基于UDP服务设计可靠传输协议并编程实现

李娅琦
2213603
计算机科学与技术

实验要求

在实验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

实验设置

代码文件

本次文件的代码分为三个文件：

- packet.h: 对UDP数据包的协议设计
- client.cpp: 客户端代码
- server.cpp: 服务器代码

条件设置

- 设置服务器和客户端的IP都为：127.0.0.1
- 服务器端口号：8888
- 客户端端口号：9999
- 发送端窗口大小：**30**
- 设置丢包率：10%，延时：20ms

实验原理

滑动窗口

滑动窗口协议的基本原理就是在任意时刻，发送方维持一个连续的允许发送的包的序号，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包的序号，称为接收窗口。

- 发送窗口和接收窗口的序号上下界不一定一样，大小也可不同
- 发送方窗口内的序列号代表**可以被发送还未发送的包**或**已经被发送但未被确认的数据包**

GBN累积确认

后退N包协议（GBN）是一种流水线协议，发送方在发完一个数据包后，不停下来等待应答包，而是**连续发送若干个数据包**，即使在连续发送过程中收到了接收方发来的应答包，也可以继续发送。且发送方在每发送完一个数据包时都要设置超时定时器。只要在所设置的超时时间内仍未收到确认包，就要重发相应的数据包。
GBN 使用累积确认，这意味着接收方只需确认最近接收的连续数据包序列号。如果接收方收到的数据包序列号

大于期望的序列号，则该包及其后续包会被暂存，直到缺失的包到达并被确认。

对于GBN协议：

- 发送窗口大小>1
- 接受窗口大小=1

超时重传

在GBN协议中，超时重传是一个关键机制。如果发送方在预定的超时时间内没有收到某个数据包的确认，它会重新发送从该数据包开始的所有数据包。超时机制确保了数据的可靠传输，即使在网络状况不佳的情况下。
为了实现超时重传，发送方需要维护一个**定时器**，当发送数据包后启动，如果在定时器到期前未收到确认，则触发重传。

累计确认

- 接收方不需要为每个接收到的数据包发送单独的确认
- 接收方通过发送一个确认来表明它已成功接收到包括**该确认号及其之前所有的数据包**
- 降低了网络负载，提高通信效率，特别是在高延迟或高带宽的网络环境

然而，累计确认机制也有缺点：如果一个数据包丢失，发送方可能需要重传所有未被确认的数据包，即使其中一些包已经被接收方正确接收。这可能导致网络上的**数据包重复**，从而浪费带宽。

Packet协议设计

关于Packet，采用实验一的设计：

```
#define SIZE 32768  // 数据大小
// 定义标志位
#define ACK 1  // ACK 标志位 (00000001)
#define SYN 2  // SYN 标志位 (00000010)
#define FIN 4  // FIN 标志位 (00000100)
#define ACK_SYN 3
#define FIN_ACK 5
#define OVER 8
#define NAME 6
// 数据包（伪首部+长度+校验和+数据）
struct Packet {
    uint32_t seq_num = 0;  // 序列号（第几个包）
    uint32_t ack_num = 0;  // 确认号
    uint8_t sign = 0; //协议，低三位ACK,SYN,FIN
    uint16_t len=0;  // 数据长度
    uint16_t checksum=0;  // 校验和
```

```
    char data[SIZE] = { 0 };  // 数据内容
};
```

差错检测

校验和和数据包的差错校验函数同实验3-1，未作变动。

实验流程

1. **建立连接：**
    - 发送端发送第一次握手
    - 接收端接收第一次握手，并发送第二次握手
    - 发送端接收第二次握手，并发送第三次握手
2. **文件数据传输：**
    - 发送端发送文件数据
    - 接收端接收文件数据，对每个数据包进行校验和验证，如果校验和正确，发送确认 ACK 给发送端，表示成功接收数据
3. **断开连接：**
    - 接收端发送第一次挥手
    - 发送端接收第一次挥手，并发送第二次挥手
    - 接收端接收第二次挥手，并发送第三次挥手
    - 发送端接收第三次挥手，并发送第四次挥手

