Traffic Management
Outline

Application requirements
  Guaranteeing performance
Ways of guaranteeing performance
Leaky Buckets
Scheduling
  Goals:
    Fairness
    Protection
    Ease of implementation
    Ability to guarantee performance
    Efficient admission control
Mechanisms:
  First Come First Served
  Round Robin
  Generalised Processor Sharing
  Fair Queueing
  Performance guarantees
  Limitations ⇒ non-work conserving schedulers

Focus on fair sharing for adaptive applications
Focus on perf. guarantees for non-adaptive apps
Resources

Keshav Chapter 9 (Scheduling), § 13.3.4 (Leaky Buckets), § 14.9 (admission control)


§ 14.4 of Keshav and § 3.1 of RFC1633 proposes a taxonomy of application requirements.
Application requirements

“adaptive”/“best-effort” applications: insensitive to typical variation of network performance, e.g. transfer of filesystem backup, email to opposite side of world
c.f. “non-adaptive”/“real-time” applications,
e.g. effectiveness of voice call degrades with delay>150ms, throughput<2kb/s

Many applications would like to be guaranteed a minimum level of service (and will take more if available):
• slow transfer of web image irritates user (difference from backup is a matter of scale)
• instant messaging

Such applications require/desire performance guarantees “Quality of Service” (QoS)
Performance Parameters

- **Data rate** ("Bandwidth"): Volume per unit time. Meaningless without specifying unit of time. Possibilities include:
  - minimum interarrival time ("Peak rate")
  - some averaging interval (more in Leaky Bucket discussion)

- **Delay**
  - Mean
  - Measures of variability ("jitter"): variance, range, 99th percentile, etc
  - Excessive = loss

Figure from Keshav
Traffic management

**Definition:** Regulating the flow of traffic into points of the network in order to improve the service that the network provides.

**Goals:**
- Fair/equitable allocation to “adaptive” applications
- Performance guarantees for non-adaptive applications
- Low cost (e.g. simple implementation, efficient resource utilisation)
Review

Multiplexing points affect burstiness

Drawing from J. Turner
Multiplexing traffic can cause delay

Input:
- Source A: 2 pkt/2 sec
- Source B: 1 pkt/1 sec
- Source C: 2 pkt/4 sec

Output (FCFS, top-to-bottom): 4 pkt /1 sec

Delay that a stream experiences may vary with load from other streams. Traffic pattern may be distorted through multiplexing with other streams e.g. source B
Network can guarantee service if sources limit traffic

Bursty arrivals impede ability to provide low delay $\Rightarrow$ limit traffic

Sum of mean rates shouldn’t exceed capacity

Smoothed sources reduce delay variation
Traffic management time scales

< round-trip-time, e.g. μs
  • scheduling and buffer management
  • routing datagrams around congested areas

≥ round-trip time, e.g. several ms
  • feedback flow control
  • (re)negotiation of traffic profile

“session” level (aka conversation, call, flow), e.g. minutes
  • admission control
  • routing connections (assuming static)

day – pricing to shift part of peak load to off-peak hours

longer intervals: provisioning – deploying new links

See Keshav Section 14.5
Outline
Service models

Connectionless (details on ensuing slides)
✓ Simple
✗ Poor guarantees

Connection-oriented (details on ensuing slides)
✓ Strong guarantees
✗ Complicated

The key question: Is it “cheaper to build intelligent [connection-oriented] traffic management schemes, or use the same money to increase the raw capacity of the [connectionless] network” [Keshav p. 450]

See Keshav p. 450
Connectionless service

1. Source marks “priority”/class of traffic (e.g. voice=low delay, file transfer=high delay); pushes it into the network.

2. Network attempts to give higher priority traffic preferential service.
   Need disincentives for claiming overly high priority.
   “Priority” here is at the packet level.
   Later: weightings on throughput=prioritisation at flow/conn’n level

3. Sources might inundate network with high-priority traffic ⇒ network can’t guarantee service.

Evaluation:
✓ Simple compatibility with historically connectionless applications
• Works if high-priority traffic can’t overload network (e.g. \( \sum \text{voice/video} \) increasing slower than Ethernet speeds)
✗ Poor guarantees of service.

   e.g. used by Diffserv (connectionless flows, connection-oriented classes)
Connection-oriented

