基于 eBPF 的虚拟化网络与云原生网络应用

施苏峰[1,2]
(1.东南大学 网络空间安全学院,江苏 南京 211189;2.网络通信与安全紫金山实验室,江苏 南京 211111)

摘 要:近年来,随着 eBPF 的内核代码安全注入机制的发展,eBPF 已经在网络优化、性能监控等方面获得大量应用。介绍了 eBPF 在网络功能虚拟化领域的应用概述,以及其基于容器架构发展而来的云原生网络功能领域的应用概述,并举出了 eBPF 用于上述领域的典型应用:网络功能虚拟化领域的负载均衡、快速丢包、限流应用,以及云原生网络功能领域的 Kubernetes 容器网络加速、服务网格加速应用。

关键词:eBPF;网络功能虚拟化;云原生;负载均衡;Kubernetes;服务网格
中图分类号:TP393 文献标识码:A DOI:10.19358/j.issn.2097-1788.2023.02.002
引用格式:施苏峰. 基于 eBPF 的虚拟化网络与云原生网络应用[J].网络安全与数据治理,2023,42(2):9-18.

Virtual network and cloud native network application based on eBPF

Shi Sufeng[1,2]
(1.School of Cyber Science and Engineering, Southeast University, Nanjing 211189, China;
2.Purple Mountain Laboratories, Nanjing 211111, China)

Abstract: In recent years, with the development of eBPF kernel code security injection mechanism, eBPF has been widely used in network optimization, performance monitoring and other aspects. This paper introduces the application of eBPF in the field of network function virtualization and the application of eBPF in the field of cloud native network function based on container architecture. Typical applications of eBPF used in the above fields are presented, including load balancing, fast packet loss and network traffic rate limiting applications in the field of network virtualization. In addition, the Kubernetes container network acceleration and service grid acceleration applications in the field of cloud native network functions are also mentioned.

Key words: eBPF; network function virtualization; cloud native; load balancing; Kubernetes; service mesh

0 引言

随着 eBPF(extended Berkeley Packet Filter)技术在网络功能虚拟化以及云原生网络功能领域的逐渐成熟,其复用内核协议栈、内核安全校验、流量短路等优势使得传统的网络功能虚拟化以及云原生网络功能拥有了创新性的发展。

1 相关技术介绍

1.1 eBPF

1.1.1 eBPF 相关研究背景

eBPF 技术是伯克利包过滤器(Berkeley Packet Filter,BPF,原本用于在内核中筛选数据包)的延伸与拓展。近年来 eBPF 技术不断发展,已逐步成为一种在内核中运行的类虚拟机。开发者可以通过编写 eBPF 程序与 Verifier 安全验证后将字节码载入内核运行,达到在内核中执行自定义的代码逻辑,完全不会影响到内核的安全性。这些优点及特性使得 eBPF 成为了 Linux 社区网络领域最受开发者关注的新技术。

除了上述优点外,eBPF 技术还简化了内核开发者的开发难度。在 eBPF 诞生之前,为了满足性能、可拓展性、向后兼容等需求,内核开发者不得不保证在引入新代码时不影响原先的代码,使得内核开发成为非常复杂繁琐的工作。eBPF 技术通过内核内置的字节码虚拟机机制,可以实现数据包过滤、调用栈分析跟踪等高级功能,大大简化了内核开发的难度。

eBPF 技术的上述优点和特性备受网络领域开发者的欢迎。尽管有大量网络监控、可观测性、安全

等功能方面的各类需求,然而由于内核开发工作需要修改内核源码或加载内核模块,工程复杂且难度非常大,导致网络可观测性、安全、优化等网络功能发展很慢。eBPF 的诞生可以改变这一现状。eBPF 技术能在内核中运行沙箱程序,保证了内核的安全性,同时还无需修改内核源码或者加载内核模块。将内核赋予可编程的特性之后,就能在现有的内核的功能的基础上,通过编写 eBPF 程序来实现特定的功能,大大减小了开发者进行内核功能开发的难度。随着网络功能虚拟化应用的需求增多以及云原生的发展,eBPF 技术越来越受到重视。

1.1.2 eBPF 优点

本文主要的研究重点在于基于 eBPF 的网络功能虚拟化应用与云原生网络功能应用,在网络功能虚拟化领域,典型的数据面加速方案的实现方式是数据平面开发套件(Data Plane Development Kit,DPDK)技术。DPDK 采用内核旁路的方式,无法复用内核网络协议栈。相比于内核旁路的 DPDK,eBPF 使用快速数据路径(eXpress Data Path,XDP)Hook 来操作,eBPF/XDP 的优点在于:

(1)eBPF/XDP 可以复用内核网络协议栈的功能,也可以选择性地跳过内核网络协议栈,加速关键性能路径。

(2)由于 eBPF/XDP 将控制权保留在了内核的控制范围内,相比于 DPDK 这种完全跳过内核的技术形式,eBPF/XDP 保持了内核的安全边界。

(3)eBPF/XDP 不需要额外的硬件特性支持,通用版本的内核即可支持,且新版本的内核支持性更好,例如发行版 Linux 系统的 Ubuntu 20.10 及以上版本的内核甚至会默认打开 BTF(BPF Type Format,一种元数据格式)内核配置选项。AF_XDP 在复用内核网卡驱动的情况下,能达到接近 DPDK 的性能。

而云原生网络功能的由来,源于传统网络功能虚拟化技术提供商将网络功能迁移到基于容器的架构,这种网络功能虚拟化也称为云原生网络功能。在以 Kubernetes 为基础设施的云原生网络功能方面,eBPF 可以对 Kubernetes 的 kube-proxy/Iptables 进行优化,它的优势在于:

(1)针对 Kubernetes 的 Node 节点的东西向流量而言,eBPF 程序可以在 Socket 层拦截 CONNECT、SENDMSG、RECVMSG 等请求的系统调用,根据其 IP 和端口号将请求做反向变换后映射到后端 Pod,实现高性能的东西向流量转发。

(2)针对从 Kubernetes 集群外进入节点,或者从节点出 Kubernetes 集群的南北向流量。当南北向流量到达 Node 时,eBPF 程序可以在 XDP 和 TC 等 Hook 处进行处理,将 IP 和端口号映射到后端的 PodIP 和端口号,或是将其发到其他目的地址的 Node 的 Pod 中。

