POLYTECHNIC OF TURIN

Master Degree Thesis in Computer Engineering

High-performance Load Balancer in IOVisor

Supervisors:
prof. Fulvio Giovanni Ottavio Risso
dipl. ing. Sebastiano Miano

Candidates
Aniello Paolo Malinconico
matricola: s229171

Academic Year 2017-2018

Summary

In recent years, the increasingly frequent and daily use of the Internet has made the cloud sector, one of the fastest growing IT sectors especially thanks to all companies that base their services on the Cloud.
In fact, all these companies receive billion of requests for second to access their web contents and so their main purpose is to give to the final user the most performing experience in term of quality of the service and especially, in the response time.
For this reason, in the last years, new technologies were born, for example the *SDN*[1] that suggests to centralize network intelligence in one network component by disassociating the forwarding process of network packets (Data Plane) from the routing process (Control plane) and *NFV (Network Function Virtualization)* that consists to the decoupling network functions such as routing, fire-walling, and load balancing for the enterprise data center.
All of these technologies have found its main application in the data-centers management and networking in order to increase the speed and the performance.

Despite the rapid adoption of these cloud technologies, there is still a big important problem to try to solve and it is the deficit in the ability to reliably distribute workloads across multiple clouds, multiple data-centers and hybrid infrastructures. All of these give as result a poorly distributed workloads and degraded application performance.
A possible solution is to manage the workloads at upper lever demanding this work to a single service called *Load Balancer*.

In this scenario, the development of new kind of Load Balancer that distributes workloads to backend servers in an intelligent way is needed. All of this ensures an optimized use of the resources and so we have an important increase in the performance.
In the last years, the load balancer world is changed to moving from the traditional hardware-based load balancer such as Kemp, Citrix to the software-based load balancer such as HAProxy and NGINX.
The continuous expansion of the software-based load balancer is due to the fact that they are less expensive than the hardware-based one and furthermore, the cloud load balancer can be deployed to manage both a private data-center or in the public cloud without any special configuration and limitation.

[1]SDN stands for *Software-Defined Networking* and it is a novel approach to cloud computing that facilitates network management

In software-based load balancer scenario, a traditional load balancer has a dedicated server in which it acts, forwarding all the traffic to the real backends using a multiple NICs (Network Interface Cards). Furthermore, a traditional load balancer is a single point of failure, in fact, it is deployed in a High Availability Cluster [2], in order to solve this problem.
In addition to these technologies, there is another important news in the networking field: the development of eBPF [3].
eBPF programs, more generally called *IOModules* [4], are used for networking, socket filtering, tracing and security and they can be attached to the Linux TC (traffic control) layer in order to increase the performance, manipulating packets at lowerest level.
Furthermore, in the last year a new technology, called XDP (eXpress DataPath), has been introduced in the networking world and its purpose is to provide a high performance, programmable network data-path in the Linux Kernel.
The main idea is, to create a specific *load-balancing layer* using XDP. In this way, we can run the load balancer software directly on the "back-end" server, with no dedicated central server as the traditional ones. Furthermore, we have an enormous increment in the speed, because thanks to xdp, we manipulate the packets for the load balancing at the lowerest level, before the OS has spent too many cycles on the packet. All of these means that the XDP load balancing uses much less CPU than the traditional ones, maintaining high performance in terms of transactions for second that the load balancer performs.

The thesis's purpose is to implement a new kind of load balancer using the eBPF's technology in order to increase the performance and, at the same time, minimizing the CPU's usage. The load balancer will be implemented as a service of the IOVnet's framework[5] that uses all technologies mentioned above.
In order to implement the most possible generic load balancer, this prototype ensures several features: (1) *Session Affinity* that means that each client will communicate always with the same real backend, (2) *supports* and manages all *TCP, UDP, ICMP traffic* (the load balancer is transparent to the all other protocols), (3) supports the *Weighted Backend* in order to differentiate the load on the backends and finally, (4) *minimizing the number of backend failovers.*
To ensure these features and guarantees a fast way to make the load balancing (in particular in the selection of the real backends), we have chosen, as load balancing algorithm, a new technique called *Consistent Hashing.*
Consistent Hashing is based on mapping each object to a point on the edge of a circle

[2] *HA Cluster* is used to harness redundant computers in groups or clusters that provide continued service when system components fail

[3] *eBPF* stands for *(*extended-Berkeley Packet Filtering) and it is an enhancement over cBPF (which is for classical BPF)

[4] *IO Modules* represents a general purpose input/output object that can be dynamically injected into the Linux kernel at runtime

[5] *IOVnet* is a framework to create and deploy IOvisor-based arbitrary virtual network functions, such as bridges, routers, NATs and so on using IO Modules

through a hashing function.
In fact, the idea is the following: when a packet arrives, we apply the hash function on the 5_values (session) [6] of the packet and places it on the circle, then walks around the circle until the first backend's point is found. So, we have found a backend for that packet, and all future's connection with the same session ID is sent to the same backends (guaranteeing the *Session Affinity*), even when others backends go down.
Furthermore, by using this algorithm, we minimize the number of session's changes when a real backend goes down and so the connection needs to be moved to another real-backend.
During the setting up of the load balancer, it gives the opportunity to set a specific weight to each real backend. If all the backends have the same weight, all the backends will have the same probability to be chosen to serve the request incoming in the load balancer, so in this scenario, we can have a uniform distribution of the workloads.

Finally, for the validation of this prototype, we have tested it in order to get two important values (transaction rate and CPU's usage) to be compared with the most important commercial product such as HAProxy, LVS and IPTables, that use a different technology (for example LVS and IPtables works in the Linux Kernel) and with its direct competitors such as Cilium Load Balancer that uses the same our technology (XDP).

The first tests showed that the load balancer, proposed in this thesis, has a transaction rate slightly less than LVS but greater than HAproxy and IPTables.
On the other hand, comparing the CPU's usage of all the load balancer, during the tests, we have discovered that our load balancer uses 1/5 of the CPU's system used from LVS and about 1/10 of the CPU's system used from HAProxy.
The second tests showed (comparing with its direct competitors, especially Cilium) that the load balancer, proposed in this thesis, has a transaction rate about 60% greater than Cilium's one and the throughput is 2x Cilium's one.

The work, in this thesis, show that it is possible to implement a load balancer in XDP, so having a load balancer with high performance but, at the same time, minimizing the CPU's and system's resources usage.
In the next months, we are going to update the code that performs the load balancer in order to improve its features and performance, for example the load balancer will support the *High-Availability*, so, in this scenario, if the master load balancer will go down, there will be replaced by a second load balancer that has the same configuration of the master.

[6] *5_values (session)* are the destination IP, source IP, destination port, source port, protocol

Acknowledgements

Maybe, the acknowledgments are the hardest part of my thesis, because it is the last point of my university career.
After many exams, many kilometers and in particular, after many sleepless night, finally, I have reached the goal of the Master's degree and so now, i want to give an enormous thanks to my family, especially to my parents that have supported me in each choice that I have taken, giving the most part of the time several good and special tips and my sister that has always been an example to follow in my life.
I want to thank my grandmother and my aunt for their support and a big thank to all my grandparents who will attend this day from Up There.
Thanks to all friends met during these years, from Naples to Turin, they are always in my heart.
Finally, last but not least, I want to thank my second family: the K.I.V.A.B.B.I.'s group for all the time spent together and all times they have supported (physically and abstractly) me during this adventure.

Contents

List of Figures

Chapter 1

Introduction

In order to clarify its purpose and consistency, it is of prominent interest to contextualize this work and illustrate the goals.

1.1 Context and Motivation

In recent years, the increasingly frequent and daily use of the Internet has made the cloud sector, one of the fastest growing IT sectors.
In fact, each day the main web companies receive billion of requests for second to access their web contents, based on what the company offers to their users; for this reason their main purpose is to give to the users the better product that is translated into the increasing of performance, minimizing the response time for the user.
All of these has brought to the increasing interest in the datacenter technologies (especially for the management and networking), in fact, in the last years new technologies have come in the networking world such as the SDN and NFV:

- *Network functions virtualization (NFV)* [1] technology started with service providers trying to achieve IT simplicity, agility, and cost reduction by decoupling network functions such as routing, firewalling, and load balancing for the enterprise data center;
- *Software-defined networking (SDN)* [2] technology is a novel approach to cloud computing that facilitates network management and enables programmatically efficient network configuration in order to improve network performance and monitoring. SDN suggests to centralize network intelligence in one network component by disassociating the forwarding process of network packets (Data Plane) from the routing process (Control plane);

[1] Network Function Virtualization: State-of-the-art and Research Challenges [1]

[2] Software-Defined Networking: A Comprehensive Survey,[2]

In addition to these technologies, there is another important news in the networking field: the development of *eBPF(extended-Berkeley Packet Filtering)*.
eBPF is an enhancement over cBPF (which is for classical BPF[3]), proposed by Alexei Starovoitov in 2013 and it has introduced lots of features in order to increase performance in the packet filtering. eBPF programs are used for networking, socket filtering, tracing and security.
Furthermore, eBPF programs, more generally called *IOModules* [4], can be attached to the Linux TC (traffic control) layer in order to increase the performance, manipulating packets at lowerest level.

These technologies are the basis for the future of networking, especially for the data-centers.
In fact, in the last years, a lot of projects were born: from the precursors of the Cloud Management System world the OpenStack's project to the new Networking project called IOVisor.

1.2 Thesis Organization

The report is organized as follows:

- The **Chapter 2 State of Art** in chapter two we analyze the scenario of the architecture and the currents problems related to the modern load balancers;
- The **Chapter 3 Background** in chapter three we present all the technologies and tools studied and used to implement a powerful load balancer;
- The **Chapter 4 IOV-lbrp Architecture** in chapter four we discuss the load balancer reverse proxy's architecture and our prototype of load balancer;
- The **Chapter 5 IOV-lbrp Implementation** in chapter five we discuss the details of the load balancer reverse proxy 's implementation, analyzing the data structures of the data-path and control-path;
- The **Chapter 6 Testing and Performance** in chapter six we analyze the testing focusing our attentions on the performance. Finally, we compare the results with the actual existing load balancer in commerce;

[3]The BSD Packet Filter: A New Architecture for User-level Packet Capture [3]

[4]IO Visor. Hover Framework: IOModules manager [9]

Chapter 2

State of Art

In chapter two we analyze the actual scenario for the load balancers in commerce.

2.1 Scenario

Nowadays, the main purpose of a data center is to provide flexibility and performance, in term of response time, to their users. Both the management and the networking of a data-center are complex fields, but in the recent years several new frameworks were born such as *OpenStak, Kubernetes.*
OpenStack is for the data-center orchestration and one of the most important competitors of OpenStack is Kubernetes, a new platforms sponsored by Google. While IOVisor project's main purpose is the replacement of the old networking infrastructure by using new technologies such as eBPF. This way will give an important increase in the networking performance especially for the data-center traffic.

Despite the rapid adoption of these cloud technologies, there is still a big important problem to try to solve and it is the deficit in the ability to reliably distribute workloads across multiple clouds, multiple data centers and hybrid infrastructures. All of these give as result a poorly distributed workloads and degraded application performance.
A possible solution is to manage the workloads at upper lever demanding this work to a single service called *Load Balancer* .

In this scenario, the development of new kind of Load Balancer that distributes workloads to backend servers in an intelligent way, is needed. All of this ensures an optimized use of the resources and so we have an important increase in the performance.
Recently, the load balancer world moved from the traditional load balancer such as Kemp, Citrix that are hardware-based load balancer to the new software-based load balancer such as HAProxy, Nginx and Amazon ELB .
As a software load balancer, not only cloud load balancing are less expensive than hardware-based solutions with similar capabilities but they have some advantages, for example, they can be deployed in the public cloud as well as in private data centers without any particular

configuration.

2.2 Traditional vs Untraditional Load Balancer

Traditional Load Balancer

The *traditional load balancer* like IPVS [1] has more NICs (Network Interface Cards), and forwards traffic to the back-end servers.

The current XDP implementation provides several actions such as XDP_TX, from kernel 4.8, that can only forward packets back out the same NIC they arrived on. This makes XDP not useful for implementing a traditional multi-NIC load balancer.

A traditional load balancer easily becomes a single point of failure. Thus, multiple load balancers are usually deployed, in a High Availability (HA) cluster. In order to make load balancer failover transparent to client applications, the load balancer(s) need to synchronize their state (E.g. via IPVS sync protocol sending UDP multicast, preferable send on a separate network/NIC).

Untraditional Load Balancer

The *Untraditional Load Balancer*'s purpose is to implement a load balancer without any dedicated servers for load balancing like the traditional one. Furthermore, we want a load balancer that is 100% scalable and with no single point of failure.

The main idea is, allow XDP to be the load-balancing layer. Running the XDP load balancer software directly on the "back-end" server, with no dedicated central server. It corresponds to running IPVS on the backend ("real servers"), which is possible, but it is generally not recommended (in high load situations), because it increases the load on the application server itself, which leaves less CPU time for serving requests.

In this scenario, XDP has an enormous advantage in the speed. The XDP load balance forwarding decision happens very early, before the OS has spent/invested too many cycles on the packet. This means the XDP load balancing functionality should not use a lot of CPU like the traditional load balancer. Thus, it should be okay to run the service and LB on the same server.

So, in this thesis we will present a prototype of the load balancer working in XDP with the high performance of the traditional load balancer, but reducing the CPU's usage in a significant way.

[1] IPVS (IP Virtual Server) implements transport-layer load balancing, usually called Layer 4 LAN switching, as part of the Linux kernel.

2.3 Goals

Thesis's main purpose is to extend IOVnet framework 's services by implementing a Cloud Load Balancer working as an IOModule that will distribute all datacenter workloads to proper backends in an efficient way.
The Load Balancer is implemented in according to the new eBPF's technology (that perform all the operations in the kernel space) in order to increase the performance. Furthermore, this load balancer supports the common L3/L4 protocols such as ICMP,UDP and TCP .

The main advantages of the new approach are:

- **Performance**: our goal is to implement a load balancer that gives the best performance in term of workload load balancing; this is ensured by using eBPF's technology and efficient load balancing algorithm;
- **Reliability**: the load balancer keeps connections alive when backends go down, minimizing the number of session's changes (automatic backend failover);
- **Session Affinity** to ensure that a client will be served always by the same backends in order to keep the same session (TCP traffic);
- **Scalability** our goal is to implement a load balancer that supports all kind of traffic and ensures an optimal load balance for the workloads of the datacenter;
- **CPU usage**: our goal is to reduce the cpu usage maintaining the same or greater performance;

Chapter 3

Background

In chapter three we present all the technologies studied and used for the implementation of the load balancer .
Furthermore, in the last section, we discuss the used tools for the testing and a subsection about the related works.

3.1 BPF

3.1.1 BPF (o cBPF)

BPF o cBPF (Berkeley Packet Filter) is the first in-kernel packet filter proposed, for the first time, in 1992, by Steven McCanne and Van Jacobson from Lawrence Berkeley Laboratory.
The main purpose is to increase the performance of the packet filtering by minimizing unwanted network packet copies to userspace.
It is based on BSD Unix systems and was introduced in Linux kernel version 2.1.75, in 1997.

The original idea behind BPF is to move the filtering from user space to kernel space, in this way they can be filtered as early as possible. All of this is possible thanks to a virtual machine in the kernel that runs the BPF's programs. BPF programs are written in their own *bytecode* and injected into the virtual machine. These programs have several limitations, for example in order to guarantee the safety of the programs, loops are avoided. A BPF program running in the kernel virtual machine can fetch data from the packet, perform some simple arithmetical computation and comparison. As result, the BPF program can drop or trim packets and pass them to userspace.

