Congestion Control

To do ...

- Principles and basic approaches
- TCP congestion control

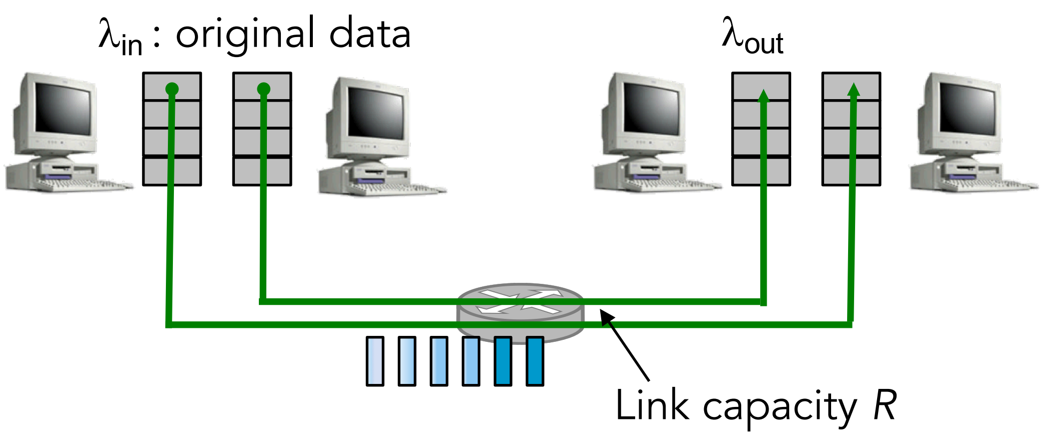
Northwestern

Congestion control

- Congestion is when the network is overloaded
  - Router queues are full, so packets are dropped
  - or length of queues leads to long queuing delays and timer to expire
- Dropped packets – inefficient and can compound the problem
  - Congestion → Packet loss → Retransmission → More congestion! → More loss! …
- *Goal:* to prevent this negative feedback cycle
- Difficult to solve because
  - Caused by many concurrent hosts at any of the hops in the path
  - State of routers along path is unknown – only see end-to-end behavior

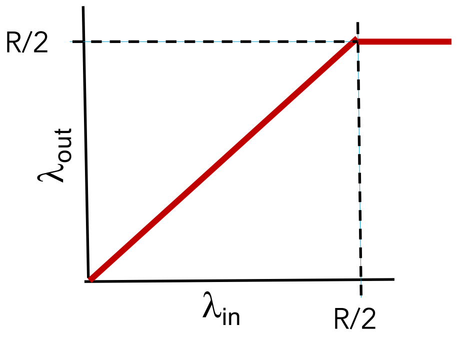
The problems with congestion

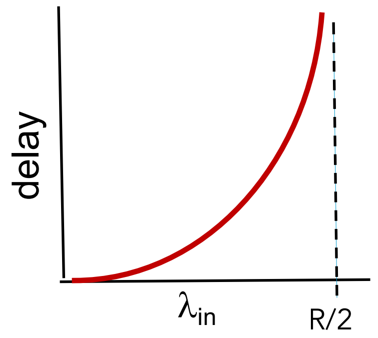
- Two hosts sharing a hop between between src and dest



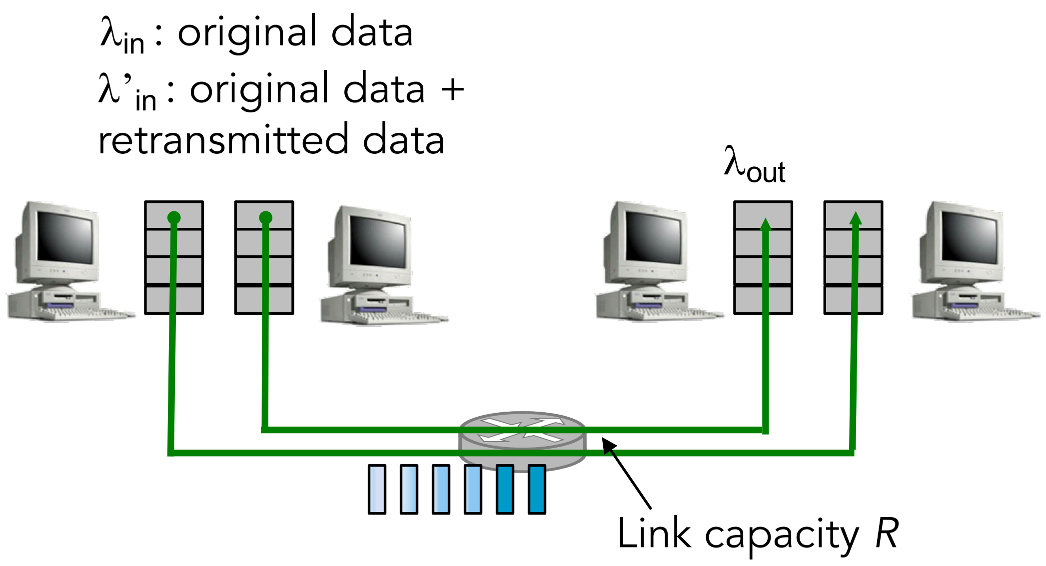
Assuming router has infinite capacity.

