**100071011 Computer Networks 2023-2024-2**

**Project-1**

**Reliable File Transfer using Go-Back-N protocol**

**Specification**

|  |  |
| --- | --- |
| **学号 (Student ID)** | **1120211099** |
| **姓名 (Name)** | **周豪捷** |
| **班号 (Class No.)** | **07032101** |
| **授课教师 (Instructor)** | **宿红毅** |

**School of Computer**

**Beijing Institute of Technology**

**April 12, 2024**

1. **Requirement Analysis**

**项目需求分析：**

1. 功能需求：文件传输功能：确保系统能够可靠地传输大文件。

GBN协议实现：理解GBN 协议并将其实现在系统中，包括发送窗口管理、超时重传、确认机制等。

UDP Socket模拟：使 UDP Socket模拟PDU的发送和接收。

全双工通信：支持双向文件传输，确保Host1和Host2之间可以相互发送文件。

1. 性能需求分析：数据传输速率：确定系统的数据传输速率需求，以确保在一定时间内能够传输完大文件。

可靠性要求：确定文件传输的可靠性需求，包括对数据丢失、错误和重传的容忍程度。

1. 非功能需求分析：安全性：确保数据传输过程中的机密性和完整性，防止数据被篡改或窃取。

可维护性：设计系统以便于维护和扩展，包括清晰的代码结构、文档和注释。

用户友好性：设计简洁直观的用户界面，使用户能够轻松地操作系统并了解其状态。

1. 约束条件：CRC-CCITT校验：使用CRC-CCITT标准进行校验，确保数据的完整性。

PDU长度限制：PDU中数据部分的长度不超过4KB。

配置文件参数限制：保证配置文件中各参数的合理范围，包括UDP端口、PDU数据字段长度、错误率、丢包率等。

**对于GBN可靠传输的理解：**

1. 发送方维护一个发送窗口，用于存放已发送但未确认的数据包。窗口大小决定了可以连续发送的数据包数量。接收方收到数据包后，如果按序接收，就会发送确认消息给发送方，确认已接收的数据包。发送方根据收到的确认消息调整发送窗口。
2. 如果发送方在一定时间内未收到确认消息，就会假定数据包丢失，触发超时重传机制，重新发送窗口内的所有未确认数据包。
3. GBN允许发送方连续发送多个数据包，提高了传输效率。GBN仅重传丢失的数据包及其后续的数据包，而不会重传已确认的数据包，减少了重传开销。
4. 接收方必须按顺序接收数据包，确保数据的正确性和完整性。超时重传机制能够及时检测和处理丢失的数据包，确保数据的可靠传输。

**主要的问题：**

1. 丢包问题：在不可靠的通信信道上，数据包可能会丢失，导致接收方无法收到完整的文件。
2. 数据包的乱序：数据包可能以乱序的方式到达接收方，接收方需要按照顺序重组这些数据包以获得完整的文件。
3. 网络延迟：网络延迟可能导致数据包的传输时间不确定，需要确保超时重传机制能够及时检测到丢失的数据包。

**关键功能：**

1. 窗口管理：发送方维护一个发送窗口，用于控制连续发送的数据包数量。窗口大小决定了发送方能够发送的最大数据包数量，同时也决定了发送方期望接收的确认消息数量。
2. 确认机制：接收方收到数据包后，向发送方发送确认消息，告知发送方已成功接收数据包，发送方根据接收到的确认消息调整发送窗口的位置。
3. 超时重传：发送方在一定时间内未收到确认消息，则假定数据包丢失，触发超时重传机制，重新发送窗口内的所有未确认数据包。
4. **Design**

**帧结构：**

class PDU:  
 def \_\_init\_\_(self, seq\_num, data=b’’):  
 self.seq = seq\_num  
 self.data = data  
 self.crc = crc16.crc16xmodem(data)

定义的数据帧结构包括数据包的序列号、数据和CRC校验值属性。

**CRC校验规则：**

采用CRC16-CCITT (XModem)校验。对传入data中的每一个字节进行遍历，在每次迭代中，先将crc左移8位，然后与table中的一个元素进行异或运算，该元素的索引为crc右移8位与当前字节进行异或运算的结果。

最后返回crc与0xffff进行与运算的结果，以确保结果是16位。

def \_crc16(data, crc, table):  
 for byte in data:  
 crc = ((crc << 8) & 0xff00) ^ table[((crc >> 8) & 0xff) ^ byte]  
 return crc & 0xffff

**滑动窗口设置和控制：**

通过配置文件获取滑动窗口的大小，用一个window\_size变量记住，它决定了发送方可以在等待确认之前发送多少个数据包，用于防止发送方将接收方的缓冲区填满，导致数据丢失。只要已发送但尚未确认的数据包数量不超过窗口大小，发送方就会继续发送数据包。如果已发送但未确认的数据包数量达到窗口大小，发送方将停止发送数据包，直到收到新的确认。

**序号设置和控制：**

在发送端，数据包的序号初始化为配置文件中InitSeq的值。在这个项目里，考虑到传输文件大小不会特别特别大，所以我为每个数据包都设置了单独的序号。每次读取文件并创建一个新的数据包的时候，数据包的序列号就会加一。

在接收端，frame\_expected变量用于跟踪期望接收的数据包的序号。如果接收到的数据包序号与frame\_expected相同，那么数据包被认为是正确的，frame\_expected会增加1，准备接收下一个数据包。在确认数据包的接收时，ack\_exp变量用于跟踪期望接收确认的数据包的序号。当接收到确认时，如果确认的序号大于等于ack\_exp，那么ack\_exp会被设置为确认的序号加1，表示下一个期望接收确认的数据包序号。

**确认规则：**

在接收方设置一个无限循环，不断地从套接字接收数据包。每次接收到一个数据包，它都会将接收计数增加1。

对于每个接收到的数据包，将其分解为序号、CRC校验和和数据，计算接收到的数据的CRC校验和，然后与数据包中的CRC校验和进行比较。如果两者不相等，那么数据包被认为是错误的，记录一个错误消息，然后继续接收下一个数据包。

如果数据包的CRC校验和是正确的，那么将检查数据包的序号，如果序号与期望接收的序号相同，那么数据包被认为是正确的。记录一个确认消息，然后发送一个确认序号frame\_expected给发送方。然后，frame\_expected增加1，准备接收下一个数据包。

如果数据包的序号与frame\_expected不同，那么数据包被认为是无错误的，但是不是期望接收的数据包，那么会发送一个确认序号frame\_expected - 1给发送方，表示接收方已经接收到了序号为frame\_expected - 1的数据包，但是还没有接收到序号为frame\_expected的数据包。

**重传机制和计时器控制：：**

重传机制主要依赖于滑动窗口协议和定时器。每当发送方发送一个数据包，就会启动计时器。此后发送方等待接收确认，在等待期间继续发送其他数据包，直到已发送但未确认的数据包数量超过窗口大小。

如果在计时器超时之前收到了确认，那么发送方将停止计时器，更新窗口，并继续发送其他数据包。如果计时器超时，那么发送方将重新发送数据包，并重新启动超时计时器。

如果发送方收到了一个错误的确认NAK，那么发送方将从窗口中查找对应序列号的数据包并重新发送。

如果发送方收到了一个正确的确认ACK，那么发送方将更新窗口，并继续发送其他数据包。 **UDP Socket：**

该项目使用UDP socket来收发数据。在我自定义的启动方法中，每次启动会通过socket.socket()函数创建一个新的套接字对象，该函数接受两个参数：socket.AF\_INET和socket.SOCK\_DGRAM。socket.AF\_INET表示这个套接字用于IPv4网络，socket.SOCK\_DGRAM表示这个套接字用于UDP协议。

socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

在初始化时从配置文件读取收发双方的ip地址的端口号，然后通过socket. bind()方法将套接字绑定到对应地址，是一个包含IP地址和端口号的二元组。

在发送数据时，使用socket的sendto()方法。这个方法接受两个参数：要发送的数据和接收方的地址。数据是字节类型，地址是一个包含IP地址和端口号的二元组。

def send(self, packet, sock, addr):  
 # 错误  
 if random.random() < self.prob\_err:  
 packet = self.error(packet)  
 # data 丢包  
 if random.random() > self.prob\_lost:  
 sock.sendto(packet, addr)  
 return

在接收数据时，使用socket的recvfrom()方法。这个方法接受一个参数：最大接收数据的长度，方法返回一个包含接收到的数据和发送方地址的二元组。

def receive(self, sock):  
 packet, addr = sock.recvfrom(self.data\_size \* 2)  
 return packet, addr

在程序结束前，调用socket的close()方法关闭套接字。

**系统模型：**

该项目是一个基于UDP的可靠文件传输系统，使用了GBN协议来确保数据的可靠传输。系统模型分为以下几个部分：

Sender：发送方，负责读取文件，将文件数据打包成数据包，并通过UDP发送给接收方。

Receiver：接收方，负责接收数据包，并将数据包的内容写入文件。

UDP：模拟UDP的类，封装了UDP socket的创建、数据的发送和接收、模拟网络的丢包和错误等功能。

start.py：系统入口，它创建并绑定UDP socket，然后根据用户的选择，启动一个新的线程来发送或接收文件。

utils.py：工具类，包含一些工具函数，例如读取配置文件的函数，以及用于计算CRC校验的函数。

config.ini：系统配置文件，包含了发送方和接收方的IP地址和端口号，以及其他一些参数。

**发送和接收函数：**

**send():**

首先读取配置文件config.ini中的一些参数，包括超时时间、窗口大小等，并存储在config字典中。

打开filename指定的发送文件，并将文件数据打包成数据包。数据包的格式是：序列号 + CRC校验值 + 数据。这个过程在一个while循环中完成，循环每次读取文件的一部分数据，然后生成一个数据包，将数据包添加到packets列表中。

读取完所有数据包后，进入一个循环来发送数据包。在每次循环中，首先检查是否有新的数据包可以发送。如果有，就发送新的数据包，然后检查是否有数据包需要重发。如果有，就重发。

