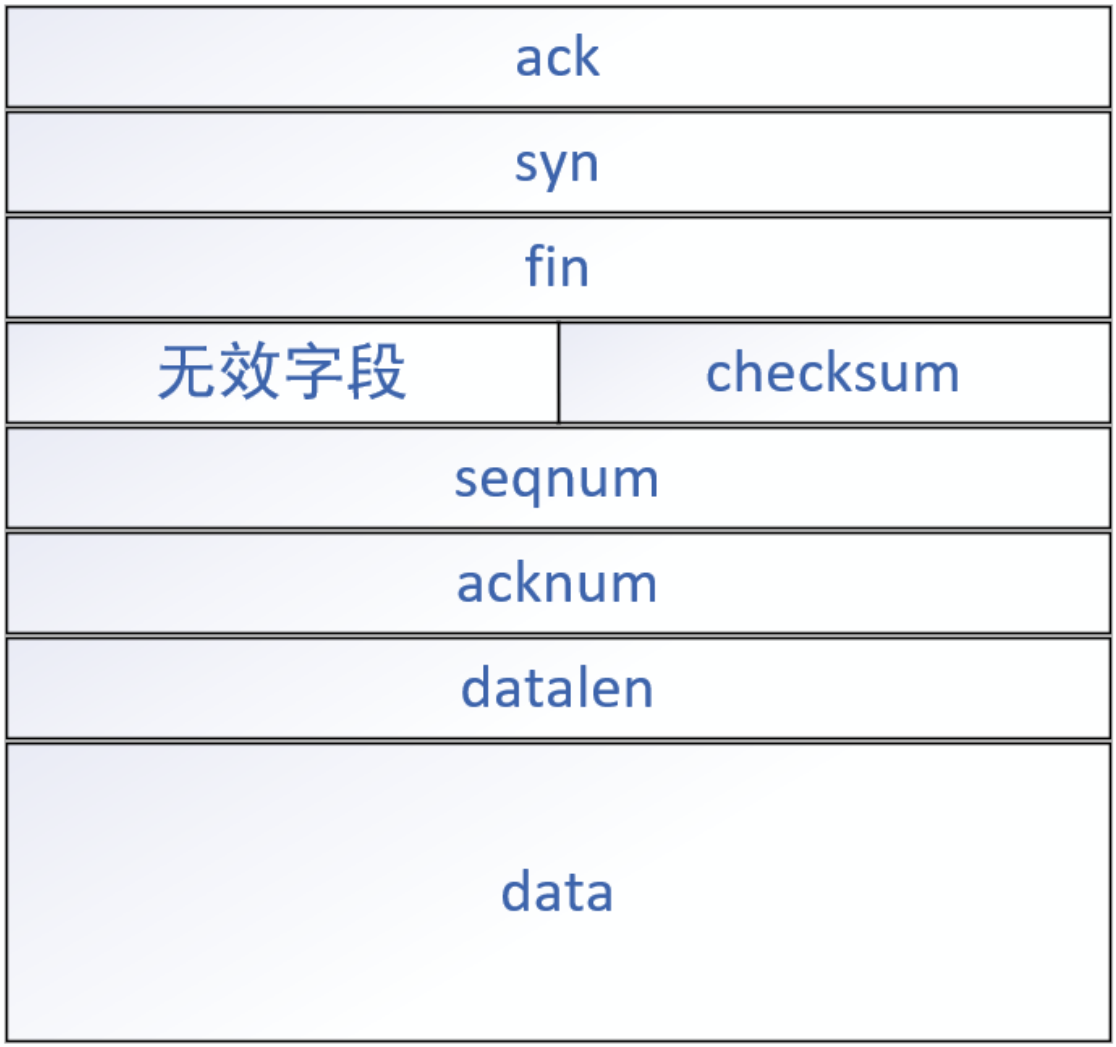
计算机网络实验3-1报告

姓名：阿斯雅 学号：2210737

一、协议设计

报文格式

本次实验中我设计的报文格式参考的TCP报文格式，也就是有`ack`字段，`syn`字段，`fin`字段，`checksum`字段，`acknum`字段，`seqnum`字段，传输数据长度`datalen`和要传输的数据`data`。



具体来说的话就是：

- **ack字段**

  该字段主要是用来服务端来给客户端发送确认信息的，标志着客户端上次发送的信息服务端已正确接收到，可以发送下一次的数据了。

- **syn字段**

该字段主要是用来客户端和服务端刚开始建立连接使用的。客户端主动发送带有syn字段的数据包就意味着客户端和服务端要开始建立连接了。

- **fin字段**

  如同syn字段，fin字段则是用来给客户端和服务端断开连接使用的。客户端主动发送带有fin字段的数据包就意味着客户端将不再给服务端发送消息。

- **checksum字段**

  这个字段是用来给服务端检查客户端发来的消息是否完整。客户端在发送数据包之前要计算出正确的checksum并写到该字段中，服务端接受数据包之后就使用checksum检查数据包是否完整。

- **seqnum字段**

  这个字段主要是客户端用来给服务端说明当前发送的序列号是多少，在服务端接受到消息之后就要根据这个字段来确定是否是自己期待的序列号。

- **acknum字段**

  跟seqnum字段一样，这个字段是服务端用来给客户端发送当前已经确认的序列号是多少，在客户端接受数据包之后根据该字段来进行下一步。

- **datalen字段**

  这个字段是用来告诉对方本次发送的数据的长度。

- **data字段**

  最后这个字段就是承载着数据的字段了。

建立连接

由于本次实验是在 UDP 协议之上实现可靠传输，而 UDP 本身并未提供连接建立的机制，因此我们可以参考 TCP 协议的三次握手过程来实现连接建立。然而，本次实验不同的是，因为服务端事先并不知道客户端计划发送的文件数量，因此在连接建立阶段，不仅需要完成三次握手的过程，客户端还需要向服务端告知本次传输的文件个数，以便服务端做好相应的准备。

