哈爾濱工業大學
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-GBN 协议的设计与实现				
姓名	朱宸慷		院系	计算机科学与技术	
班级	2103103		学号	2021110908	
任课教师	聂兰顺		指导教师	聂兰顺	
实验地点	格物 207		实验时间	10.28	
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					

哈尔滨工业大学计算学部
FACULTY OF COMPUTING, HIT

实验目的：

本次实验的主要目的：

（1）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。

（2）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：

概述本次实验的主要内容，包含的实验项等。

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：

一、数据分组格式、确认分组格式、各个域作用

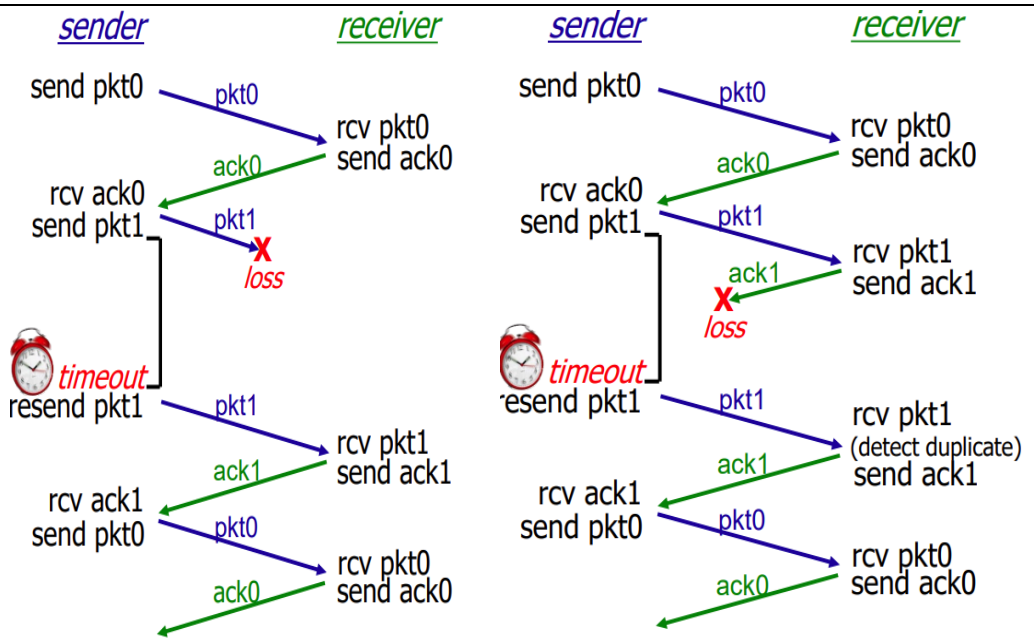
所有的数据报都来自基类Datagram，在Datagram中规定数据报文内容形如下图

其中Type需要在ACK, SEQ, CNT, FIN中选取一个：ACK是客户端向服务端的确认报文；SEQ是服务端向客户端发送的数据报文；CNT是一个连接确认报文，这个类型的存在是因为Python中建立两个套接字之间的连接需要先握手，因此设置这样一种报文；FIN类型与SEQ相同，只是为了标志着数据传输已经结束，此时服务端要等待确认所有ACK都收到。

二、停等协议的实现思路与典型的交互过程

基于上述报文结构，我们先来解释停等协议的实现思路。首先建立连接后，假设我们的客户端此时有一个文件下载请求，服务端就会开始向客户端传输文件。服务端分片文件后，会将第一片发送给客户端并启动计时器，此时服务端不断等待，直到客户端将这个数据报的确认信息返回给服务端或者等待超时，此时服务端就重新传输；与此同时，客户端等待服务器传来的数据报，一旦接收到SEQ类型的数据报，就解析后面的SeqNum，判断是否是它所想要的（即按序的报文），若是，则再检查报文的CheckSum段是否有效，有效后就返回一个对应的ACK报文。在本实验中，为了简化实现，没有设置CheckSum字段，即默认正确，当然，其实若是有该字段并且验证得知报文无效，客户端只需要简单丢弃即可，服务端等到超时会重传该报文。在服务端正确接收到确认报文后，它会开始下一轮的传输，直到所有报文传输完毕。

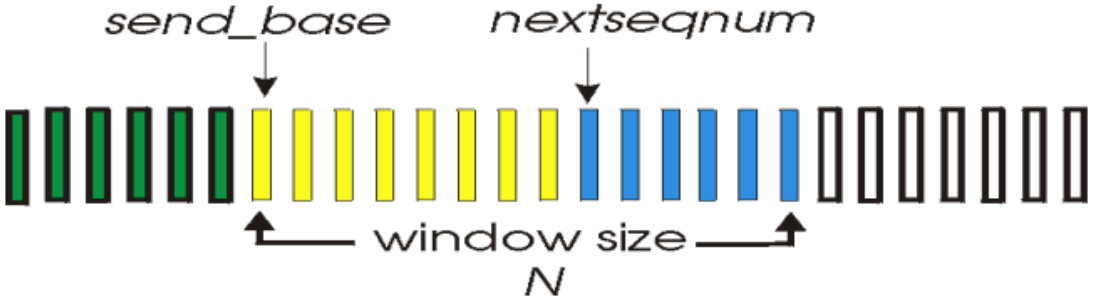
停等协议的一个典型的交互过程图如下图所示，分别代表数据丢失和ACK丢失的重传，它们其实都是由计时器控制的。



