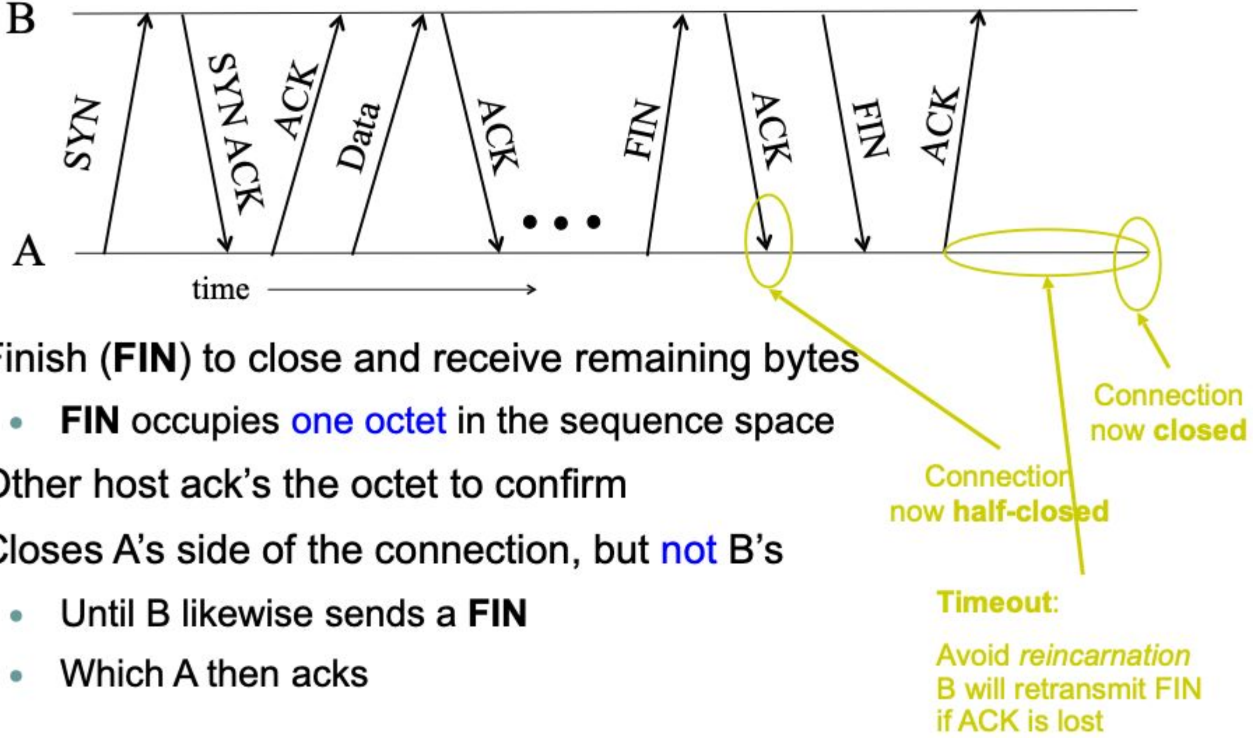
TCP Connection Teardown

Normal Termination, One Side At A Time



Finish (**FIN**) to close and receive remaining bytes

- **FIN** occupies one octet in the sequence space

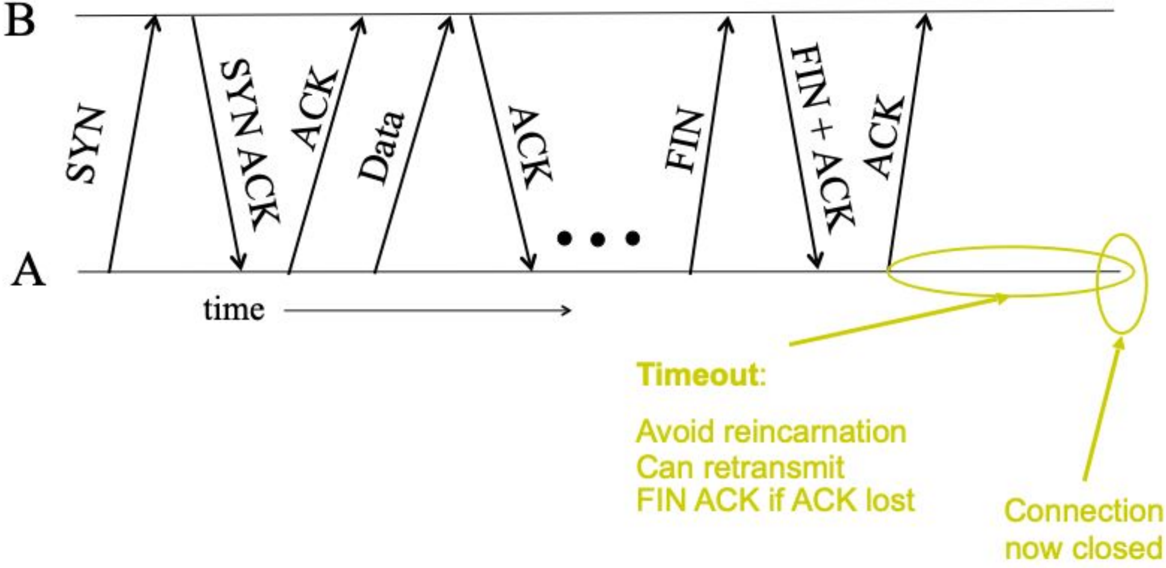
Other host ack's the octet to confirm

Closes A's side of the connection, but not B's

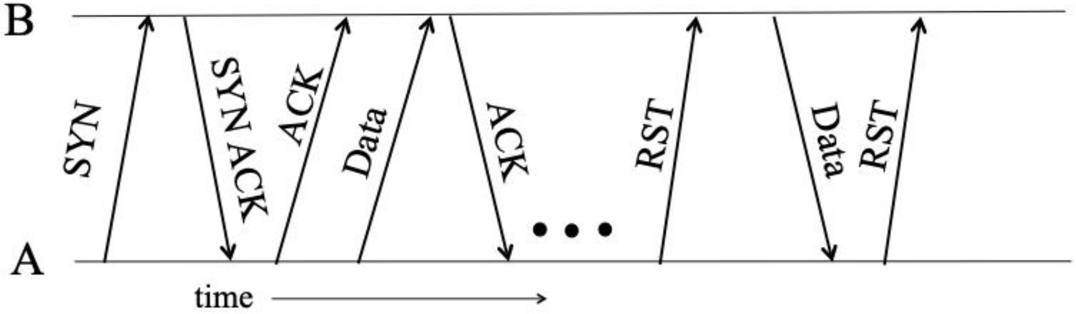
- Until B likewise sends a **FIN**
