Task3 TCP Socket programming

1. 目的

(1) 理解并掌握TCP socket程序设计的基本结构及工作过程。

(2) 掌握构造应用层自定义的报文结构、报文交互。

1. 内容和分析

(1) client和server都是基于TCP的命令行程序。client运行在host os，server运行在guest os。

(2) 构建全英文可打印字符的ASCII文件。

构建的文本文件默认是UTF-8。在python的读文件时指定文件格式为UTF-8。

出现报错“UnicodeDecodeError: 'gbk' codec can't decode byte 0x80 in position 348: illegal multibyte sequence。”仔细分析发现，从网上找到的英文文章里面可能会包含中文的标点符号，在UTF-8编码中占用3个字节。在某些编码中，例如GBK编码，中文字符和标点符号通常占用2个字节。这些都不能进行分块发送，所以需要注意处理。可打印的ASCII通常在在32到126之间，换行符、回车符不可以，但空格可以。

将文件保存为ANSI编码。ANSI编码与标准的ASCII编码（0到127）兼容，并且不会引入额外的字符或字节。并在读文件和encode、decode时指定“ansi”。

(3) client每次发送文件“不定长的一块”给server，由server把这块文本反转，并返回给client，client在终端打印显示。

在发送时会先分成好几块，然后每一块分别反转再送回。为了让整个文件中的字符串反转，需要将送回的块依次前插进list，使用list的insert。

增加一个将传输的字符串转大写（uppercase）的功能，通过终端的参数来控制使用反转还是转大写。一个简单的实现方式是在client和server的终端传参，作为处理方式的标志mode （“reverse” or “uppercase” / “1” or “2”）。但是这样明显不合理，不能确保client和server输入的mode一定是相同的。所以一个更好的处理方式是，只在client端传参mode，并将mode作为标志加入Initialization报文中，server根据接受报文的mode位来做出不同的处理。

同理继续添加lowercase/capitalize功能。上述功能均可以通过python的函数或者切片操作完成，很容易实现。

(4) client和server之间共4种类型报文，Initialization、agree、reverseRequest、reverseAnswer。四种报文类型，通过Type字段来区分。client端向server端发送Initialization报文，告知server要做reverse的块数N以及处理方式mode，server回复agree报文。client逐次向server发送reverseRequest报文，包含了要做处理的数据；server端返回reverseAnswer报文，其中包含了处理后的数据。reverseRequest、reverseAnswer报文中Length字段指的是数据的长度。

即使设计了报文格式，但其本质还是tcp数据包中的内容。和Task 2一样，将自己设计的报文格式以字节形式表示。为此需要用到b'\x00\x01，b表示这是byte string，而不是string。\x00是以十六进制表示的一个byte，值为0。在构建报文过程中，将整数转化为byte需要用到to\_bytes方法（byteorder = 'big'

，在网络通信中，通常使用大端字节序）或者bytes()类型转换。

encode将字符串转换为字节数据（bytes），用于传输；decode将接收到的字节数据（bytes）转换为字符串，方便程序对数据进行处理和显示。因此，上文中的b、to\_bytes、bytes()不再需要encode和decode，但Data仍然需要。

(5) client端在程序启动时，在命令行中指定filename、server\_ip、server\_port、len\_min、len\_max、mode。每次发送的reverseRequest中的长度随机在[len\_min，len\_max]之间，最后一块除外。mode表示Data的处理方式。

在开始发送数据块前，要先随机确定待发送File中的各块的Bytes长度，再计算块数N。使用blocks列表存放每一块数据，使用random.randint随机长度，并对最后一块数据特殊处理。随后才能执行”client->server的Initialization报文，以及接收agree报文。之后就是循环遍历blocks，每次循环发送完数据后（sendall），client.recv以阻塞的方式接收server的数据。每次client收到来自server的reverseAnswer报文，都在命令行下打印出来。最后将处理好的数据输出到文本文件，文件以处理方式、原文件名、命名。

(6) server端在程序启动时,在命令行中指定监听的端口号。处理报文交互的逻辑在函数tcp\_client\_link中。接收Initialization报文，根据处理类型设置mode，发送agree报文。然后根据数据分块大小N循环接收reverseRequest报文，并将处理后的数据以reverseAnswer报文发送回客户端。

(7) server使用多线程使得能够同时处理2个及以上client的请求，对应版本tcpserver\_v1\_1.py。使server\_socket.listen() 监听多个客户端连接请求。一开始使用无限循环来处理多个客户端连接，为每个客户端启动一个新线程，使用tcp\_client\_link处理报文交互。之后，由java课设得到的灵感，将该过程放入一个单独的线程accept\_thread，这样可以更好地管理服务器的生命周期，并使得服务器在接收连接的同时也能够处理其他任务。例如，同时再处理监听“exit”以关闭服务器的线程exit\_listener，在每次循环休眠1秒，减轻资源的损耗。需要注意的是，在启动accept\_thread线程之后，需要accept\_thread.join()以等待accept\_thread完成，不然会导致客户端无法连接到服务器。

(8) 然而，使用超时和循环机制处理客户端的连接，会增加资源的损耗，即使设置了休眠时间。借用(9)中的非阻塞方式，使用select来监听连接，但仍使用多线程来处理每个客户端，select的使用提高了服务器处理并发连接的效率。对应于版本tcpserver\_v1\_2.py，主要修改在accept\_clients函数。

(9) server使用select方式的非阻塞来同时处理2个及以上client的请求，对应版本tcpserver\_v2\_1.py。该版本基于tcpserver\_v1\_2.py修改，主要区别在于：v1\_2只是用select监听连接，并没有管理连接列表socket\_links。v2\_1使用 select 进行非阻塞 I/O 操作，并将所有客户端连接管理在socket\_links中，通过不同函数处理初始化和数据报文。在accept\_and\_handle\_clients函数中，如果是服务器套接字，则接收新的客户端连接，这和v1\_2一样。如果没有建立连接，则建立连接（tcp\_client\_build\_link）；如果已经建立连接，则进行报文交互，数据处理（data\_process）。

客户端在与服务器通信时不需要像服务器那样使用 select 来处理多个并发连接，因为客户端通常是一次与一个服务器进行通信。当然，为了提高客户端的响应速度和健壮性，可以使用非阻塞的方式和 select 来处理 I/O 操作。

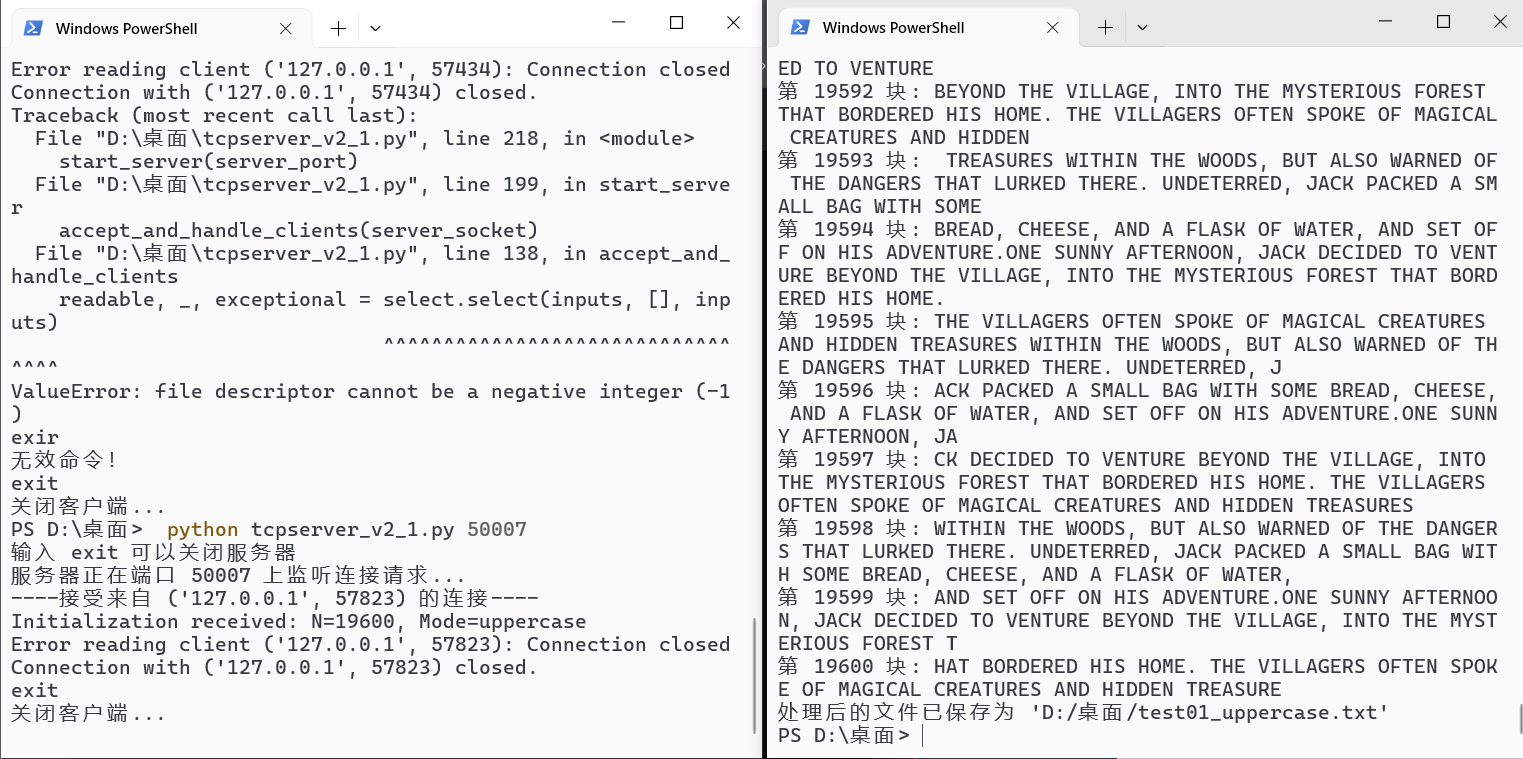
recv方法在默认情况下是阻塞的，即它会一直等待直到读取到指定数量的字节数据。在该版本，客户端套接字设置为非阻塞模式，recv方法会立即返回，如果没有数据可读，它会引发一个异常BlockingIOError。为了确保在非阻塞模式下能够准确读取固定数量的字节（例如6个字节）的数据，通过循环和缓冲区来实现，对应方法recv\_all()。

