NAME

tc-hfcs − Hierarchical Fair Service Curve

HISTORY & INTRODUCTION

HFSC (Hierarchical Fair Service Curve) is a network packet scheduling algorithm that was first presented at SIGCOMM'97. Developed as a part of ALTQ (ALTernative Queuing) on NetBSD, found its way quickly to other BSD systems, and then a few years ago became part of the linux kernel. Still, it's not the most popular scheduling algorithm − especially if compared to HTB − and it's not well documented for the enduser. This introduction aims to explain how HFSC works without using too much math (although some math it will be inevitable).

In short HFSC aims to:

1) guarantee precise bandwidth and delay allocation for all leaf classes (realtime criterion)

2) allocate excess bandwidth fairly as specified by class hierarchy (linkshare & upperlimit criterion)

3) minimize any discrepancy between the service curve and the actual amount of service provided during linksharing

The main "selling" point of HFSC is feature **(1)**, which is achieved by using nonlinear service curves (more about what it actually is later). This is particularly useful in VoIP or games, where not only a guarantee of consistent bandwidth is important, but also limiting the initial delay of a data stream. Note that it matters only for leaf classes (where the actual queues are) − thus class hierarchy is ignored in the realtime case.

Feature **(2)** is well, obvious − any algorithm featuring class hierarchy (such as HTB or CBQ) strives to achieve that. HFSC does that well, although you might end with unusual situations, if you define service curves carelessly − see section CORNER CASES for examples.

Feature **(3)** is mentioned due to the nature of the problem. There may be situations where it's either not possible to guarantee service of all curves at the same time, and/or it's impossible to do so fairly. Both will be explained later. Note that this is mainly related to interior (aka aggregate) classes, as the leafs are already handled by **(1)**. Still, it's perfectly possible to create a leaf class without realtime service, and in such a case the caveats will naturally extend to leaf classes as well.

ABBREVIATIONS

For the remaining part of the document, we'll use following shortcuts:

RT − realtime
LS − linkshare
UL − upperlimit
SC − service curve

BASICS OF HFSC

To understand how HFSC works, we must first introduce a service curve. Overall, it's a nondecreasing function of some time unit, returning the amount of service (an allowed or allocated amount of bandwidth) at some specific point in time. The purpose of it should be subconsciously obvious: if a class was allowed to transfer not less than the amount specified by its service curve, then the service curve is not violated.

Still, we need more elaborate criterion than just the above (although in the most generic case it can be reduced to it). The criterion has to take two things into account:

- idling periods
- the ability to "look back", so if during current active period the service curve is violated, maybe it isn't if we count excess bandwidth received during earlier active period(s)

Let's define the criterion as follows:

(1) For each t1, there must exist t0 in set B, so S(t1−t0) <= w(t0,t1)

Here ’w’ denotes the amount of service received during some time period between t0 and t1. B is a set of all times, where a session becomes active after idling period (further denoted as ’becoming backlogged’). For a clearer picture, imagine two situations:

a) our session was active during two periods, with a small time gap between them

b) as in (a), but with a larger gap

Consider **(a)**: if the service received during both periods meets **(1)**, then all is well. But what if it doesn’t do so during the 2nd period? If the amount of service received during the 1st period is larger than the service curve, then it might compensate for smaller service during the 2nd period *and* the gap − if the gap is small enough.

If the gap is larger **(b)** − then it’s less likely to happen (unless the excess bandwidth allocated during the 1st part was really large). Still, the larger the gap − the less interesting is what happened in the past (e.g. 10 minutes ago) − what matters is the current traffic that just started.

From HFSC’s perspective, more interesting is answering the following question: when should we start transferring packets, so a service curve of a class is not violated. Or rephrasing it: How much X() amount of service should a session receive by time t, so the service curve is not violated. Function X() defined as below is the basic building block of HFSC, used in: eligible, deadline, virtual−time and fit−time curves. Of course, X() is based on equation **(1)** and is defined recursively:

- At the 1st backlogged period beginning function X is initialized to generic service curve assigned to a class
- At any subsequent backlogged period, X() is:
  min(X() from previous period ; w(t0)+S(t−t0) for t>=t0),
  ... where t0 denotes the beginning of the current backlogged period.

