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。
- 32位的`seq`，用于标识当前发送的数据帧的序列号。
- 32位的`ACK`，用于标识确认的序列号。
- 16位的`len`，用于标识传输的文件内容的长度。
- 16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。
- 16位的`win`，用于标识发送窗口或接收窗口的大小。

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`MSS = 8192`字节。

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。

发送文件

在发送文件内容之前，需要先完成三次握手以及文件名的传输。这些操作采取停等机制。

发送文件内容采取滑动窗口机制，固定窗口大小，每发送一次可用窗口数减一，直到减到0。当收到接收方回复的`ACK`时，移动滑动窗口指针，更新窗口大小，使其可以继续发送。

如果收到的`ACK`比当前发送的`Seq`小，说明可能发生了丢包，如果连续收到三次相同的`ACK`，则认为发生丢包，触发**快速重传**，将滑动窗口的下一次发送数据的指针指向`ACK`对应的数据。

零窗口

当可用的接收窗口为零时，发送窗口也为零，这时无法继续发送消息，所以需要定时发送探测报文，`Seq`设置为0，这样接收方会将可用的窗口大小返回给发送方，同时也不会改变接受的数据。

接收文件

接收文件采取滑动窗口机制，将接受的数据保存于接收窗口中。接收方需要两个线程，主线程接收数据，定时发送`ACK`，`ACK`的值等于下一次期望接收的数据的`Seq`。另一个线程用于将数据写入内存，并更新滑动窗口指针。

在主线程中，对接收数据进行计时，如果一定时间未收到数据，则发送`ACK`，这样在发送方不实现超时重传的情况下，可以通过三次`ACK`快速重传达到相同的效果。

接收窗口大小为1，在收到的序列是下一个希望收到的序列时，接收窗口向前滑动一位，否则不滑动。

断开连接

断开连接也采取停等机制。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。

实现方法

与之前相同的内容，在这里不再赘述。

滑动窗口

为了方便发送端和接收端使用，将滑动窗口封装为一个类，可能会遇到多线程访问的内容设计为原子类型的数据，各成员变量和方法的含义如下：

```
class SlidingWindow
{
private:
    int buffSize_;
    std::atomic<uint32_t> start_;       // 起始位置下标
    std::atomic<uint32_t> start_seq_;   // 起始位置对应的Seq
    std::atomic<uint32_t> next_;        // 下一个数据的下标
    std::atomic<uint32_t> end_;         // 最后数据的下标
    uint32_t data_end_;                 // 发送端最后一个数据的下标

public:
    std::vector<fileMessage> sw_;               // 数据
    SlidingWindow(int buffSize, int windowSize);
    SlidingWindow(){}
    void movePos(slidingPos p, int size);       // 移动指针
    void setPos(slidingPos p, int pos);         // 设置指针位置
    int getWindow();                           // 获取可用的窗口大小
    void setWindow(int size);                  // 设置窗口大小
    void setStartSeq(int seq){start_seq_ = seq;}    // 设置起始Seq
    uint32_t getStartSeq() {return start_seq_;}     // 获取起始的Seq
    uint32_t getStart() {return start_;}            // 获取起始位置
    uint32_t getNext() {return next_;}              // 获取下一个位置
    uint32_t getEnd() {return next_;}               // 获取末尾的位置
    uint32_t getDataEnd(){return data_end_;}        // 获取最后一个数据的位置
    uint32_t getIndexBySeq(uint32_t seq);           // 根据Seq获取对应的下标
    uint32_t getSeqByIndex(uint32_t index);         // 根据下标获取对应的Seq
    uint32_t getNextSeq();                         // 获取下一个数据的Seq
    void setData(int index, char* msg, int len);    // 设置一个位置的数据
    void setFlag(int index, WORD flag);             // 设置一个位置的flag
    void setDataEnd(int pos) {data_end_ = pos % buffSize_;} // 设置数据末尾的位置
    void printSliding();                            // 打印滑动窗口指针的位置
    ~SlidingWindow();
```

```
};

```

在这里有几个比较重要的成员变量：

- `start_`：在发送端，代表没有收到`ACK`确认的第一个数据；在接收端，代表还未写入内存的第一个数据。
- `next_`：在发送端，代表下一个将要发送的数据；在接收端，代表下一个需要发送`ACK`的数据。
- `end_`：在发送端，代表在窗口大小限制内可以发送的最后一个数据；在接收端，代表在窗口大小限制内可以接收的最后一个数据。

