应用容器化后为什么性能下降这么多?

云之舞者 2023-08-19 5,671 阅读5分钟

1. 背景

随着越来越多的公司拥抱云原生，从原先的单体应用演变为微服务，应用的部署方式也从虚机变为容器化，容器编排组件k8s也成为大多数公司的标配。然而在容器化以后，我们发现应用的性能比原先在虚拟机上表现更差，这是为什么呢？。

2. 压测结果

2.1 容器化之前的表现

应用部署在虚拟机下，我们使用wrk工具进行压测，压测结果如下：

```
Running 30s test @ http://                    /message/myGet
  10 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.68ms    1.65ms  37.60ms   90.56%
    Req/Sec   716.09    127.84     1.05k    70.03%
  213932 requests in 30.02s, 88.79MB read
Requests/sec:   7126.61
Transfer/sec:      2.96MB
```


从压测结果看，$平均RT为1.68ms，qps为716/s$，我们再来看下机器的资源使用情况，cpu基本已经被打满。

```
Tasks: 148 total,   1 running, 147 sleeping,   0 stopped,   0 zombie
%Cpu(s): 59.2 us, 24.9 sy,  0.0 ni,  9.5 id,  0.0 wa,  0.0 hi,  6.4 si,  0.0 st
MiB Mem :   3919.7 total,    343.3 free,   2042.6 used,   1533.8 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   1656.3 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   8049 root      20   0 3452144 504412  16380 S 164.5  12.6   2:35.25 java
```


2.2 容器化后的表现

使用wrk工具进行压测，结果如下：

```
Running 30s test @ http://                    ge/myGet
  10 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     2.11ms    1.91ms  34.57ms   89.62%
    Req/Sec   554.51     85.23   808.00     70.07%
  165651 requests in 30.02s, 68.75MB read
Requests/sec:   5517.78
Transfer/sec:      2.29MB
```

从压测结果看，$平均RT为2.11ms，qps为554/s$，我们再来看下机器的资源使用情况，cpu基本已经被打满。

```
top - 17:56:11 up 47 min,  1 user,  load average: 1.67, 1.82, 1.17
Tasks: 160 total,   1 running, 159 sleeping,   0 stopped,   0 zombie
%Cpu(s): 55.0 us, 22.8 sy,  0.0 ni,  9.1 id,  0.0 wa,  0.0 hi, 13.1 si,  0.0 st
MiB Mem :   3919.7 total,    684.9 free,   1618.2 used,   1616.7 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2087.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   3330 root      20   0 3299132 213768  16448 S 161.8   5.3   5:34.73 java
```