遇到如下问题：

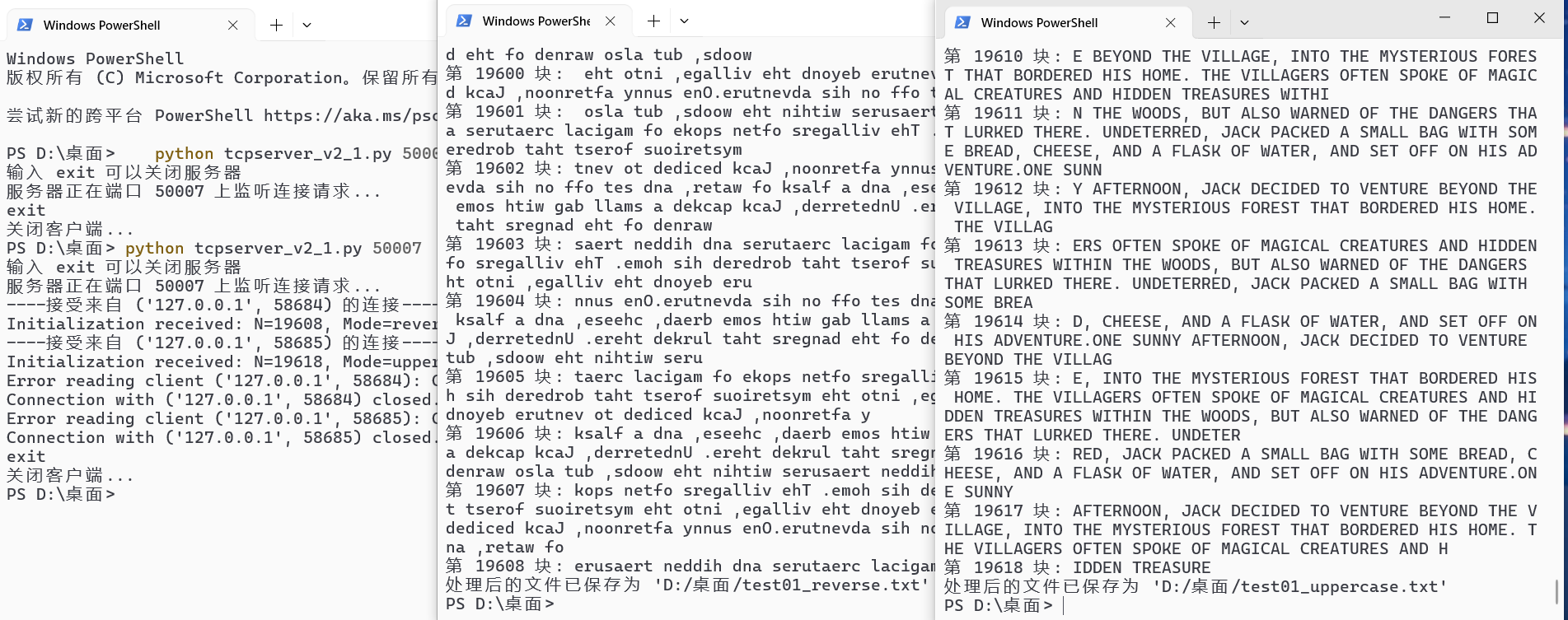


问题出在当一个客户端连接关闭时，select.select(socket\_links, [], socket\_links)仍然包含已关闭的套接字。这会导致 select 在下次调用时尝试处理已关闭的套接字，从而引发 ValueError: file descriptor cannot be a negative integer (-1) 错误。经过尝试，当只有一个客户端时也会出现这种问题。

经过分析，认为在data\_process函数中处理完客户端后，应该立即从 socket\_links列表中移除客户端套接字，并关闭它们。经过修改，已经可以正常处理。如下所示：

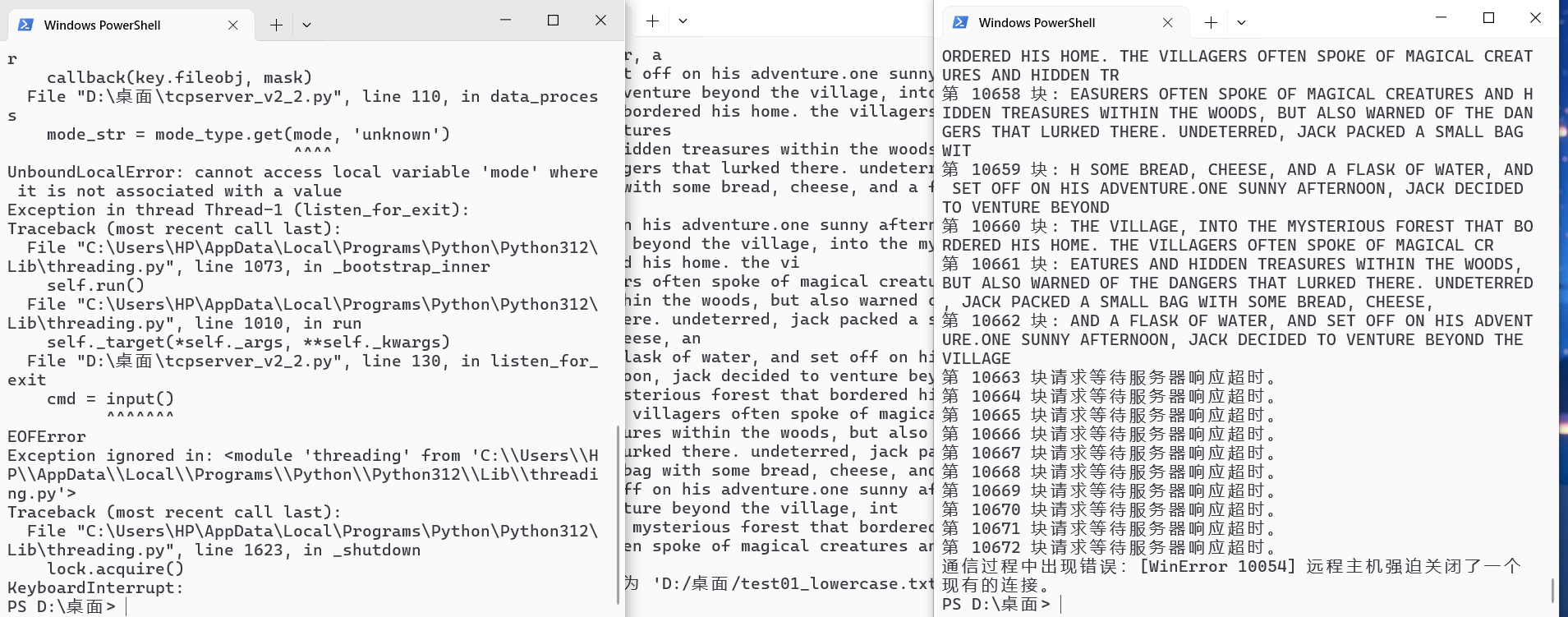


但是在键入“exit”时，未能正常关闭服务器。经过检查，需要在 select.select 调用中设置超时为 1 秒（通过 timeout 参数），以便在每次迭代时检查 is\_running 状态，并在关闭服务器时关闭所有打开的套接字。正确运行两个客户端并正常通过exit关闭，如下图所示：



(10) v2\_1版本使用的是select。尝试使用selectors，构建tcpserver\_v2\_2.py。使用 selectors 模块来管理多个客户端连接。使用统一的选择器对象 (sel) 管理所有的 I/O 事件。代码清晰且不需要手动管理 socket\_links 列表。更适合处理大量并发连接，提供更好的跨平台兼容性和性能。

遇到如下问题：



经过分析，发现原因在于处理类型mode的错误使用。本意是在按mode类型处理完数据后，在data\_process的except打印“来自客户端 {client\_address} 的 {mode\_str} 处理已完成”，但是此时mode局部变量已失效。因此修改代码，在except重新赋值mode即可。

