, both fast=bad
- Just needs a **single** signal from the "network" (actually, receiver)
  - Path is congested, or not.

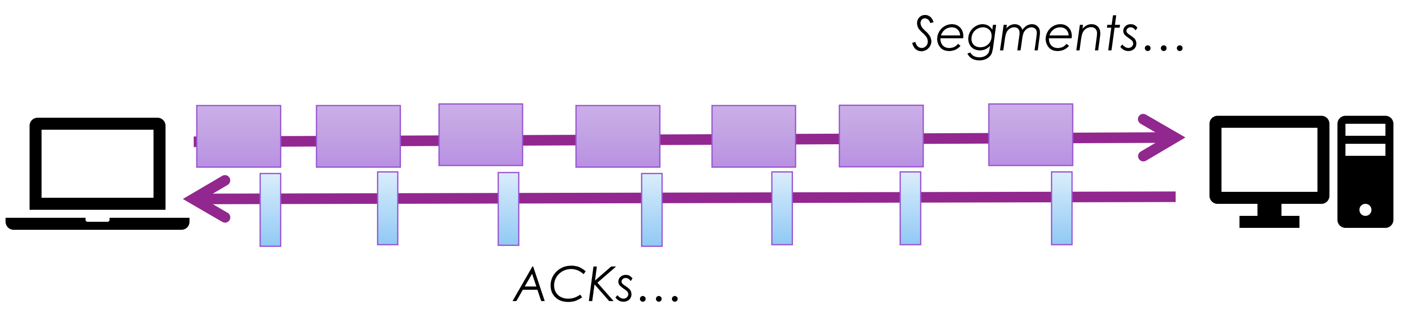
How does the network signal the sender?

Remember – multiple TCP implementations, by OS and date and …

Signal	Pros/Cons?
Packet loss	• Really obvious • Don’t detect congestion till it happens
Packet delays	• Detect congestion earlier • Detection is more inferred than actual
Router signal *Explicit Congestion Notification (ECN)*	• Detect congestion earlier • Needs the affected router and hosts to support it

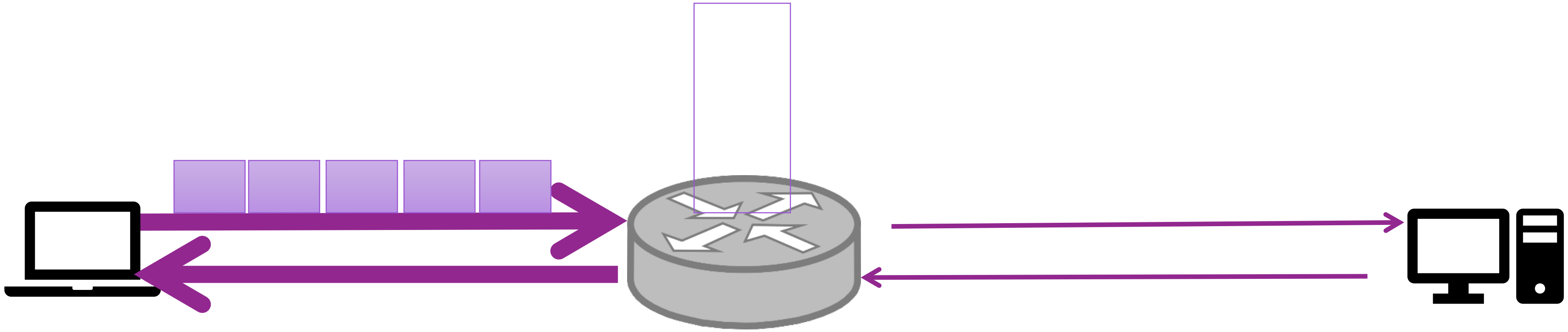
Implementing AIMD

- What are the best numbers for increase/decrease?
- Several components in TCP contribute – let’s focus on a few.
- Start with ACK clocking…



