武汉大学国家网络安全学院

课程实验(设计)报告

课程名称 ： 网络程序设计

实验内容 ： Ping 程序分析

专业(班) ： 国家网络安全学院 信安 3 班

学号/姓名 :

完成时间 ： 2021 年 6 月 5 日

2021 年 6 月 5 日

实验目的

掌握 Ping 程序的框架，能够利用原始套接字实现对底层协议的支持，通过分析 Linux 中 Ping 程序加深 Socket 编程理解，以及 Linux 中系统程序的编写方法。

实验要求

1. 掌握原始套接字的编程方法
2. 掌握 Ping 程序的基本框架，包括参数获取、ICMP 数据包构造、ICMP 数据包解包、CRC16 校验算法、时间差计算方法等
3. 分析 Linux 系统中自带的 Ping 程序源代码

- 获取 Ping 程序源代码
- 编译调试通过
- 对源代码进行梳理分析，撰写程序分析报告

实验环境

实验环境采用 **vscode remote + wsl**，操作系统的版本为 **Ubuntu 20.04.2 LTS**，如下图所示

```
❯ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.2 LTS
Release:        20.04
Codename:       focal
```

操作系统版本

内核版本如下图所示

```
❯ uname -a
Linux qiufeng 5.10.16.3-microsoft-standard-WSL2
```

内核版本

gcc 版本为 9.3.0

```
❯ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

gcc 版本

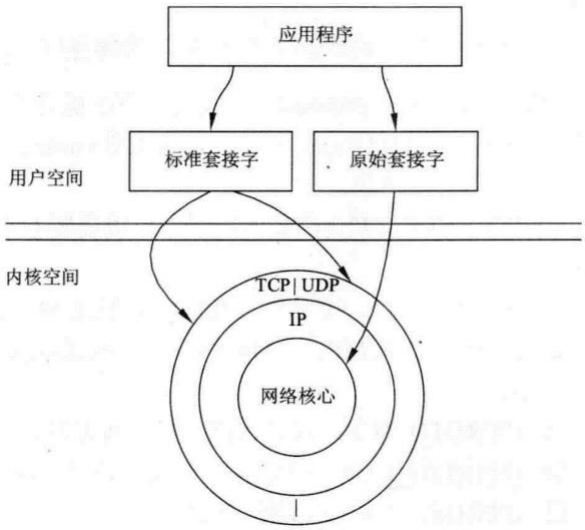
实验步骤

1. 原始套接字的编程方法

1.1 原始套接字简介

原始套接字与 **IP** 层级网络协议栈核心打交道，其与标准套接字的关系如下图

所示



与标准套接字关系

原始套接字具有如下标准套接字不具备的功能

- 可以读/写 ICMP、IGMP 分组，这些都是 IP 层的协议
- 可以读写特殊的 IP 数据包，内核不处理这些数据包的协议字段
- 可以对 IP 头部进行操作，构造特定类型的 TCP 和 UDP 分组

1.2 原始套接字的编程方法

1.2.1 创建

我们使用 ***socket*** 函数创建原始套接字。与标准套接字不同的是，第二个参数需要指定为 ***SOCK_RAW***

```
//！ 创建原始套接字
rawsock = socket(AF_INET, SOCK_RAW, protocol→p_proto);
```

创建原始套接字

1.2.2 发送

通常情况下可以使用 ***sendto*** 函数来指定并向目的地址发送数据，当已经使用了 ***bind*** 函数绑定目标地址之后可以直接使用 ***write*** 或者 ***send*** 函数来发送数据，例如

```
//! 发送数据
size = sendto(rawsock, send_buff, 64, 0,
              (struct sockaddr *)&dest, sizeof(dest));
```

发送数据

1.2.3 接受

通发送报文的规则，可以使用 ***recvfrom*** 或者 ***recv*** 或者 ***read*** 来获取数据，例如

```
//! 接收数据
int size = recv(rawsock, recv_buff, sizeof(recv_buff), 0);
```

接收数据

2. Ping 程序的基本框架

在本节中，将以书上的例子进行分析，在后一节中将分析 Linux 中 Ping 程序的源代码

2.1 参数获取

参数获取的逻辑如下

1. 检验参数的个数是否正确，如果不正确则打印帮助信息

```c
if (argc < 2)
{
    icmp_usage();
    return -1;
}
```

参数个数校验

2. 如果输入的参数是域名地址，则先通过 ***gethostbyname*** 函数将其转换为 IP 地址，再复制到 ***dest*** 中，否则直接复制到 ***dest*** 中

