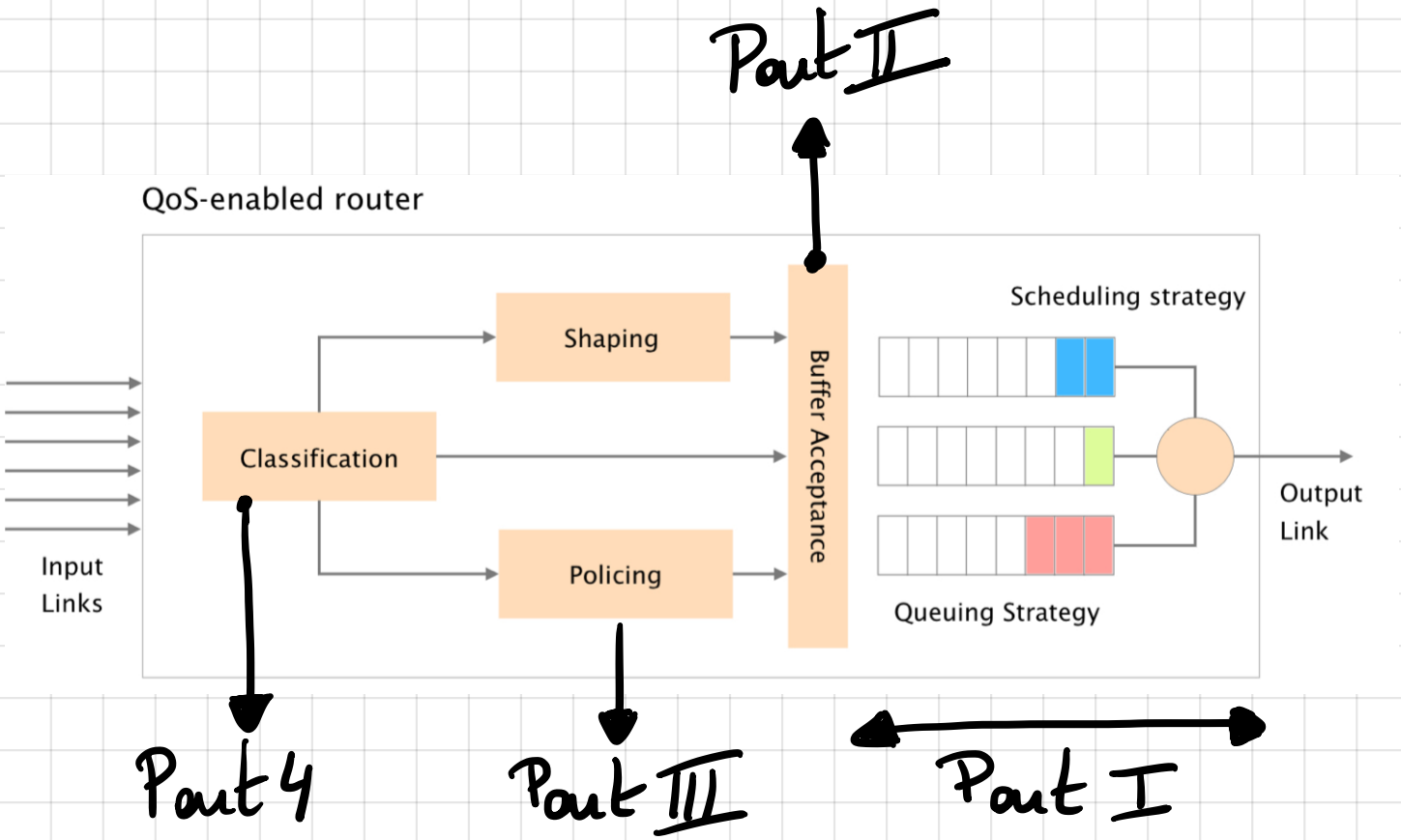
Advanced Topics in Communication Networks

Lecture 6: Quality of Service / 20.10.2020

Prof. Laurent VANBEVER - nsg.ee.ethz.ch



Scheduling

Problem: Which packets gets send out next, and when?