While packets will get to the other end, if sending at too high a rate (each >*R/2*) packets will be queued adding to end-to-end delay



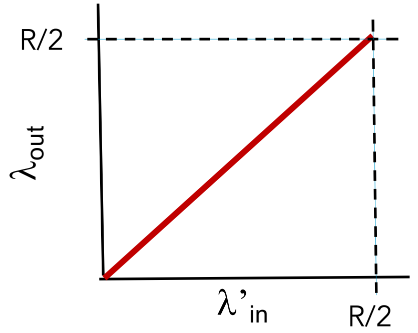


The problems with congestion

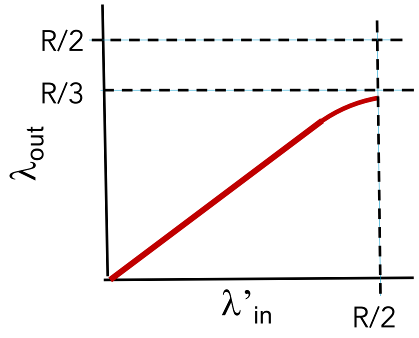


Now assume router has finite capacity

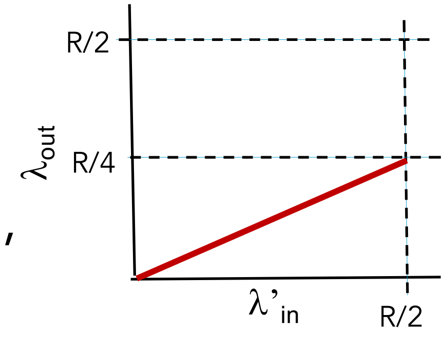
Src only send when there's room in the buffer, $\lambda_{in} = \lambda'_{in}$



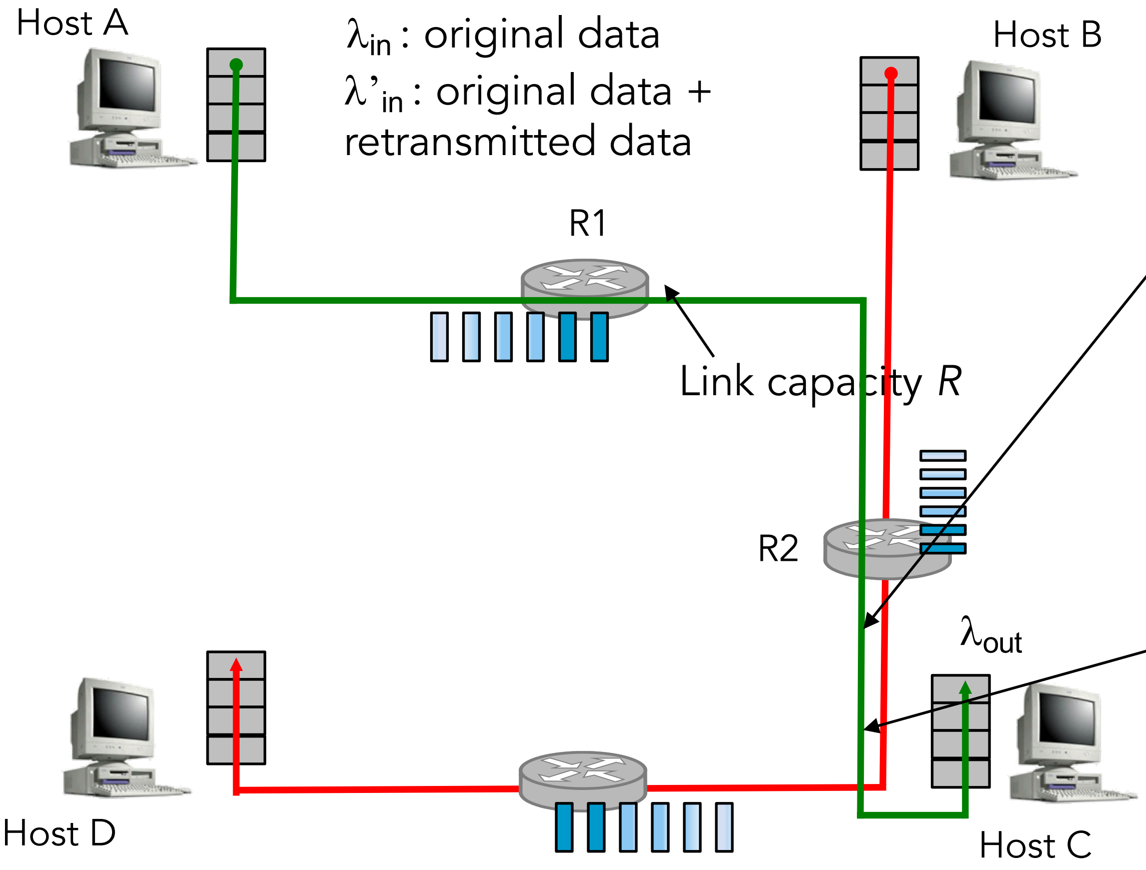
Src only re-sends a packet it knows it has been dropped (setting the *perfect* timeout)