三次握手和四次挥手

基于3-1中的实验代码，未作过多的改进

文件传输

读取写入

对于文件传输，在while循环中不断让用户进行选择：

- 输入文件(Y|y): 用户将输入文件名进行文件的传输
- (N|n): 用户不再进行文件传输则退出循环

```
while (true)
{
    cout << "\033[1;34mChoice: \033[0m 请选择是否输入文件(Y/N)" << endl;
    cin >> choice;
    if (ifconnect && (choice == ('Y' | 'y'))) {
        //传输文件
    }
    if (choice == ('N' | 'n')) {
        break;
    }
    ...//省略文件读取和传输的函数，后面详细解释
}
```

1. 文件读取 在用户输入文件名字后，以二进制的方式打开文件，并利用函数进行文件读取：

- seekg(0, ios::end)： 将文件指针移动到文件的末尾
- tellg()： 获取当前位置（即文件的长度）
- seekg(0, ios::beg)： 将文件指针移动回文件的开头
- read(buffer, len)： 二进制文件的读取（读取len长度到buffer中）

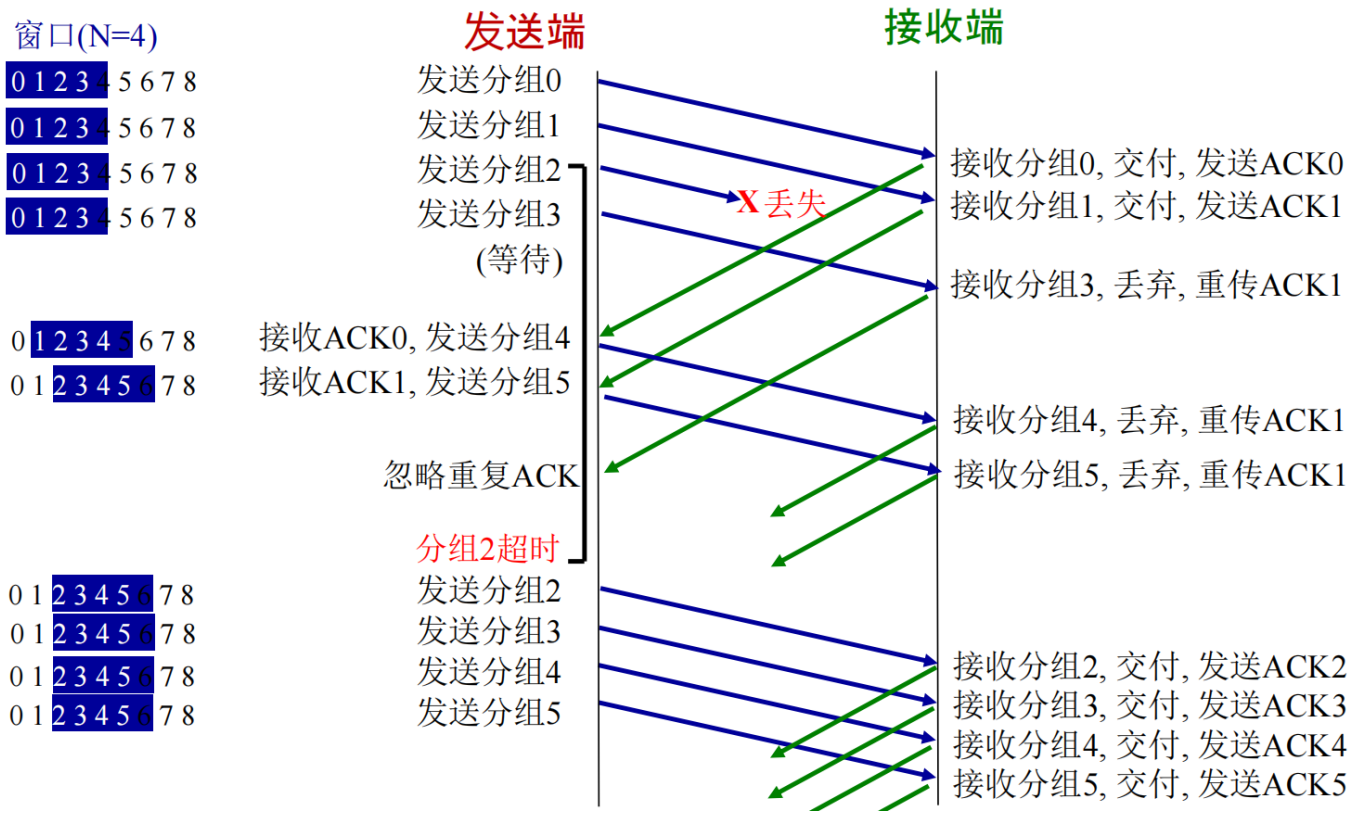
```
    cin >> file;
    ifstream f(file.c_str(), ifstream::binary); // 以二进制方式打开文件
    if (!f.is_open()) {//无法打开文件
        return false;
    }
    f.seekg(0, ios::end);
    int len = f.tellg();
    f.seekg(0, ios::beg);
    char* buffer = new char[len];
    f.read(buffer, len);
    f.close();
```

