实验3-2基于UDP服务设计可靠传输协议

姓名: 孟笑朵
学号: 2010349

实验要求

在实验3-1的基础上，将停等机制改成`基于滑动窗口的流量控制机制`，采用`固定窗口大小`，`支持累积确认`，完成给定测试文件的传输。

滑动窗口部分:
本实验参考TCP的滑动窗口设计, 并结合GBN协议和SR协议, 实现了如下滑动窗口机制:

1. 发送端实现固定窗口设计(因为数据包为固定大小的数据包, 不需要计算传输字节数);
2. 重传需要从错误发生的数据包开始，进行回退, 全部重传;
3. 对数据包进行计时, 超时数据包进行重传;

累计确认机制:
本实验参考TCP的累计确认机制, 并结合SACK协议设计了如下累积确认机制:

1. 收到数据包如果序列号不是连续的序列号, 重传上一个序列号的ACK;
2. 接受到新的数据包会覆盖之前获得的数据包;

建立连接为三次握手模式🤝, 断开连接为两次挥手模式👋与实验3-1一致, 这里不做赘述.

参考资料

1. [4.17 如何基于 UDP 协议实现可靠传输？ | 小林coding (xiaolincoding.com)](#)
2. [4.2 TCP 重传、滑动窗口、流量控制、拥塞控制 | 小林coding (xiaolincoding.com)](#)

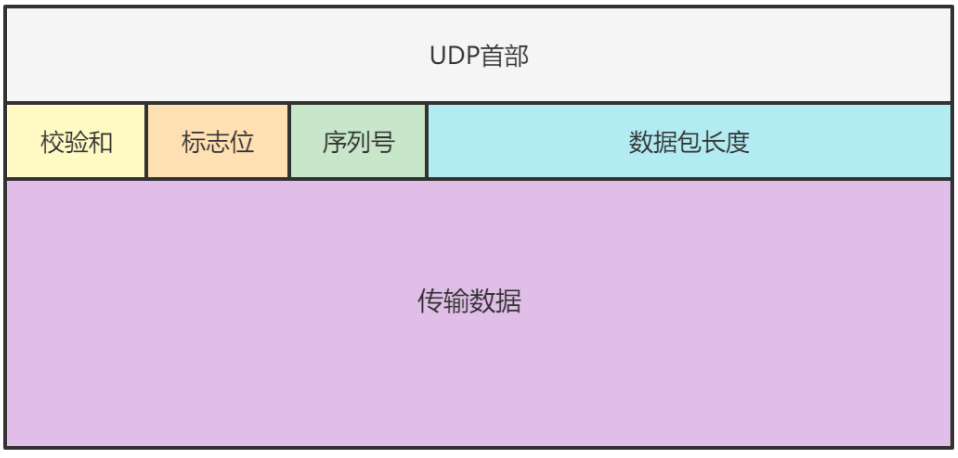
实验过程

协议设计

在协议设计部分, 与实验3-1协议设计部分一致.

我们知道UDP传输本身有自己的数据报格式, 如下图所示

另外, UDP在发送和接收数据报的时候会产生**伪首部**, 用于计算校验和, 伪首部包含了源IP地址和目的IP地址, 以及协议类型等字段, 在这里为了专门检验传输数据部分, 我们专门计算了针对当前传输数据的校验和, 在UDP本身数据报报头的基础上, 我们在数据部分增加专门用于可靠传输的协议头部分, 如下所示为当前数据报报头:



说明

1. 校验和: 一个字节, 专门用于计算标志位数据部分的校验和;
2. 标志位: 一个字节, 包含三次连接的SYN, ACK, 两次挥手的FIN, 数据传输的ACK, 数据传输是否为最后一个数据报的END标志位;
3. 序列号: 一个字节, 数据传输的SEQ
4. 数据包长度: 传输数据为可选长度, 标记传输数据的长度, 4个字节(也就是说Data部分不能超过4个字节表示的数据范围)

```C++
//设计报文格式
struct msgHead {
    unsigned char checknum;
    unsigned char flag;//标志位
    unsigned char seq; //表示数据包的序列号，一个字节为周期
    int length; //表示数据包的长度（包含头）
};
```

