Packet Scheduling and QoS

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Scheduling

- Sharing always results in contention
- A scheduling discipline resolves contention:
  - who's next?
- Key to fairly sharing resources and providing performance guarantees

Components of Scheduling

- A scheduling discipline does two things:
  - decides service order
  - manages queue of service requests
- Example:
  - consider queries awaiting web server
  - scheduling discipline decides service order
  - and also if some query should be ignored

Where do we implement scheduling?

- Anywhere where contention may occur
- At every layer of protocol stack
- Usually studied at network layer, at output queues of switches

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Why do we need scheduling?

- Because future applications need it
- We expect two types of future applications
  - best-effort (adaptive, non-real time)
    - e.g. email, some types of file transfer
  - guaranteed service (non-adaptive, real time)
    - e.g. packet voice, interactive video, stock quotes

What can Scheduling disciplines do?

- Give different users different qualities of service
- Example of passengers waiting to board a plane
  - early boarders spend less time waiting
  - bumped off passengers are “lost”!
- Scheduling disciplines can allocate
  - bandwidth
  - delay
  - loss
- They also determine how fair the network is

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Requirements of a Scheduling Discipline

- An ideal scheduling discipline
  - is easy to implement
  - provides fairness and protection (for best-effort connections)
  - provides performance bounds (for guaranteed-service connections)
  - allows easy admission control decisions (for guaranteed-service connections)
    - to decide whether a new flow can be allowed

Requirement 1: Ease of Implementation

- Scheduling discipline has to make a decision once every few microseconds!
- Should be implementable in a few instructions or hardware
  - If there are N connections and the complexity is $O(N)$, it doesn't scale. We want $O(1)$.
  - for hardware: critical constraint is VLSI space to remember scheduling state.
- Work per packet should scale less than linearly with number of active connections

Requirement 2: Fairness and Protection

- Scheduling discipline allocates a resource
- An allocation is fair if it satisfies **Max-min** fairness
- Intuitively
  - each connection gets no more than what it wants
  - the excess, if any, is equally shared

Max-Min Fairness

- Resources are allocated in order of increasing demand
- No source gets a resource share larger than its demand
- Sources with unsatisfied demands get an equal share of the resource.

Max-Min Fairness Algorithm

Let there be n sources and source i demand is x_i. With loss of generality, assume $x_1 \leq x_2 \cdots \leq x_n$ and the server has capacity C.

- Initially, everyone has C/n of the resource.
- If $x_1 > C/n$, then source 2 to n evenly divide $C/n - x_1$ remaining resource so they get $C/n + (C/n - x_1)/(n-1)$ of the resource.
- Process repeats until when each source gets no more than what it asks for and if its demand was not satisfied, no less than what any other source with a higher requirement.

Example of Max-Min Fairness

Four sources with demands 2, 2.6, 4 and 5. The capacity is 10.

- Initially, every source gets 2.5 of resource.
- The excess of source 1 is 0.5, so source 2 to 4 get 2.666 of resource.
- The excess of source 2 is 0.0666, so source 3 and 3 get 2.7 of resource.
- The max-min fair allocation is 2, 2.6, 2.7, 2.7.

Example of Weighted Max-Min Fairness

Each source i specifies its demand x_i and weight w_i. For example, 4 sources with demand 4,2,10 and 4 and weights are 2.5, 4, 0.5, 1. The capacity is 16.

- normalize so that smallest weight is 1. We have weight 5,8,1,2.
- Assign resource based on this weight, so the initial round is $[5, 8, 1, 2]$.
- The remaining resources are $(5 - 4) + (8 - 2) = 7$.
- Divide remaining resource according to the normalized weight of source 3 and 4. Therefore, source 3 and 4 have $[3.3, 6.7]$.
- The remaining resource from source 4 is $(6.7 - 4) = 2.7$. Give this to source 3.
- The final weighted max-min fairness allocation is $[4, 2, 6, 4]$.

Fairness (continue)

- Fairness is intuitively a good idea
- But it also automatically provides protection
  - traffic hogs cannot overrun others
  - automatically builds firewalls around heavy users
- Fairness is a global objective, but scheduling is local
- Each endpoint must restrict its flow to the smallest fair allocation
- Dynamics + delay $\Longrightarrow$ global fairness may never be achieved

