6.033: Networking - In-network Resource Management
Lecture 11
Katrina LaCurts, lacurts@mit.edu

0. Introduction
  - Last time: TCP CC. Massive success. Doesn't require us to change the network, is something machines can opt-in to (don't have to have reliable transport if you don't need it), lets us prevent congestion in a distributed manner.
  - But:
    - Can result in long delays when routers have too much buffering (Bufferbloat)
    - Doesn't work well in some scenarios (DCTCP)
    - Most important for today: doesn't react to congestion until queues are full.
  - Full queues = long delay
  - Queues = necessary to absorb bursts
  - Goal: Transient queues, not persistent queues
  - Idea: drop packets *before* the queues are full. TCP senders will back off before congestion is too bad.

1. DropTail
  - The original queue management scheme. When a packet arrives, if the queue is full, drop it; else, enqueue it.
  - Simple (+)
  - Only drops packets when it needs to (+/-)
    - Remember: dropped packet => retransmission, which wastes resources
  - Synchronizes sources (-)

    Consider the following scenario, where one source sends a burst of traffic: x x x x [|x|x|x|x]

    Queue will drop three packets at the tail of the burst. TCP sender will (likely) timeout, drop its window to 1.

    If multiple senders do this: all sources bursts, packets dropped from all, all sources throttle back (reduces utilization), sources increase, cycle repeats.

    Flow synchronization = decreased utilization

  - Not very fair (-)
  - Tends to result in mostly-full queues (-)
  - Bad for bursty traffic (-)

2. RED
  - Active queue management scheme
  - Idea: drop packets before the queue is full to give senders an early signal

- Requires a measure of the average queue size, q_avg.
  - q_avg = a*q_instant + (1-a)*q_avg ; 0 < a << 1
- Drop packets with probability p. What is p?
  - q_avg <= min_q; p = 0
  - min_q < q_avg <= max_q; p increases linearly
  - q_avg > max_q; p = 1

  (see slides for diagram)
- Results:
  - Queue length doesn't oscillate as much (+)
    - Because q_avg is a low-pass filter, and because of the next point
  - Smooth change in drop rate with congestion (+)
    - As q_avg increases, so does p. Keeps q_avg stable
  - Flows are desynchronized (+)
    - Spreads the drops out
- But, it still drops packets (-)

3. ECN
   - RED, but "mark" packets instead of dropping them
     - "Mark" = set a bit in the header to 1. Sources learn about congestion via marked ACKs
   - Seems great! But sources have to know to do this. They already know to react to packet drops, but not to marks.

4. RED/ECN vs. DropTail
   - Advantages of RED/ECN
     - Smaller persistent queues => smaller delays
     - Less dramatic queue oscillation
     - Less biased against bursty traffic (in theory)
   - Disadvantages
     - More complex
     - Hard to pick parameters (q_min, q_max, etc.)
       - "Right" parameters depend on number of flows, bottleneck, etc.
       - Bad parameters make things worse
   - Neither RED nor ECN are the final word on active queue management

5. Traffic Differentiation
   - As long as we're changing the switches themselves, why stop at queue management?
   - Idea of traffic differentiation: put different types of traffic in different queues, and do something fancy with the queues.

6. Delay-based scheduling
   - Suppose we want to prioritize latency-sensitive traffic. Say, xbox live traffic (latency-sensitive) over email (not)
   - Solution: priority queueing
     - Two queues: xbox queue, email queue. Serve xbox queue if it has a packet. If not, serve email queue.
     - (Can extend this idea to more than two queues)

- "What queue to send a packet from" is the problem of scheduling. That's different from queue management: "When to drop/mark packets in a single queue"
- Lingering problem: a lot of xbox traffic => starving out the email traffic. We'll come back to that.

7. Bandwidth-based scheduling
   - What if we, instead, want to allocate a certain amount of bandwidth to each queue?

8. Round-robin

   (Note: in class, all of my examples used Skype/Spotify and Dropbox. Below you have the same examples, just with different apps.)

   - First case: want xbox and email traffic to each get 50% of bandwidth
   - Solution: round-robin scheduler
     - Take a packet from the xbox queue, then the email queue, then the xbox queue, then the email queue, ...
   - But, what if packet sizes are different:

     ```
     xbox:  [ 10  | 10  | 10  | 10 ]
     email: [ 100 | 100 | 100 | 100 ]
     ```

     With this scheme we'll send 10 bytes of xbox traffic for every 100 bytes of email traffic. Not what we want!
   - => Can't handle variable packet sizes (-)
   - Also, in its purest form, RR doesn't allow us to weight traffic differently (e.g., 66% xbox 33% email instead of a 50/50 split)

9. Weighted RR
   - Take the weights, but factor packet size in as well.
   - Algorithm:

     ```
     in each round:
       for each queue q:
         q.norm = q.weight / q.mean_packet_size
       min = min of q.norm’s over all flows
       for each queue q:
         q.n_packets = q.norm / min
         send q.n_packets from queue q
     ```

   - Example 1:

     ```
     xbox:  [ 10  | 10  | 10  | 10 ]
     email: [ 100 | 100 | 100 | 100 ]

     xbox.weight = 2/3      email.weight = 1/3   <-- normalize
                                                     weights
     ```

```
  xbox.mean = 10         email.mean = 100     <-- mean packet size
  xbox.norm = 2/3/10     email.norm = 1/3/100
            = 1/15                  = 1/300

              min norm = 1/300

  xbox.packets = 1/15/(1/300)  email.packets = 1/300/(1/300)
               = 20                          = 1
```

So we send 20 packets = 20*10 bytes = 200 bytes of xbox traffic for every 1 packet = 1*100 bytes = 100 byets of email traffic.