而从 eBPF 本身原理的角度出发,它的优点包括快速、灵活、数据与功能分离三方面。第一个方面,首先 eBPF 程序本身并不比 Iptables 快,但是 eBPF 程序更短,比 Iptables 处于内核网络路径中更靠前的位置,而 Iptables 基于冗长的 Netfilter 内核网络框架,相较于 Iptables,eBPF 程序的效率会在快速丢包等场景中高很多,这将在本文的 2.3 小节介绍。第二个方面,eBPF 具有灵活性,eBPF 程序仅适用内核提供的接口,运行 JIT 编译的字节码,至于在程序中运行何种代码逻辑,或是给内核返回 PASS、DROP、REDIRECT 中的哪一种结果,将完全取决于开发者。第三个方面,eBPF 提供了数据与功能分离的能力,可以体现在如 2.3 小节中介绍的快速丢包场景下的 IP 列表黑名单应用。

1.1.3 eBPF 研究现状

在 eBPF 技术方面,2013 年之前,eBPF 技术尚未兴起。当时的软件定义网络(Software Defined Network,SDN)技术蓝图中,对网络数据路径进行操作的工具有 OpenvSwitch(OVS)、TC(Traffic Control)以及涵盖 Iptables、IPVS 等工具的内核 Netfilter 子系统。而 eBPF 技术当时还只是其前身 BPF 技术,仅仅用于 TCPDump,在内核中进行抓包,使用场景和功能非常受限。当时的 TC 和 Netfilter 子系统中还存在代码的重复,而当时内核中最先进的数据平面 OVS 对内核中其他网络模块的集成不理想。

2013 年后,eBPF 被内核社区提出,它提供在内核中加载定制化代码的能力,从而完成在不修改、编译内核的情况下对内核网络数据路径完成修改。2014 年,eBPF 技术正式合并入内核,用一个拓展的指令集全面替换老的 BPF 解释器,正式由 BPF 技术发展到了 eBPF 技术。

2015 年,eBPF 的开发分成了两个方向,分别是网络方向与观测方向,在网络方向,eBPF 可以灵活地对内核网络路径进行编程,从而优化内核网络路径的性能;在观测方向,eBPF 程序可以附着在 Kprobes

上，获取系统的观测数据。

2016 年，XDP 被合并到内核，支持在驱动的 Ingress 层附加 BPF 程序，支持将 eBPF 程序在 XDP Hook 点。

2017 年，eBPF 逐渐兴起，开始被 Netflix、Facebook、Cloudflare 等大型公司广泛使用。Netflix 使用 eBPF 技术实现 Linux 系统观测；Facebook 使用 eBPF 技术与 XDP 完成 DDoS 防护和负载均衡，改善了原先的 IPVS 四层负载均衡的性能；Cloudflare 使用 eBPF 技术和 XDP 完成了公司内部的 DDoS 产品和负载均衡环境架构。

同年，随着 eBPF 社区的活跃发展，eBPF 成为了内核独立的子系统，同年 bpftool 和 libbpf 工具被添加。bpftool 工具用于检查内核内 eBPF 的状态，以及查看内核加载了哪些 BPF 程序。libbpf 工具用于方便用于空间应用使用 eBPF，它接管了所有加载的工作，不再需要用户处理复杂的加载过程。

2018 年，内核添加了 BTF 组件，它是一种元数据格式，通过 BTF，内核内置了自己的数据格式和内部数据结构，给后来的“一次编译，到处运行”等 BPF 特性提供了基础。

同年，新 Socket 类型 AF_XDP 添加进了内核，它提供了在零拷贝的前提下将包从网卡驱动送到用户空间的能力。而在此之前，数据包到达网卡后，需要先经过 XDP，然后才能为这个包分配内存，所以在 XDP 层直接将包送到用户态是无需拷贝的[1]。AF_XDP 提供与 DPDK 类似的能力，但是 DPDK 需要重写网卡驱动，需要额外维护用户空间的驱动代码，而 AF_XDP 提供在服用内核网卡驱动的情况下，接近 DPDK 的性能。相比与 DPDK，AF_XDP 的优点在于：DPDK 这种内核旁路的方式会导致内核基础设施里的网络管理工具都无法复用，而 AF_XDP 复用了内核基础设施，可以复用网络管理工具。因为这些操作都发生在 XDP 层，所以它被命名为 AF_XDP，插入到这里的 eBPF 代码可以直接将包送到 Socket[1]。

同年，Bpffilter 被提出，它提供了通过用户空间驱动将 Iptables 规则转换成 eBPF 代码的能力，是在用户无感知的情况下将 Iptables 转换成 eBPF 的初次尝试。

2019 年，Brendan Gregg 等人[1]编写了 BPF 性能工具著作，同年 Cilium 完全替代了 Kubernetes 中基于 Iptables 的 kube-proxy，实现了基于 eBPF 的网络接口全功能。

2020 年，eBPF 开始运用于 Linux 的安全模块，Google、Facebook、Microsoft 等大型互联网厂商[1]均开始了针对 eBPF 的研究，Google 贡献了 BPF LSM，并将其部署在了他们的数据中心服务器上；Cilium 开始支持基于 XDP 的 Service 负载均衡和 Host 网络策略；Facebook 研发了基于 BPF 的 TCP 拥塞控制模块；Microsoft 基于 eBPF 实现了他们的操作系统监控工具。

1.1.4 eBPF 编程简析

eBPF 编程分为用户态程序和内核态程序两部分，其步骤如图 1 所示。其中，内核态程序一般使用 C 语言开发，用户态程序可以使用 BCC 或者 libbpf 等工具开发。对于初学者来说，使用 Python 语言的 BCC 可以更快上手。本文使用 libbpf 进行开发，libbpf 的可操作性更高。

