

**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、GBN协议和SR协议的设计与实现 | | | | | |
| 姓名 | 许坤彬 | | 院系 | 计算学部 | | |
| 班级 | 2203103 | | 学号 | 2022113586 | | |
| 任课教师 | 聂兰顺 | | 指导教师 | 聂兰顺 | | |
| 实验地点 | G001 | | 实验时间 | 2024/10/17 | | |
| 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： |
| 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： |
| 1. 停等协议  （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。  2. GBN/SR协议  （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。 |
| 实验过程： |
| 1. 停等协议的设计与实现  1.1 停等协议的数据分组格式与确认分组格式  停等协议的数据分组主要需要包含两部分的内容，一方面是当前数据包的序列号，另一方面是数据包的内容。数据包的内容采用utf-8的格式进行编码。   |  |  | | --- | --- | | seq | data\_packet |   其中，seq为1位（0或1），数据包大小设置为1024个字节。  确认分组的格式同样由两个部分构成，一方面是ACK的确认信息，另一方面则是序列号（0或1）。  而在双向传输的实现中，则需要对数据分组的格式进行进一步的修改。双向传输意味着服务端在发送数据包的同时需要对先前接受的数据包发送ACK，而接受方在向服务端发送ACK之后同时需要发送向服务端发送的数据包。因此在双向传输中，数据分组格式变为如下的部分：   |  |  |  |  | | --- | --- | --- | --- | | seq | data\_packet | ACK | Ack\_seq |   1.2 停等协议的确认分组格式   |  |  | | --- | --- | | ACK | seq |   其中，在停等协议中，序列号只有0或1。将两种信息结合起来后使用utf-8进行编码而得到ACK的信息。  在双向传输的部分，接受方在向服务端发送ACK之后同时需要发送向服务端发送的数据包。因此在双向传输中，确认分组格式变为如下的部分：   |  |  |  |  | | --- | --- | --- | --- | | ACK | ACK\_seq | seq | data\_packet |   1.1 基于UDP的简单停等协议实现  UDP的停等协议主要分为server端和client端两部分，数据报格式主要设计为“state-data\_packet”，这里的state是用来作为序列区分的序列号，停等协议的实现可以用状态机来进行描述。    发送的同时会为发送的每一个数据包设定定时器，当超时重新发送当前状态的数据包。发送之后进入等待ACK的状态，当收到ACK且ACK状态为1时，发送下一个数据包并且反转状态为0，否则重发数据包。  客户端的实现主要需要注意的是判断所接受的数据包序列号是否与自己想要的一致，如果一致则发送正确的ACK，否则不做任何处理，也不接受数据包，等待服务端超时后重新传输。客户端主要的流程可以使用下面的FSM进行描述：    1.2 典型简单停等协议的交互过程  在停等协议中，发送方在发送每个数据帧后需等待接收方的确认（ACK）才能发送下一帧。发送方在发送数据帧后启动计时器防止数据丢失。接收方收到帧后进行校验，确认帧的序号是否正确。如果帧无误且序号正确，接收方提交数据并发送ACK；如果帧有误或序号不对，接收方丢弃帧且不发送ACK。发送方等待ACK，如果超时未收到ACK则重传数据帧。一旦收到ACK确认，发送方继续发送下一个帧。此过程反复进行，直到所有数据传输完成。    1.4 数据分组丢失的验证模拟  使用随机数函数进行丢包的模拟，定义一个0~1之间的随机数，当随机数<0.2时只在控制台输出丢包，而不发送数据包。通过概率的设定，可以模拟不同严重程度的网络拥塞所造成的丢包，进而对协议的准确性和稳定性进行测试。  # 模拟丢包函数  def simulate\_packet\_loss():      return random.random() < 0.2  # 20%的概率丢包  1.4 程序主要的实现函数及其作用  （1）基础停等协议的实现  在基础的停等协议实现中，主要包含了三种函数，分别是simulate\_packet\_loss，udp\_server和udp\_client。其中，simulate\_packet\_loss函数作为丢包函数的模拟，已经在1.4的部分做了讲述，接下来主要侧重于对另外两个函数的描述。  udp\_server中首先创建socket对象，并将创建的socket绑定在服务器的IP和端口上。随后对数据包进行读取，为每一个数据包按照发送分组的格式进行封装。为了模拟方便，数据包列表主要通过python列表的方式进行模拟，server端发送的条件判断主要依赖于判断包列表是否为空（即判断长度是否为零），每一次成功发送并且收到ACK之后都会将列表中的第0个数据包弹出。同时，在每个包发送之后，会通过socket的settimeout方法来设定超时时间。之后等待ACK，若超时之前，接收到了来自client的ACK，则将序列号反转，并进入下一个数据包的发送。若超时没有收到ACK或者接收到了序列号错误的ACK，那么重发当前的数据包。  而在udp\_client中，在完成socket的创建和绑定之后，进入到接受状态，接受并且判断所接受的包是否符合接受方想要的序列号。如果是则发送ACK，否则直接将包丢弃。  （2）双向传输的实现  在双向传输的实现当中主要用到的函数名与基础停等协议相同，这里的区别在于为了能够让协议为外部调用，需要对函数的接口进行一定的修改。其中，udp\_server的函数接口变为udp\_server(sever\_ip, sever\_port, client\_ip, client\_port, data\_packets)，udp\_client的函数接口变为udp\_client(client\_ip, client\_port)，同时为了能够实现双向传输，通过线程调用的方式对协议中的udp\_server和udp\_client在main中进行调用。  （3）cs架构应用的实现  在cs架构应用的实现上仍然主要实现的是udp\_server和udp\_client两个函数，但在先前的基础上增加了文件读写的部分，通过文件读写将内容读入，再按照缓冲大小进行数据分组的封装。随后使用协议进行传输。  1.5 UDP编程的主要特点  （1）无连接：UDP是无连接协议，不需要建立连接即可发送数据。这意味着发送方和接收方之间无需握手，数据包可以立即发送，传输效率较高。  （2）不可靠传输：UDP不保证数据的传输可靠性，也不确保数据包的顺序或送达。为此，停等协议会在应用层通过超时和重传机制来确保数据可靠传输。  （3）面向消息：UDP是面向数据报的协议，发送方每次发送的数据作为一个独立的数据报发送，接收方每次接收一个完整的数据报。因此，UDP不合并或拆分数据报，开发者必须自行处理数据报的大小。  （4）轻量级：UDP没有像TCP那样复杂的连接管理、流控制和拥塞控制机制，因而开销较小，适合对实时性要求高但对可靠性要求不高的场景。  1.6 详细注释源程序  （1）基础停等协议  server.py  import socket  import time  import random  SERVER\_IP = '127.0.0.1'  SERVER\_PORT = 12345  CLIENT\_IP = "127.0.0.1"  CLIENT\_PORT = 12346  BUFFER\_SIZE = 1024  TIMEOUT = 2  # 超时时间（秒）  # 模拟丢包函数  def simulate\_packet\_loss():      return random.random() < 0.2  # 20%的概率丢包  def udp\_server():      server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      server\_socket.bind((SERVER\_IP, SERVER\_PORT))      print(f"服务器启动，等待客户端连接...")      # 数据包的内容      data\_packets = ["数据包1:哈", "数据包2：尔", "数据包3：滨", "数据包4：工", "数据包5：业",  "数据包6：大", "数据包7：学"]      # current\_packet = 0  # 当前发送的包序号      state = 0           # 初始序列号为0      while len(data\_packets) > 0 :          # 发送数据包格式：状态-          packet = f"{state}-{data\_packets[0]}".encode('utf-8')            # 模拟丢包          if simulate\_packet\_loss():              print(f"模拟丢失：{data\_packets[0]}")              continue          server\_socket.sendto(packet, (CLIENT\_IP, CLIENT\_PORT))          print(f"发送数据包：{data\_packets[0]}")          # 设置超时接收          server\_socket.settimeout(TIMEOUT)          try:              ack, client\_address = server\_socket.recvfrom(BUFFER\_SIZE)              ack = ack.decode('utf-8')                if ack == f"ACK{state}":                  print(f"收到 ACK：{ack}")                  data\_packets.pop(0) # 弹出待发送队列的列首                  if state == 0 :                      state = 1       # 收到ACK反转状态                  elif state == 1:                      state = 0              else:                  print(f"收到错误的 ACK：{ack}，重发当前数据包。")            