**计算机网络专题实验 实验八报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 姓名： |  | 班级： |  |
| 姓名： |  | 班级： |  |
|  |  |  |  |

一、实验名称

### 实现一个可靠数据传输协议

二、实验原理

1 UDP（User Datagram Protocol）

核心原理：

- 无连接

- 直接发送数据，无需握手或维护连接状态。

- 不可靠性

- 不保证数据到达、不重传丢失报文、不保证顺序。

- 无ACK机制，发送后即丢弃本地副本。

- 轻量级

- 头部仅8字节（TCP至少20字节），包含源/目的端口、长度、校验和。

- 无流量控制、拥塞控制，传输效率高。

三、实验目的

1)掌握Sockets的相关基础知识，学习Sockets编程的基本函数和模式、框架。

2)掌握UDP、TCP协议及 Client/Server和P2P两种模式的通信原理。掌握可靠数据传输协议（GBN和SR协议）

3)掌握socket编程框架

四、实验内容

1. 基于 UDP 设计一个简单的 GBN + SR协议，实现单向可靠数据传输（服 务器到客户的数据传输）。
2. 基本功能包括：基于该可靠数据传输协议传输文件。在广域网上测试该协议的效率。
3. 进阶：改进所设计的 GBN + SR 协议，支持双向数据传输。在广域网上测试该协议的效率。