具体来说的就是：

第一次握手（SYN）

- **发起方**：客户端。
- 客户端向服务器发送一个带有 `SYN` 标志的 TCP 报文段，请求建立连接。
- 报文段携带客户端的初始序列号（`seq=x`），表示数据传输将从该序列号开始。
- **目的**：通知服务器建立连接的意图，并同步客户端的初始序列号。

第二次握手（SYN-ACK）

- **发起方**：服务器。
- 服务器接收到客户端的 `SYN` 报文后，返回一个同时带有 `SYN` 和 `ACK` 标志的 TCP 报文段。
- 报文段包含：
  - 服务器的初始序列号（`seq=y`）。
  - 对客户端序列号的确认（`ack=x+1`）。
- **目的**：表示服务器同意建立连接，告知其初始序列号，并确认客户端的序列号。

第三次握手（ACK）

- **发起方**：客户端。
- 客户端收到服务器的 `SYN-ACK` 报文后，发送一个仅带有 `ACK` 标志的 TCP 报文段作为确认。
- 报文段包含：
  - 对服务器序列号的确认（`ack=y+1`）。
  - 使用客户端的下一个序列号（`seq=x+1`）。
- **目的**：确认服务器的初始序列号，完成连接的建立。

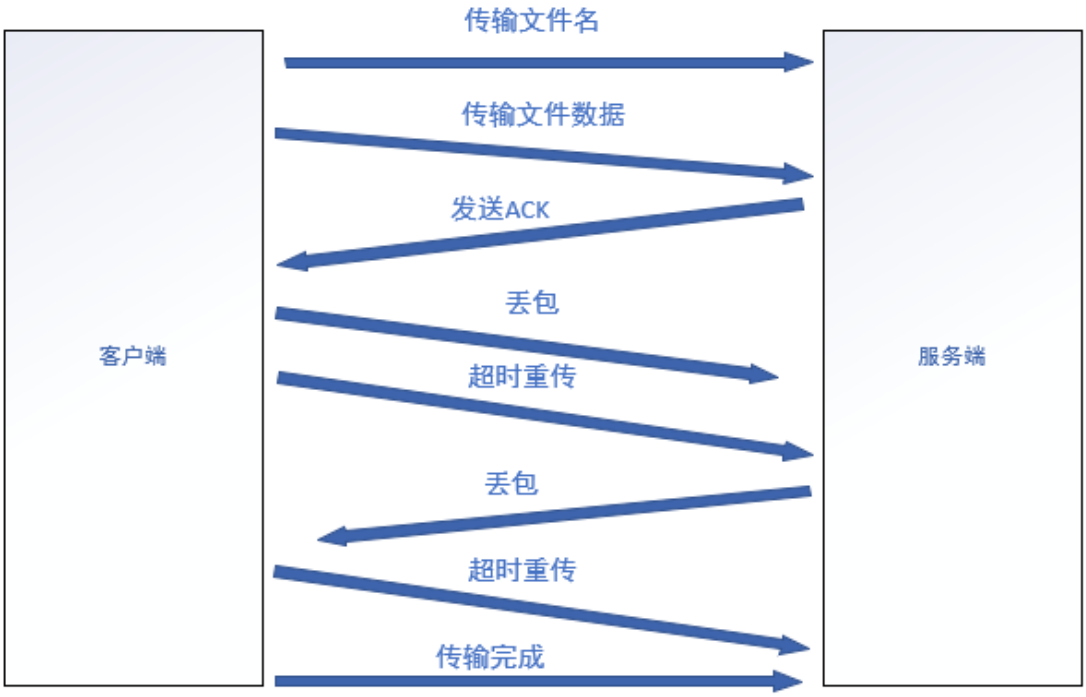
确定文件个数

- **发起方**：客户端。
- 客户端向服务器发送一个数据包，告知传输的文件数量。
- **目的**：双方明确需要传输的文件个数，作为通信的开始，同时标志连接成功。

文件传输

文件传输阶段是我们本次实验的重点。我的文件传输阶段可以分为以下几个部分：

1. 传输文件名
2. 传输文件
3. 超时重传
4. 传输完成



在传输文件数据之前客户端首先要给服务端发送文件名，这是为了在服务端那边也新建一个相同名字的文件。在等到服务端的ACK之后接下来就要传输文件数据。

在正常情况下文件传输的流程为：客户端准备好数据并且设置数据包里的校验字段，然后发送给服务端等待服务端的响应。而服务端收到数据包之后检查数据是否完整，如果完整的话就发送ACK数据包给客户端并等待客户端下一次的数据包。

因为我们本次的实验要做可靠性传输，所以不能只考虑正常情况，也要对其他错误情况给出相应的解决办法。在本次实验中错误情况一共有三种。第一种是客户端丢包，第二种是服务端丢包，第三种则是检验错误。

- 客户端丢包

  客户端丢包意味着服务端并没有接受到客户端发来的数据包，所以也就不会发送ACK数据包。而客户端自己也并不知道自己的数据包已经丢了，它只会一直等待服务端发来的ACK。如果我们不设解决情况的话，文件传输就会进入死锁。所以为了解决这种情况引入了超时重传机制。也就是客户端会等待一段时间，如果超过了设置的阈值的话就重新发送原先的数据包，这样就不会陷入死锁状态。

