Tutorial: Kubernetes BGP
True Load Balancer for Datacenters

OCT-2023

Mau Rojas
bio.site/pinrojas

MetalLB
Abstract

• LoadBalancer is a highly sought-after Kubernetes service, but it lacks an out-of-the-box solution for on-premises clusters.
• This presents a challenge for operators who have limited options like "NodePort" that come with drawbacks for production environments.
Abstract

• MetalLB comes to the rescue by seamlessly integrating with your Kubernetes cluster, providing a network load balancer implementation.

• By leveraging standard routing protocols and tight integration with your Datacenter Fabric, such as BGP and ECMP, MetalLB enables the creation of Load Balancer services in non-cloud environments.

• In this tutorial, we will guide you through building your own container-based lab to explore and experience MetalLB firsthand.
Why should I care?

- Kubernetes does not offer, by default, an implementation of a Load Balancer service for on-premises clusters.
What would I get here?

In this tutorial you will see:

• Intro
  • Kubernetes Load Balancer Services
  • MetalLB
  • Kubernetes Kind

• How to create a network lab using containerlab and Kind

• How to set up and test a BGP Load Balancer Services Between your Fabric and Kubernetes.
  • We’ll walk around BGP and ECMP settings
**K8s Native Services: What/Why?**

- Each pod has its own IP address
- Pods are ephemeral – are destroy frequently

- Kubernetes Services
  - Stable IP address
  - Load-balancing
  - Within and outside cluster

*Source: https://medium.com/@kubernetes-advocate*
K8s Native Services: ClusterIP

- Most common and default type
- Internal to the Cluster **only**
- Between backend/frontend pods

```yaml
apiVersion: v1
kind: Service
metadata:
  name: "nginx-service"
  namespace: "default"
spec:
  ports:
    - port: 80
  type: ClusterIP
  selector:
    app: "nginx"
```

Source: https://medium.com/@kubernetes-advocate
K8s Native Services: NodePort

- 30000-32767 ports range.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: "nginx-service"
  namespace: "default"
spec:
  ports:
   - port: 80
     nodePort: 30001
  type: NodePort
  selector:
    app: "nginx"
```

Source: https://medium.com/@kubernetes-advocate
K8s Native Services: Load Balancer

- Most popular Kubernetes Service
- Distributes network traffic among multiple Kubernetes Services.
  - Application Scalability
  - Containers are used more efficiently
  - Maximize the availability of your Services
- Elastic, Distributed and Easy to Orchestrate than physical/virtual appliances.
- In conjunction with an Ingress Controller. Can bring important benefits:
  - SSL Offload
  - App routing
GKE Load Balancer Example

apiVersion: apps/v1
kind: Deployment
metadata:
  name: ilb-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ilb-deployment
  template:
    metadata:
      labels:
        app: ilb-deployment
    spec:
      containers:
        - name: hello-app
          image: us-docker.pkg.dev/google-samples/containers/gke/hello-app:1.0
**externalTrafficPolicy**: Cluster or Local

- **externalTrafficPolicy** routes external traffic to node-local or cluster-wide endpoints.
- "Local" preserves the client source IP and avoids a second hop for LoadBalancer and NodePort type services.
- "Cluster" obscures the client source IP and may cause a second hop to another node.

**Diagram**:

- *Proxy*: iptables proxy rules for Service ExternalIPs or NodePort
K8s Services: LoadBalancer

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE</th>
<th>CLUSTER-IP</th>
<th>EXTERNAL-IP</th>
<th>PORT(S)</th>
<th>AGE</th>
</tr>
</thead>
</table>

[root@ctl-a1 ~]# kubectl describe svc hello-lb

Name:                     hello-lb
Namespace:                default
Labels:                   <none>
Annotations:              externalTrafficPolicy: local
Selector:                 app=hello
Type:                     LoadBalancer
IP Families:              <none>
IP:                       10.102.210.54
IPs:                      10.102.210.54
LoadBalancer Ingress:     10.254.254.240
Port:                     <unset> 8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset> 32142/TCP
Endpoints:                172.17.135.129:8080,172.17.135.130:8080,172.17.208.148:8080 + 3 more...
Session Affinity:          None
External Traffic Policy:  Cluster
Events:                   <none>
K8s Services: Ingress

Traffic

foo.mydomain.com
mydomain.com/bar
Other

Service
Pod Pod Pod

Service
Pod Pod Pod

Service
Pod Pod Pod

Kubernetes cluster

Source: https://medium.com/@kubernetes-advocate
K8s Services: Ingress