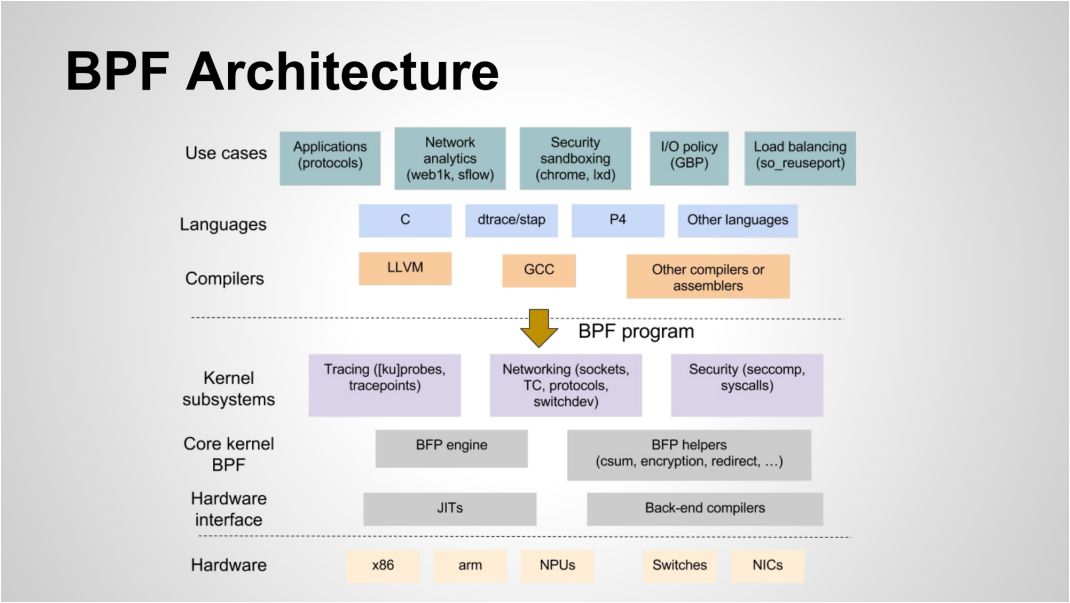

Figure 3.1. Classic BPF 's architecture

There are two registers: an accumulator and an index register. The machine also has a small scratch memory area, an implicit array containing the packet in question, and a small set of arithmetic, logical, and jump instructions. The accumulator is used for arithmetic operations, while the index register provides offsets into the packet or into the scratch memory areas.
A very simple BPF program (taken from the 1993 USENIX paper [3]) might be:

```
1     ldh [12]
2     jeq #ETHERTYPE_IP, l1, l2
3     l1: ret #TRUE
4     l2: ret #0
```

Figure 3.2. BPF 's program example

The first instruction loads the Ethernet type field. We compare this to type IP. If the comparison fails, zero is returned and the packet is rejected. If it is successful, TRUE is returned and the packet is accepted. (TRUE is some non-zero value that represents the number of bytes to save.)

BPF is used by several tools such as tcpdump/libpcap2, wireshark3, nmap, dhcp, arpd. All of these programs use the BPF virtual machine to perform in-kernel efficient packet filtering.

In particular LibPcap[1] for Linux and WinPcap [2] for Windows are two important libraries that provide high-level abstraction and helper in order to generate and compile BPF bytecode for packet filtering. Thanks to this library we can write a program that creates and loads a packet filter, easily using the C-like language, but maintaining bpf's limitations.

3.1.2 BPF just-in-time compiler

A just-in-time (JIT) [6] compiler was introduced into the kernel in 2011 to speed up BPF bytecode execution. This compiler translates BPF bytecode into a host system's assembly code. It is disabled by default on the system, and must be enabled by the admin.

3.1.3 eBPF

Extended BPF (eBPF) is an enhancement over cBPF (which is for classical BPF) proposed by Alexei Starovoitov in 2013 and it was introduced in the Linux Kernel since version 3.15.
It is like BPF but with more resources, such as 10 registers and 1-8 byte load/store instructions. In fact, now, we have not BPF's limitation about avoiding loops, so there can be a loop, which, of course, the kernel ensures terminates properly(in fact there is a limitation on the number of cycles in a loop).
eBPF introduces new elements in the architecture, for example the introduction of global data store called maps, whose state persists between events. eBPF gives us a lot of functions called *helpers* that helps us during the implementation of a bpf program.
Therefore eBPF can also be used for aggregating statistics of events. Further, an eBPF program can be written in C-like functions, which can be compiled using a GNU Compiler Collection (GCC)/LLVM compiler. eBPF has been designed to be JIT'ed with one-to-one mapping, so it can generate very optimized code that performs as fast as the natively compiled code.

[1] Libpcap is the library for *packet capture (pcap) on Linux* [4]

[2] WinPcap is the porting of libcap on Window [5]

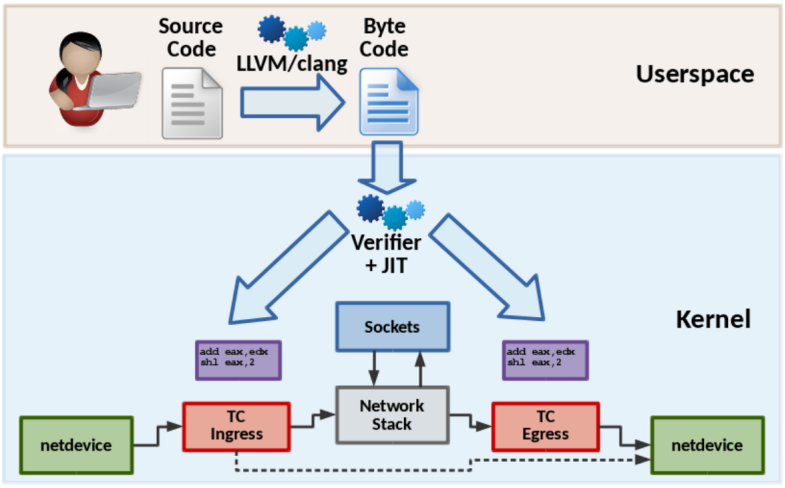

Figure 3.3. eBPF's overview

BPF is based on two important components:

- *CLANG* : acts as C,C++,objectC compiler . It is a frontend of LLVM;
- *LLVM (Low Level Virtual Machine)*: supports the compilation of programs written in C,C++,ObjectC. It supports as frontend CLANG.

BPF program is written in C and it is compiled into an object (ELF) files, which are passed by User Space kernel thought the BPF system call. The kernel verifies the BPF instructions and jits them, returning a new file descriptor for the program, which them can be attached ti a subsystem (e.g.networking). If supported, the system could then offload the BPF program to the hardware NIC (e.g XDP mode).

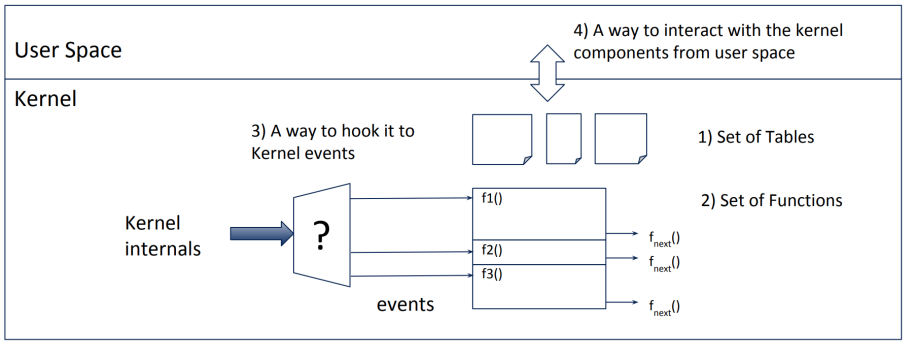

Figure 3.4. eBPF's overall

BPF Kernel Hook Points

For clarity, these are some types of eBPF programs (it depends how it is attached to the hook point):

- The **TC (Traffic Control)** programs can be attached to the *traffic control* layer of the Linux networking stack, both in ingress and in egress. These programs can perform packet manipulation, redirect, clone and in general implement a network function, see 3.1.4 .
- The **XDP (eXpress Data Path)** that provides high performance in the networking (new). XDP programs are attached at the earliest networking driver stage and trigger a run of BPF program upon the packet reception, see 3.1.5 .
- The **Socket Filters** programs can be attached to a socket and they can perform the classic packet filtering process: inject the filter program into the kernel and then the program can return the filtered packet to the userspace, and drop the other packets [8]; it was the original tcpdump use case.
- The **Tracing** the idea behind this kind of program is to trigger the eBPF program every time a kernel or system wide event is intercepted. A program can be loaded, and then attached to some events:kprobe,tracepoints,uprobes,USDT probes4 [8].

In the next chapters we will analyze the load balancer in both Traffic Control (TC) and eXpress Data Path's (XDP) modes.

Three are the main advantages of eBPF :

- **Kernel programmable plane** without having to cross kernel/user spaces boundaries. We don't move packets to user space and back into the kernel;
- **Flexibility of the Programmable Data Path**, in fact, the program can change dynamically depending by the particular USE CASE. If a container does not require IPv4, then the BPF program can be constructed only deal with IPv6 in order to save resources in the fast-path.
- **No Traffic interruption**, in fact, in case of networking (XDP), BPF programs can be updated atomically without having to restart the kernel or any containers.

BPF Verifier

The BPF verifier is the component that verifies if the bpf program is written in order to respect all bpf limitations. It disallows back edges, unreachable blocks, illegal code, finite execution. You cannot have memory accesses from off-stack, or from a unverified source.

3.1.4 Traffic Control (TC)

Traffic control[7] is the name given to the sets of queuing systems and mechanisms by which packets are received and transmitted on a router.
It is a layer in the Linux TCP/IP stack composed by two separated chains: *egress and ingress*.

In the majority of situations, traffic control consists of a single queue which collects entering packets and dequeues them as quickly as the hardware (or underlying device) can accept them. This sort of queue is a FIFO.

3.1.5 eXpress Data Path (XDP)

XDP stands for *eXpress Data Path* and it provides a high performance, programmable network datapath in the Linux Kernel.
In particular, XDP provides bare metal packet processing at the lowerest point in the software stack.
Much of the huge speed gains come from processing RX packet page directly out of drivers RX ring queue before any allocations of metadata structures like *SKBs* occurs.

LIMITATION
XDP cannot be applied for every usecases, for example, it does not provide fairness. In fact, there is no buffering layer to absorb traffic bursts when TX device is too slow and so packets will simply be dropped. You must not use XDP when RX device speed is greater than TX device speed.

We can have three places where we can put an XDP program:

- *Hardware Level*
- *Kernel Level*
- *User Level*

XDP Actions

XDP performs the following several native actions (at lowerest level):

- **XDP_PASS** means that the XDP program chooses to pass the packet to the network stack for processing;
- **XDP_DROP** instructs the driver to drop the packet. In particular, it recycles the page driver: given this action happens, at earliest RX stage in the driver, dropping a packet simply implies recycling it back into the RX ring queue it just "arrived" on;

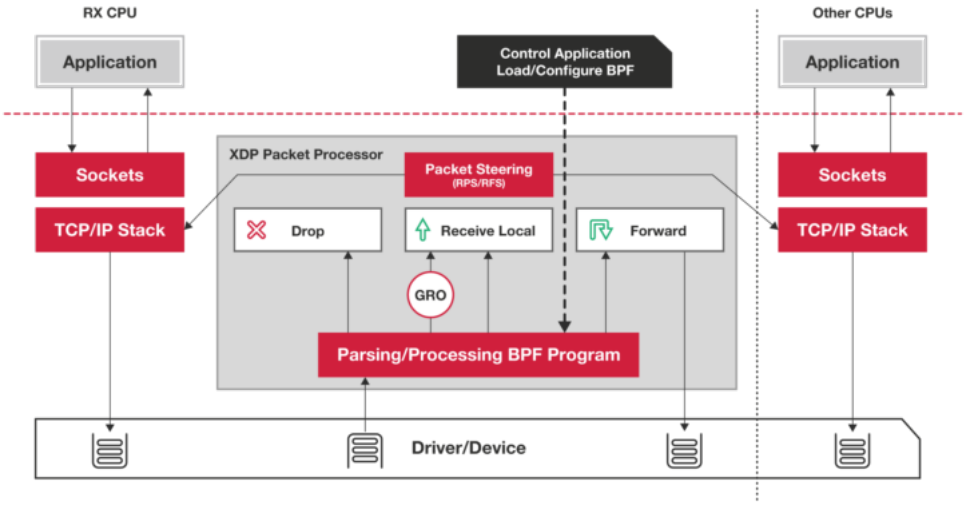

Figure 3.5. XDP's Architecture

- **XDP_TX** results in TX bouncing the received packetpage back out the same NIC arrived on. This action is usually combined with modifying the packet contents before returning action XDP_TX;
- **XDP_ABORTED** is used when an eBPF program terminates with an error. The result is that the packet is dropped;
- **FALL_THROUGHT** used when the eBPF program terminates with an unknown error. The result is that the packet is dropped;

XDP depends on drivers implementing the RX hook and set-up API, in particular when you put XDP program at the hardware level (*XDP_DRV*).
In this mode, it requires a special kind of network card that supports XDP. The following network card, actually, supports it :

- Intel ixgba
- Broadcom bnxt

Use Case

- *vSwitch*: implements a switch data plane in eBPF
- *vRouter*: implements a router/forwarding data plane in eBPF
- *Load Balancer*
- *Security (DDos Mitigator and Firewall)*

3.2 IO Visor Project

The **IO Visor Project** [12] is an open source project and a community of developers to accelerate the innovation, development, and sharing of new IO and networking functions. It is a platform that includes a set of development tools, **IOVisor Dev tools** and a set of **IO Visor Tools** for management and operation of the IO Visor Engine.
Furthermore, IO Visor Project provides a programmable data plane and development tools to simplify the creation and sharing of dynamic **IO Modules** build on top of the IO Visor framework. Its main purpose is to provide *High Performance* in Networking, Tracing, Security and others.

With IO Visor, you can write in-kernel programs that implement atomic networking, security, tracing or any generic IO function. You can also attach these programs to sockets, so that they'll be executed as traffic arrives in the kernel. The biggest benefit of IO Visor is that you can program ANY network logic there (present or future) for ANY new version of your protocol.

