### 7.1 Output Scheduling for QoS

- Single-resource (≠ crossbar) scheduling for advanced QoS
- Work-Conserving Scheduling – Delay Conservation Law
  - you can favor (delay-wise) some flows only at the expense of other flows
- Series composition: Policer, Regulator (Shaper), Scheduler
- Hierarchical comp.: schedule among, then within Flow Aggregates
- Strict Priority Scheduling (static sequence) – danger of starvation
- Round-Robin (RR) Scheduling (circular sequence)
  - Max-Min Fairness: equal “shares”, equally allocate unused BW to all others
- Weighted Round Robin (WRR), Weighted Fair Queueing (WFQ)
  - allocate throughput in proportion to arbitrary “weight factors”
  - smoothness of allocation – static (periodic) schedules, dynamic schedules
Delay Conservation Law – Sketch of Proof

- Plot “Cumulative Byte Arrivals”, \( A(t) \), and “Cumulative Byte Departures”, \( D(t) \), as functions of time, like we did in §1.1.3
- Departures curve, \( D(t) \), is independent of scheduling policy:
  - Work-Conserving Scheduling means departure rate = maximum link rate at any time there is a backlog, i.e. whenever \( D(t) < A(t) \)
- Delay of a packet = \( t_{\text{departure}} - t_{\text{arrival}} \)
  - for FIFO scheduling: \( D(t_{\text{departure}}) = A(t_{\text{arrival}}) \)
- Express the area between \( A(t) \) and \( D(t) \) as a sum of packet delays:
  - under FIFO: sum of areas of horizontal slices; delays weighted by pck size
  - exchange the departure order of two bytes: individual byte delays change, but their sum does not ⇒ total area and sum of byte delays is invariant wrt. scheduling policy (careful when translating byte delays to packet delays)
- Divide by time to translate cumulative bytes into average rates
  - \( \sum_{\text{delays}_{\text{FIFO}}} = \sum_{\text{delays}_{\text{flow1}}} + \sum_{\text{delays}_{\text{flow2}}} + \ldots + \sum_{\text{delays}_{\text{flowN}}} \)
  - \( \sum_{\text{delays}} = \text{cumBytes} \times \text{avgDelay}; \text{cumBytes} = \text{timeWindow} \times \text{avgRate} \)
### Conceptual Stages of Scheduling

- **Policer**
  - Mark or Drop non-Conforming packets
  - Some flows may not be subject to regulation (e.g., best-effort, non-real-time)

- **Regulator**
  - Delay packets so that they conform to contract (Non-Work-Conserving)
  - Not all stages are always present (e.g., police or regulate, but usually not both)
  - Can have a scheduler without a regulator

- **Scheduler**
  - Select among competing eligible packets (Work-Conserving)
  - Multiple flows may have a single (Conceptual) queue at the scheduler level
  - Implementation usually involves a single set of queues for both R & S

---

### Composite Schedulers: Aggregation & Hierarchy

- Individual Scheduler Policies:
  - Strict (static) Priorities
  - Round-Robin
  - Static Schedule (computed off-line)
  - Dynamic Schedule (WRR - Weighted Round Robin)
  - Other
Strict (Static) Priority Scheduling

Eligibility Flags

- Flow or Aggregate 1
- Flow or Aggregate 2
- Flow or Aggregate 3
- Flow or Aggregate 4
- Flow or Aggregate 5
- ...

Serve the highest-priority eligible flow or aggregate

Implementation:

- Use a priority enforcer/encoder: chain of elements with a ripple signal. "Nobody above is eligible". To speed-up the ripple signal, use ideas analogous to carry lookahead: a tree of OR-gates detects the presence of eligible entries among N entries in time ~ log N ...

7.1 output Scheduling for QoS

Starvation issue w/ strict (static) priorities:
- If level (flow) i is not policed or regulated and becomes "persistent" (i.e., always has a non-empty eligible queue), then all levels below i will be starved.
- Normally, ensure that all levels but the last one are policed or regulated.

Composition idea: Change the order of priorities in different time slots of an (offline computed) schedule.

Example: Customer A bags 50% of my throughput
- B = 25%

Other customers, C, shone whatever is left over with A and B

\[ \text{RR} = \text{round-robin among } A - B - C \]

Periodic Schedule

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>RR</td>
<td>RR</td>
<td>RR</td>
<td>RR</td>
</tr>
</tbody>
</table>
7.1 output Scheduling for QoS
Comments on Re-Insertion Point for newly-Eligible Flows

- Let us call "uncongested flows" the flows whose bottleneck is not this network link – their bottleneck may be their source (end-to-end flow control) or another network link (either a link upstream of this link, or a downstream link but with hop-by-hop flow control). Uncongested flows usually have (almost) empty queues, because these queues are served (emptied) more frequently than they are filled. Newly arriving cells or packets will usually be inserted into empty queues, causing the flow to re-become eligible. Then, the queue will be served before a second cell or packet arrives in it, causing the queue to re-become empty and the flow to become ineligible.