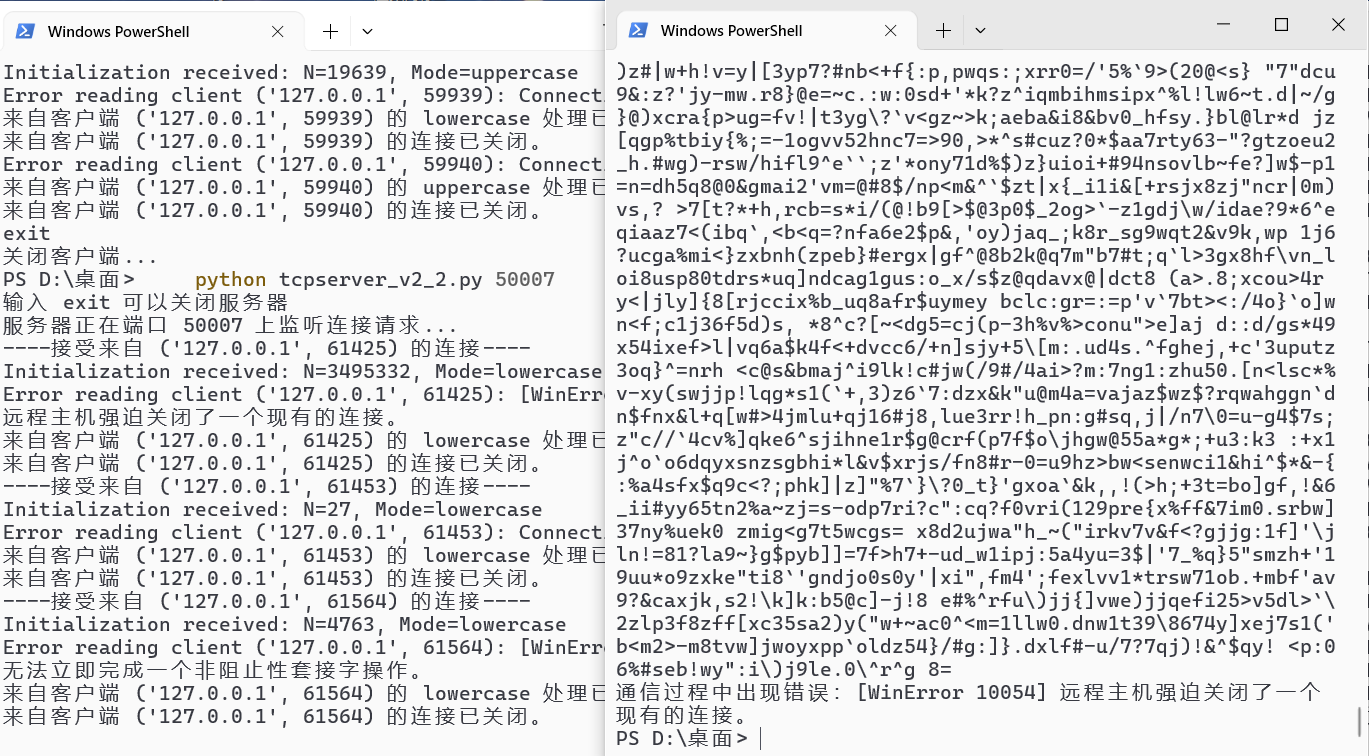


1. 测试和抓包

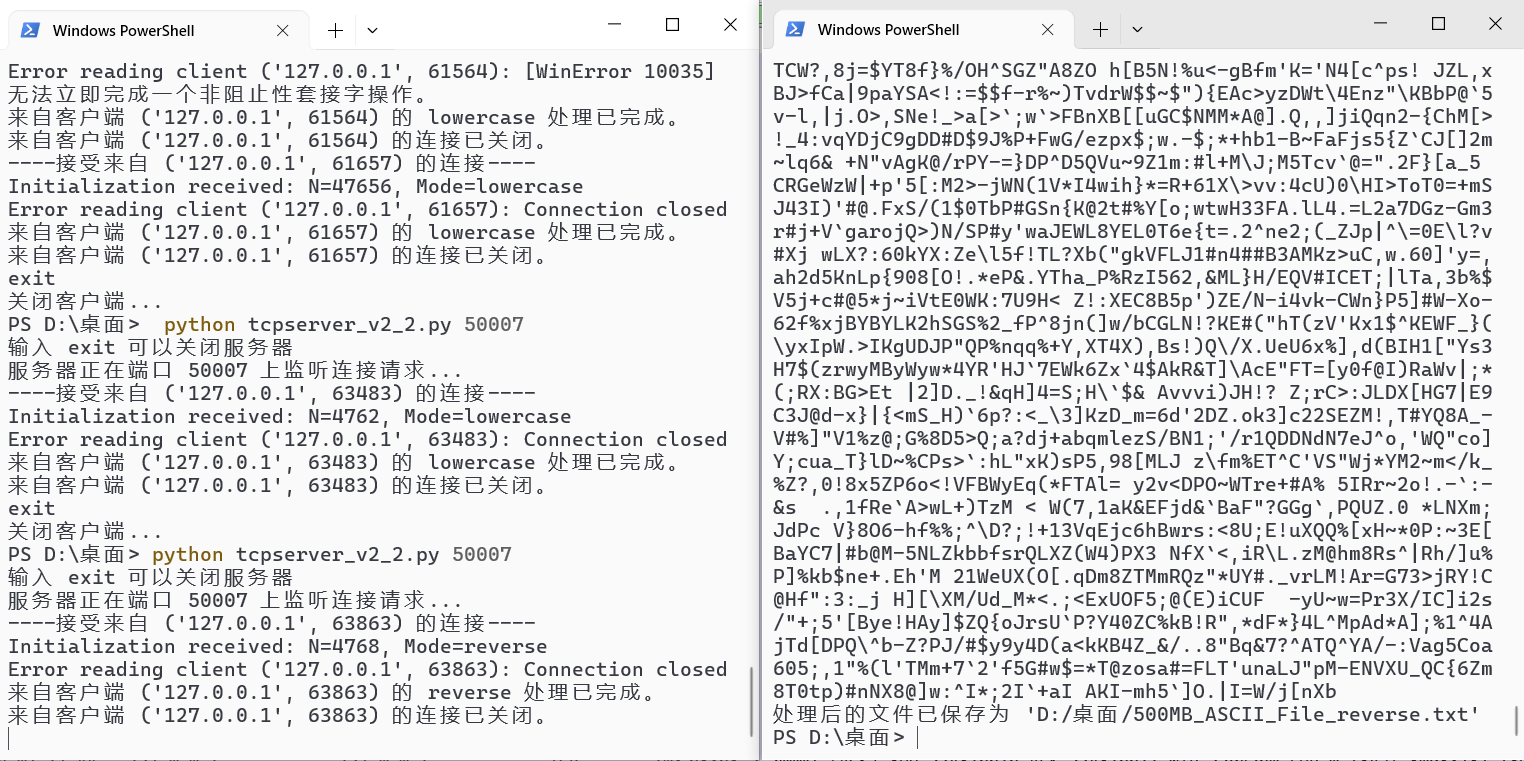
**1. 本地v2\_2下的大文件处理测试（考虑到跨主机数据包传输的内容限制）**

客户端命令行输入：python tcpclient.py D:/桌面/500MB\_ASCII\_File.txt 127.0.0.1 50007 100000 120000 lowercase。500MB\_ASCII\_File.txt文件的大小为500MB。如下图所示，使用的是v2\_2版本的服务器。500MB的数据被分成了4763块，每个包的大小在100000~120000 bytes。

