11 - TCP

TCP Overview

Tasks

- Muxing - UDP and TCP
- Reliability - TCP only
- Flow Control - TCP only
- Congestion Control - TCP only

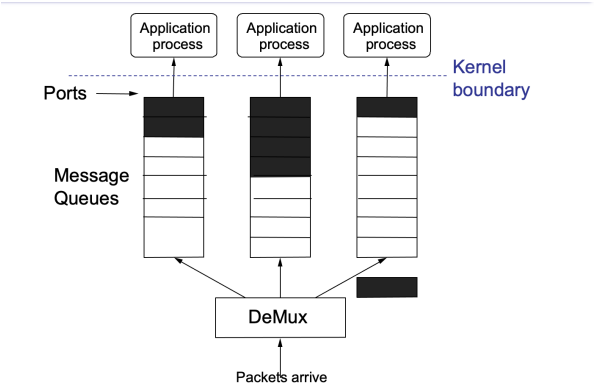
Process Abtraction

- transport protocol abstract proceses to ports
- processes are identified by IP addr (32-bits), protocol, and port (16 bits)
- services abstracted to their required protocol: http (web), smtp (mail), dns, etc.

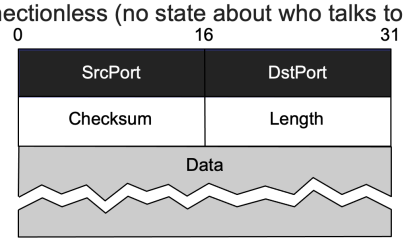
Port Selection

- OS assigns permanent ports to value $\leqslant$ 1024 e.g. http 80, smtp 25, dns 53
- OS allocates ephemeral (temp) ports to values above 1024

UDP - User Datagram Protocol



- only handles multiplexing
- no reliability in ordering, etc.
- no need for retransmit or acks
- used for low latency reqs e.g. streaming, dns, NTP (Network Time protocol - clock syncing), video games
- Connectionless (no state about who talks to whom)

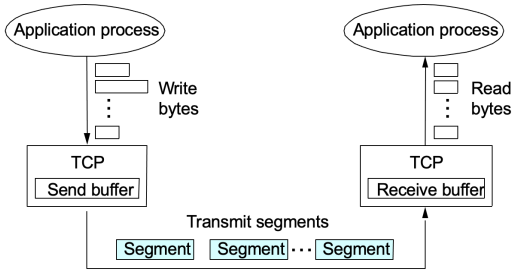


- "connectionless" packet delivery

TCP - Transmission Control Protocol

- Reliable bi-directional bytestream between processes
  - Uses a sliding window protocol for efficient transfer

    - Huge sequence numbers (4-bytes) because of possibly very old out of order packets
- Connection-oriented
    - Conversation between two endpoints with beginning and end
- Flow control (generalization of sliding window)
    - Prevents sender from over-running receiver buffers (tell sender how much buffer is left at receiver)
- Congestion control (next lecture)
    - Prevents sender from over-running network capacity



- TCP delivery

Reliability - Data Link vs TCP

- Network instead of single FIFO link
    - packets can be delayed for large amounts of time
    - duplicates can be created by packet looping: delayed duplicates imply need for large sequence numbers.
    - packets can be reordered by route changes.
- Connection management
    - Only done for Data Link when a link crashes or comes up
    - Lots of clients dynamically requesting connections
    - HDLC didn't work: here more at stake, have to do it right.
-

- Data link only needs speed matching between receiver and sender (flow control). Here we also need speed matching between sender and network (congestion control)
- Transport needs to dynamically round-trip delay to set retransmit timers.

