南開大學
Nankai University

计算机学院

计算机网络实验报告

基于 UDP 服务设计可靠传输协议

姓名：杨冰雪

学号：2110508

专业：计算机科学与技术

2023 年 11 月 17 日

目录

1 实验要求

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。

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。完成给定测试文件的传输，显示传输时间和平均吞吐率。
3. 性能测试指标：吞吐率、延时，给出图形结果并进行分析。

2 协议设计

2.1 报文格式

报文格式

```
struct message
{
    bool SYN = false;
    bool FIN = false;
    bool START = false; //标志开始传输文件
    bool END = false; //标志文件传输完成
    bool ACK = false;
    u_short seq;//序列号
    u_short ack;//确认号
    u_long len;//数据长度
    u_long num;
    u_short checksum;//校验和
    char data[datasize];//数据长度
};
```

- 设计了一个结构体类型的报文来包含我们所需要的基本信息，包括标志位、序列号、确认号、数据长度、数据包数量、检验和和要发送的数据。
- 标志位设计了 SYN、FIN、START、END、ACK。其中 START 用来标志开始传输文件，END 标志文件传输完成。
- data 为要传输的图片和文字部分，其长度设置的是 1024 个字节。

2.2 建立连接

建立连接就是仿造三次握手的过程。

第一次握手 TCP 客户端向服务端发送带有【SYN】标识的信息，同时随机初始化一个序列号 seq=x。

第二次握手 服务器收到客户端发来的消息后，会检查消息的 SYN 标识是否为 true，如果是则确认连接，向客户端发出带有【SYN，ACK】标识的确认消息。确认号 ack=x+1，同时也会随机初始化一个序列号 seq=y。

第三次握手 客户端收到确认消息后，也会同样检查消息的 SYN 和 ACK 标识是否为 true，并且确认号是否等于上一个发送消息的序列号加一，如果是则向服务器发出带有【ACK】标识的消息。确认号 ack=y+1，序列号 seq=x+1。此时客户端的连接建立，但服务器端还要检查收到的消息的 ACK 和确认号，如果满足要求则服务端的连接建立。

其具体实现代码如下：

客户端三次握手

```
void sendconnect()
{
    //第一次握手
    int iMode = 1;
    ioctlsocket(Client, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置
    cout << "【消息】开始连接！发送第一次握手！" << endl;
    message recvMsg, sendMsg;
    sendMsg.SYN=true; //【SYN】
    sendMsg.seq = getrand(); //随机产生一个序列号
    sendMsg.checksum=0;//将校验和置为0
    sendMsg.checksum= cksum((u_short*)&sendMsg, sizeof(sendMsg));//计算校验和
    if (sendto(Client, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&serveraddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
        cout << "【ERROR】send error:" <<WSAGetLastError<< endl;
    }
    //第二次握手
    clock_t start = clock();
    clock_t end;
    while (true)
    {
        message msg;
        if (recvfrom(Client, (char*)&msg, BUFFER, 0, (SOCKADDR*)&serveraddr, &len) ==
            -1 || cksum((u_short*)&msg, sizeof(msg)) != 0) {
            recvMsg = message();
        }
        else {
            recvMsg=msg;
        }
        //超时重传
        end = clock();
        if ((end - start)> MAXTIME) {
            cout << "【ERROR】连接超时,请保持网络通畅和服务端正常运行！" << endl;
            if (sendto(Client, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&serveraddr,
                sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
                cout << "【ERROR】send error:" << WSAGetLastError << endl;
            }
            start = clock();
            continue;
        }
        //接收确认
        if (recvMsg.ACK&& recvMsg.SYN && recvMsg.ack == sendMsg.seq + 1) {
```

```
            cout << "【消息】接收到第二次握手!" << endl;
            break;
        }

    }
    //第三次握手
    sendMsg.ACK = true;
    sendMsg.seq = recvMsg.ack; //序列号等于收到的确认号
    sendMsg.ack = recvMsg.seq + 1; //确认号等于收到的序列号+1
    cout << "【消息】发送第三次握手! " << endl;
    sendMsg.checksum = 0;//将校验和置为0
    sendMsg.checksum = cksum((u_short*)&sendMsg, sizeof(sendMsg));//计算校验和
    if (sendto(Client, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&serveraddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
        cout << "【ERROR】send error:" << WSAGetLastError << endl;
    }
}
```

