[译] 为容器时代设计的高级 eBPF 内核特性 (FOSDEM, 2021)

Published at 2021-02-13 | Last Update 2021-02-13

译者序

本文翻译自 2021 年 Daniel Borkmann 在 FOSDEM 的一篇分享： Advanced eBPF kernel features for the container age。

内容是 2019 和 2020 两次 LPC 大会分享的延续，

1. 利用 eBPF 支撑大规模 K8s Service (LPC, 2019)
2. 基于 BPF/XDP 实现 K8s Service 负载均衡 (LPC, 2020)

翻译已获得 Daniel 授权。

由于译者水平有限，本文不免存在遗漏或错误之处。如有疑问，请查阅原文。

以下是译文。

我来自 Isovalent（Cilium 背后的公司），是内核 **eBPF 的维护者之一**（co-maintainer）。 今天给大家分享一些 Cilium（1.9）和 eBPF 的最新进展。

1 纠正一些关于 eBPF 的错误理解

首先我想纠正一些关于 eBPF 的错误理解与不实描述。

1.1 eBPF 是什么？通用目的引擎、最小指令集架构

之前我们讨论 eBPF 时大都集中在**网络**（networking）和**跟踪**（tracing）领域， 最近可能将范围扩大到了**安全**（security）领域 —— 但我想说的是：eBPF 是一个 **通用目的执行引擎**（general purpose execution engine）。

换句话说，eBPF 是一个**最小指令集架构**（a minimal instruction set architecture），在设计时**两个主要考虑**：

1. 将 eBPF 指令映射到平台原生指令时开销尽可能小 —— 尤其是 x86-64 和 arm64 平台，因此我们针对这两种架构进行了很多优化，使程序运行地尽可能快。

2. 内核在加载 eBPF 代码时要能验证代码的安全性 —— 这也是为什么我们一 直将其限制为一个最小指令集，因为这样才能确保它是可验证的（进而是安全的）。

   很多人像我一样，在过去很长时间都在开发**内核模块**（kernel module）。 但**内核模块中引入 bug 是一件极度危险的事情 —— 它会导致内核 crash。** 此时 **BPF 的优势**就体现出来了：校验器（verifier）会检查是否有越界内存访问 、无限循环等问题，一旦发现就会拒绝加载，而非将这些问题留到运行时（导致 内核 crash 等破坏系统稳定性的行为）。

   所以出于安全方面的原因，很多内核开发者开始用 eBPF 编写程序，而不再使用传统的内核模块方式。

eBPF 提供的是基本功能模块（building blocks）和程序附着点（attachment points）。 我们可以编写 eBPF 程序来 attach 到这些 hook 点完成某些高级功能。

1.2 eBPF 不是什么？通用目的虚拟机、全功能通用指令集

BPF 是一个**通用目的虚拟机**（general purpose virtual machine）吗？ 这是人们经常问的一个问题。不是 —— BPF 并不打算模拟完整的计算机，它只是一个最小指令集和通用目的执行引擎。

BPF 是一个**全功能通用指令集**吗？也不是。它必须保持最小，这样才能保证可验证和安全。

1.3 eBPF 与 C

- BPF 在设计上有意采用了 C 调用约定（calling convention）。

  由于内核是用 C 写的，BPF 要与内核**高效**地交互，因此也采用了 C。 当需要从 BPF 程序中调用所谓的 BPF helper 甚至是真正的内核函数时，这会非常方 便；另外，用户态和内核态之间共享数据（BPF maps）必须越快越好。

- 目前 ~150 BPF helpers，~30 BPF maps。

- BPF C 与普通 C 差异有多大?

  BPF 校验器可能最清楚地见证了近几年 BPF C 的发展历史。 现在我们有了 BPF-to-BPF 函数调用、有限循环（bounded loops）、全局变量、静态 链接（static linking）、BTF（BPF Type Format，在 **tracing 场景**尤其有用；其 他方面也有用到，使内核成为可自描述的 self-descriptive）、 单个 BPF 程序的最大指令数（instructions/program）从原来的 4096 条放大到了 100 万条。

以上可以看到，BPF 已经具备了很多的基础模块和功能。基于这项功能，我们能解决许多更 加有趣的生产问题。其中之一 —— 也是我接下来想讨论的 —— 就是 Cilium。 我们来看它是如何使用 eBPF 以及用来解决什么问题的。

2 基于 eBPF 的云原生项目：Cilium 简介

Cilium 是一个基于 eBPF 技术的云原生网络、安全和可观测项目。下面简单列举一些特性。

2.1 网络功能

- 通过 CNI 集成到 Kubernetes。
- 基于 BPF 实现了 Pod-to-Pod **数据转发路径**（datapath）。
- 支持**直接路由**（direct routing）、overlay、**cloud-provider native**（例如 AWS）等网 络模式。
- 支持 IPv4、IPv6、NAT46。
- 支持**多集群路由**（multi-cluster routing）。

> 更多信息，可参考 [3,4,5,6]。译注。

2.2 负载均衡

- 实现了**高度可扩展的 L3-L4（XDP）负载均衡。**
- 能完全替代 kube-proxy，提供 K8s Service 功能。
- 支持**多集群**（multi-cluster）。

> 详见 [1,2]，译注。

2.3 网络安全

- 支持**基于身份的（identity-based）L3-L7 网络安全**
- API-aware 安全（HTTP、gRPC 等）
- DNS-aware
- 透明加密

2.4 可观测性

- Metrics：通过 BPF 收集观测数据和 tracing 数据并导出；包含网络、DNS、安全、延迟、HTTP 等方面的数据。

- 提供 flow 级别的日志（flow log），并支持设置聚合粒度（datapath aggregation）。

2.5 Servicemesh

- 注入 sidecar 时开销最小（minimized overhead）。
- 与 Istio 集成。

> 用到了sockmap/redirection 做 socket 重定向，可参考 [7]，译注。

2.6 小结

接下来介绍几个最近的新特性（我们在 Cilium 1.9 及内核方面的最新工作）。

3 深入剖析（一）：基于 BPF/XDP/Maglev 实现 K8s Service 负载均衡

首先看基于 XDP/BPF 和 Maglev 算法实现的 K8s Service 负载均衡。

3.1 原理

K8s Service 模型中，入口和后端实例位于同一组节点，即**每个节点既用于部署容器， 又负责 Service 负载均衡的实现**（co-location of service load balancer with regular user workloads on every node）的。换句话说，每个 node 上都实现了 Service 负载均衡的功能。在原生 K8s 中，实现这个功能的组件是 kube-proxy。

