基于UDP服务实现可靠传输

0 作业要求

> 任务3-1：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。流量控制采用停等机制，完成给定测试文件的传输。
>
> 任务3-2：在任务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。

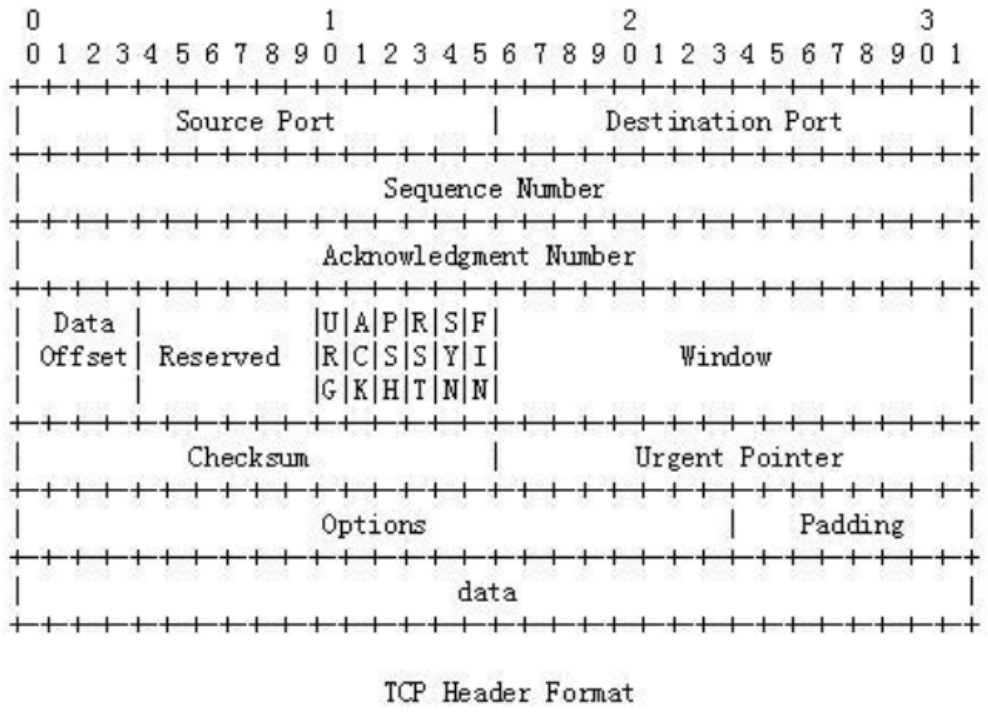
> 任务3-3：在任务3-2的基础上，选择实现一种拥塞控制算法，也可以是改进的算法，完成给定测试文件的传输。
>
> 任务3-4：基于给定的实验测试环境，通过改变延迟时间和丢包率，完成下面3组性能对比实验：（1）停等机制与滑动窗口机制性能对比；（2）滑动窗口机制中不同窗口大小对性能的影响；（3）有拥塞控制和无拥塞控制的性能比较。
>
> **实验要求：**
>
> (1) 实现单向传输。
>
> (2) 对于每一个任务要求给出详细的协议设计。
>
> (3) 给出实现的拥塞控制算法的原理说明。
>
> (4) 完成给定测试文件的传输，显示传输时间和平均吞吐率。
>
> (5) 性能测试指标包括吞吐率和时延，给出图形结果并进行分析。
>
> (6) 完成详细的实验报告（每个任务完成一份）。
>
> (7) 编写的程序应结构清晰，具有较好的可读性。
>
> (8) 提交程序源码和实验报告。

1 实现思路

参考TCP在不可靠的网络层服务之上实现可靠传输的各种机制，在应用层添加这些机制，以实现在传输层中不可靠的UDP服务之上提供可靠传输

1.1 TCP实现可靠传输的机制

1.1.1 报文分析



TCP Header Format

源端口（Source Port）：长度为16 bits（2个字节）。源端口。

目的端口（Destination Port）：长度为16 bits（2个字节）。目的端口。

序列号（Sequence Number）：长度为32 bits（4个字节）。指定了当前数据分片中分配给第一字节数据的序列号。在TCP传输流中每一个字节为一个序号。如果TCP报文中flags标志位为SYN，该序列号表示初始化序列号(ISN)，此时第一个数据应该是从序列号ISN+1开始。

确认序列号（Acknowledgment Number）：长度为32bits（4个字节）。表示TCP发送者期望接受下一个数据分片的序列号。该序号在TCP分片中Flags标志位为ACK时生效。序列号分片的方向和流的方向同方向，而确认序列号分片方向和流方向反方向。

数据偏移或首部长度（Data Offset/Header Length）: 长度为4bits。数据偏移也叫首部长度。因为首部长度实际也说明了数据区在分片中的起始偏移值。它表示TCP头包含了多少个32-bit的words。因为4bits在十进制中能表示的最大值为15，32bits表示4个字节，那么Data Offset的最大可表示15*4=60个字节。所以TCP报头长度最大为60字节。如果options fields为0的话，报文头长度为20个字节。

预留字段（Reserved field）：长度为6bits。值全为零。预留给以后使用。

标志位(Flags): 长度为6bits。表示TCP包特定的连接状态。一个标签位占一个bit，从低位到高位值依次为FIN,SYN,RST,PSH,ACK,URG。新定义的TCP头还扩展了ECE,CWR,NS.

窗口（Window）：长度16bits（2个字节）。表示滑动窗口的大小，用来告诉发送端接收端的buffer space的大小。接收端buffer大小用来控制发送端的发送数据数率，从而达到流量控制。最大值为65535.

校验和（Checksum）：长度16bits（2个字节）。用来检查TCP头在传输中是否被修改。

紧急指针（Urgent pointer）：长度为16bits（2个字节）。表示TCP片中第一个紧急数据字节的指针。只有当URG标志置1时紧急指针才有效。

选项和填充（Option和pading）：可变长度。表示TCP可选选项以及填充位。当选项不足32bits时，填充字段加入额外的0填充。

数据（Data）：长度可变。用来存储上层协议的数据信息。可以为空。比如在连接建立和连接中止时。

接下来具体分析TCP是如何依靠上述TCP报文实现可靠传输的：

> TCP为每个客户数据配上一个TCP首部，从而形成多个TCP报文段，这些报文段被下传给网络层，网络层将其分别封装在网络层IP数据报中，然后这些IP数据包被发送到网络中，当TCP在另一端收到一个报文段后，该报文段的数据就被放入该TCP连接的接收缓存中，应用程序从此缓存中读取数据流。

1.1.2 连接管理

TCP通过三次握手建立连接

第一次握手

客户端将TCP报文**标志位SYN置为1**，随机产生一个序号值seq=J，保存在TCP首部的序列号(Sequence Number)字段里，指明客户端打算连接的服务器的端口，并将该数据包发送给服务器端，发送完毕后，客户端进入`SYN_SENT`状态，等待服务器端确认。

第二次握手

服务器端收到数据包后由标志位SYN=1知道客户端请求建立连接，服务器端将TCP报文**标志位SYN和ACK都置为1**，ack=J+1，随机产生一个序号值seq=K，并将该数据包发送给客户端以确认连接请求，服务器端进入`SYN_RCVD`状态。

