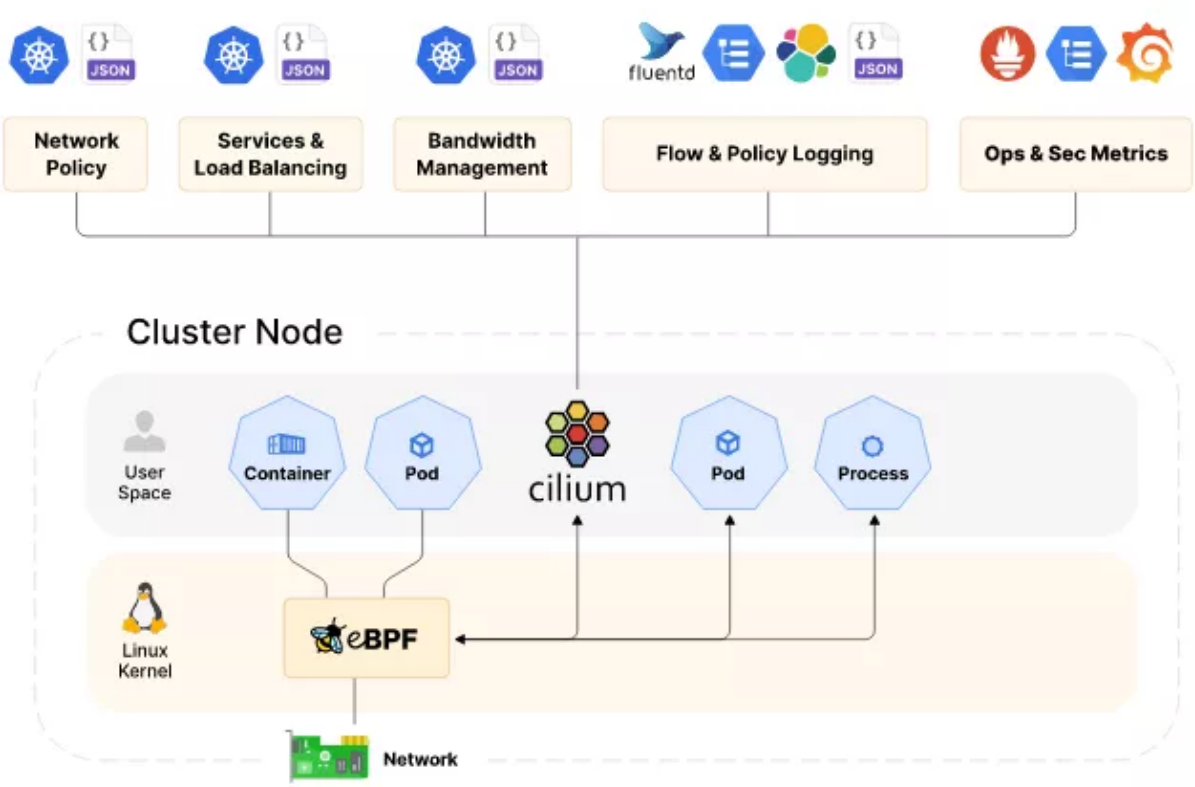
容器网络|深入理解 Cilium

CPP开发者 2022-02-20 11:50



本文翻译自 2019 年 DigitalOcean 的工程师 Nate Sweet 在 KubeCon 的一篇分享: Understanding (and Troubleshooting) the eBPF Datapath in Cilium 。

由于水平有限，本文不免存在遗漏或错误之处。如有疑问，请查阅原文。

1.1 网络成为瓶颈

大家已经知道网络成为瓶颈，但我是从下面这个角度考虑的：**近些年业界使用网络的方式，使其成为瓶颈**（it is the bottleneck in a way that is actually pretty recent）。

- **网络一直都是 I/O 密集型的**，但直到最近，这件事情才变得尤其重要。
- **分布式任务（workloads）业界一直都在用**，但直到近些年，这种模型才成为主流。 虽然何时成为主流众说纷纭，但我认为最早不会早于 90 年代晚期。
- **公有云的崛起**，我认为可能是网络成为瓶颈的最主要原因。

这种情况下，用于管理依赖和解决瓶颈的**工具都已经过时了**。

但像 eBPF 这样的技术使得网络调优和整流（tune and shape this traffic）变得简单很多。 **eBPF 提供的许多能力是其他工具无法提供的，或者即使提供了，其代价也要比 eBPF 大 的多**。

1.2 eBPF 无处不在

eBPF 正在变得无处不在，我们可能会争论这到底是一件好事还是坏事（eBPF 也确实带了一 些安全问题），但当前无法忽视的事实是：Linux 内核的网络开发者们**正在将 eBPF 应用 于各种地方**（putting it everywhere）。其结果是，eBPF **与内核的默认收发包路径（ datapath）耦合得越来越紧**（more and more tightly coupled with the default datapath）。

1.3 性能就是金钱

“Metrics are money”， 这是今年 Paris Kernel Recipes 峰会上，来自 Synthesio 的 Aurelian Rougemont 的 精彩分享。

他展示了一些史诗级的调试（debugging）案例，感兴趣的可以去看看；但更重要的是，他 从更高层次提出了这样一个观点：**理解这些东西是如何工作的，最终会产生资本**

收益（understanding how this stuff works translates to money）。为客户节省金钱，为 自己带来收入。

如果你能从更少的资源中榨取出更高的性能，使软件运行更快，那 显然你对公司的贡献就更大。**Cilium 就是这样一个能让你带来更大价值的工具。**

在进一步讨论之前，我先简要介绍一下 eBPF 是什么，以及为什么它如此强大。

BPF 程序有多种类型，图 2.1 是其中一种，称为 XDP BPF 程序。

- XDP 是 **eXpress DataPath**（特快数据路径）。
- XDP 程序可以直接**加载到网络设备上。**
- XDP 程序在数据包收发路径上**很前面的位置**就开始执行，下面会看到例子。

BPF 程序开发方式：

1. **编写**一段 BPF 程序
2. **编译**这段 BPF 程序
3. 用一个特殊的系统调用将编译后的代码**加载到内核**

这实际上就是编写了一段内核代码，并动态插入到了内核（written kernel code and dynamically inserted it into the kernel）。

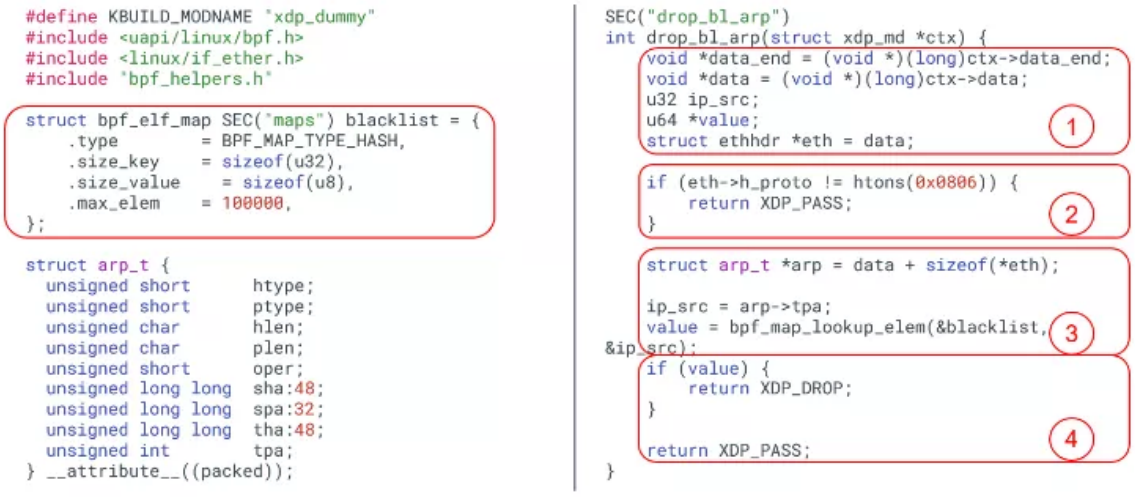


图 2.1. eBPF 代码示例：丢弃源 IP 命中黑名单的 ARP 包

图 2.1 中的程序使用了一种称为 **map** 的东西，这是一种特殊的数据结构，可用于 **在内核和用户态之间传递数据**，例如通过一个特殊的系统从用户态向 map 里插入数据。