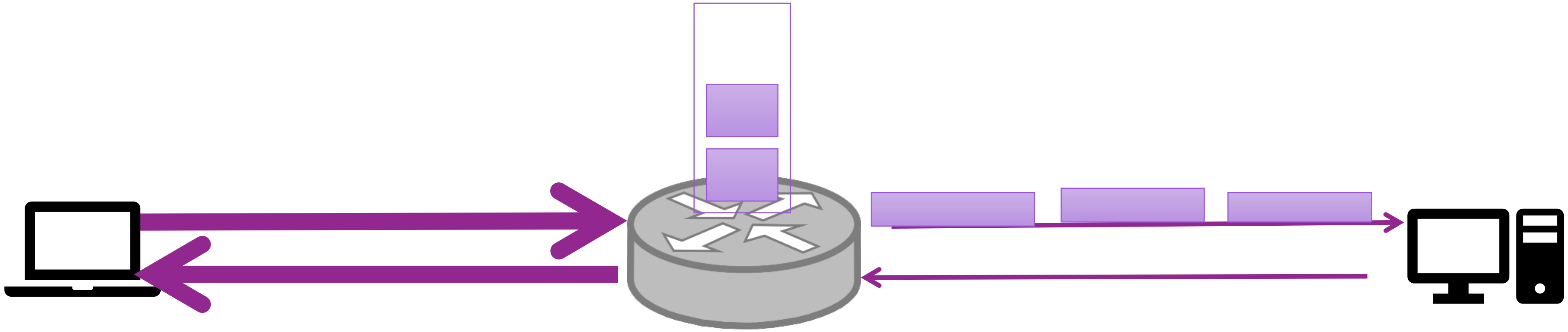
ACK clocking process

- High-speed link, talking to low-speed (or congested) link
- Sender sends a burst of packets to destination (to router with big buffer)
  - Doesn't know any better!



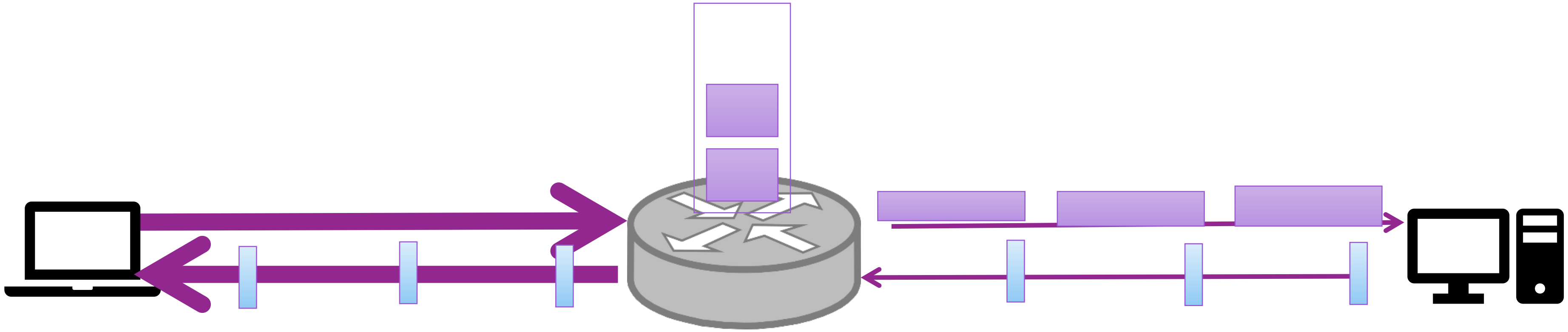
ACK clocking process

- Packets get buffered, and
- Low-speed link takes longer – packets get ‘longer’



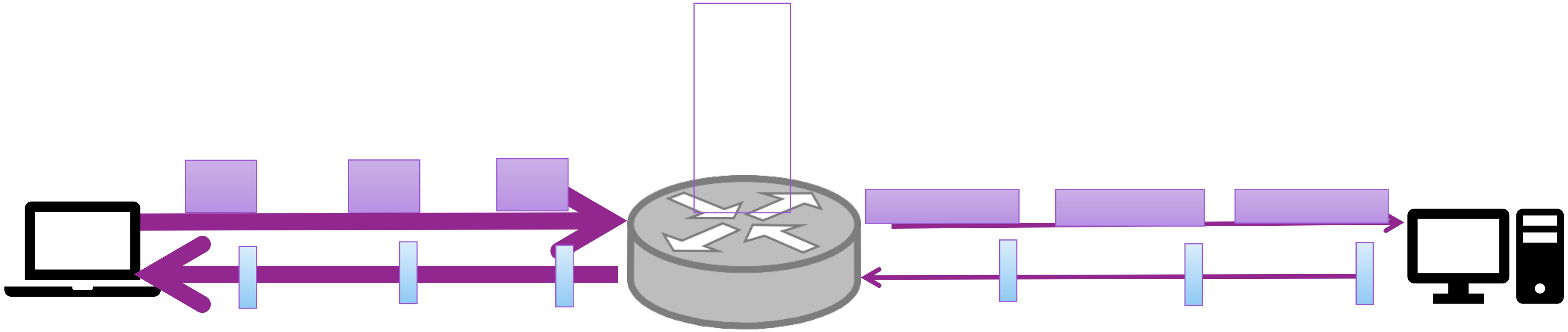
ACK clocking process

- ACKs returned at rate of slowest link!
- Sender learns to back off



ACK clocking process

- Sender matches ACK rate. <u>Buffers can drain</u> – congestion avoided
- Bursty traffic has become a smoother stream
- And we get a new measure – the **<u>'Congestion Window'</u>** (CWND)
  - Smaller than W (=2*B*delay). *[and <u>not</u> related to Flow-control Window (WIN)]*



