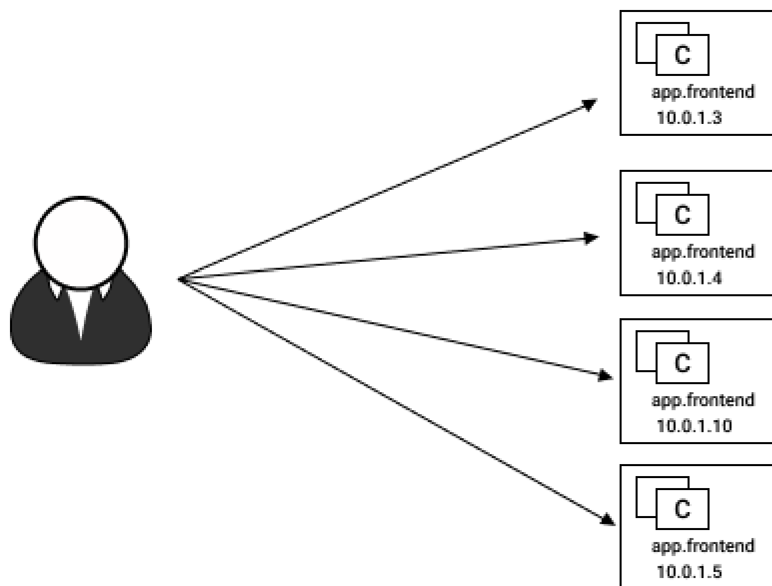
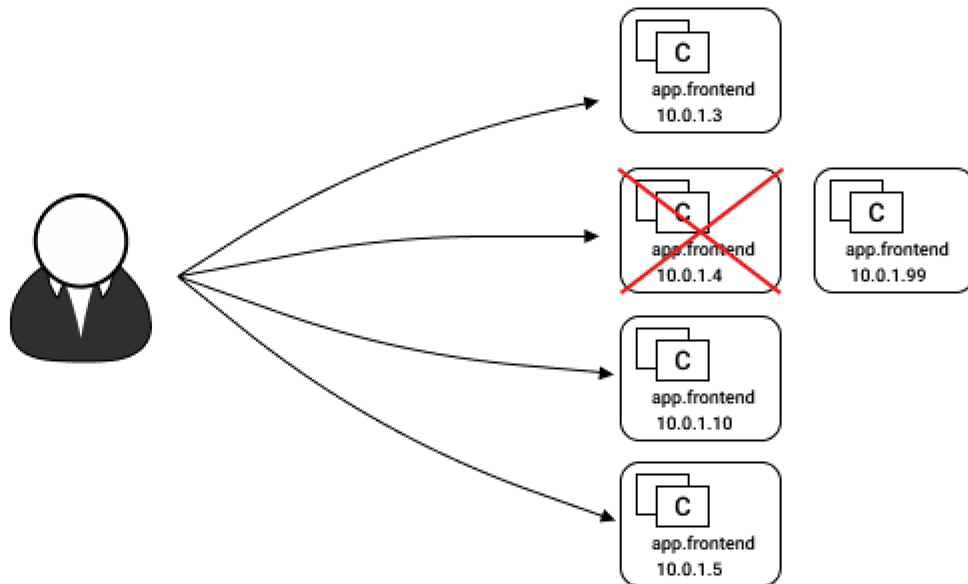
KUB 13: Ресурсы Service в Kubernetes

Описание:

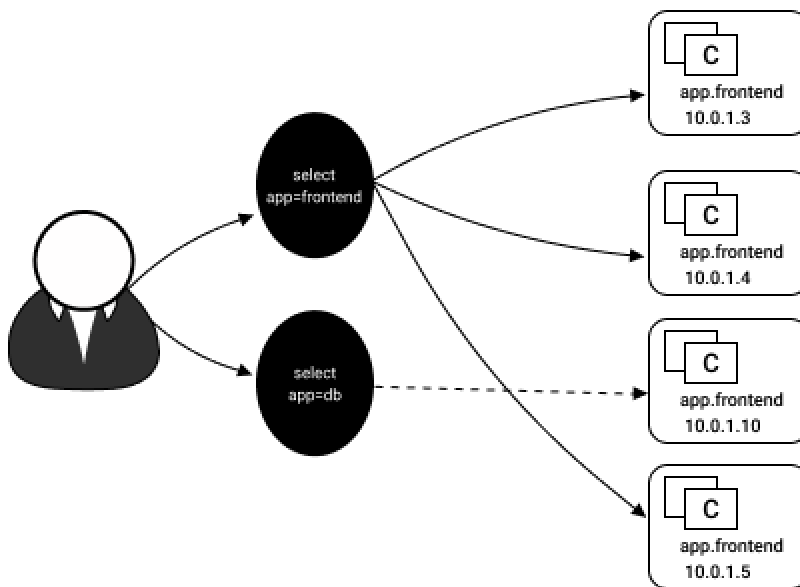
До текущего момента мы не получали доступ к приложению внутри Pod. Пора это поправить и рассказать про ресурс service. Этот ресурс разрешит другим Pod или внешним пользователям получить доступ к приложению внутри пода. Представьте, что вы знаете IP каждого Pod и делаете curl http://$POD_IP.