第三次握手

客户端收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给服务器端，服务器端检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，客户端和服务器端进入`ESTABLISHED`状态，完成三次握手，随后客户端与服务器端之间可以开始传输数据了。

1.1.3 确认应答

TCP将数据看成是一个无结构的、有序的字节流。

传送的字节流之上，一个报文段的序号是该报文段首字节的字节流编号。

TCP发送报文时，会携带Seq，接收到该报文段的另一端发送确认报文回去，报文中包含确认号（ack=seq+1），为期待接收的下一字节的序号

1.1.4 超时重传

TCP发送一个报文时，会启动一个计时器，当超出某一时间还未收到该报文的应答时，TCP认为该报文丢失（当然也不一定是该报文丢失，也可能是确认报文丢失，或者是由于网络状况该确认报文还未传输过来），并重传该报文。

1.1.5 滑动窗口

为了提高性能，TCP中还引入了流水线机制，TCP发送端可以一次性发送多个报文段，滑动窗口用于描述接收方的TCP数据报缓冲区大小的数据，发送方根据这个数据来计算自己最多能发送多长的数据，如果发送方收到接收方的窗口大小为0的TCP数据报，那么发送方将停止发送数据，等到接收方发送窗口大小不为0的数据报的到来。这也是TCP实现流量控制的机制。

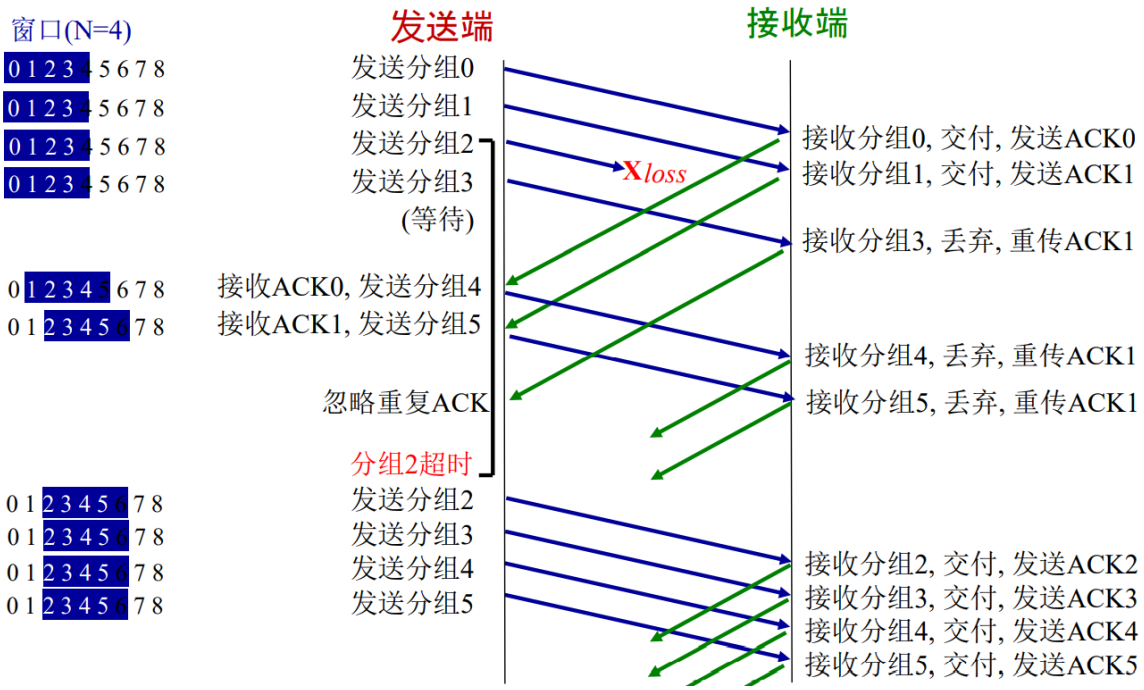
1.1.6　GBN＆SR

GBN（Go back N）回退N

采用累积确认的方式，接收方只确认连续接收分组的最大序列号，其余的全部丢弃。

发送端设置定时器，定时器超时时，重传所有未确认的分组

■ GBN交互示例



SR（Selective Repeat ）选择重传

发送端

• 接收上层数据：如果发送窗口中有可用的序号，则发送分组

• 超时(n)：重传分组n，重启定时器

• 接收ACK(n)：n在[send_base, send_base+N-1]区间，将分组n标记为已接 收，如果是窗口中最小的未确认的分组，则窗口向前滑动，基序号为下 一个未确认分组的序号

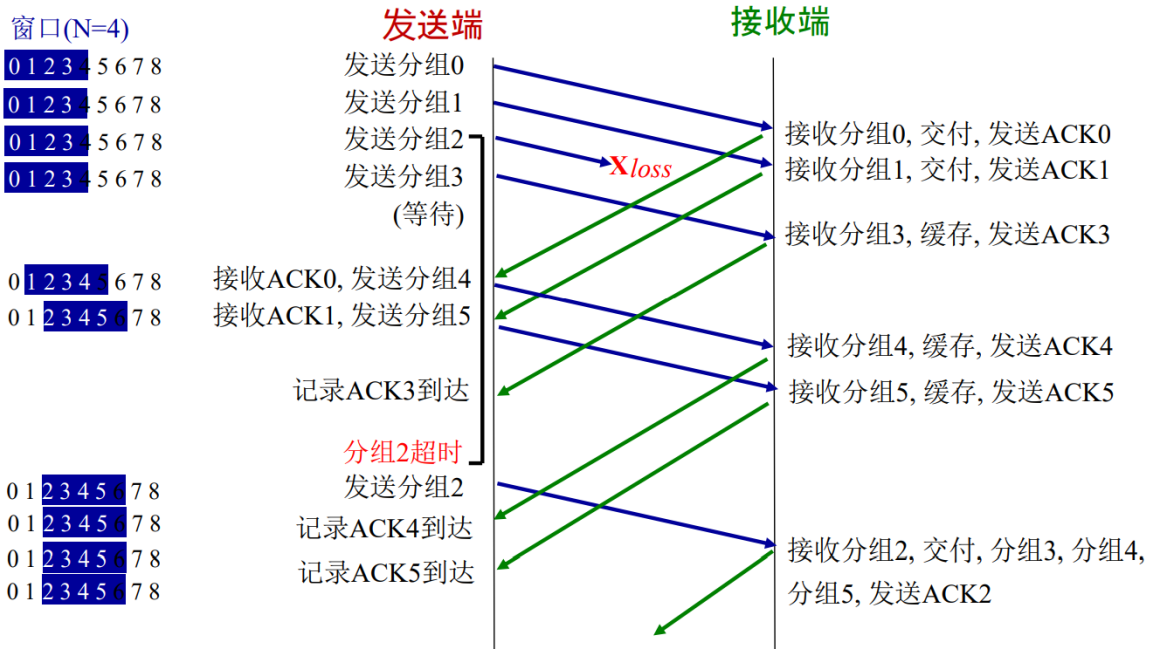
接收端

接收分组n：

• n在[rcv_base, rcv_base+N-1]区间，发送ACK(n)，缓存失序分组，按序到 达的分组交付给上层，窗口向前滑动

• n在[rcv_base-N, rcv_base-1]区间，发送ACK(n)