HFSC uses either linear, or two−piece linear service curves. In case of linear or two−piece linear convex functions (first slope < second slope), min() in X’s definition reduces to the 2nd argument. But in case of two−piece concave functions, the 1st argument might quickly become lesser for some t>=t0. Note, that for some backlogged period, X() is defined only from that period’s beginning. We also define Xˆ(−1)(w) as smallest t>=t0, for which X(t) = w. We have to define it this way, as X() is usually not an injection.

The above generic X() can be one of the following:

E() In realtime criterion, selects packets eligible for sending. If none are eligible, HFSC will use linkshare criterion. Eligible time ’et’ is calculated with reference to packets’ heads (et = Eˆ(−1)(w)). It’s based on RT service curve, *but in case of a convex curve, uses its 2nd slope only.*

D() In realtime criterion, selects the most suitable packet from the ones chosen by E(). Deadline time ’dt’ corresponds to packets’ tails (dt = Dˆ(−1)(w+l), where ’l’ is packet’s length). Based on RT service curve.

V() In linkshare criterion, arbitrates which packet to send next. Note that V() is function of a virtual time − see **LINKSHARE CRITERION** section for details. Virtual time ’vt’ corresponds to packets’ heads (vt = Vˆ(−1)(w)). Based on LS service curve.

F() An extension to linkshare criterion, used to limit at which speed linkshare criterion is allowed to dequeue. Fit−time ’ft’ corresponds to packets’ heads as well (ft = Fˆ(−1)(w)). Based on UL service curve.

Be sure to make clean distinction between session’s RT, LS and UL service curves and the above "utility" functions.

REALTIME CRITERION

RT criterion *ignores class hierarchy* and guarantees precise bandwidth and delay allocation. We say that a packet is eligible for sending, when the current real time is later than the eligible time of the packet. From all eligible packets, the one most suited for sending is the one with the shortest deadline time. This sounds simple, but consider the following example:

Interface 10Mbit, two classes, both with two−piece linear service curves:

- 1st class − 2Mbit for 100ms, then 7Mbit (convex − 1st slope < 2nd slope)
- 2nd class − 7Mbit for 100ms, then 2Mbit (concave − 1st slope > 2nd slope)

Assume for a moment, that we only use D() for both finding eligible packets, and choosing the most fitting one, thus eligible time would be computed as Dˆ(−1)(w) and deadline time would be computed as Dˆ(−1)(w+l). If the 2nd class starts sending packets 1 second after the 1st class, it's of course impossible to guarantee 14Mbit, as the interface capability is only 10Mbit. The only workaround in this scenario is to allow the 1st class to send the packets earlier that would normally be allowed. That's where separate E() comes to help. Putting all the math aside (see HFSC paper for details), E() for RT concave service curve is just like D(), but for the RT convex service curve − it's constructed using *only* RT service curve's 2nd slope (in our example
7Mbit).

The effect of such E() − packets will be sent earlier, and at the same time D() *will* be updated − so the current deadline time calculated from it will be later. Thus, when the 2nd class starts sending packets later, both the 1st and the 2nd class will be eligible, but the 2nd session's deadline time will be smaller and its packets will be sent first. When the 1st class becomes idle at some later point, the 2nd class will be able to "buffer" up again for later active period of the 1st class.

A short remark − in a situation, where the total amount of bandwidth available on the interface is larger than the allocated total realtime parts (imagine a 10 Mbit interface, but 1Mbit/2Mbit and 2Mbit/1Mbit classes), the sole speed of the interface could suffice to guarantee the times.

Important part of RT criterion is that apart from updating its D() and E(), also V() used by LS criterion is updated. Generally the RT criterion is secondary to LS one, and used *only* if there's a risk of violating precise realtime requirements. Still, the "participation" in bandwidth distributed by LS criterion is there, so V() has to be updated along the way. LS criterion can than properly compensate for non−ideal fair sharing situation, caused by RT scheduling. If you use UL service curve its F() will be updated as well (UL service curve is an extension to LS one − see **UPPERLIMIT CRITERION** section).

Anyway − careless specification of LS and RT service curves can lead to potentially undesired situations (see CORNER CASES for examples). This wasn't the case in HFSC paper where LS and RT service curves couldn't be specified separately.

LINKSHARING CRITERION

LS criterion's task is to distribute bandwidth according to specified class hierarchy. Contrary to RT criterion, there're no comparisons between current real time and virtual time − the decision is based solely on direct comparison of virtual times of all active subclasses − the one with the smallest vt wins and gets scheduled. One immediate conclusion from this fact is that absolute values don't matter − only ratios between them (so for example, two children classes with simple linear 1Mbit service curves will get the same treatment from LS criterion's perspective, as if they were 5Mbit). The other conclusion is, that in perfectly fluid system with linear curves, all virtual times across whole class hierarchy would be equal.