- 服务端丢包

  不止客户端，服务端也有可能丢包。这时候服务端是并不知道自己的数据包丢了，它只会等待客户端的下一次的数据包，而客户端也一样，会一直等待服务端的ACK数据包。虽然大致情况跟客户端丢包是一样的，但还是会有一些不同的。因为客户端丢包情况下服务端是并没有接到数据包，所以它的acknum是之前的，这时候客户端重传数据包就行。但在服务端丢包情况下服务端已经更新了它的acknum，这时候在客户端不仅要超时重传，在服务端也要设置在收到了不是期待的seqnum时的解决情况，也就是重传之前的ACK数据包。

- 检验错误

  因为我们是在本地完成传输，所以在本次实验中是基本不会出现这种情况的。但还是要考虑这种情况，也就是数据包在传输过程中出现了错误，导致服务端检验的时候发现错误。在这时候服务端应该是不接受这个数据包的，也就是啥都不做，等待客户端超时重传就行。

而在文件全部正确传输完之后客户端要给服务端发送fin数据包表示本次文件传输已经全部完成，开始下一个文件的传输或者断开连接。

断开连接

客户端传输完全部文件之后，就要告诉服务端要断开连接。本次实验中我仿照的是TCP的四次挥手断开连接过程。

第一次挥手（FIN）

- **发起方**：主动关闭连接的一方（如客户端）。
- 发起方发送一个带有 `FIN`（finish）标志的TCP段，表示希望终止连接。
- 此时，发起方进入**FIN-WAIT-1**状态。
- **目的**：通知对方没有更多数据需要发送。

第二次挥手（ACK）

- **发起方**：被动关闭连接的一方（如服务器）。
- 被动关闭方收到 `FIN` 报文后，回复一个带有 `ACK` 标志的TCP段。
- `ACK` 报文确认发起方的 `FIN`。
- 被动关闭方此时进入**CLOSE-WAIT**状态，而主动关闭方进入**FIN-WAIT-2**状态。
- **目的**：确认已收到对方的终止请求，但可能还有数据需要发送。

第三次挥手（FIN）

- **发起方**：被动关闭连接的一方。
- 当被动关闭方完成剩余的数据传输后，发送一个带有 `FIN` 标志的TCP段，表示可以断开连接。
- 被动关闭方进入**LAST-ACK**状态。
- **目的**：通知主动关闭方，已经完成数据发送并准备关闭连接。

第四次挥手（ACK）

- **发起方**：主动关闭连接的一方。
- 主动关闭方收到 `FIN` 报文后，回复一个带有 `ACK` 标志的TCP段，确认对方的 `FIN`。
- 此时，主动关闭方进入**TIME-WAIT**状态，然后进入**CLOSED**状态。
- 被动关闭方收到此 `ACK` 后直接进入**CLOSED**状态。
- **目的**：确认双方已完成连接终止，确保对方收到ACK。

二、程序说明

接下来我将介绍实验程序核心代码。

首先是我的数据包结构体，除了数据字段和检验和字段我都设置成了int类型。并且定义两个全局变量，一个专门用来接受数据，一个专门用来发送数据。

```
//数据包结构体
struct data {
    int ack, syn, fin;
    unsigned short checksum;
    int seqnum, acknum;
    int dataLen;
    char data[SIZE];
} receiveData, sendData;
```

接着是计算检验和的代码。计算检验和的逻辑为首先使字节数为偶数，如果数据的字节数为奇数，在数据末尾填充一个字节的全零（0x00），使其字节数为偶数。然后按块累加所有的16位数据单元，如果累加的结果超出16位，则将溢出的高位加回到低位。

```
// 计算校验和的函数
unsigned short checksum(void* data, int len) {
    unsigned short* ptr = (unsigned short*)data;
    unsigned int sum = 0;
    unsigned short result;

    // 计算校验和的过程
    while (len > 1) {
        sum += *ptr++;
        if (sum & 0x10000) {
            sum = (sum & 0xFFFF) + 1; // 如果有溢出，则将溢出部分加回
        }
        len -= 2;
    }

    // 处理奇数个字节的情况
    if (len) {
        sum += *(unsigned char*)ptr;
        if (sum & 0x10000) {
            sum = (sum & 0xFFFF) + 1;
        }
    }

    result = (unsigned short)~sum; // 反码求和
    return result;
}
```

然后是建立连接过程，对于客户端来说，它主动发起一个带有 `SYN` 标志的数据包，表示希望建立连接。之后，客户端会等待服务端回复一个带有 `SYN+ACK` 标志的数据包，确认服务端已收到连接请求并同意建立连接。接着，客户端发送一个带有 `ACK` 标志的数据包，同时附带本次传输的文件个数，告知服务端传输需求。服务端的逻辑与客户端类似，此处不再赘述。

```
// 握手建立连接
void Handshake_connection() {
    // 第一次握手：客户端发送 SYN 包
    memset(&sendData, 0, sizeof(sendData));
    sendData.syn = 1;
    sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));

    // 第二次握手：接收服务器返回的 SYN+ACK
    memset(&receiveData, 0, sizeof(receiveData));
    recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
(SOCKADDR*)&servAddr, &addrlen);
    if (!(receiveData.ack == 1 && receiveData.syn == 1 && receiveData.acknum ==
1)) {
        cout << "连接错误：未收到正确的 SYN+ACK 包" << endl;
        return;
    }
```

```
    // 第三次握手：客户端发送 ACK
    memset(&sendData, 0, sizeof(sendData));
    sendData.ack = 1;
    sendData.acknum = 1;
    sendData.seqnum = 1;
    nextseqno = 1;
    sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));

    cout << "连接成功！" << endl;
    conect = true;

    // 发送文件数量
    char file_count[50] = "";
    printf("请输入文件个数：\n");
    scanf("%s", file_count);

    memset(&sendData, 0, sizeof(sendData));
    strcpy(sendData.data, file_count);
    sendData.seqnum = nextseqno;
    sendData.dataLen = strlen(file_count);
    sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));

    // 接收服务器确认
    memset(&receiveData, 0, sizeof(receiveData));
    recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
(SOCKADDR*)&servAddr, &addrlen);
    nextseqno++;

    filecount = int(file_count[0] - '0');
}
```

