导读：

现代互联网上的大部分web服务都是通过集群方式部署，我们每打开一个网页，或者一个手机应用，后台可能有成千上万台web服务器在同时提供服务，一个用户的请求是如何到达后台的服务器的呢？中间起着连接和桥梁作用的就是负载均衡器（LoadBalancer），负载均衡器顾名思义就是将负载（用户请求）进行平衡，均匀的分摊到后台的真实服务集群。长期以来, eBay的生产和测试环境都是用Citrix NetScaler硬件负载设备，这种专门为流量负载均衡而设计的硬件提供了强大和高效的L4/L7流量管理，但是也有两个比较明显的缺点：一个是贵，这决定了硬件负载均衡设备数目不会太多，尤其是在测试环境。另外一个缺点是不够弹性化, 负载均衡设备带宽是固定的，比如有的是30Gb，而随着流量的增加容易导致LB流量过高，尤其是在购物季的时候，通常就会需要LB运维的同事们做很多LB VIP的迁移，将一些流量过高的VIP拆分迁移到流量相对较少的LB上。
为了解决这两个痛点，软件负载均衡是一个很有希望的方案，通俗的说，它就是能将一组普通的Linux服务器变成一组软件负载均衡器，能够支持所有我们需要的L4和L7的流量管理，并且能够支持横向扩展，比如一台Linux服务器物理网卡的带宽是10Gb，理论上只需要15台这样的服务器就能够提供150Gb的总带宽。下面我将和大家分享一下eBay软件负载均衡在开发环境网络和生产环境网络具体实现的一些细节。

1. L4负载均衡

L4负载均衡是指工作在OSI模型第四层，也就是传输层的负载均衡，这一层的主要协议就是TCP/UDP, 负载均衡器在这一层能够看到数据包里面的源端口地址以及目的端口地址，并且基于这些信息通过一定的负载均衡算法将数据包转发到后端真实服务器。eBay的4层负载均衡主要是基于IPVS，同时也需要结IPTables才能实现完整的功能，所以先对IPVS以IPTables作一些介绍。

1.1. LVS

LVS(Linux Virtual Server 虚拟服务器)：是一个虚拟的四层路由交换器集群系统,根据目标地址和目标端口实现用户请求转发。LVS有两段代码组成。一个是ipvsadm，工作在用户空间，负责为ipvs内核框架编写规则，定义谁是集群服务，谁是后端真实的服务器。另一个是IPVS, 工作在内核空间，是真正生效实现调度的代码。

eBay的软件负载均衡是基于IPVS，它实现了传输层负载均衡，也就是我们常说的4层LAN交换，作为 Linux 内核的一部分，IPVS运行在主机上，在真实服务器集群前充当负载均衡器。IPVS可以将基于TCP和UDP的服务请求转发到真实服务器上，并使真实服务器的服务在单个 IP 地址上显示为虚拟服务。在内核2.4.22之后，IPVS直接被收录进内核源码树了，只要启用了相关功能就可以直接拿来使用了。

IPVS相当于工作在netfilter中的INPUT链，它挂接在IP报文遍历的LOCAL_IN链和IP_FORWARD链两处，用于截取/改写IP报文，然后再转发出去，如图1-1所示：

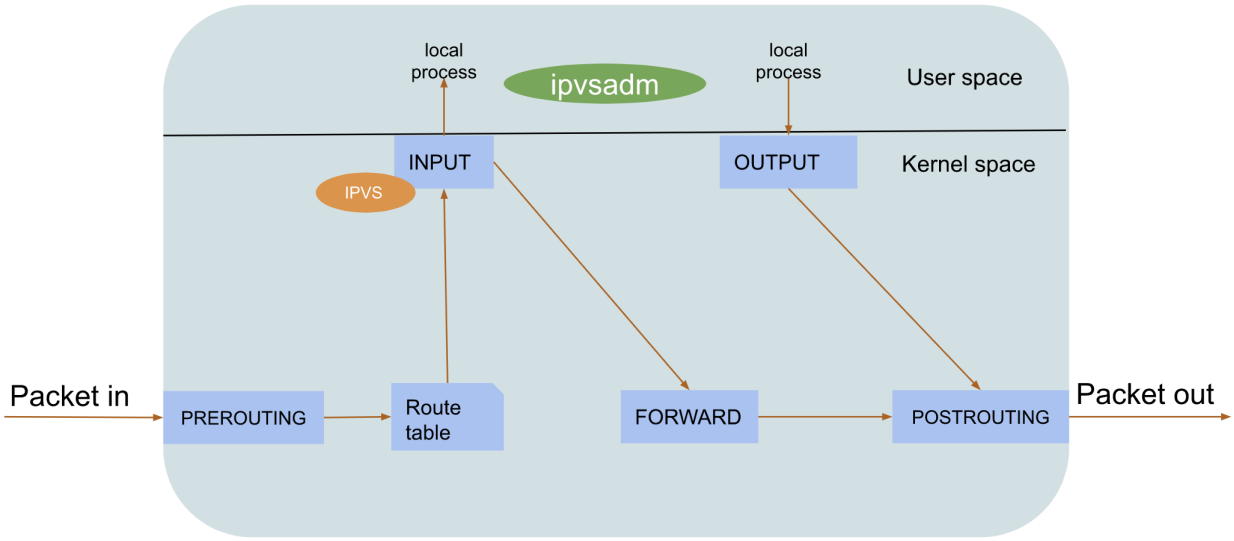

图1-1 IP报文在内核的遍历图

IPVS工作原理如下：

1. 客户端向负载均衡调度器（Director Server）发起请求，调度器将用户请求发送到内核空间。
2. PREROUTING收到用户请求后，先判断目的IP是否为本机IP，如果是就将报文发到INPUT链。
3. 由于IPVS工作在INPUT链上，当用户请求到达INPUT时，IPVS会将用户请求的目的和已经定义好的IPVS规则进行对比，如果请求目的地为VIP地址，IPVS调度器会选择一个后端真实服务器然后修改数据包里的目标IP地址，并将新的数据包发往POSTROUTING链。
4. POSTROUTING链接收数据包后发现目标IP地址是自己的后端服务器，那么此时通过路由决策，将数据包最终发送给后端的服务器。

IPVS+IPTables的报文处理流程如图1-2所示：

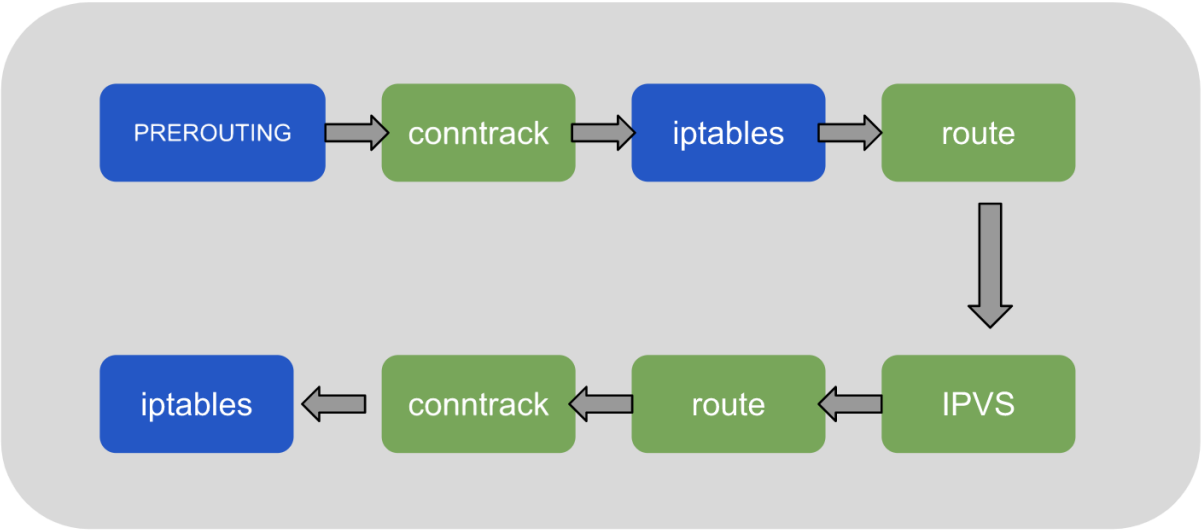

图1-2 IPVS模式报文经过的模块

1.1 IPVS调度算法

IPVS在内核中的负载均衡调度是基于连接进行调度，每个客户端请求与服务器端都需要建立一个TCP连接，调度算法负责将不同用户的连接均匀分配到后端的真实服务器上，这样可以避免单个服务器创建过多的连接而导致服务器之间的负载不平衡。
在内核中的连接调度算法上，IPVS已实现了很多种调度算法，下面是几种最常见的：

