IPv6 route lookup performance and scaling

Michal Kubeček
SUSE Labs
mkubecek@suse.cz

IPv6 parity

"It works with IPv4 so it should also work with IPv6"

IPv6 parity

"It works with IPv4 so it should also work with IPv6"

Mostly true (today) but there are still exceptions.

In particular, when it comes to performance, we can't (in general) expect IPv6 to be as efficient as IPv4.

IPv6 parity

"It works with IPv4 so it should also work with IPv6"

Mostly true (today) but there are still exceptions.

In particular, when it comes to performance, we can't (in general) expect IPv6 to be as efficient as IPv4.

But does it perform as well as it could?

Customer experience I.

November 2012: bug „Contention in softirq on systems with high number of cores“

- host with many CPU's routing intensive IPv6 traffic for 200000 hosts
- heavy contention on IPv6 FIB garbage collector lock
- triggering soft lockup detector
- similar amount of IPv4 traffic handled fine

Customer experience I.

November 2012: bug „Contention in softirq on systems with high number of cores“

- host with many CPU's routing intensive IPv6 traffic for 200000 hosts
- heavy contention on IPv6 FIB garbage collector lock,
- triggering soft lockup detector
- similar amount of IPv4 traffic handled fine

- Resolved by raising GC threshold and limit.
- Everyone happy until...

Customer experience II.

January 2013: bug „Bond flapping between interfaces under high IPv6 load“

- same host and workload as in the previous bug
- bond flapping caused by dropped packets (ARP monitor)
- NIC dropping packets, network stack processing too slow
- reason: routing lookup too slow
- reason: IPv6 FIB too large after raising GC threshold/limit

Customer experience II.

January 2013: bug „Bond flapping between interfaces under high IPv6 load“

- same host and workload as in the previous bug
- bond flapping caused by dropped packets (ARP monitor)
- NIC dropping packets, network stack processing too slow
- reason: routing lookup too slow
- reason: IPv6 FIB too large after raising GC threshold/limit

- low limits: FIB6 GC lock contention
- high limits: dropped packets, bond flapping
- no value prevents both problems

Solution

Slight change in the algorithm deciding if GC should run: if not forced and `spin_trylock_bh()` fails, it means someone is already running GC so that it doesn't make much sense for it to finish only to run it again immediatelly.

Avoids heavy GC contention with reasonable values of threshold and limit.

Customer satisfied, bug closed

Solution???

Slight change in the algorithm deciding if GC should run: if not forced and `spin_trylock_bh()` fails, it means someone is already running GC so that it doesn't make much sense for it to finish only to run it again immediatelly.

Avoids heavy GC contention with reasonable values of threshold and limit.

Customer satisfied, bug closed

But we didn't in fact resolve the real problem: IPv6 lookup performance does not scale as well as in the IPv4 case.

Solution???

Slight change in the algorithm deciding if GC should run: if not forced and `spin_trylock_bh()` fails, it means someone is already running GC so that it doesn't make much sense for it to finish only to run it again immediatelly.

Avoids heavy GC contention with reasonable values of threshold and limit.

Customer satisfied, bug closed

But we didn't in fact resolve the real problem: IPv6 lookup performance does not scale as well as in the IPv4 case.

Another customer claims to observe similar problems on SLE11-SP2 kernel containing this fix (unconfirmed).

Customer experience III.

February 2016: bug „IPv6 garbage collection lock contention“

- the same customer (and project) again
- added LXC containers to the picture
- per-netns routing tables and FIB trees
- per-netns garbage collector
- but protected by one shared lock
- handling of „walkers“ also uses shared data and locking
- a patch being tested by customer, going to submit next week (if it helps)

Routing lookup benchmarking

Microbenchmarking the routing lookup is not as easy as it seems. Lookups from userspace (using netlink) would end up benchmarking the netlink message composition and parsing rather than the lookup itself.

To minimize overhead, we need to call the lookup from kernel code.