接着介绍文件名传输函数。

首先，客户端通过 `scanf` 函数提示用户输入文件名，并将其存储在 `fileName` 字符数组中。接着，客户端使用 `fopen` 函数尝试以 `rb+` 模式（可读可写的二进制模式）打开用户指定的文件。如果文件打开失败，客户端会输出错误信息 "无法打开文件"，并立即返回，结束当前操作。

接下来，客户端清空了 `sendData` 数据结构，确保之前的数据不会干扰此次的发送。然后，客户端将输入的文件名通过 `strcpy` 复制到 `sendData.data` 中，并计算并存储文件名的长度 `dataLen`，以确保服务端能够正确接收文件名的长度信息。同时，客户端将当前的序列号 `nextseqno` 存储到 `sendData.seqnum` 字段中，以便服务端可以识别数据包的顺序。之后，客户端使用 `sendto` 函数通过 UDP 套接字 `clientSocket` 向服务器的地址 `servAddr` 发送包含文件名的 `sendData` 数据包，数据包大小为 `sizeof(sendData)`。在发送文件名之后，客户端通过 `recvfrom` 函数等待并接收服务器的响应。

此时，客户端将服务器返回的数据存储在 `receiveData` 结构中，并清空该结构以确保没有残留数据。接收到响应后，客户端更新 `nextseqno`（即序列号加1），以便为下一次数据发送做好准备。服务端的逻辑与客户端类似，此处不再赘述。

```
//传输文件名
```

```
void trans_filename() {
    char fileName[50] = "";
    printf("请输入文件名:\n");
    scanf("%s", fileName);

    // 尝试打开文件
    if (!(p = fopen(fileName, "rb+"))) {
        printf("错误：无法打开文件\n");
        return;
    }

    // 清空发送数据结构并设置文件名
    memset(&sendData, 0, sizeof(sendData));
    strcpy(sendData.data, fileName);
    sendData.seqnum = nextseqno; // 设置序列号
    sendData.dataLen = strlen(fileName); // 设置文件名长度

    // 发送文件名
    sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));

    // 接收服务器的响应
    memset(&receiveData, 0, sizeof(receiveData));
    recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
(SOCKADDR*)&servAddr, &addrlen);

    // 更新序列号
    nextseqno++;
}
```

然后是断开连接过程。跟TCP协议基本相同。

```
//挥手断开连接
void wave_disconnect() {
    // 第一次挥手：客户端发送 FIN 包
    memset(&sendData, 0, sizeof(sendData));
    sendData.fin = 1;
    sendData.seqnum = 1; // 客户端当前序列号
    sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));
    cout << "第一次挥手：客户端发送 FIN 包" << endl;

    // 第二次挥手：等待服务器返回 ACK
    memset(&receiveData, 0, sizeof(receiveData));
    recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
(SOCKADDR*)&servAddr, &addrlen);
    if (receiveData.ack == 1) {
        cout << "第二次挥手：收到服务器的 ACK，acknum=" << receiveData.acknum <<
endl;
    }
    else {
        cerr << "连接错误：未收到服务器正确的 ACK 包" << endl;
        return;
    }
```

```
    // 第三次挥手：等待服务器发送 FIN
    memset(&receiveData, 0, sizeof(receiveData));
    recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
(SOCKADDR*)&servAddr, &addrlen);
    if (receiveData.fin == 1) {
        cout << "第三次挥手：收到服务器的 FIN，seqnum=" << receiveData.seqnum <<
endl;
    }
    else {
        cerr << "连接错误：未收到服务器的 FIN 包" << endl;
        return;
    }

    // 第四次挥手：客户端发送 ACK
    memset(&sendData, 0, sizeof(sendData));
    sendData.ack = 1;
    sendData.acknum = receiveData.seqnum + 1; // 对服务器的 FIN 进行确认
    sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));
    cout << "第四次挥手：客户端发送 ACK，acknum=" << sendData.acknum << endl;

    cout << "连接已成功关闭！" << endl;
}
```

