南開大學
Nankai University

计算机学院

计算机网络实验报告

UDP 可靠数据传输

姓名：赵一名

学号：2013922

专业：计算机科学与技术

2022 年 11 月 26 日

目录

1 概述

本次实验中我在实验 3.1 的基础上进一步 UDP 实现了可靠文件传输，本程序支持固定窗口大小的流量控制机制，并支持累计确认。

程序运行的最终效果图如下图所示：

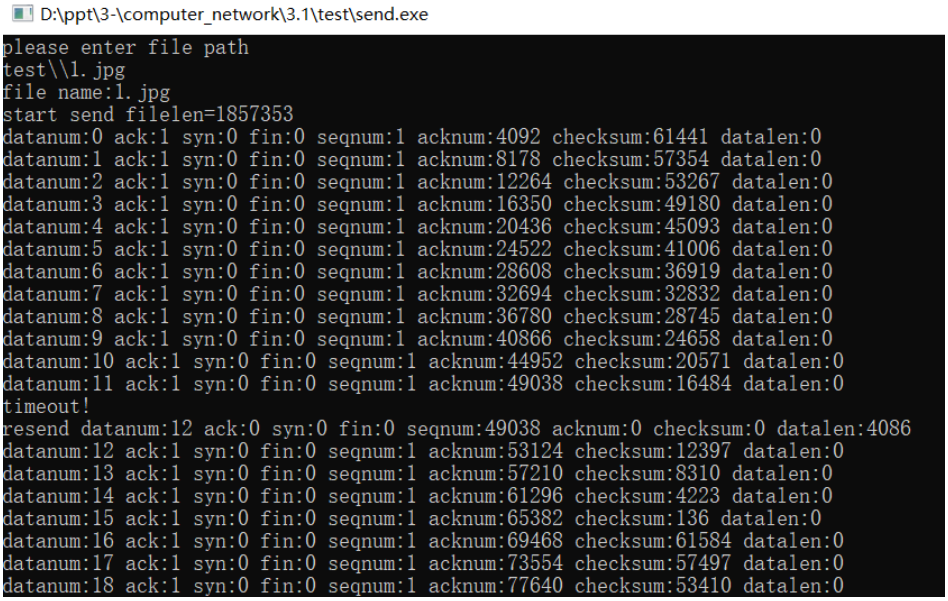

图 1.1: 发送端

未发生丢包时，程序可正常发送，当丢包事件触发计时器时，重发丢失的数据包。

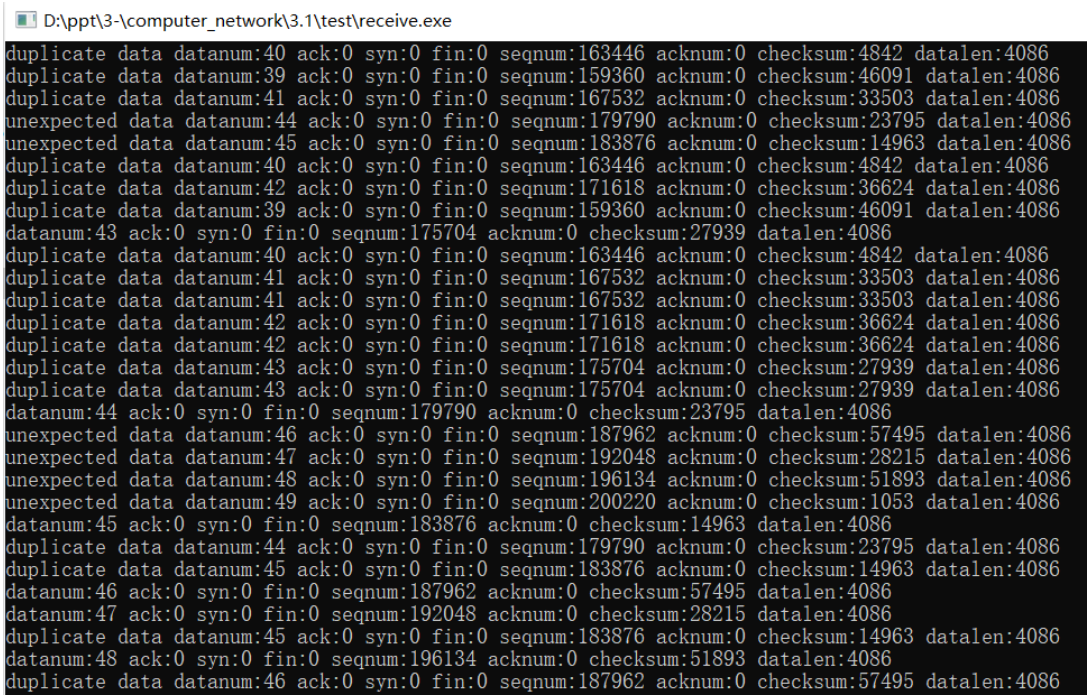

图 1.2: 接收端

本实验中我采用的是 GBN 协议，所以仅会接受当前预期的包。

一次文件发送完毕后会输出所传输的文件的大小，传输所需的时间与吞吐量等信息，并询问发送端是否继续发送下一个文件，如下图所示：

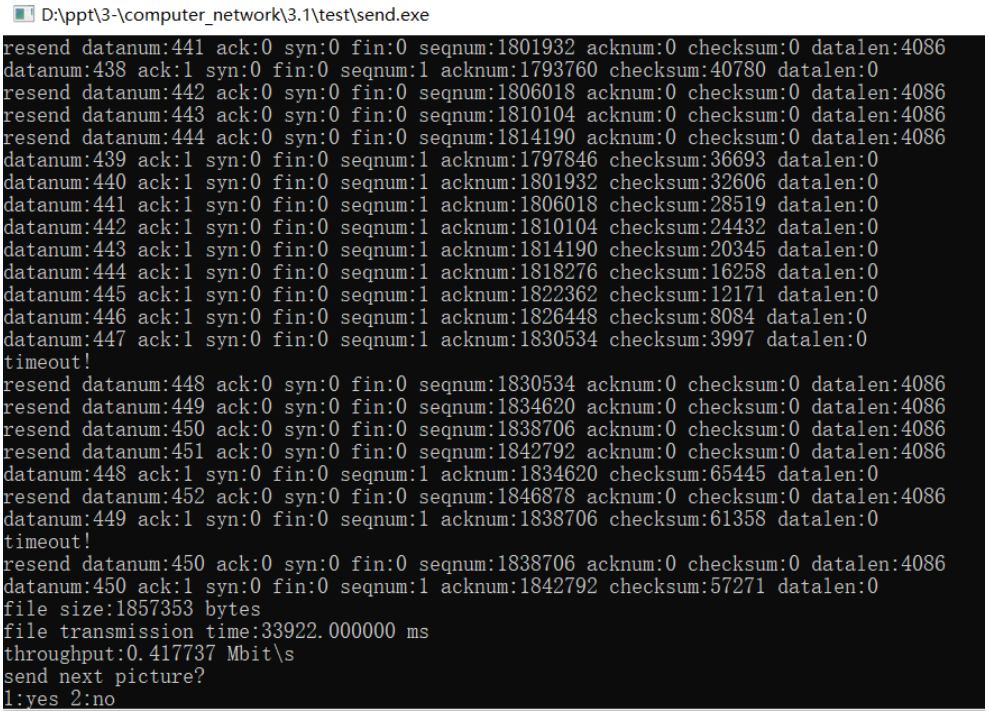

图 1.3: 发送端

1.1 程序使用说明

使用路由器小程序的各个参数如下图所示：

图 1.4: 路由器小程序

打开 *receive.exe* 与 *send.exe* 两个可执行程序后，在发送端中输入待发送的文件的路径，文件将会自动传输到接收端同级目录下的 *receive* 文件夹中。