```c
inaddr = inet_addr(argv[1]);
//! 如果输入的是域名
if (inaddr == INADDR_NONE)
{
    //! 将域名转换为 IP 地址
    host = gethostbyname(argv[1]);
    if (host == NULL)
    {
        perror("gethostbyname");
        return -1;
    }
    //* 将地址复制到dest中
    memcpy((char *)&dest.sin_addr, host->h_addr, host->h_length);
}
else
{
    memcpy((char *)&dest.sin_addr, &inaddr, sizeof(inaddr));
}
```

参数处理

2.2 ICMP 数据包构造

2.2.1 基本格式

ICMP 报头的格式如下图所示

报头格式

其中各个字段的含义如下

- **类型**：占一字节，标识 ICMP 报文的类型，目前已定义了 14 种，从类型值来看ICMP报文可以分为两大类。第一类是取值为1~127的差错报文，第 2 类是取值 128 以上的信息报文
- **代码**：占一字节，标识对应 ICMP 报文的代码。它与类型字段一起共同标识了 ICMP 报文的详细类型
- **校验和**：这是对包括 ICMP 报文数据部分在内的整个 ICMP 数据报的校验和(数据部分和报头部分)，以检验报文在传输过程中是否出现了差错。其计算方法与在 IP 报头中的校验和计算方法是一样的
- **标识符**：占两字节，用于标识本 ICMP 进程，但仅适用于回显请求和应答 ICMP 报文，对于目标不可达 ICMP 报文和超时 ICMP 报文等，该字段的值为 0

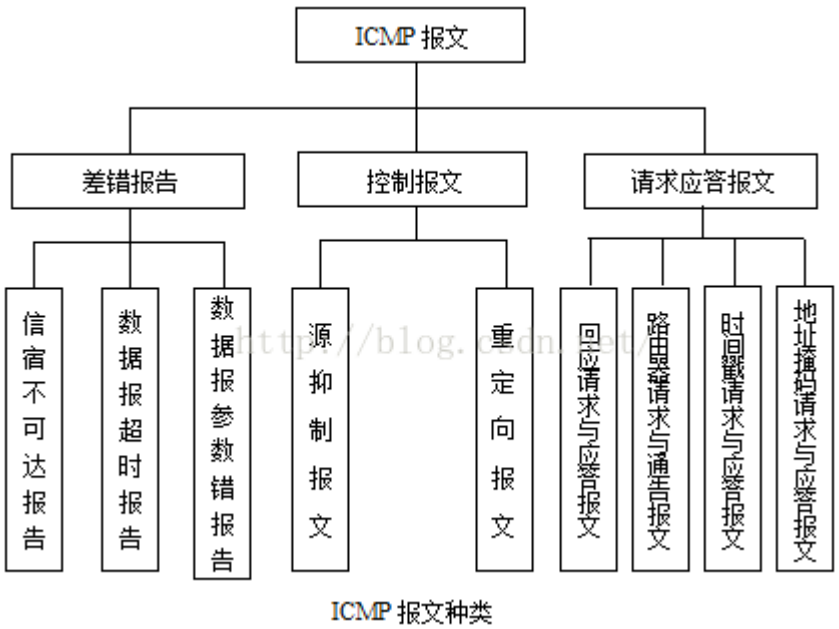
2.2.2 报文类型

ICMP 报文类型有多种，常见的如：

- **回显请求**：Ping 工具通过发送 ICMP 回显消息检查特定节点的 IPv4 连接以排查网络问题。类型值为 0
- **回显应答**：节点发送回显答复消息响应 ICMP 回显消息。类型值为 8

- **重定向**：路由器发送“重定向”消息，告诉发送主机到目标 IPv4 地址更好的路由。类型值为 5
- **源抑制**：路由器发送“源结束”消息，告诉发送主机它们的 IPv4 数据报将被丢弃——因为路由器上发生了拥塞。于是，发送主机将以较低的频度发送数据报。类型值为 4
- **超时**：这个消息有两种用途。第一，当超过 IP 生存期时向发送系统发出错误信息。第二，如果分段的 IP 数据报没有在某种期限内重新组合，这个消息将通知发送系统。类型值为 11
- **无法到达目标**：路由器和目标主机发送“无法到达目标”消息，通知发送主机它们的数据无法传送。类型值为 3

如下图所示



ICMP 报文种类

ICMP 报文类型

2.2.3 数据包构造

Ping 程序发送数据时使用的 ICMP 报文类型为**回显请求**。

要构造回显请求报文，我们需要对**报头**进行如下设置

- 类型设置为 ***8***，即 ICMP ECHO
- 代码值设置为 ***0***
- 计算校验和
- 设置报文序列号。序列号是一个 16 位的值，通常由一个递增的值生成
- 使用进程的 PID 设置 ID 字段，用于区别不同的进程

ICMP 回显报文的数据部分可以任意设置，但由于以太网包的总长度不能小于 ***46*** 字节，因此我们进行任意数据填充，将其长度增加到 64 个字节

对应的代码如下

```c
static void icmp_pack(struct icmp *icmph, int seq, struct timeval *tv, int length) {
    unsigned char i = 0;
    //* 设置请求类型为回显请求
    icmph->icmp_type = ICMP_ECHO;
    //* 设置请求码为 0
    icmph->icmp_code = 0;
    //* 校验和字段
    icmph->icmp_cksum = 0;
    //* 序列号
    icmph->icmp_seq = seq;
    //* 填写进程 pid，用于区分不同的进程
    icmph->icmp_id = pid & 0xffff;
    //* 填充数据部分，使其满足以太网最小帧长度的要求
    for (i = 0; i < length - 8; i++)
        icmph->icmp_data[i] = i;
    //* 计算校验和
    icmph->icmp_cksum = icmp_cksum((unsigned char *) icmph, length);
}
```

ICMP **数据包构造**

2.3 校验和计算

由于 ICMP 使用了原始套接字，因此我们必须手动计算校验和。需要注意的是，与 IP 头部的校验和字段不同，ICMP 头部的校验和校验 **ICMP 首部和数据部分**。

ICMP 的校验和采用 CRC16 校验算法，它是一种根据网络数据包或电脑文件

等数据产生简短固定位数校验码的一种散列函数，主要用来检测或校验数据传输或者保存后可能出现的错误。对应代码如下所示

