ZSUT

Zakład Sieci i Usług Teleinformatycznych

Kubernetes laboratory

Lab part 2: Kubernetes networking

Prepared by: dr inż. Dariusz Bursztynowski

Acknowledgements: The author is grateful to the following students for their help in preparing the lab (in alphabetical order): Hubert Daniłowicz, Franciszek Dec, Jerzy Jastrzębiec-Jankowski, Maciej Maliszewski, Miłosz Marchewka, Adrian Osędowski, Cezary Osuchowski, Piotr Polnau, Jan Sosulski, Filip Wrześniewski, Marta Zielińska.

ZSUT. Zakład Sieci i Usług Teleinformatycznych
Instytut Telekomunikacji
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Last update: February 2025

WARNING: Before powering up your devices, please make sure you are using **the correct power adapter**. The power adapters in the boxes have the same DC plug type, but they are **SIGNIFICANTLY DIFFERENT** in terms of output voltage (WiFi routers as Linksys are powered by 12V DC, while the TP-Link switch requires 53V DC). **Powering our WiFi router from the TP-Link switch power adapter will immediately damage the router.**

Table of contents

1. Introduction

In this lab, we focus on selected aspects of Kubernetes networking. In particular, we will investigate the concept of CNI based on the flannel CNI, install and configure MetalLB load balancer, check its operation and enable external access to the dashboard of Traefik[1] ingress controller, and analyse the role of kube-proxy in traffic routing by observing the operation of in-cluster load balancing (i.e., load balancing among pod replicas of a given service).

2. Kubernetes networking with flannel CNI

In this section, we investigate the details of k3s networking using flannel as the CNI in use. Flannel Intro on Kubernetes networking can be found here: https://www.tigera.io/learn/guides/kubernetes-networking/. A detailed description of what flannel does when a pod is created is given here: https://www.henrydu.com/2020/11/16/k3s-cni-flannel/.

In the following, we first quickly check if flannel has been installed correctly and then run a series of experiments to see the role of flannel in typical scenarios of pod communication.

2.1. Checking the correctness of flannel installation

Kubernetes defines a network model called the container network interface (CNI), but the actual implementation relies on network plugins. The network plugin is responsible for allocating internet protocol (IP) addresses to pods and enabling pods to communicate with each other within the Kubernetes cluster. There are a variety of network plugins for Kubernetes, but this section will use Flannel which is the default CNI in Kubernetes and for k3s in particular. Flannel is very simple and uses a Virtual Extensible LAN (VXLAN) overlay by default.

After installing the k3s with flannel enabled, run a couple of quick checks to see if the basic connectivity works properly.

a) Checking k3s and flannel versions (your versions may differ)

```
ubuntu@kpi091:~$ k3s -v
k3s version v1.24.3+k3s1 (990ba0e8)
go version go1.18.1
```

```
ubuntu@kpi091:~$ /var/lib/rancher/k3s/data/*/bin/flannel
CNI Plugin flannel version v0.18.1 (linux/arm64) commit
990ba0e88c90f8ed8b50e0ccd375937b841b176e built on 2022-07-19T01:08:03Z
```

The above confirms flannel has been installed.

b) Check if DNS is working

Use this link: https://kubernetes.io/docs/tasks/administer-cluster/dns-debugging-resolution/

2.2. Basic pod communication checks and related tips (subjective selection)

This section is auxiliary in the sense that one can learn a set of commands/utilities that will be used/helpful in completing the main part of the lab (i.e., remaining sections). If you familiar with these mechanisms you can skip this part.

a) For interested people: K3s/Flannel networking models (the first one is particularly recommended):

- https://mvallim.github.io/kubernetes-under-the-hood/documentation/kube-flannel.html
- https://www.henrydu.com/2020/11/16/k3s-cni-flannel/
  - about container-shim: https://iximiuz.com/en/posts/implementing-container-runtime-shim/

[1] Traefik is the default ingress controller in K3s. Kubernetes Ingress controllers, in basic form, serve external HTTP requests and direct them to appropriate service instances in the cluster. In doing so they define routing rules to the services inside the cluster. To this end they configure various network settings of the cluster. They thus play a complementary role to load balancers.

  - about CRI, containerd, dockerd, ctr and crictl: https://iximiuz.com/en/posts/containerd-command-line-clients/

Notice that flanneld in k3s is a daemon, not a "bridge" or "ovs bridge". In k3s it is implemented in a single process together with all remaining components of the control plane of Kubernetes on a given node. Therefore it is not shown by neither network level commands as brtcl nor Kubernetes API facing commands as kubectl. It creates flannel.1 VXLAN interface on the host and keeps the routes between the nodes (and their Pods) in the FDB forwarding base of the kernel up-to-date. flanneld agent on each host pre-assigns a subnet to the host and assigns an IP address to each created Pod. Interface flannel.1 then wraps Pod's packets into UDP and VXLAN headers.

b) List bridges on a node: `brctl show`

c) Useful commands. Listing containers in a pod of a given Kubernetes namespace and inspecting pod/container internals:

- list containers in a pod

```
kubectl get pod -n kube-system -o="custom-columns=NAME:.metadata.name,INIT-CONTAINERS:.spec.initContainers[*].name,CONTAINERS:.spec.containers[*].name"
```

- list both running and stopped containers on a given cluster node (you have to ssh):
  - `sudo ctr container list`

  or similar (but not equal) effect

  - `sudo crictl ps -a`

  Note: *ctr* is a command-line client shipped as part of the containerd project, while *crictl* is a command-line client for [Kubernetes] CRI-compatible container runtimes)..

- list *network-namespaces* on a given cluster node (each pod corresponds to a distinct net-namespace). Note: this works with containerd, but may return no results (an empty list) with Docker as the container engine.

```
ip netns list
```

- list the IP addresses of the interfaces in a given network-namespace on a given k3s node (ssh-ed):

```
sudo ip netns exec cni-061c7b42-d791-f458-1abf-15b29536d510 ip addr show
```

  - # alternatively use the command (no address info will be shown) … **`ip link show`**

- For CRI-compatible container managers (as containerd): display help info, list containers, list pods on a given cluster node, run commands in a container to check current the status of the container

```
sudo crictl help
sudo crictl ps
sudo crictl pods
sudo crictl exec #see the example below
```

Note: for system containers (e.g., metric-server or coredns) it may not be possible to run certain commands against. The latter refers in particular to the `exec` command.

```
# run a command in a container; generic: sudo crictl exec <contained-id> ip addr show
ubuntu@kpi091:~$ sudo crictl exec a60432cef5826 ip addr show
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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue state UP
    link/ether e2:ce:32:e2:00:15 brd ff:ff:ff:ff:ff:ff
    inet 10.42.0.34/24 brd 10.42.0.255 scope global eth0
```

```
       valid_lft forever preferred_lft forever
    inet6 fe80::e0ce:32ff:fee2:15/64 scope link
       valid_lft forever preferred_lft forever
# check/inspect current status of a container (below, we grep for the pid of the container)
ubuntu@kpi091:~$ sudo crictl inspect <contained-id> | grep pid
    "pid": 3052,
            "pid": 1
            "type": "pid"
```

d) List *veth* pairs in a cluster node.

Combining *veth* pairs spanning different namespaces/bridges is possible based on `ip link show` in respective network namespaces:

- Check IPs in one net-namespace, e.g. do `ip link show` in the root ns; then you will see something like this for one of the interfaces (notice the suffix @ifX in the interface name):

  ```
  6: vethaecbf8cb@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
  master cni0 state UP mode DEFAULT group default
  ```

- Do the same in some other net-namespace; then for one of the displayed interfaces we can see something like this (below, cni-061c7b42-d791-f458-1abf-15b29536d510 is net-namespace name; it can be found in the *iftree* output shown in the frame above):

  ```
  sudo ip netns exec cni-061c7b42-d791-f458-1abf-15b29536d510 ip link show
  2: eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP
  mode DEFAULT group default link/ether 6a:b8:44:71:ac:d3 brd ff:ff:ff:ff:ff:ff
  link-netnsid 0
  ```

  We can infer a given *veth* pair based on the correspondence of link indices X in the suffixes @ifX as bolded in the example.

2.3. Checking routing settings in cluster nodes

This section is for interested students. We will inspect the routing starting from the settings of pods and containers and then move on to the settings of routing on node and flannel level. Key message is that all Kubernetes network related setting are stored in configuration files.

a) Checking the pods running on a given cluster node (kpi091 in this case) in kubectl and checking pods' interfaces on that node

Checking node pods using kubectl

```
xubuntu@xubulab:~$ kubectl get pods --all-namespaces -o wide --field-selector
spec.nodeName=kpi091
NAMESPACE     NAME                                      READY   STATUS      RESTARTS
AGE   IP          NODE      NOMINATED NODE    READINESS GATES
kube-system   helm-install-traefik-crd-ddqhm            0/1     Completed   0
50d   10.42.0.4   kpi091    <none>            <none>
kube-system   helm-install-traefik-bxjnb                0/1     Completed   1
50d   10.42.0.2   kpi091    <none>            <none>
kube-system   svclb-traefik-d3b37a72-d5pgj              2/2     Running     12 (16h ago)
50d   10.42.0.26  kpi091    <none>            <none>
kube-system   local-path-provisioner-7b7dc8d6f5-zpl8h   1/1     Running     7 (16h ago)
50d   10.42.0.28  kpi091    <none>            <none>
kube-system   metrics-server-668d979685-g6d5s           1/1     Running     6 (16h ago)
50d   10.42.0.27  kpi091    <none>            <none>
```

Note: similar results can be obtained using `sudo crictl pods` *run directly on the cluster node.*

Node: mapping from container interface => container and pod name.

- **(Optional – for interested students)** Here we display pod interfaces (suffix -ethX or -lo for veth and loopback port, respectively)

```
ubuntu@kpi091:~$ sudo ls /var/lib/cni/results -al
total 32
drwx------ 2 root root 4096 Sep 17 12:57 .
drwx------ 5 root root 4096 Jul 28 15:49 ..
-rw------- 1 root root 1543 Sep 17 12:57 cbr0-
08fd8a402c5182a78fbb324d4fc997ec8175a1ecf338187d801429de4fc4ccb1-eth0
-rw------- 1 root root 1679 Sep 17 12:57 cbr0-
271c1cd19eca72d4c4c8884e1eff1a3346d4c4c0c0d9d58ca82d733921ac9e22-eth0
-rw------- 1 root root 1535 Sep 17 12:57 cbr0-
7dd8b441a1527173820a9e53b93e077dfc2ed4d633ded37ac224f7391efbe1ba-eth0
-rw------- 1 root root 1126 Sep 17 12:57 cni-loopback-
08fd8a402c5182a78fbb324d4fc997ec8175a1ecf338187d801429de4fc4ccb1-lo
-rw------- 1 root root 1262 Sep 17 12:57 cni-loopback-
271c1cd19eca72d4c4c8884e1eff1a3346d4c4c0c0d9d58ca82d733921ac9e22-lo
-rw------- 1 root root 1118 Sep 17 12:57 cni-loopback-
7dd8b441a1527173820a9e53b93e077dfc2ed4d633ded37ac224f7391efbe1ba-lo
```

