

南京大学本科生实验报告

课程名称：计算机网络

任课教师：田臣/李文中

助教：

学院	计算机科学与技术	专业（方向）	计算机科学与技术
学号	201220062	姓名	黄子睿
Email	201220062@smail.nju.edu.cn	开始/完成日期	5/6-5/8

1. 实验名称：

Lab5 Reliable Communication

2. 实验目的：

通过 blaster, blastee 与 middlebox 的设计，实现简单的可靠传输（Reliable communication），主要关注其中的滑动窗口机制与重传机制。

3. 实验内容：

Task2 Middlebox

MiddleBox 模拟了简单网络中的路由器，主要功能为转发 blaster 与 blastee 相互之间发送的报文，并以一定概率丢失 blaster 发给 blastee 的报文。

下面给出 MiddleBox 的伪代码：

1. `def` Middlebox:
2. 以一定概率丢失 `blaster` 发送给 `blastee` 的报文
3. 修改报文的 `ethernet` 头，并将 `IPv4` 报头中的 `ttl` 字段减一
4. 将修改后的报文发送给目的地址

Task3 Blastee

在收到 Blaster 发送的报文后，blastee 需要发送 ACK 应答报文。

首先填充 ACK 报文的 ethernet 报头，IPv4 报头与 UDP 报头，前两者由于网

络结构固定，因此直接硬编码就可以了；后者在本次实验中没有直接的用途，只是需要占位防止报错。

同时，ACK 报文需要构造序号 seqnum 与 payload 项，此二者利用 blaster 发送的报文实现。首先从 blaster_pkt 中取出代表序列号的二进制位串 seqbyte 与代表可变装载字段长度的字段 lengthbyte，并通过 struct.unpack 将 lengthbyte 转化为整型数 length。如果 length 小于 8，则将 ACK 中的 payload 部分全赋为 0，否则取 blaster_pkt.payload 得前 8 比特填入 ACK.payload 即可。

下面给出伪代码：

```
1. def blaste_handle_packet:
2.     get blaster_pkt from middlebox
3.     ackpkt := Ethernet() + IPv4() + UDP()
4.     implement ackpkt[0], ackpkt[1] and ackpkt[2] by hard code
5.     seqbyte := blaster_pkt.seqbyte
6.     lengthbyte := blaster_pkt.lengthbyte
7.     length = lengthbyte to int
8.     payload := None
9.     if length < 8:
10.        fill payload with 0
11.     else:
12.        fill payload with the first 8 bits in blaster_pkt.payload
13.     rpc := seqbyte + payload
14.     ackpkt append with rpc
15.     send packet ackpkt to middlebox
```