在运行过程中出现报错，没能完成文件的输出。



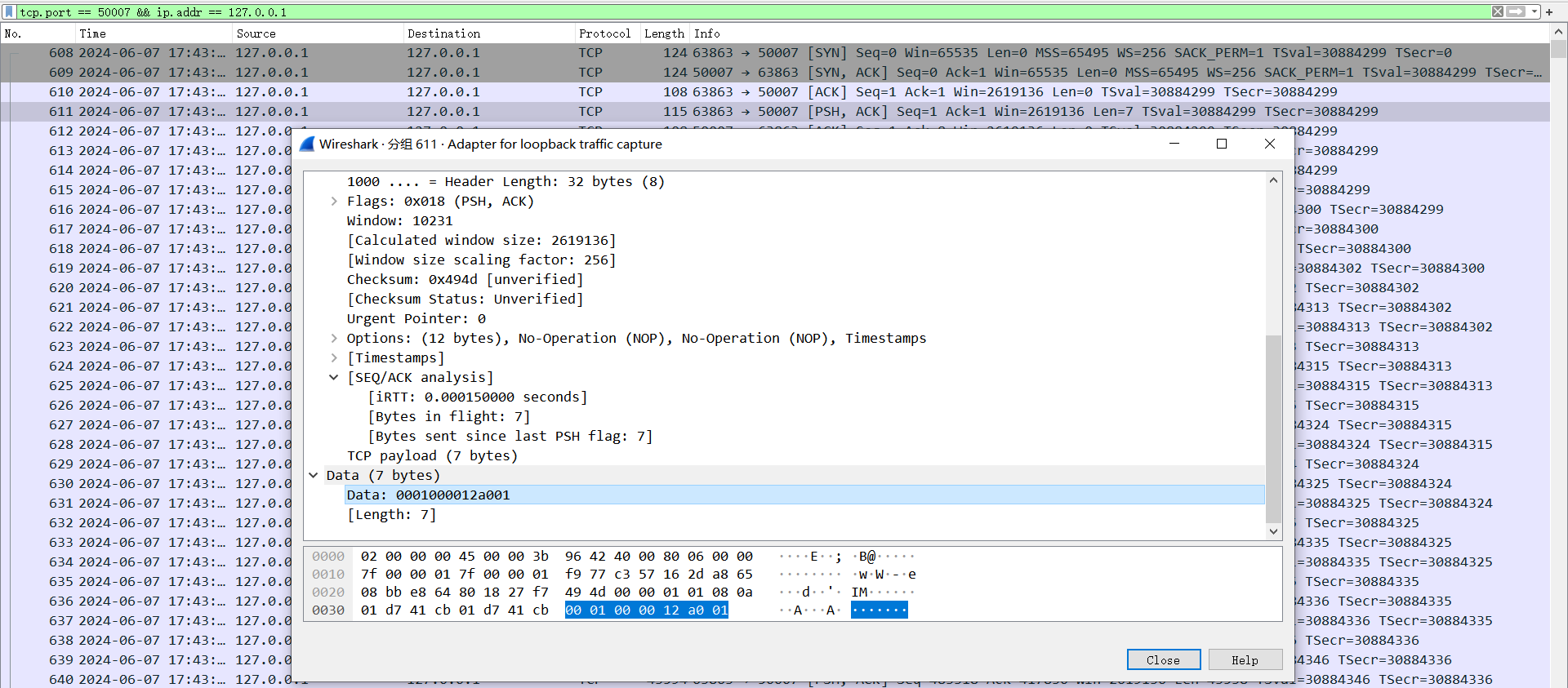
如下图所示，重新尝试，结果成功完成文件输出。



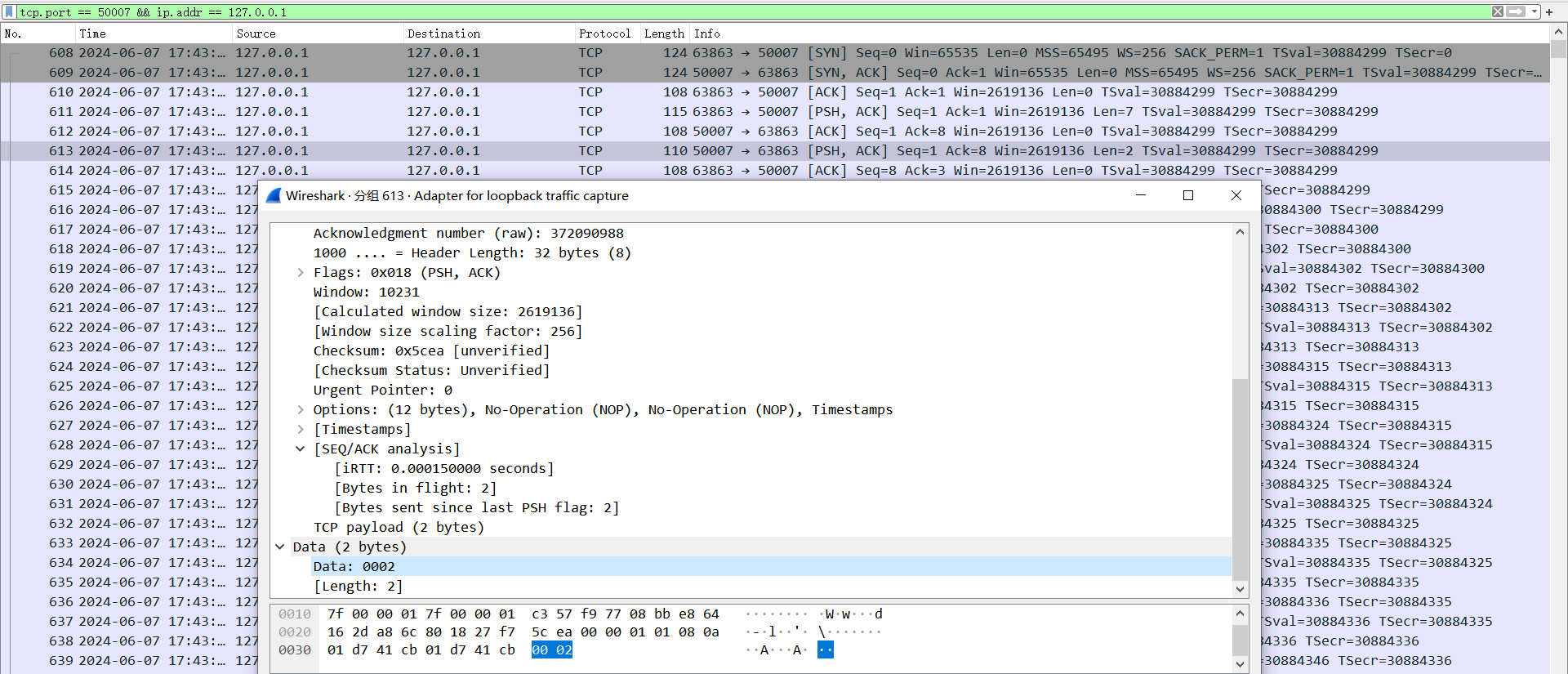
抓包如下：

观察Initialization、agree、reverseRequest、reverseAnswer报文

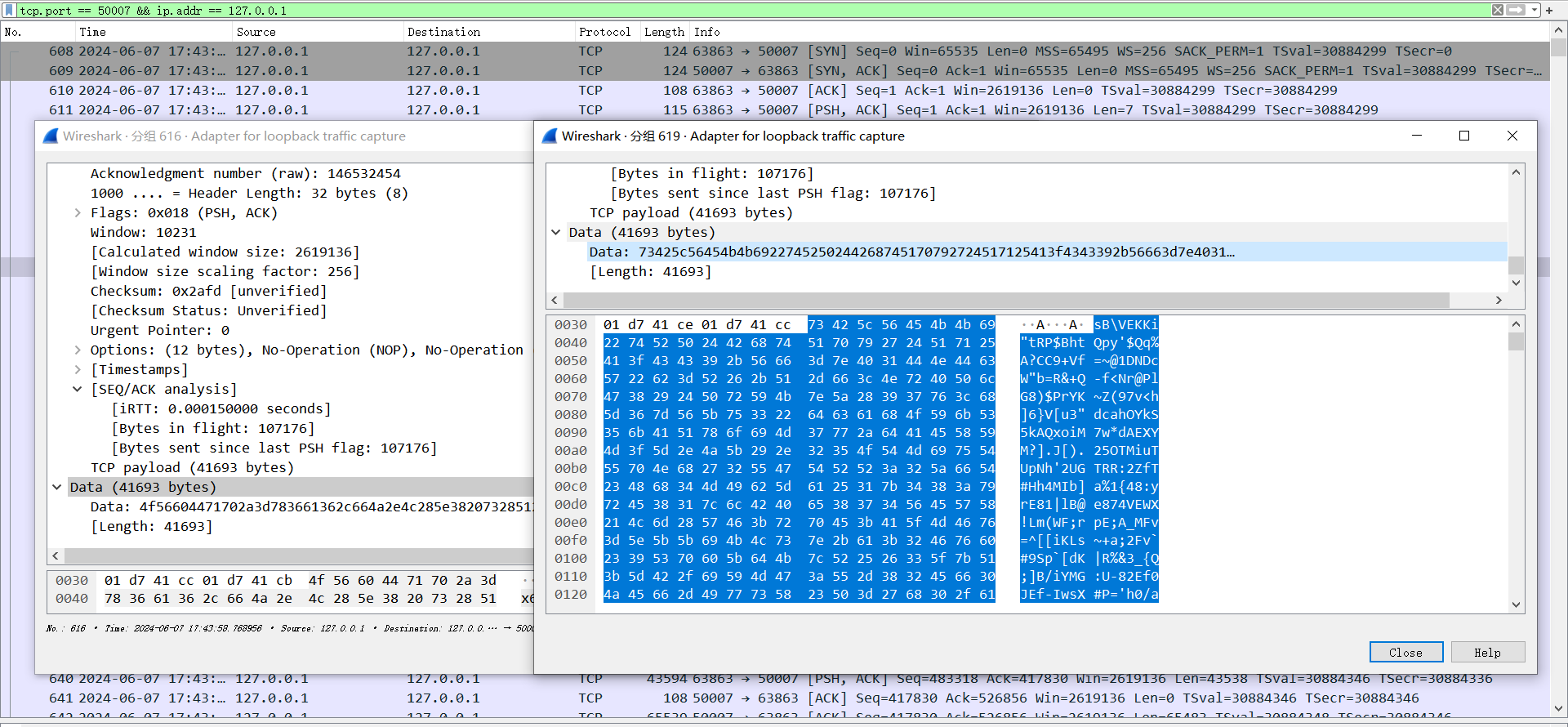
a. 可以看到SYN、SYN ACK、ACK，表示三次握手过程，成功建立TCP连接。N0.611 为客户端63863向服务器50007 PSH的第一条数据。观察Data，发现长度为7 bytes，与自行设计的Initialization报文对应，前两个字节00 01 表示类型，中间四个 bytes 表示数据分包大小，00 00 12 a0 即4768个包。最后一个字节表示数据处理模式，01 表示处理方式为reverse。



