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	实验 2：可靠数据传输协议-GBN 协议的设计与实现				
姓名	张志路		院系	计算机学院	
班级	1603106		学号	1160300909	
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	2018 年 11 月 7 日	
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					

计算机科学与技术学院 SINCE 1956...
School of Computer Science and Technology

目 录

实验二：可靠数据传输协议
——GBN 协议的设计与实现

1 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。

2 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 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。

3 实验环境

Windows 版本	Windows 10 中文版
系统类型	64 位操作系统，基于 x64 的处理器
Python 版本	Python 3.6.4
编程工具	PyCharm 5.0.3、Anaconda3-5.1.0

4 实验过程

以文字描述、实验结果截图等形式阐述实验过程，必要时可附相应的代码截图或以附件形式提交。

4.1 数据分组格式、确认分组格式、各个域作用

(1) 数据分组格式及作用

Seq	Data	0

其中，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。

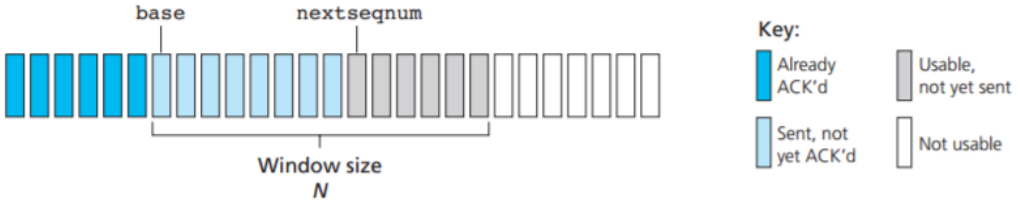
(2) 确认分组格式及作用

ACK	Seq	0

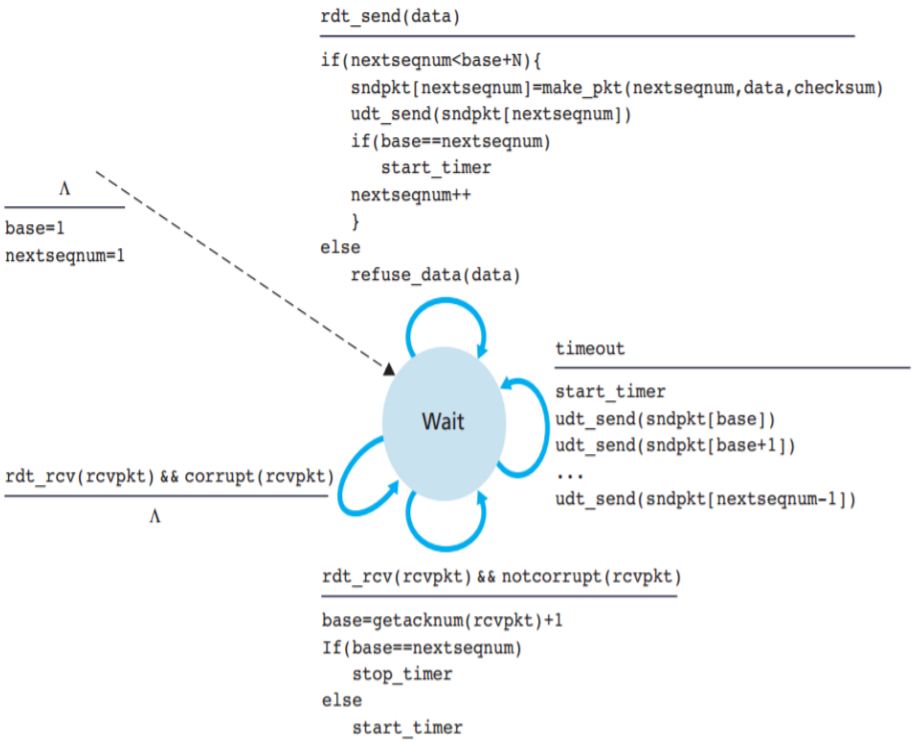
其中，Seq 为 1 个字节，标识 ack 序号；0 为 EOF0，表示结尾。

4.2 GBN 协议两端程序流程图及说明

(1) 发送方的序号空间



(2) 发送方的事件和动作



说明：GBN 发送方必须响应如下三种类型的事件。

a. 上层的调用

当上层调用 rdt_send 时，发送方首先检查发送窗口是否已满，即是否有 N 个已发送但未被确认的分组。

如果窗口未满，则产生一个分组并将其发送，并相应地更新变量。

如果窗口已满，发送方只需将数据返回给上层，隐式地指示上层该窗口已满。然后上层可能会过一会儿再试。

在实际实现中，发送方更可能缓存（并不立刻发送）这些数据，或者使用同步机制（如一个信号量或标志）允许上层在仅当窗口不满时才调用 rdt_send。

b. 收到一个 ACK

在 GBN 协议中，对序号为 n 的分组的确认采取累积确认的方式，表明接收方已正确接收到序号为 n 的以前且包括 n 在内的所有分组。

c. 超时事件

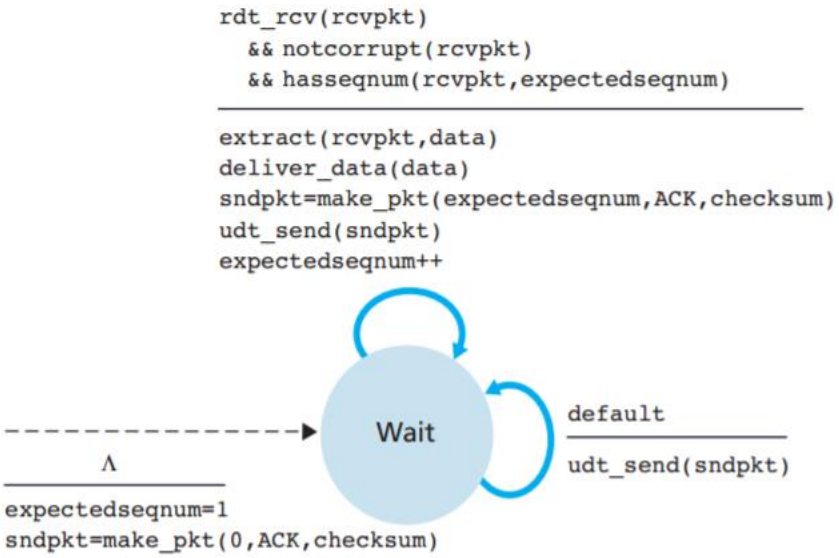
协议的名字（Go-Back-N）来源于出现丢失和时延过长分组时发送方的行为。就像在停等协议中那样，定时器将再次用于恢复数据或确认分组的丢失。

