PJ GBN实现

实验目的

- 基于UDP设计一个简单的GBN协议，实现**双向**可靠数据传输。
- 在此基础上改进并实现SR协议

实验过程

实现GBN协议

发送方

- 初始化

  设置发送方socket、发送地址以及要发送的文件，同时初始化发送方的基序号(base)和下一个序列号(nextSeq)为0

  设置滑动窗口大小为3，超时时间为10秒，模拟丢包的概率为10%

  将待发送文件进行分组并保存在dataList中等待发送

  packets缓存在发送过程中已发送的分组，重传时从该缓存中取出需要的分组进行发送

  > 序列号的范围为256，因此缓存的大小也为256

```
WINDOW_SIZE = 3
TIMEOUT = 10
LOSS_RATE = 0.1
RECV_BUFFER = 4096

def __init__(self, sender_socket, addr, file):
    # init
    self.senderSocket = sender_socket
    self.address = addr
    self.file = file
    self.windowSize = WINDOW_SIZE
    self.timeout = TIMEOUT
    self.lossRate = LOSS_RATE
    self.base = 0
    self.nextSeq = 0
    self.dataList = []
    self.packets = [None] * 256

    while True:
        data = file.read(2048)
        if len(data) <= 0:
            break
        self.dataList.append(data)
```

- 主函数`run`

通过循环依次将待发送文件的分组通过`rdt_send`发送，发送失败意味着发送窗口已满，这时转到`rdt_rcv`接收来自接收方返回的ack

- rdt_send

  发送来自上层分组，首先判断发送窗口是否已满，未满则发送数据包，已满则拒绝发送

  处理待发送的数据，首先使用`make_pkt`将数据打包成分组，然后使用`udt_send`发送该分组

  发送完成之后，如果此时的base等于nextSeq，说明是发送序列的第一个数据包，需要启动超时计时器，在超时后重新发送序列

  同时将已发送的分组缓存到packets并更新nextSeq

```
def rdt_send(self, data):
    if self.nextSeq < self.base + self.windowSize:
        sndpkt = self.make_pkt(data)
        self.udt_send(sndpkt)
        if self.base == self.nextSeq:
            self.start_timer()
        self.packets[self.nextSeq] = sndpkt
        self.nextSeq = (self.nextSeq + 1) % 256
        return True
    else:
        return False
```

- make_pkt和udt_send

  make_pkt是将数据打包成分组的函数，利用struct结构，依次将序列号、checksum和数据打包进分组

  udt_send将打包好的数据包发送到指定地址中，在发送前要根据设置好的丢包率模拟丢包

- rdt_rcv

  接收来自接受方返回的ack和期待的下一个序列号

  对于接送到的分组，如果接收方期待的下一个序列号和发送发的base相同，说明接收到冗余的ack，此时不做任何处理并继续接收来自接受饭的ack

  如果不是冗余的ack，则向前移动滑动窗口（base+1），并且当base等于nextSeq时，说明窗口内已发送的分组全部被正确接收，可以停止超时计时器，否则重新启动超时计时器

  当出现超时事件时，重新发送在窗口中的数据包

  如果累计超时超过5次，则说明连接可能已经断开

```
def rdt_rcv(self):
    terminal = 0
    while True:
        # terminal while timeout 5 times
        if terminal >= 5:
            return False
        try:
            pkt, addr = self.senderSocket.recvfrom(RECV_BUFFER)
            expect_seq = pkt[0]
            ack = pkt[1]
            if self.base == expect_seq:
                # drop duplicate packet
                pass
            self.base = (ack + 1) % 256
```

```
            if self.base == self.nextSeq:
                # finish
                self.stop_timer()
            else:
                # restart timer
                self.start_timer()
            return True
        except socket.timeout:
            for i in range(self.base, self.nextSeq):
                self.udt_send(self.packets[i])
            terminal += 1
```

- calcChecksum和notCorrupt

  根据数据计算出checksum以及判断数据和checksum是否匹配

接收方

- 初始化

  设置接收方socket和接收文件

  在GBN协议中，对于接收方只需要维护期待的下一个序列号（expectSeq），初始时设为0

- 主函数`run`

  循环接收来自发送方的分组，如果接收到失序的数据包，则将上一个正确接收到的数据包返回

- rdt_rcv

  接收来自发送方的数据

  将接收到的分组解包分析（extract_data），并且根据接收地址设置发送地址

  根据解包出的序列号判断是否和期待的序列号相同，并且判断接收到的数据是否正确

  如果接收的分组无误，将数据交付到当上层文件，并且更新期待的下一个序列号

  然后将期待的下一个序列号和接收到的序列号打包缓存并发送回发送发

  > 接收到的序列号作为ack返回

```
def rdt_rcv(self):
    pkt, addr = self.recvSocket.recvfrom(RECV_BUFFER)
    self.address = addr
    seqnum, checknum, data = self.extract_data(pkt)
    if seqnum == self.expextSeq and notCorrupt(checknum, data):
        self.deliver_data(data)
        self.expextSeq = (self.expextSeq + 1) % 256
        self.sndpkt = self.make_pkt(self.expextSeq, seqnum)
        self.udt_send(self.sndpkt)
        return True
    else:
        return False
```