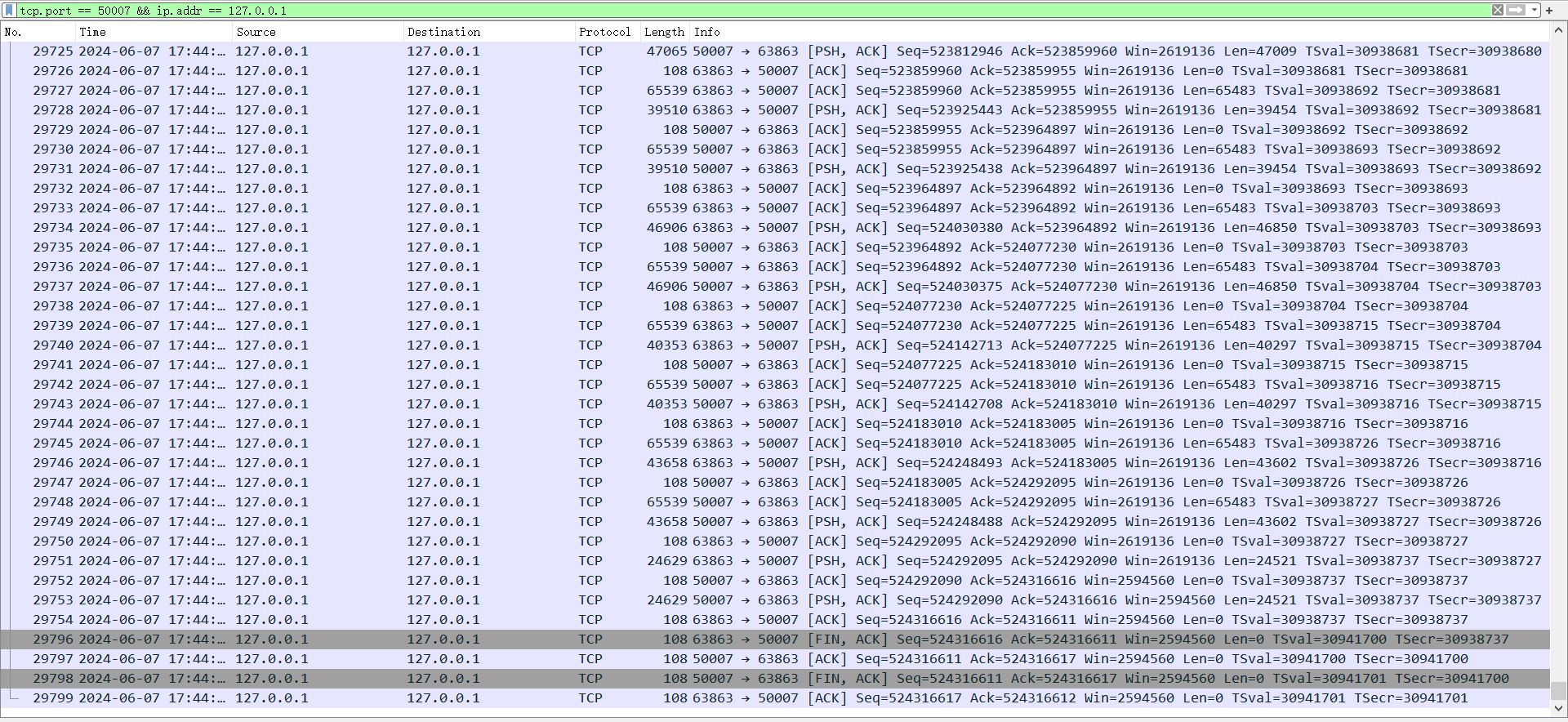
b. N0.613 为服务器向客户端PSH的第一条数据，观察Data，发现长度为2 bytes，与自行设计的agree报文对应，两个字节00 02 表示类型。



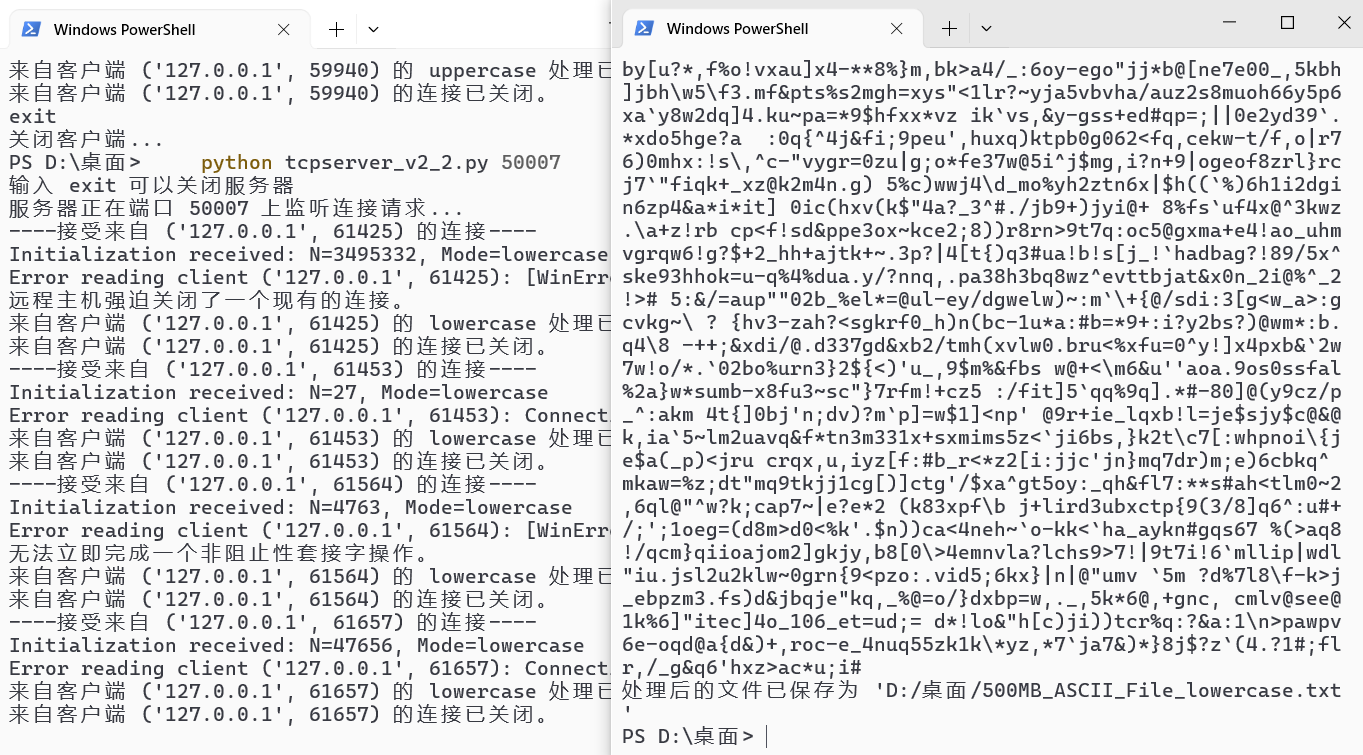
c. N0.616 、619依次为客户端向服务器PSH的reverseRequest报文、服务器向客户端返回的reverseAnswer报文。长度均为41693 bytes很明显已经达到TCP的最大内容传输。可以发现前两个与设计的类型不对应，分析为设计的Data因为长度过大，使得Length和Data占用了Type的空余位，导致十六进制的数值发生变化。



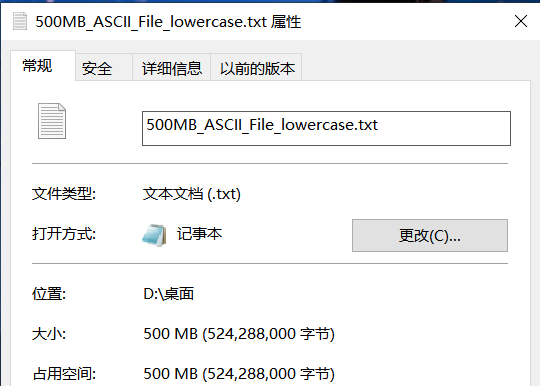
d. 可以看到FIN ACK、ACK、FIN ACK、ACK表示四次挥手过程，成功断开TCP连接。