```yaml
apiVersion: v1
kind: Service
metadata:
  name: "nginx-1-service"
  namespace: "default"
spec:
  ports:
    - port: 80
      type: NodePort
      selector:
        app: "nginx-1"

---

apiVersion: v1
kind: Service
metadata:
  name: "nginx-2-service"
  namespace: "default"
spec:
  ports:
    - port: 80
      type: NodePort
      selector:
        app: "nginx-2"

---

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: "nginx-ingress"
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
  labels:
    app: "nginx"
spec:
  rules:
  - http:
      paths:
        - path: /svc1.html
          backend:
            serviceName: "nginx-1-service"
            servicePort: 80
        - path: /svc2.html
          backend:
            serviceName: "nginx-2-service"
            servicePort: 80
```

---

apiVersion: v1
kind: Service
metadata:
  name: "nginx-1-service"
  namespace: "default"
spec:
  ports:
    - port: 80
      type: NodePort
      selector:
        app: "nginx-1"

---

apiVersion: v1
kind: Service
metadata:
  name: "nginx-2-service"
  namespace: "default"
spec:
  ports:
    - port: 80
      type: NodePort
      selector:
        app: "nginx-2"

---

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: "nginx-ingress"
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
  labels:
    app: "nginx"
spec:
  rules:
  - http:
      paths:
        - path: /svc1.html
          backend:
            serviceName: "nginx-1-service"
            servicePort: 80
        - path: /svc2.html
          backend:
            serviceName: "nginx-2-service"
            servicePort: 80
Why should I care?

- Kubernetes does not offer, by default, an implementation of it for on-premises clusters
K8s Apps providing Load Balancing capabilities (on-premises)

**NGNIX Ingress Controller**
- SSL termination, load balancing, path-based routing, and traffic control
- Large and active community
- Extensive documentation

**HAProxy**
- Widely-used load balancer
- Robust performance and scalability

**Traefik**
- Automatic configuration, dynamic routing, SSL certificate management, and multiple service discovery mechanisms
- Ease of use

**MetalLB**
- **LoadBalancer service type**
- Simple and flexible
- Popular in on-premises and bare metal environments.
MetalLB: Layer2 Configuration

- In L2 mode, only one node is elected to announce the IP from.
- Normally, all the nodes where a Speaker is running are eligible for any given IP.
- There can be scenarios where only a subset of the nodes are exposed to a given network, so it can be useful to limit only those nodes as potential entry points for the service IP.
- This is achieved by using the node selector in the L2Advertisement CR.
BGP Mode K8s Load Balancer

- BGP-based leaf-switch implements stateless load balancing
  - Add Ingress for Stateful
- No Bottlenecks.
  - BGP brings distribution across the Network.
- Resilient
  - Fast failover
  - BFD support (Experimental and not included in this demo)
- Enables True Load Balancing via ECMP
- Traffic control: Cluster vs Local
Yep! Another element in the network

The moment you realize Kubernetes will be another element in the network (a BGP peer for the leaf switches)
Demo Setup
Clab: Bringing declarativeness to networking labs

name: mylab

topology:
  nodes:
    -
  links:
    -
Containerlab: Installation

Installation commands for Fedora33

```bash
# Install docker
sudo dnf -y install docker
sudo systemctl start docker
sudo systemctl enable docker

# Install containerlab
bash -c "$(curl -sL https://get.containerlab.dev)"
```

https://containerlab.dev/install/
Using git repo for this tutorial

https://github.com/cloud-native-everything/metallb-srl-nanog89/

```
[~]# git clone https://github.com/cloud-native-everything/metallb-srl-nanog89/
Cloning into 'pygnmi-srl-apps'...
remote: Enumerating objects: 251, done.
remote: Counting objects: 100% (251/251), done.
remote: Compressing objects: 100% (154/154), done.
remote: Total 251 (delta 54), reused 251 (delta 54), pack-reused 0
Receiving objects: 100% (251/251), 9.94 MiB | 4.74 MiB/s, done.
Resolving deltas: 100% (54/54), done.
```
Creating the lab

