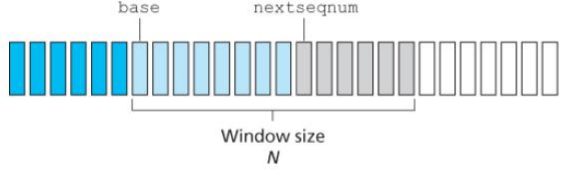




哈尔滨工业大学  
Harbin Institute of Technology

# 计算机网络 课程实验报告

实验名称	可靠数据传输协议——停等协议和 GBN 协议的设计与实现					
姓名	刘璟烁		院系	计算机科学与技术学院		
班级			学号			
任课教师			指导教师			
实验地点			实验时间			
实验课表现	出勤、表现得分(10)		实验报告 得分(40)		实验总分	
	操作结果得分(50)					
教师评语						

实验目的:
理解可靠数据传输的基本原理; 掌握停等协议的工作原理; 掌握基于 UDP 设计并实现一个停等协议的过程与技术。 理解滑动窗口协议的基本原理; 掌握 GBN 的工作原理; 掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术。
实验内容:
1)基于 UDP 设计一个简单的停等协议, 实现单向可靠数据传输 (服务器到客户的数据传输) 。 2)模拟引入数据包的丢失, 验证所设计协议的有效性。 3)改进所设计的停等协议, 支持双向数据传输; 4)基于所设计的停等协议, 实现一个 C/S 结构的文件传输应用。 5)基于 UDP 设计一个简单的 GBN 协议, 实现单向可靠数据传输 (服务器到客户的数据传输)。 6)模拟引入数据包的丢失, 验证所设计协议的有效性。 7)改进所设计的 GBN 协议, 支持双向数据传输; 8)将所设计的 GBN 协议改进为 SR 协议。
实验过程:
1.剖析实验原理 (1).GBN可靠数据传输协议原理 GBN使用 <b>流水线机制</b> 进行数据传输, 使用该协议的发送方在收到ACK 之前可以连续发送多个分组, 并且GBN 设置相应的缓存空间以缓存从应用层下达的分组。同时, 发送方使用一组序列号标识分组, 从而防止接收方重复上传同一分组给应用层。在发送过程中, GBN设置滑动窗口用于限制发送速度, 滑动窗口尺寸对应发送下一个分组前允许的最多尚未被确认的分组数。GBN应用了 <b>累计确认机制</b> , 接收方发送带有确认接受的分组序列号的ACK确认报文给发送方, 用于确认收到该序号之前的所有分组。 GBN 发送方为数据传输维护了一个定时器, 用于 <b>超时重传</b> , 当收到ack时重启或关闭定时器, 如果发生超时, 则重新发送已发送过但未被确认的所有分组。
<div><div><div><div><div>base</div><div>nextseqnum</div></div><div></div><div><div>Key:</div><div><div>Already ACK'd</div><div>Usable, not yet sent</div><div>Sent, not yet ACK'd</div><div>Not usable</div></div></div></div></div><div>图1.GBN协议的滑动窗口</div><p>GBN协议中发送方的状态机图示如下:</p></div>

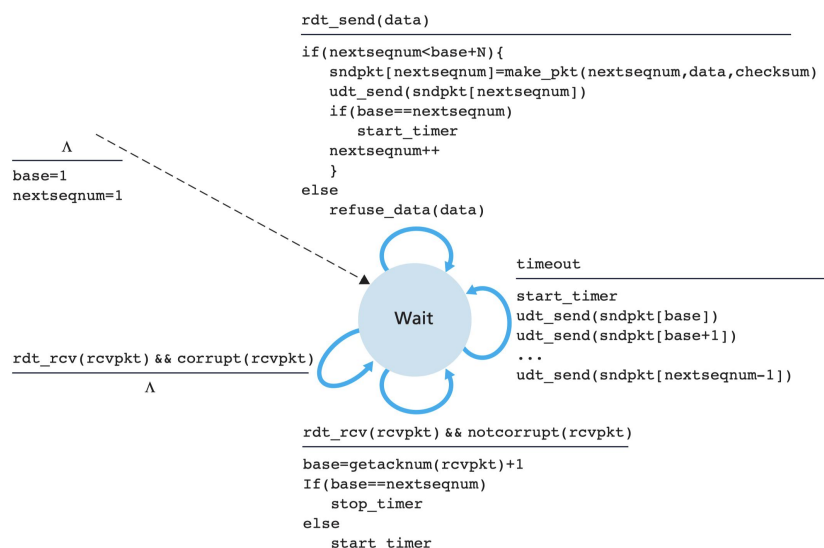


图2.发送方FSM

接收方FSM如下:

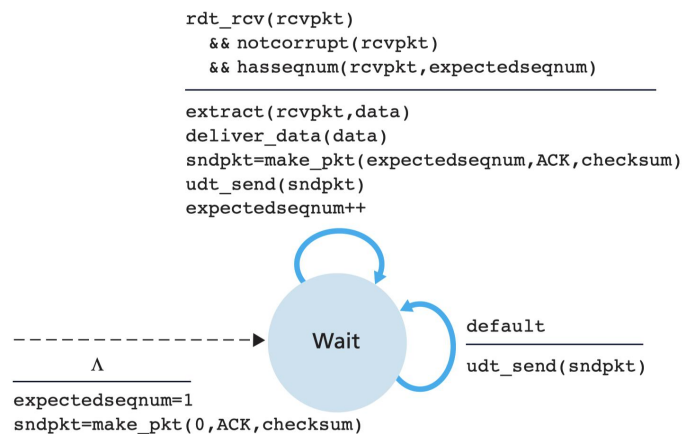
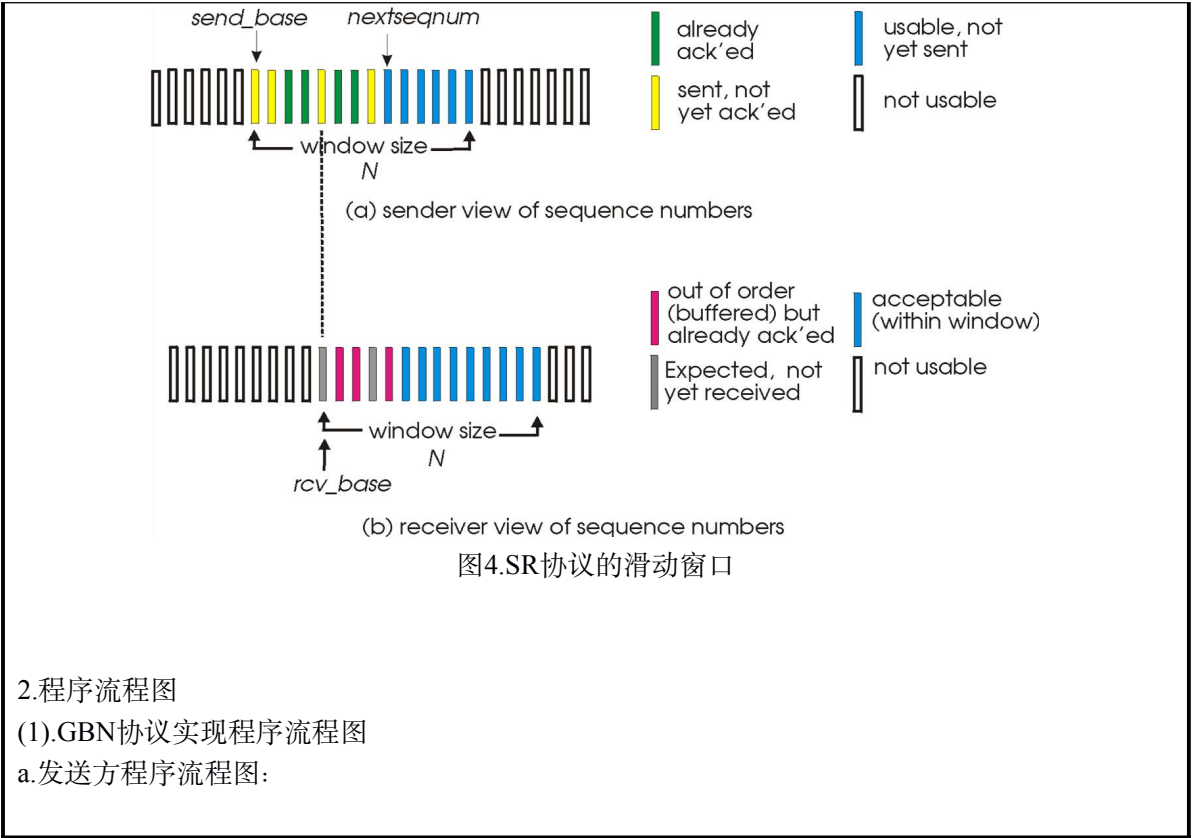


