计算机网络实验3.3-基于UDP服务设计可靠传输协议(拥塞控制)

实验要求

- 在实验3-2的基础上，选择实现一种拥塞控制 算法，也可以是改进的算法，完成给定测试文件的传输。
- RENO算法；
- 也可以自行设计协议或实现其他拥塞控制算法；
- 给出实现的拥塞控制算法的原理说明；
- 有必要日志输出（须显示窗口大小改变情况）。

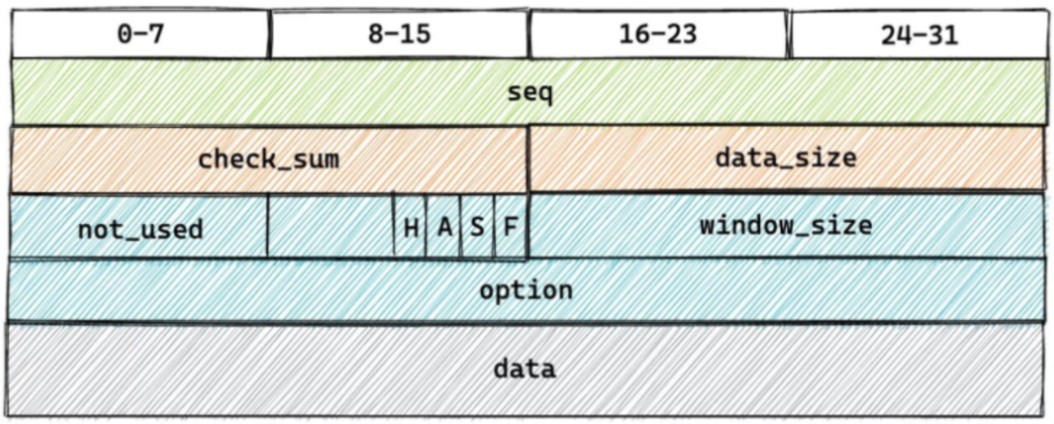
程序流程展示

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,本次实验在GBN的基础上实现了RENO拥塞控制算法，并通过令接收方缓存失序包验证了快速重传的正确性和有效性。

