Analysis of the simple token bucket filter algorithm implementation inside the netfilter's limit module.

Nicolas Bouliane
nicboul at gmail.com

August 04, 2007

Abstract

The netfilter's limit module provides a way to match at a limited rate using a token bucket filter algorithm. In this paper we analyse the design and implementation of the simple token bucket filter algorithm inside the limit match. We show that the accuracy is deficient, and that the overflow handling is broken. We then provide a more secure, flexible and clean STBF implementation.

Introduction

The limit module provides 2 options: --limit and --limit-burst. These options allow us to specify the average precision of the limit rate we want to match. The limit rate boundaries specified by the module are 10 000/sec and 1/day. The fastest is 10 000 packets per second and the slowlest is 1 packet per day.

The algorithm inconsistency

The limit module specifies that the maximum limit is 10 000 packets per second which is a period of 1/10 000 sec or 0.1ms. We have found that the maximum limit rate is rather, on an i386 architecture, of 250 packets per second which is a period of 1/250 sec or 4.0ms. The algorithm is based on a counter called jiffy. The frequency of the jiffy counter stand in the HZ macro which is arch dependant. This value is 250 by default on the i386. Jiffy counter is incremented every 4.0ms. You can see a strong relationship between the practical limit rate of 250 packets per second and the frequency of the jiffy which is 250 incrementations per second.

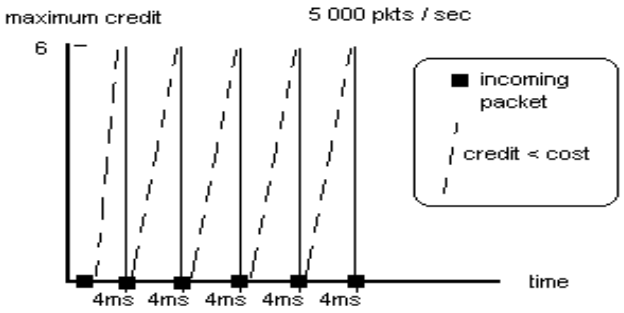
How the algorithm works

The algorithm is based on a concept of credit. We begin with an amount of credits calculated from the values specified with the --limit and --limit-burst. This amount of credits we start with is also the maximum credit we can have. We also calculate a cost that every packet that pass needs to pay. The only way to get new credit is to wait, that means that only time can give us new credit. It use the jiffies counter because it's more efficient than using a real clock on every packet. It's thus impossible to give new credits every time, we must have a checkpoint. This checkpoint is the jiffy counter which is incremented HZ times per second. As stated above, HZ is 250 on the i386 architecture.

Where stand the algorithm inconsistency

We will use a real example, let's say:
`--limit 5000/sec --limit-burst 1`

If we printk() the values from the xt_limit module we see that the maximum credit is 6 and that the cost is 6. A rate of 5000 packets per second is a packet every 0.2ms or 20 packets every 4ms. The checkpoint frequency is 250 times per second (i386), it means that every 4ms we get new credits. Every time we reach the checkpoint we can get a maximum of 6 credits, even if we try to get more credits, our maximum credits capacity is fixed to 6. The problem is that we get enough credit for only 1 packet every 4ms and that we must wait to the next checkpoint to accept new packet. So, every 4ms we get 6 credits which is the cost for only 1 packet. This is equal to 250 packets per second.



This graphic shows that every 4ms we get 6 credits and we then allow 1 packet to pass which cost 6 credits. We then must wait 4ms before allowing a packet to pass through.

Practical test with ping:

```
-A OUTPUT -p icmp -m limit --limit 5000/sec --limit-burst 1 -j ACCEPT
-A OUTPUT -p icmp -j DROP

# ping -c 500 -s 1 -i 0 192.168.0.101
       PING 192.168.0.101 (192.168.0.101) 1(29) bytes of data.
--- 192.168.0.101 ping statistics ---
500 packets transmitted, 250 received, 50% packet loss, time 2998ms
```

We have sent 500 packets in around 3 secs. Even if we allowed 5000 packets per second, only 250 packets have been received.