图3.接收方FSM

## (2)SR可靠数据传输协议原理

SR 协议区别于GBN的一点是**放弃使用累计确认机制**，它要求接收方设置缓存用来存储乱序到达的分组，并单独为每个分组发送ACK确认分组到达，类似于发送方的滑动窗口，接收方以滑动窗口的形式缓存收到的数据分组并在按序收到分组时移动窗口，SR 发送方为每个分组维护一个定时器，当超时事件发生时，只重传那些超时时间内未收到 ACK 的分组，而接收方等到数据分组已经按序排好后，将连续的缓存分组传给应用层。

如图为发送方和接收方的滑动窗口：



2.程序流程图

(1).GBN协议实现程序流程图

a.发送方程序流程图:

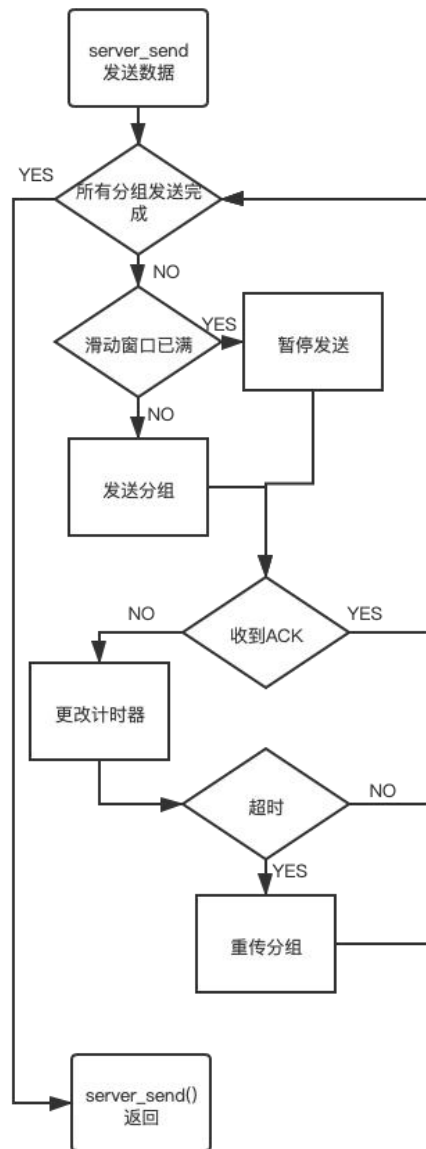


图5.GBN发送方程序流程图

b.接收方程序流程图

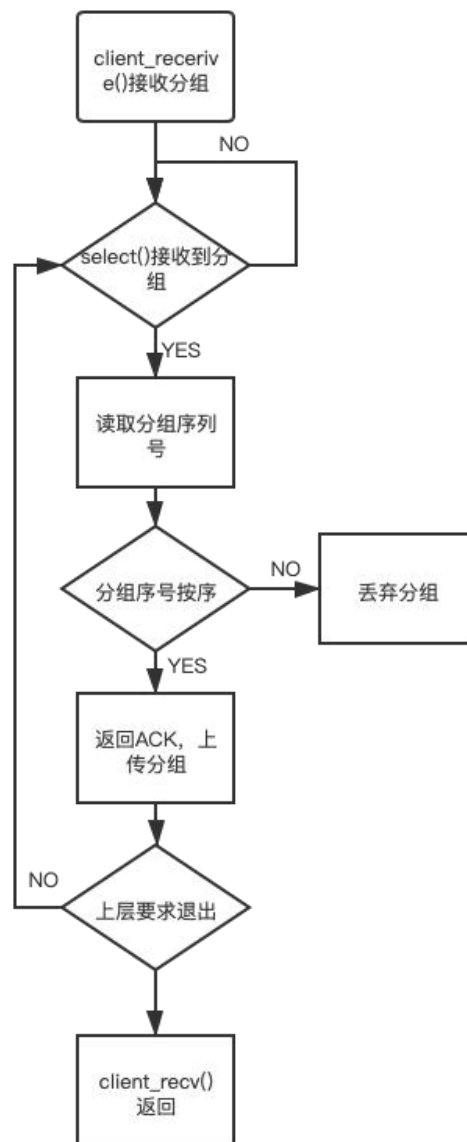


图6.GBN接收方流程图

(2).SR协议实现程序流程图

a.发送方:

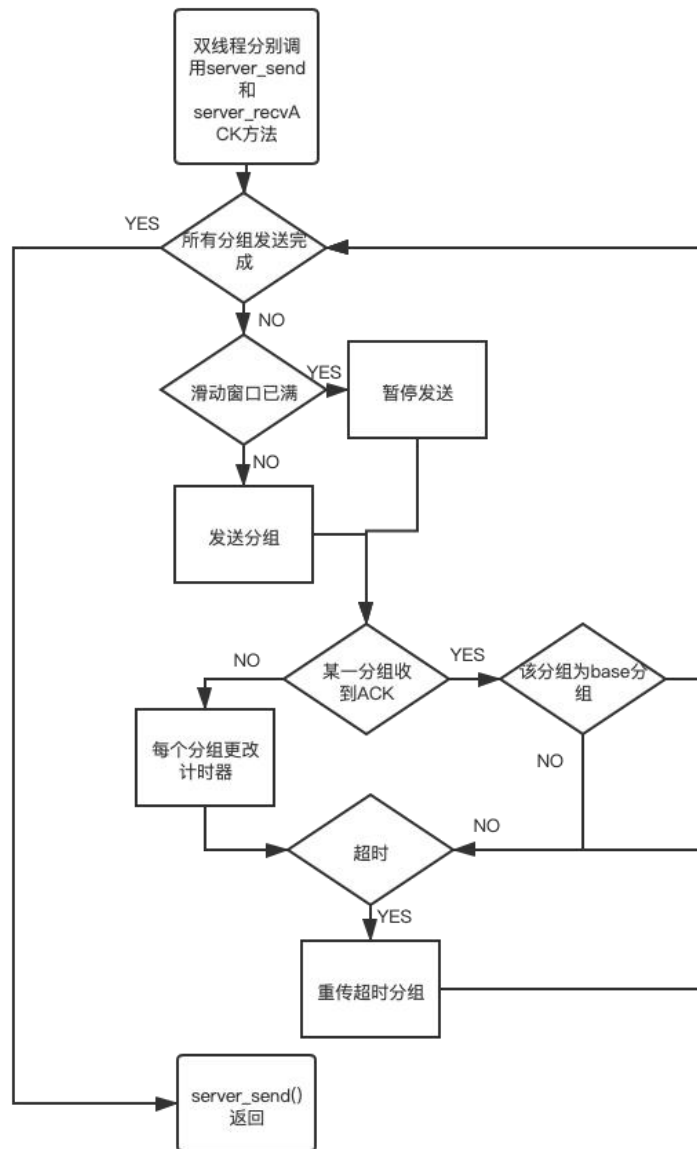


图7.SR发送方

b.接收方:

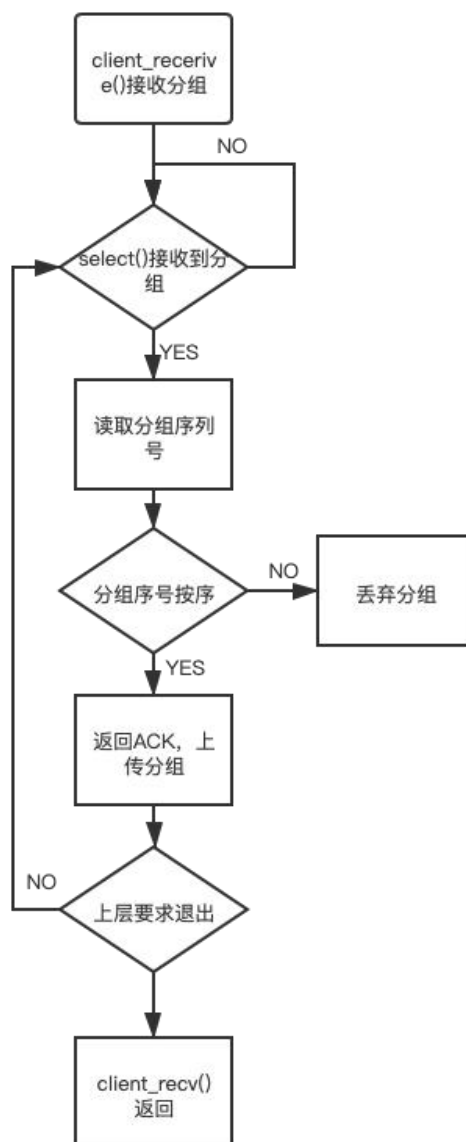


图8.SR接收方程程序框图

### 3.实验各要求实现过程

#### (1).GBN协议基本功能实现

规定:

数据分组格式: “seq data”, seq为分组的序号, data为报文信息, 比如 “2 This is a message.”

ACK格式: “ACK seq”, seq为被确认的分组的序列号, 比如 “ACK 2”

最大序列号: 16

超时间隔: 3s

滑动窗口大小: 4

##### a.发送方实现过程:

对于GBN协议的单向传输功能, 我们设定server为发送方, 需要对其实现能够发送分组 (server\_send()方法) 和接收ACK (server\_rcvACK () 方法) 的功能。



首先来说明server\_send()方法:

我们通过while循环来模拟不断向客户端发送数据的过程, 直到调用该方法的线程被强制结束。每一次循环中, 首先判断根据滑动窗口base、滑动窗口大小与最大序列号的关系判断滑动窗口是否已满:

```
if self.base + self.send_win < max_seq_num:

    if self.next_seq < self.base + self.send_win:

        ...

else: # 如果 base+滑动窗口大小已经超过最大序号, 则需要根据 next_seq 判断
      是否发报文

    if self.base <= self.next_seq < max_seq_num: # 可发送

        ...
```

这里需要注意, 当base + 窗口大小 < 最大序列号时可直接通过next\_seq判断窗口是否已满, 而base + 窗口大小 >= 最大序列号时, 需要分情况讨论以保证分组的正确发送。

当成功发送分组时, 我们将该分组放到缓存self.data\_cache\_sent中等待为其接收ACK, next\_seq加一并对最大序列号进行模运算。

随后说明server\_recvACK () 方法:

该方法同样使用while循环不断接收ACK消息, 调用select(), 当有ACK到来时判断是否为合理的序号范围的确认消息, 如果是, 则去除缓存区中已经被确认的分组, 并更改self.base:

```
if (self.next_seq < self.base and (ack_seq >= self.base or ack_seq <
self.next_seq)) \

    or (self.next_seq > ack_seq >= self.base): # 确保 ack 在等待相应序
      列号范围内

    if ack_seq >= self.base:

        self.data_cache_sent = self.data_cache_sent[ack_seq + 1 -
self.base:]

    else:

        self.data_cache_sent = self.data_cache_sent[ack_seq + 1 +
```

```
max_seq_num - self.base:]

    self.timer = 0 # 每收到一个 ack，重启定时器

    self.base = (ack_seq + 1) % max_seq_num
```

同时，该方法还实现了超时重传功能，在后面会提到。

注意，为了发送数据分组和接收从接收方发来的ACK同时进行，我们使用两个线程分别调用上述两个方法：

```
server_sendat_thread = threading.Thread(target=server.server_send)

server_recvack_thread = threading.Thread(target=server.server_recvACK)

# server 发送数据与接收 ack 同时进行
```

同时，由于两个线程均在同一个server对象上操作，为了避免多线程间的数据冲突，我们在上述两个方法每一次循环过程中均调用了锁来保证数据的安全修改：

```
while True:

    time.sleep(random.random()*0.5)

    self.lock.acquire()

    ...

    self.lock.release()
```

下面考虑接收方（client）：

client端调用receive\_from\_server()方法从服务器接收信息，方法调用while循环，每次循环读取接收到的分组并判断是否为按序传输：

```
while True:

    readable = select.select([self.recv_soc],[],[],1)[0]

    if len(readable)> 0:

        rec_pkt,addr = self.recv_soc.recvfrom(1024)

        data = rec_pkt.decode() # 为字节形式的分解码组

        seq = int(data.split())[0] # 解析分组中的序列号
```

```
if seq == self.expect_seq: # 按序收到了报文段

    if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expect_seq),(SERVER_IP,SERVER_PORT_SEN))

        print("客户端确认收到数据：" + data)

        self.expect_seq = (self.expect_seq + 1) % max_seq_num

    else: # 发送期待收到的报文序列号

        if random.random() > self.ack_loss_rate:

            self.recv_soc.sendto(pkt_format.ack_pkt(self.expect_seq - 1), (SERVER_IP, SERVER_PORT_SEN))

            print("客户端重新确认序列号" + str(self.expect_seq - 1))
```

## (2).实现超时重传

上面说到，server在while循环中调用select方法（设定超时间隔为1s）尝试接收ACK报文，每次循环中如果没有收到ACK，则时钟计数器加一，直到超过超时实验3s调用timeout\_resend方法：

```
def timeout_resend(self,remote_ip,remote_port):

    print("客户端响应超时，重新发送：")

    self.timer = 0

    for pkt in self.data_cache_sent:

        self.send_soc.sendto(pkt, (remote_ip, remote_port))

    print("客户端已重新发送：" + pkt.decode())
```

## (3).实现随机丢包：

在服务器端和客户端分别设定发送分组、ACK确认报文的丢包概率，并且在服务器端发送分组

时调用`random.random()`产生随机数，如果生成的随机数小于对应的丢包概率则并不真正发送，从而可以视作丢包：

```
data = "the seq of this mes is :"+str(self.next_seq)

new_pkt = pkt_format.data_pkt(self.next_seq,data)

if random.random() > self.send_loss_rate: # 随机发送丢包

    self.send_soc.sendto(new_pkt, remote_addr)
```