2.3 性能对比结果

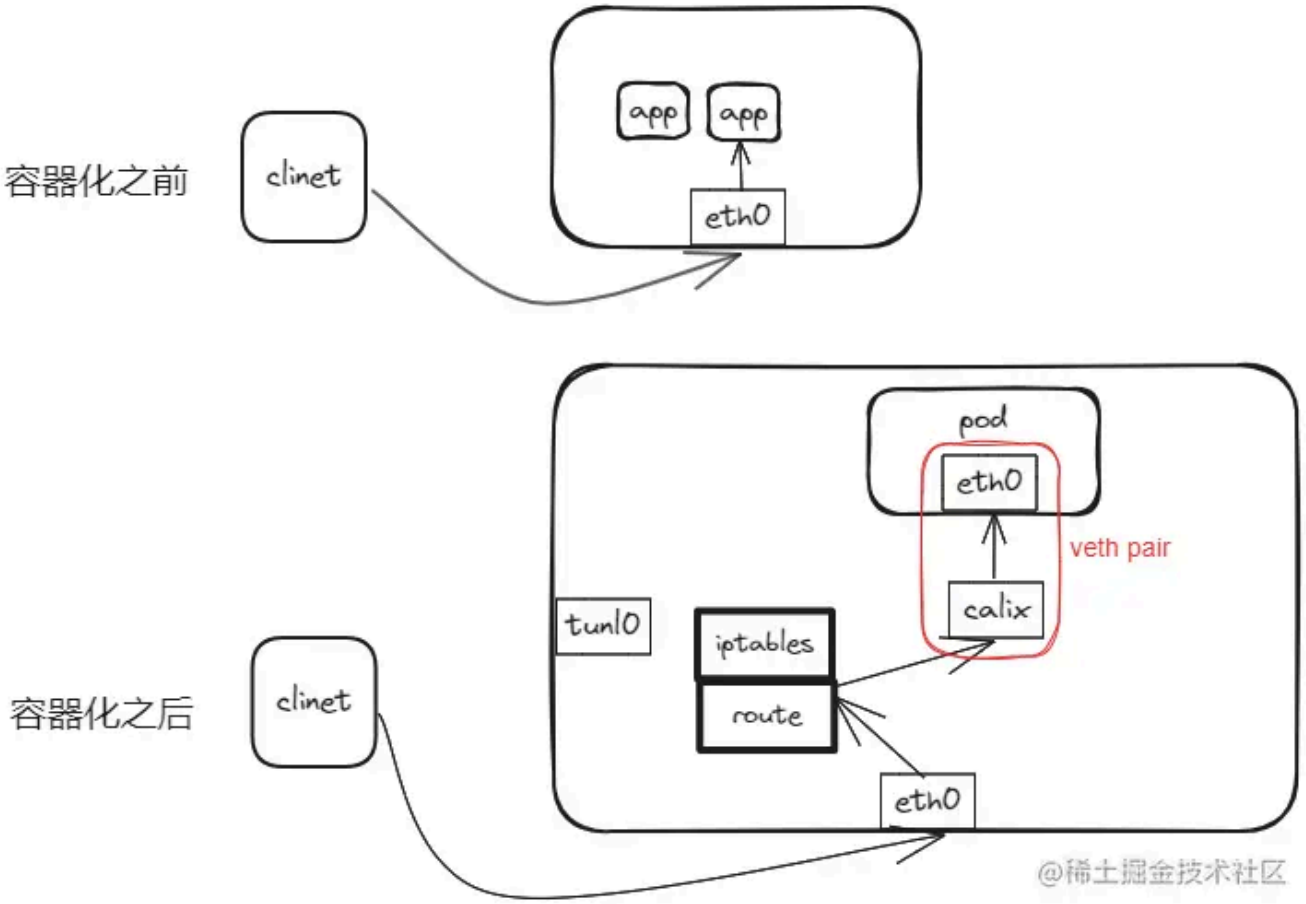
性能对比	虚拟机	容器
RT	1.68ms	2.11ms
QPS	716/s	554/s

> **总体性能下降：RT（25%）、QPS（29%）**

3. 原因分析

3.1 架构差异

由于应用在容器化后整体架构的不同、访问路径的不同，将可能导致应用容器化后性能的下降，于是我们先来分析下两者架构的区别。我们使用k8s作为容器编排基础设施，网络插件使用calico的ipip模式，整体架构如下所示。

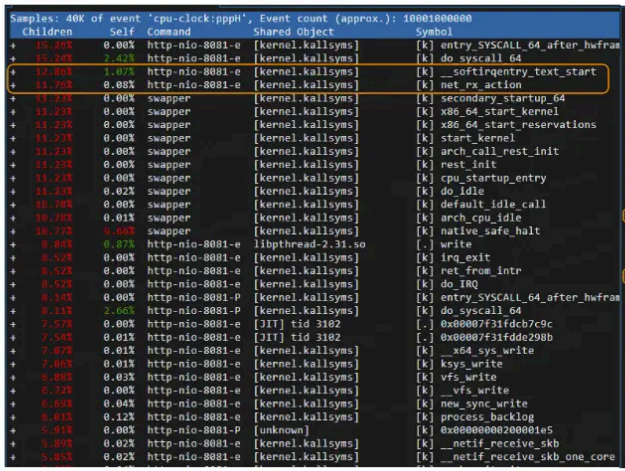


这里需要说明，虽然使用calico的ipip模式，由于pod的访问为service的nodePort模式，所以不会走tunl0网卡，而是从eth0经过iptables后，通过路由到calico的calixxx接口，最后到pod。