At a high-level, all scheduler can be divided in two categories:

1. work-conserving (fair queuing).
2. non-work conserving. (time division multiplexing system)

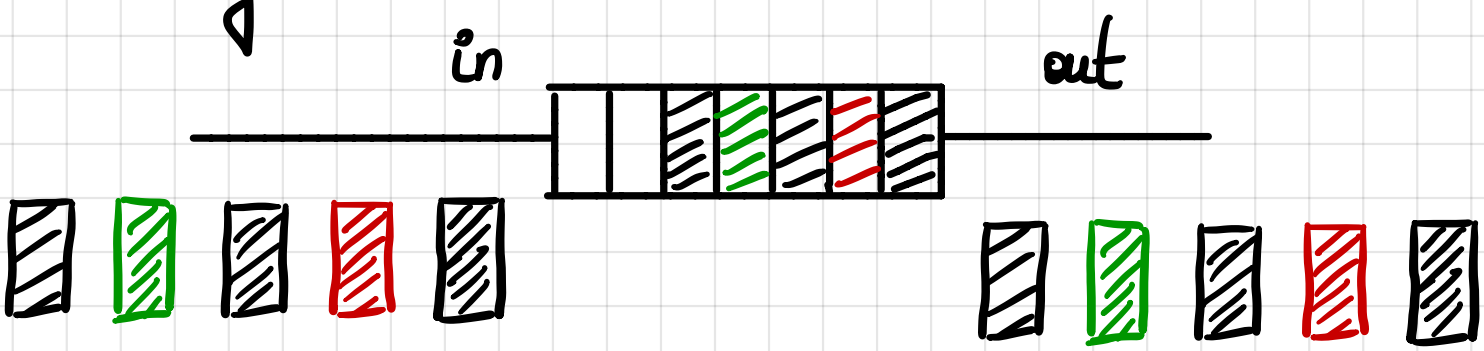
Work conserving algorithms never keep an outband link idle if there is at least one packet in the system waiting to use that link. In contrast, non-work conserving might (to provide delay bound).

An important practical consideration is how scheduling algorithms manage state. Algorithms that maintain per-flow state typically do not scale.

Let's look at the most common scheduling strategies.

A) First-In First Out (FIFO)

The simplest scheduling algorithm there is. FIFO simply queues packet in the order they arrive.