上面展示的是客户端和服务端可以通用的代码，接下来解释客户端或服务端独有的函数。

- 客户端整体框架

  首先解释客户端整体框架。在客户端的主函数中在正确建立连接过后是一个循环，用来处理多个文件的传输。而在循环内部首先使用函数trans_filename()来给服务端传输文件名，然后使用C库中有关文件的函数如fseek计算出文件的字节数。接着使用一个while循环来遍历要传输的文件，每一次读取指定大小的字节数到数据包的数据字段并设置好seqnum，datalen等字段，然后使用send_tcp()函数来给服务器传输数据包。send_tcp()函数就是我实现传输数据包的核心函数，在下面会详细介绍。待遍历完文件过后我们就关闭打开的文件句柄，并给服务端发送fin字段表示本次文件传输已经完成。

```
    // 文件传输过程
    for (int i = 0; i < filecount; i++) {
        trans_filename(); // 传输文件名

        // 文件大小处理
        fseek(p, 0, SEEK_END);
        long long fileLen = ftell(p); // 获取文件长度
        fseek(p, 0, SEEK_SET);

        printf("开始传输文件，文件大小为 %lld 字节\n", fileLen);
        sum_byte += fileLen;

        // 传输文件内容
        while (fileLen > 0) {
            memset(&sendData, 0, sizeof(sendData));
            fread(&sendData.data, 1, min(SIZE, fileLen), p);  // 读取文件内容到发送
数据
            sendData.dataLen = min(SIZE, fileLen);  // 数据长度
            sendData.seqnum = nextseqno;  // 设置序列号
```

```
            send_tcp(); // 发送数据并处理超时重传

            nextseqno++;  // 更新序列号
            fileLen -= sendData.dataLen; // 更新剩余文件长度
        }

        // 输出文件传输完成信息
        cout << "第 " << i + 1 << " 个文件传输完成" << endl;

        fclose(p);  // 关闭文件

        // 发送文件结束标志 (FIN)
        memset(&sendData, 0, sizeof(sendData));
        sendData.fin = 1;
        sendData.seqnum = nextseqno;
        sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));
    }
```

- 客户端send_tcp()函数实现

  该函数中实现了数据包传输，超时重传，接收确认等机制。因为我的定时器是使用多线程实现的，所以首先我定义了原子变量和同步工具，用来同步多线程或者保证线程之间不会发生资源竞争。然后我在函数中首先设置原子变量isAckReceived为false，这个变量是用来表示当前数据包有没有收到相应的ACK，而stopTimer变量是用来结束定时器的。接着使用前面的设置检验和的函数给数据包设置检验和，然后使用我自己写的send_data_with_loss_simulation()来模拟丢包并启动定时器线程。然后就是一个永真的循环，用来接受服务端的ACK数据包，如果获得了期望的数据包就把isAckReceived变量设置为false通知定时器线程并跳出while循环。而在函数的最后边要把stopTimer设置为true并通知定时器线程并等待定时器线程结束。

```
// 原子变量和同步工具
atomic<bool> isAckReceived(false);  // 原子变量，标志是否收到ACK
condition_variable cv;              // 条件变量，用于线程间通知和同步
mutex cvMutex;                      // 互斥锁，保护条件变量访问
bool stopTimer = false;             // 定时器线程退出标志
HANDLE mutexHandle;  // Windows互斥量句柄，用于多线程保护
bool isack = false; // ACK标志，用于表示当前是否收到确认包

// 发送数据并启动定时器
void send_tcp() {
    isAckReceived.store(false); // 初始化 ACK 标志
    setChecksum(&sendData); // 设置校验和
    send_data_with_loss_simulation(); // 模拟数据发送（包含丢包情况）
    stopTimer = false;
    thread timerThread(timerThreadFunc); // 启动定时器线程

    while (true) {
        // 接收来自服务器的 ACK 数据
        recvfrom(clientSocket, (char*)&receiveData, sizeof(receiveData), 0,
(SOCKADDR*)&servAddr, &addrlen);

        if (receiveData.ack == 1 && receiveData.acknum == sendData.seqnum +
1) {
            cout << "收到 ACK，确认号 acknum=" << receiveData.acknum << endl;
            isAckReceived.store(true); // 更新 ACK 状态
```

```
            cv.notify_one(); // 通知定时器线程
            break;
        }
    }
    // 停止定时器线程
    stopTimer = true;
    cv.notify_one();  // 唤醒被阻塞的定时器线程
    timerThread.join(); // 等待定时器线程结束
}
```

- 客户端定时器实现

  在该函数内部使用了一个 `unique_lock` 对象 `lock` 来锁定一个互斥量 `cvMutex`，确保多线程的安全。然后在一个无限循环中，函数使用条件变量 `cv` 等待一个信号，这个信号就是我们定义的原子变量isAckReceived或者是一个超时事件。`wait_for` 函数设置了超时时间为 `TIMEOUT` 毫秒，并检查是否收到了 ACK。如果 `isAckReceived` 标志被设置为 `true`，则表示收到了 ACK，函数会跳出循环并结束。如果没有收到 ACK 而是超时了，那么会执行超时的处理逻辑：打印一条消息，表明需要重新发送数据，并显示当前的序列号 `seqnum`。然后，使用 `sendto` 函数将数据 `sendData` 发送到客户端套接字 `clientSocket`。这个循环会一直进行，直到 `stopTimer` 标志被设置为 `true` 或在超时范围内收到了ACK数据包，这时循环会结束，定时器线程会停止运行。

```
// 定时器线程函数
void timerThreadFunc() {
    unique_lock<mutex> lock(cvMutex);
    while (!stopTimer) {
        // 等待 ACK 或超时
        if (cv.wait_for(lock, chrono::milliseconds(TIMEOUT), [] { return
isAckReceived.load(); })) {
            break; // 收到 ACK，退出定时器
        }
        // 超时，重新发送数据
        cout << "超时，重新发送数据，序列号 seqnum=" << sendData.seqnum << endl;
        sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));
    }
}
```

