2019秋计算机网络BackTcp协议设计实验报告

学号：PB17000023 **王家伟**

2019秋计算机网络BackTcp协议设计实验报告
- 一、实验内容
- 二、实验环境
- 三、实现步骤
  - 综述
  - 数据封装
    - 数据包结构
    - 封装函数
  - 发送方
    - 多线程
    - socket创建连接和关闭
    - 定时器实现
    - rdt_send函数
    - rdt_rcv函数
    - timeout函数
  - 接收端
- 四、实验结果展示
  - 发送端
  - 接收端
  - 测试信道
  - 数据文本对比
- 结语

一、实验内容

backTCP 的目标是实现面向无连接的可靠传输功能， 能够解决数据包在传输过程中出现的乱序以及丢包问题。 由于考虑的是无连接的网络，因此该作业不需要考虑传统 TCP 中的三次握手连接建立过程。此外， 假设传输中数据不会出现错误，因此只需要考虑如何解决数据包的乱序和丢包问题 。

二、实验环境

Ubuntu 16.04 LTS

C++

Python

三、实现步骤

综述

本次实验中，我使用了c++语言实现backTCP的服务器端和客户端，服务端主要用到了Go_back_N算法、定时器以及多线程技术，客户端则被动接受。

数据封装

数据包结构

```
typedef struct tcphdr {
    uint8_t btcp_sport;
    uint8_t btcp_dport;
    tcp_seq btcp_seq; //序列号
    tcp_seq btcp_ack; //确认号
    uint8_t data_off; //偏移地址
    uint8_t win_size; //窗口大小 用来流量控制 N
    uint8_t flag; //是否为重传包
} BTcpHeader;//size:7 bytes

typedef struct package
{
    BTcpHeader header;//7bytes
    char payload[LOAD_SIZE];//LOAD_SIZE = 64
}package;//fix the size:71
```

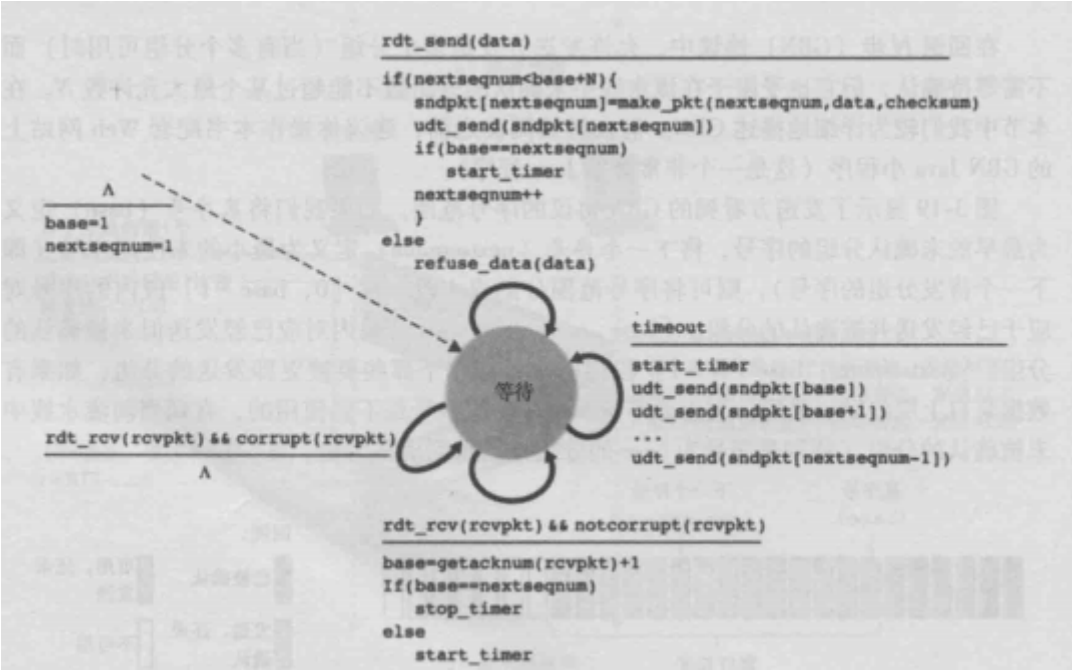
这里payload负载我直接使用了一个64bytes的数组，因为包头有data_off偏移地址，所以为了更优雅的实现，不需要动态分配负载大小。