轮叫调度（Round-Robin Scheduling）
这种算法通过轮叫的方式依次将请求调度到后端真实的服务器。假设有n台真实服务器，每次执行i = (i + 1) mod n，并选出第i台服务器。这种算法简洁并且不需要记录连接的状态所以所以它是一种无状态调度，由于它假设集群中服务处理性能是一样的，当用户请求时间变化比较快的时候容易导致服务器之间负载不平衡。

加权轮叫调度（Weighted Round-Robin Scheduling）
在rr算法的基础上引入一个weight，按weight的比例来调度真实的服务器。

最小连接调度（Least-Connection Scheduling）
这种算法会把新的连接请求发送给当前连接数最少的后端服务器。调度器会记录每个服务器当前的连接数，当一个请求被调度到一个服务器时，连接计数器加一，反之当连接断开时，计数器减一。

内核中每种调度算法均被实现为一个内核模块，在需要时加载。其中ip_vs_tbl是我们使用的自定义的调度器，会在后文介绍。

```
# lsmod | grep ip_vs
bash-5.0# lsmod | grep ip_vs
ip_vs_sh               16384  0
ip_vs_wrr              16384  0
ip_vs_rr               16384  12851
ip_vs_tbl              16384  1454
ip_vs                 184320  14313 ip_vs_rr,ip_vs_sh,ip_vs_tbl,ip_vs_wrr
```

1.2 IPTables/Netfilter

IPTables是一个配置Linux内核防火墙的命令行工具，它基于内核的Netfilter机制，Netfilter是Linux内核的包过滤框架，它提供了一系列的钩子（Hook）供其他模块控制数据包的流动，这中间涉及到“四表五链”，是在两个维度（规则功能以及规则所处链路）对规则进行分组，“四表”存放着功能一致的规则，“五链”存放着数据包所处链路一致的规则，具体如下：

四表:
Filter表:过滤数据包。
NAT表:用于网络地址转换(IP、端口)。
Mangle表:修改数据包的服务类型、TTL、并且可以配置路由实现QOS。
Raw表:决定数据包是否被状态跟踪机制处理。

五链：
INPUT链：经过路由查找后，送往本机（目的地址在本地）数据包应用此规则链中的规则。
OUTPUT链：本地生成的发往其他机器的数据包应用此规则链中的规则。
FORWARD链：非本地产生的并且目的地不是本地的包（转发数据包）应用此规则链中的规则。
PREROUTING链：刚通过数据链路层解包进入网络层的数据包，在做出路由选择前应用此链中的规则。
POSTROUTING链：数据包离开本机之前以及作路由选择后应用此链中的规则。

IPTables的表和链的流程如图1-3所示：

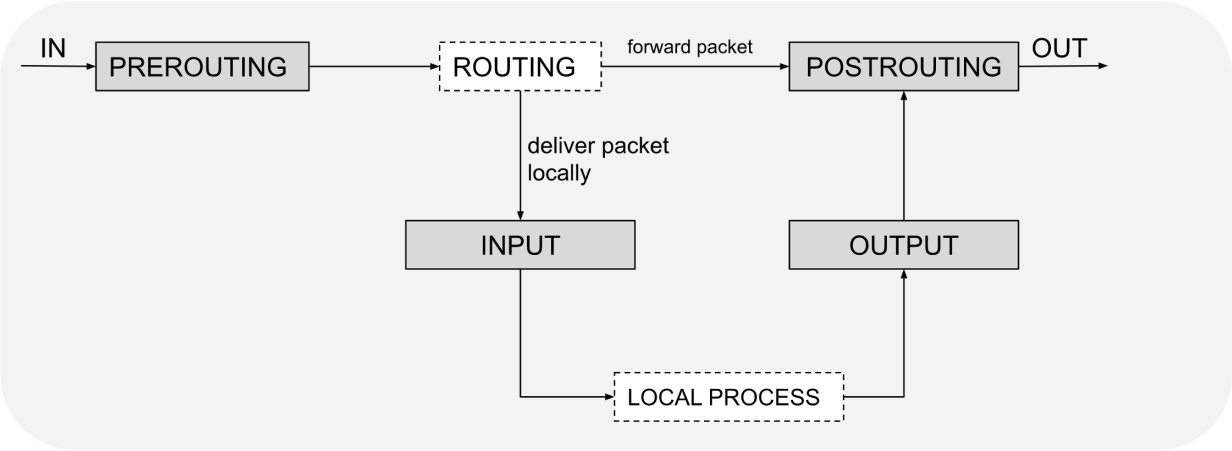

图1-3 IPtables的表和链

其中PREROUTING和POSTROUTING是最重要的两个链：

1. PREROUTING负责处理刚从网卡进入的数据包，在此之前还没有做路由决策，所以还不确认这个数据包是应该交给本机处理，或者是要转发到连接着另外一个网卡的计算机。数据包通过PREROUTING链后，将做出路由决策。如果数据包是发往本地计算机，则该数据包将被转发到相应的进程。如果目的地址不是在本地，那么就会将数据包转发到相应的接口，或者通过默认网关转发出去。
2. 在数据包离开机器之前, 它通过POSTROUTING链，然后通过网络接口离开本机。对于本地生成的数据包, 这里有一点差别：数据包不会经过PREROUTING链，而是通过OUTPUT链，然后转移到POSTROUTING链。因此，只有本地应用程序生成的数据包才会经过OUTPUT链，而所有数据包（包括从其他地方路由的数据包）都会经过POSTROUTING链。

在eBay的软件负载均衡实现中，需要IPTables配合IPVS才能实现完整的功能，IPTables在这里主要有两个作用:

1. 由于IPVS IPIP tunnel模式不支持端口转发，只有NAT模式才支持，但实际上基于Tess软件负载均衡的service是支持端口转发的，也就是VIP:port1转发到backend:port2, 这里的端口转发正是通过IPTables规则实现的。
2. 在mangle表中的PRE-ROUTING链给IPVS service的VIP创建MARK规则，TLB控制器会对所有访问某一个VIP的数据包的5元组进行哈希，再将哈希值对256取模并将结果标FWMARK值，这个值会被IPVS调度器用来寻找真实服务器。

2. TLB(Tess load balancer)

IPVS只是kernel的模块，这里我们定义了一系列TLB相关的资源以及控制器，最终实现了VIP的路由规则创建以及可编程的IPVS，并以此为核心提供4层负载均衡的功能，相关的主要资源包括以下：

TLB：是对硬件负载均衡的LB Pool的抽象.
TLBVIP：对应于负载均衡的VirtualIP。
TLBMember：对应于后端真实服务器。
TLBGroup: 用来对TLB分组，实现TLB的分区。一个TLBGroup相当于一个独立的软件负载集群，每一个TLBGroup配置了1000个VIP。
Route:用来定义路由，包括主机路由以及交换机路由。
Allocation：用来定义每一个分配的IP，包括Pod IP以及VIP。
Block: 用来定义一个VIP地址块。

以上的每一个资源基本都有一个相应的控制器，比如TLB controller， TLBMember controller, Route controller，这些控制器会list-watch相应的资源并且完成相应的配置。