数据包发送的过程使用一个定时器send\_timer来检查是否有数据包超时。如果有数据包超时，它就重发这些数据包。

另外，发送方使用一个线程来接收接收方传来的确认信息。如果接收到ACK，就更新期望的确认序列号ack\_exp。如果收到NAK，就找到对应数据包然后重新发送。

在发送完所有数据包后，发送一个特殊的final数据包来表示文件传输结束。

在传输完成后关闭文件，并在日志中记录一些统计信息，例如总的数据包数量num\_packets、总的传输时间Total\_time、超时重传次数TO\_COUNT和重传次数RO\_COUNT等。

**receive():**

首先创建一个UDP类的实例udt，并根据配置文件设置丢包率、错误率和数据包大小。

然后设置期望接收的数据包的序列号frame\_expected，并初始化接收的数据包数量recv\_count。

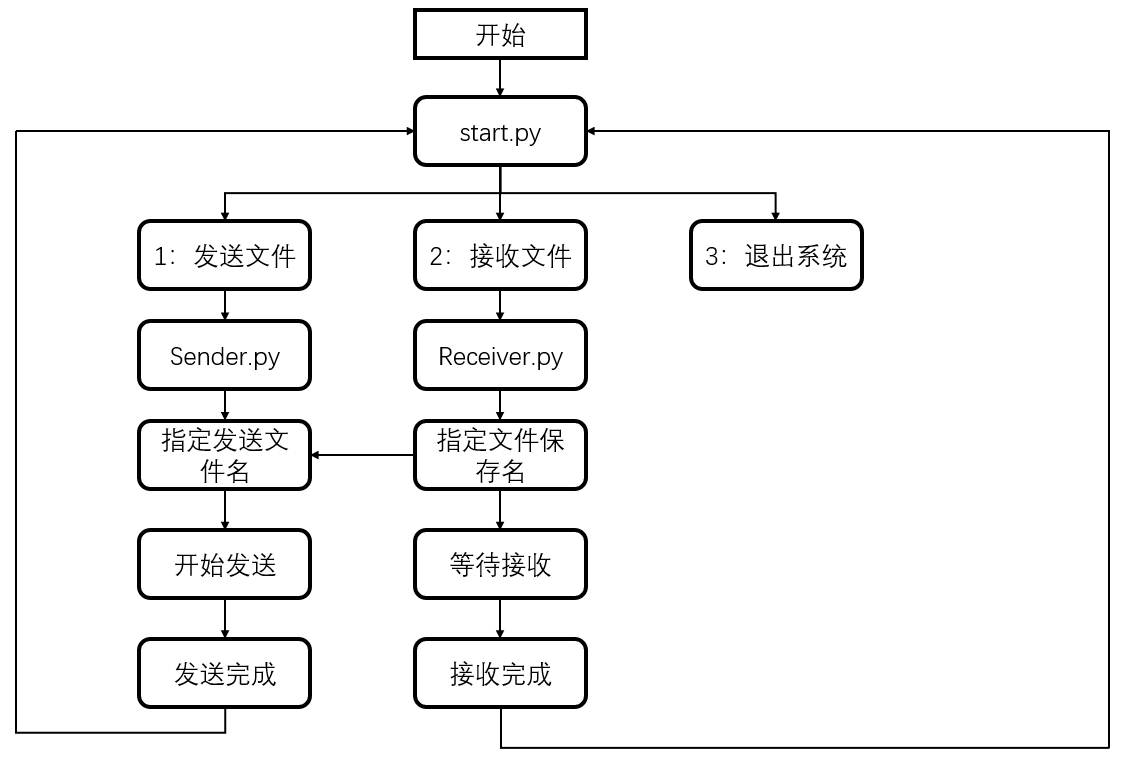
接着进入一个while循环来接收数据包。在每次循环中，它首先接收一个数据包，然后将数据包分解为序列号、CRC校验值和数据。计算接收到的数据的CRC校验值，如果计算出的CRC校验值与数据包中的CRC校验值不匹配，说明数据包有错误，函数就发送一个NAK给发送方，并继续下一次循环。

如果数据包的序列号与期望的序列号匹配，说明数据包是正确的，函数就将数据写入文件，并发送一个ACK给发送方，然后将期望的序列号加一。

如果数据包的序列号与期望的序列号不匹配，说明数据包是重复的，函数就发送一个ACK给发送方，ACK的序列号是期望的序列号减一。

当没有更多的数据包接收时，就关闭文件，并记录日志信息。

**流程图：**



**错误的模拟生成：**

我在UDP类中定义了error函数来模拟错误数据包的生成。错误数据包保留了原数据包的前16个字节，剩余部分由0-127中随机生成数来填充。

**日志记录：**

日志记录器的设置和使用主要在setup\_logger函数中。这个函数接受两个参数：ip\_port和type。ip\_port是一个包含IP地址和端口号的元组，type表示日志的类型，例如sender或receiver。 函数首先创建一个名为file\_transfer的日志记录器logger，并设置其日志级别为DEBUG。

然后，函数创建一个文件处理器fh，并设置其日志级别为DEBUG。文件处理器的文件名由IP地址、端口号、类型和当前时间组成，例如127.0.0.1\_41099\_sender\_20240101123030.log。

在发送端和接收端分别设置一个日志来记录相关信息。在发送方记录了发送方的IP地址和端口号、接收方的IP地址和端口号、要发送的文件名、数据包的大小和总数量，主体部分记录了发送数据包的序列号、发送状态和已经确认的数据包序号。发送日志的结束部分记录了总的数据包数量、总的传输时间、超时的次数、重传的次数以及传输是否成功。

在接收方，日志的开始部分记录了接收方的IP地址和端口号，以及要接收的文件名，主体部分记录了每个数据包的接收情况。每一行对应一个数据包，包含：序号、期望接收的数据包的序列号、实际接收的数据包的序列号和数据包的状态，可能的值有OK、DataErr和NoErr。OK表示数据包的序列号与期望的序列号匹配，DataErr表示数据包的数据有错误，NoErr表示数据包的序列号与期望的序列号不匹配，但数据包没有错误，接收日志的结束部分记录了文件接收是否成功

**配置文件和参数：**

配置文件为config.ini，采用键值对的方式存储配置信息：

# 发送方 IP 地址

SenderHost=localhost

# 发送方 UDP 端口

SenderPort=41098

# 接收方 IP 地址

ReceiverHost=localhost

# 接收方 UDP 端口

ReceiverPort=41099

# PDU 中数据字段的长度（单位：字节）

DataSize=1024

# PDU 错误率(n%)

ErrorRate=0.1

# PDU 丢包率(n%)

LostRate=0.1

# 发送窗口大小

SWSize=10

# 起始 PDU 的序号

InitSeqNo=0

# 超时定时器值（单位：毫秒）

Timeout=1000

1. **Development and Implementation**

操作系统：Windows

使用语言：Python

开发工具：PyCharm

依赖库：os、socket、threading、random、time、datetime、logging

项目结构：

Sender.py：负责发送文件。使用滑动窗口协议进行可靠的数据传输。send函数是主函数，它读取文件，将其分解为数据包，然后将它们发送给接收者。sender\_receive函数处理从接收者收到的确认信息。

def send(sock, filename, send\_ip\_port, recv\_ip\_port):  
 global dur  
 global status  
 global mutex  
 global ack\_exp  
 global num\_packets  
 global send\_count  
 global resend  
  
 TO\_COUNT = 0  
 RT\_COUNT = 0  
 Total\_COUNT = 0  
 scale = 100  
 start\_time = time.time()  
 timeout = float(config['Timeout']) / 1000  
 send\_timer = utils.timer(timeout)  
 logger = utils.setup\_logger(send\_ip\_port, 'Sender')  
 file = open(filename, "rb")  
 packets = []  
 seq\_num = int(config['InitSeqNo'])  
 window\_size = int(config['SWSize'])  
 next\_frame\_to\_send = int(config['InitSeqNo'])  
  
 logger.info("------------------------Sender------------------------")  
 logger.info(f"Sender: {send\_ip\_port[0]}:{str(send\_ip\_port[1])}")  
 logger.info(f"Receiver: {recv\_ip\_port[0]}:{str(recv\_ip\_port[1])}")  
 logger.info(f"Sending file: {filename}")  
  
 while True:  
 data = file.read(data\_size)  
 if not data:  
 break  
 crc\_num = crc16.crc16xmodem(data)  
 pdu = PDU.generate(seq\_num, crc\_num, data)  
 packets.append(pdu)  
 seq\_num += 1  
  
 num\_packets = len(packets)  
  
 logger.info(f'Packet size: {data\_size} Bytes, Total packets: {num\_packets}')  
 logger.info("------------------------Sender------------------------\n")  
 print(f'Packet size: {data\_size} Bytes, Total packets:{num\_packets}')  
  
 t = threading.Thread(target=sender\_receive, args=(sock,))  
 t.start()  
 overtime = False  
 start = time.perf\_counter()  
 pre = start  
 while ack\_exp < len(packets):  
 mutex.acquire()  
 while next\_frame\_to\_send < ack\_exp + window\_size:  
 send\_count += 1  
 if next\_frame\_to\_send >= len(packets):  
 break  
 if overtime is False:  
 if resend is False:  
 status = 'New'  
 logger.info(f"{send\_count}, pdu\_to\_send={next\_frame\_to\_send}"  
 f", status={status}, ackedNo={ack\_exp}\n")  
 else:  
 status = 'RT'  
 logger.info(f"{send\_count}, pdu\_to\_send={next\_frame\_to\_send}"  
 f", status=New, ackedNo={ack\_exp}\n")  
 send\_count += 1  
 logger.info(f"{send\_count}, pdu\_to\_send={next\_frame\_to\_send}"  
 f", status={status}, ackedNo={ack\_exp}\n")  
 RT\_COUNT += 1  
 elif overtime is True:  
 status = 'TO'  
 logger.info(f"{send\_count}, pdu\_to\_send={next\_frame\_to\_send}"  
 f", status={status}, ackedNo={ack\_exp}\n")  
 TO\_COUNT += 1  
  
 send\_timer.start(next\_frame\_to\_send)  
 udt.send(packets[next\_frame\_to\_send], sock, recv\_ip\_port)  
 next\_frame\_to\_send += 1  
  