2. 文件写入 同样以二进制的形式打开要写入的文件，因为进行了文件名和数据的传输，因此可以将数据写入对应文件名中。

```
ofstream fout(a.c_str(), ofstream::binary);
for (int i = 0; i < datalen; i++)
{
    fout << recvdata[i];
}
//fout.write(recvdata, datalen);
fout.close()
```

整体逻辑

参考理论知识，关于文件传输，采用以下逻辑：



基于此，设置了

- **WINDOW_SIZE：** 发送端窗口大小
- **base：** 窗口起始位置
- **nextseqnum：** 下一个可用且待发送的序列号
- **last_recvack：** 上一个接收到的ack序列号
- **stime=clock_t[]：** 记录每个包的发送时间（实时更新）

下面将基于不同的功能分别从客户端和服务器进行详细介绍。

超时重传

1. **延迟和丢包设置**

本次实验并未采用路由器，而是在程序中模拟实现丢包和延迟。

- 模拟丢包
  - 设置丢包的概率为**10%**
  - 当接收到数据包，根据丢包概率决定是否丢弃接收的数据包
  - 若丢弃，则直接**continue**以跳过当前这次循环，不进行后续判断以及**ACK包的返回**
- 模拟延迟
  - 设置延迟为**10ms**
  - 使用**sleep_for()** 对程序进行延迟

另外，需要注意的是，当收到的数据包的**校验和出错**时，也会进行丢包处理。

```
// 服务器模拟丢包：根据丢包概率决定是否丢弃接收的数据包
if (rand() / (double)RAND_MAX < 0.1)
```

```
{
    continue; // 丢包，跳过当前循环
}
if (!ifcorrect(messpack, sizeof(messpack))) {
    continue;
}
// 服务器模拟接收延迟
this_thread::sleep_for(std::chrono::milliseconds(10));
```

2. **超时判断与重传**

在客户端利用**clock()** 来判断是否超时

- 设置全局变量**TIMEOUT=500 ms**
- 利用**clock()数组**存储所有包的发送时间（会实时更新）。如果有一个包被重新发送，那么所记录的时间也会**更新为新发送的时间**
- 每次只需判定**窗口开始对应数据包**是否超时。因为窗口内的包为已发送未收到ack的数据包，而其他包时间一定迟于base对应的数据包，因此窗口左边界base对应**下一个待接收ack的数据包**
- 如果超时，则进行回退处理，**重新发送窗口内所有的数据包**

```
if (clock() - stime[base] > TIMEOUT) {
    //超时重传
    nextseqnum = base;
    cout << "\033[1;31mInfo: \033[0m 超时...... " << endl;
    break;
}
```

GBN回退与累计确认

1. **客户端**

- 在数据发送前，计算数据包的个数以及每个数据包的大小
- 对数据包进行设置，包括
    - 数据部分
    - 是否是NAME数据包
    - 序列号（从0开始计算）和校验和
- 当窗口内有数据包存在时，利用**while循环**发送窗口内所有**可用还未发送的数据包**，并记录时间
- 接收ack，超时未收到则重新发送窗口内所有的包（按照超时重传部分所述）
- 收到ack，判断是否和**前一个ack相同**，相同则说明出现了丢包或者乱序的情况，窗口内所有未确认的包都需要重传，即**nextseqnum = base**
    - 更新上一个收到的ack序列号为当前的序列号，即**last_recvack = acknum**
- 收到新的ack（即大于之前收到的任何确认号），则**向前移动窗口**，即**base = acknum +1**
    - 此时序列号小于或等于收到的确认号的所有包都被移除，因为**累计确认意味着所有这些包都被正确接收**