The quiet overflow

The algorithm calculate the maximum credit and the cost. Because the algorithm plays with big numbers we must handle integer overflow gracefully.

```
/* Check for overflow. */
if (r->burst == 0
    || user2credits(r->avg * r->burst) < user2credits(r->avg)) {
        printk("Overflow in xt_limit, try lower: %u/%u\n",
               r->avg, r->burst);
        return 0;
```

Hopefully we have a mechanism that is supposed to protect us against overflow. Unfortunately we have found that sometimes overflows are not caught.

We will use a real example, let's say `--limit 3/day --limit-burst 5`. We then can see in syslog: `Overflow in xt_limit, try lower: 288000000/5`. The overflow has been caught. But surprisingly with `--limit 3/day --limit-burst 6`, we get absolutely nothing and the rule get inserted.

It's time to get our hands in the code, and to do some mathematics.

Userspace:

```
*val = XT_LIMIT_SCALE * mult / r;
```

Kernelspace:

```
static u_int32_t user2credits(u_int32_t user)
{
        /* If multiplying would overflow... */
        if (user > 0xFFFFFFFF / (HZ*CREDITS_PER_JIFFY))
                /* Divide first. */
                return (user / XT_LIMIT_SCALE) * HZ * CREDITS_PER_JIFFY;

        return (user * HZ * CREDITS_PER_JIFFY) / XT_LIMIT_SCALE;
}

       ...

r->credit_cap = user2credits(r->avg * r->burst); /* Credits full. */
r->cost = user2credits(r->avg);
```

The calculation of --limit 3/day --limit-burst 5:

```
*val = XT_LIMIT_SCALE * mult / r;
10 000 * 24*60*60 / 3 = 288 000 000

r->credit_cap = user2credits(r->avg * r->burst);
288 000 000 * 5 = 1 440 000 000

if (user > 0xFFFFFFFF / (HZ*CREDITS_PER_JIFFY))
1 440 000 000 > 134 218

return (user / XT_LIMIT_SCALE) * HZ * CREDITS_PER_JIFFY;
1 440 000 000 / 10 000 * 250 * 128
= 4 608 000 000 (overflow) [return 313 032 705]
```

So, `user2credits(r->avg * r->burst)` returns 313 032 705. The overflow handling mechanism state that if `user2credits(r->avg * r->burst)` is lower than `user2credits(r->avg)`, we have caught an overflow. We need to calculate `user2credits(r->avg)` now.

```
r->credit_cap = user2credits(r->avg);
288 000 000

if (user > 0xFFFFFFFF / (HZ*CREDITS_PER_JIFFY))
288 000 000 > 134 218

return (user / XT_LIMIT_SCALE) * HZ * CREDITS_PER_JIFFY;
288 000 000 / 10 000 * 250 * 128
= 921 600 000 [return  921 600 000]
```

Now we know that `user2credits(r->avg * r->burst)` returns 313 032 705 and that `user2credits(r->avg)` return 921 600 000.

Because the statement 313 032 705 < 921 600 000 is true, the overflow is caught.

We will show you where stands the hidden overflow through the calculation of --limit 3/day --limit-burst 6.

```
*val = XT_LIMIT_SCALE * mult / r;
10 000 * 24*60*60 / 3 = 288 000 000

r->credit_cap = user2credits(r->avg * r->burst);
288 000 000 * 6 = 1 728 000 000

if (user > 0xFFFFFFFF / (HZ*CREDITS_PER_JIFFY))
728 000 000 > 134 218

return (user / XT_LIMIT_SCALE) * HZ * CREDITS_PER_JIFFY;
1 728 000 000 / 10 000 * 250 * 128
= 5 529 600 000 (overflow) [return 1 234 632 705]

r->credit_cap = user2credits(r->avg);
288 000 000
```

```
if (user > 0xFFFFFFFF / (HZ*CREDITS_PER_JIFFY))
288 000 000 > 134 218

return (user / XT_LIMIT_SCALE) * HZ * CREDITS_PER_JIFFY;
288 000 000 / 10 000 * 250 * 128
= 921 600 000 [return  921 600 000]
```

Now we know that `user2credits(r->avg * r->burst)` returns 1 234 632 705 and that `user2credits(r->avg)` returns 921 600 000.