五、实验实现

1. 人员分工
2. 实验设计
3. 协议

**基础：**

协议设计核心思想

1. 发送方（GBN特性）：

- 累计确认：收到ACK后，窗口基序号（`base`）直接跳至`ack\_num + 1`，认为此前所有包均被正确接收。

- 超时重传：若发生超时，重传当前窗口内所有未确认的包（从`base`到`next\_seq`）。

- 窗口滑动：发送窗口大小为固定值（`WINDOW\_SIZE`），按序发送窗口内的包。

2. 接收方（SR特性）：

- 乱序包缓存：允许缓存窗口内乱序到达的包（如收到包2但未收包1，则缓存包2）。

- 按序提交：仅当收到期望序号（`expected\_seq`）的包时，才将数据提交给上层，并检查缓冲区是否能继续提交后续包。

- 单独ACK：对每个正确接收的包发送对应的ACK（非累计）。

**进阶：**

协议核心思想

1. 发送方（GBN特性）：

- 累计确认：仅维护窗口基序号（base），收到ACK后，窗口滑动至ack\_num + 1，认为此前所有包已正确接收。

- 超时重传：超时触发时，重传窗口内所有未确认包（从base到next\_seq），确保可靠性。

- 固定窗口大小：窗口范围为[base, base + WINDOW\_SIZE)，按序发送窗口内的包。

2. 接收方（SR特性）：

- 乱序包缓存：使用buffer缓存窗口内乱序到达的包（如收到包3但未收包2，则缓存包3）。

- 按序提交：仅当收到expected\_seq的包时，将数据提交给上层，并检查缓冲区是否能继续提交后续包。

- 单独ACK：对每个正确接收的包发送独立的ACK（如收到包2，发送ACK=2）。

1. UI设计

直接利用终端界面

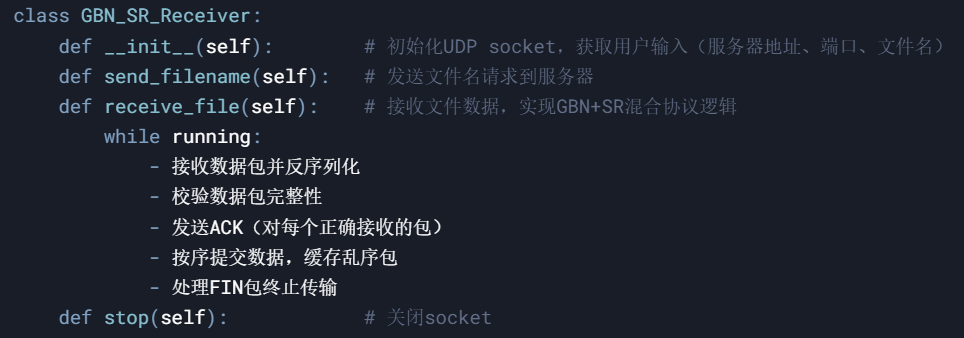
1. 框架结构

**基础：**

1. 客户端程序框架

- 核心功能：请求服务器发送指定文件，接收文件数据并保存。

- 框架结构：



- 特点：

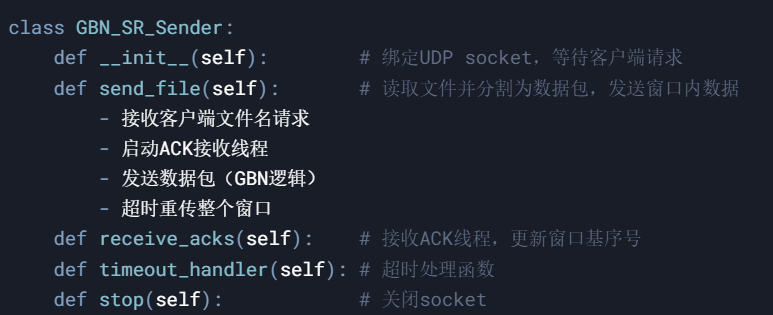
- 单线程设计，主线程处理接收逻辑。

- 使用UDP协议，无连接状态管理。

2. 服务器端程序框架

- 核心功能：监听客户端请求，按需发送文件数据。

- 当前实现（单客户端）：



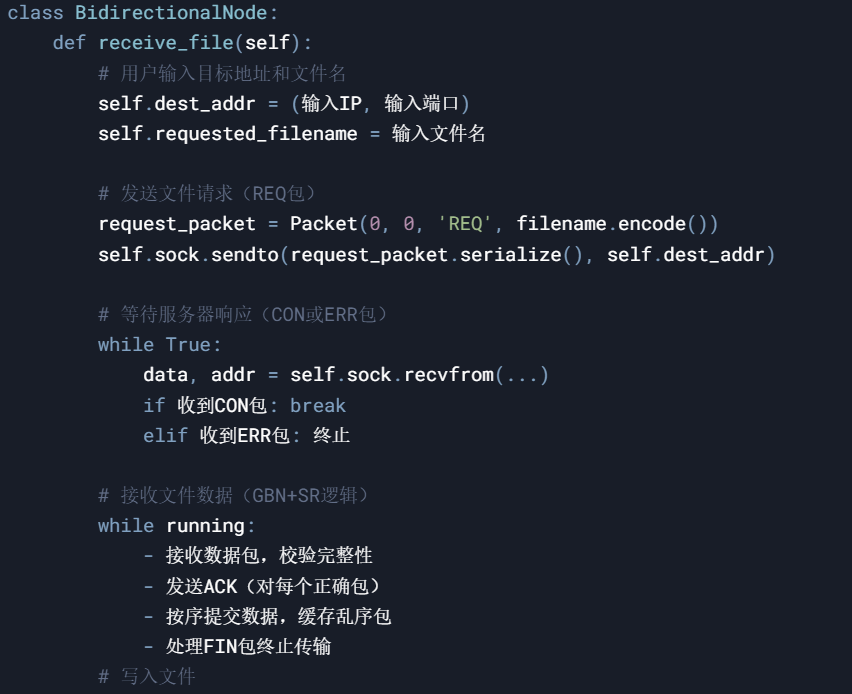
- 问题：当前仅支持单客户端连接，无法并发处理多用户请求。

**进阶：**

1. 客户端程序框架（BidirectionalNode作为Receiver）

角色：用户选择receiver时，客户端主动请求文件并接收数据。

核心流程：



- 特点：

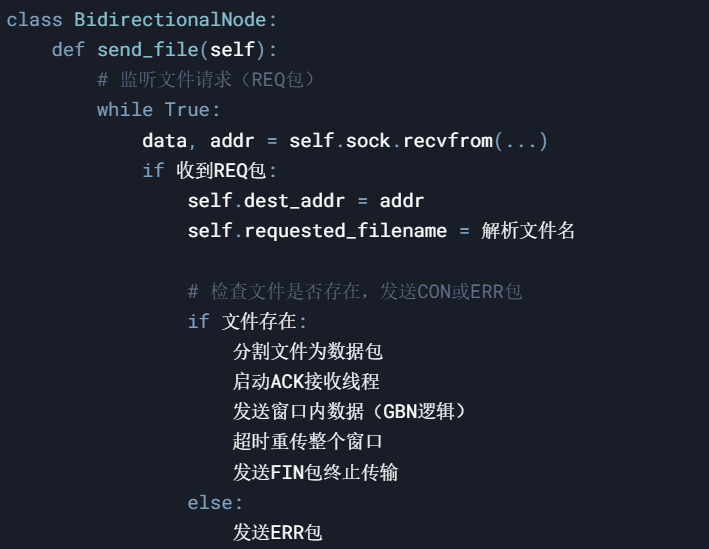
- 单线程设计，主线程处理请求和接收逻辑。

- 使用UDP协议，无连接状态管理。

- 支持SR机制的乱序包缓存。

2. 服务器端程序框架（`BidirectionalNode`作为Sender）

- 角色：用户选择`sender`时，服务器监听请求并发送文件。

- 核心流程：

