南开大学
Nankai University

南　开　大　学

网 络 空 间 安 全 学 院

计算机网络实验报告

3-3：基于 UDP 服务设计可靠传输协议并编程实现

聂志强 2012307

年级：2020 级

专业：信息安全

指导教师：徐敬东

2022 年 12 月 25 日

目录

一、 概述

(一) 实现功能

本次实验为在 UDP 上实现可靠数据传输，基于 3-2 实验添加了 RENO 算法，实现了以下 8 项基本功能和 4 项附加功能：

1. 基本功能

- 拥塞控制：在实验 3-2 的基础上，利用 RENO 算法实现拥塞控制，实现了超时检测和三次重复 ACK 检测。

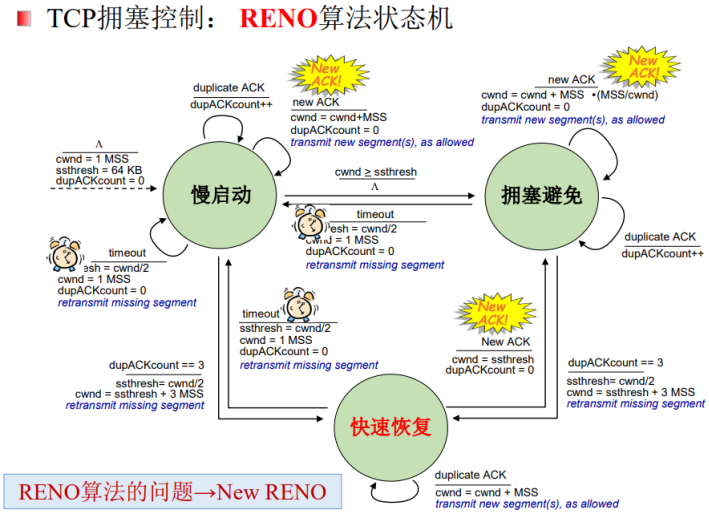

图 1: RENO

- 流量控制：在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，采用 GBN 方法。

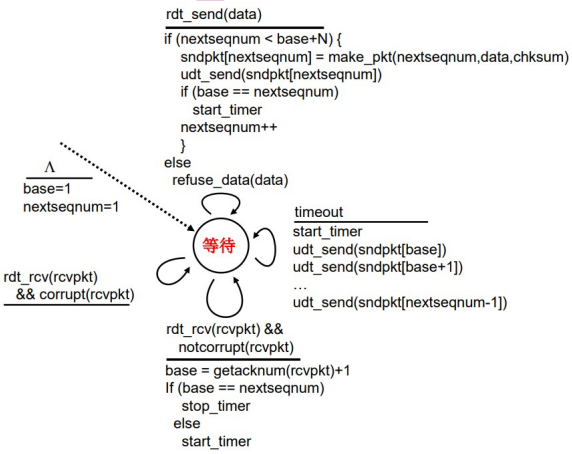

图 2: 流量控制

- 增加快速重传机制，接收到 3 个重复 ACK 时快速重传收到 ACK 序列号所指示的报文段。

- 拥塞控制：利用 RENO 算法实现拥塞控制，实现了慢启动阶段、拥塞避免阶段和快速恢复阶段机制。
- 建立连接：实现类似于 TCP 的三次握手、四次挥手过程。
- 差错检测：利用校验和进行差错检测，发送端将数据报看成 16 位整数序列，将整个数据报相加然后取反写入校验和域段，接收端将数据报用 0 补齐为 16 位整数倍，然后相加求和，如果计算结果为全 1，没有检测到错误；否则说明数据报存在差错。
- 确认重传：采用 rdt3.0 机制，由于通道既可能有差错，又可能有丢失，所以我们考虑利用 rdt3.0 机制实现可靠数据传输。
- 日志输出：打印出三次握手四次挥手过程、序列号、确认序列号、数据大小、时延、吞吐率、校验和、窗口信息（窗口左边界、Nextseqnum、窗口右边界）等。

2. **附加功能**

1. 采用多线程且采用全双工通信（两方均可发送和接收信息）
2. 使用了共享的临时缓冲区，为保证线程顺序读取数据，防止冲突，设置锁机制进行保护。
3. MSS 和 Window 大小双方协商，双方在握手期间协商 MSS 和 Window，选择双方需求的最小 MSS 和最小 Window 作为通信 MSS 和 Window。
4. 异常检测：断开方式与 TCP 基本相同，为了保证通信状态正常，在没有任何信息需要发送时，双方也会在固定的时间内发送一个小数据包，以检测连接状态和报告自身情况。当数据包出现 10 次连续丢失时，双方将认为通信异常，自动启动断开程序。

(二) 协议设计

1. **整体框架**

- 采用多线程控制，由发送线程和接收线程互相配合完成发送或者接收的任务。
- 由于使用了共享的临时缓冲区，为保证线程顺序读取数据，防止冲突，设置锁机制进行保护。
- 当用户需要发送信息时，将会把被发送的信息放到一个发送缓存中，发送线程将逐步读取发送的信息，然后进行可靠传输。
- 对于接收线程，线程将接收到的信息放入接收缓存中，当用户需要接收信息时，直接查看接收缓存里是否有内容即可，接收缓存有固定的大小。
- 当没有任何信息需要传输时，线程也会不断发送一个小信息包，以报告连接正常。