这里因为对于GBN来说，即使在**连续发送过程中收到接收方发来的应答包**，也可以继续发送：

- 在整个循环中，每发一个数据包就尝试接受一次（即发送过程中接收到）

- 未接收到或者接收到都会再次进入第一个if语句，即**继续发送数据包**
- 只有当窗口内无可以发送的数据包时，才一直进行接收

本来实现的是连续发送过程中不会接收ack,因此改为现在这种可以接收，只需：

- 第一个while循环发送换为if语句
- 第二个while循环接收取消，并更改break为continue

```
int base = 0; // 发送窗口的左边界（最小的未确认包序列号）
int nextseqnum = 0; // 下一个可用且未发送的序列号
int packnum = (len % SIZE == 0) ? len / SIZE : len / SIZE + 1; // 总包数
int last_recvack = -1; // 上一次收到的ack号
char* messbuf = new char[sizeof(Packet)];
clock_t* stime = new clock_t[packnum];// 用来记录每个包的发送时间
//总的包：0 ~ packnum-1
while (base < packnum ) {
    if (nextseqnum < base + WINDOW_SIZE && nextseqnum < packnum) {// 窗口内有数据包
可以发送就发送
        ...//设置包
        // 记录发送时间
        stime[nextseqnum] = clock();
        sendto(...);
        cout << "\033[1;33mSend: \033[0m 发送包，序列号 " << nextseqnum << endl;
        nextseqnum++; // 发送下一个包
    }
    //while (true) {
        // 设置非阻塞模式
        u_long mode = 1;
        ioctlsocket(clientsock, FIONBIO, &mode);
        sresult = recvfrom(...);
        if (clock() - stime[base] > TIMEOUT) {//超时重传
            nextseqnum = base;
            continue;
            //break;
        }
        if (sresult == -1) {
            continue;
        }
        if (ifcorrect(ackp, sizeof(ackp)) && ackp.sign == ACK) {
            cout << "\033[1;32mReceive: \033[0m 接收到来自服务器的ACK " <<
ackp.seq_num << endl;
            int acknum = ackp.seq_num;
            if (acknum >= base ) {
                base = acknum +1;  // 窗口左边界向前滑动
                if (last_recvack == acknum) {// 丢包窗口内所有未确认的包都重传
                    nextseqnum = base;
                }
                last_recvack = acknum;
            }
            //break;
        }
    //}
```

```
    u_long mode = 0;
    ioctlsocket(clientsock, FIONBIO, &mode);  // 改回阻塞模式
}
```

2. **服务器**

服务器接收端的窗口大小为1。

- 服务器会在while循环中不断接收客户端发来的数据包并发送ACK
- 校验和不对则丢弃数据包，再进行接收
- 服务器会实时记录**已经收到的seq序列号**，即期望接收的序列号为：**seq+1**
- 下面将根据接收到的数据包的序列号进行不同情况的处理：
    - 序列号大于期望序列号：表示接收到了非顺序的数据包，可能是由于中间有数据包丢失。服务器将**回退到上一个确认的序列号并发送ACK**，即**seq-1**
    - 序列号小于期望序列号：表示收到了**重复的数据包**。这可能是因为之前的ACK丢失或延迟到达。服务器也将**发送上一个确认的序列号的ACK**
    - 序列号等于期望序列号：数据包按顺序到达，数据保留至对应的数组（之后再写入文件），并更新接收的总字节数。同时，**seq++** 并发送相应的ACK

```
while (true)
{
    int length=0;
    remsult=recvfrom(...);
    if (remsult == -1)// 如果接收失败，继续尝试
    {continue;}
    else {
        ...//一些判断和丢包延迟处理
        int t = int(messpack.seq_num);
        if (seq != t) {
            Packet ackpack;
            ...ackpack.seq_num = seq-1;//返回接收到最大的序列号
            ... sendto(...);// 返回ACK
        }
        else if (messpack.len > 0) {
            if (ifname) { //是名字还是数据
                memcpy(name + total_len, messpack.data, messpack.len);
            } else {
                memcpy(recvdata + total_len, messpack.data, messpack.len);
            }
            total_len += messpack.len;
            ... sendto(...); // 返回ACK
            seq++; //改变序列号
        }
    }
}
```

