**Linux下TUN设备抓包分析**

摘要： Linux作为不少开发人员相比于Windows操作系统来说更加青睐的操作系统，其内核中的扩展模块TUN/TAP模块是一种虚拟网络设备，是VPN不可或缺的一部分，其中TUN设备是虚拟的点对点传输设备；socket又称套接字，应用程序通常通过"套接字"向网络发出请求或者应答网络请求，通过socket编程可以实现多种网络协议报文的发送以及接受。本文注重于通过socket编程发送不同类型的报文，然后通过修改路由的方式利用预先建立好的TUN设备拦截报文（抓包）并对所获取的报文进行分析。

关键词 ：Linux操作系统、TUN、socket编程、抓包

**目录**

[第一章 绪论 3](#_Toc455659609)

[1.1 选题背景、意义 3](#_Toc455659610)

[1.2工作内容 3](#_Toc455659611)

[1.3章节安排 3](#_Toc455659612)

[第二章 技术基础 4](#_Toc455659613)

[2.1 TUN/TAP设备简介 4](#_Toc455659614)

[2.2 TUN/TAP设备的安装 5](#_Toc455659615)

[2.3 Linux下socket编程技术简介 6](#_Toc455659616)

[2.3.1 socket简介 6](#_Toc455659617)

[2.3.2 socket通信流程 6](#_Toc455659618)

[2.3.3 socket编程相关API 8](#_Toc455659619)

[2.4 TCP、UDP以及ICMP报文的结构 9](#_Toc455659620)

[2.4.1 TCP 9](#_Toc455659621)

[2.4.2 UDP 10](#_Toc455659622)

[2.4.3 ICMP 11](#_Toc455659623)

[第三章 需求分析 13](#_Toc455659624)

[3.1 需求 13](#_Toc455659625)

[3.2 分析 13](#_Toc455659626)

[3.3 流程图 13](#_Toc455659627)

[第四章 详细设计 14](#_Toc455659628)

[4.1 开发环境 14](#_Toc455659629)

[4.2 socket部分 15](#_Toc455659630)

[5.3 TUN部分 20](#_Toc455659631)

[第五章 软件测试 26](#_Toc455659632)

[第六章 总结与展望 27](#_Toc455659633)

[参考文献 28](#_Toc455659634)

# 

# 第一章 绪论

## 1.1 选题背景、意义

Linux下的TUN设备是实现VPN的重要部分，socket编程则是网络编程中最重要的一环；通过实现本题目，可以初步了解TUN设备相关的基础知识及其应用，并可以熟悉socket编程，对日后其他的网络编程有所帮助，通过对报文分析还可以加深对网络协议结构的理解。

## 1.2工作内容

设计一个小型的命令行程序，可以接受外部发送至tun设备的数据包并输出相关信息，完成基于tun设备的网络数据报文的抓取和分析，具体实现如下功能：

1. socket套接字编程，对外发送数据报文；
2. 通过修改路由，tun设备拦截数据包；
3. 根据不同类型的包的特点，在终端打印如下信息：IP来源、IP目的地址、数据包大小、协议名称。

## 1.3章节安排

第一章为绪论，主要讲述此项目的设计背景和要求；

第二章为技术基础，旨在介绍开发环节所用到的TUN设备的技术以及socket编程的原理和细节。

第三章为需求分析，分析了该软件的预期功能以及测试方法。

第四章为详细设计，利用流程图，带注释代码等介绍本程序设计实现的具体过程。

第五章为程序测试，会详细介绍此软件的实际应用效果。

第六章为总结与展望，主要介绍本次课程设计过程中的主要内容和过程，存在的问题以及解决办法。

# 第二章 技术基础

## 2.1 TUN/TAP设备简介

TUN/TAP，是Linux操作系统内核中的虚拟网络设备。

在计算机网络中，TUN与TAP，不同于普通靠硬件网路板卡实现的设备，这些虚拟的网络设备全部用软件实现，并向运行于操作系统上的软件提供与硬件的网络设备完全相同的功能。

TAP 等同于一个以太网设备，它操作第二层数据包如以太网数据帧。TUN模拟了网络层设备，操作第三层数据包比如IP数据封包。

操作系统通过TUN/TAP设备向绑定该设备的用户空间的程序发送数据，反之，用户空间的程序也可以像操作硬件网络设备那样，通过TUN/TAP设备发送数据。在后种情况下，TUN/TAP设备向操作系统的网络栈投递（或“注入”）数据包，从而模拟从外部接受数据的过程。

