Packet Scheduling: Problem Overview

• When to send packets?
• What order to send them in?
Approach #1: First In First Out (FIFO)

- Packets are sent out in the same order they are received
- Benefits: simple to design, analyze
- Downsides: not compatible with QoS
  - High priority packets can get stuck behind low priority packets
Approach #2: Priority Queuing

- Operator can configure policies to give certain kinds of packets higher priority
  - Associate packets with priority queues
  - Service higher-priority queue when packets are available to be sent
- Downside: can lead to starvation of lower-priority queues
Approach #3: Weighted Round Robin

- Round robin through queues, but visit higher-priority queues more often
- Benefit: Prevents starvation
- Downsides: a host sending long packets can steal bandwidth
  - Naïve implementation wastes bandwidth due to unused slots
Overview

• Fairness
• Fair-queuing
• Core-stateless FQ
• Other FQ variants
Fairness Goals

• Allocate resources fairly
• Isolate ill-behaved users
  – Router does not send explicit feedback to source
  – Still needs e2e congestion control
• Still achieve statistical muxing
  – One flow can fill entire pipe if no contenders
  – Work conserving → scheduler never idles link if it has a packet
What is Fairness?

- At what granularity?
  - Flows, connections, domains?

- What if users have different RTTs/links/etc.
  - Should it share a link fairly or be TCP fair?

- Maximize fairness index?
  - Fairness = $(\Sigma x_i)^2/n(\Sigma x_i^2)$  $0<$fairness$<1$

- Basically a tough question to answer – typically design mechanisms instead of policy
  - User = arbitrary granularity
Max-min Fairness

- Allocate user with “small” demand what it wants, evenly divide unused resources to “big” users
- Formally:
  - Resources allocated in terms of increasing demand
  - No source gets resource share larger than its demand
  - Sources with unsatisfied demands get equal share of resource
Max-min Fairness Example

- Assume sources 1..n, with resource demands X1..Xn in ascending order
- Assume channel capacity C.
  - Give C/n to X1; if this is more than X1 wants, divide excess (C/n - X1) to other sources: each gets C/n + (C/n - X1)/(n-1)
  - If this is larger than what X2 wants, repeat process
Implementing max-min Fairness

- Generalized processor sharing
  - Fluid fairness
  - Bitwise round robin among all queues
- Why not simple round robin?
  - Variable packet length → can get more service by sending bigger packets
  - Unfair instantaneous service rate
    - What if arrive just before/after packet departs?
Bit-by-bit RR

- **Single flow**: clock ticks when a bit is transmitted. For packet $i$:
  - $P_i = \text{length}$, $A_i = \text{arrival time}$, $S_i = \text{begin transmit time}$, $F_i = \text{finish transmit time}$
  - $F_i = S_i + P_i = \max (F_{i-1}, A_i) + P_i$

- **Multiple flows**: clock ticks when a bit from all active flows is transmitted → round number
  - Can calculate $F_i$ for each packet if number of flows is known at all times
    - This can be complicated
Approach #4: Bit-by-bit Round Robin

- Round robin through “backlogged” queues (queues with pkts to send)
  - However, only send one bit from each queue at a time
- Benefit: Achieves max-min fairness, even in presence of variable sized pkts
- Downsides: you can’t really mix up bits like this on real networks!
The next-best thing: Fair Queuing

- Bit-by-bit round robin is fair, but you can’t really do that in practice

- Idea: simulate bit-by-bit RR, compute the finish times of each packet
  - Then, send packets in order of finish times
  - This is known as Fair Queuing
What is Weighted Fair Queuing?

• Each flow \( i \) given a weight (importance) \( w_i \)
• WFQ guarantees a minimum service rate to flow \( i \)
  \[ r_i = R \times w_i / (w_1 + w_2 + \ldots + w_n) \]
  • Implies isolation among flows (one cannot mess up another)
What is the Intuition? Fluid Flow

water pipes

water buckets
Fluid Flow System

• If flows could be served one bit at a time:

• WFQ can be implemented using bit-by-bit weighted round robin
  – During each round from each flow that has data to send, send a number of bits equal to the flow’s weight
**Fluid Flow System: Example 1**

<table>
<thead>
<tr>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow 1 ($w_1 = 1$)</td>
</tr>
<tr>
<td>Flow 2 ($w_2 = 1$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packet Size (bits)</th>
<th>Packet inter-arrival time (ms)</th>
<th>Arrival Rate (Kbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flow 1 1000</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Flow 2 500</td>
<td>10</td>
<td>50</td>
</tr>
</tbody>
</table>

