Episode 5. Scheduling and Traffic Management

Part 2

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

Keshav Chapter 9.4, 9.5.1, 9.5.2, 9.5.3, 13.3.4

Outline

What is scheduling?

Why do we need it?

Requirements of a scheduling discipline

Fundamental choices

Scheduling best effort connections

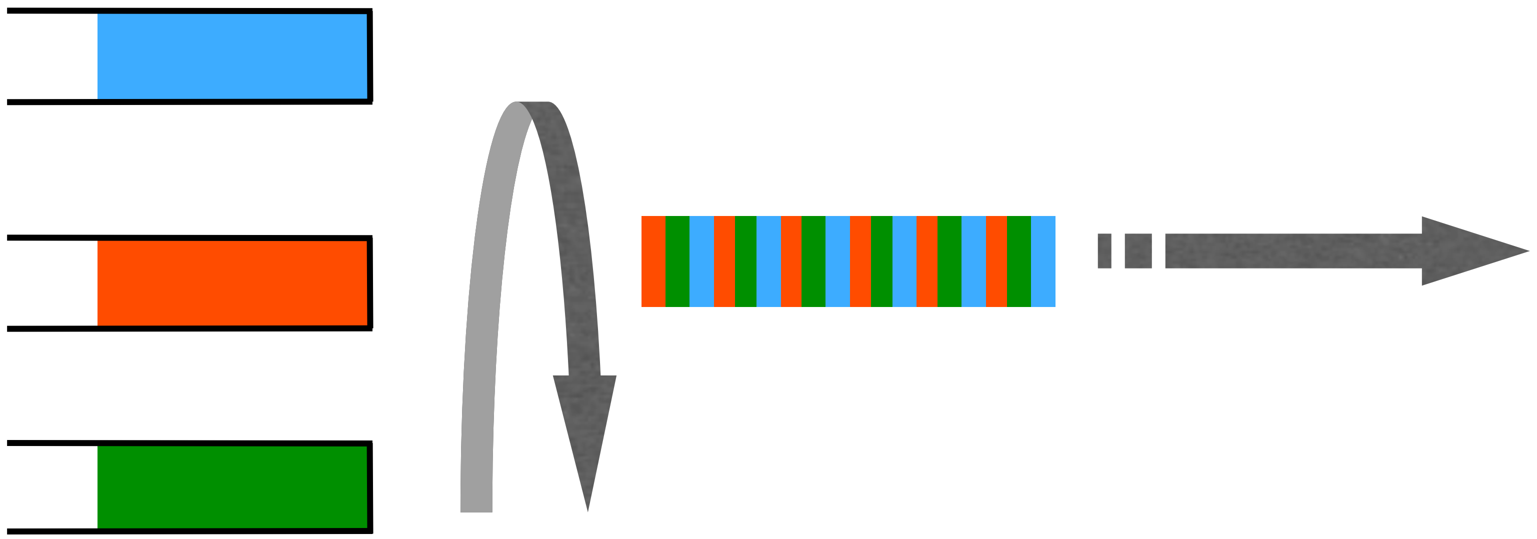
Scheduling guaranteed-service connections

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 queue in a finite time interval
- Why is this max-min fair?
- How can we give weights to connections?



More on Generalized Processor Sharing

GPS is not implementable!

we cannot serve **infinitesimals**, only packets

No packet discipline can be as fair as GPS: but how closely?

while a packet is being served, we are unfair to others

Define: work(i, a, b) = # bits transmitted for connection i in time [a,b]

Absolute fairness bound for scheduling discipline S

max $(\mathrm{work}_{GPS}(i, a, b) - \mathrm{work}_{S}(i, a, b))$

Relative fairness bound for scheduling discipline S

max $(\mathrm{work}_{S}(i, a, b) - \mathrm{work}_{S}(j, a, b))$

What is next?