#### (4).双向数据传输

可以通过分别在server端和client端实现接收分组和发送分组的方式实现双向数据传输，同时启动相应的线程调用这些方法：

```
print("服务器开始向客户端发送数据：")

server_sendat_thread = threading.Thread(target=server.server_send)

server_recvack_thread = threading.Thread(target=server.server_recvACK)

# server 发送数据与接收 ack 同时进行

client_recvdat_thread =

threading.Thread(target=client.receive_from_server)

client_recvdat_thread.start()

server_recvack_thread.start()

server_sendat_thread.start()


print("客户端开始向服务器发送数据：")

client_sendat_thread = threading.Thread(target=client.client_send)

client_recvack_thread = threading.Thread(target=client.client_recvACK) #

server 发送数据与接收 ack 同时进行

server_recvdat_thread =
```

```

threading.Thread(target=server.receive_from_client)

server_recvdat_thread.start()

client_recvack_thread.start()

client_sendat_thread.start()

```

#### (5).SR协议基本功能实现

在SR原理部分提到，区别于GBN协议，SR协议的发送方为每个已发送的分组维护一个timer时钟，并且在该分组超时后单独重传该分组；同时，接收方分别为每个已发送分组发送ACK，因此接收方需要设置相应的缓存，可以通过字典来实现这些功能。

发送方分别设定时钟，每当有分组发送便以seq:0的键值对形式存入该dic，并且在接收到ACK后从该字典中将键值对去除：

```
self.timer = {} # 所有时钟
```

接收方设置缓存，每当有分组未按序到达，便将分组按照seq: data的键值对形式存入该缓存中：

```
self.recv_cache = {} # 缓存接收的分组
```

另外，对于接收方面言，接收到按序的分组时需要将其序列号相连的已经放入缓存的接收分组一并发送给应用层，并移动其滑动窗口：

```

if seq == self.recv_base:

    if random.random() > self.ack_loss_rate: # 发送 ack 进行确认

        self.recv_soc.sendto(pkt_format.ack_pkt(seq), remote_addr)

    seq_num = seq # 循环向应用层传输按序的数据分组

    while seq_num in self.recv_cache.keys():

        print("客户端确认收到数据：" + self.recv_cache[seq_num].decode())

        self.recv_cache.pop(seq_num)

        seq_num = (seq_num + 1) % max_seq_num

    self.recv_base = (seq_num + 1) % max_seq_num

```

#### (6).停等协议的实现

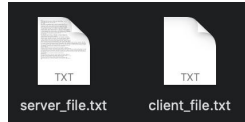
本次试验还需实现停等协议，可直接将GBN协议的滑动窗口大小设为1，并且应用文件读写功能即可，这里不再赘述。

**实验结果:**

采用演示截图、文字说明等方式, 给出本次实验的实验结果。

**1.停等协议:**

本次试验停等协议以GBN的实现基础上设置滑动窗口大小为1, 并且实现服务器与客户端的文件传输功能。服务器读取文件server\_file.txt, 发送给客户端并将数据写入client\_file.txt:



运行stopAndWait.py, 文件可以观察到两文件的内容一致, 以下为部分传输过程:

服务器开始向客户端发送数据:

服务器已发送: Start

客户端确认收到数据: Start

服务器已发送: Every single time you access a website, you leave tracks.

客户端响应超时, 重新发送:

服务器已重新发送: 1 Every single time you access a website, you leave tracks.

客户端确认收到数据: Every single time you access a website, you leave tracks.

(注: 响应超时由上述随机丢包机制造成)

**2.GBN协议:**

本次试验为GBN协议实现了双向数据传输、超时重传和随机丢包等过程, 下图为运行gbn.py的部分内容:

客户端响应超时, 重新发送:

服务器已重新发送: 0 the seq of this mes is :0

服务器已重新发送: 1 the seq of this mes is :1

服务器已重新发送: 2 the seq of this mes is :2

服务器已重新发送: 3 the seq of this mes is :3

客户端确认收到数据: 0 the seq of this mes is :0

客户端确认收到数据: 1 the seq of this mes is :1

客户端确认收到数据: 2 the seq of this mes is :2

客户端确认收到数据: 3 the seq of this mes is :3

客户端滑动窗口已满, 暂时无法发送数据

客户端滑动窗口已满, 暂时无法发送数据

这里可以看到在发送方丢包或者接收方的ACK丢失后会造成超时, 发送方会重新发送相应分组,

并且在再次收到ACK之前无法继续发送数据。

### 3.SR协议

SR协议分别确认每个分组的到来:

服务器开始向客户端发送数据:

服务器端已发送数据: the seq of this mes is :0

服务器端已发送数据: the seq of this mes is :1

ACK 1

服务器端已发送数据: the seq of this mes is :2

服务器端已发送数据: the seq of this mes is :3

ACK 3

服务器端滑动窗口已满, 暂时无法发送数据

服务器端滑动窗口已满, 暂时无法发送数据

客户端响应超时, 重新发送

服务器已重新发送: 0 the seq of this mes is :0

客户端响应超时, 重新发送

服务器已重新发送: 2 the seq of this mes is :2

如图, 当分组0和2丢失后一段时间超时, 服务器重新发送响应分组。

综上, 所有基础功能以及附加功能均已实现。

心得体会:

结合实验过程和结果给出实验的体会和收获。

(1) 加深了对流水线机制的可靠数据传输协议了解; 通过编写实验代码更加清晰的理解了GBN和SR的具体区别。

(2) 掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术, 同时学习了解了丢包和超时重传的模拟方法。

(3).通过应用list、dict等数据结构实现缓存功能, 增强了该方面的工程能力。

源代码;

gbn.py:

```
from socket import *

import pkt_format

import time

import threading

import select

import random


TIME_OUT = 3    # 超时时间

SEND_WIN_SIZE = 4    # 滑动窗口大小

max_seq_num = 16    # 最大序列号的数目

CLIENT_IP = "127.0.0.1"

SERVER_IP = '127.0.0.1'

CLIENT_PORT_REC = 10240

SERVER_PORT_SEN = 10241

CLIENT_PORT_SEN = 10242

SERVER_PORT_REC = 10243


class GBNClient:

    def __init__(self):

        self.timer = 0    # 计时器，开始计时后超过 TIME_OUT 则重传

        self.base = 0    #
```



```

self.next_seq = 0

self.expec_seq = 0

self.send_win = SEND_WIN_SIZE

self.send_soc = socket(AF_INET, SOCK_DGRAM)

self.send_soc.bind((CLIENT_IP, CLIENT_PORT_SEN))

self.recv_soc = socket(AF_INET, SOCK_DGRAM)

self.recv_soc.bind((CLIENT_IP, CLIENT_PORT_REC))

self.lock = threading.Lock()

self.data_cache_sent = [] # 缓存发送窗口中未被确认的分组

self.ack_loss_rate = 0.5 # ack 发送丢包率

self.send_loss_rate = 0.1 # 客户端向服务器端发送数据分组时的丢包
概率

def client_send(self, remote_ip =
SERVER_IP, remote_port=SERVER_PORT_REC):
    """
    发送报文给服务器
    :return:
    """

    remote_addr = (remote_ip, remote_port) # 数据将要发送的目标主
机，即服务器