■ SR交互示例



1.1.7 拥塞控制

拥塞控制的几种算法：

慢开始(slow-start)、拥塞避免(congestion avoidance)、快重传(fast retransmit)和快恢复(fast recovery)

发送方维持一个拥塞窗口 cwnd (congestion window)的状态变量。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。发送方让自己的发送窗口等于拥塞。

发送方控制拥塞窗口的原则是：只要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去。但只要网络出现拥塞，拥塞窗口就减小一些，以减少注入到网络中的分组数。

慢开始算法：当主机开始发送数据时，如果立即所大量数据字节注入到网络，那么就有可能引起网络拥塞，因为现在并不清楚网络的负荷情况。因此，较好的方法是 先探测一下，即由小到大逐渐增大发送窗口，也就是说，由小到大逐渐增大拥塞窗口数值。通常在刚刚开始发送报文段时，先把拥塞窗口 cwnd 设置为一个最大报文段MSS的数值。而在每收到一个对新的报文段的确认后，把拥塞窗口增加至多一个MSS的数值。用这样的方法逐步增大发送方的拥塞窗口 cwnd ，可以使分组注入到网络的速率更加合理。

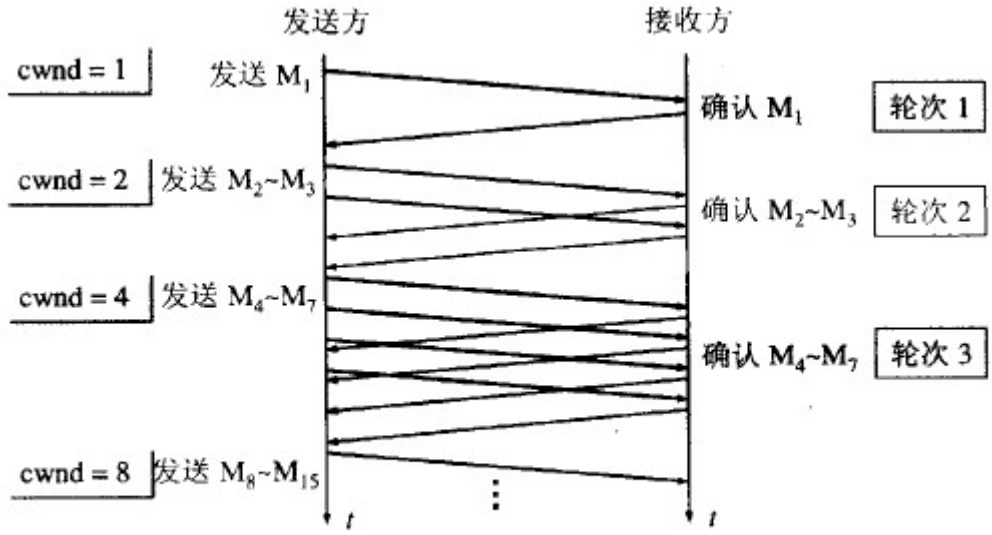


图 5-24　发送方每收到一个确认就把窗口 cwnd 加 1

为了防止拥塞窗口cwnd增长过大引起网络拥塞，还需要设置一个慢开始门限ssthresh状态变量（如何设置ssthresh）。慢开始门限ssthresh的用法如下：

当 cwnd < ssthresh 时，使用上述的慢开始算法。

当 cwnd > ssthresh 时，停止使用慢开始算法而改用拥塞避免算法。

当 cwnd = ssthresh 时，既可使用慢开始算法，也可使用拥塞控制避免算法。

拥塞避免算法：让拥塞窗口cwnd缓慢地增大，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍。这样拥塞窗口cwnd按线性规律缓慢增长，比慢开始算法的拥塞窗口增长速率缓慢得多。

如果发送方设置的超时计时器时限已到但还没有收到确认，那么很可能是网络出现了拥塞，致使报文段在网络中的某处被丢弃。这时，TCP马上把拥塞窗口 cwnd 减小到1，并执行慢开始算法，同时把慢开始门限值ssthresh减半。这是不使用快重传的情况。

快重传算法首先要求接收方每收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时才进行捎带确认。

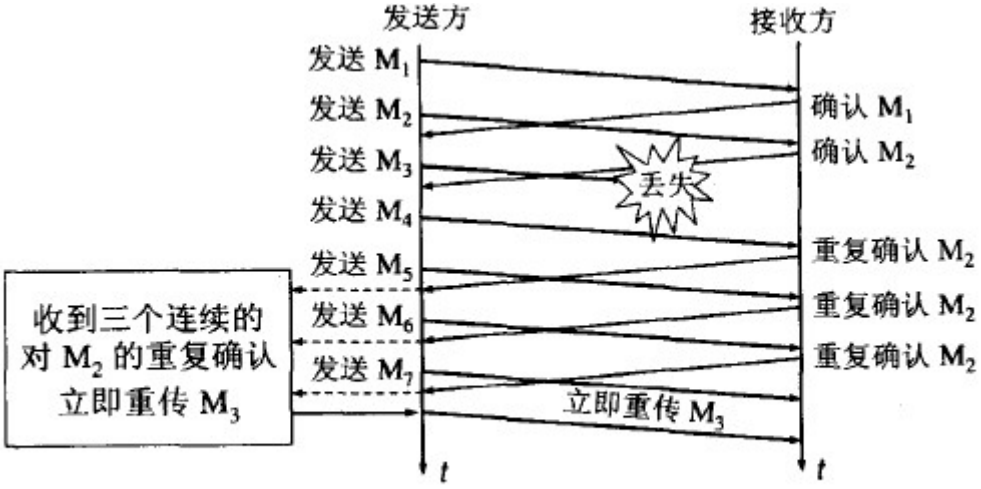


图 5-26　快重传的示意图

接收方收到了M1和M2后都分别发出了确认。现在假定接收方没有收到M3但接着收到了M4。显然，接收方不能确认M4，因为M4是收到的失序报文段。根据 可靠传输原理，接收方可以什么都不做，也可以在适当时机发送一次对M2的确认。但按照快重传算法的规定，接收方应及时发送对M2的重复确认，这样做可以让 发送方及早知道报文段M3没有到达接收方。发送方接着发送了M5和M6。接收方收到这两个报文后，也还要再次发出对M2的重复确认。这样，发送方共收到了 接收方的四个对M2的确认，其中

