

# 2021 春季计算机网络期中项目

## 1. 实验目的

请用 UDP Socket, 构建一个简单的文件传输协议, 实现可靠文件传送。具体要求:

- 1、 必须用 UDP 传输层协议;
- 2、 语言不限 (可以用 C, C++, Java, Python);
- 3、 要求有较好的性能;
- 4、 代码包括服务器端和客户端代码;
- 5、 测试请传输一个 1MB 和 1 个 10MB 左右的文件, 展示程序运行结果;
- 6、 选一个方法验证文件是一致的。

## 2. 设计思路

首先我们知道, UDP 相较于 TCP 是不可靠的。因为 UDP 不像 TCP 那样具有超时重传, 滑动窗口机制和拥塞控制等用于确保数据包按顺序完整交付的机制, 也没有在客户端和服务端之间建立连接来确保传输安全可靠。所以想要实现 UDP 下的可靠文件传输, 就必须仿照 TCP 来给 UDP 加上一些机制来提高文件传输的可靠性 (当然这显然会牺牲掉一部分 UDP 的传输效率)。

我在重新学习了课本之后, 总结了如下可以给 UDP 加上的机制:

- ①超时重传
- ②停等协议
- ③滑动窗口协议
- ④拥塞控制

在参考了网上的代码以及对时间进行了考量之后, 我选择了对 UDP 添加超时重传和停等协议这两项功能来提高 UDP 传输文件的可靠性。

### 3. 代码分析

客户端以及服务器端的全部代码如下所示（所有分析均已放在注释中，便于助教查看和上下对比）：

①客户端：

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<winsock2.h>
#include<windows.h>
#include<string.h>

#pragma comment(lib,"ws2_32.lib");//把 ws2_32.lib 这个库加入到工程文件中

#define packet_size 1024//规定了 UDP 每次传递的数据包的大小

#define GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}
//最后一行将 us 转换为 s，统一单位
//用于计算程序运行时间，也即文件传输时间

//该函数用于计算文件大小
long long file_Size(char* filename){
    FILE* fp=fopen(filename,"rb");//rb: 以二进制形式读取文件
    //若采取 r 的话会导致 txt 以外形式的文件读取出现错误和误差
    if(!fp){//若打开文件失败
        return -1;//设置文件大小为-1
    }
    fseek(fp,0,SEEK_END);//将文件指针定位到文件末尾
    long long size=ftell(fp);//ftell 用于得到文件位置指针当前位置相对于文件首的偏移字节数
    //这里相当于得到文件首和文件末尾之间的字节数，即文件大小
    fclose(fp);//关闭文件
    return size;//返回文件大小
}

int main()
{
    WSADATA data;//存放 windows socket 初始化信息,初始化网络环境
```

```

int state=WSAStartup(MAKEWORD(2,2),&data);//使用 2.2 版本的 Socket
if(state!=0){
    printf("初始化失败!\n\n");
    return 0;
    //若初始化失败则打印信息并终止程序
}

SOCKET sock=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);//生成套接字, sock
用于发送文件切割后的每个数据包
//AF_INET:用于 socket 创建通信连接的类型, 这里就是 ipv4 地址类型的通信连接
可用
//SOCK_DGRAM:传输形式为数据包形式
//设置协议为 UDP 协议
if(sock==INVALID_SOCKET){
    printf("套接字生成失败!\n\n");
    return 0;
    //生成套接字失败则打印信息并终止程序
}
SOCKET sock_end=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);//sock_end 用
于发送含有终止信息的数据包
if(sock_end==INVALID_SOCKET){
    printf("套接字生成失败!\n\n");
    return 0;
    //生成套接字失败则打印信息并终止程序
}
//struct sockaddr_in:用来处理网络通信的地址, 并把 port 和 addr 分开储存在
两个变量中
//用来处理网络通信的地址
struct sockaddr_in client_send;
struct sockaddr_in client_receive;
client_send.sin_family=AF_INET;
client_send.sin_port=htons(10000);//设置端口为 10000 (服务器上设置端口
10000 开放)
client_send.sin_addr.s_addr=inet_addr("49.232.4.77");//服务器 IP 地址
//client_send.sin_addr.s_addr=inet_addr("172.19.13.34");//服务器 IP
地址