虽然在本次实验中不会出现滑动窗口缓冲区不够用的情况，但是为了扩展，我们将其设计为一个环形结构，超出缓冲区大小后取模，放到下标较小的缓冲区中。

发送数据

在发送端完成三次握手以及文件名的传输后，需要设置发送窗口的内容：

```
void SendFile::setSendBuf()
{
    int wait_set_len = fileSize_;
    sendWindow_.setStartSeq(seq_);
    int index = 0;
    while (wait_set_len > 0)
    {
        int set_len = wait_set_len <= MSS ? wait_set_len : MSS;
        sendWindow_.setFlag(index, PSH);
        sendFileStream_.read(sendWindow_.sw_[index].msg, set_len);
        sendWindow_.sw_[index].head.len = set_len;
        index++;
        wait_set_len -= set_len;
    }
    sendWindow_.setDataEnd(index);
}
```

然后创建子线程用于接收`ACK`，创建完毕后发送数据。

发送文件内容

发送数据分为三个步骤：

- 设置`sendMsg_`指向`next`的数据
- 调用`sendMsg`发送数据
- 更新`next`指针

为了防止在这个阶段中接收线程修改`next`指针，需要使用互斥锁`mutex_`将这段代码上锁。

循环退出条件为`send_over_`为真，该变量在接收线程中设置，当收到最后一个数据的`ACK`时将该值设为真，接收线程结束。

```

```

```
setSendBuf();
thread wait_ACK(waitACK, this);
while(!getSendOver())
{
    Sleep(1);
    mutex_.lock();
    sendMsg_ = &(sendWindow_.sw_[sendWindow_.getNext()]);
    sendMsg(sendMsg_->head.len);
    sendWindow_.movePos(S_NEXT, 1);
    mutex_.unlock();
    while (sendWindow_.getNext() == sendWindow_.getDataEnd())
    {
        Sleep(WRITE_FILE_TIME);
        if(getSendOver())
            break;
    }
    // 如果没有空间，等待窗口大小更新
    if(sendWindow_.getWindow() == 0)
    {
        sendMsg_ = new fileMessage;
        sendMsg_->head.flag = PSH;
        while (sendWindow_.getWindow() == 0)
        {
            Sleep(KEEP_ALIVE_TIME);
            sendMsg(0, 0);
        }
        delete sendMsg_;
    }
}
wait_ACK.join();
```

接收ACK

接收ACK的函数是一个静态成员函数，其主体部分为一个`while`循环，循环接收消息。如果接收到的ACK与上一次的不同，则更新滑动窗口，将滑动窗口的起始指针移动到与ACK的值对应的位置；如果接收的ACK与上一次的相同，并且连续收到三次相同的ACK，则认为发生丢包，重新设定下一次发送指针的位置和序列号。

```
void SendFile::waitACK(SendFile* sf)
{
    int len;
    int ack_last = -1;
    int repeat_time = 1;
    SlidingWindow& sw = sf->sendWindow_;
    while(!sf->getSendOver())
    {
        sf->recvMsg(len);
        if(sf->recvMsg_->head.flag != ACK)
            continue;
        sw.setWindow(SEND_WINDOW_SIZE);
        if(sf->recvMsg_->head.ack != ack_last)
```

```
        {
            ack_last = sf->recvMsg_->head.ack;
            sw.movePos(S_START, ack_last - sw.getStartSeq());
        }
        else
        {
            repeat_time++;
            if(repeat_time == 3)
            {
                if(ack_last < sw.getNextSeq())
                {
                    sf->mutex_.lock();
                    sw.setPos(S_NEXT, sw.getIndexBySeq(ack_last));
                    sf->setSeq(ack_last);
                    sf->mutex_.unlock();
                }
                repeat_time = 1;
            }
        }
        if(ack_last == sw.getSeqByIndex(sw.getDataEnd()))
            sf->setSendOver(true);
        // sw.printSliding();
    }
}
```

接收文件

接收文件内容

循环接收消息，如果超时未接到消息直接回复ACK。检查接收消息的`flag`，如果为`PSH`说明还处于数据接收状态，如果接受的Seq为下一个希望得到的Seq，则将其写入接收窗口中，接收窗口向前滑动一位，回复ACK；如果`flag`为`FIN`，说明进入关闭连接状态，需要执行四次挥手

```
RC RecvFile::wait_and_send()
{
    int len;
    RC rc;
    timeval start, end;
    gettimeofday(&start, NULL);
    while(true)
    {
        fd_set rset;
        FD_ZERO(&rset);
        FD_SET(sock_, &rset);
        timeval tv = MS_TO_TIMEVAL(WAIT_TIME);
        if(select(sock_ + 1, &rset, NULL, NULL, &tv) > 0)
        {
            rc = recvMsg(len);
        }
        LOG_MSG(rc, "", "与发送方断开连接");
        if(recvMsg_->head.flag == PSH)
```

```
        {
            uint32_t &seq = recvMsg_->head.seq;
            if(seq == recvWindow_.getNextSeq())
            {
                memcpy(&recvWindow_.sw_[recvWindow_.getNext()], recvMsg_,
sizeof(fileMessage));
                recvWindow_.movePos(S_NEXT, 1);
                recvWindow_.movePos(S_END, 1);
            }
            gettimeofday(&end, NULL);
            {
                RC rc;
                setAck(recvWindow_.getNextSeq());
                rc = sendMsg();
                LOG_MSG(rc, "", "发送消息失败");
                start = end;
            }
        }
        else if(recvMsg_->head.flag == FIN)
        {
            setRecvOver(true);
            recvWindow_.setPos(S_END, recvWindow_.getNext());
            return disconnect();
        }
    }
    return rc;
}
```