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。

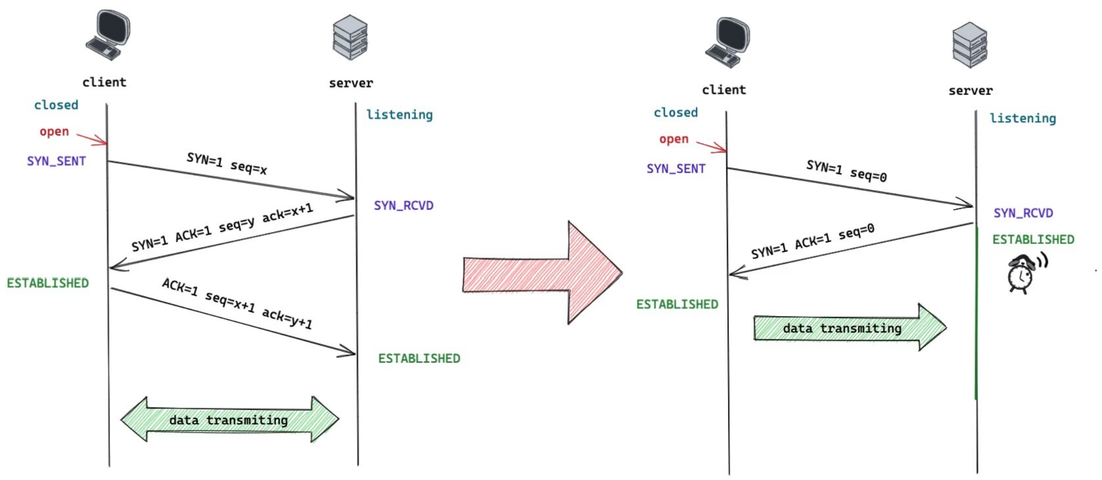
`window_size`存放接收端通告给发送端的窗口大小。

`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。

此部分代码段的定义可参阅源代码或`3.1`部分的报告。

建连和断连



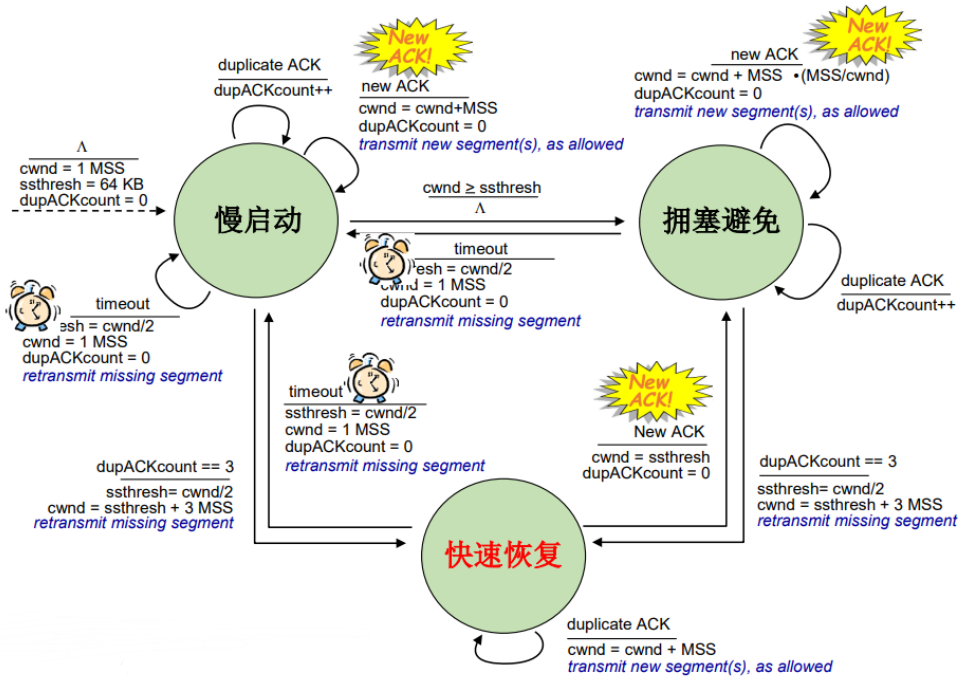
建连和断连过程与3.1无太大变化。主要是在建连过程中增加了接收方初始窗口大小的通告。

程序代码解释

文件发送过程

发送端

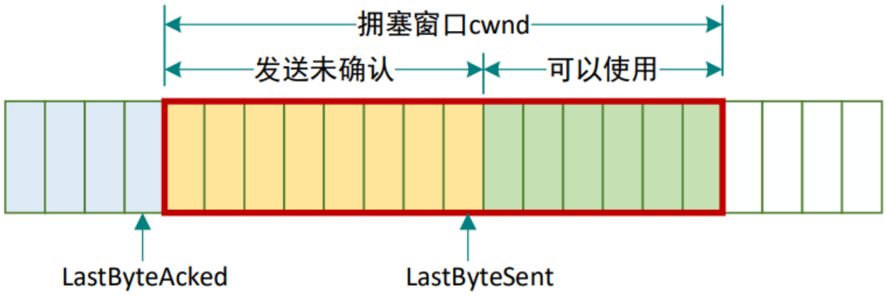
在本次实验中发送端实现的是基础的RENO算法.状态机如下所示:



下面将结合代码进行拥塞控制算法具体的原理及实现。

发送分组

首先由于拥塞控制算法中窗口大小计算是以字节为单位的，因此本次实验中计算和展示窗口大小时也改为字节计数（而不是按分组计数），如下图所示：



LastByteSent - LastByteAcked ≤ CongestionWindow (cwnd)

实际发送窗口取决于接收通告窗口和拥塞控制窗口中较小值

在发送时，只需要添加“实际发送窗口取决于接收通告窗口和拥塞控制窗口中较小值”对应的代码即可，对应于第一行：

```
//send packets
window_size = min(cwnd, advertised_window_size);
if ((LastByteSent - LastByteAcked < window_size) && (LastByteSent
< file_len)) {
    pkt_data_size = min(MAX_SIZE, file_len - nextseqnum *
MAX_SIZE);
    sndpkts[nextseqnum] = make_pkt(DATA, nextseqnum,
pkt_data_size, file_data + nextseqnum * MAX_SIZE);
    udt_send(sndpkts[nextseqnum]);
    cout << "Sent packet " + to_string(nextseqnum) + " ";
    if (base == nextseqnum) {
        timer.start_timer();
    }
    nextseqnum++;
    LastByteSent += pkt_data_size;
    print_window();
}
```

