49202 Communication Protocols

The Transport Layer - Part II

Daniel R. Franklin

Faculty of Engineering & IT

March 21, 2023

Review: TCP service model

- Connection oriented
  - Connection is established before end hosts exchange data
  - Achieved by 3-way handshake
- Byte-stream: byte oriented rather than message oriented
- Reliability: Guarantees data delivery and delivery order
- Flow control: Do not overload receiver
- Congestion control: Do not overload network

This week's lecture

- TCP
  - Problems have occurred as Internet grew
  - Various solutions - ongoing research
- UDP
  - Design and basics of operation
  - Interaction between UDP and TCP
- Problems with TCP Congestion Control

Congestion: causes, effects

- Congestion occurs in a link when the offered load to the link is *persistently* higher than the link capacity
- Results of congestion include:
  - Increase in packet delay (as queues form)
  - *Sustained* packet loss (as queues fill to capacity)
- Congestion can be removed by
  - Sources reducing load - short time scale
  - Increasing link capacity - long time scale
- Congestion **cannot** be removed by increasing router buffer size
  - This will only delay the inevitable, while massively increasing latency (delay) - potentially to ridiculous levels
  - This behaviour is surprisingly common - especially in home routers - and is called **bufferbloat**

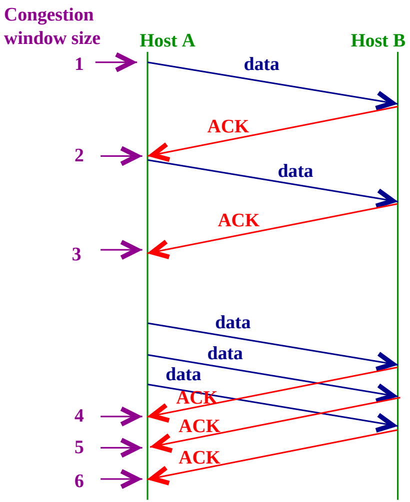
TCP congestion control

- Recall: TCP sources adjust their sending rates to avoid congesting the network according to the AIMD rule:
  - If there is **no** congestion, **linearly** increase the sending rate
  - If there is congestion, **exponentially** decrease the sending rate
- TCP uses timeout as ONE indication of congestion
- TCP maintains a variable called `CongestionWindow` - how much **unacknowledged** data may be in transit at once
- Congestion avoidance is **entirely** the responsibility of the sender, and is independently performed in each direction.

TCP slow start (or is it?)

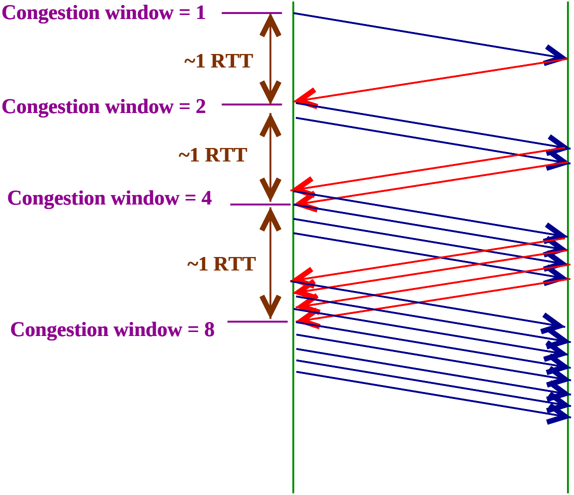
- Linear increase takes a long time to increase the congestion window from its starting value to the working range
- So, we use a different behaviour called `slow start` at the beginning of a TCP stream (after connection establishment):
  - Increment `CongestionWindow` by 1 every time an ACK is received rather than by 1 every RTT
  - For example, if our window is 4, we send 4 segments
  - If there are no losses, we will receive 4 ACKs after one RTT, and so our window increases by 4 segments to 8.
  - In the next RTT, it will double again to 16
- If the sender *always* has data to send, *and there is no packet loss*, `CongestionWindow` **doubles** every RTT

