Cloud Advanced

Stefano Lusardi

April 2025

1 Exercise 1

To do the first exercise I recycled the setup of the VMs and Docker of the Basic Cloud exam working in just one of the interconnected machine/container.

1.1 Namespace creation

Note: to avoid using sudo on the containers and allow privileged operations I used

```
docker exec --privileged -it stefanolusardi-app1-1 sh
```

The first step is to create the two namespaces:

```
sudo ip netns add ns1
sudo ip netns add ns2
```

1.2 Namespace connection

1.2.1 veth pair

First I create the veth pair simply creating a net interface of type veth (veth0) and a twin interface (veth1)

```
sudo ip link add veth0 type veth peer name veth1
```

Then I move them respectively in ns1 and ns2

```
sudo ip link set veth0 netns ns1
sudo ip link set veth1 netns ns2
```

At this point I can assign the ip addresses to the interfaces

```
sudo ip netns exec ns1 ip addr add 192.168.1.1/24 dev veth0
sudo ip netns exec ns2 ip addr add 192.168.1.2/24 dev veth1
```

and activate them

```
sudo ip netns exec ns1 ip link set veth0 up
sudo ip netns exec ns2 ip link set veth1 up
```

To see that everything works properly I can do

```
sudo ip netns exec ns1 ping -c 3 192.168.1.2
```

1.2.2 bridge

In this section I create a bridge

```
sudo ip link add br0 type bridge
sudo ip link set br0 up
```

and then the two veth pairs (veth2/veth2-br and veth3/veth3-br)

```
sudo ip link add veth2 type veth peer name veth2-br
sudo ip link add veth3 type veth peer name veth3-br
```

As before I move the veth pairs in the namespaces (this time I have two of them, one attached to each ns)

```
sudo ip link set veth2 netns ns1
sudo ip link set veth3 netns ns2
```

and connect the other veth interface to the bridge

```
sudo ip link set veth2-br master br0
sudo ip link set veth3-br master br0
```

Then I assign the ips

```
sudo ip netns exec ns1 ip addr add 192.168.2.1/24 dev veth2
sudo ip netns exec ns2 ip addr add 192.168.2.2/24 dev veth3
```

and activate the interfaces in the veth pairs

```
sudo ip netns exec ns1 ip link set veth2 up
sudo ip netns exec ns2 ip link set veth3 up
sudo ip link set veth2-br up
sudo ip link set veth3-br up
```

To check that everything is working properly

```
sudo ip netns exec ns1 ping -c 3 192.168.2.2
```

1.2.3 VXLAN[1]

In the last section I create 2 bridges

```
sudo ip link add br1 type bridge
sudo ip link add br2 type bridge
sudo ip link set br1 up
sudo ip link set br2 up
```

create and configure the 2 veth pairs connected to the namespaces and the bridges

```
sudo ip link add veth4 type veth peer name veth4-br
sudo ip link set veth4-br up
sudo ip link set veth4-br master br1
sudo ip link set veth4 netns ns1
sudo ip netns exec ns1 ip link set veth4 up
sudo ip netns exec ns1 ip addr add 10.1.1.2/24 dev veth4
sudo ip link add veth5 type veth peer name veth5-br
sudo ip link set veth5-br up
sudo ip link set veth5-br master br2
sudo ip link set veth5 netns ns2
sudo ip netns exec ns2 ip link set veth5 up
sudo ip netns exec ns2 ip addr add 10.1.2.2/24 dev veth5
```

Now it's time to introduce the VXLAN tunnel that is actually composed by 2 VXLANs

```
sudo ip link add vxlan0 type vxlan id 100 local 127.0.0.1 remote
   127.0.0.1 dstport 4789 dev lo
sudo ip link add vxlan1 type vxlan id 101 local 127.0.0.1 remote
   127.0.0.1 dstport 4790 dev lo
sudo ip link set vxlan0 master br1
```

```
sudo ip link set vxlan1 master br2
sudo ip link set vxlan0 up
sudo ip link set vxlan1 up
```

Then I configure the routing based on the ips subdomain I specified before

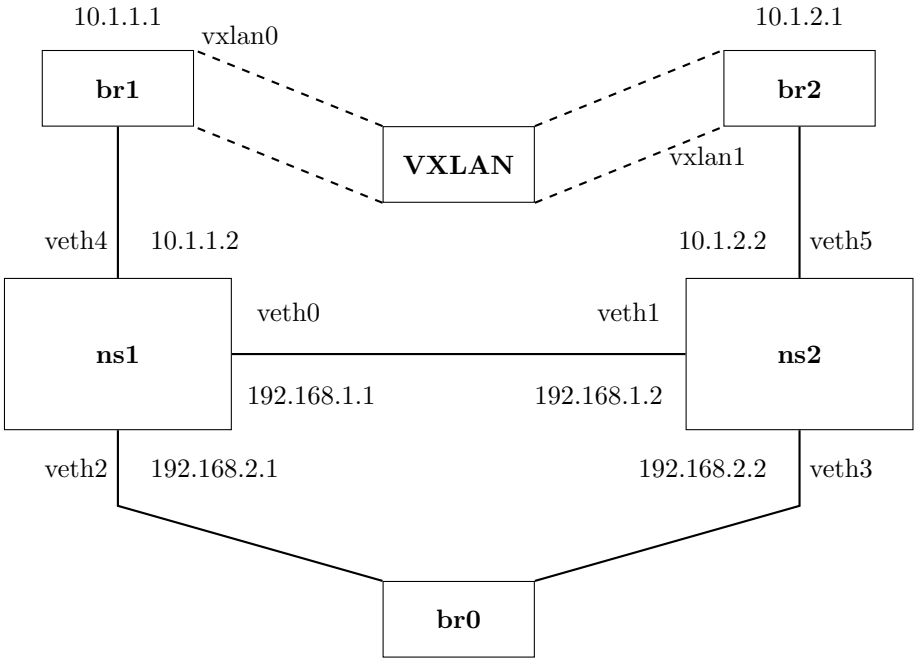
```
sudo ip addr add 10.1.1.1/24 dev br1
sudo ip addr add 10.1.2.1/24 dev br2
sudo ip netns exec ns1 ip route add 10.1.2.0/24 via 10.1.1.1
sudo ip netns exec ns2 ip route add 10.1.1.0/24 via 10.1.2.1
```

and check that everything works properly

```
sudo ip netns exec ns1 ping -c 3 10.1.2.2
```