1. Before communication: source/client specifies:
   a. its traffic characteristics, b. desired performance requirement

2. Admission control (aka CAC†):
   a. Network assesses whether it can support this new client while still honouring existing commitments
      • “Guarantees”: Existing commitments + new client < capacity
   b. Network accepts/denies client. If denied, network might be able to indicate:
      • what is available
      • when the requested service would become available (e.g. transfers of known length, e.g. movies or files)

3. Client transmits [Assuming acceptance during admission control]

4. Network may regulate incoming traffic to prevent client that exceeds contract from interfering with other clients.

5. [conceivably] Contract may be renegotiated during the transfer

† Connection / Circuit / Call Admission Control
Pillars for QoS

1. Switches must be able to **classify packets according to priority** to provide appropriate delay and loss.
   
   Possible mechanisms: packets are marked; packets are classified (e.g. based on VCI) to determine class.

2. Need to **isolate classes/flows**, in order to protect well-behaved classes/flows from others.
   
   Possible mechanisms: AQM e.g. RED; Policing/shaping inputs & scheduling flows

3. (as always) Want **efficient** use of resources
   
   Possible mechanisms: packet switching rather than static circuit-like allocation; Non-work-conserving scheduling.

4. **Call admission**: Flow declares its needs, network must be able to block flow if it cannot meet the flow’s needs.
   
   Possible mechanisms: SLAs; connection establishment

Technologies differ in:
- The mechanism used (e.g. Internet tends to use first mechanisms listed before ;)
- Granularity of what is regulated:
  - diffserv: *classes* of service
  - intserv/ATM: individual *flows*
Overall guarantees must be soft (statistical)

Factors that may impinge on service:

• **Demand for resources** may vary due to **competing traffic**
  • “hard”/“deterministic” guarantees of service despite such competition can be made if using appropriate technologies
    • “hard” e.g. “always 10Mb every second”, “never”
    • “appropriate” e.g. connection-oriented technologies

• **Supply of resources** may vary randomly:
  • e.g. noise, component failures, uncertainty from quantum effects
  • Can model these, and dimension system to accommodate their *likely* values.
    • Can only give “soft”/“statistical” guarantees, e.g. 99% chance.

Overall service depends on supply as well as demand ⇒

• Can’t provide a hard overall guarantee.
  • Need only guarantee demand interference < supply unavailability.

Copyright © 2006 Tim Moors
Statistical guarantees

The issue is how confident can the provider be that it will deliver a certain service?

In a crude bottom-line sense: Can the provider reduce Service Level Agreement violations so that they cost less (in terms of refunds and lost customers) than the cost of improving the system?

Fundamental tradeoff: Tighter guarantees require more conservative admission, reducing utilisation (increasing cost/call).

e.g. increasingly aggressive admission control:
1. **Hard guarantee** that competing traffic won’t interfere
2. **Statistical guarantee** that competing traffic, as parameterised during call admission, will cause X% degradation Y% of time.
3. **Empirical/measured admission control**: Network bases admission on *actual* utilisation by existing calls, rather than utilisation claimed through parameters during call admission. “guarantee” based on provider’s past experience; may not reflect future.
Outline
Traffic regulation

“There is a difference in kind between eating an ice cream cone every day, and eating 365 ice cream cones on your birthday” [Tanenbaum, 2nd edition, p. 86]

We often want to control the timing of when traffic enters a point in the network:

- **Average rate**: Measured over an infinite interval.
- **Shorter-term rates**, e.g. measured over the period of a second; peak rate (e.g. $1/\text{minimum interarrival time}$)
- **Maximum burst size**

We want to find ways to:

- “**parameterise**” traffic (so a network and user can agree on what load will be served with certain quality), and to
- **regulate** traffic so that it does not exceed declared parameters.

This is often done by using various forms of Leaky Buckets...

† aka “traffic conditioning”
The Leaky Bucket Algorithm

A leaky bucket with...

... water

... packets

Faucet

Leaky bucket

Water

Water drips out of the hole at a constant rate

Host computer

Packet

Unregulated flow

The bucket holds packets

Interface containing a leaky bucket

Regulated flow

Network

T Fig. 5-32
Leaky Buckets of tokens

**Warning**: The device shown on the previous slide is sometimes called a “Peak rate limiter” to distinguish it from another application of the term “Leaky Bucket” to describe a *token* regulator.

All channels effectively provide peak rate limits. Peak rate limiting (at less than channel capacity) can introduce unnecessary delays – switches may tolerate *some* bursts provided they are of limited length (volume/duration).