2 原理介绍

本程序中我采用的 GBN(回退 N 步) 协议来实现滑动窗口的流量控制机制。

2.1 滑动窗口机制

滑动窗口的基本原理就是在任意时刻，发送方都维持了一个连续的允许发送的帧的序号，称为发送窗口；同时，接收方也维持了一个连续的允许接收的帧的序号，称为接收窗口。发送窗口和接收窗口的序号的上下界不一定要一样，甚至大小也可以不同。不同的滑动窗口协议窗口大小一般不同。发送方窗口内的序列号代表了那些已经被发送，但是还没有被确认的帧，或者是那些可以被发送的帧。

若从滑动窗口的观点来统一看待 1 比特滑动窗口、GBN((回退 N 步) 及 SR(选择重传) 三种协议，它们的差别仅在于各自窗口尺寸的大小不同而已。

1 比特滑动窗口协议就是在实验 3.1 中实现的协议，其发送窗口 =1，接收窗口 =1。

GBN 协议，也即本次实验中我实现的协议，其发送窗口 >1，接收窗口 =1。

SR 协议, 其发送窗口 >1, 接收窗口 >1。

2.2 GBN 协议

GBN 协议指的是当接收方发现某数据包出错后，其后继续送来的正确 (但失序) 的包直接抛弃掉，并重发 ACK 以期待发送端超时重传，直至接收端收到预期的包。这样做的好处是接受端设计简单，不必设计缓冲区，为实现流量控制仅需在发送端维护一个发送窗口即可。