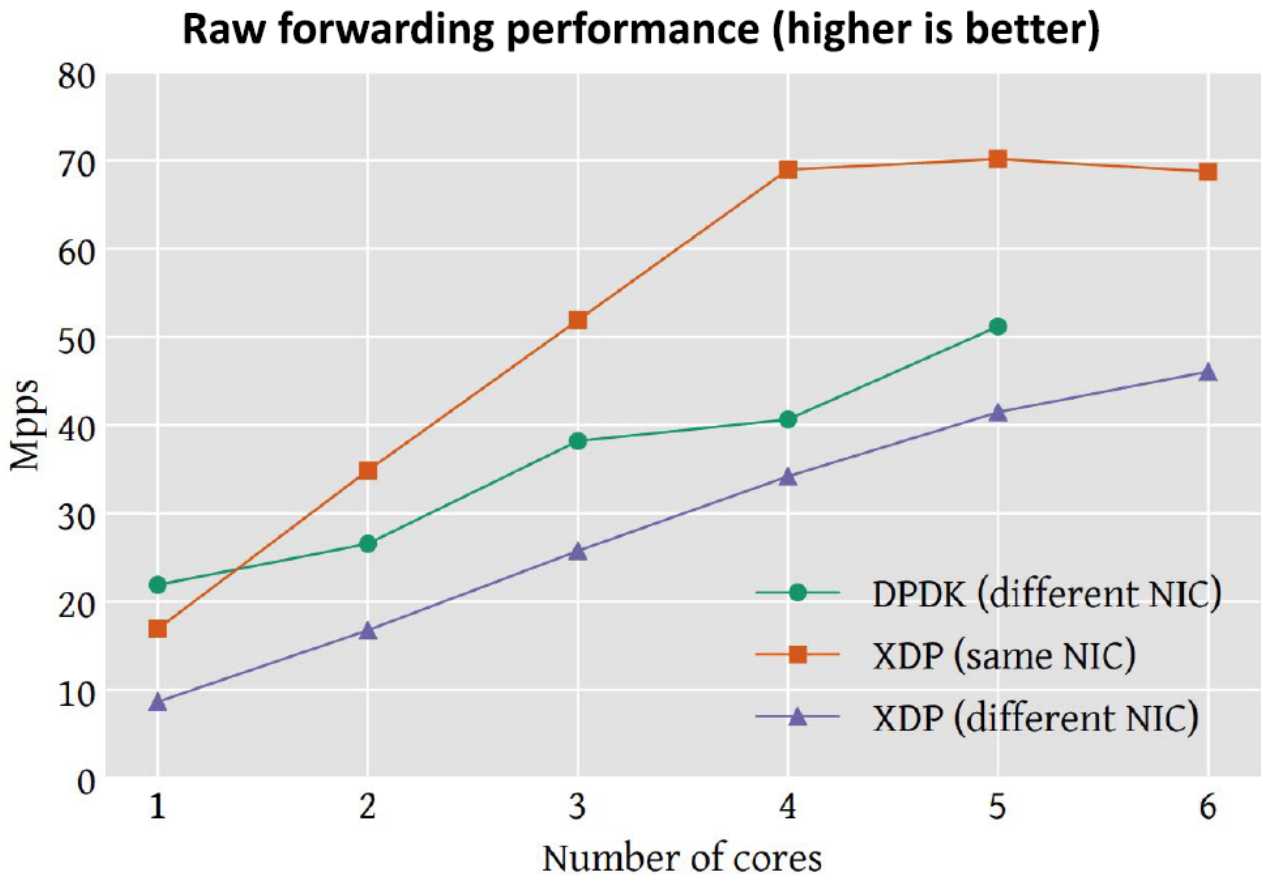
> Service 是对**服务**的抽象，服务包含一个入口和多个后端实例，因此涉及到负载均衡 ，即如何将请求分发到不同后端。但在模型及实现上，入口（负载均衡器）和后端实例 可以是分开部署的（例如负载均衡器部署在独立设备上），也可以部署在一起（负载均 衡器直接部署在运行实例的宿主机上）。 更多关于 kube-proxy 的设计及实现，可参考 [8]。
>
> 译注。

但 kube-proxy 的问题之一是性能很差。Cilium 所做的一项工作就是**基于 BPF/XDP 实现了 kube-proxy 的功能**（针对南北向流量），显著减少了 CPU 使用量，并达到（某些场景甚至 超过了）DPDK 的性能。

XDP 运行在网络驱动层，因此能**直接在驱动层将收到的包再发出去**， 这是软件栈中最早能够处理数据包的位置（此时 skb 都还没有创建），因此性能非常高。

3.2 转发性能对比（XDP vs DPDK）

以下测试结果基于 Cilium 1.9 及原生 Linux 内核中的 XDP 驱动。

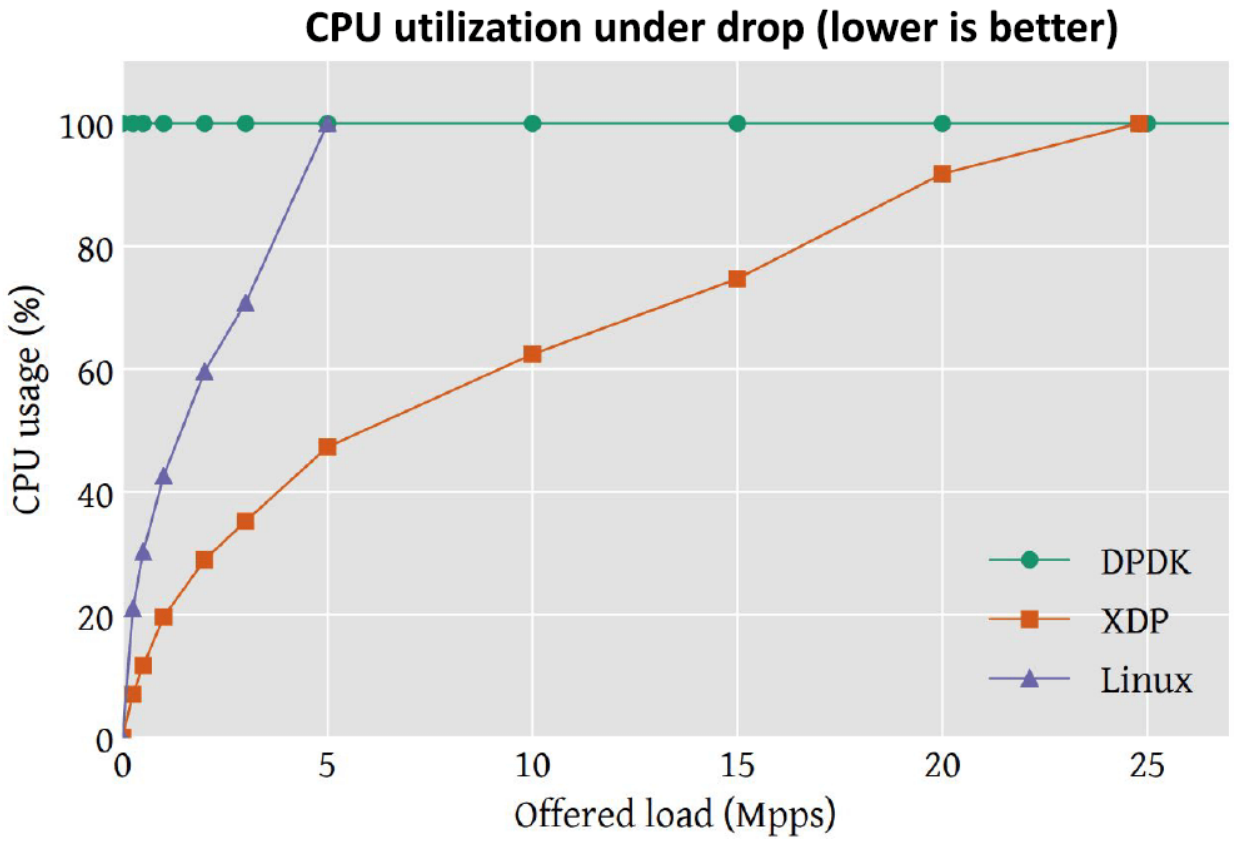


这里测试的是后端 pod 在另一台 node 的场景。三条线分别表示：

1. DPDK (different NIC): DPDK 从一个网卡收包，然后将包从**另一个网卡**发出去。
2. XDP (different NIC): Cilium/XDP 从一个网卡收包，然后将包从**另一个网卡**发出去。
3. XDP (same NIC): Cilium/XDP 从一个网卡收包，然后直接将包从**当前网卡**发出去。

