



南开大学
Nankai University

计算机网络实验三

基于**UDP**服务设计可靠传输协议并编程实现

实验3-3：拥塞控制Reno

姓名：刘伟

学号：2013029

专业：物联网工程

一、实验要求

本实验建立在实验3-1、3-2的基础上，故一些功能实现未在本报告中加以解释。

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：**建立连接、差错检测、确认重传**等。基于**滑动窗口的流量控制机制**，支持**累积确认**，实现**拥塞控制算法**完，成给定测试文件的传输。

本次实验3-3建立在实验3-2的基础上，实现一种拥塞控制算法（参考于**Reno算法**），完成数据的传输。

二、实验协议设计&功能实现

1. 报文格式：

报文设计建立在实验3-1的基础上，在本次实验中使用到了3-1中未使用的window窗口部分，实现了滑动窗口的流量控制机制。

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
datasize数据长度															
			END	OVER	FIN	ACK	SYN	window							
seq序列号								ack确认号							
checksum校验和															

- **24-31位**，标记为**window窗口**，标定了当前滑动窗口的大小。在实验设计中：发送方在**window**中填写**当前滑动窗口大小**：目的是便于观察当前数据传输处于什么样的阶段以及便于比较滑动窗口与阈值窗口之间的关系。

其余见3-1、3-2实验设计报告

2.以下部分可参见实验3-1、3-2设计报告

1. 建立连接&断开连接方式：仿照TCP协议
2. 差错检验：
3. 双向传输：
4. 流水线协议
5. 基于滑动窗口的流量控制：
6. GBN数据传输
7. 累计确认&确认重传：
8. 超时重传：

3. 拥塞控制&拥塞窗口

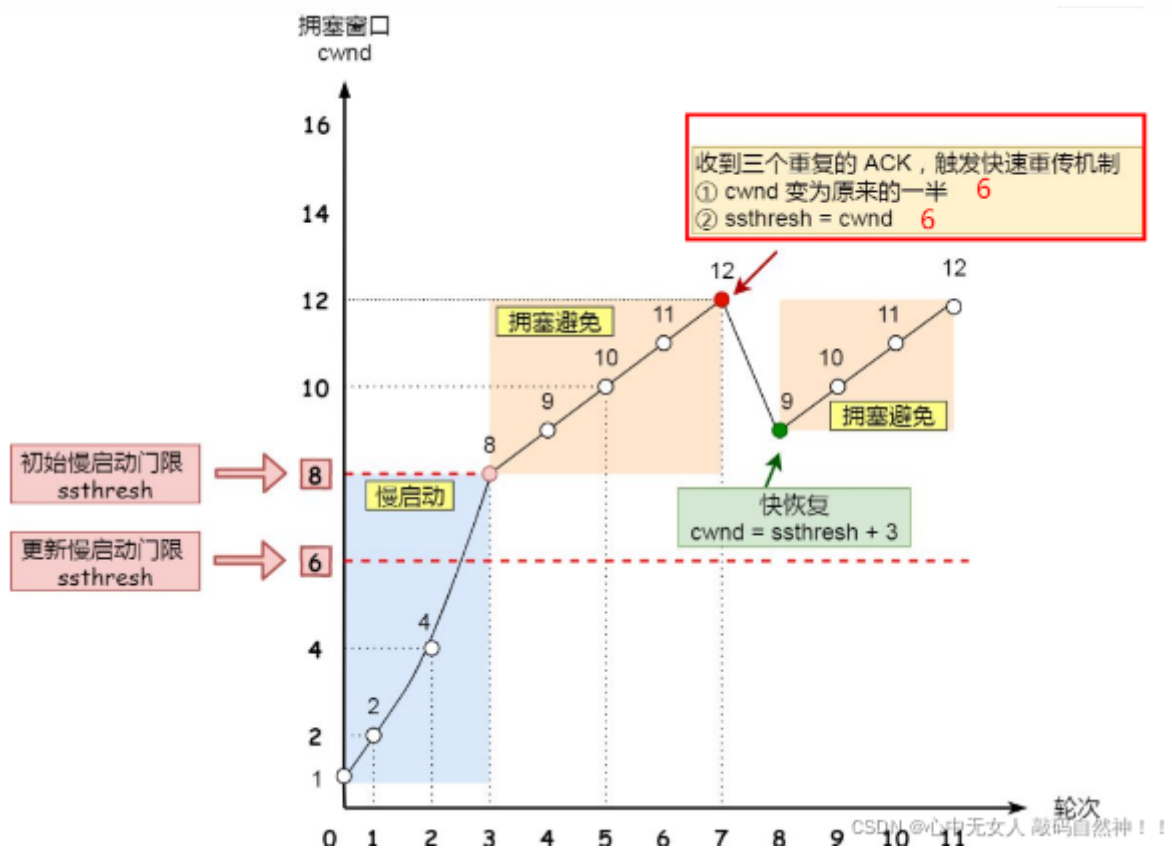
拥塞控制，就是在网络中发生拥塞时，减少向网络中发送数据的速度，防止造成恶性循环；同时在网络空闲时，提高发送数据的速度，最大限度地利用网络资源。这里数据的发送速度便通过 **拥塞窗口** 的大小变化进行调节。