struct timeval time_limit;//用于判定超时
time_limit.tv_sec=1;
time_limit.tv_usec=200;
//这里可修改超时的判定时间标准

fd_set rfd;//主要用于 select 函数 (select 函数的用处会在后面提及)

```

```

//是一组文件描述字(fd)的集合

char file_name[100];//存储文件名
printf("请输入需要传输的文件名称: \n");
scanf("%s",file_name);
long long file_size=file_Size(file_name);//得到文件大小
long long send_packet_size=0;
char packet[packet_size];//用于存储每个被切割后的数据包内容
char id[4];//该数组内容会被复制到 packet 的前 4 位, 作为每个数据包的序号。
//它是保证数据包按顺序发送的唯一标准
long long cnt=0;//用于计算重新发包的次数
long long total=0;//计算总发包次数
long long ack=0;//服务器端返回的确认信息 ack
long long packet_id=0;//包的序号
long long time_limit_result;//保存是否超时的判定结果
long long current_id;//保存当前数据包的序号

FILE* fp=fopen(file_name,"rb");//打开文件, 并以二进制形式读取
double start,end;//用于存储文件传输的开始时间和结束时间

GET_TIME(start);//获取文件开始传输的时间
while(file_size>0){//只要大于 0 就说明文件还没传输完毕
    memcpy(packet,&ack,4);//将 ack 的值(int 型)以地址形式存储到 packet
    数组的前 4 位中
    //相当于在每个数据包的头部加上一个序号, 只不过该序号不是 int 型, 是 4
    位地址型(char)
    if(file_size>packet_size-4){//除了前四位, 每个数据包的剩下部分均用
    于存储信息
        send_packet_size=packet_size;//超出则强行切割至 packet_size-
        4, 剩余内容等待下次循环被继续切割
    }
    else{
        send_packet_size=file_size;//小于则直接全部存入数组即可
    }
    fread(packet+4,send_packet_size,1,fp);//从 packet+4 的位置开始, 按
    照 send_packet_size 的大小
    //读出 fp 指向的文件内容并存入 packet 数组
    long long send_code=sendto(sock,packet,sizeof(packet),0,(SOCKAD
    DR *)&client_send,sizeof(client_send));
    //发送 packet 数组(设置缓冲区为 sizeof(packet), 这里为 1024)
    if(send_code==SOCKET_ERROR){
        printf("第%d 号包发送失败!\n\n",ack);
        return 0;
    }
    //发送失败则打印信息并终止程序

```

```

    }
    total++; //发送文件总次数+1
    FD_ZERO(&rfd); //将指定的文件描述符集清空
    while(true) { //该 while 循环涉及到添加的功能：超时重传和停等协议
        long long retransmission_code;
        FD_SET(sock, &rfd); //用于在文件描述符集合中增加一个新的文件描述符

        time_limit_result = select(0, &rfd, NULL, NULL, &time_limit);
        //用于判断在 time_limit 时间内是否有收到文件这一事件发生
        if(time_limit_result == 0) { //若为 0 则代表没有事件发生，即超时
            printf("当前发送的第%d 号包未收到 ACK 确认回复，已超时，即将重传.....\n", ack);
            FD_ZERO(&rfd); //将指定的文件描述符集清空
            retransmission_code = sendto(sock, packet, sizeof(packet), 0, (SOCKADDR *)&client_send, sizeof(client_send));
            //重新发送当前数据包
            if(retransmission_code == SOCKET_ERROR) {
                printf("第%d 号包重传失败! \n\n", ack);
                return 0;
                //发送失败则打印信息并终止程序
            }
            printf("第%d 号包已重传! \n", ack);
            //反之则发送成功
            total++; //发包总次数+1
            cnt++; //重新发包次数+1
        }
        else if(time_limit_result != SOCKET_ERROR) { //代表没有超时且收到了服务器端的数据包
            int temp = sizeof(client_receive);
            long long receive_code = recvfrom(sock, id, sizeof(id), 0, (SOCKADDR *)&client_receive, &temp);
            //收取服务器端发来的数据包
            memcpy(&current_id, id, 4); //因为服务器端的数据包内容为一个 4 字节大小的 char 数组，包含确认 ack 的地址
            //所以将该地址复制到 current_id 的地址中，相当于使得 current_id 的值修改为该地址指向的整数
            if(current_id == ack) { //若相等则证明本次发包成功，服务器端成功接收到了这一数据包，且客户端收到了确认信息
                //使得可以进行下一数据包的发送，停等协议的作用正体现于此
                if(cnt != 0) {
                    printf("重传第%d 号包成功! \n\n", ack); //不为 0 则代表该包之前有丢包或没收到确认 ack，进行过重新发包
                }
                else {

```