We can’t implement GPS

So, let’s see how to emulate it

We want to be as fair as possible

But also have an efficient implementation

Weighted Round Robin

Round Robin: Serve a packet from each non-empty queue in turn

Unfair if packets are of different length or weights are not equal — Weighted Round Robin

Different weights, fixed size packets

- serve more than one packet per visit, after normalizing to obtain integer weights

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

Fair only over time scales longer than a round time

- At shorter time scales, some connections may get more service than others
- If a connection has a small weight, or the number of connections is large, this may lead to long periods of unfairness

For example —

- T3 trunk with 500 connections, each connection has a mean packet size of 500 bytes, 250 with weight 1, 250 with weight 10
- Each packet takes $500 \times 8/45$ Mbps = 88.8 microseconds
- Round time =$2750 \times 88.8 = 244.2$ ms

Deficit Round Robin (DRR)

DRR modifies weighted round-robin to allow it to handle variable packet sizes without knowing the mean packet size of each connection in advance

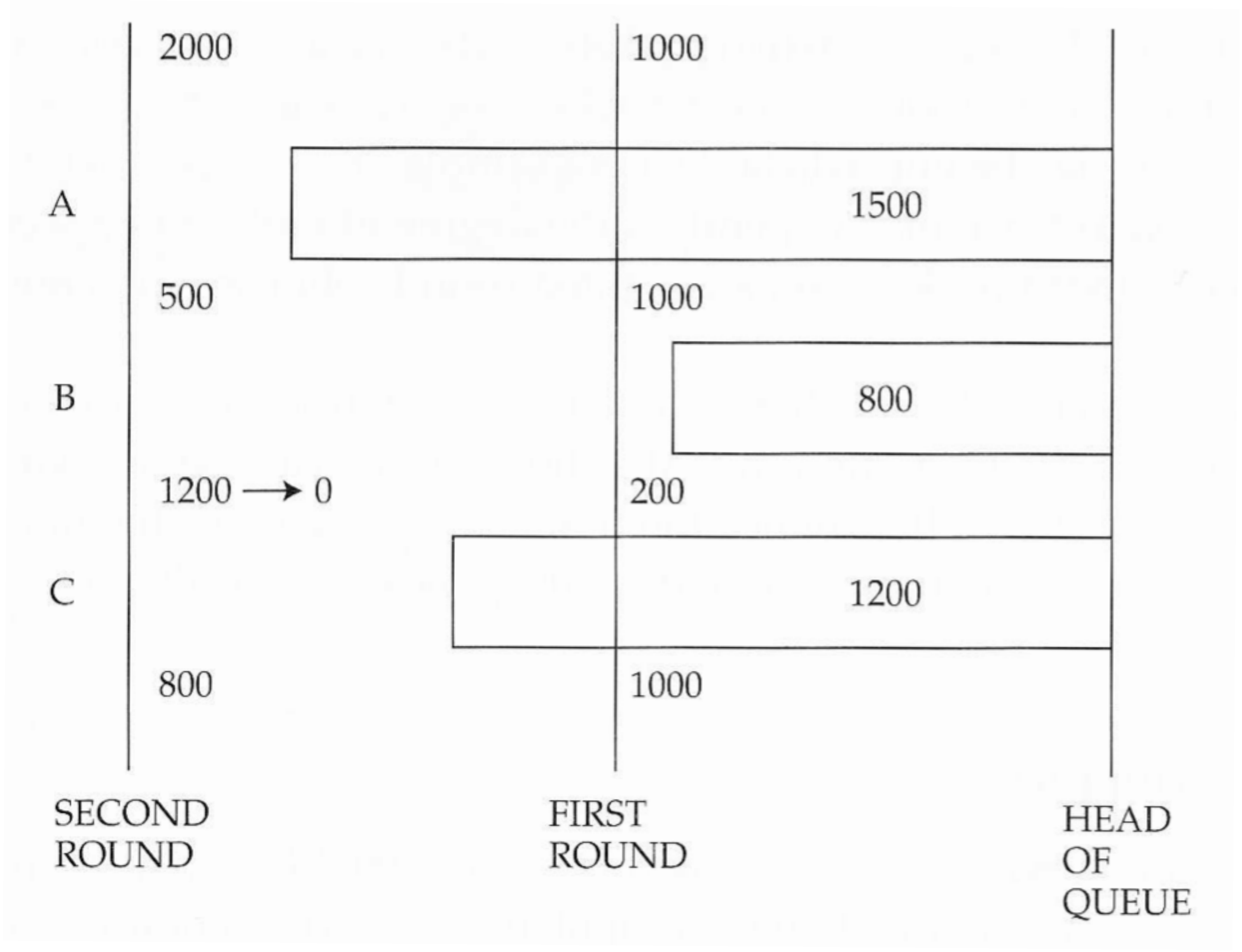
It associates each connection with a deficit counter, initialized to 0