⇒ Introduce the concept of a *token regulator*:†

† aka “Token Bucket” or merely “Leaky Bucket”
Token regulator

Packet† passage requires a token‡
• If no tokens are stored, the packet can’t pass
  (may be buffered or discarded
  → “Regulator types” slide)
• When a packet passes, a token must be
  removed from the bucket.
• Periodically, tokens “drip” into† the bucket. If the bucket is full, token is
discarded.

Bucket depth ($B$ or $\sigma$) ⇒ (but $\neq$) maximum burst size.

Transmission of packets permitted by bucket will take time, and during that time,
进一步 tokens may be earned, allowing the burst to continue.

Drip rate (of tokens entering bucket) ($R$ or $1/\rho$) determines long-term mean

† Regulation could equally be applied to bytes, e.g. if packets have variable length.
‡ The above description is in terms of tokens representing credit that allows transmission. A
  complementary description is of tokens representing debt (i.e. barriers to transmission), in
  which case passage is permitted only if there is space to add another token, and tokens
  periodically leak out of the bucket.

Fig. 13.2 from Keshav
The Token Bucket Algorithm

Before.

After.
Equivalent views about tokens

Pessimist: “The glass is half empty”

In ATM and Frame Relay, tokens represent debt: Source is allowed to accrue bucket-full of debt, which gets written-off gradually over time.

Optimist: “The glass is half full”

In IP (RSVP), tokens represent credit: Source is allowed to consume from the bucket until depleted. Earns credit gradually.

Engineer: “The glass is twice as big as it needs to be”

The two views are functionally equivalent

Optimist/pessimist material from ICNP 03 panel on Future Directions and Visions for Networking Research http://icnp03.cc.gatech.edu/ICNP-liebeherr2.ppt

Copyright © 2006 Tim Moors
Regulator types

All regulators affect the timing of packet arrivals.

Types of regulators†:

• **Policer**: Packet that can’t be sent now is discarded.

• **Marker/Tagger**: Packet that can’t be sent now is marked as having lower priority (e.g. more likely to be discarded)
  • contracts are generally more lenient for lower priority traffic, letting it pass when high priority traffic can’t.

• **Shaper**: Packet that can’t be sent now is buffered until it can be sent.

† aka “packet conditioners”
Cascades of regulators

Regulators may be cascaded, e.g.:

- First regulator regulates maximum burst size. Second regulator regulates peak rate within the burst.
- First regulator may limit rate over long durations, second regulator may limit rate over shorter durations.
- Shapers may interact when cascaded – e.g. second shaper may counteract some earlier shaping
Fig. 5-33

Input to a leaky bucket

Output from a peak rate limiter

Output from a Token Bucket with capacity of...

- 250KB
- 500KB
- 750KB

Output from a cascade: 500KB Token Bucket feeding a 10MB/s peak rate limiter

11ms = 10ms to empty bucket + 1ms using credit earned while emptying

500KB

750KB

T Fig. 5-33
Leaky bucket conclusion

• Leaky Buckets are important because they limit the flow of data.
  • Limit loss from buffer overflow
  • When used with certain scheduling disciplines (see later) the network can provide guarantees about delay and throughput.
• When switches buffer traffic, the output traffic may be burstier than the input ⇒ may need to perform shaping at switch output
Outline
Scheduling

When there is contention, decide:

**Service order**
- ⇒ queueing *delay* experienced by different users
- influences *throughput* experienced by different users

**Service time** (non-work conserving schedulers)†
- influences *delay* and *throughput* experienced by different users

*Whether to serve* or discard a “customer” – since queues have finite size → buffer management policies
- determines *loss rates* experienced by different users
- large delay (e.g. infinite, or longer than timeout) is effectively loss

† Analogy with switching fabrics: sorting determines order, switching determines order + which ports are idle. Similarly for traffic going out one port: it is one thing to determine the order, but another to determine where in this order the port should be idle.
Where scheduling is done

In many locations!

- **Switches:**
  - **Output port**
    - Must deal with contention there
    - May want to manage traffic before it hits the next switch.
  - **Input port:** To protect switch fabric, penalise excessive transmitters.