```

        printf("第%d 号包发送成功! \n\n",ack);//为 0 则代表
第一次发包就成功了
    }
    break;//可以进行下一数据包的发送了
}
else{
    continue;//若 current_id 不等于 ack，则继续循环等待，直
至匹配再跳出循环
}
}
}
ack++;//数据包序号+1
cnt=0;//重传次数清零
file_size-=send_packet_size;//文件大小减去一个数据包的大小
}
//为了使得在发送完文件之后服务器端程序可以自动终止，我们在发完所有数据之后
额外添加一个数据包
memcpy(id,&ack,4);//将 ack 当前地址以 4 字节形式复制到 id 数组中
sendto(sock_end,id,sizeof(id),0,(SOCKADDR *)&client_send,sizeof(cli
ent_send));
//发送一个只有 4 字节的数据包，服务器收到后会自动终止程序
GET_TIME(end);//获取文件传输结束时间
printf("传输时间为: %f 秒\n",end-start);//打印传输时间
printf("%d %d\n",ack,total);
double sum=(1-ack*1.0/total)*100.0;
printf("丢包率为%f%%",sum);//计算丢包率
fclose(fp);//关闭文件
closesocket(sock);
closesocket(sock_end);
//关闭套接字
WSACleanup();//清理网络环境,释放 socket 所占的资源
}

```

## ②服务器端:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/time.h>
#include<winsock2.h>
#include<windows.h>
#include<string.h>

#pragma comment(lib,"ws2_32.lib");//把 ws2_32.lib 这个库加入到工程文件中

#define packet_size 1024//规定了 UDP 每次传递的数据包的大小

#define GET_TIME(now) { \
    struct timeval t; \
    gettimeofday(&t, NULL); \
    now = t.tv_sec + t.tv_usec/1000000.0; \
}
//最后一行将 us 转换为 s，统一单位
//用于计算程序运行时间，也即文件传输时间

int main()
{
    WSADATA data;//存放 windows socket 初始化信息,初始化网络环境
    int state=WSAStartup(MAKEWORD(2,2),&data);//使用 2.2 版本的 Socket
    if(state!=0){
        printf("初始化失败! \n");
        return 0;
        //若初始化失败则打印信息并终止程序
    }
    double start,end;//用于存储文件传输的开始时间和结束时间
    fd_set rfd;//主要用于 select 函数 (select 函数的用处会在后面提及)
    //是一组文件描述字(fd)的集合
    SOCKET sock=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);//生成套接字,
sock 用于发送仅包含确认信息 ack 的数据包
    if(sock==INVALID_SOCKET){
        printf("套接字生成失败!\n\n");
        return 0;
        //生成套接字失败则打印信息并终止程序
    }
    //struct sockaddr_in:用来处理网络通信的地址，并把 port 和 addr 分开储存在
两个变量中
    //用来处理网络通信的地址
    struct sockaddr_in server_send;
```

```

    struct sockaddr_in server_receive;
    server_receive.sin_family=AF_INET;
    server_receive.sin_port=htons(10000); //设置端口为 10000（服务器上设置
    端口 10000 开放）
    server_receive.sin_addr.s_addr=INADDR_ANY; //监听本机的所有 IP
    //绑定 INADDR_ANY，使得只需管理一个套接字，不管数据是从哪个 IP 过来的，只
    要是绑定的端口号过来的数据，都可以接收到。
    //我使用的服务器有 2 个 IP: 10.0.8.8 和 49.232.4.77
    if(bind(sock,(LPSOCKADDR)&server_receive,sizeof(server_receive))==S
    OCKET_ERROR) {
        printf("套接字绑定端口失败！\n");
        return 0;
        //将套接字与指定端口进行绑定，失败则打印信息并终止程序
    }
    long long ack=0; //服务器端返回的确认信息 ack
    char packet[packet_size]; //用于存储收到的每个数据包
    char id[4]; //存储数据包的前四位，即序号
    FILE* fp = fopen("receive7.png","wb"); //这里不太智能，需要提前输入准备
    接收的文件的后缀名
    //并设置为以二进制形式写入文件
    long long current_id=0; //保存当前数据包的序号
    int temp=sizeof(server_receive);
    while(true){ //该循环内用于循环确认是否收到数据包和是否需要重传确认信息
ack
        long long receive_code=recvfrom(sock,packet,sizeof(packet),0,(S
    OCKADDR *)&server_send,&temp);
        if(receive_code==4){ //若等于 4 则代表收到了最后一个特有的数据包：只含
        有 4 字节 ack 信息的数据包
            break; //收到该数据包后即可跳出接收数据包的循环
        }
        if(receive_code==SOCKET_ERROR){
            printf("recvfrom() failed!Error code:%d\n",WSAGetLastError(
    ));
            break;
            //若接收数据包失败则打印出错误代码并跳出循环
        }
        memcpy(&current_id,packet,4); //将数据包中的前 4 位信息复制到
current_id 的地址中
        //相当于使得 current_id 的值修改为该地址指向的整数
        if(current_id<ack){ //若小于则证明存在数据包丢失或 ack 丢失的情况发生
            memcpy(id,&current_id,4); //将 current_id 的地址以 4 字节形式赋值
            到 id 数组中
            printf("客户端可能未收到服务器端返回的第%d 号包的确认 ACK，即将重
            传该包的确认 ACK.....\n",current_id);