Because the statement 1 234 632 705 < 921 600 000 is false, the overflow has not been caught. You can extrapolate and see all the pairs of --limit and --limit-burst that would overflow without being caught.

Analysis conclusion

The limit module is fragile from an implementation perspective and flawed from a mathematical perspective. The limit rate scalability cannot achieve the expectation due to a misunderstanding of the implication of the jiffy counter frequency. The overflow handling mechanism fails to catch all possible overflows, possibly because of its subpar design and testing. Furthermore, the implementation use a rather obscure way to calculate the maximum credit and cost which is prone to bugs.

What we propose

Our goal was to propose an implementation aware of the jiffy frequency, mathematically easy to understand and with clear boundaries.

The mathematics

credit	The number of credit we have
credit_max	The maximum credit we can have
cost	The passing cost for one packet
cpj	The number of credits per jiffy

L	The number of packet per period
t	The period (sec, min, hour, day)
HZ	The jiffy counter frequency
B	The limit burst

L / t > HZ

```
credit =  (L / (t * HZ)) * B
credit_max = (L / (t * HZ)) * B
cost = 1
cpj = L / HZ
```

L / t < HZ

```
credit =  ((t * HZ) / L) * B
credit_max = ((t * HZ) / L) * B
cost = (t * HZ) / L
cpj = 1
```

Real Life examples

We want match 1000 packets per second, without limit burst.

```
1000 / 1 > 250
       credit        = (1000 / (1 * 250)) * 1
       credit_max    = (1000 / (1 * 250)) * 1
       cost          = 1
       cpj           = 1000 / 250
```

We start with 4 credits and it's the maximum credit we can ever have. The cost per packet is 1 and the number of credits we receive per jiffy is 4. Every time we hit the checkpoint, every 4ms with HZ set to 250, we get 4 new credits. A rate of 1000 packets per second means that between each checkpoint we can let 4 packets pass.

We want match 600 packets per minute, without limit burst.

```
600 / 60 < 250
       credit        = ((60 * 250) / 600) * 1
       credit_max    = ((60 * 250) / 600) * 1
       cost          = ((60 * 250) / 600)
       cpj           = 1
```

We start with 25 credits and it's the maximum credit we can ever have. The cost per packet is 25 and the number of credit we receive per jiffy is 1. Every time we hit the checkpoint, every 4ms with HZ set to 250, we get 1 new credit. A rate of 600 packets per minute means that we let pass 1 packet every 25th checkpoints.

The boundaries

Defining the boundaries is not trival to do. In our formula we have four variables we must take in consideration: L, t, B and HZ.

```
L / ( t * HZ)  * B  <  MAX_INT
( t * HZ ) / L * B  <  MAX_INT
```

We want also be able to inform the user what the boundaries are, which is tricky because the HZ is only visible from the kernelspace. We know that for almost every architecture HZ is set to 250. The only exception is maybe the Alpha with an HZ set to 1000. The period can be 1(sec), 60(min), 60*60(hour) or 24*60*60(day). We could try to do some mathematical calculation to find the exact boundaries for every values of L and B with the different possible values of HZ and t, is it worth it? We have decided to set the boundaries so it can reflect realistically useful values. This way, it's easier to test overflows, to inform the user what the boundaries are and to keep the code clean.

The upper bound is set to :

```
L * B <= 100 000
```

The product of L and B must be lower or equal to 100 000. The throughput of a gbit ethernet is roughly 100 000 packets per second. Here are some valid limit/limit-burst pair :

```
--limit 100000/sec --limit-burst 1
--limit 1/sec --limit-burst 100000
--limit 316/sec --limit-burst 316
```

Conclusion

If you found something weird, obscure, wrong, stupid, etc. Please contact me.

Discussion

The implementation we proprose could be improved from a granularity perspective with scaled math.