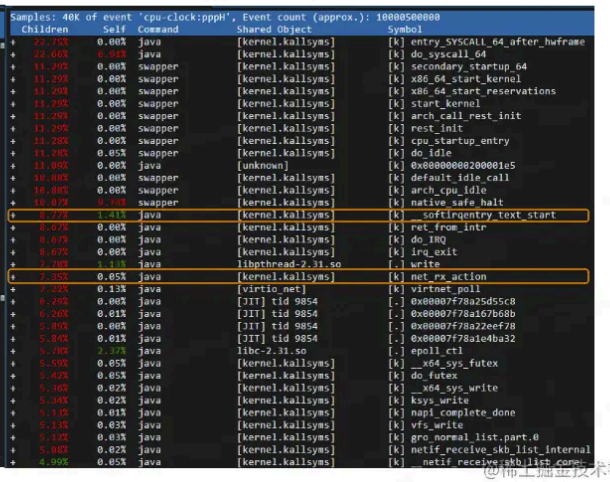
3.2性能分析

在上面压测结果的图中，我们容器化后，cpu的软中断si使用率明显高于原先虚拟机的si使用率，所以我们使用perf继续分析下热点函数。

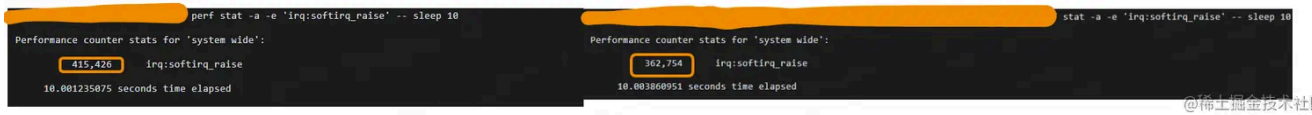
K8S宿主机的软中断



虚拟机的软中断



为了进一步验证是否是软中断的影响，我们使用perf进一步统计软中断的次数。



> **我们发现容器化后比原先软中断多了14%，到这里，我们能基本得出结论，应用容器化以后，需要更多的软中断的网络通信导致了性能的下降。**

3.3 软中断原因

由于容器化后，容器和宿主机在不同的网络namespace,数据需要在容器的namespace和host namespace之间相互通信,使得不同namespace的两个虚拟设备相互通信的一对设备为veth pair,可以使用ip link命令创建，对应上面架构图中红色框内的两个设备，也就是calico创建的calixxx和容器内的eth0。我们再来看下veth设备发送数据的过程

scss 代码解读 复制代码

```
static netdev_tx_t veth_xmit(struct sk_buff *skb, struct net_device *dev)
{
...
    if (likely(veth_forward_skb(rcv, skb, rq, rcv_xdp)
...
}

static int veth_forward_skb(struct net_device *dev, struct sk_buff *skb,
                      struct veth_rq *rq, bool xdp)
{
  return __dev_forward_skb(dev, skb) ?: xdp ?
          veth_xdp_rx(rq, skb) :
          netif_rx(skb);//中断处理
}


/* Called with irq disabled */
static inline void ____napi_schedule(struct softnet_data *sd,
                                struct napi_struct *napi)
{
  list_add_tail(&napi->poll_list, &sd->poll_list);
  //发起软中断
```

```
23     __raise_softirq_irqoff(NET_RX_SOFTIRQ);
24 }
```