封装函数

```
package* data_package(tcp_seq btcp_seq,tcp_seq btcp_ack,uint8_t data_off,uint8_t
win_size,uint8_t flag,char* payload)
{
    package* newPackage = (package*)malloc(sizeof(package));
    newPackage->header.btcp_sport = 0;
    newPackage->header.btcp_dport = 0;
    newPackage->header.btcp_seq = btcp_seq;
    newPackage->header.btcp_ack = btcp_ack;
    newPackage->header.data_off = data_off;
    newPackage->header.win_size = win_size;
    newPackage->header.flag = flag;
    for (uint8_t i = 0; i < data_off; ++i)
    {
        newPackage->payload[i] = payload[i];
    }
    return newPackage;
}
```

封装函数没什么好说的，只要把相应的内容填入包结构体即可。但由于我们不需要在端口标识进程，所以就直接填0了。

发送方



此图是Go_back_N的FSM，只有一个状态，但是需要并行运行发送和接受函数，因此需要多线程技术使得能够同时运行发送函数和接受函数，并进行相应的处理。

多线程

```
    pthread_t threads[2];
    int s = 0;
    int rc;
    while(1)
    {
        if(s == 0)
        {
            input_send in = {.fp = fp,.clnt_sock = clnt_sock};
            rc = pthread_create(&threads[s], NULL, rdt_send, (void *)&in);
            if (rc){
                cout << "Error:无法创建线程," << rc << endl;
                exit(-1);
            }
            s++;
        }
        else if(s == 1)
        {
            rc = pthread_create(&threads[s], NULL, rdt_rcv, (void *)&clnt_sock);
            if (rc){
                cout << "Error:无法创建线程," << rc << endl;
                exit(-1);
            }
            s++;
        }
        if(result_send == EMPTY)
        {
            finish = 1;//标志结束
            break;
        }
    }
```

这是主函数体，在开始阶段，创建了两个线程，分别运行rdt_send和rdt_rcv。标志finish用于结束线程。

socket创建连接和关闭

本次实验除了实现go_back_N等算法，也考察了学生对socket编程的掌握。

```
    //创建套接字
    int serv_sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);  //参数 AF_INET 表示使用 IPv4 地址，SOCK_STREAM 表示使用面向连接的套接字，IPPROTO_TCP 表示使用 TCP 协议
    //将套接字和IP、端口绑定
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));  //每个字节都用0填充
    serv_addr.sin_family = AF_INET;  //使用IPv4地址
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");  //具体的IP地址
    serv_addr.sin_port = htons(1230);  //端口
    bind(serv_sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    //进入监听状态，等待用户发起请求
    listen(serv_sock, 20);
    //接收客户端请求
    struct sockaddr_in clnt_addr;
    socklen_t clnt_addr_size = sizeof(clnt_addr);
    clnt_sock = accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
```

以上是socket的创建连接过程。

```
    shutdown(clnt_sock,SHUT_WR);
    close(clnt_sock);
    close(serv_sock);
```

以上是关闭过程

定时器实现

定时器使用linux下的 sys/time.h

```
void start_timer()
{
    cout<<"start"<<endl;
    struct itimerval itv;
    itv.it_interval.tv_sec = 1;
    itv.it_interval.tv_usec = TIME_INTERVAL;
    itv.it_value.tv_sec = 1;
    itv.it_value.tv_usec = TIME_INTERVAL;
    setitimer(ITIMER_REAL, &itv, &oldtv);
}

void stop_timer()
{
    cout<<"end"<<endl;
    struct itimerval itv;
    itv.it_interval.tv_sec = 0;
    itv.it_interval.tv_usec = TIME_INTERVAL;
    itv.it_value.tv_sec = 0;  //不触发
    itv.it_value.tv_usec = 0;
    setitimer(ITIMER_REAL, &itv, &oldtv);
}

void signal_handler(int m)
{
    timeout(clnt_sock);
}
```

