Switches

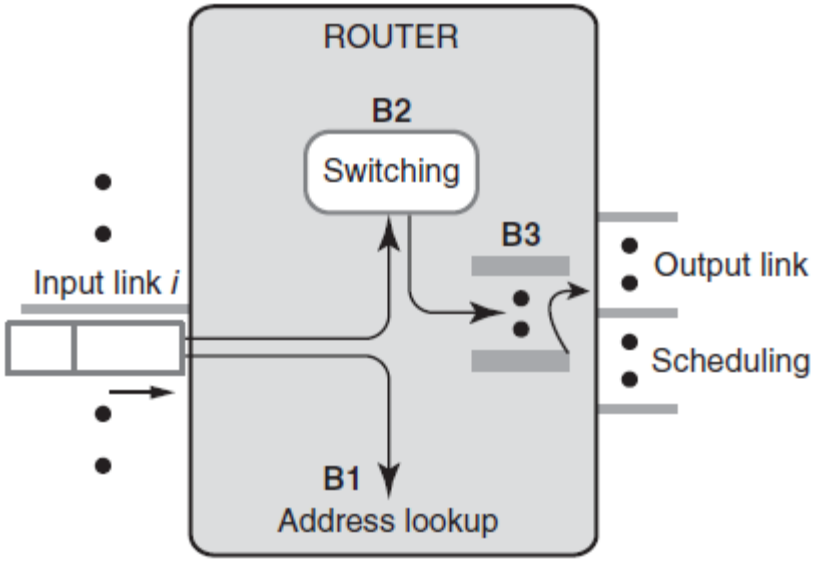
Buffer Management & Scheduling

Buffer Management and Scheduling

When the output link is congested, packets are placed in a queue.

Defaul option is FIFO with tail-drop

- First In First Out
- when the buffer is full, drop all the new packets



Why FIFO and tail drop is a bad idea

Most Internet traffic is TCP.

In TCP packet discard is seen as an (implicit) indication of congestion.

Sources react by slowing down.

Problems:

- sources react when it is too late (i.e. when the buffer is dropping a lot of packets)
- bursts of packet losses make the retransmission timeout expire
- the indication of congestion is notified to all the sources at the same time and sources synchronize (this is bad!)
  - sources slow down at the same time (inefficient)
  - sources increase their speed at the same time (causing congestion)
- the greedy sources starve well-behaving sources

The need for better buffer management

- Better support for congestion management
  - Routers should provide useful information to the TCP sources by choosing which packets to discard and when
- Fair sharing of links among competing flows
  - Greedy, long-lived flows should not grab all the buffers starving other flows
- Providing Quality of Service
  - Ideally we want to provide precise bandwidth and delay guarantees to flows, depending on the application

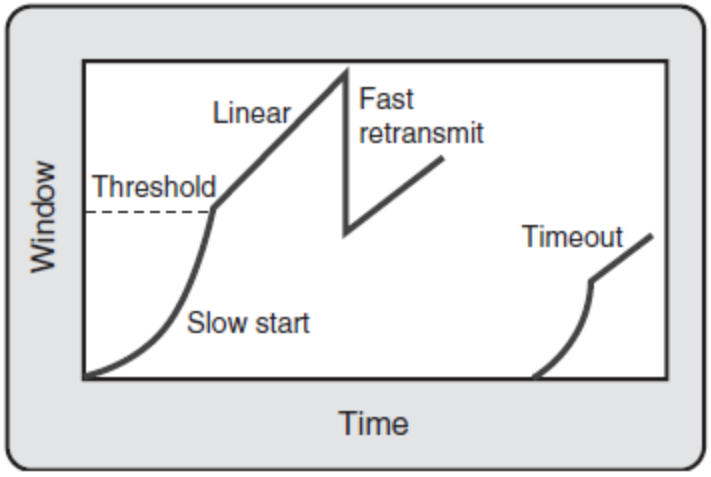
Buffer Management
Random Early Detection

De-facto standard for buffer managment in modern routers.

