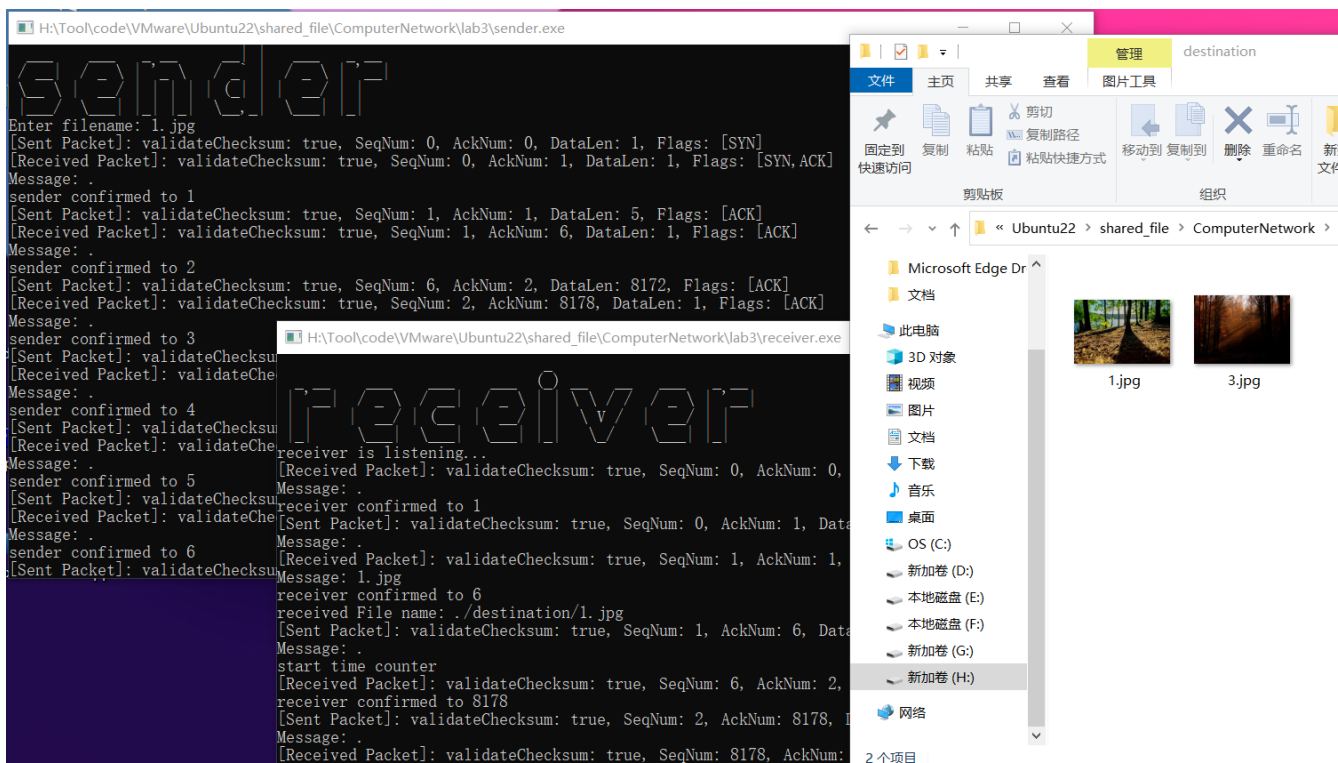


lab3-1实验报告

杜怡兴-2112847

注:在助教答辩过程当中,我提到了本路由器特性:接收方回复的ACK不会丢包,因而不必处理也可以得到正确结果,经过测试,按照原来的设计,即使接收方丢包也可以得到正确结果,详情见最终实现-9部分



lab3-1实验报告

杜怡兴-2112847

协议设计

1. 数据设计
2. 时序设计
 - 四次握手
 - 四次挥手过程
3. 超时重传机制
4. 差错校验

实验环境

- 如何编译
- 如何传输

基础实现

协议实现

最终实现

0. 建立连接
1. 超时重传
2. 差错校验
3. 接收确认
4. 动态调整RTO
5. 停等机制
6. 传输时间和平均吞吐率显示
7. 解决最后一次挥手问题
8. RTT大于RTO时正确传输
9. 接收方ACK回复丢失时正确传输

实验中的坑

协议设计

1. 数据设计

变量名称	含义	长度 (字节)
checksum	校验和	2
seqNum	序列号	4
ackNum	累计确认号	4
dataLen	数据长度	2
flags	标志位	1
packetNum	数据包序号	2
reserved	保留字段	5

- 总长度: 20字节
- 最大文件总长度: 8192字节
- 最大数据长度: 8172字节

双方代码都需要引入协议头文件protocol.h

```
struct Packet {
    uint16_t checksum = 0;
    uint32_t seqNum = 0;
    uint32_t ackNum = 0;
    uint16_t dataLen = 0;
    uint8_t  flags = 0;
    uint16_t packetNum = 0;
    char     reserved[5] = {0, 0, 0, 0, 0};
    char     message[8172];
};
#pragma pack(pop)

enum Flag {
    SYN = 1,
    ACK = 2,
    FIN = 4,
    // 更多标志位
};
```

说明:

- 使用 #pragma pack(push, 1) 和 #pragma pack(pop) 来确保结构体成员紧密排列，无额外填充
- enum Flag 定义了TCP标志位: ACK, SYN, FIN

2. 时序设计

特别注意:握手挥手和数据确认的数据包里面没有数据,此时用一个小数点来表示数据,数据段长度1字节,目的是把所有数据包区分开来,方便接收确认。

这样做的好处是，例如发送方每次确认，ackNum恰好加1，如图，很容易差错检测

The screenshot displays a terminal window with two columns of output. The left column shows the sender's perspective, and the right column shows the receiver's perspective. A file explorer window is overlaid on the right side of the terminal.