发送方维持一个拥塞窗口cwnd的状态变量，目的是为了控制注入到网络中的数据量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。

发送方控制拥塞窗口的原则是：只要当前网络中没有出现拥塞，拥塞窗口就可以再增大一些，以便把更多的分组发送出去。但只要网络出现拥塞，拥塞窗口就应该减小一些，以减少注入到网络中的分组数。

4. Reno拥塞控制算法

Reno机制状态切换示意图：



- Slow Start（慢启动）

当cwnd的值小于ssthresh（ssthresh: slow start thresh, 慢启动门限值）时，发送方处于slow start阶段。该阶段下每收到一个ACK，cwnd的值就会加1。

注：慢启动并不慢，经过一个RTT的时间，cwnd的值就会变成原来的两倍，实为指数增长。

- Congestion Avoidance（拥塞避免）

当cwnd的值超过ssthresh时，就会进入Congestion Avoidance阶段，在该阶段下，cwnd以线性方式增长，大约每经过一个RTT，cwnd的值就会加1

- Fast Recovery (快速恢复)

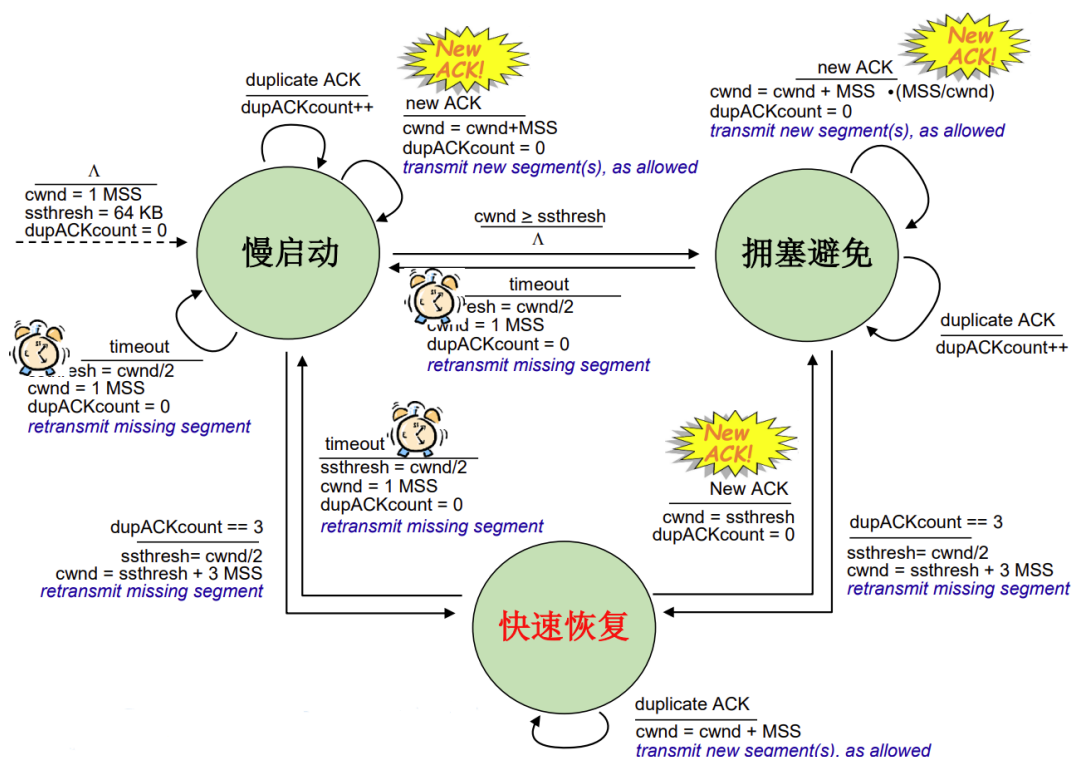
按照拥塞避免算法中cwnd的增长趋势，迟早会造成拥塞（一般通过是否丢包来判断是否发生了拥塞）。

如果网络中发生了丢包，通过等待一个RTO时间后再进行重传，是非常耗时的，因为RTO通常设置得会比较大（避免伪重传：不必要的重传）。

快速恢复是建立在Tahoe快重传上的优化。其的思想是：只要发送方收到了三个重复的ACK，就会立马重传，而不用等到RTO到达。且Reno会把当前的sssthresh的值设置为当前cwnd的一半，但是并不会回到slow start（慢启动）阶段，而是将cwnd设置为（更新后的）sssthresh+3MSS，回到Congestion Avoidance（拥塞避免）阶段，之后cwnd呈线性增长。