Note that I had to enable IP forwarding

```
sudo sysctl -w net.ipv4.ip_forward=1
```



1.3 Performance testing

To run the iperf test I executed

```
sudo ip netns exec ns1 iperf3 -s
```

to start the server mode on ns1 and consequently

```
sudo ip netns exec ns2 iperf3 -c 192.168.1.1
sudo ip netns exec ns2 iperf3 -c 192.168.2.1
sudo ip netns exec ns2 iperf3 -c 10.1.1.2
```

The complete results can be found on the Github `https://github.com/onafetsidrasul/Cloud`.
First of all we need to consider the fact that the retransmissions are always smaller compared to the VMs cluster and Docker cluster cases, then we notice that the Docker cluster configuration is still the most efficient in terms of bitrate since it uses the resources in the most efficient way reducing the overhead.
The communication between the virtual machines is still the least efficient. By the way, testing locally the communication between the namespaces on the virtual machine, the throughput is placed in between the two previous cases.
Here the performances are similar one to the other with the veth case that performs slightly better than the bridge case and the vxlan that performs slightly worse: this is due to the increasing of complexity

in virtualization and the packets encapsulation introduced by vxlan.
The same tests in the container highlight the same pattern shifted. The Docker cluster throughput performs better than the namespace setups: the explanation can be the fact that when the containers are communicating they use the host net or something very close to it, while when they internally make namespaces communicate they introduce more massively virtualization overhead.

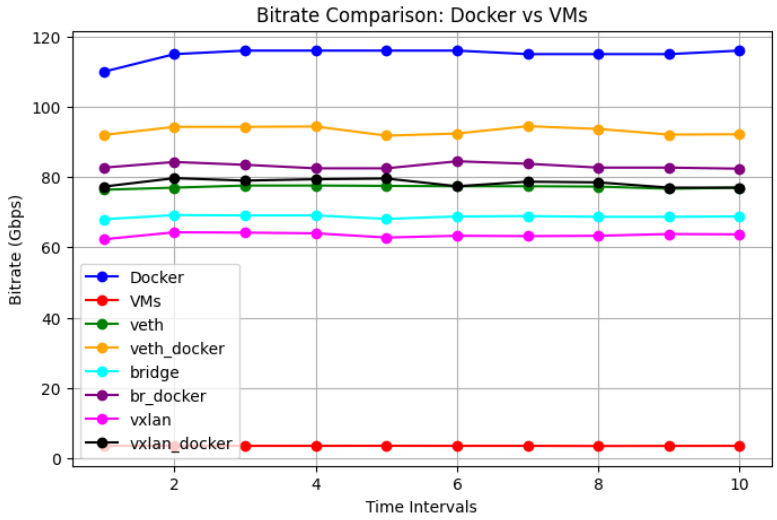


Figure 1: iperf

To measure the latency I simply used ping

```
sudo ip netns exec ns1 ping -c 10 192.168.1.2
sudo ip netns exec ns1 ping -c 10 192.168.2.2
sudo ip netns exec ns1 ping -c 10 10.1.2.2
```

Here we notice that the namespace connection in both the Virtual Machine and the container performs in a comparable way. The latency performances of the Docker containers cluster is comparable with the bridge and vxlan configurations and with the veth for the Docker case, while the VMs cluster performs way worse as we expected.

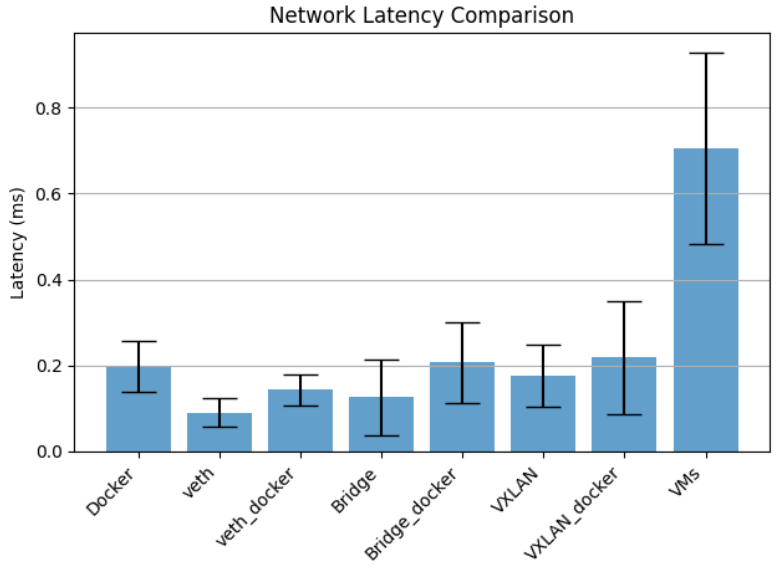


Figure 2: Latency

1.4 Conclusions

The usage of namespaces is motivated by the isolation they implement joined with their security. They are a valuable choice for testing and developing due to the easy way to create and manage them.
In the context of network namespaces they also show to be a good choice in terms of performance, having in mind that the more you virtualize, the more you slow down the performances.
In the sense of bitrate performance, the Docker cluster is still the best choice because it introduces less virtualization overhead.

