



哈尔滨工业大学  
Harbin Institute of Technology

# 计算机网络 课程实验报告

实验名称	可靠数据传输协议的设计与实现					
姓名	杨明达		院系	未来技术学院		
班级	22R0311		学号	2022110829		
任课教师	李全龙		指导教师	李全龙、郭勇		
实验地点	G001		实验时间	周六 5、6 节		
实验课表现	出勤、表现得分(10)		实验报告 得分(40)		实验总分	
	操作结果得分(50)					
教师评语						



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

**实验目的：**

理解可靠数据传输的基本原理；掌握停等协议的工作原理；掌握基于 UDP 设计并实现一个停等协议的过程与技术。

理解滑动窗口协议的基本原理；掌握 GBN 的工作原理；掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术。

**实验内容：**

- 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；（选作内容，加分项目，可以当堂完成或课下完成）
- 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。（选作内容，加分项目，可以当堂完成或课下完成）
- 5) 将所设计的 GBN 协议改进为 SR 协议。（选作内容，加分项目，可以当堂完成或课下完成）

实验过程：（由于停等协议是 GBN 协议的特殊情况，故该环节仅分析 GBN 协议）

**一、实验注意要点**

- 1) 基于 UDP 实现的停等协议，可以不进行差错检测，可以利用 UDP 协议差错检测；
- 2) 为了验证所设计协议是否可以处理数据丢失，可以考虑在数据接收端或发送端引入数据丢失。
- 3) 在开发停等协议之前，需要先设计协议数据分组格式以及确认分组格式。
- 4) 计时器实现方法：对于阻塞的 socket 可用 `int setsockopt(int socket, int level, int option_name, const void* option_value, size_t option_len)` 函数设置套接字发送与接收超时时间；对于非阻塞 socket 可以使用累加 `sleep` 时间的方法判断 socket 接受数据是否超时（当时间累加量超过一定数值时则认为套接字接受数据超时）。
- 5) 自行设计数据帧的格式，应至少包含序列号 Seq 和数据两部分；
- 6) 自行定义发送端序列号 Seq 比特数 L 以及发送窗口大小 W，应满足条件  $W+1 \leq 2L$ 。
- 7) 一种简单的服务器端计时器的实现办法：设置套接字为非阻塞方式，则服务器端在 `recvfrom` 方法上不会阻塞，若正确接收到 ACK 消息，则计时器清零，若从客户端接收数据长度为 -1（表示没有接收到任何数据），则计时器+1，对计时器进行判断，若其超过阈值，则判断为超时，进行超时重传。（当然，如果服务器选择阻塞模式，可以用到 `select` 或 `epoll` 的阻塞选择函数，详情见 MSDN）。
- 8) 为了模拟 ACK 丢失，一种简单的实现办法：客户端对接收的数据帧进行计数，然后对总数进行模 N 运算，若规定求模运算结果为零则返回 ACK，则每接收 N 个数据帧才返回 1 个 ACK。当 N 取值大于服务器端的超时阈值时，则会出现服务器端超时现象。
- 9) 当设置服务器端发送窗口的大小为 1 时，GBN 协议就是停-等协议。

**二、实验实现核心技术——GBN 协议**
**（1）基本原理**

GBN 是一种基于滑动窗口机制的可靠数据传输协议，主要解决网络中的数据包丢失、出错及顺序问题，确保接收方接收到所有数据包。其核心思想是发送方可以连续发送多个数据包，而无需等待每个数据包的 ack 确认，如果在滑动窗口内某个数据包丢失或出错，接收方会拒绝接收后续的包，发送方必须回退到该出错包并重新发送该包及其后续所有包。

对于滑动窗口机制，GBN 使用发送窗口和接收窗口来管理数据包的发送与接收。其中，发送窗口是发送方维护的窗口，用于控制当前能够发送但尚未被确认的数据包数量；接收窗

口是接收方一般只接受按序到达的数据包，如果接收到的包不符合期望序号，就会丢弃并只发送最后按序接收的包的确认。

对发送方分析。在发送数据包环节，发送方可以连续发送数据包，序号范围为 $[base, base+N-1]$ ，其中  $base$  是窗口的起始序号， $N$  是窗口大小。在等待  $ack$  确认包环节，如果接收到某个数据包的  $ack$ ，滑动窗口会向前移动；如果在等待时间内没有收到  $ack$ （超时），发送方会回退到第一个未确认的包，并重新发送该包及其后的所有包。在超时重传环节，如果发送窗口内的某个包超时未收到  $ack$ ，则从超时包开始重发整个窗口内的所有包。

对接收方分析。按序接收数据包，如果接收方收到的数据包与期望序号一致，则正常接收，并发送  $ack$ ；如果收到的包序号不符合期望值，则会丢失该包，并重新发送最后按序接收到的包  $ack$ 。 $ack$  累计确认，GBN 使用累计确认，即接收方发送的  $ack$  表示所有序号小于等于该  $ack$  的包均已成功接收。

## (2) 原理图

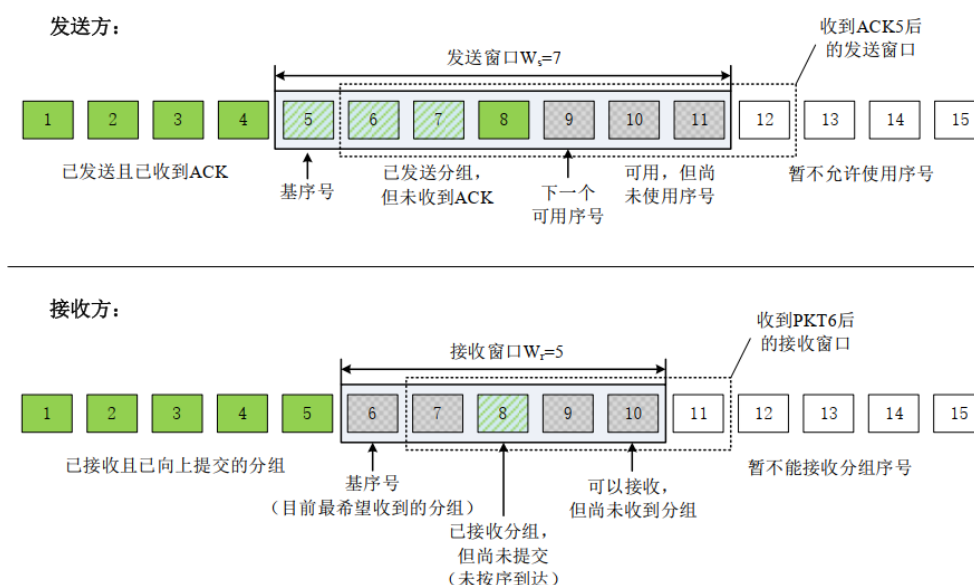


图 2-1 GBN 协议结构图

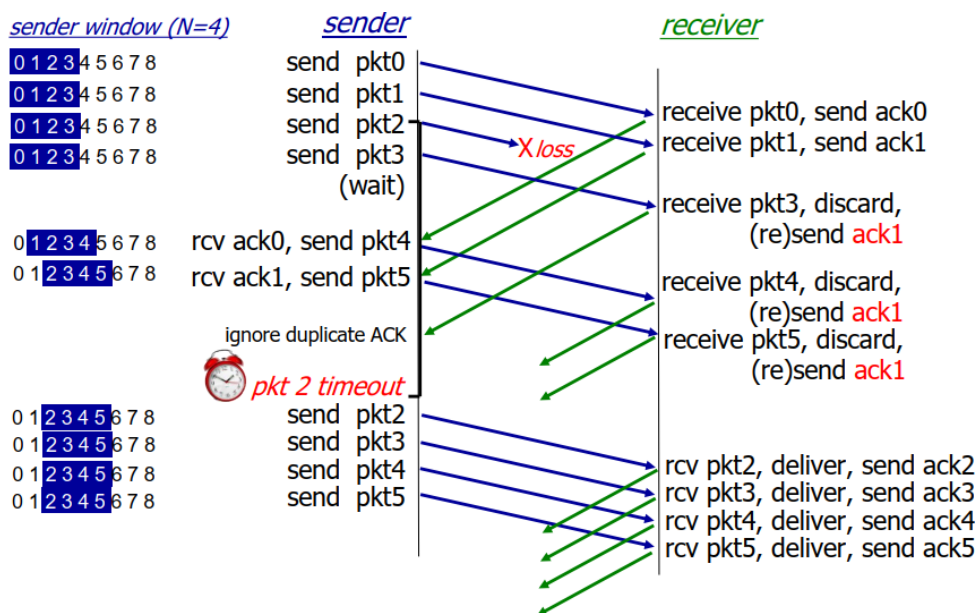


图 2-2 GBN 协议流程图

## (3) 功能分析

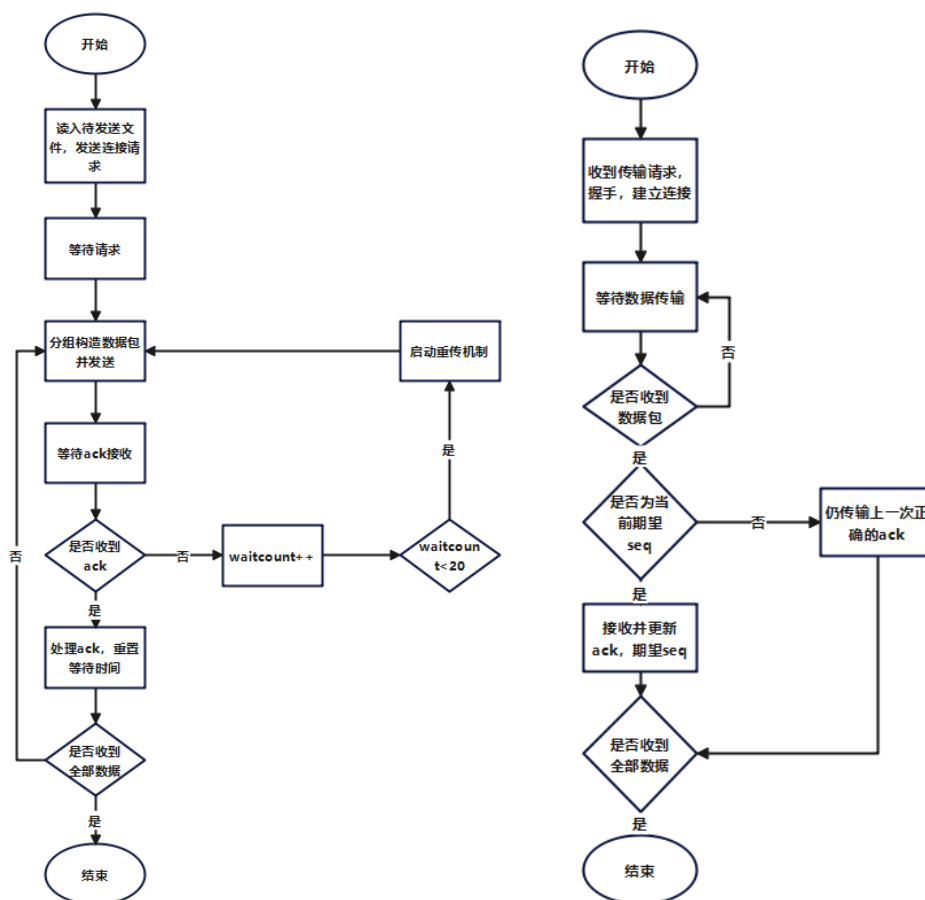


图 2-3 传输端（左）和接收端（右）流程图

## (a) seqIsAvailable()函数

该函数用来检测当前序列号 `curSeq` 是否可用。首先计算当前序列号与当前确认号的差值，之后处理差值，如果结果大于等于 0，则不变；否则加上 `SEQ_SIZE` 窗口大小。之后进入判断如果序列号不在窗口内，则表示未收到 `ack`；如果收到 `ack`，则表示当前序列号可用。

## (b) timeoutHandler()函数

该函数是超时重传处理函数，滑动窗口内的数据帧都要重传。在函数内，变量滑动窗口内的数据帧，计算序列号并标记为重传，更新已发送的总序列号数，重置当前发送序列号。

## (c) ackHandler()函数

该函数收到 `ack`，累计确认，取数据帧的第一个字节，由于发送数据时，第一个字节（序列号）为 0 时发送失败，因此加一了，此处需要减一还原。在函数中，当 `curAck` 小于等于 `index` 时，表示 `ack` 没有回到 `curAck` 的左边，所以依次标记 `curAck` 到 `index` 之间的 `ack` 为 `TRUE` 并更新 `curAck`。当 `curAck` 大于 `index` 时，表示 `ack` 超过了最大值，回到了 `curAck` 的左边，所以分两部分标记 `ack` 为 `TRUE` 并更新 `curAck`。

## (d) getCurTime()函数

该函数获取当前系统时间，将结果存入 `pTime` 中。

## (e) LossInLossRatio()函数

该函数根据丢失率随机生成一个数字，判断是否丢失，丢失则返回 `TRUE`，否则返回 `FALSE`。

## (f) -testgbn 命令

该命令的实现有两个阶段：等待握手阶段、等待接受数据阶段。

在等待握手阶段，分析状态码，如果收到 205，则表示服务器准备好了，可以发送数据，则开始发送 200 表示客户端准备好了，可以接收数据。

在等待接收数据阶段，如果收到的数据包为空，则表示文件传输结束。之后判断是否丢数据包，如果丢包则不处理。之后处理数据包，如果是期待的包，则正常接收，并更新期待数；如果当前一个包都没有收到，则等待 Seq 为 1；如果收到乱序的包，则返回上一次正确接收的序号。接着模拟 ack 丢失，如果 ack 丢失则不处理，否则通过 sendto 发送 ack 给服务器。

#### (g) -optestgbn 命令

该命令的实现有三个阶段：握手阶段、等待接受数据阶段、数据传输阶段。

在握手阶段，发送 205 表示客户端准备好可以开始传输。

在等待接收数据阶段，等待接收 200。如果如果等待计数器超过上限，则超时放弃此次连接。如果收到 200 状态码，客户端进入数据传输阶段。

在数据传输阶段，检查是否可以发送该序列号的数据。如果可以，则正常发送，准备接收 ack；如果超时，则重传。如果为收到 ack，则计数器累计，直到超时重发。

#### (h) -download 命令

该命令的实现有三个阶段：等待握手阶段、等待接收数据阶段、结束阶段。

在等待握手阶段，分析状态码，如果收到 205，则表示服务器准备好了，可以发送数据，则开始发送 200 表示客户端准备好了，可以接收数据。

在等待接收数据阶段。每次收到一个数据包后，客户端检查其序列号是否与预期一致，若包的序列号正确，发送对应的 ACK。若不符合预期，客户端根据上一个确认的序列号发送 ACK。如果收到的数据包为空，则表示文件传输结束。之后判断是否丢数据包，如果丢包则不处理。之后处理数据包，如果是期待的包，则正常接收，并更新期待数；如果当前一个包都没有收到，则等待 Seq 为 1；如果收到乱序的包，则返回上一次正确接收的序号。接着模拟 ack 丢失，如果 ack 丢失则不处理，否则通过 sendto 发送 ack 给服务器。

在结束阶段，更新已发送的包的总数。

除此之外，设置 stage 检测结束阶段，如果结束，则输出文件传输结束，输出文件内容，将数据写入文件中。

#### (i) -upload 命令

该命令的实现有四个阶段：等待握手阶段、等待接收数据阶段、数据传输阶段、结束阶段。

在握手阶段，发送 205 表示客户端准备好可以开始传输。

在等待接收数据阶段，等待接收 200。如果如果等待计数器超过上限，则超时放弃此次连接。如果收到 200 状态码，客户端进入数据传输阶段。

在数据传输阶段，检查是否可以发送该序列号的数据。如果可以，则正常发送，准备接收 ack；如果超时，则重传。如果为收到 ack，则计数器累计，直到超时重发。

在结束阶段，输出文件传输结束，结束循环。

#### (j) -time 命令

获取时间，将时间转化为字符串。

#### (k) -quit 命令

退出程序。

## 二、实验实现核心技术——SR 协议

### (1) 基本原理

SR 协议是一种基于滑动窗口的可靠数据传输协议，它的主要特点是只重传出错或丢失的数据包，而不是像 GBN 那样回退到丢失包处重新发送整个窗口内的所有包。SR 协议提

高了网络资源的利用率。其核心思想是发送方和接收方都维护滑动窗口，用于追踪已发送/接收的包；当某个数据包丢失时，只需重传该丢失的包，接收方可以接收乱序数据包并缓存，直到所有前序包达到为止。

对于滑动窗口机制，SR 协议和 GBN 协议一样使用发送窗口和接收窗口，但 SR 协议允许数据包乱序接收并独立确认 ack。其中，发送窗口大小为  $N$ ，表示发送方在未收到确认前最多能发送  $N$  个包，窗口随着数据包的确认逐步向前滑动。接收窗口允许接收乱序的数据包并缓存，当某个丢失的数据包重传并被接收后，接收窗口会移动，释放缓存的数据。

分析发送方。在发送数据包环节，发送方可以连续发送数据包，序号范围为  $[\text{base}, \text{base}+N-1]$ ，每个包单独等待确认 ack。在接收 ack 环节，每个包的确认独立进行，窗口会向前滑动，只要收到包序号小于 base 的 ack，说明包已被确认。在超时重传环节，发送方为每个数据维护一个定时器，如果某包超时未收到 ack，只重发该包。

分析接收方操作。在缓存环节，接收方可以接收乱序数据包并存入缓存，等待缺失的数据包到达。在发送独立 ack 环节，每接收一个包，就立即发送对应的 ack，告知发送方该包已成功接收。在滑动窗口环节，当连续序号的所有包到达时，接收窗口向前滑动，并释放缓存的数据。

## (2) 原理图

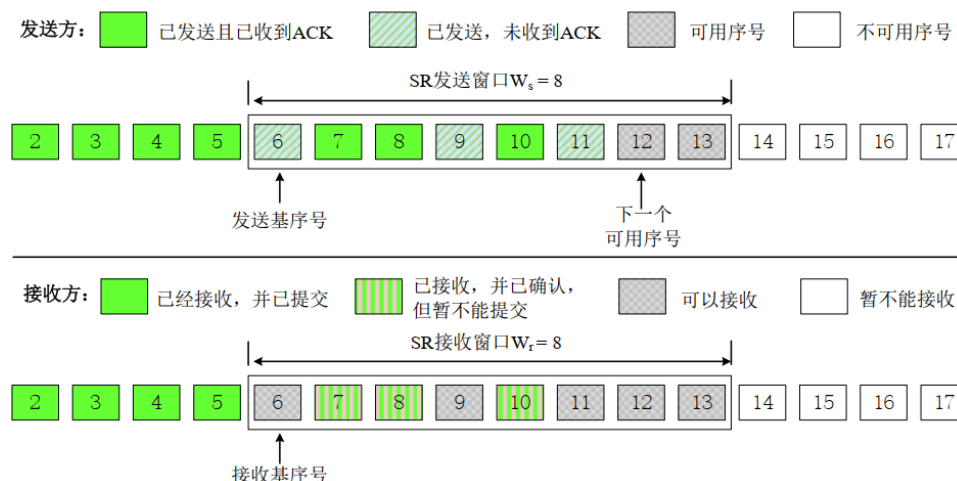


图 2-4 SR 协议结构图

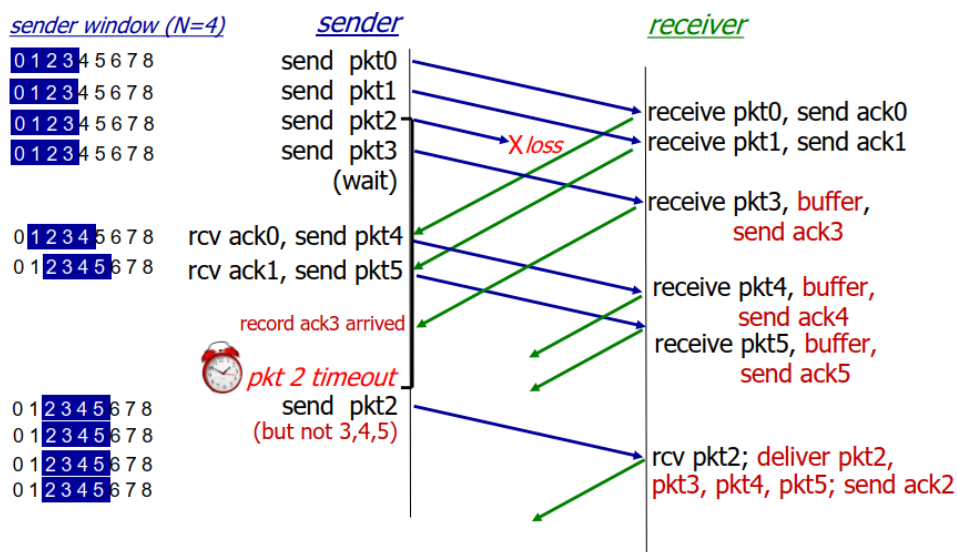


图 2-5 SR 协议流程图

### (3) 功能分析

#### (a) Deliver()函数

该函数处理已确认的数据包，并将其内容追加到指定的文件字符串中。当接收窗口中的数据已经被确认时，将该数据包标记为未使用，将数据包的内容追加到文件字符串中，并更新确认号。

