DMAC-7 - Dynamic MAC allocation to Pods Practical Study

Kacper Karafa
Adam Klekowski
Filip Tłuszcz
Michał Szczurek

Środowiska udostępniania usług 2023, group 6

Table of contents

1. Introduction

Nowadays, applications are often designed as a collection of smaller services that are independently deployable and loosely coupled [1]. In order to assure portability of the services, they are usually containerized with use of tools such as Docker [2] and kept as images. Images are then being used to define pods which are maintained and orchestrated by external applications like Kubernetes [3] and Nomad [4]. Those applications also provide virtual networks that allow containerized services to communicate with each other with use of classical internet protocols.

In this raport we will focus on achieving dynamic MAC address allocation for pods. In order to achieve this we attempted to use Multus [5] with Kubernetes plugin alongside DPDK [6], however in the final version GNU MAC Changer [15] is used instead of DPDK.

2. Theoretical background/technology stack

In this section we will describe a few of the technologies that were researched and/or used to achieve dynamic MAC address allocation, including Multus CNI, Data Plane Development Kit (DPDK) and GNU MAC Changer.

Multus CNI

Multus CNI is a container network interface (CNI) plugin for Kubernetes that enables attaching multiple network interfaces to pods. Typically, in Kubernetes each pod only has one network interface (apart from a loopback) - with Multus you can create a multi-homed pod that has multiple interfaces. Additional network interfaces have numerous uses such as: multi-tenant networks, performance and security enhancement via traffic isolation, VPN applications requiring additional encryption [7].

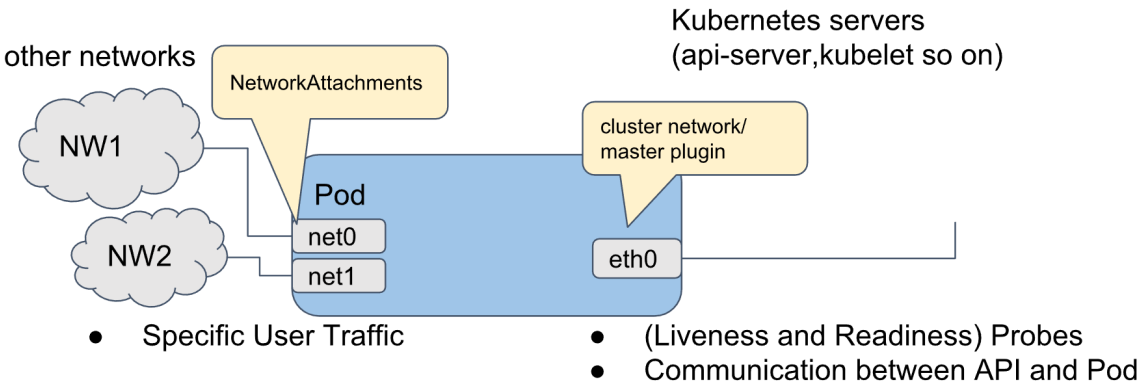

Figure 1. Example configuration utilizing Multus, source: Multus CNI Github page [5]

Multus follows Kubernetes Network Custom Resource (CDR) Definition which provides a standardized method by which to specify the configurations for additional network interfaces. CDRs can be easily defined in yaml format.

Important thing to note is to

DPDK

DPDK is the Data Plane Development Kit that consists of libraries to accelerate packet processing workloads running on a wide variety of CPU architectures. It provides a set of libraries and network interface controller polling-mode drivers for offloading packet processing from the operating system kernel to processes running in user space. This offloading achieves higher computing efficiency and higher packet throughput than is possible using the interrupt-driven processing provided in the kernel. The framework creates

a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL). These environments are created through the use of meson files and configuration files. Once the EAL library is created, the user may link with the library to create their own applications. DPDK provides a set of powerful utilities, which include changing network interface MAC addresses as presented in sample application [8]. However, from the perspective of project completion we can indicate that set up, deployment and usage of DPDK MAC changing capabilities turns out to be problematic due to the fact that it requires the highest possible superuser privileges to make it run, which are not always possible to achieve in a cloud environment. Moreover certain functionalities require configuration of hugepages [18] which also turns out to be problematic.