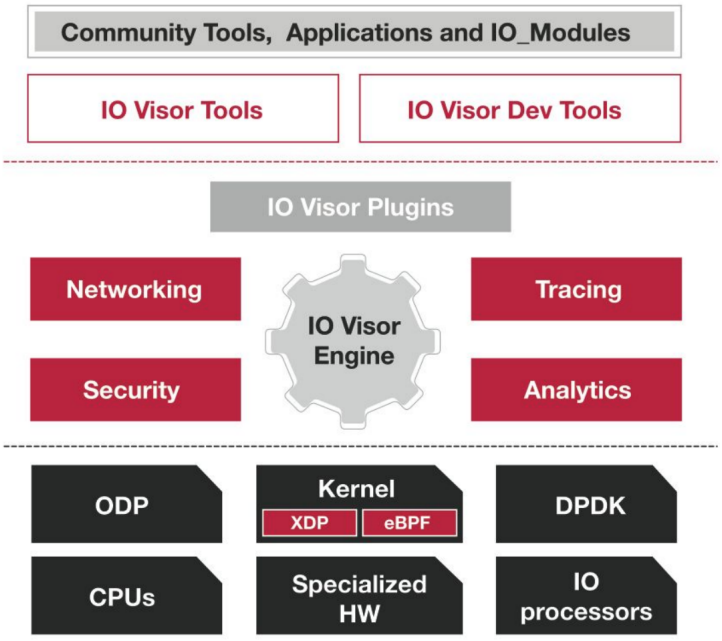

Figure 3.6. IO Visor Project overall

3.2.1 BPF Compiler Collection (BCC)

BCC [3] is a toolkit to make eBPF programs easier to write, with front-ends in Python and Lua. BCC requires LLVM and clang (in version 3.7.1 or newer) to be available on target, because BCC programs do runtime compilation of the restricted-C code into eBPF instructions.
As limitation , it require a linux kernel version greater than 4.1.
It is described by *Ingo Monar* as :

> One of the more interesting features in this cycle is the ability to attach eBPF programs (user-defined, sandboxed bytecode executed by the kernel) to kprobes. This allows user-defined instrumentation on a live kernel image that can never crash, hang or interfere with the kernel negatively.[11]

3.2.2 IO Modules

An IO Modules is a generic input/output object that can be dynamically injected into Linux Kernel and in which can run eBPF programs to perform several operations such as *tracing, security, networking, ecc.*
An IO Module is implemented using eBPF engine, the powerful inkernel virtual machine, already described in the previous section.

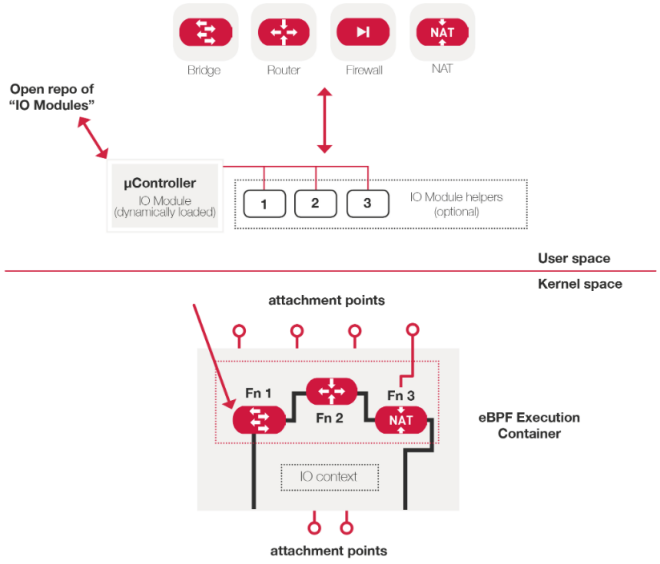

Figure 3.7. IO Modules

[3] BCC stands for BPF Compiler Collection [10]

As we can see from 3.7 an IO module exists both in User and Kernel Space and so we can divide the IO module jobs depending if it is in kernel or user space. Inside the kernel, eBPF programs can be run, chained together by a generic logic, and attached to different kernel hooks (bridge, router as IO modules). Here, you can write the logic of the IO module, for example if your IO module is a router , you should write the data-path and so all functions that a router makes to work correctly. In the *control plane* , considered as userspace component, a controller manages the iomodule injected. It's also possible to communicate between user and kernel space, using maps, and other communication mechanism useful for two reasons: statistic and information read, and configuration and status update of the iomodule.

3.3 Kubernetes

K8s (o Kubernetes) [13] is an open source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications.
Its predecessor was *Borg*, originally designed by Google. The open source project is hosted by the Cloud Native Computing Foundation (CNCF).

Kubernetes is useful when you have several applications distributed on more containers that are hosted on several server hosts. It gives the power to manage and orchestrate your container in one or more cluster. It helps the management and the load balance on all the server hosts, in fact you can scale up and scale down a single application. So, Kuberbetes ensures us scalability and integrity of the application deployed on it.
The basic component is the so-called **Node**. A node is a worker machine in Kubernetes, previously known as a *minion*. A node may be a VM or physical machine, depending on the cluster. Each node has the services necessary to run pods and is managed by the master components. The services on a node include Docker, kubelet and kube-proxy.
There are two kind of node: Master and Worker's node. In a cluster we can have one *Master*, in which there are running components useful for kubernetes's life, and several *Minions* in which you can deploy your application.

Kubernetes is composed of several parts. All its component are designed to be *loosely coupled*, so its components have no knownledge of the definition of other separated components:

- **Pod** is the basic scheduling unit in K8s. It consists of one or more containers that are colocated on the host machine and we can share resources. It can be managed

by API or by Controller;

- **Controller** is a daemon that embeds the core control loops shipped with Kubernetes. In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state;
- **KubeScheduler** is a component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.
- **KubeApiServer** is a component on the master that exposes the Kubernetes API;
- **etcd** is a consistent and highly-available key-value store used as Kubernetes's backing store for all cluster data;

3.3.1 K8s Networking

Kubernetes let us the choice to decide which Network frameworks it will use.
Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- all containers can communicate with all other containers without NAT;
- all nodes can communicate with all containers (and vice-versa) without NAT;
- the IP that a container sees itself as is the same IP that others see it as;

There are several *Network plugins* to replace K8s's Networking.
In this thesis we have created a Load Balancer, that can be supported in Kubernetes, if it uses as Network CNI: the IOVnet's CNI [14]. In this scenario, the load balancer will replace the k8s's load balancer that was performed by using ipTables. In this way we can give to K8s an efficient and performant way to load balance the traffic in its network.

3.3.2 Kubernetes's Service

A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a *microservice*. Kubernetes gives us several types of Service:

- **ClusterIP** exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster.
- **NodePort** exposes the service on each Node's IP at a static port (the NodePort). A ClusterIP service, to which the NodePort service will route, is automatically created. You'll be able to contact the NodePort service, from outside the cluster, by requesting NodeIP:NodePort.

- **LoadBalance** exposes the service externally using a cloud provider's load balancer. NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.

The load balancer implemented in this thesis will support both NodePort and ClusterIP's deployments.

3.4 IOVnet

Iovnet[14] is a framework to create and deploy IOvisor-based arbitrary virtual network functions, such as bridges, routers, NATs, firewalls and more.
It uses the IOModule described in the last section and so enables the creation of complex service chains by connecting together the above modules.
Iovnet includes some existing components such as the *eBPF virtual machine* and the *BCC compiler collection*, including them in a powerful software framework that enables the creation of lightweight and fast network functions, running in the Linux kernel.
The main features of this framework are :

- support generic network services through the definition of a fast (in the kernel space) and slow path (in the user space), hence overcoming the limitations of the eBPF in terms of supporting arbitrary processing;
- integrates a general way to configure network services;
- offers a service-agnostic configuration and control interface that allows interacting with all running network services through the same interface and tools, namely both *REST* (for machine-to-machine communication) and *CLI* (for humans);
- enables the creation of arbitrary and complex service chains in a very efficient and simple way;

Iovnet targets either *end users*, who can leverage its already available elementary services to create complex service chains, and *service developers* who can create the network service they need through a simple and elegant API, which takes care of handling (and automatically generating) most of the glue logic needed for the service to operate, allowing developers to concentrate on the main logic of their service.

3.4.1 Architecture

The IOVnet's architecture is quite complex but all its components can be grouped into 4 main components:

- **iovnetd** is a service-agnostic daemon that checks the iovnet service;
- **iovnet's services** are the Network services(bridge,router, load balancer ecc.) that you can deploy and use them;

- **iovnetctrl** a service-agnostic CLI that allows to interact with iovnet daemon and set uo all the available services
- **libiovnet** a library that keeps some common code to be reused across multiple network functions;

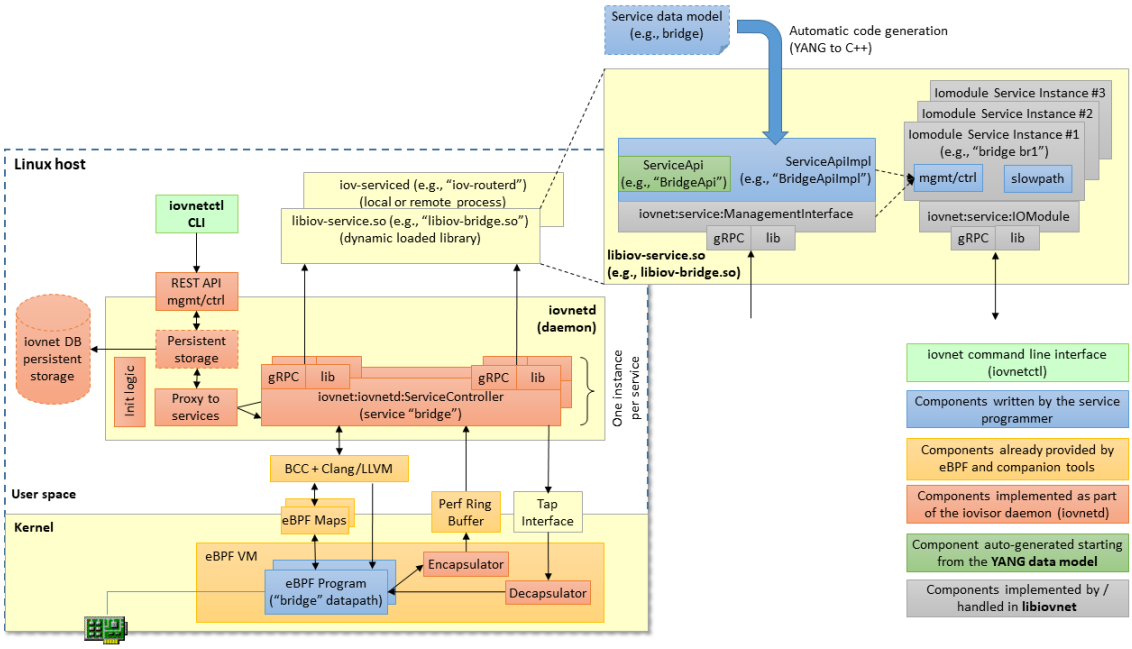

Figure 3.8. IOVnet's architecture

iovnetd

iovnetd is the main component of the iovnet architecture. It is a service-agnostic daemon that allows controlling the entire iovnet service. In fact, it manages the service life (it can start,stop,remove a service) .
This module acts mainly as a proxy: it receives a request from its northbound REST interface, it forwards it to the proper service instance, and it returns back the answer to the user. Iovnet supports both *local services*, implemented as shared libraries, which are installed in the same server of iovnetd and whose interaction is through direct calls, and *remote services*, implemented as remote daemons possibly running on a different machine, which communicates with iovnetd through gRPC.

iovnet's service

By using Iovnet, we can deploy several network services (a.k.a., network functions such as bridge, router, NAT, etc.), in a simple way. Iovnet services can be imagined as a single

program that manages all network's functions related to its, running in an IOModule and all of this gives the opportunity to change the code and the behaviour of that service and reload the service at run time, without interfering with the others services.
Each service implementation includes the *data-path*, namely the eBPF code to be injected in the kernel, the *control/management plane*, which defines the primitives that allow configuring the service behavior, and the slow path, which handles packets that cannot be fully processed in the kernel. While the former code runs in the Linux kernel (*fast-path*), the latter components are executed in user-space *(slow-path)*.

In this thesis, we will implement the Load Balancer Reverse Proxy as a service of IOVnet's framework.

3.5 Tools used for testing

In this section, we will present all tools studied and used for testing our load balancer prototype.

Siege

Siege [4] is a multi-threaded http load testing and bench-marking utility. It was designed to let web developers measure the performance of their code under duress.
It allows one to hit a web server with a configurable number of concurrent simulated users. Those users place the webserver "under siege." Performance measures include elapsed time, total data transferred, server response time, its transaction rate, its throughput, its concurrency and the number of times it returned OK.
These measures are quantified and reported at the end of each run.

The command used for the test is the following :

siege -b -c *concurrent users* -t *time (virtual IP : port)* -v

where the flags :

- **b** stands for *bench-mark*; it is used to have maximum performances ; without this flag the tool performs a request for second;
- **c** stands for *concurrent users* and it allows you to set the concurrent number of simulated users to num;
- **t** stands for *time*; it allows you to run the test for a selected period of time in terms of seconds or minutes;
- **vip** stands for *virtual ip*, the destination ip of the requests;

[4] Siege, linux tool for bench-marking [21]

- **v** (option) stands for *VERBOSE*; it prints the HTTP return status and the GET request to the screen;

iPerf3

iPerf3[5] is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). For each test, it reports the bandwidth, loss, and other parameters. This is a new implementation that shares no code with the original iPerf and also is not backward compatible.
iPerf was originally developed by NLANR/DAST. iPerf3 is principally developed by ESnet / Lawrence Berkeley National Laboratory. It is released under a three-clause BSD license.

The command used for the test is the following (server side), on each linux network namespace in the PC3:

iperf3 -s -p *port*

The command used for the test is the following (client side):

iperf3 -c *vip* -p *port*

htop

htop [6] is an interactive system-monitor process-viewer and process-manager. It is designed as an alternative to the Unix program top. It shows a frequently updated list of the processes running on a computer, normally ordered by the amount of CPU usage. Unlike top, htop provides a full list of processes running, instead of the top resource-consuming processes. Htop uses color and gives visual information about processor, swap and memory status.
htop is written in the C programming language using the ncurses library.
Because system monitoring interfaces are not standardized among Unix-like operating systems, much of htop's code must be rewritten for each operating system. Cross-platform support was added in htop 2.0.

The command used for system monitoring is the following :

htop

[5] iPerf3, Linux tool for networking traffic measurement [22]

[6] htop, Linux tool for monitoring CPU and Memory of the system [23]

3.6 Related Work

3.6.1 Commercial Load Balancer

In order to understand the results previously got, we have compared our load balancer with the most used and performant load balancer in commerce. In particular, we have compared our results with other three kinds of load balancers:

- **HAProxy**[7] is an open source software that provides a high availability load balancer and proxy server for TCP and HTTP-based applications that spreads requests across multiple servers;
- **LVS** The Linux Virtual Server [25] is a highly scalable and highly available server built on a cluster of real servers, with the load balancer running on the Linux operating system.
- **iPtables** based on Netfilter kernel module. Netfilter[26] is a flexible packet manipulation framework built into the Linux 2.4 and 2.6 series of kernels.

HAProxy Load Balancer

HAproxy is designed for distributing TCP and HTTP traffic across multiple computing resources in a network.
Load balancing is intended to help high traffic sites and sites that experience routine spikes in request traffic. It provides intelligent load distribution for some of the largest websites on the internet which processes over thousand and thousand requests per second.

HAproxy is designed to do three things and to do them quickly and reliably. These three things are processing incoming request traffic, distributing the request traffic to the servers in the back end, and querying those back-end servers' health.
HAproxy comes with many distributions of Linux and is also available in the apt and yum package libraries.

IPVS (Linux Virtual Server)

IPVS [8] implements transport-layer load balancing in the Linux Kernel, providing the load balancing at the lowerest level.
IPVS is incorporated into the *Linux Virtual Server (LVS)*, where it runs on a host and acts as a load balancer in front of a cluster of real servers.
LVS supports TCP and UDP traffic and, furthermore, it makes services of the real servers appear as virtual services on a single IP address.