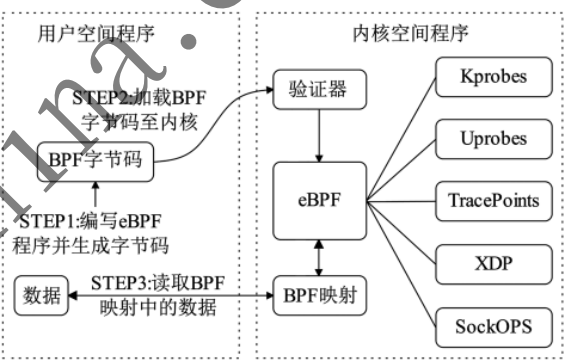

图 1 eBPF 程序编程简析

主要的编程开发步骤分为以下三个步骤：

(1)开发内核态 eBPF 程序，一般使用 C 语言。将 C 语言编写的内核态 eBPF 程序编译为 BPF 字节码。

(2)通过用户态 eBPF 程序系统调用将内核态 eBPF 程序加载到内核。

(3)从 BPF 映射读取数据。

其中，BPF 映射是 BPF map，它是一种特殊的数据结构，eBPF 程序可以使用 BPF 映射进行存储，除此以外，用户空间程序也可以使用 BPF 映射来与正在运行中的内核态 eBPF 程序产生交互、获取数据。可以说 BPF 映射是内核和用户态之间传递数据的桥梁。而 BPF Helper 函数赋予了 eBPF 程序无需穿越整个内核网络栈即可操作某些内核功能的能力[1]。

使用 libbpf 开发的步骤类似，也是先开发内核态程序，再完成用户态的加载、挂载、映射读取以及输出程序等。除此以外，在类似于 Ubuntu 20.10 及

以上的 Linux 发行版中的高版本的内核中,默认开启了 BTF 内核选项,支持了 BTF 特性,因此不需要再去引入复杂繁琐的内核头文件就可以获取到内核数据结构的定义。只需要通过 bpftool 工具来生成 vmlinux.h 的头文件,在这个头文件中,就包含了内核数据结构的定义,具体步骤如下:

(1)使用 bpftool 工具生成内核数据结构定义头文件;

(2)使用 C 语言编写"<程序名>.bpf.c"的内核态 eBPF 程序;

(3)将内核态 eBPF 程序编译为 BPF 字节码;

(4)使用 bpftool 为 BPF 字节码生成脚手架头文件;

(5)用户态程序中引入步骤(4)中生成的头文件,开发将内核态 eBPF 程序加载、挂载到内核函数和跟踪点等的用户态程序。

1.2 网络功能虚拟化

随着软件定义网络思想的逐步普及,网络功能虚拟化技术逐渐成为了该领域研究的热点之一,受到企业、开发者、学术研究者的关注。本小节主要对网络功能虚拟化技术进行简述,在网络功能虚拟化技术诞生之前,网络的功能只能通过传统的网络设备来提供,如交换机、路由器等。网络功能虚拟化是一种将网络基础设施虚拟化的技术,它提出了抽象化网络功能的思想,在标准化的计算节点中安装控制软件来控制网络功能,使得通用架构的计算节点可以拥有传统网络设备的能力。它通过虚拟化技术来将传统网络设备的功能以软件可编程的形式来实现,这种方式不仅降低了网络设备的成本,而且赋予了开发人员定制网络功能的能力。

网络功能虚拟化的核心思想是使用通用服务器的虚拟化技术来实现网元的功能。它使用虚拟化技术和通用服务器来实现传统网络设备的功能,用于替代网络中传统的网络设备,从而减少网络搭建和运维的成本。同时,其可编程的特点提高了网络功能的定制性、灵活性、可拓展性。

在网络功能虚拟化领域,数据面加速的解决方案通常是内核旁路,具体的代表的例子如业界非常热门的 DPDK 技术。然而,这种内核旁路的形式存在弊端,由于内核旁路绕过了内核协议栈,因此内核协议栈的功能无法被复用,同时内核本身所提供的原生的应用隔离与安全机制等功能也无法继续发挥作用。针对 DPDK 的弊端,eBPF/XDP 技术提供了一种与 DPDK 截然不同的、复用内核网络协议栈的网络功能虚拟化解决方案,在本文第 2 节中详细展开[2-3]。

1.3 云原生网络功能

网络功能虚拟化这一概念原本指的只是传统网络功能基于虚拟化技术在通用设备上的实现,而随着网络技术的进一步发展,许多网络功能虚拟化技术的提供商开始将他们的许多网络功能迁移到基于容器的架构,这种网络功能虚拟化也称为云原生网络功能。

随着云计算的发展进入云原生时代,Kubernetes 成为社区热点,受到企业、开发者、学术研究者的关注,逐步成为云时代的操作系统。本文对 eBPF 技术在云原生网络方面的应用进行研究,主要关注于 Kubernetes 的容器网络。

在 Kubernetes 的容器网络技术架构中,KubeProxy 是一个重要的组件。它基于 Linux 内核的 Iptables 等功能开发,部署在 Node 节点上,用于 Kubernetes Service 的通信与负载均衡机制。

本文在云原生网络功能领域的研究重点主要在 eBPF 技术在 Kubernetes 容器网络加速以及服务网格加速方面,在本文第 3 节中详细展开[4]。

2 基于 eBPF 的网络功能虚拟化应用研究

2.1 基于 eBPF 的网络功能虚拟化应用概述

基于 eBPF 的网络功能虚拟化应用主要通过 XDP 来实现,XDP 基于 eBPF,是 Linux 内核中提供的高性能、可编程网络数据包处理框架。但在具体使用的层面,可以理解为内核中提供的处理网络包的一个位置。XDP 的功能是,可以在 Linux 内核协议栈中的 XDP 处加载 eBPF 程序,通过在 XDP 处所运行的 eBPF 程序完成对网络数据包的操作[5]。

实现虚拟化网络功能数据面加速的主流传统方式是内核旁路,以业界非常热门的 DPDK 技术为代表。通常,通过软件能力实现的网络功能的虚拟化,需要拥有高性能的数据包的处理能力,它对每一个传输的数据包都有着极致的性能要求,而在通用的操作系统中,网络内核的协议栈通常只是针对通用数据包的处理,通常会比较复杂和冗余,包处理的时延相对较高,在网络功能虚拟化的背景下无法达到高吞吐量的需求。而 DPDK 所使用的内核旁路的形式实现了可编程的包处理过程,将包处理的能力交由专门的用户空间应用来负责,从而避免了