调用start_timer时，设置定时器间隔值，并且此定时器也会在超时时执行signal_handler函数。stop_timer函数是我自己写的一个trick，即将间隔值设成0，它会不触发定时器，也就是关闭。

rdt_send函数

```
void* rdt_send(void* args)
{
    input_send in = *((input_send*)args);
    FILE* fp = in.fp;
    int clnt_sock = in.clnt_sock;
    while(1)
    {
        if(finish)
            break;
        char buffer[BUF_SIZE];
        int nCount;
        if((nextseqnum>=SEQ_MAX-WIN_SIZE&&nextseqnum<base + WIN_SIZE)||
(base+WIN_SIZE<SEQ_MAX&&nextseqnum<base+WIN_SIZE)||
(nextseqnum<(base+WIN_SIZE)%SEQ_MAX)){
            //cout<<"yes"<<endl;
            if((nCount = fread(buffer,1,LOAD_SIZE,fp))>0)
            {
                package_buffer_send[nextseqnum] =
data_package(nextseqnum,0,nCount,WIN_SIZE,0,buffer);
                cout<<"send window:"<<base<<' '<<nextseqnum<<endl;
                write(clnt_sock,
(void*)package_buffer_send[nextseqnum],sizeof(package));
            }
            else
            {
                //finish close
                start_timer();
                cout<<"EMPTY"<<endl;
                result_send = EMPTY;
            }
            if(base == nextseqnum)
            {
                start_timer();
                //start_timer
            }
            nextseqnum = (nextseqnum+1)%SEQ_MAX;
        }
        else
        {
            //cout<<"nextseqnum:"<<nextseqnum<<endl;
            result_send = FULL;//full of window
            //do nothing
        }
        usleep(1000);
    }
    pthread_exit(NULL);
}
```

此函数是发送函数，也是根据FSM发送端设计的线程函数。一个值得注意的细节是窗口序号是有限的，需要取模运算，所以在判断序号是否在窗口内需要进行判断设计。

rdt_rcv函数

```
void* rdt_rcv(void* args)
{
    int clnt_sock = *((int*)args);
    while(1)
    {
        if(finish)
            break;
        char buffer[sizeof(package)];


        int nCount;
        package* recvPackage;
        if((nCount = read(clnt_sock, buffer, sizeof(package)))>0)
        {
            printf("recv!\n");
            recvPackage = (package*)buffer;
            base = (recvPackage->header.btcp_ack+1)%SEQ_MAX;
            cout<<"base:"<<base<<endl;
            if(base == nextseqnum)
            {
                stop_timer();
                //stop_timer
            }
            else{
                start_timer();
                //start_timer
            }
            result_recv = recvPackage->header.btcp_seq;
        }
        usleep(1000);
    }
    pthread_exit(NULL);
}
```

timeout函数

```
void timeout(int clnt_sock)
{
    //start_timer
    cout<<"timeout"<<endl;
    if(base >=nextseqnum)
    {
        stop_timer();
        package_buffer_send[nextseqnum-1]->header.flag = 1;
        package_buffer_send[nextseqnum-1]->header.btcp_seq = nextseqnum;
        package_buffer_send[nextseqnum-1]->header.data_off = 0;
        write(clnt_sock,(void*)package_buffer_send[nextseqnum-1],sizeof(package));
    }
    for (int i = base; i < nextseqnum; ++i)
    {
        cout<<"send again:"<<i<<endl;
        package_buffer_send[i]->header.flag = 1;
        write(clnt_sock,(void*)package_buffer_send[i],sizeof(package));
    }
}
```