Sender Output:

```

Enter filename: 1.jpg
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
[Received Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
Message: .
sender confirmed to 1
[Sent Packet]: validateChecksum: true, SeqNum: 1, AckNum: 0, DataLen: 1, Flags: [ACK]
[Received Packet]: validateChecksum: true, SeqNum: 1, AckNum: 0, DataLen: 1, Flags: [ACK]
Message: .
sender confirmed to 2
[Sent Packet]: validateChecksum: true, SeqNum: 2, AckNum: 0, DataLen: 1, Flags: [ACK]
[Received Packet]: validateChecksum: true, SeqNum: 2, AckNum: 0, DataLen: 1, Flags: [ACK]
Message: .
sender confirmed to 3
[Sent Packet]: validateChecksum: true, SeqNum: 3, AckNum: 0, DataLen: 1, Flags: [ACK]
[Received Packet]: validateChecksum: true, SeqNum: 3, AckNum: 0, DataLen: 1, Flags: [ACK]
Message: .
sender confirmed to 4
[Sent Packet]: validateChecksum: true, SeqNum: 4, AckNum: 0, DataLen: 1, Flags: [ACK]
[Received Packet]: validateChecksum: true, SeqNum: 4, AckNum: 0, DataLen: 1, Flags: [ACK]
Message: .
sender confirmed to 5

```

Receiver Output:

```

receiver is listening...
[Received Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
Message: .
receiver confirmed to 1
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 1, DataLen: 1, Flags: [ACK]
Message: .
[Received Packet]: validateChecksum: true, SeqNum: 1, AckNum: 1, DataLen: 1, Flags: [ACK]
Message: 1.jpg
receiver confirmed to 6
received File name: ./destination/1.jpg
[Sent Packet]: validateChecksum: true, SeqNum: 1, AckNum: 6, DataLen: 1, Flags: [ACK]
Message: .
start time counter

```

File Explorer Window:

The file explorer window shows the path `H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab3\` and contains a file named `1.jpg`.

四次握手

1. **发送方发送SYN:**
 - 发送方向接收方发送一个带有SYN标志的数据包，以初始化连接。
2. **接收方发送SYN, ACK:**
 - 接收方回应一个带有SYN和ACK标志的数据包，确认连接请求。
3. **发送方发送ACK + 文件名:**
 - 发送方发送一个带有ACK标志的数据包和文件名。
4. **接收方发送ACK:**

- 接收方发送一个带有ACK标志的数据包，确认文件名接收。



四次挥手过程

1. 发送方发送FIN ACK:

- 发送方完成数据传输后发送一个带有FIN ACK标志的数据包，请求关闭连接。

2. 接收方发送ACK:

- 接收方回应一个带有ACK标志的数据包，确认关闭请求。

3. 接收方发送FIN ACK:

- 接收方完成数据处理后发送一个带有FIN ACK标志的数据包，请求关闭连接。

4. 发送方发送ACK:

- 发送方发送一个带有ACK标志的数据包，确认关闭请求。



3. 超时重传机制

1. 超时重传逻辑:

发送方等待接收方响应的逻辑。在等待期间，发送方使用 `recvfrom` 函数来接收来自接收方的响应。此函数是**阻塞的**，意味着它会一直等待，直到有数据到达或者等待超时。**阻塞状态会在两种情况下终止：**

1. **计时超时**：发送方设置了一个超时时间（RTO）。如果超过了这个时间，发送方会认为接收方没有响应。在这种情况下，发送方会触发超时重传机制，重新发送之前的数据包，并等待接收方的响应。
2. **收到包**：当 `recvfrom` 函数返回时，发送方会检查接收到的数据包是否满足期望的条件。如果数据包不符合预期，发送方也会认为接收方没有正确响应，此时重新等待接收方的响应。

2. 动态调整 RTO 的设计:

- 使用 `timeout` 变量表示超时时间RTO，初始值为 200 毫秒，最大值为 2000 毫秒，最小值为 100 毫秒。
- 如果**发生超时重传**，则会**增加超时时间 `timeout` 的值**，以延长下一次的等待时间，从而降低丢包的可能性。
- 如果**成功接收到期望的数据包**，会**降低超时时间 `timeout` 的值**，以缩短下一次的等待时间，从而加快数据传输的速度。

4. 差错校验

1. `calculateChecksum` 计算校验和:

- 将数据包按照16位分组，每次将两个16位的字节相加，并将结果累加到 `sum` 中
- 接着，将 `sum` 变量的高16位加到低16位上，确保没有进位
- 最后，通过对 `sum` 变量取反得到校验和的值

2. `setChecksum` 设置校验和:

- 将数据包的校验和字段初始化为0，以确保之前的校验和值被清除
- 调用 `calculateChecksum` 函数来重新计算校验和
- 将新的校验和值设置到数据包的校验和字段中

3. `validateChecksum` 验证校验和:

- 验证数据包的校验和是否为0，以确定数据包是否在传输过程中损坏。
- 用 `calculateChecksum` 函数来计算数据包的校验和。
- 校验和与0进行比较，如果相等，说明校验和有效

实验环境

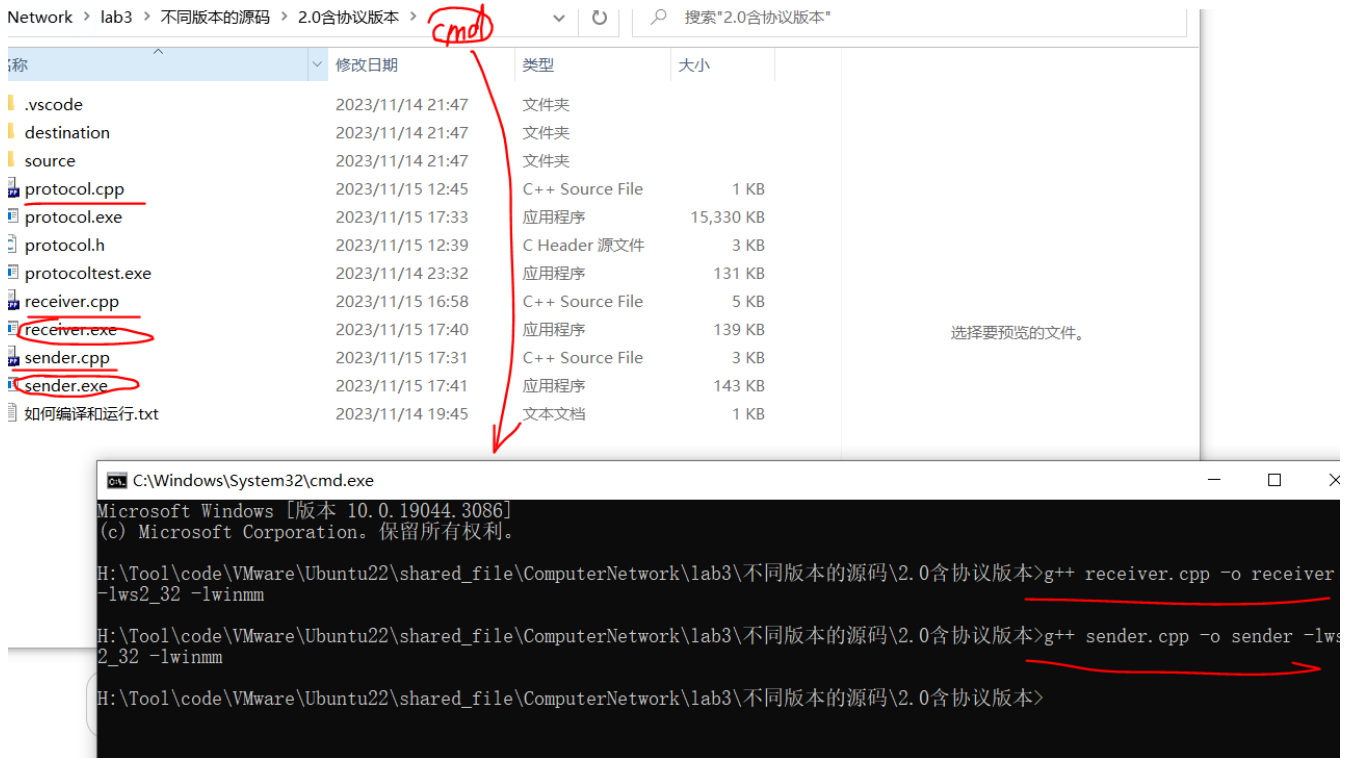
如何编译

前提: windows安装g++并添加到**环境变量**

在含有receiver.cpp、sender.cpp和protocol.cpp的文件夹搜索栏输入cmd打开控制台，输入

```
g++ receiver.cpp -o receiver -lws2_32
g++ sender.cpp -o sender -lws2_32
```

就可以看到生成了sender.exe和receiver.exe



如何传输

1. 在可执行文件的同一目录下，创建两个文件夹 destination 和 source。
2. 将要传输的测试文件放入 source 文件夹。
3. 启动路由器并进行必要的配置。
4. 启动 sender.exe 和 receiver.exe 可执行文件。
5. 在 sender.exe 窗口中输入要传输的文件名。
6. 等待传输完成。
7. 传输完成后，文件将会出现在 destination 文件夹中。

基础实现

最简单的版本，实现了基于UDP协议的基本文件传输功能。具体来说：

- **发送方 (sender.cpp)** 能够让用户输入文件名，从本地 ./source 目录中读取指定文件，并将文件分段发送到接收方。
- **接收方 (receiver.cpp)** 监听指定端口，接收来自发送方的数据包，并将收到的数据写入 ./destination 目录下的文件中。

效果如下

```
H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab3\不同版本的源码\1.0单次发送\receiver.exe
Packet received: 3 bytes
File received and saved as: received_file.txt

H:\Tool\code\VMware\Ubuntu22\shared_file\ComputerNetwork\lab3\不同版本的源码\1.0单次发送\sender.exe
Enter file name: test.txt
Packet sent: 3 bytes
File transfer completed for: test.txt
Enter file name:
```

