NAME

CBQ - Class Based Queueing

SYNOPSIS

tc qdisc ... dev dev **(parent** classid **| root) [handle** major: **] cbq avpkt** bytes **bandwidth** rate **[cell** bytes **] [ewma** log **] [mpu** bytes **]**

tc class ... dev dev **parent** major:[minor] **[classid** major:minor **] cbq allot** bytes **[bandwidth** rate **] [rate** rate **] prio** priority **[weight** weight **] [minburst** packets **] [maxburst** packets **] [ewma** log **] [cell** bytes **] avpkt** bytes **[mpu** bytes **] [bounded isolated] [split** handle **& defmap** defmap **] [estimator** interval timeconstant **]**

DESCRIPTION

Class Based Queueing is a classful qdisc that implements a rich linksharing hierarchy of classes. It contains shaping elements as well as prioritizing capabilities. Shaping is performed using link idle time calculations based on the timing of dequeue events and underlying link bandwidth.

SHAPING ALGORITHM

Shaping is done using link idle time calculations, and actions taken if these calculations deviate from set limits.

When shaping a 10mbit/s connection to 1mbit/s, the link will be idle 90% of the time. If it isn't, it needs to be throttled so that it IS idle 90% of the time.

From the kernel's perspective, this is hard to measure, so CBQ instead derives the idle time from the number of microseconds (in fact, jiffies) that elapse between requests from the device driver for more data. Combined with the knowledge of packet sizes, this is used to approximate how full or empty the link is.

This is rather circumspect and doesn't always arrive at proper results. For example, what is the actual link speed of an interface that is not really able to transmit the full 100mbit/s of data, perhaps because of a badly implemented driver? A PCMCIA network card will also never achieve 100mbit/s because of the way the bus is designed - again, how do we calculate the idle time?

The physical link bandwidth may be ill defined in case of not-quite-real network devices like PPP over Ethernet or PPTP over TCP/IP. The effective bandwidth in that case is probably determined by the efficiency of pipes to userspace - which not defined.

During operations, the effective idletime is measured using an exponential weighted moving average (EWMA), which considers recent packets to be exponentially more important than past ones. The Unix loadaverage is calculated in the same way.

The calculated idle time is subtracted from the EWMA measured one, the resulting number is called 'avgidle'. A perfectly loaded link has an avgidle of zero: packets arrive exactly at the calculated interval.

An overloaded link has a negative avgidle and if it gets too negative, CBQ throttles and is then 'overlimit'.

Conversely, an idle link might amass a huge avgidle, which would then allow infinite bandwidths after a few hours of silence. To prevent this, avgidle is capped at **maxidle.**

If overlimit, in theory, the CBQ could throttle itself for exactly the amount of time that was calculated to pass between packets, and then pass one packet, and throttle again. Due to timer resolution constraints, this may not be feasible, see the **minburst** parameter below.

CLASSIFICATION

Within the one CBQ instance many classes may exist. Each of these classes contains another qdisc, by default tc-pfifo(8).

When enqueueing a packet, CBQ starts at the root and uses various methods to determine which class should receive the data. If a verdict is reached, this process is repeated for the recipient class which might have further means of classifying traffic to its children, if any.

CBQ has the following methods available to classify a packet to any child classes.

(i) **skb->priority class encoding.** Can be set from userspace by an application with the **SO_PRIORITY** setsockopt. The**skb->priority class encoding** only applies if the skb->priority holds a major:minor handle of an existing class within this qdisc.

(ii) tc filters attached to the class.

(iii) The defmap of a class, as set with the **split & defmap** parameters. The defmap may contain instructions for each possible Linux packet priority.

Each class also has a **level.** Leaf nodes, attached to the bottom of the class hierarchy, have a level of 0.

CLASSIFICATION ALGORITHM

Classification is a loop, which terminates when a leaf class is found. At any point the loop may jump to the fallback algorithm.

The loop consists of the following steps:

(i) If the packet is generated locally and has a valid classid encoded within its **skb->priority,** choose it and terminate.