```



```

        //打印提示信息
        sendto(sock,id,sizeof(id),0,(SOCKADDR *)&server_send,sizeof
(server_send));//重新发送确认 ack
        printf("已重传! \n");
    }
    else{//证明 current_id==ack, 即上一个 ack 已被客户端接收到
        printf("服务器端已收到第%d 号包! \n",current_id);
        fwrite(packet+4,sizeof(byte),receive_code-4,fp);//将数据包中
除了代表序号的前 4 位信息以外的所有内容以字节为单位写入到 fp 指向的文件中
        memcpy(id,&ack,4);//将 ack 的地址以 4 字节形式赋值到 id 数组中
        sendto(sock,id,4,0,(SOCKADDR *)&server_send, sizeof(server_
send));//发送确认 ack
        ack++;//ack+1,用于和下一个数据包的序号进行比较和发送下一个数据包
的确认信息
    }
}
fclose(fp);//关闭文件
closesocket(sock);//关闭套接字
WSACleanup();//清理网络环境,释放 socket 所占的资源
}

```

## 4. 实验结果

①文件大小为 1M 左右时:

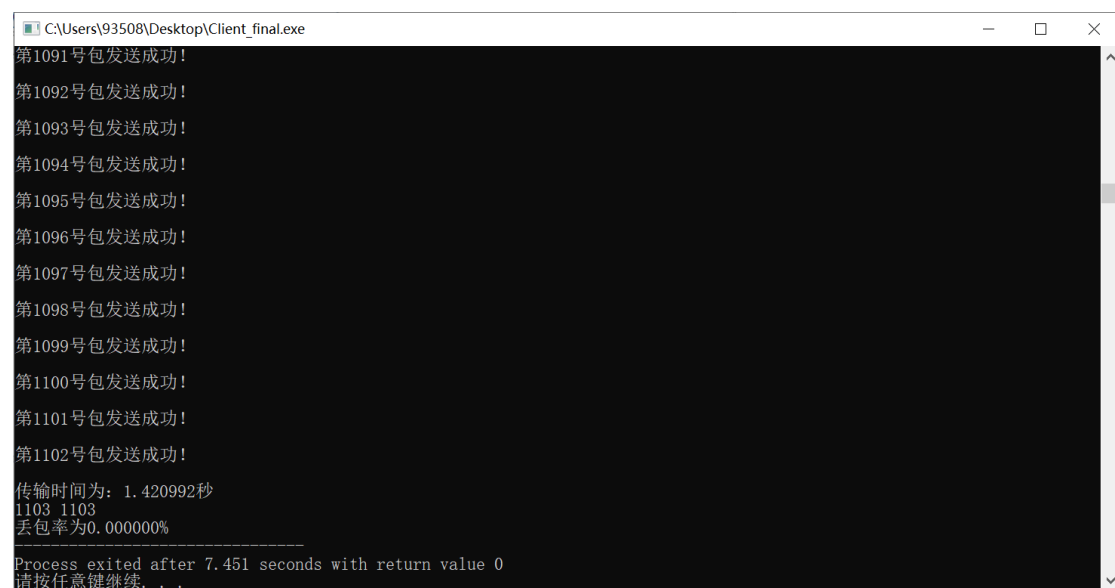
(1) 设置超时时间为

```

time_limit.tv_sec=1;
time_limit.tv_usec=200;