鉴于我们可以将停等协议认为是发送窗口为1的GBN协议，对于停等协议的流程图、双向传输和文件传输功能的实现等将在GBN协议中详细介绍。

三、GBN协议的实现思路与典型的交互过程

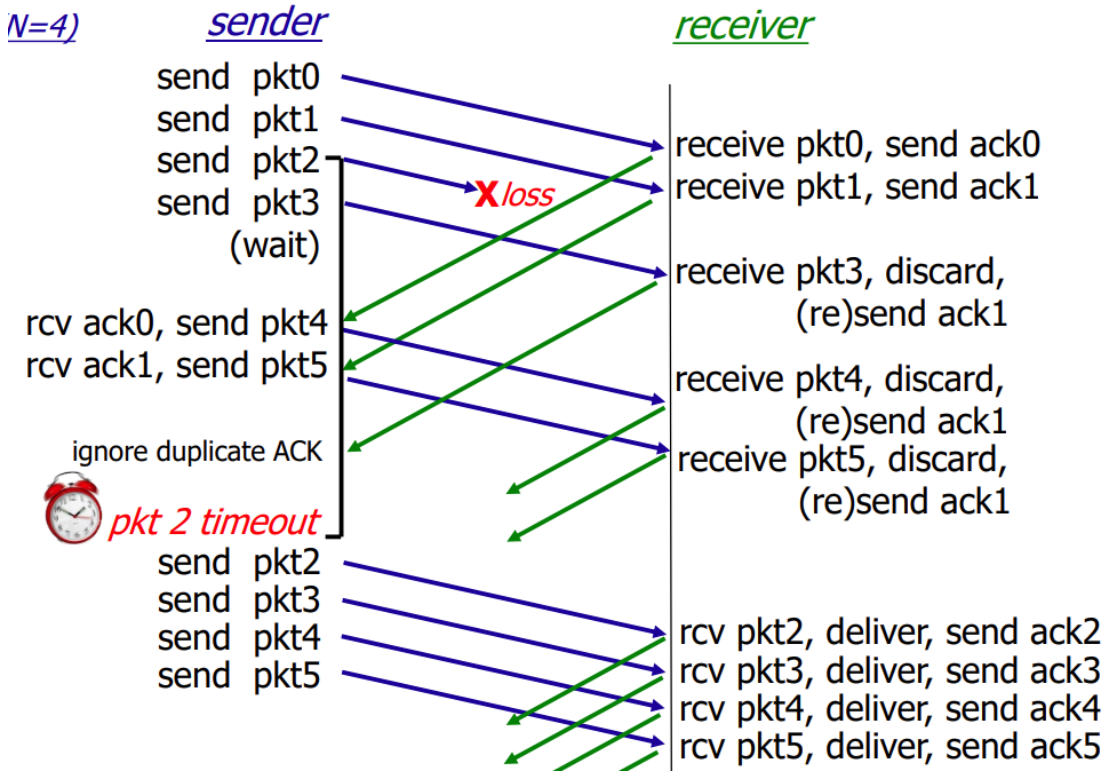
GBN协议与简单的停等协议相比，其实就是将发送方的可发送数从1变成了一个由发送窗口的大小所决定的N，当然，与SR相比，此时接收窗口仍然是1，也就是说，GBN协议中，我们的客户端仍然只支持按序的接收报文。GBN服务器维护一个如下图所示的窗口：



当我们初始化服务端时，服务端会将send_base与nextseqnum置为0。在理想情况下，服务器开始发送数据时，会一直发送到窗口满为止，此时nextseqnum＝base＋N，N即窗口长度。当满足上述等式时，服务端会停止发送并开始等待客户端返回ACK报文。此时，客户端开始按序接收报文并返回ACK，当遇到乱序报文时，客户端直接丢弃。在GBN协议中，我们要求双方都要作出一定的保证，例如客户端保证只按序接受，因此服务端就可以相信ACK信号一定有效，于是可以做出累计确认ACK的策略，而不需要按序接受ACK，也就是说服务端每次收到窗口内的ACK，如果序号大于当前send_base，就将send_base移至序号加一，即认为客户端正确接收了之前的数据报。

对于GBN协议中的数据报丢失现象，由于服务端要确保客户端可以收到此时窗口内的全部数据报，因此，每当计时器触发超时，服务端都有义务重传它窗口内从send_base到nextseqnum的所有报文，也就是它发送过但不确定客户端是否收到的报文，然后重新等待客户端传来ACK。当客户端在接受报文时，它保证只按序接收并且返回ACK，当遇到超前或者落后的报文，都直接丢弃，并且返回它最后一次返回的ACK。GBN协议的一个典型的交互过