- 当前限制：仅支持单客户端顺序处理，无法并发。

1. 关键代码的描述

【主要代码源文件名及功能描述，是借鉴还是自主编写，自主编写需注明编写人。】

**基础：**

1. Packet类（公共模块）。

功能：定义数据包结构，实现序列化、反序列化及校验功能。



解释：

序列化：将数据包头部（序列号、确认号、标志位）和数据部分打包为二进制，并附加CRC32校验和。

反序列化：通过deserialize函数还原数据包对象。

校验：使用CRC32校验确保数据完整性。

1. 客户端核心逻辑（GBN\_SR\_Receiver类）exp8\_client\_receiver.py

功能：请求文件并接收数据，实现GBN+SR混合协议。

解释：

- 按序提交：仅当收到期望序列号的包时，提交数据并递增`expected\_seq`。

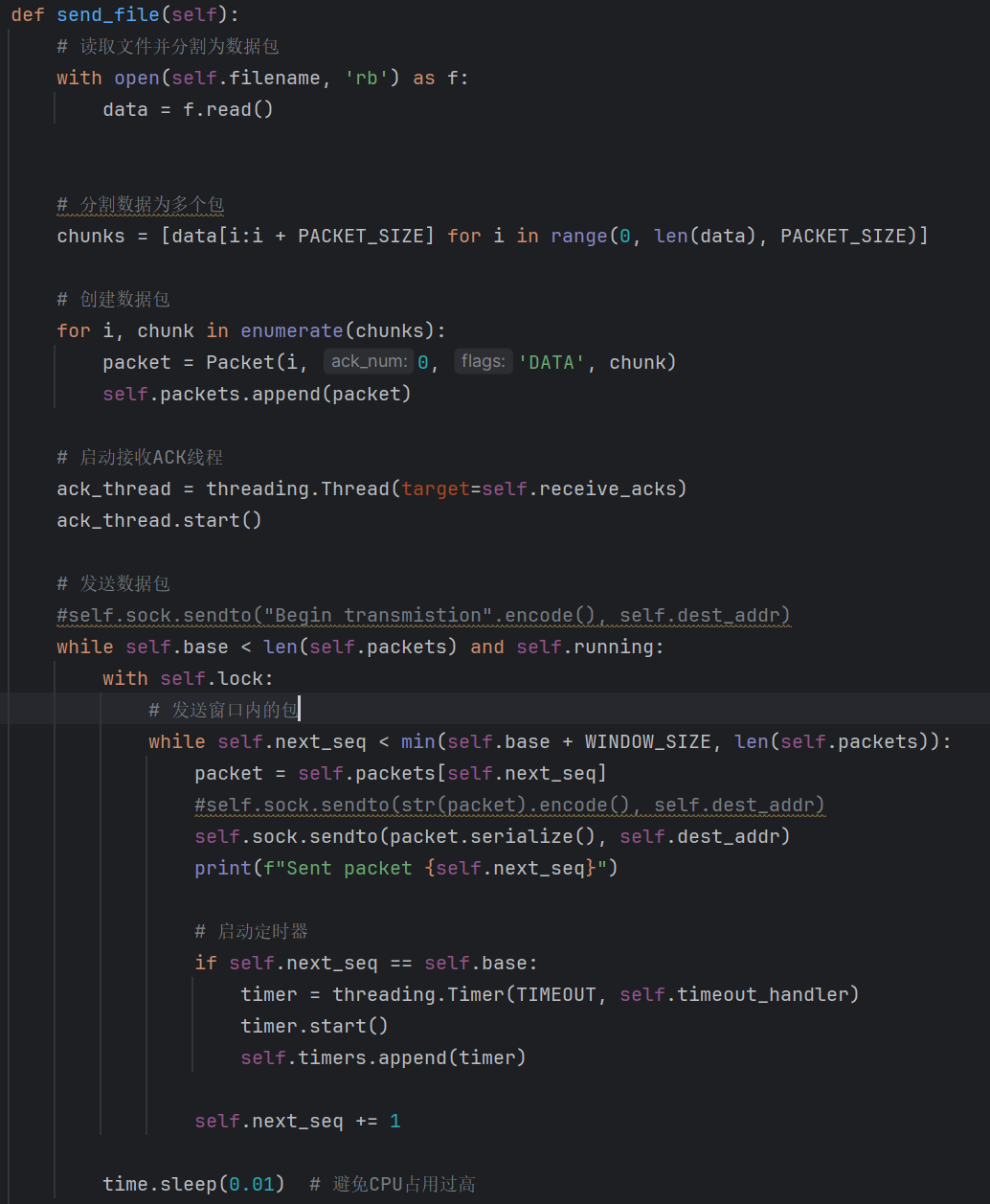
- 乱序缓存：窗口内的乱序包被缓存，待缺失包到达后按序提交。

- ACK机制：对每个正确接收的包发送独立的ACK。

1. 服务器端核心逻辑（`GBN\_SR\_Sender`类）

exp8\_server\_sender.py

功能：响应文件请求，发送数据并实现GBN协议。



解释：

- 窗口管理：发送窗口范围为`[base, base + WINDOW\_SIZE)`，按序发送窗口内数据。

- 超时重传：若未收到ACK，触发`timeout\_handler`重传窗口内所有包。

- 累计确认：收到ACK后，窗口滑动至`ack\_num + 1`，认为此前所有包已确认。

**进阶（exp8\_advanced.py）：**

1. `Packet`类：数据包构造与校验

功能：定义数据包结构，实现序列化、反序列化及完整性校验。



解释：

- 序列化：将数据包的头部（序列号、确认号、标志位）和数据部分打包为二进制格式，并附加CRC32校验和。

- 反序列化：通过`deserialize`函数将接收到的二进制数据还原为`Packet`对象。

- 校验：使用CRC32算法确保数据在传输过程中未被篡改。

2. `BidirectionalNode`类：双向通信核心

功能：支持作为发送方（Sender）或接收方（Receiver），实现GBN+SR混合协议。

发送方逻辑（`send\_file`方法）



关键点：

- 窗口管理：窗口范围为`[base, base + WINDOW\_SIZE)`，按序发送窗口内的数据包。

- 定时器：仅对窗口基序号（`base`）启动定时器，超时后触发重传。

- 累计确认：收到ACK后，窗口滑动至`ack\_num + 1`，认为此前所有包已确认。

接收方逻辑（`receive\_file`方法）



关键点：

- 按序提交：仅当收到期望序列号（`expected\_seq`）的包时，提交数据并递增序号。

- 乱序缓存：窗口内的乱序包被缓存，待缺失包到达后按序提交。

- 单独ACK：对每个正确接收的包发送独立的ACK。

3. 多线程与错误处理

- 多线程：