Src timeouts prematurely and re-sends packet that have not been dropped (e.g., each packet is forwarded twice)



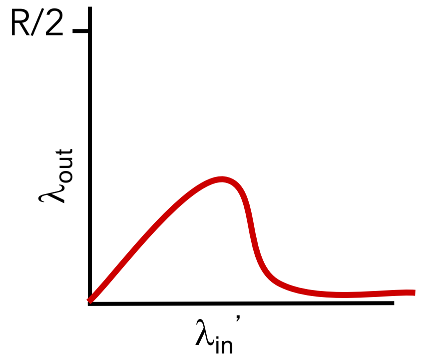
The problems with congestion



Traffic arrival rate from A-C to R2 is ≤ R, the capacity of the link from R1 to R2, regardless of λ_{in}

If l'$_{in}$ is very large for all connections, including B-D, the arrival rate at R2 << than that from A-C traffic

So A-C traffic that gets through R2 gets smaller as B-D traffic gets larger



So the transmission capacity we used in the upstream links ends up being wasted!

The problems with congestion – A summary

- Queueing will impact end-to-end latency
- Packet loss will trigger resend (ok) but you may resend packets that have *not* been lost
- And as you drop packets down the path, the transmission capacity used in the upstream links ends up being wasted

Approaches to congestion control

- Two broad approaches – does the network layer helps or not?
- End-to-end congestion control
  – Network layer provides no explicit support for congestion control
  – Even detection must be done on the end systems (e.g., TCP’s segment loss or, more recently, increasing round-trip segment delay)
- Network-assisted congestion control
  – Routers provide explicit feedback to sender and/or receiver
    - A simple bit as in DEC DECnet or IBM SNA
    - More details like max sending rate the router can support (ATM Available Bit Rate)
  – From router to the sender – like a ”choke” packet saying it’s congested
  – Router marks a filed in a packet flowing through towards the receiver, receiver then notifies sender

TCP congestion control

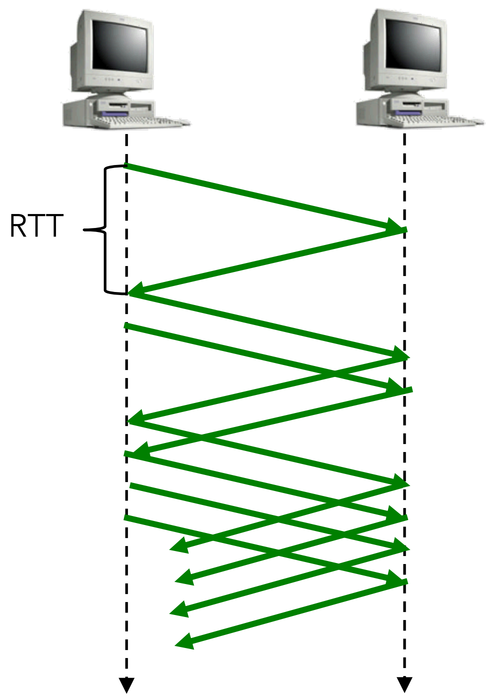
- End-to-end, each sender limits that rate at which sends traffic into the network based on hints about congestion
  - There are some variations using ECN (e.g., DCTCP for data centers)
- How does a sender limit its sending rate?
  - Sender keeps track of its *congestion window*, `cwnd`
  - `LastByteSent - LastByteAcked` ≤ `min(cwnd, rwnd)` (amount of unack'ed data)
  - Sending rate ~ cwnd/RTT bytes/sec
- How does it perceive there's congestion?
  - Lost packet, either a timeout or three duplicate ACKs
- What algorithm should it use to adjust sending rate?
  - …

TCP guiding principles for congestion

- Adjusting sending rate
  - Lost segment implies congestion, decrease rate
  - ACK means a delivered segment, increase rate
  - Self-clocking – fast/slow arriving ACK, fast/slow growing `cwnd`
- Bandwidth probing
  - Increase rate in response to ACKs until
    - A loss occurs, decrease transmission rate
  - And try again in cased it has changed
- TCP congestion control operates in three phases
  - Slow start
  - Congestion avoidance
  - Fast recovery (recommended but not required)