- Which A then acks

Normal Termination, Both Together

Same as before, but B sets **FIN** with their ack of A's **FIN**



Abrupt Termination



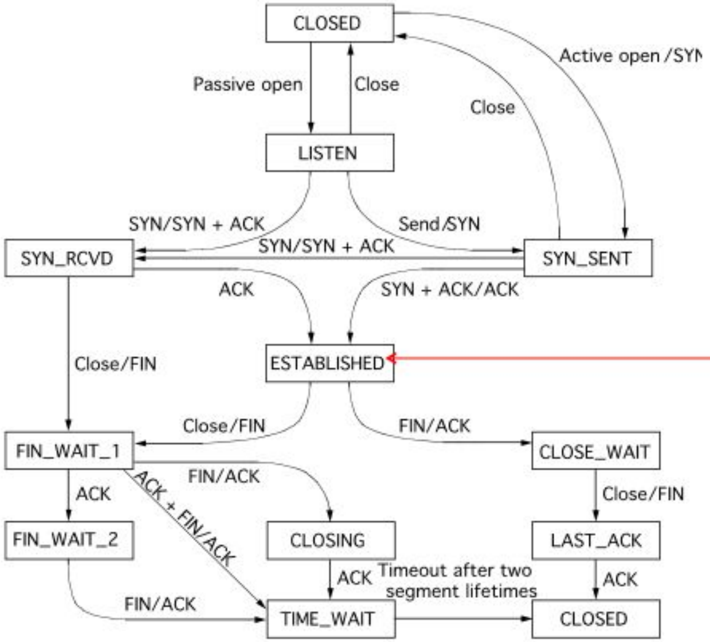
A sends a RESET (**RST**) to B

- E.g., because app. process on A crashed

That's it

- B does not ack the **RST**
- Thus, **RST** is not delivered reliably
- And: any data in flight is lost
- But: if B sends anything more, will elicit another **RST**