[7] HAproxy stands for High Availability proxy [24]

[8] IPVS stands for *IP Virtual Server*

IPVS is merged into versions 2.4.x and newer of the Linux kernel mainline.
There are three versions of LVS load balancer:

- **LVS-NAT**
  In the NAT mode, all the incoming packets are rewritten by changing destination address from the VIP to the address of one of the real-servers and then forwarded to the real-server.
  On the returning way, all packets from the real-server are sent to the director where they are rewritten and returned to the client with the source address changed from the RIP to the VIP.

- **LVS-DSR**
  In the DSR (Direct Server Return) mode, the load balancer manipulates all the incoming packets in order to load balance them to the several real backends. On the returning way, the packets do not pass into the load balancer, but they are forwarded, directly, to the client that has sent the request, relieving the network load balancer of the need to handle the heavy traffic load.

- **LVS-TUN**
  LVS-Tun is based on LVS-DR but it operates by using an IP-IP tunnel to send the packets to the real backends.
  In fact, the LVS code encapsulates the original packet (CIP->VIP) inside an ipip packet of DIP->RIP and routed to the real-server.

> Most commercial load balancers are not set up anymore using the LVS-DR mode (at least in the projects I've been involved). The load balancer is becoming more and more a router, using well-understood key technologies like VRRP[9] and content processing.

IPTables

Iptables is based on Netfilter's module[10] in the Linux kernel and it is implemented as a framework that allows callback functions to be attached to network events. These callback functions can be implemented as kernel modules, thus allowing IPTables to inherit the flexibility of the Linux kernel module system.
IPTables is used to set up, maintain, and inspect the tables of IPv4 packet filter rules in the Linux kernel. Several different tables may be defined. Each table contains a number of built-in chains and may also contain user-defined chains. Each chain is a list of rules which can match a set of packets. Each rule specifies what to do with a packet that matches.

[9] VRRP, Virtual Router Redundancy Protocol, RFC5798 https://tools.ietf.org/html/rfc5798

[10] Netfilter, Linux's kernel module for packets manipulation, https://www.netfilter.org/

3.6.2 Cilium Load Balancer

Cilium [27] is open source software for transparently securing the network connectivity between application services deployed using Linux container management platforms like Docker and Kubernetes.
At the foundation of Cilium is a new Linux kernel technology called BPF, which enables the dynamic insertion of powerful security visibility and control logic within Linux itself. Cilium's framework implements a load balancer as service defined as :

> Distributed load balancing for traffic between application containers and to external services. The loadbalancing is implemented using BPF using efficient hashtables allowing for almost unlimited scale and supports direct server return (DSR) if the loadbalancing operation is not performed on the source host [11].

[11] Cilium Load Balancer, https://github.com/cilium/cilium

Chapter 4

IOV-lbrp Architecture

In the following chapter, we discuss the *Load Balance Reverse Proxy*'s features and analyze our solution of a load balancer reverse proxy in the IOVnet's framework.

4.1 Load Balancer Reverse Proxy

The *Load Balancer Reverse Proxy* joins two main key words: Load Balancer and Reverse Proxy.

- A **Load balancer** is a service whose task is to load balance the workloads of the traffic in a system such as in a data-center network to distribute traffic across many existing paths between any two servers.
  Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource.
  A Load balancer can be hardware or software (as the prototype implemented in this thesis).
  Furthermore, it can work at L4 (Transport Layer) or L7 (Application Layer) of ISO-OSI stack.[1]
  Our prototype is an L4 load balancer since making load balancing at L4 is faster than at L7.

- **Reverse Proxy** means reverse proxy server that can act as a "traffic cop", sitting in front of your backend servers and distributing client requests across a group of servers. It intercepts all requests for a specific *VIP (Virtual IP)* and send to proper backend servers which are in another subnet.

[1] ISO-OSI: standard of networking stack, composeb by 7 layers

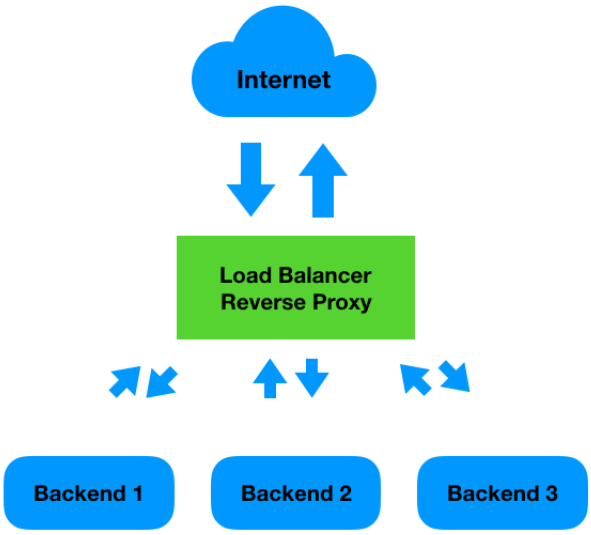

Figure 4.1. Load Balancer's simple schema

4.2 IOV-lbrp

In this thesis, we have implemented a load balancer reverse proxy as service for the IOVnet's framework.
According to the algorithm, the load balancer receives all incoming IP packets and it loads balances only the packets for a specific Virtual IP and it forwards them to the real servers by replacing their IP destination address with the one of the real server, chosen by the load balancing logic.
Hence, IP address rewriting is performed in both directions, for traffic coming from the Internet and the reverse.
This load balancer supports all kind of traffic but it manages only TCP, UDP, and ICMP traffic; all others packets , called *unknown packets* (e.g., ARP; IPv6) are simply forwarded as they are.

4.2.1 Architecture

First of all, the load balancer has *two ports* [2], by which it can recognize the incoming traffic and the outcoming traffic.
In fact :

- *Frontend port*: is the load balancer interface attached to the interface in which all requests arrive; it connects the LB to the clients that connect to the virtual server,

[2]This implementation will be explained in next section

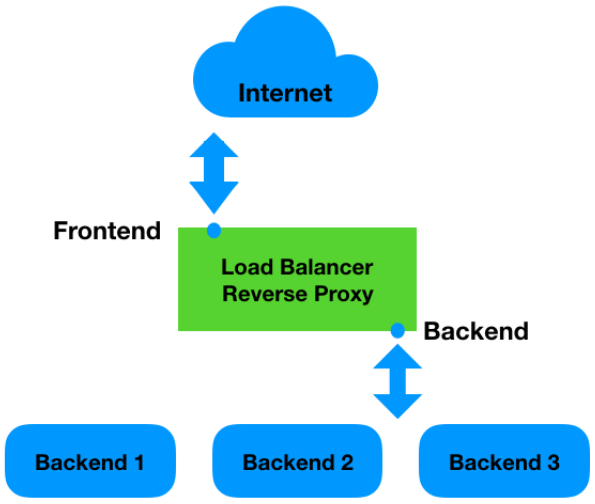

Figure 4.2. Load Balancer's ports schema

likely running on the public Internet;

- *Backend port*: is the load balancer interface attached to the interface that connects the LB to backend servers;

Because its kind of implementation (*reverse proxy*), the load balancer architecture can be split into two subarchitecture depending if the traffic flow is in ingress or in egress from/to load balancer.
In fact, we have :

- **Ingress** the main purpose is to load balance all packets with destination IP as a *Virtual IP*, while all other packets are forwarded as they are;
- **Egress** the main purpose is to do reverse proxy of all packets previously load balanced and so put the *Virtula IP* as the source of the packets;

Architecture - INGRESS

As written above, in this scenario we refer to all packets coming from *Frontend port*. The load balancer receives and intercepts all packets whose destination IP is a *Virtual IP* and makes load balancing, choosing a backend of that specific virtual IP and change the destination IP and port (if needed).

So, we have :

- if the destination IP is a *Virtual IP*, the service must do load balancing to one of VIP backend's set ;

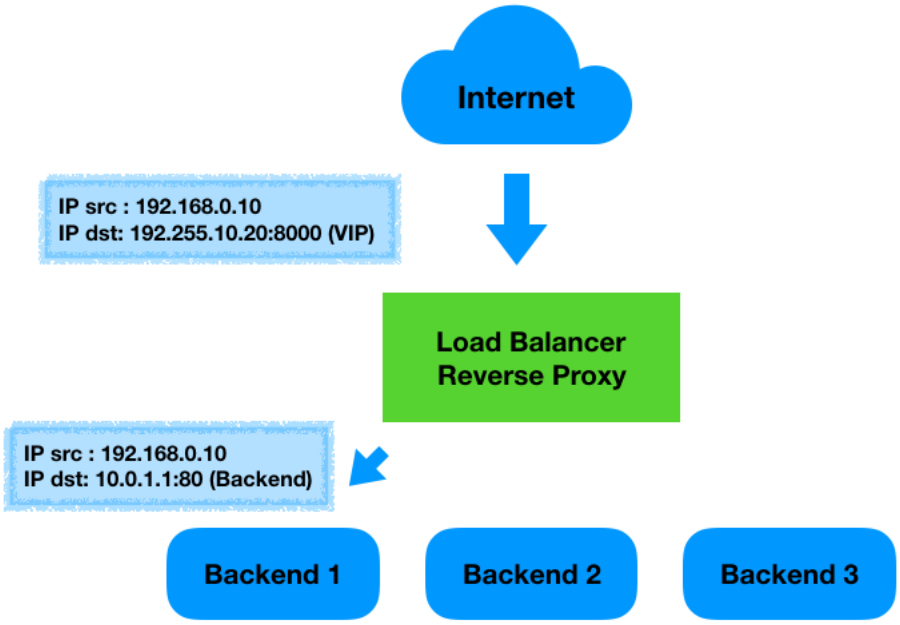

Figure 4.3. Load Balancer's ingress architecture (Load Balancing 's example)

- otherwise, for all others cases the load balancer forwards the packets in a transparent way;

As we can see from the picture above 4.3 , the load balancer checks if the destination IP of the packet is a Virtual IP (in this short example , the Virtual IP is 192.255.10.20:8000), previously configured in the load balancer, and after choosing the real backend IP to send the packet, it finally forwards the packet to the real backend, in fact after the load balancer the destination IP is 10.0.0.1 .

Architecture - EGRESS

This architecture refers to a scenario in which all packets come in the load balancer thought the *backend port* and so the load balancer should manage all returning packets, making reverse proxy only to the packets that previously have been load balanced and all others packets are simply forwarded.

In this scenario, we apply a hashing function to the session of the packet and checks if the state was saved previously:

- if it is found in the map , the load balancer makes reverse proxy changing the source IP from real backend to the VIP;

- otherwise the load balancer forwards the packets in a transparent way;

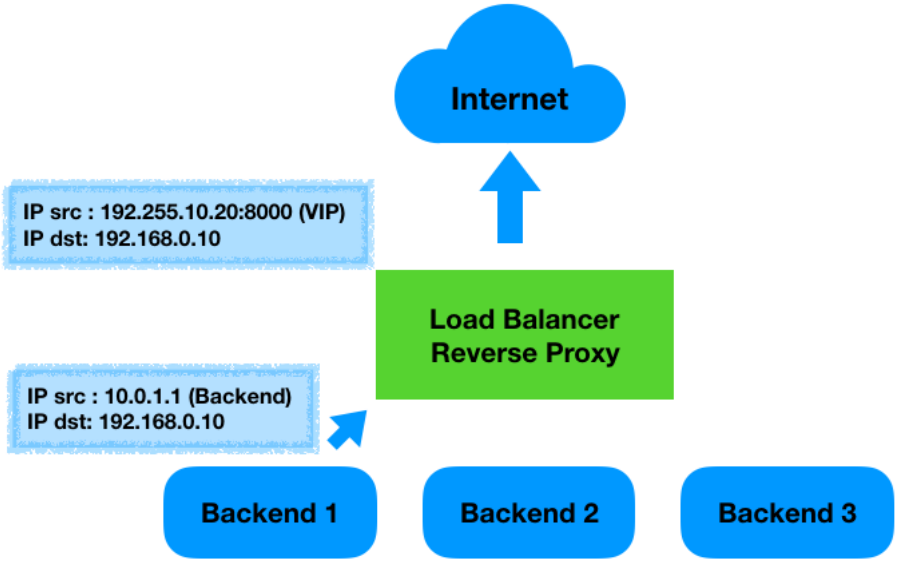

Figure 4.4. Load Balancer's egress architecture (Load Balancing 's example)

As we can see from the picture above 4.4, the load balancer checks if the packet is related to an existing session or not; if it exists, the packet is manipulated by changing the source IP and port (if necessary) from real backend 10.0.0.1 to the Virtual IP 192.225.10.20.

4.2.2 Features

The IOV-lbrp has several features in order to create a load balancer that guarantees the following features :

- *Multiple virtual services* support, each mapped on a Virtual IP with its backend pool;
- *ICMP traffic*'s support, so you have the possibility to ping any Virtual Service (VIP);
- *Session affinity*'s support: a TCP session is always terminated to the same backend server even in case the number of backend servers changes at run-time (e.g., a new backend is added);
- Each virtual service can be mapped depending on the traffic protocol (you can have a virtual service that supports only UDP or TCP), in which each Virtual IP can have a different number of backend servers;
- *Weighted Backend*: Each backend has its weight that determines the probability to be chosen during the load balancing mechanism;

4.2.3 Load Balancing Algorithm

In this subsection, we will explain, in the details, the idea behind the load balancer. We will show, starting from the first proposal of the algorithm, the weak points and how we can solve them, giving, at the end, the final version of the load balancer.

We can summarize the load balancer jobs in the following three points:

- **Load Balancing**
  The main operation of the load balancer that will manipulate only the packets whose destination IP is a Virtual IP; it affects the destination ip of the packet by changing the destination ip (Virtual IP) to one of the real backends of that service;

- **Reverse Proxy**
  This operation will be applied to all packets returning from the backends that, previously, have been load balanced; it affects the source IP of returning packets by changing the source IP from real backend to corresponding Virtual IP;

- **Redirect**
  This operation affects all others packets (whose destination ip is not a Virtual IP and the packet has not been load balanced); in this case, the load balancer is transparent to forward all *unknown packets* to the opposite interface arrived in;

I Proposal

The following flow-graph describes the first idea of the load balancer algorithm:

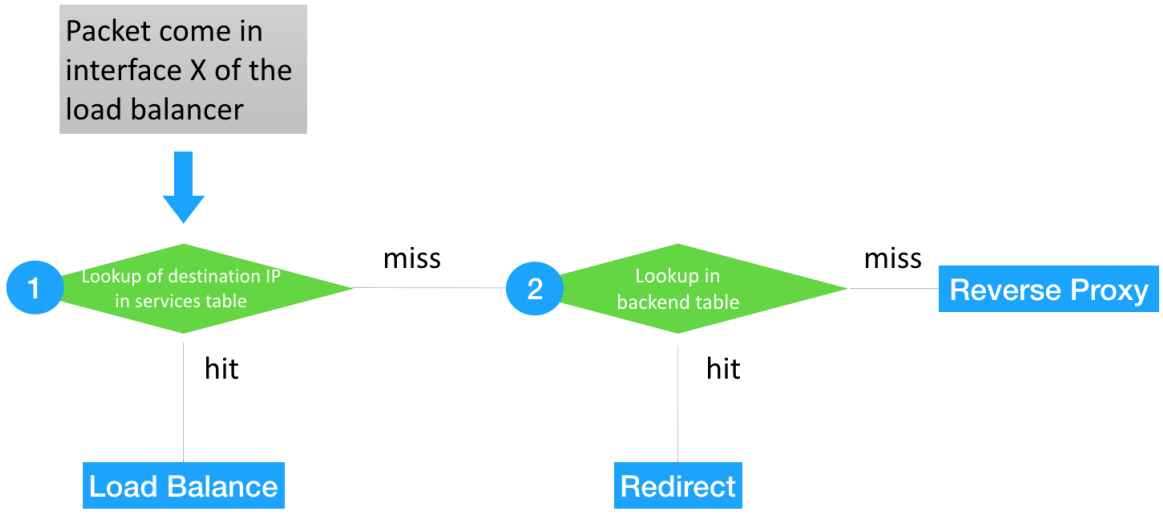

Figure 4.5. I proposal of load balancer algorithm

As you can see from the picture above, we should make two look-up into the maps to choose what the load balancer should operate (load balancing, redirect, reverse proxy). All packets, coming into the load balancer, follows the following flow :

1. **I look-up** on destination IP to checks if it is a Virtual IP:
   - *hit* -> the packet is load balanced to one of backends associated to that Virtual IP;
   - *miss* -> go to point 2;
2. **II look-up** on destination IP to checks if it is a backend;
   - *hit* -> the packet is redirect to the opposite port of the load balancer (from Frontend to Backend and vice versa);
   - *miss* -> this is the case when a packet previuosly load balanced, returns in the load balancer and so we must make reverse proxy to mask the real backend;

This algorithm works as well but it is not much optimized, in fact in the incoming case (packets coming into the load balancer with destination IP as a virtual IP) we should make one lookup on a map, while all other cases we should make two look-ups.