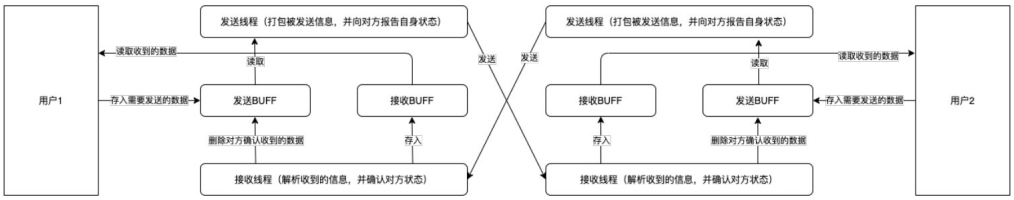

图 3: 整体框架

2. 协议头设计

用户数据报协议 UDP 为进程间通信提供非连接的、不可靠的传输服务；而传输控制协议 TCP 为进程间通信提供面向连接的、可靠的传输服务。为在 UDP 上实现面向连接的可靠传输，设计类似 TCP 的协议。协议格式为：

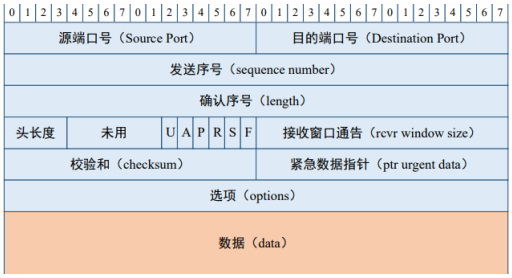

图 4: 协议格式

- 源端口：长度为 16 bits（2 个字节）。
- 目的端口：长度为 16 bits（2 个字节）。
- 序号部分 SEQ 用于标记此报文段数据部分首字节的字节流编号；确认序号 ACK 部分用于标记接收方期望收到的下一个字节的字节流编号。
- 首部长度用于标记协议头长度，本次协议中设置的最大段长度 MSS 在协议中，所以首部长度为 24。
- 预留字段：长度为 4bits，值全为零。
- 标志位：本次实验主要使用的有 ACK(ACK 有效)、END(最后一个报文段)、SYN(建立连接)、FIN(连接关闭)
- 窗口：长度 16bits（2 个字节），表示滑动窗口的大小，用来告诉发送端接收端的 buffer space 的大小。接收端 buffer 大小用来控制发送端的发送数据数率，从而达到流量控制，最大值为 65535。
- 校验和：用于进行差错检测，将整个报文和头部相加然后取反产生校验和。
- 最大段长度 MSS：用于双方协商 MSS，取双方要求的 MSS 最小值作为传输过程中的 MSS。

3. 协议头设计

- 可靠性设计采取收到-确认的方法，发送方标记数据包，接收方收到并确认无误后，向发送方说明已收到。
- 此协议中，序号部分用于标记此报文段数据部分首字节的字节流编号，确认号部分用于标记接收方期望收到的下一个字节的字节流编号。
- 发送方利用序号，对自己发送的数据包的编号进行标记，接收方利用确认号来说明已经成功收到了多少数据。

4. **连接的建立与断开设计**

- 本协议通过三次握手建立连接，同时双方将会协商 MSS，选择双方需求的最小 MSS 作为通信 MSS，同时也会协商窗口大小，保证双方都可以使用。
- 断开方式与 TCP 基本相同，为了保证通信状态正常，在没有任何信息需要发送时，双方也会在固定的时间内发送一个小数据包，以检测连接状态和报告自身情况。
- 当数据包出现 10 次连续丢失时，双方将认为通信异常，自动启动断开程序。

二、 代码实现

本节先将拥塞控制部分单独拿出进行详细代码及核心思想讲解，后续为其他各模块核心部分分析。下图为拥塞控制整体框架流程图:

