计算机网络实验3.2-基于UDP服务设计可靠传输协议(流量控制)

写在前面：这次由于突然发烧隔离（虽然不是阳，也是挺重的一次感冒）导致没能及时检查作业，比较可惜。也借此提醒自己：一定要注意身体。另外也得接受教训。如金哥所言，提高应急处突能力还是非常重要的。

实验要求

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，采用固定窗口大小，支持累积确认，完成给定测试文件的传输。

- 多个序列号；
- 发送缓冲区、接受缓冲区；
- 滑动窗口：Go Back N；
- 有必要日志输出（须显示传输过程中发送端、接收端的窗口具体情况）。

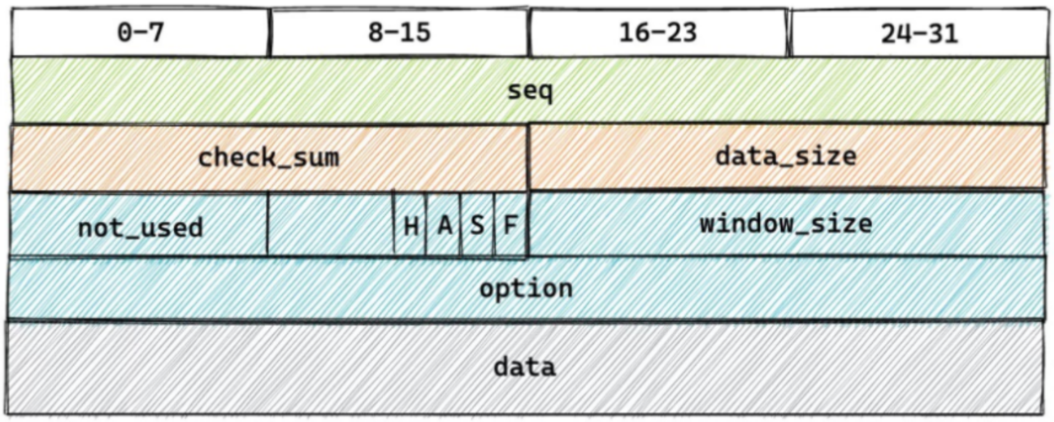
程序流程展示

注：与实验3.1相同的部分仅作简要叙述，详细可以参见计算机网络实验实验3.1

协议设计

基于rdt3.0,本次实验实现了GBN和SR两种流水线协议，并采用多线程编程。

报文结构



如图所示，报文头长度共`128Bits`。下面介绍报文结构如下所示：

整个实验只使用一个序列号字段。对于发送端对应`TCP`中的`seq`,接收端对应`TCP`中的`ack`。

下面是十六位校验和以及数据报字段长度，与`TCP`相同。

使用`u_short`来存放`flag`。其字段含义如下：

`F`:`FIN`

`S`:`SYN`

`A`:`ACK`

`H`:`FILE_HEAD`

`FILE_HEAD`用于指示接收端此报文包含文件信息的字段。

`window_size`存放接收端通告给发送端的窗口大小。

`option`为可选字段，在本次实验中暂时用于存放文件长度。

`data`的最大长度可以调节，本次实验定义为1024字节。

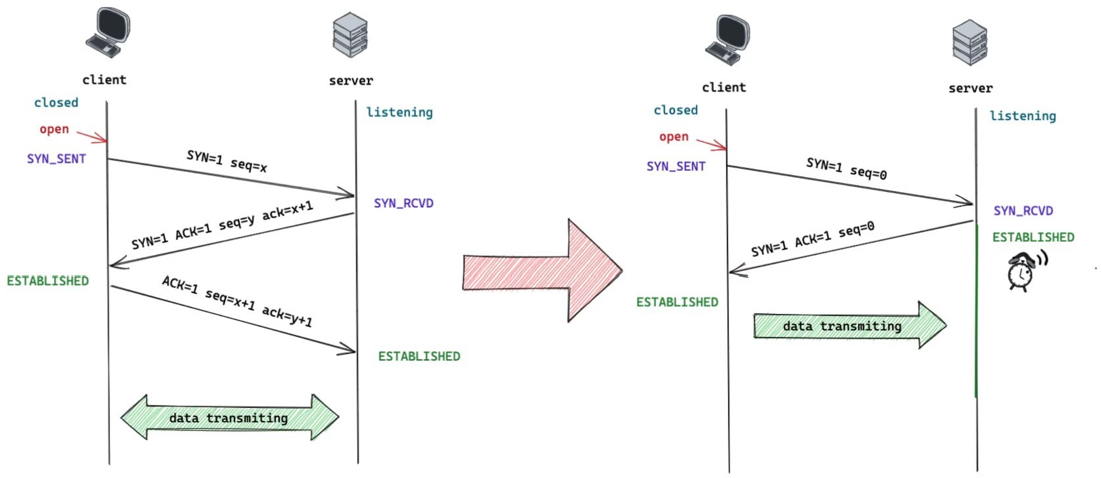
此部分定义代码段如下;

```
#define MAX_SIZE 1024
#define DATA 0x0
#define FIN 0x1
#define SYN 0x2
```

```
#define ACK 0x4
#define ACK_SYN 0x6
#define ACK_FIN 0x5
#define FILE_HEAD 0x8
// datagram format:
#pragma pack(1)
struct packet_head {
    u_int seq;
    u_short check_sum;
    u_short data_size;
    u_short flag;
    u_short window_size;
    u_int option;

    packet_head() {
        seq = 0;
        check_sum = 0;
        data_size = 0;
        flag = 0;
        window_size = 0;
        option = 0;
    }
};

struct packet {
    packet_head head;
    char data[MAX_SIZE]{};

    packet() {
        packet_head();
        memset(data, 0, MAX_SIZE);
    }
};
#pragma pack()
```