Similarity to Project

Similarities

- 3-way handshake with syn acks
- retransmission after 3 dup acks
- large seq num to handle retransmitted old packets

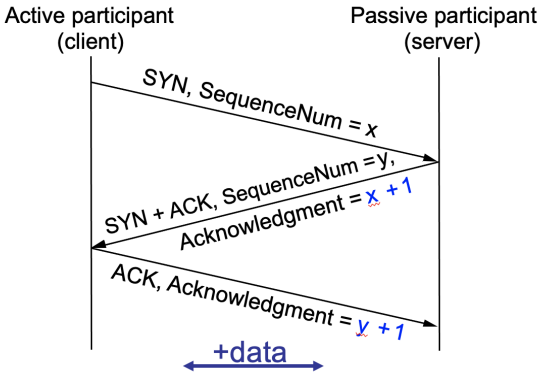
Differences

- retransmission timer set dynamically by calc round trip delay/time
- window is fixed but TCP calcs dynamically based on flow and congestion control
- selective reject support (TCP SACK) in addition to go back N

TCP Connection Handling

- nodes identified by 4/5-tuple
  `(src IP, src PORT, dst IP, dst PORT, protocol)`
- seq number assigned per byte (allows for packet sizes to change between transmissions)

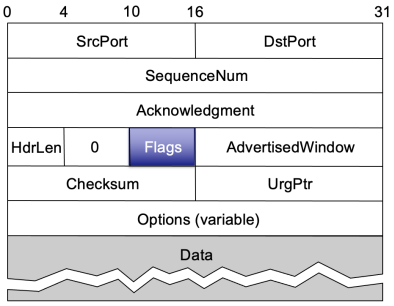
3-Way Handhake



- 3-Way Handshake with nonce seq
- We could abbreviate this setup, but it was chosen to be robust, especially against delayed duplicates
  - Three-way handshake first described inTomlinson 1975
- Choice of changing **initial sequence numbers (ISNs)** minimizes the chance of hosts that crash getting confused by a previous incarnation of a connection
- How to choose ISNs?
  - Maximize period between reuse
  - Minimize ability to guess (why?)
  - Random works OK, as in Project 2
- Operation
  1. Server: If in LISTEN and SYN arrives, then transition to SYN_RCVD state, replying with ACK+SYN.
  2. Client: active open, send SYN segment and transition to SYN_SENT.
  3. Arrival of SYN+ACK causes the client to move to ESTABLISHED and send an ack
  4. When this ACK arrives the server finally moves to the ESTABLISHED state.

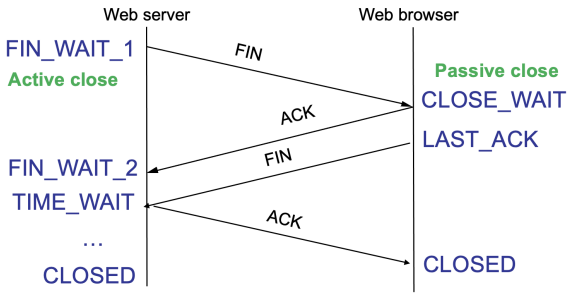
TCP Header

- src dest identified by IP + PORT
  - Flags may be ACK, SYN, FIN, URG, PSH, RST



TCP Disconnect

1. Need timers anyway to get rid of connection state to dead nodes.
2. However, timer should be large so that "keepalive" hello overhead is low.
3. If communication is working, would prefer graceful closing (so receiver process knows quickly) to long timers.
4. Hence 3 phase disconnect handshake After sending disconnect and receiving disconnect ack, both sender and receiver set short timers.



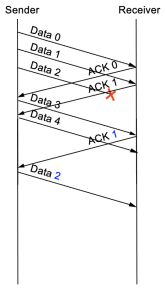
- enter `TIME_WAIT` to handle very old packets that may not have been retransmit or have not been acked yet
- We wait $2 \times MSL$ (maximum segment lifetime of 60 seconds) before completing the close
- Why?
  - ACK might have been lost and so FIN will be resent
  - Could interfere with a subsequent connection
- Real life: Abortive close
  - Don't wait for $2 \times MSL$, simply send Reset packet (RST)