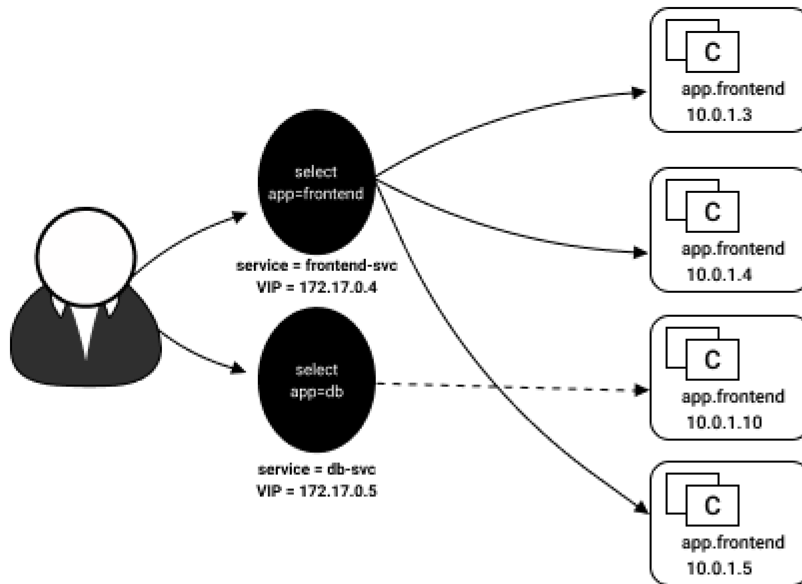
Все замечательно до момента падения одного из Pod - его тут же пересоздает ReplicaSet. Появляется новый Pod с новым IP. Опять нужно узнавать адрес Pod для получения к нему доступа.



Service позволяет обойти эту проблему. Сервис постоянно мониторит все поды, которые имеют правильные labels и автоматически конфигурирует правила балансировки в iptables / ipvs с помощью kube-proxy. Таким образом, если под пересоздался - правила обновятся автоматически и трафик пойдет на новый под.



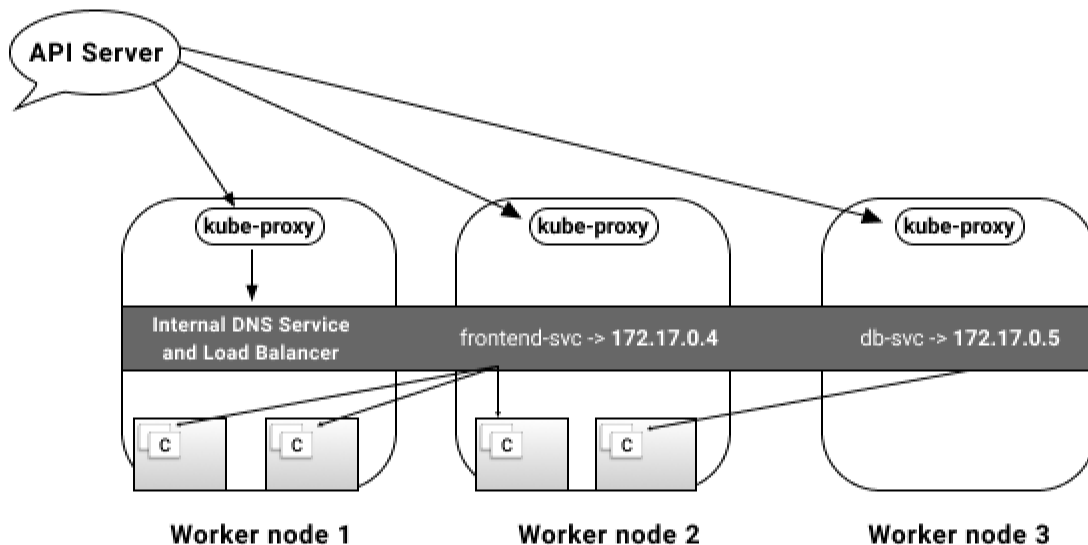
Это работает как простой round-robin балансировщик. Каждому объекту service выделяется свой IP - сервисный ip адрес.



