

首发于
Kubernetes

IPVS从入门到精通kube-proxy实现原理

**int32bit**

Linux、云计算、OpenStack，欢迎关注个人微信公众号：int32bit

关注他

59 人赞同了该文章

1 kube-proxy介绍

1.1 为什么需要kube-proxy

我们知道容器的特点是快速创建、快速销毁，Kubernetes Pod和容器一样只具有临时的生命周期，一个Pod随时有可能被终止或者漂移，随着集群的状态变化而变化，一旦Pod变化，则该Pod提供的服务也就无法访问，如果直接访问Pod则无法实现服务的连续性和高可用性，因此显然不能使用Pod地址作为服务暴露端口。

▲ 赞同 59



● 7 条评论



分享



喜欢



收藏



申请转载





Kubernetes Service。

这个Service就是由kube-proxy实现的，ClusterIP不会因为Pod状态改变而变，需要注意的是VIP即ClusterIP是个假的IP，这个IP在整个集群中根本不存在，当然也就无法通过IP协议栈无法路由，底层underlay设备更无法感知这个IP的存在，因此ClusterIP只能是单主机(Host Only)作用域可见，这个IP在其他节点以及集群外均无法访问。

Kubernetes为了实现在集群所有的节点都能够访问Service，kube-proxy默认会在所有的Node节点都创建这个VIP并且实现负载，所以在部署Kubernetes后发现kube-proxy是一个DaemonSet。

而Service负载之所以能够在Node节点上实现是因为无论Kubernetes使用哪个网络模型，均需要保证满足如下三个条件：

1. 容器之间要求不需要任何NAT能直接通信；
2. 容器与Node之间要求不需要任何NAT能直接通信；
3. 容器看到自身的IP和外面看到它的IP必须是一样的，即不存在IP转化的问题。

至少第2点是必须满足的，有了如上几个假设，Kubernetes Service才能在Node上实现，否则Node不通Pod IP也就实现不了了。

有人说既然kube-proxy是四层负载均衡，那kube-proxy应该可以使用haproxy、nginx等作为负载后端啊？

事实上确实没有问题，不过唯一需要考虑的就是性能问题，如上这些负载均衡功能都强大，但毕竟还是基于用户态转发或者反向代理实现的，性能必然不如在内核态直接转发处理好。

因此kube-proxy默认会优先选择基于内核态的负载作为后端实现机制，目前kube-proxy默认是通过iptables实现负载的，在此之前还有一种称为userspace模式，其实也是基于iptables实现，可以认为当前的iptables模式是对之前userspace模式的优化。

本节接下来将详细介绍kube-proxy iptables模式的实现原理。

1.2 kube-proxy iptables模式实现原理

1.2.1 ClusterIP

首先创建了一个ClusterIP类型的Service:

▲ 赞同 59

▼ 7 条评论

分享

喜欢

收藏

申请转载

...



kubernetes-bootcamp-v1	ClusterIP	10.106.224.41	<none>	8080/TCP	163m
------------------------	-----------	---------------	--------	----------	------

其中ClusterIP为10.106.224.41，我们可以验证这个IP在本地是不存在的：

```
root@ip-192-168-193-172:~# ping -c 2 -w 2 10.106.224.41
PING 10.106.224.41 (10.106.224.41) 56(84) bytes of data.

--- 10.106.224.41 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1025ms

root@ip-192-168-193-172:~# ip a | grep 10.106.224.41
root@ip-192-168-193-172:~#
```

所以不要尝试去ping ClusterIP，它不可能通的。

此时在Node节点192.168.193.172上访问该Service服务，首先流量到达的是OUTPUT链，这里我们只关心nat表的OUTPUT链：

```
# iptables-save -t nat | grep -- '-A OUTPUT'
-A OUTPUT -m comment --comment "kubernetes service portals" -j KUBE-SERVICES
```

该链跳转到 KUBE-SERVICES 子链中：

```
# iptables-save -t nat | grep -- '-A KUBE-SERVICES'
...
-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.106.224.41/32 -p tcp -m comment --comment "d
-A KUBE-SERVICES -d 10.106.224.41/32 -p tcp -m comment --comment "default/kubernetes-b
```

我们发现与之相关的有两条规则：

- 第一条负责打标记MARK 0x4000/0x4000，后面会用到这个标记。
- 第二条规则跳到 KUBE-SVC-RPP7DHNMGOIIFDC 子链。

其中 KUBE-SVC-RPP7DHNMGOIIFDC 子链规则如下：

```
# intables-save -t nat | grep -- '-A KUBE-SVC-RPP7DHNMGOIIFDC'
```

[▲ 赞同 59](#) [▼](#) [7 条评论](#) [分享](#) [喜欢](#) [收藏](#) [申请转载](#) ...

