## 软件运行环境

1. 在windows10 系统下
2. 协议软件使用python 语言编写是于用户空间而非内核空间运行，所以使用了抽象的物理层和网络层来和我们定义的 gobackN 的slidingwindow 方法(作为数据链路层)来进行交互
3. 此自定义的软件协议将以函数调用的方式运行。

## 对外接口分析和设计

1. 持续监听对应的地址接收到来的信息并对发送方返回对应的消息的listen函数
   1. 设计如下

    def listen(self,addr,peerAddr,buffer,recv\_buff): *# 客户端监听端口*

*self*.\_\_ready(addr,peerAddr,buffer)*#准备网络层和物理层*

*while* True:*# 无阻塞socket连接监控*

            psocket = *self*.physc\_layer.socket

*# 使用select来实现 非阻塞的接收，当readable的长度不为0的时候即有消息到达*

            readable, writeable, errors = select.select([psocket, ], [], [], 1)

*if* (len(readable) > 0):

                message, address = psocket.recvfrom(*self*.physc\_layer.buff\_size)

*if* message.decode() == '-testgbn': *#连接建立，开始信息交互*

*self*.gbn.send() # 通过数据链路层协议来发送消息

*# 将得到的消息存储到接收用的buffer 中,*

*# 以供接收方的*

                    recv\_buff.append(*self*.network\_layer.recvDataBuffer)

* 1. 这样通过在服务器端使用 该函数，就可以开启监听(使用一个单独的线程来执行该函数)。

1. 直接向指定的ip 地址的对应的 端口发送信息的 send函数，

    def send(self,addr,peerAddr,buffer):

*self*.\_\_ready(addr,peerAddr,buffer) *#准备网络层和物理层*

*self*.physc\_layer.socket.sendto('-testgbn'.encode(), *self*.physc\_layer.peerAdd)

*self*.gbn.send()# 通过数据链路层协议来发送消息

*#返回从发送端收到的消息*

*return* *self*.network\_layer.recvDataBuffer

* 1. 通过在用户的应用程序中使用该函数可以直接向 peerAddr对应的地址:端口发送信息，前提是对端必须使用了 listen函数来对指定的端口进行监听

## 协议软件设计

协议的整体如下，统合虚拟的网络层和物理层，以及数据链路层(gbn即数据链路层的gobackN协议)

class GBNNetWork:

    def \_\_init\_\_(self):

*self*.network\_layer = NetWorkLayer()*#网络层构建*

*self*.physc\_layer = PhysicalLayer(1024) *#物理层构建*

*self*.gbn = GBN(*self*.physc\_layer,*self*.network\_layer) *#数据链路层构建*

def ……………………

def ……………………

……………………

……………………

### 虚拟网络层

虚拟网络层的主要任务就是向数据链路层的SlidingWindow中放入网络层发送缓冲区的数据（from\_network\_layer），此外将使用接收缓冲区来接收数据链路层接收到的对端数据(to\_network\_layer)

1. from\_network\_layer

    def from\_network\_layer(self,send\_windows):

*if* (*self*.nextseqnum > *self*.pkg\_num):

*return* False *# 没有数据了*

*# 将第nextseqnum-1个包加入窗口内，并封装成数据,type为0*

        data = Data(*self*.buffer[*self*.nextseqnum - 1], 0, seq=*self*.nextseqnum)

        send\_windows.append(data)*# 向指定的window中塞入消息*

*self*.nextseqnum += 1

*return* True *# 还有数据*

* 1. 逻辑和语句的解释如上，数据链路层将在实现中调用此函数(即调用上层的接口)，给定接收数据用的 window 即可向window中塞入消息

1. to\_network\_layer

    def to\_network\_layer(self,revData):

*if*(revData == 1):*# 告知网络层发送结束*

            print(' 发送/接收完毕, 退出')

*else*:*# 将得到的数据保存下来*

*self*.recvDataBuffer += revData

* 1. 数据链路层确认到接收的消息是需要的消息之后将其传递给网络层，让网络层进行接收处理

### 虚拟数据链路层(gobackN协议)

虚拟的数据链路层为本协议软件的主要内容，用于应对来自物理层的错误(丢包情况)。由于代码较长故不贴出，简要展示其流程:

1. Send() 函数，即gobackN协议的运行主体,以下操作位循环
   1. 处理超时情况，标记window中需要重新发送的数据，因为发送出去的数据将不会在存储在 send\_windows中(因为窗口移动)

*# 当超时后，将窗口内的数据更改为未发送状态*

*if* pkg\_timer > *self*.max\_time:

                resend = []

*for* data in *self*.send\_windows:

                    data.state = 0*#设置为未发送状态*

                    resend.append(data.seq)

*if* len(resend) > 0:

                    print(' 发生超时，重传', resend)