When the average queue length goes beyond a given threshold, RED starts randomly dropping packets, even if there is room.

Why?

First we must review TCP congestion control.



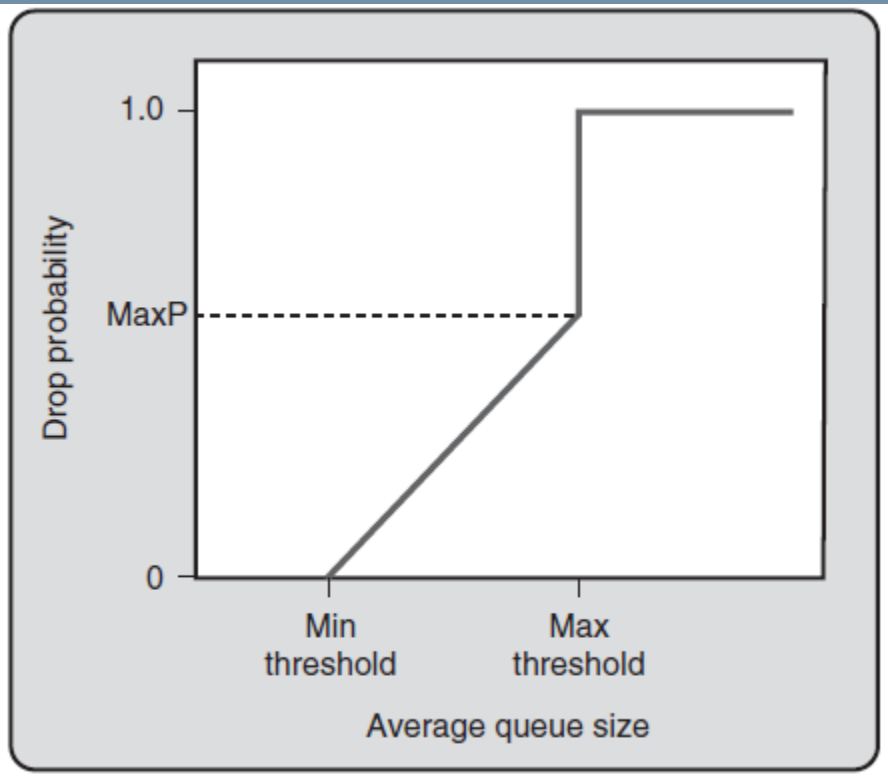
TCP Congestion Control

- When no packets are dropped, the congestion window grows exponentially, then linearly.
- If a single packet is dropped (detected by duplicate acks), the gap is repaired by retransmitting the missing packet and halving the window and continuing the linear growth
- If several packets are dropped, the receiver stops until the timeout expires (hundreds of ms), then the window drops to 1
- It is important to understand that eventually TCP will always saturate the bandwidth and lose packets. This is an effect of TCP exploratory behavior.
- Ideally we want a single packet loss when TCP approaches the maximum bandwidth and small oscillations
- Expensive timeouts should be avoided

Random Early Detect (RED)

- RED drops a packet with a probability that depends on the average queue size
- Therefore, drops are isolated and trigger Fast Retransmit to few sources at a time. This also avoids synchronization.
- At equilibrium, in the bottleneck router, queue size floats between the thresholds.
- Using the average queue size allows bursts to pass without triggering losses.

Random Early Detect (RED)



$$\text{Average}Q = (1 - W) * \text{Average}Q + (W * \text{Sample}Q\text{size})$$

Optimal choice of system parameters is difficult and depends on the link speed and propagation delay and on the expected RTT of the connections.

Discussion of RED

Given the packet drop probability p_b, how do we drop a packet?