这几条规则看起来复杂，其实实现的功能很简单：

- 1/3的概率跳到子链 KUBE-SEP-FTIQ6MSD3LW05HZX ,
- 剩下概率的1/2, $(1 - 1/3) * 1/2 == 1/3$, 即1/3的概率跳到子链 KUBE-SEP-SQBK6CVV7ZCKBTVI ,
- 剩下1/3的概率跳到 KUBE-SEP-IAZPHGLZV02SWOVD 。

我们查看其中一个子链 KUBE-SEP-FTIQ6MSD3LW05HZX 规则：

```
# iptables-save -t nat | grep -- '-A KUBE-SEP-FTIQ6MSD3LW05HZX'
...
-A KUBE-SEP-FTIQ6MSD3LW05HZX -p tcp -m tcp -j DNAT --to-destination 10.244.1.2:8080
```

可见这条规则的目的是做了一次DNAT，DNAT目标为其中一个Endpoint，即Pod服务。

由此可见子链 KUBE-SVC-RPP7DHNMGOIIFDC 的功能就是按照概率均等的原则DNAT到其中一个Endpoint IP，即Pod IP，假设为10.244.1.2，

此时相当于：

```
192.168.193.172:xxxx -> 10.106.224.41:8080
|
|
|   DNAT
|
V
192.168.193.172:xxxx -> 10.244.1.2:8080
```

接着来到POSTROUTING链：

```
# iptables-save -t nat | grep -- '-A POSTROUTING'
-A POSTROUTING -m comment --comment "kubernetes postrouting rules" -j KUBE-POSTROUTING
# iptables-save -t nat | grep -- '-A KUBE-POSTROUTING'
-A KUBE-POSTROUTING -m comment --comment "kubernetes service traffic requiring SNAT" -
```

这两条规则只做一件事就是只要标记了 0x4000/0x4000 的包就一律做MASQUERADE (SNAT)，



```

    | DNAT
    v
192.168.193.172:xxxx -> 10.244.1.2:8080
    |
    | SNAT
    v
10.244.0.0:xxxx -> 10.244.1.2:8080

```

剩下的就是常规的走Vxlan隧道转发流程了，这里不再赘述，感兴趣的可以参考我之前的文章[浅聊几种主流Docker网络的实现原理](#)。

1.2.2 NodePort

接下来研究下NodePort过程，首先创建如下Service:

```

# kubectl get svc -l owner=int32bit
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
kubernetes-bootcamp-v1   NodePort   10.106.224.41   <none>        8080:30419/TCP   3h3

```

其中Service的NodePort端口为30419。

假设有一个外部IP 192.168.193.197，通过 192.168.193.172:30419 访问服务。

首先到达PREROUTING链:

```

# iptables-save -t nat | grep -- '-A PREROUTING'
-A PREROUTING -m comment --comment "kubernetes service portals" -j KUBE-SERVICES
# iptables-save -t nat | grep -- '-A KUBE-SERVICES'
...
-A KUBE-SERVICES -m addrtype --dst-type LOCAL -j KUBE-NODEPORTS

```

PREROUTING的规则非常简单，凡是发给自己的包，则交给子链 KUBE-NODEPORTS 处理。注意前面省略了判断ClusterIP的部分规则。

KUBE-NODEPORTS 规则如下:

▲ 赞同 59

▼ 7 条评论

分享

喜欢

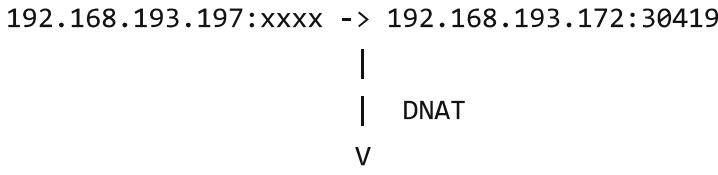
收藏

申请转载

...



这个规则首先给包打上标记 `0x4000/0x4000`，然后交给子链 `KUBE-SVC-RPP7DHNHMG0IIFDC` 处理，`KUBE-SVC-RPP7DHNHMG0IIFDC` 刚刚已经见面过了，其功能就是按照概率均等的原则DNAT到其中一个Endpoint IP，即Pod IP，假设为`10.244.1.2`。