Getting started

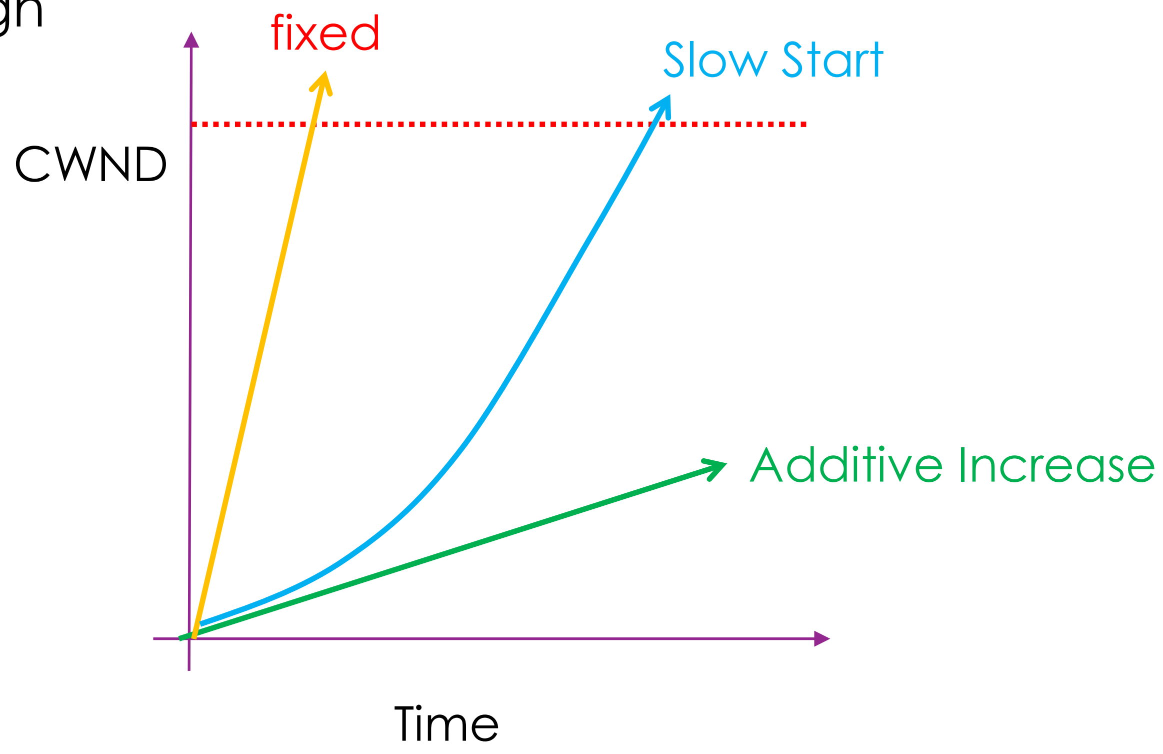
- On initial TCP connection, <u>what is CWND</u>?
  - Guess? Too many variables (bandwidth, delay, congestion, …)
  - Pick something? Could be way under, or over.

- TCP Additive Increase (on start):
  - Start with CWND of N bytes (~1 packet).
  - Every round trip without loss, make CWND bigger by 1 packet
- Increase very gently, but it could be a <u>long</u> time to reach the ideal CWND
  - Whatever it currently is…

- Want an algorithm for TCP CWND growth to **start** a bit **faster** - and it's called…

TCP Slow-Start…

- Instead of adding, double CWND every RTT (1,2,4,8,…)
- Start slow, but quickly reach high