如果出现超时，发送方重传所有已发送但还未被确认过的分组。

如果收到一个 ACK，但仍有已发送但未确认的分组，则定时器被重新启动。

如果没有已发送但未被确认的分组，该定时器被终止。

(3) 接收方的事件和动作

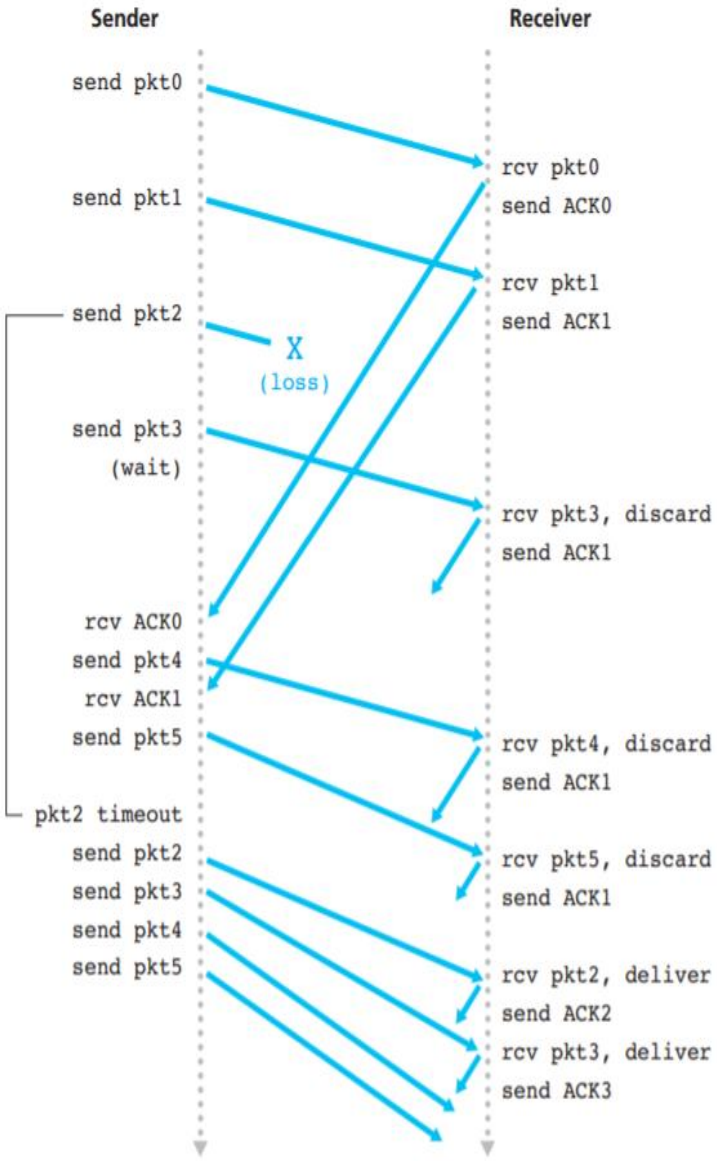


说明：GBN 接收方的具体流程如下。

如果一个序号为 n 的分组被正确接收到，并且按序到达，则接收方为分组 n 发送一个 ACK，并将该分组中的数据部分交付到上层。

在所有其他情况下，接收方丢弃该分组，并为最近按序接收的分组重新发送 ACK。

4.3 GBN 协议典型交互过程



上图给出了窗口长度为 4 个分组的 GBN 协议的运行情况。

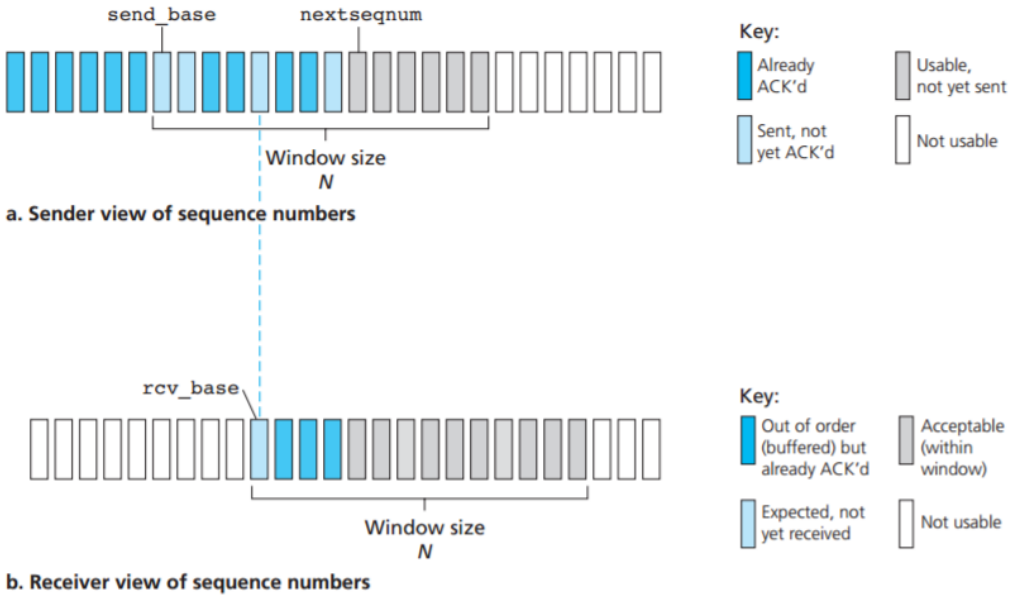
因为该窗口长度的限制，发送方发送分组 0~3,然后在继续发送之前，必须等待直到一个或多个分组被确认。

当接收到每一个连续的 ACK(例如 ACK 0 和 ACK 1)时，该窗口便向前滑动，发送方便可以发送新的分组（分别是分组 4 和分组 5)。

在接收方，分组 2 丢失，因此分组 3、4 和 5 被发现是失序分组并被丢弃。

4.4 SR 协议两端程序流程说明

(1) SR 发送方与接收方的序号空间



(2) 发送方的事件和动作

a. 从上层收到数据

当从上层接收到数据后，SR 发送方检查下一个可用于该分组的序号。

如果序号位于发送方的窗口内，则将数据打包并发送。否则就像在 GBN 中一样，要么将数据缓存，要么将其返回给上层以便以后传输。

b. 超时

定时器再次被用来防止丢失分组。然而，现在每个分组必须拥有其自己的逻辑定时器，因为超时发生后只能发送一个分组。可以使用单个硬件定时器模拟多个逻辑定时器的操作。

c. 收到 ACK

如果收到 ACK，倘若该分组序号在窗口内，则 SK 发送方将那个被确认的分组标记为已接收。

如果该分组的序号等于 send_base，则窗口基序号向前移动到具有最小序号的未确认分组处。

如果窗口移动了并且有序号落在窗口内的未发送分组，则发送这些分组。

(3) 接收方的事件和动作

接收端的事件和动作如下。

a. 序号在[rcv_base，rcv_hase+N-l]内的分组被正确接收。

在此情况下，收到的分组落在接收方的窗口内，一个选择 ACK 被回送给发送方。

如果该分组以前没收到过，则缓存该分组。

如果该分组的序号等于接收窗口的基序号 rcv_base，则该分组以及以前缓存的序号连续的（起始于 rcv_base 的）分组交付给上层。

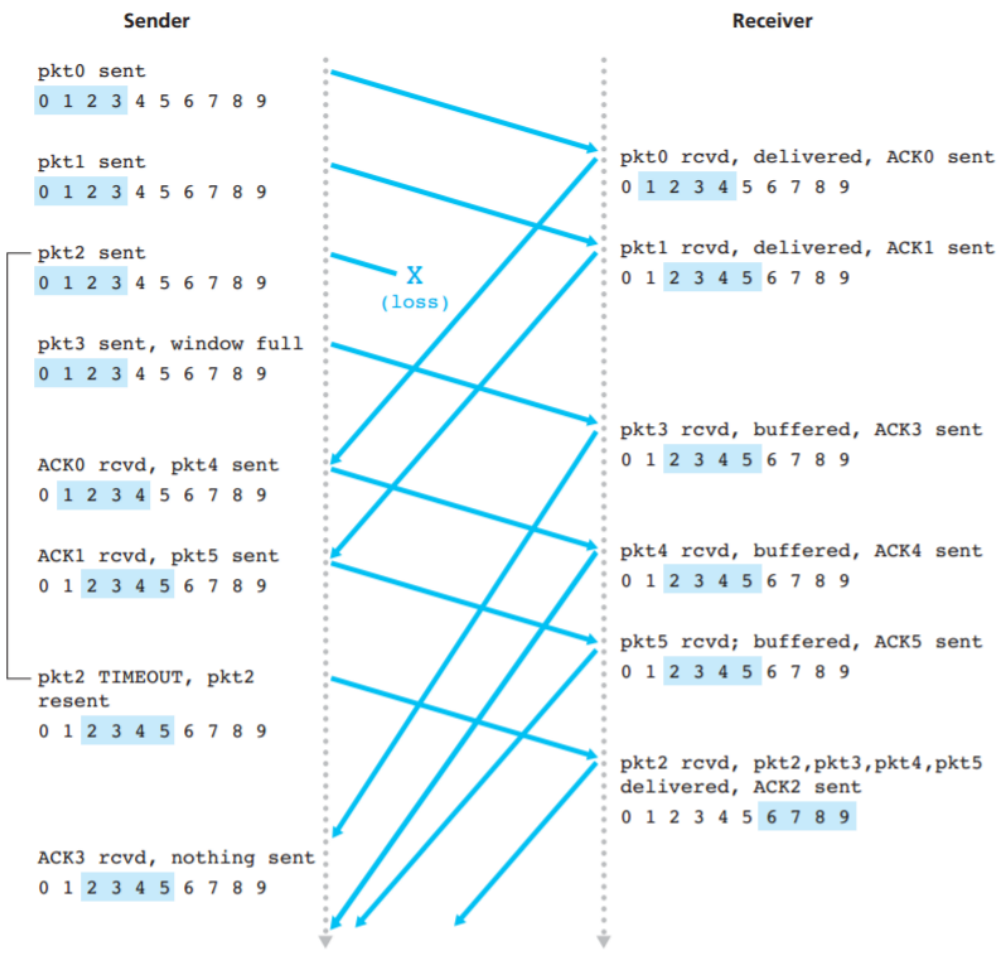
然后，接收窗口按向前移动分组的编号向上交付这些分组。

b. 序号在[rcv_base-N，rcv_hase-l]内的分组被正确收到。

在此情况下，必须产生一个 ACK，即使该分组是接收方以前已确认过的分组。

c. 其他情况。忽略该分组。

4.5 SR 协议典型交互过程



SR 接收方将确认一个正确接收的分组而不管其是否按序。失序的分组将被缓存直到所有丢失分组（即序号吏小的分组）皆被收到为止，这时才可以将一批分组按序交付给上层。

上图详细列出了 SR 接收方所采用的各种动作，并给出了一个例子以说明出现丢包时 SR 的操作。值得注意的是，在上图中接收方初始时缓存了分组 3、4、5，并在最终收到分组 2 时，才将它们一并交付给上层。

4.6 数据分组丢失验证模拟方法

丢包率设置为 0.25，利用 python 的 random()函数随机生成[0,1)范围内的一个实数，当它小于 0.25 时，数据报丢失。

```
1.  if random() < util.LOST_PACKET_RATIO:
2.  # 模拟数据报丢失 LOST_PACKET_RATIO 设置为 0.25
3.      continue
```