内核与用户态的上下文切换的大量性能开销。

然而，这种内核旁路的形式存在弊端，由于内核旁路绕过了内核协议栈，因此内核协议栈的功能无法被复用，需要手动在用户空间应用中实现；除此以外，由于内核本身被绕过了，它所提供的原生的应用隔离与安全机制等功能也无法继续工作。

在 eBPF 技术出现之前，网络功能虚拟化的解决方案主要基于 DPDK 内核旁路技术实现。而内核旁路的方式无法复用内核网络协议栈，缺失了内核的安全边界。相较于前者，eBPF 技术则可以复用内核网络协议栈的功能，也可以选择性地跳过内核网络协议栈，加速关键性能路径。不仅如此，eBPF 技术将控制权保留在了内核的控制范围内，相比于内核旁路这一种的技术形式，eBPF 保持了内核的安全边界，具有灵活性和安全性。

eBPF 程序仍然工作在操作系统内核的安全执行环境中，不仅可以复用内核协议栈的能力，而且可以手动定制各种包处理的应用，比如负载均衡、安全等，为网络功能的虚拟化提供了一种合适的解决方案。相比于 DPDK，XDP 是内核网络路径的一部分，与内核的网络协议栈完全兼容，可以协同工作，eBPF/XDP 程序通过 C 语言编写，然后编译成字节码文件，并且需要通过安全性的分析校验，保证了内核的安全边界[6]。

而在对基础设施的需求的角度上，eBPF/XDP 程序完全依赖 Linux 内核，无需任何硬件依赖。它可以选择是否绕过内核协议栈，如果不绕过，那么还可以与内核的网络协议栈协同工作，可以复用内核协议栈，可以在 XDP 处基于特定的需求场景来开发 eBPF 程序从而加速关键性能路径。当然，XDP 程序也能够完全绕过内核网络协议栈从而达到更高的转发性能。

综上所述，eBPF/XDP 提供可选择的数据包处理能力：如果是高性能场景需求，其可以将处理逻辑下放到网卡驱动中，从而提升包处理的性能；如果是通用的处理逻辑，则可以继续走内核网络协议栈，以复用内核网络协议栈的能力。

2.2 基于 eBPF 的负载均衡场景网络功能虚拟化典型应用方案

负载均衡是一种将单台服务器的负载进行平衡，分摊到多台服务器上进行运行的技术。负载均衡场景中，服务器与负载均衡器之间的通信需要满足高速率、低时延的要求，非常适合使用 eBPF 技术来进行优化。

在负载均衡这一类高性能网络场景中，如果地址的转换发生在繁琐冗长的内核协议栈中，那么它的效率将不尽人意，而通过 eBPF 程序在网络路径最开始的 XDP 处进行处理，可以在 Linux 内核协议栈之前处理网络数据包，达到较快较优的效率，可以在负载均衡这一类的高性能网络场景中获得应用。

由于 XDP 程序在内核协议栈初始化之前运行，并且可以在 XDP 处理后正常通过内核协议栈继续处理，因此不需要重新实现内核协议栈的通用逻辑，只需要在 XDP 程序中实现最核心的网络逻辑。

图 2 展示了数据包进入网卡之后的流程，可以看到，XDP Hook 中的 eBPF 程序会在内核协议栈处理数据包之前完成，此时内核还没有为包分配 struct sk_buff 结构体，在这个 eBPF 程序里，可以选择丢弃这个数据包，可以通过当前网卡再将这个数据包发送出去，可以将这个数据包重定向到其他网络接口，还可以将这个数据包放行到常规的内核网络协议栈。基于 eBPF 的网络功能虚拟化典型应用方案中，典型的是负载均衡应用。搭建如图 3 所示的负载均衡环境，使用 eBPF 程序开发负载均衡器。

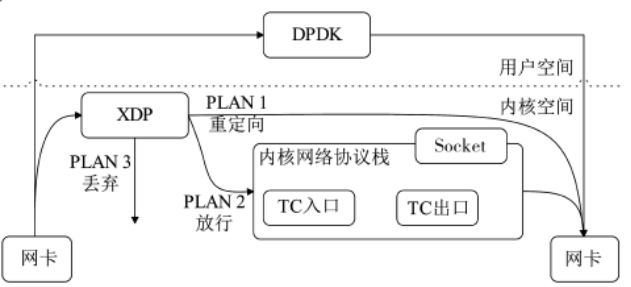

图 2 eBPF/XDP 与 DPDK 包网络路径对比图

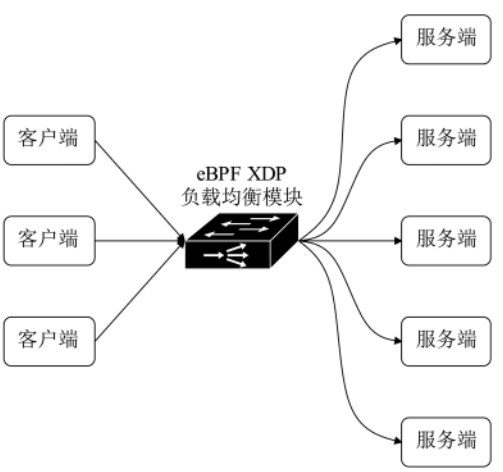

图 3 负载均衡场景架构图

XDP 程序数据包处理主要流程如下,如图 4 所示:

(1)当网卡收到网络包时,XDP 程序提取数据包头字段信息(如 IP、MAC、Port、Proto 等);

(2)根据字段信息作相应处理,如修改转发地址、丢弃等。

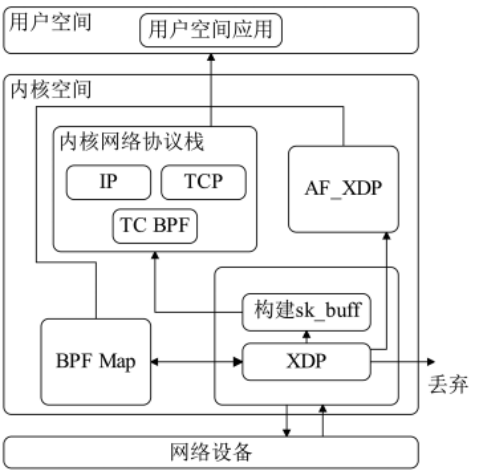

图 4 eBPF/XDP 程序包处理流程

当程序被执行时,会被系统传递一个 struct xdp_md *ctx 对象,在这个对象中包含了指向原始包数据的指针以及接收方网卡接口的字段信息。对这个数据包进行解析后,XDP 程序可以读取 ctx 对象中的信息字段,还可以通过 1.1.4 小节中介绍的 BPF map 定义和访问自定义的数据、通过 BPF Helper 函数操作内核指令。