`#pragma pack(1)`用于指示结构体内容按1Byte对齐，以保证报文大小是我们期望的紧凑形式。

建连和断连



建连和断连过程与3.1无太大变化。主要是在建连过程中增加了接收方初始窗口大小的通告。

流程设计

程序支持一次建连发送多个文件。

本次实验与上次相比，整体逻辑没有变化，但序列号递增使用。在本次实验中序列号为`u_int`类型，可存储2^{32}个序列号，每个数据包最大为1024即2^{10}字节，故最大可传输$2^{42}=8TB$的单个文件，故不需循环使用即可满足需求。当需求确定时，整个系统是为可用性和简洁性而不是复杂性服务的。

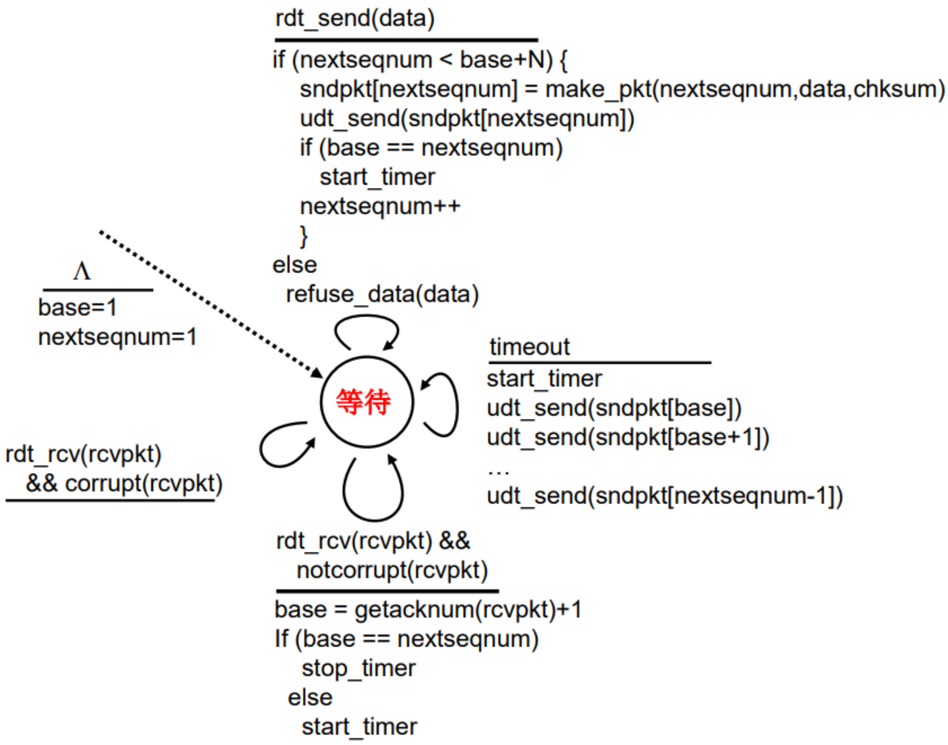
程序代码解释

文件发送过程

GBN

下面先以基础的GBN为基础分析程序代码：

发送端



课件中讲述也很直观，对照伪代码实现即可。

发送进程（主进程）

```
//wasted space but saved time for "shifting" sndpkt window
auto *sndpkt = new packet[pkt_total + 1];
while (base < pkt_total) {
    //send packets
    if (nextseqnum < base + N && nextseqnum < pkt_total) {
        pkt_data_size = min(MAX_SIZE, file_len - nextseqnum * MAX_SIZE);
        sndpkt[nextseqnum] = make_pkt(DATA, nextseqnum, pkt_data_size, file_data + nextseqnum * MAX_SIZE);
        udt_send(sndpkt[nextseqnum]);
        cout << "Sent packet " + to_string(nextseqnum) + " ";
        if (base == nextseqnum) {
            timer.start_timer();
```

```
        }
        nextseqnum++;
        print_window();
    }
    //handle timeout
    if (timer.timeout()) {
        print_message("Timeout, resend packets from " +
to_string(base) + " to " + to_string(nextseqnum - 1),
                      WARNING);
        for (u_int i = base; i < nextseqnum; i++) {
            udt_send(sndpkt[i]);
        }
        timer.start_timer();
    }
}
```

这一部分逻辑与状态机中右上角两个部分完全一致。其中`timer`是一个全局的计时器，为自己编写的类。用法可以顾名思义。`print_window();`按照`[base|nexeseqnum|base+n]`的格式将当前窗口状态打印出来。

