

电子科技大学

硕士学位论文

网络数据流高速采集系统设计与实现

姓名：陈卫屏

申请学位级别：硕士

专业：信息与通信工程

指导教师：秦志光

20090501

摘 要

随着人类进入信息时代, 网络和人们的关系越来越密切, 它在带来很多方便的同时, 也带来了许多新的问题。其中互联网的安全性引起了各个国家、公司的高度重视。被动数据捕捉技术在网络安全领域有着极其丰富的应用, 如 IDS、防火墙等, tcpdump, ethereal, snort 等软件都采用此项技术。随着诸多应用层服务的不断投入应用, 使得网络的承载能力大为紧张。由于 CPU 性能、操作系统的处理机制等原因, 传统的捕包方式已经不能适应千兆网络的要求。

高性能数据采集平台是定位于 Internet 主干网络的数据采集和数据分析设备, 支持千兆级及以上的网络数据采集和数据分析。它采集网络上传输的所有数据报文; 数据分析功能支持从链路层, IP 层, 传输层, 直到应用层的数据分析。该平台提供方便的二次开发接口, 在此平台上可以方便的开发基于内容的网络安全设备, 如 IPS 系统、DDOS 防御系统、垃圾邮件过滤系统等。本文研究和实现了高性能的网络数据采集平台中的底层数据包捕获子系统。它结合“零拷贝”技术, 实现了对底层数据包高效, 安全的捕获。它主要由下面几个技术部分组成:

(1) 内存映射技术。本系统采用内存映射技术, 使用户空间和内核空间共享一片内存区域。这片内存区域将用于存放数据包和必要的内存管理数据结构, 通过对这块共享内存的使用和管理, 为数据包在内核空间和用户空间之间的高速传递提供支持。

(2) “零拷贝”技术。系统将改变网络中数据包提交流程, 使数据包在由网卡到上层应用程序的传送过程中绕过内核协议栈, 避免了由内核空间到用户空间的数据拷贝, 从而提高数据包捕获效率。

(3) 与协议还原和处理系统的接口模块。本系统中, 位于用户空间的协议还原子系统通过依层次注册回调函数的方式实现对网络数据包的分析 and 处理。该处理方式上层协议还原和处理子系统提供了很高的灵活性, 上层系统可以通过注册不同的回调函数组合来满足在不同应用场景下的需求。

此外, 我们根据项目进度及要求, 在装有 Linux2.6 内核版本的 PC 机上实现了网络数据流高速采集系统。

关键词: 数据捕获, “零拷贝”技术, 内存映射, Linux 内核

Abstract

As mankind enters the information age, the relationship between internet and people is closer. The internet brings a lot of convenience, but it also has brought many new problems. Internet security is highly valued by many countries and companies. Passive capture is widely used in internet security IDS and firewall. Many software used in information security such as tcpdump, ethereal and snort use this mechanism. nowadays, more and more applications become popular in the network, which overload the net.

The high performance data collect system is a equipment that use to capture and analyse datas in the internet. It can support the data capturing and analysing in a 1000MB level network. The data capture subsystem capture all the data packets while the analyse subsystem can analyse the data from link layer to application layer. this system offer a interface for a secondly development, which can let others to develop many other content-based network security equipment, such as IDS, firewall, and so on. Based on the above points, we realized the data packets capture subsystem which belong to the high performance data collect system by using the "zero copy" technology. It can capture the data packets effectively and safely. It is composed of the following parts:

(1) Memory map technology. We use this technology in the system to make the kernel and the application to share the memory. This memory is use to store the data packets and the data structures that use to manage the memory. This memory will be a support to the high-speed transporting of the data packets from kernel and application.

(2) "Zero copy" technology. This system will change the way that submitting a data packet from the netcard to the application. It avoids the entering the TCP/IP protocol stack in the kernel, which will reduce the efficiency of a data transport.

(3) The interface to the protocol revert subsystem. in this system. the protocol revert subsystem which located in the application space will analyse the data packets by regist callback functions. This mechanism give a effective and flexible way for the protocol revert subsystem. it can satisfy different request by registering different combination of callback functions.

Additionally, a low-level data packets capture subsystem based on PC with linux 2.6 was designed according to the article requirements.

Keywords: data capture; zero copy; memory map; linux kernel;

独 创 性 声 明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，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作了明确的说明并表示谢意。

签名： 陈卫屏 日期：2009年6月4日

关于论文使用授权的说明

本学位论文作者完全了解电子科技大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权电子科技大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后应遵守此规定）

签名： 陈卫屏 导师签名： 李毅
日期：2009年6月4日

第一章 绪论

1.1 背景

随着网络的普及，安全问题正威胁着每个网络用户。因此对计算机的网络监控十分必要，而其中对网络数据流的捕获和分析尤为重要。网络数据流捕获被广泛地应用于分布式实时控制系统、网络故障分析、IDS 系统、网络监控系统、计算机取证系统等领域中，是其中的关键技术之一。随着网络带宽不断增加，监控高速网络数据流的需求日益增加，高性能的网络数据流采集在面向大规模宽带网络的入侵检测系统、大流量网络数据情况下的网络协议分析、防火墙、高性能通信系统、高性能路由器、主机路由器等领域中，都有广泛的应用前景。

但是，仅仅是几年前，以太网的带宽还是 100Mbps，网络的飞速发展，使得以太网的带宽在这短短的几年间，从 100Mbps 发展到 1000Mbps，再到 10Gbps。相信不久的将来，10Gbps 的以太网将会像现在的 1000Mbps 以太网一样被广泛应用。但是处理器的性能却没有紧跟其步伐，尤其是内存带宽和访问延迟的改进速度大大落后于网络带宽的发展。根据中国互联网络信息中心 2005 年一份统计报告显示，我们 Internet 网络发展迅速，是 1997 年的 139.1 倍。同时，网络数据包的处理非常耗 CPU 资源，即使当今最快的 CPU，TCP/IP 协议处理能够轻易地消耗掉 80%到 90%的计算资源，并且还达不到千兆线速的性能。在网络中，流传着一句经验之谈：在最好的情况下，1MHz 的 CPU 能处理 1Mb 的网络带宽，按照这个规律，现有 3.9GHz 的 CPU 只能处理 3.9Gb 的网络带宽，这在 10Gps 网络没有被广泛应用的时候处理器的性能是能满足网络带宽的需求的，验证了当更快的网络被广泛的应用时，总能找到足够快的 CPU 满足其需求。但是相对而言，内存带宽与访问延迟却发展不够快，与 CPU 性能间的差距在不断增大，因此随着相关技术发展的步调不一致，仅仅靠 CPU 主频的提高并不能完全解决高网络带宽带来的问题。为了能更快地处理网络数据包以满足当今网络带宽的需要，而又不过多的占用系统资源，使得系统能有更多的资源用于应用处理，有必要再一次对网络数据包处理加速技术进行研究，以弥补各种处理技术的发展跟不上网络发展速度的步伐所带来的缺陷。

所以我们基于网络数据包的捕获和分析的重要性和网络数据包的捕获和分析

的效率问题提出本项目，目的在于解决网络数据包的捕获和分析的效率问题，保证网络数据包的捕获和分析可靠性，实用性。

1.2 课题来源和研究意义

本课题来源于华为公司高校基金项目《智能蜜网系统》的延伸预研项目《网络数据流高速采集系统》。

目前众多信息安全产品的开发过程中，要么直接利用 Libpcap 等开源软件实现底层数据捕获，要么各自针对上层应用采取不同手段单独实现网络数据包采集工作。这样开发出的产品往往在效率难以令人满意，同时大量的人力物力被投入到重复的工作中，造成人力资源浪费。

随着网络带宽的不断增大，上层各种应用对数据包采集效率的要求也在不断提高，类似 Libpcap 的传统的数据包采集模式在实际应用中已显得捉襟见肘。而 Linux 操作系统对开发提供了众多方便的接口，同时，专属的网络数据包采集硬件的灵活性和扩展性又难以达到要求。因此，在目前广泛使用的 Linux 平台下开发一种通用，高效的网络数据流高速采集系统就具有了非常重要的意义。

1.3 论文的主要工作

本论文系统讨论了被动数据捕获方法，并阐述了 Linux 内核中与被动数据捕获相关机制，主要完成了基于 Linux 平台的网络数据流高速采集系统的设计和实现。

(1) 对现有的数据捕获技术进行了综合归类和分析研究，讨论了各种技术的特点，并对现有的提高数据捕获效率的零拷贝，NAPI 技术进行研究，客观评价了各技术的适用环境和性能。

(2) 分析了 Linux 内核协议栈工作机制，研究了网络数据包在 Linux 系统中的传输过程。

(3) 利用“零拷贝”实现了高速网络数据流采集，并将之实现到系统中。

(4) 主要设计了通用数据采集平台系统中的几个有关高速网络数据流采集的子模块：修改后的网卡驱动模块，内核虚拟模块，上层协议分析接口模块。

(5) 结合本研究课题中使用的硬件设备，将高速网络数据流采集技术和 Linux 内核特性相结合，设计并实现了一个网络数据流高速采集系统。

1.4 论文结构

本文在分析了 Linux 内核工作机制和常见数据捕获技术的基础上，结合网络数据流高速采集系统的需求，设计并实现高速数据流采集系统。

本轮文章节的安排如下：

第一章解释了课题背景，解释了为什么要研究网络数据流采集技术，以及本课题的来源和主要工作。

第二章介绍了当前被动数据捕获的相关技术，分析了各种现有技术存在的问题以及对现有技术的改进，同时介绍了与数据捕获技术相关的 Linux 内核机制，包括 Linux 内核协议栈，Linux 模块机制以及 Linux 网卡驱动程序。

第三章简要介绍了网络数据流高速采集系统，并说明了底层数据捕获子系统在网络数据流高速采集系统中的地位，以及对应模块，并对与之协同工作的上层协议还原子系统工作方式进行了概述。

第四章详细阐述了底层数据捕获子系统的构造，以及其中主要功能模块的设计和实现。

第五章对高速数据流采集平台进行了测试，通过图表方式展示了系统的功能与性能。

最后对研究课题和论文工作进行了总结，指出工作中存在的不足和下一步工作方向。

1.5 小结

网络信息的安全性和安全服务质量在信息安全领域有着重要的地位，保持一个安全的网络信息环境具有重要意义，这一点已经引起了国家的高度重视。随着英特网技术的快速发展和网络应用环境的不断普及，各种信息安全产品也得到了越来越广泛的应用，这些种类繁多的安全产品保障着网络安全秩序。尽管这些产品功能不同，但其中绝大部分功能的实现都以网络数据流的采集和分析技术为基础，网络数据包捕获技术的研究对于保障网络的安全有重要的意义^[1]。在 IDS 以及许多病毒监控、防垃圾邮件和防火墙等应用中，数据包捕获技术都是基础。因此，为了提高各信息安全产品的工作效率，提高网络数据采集的效率已迫在眉睫。

第二章 被动数据捕获相关技术和 Linux 内核机制简介

本论文主要研究基于 Linux 系统的网络数据流采集技术，在此本章先分别就现有的被动数据捕获相关技术^[2-5]和与之相关的 Linux 内核机制^[6]作简要介绍，作为后续章节的基础。

2.1 被动数据捕获相关技术简介

由于数据捕获技术在网络安全领域所具有的基础作用，多年来，数据捕获技术一直是网络安全相关领域研究热点，各种效率不断提高的新技术也层出不穷。下面就简单的介绍这些数据包捕捉技术。

2.1.1 现有数据包采集技术原理

目前可以用两种方法来实现网络底层数据流采集，即底层数据包捕获，一种是利用以太网络的广播特性实现，另一种是通过设置路由器的监听端口实现，两种方法在不同情况下分别适用。

2.1.1.1 利用以太网的广播特性进行监听

在以太网中，数据的传播是通过广播方式实现的，这意味着在同一个局域网的所有网络接口都可以接收到该局域网中的所有数据包。但是在操作系统正常运行时，网络应用程序只将那些目的地址为本主机的数据包提交给应用程序，而其他数据包过滤后将被丢弃，操作系统可以在链路层、网络层和传输层这几个层次来实现该过滤机制。

要想捕获到网卡收到的但目的地址不是自己主机的网络数据包，必须想办法绕过系统正常的工作机制，直接访问网络底层。我们可将网络接口设置为混杂模式。当网络接口处于这种“混杂”方式时，它对所有接收到的数据包都产生硬件中断，操作系统收到该中断后，会对数据包进行接收处理。（目前绝大多数的网卡都支持设置成混杂模式）。操作系统会直接访问数据链路层，捕获相关数据包，由应用程序对数据包进行处理，而非传统的上层 TCP/IP 协议如 IP 层、传输层协议

对数据包进行处理，这样就可以捕获到流经网络接口的所有数据包。可见，监听程序工作在网络环境中的最底层，它会捕获所有的正在网络上传送的数据包，并且通过相应的上层应用程序处理，可以实时分析这些数据包的内容，进而分析所处的网络状态。值得注意的是监听程序是极其安静的，它是一种被动的捕获方式，且这种监听方法对所有共享式广播网络都同样适用，如令牌网。

目前，UNIX 系统中有 3 种常见的数据链路层访问设备，可以用来捕获和过滤链路层上的数据包，它们分别是：BSD 系统中采用的 BSD 分组过滤器(BPF)，SVR4 中的数据链路层接口(DLPI)和 Linux 系统的 SOCK_PACKET 接口。

2.1.1.2 基于路由器或交换机的网络底层信息监听技术

在实际应用中也存在如光纤接入等其他网络接入方式，在其他这些不是以太网接入方式下，就不能利用以太网的广播特性进行监听了。而是在路由器或交换机中设置镜像端口，将流经路由器或交换机中某一端口中的数据完全复制到镜像端口中，通过对镜像端口中的数据进行分析来达到监听的目的。目前大多数交换机，路由器都提供镜像端口的功能。

2.1.2 现有数据包采集技术

包捕获机制是依赖于操作系统的，这种机制下网络数据包将被复制一份，在传统的 TCP/IP 协议栈处理的同时，复制出的包会经历数据捕获的机制的流程。Linux 操作系统提供了一种链路层的套接字 SOCK_PACKET，这种套接字绕过内核中的 TCP/IP 协议栈，而直接从网卡驱动程序读取数据包，所以用户使用这种套接字可以直接从链路层获取原始网络数据报文。如果先用 Linux 下的设备管理函数 ioctl 把网卡设置成混杂模式，用户就能用此套接字监听流经所在局域网的所有数据包。这种方法缺点是效率较低。在其他操作系统上也有类似的机制，如：WINDOWS 和 DOS 环境下，可通过 vpacket.vxd 网卡驱动程序来捕包；SunOS 中有 NIT 接口；在 DEC 的 Unix 环境下有 Unix Packet Filter；在 SGI 的 IRIX 中有 SNOOP；另外还有 BSD 的 BSD Packet Filter(BPF)。其中 BPF 是一种效率较高、应用广泛的数据包捕获工具。

2.1.2.1 BPF: BSD 分组过滤器

BPF(Bedeley Packet Filter), 伯克利数据包过滤器, 是由美国加州大学伯克利分校在 BSD 操作系统上设计的一种数据包接收和过滤机制, 现在被应用于很多系统, 如 Linux, BSD 等^[7-8]。

BPF^[9]有两个主要组成部分, 网络分接头(network tap)和过滤器(packet filter)。网络分接头从网络设备驱动程序中获取数据包并转发到监听程序。过滤器决定是否捕获该数据包, 以及接收该复制数据包的哪些部分。BPF 过滤器的过滤功能是通过虚拟机执行过滤程序来实现的。结合图 2-1, 数据包捕获的过程可描述如下: 当数据包到达网卡时, 网卡驱动程序将其提交到内核 TCP/IP 协议栈; 如果 BPF 正在此网络接口监听, 驱动程序将首先调用 BPF, BPF 将数据包发送给过滤器, 过滤器对数据包进行过滤, 并将数据向上提交给与过滤器关联的应用程序; 然后系统控制权将交还给网卡驱动程序, 将数据包提交给上层的内核协议栈处理。由于网络数据包到达的间隔是很短的, 如果对每个数据包都调用一次系统调用来提交, 则效率上开销是很大的, 为此, BPF 过滤器会缓存多个包, 然后将它们作为一个整体提交给应用程序。为了在这个整体中区分原始的每一个数据包, BPF 给每个原始数据包加上了一个包头, 它包括了包头长度, 原始数据包长度, 时间戳等信息。

各种应用程序往往只关注一部分数据包, 如果对每一个数据包都进行一次复制操作, 这无疑浪费了系统资源。对此, BPF 的做法是将数据包的指针提交给过滤器, 当过滤器决定接收该数据包时, 才将数据包复制一份。这种方案大大减少了内存拷贝的操作量, 极大地提高了系统的性能。

另外, 在图中我们还可以看到, BPF 使用了一个简单的 buffer 模型, 即非共享的 buffer, 在当前计算机系统一般有较大 RAM 的情况下, 对于常规的数据包的捕获是很有效率的。

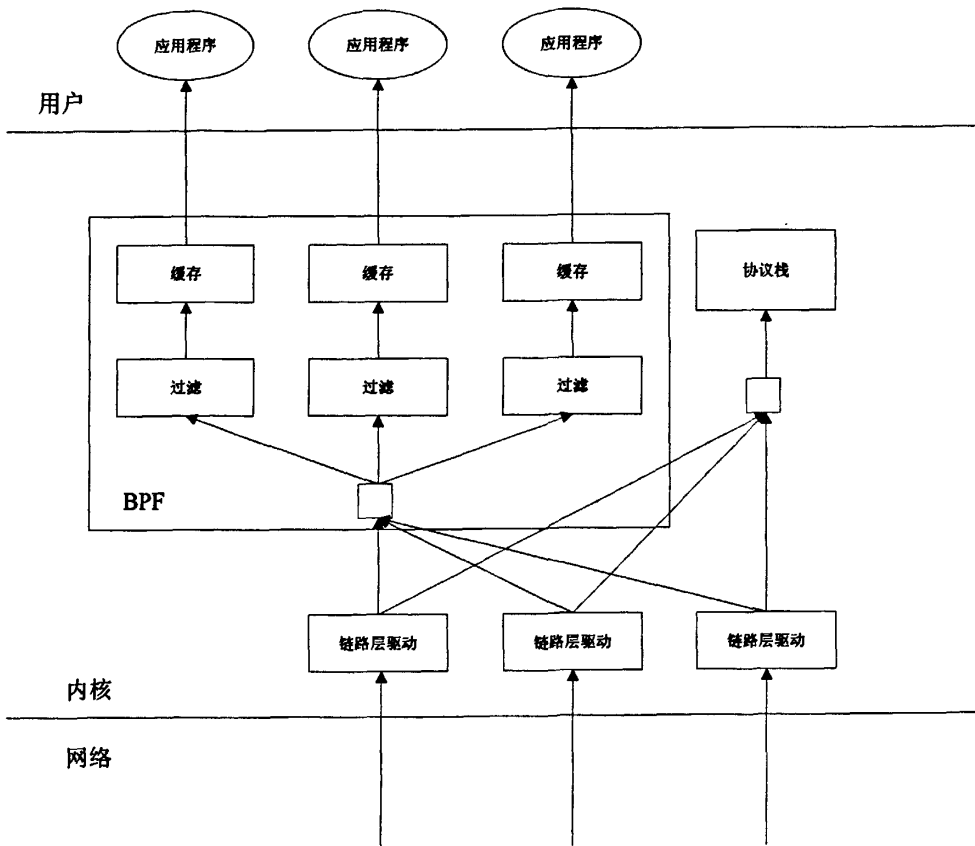


图 2-1 BPF 数据捕获原理

2.1.2.2 DLPI: 数据链路提供者接口

SVR4 通过 DLPI 来实现数据链路访问。DLPI 是 AT&T 设计的独立于协议的访问数据链路层所提供的服务接口，其访问通过发送和接收流消息来实现。

DLPI 规定了一个符合 ISO 的数据链路服务定义 (Data Link Service Definition, ISO8886) 和逻辑链路控制 (Logical Link Control, ISO8802/2) 的内核流实现。DLPI 定义了流消息 (STREAM message) 的集合、状态表和以及相应的约定，并且定义了数据链