首先需要定义两方的IP和端口，接收方对应127.0.0.1:61000,然后要和路由器对应上

```
#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 61000
#define CLIENT_PORT 60000
#define BUF_SIZE 8172

int main() {
    WSADATA wsa;
    SOCKET cliSock;
    struct sockaddr_in serverAddr;
    char buffer[BUF_SIZE];

    WSStartup(MAKEWORD(2, 2), &wsa);
    cliSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(SERVER_PORT);
    serverAddr.sin_addr.s_un.S_addr = inet_addr(SERVER_IP);

    while (true) {
        string fileName;
        cout << "Enter file name: ";
        cin >> fileName;
        ifstream file("./source/" + fileName, ios::binary);

        if (!file.is_open()) {
            cerr << "Could not open file: " << fileName << endl;
            continue;
        }

        while (!file.eof()) {
            file.read(buffer, BUF_SIZE);
            int readBytes = file.gcount();
```

Router

路由器IP:	127 . 0 . 0 . 1	服务器IP:	127 . 0 . 0 . 1
端口:	60000	服务器端口:	61000
丢包率:	0 %	延时:	0 ms

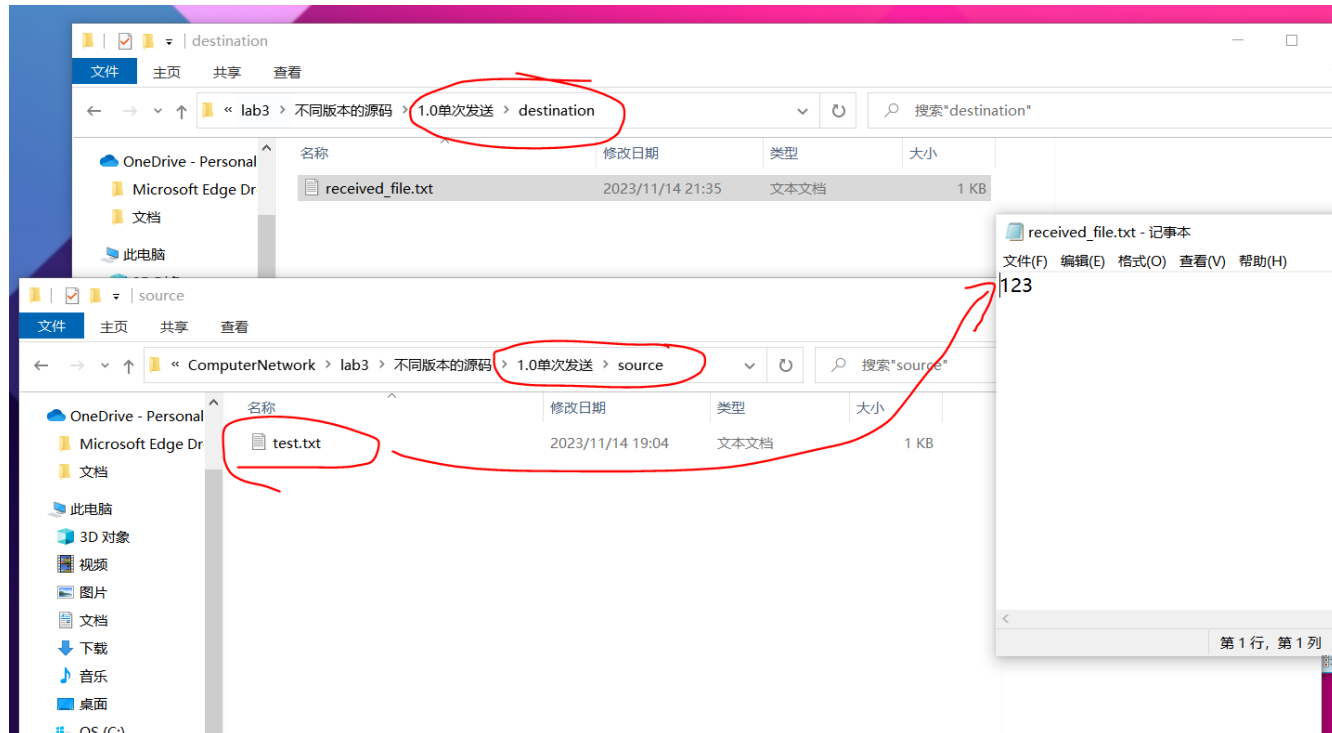
确定 修改

日志

```
Router Ready!
Misscount :0 .
Delay :0 ms .
```

基础版本确实可以输入source文件夹下的文件名，发送后在destination文件夹下面找到。

完整代码已经放在lab3/不同版本源码下面了



协议实现

protocol.h 中定义了以下内容，体现了协议当中的数据部分。并且由于封装了函数，使得代码很简洁。

```
struct Packet {
    uint16_t checksum = 0;
    uint32_t seqNum = 0;
    uint32_t ackNum = 0;
    uint16_t dataLen = 0;
    uint8_t flags = 0;
    uint16_t packetNum = 0;
    char reserved[5] = {0, 0, 0, 0, 0};
    char message[8172];

    // 构造函数
    Packet(uint32_t seq = 0, uint32_t ack = 0, uint16_t len = 0, uint8_t flgs = 0, uint16_t pkt = 0, const char* msg
= "") {
    }

};

enum Flag {
    // 标志位
};

// 计算校验和
uint16_t calculateChecksum(const Packet* packet) {}

// 设置数据包的校验和
void setChecksum(Packet* packet) {}

// 验证校验和
bool validateChecksum(const Packet* packet) {}

// 打印数据包
void printPacket(const Packet& packet, bool isSent, bool showMessage) {}
```