Why is VC defined in term of virtual time (and what is it)?

Imagine an example: class A with two children − A1 and A2, both with let's say 10Mbit SCs. If A2 is idle, A1 receives all the bandwidth of A (and update its V() in the process). When A2 becomes active, A1's virtual time is already *far* later than A2's one. Considering the type of decision made by LS criterion, A1 would become idle for a long time. We can workaround this situation by adjusting virtual time of the class becoming active − we do that by getting such time "up to date". HFSC uses a mean of the smallest and the biggest virtual time of currently active children fit for sending. As it's not real time anymore (excluding trivial case of situation where all classes become active at the same time, and never become idle), it's called virtual time.

Such approach has its price though. The problem is analogous to what was presented in previous section and is caused by non−linearity of service curves:

1) either it's impossible to guarantee service curves and satisfy fairness during certain time periods:

   Recall the example from RT section, slightly modified (with 3Mbit slopes instead of 2Mbit ones):

   - 1st class − 3Mbit for 100ms, then 7Mbit (convex − 1st slope < 2nd slope)
   - 2nd class − 7Mbit for 100ms, then 3Mbit (concave − 1st slope > 2nd slope)

   They sum up nicely to 10Mbit − the interface's capacity. But if we wanted to only use LS for guarantees and fairness − it simply won't work. In LS context, only V() is used for making decision which class to schedule. If the 2nd class becomes active when the 1st one is in its second slope, the fairness will be preserved − ratio will be 1:1 (7Mbit:7Mbit), but LS itself is of course unable to guarantee the absolute values themselves − as it would have to go beyond of what the interface is capable of.

2) and/or it's impossible to guarantee service curves of all classes at the same time [fairly or not]:

   This is similar to the above case, but a bit more subtle. We will consider two subtrees, arbitrated by their common (root here) parent:

   R (root) - 10Mbit

   A − 7Mbit, then 3Mbit
   A1 − 5Mbit, then 2Mbit
   A2 − 2Mbit, then 1Mbit

   B − 3Mbit, then 7Mbit

   R arbitrates between left subtree (A) and right (B). Assume that A2 and B are constantly backlogged, and at some later point A1 becomes backlogged (when all other classes are in their 2nd linear part).

   What happens now? B (choice made by R) will *always* get 7 Mbit as R is only (obviously) concerned with the ratio between its direct children. Thus A subtree gets 3Mbit, but its children would want (at the point when A1 became backlogged) 5Mbit + 1Mbit. That's of course impossible, as they can only get 3Mbit due to interface limitation.

   In the left subtree − we have the same situation as previously (fair split between A1 and A2, but violated guarantees), but in the whole tree − there's no fairness (B got 7Mbit, but A1 and A2 have to fit together in 3Mbit) and there's no guarantees for all classes (only B got what it wanted). Even if we violated fairness in the A subtree and set A2's service curve to 0, A1 would still not get the required bandwidth.

UPPERLIMIT CRITERION

UL criterion is an extensions to LS one, that permits sending packets only if current real time is later than fit−time ('ft'). So the modified LS criterion becomes: choose the smallest virtual time from all active children, such that fit−time < current real time also holds. Fit−time is calculated from F(), which is based on UL service curve. As you can see, its role is kinda similar to E() used in RT criterion. Also, for obvious reasons − you can't specify UL service curve without LS one.

The main purpose of the UL service curve is to limit HFSC to bandwidth available on the upstream router (think adsl home modem/router, and linux server as NAT/firewall/etc. with 100Mbit+ connection to mentioned modem/router). Typically, it's used to create a single class directly under root, setting a linear UL service curve to available bandwidth − and then creating your class structure from that class downwards. Of course, you're free to add a UL service curve (linear or not) to any class with LS criterion.

An important part about the UL service curve is that whenever at some point in time a class doesn't qualify for linksharing due to its fit−time, the next time it does qualify it will update its virtual time to the smallest virtual time of all active children fit for linksharing. This way, one of the main things the LS criterion tries to achieve − equality of all virtual times across whole hierarchy − is preserved (in perfectly fluid system with only linear curves, all virtual times would be equal).