except socket.timeout:              print(f"超时未收到 ACK{state}，重发数据包。")  if \_\_name\_\_ == "\_\_main\_\_":      udp\_server()  client.py  import socket  CLIENT\_IP = '127.0.0.1'  CLIENT\_PORT = 12346  BUFFER\_SIZE = 1024  def udp\_client():      client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      client\_socket.bind((CLIENT\_IP, CLIENT\_PORT))      print("客户端启动，等待数据接收...")      expected\_packet = 0      try:          while True:              data, server\_address = client\_socket.recvfrom(BUFFER\_SIZE)              data = data.decode('utf-8')              sequence\_number, packet\_content = data.split('-')              if int(sequence\_number) == expected\_packet:                  print(f"收到正确的包：{packet\_content}")                  ack = f"ACK{expected\_packet}".encode('utf-8')                  client\_socket.sendto(ack, server\_address)                  if expected\_packet == 1:                      expected\_packet = 0                  elif expected\_packet == 0:                      expected\_packet = 1              else:                  print(f"接收到错误包，期望包号：{expected\_packet}")        except KeyboardInterrupt:          print("\n客户端已被中断，正在关闭...")      finally:          client\_socket.close()          print("客户端已关闭。")  if \_\_name\_\_ == "\_\_main\_\_":      udp\_client()  （2）双向传输的实现  run\_wait.py  import wait as wait  import threading  SERVER\_IP = '127.0.0.1'  SERVER\_SEND\_PORT = 12345  SERVER\_RECV\_PORT = 12347  CLIENT\_IP = "127.0.0.1"  CLIENT\_SEND\_PORT = 12346  CLIENT\_RECV\_PORT = 12348  # 模拟要传输的数据  data\_packet1 = ["数据包1:哈", "数据包2：尔", "数据包3：滨", "数据包4：工", "数据包5：业",  "数据包6：大", "数据包7：学"]  data\_packet2 = ["数据包1:规", "数据包2：格", "数据包3：严", "数据包4：格", "数据包5：功",  "数据包6：夫", "数据包7：到", "数据包8：家"]  Thread\_Server\_send = threading.Thread(target=wait.udp\_server, args=(SERVER\_IP, SERVER\_SEND\_PORT, CLIENT\_IP, CLIENT\_RECV\_PORT, data\_packet1, ))  Thread\_Server\_recv = threading.Thread(target=wait.udp\_client, args=(SERVER\_IP, SERVER\_RECV\_PORT, ))  Thread\_Client\_send = threading.Thread(target=wait.udp\_server, args=(CLIENT\_IP, CLIENT\_SEND\_PORT, SERVER\_IP, SERVER\_RECV\_PORT, data\_packet2, ))  Thread\_Client\_recv = threading.Thread(target=wait.udp\_client, args=(CLIENT\_IP, CLIENT\_RECV\_PORT, ))  Thread\_Server\_send.start()  Thread\_Client\_recv.start()  Thread\_Client\_send.start()  Thread\_Server\_recv.start()  wait.py  import socket  import time  import random  BUFFER\_SIZE = 1024  TIMEOUT = 2  # 超时时间（秒）  # 模拟丢包函数  def simulate\_packet\_loss():      return random.random() < 0.2  # 20%的概率丢包  def udp\_server(sever\_ip, sever\_port, client\_ip, client\_port, data\_packets):      server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      server\_socket.bind((sever\_ip, sever\_port))      print(f"服务器启动，等待客户端连接...")      # 数据包的内容      # data\_packets = ["数据包1:小", "数据包2：狗", "数据包3：汪", "数据包4：汪", "数据包5：队"]      # current\_packet = 0  # 当前发送的包序号      state = 0           # 初始序列号为0      while len(data\_packets) > 0 :          # 发送数据包格式：状态-数据          packet = f"{state}-{data\_packets[0]}".encode('utf-8')            # 模拟丢包          if simulate\_packet\_loss():              print(f"模拟丢失：{data\_packets[0]}")              continue          server\_socket.sendto(packet, (client\_ip, client\_port))          print(f"发送数据包：{data\_packets[0]}")          # 设置超时接收          server\_socket.settimeout(TIMEOUT)          try:              ack, client\_address = server\_socket.recvfrom(BUFFER\_SIZE)              ack = ack.decode('utf-8')                if ack == f"ACK{state}":                  print(f"收到 ACK：{ack}")                  data\_packets.pop(0) # 弹出待发送队列的列首                  if state == 0 :                      state = 1       # 收到ACK反转状态                  elif state == 1:                      state = 0              else:                  print(f"收到错误的 ACK：{ack}，重发当前数据包。")            except socket.timeout:              print(f"超时未收到 ACK{state}，重发数据包。")      def udp\_client(client\_ip, client\_port):      client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      client\_socket.bind((client\_ip, client\_port))      print("客户端启动，等待数据接收...")      expected\_packet = 0      try:          while True:              data, server\_address = client\_socket.recvfrom(BUFFER\_SIZE)              data = data.decode('utf-8')              sequence\_number, packet\_content = data.split('-')              if int(sequence\_number) == expected\_packet:                  print(f"收到正确的包：{packet\_content}")                  ack = f"ACK{expected\_packet}".encode('utf-8')                  client\_socket.sendto(ack, server\_address)                  if expected\_packet == 1:                      expected\_packet = 0                  elif expected\_packet == 0:                      expected\_packet = 1              else:                  print(f"接收到错误包，期望包号：{expected\_packet}")        except KeyboardInterrupt:          print("\n客户端已被中断，正在关闭...")      finally:          client\_socket.close()          print("客户端已关闭。")  (3) c/s架构应用的实现  server\_cs.py  import socket  import os  import random  SERVER\_IP = '127.0.0.1'  SERVER\_PORT = 12345  CLIENT\_IP = '127.0.0.1'  CLIENT\_PORT = 12346  BUFFER\_SIZE = 1024  TIMEOUT = 2  # 超时时间（秒）  FILE\_PATH = 'server\_file.txt'  # 要发送的文件  # 模拟丢包函数  def simulate\_packet\_loss():      return random.random() < 0.2  # 20%的概率丢包  def udp\_server():      # 创建socket套接字，并且绑定在本地端口上      server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      server\_socket.bind((SERVER\_IP, SERVER\_PORT))      print(f"服务器启动，等待客户端连接...")      try:          # 首先打开需要传递的文件          with open(FILE\_PATH, 'rb') as file:              # 获取文件大小              file\_size = os.path.getsize(FILE\_PATH)              # 计算文件需要多少数据包              num\_packets = (file\_size // BUFFER\_SIZE) + 1              state = 0  # 初始序列号为0                # 根据文件读取结果确定什么时候结束循环              for i in range(num\_packets):                  # 读取文件的一块数据并且分别进行封装                  file\_chunk = file.read(BUFFER\_SIZE)                  packet = f"{state}-".encode('utf-8') + file\_chunk                  # # 模拟丢包                  # if simulate\_packet\_loss():                  #     print(f"模拟丢失：数据包{state}")                  #     continue                  # 向客户端发送对应的数据包                  server\_socket.sendto(packet, (CLIENT\_IP, CLIENT\_PORT))                  print(f"发送数据包{state}")                  # 设置超时接收                  server\_socket.settimeout(TIMEOUT)                  try:                      ack, client\_address = server\_socket.recvfrom(BUFFER\_SIZE)                      ack = ack.decode('utf-8')                      if ack == f"ACK{state}":                          print(f"收到 ACK：{ack}")                          state = 1 - state  # 切换状态                      else:                          print(f"收到错误的 ACK：{ack}，重发当前数据包。")                  except socket.timeout:                      print(f"超时未收到 ACK{state}，重发数据包。")              # 文件传输完成后，单独发送结束标记              end\_packet = "EOF".encode('utf-8')              server\_socket.sendto(end\_packet, (CLIENT\_IP, CLIENT\_PORT))              print("文件传输结束标记已发送。")      except FileNotFoundError:          print("要发送的文件不存在。")      finally:          server\_socket.close()  if \_\_name\_\_ == "\_\_main\_\_":      udp\_server()  client\_cs.py  import socket  CLIENT\_IP = '127.0.0.1'  CLIENT\_PORT = 12346  BUFFER\_SIZE = 1024  FILE\_PATH = 'received\_file.txt'  # 接收后保存的文件  def udp\_client():      # 创建客户端套接字      client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      client\_socket.bind((CLIENT\_IP, CLIENT\_PORT))      print("客户端启动，等待文件接收...")      expected\_packet = 0      try:          with open(FILE\_PATH, 'wb') as file:              while True:                  data, server\_address = client\_socket.recvfrom(BUFFER\_SIZE + 10)  # 增加缓冲区以处理数据包头                    # 当内容为结束符号时                  if data.decode('utf-8') == "EOF":                      print("文件接收完毕，收到结束标记。")                      break  # 停止接收                  # 使用'-'分隔符将序列号和文件块分开                  sequence\_number\_str, file\_chunk = data.decode('utf-8').split('-', 1)                  try:                      sequence\_number = int(sequence\_number\_str)  # 将序列号转换为整数                  except ValueError:                      print(f"解析序列号失败，收到无效数据：{sequence\_number\_str}")                      continue                  if sequence\_number == expected\_packet:                      # 写入接收到的文件块                      file.write(file\_chunk.encode('utf-8'))  # 将文件块写入文件                      print(f"收到数据包：{expected\_packet}")                      # 发送 ACK 确认                      ack = f"ACK{expected\_packet}".encode('utf-8')                      client\_socket.sendto(ack, server\_address)                      expected\_packet = 1 - expected\_packet  # 切换期望的数据包序号                  else:                      print(f"接收到错误包，期望包号：{expected\_packet}")        except KeyboardInterrupt:          print("\n客户端已被中断，正在关闭...")      finally:          client\_socket.close()          print("客户端已关闭。")  if \_\_name\_\_ == "\_\_main\_\_":      udp\_client()  2. GBN协议的实现  2.1 数据分组格式  GBN/SR数据分组格式与停等协议基本一致，在停等协议的基础上，将序列号的位数由1位扩展为了多位，从而对滑动窗口进行支持。   |  |  | | --- | --- | | seq | data\_packet |   这里的seq位数则主要由宏观全局变量SEQ\_SIZE来进行确定。  而在双向传输的过程中，类似于停等协议的双向传输，需要同时包含数据包和ACK的部分：   |  |  |  |  | | --- | --- | --- | --- | | seq | data\_packet | ACK | Ack\_seq |   2.2 确认分组格式  GBN/SR的确认分组格式则需要在停等协议的基础上，将ACK的序列修改为多位。