

实验3-1 可靠数据传输编程

郭坤昌 2012522

要求

利用数据报套接字在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、确认重传等。流量控制采用停等机制，完成给定测试文件的传输。

执行流程

本实验是建立在UDP套接字上的单向可靠数据传输实现（客户端向服务器发送数据）。

开始时由客户端选择传输文件，建立连接时与服务器约定最大数据段大小（MSS，maximum segment size），并发送文件相关参数（文件名、文件大小）；建连后以校验确认、超时重传、停等协议保障客户端向服务器的单向可靠数据传输；文件发送完毕后客户端主动提起断开连接，与服务器相互“挥手”后各自释放资源。

程序在执行时，打印相关日志：时间、日志类型、具体动作或数据包内容或状态等。

协议设计

数据包格式设计

数据应包含传输的数据以及对数据的相关说明（数据包首部），设计如下：

```
1  #pragma pack(1) // align to 1 byte
2  struct UDP_Header {
3      u32 seq_num;                // 4B, sequence number
4      u32 ack_num;                // 4B, acknowledgement number
5      u16 length;                 // 2B, length of data segment
6      u16 checksum;               // 2B, checksum of the packet
7      u8 flags;                   // 1B, flags of the packet, one hot encoded,
                                // 0x01 for SYN, 0x02 for ACK, 0x04 for FIN, 0x08 for RST
8      u8 filter;                  // 1B, filler to make the header align to 16
                                // bits
9  };
10 struct UDP_Packet{
11     UDP_Header header;
12     u8 data[DATA_MAX_SIZE];    // data of the packet
13 };
14 struct File_Info { // this struct is used to send file info to the server
15     u16 MSS;                // maximum segment size
16     u32 file_size;
17     u16 file_name_len;
18     u8 file_name[FILE_NAME_MAX_SIZE];
19 };
20 #pragma pack() // align to default
```

首部包含了序列号、确认号、数据字段长度、校验和以及数据包类型标识。需要注意的是，这里是模拟的UDP数据包，低层通过IP协议传输，其指示数据长度的字段有16位，因此这里能够传输的数据字段长度不应当大于 $2^{16} - \text{HEADER_SIZE}$

另外设计了用以“握手”的数据格式 `File_Info`，包含 `MSS`、文件大小、文件名长度（硬性约定其大小不超过 256）和文件名，这样客户端将该数据传递给服务器解析，并由服务器确认，完成连接时的相互约定。

面向连接——握手&挥手

连接的建立

连接的建立完全参照了TCP协议建立连接的“三册握手”过程：

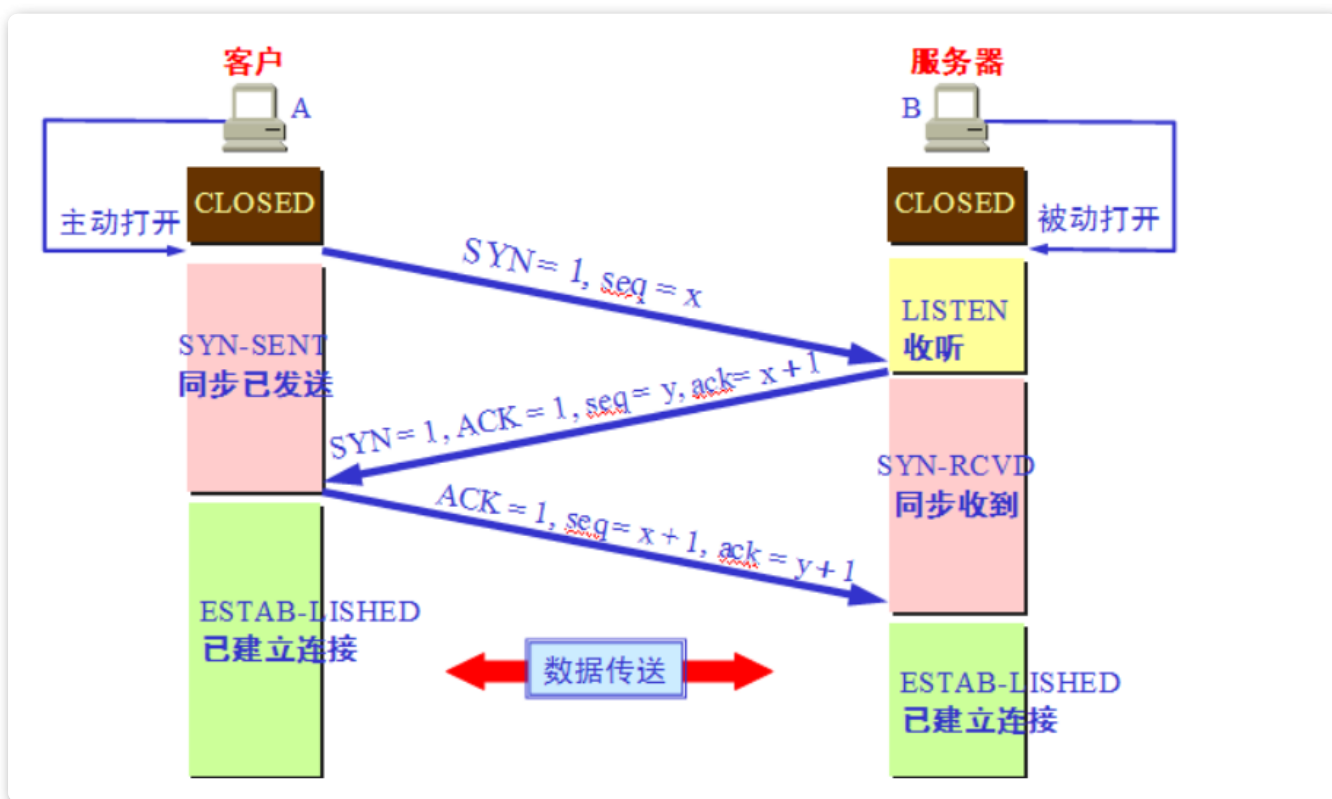
1. 客户端主动打开，发送SYN，其中包含约定内容（如前所述的 `File_Info` 数据）
2. 服务器解析SYN数据包，确定 `MSS`、文件名、文件大小，向客户端发送确认