Requirement3: Performance Bounds

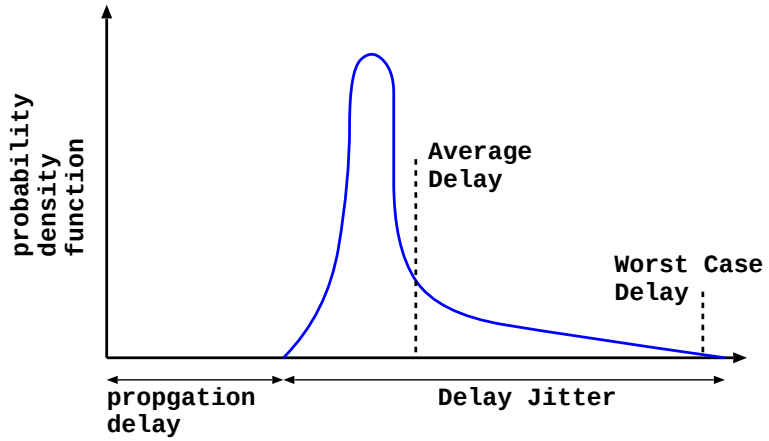
- What is it?
  - A way to obtain a desired level of service
- Can be deterministic or statistical
- Common parameters are
  - bandwidth
  - delay
  - delay-jitter
  - loss

Bandwidth

- Specified as minimum bandwidth measured over a pre-specified interval
- E.g. $>$ 5 Mbps over intervals of $>$ 1 sec
- Meaningless without an interval!
- Can be a bound on average (sustained) rate or peak rate
- Peak is measured over a "small" interval
- Average is asymptote as intervals increase without bound

Delay and Delay-Jitter

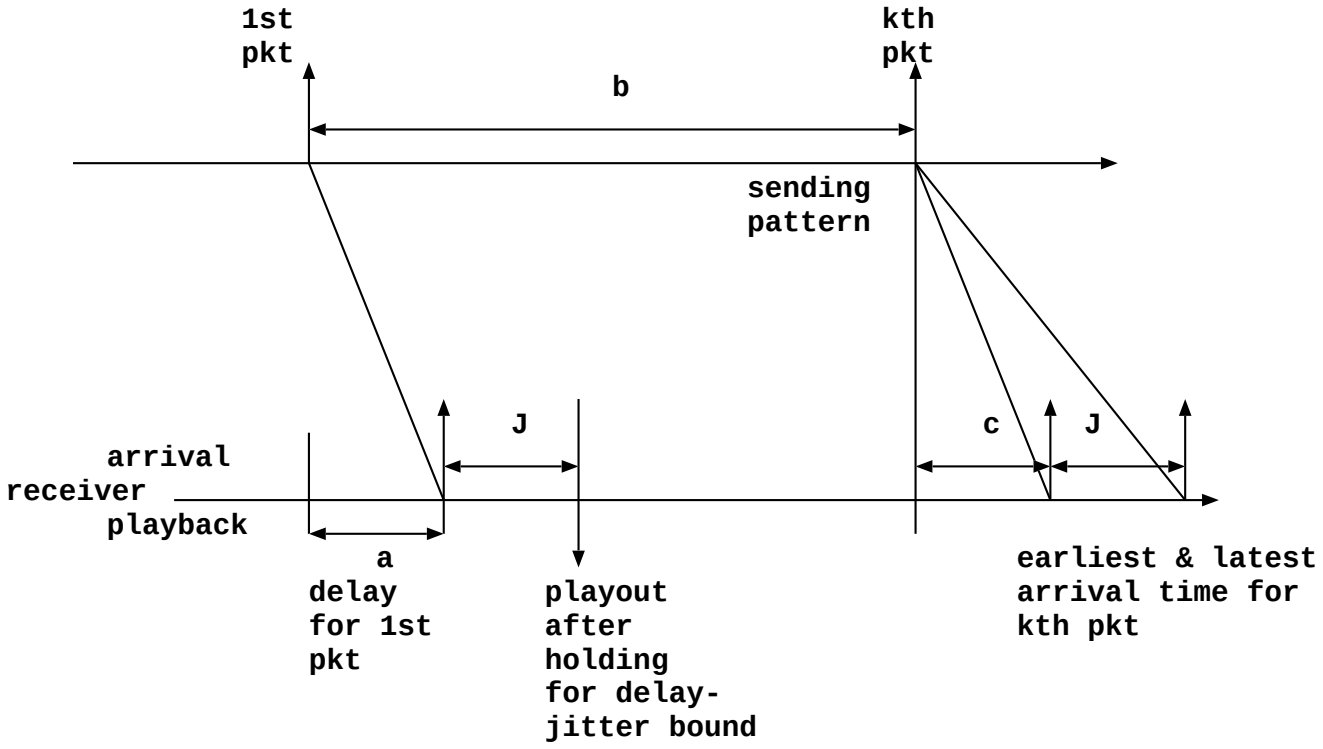
- Bound on some parameter of the delay distribution curve



- Delay Jitter bound requires that the networks bound the difference between the largest and smallest delays received by packets on that connection.
- Trivially as large as the delay bound minus the propagation delay.

