LAB 1 Ping程序的实现

051720205 岳雨涵 1617101班

1 Socket知识复习/学习

由于上次写Socket还是今年四月份的时候，导致很多函数已经忘记了，在这里记录一下方便以后查阅。

- **struct sockaddr_in**

  此结构用做bind/connect/recvfrom/sendto等函数的参数，指明地址信息.

```
struct   sockaddr_in
{
    short int           sin_family;    /* 地址族,形如AF_xxx,通常用的是AF_INET,
2字节 */
    unsigned short int  sin_port;      /* 端口号（使用网络字节顺序）2字节 */
    struct in_addr      sin_addr;      /* 存储IP地址,4字节,就是32位的ip地址 */

    unsigned char        sin_zero[8]; /* 总共8个字节,实际上没有什么用,只是为了和
struct
                                        sockaddr保持一样的长度 */
};
```

  P.S. sockaddr_in和sockaddr是并列的结构，指向sockaddr_in的结构体的指针也可以指向sockadd的结构体，并代替它。可以使用sockaddr_in建立所需要的信息, 在最后用进行类型转换即可`bzero((char * )&mysock,sizeof(mysock));`

- **inet_addr()/inet_ntoa()**

  inet_addr 将字符串形式的IP地址 -> 网络字节顺序 的整型值，inet_ntoa 网络字节顺序的整型值 ->字符串形式的IP地址.

- **inet_pton()/inet_ntop()** 网络地址转化函数

  这两个函数是随IPv6出现的函数，对于IPv4地址和IPv6地址都适用，函数中p和n分别代表表达（presentation)和数值（numeric)。地址的表达格式通常是ASCII字符串，数值格式则是存放到套接字地址结构的二进制值。

```
#include <arpe/inet.h>
int inet_pton(int family, const char *strptr, void *addrptr); //将点分十进制的ip地址转化为用于网络传输的数值格式
                                                              //返回值：若成功则为1，若输入不是有效的表达式则为0，若出错则为-1

const char * inet_ntop(int family, const void *addrptr, char *strptr, size_t len);
                                                              //将数值格式转化为点分十进制的ip地址格式
                                                              //返回值：若成功则为指向结构的指针，若出错则为NULL
```

- **setsockpt()**

  摘自CSDN：

  在TCP连接中，recv等函数默认为阻塞模式(block)，即直到有数据到来之前函数不会返回，而我们有时则需要一种超时机制使其在一定时间后返回而不管是否有数据到来，这里我们就会用到setsockopt()函数：
  `int setsockopt(int s, int level, int optname, void* optval, socklen_t* optlen);`
  这里我们要涉及到一个结构：