服务端三次握手

```
void recvconnect()
{
    //第一次握手
    int iMode = 1; //非阻塞
    ioctlsocket(Server, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置
    cout << "【消息】服务器等待连接! " << endl;
    message recvMsg, sendMsg;
    while (true)
    {
        message msg;
        if (recvfrom(Server, (char*)&msg, BUFFER, 0, (SOCKADDR*)&clientaddr, &len) ==
            -1 || cksum((u_short*)&msg, sizeof(msg)) != 0) {
            recvMsg= message();
        }
        else {
            recvMsg = msg;
        }
        if (recvMsg.SYN==true)
        {
            cout << "【消息】收到第一次握手! " << endl;
            break;
        }
    }
    //第二次握手
    sendMsg.SYN=true; // 【SYN, ACK】
    sendMsg.ACK=true;
    sendMsg.ack = recvMsg.seq + 1;    // 确认号等于收到的序列号+1
    sendMsg.seq = getrand();  //随机获取序列号
    sendMsg.checksum = 0;//将校验和置为0
```

```
    sendMsg.checksum = cksum((u_short*)&sendMsg, sizeof(sendMsg));//计算校验和
    if (sendto(Server, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&clientaddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
        cout << "【ERROR】send error:" << WSAGetLastError << endl;
    }
    else {
        cout << "【消息】发送第二次握手! " << endl;
    }
    //第三次握手
    clock_t start=clock();
    clock_t end;
    while (true) {
        message msg;
        if (recvfrom(Server, (char*)&msg, BUFFER, 0, (SOCKADDR*)&clientaddr, &len) ==
            -1 || cksum((u_short*)&msg, sizeof(msg)) != 0) {
            recvMsg= message();
        }
        else{
            recvMsg = msg;
        }
        end = clock();
        if ((end - start) > MAXTIME) {
            cout << "【ERROR】等待时间太长, 取消连接! " << endl;
            return recvconnect();
        }
        if (recvMsg.ACK==true && recvMsg.ack == sendMsg.seq + 1) {
            break;
        }
    }
    cout << "【消息】接收到第三次握手, 连接成功! " << endl;
}
```

2.3 断开连接

断开连接采用的是四次挥手，客户端首先向服务端发出第一次挥手，服务端收到挥手后向客户端发送第二次挥手，然后服务端向客户端发送第三次挥手，客户端收到后向服务端返回第四次挥手，然后断开连接，程序运行结束。

客户端四次挥手

```
void closeconnect() {  // 断开连接
    //第一次挥手
    message recvMsg, sendMsg1;
    sendMsg1.FIN = true; //【FIN】
    sendMsg1.seq = 65533;//序列号
    sendMsg1.checksum = 0;//将校验和置为0
    sendMsg1.checksum = cksum((u_short*)&sendMsg1, sizeof(sendMsg1));//计算校验和
    if (sendto(Client, (char*)&sendMsg1, BUFFER, 0, (SOCKADDR*)&serveraddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
```

```
        cout << "【ERROR】send error:" << WSAGetLastError << endl;
    }
    else {
        cout << "【消息】断开连接! 发送第一次挥手! " << endl;
    }
    //第二次挥手
    clock_t start = clock();
    clock_t end;
    while (true) {
        end = clock();
        if ((start - end) > disconnecttime) {
            cout << "【ERROR】等待时间太长, 退出连接" << endl;
            return closeconnect();
        }
        message msg;
        if (recvfrom(Client, (char*)&msg, BUFFER, 0, (SOCKADDR*)&serveraddr, &len) ==
            -1 || cksum((u_short*)&msg, sizeof(msg)) != 0) {
            recvMsg = message();
        }
        else {
            recvMsg = msg;
        }
        if (recvMsg.ACK == true && recvMsg.ack == sendMsg1.seq + 1) {
            cout << "【消息】客户端收到第二次挥手" << endl;
            break;
        }
    }
    //第三次挥手
    start = clock();
    while (true) {
        end = clock();
        if ((start - end) > disconnecttime) {
            cout << "【ERROR】等待时间太长, 退出连接" << endl;
            return closeconnect();
        }
        message msg;
        if (recvfrom(Client, (char*)&msg, BUFFER, 0, (SOCKADDR*)&serveraddr, &len) ==
            -1 || cksum((u_short*)&msg, sizeof(msg)) != 0) {
            recvMsg = message();
        }
        else {
            recvMsg = msg;
        }
        if (recvMsg.FIN==true) {
            cout << "【消息】客户端收到第三次挥手" << endl;
            break;
        }
    }
    //第四次挥手
```

```
    message sendMsg2;
    sendMsg2.ACK = true; // 【ACK】
    sendMsg2.ack = recvMsg.seq + 1;//确认号等于序列号+1
    sendMsg2.checksum = 0;//将校验和置为0
    sendMsg2.checksum = cksum((u_short*)&sendMsg2, sizeof(sendMsg2));//计算校验和
    if (sendto(Client, (char*)&sendMsg2, BUFFER, 0, (SOCKADDR*)&serveraddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
        cout << "【ERROR】send error:" << WSAGetLastError << endl;
    }
    else {
        cout << "【消息】断开连接成功! " << endl;
    }
}
```