Task4 Blaster

对于 Blaster 而言，首先考虑以下三个功能的实现：

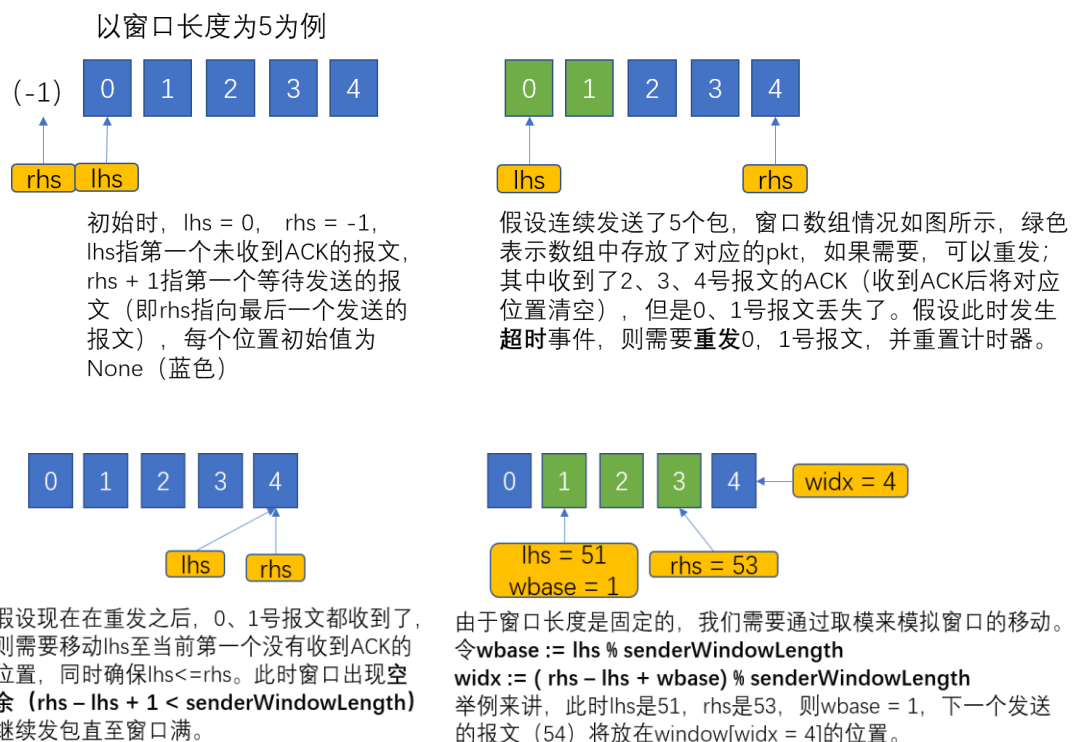
1. 实现滑动窗口
2. 实现超时重传
3. 为了结束时的状态打印，记录运行时的相关数据

首先考虑滑动窗口的实现：

滑动窗口的实现需要增加三个数据成员，分别是滑动窗口的左端，右端以及代表滑动窗口的数组本身。

```
1. # for snederWindow
2. self.lhs = 0
3. self.rhs = -1
4. self.window = [None]*self.senderWindowLenth
```

可以看到自始至终，我们只需要 `self.senderWindowLenth` 长度的滑动窗口，在具体的运行过程中，只需要不断取模就可以实现窗口滑动的效果。具体来讲，固定 `lhs` 后，`rhs` 不大于 `rhs+4`，随着不断发送新的报文与接受到 ACK 报文，`rhs` 与 `lhs` 将以取模加的方式移动，下图为一个例子：



利用 `wbase` 与 `widx` 的取模操作，`lhs` 与 `rhs` 在实际运行时只需要不断增加就可以了，同时利用 `senderWindowLength` 长度的窗口数组即可实现需要的功能。不过其中还有一些细节需要阐述，即每一次程序将丢弃序号小于 `lhs` 或大于 `rhs` 的 ACK；并且假如返回的 `window[ACK.seqNum]`为 `None`，直接忽略该冗余应答，否则将对应位置 `window[ACK.seqNum]`清空为 `None`，表示已经收到应答；同时，

一旦 `window[lhs % sendeWindowLength] = None`, 需要向前移动 lhs, 没法送一个新的包, 就像前移动 rhs。

其次考虑**超时重传**的问题, 为实现这一功能, 首先在数据成员中增加 `self.reTXlist` 数组, 用来记录等待重传的所有报文。在没有收到 ACK 的时间段里, 程序需要不断检测计时器, 一旦发生超时(`time.time() - timestamp >= maxTime`), 就立即重传此时窗口中等待 ACK 的所有报文 (根据上述滑动窗口的设计, 所有需要重传的报文应当是 `window` 数组中非 `None` 的部分) 加入 `reTXlist` 数组中。在每一次 lhs 向前移动或者发生超时之后, 就更新时间戳。为了实现这一功能, 需要增加数据成员 `self.roundTime = time.time()`。注意到重传的优先级高于发送新报文, 因此只有当 `reTXlist` 数组为空时, 才考虑发送新报文。

最后考虑程序结束时**状态打印**的问题。首先需要增加相关的数据成员, 如下所示:

```
1. # for printing stats
2. self.initTime = time.time()
3. self.reTXnum = 0
4. self.timeoutnum = 0
5. self.throughput = 0
6. self.goodput = 0
```

