南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验 3-1

基于 UDP 服务设计可靠传输协议并编程实现

杨浩甫　2113824

年级：2021 级

专业：计算机科学与技术

指导教师：张建忠、徐敬东

2023 年 11 月 17 日

目录

一、 实验要求

利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、接收确认、超时重传等。流量控制采用停等机制，完成给定测试文件的传输。

1. 实现单向数据传输（一端发数据，一端返回确认）。
2. 对于每个任务要求给出详细的协议设计。
3. 完成给定测试文件的传输，显示传输时间和平均吞吐率。
4. 性能测试指标：吞吐率、延时，给出图形结果并进行分析。
5. 完成详细的实验报告（每个任务完成一份，主要包含自己的协议设计、实现方法、遇到的问题、实验结果，不要抄写太多的背景知识）。
6. 编写的程序应该结构清晰，具有较好的可读性。
7. 提交程序源码、可执行文件和实验报告。

二、 报文格式

在本次实验中，我们的报文仿照 TCP 数据报的格式进行设计。其中包含了确认号，序列号，标志位，长度，校验和以及数据段。同时使用了 #pragma pack(1) 取消结构体成员之间的内存对齐，确保结构体的所有成员都紧密地连续存储在内存中，没有额外的填充字节。

报文设计

```
#pragma pack(1)
    u_long flag{};
    u_short seq{};//序列号
    u_short ack{};//确认号
    u_long len{};//数据部分长度
    u_long num{}; //发送的消息包含几个包
    u_short checksum{};//校验和
    char data[1024]{};//数据长度
#pragma pack()
    message() {
        memset(this, 0, sizeof(message));
    }
    bool isSYN() {
        return this->flag & 1;
    }
    bool isFIN() {
        return this->flag & 2;
    }
    bool isSTART() {
        return this->flag & 4;
    }
    bool isEND() {
        return this->flag & 8;
```

```
    }
    bool isACK() {
        return this->flag & 16;
    }
    bool isEXT() {
        return this->flag & 32;
    }
    bool isRE() {
        return this->flag & 64;
    }
    void setSYN() {
        this->flag |= 1;
    }
    void setFIN() {
        this->flag |= 2;
    }
    void setSTART() {
        this->flag |= 4;
    }
    void setEND() {
        this->flag |= 8;
    }
    void setACK() {
        this->flag |= 16;
    }
    void setEXT() {
        this->flag |= 32;
    }
    void setRE() {
        this->flag |= 64;
    }
```


我们可以得到报文的整体结构:

字段	位置 (比特位)	大小 (比特位)
flag	0-31	32
seq	32-47	16
ack	48-63	16
len	64-95	32
num	96-127	32
checksum	128-143	16
data	144-1175	8192

表 1: 报文结构

由于 flag 有 32 个比特位，所以我可以设置 32 个标识符，本实验暂时只用到 6 个标识符:

- SYN: 用于初始化一个连接。当一个端点希望建立连接时，它会发送一个带有 SYN 标志的

数据包。

- FIN：指示一个端点要关闭连接。当一个端点发送一个带有 FIN 标志的数据包时，它表明不再有数据要发送，但允许接收方发送数据。
- START：可能指示传输开始的标志。
- END：可能指示传输结束的标志。
- ACK：用于确认接收到的数据包。
- EXT：扩展位，用于检查是否为空消息。

32 - 8	7	6	5	4	3	2	1
	RE	EXT	ACK	END	START	FIN	SYN

表 2: 标志位设计

三、 建立与关闭连接

我的设计旨在通过 UDP 协议实现类似于 TCP 的连接建立和断开机制。虽然 UDP 是一个无连接的、不保证可靠传输的协议，但我可以通过额外的确认消息和状态检查来确保可靠性。

