

Network Virtualization with Docker

Jesús Ávila Sumariva

**An analyses of how and why implement a virtual network like
the final project for the Electrical engineering department from
ISEP.**

Supervisor: Dr. Miguel João Miguel Queirós Magno Leitão

Porto, July 11, 2019

Dedictory

To my family who always cheer on me to continue my way. To Alejandro, who always help me during my personal and professional path.

Abstract

This document started with the **pedagogical purpose** to help the students of this center about the understanding of networks and their working. For this we will show how to virtualize a network, why it's useful and how we get it. We will start with the debate about Virtual machines or containers. After that we will introduce the tools we'll use, and will be exposed a little guide of use to help at the scope of the tools. The most important tool will be **Docker** because of is the most complex and with more possibilities of the tools but will be introduced Iptables, bash scripting and, in some how, other possibilities to simulate virtual networks in our own computer between inner network configuration in Linux environment.

After the introductions will be exposed two different typologies, one with different interconnected networks and other known in this document like "The Unique Network", where our system just will have one network as big as we could create.

With this examples is easy to amplify to more complex system, even mix them. Mainly the work in this point is the docker deployment, It will be exposed two different way, in each topology, to deploy it, with docker-compose and a Bash-script, to understand better what's happening and how to modify in case of being needed.

And in the last point we will explain the inner implementation in each host, the possibilities and the configurations to get the result we want and a deep look to the important points here.

At the end will be attached an appendix with a proposed exercises to the student explaining partially the solutions and with some proposed questions about the network working.

Keywords: Docker, network, network simulation, Iptables, docker, network deployment, pedagogy

Resumo

Este trabalho surgiu com o propósito pedagógico de facilitar a realização de experiências práticas sobre redes de dados em ambiente laboratorial. Para isso foram estudadas metodologias de virtualização de uma rede, de forma a possibilitar a realização de trabalhos experimentais sobre redes complexas simuladas em apenas um computador. Para este efeito, foram analisadas e discutidas alternativas de implementação de máquinas virtuais e de contentores. Para a implementação deste trabalho foi selecionado o Docker como ferramenta de virtualização. Neste documento são também apresentadas outras ferramentas necessárias à realização deste projeto como: Docker-compose, Iptables e Bash.

Como demonstração da exequibilidade da metodologia proposta, são apresentadas duas tipologias de rede distintas. Para cada uma destas redes, são apresentadas e explicadas duas metodologias de implementação, uma através de um script Bash e outra através da ferramenta docker-compose.

Um possível guia de trabalho laboratorial suportado pela metodologia proposta foi também preparado e é apresentado em anexo. A resolução passo a passo dos exercícios propostos no guia encontra-se também disponibilizada neste documento.

Abstract

La idea de éste trabajo nace con un propósito pedagógico para facilitar la realización de experimentos prácticas sobre redes de datos en un ambiente controlado y accesible para equipos no especialmente potentes. Para ello fueron estudiado los posibles métodos de virtualización de redes, posibilidades de realización redes de relativa complejidad en un sólo equipo. Para la implementación de esta tarea fue designado 'Docker' como herramienta de virtualización, tanto de equipos como de redes. En este documento serán presentadas otras herramientas, paralelas a Docker, tales como: Docker-compose, Iptable y Bash.

Como demostración de la viabilidad de la metodología propuesta, serán presentadas dos tipos de redes distintas. Cada una con su correspondiente implementación, la cuál se llevará a cabo de dos formas distintas, usando un Script en Bash, equivalente a comandos en terminal, y usando un archivo docker-compose.yml

Un posible guión de trabajo será propuesto al final del documento en forma de anexo, teniendo en el 'Anexo A' un problema propuesto y su solución mediante métodos formales, utilizando el propio docker-compose. Y en el 'Anexo B' un guión dirigido a explícitamente a un estudiante con preguntas propuestas como ejercicios. **Keywords:** Docker, network,

network simulation, Iptables, docker, network deployment, pedagogy

Acknowledgement

All those good friends who help me in this each time I needed, for the shared way and the not. Remember specially at my flatmates in "rua cedofeita".

Contents

List of Figures	xvii
------------------------	-------------

List of Source Code	xvii
----------------------------	-------------

1 Introduction	1
1.1 Introduction	1
1.2 Goal	2
1.3 Thesis Structure	3
Chapter 2	3
Chapter 3	3
Chapter 4	3
Chapter 5	3
2 Virtualization	5
2.1 Virtual Machines	5
2.1.1 Advantages	6
2.1.2 Disadvantages	6
2.2 Containers	7
2.3 Conclusions	9
3 Tools and environment deploy	11
3.1 BASH Scripting	11
3.1.1 Streams	11
3.1.2 Variables	12
3.2 Summary	12
3.3 IPTABLES	13
3.3.1 Filter Table	13
3.3.2 NAT table	14
3.3.3 Mangle	15
3.3.4 Raw (Non-default)	15
3.3.5 Security (Non-default)	15
3.3.6 Summary	15
3.4 Docker	17
3.4.1 Advantages	17
3.4.2 Disadvantage	18
3.4.3 Conclusions	18
3.4.4 Dockerfile(Download and create Image)	18
Building the image	20
3.4.5 Docker-compose (Environment)	21
Docker-compose, Commands	22
3.4.6 Networks	24

Bridge	24
Host	25
Overlay	25
none	25
Macvlan	26
Conclusions	26
3.4.7 Docker Commands	27
Containers Commands	27
Docker Network Commands	28
3.4.8 Dockerfile commands	29
Docker-compose commands	29
4 Typologies and deployment	31
4.1 Topology I: Networks Interconnection	32
4.1.1 Networks	32
4.1.2 Simulated Hosts	32
4.1.3 Dockercompose	33
Customizing Networks properties	37
4.1.4 BASH Script	38
4.1.5 Benefits Drawbacks	39
4.2 Topology II: The Unique network	40
4.2.1 Network	40
Isolating default bridge in a corner	41
Simulate Network: The Unique Net	41
Docker-compose	44
BASH	45
5 Network Interconnection	47
5.1 Topology I: Network interconnections	47
5.1.1 Bash Script	50
Notes	50
5.2 Topology II: Unique Network, Firewall Restrictions	51
5.2.1 Docker-compose	52
Iptables	55
host1	55
host3	55
5.2.2 Routing	55
host1	55
host3	56
host2	56
Docker-compose.yml	57
5.2.3 Bash Script	58
5.3 Results and conclusions	59
Bibliography	61
A Deploy for a student	63
A.1 Deployment	63
A.1.1 Networks	64
A.1.2 Machines	64

host1	64
host2	64
host3	65
host4	65
A.1.3 Docker-compose	66
A.2 Solution	67
A.2.1 Machines	67
host1	67
host2	68
host3	68
host4	69
A.2.2 Docker-Compose system working	69
B Script for the student	71
B.1 0. Before starting	71
B.2 Connect host1 with host3	72
B.2.1 Solutions	73
1. Enable ip_forward in host2	73
2. Assign an IP in eth0 and modify the route table in host1	74
3. Assign an IP in eth0 and modify the route table in host3	74
B.2.2 Questions:	75
B.2.3 Proposed exercises	75
Connect host2 with host4	75
Connect host1 with host4	75
C Docker0 Interface	77

List of Figures

2.1	Virtual Machine	6
2.2	Virtual Machine Kernel	7
2.3	Virtual Machine Kernel	8
3.1	Shell landscape	12
3.2	Shell landscape	14
3.3	Iptables-diagram	15
3.4	Docker Structure	17
3.5	Installing Image	20
3.6	Available images in docker after install our customized image	21
3.7	docker-compose deploy and status	22
3.8	docker-compose deploy up	23
3.9	Bridge network driver, [17]	25
3.10	Macvlan network driver, [17]	26
4.1	Basic network interconnections.	32
4.2	ifconfig-host1	35
4.3	ifconfig-host2, notice both interfaces	35
4.4	ifconfig-host3	36
4.5	Scheme of the deployment	40
4.6	docker0 interface after changing	41
4.7	ifconfig answer	43
4.8	Scheme of the deployment II	44
5.1	Basic network interconnections.	47
5.2	routing table from host1, practically same that for the others containers.	48
5.3	Possible system to assemble	51
5.4	figure	52
5.5	figure	54
5.6	figure	54
5.7	figure	54
A.1	Exercise	63
A.2	Exercise	68
A.3	Exercise	68
A.4	Exercise	68
A.5	Exercise	69
B.1	Exercise	72

Chapter 1

Introduction

1.1 Introduction

The point to treat and to solve here, in this document is, using a virtual network, create a secure environment to teach how networks work and how we can configure them to a customized working. Here we present the possibilities, problems and partial solutions. We want to solve the physics and resources limitations in the real world virtualizing a network as much realistic as possible.

For this task we will use an environment based in Docker but I would like introduce other "rawer" options (without direct software implementation) like presentation of the problem and explain what are the important points, the different solutions we propose and the steps for the developing.

Why a virtual network?

Mainly for rising accessibility at the content to practices, being not necessary a physical network with several machines available, it's a useful way to design networks, develop changes and to understand as well. But we can use virtualization for security, scalability, redundancy or easiness of implementation[16]. Other interesting point of view in favor of network virtualization is be able to apply model and structures more advance or modern in our local environment (for decrease security risk for example), because the popularity and grow of the networks in the daily life of everyone made it as necessarily big that barely changed in the last 30 years, and this is a good point to implement an own network treat in your local network, for example with advanced options in Iptables.

The pedagogic scope of this article is better than I thought when I started due to we can experiment with Iptables even use this deployment like a pentesting lab, we could configure at low level one machine like a router with (or without) a NAT for example and test remote attacks.

- At the beginning our purpose is to simulate a network with the lest amount of resource possible, to have the possibility of rise the complexity.
- Develop two different kind of topology, that will work like base for any (or at least a lot of options) implementation we could thought, and go deeper in cons, pros and restrictions.
- For all of that a presentation of some tools we need to know and manage, all the information we need about that could be find in this report.

For this task we need to explain the concept of "virtualization", all his possibilities and the scope of each of them.

For finishing the introduction I'm going to specify the specification in the software that I used for the project:

- **docker version:**

```
Server: Docker Engine - Community
Engine:
Version: 18.09.3
API version: 1.39 (minimum version 1.12)
Go version: go1.10.8
Git commit: 774a1f4
Built: Thu Feb 28 05:59:55 2019
OS/Arch: linux/amd64
Experimental: false
Client:
Version: 18.09.3
API version: 1.39
Go version: go1.10.8
Git commit: 774a1f4
Built: Thu Feb 28 06:34:04 2019
OS/Arch: linux/amd64
Experimental: false
```

```
Server: Docker Engine - Community
Engine:
Version: 18.09.3
API version: 1.39 (minimum version 1.12)
Go version: go1.10.8
Git commit: 774a1f4
Built: Thu Feb 28 05:59:55 2019
OS/Arch: linux/amd64
Experimental: false
```

- **OS:** *Linux debian 4.9.0-8-amd64 1 SMP Debian 4.9.144-3.1 (2019-02-19) x86_64 GNU/Linux*

1.2 Goal

This document started with a pedagogical purpose to help students of this center to understand networks and how they work. For this we will show how to virtualize a network, why it's useful and why we use Docker.. We will focus in some concepts mainly trying to accomplish each feature and in case we won't be able define where we find the problem.

The concepts we will have in our mind during the deployment are:

- **Network deployment:** The working network creation in a computer.
- **Host configuration:** Think about the important points we have to consider in each machine.
- **Routing protocol:** We will study how to implement routing options in the host inside and use some of them like router properly.
- **Network planning:** The student will try to deploy a custom network for adapting to different situations.

1.3 Thesis Structure

Chapter 2

Here we will discuss differences between VM and containers, how they work generally and which is more useful for us. It's interesting knowing the differences between both point of views, advantages and disadvantages.

Chapter 3

Here we will have a brief presentation of the tools we need and how to use it generally and in the particular way for our purpose. Mainly speak about:

- Iptables
- Bash-scripting
- Docker and all his implications (like docker-compose)

Explaining all the applications we will use later in the deployment.

Chapter 4

In this part we build two different topologies focus in the container deployment, we could say implement the base for an infinity number of possibilities. Just implementing the diagram showed and not the functionalities for clarify the evolution and the path we have to follow for deploying the environment. This chapter correspond mainly to the 'Network deployment'.

Chapter 5

Here it's when we take the topologies from chapter 4 and made it works with the features we want, interconnecting networks, avoiding the connection, creating routing path or in case we could customize the network. This chapter correspond to the 'Host Configuration'.

Chapter 2

Virtualization

Virtualization is a concept introduced around the 1960's by IBM, with the main propose of sharing physical computer resources when they were expensive and limited. A common current use is running different O.S. over other one, not directly over the physical components, with all the advantage (and disadvantage) that It could bring us. [23]

Currently with this concept we are able to virtualize[22]

- Hardware: We can create a virtual machine that act like a real one, using virtual components like HDD, RAM or even the Ethernet interface. Technically is as easy like reserve some resources form the physical Hardware and dedicate it for and specific task, for example to support a virtual desktop.
- Virtual Desktop: Is the concept of separate the logical desktop from the physical machine, creating the virtual hardware necessary for its working.
- Containerization: Is one of the possibilities for an OS virtualization based in the feature that the kernel allows the existence of multiple isolated "user-space" instance. We will explain it better after.

Our objective at the moment is create (virtualize) several host (with Ethernet interface) and assemble a sandbox environment to allow us experiment directly with hosts, trying to be an useful tool to clarify doubts about the working in the network.

So, now first decision will be how virtualize several host in our machine.

2.1 Virtual Machines

Virtual Machine is the complete emulation (virtualization) of hardware for running a isolated machine, one of the most important and relevant point is that isolated machine is limited by the virtual hardware that we offer to the virtualization. Like inconveniences we can point that the virtualization have the waste resources, not to much but unavoidable and that all the resources assigned to it will be reserved full-time for the virtualization, being not important if it is using it or not. [24].

Hypervisor Software is used in every Virtual Machine and is executed like an application on the host operative system (hosted hypervisor) or rest directly over the physical machine (bare-metal hypervisor) and manages the resources assigned by the host system. The hypervisor software creates an abstraction layer between physical hardware and virtual machines. The

VM run isolated from the host system and from other virtual systems so process executed in a virtual machine don't affect o the host system.

So we can say that the hypervisor(VM Monitor) is who manage how to share the physical resources without need initialized a OS to control it.

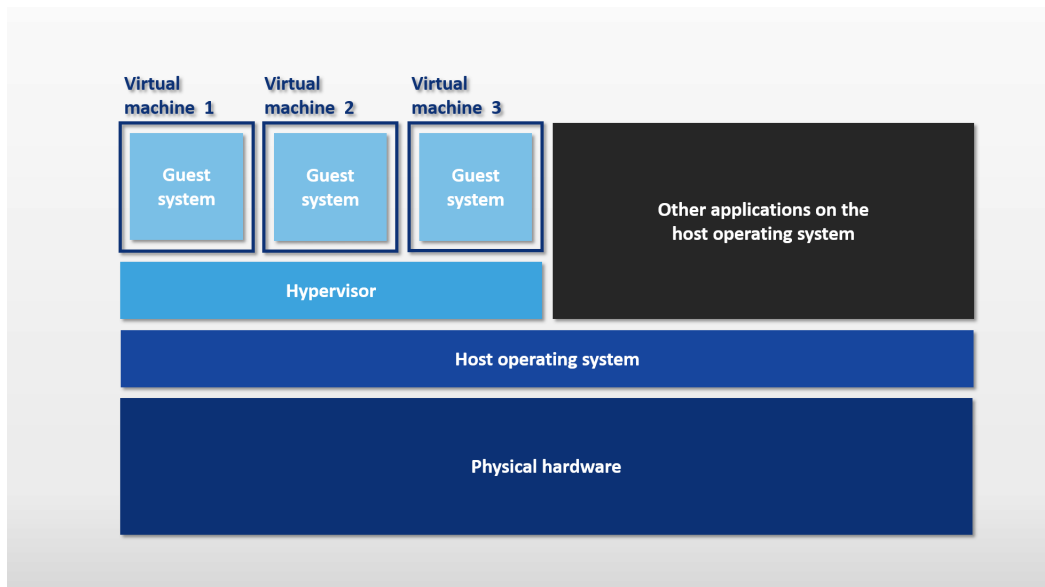


Figure 2.1: Inner Schedule VM, Layer where to find Hypervisor, [1]

2.1.1 Advantages

- Hypervision-based virtualization allow several operative systems on the same hardware base
- This method (hypervisor) offer emulation functions with which are solve incompatibility problems between system architecture.
- We can have totally isolated machines running just with one machine working.

2.1.2 Disadvantages

- A VM is less efficient that physical machines because of the functions for solve the incompatibilities of the architecture systems.
- Speaking about security level, if an attacker attack **directly** over the virtualization software every VM are compromised.
- The resources to run a VM are to be enough to run the OS in a physical host, so you need a lot of resource to have one VM. If we want to run for example 10 VM we could have a problem in a standard computer.
- The resources are reserved if the virtual machine is using them or not.