4.7 程序实现的主要类（或函数）及其主要作用

(1) gbn.py

该类实现 GBN 的发送方和接收方功能。

① send_data 函数

GBN 发送方，发送数据并接收 ack，按 4.2 所述流程设计代码逻辑。

② recv_data 函数

GBN 接收方，接收数据并发送确认，按 4.2 所述流程设计代码逻辑。

(2) sr.py

该类实现 SR 的发送方和接收方功能。

① send_data 函数

SR 发送方，发送数据并接收 ack，按 4.4 所述流程设计代码逻辑。

② recv_data 函数

SR 接收方，接收数据并发送确认，按 4.4 所述流程设计代码逻辑。

(3) server.py

该类进行命令解析，实现服务器（相对于单向而言）的接收和发送的控制功能。

① create_parser 函数

设置命令行运行的命令格式。

② server_send_data 函数

服务端发送数据，创建套接字并进行绑定，根据不同命令格式（协议、单双向）调用 sr.py 或者 gbn.py 中的 send_data 函数。

③ server_receive_data 函数（用于双向）

服务端接收数据，创建套接字并进行绑定，根据不同命令格式（协议、单双向）调用 sr.py 或者 gbn.py 中的 recv_data 函数。

④ 主模块

解析命令，根据不同命令格式（单向或者双向）进行相应处理，如果是双向，需要创建新线程，然后调用上述不同的函数。

(4) client.py

该类进行命令解析，实现客户端（相对于单向而言）的接收和发送的控制功能。

① create_parser 函数

设置命令行运行的命令格式。

② client_send_data 函数（用于双向）

客户端发送数据，创建套接字并进行绑定，根据不同命令格式（协议、单双向）调用 sr.py 或者 gbn.py 中的 send_data 函数。

③ server_receive_data 函数

