哈尔滨工业大学

<<计算机网络>>

实验报告

(2017 年度春季学期)

姓名:	张茗帅
学号:	**1140310606**
学院:	计算机科学与技术学院
教师:	聂兰顺

实验二 可靠数据传输协议-GBN 协议的设计与实现

一、实验目的

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。

实验环境：

我的 PC 电脑
Windows 10, eclipse
开发语言：python

二、实验内容

1. 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）。
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 GBN 协议，支持双向数据传输；
4. 将所设计的 GBN 协议改进为 SR 协议。

三、实验过程及结果

实验原理说明：

1 GBN 协议数据分组格式、确认分组格式、各个域作用

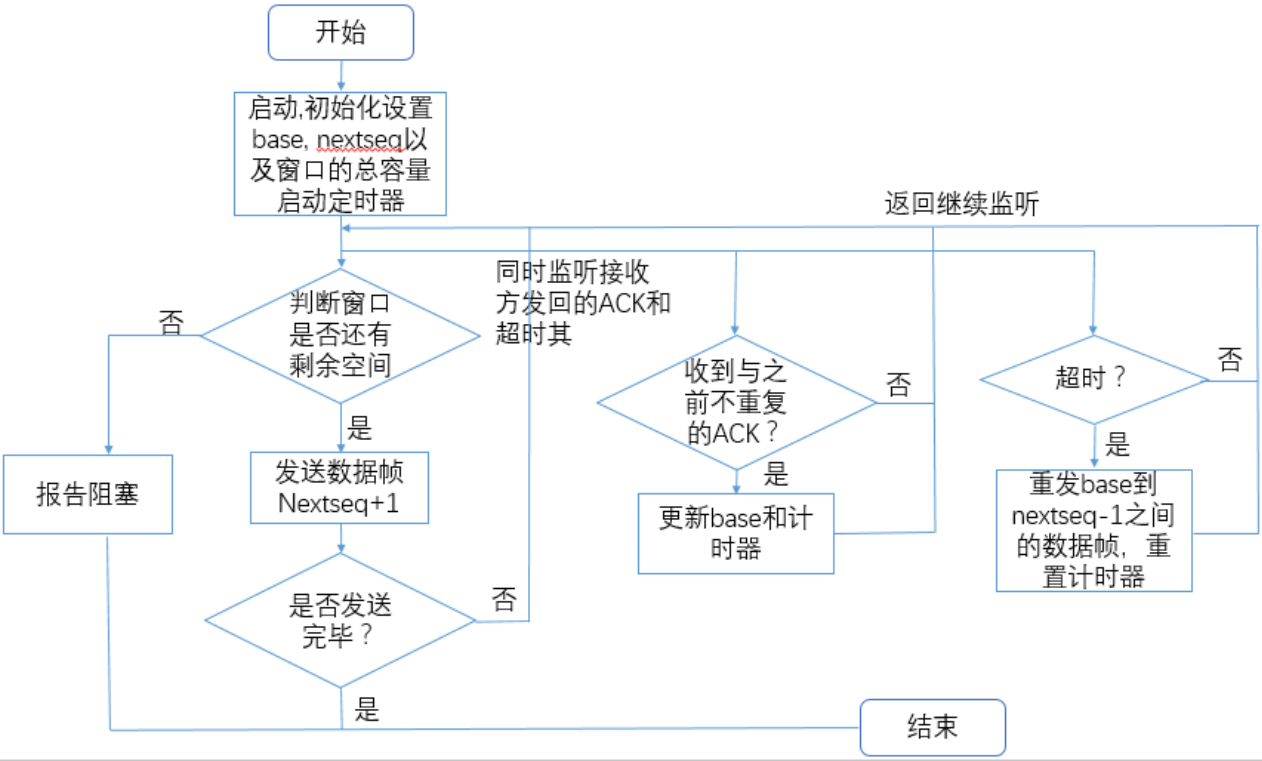
在以太网中，数据帧的 MTU 为 1500 字节，所以 UDP 数据报的数据部分应小于 1472 字节（除去 IP 头部 20 字节与 UDP 头的 8 字节），为此，定义 UDP 数据报的数据部分格式为：

