3-1的内容传输、校验和、握手挥手等就不讲了，3-2中主要涉及的新内容有：

- 在send端将发送packet功能和接受ack功能区分开来，也就是说要创建一个接收线程执行接收函数：`recvRespondThread()`
- 在send端实现滑动窗口功能，并采取累计确认模式
- 实现超时重传，意味着要创建一个计时器线程，专门用来记录需不需要将nextseqnum（在本程序中写作seq，下不再注释）置回base，重传发送窗口中的内容

一些全局变量和标志位

如果您真的打算认真读这份报告的话，接下来这些全局变量对于阅读代码是比较重要的：

```
#define PACKETSIZE 1500
#define HEADERSIZE 14
#define DISCARD_RATE 0.02 // 丢包率
#define DELAY_RATE 0.05
#define TIMEOUT 50 // 超时时间（单位：ms）
#define TEST_STOPTIME 50 // send window 满后发送区等待的时间（单位：ms）
#define DELAY_TIME 60 // 延时时间（单位：ms）

clock_t start, last; // 用于检测是否超时的计时器变量
int hasSent = 0; // 已发送的文件大小
int fileSize = 0;
int sendResult = 0; // 每次sendto函数的返回结果
int sendSize = 0; // 每次实际发送的报文总长度
int seq = 1, ack = 0; // 发送包时的seq, ack
int base = 1; // 滑动窗口起始
int seq_opp = 0, ack_opp = 0; // 收到的receive端响应报文中的seq, ack
int dataLength = 0; // 每次实际发送的数据部分长度(= sendSize - HEADERSIZE)
u_short checksum = 0; // 校验和
bool resend = false; // 重传标志，为true时在sendfile()中会进入重传的部分
char recvBuf[HEADERSIZE] = {0}; // 接受响应报文的缓冲区
int recvResult = 0; // 接受响应报文的返回值
bool finishSend = false; // 是否结束了一个文件发送的标志

bool THREAD_END = false; // 通过这个变量告诉recvRespondThread和timerThread退出
int THREAD_CREAT_FLAG = 1; // 通过这个变量创建recvRespondThread和timerThread，这两个线程当然只被创建一次
int index = 0; // 用于拯救receive发过来的最后一个确认包丢失，send端卡在hasSent == fileSize内的出不来的情况的变量（在1 - DISCARD_RATE(0.02) - DELAY_RATE(0.05)的概率上是不会发生的）
```

接收ack的线程 & 窗口滑动 & 累计确认

在3-2流水线协议中，为了实现同时的收发，需要把3-1停等协议中的接收消息的内容专门放到一个线程中。需要注意的是，根据GBN协议，如果收到了正确的ack（ack_opp >= base），那就移动base到ack_opp + 1，并且重置计时器；如果收到的是错误的ack（ack_opp < base，实际上就是base - 1），那就从什么也不做，等待超时后从base重传窗口中的内容。

