南開大學

# 计算机网络实验报告

## 实验 3-1


专　　业　信息安全

学　　号　2113662

姓　　名　张从

班　　级　信安一班

# 一、实验目的

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、接收确认、超时重传等。流量控制采用停等机制，完成给定测试文件的传输。

# 二、实验要求

（1）实现单向数据传输（一端发数据，一端返回确认）。

（2）对于每个任务要求给出详细的协议设计。

（3）完成给定测试文件的传输，显示传输时间和平均吞吐率。

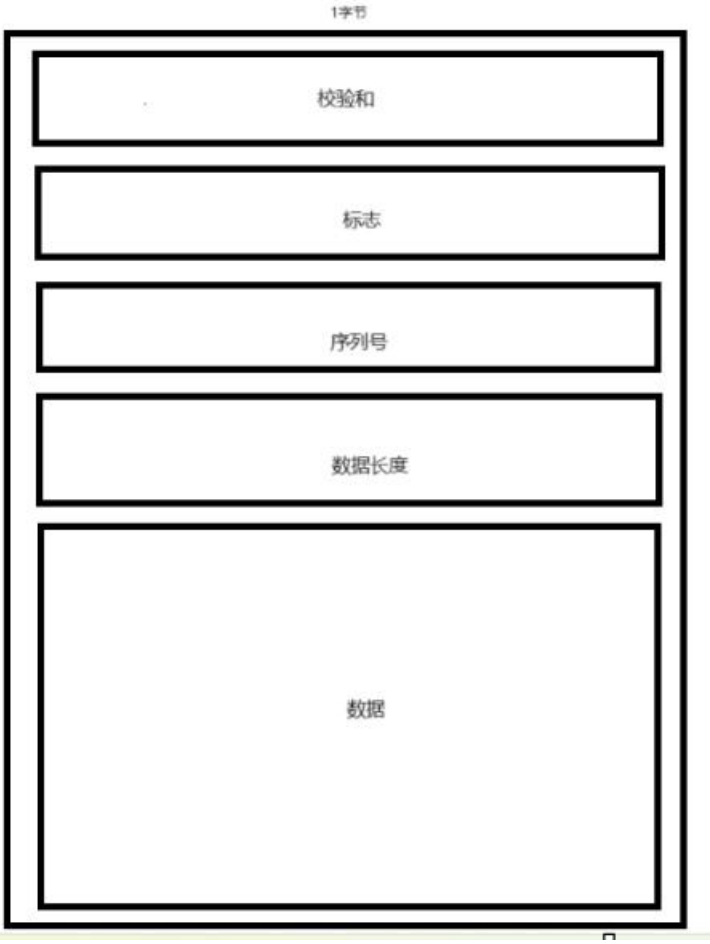
（4）性能测试指标：吞吐率、延时，给出图形结果并进行分析。

# 三、实验内容

## 协议设计：

- 数据报文划分为头部和数据两部分
- 头部包含：校验和、标志位、序列号、数据长度

简易表示如下：




定义报首如下:

```
/*
struct Head {
    unsigned char checknum; //校验和
    unsigned char flag;     //标志位
    uint8_t seq;            //表示数据包的序列号，一个字节为周期
    uint8_t len;            //表示数据包的长度（包含头）
};
*/
```

其中:

**校验和**用于差错检测;

**标志位**的内容在实验 3-1 中有如下定义:

```
const unsigned char SYN = 0x01;
const unsigned char SYN_ACK = 0x02;
const unsigned char FIN = 0x03;
const unsigned char FIN_ACK = 0x04;
const unsigned char ACK = 0x05;
const unsigned char END = 0x06;
const unsigned char NOT_END = 0x07;
```

前四个标志用于握手和挥手阶段，后三者主要用于数据传输阶段；

**序列号**所在的序列空间为 0~99；

**数据包长度**为 MAX_UDP_LEN+4 = 10004 字节。（除了最后一个可能没有达到最大长度的数据包。）

## 差错检测：

发送方发送数据包时会计算报首和数据的校验和，将结果放到报首的校验和字段。

接收方在接收数据后，会对整个数据包计算校验和，此时校验和为 0 则证明传输过程中没有发生数据损坏，完成了差错检测。

计算检验和代码如下：

```
unsigned char check_sum(char* package, int len)
{
 if (len == 0) {
  return ~(0);
 }
 unsigned long sum = 0;
 int i = 0;
 while (len--) {
  sum += (unsigned char)package[i++];
  if (sum & 0xFF00) {
   sum &= 0x00FF;
```

```
    sum++;
   }
  }
  return ~(sum & 0x00FF);
 }
```