Seq	Data	0

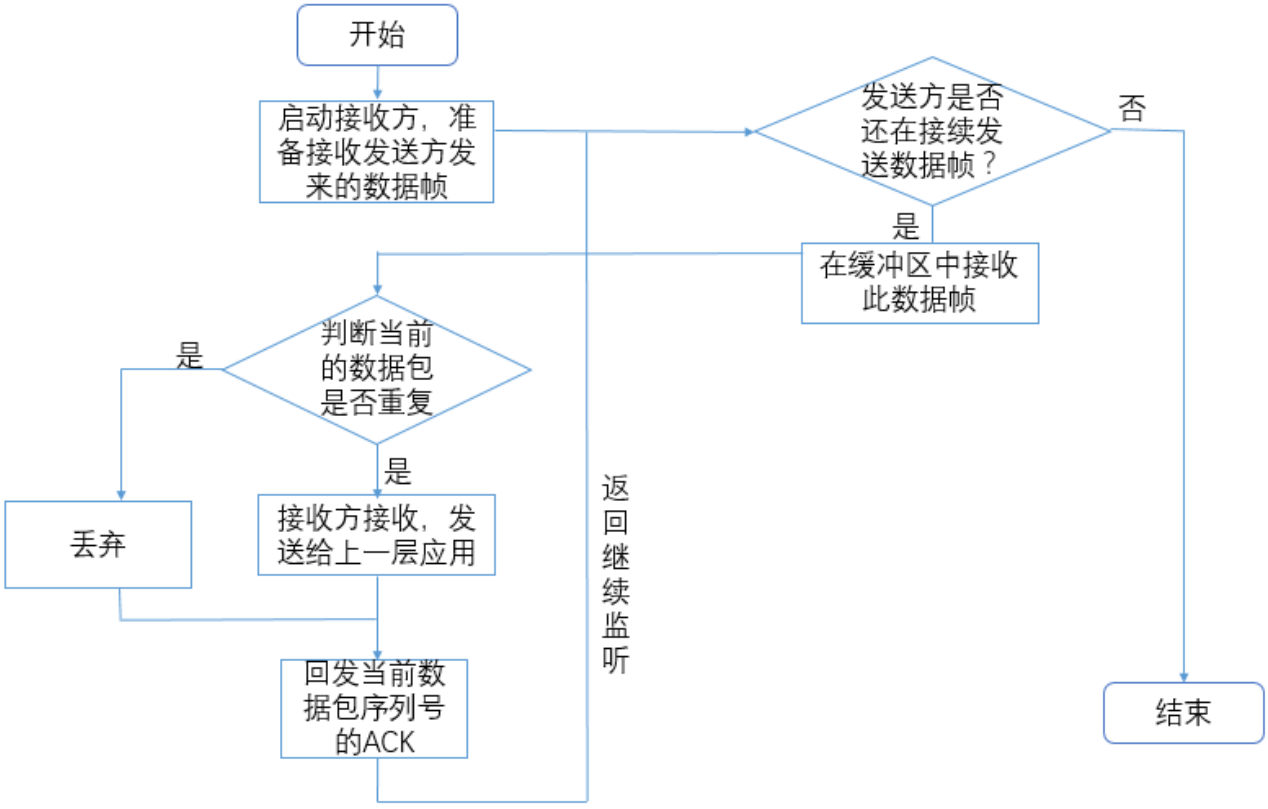
Seq 为 1 个字节，取值为 0~255（故序列号最多为 256 个）；
Data≤1024 个字节，为传输的数据；
最后一个字节放入 EOF0，表示结尾。