在此基础上,eBPF/XDP 程序就可以对数据包进行修改、添加、删除,提供了修改包数据的地址后转发的能力。最后,对该数据包进行目标的分配,比如丢弃、通过网卡重发、进入内核网络协议栈,以及将包重定向到指定的网卡、CPU、用户态 Socket。

综上所述,XDP 程序处理数据包大体分为四个步骤,分别是读取包数据、处理元数据、重写包数据、确定包去向。而在典型负载均衡应用中,思想上也大体按照这四个步骤来处理。在图 3 所示的典型负载均衡环境中,eBPF 程序执行在负载均衡器这一部分,通过 eBPF 程序来实现负载均衡的能力。

接下来,使用 XDP 程序来进行负载均衡,修改数据包头。在 struct xdp_md 中,在 XDP 处尚未存在 SKB(Socket Buffer)数据结构,如图 5 所示。

因此只能通过指针来对网络数据报文进行访问,TCP/IP 的报文格式如图 5 所示,分为以太网头、IP 头以及 TCP 头等部分,对比 struct xdp_md 结构,可以通过指针将网络数据报文进行解析,获取、修改报头数据。内核态程序模型方案如下:

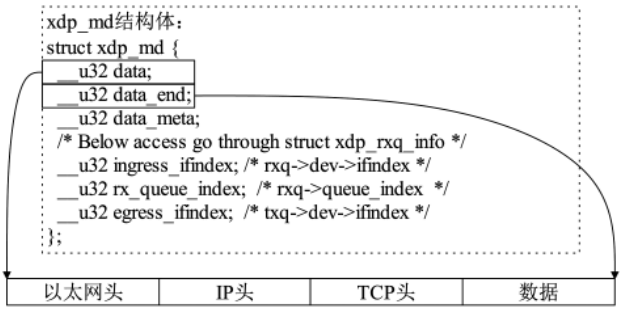

图 5 移动指针操作 xdp_md 结构体

PLAN A:移动指针解析截获的网络数据报文的报头,判断目的地址,如果是从客户端发向负载均衡器的请求,将其目的地址修改为后端服务器的地址,然后将原地址修改为负载均衡器的地址。此时,可以自定义负载均衡策略为轮询或随机等。

PLAN B:移动指针解析截获的网络数据报文的报头,判断目的地址,如果是从服务端返回的响应,则将目的地址修改为客户端的地址,然后将原地址修改为负载均衡器的地址。

综上所述,使用 libbpf 开发 eBPF/XDP 负载均衡程序方案模型为:

(1)使用 bpftool 工具生成内核数据结构定义头文件。

(2)使用 C 语言编写内核态 eBPF 负载均衡程序。解析截获的网络数据报文的报头,判断目的地址:①如果是从客户端发向负载均衡器的请求,将其目的地址修改为后端服务器的地址,然后将原地址修改为负载均衡器的地址,此时,可以自定义负载均衡策略为轮询或随机等;②如果是从服务端返回的响应,则将目的地址修改为客户端的地址,然后将原地址修改为负载均衡器的地址。

(3)将 C 语言编写内核态 eBPF 程序编译为 BPF 字节码。

(4)使用 bpftool 为 BPF 字节码生成脚手架头文件。

(5)用户态程序引入步骤(4)中生成的头文件,开发 eBPF 程序加载、挂载到内核函数的用户态程序。

2.3 基于 eBPF 的安全场景网络功能虚拟化典型应用方案

在安全领域，黑名单源 IP 校验与快速丢包是一个较为经典的案例。同样地，使用 eBPF/XDP 和 BPF map，还可以在网络路径的早期完成黑名单 IP 的校验、数据包的丢弃，开发 eBPF/XDP 程序的步骤与负载均衡类似，只不过把核心逻辑从负载均衡场景下的根据负载均衡策略替换数据包的源、目标地址改变为根据 BPF map 中的黑名单丢弃源 IP 命中的数据包，这一核心逻辑细化为：

(1)捕获 XDP 处的数据包，通过指针移动解析数据报头，判断包类型是否满足要求，若不满足，直接执行 XDP_PASS，将数据包交由内核继续处理；

(2)指针移动解析数据报头获取源 IP 地址，与 BPF map 中黑名单地址校验；

(3)若数据报头源 IP 地址与 BPF map 中黑名单地址相符，则进行 XDP_DROP 丢弃；

(4)若不存在于 BPF map 中黑名单地址中，则执行 XDP_PASS，将数据包交由内核继续处理。

相较于在繁琐复杂冗长的内核协议栈中处理，通过上述 eBPF 程序在网络路径最开始的 XDP 处进行处理，如图 6 所示，就可以使得黑名单校验丢包的性能达到较优的效果。

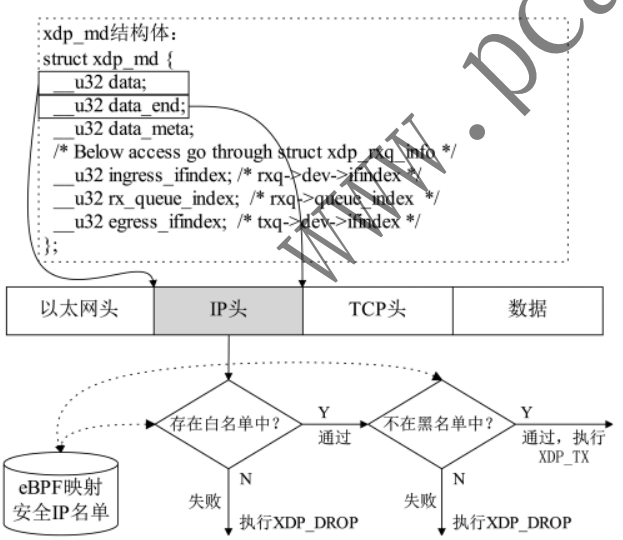

图 6 eBPF 安全场景应用流程

2.4 基于 eBPF 的限流场景网络功能虚拟化典型应用方案

基于 XDP eBPF 程序的云数据中心限流场景的核心目标是限制请求的数据包的 QPS，防止资源过载。处理方式是，通过捕获 XDP 处的数据包，将单位时间内的数据包流量个数、速率记录在 BPF map 中，记录为当前系统入口的数据包流速。若及时流量达到某个预先设定的门限值，则对后续收到的数据包执行 XDP_DROP 操作，对后续收到的数据包执行丢弃操作；反之，若及时流量未达到门限值，则正常执行 XDP_PASS 操作，将其将放行进入内核协议栈继续处理或重定向到其他地方。

