Load Balancing with nftables II

Laura García

Zen Load Balancer
October 2016 - Tokyo, Japan
lauragl@sofintel.net

Abstract

The load balancing with netfilter framework presented in the last Netdev 1.1 allowed to design a use case based prototype with nftables in order to create the infrastructure required to build a complete load balancer. In this document we present the advances of the developments that will allow to build a complete load balancer with nftables and more performance compared to lvs.

In this document we present the developments done to achieve those requirements, the review of some use cases to show the definitive syntax, benchmarks and the work to do to continue improving the performance obtained.

Keywords

Load Balancing, nftables, netfilter, lvs, ipvs, dnat, snat, dsr, direct routing, direct server return, scalability, benchmark, performance.

Introduction

This paper presents the developments done to build a complete load balancer with nftables infrastructure (kernel side, libnftnl and nftables user space tool), then the review of some use cases with the definitive syntax, benchmarks of lvs and nftables use cases and finally, the next steps to progress this work.

Development Evolution

In order to provide packet scheduling in the nft infrastructure, we've included two new expressions number generation (nft_numgen) and hash (nft_hash).

Numgen

nft_numgen is based on the xt_statistiscs extension (from iptables) and provides the ability to scale the values generated with two different modes or operations available: incremental and random.
The numgen expression uses mod as a modulus and offset as optional parameter to be added to the value generated.

The **incremental** operation (inc) is a connection counter that act as a round robin scheduler. Example:

```
ip daddr <vip> tcp dport <vport> dnat to numgen inc mod 2 \
   map { 0 : <ipaddr0>, 1 : <ipaddr1> }
```

This expression also permits to generate series of incremental numbers with an offset like:

```
meta mark set numgen inc mod 3 offset 100
```

In the example above, the connections are marked following the series: 100, 101, 102, 100, …

The **random** operation (random) generates a random number which acts as a weight scheduler. Example:

```
ip daddr <vip> tcp dport <vport> dnat to numgen random \
  mod 2 map { 0 : <ipaddr0>, 1 : <ipaddr1> }
```

This expression also permits to generate series of random numbers with an offset like:

```
meta mark set numgen random mod 3 offset 100
```

In the example above, the connections are marked following a series of numbers between 100 and 102.

Hash

nft_hash generates a hash for any selector concatenation and currently, only one mode is available: Jenkins hash. Within a load balancing scheme, this expression will be used to create persistent connections. Example:

```
ip daddr <vip> tcp dport <vport> dnat to jhash \
  ip saddr mod 2 map { 0: <ipaddr0>, 1: <ipaddr1> }
```

The example above performs a nat based on the source ip address, and all connections from such ip address will be assigned to a certain destination address in the map.

Another example could be the packet marking based on the hash from the source ip address.

```
meta mark set jhash ip saddr mod 3 seed 0xabcd \
  offset 100
```

The marks in this case generates hash numbers between 100 and 102.

Requirements

The requirements in order to use the numgen and hash extensions are the following:

- kernel version >= 4.8.0-rc4+ (nf-next branch at the time of writing this paper)
- libnftnl > 1.0.6
- nftables >= 0.7 (not yet released at the time of writing this paper)

Use Cases Review

In this section we present the definitive syntax to be used in nftables for every use case studied.

sNAT Topology

The sNAT topology requires 4 steps to complete a flow, where the client connects to a certain virtual IP and port and then, the load balancer changes the source and destination ip addresses to the scheduled backend.
The backend then returns the response to the load balancer and the load balancer to the client masquerading the connection.

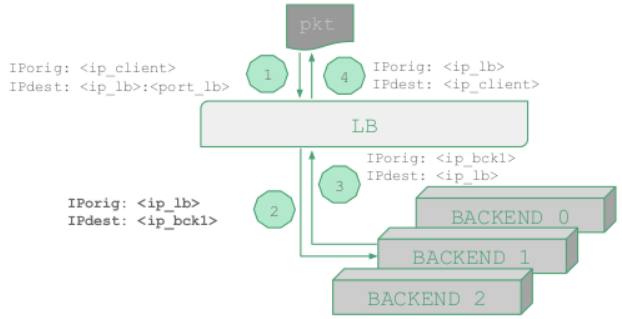

Figure 1. Load balancing with sNAT Topology.

The nft syntax to be able to behave as sNAT load balancer is the following:

```
table ip nat {
 chain prerouting {
  type nat hook prerouting priority 0; policy accept;
  ip daddr <ip_lb> tcp dport <port_lb> dnat to \
   numgen inc mod 3 map { \
    0 : <ip_bck0>, \
    1 : <ip_bck1>, \
    2 : <ip_bck2> }
  }
 chain postrouting {
  type nat hook postrouting priority 100; policy accept;
  masquerade
 }
}
```

The packets in prerouting stage going to a certain virtual ip and port will be natted using a round robin scheduler between the three available backends. Finally, the postrouting chain needs to perform a masquerade in order to hide the backends ip addresses to the client.

dNAT Topology

In the case of dNAT topology, 4 step packet flow will take in place. The client access to the virtual ip and port, and the output packet from the load balancer changes the destination address so the backend is able to see the real ip address from the client.