TCP congestion control – Slow start

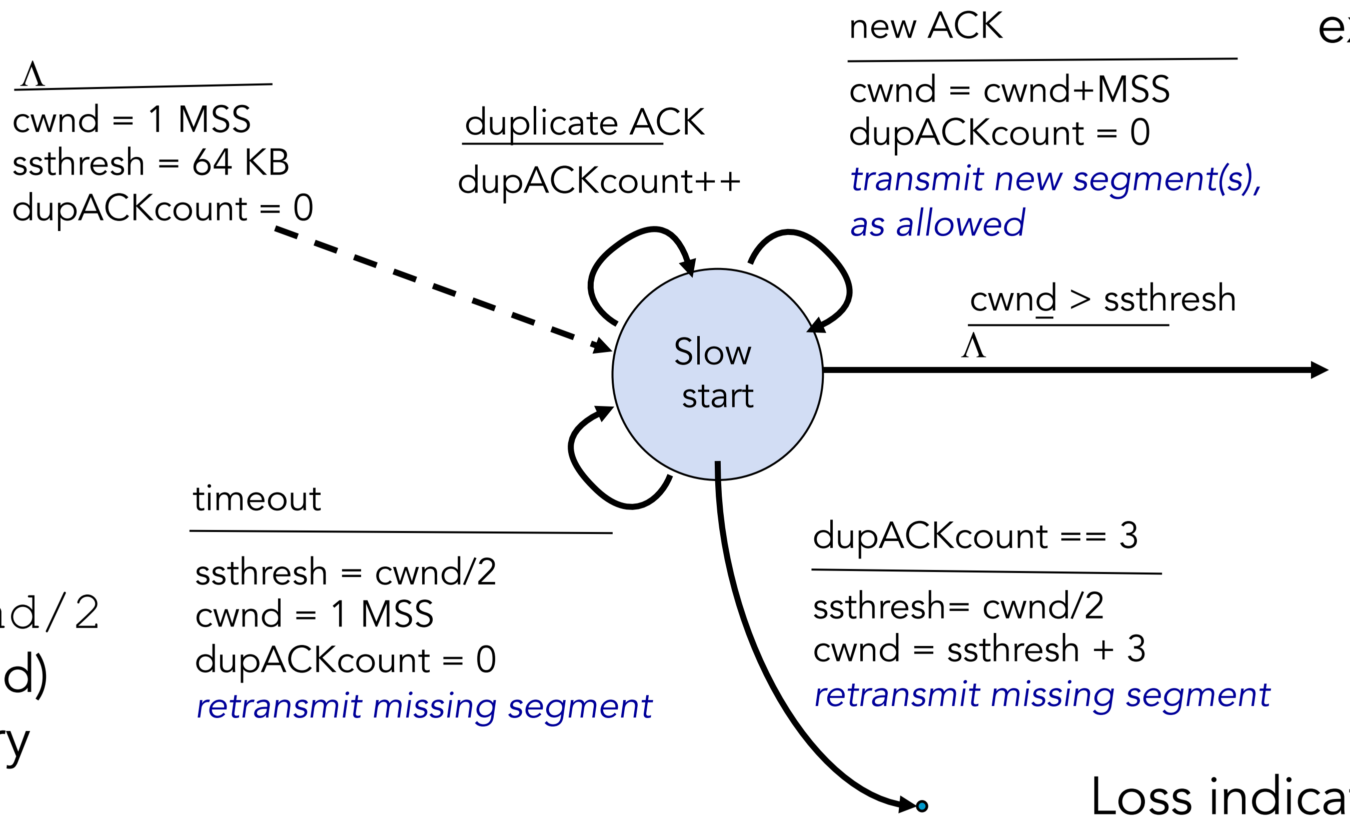
- Initially, net capacity is unknown, start with `cwnd= 1 MSS,` so a sending rate of `MSS/RTT`
- Increase by 1 MSS every time it gets an ACK, so doubling sending rate every RTT – exponential growth
- When does it end?
  - Loss indicated by timeout → `ssthresh = cwnd/2` (slow start threshold) and `cwnd = 1,` try again
  - If `cwnd` ≥ `ssthresh` (previous congestion), avoid congestion by slowing down increasing rate
  - Loss indicated by 3 duplicate ACKs, make a fast retransmit and move to fast recovery



TCP congestion control – Slow start

Initially, net capacity is unknown, so

Increase by 1 MSS every time it gets an ACK, so doubling sending rate every RTT – exponential growth



If `cwnd ≥ ssthresh` (previous congestion), move to *Congestion Avoidance*, risky increasing at the same rate

Loss indicated by timeout → `ssthresh = cwnd/2` (slow start threshold) and `cwnd = 1`, try again

