MASTER THESIS XIDIAN UNIVERSITY

西安电子科技大学

硕士学位论文

用户态IPSec协议栈的研究与实现

作者姓名 吴 承 学校导师姓名、职称 权义宁 副教授

领 域 计算机技术 企业导师姓名、职称 刘建港 研究员

申请学位类别 工程硕士 提交毕业论文日期 2014 年 12 月

西安电子科技大学

毕业论文独创性（或创新性）声明

秉承学校严谨的学风和优良的科学道德，本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢中所罗列的内容以外，论文中不包含其他人已经发表或撰写过的研究成果；也不包含为获得西安电子科技大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中做了明确的说明并表示了谢意。

毕业论文与资料若有不实之处，本人承担一切的法律责任。

本人签名：吴承　　日　期：2014.12.23

西安电子科技大学

关于论文使用授权的说明

本人完全了解西安电子科技大学有关保留和使用毕业论文的规定，即：研究生在校学习期间论文工作的知识产权单位属西安电子科技大学。学校有权保留送交论文的复印件，允许查阅和借阅论文；学校可以公布论文的全部或部分内容，可以允许采用影印、缩印或其它复制手段保存论文。同时本人保证，毕业后结合毕业论文研究课题再撰写的文章一律署名单位为西安电子科技大学。

（保密的论文在解密后遵守此规定）

本论文属于保密，在_年解密后适用本授权书。

本人签名：吴承　　导师签名：

日　期：2014.12.23　日　期：2014.12.23

学校代码 10701 学 号 1203121781
分 类 号 TP39 密 级 公 开

西安电子科技大学

硕士研究生毕业论文

用户态IPSec协议栈的研究与实现

作者姓名 吴 承

领 域: 计算机技术

学位类别: 工程硕士

学校导师姓名、职称: 权义宁副教授

企业教师姓名、职称: 刘建港研究员

提交日期: 2014 年 12 月

Research and Implementation of User-Mode IPSec Protocol Stack

A thesis submitted to
XIDIAN UNIVERSITY
in partial fulfillment of the requirements
for the degree of Master
in Computer Technology

By
Wu Cheng
Supervisor: Quan Yining Liu Jiangang
December 2014

摘要

智能手机的普及带来了移动互联网的浪潮，用户数量与网络流量急剧增长。为了提供更好的服务，互联网公司都在积极建设数据中心(IDC)。分布于各地 IDC 的数据传输通常使用专线，但专线费用高昂，因此 IDC 的公网数据传输逐渐得到发展。为保证安全，IDC 通常通过 IPSec VPN 的方式在 Internet 上传输数据。IPSec VPN 解决方案基于传统协议栈。在高速网络环境下，传统协议栈系统性能已经到达一个瓶颈，用户态协议栈成为热门研究课题, Intel DPDK 是一款优秀的用户态协议栈开发平台。

本文针对传统协议栈在数据处理过程中面临的中断频繁、数据冗余拷贝、不支持多核框架、锁竞争开销大等问题，详细介绍了 DPDK 中解决这些问题的关键技术：大内存页、用户空间 I/O 与处理器亲和性。然后设计了基于 DPDK 的用户态协议栈框架，并详细介绍了其中的四个主要模块。底层驱动模块负责对多核与分布式存储提供支持。数据收发模块负责快速的收发数据、存储数据包与减小多核竞争开销。三层转发协议栈模块提供路由转发以及用户态协议栈与传统协议栈的通信。IPSec 处理模块提供数据包的认证与加解密。基于各个模块的详细设计，本文实现了一个基于 DPDK 的用户态 IPSec 协议栈。通过在高速网络环境下与传统协议栈的 IPSec 进行测试对比，可以得到结论，用户态 IPSec 协议栈具有更高的性能，能够解决传统协议栈的问题。

关键词：用户态协议栈， DPDK， 多核， IPSec

论文类型： 应用基础技术

ABSTRACT

The popularity of smart phone brings a fresh wave of mobile internet. Not only the number of clients but also the network traffic has a dramatic increase. In order to provide better quality of services, major internet companies are actively building data centers (IDC). Data transfers between IDC of scattered departments is usually completed using private lines with high cost, thus the public network has gradually played a role in data transfers between IDC. To ensure safety, IPSec VPN is often applied to transfer data for IDC in public network. IPSec VPN solution is based on traditional protocol stack, which has already approached a bottleneck under high speed network environment. User-mode protocol stack has become a popular research subject, while Intel DPDK is an excellent development platform for user-mode protocol stack.

To deal with problems faced in data processing for traditional protocol stack such as frequent interruptions, redundant data copies, not supporting multi-core framework and high cost of lock contention, this thesis introduced the key techniques to fight with these problems in DPDK: large pages, user space I/O and processor affinity. Then a user-mode protocol stack framework based on DPDK is designed and four major modules are introduced in detail. The driver module is responsible for supporting multi-cores and distributed storage. The data receiving and dispatching module is responsible for quick data receiving and dispatching, data storage and less multi-core competition. The third layer protocol stack module is responsible for route switchover and communication between kernel-mode protocol stack and user-mode protocol. The IPSec processing module is responsible for the authentication and encryption for data packets. Based on detailed design of each module, a user-mode IPSec protocol stack based on DPDK is complemented. Tests against IPSec in traditional protocol stack in high speed network environment can provide the conclusion that user-mode IPSec protocol stack has superior performance and the ability to solve problems faced in traditional protocol stack.

Keywords : User-mode protocol stack, DPDK, multi-core, IPSec

Type of Dissertation: Applied Basic Technology

插图索引

插表索引

缩略语对照表

缩略语	英文全称	中文对照
IDC	Internet Data Center	互联网数据中心
VPN	Virtual Private Network	虚拟专用网
DPDK	Data Plane Development Kit	数据空间开发包
TOE	TCP/IP Offload Engine	TCP/IP 卸载引擎
OSI	Open System Interconnection	开发系统互联
UIO	Userspace Input and output	用户空间输入输出
AH	Authentication Header	认证头
ESP	Encapsulating Security Payload	载荷封装
IKE	Internet Key Exchange	英特网密钥交换
SMP	Symmetric Multi-Processor	对称多处理器
UMA	Uniform Memory Access Architecture	统一内存访问
NUMA	Non Uniform Memory Access Architecture	非同一内存访问
SA	Security Association	安全关联
SPI	Security Parameter Index	安全参数索引

目录

第一章 绪论

1.1 研究背景

随着网络通信技术的快速发展，尤其是近几年移动互联网的崛起，智能手机的普及，用户数与网络流量呈现井喷式增长。数据量的大规模增长，使传统互联网下的性能瓶颈越发的突显。各大互联网公司为了保证能够快速的服务用户，以及增强数据的安全性，都在积极建数据中心(IDC)。

IDC 的全称是 Internet Data Center[1]，是公司机构建立的 Internet 数据中心，数据中心包含收发、处理、存储数据的硬件设备，并通过这些设备提供相关的服务。由于分布各地的 IDC 之间的数据需要相互备份来保障就近服务与冗余备份，因此数据吞吐量十分巨大。IDC 之间的数据传输往往通过运营商提供的专线来进行传输，但专线高昂的成本对企业来说是巨大的负担，因此越来越多的人研究 Internet 下大数据的高速传输。

Internet 作为人类 20 世纪最伟大的发明，已经深入的走入了人类的生活。随着 Internet 的组成日趋复杂，数据安全性受到更加严峻的挑战。为了保证数据的安全传输，IDC 在公网 Internet 传输数据常常通过 VPN 的方式。IPSec VPN[2]是目前使用十分广泛的一种 VPN。但 IPSec VPN 解决方案都是基于传统 TCP/IP 协议栈。随着数据规模的巨幅增长，协议栈遭遇了性能瓶颈，导致目前的 VPN 解决方案不能满足 IDC 之间的海量数据传输需求。经过 IPSec VPN 的数据流吞吐率衰减严重，不能满足目前 IDC 之间大数据的高速传输。协议栈的性能主要受以下三方面因素的影响。

（1）协议栈处理开销

在传统协议栈里，网络数据包到达网卡后，网卡驱动产生一个中断告知内核；内核将数据包从网卡缓冲区复制到内核缓冲区中；协议栈每个层次对数据包进行处理，最后将数据包拷贝到应用程序的用户态缓冲区。在协议栈的整个处理流程中，中断通知、数据包的拷贝、用户态和内核态的切换等阶段是主要的处理开销，这些阶段成为了性能提升的瓶颈。

（2）传统协议栈不能发挥多核处理器框架的优势[3]

随着用户对网络服务的质量需求不断提升，传输更快的光纤已经接入了千家万户。网络传输速率从 10Mbps 发展到目前的 10Gbps 主干网络。根据传统协议栈下“1bit 网络数据消耗大约 1Hz 的 CPU 处理能力”的理论[4]，在 10Gb、40Gb 网络已经开始普及情况下， CPU 的主频需要达到 10GHz 到 40GHz，才能完全满足高速网络的处理能力。然而近几年，由于受处理器生产技术的制约，摩尔定律已经不再有效，处理器主频达一个瓶颈，单核处理器下的 TCP/IP 协议栈已经无法

满足高速网络下的海量数据传输要求。在 CPU 主频无法进一步提升的情况下，多核 CPU 成为目前处理器的主要发展方向，将多个核集中到一个处理器上可以有效提升单核处理器的性能。但是伴随着带宽和硬件快速发展的 20 年，网络协议栈的核心基础还是上世纪 80 年代的 TCP/IP 协议栈，不能满足目前的多核框架。

（3）内核协议栈虽然很完善地实现了 TCP／IP 协议栈的各个协议层次，但是由于协议栈含有许多层次与不同的协议，使得协议栈的处理流程日趋复杂，处理时延不断增大。内核协议栈提供的可靠支付，在网络质量和速度明显提高的环境下，优势越来越小，反而过大的时延与昂贵的处理开销等副作用越发凸显，制约了协议栈的处理性能。

因此，越来越多的人开始研究在用户空间层次上实现 TCP／IP 协议栈。通过绕过操作系统内核，可以在用户空间直接访问网卡缓冲区数据包。避免协议栈在处理数据包时，需要数据包在内存中多次复制。针对具体的业务需求，可以精简协议栈处理流程，只保留部分协议处理流程，并添加自己需要的协议流程。因此与传统协议栈相比，用户态协议栈拥有出色的定制性和较强的扩展性。

DPDK[5]是 Data Plane Development Kit 的缩写，是 Intel 推出的针对数据包快速处理的开发套件，包含了一系列的包处理库与驱动程序。DPDK 的主要技术点是大内存页、用户空间 I/O 和处理器亲和性。DPDK 不是协议栈，但能利用它快速的开发针对数据包处理的用户态程序。为了解决目前使用基于传统协议栈的 IPSec VPN 传输数据时的开销过大，造成 IDC 之间的吞吐量过低的问题，本文研究与实现基于 DPDK 的用户态协议栈，并实现其中的 IPSec 功能。

1.2 课题研究的意义

协议栈通常都集成在操作系统的内核中。在高速网络环境下，传统协议栈面临着中断频繁、内存复制次数过多、功能冗余、裁剪困难等缺点。通过构建用户态协议栈，弥补了传统协议栈的这些缺点，以满足高速网络下的性能需求。

本课题研究基于 DPDK 的用户态协议栈，并实现了其中的 IPSec 功能。旨在通过比较用户态协议栈的 IPSec 协议与传统协议栈的 IPSec 协议之间性能的差别，来深刻了解用户态协议栈在面对传统协议栈无法满足高速网络需求的解决方案[6]。

(1)用户态协议栈避免了频繁的系统调用与上下文切换。在 Linux 系统下，用户进行 socket 网络通信是通过系统调用的方式。系统调用以软件中断 int 指令进入内核，然后进行从用户态到内核态的上下文切换。当系统调用结束后，再从内核态切换回用户态。用户态下的协议栈运行在用户空间里，避免了用户态和内核态的切换，有效的减少了开销。

(2)用户态协议栈避免数据包在协议的栈处理流程中的多次拷贝。在传统的TCP/IP 协议栈中，网络数据包到达网卡后，网卡驱动产生一个中断告知内核，内核将网络数据包从网卡拷贝到内核空间里的缓冲区，协议栈对数据包进行各层次的处理后，数据包中的载荷部分最终被复制到用户程序的缓冲区中。用户态协议栈通过零拷贝等技术可以减少复制数据包的次数，在高速网路环境下可以显著减少内存拷贝的开销。

(3)用户态协议栈具有良好的动态扩展性与定制性。Linux 协议栈完整的实现了各层协议，适用性广，稳定性高。但是在某些特定的环境和业务需求下，并不需要 Linux 协议栈中的全部处理流程，只需要实现特定的协议功能即可，许多功能都过于冗余。用户态协议栈可以根据具体的需求，对传统协议栈进行裁剪，并添加自己的协议处理流程。同时，在用户空间开发调试也比内核协议栈简单方便。

1.3 国内外研究现状

在通过硬件来加速协议栈的处理速度上，TOE 是一种被广泛使用的加速方案，TOE 的含义是 TCP/IP 卸载引擎，即将协议栈的全部或者部分任务交给具有处理能力的硬件来执行。TOE 可以比较有效的提升协议栈的处理能力，减轻处理器的负荷。但是，一般的硬件加速方案对硬件都有特殊的要求，例如具有处理能力的网卡。这些硬件的成本昂贵，不被普通消费者所认可，不能大规模的普及。因此，用户态协议栈成为了备受关注的研究领域。

最近几年，国内外的各大高校与研究机构纷纷开始研究用户态协议栈的相关技术，并且取得了一些颇具借鉴意义的创新成果[7]。

华盛顿大学设计实现了一个基于 FreeBSD 协议栈的用户态框架 Alpine，它有效的解决了传统应用程序移植到用户态协议栈上需要做过多改动的问题。Alpine 将内核空间的服务虚拟化为一个用户空间的协议栈，并且能够共享内核状态。Alpine 减少了接收与发送数据包的周期，并且在用户空间调试协议栈更加方便[8]。

华盛顿大学在用户态协议栈方面的另一个研究成果是设计了一个协议栈框架。网络接口层、多线程库和操作系统层是该协议栈框架的主要组成部分。网络接口层提供了用于数据包接收与发送的 API 接口，操作系统服务层的主要任务是完成建立连接，释放连接以及处理 ARP 数据包的请求等。操作系统服务层负责协议状态的维护，例如 TCP 状态和动态路由表的信息；多线程库实现了对数据包的收发功能[9]。

Wattcp 是加拿大的 Erick Engelke 在 DOS 下开发的一个用户态协议栈 。Wattcp 实现了协议栈中传输层和网络层的处理流程，包括传输层协议的相互交互，

网络层数据包的重组分片；在另一方面，为了用户程序的无缝迁移，它也实现了兼容 BSD 网络 套接字层的 Socket[10]。

OpenOnload[11]是来自于 Solarflare 的高性能网络协议栈。OpenOnload 一方面有效的减少了延迟和处理器的负荷，另一方面有效的提高了数据速率和带宽。Openonload 协议栈运行在 Linux 操作系统上，实现了标准的 BSD 套接字 API，提供对 TCP/IP 网络协议栈的透明传输，用户的网络程序无需更改就可以运行在 OpenOnload 协议栈上。通过在数据传输路径上完全绕过操作系统内核，OpenOnload 在用户空间实现了网络数据包处理能力的提升。同时，没有造成安全性的降低，并且也支持传统协议栈的复用功能。

在国内，清华大学设计了一个用户空间的通信协议 RCP。在数据传输路径上，RCP 绕过操作系统内核，使网络设备和用户空间直接互通。RCP 协议与网卡设备之间的通路，减少网络数据包传递到用户空间的复制次数，提高了网络吞吐量。同时，通过 RCP 提供的接口，进行自定义的编程也十分简单。RCP 支持 PVM 并行环境，只需少量的修改， PVM 的原有应用程序即可与 RCP 协议相互交互 [12]。

1.4 论文主要工作

本文旨在通过 DPDK 开发用户态协议栈，并实现其中的 IPSec 功能，具体工作如下。

本文首先概括了传统 TCP/IP 协议栈在高速网络环境下的不足，并罗列了目前主流的协议栈加速方案；接下来介绍了 DPDK 解决传统协议栈缺点的关键技术，给出了一种基于 DPDK 实现的用户态协议栈方案。在方案设计时，通过运行环境抽象层提供了对多核的动态支持和负载均衡，用户空间的轮询使用户程序快速的收发数据，避免中断。无锁队列解决了多核对共享数据访问的锁竞争开销过大，KNI 接口提供了用户态协议栈与传统协议栈之间的交互通道。最后在 10Gbps 高速以太网与传统协议栈的 IPSec 方案进行了对比测试。实验结果表明基于 DPDK 的用户态协议栈可以有效提高系统的吞吐性能。

1.5 论文章节安排

本文的内容结构划分了六章，每一章节的主要内容如下所示。

第一章为绪论。首先简要地介绍下研究用户态协议栈的意义，以及目前学术界的研究情况与 DPDK 的背景，并简要介绍了下本文的主要内容以及文章的组织结构。

第二章研究了传统协议栈面临的问题、目前主流的协议栈加速方案以及 DPDK。首先叙述了协议栈的架构以及造成高速网络下传统协议栈瓶颈的因素，

接下来介绍了目前主流的三种协议栈加速方案。然后简要的介绍了并行计算的相关理论与 DPDK 在解决传统协议栈的问题的关键技术，最后叙述了目前 VPN 的常见实现方法。

第三章详细介绍了基于 DPDK 的用户态协议栈设计方案。基于设计方案，依次详细的描述了底层驱动模块、数据收发模块、三层转发协议栈模块、IPSec 处理模块等。为以后的实现打下了框架，奠定了基础。

第四章实现了基于 DPDK 的用户态协议栈。依据第三章的设计框架，首先讲述了底层驱动模块的初始化过程，然后罗列了数据收发模块、三层转发协议栈模块、IPSec 处理模块的核心代码，实现了一个完整的基于 DPDK 的用户态 IPSec 协议栈。

第五章是性能测试与分析，在高速网路环境下对开发的用户态 IPSec 协议栈和传统的 IPSec 方案进行测试对比。介绍了测试环境、测试结果，最后根据结果给出了结论。