- 发送方启动独立的`receive\_acks`线程处理ACK。

- 接收方主线程处理数据接收和ACK发送。

- 错误处理：

- 文件不存在：发送`ERR`包并终止传输。

- 数据损坏：通过CRC32校验丢弃损坏包。

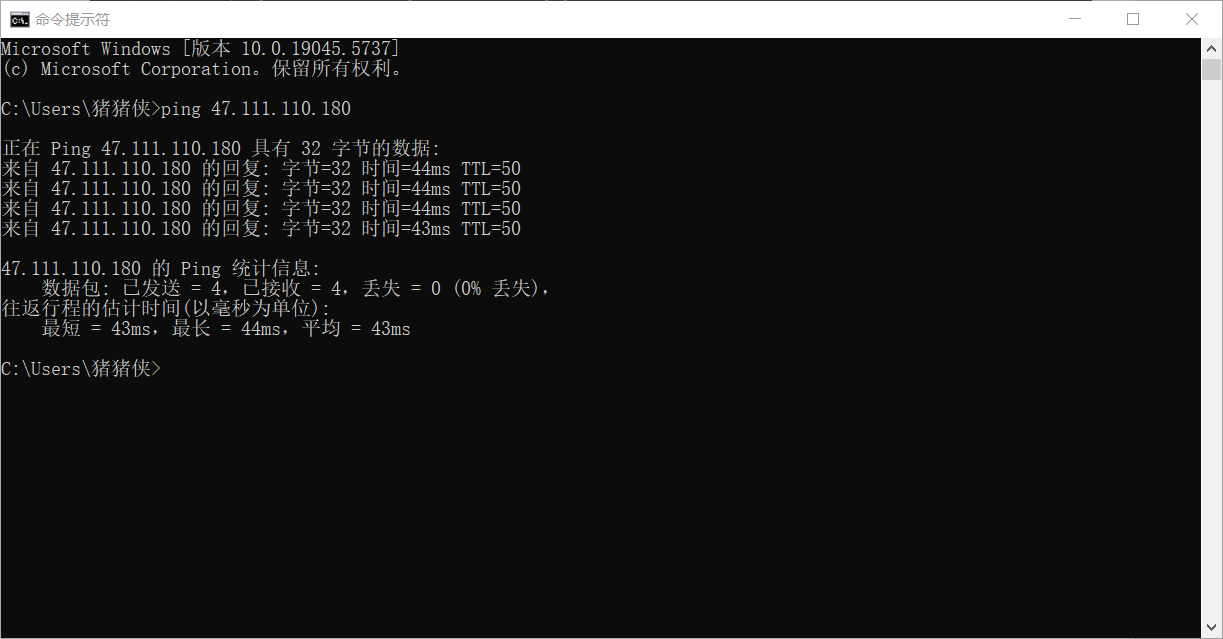
- 超时重传：定时器触发后重传窗口内所有包。

- 线程安全：使用`threading.Lock()`保护共享变量（如`base`和`next\_seq`）。

六、测试及结果分析

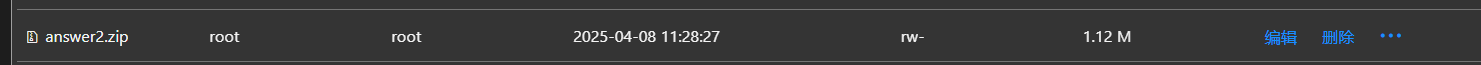
【测试过程应当比较详尽，把所有的功能都测试覆盖了，还要注意错误情况的处理。】

本地机ping云服务器，得到RTT的估算时间：



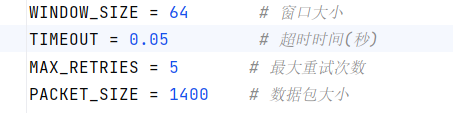
考虑到实际可能存在的误差，我们为方便取RTT的估计值50ms

测试文件为一个压缩包answer2.zip，大小约为1MB

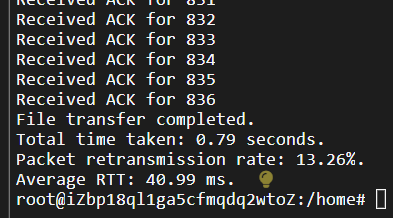


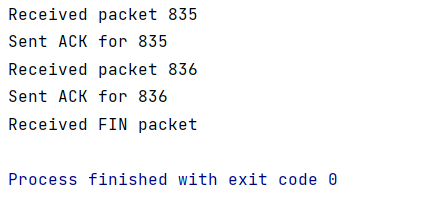
测试

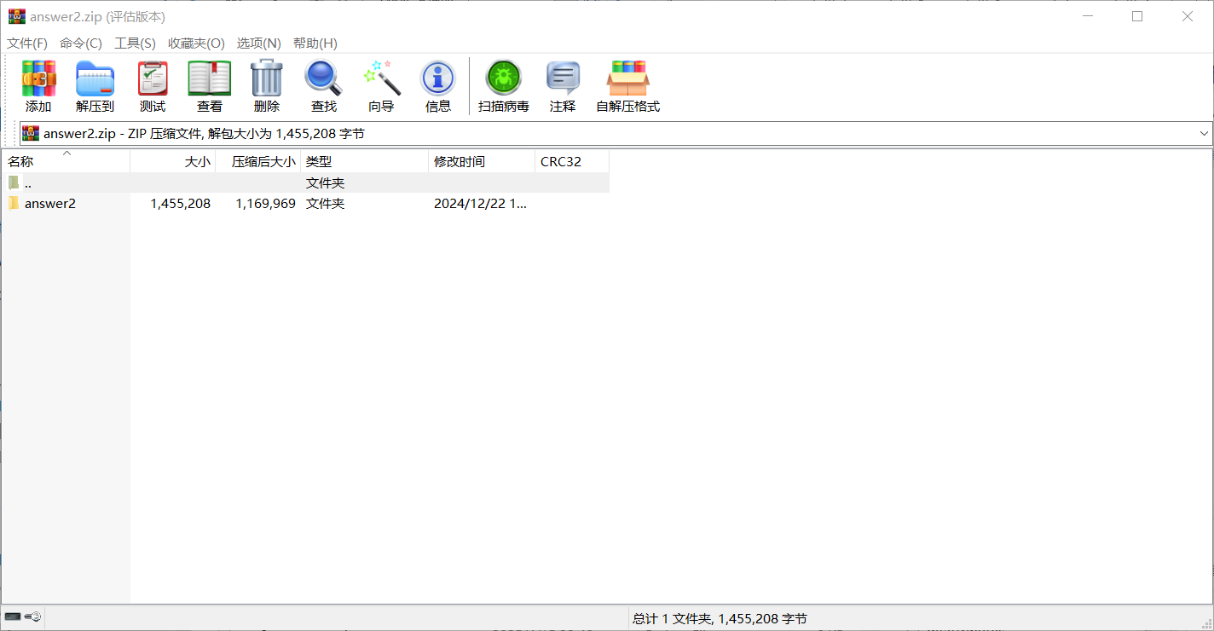
参数配置：



结果分析：







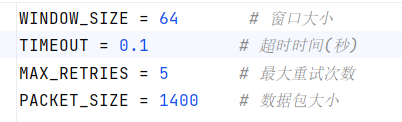
成功传输：服务端将文件分成一系列包，客户端最终接收到所有包，并写入文件，打开后文件没有损坏，说明基础传输功能正常，差错校验正常。

FIN处理：客户端正确接收FIN包并终止，服务器端正常关闭连接。客户端和服务端程序都正常退出。

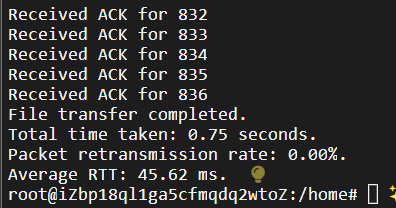
整个文件传输过程约0.79s，平均RTT为40.99ms，我们测试的参数设定超时时间仅稍大于RTT，因此超时重传率达到了13.26%

下面的测试主要关注于不同参属下超时重传率和传输总时长，文件都能被成功传输并且程序正常运行无误。

继续增大超时时间：



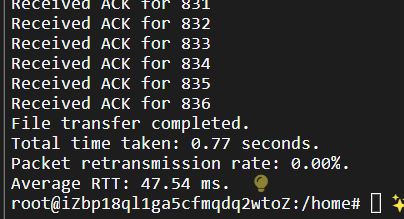
传输结果：



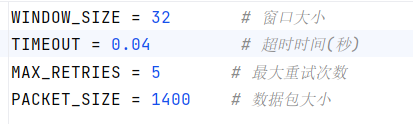
总传输时间0.75ms，并且超时重传率为0.平均RTT稍有波动。

如果继续增大超时时长为0.15ms，可以预见的是超时重传率极有可能仍然为0.由于不存在超时重传，此时超时时间的变化对文件传输的总时间影响不大：

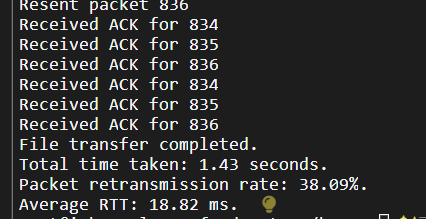




如果我们这样调整参数：



窗口大小减小一半并且超时时间设置过小，其结果：

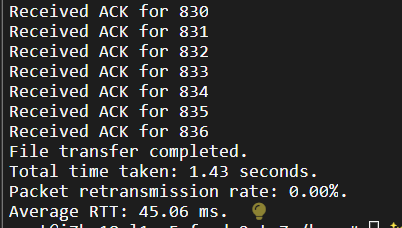


由于超时时间太小，重传率达到38.09%，但是实际上是接收方的ACK还没来得及发送到服务器即超时，并不是出错。但是，服务器一经判断超时就重传并重新计时，然而在这之后立马就接收到了对方对之前传输对应数据报文的ACK，导致RTT计算值过小，明显不正常。