Playback application

Removing delay jitter using playback buffer



- 1st packet has a delay a and the delay-jitter bound is J.
- the k^{th} packet is sent b seconds after the 1st packet
- Propagation delay is c, the k^{th} packet latest arrival is $c + J$.
- Since $a \geq c$, $a + b + J$, which is the time at which the k^{th} packet is played out, is guaranteed larger than $c + b + J$, which is the worst arrival time.
- Using delay buffer, we guaranteed a packet has always arrived by the time the receiver is ready to play it.

Requirements 4: Ease of admission control

- Admission control needed to provide QoS
- Overloaded resource cannot guarantee performance
- Choice of scheduling discipline affects ease of admission control algorithm
- E.g., FCFS is the easiest by no performance guarantee.
- Tradeoff between preciseness of admission control and efficiency.

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Fundamental choices

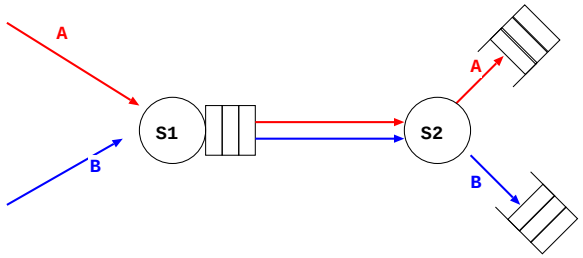
1. Number of priority levels
2. Work-conserving vs. non-work-conserving
3. Degree of aggregation (within a level)
4. Service order within a level

Choices 1: Number of Priority Level

- Packet is served from a given priority level only if no packets exist at higher levels (multilevel priority with exhaustive service)
- Highest level gets lowest delay
- Watch out for starvation!
- Usually map priority levels to delay classes
- Example:
  - Class 1: non-realtime
  - Class 2: Realtime
  - Class 3: Low bandwidth urgent message

Choices2: Work conserving vs. non-work-conserving

- Work conserving discipline is never idle when packets await service
- Why bother with non-work conserving?



- many pkts from A pile up behind burst of B. When pkts from B is served, pkts from A will arrive to S_2 in a burst.
- makes traffic arriving at down stream more **predictable**, **reduces buffer size**, and **reduces delay jitter** experienced by connection.

Non-work-conserving disciplines

- Key conceptual idea: delay packet till eligible
- Reduces delay-jitter $\Longrightarrow$ fewer buffers in network
- How to choose eligibility time?
  - rate-jitter regulator
    - bounds maximum outgoing rate
  - delay-jitter regulator
    - compensates for variable delay at previous hop

Rate-jitter regulator

- the scheduler guarantees that source traffic leaving the scheduler conforms to a given *rate descriptor*.
- Assume all packets are the same size, the scheduler guarantees a particular *peak rate*.
- Eligibility time for a packet is the sum of the eligibility time of the previous packet on the same connection and the inverse of the peak rate.
- Let $E(k)$ be the eligibility time of the k^{th} packet, $A(k)$ is its arrival time. X_{min} is the interarrival time of packets under peak rate. We have:

$$\begin{aligned} E(1) &= A(1) \\ E(k) &= \max\left(E(k-1) + X_{min}, A(k+1)\right) \end{aligned}$$

Delay-jitter regulator

- guarantees that sum of the queueing delay in the previous switch and the *regulation* delay in the current switch is a constant so as to remove queueing-delay variability.
- Output stream is a *time-shifted* and *fully-reconstructed* version of the traffic entered the previous switch. So we remove the possibility of build-up burstiness.
- Let $E(i,k)$ be the eligibility time of the k^{th} packet at the i^{th} switch. D is the delay bound of the previous switch, L is the largest possible delay on the transmission between switch i and $i+1$.

$$\begin{aligned} E(0,k) &= A(0,k) \\ E(i+1,k) &= E(i,k) + D + L \end{aligned}$$

- If a packet is served before its delay bound at the previous switch, or receives a delay smaller than L on the link, the delay-jitter regulator at the downstream will add sufficient delay.
- Thus, the output of the regulator is a $(D + L)$ second time-shifted of the version of the traffic that left the previous regulator.
- Delay-jitter regulator is **hard** to implement because it requires knowledge of D and L (difficult if we cross different network domains).
- It also required clock synchrony between adjacent switches at all times.

Do we need non-work-conservation?