 Total\_COUNT = send\_count  
 overtime = False  
 if send\_timer.overtime(ack\_exp):  
 overtime = True  
 next\_frame\_to\_send = ack\_exp  
  
 if (time.perf\_counter() - pre) > 1:  
 pre = time.perf\_counter()  
 param = num\_packets / scale  
 i = int(next\_frame\_to\_send / param + 1)  
 a = '\*' \* i  
 b = '.' \* (scale - i)  
 c = (i / scale) \* 100  
 dur = pre - start  
 print("\rSending {:^3.0f}% [{}=>{}] Time: {:.2f} secs".format(c, a, b, dur), end='')  
 mutex.release()  
  
 print("\rSend Over {:^3.0f}% [{}] Time: {:.2f} secs".format(100, '\*' \* scale, dur), end='')  
 print("\n-------------------------Send succeed!-------------------------")  
 udt.send(PDU.final\_packet(), sock, recv\_ip\_port)  
 end\_time = time.time()  
 Total\_time = end\_time - start\_time  
 logger.info(f"Total packets: {num\_packets}")  
 logger.info(f"Total time: {round(Total\_time, 2)} seconds")  
 logger.info(f"TO count: {TO\_COUNT}")  
 logger.info(f"RT count: {RT\_COUNT}")  
 logger.info("Success.")  
 file.close()

Receiver.py：接收方，定义了接收者类，负责接收数据包，并将数据包的内容写入文件。receive函数通过接收UDP socket中传来的数据包，将其分解为序列号、CRC校验和和数据，再根据计算来判断数据是否正确，并发送相应的确认信息给发送端。

class Receiver:  
 def \_\_init\_\_(self):  
 self.error\_rate = float(config['ErrorRate']) / 100  
 self.lost\_rate = float(config['LostRate']) / 100  
 self.data\_size = int(config['DataSize'])  
  
 def receive(self, sock, filename, IP\_PORT):  
 udt = utils.UDP(self.lost\_rate, self.error\_rate, self.data\_size)  
 logger = utils.setup\_logger(IP\_PORT, 'Receiver')  
 file = open(filename, "wb")  
 logger.info("------------------------Receiver------------------------")  
 frame\_expected = int(config['InitSeqNo'])  
 logger.info(f"Receiver: {IP\_PORT[0]}:{str(IP\_PORT[1])}")  
 logger.info(f"Receiving file: {filename}")  
 logger.info("------------------------Receiver------------------------\n")  
 recv\_count = 0  
  
 while True:  
 recv\_count += 1  
 pdu, addr = udt.receive(sock)  
  
 if not pdu:  
 break  
 seq\_num, crc\_num, data = PDU.split(pdu)  
  
 print('\nPDU:', seq\_num)  
  
 crc\_expected = crc16.crc16xmodem(data)  
 if crc\_expected != crc\_num:  
 logger.info(f"{recv\_count}, pdu\_exp={seq\_num}, pdu\_recv={frame\_expected}, status=DataErr\n")  
 print("DataErr data")  
 udt.send\_nak(-frame\_expected, sock, addr)  
 continue  
  
 if seq\_num == frame\_expected:  
 print('Correct data')  
 logger.info(f"{recv\_count}, pdu\_exp={seq\_num}, pdu\_recv={frame\_expected}, status=OK\n")  
 print('ACK:', frame\_expected)  
 udt.send\_ack(frame\_expected, sock, addr)  
 frame\_expected += 1  
 file.write(data)  
  
 else:  
 print('NoErr data')  
 logger.info(f"{recv\_count}, pdu\_exp={seq\_num}, pdu\_recv={frame\_expected}, status=NoErr\n")  
 print('ACK:', frame\_expected-1)  
 udt.send\_ack(frame\_expected - 1, sock, addr)  
  
 print("\n-------------------------Receive succeed!-------------------------")  
 logger.info("Success.")  
 file.close()

start.py：系统入口，它创建并绑定UDP socket，然后根据用户的选择，启动一个新的线程来发送或接收文件。

utils.py：工具类，包含一些工具函数，例如读取配置文件的函数，以及用于计算CRC校验的函数。

工具类的主要函数：

read\_config():读取配置文件的。它接受一个参数config\_file，这个参数是配置文件的路径。函数首先创建一个空的字典config，然后打开配置文件进行读取。对于文件中的每一行，如果这一行以#开头或者是空行，那么就跳过这一行。否则，将这一行按照=分割成键和值，然后将键和值去掉前后的空格，存入config字典中。最后，函数返回config字典。

def read\_config(config\_file):  
 config = {}  
 with open(config\_file, 'r', encoding='utf-8') as file:  
 for line in file:  
 if line.startswith('#') or not line.strip():  
 continue  
 key, value = line.strip().split('=')  
 config[key.strip()] = value.strip()  
 return config

setup\_logger():设置日志记录器的。它接受两个参数，ip\_port和type。函数首先创建一个名为file\_transfer的日志记录器，并设置其日志级别为DEBUG。然后，创建一个文件处理器fh，文件名由ip\_port、type和当前的日期时间组成，文件处理器的日志级别也设置为DEBUG。接着，创建一个格式器，并设置到文件处理器上。最后，将文件处理器添加到日志记录器上，并返回日志记录器。

def setup\_logger(ip\_port, type):  
 logger = logging.getLogger('file\_transfer')  
 logger.setLevel(logging.DEBUG)  
  
 fh = logging.FileHandler(f'{ip\_port[0]}\_{ip\_port[1]}\_{type}\_{datetime.now().strftime("%Y%m%d%H%M%S")}.log')  
 fh.setLevel(logging.DEBUG)  
  
 formatter = logging.Formatter('%(asctime)s - %(message)s')  
 fh.setFormatter(formatter)  
  
 logger.addHandler(fh)  
  
 return logger

timer: 计时器。它有一个构造函数，接受一个参数timeout，表示超时时间。类中有一个start方法，接受一个参数seq，表示序列号，方法将当前的时间存入timer字典中，键为序列号。类中还有一个get\_time方法，返回当前的时间。类中的overtime方法接受一个参数seq，表示序列号，方法返回当前时间与timer字典中对应序列号的时间之差是否大于超时时间。

class timer:  
 timer = {}  
  
 def \_\_init\_\_(self, timeout):  
 self.start\_time = -1  
 self.timeout = timeout  
  
 def start(self, seq):  
 self.timer[seq] = time.time()  
  
 def get\_time(self):  
 return time.time()  
  
 def overtime(self, seq):  
 if seq >= len(self.timer):  
 seq -= 1  
 return True if time.time() - self.timer[seq] > self.timeout else False

PDU: 它有一个构造函数，接受三个参数，lost表示丢包率，err表示错误率，data\_size表示数据大小。类中有一个send方法，接受三个参数，packet表示要发送的数据包，sock表示套接字，addr表示地址，方法首先检查是否需要产生错误，如果需要，就对数据包产生错误，然后检查是否需要丢包，如果不需要，就发送数据包。类中还有一个receive方法，接受一个参数sock表示套接字，方法接收一个数据包并返回。类中的send\_ack方法和send\_nak方法分别用来发送ACK和NAK，它们的参数和send方法相同。类中的receive\_ack方法接受一个参数sock表示套接字，方法接收一个ACK并返回。类中的error方法接受一个参数packet表示数据包，方法对数据包产生错误并返回。

class PDU:  
 def \_\_init\_\_(self, lost, err, data\_size):  
 random.seed(time.time())  
 self.prob\_lost = lost  
 self.prob\_err = err  
 self.prob\_lost\_ = lost\*c  
 self.data\_size = data\_size  
  
 def send(self, packet, sock, addr):  
 # 错误  
 if random.random() < self.prob\_err:  
 packet = self.error(packet)  
 # data 丢包  
 if random.random() > self.prob\_lost:  
 sock.sendto(packet, addr)  
 return  
  
 def receive(self, sock):  
 packet, addr = sock.recvfrom(self.data\_size \* 2)  
 return packet, addr  
  
 def send\_ack(self, ack, sock, addr):  
 ack\_bytes = ack.to\_bytes(4, byteorder='big', signed=True)  
 # ack 丢包  
 if random.random() > self.prob\_lost:  
 sock.sendto(ack\_bytes, addr)  
 return  
  
 def send\_nak(self, nak, sock, addr):  
 nak\_bytes = (-nak).to\_bytes(4, byteorder='big', signed=True)  
 if random.random() > self.prob\_lost\_:  
 sock.sendto(nak\_bytes, addr)  
 return  
  
 def receive\_ack(self, sock):  
 ack\_bytes, addr = sock.recvfrom(self.data\_size \* 2)  
 ack = int.from\_bytes(ack\_bytes, byteorder='big', signed=True)  
 return ack, addr  
  
 def error(self, packet):  
 ErrData = b''  
 for i in range(len(packet) - 16):  
 byte = random.randint(0, 127)  
 ErrData = ErrData + byte.to\_bytes(1, byteorder='big', signed=True)  
 return packet[0:16] + ErrData

