

lab3-1实验报告

姓名：吴静

学号：2113285

专业：信息安全

一、实验目的

利用数据包套接字再用户空间实现面向连接的可靠数据传输，功能包括：建立连接，差错检测，接收确认，超时重传等。流量控制采用停等机制，完成给定测试文件的传输。具体包括：

1. 数据包套接字：UDP
2. 建立连接，断开连接：需要实现可靠数据传输，类似TCP的握手和挥手功能
3. 差错检测：校验和的计算和检测
4. 接收确认：在收到对应数据包后返回确认
5. 超时重传：在发出一个数据包后，如果在给定时间内没有返回上一个数据包的确认，则可能发生了丢包或者延时，需要重新发送该数据包
6. 单向传输：需要从客户端向服务器端发送文件，服务器端能将文件下载到本地，客户端发送的文件和cpp文件在同一目录下，服务器端接收的文件和cpp文件也在同一目录下
7. 日志输出：输出收到/发出的数据包的序号，ACK，校验和等，最后输出传输时间和吞吐率
8. 测试文件：测试助教发的测试文件，包括helloworld.txt，1.jpg，2.jpg和3.jpg，需要成功从客户端发送至服务器端

二、实验原理

1. 超时重传

设置一个超时时间，发送端在发送数据报文段后开始计时，如果在超时时间内没有收到接收端发来的确认包，则发生超时，发送端将会重新发送没有收到确认序号的那一个包（重传分组），并且重置计时器。

- 延时：
 - 如果超时重传的超时时间（RTO）设置的太小了，即使发送端可以更敏锐地检测出报文段的丢失，但是同时发送端有可能将一些延迟稍大的报文段误认为是丢失的报文段，造成没有必要重传，此时服务器端会受到两个重复的数据包，此时在服务器端首先要进行一次判断，如果收到了重复的数据包，或者说收到的数据包的序号不是我想要的序号，则进行丢弃。
 - 另一方面，如果RTO设置的太大了，会使得发送端经过较长的时间才发现报文段的丢失，从而降低连接数据传输的吞吐量。
 - 于是设置合理大小的RTO是很重要的，一种方法是根据网络延迟动态调整超时时间，但是本次实验仍然使用固定值的RTO。
- 丢包
 - 在传输过程中因为某些原因导致发送端发送的数据包没有正常到接收端，接收端没有收到数据包，不发送确认，在发送端超时后进行重传。

注意：在所有的超时重传前都要设置为非阻塞模式，否则recvfrom没有接收到数据的话会一直卡在那里，无法达到超时重传的目的，在超时重传后再设置为阻塞模式。

代码如下：

设置为非阻塞模式：

```
1 u_long mode = 1;
2 ioctlsocket(client, FIONBIO, &mode);
```

设置为阻塞模式：

```
1 u_long mode = 0;
2 ioctlsocket(client, FIONBIO, &mode);
```

2. UDP

UDP 是一种无连接的传输层协议，提供面向事务的简单的不可靠信息传送服务。要想实现传输的可靠性，必须在用户应用程序中实现，也就是本次实验主要完成的内容和目标。

3. 差错检测

使用校验和进行差错检测：

- 发送方生成校验和：
 - 填充发送数据包，将校验和部分清零，将数据包用0补齐为16位整数倍；
 - 将发送的数据报分成若干个16位（即2个字节）的位串，每个位串看成一个二进制数；
 - 进行16位二进制反码求和运算（即带循环进位的加法，当最高位有进位时循环到最低位），计算结果取反码即为校验码，写进校验码字段。
- 接收方通过校验和判断：
 - 接收方将接收到的数据包用0补齐为16位整数倍；
 - 按16位整数序列，采用16位二进制反码求和运算；
 - 如果计算结果中16位全为1，则发送的校验包没有错误（即计算结果取反后为0），反之则有错误。

校验和计算代码如下：

```
1 u_short cksum(u_short* message, int size) {
2     int count = (size + 1) / 2; //16位是两个字节
3     u_long sum = 0;
4
5     unsigned short* buf = (unsigned short*)malloc(size);
6     memset(buf, 0, size);
7     memcpy(buf, message, size);
8     //防止message中的数据没有按照16位对其而导致的错误
9
10    while (count--) {
11        sum += *buf++;
```

```

12         if (sum & 0xffff0000) {
13             sum &= 0xffff;
14             sum++;
15         }
16         //处理了反码溢出的情况
17     }
18     u_short result = ~(sum & 0xffff);
19     return result;
20 }

```

4. 停等协议

- 核心：发送方发送完数据包后，等待接收方应答后再继续传输下一个数据包，在此过程中发送方不发送数据包，于是称为停等协议；如果给定时间内没有收到确认数据包，发送方将重传数据包。
- 发送方发送的数据包需要进行编号（seq），区分不同的数据包，由于只需要确认不一样的前后两个数据包，所以理论上只需要1个比特即可。
- 接收方接收到数据包后发送的确认包也需要进行编号（ack），区分不同的确认包，同seq编号一样，理论上只需要一个比特即可。

5. 吞吐量

- 吞吐量是指在单位时间内通过某个网络或系统的数据量。
- 计算吞吐率的公式：

$$\text{吞吐量} = \frac{\text{传输的数据量}}{\text{传输所花费的时间}}$$

三、协议设计

1. 数据包头部Header格式设计

数据头格式如下：



该头部长度为48位，其中0-15位是数据长度，即头部后面跟着的数据的长度；16-31位是校验和，用于检验传输的正确性，即传输过来的包是否损坏；32-39位是标志位，本次实验只使用低3位：

- FIN: 0b100
- ACK: 0b10
- SYN: 0b1
- OVER: 0b111

40-47位是传输的数据包的序号，注意本次实验将序列号和确认号画上等号，对实验结果没有影响，ack从0至255循环使用。

代码实现：

```

1  #pragma pack(2)
2  class Header {
3      /*
4       根据数据长度选择类型：
5       char-8位;short-16位;int-32位;long-32/64位;long long-64位
6       */
7       unsigned short datasize = 0;
8       //数据长度为16位，而int为32位所以不选择int
9       unsigned short sum = 0;
10      //校验和sum同样为16位
11      unsigned char tag = 0;
12      //tag为标志位，低三位分别为FIN,ACK,SYN
13      unsigned char ack = 0;
14      //ack为传输数据包的序号，0-255循环使用
15  public:
16      Header() :datasize(0), sum(0), tag(0), ack(0) {};
17      //初始化
18      void set_tag(unsigned char tag) {
19          this->tag = tag;
20      }
21      void clear_sum() {
22          sum = 0;
23      }
24      void set_sum(unsigned short sum) {
25          this->sum = sum;
26      }
27      unsigned char get_tag() {
28          return tag;
29      }
30      unsigned short get_sum() {
31          return sum;
32      }
33      void set_datasize(unsigned short datasize) {
34          this->datasize = datasize;
35      }
36      void set_ack(unsigned char ack) {
37          this->ack = ack;
38      }
39      int get_datasize() {
40          return datasize;
41      }