最后第六章为总结与展望。总结本文的工作，罗列了不足，并展望了以后的研究工作。

第二章 协议栈加速与 DPDK 简介

本章研究了传统协议栈面临的问题、目前主流的协议栈加速方案以及 DPDK。首先叙述了协议栈的架构以及造成高速网络下传统协议栈瓶颈的因素，接下来介绍了目前主流的三种协议栈加速方案。然后简要的介绍了并行计算的相关理论，然后介绍了 DPDK 在解决传统协议栈的问题的关键技术，最后叙述了目前 VPN 的常见实现方法。

2.1 协议栈的基本概念

协议栈是指网络中各层协议的总和，形象的反映了数据在网络中的传输过程。协议栈划分了多个层次，每个层次都完成特定的功能，下层协议为上层协议提供服务原语，每一层都建立在下层结构上。发送端数据包从上层协议到底层协议，接收端再从底层协议到上层协议，组成了一个完整的数据包处理流程。

2.1.1 传统协议栈模型

国际标准化组织首先提出了 OSI 协议栈参考模型[13]。OSI 定义了网络的七层框架，然后详细的定义了每一层的功能，OSI 参考模型有七层结构，每一层的功能、处理的对象如表 2.1 所示。

TCP/IP[14]协议栈是英特网使用的参考模型，TCP/IP 自上倒下可以分为四层，分别是应用层、传输层、网络层和主机到网络层，OSI 参考模型和 TCP/IP 协议栈之间的对应关系如图 2.1 所示。

主机到网络层，该层协议未定义，随网络和主机的不同而不同，因此又被称为网络接口层。主机到网络层位于 T C P / I P 模型的最底层，负责将从高层传下来的数据报发送到物理链路上，或者负责从底层链路上接收数据报文，进行校验和帧头结构剥离，将数据报文载荷部分投递到高层网络层。网络接口层主要分为两种。第一种是包含各种数据链路协议的子系统，这个子系统类似 OSI 参考模型中的数据链路层。第二种是网卡设备的驱动程序，例如各种不同的网卡设备。TCP/IP 模型没有定义数据链路层，是因为在设计 TCP/IP 模型的时候，包括了令牌环网、以太网、X.25、FDDI 网和 ISDN 等多种数据链路层协议。

互联网层的主要功能是负责相邻数据的路由传送。它的功能包括以下三个方面。第一个功能是将来自上层的传输层数据报文，填充上网络层的 IP 报头，进行路由选择，选择到达下一跳网络节点的路径，然后将数据报文发送到相应的网络接口上。第二个功能是对应从下层传上来的数据报文，根据 IP 层头部的校验和来判断数据包是否有效，然后再判断下一跳路由，如果该数据包的目的地址就是本机，那么将 IP 层头部剥离后，交给上层的传输层协议，否则如果数据报的目的地址不是本机，则根据路由，将数据包投递到下一个节点。第三个功能是处理

ICMP（因特网报文控制协议)报文，ICMP 主要用来反馈网络是否通畅，主机是否可达和网络路由是否可用等信息。

表 2.1 OSI 协议栈模型

协议层	功能	数据单元
应用层	直接向用户提供服务，完成用户希望在网络上完成的各种工作，负责对软件提供接口以使程序能使用网络服务	数据Data
表示层	提供格式化的表示和转换数据等服务	
会话层	向两个实体的表示层提供建立和使用连接的方法	
传输层	向用户提供可靠的端到端的差错控制和流量控制，保证报文的正确传输，传输层向高层屏蔽下层通信的细节	报文Segment
网络层	通过路由选择算法，为数据包选择最适当的路径。该层控制数据链路层与传输层之间的信息转发，建立、维持和终止网络的连接	数据包Packet
数据链路层	负责管理节点间的链路，通过各种控制协议将有差错的物理信道变为无差错的、能可靠传输数据帧的数据链路	帧Frame
物理层	利用传输介质为数据链路层提供物理连接，实现比特流的透明传输，涉及机械、电子时序接口、以及传输介质	比特Bit

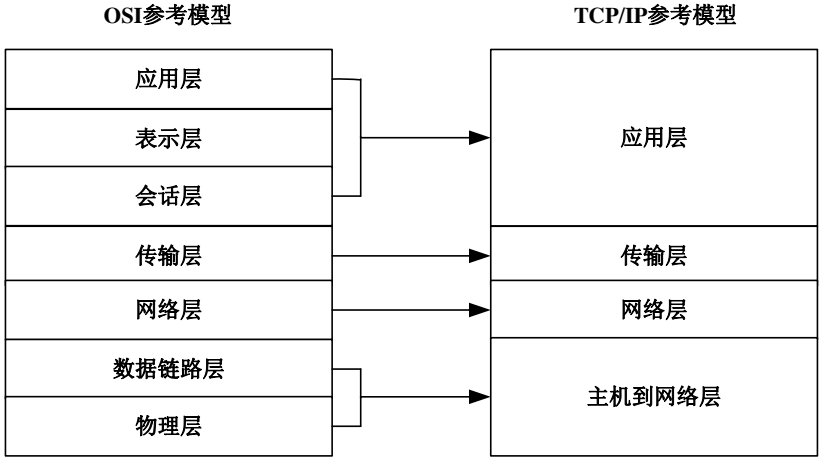

图 2.1 TCP/IP 参考模型

互联网层是提供了无连接的不可靠分组交换服务，它是网络相互连接的基础。互联网层的的其任务是将分组报文独立的发送出去，由于网络情况在动态的变化，每个分组的传递路径也有所不同。分组到达对方的顺序与分组报文的发送顺序不同，分组的排序是就交给上层的传输层进行。另外一方面，为了避免分组大小过大，传递给数据链路层后，超过了链路的最大传输长度(MTU)限制,互联网层负责对分组进行分片和重组。

传输层是整个 TCP/IP 协议体系中最关键的一环，是唯一总体负责数据的传输和控制的一层。传输层的核心协议是 TCP 和 UDP。TCP 提供面向连接的可靠通信服务。TCP 负责进行流量控制，拥塞控制，对接收到的数据报文进行确认，已经对没有按序到达的分组进行排序。而 UDP 提供面向报文的不可靠传输，UDP 不对数据包进行确认，也没有流量控制等功能，它负责尽最大努力将数据报文发送到对端。

应用层，TCP/IP 参考模型中没有会话层与表示层，在传输层之上就是应用层，它包含了所有高层应用程序协议，如 HTTP、FTP 等[15]。

2.1.2 现有协议栈加速技术

1. 零拷贝技术

零拷贝[16]就是一种避免将数据在内存空间相互复制的技术。零拷贝技术主要用于网络协议栈，文件系统以及设备驱动程序。通过使用零拷贝技术，应用程序的重复冗余拷贝操作减少，性能有了明显的提升，并且对于操作系统的资源，使用起来更加的合理。同时，性能的提升是往往通过 DMA 的方式，数据拷贝不占用 CPU，CPU 可以同时执行其他的进程。零拷贝技术减少了数据对总线的占用时间以及复制的次数，避免了数据报文在内存之间不必要的复制拷贝，因此数据传输效率得到了有效的改善。另外一方面，内核态缓冲区和用户态缓冲区相互拷贝数据，会导致用户态和内核态之间的上下文切换。零拷贝技术减少了数据拷贝的次数，因此上下文切换的开销也减小了许多。另外如果让计算能力强大的 CPU 去执行时间 IO 很长的大量数据拷贝操作，这样一个简单的任务对 CPU 来说是一种极大的浪费；通过把这部分简单的任务转转移到其他硬件上，让 CPU 负责计算型的复杂任务，则可以更加有效的利用系统资源。综上所述，零拷贝技术的主要技术特点总结为下：

(1) 避免多余的数据拷贝

① 避免内核的数据缓冲区和应用程序的数据缓冲区相互拷贝数据，带来昂贵的上下文切换开销。

② 避免操作系统内核地址空间的数据缓冲区相互进行数据的复制拷贝。

③ CPU 避免等待长时间的数据 IO，而是将数据通过 DMA 的方式传输拷贝。

④ 用户程序和硬件驱动程序形成数据通路，使用户程序可以直接访问其数据空间。

(2) 结合多种操作

① 尽量将处理数据的任务卸载到硬件上进行

② 提前缓存要拷贝的数据

③ 避免进行系统调用来消除用户态和内核态进行昂贵的上下文切换所造成的开销。

由于目前 10Gb、40Gb 的高速网络已经开始逐渐普及，伴随着移动互联网的井喷式发展带来流量的极速增长。网卡设备每收到一个数据包，就要用中断的方式通知 CPU。CPU 然后将数据包从网卡设备驱动程序对应的缓冲区拷贝到内核协议栈的缓冲区。在高速网络下，CPU 需要花费许多时间来进行数据包的复制拷贝，导致给予其他进程任务的时间减少，或者由于 CPU 来不及处理数据包，导致数据包的严重丢包。根据理论，1bit 网络数据需要消耗大约 1Hz 的 CPU 处理能力”的理论，在 40G 的主干网开始建设的情况下，处理器的主频需要达到 40GHz，才能完全满足高速网络的处理能力，因此在高速网络下，零拷贝技术的作用就越发的突显。由于目前大型的 IDC 数据中心之间的主干网络的速率已经达到 10Gb 以上，零拷贝技术也在各个数据中心中发挥作用，来降低 CPU 的负载，提高数据吞吐率。而且，目前普通用户的网络接入速率越来越高，光纤已经走进了千家万户。未来，普通用户也可以用上 1Gbit 以上的高速网络，零拷贝技术发挥的作用会越来越大。在传统的 Linux 协议栈中，数据包从网卡到达用户程序的缓冲区，往往需要进行至少两次拷贝。通过使用零拷贝技术，应用程序直接从网卡缓冲区将数据拷贝到应用程序缓冲区，数据复制拷贝的次数减少，进一步的精简协议处理流程，应用程序能够更加快速的将数据包发送到物理链路上，数据发送与接收的延迟进一步缩小，网络吞吐量得到进一步的跳。在目前路由器、交换机等高速网络设备中，零拷贝是主要的技术之一。

零拷贝技术经过多年的发展，存在多种种类，但目前没有一种在各种情况下都适用的通用技术。随着 Linux 不断推出更加稳定成熟的版本，零拷贝技术也发生了多种变化。针对所使用的场景，零拷贝技术可以被分为多种 Linux 中的零拷贝技术可以被划分为三种：

(1) 避免传输数据的时候，操作系统内核空间和用户空间进行相互复制。当应用程序只对数据进行传输，不对他进行访问，那么就可以完全避免把数据从内核的页缓存复制到用户进程的缓冲区，可以直接在内核页缓存中处理数据。通过减少缓存之间的复制次数，可以有效的提升性能。Linux 中通过 sendfile、splice 和 mmap 等系统调用来提供相似的功能。

(2) 直接 I/O：就是硬件的缓冲区完全暴露给应用程序，使应用程序可以直接访问，在整个数据传输中，操作系统只是扮演一个辅助的作用。通过绕过操作系统，操作系统内核不对数据包进行传输，硬件和应用程序之间形成数据通路，应用程序可以直接从硬件的缓冲区中拷贝数据。

(3) 优化用户进程的缓冲区和内核的页缓存之间的数据传输。这种零拷贝技术主要用于处理数据在操作系统的页缓存和用户进程的缓冲区之间的相互拷贝。它使数据传输更加灵活，这种技术主要使用了写时复制。

这三种方法各有侧重点，前两种方法通过避免操作系统内核地址空间和应用程序的地址空间之间进行数据的复制拷贝。在一些场景下，如数据不需要经过应用程序的处理或者操作系统内核的处理下，常使用这两种技术。第三种方法是在传统的数据传输的基础上，进而针对数据传输本身进行优化。软件从硬件收发数据可以采用 DMA 的方式，DMA 传输数据与处理器运行其他程序可以同步进行。但 DMA 不能应用于用户应用程序地址空间缓冲区和内核缓冲区之间数据的复制拷贝，此时必须要有处理器来完成。通过第三种零拷贝技术可以减少处理器参与的频率，有效的提升传输效率。

2. 协议栈硬件加速方案

在通过硬件来加速协议栈的处理速度上，TOE 是一种被广泛使用的加速方案，TOE 的含义是 TCP/IP 卸载引擎，即将协议栈的全部或者部分任务交给具有处理能力的硬件来执行[17]。这种技术减轻了 CPU 的工作负荷，使 CPU 不把时间集中在数据拷贝，协议栈处理等工作，有效减少中断操作、数据复制、协议处理的开销，另外因为硬件的执行速度快，解决了高速网络下传统协议栈的瓶颈问题。TOE 分为完全卸载和部分卸载[18]。

完全卸载是硬件完全完成整个协议栈的工作，但因为牵涉了操作系统管理用户线程的机制，面临的问题是 CPU 和 TOE 网卡的处理器对用户地址空间同时访问造成的冲突，同时需要解决物理内存地址与虚拟内存地址的相互转换。另外为了和 TOE 网卡进行通信，需要实现位于用户程序和网卡设备驱动之间的中间接口。完全卸载实现面临的困难很多，它需要修改操作系统的存储管理与线程管理等相关机制，具有一定的高风险。

相比完全卸载，部分卸载只是由硬件来执行部分的协议栈工作。因为部分卸载只是参与了一部分的协议栈工作，不涉及操作系统的存储管理、线程管理等，协议栈的主要工作还是要 CPU 来完成，例如连接的建立和关闭、状态的维护等。协议栈的校验卸载是一种最常见的 TOE 部分卸载技术，是把计算量较大的数据包校验工作交给具有一定计算能力的网卡。另外还可以让网卡完成数据包的重组工作，将重组好的数据包交给内核协议栈，以此来减轻处理器的重组开销。在 1Gb

高速网络进行测试，部分卸载技术可以减少 2%-7%左右的 CPU 处理开销；在 10Gb 的高速网络下，可以减少大约 20%的 CPU 处理开销。

3. 用户态协议栈加速技术

由于在高速网络下，传统协议栈的数据经过多次的拷贝以及多次系统调用严重影响协议栈的性能。相比使用硬件加速方案，许多人研究把传统协议栈移植到用户态下。在用户态实现协议栈有以下几点优势[19]。

(1) 用户态协议栈避免了频繁的系统调用。在 Linux 系统下，用户进行 socket 网络通信是通过系统调用的方式，系统调用以软件中断 int 指令进入内核，然后进行从用户态到内核态的上下文切换。当系统调用结束后，再从内核态切换回用户态。用户态下的协议栈运行在用户空间里，避免了用户态和内核态的切换，有效的减少了开销

(2) 用户态协议栈避免数据包在协议栈处理流程中的多次拷贝。在传统的 TCP/IP 协议栈中，网卡收到数据包后，首先通过硬件中断的方式通知处理器，处理器将网络数据包从网卡拷贝到内核空间里的缓冲区，协议栈对数据包进行一些列的处理后，数据包被复制到用户程序的缓冲区中。用户态协议栈通过零拷贝等技术可以减少复制数据包的次数，在高速网路环境下可以显著减少内存拷贝的开销。

(3) 用户态协议栈具有良好的动态扩展性与定制性。Linux 协议栈完整的实现了各层协议，适用性广，稳定性高。但是在某些特定的环境和业务需求下，并不需要 Linux 协议栈中的全部处理流程，只需要实现特定的协议功能即可，许多功能都过于冗余，用户态协议栈可以根据具体的需求，对传统协议栈进行裁剪，并添加自己的协议处理流程。同时，在用户空间开发调试也比内核协议栈简单方便。

A1 是用户空间传输协议，它是由 Glenford Mapp 等人实现，A1 是面向数据块的传输，并且使用了 TCP 中的选择性重传，然后显式的测量往返时间 RTT，采用端到端的流控方法，并且具有 UDP 的组播功能。通过实验表明， Al 的网络吞吐量相比传统的 TCP 实现有了明显的提高。

OpenOnload 是来自于 Solarflare 的高性能网络协议栈，OpenOnload 一方面有效的减少了延迟和处理器的负荷，另一方面 OpenOnload 有效的提高了数据速率和带宽。OpenOnload 协议栈运行在 Linux 操作系统上，实现了标准的 BSD 套接字 API，提供对 TCP/IP 网络协议站的透明传输，用户的网络程序无需通过更改就可以运行在 OpenOnload 协议栈上。通过在数据传输路径上完全绕过操作系统内核，OpenOnload 在用户空间实现了网络数据包处理能力的提升，同时，没有造成安全性的降低，并且也支持传统协议栈的复用功能。OpenOnload 的架构如图 2.2 所示。

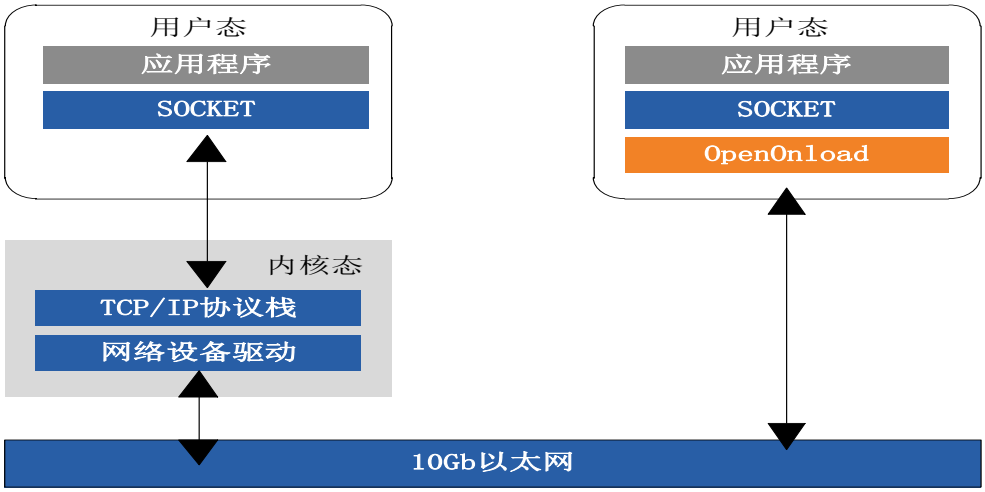

图 2.2 OpenOnload 架构图

DPDK 是 Intel 推出的针对数据包快速处理的开发套件。运行在 X86 平台下的 Linux 环境，利用 DPDK 可以快速的开发出数据包处理程序，例如针对某具体需求的协议栈。DPDK 的主要技术技术核心是 Hugetlb[20](大内存页)、UIO[21](用户空间 I/O)、CPU Affinity[22](处理器亲和性)。DPDK 对并行化拥有优秀的支持，程序运行的时候可以通过一个动态 CPU 核掩码来制定要运行的核。DPDK 通过一个环境抽象层（EAL）来隐藏底层的硬件与存储结构，给用户提供一个隐藏底层细节的通用接口。EAL 完成 DPDK 的装载和初始化，CPU 核心绑定，PCI 地址抽象等。