程如下图所示：



从图中我们可以看到，由于pkt2丢失，因此客户端只希望获得序号为2的报文，因此只返回ACK1，发送方据此移动send_base并发送4、5。当超时发生后，发送方将重发2到5的所有报文。代码实现如下：

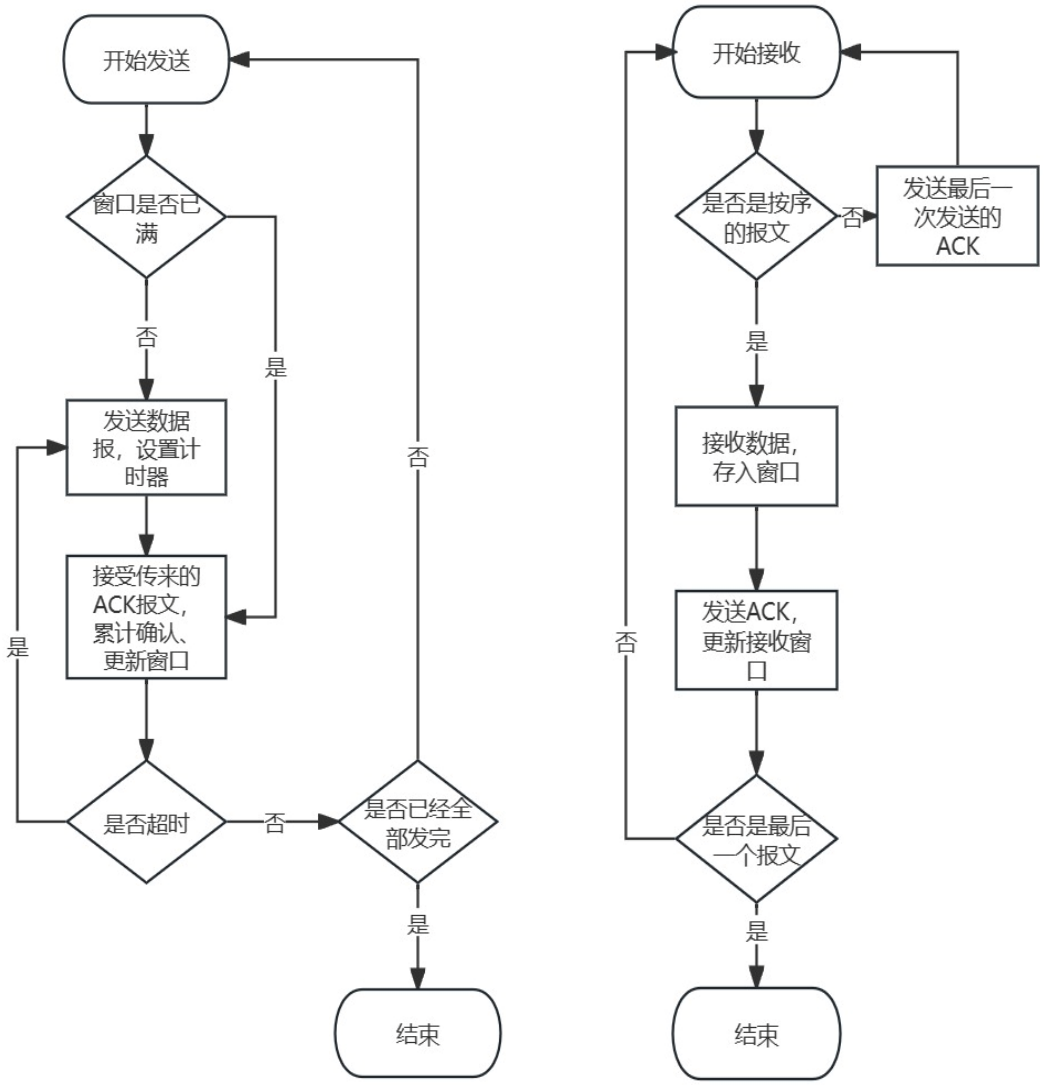
```
# 发送窗口
while self.base < len(self.frames):
    self.timers = threading.Timer(5, self.GBNSendALL)
    self.timers.start()
    while self.next_seq < self.base + self.windowSize and self.next_seq <
len(self.frames):
        self.sendFrame(self.frames[self.next_seq])
        self.next_seq += 1
    while True:
        rcvDatagram, self.ClientAddr = self.ServerSocket.recvfrom(self.BUFSIZ)
        if self.randomDrop():
            print("丢弃 ACK", rcvDatagram.decode("UTF-8")[4:])
            continue
        if rcvDatagram.decode("UTF-8")[0:3] == "ACK":
            self.acks[int(rcvDatagram.decode("UTF-8")[4:])] = True
            print("确认收到 ACK" + rcvDatagram.decode("UTF-8")[4:])
            # 累计确认

            self.base = int(rcvDatagram.decode("UTF-8")[4:]) + 1
            if self.base == self.next_seq:
                self.timers.cancel()
```

```
            print("传输完成")
            return
        else:
            self.timers.cancel()
            self.timers = threading.Timer(5, self.GBNSendALL)
            self.timers.start()
            break

# 接受服务器的数据
while True:

    rcvDatagram, ServerAddr = ClientSocket.recvfrom(BUFSIZ)
    if client.randomDrop():
        print("丢弃序号为", rcvDatagram.decode("UTF-8")[
                        rcvDatagram.decode("UTF-8").find(" "):rcvDatagram.decode("UTF-
8").find("\n")],
              "的数据包")
        continue
    if rcvDatagram.decode("UTF-8")[0:3] == "SEQ":
        # print(rcvDatagram.decode("UTF-8"))
        # 提取序号，序号是在第一个空格之后，第一个换行符之前
        seq = int(rcvDatagram.decode("UTF-8")[
                rcvDatagram.decode("UTF-8").find(" "):rcvDatagram.decode("UTF-
8").find("\n")])
        # 提取数据，数据是在第一个换行符之后
        data = rcvDatagram.decode("UTF-8")[rcvDatagram.decode("UTF-8").find("\n") + 1:]

        # 检查序号是否正确
        if seq != len(fileBuffer) and seq != 0:
            print("序号错误，期望序号为", len(fileBuffer), "实际序号为", seq)
            continue
        # 将数据写入 fileBuffer
        print("接收序号为", seq, "的数据包")
        fileBuffer.append(data)
        # 发送 ACK
        ack = DataGram("ACK", seq, "")
        ClientSocket.sendto(ack.makePacket, ServerAddr)
        # print(ack.makePacket.decode("UTF-8"))

    if rcvDatagram.decode("UTF-8")[0:3] == "FIN":
        # print(rcvDatagram.decode("UTF-8"))
        # 发送 ACK
        seq = int(rcvDatagram.decode("UTF-8")[
                rcvDatagram.decode("UTF-8").find(" "):rcvDatagram.decode("UTF-
```

```
8").find("\n")])
        ack = DataGram("ACK", seq, "")
        print("接收序号为", seq, "的数据包")
        ClientSocket.sendto(ack.makePacket, ServerAddr)
        fileBuffer.append(rcvDatagram.decode("UTF-8")[rcvDatagram.decode("UTF-
8").find("\n") + 1:])
        writeBufferToFile(fileBuffer, self.outFilePath)
        break
```