- Can remove delay-jitter at an endpoint instead
  - but also reduces size of switch buffers, which is more expensive.
- Increases mean delay
  - not a problem for playback applications, may be for others
- Wastes bandwidth
  - can serve best-effort packets instead
- Always punishes a misbehaving source
  - source needs to characterize its traffic
- Bottom line: not too bad, implementation cost may be the biggest problem

Choices 3: Degree of aggregation

- More aggregation
  - less state (e.g., FCFS, few states but no QoS guarantee)
  - cheaper
    - smaller VLSI
    - less to advertise
  - BUT: less individualization
  - more states (e.g., sophisticated scheduling may not be scalable).
- Solution
  - aggregate to a class, members of class have same performance requirement (e.g., DiffServ)
  - no protection within class

Choices 4: Service within a priority level

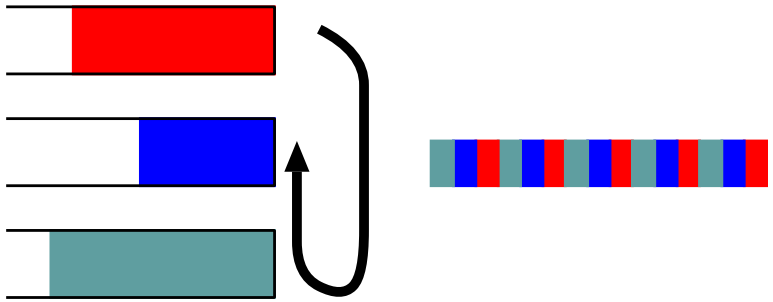
- In order of arrival (FCFS) or in order of a service tag
- Service tags $\Longrightarrow$ can arbitrarily reorder queue
  - Need to sort queue, which can be expensive
- FCFS
  - bandwidth hogs win (no protection)
  - no guarantee on delays
- Service tags (*read book*)
  - with appropriate choice, both protection and delay bounds possible

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Scheduling best-effort connections

- Main requirement is fairness
- Achievable using Generalized processor sharing (GPS)
  - Visit each non-empty queue in turn
  - Serve infinitesimal from each
  - Why is this max-min fair? (EXPLAIN)
  - How can we give weights to connections?



More on GPS

- GPS is unimplementable! *we cannot serve infinitesimal, only packets*
- No packet discipline can be as fair as GPS
  - while a packet is being served, we are unfair to others
- Degree of unfairness can be bounded
- Define $g(i,k)$ be the service rate allocated to connection i at the k^{th} switch. $r(k)$ be the service rate of the k^{th} switch, We have:

$$g(i,k) = \phi(i,k)r(k) / \sum_{\forall j} \phi(j,k),$$

where $\phi(i,k)$ is the weight of the i^{th} connection at the k^{th} switch. Let $g(i) = \min_k g(i,k)$.

- **Relative Fairness Bound (RFB):** given two active flows i and j with $g(i)$ and $g(j)$. Let $S(i,\tau,t)$ be the amount of received service in time $[\tau,t]$ for flow i. We have:

$$\textbf{RFB} = \left| \frac{S(i,\tau,t)}{g(i)} - \frac{S(j,\tau,t)}{g(j)} \right|$$

- **Absolute Fairness Bound (AFB):** Let $G(i,\tau,t)$ be the amount of received service in time $[\tau,t]$ for flow i under **GPS**. We have:

$$\textbf{AFB} = \left| \frac{S(i,\tau,t)}{g(i)} - \frac{G(i,\tau,t)}{g(i)} \right|$$

- If a scheduler S has an AFB of x, it automatically has a RFB of $2x$ because one flow could receive x more service than GPS and other flow could receive x less service.

What can we do?

- We can't implement GPS
- So, lets see how to emulate it
- We want to be as fair as possible
- But also have an efficient implementation

Weighted Round Robin

- Serve a packet from each non-empty queue in turn
- Unfair if packets are of different length or weights are not equal
- Different weights, fixed packet size
  - after normalizing to obtain integer weights, we actually serve more than one packet per visit
- Different weights, variable size packets
  - normalize weights by mean packet size e.g. weights {0.5, 0.75, 1.0}, mean packet sizes {50, 500, 1500}
  - normalize weights: {0.5/50, 0.75/500, 1.0/1500} = {0.01, 0.0015, 0.000666}, normalize again {60, 9, 4}

Problems with Weighted Round Robin

- With variable size packets and different weights, need to know mean packet size in advance
- Can be unfair for long periods of time
- E.g.
  - T3 trunk (45 Mbps) with 500 connections, each connection has mean packet length 500 bytes, 250 with weight 1, 250 with weight 10
  - Each packet takes 500 * 8/45 Mbps = 88.8 microseconds
  - Round time = (250+ 250*10)*88.8 = 2750 * 88.8 = 244.2 ms
- Since packet transmission time is much less than round time, we have a long period of unfairness.

