编程作业三实验报告

实验环境

开发环境如下：

- 操作系统：Windows11
- IDE：VS Code
- 编译器：MINGW64

由于使用的是VS Code环境，所以编写Makefile避免重新编译。

协议设计

传输方式

本次传输基于UDP传输协议，采取停等机制控制流量，实现可靠的数据传输。

传输格式

在文件传输时，需要额外携带如下信息：

- 32位的`crc32`校验码，用于校验数据帧是否正确。
- 8位的`flag`，用于标识传输数据类型，包括如下几类：
  - `ACK`：确认数据接收。
  - `PSH`：表明当前帧为数据。
  - `SYN`：建立连接请求。
  - `FIN`：断开连接请求。
  - `RST`：重新连接。
- 8位的`seq`，用于标识当前发送的数据帧的序列号，在使用停等协议时，该值为0或1。

后续字节为传输的具体信息，定义单词传输最长的信息长度为`MAX_SEND_SIZE = 8192`字节。

每次发送方发送数据时`seq`都会改变：如果当前`seq`为0，则变为1，如果当前`seq`为1，则变为0。

接收方收到数据时，会将收到的`seq`附加在回复消息的`seq`中。

状态转移

仿照TCP协议， 我们设计了发送方与接收方总共九种状态：

- `CLOSED`： 发送端和接收端发送完成后的状态
- `LISTEN`： 发送端和接收端的初始状态
- `SYN_SENT`： 发送端发送第一次握手（SYN）后进入的状态
- `SYN_RCVD`： 接收端发送第二次握手（SYN+ACK）后进入的状态
- `ESTABLISHED`： 数据传输状态，发送或接收到第三次挥手后进入的状态

- `CLOSE_WAIT`：接收端发送第二次挥手（ACK）后进入的状态
- `FIN_WAIT_1`：发送端发送第一次挥手（FIN+ACK）之后进入的状态
- `LAST_ACK`：接收端发送（FIN+ACK）之后进入的状态
- `FIN_WAIT_2`：发送端收到第二次挥手（ACK）后进入的状态

状态之间的转换会在发送或者接收消息之后进行，也可以根据需要自行设定。

建立连接

仿照HTTP协议，采取三次握手的方式建立连接，即：

- 第一次握手：发送方向接收方发送`SYN`，请求建立连接。
- 第二次握手：接收方向发送方发送`SYN+ACK`，响应建立连接。
- 第三次握手：发送方向接收方发送`ACK`，连接建立成功。

差错检测

由于UDP本身也会采取一种校验方式，所以在这里我们采取另一种`crc32`的校验方式，不同于传统的`crc32`校验，我们将校验信息放在UDP传输数据的起始位置，后续的所有位进行`crc32`校验的查表方式的校验计算。

文件传输

第一次传输文件名，后续以二进制的方式读入并传输传输文件内容，如果文件内容大于一次发送的部分，则会截取`MAX_SEND_SIZE`长度的部分发送，其余部分后续发送，直到文件传输完毕为止。

在接收文件名时，需要判断当前目录下是否有`recv`文件夹，如果没有则创建，在该文件夹中创建与发送文件相同文件名的空文件，并按二进制方式写入后续数据。

文件传输过程中`flag`为`PSH`。

接收确认

在接收方接受一次文件内容，需要判断当前收到的`seq`是否与上一次的相同，如果相同则丢弃。如果不相同且数据正确，则返回`ACK`。

超时重传

初始停等时间设为`100ms`，后续每进行一次消息的发送与接收，就计算一次从数据发送时间与往返时间之和，将其乘以`1.2`作为停等时间。

如果停等时间为0，则将停等时间设置为2微秒。在实际传输的过程中几乎不会出现该情况。

最大重传次数为10次，如果重传次数小于5次，则每次停等时间加倍；如果停等时间等于5次，则判断停等时间是否小于1秒，如果小于1秒，将停等时间设置为1秒，后续每次停等时间加倍。

如果发送方收到的数据有误，也会认为接收方未收到数据，从而发生重传。

断开连接

仿照HTTP协议，采取四次挥手的方式断开连接，同时作为文件传输结束的标识：

- 第一次挥手：发送方向接收方发送`FIN`，表示文件传输完成，可以断开连接。
- 第二次挥手：接收方向发送方发送`ACK`，表示收到消息。
- 第三次挥手：接收方向发送方发送`FIN+ACK`，表示断开连接。
- 第四次挥手：发送方向接收方发送`ACK`，表示收到消息，传输结束。