(一) 建立连接的设计（三次握手）

1. 第一次握手：客户端发送 syn 包。客户端将标志位 SYN 置为 1，另 seq=0，并将该数据包发送给服务器，客户端进入循环，等待服务器的 SYN-ACK 响应。
2. 第二次握手：服务器返回客户端 SYN+ACK 段。服务器收到客户端发来的 SYN 数据包后，由标志位 SYN=1 知道客户端请求建立连接服务器将标志位 SYN 和 ACK 都置为 1，ack=0+1，并将该数据包发送给 Client 以确认连接请求。
3. 第三次握手：客户端收到服务器发送的 SYN+ACK 段，给服务器响应 ACK 段。客户端收到确认后，检查 ack，之后客户端向服务器发送 ACK 消息，确认号为服务器 SYN-ACK 消息的序列号 +1，服务器检查正确则连接建立成功。

建立连接（以服务器为例）

```
    // 建立连接
void Connect(bool &connectionEstablished) {
    message msg = recvmessage(server, clientaddr);

    if (msg.isSYN()) {
        cout << "Server:␣接收到一个客户端的连接请求。" << endl;
        // 发送确认消息
        message ackMsg;
        ackMsg.setACK();
        ackMsg.setSYN();
        ackMsg.ack = msg.seq + 1;
        sendmessage(server, clientaddr, ackMsg);
```

```
        cout << "Server: 发送服务器的连接请求。" << endl;
        //ackMsg.output();
        int count = 0;
        while (true) {
            Sleep(50);
            if (count >= 10) {
                cout << "Server: 等待时间太长，退出连接" << endl;
                return ;
            }
            message msg = recvmessage(server, clientaddr);
            if (!msg.isEXT()) {
                continue;
            }
            if (msg.isACK() && msg.ack == ackMsg.seq + 1) {
                break;
            }
            count++;
        }
        cout << "Server: 连接请求已确认，三次握手已完成。" << endl;
        connectionEstablished = true;
    }
    int iMode = 0; //1: 非阻塞，0: 阻塞
    ioctlsocket(server, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置
}
```

(二) 断开连接的设计（四次挥手）

由于第二次挥手和第三次挥手之间不再有数据报的传输，所以我将第二次和第三次挥手合并，在我的代码中只进行了三次挥手。

1. 第一次挥手：客户端创建一个 FIN 消息，计算并设置校验和，然后发送给服务器以开始断开连接的过程。客户端进入循环，等待服务器的 FIN 消息。

2. 第二次挥手：服务器接收到客户端的关闭请求后，服务器构造一个 FIN 消息，并计算校验和。然后，服务器发送 FIN 消息给客户端，表示希望关闭连接。服务器输出日志，表示关闭连接。

3. 第三次挥手：当收到服务器的 FIN 消息时，客户端确认服务器也准备关闭连接。客户端发送最后一个 ACK 消息，确认号为服务器 FIN 消息的序列号 +1。客户端输出日志，表示已确认服务器的关闭请求，并完成四次挥手。

关闭连接（以客户端为例）

```
// 四次挥手关闭连接
void Close() {
    message finMsg;
    finMsg.setFIN();
    finMsg.setchecksum();

```

```
    sendmessage(client, serveraddr, finMsg);
    cout << "Client:␣发送FIN" << endl;


    // 等待服务器发送FIN
    while (true) {
        message finMsg = recvmessage(client, serveraddr);
        if (finMsg.isFIN()) {
            cout << "Client:␣收到服务器的FIN" << endl;
            break;
        }
    }

    // 发送确认ACK
    message ackMsg;
    ackMsg.setACK();
    ackMsg.ack = finMsg.seq + 1;
    sendmessage(client, serveraddr, ackMsg);
    cout << "Client:␣发送确认ACK" << endl;

    cout << "Client:␣四次挥手关闭连接成功" << endl;
    closesocket(client);
    WSACleanup();
}
```