TLB架构图如图2-1所示：

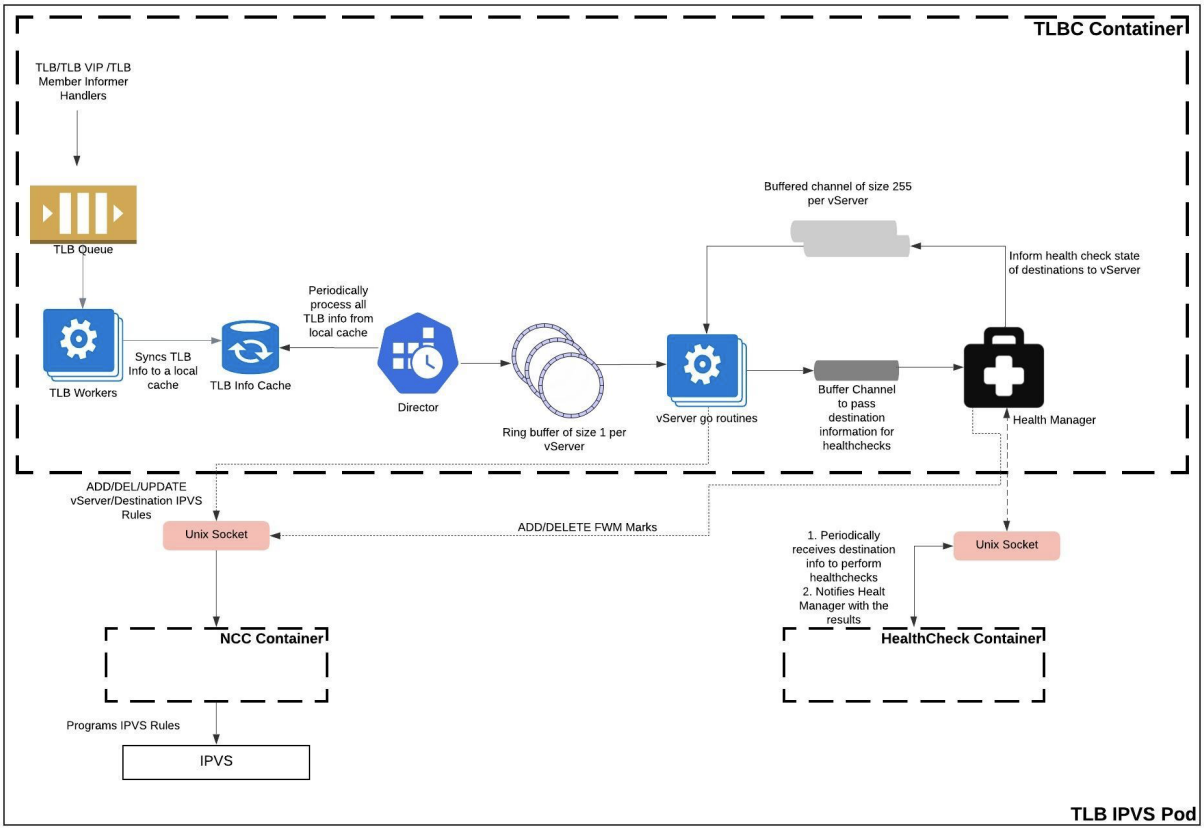

图2-1 TLB架构图

每一个TLB Pod都有三个容器，分别是tlbc(TLB controller), ncc(network control Centre)以及health check容器。

2.1 TLBC Container

TLB控制器是一个典型的k8s控制器，它list-watch TLB，TLBVIP以及TLBMember，然后分别处理每个资源的event。TLB控制器的worker会检查每一个被更新的TLB对象并且更新TLB info-cache, 这个缓存里面包含了相应的VIP和TLBMember信息。

TLB控制器里面有一个L4 Director，它的主要作用是周期性的遍历一遍TLB info cache，对cache里面的每个条目会进行如下处理：

1. 基于每一个TLB info cache生成一份service的spec，并且确保每个service会启动一个vServer的go routine，它其实就是一个vSever manager。
2. 在TLB Pod的network namespace给VIP创建HMARK规则，这个规则会被IPVS调度器用来给每一个进入TLB Pod的连接选择一个后端服务器。
3. 给IPVS的destination(也就是后端真是服务器）分配一个weight，这个weight也被调度器用来选择真实服务器。这个值对应的其实就是#2里面HMARK规则对五元组对256取模的值。

4. 给每个IPVS service的destination生成一致的哈希表，并将其提供给vServer Manager。
5. 删除已经不在TLB info cache里面的IPVS service的HMARK以及IPVS规则。

VServer go routine会分别处理自己相应的Virture Server Spec，其中包含VIP和真实服务器的信息, 对每一份VServer Spec它会做下面几件事情：

1. 给IPVS service构建一份health check 的spec，里面包含destination的信息以及健康检查的类型，我们用到的是TCP类型，健康检查的spec通过buffer channel转交给HealthCheck Manager。
2. 注册一个channel从HealthCheck manager接收destination的健康检查结果, 如果至少有一个健康的后端服务器，就会通过unix域套接字与NCC容器进行交互，然后给这个service创建或者更新IPVS规则。

其工作流程如图2-2所示：

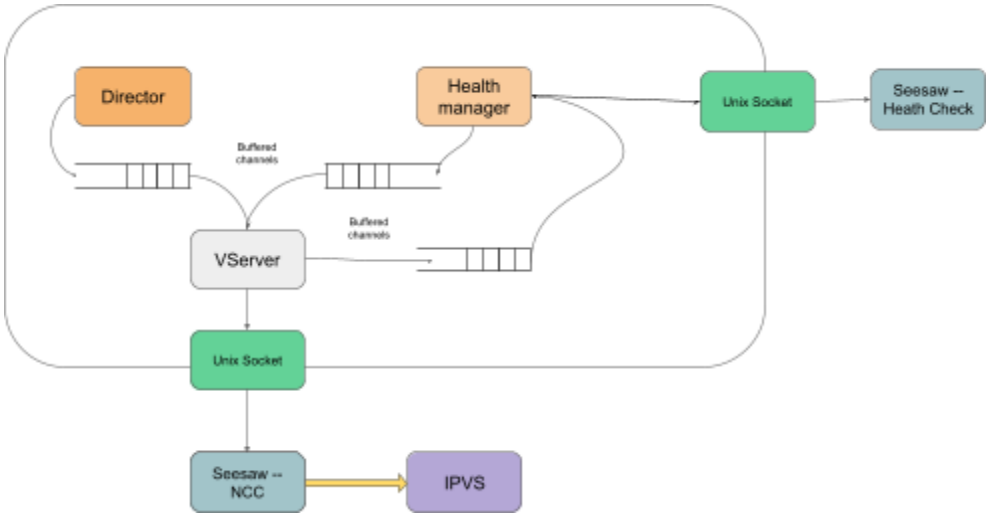

图2-2 TLBC工作流程图

2.2 Health Check Container

Health Check模块是从Google的一个基于LVS的开源负载均衡项目Seesaw fork出来的, 它的主要功能就是通过UNIX域套接字从TLBC容器接收健康检查的配置，并且通过套接字将健康检查结果返回给TLBC容器。它主要有三个组件：HC Updater, HC Manager and HC Notifier , 并且支持多种协议的健康检查类型，包括TCP，UDP，HTTP以及ICMP。健康检查容器会基于指定的检查间隔周期性的执行健康检查并且通过gRPC机制将结果返回给TLBC，各个组件具体功能如下：

1. HC Updater 基于配置的间隔从TLBC容器接收健康检查的配置，并且通过buffered channel传递给HC Manager。
2. HC Manager创建HealthCheck go routine，并且批量的为每一个五元组周期性的执行健康检查，如果发生了检查结果发生了状态改变便会更新HC Notifier。

