Introduction

Making a given computer network support more traffic than before by replacing the routing algorithm is an open challenge from the networks area.

The article "On low-latency-capable topologies, and their impact on the design of intra-domain routing" from SIGCOMM 2018 presents a metric that indicates how much room to scale the routing algorithm will have and an algorithm that takes advantage of such space.

On this video, the latter was reimplemented only from the words in the article and tested against OSPF on some topologies.

Their Algorithm

The algorithm avoids overload and then, as a tie-breaker, chooses the lowest latency.

That feeds a loop that simulates the network and decides if the overall state will be better than before.

As the original artifacts used on the article weren't located, this is a blind reimplementation which only uses what's written in the article as source.

Initial challenges

Some constants weren't given and their descriptions implies they're topology-dependant. Due to lack of better explanation, trial-and-error optimization was attempted, but certainly there's room for improvement here.

In places where the exact algorithm wasn't specified, it was granted the freedom to use a trivial solution rather than an optimized one.

Our implementation's shortcuts

- ARP and IPv4 only.
- The controller gets the file which describes the topology as an commandline argument.
- The topology is static.
- Trivial implementations rather than optimized ones:
- No impact, as all flow rules are added prior to traffic.
- ARP will only use OSPF:
- No big deal, as such traffic can be neglected.

Performance tests

All performance tests were automated and ran on an i7-4790 (4x4GHz + HyperThreading) with enough free RAM (DDR3@1600MHz).

Let's say we have 9 hosts...
[h1, h2, h3, h4, h5, h6, h7, h8, h9]
...and we want to test one PING and some IPERFs.

Our solution was breaking that list in half...
[h1, h2, h3, h4, h5]
[h6, h7, h8, h9]
...discarding the middle element, if necessary to make the sizes the same...
[h1, h2, h3, h4]
[h6, h7, h8, h9]
...reversing the last half...
[h1, h2, h3, h4]
[h9, h8, h7, h9]
...making them one list of pairs...
[(h1, h9), (h2, h8), (h3, h7), (h4, h9)]
...and then saving the first pair for PING and the rest for IPERFs.
PING → (h1, h9)
IPERF → [(h2, h8), (h3, h7), (h4, h9)]

With the tests that will be ran defined, we do them all sequentially and in parallel.

Topologies

- A strange bow tie (named bigtopo).
- A 3-stage CLOS (named CLOS)

-
  - whose switches are equivalent to the bipartite graph k3,2
- A 5-stage CLOS.
- A biparpartite graph k5,2,1
- A triangular topology (named principle)
-
  - The longer path is also slower, but becomes an interesting alternative when the main path becomes congested.

Performance results

While running the algorithms, it was observed that the algorithm switches paths very often, sometimes enabling and disabling a path. It's worth mentioning that both the parallel bandwitdh and latency tests were run simultaneously.

On **sequential bandwidth tests**, the implemented algorithm didn't clearly display a major advantage compared to OSPF. It can be obseved on the table below that the difference is small (if not neglectable).

sequential_iperf	bigtopo	bipartite	clos	clos5	principle
ospf	233.6 kbps	24210.5 kbps	1158.0 kbps	1150.0 kbps	1150.0 kbps
lowlat	244.0 kbps	24207.2 kbps	1162.0 kbps	1150.0 kbps	1020.0 kbps

On **parallel bandwidth tests**, the same thing. While running this test, it was observed that the algorithm switched back and foward between two or more paths, leaving at times, a path completely unused, which indicates that something is wrong. When adjusting the M_1 and M_2 constants, such behaviour modified, giving a clue that they needed a better tuning.

parallel_iperf	bigtopo	bipartite	clos	clos5	principle
ospf	183.1 kbps	24197.8 kbps	1126.0 kbps	151.4 kbps	1170.0 kbps
lowlat	186.1 kbps	24179.0 kbps	1126.0 kbps	150.1 kbps	1000.0 kbps

On **sequential latency tests**, it can also be observed that the difference isn't that noticeable.

sequential_ping	bigtopo	bipartite	clos	clos5	principle
ospf	0.3690	0.1220	0.1100	0.1740	0.1090
lowlat	0.2620	0.1700	0.1100	0.1420	0.1330

On **parallel latency tests**, where this algorithm should have shown a great difference, the difference is very subtle.

parallel_ping	bigtopo	bipartite	clos	clos5	principle
ospf	7.4440	902.5610	1.6980	621.8470	79.3170
lowlat	4.3960	780.7750	0.7610	615.6790	65.0350

One-time results

Sometimes, things doesn't work in a consistent manner. Not all results we got were the way they’re on the tables above:

- While manually testing `principle` topology on early development stages, we got a latency around 600ms on OSPF and 6ms on their algorithm. We were unable to reproduce this ever again and even suspect this was the result of some sort of mistake.
- There was a quirck on `bigtopo`’s latency on sequential tests under the row `lowlat` where the value was 145.257ms, which makes no sense why after a 4s cooldown after the last bandwidth test it would be a sudden latency. Such test was re-run and the tables were generated once again.

Conclusion

The article claimed some ground-breaking results, which we were unable to reproduce. We found some undocumented odd behaviours with our reimplementation. It might be too early to say their methodology doesn't work, but it surely needs a deeper clarification on what M_1 and M_2 constants really does, how to find a good value, and more semantically-precise flow diagrams that would make it harder to reimplement things wrong.