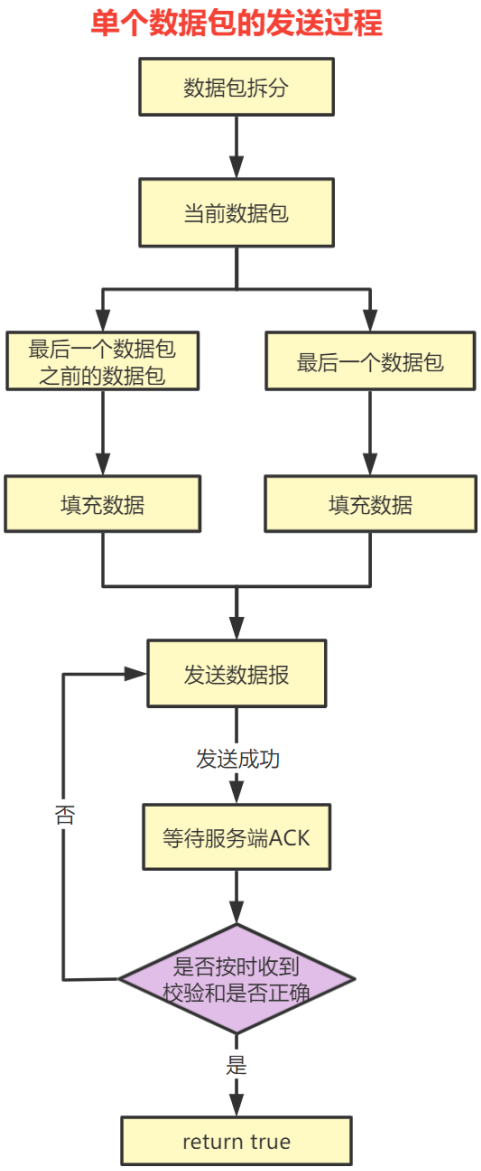
滑动窗口设计

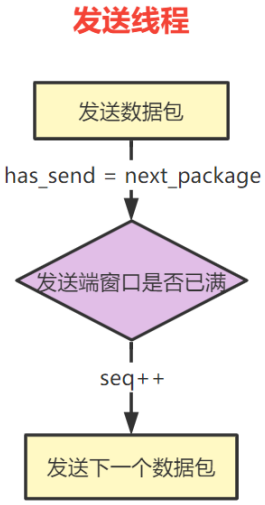
重点说明客户端进行数据发送时的滑动窗口的设计, 如下是滑动窗口设计时需要用到的一些变量:

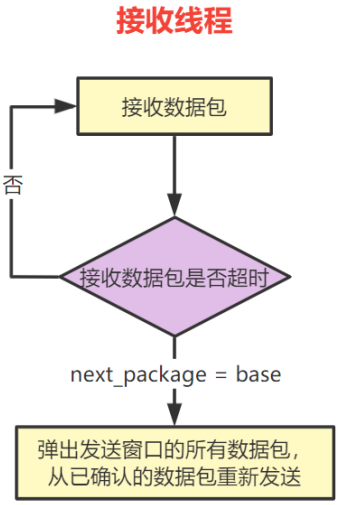
- **WINDOW_SIZE**: 窗口的大小, 本次实验取窗口大小为7, 为常量;
- **TIMEOUT**: 超时时间, 本次实验选取超时时间为200ms, 为常量;
- **package_num**: 总的数据包的数目, 依据传输数据的大小确定;
- **window**：发送端窗口, 这里设置的是queue数据结构, 先入先出;
- **base**：窗口底部，代表数据包个数小于base的部分已经全部传输完毕，且收到了对方返回的对应ACK，得到了确认;
- **has_send**: 代表已经发送的数据包序列号;
- **next_package**: 代表下一个要发送端数据包;
- **ack_send**: 代表已经确认的数据包的序列号;

为了解决超时的情况, 客户端采用**非阻塞**的接收消息方式, 此时可以收发数据包同时进行, `setsockopt(m_ClientSocket, SOL_SOCKET, SO_RCVTIMEO, (char*)&TIMEOUT, sizeof(int));`

当一个分组丢失或被破坏(发送超时)，发送方要重新发送所有未经确认的分组。在这部分进行接收状态判断时，将套接口设置为非阻塞模式, 保证套接口在没有接收到数据包时可以立即返回错误信息而不会一直停留在等待数据包来临的状态。