- **(Optional – for interested students)** Here, we retrieve and check network configuration data for one pod to see its final network settings. We use the pod's id (cbr0-prefixed and eth0-suffixed) we retrieved by using the command `sudo ls /var/lib/cni/results -al` shown above.

```
ubuntu@kpi091:~$ sudo cat /var/lib/cni/results/cbr0-
08fd8a402c5182a78fbb324d4fc997ec8175a1ecf338187d801429de4fc4ccb1-eth0 | jq
{
  "kind": "cniCacheV1",
  "containerId": "08fd8a402c5182a78fbb324d4fc997ec8175a1ecf338187d801429de4fc4ccb1",
  "config": REDACTED,
  "ifName": "eth0",
  "networkName": "cbr0",
  "cniArgs": [
    [
      "IgnoreUnknown",
      "1"
    ],
    [
      "K8S_POD_NAMESPACE",
      "kube-system"
    ],
    [
      "K8S_POD_NAME",
      "local-path-provisioner-7b7dc8d6f5-zpl8h"
    ],
    [ # first part of this id is shown as POD ID in “sudo crictl pods” and “sudo crictl ps”
      "K8S_POD_INFRA_CONTAINER_ID",
      "08fd8a402c5182a78fbb324d4fc997ec8175a1ecf338187d801429de4fc4ccb1"
    ],
    [
      "K8S_POD_UID",
      "0125992b-e574-4354-a765-6ce9b87243fd"
    ]
  ],
  "capabilityArgs": {
    "dns": {
      "Servers": [
        "10.43.0.10"
      ],
      "Searches": [
        "kube-system.svc.cluster.local",
        "svc.cluster.local",
        "cluster.local",
        ""
      ],
      "Options": [
        "ndots:5"
      ]
    },
    "io.kubernetes.cri.pod-annotations": {
      "kubernetes.io/config.seen": "2022-09-17T12:57:49.040662889Z",
      "kubernetes.io/config.source": "api"
    }
  },
  "result": {
    "cniVersion": "1.0.0",
    "dns": {},
    "interfaces": [
      { # this is the cni0 bridge itself (check by running ip link show master cni)
        "mac": "c6:76:6d:5c:2c:43",
        "name": "cni0"
      },
      { # this is veth interface in cni0 bridge (check by running ip link show master cni)
        "mac": "da:42:40:a4:2f:77",
        "name": "vetha7541b74"
      },
      { # this is veth interface in the pod namespace
        "mac": "36:e6:d4:9d:18:57",
        "name": "eth0",
        "sandbox": "/var/run/netns/cni-1984eda9-38ae-66cb-c817-34f8ec6a3ea8"
      }
    ],
    "ips": [
      { # this is pod's cluster-level IP address (all its containers will share this address)
        "address": "10.42.0.28/24",
        "gateway": "10.42.0.1",
        "interface": 2
      }
    ],
    "routes": [
      {
        "dst": "10.42.0.0/16"
      },
      {
        "dst": "0.0.0.0/0",
        "gw": "10.42.0.1"
      }
    ]
  }
}
ubuntu@kpi091:~$
```

- **(Optional – for interested students)** Also, cni0 bridge configuration for its particular veth interfaces (per pod network-namespace) is stored in folder `/var/lib/cni/flannel/` and this information is used to create cni0. In particular, it instructs that cni0 should be of type *bridge* and that cni0 should use the *host-local* IPAM for assigning IP addresses to pods (the range of IP addresses for use by IPAM is given by the attribute *subnet*).

```
# list cni0 information for veth interfaces in cni0
ubuntu@kpi091:~$ sudo ls /var/lib/cni/flannel
08fd8a402c5182a78fbb324d4fc997ec8175a1ecf338187d801429de4fc4ccb1
271c1cd19eca72d4c4c8884e1eff1a3346d4c4c0c0d9d58ca82d733921ac9e22
7dd8b441a1527173820a9e53b93e077dfc2ed4d633ded37ac224f7391efbe1ba

# list cni0 information for a selected veth
kpi091:~$ sudo cat
/var/lib/cni/flannel/08fd8a402c5182a78fbb324d4fc997ec8175a1ecf338187d801429de4fc4ccb1 | jq
{
  "cniVersion": "1.0.0",
  "forceAddress": true,
  "hairpinMode": true,
  "ipMasq": false,
  "ipam": {
    "ranges": [
      [
        {
          "subnet": "10.42.0.0/24"
        }
      ]
    ],
    "routes": [
      {
        "dst": "10.42.0.0/16"
      }
    ],
    "type": "host-local"
  },
  "isDefaultGateway": true,
  "isGateway": true,
  "mtu": 1450,
  "name": "cbr0",
  "type": "bridge"
}
ubuntu@kpi091:~$
```

b) Check routing settings in a given pod. Notice default gateway is set to `10.42.0.1` which is consistent with what we can find in the table above.

```
 xubuntu@xubulab:~$ kubectl exec -it -n kube-system local-path-provisioner-7b7dc8d6f5-zpl8h --
/bin/sh
/ # ip addr show
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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue state UP
    link/ether 46:8d:d4:ef:84:7c brd ff:ff:ff:ff:ff:ff
    inet 10.42.0.48/24 brd 10.42.0.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::448d:d4ff:feef:847c/64 scope link
       valid_lft forever preferred_lft forever

/ # ip route sh
default via 10.42.0.1 dev eth0
10.42.0.0/24 dev eth0 scope link  src 10.42.0.48
10.42.0.0/16 via 10.42.0.1 dev eth0
/ #
```

c) Check flannel settings – run the following commands in a selected cluster node (here, we used our master). Notice that we can enable/disable flannel SNAT capability (see the second part of the screen capture below).

Note: it is recommended to do this exercise checking Figure 2 (MAC addresses in Figure 2 may differ, but the kernel structures involved as fdb, etc., are valid).

```
# show vxlan devices
ubuntu@kpi091:~$ ip link show type vxlan
4: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT
group default
    link/ether 5e:14:d0:84:69:81 brd ff:ff:ff:ff:ff:ff

# show flannel.1 VXLAN details;
# notice configuring flannel.1 VXLAN device to use eth0 and its IP for traffic sending/receiving
ubuntu@kpi091:~$ ip -d a show flannel.1
```

```
4: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group default
    link/ether 5e:14:d0:84:69:81 brd ff:ff:ff:ff:ff:ff promiscuity 0 minmtu 68 maxmtu 65535
    vxlan id 1   local 192.168.1.38   dev eth0 srcport 0 0 dstport 8472 nolearning ttl auto ageing
300 udpcsum noudp6zerocsumtx noudp6zerocsumrx numtxqueues 1 numrxqueues 1 gso_max_size 65536
gso_max_segs 65535
    inet 10.42.0.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::bcb1:aaff:fe6b:7485/64 scope link
       valid_lft forever preferred_lft forever

# check the ARP table on kpi091 created by flannel (mapping IP/MAC of flannel interfaces on
remaining nodes); this, for each remote node, maps its flannel.1 IP address onto the MAC address
# PERMANENT flag denotes that the entry is permanent and flannel.1 device needs not broadcast ARP
queries “who has”
ubuntu@kpi091:~$ ip neigh show dev flannel.1
10.42.2.0 lladdr 2e:cb:94:23:86:79 PERMANENT   <= here, MAC address becomes the VXLAN inner MAC addresses
10.42.1.0 lladdr 36:84:0d:94:d6:34 PERMANENT                     -“-
10.42.3.0 lladdr 9a:29:f6:23:6f:d9 PERMANENT                     -“-

# check the forwarding data base for flannel.1 vxlan device; mind we are on node 192.168.1.38
# this, for each remote node, maps node’s flannel.1 MAC address onto node’s eth0 IP address
ubuntu@kpi091:~$ bridge fdb show dev flannel.1
36:84:0d:94:d6:34 dst 192.168.1.41 self permanent
2e:cb:94:23:86:79 dst 192.168.1.39 self permanent  <= VXLAN outer IP address
9a:29:f6:23:6f:d9 dst 192.168.1.40 self permanent

# Nitty-gritty detail: by default, flannel.1 does SNAT for IP packets from the outside of the flannel network
(so, not inter-pod communication) to cni0 (10.42.0.0/16). Although this is normal, it makes the analysis of
external traffic inside the cluster harder (insightful reader can check it). If you plan to analyse external
traffic (this is BEYOND the scope of the lab) you can disable flannel.1 SNAT: on each node where you want to
sniff external traffic, edit file 10-flannel.conflist and add ‘"ipMasq":false’ after the line
‘"isDefaultGateway":true,’ and reboot.

# Check iptables on the node for flannel (NOTE: in older releases of K3s iptables-save produces slightly
different output where you can see FORWARD and POSTROUTING instead of FLANNEL-FWD and FLANNEL-POSTRTG,
respectively). Below, SNAT rules are enabled and commented with "flanneld masq".
ubuntu@kpi091:~$ sudo iptables-save | grep flanneld
[sudo] password for ubuntu:
-A FORWARD -m comment --comment "flanneld forward" -j FLANNEL-FWD
-A FLANNEL-FWD -s 10.42.0.0/16 -m comment --comment "flanneld forward" -j ACCEPT
-A FLANNEL-FWD -d 10.42.0.0/16 -m comment --comment "flanneld forward" -j ACCEPT
-A POSTROUTING -m comment --comment "flanneld masq" -j FLANNEL-POSTRTG
-A FLANNEL-POSTRTG -m mark --mark 0x4000/0x4000 -m comment --comment "flanneld masq" -j RETURN
-A FLANNEL-POSTRTG -s 10.42.0.0/24 -d 10.42.0.0/16 -m comment --comment "flanneld masq" -j RETURN
-A FLANNEL-POSTRTG -s 10.42.0.0/16 -d 10.42.0.0/24 -m comment --comment "flanneld masq" -j RETURN
-A FLANNEL-POSTRTG ! -s 10.42.0.0/16 -d 10.42.0.0/24 -m comment --comment "flanneld masq" -j RETURN
-A FLANNEL-POSTRTG -s 10.42.0.0/16 ! -d 224.0.0.0/4 -m comment --comment "flanneld masq" -j
MASQUERADE --random-fully
-A FLANNEL-POSTRTG ! -s 10.42.0.0/16 -d 10.42.0.0/16 -m comment --comment "flanneld masq" -j
MASQUERADE --random-fully
# Check current version of 10-flannel.conflist and optionally disable SNAT
ubuntu@kpi091:~$ sudo cat /var/lib/rancher/k3s/agent/etc/cni/net.d/10-flannel.conflist
{
  "name":"cbr0",
  "cniVersion":"1.0.0",
  "plugins":[
    {
      "type":"flannel",
      "delegate":{
        "hairpinMode":true,
        "forceAddress":true,
        "isDefaultGateway":true
        “ . . . “     # <== here insert "ipMasq":false if you want to disable flannel SNAT
      }
    },
    {
      "type":"portmap",
      "capabilities":{
        "portMappings":true
      }
    }
  ]
}
ubuntu@kpi091:~$ reboot
```

Observations:

- It may happen that some of the ehe entries in *flannel.1* vxlan device forwarding data base (the table shown above) contain also entries that can not be associate with any existing object. Do not bother with them.
- Flannel provides VXLAN device (interface) named *flannel.1* and this uses eth0 interface of the cluster node to send/receive VXLAN encapsulated traffic to other cluster nodes.
- Flannel daemon flanneld has also created PERMANENT (non expiring) entries in the ARP table of the node. For example, the entry `10.42.2.0 lladdr 2e:cb:94:23:86:79` denotes IP/MAC addresses mapping for *flannel.1* interface in a remote cluster node (in this case this is *flannel.1* sitting on node kpi092). In other words, local ARP table informs that *flannel.1* interface with IP `10.42.2.0` is reachable within the flannel VXLAN on the **MAC address `2e:cb:94:23:86:79`**. So, flannel.1 device will use this address as the MAC address in the **inner Ethernet fram**e of the VXLAN. Also, it will use it for mapping onto the IP of the remote cluster node containing the VTEP to which VXLAN tunnel extends – see the following bullet point.
- Based on the above, flanneld populates the forwarding data base of the node for flannel.1 VXLAN device. For example, entry `2e:cb:94:23:86:79 dst 192.168.1.39` tells flannel.1 in node kpi091 that flannel.1 VXLAN port with (inner) MAC address `2e:cb:94:23:86:79` is available in the network on IP address `192.168.1.39`. Here, address `192.168.1.39` will be the outer VXLAN IP address.
- Notice that the two tables described above together provide a two-step mapping from the IP address of the destination pod subnetwork to the IP address of the cluster node where that pod resides.
- Check `ip n` or `ip r`
- Check for ARP: `ip neigh show`

3. Checking inter-container connectivity

In this section we analyse the routing of packets between containers for three following scenarios: when the containers belong to same Pod, belong to different Pods running in same node, and are located in different nodes. Follow the instructions given below. A big picture of the series of experiments is show in **Figure 1**. More detailed schemes corresponding to particular experiments are provided in the companion slide deck from the lecture.

As shown in the figure, our goal in this section is to check inter—container connectivity in various configurations, i.e., for various locations of the containers with respect to each other (inside same pod, same node but different pods, different nodes). In each experiment, we will look into details of the communication and analyse the role of networking devices of Linux involved. Even though the experiments are conducted using a very basic CNI (flannel), nonetheless, they will give us sufficient insight into the principles of operation and the role of CNI (Container Networking Interface) in Kubernetes.