2 协议两段程序流程图

客户端



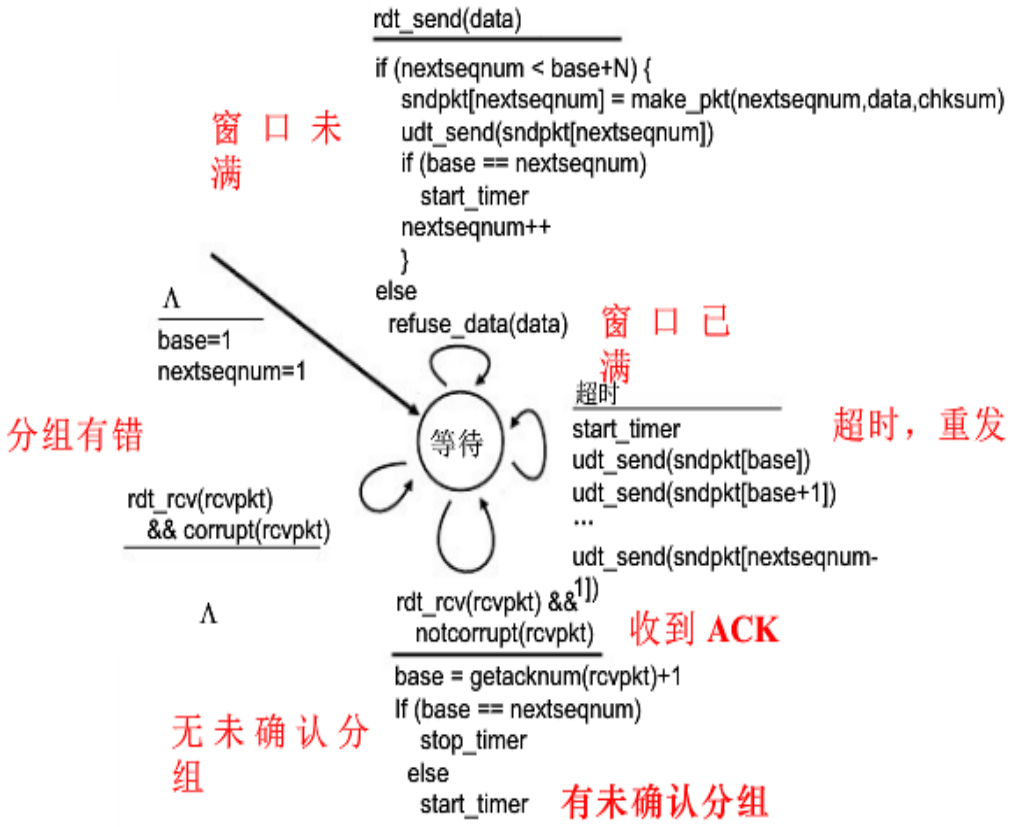
服务器端



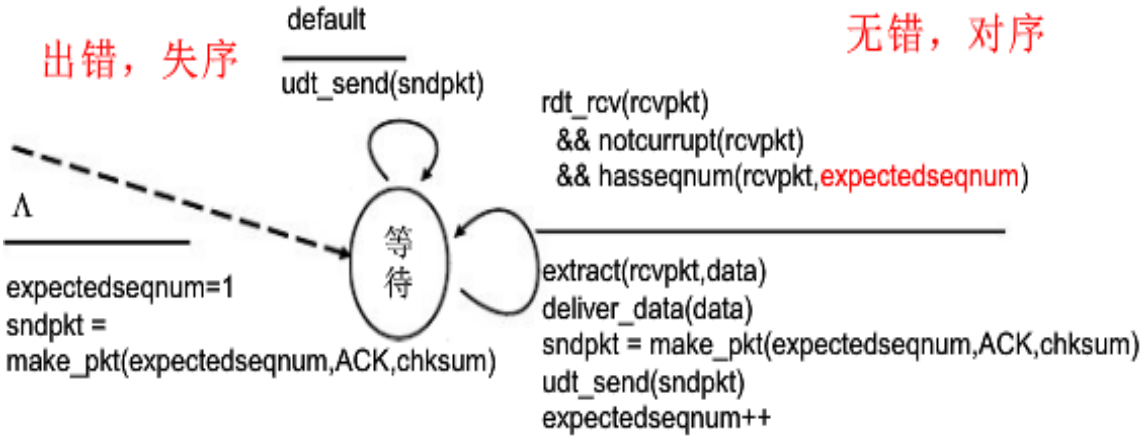
3 协议典型交互过程

先看一下客户端和服务器端的自动机

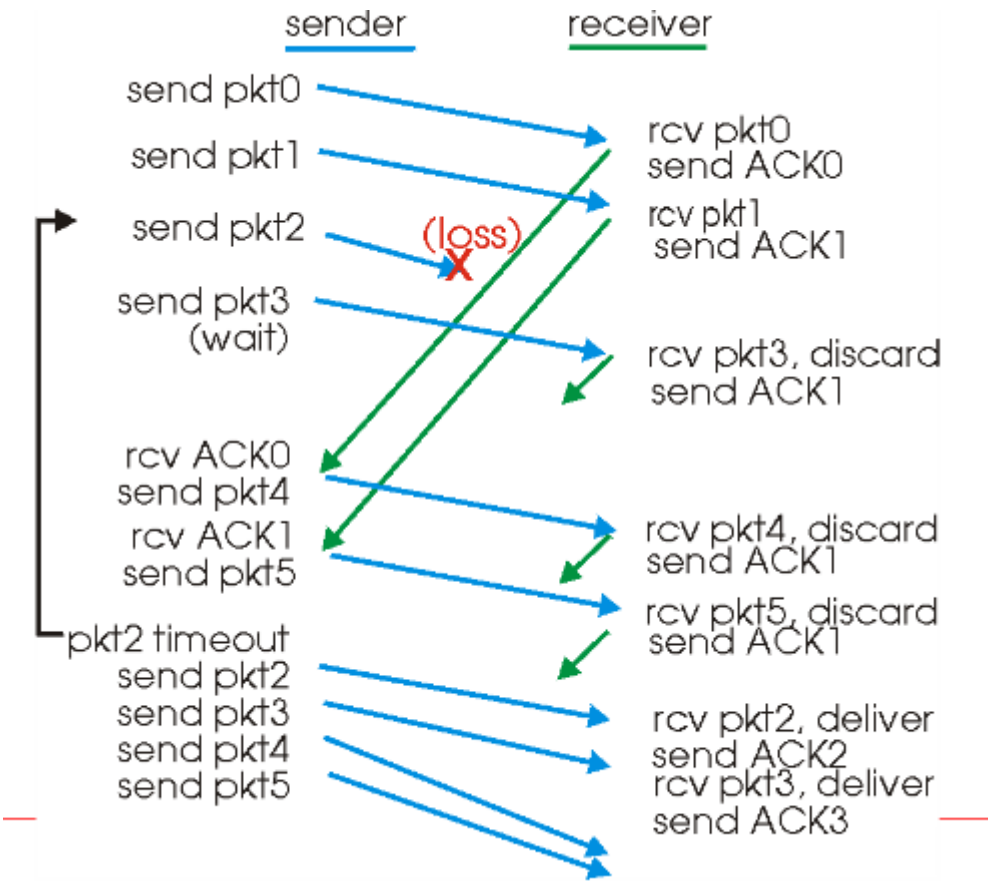
客户端（发送方）：



服务器端（接收方）：



然后我们来通过一个实例看一下两者交换的具体过程



4 数据分组丢失验证模拟方法

客户端每次在发送数据祯的时候，程序会生成一个 0-1 范围之间的随机数，并将此随机数与 0.2 进行比较，如果小于 0.2 的话，则发送端直接不发送此数据帧，通过这种手段来模拟数据帧的丢失情况。接收方即只能回复已经收到之前分组的累计 ACK，等到客户端的定时器超时之后，客户端再把自己收到 ACK 之后的段开始一一重新发送给接收方。