注：（如果没有3个重复的ACK而包丢失了，就只能超时重传）

RENO算法状态机



5.多线程函数

代码设计上：由于在收到ACK的时候，发送窗口需要向前滑动。为了分开这两个处理过程，采用多线程的方法实现传输功能。

具体代码分析可见下文。

三、核心函数&代码实现

1. 建立连接、断开连接、接收方接收数据

该部分与3-2一致，可见3-2实验设计。

2. 发送方与接收方协定单次传输数据大小&初始阈值窗口大小

```
int Client_Server_Size(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrLen,
int size, int ssthresh, int*& A) {
    Packet_Header packet;
    char* buffer = new char[sizeof(packet)]; // 发送buffer

    // 接收客户端告知的缓冲区数据大小
    while (1) {
        if (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
&servAddrLen) == -1) {
            cout << "无法接收客户端发送的连接请求" << endl;
            return -1;
        }
        memcpy(&packet, buffer, sizeof(packet));
        if (packet.tag == SYN && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
            cout << "成功收到对方发来的协商信息" << endl;
            break;
        }
    }
    int client_size = packet.datasize;
    int client_win = packet.window;

    // 服务端告知客户端数据发送缓冲区以及初始阈值大小信息
    packet.tag = ACK;
    packet.datasize = size;
    packet.window = ssthresh; //初始阈值大小
    packet.checksum = 0;
    packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
    memcpy(buffer, &packet, sizeof(packet));
    if (sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr,
servAddrLen) == -1) {
        return -1;
    }
    cout << "成功告知对方数据发送缓冲区以及初始阈值大小信息" << endl;
    // 比较协商大小，取双方之间的最小值
    A[0] = (size > client_size) ? client_size : size;
    A[1] = (ssthresh > client_win) ? client_win : ssthresh;
    return 1;
}
```

3. 发送方发送数据

主函数：

```
void Send_Message(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrLen, char*
Message, int mes_size, int max_SIZE, int init_ssthresh) {
    // 初始化全局变量 (全局变量的声明在函数体外)
    packet_num = mes_size / (max_SIZE) + (mes_size % max_SIZE != 0);
    Seq_num = 0; //初始化序列号
    Ack_num = 1; //初始化确认号
    confirmed = -1; //已确认的数据索引
    cur = 0; //当前发送数据索引
    MSS_count = 0; //拥塞阶段窗口增加计数器
    MAX_SIZE = max_SIZE; // 单次数据发送大小
    Window = 1; //当前滑动窗口大小
    ssthresh = init_ssthresh; //初始阈值大小
    dupACKcount = 0; //重复ACK数目
    // 0表示慢启动 1表示拥塞避免 2表示快速恢复
    state = 0;
    is_end = false; //数据发送完毕

    //缓冲区：利用动态数组建立数据缓冲区
    staging_size = 2 * init_ssthresh;
    begin_index = 0;
    end_index = -1;
    staging_data = new char* [staging_size];
    staging_data_len = new int[staging_size];
    staging_time = new clock_t[staging_size];
    for (int i = 0; i < staging_size; i++) {
        staging_data[i] = new char[sizeof(Packet_Header) + MAX_SIZE];
    }

    Parameters Para; //传递给线程的参数
    Para.socketClient = socketClient;
    Para.serverAddr = servAddr;
    Para.Message = Message;
    Para.mes_size = mes_size;

    // 创建线程：发送数据报 & 接受ACK滑动窗口
    HANDLE hthread[2];
    hthread[0] = CreateThread(NULL, 0, Recv_handle, (LPVOID)&Para, 0, NULL);
    hthread[1] = CreateThread(NULL, 0, Send_handle, (LPVOID)&Para, 0, NULL);
    WaitForSingleObject(hthread[0], INFINITE);
    WaitForSingleObject(hthread[1], INFINITE);
}
```

