哈爾濱工業大學
Harbin Institute of Technology

计算机网络
课程实验报告

实验名称	可靠数据传输协议的设计与实现				
姓名	张智雄		院系	计算学部	
班级	2103601		学号	2021112845	
任课教师	刘亚维		指导教师	刘亚维	
实验地点	格物 207		实验时间	2023.10.28	
实验课表现	出勤、表现得分(10)		实验报告得分(40)		实验总分	
	操作结果得分(50)				
教师评语					

哈尔滨工业大学计算学部
FACULTY OF COMPUTING, HIT

实验目的：

1. 理解**可靠数据传输**的基本原理；掌握停等协议的工作原理；掌握基于 UDP 设计并实现一个停等协议的过程与技术。
2. 理解**滑动窗口**协议的基本原理；掌握 GBN 的工作原理；掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术。

实验内容：

1. 基于 UDP 设计一个简单的停等和 GBN 协议，实现**单向**可靠数据传输（服务器到客户的数据传输）；
2. 模拟引入**数据包的丢失**，验证所设计协议的有效性；
3. 基于所设计的停等协议，实现一个 **C/S 结构**的文件传输应用。
4. 改进所设计的 GBN 协议，支持**双向**数据传输；
5. 将所设计的 GBN 协议**改进为 SR 协议**。

实验过程：

一、 GBN协议可靠数据传输

当设置服务器端发送窗口的大小为 1 时， GBN 协议就是停-等协议。

1.服务器端Server

使用 UDP 协议传输数据（比如传输一个文件），等待客户端的请求，接收并处理来自客户端的消息（如数据传输请求）。

在服务端运行之后，首先会初始化套接字并绑定端口地址监听（**非阻塞接收**），监听到客户端发送的命令执行函数，当接收到“-time”或者“-quit”报文时，则执行返回时间并打包发给客户端和退出程序。

```
while (true) {
    for (int i = 0; i < SEQ_SIZE; ++i) {
        ack[i] = TRUE;
    }
    int recvNum = 0;
    int iMode = 1; //1:  非阻塞,  0:  阻塞
    ioctlsocket(sockServer, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置
    //非阻塞接收,  若没有收到数据,  返回值为-1
    recvSize = recvfrom(sockServer, buffer, BUFFER_LENGTH, 0, ((SOCKADDR*)&addrClient), &length);
    if (recvSize < 0) {
        Sleep(200);
        continue;
    }
    printf("recv from client: %s\n", buffer);
    if (strcmp(buffer, "-time") == 0) {
        getCurTime(buffer);
    }

    else if (strcmp(buffer, "-quit") == 0) {
        strcpy_s(buffer, strlen("Good bye!") + 1, "Good bye!");
    }
```

当客户端开始请求数据，即服务器端接收到“-testgbn”报文的话，就会进入握手阶段，建立连接状态后（并不是真正的连接，只是一种类似连接的数据发送的状态），将数据打包成数据报文发送，然后等待客户端的 ACK 信息，同时启动计时器。

握手阶段：首先服务器端（服务器端处于 0 状态）向客户端发送 205 状态码（服务器端进入 1 状态），而后服务器端调用 recvfrom()等待客户端回复 200 状态码。

如果收到（服务器端进入 2 状态），初始化现在的序列号 curSeq，现在已确认报文序列号 curAck，计时器 waitCount，开始传输文件；否则延时等待直至超时，超时则放弃此次“连接”，等待从第一步开始。

```
int stage = 0;
bool runFlag = true;
while (runFlag) {
    switch (stage) {
    case 0://发送 205 阶段
        buffer[0] = 205;
        sendto(sockServer, buffer, strlen(buffer) + 1, 0, (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
        Sleep(100);
        stage = 1;
        break;
    case 1:
        //等待接收 200 阶段， 没有收到则计数器+1， 超时则放弃此次“连接”， 等待从第一步开始
        recvSize = recvfrom(sockServer, buffer, BUFFER_LENGTH, 0, ((SOCKADDR*)&addrClient), &length);
        if (recvSize < 0) {
            ++waitCount;
            if (waitCount > 20) {
                runFlag = false;
                printf("Timeout error\n");
                break;
            }
            Sleep(500);
            continue;
        }
        else {
            if ((unsigned char)buffer[0] == 200) {
                printf("Begin a file transfer\n");
                printf("File size is %dB, each packet is 1024B and packet total num is % d\n", sizeof(data), totalPacket);
                curSeq = 0;
                curAck = 0;
                totalSeq = 0;
                waitCount = 0;
                stage = 2;
            }
        }
        break;
```

传输阶段：服务器端首先将数据打包成数据报文，调用函数 seqIsAvailable()函数判断序列号是否在当前发送窗口之内，然后按数据帧格式将数据包发送给客户端。发完之后 curSeq 加 1 并对 SEQ_SIZE 取模得到新的 curSeq。

```
bool seqIsAvailable() {
    int step;
    step = curSeq - curAck;
    step = step >= 0 ? step : step + SEQ_SIZE;
    //序列号是否在当前发送窗口之内
    if (step >= SEND_WIND_SIZE) {
        return false;
    }
    if (ack[curSeq]) {
        return true;
    }
    return false;
}
```