可以看出，XDP (different NIC) 的性能已经接近 DPDK；但 **same NIC 组，四核 及以上的性能已经远超过 DPDK。**

再看 CPU 消耗：



1. DPDK 是 busy-poll 模型，因此 CPU 一直是 100%，即需要独占一个或多个 CPU。
2. 相比之下，XDP 的 CPU 消耗基本上是线性的，远小于 DPDK 的消耗。
3. 图中还给出了 Linux kernel 收发包的 CPU 消耗作为 benchmark。可以 看到在 5Mpps 时它就已经饱和了，转发性能无法进一步提高。

3.2 Source IP 处理：SNAT/DSR

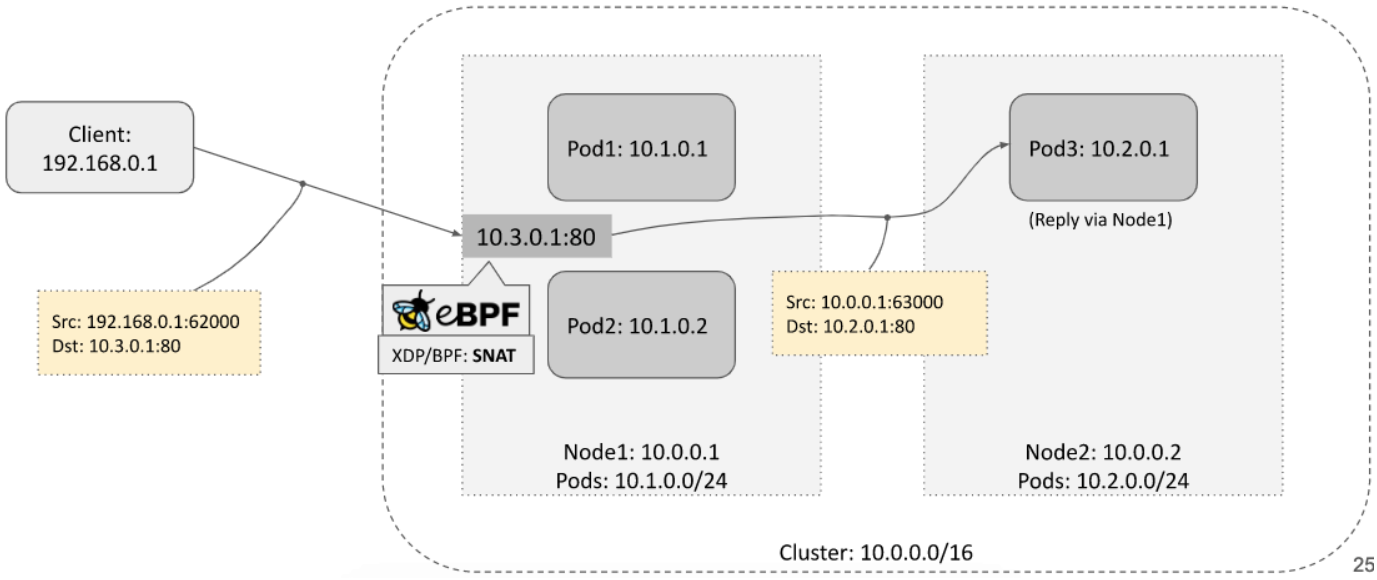
简要重温 Service LB 的 NAT/DSR 模式。

假设 Service 请求到达了 node1，而这个 Service 的 backend pod 在 node2。

SNAT

这种模式下，node1 **对源 IP 地址（Source IP）的处理**：

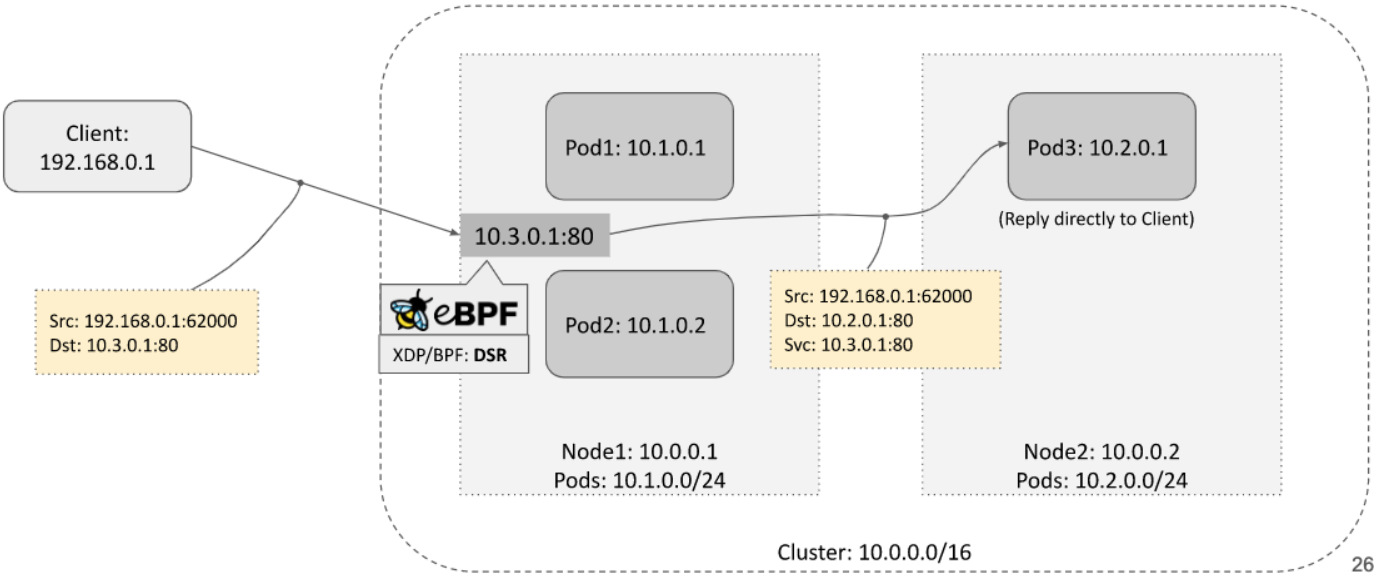
- 请求：做源地址转换（SNAT）
- 响应：对源地址做反向转换（rev-SNAT）