config.ini：系统配置文件，包含了发送方和接收方的IP地址和端口号，数据帧的大小、错误率、丢包率、发送窗口大小、起始数据帧序号、超时阈值等信息。

1. **System Deployment, Startup, and Use**

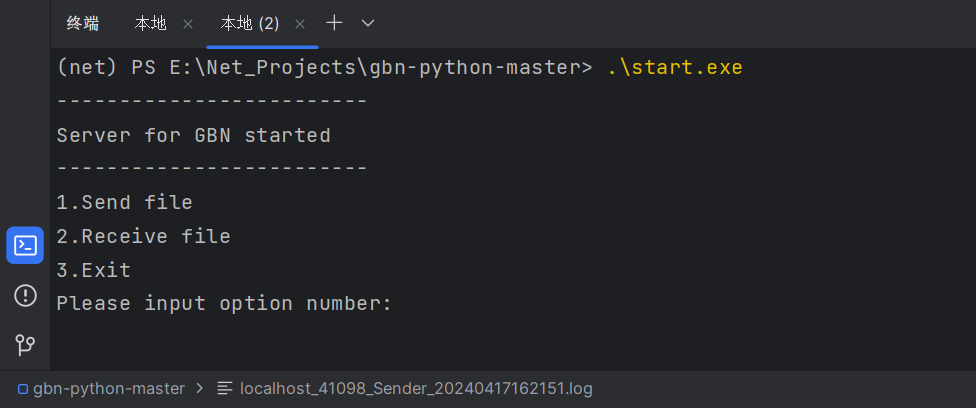
**请确保config.ini和start.exe在同一个文件夹下！**

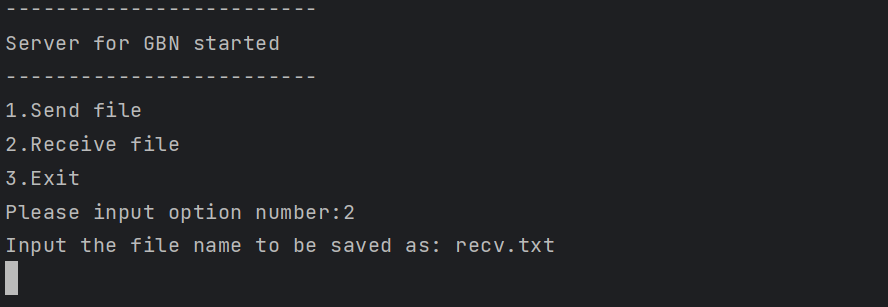
**环境部署：**

Python：>= 3.6

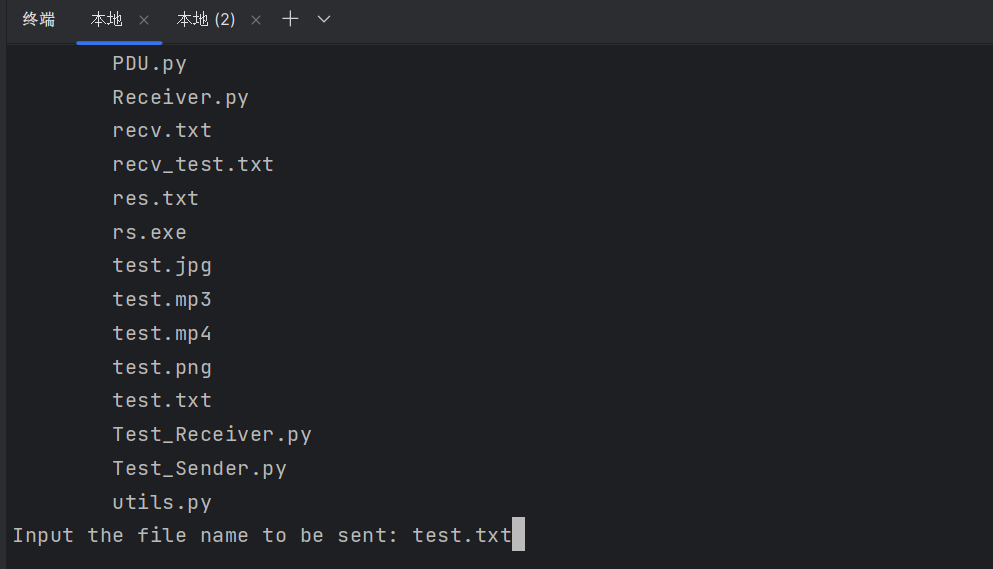
操作系统：Windows

**启动项目：**

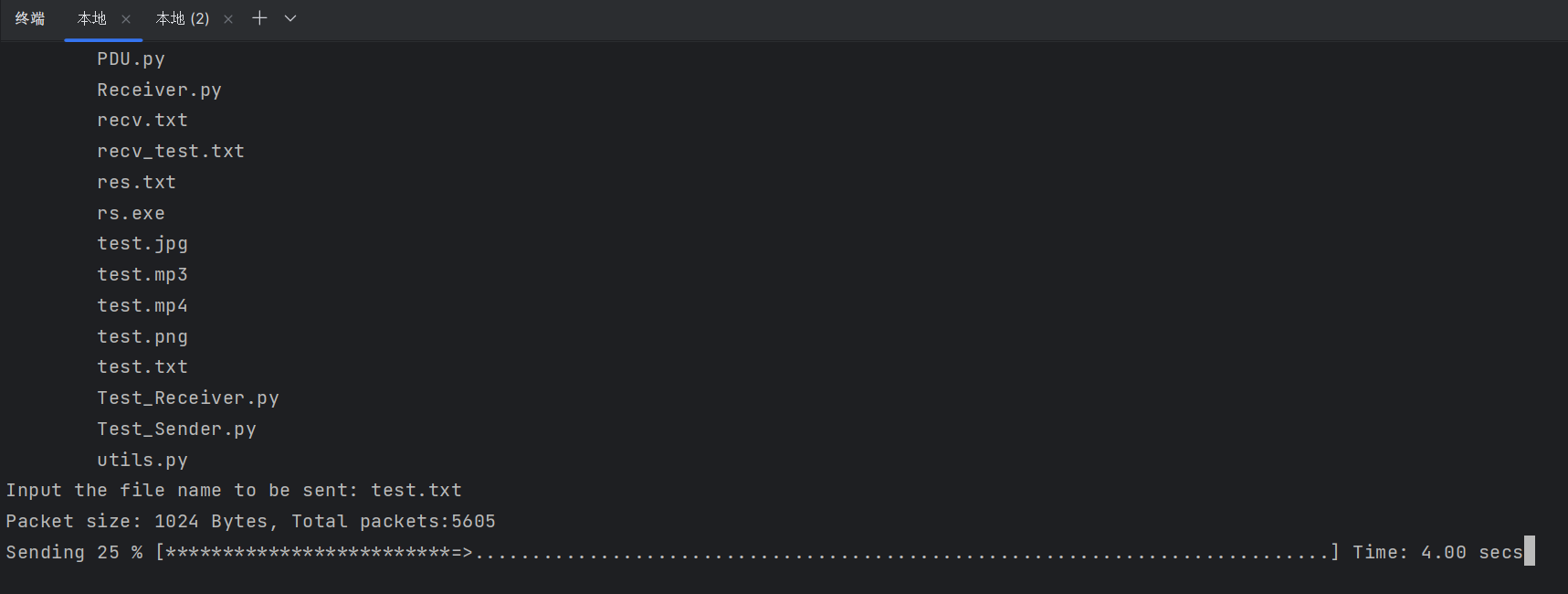
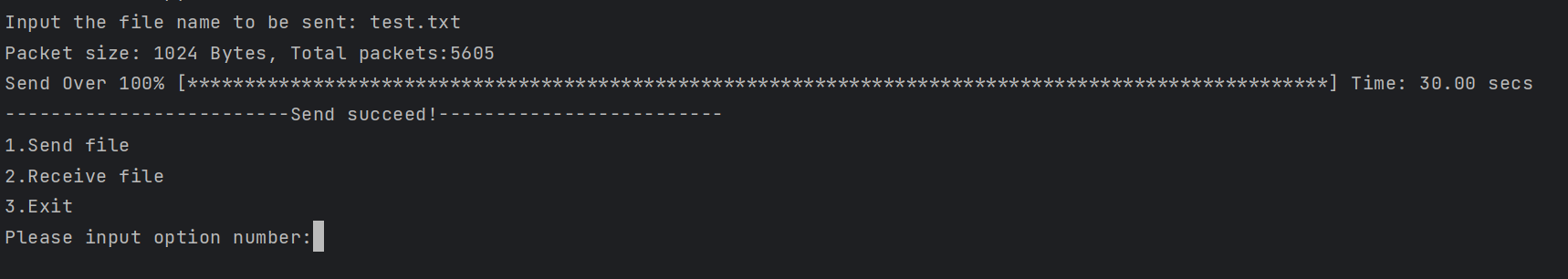
1. 分别在发送方和接收方通过控制台终端启动exe文件
2. 确保收发双方能互相ping通
3. 先在接收方指定保存文件的名称，启动接收方对socket的监听



