KUB 09: Kubernetes Networking

Описание:

CNI

В других заданиях мы с вами изучили основные компоненты, которые запускаются на мастер хостах, а так же на рабочих нодах. Но присутствует один тонкий момент, который хочется обсудить отдельно - сети.
Давайте пройдем от общего к частному и поговорим для начала какую задачу должна решать сеть в Kubernetes. Для нее существует несколько требований:

- Все поды должны иметь уникальные IP адреса
- Все поды должны иметь возможность обращаться друг к другу по этим IP адресам без использования NAT

То есть верхнеуровнево схема выглядит вот таким образом:

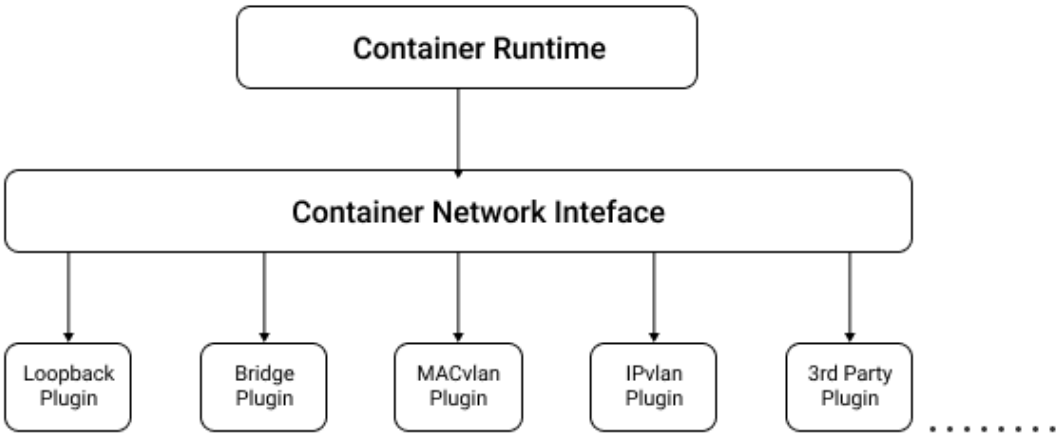
- Kubelet получает от API задачу на запуск пода
- Kubelet отдает команду на запуск настроенному Runtime (docker, cri-o, containerd, ...)
- Runtime стартует настройку сети для контейнера - выделяет IP адрес, создает интерфейс и подключает его к контейнеру

После этой операции наш котейнер запущен и готов общаться со всеми соседями в кластере так, как если бы они находились в одной сети. То есть, если мы представим, что в нашем кластере Kubernetes существует два узла - hostA и hostB и на них запущены поды podA (10.0.0.3) и podB (10.0.1.5), то в этом случае podA должен иметь возможность спокойно обращаться к podB по его IP адресу.
Чтобы данная схема заработала нам необходимо решить три задачи:

1. IPAM - IP Address Management - нам нужно как-то научиться выдавать IP адреса нашим подам так, чтобы они были уникальны в рамках всего кластера и не пересекались
2. Network Management - нам нужно создавать и подключать сетевой интерфейс к контейнеру таким образом, чтобы контейнер после старта имел возможность отправлять и принимать данные с других подов.
3. Routing - нам нужно позаботиться о том, чтобы поды могли видеть друг друга без использования NAT - то есть пакеты, отправленные от пода на одном хосте могли спокойно дойти до пода на втором узле без трансляции адресов.

Для решения этих задач в Kubernetes был внедрен CNI - Container Network Interface. Изначально команда разработки выбирала между CNM (Container Network Model) от Docker и CNI от RKT и в итоге остановила свой выбор на CNI. Подробнее о том, почему так произошло можно почитать здесь, но в двух словах стоит сказать, что CNM от Docker был более сложным в реализации и требовал дополнительных внешних зависимостей для своей работы - таких как key value storage для работы с группой машин. В целом это

не соответствовало идеалогии UNIX - делать одно дело, но хорошо. Поэтому и был принят вариант с CNI.