响应包需要先回到 node1，再回到客户端。来回路径是相同的。

DSR

这种模式下，node1 不需要做 SNAT：



响应包也无需回到 node1，而是直接从 node2 回到客户端。

> 详见 [2,10]，译注。

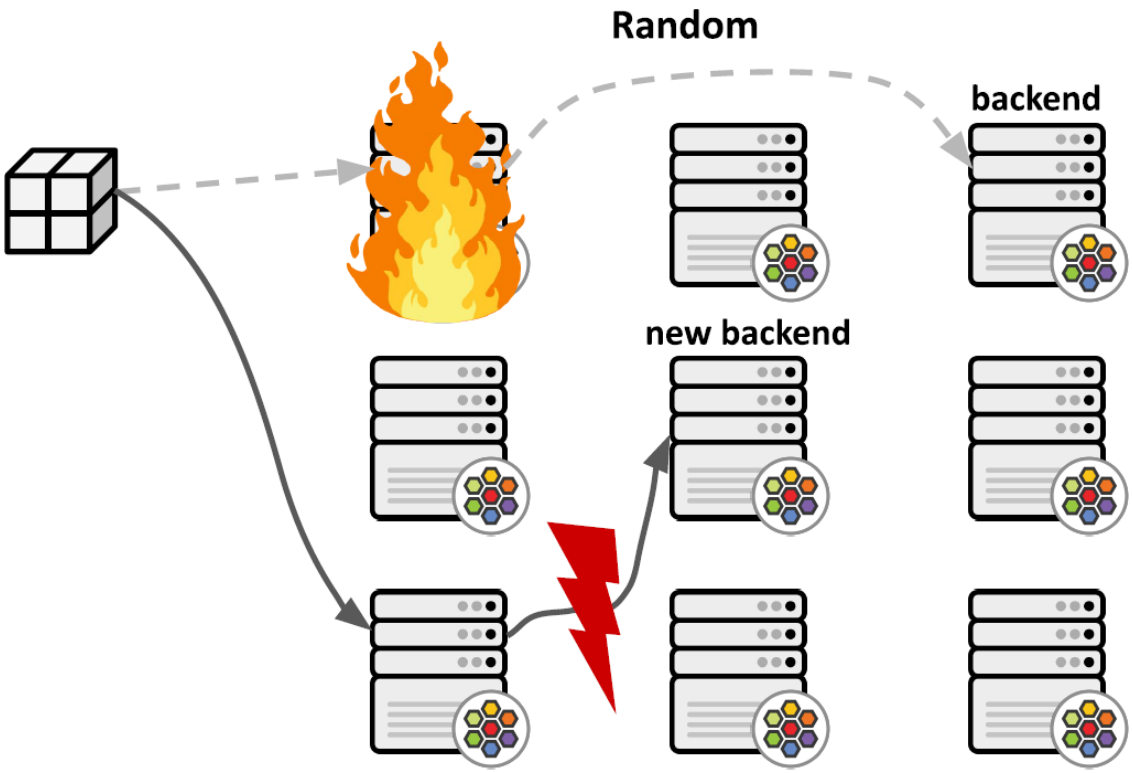
3.3 Destination IP 处理 —— Maglev 负载均衡

上面小节讨论的是对源 IP 地址的处理，接下来看对目的 IP 地址的处理，即，如何选择后 端 pod。

kube-proxy 选择后端 pod 时都是随机的（select a random backend）。 **Cilium 1.9 之前也是这个行为**，并将这个状态存储到本节点的 BPF conntrack table。

> conntrack，连接跟踪，可参考 [9]，译注。

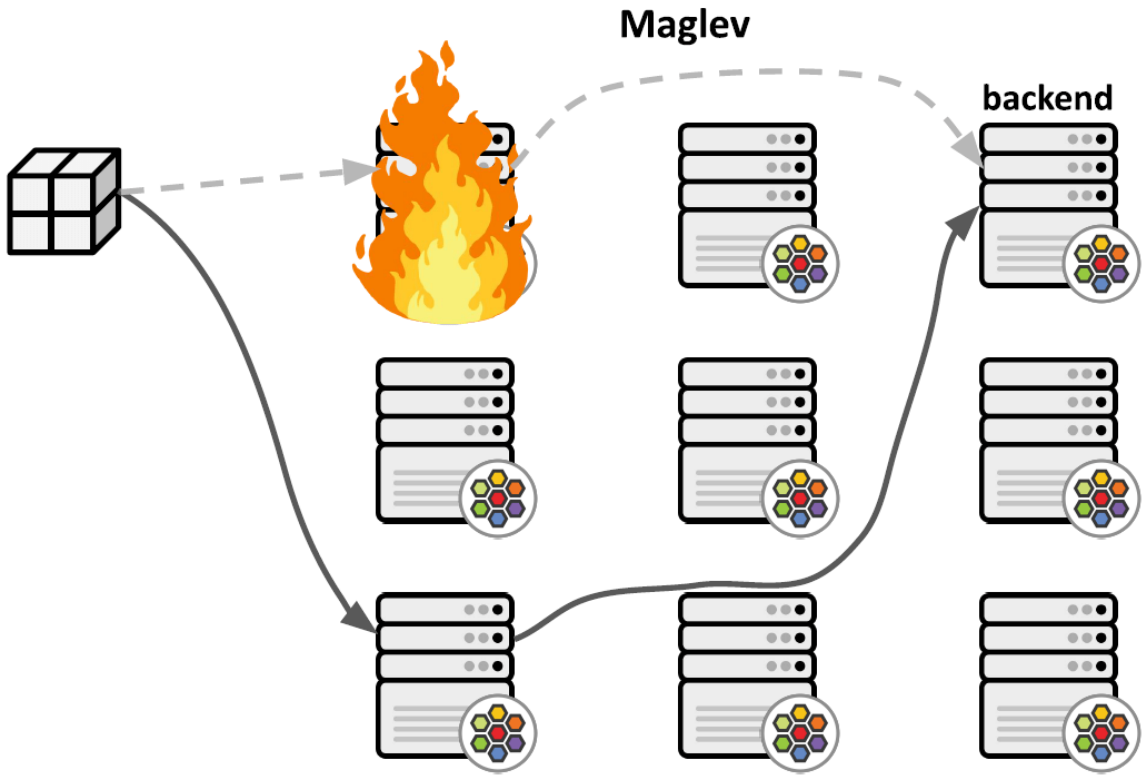
但这里存在一个问题是，如下图所示，



1. 当负责处理某个 Service 的 node 挂掉时，这个连接接下来的包会被转发到另一个 node （例如通过 ECMP）—— **此时还没有问题**；
2. 但新 node 没有之前的 service -> backend1 上下文信息（因为这些信息 只存在于 node1 的 conntrack table），因此会随机再选择一个 backend， 假设这里选择了 backend2，然后建立新的映射 service -> backdend2，并将包转 发过去 —— **这里也没问题，能发送过去**，但接下来 ——
3. 包到达 backend2 时会被拒绝，因为 backend2 上并不存在这条连接。

总结起来就是：一旦 node 挂掉，所有经过这台 node 的 Service 连接会全部中断。

为了解决这个问题，我们在 1.9 中引入了一致性哈希：在 Cilium 中实现了 **Google Maglev 一致性哈希算法**，用于 BPF/XDP Service 负载均衡。简单来说， 对于同一条 Service 连接，该算法能保证哈希到相同的后端。因此一台 node 挂掉后 ，只要接下来的包能到达其他 node，该 node 就能保证将包转发到原 来的 backend pod，因此**连接不会中断**，如下图所示：