路服务的使用者和提供者之间交换的原语集合以及原语的使用规则，允许数据链路服务的提供者 (DLS provider) 和使用者 (DLS user) 的交互动作。DLPI 使得数据链路服务的使用者能够直接访问数据链路层，如获取数据链路层的状态，

接收和发送数据链路层的原始帧，而不需要知道数据链路层本身的协议。只要数据链路层服务的提供者支持标准的 DLPI 接口，不论数据链路层是 X.25 LAPB, SDLC 还是 Ethernet, FDDI, Token Ring 的协议，使用者都可以通过 DLPI 接口来对数据链路层进行访问。

2.1.2.3 SOCK_PACKET: Linux 系统中的套接字

通常使用的 SOCKET^[10]包括：流式套接字(SOCK_STREAM)和原始套接字(SOCK_RAW)以及数据报套接字(SOCK_DGRAM)。Linux 除了提供这几种 SOCKET 外还提供另外一种套接字：SOCK_PACKET，通过它可以直接对网络底层进行操作。这种 SOCKET 接口是操作系统内核提供的应用程序访问网络链路层的编程接口。通过这个 SOCKET，应用程序可以获取到网卡传递到系统内核的所有数据包。因为这些报文同样会按传统方式提交给内核中的 TCP/IP 协议栈，所以在应用程序中使用 SOCK_PACKET 套接字不会对正常的网络通讯造成任何影响。由于 SOCK_PACKET 套接字功能强大，能进行底层的操作，因此使用起来需要开发者进行良好的控制。

2.1.3 Libpcap 简介

上面所讲的数据包捕获模块和过滤模块机制通常在内核层，由操作系统来具体实现它们。但是一般用户往往希望这样的机制不依赖于操作系统，能够在应用层得到实现，以提高数据捕获机制的灵活性。这样就需要这样的一个函数库：它需要操作系统的支持，建立在数据包捕获和过滤模块之上，但却能提供一套和操作系统无关的函数调用接口供应用程序使用，应用程序能通过这些函数于操作系统内核中的数据捕获模块通信，但同时又独立于具体的操作系统，从而达到数据捕获的目的。

Libpcap^[2] (Packet Capture Library) 是 Unix/Linux 平台下的网络数据包捕获的函数库。它是一个独立于系统，提供给用户层的包捕获 API 接口，为底层网络监听提供了一个可移植的框架。Libpcap 支持 BPF(Berkeley Packet Filter) 过滤机制。目前很多优秀的网络数据包捕获软件都是以 Libpcap 为基础，如著名的 Tcpdump、Ethereal 等。

Libpcap 主要包括三个部分：最底层的是针对硬件设备接口的数据包捕获机制，中间的是针对内核级的包过滤机制，第三层是针对用户程序的接口。

Libpcap 提供了几个重要函数来实现数据包的捕获。首先, `pcap_lookupdev()` 函数可以用于查找系统中所有可用的网络设备, 返回网络接口名称。然后, 利用 `pcap_t *pcap_open_live()` 函数来打开所查找到的某个特定网络接口, 获得一个句柄, 作好了捕获数据包的准备。同时我们可以利用该函数来设置各个与网络数据包捕获相关的参数, 如每次捕获数据包的最大长度、接口设备的状态、分配错误信息缓冲区的大小等。Libpcap 中可以选在用户空间或者是内核空间来执行过滤器, 但是由于将数据包由内核空间拷贝到用户空间要耗费大量的系统资源, 为了减少从内核空间向用户空间拷贝的数据包的数量, 提高捕获数据包的效率, Libpcap 使用内核级过滤器 BPF, BPF 过滤代码从逻辑上看类似于汇编语言, 但它实际上是机器语言。Libpcap 提供一组高层的, 容易理解的过滤字符串供用户设置他们所需的过滤规则, 然后通过 `pcap_compile()` 函数将这些过滤规则其编译成 BPF 代码, 并通过 `pcap_setfilter()` 函数把 BPF 过滤器的代码安装到内核中。在内核中, BPF 过滤器就可以按照用户设置的过滤规则对所有数据包进行过滤操作了。对于通过了过滤的数据包, 会复制以后放入缓冲区, 而那些没有通过过滤规则的数据包被直接丢弃。Libpcap 在内核中使用两个缓冲区: 分别是存储缓冲区和保持缓冲区。存储缓冲区中会保存通过过滤器过滤规则的数据包, 而保持缓冲区则用来向用户缓冲区复制数据。当数据包通过过滤器后, 被不断的放入存储缓冲区, 同时用户程序调用函数不断的从保持缓存区中取数据。当存储缓冲区满并且保持缓冲区为空时, 两个缓冲区角色被交换, 重复上述过程。

但是如果存储缓冲区空间耗尽时, 而保持缓冲区中的数据包还没有被完全提交, 那么接下来流经网卡的数据包将不再被存入缓冲区中, 而是被直接丢弃, 直到保持缓冲区重新变成空。而且 Libpcap 在内核和应用层使用的缓冲区的大小都是固定的 32KB。因此, 这些特性决定了 Libpcap 在捕包效率上有缺陷, 在网络速度过高的环境中会造成很高的丢包率。

2.1.4 现有数据包捕获技术的改进技术

针对 Libpcap 捕包机制的缺陷, 国内外许多学者做了深入研究和改进, 提出了许多新技术, 如 NAPI, `pf-ring` 等, 分别对 Libpcap 即机制的各个环节进行优化和改进以便减少系统资源消耗, 同时提高 Libpcap 在大流量网络环境下的捕包性能。

以下对比较常用的技术进行介绍。

2.1.4.1 NAPI

NAPI 是 Jamal Hadi Salim、Robert Olsson 和 Alexey Kuznetsov 提出的对当前的 Linux 协议栈 softnet 进行改进的一种方法。其主要的思想是结合中断与轮询的优点，有效解决网络高负载情况下带来的拥塞冲突的问题。NAPI 技术主要用于路由器、防火墙等网络数据包转发设备。借鉴 NAPI 技术的思想，可以用于解决包采集平台中网络高负载情况下中断次数过于频繁的问题^[11-13]。

传统的数据包获取是由中断驱动的，由于中断的系统开销是昂贵的，当数据包到达速率过快的时候，系统会陷入不断的中断处理而无法跳出，因此产生中断活锁^[14]，从而导致报文吞吐率的急剧下降。若带宽占用率相同，当传输的数据包大小都很小的情况下，此时中断会相当频繁，系统的处理能力也会由此会降到最低。

既然传统的中断驱动方式存在中断活锁问题，有的系统采用完全轮询驱动的模式来提高报文接收能力，但是一旦数据包到达的频率远小于系统设定的轮询频率时，许多轮询操作就是毫无疑义的，因此在资源的利用上，这种完全的轮询方法也不是最优的。在低报文频率情况下，远大于报文频率的轮询操作加重 CPU 负载。相反，中断产生的消息处理开销影响系统的性能。单独使用轮询或中断都不是改善报文吞吐率的有效途径。一种比较好的实现方法就是采用半轮询驱动进行传输控制，在网络负载较低的情况下，报文到达时间是不可预测的，此时利用中断处理机制能够很好地避免由于报文到达的随机性而产生的延迟；而在网络负载较高时，报文的输入速率基本达到一个稳定状态，这时轮询机制就能够充分发挥作用，保证系统的高吞吐率。所以，如何准确的判断网络报文流量，并灵敏的进行模式转换就成了提高半轮询系统处理能力的关键。

2.1.4.2 “零拷贝”技术

在传统的数据捕获框架中，网络数据传输必须要经过操作系统并且进行了多次内存拷贝，内存拷贝操作要消耗大量的 CPU 周期和内存资源，从而严重地增加了系统的处理开销。为了提高大流量网络环境下数据捕获系统报文处理的性能，有必要减少报文传输过程的中间环节，绕过操作系统内核，减少或消除数据拷贝次数，降低系统有限资源的消耗，以便让更多的 CPU 和内存资源用于关键的入侵检测与分析过程。由此诞生了零拷贝技术。

零拷贝(zero-copy)^[15-19]的基本思想是:数据分组从网络设备到用户程序空间传递的过程中,减少数据拷贝次数和系统调用,实现 CPU 的零参与,彻底消除 CPU 在这方面的负担。实现零拷贝的最主要技术是 DMA 数据传输技术和内存区域映射技术。

传统的网络数据包处理,需要经过网络设备到操作系统内存空间,系统内存空间到用户应用程序空间这两次拷贝,同时在拷贝过程中用户还要向系统发出系统调用。而零拷贝技术则首先利用 DMA 技术将网络数据包直接存储到系统内核预先分配的地址空间中,避免了 CPU 的参与;同时将系统内核中存储数据包的内存区域映射到应用程序的存储空间,应用程序直接对这块内存进行访问,从而减少了系统内核向用户空间的内存拷贝和系统调用的开销,实现真正的“零拷贝”。

零拷贝中存在的最关键问题是同步问题,一边是处于内核空间的网卡驱动向缓存中写入网络数据包,一边是用户进程直接对缓存中的数据包进行分析,由于两者处于不同的空间,这使得同步问题变得更加复杂。

2.1.4.3 内存映射技术

内存映射技术^[20]要构造一个与内核空间共享的环形缓冲区。该缓冲区中存放的是 Linux 内核捕获的数据包在内存中的地址。内核每捕获一个数据包,就将该数据包存放的地址放在环形的共享缓冲区中。在 pcap_read 函数中,调用了一个叫做 pcap_ring_recv()的函数。该函数不断的检查共享的环形缓冲区中是否有数据,如果有数据,就根据这个地址取得内存中的数据包;如果没有数据,就会将自己阻塞,直到有数据包到达时才被唤醒,开始读取数据。这样,如果网络上有大量的数据包到达,内核就可以不断的向环形缓冲区中写数据,Libpcap 就可以不停的从环形的缓冲区中读取数据,数据包的读写并行工作,互不影响。因此系统的处理速度加快。而且,在这个过程中,不需要通过系统调用从内核中获取数据包,从而消除了系统调用的开销。因而,有效的提高了系统的性能。

2.1.4.4 TOE 技术

TOE 技术^[21-22]是将原存在于操作系统内核中的协议栈卸载到网络接口设备上。这里所说的卸载是指在网卡上增加专用的处理器,负责对网络数据包的处理。目前典型的例子是普通千兆网卡基本上都有校验和的处理功能。

与传统方式对比, TOE 引入了一种新的网络接口体系结构。它将 TCP/IP 协议栈的处理工作从服务器上卸载下来, 交给网卡来处理。在具有 TOE 的网卡上有专门的处理器或硬件来完成 TCP/IP 协议处理, 简化了整个 TCP/IP 的处理路径, 从而减轻了 CPU 和服务器 I/O 系统的处理负担, 消除了服务器的网络瓶颈。

TOE 技术对 TCP/IP 协议栈进行了软件扩展, 使部分 TCP/IP 功能调用从 CPU 转移到了网卡上集成的 TOE 硬件。TOE 一般由软硬两部分组件构成, 将传统的 TCP/IP 协议栈的功能进行延伸, 把网络数据流量的处理工作全部转到网卡上的集成硬件中进行, 服务器只承担 TCP/IP 控制信息的处理任务。提倡 TOE 技术的学者们认为它能增加服务器的网络吞吐量而又能同时减少 CPU 的利用率。Roland Westrelin 等人在一台 SMP 机器上开发了一个 TOE 模拟器, 经测试, 性能提高了 600% 到 900%。在某些应用场景, 特别是那些进行大块数据传输而连接数又少的应用场景, 如基于 IP 的数据存储领域, 使用 TOE 是能够增加应用的吞吐量和 CPU 利用率的。在网络数据包采集时, 如果条件允许, 我们可以利用 TOE 的思想, 在网卡上作简单的判别工作, 将那些在驱动一级就可识别的无用报文(如上层应用协议不关心的非 TCP 且非 UDP 报文, 无助于应用协议还原的无数据的 ack 报文)在驱动一级识别出来并进行抛弃。从而提升数据包采集的效率。

2.1.4.5 各改进技术对比

现总结各种技术优点和不足如下表所示:

表2-1 各种数据捕获技术优缺点

相关技术	优点	不足
NAPI	避免中断活锁, 减轻 CPU 负担	上层应用程序接受数据包实时性受影响。
零拷贝	减少系统开销, 提高数据包提交速度	绕过了协议栈, 须应用程序自身按需提供协议栈相关支持;需实现同步机制。
TOE	提高 CPU 利用率	需要特殊硬件支持。

2.1.5 被动数据捕获相关技术小结

网络流量的迅速增加和上层应用对数据包采集时事性和可靠性越来越高的要求使得数据包采集技术面对越来越多的挑战。本章总结了数据包采集的原理和目前研究现状, 对目前存在的提高数据包采集性能的各种方法进行了概述和总结。

各种不同的方法存在着各自的优缺点，也各自受到来自各方面环境的限制（如硬件设施），我们可以合理地综合采用并根据实际条件改进这些方法，达到提高数据包采集的效率，以满足上层应用越来越高的要求。

2.2 Linux 内核机制简介

本节主要研究本章重点介绍 Linux 内核机制，而与网络数据流采集相关的主要内核机制包括内核网络协议栈机制、内核模块化机制、内核内存管理机制、内核网卡驱动程序等。

Linux 内核的一个最基本概念是内核态和用户态：各种内核模块运行在内核态，而应用程序运行在用户态。目前所有 CPU 都具有不同的保护级别，X86 具有 4 个不同保护级别。但并不是所有运行于 X86 上的操作系统都会完全使用这些保护级别，比如 Linux 操作系统，它仅仅使用最高和最低两个级别。而在 Linux 当中，内核运行在最高级别，也就是内核态，在内核态中可以运行所有操作。而应用程序运行在最低级别，也就是用户态，在这用户态中，对硬件的直接访问以及对内存的非授权访问都由 CPU 控制着。我们通常将运行模式称作内核态和用户态 [23]。

2.2.1 内核网络协议栈简介

TCP/IP 协议栈是网络中广泛使用的事实网络通信标准。最初的 TCP 实现源自 4.4BSD lite，在 Linux 兴起后，也不可避免得支持它。但 Linux 的实现自成体系，仅与传统实现保持接口上的兼容。

对用户进程来说，我们可以调用 `send`，`sendto`，`sendmsg` 等函数来发送一段数据，或者可以使用文件系统上的 `write` 和 `writv` 来发送数据。同理，接收数据可以使用相应的 `recv`，`recvmsg`，`recvfrom`，也可以使用文件系统提供的 `read`，`readv` 来接收一段数据。对于接收来说，这是异步进行的，也就是说，这是中断驱动的。为简单起见，同时不失一般性，我们将分析 TCP 协议的输入输出全过程。Linux 操作系统的内核协议栈基本可以分为数据链路层、IP 层、TCP/UDP 层、Socket 层和应用层等几个部分 [24]。

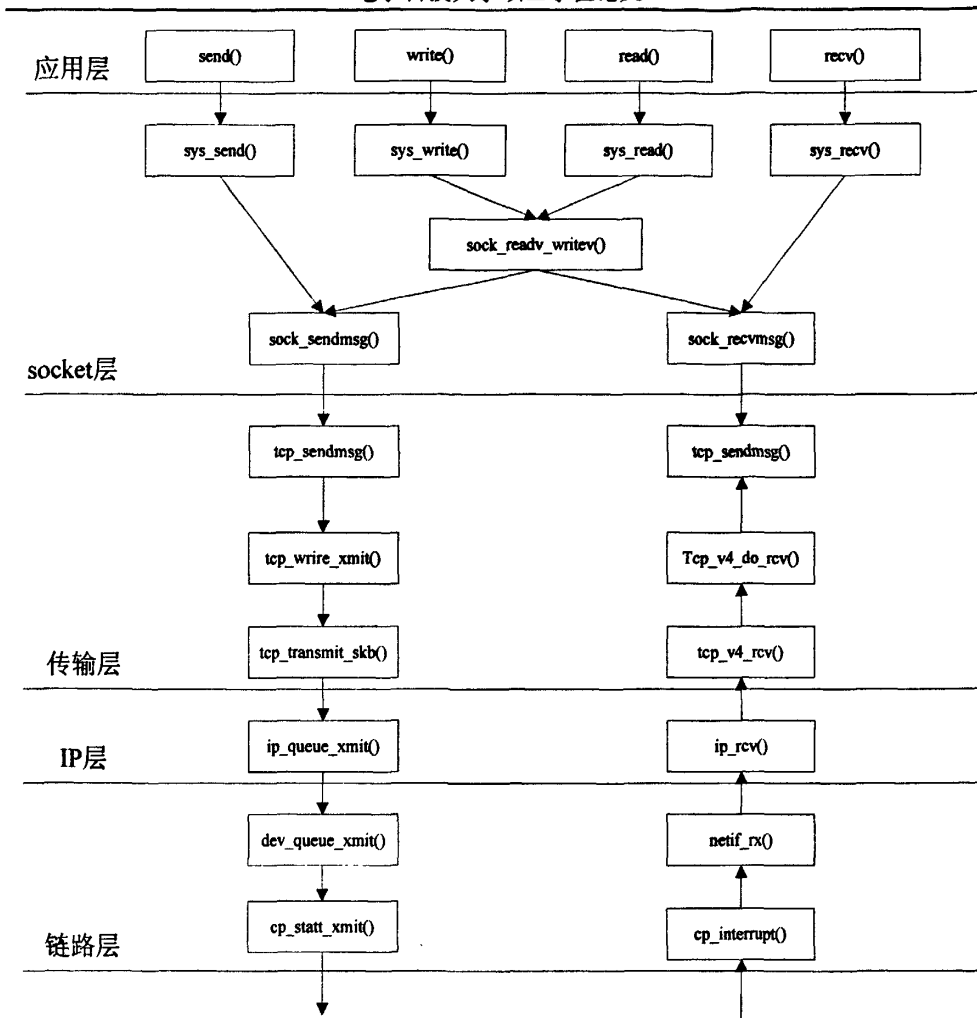


图 2-2 内核网络数据包收发流程图

在 Linux 上，socket 被实现为一个文件系统，这样可以通过 `vfs` 的 `write` 来调用，也可以直接使用 `send` 来调用，它们最终都是调用 `sock_sendmsg`。`sock_sendmsg` 通过它的内核版本 `_sock_sendmsg` 直接调用 `tcp_sendmsg` 来发送数据。在 `tcp_sendmsg` 中，同时完成数据复杂和数据校验，这样节省了一次遍历操作，这也是和 FreeBSD 不同的地方。Linux 使用 `skb` 结构来管理数据缓冲，这和 FreeBSD 的 `m_buf` 大同小异。当复制完数据后，使用 `tcp_push` 来进行下一步发送。`tcp_push` 通过 `_tcp_push_pending_frames` 来调用 `tcp_write_xmit` 将数据填入 `tcp` 的发送缓冲区。这里的填充仅是指针引用而已。下一步，`tcp_transmit_skb` 将数据放入 IP 的发送队列。`ip_queue_xmit` 函数完成 IP 包头的设置以及数据校验，并调用 `ip_output`