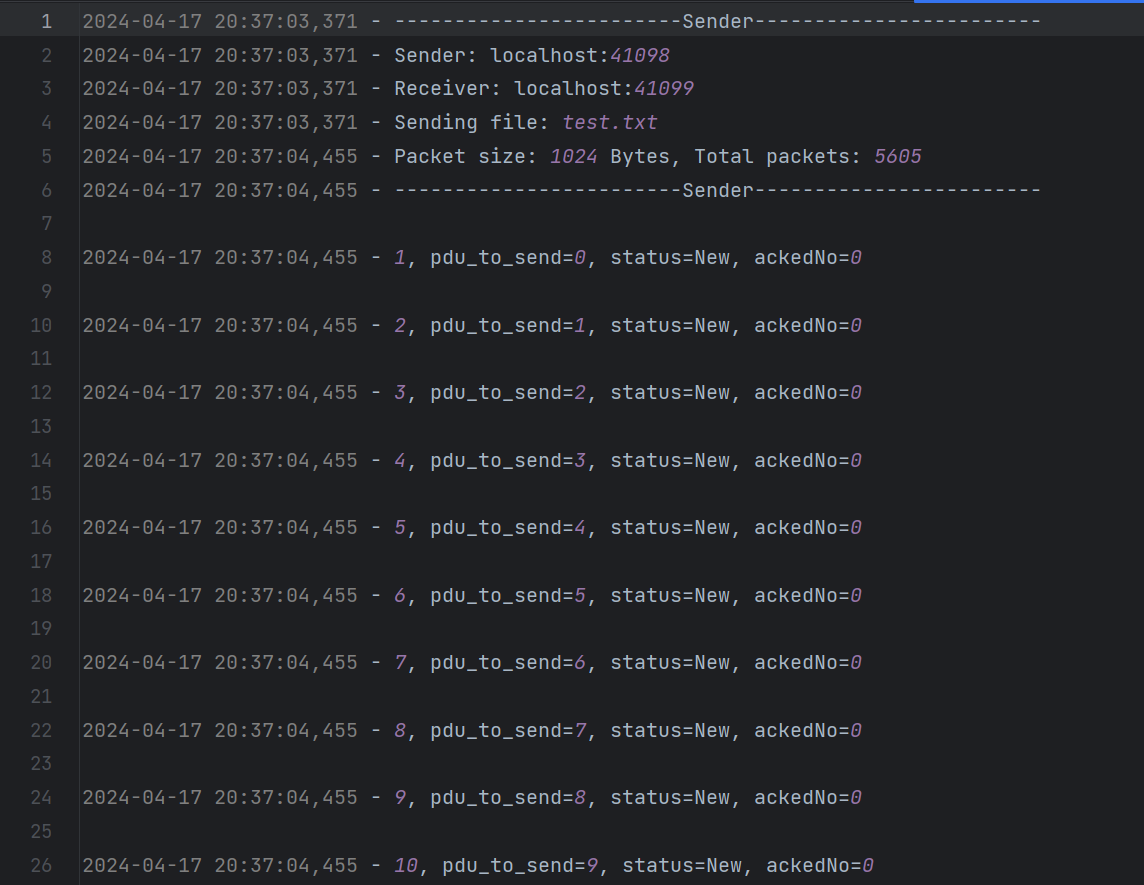
1. 在发送方指定发送文件的名称或完整路径名，即可启动发送

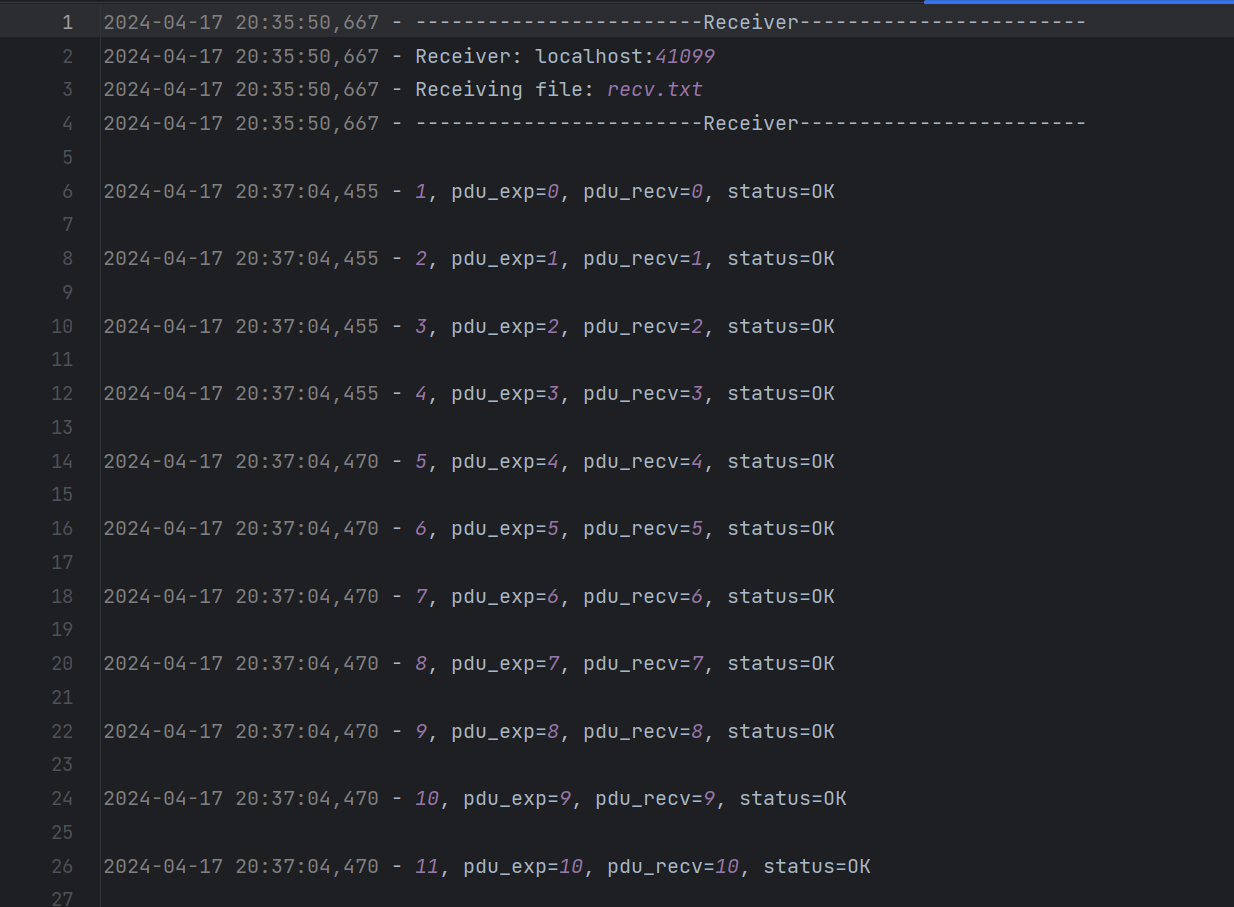


1. 等待发送进度条走满，表示发送成功



1. 生成了发送和接收两个日志文件，里面记录了每个数据包发送的具体信息以及整体统计信息





1. **System Test**

**单元测试：**

**测试工具：unittest**

**针对发送端的测试类：**

class TestSender(unittest.TestCase):  
 def test\_send(self):  
 sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  
 Sender\_IP\_PORT = ('localhost', 12345)  
 Receiver\_IP\_PORT = ('localhost', 12346)  
 sock.bind(Sender\_IP\_PORT)  
 filename = 'test.txt'  
 send\_thread = threading.Thread(target=Sender.send, args=(sock, filename, Sender\_IP\_PORT, Receiver\_IP\_PORT))  
 send\_thread.start()  
 send\_thread.join()

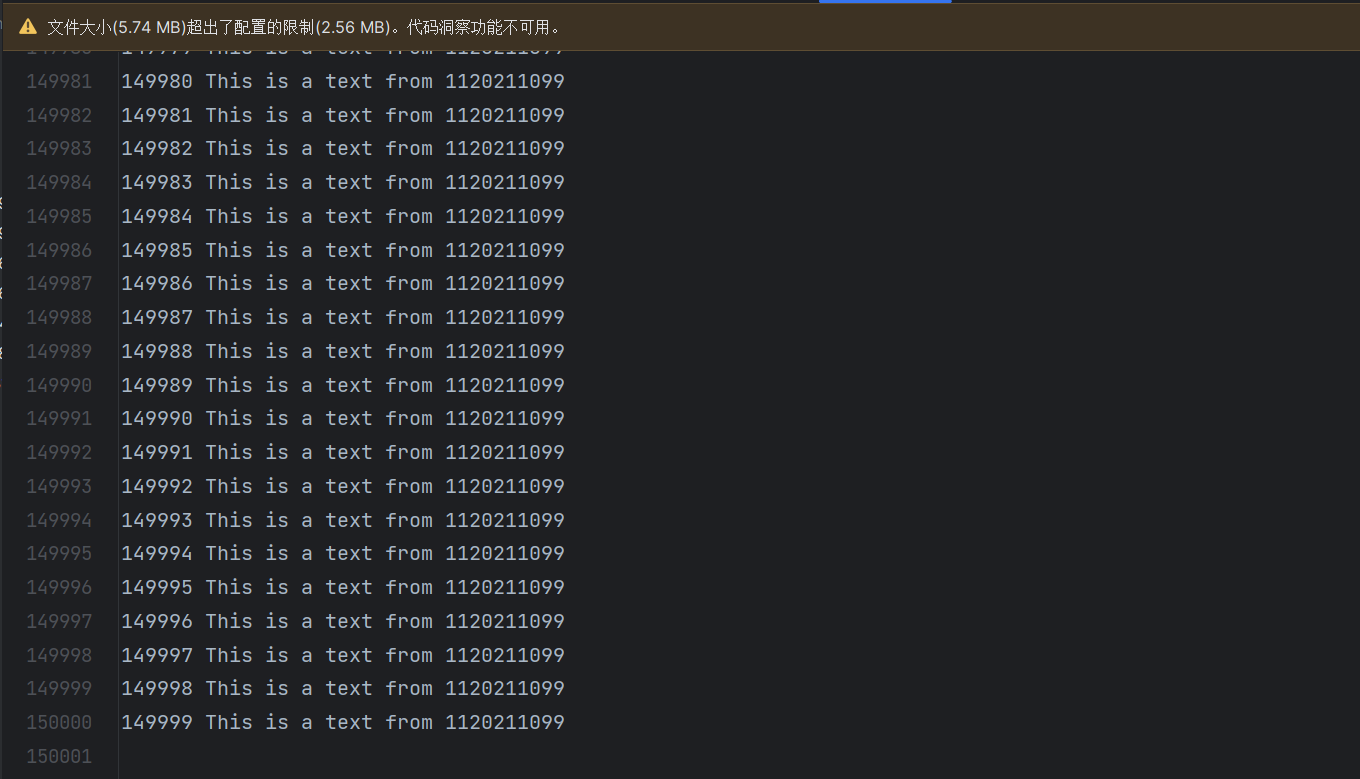
**针对接收端的测试类：**

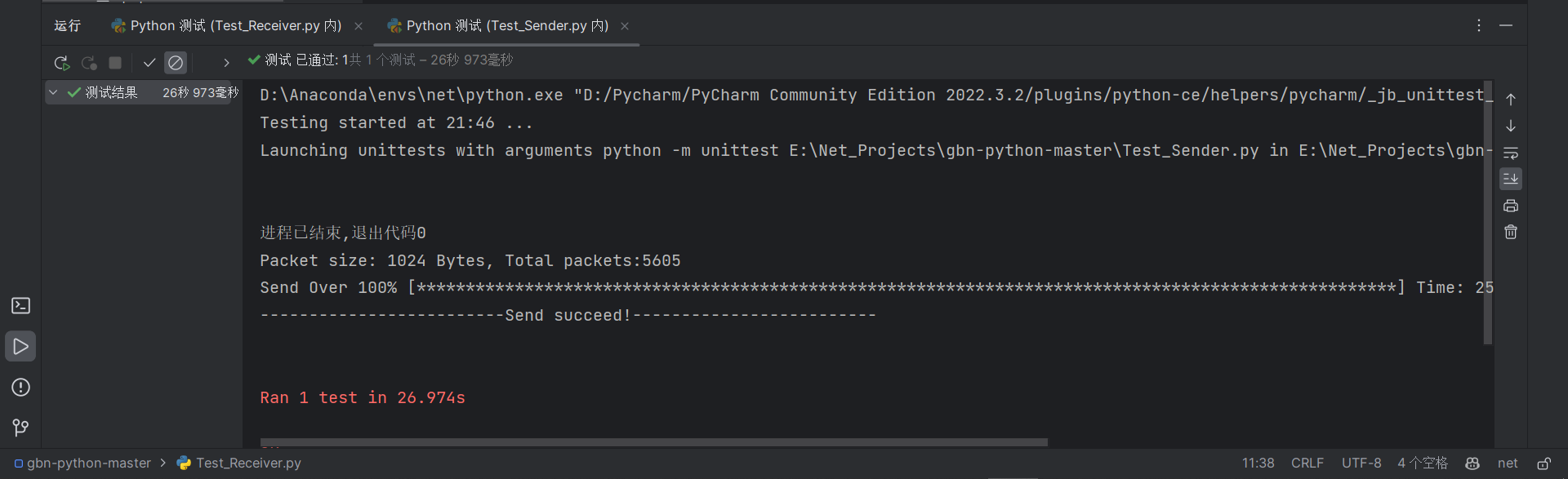
class TestSender(unittest.TestCase):  
 def test\_receive(self):  
 sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)  
 Receiver\_IP\_PORT = ('localhost', 12346)  
 sock.bind(Receiver\_IP\_PORT)  
 filename = 'recv\_test.txt'  
 receiver = Receiver.Receiver()  
 receive\_thread = threading.Thread(target=receiver.receive, args=(sock, filename, Receiver\_IP\_PORT))  
 receive\_thread.start()  
 receive\_thread.join()