TCP State Transitions



Data, ACK exchanges are in here

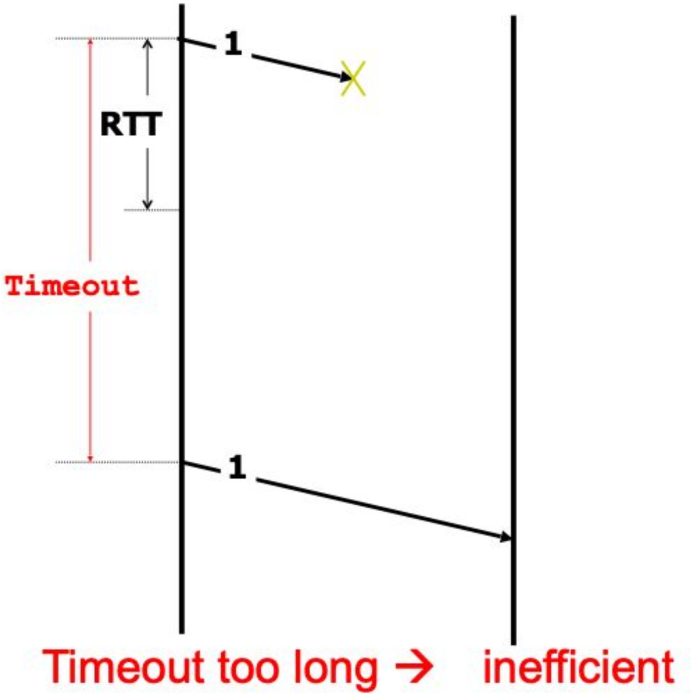
Reliability: TCP Retransmissions

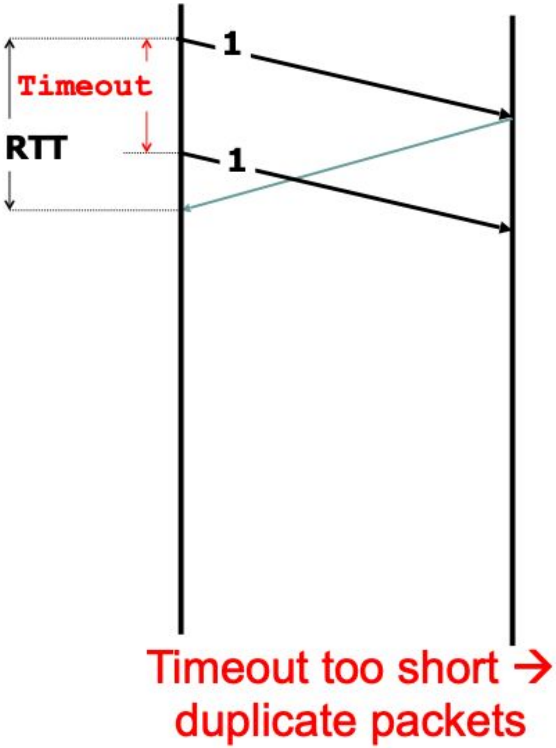
- Reliability requires retransmitting lost data
- Involves setting timer and retransmitting on timeout
- TCP resets timer whenever new data is ACKed
    - Retx of packet containing “next byte” when timer goes off

Example

- Arriving ACK expects 100
- Sender sends packets 100, 200, 300, 400, 500
  - Timer set for 100
- Arriving ACK expects 300
  - Timer set for 300
- Timer goes off
  - Packet 300 is resent
- Arriving ACK expects 600
  - Packet 600 sent
  - Timer set for 600