- 客户端丢包函数

  丢包函数实现的逻辑很简单，就是设置一个随机值，然后判断是否小于我们设定的值，如果小于了就啥都不做，也就是丢包。如果大于，就正常发送数据包即可。

```
// 模拟丢包的发送函数
void send_data_with_loss_simulation() {
    // 模拟丢包，根据丢包概率决定是否丢包
    float randomValue = (float)rand() / RAND_MAX;
    if (randomValue < LOSS_PROBABILITY) {
        cout << "模拟丢包：seqnum=" << sendData.seqnum << endl;
        return; // 丢包，不发送
    }
    // 如果没有丢包，则正常发送数据
    sendto(clientSocket, (char*)&sendData, sizeof(sendData), 0,
(SOCKADDR*)&servAddr, sizeof(SOCKADDR));
    cout << "发送数据包：seqnum=" << sendData.seqnum << endl;
}
```

- 服务端整体框架

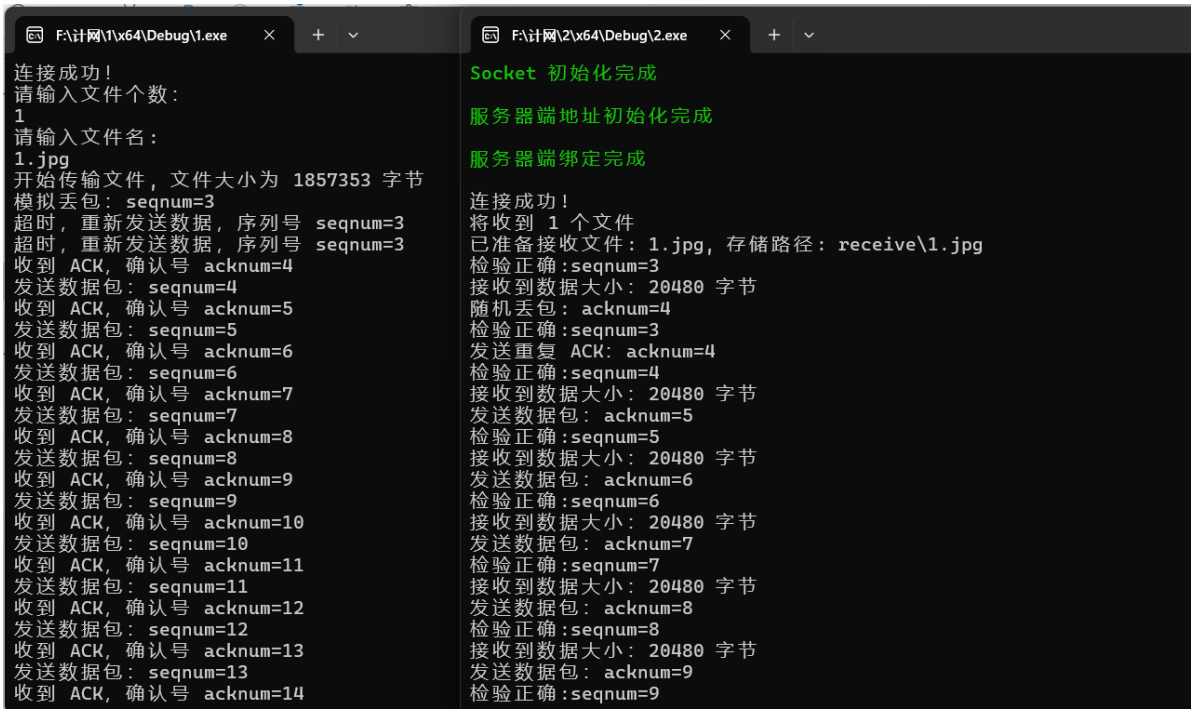
  服务端没有太多函数，所以在此仅解释整体框架。首先客户端相同，外面是一个for循环，用来接收多个文件，然后使用 receive_filename()函数来接受客户端发来的文件名信息，并在自己的目录下打开一个文件句柄，后面使用这个文件句柄就可以写入数据了。接着是一个永真循环，循环里面不断接受客户端的数据包，判断fin字段是否为1，如果为1，就标志着客户端本次的文件传输已经完成，就要跳出循环。如果不为1，首先使用检验和字段检查数据包是否完整，如果完整接着判断是否是服务端期望的数据包，如果是期望的数据包，就接受数据包并写入到文件中并给客户端发送ACK数据包。在本次实验中我也模拟了服务端的丢包，所以跟客户端相同使用send_data_with_loss_simulation()函数传输数据包。如果不是自己期望的数据包说明收到了重复的数据包。我们可以判断为什么收到了重复的数据包，这是因为服务端丢包了，然后客户端没收到ack超时重传了，但这时候服务端已经把相应的数据包写进了文件中，所以并不需要再次写入，所以直接发送重复的ack数据包就行。然后如果检验失败的话服务端已经是什么都不干，直接等待客户端超时重传即可。