2 Exercise 2

For this exercise I needed to perform a small vertical scaling of the Vms cluster to meet the requirement of Kubernetes: practically now I'm using 2 cpus and 2048MB of ram for each node. For the master, since it will run also the Kube-Prometheus stack, I use 3 cpus. Without this setting I was getting an error due to the fact that the master hadn't enough cpu to schedule the pod.

2.1 Setup with K8s[2] [3]

First I define the version I want to use

```
export KUBERNETES_VERSION=v1.32
export CRIO_VERSION=v1.32
```

and then I install crio-o, kubelet, kubeadm, and kubectl.

```
sudo apt-get update
sudo apt-get install -y software-properties-common curl gnupg lsb-
    release
sudo curl -fsSL https://pkgs.k8s.io/core:/stable:/$KUBERNETES_VERSION/
    deb/Release.key |    sudo gpg --dearmor -o /etc/apt/keyrings/
    kubernetes-apt-keyring.gpg
sudo echo "deb␣[signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
    ␣https://pkgs.k8s.io/core:/stable:/$KUBERNETES_VERSION/deb/␣/" |
     sudo tee /etc/apt/sources.list.d/kubernetes.list
sudo curl -fsSL https://download.opensuse.org/repositories/isv:/cri-o:/
    stable:/$CRIO_VERSION/deb/Release.key |
     sudo gpg --dearmor -o /etc/apt/keyrings/cri-o-apt-keyring.gpg
sudo echo "deb␣[signed-by=/etc/apt/keyrings/cri-o-apt-keyring.gpg]␣
    https://download.opensuse.org/repositories/isv:/cri-o:/stable:/
    $CRIO_VERSION/deb/␣/" |
     sudo tee /etc/apt/sources.list.d/cri-o.list
sudo apt-get update
sudo apt-get install -y cri-o kubelet kubeadm kubectl
```

To do everything in a smooth way I changed the name of a file

```
sudo -s
root@master:/etc/cni# cd net.d/
root@master:/etc/cni/net.d# ls
10-crio-bridge.conflist.disabled
root@master:/etc/cni/net.d# mv 10-crio-bridge.conflist.disabled 10-crio
    -bridge.conflist
root@master:/etc/cni/net.d# ls
10-crio-bridge.conflist
```

and following professor's Lot tutorial to configure the internal network

```
sudo sed -i 's/10.85.0.0\/16/10.17.0.0\/16/' /etc/cni/net.d/10-crio-
    bridge.conflist
sudo systemctl enable --now crio
sudo systemctl enable --now kubelet
```

basically I have changed the CIDR subnet of the bridge used by cri-o and enabled cri-o and kubelet. Now I have to change the value in the file

```
/proc/sys/net/ipv4/ip_forward
```

to enable the ip forwarding and then

```
sudo kubeadm init --pod-network-cidr=10.17.0.0/16 --apiserver-advertise
   -address=192.168.0.1 --cri-socket=unix:///var/run/crio/crio.sock
```

to initialize the master (control-plane). This last command also prints a token that can be used to later connect the other nodes. Following also the other suggestions this command prints I did

```
mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Now it is time to try to connect the different machines to the master after having installed crio-o, kubelet, kubeadm, and kubectl. Note that also here the change of the file specified above (to enable ip forwarding) is necessary as well as the change of the name and the internal net configuration.
To connect the machines I simply used the token generated initializing the control-plane specifying that i want to use crio.

At this point I need to install the kubernetes network, I used Calico starting from the manifest.

```
curl https://calico-v3-25.netlify.app/archive/v3.25/manifests/calico.
   yaml -O
```

Here I need to modify the part concerning CALICO IPV4POOL CIDR.

```
- name: CALICO_IPV4POOL_CIDR
              value: "10.17.0.0/16"
```

using the ip I want to use.

To check that everything is working correctly I created 2 simple alpine pods on the 2 nodes

```
kubectl run ping-test-1 --image=alpine:latest --overrides='{"spec":␣{"
   nodeName":␣"node01"}}' -- sleep 3600
kubectl run ping-test-2 --image=alpine:latest --overrides='{"spec":␣{"
   nodeName":␣"node02"}}' -- sleep 3600
```

on 2 different terminal windows

```
kubectl exec -it ping-test-1 -- /bin/sh
kubectl exec -it ping-test-2 -- /bin/sh
```

after having had a look at the ip of the pods with

```
kubectl get pods -o wide
```

on pod 2 I do

```
nc -l -p 8888
```

and on pod 1

```
echo "Hello␣from␣pod␣1" | nc <ip_of_pod2> 8888
```

2.2 Kube-Prometheus stack

To get the Kube-Prometheus stack I installed Helm and added the repository

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-
    helm-3 | bash
helm repo add prometheus-community https://prometheus-community.github.
    io/helm-charts
helm repo update
```

I create the monitoring namespace

```
kubectl create namespace monitoring
```

and install everything

```
helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace monitoring \
  --set grafana.adminPassword=admin \
  --set prometheus.prometheusSpec.retention=10d
```

To visualize the dashboards I access to the master using

```
ssh -L 9090:localhost:9090 -p 2222 vboxuser@127.0.0.1
```

I start the port forwarding

```
kubectl port-forward -n monitoring svc/prometheus-operated 9090:9090
```

and on my browser I go to

```
http://localhost:9090
```

This is for Prometheus but it is similar for Grafana:

```
ssh -L 3000:localhost:3000 -p 2222 vboxuser@127.0.0.1
```

```
kubectl port-forward -n monitoring svc/prometheus-grafana 3000:80
```

```
http://localhost:3000
```

