电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

学士学位论文

BACHELOR THESIS

论文题目 基于 BPF 和 XDP 改善容器网络的性能

专　　业 网络工程

学　　号 **2014010901005**

作 者 姓 名 李啸宇

指 导 教 师 任丰原 教授

摘　要

容器网络应用于数据中心，性能要求非常高。本文首先分析传统基于 iptables 容器网络的性能瓶颈，指出基于 BPF 容器网络的优势；然后在基于 BPF 已有方案的基础上进行改进，并结合 XDP，进一步提升容器网络的性能；最后通过实验，验证基于 BPF 本文方案的可行性以及 BPF+XDP 对容器网络性能的进一步提升作用。实验结果表明，基于 BPF 的本文方案优于已有方案，BPF+XDP 能够进一步提升容器网络的性能。

关键词： Docker 容器平台，iptables，BPF，XDP，网络性能测试

ABSTRACT

Container network is applied in data center and requires high performance. Firstly, the article analyses bottleneck of traditional container network based on iptables and indicates the advantages of that based on BPF. Then, the article makes improvement of existing design based on BPF and further improves the performance combined with XDP. Finally, experiments are conducted to verify the feasibility of BPF-based design in this article and the further improvement of BPF+XDP. The result shows that the BPF-based design in this article outperforms the existing one and that BPF+XDP can further improve the performance of container network.

Keywords: Docker container platform, iptables, BPF, XDP, network performance measurement

目 录

第一章 绪 论.....1
1.1 课题背景.....1
1.2 课题的价值及意义.....1
1.3 课题的研究现状.....2
1.4 本文主要工作.....3
1.5 论文组织结构.....3
第二章 相关技术概述.....4
2.1 Docker 容器平台.....4
2.1.1 平台概述.....4
2.1.2 网络驱动.....6
2.1.3 命名空间.....9
2.2 传统容器网络.....10
2.2.1 NetFilter 的架构.....10
2.2.2 iptables 的规则匹配过程.....11
2.3 BPF 与 XDP.....12
2.3.1 网络数据包路径.....12
2.3.2 BPF.....13
2.3.3 XDP.....15
2.4 本章小结.....16
第三章 容器网络性能提升的方案设计.....17
3.1 iptables 的性能分析.....17
3.2 基于 BPF 已有方案的分析.....17
3.3 基于 BPF 的方案改进.....20
3.3.1 总体思路.....20
3.3.2 容器网络的创建.....21
3.3.3 BPF 程序设计.....23
3.4 基于 XDP 的方案设计.....24
3.5 本章小结.....26
第四章 方案实现与实验测试.....27
4.1 方案的实现过程.....27

4.1.1 软硬件环境的配置 27
4.1.2 容器网络的搭建 28
4.1.3 数据包处理规则的配置 29
4.2 测试工具的选择 30
4.3 实验验证与结果 32
4.3.1 方案的比较 32
4.3.2 BPF+XDP 性能提升作用的验证 34
4.4 本章小结 39
第五章 结束语 40
5.1 总结 40
5.2 展望 40
致 谢 41
参考文献 42
外文资料原文 44
外文资料译文 47

第一章 绪 论

在数据中心网络中，需要将大型业务分成一个个轻量级、松散耦合、协作运行的应用，提高业务布署的灵活性。容器技术通过命名空间使直接运行在操作系统上的不同容器所需的资源相互隔离，满足了轻量级和松散耦合的需求。

为保证协作运行，容器之间需通过容器网络来传递信息。在传统的容器网络中，容器均连接到虚拟桥接接口，与主机内部的其它容器相互通信；同时，为保证容器与主机外部网络设备的通信，网络协议栈上须配置 iptables 规则。但是，在庞大的数据中心网络中，使用 iptables 规则会导致容器网络性能的大幅下降；通过 BPF 和 XDP 处理网络数据能够有效减少容器网络的性能损失。

1.1 课题背景

近年来，数据中心网络的发展给我们的生活提供了很多便利。在数据中心网络中，一个业务往往要承载用户管理、数据查询等多种功能，因此，需要将大型业务分解为一个个轻量级、松散耦合、协作运行的应用，以方便业务的布署、投产和扩展。容器技术能够满足这一需求，因为容器直接运行在操作系统之上，运行一个容器相当于运行一个进程，因此，与传统的基于管理程序的虚拟技术相比，容器技术更加轻量化；此外，容器技术通过命名空间将所需的资源相互隔离，使得容器之间松散耦合，互不干扰。

为保证容器的协作运行，容器之间需要通过容器网络传递业务信息。在传统的容器网络中，每个容器均与虚拟接口相连，在主机内部通信；与此同时，为保证主机外部的设备能够访问内部的容器，网络协议栈上须配置 iptables 规则。然而，在庞大的数据中心网络中，配置 iptables 规则会导致容器网络性能的显著下降，因为数据中心网络中往往有成千上万的容器在运行，这意味着每台主机所配置的 iptables 规则数非常庞大，过多的 iptables 规则会使容器网络的性能遭受巨大损失。

1.2 课题的价值及意义

首先，容器网络是计算机领域的一大研究重点和热点。为满足信息技术飞速发展的需求，数据中心网络的规模需不断扩大，这要求容器网络的性能不断提升。如今，作为数据中心网络的关键部分，容器网络成为了计算机领域的一大研究重点和热点。

其次，容器网络的性能问题将阻碍数据中心网络的发展。通过 iptables 规则来

处理容器网络的业务数据会使容器网络的性能遭受巨大损失，不利于用户数据和管理数据的高效处理，进而阻碍数据中心网络的进一步发展。因此，容器网络的性能问题亟待解决。

此外，容器技术的研究顺应云计算发展的大趋势。目前，容器技术已使云计算应用可在不同的平台之间无缝迁移；容器技术的进一步研究可进一步推动云计算发展的潮流。

总而言之，容器技术具有很大的应用前景，蕴藏巨大的应用价值，容器网络性能的提升意义重大。

1.3 课题的研究现状

容器技术在云计算的大背景下发展而来。2006 年，云计算的概念首次提出，此后在各大高校推广；2011 年，美国国家标准与技术研究院提出了云计算的定义[1]。然而，随着云计算平台的规模不断扩大，资源利用率低、调度分发缓慢、应用与平台无法解耦等问题不断凸显[2]。为了解决这些问题，Solomon Hykes 带领团队研发自己的容器技术，并将此研发项目命名为“Docker”。Docker 是封装整个软件运行环境、用于构建、发布和运行分布式应用的可移植、简单易用的容器平台，通过命名空间技术实现容器资源的隔离[2]。

为实现彼此的相互通信，容器之间需要通过容器网络相互连接。传统的容器网络通过 iptables 处理容器网络的业务信息，保证外部设备能够访问内部的容器。然而，基于 iptables 的容器网络存在性能瓶颈。早在 2003 年，Daniel Hoffman 等学者通过实验测试了 iptables 规则数的变化对网络吞吐量、网络延时的影响[3]，从他们的研究可以看出，在链路带宽较高的情况下，随着 iptables 规则数的增加，网络吞吐量大幅下降，网络延时显著上升。在数据中心网络中，容器网络的规模非常大，每台主机所配置的 iptables 规则往往非常多，这容易导致容器网络性能的严重受损。

2016 年，Daniel Borkmann 带领团队开发了 cilium 开源项目[4]，通过 BPF 技术，在 Linux 系统的 TC 模块处理数据包，提升容器网络的性能。2017 年，N. de Bruijn 利用开源项目 cilium，通过实验，将 iptables 容器网络和 BPF 容器网络进行性能比较[5]，发现基于 BPF 的容器网络在性能上优于基于 iptables 的容器网络。与此同时，在 2016 年，XDP 技术出现[6]。XDP(eXpress Data path) 使 BPF 程序直接访问 DMA 缓存的数据包，在数据包进入 TC 模块前就能得到处理。

由于 XDP 技术出现的时间较晚，目前没有学者研究 XDP 对容器网络性能的提升作用。此外，cilium 开源项目所实现的功能较为复杂，直接通过 cilium 研究

BPF 对容器网络性能的提升可能会引入误差。

1.4 本文主要工作

在已有文献的基础上，本文主要进行以下研究工作：

- 对基于 BPF 的已有方案进行改进。本文分析基于 BPF 已有方案[5] 的不足之处，并据此对已有方案进行改进；
- 结合 XDP，进一步提升容器网络的性能。在改进方案的基础上，本文结合 XDP，进一步设计容器网络性能提升方案，并验证 BPF+XDP 对性能的进一步提升作用；
- 实验验证。本文测试了基于 BPF 的本文方案与已有方案的容器网络吞吐量，以及基于 iptables、BPF 和 BPF+XDP 容器网络的吞吐量与网络延时，以验证本文方案的可行性和 BPF+XDP 的性能提升作用。

1.5 论文组织结构

本文的篇章结构如下：

- 第二章：相关技术概述。本章介绍 Docker 容器平台、传统容器网络以及 BPF、XDP 的背景知识；
- 第三章：容器网络性能提升的方案设计。本章分析基于 iptables 容器网络的性能瓶颈，指出基于 BPF 容器网络的性能优势；然后在基于 BPF 已有方案的基础上进行方案的改进；最后，结合 XDP，进一步设计性能提升方案；
- 第四章：方案实现与实验测试。本章根据上一章的方案设计思路，详细阐述方案的实现过程，并通过实验，测试基于 BPF 已有方案和本文方案的容器网络吞吐量以及基于 iptables、BPF 和 BPF+XDP 容器网络的吞吐量和网络延时，验证本文方案的可行性以及 BPF+XDP 对容器网络性能的提升作用；
- 第五章：结束语。本章总结了本文的研究工作，并对本课题进行展望。

第二章 相关技术概述

本章重点介绍 Docker 容器平台、传统容器网络的数据包处理、BPF 以及 XDP 的相关知识。

2.1 Docker 容器平台

Docker 是用于构建和运行分布式应用的容器平台，通过命名空间实现资源隔离，使每个小应用能够在相互独立的“容器”中运行。本节首先对 Docker 容器平台进行概述，然后介绍容器网络驱动和命名空间技术。

2.1.1 平台概述

Docker 平台通过命令行终端为用户提供交互的界面。用户可通过命令的配置创建容器网络。每个容器均运行较小的应用，这些应用组合在一起，共同支撑整个网络的业务。从外部网络的角度来看，用户访问的是主机本身，其背后隐藏着若干个容器；而实际上，用户访问的是主机上的容器。为保证外部网络能够访问内部的容器，主机上需对静态 NAT 进行配置。在用户访问过程中，为实现信息的共享，容器之间也有互访的过程，此时，一个容器是客户端，另一个容器为服务器。

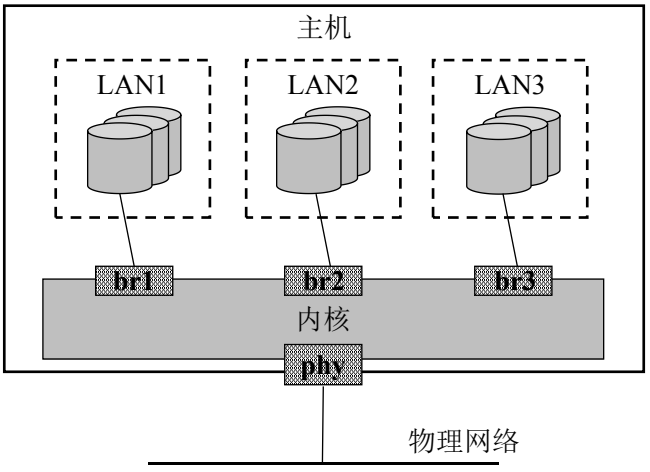

图 2-1 单主机容器网络示意图

当应用需求简单、容器网络规模不大的时候，可以考虑在一台主机上布署容器网络。该网络的拓扑大致如图2-1所示。图中 phy 为主机的物理接口，与物理网络相连。主机上运行了若干个容器，通过命名空间技术对容器的资源进行隔离，使主机上的容器相对独立地运行。容器通过虚拟的网桥接口与内核相连，组成若干

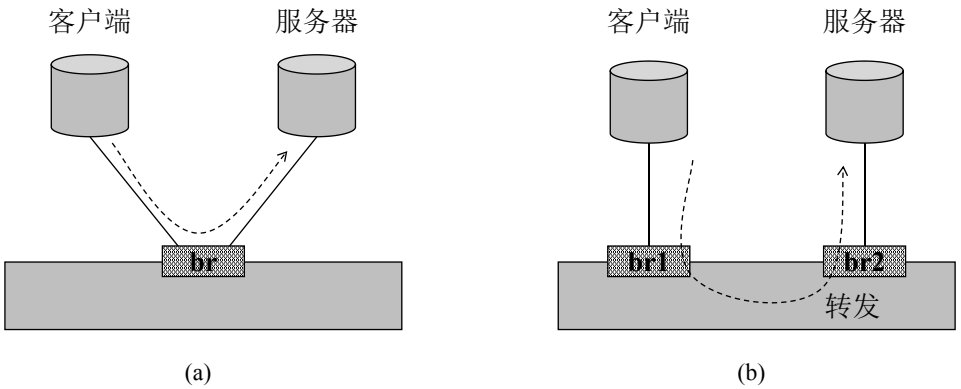

图 2-2 单主机容器网络的互访过程：(a) 局域网内部的容器访问过程；(b) 局域网之间的容器访问过程

个局域网。容器互访的过程如图2-2所示。若局域网内部的容器相互访问，则两容器直接在数据链路层通信，不需要经过内核转发（如图2-2(a)所示）；若局域网之间的容器相互访问，则客户端产生的数据由系统转发至服务器（如图2-2(b)所示）。

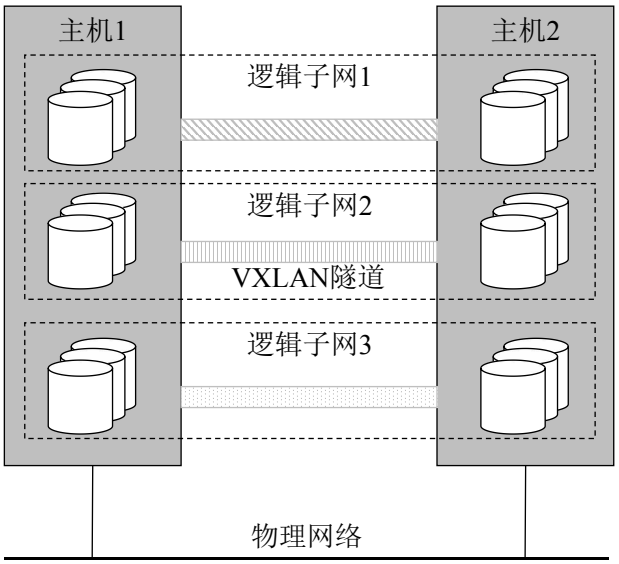

图 2-3 多主机容器网络示意图

当应用较为复杂、网络规模较大的时候，需要在多台主机上布署容器网络。多主机容器网络的拓扑大致如图2-3所示。网络中有两台主机通过物理网络相连，主机 1 和主机 2 上都运行有若干个容器。大型的数据中心网络往往是多租户的，为将所有租户所部署的容器网络隔离开，需要通过 VXLAN 协议在已有的物理拓扑上构建相互隔离的逻辑拓扑。VXLAN (Virtual eXtensible Local Area Network)[7] 意为“虚拟可扩展局域网协议”，旨在实际的物理拓扑之上建立逻辑网络拓扑。

VXLAN 协议通过隧道技术，将虚拟机的数据链路层报文封装在主机的 UDP 报文中，并通过 VNI (VXLAN Network Identifier) 标识不同的虚拟子网。多主机容器网络同样有容器之间的互访过程。若客户端容器跨主机访问同一逻辑子网中的服务器容器，客户端容器产生的报文被所在的主机封装在 VXLAN 报文中，在物理网络中传输。运行服务器容器的主机将报文解封，并将容器的报文交付给对应的容器（如图2-4所示）。