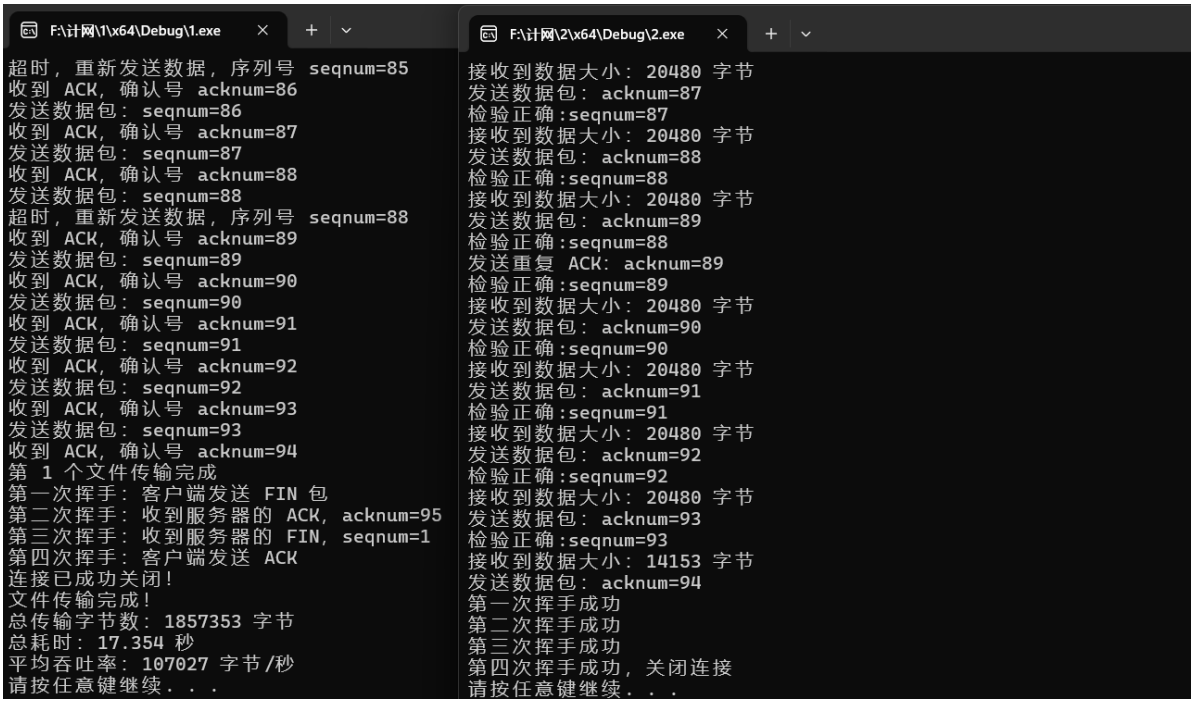
```
// 文件接收循环
    for (int i = 0; i < filecount; i++) {
        receive_filename(); // 接收文件名
        while (1) {
            // 清空接收数据结构并接收数据包
            memset(&receiveData, 0, sizeof(receiveData));
            recvfrom(serverSocket, (char*)&receiveData, datalen, 0,
                (SOCKADDR*)&sourceAddr, &addrlen);

            // 如果接收到结束标志（FIN），关闭文件并退出循环
            if (receiveData.fin == 1) {
                fclose(p);
                break;
            }

            // 检查接收数据是否有效
            if (checkReceivedData(receiveData)) {
                // 如果接收到期望的序列号
                if (receiveData.seqnum == nextAckNum) {
                    nextAckNum++; // 更新期望序列号
                    cout << "接收到数据大小：" << receiveData.dataLen << " 字节"
<< endl;

                    // 将数据写入文件
                    fwrite(receiveData.data, receiveData.dataLen, 1, p);
```

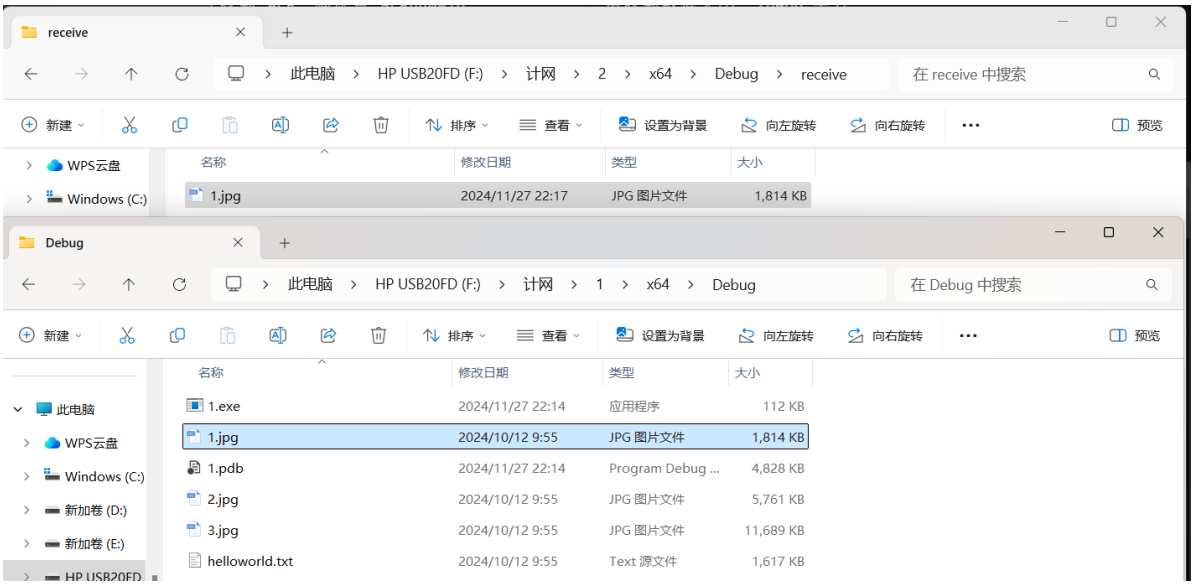
```
                    // 准备 ACK 数据包
                    memset(&sendData, 0, sizeof(sendData));
                    sendData.seqnum = 1;
                    sendData.ack = 1;
                    sendData.acknum = nextAckNum;

                    // 发送 ACK，模拟丢包
                    send_data_with_loss_simulation();
                }
                else {
                    // 如果收到重复数据包，重新发送 ACK
                    sendto(serverSocket, (char*)&sendData, datalen, 0,
                        (SOCKADDR*)&sourceAddr, addrlen);
                    cout << "发送重复 ACK：acknum=" << sendData.acknum <<
endl;
                }
            }
        }
    }
```