服务器如果拥有TUN/TAP模块，就可以开启VPN代理功能。

虚拟网卡TUN/TAP 驱动程序设计原理如图2-1：

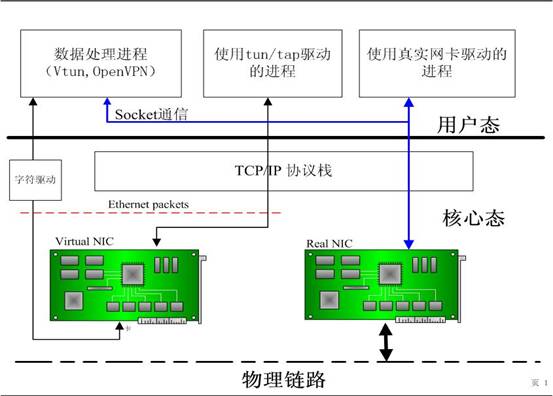


图2-1 虚拟网卡TUN/TAP 驱动程序设计原理

TUN设备的应用主要集中于虚拟专用网络（Virtual Private Network）服务上，有两个相关的开源项目（vTun和openVPN），可以实现加密通讯。

## 2.2 TUN/TAP设备的安装

由于不是所有的Linux发行版都有编译好的TUN/TAP设备模块，故在此简介如何在一台可工作的Linux上安装TUN/TAP设备。

第一步：使用modinfo命令确认TUN设备是否安装；

第二步：获取最新的linux源码（Ubuntu用户可以使用apt-get，arch用户可以使用pacman命令）

第三步：解压并进入安装菜单并修改相关项（使用make menuconfig命令），大致步骤见图2-2和图2-3：

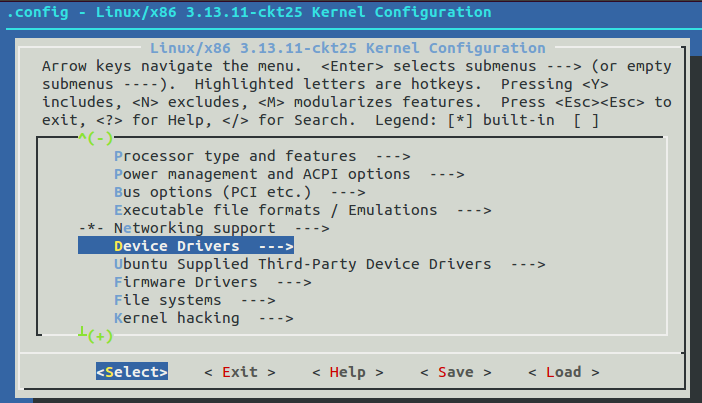


图2-2 编译内核的菜单（1）

选中Universal TUN/TAP device driver support项后输入‘M’，表示将其编译成模块；

第四步：使用make modules命令编译所有的模块驱动；

第五步：将编译好的内核模块加入内核，重启；

第六步：使用modprobe命令加载模块。

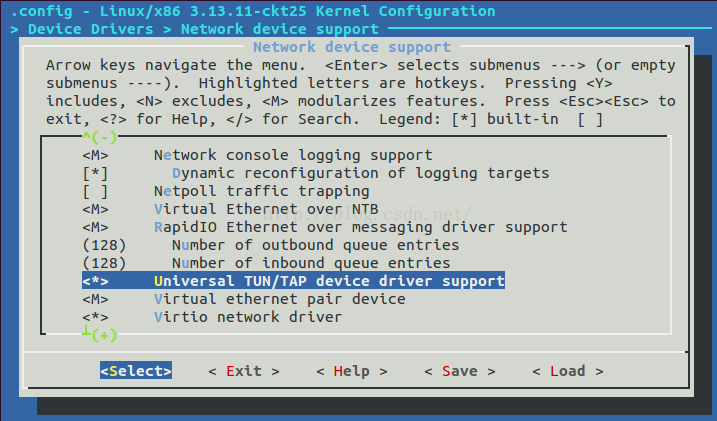


图2-3 编译内核的菜单（2）

## 2.3 Linux下socket编程技术简介

### 2.3.1 socket简介

众所周知，两个进程之间进行通信的基本前提是可以唯一标识，在本地可以使用进程的pid来进行唯一标识，但如果在网络中显然pid就并不能作为唯一标识符使用了。由于IP层的ip地址可以唯一标示主机，而TCP层协议和端口号可以唯一标示主机的一个进程，这意味着可以利用“ip地址＋协议＋端口号”唯一标示网络中的一个进程。

唯一标识的问题解决之后，就可以使用socket进行通信了，socket是在应用层和传输层之间的一个抽象层，用于描述IP地址和端口，是一个通信链的句柄，可以用来实现上述的通信过程。

socket起源于UNIX，在Unix一切皆文件哲学的思想下，socket是一种"打开—读/写—关闭"模式的实现，服务器和客户端各自维护一个"文件"，在建立连接打开后，可以向自己文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。