**测试文件类型：txt**

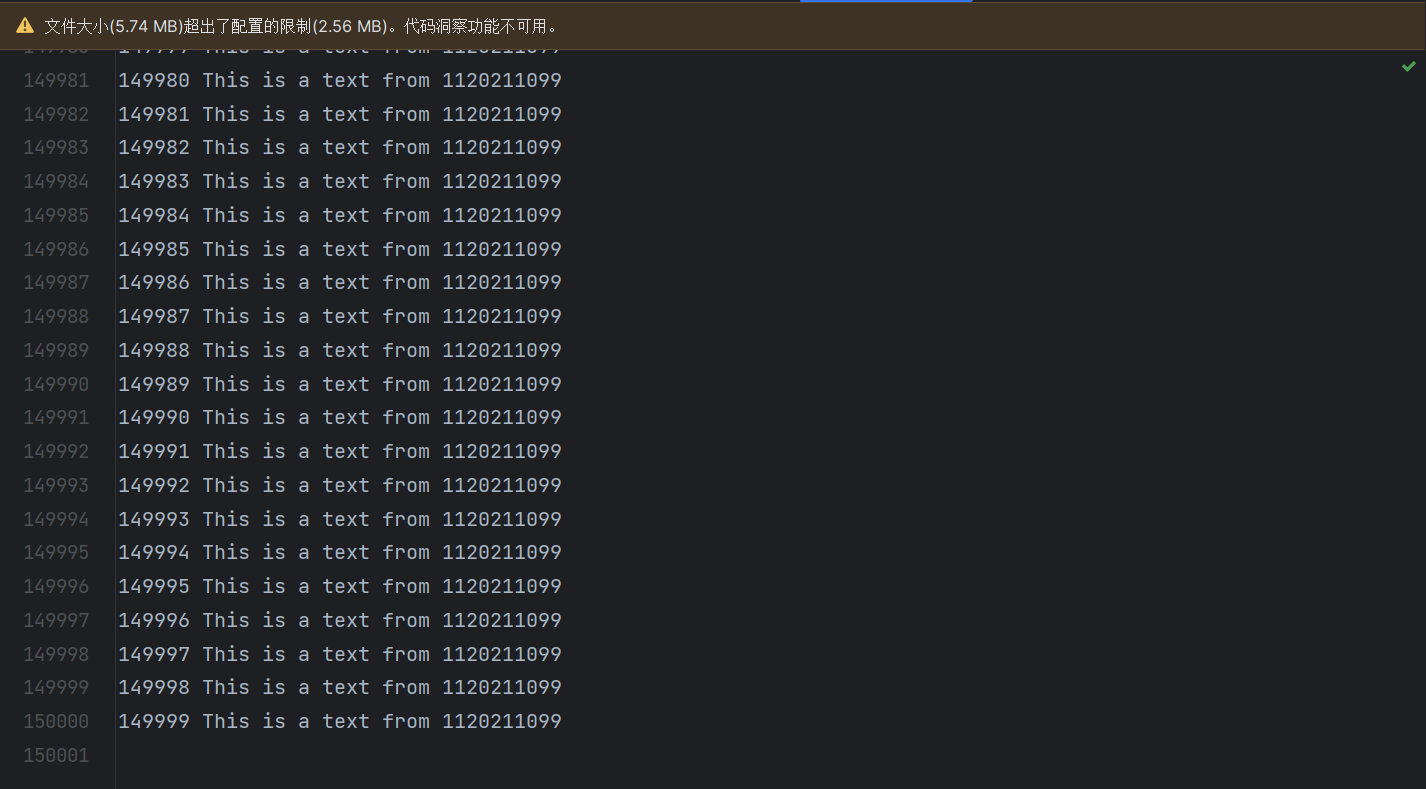
**测试结果：成功发送和接收**

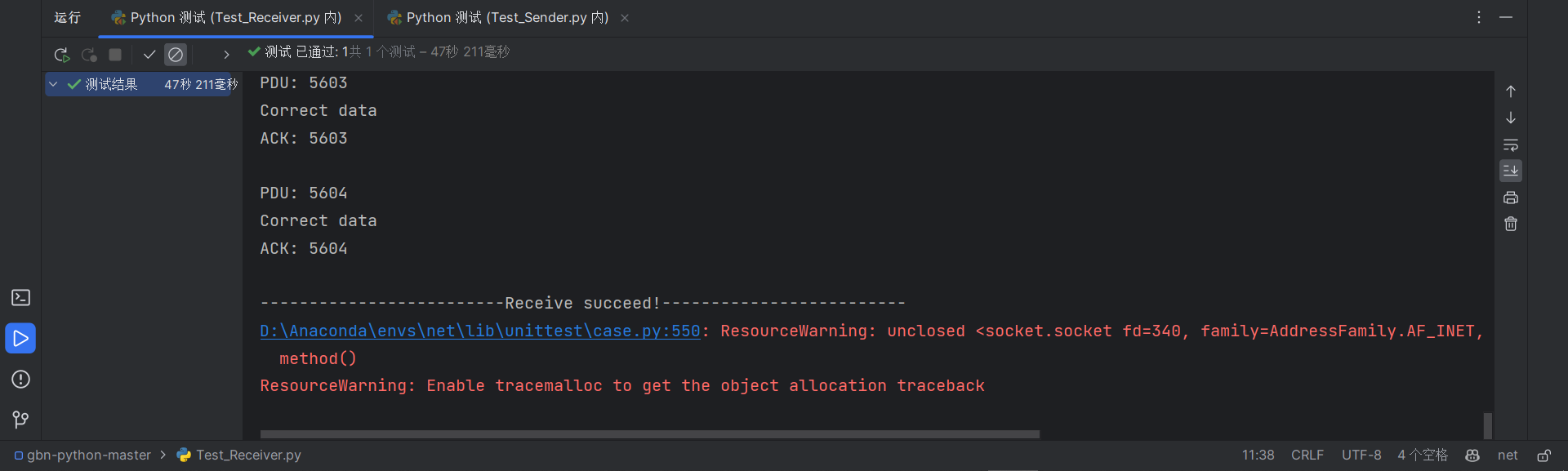
**发送端：**





**接收端：**





**集成测试（1）：**

**测试的相关参数设置：**

传输文件类型：mp4

传输文件大小：7.16MB

发送方IP：localhost

发送方端口号：41098

接收方IP：localhost

接收方端口：41099

数据帧大小：1024B

错误率：0.1%

丢包率：0.1%

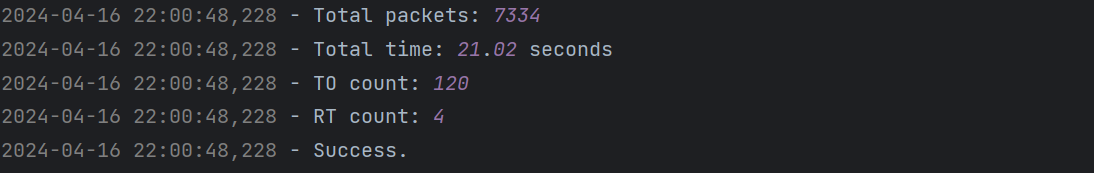
窗口大小：10

其实PDU序号：0

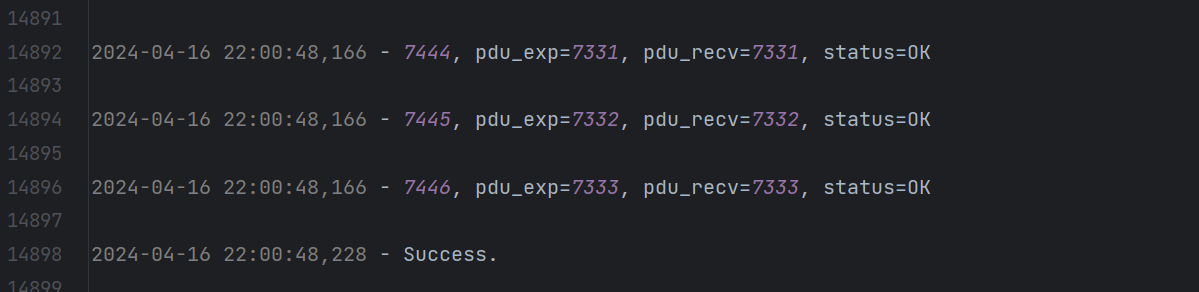
定时器设定：1000ms

**测试结果：**

**发送端：**



**接收端：**



**集成测试（2）**

传输文件类型：exe

传输文件大小：3.66MB

发送方IP：localhost

发送方端口号：41098

接收方IP：localhost