进入下一步发送。如果不用分片，将使用 `ip_finish_output` 继续发送。在这里，填充数据的以太网包头后调用 `dev_queue_xmit` 函数来进一步处理。`dev_queue_xmit` 函数将数据转移至网络核心层的待发送队列，调用具体的驱动程序 `cp_start_xmit` 来完成数据的最终发送。最后的 `cp_start_xmit` 做的事情和 `freebsd` 的相应函数差不多，检查数据，并复制进硬件缓冲。

当接收到一个数据包的时候，网卡会产生中断，这样网卡驱动的 `cp_interrupt` 会被调用。`cp_interrupt` 做的事情很少，只进行必要的检查后就返回了，更多的事情通过 `cp_rx_poll` 来完成，`cp_rx_poll` 在软中断中被调用，这样做是为了提高驱动的处理效率。`cp_rx_poll` 做的事情主要就是把申请并将数据复制进一个 `skb` 缓冲中。`netif_rx` 函数将数据从这个队列中转移至网络核心层队列中，`netif_receive_skb` 从这里接收数据，调用 `ip_rcv` 来处理。`ip_rc` 和 `ip_rcv_finish` 一起检查数据，得到包的路由，并调用相应的 `input` 函数来完成路由，在这里就是 `ip_local_deliver`，`ip_local_deliver` 完成 IP 包的重组后，使用 `ip_local_deliver_finish` 来进入 `tcp` 的处理流程，`tcp_v4_rcv` 完成数据校验以及一些简单的检查，主要的工作在 `tcp_v4_do_rcv` 中完成。`tcp_v4_do_rcv` 先判断是否正常的用户数据，如果是则用 `tcp_rcv_established` 处理，否则用 `tcp_rcv_state_process` 来更新连接的状态机。`tcp_rcv_established` 中同样有首部预测。如果一切顺便，将唤醒等待在 `tcp_recvmsg` 中的用户进程。后者将数据从 `skb` 缓冲中复制进用户进程缓冲，并进行逐级返回。

2.2.2 Linux 内核模块机制简介

Linux 内核包括设备驱动程序，有两种方法使设备驱动程序成为 Linux 内核的一部分。

(1) 将程序直接编译进内核，在 Linux 系统启动时，该程序会随着内核的启动而被自动加载；

(2) 将程序编译成一个模块，这个模块是可以通过命令动态添加进内核或从内核删除的。使用命令 `insmod` 加载，或用 `rmmmod` 删除该模块。这种方式可以更灵活的控制 Linux 内核的功能，也是较为广泛采用的方式。

2.2.2.1 内核模块的编写和编译

下面我们给出一个内核模块动态加载和卸载的例子：

```
#include <linux/module.h>
```

```

#include <linux/init.h> // init&exit
MODULE_LICENSE("GPL");
static int _init test_init (void)
{
    printk("test module init\n");
    return 0;
}
static void _exit test_exit (void)
{
    printk("test module exit\n");
}
module_init(test_init);
module_exit(test_exit);

```

通过该程序可以看出，一个内核模块至少需要包括两个函数，分别用于模块初始化和卸载。初始化函数在 `insmod` 时得到执行而卸载函数在 `rmmod` 时得到执行。

除此两个函数以外，还需要一个 `MAKEFILE` 文件来将负责编译，该 `MAKEFILE` 文件如下：

```

CC=/usr/local/arm/2.95.3/bin/arm-linux-gcc
KERNELDIR=/home/work/linux-2.4
CFLAGS=-D__KERNEL__ -DMODULE -I$(KERNELDIR)/include -O -Wall
test.o:test.c
    $(CC)$(CFLAGS) -c $<
clean:
    rm -rf *.o

```

由此可见，Linux 内核模块的编译需要给 `arm-linux-gcc` 指示 `-D__KERNEL__` `-DMODULE` 参数。 `-I` 选项跟着 Linux 内核源代码中 `Include` 目录的路径。

接下来可用 `insmod` 加载该模块：

```
insmod ./hello.o
```

或者用 `rmmod` 卸载该模块：

```
rmmod hello
```

2.2.2.2 内核模块与进程的关系

Linux 内核模块^[25] 作为一段代码存在与内核中，等待系统的调度。它不是以独立的进程方式来独占系统资源，只是在被调用的同时被加入到进程的相关结构中，成为调用它的进程的一部分，和该进程一起遵守内核中的各种管理机制。

2.2.3 网卡驱动程序简介

网卡驱动程序是网卡硬件和 Linux 内核中 TCP/IP 协议栈直接的接口。它的主要任务就是发送和接收数据包。而网卡驱动程序中包括各种对发送和接受数据包提供支持的函数，在系统需要完成发送和接受数据包的操作的时候，会调用网卡驱动程序提供的这些函数来完成这些动作。一般来说，网卡驱动程序会在 Linux 系统启动时随着内核的初始化而完成自己的初始化工作，但也可以在系统运行过程中通过动态加载的方式加载入内核。网卡驱动直接对硬件进行操作，除了发送和接收数据包外，它还关心中断处理和时钟同步等问题。负责提供对硬件的一些设置和操作。

2.2.3.1 网络驱动程序的基本方法

网络设备做为一个对象，提供一些方法供操作系统访问。正是这些有统一接口的方法，掩蔽了硬件的具体细节，让系统对各种网络设备的访问都采用统一的形式，做到硬件无关性^[26-27]。下面解释最基本的方法。

初始化(initialize)：驱动程序必须有一个初始化方法。在把驱动程序载入系统的时候会调用这个初始化程序。它做以下几方面的工作：检测设备。在初始化程序里你可以根据硬件的特征检查硬件是否存在，然后决定是否启动这个驱动程序。配置和初始化硬件。在初始化程序里你可以完成对硬件资源的配置，比如即插即用的硬件就可以在这个时候进行配置(Linux 内核对 PnP 功能没有很好的支持，可以在驱动程序里完成这个功能)。配置或协商好硬件占用的资源以后，就可以向系统申请这些资源。有些资源是可以和别的设备共享的，如中断。有些是不能共享的，如 IO、DMA。接下来你要初始化 device 结构中的变量。最后，你可以让硬件正式开始工作。

打开(open)：open 这个方法在网络设备驱动程序里是网络设备被激活的时候被调用(即设备状态由 down-->up)。所以实际上很多在 initialize 中的工作可以放到

这里来做。比如资源的申请，硬件的激活。如果 `dev->open` 返回非 0(error)，则硬件的状态还是 down。`open` 方法另一个作用是如果驱动程序做为一个模块被装入，则要防止模块卸载时设备处于打开状态。在 `open` 方法里要调用 `MOD_INC_USE_COUNT` 宏。

关闭(stop)：`close` 方法做和 `open` 相反的工作。可以释放某些资源以减少系统负担。`close` 是在设备状态由 up 转为 down 时被调用的。另外如果是做为模块装入的驱动程序，`close` 里应该调用 `MOD_DEC_USE_COUNT`，减少设备被引用的次数，以使驱动程序可以被卸载。另外 `close` 方法必须返回成功(0==success)。

发送(hard_start_xmit)：所有的网络设备驱动程序都必须有这个发送方法。在系统调用驱动程序的 `xmit` 时，发送的数据放在一个 `sk_buff` 结构中。一般的驱动程序把数据传给硬件发出去。也有一些特殊的设备比如 `loopback` 把数据组成一个接收数据再回送给系统，或者 `dummy` 设备直接丢弃数据。如果发送成功，`hard_start_xmit` 方法里释放 `sk_buff`，返回 0(发送成功)。如果设备暂时无法处理，比如硬件忙，则返回 1。这时如果 `dev->tbusy` 置为非 0，则系统认为硬件忙，要等到 `dev->tbusy` 置 0 以后才会再次发送。`tbusy` 的置 0 任务一般由中断完成。硬件在发送结束后产生中断，这时可以把 `tbusy` 置 0，然后用 `mark_bh()` 调用通知系统可以再次发送。在发送不成功的情况下，也可以不置 `dev->tbusy` 为非 0，这样系统会不断尝试重发。如果 `hard_start_xmit` 发送不成功，则不要释放 `sk_buff`。传送下来的 `sk_buff` 中的数据已经包含硬件需要的帧头。所以在发送方法里不需要再填充硬件帧头，数据可以直接提交给硬件发送。`sk_buff` 是被锁住的(locked)，确保其他程序不会存取它。

接收(reception)：驱动程序并不存在一个接收方法。有数据收到应该是驱动程序来通知系统的。一般设备收到数据后都会产生一个中断，在中断处理程序中驱动程序申请一块 `sk_buff(skb)`，从硬件读出数据放置到申请好的缓冲区里。接下来填充 `sk_buff` 中的一些信息。`skb->dev = dev`，判断收到帧的协议类型，填入 `skb->protocol`(多协议的支持)。把指针 `skb->mac.raw` 指向硬件数据然后丢弃硬件帧头(`skb_pull`)。还要设 `skb->pkt_type`，标明第二层(链路层)数据类型。最后调用 `netif_rx()` 把数据传送给协议层。`netif_rx()` 里数据放入处理队列然后返回，真正的处理是在中断返回以后，这样可以减少中断时间。调用 `netif_rx()` 以后，驱动程序就不能再存取数据缓冲区 `skb`。

2.2.4 Linux 内核内存相关机制简介

内存管理系统是操作系统中最为重要的部分，因为系统的物理内存总是少于系统所需要的内存数量。Linux 内存管理体制^[28]十分庞大，在此不作详述。下面仅介绍和内核编程相关的各内存管理相关系统支持。

2.2.4.1 内存申请和释放

`include/linux/kernel.h` 里声明了 `kmalloc()` 和 `kfree()`。用于在内核模式下申请和释放内存。

```
void *kmalloc(unsigned int len, int priority);
```

```
void kfree(void * __ptr);
```

与用户模式下的 `malloc()` 不同，`kmalloc()` 申请空间有大小限制。长度是 2 的整次方。可以申请的最大长度也有限制。另外 `kmalloc()` 有 `priority` 参数，通常使用时可以为 `GFP_KERNEL`，如果在中断里调用用 `GFP_ATOMIC` 参数，因为使用 `GFP_KERNEL` 则调用者可能进入 `sleep` 状态，在处理中断时是不允许的。

`kfree()` 释放的内存必须是 `kmalloc()` 申请的。如果知道内存的大小，也可以用 `kfree_s()` 释放。

2.2.4.2 物理页的分配与释放

在 `include/linux/gfp.h` 中定义了 `alloc_pages_node()` 函数，该函数用于申请连续物理页，并返回第一个 `PAGE` 结构指针。同样还有 `get_free_pages()` 可以用于申请物理页。

相应释放物理页的函数为 `free_pages()`。

2.2.4.3 物理页的分配与释放

自 Linux 2.4 起，内核开始提供对内存映射的支持，即通过映射的方式让用户态和内核态共享一片内存空间。提供的接口函数为 `mmap()`，该函数要求传入映射的长度，权限，映射内存所属的内核模块等参数，返回被映射内存对应的用户空间首地址，用户程序通过这个首地址来访问被映射的内存。

同样，Linux 也提供了 `munmap()` 函数用于撤销一段已被映射的地址。

2.2.5 LINUX 内核机制简介小结

网络数据流高速采集系统建立在 Linux 平台上，因此有必要对 Linux 内核机制有相应了解。本章讨论了 Linux 内核相关机制，特别针对数据捕获系统的需求，对内核中相关的网络协议栈，模块机制，网卡驱动程序即 Linux 内核相关机制等作了简要介绍。为下一步底层数据捕获子系统的设计作了铺垫，提供了理论依据。

第三章 网络数据流高速采集系统需求描述

本文设计了一个底层数据捕获子系统。本系统隶属于网络数据流高速采集系统。因此本章先介绍网络数据流高速采集系统，再对其中各模块进行需求描述。

3.1 网络数据流高速采集系统简介

3.1.1 网络数据流高速采集系统总体描述

网络数据流高速采集系统是定位于 Internet 主干网络的数据采集和数据分析设备，支持千兆级及以上的网络数据采集和数据分析。网络数据流高速采集系统使用自有网络接口串接在网络中或者通过旁路方式监听网络，采集网络上传输的所有数据报文；数据分析功能支持从链路层、IP 层、传输层，直到应用层的数据分析，通过协议还原等技术实现网络上传输数据的深层协议解析（DPI），为用户提供应用层数据监控、应用层数据分析等功能；同时网络数据流高速采集系统还提供数据存储功能，系统提供从链路层、IP 层、传输层到应用层的数据存储，用户可以方便地选择存储不同的数据，如果用户需要存储全部数据，可以简单的选择链路层数据存储，相反，如果用户仅仅对某一种应用的数据感兴趣，可以设定为仅仅保存某一类型应用数据；另外，网络速度越来越快，网络数据流高速采集系统还提供数据分发功能来解决网络设备处理能力不够的问题，提供从 IP 层直到应用层的数据分发，设置不同的分发策略可以达到基于 IP 分发，基于不用应用分发等不同的效果。

最后，网络数据流高速采集系统提供方便的二次开发接口，在此平台上可以方便的开发基于内容的网络安全设备，如 IPS 系统、DDOS 防御系统、垃圾邮件过滤系统等。

3.1.2 网络数据流高速采集系统功能描述

网络数据流高速采集系统主要提供如图 3-1 所示的四种功能：

- (1) 数据流采集：用于捕获网络上传输的所有数据报文。
- (2) 数据分析：用于捕获到的数据报文的链路层、IP 层、传输层、应用层

协议分析：在链路层、IP 层、传输层的主要功能为协议还原，在应用层的主要功能为协议分析，区别出不同的应用层协议，提供不同的应用层数据应用。

- (3) 数据存储：用于当用户需要对数据进行备份的时候，系统将捕获的数据报文按照用户需求保存到存储系统，以供用户后续使用。
- (4) 数据分发：用于当设备处理能力不够处理所有数据报文的时候，将不同的数据报文分发到不同的设备处理，保证系统捕获到所有需要的数据报文。

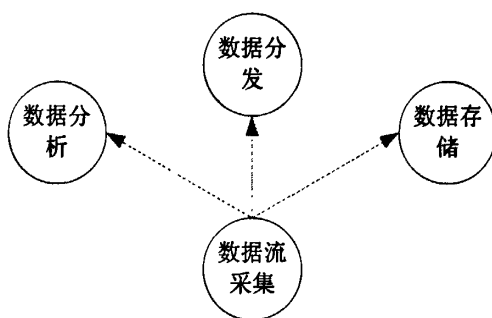


图 3-1 系统功能图

3.1.3 网络数据流高速采集系统需求描述

3.1.3.1 功能模块逻辑结构

网络数据流高速采集系统包括四大部分功能模块：数据捕获模块、数据分析模块、数据分发模块和数据存储模块。网络数据流高速采集系统的四大功能模块中，数据捕获位于最底层，其他三个功能在逻辑上位于数据捕获之上，三个功能之间的逻辑功能相互交错，如图 3-2 所示：

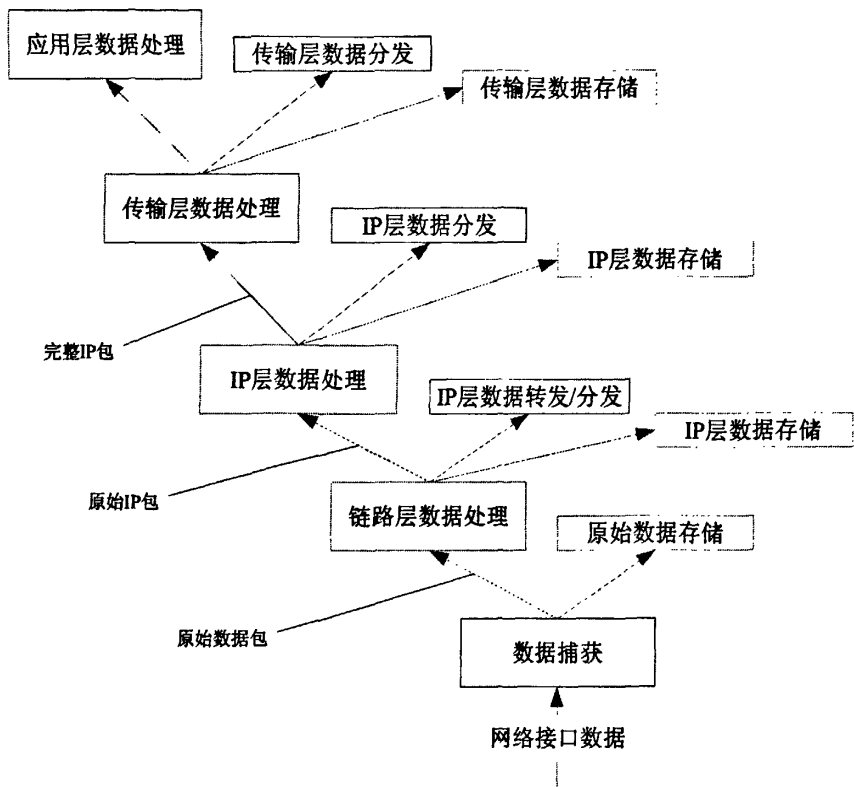


图 3-2 功能模块逻辑结构图

数据捕获模块位于最底层，负责数据报文的捕获；

数据分析模块包括 4 个部分：链路层数据处理、IP 层数据处理、传输层数据处理、应用层数据处理；其中每个部分的输出接口又与数据分发和数据存储相联系；以链路层为例说明，链路层数据处理完成后，输出数据报文为原始 IP 数据包，然后数据包有 3 中可选择的方式：提交到 IP 层处理、IP 层转发、IP 层数据存储；这 3 种方式可以同时选择，也可以选择其中的任意组合。IP 层数据处理、传输层数据处理、应用层数据处理的方式类似，只是基于不同的协议栈层次而已。

3.1.3.2 系统数据流

网络数据流高速采集系统的数据流如图 3-3 所示，系统基于数据报文驱动方式，网络接口设备（如以太网卡）接收到数据报文，由数据捕获模块获得数据，将数据报文放入原始数据包队列，再通知其上的模块：链路层数据处理模块和原始数据存储模块；链路层数据处理模块接收到数据报文后进行链路协议处理，根

据包类型通知不同的上层模块处理；原始数据存储模块接收到数据报文后将数据报文封装为标准格式，再存储到相应的存储设备；上层的各个模块处理方式与数据捕获模块类似，都是本模块处理完成后通知其上层的模块继续处理；各个模块的详细说明在本章后续部分给出。