这段程序的功能：丢弃所有源 IP 命中黑名单的 ARP 包。右侧四个框内的代码功能：

1. 初始化以太帧结构体（ethernet packet）。
2. 如果不是 ARP 包，直接退出，将包交给内核继续处理。
3. 至此已确定是 ARP，因此初始化一个 ARP 数据结构，对包进行下一步处理。例 如，提取出 ARP 中的源 IP，去之前创建好的黑名单中查询该 IP 是否存在。
4. 如果存在，返回丢弃判决（XDP_DROP）；否则，返回允许通行判决（ XDP_PASS），内核会进行后续处理。

你可能不会相信，就这样一段简单的程序，会让服务器性能产生质的飞跃，因为它此时已 经拥有了一条极为高效的网络路径（an extremely efficient network path）。

三方面原因：

1. 快速（fast）
2. 灵活（flexible）
3. 数据与功能分离（separates data from functionality）

3.1 快速

eBPF 几乎总是比 iptables 快，这是有技术原因的。

- eBPF 程序本身并不比 iptables 快，但 eBPF 程序更短。
- iptables 基于一个非常庞大的内核框架（Netfilter），这个框架出现在内核 datapath 的多个地方，有很大冗余。

因此，同样是实现 ARP drop 这样的功能，基于 iptables 做冗余就会非常大，导致性能很低。

3.2 灵活

这可能是最主要的原因。**你可以用 eBPF 做几乎任何事情。**

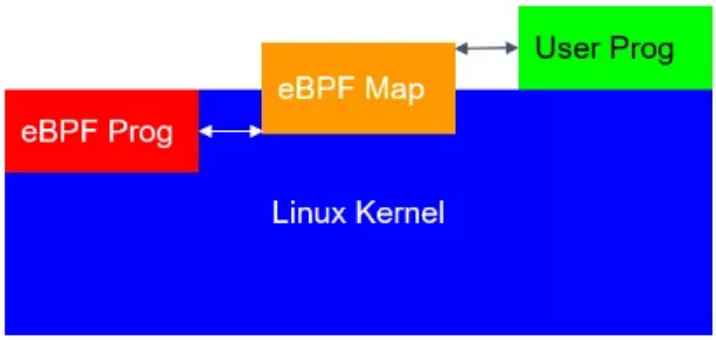
eBPF 基于内核提供的一组接口，运行 JIT 编译的字节码，并将计算结果返回给内核。例如 **内核只关心 XDP 程序的返回是 PASS, DROP 还是 REDIRECT。至于在 XDP 程序里做什么， 完全看你自己。**

3.3 数据与功能分离

eBPF separates data from functionality.

nftables 和 iptables 也能干这个事情，但功能没有 eBPF 强大。例如，eBPF 可以使 用 per-cpu 的数据结构，因此能取得更极致的性能。

eBPF 真正的优势是将“数据与功能分离”这件事情做地**非常干净**（clean separation）：可以在 eBPF 程序不中断的情况下修改它的运行方式。具体方式是修改它访 问的配置数据或应用数据，例如黑名单里规定的 IP 列表和域名。



这里是简单介绍几句，后面 datapath 才是重点。

两篇论文，可读性还是比较好的，感兴趣的自行阅读：

- Steven McCanne, et al, in 1993 - **The BSD Packet Filter**
- Jeffrey C. Mogul, et al, in 1987 - first open source implementation of a packet filter.

我认为理解 eBPF 代码还比较简单，多看看内核代码就行了，但配置和编写 eBPF 就要难多了。

Cilium 是一个很好的 eBPF 之上的通用抽象，覆盖了分布式系统的绝大多数场景。Cilium 封装了 eBPF，提供一个更上层的 API。如果你使用的是 Kubernetes，那你至少应该听说过 Cilium。

Cilium 提供了 CNI 和 kube-proxy replacement 功能，相比 iptables 性能要好很多。

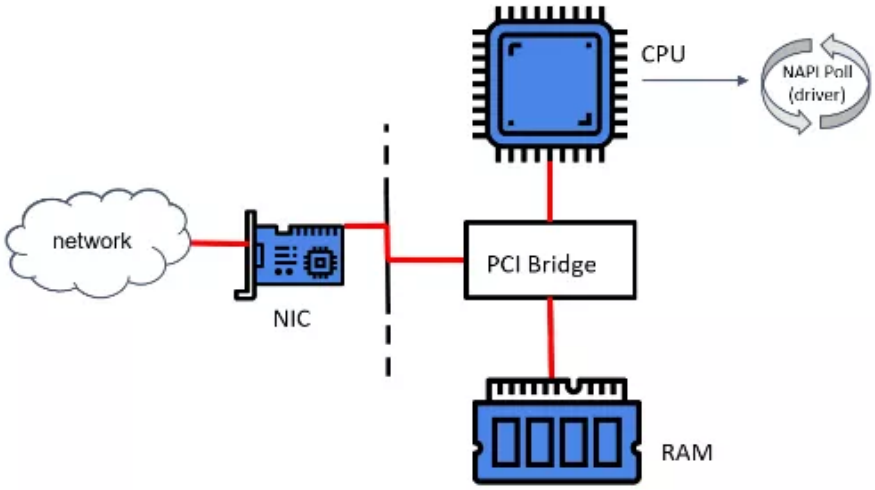
接下来开始进入本文重点。

本节将介绍**数据包是如何穿过 network datapath（网络数据路径）的**：包括从硬件到内核，再到用户空间。

这里将只介绍 **Cilium 所使用的 eBPF 程序**，其中有 Cilium logo 的地方，都是 datapath 上 Cilium 重度使用 BPF 程序的地方。

本文不会过多介绍硬件相关内容，因为理解 eBPF 基本不需要硬件知识，但显然理解了硬件 原理也并无坏处。另外，由于时间限制，我将只讨论接收部分。

6.1 L1 -> L2（物理层 -> 数据链路层）



网卡收包简要流程：

1. 网卡驱动初始化。
   a. 网卡获得一块物理内存，作用收发包的缓冲区（ring-buffer）。这种方式称为 DMA（直接内存访问）。
   b. 驱动向内核 NAPI（New API）注册一个轮询（poll ）方法。
2. 网卡从云上收到一个包，将包放到 ring-buffer。
3. 如果此时 NAPI 没有在执行，网卡就会触发一个硬件中断（HW IRQ），告诉处理器 DMA 区域中有包等待处理。
4. 收到硬中断信号后，处理器开始执行 NAPI。
5. NAPI 执行网卡注册的 poll 方法开始收包。

关于 NAPI poll 机制：

- 这是 Linux 内核中的一种通用抽象，任何等待**不可抢占状态**发生（wait for a preemptible state to occur）的模块，都可以使用这种注册回调函数的方式。
- 驱动注册的这个 poll 是一个**主动式 poll**（active poll），一旦执行就会持续处理 ，直到没有数据可供处理，然后进入 idle 状态。
- 在这里，执行 poll 方法的是运行在某个或者所有 CPU 上的**内核线程**（kernel thread）。 虽然这个线程没有数据可处理时会进入 idle 状态，但如前面讨论的，在当前大部分分布 式系统中，这个线程大部分时间内都是在运行的，不断从驱动的 DMA 区域内接收数据包。
- poll 会告诉网卡不要再触发硬件中断，使用**软件中断**（softirq）就行了。此后这些 内核线程会轮询网卡的 DMA 区域来收包。之所以会有这种机