It visits each connection in turn, tries to serve one quantum worth of bits from each visited connection

A packet at the head of queue is served if it is no larger than the sum of the quantum and the deficit counter (which is then adjusted)

If it is larger, the quantum is added to the connection's deficit counter

Deficit Round Robin: Example



Deficit Round Robin (DRR): Pros and Cons

Pros

Ease of implementation: O(1) scheduler — constant amount of work per step of its execution

Relative fairness bound: $3 \times F / r$, where F is the frame time (the largest possible time taken to serve each of the backlogged connections), and r is the rate of the server

Cons

Unfair at time scales smaller than a frame time

For example: T3 trunk (45 Mbps) with 500 connections and a packet size of 8 Kbytes, the frame time is

as large as 728 ms

Suitable when fairness requirements are loose or when packets are small

Live Demo: Deficit Round Robin in ns.py

Next-generation, Python-based discrete-event simulator for network protocols

https://github.com/TL-System/ns.py

hand-crafted by yours truly

Weighted Fair Queueing (WFQ)

Deals better with variable size packets and weights

Do not need to know a connection's mean packet size in advance

Basic idea —

- Simulates GPS "on the side" and uses the results to determine service order
- Compute the time a packet would complete service (finish time), had we been serving packets with GPS, bit by bit
- Then serve packets in order of their finish times
  - The finish time is just a service tag, and has nothing to do with the actual time at which the packet is served
  - We can more appropriately call it a **finish number**

WFQ: a First Cut

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

If a packet of length p bits arrives to an empty queue when the round number is R, it will complete service when the round number is R + p => finish number is R + p

- independent of the number of other connections!

If a packet arrives to a non-empty queue, and the previous packet has a finish number of f, then the packet's finish number is f + p

Serve packets in the order of finish numbers

WFQ: A 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

Solution: 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: Summary so far

To sum up, assuming we know the current round number R

Finish number of packet of length p

- If arriving to active connection = previous finish number + **p**
- If arriving to an inactive connection = **R + p**

To implement WFQ, we need to know two things:

- Is the connection active?
- If not, what is the current round number?

Answer to both questions depends on computing the current round number

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 connections join 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

With this change, WFQ emulates GPS instead of bit-by-bit Round Robin

WFQ: An Example

Packets of size 1, 2, and 2 units arrive at a WFQ scheduler at time 0, on equally weighted connections A, B, and C