发送数据线程：

```
DWORD WINAPI Send_handle(LPVOID lparam) {
    // 参数传递
    Parameters* Para = (Parameters*)lparam;
    SOCKET socketClient = Para->socketClient;
    SOCKADDR_IN serverAddr = Para->serverAddr;
    int servAddrLen = sizeof(Para->serverAddr);
    char* Message = Para->Message;
    int mes_size = Para->mes_size;
    Packet_Header packet;

    // 发送窗口内的数据报
    char* send_buffer = new char[sizeof(packet) + MAX_SIZE];
    while (confirmed < (packet_num - 1)) { // 所有发送数据均被确认
        while (cur <= confirmed + Window && cur < packet_num) {
            end_index++; //缓冲区索引向前
            int data_len = (cur == packet_num - 1 ? mes_size - (packet_num - 1) *
MAX_SIZE : MAX_SIZE);
            packet.tag = 0;
            packet.seq = Seq_num++;
            Seq_num = (Seq_num > 255 ? Seq_num - 256 : Seq_num);
            packet.datasize = data_len;
            packet.window = Window; //当前数据发送滑动窗口大小
            packet.checksum = 0;
            packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
            memcpy(send_buffer, &packet, sizeof(packet));
            char* mes = Message + cur * MAX_SIZE;
            memcpy(send_buffer + sizeof(packet), mes, data_len);

            // 记录每一次的发送时间
            clock_t now = clock();
            staging_time[end_index % staging_size] = now;

            // 发送数据
            sendto(socketClient, send_buffer, sizeof(packet) + data_len, 0,
(sockaddr*)&serverAddr, servAddrLen);

            //将数据保留在缓冲区
            memcpy(staging_data[end_index % staging_size], send_buffer, sizeof(packet) +
data_len);
            staging_data_len[end_index % staging_size] = data_len;

            cout << "Begin sending message..... datasize:" << data_len << " bytes!"
                << " seq:" << int(packet.seq) << " window:" << int(packet.window)
                << "当前阈值大小: " << ssthresh << "当前state: " << state << endl;
            cur++; //发送数据索引增加
        }

        // 快速重传
        if (dupACKcount == 3) { //转至快速恢复状态
```

```

        for (int temp = begin_index; temp <= end_index; temp++) {
            sendto(socketClient, staging_data[temp % staging_size], sizeof(packet) +
staging_data_len[temp % staging_size], 0, (sockaddr*)&serverAddr, servAddrLen);
            // 输出缓冲区数据检查
            Packet_Header p_temp;
            memcpy(&p_temp, staging_data[temp % staging_size], sizeof(packet));
            cout << "dupACK! 重新发送暂存区内所有数据: 序列号为" << int(p_temp.seq) <<
endl;

            clock_t now = clock();
            staging_time[temp % staging_size] = now;
        }
        // 阈值、窗口均需要改变
        ssthresh = Window / 2;
        Window = ssthresh + 3;
        MSS_count = 0;
        state = 2;
    }
    else { // 超时机制
        if ((clock() - staging_time[begin_index % staging_size]) / 1000 > 2 &&
(end_index >= begin_index)) { //时间超过3秒 标记为超时重传
            //重传所有未被ack的数据
            for (int temp = begin_index; temp <= end_index; temp++) {
                sendto(socketClient, staging_data[temp % staging_size],
sizeof(packet) + staging_data_len[temp % staging_size], 0, (sockaddr*)&serverAddr,
servAddrLen);

                Packet_Header p_temp;
                memcpy(&p_temp, staging_data[temp % staging_size], sizeof(packet));
                cout << "超时! 重传暂存区内所有数据: 序列号为" << int(p_temp.seq) <<
endl;

                clock_t now = clock();
                staging_time[temp % staging_size] = now;
            }
            //转至慢启动状态
            ssthresh = Window / 2;
            Window = 1;
            dupACKcount = 0;
            MSS_count = 0;
            cout << "超时进入慢启动阶段" << endl;
            state = 0;
        }
    }
}
//发送结束标志
packet.tag = OVER;
char* buffer = new char[sizeof(packet)];
packet.checksum = 0;
packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
memcpy(buffer, &packet, sizeof(packet));
sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&serverAddr,
servAddrLen);
clock_t start = clock();
cout << "The end tag has been sent" << endl;

```



```

// 注意：需要引入控制锁 唯有接收线程确认到ACK 才能结束程序
while (!is_end) {
    if ((clock() - start) / 1000 > 2)    //时间超过2秒 标记为超时重传
    {
        sendto(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&serverAddr,
servAddrLen);
        cout << "最终结束标志数据发送超时，正在重传！！!" << endl;
        start = clock();
    }
}
return 0;
}

```

接收ACK线程：

```

DWORD WINAPI Recv_handle(LPVOID lparam) {
    // 初始化参数
    Parameters* Para = (Parameters*)lparam;
    SOCKET socketClient = Para->socketClient;
    SOCKADDR_IN serverAddr = Para->serverAddr;
    int servAddrLen = sizeof(Para->serverAddr);
    char* Message = Para->Message;
    int mes_size = Para->mes_size;
    Packet_Header packet;

    // 接收数据
    char* re_buffer = new char[sizeof(packet)];
    char* send_buffer = new char[sizeof(packet) + MAX_SIZE];
    while (confirmed < (packet_num - 1)) {
        if (recvfrom(socketClient, re_buffer, sizeof(packet), 0, (sockaddr*)&serverAddr,
&servAddrLen) > 0) {
            //成功收到响应，数据校验出错说明不是正确的数据，直接丢弃
            memcpy(&packet, re_buffer, sizeof(packet));
            if ((compute_sum((WORD*)&packet, sizeof(packet)) != 0)) { //数据校验出错
                continue;
            }

            // 不同状态下对接收端回传的不同处理
            switch (state)
            {
            case 0:    //慢启动状态下
                if (packet.ack == Ack_num) {
                    Ack_num = (Ack_num + 1) % 256;
                    Window++;
                    dupACKcount = 0;
                    confirmed++;
                    begin_index++;
                }
                else {
                    int dis = (int(packet.ack) - Ack_num) > 0 ? (int(packet.ack) -
Ack_num) : (int(packet.ack) + 256 - Ack_num);