四、GBN协议的流程图

GBN协议的发送方和接收方流程图如下图所示：



五、GBN协议的双向传输

当需要将C/S模式的GBN协议扩展到全双工时，只需要为双方分别各自维护一个接收窗口和发送窗口，由于报文可以以报文头部的类型区分，因此当连接正常建立后，双方只需要各自针对报文的类型，独立完成发送方和接收方的任务即可。

六、GBN协议的文件传输

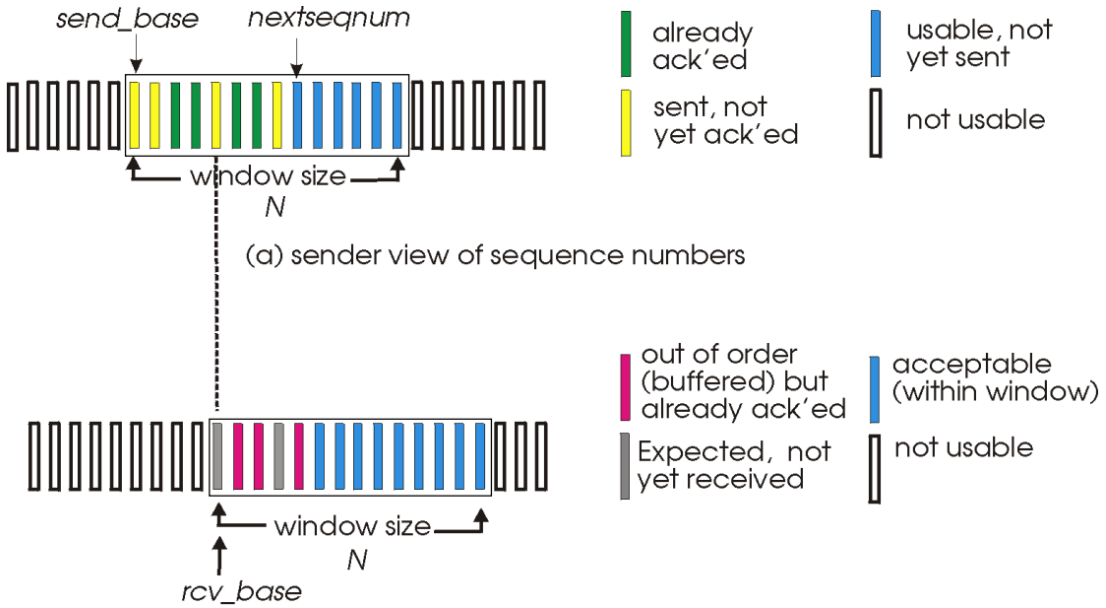
在实现文件传输中，只需要按照一定的规律对于源文件进行分割即可。在服务端实现代码如下：

```
# 读取文件
with open(self.inFilePath, "r") as f:
    data = f.read()
    # 将文件内容按照 BUFSIZ 分割，分别编号发给客户端
    for i in range(0, len(data), self.BUFSIZ):
        if i + self.BUFSIZ < len(data):
            datagram = DataGram("SEQ", i // self.BUFSIZ, data[i:i + self.BUFSIZ])
            self.frames.append(datagram)
        else:
            datagram = DataGram("FIN", i // self.BUFSIZ, data[i:])
            self.frames.append(datagram)
```