TCP Reliability

- Usual sequence numbers except:
  - Very large to deal with out of order (modulus > 2 W etc. only works on FIFO links). As in Project 2
  - TCP **numbers bytes** not segments: allows it to change packet size in the middle of a connection
  - The sequence numbers don't start with 0 but with an **ISN**.
- Reliable Mechanisms similar except:
  - TCP has a quicker way to react to lost messages
  - TCP does a **crude form of selective reject** not go-back N
  - TCP does flow control by allowing a dynamic window which receiver can set to reduce traffic rate (next lecture)
- recall go-back-N: retransmit all packets from last ack
- real timer for retransmit is based on calculated **Round Trip Time (RTT)**

TCP Fast Retransmit

- like selective reject but immediate retransmit with sliding window buffer

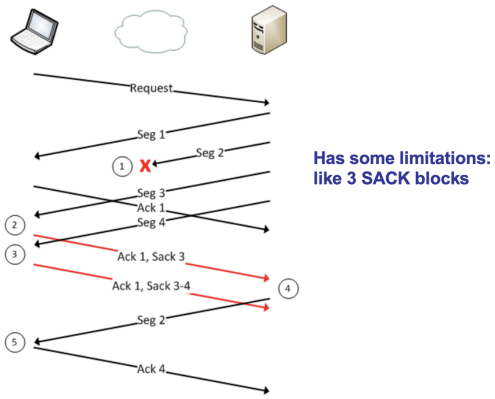


- Don't bother waiting
  - Receipt of duplicate acknowledgement (dupACK) indicates loss
  - Retransmit immediately
- Used in TCP
  - Need to be careful if frames can be reordered
  - Today's TCP identifies a loss if there are three duplicate ACKs in a row. Project 2 did this!

- timer as backup, use 3 dup acks to identify

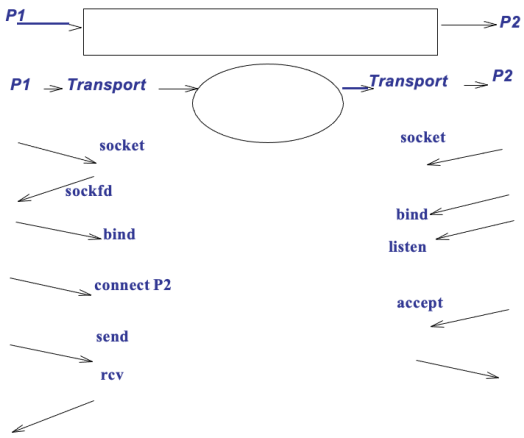
TCP SACK - Selective ACK

- ack with last received in-order packet, BUT also send sack with received out of order packets



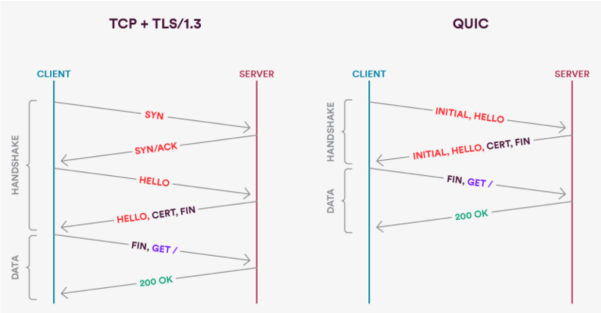
- limitations: ack/sack info could be very long if many drops and many out of order packets
- solution: limit to 3 sack blocks (ranges) - act like any out of order packets after reporting at most 3 sack ranges are considered to be lost
- e.g., ACK 1, SACK 3-8, SACK 10-15, SACK 17-25 ⇒ packets 2,9,16 are lost and any after 25 are considered lost or not sent yet

Socket API



QUIC - Quick UDP Internet Connections

- developed by google to make initial handshake faster with TLS



- Runs on UDP
- also adds other stuff like congestion and flow control for UDP

The trick to lower latency

- Idea 1: Combine security handshake and sequence number handshake on first connection to server