```c
static unsigned short icmp_cksum(unsigned char *data, int len) {
    //* 存储最后的计算结果
    int sum = 0;
    //* 数组长度是否为奇数
    int odd = len & 0x01;
    //* 将数据按照两字节为单位进行累加
    while (len & 0xfffe) {
        sum += *(unsigned short *) data;
        data += 2;
        len -= 2;
    }
    //* 如果为奇数个数据，则会剩下最后一个字节
    if (odd) {
        unsigned short tmp = ((*data) << 8) & 0xff00;
        sum += tmp;
    }
    //* 高低位相加
    sum = (sum >> 16) + (sum & 0xffff);
    //* 加入溢出位
    sum += (sum >> 16);
    //* 最终结果为取反值
    return ~sum;
}
```

crc16 **校验**

2.4 ICMP 数据包解包

Ping 程序接收数据时使用的 ICMP 报文类型为**回显应答**。

其解包的主要逻辑如下

- 获取 IP 头部
- 解析 IP 头部字段，获取 IP 头部长度，其中 ICMP 报文开始于 IP 数据部分
- 计算 ICMP 长度并判断是否过小(太小不是 ICMP 包)
- 判断 ICMP 类型是否为**回显应答**
- 判断 ID 字段是否为本进程的 pid

- 获取发送时间，并根据当前时间计算出时间差

对应的代码如下

```
//* IP 头部
ip = (struct ip *)buf;
//* IP 头部长度
iphdrlen = ip->ip_hl * 4;
//! ICMP 从 IP 数据字段开始
icmp = (struct icmp *)(buf + iphdrlen);
//* 获取 ICMP 长度，并判断是否过小
len -= iphdrlen;
if (len < 8)
{
    printf("ICMP packets\'s length is less than 8\n");
    return -1;
}
//* ICMP 类型是否为回显应答，且 ID 字段是否和本进程 pid 相等
if ((icmp->icmp_type == ICMP_ECHOREPLY) && (icmp->icmp_id == pid))
{
    struct timeval tv_internel, tv_recv, tv_send;
    //* 根据 seq 查找发送的数据包
    pingm_pakcet *packet = icmp_findpacket(icmp->icmp_seq);
    if (packet == NULL)
        return -1;
    packet->flag = 0;
    //* 获取本包的发送时间
    tv_send = packet->tv_begin;
    //* 获取当前时间
    gettimeofday(&tv_recv, NULL);
    //* 计算时间差
    tv_internel = icmp_tvsub(tv_recv, tv_send);
    rtt = tv_internel.tv_sec * 1000 + tv_internel.tv_usec / 1000;
```

ICMP 解包

2.5 时间差计算

通过函数 ***gettimeofday*** 可以获取微秒级精度的当前时间，其返回值结构体 ***timeval***，定义如下

```c
/* A time value that is accurate to the nearest
   microsecond but also has a range of years.  */
struct timeval
{
  __time_t tv_sec;    /* Seconds.  */
  __suseconds_t tv_usec;  /* Microseconds.  */
};
```

timeval 结构体

当发送 ICMP **回显请求**报文时，我们将当前时间存储到数组 ***pingpacket*** 中，当接收到**回显应答**报文时，我们从数组中找到该报文对应的发送时间 ***t_old***，并通过 ***gettimeofday*** 获取当前时间 ***t_now***，用 ***t_now*** 减去 ***t_old*** 即可以计算出从发送 ICMP 报文到接受时的时间差

3. Linux 中 Ping 程序源码

3.1 获取 Ping 程序源代码

执行命令 ***dpkg -S /bin/ping*** 可以获取 Ping 的源码对应的包为 **iputils**

```
❯ dpkg -S /bin/ping
iputils-ping: /bin/ping
```

iputils

通过网上搜索可以发现其开源在 github 的 [iputils/iputils](https://github.com/iputils/iputils) 仓库中，于是执行如下命令可以获取 Ping 源代码

```
❯ git clone https://github.com/iputils/iputils.git
Cloning into 'iputils'...
remote: Enumerating objects: 3965, done.
remote: Counting objects: 100% (127/127), done.
remote: Compressing objects: 100% (71/71), done.
remote: Total 3965 (delta 57), reused 111 (delta 50), pack-reused 3838
Receiving objects: 100% (3965/3965), 2.28 MiB | 1.08 MiB/s, done.
Resolving deltas: 100% (2687/2687), done.
```

下载 Ping 源码

3.2 编译调试通过

执行如下命令配置安装环境

./configure builddir -DBUILD_MANS=false -DBUILD_HTML_MANS=false

```
❯ ./configure builddir -DBUILD_MANS=false -DBUILD_HTML_MANS=false
Meson configurator encountered an error:

ERROR: Directory /home/qiufeng/courses/networking-programming/iputils/builddir is neither a Meson build directory no
r a project source directory.