2.3 滑动窗口算法

为实现 GBN 协议的滑动窗口算法，其维护 3 个变量，发送窗口大小 $windowSize$，最早未确认的分组序号 $sendBase$，下一个待发送的序号 $nextSeqNum$，并始终维护如下不等式成立:

$$nextSeqNum < sendBase + windowSize$$

在本次实验中，由于采用固定窗口大小，所以上式中 $windowSize$ 为一常量。当一个确认到达时，发送方向右移动 $sendBase$，从而允许发送方发送另一帧。同时，发送方为始终对 $sendBase$ 处的数据包维护一个定时器，如果定时器在 ACK 到达之前超时，则重发此数据包。同时需要注意的是，发送方必须存储 $sendBase$ 到 $nextSeqNum$ 之间的所有数据包，因为在它们得到确认之前必须准备重发。

对于接收端，GBN 协议要求其必须按序接受发送端数据包，当遇到一个正确（但失序）的包后，接受端直接丢弃该包，并发送上一个已经按序接受的包的 ACK, 以发送端再次发送丢失的包，当重复接收到一个已经收到的包时，接收端不做任何动作，即直接抛弃该包。

在本次实验中我采用的 GBN 协议的滑动窗口算法的一个示意图如下:

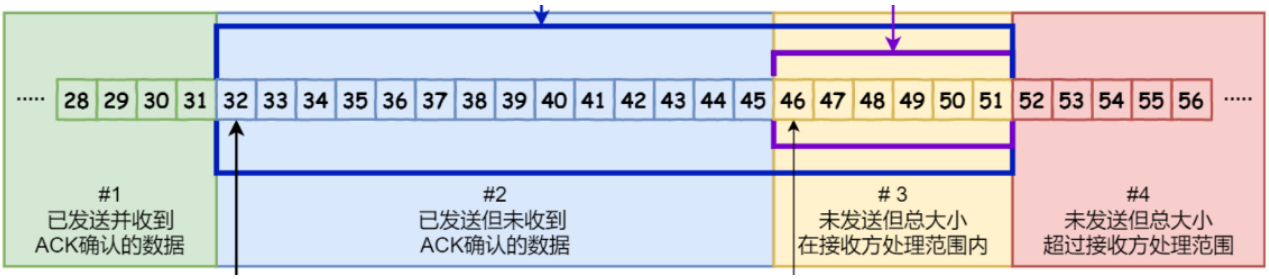

图 2.5: 滑动窗口示意图

3 具体实现

3.1 基于滑动窗口的流量控制机制

发送端负责发送的的线程的核心代码如下图所示：

```
i=0;
while(i<dataNum)
{
    if(buffer[i].seqnum+maxDataBufferSize>=sendBase+windowSize)
    {
        Sleep(200);
        continue;
    }
    sd(i);
    if(noTimer)
    {
        noTimer=0;
        timeoutData=i;
        CreateThread(NULL,NULL,&handleTimer,NULL,0,NULL);
    }
    i++;
}
```

图 3.6: 滑动窗口机制实现

第一个 *if* 语句用于实现固定窗口大小的流量控制机制，每隔 0.2 秒检测一次当前是否满足 2.3 节中的不等式，若满足不等式，则可继续发送，若不满足不等式，则阻塞直到可以发送为止。这里使用睡眠的方式来等待不等式满足条件，但实际中理想实现应该是通过信号量机制，即当接受端由于接受到符合预期的数据包导致 *sendBase* 右移时向发送端发送一个表示可以继续发送的信号量，但在实现过程中，我发现使用信号量也是相当消耗系统资源的，所以我采用了消耗系统资源小，且实现容易的睡眠等待方式来实现这个机制。

3.2 累计确认机制实现

实际中主要由接收端来实现累计确认机制，其实现代码如下图所示：

```
while(1)
{
    memset(&receiveData,0,datagramLen);
    rd();
    checkSum(receiveData);
    if(receiveData.seqnum==nextAckNum)
    {
        nextAckNum+=receiveData.dataLen;
        fwrite(receiveData.data,receiveData.dataLen,1,p);
        showDataInfo(receiveData);
        memset(&sendData,0,datagramHeaderLen);
        sendData.seqnum=1;
        sendData.ack=1;sendData.acknum=nextAckNum;
        sendData.rwnd=receiveData.rwnd;
        checkSum(sendData);
        if(receiveData.fin==1){
            sendData.fin=1;
            sd();
            break;
        }
        sd();
    }
    else if(receiveData.seqnum>nextAckNum)
    {
        printf("unexpected data ");
        sd();
        showDataInfo(receiveData);
    }
    else if(receiveData.seqnum<nextAckNum)
    {
        printf("duplicate data ");
        showDataInfo(receiveData);
    }
```