### 2.3.2 socket通信流程

以使用TCP协议的socket为例，如图2-4：

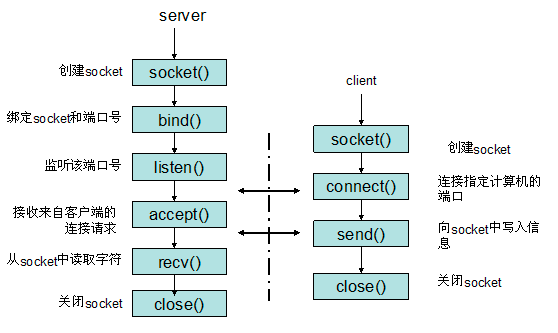


图2-4 socket通信流程

1.服务器根据地址类型（ipv4,ipv6）、socket类型、协议创建socket

2.服务器为socket绑定ip地址和端口号

3.服务器socket监听端口号请求，随时准备接收客户端发来的连接，这时候服务器的socket并没有被打开

4.客户端创建socket

5.客户端打开socket，根据服务器ip地址和端口号试图连接服务器socket

6.服务器socket接收到客户端socket请求，被动打开，开始接收客户端请求，直到客户端返回连接信息。这时候socket进入阻塞状态，所谓阻塞即accept()方法一直到客户端返回连接信息后才返回，开始接收下一个客户端谅解请求

7.客户端连接成功，向服务器发送连接状态信息

8.服务器accept方法返回，连接成功

9.客户端向socket写入信息

10.服务器读取信息

11.客户端关闭

12.服务器端关闭

### 2.3.3 socket编程相关API

（1）int socket(int domain, int type, int protocol);

根据指定的地址族、数据类型和协议来分配一个socket的描述字及其所用的资源。

domain:协议族，常用的有AF\_INET、AF\_INET6、AF\_LOCAL、AF\_ROUTE其中AF\_INET代表使用ipv4地址

type:socket类型，常用的socket类型有，SOCK\_STREAM、SOCK\_DGRAM、SOCK\_RAW、SOCK\_PACKET、SOCK\_SEQPACKET等

protocol:协议。常用的协议有，IPPROTO\_TCP、IPPTOTO\_UDP、IPPROTO\_SCTP、IPPROTO\_TIPC等

（2）int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

把一个地址族中的特定地址赋给socket

sockfd:socket描述字，也就是socket引用

addr:要绑定给sockfd的协议地址

addrlen:地址的长度

（3）int listen(int sockfd, int backlog);

监听socket

sockfd:要监听的socket描述字

backlog:相应socket可以排队的最大连接个数

（4）int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

连接某个socket

sockfd:客户端的socket描述字

addr:服务器的socket地址

addrlen:socket地址的长度

（5）int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

  TCP服务器监听到客户端请求之后，调用accept()函数取接收请求

sockfd:服务器的socket描述字

addr:客户端的socket地址

addrlen:socket地址的长度

（6）ssize\_t read(int fd, void \*buf, size\_t count);

读取socket内容

fd:socket描述字

buf：缓冲区

count：缓冲区长度

（7）ssize\_t write(int fd, const void \*buf, size\_t count);

向socket写入内容，其实就是发送内容

fd:socket描述字

buf：缓冲区

count：缓冲区长度

（8）int close(int fd);

socket标记为以关闭 ，使相应socket描述字的引用计数-1，当引用计数为0的时候，触发TCP客户端向服务器发送终止连接请求。

## 2.4 TCP、UDP以及ICMP报文的结构

### 2.4.1 TCP

TCP的首部格式如图2-5所示：

---Source Port是源端口，16位。

---Destination Port是目的端口，16位。

---Sequence Number是发送数据包中的第一个字节的序列号，32位。

---Acknowledgment Number是确认序列号，32位。

---Data Offset是数据偏移，4位，该字段的值是TCP首部（包括选项）长度除以4。[1] 

---标志位： 6位，URG表示Urgent Pointer字段有意义：

ACK表示Acknowledgment Number字段有意义

PSH表示Push功能，RST表示复位TCP连接

SYN表示SYN报文（在建立TCP连接的时候使用）

FIN表示没有数据需要发送了（在关闭TCP连接的时候使用）

Window表示接收缓冲区的空闲空间，16位，用来告诉TCP连接对端自己能够接收的最大数据长度。

---Checksum是校验和，16位。

---Urgent Pointers是紧急指针，16位，只有URG标志位被设置时该字段才有意义，表示紧急数据相对序列号（Sequence Number字段的值）的偏移。