由于这里采用单线程的方式处理，接收方向发送方发送`ACK`时数据已经接收完毕，所以这里的第二、三次挥手可以合并。但是为了后续扩展，这里依旧保留四次挥手。

实现方法

整体设计

本次实验包含的工程文件列表如下：

```
crc32.cpp           ---------- 用于CRC32校验
crc32.h             ---------- 用于CRC32校验
defs.h              ---------- 宏定义与结构体定义
defs.cpp            ---------- 变量与函数的实现
main.cpp            ---------- 程序入口
recv_file.cpp       ---------- 实现接收文件的类
recv_file.h         ---------- 实现接收文件的类
send_file.cpp       ---------- 实现发送文件的类
send_file.h         ---------- 实现发送文件的类
file_trans.h        ---------- 发送与接收数据的基类
file_trans.cpp      ---------- 发送与接收数据的基类
```

程序通过命令行执行，可以通过命令行参数规定接收和发送消息的IP地址。

我们设计了基类`FileTrans`，对状态转移以及发送接收数据做了一定的封装。将发送文件和接收文件分别设计为`FileTrans`的派生类，对外提供`setFile`、`start`等接口 。在析构函数中释放资源。

以`SendFile`类为例，其中包含的属性、方法的作用如下：

```
class FileTrans
{
protected:
    SOCKET sock_;           // 发送方绑定的套接字
    sockaddr_in recvAddr_;  // 接收方的IP地址和端口信息
    sockaddr_in sendAddr_;  // 发送方的IP地址和端口信息
    fileMessage* sendMsg_;  // 发送的消息
    fileMessage* recvMsg_;  // 接收的消息
    states state_;          // 所处状态
    int addrSize_;          // 地址大小
    enum Type               // 类型，发送或接收
    {
        F_SEND,
```

```
        F_RECV,
    } ;
public:
    FileTrans(/* args */);
    RC open();                    // 初始化状态
    virtual int getSeq() = 0;     // 获取seq
    virtual Type getType() = 0;   // 获取类型
    RC recvMsg(int &len);         // 接收消息
    RC sendMsg(int len = 0, int seq = -1);  // 发送消息
    ~FileTrans();
};
```

数据格式

数据传输结构定义如下：

```
#pragma pack(1)
struct info
{
    uint32_t crc32;
    unsigned char flag;
    unsigned char seq;
};
struct fileMessage
{
    struct info head;
    char msg[MSS];
};
#pragma pack()
```

利用结构体在内存中连续存储的特点，可以方便的取出数据的各个属性的值。

CRC32校验

我们使用查表法计算`CRC32`校验码，提高计算速度：

```
uint32_t crc32(const unsigned char *buf, uint32_t size)
{
    uint32_t i, crc;
    crc = 0xFFFFFFFF;
    for (i = 0; i < size; i++)
        crc = crc32tab[(crc ^ buf[i]) & 0xff] ^ (crc >> 8);
    return crc^0xFFFFFFFF;
}
```

数据发送

我们对发送数据进行封装，将设置`seq`、计算`crc32`和发送数据的功能封装为`send_message`方法。在调用该函数之前，需要正确设置发送的`flag`和`msg`。

```
if(len < 0) return RC::INTERNAL;
if(seq == -1)
    sendMsg_->head.seq = getSeq();
sendMsg_->head.crc32 = crc32((unsigned char*)&(sendMsg_->head.flag),len + sizeof(info) - sizeof(info::crc32));
if(sendto(sock_, (char*)(sendMsg_), len + sizeof(info), 0, (sockaddr*)&sendAddr_, sizeof(sendAddr_)) == -1)
    return RC::SOCK_ERROR;
cout << "[ send ] [ seq ] = " << (int)sendMsg_->head.seq << " [ flag ] = 0x" << hex << (int)sendMsg_->head.flag << " [ len ] = " << dec << len << " [ state ] = " << stateName[state_] << endl;
```

在发送完向相应的数据后我们需要对当前的状态进行更新：

```
switch(state_)
{
case LISTEN:
    if(getType() == Type::F_SEND)
        state_ = SYN_SENT;
    else if(getType() == Type::F_RECV)
        state_ = SYN_RCVD;
    break;
case SYN_SENT:
    if(sendMsg_->head.flag == (SYN | ACK))
        state_ = SYN_RCVD;
    else if(sendMsg_->head.flag = ACK)
        state_ = ESTABLISHED;
    break;
case ESTABLISHED:
    if(recvMsg_->head.flag == FIN)
        state_ = CLOSE_WAIT;
    else if(sendMsg_->head.flag == FIN)
        state_ = FIN_WAIT_1;
    break;
case CLOSE_WAIT:
    state_ = LAST_ACK;
    break;
case FIN_WAIT_1:
case FIN_WAIT_2:
    state_ = CLOSED;
    break;
default:
    break;
}
```