- **Any point of contention/sharing:**
  - Access to a web server (application layer)
  - Access to a transmission medium (MAC layer)
  - Beyond networking, e.g. passengers waiting to board a plane
    - “early boarders spend less time waiting
    - bumped off passengers are ‘lost’!” [Keshav]
Outline

Scheduling

Goals:
  Fairness
  Protection
  Efficiency
  of implementation
  of admission control
  Ability to guarantee performance (covered after WFQ)
Fairness

Naïve goal: Equal access for all.

... but when you look deeper:

- When charge users (e.g. public network), make service $\propto$ payment.

More twists:

- **socialism**: perhaps everyone deserves a basic level of service
- **economies of scale**: bulk service is easier to provide than multiple smaller services, e.g. fixed cost of billing
- **premium for high performance**, e.g. $(T\text{flops supercomputer}) > 1000 \times $(1G\text{flops PC})$, but can process jobs that can’t be parallelized.

- For private networks (e.g. LANs), if want fairness (boss may want more), want fair performance for end-users.

One server may support multiple end-users; should give it higher throughput than individual end-users.

⇒ More correctly: **We seek a controllable allocation of service to computers.** e.g. ability to give certain flows higher *weighting*.

flops = floating point operations per second
Measures of fairness

Fairness Index
- only used to evaluate; provides no algorithm to allocate
- assumes that everyone:
  - wants more if they can get it
  - is sharing the same resource

Max-Min fairness
- when demands vary, and some demands may be lower than equal share for all
- addresses interaction between demands through limited resources
- provides an algorithm for allocating resources
Requirements for measuring fairness

Want some measure of the “variability” of allocation sizes ($x_i$).
Can assume that allocations are nonnegative

Requirements:

• Should be independent of scale, e.g. $x_i \rightarrow Sx_i$

\[
\sum_{i=1}^{n} (x_i - \mu)^2
\]

• Variance ($\sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \mu)^2}{n}$) is affected by scale: $\sigma^2(Sx) = S^2 \sigma(x)$

• Should be bounded, e.g. allowing statements about $x\%$ fair, indicating how close to optimum/how far from worst case.
  • Coefficient of variation ($\sigma/\mu$) is unbounded

• Should account for allocations of all users, not just extremes
  • Can’t just use $\max_i(x_i)/\min_i(x_i)$ ratio
Fairness Index

The “Fairness Index”† is widely used:

\[ f(x_1, x_2, \ldots, x_n) = \frac{\left( \sum_{i=1}^{n} x_i \right)^2}{n \sum_{i=1}^{n} x_i^2} \]

\[ f(x_1, x_2, \ldots, x_n) = \frac{\left( \sum_{i=1}^{n} x_i \right)^2}{n \sum_{i=1}^{n} x_i^2} = \frac{(nx)^2}{nnx^2} = 1 \]

\[ f(x_1, x_2, \ldots, x_n) = \frac{\left( \sum_{i=1}^{e} x_i \right)^2}{n \sum_{i=1}^{e} x_i^2} = \frac{(ex)^2}{nex^2} = \frac{e}{n} \]

- \( f=1 \) ⇒ totally fair: all users receive the same allocation (\( x \))
- \( f=e/n \) when \( e \) users receive equal allocation (\( x \)) and \( n-e \) users receive 0 allocation

✓ This Fairness Index is bounded: Range \([0,1]\), independent of scale, and accounts for all allocations.

max-min fairness

(when allocating resources to adaptive flows)

Goals:

- No source should receive more than it demands. (waste prevents service to others → inefficiency)
- Sources with unsatisfied demands should receive equal\(^\dagger\) shares of the resource (fairness)
- \textbf{maximize the minimum} share of sources with unsatisfied demands (to maximize network utility)

\(\dagger\) An alternative of receiving a share \textit{proportional} to their demand tends to encourage exaggerated demands. Allocating equally avoids such gamesmanship.
max-min fairness: Algorithm

$C = \text{resource capacity}$
$s = \text{number of sources}$
$d_i = \text{demand from source } i, \ i \in [1, s]$  

Algorithm:
- Initially allocate each source $a_i = C/s$.
- Residual capacity $R = \sum \max(a_i - d_i, 0)$
- Reduce allocations s.t. don’t exceed demand: $a_i = d_i \ \forall \ i: d_i < C/s$
- Repeat by distributing $R$ amongst sources with unsatisfied demands

The Fairness Index for this allocation = 0.91, although that measure isn’t really appropriate since sources have different demands.
**Network fairness**

Achieve globally fair allocation by choosing smallest locally fair allocation.