II Proposal

In order to optimize the algorithm and so reduce the number of lookups into a map, the II Proposal introduces two *fixed interfaces* to avoid the second lookup in the table and so increase the performance of the algorithm .
As discussed in 4.2, the two interfaces are *Frontend Interface* and *Backend Interface*.

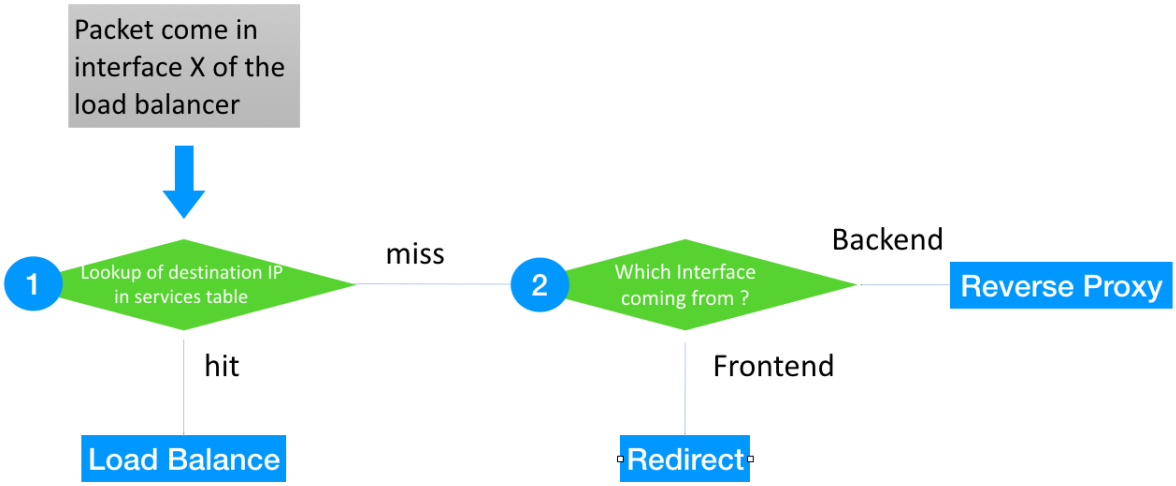

Figure 4.6. II proposal of load balancer algorithm

As you can see from the picture above, now, we should make only one look-up into the map, in fact, the second look-up of the I proposal is replaced by introducing these two fixed interfaces by which we can determinate what the load balancer should operate on the packet.
All packets, coming into the load balancer, follows the following flow :

1. **I look-up** on destination IP to checks if it is a Virtual IP:
   - *hit* -> the packet is load balanced to one of backends associated to that Virtual IP (*Load Balancer*);
   - *miss* -> go to point 2;
2. **Which Interface the packet coming in?**
   - *Frontend* -> the packet is redirect to the opposite interface of the load balancer (*Redirect*);
   - *Backend* -> this is the case when a packet previuosly load balanced, returns in the load balancer and so we must make reverse proxy to mask the real backend(*Reverse Proxy*);

With this new approach, we can avoid a look-up in a map. In fact, now, making only one look-up in the map, the algorithm should be faster than the previously showed.

Reverse Proxy Problem

All of these algorithms work only if the Virtual IP associated with the backend pool is one and only one. In fact, if there are two Virtual IP that refers to the same backend pool, when the load balancer makes the reverse proxy, it does not know which VIPs choose for this operation .

Real Case - Kubernetes

To understand better what I have written above, in Kubernetes a backend pool can be associated with multiple Virtual IP, in the most of cases they are :

- *NodePort*
- *ClusterIP*

So, suppose that we have an unique backend pool associated to two virtual service (NodePort and ClusterIP).
How we can distinguish between NodePort and ClusterIP's virtual IP?
To build a load balancer supported in Kubernetes, we have changed the algorithm in order to *keep trace of destination ip of the packets that are load balanced.*
In this way, on the return path we can change the source IP of the packet (that will be a backend) and write the correct IP (saved previously in the map).

Final Algorithm

In order to solve the previous problem, we have changed the algorithm adding a new table whose job is to store the packet session. The packet session is composed by 5 values :

- *destination IP*
- *source IP*
- *destination port*
- *source port*
- *protocol*

These five values determine an *unique session* of that packet.
So, the final flowchart of the load balancer algorithm is described in the following picture :

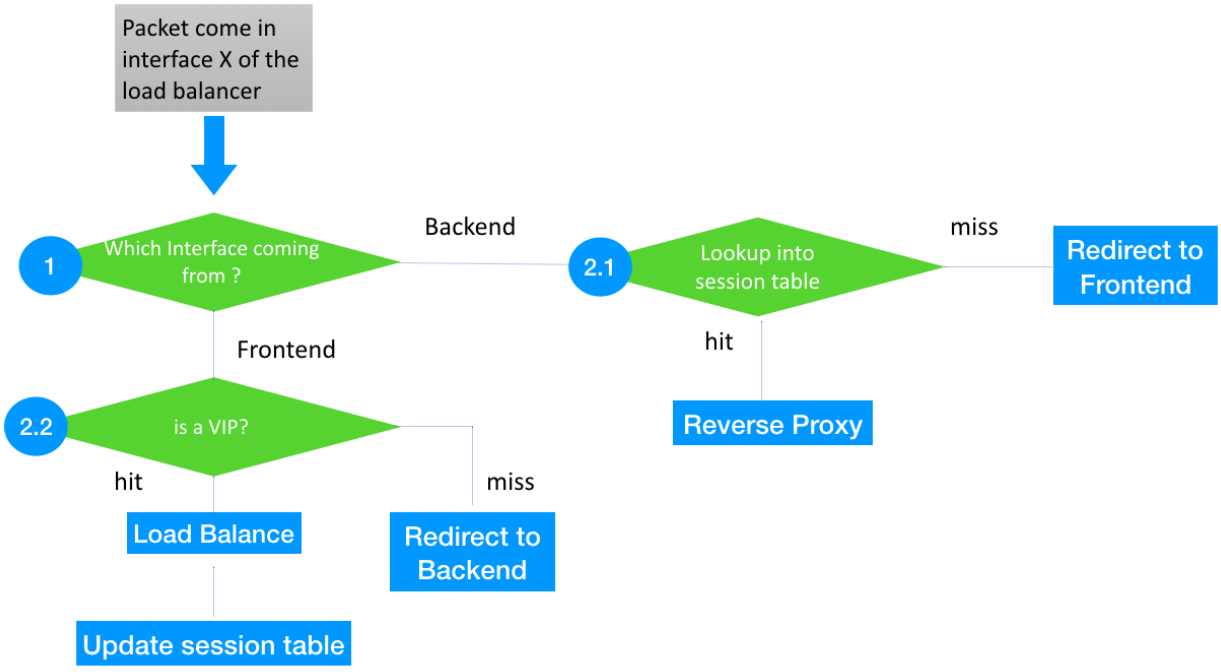

Figure 4.7. Final proposal of load balancer algorithm

As you can see from the picture above, we have introduced the session table. All packets, coming into the load balancer, follows the following flow :

1. **Which Interface the packet coming in?**
   - *Frontend*: the destination ip is a Virtual IP?

  - *hit* -> In this case we performe two operations:
    (a) the *load balancing*
    (b) *update the session table* in order to store the packet info for the reverse proxy's operation;
  - *miss* -> *Redirect* the packet to the backend interface.
- *Backend*: performs the lookup into session table
  - *hit* -> The load balancer performs the *reverse proxy* operation on the packet;
  - *miss* -> The packet is *redirected to the backend interface*; *Redirect* the packet to the fronted interface.

Alternative Algorithm (srcIPrewrite)

In order to avoid the session table, we have implemented another algorithm, called *srcIPrewrite* that is a bit more performed against the previous one (session table).

LIMITATION: This algorithm cannot be applied to every case but in some special cases such as in *Kubernetes* [3].
This limitation is due to the necessity to know the source IP of the requests that will be load balanced.

How does it work?
The structure of the algorithm is similar to the previous ones. In this case, we do not update the session table, but, we change the source IP of the packet, after load balancing. This new IP is unique and corresponds only to that source IP.
On the other hand, when the packet returns, the load balancer makes a lookup into the map to check if the dest ip is the previously set and changes it, restoring the original IP.

The execution of this algorithm should be faster than previous the one because we avoid all the operations about the session table(lookup into the map, applying hashing functions and so on).

[3] Kubernetes, open source framework for the orchestration, discussed in 3.3

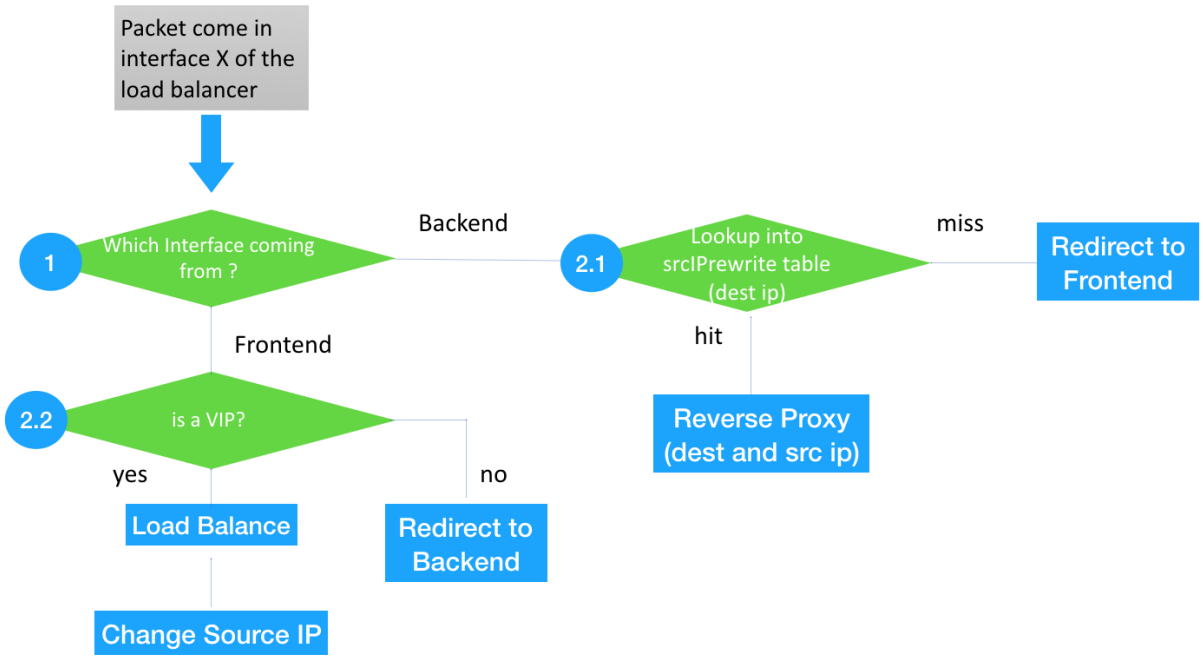

Figure 4.8. Alternative algorithm: srcIPrewrite

4.2.4 Backend selection

In this section, we will discuss the *backend's selection* that is one of the most important parts of the load balancer.
We know that each Virtual IP (Virtual Service) has an own backend pool and so, after we have checked the destination ip of the packet as a Virtual IP, we should look-up in the backend array to choose the proper backend.

The idea behind the selection of the backend is very simple and it consists to get the index of the backend array. This index is the rest of the division between the values resulted by applying a hash function on the session packet values and the number of the backend array entries, that is a fixed number.
So, it can be summarized by the following formula :

$$\mathbf{H(\ ipS\ |\ ipD\ |\ pS\ |\ pD\ |\ proto)\ \%\ N\ =\ index}$$

where:

- H: the hash function;

- *ipS, ipD, pS, pD, proto*: are the five values to which we apply the hash function to get a unique number that identifies uniquely the session;
- *N*: is the number of entries of the backend array; it is a fixed number (the reason of this is described later);
- *index*: the index of the backend's array where are stored the real backend's information;

As we can see, the backend's selection depends on the number of divisor N, that represents the backend array's size. So, if a backend is added or removed from the pool, the value N changes and so all previous established connections changes (no more session affinity). This is a problem because we want that, if the backend's pool changes, all the old connections must maintain their old session to its real backend.

Session's affinity problem

In order to understand better this problem, the following example is shown.

Suppose that there is a Virtual Service with its Virtual IP with a backend's size of 4. So there are four backends related to that Virtual IP.

- **BACKEND = 4**

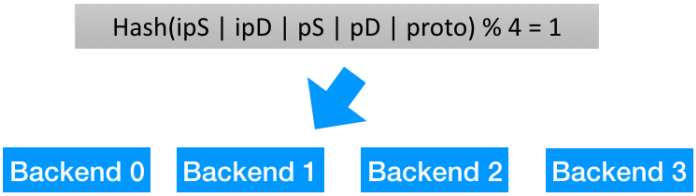

Figure 4.9. Example of backend's selection (backend = 4)

As we can see from the example above, the hashing on the 5-values of the packets gives as result a number that will be divided by 4. We have the index 1 as result, so all requests with this session ID are load balanced to the backend 1.

Now, let's suppose that the backed 0 goes down, so the number of backend's pool is 3. For the same packet used before, we have the following:

- **BACKEND = 3**

As we can see from the example above, now, the hashing value corresponding to the session in the previous example, is divided by 3 and so we have 2 as index, so all requests for this session ID are load balanced to the backend 2 while before removing the backend 0, it was backend 1.

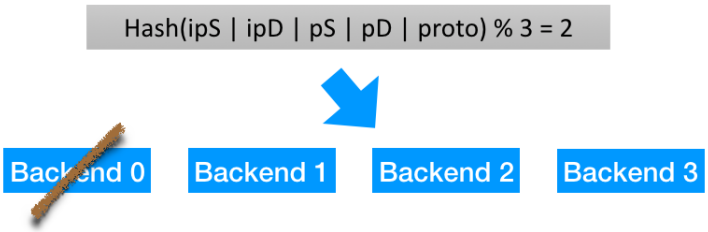

Figure 4.10. Example of backend's selection (backend = 3)

So, what we want is that, after the backend 0 goes down, all connections with the same Session ID will go always to the same backend.