```

```

42     unsigned char get_ack() {
43         return ack;
44     }
45     void print_header() {
46         printf("datasize:%d, ack:%d, tag:%d\n", get_datasize(),
47             get_ack(), get_tag());
48     };

```

这里的 `#pragma pack(2)` 表示2字节对齐，需要的时候加上就可以。

2. 数据包Packet结构设计

数据格式如下：



数据包就是在头部后面跟上一个大小为1024个字节的char数组

- 如果有要传输的数据的时候就放在char数组里面，同时对头部的数据长度进行赋值。
- 如果没有要传输的数据的时候（比如发送确认包 `ACK` 或者 `SYN` 包，`FIN` 包等，不带数据时可以只发送头部）

代码实现：

```

1  class Packet {
2  private:
3      Header header;
4      char data_content[1024];
5  public:

```

```

6   Packet() :header() { memset(data_content, 0, 1024); }
7   Header get_header() {
8       return header;
9   }
10  void set_datacontent(char* data_content) {
11      memcpy(this->data_content, data_content, sizeof(data_content));
12  }
13  int get_size() {
14      return sizeof(header) + header.get_datasize();
15  }
16  void print_pkt() {
17      printf("bytes:%d\nseq:%d\n\n", header.get_datasize(),
header.get_ack());
18  }
19  char* get_data_content() {
20      return data_content;
21  }
22  void set_datasize(int datasize) {
23      header.set_datasize(datasize);
24  }
25  int get_datasize() {
26      return header.get_datasize();
27  }
28  void set_ack(unsigned char ack) {
29      header.set_ack(ack);
30  }
31  u_char get_ack() {
32      return header.get_ack();
33  }
34  void set_tag(unsigned char tag) {
35      header.set_tag(tag);
36  }
37  void clear_sum() {
38      header.clear_sum();
39  }
40  void set_sum(unsigned short sum) {
41      header.set_sum(sum);
42  }
43  unsigned char get_tag() {
44      return header.get_tag();
45  }
46  unsigned short get_sum() {
47      return header.get_sum();
48  }
49  void printpacketmessage() {
50      printf("Packet size=%d bytes, tag=%d, seq=%d, sum=%d, datasize=%d\n",
get_size(), get_tag(), get_ack(), get_sum(), get_datasize());
51  }
52  };