而后调用 recvfrom()非阻塞等待接受 ACK。当收到 ACK 时，调用 ackHandler()函数进行窗口滑动（**累积确认**），正常发送下一个数据报，计时器重新计时；若在计时器超时前没有收到 ACK，则调用超时重传函数 timeoutHandler()**全部重传窗口内**的所有已发送的数据报。

计时器的实现方式为：设置套接字为非阻塞模式，服务器端在 recvfrom()方法上不会阻塞。如果正确接收到 ACK 消息，那么计时器将被清零。但如果从客户端接收到数据长度为-1，这表示没有接收到任何数据，那么计时器将增加。对计时器进行判断，如果计时器超过了阈值，就被认为发生了超时，此时会执行超时重传操作。

```
    if (seqIsAvailable() && curSeq < totalPacket) {
        //发送给客户端的序列号从 1 开始
        buffer[0] = curSeq + 1;
        ack[curSeq] = FALSE;
        memcpy(&buffer[1], data + 1024 * curSeq, 1024);
        printf("send a packet with a seq of %d\n", curSeq);
        sendto(sockServer, buffer, BUFFER_LENGTH, 0, (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
        ++curSeq;
```

```
//等待 Ack, 若没有收到, 则返回值为 - 1, 计数器 + 1
recvSize = recvfrom(sockServer, buffer, BUFFER_LENGTH, 0, ((SOCKADDR*)&addrClient), &length);
if (recvSize < 0) {
    waitCount++;
    //10 次等待 ack 则超时重传
    if (waitCount > 10)
    {
        timeoutHandler();
        waitCount = 0;
    }
}
else {
    //收到 ack
    ackHandler(buffer[0]);
    waitCount = 0;
    if (buffer[1] >= totalPacket) {
        runFlag = false;
        printf(" | 数据传输完成 | \n");
        buffer[0] = 245;
        sendto(sockServer, buffer, BUFFER_LENGTH, 0, (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
    }
}
Sleep(500);
break;
```

最后当序列号 curSeq 等于数据发送的总包数 totalPacket 时停止发送新的报文，而当收到的 ACK 等于总包数 totalPacket 时传输结束，并向客户端发送状态码同步传输结束。

2.客户端Client

使用 UDP 协议向服务器端请求数据，接收服务器端发送的数据报并返回确认信息 ACK（注意 GBN 为**累积确认**，即客户端返回当前接受**有序最新**数据包的确认 ACK）。

客户端首先初始化套接字并绑定端口地址监听（**阻塞接收**），使用命令行输入指令，当输入为“-time”或者“-quit”则直接作为数据包发送给服务端（查询时间以及退出）。

```
if (!strcmp(cmd, "-time") || !strcmp(cmd, "-quit")) {
    sendto(socketClient, buffer, strlen(buffer) + 1, 0, (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
    ret = recvfrom(socketClient, buffer, BUFFER_LENGTH, 0, (SOCKADDR*)&addrServer, &len);
    printf("%s\n", buffer);
    if (!strcmp(buffer, "Good bye!"))
        break;
}
```

如果输入为“-testgbn_recv [X][Y]”，则进入等待握手阶段（客户端处于 0 状态），当收到来自服务器的 205 状态码后，向服务器发送 200 状态码，同时初始化已收到的序列号 recvSeq 为 0 和等待收到的序列号 waitSeq 为 1，进入等待传输阶段（1 状态）

```
case 0:
    //等待握手阶段
    u_code = (unsigned char)buffer[0];
    if ((unsigned char)buffer[0] == 205) {
        printf("Ready for file transmission\n");
        buffer[0] = 200;
        buffer[1] = '\0';
        sendto(socketClient, buffer, 2, 0, (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
        stage = 1;
        recvSeq = 0;
        waitSeq = 1;
    }
    break;
```

而后调用 recvfrom()等待来自服务器的数据报文，如果是期待的包，正确接收，正常确认即可；否则就**丢弃此数据包**，返回上一个正确接收的包的序列号 recvSeq 的 ACK。如果当前一个包都没有收到，则等待 Seq 为 1 的数据包，不是则不返回 ACK（因为并没有上一个正确的 ACK）。

```
//等待接收数据阶段
seq = (unsigned short)buffer[0];
//随机法模拟包是否丢失
b = lossInLossRatio(packetLossRatio);
if (b) {
    printf("The packet with a seq of %d loss\n", seq);
    continue;
}
printf("recv a packet with a seq of %d\n", seq);
//如果是期待的包， 正确接收， 正常确认即可
if (!(waitSeq - seq)) {
    ++waitSeq;
    if (waitSeq == 21) {
        waitSeq = 1;
    }
    memcpy(&RecvMessage[recvNum * 1024], &buffer[1], strlen(buffer) - 1);
    ++recvNum;
    buffer[0] = seq;
    buffer[1] = recvNum;
    buffer[2] = '\0';
    recvSeq = seq;
}
else {
    //如果当前一个包都没有收到， 则等待 Seq 为 1 的数据包， 不是则不返回 ACK（因为并没有上一个正确的 ACK）
    if (!recvSeq) {
        continue;
    }
    buffer[0] = recvSeq;
    buffer[1] = recvNum;
    buffer[2] = '\0';
}
b = lossInLossRatio(ackLossRatio);
if (b) {
    printf("The ack of %d loss\n", (unsigned char)buffer[0]);
    continue;
}
sendto(socketClient, buffer, 3, 0, (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
printf("send a ack of %d\n", (unsigned char)buffer[0]);
```