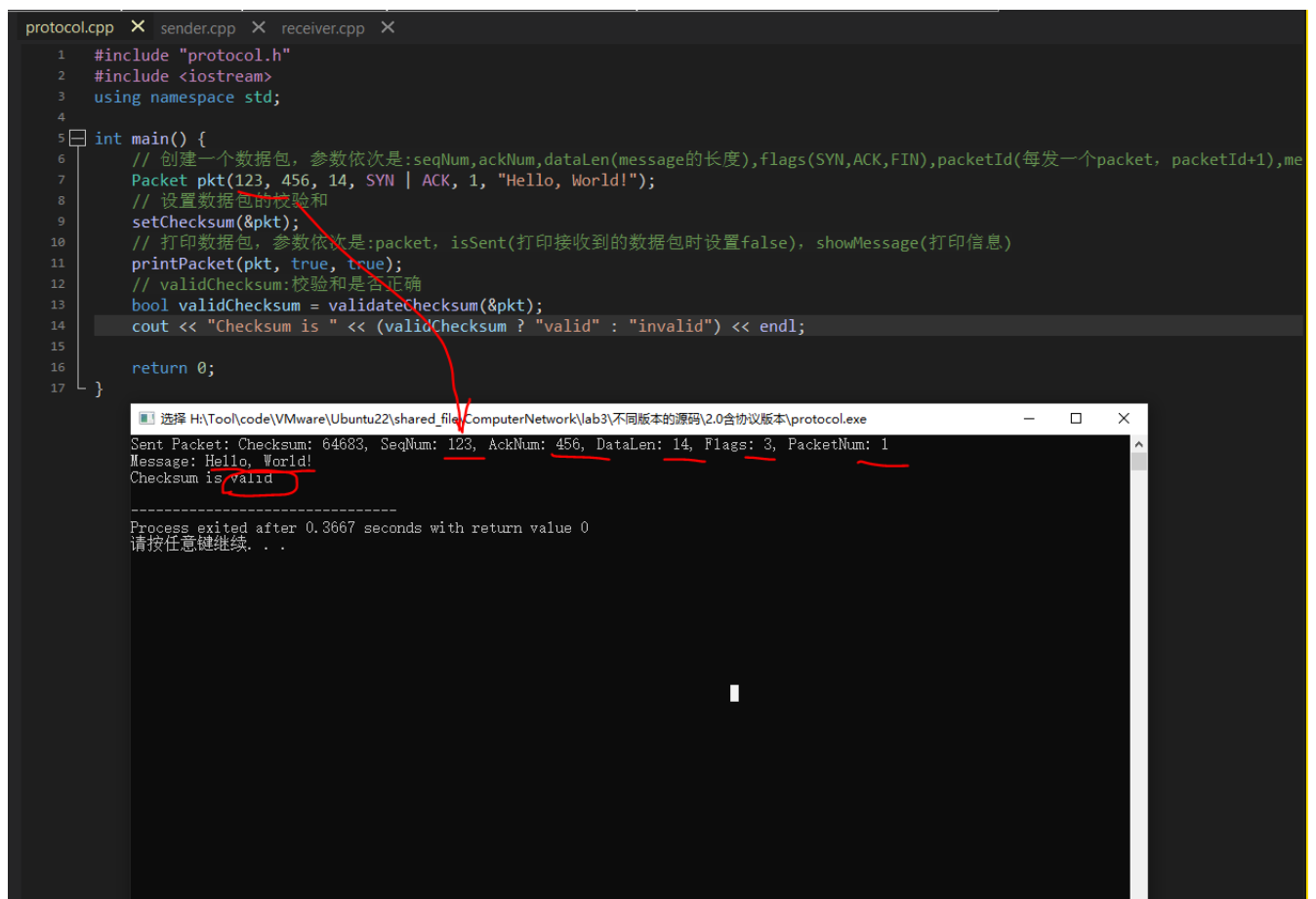

由于协议很重要，我还写了一个测试文件测试协议头文件的正确性：

protocoltest.cpp

```
#include "protocol.h"

int main() {
    // 创建一个数据包头
    PacketHeader header(12345, 54321, 10, SYN | ACK, 1);
    // 打印数据包头
    printPacketHeader(header, true);
    // 计算并设置校验和
    setChecksum(&header);
    // 打印包含校验和的数据包头
    printPacketHeader(header, true);
    // 验证校验和
    bool isValid = validateChecksum(&header);
    if (isValid) {
        std::cout << "Checksum is valid." << std::endl;
    } else {
        std::cout << "Checksum is not valid." << std::endl;
    }
    std::cin.get();
    return 0;
}
```

经过测试，可以正确设置、输出，求校验和，设置校验和



```
protocol.cpp x sender.cpp x receiver.cpp x
1 #include "protocol.h"
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     // 创建一个数据包，参数依次是:seqNum,ackNum,dataLen(message的长度),flags(SYN,ACK,FIN),packetId(每发一个packet, packetId+1),me
7     Packet pkt(123, 456, 14, SYN | ACK, 1, "Hello, World!");
8     // 设置数据包的校验和
9     setChecksum(&pkt);
10    // 打印数据包，参数依次是:packet, isSent(打印接收到的数据包时设置false)，showMessage(打印信息)
11    printPacket(pkt, true, true);
12    // validateChecksum: 校验和是否正确
13    bool validChecksum = validateChecksum(&pkt);
14    cout << "Checksum is " << (validChecksum ? "valid" : "invalid") << endl;
15
16    return 0;
17 }
```

选择 H:\Tool\code\VMware\Ubuntu22\shared_files\ComputerNetwork\lab3\不同版本的源码\2.0含协议版本\protocol.exe

Sent Packet: Checksum: 64683, SeqNum: 123, AckNum: 456, DataLen: 14, Flags: 3, PacketNum: 1
Message: Hello, World!
Checksum is valid

Process exited after 0.3667 seconds with return value 0
请按任意键继续. . .

