Harry Langford
hjel2@cam.ac.uk

1. How does the equation for being a *work-conserving* scheduler capture the idea that you shouldn't sit idle when there is work to be done. (This is best answered by explaining the terms p, q, ρ, λ; and by considering what a change in each parameter might correspond to in terms of a change to the way a packet scheduling algorithm chooses which packet to send next)

   You can do this now!

2. What does the *max-min* fair share criterion attempt to recognise as desirable behaviour? Why is the formula for this so much more complex than saying that each of N flows gets $\frac{1}{N}$ of the bandwidth each?

   The *max-min* fair share criterion attempts interprets fairness as 'every flow either gets everything it wants – or no flow got more than it did'. This means that flow is shared equally between all flows, with the caveat that if some flows don't *want* all that capacity, their unwanted capacity is split between the flows which *do* want it.

   The formula is more complicated saying that each of N flows should get $\frac{1}{N}$ of the bandwidth because such an allocation would waste bandwidth if flows did not want as much as $\frac{1}{N}$ of the capacity.

3. For each of the FCFS, GPS, WRR, DeficitRR and WFQ, state whether the algorithm is *work-conserving* and/or *max-min* fair.

   Property 1 (FCFS)**.** *work-conserving* but not *max-min* fair

   Property 2 (GPS)**.** *work-conserving* and *max-min* fair.

   Property 3 (WRR)**.** *work-conserving* but not *max-min* fair – weights are *intrinsically unfair.*

   Property 4 (DeficitRR)**.** *work-conserving* but not *max-min* fair. Only *max-min* fair if all flows have the same packet size all flows have constant packet sizes and the weight of a flow is the inverse of the packet size.

   Property 5 (WFQ)**.** *work-conserving* but not *max-min* fair – since weights are *intrinsically unfair.*

4. (a) *Tail-Drop* is a common way to handle buffer overflow at a router. Why?

   Tail-Drop is the simplest algorithm to handle buffer overflow – and it usually works quite well.

   It can penalise bursty flows – or flows which are synchronised.

   (b) What advantage does *Head-Drop* offer?

   Head-Drop offers two advantages over *Tail-Drop*, both revolving around dropping the oldest packet. Firstly, if the packet at the head of the list is dropped, then TCP connections will observe dup-ACSs earlier and so decrease the window size earlier. Secondly, on a real-time system, the packet at the head of the queue is the most likely to be "expired" *i.e.* information about a video frame which has already been played and interpolated.

   However, *Head-Drop* is empirically awful. This is because the router is dropping the packet it's put the most time into. This doesn't decrease the total amonut of work it has to do – it just decreases the amount of work that it's done! This is analagous to being overworked and dropping the courses you're closeset to completing! It leads to a positive feedback loop where the amount of useful work that rounters do is minimised.

   (c) Why might RED perform better than either *Tail-Drop* or *Head-Drop*? Why might this not work out in practice?

Both *Tail-Drop* and *Head-Drop* are protocols which work 'on the cliff' *i.e.* they wait for the network to become congested and then recover. Whereas RED operates in the neck. It attempts to ensure that the router never becomes overwhelmed. This means that the queue is kept short and so the latency of the network is lower.

However, since RED uses packet loss as a notification (rather than marking packets like DECbit or ECN), it forces packet loss. Consider a bursty flow which *does not* overflow the buffer. Under *Head-Drop* or *Tail-Drop*, it would fill the buffer and then the buffer would slowly empty without any packet loss. However, RED would induce packet loss – reducing the congestion window and *increasing* the latency…