四、 可靠数据传输

(一) 差错检测

我使用了 `setchecksum` 和 `corrupt` 两个函数，用于进行差错检测。

setchecksum 函数 此函数的目的是计算并设置数据包的校验和，以便进行差错检测。其步骤如下：

1. 将消息结构体转换为 `u_short` 类型的数组，使数据可以按 16 位整数处理。
2. 遍历此数组，累加每个 16 位整数的值，以得到总和。
3. 由于在累加过程中可能会出现溢出，因此需要将累加结果的高 16 位加到低 16 位上。此过程可能需要重复多次，直到高 16 位为零。
4. 最终，将累加的结果取反，得到校验和，并将其存储在消息结构体的校验和字段中。

corrupt 函数 此函数用于检测数据包在传输过程中是否出现错误。其步骤如下：

1. 类似于 `setchecksum` 函数，将消息结构体转换为 `u_short` 类型的数组。
2. 遍历数组，累加每个 16 位整数的值，包括校验和字段本身。
3. 同样处理可能的溢出，将累加结果的高 16 位加到低 16 位上。

4. 最后，将计算得到的和与校验和字段相加。如果数据在传输过程中完好无损，最终的和应该是 0xFFFF（因为校验和是通过取反得到的）。如果不是 0xFFFF，则表示数据在传输过程中出现了错误。

差错检测

```cpp
void setchecksum() {
    u_short* temp = reinterpret_cast<u_short*>(this);
    int words = sizeof(message) / sizeof(u_short);

    u_long sum = 0;
    // 将消息结构体视为u_short数组，计算所有16位整数的和
    for (int i = 0; i < words; i++) {
        sum += temp[i];
    }

    // 处理可能的溢出，将高16位回卷到低16位
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }

    // 将校验和设为和的按位取反
    this->checksum = static_cast<u_short>(~sum);
}

bool corrupt() {
    u_short* temp = reinterpret_cast<u_short*>(this);
    int words = sizeof(message) / sizeof(u_short);

    u_long sum = 0;
    // 将消息结构体视为u_short数组，计算所有16位整数的和
    for (int i = 0; i < words; i++) {
        sum += temp[i];
    }

    // 处理可能的溢出，将高16位回卷到低16位
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }

    // 判断消息是否损坏，校验和和消息中的校验和字段相加，如果不等于0xFFFF则表示损坏
    return (checksum + static_cast<u_short>(sum)) != 0xFFFF;
}
```

(二) 流量控制

本实验的流量控制采用的是停等机制，即：发送方发送一帧，就得等待应答信号回应后，继续发出下一帧，接收站在接收到一帧后，发送回一个应答信号给接收方，发送方如果没有收到应答信号则必须等待，超出一定时间后启动重传机制。

在停等机制下，发送方每次发送的数据包必须在收到接收方的应答响应之后，才能进行下一次的发包。若长时间未收到应答响应，那么就会启动超时重传机制。

1. 客户端

状态：

1. 等待发送状态: 等待上层应用传输数据。
2. 等待 ACK 状态: 已发送数据，等待接收端的确认（ACK）。

状态转变：

1. 从等待发送状态到等待 ACK 状态：当发送端收到上层应用的数据时，它将数据打包发送，并开始等待接收端的确认。

2. 从等待 ACK 状态到等待发送状态：当发送端收到确认时，它表示数据已经成功传输，发送端可以等待新的数据。

客户端停等机制

```
while (filelen) {
    num++;

    message msg;
    msg.seq = seq++;

    msg.len = min(filelen, 1024);
    in.read(msg.data, msg.len);
    filelen -= msg.len;

    msg.num = num;

    sendmessage(client, serveraddr, msg);

    // 等待服务器的确认 ACK
    int start = clock();
    int end;
    while (true) {

        message ackMsg = recvmessage(client, serveraddr);
        end = clock();
        if (end - start > time_wait) {
            cout << "Client:超时重传" << endl;
            sendmessage(client, serveraddr, msg);
            start = clock();
        }

        if (!ackMsg.isEXT()) {
```

```
                continue;
            }

            if (ackMsg.isACK() && ackMsg.ack == msg.seq+1 ) {
                cout << "Client:收到ack为" << ackMsg.ack << "的数据包" <<
                    endl;
                break;
            }
        }
    }
```