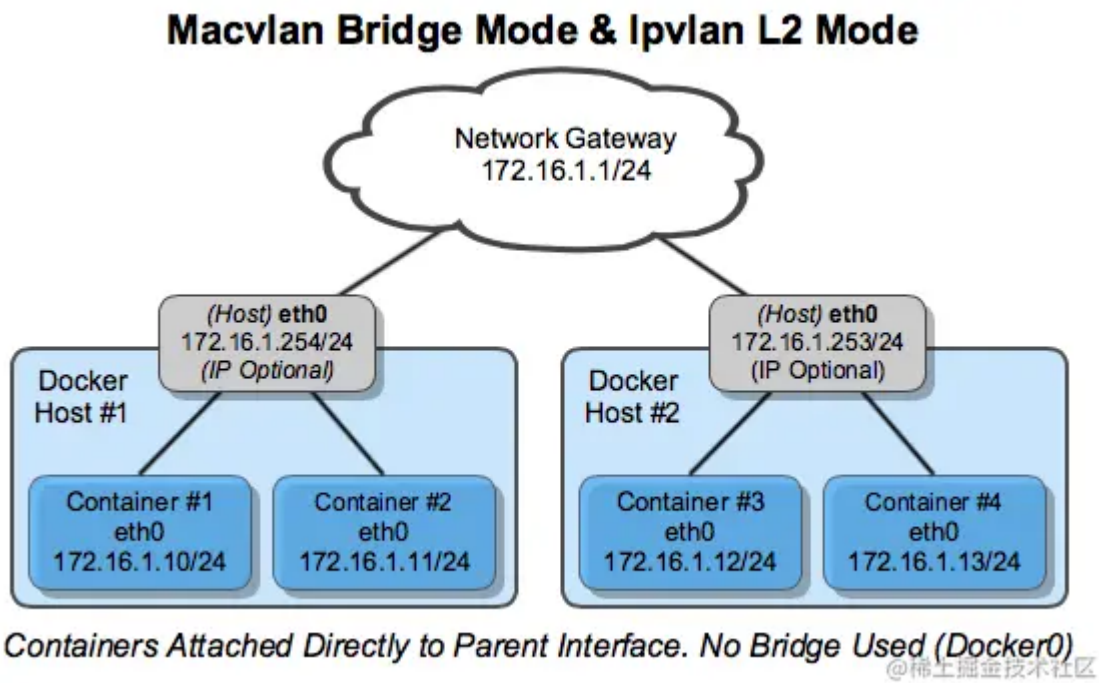
通过虚拟的veth发送数据和真实的物理接口没有区别，都需要完整的走一遍内核协议栈，从代码分析调用链路为veth_xmit -> veth_forward_skb -> netif_rx -> __raise_softirq_irqoff，**veth的数据发送接收最后会使用软中断的方式，这也刚好解释了容器化以后为什么会有更多的软中断，也找到了性能下降的原因。**

4. 优化策略

原来我们使用calico的ipip模式，它是一种overlay的网络方案，容器和宿主机之间通过veth pair进行通信存在性能损耗，虽然calico可以通过BGP，在三层通过路由的方式实现underlay的网络通信，但还是不能避免veth pari带来的性能损耗，针对性能敏感的应用，那么有没有其他underly的网络方案来保障网络性能呢？那就是macvlan/ipvlan模式，我们以ipvlan为例稍微展开讲讲。

