Project2-Bears-TP

——可靠的传输

作者（Author）

姓名：AnDJ
学号：
班级：

解决过程

1. 测试给出的 Sender

   该 project 中给出 BasicSender，是所有给出的 Sender 的基类，也就是 Sender 的一个标准实现。从这个基类派生了三个 Sender，就是 InteractiveSender（交互式 Sender）、StanfurdSender（有错误的 Sender）、UnreliableSender（不可靠的 Sender）。从代码结构可以看出，这三个都一定程度地实现了 Sender。

   该 project 中给出了测试使用的 receiver，该 receiver 实现了基础的接收端的功能，包括接收文件、发送 ack，其中会按照协议中数据包的标准解析数据包，按照协议中 ack 的标准发送 ack 包。从源代码中可以获得的信息有：接收到的文件名称为：host.port。可以根据这个名字寻找接收到的文件。

   给出测试：先运行接收端 Receiver。UnreliableSender 是不可靠的，先运行 UnreliableSender。传输完成后，在文件目录中产生了文件 host.port，将该文件和 README 文件进行 md5 校验，是一致的，因此这个 Unreliable 的 Sender 应该是说，在没有丢包的产生的时候，该 sender 是可行的，正因为这个情况因此是不可靠的。
2. 产生了问题

   问题：如果整个过程中产生了丢包等情况，怎样实现可靠传递。即实现一个“靠谱”的 Sender。
3. 探究 UnreliableSender 是怎样发包的

   解决过程：阅读源代码

   UnreliableSender 的主要发包策略是，单包传输。

   BaseSender 限制了必须实现 sender 的 start 函数，从 UnreliableSender 的 start 函数可以看出，该函数是启动 sender 机制的主函数。UnreliableSender 的策略是，每 500 个字节读取一次，将该包发送后直接发送到接收端，然后等待，等待 receiver 发送 ack 过来，然后接着再次发送。这个过程还是蛮清楚的。
4. Unreliable 在哪些地方不可靠

   包损坏。首先这个问题是直接可以被发现的，因为在发送的时候在包的结尾有该包的校验和，一旦这个包损坏，那么在接收端是会检查出来并且丢弃。这样的话结局就是无接收该包，等同于包丢失。

包丢失。这个问题就是需要重点解决的问题。接下来阐述。

5. 包丢失

首先是必须要研究发包的连接是怎样建立的。在发包之前，遵照协议是需要先发送一个 start 包，从道理上来讲，这个包的作用是，在发送至接收端创建对该 IP 的连接。

为了验证这个过程，需要读一下 receiver 的代码。大致的过程是，receiver 始终有一个循环，这个循环一直在等待接受包，如果接受到 start 包，那么判断对应这个 IP 是否建立了对应的 connection，没有建立的话就会建立该连接。Receiver 在内部缓存了每一个建立的 connection，也就是说道理上来讲，receiver 是可以同时接受很多的 IP 往该 connection 进行传输的。这个也不是重点。如果对应的 IP 连接存在，之后的包会存至对应的这个 connection。当拿到 end 包的时候，connection 关闭。

协议中给出了返回 ack 的情况。假设现在接收到了【0,1,2】，3 号包丢失了，现在 4 号包发送过来，receiver 照样会接受，现在接受到的包为【0,1,2,4】，但是会返回一个“expect 3 packet”这样的 ack。当 sender 端再次发送过来 3 号包的时候，现在接收的包为【0,1,2,3,4】，现在会发送“expect 5 packet”这样的包。这个意思就是，每次都是返回已接受的包列表中从 0 开始的递增序列的最大的包+1，即期待的包。

6. 单包测试

我首先选择实现单包传送，即 stop and go，然后再实现窗口，让多包提升速率。我的单包的实现思路是这样的，sender 设置 timeout，如果在 timeout 之内，sender 发送的包能拿到对应的 ack，那么继续发送下一个包。如果出现超时，那么这个包就认为是丢了（也可能出现环的情况，也当做丢包处理），则重发这个包，直到拿到该包已接收的 ack，再发送下一个包。这个代码实现比较简单。

7. 实现窗口

文档中要求需要实现窗口大小为 5，窗口中的 5 个包都是在“路上”的包，这五个包是需要进行缓存的，因为窗口的策略就是为了在某个包丢了可以直接在该缓存池中拿到并且发送过去。Sender 中的 self.window 的 list 用来缓存窗口中的 5 个包。