```

```
# 每发送一个数据段

while True:

    time.sleep(random.random()*0.5)

    self.lock.acquire()

    if self.base + self.send_win < max_seq_num:

        if self.next_seq < self.base + self.send_win:

            data = "the seq of this mes is :" + str(self.next_seq)

            new_pkt = pkt_format.data_pkt(self.next_seq, data)

            if random.random() > self.send_loss_rate: # 随机发送丢包

                self.send_soc.sendto(new_pkt, remote_addr)

            print("客户端已发送数据: " + data)

            if self.next_seq == self.base:

                self.timer = 0 # 如果窗口为空 开始计时

                self.next_seq += 1

                self.data_cache_sent.append(new_pkt)

            else:

                print("客户端滑动窗口已满, 暂时无法发送数据")

        else: # 如果 base+滑动窗口大小已经超过最大序号, 则需要根据next_seq 判断是否发报文

            if self.base <= self.next_seq < max_seq_num: # 可发送

                data = "the seq of this mes is :" + str(self.next_seq)
```

丢包

满的判断

max\_seq\_num:

data)

发送丢包

```

new_pkt = pkt_format.data_pkt(self.next_seq, data)

if random.random() > self.send_loss_rate: # 随机发送

    self.send_soc.sendto(new_pkt, remote_addr)

    print("客户端已发送数据: " + data)

    self.data_cache_sent.append(new_pkt)

    if self.next_seq == self.base:

        self.timer = 0

        self.next_seq = (self.next_seq + 1) % max_seq_num
    else: # next_seq 进行过模运算，按如下模式进行窗口是否已

满的判断

        if self.next_seq < (self.base + self.send_win) %

max_seq_num:

            data = "the seq of this mes is :" + str(self.next_seq)

            new_pkt = pkt_format.data_pkt(self.next_seq,

data)

            if random.random() > self.send_loss_rate: # 随机

发送丢包

                self.send_soc.sendto(new_pkt, remote_addr)

                print("客户端已发送数据: " + data)

                self.data_cache_sent.append(new_pkt)

                self.next_seq += 1
    
```

```
        else:

            print("客户端滑动窗口已满，暂时无法发送数据")

            self.lock.release()

        return

def client_recvACK(self):

    """

    另一个线程

    :return:

    """

    while True:

        readable = select.select([self.send_soc], [], [], 1)[0]

        if len(readable) > 0: # 接收 ACK 数据

            ack_byte, remote_addr = self.send_soc.recvfrom(2048)

            ack_seq = int(ack_byte.decode().split()[1])

            self.lock.acquire()

            if (self.next_seq < self.base and (ack_seq >= self.base or
ack_seq < self.next_seq)) \

                or (self.next_seq > ack_seq >= self.base): # 确保

ack 在等待相应序列号范围内

                if ack_seq >= self.base:

                    self.data_cache_sent =
```

```
self.data_cache_sent[ack_seq + 1 - self.base:]

        else:

            self.data_cache_sent =

self.data_cache_sent[ack_seq + 1 + max_seq_num - self.base:]

        self.timer = 0 # 每收到一个 ack, 重启定时器

        self.base = (ack_seq + 1) % max_seq_num

        self.lock.release()

    else:

        self.timer += 1

        if self.timer > TIME_OUT:

            self.timeout_resend(SERVER_IP, SERVER_PORT_REC)

    return

def receive_from_server(self):

    while True:

        readable = select.select([self.recv_soc,],[],[],1)[0]

        if len(readable)> 0:

            rec_pkt,addr = self.recv_soc.recvfrom(1024)

            data = rec_pkt.decode()

            seq = int(data.split())[0]

            if seq == self.expec_seq: # 按序收到了报文段

                if random.random() > self.ack_loss_rate:
```

```
self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq),(SERVER_IP,SERVER_PORT_SEN))

        print("客户端确认收到数据：" + data)

        self.expec_seq = (self.expec_seq + 1) %
max_seq_num

    else: # 发送期待收到的报文序列号

        if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq - 1), (SERVER_IP,
SERVER_PORT_SEN))

        print("客户端重新确认序列号" + str(self.expec_seq - 1))


def timeout_resend(self,remote_ip,remote_port):

    print("客户端响应超时，重新发送：")

    self.timer = 0

    for pkt in self.data_cache_sent:

        self.send_soc.sendto(pkt, (remote_ip, remote_port))

        print("客户端已重新发送：" + pkt.decode())
```

```
class GBNServer:

    """

    实现了超时重传 随机丢包 ack 确认等机制

    """

    def __init__(self):

        self.timer = 0 # 计时器，开始计时后超过 TIME_OUT 则重传

        self.base = 0 #

        self.next_seq = 0

        self.expec_seq = 0

        self.send_win = SEND_WIN_SIZE

        self.send_soc = socket(AF_INET, SOCK_DGRAM)

        self.send_soc.bind((SERVER_IP, SERVER_PORT_SEN))

        self.recv_soc = socket(AF_INET, SOCK_DGRAM)

        self.recv_soc.bind((SERVER_IP, SERVER_PORT_REC))

        self.lock = threading.Lock()

        self.data_cache_sent = []

        self.send_loss_rate = 0.1 # 发送数据包时的丢包概率

        self.ack_loss_rate = 0.1 # 发给客户端的 ack 的丢包率
```

```
def server_send(self, remote_ip = CLIENT_IP,
remote_port=CLIENT_PORT_REC):

    """

    发送报文给有需求的客户端

    :remote_ip:客户端的 ip

    :remote_port:客户端的端口号

    :return: none

    """

    remote_addr = (remote_ip, remote_port) # 数据将要发送的目标主
机，即客户端

    #每发送一个数据段

    while True:

        time.sleep(random.random()*0.5)

        self.lock.acquire()

        if self.base + self.send_win < max_seq_num:

            if self.next_seq < self.base+self.send_win :

                data = "the seq of this mes is :"+str(self.next_seq)

                new_pkt = pkt_format.data_pkt(self.next_seq,data)

                if random.random() > self.send_loss_rate: # 随机发送
丢包

                    self.send_soc.sendto(new_pkt, remote_addr)

                print("服务器已发送: "+ data)
```



```

        if self.next_seq == self.base:

            self.timer = 0 # 如果窗口为空 开始计时

            self.next_seq += 1

            self.data_cache_sent.append(new_pkt)

        else:

            print("窗口已满，暂时无法发送数据")

    else: # 如果 base+滑动窗口大小已经超过最大序号，则需要根据
next_seq 判断是否发报文

        if self.base <= self.next_seq < max_seq_num: # 可发送

            data = "the seq of this mes is :" + str(self.next_seq)

            new_pkt = pkt_format.data_pkt(self.next_seq,data)

            if random.random() > self.send_loss_rate: # 随机发送
丢包

                self.send_soc.sendto(new_pkt, remote_addr)

            print("服务器已发送：" + data)

            self.data_cache_sent.append(new_pkt)

            if self.next_seq == self.base:

                self.timer = 0

                self.next_seq = (self.next_seq + 1) % max_seq_num

            else: # next_seq 进行过模运算，按如下模式进行窗口是否已
满的判断

                if self.next_seq < (self.base + self.send_win) %

```

```

max_seq_num:

        data = "the seq of this mes is :" + str(self.next_seq)

        new_pkt =

pkt_format.data_pkt(self.next_seq,data)

        if random.random() > self.send_loss_rate: # 随机
发送丢包

                self.send_soc.sendto(new_pkt, remote_addr)

                print("服务器已发送: " + data)

                self.data_cache_sent.append(new_pkt)

                self.next_seq += 1

        else:

                print("窗口已满，暂时无法发送数据")

        self.lock.release()

    return

def server_recvACK(self):

    """

    另一个线程

    :return:

    """

    while True:

        readable = select.select([self.send_soc], [], [], 1)[0]