Slow start - example 1



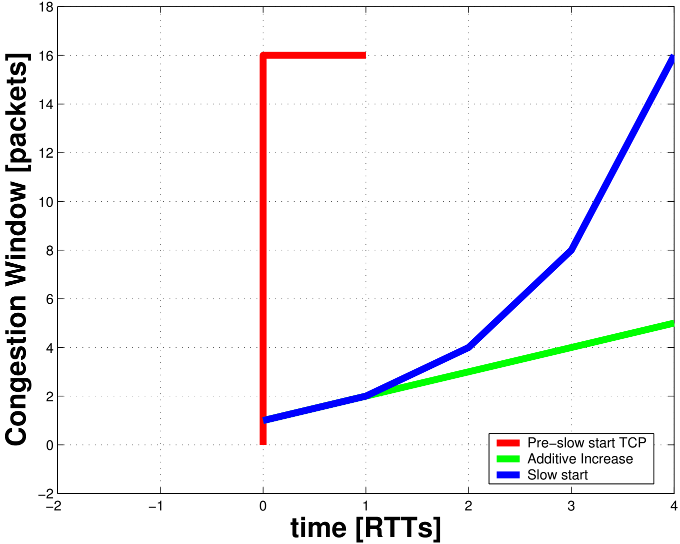
- In this example, the sender initially doesn't have enough data ready to send to 'use up' the entire available `CongestionWindow`.
- However, it still increases on every ACK
- When we do have something to send, we may now transmit a burst of up to 3 segments.

Slow start - example 2



- In this example, we always have data ready to send
- Slow start can continue until we are transmitting continuously

Slow Start vs. Additive Increase



- The congestion window increases exponentially - compare with additive!
- Early TCP just jumped straight to `AdvertisedWindow` - bad idea!
- This behaviour caused the early Internet to undergo **congestion collapse**

When is slow start used?

- Slow start is used in two situations
  - The beginning of a TCP connection, when we don't know anything about the level of congestion in the network
  - If a timeout occurs when the connection goes dead - which implies that something in the network has **changed**
- In both cases, we should not make *any* assumptions about the current state of the network.

At beginning of connection

- Aim: Try to find a suitable value for `CongestionWindow`
- TCP maintains **another variable** called `CongestionThreshold`
- Slow start operates until we reach the threshold; then it switches to AIMD:

  If CongestionWindow $<$ CongestionThreshold

      Increase CongestionWindow by 1 per ACK received

  else

      Increase CongestionWindow by 1 every RTT

- `CongestionThreshold` is initialised to 64k at the beginning of a TCP connection

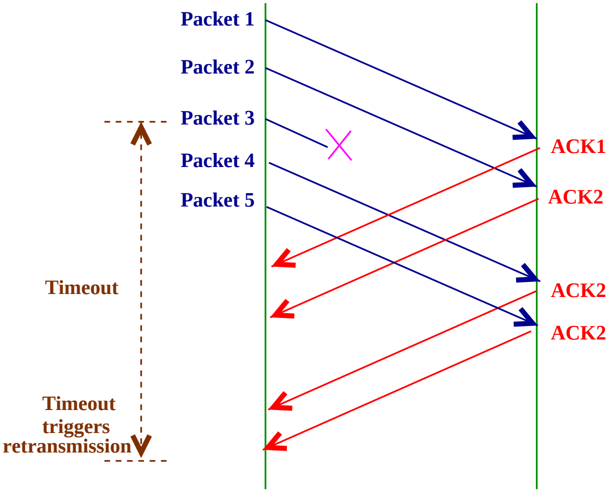
Reaction to timeout

- When a segment fails to be acknowledged before the timeout expires, slow start is triggered again
- However, timeouts indicate congestion. So now we know we are transmitting too much - we have learned something about the congestion conditions in the network.
- The threshold is now reduced by one half:
  - Set `CongestionThreshold` = $\frac{1}{2}$`CongestionWindow` at timeout
  - Set `CongestionWindow` back to 1
  - Continue with slow start up to the new threshold, then switch to AIMD
- Rationale
  - Use the `CongestionWindow` before timeout as a guide

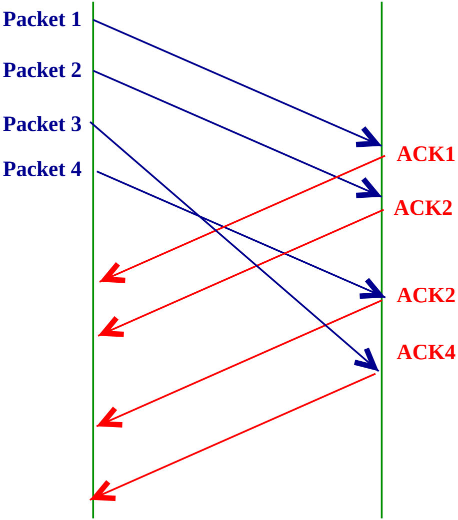
Duplicate ACKs

- With slow start / congestion avoidance, we can still have some problems:
  - Long periods of timeout
  - Inefficiency in cases where only one out of a number of packets is lost
- The sender receives duplicate acknowledgements in this case - remember, the ACK number is the sequence number of the *next byte that we want*
- So if one byte in the middle of a sequence is missing, the receiver sends back multiple ACKs asking for the missing segment each time another segment arrives
- Thus duplicate ACKs can be used by the sender as an indicator of congestion