客户端接收数据，创建套接字并进行绑定，根据不同命令格式（协议、单双向）调用 sr.py 或者 gbn.py 中的 recv_data 函数。

④ 主模块

解析命令，根据不同命令格式（单向或者双向）进行相应处理，如果是双向，需要创建新线程，然后调用上述不同的函数。

(5) util.py

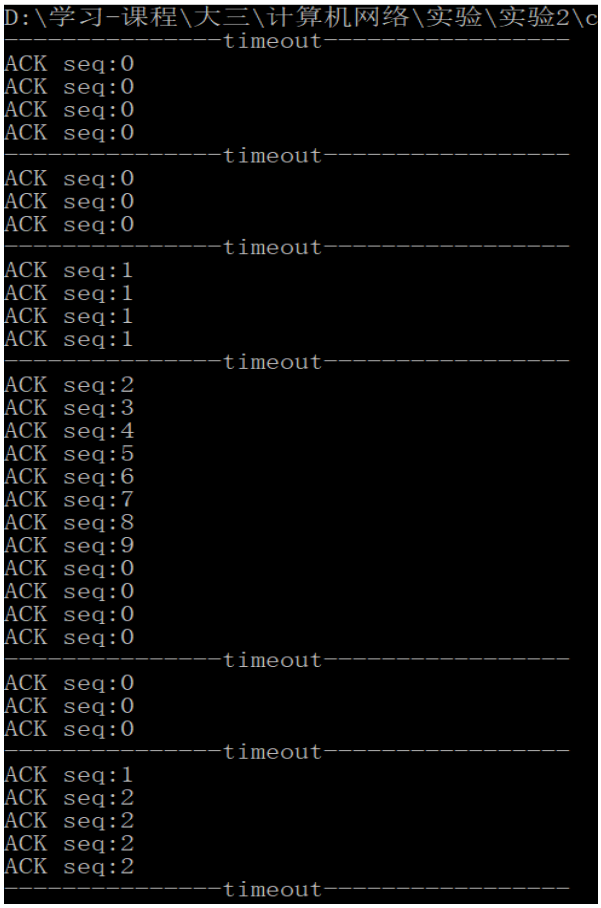
设置一系列参数，并设置数据帧的格式。

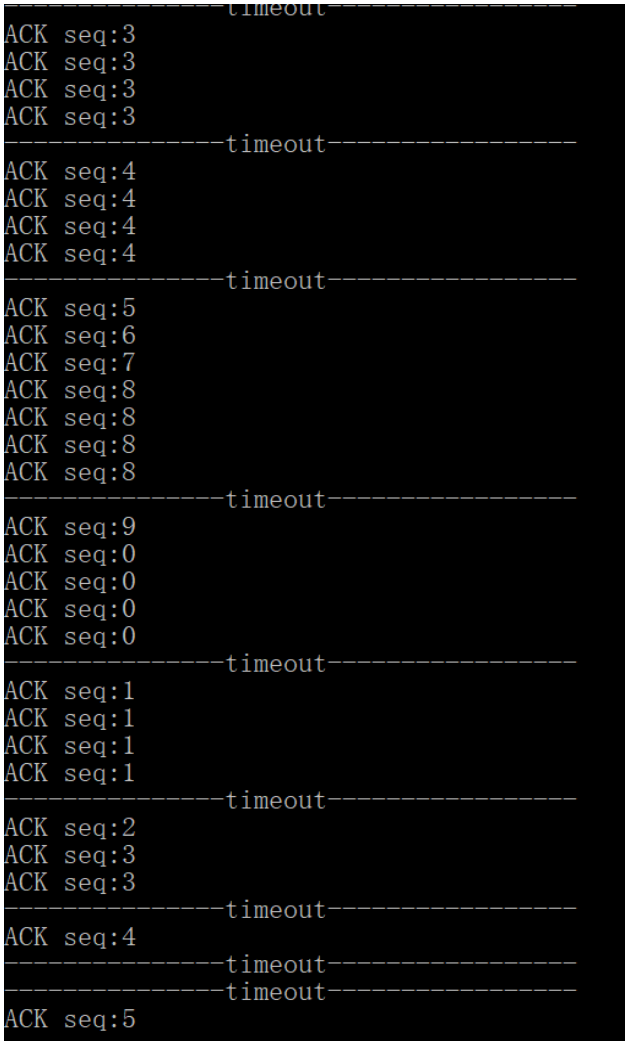
5 实验结果

设置两个文件进行传输，一个文件内容为 A-Z，一个文件内容为 a-z。

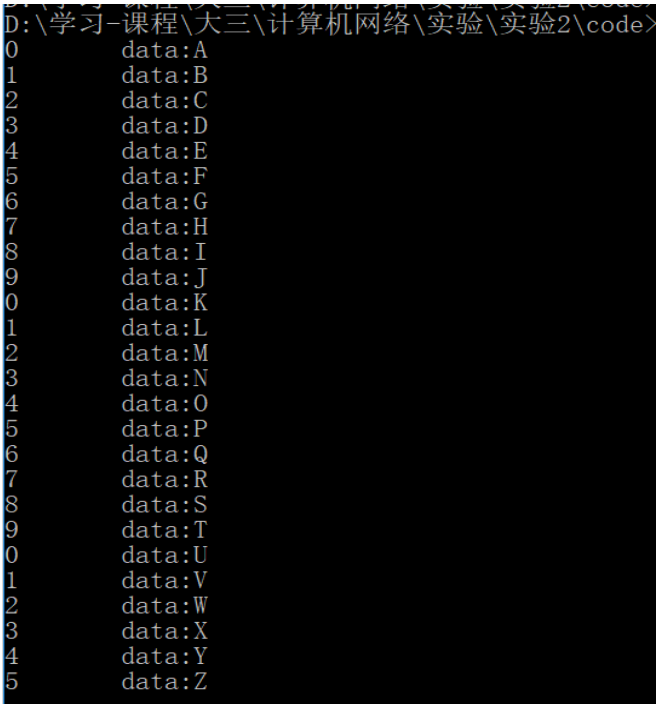
(1) 单向 gbn 演示

python server.py --protocol=GBN 命令运行服务端，数据传输时服务端接收 ack 和超时情况如下。



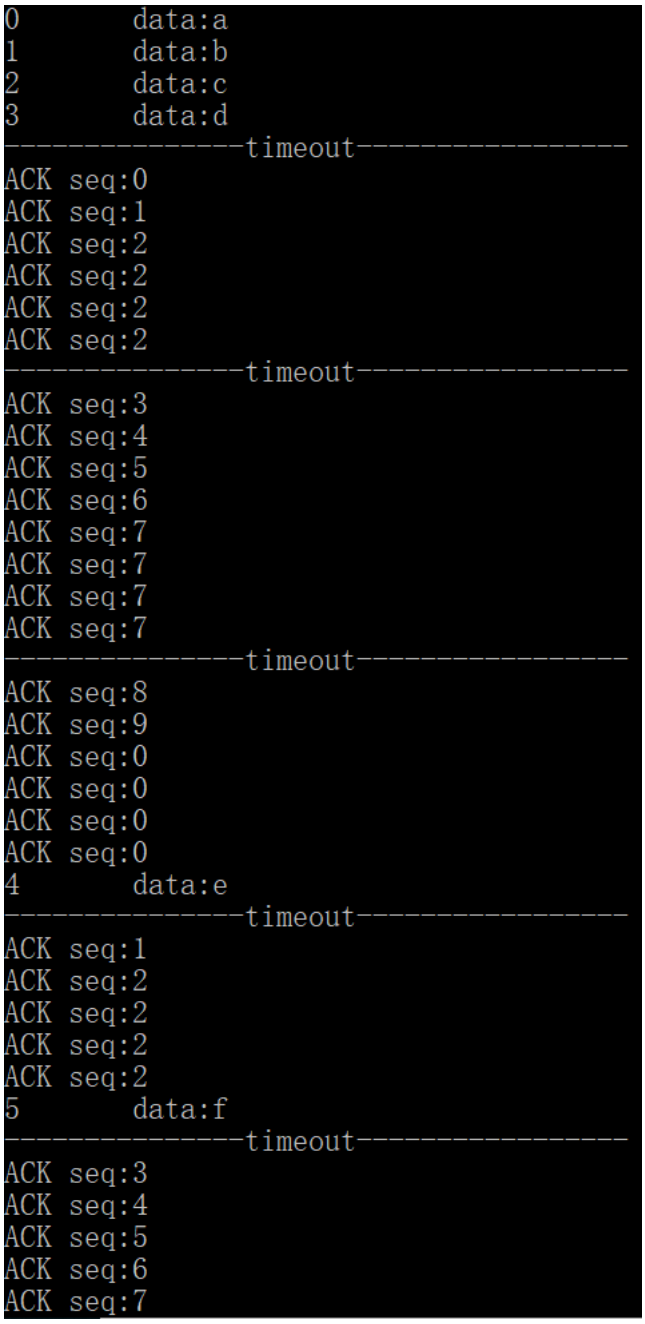


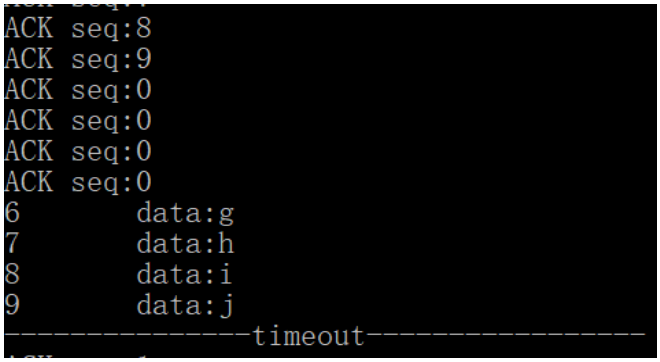
python client.py --protocol=GBN 命令运行客户端，数据传输时客户端接收数据情况如下。

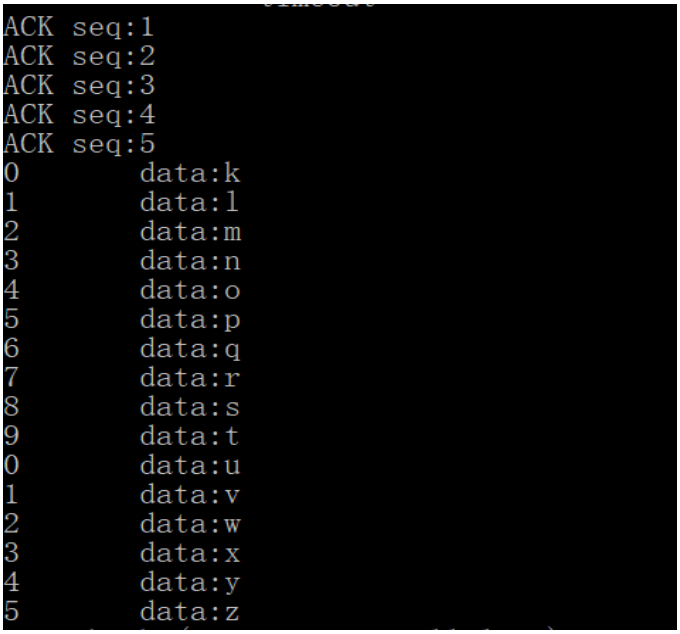


(2) 双向 gbn 演示

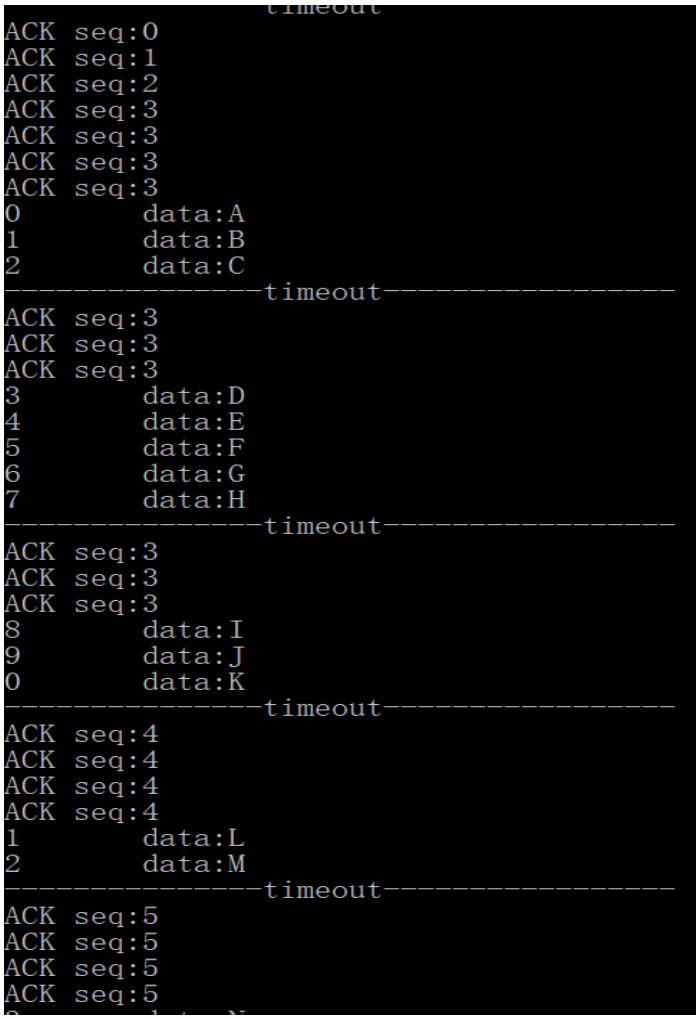
python server.py --protocol=GBN --dual=True 命令运行一端，数据传输时该端作为服务端接收 ack 和超时情况、作为客户端接收数据情况如下。

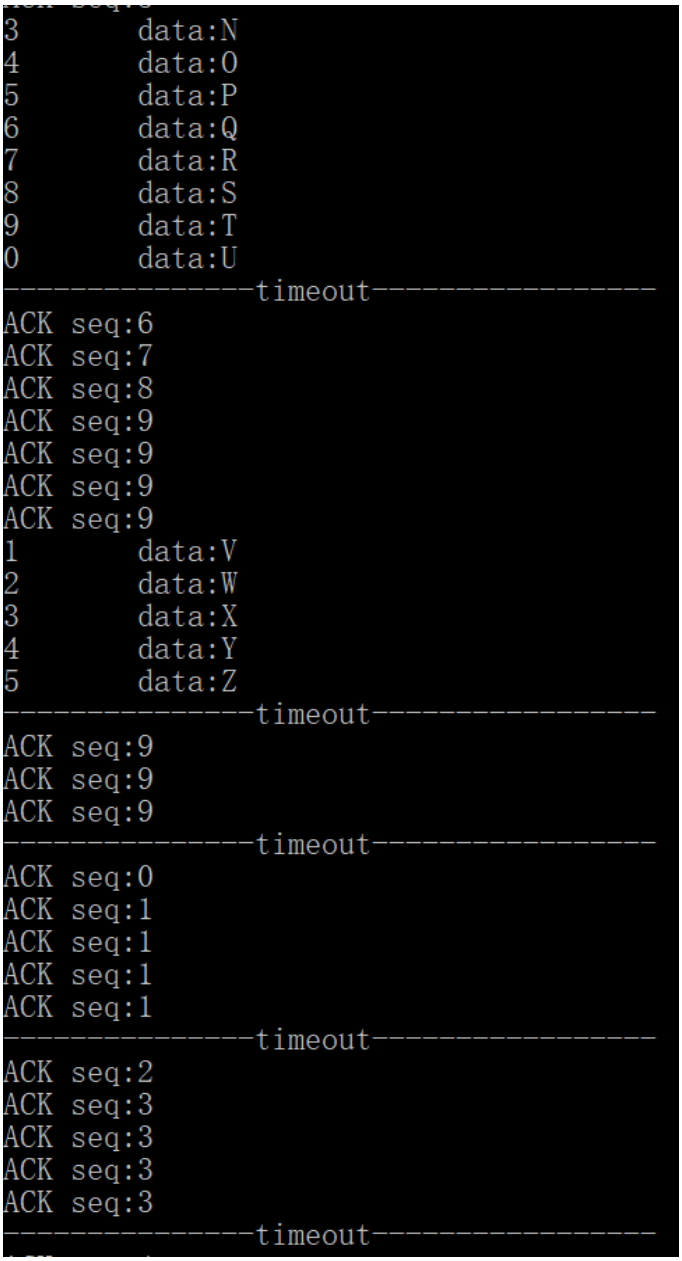


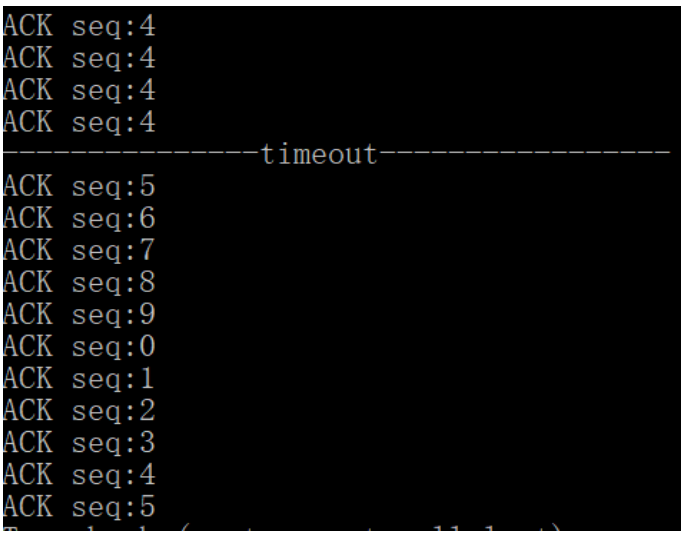




python client.py --protocol=GBN --dual=True 命令运行一端，数据传输时该端作为客户端接收数据情况、作为服务端接收 ack 和超时情况如下。