```

则实验结果为:



```

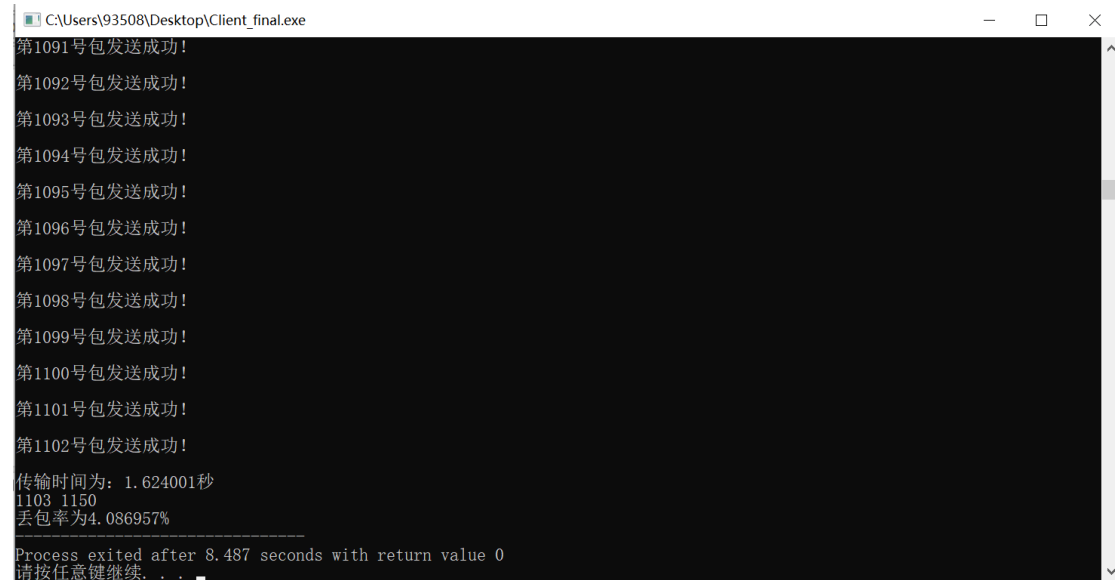
C:\Users\93508\Desktop\Client_final.exe
第1091号包发送成功!
第1092号包发送成功!
第1093号包发送成功!
第1094号包发送成功!
第1095号包发送成功!
第1096号包发送成功!
第1097号包发送成功!
第1098号包发送成功!
第1099号包发送成功!
第1100号包发送成功!
第1101号包发送成功!
第1102号包发送成功!
传输时间为: 1.420992秒
1103 1103
丢包率为0.000000%
-----
Process exited after 7.451 seconds with return value 0
请按任意键继续. . .