第一步，需要先发送 5 个包。这里存在一个情况，就是可能这个文件比较小，没有构成 5 个包就读完了，这里用了 self.isreadall 来标注是否将这个文件读完。具体的步骤是，从文件中读 5 个包，将该包缓存至 self.window 里并发送，如果读完就标记 self.readall。最后的 end 包这里采用了一种笨拙的方式判断，和 UnreliableSender 不同，读到文件末尾会多读一个“”，因此我这里实现的时候，最后的一个包始终是“end||checksum”，即 data 区是空的，这样是会多发一个包，不过代码减少了一些复杂。现在，self.window 是这样的：

0	1	2	3	4
packet0	packet1	packet2	packet3	packet4

第二步，处理各种可能出现的丢包情况。

1） Packet 0 丢失。这个是需要单独拿出来说的，因为，如果 packet 0 这个包丢失的话，那么意味着，receiver 端关于这个 IP 的连接就没有建立，这样的话，等同于 5 个包全部丢失了。这种情况只有一种解决办法，那就是重新再发一次 packet 0。

2） 除 packet 0 之外的其他包丢失。这里列举一种情况，比如，在传输过程中，

1,4 两个包丢失了，receiver 端接收情况是【0,2,3】。这种情况下，sender 会接收到三个 ack（当然我们这里先考虑 ack 不会丢失），这三个 ack 的内容都是：“expect packet 1”。这样的话，我想到有两种解决这个问题的办法，即：（1）既然 0 号包已经接收到了，那么可以首先将窗口向右滑动一个，将新加入 window 的包发出去（首先得保证加入到 window 的都在“过程中”）。接着，因为 1 号包在 window 中，说明 1 号包在发的过程中，但是现在还发回了请求 1 号包，那么我不管到底 1 号包到底到没到，索性就再把 1 号包发一遍，这样的话，我们在这种情况下发了两个包。（2）和第一种情况不同，这种情况是，现在的 window 中的所有包都在过程中，那么我就一直等，对 ack 进行决策，在既定时间如果没有拿到 expect 5，那么就把 receiver 缺少的包再发过去，直到该 window 中的包全部发过去，然后让 window 向后滑 5 个单位，继续执行该操作。

3）这里我选择了第一种策略，即每次拿到 ack 后，首先执行窗口滑动的操作，将新加入 window 的包发送出去，然后再发出可能会需要的重复包。这样的话，在出现丢包的情况下，会同时发送两个包，一个是新加入的包，一个是丢的包，比较灵活。但是，每次滑动窗口的步长需要注意。

4）对第一种情况，我这里分别套用几个情景具体解释。

（1） Receiver: [0 1 2 3 5 6]

Window:

No	0	1	2	3	4
packet	4	5	6	7	8

Ack: expect packet 4

首先，先判断滑动窗口的步长。因为 ack 反馈需要再发 4 号包，那么 4 号包必须在 window 内，因此不滑动窗口。

其次，不滑动窗口，那么现在的决策是，只发一次 4 号包。

最终决策：窗口不动，只发 4 号包。

（2） （1）情况接到 4 号包

Receiver: [0 1 2 3 4 5 6]

Window:

No	0	1	2	3	4
packet	4	5	6	7	8

Ack: expect packet 7

首先判断滑动窗口的步长。Ack 反馈需要 7 号包，那么 7 号包必须在 window 中，7 号之前的包已经被接收到了，因此可以从 window 中删去。因此可得出 window 向右滑动的步长为 3 步，那么添加 3 个包进入 window 中，现在 window 为：

No	0	1	2	3	4
packet	7	8	9	10	11

其次，发送包。因为添加了三个包： 9, 10，11，因此这三个包必须发送出去。并且，ack 要求发送 7 号包，因此，也需要将 7 号包发送出去。

最终的结果为：发送 7，9, 10，11 号四个包。

（3） （1）情况接到 4 号包，但是文件已经读完了。

Receiver: [0 1 2 3 4 5 6]

Window:

No	0	1	2	3	4
packet	4	5	6	7	8

Ack: expect packet 7

同情况（2）滑动窗口需要移动步长为 3，但是文件已经读完了。那么就不读了，只需要将 window 中左边的包删去即可。这样的话，本次发的包只有需要的 7 号包，只有一个。Window 窗口现在为：