Setting the Timeout Value





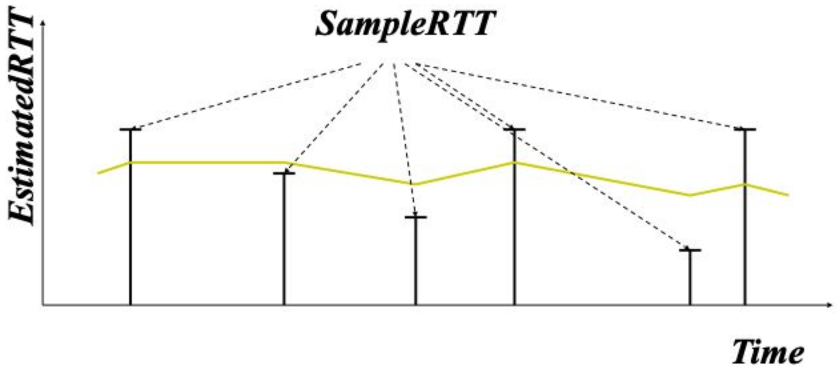
RTT Estimation

Use exponential averaging of RTT samples

$$SampleRTT = AckRcvdTime - SendPacketTime$$

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT$$

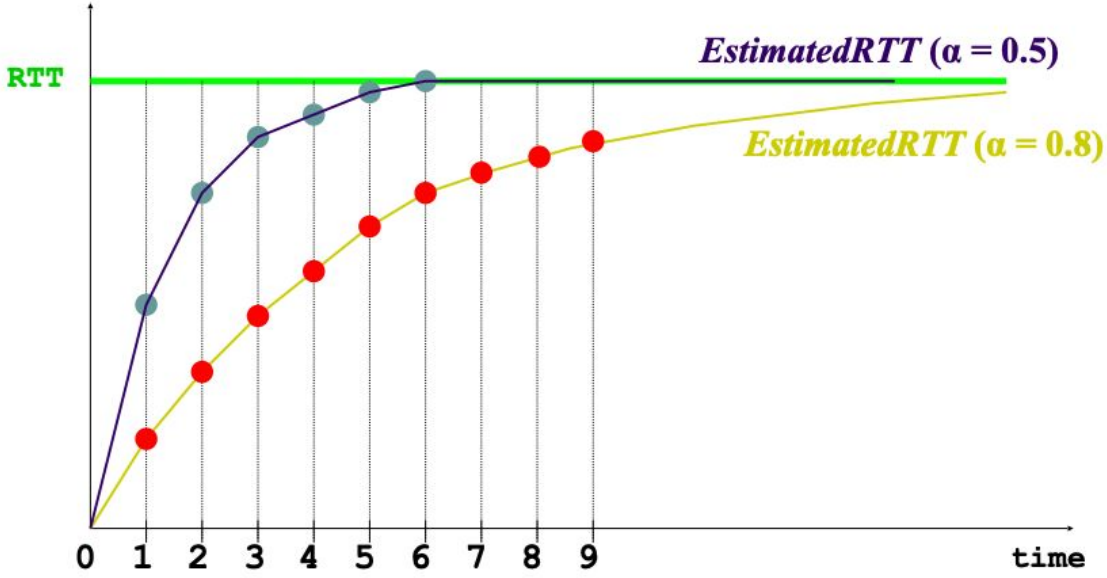
$$0 < \alpha \leq 1$$



Exponential Averaging Example

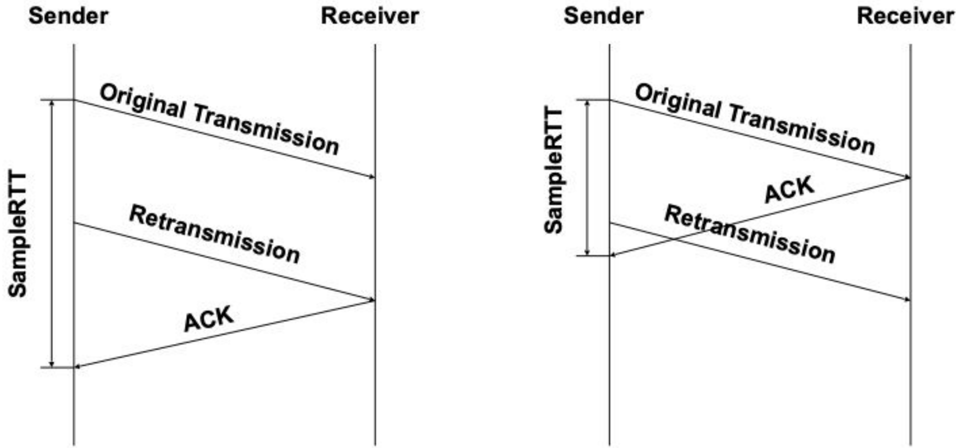
$$EstimatedRTT = \alpha * EstimatedRTT + (1 - \alpha) * SampleRTT$$

Assume RTT is constant → $SampleRTT$ = RTT



Problem: Ambiguous Measurements

How do we differentiate between the real ACK, and ACK of the retransmitted packet?