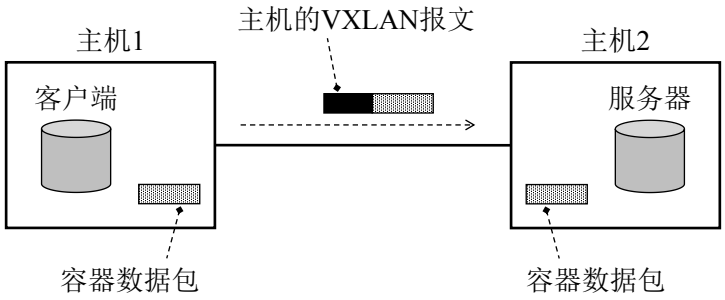

图 2-4 多主机的容器互访过程

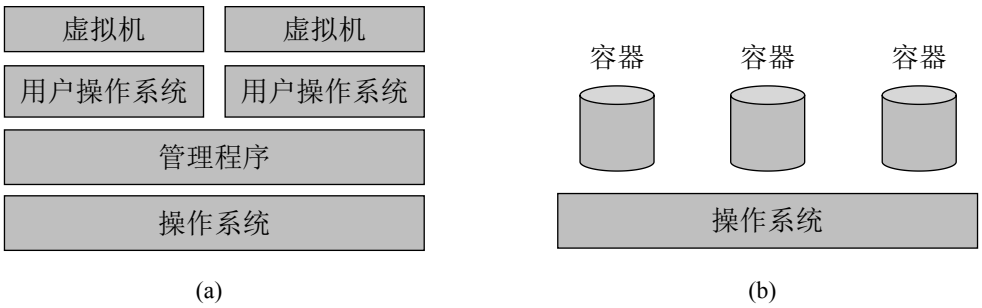

图 2-5 传统虚拟技术与容器技术的对比：(a) 传统虚拟技术示意图；(b) 容器技术示意图

与传统的虚拟技术相比，容器技术更加轻量化[8]。传统的虚拟技术是基于管理程序的，虚拟机运行在用户操作系统上，并由管理程序统一管理（如图2-5(a)所示）。管理程序和用户操作系统在虚拟机通信时引入较大的开销。而在容器技术中，容器直接运行在操作系统之上，直接利用主机提供的运算资源，相当于操作系统的进程（如图2-5(b)所示）。因此，当今的数据中心网络主要基于容器技术实现网络的虚拟化。

2.1.2 网络驱动

Docker 平台提供了多种容器网络驱动，以方便用户构建不同种类的容器网络。本小节重点介绍 bridge 驱动、host 驱动及 overlay 驱动。

(1) bridge 驱动

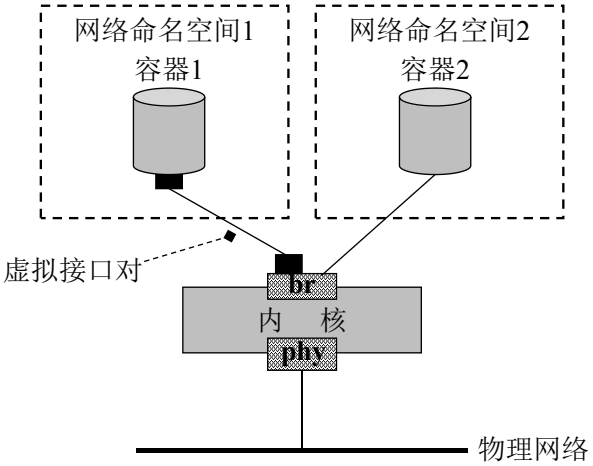

图 2-6 通过 bridge 驱动创建的容器网络

bridge 驱动将容器连接到 Linux 网桥上，使容器之间在数据链路层上就能相互通信，是 Docker 平台最基础的容器网络驱动。通过 bridge 驱动创建的容器网络如图2-6所示。phy 为主机的物理接口，与物理网络相连。br 为 Docker 平台通过 bridge 驱动创建的虚拟网桥接口，容器 1 与容器 2 与网桥接口连接，并运行在不同的命名空间中。在创建容器时，Docker 平台新建一个命名空间以及虚拟接口对，其中一个接口加入到新建的命名空间中，另一个接口则与 br 网桥接口桥接。虚拟接口对相当于一条网线，将容器与虚拟接口连接在一起。作为最基本的网络驱动，bridge 驱动可以满足容器最基本的需求；然而，在复杂场景下，bridge 驱动的使用仍会受到较多限制。[2]

(2) host 驱动

相比于 bridge 驱动，host 驱动则使容器与主机共享网络信息，简化网络的管理。通过 host 驱动创建的容器网络如图2-7所示。内核通过 phy 接口与物理网络相连。容器 1 和容器 2 在内核之上运行。与 bridge 网络中的容器不同，host 网络中的容器与宿主机均在同一个网络命名空间中，因此，它们与主机共享同一个网络协议栈。尽管如此，它们的文件系统、主机名、进程编号等资源仍然相互隔离。由于容器与主机共享同样的网络信息，host 驱动大大简化了虚拟化网络的布署与管理；但是，在大规模容器网络中，host 驱动会引发网络资源冲突，因此，host 驱动仅适用于网络规模不大的应用场景下。

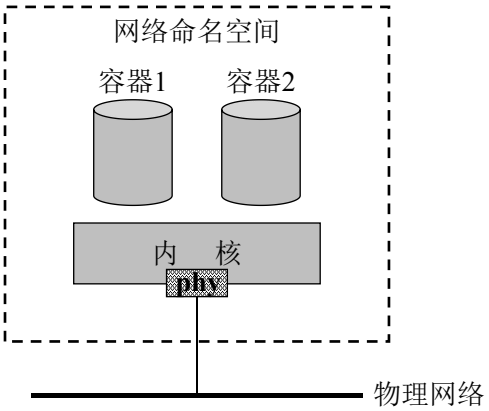

图 2-7 通过 host 驱动创建的容器网络

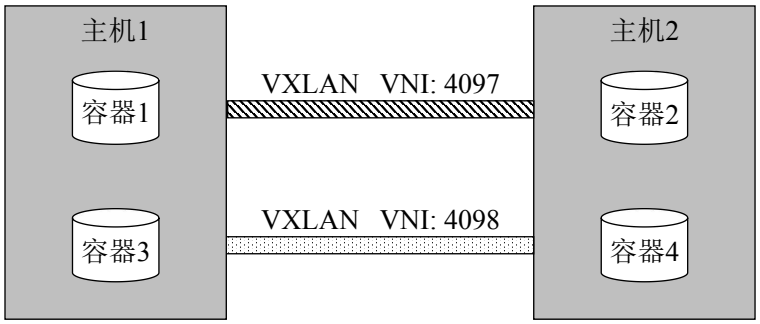

图 2-8 通过 overlay 驱动创建的容器网络

(3) overlay 驱动

overlay 驱动则通过 VXLAN 协议在多主机环境下建立容器网络，该驱动创建的容器网络如图2-8所示。主机 1 和主机 2 均连接在物理网络中，容器 1 和容器 3 运行在主机 1 上，容器 2 和容器 4 运行在主机 2 上。主机 1 和主机 2 之间配置了 VXLAN 隧道，其中容器 1 和容器 2 的 VNI 为 4097，容器 3 和容器 4 的 VNI 为 4098。尽管容器 1 和容器 2 不在同一个主机上，但它们处于同一逻辑子网中，因此它们之间能够相互通信；容器 1 和容器 3 虽然运行在同一个主机上，但由于 VNI 不同，无法相互通信。实际上，在组建此网络时，Docker 平台新建了一个网桥接口和 vxlan 类型的接口，配置好 VXLAN 的相关参数后，将 vxlan 接口桥接到网桥接口中。容器产生的报文经过网桥接口，从而进入 VXLAN 隧道中。与此同时，Docker 平台将此桥接接口加入到新的命名空间中，使其对用户不可见；因此，若直接通过 overlay 驱动创建容器网络，用户无法直接配置此网桥接口的参数。

2.1.3 命名空间

Docker 平台运用了很多内核技术，其中最为关键的是命名空间。命名空间(namespaces)[9] 是 Linux 系统为运行中的进程提供资源隔离机制的内核技术。一个命名空间中的资源仅对它所包含的进程可见，而对其它进程不可见。如今，命名空间广泛应用于容器技术中，使容器能够互不干扰地运行。

Linux 系统中的命名空间包括七种，具体名称及所隔离的资源如表2-1所示。

表 2-1 命名空间种类及隔离资源对照表

命名空间种类	所隔离的资源
主机名命名空间(UTS Namespace)	主机名、域名信息
用户命名空间(User Namespace)	用户标识、用户组标识
进程编号命名空间(PID Namespace)	进程编号信息
进程通信命名空间(IPC Namespace)	进程通信所需资源（包括POSIX消息队列、IPC标识符等）
文件系统命名空间(Mount Namespace)	文件系统挂载点
控制组命名空间(Cgroup Namespace)	cgroups根目录
网络命名空间(Network Namespace)	网络协议栈信息（包括路由表、DNS配置等）