Weighted Fair Queueing (WFQ)

by Parekh

- Deals better with variable size packets and weights
- GPS is fairest discipline
- Find the **finish time** (or finish #) of a packet, *had we been doing GPS*
- Then serve packets in order of their finish times

WFQ: determine finish

- Suppose, in each **round**, the server served one bit from each active connection
- **Round number** is the number of rounds already completed
  - can be fractional
- Under GPS, if a packet of length p arrives to an empty queue when the system round number is R, it will complete service when the round number is $R + p \Longrightarrow$ finish number is $R + p$ independent of the number of other connections!
- Under GPS, if a packet arrives to a non-empty queue, and the previous packet, either in its queue or last served from its queue, has a *finish number* of f, then the packet's finish number is $f + p$

- Serve packets in order of finish numbers
- let $P(i,k,t)$ be the size of the kth packet that arrives on flow i at time t. Let $R(t)$ be the round number, $F(i,k-1,t)$ is the finish number for the $(k-1)$th packet on that connection, then

$$F(i,k,t) = \max\{F(i,k-1,t), R(t)\} + P(i,k,t)$$

- E.g., if a packet of 10 bits arrives to an empty flow with the system round number being 3, the packet finish number is 13.
- E.g., if a packet of 10 bits arrives to a flow that already contains a packet with a finish number of 20, its finish number is 30.
- note that the round number increases at a rate *inversely proportional* to the number of active connections.

WFQ: An important catch

- A queue may need to be considered non-empty even if it has no packets in it
  - e.g. packets of length 1 from connections A and B, on a link of speed 1 bit/sec
    - at time 1, packet from A served, round number = 0.5
    - A has no packets in its queue, yet should be considered non-empty, because a packet arriving to it at time 1 should have finish number $1 + p$
- A connection is active if the last packet served from it, or in its queue, has a finish number greater than the current round number

WFQ: continued

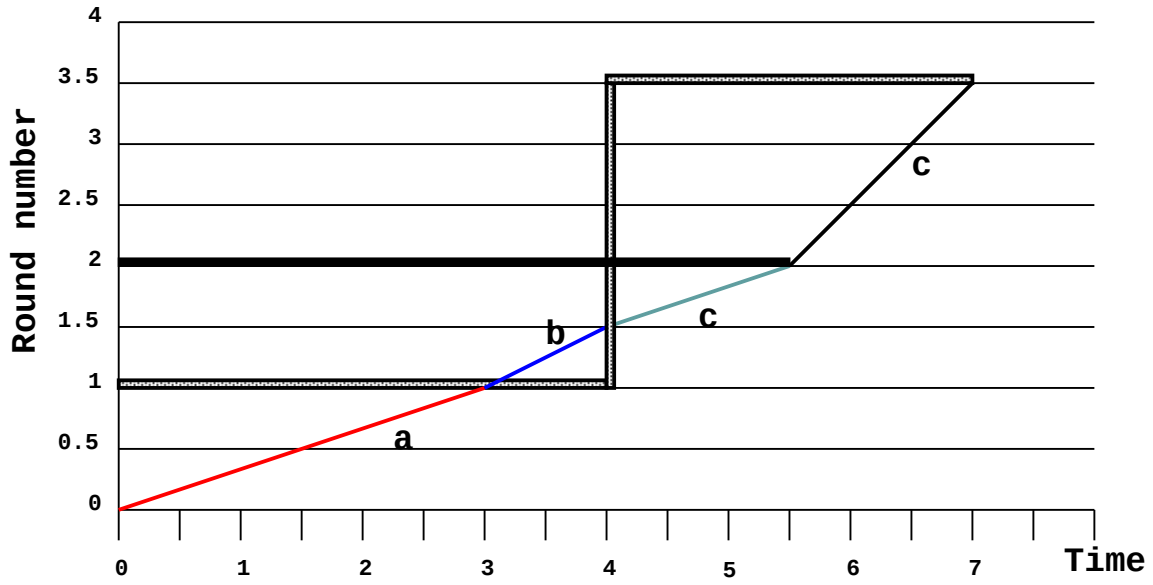
- To sum up, assuming we know the current round number R
- Finish number of packet of length p
  - if arriving to *active connection*, then finish number = previous finish number + p
  - if arriving to an inactive connection, then finish number = $R + p$
- To implement, we need to know two things:
  - is connection active?
  - if not, what is the current round number?
- Answer to both questions depends on computing the current round number (why?)