接收进程

接收进程在发送之前创建。实现如下：

```
DWORD WINAPI handle_ACK(LPVOID lpParam) {
    packet rcvpkt;
    while (true) {
        //non-blocking rdt_rcv (if not rcv any packet it, it will
return 0)
        while (!rdt_rcv(rcvpkt) || corrupt(rcvpkt) ||
!isACK(rcvpkt)) {
            //the packet must be ACK and not corrupt to jump out
of the loop
        }
//          base = get_ack_num(rcvpkt) + 1;
        acked[get_ack_num(rcvpkt)] = true;
        while (acked[base]) {
            base++;
```

```
        }
        cout << "Received ACK " + to_string(get_ack_num(rcvpkt)) +
" ";
        print_window();
        if (base == pkt_total) {
            return 0;
        }
        if (base == nextseqnum) {
            timer.stop_timer();
            continue;
        } else {
            timer.start_timer();
        }
    }
}
```

这一部分助教思考的也非常深入：虽然实验环境下ACK不会丢失且能按序到达，但真实的网络环境下ACK也会丢失，且由于传输速度可能不一样快，ACK未必是按序到达的。这一点在上一次实验的握手建连部分考虑到了且有所叙述（接收方在确认握手成功后也有可能因为ACK丢失而收到发送方重发的握手包），但这一次实验囿于伪代码的惯性思维没有考虑周全，感谢助教提醒指正。

因此此处可参考后续SR的实现，移动时不能简单的移动到当前收到的序号+1，而是从base开始移动到按序收到的包序号之后，即：

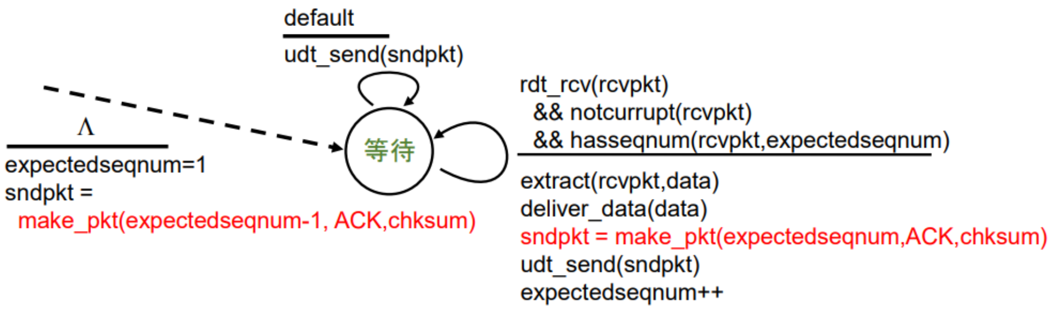
```
//          base = get_ack_num(rcvpkt) + 1;
        acked[get_ack_num(rcvpkt)] = true;
        while (acked[base]) {
            base++;
        }
```

当然这也需要把收到包的状态保存为数组。此处的空间复杂度还可以优化，也即实际上我们只需要窗口内的ACK状态即可，但这样就伴随着较为费时的“移动”。因此此处直接将所有状态都保存了下来。

接收端

接收端的变化主要一个是将窗口通告给发送方。这个任务只需要修改`make_pkt`即可，不再赘述。

另外一个是接收发送方数据的逻辑。



代码如下所示:

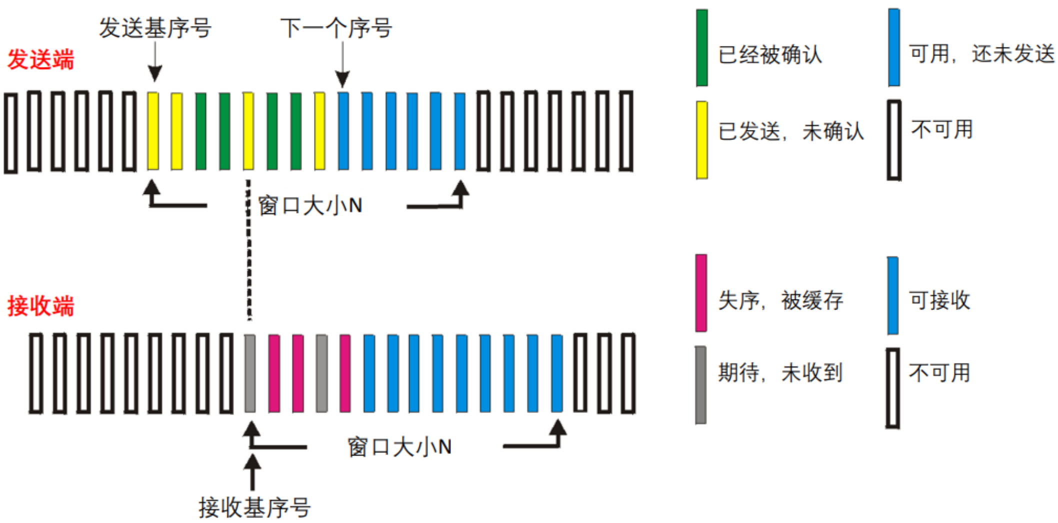
```
while (rdt_rcv(rcvpkt)) {
    if (not_corrupt(rcvpkt)) {
        if (hasseqnum(rcvpkt, expectedseqnum)) {
            pkt_data_size = rcvpkt.head.data_size;
            memcpy(file_buffer + received_file_len, rcvpkt.data,
pkt_data_size);
            received_file_len += pkt_data_size;
            packet sndpkt = make_pkt(ACK,expectedseqnum);
            udt_send(sndpkt);
            print_message("Received packet " +
to_string(expectedseqnum), DEBUG);
            expectedseqnum++;
        } else {
            //discard the packet and wait for the next one
            print_message("Received a out-of-order packet",
WARNING);
            continue;
        }
    } else {
        print_message("Received a corrupt packet", DEBUG);
```

```
            continue;
        }
        if (received_file_len == file_size) {
            ...file received, writing file to the disk...
        }
    }
```

可以从状态机看到，我们只需要关注`expectedseqnum`对应的包即可，其他的包收到直接丢弃即可。这一部分并不复杂，对照状态机容易理解。

SR

在这次实验中也额外实现了SR选择重传流水线协议。



- 发送端
  - **接收上层数据：**如果发送窗口中有可用的序号，则发送分组
  - **超时(n)：**重传分组n，重启定时器
  - **接收ACK(n)：**n在[send_base, send_base+N-1]区间，将分组n标记为已接收，如果是窗口中最小的未确认的分组，则窗口向前滑动，基序号为下一个未确认分组的序号
- 接收端：**接收分组n：**
  - n在[rcv_base, rcv_base+N-1]区间，发送ACK(n)，缓存失序分组，按序到达的分组交付给上层，窗口向前滑动
  - n在[rcv_base-N, rcv_base-1]区间，发送ACK(n)

发送端

发送端相比GBN，需要为每一个分组都加入定时器。一开始是希望每一个包发出后都创建一个相同进程用于监听ACK并处理超时事件，但实践中发现由于，传入的`packet`地址会在发送过程中被修改，在线程中无法正确每个线程获取需要等待的分组的序号和内容。因此转而采用线程池的方式进行实现，每个线程维护自己的定时器和需要重发的分组。代码如下所示：

```
//wasted space but saved time for "shifting" sndpkt window
clock_t single_file_timer = clock();
while (base < pkt_total) {
    //send packets
    if (nextseqnum < base + N && nextseqnum < pkt_total) {
        pkt_data_size = min(MAX_SIZE, file_len - nextseqnum * 
MAX_SIZE);
        sndpkt_buffer[nextseqnum] = make_pkt(DATA, nextseqnum, 
pkt_data_size,
                                             file_data + 
nextseqnum * MAX_SIZE);
        udt_send(sndpkt_buffer[nextseqnum]);
        cout << "Sent packet " + to_string(nextseqnum) + " ";
        print_window();
        thread_pool[nextseqnum] = CreateThread(nullptr, 0, SR, 
&sndpkt_buffer[nextseqnum], 0, nullptr);
        nextseqnum++;
    }
}
```