Configuration can be changed like this:
  meson configure builddir -Dprefix=/usr
  More info: http://mesonbuild.com/Configuring-a-build-directory.html
The Meson build system
Version: 0.53.2
Source dir: /home/qiufeng/courses/networking-programming/iputils
Build dir: /home/qiufeng/courses/networking-programming/iputils/builddir
Build type: native build
Project name: iputils
Project version: 20210202
C compiler for the host machine: cc (gcc 9.3.0 "cc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0")
C linker for the host machine: cc ld.bfd 2.34
Host machine cpu family: x86_64
Host machine cpu: x86_64
Found pkg-config: /usr/bin/pkg-config (0.29.1)
```

配置安装环境

执行 ***make -j12*** 命令进行编译

```
❯ make -j12
ninja -C builddir
ninja: Entering directory `builddir'
[23/23] Linking target ping/ping.
```

编译

执行编译好的 ping 程序(位于**./builddir/ping/ping** 文件夹下)

```
❯ sudo ./builddir/ping/ping -h
[sudo] password for qiufeng:

Usage
  ping [options] <destination>

Options:
  <destination>      dns name or ip address
  -a                 use audible ping
  -A                 use adaptive ping
  -B                 sticky source address
  -c <count>         stop after <count> replies
  -D                 print timestamps
  -d                 use SO_DEBUG socket option
  -f                 flood ping
  -h                 print help and exit
  -I <interface>     either interface name or address
  -i <interval>      seconds between sending each packet
  -L                 suppress loopback of multicast packets
  -l <preload>       send <preload> number of packages while waiting replies
  -m <mark>          tag the packets going out
  -M <pmtud opt>     define mtu discovery, can be one of <do|dont|want>
  -n                 no dns name resolution
  -O                 report outstanding replies
```

执行 ping 程序

Ping 百度服务器测试能否正常工作

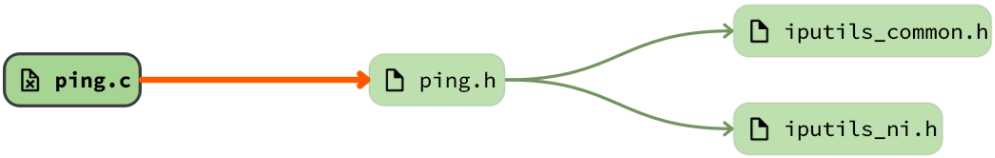
```
❯ sudo ./builddir/ping/ping baidu.com
PING baidu.com (39.156.69.79) 56(84) bytes of data.
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=1 ttl=46 time=25.3 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=2 ttl=46 time=24.9 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=3 ttl=46 time=30.5 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=4 ttl=46 time=25.7 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=5 ttl=46 time=25.9 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=6 ttl=46 time=25.1 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=7 ttl=46 time=26.4 ms
^C64 bytes from 39.156.69.79: icmp_seq=8 ttl=46 time=28.4 ms

--- baidu.com ping statistics ---
8 packets transmitted, 8 received, 0% packet loss, time 7808ms
rtt min/avg/max/mdev = 24.910/26.534/30.456/1.811 ms
```

ping 百度服务器

3.3 对源代码进行梳理分析

3.3.1 文件分析

分析 **ping.c**，其用户头文件的包含如下图所示



用户头文件

包含的系统头文件主要为下面三个

```
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <ifaddrs.h>
```

系统头文件

3.3.2 重要数据结构分析

3.3.2.1 结构体 *addrinfo*

addrinfo 结构主要在网络编程解析 hostname 时使用，***addrinfo*** 通过链表的方式存储其他地址的，可以遍历其属性 ***ai_next*** 获得。其定义如下

```
struct addrinfo
{
  int ai_flags;         /* Input flags.  */
  int ai_family;     /* Protocol family for socket.  */
  int ai_socktype;      /* Socket type.  */
  int ai_protocol;      /* Protocol for socket.  */
  socklen_t ai_addrlen;    /* Length of socket address.  */
  struct sockaddr *ai_addr;    /* Socket address for socket.  */
  char *ai_canonname;      /* Canonical name for service location.  */
  struct addrinfo *ai_next;    /* Pointer to next in list.  */
};
```

addrinfo 结构体

3.3.2.2 结构体 *ping_rts*

ping_rts 结构用来定义 ping 的运行时状态，主要包括 uid(用户 id)、hostname(主机名)、ttl(延时)、interval(数据包间隔)等属性，其定义(一部分)如下