Для создания сервиса хватит просто указать метки и порт нужных Pod. Пример:

```
kind: Service
apiVersion: v1
metadata:
  name: frontend-svc
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

После применения этого YAML будет создан объект service, но, кроме этого, kube-proxy также добавит на всех нодах правила в iptables для нового service. DNS-служба внутри кластера создаст имя для этого service.



Также после создания service в ENV контейнеров будут автоматически добавлены переменные с адресом service. Например, сервис redis-master добавит переменные:

```
REDIS_MASTER_SERVICE_HOST=172.17.0.6
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://172.17.0.6:6379
REDIS_MASTER_PORT_6379_TCP=tcp://172.17.0.6:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=172.17.0.6
```

Приложение способно использовать переменные окружения для обнаружения redis-master. DNS-имя для сервиса формируется по маске my-svc.my-namespace.svc.cluster.local, где

- my-svc — имя сервиса (metadata.name);
- my-namespace — namespace, в котором создан сервис (metadata.namespace);
- svc - сокращение от service - в этом домене указаны все сервисы во всех неймспейсах
- cluster.local — dns-суффикс кластера. Указывается при создании кластера.

ServiceType

При создании сервиса можно выбрать область доступности:

- доступен только внутри кластера — ClusterIP;
- доступен внутри кластера и для внешних клиентов — NodePort;
- предоставляет доступ к ресурсам вне кластера — ExternalService;

- балансировщик нагрузки — LoadBalancer, обычно необходима поддержка от облачного провайдера;
- внешний адрес — ExternalIP, требуется поддержка окружением/облаком.

Давайте разберем подробнее каждый из перечисленных вариантов.

ClusterIP

Данный тип используется по умолчанию при создании сервиса. Создается DNS-запись для сервиса и выделяется виртуальный IP.

Давайте попробуем создать простой deployment для nginx и сервис для него:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: nginx
  template:
    metadata:
      labels:
        app.kubernetes.io/name: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
```

svc.yaml:

```yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx-svc
spec:
 type: ClusterIP
 selector:
   app.kubernetes.io/name: nginx
 ports:
 - port: 80
   targetPort: 80
```

После применения манифестов можем проверить сервис:

```
$ kubectl get svc
```

```
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
kubernetes    ClusterIP   10.96.128.1     <none>        443/TCP
44h
nginx-svc     ClusterIP   10.96.218.218   <none>        80/TCP
2m3s
$ kubectl describe svc nginx-svc
Name:                   nginx-svc
Namespace:              default
Labels:                 <none>
Annotations:            Selector:  app.kubernetes.io/name=nginx
Type:                   ClusterIP
IP:                     10.96.218.218
Port:                   <unset>  80/TCP
TargetPort:             80/TCP
Endpoints:              10.112.128.10:80
Session Affinity:       None
Events:                 <none>
$
```

Как вы можете увидеть, сервис получил внутренний IP 10.96.218.218 и у него есть один endpoint — 10.112.128.10 — это адрес нашего пода, который был создан deployment'ом. Давайте попробуем увеличить количество реплик у деплоймента и посмотрим, как отреагирует на это сервис:

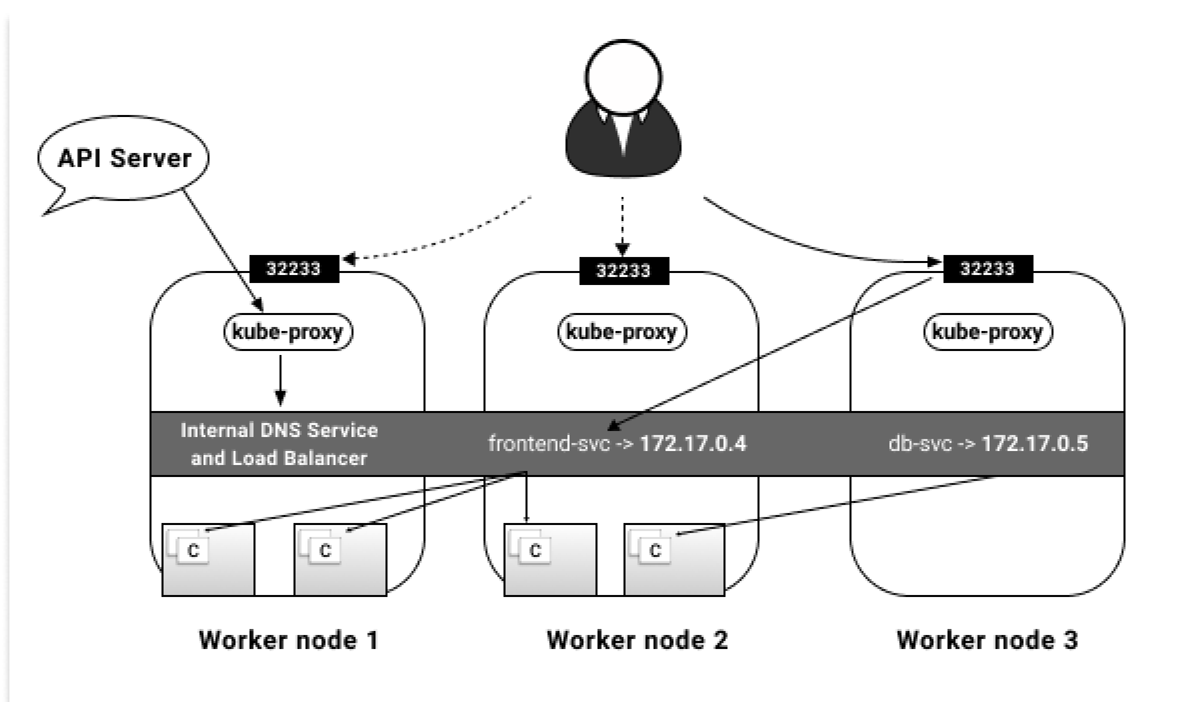
```
$ kubectl scale deploy nginx --replicas=3
deployment.apps/nginx scaled
$ kubectl describe svc nginx-svc
Name:                   nginx-svc
Namespace:              default
Labels:                 <none>
Annotations:            Selector:  app.kubernetes.io/name=nginx
Type:                   ClusterIP
IP:                     10.96.218.218
Port:                   <unset>  80/TCP
TargetPort:             80/TCP
Endpoints:
10.112.128.10:80,10.112.130.4:80,10.112.130.5:80
Session Affinity:       None
Events:                 <none>
```

Теперь в нашем сервисе уже три endpoints — то есть, три наших пода с nginx. Давайте попробуем запустить контейнер с alpine и обратиться к нашему сервису:

```
$ kubectl run -it alpine --image alpine -- sh
If you don't see a command prompt, try pressing enter.
/ # apk add curl
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.12/main/x86_64/APKINDEX.
tar.gz
fetch
http://dl-cdn.alpinelinux.org/alpine/v3.12/community/x86_64/APKI
NDEX.tar.gz
(1/4) Installing ca-certificates (20191127-r4)
(2/4) Installing nghttp2-libs (1.41.0-r0)
(3/4) Installing libcurl (7.69.1-r1)
(4/4) Installing curl (7.69.1-r1)
Executing busybox-1.31.1-r19.trigger
Executing ca-certificates-20191127-r4.trigger
OK: 7 MiB in 18 packages
/ # curl -v http://nginx-svc/
*   Trying 10.96.218.218:80...
* Connected to nginx-svc (10.96.218.218) port 80 (#0)
> GET / HTTP/1.1
> Host: nginx-svc
> User-Agent: curl/7.69.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Server: nginx/1.19.5
< Date: Fri, 04 Dec 2020 08:58:19 GMT
< Content-Type: text/html
< Content-Length: 612
< Last-Modified: Tue, 24 Nov 2020 13:02:03 GMT
< Connection: keep-alive
< ETag: "5fbd044b-264"
< Accept-Ranges: bytes
<
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

```
<style>
    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>
* Connection #0 to host nginx-svc left intact
```

Мы создали под alpine, установили туда curl и сделали запрос к нашему сервису nginx-svc.
Как видно, имя преобразовалось в адрес 10.96.218.218 и мы получили ответ от nginx.
NodePort
Похож на ClusterIP, но дополнительно выставляет открытый порт на каждом узле кластера. По умолчанию, порт назначается рандомно из диапазона 30000-32767. После выделения порта kube-proxy прописывает правила в iptables, так что неважно, на какую ноду пришел пакет, он все равно дойдет до пода, даже если тот на другой ноде.



Давайте попробуем поменять наш сервис из предыдущего примера и создать svc NodePort:

```
apiVersion: v1
kind: Service
metadata:
 name: nginx-svc
spec:
 type: NodePort
 selector:
   app.kubernetes.io/name: nginx
 ports:
 - port: 80
   targetPort: 80
```

Проверим сервис:

```
$ kubectl get svc
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP    PORT(S)
AGE
kubernetes    ClusterIP   10.96.128.1     <none>         443/TCP
44h
nginx-svc     NodePort    10.96.218.218   <none>
80:30166/TCP   11m
```

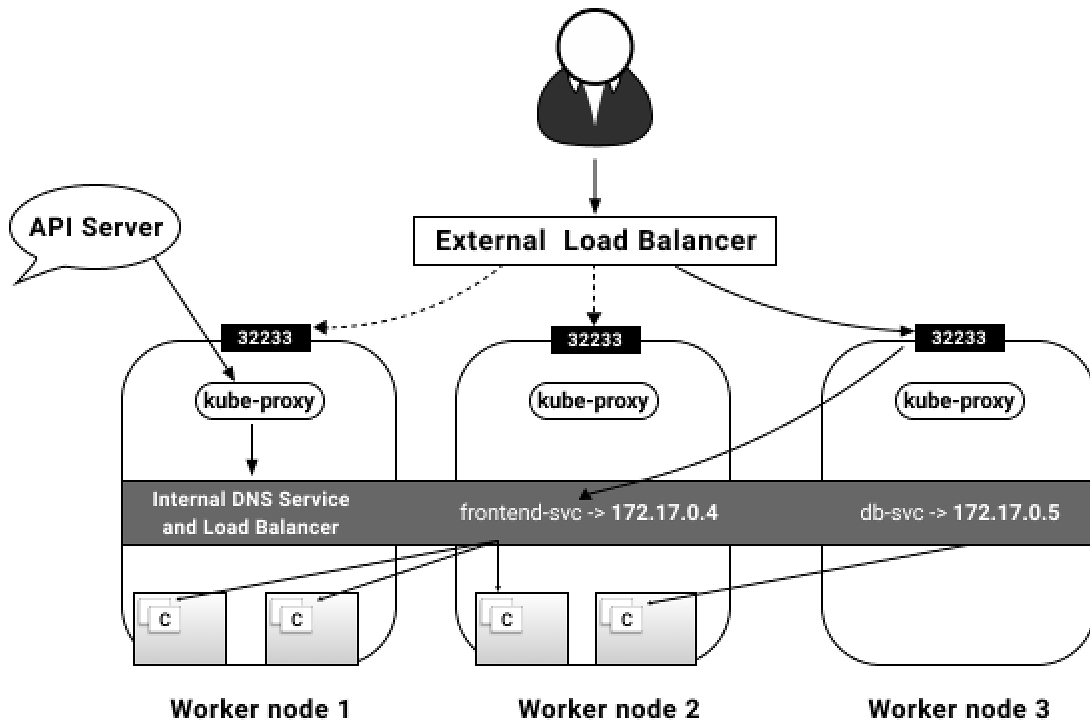
Как можно заметить, на всех нодах открылся порт 30166, к которому мы можем обратиться извне:

```
$ kubectl get nodes -o wide
NAME                            STATUS    ROLES     AGE    VERSION
INTERNAL-IP    EXTERNAL-IP      OS-IMAGE
KERNEL-VERSION     CONTAINER-RUNTIME
cl128p75cket5bspv91v-oles   Ready     <none>    44h    v1.18.9
10.1.0.26     84.201.137.28    Ubuntu 18.04.4 LTS
5.4.0-52-generic   docker://19.3.13
cl128p75cket5bspv91v-oxuq   Ready     <none>    44h    v1.18.9
10.0.0.23     130.193.50.214   Ubuntu 18.04.4 LTS
5.4.0-52-generic   docker://19.3.13
cl1ihia34v9pbjll4oic-yhes   Ready     <none>    33h    v1.18.9
10.0.0.6      130.193.39.145   Ubuntu 18.04.4 LTS
5.4.0-52-generic   docker://19.3.13

$ curl -D - -s -o /dev/null http://130.193.39.145:30166/
HTTP/1.1 200 OK
Server: nginx/1.19.5
Date: Fri, 04 Dec 2020 09:00:44 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 24 Nov 2020 13:02:03 GMT
Connection: keep-alive
ETag: "5fbd044b-264"
Accept-Ranges: bytes