此时发现`10.244.1.2`不是自己的IP，于是经过路由判断目标为`10.244.1.2`需要从flannel.1发出去。

接着到了 FORWARD 链，

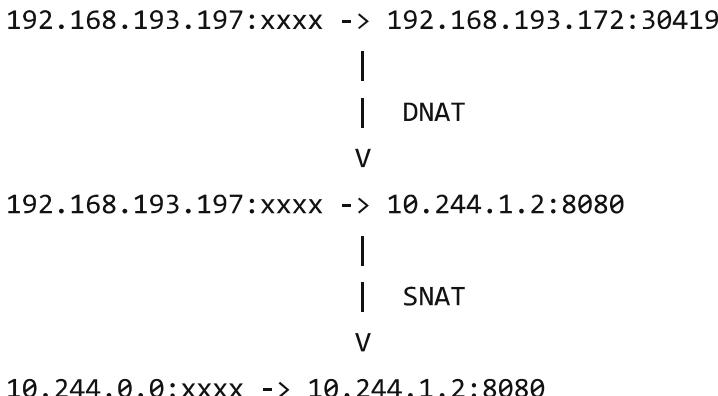
```

# iptables-save -t filter | grep -- '-A FORWARD'
-A FORWARD -m comment --comment "kubernetes forwarding rules" -j KUBE-FORWARD
# iptables-save -t filter | grep -- '-A KUBE-FORWARD'
-A KUBE-FORWARD -m conntrack --ctstate INVALID -j DROP
-A KUBE-FORWARD -m comment --comment "kubernetes forwarding rules" -m mark --mark 0x40

```

FORWARD表在这里只是判断下，只允许打了标记 `0x4000/0x4000` 的包才允许转发。

最后来到 POSTROUTING 链，这里和ClusterIP就完全一样了，在 `KUBE-POSTROUTING` 中做一次 MASQUERADE (SNAT)，最后结果：



▲ 赞同 59

▼ 7 条评论

分享

喜欢

收藏

申请转载

...

首发于
Kubernetes

- iptables规则复杂零乱，真要出现什么问题，排查iptables规则必然得掉层皮。LOG + TRACE 大法也不好使。
- iptables规则多了之后性能下降，这是因为iptables规则是基于链表实现，查找复杂度为O(n)，当规模非常大时，查找和处理的开销就特别大。据官方说法，当节点到达5000个时，假设有2000个NodePort Service，每个Service有10个Pod，那么在每个Node节点中至少有20000条规则，内核根本支撑不住，iptables将成为最主要的性能瓶颈。
- iptables主要是专门用来做主机防火墙的，而不是专长做负载均衡的。虽然通过iptables的statistic 模块以及DNAT能够实现最简单的只支持概率轮询的负载均衡，但是往往我们还需要更多更灵活的算法，比如基于最少连接算法、源地址HASH算法等。而同样基于netfilter的ipvs却是专门做负载均衡的，配置简单，基于散列查找O(1)复杂度性能好，支持数十种调度算法。因此显然ipvs比iptables更适合做kube-proxy的后端，毕竟专业的人做专业的事，物尽其美。

本文接下来将介绍kube-proxy的ipvs实现，由于本人之前也是对ipvs很陌生，没有用过，专门学习了下ipvs，因此在第二章简易介绍了下ipvs，如果已经很熟悉ipvs了，可以直接跳过，这一章和Kubernetes几乎没有任何关系。

另外由于本人对ipvs也是初学，水平有限，难免出错，欢迎指正！

2 IPVS 简易入门

2.1 IPVS简介

我们接触比较多的是应用层负载均衡，比如haproxy、nginx、F5等，这些负载均衡工作在用户态，因此会有对应的进程和监听socket，一般能同时支持4层负载和7层负载，使用起来也比较方便。

LVS是国内章文嵩博士开发并贡献给社区的（章文嵩博士和他背后的负载均衡帝国），主要由ipvs和ipvsadm组成，ipvs是工作在内核态的4层负载均衡，和iptables一样都是基于内核底层netfilter实现，netfilter主要通过各个链的钩子实现包处理和转发。ipvsadm和ipvs的关系，就好比netfilter和iptables的关系，它运行在用户态，提供简单的CLI接口进行ipvs配置。

由于ipvs工作在内核态，直接基于内核处理包转发，所以最大的特点就是性能非常好。又由于它工作在4层，因此不会处理应用层数据，经常有人问ipvs能不能做SSL证书卸载、或者修改HTTP头部数据，显然这些都不可能做的。

我们知道应用层负载均衡大多数都是基于反向代理实现负载的，工作在应用层，当用户的包到达负

▲ 赞同 59



● 7 条评论



● 喜欢

● 收藏

● 申请转载

...



重DNAT，按照一定的算法把ip包的目标地址DNAT到其中真实服务的目标IP。针对如上两种情况分别对应ipvs的两种模式—网关模式和NAT模式，另外ipip模式则是对网关模式的扩展，本文下面会针对这几种模式的实现原理进行详细介绍。

2.2 IPVS用法

ipvsadm命令行用法和iptables命令行用法非常相似，毕竟是兄弟，比如 -L 列举， -A 添加， -D 删除。

```
ipvsadm -A -t 192.168.193.172:32016 -s rr
```

但是其实ipvsadm相对iptables命令简直太简单了，因为没有像iptables那样存在各种table，table嵌套各种链，链里串着一堆规则，ipvsadm就只有两个核心实体，分别为service和server，service就是一个负载均衡实例，而server就是后端member,ipvs术语中叫做real server，简称RS。