Flow 1 (arrival traffic)

Flow 2 (arrival traffic)

Service in fluid flow system

0 10 20 30 40 50 60 70 80

1 2 3 4 5 6

100 Kbps

1 2 3 4 5

1 2 3 4 5 6
Fluid Flow System: Example 2

- Red flow has packets backlogged between time 0 and 10
  - Backlogged flow → flow’s queue not empty
- Other flows have packets continuously backlogged
- All packets have the same size
Implementation in Packet System

• Packet (Real) system: packet transmission cannot be preempted. Why?

• Solution: serve packets in the order in which they would have finished being transmitted in the fluid flow system
Packet System: Example 1

- Select the first packet that finishes in the fluid flow system
Packet System: Example 2

Service in fluid flow system

Packet system

- Select the first packet that finishes in the fluid flow system
Implementation Challenge

• Need to compute the finish time of a packet in the fluid flow system...
• ... but the finish time may change as new packets arrive!
• Need to update the finish times of all packets that are in service in the fluid flow system when a new packet arrives
  – But this is very expensive; a high speed router may need to handle hundred of thousands of flows!
Example

- Four flows, each with weight 1

Finish times computed at time 0

Finish times re-computed at time $\epsilon$
Approach #5: Self-Clocked Fair Queuing

Virtual time

Real time (or, # bits processed)

Output queue

A 9 8 7 6 5 4 3 2 1

4 3 2 1

2 1
Solution: Virtual Time

- Key Observation: while the finish times of packets may change when a new packet arrives, the order in which packets finish doesn’t!
  - Only the order is important for scheduling

- Solution: instead of the packet finish time maintain the \textit{round \#} when a packet finishes (\textit{virtual finishing time})
  - Virtual finishing time doesn’t change when a packet arrives
• Suppose each packet is 1000 bits, so takes 1000 rounds to finish
• So, packets of F1, F2, F3 finishes at virtual time 1000
• When packet F4 arrives at virtual time 1 (after one round), the virtual finish time of packet F4 is 1001
• But the virtual finish time of packet F1, 2, 3 remains 1000
• Finishing order is preserved
System Virtual Time (Round #): V(t)

- V(t) increases inversely proportionally to the sum of the weights of the backlogged flows.
  - During one tick of V(t), all backlogged flows can transmit one bit.
- Since round # increases slower when there are more flows to visit each round.
Is Fair Queuing perfectly fair?

• No. Example: Once we begin transmission of a packet, it’s possible a new packet arrives that would have a smaller finishing time than the current packet
  – FQ is non-preemptive, so keep transmitting current packet

• However, if a packet is sitting in an output queue with its finish time calculated, and a new packet arrives with a sooner finish time, the new packet will be sent first
Fair Queueing Implementation

• Define
  - \( F_{ik} \) - virtual finishing time of packet \( k \) of flow \( i \)
  - \( a_{ik} \) - arrival time of packet \( k \) of flow \( i \)
  - \( L_{ik} \) - length of packet \( k \) of flow \( i \)
  - \( w_i \) - weight of flow \( i \)

• The finishing time of packet \( k+1 \) of flow \( i \) is
  \[
  F_{ik}^{k+1} = \max( V( a_{ik}^{k+1} ), F_{ik}^k ) + L_{ik}^{k+1} / w_i
  \]

• Smallest finishing time first scheduling policy
Properties of WFQ

• Guarantee that any packet is transmitted within \( \text{packet\_length/link\_capacity} \) of its transmission time in the fluid flow system
  – Can be used to provide guaranteed services

• Achieve fair allocation
  – Can be used to protect well-behaved flows against malicious flows
Fair Queuing Tradeoffs

• FQ can control congestion by monitoring flows
  – Non-adaptive flows can still be a problem – why?