```
[pygnmi-srl-apps]# clab deploy -t topo.yml
INFO[0000] Containerlab v0.25.1 started
INFO[0000] Parsing & checking topology file: topo.yml
INFO[0000] Creating lab directory: /root/pygnmi-srl-apps/clab-dc-k8s
INFO[0000] Creating docker network: Name="kind", IPv4Subnet="172.18.100.0/16", IPv6Subnet="", MTU="1500"
INFO[0000] Creating container: "grafana"
INFO[0000] Creating container: "SPINE-DC-2"
INFO[0000] Creating container: "prometheus"
INFO[0000] Creating container: "LEAF-DC-1"
INFO[0000] Creating container: "BORDER-DC"
INFO[0000] Creating container: "SPINE-DC-1"
INFO[0000] Creating container: "LEAF-DC-2"
```
Inspecting the lab

```
[root@rbc-r2-hpe4 pygnmi-srl-apps]# clab inspect -t topo.yml
INFO[0000] Parsing & checking topology file: topo.yml

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Container ID</th>
<th>Image</th>
<th>Kind</th>
<th>IPv4 Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clab-dc-k8s-BORDER-DC</td>
<td>9876f09a5580</td>
<td>ghcr.io/nokia/srlinux:21.6.4</td>
<td>srl</td>
<td>172.18.100.125/16</td>
</tr>
<tr>
<td>2</td>
<td>clab-dc-k8s-LEAF-DC-1</td>
<td>830369bb4d39</td>
<td>ghcr.io/nokia/srlinux:21.6.4</td>
<td>srl</td>
<td>172.18.100.121/16</td>
</tr>
<tr>
<td>3</td>
<td>clab-dc-k8s-LEAF-DC-2</td>
<td>05d303e50816</td>
<td>ghcr.io/nokia/srlinux:21.6.4</td>
<td>srl</td>
<td>172.18.100.122/16</td>
</tr>
<tr>
<td>4</td>
<td>clab-dc-k8s-SPINE-DC-1</td>
<td>574ff19416fb</td>
<td>ghcr.io/nokia/srlinux:21.6.4</td>
<td>srl</td>
<td>172.18.100.123/16</td>
</tr>
<tr>
<td>5</td>
<td>clab-dc-k8s-SPINE-DC-2</td>
<td>e44d29973290</td>
<td>ghcr.io/nokia/srlinux:21.6.4</td>
<td>srl</td>
<td>172.18.100.124/16</td>
</tr>
<tr>
<td>6</td>
<td>clab-dc-k8s-grafana</td>
<td>e6d5221fa472</td>
<td>grafana/grafana:latest</td>
<td>linux</td>
<td>172.18.100.116/16</td>
</tr>
<tr>
<td>7</td>
<td>clab-dc-k8s-prometheus</td>
<td>533473420ff1</td>
<td>prom/prometheus:latest</td>
<td>linux</td>
<td>172.18.100.115/16</td>
</tr>
</tbody>
</table>
```
Kind Kubernetes

• Local Kubernetes clusters using Docker container “nodes”.
• Designed for testing Kubernetes.
• Requirements:
  • go (1.17+)
  • Uses docker for node instances

https://github.com/kubernetes-sigs/kind/
Install K8s Kind

• You can install kind with:
  
  `go install sigs.k8s.io/kind@v0.18.0`

• This will put kind in `$(go env GOPATH)/bin`.
  
  • You may need to add that directory to your `PATH` as shown here if you encounter the error `kind: command not found` after installation.

• Kind uses docker for node instances

• Once you have docker running you can create a cluster with:
  
  `kind create cluster`
Kind config

• For this demo we’ll use 2 workers and one controller
• Use “kind load” to upload images for apps and metallb after cluster is created
  • Unless you want to setup a private registry