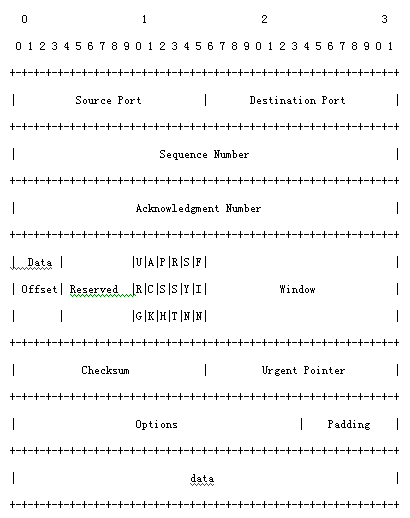


图2-5 TCP首部

### 2.4.2 UDP

UDP报头由4个域组成，其中每个域各占用2个字节，具体如图2-6：

[](http://baike.baidu.com/pic/UDP/571511/0/b2de9c82d158ccbf4049fb531bd8bc3eb03541e5?fr=lemma&ct=single)源[端口号](http://baike.baidu.com/view/642103.htm" \t "_blank)

图2-6 UDP首部

目标端口号

[数据报](http://baike.baidu.com/view/121589.htm)长度

校验值

UDP协议使用端口号为不同的应用保留其各自的数据传输通道。UDP和[TCP协议](http://baike.baidu.com/view/1240122.htm" \t "_blank)正是采用这一机制实现对同一时刻内多项应用同时发送和接收数据的支持。数据发送一方（可以是客户端或服务器端）将UDP数据包通过源端口发送出去，而数据接收一方则通过目标端口接收数据。有的网络应用只能使用预先为其预留或注册的静态端口；而另外一些网络应用则可以使用未被注册的动态端口。因为UDP报头使用两个字节存放端口号，所以端口号的有效范围是从0到65535。一般来说，大于49151的端口号都代表动态端口。

数据报的长度是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要被用来计算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为65535字节。不过，一些实际应用往往会限制数据报的大小，有时会降低到8192字节。

UDP协议使用报头中的校验值来保证数据的安全。校验值首先在数据发送方通过特殊的算法计算得出，在传递到接收方之后，还需要再重新计算。如果某个数据报在传输过程中被第三方篡改或者由于线路噪音等原因受到损坏，发送和接收方的校验计算值将不会相符，由此UDP协议可以检测是否出错。这与TCP协议是不同的，后者要求必须具有校验值。

### 2.4.3 ICMP

ICMP 经常被认为是 IP 层的一个组成部分，它传递差错报文以及其他需要注意的信息。ICMP 报文通常被 IP 层或更高层协议(TCP 或 UDP)使用。ICMP 报文是在 IP 数据报内部传输的。IP 协议是不可靠协议，不能保证 IP 数据报能够成功的到达目的主机，无法进行差错控制，而 ICMP 协议能够协助 IP 协议完成这些功能。图2-7是 ICMP 报文的数据结构：

类型：一个 8 位类型字段，表示 ICMP 数据包类型;

代码：一个 8 位代码域，表示指定类型中的一个功能，如果一个类型中只有一种功能，代码域置为 0;

检验和：数据包中 ICMP 部分上的一个 16 位检验和;

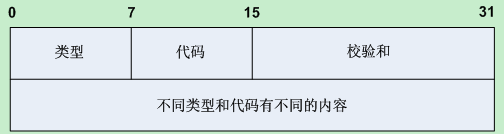


图2-7 ICMP结构

ICMP 报文类型：

ICMP 报文大致可分为两类：差错报文、查询报文。具体消息类型如图2-8所示：

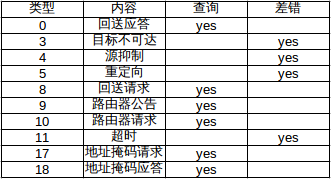


图2-8 ICMP报文类型

Ping命令使用的就是ICMP报文，通过发送类型8和接收类型0来完成。

# 第三章 需求分析

## 3.1 需求

1. socket套接字编程，对外发送数据报文；
2. 通过修改路由，tun设备拦截数据包；
3. 根据不同类型的包的特点，在终端打印如下信息：IP来源，IP目的地址，数据包大小，应用层协议。