$
```

Мы взяли внешний адрес нашей ноды, обратились по выделенному порту и получили ответ от nginx.

LoadBalancer

Данный тип сервиса зависит от поддержки окружения - точнее от cloud controller manager, который мы упоминали в самом начале. Например, ELB на AWS или Load Balancer в Yandex Cloud. Как и nodeport, выставляет порты на каждом узле кластера, но дополнительно запускает внешний балансировщик нагрузки с помощью cloud controller manager'a.



Давайте изменим наш сервис с nodeport на loadbalancer:

```
apiVersion: v1
kind: Service
metadata:
 name: nginx-svc
spec:
 type: LoadBalancer
 selector:
   app.kubernetes.io/name: nginx
 ports:
 - port: 80
   targetPort: 80
```

В данном случае после создания сервиса cloud controller manager начнет создавать балансировщик нагрузки в облаке, и, как только он создастся, мы увидим внешний IP у сервиса:

```
$ kubectl get svc
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)        AGE
kubernetes   ClusterIP      10.96.128.1     <none>
443/TCP        44h
```

```
nginx-svc    LoadBalancer   10.96.218.218   84.201.135.235
80:30166/TCP   14m
```

Теперь мы можем обратиться по этому IP и попасть в наши поды:

```
$ curl -D - -s -o /dev/null http://84.201.135.235/
HTTP/1.1 200 OK
Server: nginx/1.19.5
Date: Fri, 04 Dec 2020 09:03:35 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 24 Nov 2020 13:02:03 GMT
Connection: keep-alive
ETag: "5fbd044b-264"
Accept-Ranges: bytes
```

Но в локальных окружениях - то есть с self hosted версией Kubernetes вы без дополнительных телодвижений не сможете использовать тип load balancer, поскольку у вас некому выдавать внешние айпи адреса и настраивать маршрутизацию. Для решения этой задачи существует проект - [MetalLB](). По сути он добавляет в kubernetes controller, который отвечает за некоторые ресурсы - в том числа и за сервисы с типом LoadBalancer. Когда такой сервис будет создан, metallb controller увидит это и выделит ему адрес из преднастроенного блока адресов.
У metallb есть два режима работы - либо на втором уровне (layer 2) либо на третьем - layer 3.
На втором уровне metallb очень похож на keepalived. Когда metallb выдает сервису адрес из доступного пула адресов, он так же настраивает данные адрес на интерфейсе хост системы - таким образом пользователи смогут обратить к данному адресу из локальной сети. При этом если нода, на которой настроен данный адрес упадет - metallb перенесет ее на другую ноду и она начнет принимать входящий трафик - совсем как keepalived.
На третьем уровне metallb использует bgp для анонса выделенных адресов - таким образом ваши роутеры или роутеры провайдера смогут узнать про ваш новый ip адрес.
Мы же посмотрим на metallb на примере layer 2. Итак, для начала нам нужно установить его в систему. Сделать это можно с помощью установки нужных нам манифестов, как описано на [странице metallb]():

```
# kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.9.6/manifes
ts/namespace.yaml
namespace/metallb-system created
# kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.9.6/manifes
ts/metallb.yaml
podsecuritypolicy.policy/controller created
```

```
podsecuritypolicy.policy/speaker created
serviceaccount/controller created
serviceaccount/speaker created
clusterrole.rbac.authorization.k8s.io/metallb-system:controller
created
clusterrole.rbac.authorization.k8s.io/metallb-system:speaker
created
role.rbac.authorization.k8s.io/config-watcher created
role.rbac.authorization.k8s.io/pod-lister created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:cont
roller created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:spea
ker created
rolebinding.rbac.authorization.k8s.io/config-watcher created
rolebinding.rbac.authorization.k8s.io/pod-lister created
daemonset.apps/speaker created
deployment.apps/controller created
# kubectl create secret generic -n metallb-system memberlist
--from-literal=secretkey="$(openssl rand -base64 128)"
secret/memberlist created
```

Так же metallb просит выставить параметр strictARP: true в настройках kube-proxy. Сделать это можно, отредактировав configmap kube-proxy:

```
kubectl edit configmap -n kube-system kube-proxy
```

И изменим параметр:

```
strictARP: true
```

После чего можно сохранить изменения и выйти из редактора.
Теперь нам осталось задать конфигурацию - какие айпи адреса будет выдавать metallb:

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
```

```
    - 192.168.1.240-192.168.1.250
```