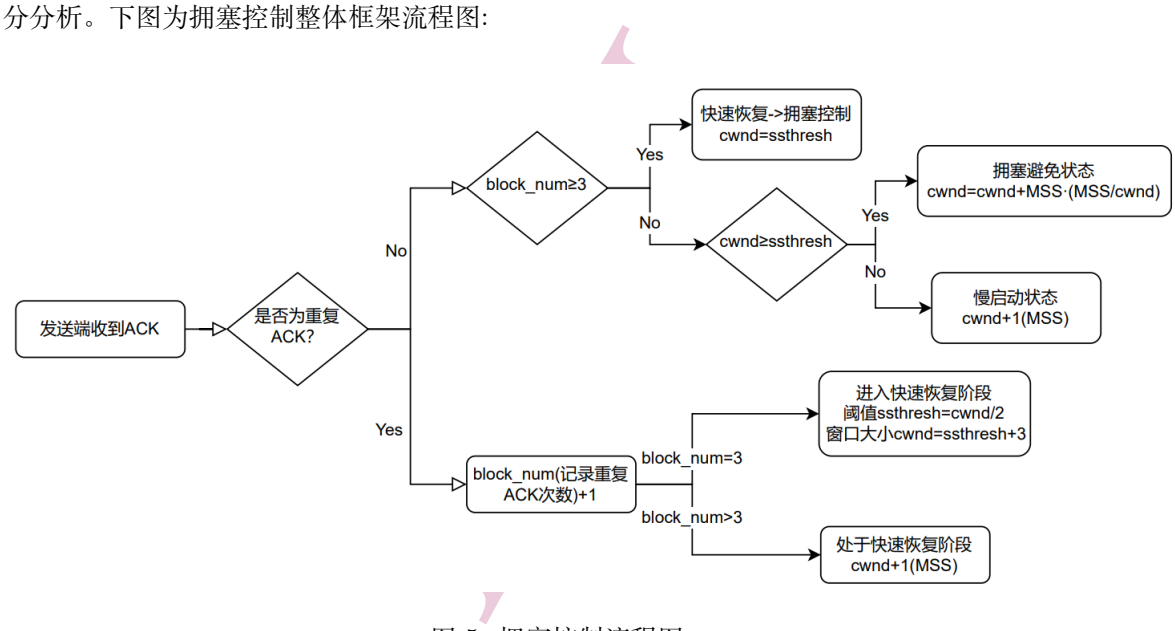

图 5: 拥塞控制流程图

(一) 拥塞控制

基于 RENO 算法实现拥塞控制，划分为慢启动、拥塞避免和快速恢复三个阶段，动态调整阈值和窗口大小以控制发送速率

1. TCP **拥塞控制——慢启动阶段**

- 初始拥塞窗口：cwnd=1(MSS)
- 每个 RTT，cwnd 翻倍（指数增长）
- 每接收一个 ACK，cwnd+1(MSS)
- 当**连接初始建立**或**报文段超时未得到确认**时，进入慢启动阶段

如下图代码所示，当获得一个新 ACK，先通过 block_num 重复 ACK 数量来判断所处状态，如果 block_num 3 则说明此时处于快速恢复状态，收到新 ACK 后开始进入拥塞控制状态，即 cwnd=ssthresh，否则说明此时处于慢启动或拥塞控制阶段，通过窗口大小 cwnd 和阈值大小 ssthresh 关系进行区分，如果窗口大小 cwnd ssthresh 则说明此时处于拥塞避免状态，即 cwnd=cwnd+MSS · (MSS/cwnd)，如果 cwnd<ssthresh 则说明此时处于慢启动阶段，即 cwnd+1(MSS)

拥塞避免阶段

```
if (cls->seq != cls->get_ack(buf)) {
    if (block_num >= 3)  //快速恢复 -> 拥塞避免
    {
        cout << "【快速恢复 -> 拥塞避免】" << endl;
        cls->window_size = cls->max_window_size;
    }
    else if (cls->window_size < cls->max_window_size)  // 慢启动阶段
        cls->window_size++;
    else {
        // 拥塞避免阶段: 拥塞窗口达到阈值时，进入拥塞避免阶段
        slow_num++;
        if (slow_num >= cls->window_size) {
            cls->window_size += 1;
            slow_num = 0;
        }
}
```

2. **拥塞控制——拥塞避免阶段**

- 阈值 ssthresh：拥塞窗口达到阈值时，慢启动阶段结束，进入拥塞避免阶段
- 每个 RTT，cwnd+1(线性增长)

代码分析同上

3. **拥塞控制——快速恢复阶段**

- ACK 重复次数等于 3【拥塞控制-> 快速恢复】:
  - 阈值 ssthresh=cwnd/2 阈值减为拥塞窗口的一半
  - cwnd=ssthresh+3
- ACK 重复次数大于 3【快速恢复阶段】:
  - cwnd+1(MSS)

如下图代码，如果为重复 ACK 则 block_num+1，当 block_num==3 时开始进入快速恢复状态，即阈值 ssthresh=cwnd/2 阈值减为拥塞窗口的一半，窗口大小 cwnd=ssthresh+3，当 block_num>3 则说明此时本身就处于快速恢复状态，即 cwnd+1(MSS)；

三次重复 ACK 检测丢失

```
if (length <= cls->head_length)
{
    // 三次重复ACK检测丢失：阈值减为拥塞窗口的一半，cwnd=SST+3，进入线性增长（拥塞避免阶段）
    block_num++;
    // 三次重复ACK，准备进入快速恢复
    if (block_num == 3) {
        slow_num = 0;
        cls->max_window_size = cls->window_size / 2;
        cls->window_size = cls->max_window_size + 3;
        seq_temp = min(cls->seq + cls->window_size*cls->max_send_size, seq_temp);
        cout << "三次重复 ACK!" << endl;
        cls->re = 1;
    }
    // 快速恢复阶段
    else if (block_num > 3)
    {
        cls->re = 1;
        cls->window_size += 1;
    }
}
```

4. **拥塞控制——超时检测**

- 阈值 ssthresh=cwnd/2 阈值减为拥塞窗口的一半
- cwnd=1，进入慢启动阶段

如下图代码，当接收线程检测到超时，阈值 SST 减为窗口的一半，cwnd=1，进入慢启动阶段，并记录超时次数，如果连续 10 次丢包或超时则自动断开

超时检测

```
if (length == -1)
{
    // 超时检测：阈值SST减为窗口的一半，cwnd=1,进入慢启动阶段
    timeout_round++;  //超时次数记录
    slow_num = 0;       // 拥塞避免阶段计数（用作窗口大小变化）
    block_num = 0;    //计算重复ACK次数（满三次进入快速恢复阶段）
    cls->max_window_size = cls->window_size / 2;
    cls->window_size = 1;
    cout << "Time Out!" << endl;
    // 10次丢失，通信异常，自动断开
    if (timeout_round >= cls->autoclose_tcp_loop)
        break;
    Sleep(0);
    }
```