```

```

        //重复ACK
        if (packet.ack == (Ack_num + 256 - 1) % 256) {
            dupACKcount++;
        }
        else if (dis < Window) {
            // 累计确认
            for (int temp = 0; temp <= dis; temp++) {
                Window++;
                confirmed++;
                begin_index++;
            }
            dupACKcount = 0;
            Ack_num = (int(packet.ack) + 1) % 256; //更新下次期待的ACK序号
        }
        else { //数据校验出错、ack不合理 重发数据
            cout << "Wrong Message !!!" << endl;
        }
    }
    if (Window >= ssthresh && dupACKcount != 3) { //转至拥塞控制状态
        cout << "慢启动阶段转至拥塞控制阶段" << endl;
        state = 1;
    }
    break;
case 1: //拥塞控制状态
    if (packet.ack == Ack_num) {
        Ack_num = (Ack_num + 1) % 256;
        MSS_count++;
        if (MSS_count == Window) {
            Window++;
            MSS_count = 0;
        }
        dupACKcount = 0;
        begin_index++;
        confirmed++;
    }
    else {
        int dis = (int(packet.ack) - Ack_num) > 0 ? (int(packet.ack) - Ack_num) : (int(packet.ack) + 256 - Ack_num);
        //重复ACK
        if (packet.ack == (Ack_num + 256 - 1) % 256) {
            dupACKcount++;
        }
        else if (dis < Window) {
            // 累计确认
            for (int temp = 0; temp <= dis; temp++) {
                MSS_count++;
                if (MSS_count == Window) {
                    Window++;
                    MSS_count = 0;
                }
            }
            begin_index++;
            confirmed++;
        }
    }
}

```

```

    }
    dupACKcount = 0;
    Ack_num = (int(packet.ack) + 1) % 256; //更新下次期待的ACK序号
}
else { //数据校验出错、ack不合理 重发数据
    cout << "Wrong Message !!!" << endl;
}
}
break;
case 2: //快速恢复状态
    if (packet.ack == Ack_num) {
        Ack_num = (Ack_num + 1) % 256;
        Window = ssthresh;
        dupACKcount = 0;
        cout << "快速回复阶段转至拥塞控制阶段" << endl;
        state = 1;
        begin_index++;
        confirmed++;
    }
    else {
        int dis = (int(packet.ack) - Ack_num) > 0 ? (int(packet.ack) -
Ack_num) : (int(packet.ack) + 256 - Ack_num);
        //重复ACK
        if (packet.ack == (Ack_num + 256 - 1) % 256) {
            dupACKcount++;
            Window = Window + 1;
        }
        else if (dis < Window) {
            // 累计确认
            for (int temp = 0; temp <= dis; temp++) {
                begin_index++;
                confirmed++;
            }
            cout << "快速回复阶段转至拥塞控制阶段" << endl;
            state = 1;
            Window = ssthresh;
            dupACKcount = 0;
            Ack_num = (int(packet.ack) + 1) % 256; //更新下次期待的ACK序号
        }
        else { //数据校验出错、ack不合理 重发数据
            cout << "Wrong Message !!!" << endl;
        }
    }
}
break;
default:
    break;
}
}
}

char* buffer = new char[sizeof(packet)];
while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&serverAddr,
&servAddrLen) <= 0) {}

```

```
memcpy(&packet, buffer, sizeof(packet));
if (packet.tag == OVER && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
    cout << "The end token was successfully received" << endl;
}
else {
    cout << "无法接收客户端回传结束标志" << endl;
}
is_end = true; //更新锁，这样发送线程可以退出程序
return 0;
}
```

四、实际效果展示

- 本实验是建立在3-1、3-2的基础上，部分效果已在之前报告中有所展示，不在本报告中重复描述。
- 打开服务端和客户端。在客户端应用界面输入本次客户端的功能定位：**发送方OR接收方**。双方在三次握手建立好连接后，首先需要协定好**单次数据发送大小**和**初始阈值的大小**。

```
F:\C++_Project\NET_UDP\NET_UDP_Server\Debug\NET_UDP_Server.exe
服务端套接字创建成功！
等待客户端提出连接请求.....
成功接收第一次握手信息
成功发送第二次握手信息
成功收到第三次握手信息
客户端与服务端成功进行三次握手建立连接！-----可以开始发送/接收数据
请输入本端单次容纳数据缓冲区大小: 1024
请输入本端窗口缓冲区阈值大小: 60
成功收到对方数据发送缓冲区以及初始阈值大小信息
成功告知对方数据发送缓冲区以及初始阈值大小信息
双方协定的单次数据发送大小为: 1024, 窗口阈值大小为: 50
=====
Waiting to receive!!!
```

协商结果

```
F:\C++_Project\NET_UDP\NET_UDP_Client\Debug\NET_UDP_Client.exe
客户端套接字创建成功！
请输入选项告知——（发送0 / 接收1）数据: 0
开始向服务端建立连接请求.....
成功发送第一次握手信息
成功收到第二次握手信息
成功发送第三次握手信息
客户端与服务端成功进行三次握手建立连接！-----可以开始发送/接收数据
请输入本端单次容纳数据缓冲区大小: 2048
请输入本端窗口缓冲区阈值大小: 50
成功告知对方数据发送缓冲区以及初始阈值大小信息
成功收到对方缓冲区大小信息
双方协定的单次数据发送大小为: 1024, 窗口阈值大小为: 50
=====
Waiting to choose File!!!
Enter File name:
```