每一次发送一个新的包就需要在 `goodput` 上加 `pkt.size()`(注意到每一个包都是等长的); 而每发送一个包 (无论是否为重发) 就在 `throughput` 上加 `pkt.size()`; 每发送一个重传的包, 就在 `reTXnum` 上加 1; 而每次超时就在 `timeoutnum` 上加 1。最后将各个信息输出即可 (注意到总时间就是输出时刻的时间减去 `initTime`, 而 `throughput` 与 `goodput` 需要在输出时除以总时间)。这些功能被封装在了函数 `print_stats()` 中。

下面来看上述功能的函数封装实现, 除此之外还需要考虑两个最重要的功能,

收发报文。

首先考虑接收报文，将这一功能封装为函数 `self.recv_packet(self, packet)`。它的伪代码是：

```
1. def recv_packet(self, packet):
2.     if packet.seqnum is not in [ self.lhs, self.rhs ]:
3.         drop the packet
4.     else:
5.         wbase = self.lhs % self.senderWindowLenth
6.         widx = (seqnum - self.lhs + wbase) % self.senderWindowLenth
7.         self.window[widx] = None
8.         move self.lhs to the last position that is not None and make
           sure lhs < rhs
9.         reset timestamp if lhs moved
```

然后考虑接收报文，其中的细节在前面已经阐明，其伪代码为：

```
1. def send_packet(self):
2.     if self.reTXlist is not empty:
3.         retransmit packets in reTXlist
4.     else:
5.         if the window is not full:
6.             transmit a new packet
```

注意 lhs 的移动不会打断正在进行的重发，但是重发过程中的超时会重启重发过程。

可以看到发送报文被分为重发报文与发送新报文两个函数。并且前者的优先级更高。下面分别给出两者的伪代码：

```
1. def send_reTX_packet(self): # retransmit
2.     if self.reTXlist is not empty:
3.         transmit the packet in reTXlist[0]
4.         del reTXlist[0]
5.         increment of reTXnum and throughput
6.
7. def send_new_packet(self):
8.     generate a new packet with headers and seqnum, length and payload
       and send it
9.     self.totnum += 1
10.    increment of throughput and goodput
```

```
11.     put the new packet into window[widx]
        (widx is calculated by lhs and rhs), waiting for ACK
```

同时我们将超时处理封装为 `handle_packet` 函数，其伪代码如下：

```
1. def handle_packet(self):
2.     if time.time() - timestamp >= timeout:
3.         clear self.reTXlist
4.         put every packet in window that has not received ACK into
           self.reTXlist
5.         increment of timeoutnum
6.         Reset timestamp
```

通过上述函数的封装，`handle_packet` 与 `handle_no_packet` 函数就可以很轻松地实现。首先考虑 `handle_no_packet`，此时由于没有发来地 ACK，只需要考虑发包即可，其为代码为：

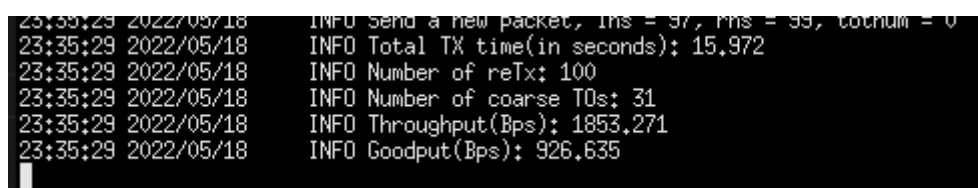
```
1. def handle_no_packet:
2.     if totnum > 0:
3.         send_packet()
4.     elif totnum == 0: # all packets are sent
5.         print_stats()
6.     else:
7.         return
```

在 `handle_no_packet` 的基础上，`handle_packet` 的发包逻辑与之一致，只需要增加收包逻辑即可，其伪代码为：

```
1. def handle_packet(self, packet):
2.     recv_packet(packet)
3.     handle_no_packet()
```