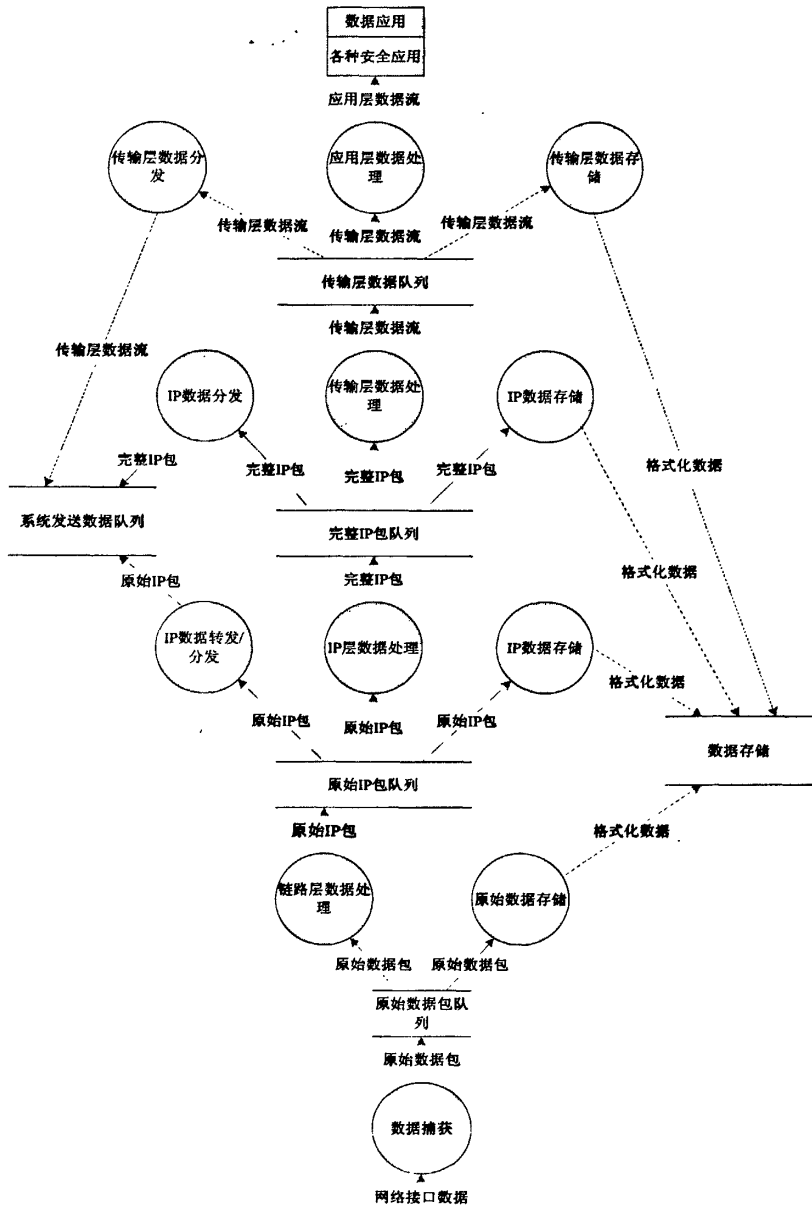


图 3-3 系统数据流图

3.1.3.3 系统描述

网络数据流高速采集系统要求的重点在于其通用性，所以系统的设计要突出其通用性的要求，参考图 3-2 所示的系统功能模块逻辑结构图，本系统设计了一种灵活的模块化结构，图中所示的所有模块中，数据捕获模块属于固定模块，其他的模块都是动态模块，可以根据用户不同的要求配置不同的模块组合，达到其灵活和通用性的目的；例如，如果我们仅仅想达到数据镜像并存储功能，则配置数据捕获和原始数据存储模块则可；如果我们仅仅想捕获数据分发设备，则配置数据捕获、链路层数据处理、IP 层数据转发/分发模块即可。模块的配置根据用于需求而定，基于协议栈层次提供的丰富接口可以满足用户多样化的需求。下面详细说明几种应用方式：

(1) 原始数据镜像应用

原始数据镜像就是将网络上传输的数据报文捕获，然后按照一定的格式存储到文件存储设备，数据文件可用于离线的数据分析应用。这种应用方式仅仅需要配置系统的两个模块就可以满足需求，如图 3-4 所示，配置数据捕获和原始数据存储模块，数据捕获模块捕获数据包，原始数据存储模块将原始数据包安装一定的格式封装后存储到文件存储设备即可。

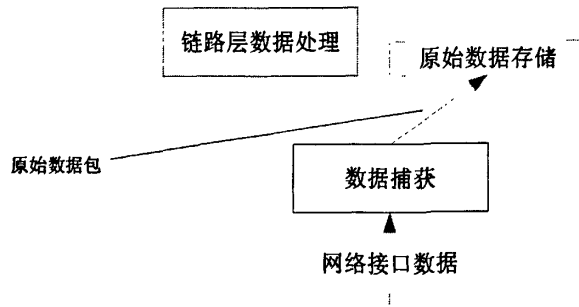


图 3-4 数据镜像功能模块图

(2) 负载均衡应用

简单的负载均衡设备可以基于 IP 数据包实现，将数据包分发到后续的多台设备分别处理。这种应用方式需要配置系统的三个模块就可以满足需求，如图 3-5 所示，配置数据捕获、链路层数据处理和 IP 层数据转发/分发三个模块，数据捕获模块捕获数据包，链路层处理用于处理链路层协议，输出原始 IP 数据包，IP 层数据转发/分发模块根据配置的分发策略就可以实现 IP 层的数据分发，达到负载

均衡设备的效果。

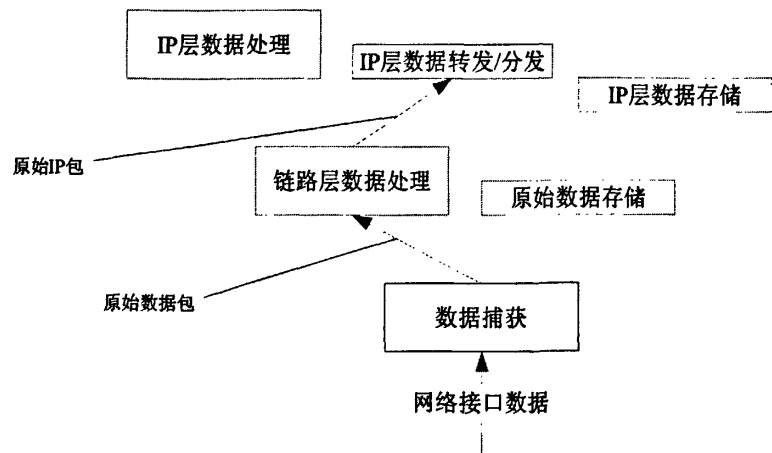


图 3-5 网络层负载均衡功能模块图

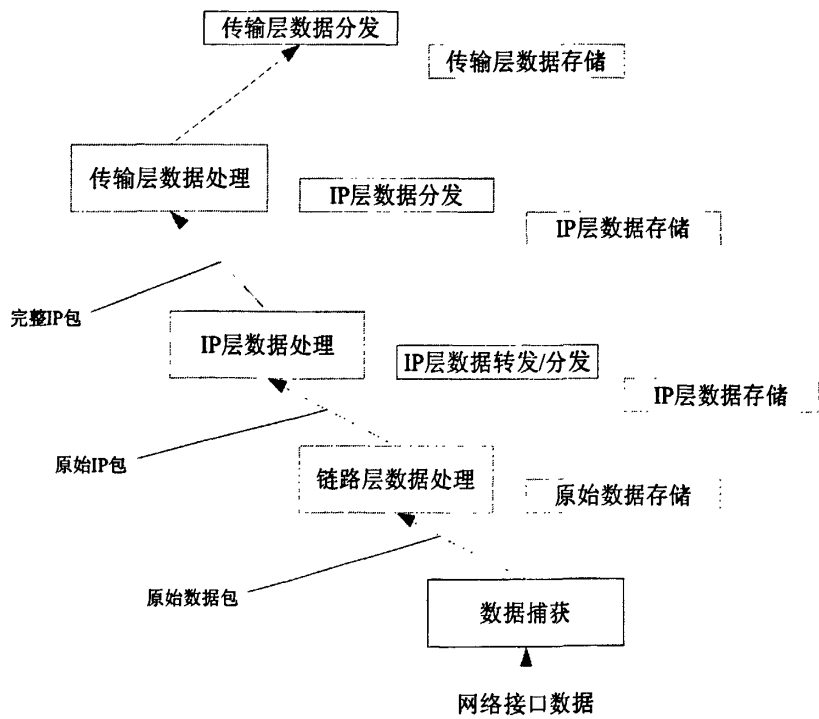


图 3-6 传输层负载均衡功能模块图

(3)垃圾邮件过滤应用

上面的两种应用都属于协议栈比较底层的应用，垃圾邮件过滤是基于应用层数据的应用，需要使用的模块比较多一些，如图 3-7 所示，配置的模块包括：数据捕获、链路层数据处理、IP 层数据处理、传输层数据处理、应用层数据处理。其中数据捕获中增加 BPF 过滤规则仅仅通过邮件数据报文，应用层数据处理将邮件数据分发到垃圾邮件分析程序，如果垃圾邮件分析程序判断为垃圾邮件，IP 层数据转发模块还提供接口动态配置转发规则，过滤掉垃圾邮件。

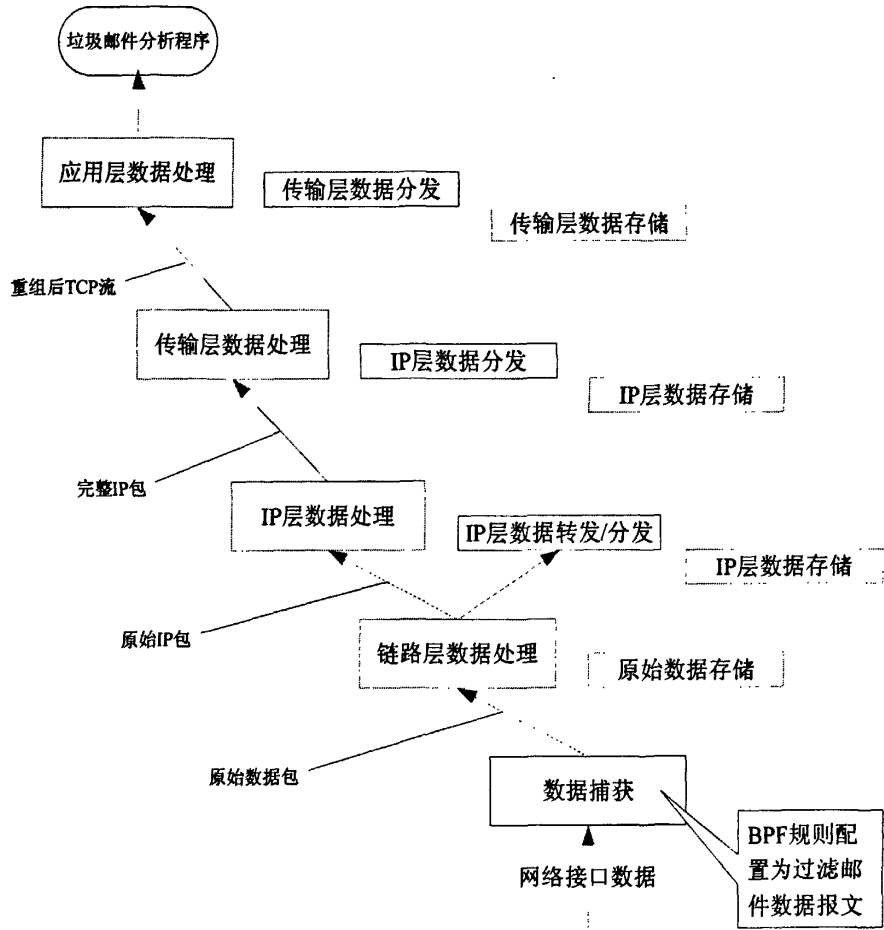


图 3-7 垃圾邮件过滤系统模块图

3.2 底层数据捕获子系统需求描述

3.2.1 底层数据捕获子系统总体描述

网络数据流高速采集系统首先需要对流经网络的数据包进行捕获。平台串联或者旁路在网络中，需具备捕获所有流经的网络数据流的能力，并能够做到实时捕获。平台从物理层捕获数据包，然后提交后续模块处理。另外，平台必须要能适应千兆网络的数据传输速度，因此系统对数据捕获模块的性能要求很高。

现有的数据包捕获多采用 Libpcap 机制。相对于传统内核协议栈的收包过程而言，Libpcap 绕过了传输层和 IP 层的处理过程，直接将数据包从数据链路层拷贝到应用程序缓冲空间中。这样能够节省数据包在接收过程中所消耗的时间。但是 Libpcap 捕包过程中，系统调用、数据拷贝和内核中断处理仍然是系统主要性能瓶颈。为提高 Libpcap 效率，增加了 BPF 内核过滤机制，判断接收的数据包是否是应用程序感兴趣的报文。如果是才将其复制到应用层缓冲区内，否则就丢弃。这样可以大大降低系统实际处理的报文数量。然而，该机制完成捕获工作需多次系统调用和内核与用户进程间的数据拷贝，不能满足千兆网络的要求。现有方案中也有采用一块映射到用户空间的环状缓存和 DMA 技术实现的高性能网络数据捕获技术，称为零拷贝技术。该技术能避免内核与用户进程间的数据拷贝，提高捕包效率。

底层数据捕获子系统属于网络数据流高速采集系统，它综合了多种技术，实现了对底层网络数据包的高速采集。

输入：由网络设备（如网卡）传来的网络原始数据报文。

加工：数据捕获模块从网络驱动程序处收集数据包，首先通过 BPF 过滤器过滤出感兴趣的数据包，并将数据包通过 DMA 技术保存到用户空间的原始数据包队列，交给后续模块处理。

输出：原始数据包队列。

外部接口：数据捕获模块需和网卡驱动程序交互，同时还提供上层功能模块（链路层数据处理和原始数据存储）注册配置功能，上层模块可选择注册，也可同时注册。

3.2.2 底层数据捕获子系统功能模块逻辑结构

整个底层数据捕获子系统分为三个子模块：网卡驱动程序模块。构建的内核

VCDM 模块、协议分析处理接口模块。VCDM 为 Virtual Capture Device Module 的缩写，意为虚拟捕获设备模块。

(1) 网卡驱动程序模块系统启动后自动加载运行，接受数据包。然后将数据包交给下一个模块处理；

(2) VCDM 模块的作用在于实现内核和应用程序共享内存空间。

(3) 上层协议分析处理接口模块有两个功能，一是提供上层应用程序和网卡无冲突的访问 VCDM 模块的机制，而是实现上层协议分析与处理子系统与底层数据捕获子系统的接口。

底层数据捕获子系统的模块逻辑结构如图 3-8 所示：

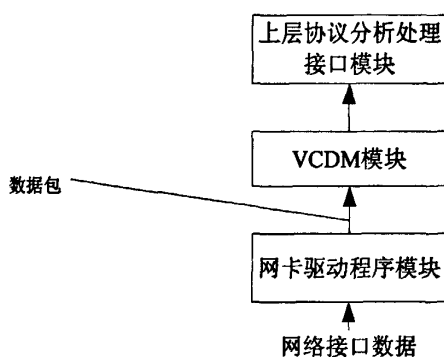


图 3-8 功能模块逻辑结构图

3.2.3 底层数据捕获子系统数据流

底层数据捕获子系统的数据流如图 3-9 所示，底层网卡获得网络数据包，经网卡驱动程序进入 Linux 系统内核。驱动程序将数据包交给 VCDM 模块，该模块实现了与用户空间的数据共享。最后数据包被协议分析接口模块从 VCDM 模块提取并交给位于用户空间的协议分析子系统。

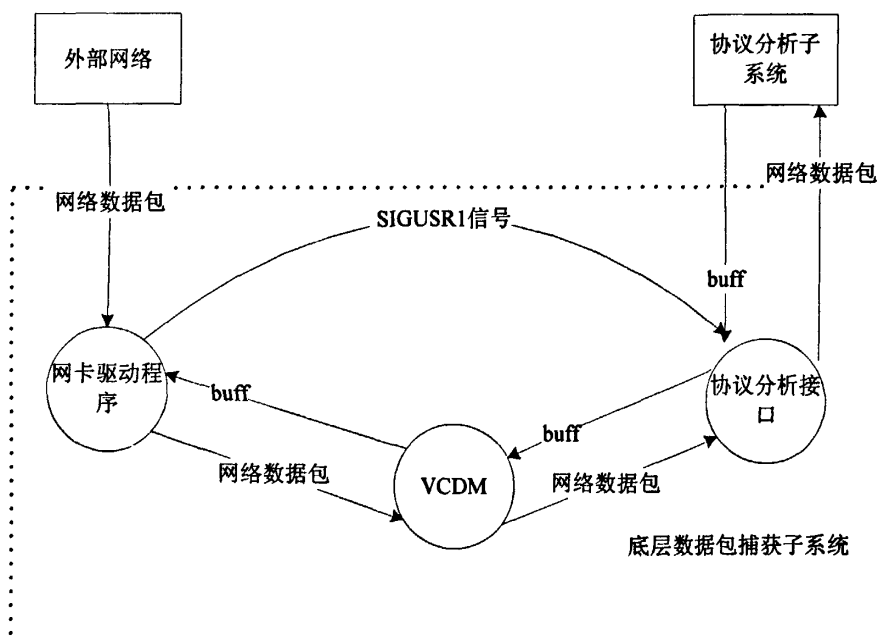


图 3-9 底层数据捕获子系统数据流图

3.2.3.1 修改后的网卡驱动模块数据流图

修改后的网卡驱动模块的作用在于处理底层硬件接受到的数据包，并使其绕过 linux 协议栈而将其转交给 vcdm 模块。数据流如图 3-10 所示：数据包由外部网络进入网卡，网卡以原始的中断方式或 NAPI 方式处理数据包，然后交给由 VCDM 提供的缓冲区 buf，同时向协议分析接口子模块发送 SIGUSER1 信号表示有数据包到达。

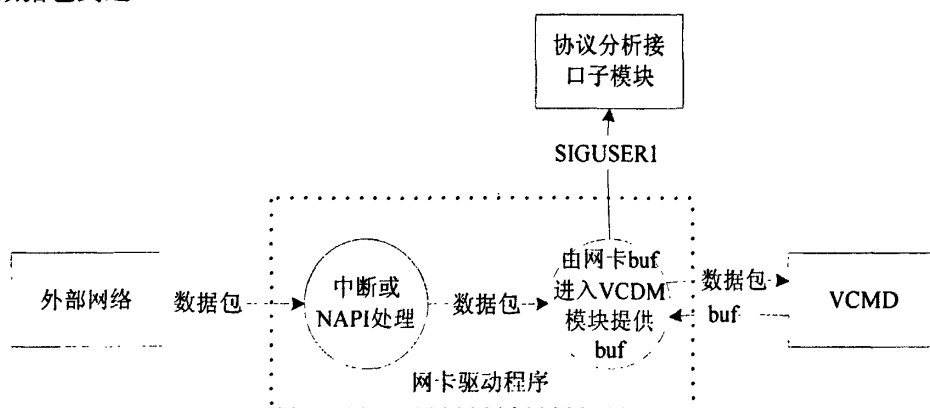


图 3-10 网卡驱动程序数据流图

3.2.3.2 VCDM 模块数据流图

VCDM 模块是本系统中实现底层数据捕获的一个重要组成部分, VCDM 模块能够实现内核态和用户态内存共享, 为绕过 TCP/IP 协议栈, 将网络数据包由内核数据空间高速地传递给上层应用程序提供平台。VCDM 模块数据流如图 3-11 所示: VCDM 模块获取数据包后将它挂入 Rx_busy 环, 然后发送给协议分析接口。协议分析接口处理完数据包后, 将存储数据包的 BUF 归还给 VCDM 模块的 Rx_free 环。