如下命令创建一个service实例 172.17.0.1:32016， -t 指定监听的为 TCP 端口， -s 指定算法为轮询算法rr(Round Robin)，ipvs支持简单轮询(rr)、加权轮询(wrr)、最少连接(lc)、源地址或者目标地址散列(sh、dh)等10种调度算法。

```
ipvsadm -A -t 172.17.0.1:32016 -s rr
```

然后把10.244.1.2:8080、10.244.1.3:8080、10.244.3.2:8080添加到service后端member中。

```
ipvsadm -a -t 172.17.0.1:32016 -r 10.244.1.2:8080 -m -w 1
ipvsadm -a -t 172.17.0.1:32016 -r 10.244.1.3:8080 -m -w 1
ipvsadm -a -t 172.17.0.1:32016 -r 10.244.3.2:8080 -m -w 1
```

其中 -t 指定service实例， -r 指定server地址， -w 指定权值， -m 即前面说的转发模式，其中 -m 表示为 masquerading，即NAT模式， -g 为 gatewaying，即直连路由模式， -i 为 ipip，即 IPIP隧道模式。

与iptables-save、iptables-restore对应的工具ipvs也有ipvsadm-save、ipvsadm-restore。

2.3 NAT(network access translation)模式

▲ 赞同 59

▼

7 条评论

分享

喜欢

收藏

申请转载

...



现环境中LB节点IP为192.168.193.197，三个RS节点如下：

- 192.168.193.172:30620
- 192.168.193.194:30620
- 192.168.193.226:30620

为了模拟LB节点IP和RS不在同一个网络的情况，在LB节点中添加一个虚拟IP地址：

```
ip addr add 10.222.0.1/24 dev ens5
```

创建负载均衡Service并把RS添加到Service中：

```
ipvsadm -A -t 10.222.0.1:8080 -s rr
ipvsadm -a -t 10.222.0.1:8080 -r 192.168.193.194:30620 -m
ipvsadm -a -t 10.222.0.1:8080 -r 192.168.193.226:30620 -m
ipvsadm -a -t 10.222.0.1:8080 -r 192.168.193.172:30620 -m
```

这里需要注意的是，和应用层负载均衡如haproxy、nginx不一样的是，haproxy、nginx进程是运行在用户态，因此会创建socket，本地会监听端口，而ipvs的负载是直接运行在内核态的，因此不会出现监听端口：

```
68-193-197:/var/log# netstat -lnpt
net connections (only servers)
Send-Q Local Address          Foreign Address        State      PID/Program name
  0 127.0.0.53:53            0.0.0.0:*
                           LISTEN     674/systemd-resolve
  0 0.0.0.0:22              0.0.0.0:*
                           LISTEN     950/sshd
  0 :::22                   :::*                  LISTEN     950/sshd
```

可见并没有监听10.222.0.1:8080 Socket。

Client节点IP为192.168.193.226，为了和LB节点的虚拟IP 10.222.0.1通，我们手动添加静态路由如下：

```
ip r add 10.222.0.1 via 192.168.193.197 dev ens5
```



```
64 bytes from 10.222.0.1: icmp_seq=1 ttl=64 time=0.345 ms
64 bytes from 10.222.0.1: icmp_seq=2 ttl=64 time=0.249 ms
```

```
--- 10.222.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1022ms
rtt min/avg/max/mdev = 0.249/0.297/0.345/0.048 ms
```

可见Client节点到VIP的链路没有问题，那是否能够访问我们的Service呢？

我们验证下：

```
root@ip-192-168-193-226:~# curl -m 2 --retry 1 -sSL 10.222.0.1:8080
curl: (28) Connection timed out after 2001 milliseconds
```

非常意外的结果是并不通。

在RS节点抓包如下：

我们发现数据包的源IP为Client IP，目标IP为RS IP，换句话说，LB节点IPVS只做了DNAT，把目标IP改成RS IP了，而没有修改源IP。此时虽然RS和Client在同一个子网，链路连通性没有问题，但是由于Client节点发出去的包的目标IP和收到的包源IP不一致，因此会被直接丢弃，相当于给张三发信，李四回的信，显然不受信任。

既然IPVS没有给我们做SNAT，那自然想到的是我们手动做SNAT，在LB节点添加如下iptables规则：

▲ 赞同 59 ▾ 7 条评论 分享 喜欢 收藏 申请转载 ...



再次检查Service是否可以访问:

```
root@ip-192-168-193-226:~# curl -m 2 --retry 1 -sSL 10.222.0.1:8080
curl: (28) Connection timed out after 2001 milliseconds
```

服务依然不通。并且在LB节点的iptables日志为空:

```
root@ip-192-168-193-197:~# cat /var/log/syslog | grep 'int32bit ipvs'
root@ip-192-168-193-197:~#
```

也就是说，ipvs的包根本不会经过iptables nat表POSTROUTING链?