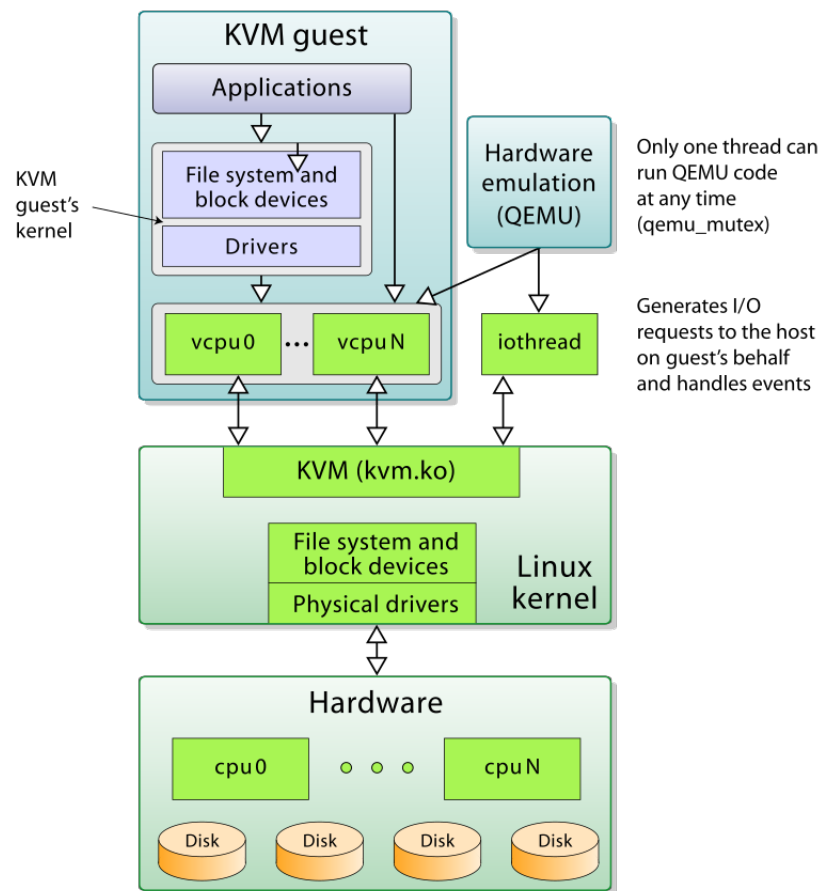


Figure 2.2: Hosted hypervisor. A virtual machine using the physic elements from the host system (CPU time, RAM, disk space, and peripherals) to each VM. KVM it's technology built in Linux to allow use Hypervisor to create guest machines. "Kernel-based Virtual Machine"[28]

2.2 Containers

Containers, also known like Virtualization in the OS level, like the VMs, is a virtualization method, but much more lightweight. With this virtualization method the kernel of the containers the host machine are the same, it is not created an abstraction layer between software and hardware, being allowed the existence of several isolated instances. Containers are not a new concept but free-software project like Docker made it really popular because the advantages and because is a free-source project.

The idea was created in 1979, when UNIX start to use the **chroot**[20] system call, that allow user change the root folder temporarily. After that with the years would appear the concept of "jail", that using the chroot (file system isolation) and expanding it will isolate the set of users and the networking system as well.

A container can be considered an application where all his information is found in a image and due to is charged in the machine a image where can be found all the information that the system need to run and after that the application is deployed in a virtual environment(For avoiding install the application in the own OS besides other features).[21] [25]

Like we saw VM works using "hypervisor" running on the top of the physical layer, containers, by other hand, run in the user space on top of the Operative system kernel of the host machine. Each container will have its isolated user space being able to run several containers, in the same machine.[5]

The **isolation** between containers is achieved using two features in the Linux kernel.

Namespaces: Is a Global System resource implemented in Linux that is able to label a namespace to a process where the changes made by a program in a determined Namespace are just noticed by other softwares in the same space. This point is the fundamental aspect of *containers*. The Linux namespace was inspired by the Plan 9 from Bells Labs, treating each process like a file with enabled permissions and with not access to other part of the memory that itself space.

The linux kernel provide 6 types of namespaces:

- **mnt**
- **pid**
- **net**
- **ipc**
- **uts**
- **user**

[14]

Control group: Is the other important tool integrated in the Linux Kernel to manage the resource, as CPU, disk, networks and RAM, used by a single application, being used to assign how much will be use by this. Is the way that the Linux Kernel control the Containers, resources limit, prioritization and control[13]

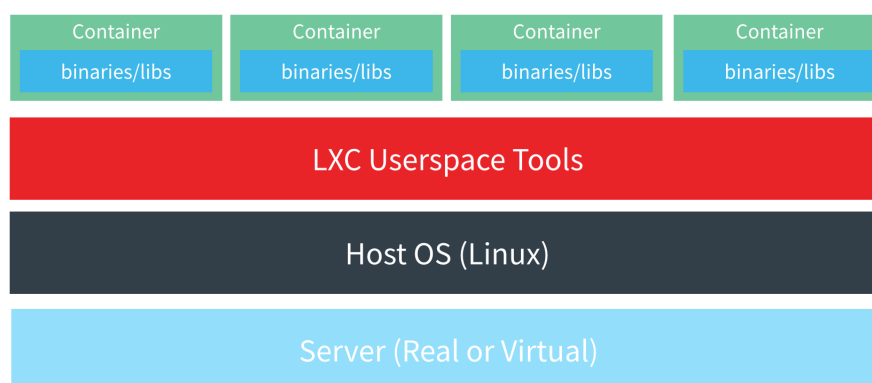


Figure 2.3: Inner Schedule VM Kernel, [29]

Note: For a host machine running Linux is not possible run a windows container because there is not support offered by the Linux kernel.

2.3 Conclusions

Like we can see VM and containers are and are not the same thing, some aspect has the same effect practically but other are really different, as in result as in the implementation. Mainly for our network simulation we just isolate the hosts, we could have a really good implementation with virtual machines (maybe even better), but not perfect neither, and due to the limited resources we have available the best option for be able to deploy more complex environment pass through use containers seizing the inner network manager that *Docker* offer.[3].

In particular we will use Docker like container manager but we will deep dive in the next chapter in the tool introduction.

So mainly our propose is know how docker manage the networks and use for get the environment we would like.

Chapter 3

Tools and environment deploy

Let's do a brief introduction to the tools we needed for the deployment. I would like you to think of this chapter as a command summary, written in order to understand each step given in the implementation. I have tried make it as clear as possible, although with the Docker-compose, far from being difficult, we found that a particular context was necessary, and so we chose to add it directly in the implementation itself.

3.1 BASH Scripting

Starting from the scratch Bash is a Unix Shell language, commonly found as the default terminal interpreter in Linux systems. In it, It's possible execute commands directly in the terminal or create a file (given a list of instructions—Script—) that will be executed. One of the most important feature of this language is that is compatible with almost all the other shell languages.

*Note: Just to clarify that when symbol

1 \$

appear will mean that the line is being introduced directly into the terminal (without writing '\$'), and when the symbol don't appear will mean that is in a written file.

3.1.1 Streams

The streams are the processes that we can observe in the black windows, executing commands or scripts. Each type of stream has a numeration to be designed in inner operations. I shall expose the cases with the respective examples.

- **0.** stdin, referred for the inputs that the terminal receive.
- **1.** stdout, referred at the answer that the terminal show after the correct execution of a command.
- **2.** stderr, is a category into the 'outputs' but to point some error, for example doing

1 \$ ls

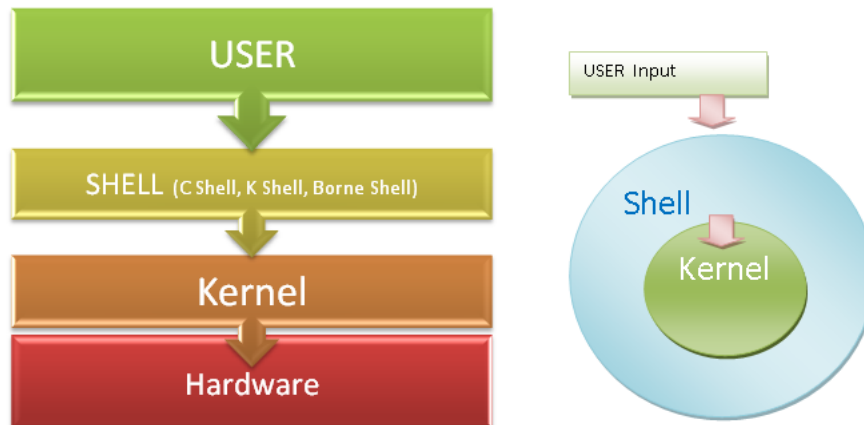


Figure 3.1: Shell landscape [15]

to a non-existent file

We can use the streams for detecting error or successes operations and act in consequences. The reason why we do not treat the errors in the script explained above is dual: first, because it would complicate the code and its understanding. But like tool is really useful to know that we can use this possibility in our projects.

3.1.2 Variables

Like in others programming languages we need use variables to save a value, for using is has easy like

```
1 $ example=ISEP2019
2 $ echo $example
3 ISEP2019
```

This way we can create variables and invoke it to use its value. For give a new value to the variable is as easy like to redefine the variable with other value. *Notice that the first dollar symbol is to be interpreted that we are in the command line, you mustn't to write it for the correct execution.

3.2 Summary

We would like to think to the written as an introduction of how a Shell language works, and to learn a little about the scope. In here (In the general project) the code applied is really raw, having simple instructions mainly from Docker and sometimes showing errors, that obviously won't affect to the result.

Finishing the introduction to BASH, we would like just to say that the file where you write your script needs execution permissions for obvious reasons. To add it to the file we need to put in a terminal:

```
1 $ chmod u+x filename.sh
```

[4]

As stated above, this is only a basic introduction to the matter, but we think it is useful to knowing what we can do. Not much of this will have further use, but it might be helpful in any case.

3.3 IPTABLES

Iptables is user-space utility built on top of netfilter. Its mechanism consists on applying a restriction table, acting as a firewall. It's include in most Linux's distributions by default. It compares all the operation through the network layer with a set of rules and, if some especial treatment for an operation is to be found, then it applies.

The rules are designed like "chains" and there are three tables by default, each one of them with different applications.

3.3.1 Filter Table

This table, is used for packet filtering, being the most commonly used, and its work is divided (by default) in three different kind of chains:

- **INPUT:** It manages all the packets that are addressed to the machine.
- **OUTPUT:** It manages all the packet that go out from the machine.
- **FORWARD:** It manages all the packets that pass through the machine but weren't created by the machine or the machine isn't the last destiny.

By default Iptables work in this table, and we can check the information of the table just write in the terminal (With the proper privileges)

```
1 $ iptables -t filter -L
```

To create news chains we have to follow the next structure:

```
1 $iptables -A INPUT -s 192.168.0.0/24 -j DROP #1  
2 or  
3 $iptables -I INPUT 192.168.0.55 -j ACCEPT #2
```

(1.) where **-A** (add) create a new chain at the end of the table, **INPUT** refer to the kind of packet, could be **OUTPUT** or **FORWARD** as well. **-s** refer to source, "when the packet

```

root@osboxes:~# iptables -t filter -L
Chain INPUT (policy ACCEPT)
target     prot opt source                destination

Chain FORWARD (policy ACCEPT)
target     prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source                destination
root@osboxes:~#

```

Figure 3.2: Example of showing an empty filter table

comes from..." and `-j DROP` it's to determinate the action, our options could be ACCEPT, DROP and REJECT. In this case we are dropping every packet from "192.168.0.0/24", a full network.

(2.) The main different is `-I`, this create a new chain at the beginning of the table, in this case will ACCEPT every packet from a particular host. It should be noted that the host is in the previous net, but, since we are creating the chain as the first one, the subsequent restriction will not be affected.

It should be noted that the order of the tables are fundamental; it shall not be a problem in our implementations, but it can be a possible focus of further problems in case we forget about this.

We will need basically the MAC restriction orders, for that the commands is:

```
1 iptables -A INPUT -m mac --mac-source F1:42:A3:B3:C2:03 -j REJECT
```

¹ just changing the *mac-source* value and the action (`-j REJECT`) we would like to execute. I may spoil you that that will be, if not the unique restriction, at least the main.

3.3.2 NAT table

A NAT device is used for address translation, is able to change the source IP and the target IP, thus "separating" the connection in independent networks. The NAT table adds two new chains.

- **PREROUTING:** Manage the packets that are going out from the host
- **POSTROUTING:** Manage the packets that are going in to the host

```
1 $ iptables -t nat -L
```

We will not explore this point in a deeper approach, because it is not applicable in our scope. But it could be interesting to create, or simulate, different networks, and to communicate them through the NAT table, in some intermediate host.

¹Notice that the mac-source is a arbitrary MAC address just for the example

3.3.3 Mangle

- TOS: It is used to set and/or changing the type of Service field in the packet, it is rather useful for improving the interactive traffic,
- TTL: It is used to modify the TTL (Time to Live) of the packets. It can be used, for example, for detecting multiple computers connected to the same session.
- MARK: Used to put a customized mark in the packet that iproute2 will be able to recognize to treat it in a different way if it's necessary, for example in a bandwidth limit.

```
1 $ iptables -t filter -L
```

3.3.4 Raw (Non-default)

It's a table to treat the packets before that the Kernel could define an state for the packet (Output, Input....)

3.3.5 Security (Non-default)

This table is used to set internal security context marks on the packets. This rules may be applied on packets or in connections.

The same happen here that with the NAT table, it is not applicable to the scope of this project but it could be easily be an implementation for understanding and use Iptables in our, soon to be implemented, networks.

3.3.6 Summary

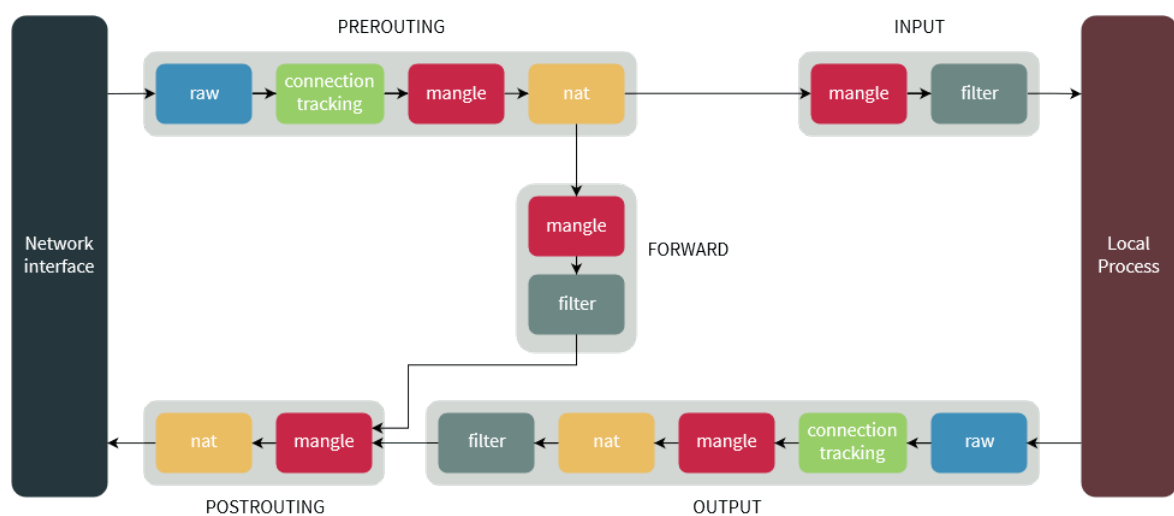


Figure 3.3: Iptables-diagram [26]

All the information here were acquired from this sources. [2] [7]

3.4 Docker

I would like to clarify that the Docker works with containers, but not all containers use Docker; however, it is the software implementation we will be using, and it is also the most popular currently for managing containers. We already know how the isolation is achieved, but, how are the containers executed? Docker uses a structure client-server, in which the docker client communicates with the Docker daemon (server) by means of a command line, which enables access to the docker registry.