3 data:N
4 data:O
5 data:P
6 data:Q
7 data:R
8 data:S
9 data:T
0 data:U
----------timeout----------
ACK seq:6
ACK seq:7
ACK seq:8
ACK seq:9
ACK seq:9
ACK seq:9
ACK seq:9
1 data:V
2 data:W
3 data:X
4 data:Y
5 data:Z
----------timeout----------
ACK seq:9
ACK seq:9
ACK seq:9
----------timeout----------
ACK seq:0
ACK seq:1
ACK seq:1
ACK seq:1
ACK seq:1
----------timeout----------
ACK seq:2
ACK seq:3
ACK seq:3
ACK seq:3
ACK seq:3
----------timeout----------


ACK seq:4
ACK seq:4
ACK seq:4
ACK seq:4
----------timeout----------
ACK seq:5
ACK seq:6
ACK seq:7
ACK seq:8
ACK seq:9
ACK seq:0
ACK seq:1
ACK seq:2
ACK seq:3
ACK seq:4
ACK seq:5

(3) 单向 sr 演示

python server.py --protocol=SR 命令运行服务端，数据传输时服务端接收 ack 和超时情况如下。

```
ACK seq:0
ACK seq:2
ACK seq:4
-----------------timeout-----------------
-----------------timeout-----------------
ACK seq:1
ACK seq:5
-----------------timeout-----------------
-----------------timeout-----------------
ACK seq:6
-----------------timeout-----------------
ACK seq:3
ACK seq:8
ACK seq:9
-----------------timeout-----------------
-----------------timeout-----------------
ACK seq:7
ACK seq:0
ACK seq:1
ACK seq:2
ACK seq:3
ACK seq:5
ACK seq:6
-----------------timeout-----------------
-----------------timeout-----------------
ACK seq:4
ACK seq:7
ACK seq:8
ACK seq:9
ACK seq:0
ACK seq:1
ACK seq:2
ACK seq:3
ACK seq:4
ACK seq:5
```

python client.py --protocol=SR 命令运行客户端，数据传输时客户端接收数据情况如下。

```
0       data:A
0       data:A
0 data:A(交付上层)
1       data:B
2       data:C
2       data:C
3       data:D
4       data:E
4       data:E
1       data:B
1       data:B
1 data:B(交付上层)
2 data:C(交付上层)
```

```
3       data:D
5       data:F
5       data:F
6       data:G
3       data:D
6       data:G
6       data:G
3       data:D
3       data:D
3 data:D(交付上层)
4 data:E(交付上层)
5 data:F(交付上层)
6 data:G(交付上层)
7       data:H
8       data:I
8       data:I
9       data:J
9       data:J
0       data:K
7       data:H
7       data:H
7 data:H(交付上层)
8 data:I(交付上层)
9 data:J(交付上层)
0       data:K
0       data:K
0 data:K(交付上层)
1       data:L
1       data:L
1 data:L(交付上层)
2       data:M
2       data:M
2 data:M(交付上层)
3       data:N
3       data:N
3 data:N(交付上层)
```

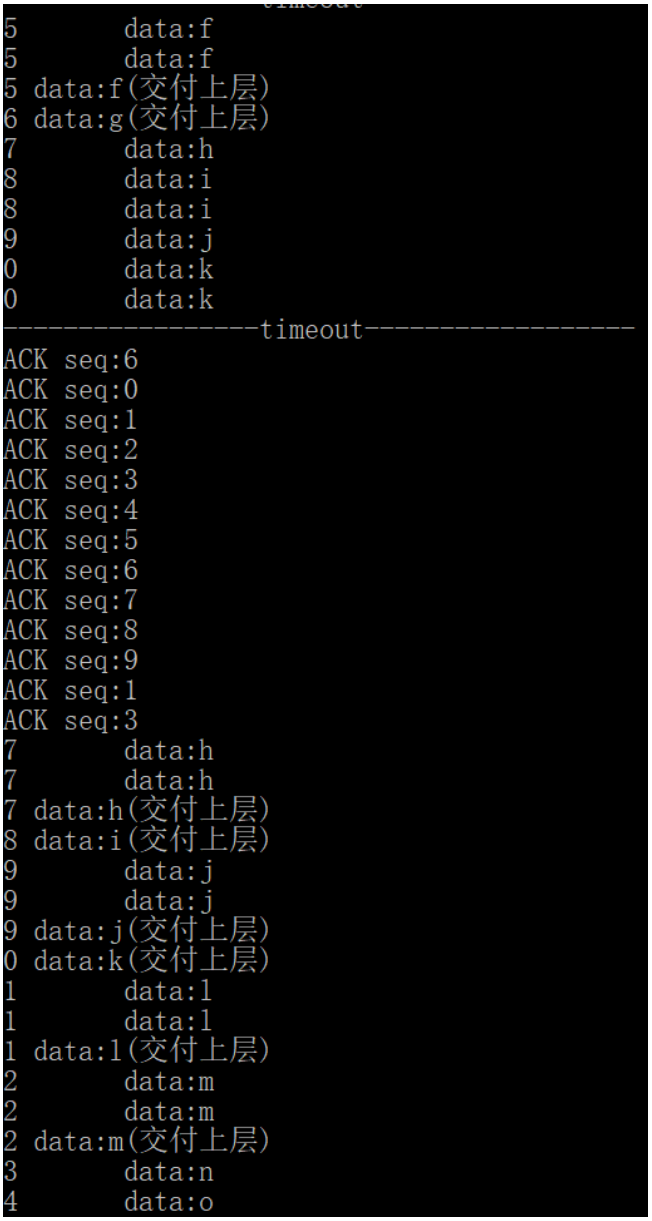
```
4       data:O
5       data:P
5       data:P
6       data:Q
6       data:Q
7       data:R
4       data:O
4       data:O
4 data:O(交付上层)
5 data:P(交付上层)
6 data:Q(交付上层)
7       data:R
7       data:R
7 data:R(交付上层)
8       data:S
8       data:S
8 data:S(交付上层)
9       data:T
9       data:T
9 data:T(交付上层)
```

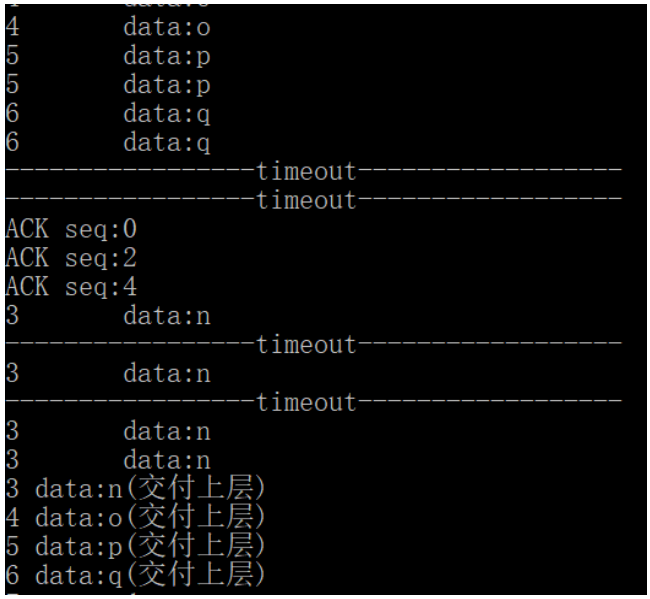
```
0       data:U
0       data:U
0 data:U(交付上层)
1       data:V
1       data:V
1 data:V(交付上层)
2       data:W
2       data:W
2 data:W(交付上层)
3       data:X
3       data:X
3 data:X(交付上层)
4       data:Y
4       data:Y
4 data:Y(交付上层)
5       data:Z
5       data:Z
5 data:Z(交付上层)
```