```bash
[~/kind]# cat cluster_datacenter.yaml
---
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
  - role: worker
```
Connecting K8s Cluster to Clab

- You can use `clab tools veth`
- This containerlab tool helps you to connect containers after the lab is created like for example connecting leaf switches to K8s worker nodes.
We got your back

- We have included an app (go lang) that you can use to deploy the lab and the Kubernetes clusters
- Load images
- Connect elements
- Still under development
  - Contributions are welcome
We got your back

• You can use “srklab” to deploy everything
  ./srklab start -c srklab.yml
• All details of the configuration are in srklab.yml file

# this is the info clab tool will use to interconnect clusters and containerlab instances links:
  - k8sNode: "datacenter-worker:e1-1"
    clabNode: "clab-dc-k8s-LEAF-DC-1:e1-10"
    k8sIpv4: "192.168.101.101/24"
    k8sIpv4Gw: "192.168.101.1"
  - k8sNode: "datacenter-worker2:e1-1"
    clabNode: "clab-dc-k8s-LEAF-DC-2:e1-10"
    k8sIpv4: "192.168.101.102/24"
    k8sIpv4Gw: "192.168.101.1"
We got your back

- You can use "srklab" to deploy everything
  
  ./srklab start -c srklab.yml
- All details of the configuration are in srklab.yml file

```yaml
network: "kind"
prefix: "172.18.0.0/16"
clabTopology: "/root/srklab/topo.yml"
clusters:
  - name: "datacenter"
    kubeconfig: "/root/.kube/config-datacenter"
    config: "/root/srklab/kind/cluster_datacenter.yaml"
    image: "enc-kind-worker:v1.22.2"
    imagesToLoad:
      - image: alpine:latest
      - image: python:latest
      - image: quay.io/metallb/speaker:v0.12.1
      - image: quay.io/metallb/controller:v0.12.1
    resources:
      - app: "/root/srklab//metallb/metallb-namespacex.yaml"
      - app: "root/srklab//metallb/metallb-manifest.yaml"
      - app: "root/srklab//metallb/metallb-bgp-setup.yaml"
      - app: "root/srklab/app/hello-app-python-datacenter.yaml"
      - app: "/root/srklab/app/hello-app-lb-datacenter.yaml"
```
Starting the lab

- You can use “srklab” to deploy everything:
  
  ```bash
  ./srklab start -c srklab-nolb.yml
  ```

- Removing app load balancer setup from configuration file.
Lab Topology: eBGP Underlay Topology
Lab Topology: iBGP EVPN Overlay

AS65123

LEAF1

K8S WRK

LEAF2

K8S WRK

SPINE1

BORDER-LEAF

SPINE2

CLIENT
Pods and Containers

- Every Pod has a unique IP address
- IP address reachable from all other Pods in the K8s cluster

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: postgress
labels:
  app: postgress
spec:
  containers:
  - name: postgress
    image: postgress:9.6.17
    ports:
      - containerPort: 80
    env:
      - name: POSTGRESS_PASSWORD
        value: "pwd"
```
**ConfigMap**

- An API object
- Non-confidential data in key-value pairs
- Pods can consume ConfigMaps
  - Environment variables
  - Command-line arguments
  - Configuration files in a volume
MetalLB BGP setup

- Setting up ConfigMap
  - Currently using CRDs
- BGP Peers (EVPN Subnet)
- Local and peer ASN
- Address Pool
- Speakers

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    peers:
      - peer-address: 1.1.1.202
        peer-asn: 65302
        my-asn: 65201
        router-id: 6.5.2.2
        node-selectors:
          - match-expressions:
              - key: kubernetes.io/hostname
                operator: In
                values: [datacenter-worker2]
  address-pools:
    - name: default
      protocol: bgp
      addresses:
        - 10.254.254.240/28
```
MetalLB BGP setup

- Setting up ConfigMap
  - Currently using CRDs
- BGP Peers (EVPN Subnet)
- Local and peer ASN
- Address Pool
- Speakers
Python Simple HTTP Server

- Stateless Simple HTTP Python App
Python Simple HTTP Server

- Stateless Simple HTTP Python App

```python
import os
from http.server import HTTPServer, BaseHTTPRequestHandler
class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(bytes("Hello world\n", "utf8"))
port = int(os.environ['HTTP_APP_PORT'])
httpd = HTTPServer(('0.0.0.0', 8081), SimpleHTTPRequestHandler)
httpd.serve_forever()
```
Python Simple HTTP Server

- Stateless Simple HTTP Python App

[root@]# curl http://localhost:8081
Hello world
[root@]# OR

