Load Balancing with nftables

Laura García

Zen Load Balancer
Seville, Spain
lauragl@sofintel.net

Abstract

The motivation to design a load balancer prototype with nftables is to provide a flexible network management system with complete load balancing capabilities for Linux-based systems, but also improve Layer 4 load balancing performance using the nftables infrastructure. The iptables approach in this topic lacks features for a complete and high performance load balancing system and those shortcomings have been taken into account in order to be solved in nftables.

Keywords

Load Balancing, nftables, conntrack, netfilter, Linux networking, iptables, LVS.

Introduction

LVS allows very easy deployment of Linux-based load balancers. Probably less well-known is the fact that you can also use iptables rules using the existing matches and targets to implement many of the core load balancing features such as different scheduling approaches and dispatching methods, flow persistence, etc.

This paper discusses the implementation of a Linux-based load balancers using iptables, we will describe our ruleset configurations, lessons learned from integration issues with Netfilter and other networking software and existing limitations. Moreover, we have planed a prototype based on nftables, detailing what is missing and what we consider good to have to improve its load balancing capabilities.

Load Balancing Solutions

The most popular and extended solution currently available for load balancing at layer 4 is Linux Virtual Server. Less well known solution but provides a very good results is to perform load balancing using iptables extensions. Finally, such iptables approach and knowledge gathered will be used to present a design of a high performance load balancing prototype with nftables.

LVS

LVS is a wide used load balancer at layer 4 which provides a full set of complete and versatile schedulers, several forwarding methods like Direct Routing, tunneling and sNAT, and additionally some basic integrated health checks. LVS provides an additional layer on top of netfilter and it's mostly kernel code base with an user space daemon for control. In some cases, it's needed to use iptables to mark packets and the support of content parsing is performed using additional modules than iptables.

iptables

Load balancing with iptables implies the use of the xtables extensions in order to build a set of rules that behave with the desired scheduler and forwarding method. The available forwarding methods includes sNAT and dNAT, according to the transparency required in a certain infrastructure.

The mechanism used in this case requires marking the packets and then forwarding to the determined backend or real server.

The backend health checks needs to be performed from user space as a daemon at different layers (icmp, protocol or application checks, etc.), as it's show in the *Figure 1*.

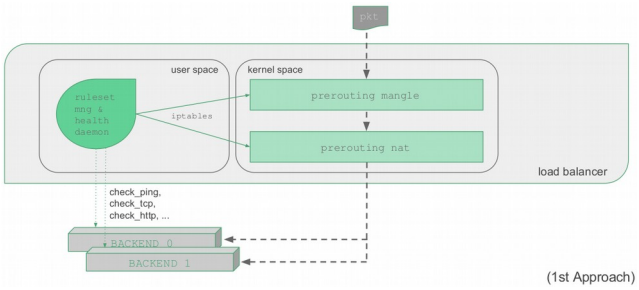

Figure 1. Load balancing with iptables approach.

With this approach, all the complexity of ruleset management and health checks are moved to user space and in the kernel only remains the packet handling, obtaining very good results regarding the performance required. But some concerns could raise into a problem such as:

- The ruleset is handled sequentially which could produce not desired side effects during the packet handling.
- The marking and forwarding rules must be synchronized in order to behave as expected.

nftables

For all these reasons, and following the approach described, we propose to supply the features and

improvements needed within nftables in order to provide a full featured load balancing tool.
Load balancing with nftables is possible through the nftables infrastructure: nft libraries, nftables virtual machine and it's instructions. Inherent properties like dynamic ruleset and atomicity through nft scripting are major keys regarding the reliability in this design and avoid side effects concerns that raised the approach with iptables.
Another enhancement in terms of performance and throughput is the fact that matching packets is not needed anymore as it's shown in the *Figure 2*, thanks to the dynamic ruleset and maps structures.
Regarding the forwarding methods available in nftables it's possible to provide sNAT and dNAT as the iptables approach.

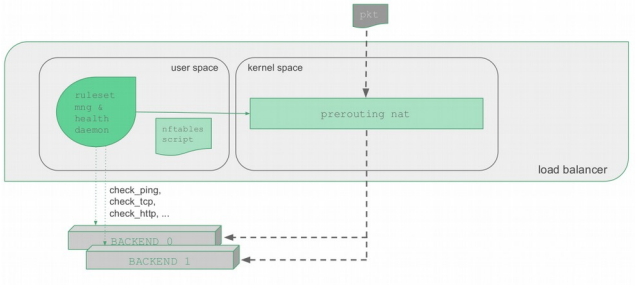

Figure 2. Load balancing with nftables design.