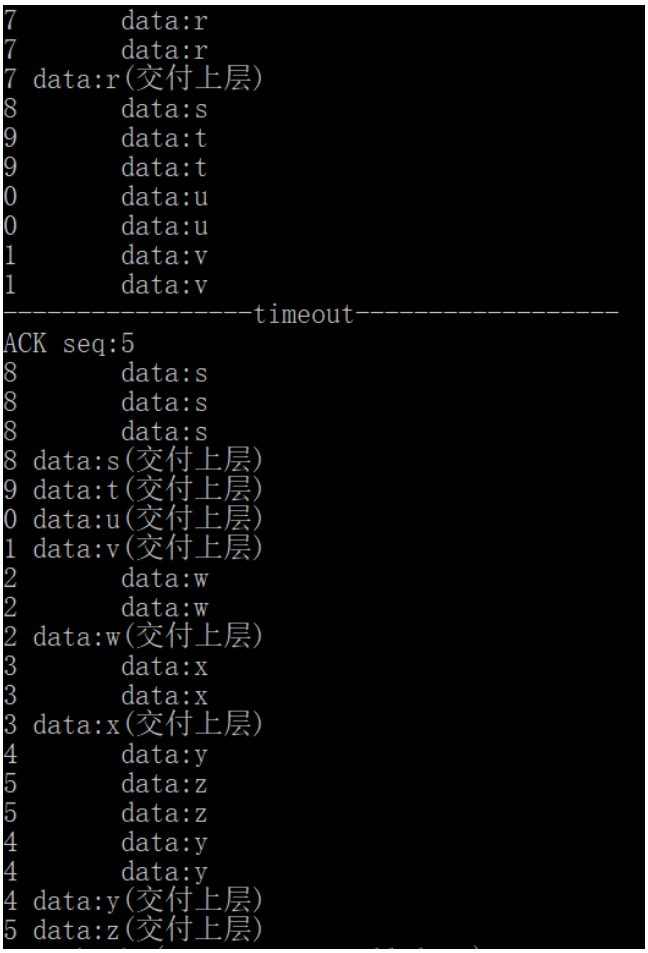
(4) 双向 sr 演示

python server.py --protocol=SR --dual=True 命令运行一端，数据传输时该端作为服务端接收 ack 和超时情况、作为客户端接收数据情况如下。

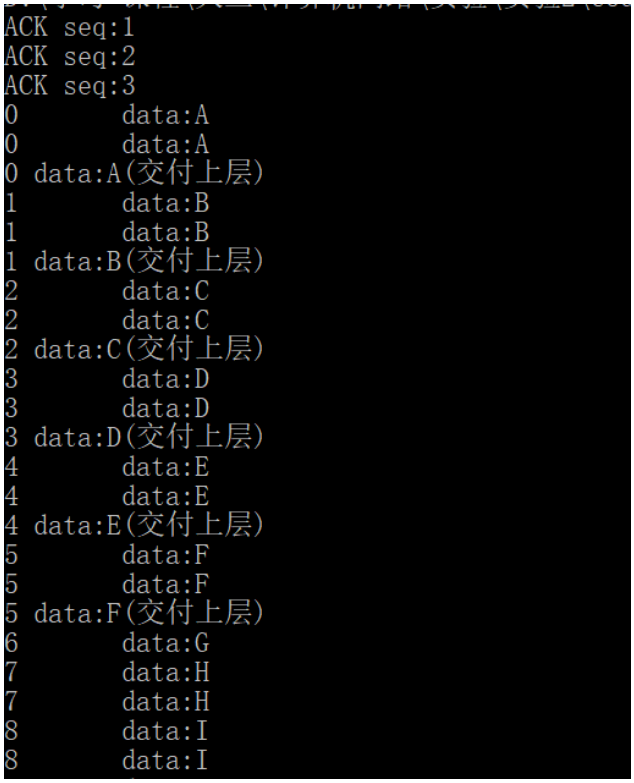
```
0       data:a
1       data:b
1       data:b
2       data:c
2       data:c
3       data:d
3       data:d
-----------------timeout-----------------
-----------------timeout-----------------
-----------------timeout-----------------
-----------------timeout-----------------
ACK seq:0
ACK seq:1
ACK seq:2
ACK seq:3
ACK seq:4
ACK seq:5
ACK seq:7
ACK seq:8
ACK seq:9
0       data:a
0       data:a
0 data:a(交付上层)
1 data:b(交付上层)
2 data:c(交付上层)
3 data:d(交付上层)
4       data:e
4       data:e
4 data:e(交付上层)
5       data:f
6       data:g
6       data:g
7       data:h
8       data:i
-----------------timeout-----------------
```


5 data:f
5 data:f
5 data:f(交付上层)
6 data:g(交付上层)
7 data:h
8 data:i
8 data:i
9 data:j
0 data:k
0 data:k
----------timeout----------
ACK seq:6
ACK seq:0
ACK seq:1
ACK seq:2
ACK seq:3
ACK seq:4
ACK seq:5
ACK seq:6
ACK seq:7
ACK seq:8
ACK seq:9
ACK seq:1
ACK seq:3
7 data:h
7 data:h
7 data:h(交付上层)
8 data:i(交付上层)
9 data:j
9 data:j
9 data:j(交付上层)
0 data:k(交付上层)
1 data:l
1 data:l
1 data:l(交付上层)
2 data:m
2 data:m
2 data:m(交付上层)
3 data:n
4 data:o


4 data:o
5 data:p
5 data:p
6 data:q
6 data:q
----------timeout----------
----------timeout----------
ACK seq:0
ACK seq:2
ACK seq:4
3 data:n
----------timeout----------
3 data:n
----------timeout----------
3 data:n
3 data:n
3 data:n(交付上层)
4 data:o(交付上层)
5 data:p(交付上层)
6 data:q(交付上层)

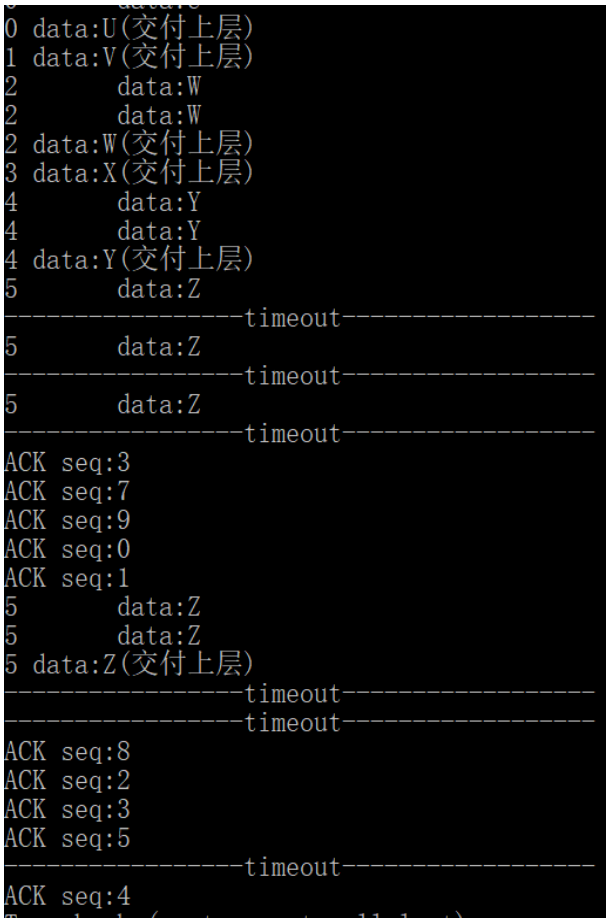


python client.py --protocol=SR --dual=True 命令运行一端，数据传输时该端作为客户端接收数据情况、作为服务端接收 ack 和超时情况如下。



```
9       data:J
9       data:J
-----------------timeout-----------------
ACK seq:0
ACK seq:4
ACK seq:6
6       data:G
-----------------timeout-----------------
-----------------timeout-----------------
-----------------timeout-----------------
ACK seq:5
ACK seq:8
ACK seq:0
6       data:G
6       data:G
6 data:G(交付上层)
7 data:H(交付上层)
8 data:I(交付上层)
9 data:J(交付上层)
0       data:K
0       data:K
0 data:K(交付上层)
1       data:L
1       data:L
1 data:L(交付上层)
2       data:M
2       data:M
2 data:M(交付上层)
3       data:N
3       data:N
3 data:N(交付上层)
4       data:O
4       data:O
4 data:O(交付上层)
5       data:P
5       data:P
5 data:P(交付上层)
```

```
6       data:Q
6       data:Q
6 data:Q(交付上层)
7       data:R
7       data:R
7 data:R(交付上层)
8       data:S
8       data:S
8 data:S(交付上层)
9       data:T
9       data:T
9 data:T(交付上层)
0       data:U
1       data:V
1       data:V
2       data:W
3       data:X
3       data:X
-----------------timeout-----------------
-----------------timeout-----------------
ACK seq:7
ACK seq:9
ACK seq:1
ACK seq:2
ACK seq:4
ACK seq:5
ACK seq:6
0       data:U
0       data:U
```



6 问题讨论

6.1 GBN 协议中，接收方丢弃所有失序分组的原因

在 GBN 协议中，接收方丢弃所有失序分组。尽管丢弃一个正确接收但失序的分组有点愚蠢和浪费，但这样做是有理由的。

假定现在期望接收分组 n，而分组 n+1 却到了。因为数据必须按序交付，接收方可能缓存分组 n+1，然后，在它收到并交付分组 n 后，再将该分组交付到上层。然而，如果分组 n 丢失，则该分组及分组 n+1 最终将在发送方根据 GBN 重传规则而被重传。因此，接收方只需丢弃分组 n+1 即可。

这种方法的优点是接收缓存简单，即接收方不需要缓存任何失序分组。因此，虽然发送方必须维护窗口的上下边界及 nextseqnum 在该窗口中的位置，但是接收方需要维护的唯一信总就是下一个按序接收的分组的序号。该值保存在 expectedseqnum 变量中。