```

这里面实现变量的长度的控制的是变量类型：

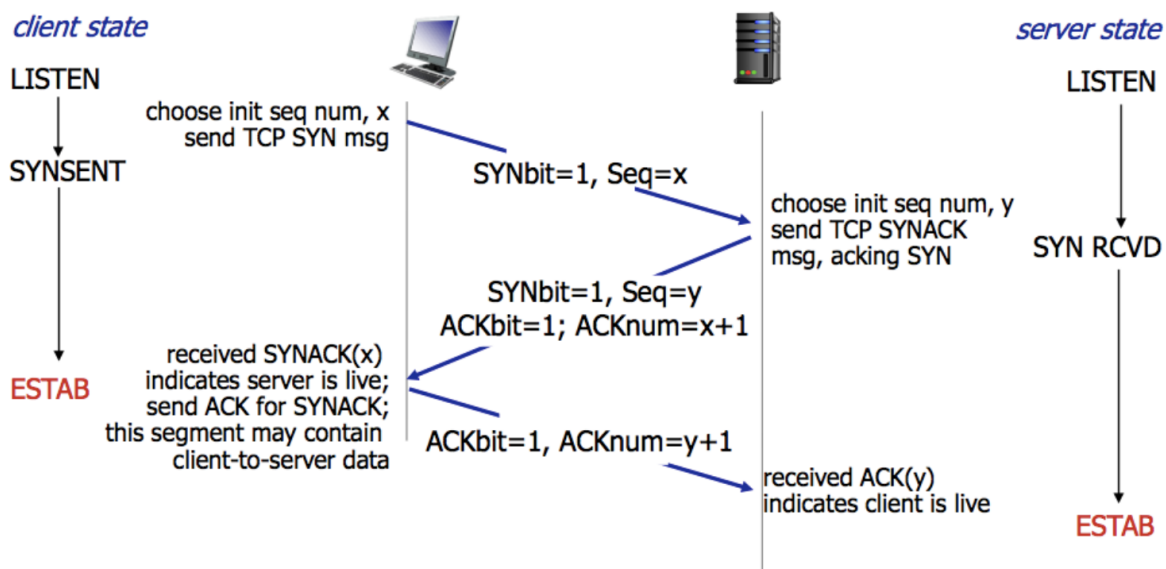
- `u_short`: 16位
- `u_char`: 8位

也可以使用 `typedef` 的重定义来实现（在不同平台下，使用下列名称可以保证固定长度）来实现：

- `int16_t`: 16位
- `int8_t`: 8位

3. 建立可靠连接——3次握手

仿照TCP连接过程中的三次握手，这里也使用三次握手建立可靠连接，流程如下：



假设发送端（客户端）为A，接收端（服务器）为B：

第一次握手：A向B发送 `SYN` 申请建立连接

第二次握手：B收到 `SYN` 后，发送 `SYN + ACK` 给A表示可以建立连接

第三次握手：A收到 `SYN + ACK` 的确认后，再次发送 `ACK` 给B

经过三次握手后，成功建立连接。

代码如下：

客户端：

```
1  int connect(SOCKET& client, SOCKADDR_IN& serv_addr, int& servaddr_len) {
2      printf("-----<begin connect>-----\n\n");
3      //三次握手建立连接
4      Header header;
5      //进行第一次握手：发送SYN
6      header.set_tag(SYN);
7      //置位
8      header.clear_sum();
9      //将校验和置0
10     header.set_sum(cksum((unsigned short*)&header, sizeof(header)));
```

```

11
12     char* send_buffer = new char[sizeof(header)]; //创建发送缓存区
13     /*
14     注意这里sizeof(header)和sizeof(sned_buffer)不一样
15     一个是指针所占的字节数，一个是类所占的字节数*/
16     memcpy(send_buffer, &header, sizeof(header)); //将header中的内容复制给发送缓存区
17     if (sendto(client, send_buffer, sizeof(header), 0, (sockaddr*)&serv_addr,
servaddr_len) == -1)
18     {
19         printf("[Failed send]\nSYN\n\n");
20         return -1;
21     }
22     printTime();
23     printf("[send]\nSYN\n\n");
24     /*
25     int WSAAPI sendto(
26         [in] SOCKET      s,
27         [in] const char   *buf,
28         [in] int          len,
29         [in] int          flags,
30         [in] const sockaddr *to,
31         [in] int          tolen //由 to 参数指向的地址的大小（以字节为单位）
32     );*/
33
34     clock_t start = clock();
35     //开启计时器
36
37     /*此时默认是阻塞模式，当应用程序执行套接字操作（如读取或写入数据）时，操作会一直等待，直到它完成为止。*/
38     u_long mode = 1;
39     ioctlsocket(client, FIONBIO, &mode);
40
41     //第二次握手：接收ACK+SYN
42     //首先有一个超时重传
43     char* recv_buffer = new char[sizeof(header)];
44     while (recvfrom(client, recv_buffer, sizeof(header), 0,
(sockaddr*)&serv_addr, &servaddr_len) <= 0) {
45         /*
46         int recvfrom(
47             [in]          SOCKET    s,
48             [out]         char      *buf,
49             [in]          int       len,
50             [in]          int       flags,
51             [out]         sockaddr  *from,
52             [in, out, optional] int  *fromlen
53         );
54         如果未发生错误，recvfrom 将返回收到的字节数。
55         如果连接已正常关闭，则返回值为零。
56         否则，将返回值 SOCKET_ERROR */
57         if (clock() - start > MAX_TIME) {
58             //超时重传，重新进行第一次握手
59             header.set_tag(SYN);

```



```

60         header.clear_sum();
61         header.set_sum(cksum((unsigned short*)&header, sizeof(header)));
62
63         memcpy(send_buffer, &header, sizeof(header)); //将header中的内容复制给
发送缓存区
64         if (sendto(client, send_buffer, sizeof(header), 0,
(sockaddr*)&serv_addr, servaddr_len) == -1)
65         {
66             printf("[Failed send]\nSYN\n\n");
67             return -1;
68         }
69         start = clock();
70         printf("[timeout]\nRetransmitting SYN.....\n");
71     }
72 }
73
74 mode = 0;
75 ioctlsocket(client, FIONBIO, &mode);
76
77 //如果没有超时的话, 就进行校验和检验并且查看是否是0, 且传回的头部中SYN和ACK是否置1
78 memcpy(&header, recv_buffer, sizeof(header));
79 if (header.get_tag() == ACK_SYN && cksum((unsigned short*)&header,
sizeof(header)) == 0) {
80     printTime();
81     printf("[recv]\nSYN & ACK\n\n");
82
83     //如果接收成功, 进行第三次握手: 发送ACK
84     header.set_tag(ACK);
85     header.clear_sum();
86     header.set_sum(cksum((unsigned short*)&header, sizeof(header)));
87     memcpy(send_buffer, &header, sizeof(header)); //将header中的内容复制给发送
缓存区
88     if (sendto(client, send_buffer, sizeof(header), 0,
(sockaddr*)&serv_addr, servaddr_len) == -1)
89     {
90         printf("[Failed send]\nACK\n\n");
91         return -1;
92     }
93     printTime();
94     printf("[send]\nACK\n\n");
95     printf("-----<connect!>-----\n\n");
96     return 1;
97 }
98 else {
99     //收到的数据包有误
100     printf("[incompactible]\nRetransmitting SYN.....\n\n");
101     return -1;
102 }
103
104 }

```

具体思路如下:

1. 初始化发送的头部信息

- tag 为 SYN
- 因为不带数据所以 datasize 为0
- ack/seq 暂时用不上所以先置0
- sum 清零后进行校验和的计算再写回校验和字段

2. 发送头部，并且设置定时器，等待服务器端发回来的确认包

3. 如果定时器超时，则进行超时重传，并且重新设置定时器

4. 如果接收到服务器端发来的消息，进行校验和的检验以及标志位的确认（是否有 ACK 和 SYN），如果都满足，则第二次握手结束，发送确认包 ACK 给服务器端，连接建立。

服务器端：

```
1  int connect(SOCKET& server, SOCKADDR_IN& client_addr, int& clientaddr_len) {
2      printf("-----<begin connect>-----\n\n");
3
4      Header header;
5      char* recv_buffer = new char[sizeof(header)];
6
7      //第一次握手: 接受SYN
8      while (true) {
9          if (recvfrom(server, recv_buffer, sizeof(header), 0,
10 (sockaddr*)&client_addr, &clientaddr_len) == -1)
11          {
12              printf("错误代码: %d", WSAGetLastError());
13              return -1;
14          }
15
16          //进行校验和的检验并判断其中标志位是否为SYN
17          memcpy(&header, recv_buffer, sizeof(header));
18          if (header.get_tag() == SYN && cksum((u_short*)&header, sizeof(header))
19 == 0)
20          {
21              printTime();
22              printf("[recv]\nSYN\n\n");
23              break;
24          }
25
26          char* send_buffer = new char[sizeof(header)];
27          //第二次握手: 发送SYN+ACK
28          header.set_tag(ACK_SYN);
29          header.clear_sum();
30          header.set_sum(cksum((u_short*)&header, sizeof(header)));
31          memcpy(send_buffer, &header, sizeof(header));
32          if (sendto(server, send_buffer, sizeof(header), 0, (sockaddr*)&client_addr,
33 clientaddr_len) == -1)
34          {
35              printf("[Failed send]\nACK & SYN\n\n");
36          }
37      }
38  }
```

```

34     return -1;
35 }
36 printTime();
37 printf("[send]\nSYN & ACK\n\n");
38
39 //记录第二次握手的时间
40 clock_t start = clock();
41
42 //设置为非阻塞模式
43 u_long mode = 1;
44 ioctlsocket(server, FIONBIO, &mode);
45
46 //第三次握手: 接收ACK
47 //memcpy(recv_buffer, 0, sizeof(recv_buffer));
48 while (recvfrom(server, recv_buffer, sizeof(header), 0,
(sockaddr*)&client_addr, &client_addr_len) <= 0) {
49     //进行超时检测
50     //printf("%d", WSAGetLastError());
51     if (clock() - start > MAX_TIME) {
52         //超时, 重新发送ACK+SYN
53         printf("[time out]\nRetransmitting ACK & SYN again.....\n\n");
54
55         header.set_tag(ACK_SYN);
56         header.clear_sum();
57         header.set_sum(cksum((u_short*)&header, sizeof(header)));
58         memcpy(send_buffer, &header, sizeof(header));
59         if (sendto(server, send_buffer, sizeof(header), 0,
(sockaddr*)&client_addr, client_addr_len) == -1)
60         {
61             printf("[Failed send]\nACK & SYN\n\n");
62             return -1;
63         }
64         //更新第二次握手的时间
65         clock_t start = clock();
66     }
67 }
68
69 //设置为阻塞模式
70 mode = 0;
71 ioctlsocket(server, FIONBIO, &mode);
72
73 //接收到数据后: 开始检验和检测ACK的判断
74 memcpy(&header, recv_buffer, sizeof(header));
75 if (header.get_tag() == ACK && cksum((unsigned short*)&header,
sizeof(header)) == 0) {
76     printTime();
77     printf("[recv]\nACK\n\n");
78     printf("-----<connect!>-----\n\n");
79     return 1;
80 }
81 else {
82     printf("[Failed recv]\nACK\n\n");