拥塞控制相关的逻辑主要在于接收ACK和超时时的窗口变化。下面将着重讲解。

接收ACK

首先我们先来看处理正常收到ACK的情况：

上一个实验中提到，实际网络环境中`ACK`未必是按序到达的，因此将`base`的移动改为判断按照按序接收到ACK为标准进行滑动，如下所示：

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

但是，实际上注意到，当接收方发出一个更大的ACK，说明之前的ACK事实上已经按序收到。当发送方收到一个更大的ACK时，不管之前的ACK有没有返回到发送方，发送方仅通过这个信息就已经可以确信接收方按序收到了之前的ACK了。因此其实是可以放心的进行`base = get_ack_num(rcvpkt) + 1`的。

之后，我们据此更新`LastByteAcked`，然后根据状态机对RENO算法的状态进行更新，最后更新窗口大小。据此，代码如下所示：

```
        u_int ack_num = get_ack_num(rcvpkt);
    if (ack_num >= base) {
            u_int gap = ack_num - base + 1;
            //update the base and LastByteAcked
            for (int i = 0; i < gap; i++) {
                LastByteAcked += sndpkts[base + i].head.data_size;
            }
            base = ack_num + 1;
            switch (RENO_STATE) {
                case SLOW_START:
                    cwnd += gap * MSS;
                    dupACKcount = 0;
                    if (cwnd >= ssthresh) {
                        RENO_STATE = CONGESTION_AVOIDANCE;
                    }
                    break;
                case CONGESTION_AVOIDANCE:
```

```
                    cwnd += gap * MSS * MSS / cwnd;
                    dupACKcount = 0;
                    break;
                case FAST_RECOVERY:
                    cwnd = ssthresh;
                    RENO_STATE = CONGESTION_AVOIDANCE;
                    dupACKcount = 0;
                    break;
                default:
                    break;
            }
            window_size = min(cwnd, advertised_window_size);
        }
```

处于慢启动阶段时，窗口大小增大1MSS。每过一个RTT，cwnd翻倍，窗口大小呈指数增长；而处于拥塞避阶段时，免`cwnd = cwnd + MSS*(MSS/cwnd)`。相当于每过一个RTT，cwnd加1。

当收到冗余ACK时，我们需要进行快速重传，并进行窗口大小更新。

```
            //duplicate ACK
            dupACKcount++;
            if (RENO_STATE == SLOW_START || RENO_STATE ==
CONGESTION_AVOIDANCE) {
                if (dupACKcount == 3) {
                    //fast retransmit
                    ssthresh = cwnd / 2;
                    cwnd = ssthresh + 3 * MSS;
                    window_size = min(cwnd,
advertised_window_size);
                    RENO_STATE = FAST_RECOVERY;
                    print_message("Fast
resend"+to_string(ack_num), WARNING);
                    //resend the packet
                    udt_send(sndpkts[ack_num + 1]);
                } else {
                    cwnd += MSS;
                }
```

```
                }
```