制，是因为硬件中断代价太 高了，因为它们比系统上几乎所有东西的优先级都要高。

我们接下来还将多次看到这个广义的 NAPI 抽象，因为它不仅仅处理驱动，还能处理许多 其他场景。内核用 NAPI 抽象来做驱动读取（driver reads）、epoll 等等。

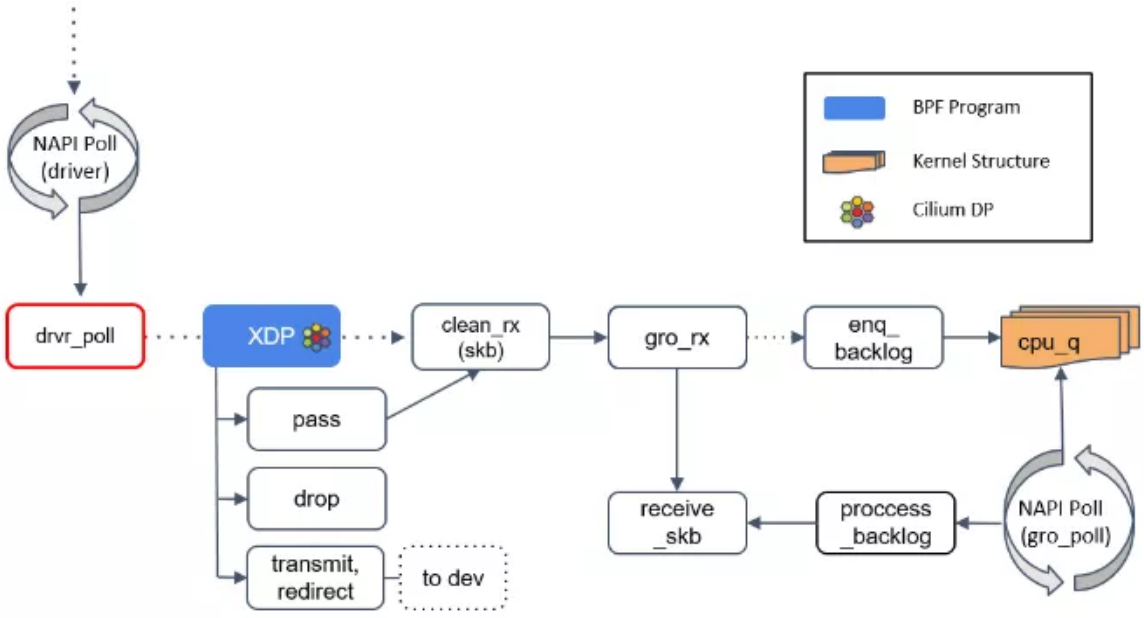
NAPI 驱动的 poll 机制将数据从 DMA 区域读取出来，对数据做一些准备工作，然后交给比 它更上一层的内核协议栈。

6.2 L2 续（数据链路层 - 续）

同样，这里不会深入展开驱动层做的事情，而主要关注内核所做的一些更上层的事情，例如

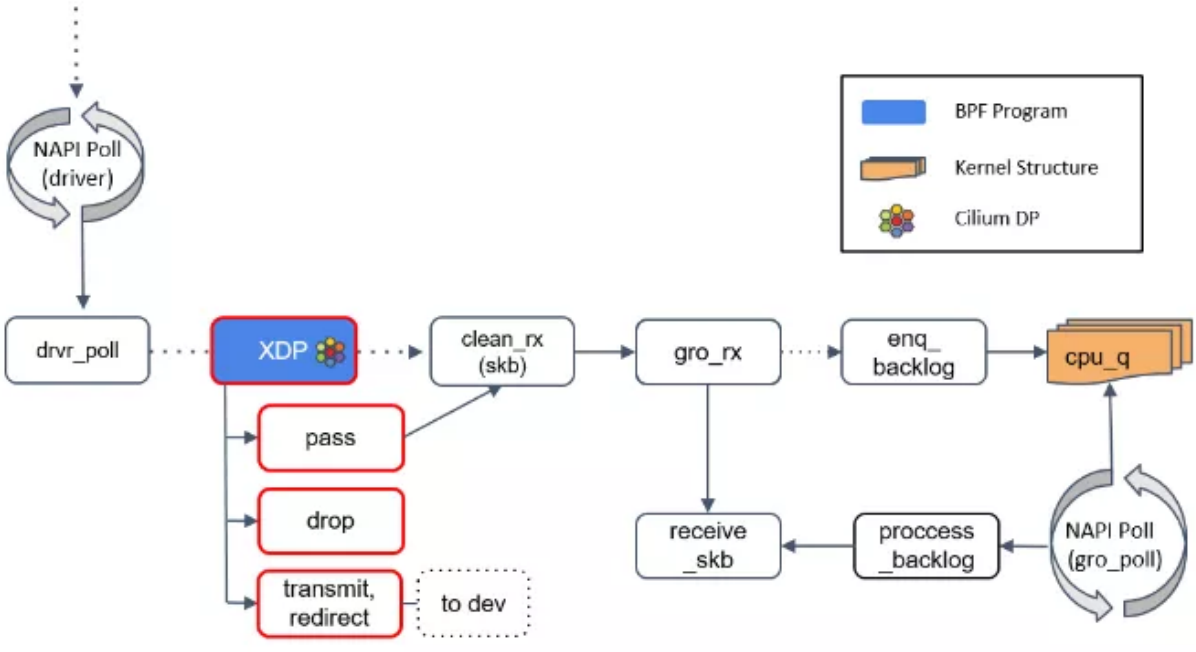
- 分配 socket buffers（skb）
- BPF
- iptables
- 将包送到网络栈（network stack）和用户空间

Step 1：NAPI poll



首先，NAPI poll 机制不断调用驱动实现的 poll 方法，后者处理 RX 队列内的包，并最终 将包送到正确的程序。这就到了我们前面的 XDP 类型程序。

Step 2：XDP 程序处理



如果驱动支持 XDP，那 XDP 程序将在 poll 机制内执行。如果不支持，那 XDP 程序将只能**在更后面执行**（run significantly upstack，见 Step 6），性能会变差， 因此确定你使用的网卡是否支持 XDP 非常重要。

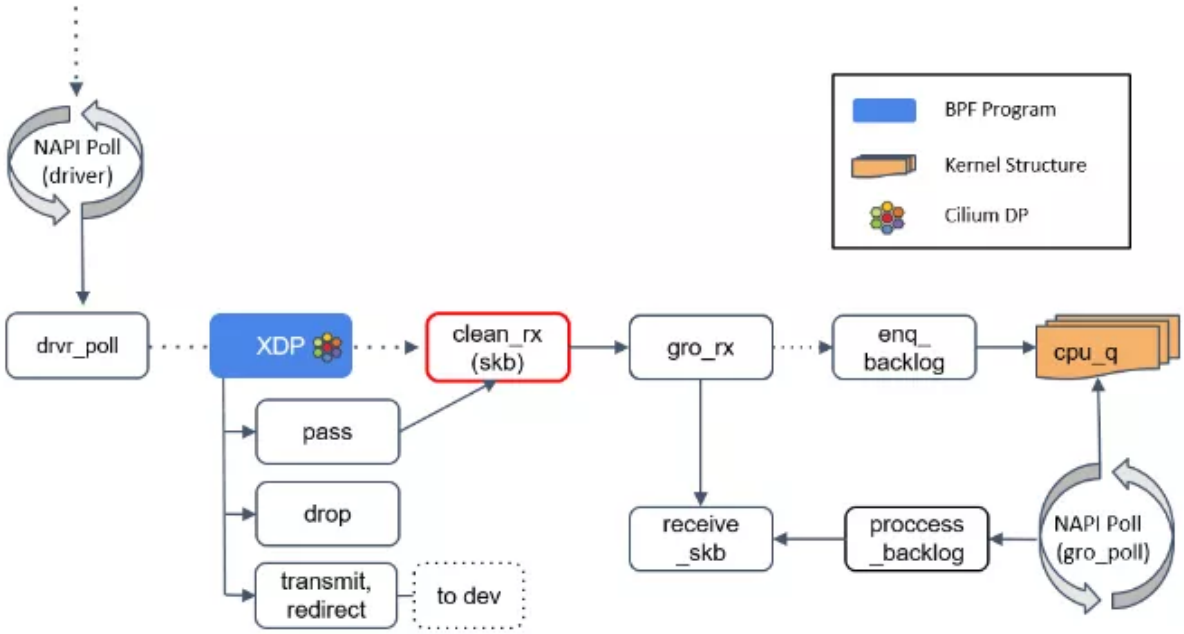
XDP 程序返回一个判决结果给驱动，可以是 PASS, TRANSMIT, 或 DROP。