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
int setns(int fd, int nstype);
int unshare(int flags);
```

图 2-9 命名空间系统调用函数的参数列表

与命名空间相关的系统调用函数有 3 个，分别是 clone()、setns() 和 unshare()，它们的参数列表如图2-9所示。其中，clone() 函数根据 flags 值所指定的命名空间种类建立新的命名空间，然后在新的命名空间中创建进程，并在此进程中调用 int fn(void *arg) 函数；当 fn() 函数运行完毕时，进程随之终止。setns() 函数则将当前运行的进程加入到已经存在的命名空间；其中 fd 是文件描述符，用于指定具体的命名空间，nstype 则指明所加入命名空间的种类。与 setns() 函数不同，unshare() 函数的主要功能是新建一个命名空间，然后将当前运行的进程加入到新的命名空间中；其中 flags 参数指明新命名空间的种类。

每创建和运行一个进程，Linux 系统在/proc 目录中新建以该进程编号命名的目录，并在/proc/*[PID]*/ns/目录中为进程的每种命名空间创建一个软链接，为上述系统调用函数提供命名空间配置的入口。此外，为限制每种命名空间的个数，

Linux 系统在/proc/sys/user/目录下建立文件，定义各种命名空间的使用上限，例如，若 max_net_namespaces 中的数值为 15145，则系统的网络命名空间不得超过 15145 个。Docker 平台每新建一个命名空间，就将命名空间信息保存在/var/run/docker 目录下，方便对命名空间的配置和管理。

2.2 传统容器网络

传统容器网络将容器连接到内核的虚拟网桥接口，并通过 iptables 处理容器与主机外部设备交互的网络数据包。iptables 是 NetFilter 防火墙的配置命令，通过五个规则链配置和管理数据包处理规则。它提供了数据包过滤和 NAT 机制，隐藏和保护内部网络，有效抵制来自外部网络的攻击；尽管如此，iptables 在数据中心网络中存在性能瓶颈。本节主要介绍 NetFilter 的架构、iptables 的规则匹配过程，并分析传统容器网络的性能瓶颈。

2.2.1 NetFilter 的架构

NetFilter[10] 是 Linux 系统的防火墙，在系统网络层和传输层上对数据包进行过滤和地址转换操作，有效保护内部网络，抵制外部网络的入侵。

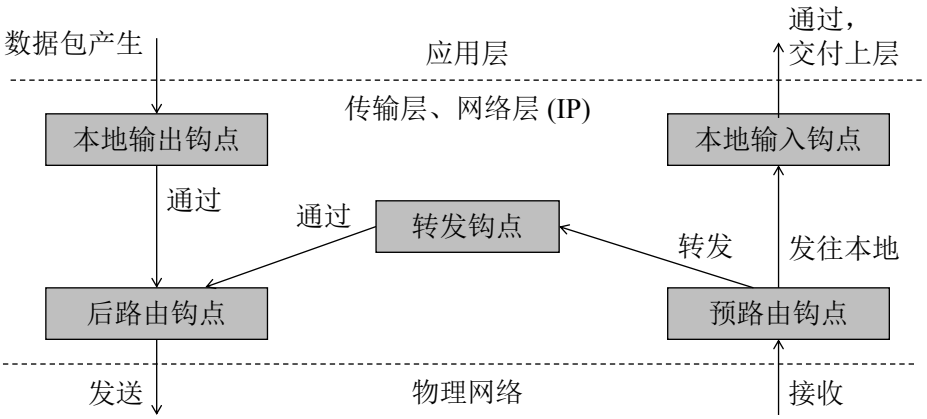

图 2-10 NetFilter 结构示意图[10]

NetFilter 的主要结构如图2-10所示[10]。为保护内部网络，NetFilter 主要检查和处理三类数据包：第一类是由本系统应用层产生的数据包，第二类是发送至本系统应用层的数据包，第三类则是由本系统转发的数据包。为处理这些数据包，Linux 系统在 NetFilter 中设置了以下 5 个钩点：

- 预路由钩点 (PRE_ROUTING)：预路由钩点处理由本系统接收的，但路由信息尚未确定的网络报文，主要应用于目的地址和目的端口的转换；
- 后路由钩点 (POST_ROUTING)：后路由钩点处理即将由本系统发送（或转

发）的网络报文，应用于源地址和源端口的转换；

- 本地输入钩点 (LOCAL_IN)：本地输入钩点处理发送至本地应用的网络报文，处理方式包括过滤数据包和修改数据包的字段等。数据包通过本地输入钩点后，由系统交付给应用进程的套接字；
- 本地输出钩点 (LOCAL_OUT)：本地输出钩点处理由本系统应用层产生的网络报文。路由信息确定后，所产生的报文首先通过此钩点；
- 转发钩点 (FORWARD)：转发钩点处理由本系统转发的网络报文，主要处理方式为过滤数据包和修改数据包的字段等。

应用层产生报文后，Linux 系统首先确定报文的路由信息，然后在本地输出钩点检查该报文，进行过滤、字段修改等处理。通过本地输出钩点的检查和处理后，报文到达后路由钩点，进行源地址和源端口的转换处理。最后，系统将报文发送至物理网络。若系统从物理网络中接收报文，则接收的数据包首先在预路由钩点进行目的地址和目的端口的转换处理。若接收的数据包发送至本地应用，则系统通过本地输入钩点对报文进行检查；若接收的数据包由本系统转发，则该报文经过转发钩点，通过转发钩点的数据报文将经过预路由钩点，并由 Linux 系统发送至物理网络。

2.2.2 iptables 的规则匹配过程

为方便对五个钩点进行配置，iptables 设置了五个规则链，通过四个表来管理这五个规则链。这些表的名称按照优先级从低到高排列如下：

- Filter 表：Filter 表用于配置数据包的过滤规则，使系统将存在安全隐患的网络流量丢弃。Filter 表作用于本地输出、本地输入及转发规则链；
- NAT 表：NAT 表用于配置与网络地址转换相关的规则，隐藏内部网络。NAT 表作用于本地输出、预路由和后路由这三个规则链；
- Mangle 表：Mangle 表用于修改数据报文的字段，主要配置与 ToS (Type of Service)、TTL (Time To Live)、QoS (Quality of Service) 相关的规则。Mangle 表作用于所有规则链；
- Raw 表：Raw 表是优先级最高的表，仅作用于预路由和本地输出这两个规则链。Raw 表也用于网络地址转换，但与 NAT 表不同，若数据包与 Raw 表中的地址转换规则匹配，则系统不对数据包进行跟踪，提高了系统的性能，因此，Raw 表主要应用在接受频繁访问的服务器中。

对于同一个规则链，系统按照优先级从高到低的顺序依次查询上述 4 个表。若在优先级高的表中查询到与数据包信息匹配的规则，则系统根据此规则处理数据包；若优先级高的表中没有匹配的规则，则系统查询优先级较低的表。

在转发过程中，系统所查询的规则个数很大程度上决定了数据包的处理效率，检查过程中系统查询的规则数越多，则数据包的处理效率越低，网络的吞吐量也随之下降。尽管 iptables 能够保证网络的安全性，但由于数据中心网络的规模非常庞大，每台主机所配置的 iptables 规则往往有上千条，使用 iptables 处理数据中心网络的报文将显著影响容器网络的性能。

2.3 BPF 与 XDP

BPF 是在 Linux 的 TC (Traffic Control) 上对数据包进行分类和处理的技术，XDP 则使 BPF 程序直接访问存放数据包的 DMA 缓存，在网卡附近处理由系统接收的数据包，以减少将数据包从 DMA 缓存复制到内核 sk_buff 中的开销。本节将介绍 Linux 系统的网络数据包路径，并在此基础之上介绍 BPF 和 XDP 的工作原理。

2.3.1 网络数据包路径

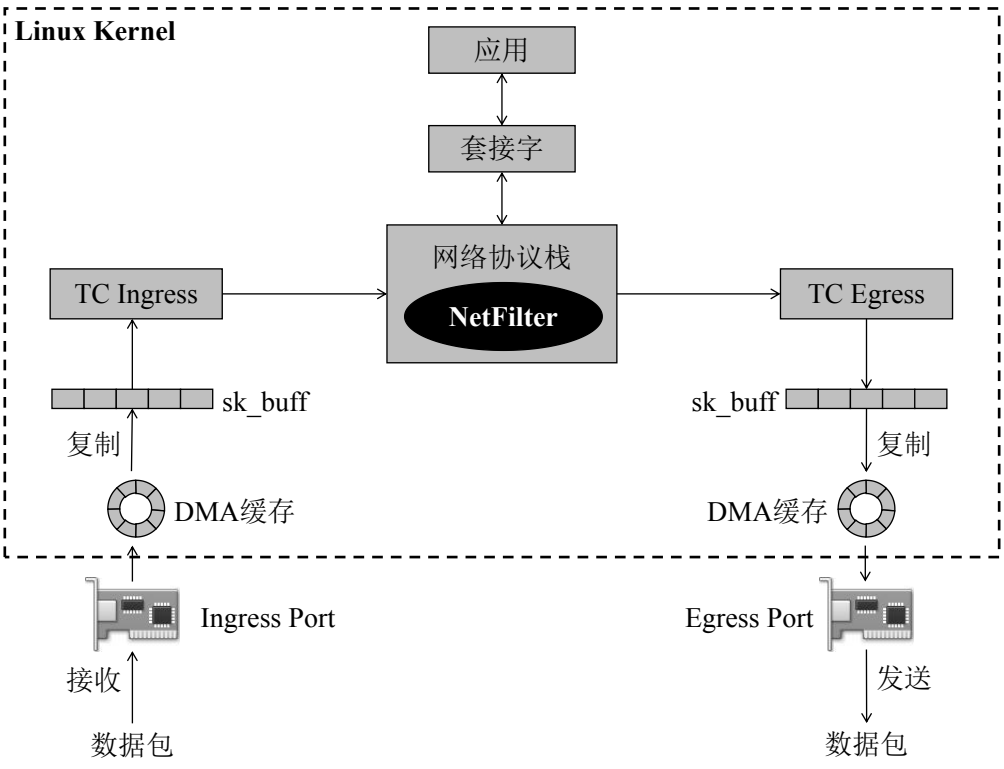

图 2-11 网络数据包路径示意图 [11]

网络数据包流经 Linux 内核的路径如图2-11所示[11]。输入端网卡收到数据包后，对报文进行检查。若数据包发往本系统，则网卡将数据包存到 DMA 缓存中。系统将 DMA 缓存中的报文复制到 sk_buff 中，方便数据包在不同层协议的处理。进入 Linux 系统的流量首先经过 TC Ingress 的处理，然后进入网络协议栈中；iptables 规则的查询与匹配在网络协议栈中进行。若数据包发送至本地应用，则系统通过套接字将报文向上交付；若系统转发此报文，则在 TC Egress 中对数据包进行处理。处理完成后，Linux 系统将 sk_buff 中的数据包拷贝至 DMA 缓存中，等待输出端网卡的读取和发送。

TC (Traffic Control)[12] 是 Linux 系统的网络流量控制模块，主要包括流量策略、流量整形、流量调度及流量丢弃这四大功能。为高效地应对网络中的流量，Linux TC 实现了以下三个部件：

- 队列规程 (QDISC)：队列规程是将准备发送的数据包入队时所遵循的入队规则。在不实施流量控制的情况下，Linux 系统基于“先进先出”(FIFO) 规则将待发送的数据包加入队列；
- 类别 (CLASS)：类别指的是将队列中待发送的数据包分类处理的标准。若队列规则中包含类别，则原有的队列被分为包含不同类型流量的子队列；Linux 系统根据子队列的优先级顺序执行出队操作，实现“流量调度”的功能；
- 过滤器 (FILTER)：过滤器是将队列中的数据包分类的关键部件，可通过 BPF 程序实现；

此外，新版的 TC 还实现了 ACTION 模块[13]，用户可在网络接口上添加 clsact 队列规程，然后在此队列规程上配置 ACTION，处理或丢弃流经 TC 的网络流量。TC Action 也可通过 BPF 程序实现。

2.3.2 BPF

BPF (Berkeley Packet Filter)[14] 是 TC 模块的可编程流量分类器和处理器。用户将流量分类和处理过程编写成汇编代码或高级语言程序，然后通过编译器生成可执行程序，导入 TC 模块中。BPF 将流量的分类和处理过程可编程化，使内核的网络数据通路变得更加灵活。BPF 分为 cBPF 和 eBPF。

(1) cBPF

cBPF (classic BPF) 实现 BPF 的基础功能，其工作流程如图2-12所示。图中

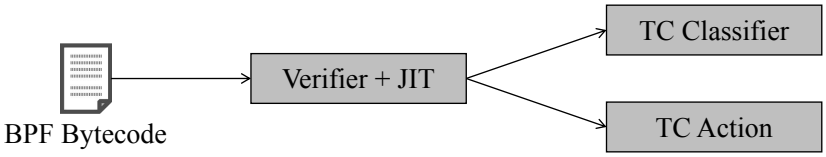

图 2-12 cBPF 的工作流程图

的汇编程序定义了流量的分类过程或数据包的处理过程，该程序由内核提供的 JIT 进行编译，产生可执行程序；与此同时，内核的 Verifier 进行程序的安全验证，以保证程序的运行不会导致系统的崩溃。确保程序能够安全运行后，内核将 BPF 程序导入 TC 模块中。TC 模块主要包括流量分类器和流量处理器。流量分类器通过 BPF 程序将队列中流量进行分类，以进行流量的调度；流量处理器则通过 BPF 程序对数据包的内容进行修改，并根据程序的返回值决定数据包的去向。BPF 作为处理器时，程序的返回值主要有 TC_ACT_OK (0)、TC_ACT_SHOT (2) 和 TC_ACT_UNSPEC (-1)。若返回值为 TC_ACT_OK，则数据包通过 TC 模块；若返回值为 TC_ACT_SHOT，则 TC 模块将数据包丢弃；若返回值为 TC_ACT_UNSPEC，则 TC 模块按照默认的设置处理数据包。

尽管 cBPF 使流量分类和处理过程可编程化，但 cBPF 也具有一定的局限性。首先，流量分类和处理是通过汇编代码实现的，这要求用户必须直接面对 BPF 底层的汇编指令，编程界面不够友好；其次，cBPF 作为流量处理器使用时有较多限制，因此通过 cBPF 实现对数据包的修改和丢弃较为困难。因此，总体来说，cBPF 依然不够灵活。

(2) eBPF

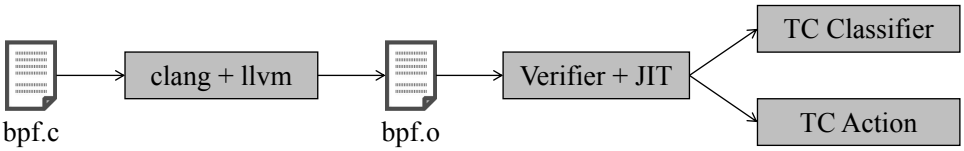

图 2-13 eBPF 的工作流程图

eBPF (extended BPF) 在 cBPF 的基础上做了以下扩展，使 BPF 程序作为流量分类器和处理器时更加灵活：

- 用户可通过高级语言编写 BPF 源程序。用户不需编写汇编代码，只需要通过高级语言程序实现流量的分类和处理。如图2-13所示，用户编写 BPF 的 C 语言源代码后，通过 clang+llvm 编译器编译，生成 BPF 汇编程序；内核