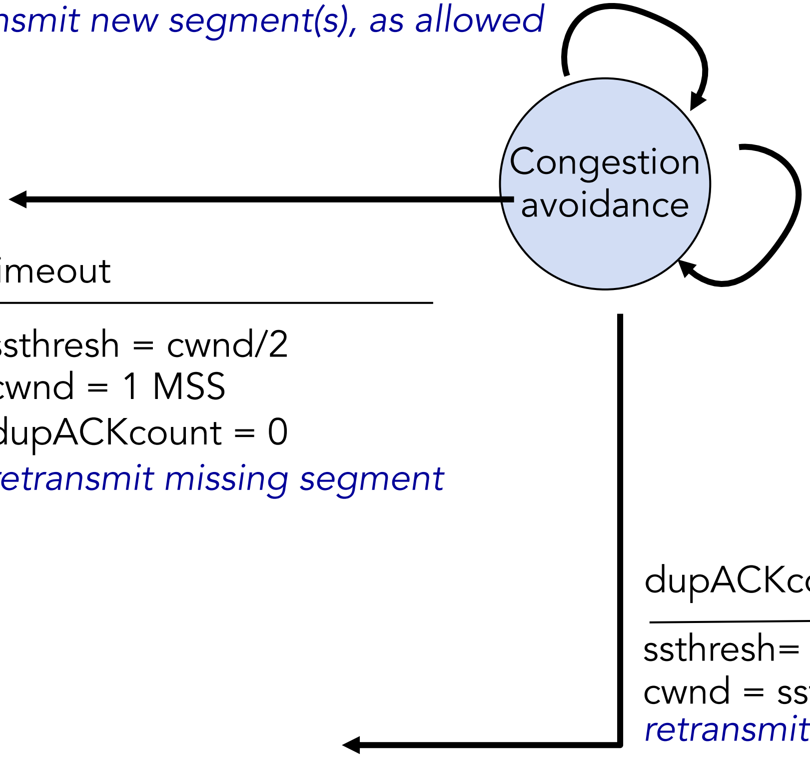
Loss indicated by 3 duplicate ACKs, make a fast retransmit and move to *Fast Recovery*

TCP congestion control – Congestion avoidance

Linear growth now to avoid congestion

new ACK

cwnd = cwnd + MSS * (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed



duplicate ACK

dupACKcount++

timeout

ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment

Serious issue, go back to Slow Start

dupACKcount == 3

ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment

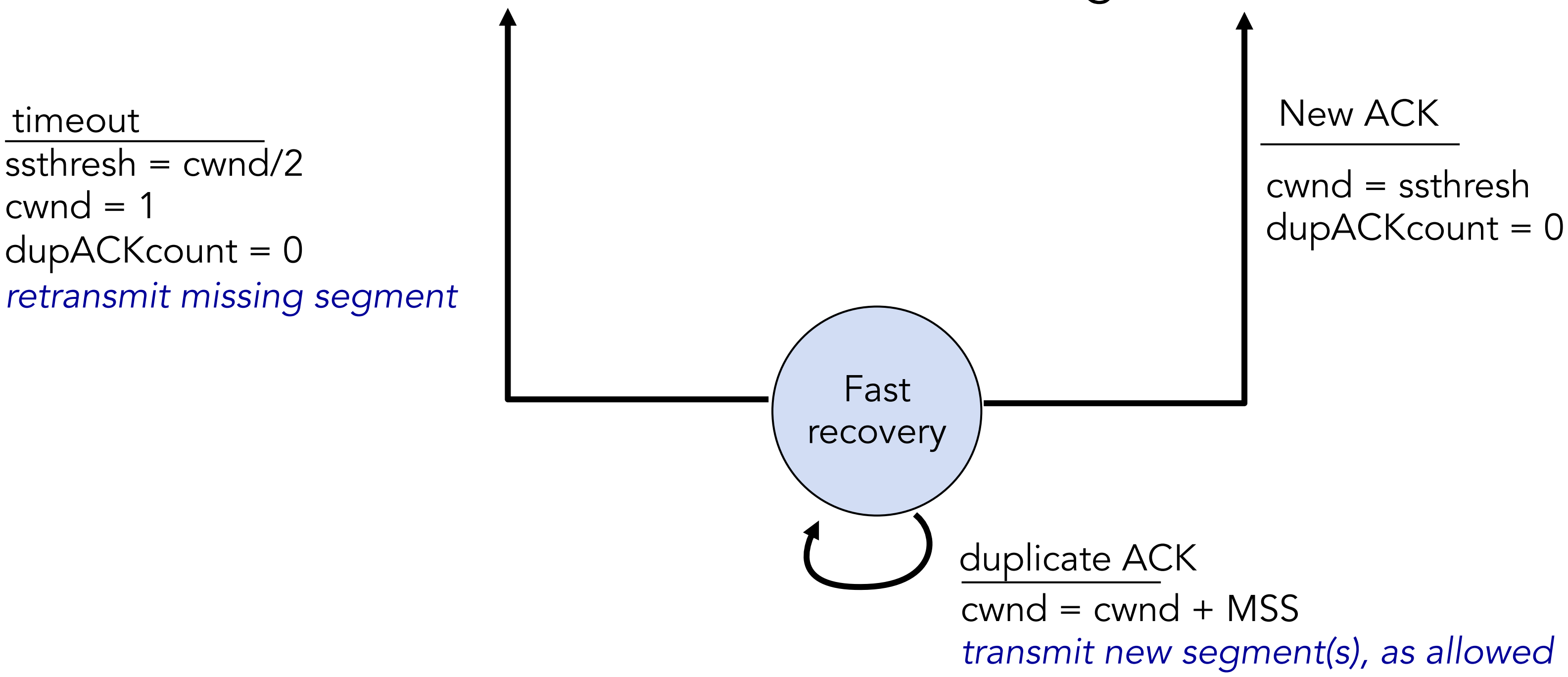
On loss by triple-duplicate ACK, Fast Retransmit

TCP congestion control – Fast recovery

The network is delivering segments (that's how we get duplicates), no need for drastic measures