In this case, the user space daemon takes care of all the complexity of the health checks at different layers and the ruleset management, but in this case some concerns are solved:

- The ruleset is handled atomically through a nft ruleset batch and loaded by the nft virtual machine.
- Only it's needed to manage the *nat* table, avoiding marking packets.

Features to accomplish

The basic features that this new load balancing system with nftables should provide in this first prototype are described below.

Schedulers

The schedulers most used and required in this prototype are:

- Round Robin.
- Weight.
- Least Connections.

Persistence

The persistence is required in this kind of technology, at least:

- Persistence per source IP address.

Forwarding methods

The required forwarding methods to be implemented at layer 4 are basically the following in terms of transparency:

- sNAT (transparency off)
- dNAT (transparency on)

Health checks

The backends or real servers monitoring will be performed in user space and at different layers, configurable regarding the application or protocol used.

Good integration

The good integration with other features such as QoS and filtering could be taken into account for every load balancing service.

Use Cases

Some use cases with all these three solutions presented with the basic and required features are shown in this section, according to the example in the *Figure 3*.

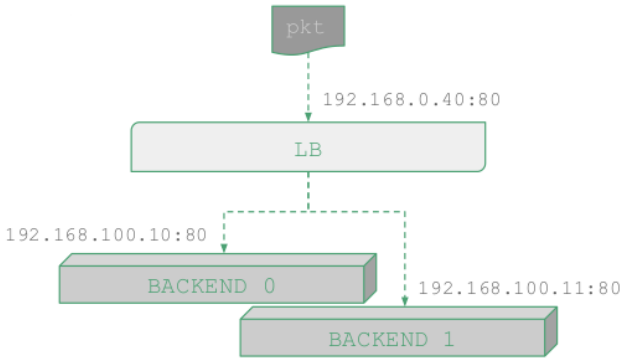

Figure 3. Use cases environment.

Round Robin LB with LVS

According to the given scenario, a simple load balancing service can be created with the 3 commands shown below.

```
ipvsadm -A -t 192.168.0.40:80 -s rr
ipvsadm -a -t 192.168.0.40:80 -r 192.168.100.10:80 -m
ipvsadm -a -t 192.168.0.40:80 -r 192.168.100.11:80 -m
```

The first command creates the virtual service over a certain IP address and one port with a round robin scheduler. The last 2 commands add the backends into the virtual service created.

Round Robin LB with iptables

With the iptables approach, a simple load balancer can be built with these two rules shown below.

```
iptables -t nat -A PREROUTING -m statistic --mode nth \
 --every 2 --packet 0 -d 192.168.0.40 -p tcp \
 --dport 80 -j DNAT --to-destination 192.168.100.10:80

iptables -t nat -A PREROUTING -m statistic --mode nth \
 --every 2 --packet 1 -d 192.168.0.40 -p tcp \
 --dport 80 -j DNAT --to-destination 192.168.100.11:80
```

The ruleset changes the *prerouting* chain in the *nat* table in order to forward the packets detected in the given virtual service under a certain conditions. In this case, the nth extension is used to match the packets every 2 packets per each backend.

Round Robin LB with nftables

The nftables proposal is to have, for this simple case, just one rule to build a load balancer.
Once the table *lb* and the chain *prerouting* are created and associated to the *nat prerouting hook*, with just one rule we can build a round robin scheduler over a certain virtual service over one IP address and one TCP port, as shown in the commands below.

```
table ip lb {
 chain prerouting {
 type nat hook prerouting priority 0; policy accept;
  ip daddr 192.168.0.40 tcp dport http dnat nth 2 map {
   0: 192.168.100.10,
   1: 192.168.100.11
  }
 }
}
```

In this rule the virtual service is associated to the destination IP address and TCP port indicated in the given scenario. Then, it's needed to set the forwarding mechanism to *dNAT* and the scheduler method to nth every 2 packets. The map created will allow to associate the packet numbering with the backend or real server to be used.
Note that by the time this paper is written, the *nth* instruction is not consolidated in nftables yet, so the syntax could change.

Weight LB with LVS

The commands below are used in LVS to create a virtual service with a weighed scheduling method.

```
ipvsadm -A -t 192.168.0.40:80 -s wrr
ipvsadm -a -t 192.168.0.40:80 -r 192.168.100.10:80 \
 -m -w 100
ipvsadm -a -t 192.168.0.40:80 -r 192.168.100.11:80 \
 -m -w 50
```

The first command creates the virtual service over a certain IP address and one port with a weight scheduler. The last 2 commands add the backends into the virtual service created setting a certain weight per backend, with values 100 and 50 respectively in this case.