接收端处理流程

关键代码如下所示

```C++
//发送数据函数
//涉及到滑动窗口和累计确认机制
/*                发送方的可用窗口
                  |
+ + + + + + + + [+ + + + + + + + ] + + + + + + + +
                  |           | |
                base has_send next
*/
void send_msg(char* message, int length) {
    int timeout_num = 0;
    //1. 确定窗口的base值
    static int base = 0;
    //2. 确定当前传输到的seq
```

```
    int has_send = 0;
    int next_package = base;
    //3. 确定已经确认的seq
    int ack_send = 0;
    int package_num = length / MAX_UDP_LEN + (length % MAX_UDP_LEN != 0);
    //4. 将包保存到窗口当中
    queue<int> window;
    while (1) {
        //确认完毕所有数据包
        if (ack_send == package_num)
            break;
        //发送窗口
        while (window.size() < WINDOW_SIZE && has_send != package_num) {
            send_single_package(message + has_send * MAX_UDP_LEN,
                has_send == package_num - 1 ? length - (package_num - 1)
* MAX_UDP_LEN : MAX_UDP_LEN,
                next_package % ((int)UCHAR_MAX + 1),
                has_send == package_num - 1);
            window.push(next_package % ((int)UCHAR_MAX + 1));
            //将当前数据包的放入窗口中
            printf("%s%d\n", "sending seq:", next_package %
((int)UCHAR_MAX + 1));
            printf("%s%d\n", "发送窗口大小:", WINDOW_SIZE -
window.size());
            next_package++;
            has_send++;
        }
        char recv[3];
        int lentmp = sizeof(m_ServerAddress);
        //cout << "正在监听" << endl;
        setsockopt(m_ClientSocket, SOL_SOCKET, SO_RCVTIMEO,
(char*)&TIMEOUT, sizeof(int));
        if (recvfrom(m_ClientSocket, recv, 3, 0,
(sockaddr*)&m_ServerAddress, &lentmp) != SOCKET_ERROR && check_sum(recv,
3) == 0 &&
            recv[1] == ACK ) {
            //接收端返回最后一个确认的数据包，在此之前的都已经确认
            while (window.front() != (unsigned char)recv[2]) {
                printf("%s%d\n", "已经确认: ", (unsigned char)recv[2]);
                //确认包数+1
```

```
                //窗口base+1
                ack_send++;
                base++;
                window.pop();
            }
            printf("%s%d\n", "已经确认: ", (unsigned char)recv[2]);
            ack_send++;
            base++;
            timeout_num = 0;
            window.pop();
        }
        else {
            next_package = base;
            timeout_num++;
            has_send -= window.size();//将发送数据包回退
            while (!window.empty()) window.pop();
        }
        //同一个包丢5次的处理模式
        if (timeout_num >= 5) {
            return;
        }
    }
}
```

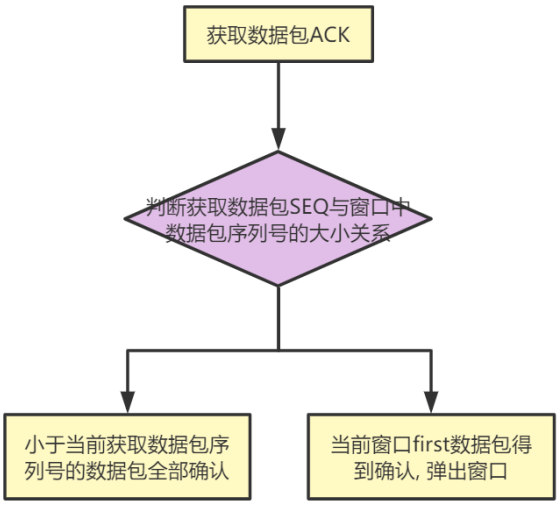
累计确认机制

GBN协议对确认帧采用累积确认的方式，返回ACK = n 时，表示n号帧以及n号帧之前的帧都已经收到了。当一个分组丢失或被破坏(发送超时)，发送方要重新发送所有未经确认的分组。