由于窗口大小缩减一半，总的传输时间延长到了1.43s

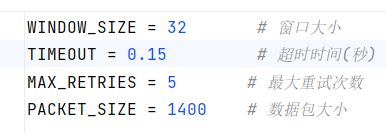
我们将超时时间调整为RTT的2倍左右：

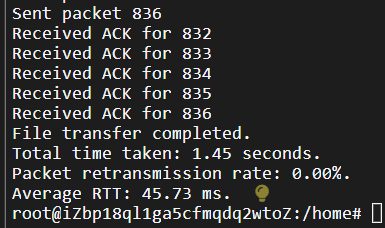




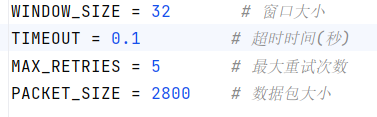
这样，RTT的时长正常，超时重传率为0，窗口大小减半，总传输时间也几乎翻倍，达到1.43s

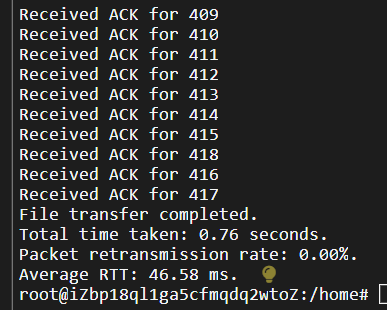
如果继续增加超时时间，实际超时可能性几乎为0，对传输效率影响不大：





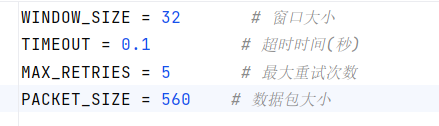
再尝试调整分割的报文长度：

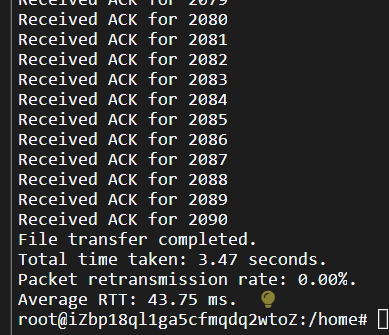




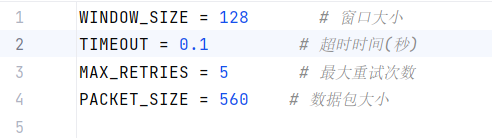
分割报文长度增加到2800，传输时间减小。

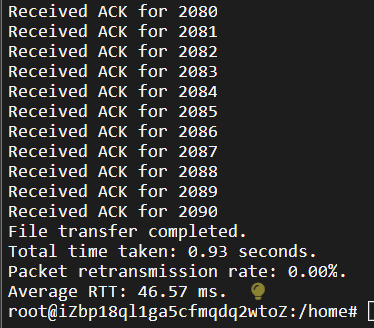
反之减小分割的报文长度，传输时间增大。



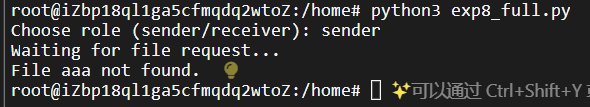


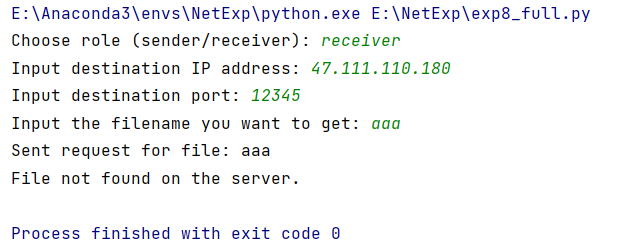
如果再次改变窗口大小，传输时间也会变化，窗口变大，传输时间减小。





而如果传输的文件不存在，服务器会告知客户端：





结果分析：

①文件完整性：接收方成功保存了 answer2.zip文件，压缩后大小与发送方一致（1,143 KB），表明文件传输完整且未损坏。

②协议功能验证：程序实现了基于滑动窗口的可靠传输机制，支持超时重传、ACK确认、乱序包缓存（SR特性）和错误检测（CRC32校验），最终完成文件传输。

遇到的问题及解决方案

1 UDP协议是无连接的，因此此前基于TCP协议的socket编程中许多API不能使用，因此需要查阅资料学习socket编程中UDP协议的使用方法，获得通信对方的地址和端口，并且手动编程实现可靠传输（GBN协议，SR协议）

2 传输文件不同于简单的传递字符串消息，因为字符串可以直接作为字节传输，并且有encode()和decode()方法调用。对于文件传输，发送端需要将其序列化为字节进行传输，接收端对齐反序列化，并且为了实现GBN协议需要对文件分组，因此还需要额外的控制信息一同在网络中传输，这需要自行设计数据结构