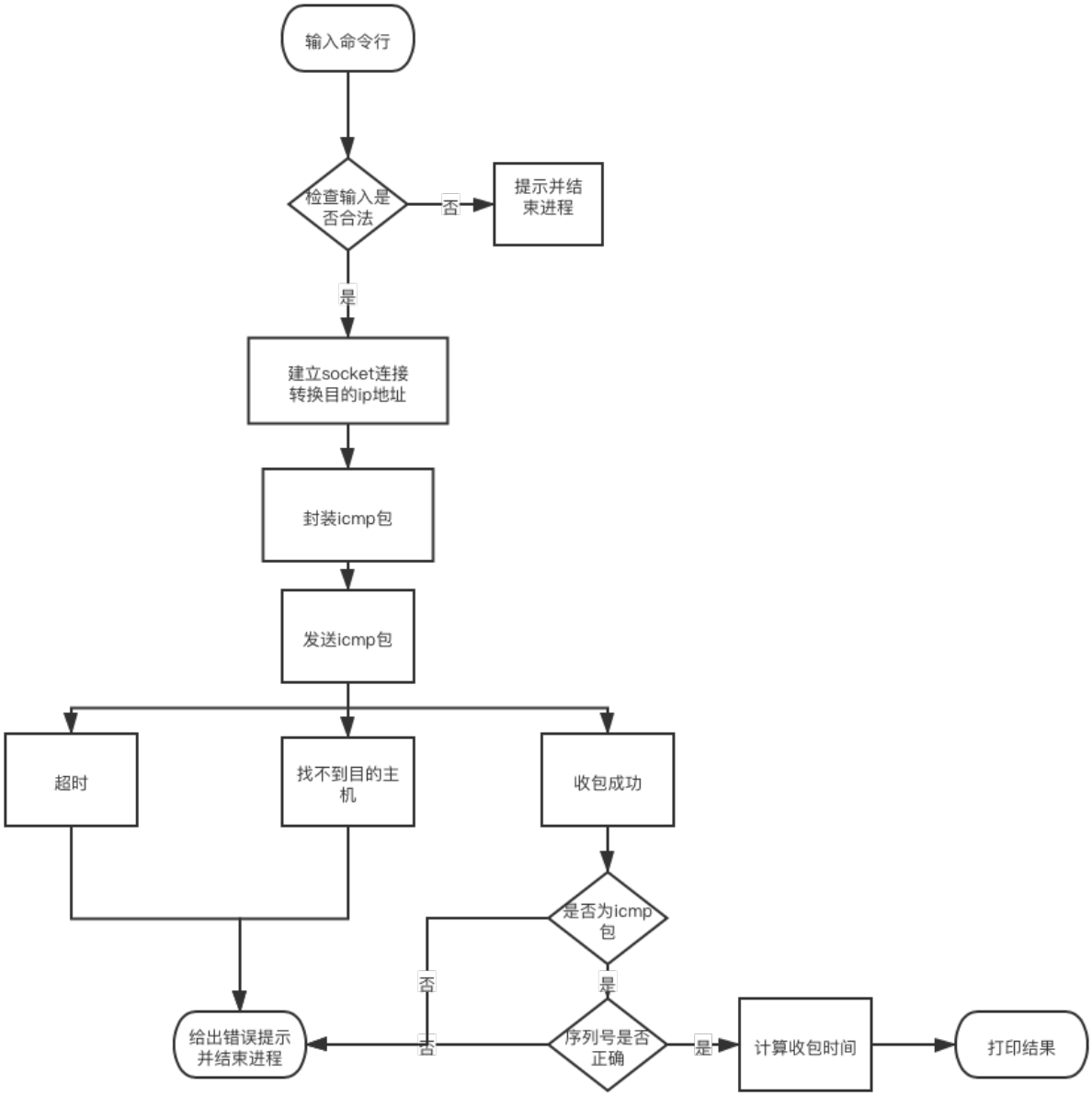
  ```
  struct timeval
  {
      time_t tv_sec;
      time_t tv_usec;
  };
  ```

  然后可以调用：`setsockopt(fd, SOL_SOCKET, SO_RCVTIMEO, &tv_out, sizeof(tv_out));`这样我们就设定了recv()函数的超时机制，当超过tv_out设定的时间而没有数据到来时recv()就会返回0值。

2 实现思路

总体思路：获取命令行 -> 检验参数正确性 -> 建立socket连接 -> 构造icmp报文并发送 -> 等待接受报文 ->检查接收包的情况-> 输出结果(success/fail)

3 代码结构



4 主要函数设计及功能

```
// 函数声明

int validNumber(char *);//判断输入参数正确性
int ping(char *, int, int);
void assembleIcmpPackage(struct icmp *, int, int, pid_t);//打包icmp包
unsigned short checkSum(unsigned short *, int);//校验和计算
struct timeval getOffsetTime(struct timeval, struct timeval);//计算从发送到收包的时间差
```

- **int validNumber(char *)**;

  在接收到形如`ping addressArg [-n] [sendTimes] [-l] [packageLength]`的命令行后，要判定发送次数和报文长度是否违法，违法则输出提示，若有效则返回所输入的参数.