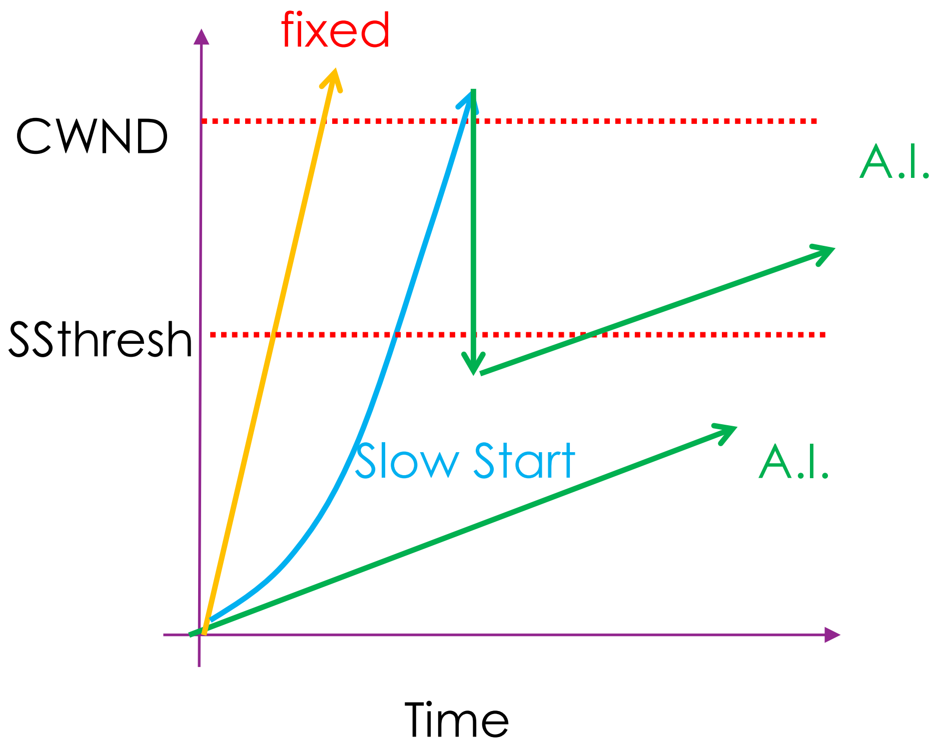


Slow-start overshoot

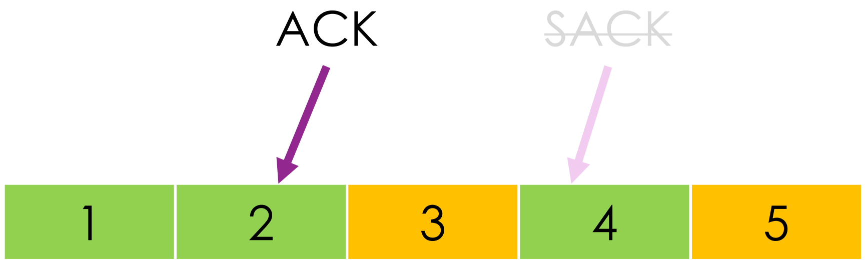
- Get to the right CWND more quickly
- But will still go (suddenly) over it
  - Get packet loss/feedback
  - Multiplicative decrease (big drop in CWND)
- So combine Slow-Start with Additive Increase:
  - Initial connection, get MD'd down. Below right CWND, but still close?
  - Define a threshold: ss-thresh = ½ * CWND(@loss)
  - Stop doubling, start adding

Combined behaviour…

- After the first overshoot…
- Start with slow-start
- Move to A.I. phase
- Gets you there quicker
- Keeps you there longer
  - Within that good Ssthresh/CWND band
  - Trying to maximise performance, politely.