#### (b) Send()函数

该函数从输入文件流中读取数据，并通过网络发送数据帧。获取当前时间，遍历发送窗口大小，计算当前发送窗口的索引，如果当前数据帧已经发送，跳过当前数据帧。如果当前数据帧未发送，从文件流中读取数据，更新当前数据帧的发送时间，设置数据帧的类型和序列号；如果当前数据帧是发送窗口的第一个数据帧，返回 0，表示发送完毕；如果当前数据帧不是发送窗口的第一个数据帧，跳出循环。如果当前数据帧发送超时，更新当前数据帧的发送时间，如果当前数据帧未发送超时，跳过当前数据帧。最后发送数据帧到指定的地址。

#### (c) MoveSendWindow()函数

该函数移动发送窗口，找到下一个可以发送的序列号。循环检查发送窗口中当前序列号 seq 对应的数据包状态，当当前序列号的数据帧已经发送，更新当前序列号的数据帧的发送时间，更新序列号。

#### (d) upload 命令

该命令的实现有三个阶段：请求上传阶段、发送文件数据阶段、等待完成确认阶段。

在请求上传阶段，如果接收到数据并且 100，并且数据内容 OK，则申请通过，准备状态；如果为 NO，则设置状态为-1 并跳出循环。获取时间，如果时间间隔大于 1 秒则更新发送时间。

在发送文件数据阶段，如果接收到数据，并且数据类型为 101，进行准备，如果确认号等于序列号，则开始交付数据。如果文件已发送完毕，则上传完成。

在等待完成确认阶段，如果接收到数据，并且数据类型为 100 并且数据内容为 OK，则发送数据。获取时间，如果时间间隔大于 1 秒则更新发送时间。

如果状态 status 为-1、为 3、时间差大于 5 秒，则结束通信。如果接收到的数据大小小于等于 0，则延时。

#### (e) download 命令

该命令的实现有三个阶段：请求下载阶段、开始下载数据阶段、处理数据帧阶段。

在请求下载阶段，如果接收到数据并且 100，并且数据内容 OK，则申请通过，准备状态；如果为 NO，则设置状态为-1 并跳出循环。获取时间，如果时间间隔大于 1 秒则更新发送时间。

在开始下载数据阶段，如果接收到数据，并且数据类型为 200，则开始下载。如果确认号等于序列号，则开始交付数据。如果文件已发送完毕，则上传完成。

在处理数据帧阶段，如果接收到数据且收到 200 时，则正常处理数据帧，如果收到 100 时，则表示传输结束。

如果 status 状态为-1、为 3、时间差大于 5 秒，则结束通信。如果接收到的数据大小小于等于 0，则延时。

#### (f) quit 命令

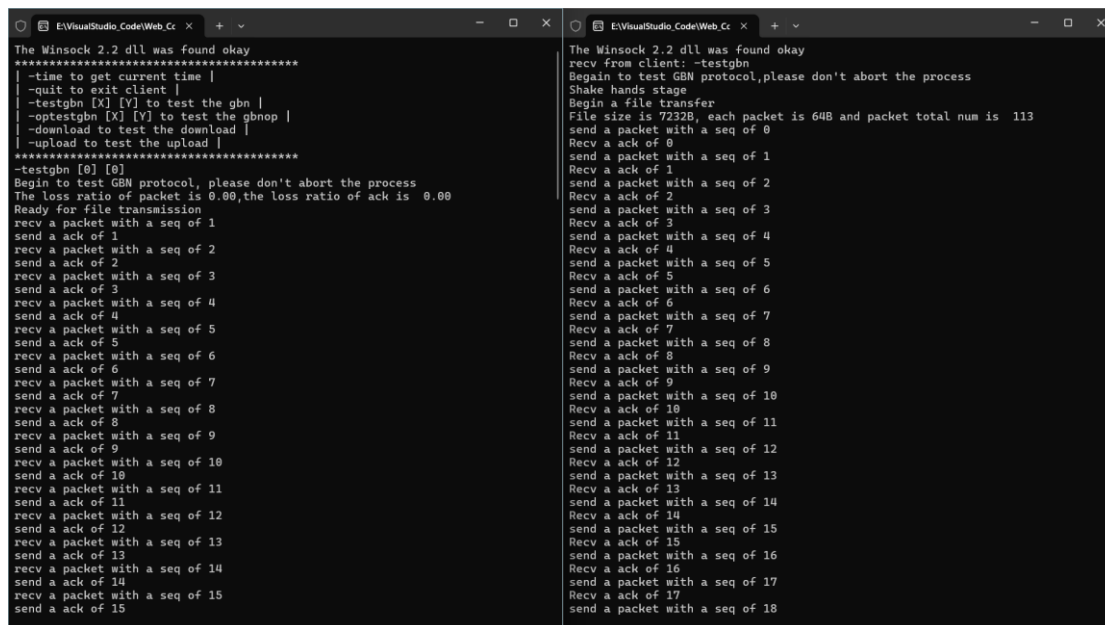
直接退出循环。



## 实验结果:

## (1) GBN 协议（实现单向可靠数据传输）

通过客户端和服务端的结果分析，看出这是从服务器到客户端的可靠数据传输，并且没有数据包丢失，因为客户端和服务端中 `recv` 和 `send` 的序号是连续的。



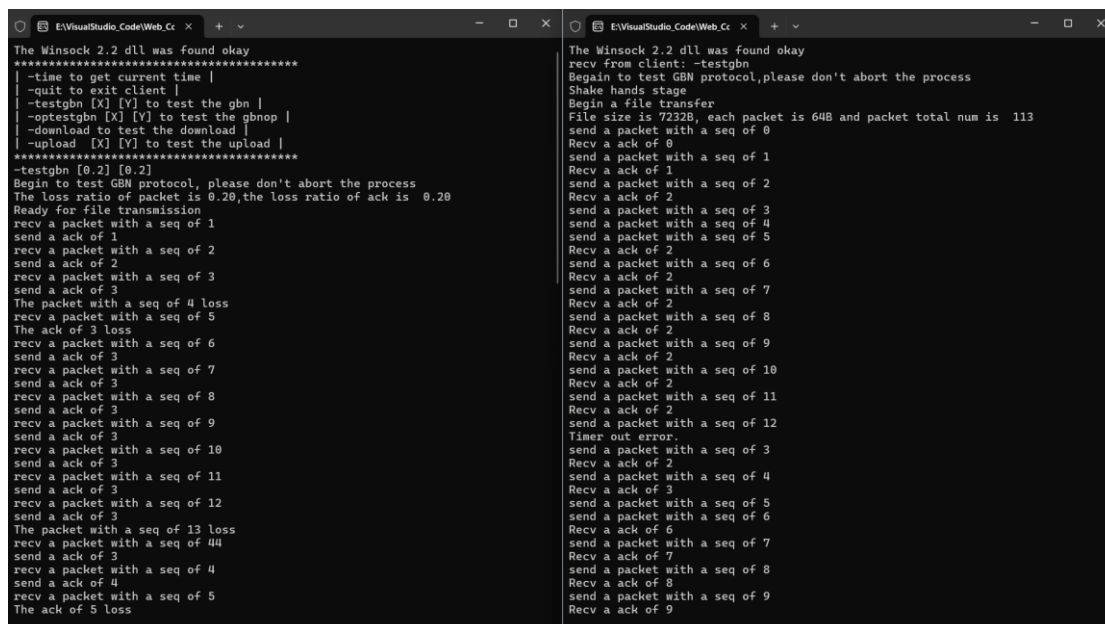
```
The Winsock 2.2 dll was found okay
*****
| -time to get current time |
| -quit to exit client |
| -testgbn [X] [Y] to test the gbn |
| -optestgbn [X] [Y] to test the gbnop |
| -download to test the download |
| -upload to test the upload |
*****
-testgbn [0] [0]
Begin to test GBN protocol, please don't abort the process
The loss ratio of packet is 0.00,the loss ratio of ack is 0.00
Ready for file transmission
recv a packet with a seq of 1
send a ack of 1
recv a packet with a seq of 2
send a ack of 2
recv a packet with a seq of 3
send a ack of 3
recv a packet with a seq of 4
send a ack of 4
recv a packet with a seq of 5
send a ack of 5
recv a packet with a seq of 6
send a ack of 6
recv a packet with a seq of 7
send a ack of 7
recv a packet with a seq of 8
send a ack of 8
recv a packet with a seq of 9
send a ack of 9
recv a packet with a seq of 10
send a ack of 10
recv a packet with a seq of 11
send a ack of 11
recv a packet with a seq of 12
send a ack of 12
recv a packet with a seq of 13
send a ack of 13
recv a packet with a seq of 14
send a ack of 14
recv a packet with a seq of 15
send a ack of 15

The Winsock 2.2 dll was found okay
recv from client: -testgbn
Begin to test GBN protocol,please don't abort the process
Shake hands stage
Begin a file transfer
File size is 7232B, each packet is 64B and packet total num is 113
send a packet with a seq of 0
Recv a ack of 0
send a packet with a seq of 1
Recv a ack of 1
send a packet with a seq of 2
Recv a ack of 2
send a packet with a seq of 3
Recv a ack of 3
send a packet with a seq of 4
Recv a ack of 4
send a packet with a seq of 5
Recv a ack of 5
send a packet with a seq of 6
Recv a ack of 6
send a packet with a seq of 7
Recv a ack of 7
send a packet with a seq of 8
Recv a ack of 8
send a packet with a seq of 9
Recv a ack of 9
send a packet with a seq of 10
Recv a ack of 10
send a packet with a seq of 11
Recv a ack of 11
send a packet with a seq of 12
Recv a ack of 12
send a packet with a seq of 13
Recv a ack of 13
send a packet with a seq of 14
Recv a ack of 14
send a packet with a seq of 15
Recv a ack of 15
send a packet with a seq of 16
Recv a ack of 16
send a packet with a seq of 17
Recv a ack of 17
send a packet with a seq of 18
```

图 3-1 GBN 协议（实现单向可靠数据传输）

## (2) GBN 协议（模拟数据包丢失）

通过客户端和服务端的结果分析，看出这是从服务器到客户端的含有数据包丢失的数据传输。在客户端中，看到在 `recv a packet with a seq of 5` 之后一直发送 `send a ack of 3`，而服务器仍然发送下一个数据，并且一直 `recv a ack of 2`，说明客户端没有接收到序号 3 的数据。在 `Timer out error` 后，开始重新发送 3，故分析和结果吻合。



```
The Winsock 2.2 dll was found okay
*****
| -time to get current time |
| -quit to exit client |
| -testgbn [X] [Y] to test the gbn |
| -optestgbn [X] [Y] to test the gbnop |
| -download to test the download |
| -upload [X] [Y] to test the upload |
*****
-testgbn [0.2] [0.2]
Begin to test GBN protocol, please don't abort the process
The loss ratio of packet is 0.20,the loss ratio of ack is 0.20
Ready for file transmission
recv a packet with a seq of 1
send a ack of 1
recv a packet with a seq of 2
send a ack of 2
recv a packet with a seq of 3
send a ack of 3
The packet with a seq of 4 loss
recv a packet with a seq of 5
The ack of 3 loss
recv a packet with a seq of 6
send a ack of 3
recv a packet with a seq of 7
send a ack of 3
recv a packet with a seq of 8
send a ack of 3
recv a packet with a seq of 9
send a ack of 3
recv a packet with a seq of 10
send a ack of 3
recv a packet with a seq of 11
send a ack of 3
recv a packet with a seq of 12
send a ack of 3
The packet with a seq of 13 loss
recv a packet with a seq of 14
send a ack of 3
recv a packet with a seq of 4
send a ack of 4
recv a packet with a seq of 5
The ack of 5 loss

The Winsock 2.2 dll was found okay
recv from client: -testgbn
Begin to test GBN protocol,please don't abort the process
Shake hands stage
Begin a file transfer
File size is 7232B, each packet is 64B and packet total num is 113
send a packet with a seq of 0
Recv a ack of 0
send a packet with a seq of 1
Recv a ack of 1
send a packet with a seq of 2
Recv a ack of 2
send a packet with a seq of 3
Recv a ack of 2
send a packet with a seq of 4
Recv a ack of 2
send a packet with a seq of 5
Recv a ack of 2
send a packet with a seq of 6
Recv a ack of 2
send a packet with a seq of 7
Recv a ack of 2
send a packet with a seq of 8
Recv a ack of 2
send a packet with a seq of 9
Recv a ack of 2
send a packet with a seq of 10
Recv a ack of 2
send a packet with a seq of 11
Recv a ack of 2
send a packet with a seq of 12
Timer out error.
send a packet with a seq of 3
Recv a ack of 2
send a packet with a seq of 4
Recv a ack of 3
send a packet with a seq of 5
send a packet with a seq of 6
Recv a ack of 6
send a packet with a seq of 7
Recv a ack of 7
send a packet with a seq of 8
Recv a ack of 8
send a packet with a seq of 9
Recv a ack of 9
```

图 3-2 GBN 协议（模拟数据包丢失）

## (3) GBN 协议（双向数据传输）



通过客户端和服务端的结果分析,看出这是从客户端到服务器端的含有数据包丢失的数据传输,因此完成双向数据传输的功能。在服务器端中,看到在 recv a packet with a seq of 20 之后一直发送 send a ack of 18,而客户端仍然发送下一个数据,并且一直 recv a ack of 17,说明客户端没有接收到序号 17 的数据。在 Timer out error 后,开始重新发送 17,故分析和结果吻合。

```

The Winsock 2.2 dll was found okay
*****
| -time to get current time |
| -quit to exit client |
| -testgbn [X] [Y] to test the gbn |
| -optestgbn [X] [Y] to test the gbnop |
| -download to test the download |
| -upload [X] [Y] to test the upload |
*****
-optestgbn [0.2] [0.2]
Beginn to test GBN protocol, please don't abort the process
Shake hands stage
Begin a file transfer
File size is 7232B, each packet is 1024B and packet total num is 113
send a packet with a seq of 0
Recv a ack of 0
send a packet with a seq of 1
Recv a ack of 1
send a packet with a seq of 2
Recv a ack of 2
send a packet with a seq of 3
Recv a ack of 3
send a packet with a seq of 4
Recv a ack of 4
send a packet with a seq of 5
Recv a ack of 5
send a packet with a seq of 6
Recv a ack of 6
send a packet with a seq of 7
Recv a ack of 7
send a packet with a seq of 8
Recv a ack of 8
send a packet with a seq of 9
Recv a ack of 9
send a packet with a seq of 10
Recv a ack of 10
send a packet with a seq of 11
Recv a ack of 11
send a packet with a seq of 12
Recv a ack of 12
send a packet with a seq of 13
Recv a ack of 13
send a packet with a seq of 14

recv from client: -optestgbn [0.2] [0.2]
Begin to test GBN protocol, please don't abort the process
The loss ratio of packet is 0.20, the loss ratio of ack is 0.20
Ready for file transmission
recv a packet with a seq of 1
send a ack of 1
recv a packet with a seq of 2
send a ack of 2
recv a packet with a seq of 3
send a ack of 3
recv a packet with a seq of 4
send a ack of 4
recv a packet with a seq of 5
send a ack of 5
recv a packet with a seq of 6
send a ack of 6
recv a packet with a seq of 7
send a ack of 7
recv a packet with a seq of 8
send a ack of 8
recv a packet with a seq of 9
send a ack of 9
recv a packet with a seq of 10
send a ack of 10
recv a packet with a seq of 11
send a ack of 11
recv a packet with a seq of 12
send a ack of 12
recv a packet with a seq of 13
send a ack of 13
recv a packet with a seq of 14
send a ack of 14
recv a packet with a seq of 15
send a ack of 15
recv a packet with a seq of 16
The ack of 16 loss
recv a packet with a seq of 17
send a ack of 17
recv a packet with a seq of 18
The ack of 18 loss
The packet with a seq of 19 loss
recv a packet with a seq of 20

send a packet with a seq of 14
Recv a ack of 14
send a packet with a seq of 15
send a packet with a seq of 16
Recv a ack of 16
send a packet with a seq of 17
send a packet with a seq of 18
send a packet with a seq of 19
Recv a ack of 17
send a packet with a seq of 0
send a packet with a seq of 1
send a packet with a seq of 2
Recv a ack of 17
send a packet with a seq of 3
send a packet with a seq of 4
Recv a ack of 17
send a packet with a seq of 5
send a packet with a seq of 6
Recv a ack of 17
send a packet with a seq of 7
Recv a ack of 17
Timer out error.

send a packet with a seq of 18
Recv a ack of 18
send a packet with a seq of 19
Recv a ack of 19
send a packet with a seq of 0
Recv a ack of 0
send a packet with a seq of 1
send a packet with a seq of 2
Recv a ack of 2
send a packet with a seq of 3
Recv a ack of 4
send a packet with a seq of 5
Recv a ack of 5
send a packet with a seq of 6
Recv a ack of 6
send a packet with a seq of 7
Recv a ack of 7

recv a packet with a seq of 20
send a ack of 18
recv a packet with a seq of 1
The ack of 18 loss
recv a packet with a seq of 2
The ack of 18 loss
recv a packet with a seq of 3
send a ack of 18
The packet with a seq of 4 loss
recv a packet with a seq of 5
send a ack of 18
recv a packet with a seq of 6
The ack of 18 loss
recv a packet with a seq of 7
send a ack of 18
recv a packet with a seq of 8
send a ack of 18
recv a packet with a seq of 19
send a ack of 19
recv a packet with a seq of 20
send a ack of 20
recv a packet with a seq of 1
send a ack of 1
recv a packet with a seq of 2
The ack of 2 loss
recv a packet with a seq of 3
send a ack of 3
recv a packet with a seq of 4
The ack of 4 loss
recv a packet with a seq of 5
send a ack of 5
recv a packet with a seq of 6
send a ack of 6
recv a packet with a seq of 7
send a ack of 7
recv a packet with a seq of 8
send a ack of 8
recv a packet with a seq of 9
send a ack of 9
recv a packet with a seq of 10
send a ack of 10
recv a packet with a seq of 11
The ack of 11 loss
  