如下图所示，更换条件，尝试将500MB的数据被分成了47656块，每个包的大小在10000~12000 bytes。

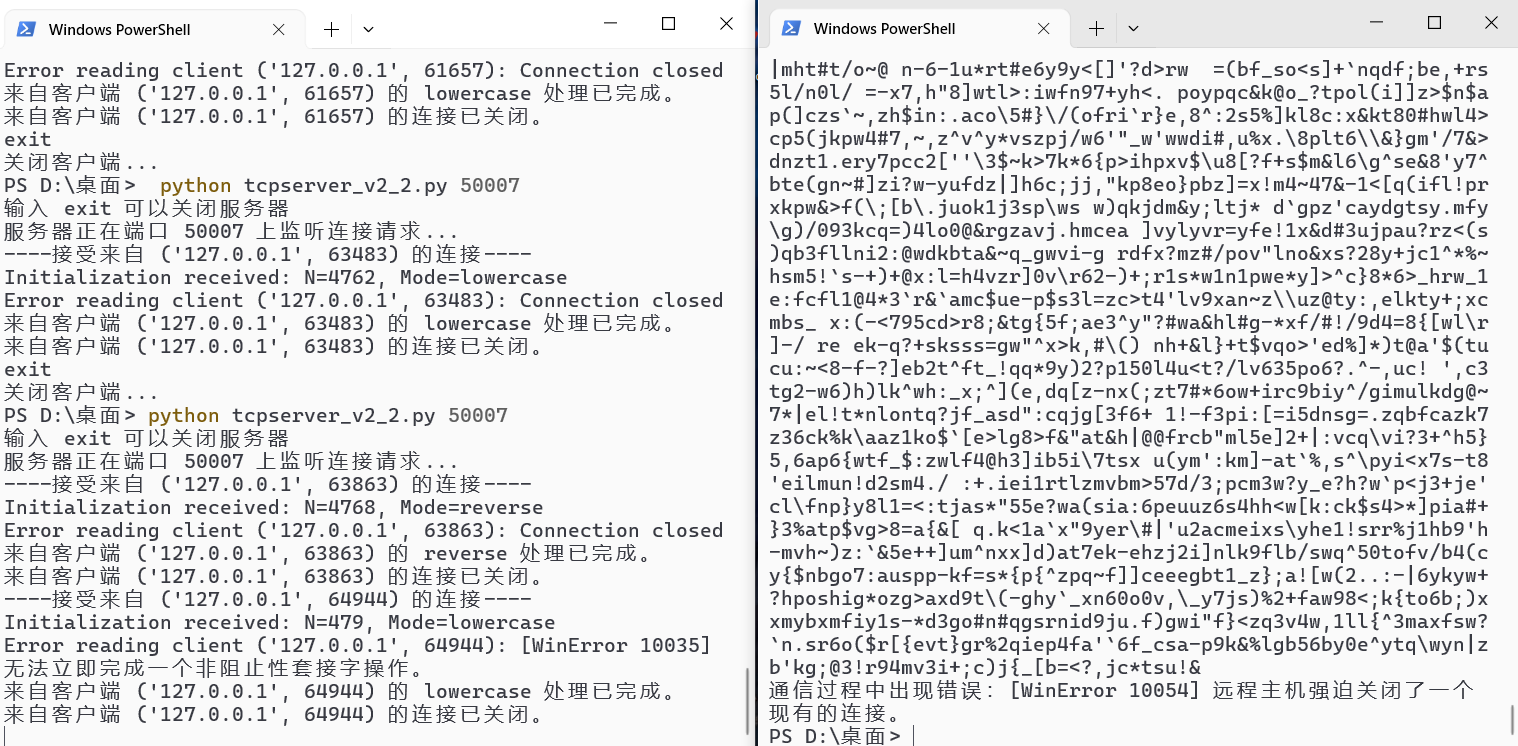


最后成功处理，输出文件。

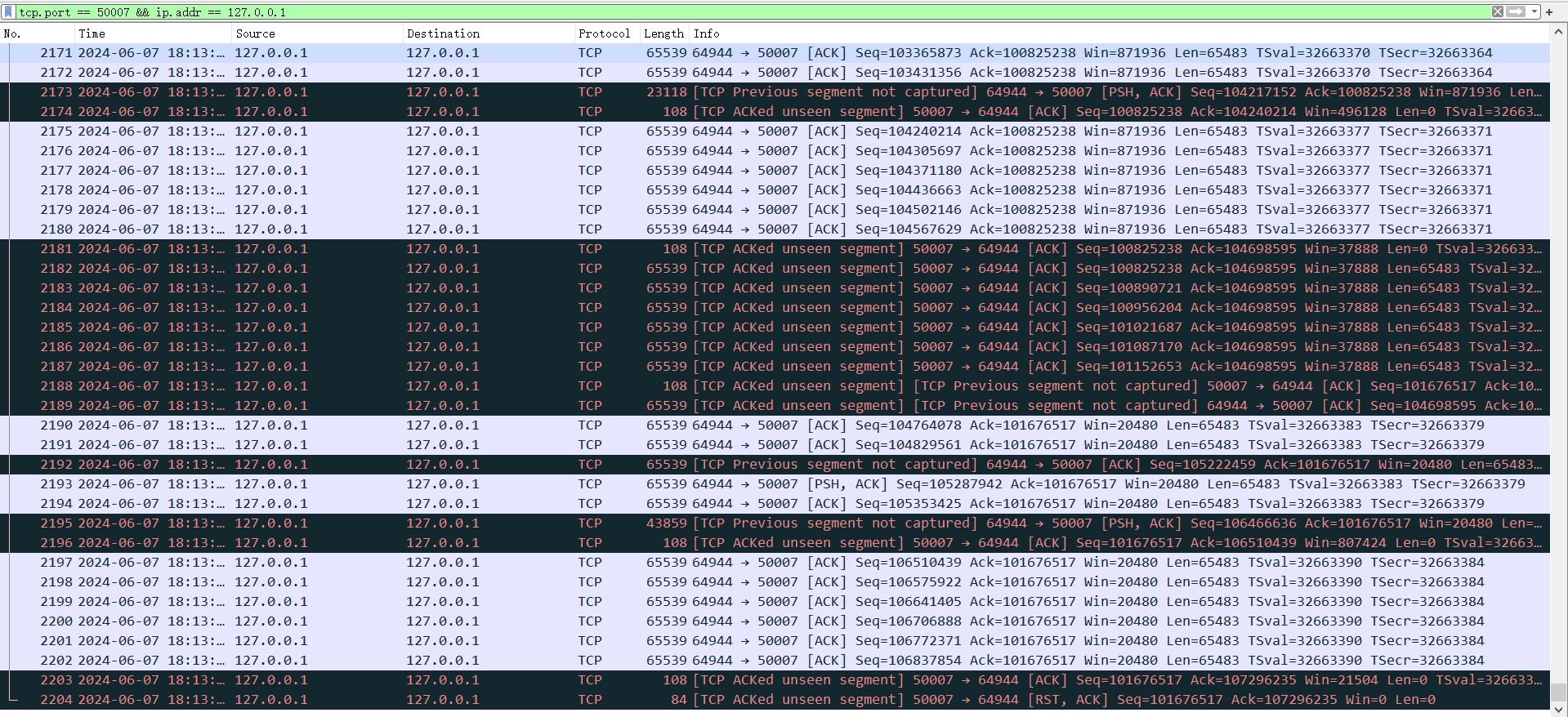


**思考：为什么发送的每块数据包的大小为10万 bytes时偶尔会导致处理中断。**

**继续尝试100万~120万的数据块，如下所示，出现问题：**



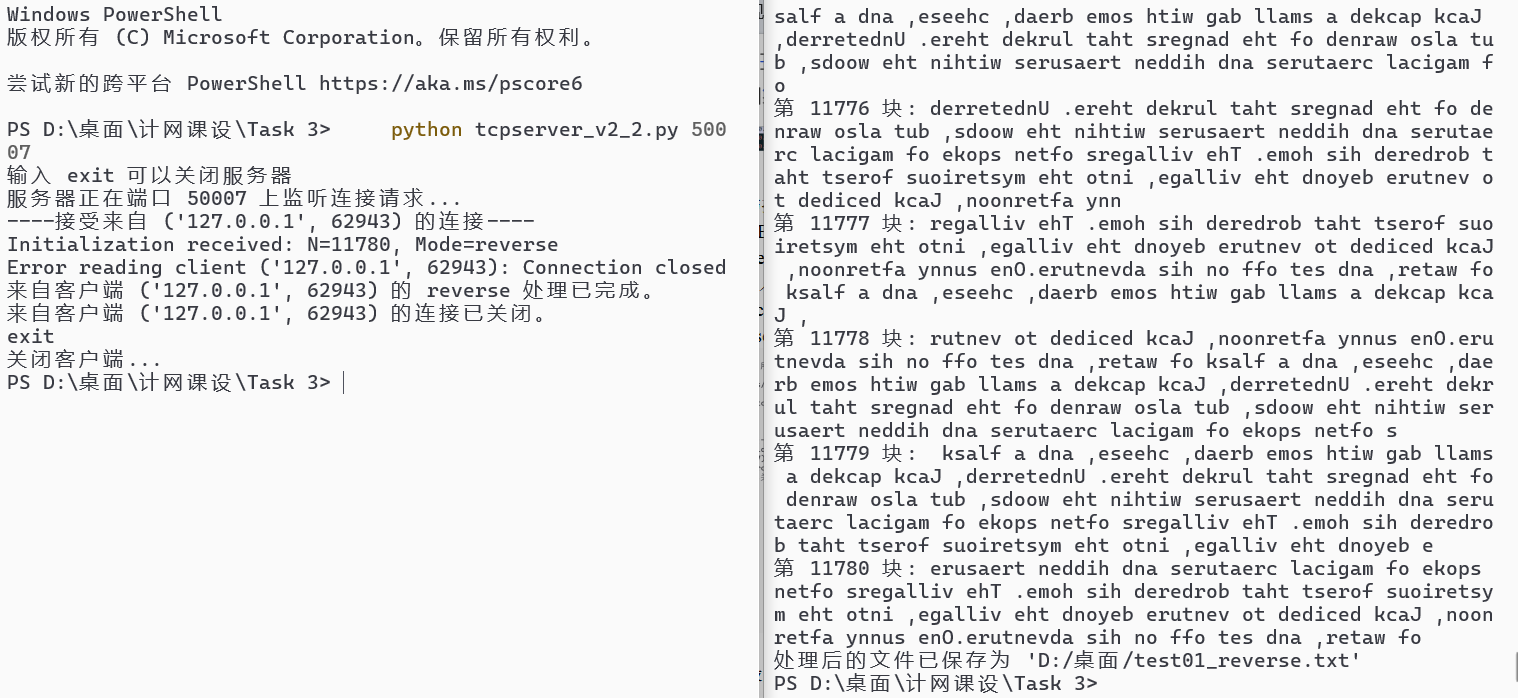
抓包如下，可以看到Wireshark没有捕获到所有相关的TCP分片。由此推测问题的原因在于单次传输的数据太大，超过了tcp数据包的切片上限，导致丢失部分分片数据，使得断开连接。