通过将文件按照最大数据报中data段长度拆分并标上序号，在之前的每一段都标记为SEQ类型，最后一段标记为FIN类型，将这些报文装入发送方窗口的缓冲区后，即可正常发送。接收方在接收数据时，只需要提取出data段并按序拼接即可。

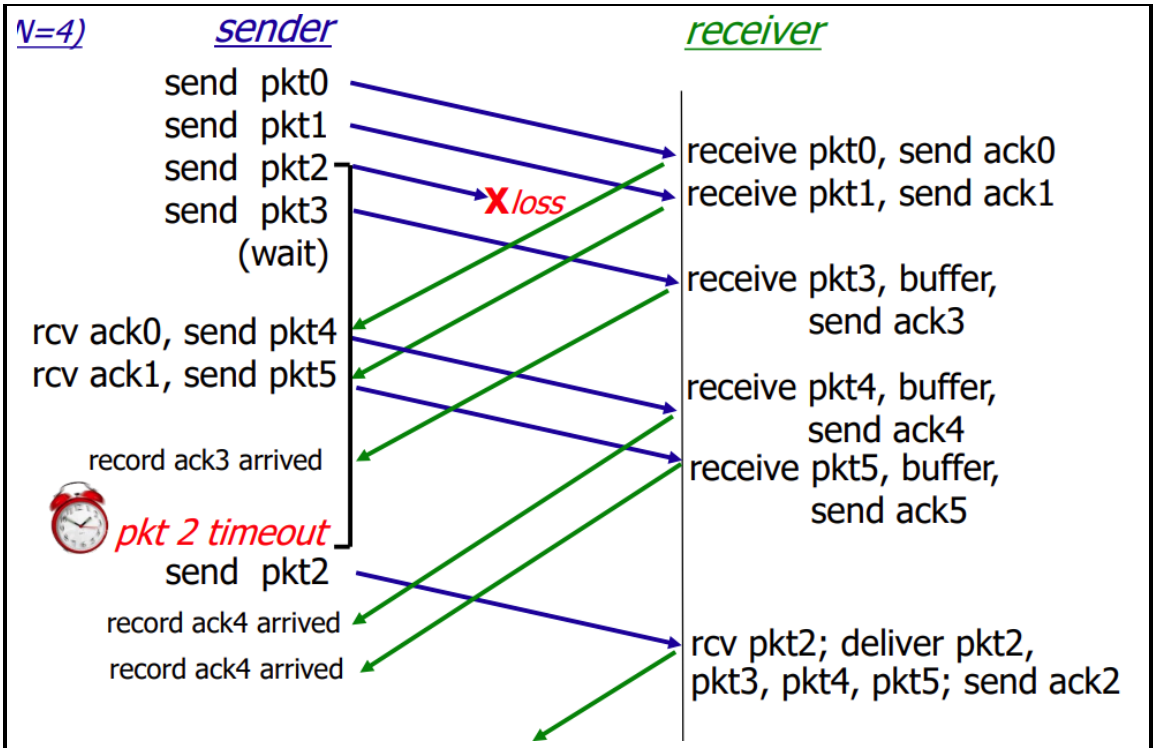
七、SR协议的实现思路与典型的交互过程

在SR协议中，我们需要为发送方和接收方都维护一个窗口。SR协议的窗口示意如下图所示：



与GBN协议类似，当我们开始SR协议时，发送方首先试图发满窗口并等待ACK报文。此时发送方为每一个报文都设置计时器，之后超时触发后，发送方仅仅重传超时的报文。对于接收方，它维护一个指示接收范围的窗口，丢弃范围外的报文，但是要对所有接收到的序号在send_base＋N之前的报文都返回一个对应的ACK，防止ACK丢失导致发送方卡住。发送方每次收到ACK时都要检查是否在某个序号之前的报文都全收到了ACK，如果有，就移动send_base并继续发送。

SR协议的一个典型的交互过程如下图所示：



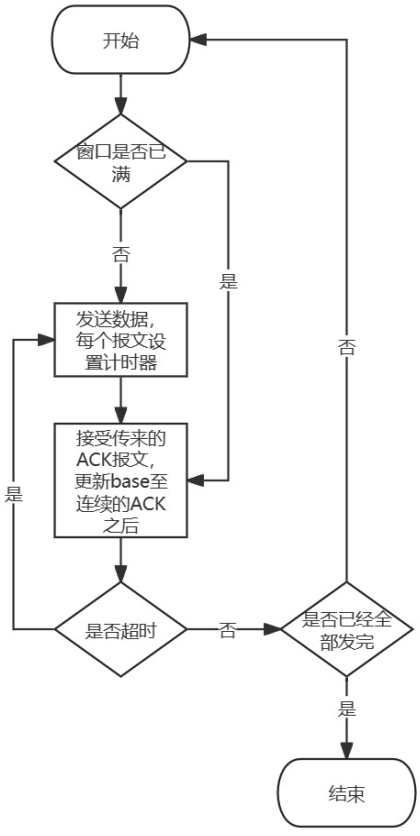
可以看到，当pkt2丢失时，接收方仍然正常给出ACK报文，直到pkt2对应的计时器超时后，发送方才重传pkt2。期间由于正常接收到ACK0, ACK1于是发送方可以移动窗口发送pkt4和pkt5。代码实现如下：