通过 JIT 编译器将汇编程序编译成可执行程序，并将此程序导入 TC 中；在此过程中，内核通过 Verifier 确保程序安全运行。这一流程使用户编程界面更加友好；

- eBPF 引入数据结构 map，使数据的处理过程更加灵活。为实现更加复杂的功能，多个 BPF 程序往往需要共享数据。eBPF 引入了新的数据结构 map，在文件系统中存储 eBPF 程序运行过程中变量的参数。map 是 key-value 映射表，key 和 value 可以是基本数据类型（如 int、char 等），也可以是组合数据类型；用户在创建 map 时，须指定 key 和 value 的数据类型大小。一个 map 可由多个 eBPF 程序访问，因此，多个 eBPF 程序可通过 map 进行数据共享，协作运行，提高数据处理过程的灵活性。

由于 eBPF 更加灵活，本课题中基于 BPF 的容器网络通过 eBPF 来实现。

2.3.3 XDP

XDP (eXpress Data Path)[6] 在进入内核的数据包到达 TC 前处理数据包，使 BPF 程序直接访问存放数据包的 DMA 缓存，在网卡附近修改数据包的字段，并根据程序的返回值决定数据包的去向。XDP 具有以下优点：

- 减少网络数据包不必要的复制开销。若仅通过 iptables 和 BPF 处理数据包，内核须将 DMA 缓存中的数据包复制到 sk_buff 中；若数据包最终被系统丢弃，那么这一复制过程完全没有必要，这导致系统资源的浪费。XDP 使 BPF 程序可直接访问存放报文的 DMA 缓存，即使数据包被丢弃，也不会引入复制开销；因此，XDP 可减少不必要的复制开销；
- 与原有的 TCP/IP 协议栈兼容。与一般的计算机网络不同，数据中心网络的数据流量大，性能要求高，传统的 TCP/IP 协议栈无法满足数据中心网络的需求；尽管如此，改进或重新设计新的协议栈又会加大数据中心应用的开发周期，不利于应用的投产。XDP 保留了上层协议栈的完整性，与原有的 TCP/IP 协议栈兼容；
- 不需要内核旁路技术。部分学者提出，将网卡收到的数据直接交付给应用层，将内核旁路，以减少网络协议栈处理数据的开销，提升网络的性能。然而，内核旁路技术与大部分应用都不兼容，不利于数据中心应用的布署。XDP 在不使用内核旁路技术的情况下，可加速数据包的处理过程（在后面的实验中得到证明）。

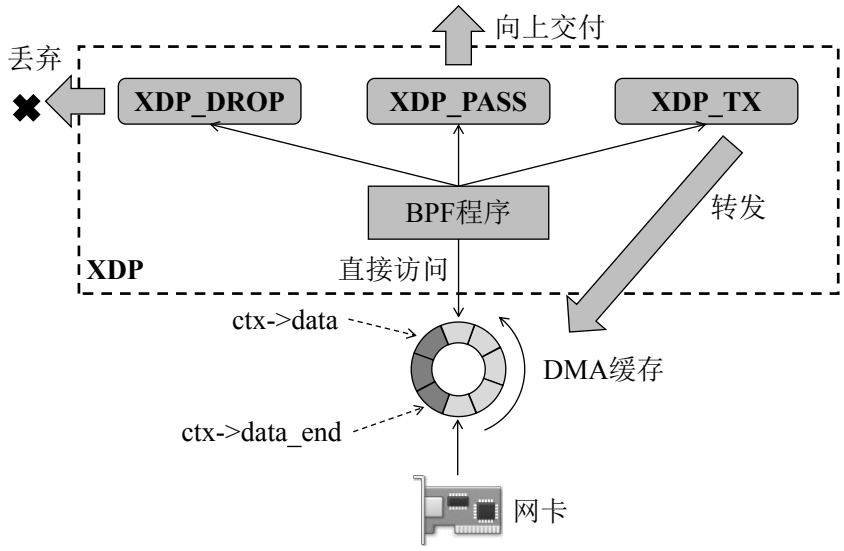

图 2-14 XDP 的工作流程图

XDP 的工作流程如图2-14所示。网络中的数据链路层报文存放在网卡的 DMA 缓存中。BPF 程序通过 ctx 变量来指示报文在缓存中的位置，其中 ctx->data 指针变量指向数据链路层报文的第一个字节，ctx->data_end 指针变量则指向数据链路层报文的最后一个字节。用户可通过 ethhdr、iphdr 等结构体类型访问网络协议的头部信息，并修改网络报文的某些字段。BPF 程序主要通过 3 个返回值决定网络报文的去向，这 3 个返回值分别为 XDP_PASS、XDP_DROP 和 XDP_TX。若返回值为 XDP_PASS，则系统将数据包向上层交付；若返回值为 XDP_DROP，系统将网络数据包丢弃；若返回值为 XDP_TX，则系统将数据包填入 DMA 缓存中，重新发送，此功能主要应用于服务器的负载均衡。

2.4 本章小结

本章主要介绍了课题的背景知识。本章首先介绍 Docker 容器平台的架构、网络空间以及命名空间技术，然后介绍传统容器网络，最后以数据包路径为基础，介绍了 BPF 和 XDP 的工作流程。

第三章 容器网络性能提升的方案设计

基于上一章的背景知识，本章分析基于 iptables 容器网络的性能瓶颈以及基于 BPF 已有方案的不足，并在基于 BPF 已有方案的基础上进行改进。最后，本章将 BPF 与 XDP 结合，进一步设计性能提升方案。

3.1 iptables 的性能分析

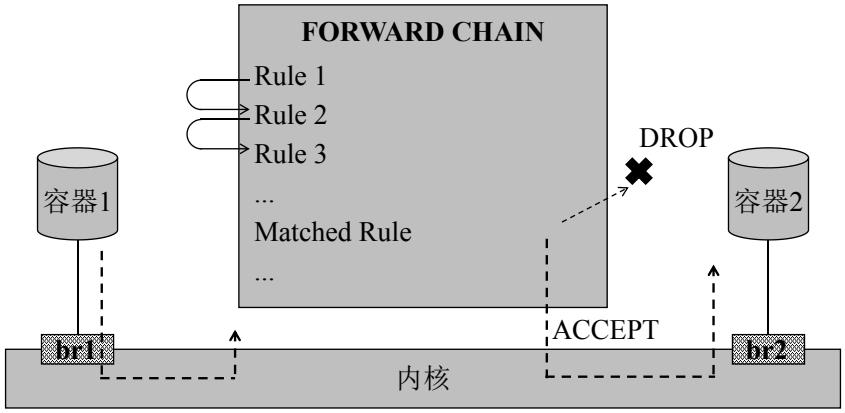

图 3-1 iptables 规则匹配过程示意图

传统的容器网络通过 iptables 处理网络数据包，其规则匹配过程如图3-1所示。当容器 1 产生的报文由网桥接口进入内核后，Linux 系统从第一个表项开始，在转发规则链中查询与数据包信息匹配的规则表项。若第一个表项与数据包信息匹配，则系统按照该表项处理数据包；否则，系统查询第二个表项。若第二个表项与数据包信息匹配，则系统按照第二个表项处理数据包；否则，系统继续查询第三个表项；以此类推。若规则链中没有与数据包信息匹配的规则表项，则 Linux 系统按照转发规则链的默认方式处理数据包。

该容器网络的性能主要取决于规则匹配所需要查找的表项个数。若规则链中所配置的规则项数目较少，则 Linux 系统的查表过程对容器网络的性能几乎没有影响；然而，当规则链中所配置的规则数目较多，Linux 系统的查表过程将占据较多的 CPU 周期，从而影响容器网络的吞吐量。

3.2 基于 BPF 已有方案的分析

BPF 程序作用在 Linux 的 TC 模块中，访问 sk_buff 中的网络数据。前面提到，sk_buff 是表示数据包信息及数据包存放地址的结构体变量，旨在不同协议层次间

对数据包进行处理。若作用在 TC Ingress 处，BPF 程序则可使数据包在进入网络层前就能得到处理。用户首先编写 BPF 源程序，然后通过 clang+llvm 编译器编译源程序，并将产生的汇编代码导入接口的 TC Ingress 处。为保证网络性能不遭受巨大损失，系统规定编译所产生的 BPF 汇编指令个数不得超过 4096 条。

与一条规则项的查询相比，一条汇编指令的执行效率更高。同时，无论 BPF 程序如何编写，编译产生的汇编指令个数不超过 4096 条。因此，相比于 iptables，在 TC 上处理网络流量可以显著提高容器网络的性能。

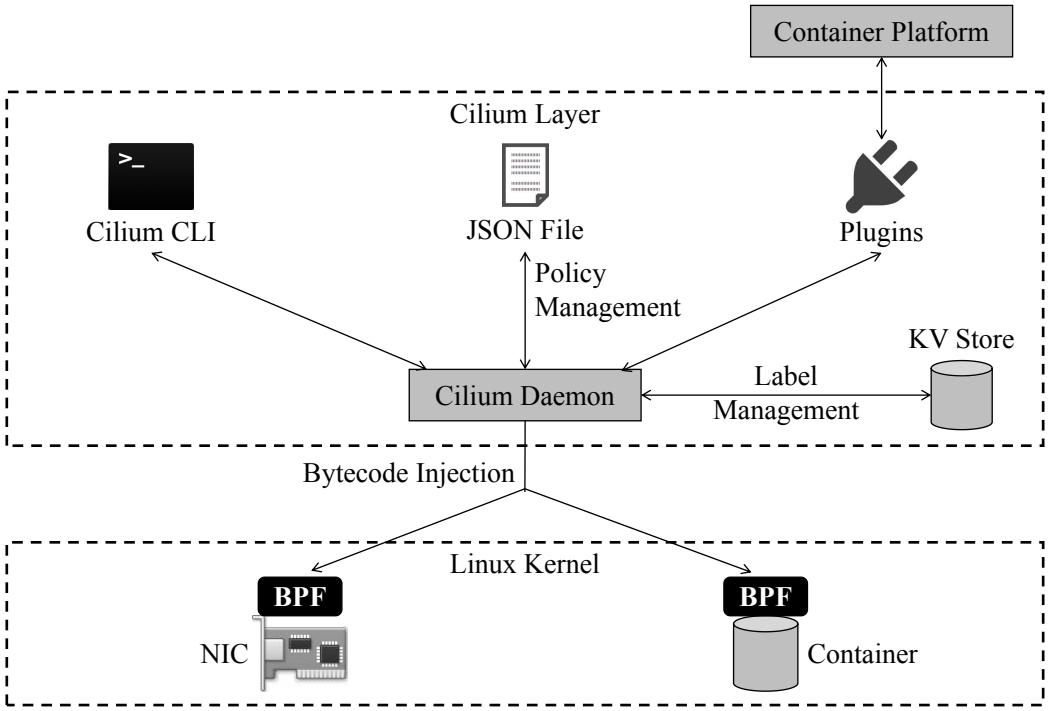

图 3-2 cilium 开源软件的工作流程图 [15]

目前，已经有学者基于 BPF 设计了容器网络性能提升方案，并将此方案与基于 iptables 的容器网络进行性能比较 [5]。该方案基于 cilium 开源软件来设计。cilium 开源软件 [4] 以 BPF 为基础，对流经网卡和容器的网络流量进行高效、灵活地处理，为容器网络提供安全和负载均衡机制；与此同时，为提高容器网络的可扩展性，cilium 开源软件提供了标签机制，为功能相近的容器分配同一个标签，以降低规则配置的复杂度。cilium 软件的工作流程如图3-2所示 [15]，该软件主要包含以下组件：

- Cilium 后台进程 (Cilium Daemon)：Cilium 后台进程是整个开源软件的核心，与 KV Store 结合，进行标签管理。与此同时，后台进程对存放规则条目的 JSON 文件进行解析，并生成 BPF 字节代码，注入网卡或容器中；
- Cilium 终端 (Cilium CLI)：Cilium 终端为用户提供与 Cilium 后台进程交互的命令行界面。用户输入的命令通过该终端发送至后台进程，并由后台进

程执行；

- KV(Key-Value) Store：KV Store 是容器标签的管理程序。每创建一个容器，其标签均记录在 KV Store 中。KV Store 既可以运行在宿主机上，也可以运行在容器中；
- 插件 (Plugins)：为了与现有容器平台协作运行，cilium 开源软件定制了不同的插件，包括自定制的容器网络驱动、IP 地址管理程序等。

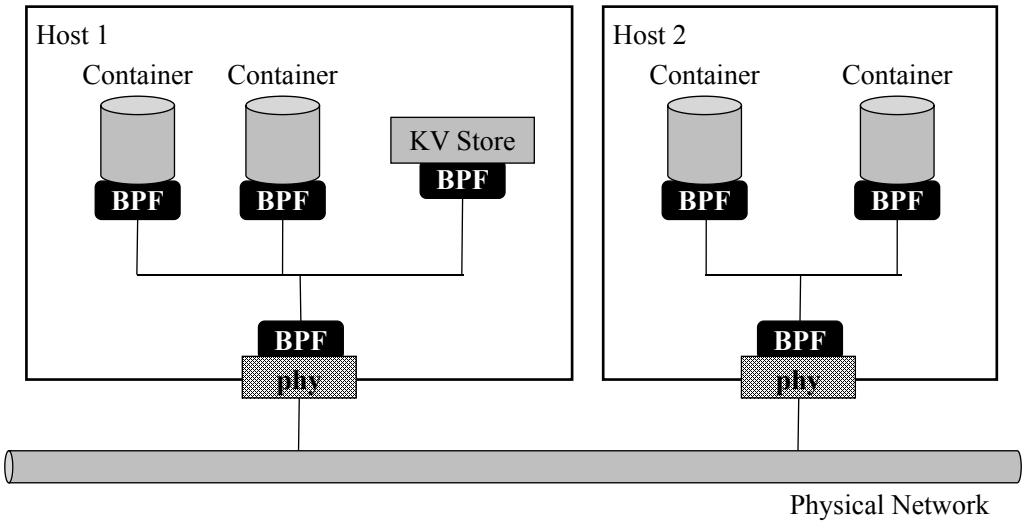

图 3-3 已有方案示意图[5]

基于此开源平台，N. de Bruijn 设计了容器网络的提升方案[5]，其拓扑如图3-3所示[5]。两个主机通过物理接口彼此相连。主机 1 和主机 2 上各运行 Cilium 后台进程，该后台进程对存放规则条目的 JSON 文件进行解析、编译，并将生成的 BPF 字节代码注入物理网卡和容器中。与此同时，为提高可扩展性，主机 1 上运行 KV Store 管理程序，管理整个网络的容器标签。

已有方案尽管优化了容器网络的性能，提高了容器网络布署的可扩展性，但仍然存在以下不足：

- cilium 定制的网络驱动会引入额外的开销。cilium 软件通过自定制的网络驱动为容器分配 IP 地址，并与 KV Store 相结合，共同支持标签的管理。然而，在规则的查询和匹配过程中，cilium 软件需对标签和容器的 IP 地址进行转换，这在容器的通信过程中引入额外的开销；
- 创建容器时有明显卡顿。创建容器网络时，cilium 软件需通过 KV Store 来记录容器的 IP 地址与对应的标签。这一过程需要 cilium 软件对 KV Store 的配置，引入额外的开销，因而容器的创建出现较明显的卡顿；

- 数据包处理规则配置的效率不够高。用户配置数据包处理规则时，须将规则条目写在 JSON 格式的文件中，由 cilium 开源软件去解析。然而，数据包的解析在配置过程中也会引入一定的开销，使得数据包处理规则配置的效率受到一定的影响。

这些在容器创建、配置和通信过程中所引入的开销，都会占用一定的 CPU 处理周期，影响 CPU 对网络数据包的处理能力，从而降低容器网络的吞吐量以及部署、配置效率。因此，以上方案需要进一步改进。

3.3 基于 BPF 的方案改进

针对已有方案的不足，本节进行了实验方案的改进。本节首先列出方案的总体思路，然后阐述单主机容器网络、多主机容器网络的创建，最后阐述 BPF 的软件设计。

3.3.1 总体思路

针对 cilium 开源软件所引入的各方面开销，进一步提高容器网络的吞吐量及布署效率，本课题从以下思路来设计容器网络性能的提升方案。

首先，通过容器平台原生的网络驱动来创建容器网络。Docker 容器平台为方便容器网络的创建、管理和维护，提供了 bridge、overlay 等容器网络驱动，这些驱动能够高效地实现 IP 地址分配、数据包处理等功能，与容器平台本身兼容；而 cilium 开源软件为不同容器平台定制自己的驱动，无论在 IP 地址分配和数据包处理上，都引入一定的开销，降低 CPU 对数据包的处理能力，从而影响容器网络的吞吐量。因此，使用 Docker 平台原生的网络驱动可以有效提高容器网络的性能。

其次，使用 Linux 内核提供的工具直接对 BPF 和 XDP 进行配置。Linux 内核通过 iproute2 工具将 BPF 程序加载到 TC 模块和 DMA 缓存中，这一工具本身就能够高效地对 BPF 和 XDP 进行配置。而 cilium 开源软件在配置规则的时候，需要读取和识别存放处理规则的 JSON 文件，再通过 iproute2 工具进行配置。尽管 JSON 文件提高规则的可读性，但 JSON 文件的翻译过程降低了 BPF 和 XDP 的配置效率。因此，使用内核自带的工具能够高效地配置数据包处理规则。

此外，每一项处理规则均采用 IP 地址标识网络中的容器。cilium 开源软件通过标签来标识一组功能相近的容器，由 KV Store 来管理这些标签，从而减少所需规则的个数。然而，在容器创建过程中，这种标签机制引入额外的开销，使得容器的创建出现明显的卡顿。之所以在规则中使用 IP 地址来标识容器，一方面是为

了减少容器创建的卡顿，另一方面则为了简化规则的匹配过程。为减少所需规则的个数，提高可扩展性，可以将功能相近的容器划分到同一个 IP 地址段中，实现与 cilium 开源软件一样的效果。