No	0	1
packet	7	8

5）由这几种情况可知，当 ack 到达时，先判定滑动窗口步长，将窗口滑动后，再将需要应答 ack 的包发送出去，单次应答 ack 多个包，效率可能会好一点。

6）漏了一点，超时重发的情况。在上面的决策过程中，始终保持着 window 中的包都是在过程中的。在发送包的同时，用一个关键字 self.resendno 标记需要超时重发的包号，并且更新该值为刚刚发出的包。因此当遇到超时的情况，将发出 sender 上一次发出的那个包。尽管不一定是 receiver 需要的包，但是这个包如果没有丢失，也可以拿到 receiver 的 ack 进行下一步操作。

测试

该 project 提供了测试框架，主要还是不会用，因此首先是看测试的文档或者代码。通过代码注释得知，TestHarness 是整个自动化测试的框架，它通过在函数 tests_to_run 注册的测试用例，在 start 函数中启动 sender 和 receiver 脚本，遍历每个测试用例进行测试，测试的结果是通过传输后文件和源文件进行 MD5 校验判断的，测试成功会输出“Test passes”。这块通过文档理解不够透彻，因此读了很多的代码，花费的时间较长。

1. 测试用例样例

测试用例在框架代码中已经给出了两个例子，一个是 BasicTest，一个是 RandomDropTest，其中 baseTest 是测试用例的框架，原则上是不丢包的。在 RandomDropTest 中该类重写了这个 BaseTest 的 handle_packet 函数，有 50%的概率将丢包。

这就解释了之前我们为什么说 ack 包的丢失是可能的，我在这里将要丢的包打印的操作，发现这里不仅会丢数据包，还会丢 ack 包，之前的策略在这种情况下也是没有问题的。

2. 测试用例的编写

框架代码中给出了两个测试用例，RandomDropTest 会 50%丢包，但这只是会产生丢包的情况，还有两种情况没有解决，一个是数据包损坏，一个是环路。环路会产生超时，这个我这里直接简单归并到超时的情况中，但是需要编写数据包损坏的测试用例。该用例和 RandomDrop 差不多，不过是在不丢包的情况下，有 50%的概率数据包损坏。该测试用例如下：

```
class AllCaseTest(BasicTest):
    def handle_packet(self):
        for p in self.forwarder.in_queue:
```

```
                if random.choice([True, False]):
                    # print "AnDJ-AllCaseTest: drop this packet"
                if random.choice([True, False]):
                    p.checksum = str(int(p.checksum) - 1)
                    # print 'AnDJ-AllCaseTest: change packet checksum',
                # else:
                # print 'AnDJ-AllCaseTest: not change packet checksum',
                self.forwarder.out_queue.append(p)

            # empty out the in_queue
            self.forwarder.in_queue = []
```

3. 测试结果，丢包和数据包损坏的测试用例都是通过的。

遇到的问题及解决

文件格式问题。
框架代码中给出的测试是：README 文本文件，UTF-8 存储格式。

框架代码的 sender 和 receiver 给出的文件读写方式为 "r"和"w"。这样的读写是针对文本文件读写的，而且在 windows 上测试的话，因为 windows 上的编码格式为 GBK，因此在此系统上 README 的发送会出现问题，这个问题是，文本内容都是一样的，但是由于两个文件的编码格式不同，因此存储的大小不同，MD5 匹配不同，因此测试不通过。Linux 上测试通过验证了这个想法。

同时，非 TXT 格式文件也不能通过测试。

因此，我这里采取的建议是，给出 Linux 和 Windows 上的两种读的选择，但是，希望老师如果在 Windows 系统上测试的话，那么希望能够在 receiver 的文件写操作上将写操作的“w”修改为“wb”

附加分

无

Sender.py 文件结构

```
'''
This is a skeleton sender class. Create a fantastic transport protocol here.
'''
class Sender:

    # Waits until packet is received to return.
    def receive(self, timeout=None):

    # Sends a packet to the destination address.
```

```
def send(self, message, address=None):

# Prepares a packet
def make_packet(self, msg_type, seqno, msg):

# Split the given packet
def split_packet(self, message):

# Handles a response from the receiver.
def handle_response(self, response_packet):

def __init__(self, dest, port, filename, debug=False):

# handle the case which need read a packet from file to the window
def handle_new_ack(self, ack):

# handle the duplicate case.
def handle_dup_ack(self, ack):

# Main sending loop.
def start(self):

# skip the window from left to right by length step.
def skip_window(self, length):

# handle the timeout case which may be caused by packet loss or circle rote.
def handle_timeout(self):

# split the ack.
def split_ack_packet(self, message):
```