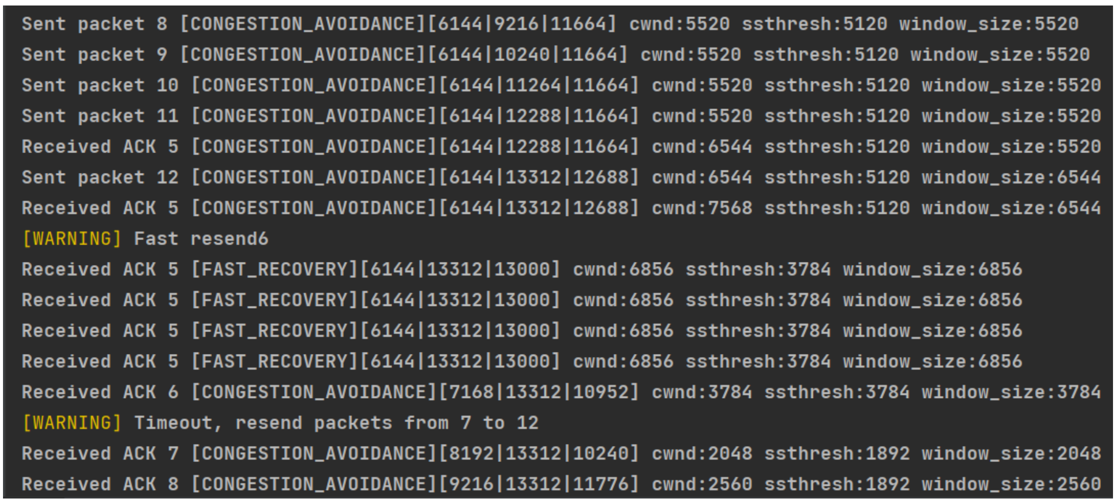
超时处理

注意到，不管什么状态下，超时后都需要恢复到慢启动状态，因此直接在超时事件上进行改动即可。

```
//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(sndpkts[i]);
    }
    ssthresh = cwnd / 2;
    cwnd = MSS;
    dupACKcount = 0;
    RENO_STATE = SLOW_START;
    timer.start_timer();
}
```

接收端

首先我们可以运行查看接收端不进行改变的情况：



```
Sent packet 8 [CONGESTION_AVOIDANCE][6144|9216|11664] cwnd:5520 ssthresh:5120 window_size:5520
Sent packet 9 [CONGESTION_AVOIDANCE][6144|10240|11664] cwnd:5520 ssthresh:5120 window_size:5520
Sent packet 10 [CONGESTION_AVOIDANCE][6144|11264|11664] cwnd:5520 ssthresh:5120 window_size:5520
Sent packet 11 [CONGESTION_AVOIDANCE][6144|12288|11664] cwnd:5520 ssthresh:5120 window_size:5520
Received ACK 5 [CONGESTION_AVOIDANCE][6144|12288|11664] cwnd:6544 ssthresh:5120 window_size:5520
Sent packet 12 [CONGESTION_AVOIDANCE][6144|13312|12688] cwnd:6544 ssthresh:5120 window_size:6544
Received ACK 5 [CONGESTION_AVOIDANCE][6144|13312|12688] cwnd:7568 ssthresh:5120 window_size:6544
[WARNING] Fast resend6
Received ACK 5 [FAST_RECOVERY][6144|13312|13000] cwnd:6856 ssthresh:3784 window_size:6856
Received ACK 5 [FAST_RECOVERY][6144|13312|13000] cwnd:6856 ssthresh:3784 window_size:6856
Received ACK 5 [FAST_RECOVERY][6144|13312|13000] cwnd:6856 ssthresh:3784 window_size:6856
Received ACK 5 [FAST_RECOVERY][6144|13312|13000] cwnd:6856 ssthresh:3784 window_size:6856
Received ACK 6 [CONGESTION_AVOIDANCE][7168|13312|10952] cwnd:3784 ssthresh:3784 window_size:3784
[WARNING] Timeout, resend packets from 7 to 12
Received ACK 7 [CONGESTION_AVOIDANCE][8192|13312|10240] cwnd:2048 ssthresh:1892 window_size:2048
Received ACK 8 [CONGESTION_AVOIDANCE][9216|13312|11776] cwnd:2560 ssthresh:1892 window_size:2560
```

可以看到，当接收方不缓存失序的包时，即使有了快速重传，由于之前发过的，对接收端来说失序的包没有进行缓存，仍旧相当于丢失了。因此快速重传当前期望的包只是解决了“眼前的问题”，其余的包迟早还要超时重传，因此尽管进行了拥塞控制，但重传的行为在实际网络环境中事实上加剧了拥塞。

因此基于此可以对接收端的逻辑进行改进：

在前面提到，接收端必须是累计确认的（确认按序收到的最大序号），发送端才有理由在移动窗口时移动到当前`ack+1`的位置。同时我们还想让接收端缓存失序的包，以避免发送端进行已收到的包的重传而加剧拥塞。