4.1 ipvlan L2 模式

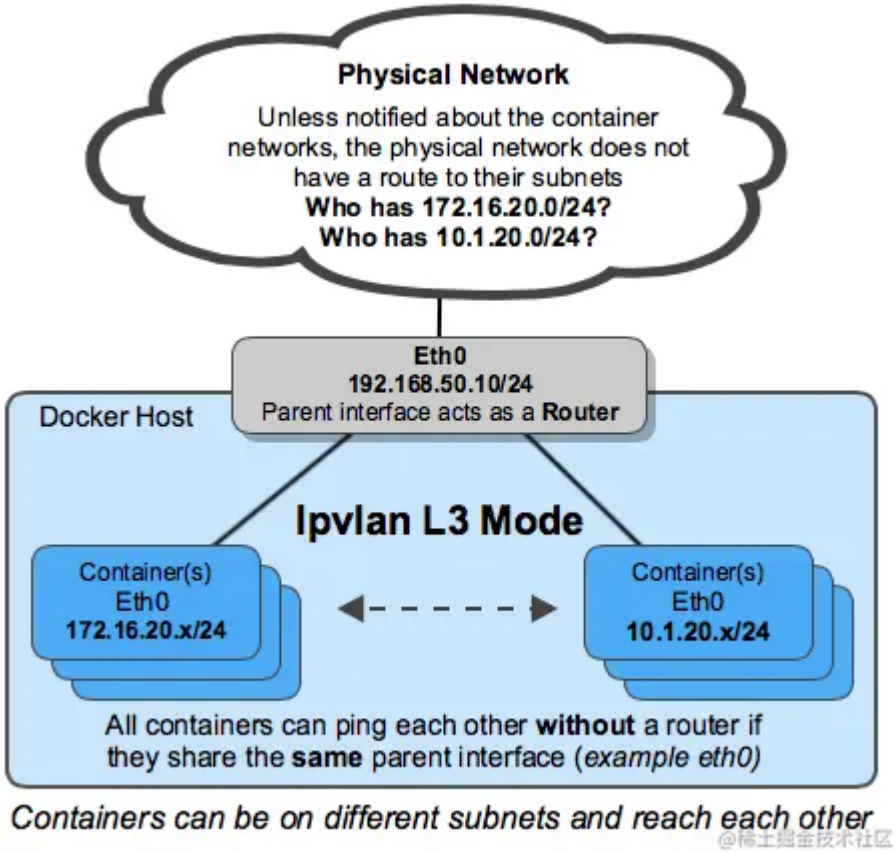
IPvlan和传统Linux网桥隔离的技术方案有些区别，它直接使用linux以太网的接口或子接口相关联，这样使得整个发送路径变短，并且没有软中断的影响，从而性能更优。如下图所示：