```

```

83         return -1;
84     }
85 }

```

具体思路如下：

1. 等待客户端的消息
2. 如果接收到了一个数据包，对其进行校验和的校验以及标志位的确定（标志位是否为 SYN），如果都满足则准备进行第二次握手
3. 初始化数据包：
 - tag 为 SYN + ACK
 - 因为不带数据所以 datasize 为0
 - ack/seq 暂时用不上所以先置0
 - sum 清零后进行校验和的计算再写回校验和字段
4. 发送数据包，完成第二次握手，同时设置定时器，进入循环等待客户端的第三次握手
5. 如果超时则重新发送数据包，重置定时器
6. 接收到数据包后，进行校验和的检测和标志位的确认（标志位是否为 ACK），如果都满足，则连接建立。

4. 文件传输

在三次握手后，客户端进入发送文件的状态，服务器端进入接收文件的状态。

(1) 文件读取操作

- 使用文件流 ifstream 打开用户输入的文件，同时以二进制模式读取
- 从文件流中不断读取字节数据，并进行存储，读取结束后关闭文件流
- 将读取到的字节数据（即文件内容）发送到服务器
- 本次实验采取文件名和文件内容分两次发送的情况。

代码如下：

```

1     string file;
2     printf("请输入要传输的文件名称: \n");
3     cin >> file;
4     ifstream fin(file.c_str(), ifstream::binary);
5     int ptr = 0;
6     unsigned char temp = fin.get();
7     int index = 0;
8     char* buffer = new char[1000000000];
9     while (fin) {
10         buffer[index] = temp;
11         temp = fin.get();
12         index++;
13     }
14     fin.close();

```

```

15
16     send(server, server_addr, len, (char*)(file.c_str()), file.length());
17
18     send(server, server_addr, len, buffer, index);

```

(2) 文件存储操作

- 服务器端接收到数据后，会返回数据长度
- 通过文件名长度，恢复客户端发送过来的文件名
- 使用文件输出流 `ofstream` 打开本地文件，文件名为刚刚恢复的文件名
- 使用循环将收集到的文件中的数据写入本地文件中

代码如下：

```

1     char* name = new char[50];
2     char* data = new char[1000000000];
3     int namelen = recvdata(server, server_addr, len, name);
4     int datalen = recvdata(server, server_addr, len, data);
5     string a;
6     for (int i = 0; i < namelen; i++)
7     {
8         a = a + name[i];
9     }
10    ofstream fout(a.c_str(), ofstream::binary);
11    for (int i = 0; i < datalen; i++)
12    {
13        fout << data[i];
14    }
15    fout.close();
16    cout << "文件已成功下载到本地" << endl;
17 }

```

(3) 发送文件

发送文件中包含两个函数：`send()` 和 `send_package()`。

`send()`：

参数除了基本的服务器端的信息外，还包含了一个需要发送的数据数组（读取文件时读取的二进制数据）以及需要发送数据的长度。将类似“控制系统”对将一个大包切割成很多小包进行发送。

- 根据一个小包可以传输的数据的大小 `MAXSIZE` 计算需要发送多少个包才可以将已有数据发送出去
- 初始化序列号为0
- 循环发送所有小包
 - 初始化每一个小包的数据内容：
 - 除了最后一个包外，其余的包的数据长度为 `MAXSIZE`
 - 最后一个包的数据长度为整个数据长度减去前面已经发送的数据包的数据长度

- 将消息中属于这个包的数据的部分分出来（定位头指针），放在 `send_package()` 中便于该函数进行查找
 - 如果发送成功，对序号进行变化，本次实验中是0-255循环使用
- 文件内容发送结束，最后发送一个 `OVER` 包表示发送结束，同时设置定时器，接收服务器端发来的 `OVER`
- 接收到了服务器端发来的数据包后，对其进行校验和的检验以及标志位的检验，如果都无误，则输出提示信息：“对方已接收到文件”
- 发送结束

代码如下：

```
1  int send(SOCKET& client, SOCKADDR_IN& server_addr, int &serveraddr_len, char*
    data_content, int datasize) {
2      //先算要发多少包: num=len/MAXSIZE+是否有余数（有余数就要多一个包）
3      int package_num = datasize / MAXSIZE + (datasize % MAXSIZE == 0 ? 0 : 1);
4
5      //确认号（序列号）从0开始
6      int seqnum = 0;
7
8      printf("-----即将开始发送当前文件-----\n\n");
9
10     for (int i = 0; i < package_num; i++) {
11         //循环发送所有分开后的数据包
12         printf("即将发送当前文件中的第%d号数据包: \n", i);
13         int len=0; //每次要发送数据包的长度，前面都为MAXSIZE，最后一次发送剩下的
14         if (i == package_num - 1)
15             len = datasize - (package_num - 1) * MAXSIZE;
16         else
17             len = MAXSIZE;
18
19         //发送每一个数据包
20         if (send_package(client, server_addr, serveraddr_len, data_content + i
    * MAXSIZE, len, seqnum) == -1) {
21             //如果发送失败的话
22             printf("[send package Failed]\n\n");
23             //重新发送该数据包
24             i--;
25             continue;
26         }
27         //如果发送成功
28         seqnum = (seqnum + 1) % 256;
29         printf("当前文件中的第%d号数据包发送成功! \n\n", i);
30     }
31
32     //for循环结束，发送数据包结束，接着开始发送结束标志: over
33     //初始化要发送的结束包:
34     Header header;
35     header.set_tag(OVER);
36     header.set_datasize((u_short)0);
37     header.set_ack((u_char)0);
38     header.clear_sum();
```

```

39     header.set_sum(cksum((u_short*)&header, sizeof(header)));
40
41     //初始化要发送的数据:
42     char* send_buffer = new char[sizeof(header)];
43     memcpy(send_buffer, &header, sizeof(header));
44     if (sendto(client, send_buffer, sizeof(header), 0, (SOCKADDR*)&server_addr,
serveraddr_len) == -1) {
45         printf("[Failed send]\nOVER\n\n");
46         return -1;
47     }
48     printTime();
49     printf("[send]\nOVER\n\n");
50
51     //存储当前时间
52     clock_t start = clock();
53
54     u_long mode = 1;
55     ioctlsocket(client, FIONBIO, &mode);
56
57     char* recv_buffer = new char[sizeof(header)];
58
59     while(true)
60     {
61         while (recvfrom(client, recv_buffer, sizeof(header), 0,
(SOCKADDR*)&server_addr, &serveraddr_len) <= 0) {
62             if (clock() - start > MAX_TIME) {
63                 printf("[timeout]\nresend OVER again.....\n\n");
64                 if (sendto(client, send_buffer, sizeof(header), 0,
(SOCKADDR*)&server_addr, serveraddr_len) == -1) {
65                     printf("[Failed send]\nOVER\n\n");
66                 }
67                 start = clock();
68             }
69         }
70
71         mode = 0;
72         ioctlsocket(client, FIONBIO, &mode);
73
74         memcpy(&header, recv_buffer, sizeof(header));
75         if (header.get_tag() == OVER && cksum((u_short*)&header,
sizeof(header)) == 0) {
76             printTime();
77             printf("[recv]\nOVER\n\n");
78             printf("-----对方已接受到文件-----\n\n");
79             break;
80         }
81         else
82             continue;
83     }
84     return 1;
85 }