虽然GBN协议的接收窗口大小仍然为1，但是为了能够与发送方发送的序列号进行比对，所以需要将确认分组的序列号位数修改为多位。   |  |  | | --- | --- | | ACK | seq |   这里的seq位数则取决于发送方的seq，从而用于server端确认分组在窗口中的哪一个具体位置。  而在双向传输中，接收方发送的信息同样需要包含ACK和数据包的信息。   |  |  |  |  | | --- | --- | --- | --- | | ACK | Ack\_seq | seq | data\_packet |   2.3 协议两端的程序流程图  （1）服务端    （2）客户端    2.4 协议典型的交互过程  GBN协议中，发送方在未收到确认的情况下连续发送多个数据帧，但接收方只能按序接收。发送方维持一个发送窗口，窗口内的帧可以连续发送。每当发送方发送一个数据帧，它会等待接收方的ACK确认。如果接收方收到的帧无误且序号正确，则发送ACK；如果帧有误或序号不正确，接收方丢弃该帧，只确认上一个按序接收到的帧。当发送方超时或收到的ACK号小于预期，发送方会从出错帧开始重传整个发送窗口内的所有帧。GBN通过窗口机制提高了传输效率，但在发生错误时会导致多帧重传。    2.5 数据分组丢失验证模拟方法  数据分组的丢失验证主要也是通过随机数的方式进行实现，通过设定随机数的模拟比例输入来进行丢失率的模拟。  # 利用随机数来模拟ACK丢失的情况  def loss\_in\_loss\_ratio(loss\_ratio):      return random.random() < loss\_ratio  2.6 程序实现的主要函数及其作用  在GBN协议中主要实现了定时器Timer类、loss\_in\_loss\_ratio函数、send\_window\_data函数、server\_program函数和client\_program函数。其中loss\_in\_loss\_ratio函数用于模拟数据分组丢失的验证，在2.5中描述。  定时器Timer类主要用于控制超时重传，属性包括超时时间和超时处理线程。方法包括定时器的启动start函数和stop函数，通过单独实现定时器Timer类能够提供更为细节化的朝时期设置和停止的设定。在start方法中，主要启动一个定时器线程，当达到设定的timeout之后触发重传函数。而在stop方法中首先判断是否还有定时器线程，如果有的话则用cancel方法将其取消掉。  send\_window\_data(sock, addr, data\_list, base, next\_seq\_num)函数主要用于将窗口内的全部数据进行发送，这里sock是发送的套接字，addr是目标client地址，data\_list是全部数据的队列，base是当前窗口的的基地址，next\_seq\_num是还没有被发送的下一个数据的index。  server\_program函数则是服务端运行的函数，用于按照gbn协议对数据进行发送。client\_program函数则是客户端运行的函数，用于按照gbn协议对数据进行接受  2.7 详细注释源程序  (1) 基础gbn协议的实现  server.py  import socket  import time  import threading  import random  SERVER\_IP = '127.0.0.1'  SERVER\_PORT = 12345  BUFFER\_SIZE = 1024  SEQ\_SIZE = 4  # 序列号比特数 L = 4，修改时需要同时修改服务端和客户端的SEQ\_SIZE  WINDOW\_SIZE = 8  # 发送窗口大小 W，满足 W + 1 <= 2^L  TIMEOUT = 3  # 超时时间为 3 秒  PACKET\_LOSS\_RATE = 0.2  # 模拟包丢失率    # 计时器类，用于控制超时重传  class Timer:      def \_\_init\_\_(self, timeout):          self.timeout = timeout          # 超时时间          self.timer\_thread = None        # 超时处理线程      def start(self, callback):          # 如果之前有定时器在运行，它会调用 stop() 方法停止当前定时器，防止重复启动。          if self.timer\_thread is not None:              self.stop()            # 使用 threading.Timer 创建一个定时器线程，当达到设定的 timeout 时间后，触发重传的函数          self.timer\_thread = threading.Timer(self.timeout, callback)          self.timer\_thread.start()      def stop(self):          #  如果有定时器在运行，那么停止它          if self.timer\_thread is not None:              self.timer\_thread.cancel()              self.timer\_thread = None  # 利用随机数模拟随机丢失  def loss\_in\_loss\_ratio(loss\_ratio):      return random.random() < loss\_ratio  # 发送滑动窗口中的数据  # sock是发送的套接字，addr是目标client地址  # data\_list是全部数据的队列  # base是当前窗口的的基地址  # next\_seq\_num是还没有被发送的下一个数据的index  # 重新传输所有已发送但是没有收到ACK的数据包，也就是从基序号base到next\_seq\_num前  def send\_window\_data(sock, addr, data\_list, base, next\_seq\_num):      # base      for i in range(base, next\_seq\_num):          #          if i < len(data\_list):              data = f"{i % (2 \*\* SEQ\_SIZE)}:{data\_list[i]}"                # 模拟丢失              if not loss\_in\_loss\_ratio(PACKET\_LOSS\_RATE):                  sock.sendto(data.encode(), addr)                  print(f"Sent packet: {data}")              else:                  print(f"Packet loss, Seq: {i % (2 \*\* SEQ\_SIZE)}")  # 服务端运行函数  def server\_program():      # 创建套接字，并且绑定在对应的IP和端口      sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      sock.bind((SERVER\_IP, SERVER\_PORT))      # 模拟要发送的数据      data\_list = [f"Data {i}" for i in range(50)]  # 模拟要传输的数据          base = 0                # 滑动窗口的第一个序号，也就是序列号最小的已发送但没收到ACK的数据包      next\_seq\_num = 0        # 下一个可用的序列号，也就是第一个还没发送的数据报      client\_addr = None      # 客户端地址，使用recv方法来获取      timer = Timer(TIMEOUT)  # 实例化计时器的对象，设定超时时间为3s      # 发生超时，重新传输所有已发送但是没有收到ACK的数据包，也就是从基序号base到next\_seq\_num前      def timeout\_callback():          print("Timeout! Resending window...")          send\_window\_data(sock, client\_addr, data\_list, base, next\_seq\_num)          timer.start(timeout\_callback)   # 重传结束后再次启动计时器      print(f"Server is listening on {SERVER\_IP}:{SERVER\_PORT}")      while True:          message, client\_addr = sock.recvfrom(BUFFER\_SIZE)          message = message.decode()          # 首先接收客户端的开始信息          if message == 'start':              # 开始传输数据，设定计时器开始计时              print("Start sending data...")              # timer.start(timeout\_callback)              # 当基序号在列表范围内时，重复尝试发送              while base < len(data\_list):                  # 如果下一可用序列在滑动窗口范围内，且每超出待发送队列范围，那么直接发送                  if next\_seq\_num < base + WINDOW\_SIZE and next\_seq\_num < len(data\_list):                      # 当滑动窗口还没有结束，且base = next\_seq\_num时，还要继续启动计时器                      if base == next\_seq\_num:                          timer.start(timeout\_callback)                      send\_window\_data(sock, client\_addr, data\_list, base, next\_seq\_num + 1)                      # send\_window\_data(sock, client\_addr, data\_list, base, next\_seq\_num + 1)                      next\_seq\_num += 1                  # 接收ack，并获取ack序列号                  ack\_message, \_ = sock.recvfrom(BUFFER\_SIZE)                  ack\_num = int(ack\_message.decode())                  print(f"Received ACK: {ack\_num}")                    # 如果接收到ack，那么更新base的数字（base之前全被接收）                  if ack\_num >= base:                      # 当乱序到达时，可以确保base回退到概要发送的第一个                      base = ack\_num + 1                        if base == next\_seq\_num:                          timer.stop()            # 当base追赶上了next\_seq\_num，说明结束，停止计时器                      else:                          timer.start(timeout\_callback)          elif message == 'quit':              print("Client requested to quit.")              break      sock.close()  if \_\_name\_\_ == "\_\_main\_\_":      server\_program()  client.py  import socket  import random  SERVER\_IP = '127.0.0.1'  SERVER\_PORT = 12345  BUFFER\_SIZE = 1024  ACK\_LOSS\_RATE = 0.2  # 模拟 ACK 丢失率  SEQ\_SIZE = 4  # 利用随机数来模拟ACK丢失的情况  def loss\_in\_loss\_ratio(loss\_ratio):      return random.random() < loss\_ratio  # 客户端程序  def client\_program():      # 首先创建两个socket，并绑定在对应的IP和端口号上      sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      server\_addr = (SERVER\_IP, SERVER\_PORT)      # 向服务端发送开始信号      sock.sendto(b'start', server\_addr)  # 向服务器请求开始数据传输      # 期待的序列号      expected\_seq\_num = 0      while True:          try:                # 从绑定端口接收数据，并对数据进行解码从而获取信息              data, \_ = sock.recvfrom(BUFFER\_SIZE)              message = data.decode()              # 依据':'分割序列号和数据信息              seq\_num, content = message.split(':', 1)              seq\_num = int(seq\_num)              # 依据序列号的位数，进行取余比对，如果比对一致，那么直接输出信息，接收到了这个包              if seq\_num == expected\_seq\_num % (2 \*\* SEQ\_SIZE):                  print(f"Received packet: {message}")                  # 期待的包序列号+1                  expected\_seq\_num += 1              else:                  print(f"Out of order packet: {message}, expected: {expected\_seq\_num}")              # 模拟 ACK 丢失              if not loss\_in\_loss\_ratio(ACK\_LOSS\_RATE):                  # 发送确认的ACK信息，ACK序列号指的是1——ACK的包都已接收                  ack\_message = str(expected\_seq\_num - 1).encode()                  sock.sendto(ack\_message, server\_addr)                  print(f"Sent ACK: {expected\_seq\_num - 1}")              else:                  # 如果模拟ACK丢失，啥也不发送                  print(f"ACK {expected\_seq\_num - 1} lost")          except KeyboardInterrupt:              sock.sendto(b'quit', server\_addr)              break      sock.close()  if \_\_name\_\_ == "\_\_main\_\_":      client\_program()  (2) 双向传输的实现  gbn.py  import socket  import time  import threading  import random  BUFFER\_SIZE = 1024  SEQ\_SIZE = 4  # 序列号比特数 L = 4，修改时需要同时修改服务端和客户端的SEQ\_SIZE  WINDOW\_SIZE = 8  # 发送窗口大小 W，满足 W + 1 <= 2^L  TIMEOUT = 3  # 超时时间为 3 秒  PACKET\_LOSS\_RATE = 0.1  # 模拟包丢失率  ACK\_LOSS\_RATE = 0.1  # 模拟 ACK 丢失率    ########## server端部分  # 计时器类，用于控制超时重传  class Timer:      def \_\_init\_\_(self, timeout):          self.timeout = timeout          # 超时时间          self.timer\_thread = None        # 超时处理线程      def start(self, callback):          # 如果之前有定时器在运行，它会调用 stop() 方法停止当前定时器，防止重复启动。          if self.timer\_thread is not None:              self.stop()            # 使用 threading.Timer 创建一个定时器线程，当达到设定的 timeout 时间后，触发重传的函数          self.timer\_thread = threading.Timer(self.timeout, callback)          self.timer\_thread.start()      def stop(self):          #  如果有定时器在运行，那么停止它          if self.timer\_thread is not None:              self.timer\_thread.cancel()              self.timer\_thread = None  # 利用随机数模拟随机丢失  def loss\_in\_loss\_ratio(loss\_ratio):      return random.random() < loss\_ratio  # 发送滑动窗口中的数据  # sock是发送的套接字，addr是目标client地址  # data\_list是全部数据的队列  # base是当前窗口的的基地址  # next\_seq\_num是还没有被发送的下一个数据的index  # 重新传输所有已发送但是没有收到ACK的数据包，也就是从基序号base到next\_seq\_num前  def send\_window\_data(sock, addr, data\_list, base, next\_seq\_num):      # base      for i in range(base, next\_seq\_num):          #          if i < len(data\_list):              data = f"{i % (2 \*\* SEQ\_SIZE)}:{data\_list[i]}"                # 模拟丢失              if not loss\_in\_loss\_ratio(PACKET\_LOSS\_RATE):                  sock.sendto(data.encode(), addr)                  print(f"服务端：Sent packet: {data}")              else:                  print(f"服务端：Packet loss, Seq: {i % (2 \*\* SEQ\_SIZE)}")  # 服务端运行函数  def server\_program(server\_ip, server\_port, client\_ip, client\_port, data\_list):      # 创建套接字，并且绑定在对应的IP和端口      sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      sock.bind((server\_ip, server\_port))        base = 0                # 滑动窗口的第一个序号，也就是序列号最小的已发送但没收到ACK的数据包      next\_seq\_num = 0        # 下一个可用的序列号，也就是第一个还没发送的数据报      # client\_addr = None      # 客户端地址，使用recv方法来获取      timer = Timer(TIMEOUT)  # 实例化计时器的对象，设定超时时间为3s      # 发生超时，重新传输所有已发送但是没有收到ACK的数据包，也就是从基序号base到next\_seq\_num前      def timeout\_callback():          print("服务端：Timeout! Resending window...")          