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**，将该ACK对应的消息指针加入丢包消息的集合中，后续发送的消息将优先发送丢包集合中的消息。

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；如果收到的序列大于该序列，则认为发送数据丢失，将这两个序列之间的数据加入丢包集合中，后续回复ACK的值将优先回复该集合中的值。

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。本次改动的方面主要有以下三点。

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，新添加的各成员变量（主要是丢包集合）和方法的含义如下：

```
std::atomic<std::set<uint32_t>*> loss_ack_;          // 丢包集合
void addLossAck(uint32_t ack);                       // 添加一个丢包序列
int getLossNum(){return loss_ack_.load()->size();}   // 获取当前丢包的总数
uint32_t getNextAck();                               // 获取接收端下一个回复的ACK
int32_t getNextSend();                               // 获取下一个发送的序列
void updateStart();                                  // 根据丢包集合和next_指针确定滑动窗口位置
void updateMsg(fileMessage* msg);                    // 根据数据序列更新滑动窗口的内容
```

在这里使用集合数据结构，是为了去除重复添加的ACK（比如发送端接收到的重复的ACK），同时也使用了集合元素的有序性，集合的第一个元素为最小的序列。

发送数据

发送窗口主要包含三个指针：

- `start_`：发送窗口的起始位置
- `next_`：按照顺序的下一个发送的数据
- `end_`：发送窗口的终止位置

使用`getNextSend`方法获取下一个发送的数据，该方法的思想为：

- 如果丢包集合为空，返回`next_`
- 如果丢包集合不为空，返回集合的第一个元素

在发送数据时，依然是先将文件读入发送缓冲区，也就是发送窗口，再进行发送。

主体部分依旧是`while`循环，不同的是这次使用`updateNext`方法更新`next_`指针，更新原则如下：

- 如果这次发送的是`next_`指针指向的数据，将`next_`向后移动一位
- 如果不是，则从丢包集合中移除该元素

```
while(!getSendOver())
{
    Sleep(1);
    mutex_.lock();
    volatile int send_index = sendWindow_.getNextSend();
    sendMsg_ = &(sendWindow_.sw_[send_index]);
    setSeq(sendWindow_.getSeqByIndex(send_index));
    sendMsg(sendMsg_->head.len);
    sendWindow_.updateNext(send_index);
    mutex_.unlock();
    while (sendWindow_.getNextSend() == 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_.getSendWindow() == 0)
    {
        sendMsg_ = new fileMessage;
        sendMsg_->head.flag = PSH;
        while (sendWindow_.getSendWindow() == 0)
        {
            Sleep(KEEP_ALIVE_TIME);
            sendMsg(0, 0);
        }
        delete sendMsg_;
    }
}
```

在发送端接收ACK时，需要调用`updateStart`更新`start_`指针，更新原则为

- 如果丢包集合为空，则更新为`next_`
- 如果丢包集合不为空，则更新为集合第一个元素

如果连续三次收到相同ACK，则认为发生丢包，将其加入丢包集合中。

接收数据

当接收端接收到数据时，需要调用`updateMsg`更新滑动窗口的数据，更新原则为：

- 如果接受的数据序列号为按照顺序的下一个希望接受的序列，将其拷贝进入合适的位置，滑动窗口向前滑动一位。
- 如果接受的数据序列号在丢包集合中，将其拷贝入对应的位置，从丢包集合中将其删除。
- 如果序列号大于按顺序的下一个序列号，将其拷贝入合适的位置， 将二着之间的元素下标加入丢包集合中。

接收端主体部分依然是一个`while`循环：

```
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;
        recvWindow_.updateMsg(recvMsg_);
        gettimeofday(&end, NULL);
        {
            RC rc;
            setAck(recvWindow_.getNextAck());
            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();
}
```