其中`SR`线程的实现思路如下：

`acked`数组中保存了当前分组确认的状态，和GBN中所叙述的类似。这个数组的内容由接收线程（与GBN相同，但只标记`acked`数组状态，不进行窗口滑动）进行修改。当第`wait_seq`个包对应的SR线程发现监听到了对应`acked`数组变化（即接收`ACK(n)`）判断是否进行窗口滑动。若超时，则重发对应的包，重启定时器。

下面是对应的代码：

```
DWORD WINAPI SR(LPVOID lpParam) {
    packet sndpkt = *reinterpret_cast<packet *>(lpParam);
    u_int wait_seq = sndpkt.head.seq;
    int resend_times = 0;
    //start a timer
    clock_t start = clock();
    while (!acked[wait_seq]) {
        if (timeout(start)) {
            udt_send(sndpkt);
            start = clock();
            if (resend_times > MAX_RESEND_TIMES) {
                print_message("Resend times exceed the limit, 
there must be something wrong with the network", ERR);
                return 1;
            } else {
                cout << "Resend packet " + 
to_string(sndpkt.head.seq) + " ";
                print_window();
                resend_times++;
            }
        }
    }
    //if reach here, the packet is ACKed
    if (wait_seq == base) {
        //if the ACKed packet is the base, move the window to the 
first unACKed packet
        while (acked[base]) {
            base++;
        }
    }
    return 0;
}
```

接收端

接收端比GBN的情况要复杂一些，因为需要缓存失序的包。实现如下：

```
//"blocking receive" here
while (rdt_rcv(rcvpkt)) {
    if (not_corrupt(rcvpkt)) {
        u_int pkt_seq = rcvpkt.head.seq;
        if (pkt_seq >= rcv_base && pkt_seq < rcv_base + N) {
            //in the window
            if (!acked[pkt_seq]) {
                if (pkt_seq == rcv_base) {
                    //the first packet in the window
                    pkt_data_size = rcvpkt.head.data_size;
                    memcpy(file_buffer + pkt_seq * MAX_SIZE,
rcvpkt.data, pkt_data_size);
                    acked[pkt_seq] = true;
                    packet sndpkt = make_pkt(ACK, rcv_base);
                    udt_send(sndpkt);
                    print_message("Received packet " +
to_string(pkt_seq), DEBUG);
                    //slide the window
                    while (acked[rcv_base]) {
                        rcv_base++;
                    }
                } else {
                    //not the first packet in the window, cache it
                    pkt_data_size = rcvpkt.head.data_size;
                    memcpy(file_buffer + pkt_seq * MAX_SIZE,
rcvpkt.data, pkt_data_size);
                    acked[pkt_seq] = true;
                    packet sndpkt = make_pkt(ACK, pkt_seq);
                    udt_send(sndpkt);
                    print_message("Received packet " +
to_string(pkt_seq)+", cached", DEBUG);
                }
            } else {
                //already acked in the window, resend the ack
```