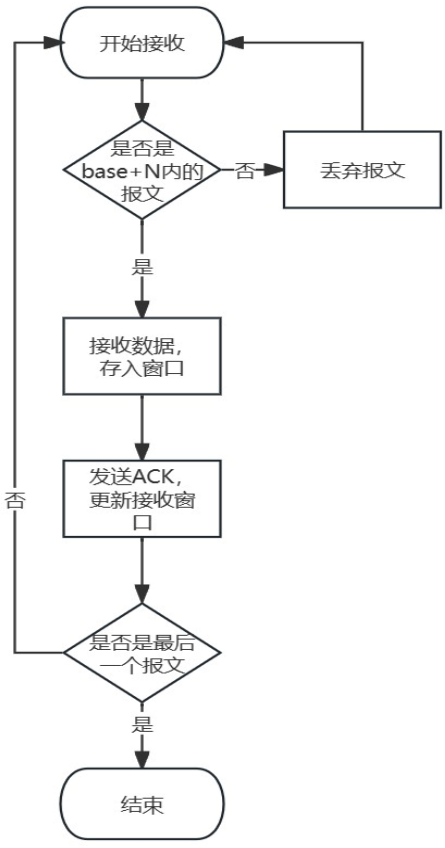
```
# 发送窗口
while self.base < len(self.frames):
    # 发送窗口内的数据包
    while self.next_seq < self.base + self.windowSize and self.next_seq <
len(self.frames):
        print("发送序号为", self.next_seq, "的数据包")
        self.sendFrame(self.frames[self.next_seq])
        self.timers[self.next_seq] = threading.Timer(2, self.SRSendALL, [self.next_seq])
        self.timers[self.next_seq].start()
        self.next_seq += 1 # next_seq 指向下一个要发送的数据包,每发一个数据包 next_seq+1
    # 循环接收 ACK
    while True:
        rcvDatagram, _ = self.ServerSocket.recvfrom(self.BUFSIZ)
        if self.randomDrop():
            print("丢弃 ACK", int(rcvDatagram.decode("UTF-8")[4:]))
            continue

        if rcvDatagram.decode("UTF-8")[0:3] == "ACK":
            self.acks[int(rcvDatagram.decode("UTF-8")[4:])] = True
            print("确认收到 ACK", int(rcvDatagram.decode("UTF-8")[4:]))
            self.timers[int(rcvDatagram.decode("UTF-8")[4:])].cancel()
            # 将 base 更新为在 base 和 next_seq 之间第一个未收到 ACK 的数据包
            if self.acks[self.base]:
```

```
            self.base += 1
        for i in range(self.base, self.next_seq):
            if not self.acks[i]:
                self.base = i
                # print("更新 base 为", self.base)
                # 重发 base 指向的数据包，这里是不符合协议操作的，只是为了产生大量的包，方便演示
                self.sendFrame(self.frames[self.base])
                break

        if self.base == self.next_seq:
            print("传输完成")

            return
        break

# 接受服务器的数据
while True:

    rcvDatagram, ServerAddr = ClientSocket.recvfrom(BUFSIZ)

    if client.randomDrop():
        print("丢弃序号为", rcvDatagram.decode("UTF-8")[
                        rcvDatagram.decode("UTF-8").find(" "):rcvDatagram.decode("UTF-
8").find("\n")],
              "的数据包")
        continue
    if rcvDatagram.decode("UTF-8")[0:3] == "SEQ":
        seq = int(rcvDatagram.decode("UTF-8")[
                rcvDatagram.decode("UTF-8").find(" "):rcvDatagram.decode("UTF-
8").find("\n")])
        # 检查 seq 是否在窗口内
        if seq < client.recvBase or seq >= client.recvBase + client.recvWindow:
            # 返回 ACK
            ack = DataGram("ACK", seq, "")
            ClientSocket.sendto(ack.makePacket, ServerAddr)
            continue

        print("接收序号为", seq, "的数据包")
        self.rcv[seq] = True
        fileBuffer[seq] = rcvDatagram.decode("UTF-8")[rcvDatagram.decode("UTF-
8").find("\n") + 1:]
        ack = DataGram("ACK", seq, "")
        print("发送 ACK", seq)
        ClientSocket.sendto(ack.makePacket, ServerAddr)
```

```
        while self.rcv[client.recvBase]:
            client.recvBase += 1
        continue

    if rcvDatagram.decode("UTF-8")[0:3] == "FIN":
        # 发送 ACK
        seq = int(rcvDatagram.decode("UTF-8")[
                rcvDatagram.decode("UTF-8").find(" "):rcvDatagram.decode("UTF-
8").find("\n")])
        ack = DataGram("ACK", seq, "")
        print("接收序号为", seq, "的数据包")
        ClientSocket.sendto(ack.makePacket, ServerAddr)
        self.rcv[seq] = True
        print("发送 ACK", seq)
        fileBuffer[seq] = rcvDatagram.decode("UTF-8")[rcvDatagram.decode("UTF-
8").find("\n") + 1:]
        # 如果在 seq 之前的 rcv 都为 True，说明文件接收完成
        if np.all(self.rcv[:seq + 1]):
            writeBufferToFile(fileBuffer)
            ClientSocket.close()
            break
        continue
```