接收方端口：41099

数据帧大小：2048B

错误率：1%

丢包率：1%

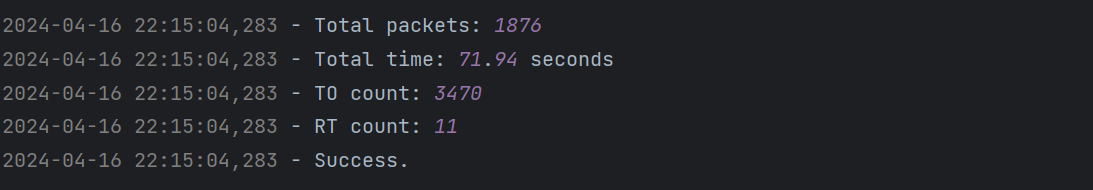
窗口大小：50

起始PDU序号：0

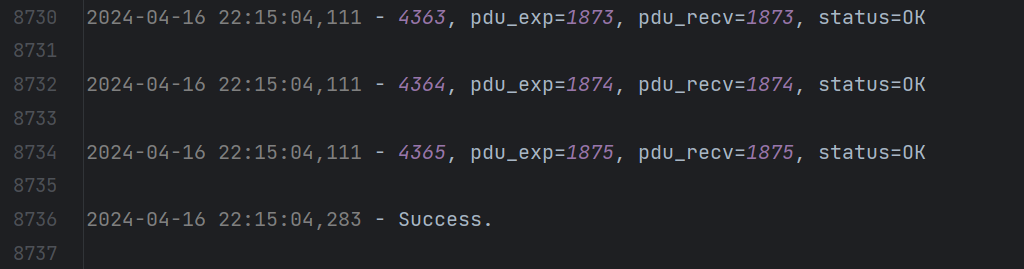
定时器设定：1000ms

**测试结果：**

**发送端：**



**接收端：**



1. **Performance and Analysis**

**统一设定**：

发送方IP：localhost

发送方端口：41098

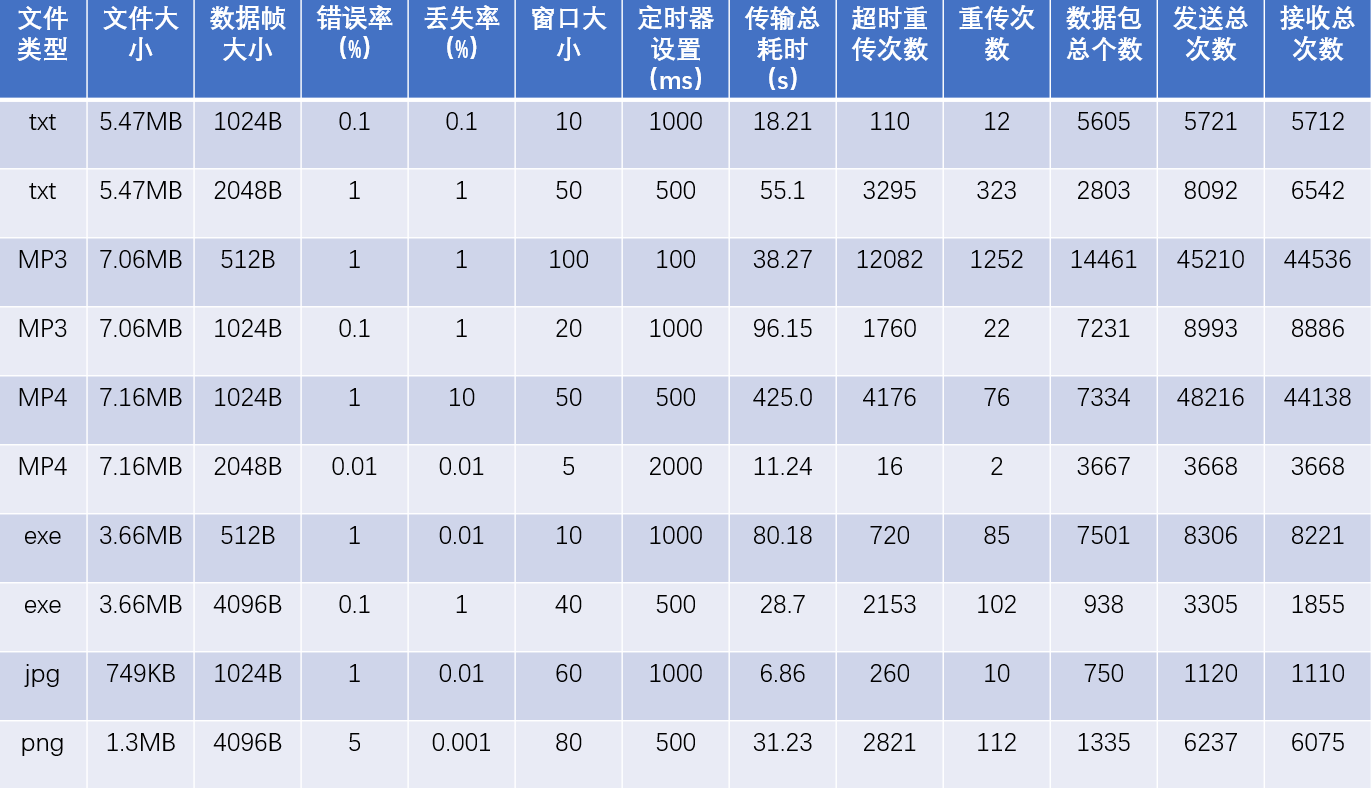
接收方IP：localhost

接收方端口：41099

起始PDU序号：0

**结果分析：**

**对于不同类型文件，结果如下：**



**对于同种类型文件，分别调整各项设置的结果如下：**

**标准设置：**

**数据帧大小：1024B**

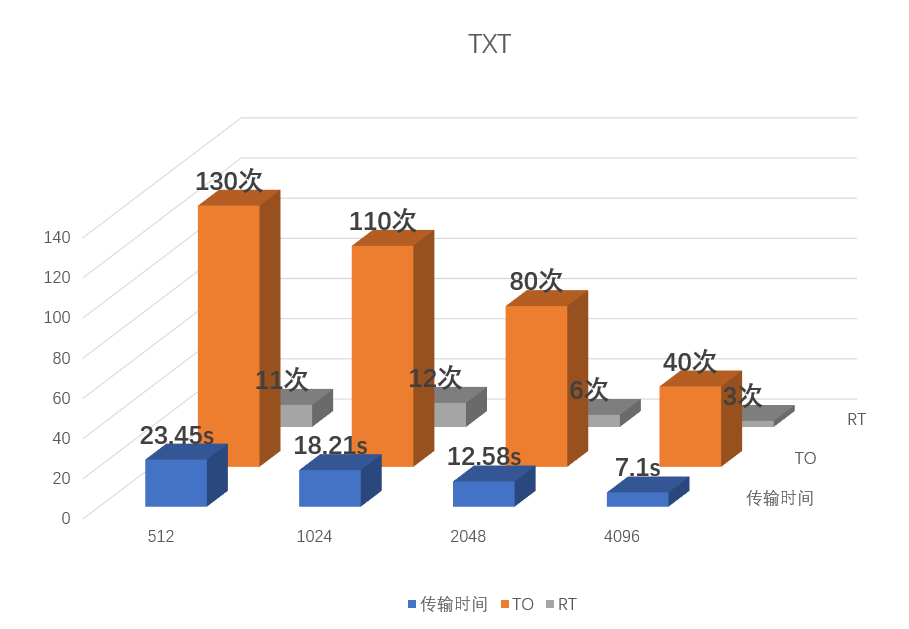
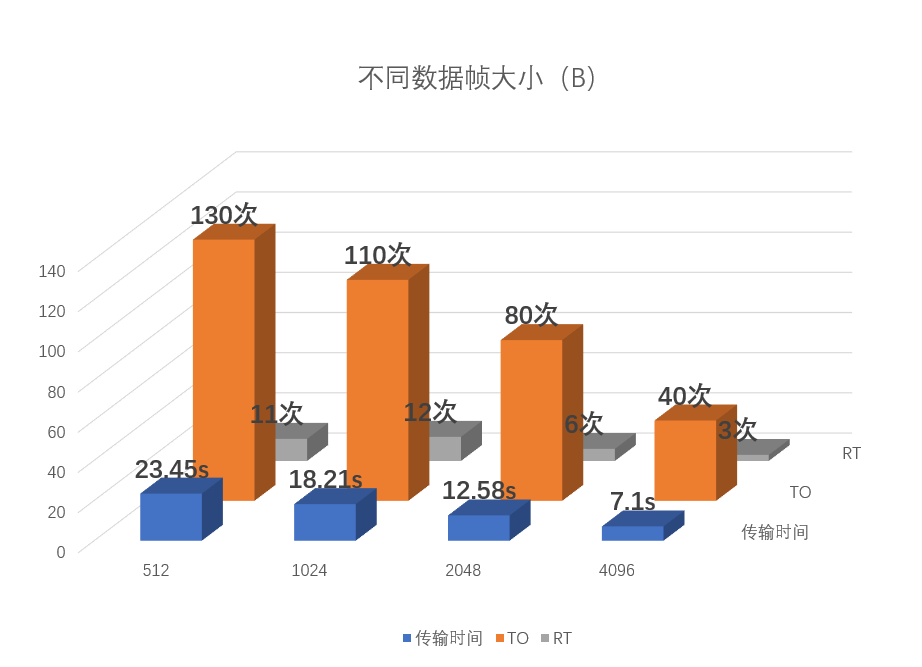
**错误率：0.1%**