服务端四次挥手

```
void closeconnect(message msg) {
    //第二次挥手
    message sendMsg1;
    sendMsg1.ACK=true; // 【ACK】
    sendMsg1.ack = msg.seq + 1;//确认号等于序列号+1
    sendMsg1.checksum = 0;//将校验和置为0
    sendMsg1.checksum = cksum((u_short*)&sendMsg1, sizeof(sendMsg1));//计算校验和
    if (sendto(Server, (char*)&sendMsg1, BUFFER, 0, (SOCKADDR*)&clientaddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
        cout << "【ERROR】send error:" << WSAGetLastError << endl;
    }
    else {
        cout << "【消息】发送第二次挥手! " << endl;
    }
    //第三次挥手
    message sendMsg2;
    sendMsg2.FIN = true;// 【FIN】
    sendMsg2.seq = 65534;//序列号
    sendMsg2.checksum = 0;//将校验和置为0
    sendMsg2.checksum = cksum((u_short*)&sendMsg2, sizeof(sendMsg2));//计算校验和
    if (sendto(Server, (char*)&sendMsg2, BUFFER, 0, (SOCKADDR*)&clientaddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
        cout << "【ERROR】send error:" << WSAGetLastError << endl;
    }
    else {
        cout << "【消息】发送第三次挥手! " << endl;
    }
    //第四次挥手
    message recvMsg;
    clock_t start = clock();
    clock_t end;
    while (true) {
        end = clock();
```

```
        if ((end - start) > MAXTIME) {
            end = clock();
            if ((start - end) > disconnecttime) {
                cout << "【ERROR】等待时间太长，退出连接" << endl;
                recFileName();
                return;
            }
        }
        message msg;
        if (recvfrom(Server, (char*)&msg, BUFFER, 0, (SOCKADDR*)&clientaddr, &len) ==
            -1 || cksum((u_short*)&msg, sizeof(msg)) != 0) {
            recvMsg= message();
        }
        else {
            recvMsg = msg;
        }
        if (recvMsg.ACK == true && recvMsg.ack == sendMsg2.seq + 1) {
            cout << "【消息】断开连接成功！" << endl;
            break;
        }
    }
}
```

2.4 差错检测

收到消息后，我们需要进行差错检测。首先，需要检查消息类型是否正确。其次，需要检查消息的序列号是否正确。最后，需要检查校验和是否正确。

校验和的计算方法如下：

- 将消息头的校验和设置为 0
- 将消息头和数据看成 16 位整数序列，不足 16 位的最后补 0
- 每 16 位相加，溢出的部分加到最低位上
- 最后的结果取反

当接收端接收到数据时，需要用同样的方法计算校验和，如果校验和结果全为 0，说明消息正确，否则，说明消息损坏。实现校验和的代码如下：

校验和

```
u_short cksum(u_short* mes, int size) {
    int count = (size + 1) / 2;
    u_short* buf = (u_short*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, mes, size);
    u_long sum = 0;
    while (count--) {
        sum += *buf++;
```

```
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

实现检验消息类型和序列号的代码如下：

检验消息类型和序列号