接收端

接收端相比于发送端简单太多了，只需要考虑接收到的包是否失序，发送怎样的确认包。

```
    while((nCount = read(sock,buffer,sizeof(package)))>0)
    {
            recvPackage = (package*)buffer;
            cout<<"expectedseqnum:"<<expectedseqnum<<endl;
            if(recvPackage->header.btcp_seq < expectedseqnum)
            {
                cout<<"ack old"<<endl;
                package* sndpkt = data_package(expectedseqnum,recvPackage-
>header.btcp_seq,0,WIN_SIZE,0,NULL);
                write(sock,(void*)sndpkt,sizeof(package));
            }
            if(recvPackage->header.btcp_seq == expectedseqnum)
            {
                fwrite(recvPackage->payload,recvPackage->header.data_off,1,fp);
                package* sndpkt = data_package(expectedseqnum,recvPackage-
>header.btcp_seq,0,WIN_SIZE,0,NULL);
                write(sock,(void*)sndpkt,sizeof(package));
                cout<<expectedseqnum<<endl;
                expectedseqnum = (expectedseqnum+1)%SEQ_MAX;
            }
    }
```

以上是接收端的主体，只需要不断接受即可。如果接收到的包的序号比期待的要小（已经接收过），就发送一个对此序号的确认包；如果接收到的包序号就是期待的序号，则提取数据到文件。

四、实验结果展示

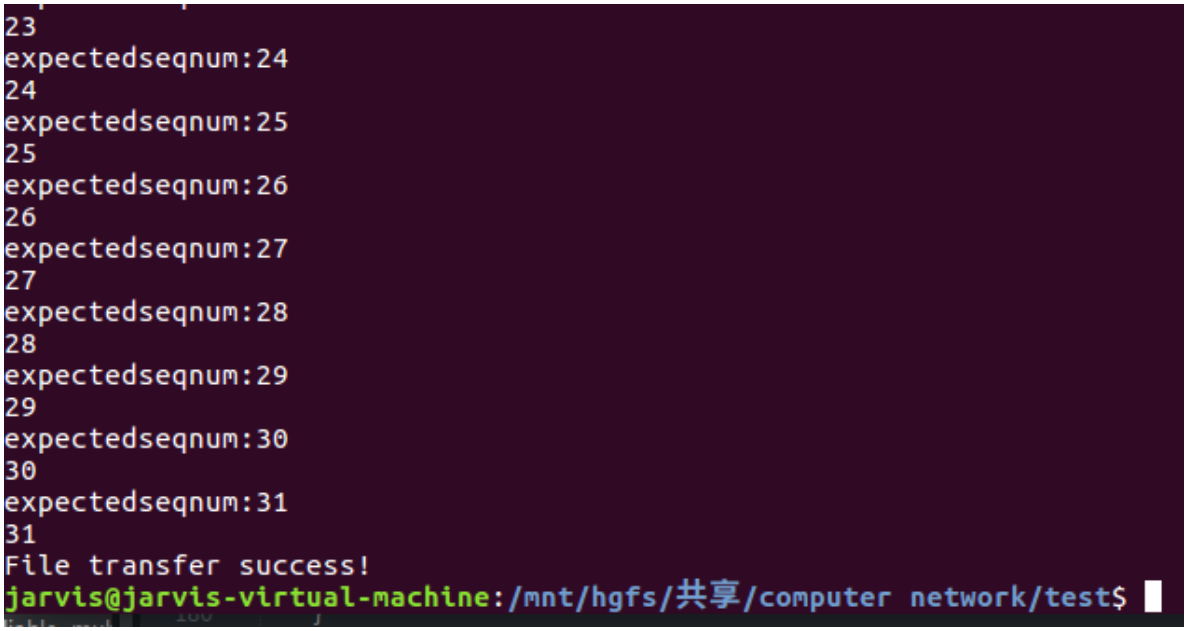
主要的实验结果是最终两个数据文档是否一致，但是我也在程序运行中间输出一些信息，也值得参考。

发送端

```
send window:22 28
recv!
base:24
start
send window:24 29
recv!
base:27
start
send window:27 30
recv!
base:26
start
send window:26 31
start
EMPTY
jarvis@jarvis-virtual-machine:/mnt/hgfs/共享/computer network/test$
```