Итак, что же такое CNI? Container Network Interface - это спецификация, говорящая о том как container runtime должен вызывать скрипты, чтобы настроить сеть в контейнере. Посмотреть ее можно здесь. А теперь простыми словами. Когда runtime (docker, cri-o, containerd и другие) запускают контейнер им требуется как-то настроить сеть внутри этого контейнера. И если раньше это делал сам runtime - то есть находил свободные адреса, создавал интерфейсы и подключал их в контейнер, то в этом случае это делают CNI плагины, которые runtime должен запустить и указать для какого контейнера они должны настроить сеть. А внутри плагинов может происходить что угодно - они могут просто выдать статический адрес или же выбрать свободный IP из всего кластера, используя внешнее хранилище. Так же они могут создать новый бридж и подключить в него контейнер или же создать какой-то vxlan интерфейс, который позволит пакетам ходить между несколькими хостами.
Давайте посмотрим на простой пример. Если мы зайдем на любую ноду нашего кластера, который был настроен kube spray'ем, то мы сможем найти там CNI плагины, которые находятся в /opt/cni/bin:

```
root@node2:~# ls -la /opt/cni/bin/
total 181368
drwxr-xr-x 2 root root     4096 Dec  9 17:49 .
drwxr-xr-x 3 kube root     4096 Mar 30 10:35 ..
-rwxr-xr-x 1 root root  4151672 Dec  9 17:48 bandwidth
-rwxr-xr-x 1 root root  4527563 Dec  9 17:48 bridge
-rwxr-xr-x 1 kube root 37261312 Mar 30 10:52 calico
-rwxr-xr-x 1 kube root 37261312 Mar 30 10:52 calico-ipam
-rwxr-xr-x 1 root root 10261898 Dec  9 17:48 dhcp
... skipped ...
```

Это простые бинарные файлы, которые нужно запустить, чтобы они настроили наш контейнер. Давайте теперь перейдем в директорию /etc/cni/net.d/ и создадим простую конфигурацию 10-local.

```
{
  "cniVersion": "0.4.0",
  "name": "localnet",
  "type": "bridge",
  "bridge": "lcl0",
  "ipam": {
    "type": "host-local",
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1"
  },
  "dns": {
    "nameservers": [ "10.1.0.1" ]
  }
}
```

В таком файле мы в первую очередь указываем версия CNI. В зависимости от версии доступны различные команды и параметры настроек, которые можно посмотреть в спецификации. После этого мы указываем плагин, который необходимо использовать при настройке сети - bridge, а так же специфичные параметры для этого плагина - bridge: lcl0. На самом деле тип плагина - это название бинарного файла, который лежит в /opt/cni/bin. Помимо этого в разделе ipam указывается какой плагин будет отвечать за назначание IP адресов подам. В нашем случае - это host-local, который так же находится в той же директории /opt/cni/bin.
Итак, cni мы настроили. Теперь давайте сделаем отдельный сетевой namespace, как это бы сделал runtime:

```
CID=34254425667788686 78
ip netns add $CID
```

В первой строке мы придумали уникальный идентификатор нашего namespace'a, а второй командой создали его. Теперь мы можем посмотреть какие интерфейсы есть внутри нашего пространства имен:

```
# ip netns exec $CID ifconfig
```

Замечательно! Давайте теперь поработаем за рантайм и запустим плагины, которые мы прописали в конфигурации cni. На самом деле runtime должен сам пробежаться и посмотреть все конфигурационные файлы, которые находятся в /etc/cni/net.d/ и последовательно запустить все плагины, которые там описаны. Мы написали только один конфиг, которые вызывает bridge плагин, поэтому запустим только его:

```
# Указываем команду на добавление интерфейса для плагина
```

```
export CNI_COMMAND=ADD
# Указываем путь до плагинов CNI
export CNI_PATH=/opt/cni/bin
# Добавляем директорию с плагинами в область поиска команд
export PATH=/opt/cni/bin:$PATH
# Указываем ID нашего контейнера
export CNI_CONTAINERID=$CID
# Указываем путь к нашему сетевому namespace'у
export CNI_NETNS=/run/netns/34254425667788686 78
# Указываем название интерфейса, который добавим внутрь
контейнера
export CNI_IFNAME=eth10
# Запускаем плагин
bridge </etc/cni/net.d/10-local.conf
{
    "cniVersion": "0.4.0",
    "interfaces": [
        {
            "name": "lcl0",
            "mac": "6e:3e:a8:58:69:d9"
        },
        {
            "name": "vethbc8ef1e1",
            "mac": "6e:3e:a8:58:69:d9"
        },
        {
            "name": "eth10",
            "mac": "06:f6:4e:56:7d:fa",
            "sandbox": "/run/netns/34254425667788686 78"
        }
    ],
    "ips": [
        {
            "version": "4",
            "interface": 2,
            "address": "10.1.0.2/16",
            "gateway": "10.1.0.1"
        }
    ],
    "dns": {
        "nameservers": [
```

```
            "10.1.0.1"
        ]
    }
}
```