(ii) Consult the tc filters, if any, attached to this child. If these return a class which is not a leaf class, restart loop from the class returned. If it is a leaf, choose it and terminate.

(iii) If the tc filters did not return a class, but did return a classid, try to find a class with that id within this qdisc. Check if the found class is of a lower **level** than the current class. If so, and the returned class is not a leaf node, restart the loop at the found class. If it is a leaf node, terminate. If we found an upward reference to a higher level, enter the fallback algorithm.

(iv) If the tc filters did not return a class, nor a valid reference to one, consider the minor number of the reference to be the priority. Retrieve a class from the defmap of this class for the priority. If this did not contain a class, consult the defmap of this class for the **BEST_EFFORT** class. If this is an upward reference, or no **BEST_EFFORT** class was defined, enter the fallback algorithm. If a valid class was found, and it is not a leaf node, restart the loop at this class. If it is a leaf, choose it and terminate. If neither the priority distilled from the classid, nor the **BEST_EFFORT** priority yielded a class, enter the fallback algorithm.

The fallback algorithm resides outside of the loop and is as follows.

(i) Consult the defmap of the class at which the jump to fallback occurred. If the defmap contains a class for the **priority** of the class (which is related to the TOS field), choose this class and terminate.

(ii) Consult the map for a class for the **BEST_EFFORT** priority. If found, choose it, and terminate.

(iii) Choose the class at which break out to the fallback algorithm occurred. Terminate.

The packet is enqueued to the class which was chosen when either algorithm terminated. It is therefore possible for a packet to be enqueued *not* at a leaf node, but in the middle of the hierarchy.

LINK SHARING ALGORITHM

When dequeuing for sending to the network device, CBQ decides which of its classes will be allowed to send. It does so with a Weighted Round Robin process in which each class with packets gets a chance to send in turn. The WRR process starts by asking the highest priority classes (lowest numerically - highest semantically) for packets, and will continue to do so until they have no more data to offer, in which case the process repeats for lower priorities.

CERTAINTY ENDS HERE, ANK PLEASE HELP

Each class is not allowed to send at length though - they can only dequeue a configurable amount of data during each round.

If a class is about to go overlimit, and it is not **bounded** it will try to borrow avgidle from siblings that are not **isolated.** This process is repeated from the bottom upwards. If a class is unable to borrow enough avgidle to send a packet, it is throttled and not asked for a packet for enough time for the avgidle to increase above zero.

I REALLY NEED HELP FIGURING THIS OUT. REST OF DOCUMENT IS PRETTY CERTAIN AGAIN.

QDISC

The root qdisc of a CBQ class tree has the following parameters:

parent major:minor | root
: This mandatory parameter determines the place of the CBQ instance, either at the **root** of an interface or within an existing class.

handle major:
: Like all other qdiscs, the CBQ can be assigned a handle. Should consist only of a major number, followed by a colon. Optional.

avpkt bytes
: For calculations, the average packet size must be known. It is silently capped at a minimum of 2/3 of the interface MTU. Mandatory.

bandwidth rate
: To determine the idle time, CBQ must know the bandwidth of your underlying physical interface, or parent qdisc. This is a vital parameter, more about it later. Mandatory.

cell
: The cell size determines he granularity of packet transmission time calculations. Has a sensible default.

mpu
: A zero sized packet may still take time to transmit. This value is the lower cap for packet transmission time calculations - packets smaller than this value are still deemed to have this size. Defaults to zero.

ewma log
: When CBQ needs to measure the average idle time, it does so using an Exponentially Weighted Moving Average which smooths out measurements into a moving average. The EWMA LOG determines how much smoothing occurs. Defaults to 5. Lower values imply greater sensitivity. Must be between 0 and 31.

A CBQ qdisc does not shape out of its own accord. It only needs to know certain parameters about the underlying link. Actual shaping is done in classes.