- Each packet is dropped with probability p_b
  - This results in intervals between losses distributed as a geometric random variable and potentially long intervals between losses alternating with bursty losses
- Each packet is dropped with a probability increasing after each drop event
  - Prob. of dropping each packet: $p_b / (1 - c\ p_b)$
  - c = number of non dropped packets after the last drop
  - The interval between losses is uniform

- Weighted RED: the drop parameters are calculated according to some values in the ToS field

Token Bucket Policing

Sometimes it is useful to limit the speed of a flow to some fixed maximum speed without managing separate per-flow queues

The token bucket policer limits the maximum rate of a flow and the maximum number of packets in a burst. It requires a counter and a timer per flow

The counter increases over time with rate R up to a maximum value B

When a packet with size L arrives:

- If the counter is smaller than L, the packet is dropped
- If the counter is greater or equal to L, the counter is decremented by L and the packet is admitted

A variant (the token bucket shaper) does not drop packets but parks them until enough tokens are available. It is much more complex because it requires multiple, per-flow queues.

Scheduling

To provide Quality of Service, the router should give each “flow”:

- bandwidth guarantees (bits served over a given time scale)
- delay bounds (time between the arrival of the packet and its service)

We need multiple outbound queues, and thus:

- a classifier to choose the queue for each packet
- a per-queue buffer management scheme
- a scheduler to choose which queue to serve

Simple schedulers:

- strict priority
- Round Robin (RR) and its extensions
  - Deficit Round Robin (DRR)
  - Class-Based Queuing (CBQ)

Complex schedulers:

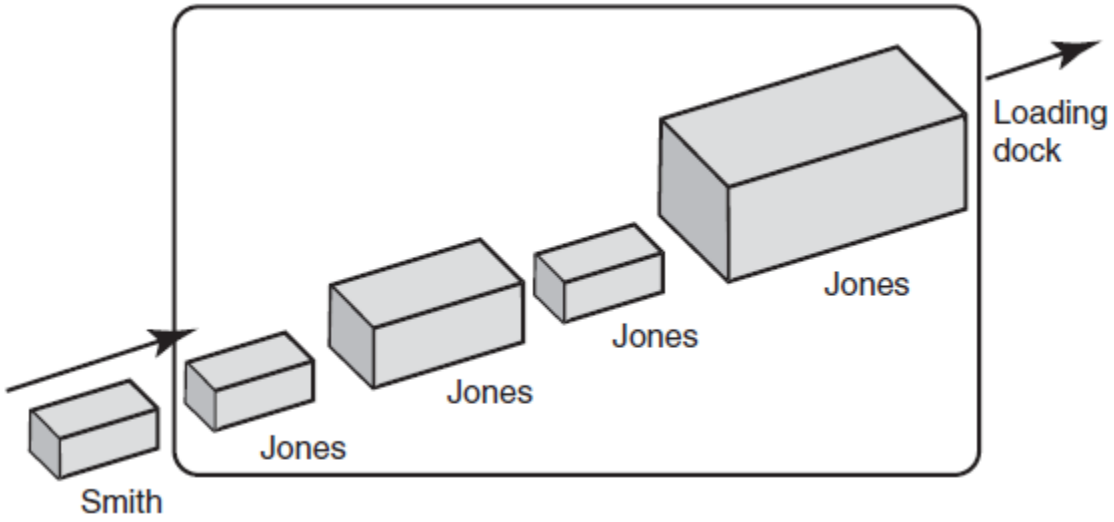
- bit-to-bit round robin and its practical realizations