八、SR协议的流程图





九、数据分组丢失验证模拟方法

在初始化类server和client时，内置一个参数dropRate，这个参数会在服务端收到ACK报文和客户端受到数据报文时用于随机数判断，若判定通过，就直接continue当前处理过程循环，意为丢弃该报文，代码如下：

```
if client.randomDrop():
    print("丢弃序号为", rcvDatagram.decode("UTF-8")[
                        rcvDatagram.decode("UTF-8").find(" "):rcvDatagram.decode("UTF-
8").find("\n")],
          "的数据包")
    continue
```

十、程序实现的主要类（或函数）及其主要作用

1、 writeBufferToFile函数，用于将接收方缓冲区中的内容输出到文件中

```
def writeBufferToFile(_fileBuffer, _outFilePath="result.txt"):
    """
    将 fileBuffer 中的数据写入文件
    """
    writeBuffer = _fileBuffer.tolist()
    with open(_outFilePath, "a") as f:
        for i in range(len(writeBuffer)):
            if writeBuffer[i] is not None:
                f.write(writeBuffer[i])
    print("文件接收完成")
    return
```

2、 GBNSendALL函数，用于在GBN协议中重传base到nextseq之间的所有报文

```
def GBNSendALL(self):
    '''
    重发在 base 和 next_seq 之间所有的数据包
    :return:
    '''
    if self.base == self.next_seq:
        return
    for i in range(self.base, self.next_seq):
        if i >= len(self.frames):
            break
        self.sendFrame(self.frames[i])
    print("重传窗口为", self.base, "到", self.next_seq - 1)
    self.timers = threading.Timer(5, self.GBNSendALL)
```

3、 SRSendALL函数，用于重传在base和nextseq之间所有超时的报文

```
def SRSendALL(self, _seq_num):
    """
    超时处理
```

```
    重发在 base 和 next_seq 之间某个数据包
    """
    if not self.acks[_seq_num]:
        print("超时，重发序号为", _seq_num, "的数据包")
        self.sendFrame(self.frames[_seq_num])
        self.timers[_seq_num] = threading.Timer(2, self.SRSendALL, [_seq_num])
        self.timers[_seq_num].start()
```

实验结果：

一、 GBN协议

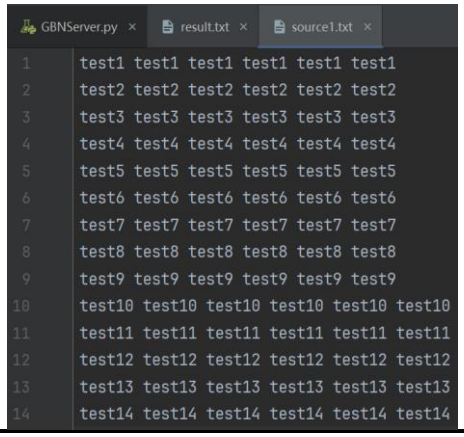
GBN协议的接收方接受情况如下图左所示，可以看到序号为11的包被随机丢弃，此后的12，13，14接收方都拒绝接受：

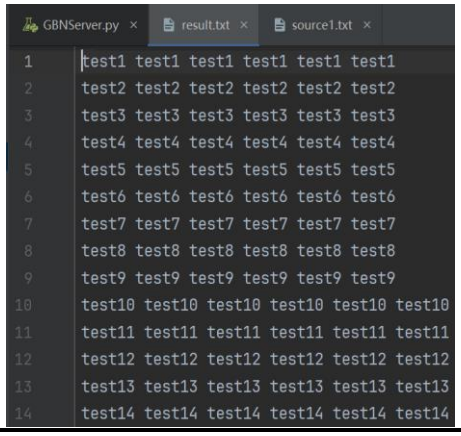
```
接收序号为 4 的数据包
接收序号为 5 的数据包
接收序号为 6 的数据包
接收序号为 7 的数据包
接收序号为 8 的数据包
接收序号为 9 的数据包
接收序号为 10 的数据包
丢弃序号为  11  的数据包
序号错误，期望序号为 11 实际序号为 12
序号错误，期望序号为 11 实际序号为 13
序号错误，期望序号为 11 实际序号为 14
接收序号为 11 的数据包
序号错误，期望序号为 12 实际序号为 11
接收序号为 12 的数据包
丢弃序号为  11  的数据包
序号错误，期望序号为 13 实际序号为 11
```

```
确认收到ACK6
确认收到ACK7
确认收到ACK8
确认收到ACK9
确认收到ACK10
确认收到ACK11
重传窗口为 11 到 14
重传窗口为 11 到 14
重传窗口为 11 到 14
```