* 1. 检查窗口，如果窗口中的消息有空余，则向网络层请求(调用 from\_network\_layer)
     1. 如果网络层没有消息了，则发送完毕退出函数

*# 窗口中数据少于最大容量时，尝试添加新数据*

*while* len(*self*.send\_windows) < *self*.N:

*if*(not *self*.nl.from\_network\_layer(*self*.send\_windows)):

*break*

*# 窗口内无数据则退出总循环*

*if* not *self*.send\_windows:

*if* pkg\_timer > *self*.max\_time and msg\_timer > *self*.wait\_time: *# 发现网络层 超时 即说明信息中断 发送被中断了*

*self*.nl.to\_network\_layer(1) *# 告知网络层传输结束*

*break*

* 1. 遍历窗口，发送被标识为未发送的数据。

*# 遍历窗口内数据，如果存在未成功发送的则发送*

*for* data in *self*.send\_windows:

*if* not data.state:

                    print(' 发送数据 ', data.seq)

*self*.pl.to\_physical\_layer(data,*self*.pl.peerAdd)

                    data.state = 1*# 设置状态为1 表示发送成功*

* 1. 尝试从物理层获取数据，如果有数据 则进行 gobackN的具体逻辑  
     具体将在 算法和数据结构部分讲解
     1. Ack对比，
     2. 接收计时

### 虚拟物理层

虚拟物理层在本虚构协议软中的任务是使用socket 来进行对端的交互进行数据的收发，并模拟丢包状况。以此来模拟真实的物理层。

1. from\_physical\_layer

    def from\_physical\_layer(self):

*# 使用 select来进行无阻塞socket连接监控*

*# 不同于 直接使用 recvfrom ，这种方式先进行socket 的缓冲区监察*

*# 如果有数据得到 即为 可读，然后再调用 recvfrom*

        readable, writeable, errors = select.select([*self*.socket, ], [], [], 1)

*if* len(readable) > 0: *# socket接收到消息了*

            message,address = *self*.socket.recvfrom(*self*.buff\_size)

            msg = message.decode()*# 对信息进行解码*

            ackNum = int(msg[0])*# 获取信息的 ack*

*if* random() < 0.2: *# 丢包模拟 概率为 0.2*

                print(' 模拟发生丢包, 丢失的包的seq为', str(ackNum))

*return* 0 *#发送了丢包。无消息到来*

*return* (message,address) *# 有消息到来*

*else*:

*return* 0 *#无消息到来*

* 1. 数据链路层将会调用 该函数从物理层获取 接收到的信息，如果不为0则代表有消息到来

1. to\_physical\_layer

    def to\_physical\_layer(self,data,address):

*# 使用socket来向指定的 address发送指定的data*

*self*.socket.sendto(str(data).encode(),address)

## 主要算法与数据结构

### 数据结构

1. 数据结构主要是 Data,代表的即为物理层传输的抽象帧

class Data(object):

    def \_\_init\_\_(self, msg, type, seq=0, state=0):

*self*.msg = msg.decode()

*self*.type = str(type)

*self*.state = state

*self*.seq = str(seq)

    def \_\_str\_\_(self):

*return* *self*.seq + *self*.type + *self*.msg

* 1. msg为报文内容 字节序列
  2. type表示类型，0为数据报文，1为ack报文
  3. seq表示序列号
  4. state只在发送端使用，不发送到接收端 state为0表是带发送，为1表示以发送， 为2表示以发送且已收到ACK

### 算法

1. 算法主要涉及到 gobackN 的帧发送和接收实现逻辑

physical\_msg = *self*.pl.from\_physical\_layer()

* 1. **先尝试从物理层获取信息，如果有信息则进行接收操作**

*if* physical\_msg != 0: *# 判断物理层是否有消息*

    message = physical\_msg[0]; address = physical\_msg[1]

msg = message.decode()

* + 1. **发现是ACK，不带有数据**

*if* msg[1] == '1':

*# 收到ACK， 重新计时*

        pkg\_timer = 0

        print(' 收到ACK ', msg[0])

        ack\_num = msg[0]

* + - 1. **判断ack 的范围，以此来更新发送窗口send\_windows**

*for* i in range(len(*self*.send\_windows)):

*if* ack\_num == *self*.send\_windows[i].seq:

*self*.send\_windows = *self*.send\_windows[i + 1:]

*break*

*else*:

* + 1. **非ACK，说明是收到的数据**

        pkg\_timer += 1

        print(' 收到MSG = ', msg[0])

        ackNum = int(msg[0])

        msg\_timer = 0

* + - 1. **判断ack 是否是需要的ack**

*if* last\_ack == ackNum – 1:

1. **如果是，则发送ack 表示消息到达**

*self*.pl.to\_physical\_layer(Data(''.encode(), 1, ackNum),address)

            print(' 发送ACK ', str(ackNum))

            last\_ack = ackNum

*self*.nl.to\_network\_layer(msg[2:])  *# 将得到的信息传送到网络层*

1. **如果不是，则发送上一次的ack告知对端重发指定的信息**

*else*:

            print(' 收到的MSG不是需要的，发送当前收到的最大的ACK ', last\_ack)

*self*.pl.to\_physical\_layer(Data(''.encode(), 1, last\_ack),address)

* 1. **如果物理层没有接收到信息，则增加计时器，以在超时的时候重发**

*else*:

    pkg\_timer += 1

    msg\_timer += 1

## 测试环境设计

测试环境即为windows10 系统下。 使用python编写出服务器端和客户端，让二者进行消息的交互

1. 客户端  
   客户端读取 client\_send.txt 文件夹的内容，准备发送给服务器端

*from* GBN *import* GBNNetWork

* 1. **客户端构建 GBNNetWork即 gobackN协议接口**

gbn = GBNNetWork()

buffer = []

* 1. **客户端读取**client\_send.txt**中的文件**

*with* open('client\_send.txt', 'rb') as f:

*while* True:

        seq = f.read(500)

*if* len(seq) > 0:

            buffer.append(seq)

*else*:

*break*

* 1. **客户端调用接口来向服务器端发送消息，并获取打印出得到的响应消息**

response = gbn.send(('127.0.0.1', 12345),('127.0.0.1',31500),buffer)

print(response)

1. 服务器端  
   服务器端读取 server\_send.txt 文件中的内容，作为对发送方的响应消息

*from* GBN *import* GBNNetWork

*from* threading *import* Thread

recvBuffer = []

* 1. **定义 Sever类，用作监听的主体。通过使用线程方法来为监听单独分配一个线程以来持续监听来自指定端口的信息，并使用 recvBuffer来存储服务器接收到的消息**

class Server (Thread):

    def run(self):

        global recvBuffer

        buffer = []

*with* open('server\_send.txt', 'rb') as f:

*while* True:

                seq = f.read(500)

*if* len(seq) > 0:

                    buffer.append(seq)

*else*:

*break*

        gbn = GBNNetWork()

        gbn.listen(('127.0.0.1', 31500),('127.0.0.1', 12345),buffer,recvBuffer)

server = Server()

server.start()

* 1. **程序的应用部分，当 recvBuffer的长度不为0 则说明有数据达到，则用户在服务器端的其它线程中可以判断和使用这些接收到的数据**

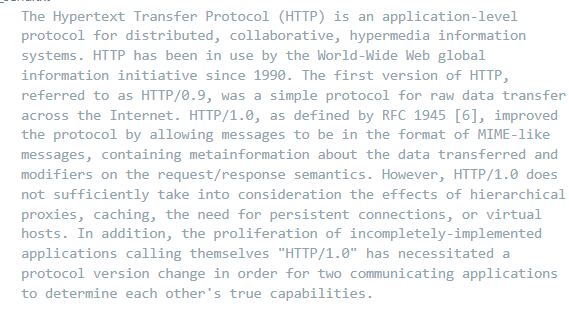
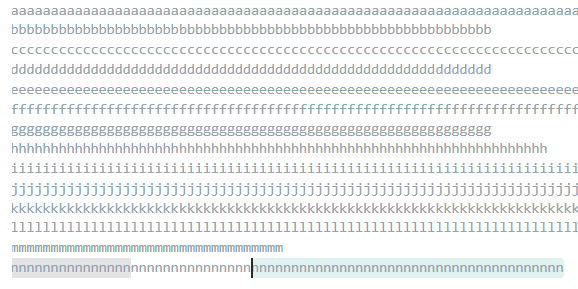
*while*(True):

*if*(len(recvBuffer)!=0):

        print("\n 收到的信息为:\n"+recvBuffer[0])

        recvBuffer = []

## 测试执行结果分析

1. 数据设置
   1. 客户端发送的数据：  
      
   2. 服务器端的响应数据:  
      
2. 运行结果:
   1. 运行的具体结果如下，会打印出每一次交互进行的操作内容
   2. 对于异常状态的处理
      1. 丢包  
         对于丢包的情况，无论是客户端还是服务器端 都可以在计时到了之后进行重发操作，如下图中所示的绿色部分。服务器端接收到的序号2的包丢失了，所以后继接收到的 序号3 的包 后向客户端返回了ACK 1 告知客户端服务器端只接收到了序号1 的包。这样 客户端之后推进send\_windows 的时候只会推进到 序号2的包为止，并且之后将会触发客户端的计时器到时：  
         
      2. 超时  
         如下图的 红色部分，由于 ack消息的丢失，导致了 服务器端的计时器没有得到重置，所以这里将会触发超时情况，让服务器端向客户端重新发送序号 1 之后的包