2.3 HPCC pods

To create the 3 pods on the 3 nodes (I included the master as I did in the cloud basic configuration) I wrote a yaml file you can find in the repository. As in the basic configuration each pod has a container that uses 1 cpu and 1024 MB of Ram.
NOTE: here a crucial role is played by the lines

```
securityContext:
      privileged: true
      capabilities:
        add: ["NET_RAW"]
```

that allow to use ssh and mpi.
To make it work also on the master I removed the taint that prevent the pods to be run on it.

```
kubectl taint nodes master node-role.kubernetes.io/control-plane:
    NoSchedule-
```

At this point

```
kubectl apply -f hpl-pods.yaml
```

and it is possible to enter in the pods

```
kubectl exec -it hpl-pod1 -- bash
```

(changing the number I can enter the different ones). Inside the pods I proceeded installing the required packages

```
apt-get update
apt-get install -y build-essential libopenmpi-dev openmpi-bin wget
    gfortran openssh-server net-tools iputils-ping
```

and hpcc from apt

```
apt-get install -y hpcc
```

After having configured the ssh in order to make the communication happen everything is ok to start the tests.

2.4 HPCC tests

To distribute the workloads I create a simple hostfile

```
10.17.219.76 slots=1
10.17.196.147 slots=1
10.17.140.94 slots=1
```

Then I recycled the hpccinf.txt files from the basic part and run the test for the cluster

```
mpirun --allow-run-as-root --hostfile hostfile -np 3 --mca btl ^vader
    --mca btl_tcp_if_include eth0 hpcc
```

and for the pods.
Let us have a look at the results of the HPL test:

	cluster	**master**	**node01**	**node02**
GFLOPS/s VM	70.4	47.2	47.0	47.1
GFLOPS/s Kubernetes	36.4	37.1	35.7	37.0

As we can notice there isn't a convenience in terms of performance in using the cluster rather than the single node in Kubernetes. This could be explained by the fact that in this scenario the communication overhead enters more massively compared to the simpler (with less virtualization) VMs case.
Comparing the VMs and the Kubernetes performance on the single machines we notice that the performances are similar with the VMs that perform slightly better.
Complete hpcc results can be found in the repository.
Looking at the alert manager, the alert I get is

```
0
annotations
description     "Cluster␣␣has␣overcommitted␣CPU␣resource␣requests␣for␣
    Pods␣by␣0.60␣CPU␣shares␣and␣cannot␣tolerate␣node␣failure."
runbook_url     "https://runbooks.prometheus-operator.dev/runbooks/
    kubernetes/kubecpuovercommit"
summary "Cluster␣has␣overcommitted␣CPU␣resource␣requests."
```

This is an indicator of the fact that the cluster is running out of cpus. By the way to have comparable results I decided to not change the configuration. An alternative would have been to perform a further vertical scaling, but at this point this would have meant to use half of my computer resources.
Luckily, I did not experience any node failure.

We can also look at the behavior of the nodes using the Kube-Prometheus stack.
Since the graphs show the same thing (the query is the same for both Grafana and Prometheus) I report here sometimes the Prometheus and sometimes the Grafana screenshot, you can find some of the others

in the repository.
For the single pod metrics, I report just the ones from pod 1 for brevity, the others are very similar.

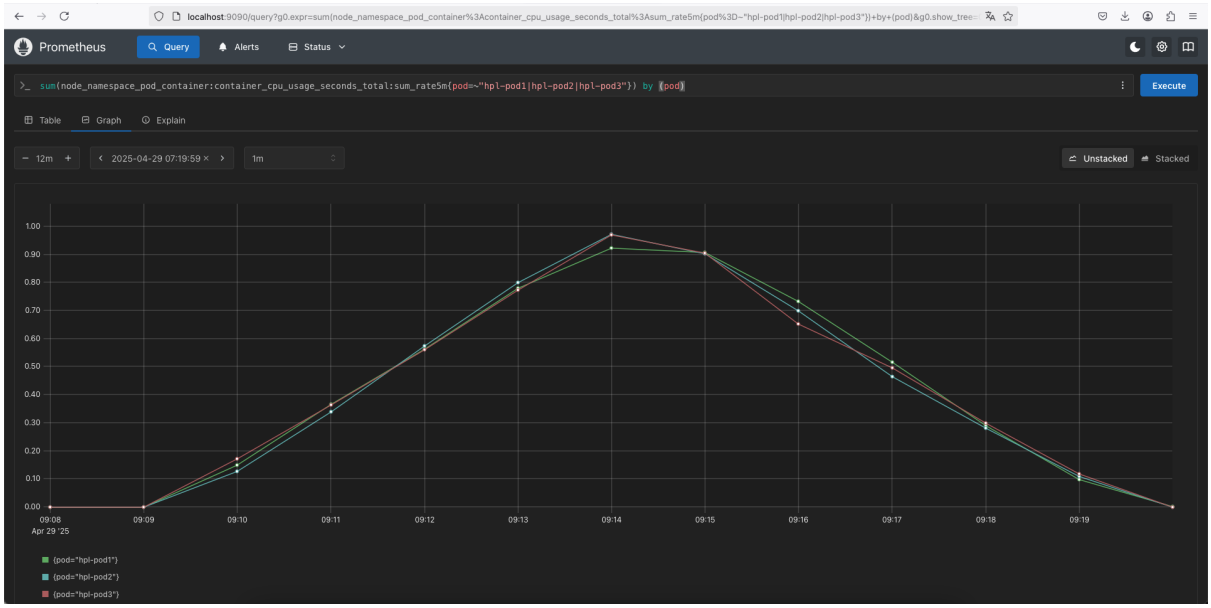


Figure 3: CPU usage for **cluster**

The 3 pods reach a very high use of the cpu during the test, this is probably related to the alert I was getting and can link the performance degradation to resource overcommitment.