(二) 接收线程

接收线程中，当超过设定的时限 CONNECT_RECV_TIMEOUT，则设置 time_flag 立即重传。当超过 10 次丢失，通信异常，自动断开连接，或者收到断开请求 FIN 标志置位也利用四次挥手断开连接。当发送端接收到新 ACK 时，移动窗口左端 base，并标记立即重传。

在接收线程处理接收数据的时，对数据报拆包去掉数据报头，更新 ACK，并将数据放入接收缓冲区，并设置立即重传使接收端回复 ACK。

由于使用了共享的临时缓冲区，为保证线程顺序读取数据，防止冲突，设置锁机制进行保护。

1. 接收线程——【发送端】窗口 base 移动 +【发送缓冲区】改变

此段代码仅为接收线程中（拥塞控制窗口大小动态变化）属于流量控制范畴内的部分，在此实验中具体阐述，其余窗口动态变化部分在（一）拥塞控制已做具体阐述。当发送端收到传回的 ACK 后，因为接收端采用累计确认的方式，因此默认该 ACK 之前的报文分组接收端已经正确收到，因此将发送端 windows 窗口 base 移动至 ACK 序列号所指报文段处。并且发送缓冲区将 ACK 所确认的报文段进行剔除，无需再保存在发送缓冲区。

接收线程——窗口移动 + 发送缓冲区数据改变

```
// 发送缓冲区begin指针后移
cls->sendbuf.begin()->assign(*(cls->sendbuf.begin()), cls->get_ack(buf) - cls
    ->seq);

// 当发送缓冲区中已经发完一个完整文件后，将该文件整个剔除
if (cls->sendbuf.begin()->length() == 0)
        cls->sendbuf.pop_front();
// 窗口移动
cls->seq = cls->get_ack(buf);

// 继续发送下一个报文分组
cls->immsend = true;
```

2. 接收线程——【接收端】解析报文数据并更新 ACK

接收线程中，当报文段长度大于数据报头的长度则为接收报文数据，通过 cls->ack == cls->get_seq(buf) 来判断是否为按序到达接收端的报文分组，如果按序到达则进行拆包，去掉数据报头并将数据放入接收缓冲区，更新 ACK 值，如果接收到的为失序分组或重复分组，则不更新 ACK，通过 cls->immsend = true 控制发送线程进行转发 ACK 给发送端，并打印日志信息。

接收线程——接收数据

```
if (length > cls->head_length) {
        if (cls->ack == cls->get_seq(buf)) {    //按序收到所需分组
                EnterCriticalSection(&(cls->recvbuf_lock));
                if (cls->recvbuf.max_size() > cls->recvbuf.size()) {
                        // 拆包: 去掉数据报头
                        unsigned char* temp = new unsigned char[length - cls
                            ->head_length];
                        memcpy(temp, buf + cls->head_length, length - cls->
                            head_length);
```

```
                        // 将数据放入接收缓冲区
                        cls->recvbuf.push_back({ length - cls->head_length,
                            cls->get_flag_end(cls->get_flag(buf)), temp });

                        // 更新ACK
                        // 如果接收到的是失序分组或者重复分组，则不更新ACK
                        cls->ack = cls->get_seq(buf) + length - cls->
                            head_length;
                }
                LeaveCriticalSection(&(cls->recvbuf_lock));
        }
        // 继续发送下一个报文分组
        cls->immsend = true;
        cout << "Recv: " << length - cls->head_length << " [SEQ] " << cls->
            seq << " [ACK] " << cls->ack << " [checksum] " << cls->
            get_checksum(buf) << endl;
}
```

3. **接收线程——recvmeg 函数设计**

接收线程中调用该函数进行报文接收，并添加超时检测、差错检测和丢包检测，超时检测通过 setsockopt 函数实现，

接收线程——【超时检测 + 差错检测 + 丢包检测】

```
int UDP::recvmeg(unsigned char* buf, size_t buf_size, int timeout) {
        int addr_length = sizeof(sockaddr);
        // 设置接收时限
        int tim = setsockopt(this->sock, SOL_SOCKET, SO_RCVTIMEO, (char*)&
            timeout, sizeof(timeout));
        // 接收数据报
        int result = recvfrom(this->sock, (char*)buf, buf_size, 0, (sockaddr
            *)(this->local_addr), &addr_length);
        // 差错检测
        if (result != -1 && !check_message(buf, result) && tim==-1) result =
            -1;
        return result;
}
```