上图是ipvlan L2模式的通信模型，可以看出container直接使用host eth0发送数据，可以有效减小发送路径，提升发送性能。

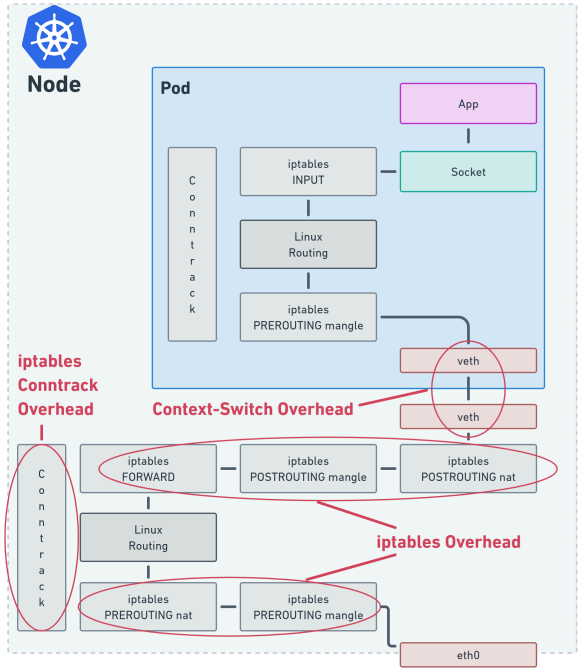
4.2 ipvlan L3 模式

ipvlan L3模式，宿主机充当路由器的角色，实现容器跨网段的访问，如下图所示：

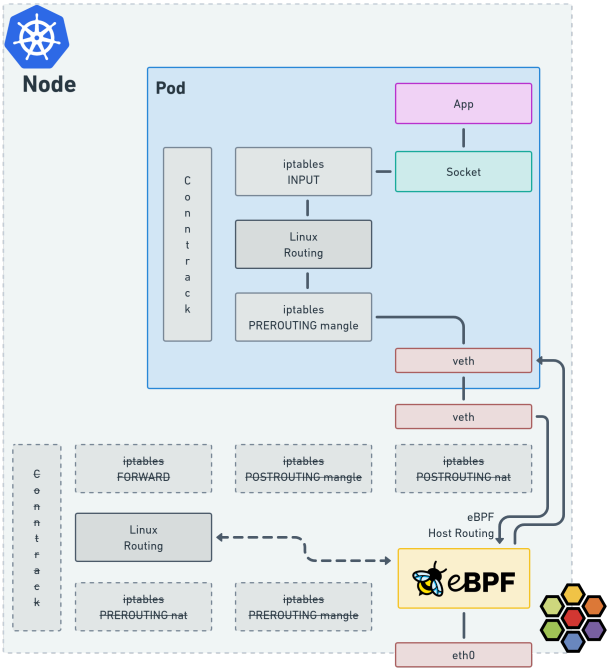


4.3 Cilium

除了使用macvlan/ipvlan提升网络性能外，我们还可以使用Cilium来提升性能，Cilium为云原生提供了网络、可观测性、网络安全等解决方案，同时它是一个高性能的网络CNI插件，高性能的原因是优化了数据发送的路径，减少了iptables开销，如下图所示：