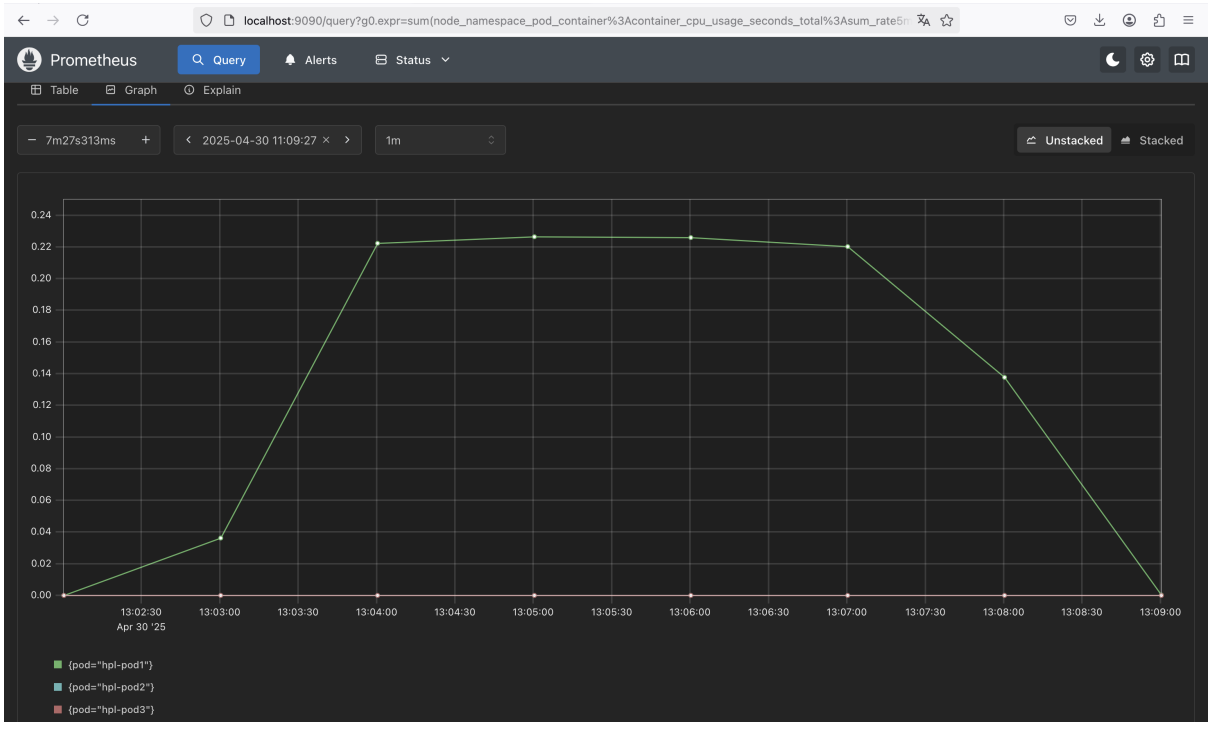


Figure 4: CPU usage for **pod 1**

The single pod uses roughly a quarter of the cpu used in the cluster setting.

Figure 5: Memory usage for **cluster**

The memory usage is quite low compared to the limits I imposed.

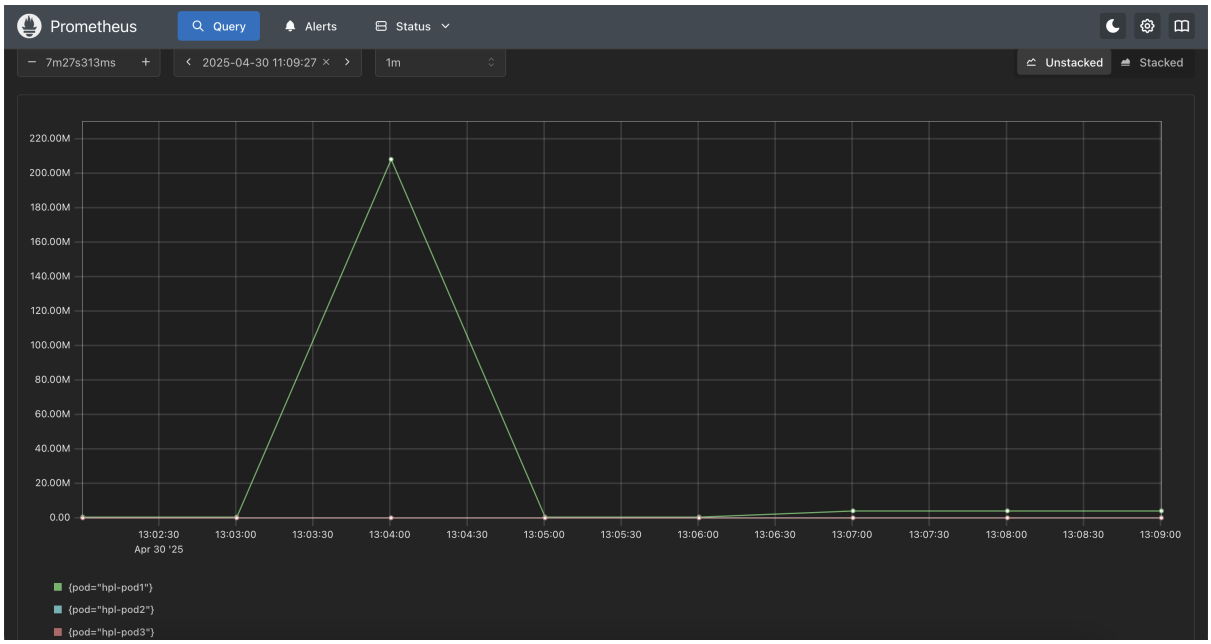


Figure 6: Memory usage for **pod 1**

With respect to the cluster setting we can notice an order of magnitude higher use of the memory in the single node test. By the way the amount of memory usage is still reasonably below the limit I imposed (less than 0.25 of the limit).