```
struct ping_rts {
    int mark;
    unsigned char *outpack;

    struct rcvd_table rcvd_tbl;

    size_t datalen;
    char *hostname;
    uid_t uid;
    int ident;          /* random id to identify our packets */

    int sndbuf;
    int ttl;

    long npackets;          /* max packets to transmit */
    long nreceived;         /* # of packets we got back */
    long nrepeats;          /* number of duplicates */
    long ntransmitted;      /* sequence # for outbound packets = #sent */
    long nchecksum;         /* replies with bad checksum */
    long nerrors;           /* icmp errors */
    int interval;           /* interval between packets (msec) */
    int preload;
    int deadline;           /* time to die */
```

ping_rts 结构体

3.3.2.3 结构体 *sockaddr_in*

sockaddr_in 结构体用来表示互联网中的套接字地址，它包括 IP 地址和端口号，其定义如下

```
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;     /* Port number.  */
    struct in_addr sin_addr;    /* Internet address.  */

    /* Pad to size of `struct sockaddr'.  */
    unsigned char sin_zero[sizeof (struct sockaddr)
            - __SOCKADDR_COMMON_SIZE
            - sizeof (in_port_t)
            - sizeof (struct in_addr)];
  };
```

sockaddr_in 结构体

3.3.2.4 结构体 *sockaddr*

sockaddr 结构体用来表示通用的套接字地址，它主要是由 14 个字节长度的

字符数组构成，其定义如下

```
struct sockaddr
  {
    __SOCKADDR_COMMON (sa_);  /* Common data: address family and length.  */
    char sa_data[14];   /* Address data.  */
  };
```

sockaddr 结构体

3.3.2.5 结构体 *icmphdr*

icmphdr 结构体主要用来表示 icmp 报头信息，包括，类型、代码、校验和等信息，其定义如下

```
struct icmphdr
{
  uint8_t type;   /* message type */
  uint8_t code;   /* type sub-code */
  uint16_t checksum;
  union
  {
    struct
    {
      uint16_t  id;
      uint16_t  sequence;
    } echo;     /* echo datagram */
    uint32_t  gateway;  /* gateway address */
    struct
    {
      uint16_t  __glibc_reserved;
      uint16_t  mtu;
    } frag;     /* path mtu discovery */
  } un;
};
```

icmphdr

3.3.3 重要函数分析

3.3.3.1 函数 *create_socket*

create_socket 函数的功能为根据参数调用 ***socket*** 函数创建 ping 所需的套接字，其定义如下

```c
static void create_socket(struct ping_rts *rts, socket_st *sock, int family,
                          int socktype, int protocol, int requisite)
{
    int do_fallback = 0;

    errno = 0;

    assert(sock->fd == -1);
    assert(socktype == SOCK_DGRAM || socktype == SOCK_RAW);

    if (socktype == SOCK_DGRAM)
        sock->fd = socket(family, socktype, protocol);

    /* Kernel doesn't support ping sockets. */
    if (sock->fd == -1 && errno == EAFNOSUPPORT && family == AF_INET)
        do_fallback = 1;
    if (sock->fd == -1 && errno == EPROTONOSUPPORT)
        do_fallback = 1;

    /* User is not allowed to use ping sockets. */
    if (sock->fd == -1 && errno == EACCES)
        do_fallback = 1;
```

create_socket 函数

3.3.3.2 函数 *setup*

setup 函数的作用为设置时间、获取进程 ID、注册信号处理函数、检查参数等准备工作，其对应代码(一部分)如下

```c
void setup(struct ping_rts *rts, socket_st *sock)
{
    // ...
    // 配置相关参数

    if (sock->socktype == SOCK_RAW)
        rts->ident = rand() & 0xFFFF;

    set_signal(SIGINT, sigexit);
    set_signal(SIGALRM, sigexit);
    set_signal(SIGQUIT, sigstatus);

    sigemptyset(&sset);
    sigprocmask(SIG_SETMASK, &sset, NULL);

    clock_gettime(CLOCK_MONOTONIC_RAW, &rts->start_time);
```

获取进程ID，识别包要用到

注册信号处理钩子函数

setup 函数

3.3.3.3 函数 *pinger*

pinger 函数负责组织并传送一个 ICMP ECHO 请求包，其中报头的 ID 为 UNIX 进程的 ID，sequence number 是一个递增的整数，data 段的头 8 个字节用来装 UNIX 的时间戳，用来计算往返时间。其对应的代码(一部分)如下

```
int pinger(struct ping_rts *rts, ping_func_set_st *fset, socket_st *sock)
{
    // ...
    //! 如果发够了就随即返回一个正数
    if (rts->exiting || (rts->npackets && rts->ntransmitted >= rts->npackets && !rts->deadline))
        return 1000;

    //! 发送数据包
    i = fset->send_probe(rts, sock, rts->outpack, sizeof(rts->outpack));

    //! 发送成功
    if (i == 0) {
        oom_count = 0;
        advance_ntransmitted(rts);
        if (!rts->opt_quiet && rts->opt_flood) {
            /* Very silly, but without this output with
             * high preload or pipe size is very confusing. */
            if ((rts->preload < rts->screen_width && rts->pipesize < rts->screen_width) ||
                in_flight(rts) < rts->screen_width)
                write_stdout(".", 1);
        }
        return rts->interval - tokens;
    }