Without that, 'vt' would lag behind other virtual times, and could cause problems. Consider an interface with a capacity of 10Mbit, and the following leaf classes (just in case you're skipping this text quickly − this example shows behavior that ***doesn't happen***):

A − ls 5.0Mbit
B − ls 2.5Mbit
C − ls 2.5Mbit, ul 2.5Mbit

If B was idle, while A and C were constantly backlogged, A and C would normally (as far as LS criterion is concerned) divide bandwidth in 2:1 ratio. But due to UL service curve in place, C would get at most 2.5Mbit, and A would get the remaining 7.5Mbit. The longer the backlogged period, the more the virtual times of A and C would drift apart. If B became backlogged at some later point in time, its virtual time would be set to (A's vt + C's vt)/2, thus blocking A from sending any traffic until B's virtual time catches up with A.

SEPARATE LS / RT SCs

Another difference from the original HFSC paper is that RT and LS SCs can be specified separately. Moreover, leaf classes are allowed to have only either RT SC or LS SC. For interior classes, only LS SCs make sense: any RT SC will be ignored.

CORNER CASES

Separate service curves for LS and RT criteria can lead to certain traps that come from "fighting" between ideal linksharing and enforced realtime guarantees. Those situations didn't exist in original HFSC paper, where specifying separate LS / RT service curves was not discussed.

Consider an interface with a 10Mbit capacity, with the following leaf classes:

A − ls 5.0Mbit, rt 8Mbit
B − ls 2.5Mbit
C − ls 2.5Mbit

Imagine A and C are constantly backlogged. As B is idle, A and C would divide bandwidth in 2:1 ratio, considering LS service curve (so in theory − 6.66 and 3.33). Alas RT criterion takes priority, so A will get 8Mbit and LS will be able to compensate class C for only 2 Mbit − this will cause discrepancy between virtual times of A and C.

Assume this situation lasts for a long time with no idle periods, and suddenly B becomes active. B's virtual time will be updated to (A's vt + C's vt)/2, effectively landing in the middle between A's and C's virtual time. The effect − B, having no RT guarantees, will be punished and will not be allowed to transfer until C's virtual time catches up.

If the interface had a higher capacity, for example 100Mbit, this example would behave perfectly fine though.

Let’s look a bit closer at the above example − it "cleverly" invalidates one of the basic things LS criterion tries to achieve − equality of all virtual times across class hierarchy. Leaf classes without RT service curves are literally left to their own fate (governed by messed up virtual times).

Also, it doesn’t make much sense. Class A will always be guaranteed up to 8Mbit, and this is more than any absolute bandwidth that could happen from its LS criterion (excluding trivial case of only A being active). If the bandwidth taken by A is smaller than absolute value from LS criterion, the unused part will be automatically assigned to other active classes (as A has idling periods in such case). The only "advantage" is, that even in case of low bandwidth on average, bursts would be handled at the speed defined by RT criterion. Still, if extra speed is needed (e.g. due to latency), non linear service curves should be used in such case.

In the other words: the LS criterion is meaningless in the above example.

You can quickly "workaround" it by making sure each leaf class has RT service curve assigned (thus guaranteeing all of them will get some bandwidth), but it doesn’t make it any more valid.

Keep in mind - if you use nonlinear curves and irregularities explained above happen *only* in the first segment, then there’s little wrong with "overusing" RT curve a bit:

A − ls 5.0Mbit, rt 9Mbit/30ms, then 1Mbit
B − ls 2.5Mbit
C − ls 2.5Mbit

Here, the vt of A will "spike" in the initial period, but then A will never get more than 1Mbit until B & C catch up. Then everything will be back to normal.

LINUX AND TIMER RESOLUTION

In certain situations, the scheduler can throttle itself and setup so called watchdog to wakeup dequeue function at some time later. In case of HFSC it happens when for example no packet is eligible for scheduling, and UL service curve is used to limit the speed at which LS criterion is allowed to dequeue packets. It’s called throttling, and accuracy of it is dependent on how the kernel is compiled.

There’re 3 important options in modern kernels, as far as timers’ resolution goes: ’tickless system’, ’high resolution timer support’ and ’timer frequency’.

If you have ’tickless system’ enabled, then the timer interrupt will trigger as slowly as possible, but each time a scheduler throttles itself (or any other part of the kernel needs better accuracy), the rate will be increased as needed / possible. The ceiling is either ’timer frequency’ if ’high resolution timer support’ is not available or not compiled in, or it’s hardware dependent and can go *far* beyond the highest ’timer frequency’ setting available.