最终实现

0.建立连接

由于接收方和发送方需要路由器中转,需要设置**路由器IP:127.0.0.1:60000**[数字够大不容易和端口冲突]

服务器IP:127.0.0.1:61000

接收方serverSocket绑定IP:127.0.0.1:61000,发送方clientSocket可以不绑定,向127.0.0.1:60000发送数据即可,接收方接收到数据,将来源地址存入remoteAddr,之后发回数据

```
#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 61000
#define CLIENT_PORT 60000
#define BUF_SIZE 8172

int main() {
    WSADATA wsa;
    SOCKET cliSock;
    struct sockaddr_in serverAddr;
    char buffer[BUF_SIZE];

    WSStartup(MAKEWORD(2, 2), &wsa);
    cliSock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    memset(&serverAddr, 0, sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(SERVER_PORT);
    serverAddr.sin_addr.S_un.S_addr = inet_addr(SERVER_IP);

    while (true) {
        string fileName;
        cout << "Enter file name: ";
        cin >> fileName;
        ifstream file("./source/" + fileName, ios::binary);

        if (!file.is_open()) {
            cerr << "Could not open file: " << fileName << endl;
            continue;
        }

        while (!file.eof()) {
            file.read(buffer, BUF_SIZE);
            int readBytes = file.gcount();
```



1. 超时重传

```
// 设置接收超时
setsockopt(cliSocket, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout, sizeof(timeout));

if (recvResult < 0){
    shouldResend = true;
    cout<<"timeout resending... RTO:"<< timeout <<" increased "<<endl;
    if(timeout < maxTimeout) timeout += 100;
}
```

通过设置套接字选项 SO_RCVTIMEO, recvfrom 函数在指定的超时时间内等待数据的到来。如果超时发生 (即 recvfrom 返回值小于0), 则触发数据包的重传, 并适当增加超时时间 (RTO)。

```

SENDER
Enter filename: 1.jpg
[Sent Packet]: validateChecksum: true, SeqNum: 0,
timeout resending... RTO:200 increased
[Sent Packet]: validateChecksum: true, SeqNum: 0,
[Received Packet]: validateChecksum: true, SeqNum
Message: .
sender confirmed to 1
[Sent Packet]: validateChecksum: true, SeqNum: 1,
[Received Packet]: validateChecksum: true, SeqNum
Message: .

```

如图，展示了发送方超时重传并调整RTO的输出timeout resending...

2. 差错校验

```

if (!validateChecksum(&receivedPacket) || receivedPacket.flags != firstExpectedFlags) {
    // ...
}

```

使用了 validateChecksum 函数来进行差错校验。确保了只有在校验和验证通过的情况下，接收到的数据包才被视为有效。

```
validateChecksum: true,
```

3. 接收确认

```

ACKNum += receivedPacket.dataLen;
cout << "sender confirmed to " << ACKNum << endl;

```

ACKNum全局变量用于跟踪已确认接收的数据量。每次成功接收数据包后，ACKNum 增加接收到的数据长度。

```
sender confirmed to 2
```

4. 动态调整RTO

```

int timeout = 200, maxTimeout = 2000, minTimeout = 100; // (以毫秒为单位)

if(timeout < maxTimeout) timeout += 100;
// ...
if(timeout > minTimeout) timeout -= 100;

```

timeout 变量动态调整。在超时发生时增加超时时间，而在成功接收数据包时减少超时时间。

```

SENDER
Enter filename: test.txt
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
timeout resending... RTO:200 increased
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
timeout resending... RTO:300 increased
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
timeout resending... RTO:400 increased
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
timeout resending... RTO:500 increased
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
timeout resending... RTO:600 increased
[Sent Packet]: validateChecksum: true, SeqNum: 0, AckNum: 0, DataLen: 1, Flags: [SYN]
timeout resending... RTO:700 increased