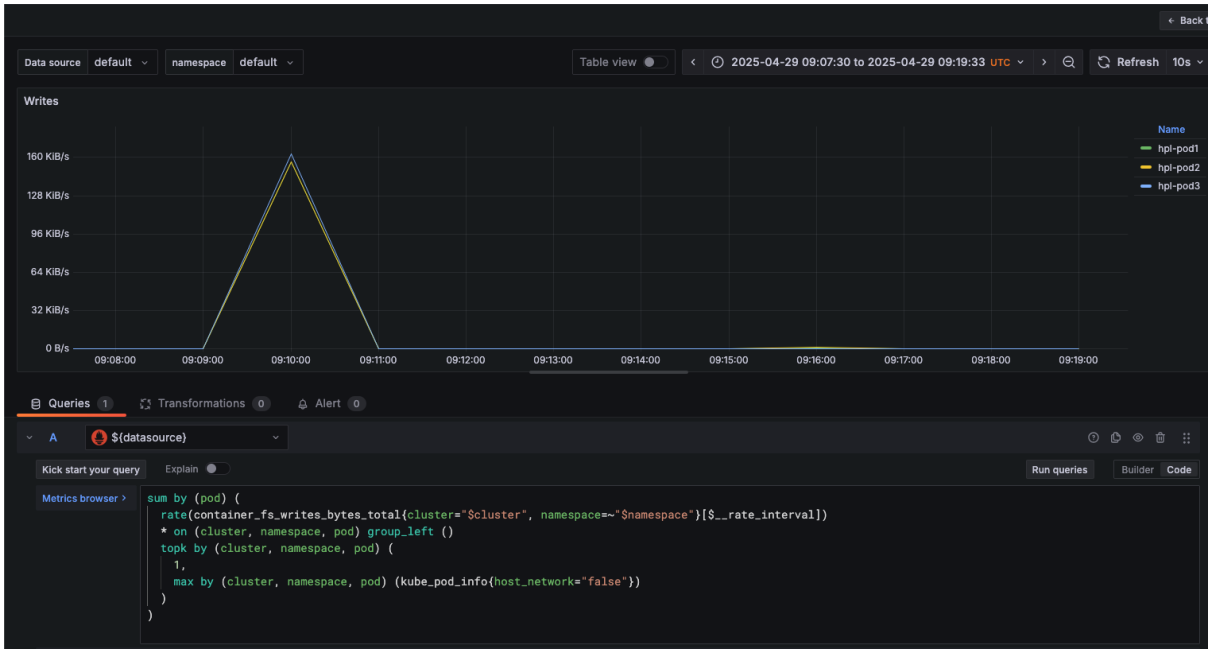


Figure 7: Write speed for the **cluster**

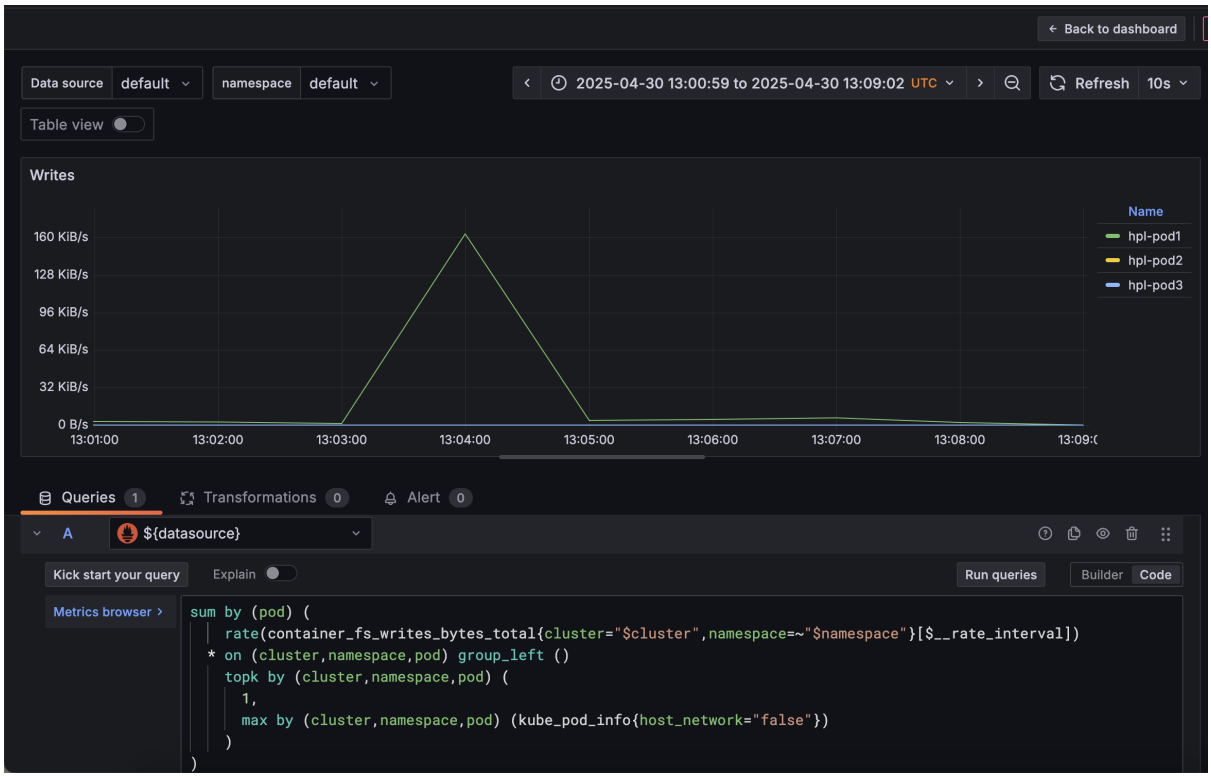


Figure 8: Write speed for **pod1**

The write speed in the 2 cases is very close.