(三) 发送线程

在发送线程中，当发送缓冲区不为空时，读取缓冲区数据打包数据报并发送，设置 END 标识来标记是否为最后一个数据报，打印相应的序列号信息。

1. **发送线程——控制【超时重传】【快速重传】【正常发送】【等待】**

具体解析见下面代码

发送线程——核心控制部分

```
unsigned long long last_stamp = GetTickCount64(); // 计时开始
while (cls->isconnect)
{
        if (!cls->immsend && GetTickCount64() - last_stamp <
            CONNECT_RECV_TIMEOUT) {
                Sleep(1);
                continue;
        }
        if (GetTickCount64() - last_stamp > CONNECT_RECV_TIMEOUT)
        {
                cout << "已超时" << endl;
                time_flag = 1;
        }
        //cout << "cls->immsend" << cls->immsend << "time" << time_flag<<
            endl;
        last_stamp = GetTickCount64();  //重新计时（超时/base移动且base!=
            nextseqnum/刚开始）
        cls->immsend = false;
        flag = 0;
        cls->set_flag_end(&flag, true); // 初始化为最后一个数据报
        cls->set_flag_ack(&flag, true); // ACK有效
        string sendcontent;   //发送内容
        EnterCriticalSection(&(cls->sendbuf_lock));  //加锁 接下来的代码处理
            过程中不允许其他线程进行操作，除非遇到LeaveCriticalSection
        if (cls->sendbuf.size())
        {
                string& sendpkg = *(cls->sendbuf).begin();
                int remain = cls->max_send_size * cls->window_size - (
                    seq_temp - cls->seq);
                //cout << "remain:" << remain << endl;
                if (time_flag )
                {
                        sendcontent.assign(sendpkg, 0, seq_temp - cls->seq);
                            //重发
                        seq_temp = cls->seq; // seq_temp代表nextseqnum
                }
                if (cls->re)
                {
                        cls->re = 0;
                        sendcontent.assign(sendpkg, 0, cls->max_send_size);
                             //快速重传
                        seq_temp = cls->seq; // seq_temp代表nextseqnum
                }
                else
                {
                        if (remain){
                                if (remain < sendpkg.length() - (seq_temp -
```

```
                                cls->seq))
41                                  sendcontent.assign(sendpkg, seq_temp
                                        - cls->seq, remain);
42                              else
43                                  sendcontent = sendpkg;
44                      }
45                      else{
46                              cout << "窗口内满了" << endl;
47                              Sleep(5);
48                              continue;
49                      }
50              }
51      }
52      LeaveCriticalSection(&(cls->sendbuf_lock));  //解锁
53
54      ......
55  }
```

2. **发送线程——【发送端】分组发送数据报 +【窗口】Nextseqnum 移动**

先判断该报文分组是否为最后一个,如果是的话将标志位 end 调用 3-1 中写好的 set_flag_end 函数更新标志位。设置随机丢包判断是否调用 sendmeg 函数发送该分组，并更改窗口中 Nextseqnum 位置，输出发送日志。

发送线程——发送数据报 +Nextseqnum 移动

```
1   unsigned char flag_copy = flag;
2   for (size_t i = 0; i < sendcontent.length(); i += cls->max_send_size) {
3           flag = flag_copy;
4           // 判断是否为最后一个报文分组
5           if (i + cls->max_send_size < sendcontent.length())
6                   cls->set_flag_end(&flag, false);
7           // 发送该报文分组
8           int len = ((i + cls->max_send_size) >= sendcontent.length() ?
                sendcontent.length() - i : cls->max_send_size);
9           if ((++book) % 100 || time_flag == 1)
10                  cls->sendmeg(sendcontent.substr(i, len), flag, &seq_temp);
11          else
12                  cout << "【该分组丢包】";
13          Sleep(1);
14          // 改变可用还未发送位置
15          seq_temp += len;
16          cout << "Send: " << len << " [SEQ](Base) " << cls->seq << "[
                nextseqnum]" << seq_temp << "[LimitWindow]"<< cls->seq+ cls->
                max_send_size * cls->window_size <<" [ACK] " << cls->ack << " [
                SST] " << cls->max_window_size << " [WSZ] " << cls->window_size
                << endl;
17  }
```

3. 发送线程——sendmeg 函数设计

我们对 UDP 进行封装，在发送数据 sendto 之前，打包数据报，调用 3-1 已经实现的函数 generate_meg_head 封装协议头，然后利用 sendto 将数据和协议头一起发送到接收端，同时在接受端接收数据时，设置了接收时限。

发送线程——打包并发送数据报

```
int UDP::sendmeg(const string& data, unsigned char flag, size_t* seq_spec) {
        // 打包数据报，将数据copy到message缓冲区，并封装协议头
        unsigned char* meg_buf = new unsigned char[this->head_length + data.
            length()];
        memcpy(meg_buf + this->head_length, data.data(), data.length());
        generate_meg_head(meg_buf, (size_t)(this->head_length + data.length()
            ), flag, seq_spec);

        // 发送数据报
        int result = sendto(this->sock, (const char*)meg_buf, (size_t)(this->
            head_length + data.length()), 0, this->addr, sizeof(sockaddr));
        delete[] meg_buf;
        return result;
}
```

(四) MSS 和 Windows 大小协商

本协议的连接建立方式和 TCP 基本相同，但是在建立连接的过程中，双方将会协商 MSS 和 Windows 大小，选择双方需求的最小 MSS 和 Windows_Size 作为通信 MSS 和窗口大小。

发送线程——服务器端和客户端协商

```
// 服务器端
// 协商MSS
if (get_max_send_size(buf) < this->max_send_size) {
        this->max_send_size = get_max_send_size(buf);
        recvbuf.resize(ceil(this->bufsize / (float)(this->max_send_size)));
}
// 协商窗口大小
if (get_window_size(buf) < this->window_size) {
        this->window_size = get_window_size(buf);
}

// 客户端
// 设置MSS和窗口大小
this->max_send_size = get_max_send_size(buf);
this->window_size = get_window_size(buf);
recvbuf.resize(ceil(this->bufsize / (float)(this->max_send_size)));
```