Implementation:

- a kernel module running the benchmark on loading
- starts given number of threads on different CPU's
- each thread runs given number of lookups for pseudorandom addresses
- repeated given number of times
- either IPv4 or IPv6 lookups can be benchmarked
- results collected through a file in `/proc`

Benchmarking: details

All tests performed on `albali.suse.cz` (two 12-core Intel E5-2697 CPU, 24 CPU, 2.7 GHz)

- tested with SLE12 kernel, 4.2 and 4.4
- results presented here are with 4.4.1
- tests performed in runlevel 1 (to minimize other effects)
- routing table pre-filled with 1–100000 random routes
- 1000000 lookups per thread, 12 iterations
- values collected, procesed using awk script
- highest and lowest value discarded
- average and mean deviation calculated from the rest
- subtracted results from a dry run where no lookup is actually performed

Scaling: table size

All values are average times per lookup in nanoseconds.

routes	1	10	100	1000	10000	100000
IPv4	24.4	27.1	31.7	64.0	89.7	190.0
IPv6	1256.7	1096.2	1130.7	1158.3	1265.2	1402.1

- lookups gets slower with table size
- increase rate is reasonable
- absolute increase similar for IPv4 and IPv6

Scaling: number of threads

threads	1	2	4	8	16	24
IPv4 (1 route)	24.4	26.3	24.3	24.1	26.8	29.0
IPv4 (100000 routes)	199.0	191.2	188.4	186.1	184.6	183.8
IPv6 (1 route)	1256.7	1552.6	1959.6	4268.0	13353.3	23030.9
IPv6 (100000 routes)	1402.1	2102.7	2370.0	4344.9	13576.3	23185.2

- IPv4 scales nicely, no visible concurrency
- IPv6 lookup times grow fast with number of threads

Let's try a different view...

Scaling: number of threads

This time, the values are calculated as the elapsed time for the test divided by the total number of lookups executed (i.e. inverse of lookups per second).

threads	1	2	4	8	16	24
IPv6 (1 route)	1256.8	776.3	491.6	535.8	835.4	960.3
IPv6 (100000 routes)	1402.1	1053.2	595.3	546.1	849.4	966.9

Conclusion: IPv6 lookups scale really poorly for higher number of threads. In some tests, the results were even worse than if the lookups were serialized.

Why is that?

IPv6 FIB:

- an ordered tree
- per-table read-write spinlock
- sophisticated logic for “walkers” (minimize critical section)
- `RTF_CACHE` entries to cache lookups
- garbage collector

Why is that?

IPv6 FIB:

- an ordered tree
- per-table read-write spinlock
- sophisticated logic for "walkers" (minimize critical section)
- `RTF_CACHE` entries to cache lookups (much less since 4.2)
- garbage collector

IPv4 FIB:

- a LPC trie (prefix tree)
- RCU based locking
- no cache entries, separate exception table
- no garbage collector

More fair comparison

The tests presented so far were in fact unfair to IPv6 FIB:

- IPv4 tests call `fib_lookup()`
- IPv6 tests call `ip6_route_output()`

Due to design differences, it's difficult to find a function which would be similar to `fib_lookup()`.

Let's do it the other way around: use `ip_route_output_key()` for IPv4.

More fair comparison

Average lookup duration:

threads	1	2	4	8	16	24
IPv4 (1 route)	92.7	150.4	228.6	418.3	1166.1	1978.9
IPv4 (100000 routes)	251.5	265.4	266.1	269.3	257.9	260.6

Elapsed time divided by total number of lookups:

threads	1	2	4	8	16	24
IPv4 (1 route)	92.7	75.5	58.0	54.7	80.6	86.7
IPv4 (100000 routes)	251.4	137.6	69.0	35.8	17.6	11.8

More fair comparison

While this comparison looks a bit better for IPv6, IPv4 still performs much better:

- IPv4 scales nicely (w.r.t. number of threads) for big routing tables
- IPv6 scales badly even for 100000 routes

- these tests didn't trigger garbage collector; if they did, results could be much worse
- any other changes to routing tables can harm FIB6 performance
- we are writing cache entries into the same data structure (not so many of them but still some)

Weak points of the analysis

- the tests are unrealistic, in real life, the level of concurrency is much lower
- with completely different design, it's hard to find comparable functions to call (but still low-level enough)
- large routing tables were generated randomly; not sure how well they represent larget tables from real life routers
- the performance can be affected by many other factors

However, I believe the problem exists and the results indicate IPv6 FIB design is not suitable for high traffic on systems with many CPUs.

What can be done?

An obvious idea: replace current FIB6 implementation with RCU based LPC trie similar to one used for IPv4.

One cannot expect to fully match FIB4 performance but we could get quite close and improve the concurrency greatly.

What can be done?

An obvious idea: replace current FIB6 implementation with RCU based LPC trie similar to one used for IPv4.

One cannot expect to fully match FIB4 performance but we could get quite close and improve the concurrency greatly.

The devil's in the details. There are some differences between IPv4 and IPv6 that need to be addressed to avoid functional regressions. But the reward is really promising.