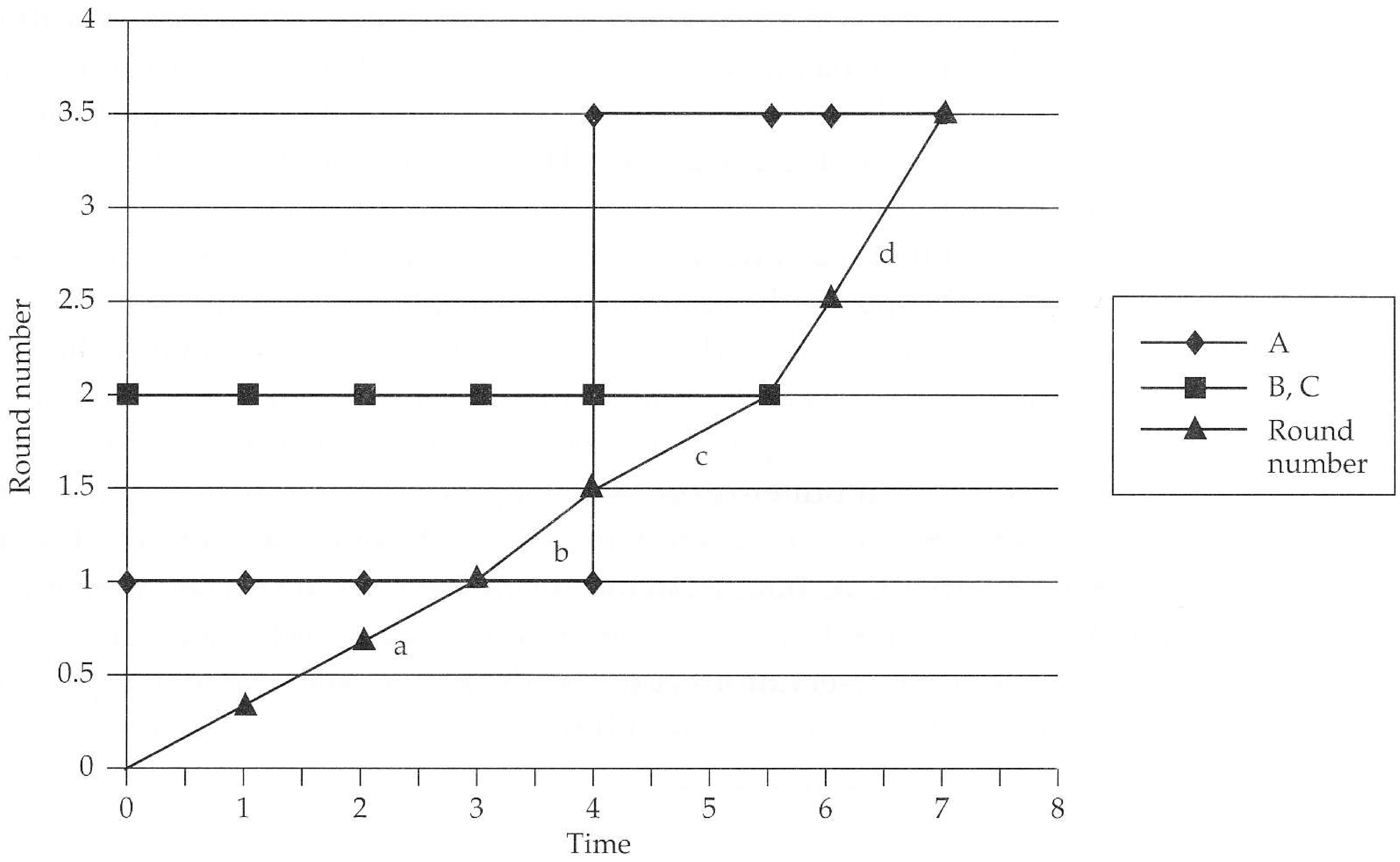
A packet of size 2 arrives at connection A at time 4

The link service rate is 1 unit/second

Questions

- What are the finish numbers of all packets?
- What is the round number when the system becomes idle?
- When does this happen?

WFQ: An Example



Problem: Iterated Deletion

Suppose the round number at time 0 is 0, and has a slope of 1/5

We may incorrectly suppose that, at time 5, it would have a value of $0 + 1/5 \times 5 = 1$

However, our computation may be wrong, because one of the five connections active at time 0 may become inactive before time 5

Say, connection 1 becomes inactive at time 4, so at time 4, the slope of the round number line increases from 1/5 to 1/4

At time 5, the round number would be $4 \times 1/5 + 1 \times 1/4 = 1.05$

Suppose connection 2 has a finish number smaller than 1.05

It would also become inactive before time 5, further affecting the slope of the round number — which may "delete" more connections!