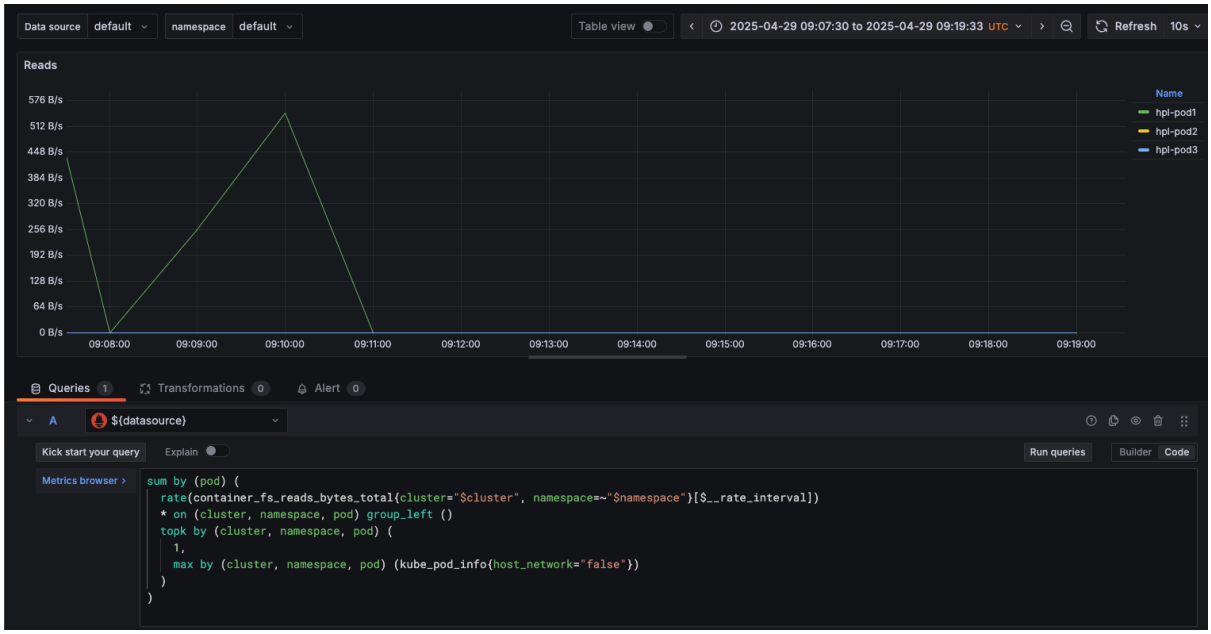


Figure 9: Read speed for **cluster**

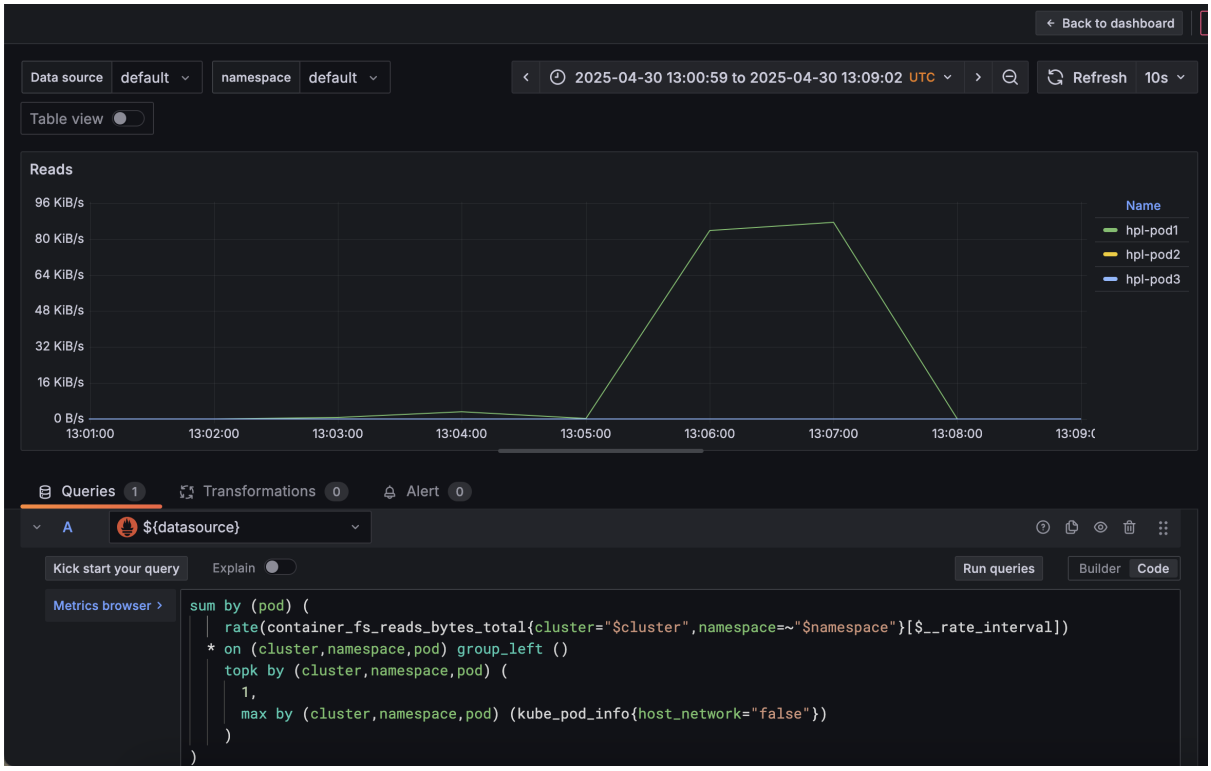


Figure 10: Read speed for **pod1**

In this case the read speed of the cluster is very low compared to the one of the single pod.