## 3.2 分析

需要分为两部分编程，一个是socket套接字编程部分，另一个部分是读取tun设备信息并打印。

测试使用Linux下的tcpdump程序。

大致流程如下：

1. 创建TUN设备，准备从中读取信息；

2. 给创建好的TUN设备分配ip，修改路由；

3. 使用socket发送报文；

4. 使用tcpdump验证截获信息。

## 3.3 流程图

如图3-1所示

开始

开始

创建TUN设备并监听

创建对应类型socket

结束

判断类型

发送对应类型报文

对应类型报文的处理

结束

图3-1 流程图

# 第四章 详细设计

## 4.1 开发环境

操作系统为ArchLinux，桌面环境为GNOME，终端使用zsh，代码编辑使用的是gedit，代码编译使用g++。

ArchLinux，见图4-1：

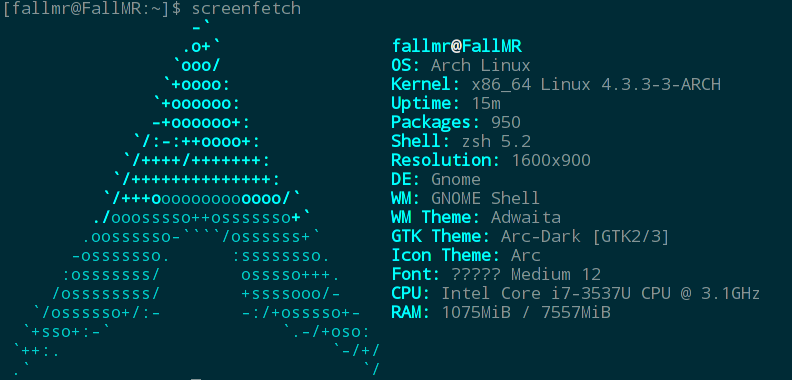


图4-1 ArchLinux

GNOME，见图4-2：



图4-2 GNOME

GEdit，见图4-3：

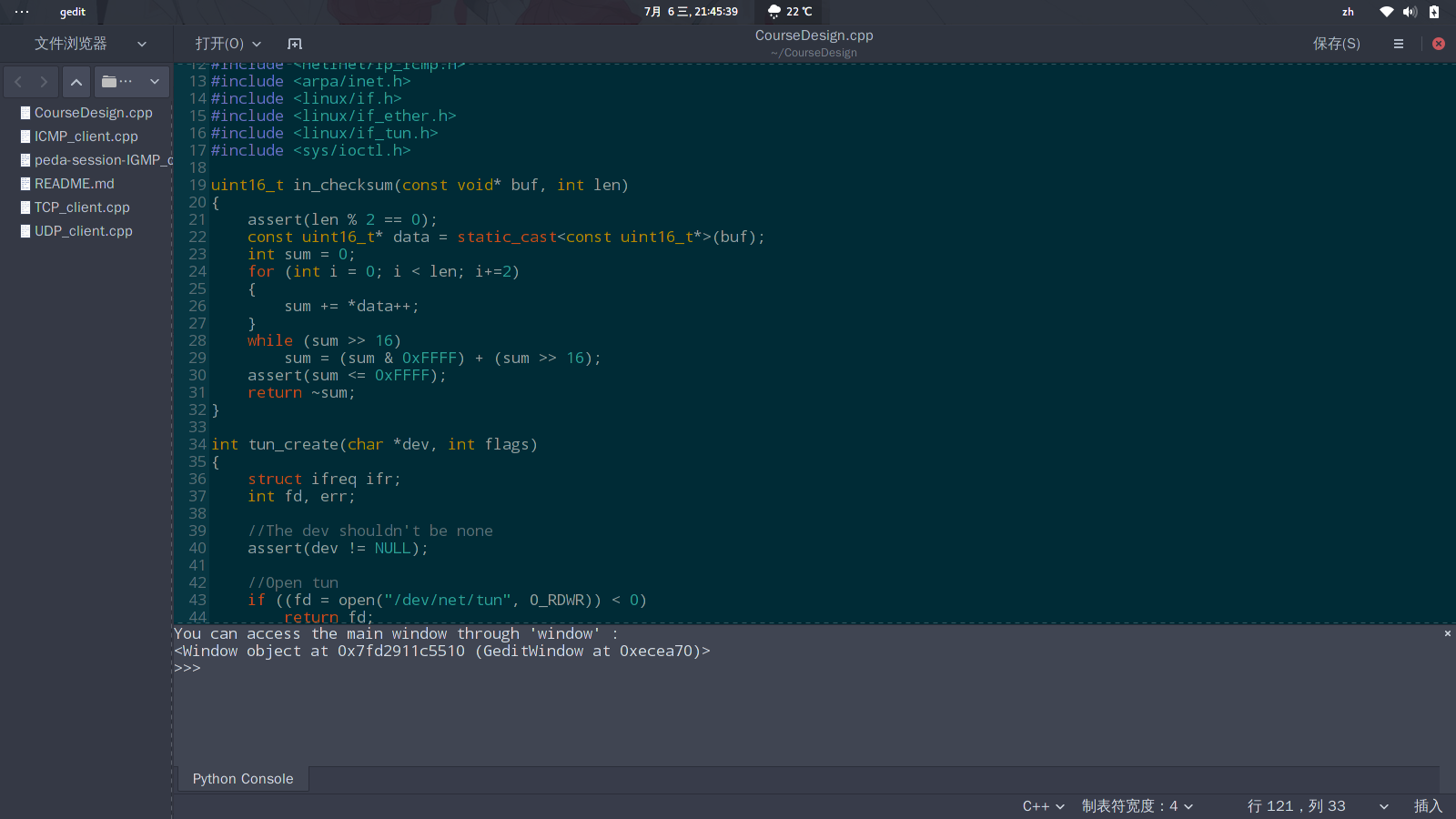


图4-3 GEdit

g++，见图4-4：

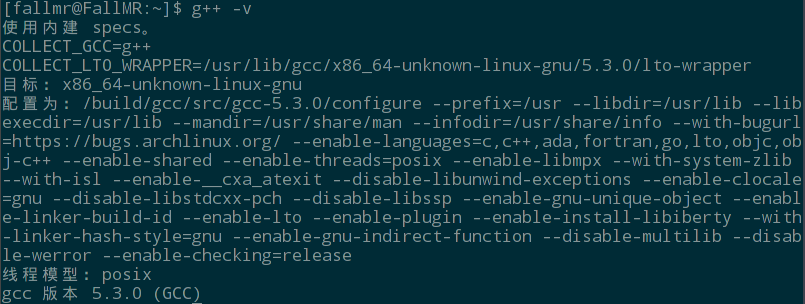


图4-4 g++编译器版本信息

## 4.2 socket部分

本部分代码的功能是使用socket进行TCP、UDP、ICMP报文的发送。

1.TCP报文的发送：

//TCP\_client.cpp

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/stat.h>

#include <arpa/inet.h>

#define MAXBUF 256

int main()

{

int fd, len;

struct sockaddr\_in server\_addr;

char buf[MAXBUF];

//创建TCP类型socket

if((fd = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP)) < 0)

{

perror("socket error:");

exit(1);

}

len = sizeof(server\_addr);

memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = inet\_addr("192.168.0.2");

server\_addr.sin\_port = htons(7758);

//尝试连接192.168.0.2:7758

if(connect(fd, (struct sockaddr \*)&server\_addr, len) < 0)

{

perror("connect error:");

exit(1);

}

memset(buf, 0, MAXBUF);

if(read(fd, buf, MAXBUF)<=0)

{

perror("read error:");

exit(1);

}

close(fd);

printf("\nread: %s\n", buf);

return 0;

}

2.UDP报文的发送：

//UDP\_client.cpp

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <sys/stat.h>

#include <arpa/inet.h>