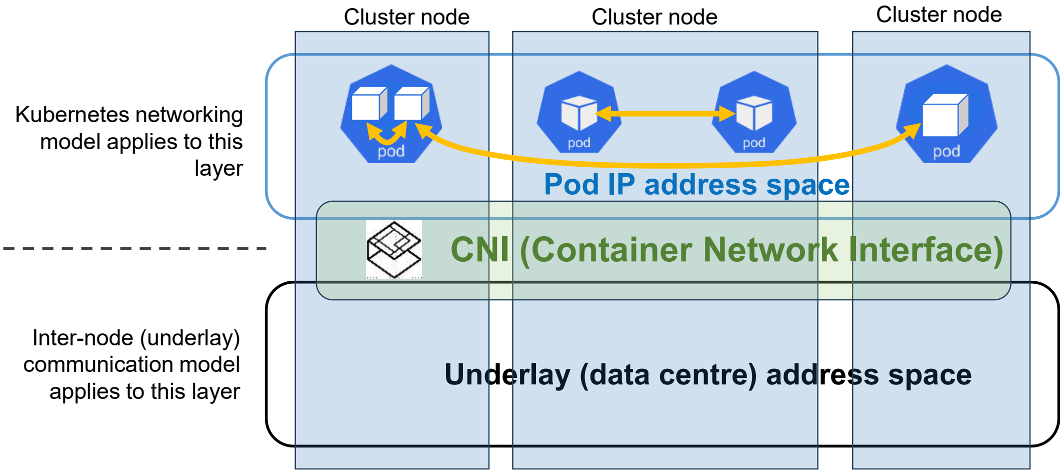

Cluster node
Cluster node
Cluster node
Kubernetes networking model applies to this layer
pod
pod
pod
pod
Pod IP address space
CNI (Container Network Interface)
Inter-node (underlay) communication model applies to this layer
Underlay (data centre) address space

Figure 1 L2/L3 connectivity experiments – top level view.

3.1. Between containers inside the same Pod

For this scenario, we create a pod with two containers. The nginx container serves a default web page, and the busybox container sleeps indefinitely ready to ssh to it an run shell commands, e.g., curl/wget to nginx. These two container images are often used in Kubernetes courses for learning Kubernetes networking. Some of the steps in this section are the repetition of the steps form section 2.3. However, while the description in sec. 2.3 was general, currently we analyze the network settings in the context of a particular instance of communication between pods.

a) Create a manifest file of a pod, say nginx-busybox.yaml, with the following contents and run it:

```
# manifest file contents
apiVersion: v1
kind: Pod
metadata:
  name: nginx-busybox
spec:
  containers:
  - command:
    - sleep
    - infinity
    image: busybox
    name: busybox
  - image: nginx
    name: nginx

xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl apply -f nginx-busybox.yaml
pod/nginx-busybox created
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl get pods -o wide
NAME            READY   STATUS    RESTARTS       AGE     IP           NODE     NOMINATED NODE   READINESS GATES
dnsutils        1/1     Running   41 (38m ago)   8d      10.42.3.37   kpi093   <none>           <none>
nginx-busybox   2/2     Running   0              3m23s   10.42.1.30   kpi094   <none>           <none>
```

b) Launch a shell for the busybox container inside the pod nginx-busybox and check if busybox container can communicate with nginx over pod's *localhost* (wget should be able to retrieve a document in HTML format).

```
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl exec -it -c busybox nginx-busybox -- /bin/sh

# check the connectivity over local host (both containers use port 80 for HTTP)
/ # wget localhost -O - 2>/dev/null
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

c) **(optional/auxiliary – for very interested students)** You can run the following extra checks to see both containers run in the same network namespace but in different process namespaces. They can communicate (same network namespace), but do not see each other's process identifier (PID). The output shows busybox can see that “something” is listening on localhost:80 (0.0.0.0:80), but the PID of the listener (nginx in this case) is not visible to busybox. This part can be done referring to the pod structure in slide 26 (*Containers inside the same Pod*) from the lecture slide deck.

On the other hand, if you do similar check after logging (kubectl exec) into nginx then you will notice nginx can see its own PID (equal 1). The latter is visible in the second part of the capture in the table below.

```
# still on the busybox container
/# netstat -tlpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      -
tcp        0      0 :::80                   :::*                    LISTEN      -

# do the same for nginx (exit, exec to nginx and install net-tools first)
/# exit
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl exec -it -c nginx nginx-busybox -- /bin/bash
root@nginx-busybox:/# apt update
root@nginx-busybox:/# apt install net-tools

root@nginx-busybox:/# netstat -tlpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      1/nginx: master
pro
tcp6       0      0 :::80                   :::*                    LISTEN      1/nginx: master
pro
```

d) **(cntd On namespace isolation in pods: optional – as above)** Now we can take a look at containers from the pod/network namespace perspective. Exit the container and then follow the scenario from the table below (part 1 in the table is optional and can be skipped).

```
# 1. optional part
# get pods (we are in default namespace of k3s)
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl get pod -o json nginx-busybox | jq

# find the following part of the description of the pod where (long) container IDs are given
...
    "containerStatuses": [
      {
        "containerID":
"containerd://24a0a0e95f8dd20bd60aa8bbe075e9ad0518217976f0243f1c16243dcec35458",
        "image": "docker.io/library/busybox:latest",
        "imageID":
"docker.io/library/busybox@sha256:ad9bd57a3a57cc95515c537b89aaa69d83a6df54c4050fcf2b41ad367bec0c
d5",
        "lastState": {},
        "name": "busybox",
        "ready": true,
        "restartCount": 0,
        "started": true,
        "state": {
          "running": {
            "startedAt": "2022-09-22T20:11:01Z"
          }
        }
      },
      {
        "containerID":
"containerd://67336c7aa0c399d106947e846651019b9adec5e980f14242761836ff0d1b9a4c",
        "image": "docker.io/library/nginx:latest",
        "imageID":
"docker.io/library/nginx@sha256:0b970013351304af46f322da1263516b188318682b2ab1091862497591189ff1
",
        "lastState": {},
        "name": "nginx",
        "ready": true,
        "restartCount": 0,
        "started": true,
        "state": {
          "running": {
            "startedAt": "2022-09-22T20:11:24Z"
          }
        }
...

# 2. sufficient part
# identify the node where our nginx-busybox pod runs
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl get pods -o wide
NAME             READY   STATUS    RESTARTS       AGE   IP            NODE     NOMINATED NODE
READINESS GATES
nginx-busybox    2/2     Running   0              62m   10.42.1.30    kpi094   <none>
<none>
dnsutils         1/1     Running   42 (37m ago)   8d    10.42.3.37    kpi093   <none>
<none>

# ssh to the node where the pod runs and find container process IDs one by one
xubuntu@xubulab:~$ ssh kpi094
ubuntu@kpi094:~$ sudo crictl ps
CONTAINER           IMAGE               CREATED             STATE               NAME
ATTEMPT             POD ID              POD
67336c7aa0c39       0c404972e1305       About an hour ago   Running             nginx
0                   d5635f85512eb       nginx-busybox
24a0a0e95f8dd       410fde8b14eed       About an hour ago   Running             busybox
0                   d5635f85512eb       nginx-busybox
90c00c4c3d34d       b12bbec1f4615       5 hours ago         Running             lb-tcp-443
13                  fa440c15de16f       svclb-traefik-d3b37a72-mxqk4
44c1dca6f0402       2ef507d0470ec       5 hours ago         Running             traefik
13                  de5ac56b7f02a       traefik-7cd4fcff68-tg94h
f636764728bbf       b12bbec1f4615       5 hours ago         Running             lb-tcp-80
13                  fa440c15de16f       svclb-traefik-d3b37a72-mxqk4

# get the PIDs of the containers in the root
ubuntu@kpi094:~$ sudo crictl inspect 67336c7aa0c39 | jq '.["info"].pid'
6677
ubuntu@kpi094:~$ sudo crictl inspect 24a0a0e95f8dd | jq '.["info"].pid'
6587

# obtain namespaces for each container process; it can be seen that both containers/processes \
# share network/ipc/uts namespaces (column NPROSC) while other namespaces as pid are different
ubuntu@kpi094:~$ sudo lsns -p 6677
        NS TYPE   NPROCS   PID USER  COMMAND
4026531834 time      164     1 root  /sbin/init fixrtc splash
4026531837 user      164     1 root  /sbin/init fixrtc splash
4026532368 net         7  6549 65535 /pause
4026532602 uts         7  6549 65535 /pause
4026532603 ipc         7  6549 65535 /pause
4026532608 mnt         5  6677 root  nginx: master process nginx -g daemon off;
4026532609 pid         5  6677 root  nginx: master process nginx -g daemon off;
4026532610 cgroup      5  6677 root  nginx: master process nginx -g daemon off;

ubuntu@kpi094:~$ sudo lsns -p 6587
        NS TYPE   NPROCS   PID USER  COMMAND
4026531834 time      163     1 root  /sbin/init fixrtc splash
4026531837 user      163     1 root  /sbin/init fixrtc splash
4026532368 net         7  6549 65535 /pause
4026532602 uts         7  6549 65535 /pause
4026532603 ipc         7  6549 65535 /pause
4026532605 mnt         1  6587 root  sleep infinity
4026532606 pid         1  6587 root  sleep infinity
4026532607 cgroup      1  6587 root  sleep infinity
```

the same

different

e) Conclusion

A single pod may seem like a very simple construct, but there is more going on under the hood of the Linux network stack that enables pods to function. Additionally, the ability for containers within a pod to communicate over the localhost address facilitates the use of **patterns as *sidecar* and *init* containers**.

3.2. Between containers in different Pods inside the same node

This case is left to the students for investigation as a **bonus task (2 points)**. Please, elaborate on the communication to show relevant details by adopting selected mechanisms used in points 3.1 and 3.3. **If you decide to complete this task, please deliver a separate 1-page report as a proof-of-attempt**.

3.3. Between containers in different nodes

This is the most complex case of flannel operation. We will use two pods again, this time instantiated in different nodes, to check their routing settings and inspect protocol encapsulation during the communication of those pods.

a) Setting the environment

To set the environment we will create two pods, nginx and busybox[2], this time placed in different nodes. To control pod placement we will use Kubernetes *nodeSelector* attribute that allows to specify the allowed set of cluster nodes where a pod can be instantiated (refer to Kubernetes documentation for a detailed description of *nodeSelector*). Follow the instruction from the box given below.

```
# Create manifest file with the following contents (adjust node names to your environment)

xubuntu@xubulab:~/cluster-pi/manifests/lab$ cat nginx-busybox-diffnodes.yaml
# Two pods will be created in predefined (different) cluster nodes

apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeSelector:
    kubernetes.io/hostname: kpi092
  containers:
  - image: nginx
    name: nginx

---
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  nodeSelector:
    kubernetes.io/hostname: kpi093
  containers:
  - command:
    - sleep
    - infinity
    image: busybox
    name: busybox

# Follow the commands to get the pods running and verify they have been created as expected

xubuntu@xubulab:~/cluster-pi/manifests/lab$ ls
nginx-busybox-diffnodes.yaml  nginx-busybox.yaml
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl get pods
NAME             READY   STATUS    RESTARTS       AGE
```

[2] It is recommended to set a unique name to each pod to make it distinguishable from other pods during experiments.

```
dnsutils         1/1     Running   49 (37h ago)    17d
nginx-busybox    2/2     Running   4 (37h ago)     8d
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl apply -f nginx-busybox-diffnodes.yaml
pod/nginx created
pod/busybox created
xubuntu@xubulab:~/cluster-pi/manifests/lab$ kubectl get pods -o wide
NAME            READY   STATUS    RESTARTS       AGE   IP            NODE     NOMINATED NODE
READINESS GATES
dnsutils        1/1     Running   49 (37h ago)   17d   10.42.3.43    kpi093   <none>           <none>
nginx-busybox   2/2     Running   4 (37h ago)    8d    10.42.1.36    kpi094   <none>           <none>
busybox         1/1     Running   0              64s   10.42.3.44    kpi093   <none>           <none>
nginx           1/1     Running   0              64s   10.42.2.19    kpi092   <none>           <none>
```

b) Layer 2 settings

Later on in this exercise we will log to busybox and wget a file from nginx pod. First, let’s check layer 2 configuration relevant to the communication between those pods.

Settings for nginx pod (notice the use of colours to improve the readability of the screen capture)

```
# Enter the nginx pod
xubuntu@xubulab:~$ kubectl exec -it nginx -- /bin/bash

# Install missing packages
root@nginx:/# apt update
Get:1 http://deb.debian.org/debian bullseye InRelease [116 kB]
...
root@nginx:/# apt install -y iproute2
Reading package lists...
...
root@nginx:/# apt install ethtool
Reading package lists...
...