Fast Retransmit



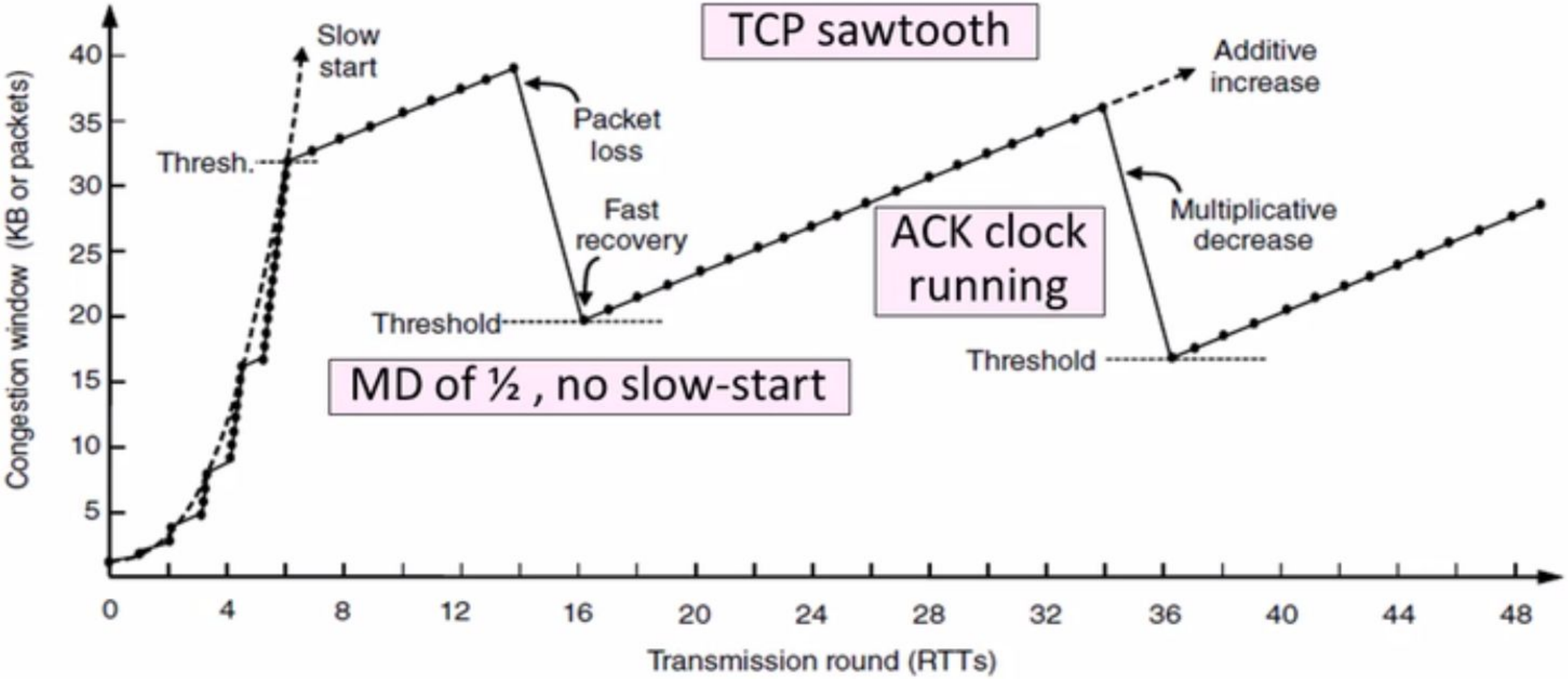
- Loss -> timeouts
- If timeout is too long, lose ACK clock
  - Start all over again, with a CWND's of packets out.
  - Slow start (CWND=1) then additive increase - ugh
- Recall ACKs (Seq#) are cumulative, sequential
  - If packet is lost, but later ones arrive, receiver sends a <u>duplicate ACK</u>
  - "New data arrived, but it wasn't the next segment"
    - Probably the next segment is lost
  - <u>Third duplicate ACK</u> triggers a resend of Seq#+1 (lost?) segment: **Fast Retransmit**
    - Hopefully repairs the single-segment loss quickly
    - And ACK Seq# catch up with what's been sent before loss?

Fast Recovery

- Had loss, so still need to multiplicative-decrease the CWND
- Also have to wait for receiver to tell you where it's Seq# is up to.
- Hang on:
  - Additional (duplicate) ACKs are arriving = receiver got more segments
    - Probably the next one(s)!
    - And they maintain the ACK clock
  - Take a chance: advance the sliding window as if everything is ok (count ACKs)
- MD the CWND (½ it!) and then <u>continue</u> sending **(Fast Recovery)**
  - Somewhat slower, but hopefully little loss, and no re-start.
- Receiver will sort things out and let you know (ACK)

TCP Reno

- Fairly common TCP codebase (1990s)



And beyond?

- TCP Reno
  - Can repair one loss per RTT
  - Multiple losses = timeout = (slow) start all over

- TCP NewReno
  - Better ACK analysis
  - Can repair multiple losses per RTT

- TCP SACK
  - Far better!
  - Receiver sends ACK ranges (set) – sender can retransmit without guessing

Can routers help?

- **Explicit Congestion Notification**
  - Still being deployed (routers and hosts) – only standardised in 2001…
- TCP drives network to congestion, then backs off
  - Prefer to detect congestion (well) before it happens
- Really simple, with in-band signalling
  1. Router notices queues getting full
  2. Marks packets in queue (ECN "congestion looming" – IP header)
  3. Forwards on to receiver
  4. Receiver marks TCP segments sent back to Sender (ACK or normal)
  5. Sender notices, and backs down (MD of CWND)
  6. No additional packets needed!

And more router help

- Haven’t yet discussed
  - Quality of Service, Differentiated Services
  - Traffic Shaping and Policing
  - Fair queuing
  - Rate and Delay guarantees
  - *Software Defined Networking*
  - And how they interact with routing and administrative domains
- And won’t!
- All “managing” packets randomly running through a network
  - Non-trivial…