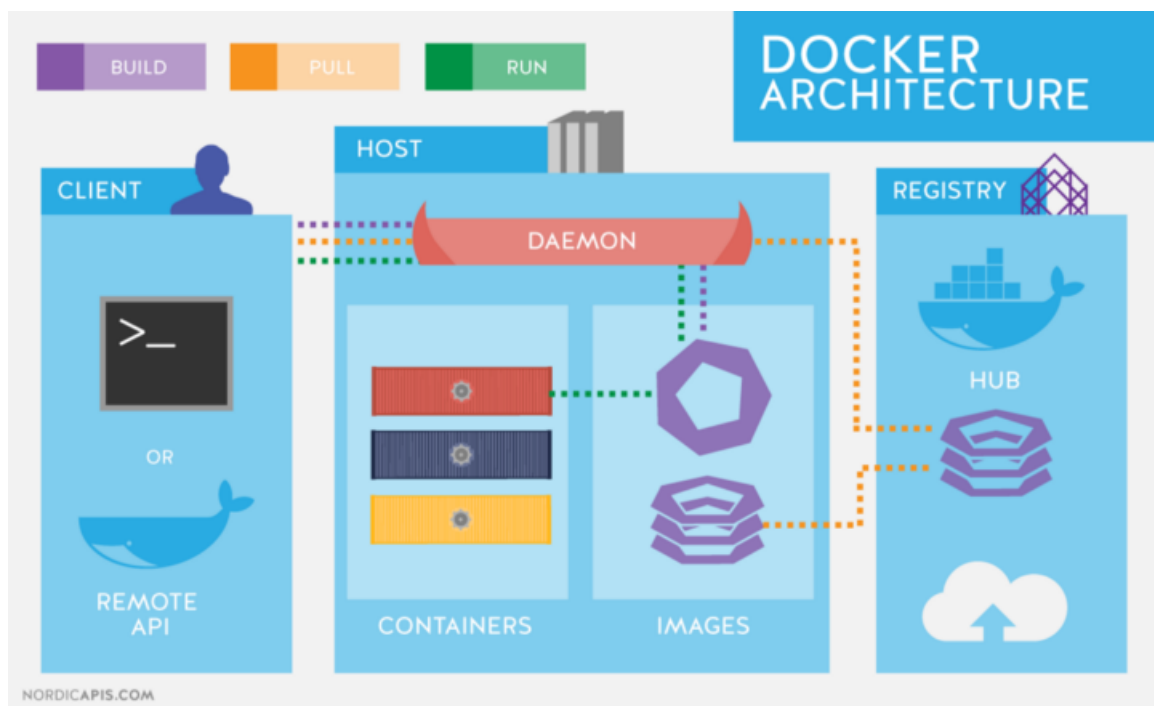


Figure 3.4: Docker Structure client-server[12]

Like we see in the picture Docker takes a file (an image) and create a running container (application) with the proper management to isolate it.

3.4.1 Advantages

- Quick set-up and deploy thanks to the fact that it has all the necessary information in a file (the image).
- Standardization of environments, which makes it depend solely of one file.
- Virtualization losses are much less that with physical virtualization.
- And mainly isolation

3.4.2 Disadvantage

The price we need to pay in order to get these advantages is our own security. Since the kernel used by the containers is the system host, if one container is compromised, as minimum, all the other can be compromised as well, in the worst case scenario the host system too. It is not a problem that, since this deployment is not really dangerous due to it being in a controlled environment.

Other problems about containers but that are not a inconvenient for this implementation, are:

- Don't bare-metal speed
- Graphical application don't work really good

[27]

3.4.3 Conclusions

For this project, we only have available a normal resource host to deploy and to experiment with the networks, so our option is Docker. It will be mainly for resources because with more possibilities, is easier to rise the complexity. [6]

After reminding that the containers are created based on an image let's start then creating our own image by default and how to customize it.

3.4.4 Dockerfile(Download and create Image)

With Docker we have the possibility to download a image from Docker Store but what happens when any image there has the requisites we need? Is then that we use Dockerfile to custom our image.

Docker allow us to create custom images using the information in a document (Dockerfile), since it is able to define the commands that an user could execute in a terminal windows. The main steps are:

- Create the Dockerfile and define to build up your image
- Define the steps to build up the image
- Build the image using the file
- Now you can create a container using the image

To create own customized image we can use one that exist already in the Docker Store as the basic of the structure. One must download a image from github or creating your own with Docker Register (Complex proceeding). We are choosing the first option because it is more than enough for our purposes.

Thus, we define the the custom steps in the Dockerfile and Docker will create an image using the file. Let us now explain the main parts that compose a Dockerfile.

- **FROM** is for define a base image: every Dockerfile must start with FROM instruction. It can appear more that once in the same Dockerfile document to build several images up. We have the option to define "FROM SCRATCH" that is an explicit empty image. Let's use for this example fedora that will be useful later too.

```
1 FROM fedora:23
```

- **ADD packets that we could need**, this part depend on the different command of each O.S. for download the packets, in fedora we must to use

```
1 FROM fedora
2
3 MAINTAINER 1180226 1180226@isep.ipp.pt || jazz.sumariva@gmail.com
4
5 RUN yum -y install net-tools #Tools that we need in the own
   container
6 RUN yum -y install iputils #Tools that we need in the own container
7 RUN yum install nc #it could be useful.
8 RUN yum -y install iptables-services
9
10 RUN yum -y update
11 RUN yum clean all
```

but in Debian we should use the "apt-get install..."

- **COPY**(*Not Used*) copy new files or directories from <src> (the host machine) to <dest>(Referred to the absolute path or the relative "WORKDIR").

```
1 COPY hom* /mydir/ # adds all files starting with "hom"
2 COPY hom?.txt /mydir/ # ? is replaced with any single character ,
   e.g., "home.txt"
```

- **LABEL:**(*Not Used*) The purpose of this command is add metadatas in the image. It is very useful as additional information.

```
1 LABEL maintainer "1180226 1180226@isep.ipp.pt || jazz.sumariva@gmail
   .com"
```

- **ENV** It allows to us environment variables, it could be useful to define a path for example

```
1 ENV myName="John Doe" myDog=Rex\ The\ Dog \
2 myCat=fluffy
3
```

Building the image

After have our Dockerfile with all the necessary steps

```

1 #Dockerfile
2 FROM fedora
3 LABEL maintainer 1180226 1180226@isep.ipp.pt || jazz.sumariva@gmail.
   com #Author
4
5 RUN yum -y install net-tools
6 RUN yum -y install iputils
7 RUN yum -y install nc
8 RUN yum -y install iptables-services
9
10
11 RUN yum -y update
12 RUN yum clean all
13
14 CMD bash

```

Netcat it is installed but not used, I thought that anyway could be interesting have it in the container in case to dive deep in the possibilities of Iptables.

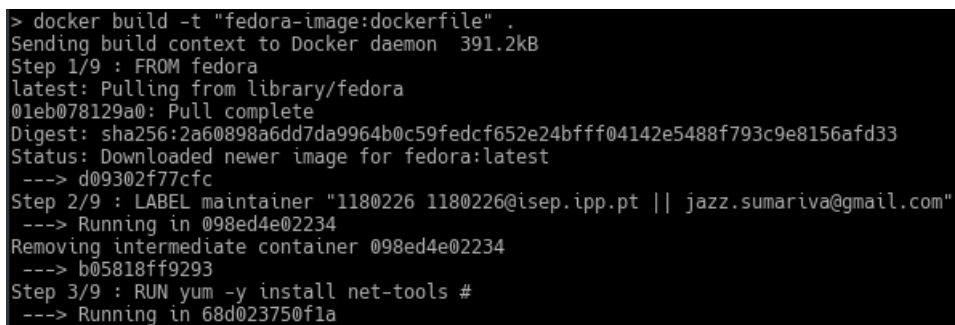
We just need to put a terminal session (with properties privileges) in the folder where our Dockerfile is found and write.

Notice that the '.' at the end of the command represent the path where the terminal is working, it's going to check in the current directory if exist the Dockerfile and use that one to build our image

```

1 $ docker build -t "fedora-image:dockerfile" .

```



```

> docker build -t "fedora-image:dockerfile" .
Sending build context to Docker daemon 391.2kB
Step 1/9 : FROM fedora
latest: Pulling from library/fedora
01eb078129a0: Pull complete
Digest: sha256:2a60898a6dd7da9964b0c59fedcf652e24bfff04142e5488f793c9e8156afd33
Status: Downloaded newer image for fedora:latest
----> d09302f77cfc
Step 2/9 : LABEL maintainer "1180226 1180226@isep.ipp.pt || jazz.sumariva@gmail.com"
----> Running in 098ed4e02234
Removing intermediate container 098ed4e02234
----> b05818ff9293
Step 3/9 : RUN yum -y install net-tools #
----> Running in 68d023750f1a

```

Figure 3.5: This should look executing *build*

This command (with the proper dockerfile, will take several minutes to accomplish the download, at the end the weight of the image should be around 1.05GB.

At this point, if you need to list the images that you have in your Docker just need to write in the terminal

```
1 $ docker image ls
```

```
> docker image ls
REPOSITORY      TAG                IMAGE ID           CREATED            SIZE
fedora-image    dockerfile        264f388c41cf      About a minute ago 1.05GB
fedora          latest            d09302f77cfc      2 months ago      275MB
```

Figure 3.6: Available images in docker after install our customized image

It should be noted that in the image that there are two containers with different TAGS, the one with the "dockerfile" tag in the one we define, with the updates and the extra packets installed, and the "latest" tag is the image that we use like base.

Then after the download from Docker repositories the base image and execute the updates we will have our available image already. [10] [11]

3.4.5 Docker-compose (Environment)

With a docker-compose.yml we can deploy an customized environment just using the configuration on this file. An alternative way to deploy and environment It is using a script in BASH (for example), and use directly the docker's commands, but that method is weak enough to consider other ones. I will do the examples in both cases just to clarify doubts and have the possibility to choose.²

- The first step is **to install docker-compose**, the interpreter of the configuration file, for this we can just write in the terminal:

```
1 $ sudo curl -L https://github.com/docker/compose/releases/download
  /1.18.0/docker-compose-$(uname -s)-$(uname -m) -o /usr/local/bin/
  docker-compose
```

- After the installation we need to give execution privileges to the application

```
1 sudo chmod +x /usr/local/bin/docker-compose
```

- We can now check the docker-compose version we have

```
1 $docker-compose --version
2 docker-compose version 1.18.0, build 8dd22a9
```

[18]

Now We can create our docker-compose.yml, Being here explained until the knowledge we need for our purpose:

²In fact Docker-compose is the weakest of the formal deploy methods, for bigger projects are used Kubernetes or Swang.

```

1 host1:
2   image: fedora-image:dockerfile
3
4 host2:
5   image: fedora-image:dockerfile

```

Thus we can deploy the system writing:

```

1 $ docker-compose up -d

```

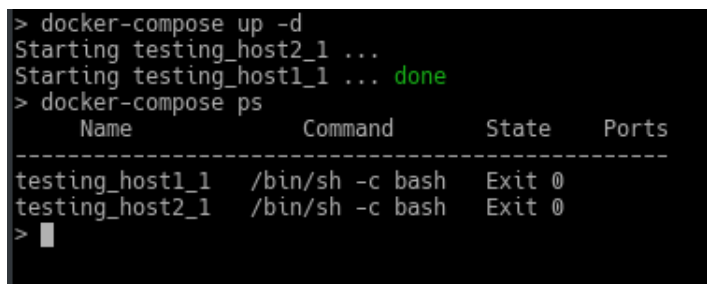
`-d` referred to detached mode, it is for avoiding execute both sessions in the same terminal this would return us an error.

Here we are executing two containers, with the names `host1` and `host2` respectively, using the image of `fedora-image:dockerfile`, the one we create upper but notice that the container was just created, it's not started, if we would like to see how is the **status of the hosts** just write:

```

1 $ docker-compose ps

```



```

> docker-compose up -d
Starting testing_host2_1 ...
Starting testing_host1_1 ... done
> docker-compose ps

```

Name	Command	State	Ports
testing_host1_1	/bin/sh -c bash	Exit 0	
testing_host2_1	/bin/sh -c bash	Exit 0	

```

>

```

Figure 3.7: docker-compose deploy and status

Here we can see that both containers are in state *Exit 0*, that means that everything run correct, at least technically. We need interactive session with them, we will solve this problem during the deployment of our system, every step will be correctly explained later, it is much more useful go deeper in the docker-compose structure with the proper context.

Docker-compose, Commands

To execute this commands, the Shell session have to be in the folder where is the file `docker-compose.yml`:

- For executing the environment developed in the docker-compose file in detached mode:

```

1 $ docker-compose up -d

```

For seeing the status of the system deployed:

- ```
1 $docker-compose ps
```

- For stopping the environment deployed previously by docker-compose:

```
1 $ docker-compose stop
```

Other example more complete

```
1 host1:
2 container_name: host1
3 image: fedora-image:dockerfile
4 stdin_open: true #For have an interactive terminal
5 tty: true #
6
7
8
9 host2:
10 container_name: host2
11 image: fedora-image:dockerfile
12 #build: .
13 stdin_open: true
14 tty: true
```

We will add more attributes during the develop but this container deployment offer, not only start container but also define the inner configuration of them, we will exploit the options as we need it and showing with example the structure that we have to follow in the document *docker-compose.yml*.

Here I comment the line "*#build: .*" it could be use to build the image directly with the docker-compose from the "dockerfile". You have to choose do or built or image tag but not both. If you want to build the image with docker-compose you can add the directory where to find the dockerfile or put the dockerfile in the same folder that docker-compose.yml, in the last case you just have to add '.' in the directory to refer at the "local-folder".

```
> docker-compose up -d
Recreating testing_host2_1 ...
Recreating testing_host1_1 ... done
> docker-compose ps
Name Command State Ports

host1 /bin/sh -c bash Up
host2 /bin/sh -c bash Up
> █
```

Figure 3.8: docker-compose deploy up

Here we can check that the state of our containers are up, in this situation we could interact with them using the command

```
1 $ docker attach host1
2 $ docker attach host2
```

In different windows to execute the container session in each of that, notice that the name is reference enough to indicate it. We will go deeper after to understand how to use containers.

### 3.4.6 Networks

One of the main advantages about using Docker is the network management with the containers, and how much invisible it is, so adaptable. Docker offer the possibility of using several kind of drivers each one with particular features.

#### Bridge

The default network driver, if you don't specify a driver, this one will be assigned automatically to the network. A bridge network, in terms of networking, can be considered a Link Layer device. Docker use this kind of networks for allowing the communication between the containers connected to the same bridge network, isolating the containers in the outside. The Docker bridge driver automatically apply rules in the host machine (Iptables) for avoiding the communication between different bridge networks but by default use the host machine like bridge for connecting to "Internet".

*We can find differences between the default and an user-defined bridge:*

- An User defined bridge offer better isolation between applications and better inner connection. Containers connected in the same user-defined bridge automatically expose all ports to each other containers in the same network unlike to the default-bridge, that each container have to specify what Port will be exposed with the *-publish* flag.
- User define bridge provide automatic DNS resolution, thus the containers can communicate using the IP or the alias (name). Unlike is the default bridge network, where you need used the *-link* flag to be able to communicate containers by the alias and (like extra information) share the environment variables of both containers.

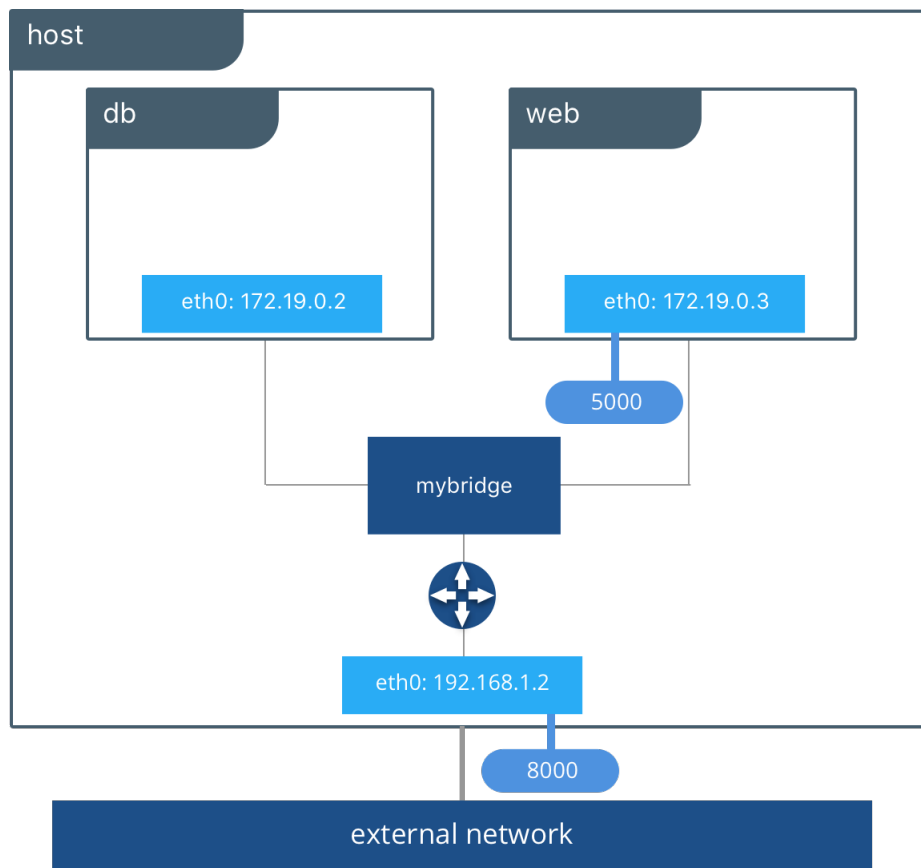


Figure 3.9: Bridge network driver, [17]

## Host

The host network driver in a container will share all the host machine network interfaces, if you **expose** a port in a container running in a "host network" you are able to access by the address of the host machine as well, if ports are not exposed then this has not effect. This option just works on Linux host systems.