结束标志

当文件传输结束时（具体代码参考.cpp文件）：

- 客户端发送**OVER数据包**
- 服务器接收到后，**返回OVER数据包**
- 客户端接收到服务器的OVER数据包，表明此刻数据传输结束
- 最后会返回接收到的所有数据及长度进行**文件写入**

吞吐率和传输时间

在客户端进行文件名字和数据的发送，并且在发送数据时记录吞吐率和传输时间：

```
while (true)
{
    ...
    sendfile((char*)(file.c_str()), file.length(), true);// 发送文件名
    clock_t start1 = clock();
    sendfile(buffer, len, false);// 发送文件内容
    clock_t end1 = clock();
    cout << "传输总时间为:" << (end1 - start1) / CLOCKS_PER_SEC << "s" << endl;
    cout << "吞吐率为:" << fixed << setprecision(2) << (((double)len) / ((end1 -
start1) / CLOCKS_PER_SEC)) << "byte/s" << endl;
}
```

结果展示

建立连接

先运行server端，再运行client端：

```
Info:  正在监听客户端.....

-----开始握手-----

First:
Receive:  接收到客户端连接请求.....

Second:
Send:  服务器发送ACK确认.....

Third:
Receive:  接收到客户端ACK确认.....
Success:  服务器与客户端成功连接
Choice:  是否接收文件(Y/N)
```

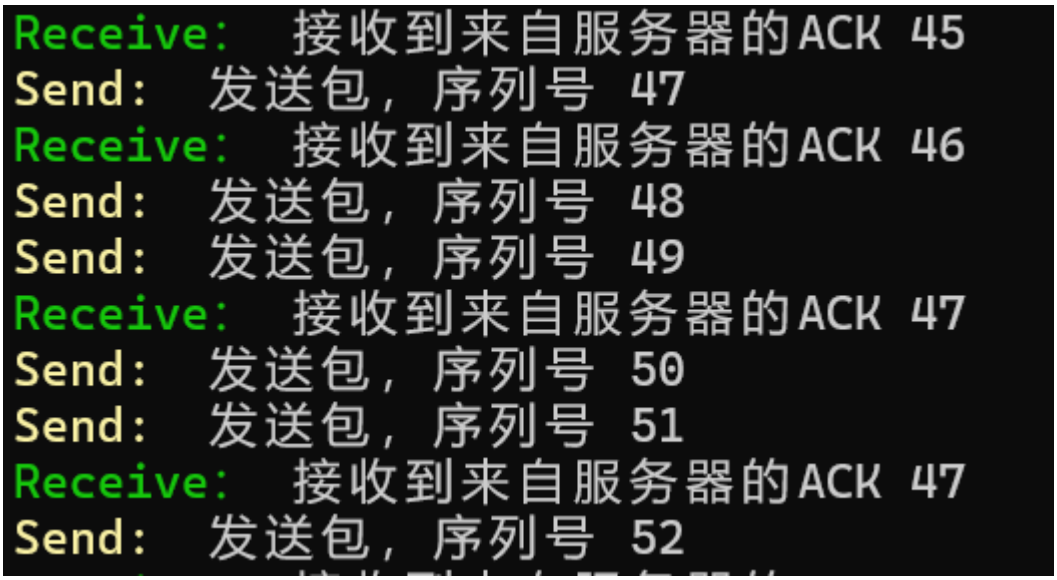
```
-----开始握手-----

First:
Send:  发送连接请求成功,等待响应中...

Second:
Receive:  接收到来自服务器的ACK...

Third:
Send:  客户端发送ACK确认.....
Success:  服务器与客户端成功连接

Choice:  请选择是否输入文件(Y/N)
```