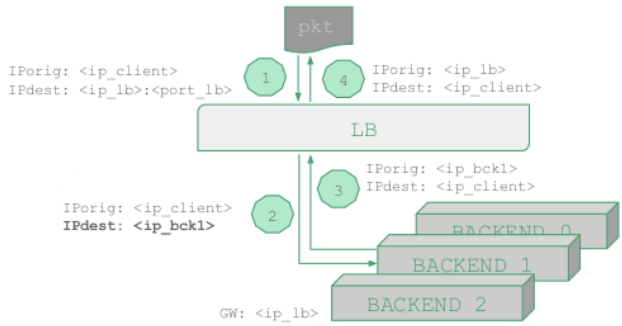

Figure 2. Load balancing with dNAT Topology.

The nft syntax to be able to behave as dNAT load balancer is the following:

```
table ip nat {
 chain prerouting {
  type nat hook prerouting priority 0; policy accept;
  ip daddr <ip_lb> tcp dport <port_lb> dnat to \
   numgen random mod 3 map { \
    0 : <ip_bck0>, \
    1 : <ip_bck1>, \
    2 : <ip_bck2> }
 }
 chain postrouting {
  type nat hook postrouting priority 100; policy accept;
 }
}
```

The dNAT case implementation in nftables is quite similar than sNAT, only the masquerade should be discarded from the postrouting chain. Create the chain postrouting
In the example above, the numgen expression will use a random operation to create a weighted scheduler among the backend ip addresses.

DSR Topology (non connection oriented)

The DSR topology requires 3 steps to complete a flow, where the client connects to a certain virtual IP and port and then, the load balancer changes the source and destination MAC addresses to the scheduled backend.
The backend then returns the response to the client directly, so this approach permits only the incoming packets to pass through the load balancer at L3 level.

In this case, as the flow is not connection oriented, the load balancer doesn't need to gather connections knowledge and only it take cares about packets.

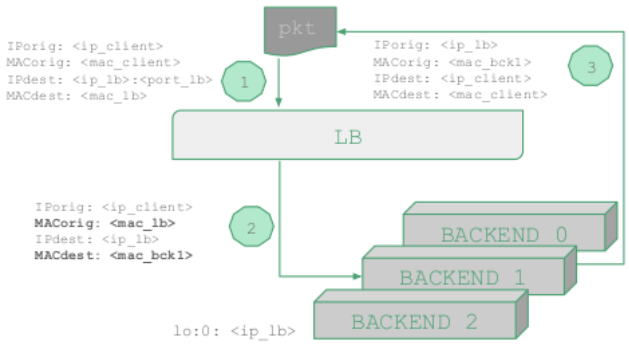

Figure 3. Load balancing with DSR Topology.

The nft syntax to be able to behave as DSR non-connection oriented load balancer is the following:

```
table netdev filter {
 chain ingress {
  type filter hook ingress device <if_lb> \
   priority 0; policy accept;
  ip daddr <ip_lb> udp dport <port_lb> \
   ether saddr set <mac_lb> \
   ether daddr set numgen inc mod 3 \
   map { \
    0: <mac_bck0>, \
    1: <mac_bck1>, \
    2: <mac_bck2> } \
   fwd to <if_lb>
 }
}
```

We can achieve this architecture from ingress with just one rule in order to set the source and destination MAC addresses for every packet with a round robin scheduling method to distribute the packets traffic and then, send to the device again.

DSR Topology (connection oriented)

For DSR topology with connection oriented approach needs the behavior presented in the section above but requires to add additional knowledge to maintain every flow in the assigned backend. There is no conntrack information that we can use from ingress.
Then, for this approach we propose to take advantage of the hash expression in order to generate a kind of persistence with the concatenation of source IP address and source port.

The nft syntax to be able to behave as DSR connection oriented load balancer is the following:

```
table netdev filter {
 chain ingress {
  type filter hook ingress device <if_lb> \
   priority 0; policy accept;
  ip daddr <ip_lb> tcp dport <port_lb> \
   ether saddr set <mac_lb> \
   ether daddr set \
   jhash ip saddr . tcp sport mod 3 seed 0xabcd \
   map { \
    0: <mac_bck0>, \
    1: <mac_bck1>, \
    2: <mac_bck2> } \
   fwd to <if_lb>
 }
}
```

Same behavior than the non-oriented approach but it's proposed to use a hashing function instead of using incremental numgen for two reasons: create a traffic distribution and maintain a persistence for every flow.

Benchmarks

It's presented some benchmarks to know the state of the load balancing with nftables approach and to compare the performance with LVS that is being used as a reference.