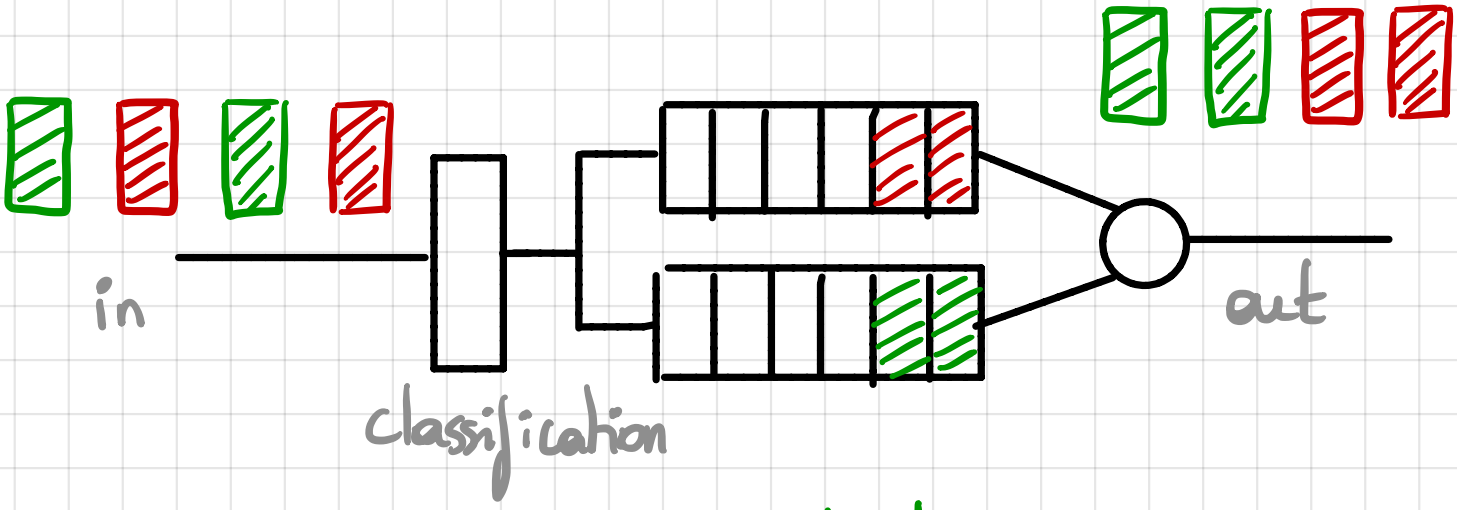


Pros:
- Dead Simple.
- No starvation.

Cons:
- No prioritization (no way to meet deadlines)
- No fairness

B) Priority Queuing (PQ)

PQ schedules traffic such that hipher priority queues always get serviced first. Packets are placed in a queue according to their classification. Packets are serviced in a FIFO order within each queue.



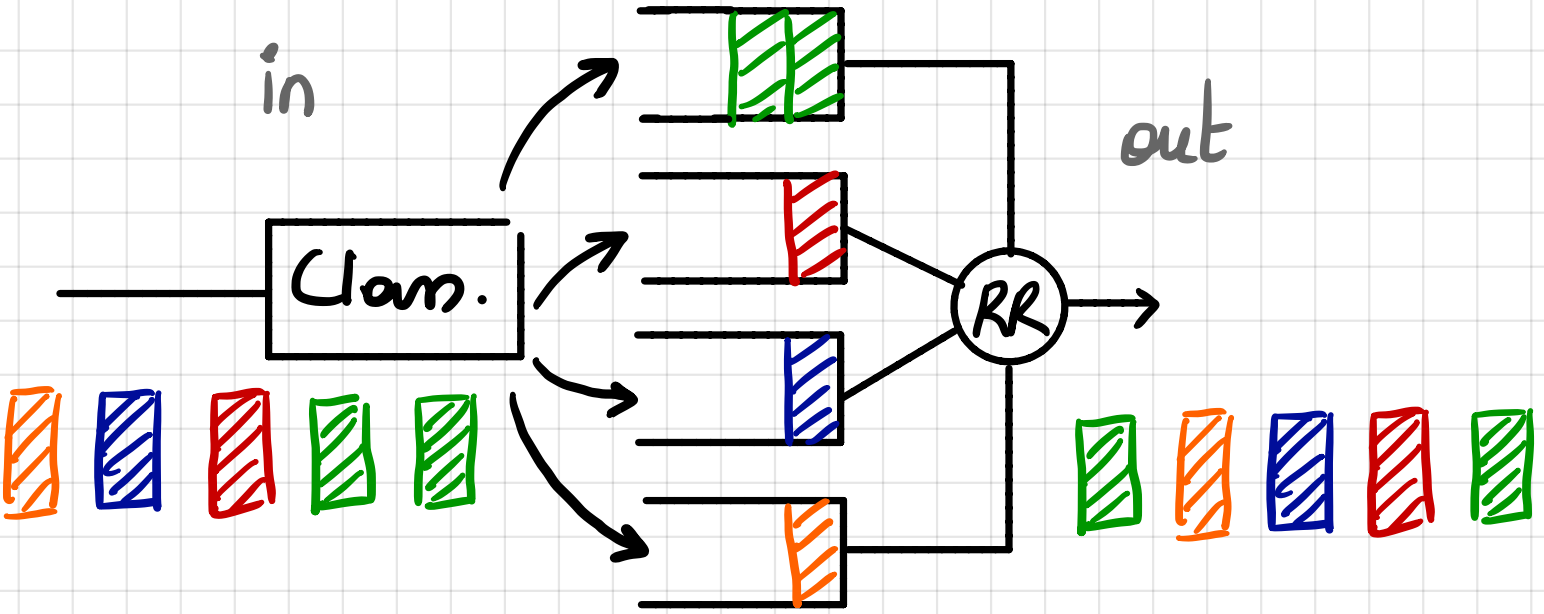
Pros:
- Allow differentiation
- Isolate high priority traffic
  Almost like it has a dedicated link
  (delay is only slightly hipher because of the transmission time of low priority packets).
- Still pretty simple.