# Check the network configurations on nginx pod
root@nginx:/# ip addr show
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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 4a:ec:0e:72:6e:45 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.42.2.21/24 brd 10.42.2.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::48ec:eff:fe72:6e45/64 scope link
       valid_lft forever preferred_lft forever

# eth0@if7 means that the remote end of the veth pair eth0 belongst to (on the host) is 7.
# it can be confirmed as follows with ethtool or from the root network namespace on the host
root@nginx:/# ethtool -S eth0
NIC statistics:
     peer_ifindex: 7
...
# confirming by checking network configuration on kpi092 (host for our nginx pod) (we skip
# remaining interfaces for brevity) - @if2 and peer_ifindex: 2 relate to eth0 in our nginx pod
root@nginx:/# exit
exit
xubuntu@xubulab:~$ ssh kpi092
ubuntu@kpi092:~$ ip link show
...
7: veth3a7939d6@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master cni0 state UP
mode DEFAULT group default
    link/ether 72:12:d0:d0:b5:60 brd ff:ff:ff:ff:ff:ff link-netns cni-034fea82-9ee2-5bf7-eaba-
3aaeebdfcbf7
ubuntu@kpi092:~$ ethtool -S veth3a7939d6
NIC statistics:
     peer_ifindex: 2
...
```

```
# The eth0-veth3a7939d6 virtual Ethernet pair connects to a bridge on the host. Once you
# accomplish the exercise in section 2.3 you will see this setting allows pods on the same host
# to communicate directly with each other over the bridge. The cni0 bridge interface has an IP
# address assigned to it, which will be important in the Layer 3 routing process:
ubuntu@kpi092:~$ brctl show
bridge name       bridge id                  STP enabled       interfaces
cni0              8000.868c45b22963 no                 veth1ad1bd98
                                                                veth3a7939d6
ubuntu@kpi092:~$ ip -br addr show cni0
cni0             UP             10.42.2.1/24 fe80::848c:45ff:feb2:2963/64
```

Settings for busybox pod

These settings can be checked in an analogous way as above, except that entering the `busybox` pod has to be done using shell, .i.e., `kubectl exec -it busybox -- /bin/sh` .

c) Layer 3 settings

L3 packet has to follow a path going from the source pod via cni0, then flannel.1 and eth0 on the source node, all the way down across the cluster network towards the eth0, flannel.1, cni0 and the target pod on the destination node. In the following we will check the routing settings of respective network elements in the direction form `busybox` to `nginx`.

Analyse your cluster as below by adjusting the addresses to your case: First, our `busybox` pod must decide how to send its traffic for the remote network (mind `busybox` is in `10.42.3./24` network while `nginx` address `10.42.2.21` is derived from `10.42.2.0/24` network) . The routing table in `busybox` pod includes a route matching `10.42.2.21` that has the mask `10.42.0.0/16`, so `busybox` will send the traffic to `cni0` bridge through the `eth0` interface of the pod (`via 10.42.3.1 dev eth0`). Notice the default gateway for `busybox` is at `10.42.3.1`, being equal to the IP address of the `cni0` bridge on the host. Notice that for the execution of a single command in a pod (*ad-hoc* - without entering the pod's shell interactively) it is sufficient to run something like `kubectl exec <pod-name> <shell-command>` .

```
# routing table on the busybox pod with entering pod's shell
xubuntu@xubulab:~$ kubectl exec -it busybox -- /bin/sh
/ # ip route show
default via 10.42.3.1 dev eth0
10.42.0.0/16 via 10.42.3.1 dev eth0
10.42.3.0/24 dev eth0 scope link src 10.42.3.47
```

Once the traffic has reached `cni0` bridge, the next routing decision will determine how to forward that traffic to the desired network (`10.42.2.0/24`). The host's routing table shows that traffic destined to the `10.42.2.0/24` network will be sent via `10.42.2.0` on the `flannel.1` interface:

```
# routing table on node kpi093 (busybox host)
ubuntu@kpi093:~$ ip route sh
default via 192.168.1.1 dev eth0 proto dhcp src 192.168.1.40 metric 100
10.42.0.0/24 via 10.42.0.0 dev flannel.1 onlink
10.42.1.0/24 via 10.42.1.0 dev flannel.1 onlink
10.42.2.0/24 via 10.42.2.0 dev flannel.1 onlink
10.42.3.0/24 dev cni0 proto kernel scope link src 10.42.3.1
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.40 metric 100
192.168.1.1 dev eth0 proto dhcp scope link src 192.168.1.40 metric 100
```

Now, `flannel.1` interface is responsible for providing VXLAN service between `cni0` and peer cni0 bridges in remaining cluster nodes. `flannel.1` interface description contains the basic parameters needed to create VXLAN tunnels to the remaining cluster nodes, i.e., `VXLAN Id = 1`, source (`local`) IP address for the outer IP packets with the tunnelled traffic, and the destination port (`dstport`) of UDP packets that encapsulate VXLAN frames of the tunnels (same VXLAN id and UDP `dstport` number for all tunnels in the cluster):

```
# flannel.1 interface on node kpi093
ubuntu@kpi093:~$ ip -d a show flannel.1
4: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN group
default
    link/ether 66:16:cf:19:e8:2c brd ff:ff:ff:ff:ff:ff promiscuity 0 minmtu 68 maxmtu 65535
    vxlan id 1  local 192.168.1.40 dev eth0 srcport 0 0  dstport 8472 nolearning ttl auto ageing
300 udpcsum noudp6zerocsumtx noudp6zerocsumrx numtxqueues 1 numrxqueues 1 gso_max_size 65536
gso_max_segs 65535
    inet 10.42.3.0/32 scope global flannel.1
       valid_lft forever preferred_lft forever
    inet6 fe80::6416:cfff:fe19:e82c/64 scope link
       valid_lft forever preferred_lft forever
```

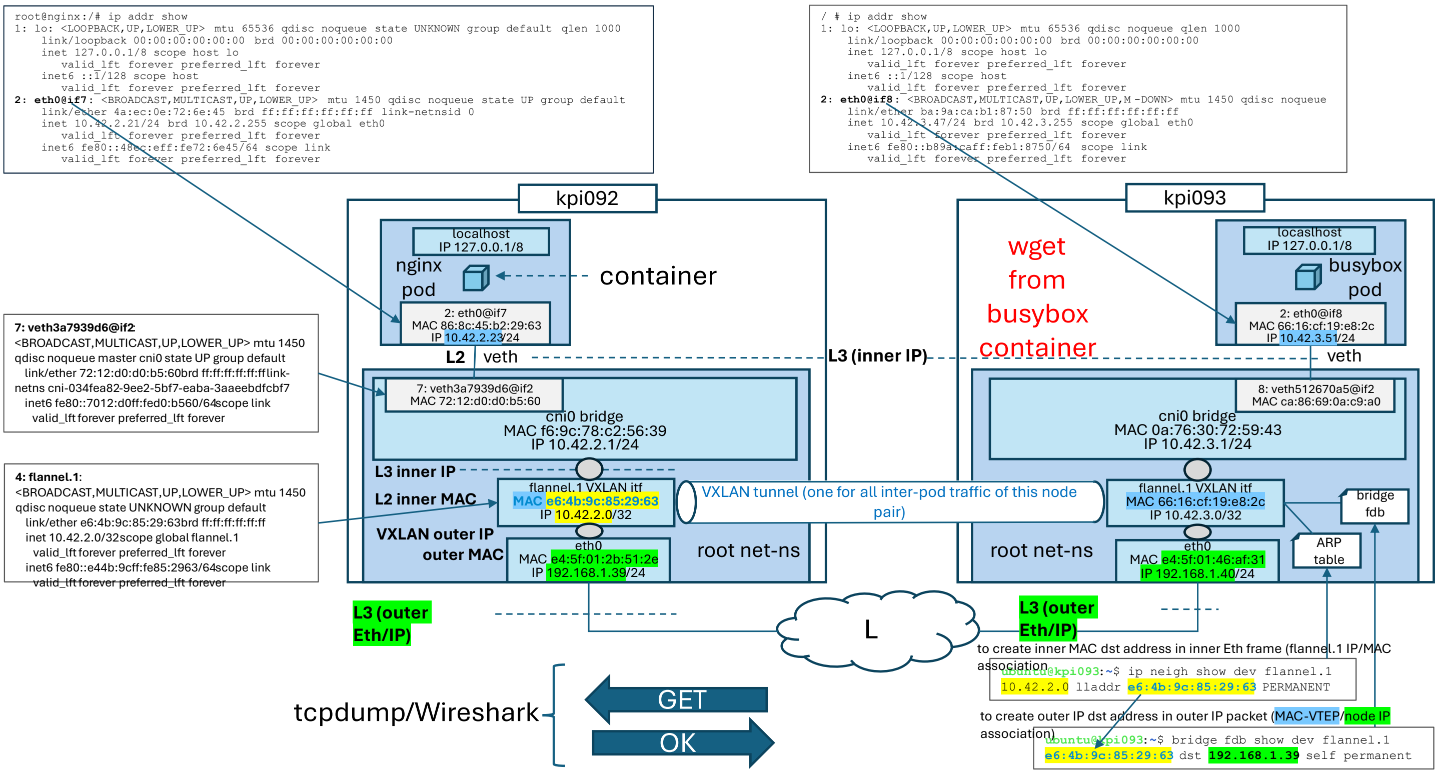
We know from section 2.3 that `flannel.1` uses the ARP table of the host and the internal flannel.1 forwarding table to map a remote pod's IP subnetwork address (and the MAC address of remote flannel1 device) the frame received from `cni0` (directed to a pod in another node) onto the destination IP address of corresponding physical host (outer IP address in VXLAN terminilogy).

We also know from the routing table of (host) node kpi093 (see above) that the next hop for the subnet of a packet from `busybox` to `nginx` is `10.42.2.0` (that is, flannel.1 device in another cluster node). However, no directly reachable interface on the cluster network node contains the `10.244.2.0` address (see the box below). Instead, 10.42.2.0 is pointed as the default gateway, reachable via flannel.1 device. And a static Address Resolution Protocol (ARP) entry exists on the ARP table on the host kpi093 that indicates that the MAC address for `10.42.2.0` is `e6:4b:9c:85:29:63`. Then the flannel.1 forwarding table (bridge forwarding database) directs traffic for this MAC address to a remote destination of `192.168.1.39`. This remote IP address, which is the physical interface eth0 on the `kpi092` node (hosting our `nginx` pod), indicates where the other side of the VXLAN tunnel in the physical (cluster) network resides (actually, the VXLAN tunnel starts and terminates in flannel.1 interfaces). One can check this as shown in the box presented below; notice the correspondence between the addresses in the bolded lines. Notice also the entries are permanent – they are set once by flanneld deamon and used throughout the life of the cluster.

```
# Check kpi093 ARP table for flannel information (mapping form the IP address to the MAC address
# of flannel.1 VXLAN interfaces in the remote cluster nodes)
ubuntu@kpi093:~$ ip neigh show dev flannel.1
10.42.2.0 lladdr e6:4b:9c:85:29:63 PERMANENT   # <= mapping for the remote flannel.1 from its IP to MAC
10.42.0.0 lladdr 46:32:55:ca:55:1f PERMANENT  # the MAC here serves as VXLAN inner MAC address
10.42.1.0 lladdr 32:6d:8a:c8:c2:af PERMANENT

# Check flannel.1 forwarding table (maps form the MAC address of the remote flannel.1 VXLAN
# interface to the IP address of the remote cluster node)
ubuntu@kpi093:~$ bridge fdb show dev flannel.1
1a:b8:85:b9:77:bf dst 192.168.1.41 self permanent   # the IP here serves as VXLAN outer IP address
address
e6:4b:9c:85:29:63 dst 192.168.1.39 self permanent   # <= mapping from the remote flannel.1 MAC to the
                                                    # remote node IP
32:6d:8a:c8:c2:af dst 192.168.1.41 self permanent
46:32:55:ca:55:1f dst 192.168.1.38 self permanent
```

To see a complete picture of address mappings relevant to our example of communication between the pods one should run similar checks also for the `nginx` side (node kpi092 in our case; we omit this for sake of brevity). After consolidating all results, a complete network configuration diagram can be obtained as depicted in

Worth of noting is the fact that flannel VXLAN extends between the pairs of flannel.1 VXLAN devices with inner Eth frames also terminating in (being stripped by) flannel.1. coming from/passed to `cni0` bridges constituting the *inner* Eth frames which, together with the VXLAN header, serve as the (VXLAN) payload of outer IP packets. In particular, shown in the figure are local IP address and MAC mappings needed by flannel.1 to figure out the right remote host IP address (and get from ARP remote host MAC address).