Lab Environment

The lab environment used for these benchmarks are the following:

1. Hardware:
   - 2 clients, 3 backends & 1 LB
   - 2 cores (3.33 GHz each) i5 660 with 2 threads/core, 4GB RAM @1333 MHz
   - 2 Intel Gigabit Network 82578DM & 82574L per machine
2. Software:
   - Kernel version 4.8.0-rc4+ branch nf-next
   - System tuning considerations from József paper
   - HTTP protocol transferring 229 bytes per connection (client wrk/server nginx)
3. Considerations:
   - Both IPv4 & IPv6
   - LB was never saturated during a test of 30 seconds
   - LVS performance used as a reference

IPv4 Benchmarks

During the IPv4 benchmarking we obtained similar performance between LVS-SNAT than NFT-SNAT, but NFT-DNAT performs better as the masquerade is not required.

method	req/sec	%cpu	
LVS-SNAT	313427.91	24.11	
NFT-SNAT	289035.54	23.2	
NFT-DNAT	303356.59	23.12	
LVS-DSR	356212.05	4.78	
NFT-DSR	393672.35	0.54	+9.78x

Figure 4. IPv4 benchmark results.

But, what is quite significant is the improvement of the NFT-DSR compared to LVS-DSR, where we get almost 10 times faster with the nftables approach.

In the graph below is shown the performance for every topology of nft compared to lvs regarding the per cent of CPU consumed and the number of flows per second.

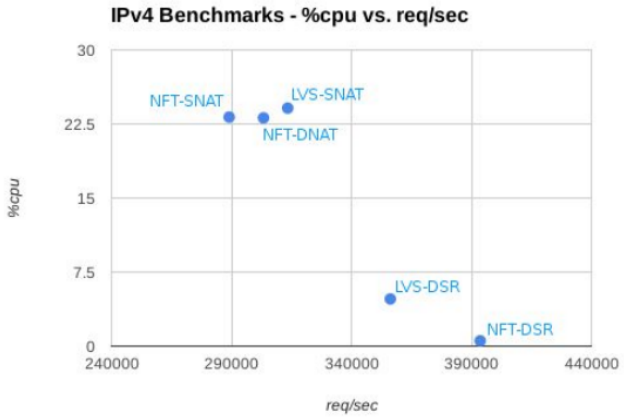

Figure 5. IPv4 performance graph comparing LVS and NFT.

IPv6 Benchmarks

The IPv6 performance obtained with the same use cases than the previous section are better in general with both lvs and nft approaches but similar if we compared the results between them.

method	req/sec	%cpu	
LVS-DNAT	289320.17	24.65	
NFT-SNAT	271790.98	24.85	
NFT-DNAT	287978.41	23.37	
LVS-DSR	418067.65	4.43	
NFT-DSR	432399.38	0.8	+5.72x

Figure 6. IPv6 benchmark results.

In regards to the performance of NFT-DSR compared to LVS-DSR is about almost 6 times faster.

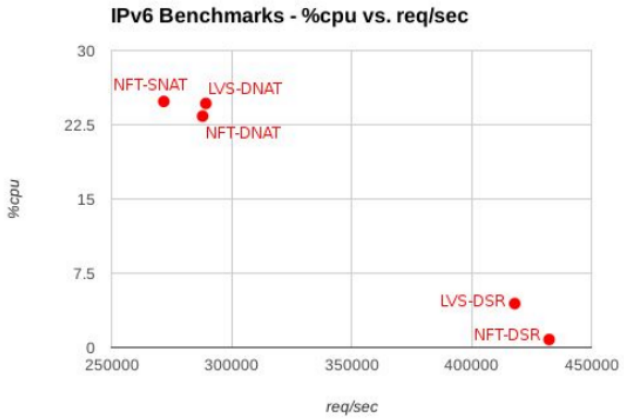

Figure 7. IPv6 performance graph comparing LVS and NFT.

The performance obtained with nft and lvs natted are similar, but nft from ingress makes the big difference.

Work To Do

In order to progress with the work of providing high performance load balancing capabilities in the nftables infrastructure, some work to be done are:

- The implementation of a lightweight NAT from the hook ingress to improve the NAT results.
- User space rule manager to compile more complex schedulers than round robin and weight, and manage different topologies easily.
- Health checks monitor with layered support and allowing internal and external monitors, so the load balancer doesn't need to behave as a monitor.

Acknowledgements

From the Zen Load Balancer Team, we would like to thank Pablo Neira for his mentoring during this development and his continuous support. We've to thank as well to the Outreachy Program for supporting this development and Cumulus Network for their invitation to the Netdev 1.2.

Bibliography

1. Nftables wiki, http://http://wiki.nftables.org

Author Biography

Laura García studied Computer Science in the University of Seville and she has been a Software Engineer for HP and Schneider Electric, with more than 10 years of experience with embedded Linux systems. Currently, she is CEO and co-founder of Sofintel IT Engineering SL company in order to continue the development and evolution of the open source project Zen Load Balancer.