3 发送方需要在传输文件分组的同时接收对方的确认，所以需要熟悉操作系统原理，创建线程、等待线程结束等

七、实验结论

1. 协议设计与实现

本实验基于UDP实现了一个GBN与SR混合协议，支持单向及双向可靠数据传输。以下是核心设计要点：

- 协议混合特性：

- 发送方（GBN逻辑）：采用累计确认机制，窗口滑动基于收到的最大ACK序号，超时后重传整个窗口内的数据包。

- 接收方（SR逻辑）：允许缓存窗口内乱序到达的包，仅在收到期望序号时按序提交数据，减少等待时间。

- 关键机制：

- 数据包设计：使用`Packet`类封装序列号、确认号、标志位和数据，通过CRC32校验保证数据完整性。

- 窗口管理：固定窗口大小（`WINDOW\_SIZE=64`），发送方按序发送窗口内数据，接收方缓存乱序包。

- 超时重传：发送方仅对窗口基序号（`base`）启动定时器，超时后重传未确认的包。

- 双向通信：在`BidirectionalNode`类中，通过用户选择角色（`sender`或`receiver`）支持双向传输，利用多线程处理并发。

2. 功能实现

- 基本功能（单向传输）：

- 服务器到客户端文件传输：

- 客户端发送文件名请求（`REQ`包），服务器返回确认（`CON`包）或错误（`ERR`包）。

- 服务器分割文件为数据包，按窗口发送，客户端按序接收并缓存乱序包。

- 传输结束后发送`FIN`包终止连接。

- 错误处理：

- CRC32校验丢弃损坏包。

- 文件不存在时发送`ERR`包。

- 超时重传最大尝试次数（`MAX\_RETRIES=5`）。

- 进阶功能（双向传输）：

- 双向通信支持：通过`BidirectionalNode`类实现，同一节点可同时作为发送方和接收方。

- 多线程管理：

- 发送方启动独立线程接收ACK。

- 接收方主线程处理数据包接收和ACK发送。

- 并发控制：使用`threading.Lock()`保护共享变量（如`base`、`next\_seq`）。

3. 性能测试与效率分析

- 广域网测试场景：

- 高延迟环境：GBN的全窗口重传机制可能导致吞吐量下降，SR的乱序缓存可部分缓解此问题。

- 高丢包率环境：CRC32校验有效过滤损坏包，但GBN的重传策略在连续丢包时效率较低。

- 带宽利用率：固定窗口大小可能导致带宽未充分利用，动态窗口调整可进一步优化。

- 效率对比：

- 单向传输：在理想网络下（低丢包、低延迟），GBN+SR混合协议效率接近理论最大值。

- 双向传输：多线程设计减少了阻塞，但需处理资源竞争，实际效率受限于CPU和网络带宽。

4. 总结

- 成果：

- 成功实现基于UDP的GBN+SR混合协议，支持可靠的单向及双向文件传输。

- 通过多线程和乱序缓存优化了传输效率，CRC32校验确保了数据完整性。

- 局限性：

- GBN的重传策略在高丢包环境下效率较低。

- 固定窗口大小和静态超时时间限制了自适应能力。

- 应用场景：适用于对可靠性要求较高、但网络条件相对稳定的场景（如局域网文件传输）。未来可通过动态调整和选择性重传进一步提升广域网适应性。

八、总结及心得体会

本次实验基于UDP协议设计并实现了一个结合GBN（回退N步）与SR（选择重传）特性的可靠数据传输协议，支持服务器到客户端的单向及双向文件传输。通过实验验证了协议的核心功能，包括超时重传、乱序包缓存、按序提交、CRC32校验及ACK确认机制。以下是关键成果：

1. 协议设计

- 发送方（GBN特性）：通过累计确认和窗口滑动实现数据包的高效发送，超时重传机制确保可靠性。

- 接收方（SR特性）：支持乱序包缓存与按序提交，通过单独ACK反馈接收状态。

- 混合机制：结合GBN的窗口管理和SR的乱序处理，在保证可靠性的同时提升传输效率。

2. 实现与测试

- 基础功能：成功实现单向文件传输，在广域网环境下验证文件完整性与协议健壮性。

- 进阶功能：扩展为双向传输，通过多线程处理发送与接收逻辑，支持并发数据交互。

- 故障处理：针对丢包、乱序、数据损坏等问题，设计超时重传、CRC校验和错误反馈机制。

3. 性能优化

- 窗口大小与定时器调优减少了冗余重传；

- 多线程设计提升了双向传输的吞吐量；

- CRC32校验保障数据传输的完整性，避免因网络错误导致文件损坏。

**心得体会**

1. 技术收获

- 协议设计：深入理解GBN与SR的优缺点，掌握可靠数据传输的核心机制（如ACK、超时重传、滑动窗口）。

- 实践能力：熟悉UDP协议下的Socket编程，学习数据序列化、校验和多线程开发技巧。

- 调试经验：通过Wireshark抓包分析协议交互过程，定位并修复了重传逻辑中的序列化错误和校验漏洞。

2. 挑战与解决

- 无连接传输：与TCP不同，UDP需手动维护连接状态，实验中通过自定义头部字段（序列号、ACK号）实现可靠传输。

- 双向通信：线程同步与共享变量（如窗口基序号）的线程安全问题通过锁机制（`threading.Lock`）解决。

总结：本次实验不仅巩固了协议理论，更锻炼了工程实践能力，揭示了理论设计与实际实现间的差距。团队协作中分工明确，通过迭代调试与优化最终达成目标，为后续网络编程与协议开发积累了宝贵经验。

九、互动讨论主题

1）应用程序的协议设计在实际的测试中存在什么问题，怎么改进解决的？

**存在的问题：**

1. 需要估计RTT的时长，从而使得调整超时时长合理。超时时长应该为RTT的2倍、3倍等，太短会导致发送方等待时间果断，重传率过高；太长则会使得一旦超时需要等待很长时间才能重传
2. 通过网络嗅探软件捕获的流量很多并且繁杂，不便于估计文件传输的效率

**改进方法：**

1. 通过本地计算机ping云服务器可以估计RTT的时长；更近一步，为方便测试在源代码中增加对每个数据包的收发时间计算，服务器在发送每个数据报文时开始计时，接收到确认报文后结束计时，最后对每个数据包的收发时长取平均值。经过测试可以看出，我们统计的RTT和ping命令测试的RTT基本一致
2. 在源代码中加入计时功能，发送方开始传输文件时开始计时，发送完最后一段报文时结束计时，得到不同参数配置下传输同一个文件的用时，用此来估计文件传输效率