```
root@nginx:/# ip addr show
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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 4a:ec:0e:72:6e:45 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.42.2.21/24 brd 10.42.2.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::48ec:eff:fe72:6e45/64 scope link
       valid_lft forever preferred_lft forever
```

```
/ # ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue 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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M -DOWN> mtu 1450 qdisc noqueue
    link/ether ba:9a:ca:b1:87:50 brd ff:ff:ff:ff:ff:ff
    inet 10.42.3.47/24 brd 10.42.3.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::b89a:caff:feb1:8750/64 scope link
       valid_lft forever preferred_lft forever
```

```
7: veth3a7939d6@if2:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450
qdisc noqueue master cni0 state UP group default
  link/ether 72:12:d0:d0:b5:60brd ff:ff:ff:ff:ff:fflink-
netns cni-034fea82-9ee2-5bf7-eaba-3aaeebdfcbf7
  inet6 fe80::7012:d0ff:fed0:b560/64scope link
    valid_lft forever preferred_lft forever
```

```
4: flannel.1:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450
qdisc noqueue state UNKNOWN group default
  link/ether e6:4b:9c:85:29:63brd ff:ff:ff:ff:ff:ff
  inet 10.42.2.0/32scope global flannel.1
    valid_lft forever preferred_lft forever
  inet6 fe80::e44b:9cff:fe85:2963/64scope link
    valid_lft forever preferred_lft forever
```

to create inner MAC dst address in inner Eth frame (flannel.1 IP/MAC association

```
ubuntu@kpi093:~$ ip neigh show dev flannel.1
10.42.2.0 lladdr e6:4b:9c:85:29:63 PERMANENT
```

to create outer IP dst address in outer IP packet (MAC-VTEP/node IP association)

```
ubuntu@kpi093:~$ bridge fdb show dev flannel.1
e6:4b:9c:85:29:63 dst 192.168.1.39 self permanent
```

Figure 2 Decoding the communication over flannel VXLAN.

d) Communication on the wire

We will use packet capture with tcpdump and Wireshark[3] to observe the communication on the wire. For the setup procedure and usage of both tools we recommend the link https://www.comparitech.com/net-admin/tcpdump-capture-wireshark/. One can find many useful tips for tcpdump therein. **Notice that a more complete set of slides documenting the scenario is included in the lecture slide dec for Lab3.**

We create three terminals: one terminal used to log to the cluster node where the `busybox` runs (kpi093 in our case) and capture the packets from the physical network interface eth0 of the node; the second terminal is used to log into the `busybox` container and send `wget` to `nginx`; third terminal is used to download tcpdump.pcap file from node kpi093 for subsequent analysis in Wireshark. The commands entered in each terminal are gathered in the boxes below, each box corresponding to one terminal. Use common sense to execute the commands in appropriate order. Notice that *xubulab* is my management host I use to *ssh* to cluster nodes and *kubectl* to Kube API.

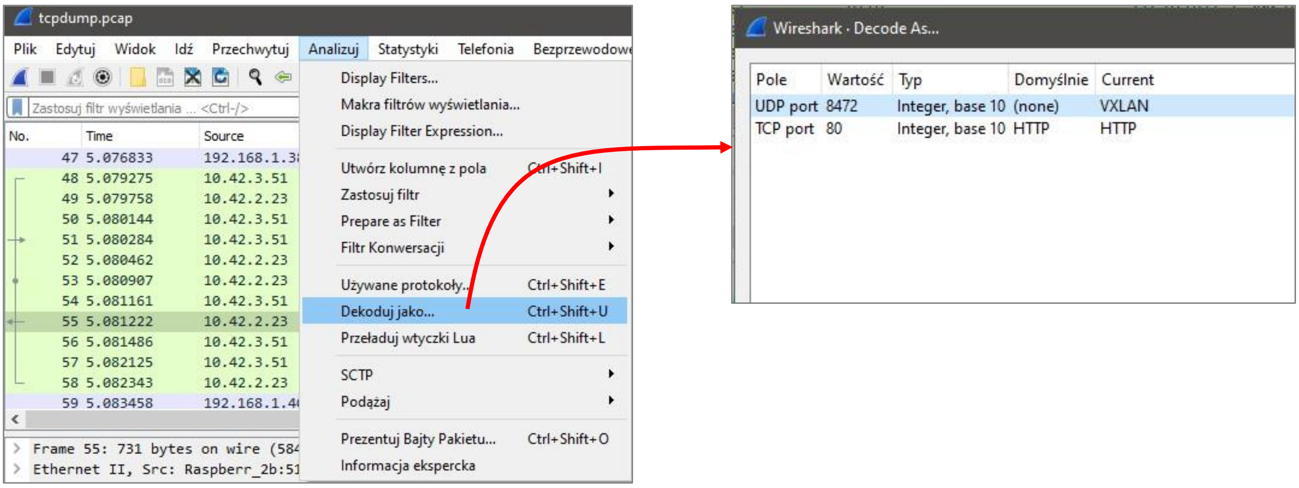
```
# Get to the terminal of cluster node where the busybox pod runs
xubuntu@xubulab:~$ ssh kpi093
ubuntu@kpi093:~$ sudo tcpdump -s 0 -i eth0 -w tcpdump.pcap
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C113 packets captured
123 packets received by filter
0 packets dropped by kernel
ubuntu@kpi093:~$ ls
tcpdump.pcap
ubuntu@kpi093:~$ sudo chmod 644 tcpdump.pcap
ubuntu@kpi093:~$
```

[3] Alternatively, you can use the Wireshark remote packet capture approach as described in this guide. You won't need to download the tcap file.

```
# Get to the terminal of the busybox container in the busy box pod to wget nginx from there
xubuntu@xubulab:~$ kubectl exec -it -c busybox busybox -- /bin/sh

/ # wget 10.42.2.23 -O - 2>/dev/null
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
/ #
```

```
# Terminal on the host where Wireshark will be run
xubuntu@xubulab:~/cluster-pi/lab$ kubectl get pods -o wide
NAME            READY   STATUS    RESTARTS      AGE   IP           NODE     NOMINATED NODE
READINESS GATES
dnsutils        1/1     Running   58 (19h ago)  18d   10.42.3.50   kpi093   <none>           <none>
nginx           1/1     Running   2 (19h ago)   29h   10.42.2.23   kpi092   <none>           <none>
busybox         1/1     Running   2 (19h ago)   29h   10.42.3.51   kpi093   <none>           <none>
nginx-busybox   2/2     Running   8 (19h ago)   9d    10.42.1.42   kpi094   <none>           <none>
xubuntu@xubulab:~/cluster-pi/lab$ scp ubuntu@192.168.1.40:/home/ubuntu/tcpdump.pcap ./
tcpdump.pcap                                                      100%   19KB   4.5MB/s   00:00
xubuntu@xubulab:~/cluster-pi/lab$
```

Once we download the tcpdump.pcap file, we can start analyzing it in Wireshark. Remember to configure Wireshark to decode VXLAN traffic first. To this end, use the `Analyse/Decode as/` window to adjust the settings for UDP and HTTP protocols as shown in figure **Figure 3** (set the columns Field, Value and Current).


tcpdump.pcap
Plik
Edytuj
Widok
Idź
Przechwytuj
Analizuj
Statystyki
Telefonia
Bezprzewodowe
Zastosuj filtr wyświetlania ... <Ctrl-/>
Display Filters...
Makra filtrów wyświetlania...
Display Filter Expression...
Utwórz kolumnę z pola Ctrl+Shift+I
Zastosuj filtr
Prepare as Filter
Filtr Konwersacji
Używane protokoły... Ctrl+Shift+E
Dekoduj jako... Ctrl+Shift+U
Przeładuj wtyczki Lua Ctrl+Shift+L
SCTP
Podążaj
Prezentuj Bajty Pakietu... Ctrl+Shift+O
Informacja ekspercka
Wireshark · Decode As...

No.	Time	Source
47	5.076833	192.168.1.3
48	5.079275	10.42.3.51
49	5.079758	10.42.2.23
50	5.080144	10.42.3.51
51	5.080284	10.42.3.51
52	5.080462	10.42.2.23
53	5.080907	10.42.2.23
54	5.081161	10.42.3.51
55	5.081222	10.42.2.23
56	5.081486	10.42.3.51
57	5.082125	10.42.3.51
58	5.082343	10.42.2.23
59	5.083458	192.168.1.4

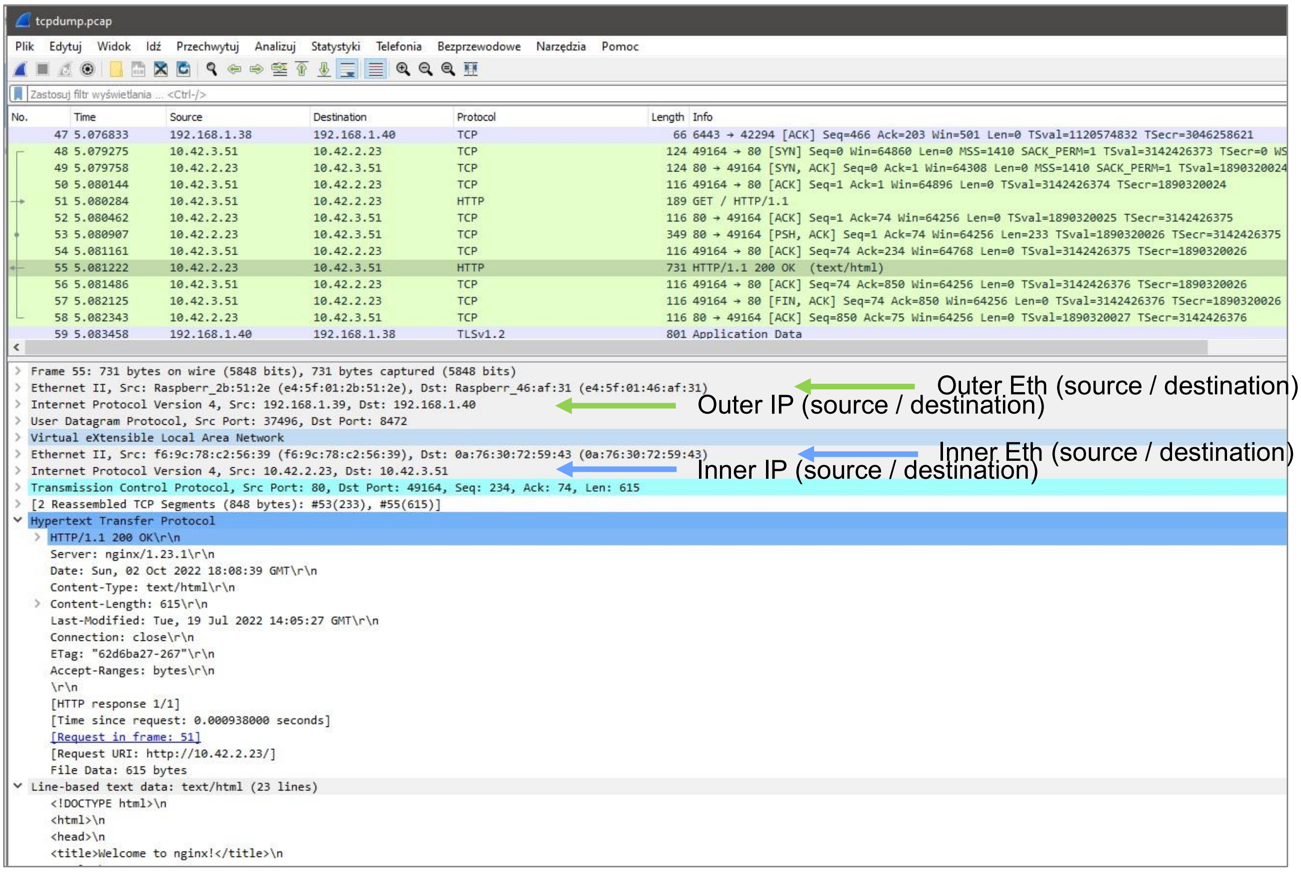
```
> Frame 55: 731 bytes on wire (584
> Ethernet II, Src: Raspberr_2b:51
```

Pole	Wartość	Typ	Domyślnie	Current
UDP port	8472	Integer, base 10	(none)	VXLAN
TCP port	80	Integer, base 10	HTTP	HTTP

Figure 3 Setting Wireshark to decode UDP/VXLAN.

Decoded trace will resemble the one shown in **Figure 4**. Decoded HTTP OK message received in response to wget/GET is shown below. One can easily identify all elements of VXLAN stack in this screenshot.

> *Notice: we did our best to keep the IP and MAC addresses of the entities involved (pods, cni0, flannel.1) consistent in this presentation, but it may happen that some of them in current point (d) are different form the addresses presented in previous points (a), (b), (c). This is because internal pod addresses are ephemeral in Kubernetes and change between restarts of the cluster, and we run the final packet capture session after such a restart. Fortunately, the assignment of new addresses in a figure similar to* **Figure 2** *can easily be inferred from your packet capture.*


tcpdump.pcap
Plik
Edytuj
Widok
Idź
Przechwytuj
Analizuj
Statystyki
Telefonia
Bezprzewodowe
Narzędzia
Pomoc
Zastosuj filtr wyświetlania ... <Ctrl-/>
Outer Eth (source / destination)
Outer IP (source / destination)
Inner Eth (source / destination)
Inner IP (source / destination)