Session's affinity solution

A very tricky solution to this problem is the so-called **Consistent Hashing**[15].
This technique was introduced in 1997 by Karger and others members, at MIT for use in distributed caching.

Consistent Hashing

Consistent Hashing is based on mapping each object to a point on the edge of a circle, for example the system maps each backend to one point on the edge of the same circle (let call it *backend point*).
The idea is the following, when a packet arrives, we apply the hash function on the 5_values (session) [4] of the packet and places it on the circle, then walks around the circle until the first backend's point is found. So, we have found a backend for that packet, and all future's connection with the same session ID is sent to the same backends, even when others backends go down.
Suppose to have two sessions S1 and S2 and four backends, the following picture shows how this algorithm works.

[4] 5_values (session) are the destination ip, source ip, destination port, source port, protocol

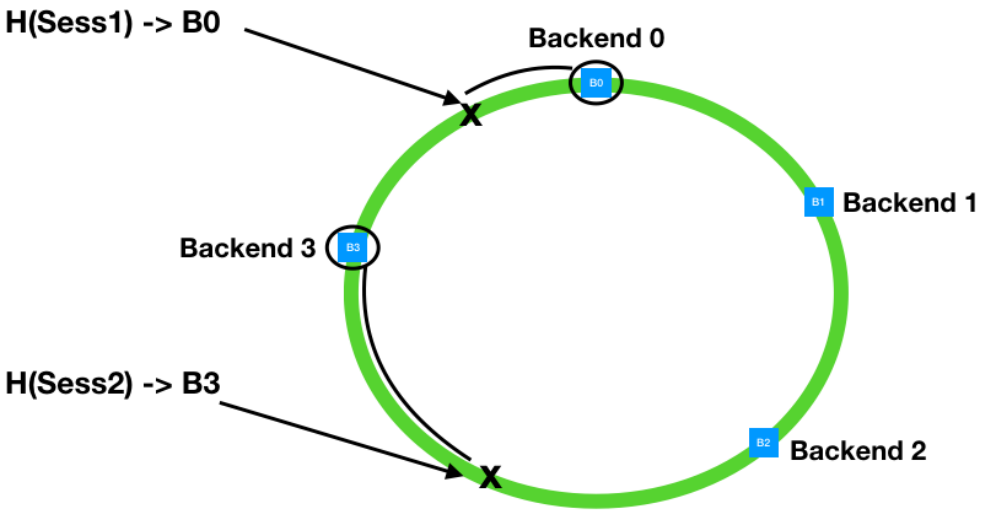

Figure 4.11. Consistent Hashing 's example

On the other hand, if that backend goes down, the point it maps to will be removed. In this scenario, the packet will choose the next backend's point (that will place after the ex-backend's point).
For example, if backend 3 goes down, all packets referring to the session 2 will have as backend's destination, not more backend 3, but the next backend in the circle: Backend 0.
So, we use the Consistent Hashing for our load balancer in the backend's selection .
The main important thing is to use a fixed array's size for the backend, so the value N must be fixed.
Furthermore, suppose N the backend array's size and M is the number of backends.

- **BACKEND = 4**

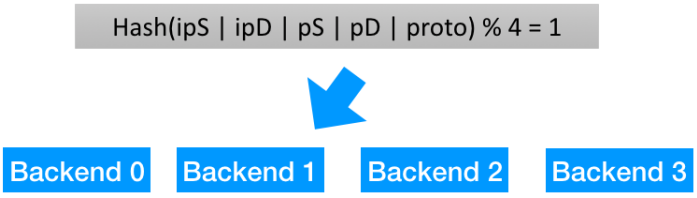

Figure 4.12. Example of backend's selection (backend = 4)

- **BACKEND = 4**

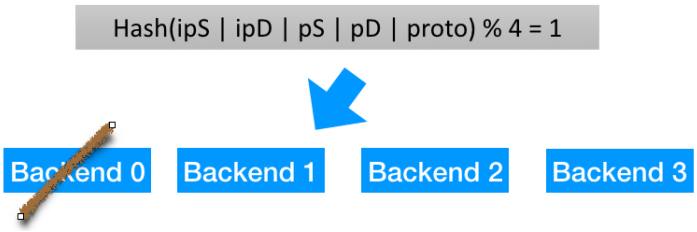

Figure 4.13. Example of backend's selection (when a backend goes down)

As we can see, now, the backend 0 's fail does not influence the backend's selection.So the packet still goes to the backend 1.

This solution ensures:

- *Connection Tracking* the load balancer saves the state of all sessions;
- *Backend's failover resilience* the load balancer ensures that, when a backend goes down, all its connection are moved on others backends;
- when a backend goes down, we need to *remap only N/M entries plus one*, in order to minimize the number of entry's changes;

So, now, we have a fixed array filled with the backend's information. Each entry of the array refers to a backend. But, how this array is populated? And, when a backend goes down, how this array changes?

How to populate the backend's array

We need an algorithm with the following features:

- *Fast Convergence* the algorithm should converge in a short time even if the number of backend's and its array's size are huge;
- *Minimize the number of changes* when a backend fails;

This algorithm is based on two main structures:

- **Permutation Matrix** a huge matrix with N rows and M column where each column is filled by a range of numbers from 0 to N-1; each value is set randomly; for this operation, we use a random number generator's algorithm called *Fisher-Yates Shuffle algoritm*[16];

Fisher-Yates Shuffle algorithm

The Fisher-Yates shuffle is an algorithm for generating a random permutation of a finite sequence of values. The algorithm produces an unbiased permutation: every permutation is equally likely.
The modern version of the algorithm is efficient: it takes time proportional to the number of items being shuffled and shuffles them into place.

- **Final Backend's array (lookup table)** the final array containing a backend's information in each entry;

Once filled the *Permutation Matrix*, it is the starting point to create or update(when a backend is added or removed) to create the *Final Backend's Array* with the following algorithm:

1. Start from the cell [0,0] , get the value X stored in that cell and looks if the entry of the backend's array corresponding to the X's position is empty;
   1.1. if it is empty, stores the corresponding backend (that is represented by the column) at X's position, and go to the point 2;
   1.2. if it exists, go to point 3;
2. once stored the value in the array, we increment the column, so re-start from point 1, but analyzing the next cell [0,1] (next backend);
3. we increment the row, so re-start from point 1, but analyzing the next cell [1,0](next row of the previous backend);

Example

To understand better how the entire algorithm works, we can see the following example. Suppose to have 3 backends and that the array's size is 7, so we have M=3 and N=7. By using the *Fisher-Yates Shuffle algorithm*[5], the permutation table is filled randomly like the following example. As you can see, each column(backend) is filled by a range number from 0 to N-1.

[5]We use this algorithm for simplicity, but you can fill the *Permutation Matrix* using any algorithm that generates random numbers into a fixed range

Permutation Table

	B0	B1	B2
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Lookup Table

0	
1	
2	
3	
4	
5	
6	

Figure 4.14. Backend's selection algorithm (Step 0)

The first iteration acts as in the picture below:

Permutation Table

	B0	B1	B2
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Lookup Table

0	
1	
2	
3	B0
4	
5	
6	

Figure 4.15. Backend's selection algorithm (Step 1)

As you can see from the picture above, we start from the cell [0,0] inside that there is stored the number 3. So, in the *Look-up Table*, we check if the third position is empty or not; if empty, we store the backend B0 (column 0) and go on to the next column (B1).

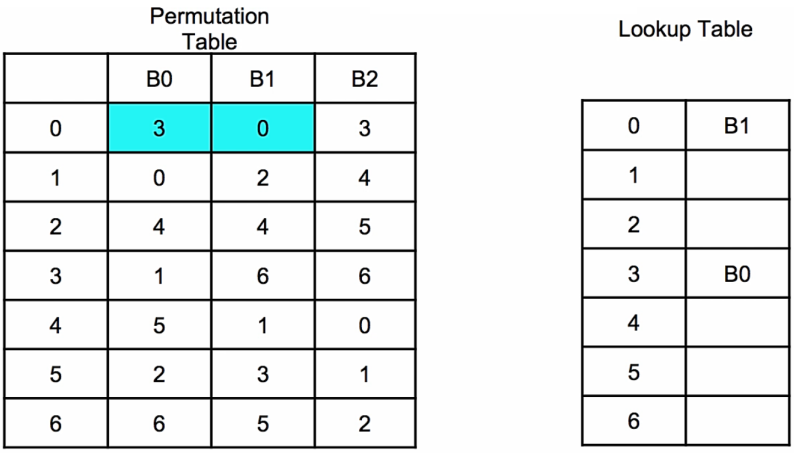

Figure 4.16. Backend's selection algorithm (Step 2)

As the previous step, now, we analyze the cell [0,1] (first item of backend B1) that stores the value 0. So, in the *Look-up Table*, at zero position (that is free), we store the backend B1(column 1) and go on to the next column (B2).

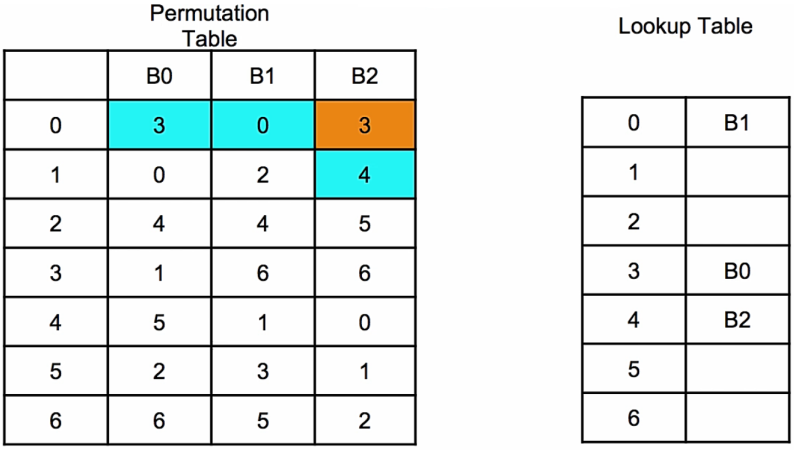

Figure 4.17. Backend's selection algorithm (Step 3)

In the third step, we analyze the cell [0,2] (first item of backend B2) that stores the value 3. But, in this case, the third's position of the *Look-up Table* is not free, so we need to find the next row of B2 column.
The cell [1,2] stores the value 4; the fourth position of the array is free and so we store here the backend B2(column 2) and go on to the next column (B0).

Permutation Table

	B0	B1	B2
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Lookup Table

0	B1
1	B0
2	
3	B0
4	B2
5	
6	

Figure 4.18. Backend's selection algorithm (Step 4)

So, the algorithm restarts from the first column (B0). The next B0 free's value in the *Look-up Table* is at first position (element in the cell [3,0]). So, we store the backend B0(column 0), and go on to the next column (B1).

Permutation Table

	B0	B1	B2
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Lookup Table

0	B1
1	B0
2	B1
3	B0
4	B2
5	B2
6	B0

Figure 4.19. Backend's selection algorithm (Final Step)

After some cycles, the algorithm ends giving us the *Look-up Table* completely filled. As we can see how the distribution of backends is homogeneous less than one.

Suppose, now, that the backend B1 goes down, how the *Look-up Table* changes? When a backend is removed, the *Look-up Table* must be updated; this operation consists into recalculating the *Look-up Table* using the same previously *Permutation Table* but ignoring the column corresponding to the removed backend, in this case, B1.

Permutation Table

	B0	B1	B2
0	3	0	3
1	0	2	4
2	4	4	5
3	1	6	6
4	5	1	0
5	2	3	1
6	6	5	2

Lookup Table

	Before	After
0	B1	B0
1	B0	B0
2	B1	B0
3	B0	B0
4	B2	B2
5	B2	B2
6	B0	B2

Figure 4.20. Backend's selection algorithm (Step after B1 down)

As we can see from the picture above, this algorithm minimizes the number of backend's changes. In fact, the number of *Look-up Table* 's entries changed is 7/3 plus 1.

Weighted Backends

In this example, the backend's distribution in the *Look-up Table* is homogeneous. In fact, each backend has the same probability be chosen.
In our load balancer, we have implemented the feature by which you can set the probability for each backend to be chosen.

For example, suppose that the *Look-up Table*'s size is 25 and suppose to set the following weights :

- *B0*: weight 10
- *B1*: weight 100
- *B2*: weight 5

The distribution will be the following:

- *B0* will have 22 entries
- *B1* will have 2 entries
- *B2* will have 1 entry

Chapter 5

IOV-lbrp Implementation

The starting point of the Load Balancer 's implementation is its *YANG data model*. From the YANG, we have generated the classes, some of them, have been updated in order to implement the load balancer's logic.
All files are stored in the following folders:

- *iov-loadbalancer-rp/example* this folder contains several examples for the load balancer's usage;
- *iov-loadbalancer-rp/resources* this folder contains the yang datamodel *lbrp.yang*;
- *iov-loadbalancer-rp/src* this folder contains all sources codes of the load balancer's implementation; it is diveded in several folders containing from the cli's code to the controlpath 's implementation;
- *iov-loadbalancer-rp/test* this folder contains several tests that can u perform on the load balancer;

5.1 YANG Model

YANG (Yet Another Next Generation) [1] is a data modeling language for the definition of data sent over the *NETCONF network configuration protocol*.
YANG is a modular language representing data structures in an XML tree format. The data modeling language comes with a number of built-in data types. Additional application specific data types can be derived from the built-in data types.

5.1.1 Lbrp YANG Model

In this section, we will show some extract of load balancer's YANG data model.

[1] Yang data model, Data Modeling Language for the NetworkConfiguration Protocol (NETCONF) [17]

- **Ports Data Type** two kinds of ports (*frontend and backend*), this value is mandatory during the load balancer configuration;

```
uses "basemodel:base-yang-module" {
    augment ports {
        leaf type {
            type enumeration {
                enum FRONTEND { description "Port
                   connected to the clients"; }
                enum BACKEND { description "Port
                   connected to the backend
                   servers"; }
            }
            mandatory true;
            description "Type of the LB port (e.g.
               FRONTEND or BACKEND)";
        }
    }
}
```

Listing 5.1. Port Data Type (YANG Data Model)

- **Service Data Type** represents the Service object that is identify uniquely by the key:
  - *key* is grouped into 3 values: *vip (virtual ip) - port - proto* that identify uniquely the service object;
  - *value* is the pool of backends related to the service with that key;

```
list service {
    key "vip vport proto";
    description "Services (i.e., virtual
       ip:protocol:port) exported to the client";
    leaf name {
        type string;
        description "Service name related to the
           backend server of the pool is connected
           to";
    lbrp:cli-example "Service-nigx";
    }
    leaf vip {
        type inet:ipv4-address;
```

```
        description "Virtual IP (vip) of the service
           where clients connect to";
    lbrp:cli-example "130.192.100.12";
    }
    leaf vport {
        type inet:port-number;
        description "Port of the virtual server
           where clients connect to (this value is
           ignored in case of ICMP)";
    lbrp:cli-example "80";
    }

    leaf proto {
        type enumeration {
            enum ICMP;
            enum TCP;
            enum UDP;
            enum ALL;
        }
        mandatory true;
        description "Upper-layer protocol associated
           with a loadbalancing service instance.
           'ALL' creates an entry for all the
           supported protocols";
    }

    list backend {
        key "ip";
        description "Pool of backend servers that
           actually serve requests";
        leaf name {
            type string;
            description "name";
        lbrp:cli-example "backend1";
        }
```

Listing 5.2. Service Data Type (YANG Data Model)

- **Backend Data Type** represents the Backend object that is identify uniquely by the key:
  - *key* is the *ip* of the backend that identifies uniquely the service object, according to the network topology;

– *value* is a structure based on tree values of the backend: *ip, port, weight* where ip and port are the backend's information and the weight is a feature to set the probability to be chosen;

```
        list backend {
            key "ip";
            description "Pool of backend servers that
               actually serve requests";
            leaf name {
                type string;
                description "name";
            lbrp:cli-example "backend1";
            }

            leaf ip {
                type inet:ipv4-address;
                description "IP address of the backend
                   server of the pool";
            lbrp:cli-example "10.244.1.23";
            }

            leaf port {
                type inet:port-number;
                description "Port where the actual
                   server listen to (this value is
                   ignored in case of ICMP)";
                mandatory true;
            lbrp:cli-example "80";
            }

            leaf weight {
                type decimal64 {
                    fraction-digits 2;
                }
                description "Weight of the backend in
                   the pool";
            lbrp:cli-example "0.1";
            }
        }
    }
```

Listing 5.3. Backend Data Type (YANG Data Model)

5.2 Code's Structure

The load balancer's code can be split into parts due its nature (Figure 3.8), in order to have, at the same time, a powerful cli to manage all the structures that are stored in the kernel or in the driver of the NIC, and the ebpf's program that acts as a load balancer:

- **Data-path code**
  This part of code runs in the kernel space (TC_MODE) or in the driver of the NIC (XDP_MODE). This code performs the load balancer's algorithm.
- **Control-path code**
  This part of code runs in the User Space and it is useful for the management and the setup of the load balancer;

5.2.1 Data-path

The *Data-Path Code* is the core of all the implementation of the load balancer's service. This code is the eBPF's program that runs in the *Traffic Control chain* or inside the driver of the NIC (*XDP*) in an *IO-Module*.
In fact, this part of the implementation is dedicated to make load balancing of all packets coming into the service, so here the algorithm showed in the last chapter is implemented.