Cons: - Possible starvation of lower priority queues if too much high priority traffic. (even if high priority could afford to wait)

To avoid starvation due to high priority traffic, it is common practice to limit the throughput of high priority traffic. (see shaping / policing).

c) (Weighted) Fair Queuing (FQ)

In FQ, incoming packets are classified into ≠ flows and stored into a dedicated queue. A round-robin algorithm is then used to service all queues, so that they are served in a fair way.



Pros:
- Provide isolation
  Misbehaving flows can't impair others
- Provide fairness
  Approximates max-min fair allocation
  (perfect approx. when equal pkt size)

Cons:
- Very demanding resource-wise
  One queue per flow! One device can see 100,000s of flows in busy network → doesn't scale.

A quick reminder on MAX. MIN. FAIRNESS:

Given a set of bandwidth demands r_i and a total bandwidth C, the max-min bandwidth allocations a_i are:

$$a_i = \min(f, r_i)$$

where f is the unique value s.t. $\sum a_i = C$

Important property:

If you (a flow) don't get your full demand, no one gets more than you.

Computing Max-Min Fairness

Let C = link capacity
N = number of flows
r_i = demand of flow i (f_i)

Algorithm:

1. Let $FS = C/N$ (remaining fair share)
2. Let $F = \{f_i \text{ s.t. } r_i \leq FS\}$
3. If $F \neq \phi$ {
4. $C = C - \sum_{f_i \in F} r_i$
5. $N = N - |F|$
6. GOTO (1)
7. } Else {
8. $f = C/N$
9. }

Computing Max-Min Fairness (Example)

$C = 10$; $r_1 = 8$, $r_2 = 6$, $r_3 = 2$; $N = 3$

Iteration 1: $FS = 10/3 = 3.33$

$F = \{f_3\}$

$C = C - r_3 = 8$

$N = 2$

Iteration 2: $FS = 8/2 = 4$

$F = \phi$

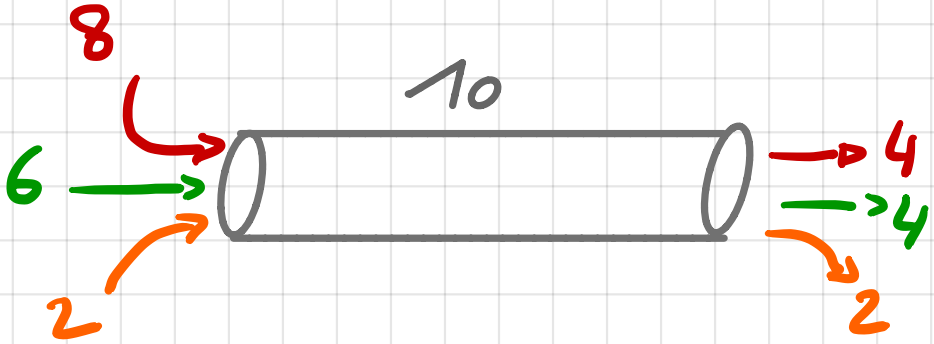
$f = 8/2 = 4$

Final Allocation:

$a_1 = \min(8, 4) = 4$

$a_2 = \min[6, 4) = 4$

$a_3 = \min(2, 4) = 2$



- One of the challenge with FQ is that variable packet sizes might cause bandwidth shares to be uneven. Flows with smaller packets get penalized.
  → packets are not "water" and cannot be divided arbitrarily.

- Yet, one can approximate "perfect FQ" by computing when each bit in the queue should be transmitted and serve the packets in the order of the transmission time of their last bit.
  → makes FQ even harder to implement.

- FQ generalizes to Weighted Fair Queuing (WFQ)
  WFQ specifies a weight to be associated to each flow / queue. The weight specifies how many bits to transmit when the queue is serviced.