Как видно - в ответ плагин вывел нам всю информацию о том как он подключил наш сетевой интерфейс в контейнер. Он создал бридж lcl0, интерфейс eth10 и подключил интерфейс eth10 к lcl0 бриджу. Помимо этого он назначил ip адрес 10.1.0.2 нашему интефрейсу. Давайте посмотрим на ifconfig внутри контейнера:

```
# ip netns exec 3425442566778868678 ifconfig
eth10: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.1.0.2  netmask 255.255.0.0  broadcast
10.1.255.255
        inet6 fe80::4f6:4eff:fe56:7dfa  prefixlen 64  scopeid
0x20<link>
        ether 06:f6:4e:56:7d:fa  txqueuelen 0  (Ethernet)
        RX packets 12  bytes 1136 (1.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11  bytes 838 (838.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions
0
```

Как видно CNI успешно настроил наш интерфейс внутри контейнера. Итак, давайте резюмировать то как работает CNI:

1. Kubelet отправляет настроенному runtime команду на запуск пода
2. Runtime ищет все доступные настройки cni в /etc/cni/net.d и поочередно выполняет их:

- Для каждой настройки Runtime запускает network плагин с нужными параметрами (это выглядит как выставление env переменных и запуск бинарника плагина из /opt/cni/bin)
- Сетевой Плагин внутри вызывает ipam плагин для назначения ip адреса новому поду
- Сетевой Плагин, получив ip адрес, создает интерфейс и подключает его к сетевому namespace контейнера и производит его настройку

3. При удалении контейнера runtime так же вызывает все плагины для настройки сети, но с командой DEL.

В примере выше мы с вами использовали самые простые плагины bridge и host-local, которые просто подключают контейнер к бриджу и выдают адрес из заданного пула. Заметьте, кстати, что данный вариант настройки предполагает, что маршрутизация настроена вручную - поды просто отправляют пакеты друг другу и они должны маршрутизироваться между узлами.

Если брать другие сетевые плагины, то они могут производить более тонкую настройку - например, создавать vxlan интерфейс, который будет пересылать пакеты от подов на соседние узлы, чтобы они могли достичь цели. Давайте посмотрим на конфигурацию calico:

```
{
  "name": "cni0",
  "cniVersion":"0.3.1",
  "plugins":[
    {
      "datastore_type": "kubernetes",
      "nodename": "node2",
      "type": "calico",
      "log_level": "info",
      "log_file_path": "/var/log/calico/cni/cni.log",
      "ipam": {
        "type": "calico-ipam",
        "assign_ipv4": "true",
        "ipv4_pools": ["10.233.64.0/18"]
      },
      "policy": {
        "type": "k8s"
      },
      "kubernetes": {
        "kubeconfig": "/etc/cni/net.d/calico-kubeconfig"
      }
    },
    {
      "type":"portmap",
      "capabilities": {
        "portMappings": true
      }
    }
  ]
}
```

В данном случае используется плагин calico, а так же указываются его специфичные настройки. Обратите внимание, что так же используется calico-ipam - плагин, который будет выделять IP адреса подам.
Мы очень советуем посмотреть на сравнение популярных плагинов для настройки сети в Kubernetes. В реальной жизни вы скорее всего будете использовать либо простой вариант с бриджом как описано выше и настраивать маршрутизацию на внутреннем роутере или же поставите calico и она все сделает за вас. Внутри калико использует etcd для хранения

своей информации - например о выданных ip адресах. Помимо этого калико использует bgp для управления таблицей маршрутизации, чтобы поды на разных хостах могли видеть друг друга. BGP демоны запускаются на каждом узле в кластере. Подробнее о калико можно почитать в полезных ссылках.

DNS

Помимо настройки сети хотелось бы рассказать про DNS сервер, встроенный в Kubernetes. Он нужен для более простого обнаружения сервисов и подов, чтобы вам не приходилось использовать IP адреса напрямую. Изначально использовался kube-dns, но с версии 1.11 его заменили на coredns. Это было связано с обнаруженными проблемами работы и безопасности.
DNS-сервис представляет собой несколько pods в namespace kube-system. Контроллеры в подах следят за изменением состояния объектов и автоматически формируют записи DNS. При запуске pod kubelet автоматически подменяет файл /etc/resolv.conf в контейнере. В этом файле добавляется search для более удобного обнаружения.
A-запись формируется по маске service.namespace.svc.cluster.local, где cluster.local — домен кластера.
В рамках одного namespace можно обращаться к сервису и подам просто по имени, без суффикса. Обратиться в другому namespace можно, добавив .namespace к имени.