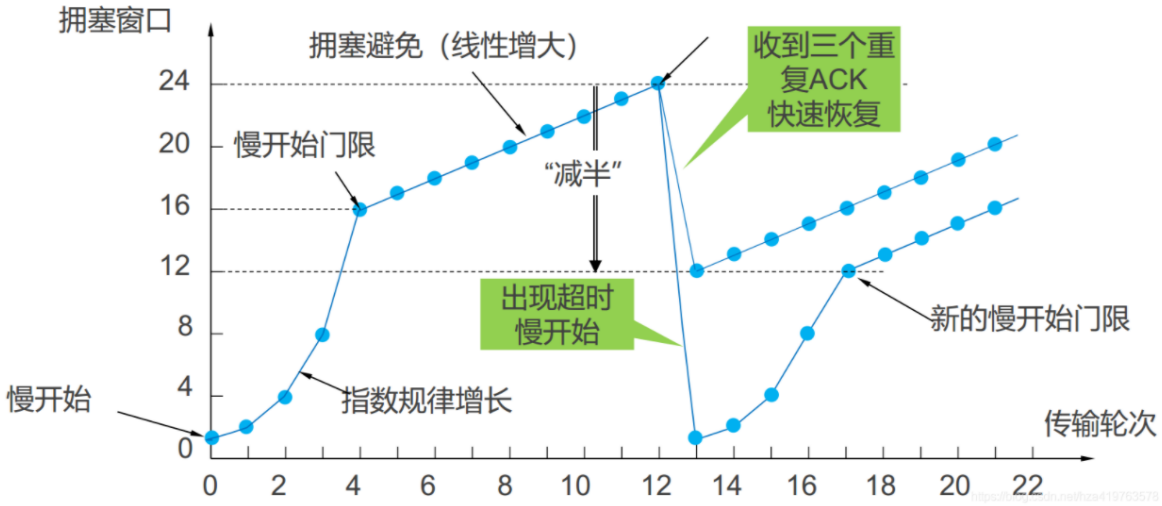
后三个都是重复确认。快重传算法还规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段M3，而不必 继续等待M3设置的重传计时器到期。由于发送方尽早重传未被确认的报文段，因此采用快重传后可以使整个网络吞吐量提高约20%。

与快重传配合使用的还有快恢复算法，其过程有以下两个要点：

<1>. 当发送方连续收到三个重复确认，就执行“乘法减小”算法，把慢开始门限ssthresh减半。这是为了预防网络发生拥塞。请注意：接下去不执行慢开始算法。

<2>. 由于发送方现在认为网络很可能没有发生拥塞，因此与慢开始不同之处是现在不执行慢开始算法（即拥塞窗口cwnd现在不设置为1），而是把cwnd值设置为 慢开始门限ssthresh减半后的数值，然后开始执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地线性增大。

下图给出了快重传和快恢复的示意图，并标明了“TCP Reno版本”。



1.1.8 累积确认

TCP最初的设计是对收到的报文段进行积累确认。接收方通告它期望接收的下一个字节的序号，并忽略所有失序到达并被保存的报文段。有时这被称为肯定积累确认或ACK。“肯定”这个词表示对于那些丢弃的、丢失的或重复的报文段都不提供反馈。在TCP首部的32位ACK字段用于积累确认，而它的值仅在ACK标志为1时才有效。

1.1.9 差错检测

每个报文都包含了一个检验和字段，用来检查报文段是否收到损伤。如果某个报文段因检验和无效而被检查出受到损伤，就由终点TCP将其丢弃，并被认为是丢失了。TCP规定每个报文段都必须使用16位的检验和。

1.2 基于UDP服务实现可靠传输协议设计

通过上述分析，我们可以看到TCP是如何通过一系列的机制实现了在不可靠的网络层的服务之上提供可靠的数据传输的，接下来我们通过在应用层实现以上机制，来达到基于UDP服务实现可靠数据传输的目标。

1、参考SYN/ACK/FIN机制，增加建立连接机制

2、增加确认应答机制（重点所在）

3、增加缓冲区，实现滑动窗口机制

4、增加超时重传机制

5、增加累积确认机制

6、增加GBN机制

1.2.1 对UDP报文进行封装

通过对UDP报文进行进一步封装，以实现上述功能。

Sequence Number		
Acknowledge Number		
TAG	Checksum	WINDOWSIZE
Source port		Destination port
Length		Checksum
Data		