在实现上，

- 相比随机方式，这种算法需要使用更多的内存；为了提高内存使用效率， 使用了动态大小的 map-in-map 数据结构：
  i. **外层**：service map
  ii. **内层**：per-service maglev map
- 根据 tuple 信息做两级查找：
  i. 第一次查全局的 service map
  ii. 然后再查找 per-service 的 Maglev map，这个表是**由用户态的 cilium-agent 来操作和更新**的。

3.4 内核 eBPF 改动

为实现以上功能，我们对内核 eBPF 的 map-in-map 做了增强，允许内层 map 的大小是动态的，见 bpf: Allow for map-in-map with dynamic inner array map entries。

如果没有这个扩展，那内层 map 只能是固定大小，而很多 Service 可能只有少量后端， 导致 map 的大部分空间都是用不到的，非常浪费内存。

4 深入剖析（二）：基于 BPF 的低延迟转发路径

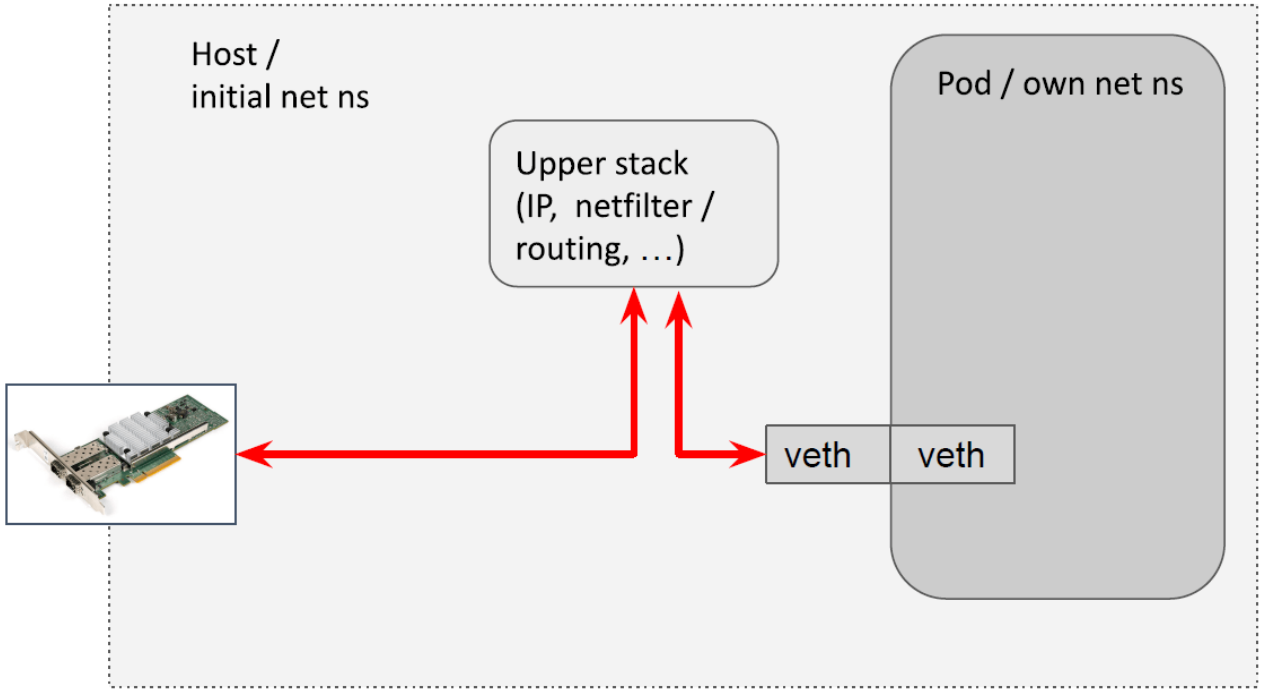
(fast-path)

这里主要是介绍绕过宿主机内核协议栈，直接将包从网卡或容器 redirect 到另一个端（容器或网卡）。

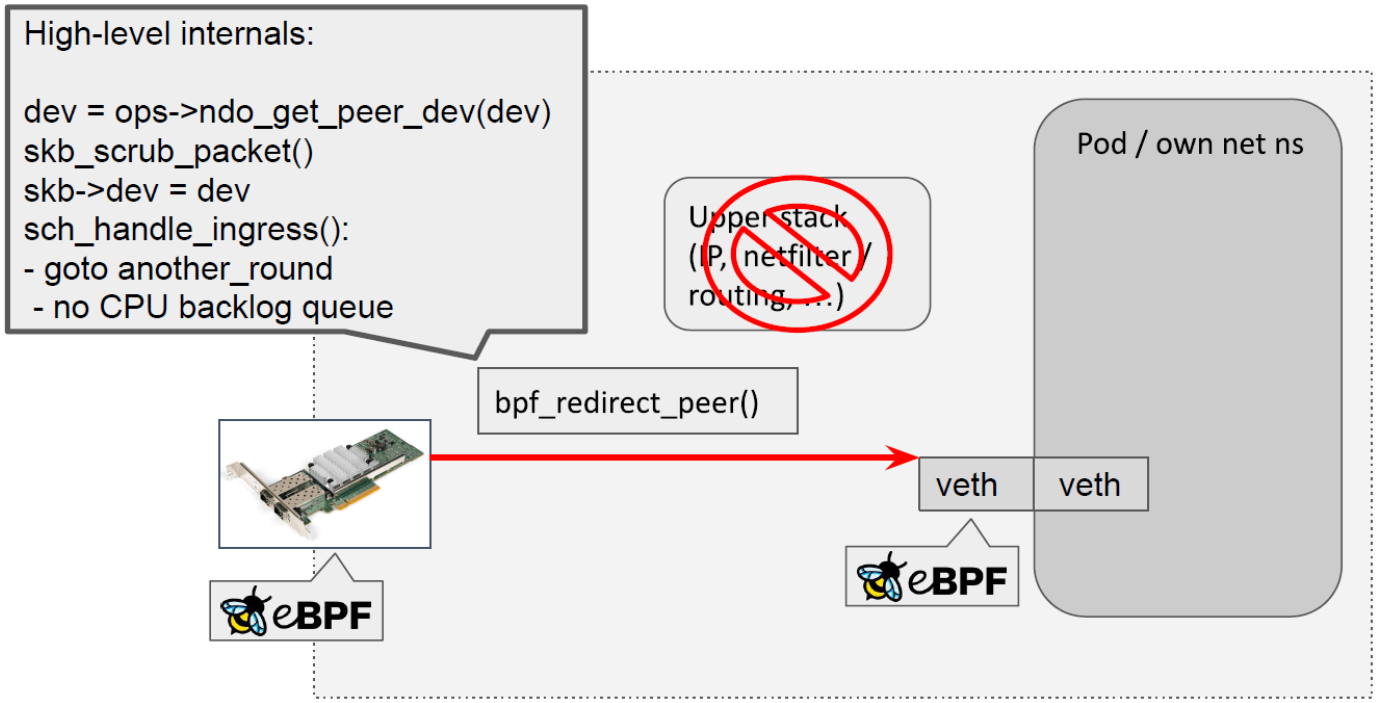
> 这些内容在去年的分享中有更详细介绍，见 基于 BPF/XDP 实现 K8s Service 负载均衡 (LPC, 2020)。 译注。