因此我在上一次实验的`SR`的基础上修改发送ACK行为，使其从选择确认转变成累计确认，同时保留其缓存失序包行为即可解决这个问题。代码如下所示：

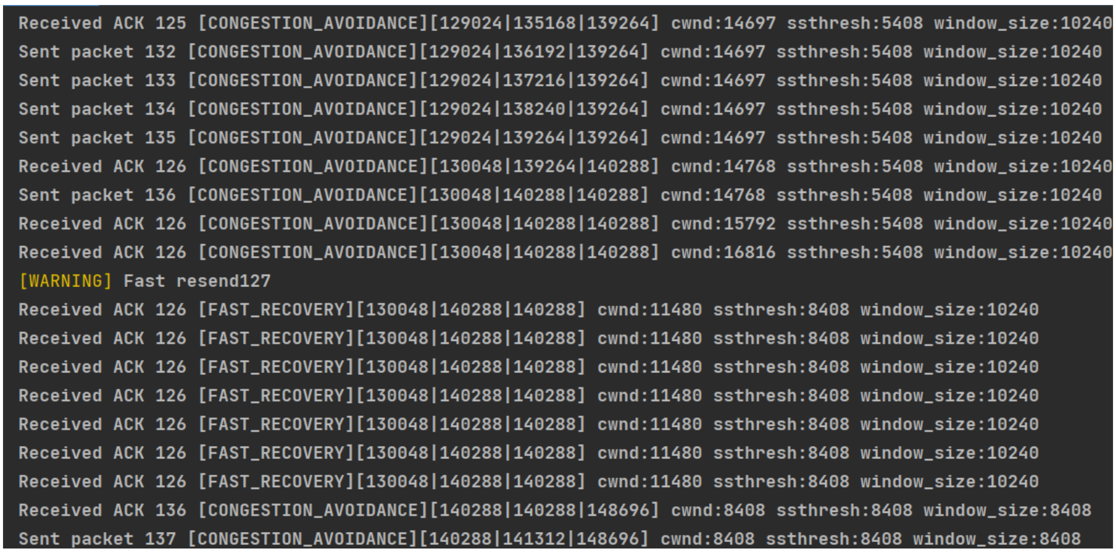
```
            if (pkt_seq >= rcv_base && pkt_seq <= rcv_base + N - 1) {
                //in the window
                if (!acked[pkt_seq]) {
                    if (pkt_seq == rcv_base) {
                        //the first packet in the window
                        pkt_data_size = rcvpkt.head.data_size;
                        memcpy(file_buffer + pkt_seq * MAX_SIZE, rcvpkt.data, pkt_data_size);
                        acked[pkt_seq] = true;
                        print_message("Received packet " + to_string(pkt_seq), DEBUG);
                        //slide the window
                        while (acked[rcv_base]) {
                            rcv_base++;
                        }
                        packet sndpkt = make_pkt(ACK, rcv_base - 1);

                        udt_send(sndpkt);
                    } 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, rcv_base
- 1);
                            udt_send(sndpkt);
                            print_message("Received packet " +
to_string(pkt_seq) + ", cached", DEBUG);
                        }
                    } else {
                        //already acked in the window, do not
resend ack
                        print_message("Received packet " +
to_string(pkt_seq) + " again", WARNING);
                          packet sndpkt = make_pkt(ACK, rcv_base -
1);
                          udt_send(sndpkt);
                    }
                }
```

当收到窗口内的包时，总是发送当前`rcv_base - 1`位置的ACK。但是如果收到的包恰巧在`rcv_base`上，那么窗口其实有潜力往前移动很多，以覆盖之前缓存过的包。移动完了之后再发送`rcv_base - 1`位置的ACK，让发送端知道在这之前的包都已经按序收到了。除此之外的情况都不发送ACK。