```
if (recvMsg.ACK==true && recvMsg.ack == seq) {
    cout << "收到服务器的确认数据包! " << endl;
    cout << "checksum=" << recvMsg.checksum << ", len=" << recvMsg.len << endl;
    cout << endl;
    return true;
}
```

2.5 确认重传

确认重传包括差错重传和超时重传。因为是类似于 rdt3.0 的设计，所以当遇到接收到的消息的校验和检验出错，或者接受到消息时出错和超过规定时间内没有接收到消息时，都会进行重传。

每次当接收到的消息出现检验和出错和接受错误时，都会将接受消息进行初始化，等待其重新接受，一旦等待超时，进行重传。这里重传的次数也进行了限制，最大重传次数设的 10 次，一旦超过 10 次，就退出循环，不再进行发送，这样能防止陷入死循环中。其代码如下：

确认重传

```
while (true) {
    message msg;
    if (recvfrom(Client, (char*)&msg, BUFFER, 0, (SOCKADDR*)&serveraddr, &len) == -1
        || cksum((u_short*)&msg, sizeof(msg)) != 0) {
        recvMsg = message();
    }
    else {
        recvMsg = msg;
    }
    end = clock();
    if ((end - start )> MAXTIME) { //超时重传
        cout << "应答超时，重新发送数据包" << endl;
        if (sendto(Client, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&serveraddr,
            sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
            cout << "【ERROR】send failed:" << WSAGetLastError << endl;
        }
        count++;
        cout << "尝试重新发送第" << count << "次数据包" << endl;
        if (count >= 10) {
            break;
```

```
        }
        start = clock();
    }
    if (recvMsg.ACK==true && recvMsg.ack == seq) {
        cout << "收到服务器的确认数据包! " << endl;
        cout << "checksum=" << recvMsg.checksum << ", len=" << recvMsg.len << endl;
        cout << endl;
        return true;
    }
}
```

2.6 停等机制

该机制下客户端每发送一条消息后，都要等待服务端确认后发送 ACK 消息，当客户端接收到返回的 ACK 后才能发送下一条。

2.6.1 客户端发送消息

停等机制-客户端发送消息

```
message recvMsg;
sendMsg.seq = seq;
sendMsg.checksum = 0;//将校验和置为0
sendMsg.checksum = cksum((u_short*)&sendMsg, sizeof(sendMsg));//计算校验和
if (sendto(Client, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&serveraddr,
    sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
    cout << "【ERROR】send error:" << WSAGetLastError << endl;
}
cout << "checksum=" << sendMsg.checksum << ", len=" << sendMsg.len << endl;
```

2.6.2 服务端发送 ACK 消息

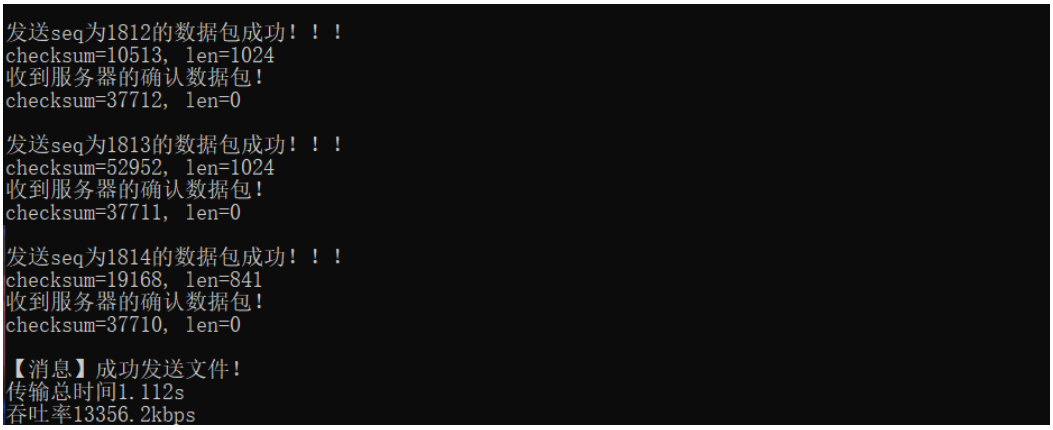
停等机制-服务端发送确认消息