- extract_data和deliver_data

  extract_data将接收的分组进行解包，其中第一个数据为序列号，第二个为checksum，之后的都是数据

deliver_data将接收的数据写入到接收文件中

主函数

因为要实现双向数据传输，所以通过多线程的方式同时创建两个发送线程和两个接收线程

两组线程分别在12000端口和12001端口发送和接收数据

实现SR协议

发送方

- 初始化

与GBN协议不同的是，发送方在维护发送窗口的同时还需要对收到的ack进行确认，并且为每个已发送的分组维护一个超时计时器

rcvCheck是一个确认分组标记列表，用于确认接受方已经正确接收到分组

timers中保存每个已发送但未确认的分组的超时计时器

```
def __init__(self, sender_socket, addr, file):
    # init
    self.senderSocket = sender_socket
    self.address = addr
    self.file = file
    self.windowSize = WINDOW_SIZE
    self.timeout = TIMEOUT
    self.lossRate = LOSS_RATE
    self.send_base = 0
    self.nextSeq = 0
    self.dataList = []
    # cache packets for resend
    self.packets = [None] * 256
    self.rcvCheck = [False] * 256
    self.timers = {}
```

- rdt_send和udt_send

  与GBN协议不同的是，不需要再判断是否为发送序列的第一个分组来启动超时计时器，而是在每次发送一个分组时启动一个计时器

  每次使用udt_send发送一个分组前，先将要发送的分组在rcvCheck中取消标记，说明分组已发送但未接收

  然后使用start_timer为将要发送的分组启动一个计时器

  最后和GBN协议中一样根据丢包概率模拟丢包

- rdt_rcv

  用于接收来自接收方的确认

  每收到一个确认ack，就在rcvCheck中进行标记，表示接收方正确接收到ack对应的分组并停止对应的计时器

  如果接收到的ack是发送窗口base处的分组，则更新发送窗口，将base向前移动到最近的未确认分组

```
if self.send_base == ack:
    # update send_base
    while True:
        self.send_base = (self.send_base + 1) % 256
        if not self.rcvCheck[self.send_base]:
            break
```

同时在每次接受时设置一个计时器，如果累计5次超时，则说明可能已经断开连接，可以关闭socket

- set_timer、start_timer和stop_timer

  set_timer根据序列号创建对应分组的计时器线程，当超时事件发生时调用回调函数resent来重发对应分组

  start_timer将创建的计时器线程保存到timer中进行维护，并启动计时器

  stop_timer关闭序列号对应分组的计时器

- resent

  出现超时事件时的回调函数，首先停止计时器，然后重发对应的分组

接受方

- 初始化

  与GBN协议不同的是，接收方也需要维护一个接收窗口

  其中rcv_base为窗口的基地址，recvCheck用于确认接收到的分组，recvPackets缓存已接受的分组

```
def __init__(self, recv_socket, out):
    self.recvSocket = recv_socket
    self.timeout = TIMEOUT
    self.windowSize = WINDOW_SIZE
    self.lossRate = LOSS_RATE
    self.rcv_base = 0
    self.address = None
    self.outputFile = out
    # for base update
    self.recvCheck = [False] * 256
    # cache receive packets
    self.recvPackets = [None] * 256
```

- rdt_rcv

  接收来自发送方的分组

  如果接收的分组在接收窗口内，则缓存该分组并在recvCheck中标记已接收同时返回ack

  如果接收到的是基地址处的分组，则向上层交付已接受的数据

- deliver

  向上层交付分组数据，同时更新基地址到最近还未接收的分组

```python
def deliver(self):
    while True:
        # deliver data and update base
        data = self.recvPackets[self.rcv_base]
        self.outputFile.write(data)
        self.unmark(self.rcv_base)
        self.rcv_base = (self.rcv_base + 1) % 256
        if not self.recvCheck[self.rcv_base]:
            break
```