Macchnager

GNU MAC Changer is a open-source simple command line utility [15] that allows for manipulation of MAC addresses of network interfaces. It is usually distributed alongside Linux distribution, e. g. it is available in the official Ubuntu PPA repository (checked on `focal`). The software offers following features:

1. setting specific MAC address of a network interface
2. setting the MAC randomly
3. setting a MAC of another vendor (there is a list of 6200 vendors available)
4. setting a MAC of the same kind (e.g. wireless interfaces)

Inspection of the source code [17] indicates that `macchanger` shutdowns given interface, via `sys/ioctl` (`ioctl` system call) interface [16] modifies device data and finally powers up the interface. Thus usage of this tool is not portable and will work only on the Unix system family.

Conclusion

Thanks to the Multus CNI plugin, users can add additional pod's network interfaces for dedicated use cases. GNU MAC Changer then allows for customization of created interfaces MAC addresses from inside a pod. Combining those two technologies allows users to perform dynamic MAC allocation to pods. DPDK is an extensive data processing framework and in theory could be used for dynamically changing MAC address of a network interface, however it requires specific privileges and setup from the host environment.

3. Case study concept description

In order to test and present the functionality of dynamic MAC address allocation we propose the following environment:

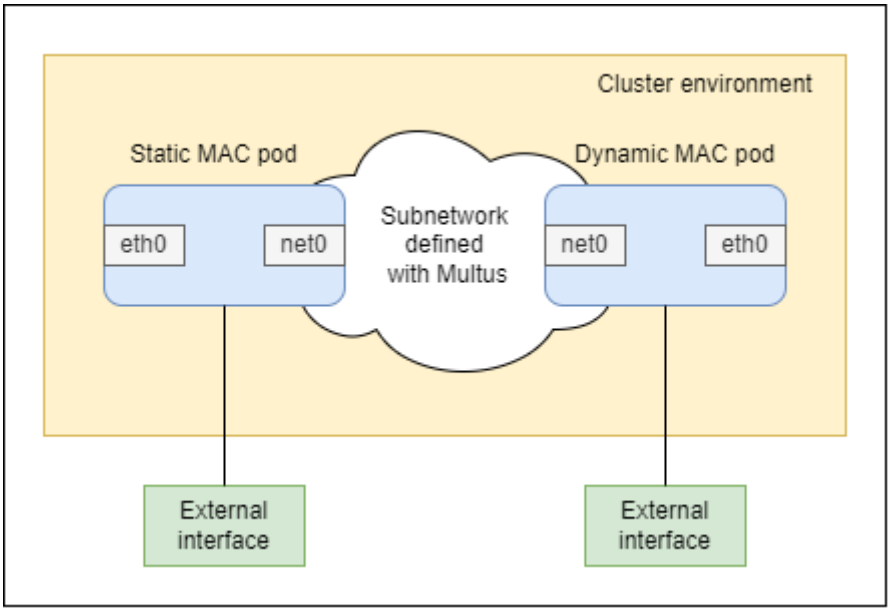

Figure 2. Case study environment.

The configuration consists of two pods, each containing one additional network interface defined using Multus CNI used for pod to pod communication. Pods are connected via subnetwork created as Multus NetworkAttachmentDefinition. The *Static MAC pod* is used to send packets to the *Dynamic MAC pod* and await for response. The sent packet content and receiver MAC can be set using an external interface - perhaps a dedicated endpoint using standard network interface exposed outside of the cluster. Such interface also allows to view information sent back from *Dynamic MAC* pod. Dynamic *MAC* pod performs simple operation on a packet and sends it back. It also comes with an attached external interface that allows setting the net0 MAC address in real time. The eth0 is a non-dynamic utility network interface assigned via Kubernetes network plugin e.g. Weave-net [10]

This configuration shows how one can take advantage of Multus CNI and GNU MAC Changer to create a pod with a dynamic MAC address. The correctness of the setup is verified with packets received by *Static MAC pod.*

4. Solution architecture

1. Technology stack

As per technologies, we chose Kubernetes, for creating two, communicating pods, using Docker for containerisation purposes. Those pods will use earlier described GNU MAC Changer utility for configuring network settings and Multus CNI for attaching multiple network interfaces to the pods.

2. Demo architecture

When it comes to how the POC will proceed, the first pod will send packets to the second one, then the second one will change its MAC address. After that, the first pod will try to send packets again and if it succeds, it would mean that the operation of dynamic MAC change was successful.

5. Environment configuration description

Kubernetes

To have better control over the experiment environment we decided to step off from cloud environment (first tests were conducted on Google Cloud Platform with usage of Google Kubernetes Engine, also AWS was tested) and deploy our solutions in local Kubernetes environment using Minikube [20].