- Idea 2: If server remembers information about client, need 0 handshakes on later connections
- Three way handshakes are required because server forgets info on client. Important in old days but no longer as severs have massive memory
- A round trip is a big deal (several hundred msec across US) at today's high speeds

Additional Stuff

- Stream multiplexing: multiple streams in a single QUIC connection between client and server for HTTP/2
- No head of line blocking: can do HTTP/2 over a single TCP connection but a single loss stalls all streams. Not so in QUIC
- Shared congestion information: as we will see it takes TCP a long time to ramp up. In QUIC all congestion information is shared.
- Wave of future: 9% of all websites use QUIC (4/2023). 40% of Chrome traffic uses QUIC 2

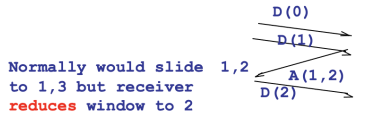
Congestion/Flow Control

- Flow Control - Changing sender speed to match receiver speed
- Congestion Control - Changing sender speed to match network speed

Flow Control

- flow control - receiver sets the recv window size = 0, but then deadlock if window size = 0, so periodically receiver needs to send a packet or probe to maintain the connection
- FREEZE - receiver could also send back a FREEZE message (wich contains a timeout) that keeps the connection open but stops the sender fro sending until it receives another packet from the node that sent the freeze - needs seq numbers for freeze because delayed duplicates may restart freeze timeout

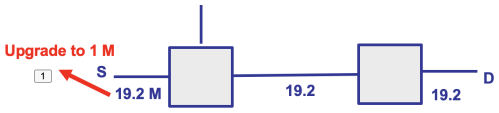
Windows provide static flow control. Can provide dynamic flow control if receiver acks indicate Lower and Upper Window Edge.



Need to avoid deadlock if window is reduced to 0 and then increase to c > 0. In OSI, receiver keeps sending c. In TCP, sender periodically probes an empty window.

Congestion Control

- different connection bandwidths leads to congestion on high output senders



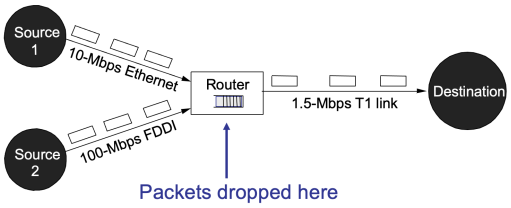
1) WHEN LINK FROM S TO FIRST ROUTER WAS UPGRADED FROM 19.2 KBPS TO 1 MBPS, THE TIME FOR A FILE TRANSFER WENT UP FROM A FEW SECONDS TO A FEW HOURS

2) THIS HAPPENED IN AN EXPERIMENT IN DEC IN THE 1980s. SHOWED THE NEED FOR CONGESTION CONTROL (DECBIT)

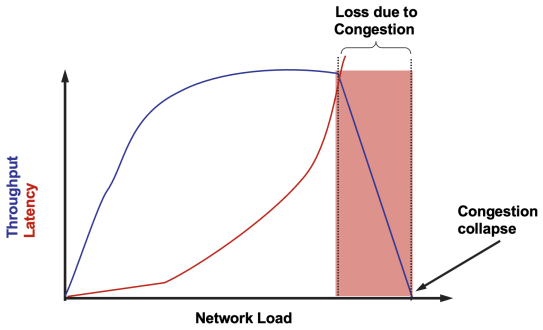
3) VERY SIMILAR EXPERIENCES IN INTERNET LED VAN JACOBSON TO PROPOSE TCP CONGESTION CONTOL. VAN IS AN AFFILIATE PROFESSOR AT UCLA!