写入磁盘

在接收文件时还需要写入磁盘，该循环每`200ms`执行一次，每次将`start_`到`next_`指针之间的内容写入磁盘。

```
void RecvFile::writeInDisk(RecvFile* rf)
{
    SlidingWindow& sw = rf->recvWindow_;
    // ofstream write_txt("a.log");
    int t = 0;
    while(!(rf->getRecvOver() && sw.getStart() == sw.getEnd()))
    {
        Sleep(WRITE_FILE_TIME);
        while (sw.getStart() != sw.getNext())
        {
            fileMessage& msg = sw.sw_[sw.getStart()];
            // write_txt << t++ << "  " << msg.head.len << "  " << *
((int*)msg.msg) << endl;
            rf->recvFileStream_.write(msg.msg, msg.head.len);
            sw.movePos(S_START, 1);
            // sw.movePos(S_END, 1);
        }
        rf->recvFileStream_.flush();
```

```
    }
    // write_txt.close();
}
```

遇到的问题

接收端的接收与发送

由于在这里我们将接收端的接收与发送放在同一个线程里，所以会出现发送端没有数据时接收端阻塞运行，无法回复`ACK`，所以这里需要对接收进行定时。

多线程输出

由于在这里需要打印日志，在发送与接收消息为两个线程时会出现打印混乱的状态，所以定义互斥锁`print_mutex_`，在需要输出时都加锁，输出完毕后释放锁。这样会增加打印的资源消耗。在后续测试性能时或许需要取消输出。