Install and configure Minikube

1. Download Minikube binary. Remember to download binary that is appropriate to your platform. Here we download `minikube-linux-amd64`. Please refer to [21].

```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

2. System wide installation

```
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

After this step `minikube` should be available as a standalone command and possible to start:

3. Starting Minikube with `calico` plugin [22]

```
minikube start –cni=calico
```

We can now run our `kubectl` commands in the Minikube environment.

Multus CNI

Install and configure Multus

Deploying Multus using a Daemonset will spin up pods that install a Multus binary and configure Multus for usage in every node in the cluster. Here are the steps for doing that. [9]

1. Clone the Multus CNI repo:

```
git clone https://github.com/k8snetworkplumbingwg/multus-cni.git && cd multus-cni
```

2. Apply Multus daemonset to your cluster:

```
kubectl apply -f ./deployments/multus-daemonset-thick-plugin.yml
```

3. Verify that you have Multus pods running:

```
aklekowski@VivoBook:~$ kubectl get pods --all-namespaces | grep -i multus
kube-system   kube-multus-ds-45rj7                          1/1     Running   0          19d
kube-system   kube-multus-ds-d486j                          1/1     Running   0          19d
```

Create Network Attachment Definition

You need to create a Network Attachment Definition for the CNI you wish to use as the plugin for the additional interface. You can verify that your intended CNI plugin is supported by ensuring that the binary corresponding to that CNI plugin is present in the node's `/opt/cni/bin` directory. [9]

```
cat <<EOF | kubectl create -f -
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
   name: ipvlan-conf
spec:
   config: '{
      "cniVersion": "0.3.0",
      "type": "ipvlan",
      "master": "eth0",
      "mode": "l3",
      "ipam": {
         "type": "host-local",
         "subnet": "198.17.0.0/24",
         "rangeStart": "198.17.0.200",
         "rangeEnd": "198.17.0.216",
```

```
        "routes": [
            { "dst": "0.0.0.0/0" }
        ],
        "gateway": "198.17.0.1"
      }
 }'
EOF
```

Note that `ens4` is used as the master parameter. This master parameter should match the interface name on the hosts in your cluster. [9]

Verify the configuration

```
aklekowski@VivoBook:~$ kubectl describe network-attachment-definitions ipvlan-conf
Name:         ipvlan-conf
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  k8s.cni.cncf.io/v1
Kind:         NetworkAttachmentDefinition
Metadata:
  Creation Timestamp:  2023-04-08T17:08:40Z
  Generation:          1
  Managed Fields:
    API Version:  k8s.cni.cncf.io/v1
    Fields Type:  FieldsV1
    fieldsV1:
      f:spec:
        .:
        f:config:
    Manager:         kubectl-create
    Operation:       Update
    Time:            2023-04-08T17:08:40Z
  Resource Version:  25606
  UID:               93b0884c-d1f2-4801-a254-cff80a0f792e
Spec:
  Config:  { "cniVersion": "0.3.0", "type": "ipvlan", "master": "eth1", "mode": "l3", "ipam": { "type": "host-local", "s
ubnet": "198.17.0.0/24", "rangeStart": "198.17.0.200", "rangeEnd": "198.17.0.216", "routes": [ { "dst": "0.0.0.0/0" } ],
 "gateway": "198.17.0.1" } }
Events:    <none>
```

Deploy sample application with network attachment

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: app1
  annotations:
    k8s.v1.cni.cncf.io/networks: ipvlan-conf
spec:
  containers:
  - name: app1
    command: ["/bin/sh", "-c", "trap : TERM INT; sleep infinity & wait"]
    image: alpine
EOF
```

```
aklekowski@VivoBook:~$ kubectl exec -it app14 -- ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0@if99: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1460 qdisc noqueue state UP
    link/ether b2:2a:b8:71:35:92 brd ff:ff:ff:ff:ff:ff
    inet 10.44.1.97/24 brd 10.44.1.255 scope global eth0
       valid_lft forever preferred_lft forever
3: net1@eth0: <BROADCAST,MULTICAST,NOARP,UP,LOWER_UP> mtu 1460 qdisc noqueue state UNKNOWN
    link/ether 42:01:0a:80:00:04 brd ff:ff:ff:ff:ff:ff
    inet 198.17.0.200/24 brd 198.17.0.255 scope global net1
       valid_lft forever preferred_lft forever
```

Base container

As a base container image for our solution we use an Ubuntu 20.04 [11] container. It was additionally configured with required network utilities and macchanger. Below you can see the Dockerfile describing the image:

```
FROM ubuntu:20.04

RUN apt-get -qq update \
  && apt-get -qq install --no-install-recommends \
       ca-certificates \
       wget \
       xz-utils \
       net-tools \
       pciutils \
       iproute2 \
       sudo \
       ethtool \
       lshw \
       macchanger \
       iputils-ping \
```

The Dockerfile is also available on our GitHub repository.

6. Installation method

There are no additional installation steps required beside appropriate environment configuration, as our solution does not require installing or building any additional artifacts.

7. How to reproduce - step by step

7.1 Solution I - Multus only solution

One of our initial solutions only required a Multus CNI plugin to be installed - no additional requirements and privileges for the container are needed. Although it does not allow for dynamically updating pod's MAC it is capable of setting one on pods start. We believe that it may be of some use and therefore find it worthy of presenting.

The solution consists of two steps:

- Preparing Multus network definition that allows mac provisioning
- Setting up a pod definition with desired mac

Full yaml files can be found on the project's Github.

The most important parts of the network attachment definition are:

```
  "capabilities": { "ips": true },
  "capabilities": { "mac": true },
```

which allows provisioning desired MAC and IP addresses on pod creation. It is also crucial to set correct network type:

```
 "type": "macvlan",
```

"macvlan" type results in assigning a MAC address to each container's virtual network interface, making it appear to be a physical network interface directly connected to the physical network.

Then, once defining Kubernetes pod, onc has to include the following annotation:

```
k8s.v1.cni.cncf.io/networks: '[
            { "name": "macvlan-confifg",
              "ips": [ "10.1.1.101/24" ],
              "mac": "c2:b0:57:49:47:f1"
            }]'
```

The ip range and desired mac can be set up according to needs. It can then be applied and verified using the following commands:

```
$ kubectl apply -f network-attachment.yaml
$ kubectl apply -f sample-pod.yaml
$ kubectl exec -it multus-only-pod /bin/bash
$ ip addr
```

The command should have output similar to the following:

```
root@multus-only-pod:/# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
4: eth0@if29: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1480 qdisc noqueue state UP group default
    link/ether 6a:74:5c:ff:45:39 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.244.120.87/32 scope global eth0
       valid_lft forever preferred_lft forever
5: net1@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether c2:b0:57:49:47:f1 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.1.1.101/24 brd 10.1.1.255 scope global net1
       valid_lft forever preferred_lft forever
```

This solution is the simplest way of creating a pod with a desired MAC address via Multus CNI. It also allows attaching a specific ip or ip range. Nonetheless, it does not provide users with a possibility of modifying MAC addresses during pod runtime.

7.2. Solution II - with use of GNU MAC Changer

Second solution takes advantage of the earlier described `macchanger` CLI tool. Although It requires additional privileges for pods, it allows dynamically changing MAC addresses from within a pod. The solution consists of deployment definition and the network attachment definition. It can be fully examined on the project's Github.
The network attachment is similar to the one previously described, but it is simplified as there is no need to allow setting MAC on port start - it can be done at any time. Once again, macvlan network type ought to be used.

As for deployment, this time we have specified replicas number to two which comes handy while demonstrating the solution. Created pods use ra77us/suu:1.0.0 image, which is based on Ubuntu 20.04 enhanced with network utilities. Its Dockerfile can be seen on the project's Github. Important difference between this definition and the one from the first solution is the need of additional capabilities:

```
securityContext:
          capabilities:
            add: ["NET_ADMIN"]
```

It grants additional network permissions, which are required in order to use the macchanger CLI tool. In order to verify the solution, one can perform the following scenario:

1. Creating required resources

```
$ kubectl apply -f network-attachment.yaml
$ kubectl apply -f sample-pod.yaml
$ kubectlget pods
```

The output should be similar to:

```
dynamic-mac-58d9b97948-j8stl              1/1     Running   0              21h
dynamic-mac-58d9b97948-tmtlt              1/1     Running   0              21h
```

3. Verifying pods network interface

```
$ kubectl exec -it dynamic-mac-58d9b97948-j8stl /bin/bash
$ ip addr
```

The output should be similar to:

```
5: net1@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
    link/ether d2:e0:a4:66:90:ea brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 198.17.0.204/24 brd 198.17.0.255 scope global net1
       valid_lft forever preferred_lft forever
```

4. Changing pod's MAC address

In the above example, the current MAC address is set to d2:e0:a4:66:90:ea. We will demonstrate how it can be changed to d2:00:11:22:33:44.

```
$ macchanger -m d2:00:11:22:33 net1
```

Then it can be verified that the address has changed.