Single outbound queue / FIFO

Greedy flows get most resources

A single flow can monopolize the service

No fairness, no bandwidth guarantee, no delay bounds

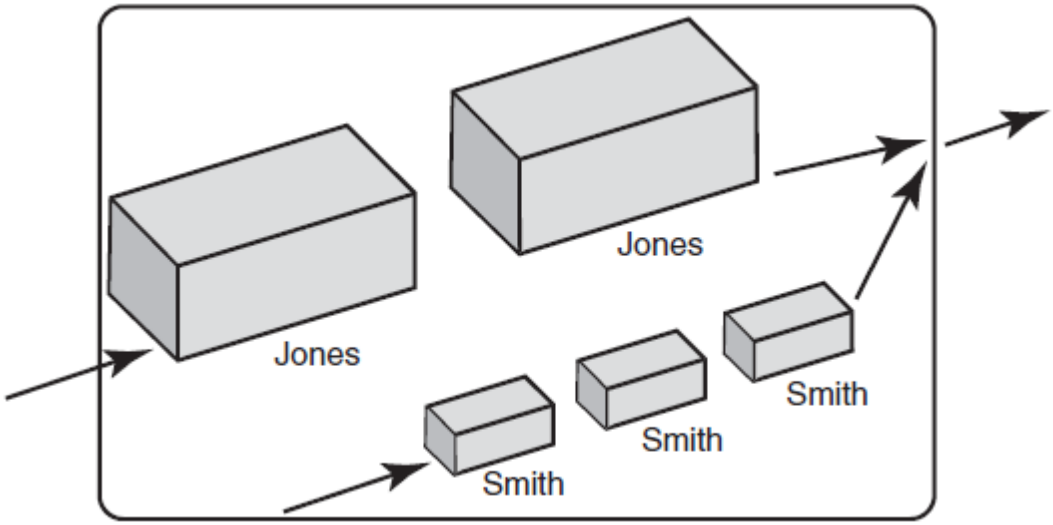


Round Robin

Round Robin keeps a queue for each flow and serves a single packet from each queue

Unfair if packets have variable sizes

Ideally, bit-by-bit round robin could provide bandwidth guarantees and delay bounds



Deficit Round Robin (DRR)

We drop the bounded delay requirement in order to gain efficiency.

For each flow *i* keep a quantum size Qi and a deficit counter Di.

- The Qi can be different
- DRR shares bandwidth among flows proportionally to Qi

Cicle all the flows as in round robin. For each flow i:

- increase the deficit by the quantum: $Di \leftarrow Di + Qi$
- send at most Di bits (or bytes) from the flow queue
  - only send whole packets!

DRR is fair in the long term.

The possibilty of sending more than one packet per round may result in short term unfairness. (And unpredictable delays)

Variant: Hierarchical DRR

DRR example

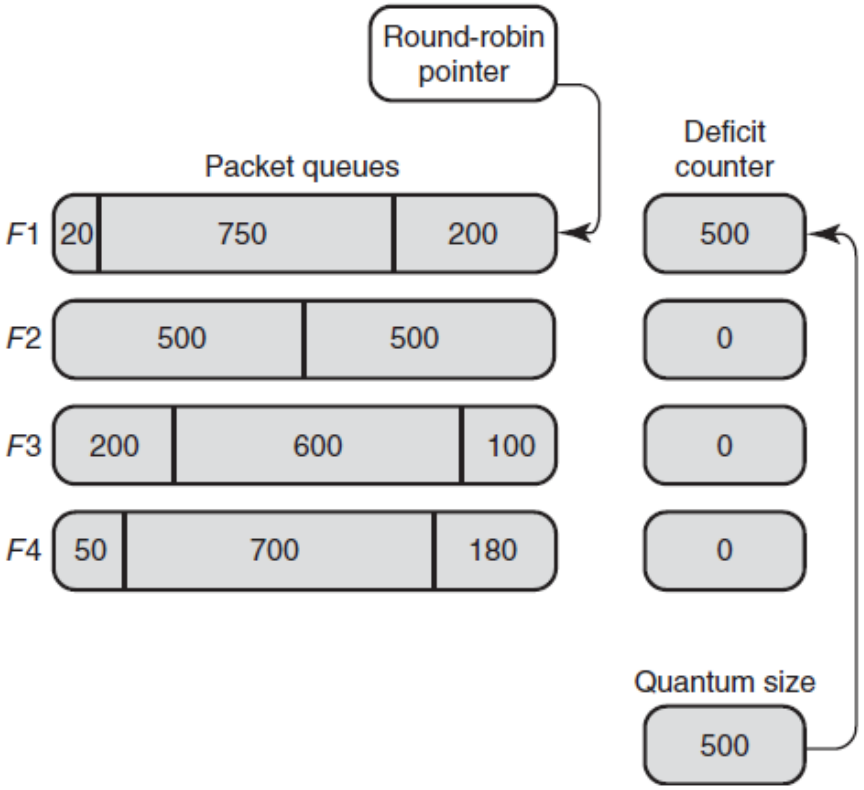
Assume that

- all queues have pakets to send (they are *backlogged*)
- Di = 0 at the beginning
- Qi = 500 (bytes)