- queue builds if send rate > drain rate → dropped packets

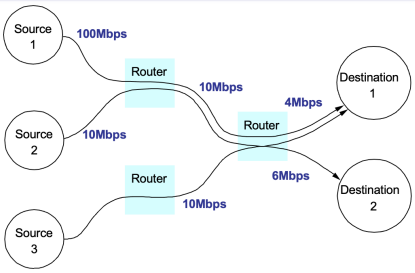
- "goodput" amount of packets that come "out" of the network



- Congestion causes both collapsed throughput ("goodput") and massive latency

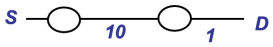


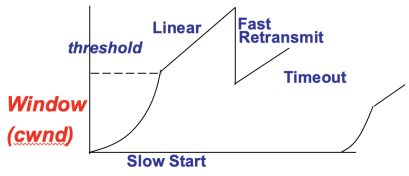
Fair Bandwidth Allocation



- Can use Ford-Fulkerson Maximum Flow Algo

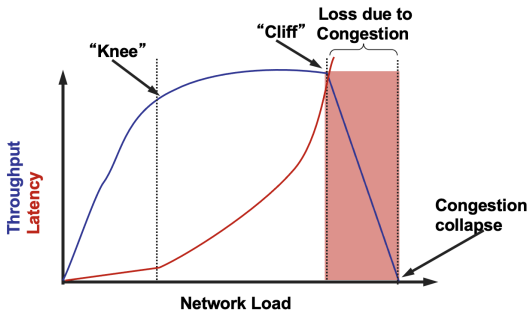
Mitigation



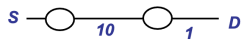


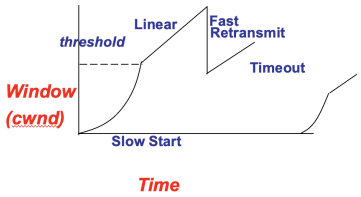
- "Slow start" - start with sender window size of 1; double every time we get a successful ack
- tighten window if timer elapses or 3 dup acks
- AIMD - Additive Increase, Multiplicative Decrease - tighten multiplicatively (div by 2) and open additively (by MSS) the window size (aperture)
  - used for congestion avoidance (proactive)
- ECN - Explicit Congestion Notification - ECN bit, requires another packet from dest to sender

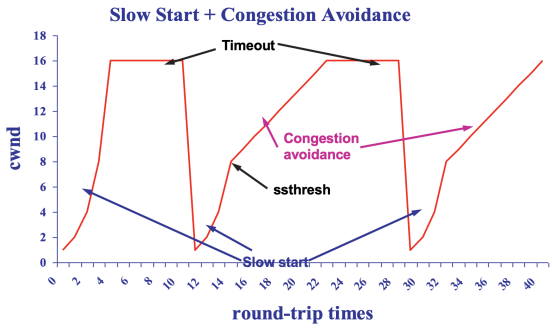
- Proactive Congestion Control - stay left of knee; Reactive - stay left of cliff



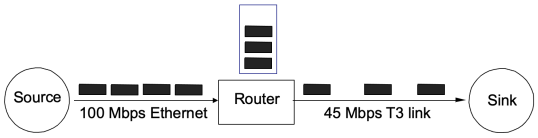
  - Compromise: adaptive approximation
    - If congestion signaled, reduce sending rate by x
    - If data delivered successfully, increase sending rate by y
      - Window-based congestion control
        - Unified congestion control and flow control mechanism
        - *rwin*: advertised flow control window from receiver
        - *cwnd*: congestion control window
          - » Estimate of how much outstanding data network can deliver in a round-trip time
        - Sender can only send MIN(*rwin*,*cwnd*) at any time
      - Idea: decrease cwnd when congestion is encountered; increase cwnd otherwise
        - Question: how much to adjust?
- Basic Algo
  - ssthresh cliff to mitigate exponential opening on slow start







Probing the Network



- Each source independently probes the network to determine how much bandwidth is available
  - Changes over time, since everyone does this
- Assume that packet loss implies congestion
  - Since errors are rare; also, requires no support from routers
-