实现思路是：遍历数据包中的每个字节，将其加到 sum 中；如果 sum 的高位不为 0，将高位清零，然后将 sum 加 1；最后，返回 sum 的补码取反。

## 三次握手与两次挥手：

通过三次握手与两次挥手实现了建立连接和断开连接的过程。

三次握手与四次挥手在上一次实验中已经进行验证，对原理和过程也有较为详细的说明，这里不再赘述。

在 3-1 中采用两次挥手主要是因为此次实验中不需要担心有未完成的任务，两次挥手即可进行断开。

发送端（Client）的三次握手代码如下：

```
void three_way_handshake()
{
 //三次握手建立连接
 while (true)
 {
  char sendBuf[2];
  sendBuf[1] = SYN;
  sendBuf[0] = check_sum(sendBuf + 1, 1);


  int ret = sendto(m_ClientSocket, sendBuf, 2, 0, (sockaddr*)&m_ServerAddress, sizeof(m_ServerAddress));
  if (ret != SOCKET_ERROR)
   cout << "已发送第一次握手请求 SYN" << endl;
```

```
    else {
     cout << "发送握手请求失败……" << endl;
     return;
    }
    //开始计时
    int begin = clock();


    char recv[2];  //接收缓存区
    int is_timeout = 0;
    //cout << "测试点2" << endl;


    u_long mode = 1;  // 1 表示非阻塞，0 表示阻塞
    ioctlsocket(m_ClientSocket, FIONBIO, &mode);
    while (recvfrom(m_ClientSocket, recv, 2, 0, (sockaddr*)&m_ServerAddress, &len_addr) == SOCKET_ERROR)
    {
     if (clock() - begin > TIMEOUT)
     {
      is_timeout = 1;
      break;
     }
    }


    //没有超时，接收到的数据包检验正确，是二次握手
    if (is_timeout == 0 && check_sum(recv, 2) == 0 && recv[1] == SYN_ACK)
    {
     cout << "接收到服务端二次握手 SYN_ACK" << endl;
     sendBuf[1] = ACK;
     sendBuf[0] = check_sum(sendBuf + 1, 1);
     sendto(m_ClientSocket, sendBuf, 2, 0, (sockaddr*)&m_ServerAddress, sizeof(m_ServerAddress));
     cout << "已发送第三次握手 ACK" << endl;
     break;
    }
   }
   cout << "连接到服务端" << endl;
   return;
  }
```

其余代码拥有相似性，不再赘述。

## 等停机制和确认重传：

**确认重传：**

维护了一个计时器，自发送方发送数据包并开始尝试捕获 ACK 时开启计时，此步骤需要接受有效 ACK，每超时一次重发一个包并重置计时器；

还可维护一个重发计数器，若超过最大重发次数则发送失败。

如下：

```
int begintime = clock();
char recv[3];
int is_send = 0;

//超时重传
//套接字，数据包，长度，0，地址，地址长度
while (recvfrom(m_ClientSocket, recv, 3, 0,
    if (clock() - begintime > TIMEOUT) {
        cout << "超时，进行重传" << endl;
        is_send = 1;
        break;
    }
}
```

**停等机制：**

未能通过差错检测的包将不予回复 ACK;

必须等对方发送一条消息我方才能发一条消息;