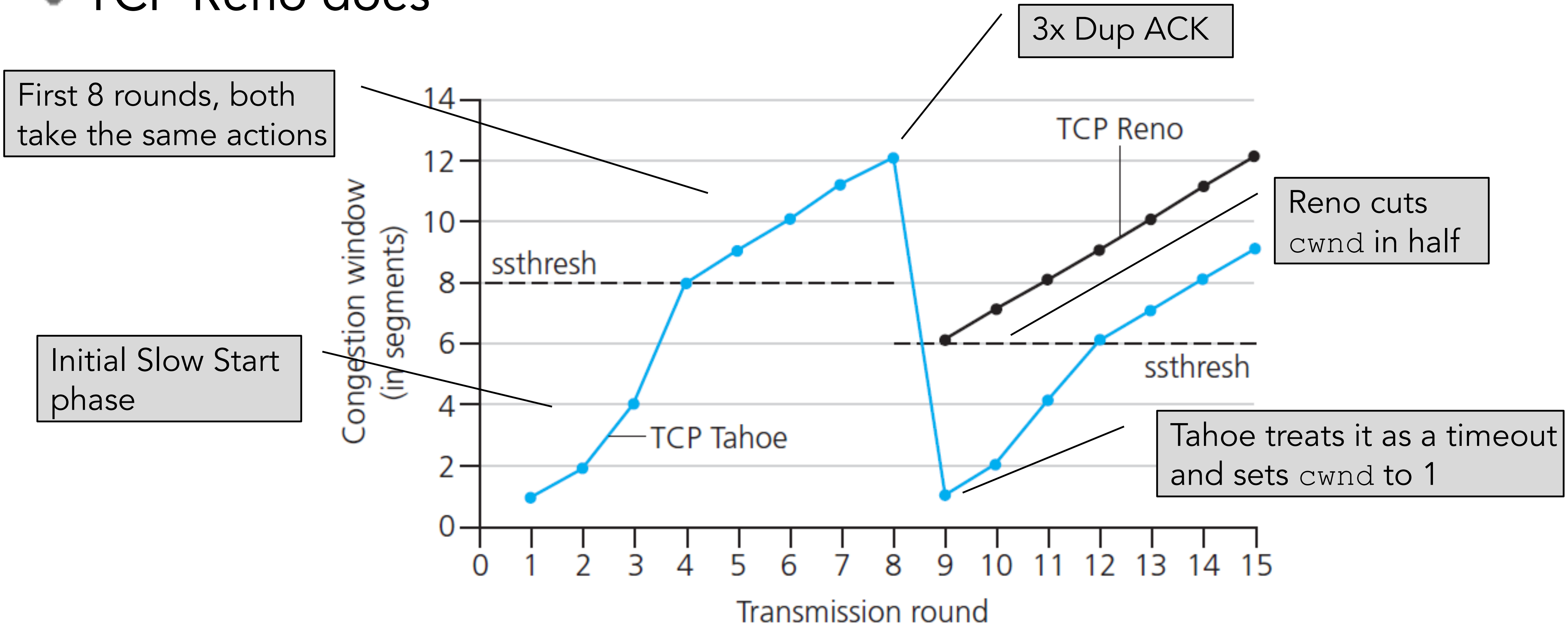
If it doesn't before timeout, a more serious issue, Slow Start