数据接收

正常的数据接收需要调用`recvfrom`，并对接收的数据进行检查：

```
RC rc = RC::SUCCESS;
len = recvfrom(sock_, (char*)recvMsg_, sizeof(fileMessage), 0,
(sockaddr*)&recvAddr_, &addrSize_);
cout << "[ recv ] [ seq ] = " << (int)recvMsg_->head.seq << " [ flag ] = 0x" <<
hex << (int)recvMsg_->head.flag << " [ len ] = " << dec << len - sizeof(info)  <<
" [ state ] = " << stateName[state_] << endl;
if(len == SOCKET_ERROR)
    return RC::SOCK_ERROR;
uint32_t check_crc32 = crc32((unsigned char*)&(recvMsg_->head.flag),len  -
sizeof(info::crc32));
if(check_crc32 != recvMsg_->head.crc32)
    return RC::CHECK_ERROR;
```

之后还需要对当前的状态进行更新，并设置下一次发送数据的`flag`（如果有必要的话）：

```
auto& flag = recvMsg_->head.flag;
switch(state_)
{
    case CLOSED:        // 传输已完成，需要重新建立连接
        return RC::STATE_ERROR;
    case LISTEN:        // 接收端正在等待第一次握手
        if(flag == SYN)
            sendMsg_->head.flag = SYN | ACK;
        break;
    case SYN_SENT:      // 发送端发送完第一次握手
        if(flag == SYN)
            sendMsg_->head.flag = SYN | ACK;
        else if(flag == (SYN | ACK))
            sendMsg_->head.flag = ACK;
        else state_ = LISTEN;
        break;
    case SYN_RCVD:      // 接收端发送第二次握手
        if(flag == ACK)
            state_ = ESTABLISHED;
        else state_ = LISTEN;
        break;
    case ESTABLISHED:   // 数据传输状态
        if(flag == FIN)
            sendMsg_->head.flag = ACK;
        else if(flag == PSH)
            sendMsg_->head.flag = ACK;
        break;
    case CLOSE_WAIT:    // 接收端发送第二次挥手后，应该发送FIN+ACK而不是接收
        return RC::STATE_ERROR;
    case FIN_WAIT_1:    // 第一次挥手后
        if(flag == ACK)
            state_ = FIN_WAIT_2;
```

```
        else if(flag == (FIN | ACK))
            sendMsg_->head.flag = ACK;
        else
            state_ = ESTABLISHED;
        break;
    case LAST_ACK:      // 第三次挥手后
        if(flag == ACK)
            state_ = CLOSED;
        else state_ = ESTABLISHED;
        break;
    case FIN_WAIT_2:    // 收到第二次挥手后，等待第三次挥手
        if(flag == (FIN | ACK))
            sendMsg_->head.flag = ACK;
        else state_ = ESTABLISHED;
        break;
    default:
        return RC::INTERNAL;
}
```

在接受文件的回应时，我们需要使用`select`函数设置定时器，经过`tv_`时间后停止接收，并返回超时的错误。

```
RC RecvFile::recv_message(int &len)
{
    RC rc;
    while(true)
    {
        rc = recvMsg(len);
        if(rc != RC::SUCCESS)
            return rc;
        if(recvMsg_->head.seq == seq_)
        {
            sendMsg_->head.flag = ACK;
            rc = sendMsg();
            if(rc != RC::SUCCESS)
                return rc;
            continue;
        }
        break;
    }
    seq_ = recvMsg_->head.seq;
    return rc;
}
```

握手与挥手

发送方和接收方正确调用`send_message`和`recv_message`即可，在此处不多赘述。

文件读取与发送

在发送文件时，我们将文件按每次发送的大小读入`sendMsg_->msg`，之后直接发送消息，避免将文件整体读入内存占用空间。读取完毕后调用`send_and_wait`发送并等待响应。

```
sendFileStream_.open(fileName, ios::binary);
sendFileStream_.seekg(0, ios::end);
int wait_send_len = sendFileStream_.tellg();
fileSize_ = wait_send_len;
cout << "发送文件大小为 " << wait_send_len << " B" << endl;
sendFileStream_.seekg(0, ios::beg);
int ret;
while(wait_send_len > 0)
{
    int send_len = wait_send_len <= MAX_SEND_SIZE ? wait_send_len :
MAX_SEND_SIZE;
    sendMsg_->head.flag = PSH;
    sendFileStream_.read(sendMsg_->msg, send_len);
    ret = send_and_wait(send_len);
    if(recvMsg_->head.flag != ACK)
    {
        cout << "接收方响应连接错误" << endl;
        return 0;
    }
    if(ret == false) return ret;
    wait_send_len -= send_len;
}
```