## 依据主函数的脉络来讲解剩余的内容：

在了解了上述的机制后，为了不显得杂乱，我们沿着主函数来看。

（以发送方为例）

首先是进行环境初始化、套接字的创建、套接字的绑定，这些步骤与实验一的 TCP 聊天程序是类似的，但此次实验使用 UDP。

如下：

```
nt main() {
    WSADATA  wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        cout << "WSAStartup error:" << GetLastError() << endl;
        return false;
    }

    //创建SOCKET
    m_ClientSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (m_ClientSocket == INVALID_SOCKET)
    {
        closesocket(m_ClientSocket);
        m_ClientSocket = INVALID_SOCKET;
        return false;
    }

    // 远端地址(路由器)
    const char* ip = "127.0.0.1";
    int port = 4003;
    m_ServerAddress.sin_family = AF_INET;                //ipv4
    m_ServerAddress.sin_port = htons(port);              //转换为网络字节
    inet_pton(AF_INET, ip, &m_ServerAddress.sin_addr);   // 将字符串格式的

    // 绑定本地地址
    sockaddr_in localAddress;
    localAddress.sin_family = AF_INET;
    localAddress.sin_port = htons(4001);
    inet_pton(AF_INET, "127.0.0.1", &localAddress.sin_addr);

    // 绑定套接字到指定的本地地址
    if (bind(m_ClientSocket, (sockaddr*)&localAddress, sizeof(localAddres
        // 处理绑定失败的情况
```

然后进行此次实验的三大块：先进行三次握手，建立连接后发送数据，在发送数据后，进行两次挥手。

```
//三次握手
three_way_handshake();

//发送数据
send_file();

//两次挥手
double_wave();
```

发送数据（发送文件）的代码如下：

```
void send_file()
{
 // 发送文件
 int len = 0;
 char* buffer = new char[100000000];
 string filename;
 cout << "请输入发送的文件名" << endl;
 while (true)
 {
  cin >> filename;
  ifstream fin(filename.c_str(), ifstream::binary);//打开输入的文件，以二进制模式读取
  if (!fin) {
   cout << "文件名有误,请重新输入" << endl;
   continue;
  }
  unsigned char t = fin.get();//按字节读取数据
  while (fin) {
   buffer[len++] = t;
   t = fin.get();
  }
  fin.close();
  break;
 }
 int time_begin = clock();
 send_buffer((char*)(filename.c_str()), filename.length()); //文件名
 send_buffer(buffer, len);         //文件内容
 int time_end = clock();
 cout << filename << "文件传输完毕" << endl;
 int time = (time_end - time_begin) / CLOCKS_PER_SEC;
 cout << "传输时间：" << time << endl;
 if (time != 0)
 {
  double kbps = ((len * 8.0) / 1000) / time;
  cout << "吞吐量：" << kbps << " kbps" << endl;
 }
 cout << "丢包率：
" << (SUCCESS_SEND - SUCCESS_RECV) * 100 / SUCCESS_SEND << "%" << endl;
 cout << endl << endl;
}
```

首先是根据输入的文件名打开文件，按字节读取，即可以一次性

读取所有字节避免反复打开文件（但文件过大时可能缓存区溢出），也可以按数据包的字节大小多次读取。我采用的是一次性读取。

通过 **send_buff()**发送完数据后，会输出传输时间、吞吐量、丢包率这些信息。

send_buff()主要实现：将文件数据分为一个一个数据包并赋予序列号然后发通过 send_package()发送，且对于最后一个数据包有赋予 END 标志，否则 NOT_END。

而关键的 **send_package()**代码如下：

```
1.    bool send_package(char* start, int length, uint8_t req, int is_END = 0) {
2.     char* package;
3.     int Len = length + 4;      //数据包长度（含头 4 字节）
4.     package = new char[length + 4];   //数据包
5.
6.     //组装数据包
7.     if (is_END) {
8.      package[1] = END;         //标志位
9.     }
10.    else
11.    {
12.     package[1] = NOT_END;
13.    }
14.    package[2] = req;       //序号
15.    package[3] = length;      //数据长度（不含头）
16.    for (int i = 0; i < length; i++) {// 数据
17.     package[i + 4] = start[i];
18.    }
19.    package[0] = check_sum(package + 1, Len-1);//校验和
20.
21.    while (1) {
22.     //套接字，数据包，长度，0，目的地址，地址大小
23.     sendto(m_ClientSocket, package, Len, 0, (sockaddr*)&m_ServerAddress, sizeof(m_Server
    Address));
24.     SUCCESS_SEND++;
25.
```

```
    int begintime = clock();
    char recv[3];
    int is_send = 0;

    //超时重传
    //套接字，数据包，长度，0，地址，地址长度
    while (recvfrom(m_ClientSocket, recv, 3, 0, (sockaddr*)&m_ServerAddress, &len_addr) == SOCKET_ERROR) {
     if (clock() - begintime > TIMEOUT) {
      cout << "超时，进行重传" << endl;
      is_send = 1;
      break;
     }
    }
    if (is_send == 0 && check_sum(recv, 3) == 0 && recv[1] == ACK && recv[2] == (uint8_t)req)
    {
     cout << "接收到 ACK:  " << int(recv[2])<< endl;
     SUCCESS_RECV++;
     return true;
    }
   }
  }
```