Karn/Partridge Algorithm

- Measure SampleRTT only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
  - Computes EstimatedRTT using $\alpha = 0.875$
- Timeout value (RTO) = 2 × EstimatedRTT
- Use exponential backoff for repeated retransmissions
  - Every time RTO timer expires, set RTO $\leftarrow$ 2·RTO
    - (Up to maximum ≥ 60 sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to 2 × EstimatedRTT

Reality

- Implementations often use a coarse-grained timer
  - 500 msec is typical

- So what?
  - Above algorithms are largely irrelevant
  - Incurring a timeout is expensive

- So we rely on duplicate ACKs

Loss with Cumulative ACKs

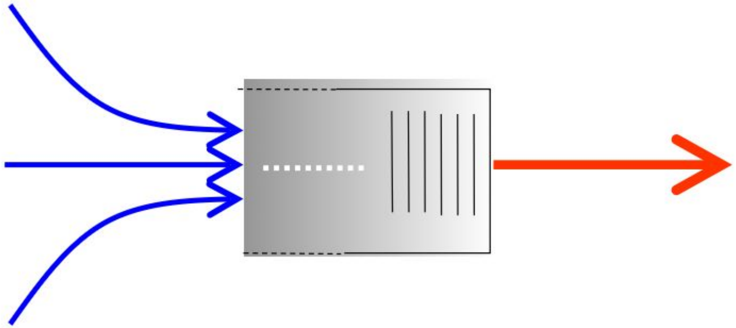
- Sender sends packets with 100B and seqnos.:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, …
- Assume the fifth packet (seqno 500) is lost, but no others
- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500,…

Loss with Cumulative ACKs

- “Duplicate ACKs” are a sign of an isolated loss
  - The lack of ACK progress means 500 hasn’t been delivered
  - Stream of ACKs means some packets are being delivered
- Therefore, could trigger resend upon receiving k duplicate ACKs
  - TCP uses k=3

Congestion Control

Because of traffic burstiness and lack of BW reservation, congestion is inevitable



If many packets arrive within
a short period of time
the node cannot keep up anymore

Congestion is not a new problem

- The Internet almost died of congestion in 1986
  - throughput collapsed from 32 Kbps to... 40 bps
- Van Jacobson saved us with Congestion Control
  - his solution went right into BSD
- Recent resurgence of research interest after brief lag
  - new methods (ML), context (Data centers), requirements

Congestion is not a new problem

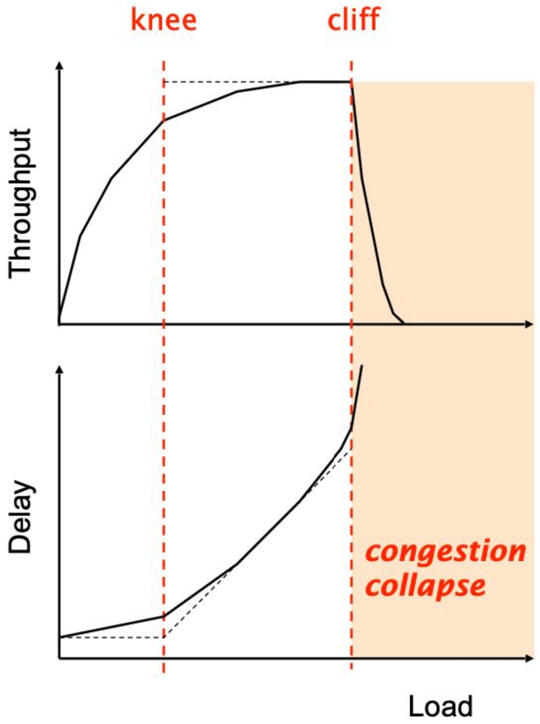
- The Internet almost died of congestion in 1986
  - throughput collapsed from 32 Kbps to... 40 bps
- Van Jacobson saved us with Congestion Control
  - his solution went right into BSD
- Recent resurgence of research interest after brief lag
  - new methods (ML), context (Data centers), requirements

Congestion is not a new problem

original behavior	On connection, nodes send full window of packets
	Upon timer expiration, retransmit packet immediately
meaning	sending rate only limited by flow control
net effect	window-sized burst of packets

Congestion collapse

Knee	point after which	
throughput	increases	slowly
delay	increases	quickly
Cliff	point after which	
throughput	decreases	quickly
delay	tends to	infinity



Congestion collapse