    //! 发送失败，处理各种错误
    // ...
}
```

pinger 函数

3.3.3.4 函数 *send_probe*

实际上这是一个函数指针，他根据 IP 地址的类型选择调用 ***ping6_send_probe*** 或 ***ping4_send_probe***，下面以 ***ping4_send_probe*** 为例进行分析。

当 ***pinger*** 做完准备工作之后，会调用 ***send_probe*** 进行正式发包，该函数会填充 ICMP 报头并组最终调用 ***sendto*** 进行数据包的发送，其对应代码如下

```c
int ping4_send_probe(struct ping_rts *rts, socket_st *sock, void *packet,
                     unsigned packet_size __attribute__((__unused__)))
{
    //! icmp 报头
    struct icmphdr *icp;
    // ...

    cc = rts->datalen + 8;          /* skips ICMP portion */

    //! 计算校验和
    icp->checksum = in_cksum((unsigned short *)icp, cc, 0);

    //! 调用 sendto 函数发送数据包
    i = sendto(sock->fd, icp, cc, 0, (struct sockaddr *)&rts->whereto, sizeof(rts->whereto));

    return (cc == i ? 0 : i);
}
```

ping4_send_probe 函数

3.3.3.5 函数 *parse_reply*

同 ***send_probe*** 一样，这同样是一个函数指针，对应有 IPv4 和 IPv6 两个版本，这里以 ***ping4_parse_reply*** 为例进行分析。

函数的作用为解析收到的数据包，其主要功能包括

- 定位 IP 头
- 定位 ICMP
- 计算校验和
- 判断 ICMP 报文类型
- 对比进程 ID
- 计算来回时间
- 错误处理

```c
int ping4_parse_reply(struct ping_rts *rts, struct socket_st *sock,
                      struct msghdr *msg, int cc, void *addr,
                      struct timeval *tv)
{
    // ...
    //! 校验 IP 头部
    ip = (struct iphdr *)buf;
    //! 计算 IP 包头长度
    hlen = ip->ihl * 4;
    //! 处理 ICMP 包部分
    cc -= hlen;
    //! 指向 ICMP 包起始位置
    icp = (struct icmphdr *)(buf + hlen);
    //! 计算校验和
    csfailed = in_cksum((unsigned short *)icp, cc, 0);
    //! 如果是 ICMP 回显类型
    if (icp->type == ICMP_ECHOREPLY) {
        if (!rts->broadcast_pings && !rts->multicast &&
            from->sin_addr.s_addr != rts->whereto.sin_addr.s_addr)
            return 1;
        if (!is_ours(rts, sock, icp->un.echo.id))
            return 1;           /* 'Twas not our ECHO */
        if (gather_statistics(rts, (uint8_t *)icp, sizeof(*icp), cc,
                              ntohs(icp->un.echo.sequence),
                              reply_ttl, 0, tv, pr_addr(rts, from, sizeof *from),
                              pr_echo_reply, rts->multicast)) {
            fflush(stdout);
            return 0;
        }
    }
    // ...
}
```

ping4_parse_reply 函数

3.3.3.6 函数 *main_loop*

main_loop 函数中是一个死循环，它根据预先设置的时间不停的重复**发包->收包->解析包**的过程，直到满足退出情况才退出，其流程主要为

- 检查是否满足推出条件
- 发送报文
- 控制发送报文的时间间隔
- 接受 ICMP 回应包
- 如果接受失败则进行错误处理

- 如果成功则解析收到的报文

对应的函数如下所示

```
int main_loop(struct ping_rts *rts, ping_func_set_st *fset, socket_st *sock,
              uint8_t *packet, int packlen)
{
    // ...

    for (;;) {
        //! 检查退出条件
        if (rts->npackets && rts->nreceived + rts->nerrors >= rts->npackets)
            break;
        if (rts->deadline && rts->nerrors)
            break;
        //! 发送报文
        do {
            next = pinger(rts, fset, sock);
            next = schedule_exit(rts, next);
        } while (next <= 0);
        //! 控制发送时间将额
        if (rts->opt_adaptive || rts->opt_flood_poll || next < SCHINT(rts->interval)) {
            int recv_expected = in_flight(rts);