```

```
if len(readable) > 0: # 接收 ACK 数据

    ack_byte, remote_addr = self.send_soc.recvfrom(2048)

    ack_seq = int (ack_byte.decode().split()[1])

    self.lock.acquire()

    if (self.next_seq < self.base and (ack_seq > self.base or
ack_seq < self.next_seq)) \
        or (self.next_seq > ack_seq >= self.base): # 确保 ack
在等待相应序列号范围内

        if ack_seq >= self.base:

            self.data_cache_sent =
self.data_cache_sent[ack_seq + 1 - self.base:]

        else:

            self.data_cache_sent =
self.data_cache_sent[ack_seq + 1 + max_seq_num - self.base:]

            self.base = (ack_seq + 1) % max_seq_num

        self.lock.release()

    else:

        self.timer += 1

        if self.timer > TIME_OUT:

            self.timeout_resend(CLIENT_IP, CLIENT_PORT_REC)

return
```

```
def timeout_resend(self, remote_ip, remote_port):

    print("客户端响应超时，重新发送：")

    self.timer = 0

    for pkt in self.data_cache_sent:

        self.send_soc.sendto(pkt, (remote_ip, remote_port))

        print("服务器已重新发送：" + pkt.decode())

def receive_from_client(self):

    while True:

        readable = select.select([self.recv_soc, ], [], [], 1)[0]

        if len(readable) > 0:

            rec_pkt, addr = self.recv_soc.recvfrom(1024)

            data = rec_pkt.decode()

            seq = int(data.split()[0]) # 分出序列号

            if seq == self.expec_seq: # 按序收到了报文段

                if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq), (CLIENT_IP,
CLIENT_PORT_SEN))

                print("服务器确认收到数据：" + data)

                self.expec_seq = (self.expec_seq + 1) %
```

```
max_seq_num

        else: # 发送期待收到的报文序列号

            if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq-1), (CLIENT_IP,
CLIENT_PORT_SEN))

                print("服务器重新确认序列号" + str(self.expec_seq-1))

#

client = GBNClient()

server = GBNServer()

print("服务器开始向客户端发送数据：")

server_sendat_thread = threading.Thread(target=server.server_send)

server_recvack_thread = threading.Thread(target=server.server_recvACK)

# server 发送数据与接收 ack 同时进行

client_recvdat_thread =

threading.Thread(target=client.receive_from_server)

client_recvdat_thread.start()

server_recvack_thread.start()

server_sendat_thread.start()
```

```
print("客户端开始向服务器发送数据：")

client_sendat_thread = threading.Thread(target=client.client_send)

client_recvack_thread = threading.Thread(target=client.client_recvACK)  #
server 发送数据与接收 ack 同时进行

server_recvdat_thread =
threading.Thread(target=server.receive_from_client)

server_recvdat_thread.start()

client_recvack_thread.start()

client_sendat_thread.start()


time.sleep(60)

client.recv_soc.close()

client.send_soc.close()

server.recv_soc.close()

server.send_soc.close()

exit(0)
```

SR.py:

```
from socket import *

import time

import threading

import pkt_format
```

```
import random

import select

WIN_SIZE = 4

max_seq_num = 16

TIME_OUT = 3    # 超时时间

CLIENT_IP = "127.0.0.1"

SERVER_IP = '127.0.0.1'

CLIENT_PORT = 10247

SERVER_PORT = 10248


class SRServer:

    def __init__(self):

        self.base = 0

        self.send_win = WIN_SIZE    # 滑动窗口大小

        self.next_seq = 0

        self.send_soc = socket(AF_INET, SOCK_DGRAM)

        self.send_soc.bind((SERVER_IP, SERVER_PORT))

        self.lock = threading.Lock()

        self.data_cache_sent = {}    # 缓存发送窗口中未被确认的分组 由序号
```

作为键值索引

```
self.send_loss_rate = 0.1 # 客户端向服务器端发送数据分组时的丢包
```

概率

```
self.timer = {} # 所有时钟
```

```
def server_send(self):
```

```
    """
```

```
    向客户端发送数据
```

```
    :return: none
```

```
    """
```

```
remote_addr = (CLIENT_IP, CLIENT_PORT)
```

```
while True:
```

```
    time.sleep(random.random() * 0.5)
```

```
    self.lock.acquire()
```

```
    if self.base + self.send_win < max_seq_num:
```

```
        if self.next_seq < self.base + self.send_win:
```

```
            data = "the seq of this mes is :" + str(self.next_seq)
```

```
            new_pkt = pkt_format.data_pkt(self.next_seq, data)
```

```
            if random.random() > self.send_loss_rate: # 随机发送
```

丢包

```
                self.send_soc.sendto(new_pkt, remote_addr)
```

```
                self.timer[self.next_seq] = 0 # 为该序号对应的分组启
```



动定时器

```
print("服务器端已发送数据：" + data)
```

```
self.data_cache_sent[self.next_seq] = new_pkt
```

```
self.next_seq += 1
```

```
else:
```

```
print("服务器端滑动窗口已满，暂时无法发送数据")
```

```
else: # 如果 base+滑动窗口大小已经超过最大序号，则需要根据
```

next\_seq 判断是否发报文

```
if self.base <= self.next_seq < max_seq_num: # 可发送
```

```
data = "the seq of this mes is :" + str(self.next_seq)
```

```
new_pkt = pkt_format.data_pkt(self.next_seq, data)
```

```
if random.random() > self.send_loss_rate: # 随机发送
```

丢包

```
self.send_soc.sendto(new_pkt, remote_addr)
```

```
print("服务器端已发送数据：" + data)
```

```
self.data_cache_sent[self.next_seq] = new_pkt
```

```
self.timer[self.next_seq] = 0 # 为该序号对应的分组启
```

动定时器

```
self.next_seq = (self.next_seq + 1) % max_seq_num
```

```
else: # next_seq 进行过模运算，按如下模式进行窗口是否已
```