- **int ping(char *, int, int);**

  1. 首先使用socket函数`socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);`建立套接字，并且设置缓冲区. ***(在send()的时候，返回的是实际发送出去的字节(同步)或发送到socket缓冲区的字节(异步);系统默认的状态发送和接收一次为8688字节(约为8.5K)；在实际的过程中发送数据和接收数据量比较大，可以设置socket缓冲区，而避免了send(),recv()不断的循环收发)***
  2. 接着进行ip地址的转换，将从命令行读入的点分十进制的ip地址用`inet_addr()`转化为用于网络传输的数值格式，并参照ping命令打印地址信息。
  3. 封装icmp包，具体实现`void assembleIcmpPakage()`函数；发送icmp包并计时。
  4. 接收icmp包的数据到缓冲区并拆解读取各部分数据，需判断传输的正确性例如当包长度小于8字节说明不是icmp包，是否是icmp回应包且是本机发送的。
  5. 打印结果。

- **void assembleIcmpPackage(struct icmp *, int, int, pid_t);**

  1. 参数：struct icmp * 为发送的包内容，其中包括imp_type/code/sckum/seq/id/data. sendBuffer起初全置零，在本函数中进行赋值. 第一个int为发送的icmp报文序号，从0开始计数；第二个int为发送的数据长度. pid_t是获取了当前进程号pid，用于作为icmp包头部id.
  2. 设置icmp包各部分的数值：icmp_code默认为0，icmp校验和默认置零，确认号为参数传入的序列号，从0开始计数，每发送一次加一；进程标识符作为id. 接着填充数据段，根据命令行获取的报文长度来填充.
  3. 计算校验和，具体实现见`unsigned short checkSum(unsigend short *, int);`

- **unsigned short checkSum(unsigned short *, int);**

  每16比特（2字节）相加，如果最后剩下一个字节，则要补全为两个字节，继续累加；最后结果取反.

- **struct timeval getOffsetTime(struct timeval, struct timeval);**

  用结束的时间剪去开始时间，得到从发送到接收icmp包的时间，用于打印结果.

5 源码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <unistd.h>
#include <signal.h>
#include <arpa/inet.h>
#include <errno.h>
#include <sys/time.h>
```

```c
#include <string.h>
#include <netdb.h>
#include <pthread.h>
#include <ctype.h>

// 调试模式
#define DEV_MODE 0

// 默认参数
#define DEFAULT_N 0
#define DEFAULT_L 56

// 缓冲区大小
#define SEND_BUFFER_SIZE 1024 * 1024
#define RECV_BUFFER_SIZE 1024 * 1024

// 超时设置
#define OUT_TIMEVAL_USEC 0
#define OUT_TIMEVAL_SEC 1000
#define TRY_TIME 3

// 函数声明
int ping(char *, int, int);
int validNumber(char *);
void assembleIcmpPackage(struct icmp *, int, int, pid_t);
struct timeval getOffsetTime(struct timeval, struct timeval);
unsigned short checkSum(unsigned short *, int);

int main(int argc, char *argv[]) {
    int i;
    int n = DEFAULT_N, l = DEFAULT_L;

    //获得指令
    if (argc < 2) {
        printf("Usage: ping addressArg [-n] [sendTimes] [-l]
[packageLength]\n");
        return 0;
    }

    for (i = 2; i < argc; i++) {
        if (!strcmp(argv[i], "-n") && i + 1 < argc) {
            //检验输入正确性
            n = validNumber(argv[i + 1]);
            //n = atoi(argv[i + 1]);
        }
        if (!strcmp(argv[i], "-l") && i + 1 < argc) {
            l = validNumber(argv[i + 1]);
            //l = atoi(argv[i + 1]);
        }
```

```
    }

    //错误提示
     if (n <= 0)
    {
        printf("-n count : you give an incorrect number (<= 0) \n");
        exit(0);
    }
    if (l <= 0 || l >= 64)
    {
        printf("-l length : you give an incorrect number(>=64 or <= 0)\n");
        exit(0);
    }

    ping(argv[1], n, l);
    return 0;
}


//检查输入参数是否正确
int validNumber(char *src)
{
    int len = strlen(src);
    for (int i = 0; i < len; i++)
        if (!isalnum(src[i]))
            return -1;
    return atoi(src);
}

int ping(char *addressArg, int n, int l) {
    if (DEV_MODE) {
        printf("n: %d\n", n);
        printf("l: %d\n", l);
    }