- Transmit 非常有用，有了这个功能，就可以用 XDP **实现一个 TCP/IP 负载均衡器。**XDP **只适合对包进行较小修改**，如果是大动作修改，那这样的 XDP 程序的性能 可能并不会很高，因为这些操作会**降低 poll 函数处理 DMA ring-buffer 的能力。**

- 更有趣的是 DROP 方法，因为一旦判决为 DROP，这个包就可以直接**原地丢弃**了，而 无需再穿越后面复杂的协议栈然后再在某个地方被丢弃，从而节省了大量资源。如果本次 分享我只能给大家一个建议，那这个建议就是：**在 datapath 越前面做 tuning 和 dropping 越好**，这会显著增加系统的网络吞吐。

- 如果返回是 PASS，内核会继续沿着默认路径处理包，到达 clean_rx() 方法。

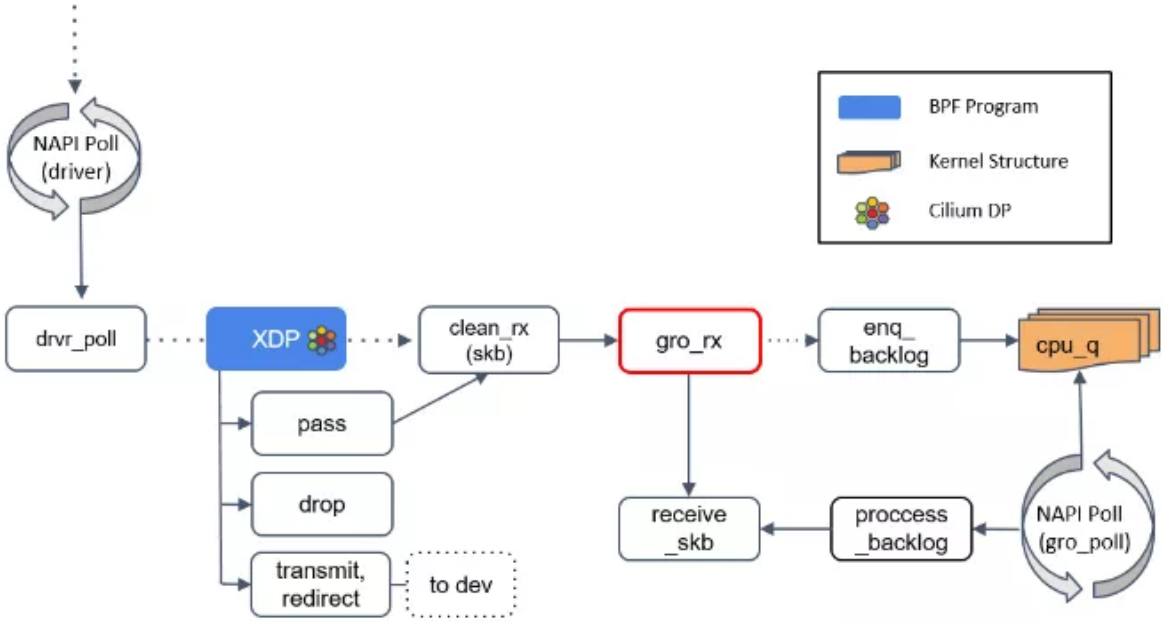
Step 3：clean_rx()：创建 skb



如果返回是 PASS，内核会继续沿着默认路径处理包，到达 clean_rx() 方法。

这个方法**创建一个 socket buffer（skb）对象**，可能还会更新一些统计信息，对 skb 进行硬件校验和检查，然后将其交给 gro_receive() 方法。

Step 4：gro_receive()



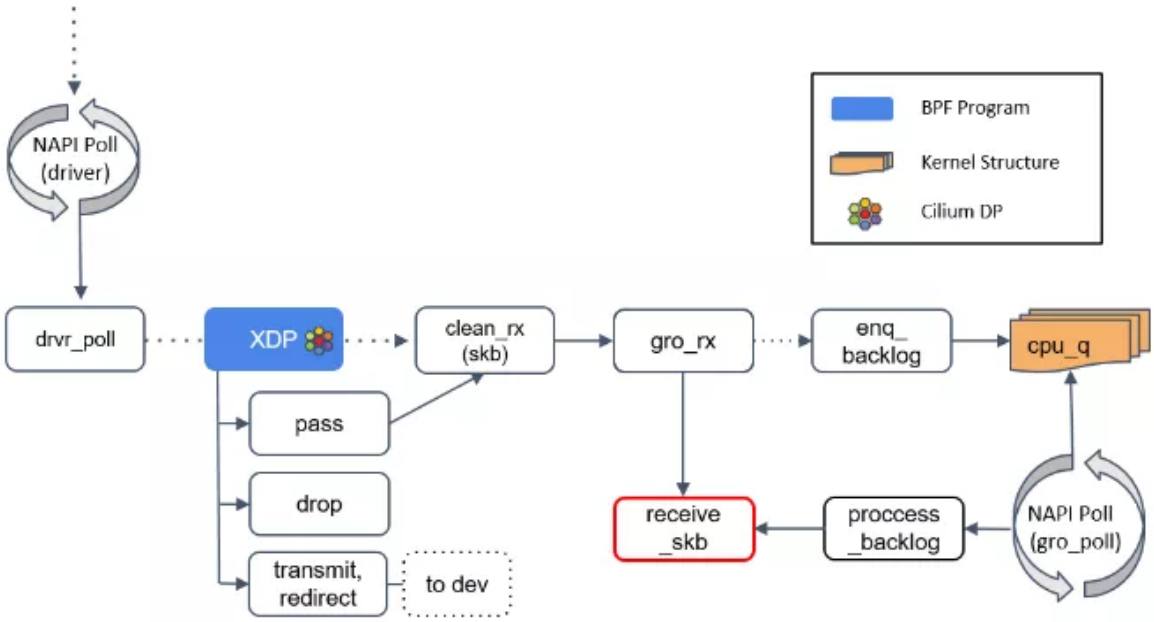
GRO 是一种较老的硬件特性（LRO）的软件实现，功能是**对分片的包进行重组然后交给更 上层**，以提高吞吐。

GRO 给协议栈提供了一次**将包交给网络协议栈之前，对其检查校验和 、修改协议头和发送应答包（ACK packets）的机会。**

1. 如果 GRO 的 buffer 相比于包太小了，它可能会选择什么都不做。

2. 如果当前包属于某个更大包的一个分片，调用 enqueue_backlog 将这个分片放到某个 CPU 的包队列。当包重组完成后，会交给 receive_skb() 方法处理。
3. 如果当前包不是分片包，直接调用 receive_skb()，进行一些网络栈最底层的处理。