int main()

{

int fd;

struct sockaddr\_in servaddr;

fd = socket(PF\_INET, SOCK\_DGRAM, 0);

bzero(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(5001);

servaddr.sin\_addr.s\_addr = inet\_addr("192.168.0.2");

char sendline[100];

sprintf(sendline, "Hello, world!");

//将数据发送至192.168.0.2:5001

sendto(fd, sendline, strlen(sendline), 0, (struct sockaddr \*)&servaddr, sizeof(servaddr));

close(fd);

return 0;

}

3.ICMP报文的发送：

//ICMP\_client.cpp

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <strings.h>

#include <netinet/ip\_icmp.h>

#include <stdlib.h>

//发送ICMP\_ECHO请求（ping）

void send\_echo\_req(int sockfd, struct sockaddr\_in \*dstaddr);

//计算校验和

uint16\_t in\_cksum(uint16\_t \*addr, int len);

int main()

{

int sockfd;

struct sockaddr\_in dstaddr;

if ((sockfd = socket(PF\_INET, SOCK\_RAW, IPPROTO\_ICMP)) == -1)

{

perror("socket");

exit(1);

}

bzero(&dstaddr, sizeof(dstaddr));

dstaddr.sin\_family = AF\_INET;

dstaddr.sin\_port = htons(0);

if (inet\_pton(AF\_INET, "192.168.0.2", &dstaddr.sin\_addr) <= 0)

{

perror("inet\_pton");

exit(1);

}

send\_echo\_req(sockfd, &dstaddr);

exit(0);

}

void send\_echo\_req(int sockfd, struct sockaddr\_in \*dstaddr)

{

char buf[100];

size\_t len = sizeof(struct icmp);

struct icmp \*icmp;

socklen\_t dstlen = sizeof(struct sockaddr\_in);

//手动设置ICMP报文属性

bzero(buf, sizeof(buf));

icmp = (struct icmp \*)buf;

icmp->icmp\_type = ICMP\_ECHO;

icmp->icmp\_code = 0;

icmp->icmp\_id = getpid();

icmp->icmp\_seq = 1;

icmp->icmp\_cksum = in\_cksum((uint16\_t \*) icmp, sizeof(struct icmp));