（五） 与缓冲区的交互

send 函数代表将数据放入发送缓冲区，recv 函数代表从接收缓冲区读取数据，并在读取期间通过加锁解锁避免其他线程操作

与缓冲区的交互

```
bool UDP::send(string data) {
        if (!this->isconnect)
                return false;
        this->sendbuf.push_back(data);
        return true;
}

string UDP::recv() {
        string res;
        while (this->isconnect) {
                EnterCriticalSection(&(this->recvbuf_lock));
                if (this->recvbuf.size() == 0) {
                        Sleep(0);
                        LeaveCriticalSection(&(this->recvbuf_lock));
                        continue;
                }
                bool isend = false;
                while (!isend && this->recvbuf.size()) {
                        auto buf = *(this->recvbuf.begin());
                        isend = buf.isend;
                        res += string((const char*)buf.buf, buf.size);
                        delete[] buf.buf;
                        this->recvbuf.pop_front();
                }
                LeaveCriticalSection(&(this->recvbuf_lock));
        }
        return res;
}
```

（六） 线程关闭

当发送缓冲区为空时关闭线程

关闭线程

```
void UDP::close() {
        while (this->sendbuf.size()) {
                Sleep(4);
        }
        this->isconnect = false;
        WaitForSingleObject(this->Send, INFINITE);
        WaitForSingleObject(this->Recv, INFINITE);
        CloseHandle(this->Send);
```

```
9           CloseHandle(this->Recv);
10          for (auto& i : recvbuf) delete[] i.buf;
11          this->sendbuf.clear();
12          this->recvbuf.clear();
13          reset();
14  }
```

三、 实验结果

此处我结合窗口动态大小变化 + 窗口【Base】【Nextseqnum】【LimitWindow】具体位置变化说明控制拥塞机制（含丢包检测 + 快速重传）

(一) 日志各变量说明 + 慢启动阶段

开始为慢启动阶段，初始窗口大小为 1，当接收到新 ACK 时，窗口增长 MSS。以第二个发送报文为例，此时【Send：2048】表示发送此报文分组数据长度为 2048（正好为 MSS），【SEQ(Base)】表示此使窗口左边界为 2049（接收到第一个报文返回的 ACK 值为 2049，因此窗口左端滑动到此位置），【Nextseqnum：4097】表示此时下一个可发送序列号为 4097（该报文发送的为 2049 4096），【LimitWindow：6145】表示此时窗口右边界为 6145（此时窗口大小为 2，则 2049+2*2048 = 6145），【SSTresh】表示拥塞控制 RENO 算法中的阈值（初始定义为 10），【Window_Size】表示此时窗口大小，由于使慢启动，发送完 1 个报文后 Window_Size：1->2，以此类推直至成功发送十个数据报【Window_Size】窗口大小变为 10 到达阈值后慢启动阶段才结束并进入拥塞控制阶段

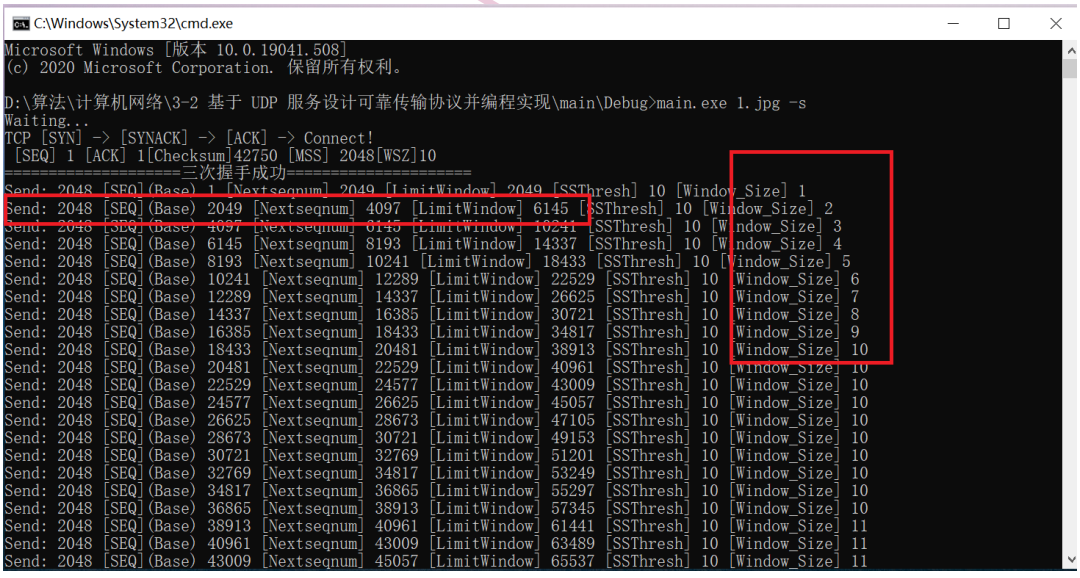

图 6: 慢启动阶段

(二) 拥塞控制阶段

当【Window_Size】窗口大小变为 10 到达阈值后进入拥塞控制阶段，cwnd=cwnd+MSS · (MSS/cwnd)，如下图所示连续发送十个数据报文后 Window_Size+1 变为 11，阈值仍保持不变