3. HC Notifier将健康检查结果更新到TLBC，前提是满足以下任意一个条件：
    a. 如果需要通知健康检查结果的数量超过了配置指定的batch数量。
    b. 如果健康检查的时间超过了一次批量检查的延时。

其工作流程如图2-3所示：

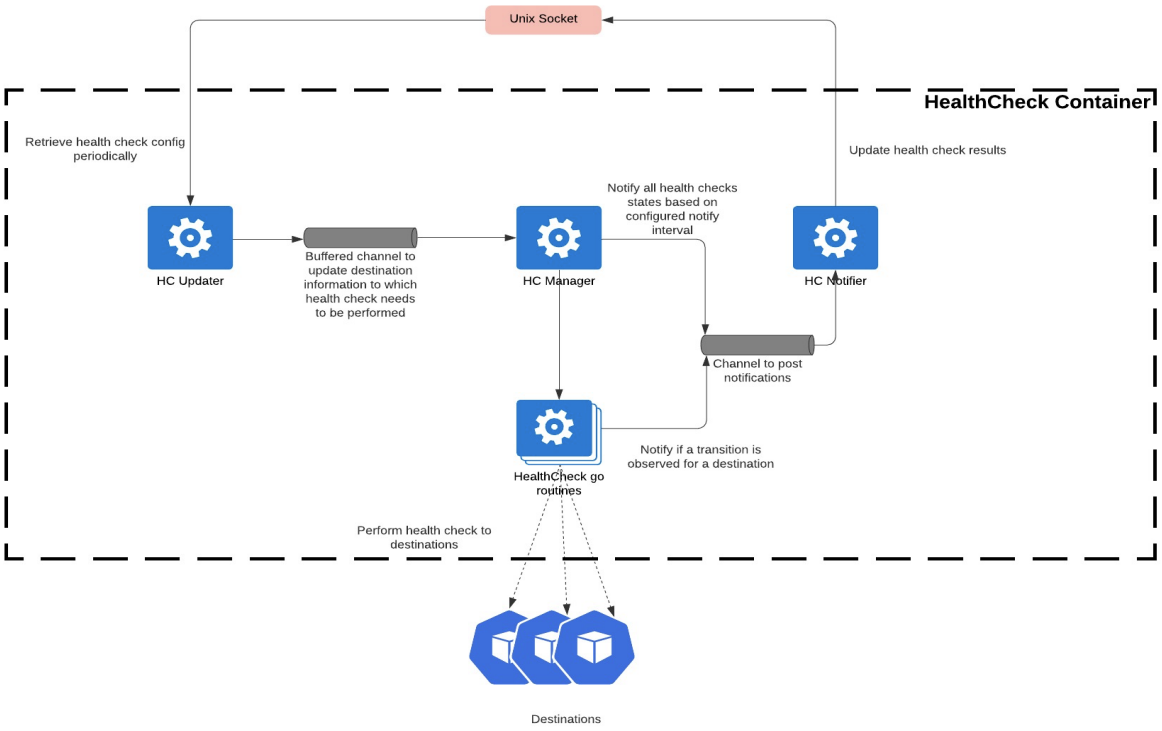

图2-3 Health Check控制器工作流程图

2.3 NCC container

NCC(network control centre) 容器也是Seesaw项目里面的一个模块，它通过UNIX域套接字与TLBC通信以创建IPVS规则。NCC本身使用netlink与IPVS内核进行通信，它相当于是TLBC与Linux kernel之间的一个桥梁，TLBC通过NCC对IPVS规则进行增删查改的管理。其架构图如图2-4所示：

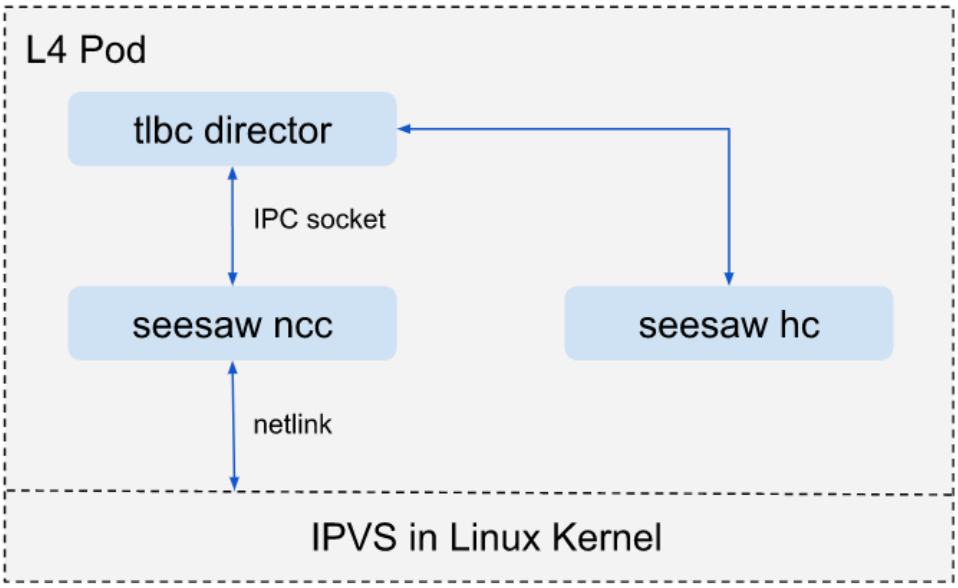

图2-4 NCC架构图

2.4 IPVS自定义调度器

前面已经介绍了IPVS，IPTables以及TLB的主要的组件，我们还引入了一个自定义的调度器。虽然IPVS本身已经提供了多种调度器，但是没有很好的满足业务需求，比如在实际的应用场景中我们对连接需要支持会话保持，也就是同一个客户端会始终连接到同一台后端真实服务器，并且不管是通过HTTP还是HTTPs访问都会到同一个真实服务器。

这里我们引入了一个ip_vs_tbl的自定义kernel模块，也就是一个基于一致性哈希表的调度器，采用的是rendezvous哈希算法。每个IPVS service的哈希表都会通过TLBC生成，也就是在用户空间生成，然后利用Seesaw的NCC通过netlink消息发送的内核的IPVS模块，具体流程如图2-5所示：

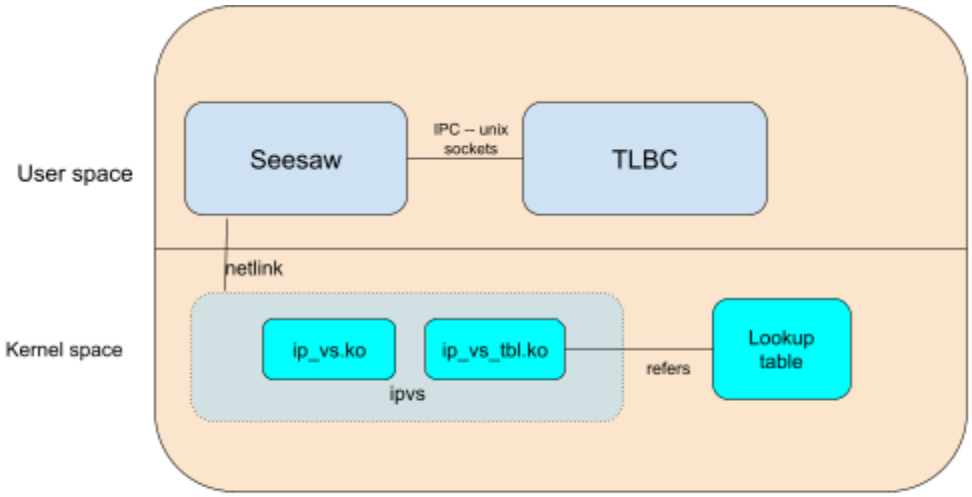

图2-5 IPVS一致性哈希表生成流程图

它具体的工作流程如下:

1. 通过iptables给用户请求目的地为VIP的数据包打上HMARK，其值为5元组mod 256
2. ip_vs将通过ip_vs_tbl调度器从哈希表中挑选出一个后端真实的服务器地址。
3. ip_vs_tbl使用数据包的HMARK作为VIP表的索引，假设mark = 2，在VIP调度表中使用index = 2。
4. 查找调度表可以知道，当index为 2时，dst值为100，也就是目的地址权重为100。
5. 由于目的地址的权重在表生成期间是唯一的，所以调度器能够根据权重从哈希表中反向查找到权重为100的IPVS目标地址，也就是这个VIP的目的地址。
6. 转发数据包到选定的目的地址。

下面是为一个VIP生成的哈希表的一个示例：

```
VIP: 10.0.0.1:80
Members - ip1:80, ip2:80
```

TLBC会给每一个目的地址分配一个唯一的权重，权重值介于0到255之间，会被用来生成哈希表以及调度器选择后端真实地址。

```
ip1:80 - 100
ip2:80 - 103
```

生成的VIP调度表:

```
index     dst
0         100
1         103
2         100
..
..
255       103
```

TLBC会在PREROUTING链中给所有VIP创建IPTables规则，这条规则基于5元组对传入的数据包进行哈希处理并对256取模，将值赋给HMARK：

```
-A PREROUTING -d 1.2.3.4/32 -p tcp -m comment --comment "myapp-feature/myapp-feature-1-app-80"
-m tcp ! --dport 0 -j HMARK --hmark-src-prefix 32 --hmark-dst-prefix 32 --hmark-sport-mask 0xffff
--hmark-dport-mask 0xffff --hmark-rnd 0x00000064 --hmark-mod 256 --hmark-offset 0
```

我们可以在TLB Pod的mangle表中看到这两条规则：

```
iptables -t mangle -v -L PREROUTING -n | grep '1.2.3.4'
167 29676 HMARK tcp  --  *      *       0.0.0.0/0 1.2.3.4   /* myapp-feature/myapp-feature-1-app-80 */
tcp dpt:!0 HMARK mod 256 + 0x0 src-prefix 32 dst-prefix 32 sport-mask 0xffff dport-mask 0xffff rnd 0x64

167 29676 HMARK tcp  --  *      *       0.0.0.0/0 1.2.3.4   /* myapp-feature/myapp-feature-1-app-443 */
tcp dpt:!0 HMARK mod 256 + 0x0 src-prefix 32 dst-prefix 32 sport-mask 0xffff dport-mask 0xffff rnd 0x64
```

正因为有这条规则，所有进入TLB Pod的目的地址为VIP数据包都会由IPTables计算出一个HMARK值，这个值也就对应真实地址的权重，调度器就可以根据这个值挑选出一台后端真实服务器并且保证了同一个client的请求会到同一台后端真实服务器。
通过ipvsadm工具，我们可以看到一个IPVS service在TLB pod上的规则，从这里也可以看出我们的IPVS service是基于自定义的tlb调度器，并且采用了ipip tunnel模式。

```
bash-5.0# ipvsadm -S -n | grep 1.2.3.4
-A -t 1.2.3.4:80 -s tbl
-a -t 1.2.3.4:80 -r 5.6.7.8:80 -i -w 20 --tun-type ipip
-A -t 1.2.3.4:443 -s tbl
-a -t 1.2.3.4:443 -r 5.6.7.8:443 -i -w 20 --tun-type ipip
```

2.5 TLB Service controller

Kubernetes本身也定义了Service的资源，用来为一组具有相同功能的容器应用提供一个统一的入口地址以及将请求进行负载分发到后端的各个容器应用上。Service有多种类型：
ClusterIP:提供一个集群内部的虚拟IP以供Pod访问。
NodePort:在每个Node上打开一个端口以供外部访问。
LoadBalancer:通过外部的负载均衡器来访问。

对于LoadBalancer的Service，k8s并没有提供具体的实现，而前面我们介绍的关于TLB的CRD, 其实是对L4负载均衡的一种抽象，所以我们可以把TLB当成是LoadBalancer Service的一种具体的实现，通过定义TLB相关的资源实现了对k8s的Service的扩展，而TLB的负载均衡既可以是通过软件实现，也就是TLB-IPVS Provider，也可以是硬件实现，基于Netscaler的硬件设备以及硬件负载均衡管理系统，也就是TLB-LBMS Provider，这两种我们都实现了。

TLB Service controller主要做下面几件事情:

1. List-watch k8s Service的事件。
2. 如果是TLB-IPVS Provider的Service, 就去创建/更新所有TLB相关的资源。

2.6 Netd DaemonSet

Netd DeamonSet实际上是eBay内部实现的容器网络的 API 接口(CNI)插件，用来配置容器的网络。它运行在主机网络模式下，并且以privileged模式运行CNI插件负责将网络接口插入容器网络名称空间（例如，veth对的一端），并在主机上进行任何必要的更改（例如，将veth的另一端连接到网桥），然后它会将IPAM控制器分配的IP配置到容器网卡并设置路由。

由于TLB是基于IPIP tunnel, 所以Netd还需要做一件事情就是List-watch属于当前node的TLBMember, 并给这些TLBMember对应的后端真实服务器Pod设置tun0的网卡，并且关闭该网卡的ARP以及将TLB VIP绑定到tun0网卡。

2.7 TLB Service时序图

图2-6是创建一个TLB service的顺序图：

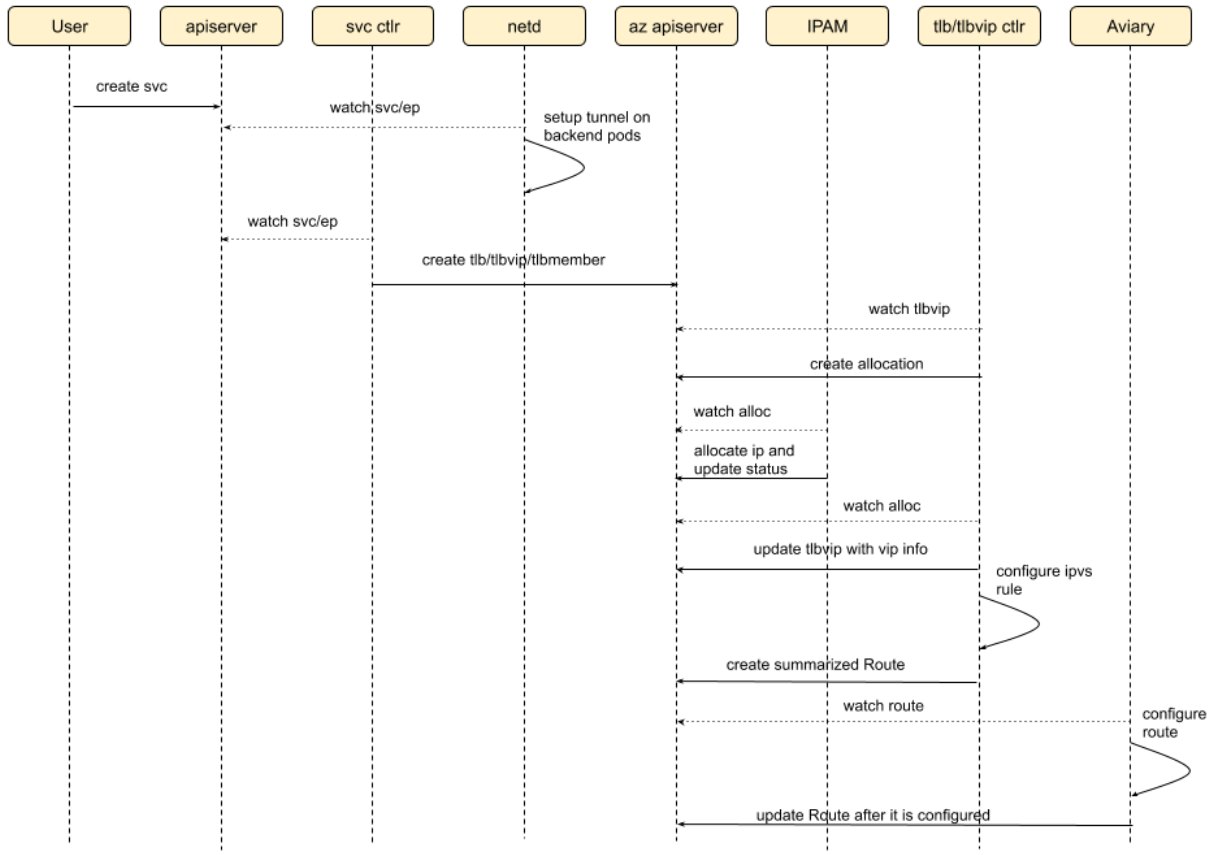

图2-6 创建TLB service的顺序图

其中大部分的控制器前面都已经有介绍过，IPAM是IP allocation manager，Aviary是eBay的route控制器, 中文意思是鸟巢，因为它里面有各种route Provider，包括支持BGP的Bird,我们自己实现的OVN route controller，还包括主机路由控制器。