4.1 进出宿主机的容器流量（`host <-> pod`）

原来需要穿越宿主机的内核协议栈：

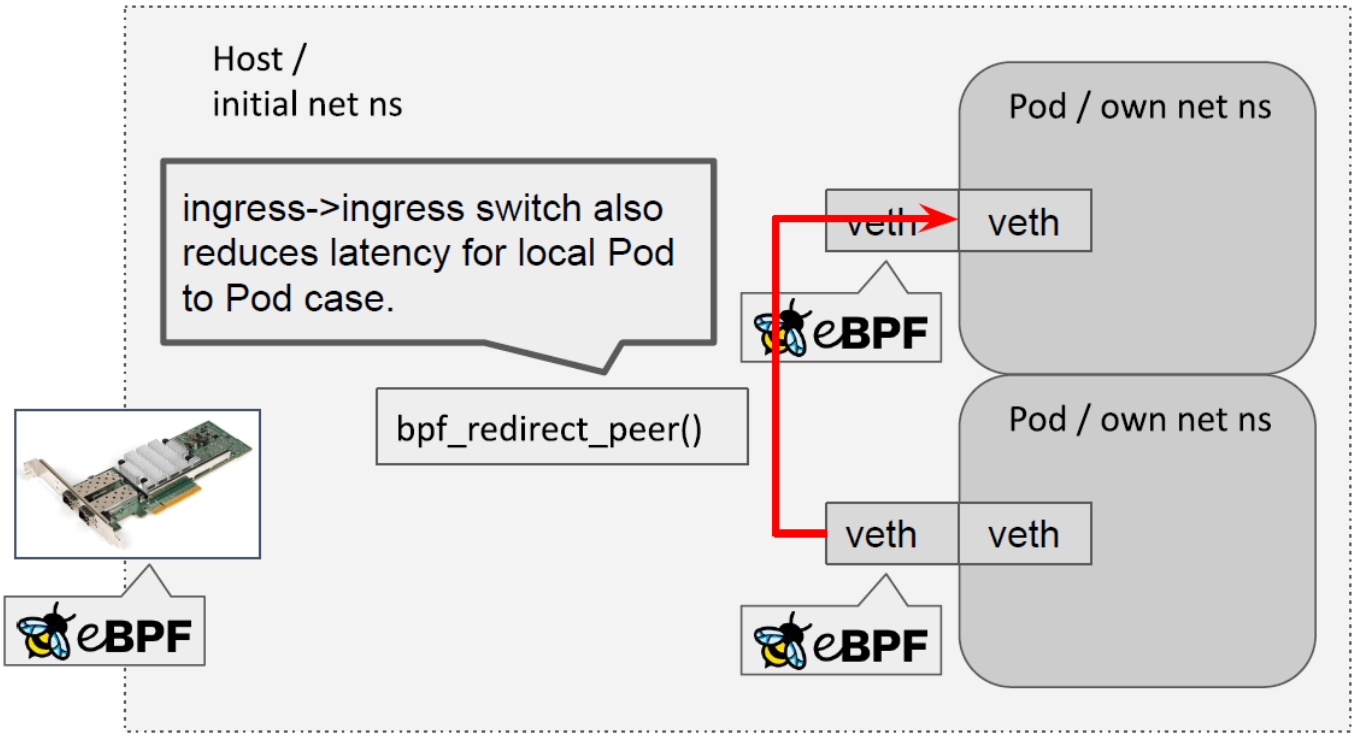


现在绕过了宿主机内核协议栈，直接将包从网卡重定向到容器网络设备：



4.2 同宿主机的容器流量（`pod <-> pod`）

与前面类似，绕过宿主机内核协议栈，直接重定向：

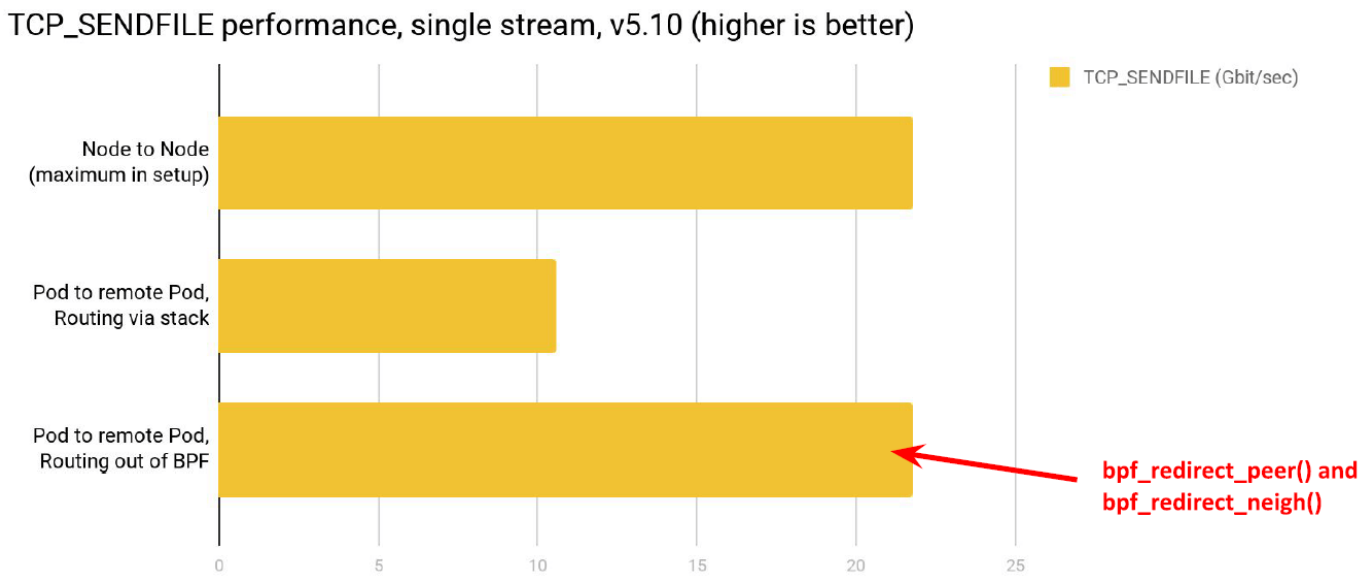


4.3 内核 eBPF 改动

1. bpf: Add redirect_peer helper
2. bpf: Add redirect_neigh helper as redirect drop-in