```

图 3-3 GBN 协议（双向数据传输）

#### (4) GBN 协议（C/S 结构的文件传输应用）

##### (a) -download

通过客户端和服务端的结果分析,看出这是从服务器端到客户端的文件传输,即从服务器端下载文件到客户端。在客户端和服务端,可以看到有四个传输部分,即分 4 次传输文件。在客户端的结束部分,将 4 个部分的数据合并在一起,形成完整的文件数据。客户端文件与服务器端文件相比较,发现二者相同,证明功能正确。



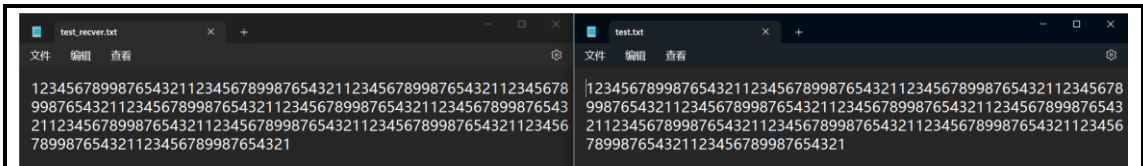


图 3-5 GBN 协议（C/S 结构的文件传输应用）upload 功能的流程页面（上）和数据页面（下）

(5) SR 协议

(1) download

通过客户端和服务端的结果分析，看出这是从服务器端到客户端的文件传输，即从服务器端下载文件到客户端。在客户端，可以看到每条信息均有 seq、data、发送 ack、起始 ack，即这些内容首尾相连组成完整的文件数据。在服务器端，可以看到有发送数据帧和接收信息，这些内容也能首尾相连组成完整的文件数据。客户端文件与服务器端文件相比较，发现二者相同，证明功能正确。

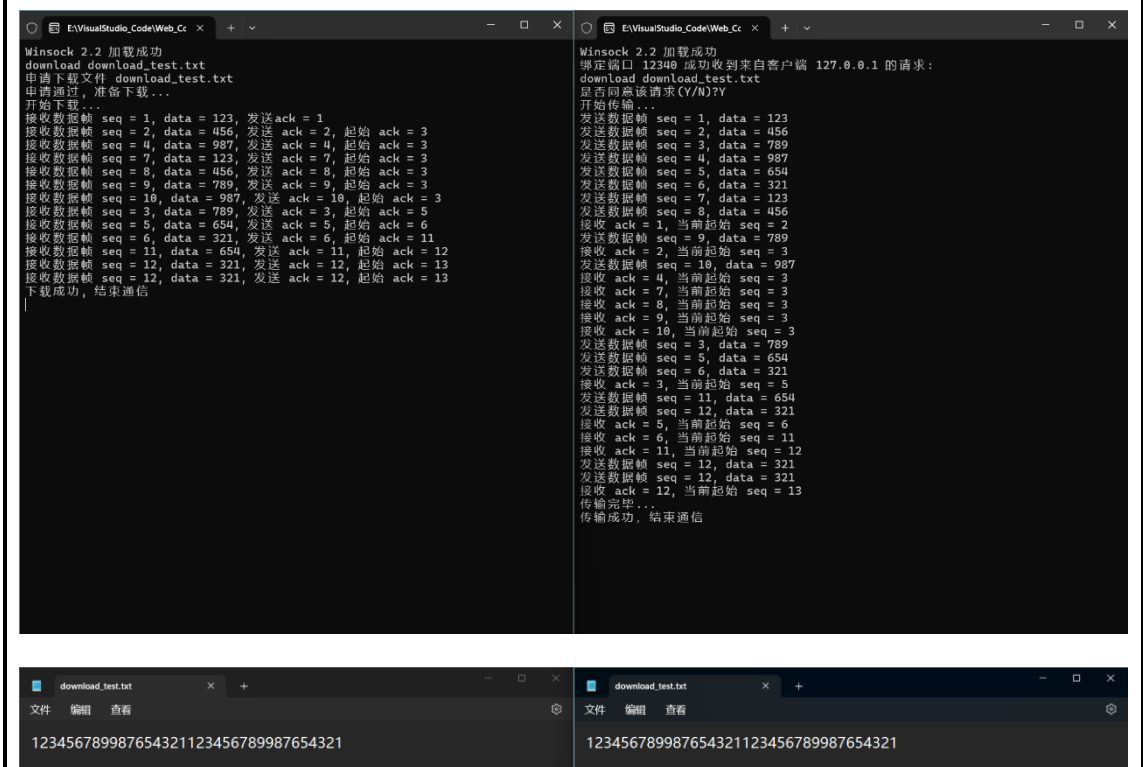


图 3-6 SR 协议 download 功能的流程页面（上）和数据页面（下）

(2) upload

通过客户端和服务端的结果分析，看出这是从客户端到服务器端的文件传输，即从客户端上传文件到服务器端。在服务器端，可以看到每条信息均有 seq、data、发送 ack、起始 ack，即这些内容首尾相连组成完整的文件数据。在客户端，可以看到有发送数据帧和接收信息，这些内容也能首尾相连组成完整的文件数据。客户端文件与服务器端文件相比较，发现二者相同，证明功能正确。

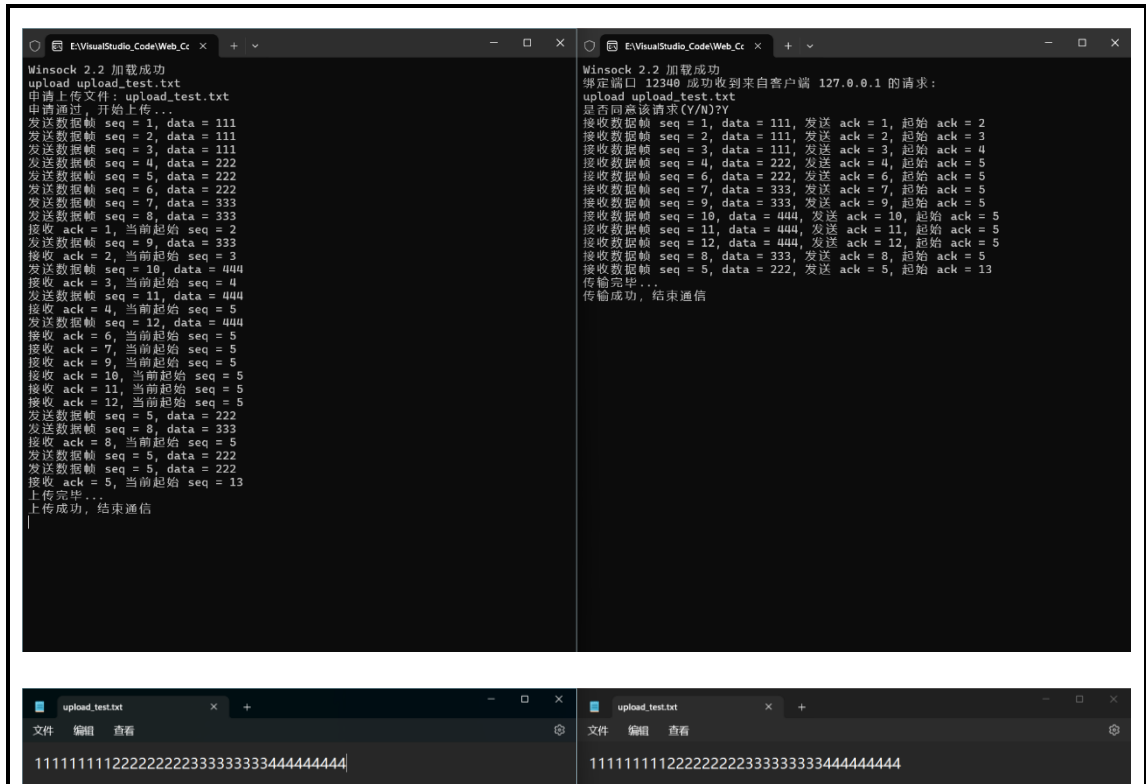


图 3-7 SR 协议 upload 功能的流程页面（上）和数据页面（下）

### 问题讨论:

(1) 针对等待超时时间，过长会使文件传输过慢；过短会导致文件明明并未丢包但是由于超时等待时间过短而导致其被判定为了丢包。针对GBN协议，也可以将超时的判定改为接受相同ACK的次数。

(2) SR协议主要为了解决在使用GBN的过程中的一定的资源浪费的情况，实上SR协议、GBN协议、停等协议这三者在本质上的区别就是接收窗口和发送窗口大小的区别。

(3) UDP协议是无连接的，无法保证可靠的数据传输，因此如果在上层使用，需要上层来实现可靠的数据传输。UDP在传输数据之前并不需要三次挥手建立连接，尽管在本次实验中一部分代码确实使用了类似于连接的方式实现，但是需要注意的是那只是逻辑上的连接并不存在真实的连接过程。

(4) SR协议为每一个发送窗口的数据都设置了一个计时器，每次都重超时的部分。在实际实现的过程中每次只需要比对发送窗口最低位是否超时，如果走时则重传，如果没有超时则表示发送窗口中没有超时的数据。

心得体会:

通过本次实验，我完成了可靠数据传输协议的设计与实现，进而理解可靠数据传输的基本原理；掌握停等协议的工作原理；掌握基于 UDP 设计并实现一个停等协议的过程与技术。理解滑动窗口协议的基本原理；掌握 GBN 的工作原理；掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术。进一步熟悉了 socket 编程中的更多细节，提高了代码能力。