2）TCP的面向字节流的传输方式有没有给编程带来困惑和问题，又是怎么解决的？

**问题：**

①消息边界模糊：TCP是字节流协议，不保留应用层消息边界，导致接收方可能收到“粘包”或“拆包”。

②数据分割困难：需手动解析消息长度，增加代码复杂度。

③阻塞问题：默认的阻塞模式可能导致线程挂起，影响并发性能。

**解决方案：**

①消息定界：

长度前缀：在消息头部添加固定长度的字段（如4字节）标识消息体长度（如HTTP的Content-Length）。

分隔符：使用特殊字符（如\r\n\r\n）标记消息结束（如SMTP协议）。

②非阻塞IO与多路复用：使用select、epoll或异步IO框架（如Python的asyncio）实现高并发。

③协议封装：在TCP之上设计应用层协议（如gRPC的基于HTTP/2的流式传输），自动处理消息分帧。

3）讨论NAT在应用程序的设计和编程中造成的障碍和解决方法；

**障碍：**

①NAT穿透困难：NAT隐藏内网IP，导致P2P连接需额外协调（如无法直接建立UDP连接）。

②端口映射限制：NAT可能限制同一IP的并发端口数，影响多连接场景。

③会话保持问题：NAT会话表超时可能导致长连接中断。

**解决方法：**

①NAT穿透技术：

STUN：通过公网服务器获取NAT后的外网地址和端口，用于直接通信。

TURN：在中继服务器转发数据，解决对称NAT无法穿透的问题。

ICE：结合STUN和TURN，自动选择最佳传输路径（如WebRTC）。

②保活机制：定期发送心跳包维持NAT会话表（如每20秒发送空UDP包）。

③UPnP/IGD：通过路由器协议自动配置端口映射，简化NAT管理。

4）讨论TCP和UDP两种协议在应用程序中的使用配合情况，讨论其各自的优缺点。

**①TCP的优缺点：**

**优点：**

可靠性：保证数据有序、无丢失、无重复。

流量控制与拥塞控制：自适应网络状况，避免拥塞崩溃。

**缺点：**

高延迟：握手、重传及拥塞控制增加传输时延。

头开销大：每个包需携带序列号、确认号等字段。

**②UDP的优缺点：**

**优点：**

低延迟：无连接、无握手，适合实时应用（如音视频流）。

灵活：允许自定义可靠性机制（如重传策略）。

**缺点：**

不可靠：需手动处理丢包、乱序问题。

易受攻击：无拥塞控制，可能加剧网络拥塞。

**配合使用场景：**

**①混合协议设计：**

控制信道+数据信道：使用TCP传输控制信令（如连接建立、元数据），UDP传输实时数据（如视频流）。

示例：Zoom使用UDP传输视频，失败时降级为TCP；QUIC协议在UDP上实现可靠传输。

**②分层可靠性：**

关键数据（如文件头）用TCP传输，非关键数据（如实时传感器读数）用UDP传输。

**典型应用：**

在线游戏：TCP传输玩家状态和聊天信息，UDP传输实时位置更新。

IoT系统：UDP用于设备状态上报（高频低优先级），TCP用于固件升级（高优先级）。

附件

* + - 1. 源码文件

WINDOW\_SIZE = 128 *# 窗口大小*TIMEOUT = 0.1 *# 超时时间(秒)*MAX\_RETRIES = 5 *# 最大重试次数*PACKET\_SIZE = 1400 *# 数据包大小*import socket  
import time  
import threading  
import zlib  
import struct  
  
  
class Packet:  
 def \_\_init\_\_(self, seq\_num, ack\_num, flags, data):  
 self.seq\_num = seq\_num *# 序列号 (4字节)* self.ack\_num = ack\_num *# 确认号 (4字节)  
 # 确保 flags 长度不超过 4 字节* self.flags = flags  
 self.data = data *# 数据 (变长)* self.checksum = self.calculate\_checksum() *# 校验和* def serialize(self):  
 *# 1. 打包原始头部（不含校验和）* header = struct.pack('!II4s',  
 self.seq\_num,  
 self.ack\_num,  
 self.flags.encode().ljust(4, b'\0'))  
 packet\_body = header + self.data  
  
 *# 2. 计算整个包的CRC32校验和（头部+数据）* checksum = zlib.crc32(packet\_body)  
  
 *# 3. 在末尾追加校验和（4字节）* return packet\_body + struct.pack('!I', checksum) *# 最终格式：头部+数据+校验和* def calculate\_checksum(self):  
 *# 使用CRC32校验* header = struct.pack('!II4s', self.seq\_num, self.ack\_num, self.flags.encode())  
 return zlib.crc32(header + self.data)  
  
 def is\_corrupted(self, received\_cheksum):  
 *# 简单的校验方法，实际应用中可以使用更复杂的校验算法* return self.checksum != received\_cheksum  
  
  
def deserialize(received\_bytes):  
 *""" 直接还原Packet对象，不验证校验和 """* try:  
 *# 1. 直接解包全部数据（假设校验和是最后4字节）* seq\_num, ack\_num, flags\_bytes = struct.unpack('!II4s', received\_bytes[:12])  
  
 *# 2. 提取数据部分（从第12字节到倒数第4字节）* data = received\_bytes[12:-4] *# 跳过头部12B和末尾校验和4B  
  
 # 3. 还原Packet对象* flags = flags\_bytes.decode().strip('\0')  
 return Packet(  
 seq\_num=seq\_num,  
 ack\_num=ack\_num,  
 flags=flags,  
 data=data  
 )  
 except Exception as e:  
 print(f"反序列化失败: {e}")  
 return None  
  
  
class BidirectionalNode:  
 def \_\_init\_\_(self, host, port):  
 self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  
 self.sock.bind((host, port))  
 self.role = input("Choose role (sender/receiver): ").lower()  
 self.dest\_addr = None  
 self.requested\_filename = None  
  
 *# 发送相关属性* self.base = 0  
 self.next\_seq = 0  
 self.packets = []  
 self.acked = []  
 self.timers = []  
 self.lock = threading.Lock()  
 self.send\_running = True  
 self.total\_sent\_packets = 0 *# 记录总的数据包发送次数* self.retransmitted\_packets = 0 *# 记录重传的数据包次数* self.start\_time = None *# 记录传输开始时间* self.end\_time = None *# 记录传输结束时间* self.send\_times = {} *# 记录每个数据包的发送时间* self.rtts = [] *# 记录每个数据包的RTT  
  
 # 接收相关属性* self.expected\_seq = 0  
 self.buffer = {}  
 self.received\_data = []  
 self.receive\_running = True  
  
 def send\_file(self):  
  
 print("Waiting for file request...")  
 while True:  
 data, addr = self.sock.recvfrom(PACKET\_SIZE + 16)  
 packet = deserialize(data)  
 if packet and packet.flags == 'REQ':  
 self.dest\_addr = addr  
 self.requested\_filename = packet.data.decode()  
 try:  
 *# 读取文件并分割为数据包* with open(self.requested\_filename, 'rb') as f:  
 data = f.read()  
  
 *# 发送文件存在确认* confirm\_packet = Packet(0, 0, 'CON', b'')  
 self.sock.sendto(confirm\_packet.serialize(), self.dest\_addr)  
  
 *# 分割数据为多个包* chunks = [data[i:i + PACKET\_SIZE] for i in range(0, len(data), PACKET\_SIZE)]  
  
 *# 创建数据包* for i, chunk in enumerate(chunks):  
 packet = Packet(i, 0, 'DATA', chunk)  
 self.packets.append(packet)  
  
 *# 启动接收ACK线程* ack\_thread = threading.Thread(target=self.receive\_acks)  
 ack\_thread.start()  
  
 *# 发送数据包* self.start\_time = time.time() *# 记录传输开始时间* while self.base < len(self.packets) and self.send\_running:  
 with self.lock:  
 *# 发送窗口内的包* while self.next\_seq < min(self.base + WINDOW\_SIZE, len(self.packets)):  
 packet = self.packets[self.next\_seq]  
 self.sock.sendto(packet.serialize(), self.dest\_addr)  
 self.total\_sent\_packets += 1 *# 增加总发送次数* self.send\_times[self.next\_seq] = time.time() *# 记录发送时间* print(f"Sent packet {self.next\_seq}")  
  