2.2 并行计算相关研究

为了满足应用程序日益增长的规模与计算量，处理器也在不断发展。以前是通过改善制造工艺，增加一块处理器上的晶体管的数目，使处理器的主频得到不断的提升。但很快由于工艺的限制，增加主频已经越来越难，处理器设计逐渐转入多核时代。但许多应用程序还是采用单核架构的串行模式，例如协议栈。在计算规模不断增大的时候，串行程序的性能低下问题日益突出。为了充分利用处理资源，提升程序的性能，许多应用程序的架构都需要重新设计成能充分使用多核处理器的并行框架。

通常情况下，设计良好的并行程序的其执行速度比串行架构的程序更快，加速比是衡量并行程序速度提升规模的评价指标。加速比就是在单个处理器上的执行时间与多个处理器上并行执行时间的比值。

假设处理器只运行测试的应用程序，串行架构的该应用程序在只有一个处理器的主机上执行总时间为 T_S。将该应用程序并行化之后，移植到拥有 P 个处理器的主机上，执行的时间为 T_P。那么这个应用程序在并行化后加速比 S_P 可定义为：

$$S_P = \frac{T_S}{T_P} \tag{2-1}$$

加速比的定义中，有一个基本的假设，多处理器主机与单处理器上的主机的硬件环境，除了处理器的数量不一样，其他的硬件环境必须完全一致。计算机领域里，用来衡量并行程序的加速比的定律主要有 Amadhal 和 Gustafson 定律。

（1）Amdahl 定律[23]

Amdahl 定律是研究并行计算的一个十分重要的定律，它的定义如下：对计算机系统的某个部分更快执行方式获得的加速比，取决于这部分被使用的频率或者执行时间在总的执行时间中所占的比例，也就是说通过不断优化系统中的某个部分，获得的性能提升最终会达到一个上限。

在完全占有处理器的情况下，假设串行化部分时间所占比例为 f，则使用 P 个处理器完成计算所需的时间 T_P 为：

$$T_P = fT_S + (1-f)T_S / P \tag{2-2}$$

由加速比公式可以得出：

$$S_P = \frac{T_S}{fT_S + (1-f)T_S / P} = \frac{P}{1+(P-1)f} \tag{2-3}$$

当处理器的数量 P 趋向于无穷大时，可以推导得到：

$$\lim_{P\to\infty} S_P = \frac{1}{f} \tag{2-4}$$

Amdahl 定律描述了在处理器的数量增加时，加速比并不是线性的递增，并行程序的加速性能有一个峰值，即性能提升的最大幅度取决于串行部分所占比例。因此，通过改善程序设计，尽量减少串行部分的比例可以有效的提升程序的性能。

（2）Gustafson 定律

Gustafson 定律描述了计算机领域里涉及到任意大的数据集可以通过并行化得到有效的性能提升的定律[24]。该定律源于 Sandia 实验室的 John Gustafson 在 1988 具有 1024 个处理器规模的超立方体结构上对三个应用程序进行观察后，发现他们发生了超线性加速现象，这 3 个应用程序在试验中达到的加速比非常接近 1024。这个结果极大的提升了人们对并行设计的信心，它表明了在实际情况中通过增加处理的规模是可以得到线性加速比的，这和 Amdahl 有些相悖。Gustafson 定律和 Amdahl 定律实际上并不互相矛盾，因为这两个定律的假设前提不同。Amdahl 定律假设在整个执行时间里串行部分所占比例是恒定不变的，Gustafson 定律[37]基于两点，第一，串行化代码的规模是固定的，计算规模是随处理器规模增加而增加的。第二，总的执行时间是一个常数。

基于以上两点假设，进行公示推导。假设应用程序在只有一个处理器的主机上运行时间为 T_S，其中串行部分的时间所占比为 f 。在具有多个处理器的主机上总执行时间为 T_P，总执行时间 T_P 由两个部分构成，其中串行串行部分的时间同一个处理器的串行时间为 fT_S ，并行部分的执行时间为 $(1-f)T_S / P$ 。那么总的执行时间 T_P 为 $fT_S + (1-f)T_S / P$。因为假设总的执行时间为一个常数，就假设这个常数为 1。通过代数变换，在一个处理器上运行的时间 T_S 等于 $P + (1-P)fT_S$。根据加速比定义，可以进行如下的公式推导：

$$S_P = \frac{fT_S + (1-f)T_S}{fT_S + (1-f)T_S / P}$$

$$= \frac{P + (1-P)fT_S}{1}$$

$$= P + (1-P)fT_S \qquad (2\text{-}5)$$

由推导的公式可以得到，并行程序的加速比和处理器规模 P 呈线性增长关系。正如前面所说， Gustafson 定律和 Amdahl 定律其实并不相悖，因为他们所基于的假设不一样, 后来有人证明了 Gustafson 定律和 Amdahl 定律的等价性。Gustafson 定律对于并行程序设计具有重大的意义，因为他提升了人们对通过并行化加速程序的信心。

2.3 DPDK

DPDK 是是 Intel 推出的针对数据包快速处理的开发套件。利用 DPDK 可以快速的开发出数据包处理程序，例如针对某具体需求的协议栈。DPDK 有许多不同功能的库组合在一起，每个库负责不同的功能，库与库之间存在依赖关系，依赖关系由箭头所示，核心组件如图 2.3 所示。

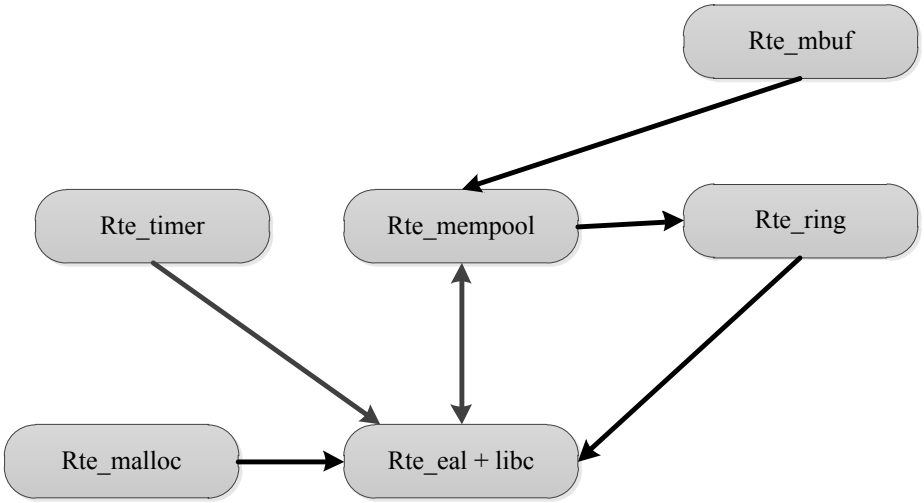

图 2.3 DPDK 核心组件图

(1) Rte_timer 库提供计时相关的功能，依赖于 EAL 库提供的时间接口

(2) Rte_malloc 负责从大内存页收集到的命名空间

(3) Rte_mbuf 提供 IP 数据包的存储缓冲区。

(4) Rte_mempool 库为每一个逻辑核提供存储大量对象的无锁队列，适用于批量的插入删除。

(5) Rte_ring 提供固定大小的无锁队列，允许多个逻辑核之间的通信

(6) Rte_eal 库提供 EAL 虚拟层，负责应用程序的装载，内存分配以及与物理网卡的通信。

DPDK 的这些库使用的主要技术核心是 Hugetlb[25](大内存页)、UIO(用户空间 I/O)、CPU Affinity (处理器亲和性)。

2.3.1 大内存页

操作系统讲内存空间分成许多个大小相等的小存储块，这些块被称为页。进程分配内存的时候，对应的存储块不一定是连续的，同一个进程的存储块也许会被分配到多个不连续的物理内存页。地址转换单元负责物业页面与虚拟内存地址的相互映射转换。这种存储管理被称为分页式存储。一般存储块的大小被设置为 4096 个字节，这也是目前 Linux 操作系统的默认页面大小。4KB 的页面大小已经持续了多年，近些年来随着操作系统与应用程序的复杂度与规模越来越高，对内存的需求越来越大。而且伴随着 64 位系统与分布式架构的普及，服务器的硬件已经从 4GB 提升到几十 GB，4KB 的小页面已经不适合大内存。因为如果采用 4KB 的页面，大内存对应的页表会过于庞大，每次进行地址转换时，查找页表的开销也过大，同时页表状态的更新也会过于频繁。其次由于服务器上的应用程序的内存需求量较大，较大的内存页面可以减小发生缺页中断的频率，提高应用程序的运行速度。

Linux 操作系统支持大内存文件系统 Hugetlbfs, Hugetlbfs 使用 2M 大小的页面，可以有效的减少高速缓存缺失与页表查找开销大的问题。DPDK 的内存池分配在大内存文件系统里，通过在内存池中分配用于存储数据包的内存，可以有效的利用大内存页的特点，提高数据包的处理速度，减少在高速网络下数据包的处理开销。

2.3.2 用户空间 I/O

用户空间 I/O 就是使用户态的应用程序与硬件驱动设备之间形成数据通路，应用程序可以直接复制拷贝驱动设备缓冲区的数据。通过用户空间 I/O，避免了数据从驱动设备缓冲区到内核缓冲区的应用程序缓冲区的多次拷贝，有效的提升了效率。一般用户空间 I/O 是通过内存映射的方式，将驱动设备的空间暴露给应用程序，使应用程序可以直接访问。

一般数据包到达网卡后，是通过异步中断的方式通知操作系统。然后操作系统将数据包拷贝到内核缓冲区，然后再到达应用程序缓冲区，期间伴随着系统调用与昂贵的内核态和用户态的上下文切换。DPDK 提供基于轮询的用户空间 I/O，通过不断的轮询网卡缓冲区，应用程序可以第一时间从网卡获得数据包，并复制到用户空间进行处理，极大的增大了数据包的吞吐率。

2.3.3 处理器亲和性

处理器亲和性：进程会被指定分配到一个固定的处理器上，不会频繁的在多个处理器间迁移，导致处理器的高速缓存频繁失效。亲和性分为软亲和和硬亲和。软亲和性是指给进程给出主要运行的处理器建议，如果指定的处理器有其他任务，才选择备用处理器。而硬亲和性则是必须在指定的处理器上进行运行，如果处理器繁忙，则阻塞等待。

处理器亲和性是指进程倾向于尽量的运行在指定的某个处理器上。Linux 自身的进程调度器在调度进程的时候本来就有软亲和性的特点，这表明着进程不会频繁的轮流被多个处理器执行。因为进程之间发生迁移的频率小，那么所产生的上下文切换开销就比较小。

处理器的硬亲和性可以通过编写代码指定某个处理器来运行某个进程来实现。这表明在要运行应用程序的 CPU 可以通过编程来显示的制定。使用硬亲和性的 3 个原因。

1. 在计算密集的应用中，例如图像数据处理、天气信息预报等。计算密集的应用程序通过处理器的硬亲和性绑定到某个特别的处理器上，可以有效的利用缓存，减小处理器迁移开销，提升计算速度。

2. 研究内核的亲和性技术的目的是测试复杂软件。对于一个可以动态扩展的应用程序，也就是当运行的主机分配的处理器数量更多时，可以动态开辟更多的线程来运行，以此获得更快的执行速度。如果随着处理器个数的增加，应用程序的运行速度也线性的提升，那么处理器每秒处理的事务个数和处理器的数量满足一种线性的关系。以这样的假设来建立数学模型，可以判断应用程序是否充分的利用了所有的处理器资源。Amdahl 法则表明了这种假设的加速比太过理想化，由于应用程序运行的时候，不仅仅和处理器的数量有关系，还和其他因素息息相关。比如高速缓存，内存数量等。因此实际实验的数据不会十分贴合这种假设，但是通过调整其他的因素，拟合的数据曲线可以无限的接近这样一个值。在实际情况下，可以假设，应用程序是有一系列的硬件来决定运行速度。在应用程序的复杂度不断的扩大时候，在不断的改善应用程序的运行速度的时候，这些硬件组合最终会形成一个应用程序的运行时间复杂度的上限。处理器较高的缓存命中率对于应用程序的性能至关重要。某个特定的应用程序如果在处理器之间频繁的迁

移，那么它就失去了利用处理器缓存带来的性能提升。为了保持缓存与内存数据的一致性，对于多个处理器而言，假如某个处理器缓存了一些数据，为了避免其他的处理器也缓存同样的数据，导致最后数据写回内存时候，数据的的不一致，其他的处理器就会抛弃这些缓存数据。对于要使用相同的数据的多个线程，在同一个处理器上运行这些线程，可以十分有效的提高缓存的命中率，使性能得到明显的提升，因此应该把这些线程绑定到同一个处理器上。

3. 研究处理器亲和性的最后一个目的是运行实施性的，时间敏感的程序。针对时间敏感的应用程序，要确保该应用程序第一时间能够响应请求。可以让该应用程序独占某个处理器，用剩余的处理器轮流的调度运行其他的应用程序。这样可以保证时间敏感的程序一直拥有处理器资源，保证它的快速运行，同时其他的应用程序也能够在剩余的处理器上得到调度执行。

2.4 传统 VPN 实现方案

为了满足各个 IDC 数据中心之间的大数据的安全传输，常常需要使用 VPN。VPN 是虚拟专用网的简称，VPN 通常运行在公共的英特网上或者其他的公共网络，为了保证数据的安全，常常建立一个逻辑的隧道，使用加密的方式来保证数据安全，使用户感觉就像在一个专线网络中一样。

VPN 的使用已经十分普遍，在各大企业和教育院校中得到了充分的应用。VPN 的实现技术主要有 PPTP/L2TP 和 IPSec 两种[26]。

1. PPTP/L2TP

PPTP 和 L2TP 是一个早期的 VPN 协议，位于链路层。PPTP 和 L2TP 基于 PPP 协议封装数据包。PPTP 协议只能在两个端点之间建立单条隧道， L2TP 可以在两端点间建立多条不同质量的隧道。L2TP 协议支持在隧道模式下进行验证，但 PPTP 协议不支持。

PPTP 和 L2TP 使用配置起来十分简单，但他们的缺点之一是对于扩展性没有良好的支持。PPTP 和 L2TP 的使用场景是客户端远程访问 VPN 网络，但由于内在安全机制的缺失，对应一些安全敏感的企业，如金融机构和电商企业，不适合使用 PPTP/L2TP[27]。

2. IPSec:

IPSec 是因特网互联工程组制定的网络层 VPN 协议，它为网络层传输的数据包提供了身份认证与加密保护[28]。IPSec 协议一整套有关网络层数据包的认证、加密的安全协议体系，包括因特网密钥交换协议 IKE(Internet Key Exchange)、封装安全载荷协议 ESP(Encapsulating Security Payload)、认证头协议 AH(Authentication Header)以及相关的一整套摘要认证算法和安全加密算法。AH

协议和 ESP 负责提供身份认证、完整性检查和加密功能，IKE 协议负责密钥的自动协商交换。IPSec 是目前使用最广泛的 VPN 协议[29]。

IPSec 提供三种安全机制:认证、完整性检查和加密。他们的定义如下所示.