send\_window\_data(sock, (client\_ip, client\_port), data\_list, base, next\_seq\_num)          timer.start(timeout\_callback)   # 重传结束后再次启动计时器      print(f"Server is listening on {server\_ip}:{server\_port}")      while True:          # message, client\_addr = sock.recvfrom(BUFFER\_SIZE)          # message = message.decode()          message = 'start'          # 首先接收客户端的开始信息          if message == 'start':              # 开始传输数据，设定计时器开始计时              # print("服务端：Start sending data...")              # timer.start(timeout\_callback)              # 当基序号在列表范围内时，重复尝试发送              while base < len(data\_list):                  # 如果下一可用序列在滑动窗口范围内，且每超出待发送队列范围，那么直接发送                  if next\_seq\_num < base + WINDOW\_SIZE and next\_seq\_num < len(data\_list):                      # 当滑动窗口还没有结束，且base = next\_seq\_num时，还要继续启动计时器                      if base == next\_seq\_num:                          timer.start(timeout\_callback)                      send\_window\_data(sock, (client\_ip, client\_port), data\_list, base, next\_seq\_num + 1)                      # send\_window\_data(sock, client\_addr, data\_list, base, next\_seq\_num + 1)                      next\_seq\_num += 1                  # 接收ack，并获取ack序列号                  ack\_message, \_ = sock.recvfrom(BUFFER\_SIZE)                  ack\_num = int(ack\_message.decode())                  print(f"服务端：Received ACK: {ack\_num}")                    # 如果接收到ack，那么更新base的数字（base之前全被接收）                  if ack\_num >= base:                      # 当乱序到达时，可以确保base回退到概要发送的第一个                      base = ack\_num + 1                        if base == next\_seq\_num:                          timer.stop()            # 当base追赶上了next\_seq\_num，说明结束，停止计时器                      else:                          timer.start(timeout\_callback)          elif message == 'quit':              print("Client requested to quit.")              break      sock.close()  ########### client端部分  # 利用随机数来模拟ACK丢失的情况  def loss\_in\_loss\_ratio(loss\_ratio):      return random.random() < loss\_ratio  # 客户端程序  def client\_program( client\_ip, client\_port):      # 首先创建两个socket，并绑定在对应的IP和端口号上      sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      sock.bind((client\_ip, client\_port))      # server\_addr = (server\_ip, server\_port)      # 向服务端发送开始信号      # sock.sendto(b'start', server\_addr)  # 向服务器请求开始数据传输      # 期待的序列号      expected\_seq\_num = 0      while True:          try:                # 从绑定端口接收数据，并对数据进行解码从而获取信息              data, server\_addr = sock.recvfrom(BUFFER\_SIZE)              message = data.decode()              # 依据':'分割序列号和数据信息              seq\_num, content = message.split(':', 1)              seq\_num = int(seq\_num)              # 依据序列号的位数，进行取余比对，如果比对一致，那么直接输出信息，接收到了这个包              if seq\_num == expected\_seq\_num % (2 \*\* SEQ\_SIZE):                  print(f"客户端：Received packet: {message}")                  # 期待的包序列号+1                  expected\_seq\_num += 1              else:                  print(f"客户端：Out of order packet: {message}, expected: {expected\_seq\_num}")              # 模拟 ACK 丢失              if not loss\_in\_loss\_ratio(ACK\_LOSS\_RATE):                  # 发送确认的ACK信息，ACK序列号指的是1——ACK的包都已接收                  ack\_message = str(expected\_seq\_num - 1).encode()                  sock.sendto(ack\_message, server\_addr)                  print(f"客户端：Sent ACK: {expected\_seq\_num - 1}")              else:                  # 如果模拟ACK丢失，啥也不发送                  print(f"客户端：ACK {expected\_seq\_num - 1} lost")          except KeyboardInterrupt:              sock.sendto(b'quit', server\_addr)              break      sock.close()    main.py  import gbn as gbn  import threading  SERVER\_IP = '127.0.0.1'  SERVER\_PORT1 = 12345  SERVER\_PORT2 = 12347  CLIENT\_IP = '127.0.0.1'  CLIENT\_PORT1 = 12346  CLIENT\_PORT2 = 12348  # 模拟要发送的数据  data1\_list = [f"Data1 {i}" for i in range(50)]  # 模拟要传输的数据  data2\_list = [f"Data2 {i}" for i in range(50)]  # 模拟要传输的数据  Thread\_Server\_Send = threading.Thread(target=gbn.server\_program, args=(SERVER\_IP, SERVER\_PORT1, CLIENT\_IP, CLIENT\_PORT2, data1\_list,))  Thread\_Server\_Recv = threading.Thread(target=gbn.client\_program, args=(SERVER\_IP, SERVER\_PORT2,  ))  Thread\_Client\_Send = threading.Thread(target=gbn.server\_program, args=(CLIENT\_IP, CLIENT\_PORT1, SERVER\_IP, SERVER\_PORT2, data2\_list))  Thread\_Client\_Recv = threading.Thread(target=gbn.client\_program, args=(CLIENT\_IP, CLIENT\_PORT2, ))  Thread\_Server\_Send.start()  Thread\_Server\_Recv.start()  Thread\_Client\_Send.start()  Thread\_Client\_Recv.start()    (3) SR协议的实现  run\_sr.py  import sr as sr  import threading  SERVER\_IP = '127.0.0.1'  SERVER\_PORT = 12345  CLIENT\_IP = '127.0.0.1'  CLIENT\_PORT = 12346  # 模拟要传输的数据  data\_list = [f"Data{i}" for i in range(50)]  Thread\_Server = threading.Thread(target=sr.server\_program, args=(SERVER\_IP, SERVER\_PORT, CLIENT\_IP, CLIENT\_PORT, data\_list, ))  Thread\_Client = threading.Thread(target=sr.client\_program, args=(CLIENT\_IP, CLIENT\_PORT, SERVER\_IP, SERVER\_PORT, ))  Thread\_Server.start()  Thread\_Client.start()  sr.py  import socket  import threading  import random  import time  BUFFER\_SIZE = 1024  SEQ\_SIZE = 4  # 序列号位数  WINDOW\_SIZE = 8  # 窗口大小，W < 2^SEQ\_SIZE  TIMEOUT = 3  # 超时时间，单位秒  PACKET\_LOSS\_RATE = 0.1  # 模拟数据包丢失率  ACK\_LOSS\_RATE = 0.1  # 模拟ACK丢失率  ########## 服务器端部分 ##########  # 计时器类，用于每个数据包独立的超时处理  class Timer:      def \_\_init\_\_(self, timeout, callback):          self.timeout = timeout  # 超时时间          self.callback = callback  # 超时回调函数          self.timer\_thread = None  # 定时器线程          self.lock = threading.Lock()          self.active = False      def start(self):          with self.lock:              self.timer\_thread = threading.Timer(self.timeout, self.callback)              self.active = True              self.timer\_thread.start()      def stop(self):          with self.lock:              if self.active:                  self.timer\_thread.cancel()                  self.active = False  # 模拟数据包丢失  def loss\_in\_loss\_ratio(loss\_ratio):      return random.random() < loss\_ratio  # 发送单个数据包  def send\_packet(sock, addr, seq\_num, data):      packet = f"{seq\_num}:{data}"      if not loss\_in\_loss\_ratio(PACKET\_LOSS\_RATE):          sock.sendto(packet.encode(), addr)          print(f"服务器：发送数据包：{packet}")      else:          print(f"服务器：数据包丢失，序列号：{seq\_num}")  # 服务器程序，使用选择性重传协议  def server\_program(server\_ip, server\_port, client\_ip, client\_port, data\_list):      # 创建socket      sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      sock.bind((server\_ip, server\_port))      base = 0  # 窗口起始序号      next\_seq\_num = 0  # 下一个发送的序列号      window = {}  # 存储已发送但未确认的数据包 {序号: 数据}      timers = {}  # 存储每个数据包的定时器 {序号: Timer对象}      client\_addr = (client\_ip, client\_port)      print(f"服务器正在监听 {server\_ip}:{server\_port}")      # 超时回调函数，重传特定序列号的数据包      def timeout\_callback(seq):          print(f"服务器：超时，重传数据包，序列号：{seq%(2\*\*SEQ\_SIZE)}")          send\_packet(sock, client\_addr, seq%(2\*\*SEQ\_SIZE), data\_list[seq])          # 重新启动该数据包的定时器          timers[seq].start()      # 等待客户端发送“start”信号      while True:          message, addr = sock.recvfrom(BUFFER\_SIZE)          message = message.decode()          if message == 'start':              print("服务器：开始发送数据...")              break      # 发送数据包      while base < len(data\_list):          # 发送窗口内的数据包          while next\_seq\_num < base + WINDOW\_SIZE and next\_seq\_num < len(data\_list):              seq\_num = next\_seq\_num % (2 \*\* SEQ\_SIZE)              send\_packet(sock, client\_addr, seq\_num, data\_list[next\_seq\_num])              # 启动该数据包的定时器              timer = Timer(TIMEOUT, lambda s=next\_seq\_num: timeout\_callback(s))              timer.start()              timers[next\_seq\_num] = timer              window[next\_seq\_num] = seq\_num              next\_seq\_num += 1          try:              sock.settimeout(TIMEOUT)              ack\_message, \_ = sock.recvfrom(BUFFER\_SIZE)              ack\_num = int(ack\_message.decode())              print(f"服务器：收到ACK：{ack\_num}")              # 查找对应的发送序号              ack\_received = False              for key, seq in list(window.items()):                  if seq == ack\_num:                      print(f"服务器：ACK确认，序列号：{seq}")                        timers[key].stop()  # 停止该数据包的定时器                      del timers[key]  # 移除定时器                      del window[key]  # 从窗口移除该数据包                        if key == base:                          # 如果确认的是窗口的最小序号，移动窗口基准                          while base not in window and base < next\_seq\_num:                              base += 1                      ack\_received = True                      break              if not ack\_received:                  print("服务器：收到不在缓存范围内的ACK")                  # # 当ACK发生过丢失，即接收方返回expected\_seq - 1                  # if base < ack\_num:                  #     base = ack\_num + 1                  #     for key, seq in list(window.items()):                  #         if seq < base:                  #             timers[key].stop()  # 停止该数据包的定时器                  #             del timers[key]  # 移除定时器                  #             del window[key]  # 从窗口移除该数据包                      print(f"服务器：收到重复或无效的ACK：{ack\_num}")            except socket.timeout:              # 可能有数据包超时，已由各自的定时器处理              continue      # 所有数据包发送并确认后，发送“quit”信号      sock.sendto(b'quit', client\_addr)      print("服务器：所有数据包已发送并确认，退出。")      