```
No.   Time        Source          Destination     Protocol   Length Info
   47 5.076833    192.168.1.38    192.168.1.40    TCP            66 6443 → 42294 [ACK] Seq=466 Ack=203 Win=501 Len=0 TSval=1120574832 TSecr=3046258621
   48 5.079275    10.42.3.51      10.42.2.23      TCP           124 49164 → 80 [SYN] Seq=0 Win=64860 Len=0 MSS=1410 SACK_PERM=1 TSval=3142426373 TSecr=0 WS
   49 5.079758    10.42.2.23      10.42.3.51      TCP           124 80 → 49164 [SYN, ACK] Seq=0 Ack=1 Win=64308 Len=0 MSS=1410 SACK_PERM=1 TSval=1890320024
   50 5.080144    10.42.3.51      10.42.2.23      TCP           116 49164 → 80 [ACK] Seq=1 Ack=1 Win=64896 Len=0 TSval=3142426374 TSecr=1890320024
   51 5.080284    10.42.3.51      10.42.2.23      HTTP          189 GET / HTTP/1.1
   52 5.080462    10.42.2.23      10.42.3.51      TCP           116 80 → 49164 [ACK] Seq=1 Ack=74 Win=64256 Len=0 TSval=1890320025 TSecr=3142426375
   53 5.080907    10.42.2.23      10.42.3.51      TCP           349 80 → 49164 [PSH, ACK] Seq=1 Ack=74 Win=64256 Len=233 TSval=1890320026 TSecr=3142426375
   54 5.081161    10.42.3.51      10.42.2.23      TCP           116 49164 → 80 [ACK] Seq=74 Ack=234 Win=64768 Len=0 TSval=3142426375 TSecr=1890320026
   55 5.081222    10.42.2.23      10.42.3.51      HTTP          731 HTTP/1.1 200 OK  (text/html)
   56 5.081486    10.42.3.51      10.42.2.23      TCP           116 49164 → 80 [ACK] Seq=74 Ack=850 Win=64256 Len=0 TSval=3142426376 TSecr=1890320026
   57 5.082125    10.42.3.51      10.42.2.23      TCP           116 49164 → 80 [FIN, ACK] Seq=74 Ack=850 Win=64256 Len=0 TSval=3142426376 TSecr=1890320026
   58 5.082343    10.42.2.23      10.42.3.51      TCP           116 80 → 49164 [ACK] Seq=850 Ack=75 Win=64256 Len=0 TSval=1890320027 TSecr=3142426376
   59 5.083458    192.168.1.40    192.168.1.38    TLSv1.2       801 Application Data
```

```
> Frame 55: 731 bytes on wire (5848 bits), 731 bytes captured (5848 bits)
> Ethernet II, Src: Raspberr_2b:51:2e (e4:5f:01:2b:51:2e), Dst: Raspberr_46:af:31 (e4:5f:01:46:af:31)
> Internet Protocol Version 4, Src: 192.168.1.39, Dst: 192.168.1.40
> User Datagram Protocol, Src Port: 37496, Dst Port: 8472
> Virtual eXtensible Local Area Network
> Ethernet II, Src: f6:9c:78:c2:56:39 (f6:9c:78:c2:56:39), Dst: 0a:76:30:72:59:43 (0a:76:30:72:59:43)
> Internet Protocol Version 4, Src: 10.42.2.23, Dst: 10.42.3.51
> Transmission Control Protocol, Src Port: 80, Dst Port: 49164, Seq: 234, Ack: 74, Len: 615
> [2 Reassembled TCP Segments (848 bytes): #53(233), #55(615)]
v Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
    Server: nginx/1.23.1\r\n
    Date: Sun, 02 Oct 2022 18:08:39 GMT\r\n
    Content-Type: text/html\r\n
  > Content-Length: 615\r\n
    Last-Modified: Tue, 19 Jul 2022 14:05:27 GMT\r\n
    Connection: close\r\n
    ETag: "62d6ba27-267"\r\n
    Accept-Ranges: bytes\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.000938000 seconds]
    [Request in frame: 51]
    [Request URI: http://10.42.2.23/]
    File Data: 615 bytes
v Line-based text data: text/html (23 lines)
    <!DOCTYPE html>\n
    <html>\n
    <head>\n
    <title>Welcome to nginx!</title>\n
```

Figure 4 Decoding the communication over flannel VXLAN.

The key observation from this experiment relates to the role of CNI in separating cluster traffic in the physical network infrastructure where the cluster has been deployed.

4. Installing MetalLB from manifest files

4.1. About MetaLB

Kubernetes does not offer an implementation of network load balancers (Services of type LoadBalancer[4]) for bare-metal clusters. The implementations of network load balancers that Kubernetes does ship with are all glue code that calls out to various IaaS platforms (GCP, AWS, Azure…). If you're not running on a supported IaaS platform (GCP, AWS, Azure…), LoadBalancers will remain in the "pending" state indefinitely when created. In contrast to tat, by default, K3s provides a load balancer known as ServiceLB (formerly Klipper LoadBalancer). However, ServiceLB uses available host ports so the load balancing model it provides is limited. That is the reason we have installed our k3s cluster without load balancer[5].

In plain Kubernetes, bare-metal cluster operators are left with two lesser tools to bring user traffic into their clusters, `NodePort` and `ExternalIPs` services. Both of these options have significant downsides for production use, which makes bare-metal clusters second-class citizens in the Kubernetes ecosystem. MetalLB aims to supress this imbalance by offering a network load balancer implementation that integrates with standard network equipment, so that external services on bare-metal clusters also "just work" as much as possible.

MetalLB hooks into your Kubernetes cluster, and provides a network load-balancer implementation. In short, it allows you to create Kubernetes services of type LoadBalancer in clusters that don't run on a cloud provider, and thus cannot simply hook into paid products to provide load balancers. It allows enabling `LoadBalancer` service addresses in any bare-metal Kubernetes installation using standard routing protocols. It has two features that work together to provide this service: **address allocation**, and **external announcement**.

- Address allocation

In a Kubernetes cluster on a cloud provider, you request a load balancer, and your cloud platform assigns an IP address to you. In a private bare-metal cluster, MetalLB is responsible for that allocation.

MetalLB cannot create IP addresses out of thin air and you have to allocate pools of IP addresses that it can use. MetalLB will take care of assigning and unassigning individual addresses as services come and go, but it will only hand out IPs that are part of its configured pools. Below in point 4.4, we will configure a pool of IP addresses for Metallb in our cluster.

- External announcement

After MetalLB has assigned an external IP address to a service, it needs to make the network beyond the cluster aware that the IP "lives" in the cluster. MetalLB uses standard networking or routing protocols to achieve this, depending on which mode is used: layer 2 ARP/NDP, or layer 3 BGP.

In layer 2 mode, one machine (typically the master node) in the cluster takes ownership of the service, and uses standard address discovery protocols (ARP for IPv4, NDP for IPv6) to make those IPs reachable on the local network. From the LAN's point of view, the announcing machine simply has multiple IP addresses. Under the hood, MetalLB responds to ARP requests for IPv4 services, and NDP requests for IPv6. The major advantage of the layer 2 mode is its universality: it will work on any Ethernet network, with no special hardware required, not even fancy routers. This is the mode we are going to use in our cluster. More on layer 2 mode can be found here: https://metallb.universe.tf/concepts/layer2/.

In BGP mode, all machines in the cluster establish BGP peering sessions with nearby routers that you control, and tell those routers how to forward traffic to the service IPs. Using BGP allows for true load balancing across multiple nodes, and fine-grained traffic control thanks to BGP's policy mechanisms. Operating external router(s) is the additional cost to be paid for this flexibility. More on BGP mode can be found here: https://metallb.universe.tf/concepts/bgp/.

MetalLB speaker is in charge of IP advertisement. It is based on leader election on per-service-instance basis so that traffic incoming to the cluster can be load balanced evenly among cluster nodes. Leader election is deterministic based

[4] A quick introduction to service types in Kubernetes would be helpful here. For example this video can serve the purpose: https://www.youtube.com/watch?v=T4Z7visMM4E. Many other sources are available in the net.

[5] See the *master* role in our Ansible playbook.

on hashes so no consensus protocol is needed for the speakers for consistent election decision. Current leader is kept track of by remaining speakers based on updating a dedicated resource. Actually, it is kube-proxy on the leader node of the service which load balances service's traffic (connections) among service's pods. However, there is a restriction that can limit the access to services through the ingress mechanism if nodes have taints that disallow MetalLB speakers on them: the speaker pod that announces the external IP address for a service must be on the same node as an endpoint (a pod) for the service and the endpoint must be in the Ready condition.

More on MetalLB can be found here: https://metallb.universe.tf/concepts/ and https://docs.openshift.com/container-platform/4.9/networking/metallb/about-metallb.html

4.2. Installation of MetalLB

a) Pre-installation check of services running in the cluster

Before installation, lets list all services in our cluster created as a result of cluster installation. Notice that the Traefik service of Kubernetes type LoadBalancer has not been assigned EXTERNAL-IP address (Traefik is the ingress controller in our configuration). That means that there were no service running to make this assignment – in fact, we installed k3s with the embedded k3s load balancer disabled setting `--disable servicelb` (ServiceLB is the embedded load balancer in K3s). Wewill install MetalLB in our cluster instead of ServiceLB and once it's up and running with IP address ranges to assign from configured (sec. 4.4), it is expected to assign EXTERNAL-IP address to Traefik.

```
xubuntu@xubulab:~/cluster-pi$ kubectl get services -A
NAMESPACE     NAME              TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)                      AGE
default       kubernetes        ClusterIP      10.43.0.1       <none>         443/TCP                      3d3h
kube-system   kube-dns          ClusterIP      10.43.0.10      <none>         53/UDP,53/TCP,9153/TCP       3d3h
kube-system   metrics-server    ClusterIP      10.43.172.124   <none>         443/TCP                      3d3h
kube-system   traefik           LoadBalancer   10.43.42.129    <pending>      80:31446/TCP,443:31051/TCP   3d3h
```

b) Labelling worker nodes – just for any case

For each worker node add label "worker" by issuing the following command. Setting his label may turn out to be useful during future operations. Note: in RPi5 clusters, we suggest that nodes including the *master* are labelled *worker.*

```
$ kubectl label node <node-name> node-role.kubernetes.io/worker=true
```

c) Installing MetalLB

Execute the command in the frame below and observe the output similar to the one presented. **Before that, always check the current version of MetalLB** https://metallb.universe.tf/installation/#installation-by-manifest **and use it in your `kubectl apply` command** (in our example, version v0.14.9 was used, but a newer one can be available now).