*Throughput demanded by each flow*

<table>
<thead>
<tr>
<th>Flow</th>
<th>Link</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Allocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>F2</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>F3</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>F4</td>
<td></td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>F5</td>
<td></td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

All link capacities = 12b/s
Network fairness (ctd)

After Round 1:
Throughput demanded by each flow

F1@2b/s, F2@10b/s, F3@20b/s
F4@8b/s, F5@50b/s

All link capacities=12b/s

F1=2b/s, F4=5b/s, F5=5b/s
Remaining capacity: A=0, B=7, C=7

Round 2:
• F2 gets 7b/s
• F3 gets 7b/s
⇒ Final max-min fair allocation:
  F1=2b/s, F2=7b/s, F3=7b/s, F4=5b/s, F5=5b/s
Fairness vs optimality

Fairness comes at the price of “optimality” (maximizing aggregate throughput):

For max-min fairness: \(2 + 7 + 7 + 5 + 5 = 26 \text{ b/s}\) \(f()=0.89\)

If flow 4 received nothing: \(2 + 10 + 12 + 0 + 10 = 34 \text{ b/s}\) \(f()=0.66\)

Aside: More inappropriate Fairness Index measures
Protection†

**Protection**: Mis-behaving flows should not degrade the service received by other flows.

Mechanisms for protecting:
- Provide fairness. Fairness $\Rightarrow$ protection
- Regulating traffic using Leaky Buckets
  
  Protection $\not\Rightarrow$ fairness: e.g. policing may protect by limiting rates, but limits may be unfair.

† aka “isolation”, “segregation”
Ease of implementation

The switch must schedule the transmission of each packet
⇒ must be done at high speed (e.g. 72B Ethernet frame @ 1Gb/s = 0.576us)
⇒ desire:
  • simplicity: low complexity, especially to facilitate hardware implementation
  • scalability: work per packet should be insensitive to the number of connections
    e.g. packet arriving for one connection could affect output times of packets on many other connections. Ideally, work should scale less than linearly with # of connections.
One way to facilitate scalability is to aggregate many connections together, e.g.
  • ATM uses “virtual paths” to aggregate multiple “virtual circuits”.
  • Diffserv uses Behavior Aggregates – packets that belong to the same service class.
However mechanisms that handle aggregates cannot isolate flows within the aggregate. i.e. scalability comes at the cost of protection.
Efficient Admission Control

During admission control, the network needs to determine resource requirements for new connections, and measure what capacity remains.

- Trivial if these are scalars (e.g. “bandwidth”)
- Complicated if they are vectors (e.g. data rate, delay, ...).

A “schedulable region” (Fig. 9.3 of Keshav) describes what combinations of connections are admissible.

Efficient scheduling algorithms create a large schedulable region, allowing many connections to co-exist.

Weighted Fair Queueing can provide performance guarantees, but not as efficiently as non-work conserving schemes.
Outline
Scheduling algorithms

Algorithms aim to provide fair sharing & performance guarantees

Simplest: **First Come First Served**

**Round Robin**: Make a round, covering all sources, then repeat

**Fair Queueing/scheduling**

Class Based Queueing
Earliest Due Date

Ideal: **Generalised Processor Sharing**
First Come First Served (FCFS)†

Source A: 2 pkt/2 sec
Source B: 1 pkt/1 sec
Source C: 2 pkt/4 sec

Input:

Output (FCFS, top-to-bottom): 4 pkt /1 sec

- Provides no protection: “Bandwidth hogs” “come” often, and so get served often.
- Simple to implement

† Sometimes called First In First Out (“FIFO”); though with parallel servers, only equivalent if all jobs take the same time to complete.
Round Robin

Do “rounds” of queues of packets from waiting connections, serving one packet from each waiting connection in each round.

Weighting is possible, e.g. serve connection with weighting of 2 twice as many times in the round as another connection with weighting of 1.