下面给出运行的截图：

首先是 xterm 下窗台输出的例子：



```
23:35:29 2022/05/18 INFO Send a new packet, lhs = 97, rhs = 99, totnum = 0
23:35:29 2022/05/18 INFO Total TX time(in seconds): 15.972
23:35:29 2022/05/18 INFO Number of reTx: 100
23:35:29 2022/05/18 INFO Number of coarse T0s: 31
23:35:29 2022/05/18 INFO Throughput(Bps): 1853,271
23:35:29 2022/05/18 INFO Goodput(Bps): 926,635
```

```

23:38:34 2022/05/18 INFO send a new packet, lhs = 31, rhs = 33, cooham = 0
23:38:34 2022/05/18 INFO Total TX time(in seconds): 11.273
23:38:34 2022/05/18 INFO Number of reTx: 53
23:38:34 2022/05/18 INFO Number of coarse T0s: 18
23:38:34 2022/05/18 INFO Throughput(Bps): 2008.621
23:38:34 2022/05/18 INFO Goodput(Bps): 1312.824

```

上图包含了两次状态输出（即两次运行的结果），此时输入对的参数即为手册中给出的参数，丢包率是 0.19。可以看到第一次运行中，发生了 31 次超时（即丢失了 31 个包），而第二次运行丢失了 18 个包（超时 18 次），可以明显看到后者的效率（主要是 goodput 一项）比前者高不少。同时重传的报文数基本是丢包数的两到三倍，这是因为当窗口中的第一个报文超时，后面的报文的 ACK 可能还没有及时发到，这会倒是冗余 ACK。

下面不修改 blaster 与 blastee 的参数，仅仅改变 middlebox 的丢包率，调整至 0.30，运行结果如下：

```

23:41:16 2022/05/18 INFO send a new packet, lhs = 99, rhs = 99, cooham = 0
23:41:16 2022/05/18 INFO Total TX time(in seconds): 16.608
23:41:16 2022/05/18 INFO Number of reTx: 113
23:41:16 2022/05/18 INFO Number of coarse T0s: 35
23:41:16 2022/05/18 INFO Throughput(Bps): 1898.092
23:41:16 2022/05/18 INFO Goodput(Bps): 891.123

```

比较一下两次不同的参数（丢包率 0.19 与 0.30），可以明显看到，throughput 的效率没有多大改变（这是由 senderWindowLength, recvTimeout 与 timeout 决定的），而 goodput 明显变小，发生超时的次数也明显变多了。

下面考虑改变 senderWindowLength, recvTimeout 与 timeout 的值，使 senderWindowLength = 7，其余参数与手册上举的例子一致，则运行结果如下：

```

23:43:37 2022/05/18 INFO send a new packet, lhs = 31, rhs = 33, cooham = 0
23:43:37 2022/05/18 INFO Total TX time(in seconds): 10.457
23:43:37 2022/05/18 INFO Number of reTx: 65
23:43:37 2022/05/18 INFO Number of coarse T0s: 19
23:43:37 2022/05/18 INFO Throughput(Bps): 2335.172
23:43:37 2022/05/18 INFO Goodput(Bps): 1415.256
^C23:43:41 2022/05/18 INFO Restoring saved iptables state

```

与第一张图片里的运行结果相比，可以看到通过改变 senderWindowLength，运行时的 throughput 值明显增加，从 2008 增加到 2335，两者之间是正相关的

关系，即窗口越长，总的发包效率就越高，当然 goodput 的大小除了与此相关，还与丢包率相关，由于丢包率具体到每一次运行上的丢包数不同，少量测试未必符这一差异，但平均情况下该正相关还是成立的。如果尝试将窗口长度改为 20:

```
23:46:13 2022/05/18 INFO Send a new packet, lhs = 96, rhs = 99, totnum = 0
23:46:13 2022/05/18 INFO Total TX time(in seconds): 8.062
23:46:13 2022/05/18 INFO Number of reTx: 86
23:46:13 2022/05/18 INFO Number of coarse T0s: 14
23:46:13 2022/05/18 INFO Throughput(Bps): 3414.531
23:46:13 2022/05/18 INFO Goodput(Bps): 1835.769
```