• Complex state
  – Must keep queue per flow
    • Hard in routers with many flows (e.g., backbone routers)
    • Flow aggregation is a possibility (e.g. do fairness per domain)

• Complex computation
  – Classification into flows may be hard
  – Must keep queues sorted by finish times
  – Finish times change whenever the flow count changes
Overview

- Fairness
- Fair-queuing
- Core-stateless FQ
- Other FQ variants
Core-Stateless Fair Queuing

- Key problem with FQ is core routers
  - Must maintain state for 1000’s of flows
  - Must update state at Gbps line speeds
- CSFQ (Core-Stateless FQ) objectives
  - Edge routers should do complex tasks since they have fewer flows
  - Core routers can do simple tasks
    - No per-flow state/processing → this means that core routers can only decide on dropping packets not on order of processing
    - Can only provide max-min bandwidth fairness not delay allocation
Core-Stateless Fair Queuing

- Edge routers keep state about flows and do computation when packet arrives
- DPS (Dynamic Packet State)
  - Edge routers label packets with the result of state lookup and computation
- Core routers use DPS and local measurements to control processing of packets
Edge Router Behavior

• Monitor each flow $i$ to measure its arrival rate ($r_i$)
  – EWMA of rate
  – Non-constant EWMA constant
    • $e^{-T/K}$ where $T =$ current interarrival, $K =$ constant
    • Helps adapt to different packet sizes and arrival patterns

• Rate is attached to each packet
Core Router Behavior

• Keep track of fair share rate $\alpha$
  – Increasing $\alpha$ does not increase load ($F$) by $N \times \alpha$
  – $F(\alpha) = \Sigma_i \min(r_i, \alpha) \rightarrow$ what does this look like?
  – Periodically update $\alpha$
  – Keep track of current arrival rate
    • Only update $\alpha$ if entire period was congested or uncongested

• Drop probability for packet $= \max(1-\alpha/r, 0)$
F vs. Alpha

- F vs. \( C \) [linked capacity]
- New alpha
- Old alpha
Estimating Fair Share

• Need $F(\alpha) = \text{capacity} = C$
  – Can’t keep map of $F(\alpha)$ values → would require per flow state
  – Since $F(\alpha)$ is concave, piecewise-linear
    • $F(0) = 0$ and $F(\alpha) = \text{current accepted rate} = F_c$
    • $F(\alpha) = F_c / \alpha$
    • $F(\alpha_{\text{new}}) = C \Rightarrow \alpha_{\text{new}} = \alpha_{\text{old}} * C / F_c$

• What if a mistake was made?
  – Forced into dropping packets due to buffer capacity
  – When queue overflows $\alpha$ is decreased slightly
Other Issues

• Punishing fire-hoses – why?
  – Easy to keep track of in a FQ scheme

• What are the real edges in such a scheme?
  – Must trust edges to mark traffic accurately
  – Could do some statistical sampling to see if edge was marking accurately
Overview

- Fairness
- Fair-queuing
- Core-stateless FQ
- Other FQ variants
Stochastic Fair Queuing

- Compute a hash on each packet
- Instead of per-flow queue have a queue per hash bin
- An aggressive flow steals traffic from other flows in the same hash
- Queues serviced in round-robin fashion
  - Has problems with packet size unfairness
- Memory allocation across all queues
  - When no free buffers, drop packet from longest queue
Deficit Round Robin

- Each queue is allowed to send $Q$ bytes per round
- If $Q$ bytes are not sent (because packet is too large) deficit counter of queue keeps track of unused portion
- If queue is empty, deficit counter is reset to 0
- Uses hash bins like Stochastic FQ
- Similar behavior as FQ but computationally simpler
  - Bandwidth guarantees, but no latency guarantees
Deficit Round Robin Example

Quantum Size = 1000

1. Increment deficit counter by Quantum Size
2. Send packet if size is greater than deficit
3. When you send a packet, subtract its size from the deficit

Outbound queue

Matthew Caesar (caesar@uiuc.edu)
Self-clocked Fair Queuing

- Virtual time to make computation of finish time easier
- Problem with basic FQ
  - Need be able to know which flows are really backlogged
    - They may not have packet queued because they were serviced earlier in mapping of bit-by-bit to packet
    - This is necessary to know how bits sent map onto rounds
    - Mapping of real time to round is piecewise linear \( \rightarrow \) however slope can change often
Self-clocked FQ

- Use the finish time of the packet being serviced as the virtual time
  - The difference in this virtual time and the real round number can be unbounded

- Amount of service to backlogged flows is bounded by factor of 2
Start-time Fair Queuing

- Packets are scheduled in order of their start not finish times
- Self-clocked $\rightarrow$ virtual time = start time of packet in service
- Main advantage $\rightarrow$ can handle variable rate service better than other schemes
Mobility models

CS 598: Advanced Internetworking
Matthew Caesar
March 3, 2011
Entity model: Random Walk

• A mobile node moves from its current location to a new location by randomly choosing a direction and speed in which to travel.