```

## (2) 设置超时时间为

```
time_limit.tv_sec=0;  
time_limit.tv_usec=100;
```

则实验结果为：



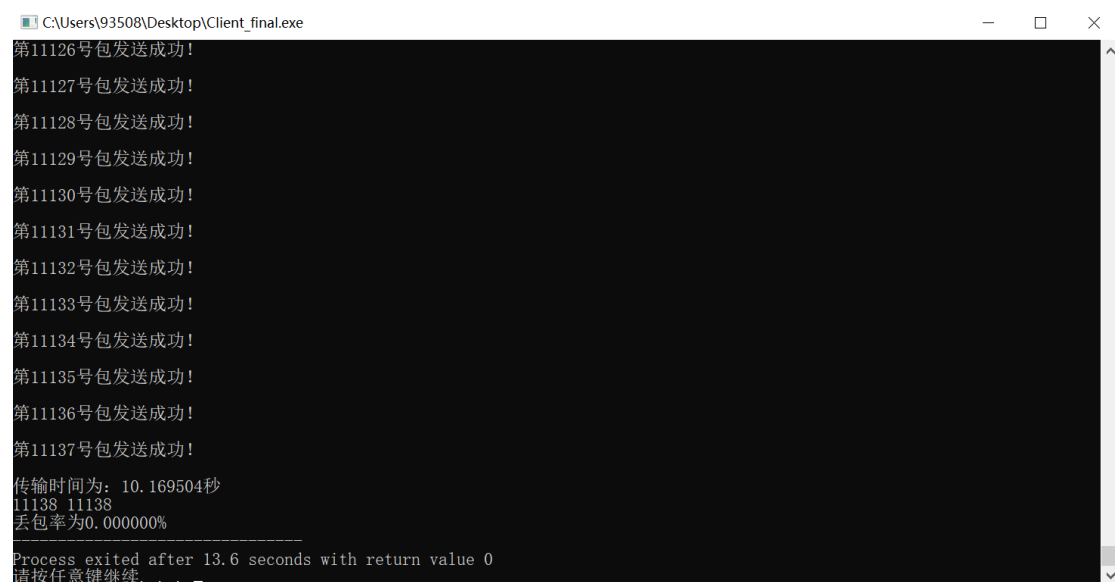
```
C:\Users\93508\Desktop\Client_final.exe  
第1091号包发送成功!  
第1092号包发送成功!  
第1093号包发送成功!  
第1094号包发送成功!  
第1095号包发送成功!  
第1096号包发送成功!  
第1097号包发送成功!  
第1098号包发送成功!  
第1099号包发送成功!  
第1100号包发送成功!  
第1101号包发送成功!  
第1102号包发送成功!  
传输时间为: 1.624001秒  
1103 1150  
丢包率为4.086957%  
-----  
Process exited after 8.487 seconds with return value 0  
请按任意键继续. . .
```

## ②文件大小为 10M 左右时：

### (1) 设置超时时间为

```
time_limit.tv_sec=1;  
time_limit.tv_usec=200;
```

则实验结果为：

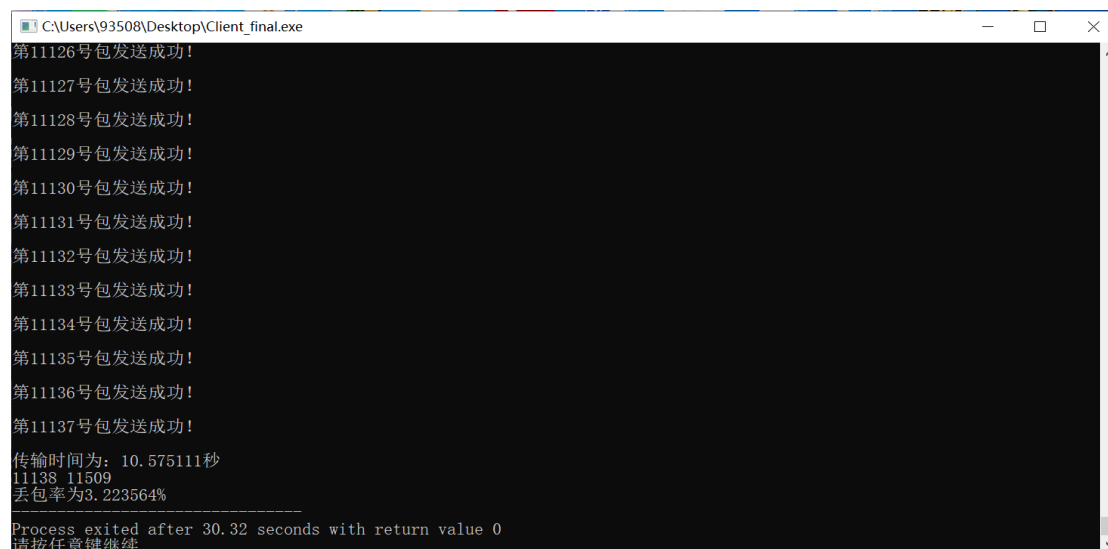


```
C:\Users\93508\Desktop\Client_final.exe  
第11126号包发送成功!  
第11127号包发送成功!  
第11128号包发送成功!  
第11129号包发送成功!  
第11130号包发送成功!  
第11131号包发送成功!  
第11132号包发送成功!  
第11133号包发送成功!  
第11134号包发送成功!  
第11135号包发送成功!  
第11136号包发送成功!  
第11137号包发送成功!  
传输时间为: 10.169504秒  
11138 11138  
丢包率为0.000000%  
-----  
Process exited after 13.6 seconds with return value 0  
请按任意键继续. . .
```