Problem: Iterated Deletion

$$\begin{tikzcd}\text{Round number} \arrow[r, bend left] & \text{\# active connections} \arrow[l, bend left]\end{tikzcd}$$

A server recomputes the round number on each packet arrival and departure

At any re-computation, the number of connections can go up at most by one, but can go down to zero

To solve this problem:

- use previous count to compute round number
- if this makes some connection inactive, recompute
- repeat until no connections become inactive

WFQ implementation

On packet arrival

use the source and destination address to classify it to a connection, and look up the finish number of the last packet served (or waiting to be served)

recompute round number

compute finish number

insert in a priority queue sorted by finish numbers

On service completion

select the packet with the lowest finish number in the priority queue

WFQ: Analysis

If GPS has served x bits from connection A by time t

WFQ would have served at least $x - P_{max}$ bits, where P_{max} is the largest possible packet in the network

WFQ: Evaluation

Pros

like GPS, it provides protection

can obtain worst-case end-to-end delay bound (more details later)

gives users incentive to use intelligent flow control

Cons

needs per-connection state

iterated deletion is complicated

requires explicit sorting of the output queue

Virtual Clock

Similar to WFQ, proposed independently by L. Zhang (1990)

Tag values are not computed to emulate GPS

- Instead, a VC scheduler emulates time-division multiplexing in the same way that WFQ emulates GPS

A packet's finish number = max(previous finish number, real time) + p

- Replaced round number (hard to compute) with real time
- But relative fairness bound is infinity if not all connections are backlogged

Live Demo: WFQ and Virtual Clock in ns.py

Next-generation, Python-based discrete-event simulator for network protocols

https://github.com/TL-System/ns.py

$\boxed{\text{hand-crafted by yours truly}}$

Outline

What is scheduling?

Why do we need it?

Requirements of a scheduling discipline

Fundamental choices

Scheduling best effort connections

Scheduling guaranteed-service connections

Scheduling guaranteed-service connections

With best-effort connections, the goal is fairness

With guaranteed-service connections

- What performance guarantees are achievable?
- How easy is admission control?

Weighted Fair Queuing revisited

Turns out that WFQ also provides performance guarantees

Bandwidth bound

ratio of weights × link capacity

Example: 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

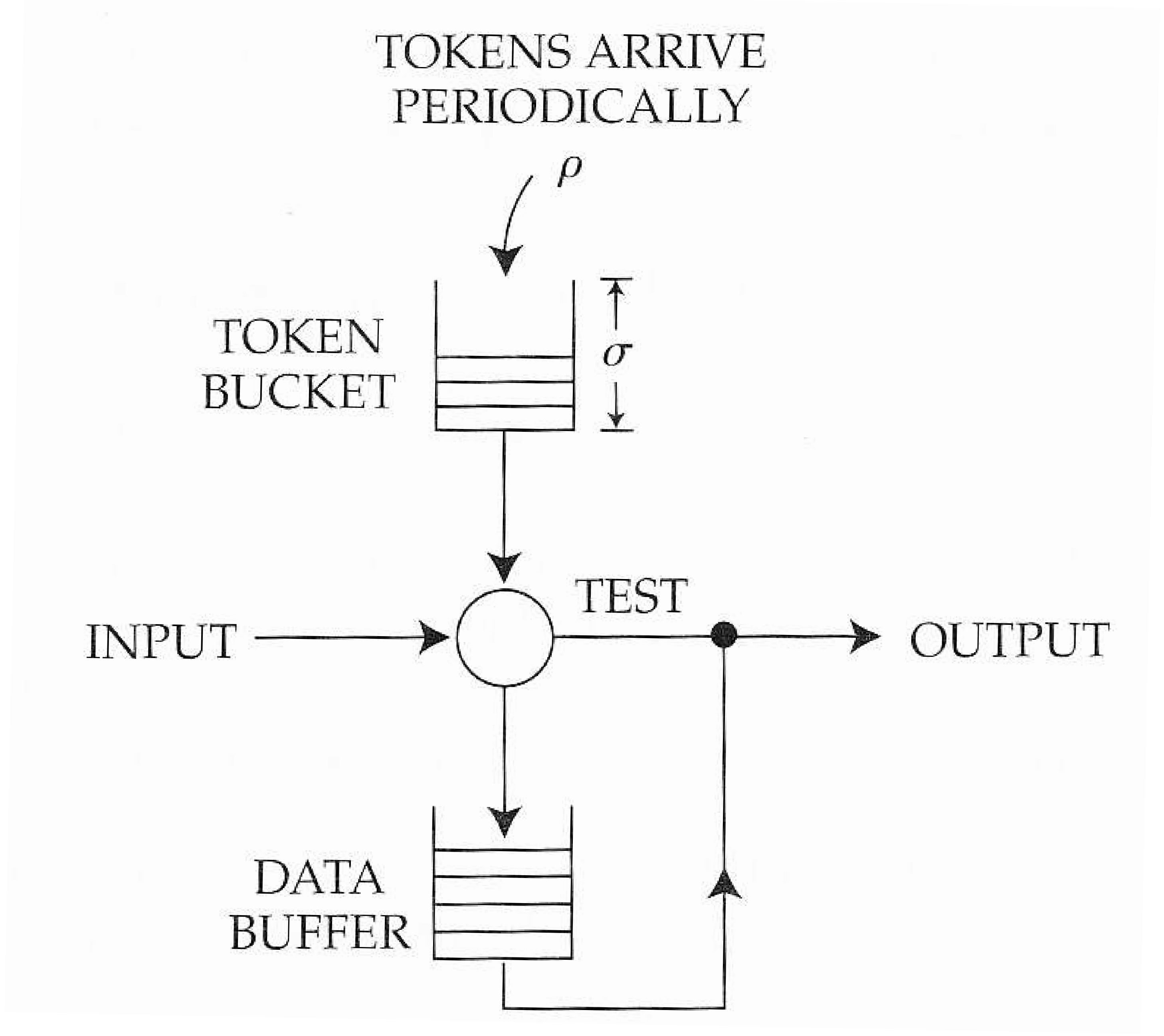
assumes that the connection doesn't send "too much" (otherwise its packets will be stuck in queues)