基于以上思路，本文对已有方案作进一步改进，进一步提高容器网络的吞吐量。由于数据中心的容器网络非常庞大，不便于赘述，本文以单主机容器网络和简单的多主机容器网络为例，阐述方案的具体设计。

3.3.2 容器网络的创建

(1) 单主机容器网络

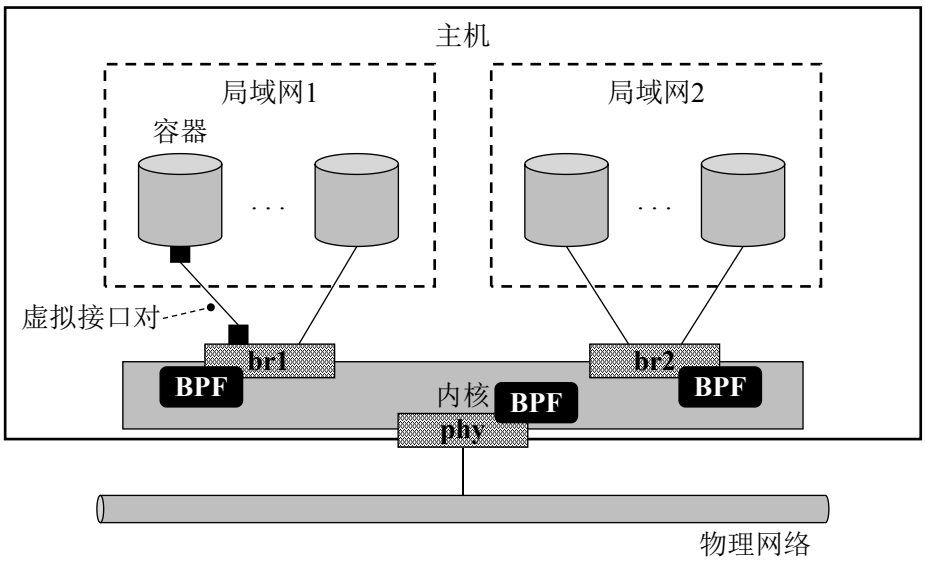

图 3-4 单主机容器网络的性能提升方案

单主机容器网络的性能提升方案如图3-4所示。主机上有三个网络接口，phy 接口为主机的物理接口，与物理网络连接；br1 与 br2 均为由 Docker 容器平台通过 bridge 驱动创建的虚拟网桥接口。内核之上为 Docker 平台创建的容器，分别位于不同的局域网中；容器通过 Docker 平台新建的虚拟接口对与网桥连接。在此环境中，BPF 程序在虚拟接口和物理接口上处理进入内核的数据包，其中，物理接口的 BPF 程序处理由外部网网络进入容器网络的数据包，虚拟接口的 BPF 程序则处理容器之间交互以及从容器网络到外部网络的数据包；这些 BPF 程序既能保证容器网络的安全，也能实现 QoS、负载均衡等功能。为方便规则的配置，网络管理人员将功能相近的容器规划到同一个局域网中；若需要往局域网中添加新的容器，管理人员无需重新配置数据包处理规则，实现与 cilium 标签机制一样的效果。

(2) 多主机容器网络

在一般情况下，应用比较复杂，所需容器较多，若将所有容器运行在同一主机上，则容易导致 CPU 过载、网络性能严重下降等问题。因此，本文针对多主机容器网络进行方案设计。由于数据中心网络往往是多租户的，不同租户在主机之上布署虚拟化网络，且彼此之间相互独立。VXLAN 协议可以满足数据中心网络的多租户需求[7]，因此，本文采用 VXLAN 协议搭建多主机容器网络。

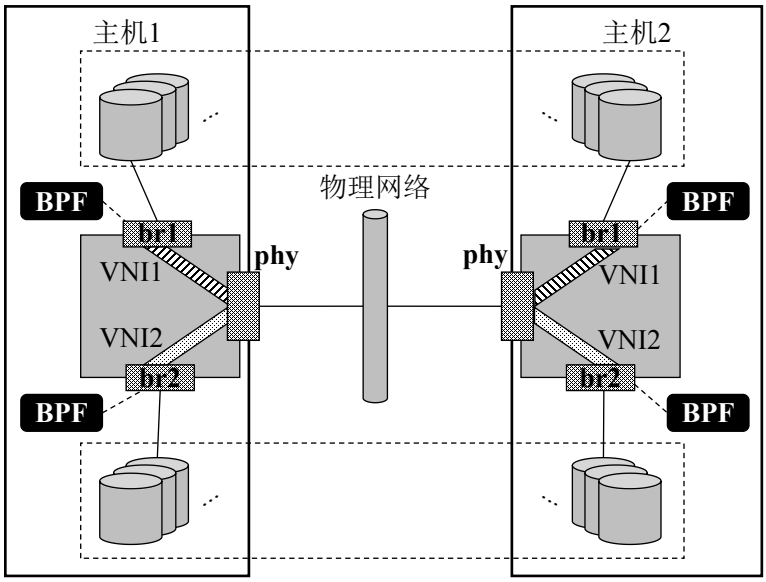

图 3-5 多主机容器网络的性能提升方案

多主机容器网络的性能提升方案如图3-5所示。多主机容器网络通过 VXLAN 协议来构建，图中主机 1 与主机 2 均与物理网络相连，主机各有两个虚拟网桥接口，每个接口都桥接若干个容器；网桥接口和容器均通过 Docker 容器平台来创建。主机上通过 iproute2 工具各创建两个 VXLAN 类型的虚拟接口，每台主机 VXLAN 接口的虚拟网络标识分别为 VNI1 和 VNI2，并分别桥接到不同的网桥接口中，以构建逻辑子网。若网桥所连 VXLAN 接口的虚拟网络标识相同，则网桥上连接的所有容器均位于同一个逻辑子网中（位于同一个虚线框中）。为对逻辑子网中的容器进行数据包处理，网桥接口上均加载 BPF 程序。之所以不通过原生的 overlay 驱动直接创建多主机容器网络，是因为 overlay 驱动在配置过程中，将网桥接口加入到新的命名空间中，不方便对网桥接口的直接配置。尽管 VXLAN 协议在容器通信时会增加一定的开销，但 VXLAN 被普遍认为最适合布署大规模云计算网络环境的协议[2]，并且满足数据中心的多租户需求。因此，该方案更贴近实际的应用。

3.3.3 BPF 程序设计

为保证外部网络能够访问主机中的容器，需要在主机上配置静态 NAT 规则；与此同时，为防止黑客入侵，还需要在主机上配置过滤规则，拒绝非法的网络流量。本小节以过滤规则和静态 NAT 规则为例，阐述 BPF 程序的设计。

(1) 过滤规则的设计

过滤规则可以基于 IP 地址、端口号等字段来配置，本文以源 IP 地址和目的 IP 地址为例，阐述过滤规则的程序设计。

若通过 iptables 命令来配置过滤规则，则命令为“iptables -A FORWARD -s SRCIP -d DSTIP -j DROP”，将源 IP 地址为 SRCIP，且目的 IP 地址为 DSTIP 的网络报文丢弃。

与之对应的 BPF 程序流程图如图3-6所示。该程序首先通过 iproute2 提供的系统调用函数从 sk_buff 中读取源 IP 地址和目的 IP 地址，并判断源 IP 地址是否为 SRCIP，目的 IP 地址是否为 DSTIP。如果以上条件均满足，则系统将此报文丢弃；否则，系统将报文向网络层交付。

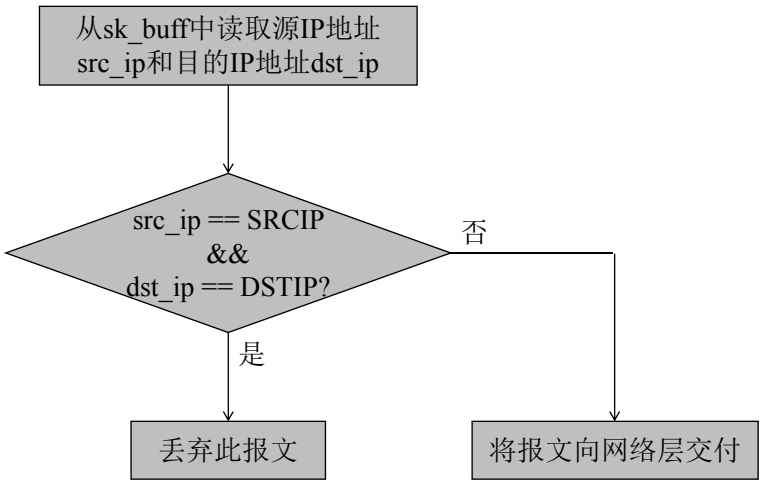

图 3-6 基于 BPF 的过滤规则程序流程图

(2) 静态 NAT 规则的设计

静态 NAT 包括源地址的静态映射和目的地址的静态映射。本文以目的地址的静态映射为例，阐述静态 NAT 规则的程序设计。

若通过 iptables 命令配置目的地址的静态 NAT，则命令为“iptables -t nat -A PREROUTING -i DEV -d IPADDR1 -p tcp –dport PORT1 -j DNAT –to-destination IPADDR2:PORT2”。通过以上配置后，系统对从 DEV 接口进入的网络报文作预路

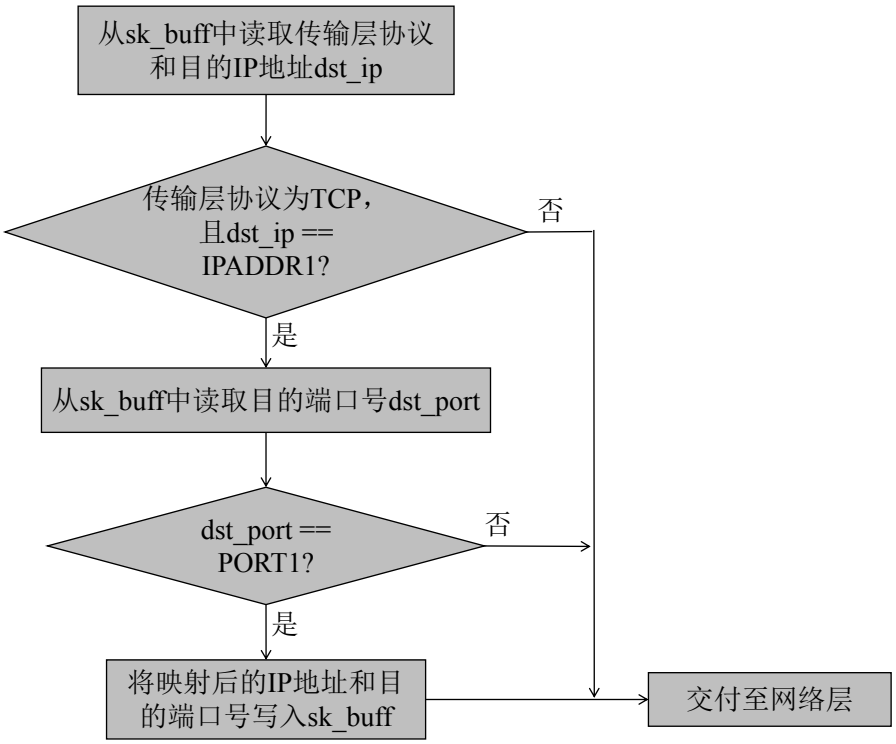

图 3-7 静态 NAT 规则的程序流程图

由检查，并将 TCP 报文中的目的地址 IPADDR1:PORT1 映射为 IPADDR2:PORT2。

BPF 程序则需要从数据包中读取 IP 地址和端口号，然后将新的 IP 地址和端口号写入数据包，程序的流程如图3-7所示。当数据包经过 TC 模块时，BPF 程序从 sk_buff 中读取目的 IP 地址及传输层协议。若传输层协议为 TCP，且目的 IP 地址为 IPADDR1，则从 sk_buff 中读取 TCP 头部的端口号，若端口号为 PORT1，则 BPF 程序将目的 IP 地址和目的端口号覆写，实现地址的映射。最后，程序将报文交付至网络层。

3.4 基于 XDP 的方案设计

XDP 是最近出现的加速网络性能的技术[6]。XDP 使 BPF 程序直接访问 DMA 缓存，在数据包到达 TC 模块前就得到处理。由于目前还没有相关文献通过 XDP 提升容器网络性能，本文对 XDP 的性能提升作用进行研究。

在前一节的方案中，BPF 程序访问的是 sk_buff。sk_buff 是内核存放数据包的结构体变量，通过链表将其它数据包链接在一起，形成网络流量。与此同时，除数据包信息外，一个 sk_buff 还包括大量的控制信息。正因为 sk_buff 的处理过程比较繁杂，iproute2 工具提供了 helper 函数，以方便对数据包的处理；然而，繁杂的处理过程以及频繁的函数调用所引入的开销在一定程度上影响 CPU 对数据包的处

理能力。在 XDP 中，BPF 程序通过指针运算直接访问 DMA 缓存中的数据，不需要专门的系统调用函数，处理效率更高。因此，与 TC 中的 BPF 程序相比，XDP 中的 BPF 程序在理论上可以加速容器网络的性能。

与前一节的方案相比，基于 BPF+XDP 方案的主要区别在于数据包处理方式的不同，因而程序的设计也有所不同。本节从过滤规则和静态 NAT 规则两方面阐述 XDP 方案的程序设计。

(1) 过滤规则的设计

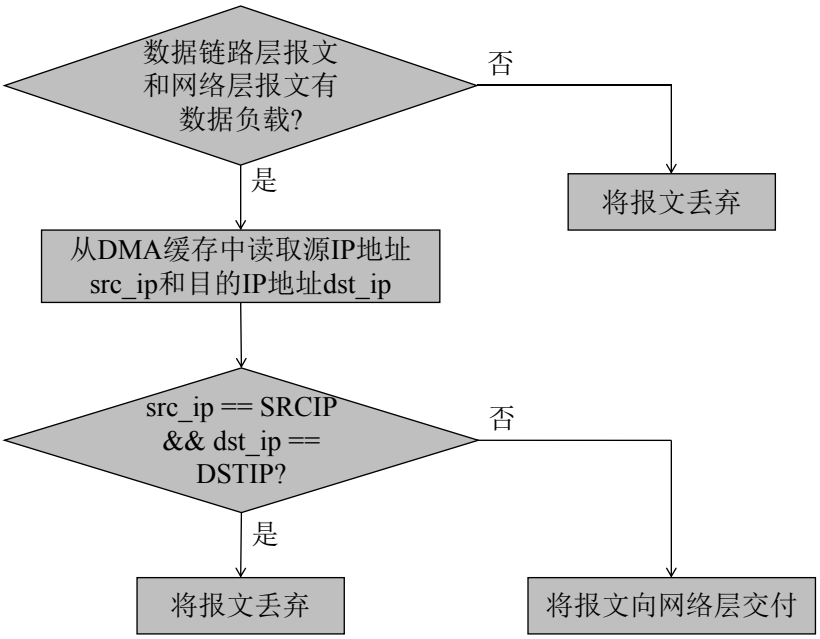

图 3-8 基于 XDP 的过滤规则程序流程图

本节同样以源 IP 地址和目的 IP 地址为例，阐述 BPF+XDP 的程序设计。基于 BPF+XDP 的过滤规则程序流程如图3-8所示。在程序的开始，需要检查数据链路层报文和网络层报文是否有数据负载，若数据包没有负载，则直接将报文丢弃。若没有此检查过程，则程序最终无法通过内核的安全检查。接着，BPF 程序通过指针运算，从 DMA 缓存中读取源 IP 地址和目的 IP 地址字段。若源 IP 地址为 SRCIP，且目的 IP 地址为 DSTIP，则将报文丢弃。

(2) 静态 NAT 规则的设计

本节同样以目的地址的静态映射为例，阐述 BPF+XDP 的程序设计。基于 BPF+XDP 的静态 NAT 规则程序流程如图3-9所示。该程序首先对数据负载进行检查。确保数据包正确后，程序通过指针运算，从 DMA 缓存中读取目的 IP 地址和

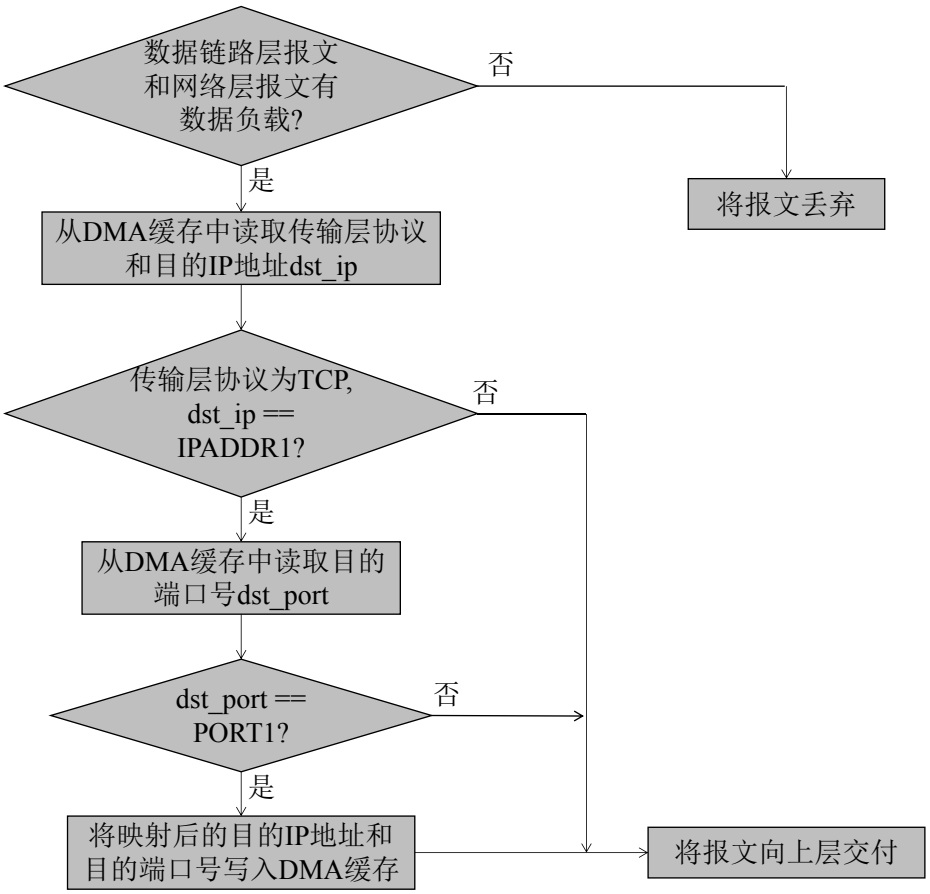

图 3-9 基于 XDP 的静态 NAT 规则程序流程图

传输层协议。如果传输层协议为 TCP，且目的 IP 地址为 IPADDR1，则继续读取目的端口号，如果目的端口号为 PORT1，则将映射后的目的地址和端口号覆写到 DMA 缓存。

3.5 本章小结

首先，本章分析 iptables 的性能瓶颈，iptables 的查表效率较低，在规则项较多时会导致性能下降。其次，本章分析基于 BPF 已有方案的不足，该方案所使用的驱动引入一定的开销，降低网络吞吐量。最后，本章结合 XDP，进一步阐述性能提升方案。

第四章 方案实现与实验测试

根据上一章的方案设计思路，本章阐述方案的实现过程，并通过实验，测试基于 BPF 已有方案和本文方案的容器网络吞吐量以及基于 iptables、BPF 和 BPF+XDP 容器网络的吞吐量和网络延时，以验证基于 BPF 本文方案的可行性以及 BPF+XDP 对性能的进一步提升作用。

4.1 方案的实现过程

本节将从软硬件环境的配置、容器网络的搭建以及 BPF 程序的编写这三方面，分小节阐述方案的实现过程。

4.1.1 软硬件环境的配置

本方案的硬件环境包含两台主机，它们均安装了 CentOS 7 操作系统，均配置有 10G 网卡，并通过 10G 网线直接相连。其中，主机 1 的内存大小为 8GB；CPU 为 Intel 公司生产的 4 核 CPU，主频为 3.40GHz，采用 x86-64 架构。主机 2 的内存大小为 64GB；CPU 为 Intel 公司生产的 12 核 CPU，主频为 2.40GHz，采用 x86-64 架构。

在此硬件环境下，通过以下步骤来搭建软件环境：

(1) 编译内核。CentOS 7 的初始内核版本为 3.10.0，而 XDP 要求内核版本至少为 4.8.0；此外，Linux-3.10.0 不支持 VXLAN 协议，因此无法在此内核上创建多主机容器网络。为更好地兼容 XDP，本文选择 Linux-4.12.0 作为本方案的操作系统内核。与此同时，为使内核加载 BPF 相关的模块，本文按照图4-1所示的方式，在.config 文件中配置内核选项；

(2) 安装 Docker 软件。本文通过 CentOS 的镜像源来安装 Docker，镜像源提供的 Docker 软件能够满足本方案的应用需求；

(3) 编译安装 iproute2。一般情况下，系统自带的 iproute2 工具版本较低，无法兼容 BPF 和 XDP 的配置。本课题要求的 iproute2 版本不低于 4.8.0；为更好地兼容 BPF 和 XDP，本文选择的版本号为 4.15.0；

(4) 编译安装 cmake。本课题的 clang+llvm 编译环境需通过 cmake 来建立，所要求的版本号至少为 3.4.3。本文所选的版本号为 3.10.3；

(5) 编译安装 clang+llvm 编译器。本课题的 clang+llvm 编译器不得低于 3.7.1，本文选择 4.0.0 版本作为本方案 BPF 程序的编译器。在编译之前，需通过

命令“cmake .. -DLLVM_TARGETS_TO_BUILD=BPF;X86”来搭建编译环境，使编译生成的编译器支持 BPF 程序的编译。