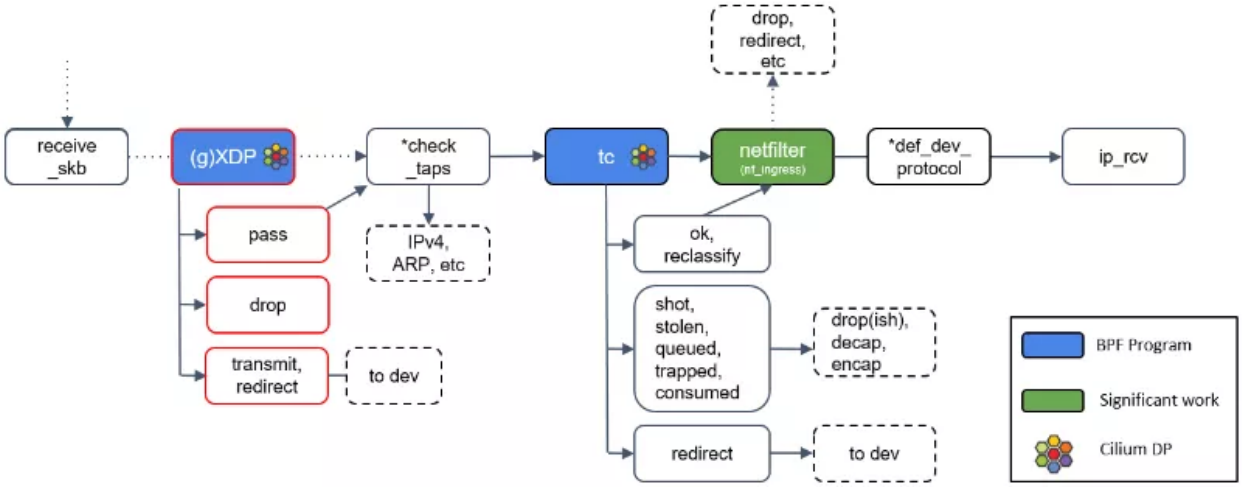
Step 5：receive_skb()



receive_skb() 之后会再次进入 XDP 程序点。

6.3 L2 -> L3（数据链路层 -> 网络层）

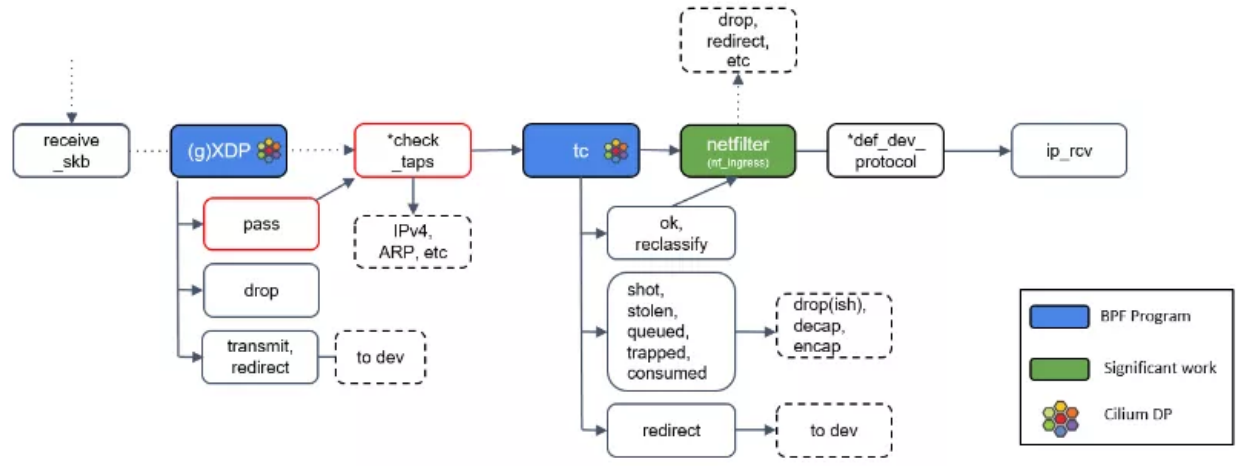
Step 6：通用 XDP 处理（gXDP）



receive_skb() 之后，我们又来到了另一个 XDP 程序执行点。这里可以通过 receive_xdp() **做一些通用（generic）的事情**，因此我在图中将其标注为 (g)XDP

Step 2 中提到，如果网卡驱动不支持 XDP，那 XDP 程序将延迟到更后面执行，这个 **“更后面”的位置指的就是这里的**（g）XDP。

Step 7：Tap 设备处理

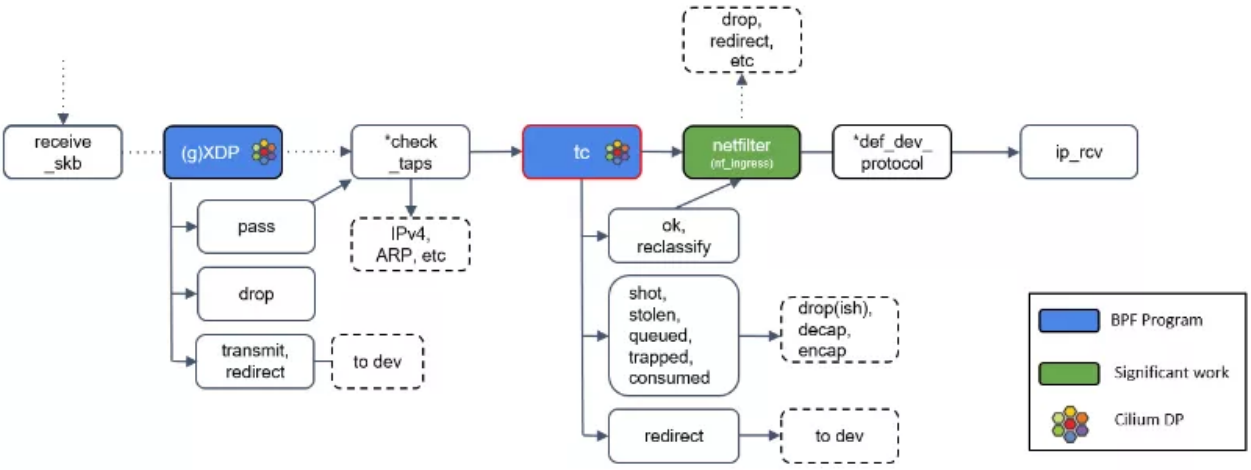


图中有个 *check_taps 框，但其实并没有这个方法：receive_skb() 会轮询所有的 socket tap，将包放到正确的 tap 设备的缓冲区。