3. TLB生产开发环境集成

为了隔离生产环境以及开发环境，eBay通过OVN对数据中心的网络做了虚拟化，这样达到的目的是:虽然生产环境和开发环境的物理网络都在数据中心，但是它们之前是隔离的，这样能够提高生成环境的安全性但是也引入了一些其他的问题：

1. 生产环境就是基于物理网络的flat network，这种网络环境对BGP有很友好的支持，这些协议本身具有健康检查机制。
2. 开发环境基于OVN，也就是overlay network，目前社区还没有实现BGP，所以不能直接通过BGP将VIP宣告出去。

也正是由于这两个问题，我们需要对TLB分别对生产和开发环境集成。

3.1 TLB on OVN

OVN是eBay开发环境的SDN Provider, 用来管理覆盖网络(Overlay Network), 它是一种创建在另一网络之上的计算机网络，OVN的网络架构如图3-1所示，它相当于是在传统的物理网络上再建立了一层虚拟的网络，然后在覆盖网络里面定义新的路由，但实际上所有的数据依旧通过物理网络传输。

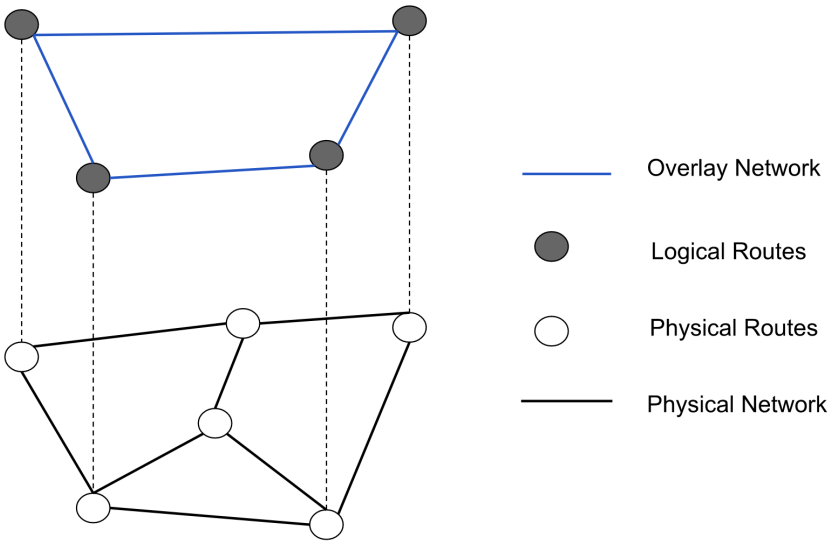

图3-1 OVN网络架构图

之所以采用OVN的主要目的是为了和生产环境隔离以及对开发环境进行网络分割。由于OVN不支持BGP以及BFD（Bidirectional Forwarding Detection），这里主要需要解决两个问题：

1. 如何将TLB上面配置的VIP宣告到OVN的logical gateway?
2. 如何支持TLB集群运行在active/active的模式?

对第一个问题，解决的方案是我们定义了Route的CRD，TLB Pod会去创建这个对象，里面指定了当前TLB Pod配置的VIP的网段以及nextHop为当前TLB Pod的IP，同时会有一个OVN route controller来list-watch这个Object，并且会在OVN的网关里面创建这条静态路由，这样也就确保了所有目的地址是在这个VIP子网的数据包会被转发到这个TLB Pod，其架构如图3-2所示：

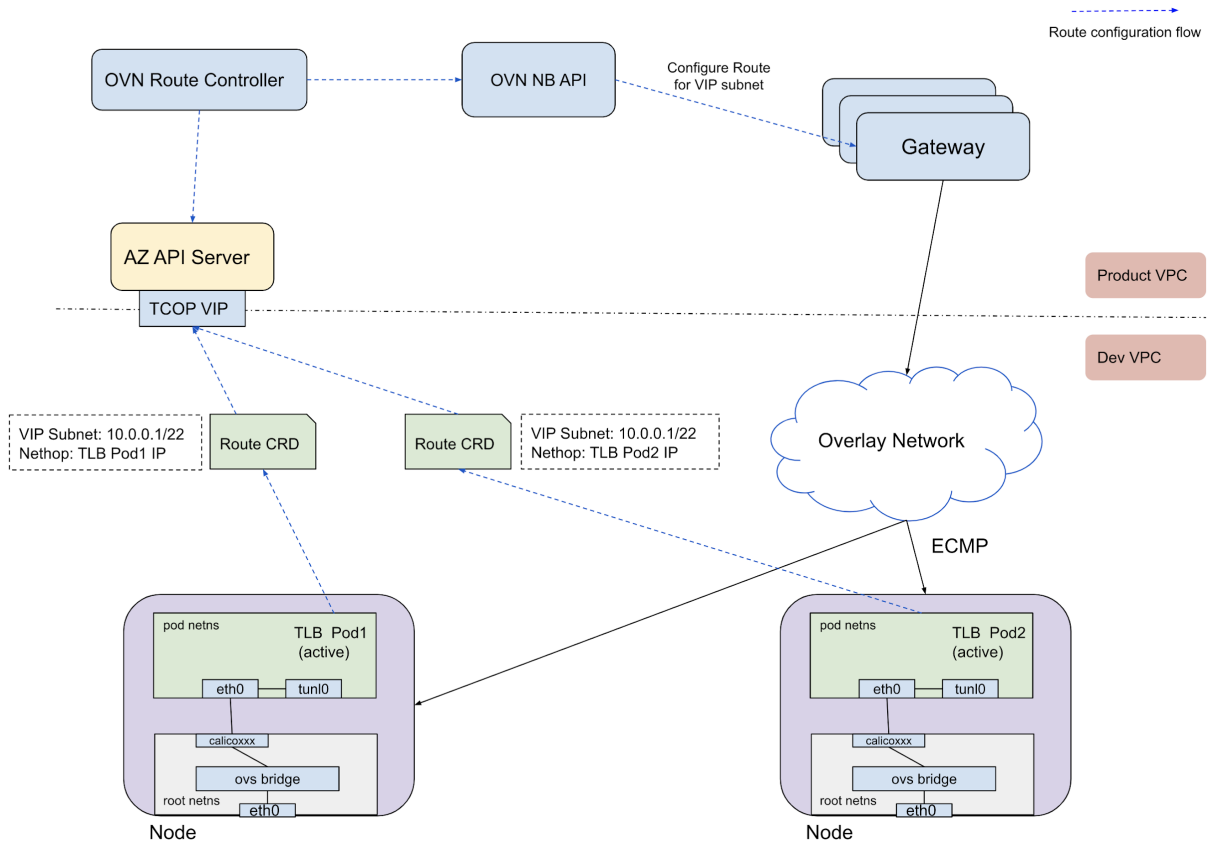

图3-2 OVN路由控制器架构图

第二个问题是怎么实现TLB active/active，这是让TLB能够真正上线的一个前提。OVN虽然不支持BGP但是支持ECMP, 也就是等价路由算法, 所以我们可以让TLB以一个集群的方式运行, 并且每个TLB Pod都能接受流量, 但是这里缺少类似BGP的心跳检测，虽然要做的事情很简单：检测坏的TLB Pod，然后将这条静态路由停用，但是为了实现尽可能快的检测以及避免网络partition导致的误判，这里我们讨论了很久，中间做了很多POC，尝试了以下的方案：