```
CONFIG_CGROUP_BPF=y
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
CONFIG_NET_SCH_INGRESS=m
CONFIG_NET_CLS_BPF=m
CONFIG_NET_CLS_ACT=y
CONFIG_BPF_JIT=y
CONFIG_LWTUNNEL_BPF=y
CONFIG_HAVE_EBPF_JIT=y
CONFIG_BPF_EVENTS=y
CONFIG_TEST_BPF=m
```

图 4-1 内核选项的配置

4.1.2 容器网络的搭建

1. 单主机容器网络的搭建

单主机容器网络的创建过程如图4-2所示。该容器网络的创建主要分以下两步：

(1) 新建网桥接口。通过命令“docker network create --driver bridge --subnet 10.0.1.0/24 --opt com.docker.net.bridge.name=br1 br1”，Docker 平台创建名为 br1 的网桥接口，并将 10.0.1.0/24 的地址段分配给网桥接口及与之相连的所有容器。br2 接口的创建与上述命令类似；

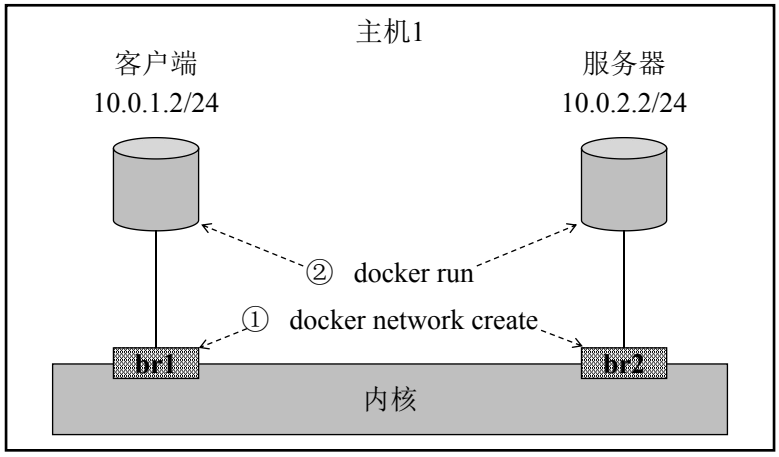

图 4-2 单主机容器网络的创建过程

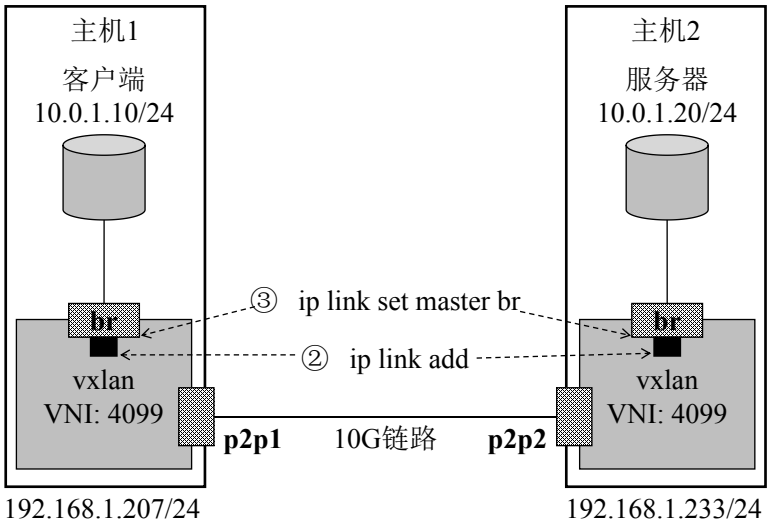

图 4-3 多主机容器网络的创建过程

(2) 创建容器。通过命令“docker run --network br1 [IMAGE]”，以 IMAGE 镜像为模版，创建容器，并初始化容器运行环境。然后，Docker 平台通过虚拟接口对，将新建的容器与网桥接口连接。

2. 多主机容器网络的搭建

多主机容器网络的创建过程如图4-3所示。多主机容器网络的创建主要分为以下四步：

(1) 新建网桥接口。该步骤的命令与单主机容器网络的创建类似；

(2) 新建 VXLAN 接口。在主机 1 上运行命令“ip link add name vxlan type vxlan id 4099 dstport 4789 dev p2p1 remote 192.168.1.233”，新建 VXLAN 接口，名为 vxlan，并将其 VNI 设为 4099，UDP 端口设为 4789，VXLAN 隧道本端为 p2p1 接口，隧道对端是 IP 地址为 192.168.1.233 的网络设备。主机 2 的配置与此类似；

(3) 将 VXLAN 接口桥接至网桥接口。在主机 1 上运行命令“ip link set vxlan master br1”，将新建的 VXLAN 接口桥接至 br1。主机 2 的配置与此类似；

(4) 创建容器。创建容器的过程与单主机容器网络的类似。

4.1.3 数据包处理规则的配置

为保证外部网络能够访问主机内部的容器，BPF 程序上须编写静态 NAT 规则。由于在实际应用中，主机上往往运行很多容器，主机端口号与容器的映射关系需要在程序中维护。与此同时，数据中心网络外部可能有黑客，不断入侵网络内部，

窃取用户数据。为避免黑客的入侵，还需要配置数据包过滤规则，将非法的网络流量丢弃。在此，本文假定网络外部的黑客已知，可针对源 IP 地址对非法入侵的报文进行过滤。

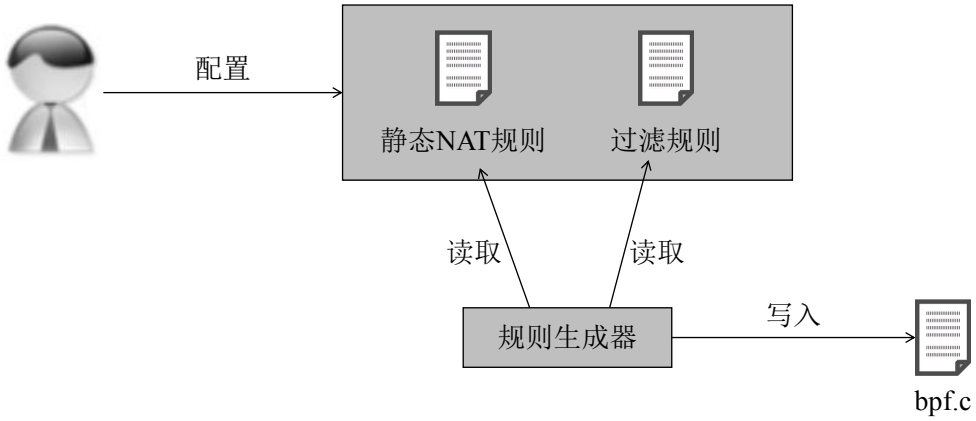

图 4-4 规则配置流程示意图

从理论上，规则表可通过 eBPF 引入的 map 来维护；然而 map 的配置较为繁杂，本文采用如图4-4所示的方式，通过 BPF 实现数据包的处理规则。静态 NAT 规则和过滤规则分别保存在两个文件中。规则生成器是通过 C 语言编写的程序，通过文件的读写操作生成 BPF 程序；规则生成器分别读取保存静态 NAT 规则和过滤规则的文件，并将文件中的每一个规则项翻译成 BPF 程序。静态 NAT 规则和过滤规则的程序编写已经在第三章详细说明。生成的 BPF 程序通过 clang+llvm 编译器编译源代码，并导入 TC 或加载到 DMA 缓存中。

4.2 测试工具的选择

在本课题中，本文需要对容器网络进行性能测试，因此性能测试工具的选择非常重要。本课题围绕网络吞吐量和网络延时这两个指标来衡量容器网络的性能，因此，本文比较了网络吞吐量和网络延时的典型测试工具。

(1) 网络延时测试工具的选择

网络延时的测试工具主要有以下三个：

- ping[16,17]：ping 工具利用 ICMP 协议来测试网络时延。发送方根据发送 ICMP 请求的时间戳和收到 ICMP 回应的时间戳来计算双方的双向链路时延，并将测试结果显示在终端。在 Linux 系统中，ping 工具所能测试的最高精度为 0.001ms；
- netperf[5,17,18]：netperf 是基于客户端–服务器模型的测试工具。客户端与服

务器建立 TCP 控制连接，传递与测试相关的参数；根据这些参数，客户端与服务器在 TCP、UDP 或 SCTP 协议下进行性能测试。测试完成后，客户端通过之前的控制连接与服务器传递测试结果，并输出终端；最后，客户端与服务器拆除控制连接。netperf 工具包含批量数据传输测试和请求–响应测试这两种模式。在请求–响应测试中，客户端不断向服务器发送请求，服务器对收到的请求一一进行回应。根据一定测试时间内请求和响应的次数，netperf 客户端计算网络的传输速率，测试者需通过计算间接求出网络延时；

- NetPIPE[19]：NetPIPE 是基于客户端–服务器模型的测试工具。客户端依次向服务器发送大小不同的数据包，并根据服务器的回应时间分别计算不同数据包大小条件下的单向链路延时；测试者可根据测试数据作出单项链路延时与数据包大小的关系曲线图，全面分析网络延时的变化情况。此工具的测试精度可达到 0.01μs。

尽管 ping 工具简单常用，但所测试的延时精度较低，在单主机容器网络环境下难以体现性能的差异；netperf 工具则需要根据输出的网络传输速率，通过中间计算间接地求出网络延时，计算的过程容易引入误差。NetPIPE 工具能够直接测试网络延时，精度较高；此外，该工具还可利用测试结果作出单向链路延时与数据包大小的关系曲线图，全面反映网络延时的变化情况。综合以上因素，本文选用 NetPIPE 来测试容器的网络延时。

(2) 网络吞吐量测试工具的选择

网络吞吐量的测试工具主要有以下两个：

- iperf3[5,16]：iperf3 是基于客户端–服务器模型的网络吞吐量测试工具。客户端程序运行后，通过 TCP 或 UDP 协议与服务器会话。在此会话中，客户端向服务器发送测试参数，并根据参数进行吞吐量测试；测试完成后，客户端通过该会话与服务器传输吞吐量测试结果，并输出至终端；
- netperf：前面提到了 netperf 工具的测试流程和测试模式。netperf 的网络吞吐量通过批量数据传输模式来测试。客户端不断向服务器发送批量数据，服务器根据收到数据的大小和测试持续的时间，来计算网络的吞吐量，并将测试过程中网络的吞吐量输出至终端。

iperf3 工具测试参数及测试结果的传递均与测试报文在同一个会话中，因此利用此工具测试会引入一定的误差；而 netperf 工具通过 TCP 控制连接传输测试参数

和测试结果，测试报文则与此连接独立，因而与 iperf3 工具相比，netperf 工具的测试误差较小。因此，本文选择通过 netperf 工具来测试容器网络的吞吐量。

4.3 实验验证与结果

本节基于前面所述的软硬件环境，利用前一节所选择的工具，测试基于 BPF 已有方案和本文方案的容器网络吞吐量以及基于 iptables、BPF 和 BPF+XDP 容器网络的吞吐量和网络延时，以验证基于 BPF 本文方案的可行性以及 BPF+XDP 对性能的进一步提升作用。

4.3.1 方案的比较

本小节在单主机容器网络环境下测试基于 BPF 已有方案和本文方案的容器网络吞吐量，以验证基于 BPF 本文方案的可行性。

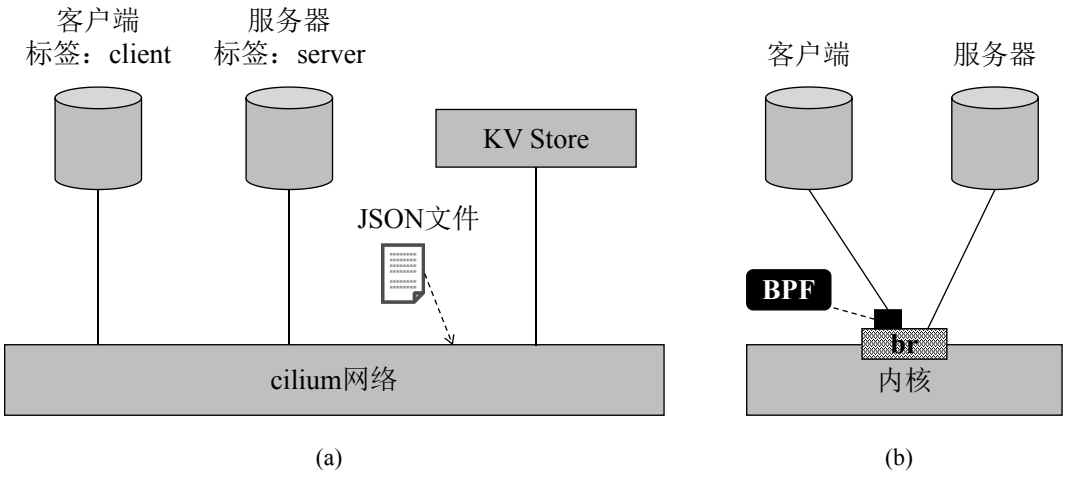

图 4-5 方案对比的实验环境：(a) cilium 容器网络；(b) BPF 容器网络

实验的网络环境如图4-5所示。其中图4-5(a)为基于 BPF 已有方案的示意图，容器网络通过 cilium 开源软件自己定制的网络驱动来构建，网络中有三个容器，一个容器为客户端，一个容器为服务器，最后一个为 KV Store。该网络的数据包处理规则由 JSON 文件来配置，并通过标签来标识客户端与服务器。图4-5(b)为基于 BPF 本文方案的示意图，br 为 Docker 平台通过原生的 bridge 驱动创建的虚拟桥接接口，将运行在内核上的两个容器桥接在一起。其中一个容器为客户端，另一个容器为服务器，均采用 IP 地址标识。本文通过 BPF 程序编写数据包处理规则，并将编译产生的汇编程序导入客户端接口的 TC Ingress 中，并依次在以上网络环境中分别添加 0、100、200、300、400、500 条过滤规则，每次规则的配置均满足匹配项在规则列表的最后一项，保证每转发一个数据包，内核都经历较长时间的查

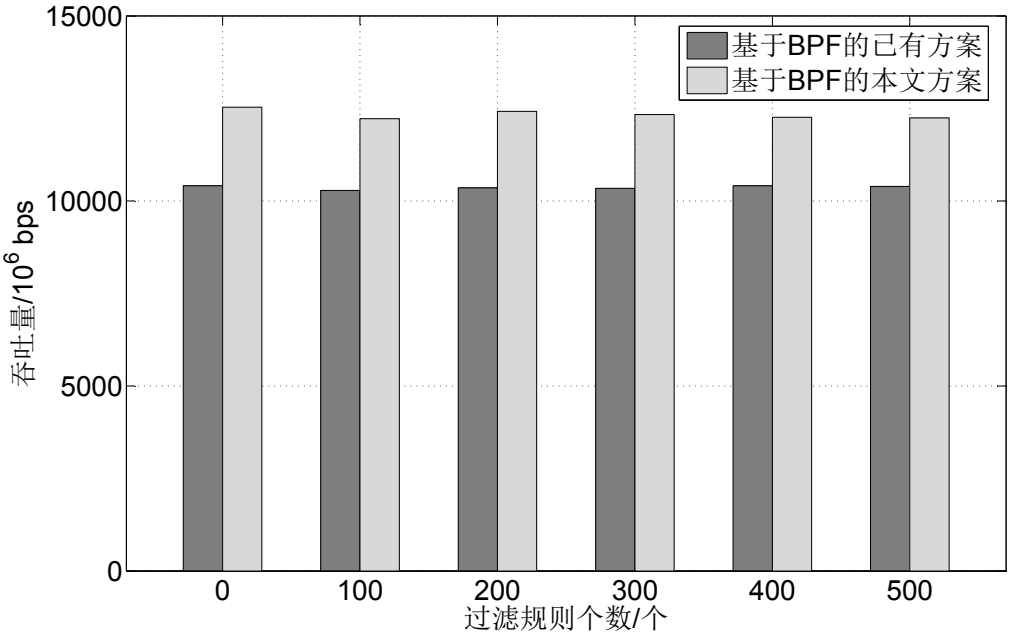

图 4-6 基于 BPF 已有方案与本文方案的对比实验结果

表过程。在不同规则数的条件下，通过 netperf 工具，测试两个容器网络的吞吐量；测试过程中，客户端不断向服务器发送 16384 字节的数据包，每次测试的时长为 1 分钟。

以上对比实验的结果如图4-6所示。当规则数在 0～500 之间变化时，cilium 容器网络的吞吐量稳定在 10～11Gbps；本文所设计的容器网络吞吐量则稳定在 12～13Gbps，在性能上优于已有方案的设计。此外，当规则数为 0 时，本文设计的容器网络吞吐量仍然比 cilium 容器网络吞吐量高 2Gbps，这是因为 cilium 容器网络是通过自定制的网络驱动建立的，在容器通信时引入额外的开销，而本文实现的容器网络是通过 Docker 平台原有的驱动建立的，与平台的兼容性较好；因此，即使不配置规则，本文的容器网络在性能上依然优于 cilium 容器网络。

由于通过 cilium 开源软件搭建多主机容器网络的过程比较繁杂，在此不通过实验比较两种方案在多主机环境下的优劣。在多主机容器网络中，决定容器网络性能的因素主要有容器网络驱动、主机 CPU 运算能力以及主机间的链路速率。在单主机环境的对比实验中，通过分析，得出了 Docker 平台的原生容器网络驱动优于 cilium 开源软件自定制的容器网络驱动这一结论，因此，可以推断出，在多主机容器网络中，在链路速率、主机硬件配置等条件一定的情况下，本文设计的容器网络吞吐量高于通过 cilium 开源软件构建的容器网络吞吐量。

综上所述，在网络吞吐量指标上，基于 BPF 的本文方案优于基于 BPF 的已有

方案。

4.3.2 BPF+XDP 性能提升作用的验证

本小节分别在单主机容器网络和多主机容器网络环境下，测试基于 iptables、BPF 和 BPF+XDP 容器网络的吞吐量和网络延时，以验证 BPF+XDP 对性能的进一步提升作用。

(1) 单主机容器网络实验

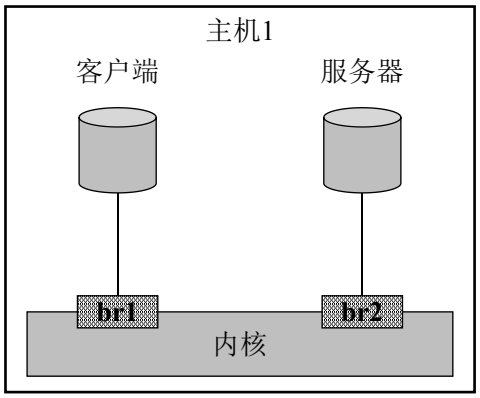

图 4-7 单主机容器网络环境示意图

单主机容器网络的实验环境如图4-7所示。br1 和 br2 为主机的网桥接口，br1 与客户端容器相连，br2 与服务器容器相连。首先针对 iptables 进行性能测试，在静态 NAT 规则的基础上，添加数据包过滤规则。接着，删除之前添加的所有规则，针对 BPF 和 BPF+XDP 分别进行性能测试。本文依次配置静态 NAT 规则和数据包处理规则，并通过自己编写的程序生成器生成 BPF 程序，然后通过 clang+llvm 编译器编译，并将产生的目标文件分别导入 TC 和 DMA 缓存中。为防止编译过程中发生地址聚合，采用随机数生成过滤规则中的源 IP 地址匹配项，保证实验的准确性。本文依次添加 0、100、200、...、900 条过滤规则，通过 netperf 工具测试单主机容器网络的吞吐量；测试过程中，客户端不断向服务器发送 16384 字节的数据包，每次测试的时长为 1 分钟。然后，本文在过滤规则数为 0、200、40、600、800 的条件下，通过 NetPIPE 工具测试单主机容器网络的单向链路时延。

单主机容器网络的实验结果如图4-8和图4-9所示。图4-8反映了不同容器网络、不同策略个数的条件下容器网络吞吐量的变化情况。当容器网络中没有添加任何策略时，基于 iptables、BPF 和 BPF+XDP 的容器网络吞吐量非常接近，约为 16 Gbps。然而，随着策略个数的增加，三种容器网络的变化趋势明显不同。iptables 容器网络的吞吐量下降非常明显，当规则数为 500 时，iptables 容器网络的吞吐量约为 10Gbps；当规则数为 900 时，吞吐量仅为 8 Gbps，与没有规则数的条件相比，

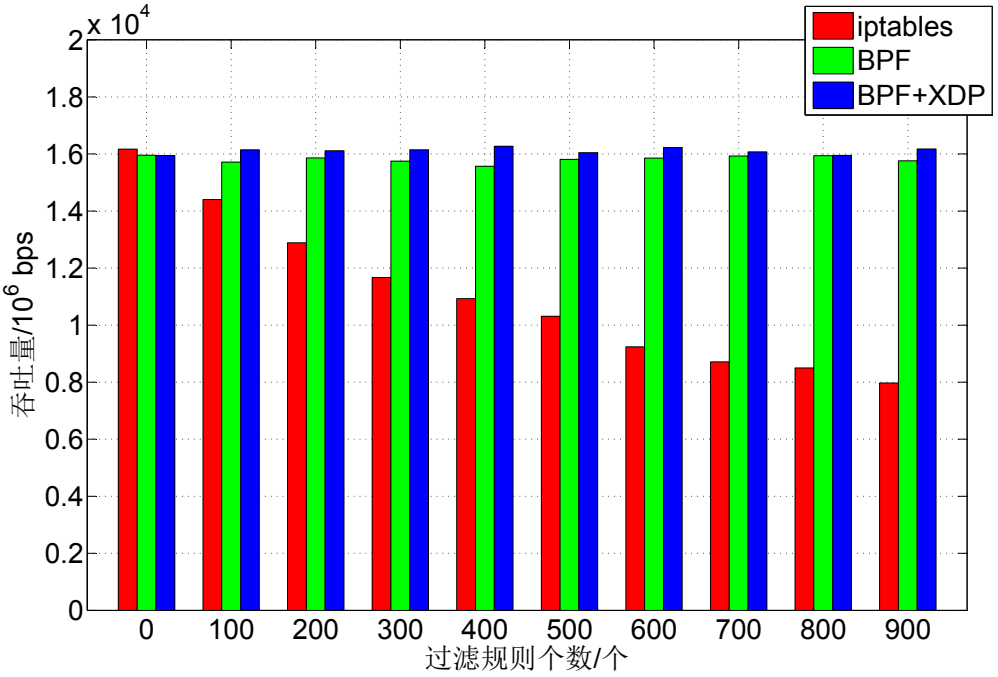

图 4-8 单主机容器网络的吞吐量柱状图

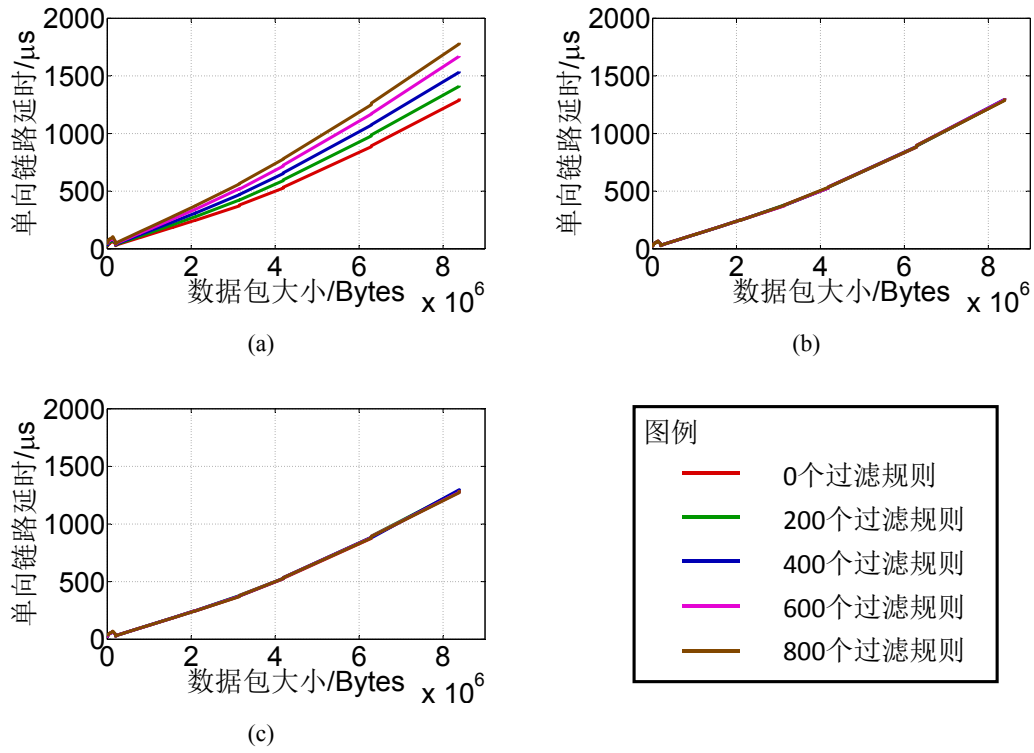

图 4-9 单主机容器网络的延时–数据包大小关系曲线图：(a) 基于 iptables 容器网络的曲线；(b) 基于 BPF 容器网络的曲线；(c) 基于 BPF+XDP 容器网络的曲线

性能损失超过一半。然而，当规则数在 0～900 之间变化时，基于 BPF 和 BPF+XDP 容器网络的吞吐量均维持稳定的水平，BPF 容器网络的吞吐量稳定在 15.8～16.0 Gbps 范围内，BPF+XDP 容器网络的吞吐量稳定在 16.0～16.1 Gbps 之间。

图4-9反映的是不同容器网络、不同策略个数的条件下单向链路延时随数据包大小的变化趋势。在 iptables 容器网络中，随着规则数的增加，单向链路时延随数据包大小的变化速率明显变陡，而在基于 BPF 和 BPF+XDP 的容器网络中，不同过滤规则个数的单向链路延时–数据包大小关系曲线几乎重合。

(2) 多主机容器网络实验

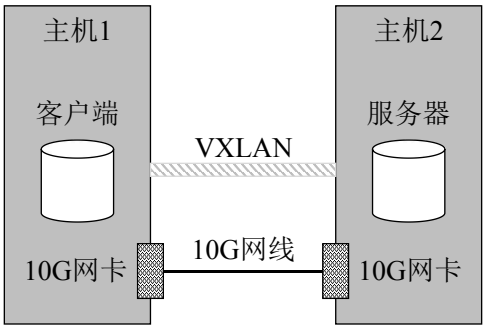

图 4-10 多主机容器网络实验环境示意图

多主机容器网络的实验环境如图4-10所示。容器网络通过 VXLAN 协议搭建，其中主机 1 运行客户端容器，主机 2 运行服务器容器，两主机均配置 10G 网卡，并通过 10G 网线直接相连。由于此时链路速率成为容器网络性能瓶颈，需设置两个容器的 CPU 利用率不超过 70%。本文首先针对 iptables 进行性能测试，在静态 NAT 规则的基础上，添加数据包过滤规则。接着，删除之前添加的所有规则，针对 BPF 和 BPF+XDP 分别进行性能测试。本文依次配置静态 NAT 规则和数据包处理规则，并通过自己编写的程序生成器生成 BPF 程序，然后通过 clang+llvm 编译器编译，并将产生的目标文件分别导入 TC 和 DMA 缓存中。为防止编译过程中发生地址聚合，采用随机数生成过滤规则中的源 IP 地址匹配项，保证实验的准确性。本文依次添加 0、100、200、...、900 条过滤规则，通过 netperf 工具测试多主机容器网络的吞吐量；测试过程中，客户端不断向服务器发送 16384 字节的数据包，每次测试的时长为 1 分钟。然后，本文在过滤规则数为 0、200、40、600、800 的条件下，通过 NetPIPE 工具测试多主机容器网络的单向链路时延。

多主机容器网络的实验结果如图4-11和图4-12所示。图4-11反映的是不同容器

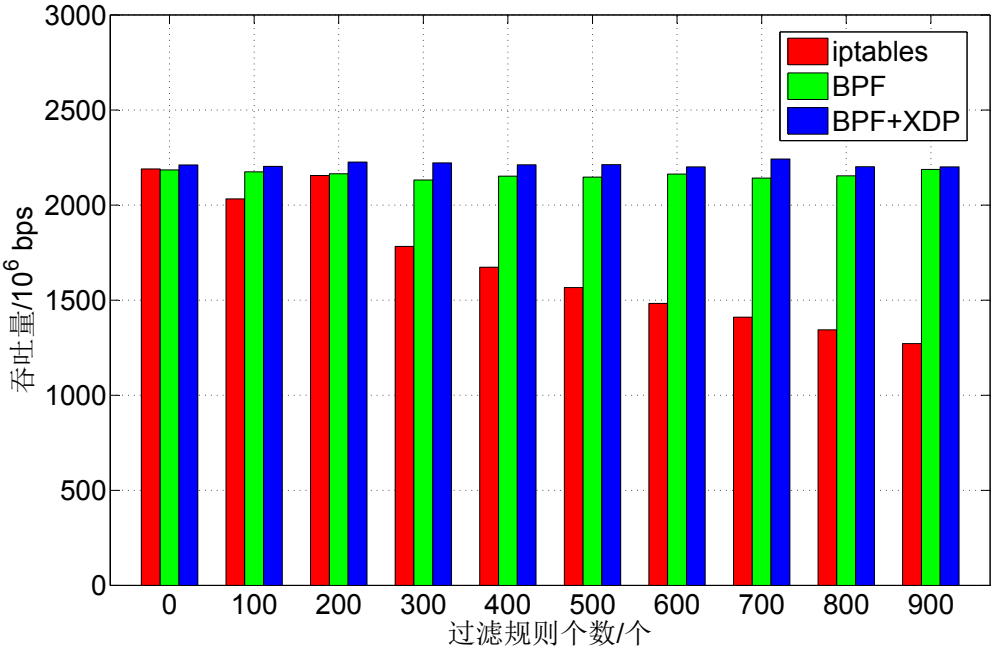

图 4-11 多主机容器网络的吞吐量柱状图

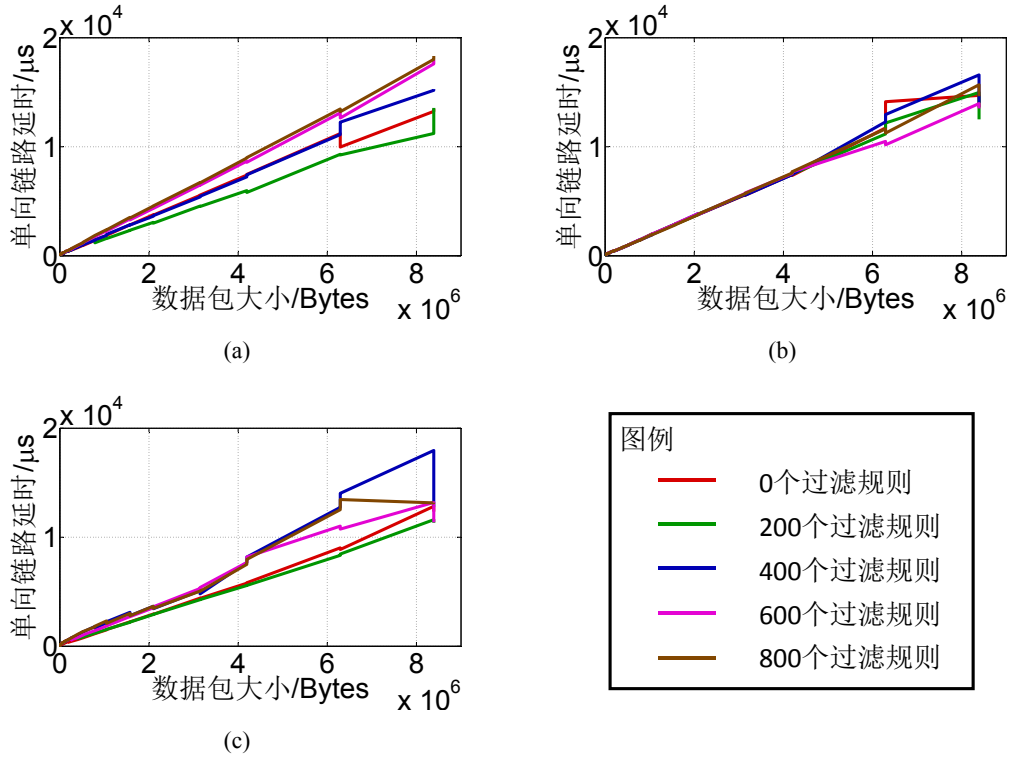

图 4-12 多主机容器网络的延时–数据包大小关系曲线图：(a) 基于 iptables 容器网络的曲线；(b) 基于 BPF 容器网络的曲线；(c) 基于 BPF+XDP 容器网络的曲线