最后当接收到服务器 245 状态码时（表示传输完毕）结束接收，等待下一条用户指令的输入。同时清除接收缓存 RecvMessage 和报文缓存 buffer，避免下次传输出错。

```
if ((unsigned char)buffer[0] == 245) {
    printf("| 数据接收成功，接受的数据为 |\n");
    printf("%s\n", RecvMessage);
    FILE* f;
    f = fopen("output.txt", "w");
    fwrite(RecvMessage, sizeof(char), strlen(RecvMessage), f);
    fclose(f);
    printf("| 文件已写入 |\n");
    ZeroMemory(RecvMessage, sizeof(RecvMessage));
    ZeroMemory(buffer, sizeof(buffer));
    break;
}
```

二、模拟数据包丢失

在客户端中引入默认包丢失率 packetLossRatio 和默认 ACK 丢失率 ackLossRatio 来模拟引入数据包的丢失和 ACK 丢失的情况。

模拟的思路是：引入 b 作为随机变量，其值对应 lossInLossRatio()函数的返回值，当接受到报文和发送 ACK 之前均对 b 进行判断，如若为 TRUE，则跳过当前接受数据循环，直接进入下一次循环，在阻塞态下，即为接受下一个数据包；反之则正常进行数据接受和反馈。

如若报文丢失，recvSeq 和 waitSeq 不会变化；ACK 丢失，recvSeq 和 waitSeq 正常移动加一。但两种情况下，后续服务器会进行超时重传处理。

lossInLossRatio()函数根据丢失率随机生成一个数字，判断是否丢失，如若丢失则返回 TRUE，否则返回 FALSE。

```
BOOL lossInLossRatio(float lossRatio) {
    int lossBound = (int)(lossRatio * 100);
    int r = rand() % 101;
    if (r <= lossBound) {
        return TRUE;
    }
    return FALSE;
```

三、基于C/S结构的文件传输

为实现基于 C/S 结构的文件传输，**服务端**首先要将测试文件 test.txt 内容读入拷贝到缓存，计算出要发送的总包数 totalPacket，而后将序号组的 ACK 设为 TURE（表明该分组还没发送成功即没有收到相应 ACK）。

```
//将测试数据读入内存
std::ifstream icin;
icin.open("test.txt");
char data[1024 * 113];
ZeroMemory(data, sizeof(data));
icin.read(data, 1024 * 113);
icin.close();
//printf("%s", data);
totalPacket = (int)ceil((double)strlen(data) / 1024);
```

客户端在收到服务端发送的数据包之后，如果是期望收到的，则将收到的数据缓存到 recvMessage 中。在接收到 245 状态码结束传输之后写入文本文件 output.txt。

```
if ((unsigned char)buffer[0] == 245) {
    printf("| 数据接收成功，接受的数据为 |\n");
    printf("%s\n", RecvMessage);
    FILE* f;
    f = fopen("output.txt", "w");
    fwrite(RecvMessage, sizeof(char), strlen(RecvMessage), f);
    fclose(f);
    printf("| 文件已写入 |\n");
    ZeroMemory(RecvMessage, sizeof(RecvMessage));
    ZeroMemory(buffer, sizeof(buffer));
    break;
}
```

四、双向数据传输

为实现双向数据传输，在客户端界面上增加一个选项为“-testgbn_send”，输入后客户端会向服务器端发送“-receive”报文，而后将扮演服务器的角色项服务器端发送数据。经历握手、数据传输等阶段。

```
void printTips() {
    printf("*****************************************\n");
    printf("| -time to get current time |\n");
    printf("| -quit to exit client |\n");
    printf("| -testgbn_recv [X] [Y] to test the gbn |\n");
    printf("| -testgbn_send [X] [Y] to test the gbn |\n");
    printf("*****************************************\n");
}
```

客户端收到“-receive”报文后，则进入接收信息的状态，经历等待握手、等待数据传输、数据传输完毕的各个阶段。

总体上看，就是合并了客户端和服务端的代码。需要注意的即是**不用换用两个套接字，但是需要在调换角色的同时更改套接字的阻塞状态**（即作为服务器端需要非阻塞态，客户端采用阻塞态）。

五、SR协议的改进

1.服务器端Server

SR 协议的服务端和 GBN 协议的服务端的最主要区别在于，GBN 只有当收到对应当前发送窗口最左侧 send_base 数据包的 ACK 时才会处理并进行窗口滑动。