```

`send_package()` :

参数除了基本的服务器端的信息外，还包含了一个需要发送的数据数组（`send()`函数需要此次发送数据包中发送的数据），需要发送数据的长度以及一个序列号。将承担实际发送数据的功能。

- 初始化数据部分：包括头部和数据内容
 - `datasize` 为需要发送的数据的长度，为参数中给定的
 - `tag` 在这里不需要，不变动即可（创建时初始化为0）
 - `seq` 为即将发送的数据包的序列号，为参数中给定的
 - `data_content` 部分为需要发送的数据，也为参数中指定
 - `sum`（校验和）先清零再计算，最后结果写回 `sum` 中
- 发送数据包，并启动定时器，准备接收服务器端传回来的确认包
- 如果一定时间内没有收到数据包，则进行超时重传操作
- 如果收到了数据包，进行序列号的检测（确认号的检测）和标志位的检查（是否为 `ACK`），如果都无误，则发送成功且对方成功接收，则函数退出。
- 该文件包发送结束

代码如下：

```
1  int send_package(SOCKET& client, SOCKADDR_IN& server_addr, int& serveraddr_len,
2  char* data_content, int datasize, int& seq) {
3      //传输单个数据包：每个数据包=头部+数据
4      //printf("%d\n", (u_char)seq);
5      Packet* sendpkt = new Packet();
6      //要发送的内容，包括头和数据部分
7
8      //初始化数据头：
9      sendpkt->set_datasize(datasize); //初始化数据长度
10     sendpkt->clear_sum(); //对序列号进行清零
11     sendpkt->set_ack((unsigned char)seq); //初始化序列号seq，注意此时seq是u_char类型
12     //sendpkt->print_pkt();
13     //此时数据部分不为0，所以要等数据部分初始化后再开始计算校验和
14
15     //初始化数据：
16     sendpkt->set_datacontent(data_content);
17
18     //初始化数据头的校验和：
19     sendpkt->set_sum(cksum((u_short*)sendpkt, sendpkt->get_size()));
20
21     //检验发送数据包
22     printTime();
23     printf("检查数据包内容: \n");
24     sendpkt->printpacketmessage();
25
26     //发送数据包
27     if (sendto(client, (char*)sendpkt, sendpkt->get_size(), 0,
28         (SOCKADDR*)&server_addr, serveraddr_len) == -1) {
29         printf("[Failed send]\nPacket\n\n");
30     }
```



```

28     return -1;
29 }
30 printTime();
31 printf("已成功发送数据包: ");
32 sendpkt->printpacketmessage();
33
34 //记录当前时间
35 clock_t start = clock();
36
37 Header* header=new Header();
38 //等待接收ACK等信息, 同时验证seq
39 while (true) {
40     //首先设置为非阻塞状态, 不然recvfrom会一直停住
41     //recvfrom() 函数在没有可用数据时将立即返回, 而不会阻塞程序
42     u_long mode = 1;
43     ioctlsocket(client, FIONBIO, &mode);
44
45     //首先进行超时检测
46     while (recvfrom(client, (char*)header, MAXSIZE, 0,
(SOCKADDR*)&server_addr, &serveraddr_len) <= 0) {
47         //进行超时检测
48         if (clock() - start > MAX_TIME) {
49             //超时, 重新发送数据
50             printf("[timeout]\nRetransmitting Packet.....\n");
51             if (sendto(client, (char*)sendpkt, sendpkt->get_size(), 0,
(SOCKADDR*)&server_addr, serveraddr_len) == -1) {
52                 printf("[Failed]\nPacket\n\n");
53                 return -1;
54             }
55             //重置发送时间
56             start = clock();
57         }
58     }
59     //接收到数据, 进行序列号的检测和ACK的确认
60     if (header->get_ack() == (u_char)seq && header->get_tag() == ACK) {
61         printTime();
62         printf("对方已接受到数据包并发送确认: ");
63         header->print_header();
64         break;
65     }
66     else {
67         continue;
68     }
69 }
70 //改回阻塞模式
71 u_long mode = 0;
72 ioctlsocket(client, FIONBIO, &mode);
73
74 return 1;
75 }

```

(4) 接收文件

发送文件中包含一个函数：`recv_data()`。

`recv_data()`：

参数除了基本的客户端的信息外，还包含了一个用于数据存储的指针。将接收数据包并对需要的文件内容进行存储。

- 循环接收所有的数据包，直到发过来的数据包的标志位是 `OVER` 时退出接收，数据接收完毕，函数退出
- 如果接收到的数据包的标志位不是 `OVER`，则进行序列号的检测，判断是不是当前我们需要的数据包
- 如果不是我们当前需要的数据包，则对数据包进行丢弃，不做存储，继续接收下一个数据包
- 如果是我们要存储的数据包，则对数据包的内容进行存储，存储的方式如下：
 - 接收所有数据包以前初始化一个变量 `file_len` 对当前已经接收的数据长度进行存储
 - 使用参数中含有的指针，加上当前已经接收的数据长度的偏移量，可以定位到当前接收到的数据包的内容应该进行存储的开始处
 - 存储完成后，更新 `file_len`
- 同时发送一个确认包，该确认包的序号/确认号是当前存储的数据包的序列号
- 改变序列号为下一次我们期待接收到的数据包的序列号
- 如果收到了一个数据包，标志位为 `OVER`，则同样发送一个 `OVER` 至客户端，代表“我接收完毕”，然后退出文件的接收。