双方成功协定之后，由发送端选择需要传输的文件进行发送，观察发送方与接收方的日志输出结果。

- 接收方日志：逐条数据包进行Ack确认

 F:\C++_Project\NET_UDP\NET_UDP_Server\Debug\NET_UDP_Server.exe

```
=====
=====
Waiting to receive!!!
预期的seq: 0 实际收到的: 0
Begin recving message..... datasize:5 bytes! flag:0 seq:0 window:1 checksum:65274
Successful recv and send response ack : 1
The end tag has been recved
The end tag has been sent
预期的seq: 0 实际收到的: 0
Begin recving message..... datasize:1024 bytes! flag:0 seq:0 window:1 checksum:64255
Successful recv and send response ack : 1
预期的seq: 1 实际收到的: 1
Begin recving message..... datasize:1024 bytes! flag:0 seq:1 window:2 checksum:63998
Successful recv and send response ack : 2
预期的seq: 2 实际收到的: 2
Begin recving message..... datasize:1024 bytes! flag:0 seq:2 window:2 checksum:63997
Successful recv and send response ack : 3
预期的seq: 3 实际收到的: 3
Begin recving message..... datasize:1024 bytes! flag:0 seq:3 window:3 checksum:63740
Successful recv and send response ack : 4
预期的seq: 4 实际收到的: 4
Begin recving message..... datasize:1024 bytes! flag:0 seq:4 window:4 checksum:63483
Successful recv and send response ack : 5
预期的seq: 5 实际收到的: 5
```

- 发送方日志：按照窗口的剩余大小进行数据包的发送

- 慢启动阶段（注意观察程序日志输出中的 **window** 变化，窗口的大小变化是随着每个ACK的到来而加一（不超过阈值sssthresh）。这里窗口window的变化也侧面的验证出程序正确的实现了多线程功能（多线程的特性：发送与接收是分开独立的）。）

```
=====
Waiting to choose File!!!
Enter File name:2.jpg
发送文件数据大小: 5898505bytes!
Begin sending message..... datasize:5 bytes! seq:0 window:1当前阈值大小: 50当前state: 0
The end tag has been sent
The end token was successfully received
Begin sending message..... datasize:1024 bytes! seq:1 window:1当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:2 window:2当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:3 window:2当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:4 window:3当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:5 window:3当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:6 window:4当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:7 window:5当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:8 window:5当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:9 window:5当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:10 window:6当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:11 window:7当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:12 window:7当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:13 window:8当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:14 window:8当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:15 window:9当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:16 window:9当前阈值大小: 50当前state: 0
```

窗口比较

慢启动

- 慢启动转至拥塞避免阶段（拥塞避免阶段，窗口大小变化成线性的增长趋势序列号 **从98增长至148时window才变化**）

```
Begin sending message..... datasize:1024 bytes! seq:96 window:49当前阈值大小: 50当前state: 0
慢启动阶段转至拥塞控制阶段
Begin sending message..... datasize:1024 bytes! seq:97 window:50当前阈值大小: 50当前state: 0
Begin sending message..... datasize:1024 bytes! seq:98 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:99 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:100 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:101 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:102 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:103 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:104 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:105 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:106 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:107 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:108 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:109 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:110 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:111 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:112 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:113 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:114 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:115 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:146 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:147 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:148 window:51当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:149 window:51当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:150 window:51当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:151 window:51当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:152 window:51当前阈值大小: 50当前state: 1
```