(1) 认证：用于对未知的通信方身份进行核查鉴定。

(2) 完整性检查：检查数据包是否被篡改过，确保数据包的完整性。

(3) 加密：对明文数据包进行加密，防止他人窥探获取。

AH 协议的全称是 Authentication Header Protocol。IP 类型字段号是 51，AH 协议能够提供数据完整性认证、对端身份鉴别、抗重播保护功能，AH 协议不具有加密功能，所以它传播的信息都是以明文暴露，不适合用来传输敏感数据。AH 的认证是通过一个消息摘要字段，这个字段是整个报文中绝大部分字段的摘要值，通常使用的摘要算法有 HMAC-SHA1 和 HMAC-MD5 等。

ESP 协议是封装安全载荷协议 Encapsulating Security Payload Protocol 的全称。IP 类型字段号是 50。ESP 协议具有对端身份界别、部分数据完整性认证、防重播和加密功能。ESP 协议在要保护的 IP 数据载荷前后分别加上 ESP 帧头和 ESP 尾部。ESP 与 AH 的区别在于，ESP 提供部分认证功能，也就是 ESP 只对 IP 数据包的载荷部分进行认证，并不对整个数据包进行认证。同时 ESP 对 IP 载荷部分提供加密保护，保证数据的安全传输。提供安全性的加密算法通常是对称加密算法，例如 AES、DES、3DES 等。提供部分数据完整性的摘要算法有 HMAC-SHA1 和 HMAC-MD5 等。

在使用 IPSec 的时候，可以根据具体的使用场景，选择合适的协议。可以只是用 AH 或者 ESP 中的一种，也可以两者结合起来使用。虽然 AH 和 ESP 都提供数据完整性保护，但 ESP 协议只提供部分数据字段的完整性保护，而 AH 协议却能对整个数据包进行保护。但 ESP 协议可以提供安全加密服务，因此两者结合使用的方法是：先使用 ESP 协议对载荷部分进行加密，然后使用 AH 协议对加密后的整个数据包进行封装，通过两者结合，密文传播与整个数据包的完整性保护都实现了。

构建 SA 的方法有手动配置和 IKE 自动协商两种，其中使用 IKE 自动协商是一种在企业中使用最为普遍的 SA 配置方式。ISAKMP（Internet Security Association and Key Management Protocol，互联网安全联盟和密钥管理协议）定义了协商、建立、修改和删除 SA 的过程和报文格式。ISAKMP 只是为建立、更新、删除 SA 以及他们的属性提供了一个标准规范，但没有明确定义整个过程中 SA 的具体格式。ISAKMP 是 IKE 的基础。IKE 协议负责对 AH 协议和 ESP 协议中所需要密钥进行自动协商、建立对等体之间的 SA，简化手动配置 IPSec SA 的工作量，提升了安全性。

IKE 自动协商的密钥的方式通过一系列数据的交换，双方根据交换的数据计算出共享的密钥，传输的数据即使被第三方捕获，也无法计算出共享秘钥。

OpenSwan 是 IPSec 成熟的开源解决方案之一。OpenSwan 是开源项目 FreeS/WAN 的后继分支，OpenSWAN 完整的在 Linux 下实现了 IPSec，OpenSwAN 的配置较为简单，OpenSwan 可以运行在 Linux 2.0 以上内核，OpenSwan 也支持不同的硬件平台，例如 X86、X86_64、ARM 等。OpenSwan 的主要由三个模块组成：内核模块（KLIPS/26sec）、密钥管理模块（pluto）、配置模块（IPSec 命令配置脚本）[30]。

IDC 之间的数据传输面向的是数据包，而不是连接，对上层的业务完全透明。所以选择使用 IPSec VPN 的解决方案。但正如之前所说，IPSec 是一个三层协议，对协议栈高度依赖，为了满足 IDC 之间的大数据传输。使用设计实现一个用户态的 IPSec 就十分必要。诸如之前所说 Intel DPDK 在开发用户态协议栈方面具有十分便利，所以本文选择使用基于 DPDK 的用户态协议栈解决方案。

2.5 本章小结

本章研究了传统协议栈面临的问题、目前主流的协议栈加速方案以及 DPDK。首先叙述了协议栈的架构以及造成高速网络下传统协议栈瓶颈的因素，接下来介绍了目前主流的三种协议栈加速方案。然后简要的介绍了并行计算的相关理论与 DPDK 在解决传统协议栈的问题的关键技术，最后叙述了目前 VPN 的常见实现方法。

第三章 用户态 IPSec 协议栈设计方案

DPDK 是 Intel 推出的针对数据包快速处理的开发套件。利用 DPDK 可以快速的开发出数据包处理程序，例如针对某具体需求的协议栈。本章主要讲述基于 DPDK 开发用户态的 IPSec 协议栈的解决方案。

3.1 框架设计

IDC 之间的数据传输面向的是所有的数据包，而不是某个具体业务的某条连接。IPSec 是一个三层的 VPN 协议，它为上层业务提供透明的安全传输。由于 IPSec 工作在网络层，与协议栈高度耦合。传统协议栈在高速网络中面临中断频繁、冗余拷贝、不支持多核框架等问题，为了保证各个 IDC 之间的数据流在经过 IPSec VPN 的处理之后，仍能保持较高的收发速率，设计的用户态协议栈协框架如图 3.1 所示。

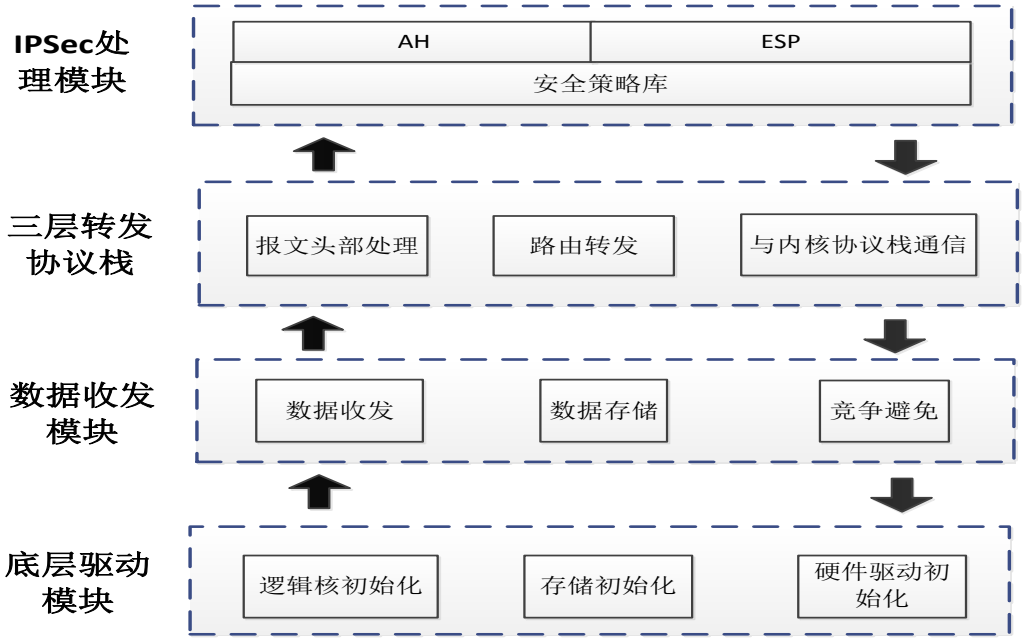

图 3.1 用户态 IPSec 协议栈框架图

3.2 底层驱动模块

底层驱动模块位于整个协议栈的最底层，是为了屏蔽硬件的差异，充分利用硬件资源。Linux 协议栈优良的移植性与稳定性被广泛的使用在各个平台与硬件架构上，但 Linux 传统协议栈对硬件的一个极大的缺陷是不支持多核框架。协议栈无法实现负载均衡，协议栈任务几乎全部压在第一个逻辑核上。其次，由于 IDC 的出口网关处多是高性能的服务器，他们都是分布式存储设计，而传统协议

栈对分布式存储的支持也十分欠佳。因此，用户态协议栈首先要解决的是以下两点:

1. 提供对多核的支持, 并且在不同的多核硬件上也可以动态的支持不同数量的处理器核。

2. 提供对分布式存储器的支持。

3.2.1 多核初始化模块

在多核环境下，我们将所有的逻辑核分为了主核和次核。一般主核是指应用程序开始运行时所在的逻辑核，次核是指剩余的核。主核的任务有初始化次核、内存池的初始化、共享变量的建立、硬件的检测以及任务的分发等。次核负责执行主核分发的任务。

为了避免分发的任务在多个逻辑核之间频繁的迁移，通过 CPU 的亲和性将任务固定分发到某一个次核。次核的任务执行由主核完全控制，不能被其他应用程序抢占，因此在整个应用程序的执行期间，处理器核都是被独占。由于往往服务器上不止一个应用程序在运行，为了保证服务器的正常运转，可以动态的制定要运行的逻辑核。以下是运行应用程序的一个示例：

```
./app -c COREMASK
```

通过一个参数 COREMASK 来指定要运行的逻辑核，COREMASK 是十六进制的 bit 掩码，用来表示要运行的 CPU 逻辑核编号。要运行的核对应的 bit 是 1，0 号核对应的 bit 位在最右侧。在 COREMASK 指定的逻辑核中，指定编号最低的为主核，其余的为次核。由于次核执行任务是由主核进行分发，为次核指定了三个状态：等待、运行和完成。相应的枚举定义如下：

```
enum    rte_lcore_state_t {WAIT, RUNNING, FINISHED}
```

- WAIT:是当前该逻辑核没有任何任务去执行，等待分发任务。
- RUNNING:是指逻辑核正在执行任务。
- FINISHED:是指逻辑核已经执行完成任务，但状态主核未获知

如图 3.2 所示，展示了三个状态之间的逻辑关系。

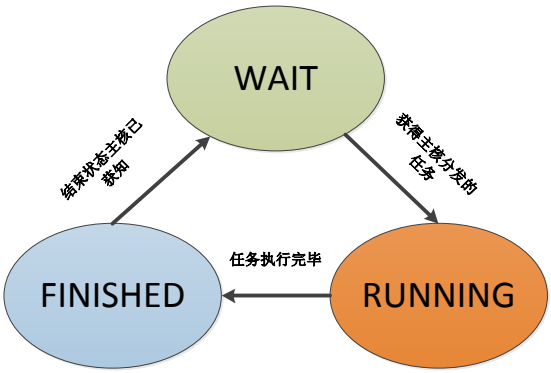

图 3.2 逻辑核状态转换图

主核的执行序列，可以用以下的伪码序列来表示。

```
params_check();                    //检查用户的参数
dev_init();                //网卡设备等初始化
Memory_init();                     //分配并初始化内存
pthread_create_for_every_slave_lcore();      //为每一个逻辑核分配一个线程
wait_all_threads();
//等待每一个逻辑核的线程初始化结束，并且他们的状态都置位了WAIT
work_lunch();                      //为某一个次逻辑核分配任务
wait_over();                       //等待此逻辑核的状态变为FININSHED
resouce_free();                    //回收资源
exit();                            //退出
```

次核的执行序列，用以下的伪码来表示。

```
pthread_init();                    //初始化线程执行环境
LOOP
{
    set_state_wait();              //将状态置为wait
    wait_for_job();                //等待任务
    get_job();                     //获得了任务
    set_state_running();           //将状态置为running
    run_job();                     //执行任务
    set_state_finished();          //将状态置为finished
    wait_master_lcore();           //等待主核获取状态
}
```

3.2.2 分布式存储分配模块

传统的多核处理器是使用 SMP 模式，SMP 模式即对称多处理器模式，是若干个相互独立的处理器共同访问同一个物理内存。因为每个处理器访问到的内存数据都是一样的，因此 SMP 架构也被称为 UMA(Uniform Memory Access Architecture)结构体系，UMA 即一致性内存放访问架构。SMP 架构的缺点内存的扩展性较差，因为所有的处理器通过同一个总线来访问，增加内存与处理器，提升的性能是有上限的[31]。

IDC 的网关都是一些高性能的服务器(比如 Dell PowerEdge 620)，这些服务器往往拥有多个处理器，每个处理器拥有 10 多个核。这些服务器往往都是 NUMA(Non Uniform Memory Access Architecture)架构[32]。NUMA 即非一致内存访问架构，是分布式存储器的一种架构。存储器和处理器被划分为多个单元。每

个单元里的处理器是类似 SMP 架构，访问自己的本地存储器。NUMA 架构适合在存储器空间较大、处理器数量较多的服务器。相比同等处理器的 SMP 架构，NUMA 架构能够获得更高的性能与出色的经济性。NUMA 架构为所有的处理器提供同样的内存访问速度，而 NUMA 架构为本单元的处理器提供高速的内存访问，为其他单元的存储器提供经济但较慢的存储访问。以下是 NUMA 架构下的存储结构如图 3.3 所示。

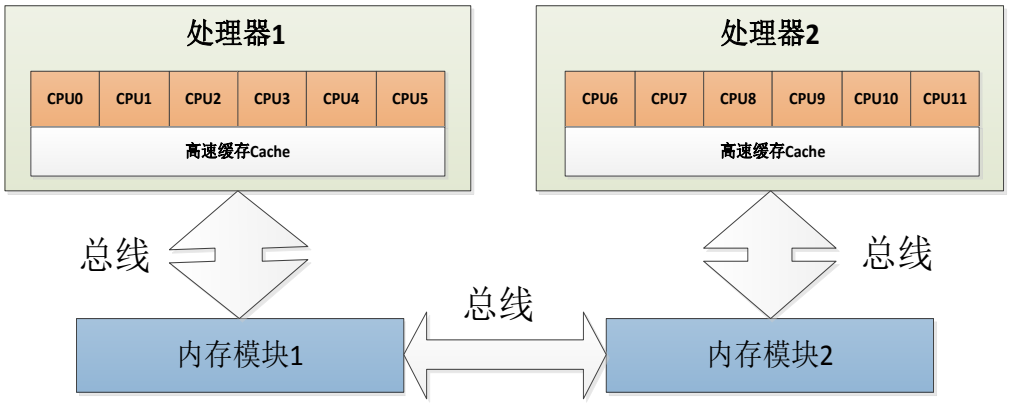

图 3.3 NUMA 架构简图

处理器逻辑核访问存储的速度依次是:本地 cache、本地内存、其他处理器内存。因此为了保证快速的访问存储在内存中的数据包，加快数据处理，在分布式存储的时候，需要为每一个处理器都分配本地内存，分配序列如以下伪码所示。

```
For var lcore_id = first_lcore_id To last_lcore_id
//从第一个逻辑核到最后一个逻辑核进行循环遍历
{
    //如果该逻辑核不是用户指定要运行的逻辑核，那么进行下一轮迭代
    IF(lcore_id is not set in COREMASK)
            CONTINUE;
    process_id = PROCESS_CPU(lcore_id);          //逻辑核属于 process_id
    //如果处理器 process_id 还没有分配存储，那么就给它分配
    IF(process_id hasn't memory)
            allocate_memory();
}
```

3.3 数据收发模块

3.3.1 收发方式

在传统协议栈中，网卡接收到数据包后，通过硬件中断的方式通知操作系统内核。在高速网络环境下，中断的开销是相当大的，而且由于 Linux 将中断都交

给第一个核来处理。所以即使在多核环境下，严重的负载不均衡，第一个核往往满负荷运转，成了数据处理的瓶颈。

数据收发模块的任务是以最快的速度收取和发送数据包，并且能够实现负载均衡和很低的占用率。为了第一时间获得数据包，通过轮询的模式从网卡获取数据包。每一个逻辑核可以分配一个发送和接收队列，将收到的数据包平均放在网卡的接收队列中，以此实现负载均衡。另外不断的检查每一个逻辑核的发送队列，将要发送的数据包发送出去。

多核的轮询模式有两种方案:独立模式与流水线模式，两种模式各有优缺点。

独立处理模式:每个逻辑核独自接收数据包，收到数据包后，对数据包进行处理，处理完毕后将数据包发送出去。各个逻辑核之间解耦合，相互独立。

流水线模式:逻辑核共同完成处理整个数据包的流程，一部分逻辑核负责接收数据包，一部分逻辑核负责处理和发送数据包。

独立模式的执行流程如下伪码所示:

```
Loop
{
    receive_packet();          //接收数据包
    process_packet();          //处理数据包
    send_packet();             //发送数据包
}
```

独立模式下的架构图如图 3.4 所示。

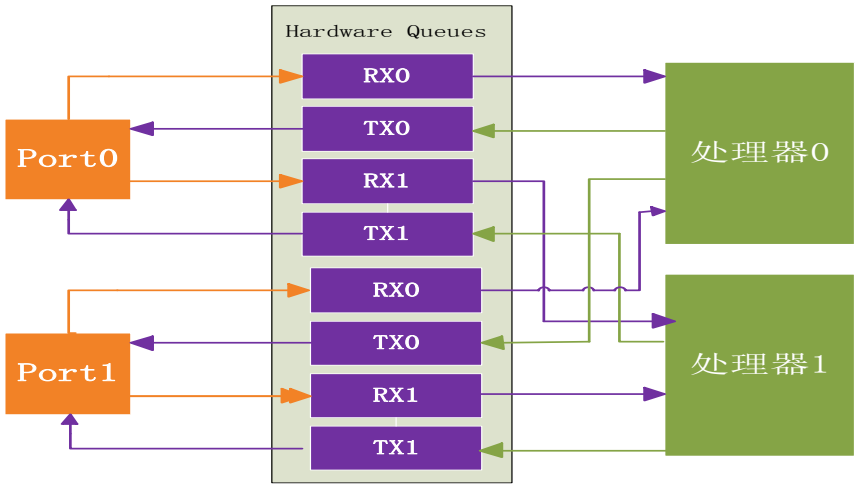

图 3.4 独立模式架构图

流水线模式下的执行流程的伪代码如下所示：

```
IF(self_lcore is receive_lcore)                    //如果当前逻辑核是接收核
{
    LOOP
    {
        Receive_packet();                          //接收数据包
```

```
            Send_to_process_packet();                    //发送到处理逻辑核
    }
}
ELSE IF(self_lcore is process_lcore)
{
    LOOP
    {
            Receive_packet_from_other_lcore();    //从别的逻辑核接收数据包
            Process_packet();                     //处理数据包
            Send_packet();                        //将数据包发送出去
    }
}
```

流水线模式下的架构图如图 3.5 所示。

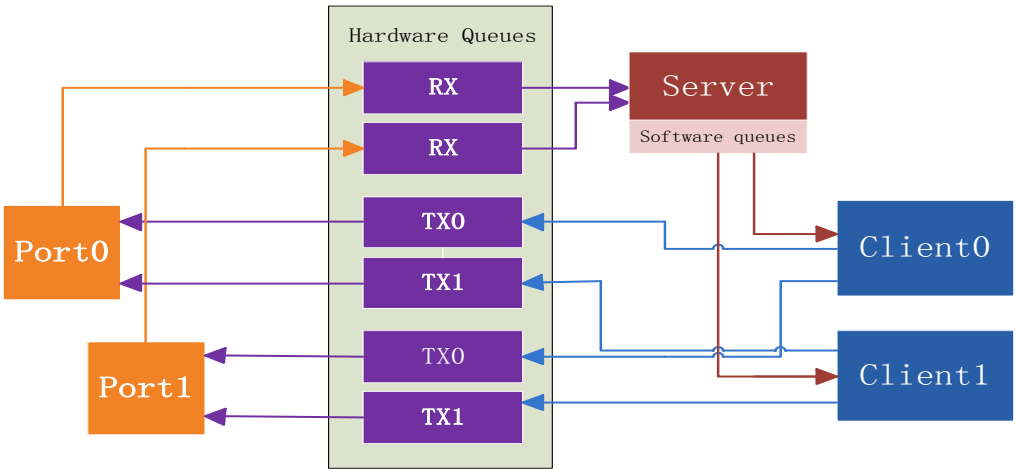

图 3.5 流水线模式架构图

两种模式各有优缺点，独立模式面向的是数据包，即使某个逻辑核崩溃仍能正常的首发数据。而流水线模式适合面向流的数据处理，接收数据包的逻辑核根据五元组(源端口，目的端口，源地址，目的地址，协议)来对数据包进行分流发送到某个特定的处理逻辑核上，一个处理处理逻辑核只负责某个流的处理，这样能保证数据包的有序到达处理逻辑核，但如果收取逻辑核发生了阻塞，则会造成收发中断。但由于 IDC 的出口网关面向的数据包，不用进行分流，而且多个核同时收取，保证稳定运转。因此使用独立模式。