**丢包率：0.1%**

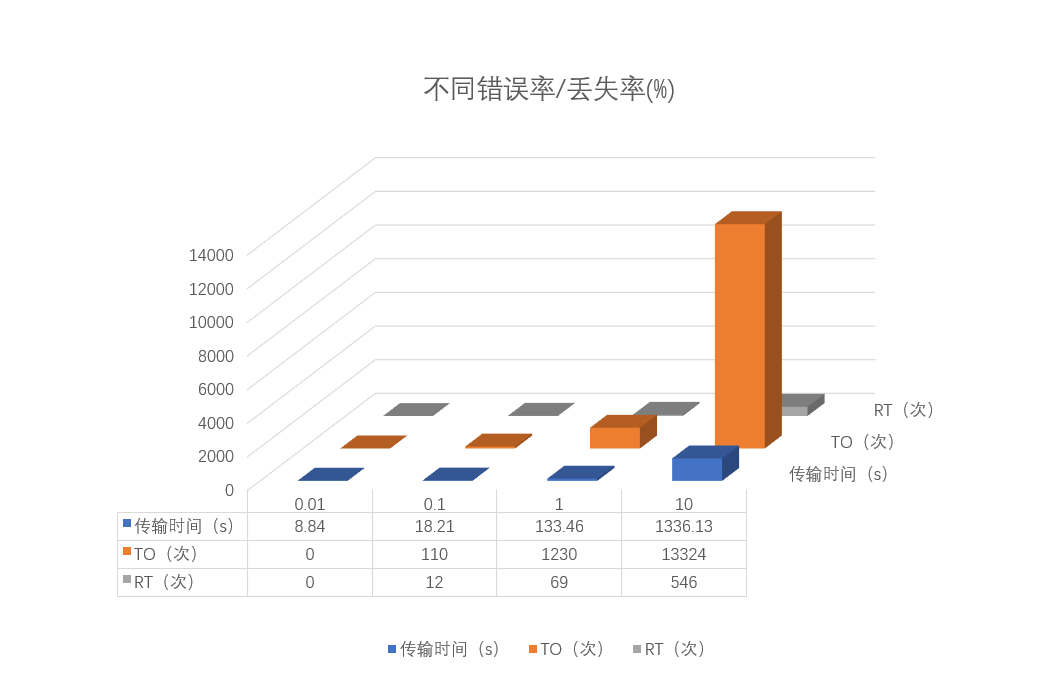
**发送窗口大小：10**

**定时器设置：1000ms**

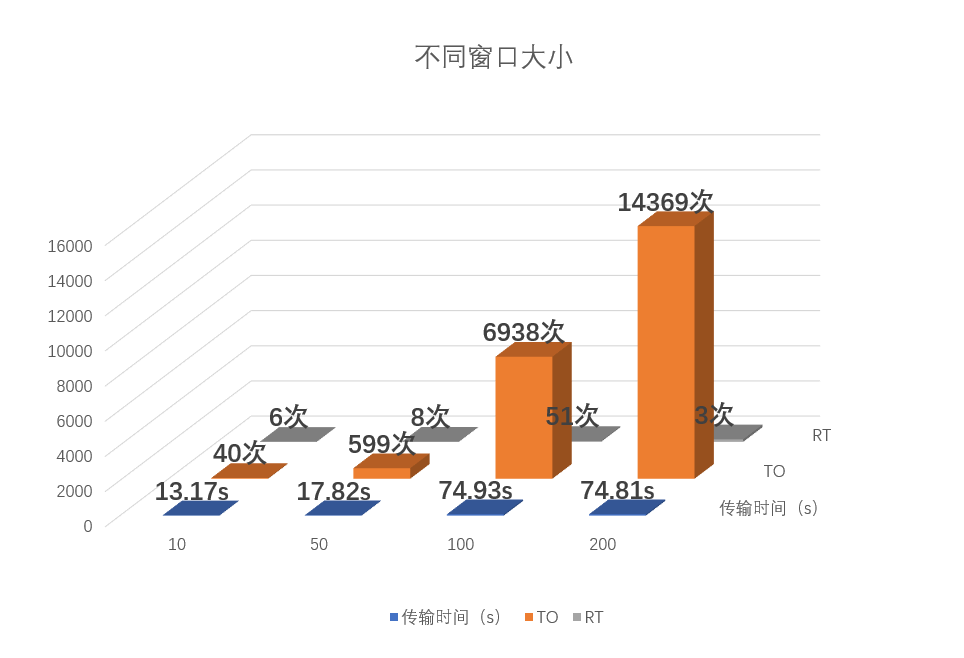
1. **不同数据帧大小：**

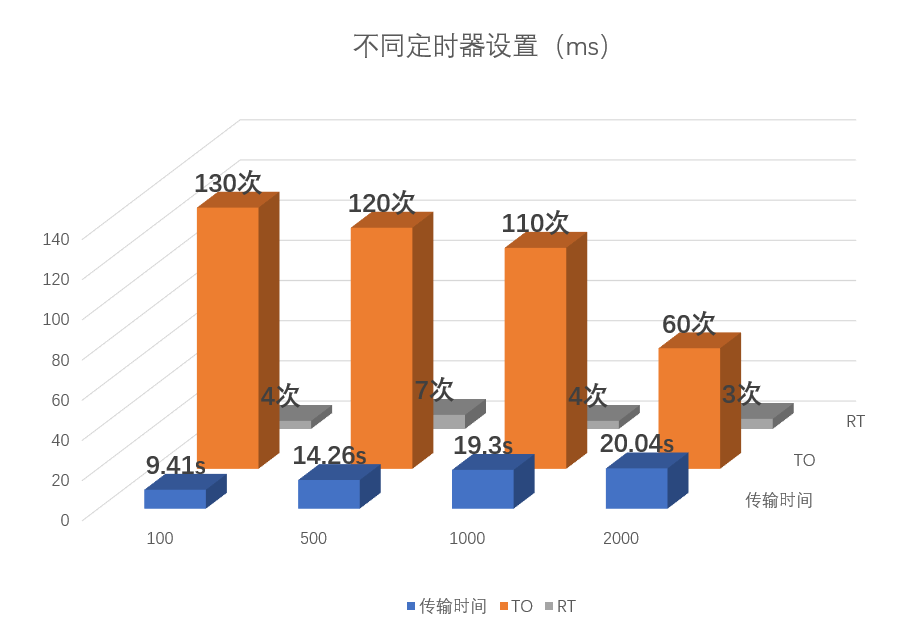


1. **不同错误率和丢包率：**



1. **不同窗口大小：**



1. **不同定时器设置：**

**结果分析：**

1. **当数据帧大小增加，传输时间变长，重传次数变少。这是因为更大的数据帧需要更长的时间来进行处理和传输，同时在接收方的确认等待时间也增加，如果发生了数据错误或者丢失，那么重传时间也较久，因此传输时间变长。另一方面，由于数据帧大小变大，那么数据包个数相应的减少，总发送次数减少，在错误率和丢包率不变的情况下，重传次数相应的也会减少。**
2. **当错误率和丢包率成10倍增长时，传输时间的延长显而易见，同时重传次数和超时重传次数也增加明显。当错误率达到10%的时候，已经发生了十分严重的网络传输问题，传输延迟已经无法让人接收。**
3. **当滑动窗口的大小增大后，传输时间变长。这可能是因为发送端发送数据的速度超过了接收端接收数据的速度，导致过长的数据在发送端排队，造成延迟累积。**
4. **当超时定时器的设定值增加，传输时间变长。这是因为当超时的阈值提升，发送端等待确认的时间也相应的增加，所以同时也造成了重传次数的减少。**

1. **Summary or Conclusions**

**研究与分析的问题:**

采用GBN协议，实现两台主机之间大文件的可靠传输。

**需要解决的问题：**

主机的通讯连接、重传机制、超时机制、确认管理、丢包处理、错误模拟、错误恢复、滑动窗口设置等等。

**系统设计方案：**

开发系统：Windows

开发语言：Python

开发工具：PyCharm

设计思想：分别为发送方和接收方视作一个对象类，为其编写对应的发送和接收函数。对于数据帧也封装为一个类，定义其属性和相应的方法。另外对于整个系统，编写一个入口程序，用于启动发送或者接收线程，并完成前期主机间通信的工作。对于整个系统中重复使用的一些方法，我将它们都放到一个工具类，提高代码的可重用性。例如配置文件的读取和日志的撰写等。最后利用python的pyinstaller库将整个系统打包为exe文件，方便运行和展示。由于系统是基于控制台操作，所以相应的提示和传输过程的展现全部都放在了控制台。

**一些问题与解决方法：**

1. 数据包的错误模拟。对于项目要求的CRC-CCITT校验规则，我在刚开始还不懂如何很好的实现，没有将其与数据帧的处理很好的结合起来。我最开始对于错误的模拟是直接将数据设置为空，然后会在接收方特地判空来处理，但是这样就失去了真实性和通用性，因为非空数据也会出现DataErr。后来弄明白了校验规则，就将错误模拟的方式改为保留部分原有数据，剩下部分填充随机数据，在接收方那边通过设置好的校验规则来判断数据是否正确，这样就没什么问题了。
2. 发送端的线程同步问题。在发送端，发送者需要同时做数据发送和接收ACK两个工作，而这两个方法对于“已确认数据帧序列号”这个变量的访问可能会导致不一致问题，因此我采用了加锁机制，分别在发送过程和接收ACK过程加互斥锁，通过牺牲一定的性能来保证发送数据和重传数据的顺序无误。
3. 数据的读取和发送。在发送端，我最开始考虑的是读一个数据帧就加入窗口，然后开始处理，但是当丢包需要重传数据时，发数据和读数据的线程并发会导致很多的问题，基本上传输过程就无法继续下去了。所以我将其改为先统一读完文件的所有内容，存入列表，再遍历进行处理。

**实验结果与系统表现**：

详细的结果以及不同影响因素的对比在上面已经展示了，在这里只做简要的总结。总体来看，在错误率和丢包率不超过5%的情况下，传输的效率还是能够让人接受的，当然越低表现越好，当低至0.01%的时候甚至基本不会发生重传现象，而高达10%的时候，在极少数情况下会发送失败。

**系统的主要特点：**

1. 采用GBN协议实现可靠传输
2. 可以通过配置文件改变参数，观察不同参数对于系统的影响
3. 会为发送和接收双方各生成一个日志文件，里面详细记录了此次发送的过程
4. 系统可以持续的收发文件，即一次传输完成后可以继续进行下一次传输，无需重启程序
5. 可以指定当前主机的角色为发送方或者接收方，但是一旦决定后就无法更改
6. 仅支持两台主机之间的文件传输，接收双方的IP和端口号在配置文件中修改
7. **References**

[1] Forouzan, B. A. (2013). Data Communications and Networking. McGraw-Hill Education.

[2] Kurose, J. F., & Ross, K. W. (2017). Computer Networking: A Top-Down Approach. Pearson.

[3] Tanenbaum, A. S., & Wetherall, D. J. (2011). Computer Networks. Pearson Education.

[4] Peterson, L. L., & Davie, B. S. (2012). Computer Networks: A Systems Approach. Morgan Kaufmann.

[5] Stevens, W. R., Fenner, B., & Rudoff, A. M. (2003). TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley Professional.

1. **Comments**

虽然上课的时候对于GBN原理的学习还是挺顺利的，但是当落实到代码层面，我感觉就是另外一回事了，况且这还是需要个人完成的作业。实现过程还是比较坎坷的，不亚于有些选修课的结课作业了。最终做出来的东西还是挺简陋的，而且也只实现了两台主机之间的收发，不过以后有机会还可以继续完善，这个动手实现的过程也帮助我学习到了很多关于GBN的知识，更好地掌握其背后的确认的重传机制，也算是收获不小吧。