```

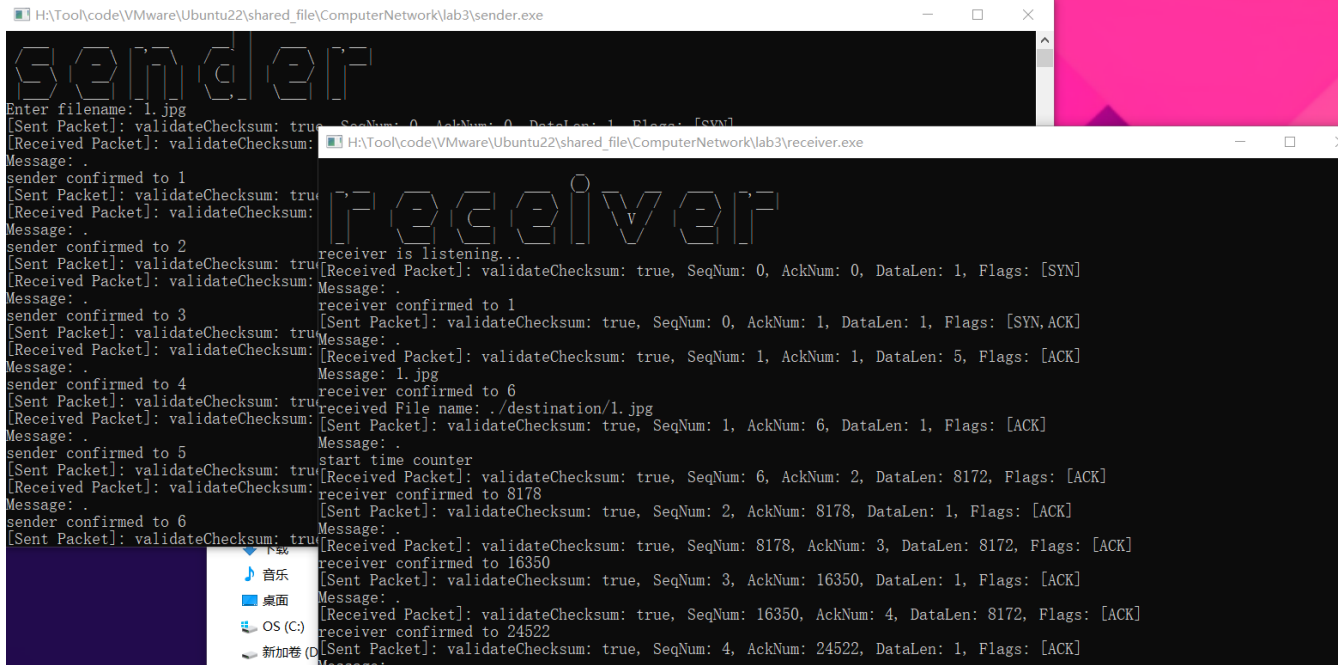
当延时有数秒之高时，RTO可以不断增加，赶上延时

5. 停等机制

```
// First two handshakes
sendAndReceive(sentPacket, SYN | ACK);

// Last two handshakes
sendAndReceive(sentPacket, ACK);
```

发送方在发送一个数据包后等待接收方的确认，然后才继续发送下一个数据包。而`sendAndReceive`封装了发送方要发送的数据包和要接受的字段，非常方便。



如图，接收方每次确认一个小数点，接收方每次确认一段文件

6. 传输时间和平均吞吐率显示

```
auto end_time = chrono::high_resolution_clock::now();
auto duration = chrono::duration_cast<chrono::seconds>(end_time - start_time);
cout<<"time counter ended , File transfer duration:"<< duration.count() <<" s "<<endl;
cout<<"file transfer size : "<<ACKNum<<" B "<<endl;
if(duration.count()!=0)cout<<"file transfer rate : "<<(ACKNum/1024/duration.count())<<" k/s "<<endl;
```

```
Message: .
[Sent Packet]: validateChecksum: true, SeqNum: 230, AckNum: 1857360, DataLen: 1, Flags: [ACK, FIN]
Message: .
[Received Packet]: validateChecksum: true, SeqNum: 1857360, AckNum: 231, DataLen: 1, Flags: [ACK]
Message: .
time counter ended , File transfer duration:6 s
file transfer size : 1857360 B
file transfer rate : 302 k/s
File transfer completed.
receiver is listening...
```

当丢包率为3%，延时为5ms时,对于第一张图片1.jpg的传输结果如下:

```
time counter ended , File transfer duration:6 s
file transfer size : 1857360 B//还有文件名等附加信息
file transfer rate : 302 k/s
```

2.jpg

```
Message: .
time counter ended , File transfer duration:24 s
file transfer size : 5898512 B
file transfer rate : 240 k/s
File transfer completed.
receiver is listening...
```

```
time counter ended , File transfer duration:24 s
file transfer size : 5898512 B
file transfer rate : 240 k/s
```

7.解决最后一次挥手问题

```
setsockopt(serverSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)&timeout, sizeof(timeout));
recvfrom(serverSocket, (char*)&receivedPacket, sizeof(Packet), 0, (struct sockaddr*)&remoteAddr, &remoteAddrSize);
printPacket(receivedPacket, false, true);
setsockopt(serverSocket, SOL_SOCKET, SO_RCVTIMEO, (const char*)&notimeout, sizeof(notimeout));
fclose(outFile);
```

最后一次挥手时，接收方可能收不到，但是接收方已经发出了ACK和FIN ACK，发送方已经关闭，接收方应该关闭，因此计时200ms，时间到了无论有没有收到ACK都关闭接收端。

之后清空状态，重新循环接收，代码实现了接收完之后可以继续输入然后接收

8.RTT大于RTO时正确传输

如图，当延时500ms大于RTO200ms，依然可以正确接收，图片1.jpg依然可以正确显示。

这是由于

1. 发送方每次超时重传都会增大RTO，直到基本匹配。

2. 接收方会接收到多个相同SeqNum，第一次接收使得确认数增加，第二次接收后SeqNum小于期望值，丢弃数据包，不会重复写文件。

The screenshot displays a network simulation environment. On the left, a terminal window shows the output of a sender program (sender.exe) and a receiver program (receiver.exe). The sender's output includes messages like "Message: .", "sender confirmed to 65", and "timeout resending... RTO:700 increased". The receiver's output shows "Message: .", "receiver confirmed to 514842", and "receiver confirmed to 539358". In the center, a "Router" configuration window is visible, showing settings for "路由器IP:" (127.0.0.1), "服务器IP:" (127), "端口:" (60000), "服务器端口:" (61000), "丢包率:" (3%), and "延时:" (500). The bottom part of the terminal shows the receiver's output for subsequent packets, including "receiver confirmed to 547530" and "receiver confirmed to 547530".