3.3.2 数据存储

传统的 Linux 内核协议栈使用 *sk_buff* 来存储数据包信息。*sk_buff* 在一个单独的内存区域存储嵌入的元数据结构体，后面紧跟固定大小的内存区域用来存储报文数据，结合 *sk_buff*，定义元数据如下所示。

```
struct rte_pktmbuf
{
    struct rte_pktmbuf *next;          //下一个分节的地址
    void *data;                        //指向本分节数据存储区的地址
    uint16_t  data_len;                //本分节存储的数据
    uint8_t  nb_segs;                  //包含的分解数
    uint8_t  in_port;                  //收到的物理网卡端口号
    uint32_t  pkt_len;                 //整个数据包的长度
}
```

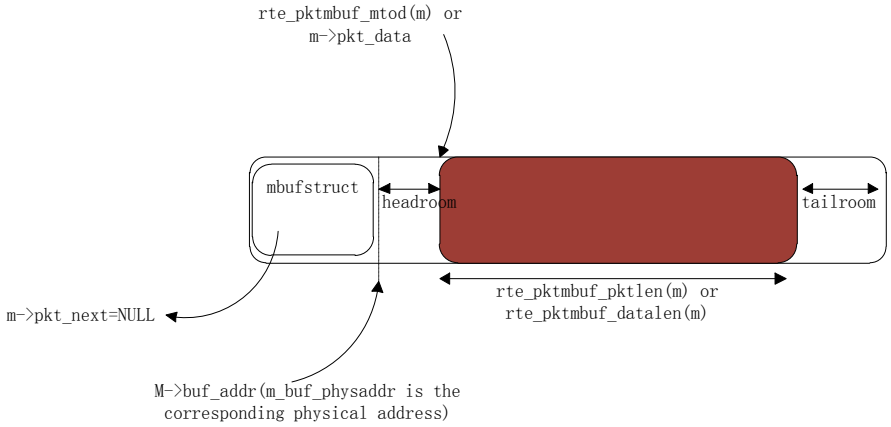

图 3.6 单分节数据包图

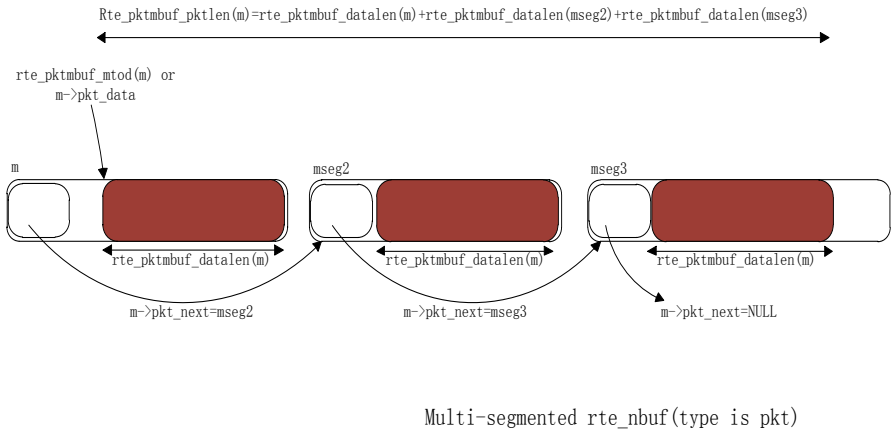

图 3.7 多分节数据包图

第一个分节的元数据结构是整个数据包的控制信息，包含本分节的长度 *date_len*，也包括所有分节的长度和 *pkt_len*。对于尺寸较大的一个数据包，需要多个分节，通过将所有的分节连接起来存储数据包。如图 3.6 和图 3.7 所示，分别展示了只有单分节和多分节的数据包的存储模式。

3.3.3 避免竞争

多核环境下，为了保证多核对共享数据访问的一致性，常常通过对共享数据进行加锁。但加锁和解锁的开销很昂贵，而且由于每次只能有一个线程访问共享数据，造成并行的多核线程变成等待共享资源的串行模式。因此，设计一种高效的共享数据存储和访问方式对于并行协议栈十分必要。

1. 无锁队列

并行协议栈中竞争十分普遍，这些竞争总结起来就是一个生产消费者模式。例如数据包收发的独立模式中，多个逻辑核同时从网卡驱动接收缓冲区收取数据包是一个多消费者模式。多个逻辑核同时将数据包发送到网卡驱动发送缓冲区就是一个多生产者模式。流水线模式中，接受逻辑核发送给处理逻辑核数据包属于单生产者多消费者模式。这些生产者和消费者模式都需要往公共的缓冲区中插入和取走数据包。在传统的生产消费者模型中，通过加锁的方式来解决竞争。但高速网络下，采用加锁的办法的开销就非常大，并行处理就会串行化，性能会极具降低。因此需要使用无锁队列，来解决竞争问题[33]。

为了满足高速网络，无锁队列需要满足以下的条件。

(1) 无锁化操作

(2) 提供多消费者或者单消费者模式的出队列

(3) 提供多生产者或者单生产者的入队列

(4) 批量入队列

(5) 批量出队列

以下是 DPDK 中大小固定基于数组的一个无锁队列的结构体的的定义，其中包含生产者和消费者的结构，并且生产者和消费者都有一个队头队尾索引。

```
struct rte_ring
{
    struct  cons;                               //消费者结构体
    struct  prod;                               //生产者结构体
    char     name[RTE_RING_NAMESIZE];  //无锁队列的名字
    void *ring[0] __rte_cache_aligned;        //指向后面的缓冲区
}
struct rte_ring::cons
{
    uint32_t        sc_dequeue;             //是否是单消费者
    uint32_t        size;                   //队列的长度
    uint32_t        mask;                   //运算掩码，通常为 size - 1
```

```
    volatile uint32_t head;                    //生产者队列头对应的索引
    volatile uint32_t tail;                    //生产者队列尾对应的索引
}
struct rte_ring::prod
{
    uint32_t        sp_enqueue;                //单生产者
    uint32_t        size;                      //队列的长度
    uint32_t        mask;                      //队列掩码，通常为size - 1
    volatile uint32_t head;                    //消费者队列头对应的索引
    volatile uint32_t tail;                    //消费者队列尾对应的索引
}
```

无锁队列的结构体图如图 3.8 下所示, *prod_head* 和 *prod_tail* 分别是生产者的对头和队尾，*cons_head* 和 *cons_tail* 指向消费者的队尾。

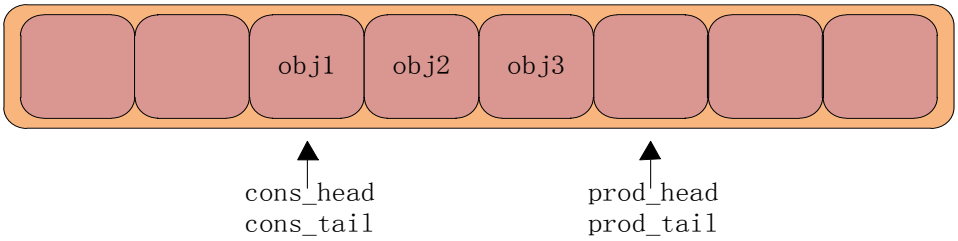

图 3.8 无锁队列结构图

多生产者同时向队列中插入对象的过程如下所述，多消费者模式类似，初始状态下 *prod_head* 和 *prod_tail* 都指向同一个位置。

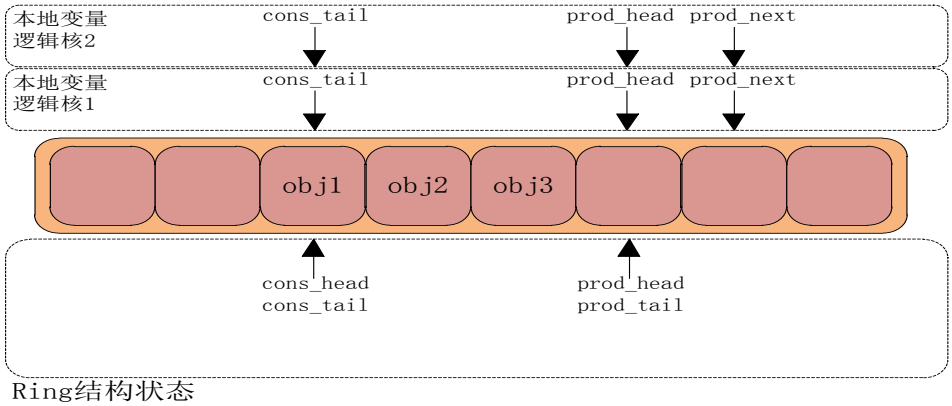

图 3.9 多生产者插入步骤 1 状态

(1) 如下图所示，在执行插入队列操作的逻辑核上，都复制一份无锁队列的 prod_head 和 cons_tail 到局部变量中。局部变量 prod_next 指向表中的下一个可用位置，如果是一次执行 n 个对象的批量插入时，就指向在第 n 个可用位置。如果可用空间不够的话，就会返回一个错误。执行步骤 1 后状态如图 3.9 所示。

(2) 修改 *prod_head* 指针，当 *prod_head* 指针等于局部变量的 *prod_head*，它被设置为局部变量的 *prod_next*，原子操作 CAS 执行成功，继续第三步的操作，否则重新执行第一步的。假设逻辑核 1 成功执行，逻辑核 2 执行失败结果状态如图 3.10 所示。

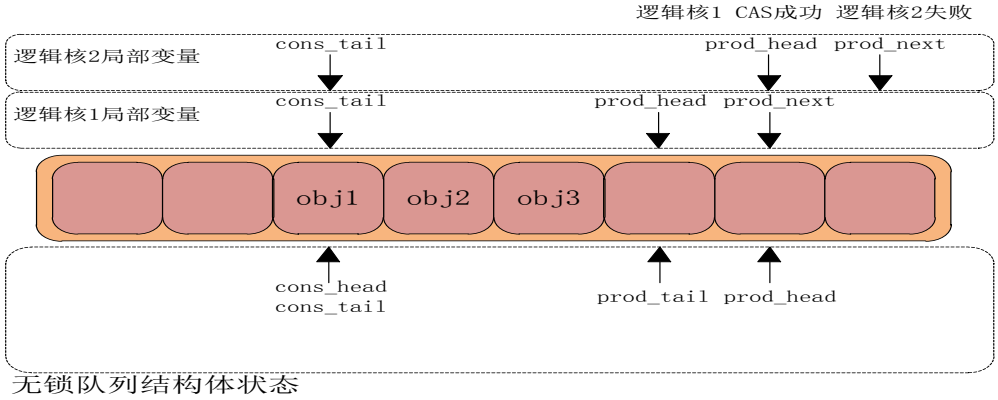

图 3.10 多生产者插入步骤 2 状态

(3) 每一个核更新要插入的对象，将对象放在 prod_head 所指向的位置，例如逻辑核 1 插入 obj 4，逻辑核 2 插入 obj 5，图示如图 3.11 所示。

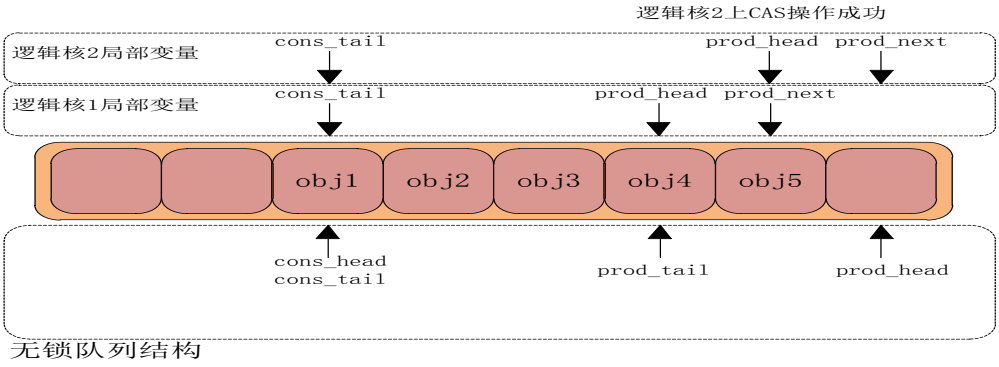

图 3.11 多生产者插入步骤 3 状态

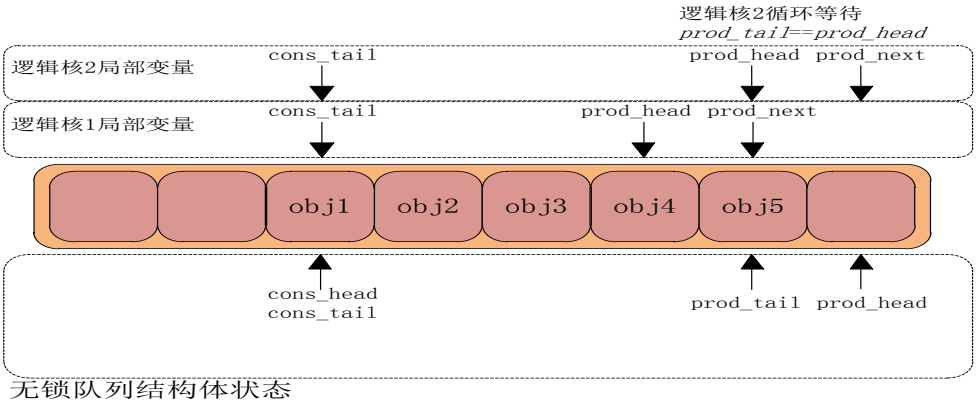

图 3.12 多生产者插入步骤 4 状态

(4) 当 prod_tail 等于局部变量 prod_head 更新 prod_tail 为 prod_next，逻辑核 1 更新成功，图示如图 3.12 所示。

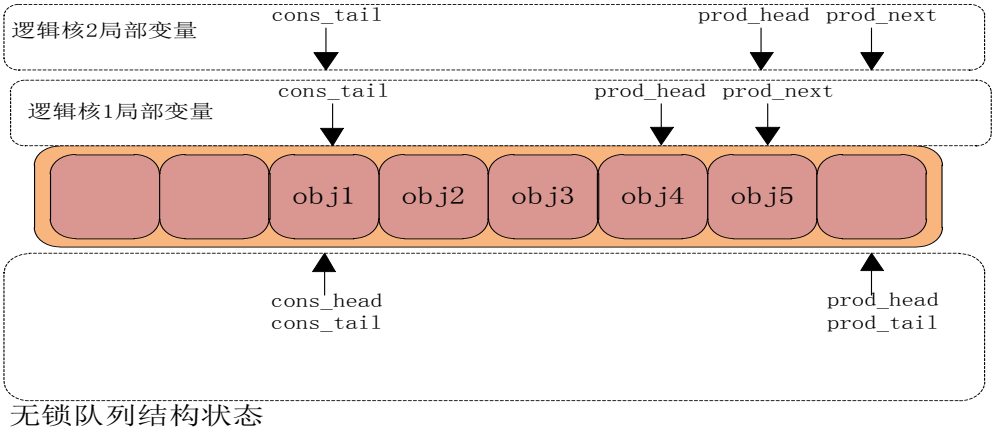

图 3.13 多生产者插入最终状态

(5) 当逻辑核 1 更新了 *prod_tail* 的时候，逻辑核 2 就可以更新 *prod_tail* 了(此时 *prod_tail* 等于 *prod_head*)，多生产者的整个插入操作结束，最终状态图如图 3.13 所示。

传统的队列往往用连接实现，由多生产者插入的整个步骤得知，无锁队列具有以下的特点。

(1) 速度更快。无锁队列的插入和删除只需要一次比较和交换指令，相比而言，传统连接队列需要两次比较和交换指令。

(2) 相比一个完全无锁化的队列更加简单

(3) 更适应批量的插入删除操作。因为对象的指针都是存在一个表中的，一次进行多个对象的出队列和入队列相比传统相连链表缓存命中率更高。

二. 内存通道均匀分配

同时为了避免动态分配造成的开销，通过缓存池技术来提高分配效率。缓存池即提前在内存中开辟的一块空间，同时保持内存对齐以使对象都均匀的分布在所有的 DRAM 或者 DDR3 通道上。这对于三层转发协议栈来说特别有用，因为只用读取包头的 64 个字节，当每个对象的开始地址平均分配在每个通道上时，因为多个通道可以同时输出数据，性能会获得巨大的提升。两通道四个内存列的双列直插式存储模块(DIMM)读取数据包如图 3.14 所示。

3. 逻辑核缓存

多个逻辑和同时访问内存池对列的代价可能会比较高，因为每个核的访问都需要一次 CAS 原子操作，原子操作也是短暂的锁定。为了避免同一时刻，过多的内存访问需求，内存池在分配的时候可以为每一个逻辑核保留一块缓存。在批量访问内存池的时候，通过缓存来访问内存池，可以大大减少内存池锁定的次数。每一个核完全独立的访问他们的缓存，当缓存满的时候，内核才会把一些对象重新冲洗到内存池中，比如使用 LRU 算法。当缓存空的时候，缓存一次可以从内