```
if (recvMsg.seq == seq) {
    cout << "收到seq为" << recvMsg.seq << "的数据包" << endl;
    cout << "checksum=" << recvMsg.checksum << ", len=" << recvMsg.len << endl;
    sendMsg.ACK = true;
    sendMsg.ack = recvMsg.seq;
    sendMsg.checksum = 0;//将校验和置为0
    sendMsg.checksum = cksum((u_short*)&sendMsg, sizeof(sendMsg));//计算校验和
    if (sendto(Server, (char*)&sendMsg, BUFFER, 0, (SOCKADDR*)&clientaddr,
        sizeof(SOCKADDR)) == (SOCKET_ERROR)) {
        cout << "【ERROR】send error:" << WSAGetLastError << endl;
    }
    else {
        cout << "向服务端发送确认数据包!" << endl;
```

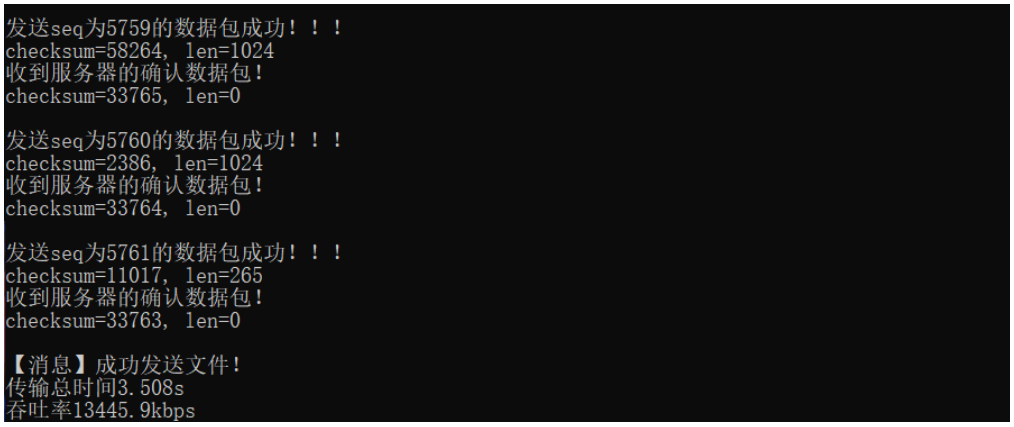
```
13      }
14      cout << endl;
15      fout.write(recvMsg.data, recvMsg.len);
16      break;
17  }
```

3 实验结果

当运行程序，进行文件传输时，第一张照片传输结果如下：



第二张照片传输结果如下：



第三张照片传输结果如下：

```
发送seq为11687的数据包成功！！！
checksum=34798, len=1024
收到服务器的确认数据包！
checksum=27837, len=0

发送seq为11688的数据包成功！！！
checksum=37855, len=1024
收到服务器的确认数据包！
checksum=27836, len=0

发送seq为11689的数据包成功！！！
checksum=3646, len=482
收到服务器的确认数据包！
checksum=27835, len=0

【消息】成功发送文件！
传输总时间7.153s
吞吐率13379.5kbps
```

helloworld.txt 文件传输结果如下：

```
发送seq为1615的数据包成功！！！
checksum=30715, len=1024
收到服务器的确认数据包！
checksum=37909, len=0