4.4 性能对比

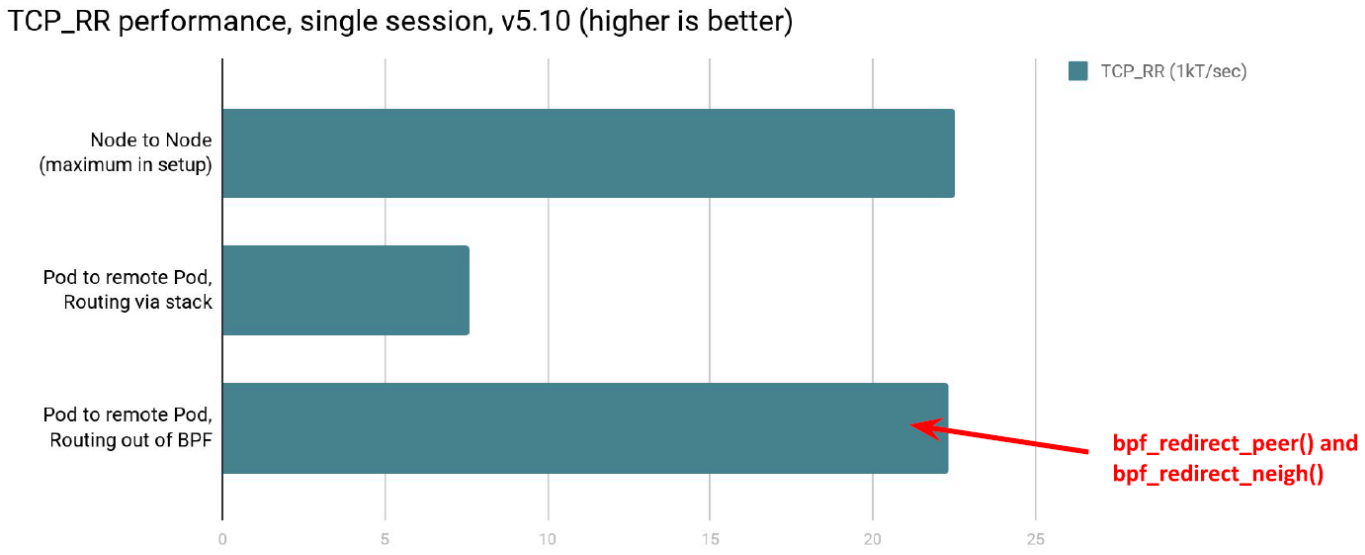
用 netperf 做了一些性能测试。



TCP_SENDFILE 性能，三组测试：

- node-to-node 转发性能：作为 benchmark 参考。
- pod-to-remote-pod，没有使用 fast-path：传统的经过宿主机内核协议栈方式。
- pod-to-remote-pod，使用 fast-path：可以看到**几乎达到了 node-to-node 性能 —— 理论上的最高性能**。

TCP_RR 测试（ping-pong）也是类似的结果：



5 深入剖析（三）：基于 BPF 实现 pod 限速（rate-limiting）

这个功能也是在最新的 Cilium 1.9 中实现的。

5.1 传统方式

传统的 pod 限速方式：加一个 CNI 插件，通过 CNI chaining 给容器设置 TBF qdisc。 甚至还会为了 ingress shaping 设置所谓的 ifb（Intermediate Functional Block）设备。可扩展性差。

> TBF、HTB、MQ、FQ 等 tc qdisc 信息，可参考 [11]。译注。

这种方式存在很多问题：

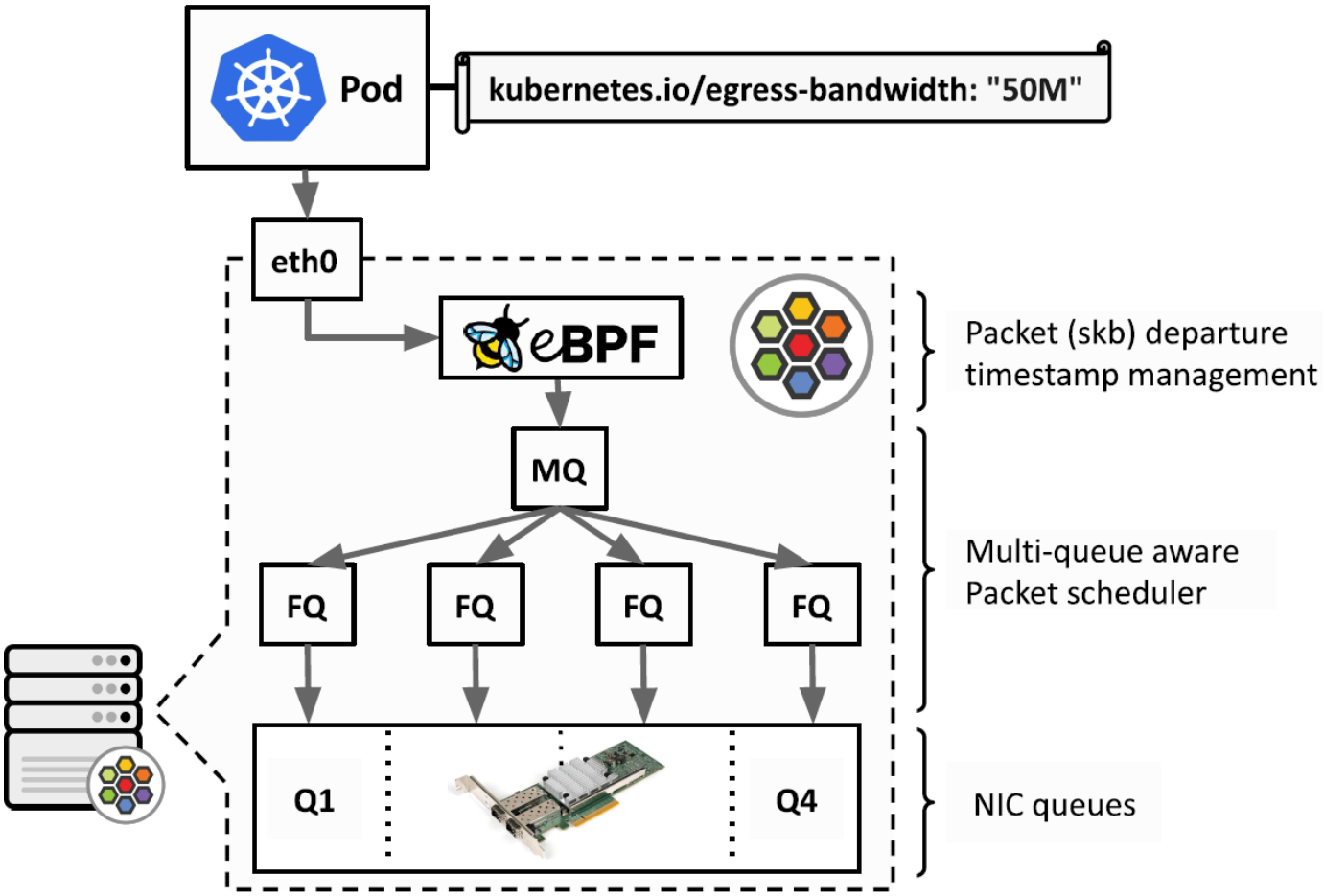
1. 效率不高，因为通常情况下这都会涉及到多队列设备（multi-queue devices）， **需要竞争 qdisc 锁**，不是一种无锁方式。
2. 通过 ifb 设备来做整形（shaping）也不是合适，因为它占用了很多的资源来做 ingress 整形，效果却不怎么样。整形都应该是在出向做的。
3. 整体上这种方式并不是可扩展的。

5.2 Cilium 方式：BPF EDT

在 Cilium 中，我们基于 multi-queue 和 BPF，实现了一种称为 Earlist Departure Time（EDT，最早离开时间） 的无锁（lockless）方式来对 pod 进行限速。