Fast retransmit - motivation



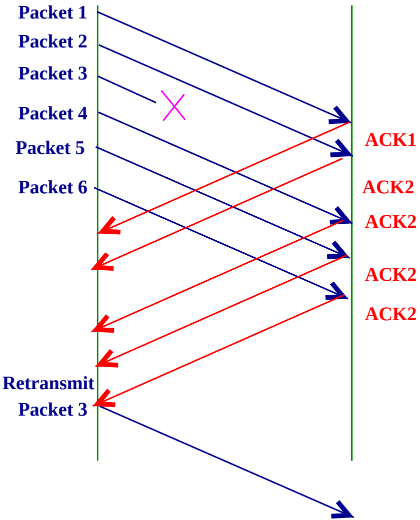
Out of order delivery



Fast Retransmit

- Duplicate ACKs can be caused by
  1. One out of a number of packets being lost
  2. Out-of-order delivery
- Fast transmit
  - TCP retransmit without waiting for timeout if it receives 3 duplicate ACKs

Fast retransmit - an illustration



TCP Tahoe

- The TCP algorithm as described up to this point is known as TCP Tahoe (AIMD + Slow Start + Fast Retransmit)
- It was first introduced in 1988, and successfully prevented further instances of Congestion Collapse.
- However, further improvements to robustness and efficiency were to come...

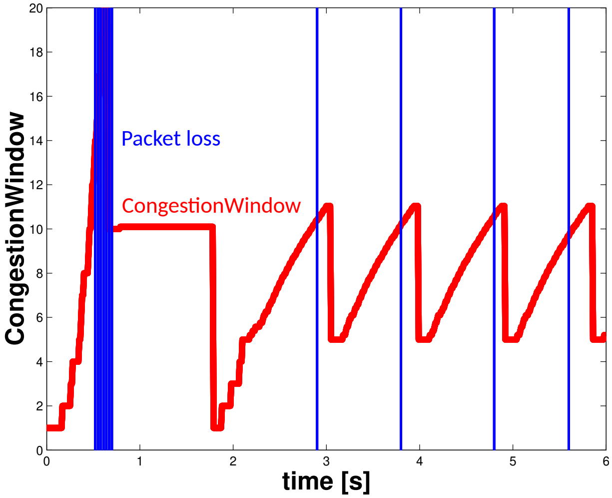
Fast recovery

- Previously, slow start follows fast retransmit
- With fast recovery,
  - After fast retransmitting the packet
  - Set `CongestionWindow` equals to half of its previous value (Multiplicative decrease)

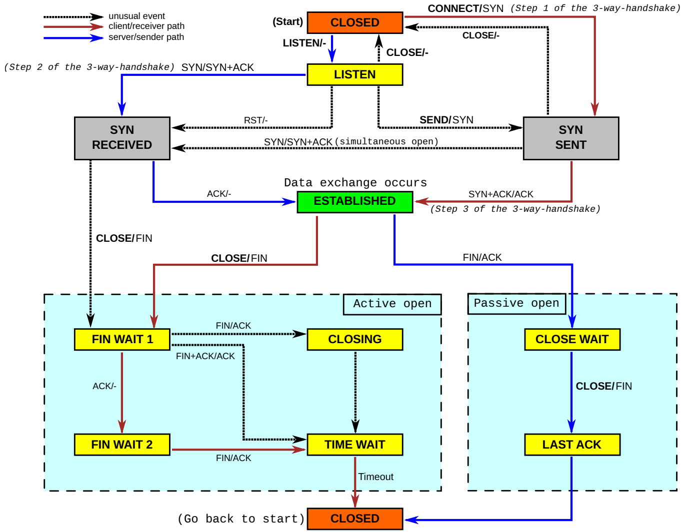
TCP Reno

- TCP Reno is TCP Tahoe (AIMD + Slow Start + Fast Retransmit) + Fast Recovery
- Reno-based TCP implementations are a kind of reference implementation against which other TCP congestion control algorithms are compared
- Further enhancements to Reno include selective acknowledgements (SACK) which reduces the number of ACKs which need to be sent
- In 2016, Google introduced an algorithm which measures **Bottleneck Bandwidth and Round-trip propagation time** - hence the name **TCP-BBR**
- It tries to avoid congestion-induced packet losses by monitoring dynamic behaviour of latency in the network.
- It performs exceptionally well when there is some connection-induced packet loss in the network.