## Overlay

*not used*

Connect several Docker daemons together and allow swarm services to communicate or use it also to allow communication between a swarm-service<sup>3</sup> and a standalone container<sup>4</sup>. It's a layer that sit on the top of (overlays) the host specific network.

## none

*not used*

This option disable all networking, even if you try to connect the container to a network

<sup>3</sup>Is a decentralized deployment of an environment where the containers are nodes of a bigger service

<sup>4</sup>The standard execution of containers, is the use we will use in this project mainly.

after rise it, you will not be able because of this network configuration. We won't use it here.

## Macvlan

Macvlan network assign a MAC address to each container "looking like" a physical device. It's useful when you are trying avoid damage your own network due to IP address exhaustion. You have to can activate the promiscuous mode in your host for assigning several MAC address to the same physical interface.

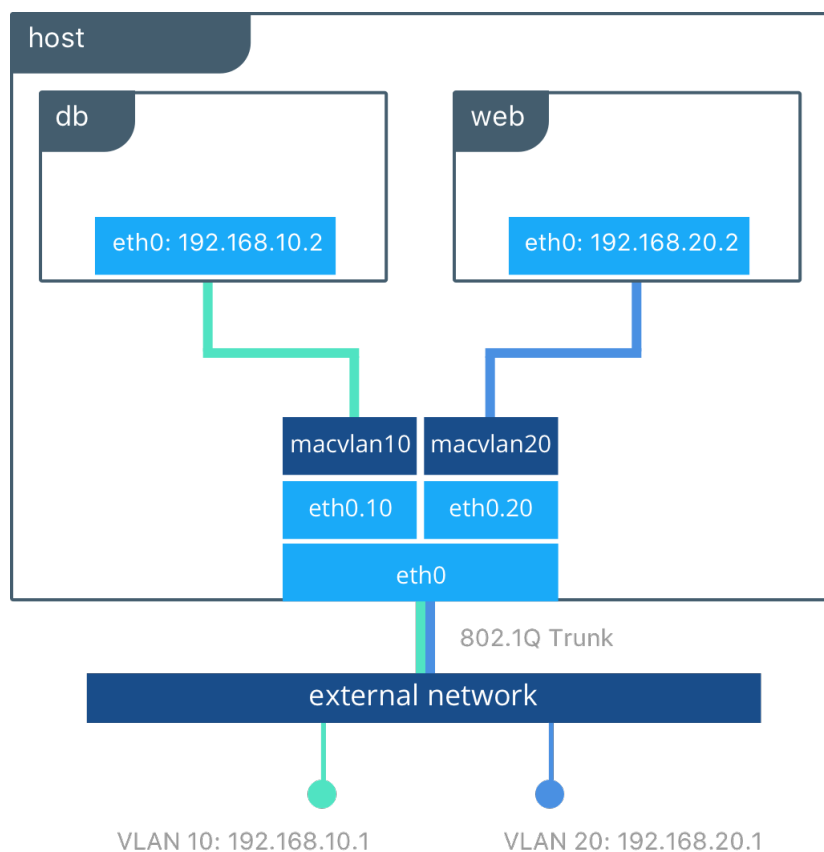


Figure 3.10: Macvlan network driver, [17]

## Conclusions

Mainly we'll work with the bridge and the macvlan drivers, both have interesting different but in our examples we will develop both pointing the different and results.

[17], [9]

Before skip to the other lesson I would like to clarify that Docker use innerly in its networks a virtual gateway, mainly we will define what IP will be assigned to the gateway to know how to avoid it. We want to ignore the existence of this gateway for three reasons: 1. to control the access 2. for avoid the dependency of this and have a clear route table, that the student will fill just with the virtual host information. 3. For avoiding in our possibilities the need-hood that the user of this application know

something about Docker and get a better simulation. This step will be defined in the chapter 5, how the professor have to deliver the context to the student.

### 3.4.7 Docker Commands

#### Containers Commands

- List all the images that docker have available

```
1 $ docker image ls
2
```

- Start a stopped container by the id-container or the name

```
1 $ docker start idcontainer_OR_name
2
```

- Check the existing containers

```
1 $ docker ps -a -q
2
```

- Create a new container using a image assigning a **name**, a **network**, **admin privileges** and a **interactive (-it) and detached<sup>5</sup> session** based in the image **fedora-image:dockerfile**.

```
1 $ docker run --name example_name --privileged=true --network
 existing_network -itd fedora-image:dockerfile
2
```

- Stop and remove a container designed by the id-container or the name

```
1 to stop one container
2 $ docker stop id_container
3
4 to remove one container (previously stopped)
5 $ docker rm id_container
6
7
```

- Stop and remove every container.

---

<sup>5</sup>In the background, to access at the container we need to use other command (attach

```
1 $ docker stop $(docker ps -a -q)
2
3 $ docker rm $(docker ps -a -q)
4
```

- Attach to an detached, and started container in the background.

```
1 $ docker attach idcontainer_OR_namecontainer
2
```

- It's possible execute a command in a started container without attach it with the command:

```
1 $ docker exec -d host1 mkdir new_folder
2
```

where *'-d'* it's for detached mode, *'host1'* it's the name of the running container and *'mkdir new\_folder'* it's the command that we want to execute in the container, in this case we create a new folder in the path where the container's terminal-session is working.

## Docker Network Commands

Here I will list the necessary commands and a brief explication about each.

- With that command you are **creating** a network using like **--driver** a macvlan, specifying the **IP range (--subnet), gateway and name**

```
1 $ docker network create -d macvlan --subnet=ip_red/mask --
2 gateway=ip_gateway nombre_red
```

- To list all existed docker networks

```
1 $ docker network ls
2
```

- Here we connect an existing container to a network previously created, this create a new Ethernet interface in the container.

```
1 $ docker network connect NetB host2
2
```

- To delete a existing docker network (delete default ones is not possible)

```
1 $ docker network rm networkname_OR_networkid
2
```

- Delete all existing networks (except the defaults)

```
1 $ docker network rm $(docker network ls)
2
```

### 3.4.8 Dockerfile commands

- For building an image

```
1 $ docker build -t "fedora-image:dockerfile" .
```

### Docker-compose commands

- Deploy the docker-compose environment in detach-**d** mode.

```
1 $ docker-compose up -d
2
```

- Show the deployment status of the docker-compose file

```
1 $ docker-compose ps
2
```

- Stop the docker-compose deployment

```
1 $ docker-compose stop
2
```





## Chapter 4

# Typologies and deployment

In this sections we are going to deploy two different topologies and implement them using the tools we described upper but focusing mainly in the docker working. How to deploy container and configure the networks. Here we will use the same image for each container in both topologies and for the rest of the document as well. So mainly the work of this section is use the **docker-composer** and the BASH script alternative to deploy a 'raw' environment, still without the knowledge enough (in the deployment) to use it for our final objective, mostly this chapter is the first step to understand the 'how' and 'why' of each step. The final implementation will have all the information here showed but in case someone need to use this document I thought was a good first step.

We work over the **network services** (be available or connected is enough for us at the moment), and **network planning** finding the limitations and deal with them.

### Dockerfile<sup>1</sup>

```

1 #Dockerfile
2 FROM fedora
3 LABEL maintainer 1180226 1180226@isep.ipp.pt || jazz.sumariva@gmail.com
 #Author
4
5 RUN yum -y update
6 RUN yum clean all
7
8 RUN yum -y install net-tools
9 RUN yum -y install iputils
10 RUN yum -y install nc
11 RUN yum -y install iptables-services
12
13 CMD bash #'Important, don't finish the application and can interact with
 it after'
```

*Remember to build the image and assign the name and don't forget the '.'*

```
1 $ docker build -t "fedora-image:dockerfile" .
```

---

<sup>1</sup>This image will be common for each topologies.

## 4.1 Topology I: Networks Interconnection

Let's start simulating the next scenario<sup>2</sup>

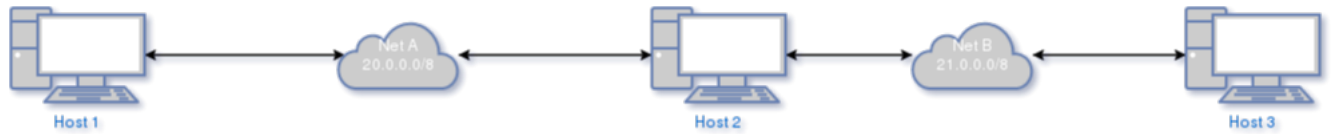


Figure 4.1: Basic network interconnections.

Here we have 3 machines where 2 of them have not a direct connection, let's explain how implement this topology and how we can use for our pedagogic environment.<sup>3</sup>

### 4.1.1 Networks

We start creating the custom networks we need in the docker space, for this example we need two different networks<sup>4</sup>:

```

1 $ docker network create -d bridge --subnet=20.0.0.0/8 --gateway
 =20.255.255.254 NetA
2 $ docker network create -d bridge --subnet=21.0.0.0/8 --gateway
 =21.255.255.254 NetB

```

Here we create the two custom networks that we need with:

- **-d macvlan**: The driver we're going use is a macvlan for assign different MAC address to each container.
- **--subnet** define the IP-range that our network will have, if you assign a range that will overlap with one existing already will warning an error in the network creation.
- **--gateway** usually is automatically assigned to the first available IP in the subnet, but like we don't want to work with it I assigned an arbitrary IP address to ignore it later.

### 4.1.2 Simulated Hosts

Let's run the three containers, each one in the proper network

- **Host1** <—> NetworkA
- NetworkA <—> **Host2** <—> NetworkB

<sup>2</sup>We need to remember that this scheme is the simulation result, how the student will perceive and not how innerly it's working, the inner working is explained in the Docker network section, I refer certainly to the virtual gateway that the networks in docker use, we will construct our system for avoiding the needhood of it.

<sup>3</sup>Our later target could be connect Host1 and Host3 through Host2

<sup>4</sup>We create networks with the driver bridge just like example, we could do it with macvlan following the same step and it's supposed that won't be a problem.

- NetworkB $\longleftrightarrow$ Host3

First we create the container with a **name**, **root privileges**, the proper **network** and run it with a **interactive detached session** using the image **fedora-image:dockerfile**<sup>5</sup>

```
1 $ docker run --name host1 --privileged=true --network NetA -itd
 fedora-image:dockerfile
```

We have to create the host2 and connect to both networks

```
1 $ docker run --name host2 --privileged=true --network NetA -itd
 fedora-image:dockerfile
2
3 $ docker network connect NetB host2
```

```
1 $ docker run --name host3 --privileged=true --network NetB -itd
 fedora-image:dockerfile
```

### 4.1.3 Dockercompose

**Remember that the networks NetA and NetB have to exist before execute the docker-compose**

After understand how is the process let's use an *Docker-compose.yml* to deploy it. Just repeat that the image has to be created already by the Dockerfile we don't need the Dockerfile anymore.

```
1 version: "3.5"
2 services:
3 host1:
4 container_name: host1
5 image: fedora-image:dockerfile
6 stdin_open: true
7 tty: true
8 privileged: true
9 networks:
10 - NetA
11
12 host2:
13 container_name: host2
14 image: fedora-image:dockerfile
15 stdin_open: true
16 tty: true
17 privileged: true
18 networks:
19 - NetA
```

<sup>5</sup>By default the images downloaded have the tag "latest" and the images built by a Dockerfile has the tag "dockerfile", if you go to the terminal a check you images with `[docker image ls]` you can see how what to use depend of you image always putting the structure "name:tag" being the default not necessary to apply

```
20 - NetB
21
22 host3:
23 container_name: host3
24 image: fedora-image:dockerfile
25 stdin_open: true
26 tty: true
27 privileged: true
28 networks:
29 - NetB
30
31 networks:
32 NetA:
33 external: true
34 NetB:
35 external: true
```

We added here the *privileged: true*, with this option the container can manipulate his own user-space like the host who created it, this will be useful for own propose.

And look about how the networks definitions are used, we can create our own networks in the docker-compose but I thought better use this way, the important point here is the structure, understanding it everything makes sense.

Now with a terminal session in the same directory that the file *docker-compose.yml*<sup>6</sup> we write to deploy:

If you look the docker-compose structure here you can check how each host has configured the network interface, ip-addresses and the 'external' in docker-compose will check the networks that innerly docker have already created, the default networks and the created. We could create our own network in the file, but I thought was more easy to understand the working in docker following this way. [19]

```
1 $ docker-compose up -d
```

Don't forget the *-d (detached)* mode for avoiding try to execute three different containers sessions in the same terminal, returning error obviously.

---

<sup>6</sup>Don't forget about the name of the file.

```
root > docker attach host1
[root@18744c9bbca2 /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 20.0.0.2 netmask 255.0.0.0 broadcast 20.255.255.255
 ether 02:42:14:00:00:02 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
 inet 127.0.0.1 netmask 255.0.0.0
 loop txqueuelen 1 (Local Loopback)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@18744c9bbca2 /]#
```

Figure 4.2: ifconfig-host1

```
root > docker attach host2
[root@bc8fd03d18ea /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 20.0.0.1 netmask 255.0.0.0 broadcast 20.255.255.255
 ether 02:42:14:00:00:01 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 21.0.0.1 netmask 255.0.0.0 broadcast 21.255.255.255
 ether 02:42:15:00:00:01 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
 inet 127.0.0.1 netmask 255.0.0.0
 loop txqueuelen 1 (Local Loopback)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@bc8fd03d18ea /]#
```

Figure 4.3: ifconfig-host2, notice both interfaces

```
root > docker attach host3
[root@9738f84374b4 /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 21.0.0.2 netmask 255.0.0.0 broadcast 21.255.255.255
 ether 02:42:15:00:00:02 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
 inet 127.0.0.1 netmask 255.0.0.0
 loop txqueuelen 1 (Local Loopback)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

[root@9738f84374b4 /]# █
```

Figure 4.4: ifconfig-host3

### Customizing Networks properties

Like we can see the IP's were automatically assigned by Docker, if we want assign them by the docker-compose (always inside the range of the proper network) we have to add to the network section the option `ipv4_address` in the next way:

```
1 version: "3.5"
2 services:
3 host1:
4 container_name: host1
5 image: fedora-image:dockerfile
6 stdin_open: true
7 tty: true
8 privileged: true
9 networks:
10 NetA:
11 ipv4_address: 20.1.1.1
12
13 host2:
14 container_name: host2
15 image: fedora-image:dockerfile
16 stdin_open: true
17 tty: true
18 privileged: true
19 networks:
20 NetA:
21 ipv4_address: 20.156.200.200
22 NetB:
23 ipv4_address: 21.123.252.96
24
25 host3:
26 container_name: host3
27 image: fedora-image:dockerfile
28 stdin_open: true
29 tty: true
30 privileged: true
31 networks:
32 NetB:
33 ipv4_address: 21.55.89.65
34
35 networks:
36 NetA:
37 external: true
38 NetB:
39 external: true
```

Look that the IP's assigned in the NetA respect the range that it accept, `20.0.0.0/8`. It's able to accept every IP which first 8 bits were '20', the same happen with the NetB, if you try assign an IP out of the range will jump a ERROR.

Note: The *version* in the docker-compose it's really important, in this case the 3.5 it's enough for our purpose but sometime you need to use previous version because of different deployment approaching.

Note2: We can customize the MAC address as well, we will do it further in the second implementation.

### 4.1.4 BASH Script

Let's deploy the same environment but using this time a Bash Script for clarifying the process and have other approach.

After create a file and assign executable permissions using:

```
1 chmod u+x script.sh
```

being the script.sh where will be the code after execution, in case that the terminal wasn't in the same directory you should put the complete path until the file. After that the script should look like

**Warning!:** This script delete *ALL* the docker network and containers started or stopped

```
1 #!/bin/bash
2
3 #Rise up Router topology.
4 #Stop and delete all the enable containers
5 docker stop $(docker ps -a -q)
6 docker rm $(docker ps -a -q)
7
8 #We delete all the networks
9 docker network rm $(docker network ls)
10
11 #We create the networks we need
12 docker network create -d bridge --subnet=20.0.0.0/8 --gateway
 =20.255.255.254 NetA
13
14 docker network create -d bridge --subnet=21.0.0.0/8 --gateway
 =21.255.255.254 NetB
15
16 #host1
17 docker run --name host1 --privileged=true --network NetA --ip
 20.123.54.98 -itd fedora-image:dockerfile
18
19 #host2
20 docker run --name host2 --privileged=true --network NetA --ip
 20.0.0.222 -itd fedora-image:dockerfile
21 docker network connect --ip 21.0.0.33 NetB host2
22
23 #host3
24 docker run --name host3 --privileged=true --network NetB --ip 21.0.0.56
 -itd fedora-image:dockerfile
```

You can just use the command that you need in your terminal, or custom the script to adapt any possible change.

The script execution will alert about several errors, for example if there is any container started, or any stopped the first two command will have warnings, the same happen with the default networks that can't be erased (at least with this method), it's a RAW version but it's working, it's enough and more easy to understand what's happening and how.

Like we can observe:



- 1. For avoiding name or IP already in use we need to be sure about the inner container situation, the most simple way it's just stop and delete each container and after that execute ours.
- 2. Not mandatory step if the network were created and are available, it's the same idea that upper.
- 3. The flag `-privileged=true` allows us to manage the user-space of the container lie the own host, exactly same that the docker-compose.