```
$ ip addr
```

The output should be similar to:

```
5: net1@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether d2:00:11:22:33:44 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 198.17.0.204/24 brd 198.17.0.255 scope global net1
       valid_lft forever preferred_lft forever
```

5. Checking the connection

In other to ensure that the interface still works correctly, one can ping the pod whose MAC was changed from the other deployed instance:

```
$ exit
$ kubectl exec dynamic-mac-deployment-58d9b97948-hmtk8 ping 198.17.0.204
```

The output should be similar to:

```
PING 198.17.0.204 (198.17.0.204) 56(84) bytes of data.
64 bytes from 198.17.0.204: icmp_seq=1 ttl=64 time=0.169 ms
64 bytes from 198.17.0.204: icmp_seq=2 ttl=64 time=0.055 ms
64 bytes from 198.17.0.204: icmp_seq=3 ttl=64 time=0.041 ms
64 bytes from 198.17.0.204: icmp_seq=4 ttl=64 time=0.038 ms
64 bytes from 198.17.0.204: icmp_seq=5 ttl=64 time=0.051 ms
```

The above confirms proper configuration - host is reachable from the network and MAC was properly changed.

8. Summary – conclusions

Throughout the development process, we have considered three solutions, two of which were implemented and tested:

- Combining Multus and DPDK to achieve dynamic MAC allocation (*unimplemented*)
- Using Multus to allow for setting MAC on pod's start
- Combining Multus and macchanger to achieve dynamic MAC allocation

We have decided to drop the first solution, nonetheless, it was a valuable opportunity to take a better look at working with DPDK, as we have managed to achieve dynamic MAC allocation with DPDK on Docker containers. DPDK requires additional permissions and resources which may not be trivial to provision in the Kubernetes environment. Having realized that DPDK offers far more powerful features than only changing MAC addresses, we have decided to look for more lightweight solutions that would also satisfy the requirement of dynamic MAC allocation. This approach has led us to using macchanger - a much simpler tool, capable of performing all the required changes.

Along the way, we have also had the opportunity to learn about Multus CNI. Although Multus is easy to use, it provides powerful utilities for complex defining network structure - we have managed to create a 'semi-working solution' capable of setting pod's MAC on start using Multus alone.

Overall, we have managed to achieve the desired functionality of dynamic MAC allocation, as well as, to deepen our knowledge regarding technologies used for networking and cloud computing.

9. References

[1] *What are microservices?*, https://microservices.io/. Accessed 30 March 2023
[2] *Docker* https://www.docker.com/. Accessed 30 March 2023.
[3] *Kubernetes*, https://kubernetes.io/. Accessed 30 March 2023.
[4] *Nomad by HashiCorp*, https://www.nomadproject.io/. Accessed 30 March 2023.
[5] *Multus CNI*, https://github.com/k8snetworkplumbingwg/multus-cn Accessed 30 March 2023.
[6] *DPDK*, https://www.dpdk.org/. Accessed 30 March 2023.
[7] *Did you know? You can run multiple network plugins in Kubernetes with Multus*, cloudification.io. Accessed 30 March 2023.

[8] *Ethtool Sample Application*, https://doc.dpdk.org/guides/sample_app_ug/flow_classify.html. Accessed 30 March 2023.
[9] Multus CNI plugin configuration *https://anywhere.eks.amazonaws.com/docs/tasks/cluster/cluster-multus/*
[10] Weave Net *https://www.weave.works/docs/net/latest/kubernetes/kube-addon/*
[11] Ubuntu *https://ubuntu.com/*
[12] Ethtool *https://linux.die.net/man/8/ethtool*
[13] Pci Utils *https://github.com/pciutils/pciutils*
[14] Virtual function I/O *https://docs.kernel.org/driver-api/vfio.html*
[15] GNU MAC Changer, https://github.com/alobbs/macchanger
[16] ioctl - control device system call https://man7.org/linux/man-pages/man2/ioctl.2.html
[17] GNU MAC Changer implementation details https://github.com/alobbs/macchanger/blob/master/src/netinfo.c#L71
[18] Debian Wiki - hugepages https://wiki.debian.org/Hugepages
[19] Multis CNI, MAC address set in conf., https://github.com/k8snetworkplumbingwg/multus-cni/blob/master/examples/macvlan-pod.yml
[20] Minikube official docs https://minikube.sigs.k8s.io/docs/
[21] Minikube installation instructions https://minikube.sigs.k8s.io/docs/start/
[22] Calico https://docs.tigera.io/calico/latest/about/about-calico
[23] Dockerfile for base image https://github.com/kkafar/suu-dynamic-mac/blob/main/base-image/Dockerfile