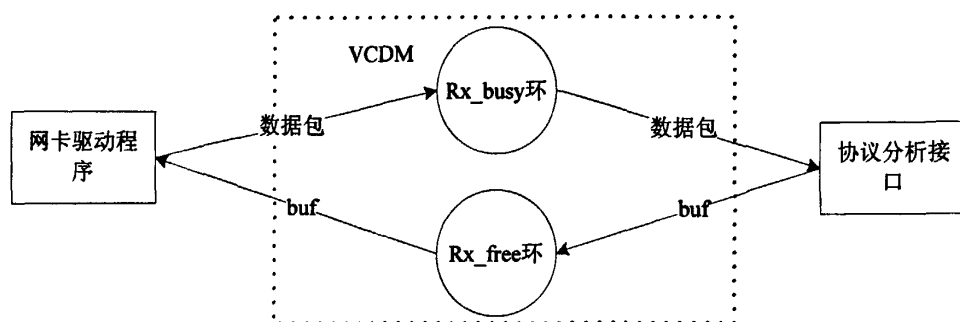


图 3-11 VCDM 数据流图

3.2.3.3 上层协议分析处理接口模块数据流图

上层协议分析处理接口模块根据 VCDM 模块所提供的内存映射平台, 实现数据包由网卡到协议分析子系统的机制, 也为协议分析子系统提供开发接口。其数据流图如 3-12 所示: 协议分析接口将数据包封装为 sk_buf 结构, 然后将该结构交由协议分析子系统提供的注册函数处理。处理完毕后, 协议分析子系统将存储数据包所用的 BUF 通过协议分析处理接口归还给 VCDM 模块。

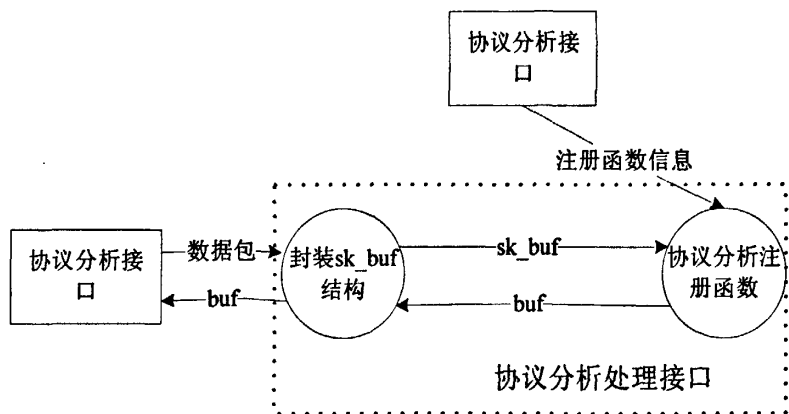


图 3-12 协议分析处理接口数据流程图

3.3 上层协议分析子系统简介

网络协议解析模块是通用数据采集系统的一个重要组成部分，包括 IP 层协议分析和传输层协议分析，由数据包捕获模块捕获到的数据包首先必须经过 IP 层协议分析处理，再根据协议类型再分发到相应的传输层协议分析模块。由于数据包直接从数据链路层捕获不经过内核中的网络协议栈，这就需要在应用程序空间建立新的网络数据报处理协议栈。而且，内核中的网络协议栈并不是为数据采集专门设计的，它在数据采集的应用过程中存在以下不足：冗余很多；协议栈长度不足，分析网络协议类型的能力有限；效率不高，如采用链表方式使得分片重组效率不高。根据数据采集系统特点，需要建立新的协议栈对网络数据包进行处理。实际的计算机网络存在多种协议分层模型，除 TCP/IP 协议族外，还存在 Novell 协议族，AppleTalk，DECnet，SNA 等协议族。IPS 可部署大型复杂的网络中，监控复杂的网络流量，会监测到属于不同协议族数据包，但由于 TCP/IP 使用的范围十分广泛，是事实上网络协议的标准，因此数据采集系统仅分析检测属于 TCP/IP 协议族的数据流。具体协议族如下表所示：

表3-1 TCP/IP协议族层次结构

应用层	SMTP	HTTP	FTP	DNS	SNMP
传输层	TCP			UDP	
网络层	IP, ARP, ICMP, RARP				
链路层	Ethernet	Token-Ring	PPP	其它	

协议分析模块首先将捕获到的数据包进行网络接口层协议的分析，识别出该数据包是否属于 TCP/IP 协议族，是则交由 TCP/IP 协议分析模块分析，否则直接转发数据包，并记录数据包信息。功能模块数据流图如图所示：

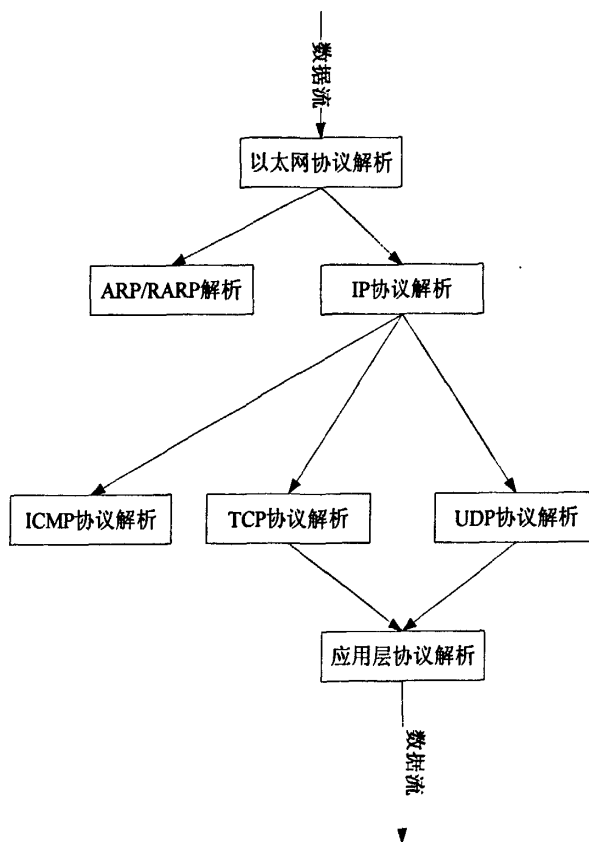


图 3-14 协议解析流程

TCP/IP 协议分析模块将模拟真实 TCP/IP 协议栈的功能，完成 PPP、ARP、RARP、IP、ICMP、IGMP、TCP、UDP 协议的解析。各协议解析具体内容属另一子系统，由另一同学在其论文中介绍，在此不作详述。

整个上层网络协议分析处理子系统除了用于协议分析的链路层协议分析模块，网络层协议分析模块，传输层协议分析模块，各应用层协议分析模块外，还包括各层的数据存储，网络层数据转发，传输层数据分发等子模块。这些模块将在后续工作中逐步完成。

3.4 小结

本章首先介绍了网络数据流高速采集系统需求，功能模块，数据流以及该系统在各种情形下的应用方式。然后介绍了底层数据捕获子系统在其中的作用，并介绍了底层数据捕获子系统主要功能需求，及其主要模块和详细数据流图。最后简要介绍了网络协议还原子系统。

第四章 底层数据速捕获子系统设计与实现

本章主要是对本人负责实现的几个模块：修改后网卡驱动模块、构建的内核 VCDM 模块、协议分析处理接口模块作详细介绍。

4.1 底层数据捕获模块总体设计与实现

作为高速数据采集平台的底层基础模块，高速数据包捕获模块负责向应用空间进程提供获取的数据包。本模块通过构建内核模块和修改网卡驱动程序，实现网络数据包从内核到用户空间的“零拷贝”。

本模块分三个子模块，经修改的网卡驱动程序，构建的内核模块，协议分析处理接口模块。通过这些模块的协同工作，实现数据包由网卡到用户程序的高速传递。

4.1.1 修改的网卡驱动程序

Linux 系统中，网卡驱动程序主要功能是将网卡收到的数据包提交给内核协议栈。协议栈的存在将减慢数据传输速度，且无法根据具体需求获取各协议层的原始数据。修改后的网卡驱动程序将使数据包绕过协议栈直接提交到 VCDM 模块中，供用户程序获取。也可根据实际需求在交给内核模块的同时让数据包按传统模式上交协议栈。本系统将在 RT8169 网卡驱动程序基础上作出修改，使之满足高速数据采集系统需求。

4.1.2 构建的内核模块 VCDM

VCDM 模块是高速数据捕获模块的核心部分。该模块从网卡接收数据包并实现用户程序对这些数据包的直接访问。

Linux 系统内核模块的设计使得直接向内核添加新功能成为可能。高速数据采集平台中通过向内核中注册 VCDM 模块实现内存映射，从而实现内核空间到用户空间的数据共享，大大提高数据包传输效率。

到来。

(3)协议分析处理模块收到信号，从 VCDM 模块中获得数据包。

(4)协议分析模块调用事先注册好的协议分析和处理函数处理数据包。

(5)数据包处理完毕，协议分析模块归还缓冲到 VCDM 模块。

具体可分为修改后的网卡驱动程序工作流程和协议分析处理程序工作流程，这两部分的详细内容将在 4.4 小节中阐述。

4.2 修改的网卡驱动程序设计与实现

本系统选择 RT8169 网卡驱动程序作为基础，通过分析其工作流程，并作出相应修改，改变该网卡驱动程序的工作流程，使其符合底层数据捕获子系统的需求，实现与 VCDM 模块与上层协议分析和还原模块的正常通信。

RT8169 网卡驱动程序是 Linux 自带的网卡驱动程序，支持千兆网络。我们在底层数据捕获子系统中选择 RT8169 网卡的主要原因是因为该网卡支持 NAPI 技术。Linux 内核自 2.4.23 起开始支持 NAPI，而早期网卡驱动程序并不提供对 NAPI 的支持。下面将以接收网络数据包为例，分析该网卡工作流程，并详细介绍我们对该网卡驱动程序的修改。

4.2.1 RT8169 网卡驱动程序流程

底层数据捕获子系统中，网卡对数据包的接收过程尤为重要，只有了解了该流程，才能针对我们自己的需要对网卡驱动程序作出正确的修改。下面分别描述接收数据包时 RT8169 网卡驱动程序工作流程：

NAPI 的流程：

- (1) 中断处理函数中直接调用 `_netif_rx_schedule`，添加该设备到 POLL 处理队列中，并开启软中断。
- (2) 软中断处理函数 `net_rx_action()`，该函数对每个设备进行配额，预算等计算后，调用 `dev->poll` 函数，进行轮询操作，本驱动中 `dev->poll` 实现为 `rtl8169_poll`。
- (3) `rtl8169_poll` 函数中调用的实际也是中断处理函数 `rtl8169_rx_interrupt`。
- (4) 在 `rtl8169_rx_interrupt` 中调用 `rtl8169_rx_sk`，该宏将调用 `netif_receive_skb` 函数完成最后的提交操作。

NON-NAPI 流程：

中断处理函数 `rtl8169_interrupt` 作一些必要处理后, 调用 `rtl8169_rx_interrupt`

(1) `rtl8169_rx_interrupt` 将包从网卡 DMA 到内存 BD 环, 再调用函数 `rtl8169_rx_skb`,

(2) `rtl8169_rx_skb` 将调用 `netif_rx` 添加到接收队列。接收队列中的数据包由软中断处理函数 `net_rx_action` 处理。

(3) `net_rx_action` 函数调用 `dev->poll`, 未定义 NAPI 模式下。DEV->POLL 为 `process_backlog()`。

(4) `Process_backlog()`将调用 `netif_receive_skb` 完成最后的提交操作。

4.2.2 对 RT8169 网卡驱动程序的修改

如 4.2.1 所述, RT8169 网卡实现了 NAPI 技术, 通过 4.2.1 的分析我们可以看到, 在该网卡驱动程序中, 不论是以中断方式提交网络数据包还是以中断加轮询的 NAPI 技术提交数据包, 该网卡驱动程序中对数据包的提交操作都是在中断处理函数 `rtl8169_rx_interrupt()`中实现的。该函数主要完成将 DMA 缓存中的数据帧以 `skb` 结构通过 `netif_rx` 或 `netif_receive_skb` 向上层提交。如果采用 NAPI 方式, 也在中断处理函数中进行处理, 虽然它不采用中断模式。这一特性就给我们修改网卡驱动程序提供了切入点。

我们只需要以我们提供的函数替换中断处理程序 `rtl8169_rx_interrupt()`中提交部分相应代码, 就可以达到我们的目的。

在该网卡驱动程序中具体添加的核心代码如下(略去原网卡驱动程序缓冲释放等代码)

```
if(vcdm_dev_state){
    vcdm_clean_buf_of_rxfreebd();
    vcdm_buf = vcdm_malloc_buf();
    if(vcdm_buf){
        memcpy(vcdm_buf->packet, skb->data, pkt_size);
        vcdm_insert_buf_to_rxbd(vcdm_buf, pkt_size);
    }else{
        printk(KERN_DEBUG"nsi vcdm: no empty buffer\n");
    }
}
```

```
}

```

该段代码先作一些必要的同步操作,然后将数据包交给 VCDM 其中最关键的函数为 `vcdm_insert_buf_to_rxbd()`,该函数用于将一个 `buf` 结构挂入 `Rx_bd_ring` 中。对网卡驱动程序的修改还包括归还由网卡驱动程序自身所分配的 BUF 空间,删除原驱动程序中向协议栈提交 BUF 等内容,在此不作详述。

其余函数及相关同步机制将在 4.4 中作详细说明。

4.3 构建的内核模块 VCDM 设计与实现

VCDM 设备是一个虚拟设备,由用户创建,以模块的形式动态加载到 Linux 内核中。该设备是本系统的关键,包括接收环 `rx_busy_ring`,空闲环 `rx_busy_ring`,发送环 `tx_ring`,以及用于存放数据包的缓冲区 `data_buf`。该模块主要作用在于实现在避免内核空间和到用户空间的数据拷贝的前提下,位于用户空间的应用程序与位于内核空间的网卡驱动程序间的数据传递。提供相关函数给用户程序和网卡驱动程序调用。

VCDM 模块需要解决以下三个问题:

- 1.位于用户态的应用程序能访问 `rx_busy_ring`, `rx_busy_ring` 和 `tx_ring` 三个环以及 `data_buf`。
- 2.让网卡能将数据直接传递给数据缓冲区 `data_buf`。
- 3.让网卡和上层协议还原子模块能够无冲突地通过 `busy_ring`, `free_ring` 访问 `data_buf`。

4.3.1 重要数据结构的设计

系统中的 VCDM 模块中的各种环结构和 BUF 都是最重要的数据结构,BUF 用于存放真正的数据包,而各种环结构则是实现“零拷贝”技术的关键,下面将依次介绍这些重要数据结构。

由于是在内核中,内存资源相对紧张,为了避免出现太多内存碎片,不能随便的频繁申请内存,所以本模块需要的内存都是事先申请好的。同时,Linux 的内存管理机制也为我们设计这些数据结构提供了参考。

基于上述一些考虑,本系统中的各重要数据结构被设计出来,其主要结构体如下:

接收环,空闲环和发送环:

```

typedef struct{
    unsigned int desc;    //描述符，是否已被用
    unsigned int length;  //挂在该描述符上的 buf 长度
    unsigned int paddr;   //挂在该描述符上的 buf 用户地址
    unsigned int kaddr;   //挂在该描述符上的 buf 内核地址
}vcdm_bd_t;
    Buf 结构:
typedef struct{
    char packet[1520];    //数据包内容
    char padded[516];    //填充字段，为保证该结构占用 2K 大小的内存
    unsigned int kaddr;  //该 buf 内核地址
    unsigned int paddr;  //该 buf 用户地址
    unsigned int phyaddr;//该 buf 物理地址(暂未使用)
}vcdm_buf_t;

```

环结构用于管理 BUF 缓冲区,实现用户空间程序和网卡可以无冲突的访问 buf 缓冲区。

Buf 结构是存放数据包真正的缓冲区,在内核中分配,并被设计为 2k 一个的结构。因为在 Linux 系统中,页大小是 4K,存放数据报文的数据块又要大于一个 MTU(1514 字节);这样每页只放 2 个数据报文,就保证了一个数据报文不会跨越两个页,而因此网卡进行 DMA 操作时每次只和一个内存页交互即可。从而进一步提高了系统效率。

VCDM 内存描述数据结构:

```

typedef struct{
    char name[VCDM_NAME_MAX]; //设备名
    int major; //设备号
    int minor;
    pid_t usripid; //用户进程的进程号

    void *original_buf; //vcdm 模块起始地址
    int rx_bd_num; //rx_busy 环中允许的最多描述符个数
    void *rx_ring; //rx_busy 环起始地址
    vcdm_bd_t *rx_bd_head; //rx_busy 环描述符结构首地址

```

```

int rx_bd_cur; //vcdm 所用的操作指针 rx_busy 环描述符结构
int rx_free_bd_num; //rx_free 环相关
void *rx_free_ring;
vcdm_bd_t *rx_free_bd_head;
int rx_free_bd_cur;

int tx_bd_num; //tx_busy 环相关
void *tx_ring;
vcdm_bd_t *tx_bd_head;
int tx_bd_cur;

int data_buf_num; //buf 缓冲区个数
void *data_buf; //buf 缓冲区头指针
struct list_head list_head; //linux 内核链表操作
struct mutex list_mutex;
int num_of_pages; //vcdm 模块所占内存页面总数
}vcdm_dev_t;

```

该数据结构用于描述 VCDM 模块中统一分配的内存，将该片内存细分为具体的各种数据结构。该结构中各项在 VCDM 模块初始化时将依次赋值。

因为在本系统中，VCDM 模块所分配的内存空间是和位于用户空间的应用程序以映射方式所共享的，所以在用户空间我们同样有类似的结构用于描述该片空间，具体结构如下：

```

typedef struct {
    void *original_buf;

    int rx_bd_num;
    void *rx_ring;
    vcdm_bd_t *rx_bd_head;
    int rx_bd_cur;

    int rx_free_bd_num;
    void *rx_free_ring;
}

```

```

vcdm_bd_t *rx_free_bd_head;
int rx_free_bd_cur;

int tx_bd_num;
void *tx_ring;
vcdm_bd_t *tx_bd_head;
int tx_bd_cur;

int data_buf_num;
void *data_buf;
vcdm_stat_t stats;
}vcdm_pdev_t;

```

可见，该结构与内核空间中所用的结构相当类似，都是用于细分所共享的内存空间。

4.3.2 VCDM 内存分配

在内核空间中，将 `rx_busy_ring`，`rx_free_ring`，`rx_ring`，以及用于存放其实数数据包的 `buf` 缓冲区放在一起，用 `vmalloc` 一起分配，并将该内存全部映射到用户态。模块中所需要的内存都被事先申请好，这样可以避免内存碎片的产生，但是也同时要求我们对这部分已经申请到的内存做一定的管理。结构结构内存安排如图所示。