2. 服务器

状态：

1. 等待数据状态: 等待接收端上层应用传输的数据。

2. 发送 ACK 状态: 接收到对应的数据包，向客户端传输接收端的确认（ACK）。

状态转变：

1. 从等待数据状态到发送 ACK 状态：当接收端收到数据时，它向上层应用传递数据，并发送 ACK 给发送端。

2. 从发送 ACK 状态到等待数据状态：当给客户端传递完对应的 ack 后，继续等待下一个数据包。

服务器停等机制

```
    while (1) {
        msg = recvmessage(server, clientaddr);
        int start = clock();
        int end;
        if (!msg.isEXT()) {
            continue;
        }

        if (msg.isEND()) {
            message ackMsg;
            ackMsg.setACK();
            ackMsg.ack = msg.ack + 1;
            sendmessage(server, clientaddr, ackMsg);
            cout << "Server:接收文件成功!!" << endl << endl;
            cout << "************************************************" <<
                endl;
            outFile.close();
            outFile.clear();
            int iMode = 0; //1: 非阻塞, 0: 阻塞
            ioctlsocket(server, FIONBIO, (u_long FAR*) & iMode);//非阻塞设置

            return ReceiveName(server, clientaddr);
        }

```

```
        if (!msg.isEXT()) {
            continue;
        }
        else if (msg.seq == expectedSeq) {
            cout << "msg.seq:" << msg.seq << "  expectedSeq:" << expectedSeq
                << endl;
            message ackMsg;
            ackMsg.setACK();
            ackMsg.ack = msg.seq + 1;

            //cout << "Server: 收到seq为" << msg.seq << "的数据包" << endl;
            msg.output();

            double random_value = (double)rand() / RAND_MAX;
            if (random_value < loss_rate) {
                cout << endl;
                cout << "Server: 模拟丢包，未发送数据包" << endl;
                cout << endl;
                sendmessage(server, serveraddr, message()); // 返回空消息
                continue;
            }

            sendmessage(server, clientaddr, ackMsg);
            cout << "Server: 发送确认收到的数据包(对应的ack)" << endl;

            cout << endl;
            outFile.write(msg.data, msg.len);

            num--;
            expectedSeq++;
            continue;
        }

    }
```

(三) 超时重传

客户端如果一直无法接收到服务器传来的对应 ack 时，客户端会一直在等待 ACK 状态停留。我通过计算当前等待时间 end - start 是否超过了预设的 time_wait，如果超时则执行重传逻辑。如果超时，会打印"Client: 超时重传" 并重新发送消息 sendmessage(client, serveraddr, msg)，然后更新 start = clock() 以重新计时。

超时重传

```
    while (true) {

        message ackMsg = recvmessage(client, serveraddr);
        end = clock();
        if (end - start > time_wait) {
```

```
            cout << "Client:␣超时重传" << endl;
            sendmessage(client, serveraddr, msg);
            start = clock();
        }

        if (!ackMsg.isEXT()) {
            continue;
        }

        if (ackMsg.isACK() && ackMsg.ack == msg.seq+1 ) {
            cout << "Client:␣收到ack为" << ackMsg.ack << "的数据包" << endl;
            break;
        }
    }
```

五、 其他设计

(一) 传输性能

这部分，我使用 clock 函数计算出传输文件的总时间并输出。

同时，吞吐率计算的为传输的总比特数除以传输所用的时间，再除以 1024（以将结果转换为千比特每秒，即 kbps)。代码如下:

传输性能

```
        int start = clock();
        SendName(filepath);
        //SendFile(filepath);
        int end = clock();
        cout << "成功发送文件! " << endl;
        double endtime = (double)(end - start) / CLOCKS_PER_SEC;
        cout << endl;
        cout << "传输总时间:␣" << endtime << "s" << endl;
        cout << "吞吐率:␣" << (double)(messagenum)*BUFFER * 8 / endtime /
            1024 << "kbps" << endl;
        cout << "**************************************************" << endl
            ;
```