满的判断

```
if self.next_seq < (self.base + self.send_win) %
```

```
max_seq_num:

        data = "the seq of this mes is :" + str(self.next_seq)

        new_pkt = pkt_format.data_pkt(self.next_seq,

data)

        if random.random() > self.send_loss_rate: # 随机
发送丢包

                self.send_soc.sendto(new_pkt, remote_addr)

                print("服务器端已发送数据：" + data)

                self.timer[self.next_seq] = 0 # 为该序号对应的分
组启动定时器

                self.data_cache_sent[self.next_seq] = new_pkt

                self.next_seq += 1

        else:

                print("服务器端滑动窗口已满，暂时无法发送数据")

                self.lock.release()

    return

def server_recvAck(self):

    """

    从客户端接收 ACK

    :return: none

    """

    while True:
```

```
readable = select.select([self.send_soc], [], [], 1)[0]

if len(readable) > 0: # 接收 ACK 数据

    ack_byte, remote_addr = self.send_soc.recvfrom(2048)

    ack_seq = int(ack_byte.decode().split()[1])

    print(ack_byte.decode())

    self.lock.acquire()

    if (self.next_seq < self.base and (ack_seq > self.base or
ack_seq < self.next_seq)) \

        or (self.next_seq > ack_seq >= self.base): # 确保 ack
在等待相应序列号范围内

        self.data_cache_sent.pop(ack_seq) # 删除发送缓存中
该序号对应的分组

        self.timer.pop(ack_seq) # 停止相应时钟

        if bool(self.data_cache_sent): # 如果还有已发送未确
认的分组

            self.base = list(self.data_cache_sent.keys())[0] #
最先加入的还没有被 ack 的序号

        else:

            self.base = (self.base + 1) % max_seq_num

            self.lock.release()

    else:

        for data in self.timer.keys():
```

```
        self.timer[data] += 1

        if self.timer[data] > TIME_OUT:

            print("客户端响应超时，重新发送")

            self.timer[data] = 0

            self.send_soc.sendto(self.data_cache_sent[data],
(CCLIENT_IP,CLIENT_PORT))

            print("服务器已重新发送：" +
self.data_cache_sent[data].decode())

        return

class SRClient:

    def __init__(self):

        self.recv_base = 0

        self.recv_win = WIN_SIZE # 滑动窗口大小

        self.recv_soc = socket(AF_INET, SOCK_DGRAM)

        self.recv_soc.bind((CLIENT_IP, CLIENT_PORT))

        self.recv_cache = {} # 缓存接收的分组

        self.ack_loss_rate = 0.1 # 客户端向服务器端发送 ack 时的丢包概率

    def client_recv(self):
```

```
remote_addr = (SERVER_IP, SERVER_PORT)

while True:

    readable = select.select([self.recv_soc], [], [], 1)[0]

    if len(readable) > 0:

        rec_pkt, addr = self.recv_soc.recvfrom(1024)

        data = rec_pkt.decode()

        seq = int(data.split())[0]

        self.recv_cache[seq] = rec_pkt

        if seq == self.recv_base:

            if random.random() > self.ack_loss_rate: # 发送 ack
进行确认

                self.recv_soc.sendto(pkt_format.ack_pkt(seq),
remote_addr)

            seq_num = seq # 循环向应用层传输按序的数据分组

            while seq_num in self.recv_cache.keys():

                print("客户端确认收到数据：" +
self.recv_cache[seq_num].decode())

                self.recv_cache.pop(seq_num)

                seq_num = (seq_num + 1) % max_seq_num

                self.recv_base = (seq_num + 1) % max_seq_num

            elif (seq >= self.recv_base and self.recv_base +
self.recv_win <= max_seq_num) \
```

```

        or (self.recv_base + self.recv_win > max_seq_num
and (seq >= self.recv_base or seq < (self.recv_base + self.recv_win) %
max_seq_num)):

        self.recv_cache[seq] = rec_pkt  # 缓存乱序到达的分组

        if random.random() > self.ack_loss_rate:  # 发送 ack
进行确认

            self.recv_soc.sendto(pkt_format.ack_pkt(seq),
remote_addr)

        else:  # 重新接收到已经接受过的分组 直接发送 ack

            if random.random() > self.ack_loss_rate:  # 发送 ack
进行确认

                self.recv_soc.sendto(pkt_format.ack_pkt(seq),
remote_addr)

        return

client = SRClient()

server = SRServer()

print("服务器开始向客户端发送数据：")

server_senddata_thread = threading.Thread(target=server.server_send)

server_recvack_thread = threading.Thread(target=server.server_recvAck)

```

```
# server 发送数据与接收 ack 同时进行

client_recvdat_thread = threading.Thread(target=client.client_recv)

client_recvdat_thread.start()

server_recvack_thread.start()

server_senddata_thread.start()


time.sleep(60)

client.recv_soc.close()

server.send_soc.close()

exit(0)
```

stopAndWait.py:

```
from socket import *

import pkt_format

import time

import threading

import select

import random


"""

停等，双向数据传输，C/S 架构文件传输

"""
```

```
TIME_OUT = 3    # 超时时间

SEND_WIN_SIZE = 1  # 滑动窗口大小

max_seq_num = 16  # 最大序列号的数目

CLIENT_IP = "127.0.0.1"

SERVER_IP = '127.0.0.1'

CLIENT_PORT_REC = 10240

SERVER_PORT_SEN = 10241

CLIENT_PORT_SEN = 10242

SERVER_PORT_REC = 10243


class GBNClient:

    def __init__(self):

        self.timer = 0  # 计时器，开始计时后超过 TIME_OUT 则重传

        self.base = 0  #

        self.next_seq = 0

        self.expec_seq = 0

        self.send_win = SEND_WIN_SIZE

        self.send_soc = socket(AF_INET, SOCK_DGRAM)

        self.send_soc.bind((CLIENT_IP, CLIENT_PORT_SEN))

        self.recv_soc = socket(AF_INET, SOCK_DGRAM)
```



```
self.recv_soc.bind((CLIENT_IP,CLIENT_PORT_REC))

self.lock = threading.Lock()

self.data_cache_sent = [] # 缓存发送窗口中未被确认的分组

self.ack_loss_rate = 0.1 # ack 发送丢包率

self.send_loss_rate = 0.1 # 客户端向服务器端发送数据分组时的丢包
概率

def client_send(self,remote_ip =
SERVER_IP,remote_port=SERVER_PORT_REC):

    """

    发送报文给服务器

    :return:

    """

    remote_addr = (remote_ip, remote_port) # 数据将要发送的目标主
机，即服务器

    while True:

        time.sleep(random.random()*0.5)

        self.lock.acquire()

        if self.base + self.send_win < max_seq_num:

            if self.next_seq < self.base + self.send_win:

                data = "the seq of this mes is :" + str(self.next_seq)

                new_pkt = pkt_format.data_pkt(self.next_seq, data)
```

丢包

```

        if random.random() > self.send_loss_rate: # 随机发送
            self.send_soc.sendto(new_pkt, remote_addr)

            print("客户端已发送数据: " + data)

            if self.next_seq == self.base:

                self.timer = 0 # 如果窗口为空 开始计时

                self.next_seq += 1

                self.data_cache_sent.append(new_pkt)

            # else:

            #     print("客户端滑动窗口已满，暂时无法发送数据")

        else: # 如果 base+滑动窗口大小已经超过最大序号，则需要根据
            next_seq 判断是否发报文

```

丢包

```

            if self.base <= self.next_seq < max_seq_num: # 可发送

                data = "the seq of this mes is :" + str(self.next_seq)

                new_pkt = pkt_format.data_pkt(self.next_seq, data)

                if random.random() > self.send_loss_rate: # 随机发送
                    self.send_soc.sendto(new_pkt, remote_addr)

                    print("客户端已发送数据: " + data)

                    self.data_cache_sent.append(new_pkt)

                if self.next_seq == self.base:

                    self.timer = 0

```

```

        self.next_seq = (self.next_seq + 1) % max_seq_num

    else: # next_seq 进行过模运算，按如下模式进行窗口是否已
满的判断

        if self.next_seq < (self.base + self.send_win) %
max_seq_num:

            data = "the seq of this mes is :" + str(self.next_seq)

            new_pkt = pkt_format.data_pkt(self.next_seq,
data)

            if random.random() > self.send_loss_rate: # 随机
发送丢包

                self.send_soc.sendto(new_pkt, remote_addr)

                print("客户端已发送数据：" + data)

                self.data_cache_sent.append(new_pkt)

                self.next_seq += 1

            # else:

            #     print("客户端滑动窗口已满，暂时无法发送数据")

        self.lock.release()

    return

def client_recvACK(self):

    """

    另一个线程