sock.close()    ########### 客户端部分 ###########  # 模拟ACK丢失  def loss\_in\_loss\_ratio(loss\_ratio):      return random.random() < loss\_ratio  # 客户端程序，使用选择性重传协议  def client\_program(client\_ip, client\_port, server\_ip, server\_port):      sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)      sock.bind((client\_ip, client\_port))      server\_addr = (server\_ip, server\_port)      expected\_seq\_num = 0  # 下一个期望的序列号      received\_packets = {}  # 缓存不按序到达的数据包 {序号: 数据}      # 发送“start”信号给服务器      sock.sendto(b'start', server\_addr)      print("客户端：发送‘start’信号给服务器。")      while True:          try:              data, addr = sock.recvfrom(BUFFER\_SIZE)              message = data.decode()              if message == 'quit':                  print("客户端：收到‘quit’信号，退出。")                  break              # 解析收到的数据包              try:                  seq\_num\_str, content = message.split(':', 1)                  seq\_num = int(seq\_num\_str)              except ValueError:                  print("客户端：收到格式错误的数据包，忽略。")                  continue              abs\_seq\_num = seq\_num              # 检查数据包是否在接收窗口内              window\_start = expected\_seq\_num              window\_end = (expected\_seq\_num + WINDOW\_SIZE) % (2\*\*SEQ\_SIZE)              judge = 0                if window\_end > window\_start:                  if window\_start <= abs\_seq\_num < window\_end:                      judge = 1                  else:                      judge = 0              else:                  if (window\_start<= abs\_seq\_num < 2\*\*SEQ\_SIZE) or (0 <= abs\_seq\_num < window\_end ):                      judge = 2                  else:                      judge = 0              if judge != 0:                  if abs\_seq\_num == expected\_seq\_num:                      print(f"客户端：收到期望的数据包，序列号：{seq\_num}，内容：{content}")                      expected\_seq\_num = (expected\_seq\_num + 1) % (2 \*\* SEQ\_SIZE)                      # 检查是否有缓存的数据包可以处理                      while expected\_seq\_num in received\_packets:                          buffered\_content = received\_packets.pop(expected\_seq\_num)                          print(f"客户端：处理缓存的数据包，序列号：{expected\_seq\_num}，内容：{buffered\_content}")                          expected\_seq\_num = (expected\_seq\_num + 1) % (2 \*\* SEQ\_SIZE)                  elif abs\_seq\_num != expected\_seq\_num:                      if abs\_seq\_num not in received\_packets:                          print(f"客户端：收到乱序数据包，序列号：{seq\_num}，内容：{content}")                          received\_packets[abs\_seq\_num] = content                      else:                          print(f"客户端：已缓存数据包，序列号：{seq\_num}，无需重复缓存。")                  # 发送ACK                  if not loss\_in\_loss\_ratio(ACK\_LOSS\_RATE):                      ack\_message = str(seq\_num).encode()                      sock.sendto(ack\_message, server\_addr)                      print(f"客户端：发送ACK，序列号：{seq\_num}")                  else:                      print(f"客户端：ACK丢失，序列号：{seq\_num}")                  judge = 0              else:                  print(f"客户端：收到不在窗口内的数据包，序列号：{seq\_num}，已丢弃。")                  # 可选：重发上一个确认的ACK                    last\_ack = seq\_num                  if not loss\_in\_loss\_ratio(ACK\_LOSS\_RATE):                      ack\_message = str(last\_ack).encode()                      sock.sendto(ack\_message, server\_addr)                      print(f"客户端：重新发送ACK，序列号：{last\_ack}")                  else:                      print(f"客户端：重新发送ACK丢失，序列号：{last\_ack}")                  judge = 0          except KeyboardInterrupt:              # 用户中断时发送“quit”信号              sock.sendto(b'quit', server\_addr)              print("客户端：用户中断，发送‘quit’信号并退出。")              break      sock.close() |
| 实验结果： |
| （1）基础停等协议的实现    （2）双向传输的实现    （3）C/S架构文件传输应用的实现    （4）基础gbn协议的实现    （5）双向gbn的实现    （6）sr协议的实现 |
| 问题讨论： |
| SR协议中由于使用了窗口，所以在比较序列号的逻辑上的先后的时候不能单纯的比较大小而是要用循环队列的思想来看待这个大小的比较。比如当序列号为4为，窗口大小为8，窗口有可能会是12-16和0-3，这个时候在进行数据包是否在接收方窗口中的比较的时候，需要做额外的判断。可以用window\_end和window\_start的大小关系做区分。  当window\_end > window\_start，那么说明没有上面的情况，直接比大小就可以。  当window\_end < window\_start，说明出现了上面的情况，那么判断调件变成：window\_start <= seq < 2 \*\* SEQ\_SIZE or 0 <= seq <= window\_end |
| 心得体会： |
| 对GBN协议和SR协议有了更加深刻的认识和理解，也对socket编程有了更多的认识和理解。 |