超时重传

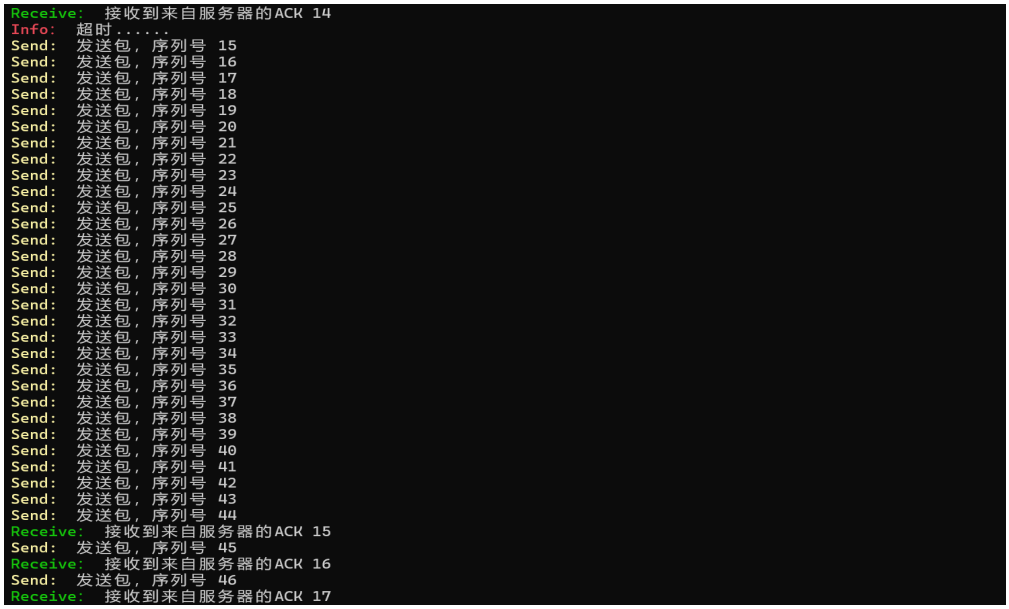
以传输第三个图像为例，**注意这不是停等机制，而是因为延迟的结果**

- 采用的是接收与发送同时发生
- 由于**延迟 20 ms**的存在，会导致发送完才会能够接收到ACK
- 如果**不延迟**，会出现**没发送完窗口内所有的包就接收到ACK**

不延迟的结果：（其中因为窗口内还有可以发送的数据包，所以收到重复ACK依然会进行数据包的发送）



延迟的后果： (延迟导致发送完才会能够接收到ACK)



通过**延迟与不延迟**，都可以看出：

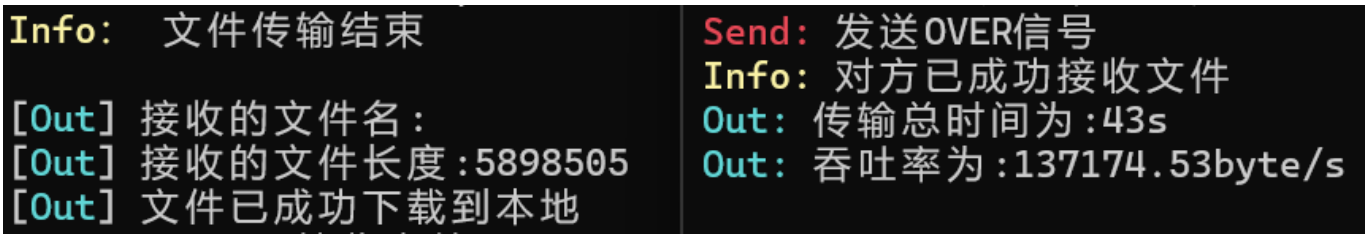
- 可以看到，每当超时出现，会重发窗口内所有的数据包；
- 当接收到一个ACK后，窗口发生移动，会再次发送一个数据包。
- 持续上面操作...

最后几个数据包的传输如下：

- 超时时，由于剩下不到30个数据包，只会发送剩下的数据包；
- 接收到ACK，由于没有额外的未发送的数据包，因此不会进行发送

数据传输

连接建立后，会给出选择，若选择Y|y，则进行文件传输，输入对应的文件名字即可，结果如下：



打开server下的文件可以看到传输得到的文件，与原有文件相同（由于检查过程中已经检查了这里不在再展示）。

断开连接

当文件传输结束，会给出选择，如果选择N|n，则进行四次挥手断开连接：

总结

在此次实验中，会发现使用GBN协议传输的时间远远长于停等协议，可能与本次实验设置有关：

- 窗口大小为30，如果头几个包超时，会导致重发的数据包非常多，大大降低了效率
- 可能与**超时**的时间设置，**延时**的设置等等有关

总而言之，通过这次实验，加深了对UDP协议、滑动窗口、Go-Back-N重传机制等概念的理解，并且进行了实践，收获颇多。