1. 利用Raft算法将TLB Pod集群转成一个raft cluster。Raft是一种共识算法，可以在毫秒级别选举出leader并且对follower做心跳检测，通过这套机制我们可以选举出一个Leader的TLB Pod并维护follower的路由状态。但是这个协议有一个致命的缺陷，就是如果集群的大多数宕机了, 整个raft算法就失效了。
2. 集中式的健康检查控制器。相当于再单独部署一个控制器专门为这些TLB pod做健康检查，这个方案的问题是引入了额外的依赖而且没有办法解决网络parition的问题。
3. 尝试利用类似etcd operator的方式来维护TLB集群的状态，operator相当于一个控制器它本质上也是通过raft协议来维护集群，但是还是没有办法解决多数TLB pod宕机的问题。
4. 最终解决方案还是依靠kubernetes的leader election再加上Lease object, 有点类似k8s node controller通过lease object管理node的状态，但这里是通过leaderelection选举出

来的leader TLB pod来检其它follower的lease object，每个TLB pod 500ms更新一次自己的状态。这里有个潜在的问题就是将etcd也变成数据面了，因为如果etcd出现问题，所有的follower TLB pod就被mark down了。为了避免这种情况，我们加上了双保险，也就是在每个TLB pod上面暴露一个健康检查的端口，follower pod只有连续三次同时没有更新lease object以及健康检查不通过的时候才会被mark down。在我们实验的时候，基于以下leader election的设置，能够2秒钟mark down follower的路由以及5秒钟mark down leader的路由。

a. DefaultLeaseDuration = 2000 * time.Millisecond
b. DefaultRenewDeadline = 1500 * time.Millisecond
c. DefaultRetryPeriod = 1000 * time.Millisecond

其架构如图3-3所示：

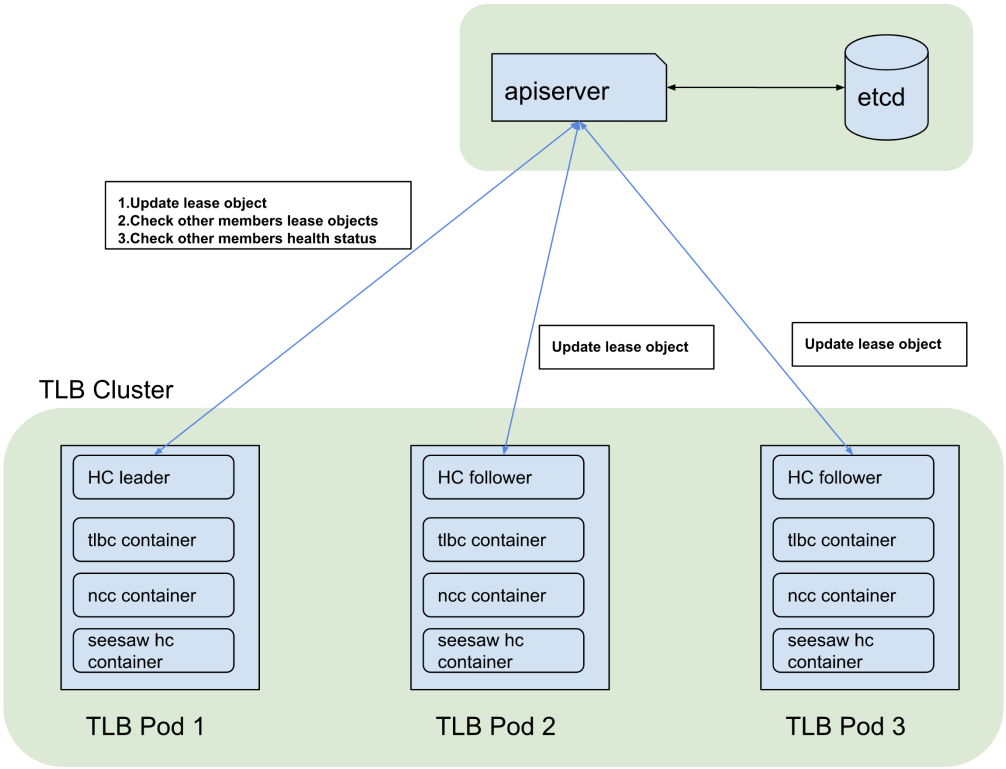

图3-3 TLB on OVN健康检查机制

3.2 TLB on BGP

eBay的生产环境和dev环境其实物理硬件设备是一样的，都是数据中心的机柜，但生产环境是flat network，能够支持BGP协议，这里我们用到了BIRD，它是一个实现多种动态路由协议（如 OSPF、BGP、RIP 等）的类Unix系统的开源路由守护程序。有了Bird，生产环境下的TLB VIP就可以直接通过BGP协议宣告到TOR。具体的实现是Route controller依旧会

list-watch Route object，然后将VIP subnet会将VIP写入411路由表，然后BIRD会将411表里面的entry宣告出去:

```
root@node:~# ip r s t 411
10.0.0.1/26 dev calia123455
```

由于Route controller运行在host network，并且TLB pod和Route controller Pod在部署时设置了affinity，这两个Pod运行在同一node，所以当数据包路由到TLB Pod所在的Node时，所有目的地址是VIP subnet的数据包都会通过411表路由到指定的接口，再通过veth pair到TLB Pod进入IPVS处理流程。

图3-4是在Production环境负载均衡的控制面和数据面的流程图，作者是Qiu Yu，第一次看到这张图是在大概3年前，当时在做UFES项目，主要的目的是让SLB承接生产环境流量。但是当时主要工作在PaaS层，也就是对SLB上的L7规则管理，虽然图上的内容看起来都懂，但是并不了解其中的细节。直到后来参与的TLB以及Feature Pool On Tess的上线才比较深入地了解其中L4/L7的细节，因为Feature Pool上线的关键是TLB，而TLB是继承于UFES。