After that to look the status, in a humbler show that with docker-compose, we can write in our terminal session.

```
1 $ docker ps -a -q
```

For accessing to one of the started containers we can write

```
1 $ docker attach host1
```

and if you want to go out from the container (virtual host), just text

```
1 $ exit
```

And you will come back to you own host session and the virtual host will be stopped, if you need start again just

```
1 $ docker start host1
```

A good way to go out from a container without stop it is with the keys **Ctrl + p + q**, denominated like *escape sequence*, you don't need to start again the container, just put the session detached but still enable.

#### 4.1.5 Benefits Drawbacks

Mainly the problem of this deployment is the limitation at the moment to choose the IP address of the Host's, each networks is limited to start by an determined number (the 8 first bits of the IP), this could be confusing to the student in case he want to change to a different range the "wire simulation" lose the efficacy, we could keep fixed the IP of the middle machine to avoid this problem, then the student can change the IP but in a range to speak with the host2 in both cases.

The good points are that is an easy scenario to imagine and the middle host have 2 different networks and interfaces, if in the document that the student receive we insist about the virtualization limitations we had we can avoid that the student get confuse in this environment.

Still this environment it's not working, because it's just deployment, we need to refine the form to be applicable in the pedagogic field, several details that we need to correct like

the assigned IP, or the route table, we can modify it using a BASH script or even in the Docker-compose, but it's a point for the next chapter, the implementation of connections, the implementation of the environment it's the most important point here and lose in the *Nuance*<sup>7</sup> just could cause confusing if it's not explained enough.

## 4.2 Topology II: The Unique network

We are going now to experiment with several machines in the same network, one as big as we were able to create, for this implementation we can avoid partially the limitation of starting in a defined number like 20 or 21, ideally our range should be 0.0.0.0/0 but for obvious reasons in a not-isolated computer we have problems with that, so our first step it's create a network as big as we can. By default the bridge network that docker use the sub-network "172.17.0.0/16", so we should start from there like the nearest or we could try to change the default subnet:

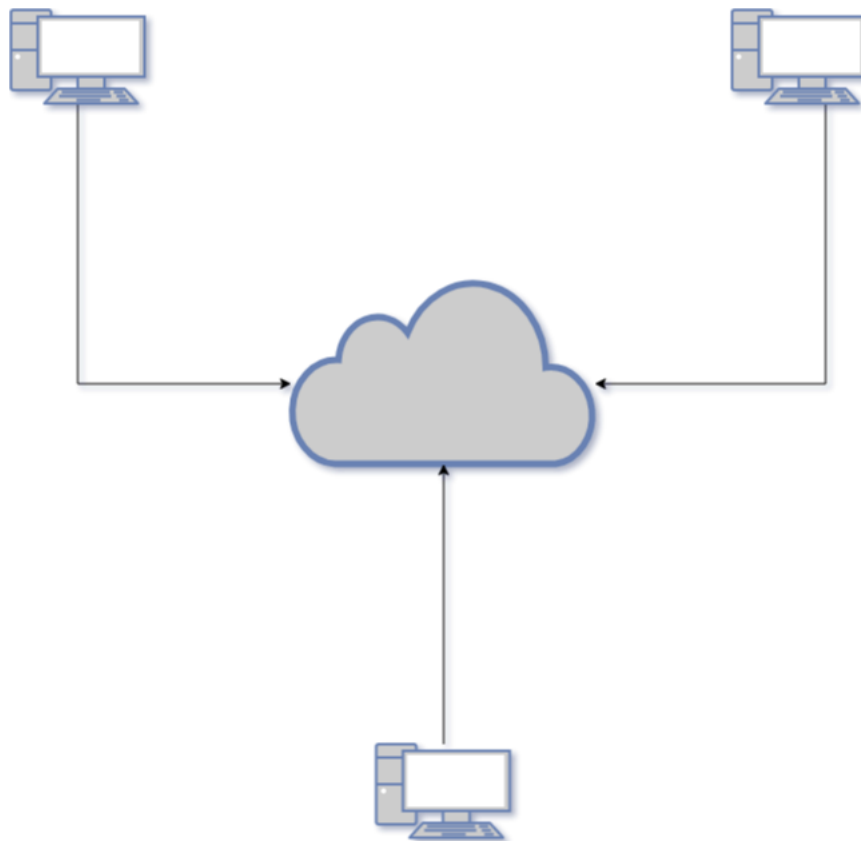


Figure 4.5: Scheme of the deployment

### 4.2.1 Network

Before of create the biggest network that we could we need to change the default values of the bridge network, we could use the default one like the main network for this purpose but

---

<sup>7</sup>French word

it would create several incompatibilities with the other possible typologies, so we are going to isolate the default network in the beginning (it could be in the end as well) and create other network that we could erase and create freely.

### Isolating default bridge in a corner

We are going to use a free code that "Kamermans" share by github, It's the easiest way to do it, I attach the script in the document to avoid possible offline services problems. Originally this script was created for the **version 1.10** but I test in the current version(**18.09.3** and it works.

<https://gist.github.com/kamermans/94b1c41086de0204750b>, You can check the code in the appendix C

Like in the script explain, we use directly the command in our terminal

```
1 curl -sS https://gist.github.com/kamermans/94b1c41086de0204750b/raw/configure_docker0.sh | sudo bash -s - Net_IP/Mask
```

Write in the Net\_IP and in the mask a useful range, as near as we could form the 0.0.0.0 and as near to the limit 255.255.255.255, for example the subnet **1.1.1.1/24**

```
1 curl -sS https://gist.github.com/kamermans/94b1c41086de0204750b/raw/configure_docker0.sh | sudo bash -s - 1.1.1.1/24
```

In my case this wasn't warning any error and you can check that every was right checking in the main host the ifconfig status and looking the docker0[8] interface<sup>8</sup>

```
> ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
 inet 1.1.1.1 netmask 255.255.255.0 broadcast 0.0.0.0
 ether 02:42:0f:c6:92:70 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure 4.6: docker0 interface after changing

### Simulate Network: The Unique Net

Now let's try to create a network, this time with macvlan driver to experiment with the MAC Address<sup>9</sup>. Let's erase every created network and let just the default ones.

<sup>8</sup>It's the virtual interface that docker use for the default bridge

<sup>9</sup>With the bridge driver each machine has a proper MAC address as well, but I decide use this one in this case just to have different examples

```
1 docker network rm $(docker network ls)
```

And now let's check how much big can be our network: The boundary in my computer, I'm not sure why or how, it's creating the a network with the *subnet: 128.0.0.0/2*

```
1 docker network create -d macvlan --subnet=128.0.0.0/2 UniqueNet
```

Being disposable from 128.0.0.2 until 191.255.255.254<sup>10</sup>device, nothing estrange but it's better think in that, this limitations in the range, makes sense if we look my ifconfig interface (In the host machine):

---

<sup>10</sup>We this big range I started to have problem with my internet connection, not with every website but with several of them, probably it's a misunderstanding coming form the network

```

> ifconfig
dm-958b29f700dc: flags=195<UP,BROADCAST,RUNNING,NOARP> mtu 1500
 inet6 fe80::4401:a2ff:feba:e168 prefixlen 64 scopeid 0x20<link>
 ether 46:01:a2:ba:e1:68 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 9 bytes 602 (602.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
 inet 1.1.1.1 netmask 255.255.255.0 broadcast 1.1.1.255
 ether 02:42:df:b8:25:84 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp3s0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
 ether fc:45:96:a7:0d:46 txqueuelen 1000 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 0 bytes 0 (0.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

ham0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1404
 inet 25.72.115.45 netmask 255.0.0.0 broadcast 25.255.255.255
 inet6 2620:9b::1948:732d prefixlen 96 scopeid 0x0<global>
 inet6 fe80::7879:19ff:fe48:732d prefixlen 64 scopeid 0x20<link>
 ether 7a:79:19:48:73:2d txqueuelen 1000 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 99 bytes 12268 (11.9 KiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
 inet 127.0.0.1 netmask 255.0.0.0
 inet6 ::1 prefixlen 128 scopeid 0x10<host>
 loop txqueuelen 1 (Local Loopback)
 RX packets 1462 bytes 122634 (119.7 KiB)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 1462 bytes 122634 (119.7 KiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

wlp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 193.137.158.52 netmask 255.255.254.0 broadcast 193.137.159.255
 inet6 fe80::3ea0:67ff:fef6:24d5 prefixlen 64 scopeid 0x20<link>
 ether 3c:a0:67:f6:24:d5 txqueuelen 1000 (Ethernet)
 RX packets 96743 bytes 95029706 (90.6 MiB)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 55674 bytes 9547408 (9.1 MiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

Figure 4.7: ifconfig answer

I can't be sure but I guess that it's because of the interfaces in my computer, like the "lo, 127.0.0.1/8" the one who break our range in two different parts, and the range of the wlp2s0<sup>11</sup>, the one I use to connect to internet, if the docker network overlapping that range will be created but the containers won't be able to connect it, I'm not sure but it's the most coherent explanation I can give after my empirical experience.

<sup>11</sup>The one I use to be connected to internet, in the eduroam net the range is 193.137.158.52, that is the real limit for our network

(The `ham0` interface has no relations with the docker deployment, it's particular from my host, in this case don't affect, because the `I0` interface make us discard the first half range of the network (if my hypothesis is right), but if we would like we could take the first half instead of the second, from `1.1.1.1` until `126.255.255.255`, it should be possible as well.

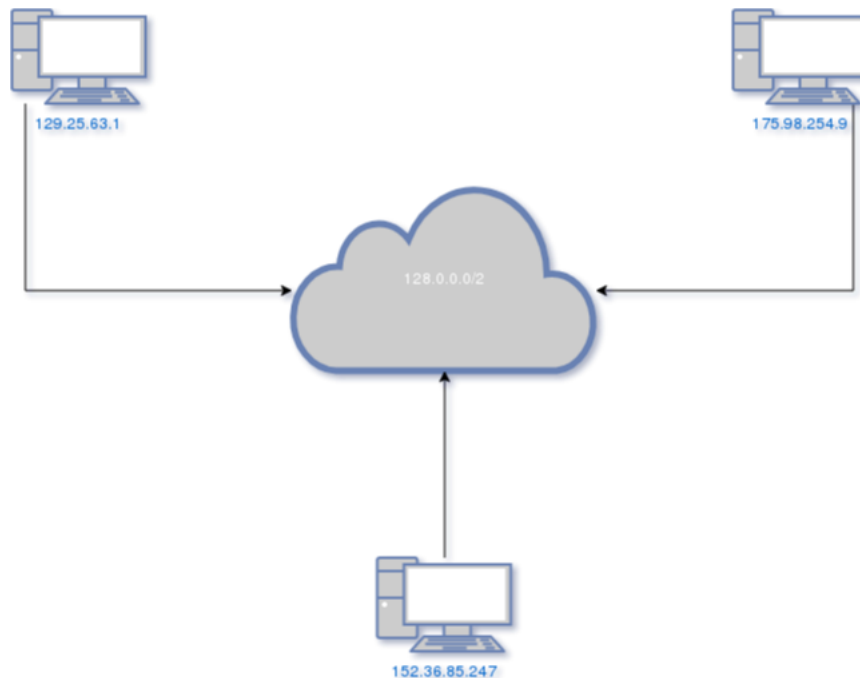


Figure 4.8: Scheme of the deployment II

## Docker-compose

With the network initialized:

```
1 docker network create -d macvlan --subnet=128.0.0.0/2 --gateway
 =191.255.255.254 UniqueNet
```

We write in the the docker-compose:

```
1 version: "3.5"
2 services:
3 host1:
4 container_name: host1
5 image: fedora-image:dockerfile
6 stdin_open: true
7 tty: true
8 privileged: true
9 networks:
10 UniqueNet:
11 ipv4_address: 129.25.63.1
12
13 host2:
14 container_name: host2
15 image: fedora23:dockerfile
```

```

16 stdin_open: true
17 tty: true
18 networks:
19 UniqueNet:
20 ipv4_address: 189.36.85.247
21
22 host3:
23 container_name: host3
24 image: fedora23:dockerfile
25 stdin_open: true
26 tty: true
27 networks:
28 UniqueNet:
29 ipv4_address: 172.98.254.9
30
31 networks:
32 UniqueNet:
33 external: true

```

Notice that each IP are in the scope of the network defined Basically starting from the other topology this one is easier, the "most" difficult here it's how to define the network, and studying the possibilities of overlapping.

## BASH

```

1 #!/bin/bash
2
3 #Rise up Unique-network topology
4
5 #Stop and delete all the enable containers
6 docker stop $(docker ps -a -q)
7 docker rm $(docker ps -a -q)
8
9 #Restart network status and create the one we need.
10 docker network rm $(docker network ls)
11 docker network create -d bridge --subnet=128.0.0.0/2 --gateway
 =191.255.255.254 UniqueNet
12
13 #host1
14 docker run --name host1 --privileged=true --network UniqueNet --ip
 129.25.63.1 -itd fedora-image:dockerfile
15
16 #host2
17 docker run --name host2 --privileged=true --network UniqueNet --ip
 189.36.85.247 -itd fedora-image:dockerfile
18
19 #host3
20 docker run --name host3 --privileged=true --network UniqueNet --ip
 172.98.254.9 -itd fedora-image:dockerfile

```

After that, we have the three virtual machines running in a (more or less) big range network, but this is just, like I don't stop to say, the environment implementation, we need create our connections (or not) to our machines and, let's go to explain it in the next chapter.

*All the networks diagram here exposed, and in future chapters, were created using <https://www.draw.io/>*



## Chapter 5

# Network Interconnection



Figure 5.1: Basic network interconnections.

In this chapter we retake the environments deployed and dive deeper in the working of the connections, it could be considered the containers configuration inside the system created already in the chapter 4. Let's explain how create to refine the virtualization and how to get, as good as possible, a functional network virtualization. How restrict or rise the scope of the machines depending of the case. I'm going to divide the works here in a summary of the deploying (professor work) and creating the connections (Student work), but that point of view will be explained better in the appendix added at the end of the document. For this we're going to introduce the option `exec` of `docker`, as in `docker-compose` as in the `bash` script like tools.

In other words let's develop the concepts stated above, **Machine configuration** and **Routing protocol** both in a basic but functional approach.

Remember use the image created with the **dockerfile** in the previous chapter 4 if you did not before.

## 5.1 Topology I: Network interconnections

In the first topology we have two machines in different networks and one sharing both. At the first sight we could try connect `host1` with `host3` passing through `host2`, like a router, having two different networks that could be considered like wires.

Now we have the same system before, we have to configure the machines to avoid, the above-mentioned, the virtual gateway. For this we have to modify the route table of each container (virtual host) to avoid reference to the gateway, if we look the route table of one of the containers it looks like:

For erasing it in with the **host1** attached:

```
1 route del -net 0.0.0.0 netmask 0.0.0.0
```

```
> docker attach host1
[root@2e53460ad22f /]# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 20.255.255.254 0.0.0.0 UG 0 0 0 eth0
20.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth0
[root@2e53460ad22f /]#
```

Figure 5.2: routing table from host1, practically same that for the others containers.

Deleting that line we avoid all the connections that pass through the virtual gateway but the the student should not to see the existence of that virtual gateway, it is not necessary anymore. Thus, after delete that line we still have in our routing table something like *"every packet that go to '20.0.0.0/8' network will be send by eth0"*, in fact if we try to send some packet to a different network, for example a *"ping"* will be returned *"connect: Network is unreachable"*. We could configure our routing table for saying something like: *"Every packet should pass by eth0"*, for that we should delete all the entries in the routing table:

```
1 route del -net 0.0.0.0 netmask 0.0.0.0 #Not necessary if you delete it
 already , it's the gateway line
2
3 route del -net 20.0.0.0 netmask 255.0.0.0
```

And after that add the condition of everything go through eth0 whose gateway will be host2:

```
1 $ route add default eth0
2
3 $ route add default gw host2 eth0
```

We need use both instructions because in other case will be impossible assign an gateway if the machine don't have instructions about what interface use first, is a hierarchies problem.

We can delete that route line of each container, for that we can add to the docker-compose **command:**

```
1 version: "3.5"
2 services:
3 host1:
4 container_name: host1
5 image: fedora-image:dockerfile
6 stdin_open: true
7 tty: true
8 privileged: true
9 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
 del -net 20.0.0.0 netmask 255.0.0.0 && route add default eth0 &&
 route add default gw host2 eth0 && /bin/bash"
10 networks:
11 NetA:
12 ipv4_address: 20.1.1.1
13
14 host2:
15 container_name: host2
```

```

16 image: fedora—image: dockerfile
17 stdin_open: true
18 tty: true
19 privileged: true
20 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && /bin/
bash"
21 networks:
22 NetA:
23 ipv4_address: 20.156.200.200
24 NetB:
25 ipv4_address: 21.123.252.96
26
27 host3:
28 container_name: host3
29 image: fedora—image: dockerfile
30 stdin_open: true
31 tty: true
32 privileged: true
33 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
del -net 21.0.0.0 netmask 255.0.0.0 && route add default eth0 &&
route add default gw host2 eth0 && /bin/bash"
34 networks:
35 NetB:
36 ipv4_address: 21.55.89.65
37
38 networks:
39 NetA:
40 external: true
41 NetB:
42 external: true

```

*command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route del -net 20.0.0.0 netmask 255.0.0.0 && route add default eth0 && route add default gw host2 eth0 && /bin/bash"*<sup>1</sup> " will execute the that route command and after will execute again the bash to keep in up-state, after that the route table have already the form we want.