## 具体实现代码

```
1. // TD_GBN_Client.cpp
2. #include <WS2tcpip.h>
3. #include <WinSock2.h>
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <time.h>
7.
8. #include <fstream>
9. #pragma comment(lib, "ws2_32.lib")
10. #define SERVER_PORT 12340 // 接收数据的端口号
11. #define SERVER_IP "127.0.0.1" // 服务器的 IP 地址
12. const int BUFFER_LENGTH = 1026;
13.
14. const int SEND_WIND_SIZE = 10; // 发送窗口大小为 10, GBN 中应满
    足  $W + 1 \leq N$  ( $W$ 
15. // 为发送窗口大小,  $N$  为序列号个
    数)
16. // 本例取序列号 0...19 共 20 个
17. // 如果将窗口大小设为 1, 则为停-等协议
18. const int SEQ_SIZE = 20; // 序列号的个数, 从 0~19 共计 20 个
19. // 由于发送数据第一个字节如果值为 0, 则数据会发送失败
20. // 因此接收端序列号为 1~20, 与发送端一一对应
21.
22. BOOL ack[SEQ_SIZE]; // 收到 ack 情况, 对应 0~19 的 ack
23. int curSeq; // 当前数据包的 seq
24. int curAck; // 当前等待确认的 ack
25. int totalSeq; // 收到的包的总数
26. int totalPacket; // 需要发送的包总数
27.
28. char fileName[40]; // 文件名
29. char filePath[50]; // 文件路径
30. char file[1024 * 1024]; // 文件内容
31. //*****
32. // Method: seqIsAvailable
33. // FullName: seqIsAvailable
34. // Access: public
35. // Returns: bool
36. // Qualifier: 当前序列号 curSeq 是否可用
37. //*****
38. bool seqIsAvailable() {
39.     int step; // 当前序列号与当前确认号的差值
40.     step = curSeq - curAck; // 当前序列号与当前确认号的差值
```

```
41.     step =
42.         step >= 0
43.         ? step
44.         : step +
45.         SEQ_SIZE; // 处理环形序列号，如果大于等于 0，则不变，否则加
    上
46.             // SEQ_SIZE
47. // 序列号是否在当前发送窗口之内
48. // 如果不在窗口内，则表示未收到 ack
49.     if (step >= SEND_WIND_SIZE) {
50.         return false;
51.     }
52.     // 如果收到 ack，则表示当前序列号可用
53.     if (ack[curSeq]) {
54.         return true;
55.     }
56.     // 如果未收到 ack，则表示当前序列号不可用
57.     return false;
58. }
59. //*****
60. // Method: timeoutHandler
61. // FullName: timeoutHandler
62. // Access: public
63. // Returns: void
64. // Qualifier: 超时重传处理函数，滑动窗口内的数据帧都要重传
65. //*****
66. void timeoutHandler() {
67.     Sleep(1000); // 等待 1s
68.     printf("Timer out error.\n\n\n"); // 超时重传
69.     int index; // 序列号
70.     for (int i = 0; i < SEND_WIND_SIZE; ++i) { // 遍历滑动窗口内的
        数据帧
71.         index = (i + curAck) % SEQ_SIZE; // 计算序列号
72.         ack[index] = TRUE; // 标记为重传
73.     }
74.     totalSeq -= SEND_WIND_SIZE; // 更新已发送的总序列号数
75.     curSeq = curAck; // 重置当前发送序列号
76. }
77. //*****
78. // Method: ackHandler
79. // FullName: ackHandler
80. // Access: public
81. // Returns: void
82. // Qualifier: 收到 ack，累积确认，取数据帧的第一个字节
```



```
83.// 由于发送数据时, 第一个字节(序列号)为
84.// 0 (ASCII) 时发送失败, 因此加一了, 此处需要减一还原
85.// Parameter: char c
86.//*****
87.void ackHandler(char c) {
88.    unsigned char index = (unsigned char)c - 1; // 序列号减一
89.    printf("Recv a ack of %d\n", index); // 输出收到的 ack 序列号
90.    // 当 curAck 小于等于 index 时, 表示 ack 没有回到 curAck 的左边
91.    if (curAck <= index) {
92.        // 依次标记 curAck 到 index 之间的 ack 为 TRUE
93.        for (int i = curAck; i <= index; ++i) {
94.            ack[i] = TRUE;
95.        }
96.        // 更新 curAck
97.        curAck = (index + 1) % SEQ_SIZE;
98.    }
99.    else {
100.        // ack 超过了最大值, 回到了 curAck 的左边
101.        for (int i = curAck; i < SEQ_SIZE; ++i) {
102.            ack[i] = TRUE; // 标记右边的 ack 为 TRUE
103.        }
104.        for (int i = 0; i <= index; ++i) {
105.            ack[i] = TRUE; // 标记左边的 ack 为 TRUE
106.        }
107.        curAck = index + 1; // 更新 curAck
108.    }
109.}
110.//*****
111.*/
111./* -time 从服务器端获取当前时间
112.-quit 退出客户端
113.-testgbn [X] 测试 GBN 协议实现可靠数据传输
114.[X] [0,1] 模拟数据包丢失的概率
115.[Y] [0,1] 模拟 ACK 丢失的概率
116.*/
117.//*****
118.*/
118.void printTips() {
119.    printf_s("*****\n");
120.    printf_s("| -time to get current time |\n");
121.    printf_s("| -quit to exit client |\n");
122.    printf_s("| -testgbn [X] [Y] to test the gbn |\n");
123.    printf_s("| -optestgbn [X] [Y] to test the gbnop |\n");
```

```
124.     printf_s("| -download to test the download |\n");
125.     printf_s("| -upload to test the upload |\n");
126.     printf_s("*****\n");
127. }
128. //*****
129. // Method: lossInLossRatio
130. // FullName: lossInLossRatio
131. // Access: public
132. // Returns: BOOL
133. // Qualifier: 根据丢失率随机生成一个数字, 判断是否丢失, 丢失则返回
    TRUE, 否则返回
134. // FALSE Parameter: float lossRatio [0,1]
135. //*****
136. BOOL lossInLossRatio(float lossRatio) {
137.     // 将传入的丢失率转换为 0~100 的整数
138.     int lossBound = (int)(lossRatio * 100);
139.     int r = rand() % 101; // 生成 0~100 的随机数
140.     if (r <= lossBound) { // 如果随机数小于等于丢失率
141.         return TRUE;
142.     }
143.     return FALSE; // 否则不丢失
144. }
145. int main(int argc, char* argv[]) {
146.     // 加载套接字库 (必须)
147.     WORD wVersionRequested;
148.     WSADATA wsaData;
149.     // 套接字加载时错误提示
150.     int err;
151.     // 版本 2.2
152.     wVersionRequested = MAKEWORD(2, 2);
153.     // 加载 dll 文件 Scket 库
154.     err = WSAStartup(wVersionRequested, &wsaData);
155.     if (err != 0) {
156.         // 找不到 winsock.dll
157.         printf("WSAStartup failed with error: %d\n", err);
158.         return 1;
159.     }
160.     // 检测版本号
161.     if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion
        ) != 2) {
162.         printf("Could not find a usable version of Winsock.dll\n
            ");
163.         WSACleanup();
164.     }
```

```
165.     else {
166.         printf("The Winsock 2.2 dll was found okay\n");
167.     }
168.     // 创建套接字
169.     SOCKET socketClient = socket(AF_INET, SOCK_DGRAM, 0);
170.     SOCKADDR_IN addrServer;
171.
172.     // int length = sizeof(SOCKADDR);
173.     // struct in_addr p;
174.     inet_pton(AF_INET, SERVER_IP, &addrServer.sin_addr); // 服务器的 IP
175.     // printf("\n%d\n\n", addrServer.sin_addr.S_un.S_addr);
176.     addrServer.sin_family = AF_INET; // 协议
177.     addrServer.sin_port = htons(SERVER_PORT); // 端口
178.     // 接收缓冲区
179.     char buffer[BUFFER_LENGTH]; // 缓冲区
180.     ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲区
181.     int len = sizeof(SOCKADDR); // 地址长度
182.     // 为了测试与服务器的连接, 可以使用 -time 命令从服务器端获得当前时间
183.     // 使用 -testgbn [X] [Y] 测试 GBN 其中[X]表示数据包丢失概率
184.     // [Y]表示 ACK 丢包概率
185.     printTips(); // 打印提示信息
186.     int ret; // 返回值
187.     int interval = 1; // 收到数据包之后返回 ack 的间隔, 默认为 1 表示每个都返回
188.     // ack, 0 或者负数均表示所有的都不返回 ack
189.     char cmd[128]; // 命令
190.     float packetLossRatio = 0.2; // 默认包丢失率 0.2
191.     float ackLossRatio = 0.2; // 默认 ACK 丢失率 0.2
192.     // 用时间作为随机种子, 放在循环的最外面
193.     // 将测试数据读入内存
194.     std::ifstream infile; // 输入流
195.     std::ofstream outfile; // 输出流
196.     infile.open("./test.txt"); // 打开文件
197.     char data[64 * 113]; // 113 个包
198.     ZeroMemory(data, sizeof(data)); // 初始化
199.     infile.read(data, 64 * 113); // 读取文件
200.     infile.close(); // 关闭文件
201.     // printf("\n%s\n\n", data);
202.     totalPacket = sizeof(data) / 64; // 计算包的总数
203.     // ZeroMemory(data, sizeof(data));
204.     int recvSize; // 接收到的数据包大小
205.     // 初始化 ack 数组
```

```
206.     for (int i = 0; i < SEQ_SIZE; ++i) {
207.         ack[i] = TRUE;
208.     }
209.     // 设置随机种子
210.     srand((unsigned)time(NULL));
211.     while (true) {
212.         gets_s(buffer, BUFFER_LENGTH); // 从键盘输入命令
213.         // 解析命令
214.         ret = sscanf_s(buffer, "%s [%f] [%f]", &cmd, (unsigned)sizeof(cmd), &packetLossRatio, &ackLossRatio);
215.         // printf("\n%s\n\n", buffer);
216.
217.         // 开始 GBN 测试, 使用 GBN 协议实现 UDP 可靠文件传输
218.         if (!strcmp(cmd, "-testgbn")) {
219.             // 进入 gbn 测试阶段
220.             // 开始测试 GBN 协议, 请不要中断进程
221.             printf("%s\n",
222.                 "Begin to test GBN protocol, please don't abort
the "
223.                 "process");
224.             // 数据包的丢失率和 ack 的丢失率的输出
225.             printf(
226.                 "The loss ratio of packet is %.2f,the loss ratio
of ack is % "
227.                 ".2f\n",
228.                 packetLossRatio, ackLossRatio);
229.             int waitCount = 0; // 等待计数器
230.             int stage = 0; // 阶段
231.             BOOL b; // 丢包标志
232.             unsigned char u_code; // 状态码
233.             unsigned short seq; // 包的序列号
234.             unsigned short recvSeq; // 接收窗口大小为 1, 已确认的
序列号
235.             unsigned short waitSeq; // 等待的序列号
236.             // 发送测试命令
237.             sendto(socketClient, "-testgbn", strlen("-
testgbn") + 1, 0,
238.                 (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
239.             while (true) {
240.                 // 等待 server 回复设置 UDP 为阻塞模式
241.                 // Sleep(500);
242.                 // 循环等待数据, 通过 recvfrom 阻塞等待服务器发送的数据包, 直到接收到数据包, 循环才会退出
```

```
243.          // recvfrom 函数用来接收数据，接收到数据后，将数据存
           放到 buffer 中，获取服务器的地址信息
244.          while (recvfrom(socketClient, buffer, BUFFER_LEN
           GTH, 0,
245.                  (SOCKADDR*)&addrServer, &len) == -1);
246.          // printf("\n%s\n\n", &buffer[1]);
247.          // memcpy(data + 64 * totalSeq, &buffer[1], 64);

248.          switch (stage) {
249.          case 0: // 等待握手阶段
250.              u_code = (unsigned char)buffer[0]; // 状态
           码
251.              // 如果收到 205，则表示服务器准备好了，可以发送
           数据
252.              if ((unsigned char)buffer[0] == 205) {
253.                  // 服务器准备好了，可以发送数据
254.                  printf("Ready for file transmission\n");

255.                  // 客户端发送 200
256.                  // 状态码，表示客户端准备好了，可以接收数据
           了
257.                  buffer[0] = 200;
258.                  buffer[1] = '\0';
259.                  // sendto 函数用来发送数据，发送数据到服务器
           端,发送成功返回发送的字节数，否则返回-1
260.                  sendto(socketClient, buffer, 2, 0,
261.                          (SOCKADDR*)&addrServer, sizeof(SOCKA
           DDR));
262.                  stage = 1; // 进入等待接收数据阶段
263.                  recvSeq = 0; // 接收窗口大小为 1，已确认
           的序列号
264.                  waitSeq = 1; // 等待的序列号
265.              }
266.              break;
267.          case 1: // 等待接收数据阶段
268.              // 如果收到的数据包为空，则表示文件传输结束
269.              if (&buffer[1] == NULL) {
270.                  break;
271.              }
272.              // 获取包的序列号
273.              seq = (unsigned short)buffer[0];
274.              // 随机法模拟包是否丢失
275.              b = lossInLossRatio(packetLossRatio);
276.              // 如果丢包，则不处理
```

```
277.         if (b) {
278.             // 输出丢包的序列号
279.             printf("The packet with a seq of %d loss\n", seq);
280.             continue; // 如果丢包，则忽略该包
281.         }
282.         // 输出接收到的包的序列号
283.         printf("recv a packet with a seq of %d\n", seq);
284.         // 如果是期待的包，正确接收，正常确认即可
285.         if (!(waitSeq - seq)) { // 如果是期待的序列号
286.             ++waitSeq; // 更新期待的下一个序列号
287.             if (waitSeq == 21) { // 如果超过了最大值
288.                 waitSeq = 1; // 回到 1
289.             }
290.             // 输出数据
291.             // printf("%s\n",&buffer[1]);
292.             buffer[0] = seq; // 序列号
293.             recvSeq = seq; // 更新已接收的序列号
294.             buffer[1] = '\0'; // 结束符
295.         }
296.         else {
297.             // 如果当前一个包都没有收到，则等待 Seq 为 1
298.             // 的数据包，不是则不返回
299.             // ACK（因为并没有上一个正确的 ACK）
300.             if (!recvSeq) { // 如果尚未接收到任何包，只能等待序列号为 1 的包
301.                 continue;
302.             }
303.             // 否则（收到乱序的包），则返回上一次正确接收的序列号的
304.             // ACK
305.             buffer[0] = recvSeq;
306.             buffer[1] = '\0';
307.         }
308.         // 随机法模拟 ACK 是否丢失
309.         b = lossInLossRatio(ackLossRatio);
310.         // 如果 ACK 丢失，则不处理
311.         if (b) {
312.             printf("The ack of %d loss\n",
313.                 (unsigned char)buffer[0]);
```



```
314.             continue; // 如果 ACK 丢失, 则跳过发送
315.         }
316.         // 如果 ACK 未丢失, 则通过 sendto 发送
317.         // ACK 给服务器, 并打印 ACK 日志
318.         while (sendto(socketClient, buffer, 2, 0,
319.             (SOCKADDR*)&addrServer,
320.             sizeof(SOCKADDR)) == -1);
321.         // 输出 ACK
322.         printf("send a ack of %d\n", (unsigned char)
323.             buffer[0]);
324.         break;
325.     }
326.     Sleep(500); // 模拟网络延迟, 等待 500ms
327. }
328. else if (!strcmp(cmd, "-optestgbn")) {
329.     // 进入 gbn 测试阶段
330.     // 首先 server (server 处于 0 状态) 向 client 发
331.     // 送 205
332.     // 状态码 (server 进入 1 状态)
333.     // server 等待 client 回复 200 状态码, 如果收到
334.     // (server 进入
335.     // 2 状态), 则开始传输文件, 否则延时等待直至超时
336.     // 在文件传输阶段, server 发送窗口大小设为
337.     // sendto 用来发送数据, 发送数据到服务器端, 发送成功返回发
338.     // 送的字节数, 否则返回-1
339.     sendto(socketClient, buffer, strlen(buffer) + 1, 0,
340.         (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
341.     // ZeroMemory(buffer, sizeof(buffer));
342.     int recvSize; // 接收到的数据包大小
343.     int waitCount = 0; // 等待计数器
344.     // 输出提示信息
345.     printf(
346.         "Begain to test GBN protocol, please don't abort
347.         the process\n");
348.     // 加入了一个握手阶段
349.     // 首先服务器向客户端发送一个 205
350.     // 大小的状态码 (我自己定义的) 表示服务器准备好了, 可以发
351.     // 送数据
352.     // 客户端收到 205 之后回复一个 200
353.     // 大小的状态码, 表示客户端准备好了, 可以接收数据了 服务
354.     // 器收到 200
355.     // 状态码之后, 就开始使用 GBN 发送数据了
```

```
350.         printf("Shake hands stage\n");
351.         int stage = 0;           // 用于跟踪握手和数据传输阶段
352.         bool runFlag = true;    // 控制 while 循环是否运行
353.         while (runFlag) {
354.             switch (stage) {
355.                 case 0: // 发送 205
356.                     // 进行握手，表示客户端准备好了可以开始传
357.                     buffer[0] = 205; // 状态码
358.                     sendto(socketClient, buffer, strlen(buffer)
359.                         + 1, 0,
360.                         (SOCKADDR*)&addrServer, sizeof(SOCKADDR)
361.                         );
362.                     Sleep(100); // 等待 100ms
363.                     stage = 1; // 进入等待接收数据阶段
364.                     break;
365.                 case 1: // 等待接收 200
366.                     // 阶段，没有收到则计数器+1，超时则放弃此次
367.                     // “连接”，等待从第一步开始
368.                     recvSize = recvfrom(socketClient, buffer, BU
369.                         FFER_LENGTH,
370.                         0, ((SOCKADDR*)&addrServer), &len);
371.                     if (recvSize < 0) {
372.                         ++waitCount;           // 等待计数器+1
373.                         if (waitCount > 20) { // 如果等待计数器超
374.                             // 过 20
375.                             runFlag = false; // 超时，放弃此次连
376.                             // 接
377.                             printf("Timeout error\n"); // 输出超
378.                             // 时错误
379.                             break;
380.                         }
381.                         Sleep(500); // 等待 500ms 后重试
382.                         continue;
383.                     }
384.                     else {
385.                         // 当收到 200 状态码时，客户端进入状态 2
386.                         if ((unsigned char)buffer[0] == 200) {
387.                             // 输出提示信息
388.                             printf("Begin a file transfer\n");
389.                             // 输出文件大小
390.                             printf(
391.                                 "File size is %dB, each packet i
392.                                 s 1024B "
```

```
385.             "and packet total num is % d\n",
386.             sizeof(data), totalPacket);
387.             curSeq = 0;      // 当前数据包的序列
   号
388.             curAck = 0;      // 当前等待确认
   的 ack
389.             totalSeq = 0;    // 收到的包的总数
390.             waitCount = 0;   // 等待计数器
391.             stage = 2;       // 进入数据传输阶段
392.         }
393.     }
394.     break;
395. case 2: // 数据传输阶段
396.     // 检查是否可以发送该序列号的数据
397.     if (seqIsAvailable()) {
398.         // 发送给服务器的序列号从 1 开始
399.         buffer[0] = curSeq + 1;
400.         // 标记该包未被确认
401.         ack[curSeq] = FALSE;
402.         // 数据发送的过程中应该判断是否传输完成
403.         // 为简化过程此处并未实现
404.         // memcpy(&buffer[1], data + 1024 * tota
   lSeq, 1024);
405.
406.         while (sendto(socketClient, buffer, BUFF
   ER_LENGTH,
407.             0, (SOCKADDR*)&addrServer,
408.             sizeof(SOCKADDR)) == -1);
409.         // 输出发送的包的序列号
410.         printf("send a packet with a seq of %d\n
   ", curSeq);
411.
412.         ++curSeq;           // 更新当前序列号
413.         curSeq %= SEQ_SIZE; // 序列号循环使用
414.         ++totalSeq;        // 更新已发送的包的总数
415.         // Sleep(500);
416.         // 接收 ack
417.         recvfrom(socketClient, buffer, BUFFER_LE
   NGTH, 0,
418.             ((SOCKADDR*)&addrServer), &len);
419.     }
420.     else { // 如果超时，则重传
421.         timeoutHandler();
```

```
422.             break;
423.             // waitCount = 0;
424.         }
425.         // 等待 Ack, 若没有收到, 则返回值为-1, 计数器
+1
426.
427.         // printf("\n%d\n\n", buffer[0]);
428.         recvSize = buffer[0]; // 接收到的 ack
429.         // 如果未收到 ack, 则等待计数器+1
430.         if (recvSize < 0) {
431.             waitCount++; // 等待计数器+1
432.             // printf("waitokdqowjfq\n");
433.             // 20 次等待 ack 则超时重传
434.             if (waitCount > 20) {
435.                 // 超时处理并重发
436.                 timeoutHandler();
437.                 // 重置等待计数器
438.                 waitCount = 0;
439.             }
440.         }
441.         else { // 如果收到 ack, 则处理 ack
442.             // 收到 ack
443.             // printf("\n%d\n\n", buffer[0]);
444.             // 调用 ackhandler 处理 ack
445.             ackHandler(buffer[0]);
446.             // 重置等待计数器
447.             waitCount = 0;
448.         }
449.         // Sleep(500);
450.         break;
451.     }
452. }
453. }
454. else if (!strcmp(cmd, "-download")) {
455.     // 进入 download 测试阶段
456.     printf(
457.         "%s\n",
458.         "Begin to test GBN protocol, please don't abort
the process");
459.     int waitCount = 0; // 等待计数器
460.     int stage = 0;    // 阶段
461.     BOOL b;           // 丢包标志
462.     BOOL flag = true; // 控制循环是否运行
463.     unsigned char u_code; // 状态码
```

```
464.         unsigned short seq;    // 包的序列号
465.         unsigned short recvSeq; // 接收窗口大小为 1, 已确认的
           序列号
466.         unsigned short waitSeq; // 等待的序列号
467.         // 向服务器发送 -download 命令
468.         sendto(socketClient, "-download", strlen("-
           download") + 1, 0,
469.             (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
470.         while (flag) {
471.             // 等待 server 回复设置 UDP 为阻塞模式
472.             // Sleep(500);