停等时长确定

我们会记录每一轮发送前的时间，与正确接收到响应的时间，将其作差并乘以一个倍数作为下一次的停等时长：

```
if(end.tv_sec == start.tv_sec)
{
    tv_.tv_sec = 0;
    tv_.tv_usec = (end.tv_usec - start.tv_usec) * 1.2;
}
else
{
    tv_.tv_sec = (end.tv_sec - start.tv_sec - 1) * 1.2;
    tv_.tv_usec = (1000000 + end.tv_usec - start.tv_usec) * 1.2;
}
if(end.tv_usec-start.tv_usec == 0)
    tv_.tv_usec = 2;
```

文件的接收与响应

接收方接收数据后，会判断当前帧是否为`PSH`，如果是，则将其写入接收文件流中，并向发送方发送`ACK`。

```
bool RecvFile::wait_and_send()
{
    int len;
    while(true)
    {
        len = recv_message();
        if(len == SOCKET_ERROR)
        {
            cout << "与发送方断开连接" << endl;
            return 0;
        }
        else
        {
            if(recvMsg_->head.flag == PSH)
            {
                seq_ = recvMsg_->head.seq;
                recvFileStream_.write(recvMsg_->msg, len - sizeof(info));
                recvFileStream_.flush();
                sendMsg_->head.flag = ACK;
                if(!send_message(0))
                {
                    cout << "发送ACK错误" << endl;
                    return 0;
                }
            }
            else if(recvMsg_->head.flag == FIN)
            {
                return disconnect();
            }
        }
    }
    return 0;
}
```

错误处理

在`defs.h`中，我们将正确或错误的信息设计为枚举类`RC`：

```
enum class RC
{
    SOCK_ERROR,          // 套接字错误
    CHECK_ERROR,         // 校验错误
    WAIT_TIME_ERROR,     // 等待超时
    SEQ_ERROR,           // seq错误
    STATE_ERROR,         // 状态错误
    UNIMPLENTED,         // 未实现
    INTERNAL,            // 其他错误
    RESET,               // 需要重传
    SUCCESS,             // 没有错误
};
```

为了方便后续的错误处理，我们为其定义一个错误处理函数，在这里后续也可以根据需要扩展错误具体的处理：

```
void handleError(RC rc)
{
    switch(rc)
    {
        case RC::CHECK_ERROR:
            cout << "[ check error ] ";
            break;
        case RC::INTERNAL:
            cout << "[ falied ] ";
            break;
        case RC::SEQ_ERROR:
            cout << "[ seq error ] ";
            break;
        case RC::SOCK_ERROR:
            cout << "[ socket error ] ";
            break;
        case RC::STATE_ERROR:
            cout << "[ state error ] ";
            break;
        case RC::WAIT_TIME_ERROR:
            cout << "[ wait time error ] ";
            break;
        case RC::UNIMPLENTED:
            cout << "[ unimplented ]";
            break;
        default:
            break;
    }
}
```

由于在代码中频繁出现判断返回类型并打印错误的情况，我们将其定义为宏函数：

```
#define LOG_MSG(rc, success, falied) \
    if(rc != RC::SUCCESS) \
    { \
        handleError(rc); \
        cout << falied << endl; \
        return rc; \
    } \
    else if(strcmp(success, "") != 0)\
        cout << success << endl;
```

出现的问题

延时的计算

由于我们使用`timeval`结构计算数据发送以及往返延时，不能直接作差，需要考虑秒和微秒之间的进制转换，不然可能会出现微秒为负数的问题，导致延时无法正确计算，从而导致等待时长错误。

建立和断开连接时的数据丢失

在连接建立和断开时可能会发生数据丢失，对于这种情况如果仅仅检测一次是不够的，所以我们将连接的建立和断开设计为`while`循环，只有当状态到达`ESTABLISHED`或`CLOSED`才能跳出循环。此外，在这里将不正确的连接和断开的状态响应，全部将状态重置为`LISTEN`，避免一直在等待。