Weight LB with iptables

In this case, the iptables approach is able to perform a weighted scheduling method using the statistic extension, random mode and with the appropriated probability per backend, converting a certain weight to probability with an easy algorithm.

```
iptables -t nat -A PREROUTING -m statistic \
 --mode random --probability 1 -d 192.168.0.40 \
 -p tcp --dport 80 -j DNAT \
 --to-destination 192.168.100.10:80
iptables -t nat -A PREROUTING -m statistic \
 --mode random --probability 0.33 -d 192.168.0.40 \
 -p tcp --dport 80 -j DNAT \
 --to-destination 192.168.100.11:80
```

The ruleset changes the *prerouting* chain in the *nat* table in order to forward the packets detected in the given virtual service. One rule per backend will be enough in this simple use case using the statistic extension to match the packets randomly with the probability calculated.
The first backend will have twice the weight of the second backend, for this reason, it's used a probability of 1 to match the first rule and the rest 0.33 will pass through the second rule.
The first rule will ensure that all the packets through the virtual service will be marked.

Weight LB with nftables

In this use case, nftables is able to provide a weighted scheduler setting up one dynamic rule.
As the last use case, once the table *lb* and the chain *prerouting* are created and associated to the *nat prerouting hook*, with just one rule we can build a weigthed scheduler over a certain virtual service over one IP address and one TCP port, as shown in the commands below.

```
table ip lb {
 chain prerouting {
 type nat hook prerouting priority 0; policy accept;
  ip daddr 192.168.0.40 tcp dport http dnat \
   random upto 100 map {
    0-66: 192.168.100.10,
    67-99: 192.168.100.11
  }
 }
}
```

In this rule the virtual service is associated to the destination IP address and TCP port indicated in the given scenario. Then, it's needed to set the forwarding mechanism to *dNAT* and the scheduler method to random. The map created will allow to associate the random range

according to the weight for every backend with the real server to be used.
Note that by the time this paper is written, the random instruction is not consolidated in nftables yet, so the syntax could change.
This weighted scheme in nftables will be used as a base in order to create more complex weighted schedulers, as it's described in the following sections.

Weight LB Multiport with LVS

The *multiport* case in *LVS* implies the use of *iptables* in order to mark the packets that matches the virtual service enabling the *multiport* extension, as it's shown below.

```
iptables -A PREROUTING -t mangle -d 192.168.0.40 \
 -p tcp -m multiport --dports 80,443 -j MARK \
 --set-mark 1

ipvsadm -A -f 1 -s wrr
ipvsadm -a -f 1 -r 192.168.100.10:0 -m -w 100
ipvsadm -a -f 1 -r 192.168.100.11:0 -m -w 50
```

The iptables command in the *mangle* table takes care of matching and marking the packets that applies to the multiport virtual service and the given IP address.
The mark value assigned to the matched packets will be used in order to create the virtual service with the weighted scheduler. After that, it'll be needed to add the backends into the new virtual service with the weight value for each one.

Weight LB Multiport with iptables

The iptables approach with multiport is similar to the last use case, as the multiport match is completely compatible with the ruleset presented.

```
iptables -t nat -A PREROUTING -m statistic \
 --mode random --probability 1 -d 192.168.0.40 \
 -p tcp -m multiport --dports 80,443 -j DNAT \
 --to-destination 192.168.100.10:80
iptables -t nat -A PREROUTING -m statistic \
 --mode random --probability 0.33 -d 192.168.0.40 \
 -p tcp -m multiport --dports 80,443 -j DNAT \
 --to-destination 192.168.100.11:80
```

The multiport match will be required in every backend rule.

Weight LB Multiport with nftables

Nftables supports natively multiport capabilities, so it's as simple as including the port list in the weight use case.

```
table ip lb {
 chain prerouting {
 type nat hook prerouting priority 0; policy accept;
  ip daddr 192.168.0.40 tcp dport { http,https } dnat \
   random upto 100 map {
    0-66: 192.168.100.10,
    67-99: 192.168.100.11
  }
 }
}
```

Note that by the time this paper is written, the random instruction is not consolidated in nftables yet, so the syntax could change.

Weight LB IP Persistence with LVS

The persistence is required in many applications and LVS integrates it quite easily associating the client source IP address to a certain backend during a configured timeout, as it's shown below.

```
ipvsadm -A -t 192.168.0.40:80 -s wrr -p 300
ipvsadm -a -t 192.168.0.40:80 -r 192.168.100.10:80 \
   -m -w 100
ipvsadm -a -t 192.168.0.40:80-r 192.168.100.11:80 \
   -m -w 50
```

The last parameter indicates that the persistence timeout will be 300 seconds.