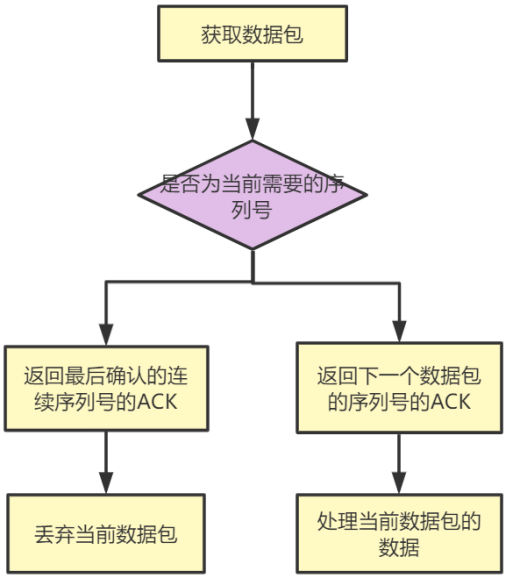
对于接收方, 当接收到顺序需要的下一个序列号的数据包时, 处理数据包, 当接收到失序的数据包时, 丢弃该数据包然后重新传输上次确认数据包的下一个序列号的ACK.

按照上述原理, 设计下图所示的累计确认的发送端处理机制和接收端处理机制:

累计确认发送端处理流程



累计确认接收端处理流程



关键代码如下所示

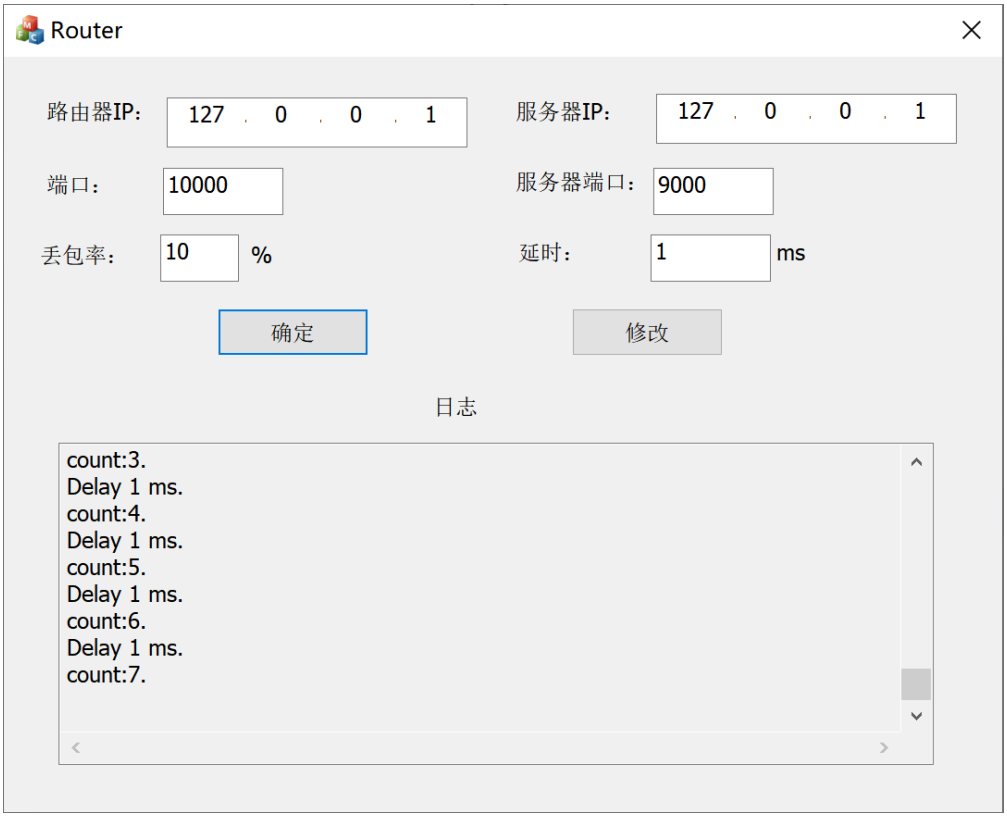
```C++
void recv_message(char* message, int& len_recv) {
    char recv[MAX_UDP_LEN + 4];
    int lentmp = sizeof(m_ClientAdress);
    //接收端的seq序列
    static unsigned char rec_seq = 0;
    len_recv = 0;
    int has_send_c = 0;
    int send_ok_c = 0;
    while (1) {
        while (1) {
            memset(recv, 0, sizeof(recv));
            while (recvfrom(m_ServerSocket, recv, MAX_UDP_LEN + 4, 0,
(sockaddr*)&m_ClientAdress, &lentmp) == SOCKET_ERROR);
            char send[3];
            int flag = 0;
            //发送ACK确认的序列号
            if (check_sum(recv, MAX_UDP_LEN + 4) == 0 && (unsigned
char)recv[2] == rec_seq) {
                printf("%s%d\n", "接收端确认序列号: ", rec_seq);
                rec_seq++;
                flag = 1;
                send[1] = ACK;
                send[2] = rec_seq - 1;
```

```
                send[0] = check_sum(send + 1, 2);
                sendto(m_ServerSocket, send, 3, 0,
(sockaddr*)&m_ClientAdress, sizeof(m_ClientAdress));
            }
            else if (check_sum(recv, MAX_UDP_LEN + 4) == 0 && (unsigned
char)recv[2] != rec_seq) {
                printf("%s%d\n", "接收端确认序列号: ", rec_seq);
                send[1] = ACK;
                send[2] = rec_seq;
                send[0] = check_sum(send + 1, 2);
                sendto(m_ServerSocket, send, 3, 0,
(sockaddr*)&m_ClientAdress, sizeof(m_ClientAdress));
            }
            if (flag)
                break;
        }
        // 接收端将消息放入缓存区中
        if (END == recv[1]) {
            for (int i = 4; i < recv[3] + 4; i++)
                message[len_recv++] = recv[i];
            break;
        }
        else {
            for (int i = 3; i < MAX_UDP_LEN + 3; i++)
                message[len_recv++] = recv[i];
        }
    }
}
```