实验结果

使用给定环境测试测试文件的传输，丢包为10%，延迟为20ms，为了加快传输速度取消了日志的打印，实验结果如下：

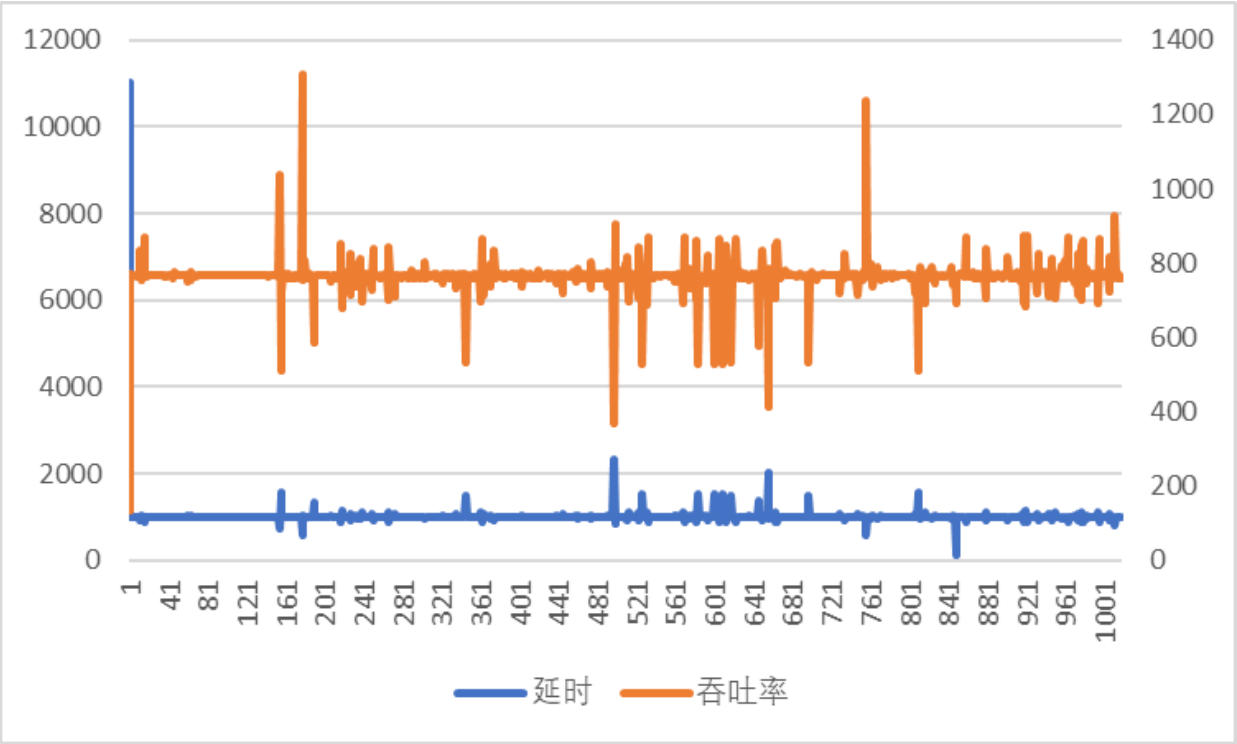
```
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 1
请输入文件名：1.jpg
建立连接成功
发送文件大小为 1857353 B
关闭连接成功
传输总用时为 8.64571 s
平均吞吐率为 1.71864 Mbps
发送文件成功
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 1
请输入文件名：2.jpg
建立连接成功
发送文件大小为 5898505 B
关闭连接成功
传输总用时为 31.5075 s
平均吞吐率为 1.49768 Mbps
发送文件成功
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 1
请输入文件名：3.jpg
建立连接成功
发送文件大小为 11968994 B
关闭连接成功
传输总用时为 65.872 s
平均吞吐率为 1.45361 Mbps
发送文件成功
```

```
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 1
请输入文件名: helloworld.txt
建立连接成功
发送文件大小为 1655808 B
关闭连接成功
传输总用时为 10.7683 s
平均吞吐率为 1.23013 Mbps
发送文件成功
```

```
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 2
建立连接成功
收到文件名: 1.jpg
关闭连接成功
接收等待时长为 8.64407 s
接收文件成功
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 2
建立连接成功
收到文件名: 2.jpg
关闭连接成功
接收等待时长为 31.5133 s
接收文件成功
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 2
建立连接成功
收到文件名: 3.jpg
关闭连接成功
接收等待时长为 65.875 s
接收文件成功
```

```
PS H:\A大三上\计算机网络\实验\实验3测试文件和路由器程序\测试文件> ./main
请选择发送或者接收，发送(1)，接收(2) : 2
建立连接成功
收到文件名: helloworld.txt
关闭连接成功
接收等待时长为 10.7777 s
接收文件成功
```

对其中一次传输进行分析：



可以发现在数据传输传输的开始吞吐率较低，因为一开始发送握手和文件名，传输的长度短。后续吞吐率与延时与延时维持在一个固定水平，偶尔会有抖动，且吞吐率与延时成负相关。