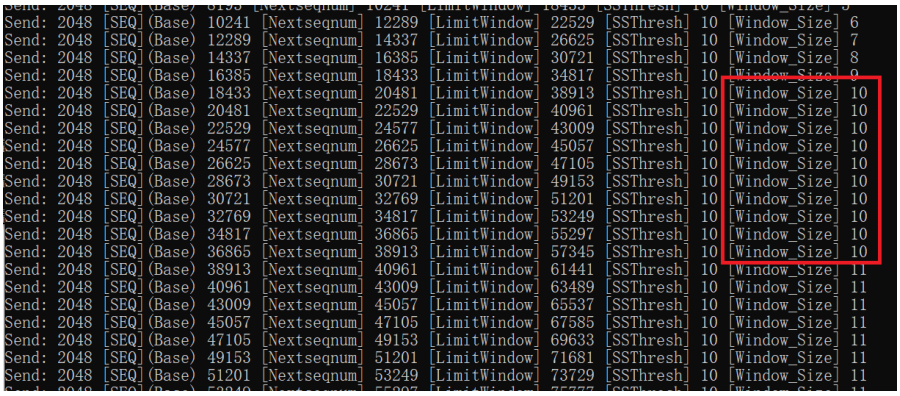

图 7: 拥塞控制阶段

(三) 丢包检测 + 快速恢复阶段

设置丢包，如图第一个红框所示，字节序为 202753 204081 的报文分组丢失时，此时发送端重复三次接收到 ACK：202753，输出日志“三次重复 ACK!”，此时进入快速回复阶段，如图第一个蓝框，阈值：16 -> 16/2=8, 窗口大小：16 -> 8+3=11，并且启动快速重传机制，重传字节序为 202753 204081 的报文分组（如最后一个红框所示），此时发送端接收到新 ACK：204081，此时由快速恢复阶段进入拥塞避免阶段，如最后一个蓝框所示：窗口大小被赋值为阈值大小 8，后面继续发送剩余发送缓冲区数据报。

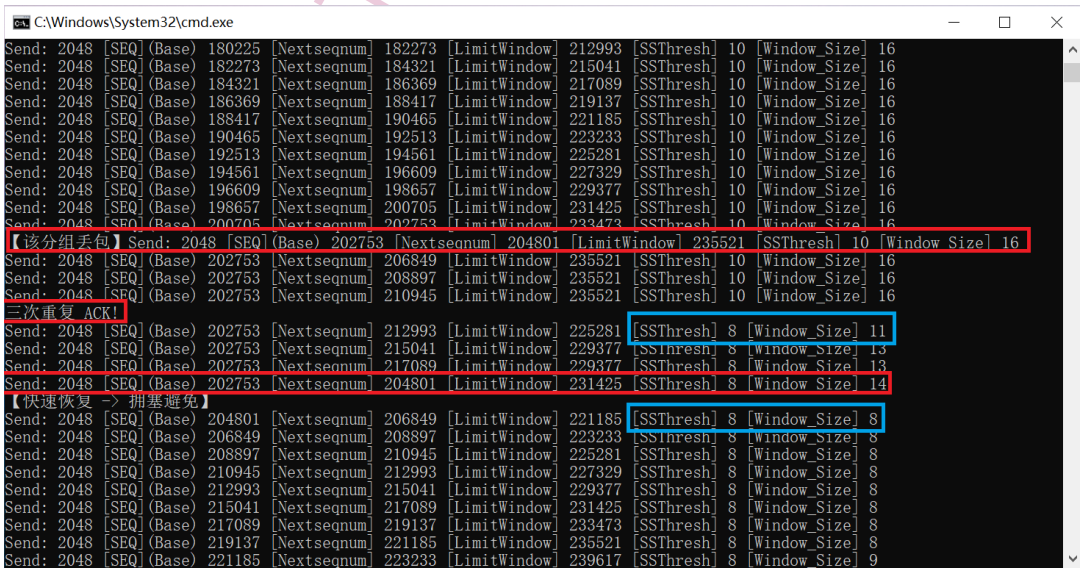

图 8: 丢包检测

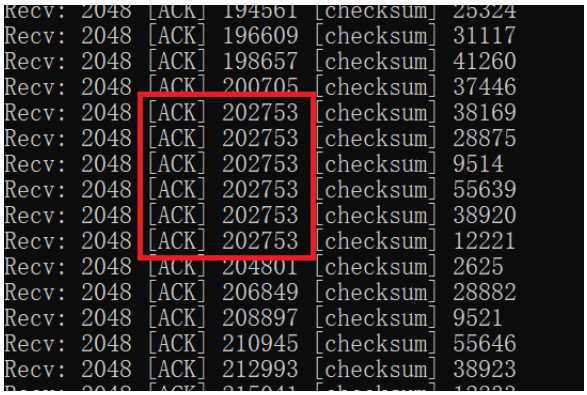

图 9: 丢包时接收端发送重复 ACK

（四） 超时检测【拥塞控制/快速回复-> 慢启动阶段】

设置延时导致超时产生，如黄色框所示输出日志”Time Out!”，如红色框所示超时前阈值【SSTresh】大小为 10，窗口【Window_Size】大小为 33，此时正处在拥塞控制阶段，当发生超时时此时转入慢启动阶段，阈值【SSTresh】减半变为 16，窗口大小变为 1，慢启动阶段中窗口【Window_Size】大小指数增长至阈值 16 再进入拥塞控制阶段。