发送seq为1616的数据包成功！！！
checksum=30714, len=1024
收到服务器的确认数据包！
checksum=37908, len=0

发送seq为1617的数据包成功！！！
checksum=30457, len=1024
收到服务器的确认数据包！
checksum=37907, len=0

【消息】成功发送文件！
传输总时间0.965s
吞吐率13719.4kbps
```

最后在文件夹下找到了传输的文件，可以看出传输的文件均正确，其接收文件的大小和传输文件的大小是一样的，打开后也是相同的，可以看出传输成功！

名称	修改日期	类型	大小
x64	2023/11/16 0:38	文件夹	
1.jpg	2023/11/17 15:26	JPG 文件	1,814 KB
2.jpg	2023/11/17 15:26	JPG 文件	5,761 KB
3.jpg	2023/11/17 15:27	JPG 文件	11,689 KB
helloworld.txt	2023/11/17 15:27	文本文档	1,617 KB
recv.vcxproj	2023/11/15 18:05	VC++ Project	7 KB
recv.vcxproj.filters	2023/11/15 18:05	VC++ Project Fil...	2 KB
recv.vcxproj.user	2023/11/13 22:36	Per-User Project...	1 KB
server.cpp	2023/11/16 22:02	C++ Source	9 KB

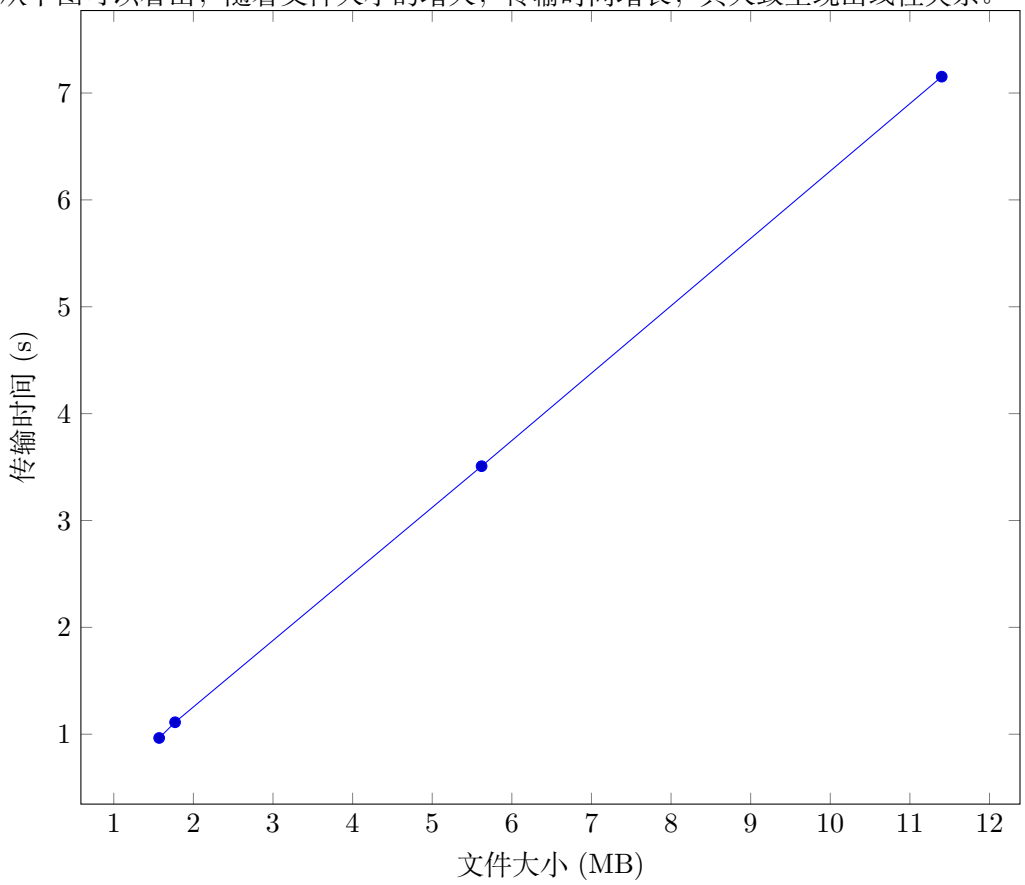
同时当我们运行 router.exe 程序进行丢包设置后，可以看出当超时后进行了重传。

```
发送seq为87的数据包成功！！！
checksum=34752, len=1024
应答超时，重新发送数据包
尝试重新发送第1次数据包
收到服务器的确认数据包！
checksum=39436, len=0