存池中获取更多的对象。图 3.15 展示了两个内核使用缓存对一个内存池进行访问的情况。

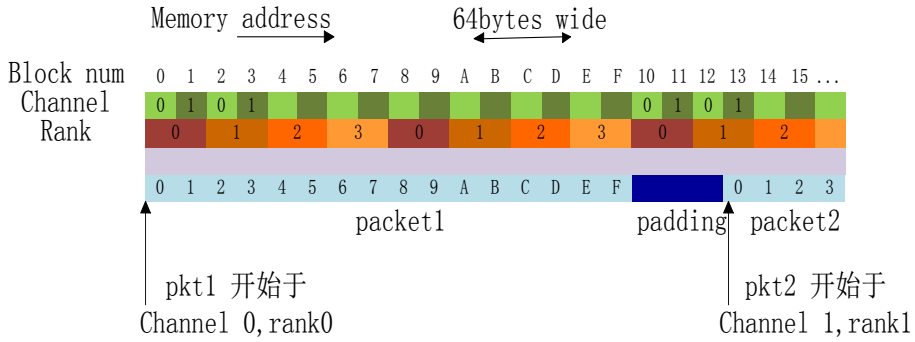

图 3.14 DIMM 内存数据包分布

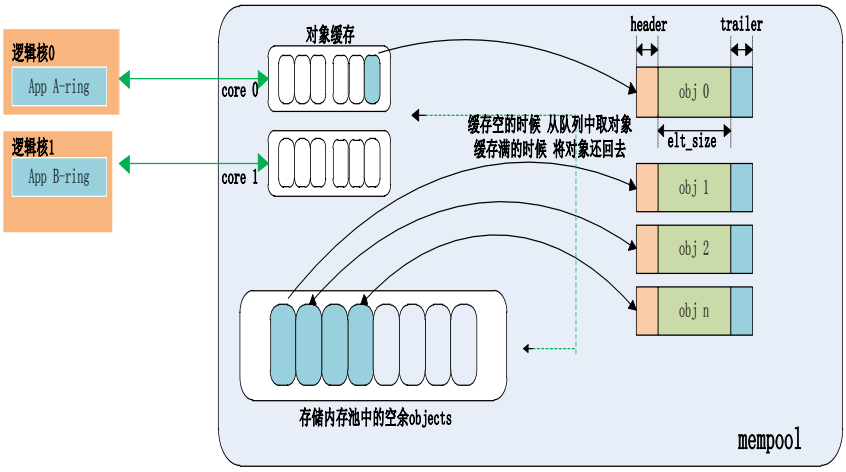

图 3.15 缓存访问内存池

3.4 三层转发协议栈

因 IPSec 协议是一个三层的 VPN 协议，工作在网络层，为了实现用户态 IPSec 协议栈，首先需要完成一个三层转发用户态协议栈， 该协议栈需要具有以下功能。

(1) 链路层、网络层的数据包头的添加和删除，能够进行校验，填充相关字段

(2) 配置路由表，对数据包进行路由转发。

(3) 能与 Linux 内核传统协议栈进行通信。

其中第一个功能，只用在对数据包进行处理的过程中根据链路层和网络层的包头格式，添加上处理包头流程即可，重点讲述后两个功能。

3.4.1 路由转发

传统协议栈中，通常使用 OSPF 等动态路由算法，为了简化工作量，使用静态路由来手动添加。以下是程序中定义的路由结构体。

```
struct ipv4_l3fwd_route {
    uint32_t ip;                //网络地址
    uint8_t  depth;             //掩码深度
    uint8_t  if_out;            //出口网卡编号
};
```

通过为路由表添加、删除路由进行路由表的配置，在进行路由转发时，通过路由查找得知出口号。以下是相关的函数。

```
//路由添加函数
int rte_lpm_add(
struct rte_lpm *      lpm,          //路由表指针
uint32_t      ip,                   //网络地址
uint8_t       depth,                //子网掩码深度
uint8_t       next_hop              //到下一跳的出口
);
//路由删除函数
int rte_lpm_delete(
struct rte_lpm *      lpm,          //路由表指针
uint32_t      ip,                   //网络地址
uint8_t       depth                 //子网掩码深度
)
//路由查找函数
static int rte_lpm_lookup(
struct rte_lpm        * lpm,        //路由表指针
uint32_t      ip,                   //IP 地址
uint8_t *     next_hop              //到下一跳的出口
);
```

3.4.2 用户态协议栈与传统协议栈通信

传统协议栈十分复杂，但功能完备。用户态协议栈主要针对的是和业务相关的协议，其他协议的数据包可以交给内核协议栈来处理，比如将 ARP 和 ICMP 协议数据包交给 Linux 内核协议栈处理，这样可以精简用户态协议栈的结构，更加的专注业务逻辑，减少重复开发，并降低开发难度。

基于 DPDK 的用户态协议栈与传统协议栈之间通过 KNI 接口进行通信，使用 KNI 接口进行通信的有以下优点。

(1) 通过减少系统调用和 *copy_to_user* 和 *copy_from_user* 操作，获得比传统的 Linux TUN/TAP 虚拟网卡更快的速度。

(2) 允许通过标准的 Linux 网络工具来对使用 DPDK 的端口进行管理，例如 *ethtool*, *ifconfig* 和 *tcpdump*。

用户态应用程序使用 KNI 接口与传统协议栈的示例如图 3.16 所示。

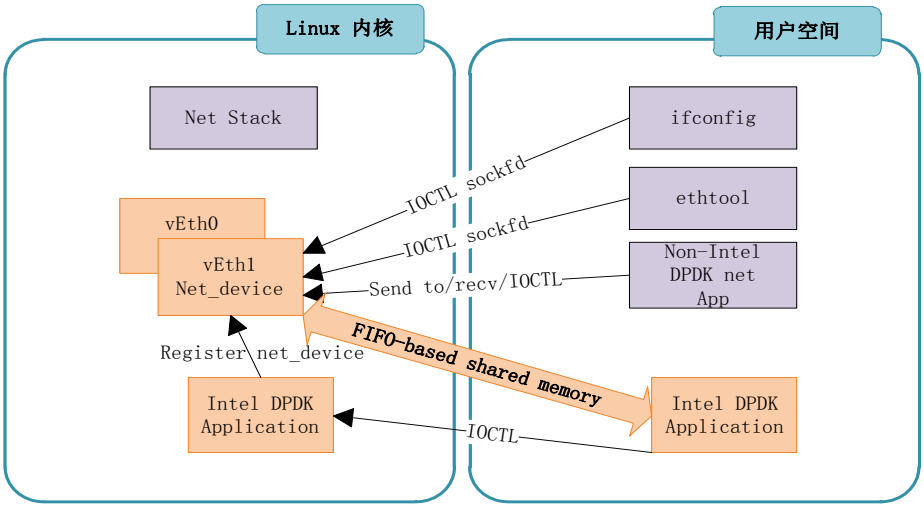

图 3.16 KNI 通信架构图

用户态协议栈通过 Intel DPDK KNI 接口数据包流如图 3.17 所示。

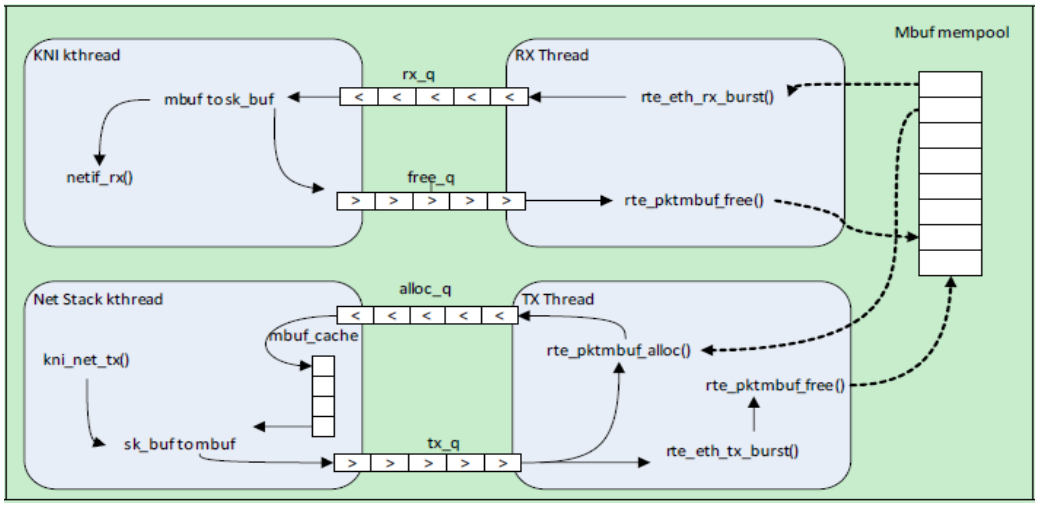

图 3.17 KNI 接口的数据流

进入 KNI 接口的数据包流: 在 Intel DPDK 的接收一侧，用来存储数据包的 *mbuf* 是通过接收线程的上下文来分配。接收线程将 *mbuf* 插入到名为 *rx_q* 的 FIFO 队列中。KNI 的线程将会轮询针对 *rx_q* 队列中的所有活动 KNI 设备。如果一个 *mbuf* 从队列中删除，那么它就会被转换成 Linux 传统协议栈中用来存储数据包的 *sk_buff*，然后将该数据包通过 *netif_rx* 发送到 Linux 协议栈中。出队列的 *mbuf* 被

放在 *free_q* 的 FIFO 队列中，在 RX 接收线程中，执行同样的循环，轮询这个 *free_q* 队列，并且在出队列后将其删除归还空间。

从 KNI 接口出去的数据包流: 首先，Intel DPDK 应用程序首先在内核侧分配 *mbuf* 缓存。通过 *kni_net_tx* 系统调用，从 Linux 内核协议栈收到数据包后，*mbuf* 从缓存中出队列并且填满从 *sk_buff* 中获取的数据，*sk_buff* 接着被释放。*mbuf* 被发送到 *tx_q* 发送队列中。TX 线程让 *mbuf* 出队列，并把它发送给轮询模块驱动。并将 *mbuf* 释放归还。

3.5 IPSec 处理模块

IPSec 处理模块工作在网络层，对于要发出去的数据包，IPSec 进行加封装。对于接收到的数据包，IPSec 进行解封装。

IPSec 有如下两种工作模式：

- 隧道（tunnel）模式：整个网络层数据包被封装到一个新的数据包中，在进行加密或者认证计算后，并添加上 AH 或 ESP 的头部。隧道模式通常适用于各个安全网关之间的通信。
- 传输（transport）模式：并不产生一个新的数据包，原有的网络数据包在进行加密或者认证计算后，添加上相应的 AH 或 ESP 头。通常情况下，传输模式适用于同一个局域网中的两台主机进行通信，或者主机和安全网关之间的通信。

AH 协议和 ESP 协议在两种模式下的的数据包封装格式如表 3.1 所示，Data 字段表示应用层的数据。

表 3.1 IPSec 数据封装格式

模式 / 协议	传输	隧道
AH	IP | AH | Data	IP | AH | IP | Data
ESP	IP | ESP | Data | ESP-T	IP | ESP | IP | Data | ESP-T
AH-ESP	IP | AH | ESP | Data | ESP-T	IP | AH | ESP | IP | Data | ESP-T

由于用户场景是两个 IDC 之间的数据传输。程序运行在两个 IDC 之间的出口网关上，所以本应用程序使用隧道模式来进行数据的传输。

IPSec 在两个主机或者网关之间提供安全的数据传输，这些主机和网关称为 IPSec 对等体。

SA 是安全关联(Security Association)的缩写。SA 是 IPSec 的基础，也是 IPSec 的本质。SA 是通信对等体间对某些要素的约定，SA 规定了使用的协议，例如是 AH 协议或者 ESP 协议或者将两者结合起来。也规定了协议的工作模式是隧道模式还是传输模式。在安全性方面，规定了加解密算法，如 AES 或者 DES，数据完整性方面的认证算法，例如 HMAC-SHA1 或者 HMAC-MD5 等。同时也规定了使用的密钥以及密钥的有效期。SA 的建立方式主要有手工配置和 IKE 自动协商两种。在本应用程序中，为了简化程序设计，使用了手工配置的方式，在程序初始化的时候，从配置文件中读取 SA 加密密钥以及 HMAC 摘要密钥。

结合三层转发协议栈，IPSec 处理模块发送普通数据包的流程如下。

(1) 根据本机地址和目的地地址，在安全策略库中相关的 SA，获取使用的协议(AH 或者 ESP)，并得到安全密钥与认证密钥，以及使用的安全算法与认证算法。

(2) 如果 SA 中定义的是 ESP 算法，则使用安全算法和安全密钥对数据进行加密，同时使用认证算法和认证密钥对消息进行摘要计算。如果是 AH 算法，则使用认证算法。

(3) 根据相关协议在模式下的消息格式，填充数据包格式。

(4) 查找路由函数，并从相应的接口将其发送出去。

接收数据包的流程如下：

(1) 如果收到的数据包是 ARP 或者 ICMP，则从 KNI 接口将其发送到传统协议栈，并从传统协议栈接口等待回复，将回复从接收端口发送出去。如果是普通数据包，根据本机地址和源地址，在安全策略库中相关的 SA，获取使用的协议(AH 或者 ESP)，并得到安全密钥与认证密钥，以及使用的安全算法与认证算法。

(2) 如果使用的协议是 ESP，则首先进行摘要计算，和数据包中的摘要字段进行比较，如果相等，则表明数据包完整，未被篡改。然后根据安全算法和密钥对数据包进行解密。根据解密后的数据包，如果目的地址不是本机，则进行路由。如果使用的是 AH 数据包，则只进行完整行认证。

3.6 本章小结

本章主要介绍了基于 DPDK 的用户态协议栈的设计框架，然后详细的介绍了在底层驱动模块、数据收发模块、三层转发协议栈、IPSec 处理模块的详细设计，根据设计框图，为快速的代码实现用户态的协议栈奠定了基础。

第四章 用户态 IPSec 协议栈的实现

4.1 底层驱动模块

Intel DPDK 的抽象环境层(EAL)负责访问底层硬件，对应用程序屏蔽底层硬件的接口，并完成诸硬件和内存空间的的初始化。在 EAL 需要对用户的参数进行检查，完成相应资源的分配。并根据用户的参数配置需要执行任务的 CPU 逻辑核和端口，接下来完成信号函数的建立，以便可以使用中断信号控制所有的执行逻辑核能够正常退出。

EAL 的初始化代码实现如下:

```
/* init EAL */
ret = rte_eal_init(argc, argv);
if (ret < 0)
      rte_exit(EXIT_FAILURE, "Invalid EAL parameters\n");
argc -= ret;
argv += ret;
```

网卡端口的初始化，初始化的过程中需要配置每个端口对应的接收队列和发送队列的数目，如下代码所示:

```
      nb_rx_queue = get_port_n_rx_queues(portid);
      n_tx_queue = nb_lcores;
      if (n_tx_queue > MAX_TX_QUEUE_PER_PORT)
            n_tx_queue = MAX_TX_QUEUE_PER_PORT;
      ret = rte_eth_dev_configure(portid, nb_rx_queue,
                        (uint16_t)n_tx_queue, &port_conf);
```

内存初始化，分配给每一个处理器对应的 mempool 缓存池的代码如下：

```
pktmbuf_pool[socketid] =
  rte_mempool_create(s, nb_mbuf, MBUF_SIZE, MEMPOOL_CACHE_SIZE,
      sizeof(struct rte_pktmbuf_pool_private),
      rte_pktmbuf_pool_init, NULL,
      rte_pktmbuf_init, NULL,
      socketid, 0);
```

由于每一个逻辑核在执行分配单元后，就进行不断的循环操作，处理数据包。为了能够控制他们退出执行单元的时机，需要对中断信号进行处理。在发生中断

的时候，中断处理函数对一个全局信号量进行修改，所有执行循环的内核在运行之前，都要检测相应的退出信号量，如果被置位，就退出执行单元。

```
static void
signal_handler(int signum)
{
  /* When we receive a SIGINT signal, stop kni processing */
  if (signum == SIGINT)
    {
        rte_atomic32_inc(&kni_stop);
        return;
    }
}
```

在初始化完毕后，主核对任务进行分发，要运行的逻辑核循环执行任务。代码如下：

```
/* launch per-lcore init on every lcore */
rte_eal_mp_remote_launch(main_loop, NULL, CALL_MASTER);
RTE_LCORE_FOREACH_SLAVE(lcore_id)
{
  if (rte_eal_wait_lcore(lcore_id) < 0)
        return -1;
}
```

整个初始化的执行序列如图 4.1 所示。

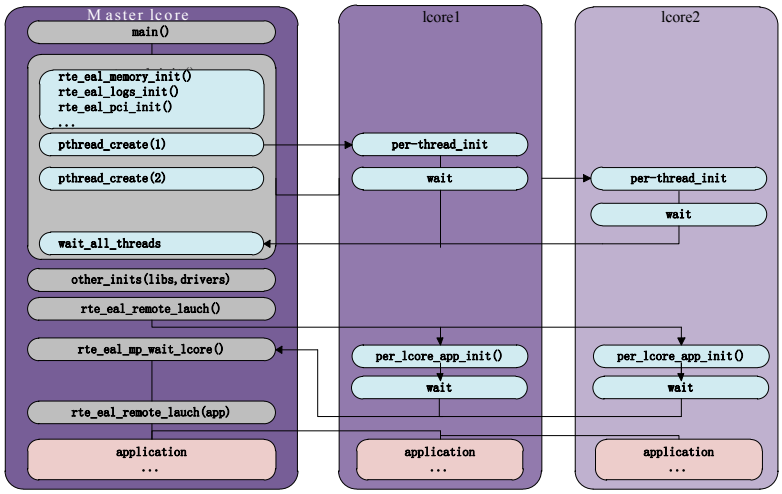

图 4.1 初始化执行序列

4.2 数据收发模块

应用程序中使用轮询的独立模式来收取数据包，两个处理器与网卡环境下的独立模式运行如图 4.2 所示。

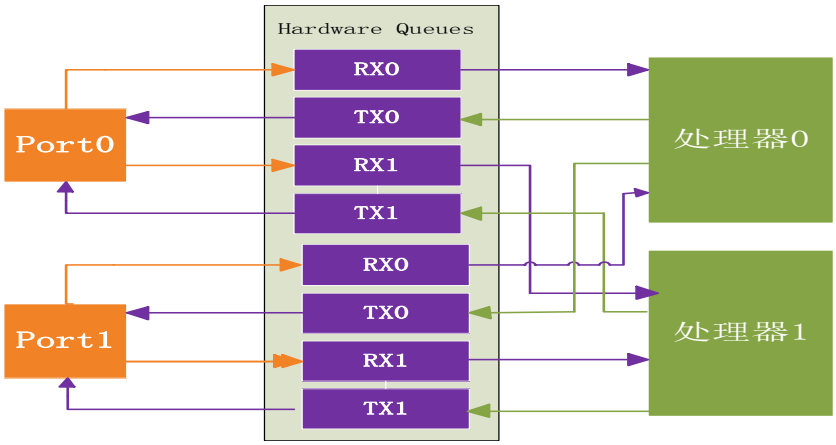

图 4.2 独立模式收发架构

每个运行的逻辑核对数据包进行批量接收，为了快速处理，将要处理的少量数据包放在逻辑核的高速缓存中，以便快速的取出，然后调用三层协议栈的处理函数 *l3fwd_simple_forward*。示例代码如下：

```
nb_rx = rte_eth_rx_burst(portid, queueid, pkts_burst,
   MAX_PKT_BURST);
if (nb_rx == 0)
   continue;
// Prefetch and forward already prefetched packets
for (j = 0; j < (nb_rx - PREFETCH_OFFSET); j++)
{
   rte_prefetch0(rte_pktmbuf_mtod(pkts_burst[
                j + PREFETCH_OFFSET], void *));
   l3fwd_simple_forward(pkts_burst[j], portid,
         qconf, 1);
}
```