Variable packet length complicates matters:

- normalise weights using mean packet size if known in advance
- e.g. weights {0.5, 0.75, 1.0} bytes/sec ⇒ bytes/round
  - mean packet sizes {50, 500, 1500} bytes/packet
  - normalize weights: bytes/round / bytes/packet = packets/round
    - \{0.5/50,0.75/500,1.0/1500\} = \{0.01,0.0015,0.000666\}
  - normalize again to integer # of packets served per round \{60, 9, 4\}

i.e. in one round 60 packets from one source, 9 from next, 4 from last

variants: weighted RR & deficit RR

- e.g. WRR used by Cisco Catalyst 8540
Limitations of Round-Robin

- Need to know mean packet length in advance. Usually don’t.
- Short-term unfairness:
  Flows with low weightings (or large packets) get rare service ⇒ “unfair” over shorter intervals.
  e.g. 500 connections
    250 fast connections with weight of 10
    250 slow connections with weight of 1
  1 round serves 2750 packets (10 for the fast connections and 1 for the slow connections)
  If packets are 500B and transmission speed is 45Mb/s, 2750 packets take 244.2ms
  i.e. may be unfair for up to approx. ¼ second.
Generalised Processor Sharing (GPS)

“Fluid flow” model:
- Easy to understand and analyse
- Reservoirs (buffers):
  - Store transient excess arrivals (up to some limit)
  - Are depleted when service rate exceeds arrival rate.
- Flows are isolated: Rate of service of a flow depends on pipe width (weighting).
- Service is of \( \text{infinitesimal} \) parts of each flow
  \( \times \) Unrealistic: real information is transmitted in discrete units: bits, packets

Don’t think too deeply about the unrealistic physics of what is shown here.
Utility of GPS

✓ Provides a benchmark for evaluating the fairness of realisable schedulers.

   Absolute† Fairness Bound = | S_{real}(i) – S_{GPS}(i) |
   where S_X(i) = service for connection i using scheduler X

✓ Indicates the order in which packets would finish transmission with such idealised scheduling.

   Justifies realisable schedulers (Fair Queueing →) that attempt to provide the same finish times as GPS.

† A relative fairness bound compares the service received by different connections from the same scheduler, i.e. |S_{real}(i) – S_{real}(j)|
Fair Queueing†

Basic algorithm (no weightings, infinite buffer):
• Find the finish time of a packet, had we been doing GPS.
• Serve packets in order of their finish times.
  Of packets with equal finish times, serve in random order.

But: Don’t want to forget about a packet once it has been transmitted.
(A recently busy connection would then compete on equal terms with
a previously quiet connection for next transmission)
⇒ connections remain “active” until the round # matches its finish #.
Details on the next slide →

✓ Better than Round Robin at dealing with variable packet lengths and
  weightings.
✓ Can provide service guarantees.

† aka Packet-by-packet Generalised Processor Sharing (PGPS)
Fair Queueing

**Rounds:** A byte is sent from each active connection in each “round”. Round number \((R)\) can be fractional (part way through a round)

**Finish number \((F)\):** Indicates order of packet transmissions finishing

Define a flow to be “active” when \(F_{\text{flow}}’s\) most current packet > \(R\)

\[
dR/dt = 1/(\# \text{ active})
\]

**Equations to determine \(F_i\):**

- Packet of length \(L_i\) arrives & is the \(i^{th}\) packet of:
  - An inactive flow: \(F_i = R + L_i\)
  - An active flow: \(F_i = F_{i-1} + L_i\)

\(\Rightarrow\) bursty arrivals have dispersed finish times
Fair Queueing example

Arrivals:
- A.1 \( F = 0 + 1 \)
- B.2 \( F = 0 + 2 \)
- C.2 \( F = 0 + 2 \)

Transmissions:

<table>
<thead>
<tr>
<th>Round number (black) + coloured finish numbers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

- Inactive: \( F_i = R + L \)
- Active: \( F_i = F_{i-1} + L \)
- Active until \( R = F_i \)
- \( dR/dt = 1/(\# \text{ active}) \)

Example based on Keshav p. 241

Remember
Fair Queueing example 2

Arrivals:
- $A_1 F=1$
- $B_1 F=1$
- $C_1 F=1$
- $A_2 F=2$
- $B_2 F=2$
- $D_1 F=1^{5/6}$

Transmissions:
- $A_1$
- $B_1$
- $C_1$
- $D_1$
- $A_2$
- $B_2$

Remember
- Inactive: $F_i = R + L$
- Active: $F_i = F_{i-1} + L$
- Active until $R=F_i$
- $dR/dt = 1/(# \text{ active})$

Because connections (e.g. A&B) remain active until the end of the round, not just until they finish transmitting, packets from previously inactive conn’s (e.g. $D_1$) may be served before others that arrived before them (e.g. $A_2$, $B_2$).
Fair Queueing with limited buffers

If buffer overflows, then **discard packets with largest finish numbers**.
i.e. when new packet \( i \) needing buffer space \( B_i \) arrives:

1. Store new packet in a temporary input buffer
   (Assume that incoming packet and any packet being transmitted can be stored in buffers that are separate from the shared buffer.)