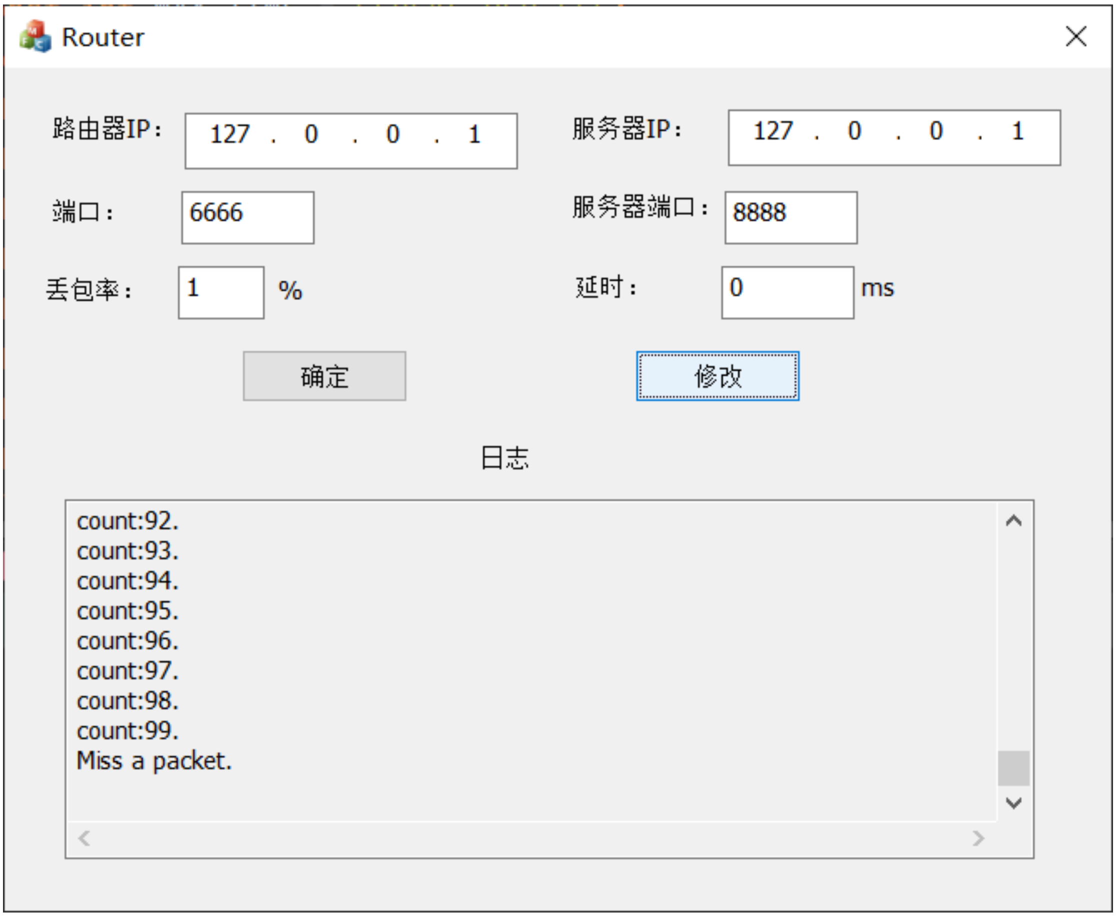
```
                print_message("Received packet " +
to_string(pkt_seq) + " again", WARNING);
                //send ack
                packet sndpkt = make_pkt(ACK, pkt_seq);
                udt_send(sndpkt);
                print_message("Sent ack " + to_string(pkt_seq),
DEBUG);
            }
        } else if ((pkt_seq >= rcv_base - N) && (pkt_seq <
rcv_base)) {
            //out of the window, but in the buffer
            print_message("Received packet " + to_string(pkt_seq)
+ " again", WARNING);
            //send ack
            packet sndpkt = make_pkt(ACK, pkt_seq);
            udt_send(sndpkt);
            print_message("Sent ack " + to_string(pkt_seq),
DEBUG);
        } else {
            //out of the window and buffer
            print_message("Received packet " + to_string(pkt_seq)
+ " out of the window", WARNING);
            //do nothing
        }
    } else {
        print_message("Received a corrupt packet", DEBUG);
        continue;
    }
    if (rcv_base * MAX_SIZE >= file_size) {
        ...file received, writing file to the disk...
    }
}
```

虽然代码较长，但无非是区分了接收的分组在窗口内，窗口之前或者超出了缓冲区三种情况。

程序演示

建立连接

路由器设置:



流程展示在上一次实验已经展示的比较完善。本次实验主要演示有丢包延时条件下的发送情况。

GBN流水线协议

可以看到发送方当发现超时候会将`base`到`nextseqnum`之间的包全部重发。

```
Sent packet 170 [162|171|172]
Received ACK 161 [162|171|172]
Sent packet 171 [162|172|172]
Received ACK 162 [163|172|173]
Sent packet 172 [163|173|173]
Received ACK 163 [164|173|174]
Sent packet 173 [164|174|174]
Received ACK 164 [165|174|175]
Sent packet 174 [165|175|175]
Received ACK 165 [166|175|176]
Sent packet 175 [166|176|176]
[WARNING] Timeout, resend packets from 166 to 175
Received ACK 166 [167|176|177]Sent packet 176 [167|177|177]

Received ACK 167 [168|177|178]
Sent packet 177 [169|178|179]
```

接收方也能判断失序的分组:

```
[DEBUG] Received packet 165
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[WARNING] Received a out-of-order packet
[DEBUG] Received packet 166
[DEBUG] Received packet 167
[DEBUG] Received packet 168
[DEBUG] Received packet 169
[DEBUG] Received packet 170
[DEBUG] Received packet 171
```