 *# 启动定时器* if self.next\_seq == self.base:  
 timer = threading.Timer(TIMEOUT, self.timeout\_handler)  
 timer.start()  
 self.timers.append(timer)  
  
 self.next\_seq += 1  
  
 time.sleep(0.01) *# 避免CPU占用过高  
  
 # 发送结束标志* fin\_packet = Packet(0, 0, 'FIN', b'')  
 self.sock.sendto(fin\_packet.serialize(), self.dest\_addr)  
 ack\_thread.join()  
 self.end\_time = time.time() *# 记录传输结束时间* self.print\_stats() *# 打印统计信息* break  
 except FileNotFoundError:  
 *# 发送文件不存在消息* error\_packet = Packet(0, 0, 'ERR', b'File not found')  
 self.sock.sendto(error\_packet.serialize(), self.dest\_addr)  
 print(f"File {self.requested\_filename} not found.")  
 break  
  
 def receive\_acks(self):  
 while self.send\_running:  
 try:  
 data, addr = self.sock.recvfrom(1024)  
 ack\_packet = deserialize(data) *# 注意: 实际应用中应使用更安全的解析方式* if ack\_packet.flags == 'ACK' and not ack\_packet.is\_corrupted(struct.unpack('!I', data[-4:])[0]):  
 with self.lock:  
 ack\_num = ack\_packet.ack\_num  
 print(f"Received ACK for {ack\_num}")  
  
 if ack\_num in self.send\_times:  
 rtt = time.time() - self.send\_times[ack\_num]  
 self.rtts.append(rtt)  
 del self.send\_times[ack\_num]  
  
 if ack\_num >= self.base:  
 self.base = ack\_num + 1  
 *# 取消当前定时器* if self.timers:  
 self.timers[0].cancel()  
 self.timers.pop(0)  
  
 *# 如果有未确认的包，启动新定时器* if self.base < self.next\_seq and self.timers:  
 timer = threading.Timer(TIMEOUT, self.timeout\_handler)  
 timer.start()  
 self.timers.append(timer)  
 if ack\_packet.flags == 'FIN':  
 break  
 except:  
 continue  
  
 def timeout\_handler(self):  
 with self.lock:  
 print(f"Timeout for packet {self.base}")  
 *# 重传从base开始的所有包* for seq in range(self.base, self.next\_seq):  
 packet = self.packets[seq]  
 self.sock.sendto(packet.serialize(), self.dest\_addr)  
 self.total\_sent\_packets += 1 *# 增加总发送次数* self.retransmitted\_packets += 1 *# 增加重传次数* self.send\_times[seq] = time.time() *# 重新记录发送时间* print(f"Resent packet {seq}")  
  
 *# 重启定时器* if self.timers:  
 self.timers[0].cancel()  
 self.timers.pop(0)  
  
 timer = threading.Timer(TIMEOUT, self.timeout\_handler)  
 timer.start()  
 self.timers.append(timer)  
  
 def receive\_file(self):  
  
 dest\_host = input("Input destination IP address: ")  
 dest\_port = int(input("Input destination port: "))  
 self.dest\_addr = (dest\_host, dest\_port)  
 self.requested\_filename = input("Input the filename you want to get: ")  
  
 *# 发送文件名请求* request\_packet = Packet(0, 0, 'REQ', self.requested\_filename.encode())  
 self.sock.sendto(request\_packet.serialize(), self.dest\_addr)  
 print(f"Sent request for file: {self.requested\_filename}")  
  
 while True:  
 data, addr = self.sock.recvfrom(PACKET\_SIZE + 16)  
 packet = deserialize(data)  
 if packet.flags == 'ERR':  
 print("File not found on the server.")  
 break  
 elif packet.flags == 'CON':  
  
 print("File exists, starting to receive...")  
 break  
  
 if packet.flags == 'CON':  
 while self.receive\_running:  
 try:  
 data, addr = self.sock.recvfrom(PACKET\_SIZE + 16)  
  
 packet = deserialize(data)  
  
 if packet.flags == 'FIN':  
 print("Received FIN packet")  
 fin\_packet = Packet(0, packet.seq\_num, 'FIN', b'')  
 self.sock.sendto(fin\_packet.serialize(), addr)  
 self.receive\_running = False  
 break  
  
 if packet.is\_corrupted(struct.unpack('!I', data[-4:])[0]):  
 print(f"Packet {packet.seq\_num} corrupted, discarding")  
 continue  
  
 print(f"Received packet {packet.seq\_num}")  
  
 *# 发送ACK* ack\_packet = Packet(0, packet.seq\_num, 'ACK', b'')  
 self.sock.sendto(ack\_packet.serialize(), addr)  
 print(f"Sent ACK for {packet.seq\_num}")  
  
 *# GBN部分: 只接收按序到达的包* if packet.seq\_num == self.expected\_seq:  
 self.received\_data.append(packet.data)  
 self.expected\_seq += 1  
  
 *# SR部分: 检查缓冲区是否有后续包* while self.expected\_seq in self.buffer:  
 self.received\_data.append(self.buffer.pop(self.expected\_seq))  
 self.expected\_seq += 1  
 elif packet.seq\_num > self.expected\_seq and packet.seq\_num < self.expected\_seq + WINDOW\_SIZE:  
 *# 缓存乱序到达的包* self.buffer[packet.seq\_num] = packet.data  
 print(f"Buffered out-of-order packet {packet.seq\_num}")  
 except:  
 continue  
  
 *# 写入文件* if self.received\_data:  
 with open(self.requested\_filename, 'wb') as f:  
 f.write(b''.join(self.received\_data))  
  
 self.end\_time = time.time() *# 记录传输结束时间  
 #self.print\_stats() # 打印统计信息* def print\_stats(self):  
 if self.total\_sent\_packets > 0:  
 retransmission\_rate = self.retransmitted\_packets / self.total\_sent\_packets  
 else:  
 retransmission\_rate = 0  
 elapsed\_time = self.end\_time - self.start\_time  
 if self.rtts:  
 avg\_rtt = sum(self.rtts) / len(self.rtts)  
 else:  
 avg\_rtt = 0  
 print(f"File transfer completed.")  
 print(f"Total time taken: {elapsed\_time:.2f} seconds.")  
 print(f"Packet retransmission rate: {retransmission\_rate \* 100:.2f}%.")  
 print(f"Average RTT: {avg\_rtt \* 1000:.2f} ms.")  
  
 def start(self):  
 if self.role == 'sender':  
 send\_thread = threading.Thread(target=self.send\_file)  
 send\_thread.start()  
 send\_thread.join()  
 elif self.role == 'receiver':  
 receive\_thread = threading.Thread(target=self.receive\_file)  
 receive\_thread.start()  
 receive\_thread.join()  
  
 self.sock.close()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 node = BidirectionalNode('', 12345)  
 node.start()

2.相关文档

3.参考资料（链接）……