拥塞阶段：线性增长

快速恢复阶段


```
Begin sending message..... datasize:1024 bytes! seq:142 window:50当前阈值大小: 50当前state: 1
Begin sending message..... datasize:1024 bytes! seq:143 window:50当前阈值大小: 50当前state: 1
dupACK! 重新发送暂存区内所有数据: 序列号为94
dupACK! 重新发送暂存区内所有数据: 序列号为95
dupACK! 重新发送暂存区内所有数据: 序列号为96
dupACK! 重新发送暂存区内所有数据: 序列号为97
dupACK! 重新发送暂存区内所有数据: 序列号为98
dupACK! 重新发送暂存区内所有数据: 序列号为99
dupACK! 重新发送暂存区内所有数据: 序列号为100
dupACK! 重新发送暂存区内所有数据: 序列号为101
dupACK! 重新发送暂存区内所有数据: 序列号为102
dupACK! 重新发送暂存区内所有数据: 序列号为103
dupACK! 重新发送暂存区内所有数据: 序列号为104
dupACK! 重新发送暂存区内所有数据: 序列号为105
dupACK! 重新发送暂存区内所有数据: 序列号为106
dupACK! 重新发送暂存区内所有数据: 序列号为107
dupACK! 重新发送暂存区内所有数据: 序列号为108
dupACK! 重新发送暂存区内所有数据: 序列号为109
dupACK! 重新发送暂存区内所有数据: 序列号为110
dupACK! 重新发送暂存区内所有数据: 序列号为111
dupACK! 重新发送暂存区内所有数据: 序列号为112
dupACK! 重新发送暂存区内所有数据: 序列号为113
dupACK! 重新发送暂存区内所有数据: 序列号为114
dupACK! 重新发送暂存区内所有数据: 序列号为115
dupACK! 重新发送暂存区内所有数据: 序列号为116
dupACK! 重新发送暂存区内所有数据: 序列号为117
dupACK! 重新发送暂存区内所有数据: 序列号为118
dupACK! 重新发送暂存区内所有数据: 序列号为119
dupACK! 重新发送暂存区内所有数据: 序列号为120
dupACK! 重新发送暂存区内所有数据: 序列号为121
dupACK! 重新发送暂存区内所有数据: 序列号为122
dupACK! 重新发送暂存区内所有数据: 序列号为123
dupACK! 重新发送暂存区内所有数据: 序列号为124
dupACK! 重新发送暂存区内所有数据: 序列号为125
dupACK! 重新发送暂存区内所有数据: 序列号为126
dupACK! 重新发送暂存区内所有数据: 序列号为127
dupACK! 重新发送暂存区内所有数据: 序列号为128
dupACK! 重新发送暂存区内所有数据: 序列号为129
dupACK! 重新发送暂存区内所有数据: 序列号为130
dupACK! 重新发送暂存区内所有数据: 序列号为131
dupACK! 重新发送暂存区内所有数据: 序列号为132
dupACK! 重新发送暂存区内所有数据: 序列号为133
dupACK! 重新发送暂存区内所有数据: 序列号为134
dupACK! 重新发送暂存区内所有数据: 序列号为135
dupACK! 重新发送暂存区内所有数据: 序列号为136
dupACK! 重新发送暂存区内所有数据: 序列号为137
dupACK! 重新发送暂存区内所有数据: 序列号为138
dupACK! 重新发送暂存区内所有数据: 序列号为139
dupACK! 重新发送暂存区内所有数据: 序列号为140
dupACK! 重新发送暂存区内所有数据: 序列号为141
dupACK! 重新发送暂存区内所有数据: 序列号为142
dupACK! 重新发送暂存区内所有数据: 序列号为143
Begin sending message..... datasize:1024 bytes! seq:144 window:51当前阈值大小: 25当前state: 2
Begin sending message..... datasize:1024 bytes! seq:145 window:52当前阈值大小: 25当前state: 2
Begin sending message..... datasize:1024 bytes! seq:146 window:55当前阈值大小: 25当前state: 2
Begin sending message..... datasize:1024 bytes! seq:147 window:57当前阈值大小: 25当前state: 2
Begin sending message..... datasize:1024 bytes! seq:148 window:50当前阈值大小: 25当前state: 2
Begin sending message..... datasize:1024 bytes! seq:149 window:快速恢复阶段转至拥塞控制阶段62当前阈值大小: 窗口大小 cwnd63 = ssthresh25当前state: 252
```

快速恢复阶段

该阶段成功接收正确ACK，从快速恢复变回
拥塞避免阶段，Window改为变更后的ssresh

多线程造成的抢占cpu输出不完整

- 利用路由器程序展开丢包测试

 Router ✕

路由器IP:

服务器IP:

端口:

服务器端口:

丢包率: %

延时: ms

日志

count:98.
count:99.
Miss a packet.
count:1.
count:2.
count:3.
count:4.
count:5.
count:6.
count:7.
count:8.

- 在开启路由器程序模拟丢包时，发现发送方在程序在后期阶段，窗口大小会一直在一个大小范围内进行跳动：拥塞避免阶段窗口增长至15左右时，便会发生丢包转至快速重传阶段。该现象出现的原因应该是由由于路由器程序在模拟丢包时（设定为1%），每次都是丢失第100个包，与发送方发送数据时窗口增长时间上出现了重合！

```
Begin sending message..... datasize:1024 bytes! seq:254 window:15当前阈值大小: 7当前state: 1
Begin sending message..... datasize:1024 bytes! seq:255 window:15当前阈值大小: 7当前state: 1
Begin sending message..... datasize:1024 bytes! seq:0 window:15当前阈值大小: 7当前state: 1
dupACK! 重新发送暂存区内所有数据: 序列号为242
dupACK! 重新发送暂存区内所有数据: 序列号为243
dupACK! 重新发送暂存区内所有数据: 序列号为244
dupACK! 重新发送暂存区内所有数据: 序列号为245
dupACK! 重新发送暂存区内所有数据: 序列号为246
dupACK! 重新发送暂存区内所有数据: 序列号为247
dupACK! 重新发送暂存区内所有数据: 序列号为248
dupACK! 重新发送暂存区内所有数据: 序列号为249
dupACK! 重新发送暂存区内所有数据: 序列号为250
dupACK! 重新发送暂存区内所有数据: 序列号为251
dupACK! 重新发送暂存区内所有数据: 序列号为252
dupACK! 重新发送暂存区内所有数据: 序列号为253
dupACK! 重新发送暂存区内所有数据: 序列号为254
dupACK! 重新发送暂存区内所有数据: 序列号为255
dupACK! 重新发送暂存区内所有数据: 序列号为0
Begin sending message..... datasize:1024 bytes! seq:1 window:16当前阈值大小: 7当前state: 2
Begin sending message..... datasize:1024 bytes! seq:2 window:17当前阈值大小: 7当前state: 2
Begin sending message..... datasize:1024 bytes! seq:3 window:18当前阈值大小: 7当前state: 2
快速回复阶段转至拥塞控制阶段窗口大小 cwnd18 = ssthresh7
```

```

Begin sending message..... datasize:1024 bytes! seq:83 window:15当前阈值大小: 7当前state: 1
Begin sending message..... datasize:1024 bytes! seq:84 window:15当前阈值大小: 7当前state: 1
Begin sending message..... datasize:1024 bytes! seq:85 window:15当前阈值大小: 7当前state: 1
dupACK! 重新发送暂存区内所有数据: 序列号为71
dupACK! 重新发送暂存区内所有数据: 序列号为72
dupACK! 重新发送暂存区内所有数据: 序列号为73
dupACK! 重新发送暂存区内所有数据: 序列号为74
dupACK! 重新发送暂存区内所有数据: 序列号为75
dupACK! 重新发送暂存区内所有数据: 序列号为76
dupACK! 重新发送暂存区内所有数据: 序列号为77
dupACK! 重新发送暂存区内所有数据: 序列号为78
dupACK! 重新发送暂存区内所有数据: 序列号为79
dupACK! 重新发送暂存区内所有数据: 序列号为80
dupACK! 重新发送暂存区内所有数据: 序列号为81
dupACK! 重新发送暂存区内所有数据: 序列号为82
dupACK! 重新发送暂存区内所有数据: 序列号为83
dupACK! 重新发送暂存区内所有数据: 序列号为84
dupACK! 重新发送暂存区内所有数据: 序列号为85
Begin sending message..... datasize:1024 bytes! seq:86 window:16当前阈值大小: 7当前state: 2
快速回复阶段转至拥塞控制阶段Begin sending message..... datasize:1024 bytes! seq:窗口大小 cwnd87 window:16当前state: 2
Begin sending message..... datasize:1024 bytes! seq:88 window:16当前阈值大小: 7当前state: 1