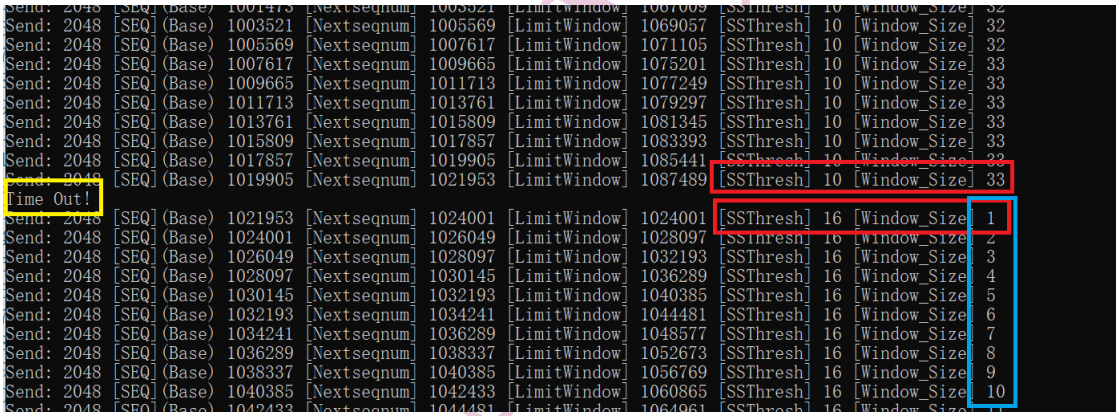

图 10: 超时检测

（五） 传输吞吐率对比

下图为设置不丢包和 10% 丢包率的传输对比，可以观察到当出现丢包情况时会影响传输时间进而影响吞吐率.

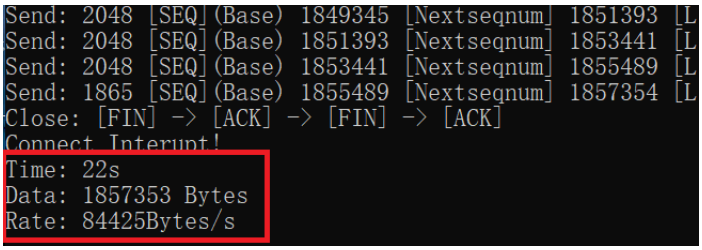

图 11: 不设置丢包

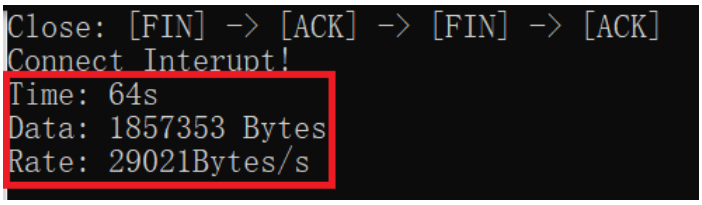

图 12: 设置 10% 丢包率

（六） 样例测试

下图为所提供测试样例的传输效果

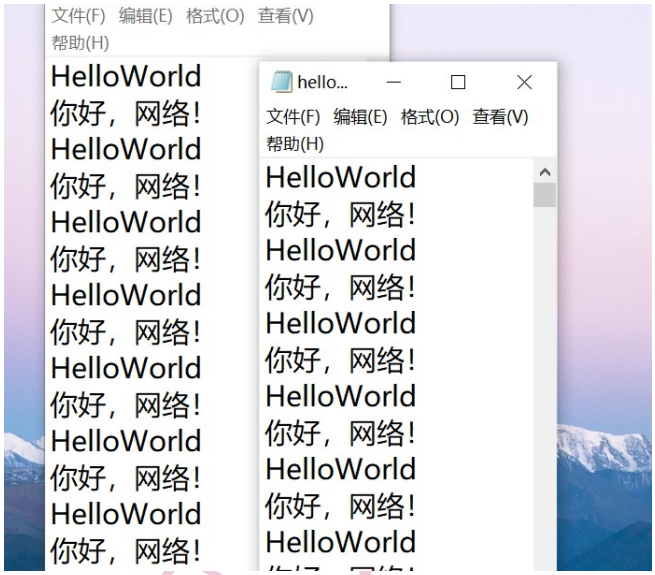

图 13: 测试样例 helloworld.txt 对比

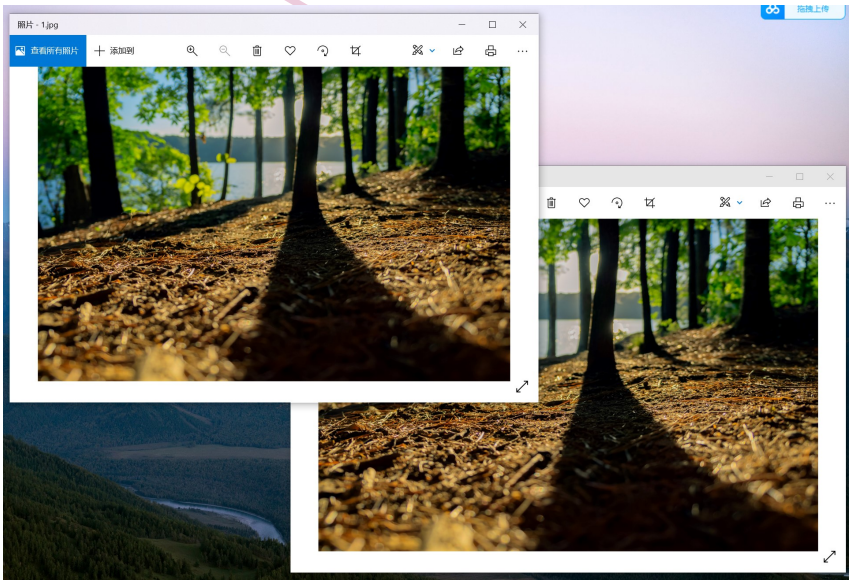

图 14: 测试样例 1.jpg 对比