            //! 超时处理
            if (1000 % HZ == 0 ? next <= 1000 / HZ : (next < INT_MAX / HZ && next * HZ <= 1000))
                    // ...
                }
            }
            // ...
        }
        //! 接受 ICMP 回应包
        for (;;) {
            // ...
            //! 如果接受失败
            if (cc < 0) {
                // ...
            }
            //! 解析收到的包
            not_ours = fset->parse_reply(rts, sock, &msg, cc, addrbuf, recv_timep);
        // ...
```

main_loop 函数

3.3.3.7 函数 *ping4/6_run*

当处理完命令行参数之后，***main*** 函数会根据 IPv4 或者 IPv6 调用 ***ping4_run*** 或者 ***ping6_run*** 进入 ping 的主要流程，下面以 ***ping4_run*** 为例进行分析。

ping4_run 包含的主要步骤如下

- 处理命令行参数并进行相应的配置和参数初始化
- 调用 ***setup*** 函数
- 调用 ***main_loop*** 函数，进入循环

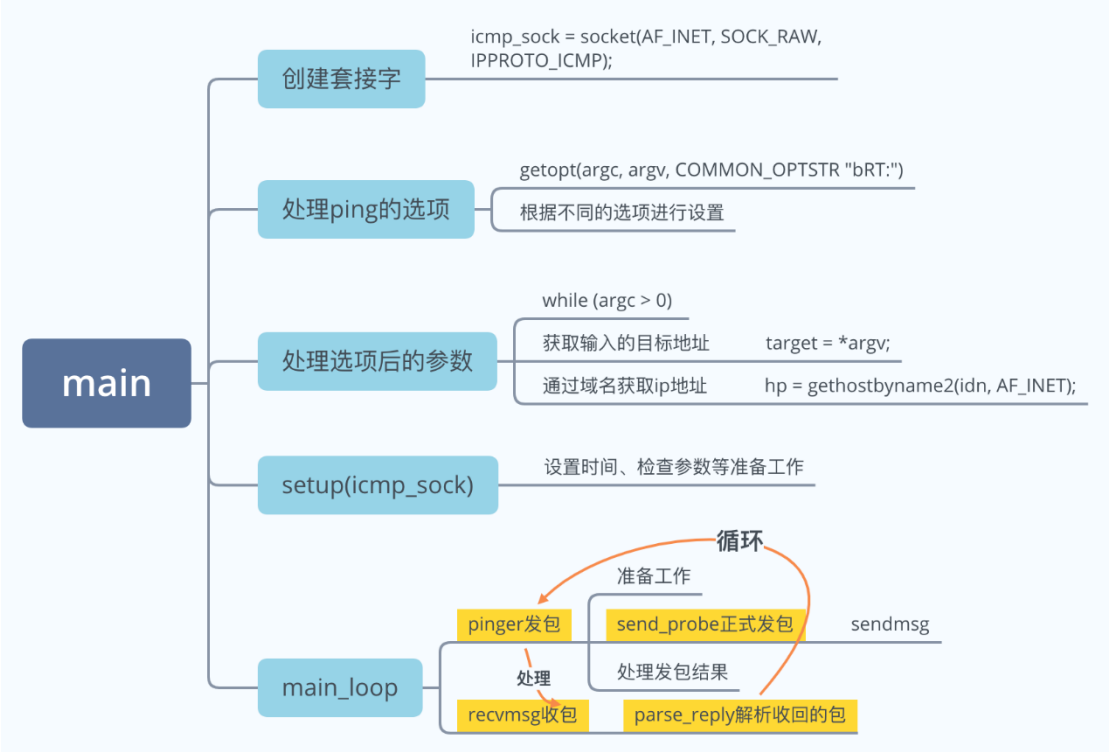
其对应的代码(一部分)如下

```c
int ping4_run(struct ping_rts *rts, int argc, char **argv, struct addrinfo *ai,
              socket_st *sock)
{
    static const struct addrinfo hints = {
        .ai_family = AF_INET,
        .ai_protocol = IPPROTO_UDP,
        .ai_flags = getaddrinfo_flags
    };
    int hold, packlen;
    unsigned char *packet;
    char *target;
    char hnamebuf[NI_MAXHOST];
    unsigned char rspace[3 + 4 * NROUTES + 1];  /* record route space */
    uint32_t *tmp_rspace;

    //! 设置配置和参数初始化
    // ...

    //! 调用 setup
    setup(rts, sock);
    //! 进入 main_loop 循环
    hold = main_loop(rts, &ping4_func_set, sock, packet, packlen);
    free(packet);
    return hold;
}
```

ping4_run 函数

3.3.4 流程分析

程序的流程图如下



流程图

首先通过 ***getopt*** 函数对输入的命令行参数进行处理

```
//! 解析命令行参数
while ((ch = getopt(argc, argv, "h?" "4bRT:" "6F:N:" "aABc:dDfi:I:l:Lm:M:n
    switch(ch) {
    /* IPv4 specific options */
    case '4':
        if (hints.ai_family == AF_INET6)
            error(2, 0, _("only one -4 or -6 option may be specified"));
        hints.ai_family = AF_INET;
        break;
    case 'b':
        rts.broadcast_pings = 1;
        break;
    case 'R':
        if (rts.opt_timestamp)
            error(2, 0, _("only one of -T or -R may be used"));
        rts.opt_rroute = 1;
        break;
    case 'T':
        if (rts.opt_rroute)
            error(2, 0, _("only one of -T or -R may be used"));
```

处理命令行参数

通过 ***create_socket*** 函数创建 ping 所需的套接字

```
//! 创建套接字
enable_capability_raw();
if (hints.ai_family != AF_INET6)
    create_socket(&rts, &sock4, AF_INET, hints.ai_socktype, IPPROTO_ICMP,
                  hints.ai_family == AF_INET);
if (hints.ai_family != AF_INET) {
    create_socket(&rts, &sock6, AF_INET6, hints.ai_socktype, IPPROTO_ICMPV6, sock4.fd == -1);
    /* This may not be needed if both protocol versions always had the same value, but
     * since I don't know that, it's better to be safe than sorry. */
    rts.pmtudisc = rts.pmtudisc == IP_PMTUDISC_DO   ? IPV6_PMTUDISC_DO   :
                   rts.pmtudisc == IP_PMTUDISC_DONT ? IPV6_PMTUDISC_DONT :
                   rts.pmtudisc == IP_PMTUDISC_WANT ? IPV6_PMTUDISC_WANT : rts.pmtudisc;
}
disable_capability_raw();
```

创建套接字

配置套接字