```

- 程序可以一直运行，发送文件数据。在用户主动按下q键的情况下，数据文件传输结束，进行断联过程，四次挥手双方断开连接

```

=====
Waiting to choose File!!!
Enter File name:q
Send a all_end flag to the server
成功发送第一次挥手信息
成功收到第二次挥手信息
成功收到第三次挥手信息
成功发送第四次挥手信息
客户端与服务端成功断开连接！
请按任意键继续. . .

```

```

=====
Waiting to receive!!!
The all_end tag has been recved
成功收到第一次挥手信息
成功发送第二次挥手信息
成功发送第三次挥手信息
成功收到第四次挥手信息
客户端与服务端成功断开连接！
请按任意键继续. . .

```

- 发送方使用fstream二进制方式打开文件，进行传输。接收方在收到所有内容之后，使用fstream二进制方式将其保存写入。

```

=====
The end tag has been recved
The end tag has been sent
接收文件名: 1.jpg
接收文件数据大小: 1857353bytes!
Successfully receive the data in its entirety and save it
=====

```

五、实验总结

基于UDP服务实现可靠传输实验第一、二部分的基础上，实现了Reno拥塞控制算法。

除却实现规定实验功能，作者还额外实现如下功能：

1. 双向传输：客户端与服务端双方都可以作为发送方，另一端作为接收方。这样的功能定义通过客户端在三次握手的第三次握手数据包发送时将label标签在双方之间进行传递，以此双方进行确定本次应用双方的角色。
2. 单次发送数据大小&初始阈值窗口大小的协商确定：客户端与服务端在文件数据send&recv之前，先进行一次协商。客户端与服务端各自在数据帧中告知对方相关数据，双方按照取min的方法进行协定。
3. 程序不会在完成一次文件数据传输后就退出程序，而是在用户主动按q键后，才会end程序。
4. 在实验的过程中，主动添加超时重传的测试环节（模拟发送方丢包，接收方无法按序接收造成超时未收到确认的情况），测试发送方与接收方遇到超时情况下是否能够正确处理。
5. 在实验的过程中，主动添加累计确认的测试环节（模拟接收方回传ACK的数据包中发生丢包，发送方收到的ACK表明接收方已正确接收到序号为n的以前且包括n在内的所有分组。）测试发送方与接收方在接收方回传ACK发生丢包时是否能够正确处理。

实验过程中关于多线程的代码实现有一些个人感悟：

由于本人最初是利用C++中的队列来实现数据缓冲区，队列的一大好处是：由于我们发送端在传输数据时，数据时按序发送的，不论是快速恢复还是超时重传，数据也都是按序重传。队列先进先出的结构特性就很好的帮助我们管理缓冲区。而且队列实现数据缓冲区的另一大好处是：无需用户维护大小，每次发送数据时只需要push进队列的尾部就可以了，而不用像动态数组那样去特意在意索引的指向维护！

不幸的是：在多线程设计的时候，由于发送与接收是分开的。接收方在收到正确的ACK响应后，会pop掉当前队列的头部。发送方在重传数据时需要pop掉头部，再把头部push进尾部，这样才可以完成整个缓冲区的重传。问题就出在这：在发送方重传的过程中，接收方可能收到了正确的ACK响应，接收方便会pop掉头部，但是当前的头部可能并不是丢失掉的数据报，因为发送方在重传过程中需要将所有数据都pop一遍。

最终的建议是：队列可以使用，不过在多线程的设计中，可能需要线程的共享数据：缓冲区上锁或者添加验证的手段！

在后续进一步的实验设计中，希望能够实现更加安全、可靠、快速的文件数据传输。