Применим данную конфигурацию:

```
# kubectl -n metallb-system apply -f cm.yaml
configmap/config created
```

И попробуем создать наш первый локальный сервис из примера выше:

```
# cat svc.yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx-svc
spec:
 type: LoadBalancer
 selector:
   app.kubernetes.io/name: nginx
 ports:
 - port: 80
   targetPort: 80
# kubectl apply -f svc.yaml
service/nginx-svc created
```

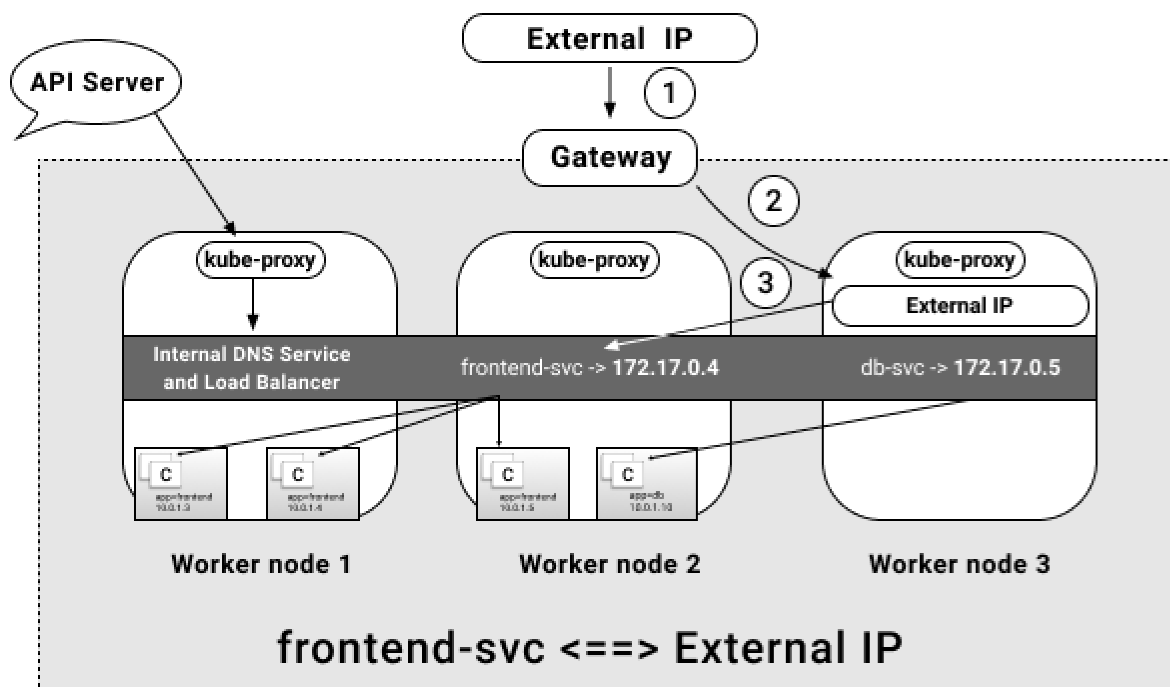
Давайте проверим что все прошло удачно:

```
root@node1:~# kubectl get svc
NAME          TYPE           CLUSTER-IP     EXTERNAL-IP    PORT(S)
AGE
kubernetes    ClusterIP      10.96.0.1      <none>         443/TCP
4m52s
nginx-svc     LoadBalancer   10.99.241.58   10.114.15.1
80:30335/TCP   18s
root@node1:~#
```

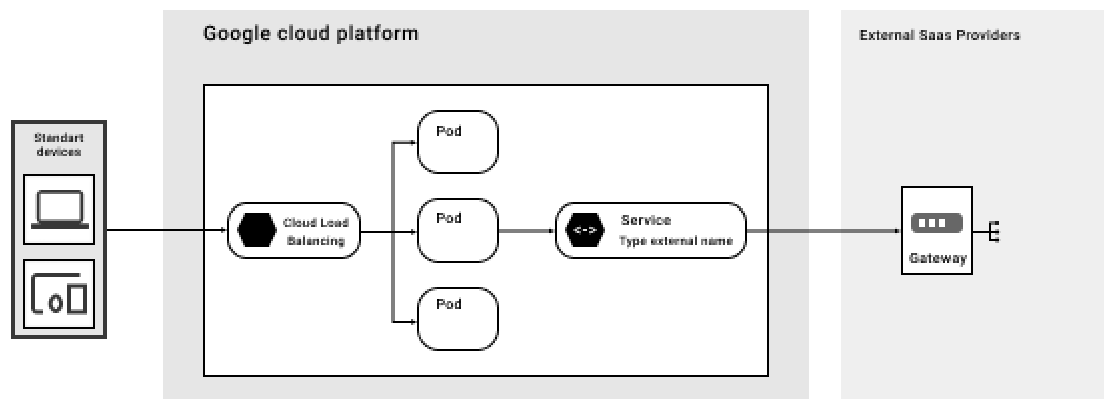
Как видно - мы получили ip адрес из заданой нами сети. Поскольку мы находимся в виртуальном окружении (в облаке digitalocean), нам нужно добавить этот адрес в маршрутизацию сети на digitalocean, поскольку адресация там работает хитрым образом не смотря на то, что ip находится в нашей локальной сети. А вот такая же конфигурация в своей локалке приведет к тому, что вы сможете спокойно обратиться к айпи адресу 10.114.15.1.