473.             if (stage == 2) { //
474.                 // 存储文件
475.                 printf("Finish\n"); // 输出文件传输结束
476.                 printf("\ndata:\n%s\n\n", &data[0]); // 输出
           文件内容
477.                 FILE* out; // 输出
           文件
478.                 // 如果文件打开成功, 则写入文件
479.                 if (fopen_s(&out, "test_recver.txt", "wb") =
           = 0) {
480.                     // 写入文件
481.                     fwrite(data, sizeof(char), strlen(data),
           out);
482.                     // 关闭文件
483.                     fclose(out);
484.                 }
485.                 flag = false; // 结束循环
486.                 break;
487.             }
488.             // 循环等待数据, 通过 recvfrom 阻塞等待服务器发送的数
           据包
489.             // 直到接收到数据包, 循环才会退出
490.             while (recvfrom(socketClient, buffer, BUFFER_LEN
           GTH, 0,
491.                 (SOCKADDR*)&addrServer, &len) == -1);
492.             // 如果服务器发送状态码为 300, 客户端存储数据并退
           出
493.             if (buffer[0] == 300) stage = 2;
494.             // printf("\nbuffer:%s\n\n", &buffer[1]);
495.             ZeroMemory(data + 64 * totalSeq, 64); // 初始
           化 data
496.             // 将数据存储在 data
497.             memcpy(data + 64 * totalSeq, &buffer[1], 64);
```

```
498.          // 输出 data
499.          //printf("\ndata:%s\n\n", data + 64 * totalSeq);

500.          printf("\ndata:%s\n\n", &buffer[1]);
501.          switch (stage) {
502.          case 0: // 等待握手阶段
503.                  // 获取状态码
504.                  u_code = (unsigned char)buffer[0];
505.                  // 如果收到 205, 表示服务器已准备好进行文件传
输
506.                  if ((unsigned char)buffer[0] == 205) {
507.                      printf("Ready for file transmission\n");

508.                      // 客户端发送 200 以确认表示准备好接收数据
509.                      buffer[0] = 200;
510.                      buffer[1] = '\0'; // 结束符
511.                      // 发送数据
512.                      sendto(socketClient, buffer, 2, 0,
513.                          (SOCKADDR*)&addrServer, sizeof(SOCKA
DDR));
514.                      stage = 1; // 进入等待接收数据阶段
515.                      recvSeq = 0; // 接收窗口大小为 1, 已确认
的序列号
516.                      waitSeq = 1; // 等待的序列号
517.                  }
518.                  break;
519.                  // 每次收到一个数据包后, 客户端检查其序列号是否
与预期一致
520.                  // 若包的序列号正确, 发送对应的
521.                  // ACK。若不符合预期, 客户端根据上一个确认的序列
号发送 ACK
522.          case 1: // 等待接收数据阶
段
523.                  if (buffer[1] == '\0') { // 如果收到的数据包
为空
524.                      stage = 2; // 进入结束阶段
525.                      break;
526.                  }
527.                  // 获取包的序列号
528.                  seq = (unsigned short)buffer[0];
529.                  // 随机法模拟包是否丢失
530.                  /*b = lossInLossRatio(packetLossRatio);
531.                  if (b) {
```



```
532.             printf("The packet with a seq of %d loss\n", seq);
533.             continue;
534.         }*/
535.         // 输出接收到的包的序列号
536.         printf("recv a packet with a seq of %d\n", seq);
537.         ++totalSeq; // 更新已接收的包的总数
538.         // 如果是期待的包，正确接收，正常确认即可
539.         if (!(waitSeq - seq)) { // 如果是期待的序列号
540.             ++waitSeq; // 更新期待的下一个序列号
541.             if (waitSeq == 21) { // 如果超过了最大值
542.                 waitSeq = 1; // 回到 1
543.             }
544.             // 输出数据
545.             // printf("%s\n",&buffer[1]);
546.             buffer[0] = seq; // 序列号
547.             recvSeq = seq; // 更新已接收的序列号
548.             buffer[1] = '\0'; // 结束符
549.         }
550.         else {
551.             // 如果当前一个包都没有收到，则等待 Seq 为 1
552.             // 的数据包，不是则不返回
553.             // ACK（因为并没有上一个正确的 ACK）
554.             if (!recvSeq) { // 如果尚未接收到任何包，只能等待序列号为 1 的包
555.                 continue;
556.             }
557.             // 如果乱序，返回上一次正确接收的序列号的 ACK
558.             buffer[0] = recvSeq;
559.             buffer[1] = '\0'; // 结束符
560.         }
561.         /*b = lossInLossRatio(ackLossRatio);
562.         if (b) {
563.             printf("The ack of %d loss\n", (unsigned
564.                 char)buffer[0]); continue;
565.         }*/
566.         // 发送 ACK
567.         while (sendto(socketClient, buffer, 2, 0,
```

```
568.                (SOCKADDR*)&addrServer,
569.                sizeof(SOCKADDR)) == -1);
570.                // 输出 ACK
571.                printf("send a ack of %d\n", (unsigned char)
    buffer[0]);
572.                break;
573.                case 2:                                // 结束阶段
574.                    totalSeq -= SEND_WIND_SIZE; // 更新已发送的
    包的总数
575.                    break;
576.                }
577.                Sleep(500); // 模拟网络延迟, 等待 500ms
578.            }
579.        }
580.        else if (!strcmp(cmd, "-upload")) {
581.            // 进入 gbn 测试阶段
582.            // 首先 server (server 处于 0 状态) 向 client 发
    送 205
583.            // 状态码 (server 进入 1 状态)
584.            // server 等待 client 回复 200 状态码, 如果收到
    (server 进入
585.            // 2 状态), 则开始传输文件, 否则延时等待直至超时
586.            // 在文件传输阶段, server 发送窗口大小设为
587.            // sendto 用来发送数据, 发送数据到服务器端, 准备进入握手阶
    段发送成功返回发送的字节数, 否则返回-1
588.            sendto(socketClient, buffer, strlen(buffer) + 1, 0,
    (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
589.            // ZeroMemory(buffer, sizeof(buffer));
590.            int recvSize; // 接收到的数据包大小
591.            int waitCount = 0; // 等待计数器
592.            // 输出提示信息
593.            printf(
594.                "Begin to test GBN protocol, please don't abort
    the process\n");
595.            // 加入了一个握手阶段
596.            // 首先服务器向客户端发送一个 205
597.            // 大小的状态码 (我自己定义的) 表示服务器准备好了, 可以发
    送数据
598.            // 客户端收到 205 之后回复一个 200
599.            // 大小的状态码, 表示客户端准备好了, 可以接收数据了 服
    务器收到 200
600.            // 状态码之后, 就开始使用 GBN 发送数据了
601.            printf("Shake hands stage\n");
602.
```



```
638.                sizeof(data), totalPacket);
639.                curSeq = 0;    // 当前数据包的序列
   号
640.                curAck = 0;    // 当前等待确认
   的 ack
641.                totalSeq = 0;  // 收到的包的总数
642.                waitCount = 0; // 等待计数器
643.                stage = 2;     // 进入数据传输阶段
644.            }
645.        }
646.        break;
647.    case 2:        // 数据传输阶段
648.        if (seqIsAvailable()) { // 检查序列号是否可
   用
649.            // 如果可用, 则准备数据包
650.            // 发送给服务器的序列号从 1 开始
651.            buffer[0] = curSeq + 1;
652.            // 标记该包未被确认
653.            ack[curSeq] = FALSE;
654.            // 数据发送的过程中应该判断是否传输完成
655.            // 为简化过程此处并未实现
656.            // 将数据存储到 buffer
657.            memcpy(&buffer[1], data + 64 * totalSeq,
   64);
658.            // 输出 buffer
659.            printf("\nbuffer:\n%s\n\n", data + 64 * t
   otalSeq);
660.            if (buffer[1] == '\0') { // 如果收到的数
   据包为空
661.                printf("Finish\n"); // 输出文件传输
   结束
662.                stage = 3;          // 进入结束阶
   段
663.                break;
664.            }
665.            // 发送数据包, 直到成功为止, 并打印发送的序
   列号
666.            while (sendto(socketClient, buffer, BUFF
   ER_LENGTH,
667.                0, (SOCKADDR*)&addrServer,
668.                sizeof(SOCKADDR)) == -1);
669.            printf("send a packet with a seq of %d\n
   ", curSeq);
670.
```

```
671.          ++curSeq;          // 更新当前序列号
672.          curSeq %= SEQ_SIZE; // 序列号循环使用
673.          ++totalSeq; // 更新已发送的包的总数
674.          // Sleep(500);
675.          // 接收 ack
676.          recvfrom(socketClient, buffer, BUFFER_LE
    NGTH, 0,
677.                  ((SOCKADDR*)&addrServer), &len);
678.          }
679.          else { // 如果序列号不可用, 则重传
680.              timeoutHandler(); // 超时处理
681.              break;
682.              // waitCount = 0;
683.          }
684.          // 等待 Ack, 若没有收到, 则返回值为-1, 计数器
    +1
685.
686.          // printf("\n%d\n\n", buffer[0]);
687.          recvSize = buffer[0]; // 接收到的 ack
688.          if (recvSize < 0) { // 如果未收到 ack
689.              waitCount++; // 等待计数器+1
690.              // printf("waitokdqowjfq\n");
691.              // 20 次等待 ack 则超时重传
692.              if (waitCount > 20) { // 如果等待计数器超
    过 20
693.                  timeoutHandler(); // 超时处理并重
    发
694.                  waitCount = 0; // 重置等待计数
    器
695.              }
696.          }
697.          else { // 如果收到 ack
698.              // 收到 ack
699.              // printf("\n%d\n\n", buffer[0]);
700.              ackHandler(buffer[0]); // 调用
    ackhandler 处理 ack
701.              waitCount = 0; // 重置等待计数
    器
702.          }
703.          // Sleep(500);
704.          break;
705.          case 3: // 结束阶段
706.              printf("Finish\n"); // 输出文件传输结束
707.              runFlag = false; // 结束循环
```

```
708.         }
709.     }
710. }
711. // printf("\ndata:%s\n\n", data);
712. // 发送数据
713. sendto(socketClient, buffer, strlen(buffer) + 1, 0,
714.         (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
715. // 接收数据
716. ret = recvfrom(socketClient, buffer, BUFFER_LENGTH, 0,
717.                (SOCKADDR*)&addrServer, &len);
718. // 输出数据
719. printf("%s\n", buffer);
720. // 退出客户端
721. if (!strcmp(buffer, "Good bye!")) {
722.     break;
723. }
724. printTips();
725. }
726. // 关闭套接字
727. closesocket(socketClient);
728. WSACleanup();
729. return 0;
730.}

1. // TD_GBN_Server.cpp
2. #include <WS2tcpip.h>
3. #include <WinSock2.h>
4. #include <stdlib.h>
5. #include <time.h>
6.
7. #include <fstream>
8. #pragma comment(lib, "ws2_32.lib")
9. #define SERVER_PORT 12340 // 端口号
10. #define SERVER_IP "0.0.0.0" // IP 地址
11. const int BUFFER_LENGTH =
12. 1026; // 缓冲区大小, (以太网中 UDP 的数据帧中包长度应小于 1480 字
    节)
13. const int SEND_WIND_SIZE = 10; // 发送窗口大小为 10, GBN 中应满
    足  $W + 1 \leq N$  ( $W$ 
14. // 为发送窗口大小,  $N$  为序列号个
    数)
15. // 本例取序列号 0...19 共 20 个
16. // 如果将窗口大小设为 1, 则为停-等协议
17. const int SEQ_SIZE = 20; // 序列号的个数, 从 0~19 共计 20 个
```



```
18.// 由于发送数据第一个字节如果值为 0，则数据会发送失败
19.// 因此接收端序列号为 1~20，与发送端一一对应
20.BOOL ack[SEQ_SIZE]; // 收到 ack 情况，对应 0~19 的 ack
21.int curSeq;          // 当前数据包的 seq
22.int curAck;          // 当前等待确认的 ack
23.int totalSeq;        // 收到的包的总数
24.int totalPacket;     // 需要发送的包总数
25.
26.//*****
27.// Method: lossInLossRatio
28.// FullName: lossInLossRatio
29.// Access: public
30.// Returns: BOOL
31.// Qualifier: 根据丢失率随机生成一个数字，判断是否丢失,丢失则返回
    TRUE，否则返回
32.// FALSE Parameter: float lossRatio [0,1]
33.//*****
34.BOOL lossInLossRatio(float lossRatio) {
35.    // 将传入的丢失率转换为 0~100 的整数
36.    int lossBound = (int)(lossRatio * 100);
37.    int r = rand() % 101; // 生成 0~100 的随机数
38.    if (r <= lossBound) { // 如果随机数小于等于丢失率
39.        return TRUE;
40.    }
41.    return FALSE; // 否则不丢失
42.}
43.//*****
44.// Method: getCurTime
45.// FullName: getCurTime
46.// Access: public
47.// Returns: void
48.// Qualifier: 获取当前系统时间，结果存入 ptime 中
49.// Parameter: char * ptime
50.//*****
51.void getCurTime(char* ptime) {
52.    char buffer[128]; // 缓冲区
53.    memset(buffer, 0, sizeof(buffer)); // 初始化
54.    time_t c_time;    // 时间
55.    struct tm* p = NULL; // 时间结构体
56.    time(&c_time);      // 获取当前时间
57.    localtime_s(p, &c_time); // 转换为本地时间
58.    printf("\n%d\n\n", p->tm_year); // 输出年份
59.    // 格式化时间
```

```
60.    sprintf_s(buffer, "%d/%d/%d %d:%d:%d", p->tm_year + 1900, p->
    tm_mon,
61.        p->tm_mday, p->tm_hour, p->tm_min, p->tm_sec);
62.    // 将格式化后的时间拷贝到 ptime 中
63.    strcpy_s(ptime, sizeof(buffer), buffer);
64.}
65.//*****
66.// Method: seqIsAvailable
67.// FullName: seqIsAvailable
68.// Access: public
69.// Returns: bool
70.// Qualifier: 当前序列号 curSeq 是否可用
71.//*****
72.bool seqIsAvailable() {
73.    int step;                // 当前序列号与当前确认号的差值
74.    step = curSeq - curAck;  // 当前序列号与当前确认号的差值
75.    step = step >= 0 ? step : step + SEQ_SIZE; // 处理环形序列
    号
76.    // 序列号是否在当前发送窗口之内
77.    if (step >= SEND_WIND_SIZE) {
78.        return false;
79.    }
80.    // 如果收到 ack, 则表示当前序列号可用
81.    if (ack[curSeq]) {
82.        return true;
83.    }
84.    // 如果未收到 ack, 则表示当前序列号不可用
85.    return false;
86.}
87.//*****
88.// Method: timeoutHandler
89.// FullName: timeoutHandler
90.// Access: public
91.// Returns: void
92.// Qualifier: 超时重传处理函数, 滑动窗口内的数据帧都要重传
93.//*****
94.void timeoutHandler() {
95.    printf("Timer out error.\n"); // 超时重传
96.    int index;                   // 序列号
97.    for (int i = 0; i < SEND_WIND_SIZE; ++i) { // 遍历滑动窗口内的
        数据帧
98.        index = (i + curAck) % SEQ_SIZE; // 计算序列号
99.        ack[index] = TRUE;               // 标记为重传
100.    }
```

```
101.    totalSeq -= SEND_WIND_SIZE; // 更新已发送的总序列号数
102.    curSeq = curAck;           // 重置当前发送序列号
103.}
104.//*****
105.// Method: ackHandler
106.// FullName: ackHandler
107.// Access: public
108.// Returns: void
109.// Qualifier: 收到 ack, 累积确认, 取数据帧的第一个字节
110.// 由于发送数据时, 第一个字节(序列号)为
111.// 0 (ASCII) 时发送失败, 因此加一了, 此处需要减一还原
112.// Parameter: char c
113.//*****
114.void ackHandler(char c) {
115.    unsigned char index = (unsigned char)c - 1; // 序列号减一
116.    printf("Recv a ack of %d\n", index); // 输出收到的 ack 序列号
117.    // 当 curAck 小于等于 index 时, 表示 ack 没有回到 curAck 的左边
118.    if (curAck <= index) {
119.        // 依次标记 curAck 到 index 之间的 ack 为 TRUE
120.        for (int i = curAck; i <= index; ++i) {
121.            ack[i] = TRUE;
122.        }
123.        // 更新 curAck
124.        curAck = (index + 1) % SEQ_SIZE;
125.    }
126.    else {
127.        // ack 超过了最大值, 回到了 curAck 的左边
128.        for (int i = curAck; i < SEQ_SIZE; ++i) {
129.            ack[i] = TRUE; // 标记右边的 ack 为 TRUE
130.        }
131.        for (int i = 0; i <= index; ++i) {
132.            ack[i] = TRUE; // 标记右边的 ack 为 TRUE
133.        }
134.        curAck = index + 1; // 更新 curAck
135.    }
136.}
137.// 主函数
138.int main(int argc, char* argv[]) {
139.    // 加载套接字库(必须)
140.    WORD wVersionRequested;
141.    WSADATA wsaData;
142.    // 套接字加载时错误提示
143.    int err;
```

```
144.    // 版本 2.2
145.    wVersionRequested = MAKEWORD(2, 2);
146.    // 加载 dll 文件 Scket 库
147.    err = WSASStartup(wVersionRequested, &wsaData);
148.    if (err != 0) {
149.        // 找不到 winsock.dll
150.        printf("WSASStartup failed with error: %d\n", err);
151.        return -1;
152.    }
153.    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2) {
154.        printf("Could not find a usable version of Winsock.dll\n");
155.        WSACleanup();
156.    }
157.    else {
158.        printf("The Winsock 2.2 dll was found okay\n");
159.    }
160.    SOCKET sockServer = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
161.    // 设置套接字为非阻塞模式
162.    int iMode = 1; // 1: 非阻塞, 0: 阻塞
163.    ioctlsocket(sockServer, FIONBIO, (u_long FAR*) & iMode); // 非阻塞设置
164.    SOCKADDR_IN addrServer; // 服务器地址
165.    // addrServer.sin_addr.S_un.S_addr = inet_addr(SERVER_IP);
166.    // addrServer.sin_addr.S_un.S_addr = htonl(INADDR_ANY); // 两者均可
167.    // printf("\n%d\n\n", addrServer.sin_addr.S_un.S_addr);
168.    inet_pton(AF_INET, SERVER_IP, &addrServer.sin_addr); // 服务器 IP 地址
169.    addrServer.sin_family = AF_INET; // 协议族
170.    addrServer.sin_port = htons(SERVER_PORT); // 端口号
171.    // 绑定端口
172.    err = bind(sockServer, (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
173.    if (err) { // 绑定失败
174.        err = GetLastError(); // 获取错误代码
175.        // 输出错误信息
176.        printf("Could not bind the port %d for socket. Error code is %d\n",
```

```
177.         SERVER_PORT, err);
178.         WSACleanup(); // 卸载库
179.         return -1;     // 返回错误
180.     }
181.     SOCKADDR_IN addrClient;           // 客户端地址
182.     int length = sizeof(SOCKADDR);    // 地址长度
183.     char buffer[BUFFER_LENGTH];       // 数据发送接收缓冲区
184.     ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲区
185.     // int len = sizeof(SOCKADDR);
186.     // 将测试数据读入内存
187.     std::ifstream infile;             // 读取文件
188.     std::ofstream outfile;            // 写入文件
189.     infile.open("./test.txt");         // 打开文件
190.     char data[64 * 113];              // 读取数据
191.     ZeroMemory(data, sizeof(data));    // 初始化
192.     infile.read(data, 64 * 113);       // 读取数据
193.     infile.close();                   // 关闭文件
194.     // printf("\n%c\n\n", data[1]);
195.     totalPacket = sizeof(data) / 64;  // 计算包的总数
196.     int recvSize;                     // 接收到的数据大小
197.     // 初始化 ack 数组
198.     for (int i = 0; i < SEQ_SIZE; ++i) {
199.         ack[i] = TRUE;
200.     }
201.     int ret; // 返回值
202.     int interval = 1; // 收到数据包之后返回 ack 的间隔, 默认
        为 1 表示每个都返回
203.         // ack, 0 或者负数均表示所有的都不返回 ack
204.     char cmd[128];
205.     float packetLossRatio = 0.2; // 默认包丢失率 0.2
206.     float ackLossRatio = 0.2;    // 默认 ACK 丢失率 0.2
207.     //
208.     while (true) {
209.         // 非阻塞接收, 若没有收到数据, 返回值为-1
210.         recvSize = recvfrom(sockServer, buffer, BUFFER_LENGTH, 0
            ,
211.             ((SOCKADDR*)&addrClient), &length);
212.         // 解析接收到的数据
213.         ret = sscanf_s(buffer, "%s [%f] [%f]", &cmd, (unsigned)s
            izeof(cmd), &packetLossRatio, &ackLossRatio);
214.         // printf("\nhaved receviced!\n\n");
215.         if (recvSize < 0) { // 没有收到数据
216.             Sleep(200);    // 延时等待
217.             continue;
```

```
218.     }
219.     printf("recv from client: %s\n", buffer); // 输出接收到的数据
220.     if (strcmp(buffer, "-time") == 0) { // 如果接收到的数据是-time
221.         time_t timep; // 时间
222.         time(&timep); // 获取从 1970 至今过了多少秒，存入 time_t 类型的 timep
223.         ctime_s(buffer, BUFFER_LENGTH, &timep); // 将时间转换为字符串
224.         // printf("time is ready:%s\n", buffer);
225.     }
226.     else if (strcmp(buffer, "-quit") == 0) { // 如果接收到的数据是-quit
227.         // 退出程序
228.         strcpy_s(buffer, strlen("Good bye!") + 1, "Good bye!");
229.     }
230.     else if (strcmp(buffer, "-testgbn") == 0) {
231.         // 进入 gbn 测试阶段
232.         // 首先 server (server 处于 0 状态) 向 client 发送 205
233.         // 状态码 (server 进入 1 状态)
234.         // server 等待 client 回复 200 状态码，如果收到 (server 进入
235.         // 2 状态)，则开始传输文件，否则延时等待直至超时
236.         // 在文件传输阶段，server 发送窗口大小设为
237.         ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲区
238.         int recvSize; // 接收到的数据大小
239.         int waitCount = 0; // 等待计数器
240.         printf(
241.             "Begain to test GBN protocol,please don't abort the process\n");