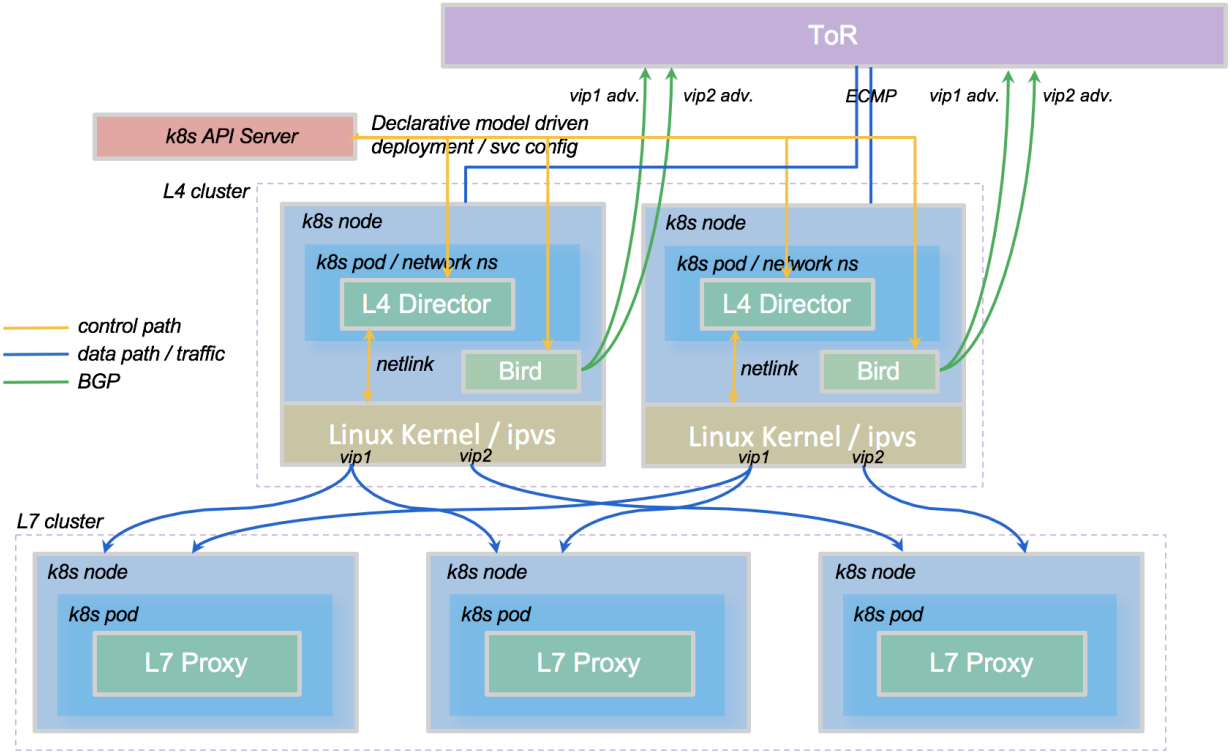

图3-4 TLB on BGP架构图

3.3 数据包流程

图3-5是OVN环境下完整的数据包流程图，涉及到容器网络，主机网络，k8s网络，OVN网络以及物理网络。

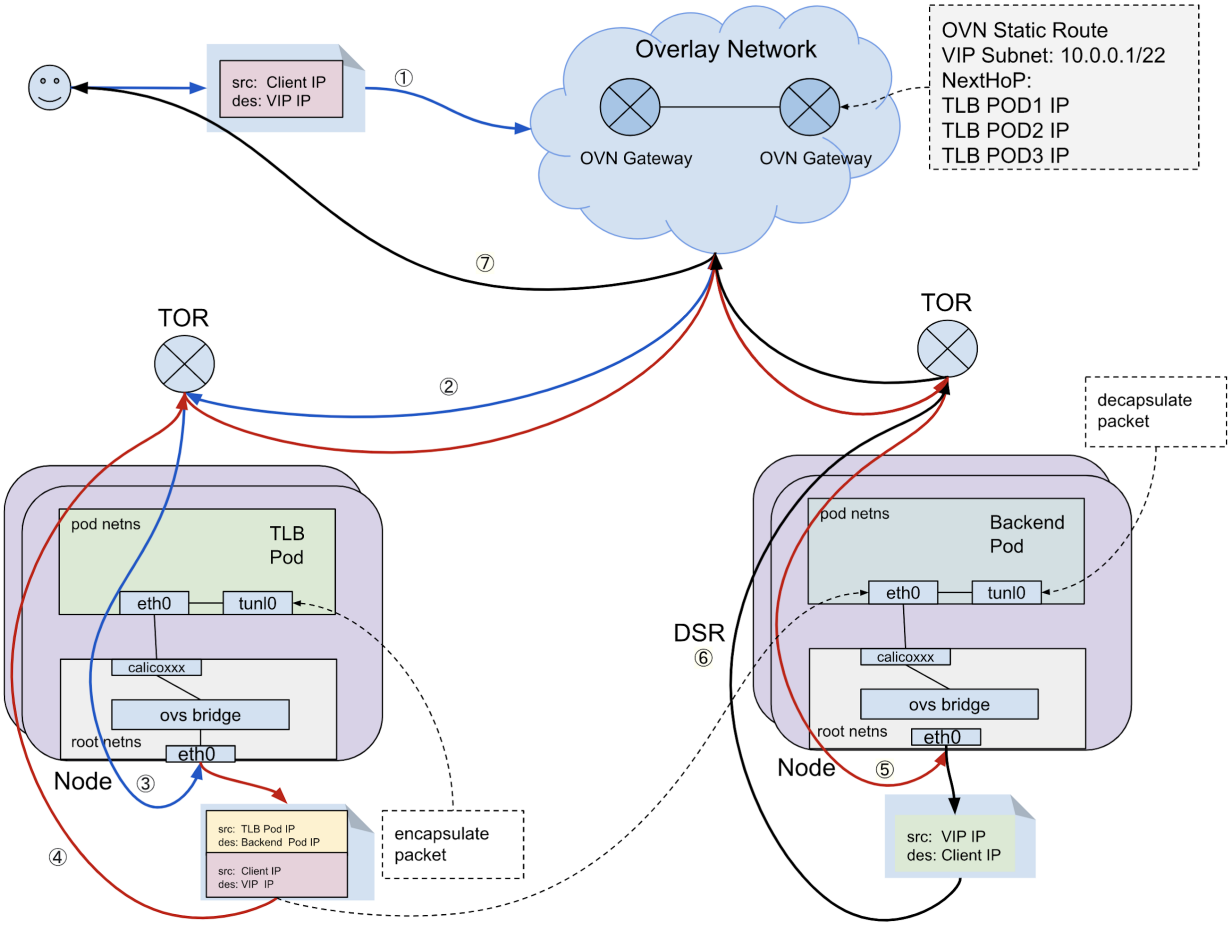

图3-5 TLB on OVN架构图

关于这张图再多说明一下，因为涉及到Node和Pod的各种网络参数的配置：

1. 所有TLB node的rp_filter(reverse path filtering)要设置为0，也就是关闭反向路由校验，否则无法正常工作，因为：
    a. 对Backend Pod来说，是在eth0收到请求，但是VIP是绑在虚拟网卡tunl0上的，也就是需要tunl0来处理，处理完之后DSR通过eth0直接返回给Client，即请求数据包进的网卡和响应数据包出的网卡不是同一个网卡，如果参数配置为1，这时候系统会判断该反向路径不是最佳路径，而直接丢弃该请求数据包。
    b. 对TLB Pod来说，也是eth0收到请求，但是VIP绑在lo设备上，原因同上。
2. TLB Node需要设置accept_local=1，要求TLB Pod允许接收从本机IP地址上发送给本机的数据包，在这里也就是需要允许接收source为VIP的数据包。因为VIP绑在TLB Pod lo设备上，在做health check的时候，内层包的source是TLB pod IP，destination是VIP，外层包的source是TLB pod IP，destination是backend podIP，backend pod返回的数据包是通过其tun0设备DSR返回的，而返回包的source就是VIP，所以需要TLB pod允许接收source为VIP的数据包，也就是要打开accept_local。

3. Overlay network环境下的数据包流程图和Flat network环境很类似，区别在于VIP subnet到TLB Pod的路由是如何配置的:要么通过OVN Route controller配置静态路由，要么通过BGP宣告。

4. 结束语

这个项目涉及到负载均衡的方方面面，通过软件的方式要达到硬件负载均衡设备的稳定性和性能, 但L4/L7的软件负载均衡已经在eBay开发和生产环境大规模应用。这里我们只介绍了L4部分, 其中也遇到了各种各样的挑战，除了前面提到过的，能想到的还有下面这些：

1. 如何确保TLB集群在做rolling upgrade的时候不产生数据面的影响？
2. 如何让后端真实服务器的Pod访问该Pod自己的Service的VIP？
3. 如何持续的监控每个TLB Goup的数据面的健康情况甚至包括所有当前TLB Group有健康的后端服务器的健康情况？
4. 如果一个VIP不通, 如何快速的定位问题？可能出问题的环节有很多，包括Tunnel/IPVS rules/OVN route/Node路由表等等。
5. VIP的hash表是TLB controller在用户空间生成，通过netlink message发送到内核模块，由于netlink消息大小有限制, 最多仅能支持一个service有大概255个后端真实服务器, 如何支持更多？

上面的这些问题基本都是我们已经解决了或者正在解决的，也希望大家看完这篇文章字后能够思考一下，就不把答案说出来了。

最后打个广告, 如果想要了解更多的k8s在eBay落地的细节的话可以去购买孟老师新出的书《Kubernetes生成化实践之路》，里面有更多的关于k8s上生产环境的实践和经验。