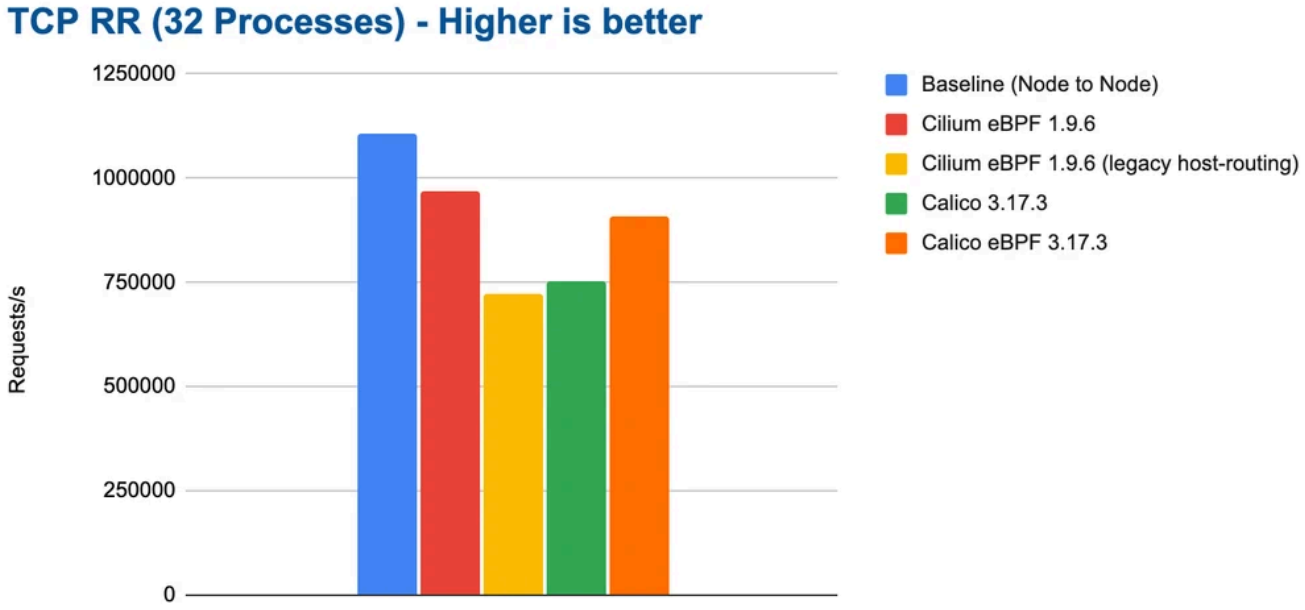


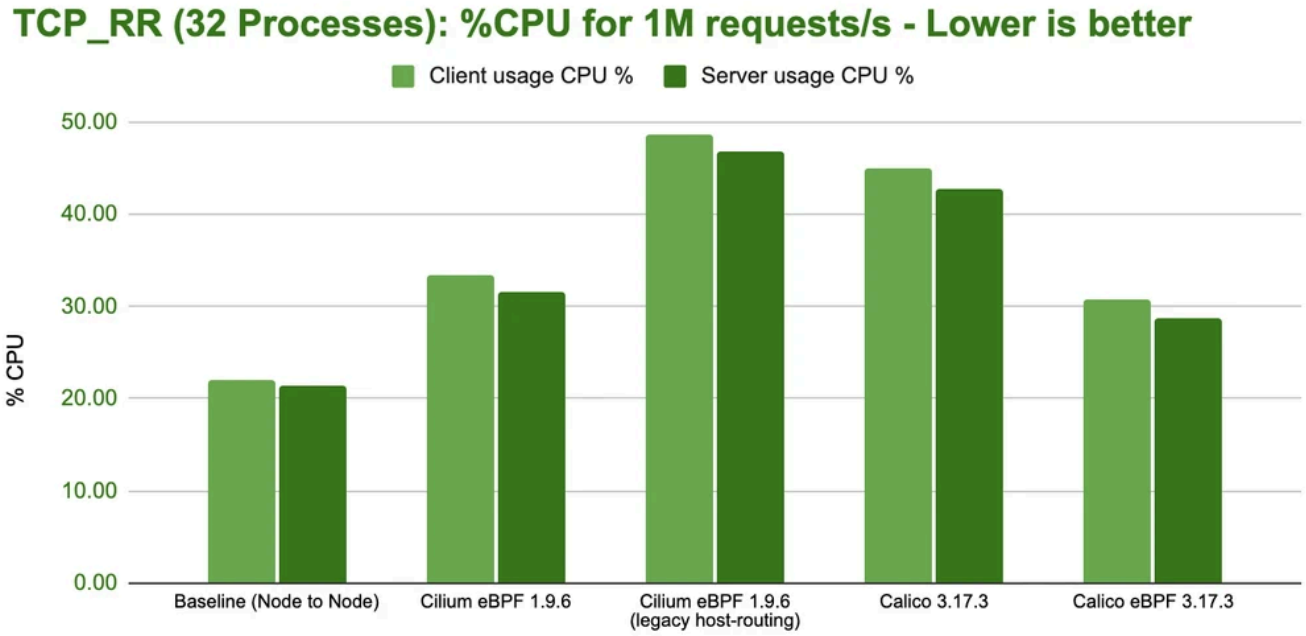
Standard Container Networking



Cilium eBPF Container Networking

虽然calico也支持ebpf，但是通过benchmark的对比，Cilium性能更好，高性能名副其实，接下来我们来看看官网公布的一些benchmark的数据，我们只取其中一部分来分析，如下图:





无论从QPS和CPU使用率上Cilium都拥有更强的性能。

5. 总结

容器化带来了敏捷、效率、资源利用率的提升、环境的一致性等等优点的同时，也使得整体的系统复杂度提升一个等级，特别是网络问题，容器化使得整个数据发送路径变长，排查难度增大。不过现在很多网络插件也提供了很多可观测性的能力，帮助我们定位问题。

我们还是需要从实际业务场景出发，针对容器化后性能、安全、问题排查难度增大等问题，通过优化架构，增强基础设施建设才能让我们在云原生的路上越走越远。

最后，感谢大家观看，也希望和我讨论云原生过程中遇到的问题。

5. 参考资料

docs.docker.com/network/dri...

cilium.io/blog/2021/0...

标签： 云原生 后端