TAG	SEQ	ACK	Length	Checksum	Window Size	Data

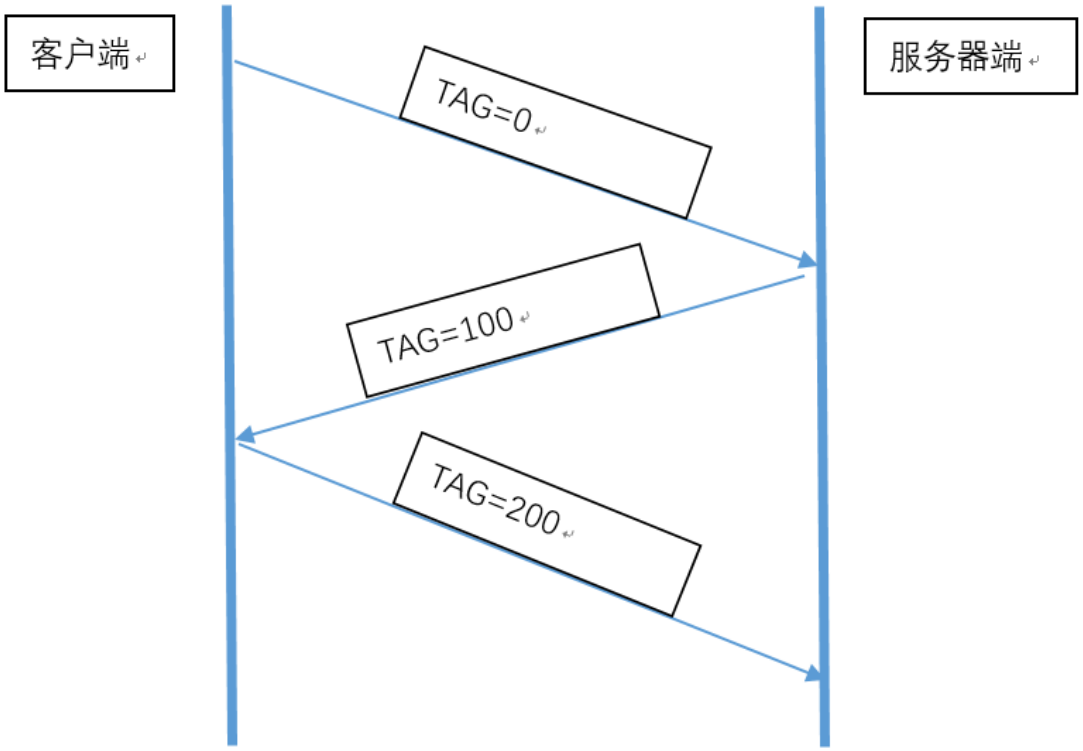
1.2.2 面向连接

由于UDP已经能够提供多路复用 / 多路分解功能，为了实现建立连接，我们仅需要一个简化的TCP建立建立过程，来标识建立连接，过程如下：

①首先客户端向服务器端发送一个一个数据报数据部分为空，TAG=0，标识请求建立连接

②服务器收到请求后，返回TAG=100，标识允许建立连接

③客户端收到服务器反馈后，向服务器发送TAG=200，标识可以开始传输



断开连接过程：

①服务器端端发送TAG=88标识文件传输完毕请求断开连接

②客户收到即断开连接

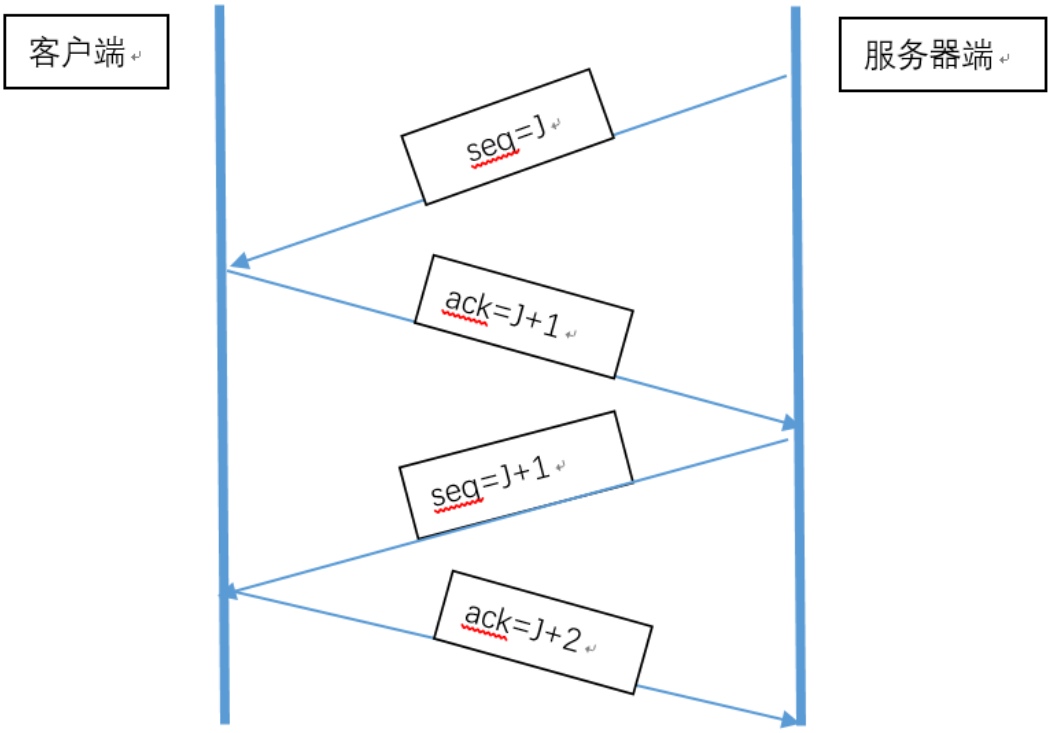
1.2.3 确认应答

参考TCP的seq/ack机制，在对UDP封装是时候增加seq和ack字段，以实现确认应答，过程如图

（注意这里的seq为报文段的序号）考虑单向传输

①服务器端给客户端发送seq=J的报文

②客户端收到后确认，发送ack=J+1.为期待接收的下一报文段



1.2.4 滑动窗口

客户端和服务器端均维护一个缓冲区，假设两端的滑动窗口大小为N，服务器端可以一次性发送N个报文段。以实现流水线机制，目的为提高传输的性能，下文将给出GBN和SR的C++实现。

1.2.5 超时重传

服务器端每发送一个报文时，启动一个计时器，当超时时，重发该数据报。

1.2.6 差错检测

模仿tcp，发送方发送报文前先计算checksum并封装到包内，接收方收到包进行校验，如果正确则正确接收

1.2.7 拥塞控制

采用Reno算法

2、C++实现（关键代码）

2.1数据包格式

```
struct packet
{
    unsigned char tag;//连接建立、断开标识
    unsigned int seq;//序列号
    unsigned int ack;//确认号
    unsigned short len;//数据部分长度
    unsigned short checksum;//校验和
    unsigned short window;//窗口
    char data[1024];//数据长度

    void init_packet()
    {
        this->tag = -1;
```

```
        this->seq = -1;
        this->ack = -1;
        this->len = -1;
        this->checksum = -1;
        this->window = -1;
        ZeroMemory(this->data, 1024);
    }
};
```

2.2 socket基本函数

服务器端

```
//初始化工作
void inithandler()
{
    WORD wVersionRequested;
    WSADATA wsaData;
    //套接字加载时错误提示
    int err;
    //版本 2.2
    wVersionRequested = MAKEWORD(2, 2);
    //加载 dll 文件 Scoket 库
    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0) {
        //找不到 winsock.dll
        cout << "WSAStartup failed with error: " << err << endl;
        return;
    }
    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2)
    {
        cout << "Could not find a usable version of Winsock.dll" << endl;
        WSACleanup();
    }
    sockServer = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    //设置套接字为非阻塞模式
    int iMode = 1; //1：非阻塞，0：阻塞
    ioctlsocket(sockServer, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置

    addrServer.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
    addrServer.sin_family = AF_INET;
    addrServer.sin_port = htons(SERVER_PORT);
    err = bind(sockServer, (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
    if (err) {
        err = GetLastError();
        cout << "Could  not  bind  the  port" << SERVER_PORT << "for  socket.
Error  code is" << err << endl;
        WSACleanup();
        return;
    }
    else
    {
        cout << "服务器创建成功" << endl;
    }
```

```
    for (int i = 0; i < WINDOWSIZE; i++)
    {
        ack[i] = 1;//初始都标记为1
    }
}
```

客户端

```
//初始化工作
void init()
{
    //加载套接字库（必须）
    WORD wVersionRequested;
    WSADATA wsaData;
    //套接字加载时错误提示
    int err;
    //版本 2.2
    wVersionRequested = MAKEWORD(2, 2);
    //加载 dll 文件 Scoket 库
    err = WSAStartup(wVersionRequested, &wsaData);
    if (err != 0)
    {
        //找不到 winsock.dll
        cout<<"WSAStartup failed with error: "<<err<<endl;
        return ;
    }
    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2)
    {
        cout<<"Could not find a usable version of Winsock.dll"<<endl;
        WSACleanup();
    }
    else
    {
        cout<<"套接字创建成功"<<endl;
    }
    socketClient = socket(AF_INET, SOCK_DGRAM, 0);
    addrServer.sin_addr.S_un.S_addr = inet_addr(SERVER_IP);
    addrServer.sin_family = AF_INET;
    addrServer.sin_port = htons(SERVER_PORT);

}
```

2.3 建立连接

服务器收到客户端发来的TAG=0的数据报，标识请求连接
服务器向客户端发送一个 100 大小的状态码，表示服务器准备好了，可以发送数据
客户端收到 100 之后回复一个 200 大小的状态码，表示客户端准备好了，可以接收数据了
服务器收到 200 状态码之后，就开始发送数据了