接收端回复ACK的原则为：

- 如果丢包集合为空，那么接收端发送的下一个ACK为`next_`指针指向的数据
- 如果丢包集合不为空，那么发送ACK为该集合的第一个元素

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。

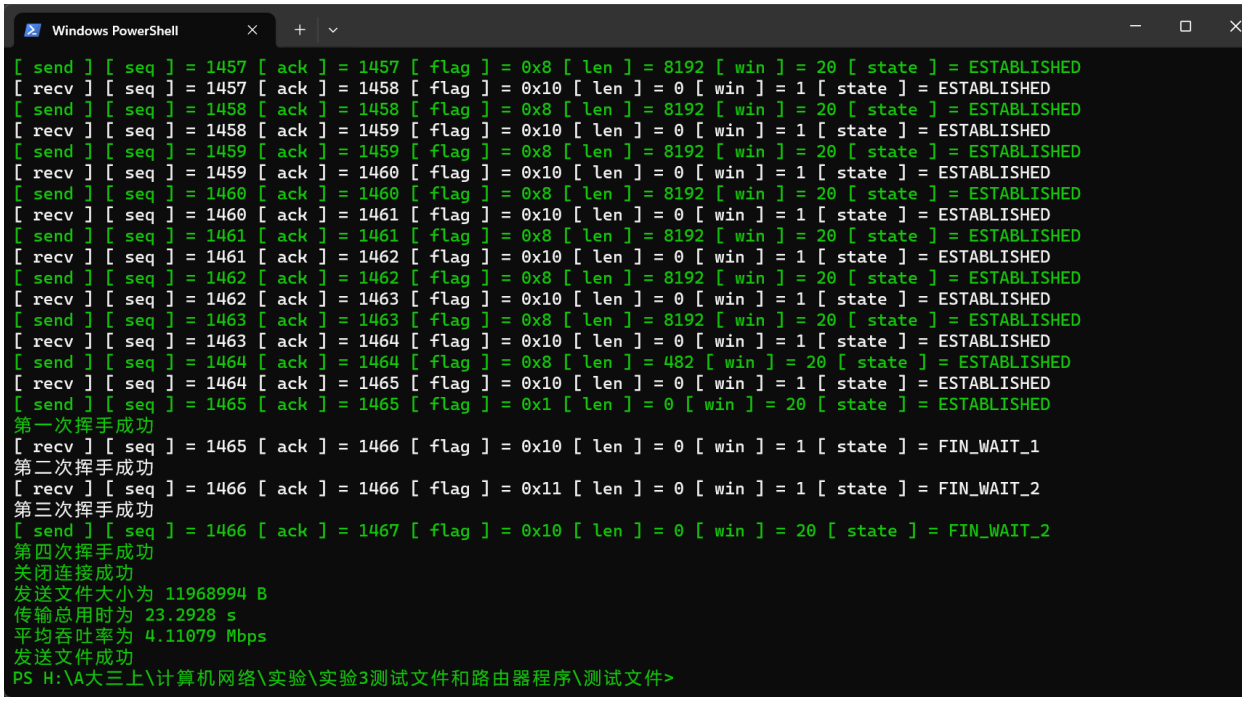
临界变量的访问

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。

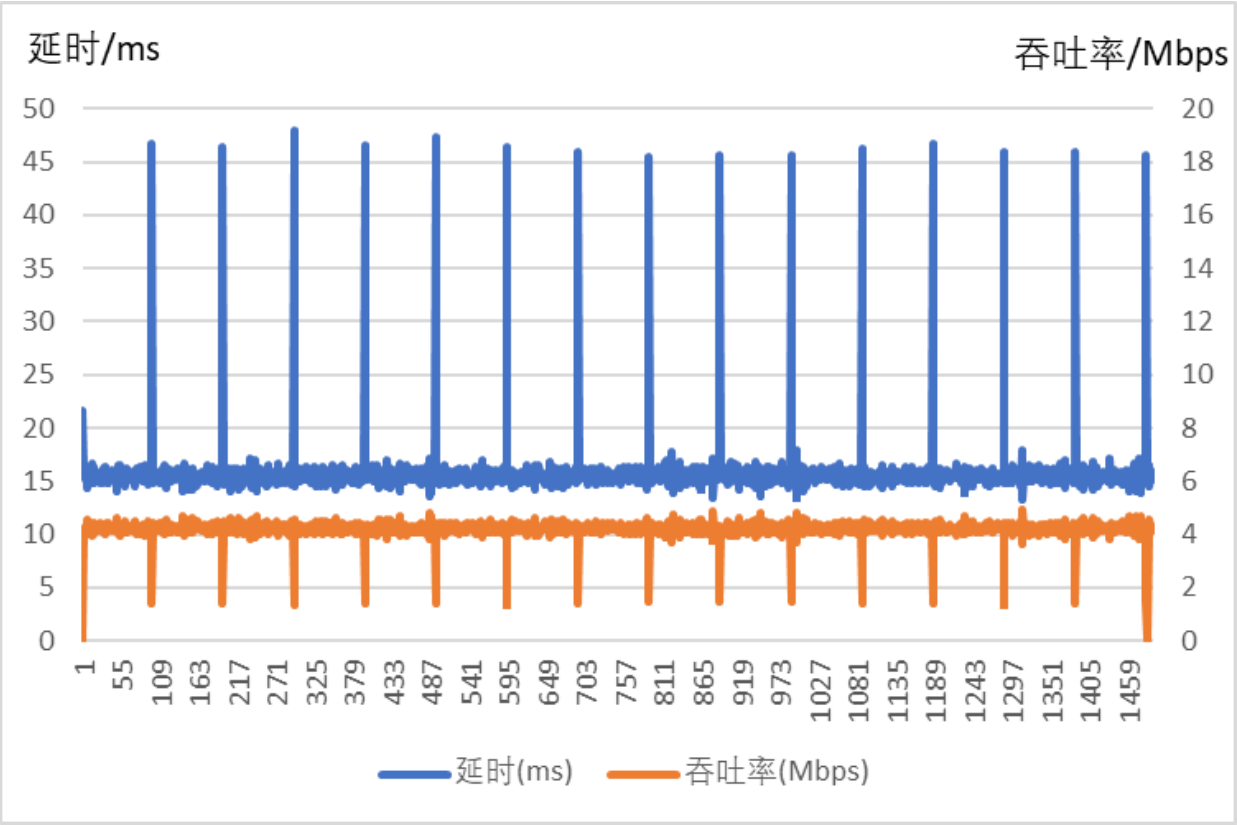
但是在发送文件过程中，主线程一致发送消息，对于我们额外添加的锁一直占用，使得子线程无法处理`ACK`，所以在发送消息的循环中添加一个`Sleep(1)`，使其在这段时间内无法持有锁。

实验结果

测试文件传输结果如下：



绘制延时与吞吐率结果如下：



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。