在数据包被处理后，数据包的发送代码如下:

```
ret = rte_eth_tx_burst(port, queueid, m_table, n);
```

4.3 三层转发协议栈模块

4.3.1 路由转发

路由转发需要简历路由表，根据路由表来决定下一跳的地址。传统的 Linux 内核协议栈使用的动态路由表，根据 OSPF 等路由算法来建立路由表。

为了简化工作量，本文使用静态配置的路由表。路由表的初始化与建立如下所示:

```
static struct ipv4_l3fwd_route ipv4_l3fwd_route_array[] = {
   {IPv4(194,1,1,0), 24, 0},
   {IPv4(199,138,1,0), 24, 1}
};
rte_lpm_add(ipv4_l3fwd_lookup_struct[socketid],
ipv4_l3fwd_route_array[i].ip,
ipv4_l3fwd_route_array[i].depth,
ipv4_l3fwd_route_array[i].if_out);
```

在建立了路由表，进行路由查找下一跳地址的出端口的代码如下所示。

```
uint8_t next_hop = rte_lpm_lookup(ipv4_l3fwd_lookup_struct,
rte_be_to_cpu_32(((struct ipv4_hdr *)ipv4_hdr)->dst_addr),
   &next_hop);
```

4.3.2 用户态协议栈与传统协议栈通信

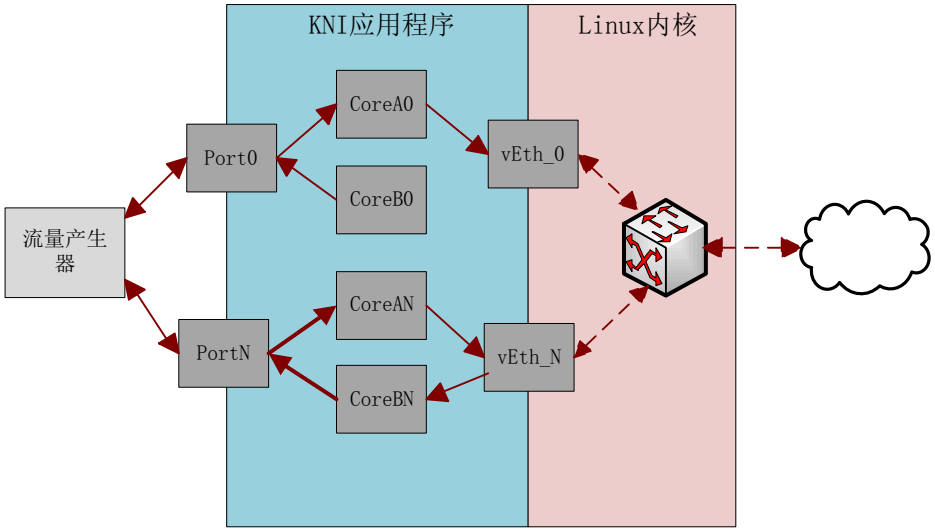

图 4.3 KNI 通信图

IDC 之间主要传输 TCP 和 UDP 的报文。但为了保证和相邻的主机相互沟通 MAC 地址，还需要 ARP 数据包进行处理。由于 ARP 数据包的量极小，将它发送给传统 Linux 内核协议栈进行处理，并将处理后的结果再发送出去。Intel DPDK 使用内核网卡接口 KNI 来与传统协议栈进行通信。同时内核网络接口 KNI 也支持 Linux 网卡配置工具 ifconfig，tcpdump 等。通过 KNI 接口与传统协议栈交互的过程如图 4.3 所示。

KNI 接口的初始化的核心代码如下:

```
ops.port_id = port_id;
ops.change_mtu = kni_change_mtu;
ops.config_network_if = kni_config_network_interface;
kni_temp = rte_kni_alloc(pktmbuf_pool[port_id], &conf, &ops);
```

在接收到 ARP 数据包后，需要将它发送给 Linux 内核协议栈，代码如下所示:

```
eth_type = rte_be_to_cpu_16(eth_hdr->ether_type);
if (eth_type == ETH_PROTO_ARP)
{
    uint16_t num = rte_kni_tx_burst(kni[portid], &m, 1);
}
```

将 ARP 发送给 Linux 内核协议栈后，不阻塞在 KNI 接口上读取内核协议栈的回复数据包，而是立刻处理下一个数据包。给每一个网卡端口分配一个逻辑核，专门负责从 KNI 接口读取内核协议栈的回复数据包，然后将其发送出去。以下是核心代码。

```
uint16_t num = rte_kni_rx_burst(kni[portid], pkts_burst, MAX_PKT_BURST);
uint16_t nb_tx= rte_eth_tx_burst(portid, 0, pkts_burst, num);
```

4.4 IPSec 处理模块

由于安全关联 SA 是单向的，在两个对等体之间需要有最少两个 SA 来保证双向通信。而且每种 SA 只能制定一种 IPSec 协议，如果需要同时使用多种协议，则在每个方向上每种协议都需要建立一个 SA。

SA 通过一个三元组来唯一标识，这个三元组包括安全参数索引 SPI（Security Parameter Index）、目的 IP 地址、使用的 IPSec 协议(ESP 或者 AH)。

SPI 是用于标识 SA 的无符号整数，在 AH 协议和 ESP 协议的数据报文包含相关的字段来表明该数据流使用的 SA。在配置文件中配置 SA 的参数时，需要明确设置 SPI。使用 IKE 方式自动协商建立 SA，SPI 是自动随机生成。

以下是 SA 的定义:

```
struct SA
{
    uint32_t src;                          //源地址
    uint32_t dst;                          //目的地址
    enum IPSEC_PROTOCOL ipsec_pro;  //采用的 ipsec 协议，如 AH 或 ESP
    enum IPSEC_MODE ipsec_mode;          //隧道模式还是工作模式
```

```
    uint32_t spi;                          //安全关联标识
    struct crypto_algo encrypt_obj;     //加密算法
    struct crypto_algo auth_obj;        //认证算法
}__rte_cache_aligned;
```

由于每对对等体之间需要两个 SA。随着对等体的增多，SA 的数量会越来越多。为了快速的查找 SA，减少开销，使用 hash 链式存储的方法，来存储 SA。以下是存储 SA 的结构体定义。

```
struct SA_list
{
    struct SA *sa_ptr; //指向 SA 指针
    struct SA_list *next;//同一个索引的下一个 SA，链式解决 hash 碰撞
}__rte_cache_aligned;
//通过源地址来进行 hash 的 SA 哈希表
static struct SA_list *sa_hashtable[LEN];
```

为了避免重复开发，安全关联使用的加密算法和认证算法由 Linux 中的 openssl 库提供，对其进行更进一步的抽象，提供一个统一的加密和认证的结构体接口。以下是结构体的定义。

```
struct crypto_algo
{
    enum CRYPTO_NAME name;      //算法名，如 3DES-CBC, SHA1-HMAC
    uint32_t key_len;           //该算法使用的密钥长度
    char key[MAX_KEY_LEN];      //密钥
    uint32_t iv_len;            //向量的长度
    char iv[MAX_IV_LEN];        //向量
    int enc;                    //标识是加密还是解密
    void (*encrypt_func)(const char *in, char *out,
            size_t length, const char *key,
            char *iv, const int enc);      //加解密函数
    void (*hmac_func)(const char *key, unsigned int key_length,
            const char *in, unsigned int in_length,
            char *out, unsigned int *out_length);     //认证函数
}__rte_cache_aligned;
```

数据包处理流程工作在第三层，根据 IP 字段的类型来判断载荷部分的数据性质。如果字段类型是 TCP 或者 UDP，那么表明这是普通的未经过 IPSec 处理的数

据包，查找 SA，进行加封装。如果载荷部分的字段类型是 ESP 或者 AH，那么该数据包是对端发过来经过 IPSec 处理的数据包，则查找 SA，进行解封装。最后对经过 IPSec 处理过的数据包进行路由。以下查找 SA 的关键代码。

```
struct SA *
sa_lookup_addr(struct selector *sel)
{
   struct SA_list *ptr = sa_array_src[sel -> src % LEN];
   if(!ptr)
      return NULL;
  struct SA *temp = NULL;

   while(ptr)
   {
      temp = ptr -> sa_ptr;
      temp = ptr -> sa_ptr;
      if(temp -> src == sel -> src &&
            temp -> dst == sel -> dst &&
            temp -> ipsec_pro == sel -> ipsec_pro)
      {
         return temp;
      }
      else
         ptr = ptr -> next;
   }
   return NULL;
}
```

AH 协议的全称是 Authentication Header Protocol。IP 类型字段号是 51，AH 协议能够提供数据完整性认证、对端身份鉴别、抗重播保护功能，AH 协议不具有加密功能，所以它传播的信息都是以明文暴露，不适合用来传输敏感数据。AH 的认证是通过一个消息摘要字段，这个字段是整个报文中绝大部分字段的摘要值，通常使用的摘要算法有 HMAC-SHA1 和 HMAC-MD5 等。AH 提供对整个数据包的完整性认证，但是对于一些会经常发生变化的字段，例如 TTL，AH 协议并不提供保护。图 4.4 是 AH 数据包的格式，黄色字段是 AH 协议保护的部分。

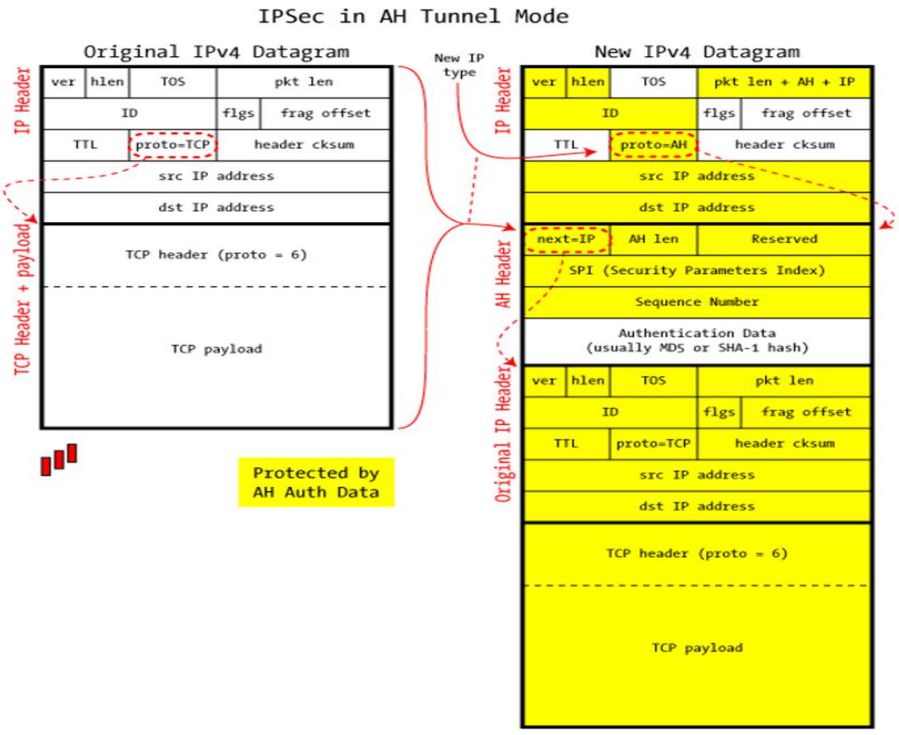

图 4.4 AH 协议数据包格式

由于 AH 协议只提供认证功能，不进行加解密。当收到一个需要进行加封装的字段后，AH 协议按照上图所示加上新的 IP 头和 AH 头后，对黄色的部分字段进行 hash 运算。最后在新加上的 IP 头部分，计算校验和。以下是核心代码。

```
struct SA *sa_op = sa_lookup_addr(&sel);   //查找 SA
ah_hdr -> spi = rte_cpu_to_be_32(sa_op -> spi);//转换为大端法
(sa_op -> auth_obj).hmac_func((sa_op -> auth_obj).key,
        (sa_op -> auth_obj).key_len,
        (const char *)ip_hdr, temp_length,
        auth_result, &auth_result_length);//计算 hash 值
rte_memcpy(ah_hdr -> auth_data, auth_result, Auth_Length);
ip_hdr -> hdr_checksum = rte_be_to_cpu_16(ip_sum_calc((uint8_t
*)ip_hdr, sizeof(struct ipv4_hdr)));  //计算 IP 校验和
```

同理对收到的数据包，进行解封装。对图中黄色的字段再次进行摘要计算，并和数据包中的摘要字段进行比较。如果一致，则表明数据未被篡改过，否则，丢弃数据包。以下是代码展示:

```
(sa_op -> auth_obj).hmac_func((sa_op -> auth_obj).key,
        (sa_op -> auth_obj).key_len,
        (const char *)ip_hdr_m, temp_length,
        auth_result, &auth_result_length); //再次计算 hash 值
```

```
if(strncmp(auth_result, icv_temp, Auth_Length)) //验证 hash 值
    return NULL;
```

4.5 本章小结

第四章讲述基于 DPDK 的用户态 IPSec 协议栈的实现过程。依据第三章的设计框架，首先讲述了底层驱动模块的初始化过程，然后罗列了数据收发模块、三层转发协议栈模块、IPSec 处理模块的核心代码，至此本文实现了一个完整的基于 DPDK 的用户态 IPSec 协议栈。

第五章 性能测试与分析

为了验证在高速网络下，基于 DPDK 的用户态协议栈相比 Linux 内核协议栈在拥有更出色的性能。构建了以下的测试环境，如图所示。其中网关 A 和网关 B 充当两个 IDC 的出口网关。测试仪 A 和测试仪 B 充当两个 IDC 中的主机，他们能够以 10Gbit/s 的速度发送数据包。

测试环境如图 5.1 所示。

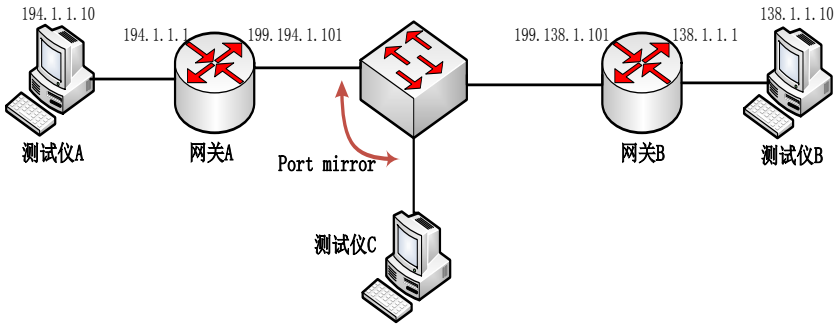

图 5.1 测试环境图

网关 A 和网关 B 是两台 Dell PowerEdge R620 服务器，拥有 2 个处理器，12 个物理核，24 个逻辑核。NUMA 架构，32G 内存。上面同时运行 32 位的 Linux 3.0 操作系统。A 和 B 各有两块万兆光口，网关 A 和网关 B 之间通过高速交换机连接，速度在万兆，不会形成瓶颈效应，故可看成网关 A,B 直连。为了判断由测试仪 A 发出的数据包经过网关 A 后，数据率发生的衰减，在与网关 A 相连的测试仪端口上做了一个 mirror，用测试仪 C 与其连接，来判断网关 A 发出的数据包速率。因此三个测试仪的数据表示的意义如下所示。

- 测试仪 A 表示发出数据包的速度。
- 测试仪 B 表示接收到数据包的速度。
- 测试仪 C 表示网关 A 发出数据包的速度。

5.1 DPDK 环境的安装与配置

因为 Intel DPDK 需要 Hugetlbfs 的支持，所以在 Linux 的启动参数上需要指定分配的大内存的空间大小与内存页的大小。在 Linux 需要加载大内存驱动，命令如下所示:

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

Intel DPDK 的安装与配置较为简单。从官网上 dpdk.org 下载源程序。这里使用的版本是 DPDK 1.7。解压后，进行编译安装。由于操作系统为 32 位的 Linux，使用的编译器为 GCC。所以执行以下的命令进行编译安装。

```
make install T=i686-default-linuxapp-gcc
```

接下来加载 DPDK 的 igb_uio 网卡驱动，执行命令如下：

```
sudo modprobe uio
sudo insmod kmod/igb_uio.ko
```

查看网卡信息的命令和结果如下所示：

```
./tools/pci_unbind.py --status
Network devices using IGB_UIO driver
====================================
0000:82:00.0 '82599EB 10-Gigabit SFI/SFP+ Network Connection'
drv=igb_uio unused=ixgbe
0000:82:00.1 '82599EB 10-Gigabit SFI/SFP+ Network Connection'
drv=igb_uio unused=ixgbe
Network devices using kernel driver
===================================
0000:04:00.0 'I350 Gigabit Network Connection' if=em0 drv=igb
unused=igb_uio *Active*
0000:04:00.1 'I350 Gigabit Network Connection' if=eth1 drv=igb
unused=igb_uio
0000:04:00.2 'I350 Gigabit Network Connection' if=eth2 drv=igb
unused=igb_uio
0000:04:00.3 'I350 Gigabit Network Connection' if=eth3 drv=igb
unused=igb_uio
Other network devices
=====================
<none>
```

这里表示有两个网卡工作在 DPDK 的 igb_uio 驱动模式下，其他网卡工作在 Linux 的 ixgbe 驱动程序下.