- Insertions (b) penalize the uncongested ("well behaved") flows by causing them to undergo the worst-case delay, while this yields no appreciable gain for the congested flows: congested flows undergo a very long delay anyway – what matters for these latter flows is throughput, not delay. Insertions (c) offer only a 50% (average) improvement over (b) for uncongested flows.

- An alternative is to use insertions (a) when we have verified that the flow is uncongested, else use insertions (b) or (c)? when it looks like the flow is congested. To verify that the flow is well behaved (uncongested), we need to maintain per-flow last-service timestamps. → [text continued on next slide] →

Max-Min Fairness

- Equally distribute link throughput among all flows on this link – determines the link’s "fair share"
- Flows bottlenecked elsewhere use up less than their fair share
- Equally distribute unused throughput among all remaining flows – increases this link’s fair share ⇒ the bottleneck of some flows may shift elsewhere ⇒ equally reallocate unused throughput, and so on and so forth ⇒ distributed process to determine max-min equilibrium (does it oscillate???)
7.1 output Scheduling for QoS

Weighted Round Robin (WRR) Service Schedules

Two extremes of schedule style:

1. Bursty Service:
   - Hard to implement.
   - Simple: flow service defined by periodic service schedule.
   - Like round robin, but on each visit to flow, serve a number of packets (bytes) proportional to the flow's weight factor.

2. Smooth Service - minimize service time jitter:
   - Hard to implement.
   - Hard to transform eligibility flags in priority circuits or re-insert in multiple positions in circular linked lists.

   Implementation:
   - Hard to transform eligibility flags in priority circuits or re-insert in multiple positions in circular linked lists.
   - Set of eligible flows varies slowly = compute schedule offline.
   - Set of eligible flows varies fast = recompute schedule online or flow weights change often = via priority queue.

Priority Queue: (quite) smooth WRR scheduling

- Maintain a (rarely) set of eligible flows.
- Associate an 'next service (virtual) time' with each of them.
- Find and serve the (eligible) flow that has the minimum (earliest) next service time.
- Recompute for a future time for that flow served.

Example:

<table>
<thead>
<tr>
<th>Flow</th>
<th>Eligibility</th>
<th>Weight</th>
<th>Service Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Yes</td>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>Yes</td>
<td>30</td>
<td>33</td>
</tr>
<tr>
<td>C</td>
<td>Yes</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>Yes</td>
<td>10</td>
<td>100</td>
</tr>
</tbody>
</table>

(b) set of eligible flows varies fast = recompute schedule online or flow weights change often = via priority queue.
WRR via Priority Queue: Real or Virtual "Time"?

Example:
Flow Group A
High Priority
Already Policed

Flow Group B
Low Priority
Intended to Absorb all remaining Capacity

(a) soon after it became ineligible
Regenerate
reinsert here (not earlier)

(b) long after it became ineligible
do NOT reinsert: "in the past"
reinsert approx. here

Reinsertion Time
Next Service Time Computation

Many Variants:
- Weighted Fair Queuing (WFQ)
- Self-Selected Fair Queuing (SSFQ)
- Weighted Fair Weighted Fair Queuing (WFQ2)
- Start-Time Fair Queuing (STFQ)
- Virtual Clock
Beware: multiple low-rate flows may create large jitter for high-rate flows.

Example:
- F0: weight = 50, service interval = 20
- F1, F2, F3, F4, F5: weight = 10 (each) + (50), service interval = 100

Case A: with favorable initialization:

Case B: with unfavorable initialization:

Solution: Hierarchical Scheduler. Focus on the aggregate of flows that have (approximately) the same weight and treat them round-robin inside it.

Leaky Bucket implemented using Priority Queue

1. Straightforward implementation:
   - Store the current credit count per flow and update it every 1/R time. It may be too much work, too often, for all flows.
   - Alternative implementation:
     - For each flow, store a past credit count, \( c_{i,t-1} \), with its timestamp, \( t \).
     - Only look at them and update them on packet arrivals and departures:
       \[
       \text{current credits} = \min \left( B_i, c_{i,t} + \lambda_i \cdot (t_{\text{now}} - t) \right)
       \]
     - After each packet departure, compute after how long the next packet will have sufficient credits for departure, and insert it at that “time” in the scheduler’s priority queue.

Priority Queue implementation:

- Heap
- Calendar Queue

See references in Reading List and Web.

http://archvlsi.ics.forth.gr/~kateveni/534/