而 SR 可以收到并处理对应发送窗口（send_base 到 send_base+SEQ_SIZE）内所有数据包的 ACK，而当收到对应当前发送窗口最左侧 send_base 数据包的 ACK 时进行窗口滑动直至第一个未确认接收 ACK 的位置。

在接受 ACK 阶段，如果发生了丢包或者其他错误，程序会遍历期望收到的 ACK，查看哪些 ACK 是还没有收到（即没有被确认），然后对应分组 i 的计时器++，超过 8 次则重新传送分组。

```
//等待 Ack，若没有收到，则返回值为-1，计数器+1
recvSize = recvfrom(sockServer, buffer, BUFFER_LENGTH, 0, ((SOCKADDR*)&addrClient), &length);
if (recvSize < 0) {
    int curseq;
    if (send_base > curSeq) curseq = send_base + curSeq;
    else curseq = curSeq;
    for (int i = send_base; i <= min(curseq - 1, totalPacket - 1); i++)
    {
        int number = i % SEQ_SIZE;
        if (!ack[number]) {
            waitCount[number]++;
            //3 次等待 ack 则超时重传该分组
            if (waitCount[number] > 8) {
                printf("Packet%d ack Time out\n", number);
                sendPacket(number, sockServer);
                waitCount[number] = 0;
            }
        }

    }
}
```

而如果收到 ACK，则会对第 i 个分组的 ACK 位置标志设为 TRUE（在读入数据时将所有 ACK 设为 FALSE 以区分是否接收），同时窗口基址 send_base 向右移动到当前未确认的第一个位置。

```
void ackHandler(char c) {
    unsigned char index = (unsigned char)c;
    printf("Recv a ack of %d\n", index);
    ack[index] = true;
    waitCount[index] = 0;
    // 窗口起始位置移到当前未确认的第一个位置
    while (ack[send_base]) {
        ack[send_base] = 0;
        send_base++;
        //窗口回到最左侧
        if (send_base >= SEQ_SIZE) {
            send_base = 0;
        }
    }
}
```

而在结束传输上，与 GBN 类似，当发送包序列号超过总包数 totalPacket 时就拒绝发送，而当发送窗口移动到 totalPacket 右侧时即传输结束。

2.客户端Client

SR 协议的客户端和 GBN 协议的客户端的区别在于 GBN 只能接受当前等待接收的数据包（**接收一定有序**）同时接受窗口向右移动，否则就丢弃。

而 SR 客户端可以接受在接收窗口内任意的数据包（**接收不一定有序**），同样只有当收到对应当前发送窗口最左侧 recv_base 数据包时才会进行窗口滑动，滑动到当前未接收的第一个序列位置。

具体实现上，同样设置一个接收序列数组 recvack[]，如果收到预期消息，接收窗口右移；否则就将接受位置对应标志设为 TRUE，缓存到 recvMessage 对应的位置中。同样采用默认包丢失率 packetLossRatio 和默认 ACK 丢失率 ackLossRatio 来模拟引入数据包的丢失和 ACK 丢失的情况。

```
//如果收到预期消息，接收窗口右移
if (recv_base == seq) {
    recvack[seq] = false;
    recv_base++;
    for (int i = recv_base; i < recv_base + SEQ_WINDOW_SIZE; i++) {
        int m = i % SEQ_SIZE;
        if (recvack[m]) {
            recvack[m] = false;
            recv_base++;
        }
    }
    recv_base %= SEQ_SIZE;
}

else {
    recvack[seq] = true;
}
recv_base %= SEQ_SIZE;
```

3.窗口大小

滑动窗口协议中发送/接收窗口大小的设置：设序列号位数为n，发送窗口大小N_{send}，接收窗口为N_{recv}，则满足$N_{send}+N_{recv}\leq 2^n$。SR 协议中一般$N_{send}=N_{recv}$，原因是发送窗口大于接收窗口会导致溢出，小于则没有意义。