并在此过程中传输文件相关信息，如文件名，文件大小等

服务器端

```
//握手建立连接阶段
//服务器收到客户端发来的TAG=0的数据报，标识请求连接
//服务器向客户端发送一个 100 大小的状态码，表示服务器准备好了，可以发送数据
```

```
//客户端收到 100 之后回复一个 200 大小的状态码，表示客户端准备好了，可以接收数据了
//服务器收到 200 状态码之后，就开始发送数据了
if (pkt->tag == 0)
{
    clock_t st = clock();//开始计时
    cout << "开始建立连接..." << endl;
    int stage = 0;
    bool runFlag = true;
    int waitCount = 0;
    packet* pkt = new packet;
    while (runFlag)
    {
        switch (stage)
        {
            case 0://发送100阶段
                pkt = connecthandler(100, totalpacket);
                sendto(sockServer, (char*)pkt, sizeof(*pkt), 0,
(SOCKADDR*)&addrClient, sizeof(SOCKADDR));
                //sendto(sockServer, buffer, BUFFER, 0, (SOCKADDR*)&addrClient,
sizeof(SOCKADDR));
                Sleep(100);
                stage = 1;
                break;
            case 1://等待接收200阶段
                ZeroMemory(pkt, sizeof(*pkt));
                recvSize = recvfrom(sockServer, (char*)pkt, sizeof(*pkt), 0,
((SOCKADDR*)&addrClient), &length);
                if (recvSize < 0)
                {
                    ++waitCount;
                    Sleep(200);
                    if (waitCount > 20)
                    {
                        runFlag = false;
                        cout << "连接建立失败！等待建立新连接..." << endl;
                        break;
                    }
                    continue;
                }
                else
                {
                    if (pkt->tag == 200)
                    {
                        pkt->init_packet();
                        cout << "开始文件传输..." << endl;
                        memcpy(pkt->data, filepath, strlen(filepath));
                        pkt->len = strlen(filepath);
                        sendto(sockServer, (char*)pkt, sizeof(*pkt), 0,
(SOCKADDR*)&addrClient, sizeof(SOCKADDR));
                        stage = 2;
                    }
                }
                break;
            case 2:
                if (totalack == totalpacket)//数据包传输完毕
                {
                    pkt->init_packet();
                    pkt->tag = 88;
```

```
                    cout << "************************************" << endl;
                    cout << "数据传输成功！" << endl;
                    cout << "传输用时: " << (clock() - st) * 1000.0 /
CLOCKS_PER_SEC << "ms" << endl;
                    sendto(sockServer, (char*)pkt, sizeof(*pkt), 0,
(SOCKADDR*)&addrClient, sizeof(SOCKADDR));
                    runFlag = false;
                    exit(0);
                    break;
                }
        }
    }
}
```

客户端

```
pkt->tag = 0;
        sendto(socketClient, (char*)pkt, sizeof(*pkt), 0,
(SOCKADDR*)&addrServer, sizeof(SOCKADDR));
        while (true)
        {
            //等待 server 回复
            switch (stage)
            {
            case 0://等待握手阶段
                recvfrom(socketClient, (char*)pkt, sizeof(*pkt), 0,
(SOCKADDR*)&addrServer, &len);
                totalpacket = pkt->len;
                cout << "准备建立连接，总共有" << totalpacket << "个数据包" << endl;
                pkt->init_packet();
                pkt=connecthandler(200);
                sendto(socketClient, (char*)pkt,sizeof(*pkt) , 0,
(SOCKADDR*)&addrServer, sizeof(SOCKADDR));
                stage = 1;
                break;
            case 1:
                recvfrom(socketClient, (char*)pkt, sizeof(*pkt), 0,
(SOCKADDR*)&addrServer, &len);
                memcpy(filename, pkt->data, pkt->len);
                out_result.open(filename, std::ios::out | std::ios::binary);
                cout << "文件名为：" << filename << endl;
                if (!out_result.is_open())
                {
                    cout << "文件打开失败！！！" << endl;
                    exit(1);
                }
                stage = 2;
                break;
```

2.4 超时重传

2.4.1GBN

服务器端 维护一个窗口大小的缓冲区，每次发送数据时向缓冲区内也拷贝一份数据 超时后重发窗口内的所有数据包

```
//超时重传
void timeouthandler()
{
    BOOL flag=false;
    packet* pkt = new packet;
    if (ack[curack % WINDOWSIZE] == 2)//快速重传之后还有没被确认的，认为包丢失
    {
        for (int i = curack; i != curseq; i = (i++) % seqnumber)
        {
            memcpy(pkt, &buffer[i % WINDOWSIZE], BUFFER);
            sendto(sockServer, (char*)pkt, sizeof(packet), 0,
(SOCKADDR*)&addrClient, sizeof(SOCKADDR));
            cout << "重传第 " << i << " 号数据包" << endl;
            flag = true;
        }
    }
    if(flag==true)
    {
        ssthresh = cwnd / 2;
        cwnd = 1;
        STATE = SLOWSTART;//检测到超时，就回到慢启动状态
        cout << "============================检测到超时，回到慢启动阶段
============================" << endl;
        cout << "cwnd=  " << cwnd << "     sstresh= " << ssthresh << endl <<
endl;
    }
}
```

客户端 如果是期望的数据包就正确接收，否则发送上一ACK

```
//GBN实现
if (pkt->seq == waitseq && totalrecv < totalpacket&&!corrupt(pkt))
{

    b = lossInLossRatio(packetLossRatio);
    if (b) {
        cout << "***************第  " << pkt->seq << " 号数据包丢失" << endl <<
endl;
        continue;
    }
    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<收到第" << pkt->seq << "号数据包" <<
endl << endl;
    recvwindow -= BUFFER;
    out_result.write(pkt->data, pkt->len);
    recvwindow += BUFFER;
    make_mypkt(pkt, waitseq, recvwindow);
    cout << "发送对第" << waitseq << "号数据包的确认" << endl;
    sendto(socketClient, (char*)pkt, sizeof(packet), 0, (SOCKADDR*)&addrServer,
sizeof(SOCKADDR));
    waitseq++;
    waitseq %= seqnumber;
```

```
    totalrecv++;
}
else
{
    make_mypkt(pkt, waitseq - 1, recvwindow);
    cout << "**********不是期待的数据包，发送了一个重复ack" << waitseq - 1 << endl;
    sendto(socketClient, (char*)pkt, sizeof(packet), 0, (SOCKADDR*)&addrServer,
sizeof(SOCKADDR));
}
```

2.4.2 SR

发送方同样维护窗口大小的缓冲区，接收方也维护一个窗口大小的缓冲区，对在窗口内的数据包进行确认，发送方超时后重传没有确认的数据包

序号连续即向上层交付

```
if (pkt.seq <= waitseq + windowsize && pkt.seq >=
waitseq&&totalrecv<totalpacket)
{

    b = lossInLossRatio(packetLossRatio);
    if (b) {
        cout << "***************第  " << pkt.seq << " 号数据包丢失" << endl <<
endl;
        continue;
    }
    cout << "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<收到第" << pkt.seq << "号数据包" <<
endl << endl;
    ack_send[pkt.seq - waitseq] = true;
    memcpy(&buffer_1[pkt.seq], &buffer[11], pkt.len);
    buffer[2] = pkt.seq;
    cout << "发送对第" << pkt.seq << "号数据包的确认" << endl;
    sendto(socketClient, buffer, BUFFER, 0, (SOCKADDR*)&addrServer,
sizeof(SOCKADDR));
    int ack_s = 0;
    totalrecv++;
    while (ack_send[ack_s] && ack_s < windowsize)
    {
        out_result.write(buffer_1[ack_s], 1024);
        out_result.flush();
        waitseq++;
        if (waitseq == 20)
            waitseq = 0;
        ack_s += 1;
    }
    //向前滑动窗口
    if (ack_s > 0)
    {
        for (int i = 0; i < windowsize; i++)
        {
            if (ack_s + i < windowsize)
            {
                ack_send[i] = ack_send[i + ack_s];
                memcpy(buffer_1[i], buffer_1[i + ack_s], pkt.len);
                ZeroMemory(buffer_1[i + ack_s], sizeof(buffer_1[i + ack_s]));
            }
```

```
            else
            {
                ack_send[i] = false;
                ZeroMemory(buffer_1[i], sizeof(buffer_1[i]));
            }
        }
    }
}
else if (pkt.seq >= waitseq - windowsize && pkt.seq <= windowsize - 1)
{
    ZeroMemory(buffer, BUFFER);
    buffer[2] = waitseq;
    cout << "不在窗口内，发送了一个重复ACK" << waitseq-1 << endl;
    sendto(socketClient, buffer, 9, 0, (SOCKADDR*)&addrServer,
sizeof(SOCKADDR));
}
break;
```