Let's put a little summary of the context

host1—> eth0: 20.1.1.1

host2—> eth0: 20.156.200.200

host2—> eth1: 21.123.252.96

host3—> eth0: 21.55.89.65

Currently, with that docker-compose, the host1, host2 and host3 are communicated, because we modify the routing table already to allow it.

<sup>1</sup>This command will do the container don't finish and stay eavesdropping until you attach the container, basically it keep the container started in the background

### 5.1.1 Bash Script

Every step is explained in the docker-compose apart so let's just put the code to automatize the same environment, just in case of needhood or for clarifying the information, it's just to add the three last lines:

```

1 #!/bin/bash
2 #Router topology , enabling host1 and host3 connection
3
4 #Stop and delete all the enable containers
5 docker stop $(docker ps -a -q)
6 docker rm $(docker ps -a -q)
7
8 #Networks
9 docker network rm $(docker network ls)
10 docker network create -d bridge --subnet=20.0.0.0/8 --gateway
 =20.255.255.254 NetA
11 docker network create -d bridge --subnet=21.0.0.0/8 --gateway
 =21.255.255.254 NetB
12
13 #execution
14 #host1
15 docker run --name host1 --privileged=true --network NetA --ip
 20.123.54.98 -itd fedora-image:dockerfile
16
17 #host2
18 docker run --name host2 --privileged=true --network NetA --ip
 20.0.0.222 -itd fedora-image:dockerfile
19 docker network connect --ip 21.0.0.33 NetB host2
20
21 #host3
22 docker run --name host3 --privileged=true --network NetB --ip 21.0.0.56
 -itd fedora-image:dockerfile
23
24 #Customizing environment
25 docker exec -d host1 route del -net 0.0.0.0 netmask 0.0.0.0
26 docker exec -d host1 route del -net 20.0.0.0 netmask 255.0.0.0
27 docker exec -d host1 route add default eth0
28 docker exec -d host1 route add default gw host2 eth0
29
30 docker exec -d host2 route del -net 0.0.0.0 netmask 0.0.0.0
31
32 docker exec -d host3 route del -net 0.0.0.0 netmask 0.0.0.0
33 docker exec -d host3 route del -net 21.0.0.0 netmask 255.0.0.0
34 docker exec -d host3 route add default eth0
35 docker exec -d host3 route add default gw host2 eth0

```

### Notes

Here we have 2 machines in different networks interconnected by other machine, we could call it "router", we have to remember 3 points:

- the `ip_forward` is by default enable (value=1) in the containers, so the host2 will redirect directly without activate, if we want disable it's as easy like use the command:

(I tried add in and script and was giving not result, but added in the docker-compose file run correctly this option)

```
1 $ echo 0 > /proc/sys/net/ipv4/ip_forward
2
```

We could use iptables for avoiding redirect packets but that would confuse to the student in case we would like show them how to modify the ip\_forward flag and understand its working.

- We are using 'host2' instead of the IP because of a property of created networks with bridge drive, containers in the same network can communicate using their alias, that's the reason host1 couldn't speak with host3 using the name.
- Following this scheme we could rise the complexity as much as we would like, we keep in the basics to go deeper and can extrapolate to any case.

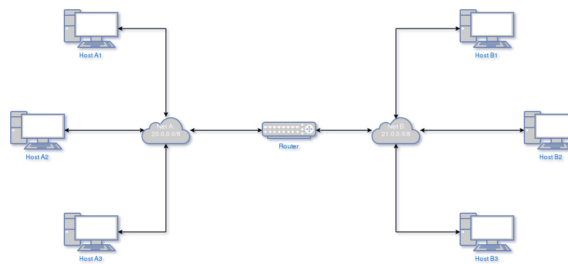


Figure 5.3: Possible system to assemble

## 5.2 Topology II: Unique Network, Firewall Restrictions

Here we have one big network with several machines interconnected, we want to get the same deployment that in the chapter 4, but this time instead of create directly the communication between host1 and host3 going through host2, due to host1 and host3 are in the same network, we have to restrict the access first. And after that we should be able to connect host1 and host3 using host2 like gateway. The advantage of this case it's that if in the host machine you are able to disable all the networks device (even I/O) you can use a big range of IP's for your experiments, but finding other problems like the only one network interface that have host2.

Why I put here this conclusions? Because it's obvious which method is better in this point of the presentation, it's much more useful the one with several networks, but in any case just like a complementary apart let's develop this possibility that will give us a introduction to use iptables, for a possible, more complex environment.

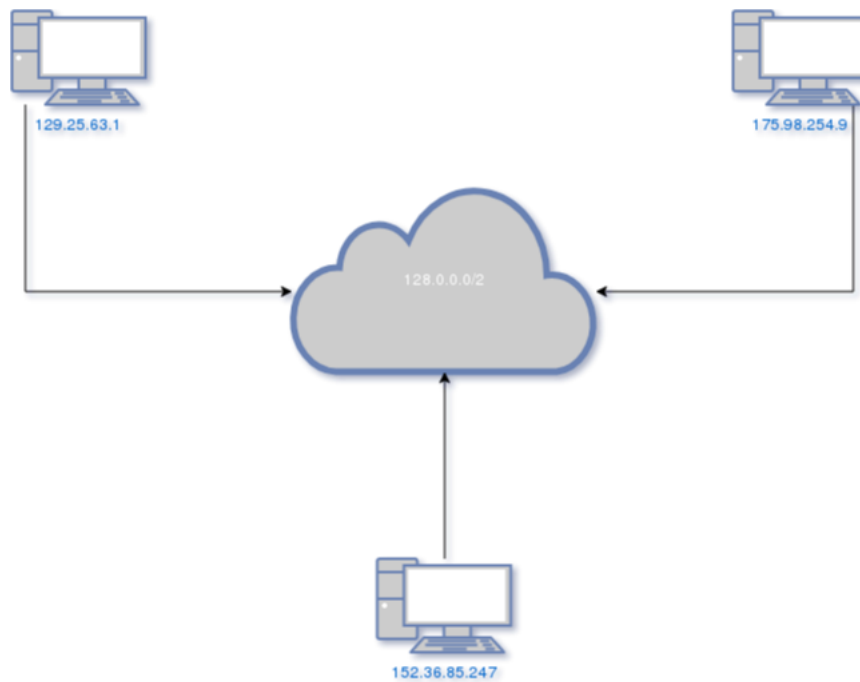


Figure 5.4: figure  
Unique network topology

The first we have to develop is a restriction between host1 and host3, our target is restrict it, we could use the IP address but that's easy to change, and the student should change it in fact to accomplish the practice, so we're going to use a lower level, using the MAC address for restriction we assure that the student couldn't change it to avoid the restriction. Notice that now we have to fix the MAC addresses in each machine to solve any possible problem.

After the restriction we should set-up each host, routing table, to make them pass all the information through host2 who will (or not) redirect to the proper machine.

**Note: Remember initialized the Unique-network like we did in the chapter 4.**

Like reference we will use the same IP's for each host:

host1—> eth0: 129.25.63.1

host2—> eth0: 189.36.85.247

host3—> eth0: 172.98.254.9

### 5.2.1 Docker-compose

Here we have to modify the routing table in the same way we did in the previous implementation and add the iptables restriction, the docker.compose.yml will look like:

```

1 version: "3.5"
2 services:
3 host1:
4 image: fedora-image:dockerfile
5 stdin_open: true
6 tty: true

```

```

7 privileged: true
8 mac_address: 02:42:01:01:01:01
9
10 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
 add -net 0.0.0.0/0 gw 189.36.85.247 eth0 && route del -net 128.0.0.0
 netmask 192.0.0.0 && iptables -A INPUT -m mac --mac-source
 02:42:03:03:03:03 -j REJECT && /bin/bash"
11
12 networks:
13 UniqueNet:
14 ipv4_address: 129.25.63.1
15 container_name: host1
16
17
18 host2:
19 image: fedora-image: dockerfile
20 stdin_open: true
21 tty: true
22 privileged: true
23 mac_address: 02:42:02:02:02:02
24
25 command: bash -c "route del -net 0.0.0.0 && echo 1 > /proc/sys/net/
 ipv4/ip_forward && /bin/bash"
26
27
28 networks:
29 UniqueNet:
30 ipv4_address: 189.36.85.247
31 container_name: host2
32
33 host3:
34 image: fedora-image: dockerfile
35 stdin_open: true
36 tty: true
37 privileged: true
38 mac_address: 02:42:03:03:03:03
39
40 command: bash -c "route del -net 0.0.0.0 && route add -net 0.0.0.0/0
 gw 189.36.85.247 eth0 && route del -net 128.0.0.0 netmask 192.0.0.0
 && /bin/bash"
41
42
43 networks:
44 UniqueNet:
45 ipv4_address: 172.98.254.9
46 container_name: host3
47
48 networks:
49 UniqueNet:
50 external: true

```

Notice that in the host2 we're running "echo 1 > /proc/sys/net/ipv4/ip\_forward", by default it's at '1' already, but to ease for future implementation that we have to understand that we can change the value just changing '1' by '0'

**Note:** We are skipping several steps because were explained in the previous topology, for example the lines to remove and add in the routing table, notice the di

The result of this docker-compose can be checked:

```
> docker-compose up -d
host3 is up-to-date
Starting host1 ...
Starting host1 ... done
> docker-compose ps
Name Command State Ports

host1 bash -c route del -net 0.0 ... Up
host2 bash -c route del -net 0.0 ... Up
host3 bash -c route del -net 0.0 ... Up
> docker attach host1
[root@6f62009678c2 /]# ping host2
PING host2 (189.36.85.247) 56(84) bytes of data.
64 bytes from host2.UniqueNet (189.36.85.247): icmp_seq=1 ttl=64 time=0.105 ms
64 bytes from host2.UniqueNet (189.36.85.247): icmp_seq=2 ttl=64 time=0.080 ms
64 bytes from host2.UniqueNet (189.36.85.247): icmp_seq=3 ttl=64 time=0.080 ms
```

Figure 5.5: figure  
Host1 communicate with Host2

```
[root@6f62009678c2 /]# ping host3
PING host3 (172.98.254.9) 56(84) bytes of data.
64 bytes from host3.UniqueNet (172.98.254.9): icmp_seq=1 ttl=63 time=0.186 ms
From host2.UniqueNet (189.36.85.247): icmp_seq=2 Redirect Host(New nexthop: host3.UniqueNet (172.98.254.9))
64 bytes from host3.UniqueNet (172.98.254.9): icmp_seq=2 ttl=63 time=0.140 ms
From host2.UniqueNet (189.36.85.247): icmp_seq=3 Redirect Host(New nexthop: host3.UniqueNet (172.98.254.9))
64 bytes from host3.UniqueNet (172.98.254.9): icmp_seq=3 ttl=63 time=0.135 ms
From host2.UniqueNet (189.36.85.247): icmp_seq=4 Redirect Host(New nexthop: host3.UniqueNet (172.98.254.9))
64 bytes from host3.UniqueNet (172.98.254.9): icmp_seq=4 ttl=63 time=0.155 ms
From host2.UniqueNet (189.36.85.247): icmp_seq=5 Redirect Host(New nexthop: host3.UniqueNet (172.98.254.9))
64 bytes from host3.UniqueNet (172.98.254.9): icmp_seq=5 ttl=63 time=0.158 ms
^C
--- host3 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 13ms
rtt min/avg/max/mdev = 0.135/0.154/0.186/0.023 ms
[root@6f62009678c2 /]#
```

Figure 5.6: figure  
Host1 communicate with Host3 through Host2(189.36.85.247)

*The same happen in the other sense, host3 to host1 communication.*

**route -n**

```
[root@6f62009678c2 /]# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 189.36.85.247 0.0.0.0 UG 0 0 0 eth0
[root@6f62009678c2 /]#
```

Figure 5.7: figure  
How will look routing table in host1 and host3

```
1 route del -net 0.0.0.0 netmask 0.0.0.0
```

After delete that line from the route table we can check that the connections from host1 to host2 or host3 are still working this is the point we would like to avoid, we need that host1 and host3 can't keep a direct connection, what's we're going to do?, easy Iptables



## Iptables

We are going to add restrictions using Iptables to host1 and to host3 to avoid connections from the respective forbidden access, but... should we put the restriction to the IP address? No, the student could change it (The target is that the fact) and skip the restriction, The point we need to restrict is the MAC Address, that's the reason we used *macvlan* for this case.

Let's check the MAC Addresses of each host to clarify (I think that the MAC assignment has relationship with the host's IP), We will create our docker-compose or the bash script to fix the mac-addresses in each machine, I will use the last two segments with the respective number of each host.

**host1:** MAC1→ 02:42:01:01:01:01

**host2:** MAC2→ 02:42:02:02:02:02

**host3:** MAC3→ 02:42:03:03:03:03

### host1

Let's add at the INPUT table packet who arrive to the host1 from the MAC3 will be dropped.

```
1 iptables -A INPUT -m mac --mac-source 02:42:03:03:03:03 -j DROP
```

### host3

Let's add the same restriction but in the other sense (MAC1)

```
1 iptables -A INPUT -m mac --mac-source 02:42:01:01:01:01 -j DROP
```

## 5.2.2 Routing

Now we have to configure innerly the host to get the connection through the virtualized machines to allow it.

### host1

1. We delete the route table

```
1 $ route del -net 0.0.0.0 netmask 0.0.0.0
2 $ route del -net 128.0.0.0 netmask 192.0.0.0
```

2. We add the host2 like gateway for the full IP range, this way each connection the host 1 would like to do will pass over host2

```
1 $ route add -net 0.0.0.0/0 gw 189.36.85.247 eth0
```

being 189.36.85.247 the host2's address.

It's better if the order of execution it's like

```
1 $ route del -net 0.0.0.0 netmask 0.0.0.0
2 $ route add -net 0.0.0.0/0 gw 189.36.85.247 eth0
3 $ route del -net 128.0.0.0 netmask 192.0.0.0
```

### host3

We have to modify the route table in the same form. 1. We delete the route table

```
1 route del -net 0.0.0.0 netmask 0.0.0.0
2 route del -net 128.0.0.0 netmask 192.0.0.0
```

2. We add the host2 like gateway for the full IP range, this way each connection the host 1 would like to do will pass over host2

```
1 route add -net 0.0.0.0/0 gw 189.36.85.247 eth0
```

being 189.36.85.247 the host2's address.

### host2

By default we can check that the ip\_forwarding flag is on:

```
1 $ docker attach host2
```

```
1 cat /proc/sys/net/ipv4/ip_forward
```

It'll return '1', we could disable using:

```
1 echo 0 > /proc/sys/net/ipv4/ip_forward
```

or activate with

```
1 echo 1 > /proc/sys/net/ipv4/ip_forward
```

Now if the ip\_forwarding is on the connection between **host1**↔**host3** will be possible because of host2.