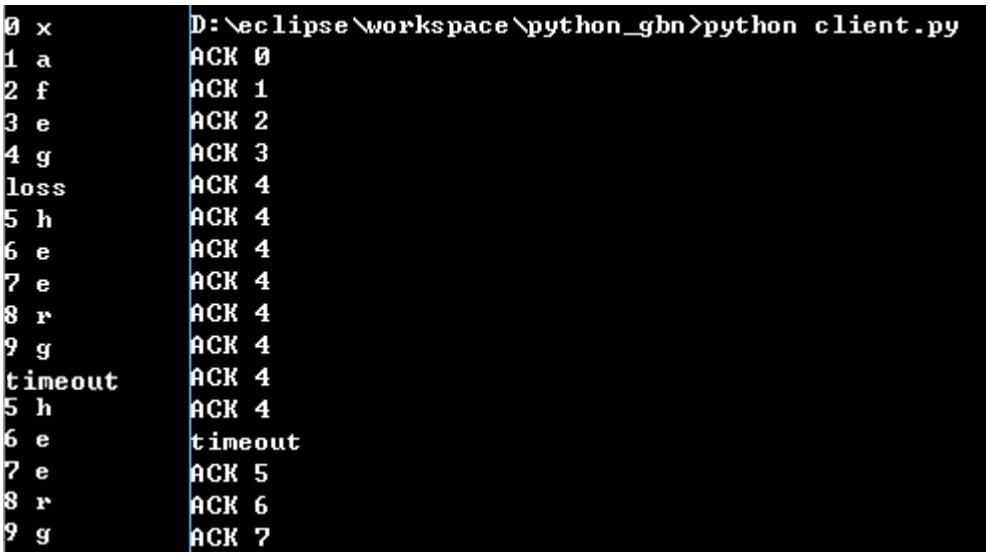
实验测试：

我的实验实现了 GBN 基本协议的功能，以及改进的 GBN 协议，也就是可以支持数据的双向传输，同时也做了 SR 协议的内容

同时打开两个控制台，测试截图如下

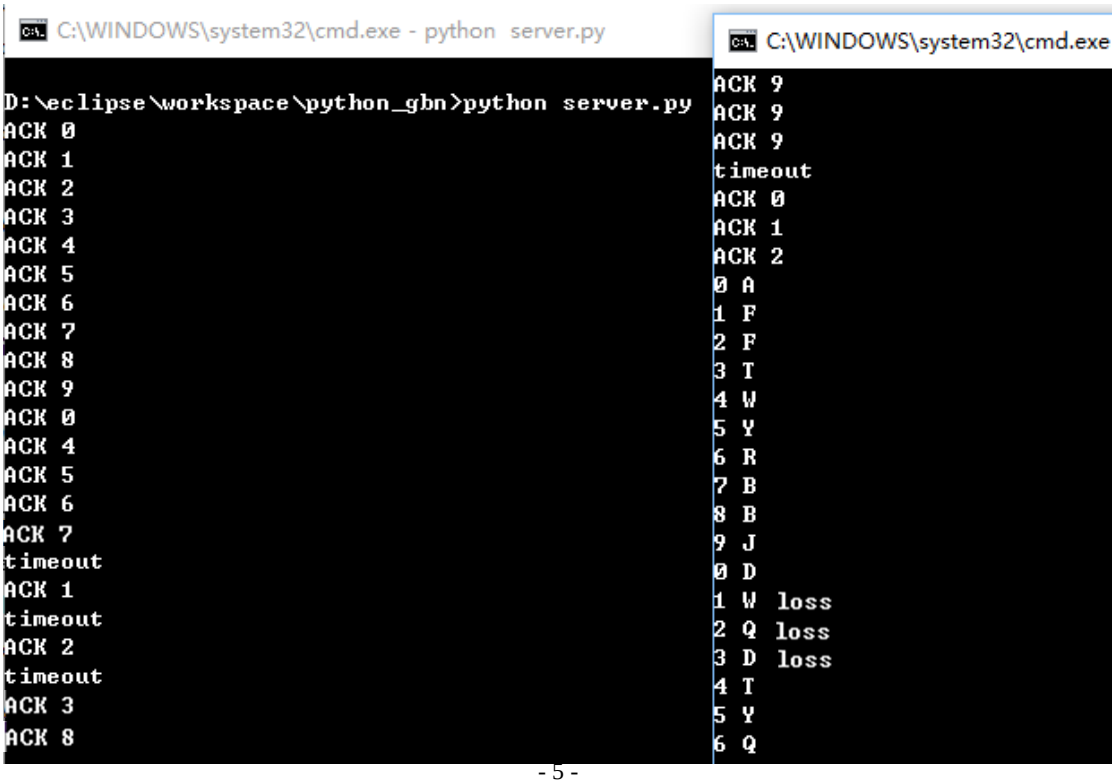
首先运行 server.py，再运行 client.py，此时测试的是 GBN 协议，client.py 作为接收方，server.py 作为发送方，具体结果如下图：