代码如下：

```
1  int recvdata(SOCKET& server,SOCKADDR_IN& client_addr,int&clientaddr_len,char*
   data) {
2      printf("-----开始接收当前文件-----\n\n");
3      Packet* recvpkt = new Packet();//接收传过来的数据包（带数据的那种）
4      Header header;//发送的确认头（不带数据的那种）
5
6      int seq_predict = 0;//seq_predict是期待收到的序列号（确认号）
7
8      int file_len = 0;
9      //目前已经保存的文件的长度——用于标识data应该从哪里开始存
10
11     char* recv_buffer = new char[MAXSIZE + sizeof(recvpkt->get_header())];
12     //接收缓存区
13     char* send_buffer = new char[sizeof(header)];
14     //发送缓存区
15
16     while (true) {
17         //循环接受所有的数据包，退出条件是数据包接收完毕
18         int length = recvfrom(server, recv_buffer, sizeof(recvpkt-
>get_header()+MAXSIZE, 0, (SOCKADDR*)&client_addr, &clientaddr_len);
19         if (length == -1) {
20             printf("[Failed recv]\n");
21         }
22     }
```

```

23     memcpy(recvpkt, recv_buffer, sizeof(recvpkt->get_header()) + MAXSIZE);
24     //判断是否结束，如果已经是最后一个包，则退出接收
25     if (recvpkt->get_tag() == OVER && cksum((u_short*)recvpkt,
sizeof(recvpkt->get_header()))==0) {
26         printTime();
27         printf("[recv]\nOVER\n\n");
28         break;
29     }
30
31     //判断当前包是否是我们需要的包
32     if (seq_predict!=int(recvpkt->get_ack())) {
33         printTime();
34         printf("该数据包重复发送，不进行存储! \n");
35         continue;//这个包不做存储，继续接受下一个包
36     }
37
38     //这个包是我们需要的包，存储数据并打印提示信息
39     printTime();
40     printf("接收到目标数据包:\n");
41     recvpkt->printpacketmessage();
42
43     memcpy(data + file_len, recvpkt->get_data_content(), int(recvpkt-
>get_datasize()));
44     //data+file_len表示数据应该从哪里开始存储
45     //recvpkt->get_data_content()表示应该存储的数据
46     //recvpkt->get_datasize()表示存储数据的长度
47
48     //更新已存储文件长度
49     file_len += recvpkt->get_datasize();
50
51     //存储完毕，返回ACK以及当前文件的ack (seq)
52     header.set_tag(ACK);
53     header.set_datasize(0);
54     header.clear_sum();
55
56     //收到需要的包，返回的确认号为当前包的序列号
57     header.set_ack((u_char)seq_predict);
58
59     //计算cksum
60     header.set_sum(cksum((u_short*) &header, sizeof(header)));
61
62     memcpy(send_buffer, &header, sizeof(header));
63
64     if (sendto(server, send_buffer, sizeof(header), 0,
(SOCKADDR*)&client_addr, clientaddr_len)==-1) {
65         printf("[Failed send]\n\n");
66     }
67     printTime();
68     printf("已发送确认: \n");
69     header.print_header();
70     printf("\n");
71

```

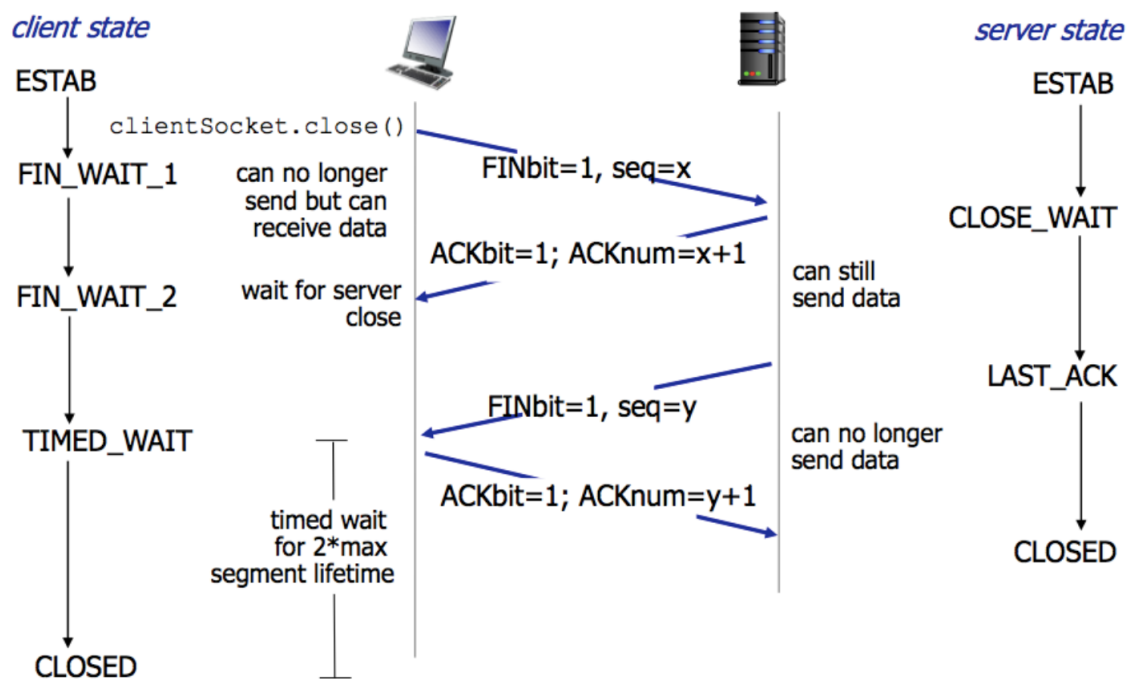
```

72      //更新seq和seq_predict
73      //下一个期待接收到的包为当前期待接收到的包的序号+1, 模256 (0-255循环使用)
74      seq_predict = (seq_predict + 1) % 256;
75
76  }
77  //文件接收完毕, 发送OVER
78  header.clear_sum();
79  header.set_tag(OVER);
80  header.set_datasize(0);
81  header.set_sum((cksum((u_short*)&header, sizeof(header))));
82  memcpy(send_buffer, &header, sizeof(header));
83  if (sendto(server, send_buffer, sizeof(header), 0, (SOCKADDR*)&client_addr,
clientaddr_len)==-1) {
84      printf("[Failed send]\n\n");
85      return -1;
86  }
87  printTime();
88  printf("[send]\nOVER\n\n");
89  printf("-----成功接收当前文件-----\n\n");
90  return file_len; //返回读取的字节数, 为了之后的存储数据
91 }