Weight LB IP Persistence with iptables

The IP persistence approach with iptables is complex as it's not supported natively. In order to be able to associate source IP addresses with backends it's used the recent extension, creating one source IP addresses list per backend.
In this complex use case with iptables, two steps will be needed, one step to mark the new connections packets, through the *mangle* table, and then a second step to forward the packets according to the mark through the *nat* table, where every mark will be associated to a certain backend, as it's shown below.

```
iptables -t mangle -A PREROUTING -j CONNMARK \
 --restore-mark
iptables -t mangle -A PREROUTING -m statistic \
 --mode random --probability 1 -d 192.168.0.40 \
 -p tcp --dport 80 -j MARK --set-xmark 1
iptables -t mangle -A PREROUTING -m statistic \
 --mode random --probability 0.33 -d 192.168.0.40 \
 -p tcp --dport 80 -j MARK --set-xmark 2
iptables -t mangle -A PREROUTING -m recent \
 --name "mark1_list" --rcheck --seconds 120 \
 -d 192.168.0.40 -p tcp --dport 80 -j MARK \
 --set-xmark 1
iptables -t mangle -A PREROUTING -m recent \
 --name "mark2_list" --rcheck --seconds 120 \
 -d 192.168.0.40 -p tcp --dport 80 -j MARK \
 --set-xmark 2
iptables -t mangle -A PREROUTING -m state --state NEW \
 -j CONNMARK --save-mark
iptables -t nat -A PREROUTING -m mark --mark 1 -j DNAT \
```

```
 -p tcp --to-destination 192.168.100.10:80 -m recent \
 --name "mark1_list" --set
iptables -t nat -A PREROUTING -m mark --mark 2 -j DNAT \
 -p tcp --to-destination 192.168.100.11:80 -m recent \
 --name "mark2_list" --set
```

The *mangle* rules ensure that the packets are going to be marked following the weighted scheduling method with the weight selected per backend. This is performed with the statistic match and random mode.
After that, it's needed to check if the IP address has already been used and stuck to any backend. The match recent is used to generate persistence creating one source IP list per backend with a certain timeout. If during the IP source list checking the IP is found, then the packet will be marked with the mark of the selected backend.
Finally, the packet will be forwarded through the *nat* table to the backend determined in the mark and store in the list the new IP address entry if needed.
For this approach, three rules will be needed per backend: ensure packet mark, check client persistence and forward packet storing the IP address in the list.
But this iptables approach with IP persistence provides several concerns like:

- The packet must pass through several *mangle* rules until it determines the most affordable backend, with a complexity of *3n*.
- The need of one list per backend implies to check every list and perform several lookups which is expensive.

Weight LB IP Persistence with nftables

The persistence in nftables is not natively integrated but it could be easily built using dynamic maps. With nftables some concerns regarding the iptables approach are going to be solved. Firstly, the packets are not needed to be marked in order to be forwarded and there is only one list per virtual service instead of per backend. With this prototype, it's easy to configure but also we could get much more performance.

```
table ip lb {
  map dnat-cache { type ipv4_addr : ipv4_addr; \
    timeout 120s; }
  chain cache-done { dnat ip saddr map @dnat-cache }
  chain prerouting {
    type nat hook prerouting priority 0; policy accept;
    ip saddr @dnat-cache goto cache-done
    ip daddr 192.168.0.40 tcp dport http dnat \
      random upto 100 map {
      0-66: 192.168.100.10,
      67-99: 192.168.100.11
    }
    map dnat-cache add { ip saddr : ip daddr }
  }
}
```

Once the new lb table is created, it's needed to add a dynamic map where the association between clients source IP addresses and backend are going to be stored with a timeout, in this case it's called *dnat-cache*.
The chain *cache-done* is executed once the IP matches in the list in order to perform directly the forward to the backend associated. The lookup it's going to be performed quite fast in this case.
The *prerouting* chain provides all the scheduling logic and the source IP cache list maintenance. Firstly, it'll be needed to link the chain with the corresponding hook. Then, check in the cache list if the IP already exists in the list and jump to the *cache-done* chain without return.
After that, it's needed to apply the dynamic rule which determines the backend to be selected for any new connection following the weighted base, as it has been shown in the cases before.
Finally, add in the dynamic map the IP address the new associations between source IP address and backend.
Note that by the time this paper is written, the instructions are not consolidated in nftables yet, so the syntax could change.
This weighted scheme with IP persistence in nftables will be used as a base in order to create more complex weighted schedulers with IP persistence, as it's described in the following sections

Weighted Least Connections LB with nftables

The weighted nftables schema shown before can be used as a base to build more complex scheduling methods like weighted least connections. In this use case, the user space daemon is able to gather the number of connections for every backend from the conntrack as it's show in the *Figure 4*. The user space daemon updates the weight for every backend according to this number of established connections, more established connections to a certain backend implies less dynamic weight assigned.