```
//! 设置套接字选项
if (rts.settos)
    set_socket_option(&sock4, IPPROTO_IP, IP_TOS, &rts.settos, sizeof rts.settos);
if (rts.tclass)
    set_socket_option(&sock6, IPPROTO_IPV6, IPV6_TCLASS, &rts.tclass, sizeof rts.tclass);
```

配置套接字

根据 IPv4 或者 IPV6 对目标主机进行 ping

```
for (ai = result; ai; ai = ai->ai_next) {
    if (target_ai_family != AF_UNSPEC &&
        target_ai_family != ai->ai_family) {
        if (!ai->ai_next) {
            /* An address was found, but not of the family we really want.
             * Throw the appropriate gai error.
             */
            error(2, 0, "%s: %s", target, gai_strerror(EAI_ADDRFAMILY));
        }
        continue;
    }
    switch (ai->ai_family) {
    case AF_INET:
        ret_val = ping4_run(&rts, argc, argv, ai, &sock4);
        break;
    case AF_INET6:
        ret_val = ping6_run(&rts, argc, argv, ai, &sock6);
        break;
    default:
        error(2, 0, _("unknown protocol family: %d"), ai->ai_family);
    }
```

ping 目标主机

实验测试

课本 Ping 程序测试

测试 ping 百度服务器

```
❯ sudo ./a baidu.com
PING baidu.com (39.156.69.79) 56(84) bytes of data.
64 byte from 39.156.69.79: icmp_seq=0 ttl=46 rtt=36 ms
64 byte from 39.156.69.79: icmp_seq=1 ttl=46 rtt=25 ms
64 byte from 39.156.69.79: icmp_seq=2 ttl=46 rtt=24 ms
64 byte from 39.156.69.79: icmp_seq=3 ttl=46 rtt=25 ms
64 byte from 39.156.69.79: icmp_seq=4 ttl=46 rtt=26 ms
^C--- baidu.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4301ms
```

ping 百度服务器

编译内核 Ping 程序测试

查看帮助信息

```
❯ sudo ./builddir/ping/ping --help
./builddir/ping/ping: invalid option -- '-'

Usage
  ping [options] <destination>

Options:
  <destination>      dns name or ip address
  -a                 use audible ping
  -A                 use adaptive ping
  -B                 sticky source address
  -c <count>         stop after <count> replies
  -D                 print timestamps
  -d                 use SO_DEBUG socket option
  -f                 flood ping
  -h                 print help and exit
  -I <interface>     either interface name or address
```

查看帮助信息

ping 百度服务器测试

```
❯ sudo ./builddir/ping/ping baidu.com -4
PING baidu.com (39.156.69.79) 56(84) bytes of data.
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=1 ttl=46 time=25.4 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=2 ttl=46 time=38.0 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=3 ttl=46 time=26.0 ms
64 bytes from 39.156.69.79 (39.156.69.79): icmp_seq=4 ttl=46 time=25.4 ms
^C64 bytes from 39.156.69.79: icmp_seq=5 ttl=46 time=25.7 ms

--- baidu.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4575ms
rtt min/avg/max/mdev = 25.420/28.119/37.958/4.924 ms
```

ping **百度服务器**

指定 ttl 为 10 进行测试，可以发现无法到达

```
❯ sudo ./builddir/ping/ping -t 10 baidu.com
PING baidu.com (39.156.69.79) 56(84) bytes of data.
From 221.183.58.97 (221.183.58.97) icmp_seq=1 Time to live exceeded
From 221.183.58.97 (221.183.58.97) icmp_seq=2 Time to live exceeded
From 221.183.58.97 (221.183.58.97) icmp_seq=3 Time to live exceeded
From 221.183.58.97 (221.183.58.97) icmp_seq=4 Time to live exceeded
^CFrom 221.183.58.97 icmp_seq=5 Time to live exceeded
```

ttl **测试**

实验体会

本次实验我了解了原始套接字的编程方法，并通过对课本 ping 代码的分析理清楚了 ping 代码的基本框架，在此基础上，**对 linux 内核代码进行了分析**。

在对内核代码的分析过程中，我对 linux 中有关网络编程的数据结构更加熟悉了，并对读源码的技巧有了进一步的体会，不应该囿于每个参数的功能实现，这样很容易陷入层层递进的函数，而是应该从整体上来把握，先把实现的框架弄懂，当遇到需要理解的地方时再去深入探究其功能。

此外，通过这次实验，我熟悉了 ICMP 报文的数据结构，如何发送 ICMP 请求，解析 ICMP 回显应答以及如何利用 ***getopt*** 函数处理命令行参数，如何挂载信号处理函数。