那mangle表呢？我们打开LOG查看下:

```
iptables -t mangle -A POSTROUTING -m ipvs --vaddr 10.222.0.1 --vport 8080 -j LOG --log
```

此时查看日志如下:

我们发现在mangle表中可以看到DNAT后的包。

只是可惜mangle表的POSTROUTING并不支持NAT功能:

对比Kubernetes配置发现需要设置如下系统参数:

▲ 赞同 59 ▾ 7 条评论 分享 喜欢 收藏 申请转载 ...



```
root@ip-192-168-193-226:~# curl -i 10.222.0.1:8080
```

HTTP/1.1 200 OK

Content-Type: text/plain

Date: Wed, 27 Nov 2019 15:28:06 GMT

Connection: keep-alive

Transfer-Encoding: chunked

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-v1-c5ccf9784-g9bkx | v=1
```

终于通了，查看RS抓包：

如期望，修改了源IP为LB IP。

原来需要配置 `net.ipv4.vs.conntrack=1` 参数，这个问题折腾了一个晚上，不得不说目前ipvs的文档都太老了。

前面是通过手动iptables实现SNAT的，性能可能会有损耗，于是如下开源项目通过修改lvs直接做SNAT：

- 小米运维部在LVS的FULLNAT基础上，增加了SNAT网关功能，参考[xiaomi-sa/dsnat](#)
- [lvs-snat](#)

除了SNAT的办法，是否还有其他办法呢？想想我们最初的问题，Client节点发出去的包的目标IP和收到的包源IP不一致导致包被丢弃，那解决问题的办法就是把包重新引到LB节点上，只需要在所有的RS节点增加如下路由即可：

```
ip r add 192.168.193.226 via 192.168.193.197 dev ens5
```

此时我们再次检查我们的Service是否可连接：

```
root@ip-192-168-193-226:~# curl -i -m 2 --retry 1 -sSL 10.222.0.1:8080
```

HTTP/1.1 200 OK

▲ 赞同 59

▼

7 条评论

分享

喜欢

收藏

申请转载

...



Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-v1-c5ccf9784-4v9z4 | v=1

结果没有问题。

不过我们是通过手动添加Client IP到所有RS的明细路由实现的，如果Client不固定，这种方案仍然不太可行，所以通常做法是干脆把所有RS默认路由指向LB节点，即把LB节点当作所有RS的默认网关。

由此可知，用户通过LB地址访问服务，LB节点IPVS会把用户的目标IP由LB IP改为RS IP，源IP不变，包不经过iptables的OUTPUT直接到达POSTROUTING转发出去，包回来的时候也必须先到LB节点，LB节点把目标IP再改成用户的源IP，最后转发给用户。

显然这种模式来回都需要经过LB节点，因此又称为双臂模式。

2.4 网关(Gatewaying)模式

网关模式 (Gatewaying) 又称为直连路由模式 (Direct Routing)、透传模式，**所谓透传即LB节点不会修改数据包的源IP、端口以及目标IP、端口**，LB节点做的仅仅是路由转发出去，可以把LB节点看作一个特殊的路由器网关，而RS节点则是网关的下一跳，这就相当于对于同一个目标地址，会有多个下一跳，这个路由器网关的特殊之处在于能够根据一定的算法选择其中一个RS作为下一跳，达到负载均衡和冗余的效果。

既然是通过直连路由的方式转发，那显然LB节点必须与所有的RS节点在同一个子网，不能跨子网，否则路由不可达。换句话说，**这种模式只支持内部负载均衡(Internal LoadBalancer)**。

另外如前面所述，LB节点不会修改源端口和目标端口，因此这种模式也无法支持端口映射，换句话说**LB节点监听的端口和所有RS节点监听的端口必须一致**。

现在假设有LB节点IP为 192.168.193.197，有三个RS节点如下：

- 192.168.193.172:30620
- 192.168.193.194:30620
- 192.168.193.226:30620

创建负载均衡Service并把RS添加到Service中：



注意到我们的Service监听的端口30620和RS的端口是一样的，并且通过 -g 参数指定为直连路由模式(网关模式)。

Client节点IP为192.168.193.226，我们验证Service是否可连接：

```
root@ip-192-168-193-226:~# curl -m 5 -sSL 192.168.193.197:30620
curl: (28) Connection timed out after 5001 milliseconds
```

我们发现并不通，在其中一个RS节点192.168.193.172上抓包：

正如前面所说，LB是通过路由转发的，根据路由的原理，源MAC地址修改为LB的MAC地址，而目标MAC地址修改为RS MAC地址，相当于RS是LB的下一跳。

并且源IP和目标IP都不会修改。问题就来了，我们Client期望访问的是RS，但RS收到的目标IP却是LB的IP，发现这个目标IP并不是自己的IP，因此不会通过INPUT链转发到用户空间，这时要不直接丢弃这个包，要不根据路由再次转发到其他地方，总之两种情况都不是我们期望的结果。

那怎么办呢？为了让RS接收这个包，必须得让RS有这个目标IP才行。于是不妨在lo上添加个虚拟

▲ 赞同 59 ▾ 7 条评论 分享 喜欢 收藏 申请转载 ...



问题又来了，这就相当于有两个相同的IP，IP重复了怎么办？办法是隐藏这个虚拟网卡，不让它回复ARP，其他主机的neigh也就不可能知道有这么个网卡的存在了，参考[Using arp announce/arp ignore to disable ARP](#)。