tap 设备监听的是三层协议（L3 protocols），例如 IPv4、ARP、IPv6 等等。如果 tap 设 备存在，它就可以操作这个 skb 了。

Step 8：tc（traffic classifier）处理

接下来我们遇到了第二种 eBPF 程序：tc eBPF。



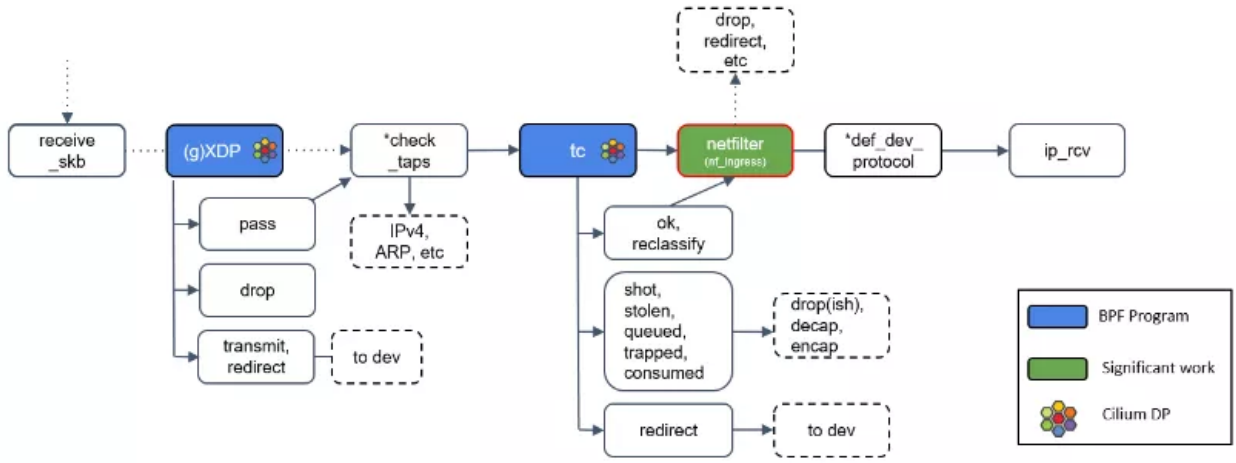
tc（traffic classifier，流量分类器）是 Cilium 依赖的最基础的东西，它提供了多种功能，例如修改包（mangle，给 skb 打标记）、重路由（reroute）、丢弃包（drop），**这 些操作都会影响到内核的流量统计，因此也影响着包的排队规则**（queueing discipline ）。

Cilium 控制的网络设备，至少被加载了一个 tc eBPF 程序。

> 译者注：如何查看已加载的 eBPF 程序，可参考 Cilium Network Topology and Traffic Path on AWS。

Step 9：Netfilter 处理

如果 tc BPF 返回 OK，包会再次进入 Netfilter。



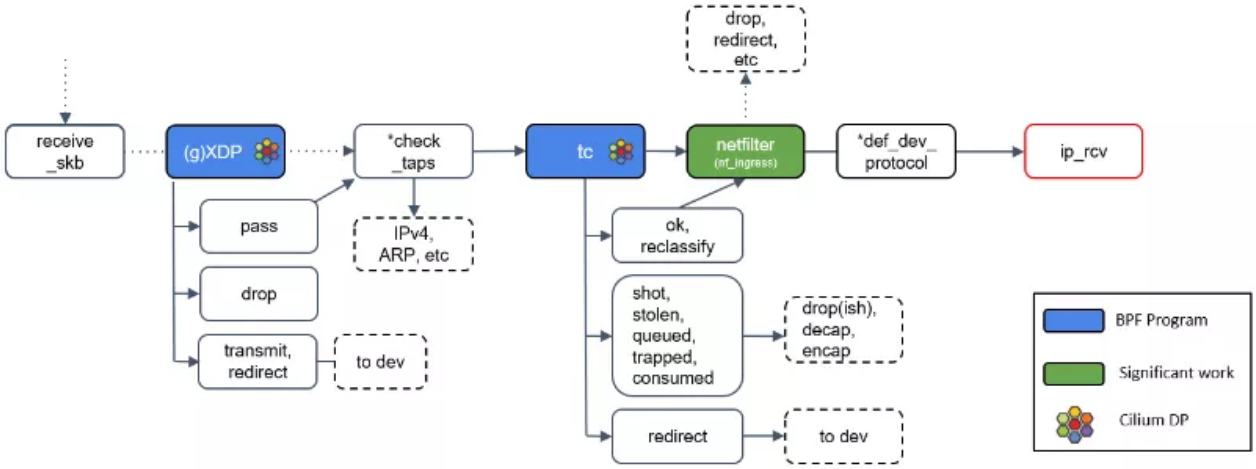
Netfilter 也会对入向的包进行处理，这里包括 nftables 和 iptables 模块。

有一点需要记住的是：**Netfilter 是网络栈的下半部分**（the “bottom half” of the network stack），因此 iptables 规则越多，给网络栈下半部分造成的瓶颈就越大。

*def_dev_protocol 框是二层过滤器（L2 net filter），由于 Cilium 没有用到任何 L2 filter，因此这里我就不展开了。

Step 10：L3 协议层处理：ip_rcv()

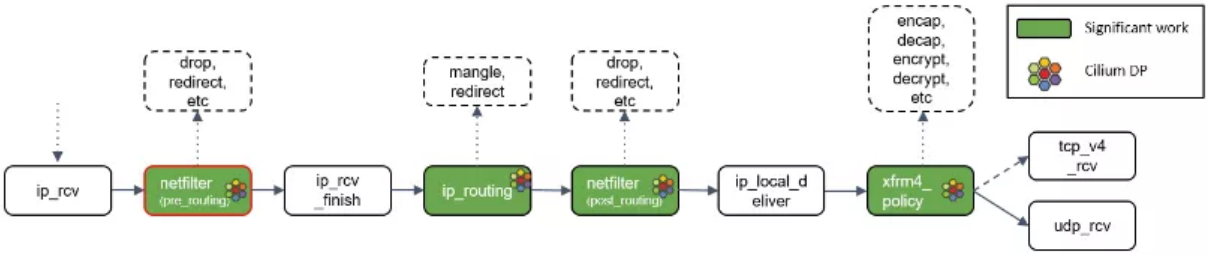
最后，如果包没有被前面丢弃，就会通过网络设备的 ip_rcv() 方法进入协议栈的三层（L3）—— 即 IP 层 —— 进行处理。



接下来我们将主要关注这个函数，但这里需要提醒大家的是，Linux 内核也支持除了 IP 之 外的其他三层协议，它们的 datapath 会与此有些不同。

6.4 L3 -> L4（网络层 -> 传输层）

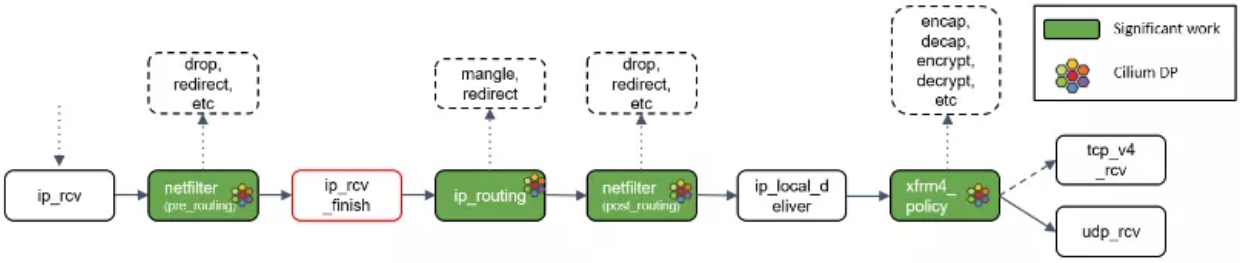
Step 11：Netfilter L4 处理



ip_rcv() 做的第一件事情是再次执行 Netfilter 过滤，因为我们现在是从四层（L4）的 视角来处理 socker buffer。因此，这里会执行 Netfilter 中的任何四层规则（L4 rules ）。