Eventually the packet arrives (got ACK), so move to Congestion Avoidance

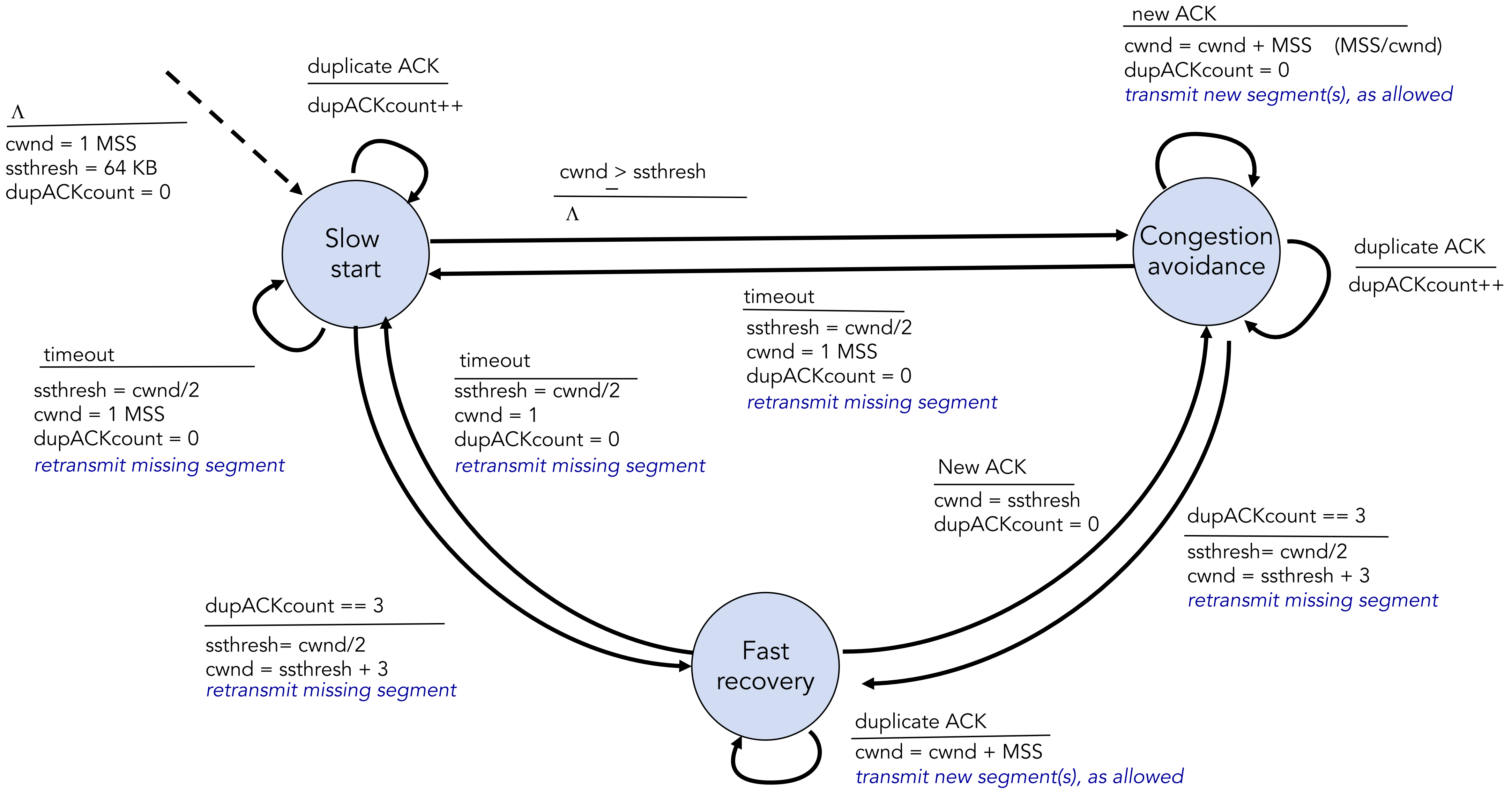


TCP Tahoe, TCP Reno and Fast Recovery

- TCP Tahoe, earlier version, doesn't include Fast Recovery
- TCP Reno does

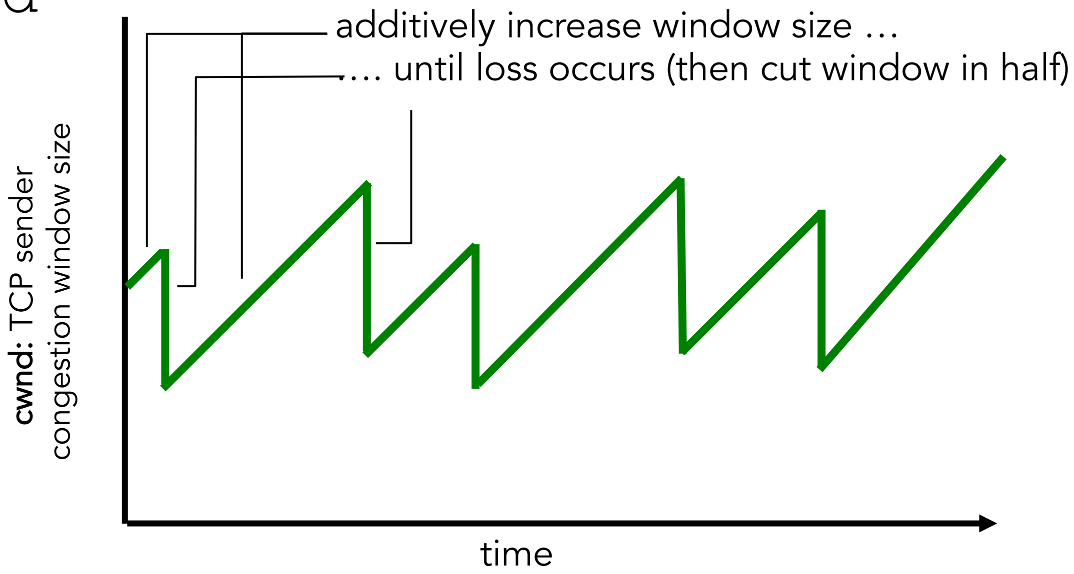


All congestion control



TCP congestion control overview

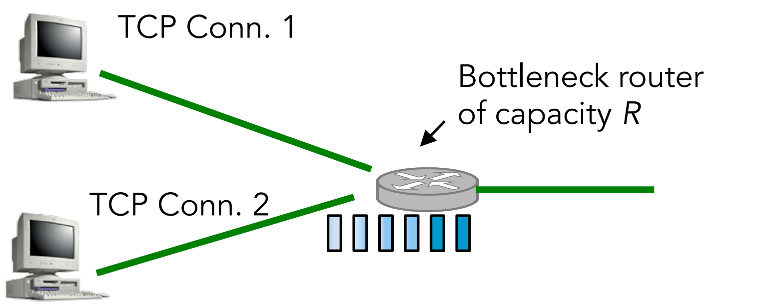
- Ignoring initial slow start period and assuming losses are indicated by triple duplicate ACKs instead of timeouts
  – Additive, linear increase of 1 MSS per RTT
  – Multiplicative decrease, halving of `cwnd`
  – So, the avg throughput ~0.75 * Max since it halves when it reaches it (Max/2*RTT and Max/RTT)
- Ongoing work, can you predict loss using RTT? How do you work with high-bandwidth or high RTT paths? …