ExternalIP

Позволяет получить доступ к сервису с внешнего IP-адреса. То есть привязывает внешний IP-адрес на определенный сервис, почти как 1-to-1 NAT. Требуется поддержка окружением/облаком.



ExternalService
Есть несколько типов внешних сервисов. ExternalName связывает сервис с внешним DNS CNAME именем. Также возможно задать внешние IP-адреса в качестве endpoint сервиса.



Каждый service содержит несколько адресов, на которые он балансирует запросы. Эти адреса называются endpoints. В большинстве случаев kubelet автоматически составляет список адресов на основе живых pod. Для externalservice адреса задаются статически.
Пример:

```
kind: Service
apiVersion: v1
metadata:
 name: postgresql
spec:
```

```yaml
 type: ClusterIP
 ports:
 - port: 5432
   targetPort: 5432
---
kind: Endpoints
apiVersion: v1
metadata:
 name: postgresql
subsets:
 - addresses:
     - ip: 10.73.10.105
     - ip: 10.73.10.106
   ports:
     - port: 5432
```

Обратите внимание, что в спецификации сервиса мы не указываем selector, то есть сервис не сможет найти поды, которые надо привязать к этому сервису. Но мы делаем это вручную, создавая ресурс EndPoint. Если pod будет обращаться к адресу postgresql, то kubelet автоматически направит трафик на указанные endpoints. Для pod это прозрачно, нет нужды записывать IP-адреса в конфигурацию.

Headless Service

Обычно каждому сервису выделяется виртуальный IP-адрес, при обращении к нему kube-proxy направит запрос на один из подов. Однако при создании сервиса можно указать None в поле clusterIP. В таком случае сервис не получит адреса, но по-прежнему получит список endpoints из подов. Такие сервисы используются для обнаружения всех подов в состоянии ready без использования виртуального адреса.
Также можно добавить аннотацию tolerate-unready-endpoints при создании сервиса и получить в списке endpoints все поды, даже не прошедшие readinesscheck.

External Traffic Policy

В случаях использования NodePort или LoadBalancer есть два варианта, как Kubernetes будет проксировать трафик к поду. Представьте, что кластер состоит из 5 нод и вы запустили 1 под на первой ноде с веб-сервером nginx.
Когда вы создаете сервис NodePort / LoadBalancer с ExternalTrafficPolicy Cluster, то при подключении к любой ноде кластера Kubernetes (а точнее iptables) пробросит трафик на под, запущенный на первой ноде. Таким образом, например, четвертая нода, по сути, станет балансировщиком, который пропустит трафик через себя до пода на первой ноде.
Если же вы выставите ExternalTrafficPolicy Local, то Kubernetes создаст правила проксирования трафика в iptables только на нодах, где запущены нужные поды — в нашем

случае только на первой ноде. При обращении к остальным нодам мы не сможем подключиться к поду.
Очень хорошо и с картинками данный параметр описан в статье здесь. Обращайте внимание на данный параметр для более точной настройки балансировки вашего трафика.

Полезные ссылки:

- Service (official docs)
- Overview of a Service
- External Traffic Policy

Задание:

1. Установите metallb в namespace metallb-system
2. Задайте пул адресов для metallb - 10.114.15.0/24
3. Создайте деплоймент httpd-dp в namespace default с образом httpd.
4. Создайте сервис http-svc-int в namespace default с типом ClusterIP, который будет смотреть на поды из третьего пункта.
5. Создайте сервис http-svc-ext в namespace default с типом LoadBalancer, который будет смотреть на поды из третьего пункта.
6. Отправьте задание на проверку.