主函数

主函数实现和GBN一样

实验结果

> 为了方便观察实验结果，以单向传输为例

- GBN

  正常发送一个分组：

  ```
  make pkt 0
  send packet  0 successfully
  receive pkt seq  0 Receiver expected seq  0
  Not Corrupt
  change expected seq to  1
  make pkt, expected num is  1 ack is  0
  Receiver send ACK: 1
  start timer
  nextSeq= 1
  ```

  发送方首先打包序列号为0的分组，同时启动计时器，并成功将分组发送到接受方

  接收方确认分组后更新期待的序列号并返回ack0

```
nextSeq= 1
make pkt 1
send packet  1 successfully
receive pkt seq  1 Receiver expected seq  1
Not Corrupt
change expected seq to  2
make pkt, expected num is  2 ack is  1
Receiver send ACK: 2
nextSeq= 2
make pkt 2
send packet  2 successfully
receive pkt seq  2 Receiver expected seq  2
Not Corrupt
change expected seq to  3
make pkt, expected num is  3 ack is  2
Receiver send ACK: 3
nextSeq= 3
refuse data
```

接收方连续发送发送窗口内的分组（窗口大小为3），当发送第四个分组时（序列号为3），因为发送窗口已满而拒绝发送分组

发送过程中出现丢包：

```
make pkt 8
send packet  8 but Packet lost.
nextSeq= 9
refuse data
receive ack  6
start timer
make pkt 9
send packet  9 successfully
receive pkt seq  9 Receiver expected seq  8
Receiver send ACK: 7
```

在发送序列号为8的分组时出现丢包，接收方没有接收到序列号为8的分组，但接收到了下一个序列号9的分组；此时接收方期待的序列号为8，因此接收方丢弃这个分组并继续返回ack为7的确认

```
Timeout!
resend pkt  8
send packet  8 successfully
receive pkt seq  8 Receiver expected seq  8
Not Corrupt
change expected seq to  9
make pkt, expected num is  9 ack is  8
Receiver send ACK: 8
resend pkt  9
send packet  9 successfully
receive pkt seq  9 Receiver expected seq  9
Not Corrupt
change expected seq to  10
make pkt, expected num is  10 ack is  9
Receiver send ACK: 9
resend pkt  10
send packet  10 successfully
receive pkt seq  10 Receiver expected seq  10
Not Corrupt
```

当出现超时事件时，将发送窗口内的分组重新发送，即序列号为8、9、10的分组，并且接收方成功接收到分组

- SR

  正常发送一个分组：

```
make pkt 0
start timer  0
send packet  0 successfully
receive pkt seq  0
Not Corrupt
deliver data
update rcv_base to  1
make pkt, ack is  0
Receiver send ACK: 0
```

发送方发送一个序列号为0的分组，接收方成功接收斌且分组序列号和接收方窗口基序号相同，因此向上交付接收到的数据并返回ack0

```
update rcv_base to  1
make pkt, ack is  0
Receiver send ACK: 0
nextSeq = 1
make pkt 1
start timer  1
send packet  1 successfully
receive pkt seq  1
Not Corrupt
deliver data
update rcv_base to  2
make pkt, ack is  1
Receiver send ACK: 1
nextSeq = 2
make pkt 2
start timer  2
send packet  2 successfully
receive pkt seq  2
Not Corrupt
deliver data
update rcv_base to  3
make pkt, ack is  2
Receiver send ACK: 2
nextSeq = 3
refuse data
```

当接收方将发送窗口中的分组全部发送后，拒绝发送窗口外的分组

发送方分组丢失：

```
send packet  22 but Packet lost.
nextSeq = 23
refuse data
receive ack  20
stop timer  20
update send_base  21
make pkt 23
start timer  23
send packet  23 successfully
receive pkt seq  23
```

```
Timeout!
resent pkt 22
start timer  22
send packet  22 successfully
receive pkt seq  22
Not Corrupt
deliver data
update rcv_base to  25
```

发送方发送分组22时出现丢包，接收方并没有接收到分组22，而是接收到了下一个分组23；

当分组22的计时器出现超时时，发送发重发22分组，接收方接收到分组后，因为22分组和接收窗口基地址相同，所以向上交付数据并更新接收窗口

接收方出现丢包：

```
Receiver send ACK: 14 , but lost.
nextSeq = 15
```

```
Timeout!
resent pkt 14
start timer  14
send packet  14 successfully
receive pkt seq  14
Not Corrupt
make pkt, ack is  14
Receiver send ACK: 14
receive ack  14
stop timer  14
update send_base  17
```

接收方发送ack14时出现丢包，接收方14号分组因为没有收到ack而出现超时，然后重发14号分组；接收方重新接收到分组并返回ack；发送方接收到ack后停止计时器并更新接收窗口