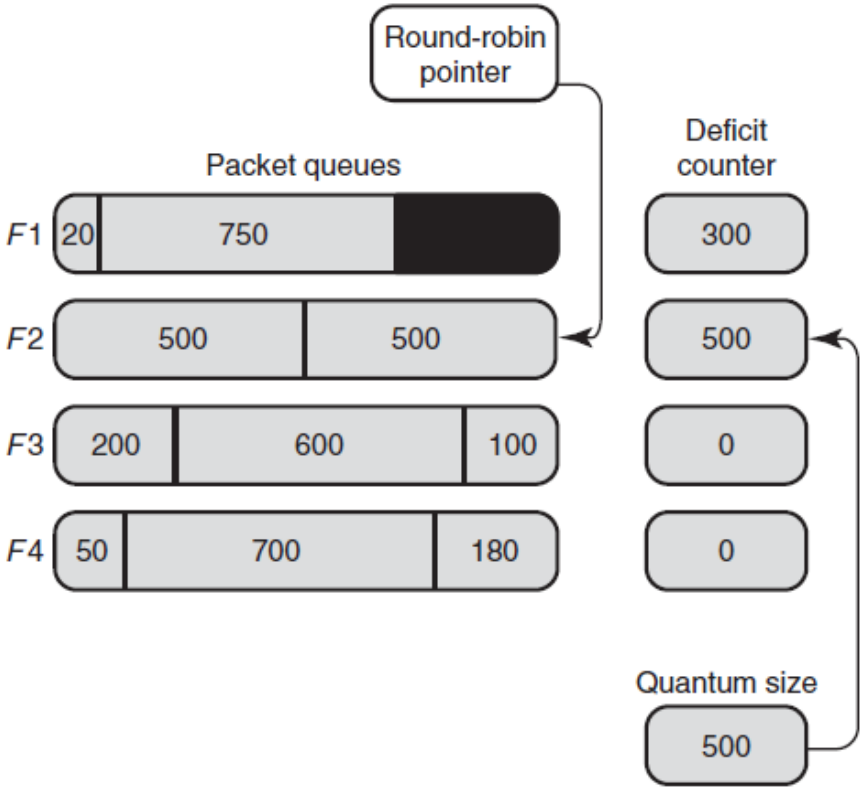
Start with flow F1

- D1 ← 0 + Q1 = 500
- send the 200-byte packet
- D1 ← 500 – 200 = 300
- the 750-byte packet is blocked
- save 300 byte

DRR example


Round-robin pointer
Packet queues
Deficit counter
F1
20
750
200
500
F2
500
500
0
F3
200
600
100
0
F4
50
700
180
0
Quantum size
500

DRR example


Round-robin pointer
Packet queues
Deficit counter
F1
20
750
300
F2
500
500
500
F3
200
600
100
0
F4
50
700
180
0
Quantum size
500

DRR details

When a queue is empty its deficit is zeroed

- otherwise, when it would grow indefinitely when the flow is silent

If there are several queues a lot of time is wasted visiting empty queues

How to fix?

- Keep a redundant queue (called *ActiveList*) with pointers to flows with nonempty queues

Using the *ActiveList*

- Pick the first flow in the list serve its packets
- if a queue becomes empty, remove it from the ActiveList
- if there are other packets, move it to the bottom

If an idle flow becomes active (a new packet arrives) put it at the bottom of the ActiveList

Final note: the algorithm cost per packet is constant if the Qi are larger than a maximum-size packet. This way at least one packet per visit is sent. (But larger Qi result in higher risk of unfairness)