Kube-dns

Использует SkyDNS+dnsmasq+sidecar.

- SkyDNS— разрешает запросы DNS,
- dnsmasq — служит как кеш,
- sidecar — контейнер формирует отчеты и выполняет проверки жизнеспособности сервиса.

Из-за проблем масштабирования skyDNS и уязвимостей в dnsmasq заменен на CoreDNS.

CoreDNS

Это отдельный сервис, написанный на GO. Поддерживает плагины, внешние резолверы, распределение нагрузки, autopath для сокращения времени ответа. Также автоматически преобразует 10.32.0.125.namespace.pod.cluster.local в 10.32.0.125, даже если этот адрес не назначен никакому сервису или поду. Сравнение и историю перехода можно увидеть по ссылке [coredns-ga-for-kubernetes-cluster-dns] Дополнительные конфигурационные параметры для pod:
DNSPolicy может быть нескольких видов. А именно — Default, CLusterFirst, ClusterFirstWithHostNet. Как работают, понятно из названия. Указываются в spec-разделе.

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox:1.28
  command:
    - sleep
    - "3600"
  imagePullPolicy: IfNotPresent
  name: busybox
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
    # Можно более точно управлять dns для pod через
spec.dnsConfig
  dnsConfig:
    nameservers:
    - 1.2.3.4
    searches:
    - ns1.svc.cluster-domain.example
    - my.dns.search.suffix
    options:
    - name: ndots
      value: "2"
    - name: edns0
```

CoreDNS AutoScaling

Помимо этого в namespace kube-system запущен специальный сервис — dns-autoscaler, который будет увеличивать количество coredns инстансов, чтобы они могли успешно справляться с нагрузкой в кластере. Ведь действительно при запуске 10 подов нагрузка на dns одна, а при запуске 1000 подов — уже совсем другая. Все настройки задаются в configmap этого сервиса, который вы сможете изменить (чуть позднее мы изучим, как работать с configmap). Но сейчас для упрощения мы можем поменять конфигурацию, используя kubectl и ваш любимый редактор, например так:

```
$ kubectl -n kube-system edit configmap dns-autoscaler
```

И дальше отредактировать параметры запуска autoscaler:

```
# Please edit the object below. Lines beginning with a '#' will
be ignored,
```

```
# and an empty file will abort the edit. If an error occurs
while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  linear:
'{"coresPerReplica":256,"min":2,"nodesPerReplica":16,"preventSin
glePointFailure":true}'
kind: ConfigMap
metadata:
  creationTimestamp: "2021-03-30T10:53:58Z"
  name: dns-autoscaler
  namespace: kube-system
  resourceVersion: "1162"
  uid: 4cbdcbff-b55d-4595-a058-5364d071d0c3
```

Основная настройка здесь — это linear — параметр масштабирования coredns инстансов. В примере будет запущен coredns на каждые 256 ядер CPU, доступных в кластере, или на каждые 16 нод, добавленных в кластер, — в зависимости от того, что случится ранее. Более подробное описание параметров можно найти здесь.

Node local DNS Cache

В любом случае каждый раз отправлять запрос на coredns для определения имени может быть достаточно затратно. Ведь coredns запускается не на всех узлах кластера, а только на некоторых, и это может привести к увеличению latency — каждому поду придется делать запрос, который пойдет по сети до coredns, и ждать ответа. Для уменьшения latency в Kubernetes есть возможность запустить локальный кеширующий dns-демон, который будет доступен на каждой ноде. Таким образом, для разрешения имени в большинстве случаев не придется использовать сеть за пределами конкретной ноды — запросы будут обработаны локально.
Для его настройки достаточно выполнить несколько простых действий (Kubespray сделал все необходимые действия за нас, поэтому мы приводим этот текст на будущее для примера):

1. Скачиваем манифест для запуска node local dns кеширующих демонов:

```
$ curl -s -o nodelocaldns.yaml
https://raw.githubusercontent.com/kubernetes/kubernetes/master/c
luster/addons/dns/nodelocaldns/nodelocaldns.yaml >
nodelocaldns.yaml
```

2. Получаем наши данные — адрес coredns сервиса, а также выбираем локальный адрес, на котором будет слушать наш кеширующий node local dns, и домен нашего кластера — cluster.local:

```
$ export kubedns=$(kubectl get svc kube-dns -n kube-system -o
jsonpath={.spec.clusterIP})
$ export domain=cluster.local
$ export localdns=169.254.20.53
$
```

3. Редактируем nodelocadns.yaml на основе полученных данных:

```
$ sed -i '' "s/__PILLAR__LOCAL__DNS__/$localdns/g;
s/__PILLAR__DNS__DOMAIN__/$domain/g;
s/__PILLAR__DNS__SERVER__/$kubedns/g;" nodelocaldns.yaml
```

4. Применяем полученный манифест:

```
$ kubectl create -f nodelocaldns.yaml
serviceaccount/node-local-dns created
service/kube-dns-upstream created
configmap/node-local-dns created
daemonset.apps/node-local-dns created
service/node-local-dns created
```

5. Убедимся, что запустились все сервисы:

```
$ kubectl -n kube-system get pods | grep local-dns
node-local-dns-phs7f                    1/1     Running   0
22s
node-local-dns-r24pb                    1/1     Running   0
22s
$
```

После этого мы можем запустить pod с ubuntu, установить туда dnsutils и проверить работу dns:

```
$ kubectl run ubuntu --image ubuntu -- /bin/bash -c 'while true;
do sleep 300; done'
$ kubectl exec -it ubuntu -- bash
root@ubuntu:/# apt update

.. skipped ...
root@ubuntu:/# apt install -y dnsutils

... skipped ...
```

```
root@ubuntu:/# dig google.com @169.254.20.53

; <<>> DiG 9.16.1-Ubuntu <<>> google.com @169.254.20.53
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24722
;; flags: qr ra; QUERY: 1, ANSWER: 6, AUTHORITY: 0, ADDITIONAL:
1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: 28428a998f550729 (echoed)
;; QUESTION SECTION:
;google.com.                    IN    A

;; ANSWER SECTION:
google.com.             30    IN    A     64.233.162.100
google.com.             30    IN    A     64.233.162.102
google.com.             30    IN    A     64.233.162.113
google.com.             30    IN    A     64.233.162.101
google.com.             30    IN    A     64.233.162.138
google.com.             30    IN    A     64.233.162.139

;; Query time: 4 msec
;; SERVER: 169.254.20.53#53(169.254.20.53)
;; WHEN: Tue Dec 15 12:46:32 MSK 2020
;; MSG SIZE  rcvd: 207

root@ubuntu:/#
```

Как вы можете увидеть — наш адрес 169.254.20.53 успешно работает и отдает нам dns-имена. Он также будет работать прозрачно для всех подов — благодаря iptables, перенаправляющим все dns-запросы на coredns на локальный local dns, который мы настроили. Кстати, node-local-dns также отдает метрики в формате prometheus:

```
root@ubuntu:/# curl -s 169.254.20.53:9253/metrics | grep
coredns_forward_requests_total
# HELP coredns_forward_requests_total Counter of requests made
per upstream.
# TYPE coredns_forward_requests_total counter
coredns_forward_requests_total{to="10.0.0.2:53"} 16
```

```
coredns_forward_requests_total{to="10.96.187.100:53"} 71
```

Так что можно подключить его к кластеру для мониторинга.

Полезные ссылки:

- Flannel Networking
- CNI Plugins Overview
- DNS for Services and Pods
- Customizing DNS Service
- Cluster Networking (official docs)
- Container Network Interface — networking for Linux containers (github)
- Introducing Linux Network Namespaces
- DNS Autoscaling
- Node Local DNS

Задание:

1. Разверните кластер kubernetes на трех нодах, используя конфиг kubespray, подготовленный в 3ем задании:
- В control plane должны находиться ноды node1 и node2
- Etcd должен быть запущен на нодах node1, node2, node3
- В качестве network plugin выберите flannel
2. Создайте два пода с image: alpine и именем alpine-1 и alpine-2 в namespace default таким образом, чтобы они были постоянно запущены.
3. Получите IP-адреса каждого пода и сохраните их в файл /ip.txt внутри соответствующего пода (IP alpine-1 должен быть сохранен в /ip.txt внутри пода alpine-1. Та же схема и для alpine-2)
4. Зайдите в под alpine-1 и попробуйте выполнить ping второго пода alpine (alpine-2).
5. Отправьте задание на проверку.