图 3.7: 累计确认机制实现

当一个具有顺序列号 $SeqNum$ 的包到达时，接收方采取如下行动：如果 $receiveData.seqnum == nextAckNum$ 表示当前接收的包在滑动窗口内，即这就是我们想要接收的包；如果 $receiveData.seqnum < nextAckNum$，表明这是重复发送的包，接收端直接抛弃掉即可；如果 $receiveData.seqnum > nextAckNum$，表明当前接受的是一个失序的包，此时重发 ACK，也就是说即使已经收到更高序号的分组，接收方仍仅确认当前已经按序收到的 $SeqNum$ 的最大值，这种确认被称为是累积的 $(cumulative)$。

4 传输速率测量

本次实验中，我对传输速率随丢包率与路由器延时的变化情况进行了测量，并与实验 3.1 中设计的程序进行了对比，接下来的测试结果都是重复测量五次并取平均值的结果。

首先比较传输速率随丢包率的变化，在测量过程中路由器时延设为 0ms

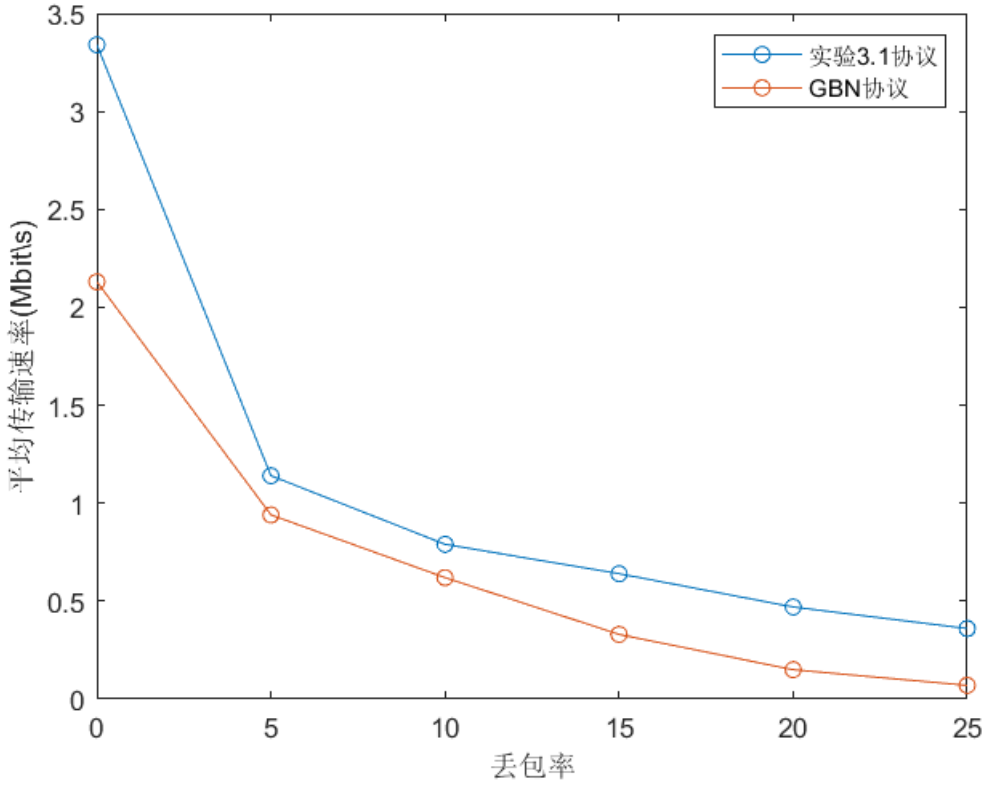

图 4.8: 传输速率随丢包率的变化

可以看到，当初始丢包率较小时，由于有流量控制的功能，GBN 协议的传输速率明显要慢一些，当丢包率为 5% 左右时，两协议传输速率相差不大，但当丢包率进一步增大，由于 GBN 协议丢弃失序分组的特性，其传输速率几乎已经变得很小。