可以看到，修改后快速恢复起到了其应有的作用。



```
Received ACK 125 [CONGESTION_AVOIDANCE][129024|135168|139264] cwnd:14697 ssthresh:5408 window_size:10240
Sent packet 132 [CONGESTION_AVOIDANCE][129024|136192|139264] cwnd:14697 ssthresh:5408 window_size:10240
Sent packet 133 [CONGESTION_AVOIDANCE][129024|137216|139264] cwnd:14697 ssthresh:5408 window_size:10240
Sent packet 134 [CONGESTION_AVOIDANCE][129024|138240|139264] cwnd:14697 ssthresh:5408 window_size:10240
Sent packet 135 [CONGESTION_AVOIDANCE][129024|139264|139264] cwnd:14697 ssthresh:5408 window_size:10240
Received ACK 126 [CONGESTION_AVOIDANCE][130048|139264|140288] cwnd:14768 ssthresh:5408 window_size:10240
Sent packet 136 [CONGESTION_AVOIDANCE][130048|140288|140288] cwnd:14768 ssthresh:5408 window_size:10240
Received ACK 126 [CONGESTION_AVOIDANCE][130048|140288|140288] cwnd:15792 ssthresh:5408 window_size:10240
Received ACK 126 [CONGESTION_AVOIDANCE][130048|140288|140288] cwnd:16816 ssthresh:5408 window_size:10240
[WARNING] Fast resend127
Received ACK 126 [FAST_RECOVERY][130048|140288|140288] cwnd:11480 ssthresh:8408 window_size:10240
Received ACK 126 [FAST_RECOVERY][130048|140288|140288] cwnd:11480 ssthresh:8408 window_size:10240
Received ACK 126 [FAST_RECOVERY][130048|140288|140288] cwnd:11480 ssthresh:8408 window_size:10240
Received ACK 126 [FAST_RECOVERY][130048|140288|140288] cwnd:11480 ssthresh:8408 window_size:10240
Received ACK 126 [FAST_RECOVERY][130048|140288|140288] cwnd:11480 ssthresh:8408 window_size:10240
Received ACK 126 [FAST_RECOVERY][130048|140288|140288] cwnd:11480 ssthresh:8408 window_size:10240
Received ACK 126 [FAST_RECOVERY][130048|140288|140288] cwnd:11480 ssthresh:8408 window_size:10240
Received ACK 136 [CONGESTION_AVOIDANCE][140288|140288|148696] cwnd:8408 ssthresh:8408 window_size:8408
Sent packet 137 [CONGESTION_AVOIDANCE][140288|141312|148696] cwnd:8408 ssthresh:8408 window_size:8408
```

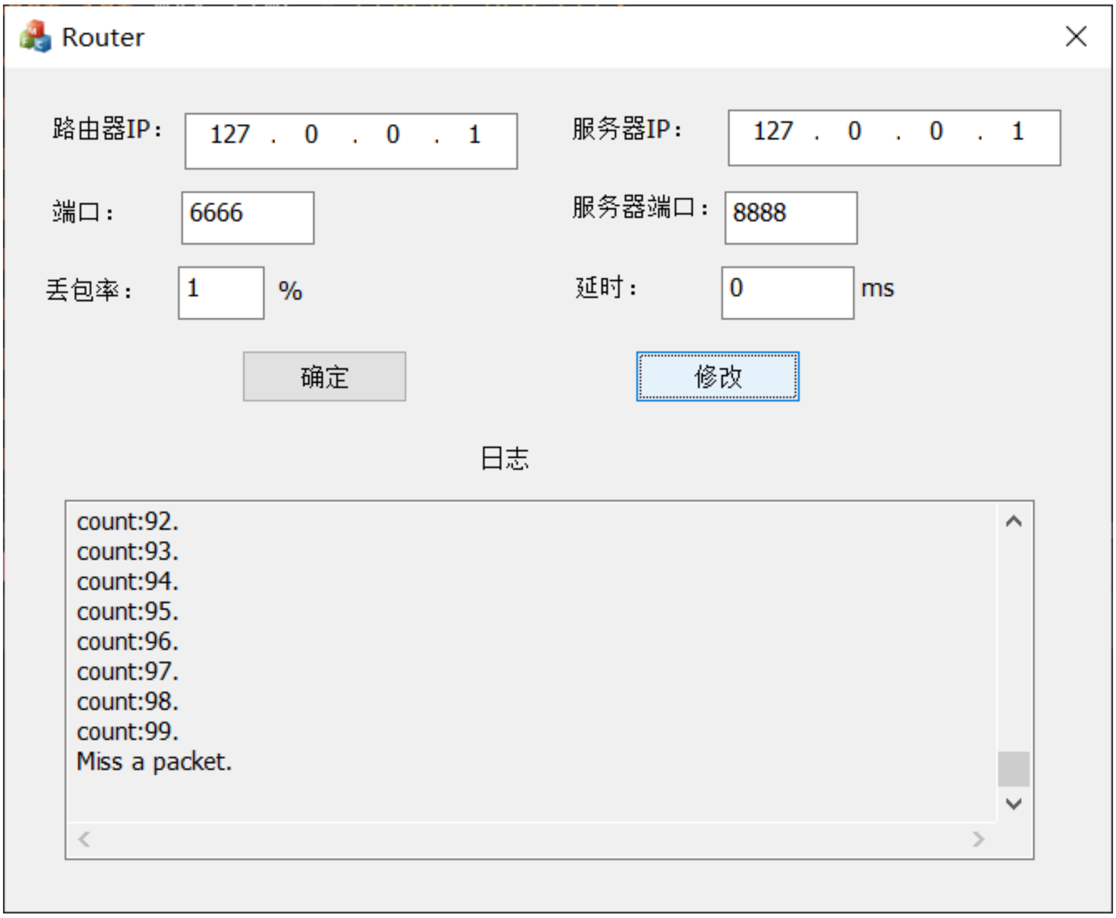
成功收取！

```
[DEBUG] Received packet 1506
[DEBUG] Received packet 1507
[DEBUG] Received packet 1508
[DEBUG] Received packet 1509
[DEBUG] Received packet 1510
[DEBUG] Received packet 1511
[DEBUG] Received packet 1512
[DEBUG] Received packet 1514, cached
[DEBUG] Received packet 1515, cached
[DEBUG] Received packet 1513
[SUCCESS] Received file successfully
[INFO] Time used: 129098ms
[SUCCESS] File saved to C:\Users\LENOVO\Desktop\rdt_v1\receiver_files\helloworld.txt
```