当然，丢弃一个正确接收的分组的缺点是随后对该分组的重传也许会丢失或出错，因此甚至需要更多的重传。

6.3 SR 协议序列号空间大小与窗口尺寸的关系

对于 SR 协议，发送窗口大小 W_t、接收窗口大小 W_r 和比特数 n 必须满足关系式：$W_t + W_r \leqslant 2^n$ 。

只有这样才能够保证接收方能够区分新帧和重复帧。

6.2 可靠数据传输机制及其用途的总结

(1) 检验和

用于检测在一个传输分组中的比特错误。

(2) 定时器

用于超时/重传一个分组，可能因为该分组（或其 ACK)在信道中丢失了。由于当一个分组延时但未丢失（过早超时），或当一个分组已被接收方收到但从接收方到发送方的 ACK 丢失时，可能产生超时事件，所以接收方可能会收到一个分组的多个冗余副本。

(3) 序号

用于为从发送方流向接收方的数据分组按顺序编号。所接收分组的序号间的空隙可使接收方检测出丢失的分组。具有相同序号的分组可使接收方检测出一个分组的冗余副本。

(4) 确认

接收方用于告诉发送方一个分组或一组分组已被正确地接收到了。确认报文通常携带着被确认的分组或多个分组的序号。确认可以是逐个的或累积的，这取决于协议。

(5) 否定确认

接收方用于告诉发送方某个分组未被正确地接收。否定确认报文通常携带着未被正确接收的分组的序号。

(6) 窗口、流水线

发送方也许被限制仅发送那些序号落在一个指定范围内的分组。通过允许一次发送多个分组但未被确认，发送方的利用率可在停等操作模式的基础上得到增加。

7 心得体会

结合实验过程和结果给出实验的体会和收获。

通过这次实验，我理解了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。理解了滑动窗口协议的基本原理，掌握 GBN 和 SR 的工作原理，掌握基于 UDP 设计并实现一个 GBN 和 SR 协议的过程与技术。

纸上得来终觉浅，绝知此事要躬行。通过实验也使得我对该部分的理论内容加深了理解，一些原来理解有偏差的点得到纠正，真正起到了巩固与提升的作用，真正做到了学以致用。

附录

(1) util.py

```
LOST_PACKET_RATIO = 0.25 # 丢包率

'''''state 状态'''
NOT_SENT = 0 # 尚未发送数据报
SENT_NOT_ACKED = 1 # 已经发送数据包但是尚未确认
ACKED = 2 # 已经确认

CLIENT_IP = '127.0.0.1' # 客户端 ip 地址
SERVER_IP = '127.0.0.1' # 服务器 ip 地址

'''''端口号'''
SERVER_PORT_S = 10240 # 服务器发送
CLIENT_PORT_S = 10241 # 客户端发送
SERVER_PORT_R = 10242 # 服务器接收
CLIENT_PORT_R = 10243 # 客户端接收


class Data(object):
    '''''构造数据帧格式'''
    def __init__(self, msg, seq=0, state=NOT_SENT, seq_num=10):
        self.msg = msg #数据
        self.state = state # 状态
        self.seq = str(seq % seq_num) # 序列号

    '''''字符串形式返回'''
    def __str__(self):
        return self.seq + '\tdata:' + self.msg
```

(2) server.py

```
import socket
import _thread
import util
from gbn import GBN
```

```
from sr import SR
import argparse

'''''设置命令格式'''
def create_parser():
    parser = argparse.ArgumentParser(formatter_class=argparse.RawDescription
HelpFormatter, description='protocol(GBN or SR)')
    parser.add_argument('--
protocol', default=GBN, help='protocol name') #GBN or SR
    parser.add_argument('--
dual', default=False, help="whether dual transmission") # 单向 or 双向
    # 例如：python server.py --protocol=SR --dual=True
    return parser

'''''服务端发送数据'''
def server_send_data(server_port, client_ip, client_port, source_data, proto
col, lock=None):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # 创建 socket,传输层
协议是 UDP
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # SOL_SOCKET，意思是正在使用的 socket 选项；这里 value 设置为 1，表示将
SO_REUSEADDR 标记为 TRUE，
    # 操作系统会在服务器 socket 被关闭或服务器进程终止后马上释放该服务器的端口，否则
操作系统会保留几分钟该端口。
    s.bind(('', server_port)) # 绑定地址到套接字

    # 根据不同协议调用相应函数
    if protocol == 'GBN':
        p = GBN(s, host=client_ip)
    elif protocol == 'SR':
        p = SR(s, host=client_ip)

    # 服务器作为数据发送方向客户端发送数据
    if lock:
        p.send_data(source_data, client_port, lock)
    else:
        p.send_data(source_data, client_port)

'''''服务端接收数据'''
def server_receive_data(server_port, protocol):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # 创建 socket,传输层
协议是 UDP
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
    # SOL_SOCKET，意思是正在使用的 socket 选项；这里 value 设置为 1，表示将 SO_REUSEADDR 标记为 TRUE，
    # 操作系统会在服务器 socket 被关闭或服务器进程终止后马上释放该服务器的端口，否则操作系统会保留几分钟该端口。
    s.bind(('', server_port)) # 绑定地址到套接字

    # 根据不同协议调用相应函数
    if protocol == 'GBN':
        p = GBN(s)
    elif protocol == 'SR':
        p = SR(s)
    p.recv_data() # 从客户端接收数据并返回 ACK

'''''主模块'''
if __name__ == '__main__':
    parser = create_parser()
    args = parser.parse_args()
    if args.dual: # 双向多线程
        lock1 = _thread.allocate_lock() # 分配一个 lockType 的锁对象
        _thread.start_new_thread(server_send_data, (util.SERVER_PORT_S, util.CLIENT_IP, util.CLIENT_PORT_R, 'sdata.txt', args.protocol, lock1))
        server_receive_data(util.SERVER_PORT_R, args.protocol)
        while lock1.locked(): # 判断锁是锁定状态还是释放状态
            pass
    else: # 单向
        server_send_data(util.SERVER_PORT_S, util.CLIENT_IP, util.CLIENT_PORT_R, 'sdata.txt', args.protocol)
```

(3) client.py

```
import socket
import _thread
import util
from gbn import GBN
from sr import SR
import argparse

'''''设置命令格式'''
def create_parser():
    parser = argparse.ArgumentParser(formatter_class=argparse.RawDescriptionHelpFormatter,description='protocol(GBN or RS)')
    parser.add_argument('--protocol', default=GBN, help='protocol name')
    parser.add_argument('--dual', default=False, help="whether dual transmission")
    # 例如：python server.py --protocol=SR --dual=True
```

```
    return parser

'''客户端发送数据'''
def client_send_data(client_port, server_ip, server_port, source_data, protocol, lock=None):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # 创建 socket,传输层协议是 UDP
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # SOL_SOCKET，意思是正在使用的 socket 选项；这里 value 设置为 1，表示将 SO_REUSEADDR 标记为 TRUE，
    # 操作系统会在服务器 socket 被关闭或服务器进程终止后马上释放该服务器的端口，否则操作系统会保留几分钟该端口。
    s.bind(('', client_port)) # 绑定地址到套接字

    # 根据不同协议调用相应函数
    if protocol == 'GBN':
        p = GBN(s, host=server_ip)
    elif protocol == 'SR':
        p = SR(s, host=server_ip)

    # 客户端作为数据发送方向服务端发送数据
    if lock:
        p.send_data(source_data, server_port, lock)
    else:
        p.send_data(source_data, server_port)

'''服务端接收数据'''
def client_receive_data(client_port, protocol):
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # 创建 socket,传输层协议是 UDP
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # SOL_SOCKET，意思是正在使用的 socket 选项；这里 value 设置为 1，表示将 SO_REUSEADDR 标记为 TRUE，
    # 操作系统会在服务器 socket 被关闭或服务器进程终止后马上释放该服务器的端口，否则操作系统会保留几分钟该端口。
    s.bind(('', client_port)) # 绑定地址到套接字

    # 根据不同协议调用相应函数
    if protocol == 'GBN':
        p = GBN(s)
    elif protocol == 'SR':
        p = SR(s)

    p.recv_data()  # 客户端从服务端接收数据，并发送 ACK
```

```
51.
52. '''''主模块'''
53. if __name__ == '__main__':
54.     parser = create_parser()
55.     args = parser.parse_args()
56.     if args.dual:  # 双向多线程
57.         lock1 = _thread.allocate_lock() # 分配一个 lockType 的锁对象
58.         _thread.start_new_thread(client_send_data, (util.CLIENT_PORT_S, util
    .SERVER_IP,util.SERVER_PORT_R, 'cdata.txt', args.protocol, lock1))
59.         client_receive_data(util.CLIENT_PORT_R, args.protocol)
60.         while lock1.locked(): # 判断锁是锁定状态还是释放状态
61.             pass
62.     else: # 单向
63.         client_receive_data(util.CLIENT_PORT_R, args.protocol)
```