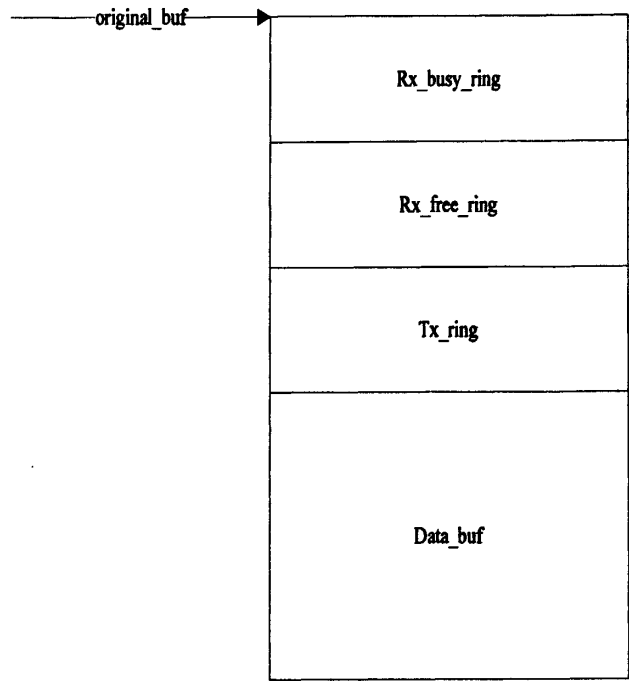


图 4-2 内存分配示意图

其中 rx_busy_ring, tx_ring, rx_free_ring 和 Data_buf 的大小均为 Linux 中页面大小的整数倍，我们一次性分配好这些内存后，在 VCDM 模块初始化过程中，buf 缓冲区被组织成链表结构，形成逻辑上和物理上都连续的链表，利于内存管理并提高模块效率。同时调用内核提供的 set_bit()和 atomic_inc()函数将分配的所有内存页锁住，使他们在系统运行过程中不被换出内存。然后准备开始以零拷贝的方式为上层应用程序提供网络数据包。本模块初始化函数为 vcdm_init()。初始化流程如下：

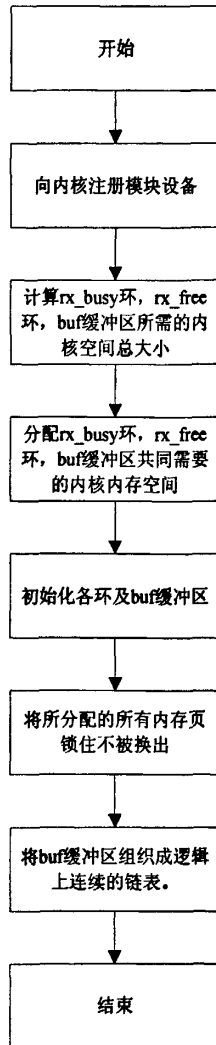


图 4-3 模块初始化流程图

4.3.3 网卡直接访问数据缓存 data_buf 的方法

在众多关于“零拷贝”的文献中多是提出在用户空间分配数据缓冲区，并利用在虚拟模块中加入专门为 UserBuff 定义的一个用户物理地址映射表 phy-addr-table。来解决网卡直接访问缓冲区的问题。该方法在初始化的时候由用户将 UserBuff 进行页对齐得到对齐后的地址，再通过系统调用将对齐后的首地址传到内核态。

VCDM 模块改进原有的数据缓冲区分配方式，改为在内核空间分配该块内存，并将该片内存锁住不让操作系统调度将其换出。同时再 buf 结构中增加一项

`unsigned int kaddr` 用于存放自己的内核空间首地址这样，同样位于内核空间的网卡驱动程序就可以直接访问该片内存，而避免了各文献中烦琐的查表和初始化操作。另外，为简化网卡对该片内存的操作，我们将该片内存中的结构组织为链表的形式。网卡只需在适当的时候从链表头取得一个节点即可获得存放数据包的缓冲区。

4.3.4 用户态访问 `busy_ring`, `free_ring`, `data_buf` 的方法

在本方案中，数据包被送到位于内核态的 `data_buf` 后，用户态的应用程序访问该片缓冲区拿走数据。网卡与应用程序之间的同步是通过修改 `busy_ring`，和 `free_ring` 中的相应元素来实现的。但是 Linux 操作系统中，由于保护模式的设置，位于用户态的应用程序是不能直接访问位于内核态的 `busy_ring`, `free_ring`, `data_buf` 等数据的。

为了解决这一问题，我们利用了 Linux 操作系统提供的一种称为内存映射的技术，它可以将数据文件直接映射到进程的地址空间，这种映射是一种直接映射而非拷贝的副本，那么，进程对这片地址空间的任何操作也就等同于对数据文件的操作。其接口函数为：`void *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)`。

通过该函数接口，我们将 VCDM 中的地址空间 `rx_busy_ring`, `rx_free_ring` 以数据缓存 `data_buf` 映射到用户空间。这样，用户空间的应用程序便可以直接访问 `busy_ring`, `free_ring`, `data_buf` 了。

当这片虚拟地址空间建立起来以后，根据 Linux 内存管理的页表机制，其最后一级页表所指向的地址是被初始化为空的。这意味着当我们试图在访问这片被映射的地址空间时系统会出现缺页异常。这就要求我们在 VCDM 模块内部提供一个函数，当访问这片被映射的地址空间产生缺页异常时，由我们来告诉系统在那里获得正确的页面。

为了实现这个功能，我们在 `vcdm` 模块的系统调用中定义了 `vcdm-mmap` 函数，当应用层对 `vcdm` 虚拟设备进行 `mmap` 时，会调用这个函数。这个函数新建一个 `vma`，同时将 `vma -> vma-start` 返回给应用层作为用户可以使用的虚拟地址。当用户程序访问这个虚拟地址空间中的数据，这个数据所在的页从未调入内存或调入后已被换出时，操作系统就会调用我们的函数 `vcdm_mm_nopage` 进行缺页处理。该函数以用户地址为参数，返回该地址所在的物理页。处理过程

如下:

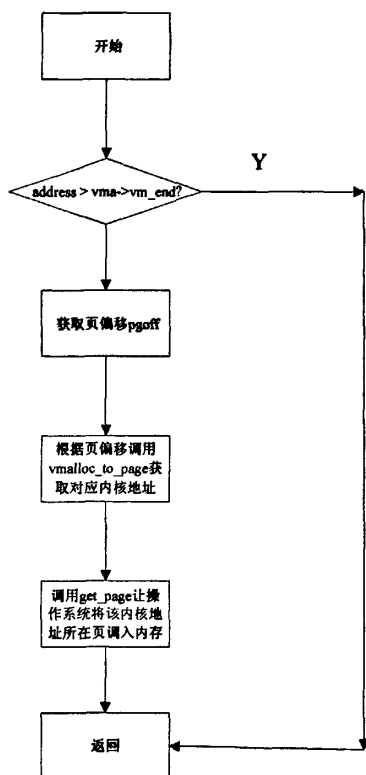


图 4-4 vcdm_mm_nopage 函数流程

4.4 协议分析处理接口设计与实现

协议分析处理接口即使为上层的协议分析处理模块提供的一套可以基于底层数据捕获子系统捕获的数据包作进一步处理的机制，同时，由于应用层读数据报文时要和网卡 DMA 传输时访问同一块缓存 data_buf。所以我们有必要采取一定措施，使得在读写内存同步的时候不影响系统性能。

4.4.1 协议分析处理开发接口设计与实现

在本系统中，我们以注册函数的方式给位于用户空间的协议分析处理子系统提供开发接口。根据协议分析子系统的需求，我们按照网络协议栈模型设计了 5 个链表，分别用于存放不同层次的注册函数。链表中用于表示一个回调函数的结构如下：

```
typedef struct mod_reg_node{
    struct list_head list;
    int protollevel;
    char modname[MAX_MOD_NAME_LEN];
    int (*callback) (struct sk_buff *skb);
}mod_reg_node_t;
```

其中 list 为 linux 提供的链表机制所需的数据结构, protollevel 用于表示该函数工作在每一层, modname 为该注册函数的模块名, callback 为对应的注册函数指针, 其参数为底层数据捕获系统提交的已被封装为 sk_buff 的网络数据包。注册函数组织情况如图 5-5 所示。

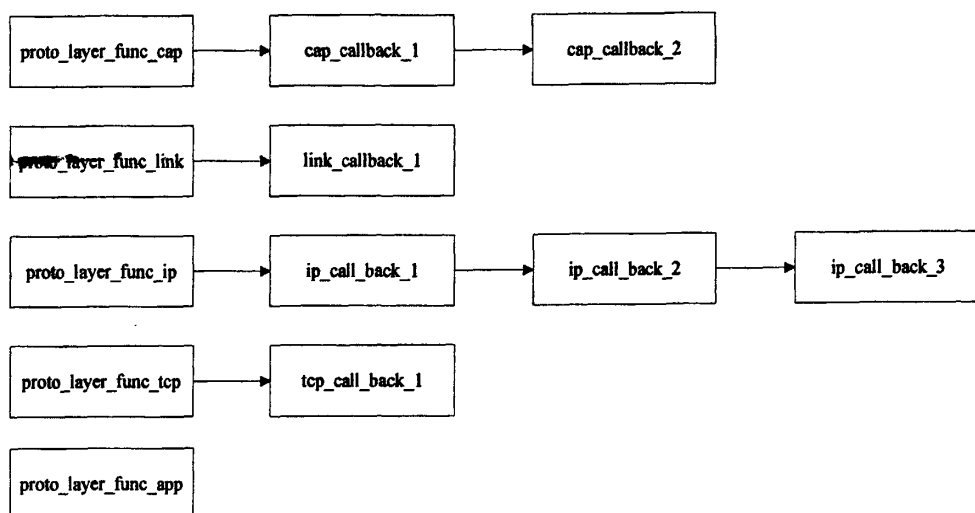


图 4-5 注册函数组织形式示意图

如图所示, 我们将该系统逻辑上分为 5 层, 分别为:

proto_layer_func_cap;proto_layer_func_link;proto_layer_func_ip;proto_layer_func_tcp;proto_layer_func_app;协议分析子系统只需按照自己的具体需求和实现, 在对应的层次上注册自己的处理函数即可。我们提供的用于注册的函数为:

```
int mod_register(char *modname, int (*callback) (struct sk_buff *skb), int
protollevel)
```

该函数根据传入的 `protollevel` 值，将函数 `callback` 加入对应逻辑层的链表中。系统运行过程中当有数据包到达后，会依次调用这 5 张链表中存在的注册函数，利用这些函数完成协议分析和处理。站在在协议分析处理子系统所处的用户的角度，则该开发接口的工作流程可以如图 4-6 所示。

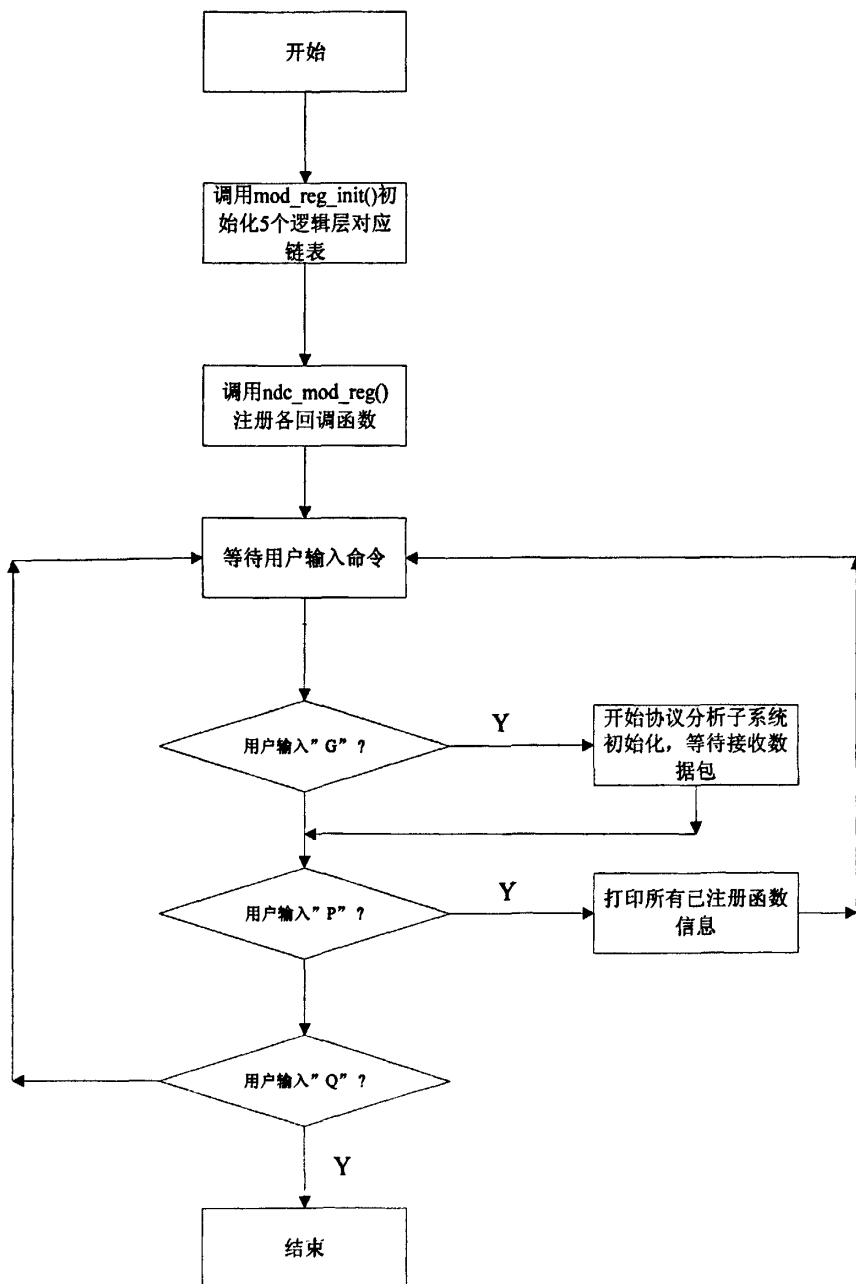


图 4-6 网络协议开发接口初始化流程

如图 4-6 所示,通过该开发接口,对上层的协议分析子系统而言,只需在初始化时调用函数 `ndc_mod_reg()`注册自己所需的各回调函数完成功能即可。

在讨论完协议分析处理子系统的开发接口后,还另一个问题:协议分析子系统和底层的网卡怎样做到无冲突的同步工作?在众多涉及到零拷贝技术的文开发献中多是提出采用对每个环加入读写指针的方式实现同步。在本系统中,我们采用更为简单的资源描述字段来实现这一功能。

我们在环结构中加入资源描述符,当环结构中挂上 `buff` 节点后,资源描述符置 1,当环结构上没挂节点时,资源描述符位置 0。位于内核态的网卡和用户态的应用程序分别维护独立的 `bd_cur`,作为指向下一个需要操作的环结构指针。网卡和用户空间的协议还原可通过判断资源描述符位来实现对环的同步操作。

下面我们分别详细描述需要无冲突访问 `VCDM` 模块的两个子模块的工作流程。

4.4.2 上层用户空间程序工作流程设计与实现

在本系统中,上层用户空间程序即协议分析处理子系统,该子系统主要由各可注册的回调函数实现其功能。我们通过在用户程序采用注册函数的形式处理 `VCDM` 模块提交的数据包。用户程序相关初始化工作完成后,将创建线程用于从 `VCDM` 模块接收数据包。上层用户空间程序总体工作流程如图 4-7 所示:

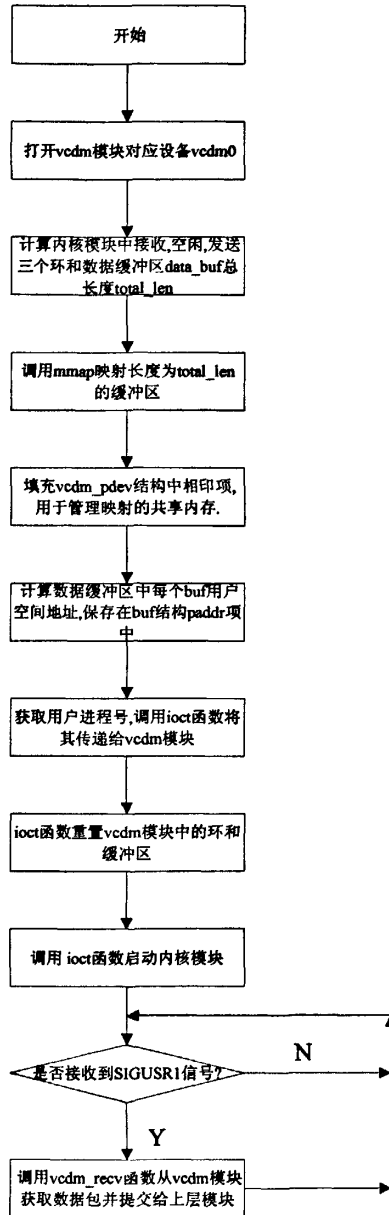


图 4-7 用户空间程序总体工作流程图

Vcdm_recv 函数是接口模块的关键函数，它负责从 rx_bd_ring 环中获得本次该处理的数据包缓存，并将其封装为 skb_buf 结构，提交给上层注册函数作进一步处理，如果发现 rx_bd_ring 环满，该函数直接返回，并等待用户程序下一次收到信号量后再次调用。我们总结用户空间程序获取数据包详细流程如下。

(1) 应用程序收到信号。

(2) 从 `rx_busy` 环中获取 `buf` 结构, 同时置环上对应被结构的描述结构中的描述符为未用

(3) 取出数据包将其封装为 `skb_buf` 结构, 将封装好的 `skb_buf` 结构交给上层注册函数处理。

(4) 处理完毕后将 `buf` 结构挂入 `rx_free` 环中。 同时置环上对应被结构的描述结构中的描述符为已用。

(5) 循环执行 2-5 步骤, 直到 `rx_buxy` 上无可用描述符。

`Vcdm_recv` 函数流程如图 4-8 所示:

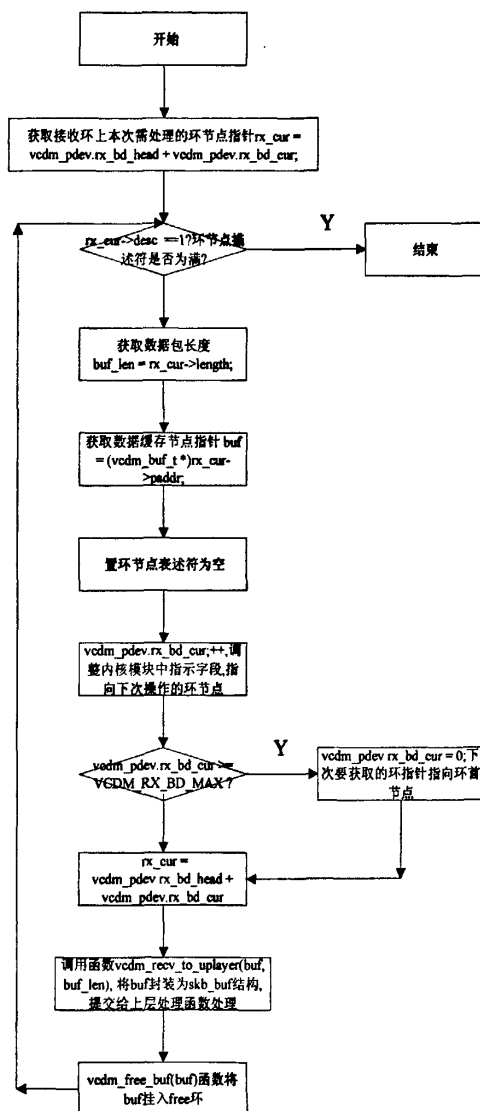


图 4-8 vcdm_recv 函数流程图

其中 vcdm_recv_to_uplayer() 函数内将实现 skb_buf 的封装和上层注册函数的依次调用。

4.4.3 修改后网卡驱动程序工作流程设计与实现

如 4.2 所述, 我们对 RT8169 网卡驱动程序作了相应修改, 使数据包可以绕过

协议栈直接提交给 VCDM 模块。为实现和上层用户空间程序之间对 VCDM 模块无冲突的访问，设计修改后的网卡驱动程序主要工作流程如下，流程如图 4-9 所示。

(1) 数据包到达后，不论是否采用 NAPI 方式，网卡均会发出硬中断，进入中断处理函数。

(2) 网卡驱动程序在中断处理函数中将 `rx_free` 环中的节点归还到 `data_buf` 缓冲区链表中。同时置环上对应被结构的描述结构中的描述符为未用。