2.5快速重传

收到三个冗余的ack后就重传

```
//快速重传
void FASTRECOhandler()
{
    packet* pkt = new packet;
    for (int i = curack; i != curseq; i = (i++) % seqnumber)
    {
        memcpy(pkt, &buffer[i % WINDOWSIZE], BUFFER);
        sendto(sockServer, (char*)pkt, sizeof(packet), 0,
(SOCKADDR*)&addrClient, sizeof(SOCKADDR));
        cout << "重传第 " << i << " 号数据包" << endl;
    }
}
```

2.6差错检测

```
/计算校验和
unsigned short makesum(int count, char* buf)
{
    unsigned long sum;
    for (sum = 0; count > 0; count--)
    {
        sum += *buf++;
        sum = (sum >> 16) + (sum & 0xffff);
    }
    return ~sum;
}

// 判断包是否损坏
bool corrupt(packet* pkt)
{
    int count = sizeof(pkt->data) / 2;
    register unsigned long sum = 0;
```

```
    unsigned short* buf = (unsigned short*)(pkt->data);
    while (count--) {
        sum += *buf++;
        if (sum & 0xFFFF0000) {
            sum &= 0xFFFF;
            sum++;
        }
    }
    if (pkt->checksum == ~(sum & 0xFFFF))
        return true;
    return false;
}
```

2.7 拥塞控制

收到ack后，执行reno算法

```
//收到数据包，判断是否是正确的ack，并做相应处理
void ackhandler(unsigned int a)
{
    long long index = a;
    switch (STATE)
    {
    case SLOWSTART:
        if ((index + seqnumber - curack) % seqnumber < minwindow(cwnd,
WINDOWSIZE))
        {
            cout << "<<<<<<<<<<<<<<<<<<<<<收到第" << index << "号数据包的ack" <<
endl << endl;
            ack[index % WINDOWSIZE] = 3;
            if (cwnd <= ssthresh)
            {
                cwnd++;
                cout << "==========================慢启动阶段
============================" << endl;
                cout << "cwnd=  " << cwnd << "     sstresh= " << ssthresh <<
endl << endl;

            }
            else
            {
                STATE = AVOID;
            }
            //累积确认
            for (int j = curack; j != (index + 1) % seqnumber; j = (++j) %
seqnumber)
            {
                ack[j % WINDOWSIZE] = 1;
                ++totalack;
                curack = (curack + 1) % seqnumber;
            }
        }
        else if (index == curack - 1)
        {
            dupack++;
```

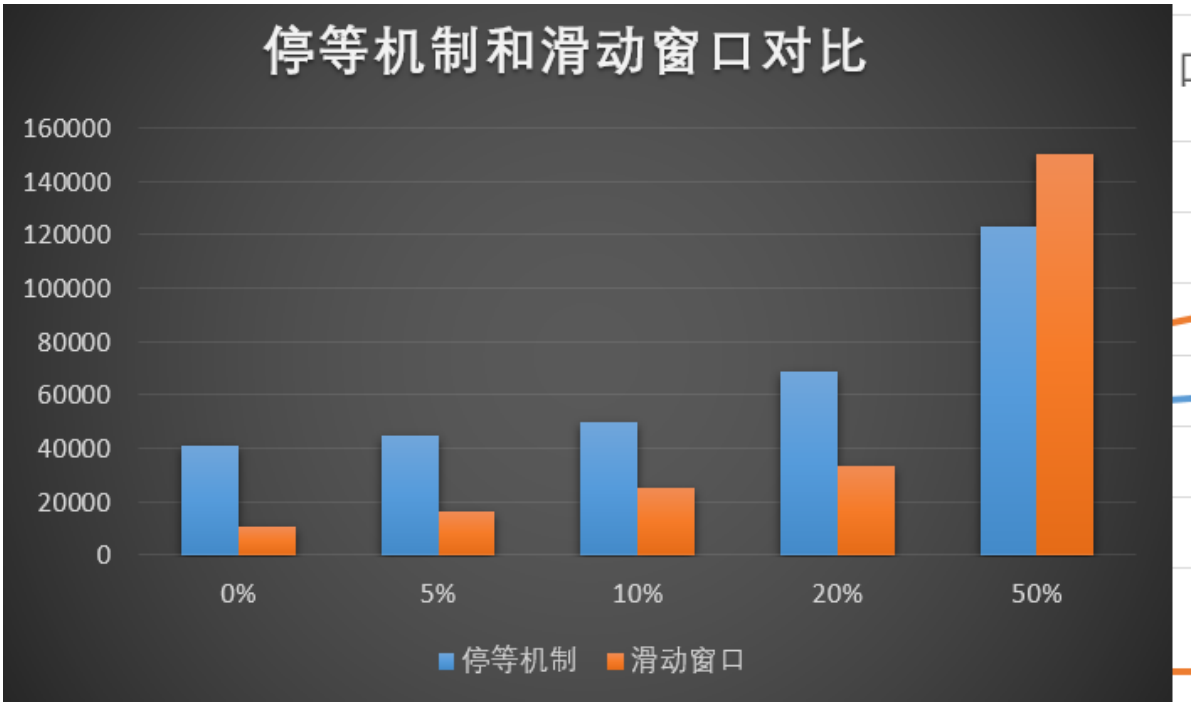
```
            if (dupack == 3)//进入快速重传 状态跳转到拥塞避免
            {
                FASTRECOhandler();
                ssthresh = cwnd / 2;
                cwnd = ssthresh + 3;
                STATE = AVOID;
                dupack = 0;
            }
        }
        break;
    case AVOID:
        if ((index + seqnumber - curack) % seqnumber < minwindow(cwnd,
WINDOWSIZE))
        {
            cout << "<<<<<<<<<<<<<<<<<<<<<收到第" << index << "号数据包的ack" <<
endl << endl;
            ack[index % WINDOWSIZE] = 3;
            cwnd = cwnd + 1 / cwnd;
            cout << "============================达到阈值，进入拥塞避免阶段
============================" << endl;
            cout << "cwnd=  " << int(cwnd) << "     sstresh=" << ssthresh <<
endl << endl;
            //累积确认
            for (int j = curack; j != (index + 1) % seqnumber; j = (++j) %
seqnumber)
            {
                ack[j % WINDOWSIZE] = 1;
                ++totalack;
                curack = (curack + 1) % seqnumber;
            }
        }
        else if (index == curack - 1)
        {
            dupack++;
            if (dupack == 3)
            {
                timeouthandler();
                STATE = AVOID;
                dupack = 0;
            }
        }
        break;
    }

}
```