窗口滑动就是base = ack_opp + 1（因为累计确认机制所以这么移动），由于`SEND_WINDOW_SIZE`是个定值（30），所以下一次能发送的seq范围就会变化（`base <= seq <= base + SEND_WINDOW_SIZE`）。

```
void recvRespondThread() {
    // 接受响应报文的线程
    while (!THREAD_END) {
        recvResult = recvfrom(sendSocket, recvBuf, HEADERSIZE, 0,
(SOCKADDR*)&recvAddr, &len);
        if (recvResult == SOCKET_ERROR) {
            cout << "receive error! sleep!" << endl;
            std::this_thread::sleep_for(std::chrono::milliseconds(2000));
            continue;
        }

        if (recvBuf[FLAG_BIT_POSITION] == 0b001) {
            // 收到了挥手前让该线程退出的报文
            break;
        }

        seq_opp = (u_char)recvBuf[SEQ_BITS_START] +
((u_char)recvBuf[SEQ_BITS_START + 1] << 8)
                + ((u_char)recvBuf[SEQ_BITS_START + 2] << 16) +
((u_char)recvBuf[SEQ_BITS_START + 3] << 24);
        ack_opp = (u_char)recvBuf[ACK_BITS_START] +
((u_char)recvBuf[ACK_BITS_START + 1] << 8)
                + ((u_char)recvBuf[ACK_BITS_START + 2] << 16) +
((u_char)recvBuf[ACK_BITS_START + 3] << 24);
        // cout << "Received response, seq_opp = " << seq_opp << endl;
        if (recvBuf[FLAG_BIT_POSITION] == 0b100 && ack_opp >= base) {
            // 对方正确收到了这个packet
            base = ack_opp + 1;
            resend = false;
            index = 0;
            cout << "Having received the correct ack = " << ack_opp << ", now
base = " << base << endl;

            // 启动计时（实际上是重置计时器）
            start = clock();
        } else {
            // 如果接受到的ack < base，实际上什么也不干。不移动base，等待超时
            cout << "Received the wrong ack = " << ack_opp << ". From base = "
<< base << ", packets in SW need to be resent." << endl;
        }

        if (base == (fileSize / (PACKETSIZE - HEADERSIZE) + 2)) {
            // 发送完毕时的base
            finishSend = true;
        }

    }
}

// 在sendfile()函数中第一次遇到相应代码会该启动线程，发送完毕后，在main函数中等用户输入"q"
(quit)时会修改THREAD_END变量(它是while的条件)，使线程自己结束
```

计时器线程

根据GBN协议，`start`（计时器变量）会在`recvRespondThread()`中的`ack_opp >= base`，和`sendfile()`中的`base == seq`发生时更新。也就是说，如果长时间没有收到正确的ack，则会超时；如果发送了滑动窗口的第一个包，则会开始计时。

```
void timerThread() {
    while(!THREAD_END) {
        last = clock();
        if (last - start >= TIMEOUT) {
            start = clock();
            resend = true;
        }
    }
}
```

重传功能

通过全局`bool`变量`resend`的设置来决定在`sendfile()`中要不要步入超时的语句块：

```
sendfile() {
    while(true) {
        // ...

        if (resend) {
            // 如果需要重传（唯一需要重传的情况就是超时，收到错误的ack并不会重传），则将seq回到base
            // 减去窗口内已经传输的数据长度(dataLength)，并且考虑在最后一组滑动窗口内的包出错的可能性
            if (dataLength == PACKETSIZE - HEADERSIZE) {
                hasSent -= dataLength * (seq - base);
            } else {
                // 如果dataLength较小，说明是发了最后一个包
                hasSent -= dataLength;
                hasSent -= (PACKETSIZE - HEADERSIZE) * (seq - base - 1);
            }

            seq = base; // Go-Back N
            resend = false;
            continue;
        }

        if (seq < base + SEND_WINDOW_SIZE) {
            // send a packet...
        }
        // ...
    }
}
```

超时后要做的事情也很简单，让seq回到base，然后删减一下已发送的长度就好了。seq回到base后，再利用`while(true)`循环从base那儿的数据开始发就行了。

进行丢包测试

在send端，每次发送前生成一个0-1之间的随机数，大于`DISCARD_RATE`才会真正发送，以此模拟丢包；

在receive端，除了每次发送ack之前生成0-1随机数决定是否要真正发外，还会生成一个0-1的随机数，和`DELAY_RATE`比一下，决定在发之前是否要睡一会儿，以此模拟send端收到ack时的延时。不过这两个干扰本质上是一样的，对send端来说都会因为没有收到ack而超时，从而准备重传。

send端：

```
// 用生成随机数模仿丢包率
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0, 1);

// 模拟丢包
if (dis(gen) > DISCARD_RATE)
    sendResult = sendto(sendSocket, sendBuf, sendSize, 0, (SOCKADDR*)&recvAddr,
sizeof(SOCKADDR));
```

receive端：

```
// 模拟延时
if (dis(gen) < DELAY_RATE)
    std::this_thread::sleep_for(std::chrono::milliseconds(DELAY_TIME));

// 模拟丢包
if (dis(gen) > DISCARD_RATE)
    sendto(recvSocket, header, HEADERSIZE, 0, (SOCKADDR*)&sendAddr,
sizeof(SOCKADDR));
```