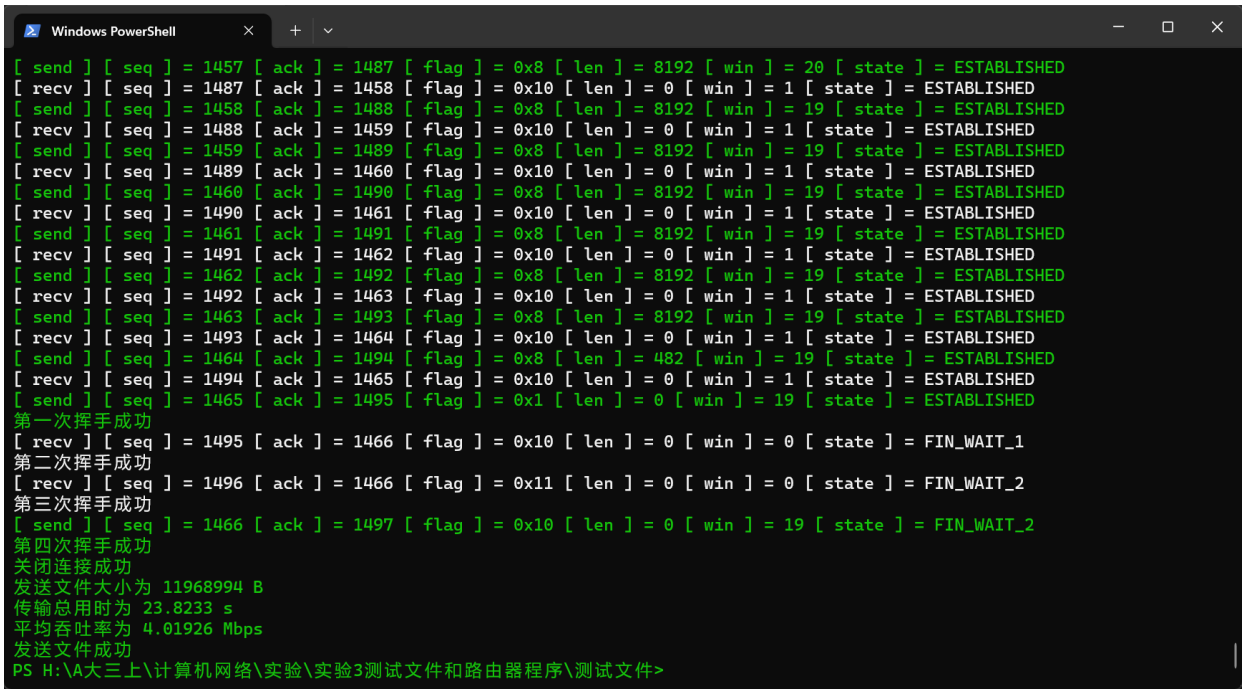
临界变量的访问

对于临界变量，需要使其在同一时间只能由一个线程访问，所以我们将滑动窗口的指针以及发送和接收方的`seq`和`ack`设置为原子数据，保证访问和读写操作的正确性。

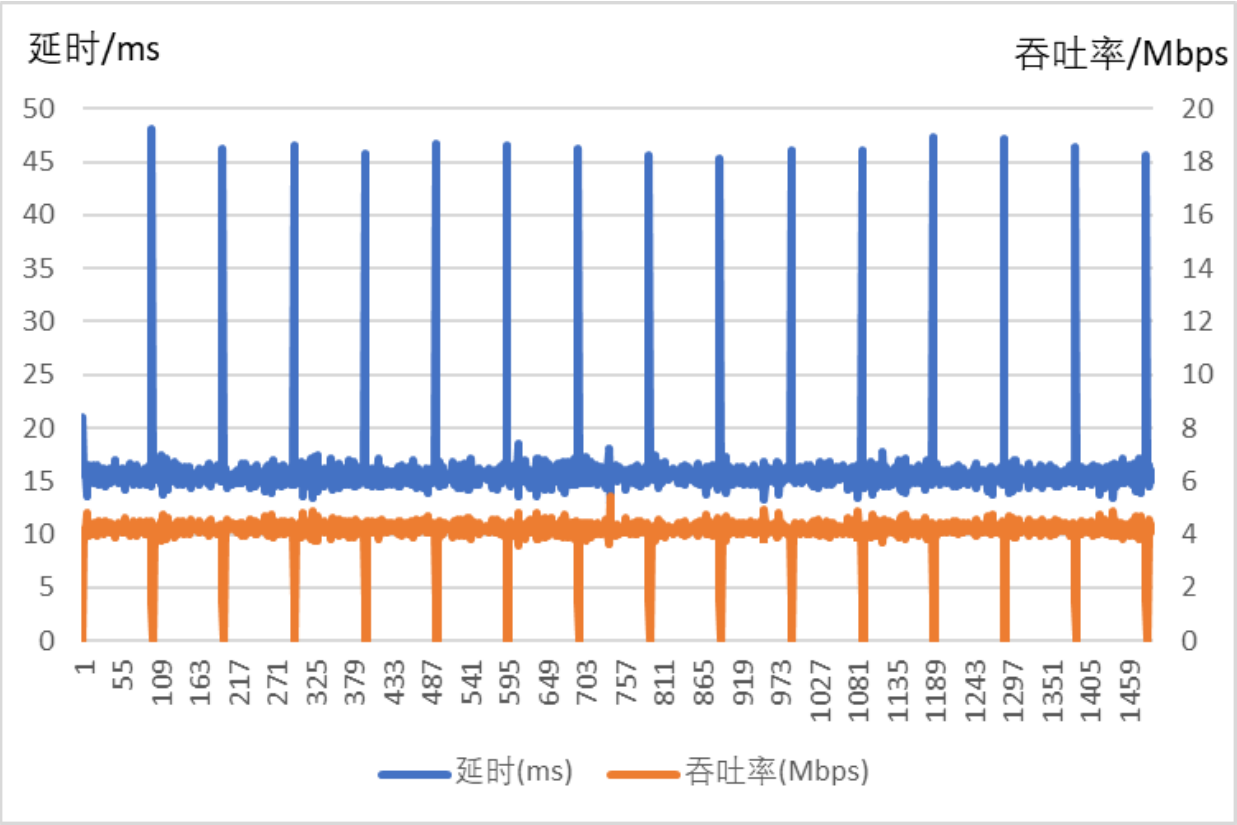
但是在发送文件过程中，主线程一致发送消息，对于我们额外添加的锁一直占用，使得子线程无法处理`ACK`，所以在发送消息的循环中添加一个`Sleep(1)`，使其在这段时间内无法持有锁。

实验结果

在发送窗口为10时，测试文件传输结果如下：



绘制延时与吞吐率图像如下：



可以看到延时和吞吐率大部分情况下都很稳定，周期性出现高延时低吞吐率，这时因为我们的丢包是周期性的丢包。延时高时吞吐率降低。