```
[root@]# curl http://localhost:8081
Hello world
[root@]#
```
Python Simple HTTP Server

- Stateless Simple HTTP Python App
- Test Load Balancer Services
- Exposes K8s Worker hostname and Pod Hostname

```python
apiVersion: v1
type: ConfigMap
metadata:
  name: py-tftpboot
data:
  http_server.py:
    import os
    from http.server import HTTPServer, BaseHTTPRequestHandler
class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
  def do_GET(self):
    self.send_response(200)
    self.send_header("Content-type", "text/html")
    self.end_headers()
    hostname = os.environ['HOSTNAME']
    worker = os.environ['K8S_NODE_NAME']
    text = f"K8s Node: {worker} - Hostname: {hostname}\n"
    self.wfile.write(bytes(text, "utf8"))
    port = int(os.environ['HTTP_APP_PORT'])
    httpd = HTTPServer(('0.0.0.0', port), SimpleHTTPRequestHandler)
    httpd.serve_forever()
```
**Hello Node App**

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-node-py-deploy
spec:
  selector:
    matchLabels:
      app: hello-node-python
  replicas: 4
  template:
    metadata:
      labels:
        app: hello-node-python
    spec:
      volumes:
        - name: tftpboot  # The name of the volume
      env:
        - name: K8S_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
      volumesMounts:
        - name: tftpboot
          mountPath: /tftpboot
  containers:
  - name: hello-node-python-app
    image: python:latest
    imagePullPolicy: Never
    command: ["python3"]
    args: ["/tftpboot/http_server.py"]
    ports:
    - containerPort: 8080
      protocol: TCP
```
Exposing Hello Node Replicas

- Exposing Deployment via Service Load Balancer
- External Traffic Policy: Local

```
apiVersion: v1
type: LoadBalancer
kind: Service
metadata:
  name: hello-lb
  annotations:
    externalTrafficPolicy: local
spec:
  ports:
    - port: 8080
      targetPort: 8080
  selector:
    app: hello-node-python
```
Simple Python HTTP Server

- Stateless Python App to test Load Balancer Services
- Exposes K8s Worker hostname and Pod Hostname
Lab Topology

K8S CTRL
- metallb ctrl

K8S WRK
- hello-node-py
  - metallb spkr

K8S WRK
- hello-node-py
  - metallb spkr

management (kind docker network)

HTTP TESTER
- ipvlan-pod
- ipvlan-pod
HTTP Tester Cassowary

$ ./cassowary run -u http://www.example.com -c 10 -n 100

Starting Load Test with 100 requests using 10 concurrent users

100% [████████████████████████████████████████] [1s:0s] 1.256773616s

TCP Connect.....................: Avg/mean=101.90ms Median=102.00ms p(95)=105ms
Server Processing...............: Avg/mean=100.18ms Median=100.50ms p(95)=103ms
Content Transfer...............: Avg/mean=0.01ms Median=0.00ms p(95)=0ms

Summary:
Total Req.........................: 100
Failed Req........................: 0
DNS Lookup........................: 115.00ms
Req/s............................: 79.57

https://github.com/rogerwel/in/cassowary
Kubernetes Deploy with IPVLAN

```yaml
spec:
  containers:
  - name: ipvlan-1001-alpine-1
    image: pinrojas/cassowary:0.33
    imagePullPolicy: Never
    command:
      - /bin/sh
      - -c
      - |
        until false; do cassowary run -u http://10.254.254.240:8080 -c 4 -n 4 -p pushsvc-cust1-to-dc:9091; sleep 2; done
    env:
      - name: K8S_NODE_NAME
        valueFrom:
          fieldRef:
            fieldPath: spec.nodeName
```
Load Balancing Hashing (Border Leaf)