Step 12：ip_rcv_finish() 处理

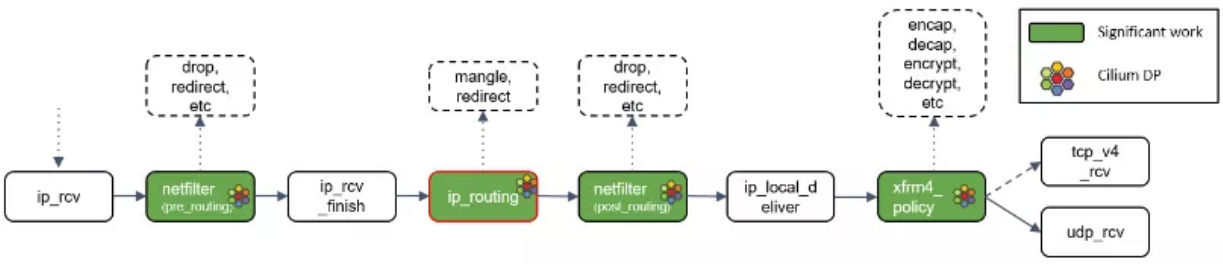
Netfilter 执行完成后，调用回调函数 ip_rcv_finish()。



ip_rcv_finish() 立即调用 ip_routing() 对包进行路由判断。

Step 13：ip_routing() 处理

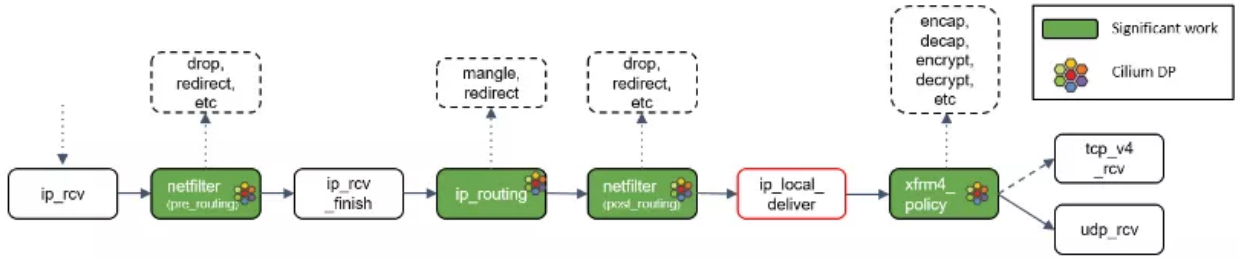
ip_routing() 对包进行路由判断，例如看它是否是在 lookback 设备上，是否能 路由出去（could egress），或者能否被路由，能否被 unmangle 到其他设备等等。



在 Cilium 中，如果没有使用隧道模式（tunneling），那就会用到这里的路由功能。相比 隧道模式，路由模式会的 datapath 路径更短，因此性能更高。

Step 14：目的是本机：ip_local_deliver() 处理

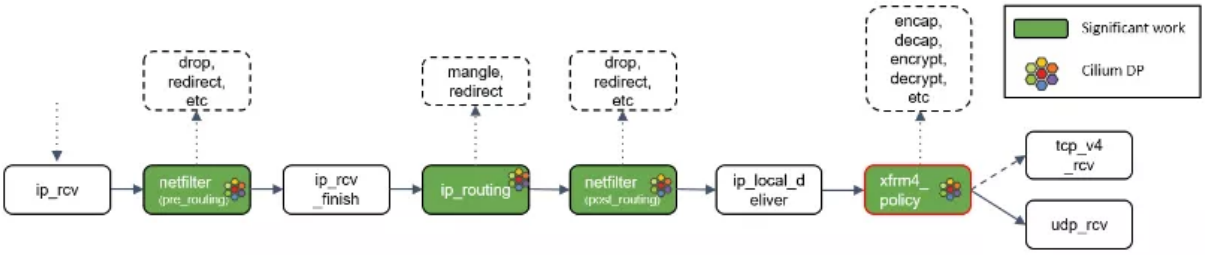
根据路由判断的结果，**如果包的目的端是本机**，会调用 ip_local_deliver() 方法。



ip_local_deliver() 会调用 xfrm4_policy()。

Step 15：xfrm4_policy() 处理

xfrm4_policy() 完成对包的**封装、解封装、加解密**等工作。例如，IPSec 就是在这里完成的。

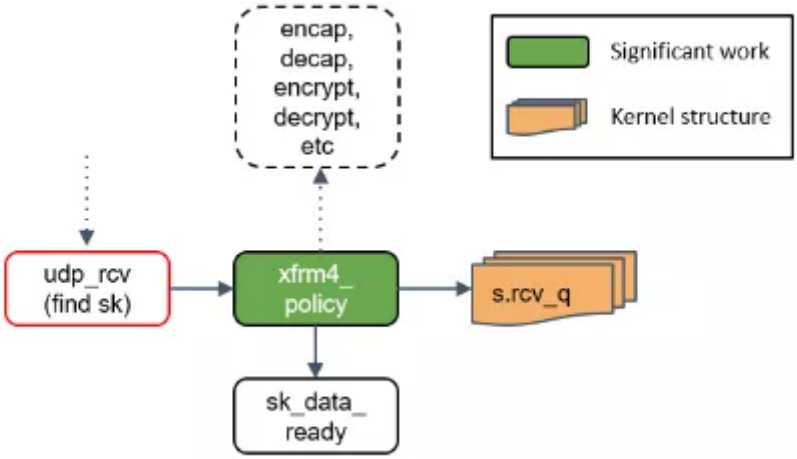


最后，根据四层协议的不同，ip_local_deliver() 会将最终的包送到 TCP 或 UDP 协议栈。这里必须是这两种协议之一，否则设备会给源 IP 地址回一个 ICMP destination unreachable 消息。

接下来我将拿 UDP 协议作为例子，因为 TCP 状态机太复杂了，不适合这里用于理解 datapath 和数据流。但不是说 TCP 不重要，Linux TCP 状态机还是非常值得好好学习的。