可以看到，接收方采用的是累计确认的方式，可以看到，当第五个数据帧丢失之后，尽管发送方还在发送 5、6、7、8、9 号祯，但是却无法收到这些祯的 ACK，故在定时器超时之后，发送方重新发送 ACK4 之后的数据帧，也就是从第五号祯开始重新发。



然后我们这个时候不用动 client.py，关闭 server.py，然后再次重新启动 server.py，此时 client.py 便作为发送方，而 server.py 作为接受方，可以在下图的右部分看到我的 client.py 没有关闭，之前还是接收方呼呼发 ACK，一转眼便转变为了发送方，因为实现了数据的双向传输,与此同时,本次执行的是 SR 协议,具体实现如下图:

可以看到 SR 协议采用的是每个分组都有一个计时器，每个分组单独确认机制，与此同时接收方还有缓存机制.当 1、2、3 分组丢失时，接收方仍然能收到 4、5、6、7 数据包并缓存，同时发送 ACK4-8 给发送方，当 1、2、3 分组的计时器超时后，发送方只需重发这三个分组即可，并不需要发送之后的了，因为接收方已经收到并缓存了。



附录代码（带有详尽注释）：

Util.py 用于实现 GBN 协议和 SR 协议的具体控制

```
# -*- coding:utf-8 -*-
import sys
import select
from random import random

# 设置在 localhost 进行测试
HOST = '127.0.0.1'

# 设置服务器端与客户端的端口号
SERVER_PORT = 5001
CLIENT_PORT = 5002

# 另开端口组实现双向通信
SERVER_PORT_EXTRA = 5003
CLIENT_PORT_EXTRA = 5004

# 单次读取的最大字节数
BUFFER_SIZE = 2048

# 窗口与包序号长度
WINDOWS_LENGTH = 8
SEQ_LENGTH = 10

# 最大延迟时间
MAX_TIME = 3


class Data(object):

    def __init__(self, msg, seq=0, state=0):
        self.msg = msg
        self.state = state
        self.seq = str(seq % SEQ_LENGTH)

    def __str__(self):
        return self.seq + ' ' + self.msg


class Gbn(object):
```

```
def __init__(self, s):
    self.s = s

def push_data(self, path, port):

    # 计时和包序号初始化
    time = 0
    seq = 0

    data_windows = []

    with open(path, 'r') as f:

        while True:

            # 当超时后，将窗口内的数据更改为未发送状态
            if time > MAX_TIME:
                print "timeout"
                for data in data_windows:
                    data.state = 0

            # 窗口中数据少于最大容量时，尝试添加新数据
            while len(data_windows) < WINDOWS_LENGTH:
                line = f.readline().strip()

                if not line:
                    break

                data = Data(line, seq=seq)
                data_windows.append(data)
                seq += 1

            # 窗口内无数据则退出总循环
            if not data_windows:
                break

            # 遍历窗口内数据，如果存在未成功发送的则发送
            for data in data_windows:
                if not data.state:
                    self.s.sendto(str(data), (HOST, port))
                    data.state = 1

            # 无阻塞 socket 连接监控
            readable, writeable, errors = select.select([self.s, ], [], [], 1)
```

```
                if len(readable) > 0:

                    # 收到数据则重新计时
                    time = 0

                    message, address = self.s.recvfrom(BUFFER_SIZE)
                    sys.stdout.write('ACK ' + message + '\n')

                    for i in range(len(data_windows)):
                        if message == data_windows[i].seq:
                            data_windows = data_windows[i+1:]
                            break
                else:
                    # 未收到数据则计时器加一
                    time += 1

        self.s.close()

    def pull_data(self):

        # 记录上一个回执的 ack 的值
        last_ack = SEQ_LENGTH - 1

        data_windows = []

        while True:

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

            if len(readable) > 0:
                message, address = self.s.recvfrom(BUFFER_SIZE)

                ack = int(message.split()[0])

                # 连续接收数据则反馈当前 ack
                if last_ack == (ack - 1) % SEQ_LENGTH:

                    # 丢包率为 0.2
                    if random() < 0.2:
                        print "loss"
                        continue

                    self.s.sendto(str(ack), address)
