# Router Assisted Congestion Control

- Routers could drop a packet before the queues become empty to signal congestion
- Or could mark a congestion bit in the IP header to note congestion is happening
  - TCP algorithms can then use this to essentially treat the packet as dropped (but still get the data)
- This is known as **queue management**
- Routers can also help schedule packets to determine which ones should be sent on the link
  - This is useful for actually allocating different bandwidths to different people

# Queue Management

- To measure the current size of the queue, we use an exponential weighted moving average
  - This helps deal with the noisy nature of the queue size and handling bursts of data
- We can use this meausre and use randomness based on this to determine which packets to drop
  - We will find a fundamental problem with this later, but it is useful to look at
- We can try a function that maps queue length to probability to drop
  - First idea: linear
    - First intuition that there is a problem with this:
      - We have a problem where this is dependent on our buffer size
      - Our buffer size could be poorly set and be 100 times the bandwidth delay product
    - We also ideally want to have our function near 0 not really change
  - **RED** is an early algorithm with a lot of research / simulations that said to use
    - Problem: it is very difficult to set the parameters of this function
      - Minimum threshold, maximum threshold, drop probability, and exponential weight
    - This is implemented in almost every router, but almost no-one turns it on because they can't tune the parameters
- Since these algorithms are hard to tune, we go in a different direction

---

- Suppose we have $N$ AIMD flows on a link of $\mu$ capacity (in packets / sec)
  - Remember that in TCP Reno, the throughput was prop to $1/(RTT * \sqrt{d})$
  - Ideally, each flow should be getting a capacity of $\mu/N$
  - We want our queue to not blow up in size, which means RTT should be staying the same and we need to increase drop rate instead to allocate the same throughput to each person
    - However, this leads to a fundamental problem with the prev approach because previously the drop rate was only going up when the buffer size increased, which indicated the delay was going up
- Instead, we need to start dropping packets as the number of connections grows instead

## PIE Algorithm

- We will choose a target delay (reference delay)
- We update our drop probability as
  - $p \leftarrow +\alpha(\text{curr delay} - \text{ref delay}) + \beta(\text{curr delay} - \text{old delay})$
  - \text{old_delay} \leftarrow \text{curr delay}
- Essentially runs a PI controller on the current delay versus the target
  - The first term is actually I and the second term is P (expand this out and telescope to see why)
- The E in the PIE algorithm name comes from "enhanced"
  - The algorithm scales up $\alpha$ and $\beta$ depending on how big the current loss rates are

# XCP (Explicit Congestion Control)

- This is not widely deployed

- Uses an idea of dynamic packet state
  - You had some data in the header that all routers along the path could update, and this would tell the receiver what to do
  - It stores the:
    - cwnd
    - rtt
    - feedback
- cwnd and rtt are set by the sender
- Each router will independently based on their congestion levels figure out how to set the feedback
  - Feedback = how to change the cwnd
  - They will then look at the feedback from previous routers and if they have a less radical change than the previous routers, they do not update
  - I.e. if previous router said to increase by 2, and this router says to increase by 3, then it will not do anything
- By the time this gets back to the sender, it will receive the bottleneck feedback
- The main problem with trying to do this: how to do it without maintaining per-router flow
  - On high-speed routers, maintaining per flow state is really hard
  - Also encryption would make it not feasible to peek into the headers
  - In networking, stateless is most of the time used to refer to no per flow state (very different)
- To do so, we compute this thing called efficiency:
  - A linear combination of spare capacity and the minimum queue length (persistent queue length) over some time period
  - If it is greater than 0, we think we are uncongested and can increase the window
  - Otherwise, decrease the window
- Tricky thing:
  - We want to achieve AIMD, so we need to for each packet, without maintaing state, figure out how to set it so that we increase the window by $1/RTT$ and decrease by a constant amount per loss
  - Recording cut off here sadge