In WFQ, the weight of a queue controls the percentage of the bandwidth it will get.

By default, a weight of 1 simply means that each queue gets $1/n$ of the bandwidth (in this case, WFQ = FQ).

Let's say we have 3 queues and configure them with the following weights:

$$w_1 = 2, \quad w_2 = 1, \quad w_3 = 3$$

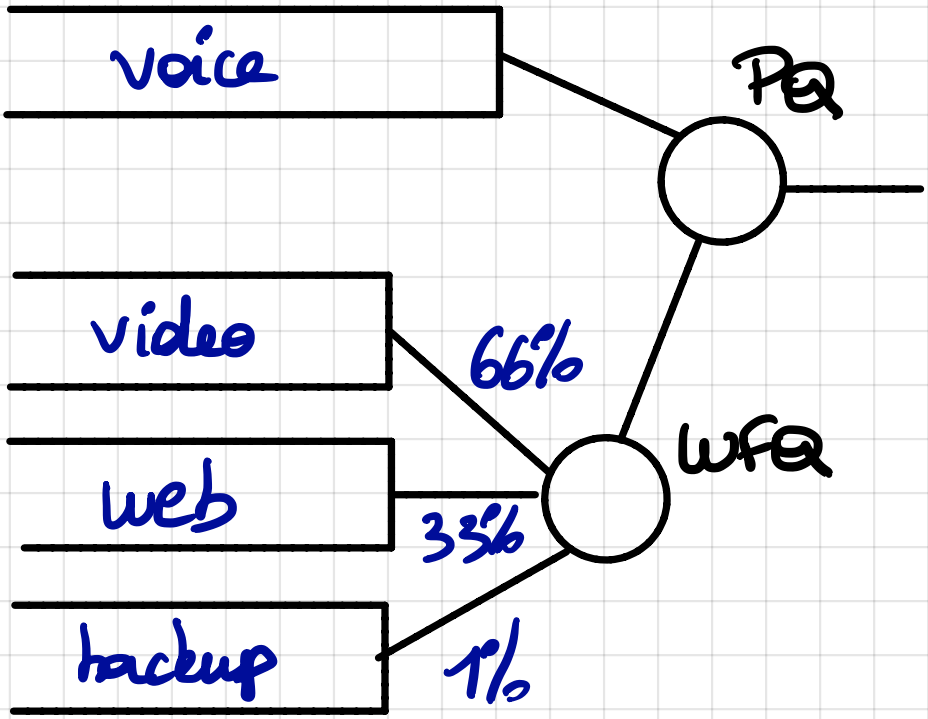
Each of these queues will in turn get

$$b_1 = 1/3, \quad b_2 = 1/6, \quad b_3 = 1/2$$

of the total link bandwidth.

Combining Schedulers:

In practice, ≠ scheduling policies are often used to accomodate ≠ classes of traffic / applications.



Shaping / Policing

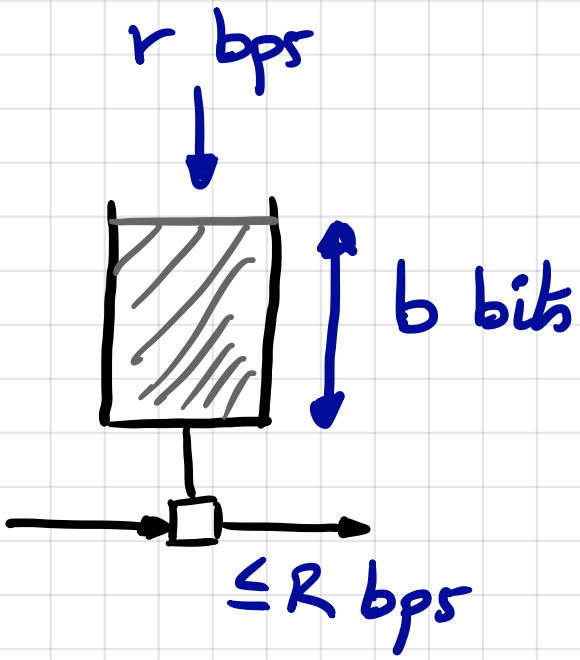
How can we unambiguously characterize / monitor the throughput requirement of an application?