(二) 丢包设计

我的丢包是用来模拟服务器无法发送 ACK。我先生成随机值在 0 到 1 之间的随机数。然后通过与预先设定的丢包率 loss_rate 比较生成的随机值，模拟丢包情况。如果生成的随机值小于丢包率，表示发生了丢包事件。在这种情况下，代码会输出一条消息提示模拟丢包，并发送一个空消息给服务器，模拟实际上没有发送数据包。如果客户端无法收到对应 ack 的话会进行超时重传。

丢包设计

```
        double random_value = (double)rand() / RAND_MAX;
        if (random_value < loss_rate) {
            cout << endl;
            cout << "Server:␣模拟丢包，未发送数据包" << endl;
            cout << endl;
            sendmessage(server, serveraddr, message()); // 返回空消息
            continue;
        }
```

(三) 文件传输

传输端:

我通过输入文件名打开指定路径的文件，并获取该文件的大小。首先，它将文件路径复制到 filepath 中，然后以二进制模式打开文件流，并将文件指针移动到文件末尾以获取文件大小。最后通过文件指针不断地从文件中读取数据块，直到整个文件被读取完毕。在每次循环迭代中，它会更新 filelen，以跟踪文件剩余的未读取部分。

接收端:

接收到发送端的文件名后，在当前目录建立并打开对应的文件，然后随着文件传输过程，依次将传输过来的内容写到目标文件中。

六、 实验结果

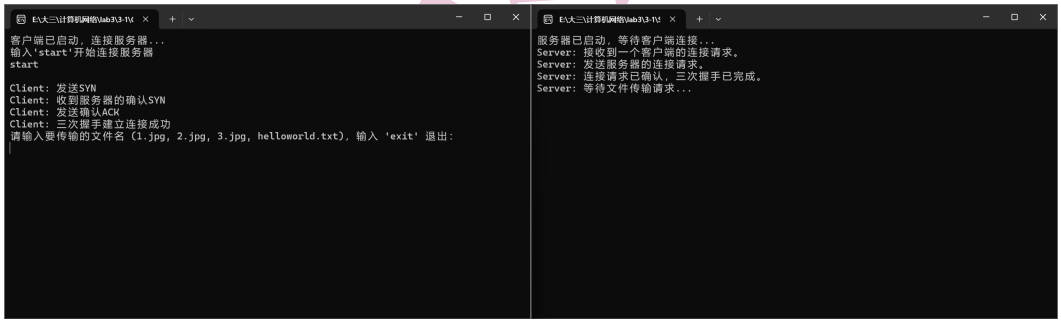

图 1: 建立连接

三次握手显示成功!