```

```
                    last_ack = ack

                    # 判断数据是否重复
                    if ack not in data_windows:
                        data_windows.append(ack)
                        sys.stdout.write(message + '\n')

                    while len(data_windows) > WINDOWS_LENGTH:
                        data_windows.pop(0)
                else:
                    self.s.sendto(str(last_ack), address)

        self.s.close()


class Sr(object):

    def __init__(self, s):
        self.s = s

    def push_data(self, path, port):

        # 计时和包序号初始化
        time = 0
        seq = 0

        data_windows = []

        with open(path, 'r') as f:

            while True:

                # 当超时后，将窗口内第一个发送成功未确认的数据状态更改为未发送
                if time > MAX_TIME:
                    print "timeout"
                    for data in data_windows:
                        if data.state == 1:
                            data.state = 0
                            break

                # 窗口中数据少于最大容量时，尝试添加新数据
                while len(data_windows) < WINDOWS_LENGTH:
                    line = f.readline().strip()
```

```
            if not line:
                break

            data = Data(line, seq=seq)
            data_windows.append(data)
            seq += 1

        # 窗口内无数据则退出总循环
        if not data_windows:
            break

        # 遍历窗口内数据，如果存在未成功发送的则发送
        for data in data_windows:
            if not data.state:
                self.s.sendto(str(data), (HOST, port))
                data.state = 1

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

        if len(readable) > 0:

            # 收到数据则重新计时
            time = 0

            message, address = self.s.recvfrom(BUFFER_SIZE)
            sys.stdout.write('ACK ' + message + '\n')

            # 收到数据后更改该数据包状态为已接收
            for data in data_windows:
                if message == data.seq:
                    data.state = 2
                    break
        else:
            # 未收到数据则计时器加一
            time += 1

        # 当窗口中首个数据已接收时，窗口前移
        while data_windows[0].state == 2:
            data_windows.pop(0)

            if not data_windows:
                break

self.s.close()
```

```
    def pull_data(self):

        # 窗口的初始序号
        seq = 0
        data_windows = {}

        while True:

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

            if len(readable) > 0:
                message, address = self.s.recvfrom(BUFFER_SIZE)

                ack = message.split()[0]

                # 丢包率为 0.2
                if random() < 0.2:
                   #    print "loss"
                    continue

                # 返回成功接收的包序号
                self.s.sendto(ack, address)
                data_windows[ack] = message.split()[1]

                # 滑动窗口
                while str(seq) in data_windows:
                    sys.stdout.write(str(seq) + ' ' + data_windows[str(seq)] + '\n')
                    data_windows.pop(str(seq))
                    seq = (seq + 1) % SEQ_LENGTH

        self.s.close()
```

server.py 用于启动 GBN 发送方（当然也是 SR 的接收方）

```
# -*- coding:utf-8 -*-
import socket
import thread
import client
from util import *


def new_server_socket(server_port, client_port, path, protocol):
    # 设置网络连接为 ipv4， 传输层协议为 udp
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # 传输完成后立即回收该端口
```