```
xubuntu@xubulab:~/cluster-pi$ kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.14.9/config/manifests/metallb-native.yaml
namespace/metallb-system created
customresourcedefinition.apiextensions.k8s.io/bfdprofiles.metallb.io created
customresourcedefinition.apiextensions.k8s.io/bgpadvertisements.metallb.io created
customresourcedefinition.apiextensions.k8s.io/bgppeers.metallb.io created
customresourcedefinition.apiextensions.k8s.io/communities.metallb.io created
customresourcedefinition.apiextensions.k8s.io/ipaddresspools.metallb.io created
customresourcedefinition.apiextensions.k8s.io/l2advertisements.metallb.io created
customresourcedefinition.apiextensions.k8s.io/servicel2statuses.metallb.io created
serviceaccount/controller created
serviceaccount/speaker created
role.rbac.authorization.k8s.io/controller created
deployment.apps/controller created
daemonset.apps/speaker created
validatingwebhookconfiguration.admissionregistration.k8s.io/metallb-webhook-configuration
created
. . .
validatingwebhookconfiguration.admissionregistration.k8s.io/metallb-webhook-configuration
created
xubuntu@xubulab:~/cluster-pi$
```

4.3. Checking if MetalLB has been installed

Depending on the complexity of the pod's software it may bee needed to wait a while until the pod gets READY. Max time counts in minutes (say, one – two minutes). Otherwise check pod's status `kubectl describe pod <pod name>` and/or the pod's logs for errors `kubectl logs <pod name>` (https://spacelift.io/blog/kubectl-logs).

```
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl get deployments --all-namespaces
NAMESPACE        NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
kube-system      coredns                  1/1     1            1           1h
kube-system      local-path-provisioner   1/1     1            1           1h
kube-system      metrics-server           1/1     1            1           1h
kube-system      traefik                  1/1     1            1           1h
metallb-system   controller               1/1     1            1           1h
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl get pods -n metallb-system -o wide
NAME                          READY   STATUS    RESTARTS       AGE     IP             NODE     NOMINATED NODE
speaker-7n2lg                 1/1     Running   0              2h15m   192.168.1.40   kpi093   <none>
speaker-nwwpk                 1/1     Running   0              2h15m   192.168.1.41   kpi094   <none>
speaker-wc6r4                 1/1     Running   0              2h15m   192.168.1.39   kpi092   <none>
controller-6c58495cbb-fwzbn   1/1     Running   1 (2h14m ago)  2h15m   10.42.2.5      kpi093   <none>
```

As seen, one controller deployment and a daemon-set with three speakers have been created (in section 5.4 we discuss why three speaker are run and how to enable metallb speakers also on master/control nodes). For metallb working in L2 mode, in a given LAN segment, metallb speaker is in charge of ARP IP advertisements for those services for which it was assigned to be the leader.

Waring: If you happen to see ImagePullBackOff STATUS of a pod, please be patient and wait for a couple of minutes. This is probably because of some temporary network problems and excessive delay experienced in accessing the image of the container. The problem should be resolved by Kubernetes controller without your intervention.

4.4. Configuring IP address pool for MetalLB

Instructions according to: https://metallb.universe.tf/configuration/

Defining the IPs to assign to the Load Balancer services and announce the service Ips to be used in Layer 2 configuration.

d) On the configuration host, create directory `metallb`

e) Create `ip-pool-config.yaml` in directory `metallb` with the following content. Adjust address range to your environment. You can always modify the IP range in `ip-pool-config.yaml` if you want, and `apply` it again.

```
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ nano ip-pool-config.yaml
---

apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: first-pool
  namespace: metallb-system
spec:
  addresses:
  - 192.168.1.200-192.168.1.254    # adjust address ranges according to your environment

---

apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: l2-pool
  namespace: metallb-system
spec:
  ipAddressPools:
 - first-pool
```

f) Apply the config and check the result

```
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl apply -f ip-pool-config.yaml
ipaddresspool.metallb.io/first-pool created
```

```
l2advertisement.metallb.io/l2-pool created
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl get -n metallb-system ipaddresspools
NAME         AGE
first-pool   6m45s
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl describe -n metallb-system ipaddresspool
first-pool
Name:         first-pool
Namespace:    metallb-system
Labels:       <none>
Annotations:  <none>
API Version:  metallb.io/v1beta1
Kind:         IPAddressPool
Metadata:
  Creation Timestamp:  2022-10-10T20:12:33Z
  Generation:          1
  Managed Fields:
    API Version:  metallb.io/v1beta1
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .:
          f:kubectl.kubernetes.io/last-applied-configuration:
      f:spec:
        .:
        f:addresses:
        f:autoAssign:
        f:avoidBuggyIPs:
    Manager:         kubectl-client-side-apply
    Operation:       Update
    Time:            2022-10-10T20:12:33Z
  Resource Version:  115222
  UID:               080a5c65-6810-46ba-b341-28f020c54be0
Spec:
  Addresses:
    192.168.1.200-192.168.1.254
  Auto Assign:       true
  Avoid Buggy I Ps:  false
Events:              <none>
```

g) Check all resources (including custom resources) in a given namespace (here metallb-system):

```
xubuntu@xubulab:~$ kubectl api-resources --verbs=list --namespaced -o name \
   | xargs -n 1 kubectl get --show-kind --ignore-not-found -n metallb-system
```

(Note: `kubectl get` command get does not get custom resources. All resource types in Kubernetes cluster are displayed using `kubectl api-resources` command. See, e.g., https://www.studytonight.com/post/how-to-list-all-resources-in-a-kubernetes-namespace)

h) Check Traefik EXTERNAL-IP address after having configured MetalLB. Observe an assigned value – address that should be reachable on the HTTP level from the outside of the cluster.

```
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl get svc -n kube-system
NAME             TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)
kube-dns         ClusterIP      10.43.0.10      <none>          53/UDP,53/TCP,9153/TCP
metrics-server   ClusterIP      10.43.172.124   <none>          443/TCP
traefik          LoadBalancer   10.43.42.129    192.168.1.200   80:31446/TCP,443:31051/TCP
```

5. Checking Traefik with external IP and basic load balancing

5.1. Checking for errors (just for any case)

a) (optional) Check the logs again, this time for errors (below, we use a command slightly different form the one used previously but producing similar output)

```
kubectl logs $(kubectl get pods --namespace kube-system | grep "^traefik" | awk
'{print $1}') --namespace kube-system
```

b) In case of getting `Error status` for pods, you can try restarting the failing pod:

https://www.containiq.com/post/using-kubectl-to-restart-a-kubernetes-pod

c) **Finally, when everything works fine**, just out of curiosity we can also check the MAC address of MetalLB speaker that has taken leadership for Traefik – run the following on the management host and catch ARP responses to "who has 192.168.1.200" (adjust the IP address of Traefik and the interface name of the management host according to your environment if needed):

```
$ sudo tcpdump -ennqti enp0s3 # this tcpdump is optional, arping will suffice
$ arping 192.168.1.200 -c 5
```

5.2. Checking the access to Traefik dashboard

In this part of the lab we will check the access to Traefik dashboard and will see that Traefik can be managed from a browser. However, we do not focus specifically on the details of Traefik configuration during this lab. Instead, our primary goal is to tackle the LoadBalancer mechanism again with an addition of an important mechanism known as *ingress*. Actually, we will use the IngressRoute utility specific to Traefik (similar to Kubernetes Ingress, still not the same).

In Kubernetes, Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Many such routes can use the same Kubernetes service (e.g., share common LoadBalancer address). HTTP traffic routing is controlled by rules defined on the Ingress resource. To touch upon this topic, we will try a very simple configuration of Traefik's IngressRoute feature to enable the access to Traefik dashboard[6] from the outside of the cluster.

a) Using a browser, check accessing from the outside of the cluster: **http://192.168.1.200/dashboard/**

b) **You will most probably get the error 404** and will need to update the IngressRoute CR[7] for Traefik manually:

- go to your manifests/traefik directory and retrieve the ingress route for Traefik by running

  `kubectl get ingressroutes -n kube-system traefik-dashboard -o yaml > traefik-dashboard-ingress.yaml` (in our gihub repo, on traefik directory, you may find two reference files named traefik-dashboard-ingress.yaml.db and traefik-service.yaml.db)

- in file `traefik-dashboard-ingress.yaml` find the `spec` section and check whether it has the value of `entryPoints` set to '`- traefik`'. If so, change '`- traefik`' to '`- web`' and the IngressRoute should now allow routing of external queries to the Traefik dashboard:
- edit: `kubectl edit -n kube-system ingressroutes.traefik.io traefik-dashboard -o yaml`

  and change entrypoint name "traefik" for "web", and save (use the *vi* editor keying):

```
spec:
  entryPoints:
    - web
```

[6] The Ingress provided by Traffic serves to expose Traffic dashboard over HTTP(S) to the outside world (our entry point "web" allows for HTTP only). Thus, from the point of view of Traffic Ingress mechanism, Traffic dashboard is a „regular" Kubernetes service.

[7] Presented method using `ingressroutes.traefik.io` (actually, the name of CRD) instead of regular plural/singular for the IngressRoute custom resource (ingressroutes or ingressroute) is odd, but we use it for now while waiting for a clarification by the Traefik team (a query is pending).

- having changed the spec section, now remove all annotations, etc. to leave only apiVesion, kind, and the two following fields in the *metadata* part: `metadata: name: traefik-dashboard namespace: kube-system`
- then apply: `kubectl apply -f traefik-dashboard-ingress.yaml` (ignore warnings if you applied the changed manifest without cleaning described in the previous bullet point)
- check again: http://192.168.1.200/dashboard/ **(remember to put the closing sign "/", otherwise you will get HTTP 404)**

Note: the adjustment of the ingress route was accomplished using `kubectl edit` command (in this case, saving the changes with the ***vi*** command ***:w*** automatically triggers updating the resource by Kubernetes). **Remember that this change will persist as long as you do not reinstall Traefik**. If you want it to persist use yaml manifest for the IngressRoute and the *kubectl apply* operation.

c) you can confirm the IngressRoute CR for Traefik has been changed using `kubectl edit` (**you'd better not save it**, i.e., quit the *vi* editor by entering `:q!`)

```
kubectl edit ingressroute -n kube-system traefik-dashboard
```

d) **NOTE: if Traefik dashboard persists unreachable** despite introducing the changes described above it is most likely because of the fact that we set additional taint **`CriticalAddonsOnly=true:NoExecute`** for the master node during k3s installation in Ansible (refer to the Ansible play for the master role in your Ansible files). This results in metallb speakers not being deployed on the master node which seems to be the source of problems (in fact, it should not be so, but the experience suggests it may happen). All in all, if you run into this problem you need to manually update DaemonSet resource/manifest as described in section 5.4. This situation is odd, but as it happened once to me it is worth commenting at least on a basic level. Hopefully you are not going to face it.

e) You can set static IP address for the LoadBalancer service. In our example, it is sufficient to add loadBalancerIP attribute to the service resource as follows (notice that the service can have a different name in your deployment of k3s, e.g. *traefic*):

```
$ kubectl get svc traefik-service -o yaml > traefik-service.yaml

# edit traefik-svc.yaml to add loadBalancerIP attribute in the spec section:
$ nano traefik-service.yaml
...
spec
   ...
   type: LoadBalancer
   loadBalancerIP: 192.168.1.200 # use any unassigned IP address from the metallb pools
$ kubectl apply -f traefik-service.yaml  (ignore warnings)
```

You can use any unassigned address from one of the existing MetalLB address pools. If you omit the line **`loadBalancerIP: 192.168.1.200`**, then MetalLB will assign a "random" IP address form its pool.

Another method to enable Traefik dashboard, using Kubernetes Ingress and service is described here.

5.3. Checking basic (in-cluster) load balancing

This section relates to internal cluster load balancing that load balances the load among multiple pods (replicas) of a given Service instance. It is beyond MetalLb load balancing (another level) and kube-proxy responsible for the configuration of iptables in cluster nodes is the main actor in this case.

Create deployment.yaml and service.yaml manifests for nginx as given in the frame below and run the set of commands from the following frame. Notice that our deployment requests 3 replicas of nginx to load balance between them. They can be seen as three `Endpoints: 10.42.1.24:80,10.42.2.17:80,10.42.3.17:80` displayed with the command `kubectl describe` below. You can can check on which nodes they are instantiated by running `kubectl get pods -o wide`.

*Notice: in my case, it once happened that one replica of nginx was instantiated on the master node which is odd considering the fact that my master was installed as in Lab1, i.e., with flag node-taint set to `--node-taint CriticalAddonsOnly=true:NoExecute`. The latter taint is expected to disable workload execution on the master. My conjecture is that setting the taint may not work properly with the install method based on **`curl -sfL https://get.k3s.io`**. I've left more detailed investigation of this behaviour for the future. If you observe it in your cluster, a provisional workaround that seems to work well is to run manually the following:*

- `kubectl taint nodes <master-node-name> CriticalAddonsOnly=true:NoExecute`
- `kubectl describe node <master-node-name> # to check the change is in effect`

Manifests