- Example 2:

```
  xbox:  [ 5 | 5 | 10 | 10 ]
  email: [ 1 | 1 | 1 | 1 ]

  xbox.weight = 2/3    email.weight = 1/3
  xbox.mean = 7.5      email.mean = 1
  xbox.norm = 4/45     email.norm = 1/3

               min norm = 4/45

  xbox.packets = 1     email.packets = 3-4
```

So for every 3-4 bytes of email, we'll send 5-10 bytes of xbox. Not quite what we want..

- Also: how do we calculate mean packet size? Over last n packets? Over all packets ever?

10. Deficit round-robin
  - Queues accumulate "credit" which specifies how many bytes they're allowed to send in the next round. Credit carries over to handle larger packet sizes.
  - Algorithm:

```
  in each round:
    for each queue q:
      q.credit += q.quantum
      while q.credit >= size of next packet p:
        q.credit -= size of p
        send p
```

  - Example 1:

```
  xbox:  [10 | 10 | 5 | 5 | 10 | 10]
  email: [10 | 10 | 10 | 10 | 10 | 10]

  xbox.Quantum = 20     <-- note: 20;10 not 2/3;1/3 (see below)
```

```
    email.Quantum = 10
    xbox.credit = 0
    email.credit = 0

    round 1:
    xbox.credit += xbox.Quantum = 20
    while xbox.credit > next packet size:
        send next packet
        decrement packet size from credit
    => we'll send 2 xbox packets, and xbox.credit = 0
       xbox queue is now: [10 | 10 | 5 | 5]

    email.credit += email.Quantum = 10
    => we'll send just the first packet, and email.credit = 0
       email queue is now [10 | 10 | 10 | 10 | 10]

    round 2:
    xbox.credit += 20 = 20
    => have enough credit to send the next three packets
       xbox.credit = 0
       xbox.queue = [10]

    email.credit += 10
    => have enough credit to send next packet
       email.credit = 0
       email.queue = [ 10 | 10 | 10 | 10 ]

    So we sent 20 bytes for every 10 bytes of email, even with
    variable packet sizes within the queue.
```

- Quantums are larger because they reflect a packet size
  - Small quantums: go through a lot of rounds before sending a packet
  - Large quantums: potentially send a lot of packets from one queue before moving onto the next

- Example 2:

```
  xbox = [ 20 | 750 | 200 ]   xbox.Quantum = 500
  email = [ 500 | 500 ]       email.Quantum = 500

  round 1:

  xbox.credit = 500
  can send first packet; xbox.credit = 300
  cannot send next packet

  email.credit = 500
  can send first packet; email.credit = 0
```

round 2:

xbox.credit = 300 + 500 = 800 <-- credit carries over!
can send first packet; xbox.credit = 50
can send second packet; xbox.credit = 30

email.credit = 500
can send first packet; email.credit = 0

- Credit carrying over helps deal with variable (and large) packet sizes)
- Pros of DRR:
  - Don't need mean packet size
  - Give near-perfect fairness (we won't prove this)
  - O(1) packet processing
- In fact: schemes that increase fairness also increase packet processing.

11. Discussion
  - Traffic differentiation: a good idea? In theory, sure. But:
    - Hard to decide what granularity of isolation makes sense (per-app? per-flow?)
      - per-app also requires deep packet inspection. Expensive and thwarted by encryption.
      - per-flow = lots of state.
    - For fair queueing:
      - Schemes (except deficit RR) are expensive
      - Have to change switches
      - How to you choose which traffic gets priority? And who should make that decision?
    - For priority queueing:
      - Unclear how multiple methods of priority queueing would interact across the Internet
    - *Should* we allow traffic to be prioritized at all?
    - Depressing conclusion: there's enough bandwidth that usually a single FIFO queue works fine :/
  - Queue-management: a good idea? Again, in theory, yes.
    - In fact, RED/ECN -- or their ideas -- are used in some environments (DCTCP)..
    - ..But not on the entire Internet
      - Hard to set parameters
      - Hard to figure out interactions between schemes
      - Have to change switches
  - In-network resource-management: a good idea?
    - Should we do any of this? Who should make these decisions? Should the network "help" the endpoints, possibly providing better performance, but also possibly providing unnecessary functionality?