Data Structures

In the *Data-Path Code*, we use several structures in order to keep information for the load balancer's work:

- **Services map**

```
BPF_TABLE("hash", struct vip, struct backend, services,
    MAX_SERVICES);
```

Listing 5.4. Services Map in the data-path

where:

  - *type hash* the key is an hashed-value;
  - *struct vip* is the key. It is composed by four values (*ip, port, protocol, index*) that identifies uniquely an entry in the map;
  - *struct backend* is the value. It is composed by three values (*ip, ip_rewrite, port*) that represents one of the backend's pool;

The *Service Map* contains multiple entries for each virtual service (i.e., virtual IP, protocol, port).

When *index = 0*, the value in the hash map is actually not a couple IP address/port but the number of backends (also called *pool size*) associated to that virtual service. Then, for following entries (i.e., when *index > 0*), the *struct backend* keeps the actual data (IP/port) for that backend.

```
Example:
VIP 1, index 0 -> Number of backends: 4
VIP 1, index 1 -> Backend 2
VIP 1, index 2 -> Backend 1
VIP 1, index 3 -> Backend 2
VIP 1, index 4 -> Backend 1

VIP 2, index 0 -> Number of backends: 2
VIP 2, index 1 -> Backend 1
VIP 2, index 2 -> Backend 1

where VIP 1 has 4 backends and VIP2 has 2 backend.
```

Listing 5.5. Services Map's example

The number of the backend in this table may be different from the number of actual servers in order to keep session affinity when a server is added/removed, without having to move the traffic from a backend1 to a backend 2 (which will break the TCP session).
The load balancing algorithm will select the proper backend based on an hash on the most significant fields (i.e., session id) of the packet:

```
__u32 backend_id = session_hash % pool_size + 1;
```

Listing 5.6. Services Map's hashing algorithm

- **hash session map**
  Keeps the sessions handled by the load balancer in this table.

```
BPF_TABLE("lru_hash", __u32 , struct vip , hash_session,
   MAX_SESSIONS);
```

Listing 5.7. hash-session map

where:

  - *type lru* that stands for *least recent used* in order to mantain only the recent connections;

  - *u32* is the key. It is the hash key (hash the translated session: Hash(ipS | ipD=BCK | pS | pD | pro));
  - *struct vip* is the value. This structure contains the virtual service ID, to be used to translate the return traffic;

  This is needed to translate the session ID back for return traffic. If there's a match, the (return) source IP address/port is translated with the VIP; if there is no match, the traffic is left untouched. Note that the hash is calculated, in both directions, using the IP.dst of the backend, not the original one (virtual IP address of the service).

- **srcIPrewrite map**
  This map contains the mapping between backen_ip_srcIPRewrite with the pair (backend_ip and VIP).

```
BPF_TABLE("hash", __u32, struct bck_vip, srcIpRewrite,
   MAX_SERVICES);
```

Listing 5.8. srcIPrewrite map

  where:

  - *type hash* the key is an hashed-value;
  - *u32* is the key and it refers to the srcIPrewrite ;
  - *struct bck_vip* is the value. This structure is composed by three values (*backend ip, virtual ip, port*);

  When packets come from Backend Port, we make a lookup on this map.
  If match, we change source and destination ip of the packets by makin the reverse proxy for VIP(srcIP) and srcIpRewrite to set the old ip as destination.
  If NOT, we use the session_table.

- **bckold_bcknew map**

```
BPF_TABLE("hash", __u32, __u32 , bckold_bcknew,
   MAX_SERVICES);
```

Listing 5.9. hash-session map

  where:

  - *type hash* the key is an hashed-value;
  - *u32* is the key and it refers to the ip of the backend;

– *u32* is the value and it refers to the srcIPrewrite corresponding uniquely to that backend;

This map contains the mapping between old backend ip and new backend ip (srcIPrewrite IP).
This map is filled only if srcIPrewrite is enabled.

Workflow

According to the load balancer's algorithms (Figure 4.7, 4.8), each packet coming in the load balancer can follow two special work-flows, in order if the srcIPrewrite's flag is enabled for that Virtual IP.
In the data-path, all packets coming in the load balancer, go thought the *rx_handler()* in which the load balancer's algorithm are implemented.
This function manages only the ICMP,TCP,UDP packets, in fact it is transparent for all others protocols (i.e. Arp, GRE, ecc.) that will be simply forwarded from an interface to the other one.

Checksum

Due to packet's manipulation after applying the load balancer (changing ports and ip's values), we need to recalculate the IP's checksum on the packet.
When we use the ebpf's program in *Traffic Control*, the newest versions of the linux kernel [2], give to the developer several *Helper's functions* that "helps" the developer in several functions, for example to recalculate the checksum there are several helper's function as *bpf_[l3,l4]_csum_diff()* and *bpf_[l3,l4]_csum_replace()*.
On the other hand, if the eBPF's program runs in *XDP*, these functions are not valid and so we need to recalculate the checksum of the packet by own function.

According to the RFC 1624 [3], we have implemented our custom checksum's functions using the following formula :

```
HC  - old checksum in header
C   - one's complement sum of old header
HC' - new checksum in header
C'  - one's complement sum of new header
m   - old value of a 16-bit field
m'  - new value of a 16-bit field

HC' = ~(C + (-m) + m')  =  ~(~HC + ~m + m')
```

[2] Starting from *version 4.11* of the Linux Kerne, currently the version 4.16 is released

[3] RFC 1624 is the third official RFC about the checksum's update (RFC 1141, RFC 1071),[18] [19] [20]

The formula above is used to recalculate the checksum, after packet's manipulation, via incremental update. This means that we do not recalculate the checksum on the entire packet but only on the changed values and so it does not influence the load balancer's algorithm performance.

5.2.2 Control-Path

The *Control Path* is the portion of code that runs in the User Space and it is useful for the management and the setup of the load balancer.
Here, we can find all codes and data structures that are used to update the ebpf's maps in the data-path, in particular :

- **cli**
  Cli stands for Command Line Interface and it is useful for the set up of the load balancer (adding/remove/update the load balancer, add/remove/update the services and their backend's pool and so on).
  All of cli's code is autogenerated by the YANG data model.

- **load balancer's management**
  It refers to all classes written for the load balancer's management such as Service Class, Backend Class, Lbrp Class and so on (derived from the YANG Data Model). In these classes, we have implemented the algorithm for the *backend's selection* and all functions to update the maps in the data-path.
  In particular, the main important data-structure is the so-called *backend_matrix* that acts as the *Permutation Table*.

Chapter 6

Testing and Performance

In this chapter, we discuss the use's case used for testing and validation of the load balancer's prototype.
Then we will analyze the data given by the testing phase to derive the relative performances. Finally, we will compare our performances with performances of others load balancer, actually in commerce.

6.1 Topology

The network topology studied and used is the same for all tests and it is the simplest possible topology in order to minimize the latency about other components during the measuring of the performances.
The standard Client/Server's system is the topology used for the tests in which there is the Load Balancer, that receives all traffic from the client and makes load balance between all backends in which there is a server running.
In particular, we have divided the entire topology into three parts in order to minimize the job of the machine that hosts the load balancer.

6.1.1 Hardware Configuration

The entire topology is distributed on three desktop pc with the same hardware configuration:

- **PC1 (client)**
  - Intel(R) Core(TM) i5-4258U CPU @ 2.40GHz
    - core : 2
    - cache size: 3072
  - Ram size 8 GB (2x4 DDR3)
  - Scheda di rete Intel 10Gbit ????
  - SO Ubuntu Linux 16.04 LTS, kernel version 4.14

- **PC2 (load balancer)**
  - Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
    - core : 4(+4)
    - cache size: 8192 kb
  - Ram size 16 GB (2x8 DDR3)
  - Scheda di rete Intel 10Gbit ????
  - SO Ubuntu Linux 16.04 LTS, kernel version 4.14
- **PC3 (backends)**
  - Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
    - core : 4(+4)
    - cache size: 8192 kb
  - Ram size 32 GB (4x8 DDR3)
  - Scheda di rete Intel 10Gbit ????
  - SO Ubuntu Linux 16.04 LTS, kernel version 4.14

6.1.2 Schema

As we have written in the last section, the network topology is distributed on three desktop pc that acts as tester machine in order to isolate the load balancer's process during the execution of the tests in order to get information about the CPU's usage.
The picture below represents the network topology chosen for the tests with all network addresses. This topology can change on the number of the backend's (number of namespaces in the PC3) for some tests.

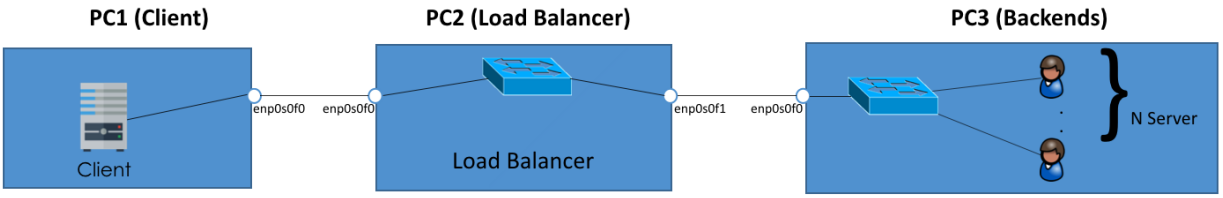

Figure 6.1. Network Topology for the test

- **PC1** acts as client.
  In fact,this machine hosts the tool used for testing the load balancer. This tool sends requests with destinaion IP, the Virtual IP.
  This machine has one physical network interface that joins to the PC2 by a physical Ethernet cable.

  - **enp0s0f0** ip 192.168.0.10/24

- **PC2** acts as load balancer.
  In fact, this machine hosts the load balancer that gets in ingress the traffic and makes load balancing thought several backends.
  This machine has two physical network interfaces that join the PC3 and PC1 by a physical Ethernet cable.
  Furthermore, there is a Linux-bridge that joins both two interfaces. This bridge has two IP addresses, the first acts as the default gateway for backend's network while the second one is the Virtual IP address.

  - **enp0s0f0** physical interface
  - **enp0s0f1** physical interface
  - **bridge** with 2 ip:
    * interface b 10.0.0.254/24 default gateway
    * interface b 192.168.0.110/24 vip

- **PC3** in this machine there are the backends (servers) that will receive and process the requests from PC1. Th number of backends depends on the particular test.
  Each server (backend) is in its own Linux Network Namespace , in which a server is running and listening on the unique interface in that network namespace .
  Each network namespace joins to the root network namespace by a veth (virtual ethernet) pair ; one end is set into root namepace joined to the linux-bridge while the other end is set in the network namespace created. The last interface has an ip assigned.
  This machine has one physical network interface that joins to the PC2 by a physical Ethernet cable. Also this physical interface is joined to the bridge.

  - **enp0s0f0** physical interface
  - **bridge** that joins N interfaces
    * interface **vethX**
  - **network namespace nsX** in which
    * interface **vethX_** ip 10.0.0.x/24

6.2 Tests

For the validation of the Load Balancer's prototype we have studied several use's cases to analyze the following parameters:

- **Transaction rate** represents the number of server hits for second that the load balancer can support during the test;
- **Response Time** represents the average time the load balancer took to respond to each simulated user's requests;
- **Longest Response Time** the maximum time waited for a single request;
- **Concurrency** allows you to set the concurrent number of simulated users that send requests to the load balancer;
- **Failed Transaction** is the number of failed transactions during the test;
- **Throughput** is the average number of bytes transferred every second from the server to all the simulated users;
- **CPU monitoring** the CPU usage during the test;

For the execution of all test , we have used three tools :

- **Siege**[1] to get specific results for testing the load balancer;
- **iPerf3**[2] to measure the throughput;
- **htop**[3] to monitor the resources on the PC2 that hosts the load balancer;

6.2.1 Test use's cases

We have divided the tests into two sections, in order to test our load balancer with the most important commercial product and with its direct competitor.
The first section is about testing our solution with the commercial products such as HAProxy, LVS and IPTables.
In particular, the first group of tests is under the siege's tool in order to get information in terms of response time, transaction rate ecc.
In this scenario, we have studied four test cases differing for two particular parameters: concurrency of simulated connections and the duration of the test.
The second group of tests is about testing and comparing our prototype with its direct competiors that use its same tecnology (eBPF ecc). These tests are done in Kubernetes's

[1] Siege, linux tool discussed in the detail in the section 3.5

[2] iPerf3, linux tool to test networking, discussed in the detail in the section 3.5

[3] htop, linux tool for monitoring system's resources, discussed in the detail in the section 3.5

framework.
The test's cases are the following :

- **TEST 1.1:** using *siege* with concurrency=10 and time=10s;
- **TEST 1.2:** using *siege* with concurrency=200 and time=10s;

The second section of the tests is different from the previous one, in particular, here, we test our prototype with its direct competitors: Cilium[4] and Calico's the load balancer. Furthermore, these two tests are executed in a different topology showed previously, in fact, they are run in Kubernetes's framework that will be one of the most useful frameworks where the load balancer will be used.

- **TEST 2.1:** using *siege* with concurrency=200 and time=10s;
- **TEST 2.2:** using *iperf3* to measure the throughput;

6.3 Results

In this section, we present several bar-charts that will explain better the performance of the load balancer discussed in this thesis.

6.3.1 Results Test 1

[4]Cilium framework, discussed in section 3.6.2

Results Test 1.1

In this test we focused our attention to analyze the transactions rate (max, min, medium), monitoring the CPU's usage, at the same time.
Test 1.1 Concurrent = 10 and time for test is 10 seconds.