本实验中 SR 协议窗口大小均设置为 8。

实验结果：

1. 单向可靠数据传输—GBN协议

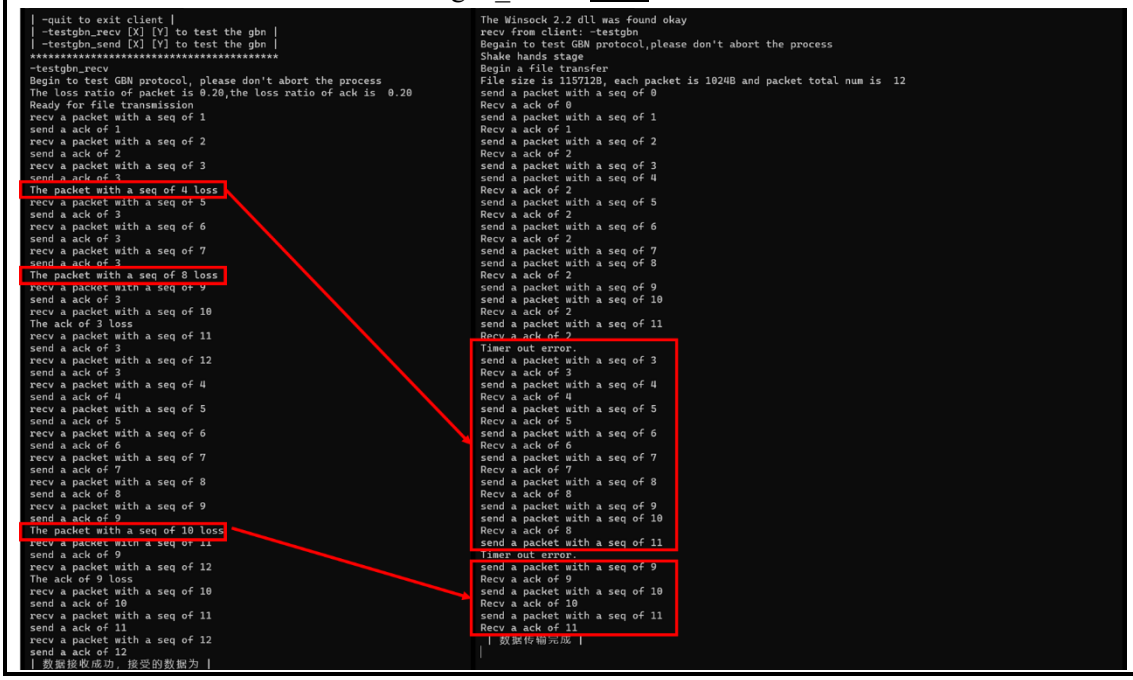
同时运行客户端和服务器端代码，初始化套接字并对端口 12340 进行监听。

a) 查询时间测试连接

客户端向服务器端发送“-time”查询当前时间，返回结果如下：

b) 数据传输

客户端向服务器端发送“-testgbn_recv”**请求**数据，建立连接后，传输结果如下：



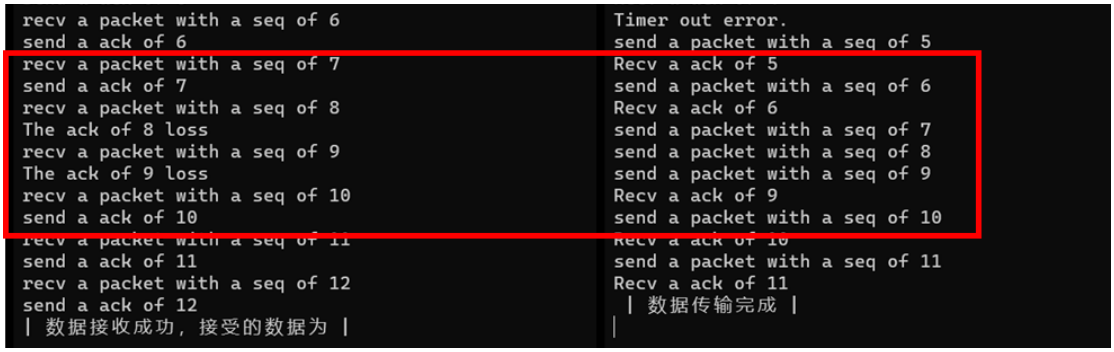
可以看到数据发送过程中：

1) 序列号为 4 和 8 的数据包丢失了，在此过程中，客户端成功接收到了数据包 5-7 和 9-12，但由于 GBN 协议不缓存乱序到达的数据包，这些数据包被直接丢弃，并且持续返回 send a ack of 3。

2) 随后，服务器端触发了超时重传机制，重新发送窗口内所有数据（也就是数据包 4-12）。此时，客户端重新接收了数据 4-9，并进行了缓存，并发送了 ACK4-9。这个过程发送和接收窗口正常移动。

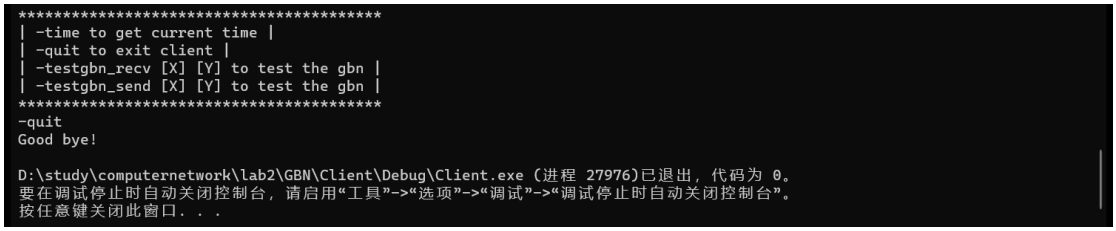
3) 此时，数据包 10 再次丢失并再次触发超时，导致重发数据包 10-12。这个重传过程会一直持续，直到接收方正确收到了 ACK 12（ACK 的序号达到 TotalSeqNum）。

同时，由于 GBN 采用累计确认机制，客户端发送的 ACK 相当于确认此序号之前的所有数据都已正确接收，因此 ACK 丢失一般不影响数据传输（除非 ACK 连续多次丢失）。当 ACK 的序号达到 TotalSeqNum 时，表示已经成功完成了全部数据传输，连接随之解除。