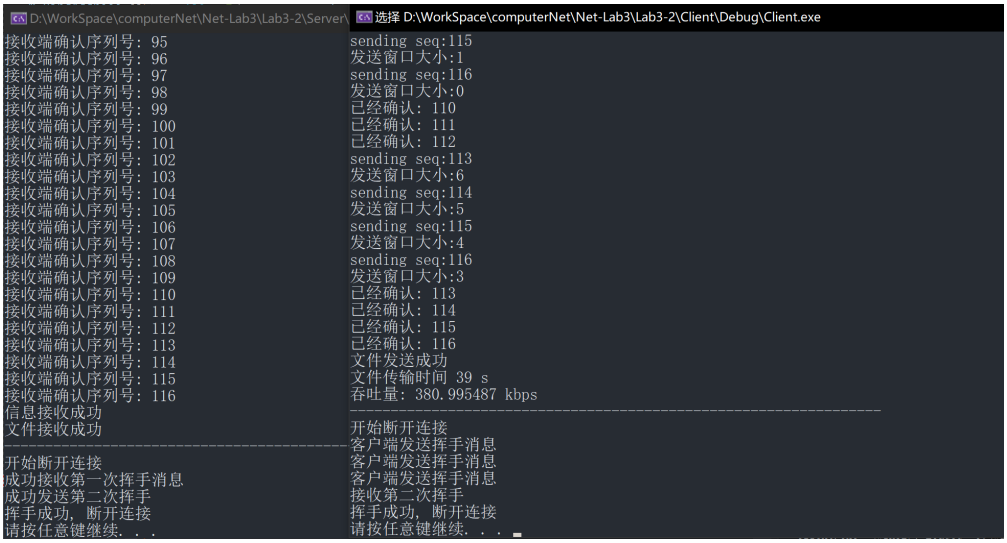
程序运行界面

程序运行方式:

1. 按照下图所示设置好路由文件的路由器IP地址和服务器IP地址



2. 先运行服务端, 再运行客户端;
3. 将对应的素材放置于Client.exe所在文件夹中, 在客户端运行输入对应文件名称即可看到程序传输运行状态, 结束后在对应的Server.exe所在文件夹中, 可以看到对应传输成功的文件.



路由程序小bug说明

在进行实验的过程中, 我发现路由程序的随机丢包设置有问题, 当设置丢包率为10%时, 我发现总是会丢第9个包, 不过不影响最后程序的运行.