CLASSES

Classes have a host of parameters to configure their operation.

parent major:minor
: Place of this class within the hierarchy. If attached directly to a qdisc and not to another class, minor can be omitted. Mandatory.

classid major:minor
: Like qdiscs, classes can be named. The major number must be equal to the major number of the qdisc to which it belongs. Optional, but needed if this class is going to have children.

weight weight
: When dequeuing to the interface, classes are tried for traffic in a round-robin fashion. Classes with a higher configured qdisc will generally have more traffic to offer during each round, so it makes sense to allow it to dequeue more traffic. All weights under a class are normalized, so only the ratios matter. Defaults to the configured rate, unless the priority of this class is maximal, in which case it is set to 1.

allot bytes

Allot specifies how many bytes a qdisc can dequeue during each round of the process. This parameter is weighted using the renormalized class weight described above.

priority priority

In the round-robin process, classes with the lowest priority field are tried for packets first. Mandatory.

rate rate

Maximum rate this class and all its children combined can send at. Mandatory.

bandwidth rate

This is different from the bandwidth specified when creating a CBQ disc. Only used to determine maxidle and offtime, which are only calculated when specifying maxburst or minburst. Mandatory if specifying maxburst or minburst.

maxburst

This number of packets is used to calculate maxidle so that when avgidle is at maxidle, this number of average packets can be burst before avgidle drops to 0. Set it higher to be more tolerant of bursts. You can't set maxidle directly, only via this parameter.

minburst

As mentioned before, CBQ needs to throttle in case of overlimit. The ideal solution is to do so for exactly the calculated idle time, and pass 1 packet. However, Unix kernels generally have a hard time scheduling events shorter than 10ms, so it is better to throttle for a longer period, and then pass minburst packets in one go, and then sleep minburst times longer.

The time to wait is called the offtime. Higher values of minburst lead to more accurate shaping in the long term, but to bigger bursts at millisecond timescales.

minidle

If avgidle is below 0, we are overlimits and need to wait until avgidle will be big enough to send one packet. To prevent a sudden burst from shutting down the link for a prolonged period of time, avgidle is reset to minidle if it gets too low.

Minidle is specified in negative microseconds, so 10 means that avgidle is capped at -10us.

bounded

Signifies that this class will not borrow bandwidth from its siblings.

isolated

Means that this class will not borrow bandwidth to its siblings

split major:minor & defmap bitmap[/bitmap]

If consulting filters attached to a class did not give a verdict, CBQ can also classify based on the packet's priority. There are 16 priorities available, numbered from 0 to 15.

The defmap specifies which priorities this class wants to receive, specified as a bitmap. The Least Significant Bit corresponds to priority zero. The **split** parameter tells CBQ at which class the decision must be made, which should be a (grand)parent of the class you are adding.

As an example, 'tc class add ... classid 10:1 cbq .. split 10:0 defmap c0' configures class 10:0 to send packets with priorities 6 and 7 to 10:1.

The complimentary configuration would then be: 'tc class add ... classid 10:2 cbq ... split 10:0 defmap 3f' Which would send all packets 0, 1, 2, 3, 4 and 5 to 10:1.

estimator interval timeconstant

CBQ can measure how much bandwidth each class is using, which tc filters can use to classify packets with. In order to determine the bandwidth it uses a very simple estimator that measures once every **interval** microseconds how much traffic has passed. This again is a EWMA, for which the time constant can be specified, also in microseconds. The **time constant** corresponds to the sluggishness of the measurement or, conversely, to the sensitivity of the average to short bursts. Higher values mean less sensitivity.

SOURCES

- Sally Floyd and Van Jacobson, "Link-sharing and Resource Management Models for Packet Networks", IEEE/ACM Transactions on Networking, Vol.3, No.4, 1995
- Sally Floyd, "Notes on CBQ and Guarantee Service", 1995
- Sally Floyd, "Notes on Class-Based Queueing: Setting Parameters", 1996
- Sally Floyd and Michael Speer, "Experimental Results for Class-Based Queueing", 1998, not published.

SEE ALSO

tc(8)

AUTHOR

Alexey N. Kuznetsov, <kuznet@ms2.inr.ac.ru>. This manpage maintained by bert hubert <ahu@ds9a.nl>