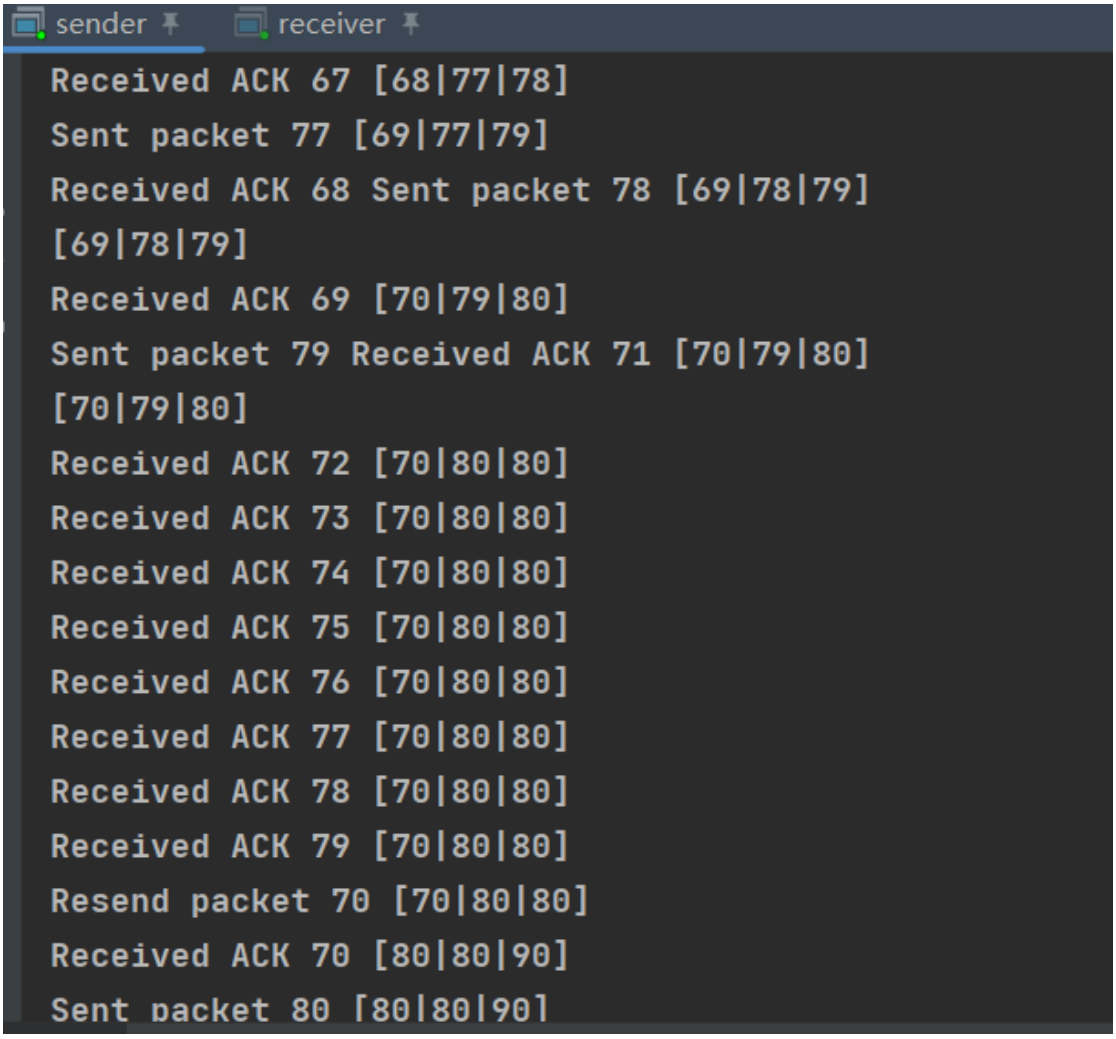
SR流水线协议

在SN协议的实践中，发现发送方会经常由于超时重发接收方能够收到的包。猜测是由于接收方串行接收且接收的逻辑比发送方发送的逻辑复杂，导致了接收的速度跟不上发送的速度。