Example Reno trace



Full TCP state transition diagram



- Left: server; Right: client
- This diagram illustrates the full connection lifecycle from both perspectives

User Datagram Protocol (UDP)

- UDP is the other transport layer protocol in the TCP/IP suite
- Features (or lack thereof...):
  - Connectionless - no setup / teardown overhead or delay
  - Provides application multiplexing via ports (independently of TCP)
  - Optional checksum
  - Message-oriented rather than stream-oriented
  - Delivery is **not** guaranteed, i.e. like IP, UDP is **unreliable**
  - Does not enforce in-order delivery
  - No flow control
  - No congestion control
- UDP is designed for **simplicity** and **efficiency.**
- If the application cares about reliability, it may add its own error/flow/congestion control mechanisms on top.

UDP header

- The optional checksum is computed on the payload and a pseudo header (UDP header + IP source/destination addresses, IP protocol field, IP total payload length field)
- The payload must be a multiple of 16 bytes - so there may be some padding.
- Length field is redundant if UDP is used on top of IP.

0 16 31

Source Port	**Destination Port**
Length	**Checksum**

Applications using UDP

- DNS (Domain Name System)
- Real-time applications
  - Two-way video/voice over IP (VoIP) - Skype, Zoom etc.
  - Online first-person games (e.g. to update your position every few tens of milliseconds)
- Features of real-time applications
  - Occasional loss of a packet is acceptable
  - Retransmission is detrimental to applications
  - Loss-tolerant, delay-sensitive!

Interaction between UDP &TCP (1)

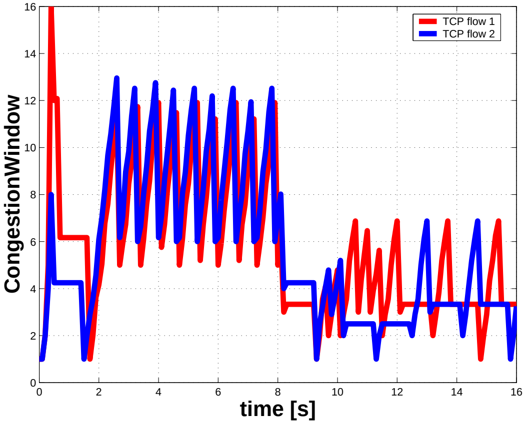
- What happens when TCP and UDP share a network link?
- The conflict:
  - UDP does *not*, by itself, perform congestion control even when packets are lost
  - TCP performs congestion control
- So what happens when there is congestion?
  - TCP slows down, UDP keeps going even if there is quite severe packet loss
  - TCP loses share of bandwidth

Interaction between UDP & TCP (2)

- Example: 2 UDP flows and 2 TCP flows share a link with capacity 1 Mbps
  - Case 1: Both UDP flows are sending at 0.3 Mbps
    - Both UDP flows get about 0.3 Mbps
    - The two TCP flows share the remaining 0.4 Mbps
  - Case 2: Both UDP flows are sending at 0.4 Mbps
    - Both UDP flows get about 0.4 Mbps
    - The two TCP flows share the rest of 0.2 Mbps
- Result: Unfair bandwidth sharing between TCP and UDP

Result of TCP and UDP interaction

- 2 UDP and 2 TCP flows share a common link
- The 2 UDP flows start sending datagrams at 8s



Can anything else cause packet loss?

- If we have a link which is experiencing packet loss that is not due to congestion, TCP will not be able to make a distinction.
- Consider a WiFi network link - it may lose some frames due to radio interference (e.g. a microwave oven!)
- What might this do to TCP performance?
  - TCP will think we have packet loss
  - Therefore, TCP slows down the rate of transmission
  - This does not help - in fact, it just reduces throughput!

Local (layer 2) repair

- The real problem is that TCP error control is designed to work over a long, end-to-end connection with possible congestion-related packet loss
- It take some time (possibly several few end-to-end round trip times) to recover - maybe up to a few seconds
- This is an expensive procedure, especially if there *is no congestion*.
- How might we help to avoid it?

Local (layer 2) repair

- The solution is for layer 2 (data link layer) protocols which expect some packet loss (wireless links mainly) to take care of retransmission of lost frames
- Why?
  - Recall: RTT is both *very low* (microseconds) and *very predictable* for one hop
  - So we can set the timeout to be very short, and immediately retransmit a lost or corrupted frame
- This means that instead of a missing TCP segment, Layer 4 will just see a very slight additional delay while the frame is being retransmitted (maybe a few milliseconds)
- TCP's mean RTT estimation algorithm won't be affected much - just a slight, temporary increase in the timeout value
- No expensive retransmissions are needed at Layer 4 and slow start won't be triggered.

Summary

- TCP congestion control
  - Slow start
  - Fast retransmit and fast recovery
  - Pro-active congestion avoidance
- UDP
  - The interaction of UDP and TCP
  - TCP over lossy networks