**Docker-compose.yml**

```
1 version: "3.5"
2 services:
3 host1:
4 image: fedora--image: dockerfile
5 stdin_open: true
6 tty: true
7 privileged: true
8 mac_address: 02:42:01:01:01:01
9
10 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
11 add -net 0.0.0.0/0 gw 189.36.85.247 eth0 && route del -net 128.0.0.0
12 netmask 192.0.0.0 && iptables -A INPUT -m mac --mac-source
13 02:42:03:03:03:03 -j REJECT && /bin/bash"
14
15 networks:
16 UniqueNet:
17 ipv4_address: 129.25.63.1
18 container_name: host1
19
20 host2:
21 image: fedora--image: dockerfile
22 stdin_open: true
23 tty: true
24 privileged: true
25 mac_address: 02:42:02:02:02:02
26
27 command: bash -c "route del -net 0.0.0.0 && echo 1 > /proc/sys/net/
28 ipv4/ip_forward && /bin/bash"
29
30 networks:
31 UniqueNet:
32 ipv4_address: 189.36.85.247
33 container_name: host2
34
35 host3:
36 image: fedora--image: dockerfile
37 stdin_open: true
38 tty: true
39 privileged: true
40 mac_address: 02:42:03:03:03:03
41
42 command: bash -c "route del -net 0.0.0.0 && route add -net 0.0.0.0/0
43 gw 189.36.85.247 eth0 && route del -net 128.0.0.0 netmask 192.0.0.0
44 && /bin/bash"
45
46 networks:
47 UniqueNet:
48 ipv4_address: 172.98.254.9
49 container_name: host3
50
51 networks:
52 UniqueNet:
53 external: true
```

\*Notice that is necessary the creation of the network before the *"docker-compose up -d"*

### 5.2.3 Bash Script

I had some problems trying to disable the default value of ip-forwarding, in any case I'm going to implement the code but **I totally recommend the docker-compose** in this case.

```

1 #!/bin/bash
2
3 #Rise up Router topology.
4 #Stop and delete all the enable containers
5 docker stop $(docker ps -a -q)
6 docker rm $(docker ps -a -q)
7
8 #We delete all the networks and create the necessary in the script.
9 #This step its not necessary if the networks are created already
10 docker network rm $(docker network ls)
11 docker network create -d macvlan --subnet=128.0.0.0/2 --gateway
 =191.255.255.254 UniqueNet
12
13 #host1
14 docker run --name host1 --privileged=true --network UniqueNet --ip
 129.25.63.1 --mac-address 02:42:01:01:01:01 -itd fedora--image:
 dockerfile
15 #host2
16 docker run --name host2 --privileged=true --network UniqueNet --ip
 189.36.85.247 --mac-address 02:42:02:02:02:02 -itd fedora--image:
 dockerfile
17 #host3
18 docker run --name host3 --privileged=true --network UniqueNet --ip
 172.98.254.9 --mac-address 02:42:03:03:03:03 -itd fedora--image:
 dockerfile
19
20 #host1
21 docker exec -d host1 route del -net 0.0.0.0
22 docker exec -d host1 route del -net 0.0.0.0 netmask 0.0.0.0
23 docker exec -d host1 route add -net 0.0.0.0/0 gw 189.36.85.247 eth0
24 docker exec -d host1 route del -net 128.0.0.0 netmask 192.0.0.0
25
26 docker exec -d host1 iptables -A INPUT -m mac --mac-source
 02:42:03:03:03:03 -j REJECT
27
28 #host3
29 docker exec -d host3 route del -net 0.0.0.0
30 docker exec -d host3 route add -net 0.0.0.0/0 gw 189.36.85.247 eth0
31 docker exec -d host3 route del -net 128.0.0.0 netmask 192.0.0.0
32
33 docker exec -d host3 iptables -A INPUT -m mac --mac-source
 02:42:01:01:01:01 -j REJECT
34
35 #host2
36 docker exec -d host2 route del -net 0.0.0.0
37 sleep 5
38 docker exec -d host2 echo 1 > /proc/sys/net/ipv4/ip__forward

```

## 5.3 Results and conclusions

Speaking about the deployment of the topologies, the second one is more complex than the first one but with more potential to look real. We have mainly two **problems** in the "Topology II".

- **1. the host2(router) just have one interface**, it's a hard problem to solve, at list like I was focusing the problem.
- **2. The IP range restriction** it's with different the worst part, still I'm not sure if it's because the route table or because the interfaces already working, and at the best case we have *lo interface* with the net 127.0.0.1/8, so at the best in each computer we could have our range divided by the half. I didn't study the possibility of disable the *lo interface*, or change the range, because even in case that it could be done easily I think that the other it's much more clarify.

About the "Topology I", truly it's not perfect, we have to fix the first 8 bits in our IP range, but I think that explaining enough the virtualization limitations it's easier conceive the fact that the first 8 bits is nominating to a virtual wire.

About the concept I have to admit that the **Network planning** have a poor implementation because the IP range have to be defined inside the docker, and not innerly through the containers. But the other concepts, **Network services, Host configuration and Routing protocol** are accomplished properly for the academic purpose.

In the appendix A I'm going to develop the proposed exercise for the student, exposing the step that the teacher should do previously to deploy, the document (a reference) that the student should have access to complete the exercise and the solutions to this problem.

And in the appendix C I'm going to develop the document that the student should open to start working with the environment.



# Bibliography

- [1] " ". *An introduction to virtual machines*. <https://www.ionos.co.uk/digitalguide/server/know-how/virtual-machines/>. [Online; accessed 04-June-2019]. (2018).
- [2] "Oskar Andreasson". *Iptables Tutorial 1.2.2*. (2006).
- [3] "Roderick Bauer". *What's the Diff: VMs vs Containers*. <https://www.backblaze.com/blog/vm-vs-containers/>. [Online; accessed 04-June-2019]. (2018).
- [4] "Sujata Biswas". *Learning Bash Shell Scripting*. (2017).
- [5] "Raouf Boutaba". *A survey of network virtualization*. <https://www.sciencedirect.com/science/article/pii/S1389128609003387>. [Online; accessed 25-May-2019]. (2010).
- [6] "Doug Chamberlain". *Docker Downsides*. <https://blog.netapp.com/blogs/containers-vs-vms/>. [Online; accessed 04-June-2019]. (2018).
- [7] *Chapter 14. iptables firewall*. <http://linux-training.be/networking/ch14.html>. [Online; accessed 05-June-2019].
- [8] *Customize the docker0 bridge*. [https://docs.docker.com/v17.09/engine/userguide/networking/default\\_network/custom-docker0/](https://docs.docker.com/v17.09/engine/userguide/networking/default_network/custom-docker0/). [Online; accessed 06-June-2019]. (2018).
- [9] *Docker network documentation*. <https://docs.docker.com/network/>. [Online; accessed 05-June-2019]. (2019).
- [10] *Dockerfile tutorial by example*. <https://takacsmark.com/dockerfile-tutorial-by-example-dockerfile-best-practices-2018/>. [Online; accessed 28-May-2019]. (2018).
- [11] "Docker Documentation". *Dockerfile reference*. <https://docs.docker.com/engine/reference/builder/>. [Online; accessed 28-May-2019]. (2019).
- [12] "Gloria Palma González". *[Docker] APIError: 400 Client Error: Bad Request ("client is newer than server (client API version: 1.26, server API version: 1.24)")*. <https://medium.com/@gloriapalmagonzalez/apierror-400-client-error-bad-request-client-is-newer-than-server-client-api-version-1-26-c2b6ad976751>. [Online; accessed 04-June-2019]. (2017).
- [13] "Michael Kerrisk". *cgroups - Linux control groups*. <http://man7.org/linux/man-pages/man7/cgroups.7.html>. [Online; accessed 04-June-2019]. (2019).
- [14] "Michael Kerrisk". *overview of Linux namespaces*. <http://man7.org/linux/man-pages/man7/namespaces.7.html>. [Online; accessed 29-May-2019]. (2019).
- [15] "Varun Kumar". *sh vs bash : A summary*. [https://medium.com/@varunkumar\\_53845/sh-vs-bash-a-summary-50f92a719e0d](https://medium.com/@varunkumar_53845/sh-vs-bash-a-summary-50f92a719e0d). [Online; accessed 05-June-2019]. (2017).
- [16] "Bruce Malaudzi". *What is the difference between VPN and virtualization?* <https://www.quora.com/What-is-the-difference-between-VPN-and-virtualization>. [Online; accessed 29-May-2019]. (2017).
- [17] "Mark Church ". *Understanding Docker Networking Drivers and their use cases*. <https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/>. [Online; accessed 05-June-2019]. (2016).

- [18] "Melissa Anderson ". *How To Install Docker Compose on Ubuntu 16.04*. <https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-16-04>. [Online; accessed 05-June-2019]. (2017).
- [19] *Networking in Compose*. <https://docs.docker.com/compose/networking/>. [Online; accessed 06-June-2019]. (2019).
- [20] "Rani Osnat". *A Brief History of Containers*. <https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>. [Online; accessed 04-June-2019]. (2018).
- [21] "Margaret Rouse". *container (containerization or container-based virtualization)*. <https://searchitoperations.techtarget.com/definition/container-containerization-or-container-based-virtualization>. [Online; accessed 04-June-2019]. (2018).
- [22] Jyotiprakash Sahoo... *Virtualization: A Survey on Concepts*. <https://ieeexplore.ieee.org/abstract/document/5474503/authors#authors/>. [Online; accessed 30-May-2019]. 2010.
- [23] Amit Singh. *An Introduction to Virtualization*. <http://www.kernelthread.com/publications/virtualization/>. [Online; accessed 24-May-2019]. 2004.
- [24] " J.E. Smith and Ravi Nair". *The architecture of virtual machines*. <https://ieeexplore.ieee.org/abstract/document/1430629>. [Online; accessed 04-June-2019]. (2005).
- [25] "Stephen Soltesz..." *Container-based Operating System Virtualization*. <https://dl.acm.org/citation.cfm?id=1273025>. [Online; accessed 25-May-2019]. (2007).
- [26] " Supriyo Biswas ". *An In-Depth Guide to iptables*. <https://www.booleanworld.com/depth-guide-iptables-linux-firewall/>. [Online; accessed 05-June-2019]. (2018).
- [27] "Christopher Tozzi". *Docker Downsides*. <https://www.channelfutures.com/open-source/docker-downsides-container-cons-to-consider-before-adopting-docker>. [Online; accessed 04-June-2019]. (2017).
- [28] *What is KVM?* <https://www.redhat.com/en/topics/virtualization/what-is-kvm>. [Online; accessed 31-May-2019].
- [29] "Bibin Wilson". *What is Docker? How Does it Work?* <https://devopscube.com/what-is-docker/>. [Online; accessed 04-June-2019]. (2016).



## Appendix A

# Deploy for a student

In this example to deploy the system with the first studied topology, the one with several networks because is over my point of view the most recommendable, exploring the details to consider by the professor side, it's just the deployment, the steps to solve it will be at the end of this appendix.

### A.1 Deployment

We are going to virtualize the next scenario with the tools and the knowledge we acquired during the document, it'll work like a synthesis summarizing the conclusions (what topology use better) and the steps to rise the environment. The network I decide use for this is the next:

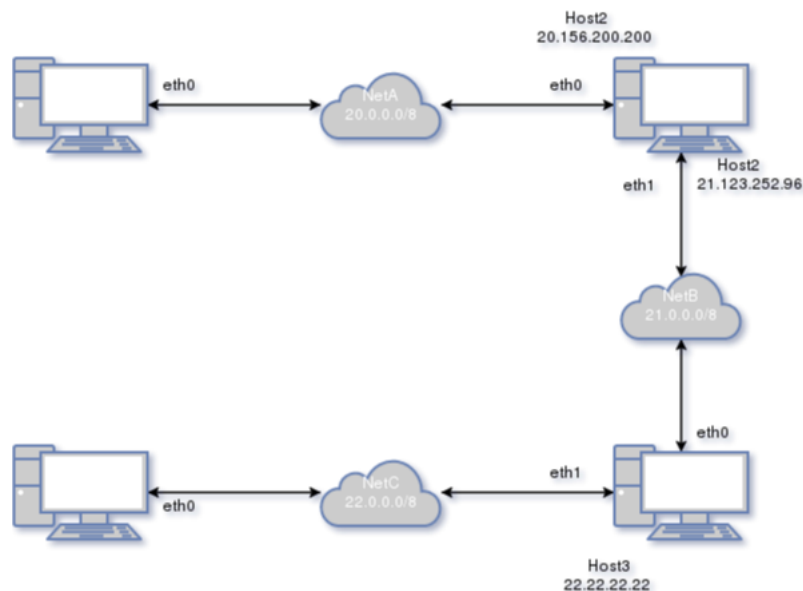


Figure A.1: Deployment Scheme

Setting some interfaces with a predefined IP we "skip" the virtualization network limitation, we could do that the middle machines were inaccessible (Don't be able to modify IP's doesn't setting privileged=true) to the student to avoid the possibility of misunderstanding networks because the limitations. I just exposes this point to make at the implementer be

consciousness of the context and make him choose the most convenient way. For that let's go deeper.

### A.1.1 Networks

We have to create three different networks with the same features that in the first topology:

```
1 docker network create -d bridge --subnet=20.0.0.0/8 --gateway
 =20.255.255.254 NetA
2 docker network create -d bridge --subnet=21.0.0.0/8 --gateway
 =21.255.255.254 NetB
3 docker network create -d bridge --subnet=22.0.0.0/8 --gateway
 =22.255.255.254 NetC
```

And we're going to assign the next MAC addresses to the machines: **host1:** MAC1→ 02:42:01:01:01:01

**host2:** MAC2→ 02:42:02:02:02:02

**host3:** MAC3→ 02:42:03:03:03:03

**host4:** MAC4→ 02:42:04:04:04:04

### A.1.2 Machines

Now we have to prepare each "machine" (container) to be a challenger to the student.

#### host1

first we have to erase the route table:

```
1 $ route del -net 0.0.0.0
2 $ route del -net 20.0.0.0 netmask 255.0.0.0
```

And after we assign to the default interface, eth0, the IP '0.0.0.0' to assign any.

```
1 ifconfig eth0 0.0.0.0
```

Now we have our host1 isolated and without an IP assigned perfect situation to develop our knowledge about networks.

#### host2

I'm not sure if it's better let the student have privileges in this host to be able to change the Ethernet configuration. Because the limitations for the correct working of the system

the IP's of the eth0 have to start by '20' and in eth1 by '21'.

By default the IP\_forwarding flag it's enable, if it's our intention introduce the concept, really important under my point of view, we should allow at the student interact with this machine to activate the flag, that we will disable after execution of the container and deleting the virtual gateway dependency in the route table with. Remember add the MAC restriction to "simulate" the wired connection:

```
1 $ route del -net 0.0.0.0
2
3 $ echo 0 > /proc/sys/net/ipv4/ip_forward
```

For change the ip\_forward flag, from '1' to '0', I had problems with a bash script. I recommend docker-compose in this case.

### host3

Is other middle machine, the configurations will be quite similar, here we could study the possibilities of set the addresses in both interface or follow the diagram and just fix one of them and let the student adapt the other one. The other point it's again the ip\_forward, we will disable here to be enable later in the solution. We will eliminate the IP assigned to the machine to be mandatory assign it later with the restriction of belong to the range that Host2 point in the draw, the interface of that network is 'eth0'.

```
1 $ route del -net 0.0.0.0
2 $ route del -net 21.0.0.0 netmask 255.0.0.0
3
4 $ echo 0 > /proc/sys/net/ipv4/ip_forward
5
6 $ ifconfig eth0 0.0.0.0
```

### host4

Same that host1:

```
1 $ route del -net 0.0.0.0
2 $ route del -net 22.0.0.0 netmask 255.0.0.0
```

And after we assign to the default interface, eth0, the IP '0.0.0.0' to assign any.

```
1 ifconfig eth0 0.0.0.0
```

### A.1.3 Docker-compose

This should be the **environment to deploy in front of the student**:

```

1 version: "3.5"
2 services:
3 host1:
4 image: fedora-image:dockerfile
5 stdin_open: true
6 tty: true
7 privileged: true
8 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
9 del -net 20.0.0.0 netmask 255.0.0.0 && ifconfig eth0 0.0.0.0 && /bin/
10 bash"
11 networks:
12 NetA:
13 ipv4_address: 20.1.1.1
14 container_name: host1
15
16 host2:
17 image: fedora-image:dockerfile
18 stdin_open: true
19 tty: true
20 privileged: true
21 command: bash -c "route del -net 0.0.0.0 && echo 0 > /proc/sys/net/
22 ipv4/ip_forward && /bin/bash"
23 networks:
24 NetA:
25 ipv4_address: 20.156.200.200
26 NetB:
27 ipv4_address: 21.123.252.96
28 container_name: host2
29
30 host3:
31 image: fedora-image:dockerfile
32 stdin_open: true
33 tty: true
34 privileged: true
35 command: bash -c "route del -net 0.0.0.0 && route del -net 21.0.0.0
36 netmask 255.0.0.0 && echo 0 > /proc/sys/net/ipv4/ip_forward &&
37 ifconfig eth0 0.0.0.0 && /bin/bash"
38 networks:
39 NetB:
40 ipv4_address: 21.55.89.65
41 NetC:
42 ipv4_address: 22.22.22.22
43 container_name: host3
44
45 host4:
46 image: fedora-image:dockerfile
47 stdin_open: true
48 tty: true
49 privileged: true
50 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
51 del -net 22.0.0.0 netmask 255.0.0.0 && ifconfig eth0 0.0.0.0 && /bin/
52 bash"
53 networks:

```

```
50 NetC:
51 ipv4_address: 22.17.18.19
52 container_name: host4
53
54
55 networks:
56 NetA:
57 external: true
58 NetB:
59 external: true
60 NetC:
61 external: true
62
63
64
```

This environment should be deployed by the student himself using a pre-made script (in bash) in the same folder where we can find this last docker-compose.yml and will be something like:

## A.2 Solution

Now let's explain the steps that the student should realize to connect host1 with host4 (both endings). The solution will be presented in a Bash script, explaining previously the steps:

### A.2.1 Machines

The solutions generally have 3 steps

- 1. Assign an IP to the interface into the range
- 2. Set up the routing table property.
- 3. Enable ip\_forwarding flag in the middle machines

#### host1

- 1. Assign IP

```
1 ifconfig eth0 20.1.1.1
2
```

- 2. Routing table will have the next aspect:
- 3. This machine don't need activate the ip\_forward flag.

```

root@debian:/home/deby# docker attach host1
[root@36de6aa2c797 /]# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 20.156.200.200 0.0.0.0 UG 0 0 0 eth0
0.0.0.0 0.0.0.0 0.0.0.0 U 0 0 0 eth0
[root@36de6aa2c797 /]#