```
sysctl net.ipv4.conf.lo.arp_ignore=1
sysctl net.ipv4.conf.lo.arp_announce=2
```

此时再次从客户端curl：

```
root@ip-192-168-193-226:~# curl -m 2 --retry 1 -sSL 192.168.193.197:30620
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-v1-c5ccf9784-4v9z4 | v=1
```

终于通了。

我们从前面的抓包中知道，源IP为Client IP 192.168.193.226，因此直接回包给Client即可，不可能也需要再回到LB节点了，即A->B,B->C, C->A，流量方向是三角形状的，因此这种模式又称三角模式。

我们从原理中不难得出如下结论：

- Client、LB以及所有的RS必须在同一个子网。
- LB节点直接通过路由转发，因此性能非常高。
- 不能做端口映射。

2.5 ipip隧道模式

前面介绍了网关直连路由模式，要求所有的节点在同一个子网，而ipip隧道模式则主要解决这种限制，LB节点IP和RS可以不在同一个子网，此时需要通过ipip隧道进行传输。

现在假设有LB节点IP为 192.168.193.77/25，在该节点上增加一个VIP地址：

```
ip addr add 192.168.193.48/25 dev eth0
```

有三个RS节点如下：

- 192.168.193.172:30620

[▲ 赞同 59](#) [▼](#) [7 条评论](#) [分享](#) [喜欢](#) [收藏](#) [申请转载](#) ...



如上三个RS节点子网掩码均为255.255.255.128，即25位子网，显然和VIP 192.168.193.48/25不在同一个子网。

创建负载均衡Service并把RS添加到Service中：

```
ipvsadm -A -t 192.168.193.48:30620 -s rr
ipvsadm -a -t 192.168.193.48:30620 -r 192.168.193.194:30620 -i
ipvsadm -a -t 192.168.193.48:30620 -r 192.168.193.226:30620 -i
ipvsadm -a -t 192.168.193.48:30620 -r 192.168.193.172:30620 -i
```

注意到我们的Service监听的端口30620和RS的端口是一样的，并且通过 `-i` 参数指定为ipip隧道模式。

在所有的RS节点上加载ipip模块以及添加VIP(和直连路由类型)：

```
modprobe ipip
ifconfig tunl0 192.168.193.48/32
sysctl net.ipv4.conf.tunl0.arp_ignore=1
sysctl net.ipv4.conf.tunl0.arp_announce=2
```

Client节点IP为192.168.193.226/25，我们验证Service是否可连接：

```
root@ip-192-168-193-226:~# curl -i -sSL 192.168.193.48:30620
```

```
HTTP/1.1 200 OK
```

```
Content-Type: text/plain
```

```
Date: Wed, 27 Nov 2019 07:05:40 GMT
```

```
Connection: keep-alive
```

```
Transfer-Encoding: chunked
```

```
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-v1-c5ccf9784-dgn74 | v=1
root@ip-192-168-193-226:~#
```

Service可访问，我们在RS节点上抓包如下：

▲ 赞同 59

▼ 7 条评论

分享

喜欢

收藏

申请转载

...

我们发现和直连路由一样，源IP和目标IP没有修改。

所以IPIP模式和网关(Gatewaying)模式原理基本一样，唯一不同的是网关(Gatewaying)模式要求所有的RS节点和LB节点在同一个子网，而IPIP模式则可以支持跨子网的情况，为了解决跨子网通信问题，使用了ipip隧道进行数据传输。

2.4 总结

ipvs是一个内核态的四层负载均衡，支持NAT、Gateway以及IPIP隧道模式，Gateway模式性能最好，但LB和RS不能跨子网，IPIP性能次之，通过ipip隧道解决跨网段传输问题，因此能够支持跨子网。而NAT模式没有限制，这也是唯一一种支持端口映射的模式。

我们不难猜想，由于Kubernetes Service需要使用端口映射功能，因此kube-proxy必然只能使用ipvs的NAT模式。

3 kube-proxy使用ipvs模式

3.1 配置kube-proxy使用ipvs模式

使用kubeadm安装Kubernetes可参考文档[Cluster Created by Kubeadm](#)，不过这个文档的安装配置有问题[kubeadm #1182](#)，如下官方配置不生效：

```
kubeProxy:  
  config:  
    featureGates:  
      SupportIPVSProxyMode: true  
    mode: ipvs
```

需要修改为如下配置：

▲ 赞同 59 ▾ 7 条评论 分享 喜欢 收藏 申请转载 ...