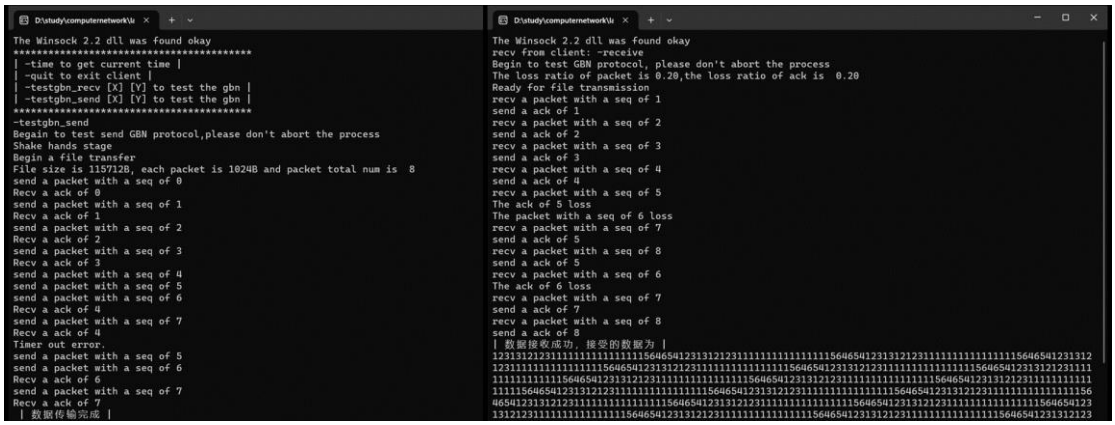
c) 退出传输

客户端向服务器端发送“-quit”退出程序，返回结果如下：



2. 双向可靠数据传输—GBN协议

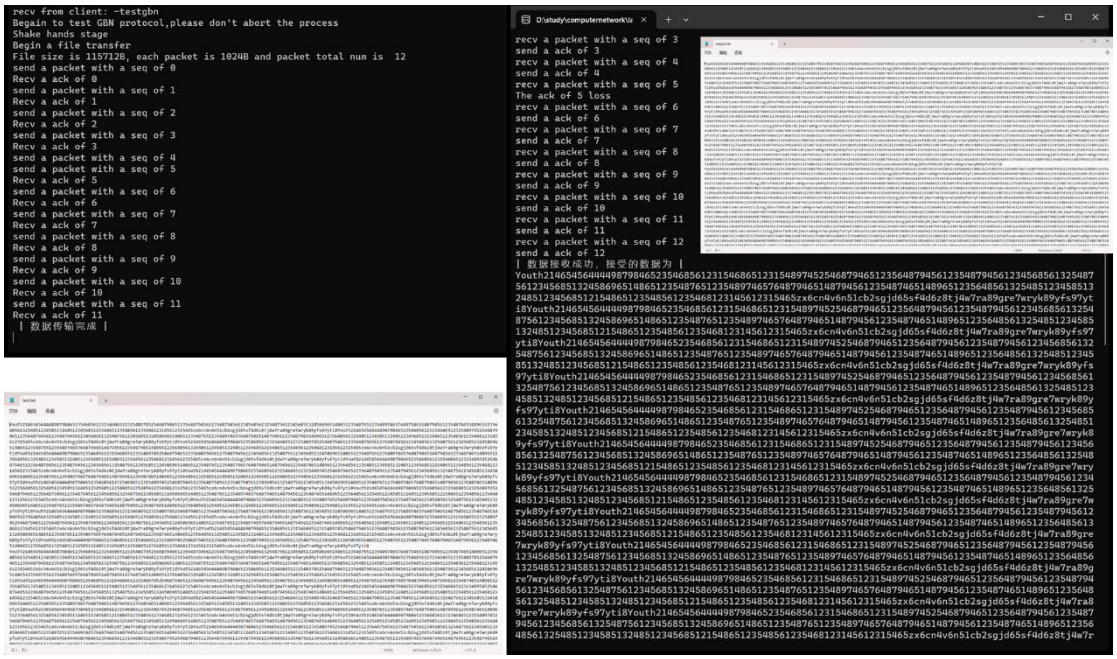
客户端向服务器端发送“-testgbn_send”上传数据，建立连接后，传输结果如下：