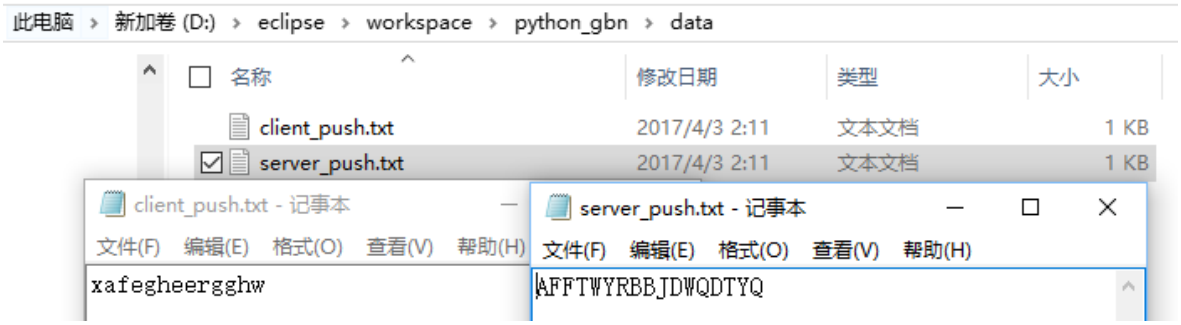
```
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # 任意 ip 均可以访问
    s.bind(('', server_port))

    p = protocol(s)
    p.push_data(path, client_port)


if __name__ == '__main__':

    thread.start_new_thread(new_server_socket, (SERVER_PORT, CLIENT_PORT,
'data/server_push.txt', Sr))

    # new_server_socket(SERVER_PORT, CLIENT_PORT, 'data/server_push.txt', Sr)
    client.new_client_socket(CLIENT_PORT_EXTRA, Gbn)
```

client.py 用于启动 GBN 接收方（当然也是 SR 的发送方）

```
# -*- coding:utf-8 -*-
import socket
import thread
import server

from util import *


def new_client_socket(client_port, protocol):
    # 设置网络连接为 ipv4， 传输层协议为 tcp
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # 传输完成后立即回收该端口
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # 任意 ip 均可以访问
    s.bind(('', client_port))

    p = protocol(s)
    p.pull_data()

if __name__ == '__main__':

    # 接收方开启多线程
    thread.start_new_thread(server.new_server_socket, (SERVER_PORT_EXTRA,
CLIENT_PORT_EXTRA, 'data/client_push.txt', Gbn))

    # server.new_server_socket(SERVER_PORT_EXTRA, CLIENT_PORT_EXTRA,
'data/client_push.txt', Gbn)
    new_client_socket(CLIENT_PORT, Sr)
```

测试数据：

小写字母用于 GBN 测试，大写字母用于 SR 测试



四、实验心得

通过本次实验，我对可靠数据传输的协议有了更加深刻的认识，对 GBN 协议和 SR 协议的具体实现流程以及控制机制（重发，确认，超时，序列号，缓存等等）掌握的十分深入，实验总是有很多的好处，不仅可以提高我们的代码能力，锻炼我们的实践，同时能够加深我们对相关知识的认识与理解，让我们去更加仔细认真地进行思考，因为如果对本质的知识没有理解到位，那么实验是不可能获得成功的。我很感谢计算机网络实验的设计，这些实验让我对每一部分的内容理解的更加深刻。有些时候，知识是需要反复刺激的，自己 MOOC 看一遍，老师带着复习一遍，我们因此会对这些知识感觉自己已经理解了，但这些理解仅仅是停留在理论层面上的，具体的细节很有可能会被忽略，只有我们亲自动手去做，去设计，我们才会透彻地理解相关的知识点。