```

```
:return:

"""

while True:

    readable = select.select([self.send_soc], [], [], 1)[0]

    if len(readable) > 0: # 接收 ACK 数据

        ack_byte, remote_addr = self.send_soc.recvfrom(2048)

        ack_seq = int(ack_byte.decode().split()[1])

        self.lock.acquire()

        if (self.next_seq < self.base and (ack_seq >= self.base or
ack_seq < self.next_seq)) \

            or (self.next_seq > ack_seq >= self.base): # 确保
ack 在等待相应序列号范围内

            if ack_seq >= self.base:

                self.data_cache_sent =
self.data_cache_sent[ack_seq + 1 - self.base:]

            else:

                self.data_cache_sent =
self.data_cache_sent[ack_seq + 1 + max_seq_num - self.base:]

                self.timer = 0 # 每收到一个 ack, 重启定时器

                self.base = (ack_seq + 1) % max_seq_num

            self.lock.release()

        else:
```

```
        self.timer += 1

        if self.timer > TIME_OUT:

            self.timeout_resend(SERVER_IP,SERVER_PORT_REC)

    return

def receive_from_server(self):

    while True:

        readable = select.select([self.recv_soc,[],[],1])[0]

        if len(readable)> 0:

            rec_pkt,addr = self.recv_soc.recvfrom(1024)

            seq = int(rec_pkt.decode().split())[0]

            data = rec_pkt.decode().replace(str(seq)+' ','')

            if seq == self.expec_seq: # 按序收到了报文段

                if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq),(SERVER_IP,SERVER_PORT_SEN))

                print("客户端确认收到数据: " + data)

                file = open('client_file.txt', 'a')

                file.write(data)

                file.close()

                self.expec_seq = (self.expec_seq + 1) %
```

```

max_seq_num

        else: # 发送期待收到的报文序列号

            if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq - 1), (SERVER_IP,
SERVER_PORT_SEN))

                print("客户端重新确认序列号" + str(self.expec_seq - 1))

def timeout_resend(self,remote_ip,remote_port):

    print("客户端响应超时，重新发送：")

    self.timer = 0

    for pkt in self.data_cache_sent:

        self.send_soc.sendto(pkt, (remote_ip, remote_port))

        print("客户端已重新发送：" + pkt.decode())

class GBNServer:

    """

    实现了超时重传 随机丢包 ack 确认等机制
    """

```

```
"""

def __init__(self):

    self.timer = 0 # 计时器，开始计时后超过 TIME_OUT 则重传

    self.base = 0 #

    self.next_seq = 0

    self.expec_seq = 0

    self.send_win = SEND_WIN_SIZE

    self.send_soc = socket(AF_INET, SOCK_DGRAM)

    self.send_soc.bind((SERVER_IP, SERVER_PORT_SEN))

    self.recv_soc = socket(AF_INET, SOCK_DGRAM)

    self.recv_soc.bind((SERVER_IP, SERVER_PORT_REC))

    self.lock = threading.Lock()

    self.data_cache_sent = []

    self.send_loss_rate = 0.1 # 发送数据包时的丢包概率

    self.ack_loss_rate = 0.1 # 发给客户端的 ack 的丢包率


def server_send(self, remote_ip = CLIENT_IP,
remote_port=CLIENT_PORT_REC):

    """

    发送报文给有需求的客户端

    :remote_ip:客户端的 ip
```

```
:remote_port:客户端的端口号

:return: none

"""

remote_addr = (remote_ip, remote_port) # 数据将要发送的目标主
机，即客户端

file = open("server_file.txt")

i = 0

#每发送一个数据段

while True:

    time.sleep(random.random()*0.5)

    self.lock.acquire()

    if self.base + self.send_win < max_seq_num:

        if self.next_seq < self.base+self.send_win :

            data = file.readline()

            if data == '':

                break

            new_pkt = pkt_format.data_pkt(self.next_seq,data)

            if random.random() > self.send_loss_rate: # 随机发送
丢包

                self.send_soc.sendto(new_pkt, remote_addr)

            print("服务器已发送: "+ data)
```



```
        if self.next_seq == self.base:

            self.timer = 0 # 如果窗口为空 开始计时

            self.next_seq += 1

            self.data_cache_sent.append(new_pkt)

        else: # 如果 base+滑动窗口大小已经超过最大序号, 则需要根据
next_seq 判断是否发报文

            if self.base <= self.next_seq < max_seq_num: # 可发送

                data = file.readline()

                if data == "":

                    break

                new_pkt = pkt_format.data_pkt(self.next_seq,data)

                if random.random() > self.send_loss_rate: # 随机发送
丢包

                    self.send_soc.sendto(new_pkt, remote_addr)

                print("服务器已发送: " + data)

                self.data_cache_sent.append(new_pkt)

            if self.next_seq == self.base:

                self.timer = 0

                self.next_seq = (self.next_seq + 1) % max_seq_num

            else: # next_seq 进行过模运算, 按如下模式进行窗口是否已
满的判断

                if self.next_seq < (self.base + self.send_win) %
```

```
max_seq_num:

        data = file.readline()

        if data == "":

            break

        new_pkt =

pkt_format.data_pkt(self.next_seq,data)

        if random.random() > self.send_loss_rate: # 随机
发送丢包

            self.send_soc.sendto(new_pkt, remote_addr)

            print("服务器已发送: " + data)

            self.data_cache_sent.append(new_pkt)

            self.next_seq += 1

        self.lock.release()

    return

def server_recvACK(self):

    """

    另一个线程

    :return:

    """

    while True:

        readable = select.select([self.send_soc], [], [], 1)[0]
```

```
if len(readable) > 0: # 接收 ACK 数据

    ack_byte, remote_addr = self.send_soc.recvfrom(2048)

    ack_seq = int (ack_byte.decode().split()[1])

    self.lock.acquire()

    if (self.next_seq < self.base and (ack_seq > self.base or
ack_seq < self.next_seq)) \
        or (self.next_seq > ack_seq >= self.base): # 确保 ack
在等待相应序列号范围内

        if ack_seq >= self.base:

            self.data_cache_sent =
self.data_cache_sent[ack_seq + 1 - self.base:]

        else:

            self.data_cache_sent =
self.data_cache_sent[ack_seq + 1 + max_seq_num - self.base:]

            self.base = (ack_seq + 1) % max_seq_num

        self.lock.release()

    else:

        self.timer += 1

        if self.timer > TIME_OUT:

            self.timeout_resend(CLIENT_IP, CLIENT_PORT_REC)

return
```

```
def timeout_resend(self, remote_ip, remote_port):

    print("客户端响应超时，重新发送：")

    self.timer = 0

    for pkt in self.data_cache_sent:

        self.send_soc.sendto(pkt, (remote_ip, remote_port))

        print("服务器已重新发送：" + pkt.decode())

def receive_from_client(self):

    while True:

        readable = select.select([self.recv_soc, ], [], [], 1)[0]

        if len(readable) > 0:

            rec_pkt, addr = self.recv_soc.recvfrom(1024)

            data = rec_pkt.decode()

            seq = int(data.split())[0]

            if seq == self.expec_seq: # 按序收到了报文段

                if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq), (CLIENT_IP,
CLIENT_PORT_SEN))

                print("服务器确认收到数据：" + data)

                self.expec_seq = (self.expec_seq + 1) %
```

```
max_seq_num

        else: # 发送期待收到的报文序列号

            if random.random() > self.ack_loss_rate:

self.recv_soc.sendto(pkt_format.ack_pkt(self.expec_seq-1), (CLIENT_IP,
CLIENT_PORT_SEN))

                print("服务器重新确认序列号" + str(self.expec_seq-1))

#

client = GBNClient()

server = GBNServer()

print("服务器开始向客户端发送数据：")

server_sendat_thread = threading.Thread(target=server.server_send)

server_recvack_thread = threading.Thread(target=server.server_recvACK)

# server 发送数据与接收 ack 同时进行

client_recvdat_thread =

threading.Thread(target=client.receive_from_server)

client_recvdat_thread.start()

server_recvack_thread.start()

server_sendat_thread.start()
```