(4) gbn.py

```
1.  import util
2.  import select
3.  from random import random
4.
5.  '''''GBN 协议的发送方和接收方实现'''
6.  class GBN(object):
7.      '''''初始化'''
8.      def __init__(self, s, host='127.0.0.1', buffer_size=1024, window_size=4,
     seq_num=10, max_time=3):
9.          """
10.         :param s: 源套接字
11.         :param host: 目的主机'127.0.0.1'
12.         :param buffer_size: buffer 大小 1024
13.         :param window_size: 滑动窗口大小 4
14.         :param seq_num: 最大序列号 10
15.         :param max_time: 最大延迟时间 3
16.         """
17.         self.s = s
18.         self.host = host
19.         self.buffer_size = buffer_size
20.         self.window_size = window_size
21.         self.seq_num = seq_num
22.         self.max_time = max_time
23.
24.     '''''发送端发送数据并接收 ack'''
25.     def send_data(self, source_data, target_port, lock=None):
26.         """
27.         :param source_data: 数据文件
```

```
        :param target_port: 目的端口号 ，默认主机为 '127.0.0.1'
        :return: None
        """
        # 初始化
        if lock: # 获取锁
            lock.acquire()
        clock = 0
        seq = 0
        window = []

        # 开始传输数据到目标服务器
        with open(source_data, 'r') as f:
            while True:
                # 超时时,重传窗口内所有数据，并重新计时
                if clock > self.max_time:
                    print('---------------timeout-----------------')
                    clock = 0
                    for data in window:
                        data.state = util.NOT_SENT

                # 窗口内有剩余空间时，增加数据至满
                while len(window) < self.window_size:
                    line = f.readline().strip()
                    if not line:
                        break
                    data = util.Data(line, seq=seq) # 构造数据帧
                    window.append(data)
                    seq += 1

                if not window:
                    break

                # 窗口中的所有数据以 NOT_SENT 状态发送出去
                for data in window:
                    if data.state == util.NOT_SENT:
                        self.s.sendto(str(data).encode(), (self.host, target
_port)) # UDP 用 sendto 放方法
                        data.state = util.SENT_NOT_ACKED # 设置状态

                # 使用 select 方法监听套接字（非阻塞）
                readable_list, writeable_list, errors = select.select([self.
s, ], [], [], 1)
                # 目标服务器返回数据(例如 ack)
                if len(readable_list) > 0:
```

```
                    if random() < util.LOST_PACKET_RATIO: # 模拟 ack 丢失
                        continue
                    try: # 接收 ack,时钟重置为零
                        ack_message, address = self.s.recvfrom(self.buffer_s
ize) # UDP 使用 recvfrom 方法接收数据
                        print('ACK seq:' + ack_message.decode())
                        # 接收 ack 然后执行滑动窗口
                        for i in range(len(window)):
                            # seq<=ack 时认为已处于 ACKED 状态
                            if ack_message.decode() == window[i].seq:
                                clock = 0
                                window = window[i+1:]
                                break
                    except BaseException as e:
                        pass
                else:
                    clock += 1
        self.s.close()
        if lock: # 释放锁
            lock.release()

    '''''接收端接收数据并发送确认'''
    def recv_data(self):
        last_ack = self.seq_num - 1 # 记录最后的 ack 序列号:9
        window = []
        while True:
            # 使用 select 方法监听套接字（非阻塞）
            readable_list, writeable_list, errors = select.select([self.s, ]
, [], [], 1)
            # 若源服务器发送了数据
            if len(readable_list) > 0:
                message, address = self.s.recvfrom(self.buffer_size)
                 ackseq = int(message.decode().split()[0]) # 提取出序列号（数
据格式： seq \t data）
                if last_ack == (ackseq - 1) % self.seq_num: # 依次到达
                    if random() < util.LOST_PACKET_RATIO: # 模拟数据报丢失
                        continue
                    self.s.sendto(str(ackseq).encode(), address) #发送 ack
                    last_ack = ackseq # 更新 ack
                    if ackseq not in window:
                        window.append(ackseq)
                        print(message.decode())
                    while len(window) > self.window_size:
                        window.pop(0)
```

```
                else: # 乱序到达或丢包时，发送最后确认的序列号
                    self.s.sendto(str(last_ack).encode(), address)
        self.s.close()
```

(5) sr.py

```
import util
import select
from random import random
import time

'''''SR 协议的发送方和接收方实现'''
class SR(object):
    '''''初始化'''
    def __init__(self, s, host='127.0.0.1', buffer_size=1024, window_size=4, seq_num=10, max_time=3):
        """
        :param s: 源套接字
        :param host: 目的主机'127.0.0.1'
        :param buffer_size: buffer 大小 1024
        :param window_size: 滑动窗口大小 4
        :param seq_num: 最大序列号 10
        :param max_time: 允许最长延迟时间 3
        """
        self.s = s
        self.host = host
        self.buffer_size = buffer_size
        self.window_size = window_size
        self.seq_num = seq_num
        self.max_time = max_time

    '''''发送端发送数据并接收 ack'''
    def send_data(self, source_data, trgt_port, lock=None):
        """
        :param source_data: 数据文件
        :param trgt_port: 目的端口号 ，默认主机为 '127.0.0.1'
        :return: None
        """
        # 初始化
        if lock:
            lock.acquire() # 获取锁
        clock = {} # 记录超时的分组序列号
        seq = 0
        window = []

```

```
39.         # 开始传输数据到目标服务器
40.         with open(source_data, 'r') as f:
41.             while True:
42.                 # 超时时,重传窗口内所有数据，并重新计时
43.                 now_time = time.time()
44.                 list = []
45.                 for k in clock:
46.                     if now_time - clock[k] > self.max_time:
47.                         print('-----------------timeout-----------------
    ')
48.                         for data in window: # 发送方重传没收到 ACK 的分组
49.                             if data.state == util.SENT_NOT_ACKED:
50.                                 list.append(str(data.seq))
51.                                 data.state = util.NOT_SENT
52.                                 break
53.                 for k in list: # 重置计时器
54.                     clock.pop(k)
55.
56.                 # 窗口内有剩余空间时，增加数据至满
57.                 while len(window) < self.window_size:
58.                     line = f.readline().strip()
59.                     if not line:
60.                         break
61.                     data = util.Data(line, seq=seq) # 构造数据帧
62.                     window.append(data)
63.                     seq += 1
64.
65.                 if not window:
66.                     break
67.
68.                 # 窗口中的所有数据以 NOT_SENT 状态发送出去
69.                 for data in window:
70.                     if not data.state:
71.                         self.s.sendto(str(data).encode(), (self.host, trgt_p
    ort)) # UDP 用 sendto 放方法
72.                         clock[str(data.seq)] = time.time() # 设置时间
73.                         data.state = util.SENT_NOT_ACKED # 设置状态
74.
75.                 # 使用 select 方法监听套接字（非阻塞）
76.                 readable_list, writeable_list, errors = select.select([self.
    s, ], [], [], 1)
77.                 #readable_list, writeable_list, errors = select.select([self
    .s],[],[self.s],1)
78.
```

```
79.                 #  目标服务器返回数据(例如 ack)
80.                 if len(readable_list) > 0:
81.                     if random() < util.LOST_PACKET_RATIO: # 模拟 ack 丢失
82.                         continue
83.                     try:# 接收 ack,时钟重置为零
84.                         ack_message, address = self.s.recvfrom(self.buffer_s
    ize) # UDP 使用 recvfrom 方法接收数据
85.                         print('ACK seq:' + ack_message.decode())
86.                         clock.pop(ack_message.decode())
87.                         for data in window: # 设置状态
88.                             if ack_message.decode() == data.seq:
89.                                 data.state = util.ACKED
90.                                 break
91.                     except BaseException as e:
92.                         pass
93.                 else:
94.                     pass
95.
96.                 # 滑动窗口
97.                 while window[0].state == util.ACKED:
98.                     window.pop(0)
99.                     if not window:
100.                         break
101.         self.s.close()
102.         if lock: # 释放锁
103.             lock.release()
104.
105.     '''''接收端接收数据并发送确认'''
106.     def recv_data(self):
107.         seq = 0 # 记录接收器窗口中的 base 序号
108.         window = {} # 记录序列号对应的数据
109.         while True:
110.             # 使用 select 方法监听套接字（非阻塞）
111.             readable_list, writeable_list, errors = select.select([self.s,
    ], [], [], 1)
112.             # 若源服务器发送了数据
113.             if len(readable_list) > 0:
114.                 message, address = self.s.recvfrom(self.buffer_size)
115.                 print(message.decode())
116.                 ack = message.decode().split()[0]
117.                 if random() < util.LOST_PACKET_RATIO: # 模拟数据报丢失
118.                     continue
119.                 if ack in self.__sr_now_acklist(seq): # 收到顺序到达的分组
120.                     print(message.decode())
```

```
121.                     self.s.sendto(ack.encode(), address)
122.                     window[ack] = message.decode().split()[1]
123.                     # 滑动窗口
124.                     while str(seq) in window:
125.                         print(str(seq) + ' ' + window[str(seq)] + '(交付上
    层)')
126.                         window.pop(str(seq))
127.                         seq = (seq + 1) % self.seq_num
128.                 elif ack in self.__sr_old_acklist(seq): # 收到乱序到达的分
    组
129.                     self.s.sendto(ack.encode(), address)
130.                 else:
131.                     pass
132.         self.s.close()
133.
134.     '''''接收器窗口中从 base 到 base+window_size-1 的 seq'''
135.     def __sr_now_acklist(self, base):
136.         ret = []
137.         end = base + self.window_size - 1
138.         while base <= end:
139.             ret.append(str(base % self.seq_num))
140.             base += 1
141.         return ret
142.
143.     '''''接收器窗口中从 base-window 到 base-1 的 seq(以避免 ack 丢失)'''
144.     def __sr_old_acklist(self, base):
145.         ret = []
146.         begin = base - self.window_size
147.         end = base - 1
148.         while begin <= end:
149.             ret.append(str(begin % self.seq_num))
150.             begin += 1
151.         return ret
```