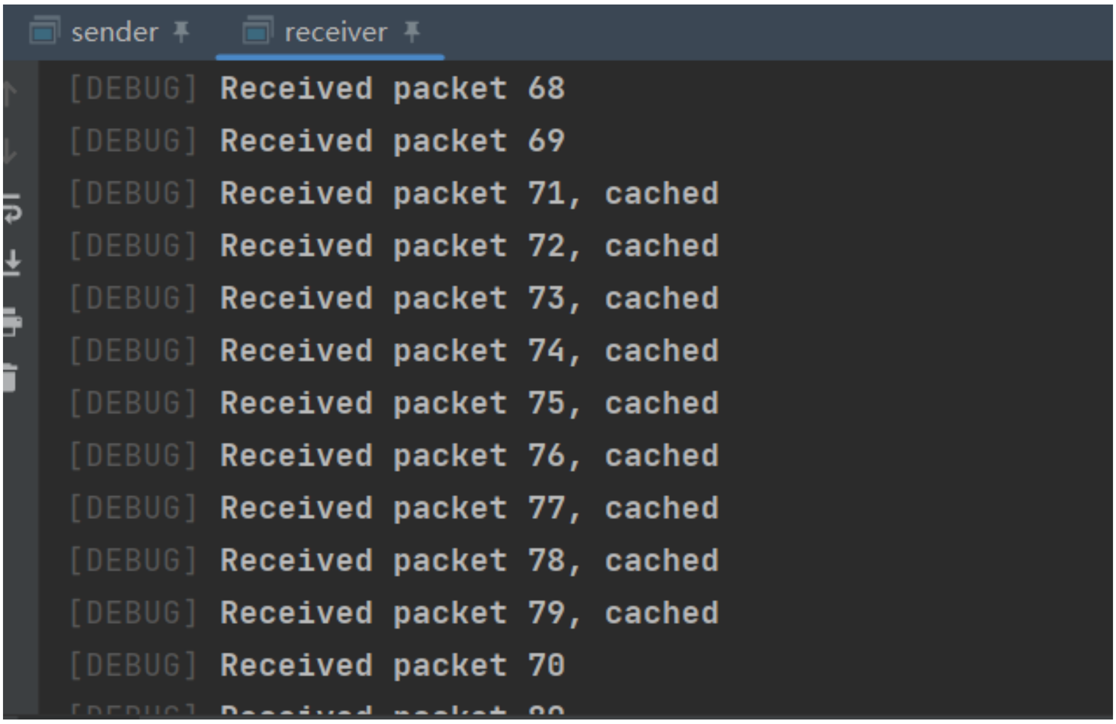
```
[DEBUG] Received packet 18
[DEBUG] Received packet 19
[DEBUG] Received packet 20
[DEBUG] Received packet 22, cached
[DEBUG] Received packet 23, cached
[DEBUG] Received packet 24, cached
[DEBUG] Received packet 25, cached
[WARNING] Received packet 16 again
[DEBUG] Sent ack 16
[WARNING] Received packet 17 again
[DEBUG] Sent ack 17
[WARNING] Received packet 18 again
[DEBUG] Sent ack 18
[WARNING] Received packet 19 again
[DEBUG] Sent ack 19
[DEBUG] Received packet 26, cached
```

```
Received ACK 13 [14|22|24]
Sent packet 23 [14|23|24]
Received ACK 14 Sent packet 24 [15|24|25]
[15|24|25]
Received ACK 15 [16|25|26]
Sent packet 25 [16|25|26]
Resend packet 16 Resend packet 17 Resend packet 18 [16|26|26]
[16|26|26]
[16|26|26]
Resend packet 19 Received ACK 16 Sent packet 26 [17|26|27]
[17|26|27]
Resend packet 20 [17|26|27]
[17|27|27]
Received ACK 17 [18|27|28]
Sent packet 27 [19|27|29]
Received ACK 18 Sent packet 28 [19|28|29]
```

因此尝试略微调大超时时间（3*MAX_TIME）可以看到正常的接收发送过程。



```
Received ACK 67 [68|77|78]
Sent packet 77 [69|77|79]
Received ACK 68 Sent packet 78 [69|78|79]
[69|78|79]
Received ACK 69 [70|79|80]
Sent packet 79 Received ACK 71 [70|79|80]
[70|79|80]
Received ACK 72 [70|80|80]
Received ACK 73 [70|80|80]
Received ACK 74 [70|80|80]
Received ACK 75 [70|80|80]
Received ACK 76 [70|80|80]
Received ACK 77 [70|80|80]
Received ACK 78 [70|80|80]
Received ACK 79 [70|80|80]
Resend packet 70 [70|80|80]
Received ACK 70 [80|80|90]
Sent packet 80 [80|80|90]
```



```
[DEBUG] Received packet 68
[DEBUG] Received packet 69
[DEBUG] Received packet 71, cached
[DEBUG] Received packet 72, cached
[DEBUG] Received packet 73, cached
[DEBUG] Received packet 74, cached
[DEBUG] Received packet 75, cached
[DEBUG] Received packet 76, cached
[DEBUG] Received packet 77, cached
[DEBUG] Received packet 78, cached
[DEBUG] Received packet 79, cached
[DEBUG] Received packet 70
```

发送成功！

```
[DEBUG] Received packet 1509
[DEBUG] Received packet 1510
[DEBUG] Received packet 1511
[DEBUG] Received packet 1512
[DEBUG] Received packet 1513
[DEBUG] Received packet 1514
[DEBUG] Received packet 1515
[SUCCESS] Received file successfully
[INFO] Time used: 32831ms
[SUCCESS] File saved to C:\Users\LENOVO\Desktop\rdt_v1\receiver_files\helloworld.
[TIP] Received 1 files till now
[INFO] Waiting for file info
[ERROR] Timeout, no packet received
[ERROR] Timeout when waiting for file info
```

GitHub:

https://github.com/Lunaticsky-tql/rdt_on_udp

GBN对应lab3.2分支，SR对应lab3.2-SR分支。