网络、不同策略个数的条件下，容器网络吞吐量的变化情况。当容器网络中没有添加任何策略时，基于 iptables、BPF 和 BPF+XDP 的容器网络吞吐量非常接近，约为 2.2 Gbps。随着策略个数的增加，三种容器网络的吞吐量呈现不同的趋势。当规则数为 500 时，基于 iptables 容器网络的吞吐量只有 1.5 Gbps；当规则数达到 900 时，其吞吐量下降到 1.25 Gbps 左右。而当规则数在 0∼900 之间变化时，基于 BPF 和 BPF+XDP 容器网络的吞吐量均维持稳定；与此同时，在大多数情况下，基于 BPF+XDP 的容器网络在吞吐量上甚至优于基于 BPF 的容器网络。图4-12反映的是不同容器网络、不同规则数条件下，多主机容器网络的单向链路时延随数据包大小的变化曲线。由于物理链路的噪声，单向链路时延随数据包大小的变化不稳定。

单主机容器网络的性能主要取决于 CPU 处理数据包的能力。在 iptables 的配置过程中，随着规则数的增加，CPU 查询规则匹配项所占用的周期越长，CPU 的数据包处理能力也随之降低，因而吞吐量显著下降，单向链路时延明显增加；而 BPF 程序的运行实质上是汇编指令的运行，相比于查表，汇编指令的运行效率更高，因而，当过滤规则数在 0 ～ 900 之间变化时，基于 BPF 和 XDP 的单主机容器网络在性能上没有明显差异。然而，TC 上的 BPF 处理的是 sk_buff 中的数据，sk_buff 的处理过程比较复杂；而 XDP 中的 BPF 程序直接访问 DMA 缓存，数据读写效率高。因此，相比于 BPF，BPF+XDP 对数据包的处理更加高效。由于容器的通信均在主机内部，几乎没有引入任何链路噪声，因此，单向链路时延随数据包大小的变化比较稳定。

除 CPU 的处理能力外，多主机容器网络的性能还同时取决于链路速率和链路质量。若在实验过程中不对容器的资源利用率进行限制，那么本实验所测试的吞吐量几乎都维持在 9Gbps 左右。从图4-12看出，多主机容器网络的网络延时变化不稳定，这是因为物理链路可能会受到噪声的干扰，尽管噪声非常小，但本实验所测的单向链路延时是 1 μs 数量级的，因此，任何小的扰动都会对单向链路延时产生影响。

尽管实验中的容器网络规模非常小，远远没有达到数据中心网络中容器网络的规模，但是 BPF、BPF+XDP 与 iptables 之间的差异主要体现在软件的执行效率上。iptables 执行的是大量的查表，而 BPF 和 BPF+XDP 本质上是执行至多 4096 条汇编指令。因此，从执行过程来看，当规则数较多时，BPF 和 BPF+XDP 的优势更加凸显。同时，BPF 处理的是较为复杂的 sk_buff，而 BPF+XDP 直接访问并处理 DMA 缓存的数据，因而 BPF+XDP 的处理效率又得到进一步提升。因此，可以推断，在大规模的容器网络中，BPF 和 BPF+XDP 对容器网络同样具有性能上的提升

作用，并且 BPF+XDP 的性能提升作用更加显著。

综合以上结果，与传统容器网络相比，BPF 和 BPF+XDP 对容器网络性能具有提升作用；与此同时，在大多数情况下，BPF+XDP 的性能提升作用更加显著。

4.4 本章小结

本章先从软硬件环境配置、容器网络的搭建以及数据包处理规则的配置这三个方面阐述方案的具体实现过程，然后通过实验，验证基于 BPF 本文方案的可行性以及 BPF+XDP 对性能的进一步提升作用。实验结果表明，在网络吞吐量指标上，基于 BPF 的本文方案优于基于 BPF 的已有方案；BPF 和 BPF+XDP 对容器网络性能具有提升作用，并且 BPF+XDP 的性能提升作用更加显著。

第五章 结束语

5.1 总结

本文基于 Docker 容器平台、BPF 和 XDP，设计了容器网络性能的提升方案，并通过实验验证方案的可行性，具体研究内容及成果如下：

(1) 研究了 Docker 容器平台、传统容器网络的数据包处理过程、BPF 和 XDP 相关的背景知识。本文通过文献资料学习和掌握本课题相关的背景知识，了解了 bridge 驱动、host 驱动及 overlay 驱动创建容器网络的过程，分析了传统容器网络的性能瓶颈，并掌握 BPF 和 XDP 的工作原理；

(2) 对原有的容器网络性能提升方案进行了改进设计。本文针对 iptables 及基于 BPF 已有方案的不足，进行了方案的改进设计。同时，结合 XDP，进一步提升容器网络的性能；

(3) 通过系统的配置实现了性能提升方案。本文根据方案实现的具体流程搭建实验环境，安装相关软件，搭建容器网络，配置数据包处理规则，并通过实验验证基于 BPF 本文方案的可行性以及 BPF+XDP 对性能的进一步提升作用。实验表明，基于 BPF 的本文方案优于基于 BPF 的已有方案；BPF 和 BPF+XDP 对容器网络性能具有提升作用，并且 BPF+XDP 的性能提升作用更加显著。

5.2 展望

关于容器网络的性能优化，还有较多可行的方案，其中包括以下两种：

- 通过 Open vSwitch 组建容器网络。OVS 在内核中实现了虚拟交换机，通过 OpenFlow 协议对数据通路进行控制。这些虚拟交换机实现了更加高效地查表算法，使网络吞吐量在流表较多的情况下也不会受损严重，因此，OVS 是容器网络性能优化的一种可行方案；
- 通过 DPDK (Data Plane Development Kit) 优化容器网络的性能。在传统的网络中，CPU 将 DMA 缓存的数据拷贝到 sk_buff 中，并将此数据交由网络协议栈处理；然而，网络协议栈会引入一定的系统开销。DPDK 通过内核旁路技术，使网卡收到的报文直接到达应用层，从而减少网络协议栈的开销。

由于以上方案分别针对流表查询和数据通路进行改进，与上述方案相比，基于 BPF 和 XDP 容器网络的性能优劣还有待验证。因此，本课题还有进一步研究的空间。

致 谢

本文的主要工作在我的导师任丰原教授的指导下完成，在此衷心感谢任老师对我的悉心指导。与此同时，我也非常感谢校内代管老师马立香副教授对本课题的监督工作。此外，本文的主要工作也离不开 NNS 研究组师兄师姐的提点与帮助，对此，我深表谢意。

在毕业之际，感谢辅导员和所有任课老师对我学习、生活等方面的引导；最后，感谢父母在大学四年期间对我的支持和鼓励。

参考文献

[1] P. M. Mell, T. Grance. The NIST Definition of Cloud Computing[R]. Gaithersburg, MD, United States: National Institute of Standards & Technology, 2011

[2] 浙江大学 SEL 实验室. 容器与容器云（第 2 版）[M]. 北京: 人民邮电出版社, 2016, 1-129

[3] D. Hoffman, D. Prabhakar, P. Strooper. Testing iptables[C]. Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research, 2003, 80-91

[4] D. Borkmann. Cilium: Networking and Security for Containers with BPF and XDP[EB/OL]. https://opensource.googleblog.com/2016/11/cilium-networking-and-security.html, Nov. 2, 2016

[5] N. de Bruijn. eBPF Based Networking[R]. Netherlands: University of Amsterdam, July 16, 2017

[6] IO Visor. XDP[EB/OL]. https://www.iovisor.org/technology/xdp, 2016

[7] The Internet Engineering Task Force. 10.17487/RFC7348. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks[S]. America: RFC Editor, August 2014

[8] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment[J]. Linux Journal, 2014, 2014(239): 76-90

[9] M. Kerrisk. Namespaces - Overview of Linux Namespaces[EB/OL]. http://man7.org/linux/man-pages/man7/namespaces.7.html, Feb. 2, 2018

[10] S. Seth, M. Venkatesulu. TCP/IP Architecture, Design, and Implementation in Linux[M]. Washington D.C: IEEE Computer Society, 2008, 635-670

[11] J. H. Salim. Linux Traffic Control Classifier-Action Subsystem Architecture[C]. Proceedings of Netdev 0.1, Ottawa, 2015

[12] B. Hubert. TC - Show / Manipulate Traffic Control Settings[EB/OL]. http://man7.org/linux/man-pages/man8/tc.8.html, Dec. 16, 2001

[13] M. Kerrisk. Actions - Independently Defined Actions in TC[EB/OL]. http://man7.org/linux/man-pages/man8/tc-actions.8.html, Aug. 1, 2017

[14] D. Borkmann. BPF - BPF Programmable Classifier and Actions for Ingress/Egress Queuing Disciplines[EB/OL]. http://man7.org/linux/man-pages/man8/BPF.8.html, May 18, 2015

[15] Cilium. Cilium Concept[EB/OL]. http://docs.cilium.io/en/latest/concepts/, 2017

[16] J. Perry, A. Ousterhout, H. Balakrishnan, et al. Fastpass: A Centralized “Zero-Queue” Datacenter Network[J]. ACM SIGCOMM Computer Communication Review, 2015, 44(4): 307-318

[17] Z. Ahmed, M. H. Alizai, A. A. Syed. InKeV: In-Kernel Distributed Network Virtualization for DCN[J]. ACM SIGCOMM Computer Communication Review, 2016, 46(3): 22-27

[18] B. Pfaff, J. Pettit, T. Koponen, et al. The Design and Implementation of Open vSwitch[C]. 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, 2015, 117-130

[19] A. Belay, G. Prekas, A. Klimovic, et al. IX: A Protected Dataplane Operating System for High Throughput and Low Latency[C]. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, CO, 2014, 49-65

外文资料原文

eBPF Based Networking

N. de Bruijn

Report of University of Amsterdam, July 16, 2017, Page 3 – 5

Chapter 1 Introduction

Recently the use of containers increased in popularity, especially in the era of continuous deployment and development where containers are being used to create infrastructures based on microservices. Microservices are applications composed of many smaller loosely coupled services [1]. By separating large services into smaller services it gets easier to develop, maintain and deploy new services.

To support such infrastructure we need a lightweight approach, which will allow such services to run in a separated manner. Therefore, we see that microservices are usually being placed in so called containers. Containers can be seen as a lightweight equivalent of a virtual machine that only virtualises the processes of a physical machine and shares the underlaying OS [2]. To provide process isolation, each container runs in its own namespace [3] and uses Cgroups [4] for resource allocation. To provide a simple and automated way of deploying containers, several management tools have been developed to accomplish this. Examples of such management tools include LinuX Container (LXC) [5] and Docker [2].

For the interconnection of containers we need a network just like any other system would need. In order to do so different approaches can be taken. However, the traditional way is to connect the virtual interface of the container to a Linux bridge, and the Linux bridge to a physical interface. It is then possible to enforce traffic policies using iptables [6, 7]. Unfortunately, the main concern of this approach is that it relies on iptables which tend to have scalability issues in microservice architectures.