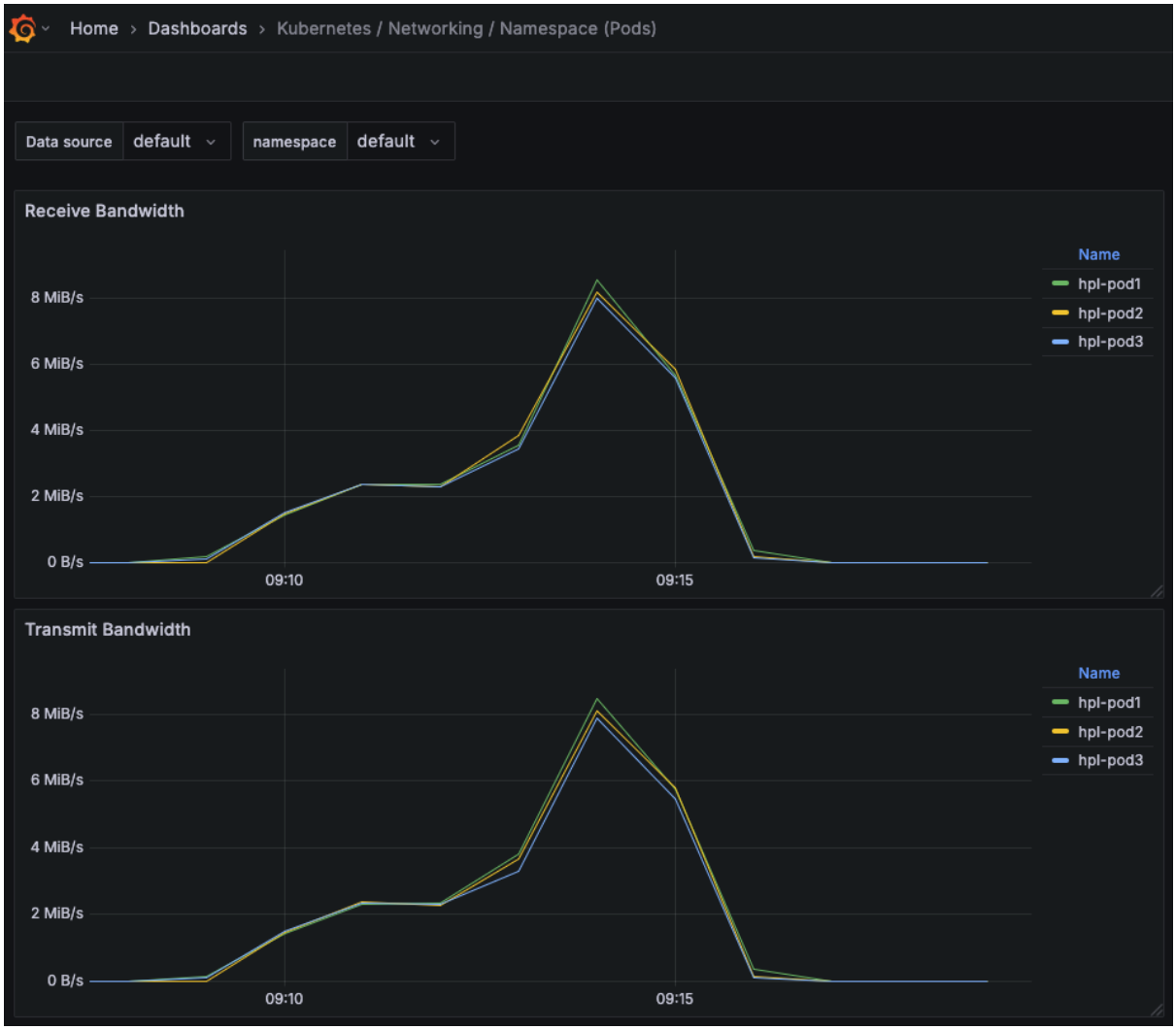


Figure 11: Receive and Transmit bandwidth for the **cluster**

2.5 Conclusions

The main bottleneck seems to be the one related to the cpu, as also pointed out by the Alert manager, running with more cpu resources on the VM node would help in having a less stressed environment and probably also in getting better results.
Even though the Kuberenetes cluster is less performing compared to the Virtual Machines setup (on which it is build), it is still a good choice in terms of portability, scalability, orchestration, networking, and for the richness of its ecosystem. The Kube-Prometheus stack, indeed, is a very powerful tool that allows for monitoring the resources in a complete way.
It should be also pointed out that running a Kubernetes cluster directly on a Linux cluster (not on a virtual machines one) could improve the overall performance.

References

[1] Shofiur. *"Creating a Simple VXLAN Overlay Network using Linux Network Namespaces and Bridges"*. 2023. URL: `https://medium.com/@mdshofiur/creating-a-simple-vxlan-overlay-network-using-linux-network-namespaces-and-bridges-7116039b4882`.

[2] CRI-O Project Authors. *"CRI-O, Lightweight Container Runtime for Kubernetes"*. URL: `https://cri-o.io/`.

[3] kubernetes. *"Creating a cluster with kubeadm"*. URL: `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/`.