3性能对比

说明：测试文件大小为18.73MB，每组测试中，采用控制变量法进行性能对比实验，测评指标简单定为传输时长

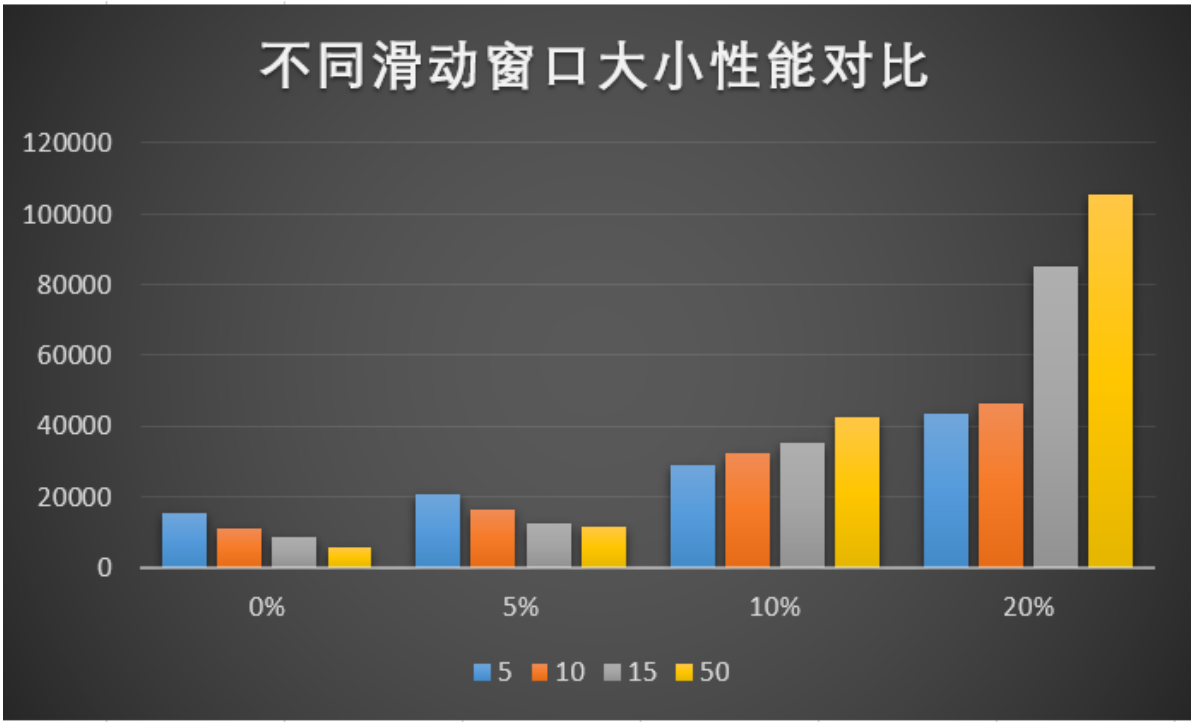
3.1停等机制与滑动窗口机制性能对比



可以看到在丢包率较低时，滑动窗口的传输速率高于停等机制，

但当丢包率较大时，滑动窗口的性能会下降，因为重发的包过多。

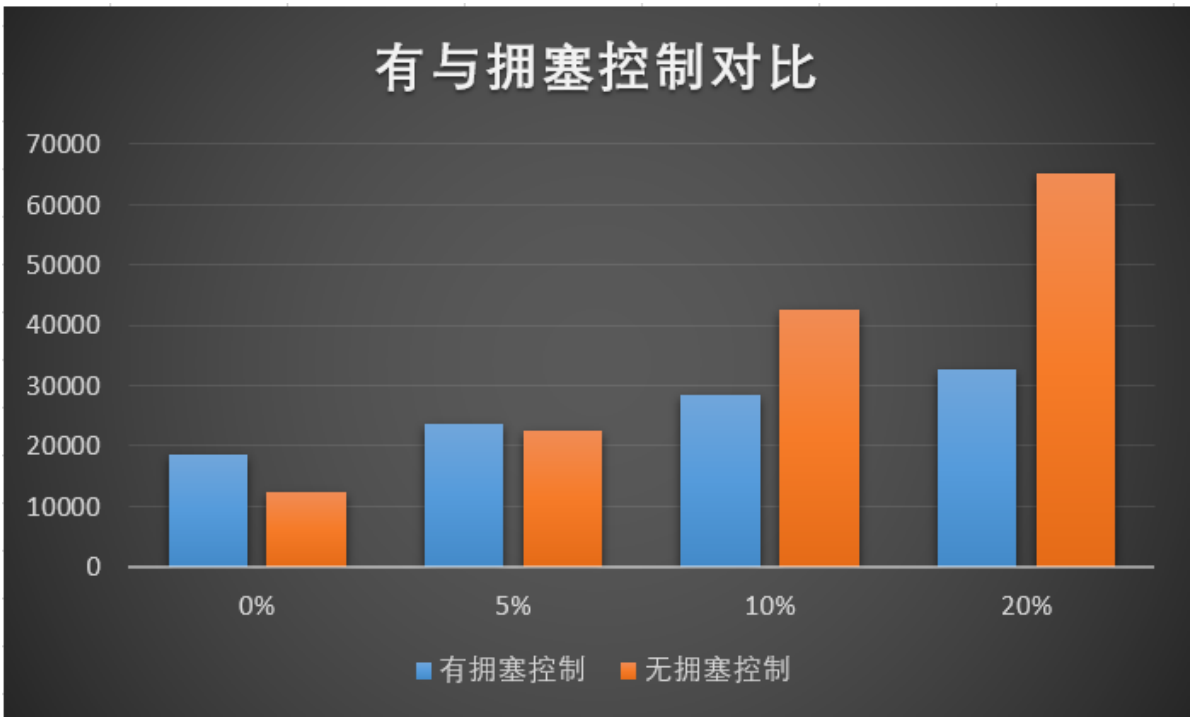
3.2滑动窗口机制中不同窗口大小对性能的影响；



在没有丢包的情况下，窗口的增大会带来传输速率的提升。

在有丢包的情况下，随着窗口的增大，传输速率会有一定提升，但当窗口增长到一定大小时，其带来的性能增长减小，原因在于GBN重发的数据包量增大，拖慢发送速率。

3.3有拥塞控制和无拥塞控制的性能比较



可以看到在丢包率较低的时候，没有拥塞控制似乎表现的更好，但当丢包率上升后，有拥塞控制的优势明显体现出来

4测试

文件	test.rar（18.73 / 18.73MB） 耗时: 2s @ 9.34MB / s
MD5	eb6ef550f6b4205e72d614b5c74cf214
SHA1	4614c8cf3b9b4abdf08ed1bdde1903902d284f63
SHA256	09cd4286f7d48047fe93c8890ff7babdfef0f32d9b936782dbf678045a8815d7

文件	test.rar（18.73 / 18.73MB） 耗时: 2.06s @ 9.05MB / s
MD5	eb6ef550f6b4205e72d614b5c74cf214
SHA1	4614c8cf3b9b4abdf08ed1bdde1903902d284f63
SHA256	09cd4286f7d48047fe93c8890ff7babdfef0f32d9b936782dbf678045a8815d7

原文件于接收到文件的hash值完全一致，得出实验成功的结论

5 总结

通过这次编程作业的练习，通过模仿TCP可靠传输的机制，在UDP服务之上实现了可靠传输，对网络的层次结构有了更深的理解，对可靠传输的基本机制：

①建立连接 ②差错检测 ③确认应答 ④超时重传 ⑤累积确认 ⑥流量控制 ⑦拥塞控制

有了更深的领悟，并且在不断的debug中了解到了很多细节的内容，总之受益良多。

源码链接：https://github.com/jinbaoT/reliable-UDP-transmission