## (2) 设置超时时间为

```
time_limit.tv_sec=0;  
time_limit.tv_usec=100;
```

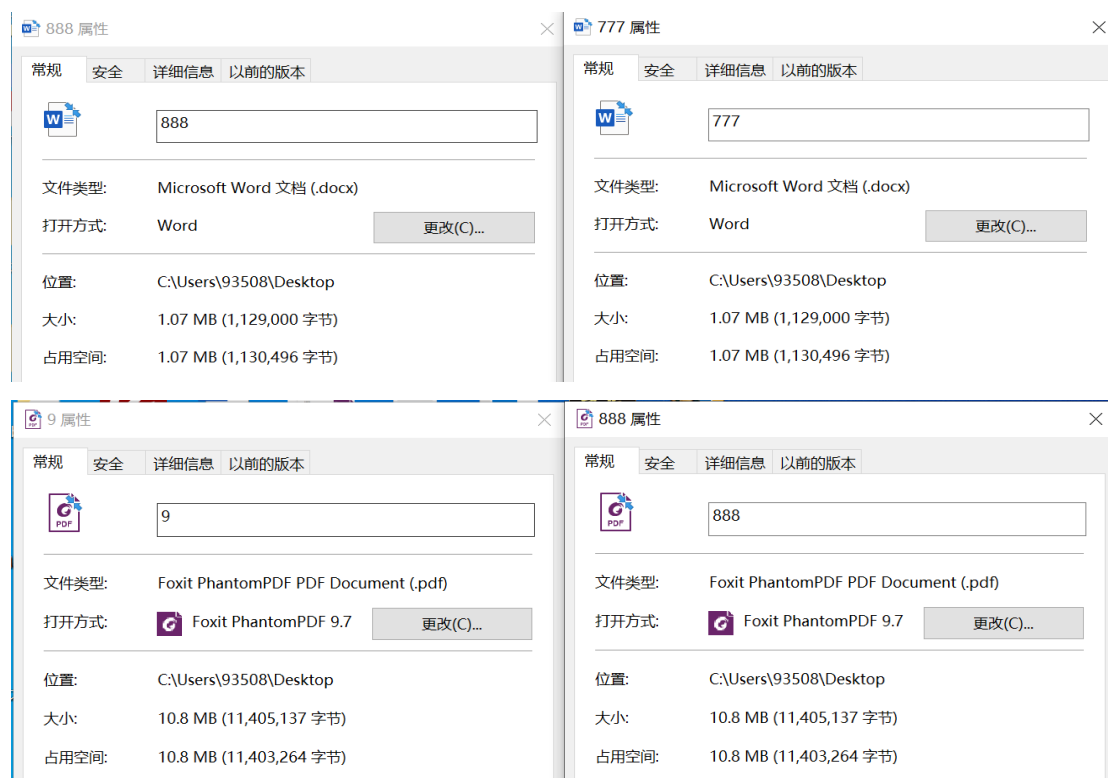
则实验结果为：



```
C:\Users\93508\Desktop\Client_final.exe  
第11126号包发送成功!  
第11127号包发送成功!  
第11128号包发送成功!  
第11129号包发送成功!  
第11130号包发送成功!  
第11131号包发送成功!  
第11132号包发送成功!  
第11133号包发送成功!  
第11134号包发送成功!  
第11135号包发送成功!  
第11136号包发送成功!  
第11137号包发送成功!  
传输时间为: 10.575111秒  
11138 11509  
丢包率为3.223564%  
-----  
Process exited after 30.32 seconds with return value 0  
请按任意键继续. . .
```

由①、②结果可知，超时判定的时间标准（time\_limit）和文件大小均对传输时间和丢包率有一定程度上的影响。

关于文件一致性的认证：我采取了最简单的办法：肉眼比较文件大小



对比可知，文件传输前后保持一致，该实验圆满完成。