- TCP uses AIMD to adjust congestion window
  - Converges to fair share of bottleneck link
  - Increases modestly in good times
  - Cuts drastically in bad times
- But what rate should a TCP flow use initially?
  - Need some initial congestion window
  - We'd like to TCP to work on all manner of links
  - Need to span 6+ orders of magnitude, e.g., 10 K to 10 Gbps.
  - Starting too fast is catastrophic! So start with cwnd = 1

ECN

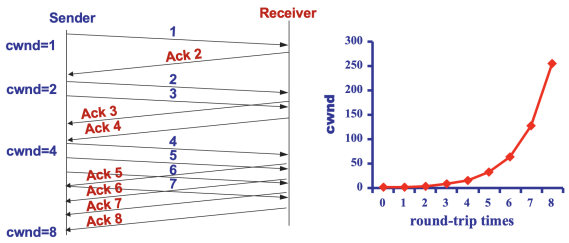
- Explicit congestion signaling
  - Source Quench: ICMP message from router to sender
  - DECBit / Explicit Congestion Notification (ECN):
    - Router marks packet based on queue occupancy (i.e. indication that packet encountered congestion along the way)
    - Receiver tells sender if queues are getting too full
- Implicit congestion signaling
  - Packet loss
    - Assume congestion is primary source of packet loss
    - Lost packets indicate congestion
  - Packet delay
    - Round-trip time increases as packets queue
    - Packet inter-arrival time is a function of bottleneck link

Throttling

- Window-based (TCP)
  - Constrain number of outstanding packets allowed in network
  - Increase window to send faster; decrease to send slower
  - Pro: Cheap to implement, good failure properties
  - Con: Creates traffic bursts (requires bigger buffers)
- Rate-based (many streaming media protocols)
  - Two parameters (period, packets)
  - Allow sending of x packets in period y
  - Pro: smooth traffic
  - Con: fine-grained per-connection timers, what if receiver fails?

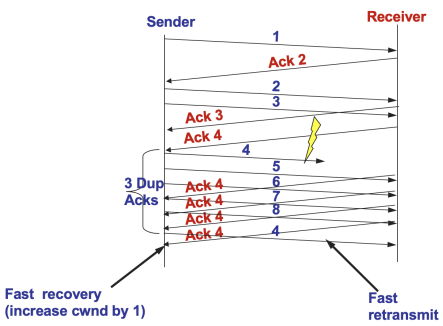
Slow Start

- Goal: quickly find the equilibrium sending rate
- Quickly increase sending rate until congestion detected
  - Remember last rate that worked and don't overshoot it
- TCP Reno Algorithm:
  - On new connection, or after timeout, set *cwnd*=1 MSS
  - For each segment acknowledged, increment *cwnd* by 1 MSS
  - If timeout then divide *cwnd* by 2, and set *ssthresh* = *cwnd*
  - If *cwnd* >= *ssthresh* then exit slow start
- Why called slow? Its exponential after all…

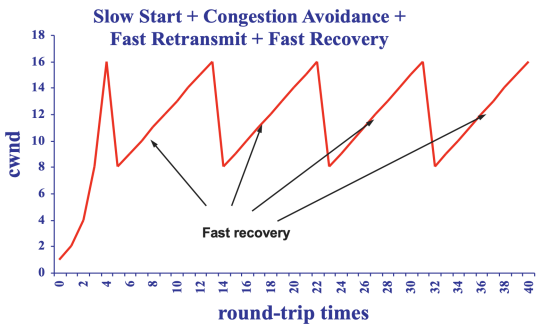


Fast Recovery

- Fast retransmit
  - Timeouts are slow (default often 200 ms or 1 second)
  - When packet is lost, receiver still ACKs last in-order packet
  - Use 3 duplicate ACKs to indicate a loss; detect losses quickly
- Fast recovery
  - Goal: avoid stalling after loss
  - If there are still ACKs coming in, then no need for slow start
  - If a packet has made it through → we can send another one
  - **Divide cwnd by 2** after fast retransmit
  - Increment cwnd by 1 MSS for each additional duplicate ACK
- e.g.