三、实验结果

在这里以1.jpg为例展示我的实验结果。

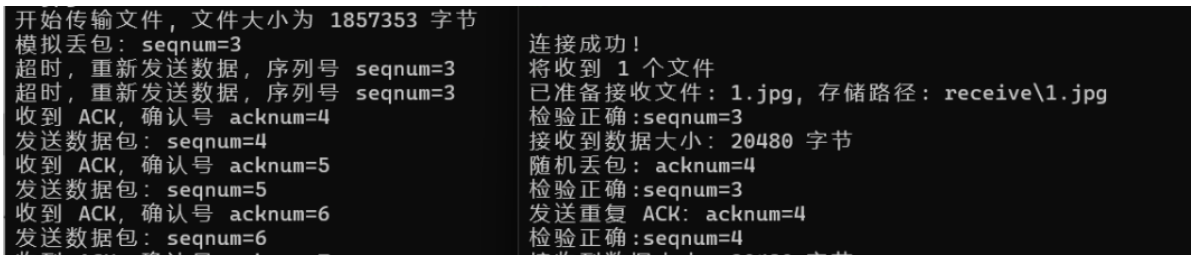






首先客户端和服务端进行握手连接，然后输入要传输的文件个数：1，输入要传输的文件名：1.jpg，然后就开始传数据。

我们可以看到客户端第一个数据包就丢包了，然后出现了第一个超时重传，接着服务端接收到了重传的3号数据包，但可以看到服务端的ACK包丢包了，这时候就出现了客户端的第二个超时重传。这时候服务端是收到了重复的数据包，就发送之前的ACK包就行。后面所有的超时重传基本都是这个逻辑。



然后可以看到当传输完成之后在服务端的相应文件夹下面已经有了1.jpg这个文件，并且和原来的文件大小是相同的。

四、实验总结

实验难点

在我的本次3-1实验中遇到的最大难点就是如何设置定时器，因为我选用的方法是通过将定时器实现为一个独立的多线程模块。定时器线程需要根据主线程的ACK信息来判断是否停止计时或执行超时重传操作，这使得两个线程之间的同步与通信问题显得尤为重要。在实验初期，我尝试直接使用一些简单的标志位进行线程间通信，但发现这种方法难以保证线程的原子性操作，容易导致竞争条件的问题，从而使程序行为变得不可预测。

通过查阅资料和实验后，我了解到互斥量是一种有效的机制，可以很好地解决多线程间的同步与竞争问题。互斥量通过确保同一时刻只有一个线程能够访问共享资源，从而避免数据不一致的问题。虽然引入互斥量会增加一定的系统开销，例如需要更多的内存资源和操作系统资源，但它提供了可靠的同步保障。在实现过程中，我将ACK标志的访问操作加锁，确保主线程和定时器线程在读取或修改这一关键变量时不会产生冲突。

此外，我还意识到定时器线程的响应性和准确性在实际传输中的重要性。为此，我对定时器的时间间隔、重试次数等参数进行了多次调整，以平衡资源开销与可靠性之间的关系。虽然在调试过程中遇到了线程死锁和响应延迟等问题，但通过进一步优化代码逻辑，例如在适当的地方释放互斥锁以及合理设计线程退出条件，最终实现了一个稳定高效的定时器模块。这次实验让我对多线程编程和同步机制有了更深刻的理解。

心得体会

通过本次3-1实验，我在UDP协议的基础上实现了超时重传、建立连接、断开连接以及接收确认等可靠传输机制，这让我对可靠传输的概念从单纯的理论理解转变为亲身实践中的具体实现。在这个过程中，我认识到网络协议设计的复杂性和细节处理的重要性。

首先，在实现超时重传机制时，我体会到了如何利用定时器和多线程来协调数据的发送与确认。这让我意识到，可靠传输的核心不仅在于发送方的数据完整性，还在于接收方的反馈机制是否及时准确。这种双向交互的设计对保证数据完整性至关重要。通过动手实践，我深入理解了TCP协议的核心思想，例如如何处理丢包、延迟以及重复确认的情况，并将这些理念运用到实验中。

其次，通过实现建立连接和断开连接的功能，我对三次握手和四次挥手的过程有了更加具体的认识。从初始化连接的同步标志到断开连接的资源释放，我发现每一步都必须严格遵循通信双方的约定，否则可能导致连接失败或资源泄漏。这让我体会到协议的设计必须具备精确性和鲁棒性，以应对复杂的网络环境。

此外，在实验中处理接收确认的过程中，我认识到网络编程的挑战之一在于如何高效地管理状态。为了处理超时重传，我需要追踪发送数据包的状态、确认ACK的时序以及计数重试次数，这让我深刻体会到状态管理和资源分配对程序稳定性的影响。

最后，这次实验让我更加理解“可靠性”的真正含义，并不仅仅是实现功能上的完整性，而是要做到在各种异常情况下都能保证数据的正确传输。这种能力需要建立在对底层通信原理的深刻理解之上，同时也需要在实践中不断优化代码结构和逻辑设计。

通过这次实验，我不仅提升了自己在网络编程方面的能力，也更加明白了实践的重要性。只有通过动手实现理论知识，才能发现隐藏的细节问题并不断提高自己的编程能力和问题解决能力。