If ’tickless system’ is not enabled, the timer will trigger at a fixed rate specified by ’timer frequency’ − regardless if high resolution timers are or aren’t available.

This is important to keep those settings in mind, as in scenario like: no tickless, no HR timers, frequency set to 100hz − throttling accuracy would be at 10ms. It doesn’t automatically mean you would be limited to ˜0.8Mbit/s (assuming packets at ˜1KB) − as long as your queues are prepared to cover for timer inaccuracy. Of course, in case of e.g. locally generated UDP traffic − appropriate socket size is needed as well. Short example to make it more understandable (assume hardcore anti−schedule settings − HZ=100, no HR timers, no tickless):

```
tc qdisc add dev eth0 root handle 1:0 hfsc default 1
tc class add dev eth0 parent 1:0 classid 1:1 hfsc rt m2 10Mbit
```

Assuming packet of ˜1KB size and HZ=100, that averages to ˜0.8Mbit − anything beyond it (e.g. the above example with specified rate over 10x larger) will require appropriate queuing and cause bursts every ˜10 ms. As you can imagine, any HFSC's RT guarantees will be seriously invalidated by that. Aforementioned example is mainly important if you deal with old hardware − as is particularly popular for home server chores. Even then, you can easily set HZ=1000 and have very accurate scheduling for typical adsl speeds.

Anything modern (apic or even hpet msi based timers + 'tickless system') will provide enough accuracy for superb 1Gbit scheduling. For example, on one of my cheap dual-core AMD boards I have the following settings:

```
tc qdisc add dev eth0 parent root handle 1:0 hfsc default 1
tc class add dev eth0 parent 1:0 classid 1:1 hfsc rt m2 300mbit
```

And a simple:

```
nc −u dst.host.com 54321 </dev/zero
nc −l −p 54321 >/dev/null
```

...will yield the following effects over a period of ˜10 seconds (taken from /proc/interrupts):

```
319: 42124229   0  HPET_MSI−edge  hpet2 (before)
319: 42436214   0  HPET_MSI−edge  hpet2 (after 10s.)
```

That's roughly 31000/s. Now compare it with HZ=1000 setting. The obvious drawback of it is that cpu load can be rather high with servicing that many timer interrupts. The example with 300Mbit RT service curve on 1Gbit link is particularly ugly, as it requires a lot of throttling with minuscule delays.

Also note that it's just an example showing the capabilities of current hardware. The above example (essentially a 300Mbit TBF emulator) is pointless on an internal interface to begin with: you will pretty much always want a regular LS service curve there, and in such a scenario HFSC simply doesn't throttle at all.

300Mbit RT service curve (selected columns from mpstat −P ALL 1):

```
10:56:43 PM  CPU  %sys     %irq   %soft   %idle
10:56:44 PM  all  20.10    6.53   34.67   37.19
10:56:44 PM    0  35.00    0.00   63.00    0.00
10:56:44 PM    1   4.95   12.87    6.93   73.27
```

So, in the rare case you need those speeds with only a RT service curve, or with a UL service curve: remember the drawbacks.

CAVEAT: RANDOM ONLINE EXAMPLES

For reasons unknown (though well guessed), many examples you can google love to overuse UL criterion and stuff it in every node possible. This makes no sense and works against what HFSC tries to do (and does pretty damn well). Use UL where it makes sense: on the uppermost node to match upstream router's uplink capacity. Or in special cases, such as testing (limit certain subtree to some speed), or customers that must never get more than certain speed. In the last case you can usually achieve the same by just using a RT criterion without LS+UL on leaf nodes.

As for the router case - remember it's good to differentiate between "traffic to router" (remote console, web config, etc.) and "outgoing traffic", so for example:

```
tc qdisc add dev eth0 root handle 1:0 hfsc default 0x8002
tc class add dev eth0 parent 1:0 classid 1:999 hfsc rt m2 50Mbit
tc class add dev eth0 parent 1:0 classid 1:1 hfsc ls m2 2Mbit ul m2 2Mbit
```

... so "internet" tree under 1:1 and "router itself" as 1:999

LAYER2 ADAPTATION

Please refer to **tc−stab**(8)

SEE ALSO

tc(8), **tc−hfsc**(8), **tc−stab**(8)

Please direct bugreports and patches to: <netdev@vger.kernel.org>

AUTHOR

Manpage created by Michal Soltys (soltys@ziu.info)