MSS在这里被设定为客户端发送缓冲区大小和服务接收缓冲区大小的最小值，保证数据能够被完整发送和接收

3. 客户端收到确认，向服务器确认收到约定，并在该数据包中负载数据，之后继续传输后续内容

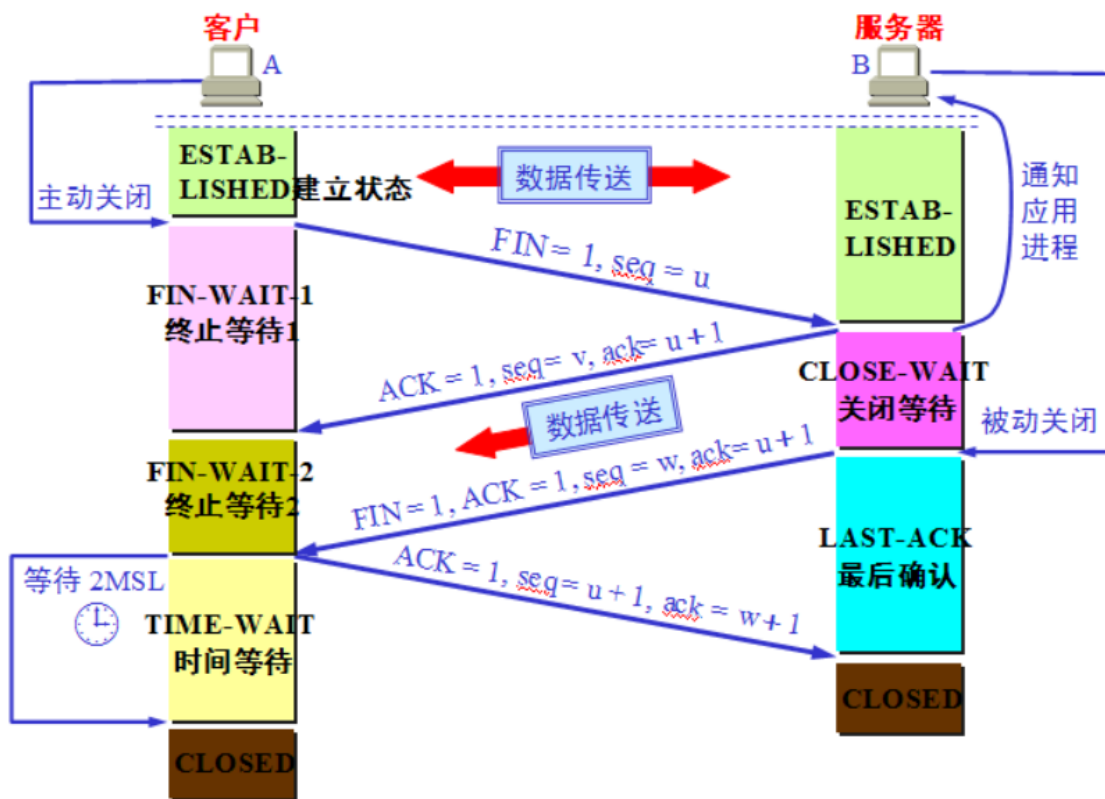
这里如何设定序列号和确认号，并进行确认的方式，在后续差错确认、超时重传部分进行详细介绍。



连接的断开

连接断开的过程类似TCP协议，但不为四次挥手，而为三次挥手的过程：

1. 客户端接收到服务器最后一个确认后，文件发送完毕，主动发送FIN类型数据包，等待服务器确认
2. 服务器接收到客户端断开请求，对此进行确认，发送一个FIN_ACK类型数据包，等待客户端确认
3. 客户端接收到服务器确认，只发送一个确认数据包，之后断开连接；服务器确认后断开连接