超时重传、数据和 ACK 的丢失与上述相同，此处不再分析。

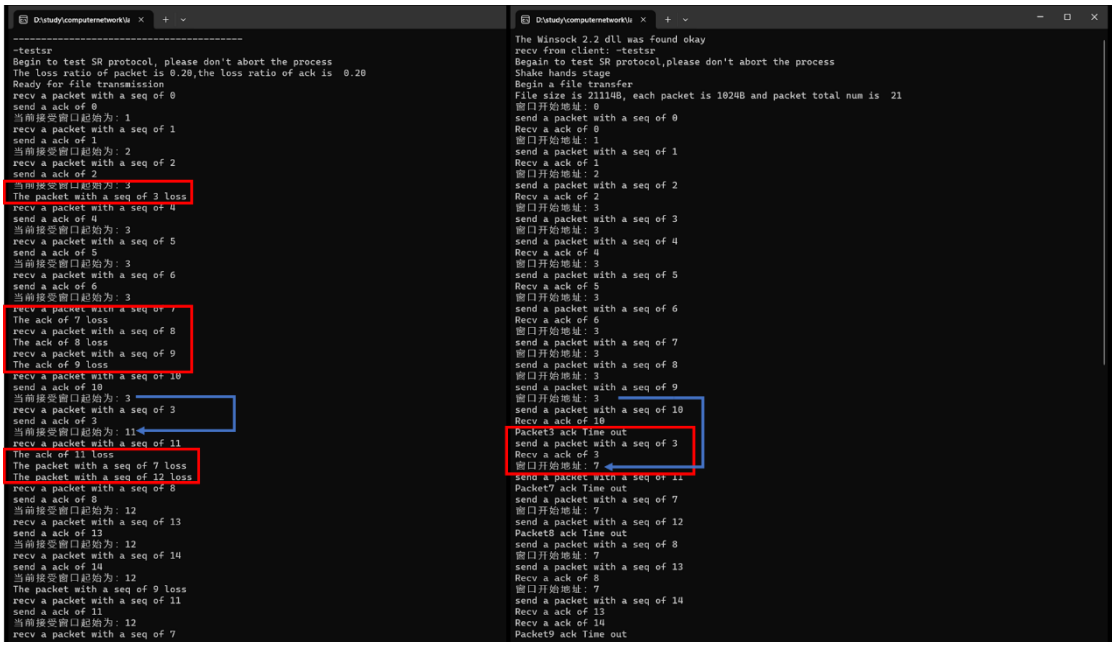
3. 基于C/S结构的文件传输

上述传输建立在文件读取的基础之上，传输结束后会将内容写入本地文件。即服务器端读入文件 test.txt 并传输，客户端正确接收并写入为 output.txt。



4. 可靠数据传输—SR协议

同时运行客户端和服务器端代码，初始化套接字并对端口 12340 进行监听。客户端向服务器端发送“-testsr”请求数据，建立连接后，传输部分结果如下：



由于原理相同，报告中只对截取部分进行分析。可以看到数据发送过程中：

1) 序列号为 3 的数据包丢失了，在此过程中，客户端成功接收到了数据包 4-10，同时正常返回对应序列号的 ACK（并不像 GBN 一样直接舍弃）。

2) 随后，服务器端触发了超时重传机制，重新发送数据包 3。此时数据包 4-10 已被接收，但只有数据包 4-6 已确认，因而接收窗口滑动至 11 位置，而发送窗口滑动至 7 位置。

3) 此时，数据包 7、8、9 会再次触发超时重传。这个重传和窗口滑动过程会一直持续，直到发送窗口移到 21 右侧（即所有数据包均已确认收到）。

同时，由于 SR 不采用累计确认机制，客户端发送的 ACK 仅仅确认此序号数据被正确接收，因此数据包和 ACK 丢失都会导致数据传输。**但当 ACK 丢失时，接收窗口正常滑动，而发送窗口则会停留在该位置直至被确认。**

问题讨论：

停等协议是 GBN 协议的特殊情况，数据分组格式、确认分组格式、程序流程图等基本一致因而报告主要针对 GBN 协议作讨论（**停等协议只需将窗口的大小设置为 1**）

1. GBN（停等）协议数据分组格式、确认分组格式、各个域的作用

以太网中，数据帧的 MTU 为 1500 字节，所以 UDP 数据报的数据部分应小于 1472 字节（除去 IP 头部 20 字节与 UDP 头的 8 字节），为此，定义 **UDP 数据报的数据部分**格式为：

Seq	Data	0

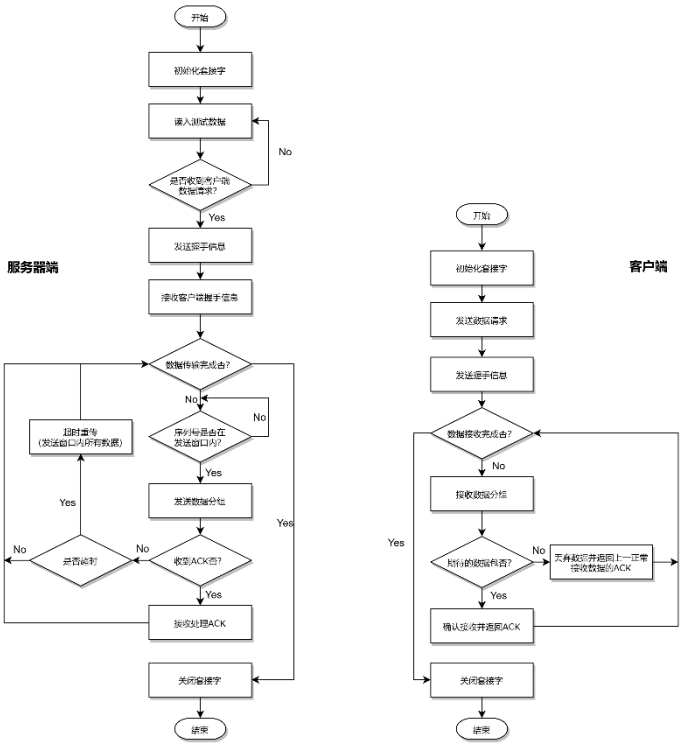
其中，Seq为 1 个字节，取值为0~255，（故序列号最多为256个）；Data ≤ 1024个字节，为传输的数据；最后一个字节放入EOF0，表示结尾。