//发送ICMP\_ECHO报文

if (sendto(sockfd, buf, len, 0, (struct sockaddr \*)dstaddr, dstlen) == -1)

{

perror("send to");

exit(1);

}

}

uint16\_t in\_cksum(uint16\_t \*addr, int len)

{

int nleft = len;

uint32\_t sum = 0;

uint16\_t \*w = addr;

uint16\_t answer = 0;

while (nleft > 1)

{

sum += \*w++;

nleft -= 2;

}

if (nleft == 1)

{

\*(unsigned char \*)(&answer) = \*(unsigned char \*)w;

sum += answer;

}

sum = (sum >> 16) + (sum & 0xffff);

sum += (sum >> 16);

answer = ~sum;

return(answer);

}

## 4.3 TUN部分

//CourseDeign.cpp

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <fcntl.h>

#include <algorithm>

#include <assert.h>

#include <netinet/in.h>

#include <netinet/ip.h>

#include <netinet/tcp.h>

#include <netinet/udp.h>

#include <netinet/ip\_icmp.h>

#include <arpa/inet.h>

#include <linux/if.h>

#include <linux/if\_ether.h>

#include <linux/if\_tun.h>

#include <sys/ioctl.h>

//同ICMP部分的校验和计算

uint16\_t in\_checksum(const void\* buf, int len)

{

assert(len % 2 == 0);

const uint16\_t\* data = static\_cast<const uint16\_t\*>(buf);

int sum = 0;

for (int i = 0; i < len; i+=2)

{

sum += \*data++;

}

while (sum >> 16)

sum = (sum & 0xFFFF) + (sum >> 16);

assert(sum <= 0xFFFF);

return ~sum;

}

//创建TUN设备

int tun\_create(char \*dev, int flags)

{

struct ifreq ifr;

int fd, err;

//The dev shouldn't be none

assert(dev != NULL);

//Open tun

if ((fd = open("/dev/net/tun", O\_RDWR)) < 0)

return fd;

//Initialize ifr

bzero(&ifr, sizeof(ifr));

ifr.ifr\_flags |= flags;

//When the name had been specified

if (\*dev != '\0')

strncpy(ifr.ifr\_name, dev, IFNAMSIZ);

//Initialize struct ifr, get information

if ((err = ioctl(fd, TUNSETIFF, (void \*)&ifr)) < 0)

{

close(fd);

return err;

}

strcpy(dev, ifr.ifr\_name);

return fd;

}

//ICMP类型报文的处理

void icmp\_input(int fd, const void\* input, const void\* payload, int len)

{

const struct iphdr\* iphdr = static\_cast<const struct iphdr\*>(input);

const struct icmphdr\* icmphdr = static\_cast<const struct icmphdr\*>(payload);

const int iphdr\_len = iphdr->ihl\*4;

if (icmphdr->type == ICMP\_ECHO)

{

char source[INET\_ADDRSTRLEN];

char dest[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &iphdr->saddr, source, INET\_ADDRSTRLEN);

inet\_ntop(AF\_INET, &iphdr->daddr, dest, INET\_ADDRSTRLEN);

printf("IP %s > %s: ", source, dest);

printf("ICMP echo request, id %d, seq %d, length %d\n",

ntohs(icmphdr->un.echo.id),

ntohs(icmphdr->un.echo.sequence),

len - iphdr\_len);

}

}

//TCP报文的处理

void tcp\_input(int fd, const void\* input, const void\* payload, int tot\_len)

{

const struct iphdr\* iphdr = static\_cast<const struct iphdr\*>(input);

const struct tcphdr\* tcphdr = static\_cast<const struct tcphdr\*>(payload);

const int iphdr\_len = iphdr->ihl\*4;

const int tcp\_seg\_len = tot\_len - iphdr\_len;

const int tcphdr\_size = sizeof(\*tcphdr);

if (tcp\_seg\_len >= tcphdr\_size

&& tcp\_seg\_len >= tcphdr->doff\*4)

{

const int tcphdr\_len = tcphdr->doff\*4;

if (tcphdr->syn)

{

char source[INET\_ADDRSTRLEN];

char dest[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &iphdr->saddr, source, INET\_ADDRSTRLEN);

inet\_ntop(AF\_INET, &iphdr->daddr, dest, INET\_ADDRSTRLEN);

printf("IP %s:%d > %s:%d: ", source, ntohs(tcphdr->source), dest, ntohs(tcphdr->dest));

printf("TCP request, seq %u , length %d\n", ntohl(tcphdr->seq), tot\_len - iphdr\_len - tcphdr\_len);

}

}

}

//UDP报文的处理

void udp\_input(int fd, const void\* input, const void\* payload, int len)