```
# deployment.yaml content
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
          ports:
            - containerPort: 80

# service.yaml content
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
    - port: 80
      targetPort: 80
  type: LoadBalancer
```

Commands

```
xubuntu@xubulab:~/cluster-pi/manifests$ kubectl apply -f example/deployment.yaml
deployment.apps/nginx created
xubuntu@xubulab:~/cluster-pi/manifests$ kubectl apply -f example/service.yaml
service/nginx created

# display the endpoints (replica Pods)

xubuntu@xubulab:~/cluster-pi/manifests$ kubectl describe service nginx
Name:                     nginx
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.43.71.110
IPs:                      10.43.71.110
LoadBalancer Ingress:     192.168.1.201
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32456/TCP
Endpoints:                10.42.1.24:80,10.42.2.17:80,10.42.3.17:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason        Age   From                Message
  ----    ------        ----  ----                -------
  Normal  IPAllocated   8s    metallb-controller  Assigned IP ["192.168.1.201"]
  Normal  nodeAssigned  8s    metallb-speaker     announcing from node "worker-2" with protocol
"layer2"

# reachability check
# here, you can also check the nodes where replicas are running (see the description above)

xubuntu@xubulab:~/cluster-pi/manifests$ curl 192.168.1.201
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>

# clean after checking

xubuntu@xubulab:~/cluster-pi/manifests$ kubectl delete -f example/deployment.yaml
deployment.apps "nginx" deleted
xubuntu@xubulab:~/cluster-pi/manifests$ kubectl delete -f example/service.yaml
service "nginx" deleted
```

5.4. Closing remarks - analysing the taint in the master for metallb speakers

You probably will not land in this section. If you do you hopefully find it instructive.

In this additional section, we analyse why metallb speaker is not run on the master node (if that’s the case in your cluster). The reason for that is the interplay between node *taints* and pod *tolerations* that express the willingness of a node to accept specific type of workloads, and the explicit permission for a given workload (pod) to be accepted by otherwise tainted node, respectively. To see this, we simply have to check and compare taints in the master node and tolerations of the speaker pods. An intended side effect of this part is the knowledge how simple placement algorithms can be tuned to optimize workload placement in Kubernetes clusters.

Note: we set the taint on the master node in the k3s install command run in Ansible playbook for the role "master"; by doing that we wanted to avoid running unnecessary workloads on the master. Therefore, as metallb speakers do not have sufficient tolerances defined they are blocked from running on the master.

Follow the commands in the frame to allow metallb speakers run on the master and analyse the taints and tolerations. Allowing metallb speakers to run on the master is optional – up to you. The effect will be that the traffic of some services (probably not too many in our case) will be routed through the master node thus contributing to additional load on this control element.

```
# get metallb pods to see there is no speaker running on master node (we saw it already during matallb
installation, but did not pay attention to)
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl get pods -n metallb-system
NAME                          READY   STATUS    RESTARTS        AGE
speaker-7n2lg                 1/1     Running   0               4h47m
speaker-nwwpk                 1/1     Running   0               4h47m
speaker-wc6r4                 1/1     Running   0               4h47m
controller-6c58495cbb-fwzbn   1/1     Running   1 (4h46m ago)   4h47m

# check the master to see it has not a speaker running on it; check the taints defined in the master
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl describe node kpi091
Name:               kpi091
Roles:              control-plane,etcd,master
(. . .)
Taints:             CriticalAddonsOnly=true:NoExecute
Unschedulable:      false
Lease:
  HolderIdentity:  kpi091
(. . .)

# describe any of the speakers to see it does not contain toleration against CriticalAddonsOnly (is
unexecutable on the master kpi091); speakers run as DaemonSet so they all share this restriction and it is
the reason why no speaker was scheduled on the master kpi091
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl describe -n metallb-system pod speaker- 7n2lg
Name:         speaker-7h8jj
Namespace:    metallb-system
Priority:     0
Node:         kpi093/192.168.1.40
(. . .)
Node-Selectors:              kubernetes.io/os=linux
Tolerations:                 node-role.kubernetes.io/control-plane:NoSchedule op=Exists
                             node-role.kubernetes.io/master:NoSchedule op=Exists
                             node.kubernetes.io/disk-pressure:NoSchedule op=Exists
                             node.kubernetes.io/memory-pressure:NoSchedule op=Exists
                             node.kubernetes.io/network-unavailable:NoSchedule op=Exists
                             node.kubernetes.io/not-ready:NoExecute op=Exists
                             node.kubernetes.io/pid-pressure:NoSchedule op=Exists
                             node.kubernetes.io/unreachable:NoExecute op=Exists
                             node.kubernetes.io/unschedulable:NoSchedule op=Exists

# We will allow speakers run on master nodes either directly from kubectl (not recommended) or modifying and
applying the yaml manifest. Both methods are described in the following.

# Directly form kubectl: according to the commands below, but refer to the edited DaemonSet shown later on
in this frame to see how the addition should exactly look like.
xubuntu@xubulab:~/cluster-pi/manifests$ kubectl edit -n metallb-system daemonset speaker   <= edit manually
daemonset.apps/speaker edited
xubuntu@xubulab:~/cluster-pi/manifests$ kubectl get pods -n metallb-system
NAME                          READY   STATUS              RESTARTS      AGE
controller-6c58495cbb-fwzbn   1/1     Running             2 (16h ago)   23h
speaker-vcq4k                 1/1     Running             0             67s
speaker-2wf45                 1/1     Running             0             35s
speaker-l6tz8                 1/1     Running             0             14s
speaker-vp2xw                 0/1     ContainerCreating   0             2s
xubuntu@xubulab:~/cluster-pi/manifests$ kubectl get pods -n metallb-system
NAME                          READY   STATUS    RESTARTS      AGE
controller-6c58495cbb-fwzbn   1/1     Running   2 (16h ago)   23h
speaker-vcq4k                 1/1     Running   0             92s
speaker-2wf45                 1/1     Running   0             60s
speaker-l6tz8                 1/1     Running   0             39s
speaker-vp2xw                 1/1     Running   0             27s
```

```
# Through the yaml file: by exporting the manifest to a file, modifying it and applying again updated
speaker daemonset as shown below in this frame. The overall procedure: delete speaker daemonset, add
toleration CriticalAddonsOnly=true:NoExecute in exported daemonset maniefest, re-apply daemonset.

xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl get daemonsets --all-namespaces
NAMESPACE        NAME      DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
metallb-system   speaker   4         4         4       4            4           kubernetes.io/os=linux

# delete daemonset
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl delete -n metallb-system daemonset speaker
daemonset.apps "speaker" deleted

# Edit speaker daemonset to add the toleration; steps 1, 2, 3:

# 1. Download metalb manifest from
https://raw.githubusercontent.com/metallb/metallb/v0.14.5/config/manifests/metallb-native.yaml and extract
and save complete Daemonset manifest (search Daemonset in the downloaded file) to a separate yaml file, say
speaker-daemonset.yaml.

# 2. Now edit this new file to add the missing toleration (see below and find the update part)
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ nano speaker-daemonset.yaml

# edited daemonset manifest - relevant part adding the toleration needed:
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    app: metallb
    component: speaker
  name: speaker
  namespace: metallb-system
spec:
  selector:
    matchLabels:
      app: metallb
      component: speaker
(. . .)
      tolerations:
      - effect: NoSchedule
        key: node-role.kubernetes.io/master
        operator: Exists
      - effect: NoSchedule
        key: node-role.kubernetes.io/control-plane
        operator: Exists
# this is to be added: toleration to enable running a speaker on the master node
      - effect: NoExecute
        key: CriticalAddonsOnly
        operator: Equal
        value: "true"
---

# 3. apply modified daemonset manifest to re-instantiate the speakers
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl apply -f speaker-daemonset.yaml
daemonset.apps/speaker created

# 4. get pods to check if there is a speaker running on every node
xubuntu@xubulab:~/cluster-pi/manifests/metallb$ kubectl get pods -n metallb-system -o wide
NAME                         READY   STATUS    RESTARTS      AGE     IP             NODE     NOMINATED NODE
controller-6d5cb87f6-5hkhl   1/1     Running   0             52s     10.42.3.75     kpi093   <none>
speaker-2qs5m                1/1     Running   0             52s     192.168.1.38   kpi091   <none>
speaker-66b5m                1/1     Running   0             52s     192.168.1.39   kpi092   <none>
speaker-l7njq                1/1     Running   0             52s     192.168.1.41   kpi094   <none>
speaker-n7mps                1/1     Running   0             52s     192.168.1.40   kpi093   <none>
```

6. Questions to answer

1. Copy here the red part of text from section 3.3, p. c) after completing it with data (the IP addresses) taken from YOUR cluster.
2. Explain which of the premises of inter-Pod communication in Kubernetes cluster is supported by device flannel.1 (not flannel as a solution, but flannel.1 device in particular). Remember that each Kubernetes node has a subnet and each Pod has an IP from this subnet (refer to the lecture slide deck).
3. Which kernel structures are handled by flanneld so that flannel.1 VXLAN interface can work (route) properly?
4. Why using LoadBalancer such as MetalLB for exposing services outside of the cluster is better suited for load balancing traffic among cluster nodes than using the Ingress controller mechanism?
5. In section 5 we have seen an example application of Ingress to expose Traefik dashboard as HTTP page. Could other HTTP(S) services in the cluster be exposed with Ingress on the same loadBalancerIP address as the one used by our Traefik dashboard (192.168.1.200 in the example above)? Provide an explanation to your answer.

7. Additional readings/videos and hints

This is a record of additional interesting/valuable sources from among many more used to prepare this lab.

- Section 3 related references:
  - https://www.sysspace.net/post/kubernetes-networking-explained-flannel-network-model
  - https://msazure.club/flannel-networking-demystify/
  - https://www.sysspace.net/post/kubernetes-networking-explained-flannel-network-model_
  - How containers inside a pod communicate: https://www.redhat.com/sysadmin/kubernetes-pod-network-communications
  - Kubernetes network stack fundamentals: How pods on different nodes communicate: https://www.redhat.com/sysadmin/kubernetes-pods-communicate-nodes
  - Capture packets in Kubernetes: https://www.redhat.com/sysadmin/capture-packets-kubernetes-ksniff
  - Kubernetes troubleshooting: https://www.redhat.com/sysadmin/kubernetes-troubleshooting
  - Check flannel forwarding base (on cluster node): `$ bridge fdb show dev flannel.1`
  - https://serverfault.com/questions/988736/possible-to-list-members-of-a-network-bridge
  - Digging into linux namespaces:
    - https://blog.quarkslab.com/digging-into-linux-namespaces-part-1.html
    - https://blog.quarkslab.com/digging-into-linux-namespaces-part-2.html
- **Fully Automated K3s** etcd High Availability Install and more:
  https://docs.technotim.live/posts/k3s-etcd-ansible/
  https://github.com/techno-tim/k3s-ansible

- In case of problems with a given namespace, it is sometimes best to delete the whole namespace and recreate it. https://stackoverflow.com/questions/47128586/how-to-delete-all-resources-from-kubernetes-one-time
- Pod troubleshooting https://able8.medium.com/automated-troubleshooting-of-kubernetes-pods-issues-c6463bed2f29
- Flannel video, detailed, good:
  https://www.youtube.com/watch?v=U35C0EPSwoY&list=PLSAko72nKb8QWsfPpBlsw-kOdMBD7sra-&index=2
- What is VXLAN and How It is Used as an Overlay Network in Kubernetes?
  https://www.youtube.com/watch?v=WMLSD2y2Ig4

- If you want to experiment with Kubernetes certificates check this You can use our cluster of use kind or minikube to start safely):
  - https://www.fullstaq.com/knowledge-hub/blogs/setting-up-your-own-k3s-home-cluster
  - https://www.youtube.com/watch?v=hoLUigg4V18