(3) 获取 `data_buf` 缓冲区链表中一个结构，将数据包填充到该结构中。

(4) 调用 `vcdm_insert_buf_to_rxbd()` 函数将该结构挂入 `rx_busy` 环，同时置环上对应被结构的描述结构中的描述符为已用。

(5) 最后发信号通知应用程序有新数据包可以获取。

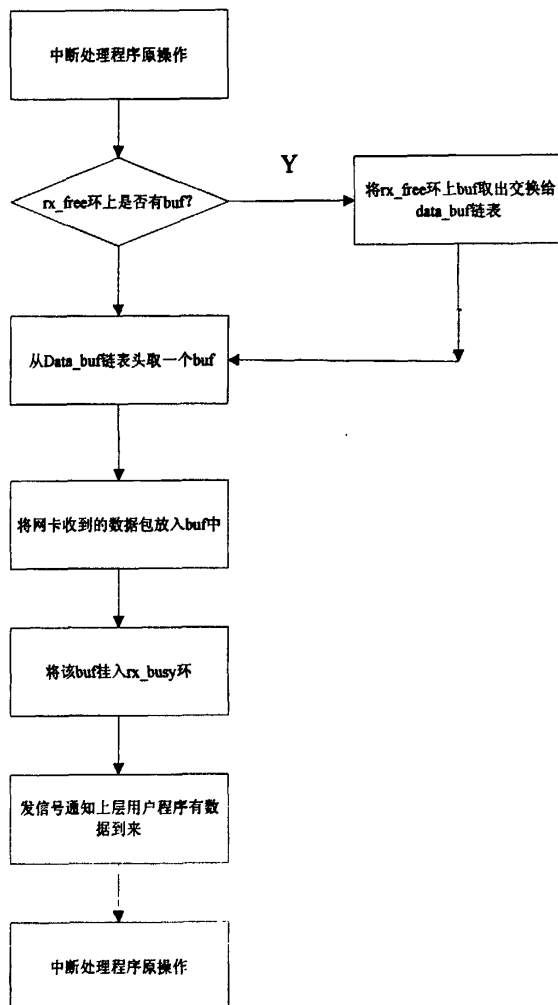


图 4-9 网卡驱动程序流接口流程

其中最关键的函数为 `vcdm_insert_buf_to_rxbd()`，该函数和上层用户空间程序所调用的函数 `vcdm_recv` 相对应，实现将一个 `buf` 结构挂入 `Rx_bd_ring` 中。

4.4.4 系统运行状态描述

前两小节所述的用户空间程序工作流程和网卡驱动程序工作流程保证了网卡和应用程序能够无冲突地通过 `Busy_ring`，`Free_ring` 访问 `data_buf`。在此我们给出系统在某个时间点上可能处于的状态如图 4-9 所示

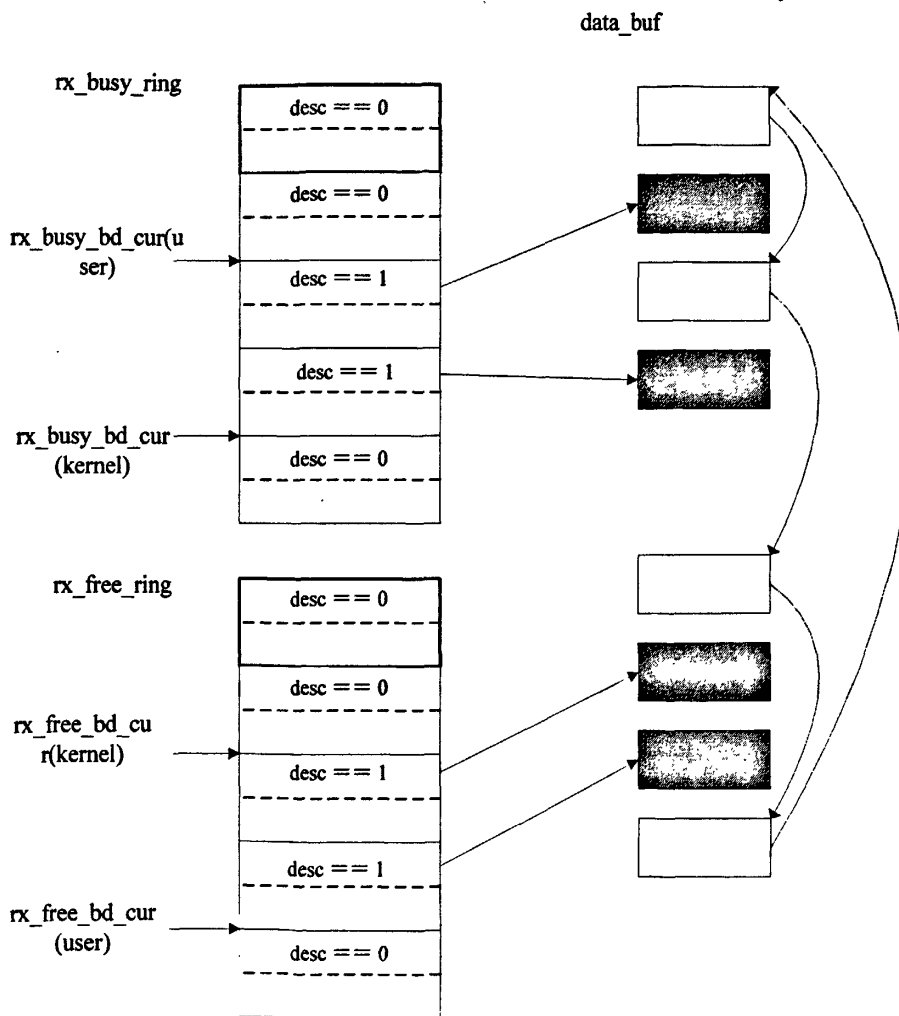


图 4-10 带资源描述字段的环结构和 `data_buf` 关系示意图

如图，`rx_busy_ring` 上结点中，`desc` 为 0 表示未用，`desc` 为 1 表示有 `buf` 挂在该结点上。`rx_busy_bd_cur(user)` 为用户进程访问该环的指针，用户程序用该指针获取存放数据包的缓冲区。`rx_busy_bd_cur(kernel)` 为内核访问该环的指针，

网卡驱动程序利用该指针获取空闲结点, 将存放数据包的 buf 挂入 busy 环供用户进程使用。Busy 环中, 正常情况下, kernel 指针总是在 user 指针前。

同理可分析 rx_free_ring 环, user 指针用于用户程序将使用完毕的 buf 归还给 free 环, kernel 指针稍有不同, 用于网卡驱动程序获取 buf, 但如前文所述, 网卡驱动程序是将其归还到 Data_buf 链表中, 并非直接使用。free 环中, 正常情况下, kernel 指针总是在 user 指针后。

4.5 底层数据捕获子系统中重要模块函数说明

除了前文所述的一些重要函数外, 我们还设计了其他一些函数, 共同实现底层数据捕获子系统的需求。现将其中较为重要的一些列表介绍如下:

函数原型: `int vcdm_init(void)`

功能: 初始化 vcdm 设备, 在内核中注册该模块, 同时分配三个 BD 环和数据缓冲 BUF, 并锁住这片内存不让系统换出, 并将数据缓冲 BUF 形成链表便于管理。

函数原型: `struct page *vcdm_mm_nopage (struct vm_area_struct * vma, unsigned long address, int *type)`

功能: 实现内存映射, 发生缺页时将系统传入的用户空间地址转化为内核空间物理页。

函数原型: `vcdm_buf_t *vcdm_malloc_buf(void)`

功能: 从数据缓冲区中获取一个 buf 结构。

函数原型: `int vcdm_free_buf(vcdm_buf_t *buf)`

功能: 归还一个 buf 结构到数据接收缓冲区链表中。

函数原型: `int vcdm_insert_buf_to_rxbd(vcdm_buf_t *buf, int length)`

功能: 将一个 buf 结构挂入 Rx_bd_ring 中

函数原型: `int vcdm_clean_buf_of_rxfreebd(void)`

功能: 将 Rx_bd_ring 环上的 buf 结构全部归还到数据缓冲链表中。

函数原型: `int vcdm_ioctl(struct inode *inode, struct file * file, unsigned int cmd, unsigned long arg)`

功能: 用于接收用户程序命令参数, 主要用于获取用户进程 ID 或重置环。

函数原型: `int vcdm_reset(void)`

功能: 将所有环清口, 所有 BUF 归还到缓冲区中。

函数原型: `void vcdm_exit(void)`

功能:模块卸载时自动调用, 释放占用的资源。

4.6 底层数据捕获子系统相关模块的编译, 加载和使用

启动 Linux 系统后, 我们首先要卸载原有的网卡驱动程序, 然后依次加载我们定制的网卡驱动程序和 VCDM 模块, 最后运行用户空间的程序。此时高速数据采集系统完整的运行起来。我们总结底层数据捕获子系统的相关模块编译和加载的方法如下:

4.6.1 VCDM 内核模块的编译和加载

可以通过如下步骤编译和加载 VCDM 模块:

首先, 需要有一份完整的, 编译了的内核源代码树。本系统所采用的 Linux 系统中, 内核源树所在目录为 `/usr/src/linux`。

接下来, 需要撰写一个 `makefile`。本模块所用的 `makefile` 文件名称为 `Makefile`, 内容如下:

```
obj-m := vcdm.o
```

然后使用以下命令编译该模块:

```
$make -C /usr/src/linux SUBDIRS=$PWD modules
```

编译完成后, 将得到名为 `vcdm.ko` 的模块文件

最后通过执行 `# insmod vcdm.ko` 将 `vcdm` 模块加载到内核空间中。

需要卸载时, 执行 `#rmmod vcdm` 可卸载该模块。

4.6.2 RT8169 网卡驱动程序的编译和加载

修改后的 `rt8169` 网卡驱动程序按如下方式编译和加载

在 `rt8169.c` 所在目录撰写一个 `makefile`。本模块所用的 `makefile` 文件名称为 `Makefile`, 内容如下:

```
obj-m := r8169.o
```

然后使用以下命令编译该模块:

```
$make -C /usr/src/linux SUBDIRS=$PWD modules
```

编译完成后, 将得到名为 `r8169.ko` 的模块文件

最后通过执行 `#rmmod 8169.ko` 卸载系统原有的网卡驱动程序模块, 在执行

`#insmod r8169.ko` 将定制的 r8169 模块加载到内核空间中。

需要卸载时，执行 `#rmmod` 可卸载该模块。

4.7 小结

本章为底层数据捕获子系统的具体描述，通过对该子系统的三个子模块的设计，能够高效，安全的捕获网络数据包，并提交给上层协议分析还原模块。

第五章 系统性能评估

为了验证底层数据捕获子系统，我们分别对该子系统进行功能和性能测试。

5.1 功能测试

本测试目的在于验证该系统在真实环境中是否能捕获底层数据包以及能否为上层协议分析子系统提供可用的开发接口。

(1) 测试网络结构

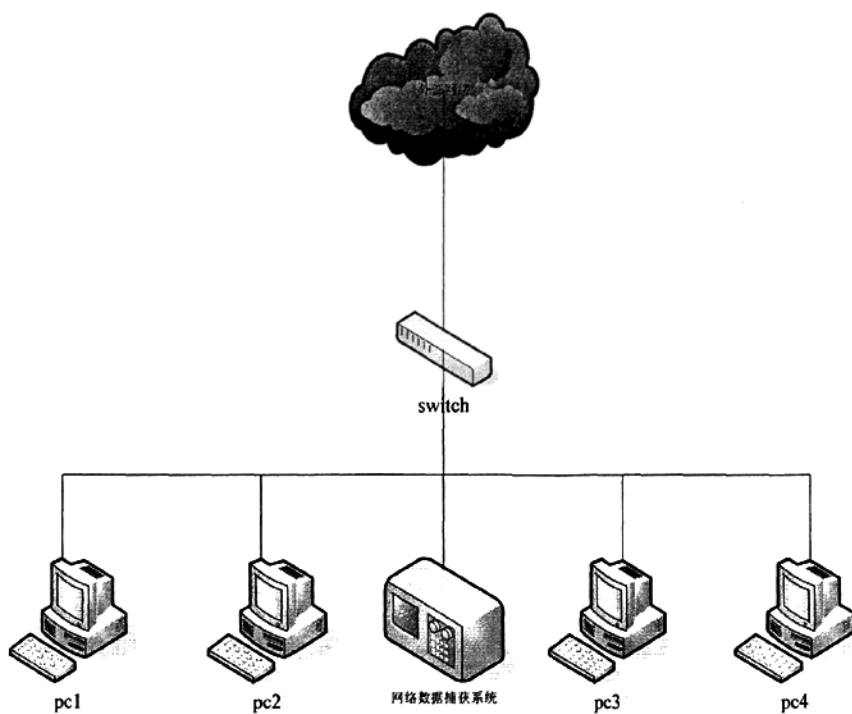


图 5-1 功能测试环境结构图

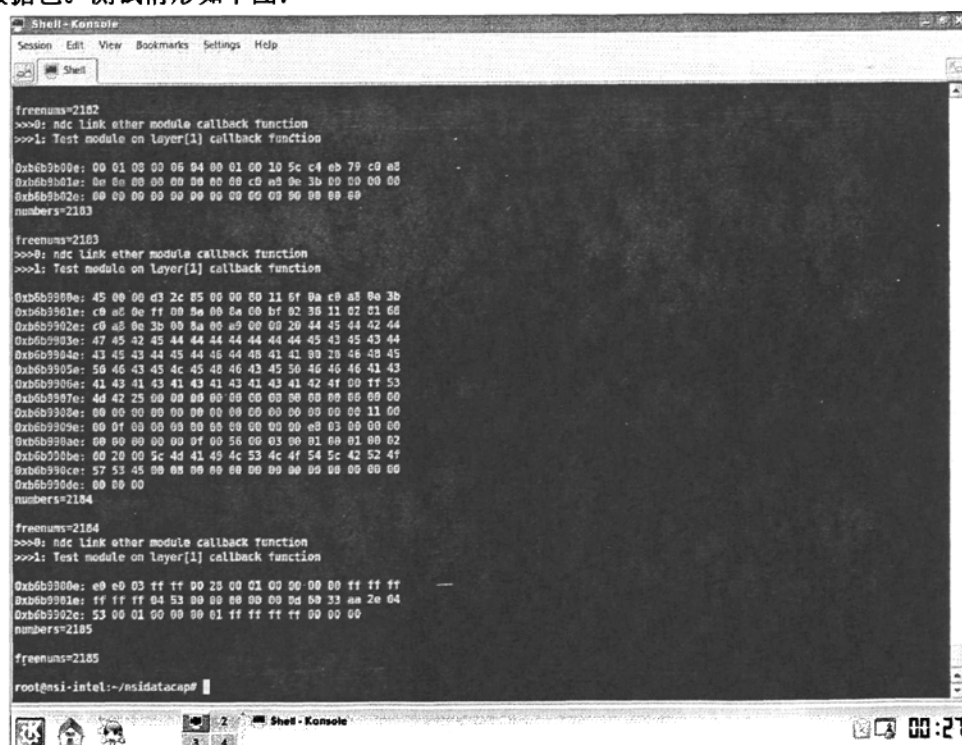
(2) 实验设备

表5-1 功能设备需求表

设备类型	作用	配置	数量	带宽能力
pci-pc4	访问外部网络， 产生网络流量	P-D 双核 3G, SDRAM 1G	4	1000M
网络数据捕获系统	捕获网络数据包	P4 双核 3G, SDRAM 2G	1	1000M
千兆位交换机	大流量数据交换	Cisco 3350 Series	1	1000M

(3) 测试方案:

我们利用交换机构建了一个内部网络,并将该网络接入电子科技大学校园网。我们将底层数据捕获子系统接入该交换机任一端口上,并在上层网络协议还原子系统中只注册一个函数,该函数用于显示底层提交的数据包内容并统计数据包数目。同时,我们用该内部网络中的 PC 访问外部网络,以产生流量。利用以太网内部数据广播特性,在正常情况下,我们可以通过该子系统捕获到流经该网络中的数据包。测试情形如下图:



```

Shell - Konsole
Session Edit View Bookmarks Settings Help

freemems=2182
>>>0: ndc link ether module callback function
>>>1: Test module on layer[1] callback function

0xb6b2b00e: 00 01 00 00 06 04 00 01 00 10 5c c4 e9 79 c9 a5
0xb6b2b01e: 0e 8e 00 00 00 00 00 c0 a3 9e 3b 00 00 00 00
0xb6b2b02e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
numbers=2183

freemems=2183
>>>0: ndc link ether module callback function
>>>1: Test module on layer[1] callback function

0xb6b2b00e: 45 00 00 c3 2c 05 00 00 00 11 5f 0a c9 a5 9e 3b
0xb6b2b01e: c0 a3 9e 3b 00 00 00 00 00 00 00 00 00 00 00
0xb6b2b02e: c0 a3 9e 3b 00 00 00 00 00 00 00 00 00 00 00
0xb6b2b03e: 47 45 42 45 44 44 44 44 44 44 45 43 45 43 44
0xb6b2b04e: 43 45 43 44 45 44 45 44 48 41 41 09 29 46 48 45
0xb6b2b05e: 56 46 43 45 4c 45 48 46 43 45 30 46 46 46 41 43
0xb6b2b06e: 41 43 41 43 41 43 41 43 41 43 41 42 4f 00 ff 53
0xb6b2b07e: 4d 42 25 00 00 00 00 00 00 00 00 00 00 00 00
0xb6b2b08e: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0xb6b2b09e: 00 0f 00 00 00 00 00 00 00 00 00 00 00 00 00
0xb6b2b0ae: 00 00 00 00 00 0f 00 56 00 03 00 01 00 01 00 02
0xb6b2b0be: 00 20 00 5c 44 41 45 4c 53 4c 4f 54 5c 42 52 4f
0xb6b2b0ce: 57 53 45 08 08 00 00 00 00 00 00 00 00 00 00
0xb6b2b0de: 00 00 00
numbers=2184

freemems=2184
>>>0: ndc link ether module callback function
>>>1: Test module on layer[1] callback function

0xb6b2b00e: e9 c9 03 ff ff 00 23 00 01 00 00 00 00 ff ff ff
0xb6b2b01e: ff ff ff 04 53 00 00 00 00 00 00 00 00 33 aa 2e 04
0xb6b2b02e: 53 00 01 00 00 00 01 ff ff ff ff 00 00 00
numbers=2185

freemems=2185
root@nsi-intel:~/nsidatcap#

```

图 5-2 功能测试效果图

可见,该子系统可以在真实环境中成功捕获长度不一的报文,上层协议分析

开发接口也能正常工作，为协议分析的开发提供支持。

5.2 性能测试

在成功验证该子系统能实现对底层数据包的捕获后，我们将验证该子系统的捕包效率。这也是该系统是否成功的关键。在该系统的性能评估中，我们将评估该子系统在以不丢包为前提的情况下，报文捕获最大速率。