242.         // 加入了一个握手阶段
243.         // 首先服务器向客户端发送一个 205
244.         // 大小的状态码 (我自己定义的) 表示服务器准备好了，可以发送数据
245.         // 客户端收到 205 之后回复一个 200
246.         // 大小的状态码，表示客户端准备好了，可以接收数据了 服务器收到 200
247.         // 状态码之后，就开始使用 GBN 发送数据了
248.         printf("Shake hands stage\n");
```

```
249.         int stage = 0;           // 阶段
250.         bool runFlag = true;      // 运行标志
251.         while (runFlag) {
252.             switch (stage) {
253.                 case 0: // 发送 205 阶段, 表示服务器准备好了
254.                     buffer[0] = 205; // 状态码 205
255.                     sendto(sockServer, buffer, strlen(buffer) +
256.                         1, 0,
257.                         (SOCKADDR*)&addrClient, sizeof(SOCKADDR)
258.                     );
259.                     Sleep(100); // 延时等待
260.                     stage = 1; // 进入下一个阶段
261.                     break;
262.                 case 1: // 等待接收 200
263.                     // 阶段, 没有收到则计数器+1, 超时则放弃此次
264.                     // “连接”, 等待从第一步开始
265.                     recvSize =
266.                     recvfrom(sockServer, buffer, BUFFER LENG
267.                         TH, 0,
268.                         ((SOCKADDR*)&addrClient), &length);
269.
270.                     if (recvSize < 0) { // 没有收到数据
271.                         ++waitCount; // 计数器+1
272.                         if (waitCount > 20) { // 如果等待次数超
273.                             过 20 次
274.                             runFlag = false; // 超时, 放弃此次连
275.                             接
276.                             printf("Timeout error\n"); // 输出错
277.                             误信息
278.                             break;
279.                         }
280.                         Sleep(500); // 等待 500ms 后重试
281.                         continue;
282.                     }
283.                     else {
284.                         // 当收到 200 状态码时, 表示客户端准备好
285.                         了
286.                         if ((unsigned char)buffer[0] == 200) {
287.                             // 输出信息
288.                             printf("Begin a file transfer\n");
289.                             // 输出文件大小信息
290.                             printf(
291.                                 "File size is %dB, each packet i
292.                                 s 64B and "
```

```
283.         "packet total num is % d\n",
284.         sizeof(data), totalPacket);
285.         curSeq = 0;      // 当前序列号
286.         curAck = 0;      // 当前确认号
287.         totalSeq = 0;    // 总序列号
288.         waitCount = 0;   // 等待计数器
289.         stage = 2;       // 进入下一个阶段
290.     }
291. }
292. break;
293. case 2: // 数据传输阶段
294.     // 检查是否有可用的序列号的数据
295.     if (seqIsAvailable()) {
296.         // 发送给客户端的序列号从 1 开始
297.         buffer[0] = curSeq + 1;
298.         // 标记该包未被确认
299.         ack[curSeq] = FALSE;
300.         // 数据发送的过程中应该判断是否传输完成
301.         // 为简化过程此处并未实现
302.         // 拷贝数据
303.         memcpy(&buffer[1], data + 64 * totalSeq,
304.             64);
305.         // printf("\n%s\n\n", &buffer[1]);
306.         // 如果数据为空, 则表示传输完成
307.         while (sendto(sockServer, buffer, BUFFER
308.             _LENGTH, 0,
309.             (SOCKADDR*)&addrClient,
310.             sizeof(SOCKADDR)) == -1);
311.         // 输出信息
312.         printf("send a packet with a seq of %d\n", curSeq);
313.         ++curSeq;      // 序列号+1
314.         curSeq %= SEQ_SIZE; // 环形序列号
315.         ++totalSeq;    // 总序列号+1
316.         Sleep(500);    // 延时等待
317.     }
318.     // 等待 Ack, 若没有收到, 则返回值为-1, 计数器
319.     +1
320.     // Sleep(500);
321.     recvSize =
322.     recvfrom(sockServer, buffer, BUFFER_LENGTH, 0,
323.         ((SOCKADDR*)&addrClient), &length);
```



```
321.          // printf("\n%d\n\n", recvSize);
322.          if (recvSize < 0) { // 没有收到数据
323.              waitCount++;    // 计数器+1
324.              // 20 次等待 ack 则超时重传
325.              if (waitCount > 20) { // 如果等待次数超
                过 20 次
326.                  timeoutHandler(); // 超时重传
327.                  buffer[0] = 300;  // 超时重传状态
                码
328.                  sendto(sockServer, buffer, strlen(bu
                ffer) + 1,
329.                          0, (SOCKADDR*)&addrClient,
330.                          sizeof(SOCKADDR));
331.                  waitCount = 0; // 重置计数器
332.              }
333.          }
334.          else { // 如果收到数据
335.              // 收到 ack
336.              ackHandler(buffer[0]); // 处理 ack
337.              waitCount = 0;        // 重置计数器
338.          }
339.          Sleep(500); // 延时等待
340.          break;
341.      }
342.  }
343.  }
344.  else if (strcmp(cmd, "-optestgbn") == 0) {
345.      // 进入 gbn 测试阶段
346.      printf(
347.          "%s\n",
348.          "Begin to test GBN protocol, please don't abort
            the process");
349.      // 读取丢包率和 ACK 丢包率
350.      printf(
351.          "The loss ratio of packet is %.2f,the loss ratio
            of ack is % "
352.          ".2f\n",
353.          packetLossRatio, ackLossRatio);
354.      int waitCount = 0;    // 等待计数器
355.      int stage = 0;       // 阶段
356.      int counter = 0;     // 计数器
357.      BOOL b;              // 丢包标志
358.      unsigned char u_code; // 状态码
359.      unsigned short seq;   // 包的序列号
```

```
360.         unsigned short recvSeq; // 接收窗口大小为 1, 已确认的
           序列号
361.         unsigned short waitSeq; // 等待的序列号
362.         // sendto(sockServer, "-testgbn", strlen("-
           testgbn") + 1, 0,
363.         // (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
364.         while (true) {
365.             // 等待 server 回复设置 UDP 为阻塞模式
366.             // Sleep(500);
367.             while (recvfrom(sockServer, buffer, BUFFER_LENGT
           H, 0,
368.             (SOCKADDR*)&addrClient, &length) == -1);
369.
370.             switch (stage) {
371.             case 0: // 等待握手阶段
372.                 u_code = (unsigned char)buffer[0]; // 状态
           码
373.                 if ((unsigned char)buffer[0] == 205) {
374.                     // 如果收到 205 状态码, 表示客户端准备好
           了
375.                     printf("Ready for file transmission\n");
376.                     buffer[0] = 200; // 回复 200 状态码
377.                     buffer[1] = '\0'; // 结束符
378.                     // sendto 函数用来发送数据, 参数分别为: 套接
           字, 发送缓冲区, 发送缓冲区大小, 标志位, 目的地址, 目的地址大小
379.                     sendto(sockServer, buffer, 2, 0,
380.                     (SOCKADDR*)&addrClient, sizeof(SOCKA
           DDR));
381.                     stage = 1; // 进入接收数据阶段
382.                     recvSeq = 0; // 接收窗口大小为 1
383.                     waitSeq = 1; // 等待的序列号
384.                 }
385.                 break;
386.             case 1: // 等待接收数据阶段
387.                 // 如果数据为空, 则表示传输完成
388.                 seq = (unsigned short)buffer[0];
389.                 // printf("\n%d\n\n", buffer[0]);
390.                 // 随机法模拟包是否丢失
391.                 b = lossInLossRatio(packetLossRatio);
392.                 if (b) { // 如果丢包
393.                     // 输出信息
394.                     printf("The packet with a seq of %d loss
           \n", seq);
```

```
395.         counter = 1;  // 计数器+1
396.
397.         buffer[0] = -1;  // 丢包状态码
398.         sendto(sockServer, buffer, 2, 0,
399.             (SOCKADDR*)&addrClient, sizeof(SOCKA
         DDR));
400.         // Sleep(2000);
401.         continue;
402.     }
403.     // 输出接收到的包的序列号
404.     printf("recv a packet with a seq of %d\n", s
        eq);
405.     // 如果是期待的包，正确接收，正常确认即可
406.     if (!(waitSeq - seq)) {  // 如果是期待的序列
        号
407.         ++waitSeq;          // 等待序列号+1
408.         if (waitSeq == 21) {  // 如果等待序列号超
            过 20
409.             waitSeq = 1;    // 重置为 1
410.         }
411.         // 输出数据
412.         // printf("%s\n",&buffer[1]);
413.         buffer[0] = seq;    // 序列号
414.         recvSeq = seq;      // 更新已接收序列号
415.         buffer[1] = '\0';   // 结束符
416.     }
417.     else {
418.         // 如果当前一个包都没有收到，则等
            待 Seq 为 1
419.         // 的数据包，不是则不返回
420.         // ACK（因为并没有上一个正确的 ACK）
421.         if (!recvSeq) {  // 如果尚未接收到任何包，
            只能等待序列号为
422.             // 1 的包
423.             continue;
424.         }
425.         // 否则（收到乱序的包），则返回上一次正确接
            收的序列号的 ack
426.         buffer[0] = recvSeq;
427.         buffer[1] = '\0';
428.     }
429.     // 随机法模拟 ack 是否丢失
430.     b = lossInLossRatio(ackLossRatio);
431.     if (b) {  // 如果 ack 丢失
```

```
432.                // 输出信息
433.                printf("The ack of %d loss\n",
434.                    (unsigned char)buffer[0]);
435.                counter = 1; // 计数器+1
436.
437.                buffer[0] = -1; // 丢包状态码
438.                sendto(sockServer, buffer, 2, 0,
439.                    (SOCKADDR*)&addrClient, sizeof(SOCKA
DDR));
440.                // Sleep(2000);
441.                continue;
442.            }
443.            // printf("\n%d\n\n",buffer[0]);
444.            while (sendto(sockServer, buffer, 2, 0,
445.                (SOCKADDR*)&addrClient,
446.                sizeof(SOCKADDR)) == -1);
447.            // 输出信息
448.            printf("send a ack of %d\n", (unsigned char)
buffer[0]);
449.            counter = 0; // 计数器清零
450.            break;
451.        }
452.        Sleep(500);
453.    }
454.    }
455.    else if (strcmp(cmd, "-download") == 0) {
456.        // 进入 gbn 测试阶段
457.        // 首先 server (server 处于 0 状态) 向 client 发
送 205
458.        // 状态码 (server 进入 1 状态)
459.        // server 等待 client 回复 200 状态码, 如果收到
(server 进入
460.        // 2 状态), 则开始传输文件, 否则延时等待直至超时
461.        // 在文件传输阶段, server 发送窗口大小设为
462.        ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲
区
463.        int recvSize; // 接收到的数据
大小
464.        int waitCount = 0; // 等待计数器
465.        printf(
466.            "Begain to test GBN protocol,please don't abort
the process\n");
467.        // 加入了一个握手阶段
468.        // 首先服务器向客户端发送一个 205
```

```
469.          // 大小的状态码（我自己定义的）表示服务器准备好了，可以发
           送数据
470.          // 客户端收到 205 之后回复一个 200
471.          // 大小的状态码，表示客户端准备好了，可以接收数据了 服
           务器收到 200
472.          // 状态码之后，就开始使用 GBN 发送数据了
473.          printf("Shake hands stage\n");
474.          int stage = 0;          // 阶段
475.          bool runFlag = true;    // 运行标志
476.          while (runFlag) {
477.              switch (stage) {
478.                  case 0: // 发送 205 阶段，表示客户端准备好了
479.                      buffer[0] = 205;
480.                      // 发送数据
481.                      sendto(sockServer, buffer, strlen(buffer) +
                           1, 0,
482.                          (SOCKADDR*)&addrClient, sizeof(SOCKADDR)
                           );
483.                      Sleep(100); // 延时等待
484.                      stage = 1;  // 进入等待接收数据阶段
485.                      break;
486.                  case 1: // 等待接收 200
487.                      // 阶段，没有收到则计数器+1，超时则放弃此次
                           “连接”，等待从第一步开始
488.                      recvSize =
489.                      recvfrom(sockServer, buffer, BUFFER LENG
                           TH, 0,
490.                          ((SOCKADDR*)&addrClient), &length);
491.                      if (recvSize < 0) {          // 如果没有收到
492.                          ++waitCount;            // 等待计数器+1
493.                          if (waitCount > 20) {    // 如果等待次数超
                           过 20 次
494.                              runFlag = false;    // 超时，放弃此次连
                           接
495.                              printf("Timeout error\n"); // 输出错
                           误信息
496.                              break;
497.                          }
498.                          Sleep(500); // 等待 500ms 后重试
499.                          continue;
500.                      }
501.                      else { // 如果收到数据
502.                          // 当收到 200 状态码时，表示客户端准备好了
```

```
503.         if ((unsigned char)buffer[0] == 200) {
504.             // 输出信息
505.             printf("Begin a file transfer\n");
506.             // 输出文件大小信息
507.             printf(
508.                 "File size is %dB, each packet i
s 64B and "
509.                 "packet total num is % d\n",
510.                 sizeof(data), totalPacket);
511.             curSeq = 0;    // 当前数据包的序列
号
512.             curAck = 0;    // 当前等待确认
的 ack
513.             totalSeq = 0;  // 收到的包的总数
514.             waitCount = 0; // 等待计数器
515.             stage = 2;     // 进入数据传输阶段
516.         }
517.     }
518.     break;
519.     case 2:                // 数据传输阶段
520.         if (seqIsAvailable()) { // 检查是否有可用的序
列号的数据
521.             // 如果有可用的序列号的数据，则准备数据包
522.             // 发送给客户端的序列号从 1 开始
523.             buffer[0] = curSeq + 1;
524.             // 标记该包未被确认
525.             ack[curSeq] = FALSE;
526.             // 数据发送的过程中应该判断是否传输完成
527.             // 为简化过程此处并未实现
528.             // 拷贝数据
529.             memcpy(&buffer[1], data + 64 * totalSeq,
64);
530.             // 输出数据
531.             printf("\nbuffer:%s\n\n", &buffer[1]);
532.             // 如果数据为空，则表示传输完成
533.             if (buffer[1] == '\0') {
534.                 // printf("Finish\n");
535.                 stage = 3; // 进入结束阶段
536.                 break;
537.             }
538.             // 发送数据包，直到成功为止，并打印发送的序
列号
539.             while (sendto(sockServer, buffer, BUFFER
_LENGTH, 0,
```

```
540.                (SOCKADDR*)&addrClient,
541.                sizeof(SOCKADDR)) == -1);
542.                printf("send a packet with a seq of %d\n", curSeq);
543.                ++curSeq;                // 更新当前序列号
544.                curSeq %= SEQ_SIZE;    // 环形序列号
545.                ++totalSeq;            // 更新总序列号
546.                Sleep(500);            // 延时等待
547.            }
548.            // 等待 Ack, 若没有收到, 则返回值为-1, 计数器
549.            +1
550.            // Sleep(500);
551.            recvSize =
552.            recvfrom(sockServer, buffer, BUFFER_LENGTH, 0,
553.                ((SOCKADDR*)&addrClient), &length);
554.            // printf("\n%d\n\n", recvSize);
555.            if (recvSize < 0) { // 如果没有收到数据
556.                waitCount++;    // 等待计数器+1
557.                // 20 次等待 ack 则超时重传
558.                if (waitCount > 20) { // 如果等待次数超过 20 次
559.                    timeoutHandler(); // 超时重传
560.                    waitCount = 0;    // 重置计数器
561.                }
562.            }
563.            else { // 如果收到数据
564.                // 收到 ack
565.                ackHandler(buffer[0]); // 调用
566.                ackhandler 处理 ack
567.                waitCount = 0;        // 重置等待计数器
568.            }
569.            Sleep(500); // 延时等待
570.            break;
571.            case 3: // 结束阶段
572.                printf("Finish\n"); // 输出信息
573.                runFlag = false;    // 结束循环
574.            }
575.        }
576.    }
577.    else if (strcmp(buffer, "-upload") == 0) {
578.        ZeroMemory(data, sizeof(data)); // 初始化数据
```



```
577.          // 进入 upload 阶段
578.          printf(
579.              "%s\n",
580.              "Begin to test GBN protocol, please don't abort
the process");
581.          int waitCount = 0;      // 等待计数器
582.          int stage = 0;          // 阶段
583.          int counter = 0;        // 计数器
584.          BOOL flag = true;       // 运行标志
585.          BOOL b;                 // 丢包标志
586.          unsigned char u_code;    // 状态码
587.          unsigned short seq;      // 包的序列号
588.          unsigned short recvSeq;  // 接收窗口大小为 1, 已确认的
序列号
589.          unsigned short waitSeq;  // 等待的序列号
590.          // sendto(sockServer, "-testgbn", strlen("-
testgbn") + 1, 0,
591.          // (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
592.          while (flag) {
593.              // 等待 server 回复设置 UDP 为阻塞模式
594.              // Sleep(500);
595.
596.              if (stage == 2) {
597.                  // 存储文件
598.                  printf("Finish\n");          // 输出
信息
599.                  printf("\ndata:\n%s\n\n", &data[0]); // 输出
数据
600.                  FILE* out;
601.                  // 如果文件打开成功, 则写入数据
602.                  if (fopen_s(&out, "test_recver.txt", "wb") =
= 0) {
603.                      // 写入数据
604.                      fwrite(data, sizeof(char), strlen(data),
out);
605.                      // 关闭文件
606.                      fclose(out);
607.                  }
608.                  flag = false; // 结束循环
609.                  break;
610.              }
611.              // 循环等待数据, 通过 recvfrom 函数阻塞等待客户端发送
数据
612.              // 直到接收到数据为止, 循环才会退出
```

```
613.         while (recvfrom(sockServer, buffer, BUFFER_LENTH
        H, 0,
614.             (SOCKADDR*)&addrClient, &length) == -1);
615.         // 拷贝数据
616.         memcpy(data + 64 * totalSeq, &buffer[1], 64);
617.         // 输出数据
618.         printf("\ndata:%s\n\n", data + 64 * totalSeq);
619.         switch (stage) {
620.         case 0: // 等待握手阶段
621.             // 获取状态码
622.             u_code = (unsigned char)buffer[0];
623.             if ((unsigned char)buffer[0] == 205) {
624.                 printf("Ready for file transmission\n");
625.                 // 服务器发送 200 以确认准备好接收数据
626.                 buffer[0] = 200;
627.                 buffer[1] = '\0'; // 结束符
628.                 sendto(sockServer, buffer, 2, 0,
629.                     (SOCKADDR*)&addrClient, sizeof(SOCKA
                        DDR));
630.                 stage = 1; // 进入接收数据阶段
631.                 recvSeq = 0; // 接收窗口大小为 1
632.                 waitSeq = 1; // 等待的序列号
633.             }
634.             break;
635.             // 每次收到一个数据包后，客户端检查其序列号是否
            与预期一致
636.             // 若包的序列号正确，发送对应的
637.             // ACK。若不符合预期，客户端根据上一个确认的序列
            号发送
638.             // ACK
639.         case 1: // 等待接收数据阶
            段
640.             if (buffer[1] == '\0') { // 如果收到的数据为
                空
641.                 stage = 2; // 进入结束阶段
642.                 break;
643.             }
644.             // 获取包的序列号
645.             seq = (unsigned short)buffer[0];
646.             // 输出接收到的包的序列号
647.             printf("recv a packet with a seq of %d\n", s
                eq);
648.             ++totalSeq; // 更新已接收的包的总数
```

```
649.          // 如果是期待的包，正确接收，正常确认即可
650.          if (!(waitSeq - seq)) { // 如果是期待的序列
        号
651.              ++waitSeq; // 更新期待的下一个序列号