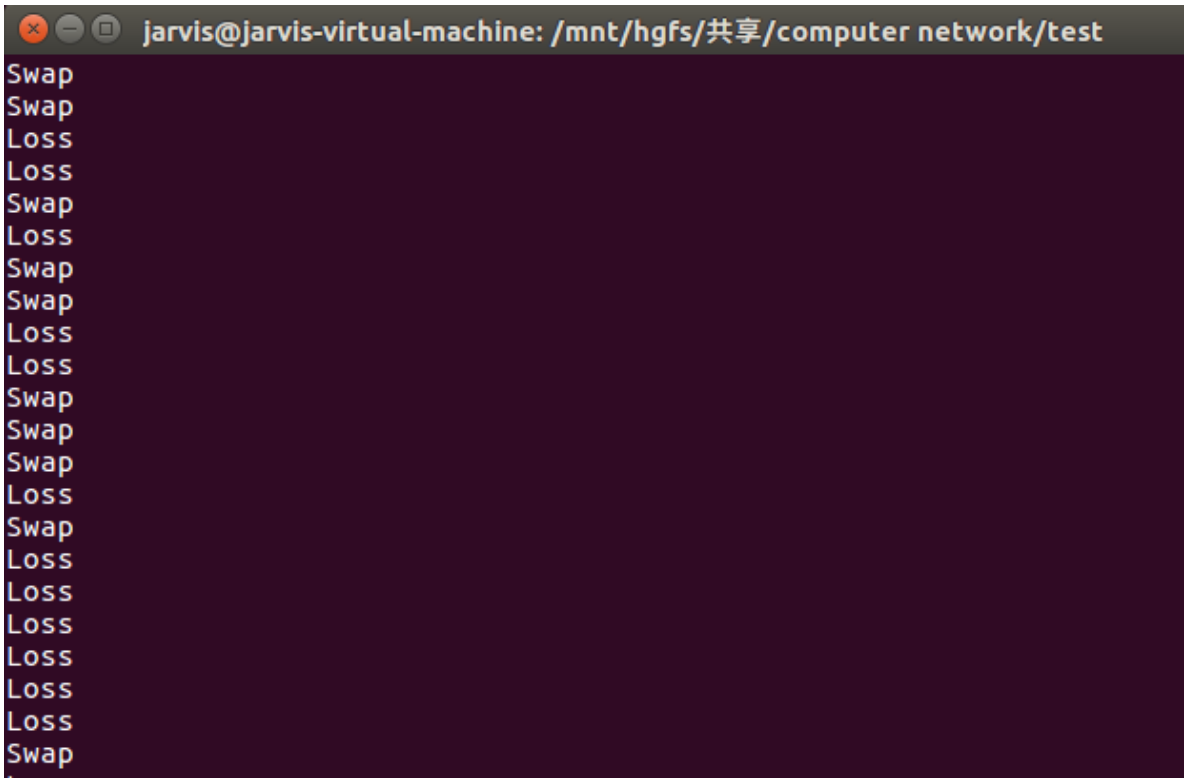
在我的实现下可以看到发送窗口在变化，由于是多线程，两个线程的运行速度不同，所以这里输出有些乱序，但整体可以看到过程是完整的。

接收端



在我的实现下接收端每次都能收到期待的序号，不会出现接受不到的情况。

测试信道



测试信道在不断的丢包和乱序，说明我实现的协议需要完成要求的任务。

数据文本对比

数据文本对比是实现是否正确最好的体现，我将接收到的文本文件和源文件对比，可以看到完全一致！因此实现是正确的！

结语

本次实验考验同学们的学习能力和思维能力，用到的socket编程、多线程、定时器等都是我们之前基本很少用到的，很有助于我们编程能力的提升。这次实验我也是使用了c++完成，虽然c++实现可能要求的细节比较多，但是很有助于我对这个协议的理解。

我会将源文件打包到压缩包，助教们可以查看压缩文档中readme.txt对我的程序进行测试。