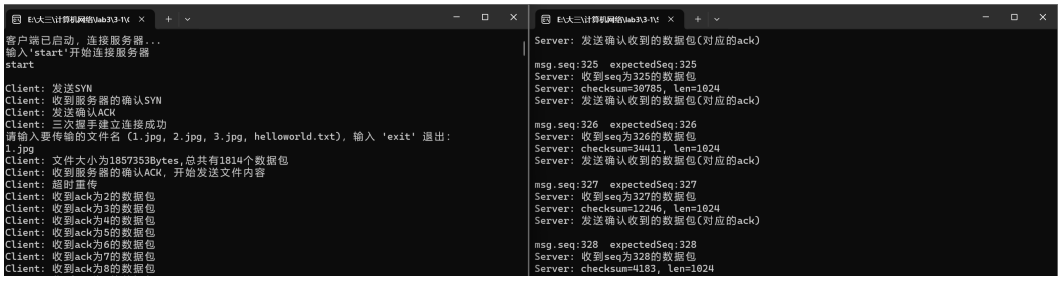

图 2: 文件传输

文件传输过程中，客户端打印传输日志，服务器打印响应日志。

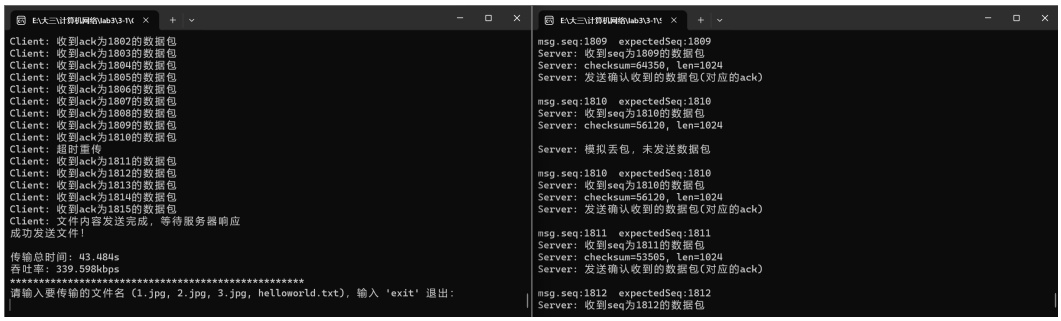

图 3: 超时重传

当模拟丢包时，客户端会对上一个数据包进行重传。如上图，在接收到第 1810 数据包后，返回的 ACK 丢失。当发送端一直无法收到第 ack=1811 后，由于响应超时，所以发送端将 1810 数据包重新发送。所以在接收端的日志中可以看到接收到两次 1810 数据包，因为在代码中有对重复数据包的处理设计，所以接收端只有在发送 ACK 后才会写入，故接收端只写入了一次 1810 数据包。

图 4: 关闭连接

采用三次握手的方式关闭连接。因为关闭连接是在所有数据传输结束后进行的，所以我省去了第二次挥手，将四次挥手改成三次挥手。

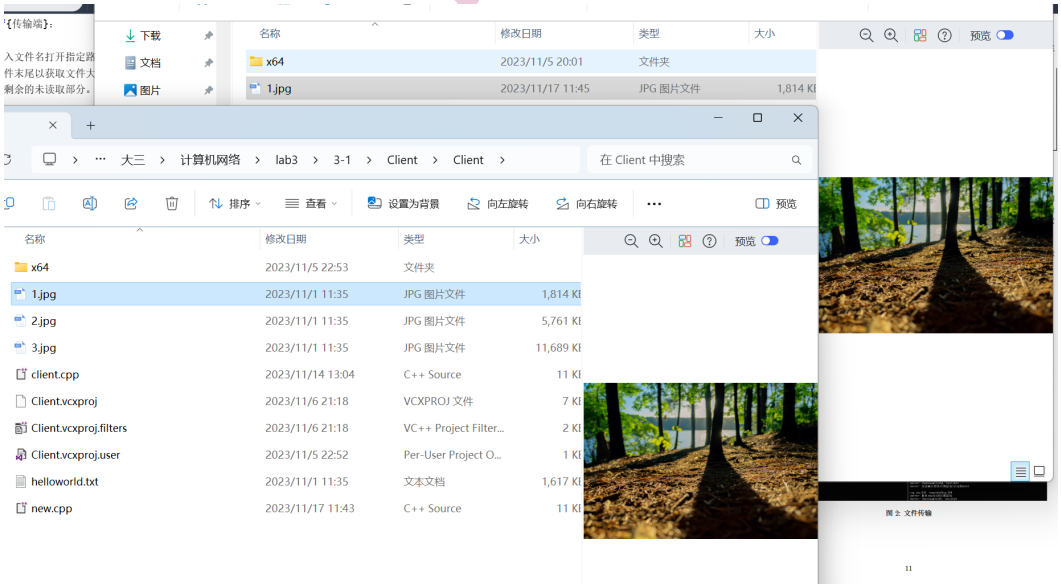

图 5: 成功接收文件

经过数次的尝试和修改之后，程序可以顺利运行，数据传输成功，上面展示了 1.jpg 成功截图，其他样例也都也可以传输成功。

七、 实验总结

基于 UDP 服务实现可靠传输实验第一部分，成功利用数据报套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传。流量控制采用停等机制，完成给定测试文件的传输。在助教学长的提醒下我也发现了自己代码的一点问题：发送端向接收端传输的时候也应该传输对应的 ack 序列号，这样可以避免 ACK 丢失造成的文件传输差错。针对此问题我在后面的实验中会进行改进。