为了能够运行 DPDK 程序，需要使网卡运行 igb_uio 驱动，DPDK 提供了切换驱动的程序，示例代码如下:

```
./tools/pci_unbind.py --bind=igb_uio 04:00.1
```

运行 DPDK 程序需要配置环境变量 RTE_SDK.将 RTE_SDK 设置成 DPDK 的安装目录，命令如下所示：

```
export RTE_SDK=/path/to/rte_sdk
```

配置好环境之后，就可以运行开发好的程序，命令如下所示:

```
./build/l3fwd -c 606 -n 4 -- -p 0x3 --config="(0,0,1),(0,1,2),(1,0,9),(1,1,10)"
```

参数解释:

-c：CPU 逻辑核的十六进制运行掩码，要运行逻辑核置位 1，0 号核最右边的 bit。606 表示逻辑核 1，2，9，10。

-n : 每个处理器对应的内存插槽数。

-p : 要运行的网卡端口掩码。0x3 表示的是 0 号端口和 1 号端口。

--config：给逻辑核分配在每个端口上的队列。(1，0，9)表示分配 1 号端口的 0 号队列给 9 号逻辑核。

5.2 内核 IPSec 的构建

由于 Linux 2.6 以上内核自带有 IPSec 模块。配合使用开源的 ipsec-tools，就能实现 IPSec 的功能。IPSec-tools 主要包含了 setkey 和 racoon 模块。Setkey 主要用于安全关联 SA 的配置，racoon 拥有自动密钥 IKE 协议上。

IPSec 的密钥配置，主要分为主动协商(IKE)和手动配置，为了简化配置过程，而且和基于 DPDK 的 IPSec 协议栈的密钥一致。使用手动配置，以下是 setkey 的配置文件 setkey.conf 中关于 SA 的定义。

```
flush;          //清空 SA 库
spdflush;              //清空安全策略 SP 库
//添加安全策略 SP
spdadd 138.1.1.0/24 194.1.1.0/24 any -P in ipsec ah/tunnel/199.138.1.101-199.194.1.101/use;
spdadd 194.1.1.0/24 138.1.1.0/24 any -P out ipsec ah/tunnel/199.194.1.101-199.138.1.101/use;
//添加安全关联 SA 这里使用的摘要算法是 hmac-sha1
add 199.138.1.101 199.194.1.101 ah 0x201 -m tunnel -A hmac-sha1 0x9e8e4cdf3e6c0291ff014dccdd03874d9e8e4cdf;
add 199.194.1.101 199.138.1.101 ah 0x301 -m tunnel -A hmac-sha1 0x1ccdd03874d9e8e4cdf14dccdd7496ceabe0536b;
```

5.3 测试结果对比

从测试仪 A 以 10Gbit/s 的极限速度向测试仪 B 发送数据。网关 A 和 B 分别运行标准的内核 IPSec 和基于 DPDK 的应用程序 Demo。认证的摘要算法分别为 HMAC-MD5 和 HMAC-SHA1。使用百分比来表示此刻速率的达到最大吞吐量的程度。100%表示 10Gbit/s。

5.3.1 测试结果

分别在三种情况下，测试两种算法的各网关的输出吞吐率。以下是三种情况

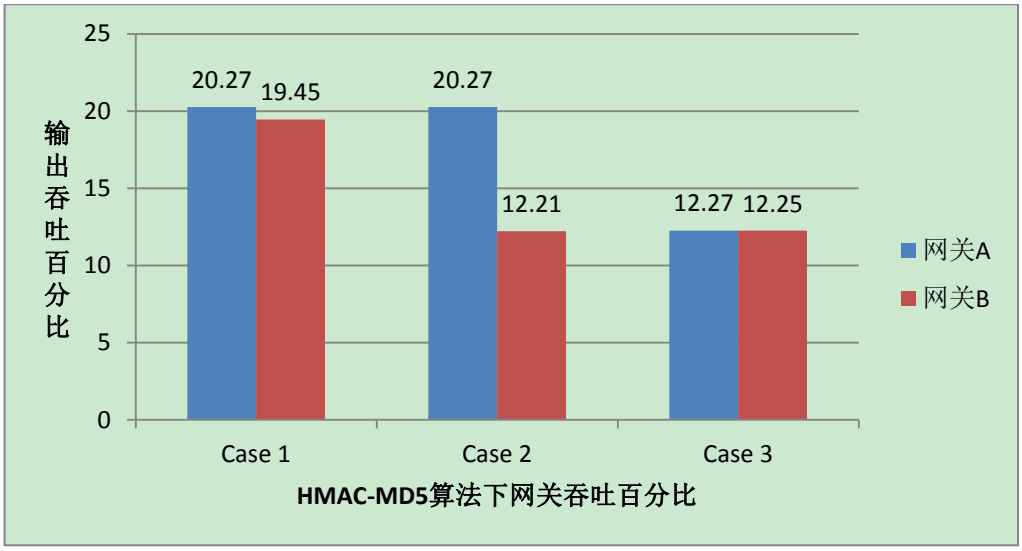

图 5.2 HMAC-MD5 算法网关吞吐百分比

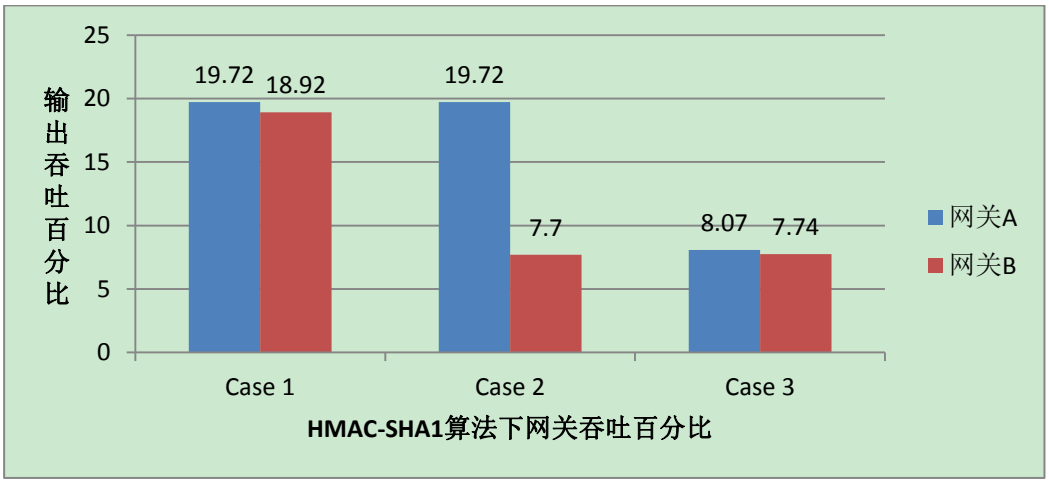

图 5.3 HMAC-SHA1 算法网关吞吐百分比

Case 1：网关 A 运行 Demo 网关 B 运行 Demo

Case 2：网关 A 运行 Demo 网关 B 运行标准 IPSec

Case 3：网关 A 运行标准 IPSec 网关 B 运行 Demo

如图 5.2 和图 5.3 所示，分别展示了在 IPSec AH 协议在 HMAC-MD5 和 HMAC-SHA1 算法下的输出吞吐占带宽总吞吐的百分比。

5.3.2 测试结论

因为网关 A 发出的是极限数据率，由于网关 A 的 IPSec 模块存在一定的处理开销，通过极限数据速率的衰减程度，可以得出以下结论:

(1) HMAC-SHA1 算法开销比 HMAC-MD5 大。

(2) 基于 DPDK 的用户态 IPSec 相比可以有效提升数据包处理能力。

(3) 在高速网络环境下，用户态的协议栈相对传统协议栈，具有明显的优势。

同时，分别在网关 A 和网关 B 运行标准 IPSec 应用程序和基于 DPDK 的 Demo，相互之间能够正常的通信，说明 Demo 满足 IPSec 流程规范。但同时，数据也表明，应用程序的 Demo 仍然还有很大的提升空间，这里只开启了 4 个核来运行 Demo，在通过调整架构，开启更多的逻辑核，会获得更高的性能提升。

5.4 本章小结

本章的主要目的是比较基于 DPDK 的用户态 IPSec 应用程序和标准 Linux 内核 IPSec 之间的性能。首先讲述了实验环境，然后简要介绍了搭建 DPDK 环境与标准 IPSec 的过程，最后通过数据对比，可以看出，基于 DPDK 的用户态程序相比标准 IPSec 应用程序，性能有了有效的提升。说明在高速网络环境下，用户态的协议栈相对传统协议栈，具有明显的优势。

第六章 总结与展望

IDC 在高速数据传输中使用的 IPSec VPN 与传统协议栈高度耦合，传统协议栈在高速网络下的中断频繁、冗余拷贝、不支持多核框架等问题使它成为高速网络下的瓶颈。并行化的用户态协议栈已经成为了热门的研究课题，Intel 公司推出了优秀的用户态应用协议栈开发平台 DPDK，通过 DPDK 可以快速的开发用户态协议栈程序。

本文的主要工作包括以下几点:

(1) 阐述了基于传统协议栈的 IPSec VPN 面临的问题，并简要介绍了国内外并行协议栈的研究现状，以及各种协议栈的加速技术。

(2) 介绍了 DPDK 在解决传统协议栈不足的关键技术：大内存页、用户空间 I/O、处理器亲和性。

(3) 阐述了基于 DPDK 的用户态 IPSec 协议栈方案。并详细的描述了框架中底层驱动模块、数据收发模块、三层转发协议栈模块、IPSec 处理模块等。这些模块针对解决了传统协议栈的中断频繁、数据冗余拷贝、多核框架支持不够等问题。

(4) 基于设计框架，实现了基于 DPDK 的用户态 IPSec 协议栈。并在高速网络环境下与传统的 IPSec VPN 方案进行了测试对比，展示了基于 DPDK 的用户态 IPSec 协议栈的性能优势。

随着移动互联网崛起，以及未来物联网的蓬勃发展，大数据的高速传输已经成为了协议栈必须面对的问题，本文虽然设计了基于 DPDK 的用户态 IPSec 协议栈，并获得了明显的性能提升，但仍然面对以下问题:

(1) 数据吞吐率有待进一步提高，目前只用到了 10Gb 带宽的 20%,未来 40Gb 网络或者更高速网络的普及，需要更高的数据吞吐支撑。

(2) 目前的 IPSec 设计较为简单，没有实现密钥的自动协商，未能实现一定程度的防攻击策略，在大规模的 DDOS 等攻击下，容易瘫痪。

(3) 移植性较差，Intel DPDK 目前支持只 Intel 网卡与特定的操作平台。

参考文献

[1] Al-Fares M, Loukissas A, Vahdat A. A scalable, commodity data center network architecture[C]//ACM SIGCOMM Computer Communication Review. ACM, 2008, 38(4): 63-74.

[2] Naganand D,Dan H.IPSec: The New Security Standard for the Internet, Intranets and Virtual Private Networks. . 1999

[3] 查奇文,张武,曾学文,宋毅. 基于多核处理器的 TCP/IP 协议栈加速技术[J]. 网络新媒体技术,2013,01:58-64.

[4]王小峰. 面向 TOE 的快速报文传输研究与实现. 硕士毕业论文. 国防科学技术大学, 2006.

[5]Intel. intel-dpdk-getting-started-guide.pdf. January 2014.

[6] Bhoedjang R A F, Ruhl T, Bal H E. User-level network interface protocols[J]. Computer, 1998, 31(11): 53-60.

[7] 章巍巍. 用户态并行协议栈关键技术的研究与实现[D].哈尔滨工程大学,2012.

[8]David Ely, Stefan Savage, and David Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. Department of Computer Science and Engineerin University of Washington, Seattle WA

[9]Chris Maeda,Brian Bershad. Protocol Service Decomposition for High. Performance Networking[C]．Proceedings of the Fourteenth ACM Symposium on Operating Sy,；terms Principles(SOSP)．Asheville，NC，USA：1993：1263—1271P

[10] Engelke E. WATTCP[J]. Home page at http://www. wattcp. com/index. shtml, 2004.

[11] Solarflare. http://www.openonload.org/

[12] 董春雷,郑纬民. 基于 Myrinet 的用户空间精简协议[J]. 软件学报,1999,03:76-80.

[13] Zimmermann H. OSI reference model--The ISO model of architecture for open systems interconnection. Communications, IEEE Transactions on, 1980, 425-432.

[14] Stevens W R. TCP/IP illustrated:The protocols, vol. 1. 1994.

[15] Fall K R, Stevens W R. TCP/IP illustrated, volume 1: The protocols[M]. addison-Wesley, 2011.

[16] Shivam P, Wyckoff P, Panda D. EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing[C]//Supercomputing, ACM/IEEE 2001 Conference. IEEE, 2001: 49-49.

[17] Boucher L B, Blightman S E J, Craft P K, et al. TCP/IP offload network interface device: U.S. Patent 6,434,620[P]. 2002-8-13.

[18] Boucher L B, Blightman S E J, Craft P K, et al. Transmit fast-path processing on TCP/IP offload network interface device: U.S. Patent 6,965,941[P]. 2005-11-15.

[19] Bhoedjang R A F, Ruhl T, Bal H E. User-level network interface protocols[J]. Computer, 1998,

31(11): 53-60.

[20] Krishnakumar R. Hugetlb—Large Page Support in the Linux Kernel[J]. Linux Gazette, 2008, 155.

[21] Boyd W T, Hufferd J L, Mena III A, et al. Method for out of user space block mode I/O directly between an application instance and an I/O adapter: U.S. Patent 7,502,872[P]. 2009-3-10.

[22] Fusco F, Deri L. High speed network traffic analysis with commodity multi-core systems[C]//Proceedings of the 10th ACM SIGCOMM conference on Internet measurement. ACM, 2010: 218-224.

[23] Hill M D, Marty M R. Amdahl's Law in the Multicore Era[J]. IEEE Computer, 2008, 41(7): 33-38.

[24] Gustafson J L. Reevaluating Amdahl's law[J]. Communications of the ACM, 1988, 31(5): 532-533.

[25] Krishnakumar R. Hugetlb—Large Page Support in the Linux Kernel[J]. Linux Gazette, 2008, 155.

[26] 蒋东毅,吕述望,罗晓广. VPN 的关键技术分析[J]. 计算机工程与应用,2003,15:173-177.

[27] 谢大吉. 基于 PPTP 的 VPN 技术研究[J]. 四川文理学院学报,2011,02:58-60.

[28] 王妍. 基于 IPSec 的 VPN 系统设计与实现[D].电子科技大学,2013

[29] 廖悦欣. IPSec 协议实现技术研究[D].华南理工大学,2013.

[30] 张强. IPSec VPN 技术研究与工程实现[D].西安电子科技大学,2009

[31] Dawkins G J, Lee V H. System and method for determining which processor is the master processor in a symmetric multi-processor environment: U.S. Patent 6,178,445[P]. 2001-1-23.

[32] Bolosky W J, Scott M L, Fitzgerald R P, et al. NUMA policies and their relation to memory architecture[C]//ACM SIGARCH Computer Architecture News. ACM, 1991, 19(2): 212-221.

[33] 刘恒. 并发数据结构及其在动态内存管理中的应用[D].重庆大学,2013.

致谢

匆匆十八载，即将阔别校园，踏上工作岗位，感慨与不舍，交织在自己心中。这一路上，自己在成长，从一个幼稚的孩童到可以去拼搏的青年。遇到了许多优秀的恩师与可爱的同学。是他们一直影响自己，让自己逐渐成熟。尤其是西电的6 年半时光里，自己在学习以外，对以后进入社会思考了许多，也得到了许多学长老师的帮助。特别是在本科期间，认识了我现在的导师权义宁副教授。权老师的乐观幽默与亲易近人让自己备受亲切，他对科研工作的一丝不苟让自己深深的感受到了无论什么事情，都需要全身心的投入。权老师丰富的知识储备使自己认识到学习不仅是在学校期间，而是在我们每一天的生活中。感谢恩师，在即将驶入社会海洋之前，稳稳的掌舵，指导我们前进的方向。

感谢实验室中小伙伴，王静，李焦贤，刘飞腾，马翔宇，刘晓静，李晓清，侯瑞红同学，我们一起科研，相互学习帮助。实验室就像一个大家庭，在权老师的带领下，其乐融融。

感谢我的女朋友，在完成这篇论文过程中，我很少陪你，谢谢你默默的支持。

感谢我的死党们，在远离父母，独自求学的时候，在最美的大学中，和你们在一起有了家的感觉。

最后一声感谢，献给我的父亲与母亲，或许这已经不能感谢来表达，这是无法度量的恩情。虽然，很久以前，你们的知识已经不能辅导我完成功课，但你们一直是推动我完成学业的坚强后盾。你们只是普通的农民，却给了我百分之百的支持。你们将我抚养成人，让我完成学业，改变自己的人生命运。我人生的每一步都有你们的操劳，诚实、坚持与奋斗从你们那里获得的品质。青丝白发，你们也老了。从现在开始，我即将独立去拼搏，相信儿子会飞的更高，将用百倍的努力。在某个城市拥有一盏属于我们的灯火，一家人一起生活，陪伴你们度过每一天。

最后，感谢对我的硕士论文进行评审和答辩的各位专家教授，你们辛苦了，感谢你们对我的论文提出宝贵意见，谢谢！

作者简介

1. 基本情况

男，陕西汉中人，1990 年 4 月出生，西安电子科技大学计算机学院计算机技术领域 2012 级硕士研究生。

2. 教育背景

2008.8～2012.07　就读于西安电子科技大学计算机学院计算机科学与技术，获工学学士学位

2012.08～2015.01　西安电子科技大学计算机学院计算机技术领域硕士研究生

3. 攻读硕士学位期间的研究成果

1.恶意程序的动态监测，2013/4~ 2013/7，利用机器学习算法对恶意程序与正常程序的行为进行提取与学习，来检测未知的恶意程序。负责模拟一个真实的网络环境，欺骗恶意程序，使其能够正常的运行，并对捕获的网络数据包进行分析，提取网络行为，为机器学习算法提供数据支持。

2. 用户态并行协议栈的研究与实现，2014/2~ 2014/6，项目研究如何在用户态上实现一个并行用户态协议栈，解决传统 TCP/IP 协议栈在高速网络网络环境下，因频繁的系统调用，造成用户态与内核态的切换成本过高，成为了传输的瓶颈。负责控制任务分配与项目进度，并承担一个主要模块的研究与实现。

西安電子科技大學

XIDIAN UNIVERSITY

地址：西安市太白南路2号

邮编：710071

网址：www.xidian.edu.cn