可见，send_package()依次实现了：组装数据包并计算校验和，发送数据包，等待接收正确的 ACK;

只有在接收到正确的 ACK 后，才会发送下一个数据包;

如果计时器超时则重新发送数据包。

这些步骤保证了 UDP 的可靠传输。

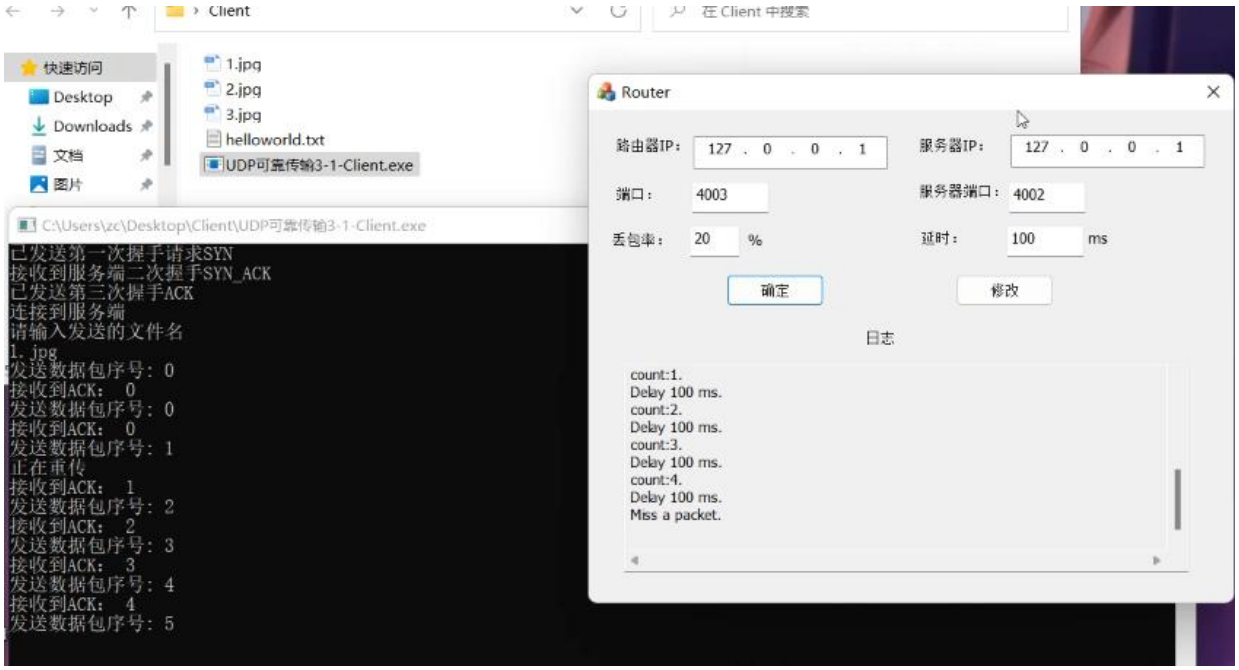
主函数在进行两次挥手后，则进行套接字关闭、清理资源。