more precisely, connection should be leaky-bucket regulated

Leaky-bucket regulators

Implement linear bounded arrival processes

# bits transmitted in time $[t_1, t_2] \leq \rho (t_2 - t_1) + \sigma$



Special cases of a leaky-bucket regulator

Peak-rate regulator: Setting the token-bucket limit to one token, and the token replenishment interval to the peak rate

Augment a leaky-bucket regulator with a peak-rate regulator: controls all three parameters: the average rate, the peak rate, and the largest burst

(Keshav, Ch. 13.3.4)

Parekh-Gallager Theorem

Let a connection be allocated weights at each WFQ scheduler along its path, so that the least bandwidth it is allocated is *g*

Let it be leaky-bucket regulated such that # bits sent in time $[t_1, t_2] \leq \rho\ (t_2 - t_1) + \sigma$

Let the connection pass through *K* schedulers, where the *k*th scheduler has a link rate *r(k)*

Let the largest packet allowed in the network be *P*

$$D^* \leq \frac{\sigma}{g} + \sum_{k=1}^{K-1} \frac{P}{g} + \sum_{k=1}^{K} \frac{P}{r(k)}$$

Example

Consider a connection with leaky bucket parameters (16384 bytes, 150 Kbps) that traverses 10 hops on a network where all the links have a bandwidth of 45 Mbps. If the largest allowed packet in the network is 8192 bytes long, what *g* value will guarantee an end-to-end delay of 100 ms? Assume a propagation delay of 30 ms.

Solution —

The queuing delay must be bounded by 100 - 30 = 70 ms. Plugging this into the previous equation, we get $70 \times 10^{-3} = \{(16384 \times 8) + (9 \text{ x } 8192 \times 8)\} / g + (10 \times 8192 \times 8) / (45 \times 10^{6})$, so that *g* = 12.87 Mbps.

This is more than 86 times larger than the source's average rate of 150 Kbps

With large packets, packet delays can be quite substantial!

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 behaviour

Problems

To get a delay bound, need to pick g

- the lower the delay bounds, the larger *g* needs to be
- large *g* => exclusion of more competitors from link
- *g* can be very large, in our example > 80 times the peak rate!

WFQ couples delay and bandwidth allocations

- low delay requires allocating more bandwidth
- wastes bandwidth for low-bandwidth low-delay sources

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

Keshav Chapter 9.4, 9.5.1, 9.5.2, 9.5.3, 13.3.4