具体步骤细化如下：

(1)通过 eBPF 程序在 XDP Hook 处截获数据包。

(2)将单位时间内的数据包流量个数、速率记录在 BPF map 中。并实时将统计数据更新到 eBPF 映射中，及时更新 eBPF map 中的及时流量速率值。

(3)对比 eBPF map 中的及时流量速率值与门限值，如果及时流量达到某个预先设定的门限值，则对后续收到的数据包执行 XDP_DROP 操作，对后续收到的数据包执行丢弃操作；反之，若及时流量未达到门限值，则正常执行 XDP_PASS 操作，将其将放行进入内核协议栈继续处理或重定向到其他地方，如图 7 所示。

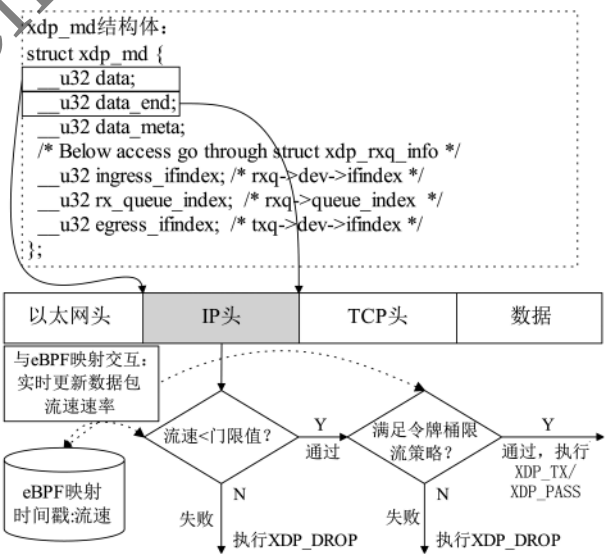

图 7 eBPF 限流场景应用流程

2.5 基于 eBPF 的网络功能虚拟化典型应用方案总结

综上所述，根据负载均衡、安全、限流这些 eBPF 用于网络领域的典型应用分析可得，XDP 作为 eBPF 最重要的 Hook 之一，它的能力在于做数据包处理的位置在整个网络路径中靠前，使得网络数据

包的处理速度受到对整个网络路径短路处理效率的加速优化。

但是,除此以外,仔细分析 XDP 的能力以及其现状,可以得出另一个结论,即 XDP 的能力其实是受限的。它在快速丢包与数据包级别重定向方面的效率很高,但从某种角度来说,它也受限于这上述方向的应用,期待后人能探索出 XDP 更多的应用。

3 基于 eBPF 的云原生网络功能应用研究

3.1 基于 eBPF 的云原生网络功能应用概述

随着云计算的发展进入云原生时代,Kubernetes 成为社区热点,受到企业、开发者、学术研究者的关注,逐步成为云时代的操作系统。本节对 eBPF 技术在云原生网络方面的应用进行研究,主要关注于 Kubernetes 的容器网络。kube-proxy 基于 Linux 内核的 Iptables 等功能开发,部署在 Node 节点上,是 Kubernetes 组件中用于网络方面的一个重要组件,用于 Kubernetes Service 的通信与负载均衡机制。

然而,使用 Iptables 或其他相关技术的 kube-proxy 效率并不高,深究其原因,在于 Iptables 处于内核网络协议栈的较深处,需要在 Linux 内核协议栈里按步骤操作,包转发的路径非常长:

(1)网卡收到包通过 DMA 放到 Ring-Buffer;

(2)包经过 XDP Hook 点后内核给包分配内存,此后生成 SKB(Socket Buffer)结构,也就是网络数据包的内核结构体表示;

(3)将包送到内核网络协议栈,经过 GRO 处理,对分片包进行重组;

(4)包进入 Traffic Control 的 Ingress Hook;

(5)在 Netfilter 繁琐而又冗长的处理点中处理,在 PREROUTING、FORWARD、POSTROUTING 的 Iptables 规则中处理,而后到达 TC Egress Hook 点,进行出方向的判断;

(6)判断包是发送到一个本机 Veth 设备,或者一个本机 Service Endpoint,或者是通过网卡发出去到主机外的目标 IP。

综上所述,网络数据包在整个网络路径 DataPath 中路径较为繁琐冗长,效率较低,从而牵连使用 Iptables 的 kube-proxy 降低了包处理的效率。

为了解决网络数据包在繁琐冗长的网络路径中耗时长的问题,Kubernetes 引入使用 eBPF 技术加速了包的处理,接下来分析 eBPF 技术在 Kubernetes 容器网络中的应用。

3.2 基于 eBPF 的容器网络优化解决方案简析

随着云原生领域的发展,容器网络技术成为了云原生领域的热点之一,本节所研究的容器网络主要基于 Kubernetes 架构。在容器网络中的流量穿过宿主机内核网络数据路径的过程中,eBPF 程序可以做很多处理,如图 8 所示。