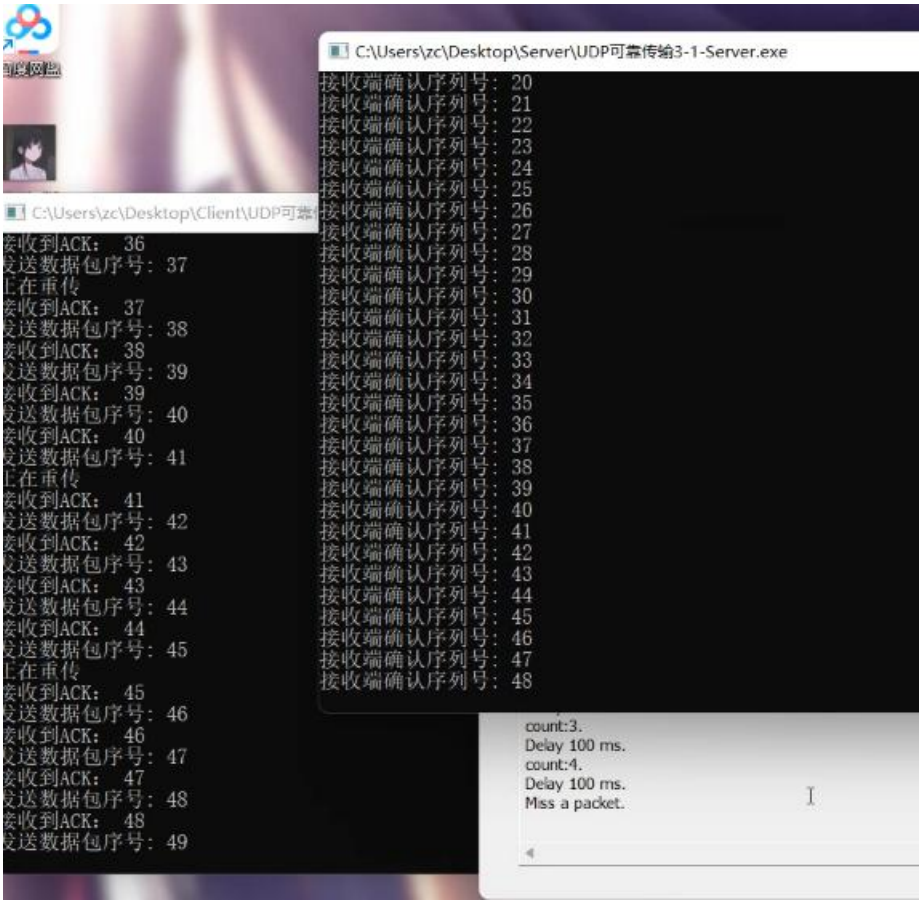
如下：

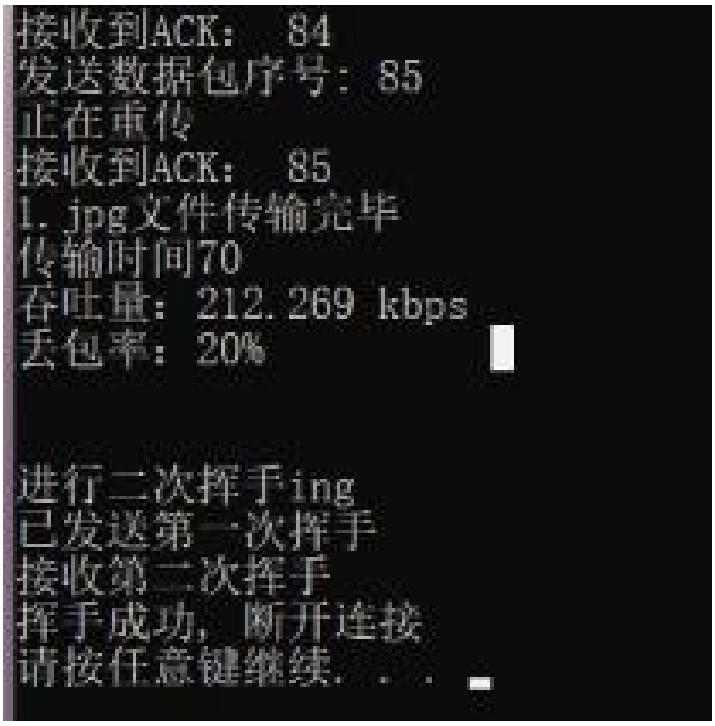
```
    //关闭套接字
    closesocket(m_ClientSocket);
    //清除资源
    WSACleanup();

    system("pause");
    return 0;
}
```