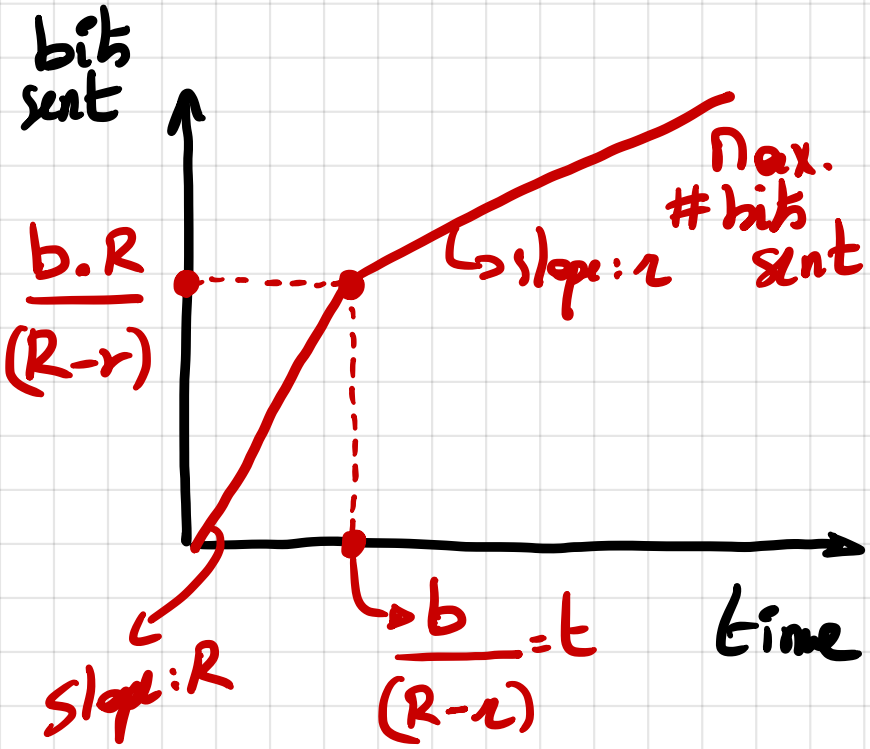
1/ Specify / Monitor the max. rate
→ wasteful! max. rate might be much higher than the average rate.

2/ Specify / Monitor the avg. rate.
→ not sufficient! network will not be able to carry all the pkts. (traffic is often bursty. This leads to bad performance.

3/ Specify / Monitor the burstiness of the traffic, i.e. both the average rate and the burst size.
→ Allow the sender to transmit bursty traffic and the network to reserve the required resources.

Token Bucket: Characterizing Burstiness





$$b + r.t = R.t$$

$$b = t\,(R-r)$$

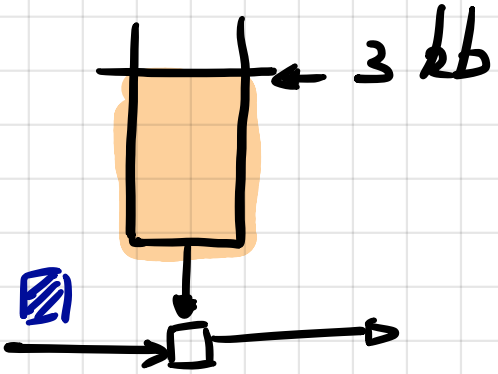
$$\frac{b}{(R-r)} = t$$

Parameters:

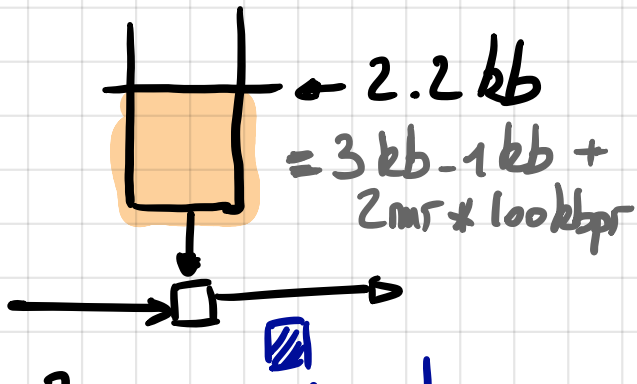
- r bps : bucket filling rate
- b bits : bucket size / max. burst size
- R bps : Maximum link capacity / peak rate

Token Bucket Example

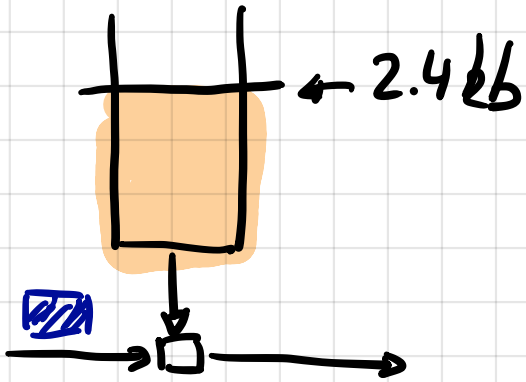
Let $r = 100$ kbps, $b = 3$ kb, $R = 500$ kbps



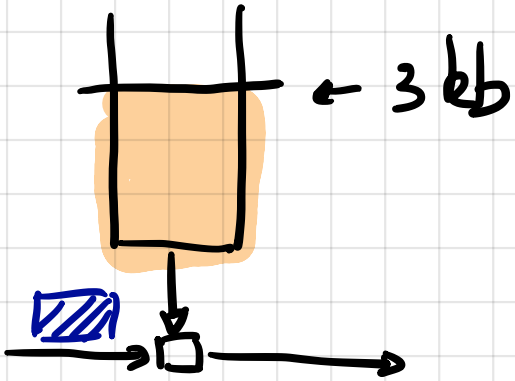
$t = 0$, 1 kb pkt arrives



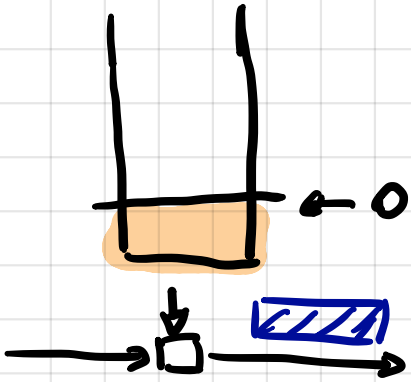
$t = 2$ ms, pkt departs.



$t = 4$ ms, 3 kb packet arrives



$t = 10$ ms, pkt needs to wait until bucket is full again.



$t = 16$ ms pkt is finally transmitted.

Enforcing traffic specification (using a TB): (3 solutions)

1. Policing : DROP all data in excess of the specification
2. Shaping : DELAY all data in excess of the specification UNTIL it obeys it.
3. Marking : MARK all data in excess of the specification and give these packets lower priority in the network.

Buffer Acceptance Algorithms

Once the router has decided which queue it needs to append the packet to it. At this time, it should decide whether to accept this packet or drop it or mark it.

Two common examples:

1. Tail Drop: simply drops any newly arriving packets when the queue is filled to its maximum capacity.

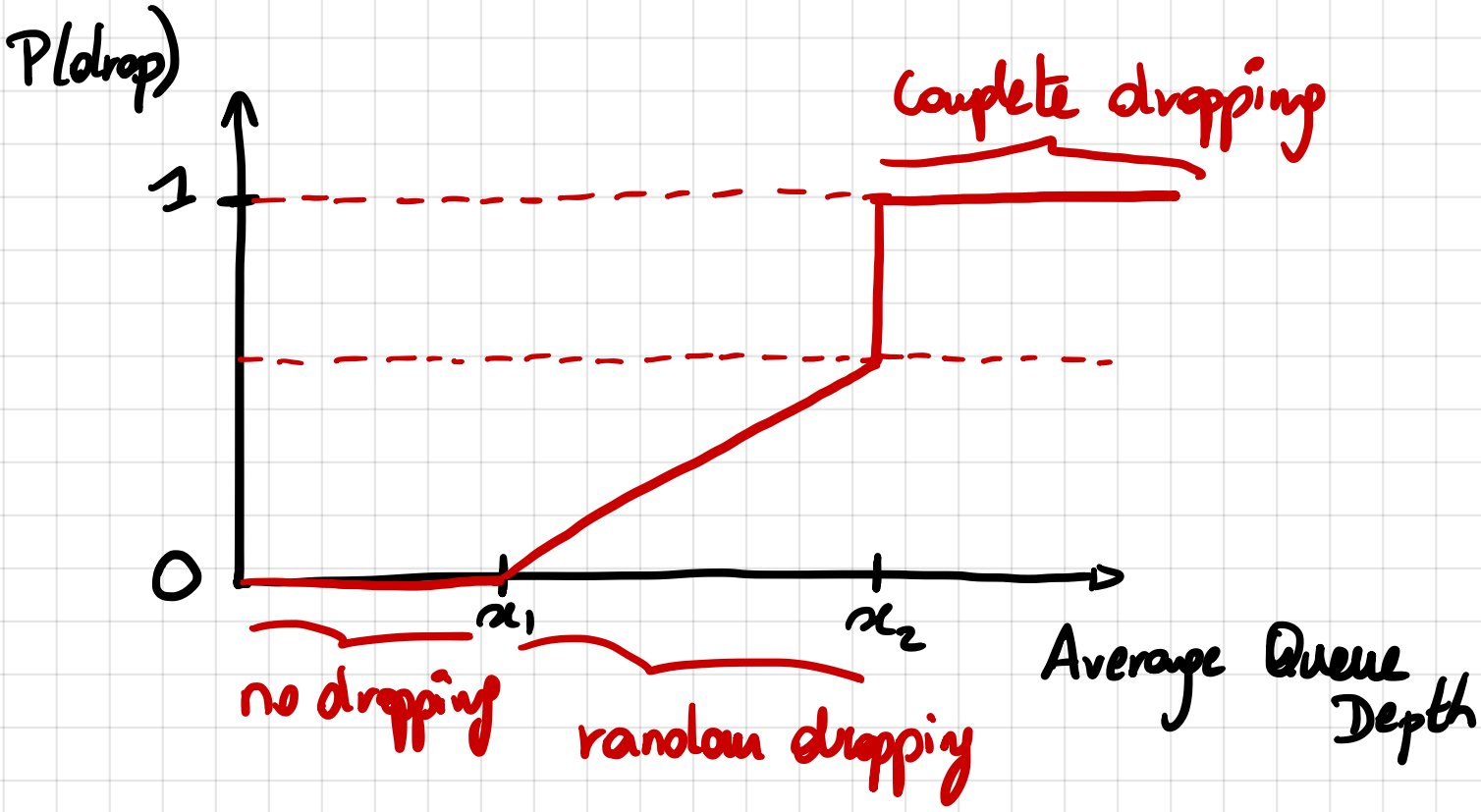
   Pros: dead simple to implement.

   Cons: tends to synchronize the TCP flows when multiple packets are lost.

2. Random Early Detection (RED):

   RED pre-emptively drop packets before the buffer/queue becomes completely full. It does so using a simple predictive model.

RED (continued):



x_1, x_2 are configured by the operator.

Pros:

RED is more fair than Tail Drop as the more a source transmits, the higher the probability its packets are dropped.
The randomness of the procedure also avoids global synchronization.

Cons: Harder to implement.

Classification

The result of classification is typically a queue that the packet is enqueued on.

In theory, one can classify packets s.t. each connection gets its own queue.

↳ prohibitive because of the large # of dynamically varying queues that the router needs to keep track of.

↳ In practice, routers are equipped with a (small) fixed number of queues and configure rules to map packet/flows to queues.

A classified queue can be fine-grained, comprising packets of only a small # of transport flows, or coarse-grained, comprising pkts from many flows.