    //发送、接收报文
    char sendBuffer[SEND_BUFFER_SIZE], recvBuffer[RECV_BUFFER_SIZE];
    memset(sendBuffer, 0, sizeof(sendBuffer));
    memset(recvBuffer, 0, sizeof(recvBuffer));

    int count = 0, nt = n == 0 ? 1 : n;



    // 建立套接字
    //struct protoent* protocol = getprotobyname("icmp");
    int sock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sock < 0) {
```

```
        printf("Can't create socket.\n");
        exit(0);
    }

    // 设置接收缓冲区大小
    int recvBufferSize = RECV_BUFFER_SIZE;
    //设置套接字
    setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &recvBufferSize,
sizeof(recvBufferSize));

    // 改造 ip 地址
    struct sockaddr_in address;
    bzero(&address, sizeof(address));
    address.sin_family = AF_INET;
    unsigned int internetAddress = inet_addr(addressArg);
    /*
    if (internetAddress == INADDR_NONE) {
        // 如果输入的是域名地址
        struct hostent *host = gethostbyname(addressArg);
        if (host == NULL) {
            printf("Fail to get host name.\n");
            return 0;
        }

        memcpy((char *) &address.sin_addr, host->h_addr, host->h_length);
    } else {
        memcpy((char *) &address.sin_addr, &internetAddress,
sizeof(internetAddress));
    }*/
    memcpy((char *) &address.sin_addr, &internetAddress,
sizeof(internetAddress));
    internetAddress = address.sin_addr.s_addr;

    // 输出信息
    printf(
        "PING %s, (%d, %d, %d, %d) (%d)%d bytes of data.\n",
        addressArg,
        internetAddress & 0x000000ff,
        (internetAddress & 0x0000ff00) >> 8,
        (internetAddress & 0x00ff0000) >> 16,
        (internetAddress & 0xff000000) >> 24,
        l,
        l + 28
    );

        // 获取进程标识符
    pid_t pid = getpid();

    // 读文件描述符
```

```
    fd_set readFd;

    // 超时设置
    struct timeval tv;
    tv.tv_usec = OUT_TIMEVAL_USEC;
    tv.tv_sec = OUT_TIMEVAL_SEC;

    // 时间结构体
    struct timeval beginTime, endTime, offsetTime;

    // 定义收到标识符和超时计数器
    int get = 0;
    int currentTryTime = 0;

    while (nt > 0) {
        // 封装 icmp 包
        assembleIcmpPackage((struct icmp *) sendBuffer, ++count, l, pid);
        // 发送 icmp 包
        if (sendto(sock, sendBuffer, l + 8, 0, (struct sockaddr *) &address,
sizeof(address)) < 0) {
            printf("Send data fail.\n");
            break;
        }

        if (DEV_MODE) {
            printf("Send data success\n");
        }

        // 计时
        gettimeofday(&beginTime, NULL);

        // 开始收包
        FD_ZERO(&readFd);
        FD_SET(sock, &readFd);

        int recvSize;

        // 重置标识符
        get = 0;
        currentTryTime = 0;

        while (!get && currentTryTime <= TRY_TIME) {
            if (DEV_MODE) {
                printf("currentTryTime: %d\n", currentTryTime);
            }

            switch (select(sock + 1, &readFd, NULL, NULL, &tv)) {
                case -1://出错
                    currentTryTime = TRY_TIME;
```

```
                    printf("Fail to select\n");
                    break;
                case 0://超时
                    currentTryTime++;
                    break;
                default:
                    currentTryTime++;
                    recvSize = recv(sock, recvBuffer, sizeof(recvBuffer), 0);
                    // 接受数据到缓冲区
                    if (recvSize < 0) {
                        printf("Fail to receive data\n");
                        continue;
                    }

                    if (DEV_MODE) {
                        printf("recvSize: %d\n", recvSize);
                    }