WFQ: computing the round number

- Naively: round number = number of rounds of service completed so far
  - what if a server has not served all connections in a round?
  - what if new conversations join in halfway through a round?
- Redefine round number as a real-valued variable that increases at a rate inversely proportional to the number of currently active connections
  - this takes care of both problems (why?)
- With this change, WFQ emulates GPS instead of bit-by-bit round robin

WFQ: Example

Packet of size 1,2, and 2 units arrive at time 0 to flows A,B and C. A packet of size 2 arrives to flow A at time $t = 4$. Server has capacity of 1 unit/s. (a) find finish number of all 4 packets, (b) round number when the system becomes idle, (3) what time will this occur?



Problem: iterated deletion

- A sever recomputes round number on each packet arrival
- At any re-computation, the number of flows can go up at most by one, but can go down to zero
- $\Longrightarrow$ overestimation
- Illustrate.
- Trick
  - use previous count to compute round number
  - if this makes some flows inactive, recompute
  - repeat until no flows become inactive

Weighted WFQ

- let $P(i,k,t)$ be the size of the kth packet that arrives on flow i at time t. Let $R(t)$ be the round number, $F(i,k-1,t)$ is the finish number for the $(k-1)$th packet on that connection, $\phi(i)$ is the weight of the ith flow, then

$$F(i,k,t) = \max\{F(i,k-1,t), R(t)\} + \frac{P(i,k,t)}{\phi(i)}$$

- Round number increases with a slope of $\frac{1}{\sum_i \phi(i)}$, where we take the sum over all active sessions.

WFQ implementation

- On packet arrival:
  - use source + destination address (or VCI) to classify it and look up finish number of last packet served (or waiting to be served)
  - recompute round number
  - compute finish number
  - insert in priority queue sorted by finish numbers
  - if no space, drop the packet with largest finish number
- On service completion
  - select the packet with the lowest finish number

Some Variants of WFQ

- Self-clock Fair Queueing (SCFQ): rather than using round number, it uses the finish number of the pkt *currently in service*.

$$F(i,k,t) = max\{F(i,k-1,t), CF\} + P(i,k,t)/\phi(i)$$

where CF is the finish number of the packet currently being served.

- Golestani has shown:

$$\text{RFB} = P_{max}(i)/g(i) + P_{max}(j)/g(j)$$

Problem with SCFQ

- SCFQ can be unfair over short time scales
- Connection C_0 has weight 50, connections C_1 to C_{50} have weight 1. Unit-size pkts arrive from C_1 to C_{50} at $t = 0$. They will have finish number of 1.
- Let a packet from C_0 arrive just after $t = 0$. Since the in service has finish number of 1, the finish number of this packet is 1.02. Therefore, it will be served after one packet from all other connections, i.e., at time 50, incuring a delay of 50 time units.
- If the scheduler is a WFQ, the packet would have a finish number of 0.02, incuring a delay of only 1 time unit.

Analysis

- Unweighted case:
  - if GPS has served x bits from connection A by time t
  - WFQ would have served at least $x - P$ bits, where P is the largest possible packet in the network
- WFQ could send *more* than GPS would $\Longrightarrow$ absolute fairness bound $> P$
- To reduce bound, choose smallest finish number only among packets that have started service in the corresponding GPS system (WF^2Q)
  - requires a regulator to determine eligible packets

Evaluation

- Pros
  - like GPS, it provides protection
  - can obtain worst-case end-to-end delay bound
  - gives users incentive to use intelligent flow control (and also provides rate information implicitly)
- Cons
  - needs per-connection state
  - iterated deletion is complicated
  - requires a priority queue

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Scheduling guaranteed-service connections

- With best-effort connections, goal is fairness
- With guaranteed-service connections
  - what performance guarantees are achievable?
  - how easy is admission control?
- We now study some scheduling disciplines that provide performance guarantees

WFQ

- Turns out that WFQ also provides performance guarantees
- Bandwidth bound
  - ratio of weights * link capacity
  - e.g.. connections with weights 1, 2, 7; link capacity 10
  - connections get at least 1, 2, 7 units of bandwidth each
- End-to-end delay bound
  - assumes that the connection doesn't send "too much" (otherwise its packets will be stuck in queues)
  - more precisely, connection should be leaky-bucket regulated with (σ, ρ).
  - # bits sent in time $[t_1, t_2]$ is upper bounded by $\sigma + \rho(t_2 - t_1)$.