```
kind: KubeProxyConfiguration
mode: ipvs
```

可以通过如下命令确认kube-proxy是否修改为ipvs:

```
# kubectl get configmaps kube-proxy -n kube-system -o yaml | awk '/mode/{print $2}'
ipvs
```

3.2 Service ClusterIP原理

创建一个ClusterIP类似的Service如下:

```
# kubectl get svc | grep kubernetes-bootcamp-v1
kubernetes-bootcamp-v1    ClusterIP    10.96.54.11    <none>           8080/TCP   2m11s
```

ClusterIP 10.96.54.11为我们查看ipvs配置如下:

```
# ipvsadm -S -n | grep 10.96.54.11
-A -t 10.96.54.11:8080 -s rr
-a -t 10.96.54.11:8080 -r 10.244.1.2:8080 -m -w 1
-a -t 10.96.54.11:8080 -r 10.244.1.3:8080 -m -w 1
-a -t 10.96.54.11:8080 -r 10.244.2.2:8080 -m -w 1
```

可见ipvs的LB IP为ClusterIP，算法为rr，RS为Pod的IP。

另外我们发现使用的模式为NAT模式，这是显然的，因为除了NAT模式支持端口映射，其他两种均不支持端口映射，所以必须选择NAT模式。

由前面的理论知识，ipvs的VIP必须在本地存在，我们可以验证:

```
# ip addr show kube-ipvs0
4: kube-ipvs0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN group default
    link/ether 46:6b:9e:af:b0:60 brd ff:ff:ff:ff:ff:ff
    inet 10.96.0.1/32 brd 10.96.0.1 scope global kube-ipvs0
        valid_lft forever preferred_lft forever
    inet 10.96.0.10/32 brd 10.96.0.10 scope global kube-ipvs0
```

[▲ 赞同 59](#) [▼](#) [7 条评论](#) [分享](#) [喜欢](#) [收藏](#) [申请转载](#) ...



首发于
Kubernetes

driver: dummy

可见kube-proxy首先会创建一个dummy虚拟网卡kube-ipvs0，然后把所有的Service IP添加到kube-ipvs0中。

我们知道基于iptables的Service，ClusterIP是一个虚拟的IP，因此这个IP是ping不通的，但ipvs中这个IP是在每个节点上真实存在的，因此可以ping通：

当然由于这个IP就是配置在本地虚拟网卡上，所以对诊断问题没有一点用处的。

我们接下来研究下ClusterIP如何传递的。

当我们通过如下命令连接服务时：

```
curl 10.96.54.11:8080
```

此时由于10.96.54.11就在本地，所以会以这个IP作为出口地址，即源IP和目标IP都是10.96.54.11，此时相当于：

```
10.96.54.11:xxxx -> 10.96.54.11:8080
```

其中xxxx为随机端口。

然后经过ipvs，ipvs会从RS ip列中选择其中一个Pod ip作为目标IP，假设为10.244.2.2：

```
10.96.54.11:xxxx -> 10.96.54.11:8080
```

▲ 赞同 59 ▾ 7 条评论 分享 喜欢 收藏 申请转载 ...



我们从iptables LOG可以验证:

我们查看OUTPUT安全组规则如下:

```
-A OUTPUT -m comment --comment "kubernetes service portals" -j KUBE-SERVICES
-A KUBE-SERVICES ! -s 10.244.0.0/16 -m comment --comment "Kubernetes service cluster i
-A KUBE-MARK-MASQ -j MARK --set-xmark 0x4000/0x4000
```

其中ipsetj集合 KUBE-CLUSTER-IP 保存着所有的ClusterIP以及监听端口。

如上规则的意思就是除了Pod以外访问ClusterIP的包都打上 0x4000/0x4000。

到了POSTROUTING链:

```
-A POSTROUTING -m comment --comment "kubernetes postrouting rules" -j KUBE-POSTROUTING
-A KUBE-POSTROUTING -m comment --comment "kubernetes service traffic requiring SNAT" -
```

如上规则的意思就是只要匹配mark 0x4000/0x4000 的包都做SNAT，由于10.244.2.2是从 flannel.1出去的，因此源ip会改成flannel.1的ip 10.244.0.0：

10.96.54.11:xxxx -> 10.96.54.11:8080

|

| IPVS

v

10.96.54.11:xxxx -> 10.244.2.2:8080

|

| MASQUERADE

▲ 赞同 59

▼

7 条评论

分享

喜欢

收藏

申请转载

...



Node节点通过之前的MASQUERADE再把目标IP还原为10.96.54.11。

3.3 NodeIP实现原理

查看Service如下：

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	30h
kubernetes-bootcamp-v1	NodePort	10.96.54.11	<none>	8080:32016/TCP	8h

Service kubernetes-bootcamp-v1的NodePort为32016。

现在假设集群外的一个IP 192.168.193.197访问192.168.193.172:32016:

192.168.193.197:xxxx -> 192.168.193.172:32016

最先到达PREROUTING链：