ACK 数据帧定义：

ACK	0

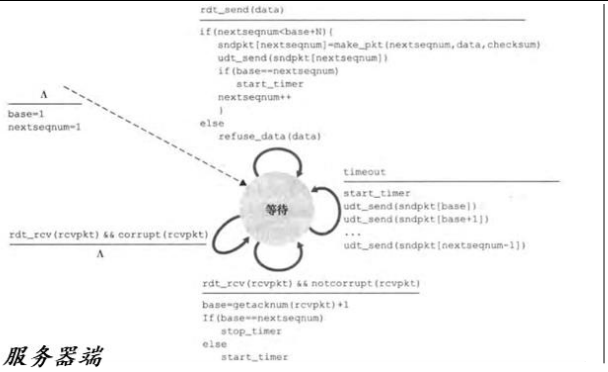
由于是从服务器端到客户端的单向数据传输，因此 ACK 数据帧不包含任何数据，只需要将 ACK 发送给服务器端即可。ACK 字段为一个字节，表示序列号数值；末尾放入 0，表示数据结束。

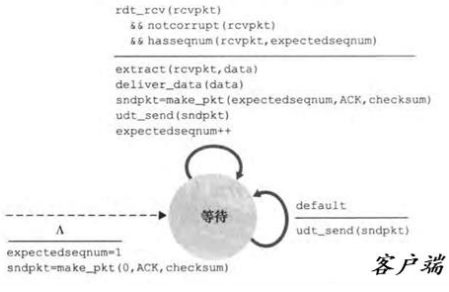
2. GBN（停等）协议两端程序流程图



3. GBN（停等）协议典型交互过程

客户端（Client）	服务器端（Server）
a) 根据目标服务器IP地址与端口号创建套接字，并连接服务器；	a) 对到来的请求创建套接字，绑定套接字的IP地址和端口号，对端口进行监听；
b) 向服务端发送请求，然后等待服务端反馈；	b) 收到客户请求消息后，读取请求数据的内容，构造数据包，向客户端进行发送数据报同时开启计时器；
c) 接收到服务端数据报后进行判断是否为当前期望的数据报，是则接收并返回此数据的ACK确认，否则丢弃，返回上一确认的数据对应的ACK。	c) 接受客户端返回的ACK确认数据报文，根据序列号判断是否滑动窗口发送帧，同时根据计时器对未确认数据段执行超时重传操作。





4. 数据分组丢失验证模拟方法

模拟的思路是：引入 b 作为随机变量，其值对应 lossInLossRatio()函数的返回值，当接受到报文和发送 ACK 之前均对 b 进行判断，如若为 TRUE，则跳过当前接受数据循环，直接进入下一次循环，在阻塞态下，即为接受下一个数据包；反之则正常进行数据接受和反馈。

如若报文丢失，recvSeq 和 waitSeq 不会变化；ACK 丢失，recvSeq 和 waitSeq 正常移动加一。但两种情况下，后续服务器会进行超时重传处理。

lossInLossRatio()函数根据丢失率随机生成一个数字，判断是否丢失，如若丢失则返回 TRUE，否则返回 FALSE。具体代码见附件或实验过程部分。

5. 程序实现的主要类（或函数）及其主要作用

函数	主要作用
void timeoutHandler()	超时重传处理函数，滑动窗口内的数据帧都要重传
void ackHandler(char c)	收到累积确认ACK，取数据帧的第一个字节
bool seqIsAvailable()	检查当前序列号curSeq是否在发送窗口内
void printTips()	打印提示信息
BOOL lossInLossRatio(float lossRatio)	随机模拟分组丢失
void getCurTime(char* ptime)	获取当前系统时间

6. UDP编程主要特点

1) UDP 是无连接的，发送数据之前不需要建立连接，因此减少了开销和发送数据之前的时延。

2) UDP 使用尽最大努力交付，不保证可靠交付，即可能丢失和乱序。需要引入可靠数据传输手段来确保数据传输正确。

3) UDP 是面向报文的，UDP 对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界，一次交付一个完整的报文。

7. 实验验证结果和详细注释源程序
   见上文实验结果部分和附件。

心得体会：

1. 本次对可靠数据传输协议的实现，更加深入的理解了滑动窗口协议，对GBN协议和SR协议有了更深的理解；
2. 对Socket编程有了进一步的学习和理解。
3. 对UDP协议有了进一步的了解，同时也更加理解了从UDP到TCP的一步步的探索过程，对运输层有了更为深刻的理解。