接下来比较传输速率随路由器时延的变化，在测量过程中丢包率设为 0%

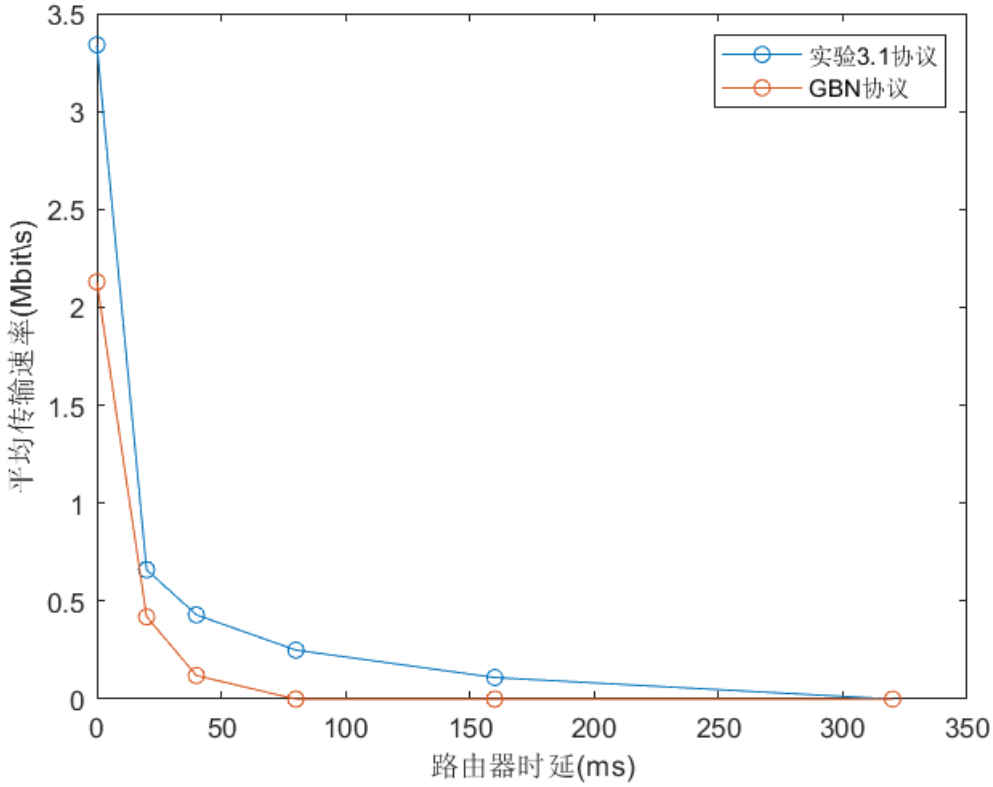

图 4.9: 传输速率随路由器时延的变化

可以看到路由器时延对两协议的影响都非常大，实际测试中我发现当时延大于 $120ms$ 时 GBN 协议传输速率已经让人无法接受，所以我直接将上图中对应位置的传输速率设为 0.

5 实验中遇到的问题

由于我采用的机制是 GBN（退回 N 步），所以当出现单个分组的差错时就会引起大量的重传发送，而且这些重传都是有计时器超时来发现的，这会大大拖垮程序的性能。最初我在超时重传的函数中是这样设计的:

```
void CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT idEvent,DWORD dwTime)
{
    if(SOCKET_ERROR==sendto(clientSocket,(char*)&(*timeoutData),datagramLen,0,(SOCKADDR*)&destinationAddrInfo,addrLen))
    {
        printf("send error\n");
    }
}
```

图 5.10: 超时重传 0

我仅仅重传了设置计时器时 $sendBase$ 处的单个数据包，但是由于接收方丢弃了所有失序分组，所以当超时事件发送时意味着从 $sendBase$ 到下一个要发送的 $nextSeqNum$ 之间的所有包接收方都未收到，在我实际测试中，采用这样的方式发送一个文件所需要的时间几乎是不可接受的，我对给出的 $1.jpg$ 这张图片进行测试，得到的数据如下图所示：

```
file size:1857353 bytes
file transmission time:97094.000000 ms
throughput:0.145946 Mbit\s
```

图 5.11: 测试 0

为了改善这一情况，我采用了类似 *reno* 算法的快速恢复机制来解决这一问题，即当遇到超时事件时不仅重发 $sendBase$ 处的数据包，而且同时发送 $sendBase$ 到 $nextSeqNum$ 处的所有数据包，这样就可以有效解决上述问题了。

改正后的超时重传函数如下:

```
void CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT idEvent,DWORD dwTime)
{
    int t=i;
    for(int j=timeoutData;j<=t;j++)
        sd(j);
}
```

图 5.12: 超时重传 1

再次测试的结果如下：可以看到，性能提升了 $0.762/0.146 = 5.21$ 倍左右，基本达到了预期的要求。

```
file size:1857353 bytes
file transmission time:18596.000000 ms
throughput:0.762018 Mbit\s
```

图 5.13: 测试 1