此时，发送方检测到序号11的确认超时，于是开始重发窗口内11到14的数据报。

传输文件原文source1.txt内容如下：





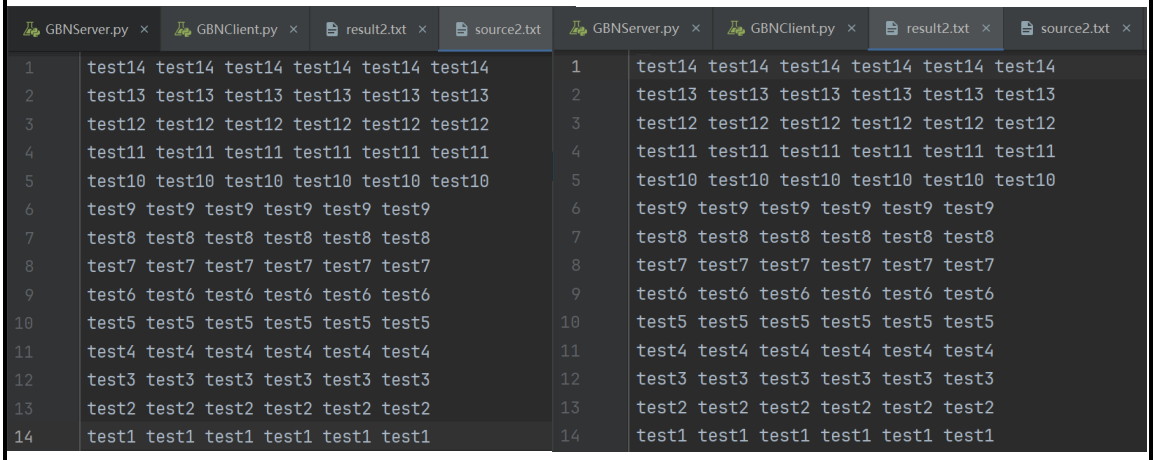
可以看到，接收方已经成功接受所有报文，并正确写入了文件。

当需要使用双向传输时，需要开启两个线程，但是会导致输出过于杂乱。这里仅通过更换代码主函数中的启动函数来做演示：

此时需要首先启动客户端来等待连接

```
(base) PS F:\Program Files (x86)\test\ComputerNetwork\Lab\pythonProject\GBN> python .\GBNClient.py
等待连接...
```

传输过程类似，当传输完成后，此时可以看到客户端将source2.txt文件成功传输给了服务端，服务端正确写入了result2.txt文件



二、SR协议

可以看到发送方接收到ACK后，确认base之后存在一段连续的ACK，于是移动窗口，因此可以发送更多的报文，而接收方也是乱序接收。最终发送方将source.txt发送给接收方，而接收方也能够正确地将结果输出到文件result.txt中。

source.txt × result.txt ×

```
test1 test1 test1 test1 test1 test1
test2 test2 test2 test2 test2 test2
test3 test3 test3 test3 test3 test3
test4 test4 test4 test4 test4 test4
test5 test5 test5 test5 test5 test5
test6 test6 test6 test6 test6 test6
test7 test7 test7 test7 test7 test7
test8 test8 test8 test8 test8 test8
test9 test9 test9 test9 test9 test9
test10 test10 test10 test10 test10 test10
test11 test11 test11 test11 test11 test11
test12 test12 test12 test12 test12 test12
test13 test13 test13 test13 test13 test13
test14 test14 test14 test14 test14 test14
```

source.txt × result.txt ×

```
test1 test1 test1 test1 test1 test1
test2 test2 test2 test2 test2 test2
test3 test3 test3 test3 test3 test3
test4 test4 test4 test4 test4 test4
test5 test5 test5 test5 test5 test5
test6 test6 test6 test6 test6 test6
test7 test7 test7 test7 test7 test7
test8 test8 test8 test8 test8 test8
test9 test9 test9 test9 test9 test9
test10 test10 test10 test10 test10 test10
test11 test11 test11 test11 test11 test11
test12 test12 test12 test12 test12 test12
test13 test13 test13 test13 test13 test13
test14 test14 test14 test14 test14 test14
```

问题讨论：

对实验过程中的思考问题进行讨论或回答。

一、GBN协议和SR协议应该都是对停等协议的一种精加工，它们在效率上都要胜过停等协议，但是需要消耗更多的资源，比如更多的计时器。在SR协议中，需要为每一个报文都设置一个计时器，而在GBN协议中，虽然计时器较少，但是每次都需要重发所有报文，也是浪费资源的。

二、虽然理论上UDP是无连接的协议，但是在python中使用socket编程会默认要先握手，也就是说直接发送报文有可能被这个机制给吞掉，因此设置了专门用于连接的CNT报文，在确定CNT之后再进入收发流程。

心得体会：

结合实验过程和结果给出实验的体会和收获。

一、深入理解了停等协议、GBN协议和SR协议的原理和实现流程

二、对于可靠数据传输有了更加深刻的认识

三、对于socket编程有了更深入的了解