Parekh-Gallager theorem

- Let $g(i,k)$ be the service rate assigned to the i^{th} connection at the k^{th} scheduler, $\phi(i,k)$ is the weight, $r(k)$ is the capacity, we have

$$g(i,k) = \frac{\phi(i,k)r(k)}{\sum \phi(j,k)} \qquad \text{for } 1 \leq k \leq K.$$

- Assume $g(i) \geq \rho(i)$, otherwise we have unstable system.
- $P_{max}(i)$ is the largest packet allowed in the i^{th} connection.
- P_{max} is the largest packet allowed in the network.
- the worst case end-to-end delay bound $D^*(i)$ is

$$D^*(i) \leq \frac{\sigma(i)}{g(i)} + \sum_{k=1}^{K-1} \frac{P_{max}(i)}{g(i,k)} + \sum_{k=1}^{K} \frac{P_max}{r(k)}$$

Example

A connection with leaky parameters (16,384 bytes, 150 Kbps). Traverse 10 hops and all links are 45 Mbps. $P_{max} = 8192$ bytes. Propagation delay is 30 ms. What $g(i)$ will garantee end-to-end delay of 100 ms?

- $$70*10^{-3} = \frac{16384 * 8}{g(i)} + \frac{9 * 8192 * 8}{g(i)} + \frac{10 * 8192 * 8}{45 * 10^6} = 12.87 \text{ Mbps}$$

Significance

- Theorem shows that WFQ can provide end-to-end delay bounds
- So WFQ provides both fairness and performance guarantees
- Bound holds regardless of cross traffic behavior
- Can be generalized for networks where schedulers are variants of WFQ, and the link service rate changes over time

Problems

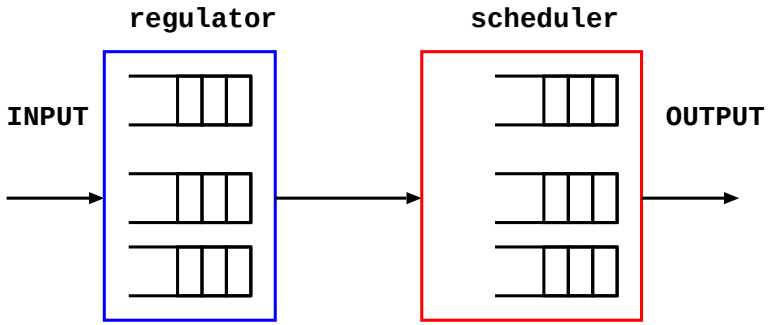
- To get a delay bound, need to pick $g(i)$.
  - the lower the delay bounds, the larger $g(i)$ needs to be
  - large $g(i) \longrightarrow$ exclusion of more competitors from link
  - $g(i)$ can be very large, in some cases 80 times the peak rate!
- Sources must be leaky-bucket regulated
  - choosing leaky-bucket parameters *may be* problematic
- WFQ couples delay and bandwidth allocations
  - low delay requires allocating more bandwidth
  - wastes bandwidth for low-bandwidth low-delay sources

Rate-controlled scheduling

- Earliest-Due-Date: packet with earliest deadline selected
- Delay-EDD prescribes how to assign deadlines to packets
- A source is required to send slower than its peak rate
- Bandwidth at scheduler reserved at peak rate
- Deadline is expected arrival time + delay bound. If source sends faster than contract, delay bound will not apply.
- Each packet gets a hard delay bound
- Delay bound is *independent* of bandwidth requirement. But reservation is at a connection's peak rate.
- Implementation requires per-connection sate and priority queue.

Rate-controlled scheduling

- A class of disciplines
  - two components: regulator and scheduler
  - incoming packets are placed in regulator where they wait to become eligible
  - then they are put in the scheduler
- Regulator shapes the traffic, scheduler provides performance guarantees



Examples

- Recall
  - rate-jitter regulator: bounds maximum outgoing rate
  - delay-jitter regulator compensates for variable delay at previous hop
- Rate-jitter regulator + FIFO
  - similar to Delay-EDD
- Rate-jitter regulator + multi-priority FIFO
  - gives both bandwidth and delay guarantees (RCSP)
- Delay-jitter regulator + EDD
  - gives bandwidth, delay,and delay-jitter bounds (Jitter-EDD)

Analysis

- First regulator on path monitors and regulates traffic $\Longrightarrow$ bandwidth bound
- End-to-end delay bound
  - delay-jitter regulator
    - reconstructs traffic $\longrightarrow$ end-to-end delay is fixed (= worst-case delay at each hop)
  - rate-jitter regulator
    - partially reconstructs traffic
    - can show that end-to-end delay bound is smaller than (sum of delay bound at each hop + delay at first hop)