> 关于 lockless，可参考下面的 patch。译注。
>
> author Daniel Borkmann 2016-01-07 22:29:47 +0100
>
> net, sched: add clsact qdisc
>
> This work adds a generalization of the ingress qdisc as a qdisc holding only classifiers. The clsact qdisc works on ingress, but also on egress. In both cases, it’s execution happens without taking the qdisc lock, …

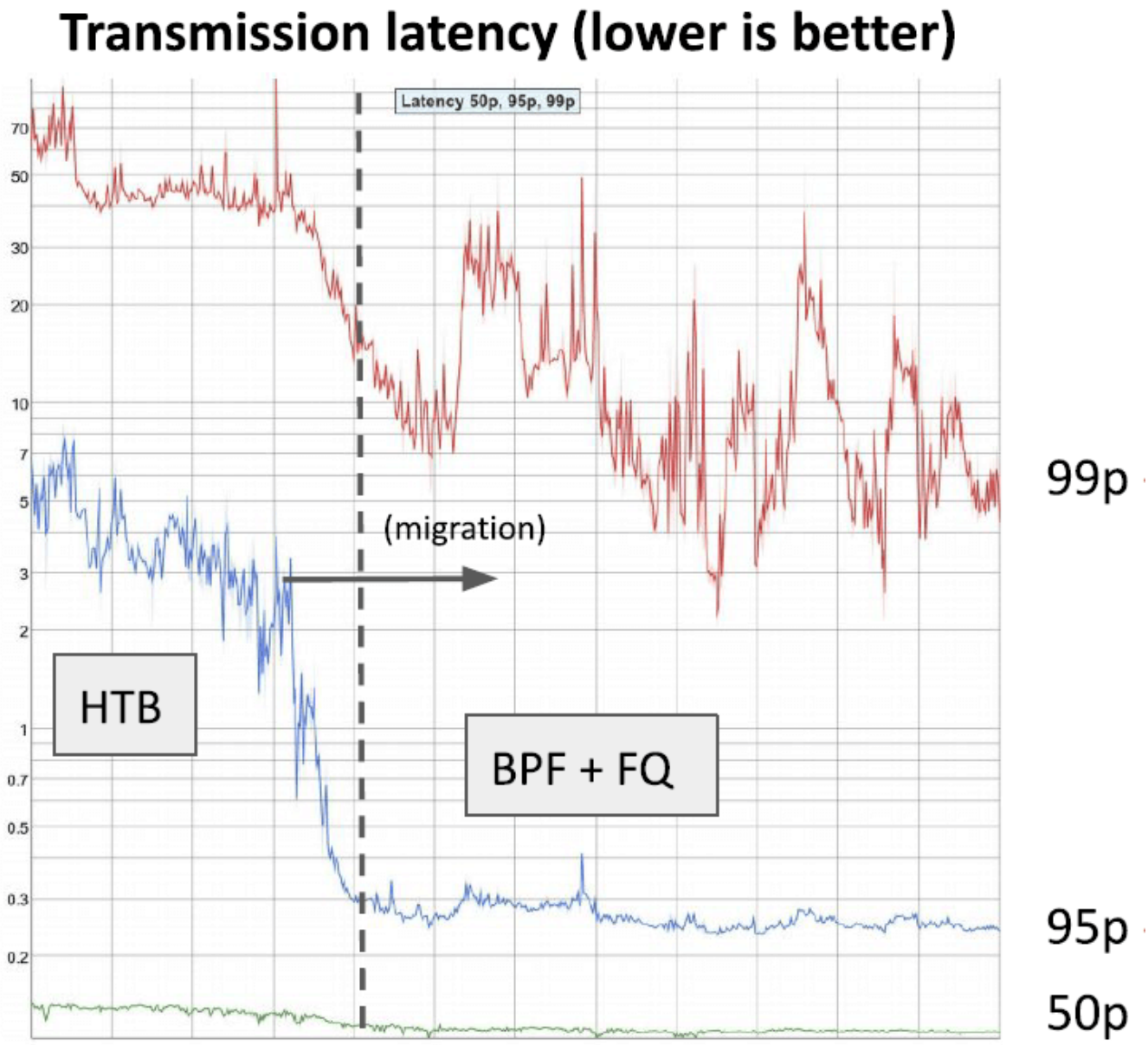
工作原理如下图所示：



- 在 K8s 里**为 pod 设置限速带宽**：`kubernetes.io/egress-bandwidth: "50M"`。
- Cilium agent 生成 BPF 程序。
- BPF 对流量进行分类，根据用户定义的限速带宽，为每个包设置一个离开时间（departure time），其实就是一个时间戳 `skb->tstamp`。
- 在物理设备上设置 FQ qdisc，**FQ qdisc 会根据这个时间戳对包进行调度**。保证不会早于这个时间戳将包发送出去。

5.3 性能对比

下面是基于一些生产流量所做的测试，



左边是传统方式 HTB，右边是 Cilium 方式 BPF+FQ，Y 轴是取了对数（`log`）的。总结：

- p99：延迟降低了 10x
- p95：延迟降低了 20x

6 总结

与其他内核子系统不同，BPF 在内核中的位置非常特殊，因此能极其高效地解决一些复杂的生产问题。 而**其他任何内核子系统，都无法只依靠自身解决这些问题。**

这次分享的主要是网络方面，但其实 BPF 也已经在跟踪、安全等领域大展拳脚，而且 还有更多更多的可能性正在被发掘。此外，我仍然觉得 BPF 才刚刚开始（ feels it's at the very beginning）。

Cilium 以 BPF 为核心，将这项技术带入了主流的 Kubernetes 社区。

> 推荐阅读：大规模微服务利器：eBPF + Kubernetes（KubeCon, 2020）。译注。

另外一件可喜的事情是，GKE（Google Kubernets Engine）已经宣布将 Cilium 作为其下一 代 dataplane。因此，如果是在 GKE 上创建 K8s 集群，那你已经能原生地使用了 Cilium 了。

最后，几个链接：

- github.com/cilium/cilium
- cilium.io
- ebpf.io

译文参考链接（扩展阅读）

1. （译）利用 eBPF 支撑大规模 K8s Service (LPC, 2019)
2. （译）基于 BPF/XDP 实现 K8s Service 负载均衡 (LPC, 2020)
3. Life of a Packet in Cilium：实地探索 Pod-to-Service 转发路径及 BPF 处理逻辑
4. 迈入 Cilium+BGP 的云原生网络时代
5. Cilium Network Topology and Traffic Path on AWS
6. Cilium ClusterMesh: A Hands-on Guide
7. （译）利用 ebpf sockmap/redirection 提升 socket 性能（2020）
8. Cracking kubernetes node proxy (aka kube-proxy)
9. 连接跟踪（conntrack）：原理、应用及 Linux 内核实现
10. L4LB for Kubernetes: Theory and Practice with Cilium+BGP+ECMP
11. （译）《Linux 高级路由与流量控制手册（2012）》第九章：用 tc qdisc 管理 Linux 网络带宽
12. （译）大规模微服务利器：eBPF + Kubernetes（KubeCon, 2020）

« [译] 利用 EBPF SOCKMAP/REDIRECTION 提升 SOCKET 性能（2020）

[译] 深入理解 TC EBPF 的 DIRECT-ACTION (DA) 模式（2020） »