- We defined Hashing “Source IP” in the Border Leaf for ECMP

```bash
A:BORDER-DC# info /system load-balancing
  system {
    load-balancing {
      hash-options {
        source-address true
      }
    }
  }
```

```bash
A:BORDER-DC#```
Load Balancing Hashing (Border Leaf)

- We defined Hashing “Source IP” in the Border Leaf for ECMP
- Add more replicas to the client app

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ipvlan-1001-deploy
spec:
  selector:
    matchLabels:
      app: ipvlan-1001-alpine
  replicas: 4
template:
  metadata:
    labels:
      app: ipvlan-1001-alpine
    annotations:
      k8s.v1.cni.cncf.io/networks: ipvlan-1001
  spec:
    containers:
      - name: ipvlan-1001-alpine-1
        image: pinrojas/cassowary:0.33
```
Lab Topology

- Ipvlan-pod
- Ipvlan-pod
- Ipvlan-pod
- Ipvlan-pod
- LEAF1
  - hello-node-py
  - hello-node-py
- LEAF2
  - hello-node-py
  - hello-node-py
- BODER-LEAF
  - Ipvlan-pod
  - Ipvlan-pod
  - Ipvlan-pod
Navigating the lab components
Testing Load Balancer
Testing Load Balancer

- We’ll show a test of the Load Balancer using curl command from the Kubernetes “client” connected to the border leaf
Load Balancing Hashing (Border Leaf)

- We defined Hashing “Source IP” in the Border Leaf (ECMP)
- Hashing algorithms are platform-specific and are considered proprietary
- In this demo we can **ADD** more replicas to the client app

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-68bjl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>47m</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-68fnk</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>47m</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-sxb6t</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>81m</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-tvjwb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>47m</td>
</tr>
<tr>
<td>pushgw-cust1-to-dc-6589cf6c-qjmsb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>81m</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-68bjl</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>46m</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-68fnk</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>46m</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-6dhf2</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m24s</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-7m8m5</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m24s</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-bzggj</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m24s</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-cqdm8</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m24s</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-n8mzc</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m24s</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-8q8q8x</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>8m24s</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-sxb6t</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>80m</td>
</tr>
<tr>
<td>ipvlan-1001-deploy-65fb9bcb8-tvjwb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>46m</td>
</tr>
<tr>
<td>pushgw-cust1-to-dc-6589cf6c-qjmsb</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>80m</td>
</tr>
</tbody>
</table>
Load Balancing Hashing (Border Leaf)

- Grafana shows how we can control forwarding distribution from the Border Leaf
- Stats are collected via GNMIc
- Prometheus pulls data directly from GNMIc server
Demo: Testing Load Balancer
Tips: Forcing multiple routes

- Reject any exchange of EVPN router type 5 between all LEAF switches and BORDER Leaf
Final Words

In this tutorial we have seen:

• How to create a network lab using containerlab and Kind Kubernetes
• How to set up and test a BGP Load Balancer Services Between your Fabric and Kubernetes
• How we can control load distribution via Load Balancing Hashing Algorithm at the Border Leaf
Additional resources

• Getting Started with Modern Time Series Database and Grafana – Damien Garros
  • [https://youtu.be/lzppzWGRHGo](https://youtu.be/lzppzWGRHGo)
• Containerlab - running networking labs with Docker UX – Roman and Karim
  • [https://youtu.be/qigC1a1qY3k](https://youtu.be/qigC1a1qY3k)
• gNMIc - an intuitive gNMI CLI and a feature-rich telemetry collector - Karim
  • [https://youtu.be/v3CL2vrGD_8](https://youtu.be/v3CL2vrGD_8)
• Tutorial: Kubernetes 101 for Network Professionals
  • [https://youtu.be/n2kgApcXij0](https://youtu.be/n2kgApcXij0)
Thanks!

bio.site/pinrojas

DONE WITH MY PRESENTATION

NOW I HAVE TO ANSWER QUESTIONS
Additional Slides
MetalLB

- MetalLB aims to redress this imbalance by offering a network load balancer implementation that integrates with standard network equipment.
- MetalLB implements a FRR Mode that uses an FRR container as the backend for handling BGP sessions. It provides features that are not available with the native BGP implementation, such as pairing BGP sessions with BFD sessions, and advertising IPV6 addresses.
- Despite being less battle tested than the native BGP implementation, the FRR mode is currently used by those users that require either BFD or IPV6, and it is the only supported method in the MetalLB version distributed with OpenShift. The long term plan is to make it the only BGP implementation available in MetalLB.