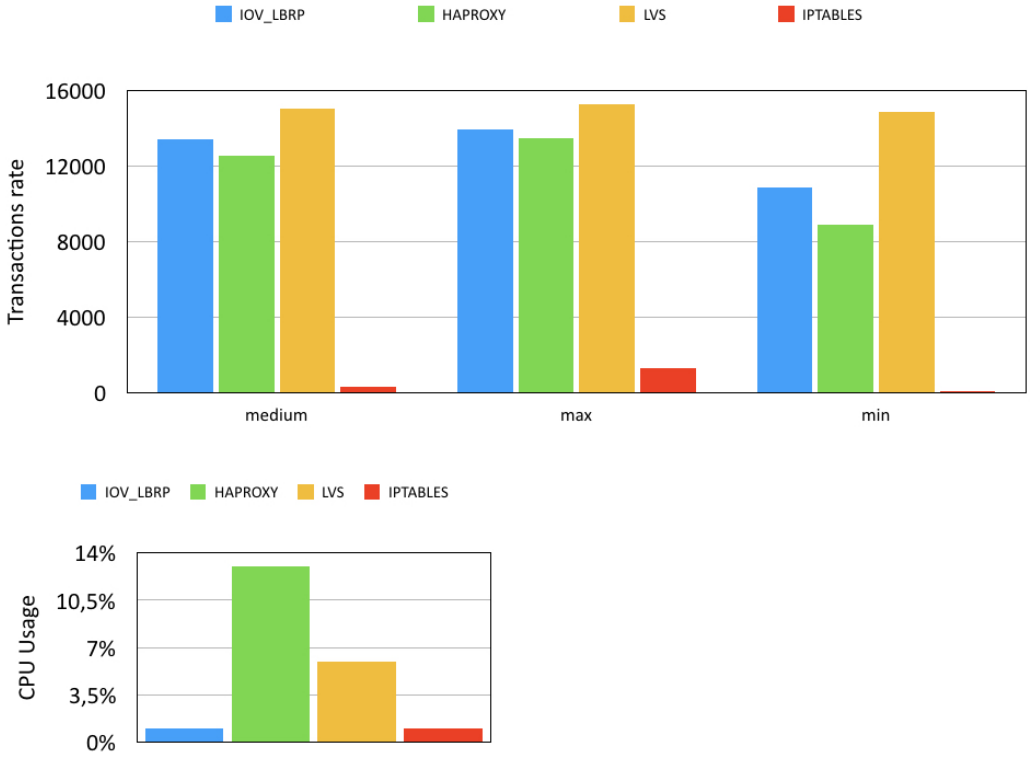

Figure 6.2. Results of Test 1.1

Analyzing the *Transaction rate chart*, our load balancer has high performance. In fact, it has a transaction rate slightly less than LVS but greater than HAproxy, in the medium case.
However, these performances remain relevant for our prototype. The worst load balancer is IPTables.
Analyzing the *CPU's usage chart*, our load balancer uses only 1% of the CPU's system such as IPTables, while LVS uses about the 5% and HAProxy bit less than 14% (about 13% more than our load balancer).

Results Test 1.2

In this test we focused our attention on analyzing the transactions rate (max, min, medium), monitoring the CPU's usage, at the same time.
Test 1.2 Concurrent = 200 and time for test is 10 seconds.

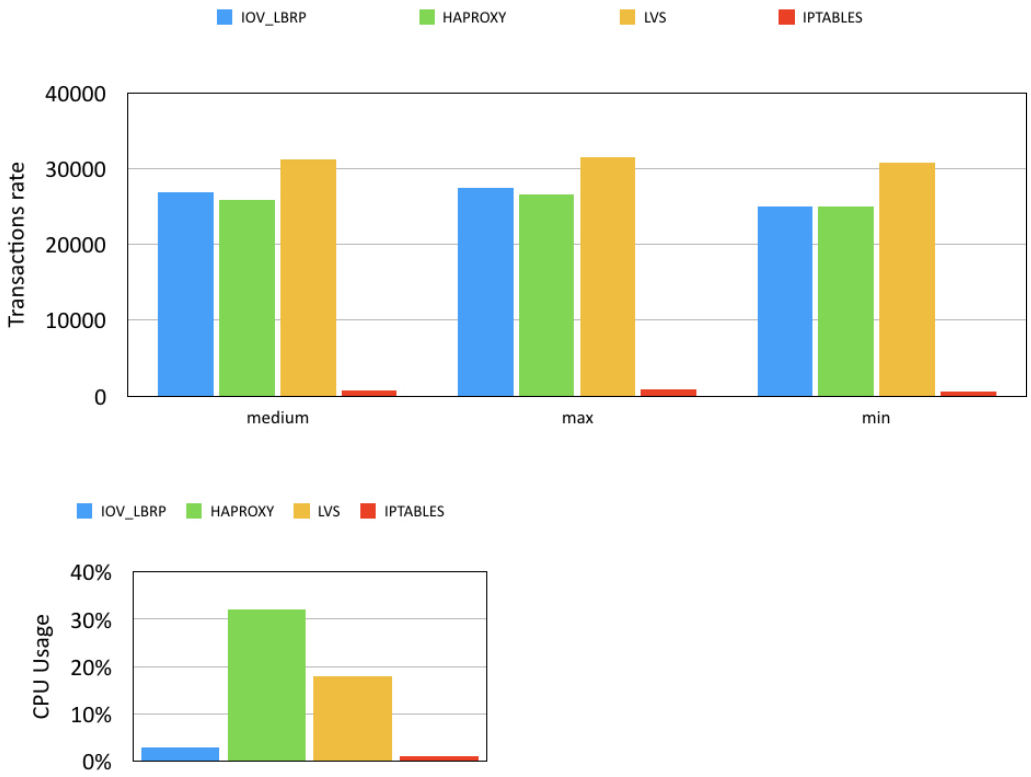

Figure 6.3. Results of Test 1.2

Analyzing the *Transaction rate chart*, our load balancer has a transaction rate slightly more than HAProxy, but less than LVS, that remains the most performing. The worst remains IPTables.
Analyzing the *CPU's usage chart*, our load balancer uses only 4% of the CPU's system, while LVS uses about the 18% (about the 14% more than our prototype) and HAProxy bit less than 35% (about 31% more than our load balancer).

So, analyzing these bar-charts, we can deduce that the prototype of the load balancer, proposed in this thesis, has the same top performances in terms of transaction's rate and throughput, such as its competitors HAPROXY and LVS.
But, the most relevant data is that our load balancer uses a very small amount of CPU for processing the requests, against its competitors that use more and more CPU, especially

HAPROXY.

6.3.2 Results Test Kubernetes

In order to understand the results previously got, we have compared our load balancer with its direct competitor in a real scenario in the *Kubernetes framework*. We compare three different Network CNI working on Kubernetes:

- **Iovnet** descibed in the section 3.4 ;
- **Cilium** described in the section 3.6.2;
- **Calico** [5] provides secure network connectivity for containers and virtual machine workloads;

Result Test 2.1

In this test we focused our attention on analyzing the medium transactions rate, for each load balancer, in two test's cases: client to client and client to service.
Test 2.1 Concurrent = 200 and time for test is 10 seconds.

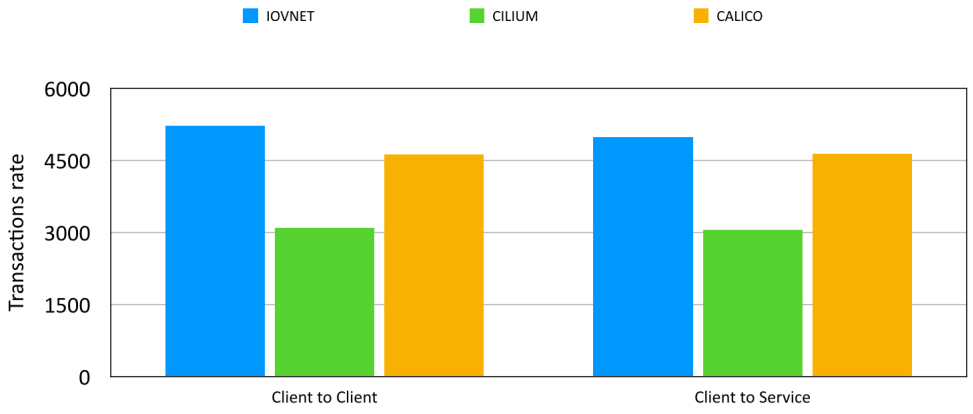

Figure 6.4. Results of Test 2.1 Transactions rate

As we can see from the bar-chart above, the Iovnet Load Balancer has the highest transaction rate between these three competitors, in both use's cases: client to client and client to service).
Our prototype is about 65% faster than the Cilium's one in the Client to Client scenario and 58% in the Client to Service's one.

[5] Calico CNI is one of the most used networking plugin for Kubernetes [28]

Result Test 2.2

In this test we focused our attention on analyzing the throughput in two test's cases: client to client and client to service.
Test 2.2 Throughput

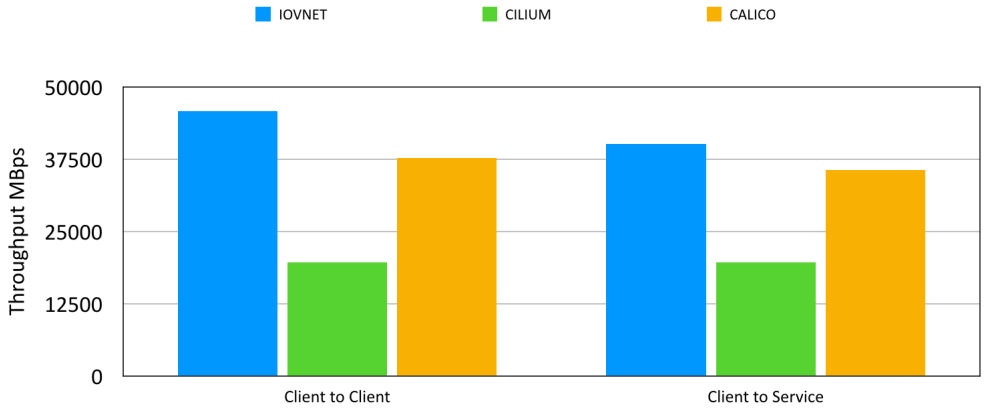

Figure 6.5. Results of Test 2.2 Throughput

As we can see from the bar-chart above, the Iovnet Load Balancer has the highest throughput between these three competitors, in both use's cases: client to client and client to service). Our prototype has a 2x throughput against the Cilium's one.

Chapter 7

Conclusions and Future work

7.1 Conclusions

The first conclusion we can reach is that it can be possible to provide an high-performance load balancer using the new eBPF's technologies.
As showed from the last chapter, the load balancer proposed in this thesis respects the goals proposed in the second chapther in order to have a load balancer with high performance but, at the same time, minimizes the CPU's usage.
The final features of this load balancer are:

- **Generic Load Balancer** it can be configured in order to work in every situations, starting from load balancing the traffic between the network namespaces to the traffic of a real data-center;
- **High Performance** it ensures the high performance in the process of load balancing;
- **CPU**'s usage is guaranteed to be minimized;
- **Session Affinity** ensuring that a client is always served by the same backend, in order to keep the same session (especially for the TCP traffic);
- **CLI** (Command Line Interface) that is very useful to set up the configuration of the load balancer;

So, we have implemented a load balancer that can compete with the actual load balancer in commerce, by minimizing the CPU's usage using the new eBPF's technology.

7.2 Future Work

From the first prototype of the load balancer, we have obtained very interesting results in terms of high performance and CPU's usage against its direct competitors.
In the next months, we are going to update the code that performs the load balancer in

order to improve its features and performance.

We want extend this prototype with the following features:

- **High Availability's support** in order to increase the availability of the load balancer. So, in this scenario, if the master load balancer will go down, there will be a second load balancer that will replace it maintaining the same configuration and the same states of the previous connections;
- **Code's optimitation**
  We want optimize the load balancer's code, especially in the data-path in order to minimize the number of instructions that the load balancer performs each time the packets coming in;

Finally, for Kubernetes scenario, we want to implement a load balancer that will be put, no more inside the node, but outside it in order to load balance all requests arriving into the cluster to several nodes.
All of this, would give an optimization in the paths that the packets follows in the cluster.

Bibliography

[1] NFV, *Network Function Virtualization: State-of-the-art and Research Challenges* , Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, Raouf Boutaba, 2015.

[2] SDN, *Software-Defined Networking: A Comprehensive Survey* , Diego Kreutz, Member, IEEE, Fernando M. V. Ramos, Member, IEEE, Paulo Verissimo, Fellow, IEEE, Christian Esteve Rothenberg, Member, IEEE, Siamak Azodolmolky, Senior Member, IEEE, and Steve Uhlig, Member, IEEE .

[3] BPF, *The BSD Packet Filter: A New Architecture for User-level Packet Capture* , Steven McCanne, Van Jacobson, 1992.

[4] libpcap, *libpcap, Lawrence Berkeley Laboratory, Berkeley, CA* , V Jacobson, C Leres, S McCanne - Initial public release June, 1994.

[5] WinPcap : una libreria Open Source per l'analisi della rete, *Lawrence Berkeley Laboratory, Berkeley, CA* , Loris Degioanni, Mario Baldi, Fulvio Risso, Gianluca Varenni, 2003.

[6] BPF, URL: https://lwn.net/Articles/437884/.

[7] Linux Traffic Control, URL: http://tldp.org/HOWTO/Traffic-Control-HOWTO, Martin A. Brown.

[8] bcc, URL: https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md, IOVisor Community.

[9] IO Visor Project, *Hover Framework: IOModules manager*, URL: https://github.com/iovisor/iomodules.

[10] BPF Compiler Collection , URL: https://github.com/iovisor/bcc, IOVisor Community.

[11] BPF Compiler Collection mail, URL: https://lkml.org/lkml/2015/4/14/232, Ingo Monar, 2015.

[12] IO Visor Project , URL: https://www.iovisor.org/, IOVisor Community.

[13] Kubernetes, URL: https://kubernetes.io, Google,2014 .

[14] IOVnet CNI, URL: https://github.com/netgroup-polito/iovnet, Polytecnic of Turin, 2017.

[15] Consistent Hashing, *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*, David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy, 1997.

[16] Fisher-Yates Shuffle, *Statistical tables for biological, agricultural and medical research*, Fisher, Ronald A., Yates, Frank ,1938.

[17] YANG Data Model, *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)* , Björklund, Martin, 2010.
[18] Computing the Internet Checksum, URL: https://tools.ietf.org/html/rfc1071, Braden, Borman, Partridge, 1988.
[19] Incremental Updating of the Internet Checksum, URL: https://tools.ietf.org/html/rfc1141, Mallory, Kullberg, 1990.
[20] Computation of the Internet Checksum via Incremental Update, URL: https://tools.ietf.org/html/rfc1624, Rijsinghani, 1994.
[21] Siege, URL: https://linux.die.net/man/1/siege, Jeffrey, Fulmer, 2004.
[22] iPerf3, URL: https://iperf.fr/, Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, Kaustubh Prabhu.
[23] htop, URL: https://hisham.hm/htop/index.php?page=author,Hisham Muhammad , some developers.
[24] HAProxy, *HAProxy-the reliable, high-performance TCP/HTTP load balancer* ,Tarreau, Willy, 2012.
[25] Linux Virtual Server, URL: http://www.linuxvirtualserver.org/,Li Wang, 2004.
[26] Netfilter and IPtables, *Netfilter and IPTables – A Structural Examination* , Alan Jones, 2004.
[27] Cilium Project, URL: https://cilium.io/.
[28] Calico Project, URL: https://www.projectcalico.org/.