Scheduling

Adnan Aziz

The University of Texas at Austin

Background

Whenever a resource is shared, it needs to be scheduled

- Student wants to take set of classes, which have precedence constraints
  - cannot take more than k classes per semester; how fast can he graduate?
- Select times for final exams
  - don't schedule two finals at the same time if a student is taking two classes, what's the smallest number of slots?

Best-effort vs. guaranteed services

Networks carry two kinds of applications:

- "elastic" "best-effort" — relatively insensitive to performance (e.g., file transfer)
- "guaranteed services" — need performance bounds (e.g., voice)
  - notion of a "connection" $\Rightarrow$ need resource reservation (on-the-fly, or at call establishment, or in advance)

Even for best-effort traffic, though no performance bounds needed, would like "fair" allocation

Simplest scheduling discipline is FCFS with tail drop

- easy to implement, cannot improve upon mean delay of FCFS
- offers no protection

Max-min fair share

Sources numbered $1 \ldots n$, resource demands $x_1, \ldots, x_n$, service capacity C

- WLOG assume $x_1 < x_2, \cdots < x_n$

Give C/n to source 1; if more than x_1, divide $c/n - x_1$ equally amnesty rest

- iterate over sources $2, \ldots, n$

Generalize to weights on sources $w_1, \ldots, w_n$

- normalize the weights, give to sources in proportion to its weight

Local/global scheduling

The above scheduling is local to the router

- what does it achieve globally?

If each connection limits its usage to the smallest locally fair allocation, the allocation is globally fair

- because of propagation delays, may not track perfectly

Scheduler requirements

- easy to implement
- offer fairness and protection
- give performance bounds
- easy and fast admission control

Measure complexity as a function of N, the number of simultaneous connections

Max-min fair share

Divide scarce resource among a set of users

- each has equal right, but some demand less

Idea of max-min fair share: users with small demand get what they want, remainder equally balanced

GPS

Generalized Processor Sharing

- N connections with equal weights — N queues

Logically, serve infinitesimal amount from each queue

- skip empty queues

Achieves max-min fair share

- in presence of weights, serve $w_i \cdot dt$

Packets aren't real numbers ⇒ unimplementable.

Weighted Round Robin

- when packet sizes same and weights are equal
  - serve a packet from each nonempty queue ⇒ good approximation to GPS
- WRR with equal sized packets: normalize weights, round time is sum of weights
  - Example — $0.5, 0.75, 1.0 \Rightarrow 2, 3, 4$
  - serve 2 packets from first, 3 from second, 4 from third in each round
- different sized packets — divide weight by mean size

Problems: (1.) hard to know mean size, (2.) fair only over a round (which could be long)

Performance parameters

- Bandwidth bound for pre-specified connections
- Delay bounds
  - worst case
  - average case (need to consider all possible arrivals for other connections — almost impossible to compute analytically)
  -
  - 99%ile
  - delay jitter (max - min)
- loss bound

Some options

- number of priority levels
  - each connection has a priority level, serve higher priority packets first
  - easy to implement, low state overhead
  - no protection within class
- work conserving/nonwork conserving
  - may be useful — reduce burstiness downstream

Efficient Implementation of Timers

Many applications, including packet scheduling

Timer module has 3 components:

- `startTimer( interval, reqID, expiryAction );` (interval relative or absolute)
- `stopTimer( reqID );`
- `perTickBookkeeping();`

First two client calls; third called every one time unit.

Two performance measures for these functions (measured by n, number of outstanding timers):

- memory used
- time taken

Scheme I

One memory location per timer: `startTimer` and `stopTimer` trivial

- `perTickBookkeeping` — decrement each outstanding timer
  - becomes 0 — call appropriate expiry action

`perTickBookkeeping` as very high cost, especially when timers last a long time. Good when few timers/stop soon/custom hardware.

Deficit Round Robin

For variable sized packets when mean size not known in advance

- scheduler has a "deficit counter" for each connection
  - there is a fixed "quantum" size (say 1000 bytes)
  - serve packet if it's smaller than quantum size, else add quantum to deficit counter
  - if quantum size + deficit counter $>$ packet size, send packet, subtract packet size from deficit counter
  - set deficit counter to 0 if queue is empty
  - setting quantum size to at least max packet size keep scheduler work conserving

Large frame size: 45 MBps, 500 connections, 8 kByte pkts $\Rightarrow$ frame size is 725 ms

Weighted Fair Queueing: compute GPS finish times to determine service order

Scheme IV

Timing wheel — insert timers into an array of lists

- overflow list for timers beyond end of the array

Each cycle is N units: suppose S cycles so far, and pointer points to j — current time is $N \cdot S + j$

- increment current time pointer modulo N; if it wraps to 0, check overflow list, add to array of lists
- if all time intervals are less than *MaxInterval*, just wrap around

More likely to insert in overflow lists as reach end $\Rightarrow$ spin around every $N/2$ units

Good when: (1.) short intervals, (2.) few cancellations

Schemes V & VI

V: analogous to hashing

- 32 bit timer, table with 256 entries
- last 8 bits are 00010100 (dec 20), current pointer is 10, store at location 30
  - keep sorted collision chain

Avg time for all is $O(1)$

VI: unsorted collision chains $\Rightarrow$ perTickBookkeeping is $O(1)$ on average, $O(n)$ over n ticks guaranteed

Scheme II

Keep ordered list of timers

- store absolute time in ordered list
- at each tick examine head

`startTimer` has $\Theta(n)$ time complexity

Scheme III

Basically, we're implementing a priority queue with deletes:

- use a balanced binary search tree (AVL, red-black, etc.)

`startTimer`, `perTickBookkeeping`, `stopTimer` take $O(\log n)$

- `stopTimer` doesn't have to be supported $\Rightarrow$ heap suffices $O(\log n)$

Scheme VII

Use hierarchy (odometer approach)

Here's how to encode 8.64 million possible times with only 244 memory locations:

- 100 elt array for days
- 24 elt array for hours
- 60 elt array for minutes
- 60 elt array for seconds

Current time is 11 days, 10 hours, 24 mins, 30 secs

- set timer for 50 mins, 45 secs in future
  - calculate: expires at 11/11/15/15
  - insert into list at 1 ahead of current hour, store 15 mins/15 secs at this location
  - update minutes at this hour
- startTimer and stopTimer take $O(m)$, where m is the number of hierarchy levels — 2 to 5