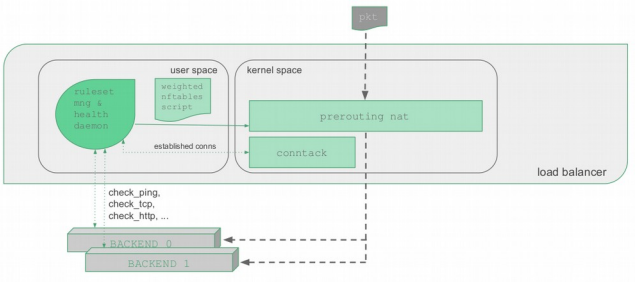

Figure 4. Weighted Least Connections LB with nftables.

Weighted Least Response LB with nftables

In this use case, the user space daemon is able to gather the response time for every health check performed against every backend. All this valuable information is used to estimate the best weight for every backend, more response time spend for a backend implies less dynamic weight assigned.

The *Figure 5* represents the behavior of this use case.

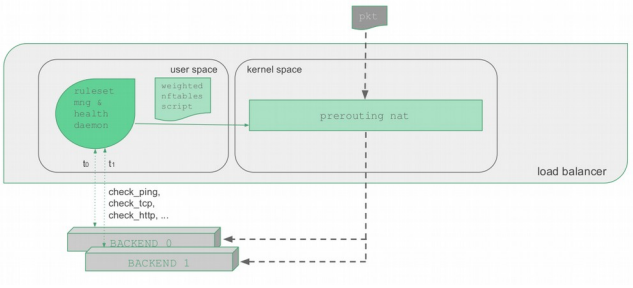

Figure 5. Weighted Least Response LB with nftables.

Weighted Least CPU Load LB with nftables

The weighted Least CPU Load use case needs to gather the CPU Load of every backend in order to estimate the dynamic weight for each one of them.
In this use case, the user space daemon is able to gather the CPU load through SNMP checks against the backends, estimating the dynamic weight where more load implies less weight.
The *Figure 6* represents the behavior of this use case.

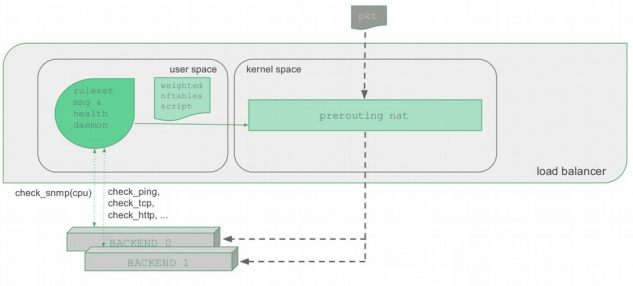

Figure 6. Weighted Least CPU load LB with nftables.

Similar cases could be performed through SNMP scheduler for memory, network consumption, etc.

Work to do

This prototype is still under design and some developments will be needed in order to be able to fulfill the requirements to provide load balancing with nftables:

- Implement some native functions in nftables: random and nth instructions and some maps enhancements.
- User space daemon nft-lbd: health checks support, dynamic weight (least connections, least response, etc.)

Conclusions

The final conclusions regarding the prototype of load balancing with nftables are:

- Simplify the kernel infrastructure, as according to this prototype the complexity is moved to user space.
- Consolidate the kernel development, as nftables could join the efforts in order to avoid duplicated work, better maintenance and native LB support.
- Unique API for network management, as nftables could be able to provide an user interface for firewalling, QoS and load balancing as well.

Even more than that, it's able to build a high performance load balancer with nftables.

Acknowledgements

From the Zen Load Balancer Team, we would like to thank Pablo Neira for his support during the preparation of this talk and mentoring to implement this prototype.

Bibliography

1. Nftables wiki, http://http://wiki.nftables.org
2. Zen Load Balancer documentation, http://www.zenloadbalancer.com/documentation/

Author Biography

Laura García studied Computer Science in the University of Seville and she has been a Software Engineer for HP and Schneider Electric. Over 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.