程序演示

建立连接

路由器设置:



增大通告窗口大小，观察慢启动和拥塞控制阶段窗口大小变化。慢启动阶段，`cwnd`的值迅速增大，当`cwnd ⩾ ssthresh`后进入拥塞控制阶段，cwnd增速减缓。

```
Sent packet 40 [SLOW_START][21504|41984|44032] cwnd:22528 ssthresh:25600 window_size:22528
Sent packet 41 [SLOW_START][21504|43008|44032] cwnd:22528 ssthresh:25600 window_size:22528
Sent packet 42 [SLOW_START][21504|44032|44032] cwnd:22528 ssthresh:25600 window_size:22528
Received ACK 21 [SLOW_START][22528|44032|46080] cwnd:23552 ssthresh:25600 window_size:23552
Sent packet 43 [SLOW_START][23552|45056|48128] cwnd:24576 ssthresh:25600 window_size:24576
Received ACK 22 [SLOW_START][23552|45056|48128] cwnd:24576 ssthresh:25600 window_size:24576
Received ACK 23 [CONGESTION_AVOIDANCE][24576|45056|50176] cwnd:25600 ssthresh:25600 window_size:25600
Received ACK 24 [CONGESTION_AVOIDANCE][25600|45056|51240] cwnd:25640 ssthresh:25600 window_size:25640
Sent packet 44 [CONGESTION_AVOIDANCE][25600|46080|51240] cwnd:25640 ssthresh:25600 window_size:25640
Sent packet 45 [CONGESTION_AVOIDANCE][25600|47104|51240] cwnd:25640 ssthresh:25600 window_size:25640
Sent packet 46 [CONGESTION_AVOIDANCE][25600|48128|51240] cwnd:25640 ssthresh:25600 window_size:25640
Sent packet 47 [CONGESTION_AVOIDANCE][25600|49152|51240] cwnd:25640 ssthresh:25600 window_size:25640
Sent packet 48 [CONGESTION_AVOIDANCE][25600|50176|51240] cwnd:25640 ssthresh:25600 window_size:25640
Sent packet 49 [CONGESTION_AVOIDANCE][25600|51200|51240] cwnd:25640 ssthresh:25600 window_size:25640
Sent packet 50 [CONGESTION_AVOIDANCE][25600|52224|51240] cwnd:25640 ssthresh:25600 window_size:25640
Received ACK 25 [CONGESTION_AVOIDANCE][26624|52224|52304] cwnd:25680 ssthresh:25600 window_size:25680
Received ACK 26 [CONGESTION_AVOIDANCE][27648|52224|53368] cwnd:25720 ssthresh:25600 window_size:25720
Received ACK 27 [CONGESTION_AVOIDANCE][28672|52224|54432] cwnd:25760 ssthresh:25600 window_size:25760
Received ACK 28 [CONGESTION_AVOIDANCE][29696|52224|55496] cwnd:25800 ssthresh:25600 window_size:25800
```

快速重传的正确性上面分析的过程中也已经验证。