```

Figure A.2: Solution-host1

**host2**

- 1. Assign IP, this machine has assigned to every interface an IP.
- 2. Routing table will have the next aspect:

```

root@debian:/home/deby# docker attach host2
[root@aec481e490d2 /]# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 21.55.89.65 0.0.0.0 UG 0 0 0 eth1
0.0.0.0 0.0.0.0 0.0.0.0 U 0 0 0 eth1
20.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth0
21.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth1
[root@aec481e490d2 /]#

```

Figure A.3: Solution-host2

- 3. To activate the ip\_forward flag just

```
1 $ echo 1 > /proc/sys/net/ipv4/ip_forward
```

**host3**

- 1. Assign IP in the interface that belong to the network '21.0.0.0/8':

```
1 ifconfig eth0 21.55.89.65
2
```

- 2. Routing table will have the next aspect:

```

root@debian:/home/deby# docker attach host3
[root@601e38c5eacb /]# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 21.123.252.96 0.0.0.0 UG 0 0 0 eth0
0.0.0.0 0.0.0.0 0.0.0.0 U 0 0 0 eth0
21.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth0
22.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 eth1
[root@601e38c5eacb /]#

```

Figure A.4: Solution-host3

- 3. To activate the ip\_forward flag just

```
1 $ echo 1 > /proc/sys/net/ipv4/ip_forward
```

### host4

- 1. Assign IP in the interface that belong to the network '22.0.0.0/8':

```
1 ifconfig eth0 22.17.18.19
2
```

- 2. Routing table will have the next aspect:

```
[root@b8156a9ce7c7 /]# route -n
Kernel IP routing table
Destination Gateway Genmask Flags Metric Ref Use Iface
0.0.0.0 22.22.22.22 0.0.0.0 UG 0 0 0 eth0
0.0.0.0 0.0.0.0 0.0.0.0 U 0 0 0 eth0
[root@b8156a9ce7c7 /]#
```

Figure A.5: Solution-host4

- 3. To activate the ip\_forward flag just

```
1 $ echo 1 > /proc/sys/net/ipv4/ip_forward
```

### A.2.2 Docker-Compose system working

```
1 version: "3.5"
2 services:
3 host1:
4 container_name: host1
5 image: fedora-image:dockerfile
6 stdin_open: true
7 tty: true
8 privileged: true
9 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
10 del -net 20.0.0.0 netmask 255.0.0.0 && route add default eth0 &&
11 route add default gw host2 eth0 && /bin/bash"
12 networks:
13 NetA:
14 ipv4_address: 20.1.1.1
15 host2:
16 container_name: host2
17 image: fedora-image:dockerfile
18 stdin_open: true
19 tty: true
20 privileged: true
```

```

20 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
21 add default eth1 && route add default gw host3 eth1 && /bin/bash"
22
23 networks:
24 NetA:
25 ipv4_address: 20.156.200.200
26 NetB:
27 ipv4_address: 21.123.252.96
28
29 host3:
30 container_name: host3
31 image: fedora-image:dockerfile
32 stdin_open: true
33 tty: true
34 privileged: true
35 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
36 add default eth0 && route add default gw host2 eth0 && /bin/bash"
37 networks:
38 NetB:
39 ipv4_address: 21.55.89.65
40 NetC:
41 ipv4_address: 22.22.22.22
42
43 host4:
44 image: fedora-image:dockerfile
45 stdin_open: true
46 tty: true
47 privileged: true
48 command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
49 del -net 22.0.0.0 netmask 255.0.0.0 && route add default eth0 &&
50 route add default gw host3 eth0 && /bin/bash"
51 # command: bash -c "route del -net 0.0.0.0 netmask 0.0.0.0 && route
52 del -net 20.0.0.0 netmask 255.0.0.0 && route add default eth0 &&
53 route add default gw host2 eth0 && /bin/bash"
54 networks:
55 NetC:
56 ipv4_address: 22.17.18.19
57 container_name: host4
58
59 networks:
60 NetA:
61 external: true
62 NetB:
63 external: true
64 NetC:
65 external: true

```

There are several ways to take and solve the architecture, this is just an example, I hope the step were clarify enough, **thank you for you time.**



## Appendix B

# Script for the student

This is an example of what could receive the student in a document and how to deploy the all environment to start working, totally inspired in the appendix A but with a more realistic implementation for the student himself.

### B.1 0. Before starting

The first step is deploy the environment, for that let's use the docker-compose.yml<sup>1</sup>

**Execute this in the terminal, with the terminal session in the path where you can find docker-compose.yml (the one developed in the Appendix A)**

```

1 $docker stop $(docker ps -a -q)
2
3 $docker rm $(docker ps -a -q)
4
5 $docker network rm $(docker network ls)
6
7 $docker network create -d bridge --subnet=20.0.0.0/8 --gateway
 =20.255.255.254 NetA
8 $docker network create -d bridge --subnet=21.0.0.0/8 --gateway
 =21.255.255.254 NetB
9 $docker network create -d bridge --subnet=22.0.0.0/8 --gateway
 =22.255.255.254 NetC
10
11 $docker-compose up -d

```

Execute them separately and without the symbol "\$"

Currently you have deployed in you machine the next Scheme.

---

<sup>1</sup>We could offer the code in the document itself of upload the files at moodle where were available to be downloaded

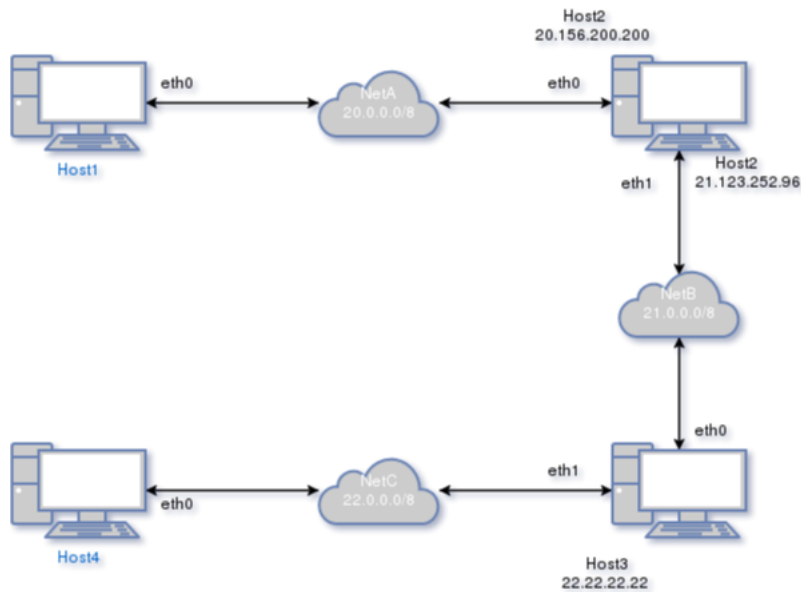


Figure B.1: Deployment Scheme

You can check that everything were right writing in the terminal (Where each container should have the status 'up'):

```
1 $docker-compose ps
```

Respecting the IP-range defined in the draw and knowing that for attaching one machine we just need to write in the terminal (with the proper privileges).

```
1 $docker attach host1
2 $docker attach host2
3 $docker attach host3
4 $docker attach host4
```

Remember after attach one of the host use the **Escape Sequence** to come back to the main session where you can access to the other host using again attach.

**Escape Sequence= Ctrl + p + q**

## B.2 Connect host1 with host3

Now you are going to configure host1, host2 and host3 for allowing a bidirectional connection between all of them. For that the steps you need to follow are:

**1.Check the status of the ip\_routing in host2 and enable if it wouldn't be:**

Tip: Try searching in google "ip forward linux", and discover what is it and how to work. After that look for the status of that parameter using in the terminal, attached to the host2 machine,

```
1 cat /proc/sys/net/ipv4/ip_forward
```

What means the '0'? And the '1'? How can you change the value?

### **2.Assign an IP to host1 in eth0 inside the range 20.0.0.0/8 and add the proper value to the route table**

Tip: In the draw are defined what are the interfaces used in each wire connection and the IP range defined, so we just need going to the host1 session and use the commands "ifconfig", for attach an IP to the interface to be able to communicate and to be communicated, and "route" to add . For going to the host1 machine write in the terminal "docker attach host1".

### **3.Assign an IP to host3 in eth0 interface inside the range 21.0.0.0 and add the proper value to the route table**

Same that in the 2º step but applying to the host3

Remember use the commands exposed above, to interact with the container session (docker attach host1), and the **escape sequence** (Ctrl + p + q) to avoid get down the container.

## **B.2.1 Solutions**

Here we define each change we have to do in each machine to get our goal.

### **1. Enable ip\_forward in host2**

This is as easy like go in to the host2 machine

```
1 $docker attach host2
```

and inside the machine write in the terminal:

```
1 cat /proc/sys/net/ipv4/ip_forward
```

and if the result is a '0' the routing property is disabled and if it's '1' it's enabled. To enable we just need to write:

```
1 echo 1 > /proc/sys/net/ipv4/ip_forward
```

## 2. Assign an IP in eth0 and modify the route table in host1

Attaching to host1 just apply and checking with 'ifconfig' that eth0 has any IP assigned

```
1 $docker attach host1
```

```
1 $ifconfig eth0 20.45.64.34
```

If we check the route table with

```
1 route -n
```

we can check that exist an entry for the *20.0.0.0/8* network but don't exist any for our destiny network, we should add it with:

```
1 route add default gw IP_host2 eth0
```

where we're are basically saying "to every packet that you are sending send it first to host2. Notice here that we use an alias, it's a implementation of docker, for the student here we should put the IP of the host2 attached to the interface 'eth0', it is 20.156.200.200.

## 3. Assign an IP in eth0 and modify the route table in host3

```
1 $docker attach host3
```

```
1 ifconfig eth0 21.58.96.76
```

If we check the route table with:

```
1 route -n
```

we can check that exist an entry for the *21.0.0.0/8* network but don't exist any for our destiny network, we should add it with:

```
1 route add -net 20.0.0.0 netmask 255.0.0.0 gw IP_host2 eth0
```

Here, like host3 is not one of the extremes in the deployment, it's not as easy like say "send every packet by the eth0 interface", that this is what 'default' do, thus, in words we say, every packet that is going to the network *20.0.0.0/8* have to be send to host2 who, working like a router, will allow the connection between host1 and host3. Remember have host2 properly configured for routing.

### B.2.2 Questions:

**1. What happens if the IP assigned to host1 is out of the network range? For example 54.99.22.11? Why?**

*Answer:* The connection is not effectuated because the eth0 that host2 use just will process packet inside this network, it's not able to answer to a packet outside the network range.

**2. Try to write in the route table an order to accomplish the connection but avoiding the default option. Check on internet how to delete lines from the route table.**

*Answer:*

```
1 route add -net 21.0.0.0 netmask 255.0.0.0 gw IP_host2 eth0
```

We specify what network we want to define in the route table, in general the default option is better for this case because the host is in one of the extremes.

**3. If a packet is sended from host1 to host3, what will be the source's address of the packet when host3 receive it?**

*Answer:* The IP that host1 had assigned.

### B.2.3 Proposed exercises

#### Connect host2 with host4

We could apply the steps we did before but considering now host2, host3, and host4 respectively. Now host3 will be the one who will route packets, and host2 need to know what way take to communicate with host4, just remember do not use the default option in the host2 for the route table, because it would interfere with the previous implementation and with the next one as well, we have to define explicitly what network and where to send. With host4 we have not that problem because is the other extreme of the system implemented.

#### Connect host1 with host4

If every task was accomplished successfully now the only thing that we have to add is the route table of host1 and host4 to define what way have to be taken for arrive to the other machine. If both machine have the route table configured with the default option this would be working already, if not it's just to define in each of them where should be sended the packet with the proper network.



## Appendix C

# Docker0 Interface

Here I attach the script we use in the "Unique Network" to modify the docker0 default IP.

```

1 #!/bin/sh -e
2 #
3 # NOTE: Since Docker 1.10 (February 4, 2016), it has been possible to
4 # configure the
5 # Docker daemon using a JSON config file. On Linux, this file is
6 # normally located at
7 # /etc/docker/daemon.json. You should use this JSON config method if
8 # you are running
9 # a version of Docker that is at least 1.10!
10 # Here is an example configuration that sets the docker0 bridge IP to
11 # 192.168.254.1/24:
12 # {
13 # "bip": "192.168.254.1/24"
14 # }
15 #
16 # You can run this script directly from github as root like this:
17 # curl -sS https://gist.githubusercontent.com/kamermans/94
18 # b1c41086de0204750b/raw/configure_docker0.sh | sudo bash -s -
19 # 192.168.254.1/24
20 #
21 # * Make sure you replace "192.168.254.1/24" with the network that you
22 # want to use
23 #
24 # NOTE: This script is intended for Debian / Ubuntu only!
25
26 if [$# -lt 1]; then
27 echo "Usage: sudo ./configure_docker0.sh <ip/CIDR>"
28 echo " examples: "
29 echo " ./configure_docker0.sh 10.200.0.57/16"
30 echo " ./configure_docker0.sh 172.31.0.21/16"
31 echo " ./configure_docker0.sh 192.168.254.1/24"
32 echo " "
33 echo " NOTE: You should stop Docker before running this script."
34 echo " When you restart it, Docker will use the new IP."
35 exit 2
36 fi
37
38 INIT_SYSTEM="sysv"
39 if ps -o comm -1 | grep -q systemd; then
40 INIT_SYSTEM="systemd"
41 fi
42
43 NEW_IP="$1"
44 DOCKER_INIT="/etc/default/docker"

```

```

38
39 if [! -f "$DOCKER_INIT"]; then
40 cat << EOF > $DOCKER_INIT
41 # Docker Upstart and SysVinit configuration file
42
43 # Customize location of Docker binary (especially for development
44 testing).
45 #DOCKER="/usr/local/bin/docker"
46
47 # Use DOCKER_OPTS to modify the daemon startup options.
48 #DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
49 DOCKER_OPTS="--bip=$NEW_IP"
50
51 # If you need Docker to use an HTTP proxy, it can also be specified here
52 .
53 #export http_proxy="http://127.0.0.1:3128/"
54
55 # This is also a handy place to tweak where Docker's temporary files go.
56 #export TMPDIR="/mnt/bigdrive/docker-tmp"
57 EOF
58
59 echo "Created a new Docker default file at $DOCKER_INIT"
60 exit 0;
61 fi
62
63 echo "Removing old docker0 network(s)"
64 NETWORKS=$(ip addr list docker0 | grep "inet " | cut -d" " -f6)
65 for NET in $NETWORKS; do
66 echo " $NET"
67 ip addr del $NET dev docker0
68 done
69
70 echo "Adding new docker0 network"
71 ip addr add $NEW_IP dev docker0
72
73 echo "Removing old iptables rules"
74 iptables -t nat -F POSTROUTING
75 iptables -F DOCKER
76
77 CURRENT_OPTS=$(cat $DOCKER_INIT | grep "^ *DOCKER_OPTS" | sed 's/^/ /' |
78 g')
79 NEW_OPTS=DOCKER_OPTS="--bip=$NEW_IP"
80
81 echo " "
82
83 if ["$CURRENT_OPTS" != ""]; then
84 TEMP_FILE="/tmp/docker_config.tmp"
85 grep -v "^ *DOCKER_OPTS" $DOCKER_INIT > $TEMP_FILE
86 echo " " >> $TEMP_FILE
87 echo DOCKER_OPTS="--bip=$NEW_IP" >> $TEMP_FILE
88 cat $TEMP_FILE > $DOCKER_INIT
89 rm -f $TEMP_FILE
90
91 echo "WARNING: The existing DOCKER_OPTS were overwritten in
92 $DOCKER_INIT: "
93 echo "Old: "
94 echo "$CURRENT_OPTS"
95 echo "New: "
96 echo " $NEW_OPTS"

```



```

93 else
94 echo " " >> $DOCKER_INIT
95 echo DOCKER_OPTS="\--bip=$NEW_IP\" >> $DOCKER_INIT
96 echo "Success: $DOCKER_INIT has been modified."
97 fi
98
99 SYSTEMD_DOCKER_DIR="/etc/systemd/system/docker.service.d"
100 if ["$INIT_SYSTEM" = "systemd"]; then
101 echo "Configuring systemd to use /etc/default/docker"
102 if [! -d $SYSTEMD_DOCKER_DIR]; then
103 mkdir -p $SYSTEMD_DOCKER_DIR
104 fi
105
106 OPTS='$DOCKER_OPTS'
107 cat << EOF > $SYSTEMD_DOCKER_DIR/debian-style-config.conf
108 # Enable /etc/default/docker configuration files with Systemd
109 [Service]
110 EnvironmentFile=/etc/default/docker
111 ExecStart=
112 ExecStart=/usr/bin/dockerd -H fd:// $OPTS
113 EOF
114 fi
115
116 echo ""
117 echo "Restarting Docker"
118 case $INIT_SYSTEM in
119 sysv)
120 service docker restart
121 ;;
122 systemd)
123 systemctl daemon-reload
124 systemctl restart docker.service
125 sleep 1
126 systemctl status --lines=0 docker.service
127 ;;
128 esac
129
130 echo "done."
131

```