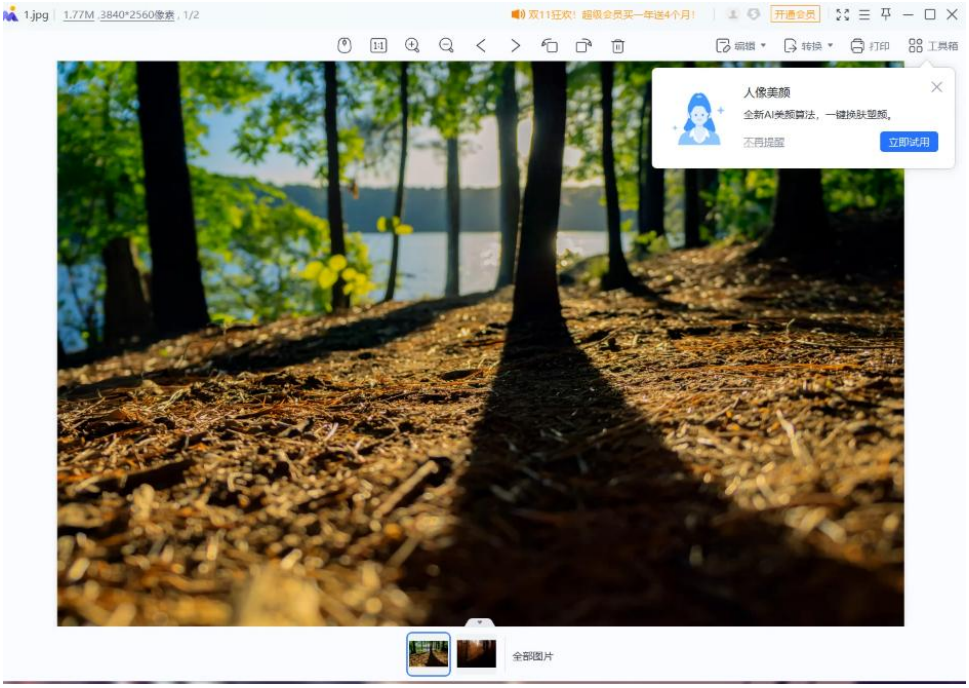
实验效果如下：




C:\Users\zc\Desktop\Server\UDP可靠传输3-1-Server.exe
接收端确认序列号: 20
接收端确认序列号: 21
接收端确认序列号: 22
接收端确认序列号: 23
接收端确认序列号: 24
接收端确认序列号: 25
接收端确认序列号: 26
接收端确认序列号: 27
接收端确认序列号: 28
接收端确认序列号: 29
接收端确认序列号: 30
接收端确认序列号: 31
接收端确认序列号: 32
接收端确认序列号: 33
接收端确认序列号: 34
接收端确认序列号: 35
接收端确认序列号: 36
接收端确认序列号: 37
接收端确认序列号: 38
接收端确认序列号: 39
接收端确认序列号: 40
接收端确认序列号: 41
接收端确认序列号: 42
接收端确认序列号: 43
接收端确认序列号: 44
接收端确认序列号: 45
接收端确认序列号: 46
接收端确认序列号: 47
接收端确认序列号: 48
C:\Users\zc\Desktop\Client\UDP可靠
接收到ACK: 36
发送数据包序号: 37
正在重传
接收到ACK: 37
发送数据包序号: 38
接收到ACK: 38
发送数据包序号: 39
接收到ACK: 39
发送数据包序号: 40
接收到ACK: 40
发送数据包序号: 41
正在重传
接收到ACK: 41
发送数据包序号: 42
接收到ACK: 42
发送数据包序号: 43
接收到ACK: 43
发送数据包序号: 44
接收到ACK: 44
发送数据包序号: 45
正在重传
接收到ACK: 45
发送数据包序号: 46
接收到ACK: 46
发送数据包序号: 47
接收到ACK: 47
发送数据包序号: 48
接收到ACK: 48
发送数据包序号: 49
count:3.
Delay 100 ms.
count:4.
Delay 100 ms.
Miss a packet.


接收到ACK: 84
发送数据包序号: 85
正在重传
接收到ACK: 85
1.jpg文件传输完毕
传输时间70
吞吐量: 212.269 kbps
丢包率: 20%
进行二次挥手ing
已发送第一次挥手
接收第二次挥手
挥手成功, 断开连接
请按任意键继续. . .

文件也确实成功发送了:




## 四、思考与总结

在检查时，经过学长指点，发现代码中确实有一个 bug:

1. 发送方序列号为 X 的发送数据包;
2. 接收方接收，发送 ACK_X，下一个有效的序列号为 X+1;
3. 接收方的 ACK_X 丢失;
4. 发送方等待超时，重新发送序列号为 X 的数据包;
5. 接收方收到 X,丢弃;
6. 从此陷入超时-发送-丢弃的循环。

解决的办法也很简单，接收方接收到非有效的数据包时，再次发送上一个有效数据包的 ACK 即可，避免 ACK 丢失出现问题。

这次实验让我更深刻体会到了可靠传输的建立，亲手编程验证了

书上的理论。