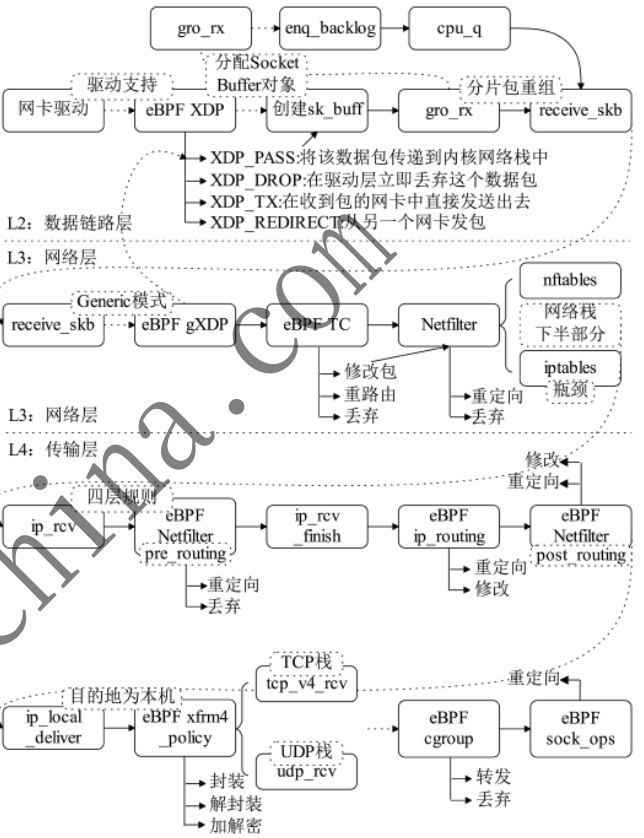

图 8 eBPF 程序操作数据包内核网络路径

在图 8 所示的内核网络数据路径中,如果网卡支持 XDP,eBPF 程序可以运行在 XDP 处,返回结果可以是 PASS、TRANSMIT、DROP。其中 PASS 是使得数据包按照默认的内核网络路径继续处理;DROP 是将该数据包直接丢弃,在 2.3 节中就对 DROP 方法进行了运用,在 XDP 处对数据包进行了丢弃处理,避免了该包穿越到后续冗长的内核网络协议栈后才被丢弃,大大增加了快速丢包的效率,提升了该场景下内核处理包的性能;Transmit 则在 2.2 节中应用过,用于对包进行修改。

如果返回的是 PASS,则会交由内核继续处理,执行 clean_rx()方法进行 SKB(Socket Buffer)对象的创建,接着交由 GRO 对分片的包重组后交给上层。

这里按默认没有分片重组的路径看，数据包将在receive_skb()处理后来到通用XDP点，这里的通用XDP点是网卡不支持XDP的情况下会直接来到的点。此后会来到第二个eBPF程序执行点：TC，它的功能非常强大，可以提供多种对数据包的处理能力，比如修改数据包、重路由、丢弃包等。

然后，数据包会进入二层Netfilter，如果Netfilter中的Iptables模块中配置了较多的规则，那么这里的效率将会较差。接着交由四层Netfilter处理，会经过pre_routing()、ip_routing()、post_routing()等多个步骤，然后完成包的封装和加解密，最终将包交由TCP或UDP协议栈，将SKB(Socket Buffer)放到socket_receive_queue中。

在该步骤接收数据后，将来到第三与第四种eBPF程序的处理点，分别是cgroup eBPF程序和SOCK_OPS eBPF程序，其中cgroup eBPF可以实现Socket层级的负载均衡，可以满足客户端应用的无感知化、内置化的负载均衡逻辑；而SOCK_OPS eBPF则可以用于Socket层限流，可以用于客户端级别的限流，除此以外，还可以应用于如2.2节中的负载均衡场景，与之不同的是，这里的SOCK_OPS eBPF程序需要依赖原有负载均衡器的通用能力，而不是像XDP那样完整地实现负载均衡的功能，通过SOCK_OPS只是对原有负载均衡器性能的加速，其原理与下面一节要讲的服务网格边车流量加速有一些类似[2]。

综上所述，总结来说有四个eBPF程序作用显著的处理点，分别是：

(1)XDP：eBPF程序可以运行在XDP处，可以对数据包进行放行、丢弃、修改，返回结果可以是PASS、TRANSMIT、DROP。XDP位于网络路径最靠前的位置，在XDP处理的性能最好，处理上述应用外，通常还可用于防火墙或者四层负载均衡等应用，例如第2节中的应用。

(2)TC：TC程序可以在网卡队列收发包时进行触发执行，可以提供修改数据包、重路由、丢弃包等能力，可以用于流量控制等场景之中。

(3)cgroup：cgroup程序支持在cgroup内所有进程的Socket创建、修改选项、连接等场景时执行，可以用于过滤、控制cgroup内所有进程的Socket；可以实现Socket层级的负载均衡，可以实现客户端应用的无感知化、内置化的负载均衡逻辑等能力。

(4)Socket：Socket程序在套接字的多个变动场景下发挥作用，比如在收发数据、创建、修改等行为时进行触发执行，可以用于过滤、观测、重定向数据包，经典的类型是PF_PROG_TYPE_SOCK_OPS。可以用于Socket层限流，如客户端级别的限流；在其他场景下还可以用作负载均衡器的加速、服务网格边车流量加速等。

综上所述，容器网络加速核心思想在于，在数据包的处理过程中，尽可能地在XDP&TC对进出集群的南北向流量进行短路处理，而在Socket处对集群内的东西向流量进行处理[7]。通过SOCK_OPS eBPF程序来将套接字信息存储在BPF map中，然后在SENDMSG系统调用时，拦截请求对收发方的IP地址进行变换从而将当前套接字直接转发给BPF map中的套接字。这种东西向流量的核心处理思想达成了同一节点内流量转发效率得到显著提升的效果，结合TC&XDP对进出集群的南北向流量进行短路处理的方案，协作共同提升了容器网络的包处理性能。

3.3 面向服务网格的容器网络性能优化方案简析

服务网格可以理解为是新一代的微服务架构，它是一个专门的基础设施层，可以用于管理服务与服务之间的通信。它所支持的能力是将服务与服务之间的通信逻辑、重试、超时等行为从单个的服务中抽离出来，使得此时模拟存在一个独立的基础设施层，来替业务服务来完成这些类似于通信的功能。在服务网格的场景下，基础设施层形成了一个大型的阵列，而这些业务服务的代理则称为边车代理(SideCar)，这些边车代理是与业务服务共存的。