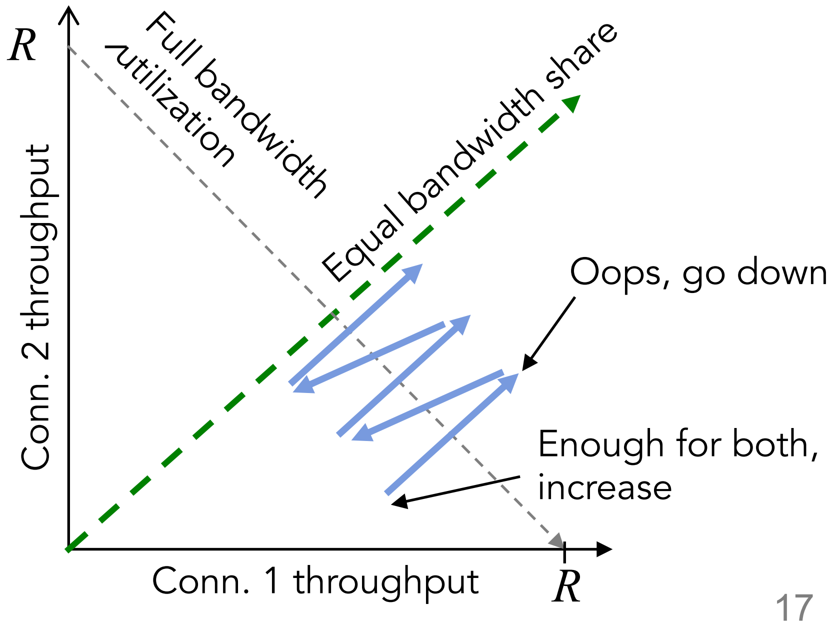


AIMD saw tooth behavior: probing for bandwidth

TCP fairness

- A congestion control algorithm is far if each connection gets equal share of the link bandwidth
- Is TCP AIMD fair? An intuitive argument
  - 2 TCP connections sharing a link, same MSS and RTT, with data to send, no other connection
- In congestion avoidance, bandwidth of both grows at same rate, moving at ~45° angle up-right
- Assuming only TCP connections traverse the bottleneck link, all have same RTT, …
  - Sessions with smaller RTT can grab bandwidth faster, so get better throughput





Nagle's algorithm merges small packets

- An app may write a series of small message to a TCP stream
  - E.g., `write("OK\n"); write("READY\n"); write("GO\n");`
- A simple implementation of TCP would send segments for each, with high overhead from the 40B of TCP packet header
  - Merging small packets into one larger one would reduce network load

$$(40+3) + (40+6) + (40+3) \rightarrow (40+12) : 132 \rightarrow 52 \text{ bytes}$$

- Wait until segment is full before sending, *unless* there are no un-ACK'ed segments outstanding (eg., send first segment immediately)

Interactive applications

- Interactive apps and bulk-transfer apps prefer different TCP behavior
- Socket options give applications some control of the underlying TCP:
  - TCP_NODELAY socket option disables Nagle's algorithm
  - Every write → segment(s) being sent immediately (if allowed by window)
  - Nagle's algorithm adds extra latency which may hurt performance of applications that send small, time-sensitive data. (eg., GUI events)
- TCP_NOPUSH is even more aggressive than standard Nagle
  - Wait until send buffer is full before sending segment(s)
  - Also, don't set PSH bit (to maximize buffering on the receiver's side as well)
- Usually the PSH bit will be set on the last segment in a write call
  - PSH tells the receiving TCP implementation to alert the receiving process that that data is ready

TCP Keepalive

- An idle TCP connection involves no data exchange
- Optionally, a TCP host may occasionally send an empty data segment, called a keepalive message, just to test whether an ACK will return
  - Keepalive has SEQ # one less than expected, to trigger an ACK response
  - Low frequency, ~once per minute
- Disabled by default, only used in special situations
  - SSH clients give the option to enable TCP keepalives
  - This forces NAT routers to keep the port mapping alive
- Some application-level protocols have their own keepalive msgs

Recap

- Congestion control can mean higher latencies, lower throughput and wasted effort
- TCP congestion control is done using a dynamic congestion window, controlled by heuristics that operate in phases
  - Slow start – exponential growth to find approximate network capacity
  - Congestion avoidance – as you get closer … linear growth, slowly trying to increase throughput
  - Fast recovery – If one packet is lost, resend and cut window in half
- Adapts to changing network conditions