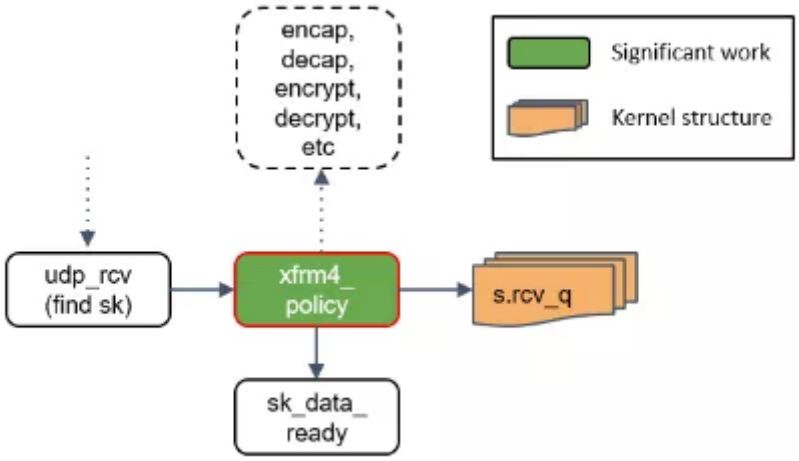
6.5 L4（传输层，以 UDP 为例）

Step 16：udp_rcv() 处理



udp_rcv() 对包的合法性进行验证，检查 UDP 校验和。然后，再次将包送到 xfrm4_policy() 进行处理。

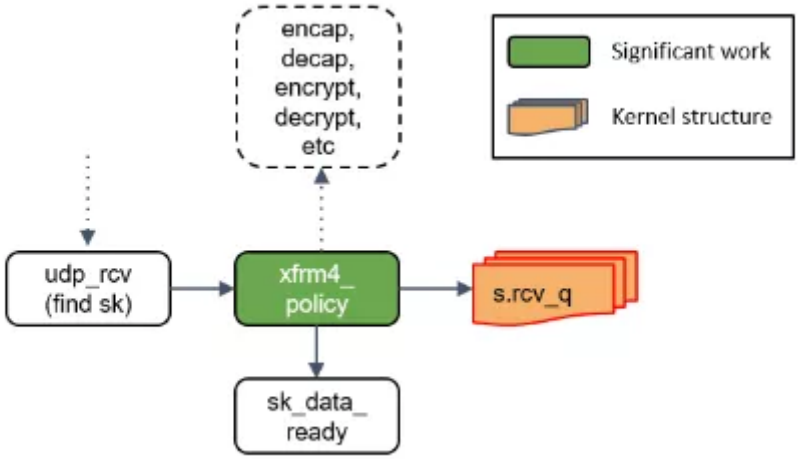
Step 17：xfrm4_policy() 再次处理



这里再次对包执行 transform policies 是因为，某些规则能指定具体的四层协议，所以只 有到了协议层之后才能执行这些策略。

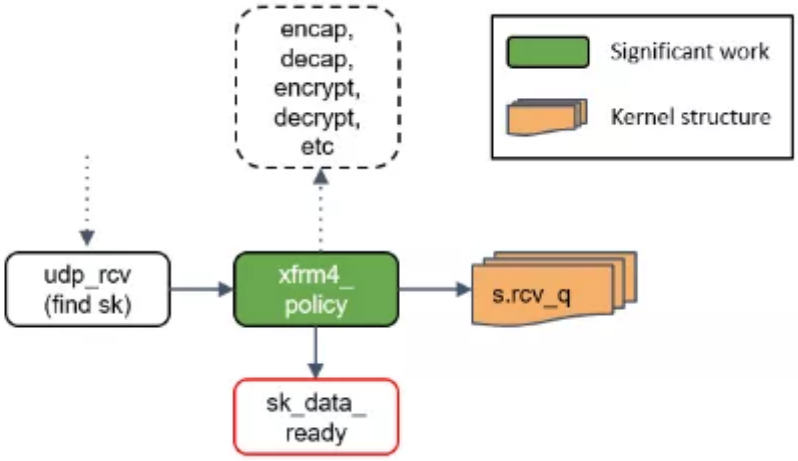
Step 18：将包放入 socket_receive_queue

这一步会拿端口（port）查找相应的 socket，然后将 skb 放到一个名为 socket_receive_queue 的链表。



Step 19：通知 socket 收数据：sk_data_ready()

最后，udp_rcv() 调用 sk_data_ready() 方法，标记这个 socket 有数据待收。



本质上，一个 socket 就是 Linux 中的一个文件描述符，这个描述符有一组相关的文件操 作抽象，例如 read、write 等等。

网络栈下半部分小结

以上 **Step 1~19 就是 Linux 网络栈下半部分（bottom half of the network stack）的全部内容。**

接下来我们还会介绍几个内核函数，但它们都是与进程上下文相关的。

6.6 L4 - User Space

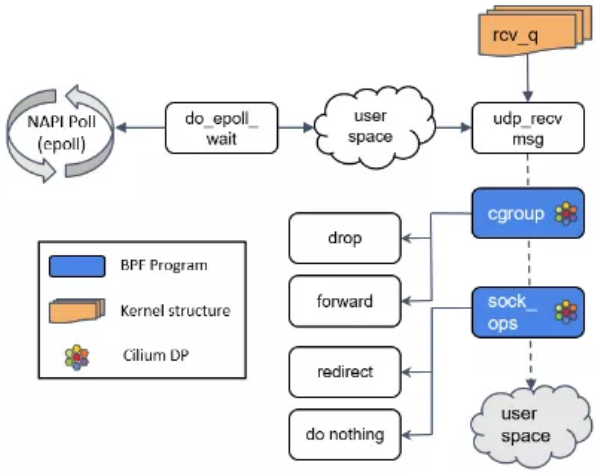
下图左边是一段 socket listening 程序，这里省略了错误检查，而且 epoll 本质上也 是不需要的，因为 UDP 的 recv 方法以及在帮我们 poll 了。

```c
int main(void) {
    int sock = socket(AF_INET, SOCK_DGRAM, 0);
    int poller =  epoll_create(0);

    struct epoll_event ee;
    ee.events = EPOLLIN;
    ee.data.fd = sock;
    epoll_ctl(poller, EPOLL_CTL_ADD, sock, &ee);

    struct epoll_event *events;
    events = calloc(10, sizeof struct epoll_event);
    void *buf = malloc(1500);
    for(;;) {
        epoll_wait(poller, events, MAX_EVENTS, 10);

        recv(sock, buf, 1500, 0);
        // do something with buf
    }
}
```



由于大家还是对 TCP 熟悉一些，因此在这里我假设这是一段 TCP 代码。**事实上当我们调 用** `recvmsg()` **方法时，内核所做的事情就和上面这段代码差不多。**对照右边的图：

1. 首先初始化一个 epoll 实例和一个 UDP socket，然后告诉 epoll 实例我们想 监听这个 socket 上的 receive 事件，然后等着事件到来。
2. 当 socket buffer 收到数据时，其 wait queue 会被上一节的 sk_data_ready() 方法置位（标记）。
3. epoll 监听在 wait queue，因此 epoll 收到事件通知后，提取事件内容，返回给用户空间。

4. 用户空间程序调用 recv 方法，它接着调用 udp_recv_msg 方法，后者又会 调用 **cgroup eBPF 程序** —— 这是本文出现的第三种 BPF 程序。**Cilium 利用 cgroup eBPF 实现 socket level 负载均衡**，这非常酷：
    - 一般的客户端负载均衡对客户端并不是透明的，即，客户端应用必须将负载均衡逻辑内置到应用里。
    - 有了 cgroup BPF，客户端根本感知不到负载均衡的存在。
5. 本文介绍的最后一种 BPF 程序是 **sock_ops BPF，用于 socket level 整流**（traffic shaping ），这对某些功能至关重要，例如客户端级别的限速（rate limiting）。
6. 最后，我们有一个用户空间缓冲区，存放收到的数据。

以上就是 **Cilium 基于 eBPF 的内核收包之旅**（traversing the kernel's datapath）。太壮观了！

Kubernetes	Cilium	Kernel
Endpoint（includes Pods）	Endpoint	tc, cgroup socket BPF, sock_ops BPF, XDP

Kubernetes	Cilium	Kernel
Network Policy	Cilium Network Policy	XDP, tc, sock-ops
Service (node ports, cluster ips, etc)	Service	XDP, tc
Node	Node	ip-xfrm (for encryption), ip tables for initial decapsulation routing (if vxlan), veth-pair, ipvlan

以上就是 Kubernetes 的所有网络对象（the only artificial network objects）。什么意思？ 这就是 k8s CNI 所依赖的全部网络原语（network primitives）。例如，LoadBalancer 对象只是 ClusterIP 和 NodePort 的组合，而后二者都属于 Service 对象，所以他们并不 是一等对象。

这张图非常有价值，但不幸的是，实际情况要比这里列出的更加复杂，因为 Cilium 本身的 实现是很复杂的。这有两个主要原因，我觉得值得拿出来讨论和体会：

首先，内核 datapath 要远比我这里讲的复杂。

1. 前面只是非常简单地介绍了协议栈每个位置（Netfilter、iptables、eBPF、XDP）能执行的动作。

2. 这些位置提供的处理能力是不同的。例如

    a. XDP 可能是能力最受限的，因为它只是设计用来做**快速丢包**（fast dropping）和 **非本地重定向**（non-local redirecting）；但另一方面，它又是最快的程序，因为 它在整个 datapath 的最前面，具备对整个 datapath 进行短路处理（short circuit the entire datapath）的能力。
    b. tc 和 iptables 程序能方便地 mangle 数据包，而不会对原来的转发流程产生显著影响。

理解这些东西非常重要，因为**这是Cilium 乃至广义 datapath 里非常核心的东西**。如果遇到底层网络问题，或者需要做 Cilium/kernel调优，那你必须要理解包的收发/转发路径，有时你会发现包的某些路径非常反直觉。

第二个原因是，eBPF还非常新，某些最新特性只有在 5.x 内核中才有。尤其是 XDP BPF， 可能一个节点的内核版本支持，调度到另一台节点时，可能就不支持。

参考

https://kccncna19.sched.com/event/Uae7/understanding-and-troubleshooting-the-ebpf-datapath-in-cilium-nathan-sweet-digitalocean

- EOF -

推荐阅读 — 点击标题可跳转

1、一个内核网络漏洞详解：容器逃逸

2、2 万字+20 图带你手撕 STL 容器源码

3、C++ STL 容器如何解决线程安全的问题？