在服务网格之前，服务与服务之间的通信需要其本身来完成，现在则可以通过边车代理来完成，Consumer服务的代理与Provider服务的代理之间直接通信，而其配置则由服务网格的控制平面来进行配置，使其可以透明拦截出入流量，最终，这些边车代理基础设施层与业务服务本身的集合形成了一个网络网格，而这个网络网格则称为服务网格。除此以外，边车还可以提供其他能力，比如日志收集、服务监控、服务治理等，服务网格使得微服务架构更成体系化。

然而，这种架构模式也存在缺陷，比如业务服务与边车之间多一跳的网络通信开销。即使在同一个Node里，也会重复经历内核网络协议栈Iptables、TCP/IP栈等冗长的包处理过程。

而这一弊端则可以通过eBPF程序来进行加速

优化,在优化前边车代理与业务服务之间的网络通信都需要经过内核网络协议栈 Iptables、TCP/IP 栈等冗长的包处理路径,而通过 eBPF 程序优化后边车代理与业务服务间的网络通信则可以绕过冗长的内核网络协议栈 Iptables、TCP/IP 栈[8],如图 9 所示。

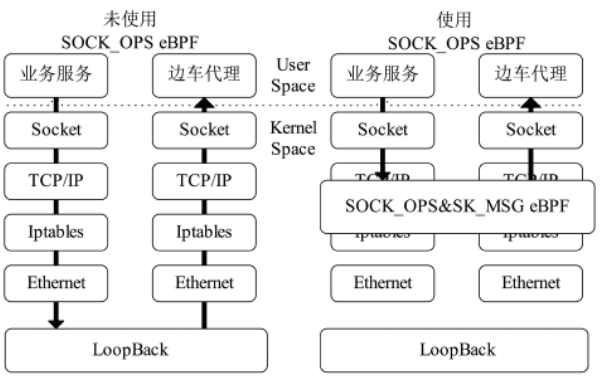

图 9 SOCK_OPS eBPF 程序服务网格加速

其原理是通过 BPF_PROG_TYPE_SOCK_OPS 类型的 SOCK_OPS eBPF 程序监听 Socket 事件,并将Socket 的信息保存在 SOCKHASH 中,如图 10 所示。这里 SOCKHASH 是 BPF map 的一种,是 BPF_MAP_TYPE_SOCKHASH 类型的 BPF 映射。

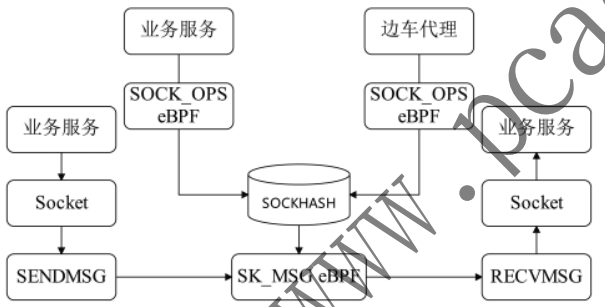

图 10 SOCK_OPS eBPF 程序服务网格加速原理

SOCK_OPS eBPF 程序可以与 SOCKHASH 进行配合来完成数据交互,它所存储的 key 是四元组(源 IP、目的 IP、源端口、目的端口),而 SK_MSG eBPF 程序则会将 SENDMSG 系统调用拦截,在拦截后会在 SOCKHASH 中查找对端 Socket 是否存在,如果存在,则会调用 bpf_msg_redirect_hash 直接将数据发送给对端 Socket,通过这种方式加速服务网格的 SideCar 网络通信的性能。

4 结论

随着 eBPF 的火热发展,其应用已遍布于网络功能虚拟化领域以及云原生网络功能领域。本文抛砖引玉,针对 eBPF 技术在上述领域的发展与应用展开分析,剖析了 eBPF 技术分别在虚拟化网络功能领域以及云原生领域所使用的基本原理,介绍了 eBPF技术在上述领域各自的应用概述,并举出了eBPF 用于上述领域的典型应用:网络功能虚拟化领域的负载均衡、快速丢包、限流应用,以及云原生网络功能领域的 Kubernetes 容器网络加速、服务网格加速应用。针对上述应用进行了分析以及原理的剖析,在力求经典、抛砖引玉的出发点上介绍了上述应用的基本原理以及应用场景,希望读者读完本文后可以对 eBPF 技术本身的原理以及 eBPF 在虚拟化网络功能领域和云原生领域的应用有了一个初步的、全貌性的了解。

参考文献

[1] BORKMANN D. eBPF and Kubernetes: little helper minions for scaling microservices[Z]. KubeCon, 2020.

[2] Nate Sweet. Understanding (and Troubleshooting) the eBPF Datapath in Cilium[Z]. KubeCon, 2019.

[3] 赵慧玲,解云鹏,史凡.网络虚拟化及网络功能虚拟化技术探讨[J].中兴通讯技术,2014,20(3):8-11.

[4] 周伟林,杨芫,徐明伟.网络功能虚拟化技术研究综述[J].计算机研究与发展,2018,55(4):675-688.

[5] 刘渊,乔巍.云环境下基于 Kubernetes 集群系统的容器网络研究与优化[J].信息网络安全,2020,20(3):36-44.

[6] HILAND-JRGENSEN T, BROUER J D, BORKMANN D, et al. The eXpress data path: fast programmable packet processing in the operating system kernel[C]//The 14th International Conference, 2018.

[7] BORKMANN D, PUMPUTIS M. K8s Service Load Balancing with BPF & XDP[C]//Linux Plumbers Conference, 2020.

[8] How to use eBPF for accelerating cloud native applications[EB/OL]. (2021-01-28)[2021-01-28]. https://cyral.com/blog/how-to-ebpf-accelerating-cloud-native/.

(收稿日期:2022-12-04)

作者简介:

施苏峰(1998-),通信作者,男,硕士研究生,主要研究方向:eBPF、云原生、网络功能虚拟化。E-mail:1257989860@qq.com。

版权声明

凡《网络安全与数据治理》录用的文章，如作者没有关于汇编权、翻译权、印刷权及电子版的复制权、信息网络传播权与发行权等版权的特殊声明，即视作该文章署名作者同意将该文章的汇编权、翻译权、印刷权及电子版的复制权、信息网络传播权与发行权授予本刊，本刊有权授权本刊合作数据库、合作媒体等合作伙伴使用。同时，本刊支付的稿酬已包含上述使用的费用，特此声明。

《网络安全与数据治理》编辑部