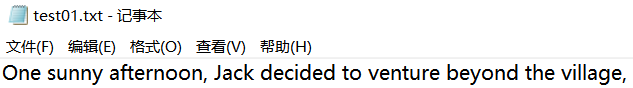
**2. 小文件下的测试与抓包**

**(1) v2\_2版本测试2.8MB文件的reverse与抓包（主要观察1中没看出来的reverseRequest、reverseAnswer报文）**

如下图所示，server输入 python tcpserver\_v2\_2.py 50007，监听端口50007。数据被分成11700块。client输入python tcpclient.py D:/桌面/test01.txt 127.0.0.1 50007 200 300 reverse，对test01.tet做反转处理，服务器ip为127.0.0.1，端口号为50007，每次处理的数据块的字节数在200~300之间。

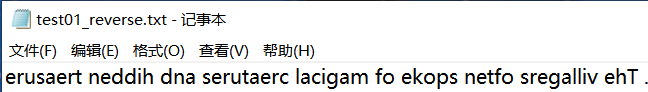


test01.tet的第一句和最后一句如下图所示：





test01\_reverse.txt的第一句和最后一句如下图所示，可以看出，确实完成了反转：

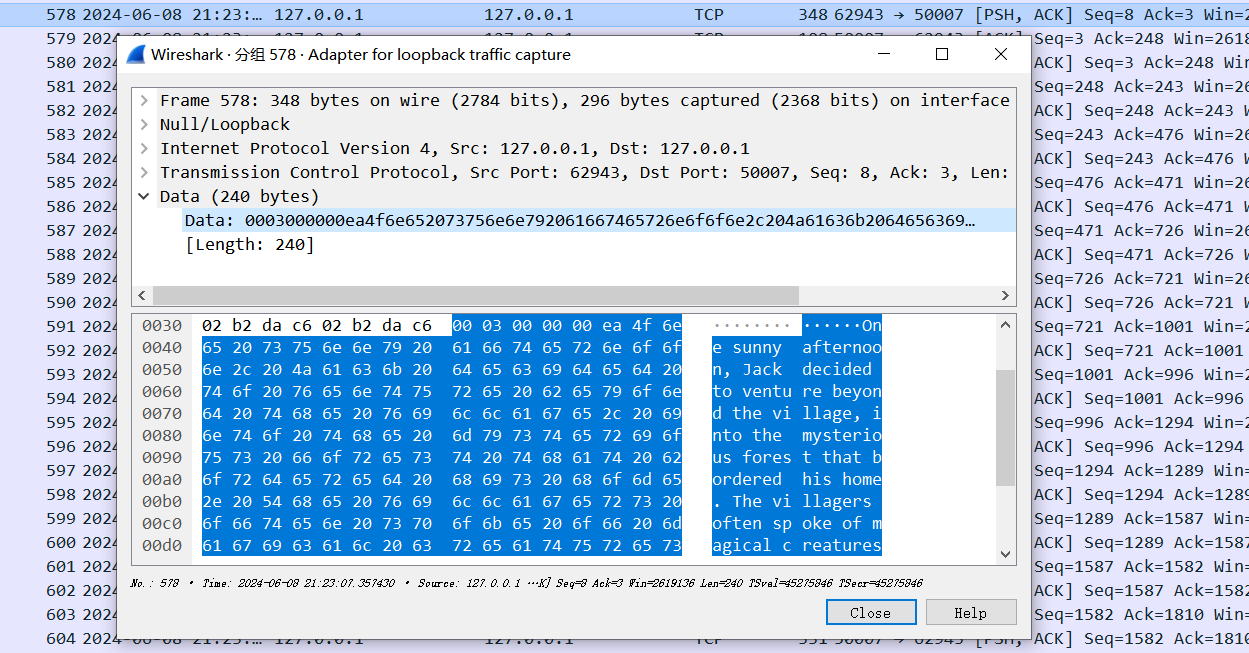


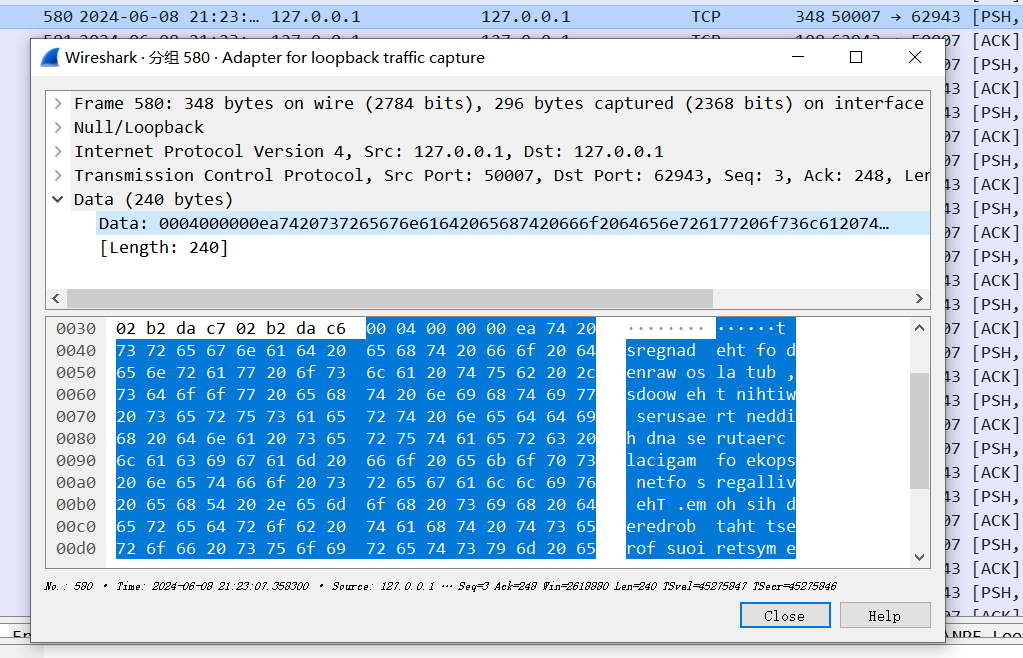


抓包如下：

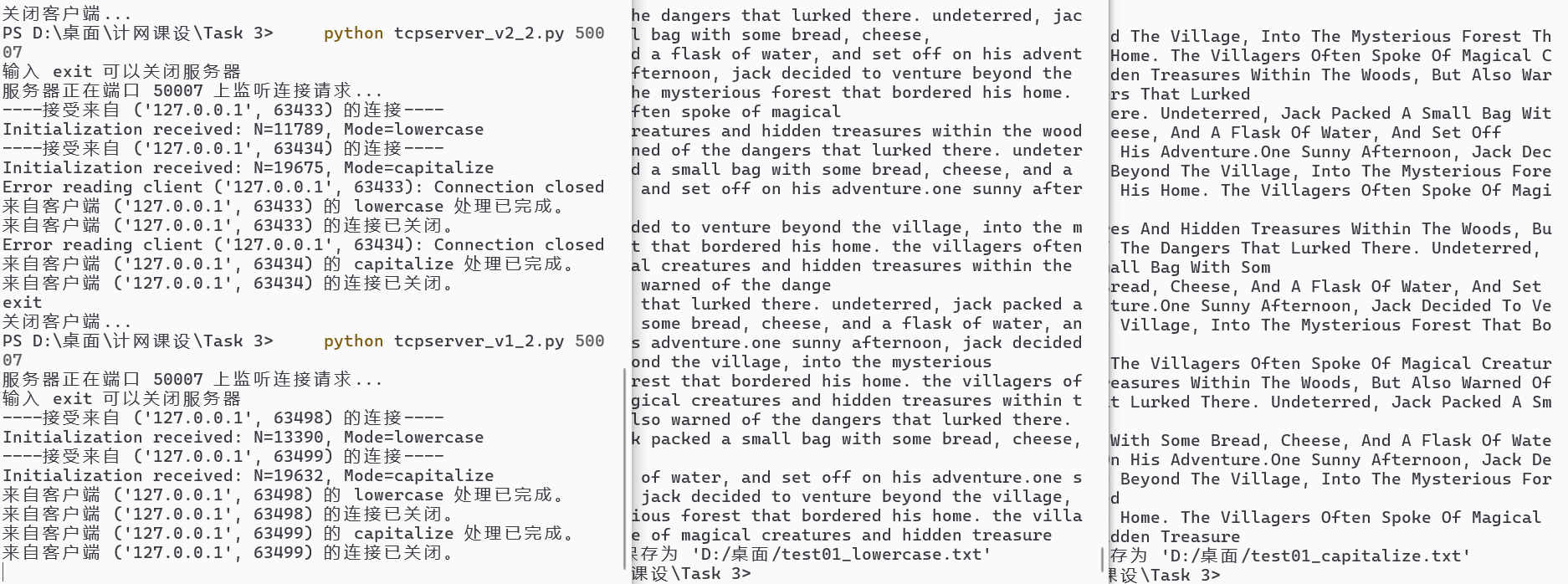
观察reverseRequest、reverseAnswer报文；Initialization、agree在1中