In a continuous deployment environment it is not unlikely that multiple microservices and therefore containers are being deployed or removed multiple times a day. Because containers use IP addresses as filter criteria there is a need to update the traffic policies

each time a change happens. This means that in the case of iptables new entries have to be made and old entries should be removed. This could lead to possible scalability issues whenever there are thousands of containers running.

A project called Cilium aims to overcome the scalability problem that arises with iptables by rethinking the way of container networking and traffic policy enforcements. Instead of using IP addresses to enforce security policies, labels are used [8]. By enforcing traffic policies based on labels it is possible to eliminate the need of iptables. The mechanism used for this is known as extended Berkeley Packet Filter (eBPF) [9]. eBPF programs run in the Linux kernel and have their own instruction set which allows extending kernel functionality.

Currently, Cilium uses eBPF programs to define a new way of processing packets and enforcing security policies at the traffic control (TC) layer [10]. This means that forwarding decisions can be made even before the traffic reaches the Linux network stack. It is expected that this early processing of packets and elimination of iptables increases the overall network performance of the container network.

1.1 Research Question

The contribution of this research is to **evaluate the usability of Cilium as a packet filtering system in a container infrastructure**. To help answering this research question we focus on the performance aspect:

- What throughput and latency we acquire in the case of using Cilium's eBPF program and Linux's iptables as packet filter?
- What effect does the number of iptables rule entries and Cilium policy entries have on the throughput and latency in both cases?
- Is there a turning point in performance when the number of traffic policies defined by Cilium policies or iptables rules increases?
- Does the complexity of Cilium's traffic policy influence the TCP throughput?

1.2 Report Outline

In Chapter 2 we present the most relevant related work for this research project. Chapter 3 explains background information on container networking. We will introduce eBPF in Chapter 4 and show how eBPF can be used in container networking. In chapter 5 we will discuss traffic policies and show how this can be applied in container networks

using iptables and Cilium labels. Our approach will be presented in Chapter 6 and contains information on the conducted experiments. The results of the experiments will be presented in Chapter 7. In chapter 8 we will discuss the results and make assumptions based on the results. Our conclusion is presented in Chapter 9. In Chapter 10 we discuss the open issues and suggest future work.

Chapter 2 Related Work

There has been a lot of research into evaluating the interest and the usability of container architectures over hypervisor based virtualization. In [11] R. Morabito, J. Kj¨allman, and M. Komu make a comparison between two hypervisor based virtualization techniques and two container based virtualization techniques. Another comparison was made in [12] by W. Felter, A. Ferreira, R. Rajamony, and J. Rubio making a performance comparison between native, Docker, and KVM. Both researchers conclude that containers reach better performance compared to other virtualization techniques.

There has also been done a large number of research evaluating the interest and the usability of containers for different use-cases. For example, in [13] C. Ruiz et al. evaluate the usability of containers in high performance computing and in [14], M. Amaral et al. evaluate the usability of containers in microservices architectures based on container performance.

As for traffic policy enforcement, in [15] D. Hoffman et al. show the impact of the number iptables rules on the throughput and latency. Furthermore, in [16] D. Hartmeier made a performance comparison of different packet filters indicating their effect on the network performance using different numbers of traffic policies.

Although the previously conducted research shows the performance benefits of containers and the impact of packet filters on network performance, no research exists that makes a clear performance comparison between different approaches of traffic policy enforcement in container based networks. Therefore, our contribution is to make a network performance comparison between a standard way of packet filtering using iptables and a new approach using BPF programs in a container network. The interest of doing so is mainly because research such as [17] and [18] show the performance benefits using BPF programs.

外文资料译文

基于 eBPF 的网络

第一章 引言

近年来，容器网络在各个方面，特别是在持续布署和发展的微服务架构中，被广泛使用。微服务是由很多较小的、松散耦合的服务组成的应用。将庞大的服务分为许多较小的服务后，服务的更新、布署与维护变得更加便捷。

为了支持这样的架构，需要一个轻量级的方法，将这些划分后的小服务运行在相对独立的管理体系下。这种相对独立的管理体系称为“容器”。容器可以被视为轻量级的虚拟机，它的轻量之处在于它只将进程虚拟化，与其它容器共享一个操作系统内核。为了隔离进程，每一个容器在各自的命名空间中运行，然后用 Cgroups 提供资源分配。目前，市面上已有不同的管理工具可以简单且自动地布署容器，如 LinuX 容器和 Docker 容器。

为了实现容器之间的互联，我们同样也需要网络。实现这样的网络有不同的方法。传统的方法是将每个容器的虚拟端口以及真实的物理接口连接到 Linux 网桥上，然后通过 iptables 实现容器网络的流量与安全策略。但是，在微服务的架构中，iptables 会引入扩展性问题。

在持续布署的为服务环境下，多次布署和移除容器是不可行的。由于 iptables 使用 IP 地址作为数据包过滤的条件，一旦容器网络的拓扑发生变化，iptables 就要重新配置。这意味着在 iptables 中，新的规则要添加进去，旧的规则要被移除。如果网络中有上千个容器在运行，iptables 所引入的扩展性问题将凸显出来。

开源项目 Cilium 就是来解决这个扩展性问题的，它重新考虑了容器组网的方式以及流量策略的执行方式。与基于 IP 地址执行流量策略的方式不同，Cilium 是基于标签执行流量策略的。这种机制被称为扩展性伯克利数据包过滤器 (extended Berkeley Packet Filter, eBPF)。eBPF 运行在 Linux 系统中，它拥有自己的指令集，因此可以进行功能扩展。

目前，Cilium 通过 eBPF，在流量控制 (Traffic Control, TC) 层上定义数据包过滤的新方法。这意味着在流量到达网络协议栈前，Linux 系统就已经做好了是否转发数据包的决定。可以期待，提前处理数据包并且限制 iptables 的使用将提升容器网络的整体性能。

1.1 研究问题

本文的贡献在于测试 eBPF 新型容器网络的可用性。对此，我们关注了以下几个问题：

- 在 eBPF 容器网络和 iptables 容器网络中，吞吐量和传输时延分别是多少？
- 在这两种容器网络中，流量策略的数量的增长会分别产生什么影响？
- 当流量策略的数量逐渐增长，这两种网络的性能是否存在转折点？
- Cilium 流量策略的复杂度是否影响 TCP 吞吐量？

1.2 本文结构

第二章介绍与本文相关的研究工作。第三章介绍容器网络的背景知识。第四章介绍 eBPF 以及 eBPF 在容器网络中的使用方法。第五章分别讨论用 iptables 和 Cilium 标签表示流量策略的方法。第六章重点展示我们的整个实验流程。第七章展示我们的实验结果。第八章对第七章的实验结果进行讨论，并做出合理的假设。第九章总结全文。第十章讨论了目前学术界尚未解决的问题，并对我们未来的研究工作进行展望。

第二章 相关工作

很多研究工作关注了基于虚拟机管理程序的容器网络架构的可用性。在 [11] 论文中，作者将两个基于管理程序的虚拟技术和两个基于容器的虚拟技术进行了比较。在 [12] 论文中，作者还比较了 Native、Docker 和 KVM 的性能。这两项研究均表明，相比于其它虚拟技术，基于容器的虚拟技术具有更好的性能。

与此同时，也有很多研究工作关注在不同应用场景下容器虚拟技术的可行性。例如，在论文 [13]，作者关注了在高性能计算领域中容器虚拟技术的可行性；在论文 [14] 中，作者关注了在微服务架构下容器虚拟技术的可行性。

关于流量策略的执行，论文 [15] 展示了吞吐量和传输时延随 iptables 策略个数的变化趋势。除此之外，在论文 [16] 中，作者将不同的数据包过滤器对网络性能的不同影响进行了比较。

尽管这些研究说明了容器的优势，也体现了不同的数据包过滤器对网络性能的不同程度的影响，但目前没有任何一个研究工作清晰地指出不同的流量策略执行方法对容器网络性能所造成影响的不同之处。因此，本文的主要贡献是在容器网络中比较了分别用 iptables 和 eBPF 执行流量策略所造成的不同程度的影响。我

们之所以想到这样做是因为论文 [17] 和 [18] 体现了用 eBPF 组建容器网络的性能优势。