发送seq为88的数据包成功！！！
checksum=56757, len=1024
收到服务器的确认数据包！
checksum=39435, len=0

发送seq为89的数据包成功！！！
checksum=58034, len=1024
应答超时，重新发送数据包
尝试重新发送第1次数据包
收到服务器的确认数据包！
checksum=39434, len=0
```

3.1 性能测试

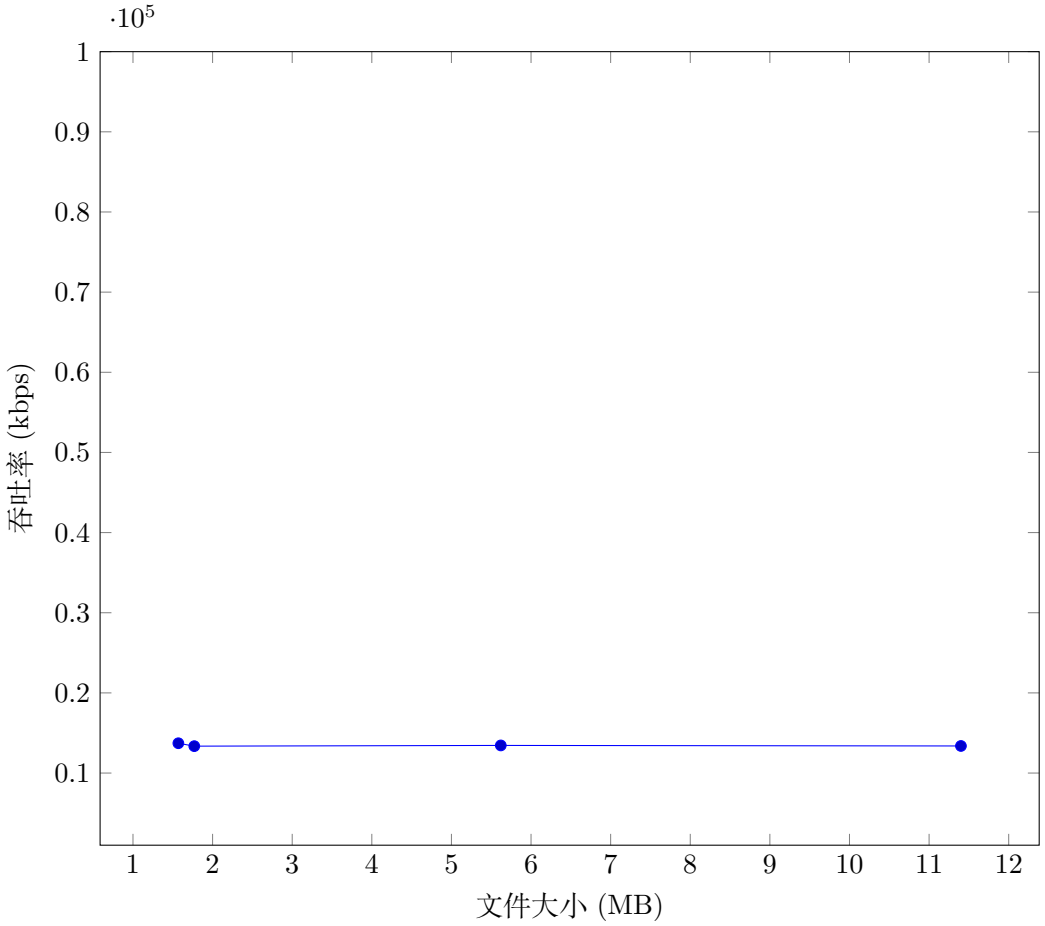
3.1.1 不同文件大小的传输时间图

从下图可以看出，随着文件大小的增大，传输时间增长，其大致呈现出线性关系。



3.1.2 不同文件大小的吞吐率图

从下图可以看出，随着文件大小的增大，其吞吐率基本上不变。



4 实验总结

通过本次实验，我学会了对 UDP 实现可靠文件传输。让我对许多建立可靠连接的机制，如三次握手、差错检验、超时重传等有了更深刻的理解。但同时我在编写代码的时候也遇到了一些问题，比如协议设计好后，当我进行三次握手时，因为最开始在 main 函数里面定义的变量，当定义的函数想要使用时，还需要通过引用的方式进行传入，不仅增加了代码量，还容易出错，所以我定义为了全局的变量。以及最开始在编写接收到消息但消息检验码错误时，如何又进行重传这些最开始有点困难。