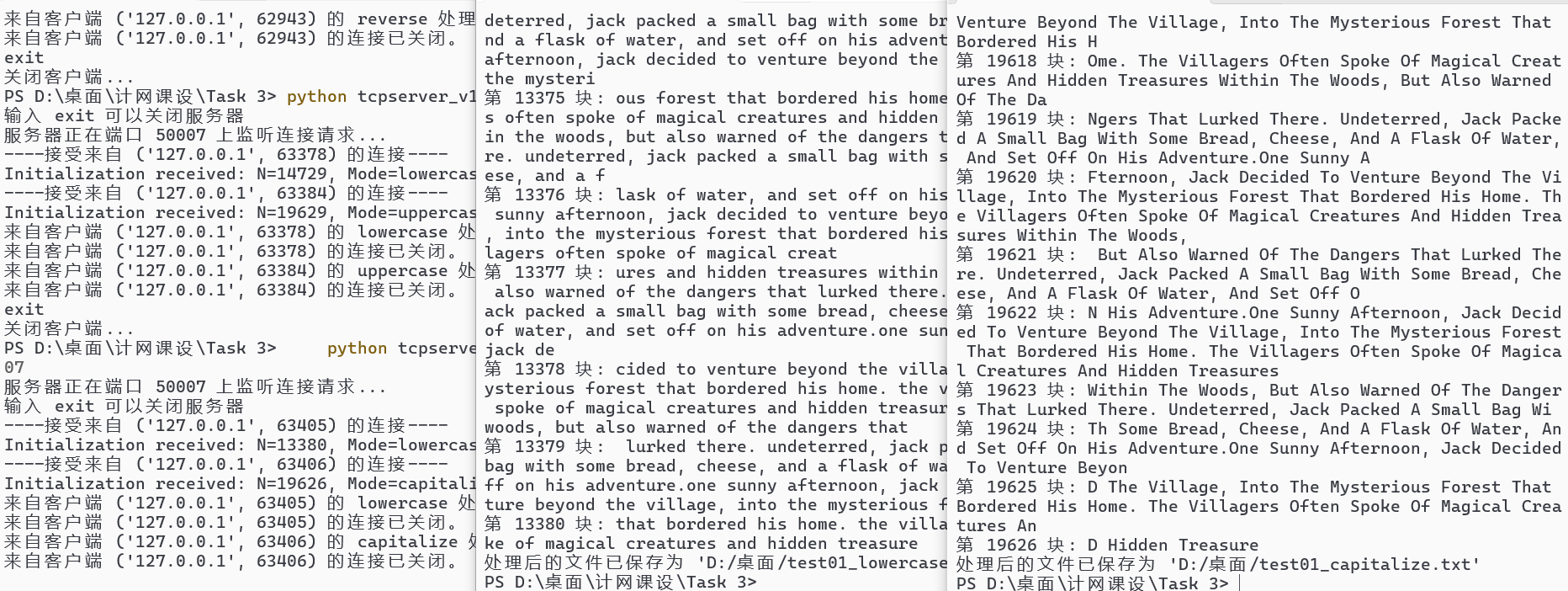
N0.578 、580依次为客户端向服务器PSH的reverseRequest报文、服务器向客户端返回的reverseAnswer报文，长度均为240 bytes。可以发现前两个字节与设计的类型对应，分别为0003、0004。中间四个字节为数据块的长度，均为000000ea，长度为234 bytes。234 bytes + 4 bytes + 2 bytes = 240 bytes。由此说明，在1中确实是因为Data太大导致的前两个字节与设计的类型不对应。

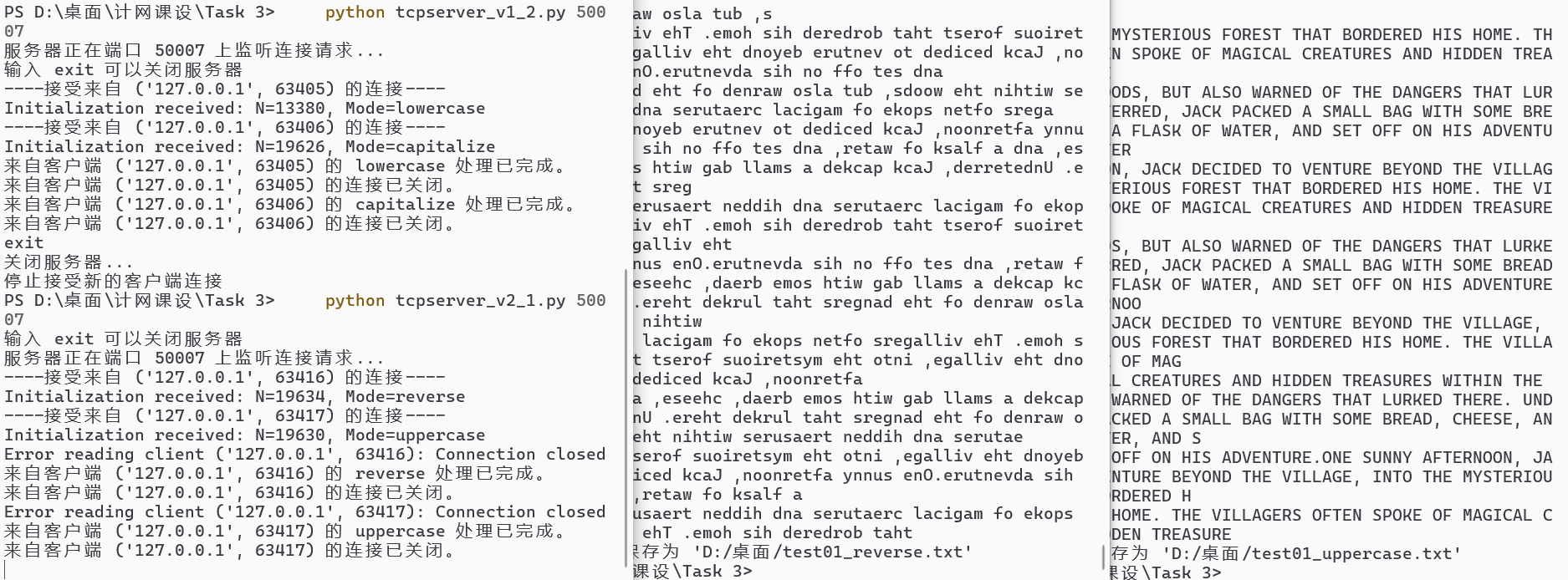


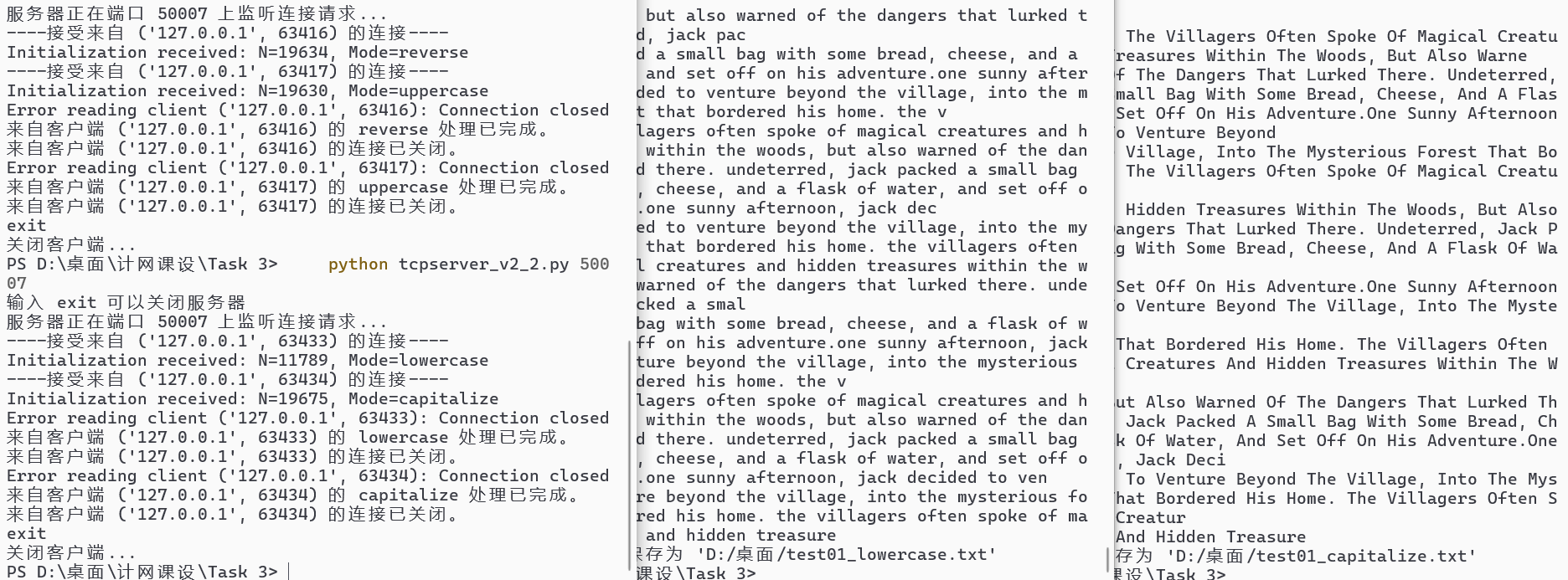


**(2)下图依次为 server v1\_1、v1\_2、v2\_1、v2\_2 同时处理多个client的测试。**

****

****

****

****