652.              if (waitSeq == 21) { // 如果等待序列号超
        过 20
653.                  waitSeq = 1; // 重置为 1
654.              }
655.              // 输出数据
656.              // printf("%s\n",&buffer[1]);
657.              buffer[0] = seq; // 序列号
658.              recvSeq = seq; // 更新已接收的序列号
659.              buffer[1] = '\0'; // 结束符
660.          }
661.          else {
662.              // 如果当前一个包都没有收到，则等
        待 Seq 为 1
663.              // 的数据包，不是则不返回
664.              // ACK（因为并没有上一个正确的 ACK）
665.              // 如果尚未接收到任何包，只能等待序列号为 1
        的包
666.              if (!recvSeq) {
667.                  continue;
668.              }
669.              // 否则（收到乱序的包），则返回上一次正确接
        收的序列号的 ack
670.              buffer[0] = recvSeq;
671.              buffer[1] = '\0'; // 结束符
672.          }
673.          // 发送 ack
674.          while (sendto(sockServer, buffer, 2, 0,
675.                        (SOCKADDR*)&addrClient,
676.                        sizeof(SOCKADDR)) == -1);
677.          // 输出信息
678.          printf("send a ack of %d\n", (unsigned char)
        buffer[0]);
679.          counter = 0; // 计数器清零
680.          break;
681.      }
682.      Sleep(500); // 延时等待
683.  }
684.  }
685.  // 发送数据
686.  sendto(sockServer, buffer, strlen(buffer) + 1, 0,
```

```
687.         (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
688.         Sleep(500); // 延时等待
689.     }
690.     // 关闭套接字, 卸载库
691.     closesocket(sockServer);
692.     WSACleanup();
693.     return 0;
694. }

1. // SR_Client.cpp
2. #include <WS2tcpip.h>
3. #include <WinSock2.h>
4. #include <stdlib.h>
5. #include <time.h>
6.
7. #include <fstream>
8. #pragma comment(lib, "ws2_32.lib")
9. #define SERVER_PORT 12340 // 接收数据的端口号
10. #define SERVER_IP "127.0.0.1" // 服务器的 IP 地址
11. #define BUFFER_SIZE 1024 // 缓冲区大小
12. #define SEQ_SIZE 16 // 序列号个数
13. #define SWIN_SIZE 8 // 发送窗口大小
14. #define RWIN_SIZE 8 // 接收窗口大小
15. #define LOSS_RATE 0.8 // 丢包率
16. using namespace std;
17.
18. char cmdBuffer[50]; // 命令缓冲区
19. char buffer[BUFFER_SIZE]; // 缓冲区
20. char cmd[10]; // 命令
21. char fileName[40]; // 文件名
22. char filePath[50]; // 文件路径
23. char file[1024 * 1024]; // 文件字符串
24. int len = sizeof(SOCKADDR); // 地址长度
25. int recvSize; // 接收到的数据大小
26. int Deliver(char* file, int ack);
27. int Send(ifstream& infile, int seq, SOCKET socket, SOCKADDR* addr
);
28. int MoveSendWindow(int seq);
29. int Read(ifstream& infile, char* buffer);
30. // 缓存结构体
31. struct Cache {
32.     bool used; // 是否使用
33.     char buffer[BUFFER_SIZE]; // 缓冲区
34.     Cache() { // 构造函数
```

```
35.         used = false; // 初始化为未使用
36.         ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲区
37.     }
38.} recvWindow[SEQ_SIZE]; // 接收窗口
39.// 数据帧结构体
40.struct DataFrame {
41.    clock_t start; // 发送时间
42.    char buffer[BUFFER_SIZE]; // 缓冲区
43.    DataFrame() { // 构造函数
44.        start = 0; // 初始化发送时间
45.        ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲区
46.    }
47.} sendWindow[SEQ_SIZE]; // 发送窗口
48.int main(int argc, char* argv[]) {
49.    // 加载套接字库
50.    WORD wVersionRequested;
51.    WSADATA wsaData;
52.    // 版本 2.2
53.    wVersionRequested = MAKEWORD(2, 2);
54.    int err = WSStartup(wVersionRequested, &wsaData); // 加
    载 Winsock.dll
55.    if (err != 0) { // 加载失
    败
56.        // 输出错误码
57.        printf("Winsock.dll 加载失败, 错误码: %d\n", err);
58.        return -1;
59.    }
60.    // 判断 Winsock.dll 的版本
61.    if (LOBYTE(wsaData.wVersion) != LOBYTE(wVersionRequested) ||
62.        HIBYTE(wsaData.wVersion) != HIBYTE(wVersionRequested)) {
63.        // 如果版本不符合要求, 输出错误信息
64.        printf("找不到 %d.%d 版本
        的 Winsock.dll\n", LOBYTE(wVersionRequested),
65.            HIBYTE(wVersionRequested));
66.        WSACleanup(); // 清理 Winsock.dll
67.        return -1;
68.    }
69.    else { // 加载成功
70.        printf("Winsock %d.%d 加载成功
        \n", LOBYTE(wVersionRequested),
71.            HIBYTE(wVersionRequested));
72.    }
```

```
73.    // 创建客户端套接字
74.    SOCKET socketClient = socket(AF_INET, SOCK_DGRAM, 0);
75.    // 设置为非阻塞模式
76.    int iMode = 1; //
    非阻塞模式
77.    ioctlsocket(socketClient, FIONBIO, (u_long FAR*) & iMode); /
    / 设置非阻塞
78.    SOCKADDR_IN addrServer; //
    服务器地址
79.    inet_pton(AF_INET, SERVER_IP, &addrServer.sin_addr); // 设置
    服务器 IP
80.    addrServer.sin_family = AF_INET; // 设置
    地址族
81.    addrServer.sin_port = htons(SERVER_PORT); // 设置
    端口号
82.    srand((unsigned)time(NULL)); // 设置随机数种子
83.    int status = 0; // 状态
84.    clock_t start; // 开始时间
85.    clock_t now; // 当前时间
86.    int seq; // 序列号
87.    int ack; // 确认号
88.    while (true) {
89.        gets_s(cmdBuffer, 50); // 从控制台读取命令
90.        // 解析命令
91.        sscanf_s(cmdBuffer, "%s", cmd, sizeof(cmd) - 1, fileName,
            e,
92.            sizeof(fileName) - 1);
93.        // 如果命令式 -upload
94.        if (!strcmp(cmd, "upload")) {
95.            // 打印上传文件
96.            printf("申请上传文件: %s\n", fileName);
97.            strcpy_s(filePath, "."); // 设置文件路径
98.            strcat_s(filePath, fileName); // 连接文件名
99.            ifstream infile(filePath); // 打开文件
100.            start = clock(); // 获取当前时间
101.            seq = 0; // 序列号
102.            status = 0; // 状态
103.            sendWindow[0].buffer[0] = 10; // 设置缓冲区
104.            // 设置缓冲区
105.            strcpy_s(sendWindow[0].buffer + 1, strlen(cmdBuffer)
                + 1,
106.                cmdBuffer);
107.            sendWindow[0].start = start - 1000L; // 设置发送时
            间
```

```
108.         while (true) {
109.             // 接收数据
110.             recvSize = recvfrom(socketClient, buffer, BUFFER
                _SIZE, 0,
111.                 (SOCKADDR*)&addrServer, &len);
112.             switch (status) { // 根据状态处理数据
113.                 case 0: // 请求上传
114.                     // 如果接收到数据，并且数据类型为 100
115.                     if (recvSize > 0 && buffer[0] == 100) {
116.                         // 如果数据内容为 OK
117.                         if (!strcmp(buffer + 1, "OK")) {
118.                             // 打印申请通过
119.                             printf("申请通过，开始上传...\n");
120.                             start = clock(); // 获取当前时间
121.                             status = 1; // 设置状态为 1
122.                             sendWindow[0].start = 0L; // 设置发送时间
123.                             continue; // 跳过当前循环
124.                         }
125.                         // 如果数据内容为 NO
126.                         else if (!strcmp(buffer + 1, "NO")) {
127.                             status = -1; // 设置状态为 -1
128.                             break; // 跳出循环
129.                         }
130.                     }
131.                     now = clock(); // 获取当前时间
132.                     // 如果当前时间减去发送时间大于 1000 毫秒
133.                     if (now - sendWindow[0].start >= 1000L) {
134.                         sendWindow[0].start = now; // 更新发送时间
135.                         // 发送数据
136.                         sendto(socketClient, sendWindow[0].buffer,
                            r,
137.                             strlen(sendWindow[0].buffer) + 1, 0,
138.                             (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
139.                     }
140.                     break;
141.                 case 1: // 发送文件数据
142.                     // 如果接收到数据，并且数据类型为 101
```



```
143.         if (recvSize > 0 && buffer[0] == 101) {
144.             start = clock();           // 获取当前
            时间
145.             ack = buffer[1];           // 获取确认
            号
146.             ack--;                     // 确认号减
            一
147.             sendWindow[ack].start = -1L; // 设置发送
            时间
148.             if (ack == seq) { // 如果确认号等于序列
            号
149.                 seq = MoveSendWindow(seq); // 交付数
            据
150.             }
151.             // 打印接收数据
152.             printf("接收 ack = %d, 当前起
            始 seq = %d\n",
153.                 ack + 1, seq + 1);
154.         }
155.         // 如果文件已发送完毕
156.         if (!Send(infile, seq, socketClient,
157.             (SOCKADDR*)&addrServer)) {
158.             // 打印上传完毕
159.             printf("上传完毕...\n");
160.             status = 2;           // 设置状
            态为 2
161.             start = clock();       // 获取当
            前时间
162.             sendWindow[0].buffer[0] = 10; // 设置缓
            冲区
163.             // 设置缓冲区
164.             strcpy_s(sendWindow[0].buffer + 1, 7, "F
            inish");
165.             // 设置发送时间
166.             sendWindow[0].start = start - 1000L;
167.             continue; // 跳过当前循环
168.         }
169.         break;
170.     case 2: // 等待完成确认
171.         // 如果接收到数据，并且数据类型为 100
172.         if (recvSize > 0 && buffer[0] == 100) {
173.             // 如果数据内容为 OK
174.             if (!strcmp(buffer + 1, "OK")) {
175.                 buffer[0] = 10; // 设置缓冲区
```

```
176.                strcpy_s(buffer + 1, 3, "OK"); //
    设置缓冲区
177.                // 发送数据
178.                sendto(socketClient, buffer, strlen(
    buffer) + 1,
179.                        0, (SOCKADDR*)&addrServer,
180.                        sizeof(SOCKADDR));
181.                status = 3; // 设置状态为 3
182.                break;      // 跳出循环
183.            }
184.        }
185.        now = clock(); // 获取当前时间
186.        // 如果当前时间减去发送时间大于 1000 毫秒
187.        if (now - sendWindow[0].start >= 1000L) {
188.            sendWindow[0].start = now; // 更新发送时
    间
189.            // 发送数据
190.            sendto(socketClient, sendWindow[0].buffer,
    r,
191.                    strlen(sendWindow[0].buffer) + 1, 0,
192.                    (SOCKADDR*)&addrServer, sizeof(SOCKA
    DDR));
193.        }
194.        default:
195.            break;
196.        }
197.        // 如果状态为 -1
198.        if (status == -1) {
199.            printf("服务器拒绝请求\n");
200.            infile.close();
201.            break;
202.        }
203.        // 如果状态为 3
204.        if (status == 3) {
205.            printf("上传成功, 结束通信\n");
206.            infile.close();
207.            break;
208.        }
209.        // 如果当前时间减去开始时间大于 5000 毫秒
210.        if (clock() - start >= 5000L) {
211.            printf("通信超时, 结束通信\n");
212.            infile.close();
213.            break;
```

```
214.         }
215.         // 如果接收到的数据大小小于等于 0
216.         if (recvSize <= 0) {
217.             Sleep(200);
218.         }
219.     }
220. }
221. // 如果命令式 -download
222. else if (!strcmp(cmd, "download")) {
223.     // 打印下载文件
224.     printf("申请下载文件 %s\n", fileName);
225.     strcpy_s(filePath, "."); // 设置文件路径
226.     strcat_s(filePath, fileName); // 连接文件名
227.     ofstream outfile(filePath); // 打开文件
228.     start = clock(); // 获取当前时间
229.     ack = 0; // 确认号
230.     status = 0; // 状态
231.     sendWindow[0].buffer[0] = 10; // 设置缓冲区
232.     // 设置缓冲区
233.     strcpy_s(sendWindow[0].buffer + 1, strlen(cmdBuffer)
+ 1,
234.             cmdBuffer);
235.     sendWindow[0].start = start - 1000L; // 设置发送时
    间
236.     while (true) { // 循环处理数
        据
237.         recvSize = recvfrom(socketClient, buffer, BUFFER
_SIZE, 0,
238.                             (SOCKADDR*)&addrServer, &len);
239.         // 模拟丢包,若随机数大于丢包率则丢包
240.         if ((float)rand() / RAND_MAX > LOSS_RATE) {
241.             recvSize = 0; // 设置接收数据大小为 0
242.             buffer[0] = 0; // 设置缓冲区
243.         }
244.         switch (status) {
245.         case 0: // 请求下载
246.             // 如果接收到数据,并且数据类型为 100
247.             if (recvSize > 0 && buffer[0] == 100) {
248.                 // 如果数据内容为 OK
249.                 if (!strcmp(buffer + 1, "OK")) {
250.                     // 打印申请通过
251.                     printf("申请通过,准备下载...\n");
252.                     start = clock(); // 获取当前时间
253.                     status = 1; // 设置状态为 1
```

```
254.                sendWindow[0].buffer[0] = 10; // 设置缓冲区
255.                // 设置缓冲区
256.                strcpy_s(sendWindow[0].buffer + 1, 3, "OK");
257.                // 设置发送时间
258.                sendWindow[0].start = start - 1000L;

259.                continue; // 跳过当前循环
260.            }
261.            // 如果数据内容为 NO
262.            else if (!strcmp(buffer + 1, "NO")) {
263.                status = -1; // 设置状态为 -1
264.                break;      // 跳出循环
265.            }
266.        }
267.        now = clock(); // 获取当前时间
268.        // 如果当前时间减去发送时间大于 1000 毫秒
269.        if (now - sendWindow[0].start >= 1000L) {
270.            sendWindow[0].start = now; // 更新发送时间
271.            // 发送数据
272.            sendto(socketClient, sendWindow[0].buffer,
273.                strlen(sendWindow[0].buffer) + 1, 0,
274.                (SOCKADDR*)&addrServer, sizeof(SOCKADDR));
275.        }
276.        break;
277.    case 1: // 开始下载数据
278.        // 如果接收到数据，并且数据类型为 200
279.        if (recvSize > 0 && (unsigned char)buffer[0] == 200) {
280.            // 打印开始下载
281.            printf("开始下载...\n");
282.            start = clock(); // 获取当前时间
283.            seq = buffer[1]; // 获取序列号
284.            // 输出接收数据
285.            printf(
286.                "接收数据帧 seq = %d, data = %s, 发送\n",
287.                "%d\n",
288.                seq, buffer + 2, seq);
```

```
289.          seq--; // 序列号减
    一
290.          recvWindow[seq].used = true; // 设置为已
    使用
291.          // 设置缓冲区
292.          strcpy_s(recvWindow[seq].buffer,
293.                  strlen(buffer + 2) + 1, buffer + 2);
294.          if (ack == seq) { // 如果确认号等于序列
    号
295.              ack = Deliver(file, ack); // 交付数
    据
296.              status = 2; // 设置状
    态为 2
297.              buffer[0] = 11; // 设置缓
    冲区
298.              buffer[1] = seq + 1; // 设置缓
    冲区
299.              buffer[2] = 0; // 设置缓
    冲区
300.          }
301.          sendto(socketClient, buffer, strlen(buff
    er) + 1, 0,
302.                  (SOCKADDR*)&addrServer, sizeof(SOCKA
    DDR));
303.          continue; // 跳过当前循环
304.      }
305.      now = clock(); // 获取当前时间
306.      // 如果当前时间减去开始时间大于 5000 毫秒
307.      if (now - sendWindow[0].start >= 1000L) {
308.          sendWindow[0].start = now; // 更新发送时
    间
309.          // 发送数据
310.          sendto(socketClient, sendWindow[0].buffe
    r,
311.                  strlen(sendWindow[0].buffer) + 1, 0,
312.                  (SOCKADDR*)&addrServer, sizeof(SOCKA
    DDR));
313.      }
314.      break;
315.      case 2: // 处理数据帧
316.          if (recvSize > 0) { // 如果接收到数据
317.              // 如果数据类型为 200
```

```
318.         if ((unsigned char)buffer[0] == 200) {
319.             seq = buffer[1];           // 获取序
           列号
320.             int temp = seq - 1 - ack; // 计算序
           列号差
321.             if (temp < 0) { // 如果序列号差小
           于 0
322.                 temp += SEQ_SIZE; // 序列号差加
           上序列号个数
323.             }
324.             start = clock(); // 获取当前时间
325.             seq--;           // 序列号减一
326.             // 如果序列号差小于接收窗口大小
327.             if (temp < RWIN_SIZE) {
328.                 // 如果接收窗口中的数据包未使用
329.                 if (!recvWindow[seq].used) {
330.                     // 设置为已使用
331.                     recvWindow[seq].used = true;

332.                     // 设置缓冲区
333.                     strcpy_s(recvWindow[seq].buf
           fer,
334.                             strlen(buffer + 2) + 1,

335.                             buffer + 2);
336.                 }
337.                 // 如果确认号等于序列号
338.                 if (ack == seq) {
339.                     ack = Deliver(file, ack); /
           / 交付数据
340.                 }
341.             }
342.             // 输出接收数据
343.             printf(
344.                 "接收数据
           帧 seq = %d, data = %s, 发送 ack "
345.                 "= %d, 起始 ack = %d\n",
346.                 seq + 1, buffer + 2, seq + 1, ac
           k + 1);
347.             buffer[0] = 11;           // 设置缓冲区
348.             buffer[1] = seq + 1;      // 设置缓冲区
349.             buffer[2] = 0;           // 设置缓冲区
350.             // 发送数据
```

```
351.                sendto(socketClient, buffer, strlen(
    buffer) + 1,
352.                0, (SOCKADDR*)&addrServer,
353.                sizeof(SOCKADDR));
354.            }
355.            // 如果数据类型为 100
356.            else if (buffer[0] == 100 &&
357.                !strcmp(buffer + 1, "Finish")) {
358.                status = 3; // 设置状态为 3
359.                outfile.write(file, strlen(file));
    // 写入文件
360.                buffer[0] = 10; // 设置缓冲区
361.                strcpy_s(buffer + 1, 3, "OK"); //
    设置缓冲区
362.                // 发送数据
363.                sendto(socketClient, buffer, strlen(
    buffer) + 1,
364.                0, (SOCKADDR*)&addrServer,
365.                sizeof(SOCKADDR));
366.                continue; // 跳过当前循环
367.            }
368.        }
369.        break; // 跳出循环
370.    default:
371.        break;
372.    }
373.    // 如果状态为 -1
374.    if (status == -1) {
375.        printf("服务器拒绝请求\n");
376.        outfile.close();
377.        break;
378.    }
379.    // 如果状态为 3
380.    if (status == 3) {
381.        printf("下载成功, 结束通信\n");
382.        outfile.close();
383.        break;
384.    }
385.    // 如果当前时间减去开始时间大于 5000 毫秒
386.    if (clock() - start >= 5000L) {
387.        printf("通信超时, 结束通信\n");
388.        outfile.close();
389.        break;
390.    }
```



```
391.          // 如果接收到的数据大小小于等于 0
392.          if (recvSize <= 0) {
393.              Sleep(20);
394.          }
395.      }
396.  }
397.      else if (!strcmp(cmd, "quit")) { // 如果命令式 -quit
398.          break;                      // 退出循环
399.      }
400.  }
401.  closesocket(socketClient); // 关闭套接字
402.  printf("关闭套接字\n");    // 打印关闭套接字
403.  WSACleanup();              // 清理 Winsock.dll
404.  return 0;
405.}
406.
407.// 从一个输入文件流中读取数据，并将读取的内容存储到一个字符数组（缓冲区）中
408.int Read(ifstream& infile, char* buffer) {
409.    if (infile.eof()) { // 判断是否到达文件末尾
410.        return 0;
411.    }
412.    infile.read(buffer, 3); // 从文件流中读取数据
413.    int cnt = infile.gcount(); // 获取读取的字节数
414.    buffer[cnt] = 0;         // 添加字符串结束符
415.    return cnt;              // 返回读取的字节数
416.}
417.// 处理已确认的数据包，并将其内容追加到指定的文件字符串中
418.int Deliver(char* file, int ack) {
419.    while (recvWindow[ack].used) { // 当接收窗口中的数据包已经被确认时
420.        recvWindow[ack].used = false; // 将该数据包标记为未使用
421.        // 将数据包的内容追加到文件字符串中
422.        strcat_s(file, strlen(file) + strlen(recvWindow[ack].buffer) + 1,