```

5. 断开连接——4次挥手

仿照TCP断开连接过程中的四次挥手，这里也使用四次挥手断开连接，流程如下：



假设发送端（客户端）为A，接收端（服务器）为B：

第一次挥手：A向B发送 `FIN` 表示数据已经发送结束，请求断开连接

第二次挥手：B向A发送 ACK 表示收到断开连接请求

第三次挥手：B向A发送 FIN 和 ACK 表示数据发送结束，可以断开连接

第四次挥手：A向B发送 ACK 表示收到断开连接数据包

经过四次挥手后，连接断开。

注意：由于这一次实验是单向传输，说明服务器端没有数据要发送给客户端，所以这里其实两次就够，但是为了模拟TCP的可靠传输，所以还是做了四次挥手。

代码如下：

客户端：

```
1  int disconnect(SOCKET& client, SOCKADDR_IN& serv_addr, int servaddr_len) {
2      printf("-----<begin disconnect>-----\n\n");
3
4      Header header;
5      char* send_buffer = new char[sizeof(header)];
6
7      //进行第一次挥手：发送FIN
8      header.set_tag(FIN);
9      header.clear_sum();
10     header.set_sum(cksum((u_short*)&header, sizeof(header)));
11     memcpy(send_buffer, &header, sizeof(header));
12     if (sendto(client, send_buffer, sizeof(header), 0, (sockaddr*)&serv_addr,
servaddr_len) == -1)
13     {
14         printf("[Failed send]\nFIN\n\n");
15         return -1;
16     }
17     printTime();
18     printf("[send]\nFIN\n\n");
19
20     clock_t start = clock();
21     //记录第一次挥手时间
22
23     /*此时仍然默认为阻塞模式，需要设置为非阻塞模式*/
24     u_long mode = 1;
25     ioctlsocket(client, FIONBIO, &mode);
26
27     //进行第二次挥手：接受ACK
28     char* recv_buffer = new char[sizeof(header)];
29     while (recvfrom(client, recv_buffer, sizeof(header), 0,
(sockaddr*)&serv_addr, &servaddr_len) <=0)
30     {
31         if (clock() - start > MAX_TIME)//超时，重新第一次挥手
32         {
33             header.set_tag(FIN);
34             header.clear_sum();
35             header.set_sum(cksum((u_short*)&header, sizeof(header)));
36             memcpy(send_buffer, &header, sizeof(header));//将首部放入缓冲区
```

```

37         if(sendto(client, send_buffer, sizeof(header), 0,
(sockaddr*)&serv_addr, servaddr_len))
38         {
39             printf("[Failed send]\nFIN\n\n");
40             return -1;
41         }
42         start = clock();
43         //更新时间
44         printf("[timeout]\nRetransmitting FIN.....\n\n");
45     }
46 }
47
48 //进行校验和检验以及ACK
49 memcpy(&header, recv_buffer, sizeof(header));
50 if (header.get_tag() == ACK && cksum((unsigned short*)&header,
sizeof(header) == 0)) {
51     printTime();
52     printf("[recv]\nACK\n\n");
53
54 }
55 else {
56     //检验包出错
57     printf("[Failed recv]\nACK\n\n");
58     return -1;
59 }
60
61 //设置为阻塞模式
62 mode = 0;
63 ioctlsocket(client, FIONBIO, &mode);
64
65 //进行第三次挥手: 等待FIN+ACK
66 while (recvfrom(client, recv_buffer, sizeof(header), 0,
(sockaddr*)&serv_addr, &servaddr_len) != SOCKET_ERROR) {
67     //进行校验和检验, 且对ACK标志位进行检测
68     memcpy(&header, recv_buffer, sizeof(header));
69     if (header.get_tag() == ACK_FIN && cksum((unsigned short*)&header,
sizeof(header) == 0)) {
70         //检测成功
71         printTime();
72         printf("[recv]\nFIN & ACK\n\n");
73
74         //第四次挥手: 发送ACK
75         header.set_tag(ACK);
76         header.clear_sum();
77         header.set_sum(cksum((u_short*)&header, sizeof(header)));
78         memcpy(send_buffer, &header, sizeof(header)); //将首部放入缓冲区
79         if (sendto(client, send_buffer, sizeof(header), 0,
(sockaddr*)&serv_addr, servaddr_len)==-1)
80         {
81             printf("[Failed send]\nACK\n\n");
82             return -1;
83         }

```

```

84         printTime();
85         printf("[send]\nACK\n\n");
86         start = clock();
87         printf("Bye.....\n\n");
88         break;
89     }
90 }
91 /*while (clock() - start == 2 * MSL) {
92     printf("Bye.....\n");
93     break;
94 }*/
95 printf("-----<disconnect>-----\n\n");
96 return 1;
97 }

```

具体思路如下：

- 初始化发送 FIN 包
 - 由于没有数据发送，所以 datasize 为0
 - tag 为 FIN
 - 由于这里用不上 seq，所以默认即可
 - sum 先清零，进行校验和的计算后再写回 sum
- 发送 FIN 包，并且启动定时器
- 若在超时时间以内没有收到数据包则进行超时重传，并且重新设置定时器
- 若接收到了数据包，要先进行校验和的检测以及标志位的确认（是否为 ACK）若都无误则第二次挥手结束
- 这里如果是双向传输，有可能服务器端会继续发送数据给客户端，但是本次实验不存在这种情况
- 等待服务器端的断开连接的数据包，注意这里不设置定时器，因为要等服务器端发完全部数据，这里时间不确定
- 收到服务器端发来的数据包后，要对数据包中的内容进行检测，包括校验和是否为0，标志位是否为 FIN + ACK
- 若都无误，则第三次挥手结束，发送一个 ACK 的数据包给服务器端，第四次挥手结束
- 本来这里应该等 2MSL 时间再断开连接的，但是本次实验在客户端发送 ACK 后直接断开连接。

服务器端：

```

1 int disconnect(SOCKET& server, SOCKADDR_IN& client_addr, int clientaddr_len) {
2     printf("-----<begin disconnect>-----\n\n");
3
4     Header header;
5
6     char* recv_buffer = new char[sizeof(header)];
7     //第一次挥手：接收FIN
8     while (true) {
9         int length = recvfrom(server, recv_buffer, sizeof(header), 0,
10             (sockaddr*)&client_addr, &clientaddr_len);