2. Calculate the new packet’s finish number \( F_i \)

3. If \( \exists \) enough packets with higher finish numbers:
   \[
   \sum_{j: F_j > F_i} B_j > B_i
   \]
   1. Discard buffered packets in order of decreasing finish number until buffer has enough space for new packet \( i \).
   2. Store new packet \( i \) in buffer.
else
   Discard new packet \( i \)
Weighted Fair Queueing

To add weightings to Fair Queueing:

1. Add packet length / weighting to finish number, e.g. Inactive: \( F_i = R + L/w \)
   e.g. connection with twice the weighting of another finishes in half the time

2. Round number increases with slope \( \frac{dR}{dt} = \frac{1}{\sum_{i\in{\text{active}}} w_i} \)

Weighted Fair Queueing (WFQ) is widely used,
e.g. in Cisco series 1700, 2600, 3600, 7200
Outline
Guarantees from scheduling

With best-effort connections, the goal is fairness
With guaranteed-service connections:
  • What performance guarantees are achievable?
  • How easy is admission control?

Weighted Fair Queueing:
  ✓ Can guarantee throughput and delay
    ✓ weightings + limited number of connections ⇒ bound (guarantee)
      on long-term average \( throughput \)
        e.g. 2 connections with equal weightings both guaranteed 50% of shared link, e.g. 1Mb/s of 2Mb/s link.
    ✓ Parekh-Gallager theorem provides \( delay \) bounds
  ✗ Has inefficient admission control

\[ \text{see next slide} \]
Parekh-Gallager theorem

g = least throughput the connection will receive at each WFQ scheduler on its path

Connection is Leaky-Bucket regulated such that # bits sent in time \([t_1, t_2]\) ≤ \(\rho(t_2 - t_1) + \sigma\)

\(r(k) = \text{rate of } k\text{th scheduler}, \ g_k \geq \rho \ \forall \ k\)

\(P = \text{size of largest packet allowed}\)

\[
\text{end\_to\_end\_delay} \leq \sigma / g + \sum_{k=1}^{K-1} P / g + \sum_{k=1}^{K} P / r(k)
\]

⇒ WFQ can provide end-to-end delay bounds

But inefficient: low delay requires high throughput \((g)\)

⇒ motivation for non-work conserving scheduling
Outline

• Non-work conserving scheduling
Work conservation

**Work conserving schemes** are idle (not transmitting) only when they have no packets awaiting service.†

- e.g. GPS
  
  Conserving work means packet transmission time depends on presence of traffic from other flows (@ output) ⇒ leads to delay variability.

**Non-work conserving schemes** can be idle when packets are queued

  - i.e. they may deliberately withhold a packet from transmission, when no other packet is being transmitted.

  ⇒ may not be able to fully utilise the output port.

- e.g. rate-controlled scheduling: Preface the scheduler with a regulator (shaper) that delays packets until they are eligible (e.g. eligible according to a Leaky Bucket)

† Non-work-conserving schemes “fall behind” / “create extra work for themselves” (hence the name) by being idle when packets are waiting.
Evaluation of non-work conserving schemes

Non-work conserving schemes:
- ☐ sacrifice average delay and throughput
  - ☑ but can use idle time to transmit lower priority (e.g. best-effort) traffic
- ☐ complicate switches
  - ☑ but reduce necessary buffer sizes
- ☑ reduce delay variability
  - ☐ could have been achieved in end-system with playback buffer
- ☑ enable tighter delay bounds across multi-hop networks (traffic profiles needn’t become looser at later switches)
- ☑ enable efficient admission control
Lecture summary

- Some applications seek service guarantees
- Guarantee service by prioritising and isolating traffic, admission control. Guarantees for real systems are likely to be statistical.
- Traffic regulators limit burstiness of traffic. Types include policers, markers and shapers. Use Leaky Buckets.
- Best effort traffic seeks fair sharing, e.g. max-min fair allocation.
- Schedulers:
  - Round Robin problems: unknown packet length, short term unfairness
  - Weighted Fair Queueing: Fair to adaptive apps, guarantee service to non-adaptive apps. Inefficient admission control.
  - Non-work-conserving schedulers can guarantee service with more efficient admission control.