可以看到 throughput 有明显提升了。

recvTimeout 代表函数处理的频率，下面将此数值改为 500，其余参数与手册中相同，下图为运行结果：

```
3:49:22 2022/05/18 INFO Send a new packet, lhs = 98, rhs = 99, totnum = 0
3:49:22 2022/05/18 INFO Total TX time(in seconds): 49.334
3:49:22 2022/05/18 INFO Number of reTx: 27
3:49:22 2022/05/18 INFO Number of coarse T0s: 29
3:49:22 2022/05/18 INFO Throughput(Bps): 380.991
3:49:22 2022/05/18 INFO Goodput(Bps): 299.993
```

明显可以看到运行速率变慢了，说明运行速率与 recvTimeout 成反相关，理论上也确实如此，处理的频率越低，总时间一定越长。值得注意的是，此时虽然超时次数大致与前面相同，但是重发包数量与超时次数的比值更加接近 1，而先前一般为 4 到 5，这是由于超时发生时需要发送的包，在函数处理速率减小后，数量减少了。（在设计中，我令重发不应 lhs 移动而停止）。

下面将 timeout 的值从原来的 300 改为 150 或 500，其余参数与手册用例一致，分别给出两种情况的运行结果：

(timeout = 150)

```
09:34:25 2022/05/19 INFO Send a new packet, lhs = 98, rhs = 99, totnum = 0
09:34:25 2022/05/19 INFO Total TX time(in seconds): 11.346
09:34:25 2022/05/19 INFO Number of reTx: 104
09:34:25 2022/05/19 INFO Number of coarse T0s: 38
09:34:25 2022/05/19 INFO Throughput(Bps): 2660.977
09:34:25 2022/05/19 INFO Goodput(Bps): 1304.401
```

通过将上图与第一张运行结果比较，不难发现，throughput 的值有明显的

提高。同时由于 timeout 的减小，虽然超时的次数增加了，但是用时却更少，这也导致 goodput 与 throughput 的比值变小，增加了不必要的重传与冗余应答。

(timeout = 500)

```
09:40:38 2022/05/19 INFO Total TX time(in seconds): 16.421
09:40:38 2022/05/19 INFO Number of reTx: 23
09:40:38 2022/05/19 INFO Number of coarse T0s: 13
09:40:38 2022/05/19 INFO Throughput(Bps): 1108.554
09:40:38 2022/05/19 INFO Goodput(Bps): 901.263
```

比较可得，timeout 调整为 500 之后明显传输效率降低了，因为丢包之后，网络将等待过长的时间才会判定需要超时重传。但是 goodput 与 through 的比值却提高了，此时网络的利用率反而更高。

考虑一个最极端的情况，将丢包率调整为 0，给出这种理想情况下的运行情况（其余参数与手册用例一致）：

```
09:45:08 2022/05/19 INFO Send a new packet, tns = 33, rns = 33, cooNam = 0
09:45:08 2022/05/19 INFO Total TX time(in seconds): 9.505
09:45:08 2022/05/19 INFO Number of reTx: 41
09:45:08 2022/05/19 INFO Number of coarse T0s: 9
09:45:08 2022/05/19 INFO Throughput(Bps): 2195.469
09:45:08 2022/05/19 INFO Goodput(Bps): 1557.07
```

可以看到尽管 middlebox 的丢包率是 0，仍然出现了重传的现象，这是因为 blastee 的处理速度太慢（recv = self.net.recv_packet(timeout=1.0)）而导致的超时现象，将 blaster 的 timeout 设置为 500，再运行可得：

```
09:46:01 2022/05/19 INFO Send a new packet, tns = 33, rns = 33, cooNam = 0
09:46:01 2022/05/19 INFO Total TX time(in seconds): 8.106
09:46:01 2022/05/19 INFO Number of reTx: 0
09:46:01 2022/05/19 INFO Number of coarse T0s: 0
09:46:01 2022/05/19 INFO Throughput(Bps): 1825.759
09:46:01 2022/05/19 INFO Goodput(Bps): 1825.759
```

这样可以看出不再有超时与重传。

下面是 wireshark 对 blastee 的抓包截图：