- all together



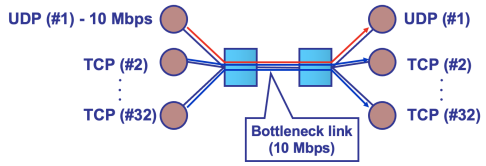
Open Problems

- most connections short, possibly no gain from low start
- magic number to run low tart or fast recovery, etc.
- shared bandwidth for UDP and TCP
- decide which packets can be dropped: syn, ack, none

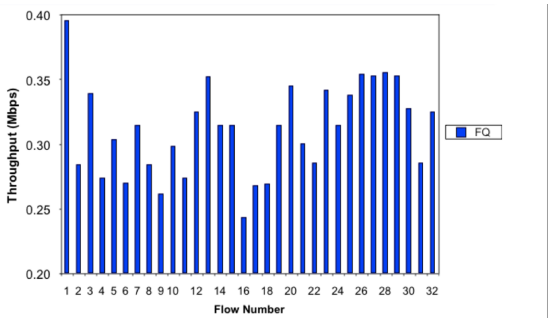
Router Scheduling

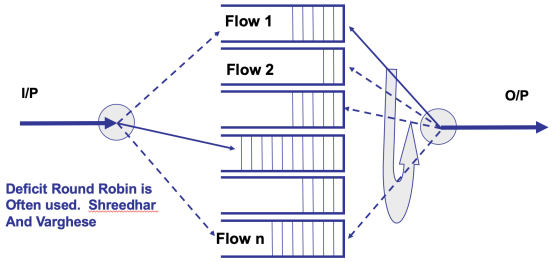
- router also need to schedule packets to support congestion control

  1 UDP (10 Mbps) and 31 TCPs sharing a 10 Mbps line



- use fair queueing → also allow TCP-UDP bandwidth share



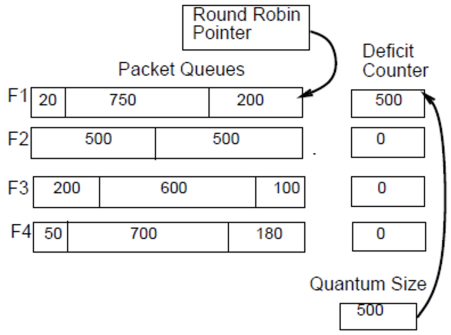


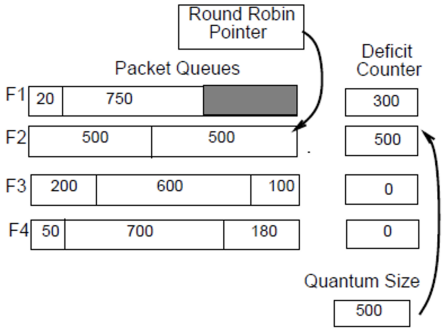
- Round-Robin
- but we need to weight round robin due to differing packet sizes on top of queue frequent use

DRR - Deficit Round Robin

- use a deficit counter that starts with some quantum size, then decrease the deficit by

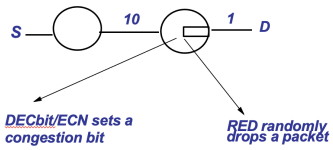
  the packet size sent





- after packet sent at robin pointer

- also enable Random Early Detect - randomly drop a packet early before congestion if ECN not set so that an ECN may be sent early



Q: Why would a router drop a perfectly good packet even if has buffer space
A: As an early form of congestion warning if one does not have a congestion bit. Many IP routers have such a bit today, called the ECN bit

Novel Approaches

- HTTP Pipelining: To reduce latency browser opens up multiple connections. Still slow over TCP.
- QUIC: Layered below HTTP and above TCP, places multiple streams in a single connection. Finesses slow start. Loss on one stream does not interrupt other stream. Avoids extra handshakes for TLS
- TCP fluctuates too much. New protocols like DCTCP adjust more smoothly in Microsoft Data Centers. Also used by Apple