• Random Walk is a memoryless mobility pattern. This characteristic can generate unrealistic movements such as sudden stops and sharp turns.
Random Walk Example

Figure 1: Traveling pattern of an MN using the 2-D Random Walk Mobility Model (time).
• The Random Waypoint Mobility Model includes pause times between changes in direction and/or speed.
  – A mobile node stays in one location for a certain period of time (i.e., a pause time).
  – Once this time expires, the node chooses a random destination in the simulation area and a speed that is uniformly distributed between \([\text{minspeed}, \text{maxspeed}]\). The node then travels toward the newly chosen destination at the selected speed.
  – Repeat above two steps

• Often in the model, the nodes are initially distributed randomly around the simulation area. This initial random distribution of MNs is not representative of the manner in which nodes distribute themselves when moving.
Figure 3: Traveling pattern of an MN using the Random Waypoint Mobility Model.
Other variants

• Restricted Random Waypoint Model
  – Observation: on earth, there are obstacles to node movement
    • E.g., Buildings, trees
    • Nodes cannot walk through these obstacles
  – Place a set of obstacles
  – Choose waypoint direction randomly, but truncate length to avoid going through an obstacle

• The Reference Point Group Mobility (RPGM) model
  – Observation: in practice, nodes move as groups
    • E.g., cell phones on a train
  – Nodes associated into groups, groups move collectively
  – Individual nodes move around with small offsets to the group’s movement

• City Section Mobility model
  – Observation: users on cars have very specific mobility pattern
    • Eg., can’t go faster than car in front of you, cars collectively slow down/speed up, cars traverse grid-like pattern of streets
  – Nodes move in car-like patterns
Challenges with mobility models

- Distributions of node speed, position, distances, etc change with time
- E.g., random waypoint:
Challenges with mobility models

- Distributions of node speed, position, distances, etc change with time
  - E.g., distribution of node position under random waypoint:

![Time = 0 sec](image1.png)  ![Time = 2000 sec](image2.png)
Finishing up DHTs

CS 598: Advanced Internetworking
Matthew Caesar
March 3, 2011
Security issues

• Sybil attacks
  – Malicious node pretends to be many nodes
  – Can take over large fraction of ID space, files

• Eclipse attacks
  – Malicious node intercepts join requests, replies with its cohorts as joining node’s fingers

• Solutions:
  – Perform several joins over diverse paths, PKI, leverage social network relationships, audit by sharing records with neighbors
One-hop DHTs

• Idea: maintain global state of all nodes
  – Might get this for free (link state routing)
  – Hash over all visible nodes

• Benefits:
  – Reduces number of hops to reach a key
  – “Worth it” when node lifetimes
    weeks/months, when hundreds/thousands
    of lookups/second per node
  – Used in Amazon dynamo, cluster load
    balancing
Consistent Hashing: Background

• Hash table: maps identifiers to keys
  – Hash function used to transform key to index (slot)
  – To balance load, should ideally map each key to different index

• Distributed hash tables
  – Stores values (e.g., by mapping keys and values to servers)
  – Used in distributed storage, load balancing, peer-to-peer, content distribution, multicast, anycast, botnets, BitTorrent’s tracker, etc.
Background: hashing

<table>
<thead>
<tr>
<th>keys</th>
<th>function</th>
<th>hashes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ahmed</td>
<td></td>
<td>00</td>
</tr>
<tr>
<td>Yan</td>
<td></td>
<td>01</td>
</tr>
<tr>
<td>John</td>
<td></td>
<td>02</td>
</tr>
<tr>
<td>Viraj</td>
<td></td>
<td>03</td>
</tr>
<tr>
<td></td>
<td></td>
<td>04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>08</td>
</tr>
</tbody>
</table>
• Example: Sum ASCII digits, mod number of bins
• Problem: ________________
Solution: Consistent Hashing

• Hashing function that reduces churn
• Addition or removal of one slot does not significantly change mapping of keys to slots
• Good consistent hashing schemes change mapping of $K/N$ entries on single slot addition
  – $K$: number of keys
  – $N$: number of slots
• E.g., map keys and slots to positions on circle
  – Assign keys to closest slot on circle
Solution: Consistent Hashing

- Slots have IDs selected randomly from [0,100]
- Hash keys onto same space, map key to closest bin
- Less churn on failure → more stable system