```

```

10     memcpy(&header, recv_buffer, sizeof(header));
11     if (header.get_tag() == FIN && cksum((unsigned
short*)&header, sizeof(header)) == 0) {
12         printTime();
13         printf("[recv]\nFIN\n\n");
14         break;
15     }
16 }
17
18 char* send_buffer = new char[sizeof(header)];
19
20 //开始第二次挥手: 发送ACK
21 header.set_tag(ACK);
22 header.clear_sum();
23 header.set_sum(cksum((unsigned short*)&header, sizeof(header)));
24 memcpy(send_buffer, &header, sizeof(header));
25 if (sendto(server, send_buffer, sizeof(header), 0, (sockaddr*)&client_addr,
clientaddr_len) == -1) {
26     return -1;
27 }
28 printTime();
29 printf("[send]\nACK\n\n");
30
31 //开始第三次挥手: 发送FIN+ACK
32 header.set_tag(ACK_FIN);
33 header.clear_sum();
34 header.set_sum(cksum((unsigned short*)&header, sizeof(header)));
35 memcpy(send_buffer, &header, sizeof(header));
36 if (sendto(server, send_buffer, sizeof(header), 0, (sockaddr*)&client_addr,
clientaddr_len) == -1) {
37     return -1;
38 }
39 clock_t start = clock();
40 //记录第三次挥手的时间
41 printTime();
42 printf("[send]\nFIN & ACK\n\n");
43
44 u_long mode = 1;
45 ioctlsocket(server, FIONBIO, &mode);
46
47 //开始第四次挥手: 等待ACK
48 while (recvfrom(server, recv_buffer, sizeof(header), 0,
(sockaddr*)&client_addr, &clientaddr_len) <= 0) {
49     //检测超时
50     if (clock() - start > MAX_TIME) {
51         printf("[time out]\nRetransmitting ACK & FIN again.....\n\n");
52
53         header.set_tag(ACK_FIN);
54         header.clear_sum();
55         header.set_sum(cksum((unsigned short*)&header, sizeof(header)));
56         memcpy(send_buffer, &header, sizeof(header));

```



```

57         if (sendto(server, send_buffer, sizeof(header), 0,
(sockaddr*)&client_addr, clientaddr_len) == -1) {
58             return -1;
59         }
60         clock_t start = clock();
61         //记录第三次挥手的时间
62         printTime();
63         printf("[send]\nFIN & ACK\n\n");
64     }
65 }
66
67 mode = 0;
68 ioctlsocket(server, FIONBIO, &mode);
69
70 //接收到信息，开始判断是否为ack且是否校验和是否为0
71 memcpy(&header, recv_buffer, sizeof(header));
72 if (header.get_tag() == ACK && cksum((u_short*)&header, sizeof(header)) ==
0) {
73     printTime();
74     printf("[recv]\nACK\n\n");
75     printf("-----<disconnect>-----\n\n");
76     return 1;
77 }
78 else {
79     //校验包出错
80     printf("[Failed recv]\nACK\n\n");
81     return -1;
82 }
83 }

```

具体思路如下：

- 等待客户端的数据包
- 接收到数据包后，对数据包进行检测，包括校验和的检测和标志位的检测（是否为 FIN）
- 如果校验无误，说明客户端已经进行了第一次挥手，即将进行第二次挥手
- 发送标志位为 ACK 的数据包给客户端
- 进行第三次挥手，发送标志位为 FIN + ACK 的数据包给客户端，同时设置定时器
- 如果在给定时间内没有收到客户端的确认包，则重新发送 FIN + ACK 的包给客户端
- 如果收到了客户端的数据包，则对数据包进行检测，包括校验和的检测和标志位的检测（是否为 ACK）
- 如果两者都检测无误，则第四次挥手结束，连接断开。

四、实验结果

设置路由器端口为8081，服务器端口为8080，丢包率为：5%，延时为3ms：

Router

路由器IP: 127 . 0 . 0 . 1 服务器IP: 127 . 0 . 0 . 1

端口: 8081 服务器端口: 8080

丢包率: 5 % 延时: 3 ms

确定 修改

日志

```
count:14.  
Delay 3 ms.  
count:15.  
Delay 3 ms.  
count:16.  
Delay 3 ms.  
count:17.  
Delay 3 ms.  
count:18.
```

客户端和服务端进行连接:

服务器:

```
Server listening on port 8080...  
-----<begin connect>-----  
  
[2023-11-17 20:7:10]  
[recv]  
SYN  
  
[2023-11-17 20:7:10]  
[send]  
SYN & ACK  
  
[2023-11-17 20:7:10]  
[recv]  
ACK  
  
-----<connect!>-----
```

客户端:

```
-----<begin connect>-----  
  
[2023-11-17 20:7:10]  
[send]  
SYN  
  
[2023-11-17 20:7:10]  
[recv]  
SYN & ACK  
  
[2023-11-17 20:7:10]  
[send]  
ACK
```

对 1.jpg 进行发送：

以最后一个数据包和OVER数据包为例：

服务器端：

```
[2023-11-17 20:10:20]
接收到目标数据包：
Packet size=847 bytes, tag=0, seq=21, sum=41299, datasize=841
[2023-11-17 20:10:20]
已发送确认：
datasize:0, ack:21, tag:2

[2023-11-17 20:10:20]
[recv]
OVER

[2023-11-17 20:10:20]
[send]
OVER

-----成功接收当前文件-----
```

客户端：

```
即将发送当前文件中的第1813号数据包：
[2023-11-17 20:10:20]
检查数据包内容：
Packet size=847 bytes, tag=0, seq=21, sum=41299, datasize=841
[2023-11-17 20:10:20]
已成功发送数据包：Packet size=847 bytes, tag=0, seq=21, sum=41299, datasize=841
[2023-11-17 20:10:20]
对方已接受到数据包并发送确认：datasize:0, ack:21
当前文件中的第1813号数据包发送成功！

[2023-11-17 20:10:20]
[send]
OVER

[2023-11-17 20:10:20]
[recv]
OVER

-----对方已接受到文件-----
```

断开连接：

服务器端：

```
-----<begin disconnect>-----  
[2023-11-17 20:10:20]  
[recv]  
FIN  
  
[2023-11-17 20:10:20]  
[send]  
ACK  
  
[2023-11-17 20:10:20]  
[send]  
FIN & ACK  
  
[2023-11-17 20:10:20]  
[recv]  
ACK  
  
-----<disconnect>-----  
  
文件已成功下载到本地
```







客户端:

```
-----<begin disconnect>-----  
[2023-11-17 20:10:20]  
[send]  
FIN  
  
[2023-11-17 20:10:20]  
[recv]  
ACK  
  
[2023-11-17 20:10:20]  
[recv]  
FIN & ACK  
  
[2023-11-17 20:10:20]  
[send]  
ACK  
  
Bye.....  
  
-----<disconnect>-----
```

传输时间和吞吐率:

```
传输时间: 135s  
吞吐率: 13758.169922bytes/s
```

传输结果展示:

名称	修改日期
 x64	2023/11/14 16:43
 1.jpg	2023/11/17 20:10
 Server.cpp	2023/11/17 19:13
 Server.vcxproj	2023/11/13 14:30
 Server.vcxproj.filters	2023/11/13 14:30
 Server.vcxproj.user	2023/11/13 14:30

至此，整个过程结束。

五、总结

通过这次实验，我熟悉了差错检测（校验和的计算），停等协议，超时重传等网络传输的内容，了解了校验和的计算，对整个文件传输过程有了个大概的认识。