423.            recvWindow[ack].buffer);
424.        ack++; // 更新确认号
425.        ack %= SEQ_SIZE; // 确认号取模
426.    }
427.    return ack; // 返回更新后的确认号
428.}
429.// 从输入文件流中读取数据，并通过网络发送数据帧
430.int Send(ifstream& infile, int seq, SOCKET socket, SOCKADDR* addr) {
```

```
431.     clock_t now = clock(); // 获取当前时间
432.     // 循环遍历发送窗口大小，计算当前发送窗口的索引，时期在序列号范围
        内循环
433.     for (int i = 0; i < SWIN_SIZE; i++) {
434.         int j = (seq + i) % SEQ_SIZE; // 计算当前发送窗口的索引
        引
435.         if (sendWindow[j].start == -1L) { // 如果当前数据帧已经发
        送
436.             continue; // 跳过当前数据帧
437.         }
438.         if (sendWindow[j].start == 0L) { // 如果当前数据帧未发
        送
439.             if (Read(infile, sendWindow[j].buffer + 2)) { // 从
        文件流中读取数据
440.                 sendWindow[j].start = now; // 更新当前数据帧的发
        送时间
441.                 sendWindow[j].buffer[0] = 20; // 设置数据帧的
        类型
442.                 sendWindow[j].buffer[1] = j + 1; // 设置数据帧的
        序列号
443.             }
444.             else if (i == 0) { // 如果当前数据帧是发送窗口的第一个
        数据帧
445.                 return 0; // 返回 0，表示发送完毕
446.             }
447.             else { // 如果当前数据帧不是发送窗口的第一个数据帧
448.                 break; // 跳出循环
449.             }
450.         } // 如果当前数据帧发送超时
451.         else if (now - sendWindow[j].start >= 1000L) {
452.             sendWindow[j].start = now; // 更新当前数据帧的发送时
        间
453.         }
454.         else { // 如果当前数据帧未发送超
        时
455.             continue; // 跳过当前数据帧
456.         }
457.         // 发送数据帧
458.         printf("发送数据帧 seq = %d, data = %s\n", j + 1,
459.             sendWindow[j].buffer + 2);
460.         // 发送数据帧到指定的地址
461.         sendto(socket, sendWindow[j].buffer, strlen(sendWindow[j]
        ].buffer) + 1,
462.             0, addr, sizeof(SOCKADDR));
```

```
463.     }
464.     return 1;
465. }
466. // 移动发送窗口，找到下一个可以发送的序列号
467. int MoveSendWindow(int seq) {
468.     // 循环检查发送窗口中当前序列号 seq 对应的数据包状态
469.     while (sendWindow[seq].start == -1L) { // 当前序列号的数据
        帧已经发送
470.         sendWindow[seq].start = 0L; // 更新当前序列号的数据帧的发
        送时间
471.         seq++; // 更新序列号
472.         seq %= SEQ_SIZE; // 序列号取模
473.     }
474.     return seq; // 返回更新后的序列号
475. }

1. // SR_Server.cpp
2. #include <WS2tcpip.h>
3. #include <WinSock2.h>
4. #include <stdlib.h>
5. #include <time.h>
6.
7. #include <fstream>
8. #pragma comment(lib, "ws2_32.lib")
9. #define SERVER_PORT 12340 // 端口号
10. #define SERVER_IP "0.0.0.0" // IP 地址
11. #define SEQ_SIZE 16 // 序列号个数
12. #define SWIN_SIZE 8 // 发送窗口大小
13. #define RWIN_SIZE 8 // 接收窗口大小
14. #define BUFFER_SIZE 1024 // 缓冲区大小
15. #define LOSS_RATE 0.8 // 丢包率
16. using namespace std;
17. // 定义结构体，用于存储接收窗口和发送窗口的数据
18. struct recv {
19.     bool used; // 标记数据包是否已经被使用
20.     char buffer[BUFFER_SIZE]; // 缓冲区
21.     recv() { // 构造函数
22.         used = false; // 初始化数据包未被使用
23.         ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲区
24.     }
25. } recvWindow[SEQ_SIZE]; // 接收窗口
26. // 定义结构体，用于存储发送窗口的数据
27. struct send {
```

```
28.    clock_t start; // 由于使用的是 SR, 因此每一个窗口位置都需要设置一个计时器
29.    char buffer[BUFFER_SIZE]; // 缓冲区
30.    send() { // 构造函数
31.        start = 0; // 初始化计时器
32.        ZeroMemory(buffer, sizeof(buffer)); // 初始化缓冲区
33.    }
34.} sendWindow[SEQ_SIZE]; // 发送窗口
35.char cmdBuffer[50]; // 命令缓冲区
36.char buffer[BUFFER_SIZE]; // 缓冲区
37.char cmd[10]; // 命令
38.char fileName[40]; // 文件名
39.char filePath[50]; // 文件路径
40.char file[1024 * 1024]; // 文件字符串
41.int len = sizeof(SOCKADDR); // 地址长度
42.int recvSize; // 接收到的数据大小
43.int Deliver(char* file, int ack);
44.int Send(ifstream& infile, int seq, SOCKET socket, SOCKADDR* addr);
45.int MoveSendWindow(int seq);
46.int Read(ifstream& infile, char* buffer);
47.// 主函数
48.int main(int argc, char* argv[]) {
49.    // 加载套接字库
50.    WORD wVersionRequested;
51.    WSADATA wsaData;
52.    // 版本 2.2
53.    wVersionRequested = MAKEWORD(2, 2); // 请求 2.2 版本的 Winsock 库
54.    int err = WSAStartup(wVersionRequested, &wsaData); // 加载 Winsock 库
55.    if (err != 0) { // 如果加载 Winsock 库失败
56.        printf("Winsock.dll 加载失败, 错误码: %d\n", err);
57.        return -1;
58.    }
59.    // 判断请求的版本号是否正确
60.    if (LOBYTE(wsaData.wVersion) != LOBYTE(wVersionRequested) ||
61.        HIBYTE(wsaData.wVersion) != HIBYTE(wVersionRequested)) {
62.        // 如果请求的版本号不正确
63.        printf("找不到 %.%.d 版本的 Winsock.dll\n", LOBYTE(wVersionRequested),
64.            HIBYTE(wVersionRequested));
```

```
65.     WSACleanup(); // 卸载 Winsock 库
66.     return -1;
67. }
68. else { // 如果请求的版本号正确
69.     printf("Winsock %d.%d 加载成功\n", LOBYTE(wVersionRequested),
70.         HIBYTE(wVersionRequested));
71. }
72. // 创建服务器套接字
73. SOCKET socketServer = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
74. // 设置为非阻塞模式
75. int iMode = 1; // 1 为非阻塞, 0 为阻塞
76. // 设置为非阻塞模式
77. ioctlsocket(socketServer, FIONBIO, (u_long FAR*) & iMode);
78. SOCKADDR_IN addrServer; // 服务器地址
79. inet_pton(AF_INET, SERVER_IP, &addrServer.sin_addr); // 设置服务器 IP 地址
80. addrServer.sin_family = AF_INET; // 设置地址族
81. addrServer.sin_port = htons(SERVER_PORT); // 设置端口号
82. // 绑定端口
83. if (err = bind(socketServer, (SOCKADDR*)&addrServer, sizeof(SOCKADDR))) {
84.     err = GetLastError(); // 获取错误码
85.     printf("绑定端口 %d 失败, 错误码: %d\n", SERVER_PORT, err);
86.     WSACleanup(); // 卸载 Winsock 库
87.     return -1;
88. }
89. else { // 绑定端口成功
90.     printf("绑定端口 %d 成功", SERVER_PORT); // 输出提示信息
91. }
92. SOCKADDR_IN addrClient; // 客户端地址
93. int status = 0; // 状态
94. clock_t start; // 开始时间
95. clock_t now; // 当前时间
96. int seq; // 序列号
97. int ack; // 确认号
98. ofstream outfile; // 输出文件流
99. ifstream infile; // 输入文件流
```

```
100.    // 进入接收状态, 注意服务器主要处理的任务是接收客户机请求, 共有上
      载和下载两种任务
101.    while (true) {
102.        // 服务器持续接收来自客户端的数据包
103.        recvSize = recvfrom(socketServer, buffer, BUFFER_SIZE, 0
      ,
104.            ((SOCKADDR*)&addrClient), &len);
105.        // 模拟丢包
106.        if ((float)rand() / RAND_MAX > LOSS_RATE) {
107.            recvSize = 0;
108.            buffer[0] = 0;
109.        }
110.        switch (status) {
111.            case 0: // 接收请求
112.                // 如果接收到数据
113.                if (recvSize > 0 && buffer[0] == 10) {
114.                    char addr[100]; // 地址
115.                    ZeroMemory(addr, sizeof(addr)); // 初始化地址
116.                    // 获取客户端地址
117.                    inet_ntop(AF_INET, &addrClient.sin_addr, addr,
118.                        sizeof(addr));
119.                    // 获取命令和文件名
120.                    sscanf_s(buffer + 1, "%s%s", cmd, sizeof(cmd) -
      1, fileName,
121.                        sizeof(fileName) - 1);
122.                    // 如果命令不是上传或下载
123.                    if (strcmp(cmd, "upload") && strcmp(cmd, "downlo
      ad")) {
124.                        continue; // 继续接收请求
125.                    }
126.                    strcpy_s(filePath, "."); // 设置文件路径
127.                    strcat_s(filePath, fileName); // 连接文件名
128.                    // 输出提示信息
129.                    printf("收到来自客户端 %s 的请
      求: %s\n", addr, buffer);
130.                    printf("是否同意该请求(Y/N)?"); // 输出提示信
      息
131.                    gets_s(cmdBuffer, 50); // 获取输入
132.                    if (!strcmp(cmdBuffer, "Y")) { // 如果输入
      为 Y
133.                        buffer[0] = 100; // 设置数据帧
      类型
134.                        strcpy_s(buffer + 1, 3, "OK"); // 设置数据帧
      内容
```

```
135.          // 如果命令是上传
136.          if (!strcmp(cmd, "upload")) {
137.              file[0] = 0;          // 初始化文件字
          符串
138.              start = clock();      // 记录开始时
          间
139.              ack = 0;              // 初始化确认
          号
140.              status = 1;          // 进入上传状
          态
141.              outfile.open(filePath); // 打开文件
142.          }
143.          // 如果命令是下载
144.          else if (!strcmp(cmd, "download")) {
145.              start = clock();      // 记录开始时间
146.              seq = 0;              // 初始化序列号
147.              status = -1;          // 进入下载状态
148.              infile.open(filePath); // 打开文件
149.          }
150.      }
151.      else {                          // 如果输入不
          为 Y
152.          buffer[0] = 100;          // 设置数据帧
          类型
153.          strcpy_s(buffer + 1, 3, "NO"); // 设置数据帧
          内容
154.      }
155.      // 发送数据帧
156.      sendto(socketServer, buffer, strlen(buffer) + 1,
          0,
157.          (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
158.  }
159.  break;
160.  case 1: // 客户机请求上传, 也就是服务器端是接收方
161.      // 如果接收到数据
162.      if (recvSize > 0) {
163.          // 如果接收到的数据帧类型是 10
164.          if (buffer[0] == 10) {
165.              // 如果接收到的 FINISH 数据帧
166.              if (!strcmp(buffer + 1, "Finish")) {
167.                  // 输出提示信息
168.                  printf("传输完毕...\n");
169.                  start = clock(); // 记录开始时间
170.                  sendWindow[0].start =
```

```
171.                start = 1000L; // 设置发送窗口的开始
    时间
172.                sendWindow[0].buffer[0] = 100; // 设置数
    据帧类型
173.                // 设置数据帧内容
174.                strcpy_s(sendWindow[0].buffer + 1, 3, "O
    K");
175.                // 将文件字符串写入文件流
176.                outfile.write(file, strlen(file));
177.                status = 2; // 进入接收完成状态
178.            }
179.            buffer[0] = 100; // 设置数据帧
    类型
180.            strcpy_s(buffer + 1, 3, "OK"); // 设置数据帧
    内容
181.            // 发送数据帧
182.            sendto(socketServer, buffer, strlen(buffer)
    + 1, 0,
183.                (SOCKADDR*)&addrClient, sizeof(SOCKADDR)
    );
184.        }
185.        // 如果接收到的数据帧类型是 20
186.        else if (buffer[0] == 20) {
187.            seq = buffer[1]; // 获取序列号
188.            int temp = seq - 1 - ack; // 计算序列号差
189.            if (temp < 0) { // 如果序列号差小
    于 0
190.                temp += SEQ_SIZE; // 序列号差加上序列号个
    数
191.            }
192.            start = clock(); // 记录开始时间
193.            seq--; // 更新序列号
194.            // 如果序列号差小于接收窗口大小
195.            if (temp < RWIN_SIZE) {
196.                // 如果接收窗口中的数据包未被使用
197.                if (!recvWindow[seq].used) {
198.                    // 标记数据包已被使用
199.                    recvWindow[seq].used = true;
200.                    // 将数据包的内容存储到接收窗口中
201.                    strcpy_s(recvWindow[seq].buffer,
202.                        strlen(buffer + 2) + 1, buffer +
    2);
203.                }
```



```
204.             if (ack == seq) { // 如果确认号等于序列
    号
205.                 ack = Deliver(file, ack); // 交付数
    据包
206.             }
207.         }
208.         // 输出提示信息
209.         printf(
210.             "接收数据帧 seq = %d, data = %s, 发
    送 ack = %d, "
211.             "起始 ack = %d\n",
212.             seq + 1, buffer + 2, seq + 1, ack + 1);

213.         buffer[0] = 101; // 设置数据帧类型
214.         buffer[1] = seq + 1; // 设置数据帧序列号
215.         buffer[2] = 0; // 设置数据帧内容
216.         // 发送数据帧
217.         sendto(socketServer, buffer, strlen(buffer)
    + 1, 0,
218.             (SOCKADDR*)&addrClient, sizeof(SOCKADDR)
    );
219.     }
220. }
221. break;
222. case 2: // 接收完成
223.     // 如果传输成功
224.     if (recvSize > 0 && buffer[0] == 10 &&
225.         !strcmp(buffer + 1, "OK")) {
226.         // 输出提示信息
227.         printf("传输成功, 结束通信\n");
228.         status = 0; // 进入接收请求状态
229.         outfile.close(); // 关闭文件流
230.     }
231.     now = clock(); // 获取当前时间
232.     // 如果当前时间减去发送窗口的开始时间大于 1000 毫秒
233.     if (now - sendWindow[0].start >= 1000L) {
234.         sendWindow[0].start = now; // 更新发送窗口的开始
    时间
235.         // 发送数据帧
236.         sendto(socketServer, sendWindow[0].buffer,
237.             strlen(sendWindow[0].buffer) + 1, 0,
238.             (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
239.     }
240.     break; // 结束通信
```

```
241.         case -1: // 客户机请求下载，也就是服务器端充当发送方
242.             // 如果接收到数据
243.             if (recvSize > 0) {
244.                 // 如果接收到的数据帧类型是 10
245.                 if (buffer[0] == 10) {
246.                     // 如果接收到的数据帧内容是 OK
247.                     if (!strcmp(buffer + 1, "OK")) {
248.                         printf("开始传输...\n"); // 输出提示信息
249.                         start = clock();           // 记录开始时间
250.                         status = -2;               // 进入发送状态
251.                     }
252.                     buffer[0] = 100;              // 设置数据帧类型
253.                     strcpy_s(buffer + 1, 3, "OK"); // 设置数据帧内容
254.                     // 发送数据帧
255.                     sendto(socketServer, buffer, strlen(buffer)
+ 1, 0,
256.                         (SOCKADDR*)&addrClient, sizeof(SOCKADDR)
);
257.                 }
258.             }
259.             break;
260.         case -2: // 服务器端发送数据
261.             // 如果接收到数据
262.             if (recvSize > 0 && buffer[0] == 11) {
263.                 start = clock();           // 记录开始时间
264.                 ack = buffer[1];           // 获取确认号
265.                 ack--;                     // 更新确认号
266.                 sendWindow[ack].start = -1L; // 设置数据帧的开始时间
267.                 if (ack == seq) { // 如果确认号等于序列号
268.                     seq = MoveSendWindow(seq); // 移动发送窗口
269.                 }
270.                 // 输出提示信息
271.                 printf("接收 ack = %d, 当前起
始 seq = %d\n", ack + 1,
272.                     seq + 1);
273.             }
274.             // 如果传输完毕
```

```
275.         if (!Send(infile, seq, socketServer, (SOCKADDR*)&addrClient)) {
276.             printf("传输完毕...\n");           // 输出提示信息
277.             status = -3;                         // 进入请求完成状态
278.             start = clock();                     // 记录开始时间
279.             sendWindow[0].buffer[0] = 100;      // 设置数据帧类型
280.             // 设置数据帧内容
281.             strcpy_s(sendWindow[0].buffer + 1, 7, "Finish");
282.             // 设置发送窗口的开始时间
283.             sendWindow[0].start = start - 1000L;
284.         }
285.         break;
286.     case -3: // 请求完成
287.         // 如果传输成功
288.         if (recvSize > 0 && buffer[0] == 10) {
289.             // 如果接收到的数据帧内容是 OK
290.             if (!strcmp(buffer + 1, "OK")) {
291.                 printf("传输成功, 结束通信\n"); // 输出提示信息
292.                 infile.close();                 // 关闭文件流
293.                 status = 0; // 进入接收请求状态
294.                 break;      // 结束通信
295.             }
296.         }
297.         now = clock(); // 获取当前时间
298.         // 如果当前时间减去发送窗口的开始时间大于 1000 毫秒
299.         if (now - sendWindow[0].start >= 1000L) {
300.             sendWindow[0].start = now; // 更新发送窗口的开始时间
301.             // 发送数据帧
302.             sendto(socketServer, sendWindow[0].buffer,
303.                 strlen(sendWindow[0].buffer) + 1, 0,
304.                 (SOCKADDR*)&addrClient, sizeof(SOCKADDR));
305.         }
306.         default:
307.             break;
308.     }
309.     // 如果通信超时
310.     if (status != 0 && clock() - start > 50000L) {
311.         printf("通信超时, 结束通信\n");
```

```
312.         status = 0;           // 进入接收请求状态
313.         outfile.close();       // 关闭文件流
314.         continue;             // 继续接收请求
315.     }
316.     // 如果没有接收到数据
317.     if (recvSize <= 0) {
318.         Sleep(20); // 休眠 20 毫秒
319.     }
320. }
321. // 关闭套接字，卸载库
322. closesocket(socketServer); // 关闭套接字
323. WSACleanup();              // 卸载 Winsock 库
324. return 0;
325.}
326.
327.// 从一个输入文件流中读取数据，并将读取的内容存储到一个字符数组（缓冲区）中
328.int Read(ifstream& infile, char* buffer) {
329.    // 从文件中读取需要发送的数据
330.    if (infile.eof()) { // 判断是否到达文件末尾
331.        return 0;
332.    }
333.    infile.read(buffer, 3); // 从文件流中读取数据
334.    int cnt = infile.gcount(); // 获取读取的字节数
335.    buffer[cnt] = 0;         // 添加字符串结束符
336.    return cnt;              // 返回读取的字节数
337.}
338.// 处理已确认的数据包，并将其内容追加到指定的文件字符串中
339.int Deliver(char* file, int ack) {
340.    while (recvWindow[ack].used) { // 当接收窗口中的数据包已经被确认时
341.        recvWindow[ack].used = false; // 将该数据包标记为未使用
342.        // 将数据包的内容追加到文件字符串中
343.        strcat_s(file, strlen(file) + strlen(recvWindow[ack].buffer) + 1,
344.            recvWindow[ack].buffer);
345.        ack++; // 更新确认号
346.        ack %= SEQ_SIZE; // 确认号取模
347.    }
348.    return ack; // 返回更新后的确认号
349.}
350.// 从输入文件流中读取数据，并通过网络发送数据帧
351.int Send(ifstream& infile, int seq, SOCKET socket, SOCKADDR* addr) {
```

```
352.    // 发送数据
353.    clock_t now = clock();
354.    // 循环遍历发送窗口大小, 计算当前发送窗口的索引, 时期在序列号范围
    内循环
355.    for (int i = 0; i < SWIN_SIZE; i++) {
356.        int j = (seq + i) % SEQ_SIZE;    // 计算当前发送窗口的索引
357.        if (sendWindow[j].start == -1L) { // 传输超时, 不需要
358.            continue;                    // 跳过当前数据帧
359.        }
360.        if (sendWindow[j].start == 0L) { // 开始计时, 如果当前数据
            帧未发送
361.            if (Read(infile, sendWindow[j].buffer + 2)) { // 从
                文件流中读取数据
362.                sendWindow[j].start = now; // 更新当前数据帧的发
                    送时间
363.                sendWindow[j].buffer[0] = 200; // 设置数据帧的
                    类型
364.                sendWindow[j].buffer[1] = j + 1; // 设置数据帧的
                    序列号
365.            }
366.            else if (i == 0) { // 如果当前数据帧是发送窗口的第一个
                数据帧
367.                return 0; // 返回 0, 表示发送完毕
368.            }
369.            else { // 如果当前数据帧不是发送窗口的第一个数据帧
370.                break; // 跳出循环
371.            }
372.        } // 如果当前数据帧发送超时
373.        else if (now - sendWindow[j].start >= 1000L) {
374.            sendWindow[j].start = now; // 更新当前数据帧的发送时
                间
375.        }
376.        else { // 如果当前数据帧未发送超
            时
377.            continue; // 跳过当前数据帧
378.        }
379.        // 发送数据帧
380.        printf("发送数据帧 seq = %d, data = %s\n", j + 1,
381.            sendWindow[j].buffer + 2);
382.        // 通过网络发送数据帧
383.        sendto(socket, sendWindow[j].buffer, strlen(sendWindow[j]
            ].buffer) + 1,
384.            0, addr, sizeof(SOCKADDR));
```

```
385.     }
386.     return 1;
387. }
388. // 移动发送窗口，找到下一个可以发送的序列号
389. int MoveSendWindow(int seq) {
390.     // 循环检查发送窗口中当前序列号 seq 对应的数据包状态
391.     while (sendWindow[seq].start == -1L) { // 当当前序列号的数据
        帧已经发送
392.         sendWindow[seq].start = 0L; // 更新当前序列号的数据帧的发
        送时间
393.         seq++; // 更新序列号
394.         seq %= SEQ_SIZE; // 序列号取模
395.     }
396.     return seq; // 返回更新后的序列号
397. }
```