                    // 解包
                    // 获取 ip 包头
                    struct ip *ipHeader = (struct ip *) recvBuffer;
                    // 获取 ip 包头长度 (ip_hl是以字节为单位)
                    int ipHeaderLength = ipHeader->ip_hl * 4;
                    // 获取 icmp 包头
                    struct icmp *icmpHeader = (struct icmp *) (recvBuffer +
ipHeaderLength);
                    // 获取 icmp 包长度
                    int icmpPackgeLength = recvSize - ipHeaderLength;

                    // 如果小于 8 字节说明不是 icmp 包
                    if (icmpPackgeLength < 8) {
                        printf("Invalid icmp package, because its length it
less than 8 byte\n");
                        continue;
                    }

                    if (DEV_MODE) {
                        printf("icmp->icmp_type: %d ICMP_ECHOREPLY:%d\n",
icmpHeader->icmp_type, ICMP_ECHOREPLY);
                        printf("icmp->icmp_id: %d pid: %d\n", icmpHeader-
>icmp_id, pid);
                        printf("icmp->icmp_seq: %d\n", icmpHeader->icmp_seq);
                    }

                    // 判断是 icmp 回应包而且是本机发的
                    if (icmpHeader->icmp_type == ICMP_ECHOREPLY && icmpHeader-
>icmp_id == (pid & 0xffff)) {
                        if (icmpHeader->icmp_seq < 0 || n != 0 && icmpHeader-
>icmp_seq > n) {
```

```
                            printf("Sequence of icmp package is out of
range.\n");
                            continue;
                        }

                        // 设置收到标识为 1
                        get = 1;

                        // 记下收包时间
                        gettimeofday(&endTime, NULL);
                        // 计算时差
                        offsetTime = getOffsetTime(beginTime, endTime);

                        if (DEV_MODE) {
                            printf("beginTime: %ds, %dus\n", beginTime.tv_sec,
beginTime.tv_usec);
                            printf("endTime: %ds, %dus\n", endTime.tv_sec,
endTime.tv_usec);
                            printf("offsetTime: %ds, %dus\n",
offsetTime.tv_sec, offsetTime.tv_usec);
                        }

                        // 输出结果
                        printf(
                            "%d byte from %s: icmp_seq=%u ttl=%d
rtt=%.3fms\n",
                            l + 8,
                            inet_ntoa(ipHeader->ip_src),
                            icmpHeader->icmp_seq,
                            ipHeader->ip_ttl,
                            offsetTime.tv_sec * 1000 + offsetTime.tv_usec *
1.0 / 1000
                        );
                    } else continue;

                    break;
                }
            }

            if (!get) {
                printf("Unreachable host.\n");
                return 0;
            }

            if (n != 0) nt--;
            sleep(1);
        }
}
```

```c
//打包Icmp报文
void assembleIcmpPackage(struct icmp *header, int sequence, int dataLength,
pid_t pid) {
    int i;

    // icmp类型：回送请求
    header->icmp_type = ICMP_ECHO;
    header->icmp_code = 0;
    header->icmp_cksum = 0;
    header->icmp_seq = sequence;
    header->icmp_id = pid & 0xffff;

    // 填充数据段
    for (i = 0; i < dataLength; i++) {
        header->icmp_data[i] = 1;
    }

    // 计算校验和
    header->icmp_cksum = checkSum((unsigned short *) header, dataLength + 8);
}

struct timeval getOffsetTime(struct timeval beginTime, struct timeval endTime)
{
    struct timeval offsetTime;

    offsetTime.tv_sec = endTime.tv_sec - beginTime.tv_sec;
    offsetTime.tv_usec = endTime.tv_usec - beginTime.tv_usec;

    if (offsetTime.tv_usec < 0) {
        offsetTime.tv_sec--;
        offsetTime.tv_usec += 1000000;
    }

    return offsetTime;
}


unsigned short checkSum(unsigned short *header, int length) {
    int count = length;
    int sum = 0;
    unsigned short *t = header;
    unsigned short result = 0;

    while (count > 1) {
        sum += *t++;
        count -= 2;
    }

    if (count == 1) {
```

```
        sum += * (unsigned char *) t;
    }

    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    result = ~sum;
    return result;
}
```