Decoupling

- Can give a low-bandwidth connection a low delay without overbooking
- E.g consider connection A with rate 64 Kbps sent to a router with rate-jitter regulation and multi-priority FCFS scheduling
- After sending a packet of length l, next packet is eligible at time (now + l/64 Kbps)
- If placed at highest-priority queue, all packets from A get low delay
- Can decouple delay and bandwidth bounds, unlike WFQ

Evaluation

- Pros
  - flexibility: ability to emulate other disciplines
  - can decouple bandwidth and delay assignments
  - end-to-end delay bounds are easily computed
  - do not require complicated schedulers to guarantee protection
  - can provide delay-jitter bounds
- Cons
  - require an additional regulator at each output port
  - delay-jitter bounds at the expense of increasing mean delay
  - delay-jitter regulation is expensive (clock synch, timestamps)

Summary

- Two sorts of applications: best effort and guaranteed service
- Best effort connections require fair service
  - provided by GPS, which is unimplementable
  - emulated by WFQ and its variants
- Guaranteed service connections require performance guarantees
  - provided by WFQ, but this is expensive
  - may be better to use rate-controlled schedulers

Outline

- What is scheduling
- Why we need it
- Requirements of a scheduling discipline
- Fundamental choices
- Scheduling best effort connections
- Scheduling guaranteed-service connections
- Packet drop strategies

Packet Dropping

- Packets that cannot be served immediately are buffered
- Full buffers $\Longrightarrow$ packet drop strategy
- Packet losses happen almost always from best-effort connections (why?)
- Shouldn't drop packets unless imperative
  - packet drop wastes resources (why?)

1. Degree of aggregation

- Degree of discrimination in selecting a packet to drop
- E.g. in vanilla FIFO, all packets are in the same class
- Instead, can classify packets and drop packets selectively
- The finer the classification the better the protection
- Max-min fair allocation of buffers to classes
  - drop packet from class with the longest queue (why?)

2. Drop Priorities

- Drop lower-priority packets first
- How to choose?
  - endpoint marks packets
  - regulator marks packets
  - congestion loss priority (CLP) bit in packet header

CLP bit: pros and cons

- Pros
  - if network has spare capacity, all traffic is carried
  - during congestion, load is automatically shed
- Cons
  - separating priorities within a single connection is hard
  - what prevents all packets being marked as high priority?

Drop priority (contd.)

- Special case of AAL5
  - want to drop an entire frame, not individual cells
  - cells belonging to the selected frame are preferentially dropped
- Drop packets from ”nearby” hosts first
  - because they have used the least network resources
  - can’t do it on Internet because hop count (TTL) decreases

Early vs. Late Drop

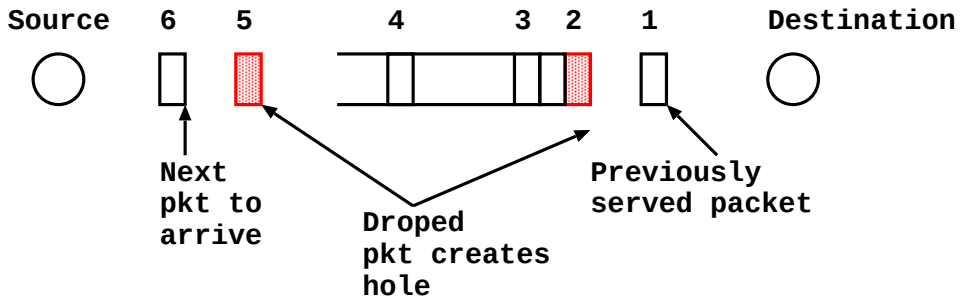
- Early drop $\Longrightarrow$ drop even if space is available
  - signals endpoints to reduce rate
  - cooperative sources get lower overall delays, uncooperative sources get severe packet loss
- Early random drop
  - drop arriving packet with fixed drop probability if queue length exceeds threshold
  - intuition: misbehaving sources more likely to send packets and see packet losses
  - doesn't work!

Early vs. late Drop: RED

- Random early detection (RED) makes three improvements
- Metric is moving average of queue lengths
  - small bursts pass through unharmed
  - only affects sustained overloads
- Packet drop probability is a function of mean queue length
  - prevents severe reaction to mild overload
- Can mark packets instead of dropping them
  - allows sources to detect network state without losses
- RED improves performance of a network of cooperating TCP sources
- No bias against bursty sources
- Controls queue length regardless of endpoint cooperation

Drop Position

- Can drop a packet from head, tail, or random position in the queue
- Tail
  - easy
  - default approach
- Head
  - harder
  - lets source detect loss earlier



Drop position (contd.)

- Random
  - hardest
  - if no aggregation, hurts hogs most
  - unlikely to make it to real routers
- Drop entire longest queue
  - easy
  - almost as effective as drop tail from longest queue