{

const struct iphdr\* iphdr = static\_cast<const struct iphdr\*>(input);

const struct udphdr\* udphdr = static\_cast<const struct udphdr\*>(payload);

char source[INET\_ADDRSTRLEN];

char dest[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &iphdr->saddr, source, INET\_ADDRSTRLEN);

inet\_ntop(AF\_INET, &iphdr->daddr, dest, INET\_ADDRSTRLEN);

printf("IP %s:%d > %s:%d: ", source, ntohs(udphdr->source), dest, ntohs(udphdr->dest));

printf("UDP request, length %d\n", ntohl(udphdr->len));

}

int main(int argc, char \*argv[])

{

int fd, ret;

char tun\_name[IFNAMSIZ];

unsigned char buf[4096];

tun\_name[0] = '\0';

//Create a tun device

fd = tun\_create(tun\_name, IFF\_TUN | IFF\_NO\_PI);

if (fd < 0)

{

perror("tun\_create");

return 1;

}

printf("TUN name is %s\n", tun\_name);

//Recieve packets

while (1)

{

union

{

unsigned char buf[ETH\_FRAME\_LEN];

struct iphdr iphdr;

};

const int iphdr\_size = sizeof iphdr;

int nread = read(fd, buf, sizeof(buf));

if (nread < 0)

{

perror("read");

close(fd);

exit(1);

}

const int iphdr\_len = iphdr.ihl\*4;

if (nread >= iphdr\_size

&& iphdr.version == 4

&& iphdr\_len >= iphdr\_size

&& iphdr\_len <= nread

&& iphdr.tot\_len == htons(nread)

&& in\_checksum(buf, iphdr\_len) == 0)

{

const void\* payload = buf + iphdr\_len;

//判断报文类型

if (iphdr.protocol == IPPROTO\_ICMP)

{

icmp\_input(fd, buf, payload, nread);

}

else if (iphdr.protocol == IPPROTO\_TCP)

{

tcp\_input(fd, buf, payload, nread);

}

else if (iphdr.protocol == IPPROTO\_UDP)

{

udp\_input(fd, buf, payload, nread);

}

}

}

return 0;

}

# 

# 第五章 软件测试

软件测试是指在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程。此次的软件测试主要是对软件的正确性进行测试。

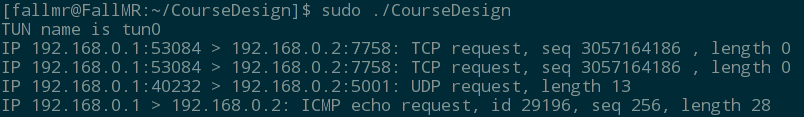
TUN设备，见图5-1：

图5-1 TUN设备的创建和监听

Socket，见图5-2：

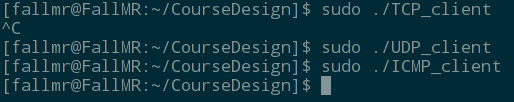


图5-2 运行使用socket编写的客户端

Tcpdump对比，见图5-3：

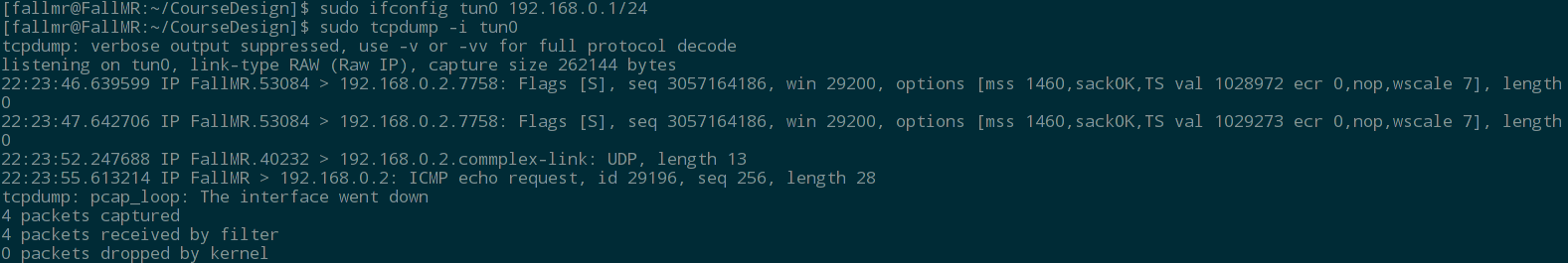


图5-3 tcpdump对比结果

# 第六章 总结与展望

通过这次课程设计，接触了Linux的tun设备内核模块，也间接带来了第一次自己手动编译内核的经历，同时加深了对各种协议结构的了解，熟悉了socket编程。我相信这对日后相关的作业有很大的帮助。

# 参考文献

[1] 百度百科

[2] Linux Manual Pages