```
-A PREROUTING -m comment --comment "kubernetes service portals" -j KUBE-SERVICES
-A KUBE-SERVICES -m addrtype --dst-type LOCAL -j KUBE-NODE-PORT
-A KUBE-NODE-PORT -p tcp -m comment --comment "Kubernetes nodeport TCP port for masque
-A KUBE-MARK-MASQ -j MARK --set-xmark 0x4000/0x4000
```

如上4条规则看起来复杂，其实就做一件事，如果目标地址为NodeIP，则把包标记 0x4000，0x4000。

我们查看ipvs：

```
# ipvsadm -S -n | grep 32016
-A -t 192.168.193.172:32016 -s rr
-a -t 192.168.193.172:32016 -r 10.244.1.2:8080 -m -w 1
-a -t 192.168.193.172:32016 -r 10.244.1.3:8080 -m -w 1
-a -t 192.168.193.172:32016 -r 10.244.3.2:8080 -m -w 1
```

▲ 赞同 59

▼ 7 条评论

分享

喜欢

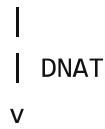
收藏

申请转载

...



192.168.193.197:xxxx -> 192.168.193.172:32016



192.168.193.197:xxx --> 10.244.3.2:8080

剩下的到了POSTROUTING链就和Service ClusterIP完全一样了，只要匹配 0x4000/0x4000 的包就会做SNAT。

3.4 总结

Kubernetes的ClusterIP和NodePort都是通过ipvs service实现的，Pod当作ipvs service的server，通过NAT MQSQ实现转发。

简单来说kube-proxy主要在所有的Node节点做如下三件事：

1. 如果没有dummy类型虚拟网卡，则创建一个，默认名称为 kube-ipvs0；
2. 把Kubernetes ClusterIP地址添加到 kube-ipvs0，同时添加到ipset中。
3. 创建ipvs service，ipvs service地址为ClusterIP以及Cluster Port，ipvs server为所有的Endpoint地址，即Pod IP及端口。

使用ipvs作为kube-proxy后端，不仅提高了转发性能，结合ipset还使iptables规则变得更“干净”清楚，从此再也不怕iptables。

更多关于kube-proxy ipvs参考[IPVS-Based In-Cluster Load Balancing Deep Dive.](#)

4 总结

本文首先介绍了kube-proxy的功能以及kube-proxy基于iptables的实现原理，然后简单介绍了ipvs，了解了ipvs支持的三种转发模式，最后介绍了kube-proxy基于ipvs的实现原理。

ipvs是专门设计用来做内核态四层负载均衡的，由于使用了hash表的数据结构，因此相比iptables来说性能会更好。基于ipvs实现Service转发，Kubernetes几乎能够具备无限的水平扩展能力。随着Kubernetes的部署规模越来越大，应用越来越广泛，ipvs必然会取代iptables成为Kubernetes Service的默认实现后端。

▲ 赞同 59



▼ 7 条评论

分享

喜欢

收藏

申请转载





发布于 2019-11-30

[Kubernetes](#) [LVS \(Linux Virtual Server\)](#) [容器（虚拟化）](#)

▲ 赞同 59 ▾ 7 条评论 分享 喜欢 收藏 申请转载 ...



首发于
Kubernetes



Kubernetes

学习、研究与分享Kubernetes相关技术

关注专栏

推荐阅读

Kubernetes 实现原理

引言书接上文，前面我们主要讲了 Kubernetes 的实践过程，这一篇文章中，我们着重介绍 Kubernetes 的实现原理。Kubernetes 实现原理现在我们已经知道了大多数可以部署到 Kubernetes 的资源...

贝克街的流... 发表于贝贝猫技术...



Kubernetes 从懵圈到熟练
群服务的三个要点和一种实

涂南Amb... 发表于白话云

7 条评论

切换为时间排序

写下你的评论...



京舟2020

2020-03-16

不错。

赞

一介刁民

2020-03-18

纠正一个问题，“因此ClusterIP只能是单主机(Host Only) 作用域可见，这个IP在其他节点以及集群外均无法访问。”不是单主机作用域，在集群内的其他节点也是可以访问的

1

int32bit (作者) 回复 一介刁民

2020-03-18

那是因为这个cluster ip在所有node节点上都配置了一遍

▲ 赞同 59



▼ 7 条评论

分享

喜欢

收藏

申请转载





首发于
Kubernetes

志人母 | Kube-proxy的云扛架研十日月月SVC的云头现日，月人义十日月元/云个人入云日：

1

展开其他 1 条回复



lx1036

2020-04-29

非常优秀的文章，精读了好几遍，还需要再来几遍。是我深入学习iptables/ipvs/kube-proxy 的首选参考资料。

1



snowtree

2020-12-27

使用CNI后还依赖kube-proxy的能力吗？

先看

▲ 赞同 59



▼ 7 条评论

分享

喜欢

收藏

申请转载