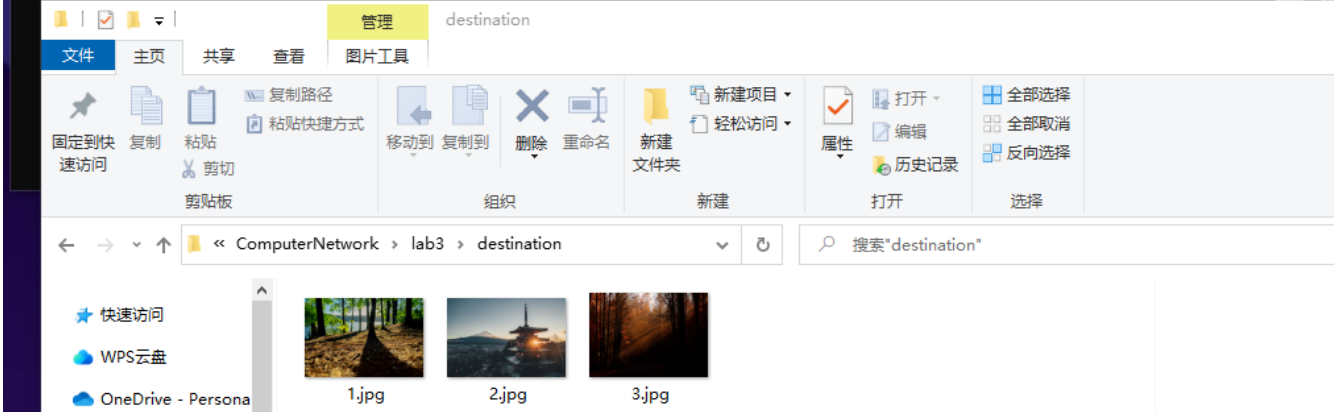
9.接收方ACK回复丢失时正确传输

在答辩的时候,我提到路由器保证了接收方不丢包,实际上,接收方丢包会导致发送方重传,而重传的两个数据包都被接收方收到,后一个将会被丢弃,逻辑和RTT大于RTO时正确传输完全一致

```

H:\Tool\code\VMware
[Sent Packet]: validateChecksum: true, SeqNum: 230, AckNum: 1857360, DataLen: 1, Flags: [ACK]
Message: .
timeout resending... [Sent Packet]: validateChecksum: true, SeqNum: 230, AckNum: 1857360, DataLen: 1, Flags: [ACK,FIN]
[Sent Packet]: validateChecksum: .
[Received Packet]: validateChecksum: true, SeqNum: 1857359, AckNum: 230, DataLen: 1, Flags: [ACK,FIN]
Message: .
sender confirmed to time counter ended , File transfer duration:247 s
[Sent Packet]: validateChecksum: file transfer size : 1857360 B
timeout resending... file transfer rate : 7 k/s
[Sent Packet]: validateChecksum: File transfer completed.
timeout resending... receiver is listening...
[Sent Packet]: validateChecksum: [Received Packet]: validateChecksum: true, SeqNum: 1857359, AckNum: 230, DataLen: 1, Flags: [ACK,FIN]
[Received Packet]: validateChecksum: .
Message: . [Received Packet]: validateChecksum: true, SeqNum: 1857359, AckNum: 230, DataLen: 1, Flags: [ACK,FIN]
[Received Packet]: validateChecksum: .
Message: . [Received Packet]: validateChecksum: true, SeqNum: 1857360, AckNum: 231, DataLen: 1, Flags: [ACK]
sender confirmed to Message: .
[Sent Packet]: validateChecksum: .
File transfer completed
Enter filename:

```



实验中的坑

1. 错误的数据复制方法

最初在 struct Packet 中, 我使用了 strncpy 来复制数据到 message 字段。由于 strncpy 在遇到串尾符 \0 时会停止复制, 对于包含 \0 的二进制数据, 这会导致数据被提前截断。将 strncpy 替换为 memcpy 即可。

2. Packet 结构中的不能有string
3. 字符串数组也不要加串尾符
4. 路由器设置更改可能导致无法传输，重启可以解决

实验总结

在本次UDP协议文件传输系统的实验中，我成功实现并集成了以下关键功能：

1. **超时重传机制**：确保了在数据包丢失或延迟时能够可靠地重新发送数据。
2. **差错校验**：通过校验和机制验证数据包的完整性，提高了数据传输的准确性。
3. **动态调整的重传超时（RTO）**：根据网络状况调整重传间隔，提高了传输效率。
4. **停等协议实现**：确保了每个数据包的发送与确认，增强了通信的可靠性。
5. **数据包确认机制**：通过ACK机制确认数据包的接收，有效跟踪数据传输进度。
6. **四次握手和挥手过程**：实现了连接的建立和断开，符合TCP协议的基本原则。
7. **接收方ACK回复丢失时正确传输**
8. **RTT大于RTO时正确传输**

此实验不仅加深了我对网络通信基本原理的理解，也提升了我在网络编程方面的实践能力。