(1) 测试网络结构

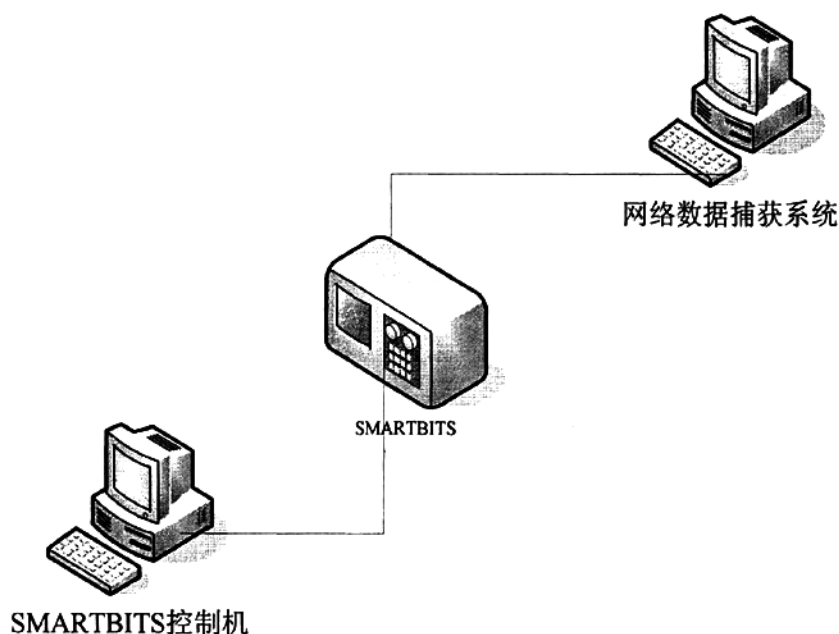


图 5-3 功能测试环境结构图

(2) 实验设备

表 5-2 功能设备需求表

设备类型	作用	配置	数量	带宽能力
SMARTBITS 控制机	控制管理 SMARTBITS	P4 双核 3G, SDRAM 1G	1	1000M
SMARTBITS	产生大流量数据	Sprint SmartBit	1	1000M
网络数据捕获系统	捕获数据包	P4 双核 3G, SDRAM 2G	1	1000M

(3) 测试方案:

在性能测试中，我们同样在上层网络协议还原子系统中只注册一个函数，该

函数用于显示底层提交的数据包内容并统计数据包数目，同时利用 SMARTBITS 直接向该子系统以不同的速率发送网络数据包，分别测试在不同包长下系统捕获数据包的能力，通过多次试验，力争找出该子系统在不发生丢包的前提下的最大速率。另外，我们以 REAL TP-LINK 8169 网卡自带驱动程序为基础，利用 Linux 自带的数据包捕获程序 TCPDUMP，在同样的环境下分析 Libpcap 的性能，以作参照。

5.2.1 各种报文长度下系统捕获性能测试

我们分别测试各种报文长度下系统的捕获性能，并作出归纳总结。定义如下性能指标：

临界发包间隔(ns):按一定时间间隔发送数据包，当两个相邻数据包的发送时间间隔到达该值时，系统会产生丢包现象。

最大速率(p/s):系统在不发生丢包的前提下，每秒能捕获的数据包最大个数目。

最大流量(bps):系统在不发生丢包的前提下，每秒能捕获的数据报文总长度。

5.2.1.1 包长 64B 时捕获性能测试

SMARTBITS 设置数据包包长为 64 BYTE，发送 10000 个数据包，通过多次重复试验，寻找丢包临界值。结果显示，在发包间隔在 1064ns 时，会出现丢包现象。可得出结论该情况下，系统最大收包速率临界值出现在为包间隔临界值为 1064ns 左右，计算可得报文捕获最大速率为 939000 p/s，折合最大流量约为 480200000bps。

5.2.1.2 包长 512B 时捕获性能测试

SMARTBITS 设置数据包包长为 512 BYTE，发送 10000 个数据包，通过多次重复试验，寻找丢包临界值。结果显示，在发包间隔在 4200ns 时，会出现丢包现象。可得出结论该情况下，系统最大收包速率临界值出现在为包间隔临界值为 4200ns 左右，计算可得报文捕获最大速率为 238000 p/s。折合最大流量约为 964800000bps。

5.2.1.3 包长 1500B 时捕获性能测试

SMARTBITS 设置数据包包长为 1500 BYTE，发送 10000 个数据包，通过多次

重复试验，寻找丢包临界值。结果显示，在发包间隔在 12000ns 时，会出现丢包现象。可得出结论该情况下，系统最大收包速率临界值出现在为包间隔临界值为 12000ns 左右，计算可得报文捕获最大速率为 83000 p/s。折合最大流量约为 960000000bps。

5.2.1.4 性能测试总结

通过上述测试我们可以总结出底层数据捕获子系统性能如下表：

表5-3 系统测试总结表

数据包长(B)	临界发包间隔(ns)	最大速率(p/s)	最大流量(bps)
64	1064 左右	939000 左右	480200000 左右
128	1075 左右	930000 左右	956300000 左右
512	4200 左右	238000 左右	964800000 左右
1500	12000 左右	83000 左右	960000000 左右

可见，系统在报文长度为 128B 的时候效率趋于稳定，接近网络千兆线速。

5.2.2 和 Libpcap 性能对比

我们同时给出数据包长为 64B, 512B, 1500B 下本子系统与 LIBPCAP 的性能对比如图所示：

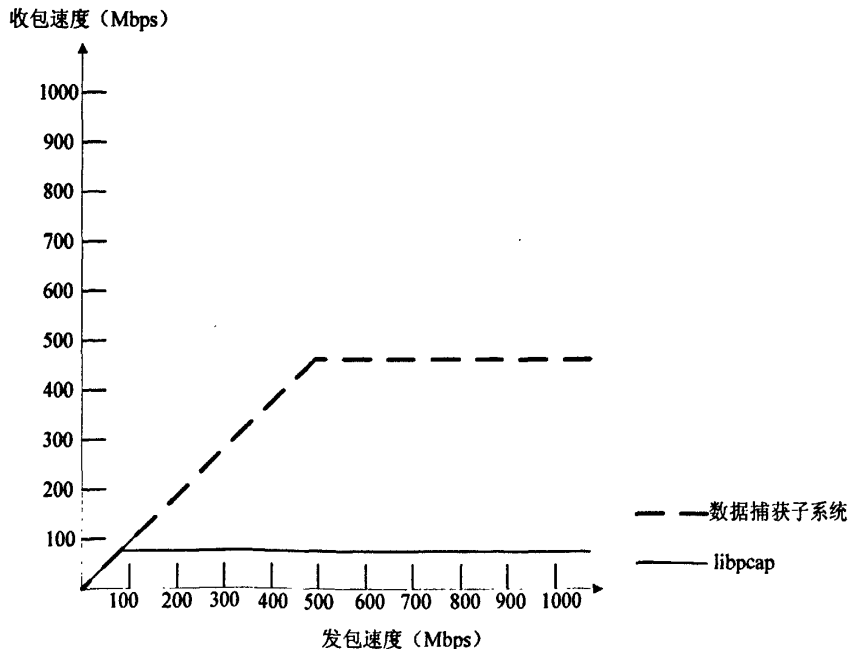


图 5-4 64B 包长时与 Libpcap 性能对比

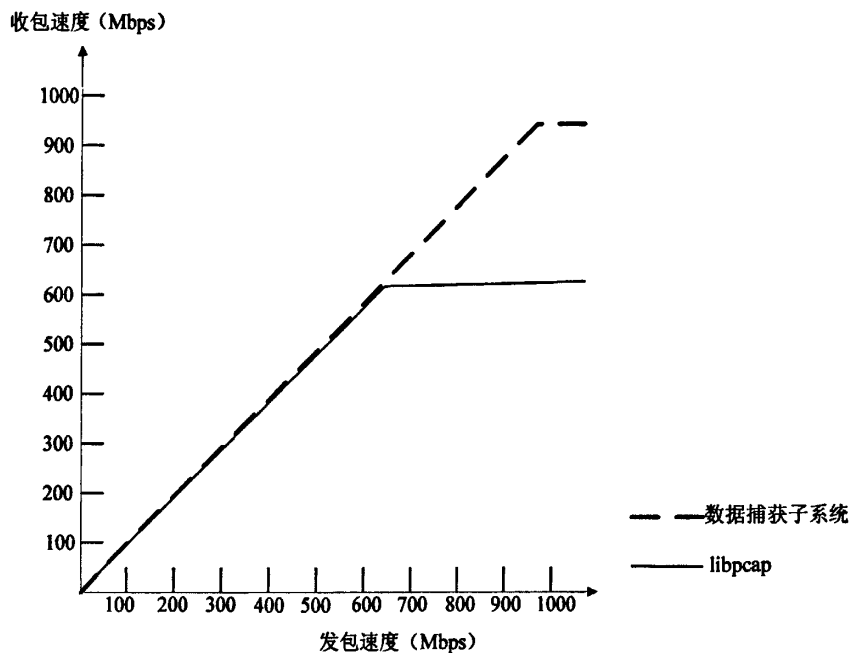


图 5-5 512B 包长时与 Libpcap 性能对比

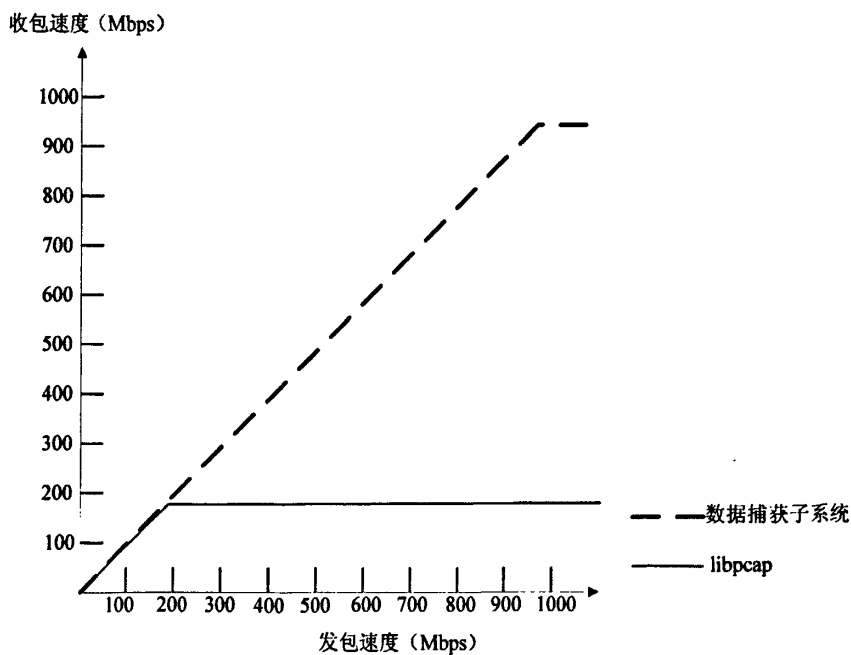


图 5-6 1500B 包长时与 Libpcap 性能对比

可以看出，针对不同长度的报文，数据捕获子系统的性能都要优于 Libpcap 系统。这种优势在报文长度较小（64B）或较大（1500B）时尤其明显。

5.3 测试小结

本章对底层数据捕获子系统作了功能测试和性能评估，经过连续测试，我们得出以下结论：

(1)该子系统能在真实环境中成功捕获网络数据包。

(2)该子系统能给上层协议分析子系统提供可用的开发接口。

(3)该子系统在各种状态下效率均高于 **Libpcap**，在报文长度较小或较大时，其优势尤为明显。

第六章 总结

网络信息采集一直是我们在信息安全领域面临的难题,而在网络信息采集中,底层数据包捕获是所有上层工作的基础,也是重点研究方向。本文研究了当前主要的数据捕获技术,并对它们进行了深入的研究,设计了一套底层数据包捕获子系统。本文的主要工作包括两部分。第一部分系统地研究数据捕获技术发展趋势、Linux 内核网络协议栈,内核模块机制和内存管理等相关理论。并重点对零拷贝技术进行了研究。第二部分根据研究的成果设计了一套模块化的底层数据包数据捕获系统。

本文的主要贡献有以下几个方面:

(1) 广泛研究了现有的被动数据捕获相关技术,重点分析了当前的基于 Linux 平台的各种数据捕获技术。

(2) 分析研究了 Linux 内核相关机制,包括网络协议栈,内核模块机制和内存管理机制,RT8169 网卡驱动程序,为下一步设计工作提供依据。

(3) 在系统中采用了内存映射技术,为零拷贝技术的实施打下基础。

(4) 在系统中实现了“零拷贝”技术,大大提高了数据包捕获的效率。

(5) 结合相关文献,改进“零拷贝”技术实现细节。将数据空间的分配放在内核中进行,避免了地址映射表的使用。设计了基于资源描述符的机制,实现了内核和用户空间内存访问的同步机制。

(6) 实现了在 PC 平台上对该数据捕获子系统的部署和运行。结果显示该子系统能有效捕获网络数据包。

在下一阶段的研究中,还需要注重以下几个方面对的研究:

(1) 系统性能有待提高。从硬件上来讲,目前我们采用 PC 机作为平台,系统性能远远没有达到最优。从底层机制上讲,数据包传输过程中系统成功避免了由内核到用户空间的数据拷贝,但在同属内核空间的网卡驱动到 VCDM 模块间的传递仍是通过拷贝方式进行的。因此,下一步工作的中点是将该子系统往嵌入式平台上移植,并进一步研究底层机制以提高数据包捕获的效率。

(2) 转发和存储功能尚未完成。本项目本阶段底层主要工作内容在于“零

拷贝”机制的实现和数据捕获这一需求的实现，上层主要工作在于协议处理和还原的实现。而项目规划中的数据转发和存储等相关模块以及底层对这些模块实现上的支持需要在后续工作中完成。

(3) 上层工作机制有待进一步研究。目前本系统中上层协议还原采用的是单线程模式，利用注册函数的方式实现协议还原和处理，该方式性能有待进一步提高。

致 谢

首先要感谢我的导师秦志光教授。在我的研究生学习阶段及论文写作阶段，秦志光教授给予了我很多指导和帮助，使我受益匪浅，同时，秦教授严谨的治学态度和让我深受感动，这些必将在今后的工作中不断激励着我。还要感谢教研室的周世杰老师，自本科以来周老师便在各项目中给予我指导，他还在百忙之中审阅了我的论文初稿，一次又一次的提出宝贵的意见，周老师一丝不苟的工作作风必将成为我今后做人的榜样。

感谢教研室其他各位老师，本课题的完成过程中得到了来自他们的很多帮助和指导，正是在他们的帮助下，我才能在过程中克服一个又一个的困难。感谢项目组的其他同学，我们一起完成了课题的研讨和实施，这种愉快的合作氛围令我终生难忘。

还要感谢我的家人对我的关心，让我没有后顾之忧的完成我的学业。

最后，衷心感谢为评阅本论文而付出辛勤劳动的专家们！

参考文献

- [1] 张志斌, 郭莉, 陈明宇, 方滨兴. 一种高效网络数据捕包平台的设计与实现. 计算机工程, 2005, 20(31):212-213
- [2] 梁理, 黄樟敏, 侯义斌, 网络信息侦听系统的研究与实现, 计算机工程与应用. 2002, 38(17): 184-186
- [3] 王磊. 基于 Linux 的千兆网络数据包捕捉技术的研究与实现:[山东大学硕士论文], 山东: 山东大学, 2004
- [4] 赵治国, 谭敏生. 基于 Linux 的网络监控技术的实现. 计算机与现代化. 2006, 8:105-109
- [5] 贺龙涛, 方滨兴, 云晓春. 网络监听与反监听[J]. 计算机工程与应用, 2001, 37(18): 20-21.
- [6] Bovet D P, Cesati M. Understanding the Linux Kernel[M]. USA: O'REILLY, 2001
- [7] 刘文涛. 网络安全开发包详解, 北京: 电子工业出版社, 2005
- [8] 刘文涛. Linux 网络入侵检测系统[M]. 北京: 电子工业出版社, 2004. 31-32.
- [9] 王景琪, 左明, 张功杰. BPF 数据包过滤器的分析与研究. 计算机工程与设计. 2005, 26(9):384-387
- [10] 刘格非, 裴昌幸, 朱畅华, 权东晓. 基于 Linux 的以太网数据包捕获方法. 通信与信息技术, 2006, 1(1):55-59
- [11] 李伟, 鲁士文. Snort 数据包捕获性能的分析与改进. 计算机应用与软件, 2005, 22(7):105-109
- [12] 田志宏, 方滨兴. RTLinux 下基于半轮询驱动的用户级报文传输机制. 软件学报, 2004, 15(6):834-840
- [13] 田志宏, 方滨兴, 张宏莉. 基于半轮询驱动的网络入侵检测单元的设计与实现. 通信学报, 2004, 25(7):146-152
- [14] J. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt driven kernel. Winter USENIX Conference, Jan. 1996.
- [15] 王佰玲, 方滨兴, 云晓春. 零拷贝报文捕获平台的研究与实现. 计算机学报, 2005, 28(1):46-51
- [16] 可向民, 龚正虎, 夏建东. 零拷贝技术及其实现的研究. 计算机工程与科学, 2000, 22(5):17-23

- [17] A.Bisvas, P Sinha. "A high Performance Packet capturing support for alarm Management systems" Proce. of the 17th LASTED International Conference on Parallel and distributed Computing and Systems, 2005
- [18] Tezuka H, O'Carroll F, Hori A, Ishikawa Y. Pin-Down cache: A virtual memory management technique for zero-copy communication. Proc.of the Int'l Parallel Processing Symp. 1998, 26(1):308-341
- [19] Walton S, Hutton A, Touch J. High-Speed Data Paths in Host-Based Routers[J]. IEEE Computer, 1998, (11): 46-52
- [20] 刘炜, 郑纬民. 底层通信协议中内存映射机制的设计与实现. 软件学报, 1999, 0(1):24-28
- [21] 朱宏峰, 刘天华, 刘杰. TCP/IP 协议卸载技术性能与实现的研究. 小型微型计算机系统. 2007, 28(4):609-612
- [22] Yatin Hoskote. A TCP offload accelerator for 10 Gb / s ethernet in 90nm CMOS[J]. IEEE Journal of Solid—State Circuits, 2003, 38(11): 1866-1874
- [23] Robert love. Linux Kernel Development second edition, 机械工业出版社, 2005
- [24] 倪继利. Linux 内核分析及编程. 北京: 电子工业出版社, 2005
- [25] 陈莉军. Linux 操作系统内核分析. 北京: 人民邮电出版社, 2000, 39-146
- [26] Rubin A, Corbet J. Linux Device Drivers[M] (2nd.) . O'REILLY, 2002-07
- [27] 张立东, 毕笃彦. 一种 Linux 下开发高效驱动程序的简便方法. 计算机工程, 2005, 31(4):88-90
- [28] 毛德操, 胡希明. Linux 内核源代码情景分析. 浙江大学出版社, 2001, 29-178

攻读硕士研究生期间的研究成果

一、参加的科研项目

- [1] 入侵防御系统与防拒绝服务产品。项目背景:国家信息产业部
- [2] 智能蜜网系统。项目背景:华为公司高校基金项目
- [3] 网络数据流高速采集系统。项目背景:电子科技大学网络安全实验预研项目

二、发表的论文

- [1] ***. 利用 LIBNIDS 实现蜜网系统中应用层数据捕获和还原的方法. 电子科技大学研究生学报计算机科学与技术增刊, 2009 年 3 月, 第 30 期.