差错检测和确认——校验和&确认号依确认数据量递增

差错检测使用16位校验和方式。为数据打包时，将所有数据以16位的方式相加、回卷，保存到校验和字段。因此在校验时，只需要将所有数据（包括校验和字段）以16位方式相加、回卷，只需检验最终计算结果是否为全1即可。

$$checksum + (-checksum) = 0xffff$$

相关代码：

```
1  bool PacketManager::is_corrupted(UDP_Packet *packet) {
2      u32 checksum = 0;
3      u16* ptr = (u16*) packet;
4      u16 total_length = TOTAL_LEN(packet);
5      while(total_length > 1) {
6          checksum += *ptr++;
7          total_length -= 2;
8      }
9      if(total_length > 0) { // if the length is odd, add the last byte
10         checksum += *(u8*)ptr;
11     }
12     checksum = (checksum >> 16) + (checksum & 0xffff); // add high 16 to low
13     16
14     checksum += (checksum >> 16);
15     return (u16)(checksum) != 0xffff;
16 }
```

确认重传——超时重传直到上限

确认重传类似rdt3.0，即不对出错接收进行相应，等待发送方超时重传解决出错问题；在序列号和确认号设置上类似TCP协议，即采用“流式”不间断地确认发送和接收的数据量。

数据的传输过程，理解为一次完整的收发和随后的数据包解析过程，通过对收发过程进行超时重传直到上限的控制，实现发送方向接收方的可靠数据传输。

在该部分表述中按如下约定：

1. 以 `self.seq_num` 和 `self.ack_num` 分别表示服务器或客户端自身的序列号和确认号
2. 以 `recv.seq_num` 和 `recv.ack_num` 表示接收数据包中的序列号和确认号字段
3. 以 `send_data_size` 和 `recv_data_size` 分别表示发送和接收的数据大小

该部分主要代码如下：

```
1  bool stop_and_wait_send_and_recv(unsigned char *data, unsigned short
    data_len, PacketManager::PacketType packet_type) {
2      seq += data_len;    // auto increment seq
3      ...//copy data to send buf
4
5      while(retry_cnt < retry_count) {
6          if (sendto(client_socket, send_buf, SEND_BUF_SIZE, 0, (SOCKADDR *)
            &server_addr, server_addr_len) == SOCKET_ERROR) {
7              ...//handling exception
8          }
9
10         if(retry_cnt == 0){
11             start = TIME_NOW;    // start timer
12         }
13
14         memset(recv_buf, 0, MSS);
15         while(true)
16         {
17             if (recvfrom(client_socket, recv_buf, MSS, 0, (SOCKADDR *)
                &server_addr, &server_addr_len) == SOCKET_ERROR) {
18                 if (TIME_OUT(start)) {    // timeout
19                     retry_cnt++;
20                     break;    // try for another send
21                 }
22                 else {
23                     ...// handling other exceptions
24                 }
25             }
26
27             recv_packet = (UDP_Packet *) recv_buf;
28             if (CORRUPTED(recv_packet)) {    // if corrupted, no answer
29                 if(!TIME_OUT(start)) continue;
30                 continue;
31             }
32             if (!ACK(recv_packet)) {    // if not ack or ack-mismatch, no
answer either
33                 if (!TIME_OUT(start)) continue;
34                 continue;
35             }
36             if(ACK_NUM(recv_packet) != seq) {
37                 if (!TIME_OUT(start)) continue;
38                 continue;
39             }
40
41             ack = SEQ_NUM(recv_packet) + DATA_LEN(recv_packet);    // auto
increment ack
42
43             return true;    // success
44         }
45     }
46     return false;    // retry_count times used up
```

序列号和确认号的设置

发送数据包后，`self.seq_num += send_data_size`，对应如上代码第2行

接收数据包后，`self.ack_num = recv.seq_num + recv_data_size`，对应如上代码第44行

这样，`self.seq_num` 记录了累计发送数据的长度，`self.ack_num` 记录了累计接收的数据长度

确认规则

检测 `self.seq_num == recv.ack_num`，即“我方发送的总长度”是否等于“对方接收的总长度”，对应如上代码第36行

出错动作——不作回应

接收到出错的数据包时，若此时没有超时，则不作任何回应。等待发送方超时重传，解决出错问题。对应第28-39行

超时动作——重传数据

当超时后，若重传次数未到上限，则退出接收循环，重新发送数据。对应如上代码第18-20行

如何判定超时呢？首先设定超时时限，将套接字接收设置为阻塞-超时模式



`sendto` 函数为非阻塞函数，将接收缓冲区交给下层IP协议进行发送；

`recvfrom` 函数默认为阻塞函数，通过如下设置，将使其阻塞特定时长，若超时则退出。

```
1 void Client::set_timeout(int sec, int usec) {
2     time_out = sec * 1000 + usec / 1000;
3     if(setsockopt(client_socket, SOL_SOCKET, SO_RCVTIMEO, (char*)&time_out,
4         sizeof(time_out)) == SOCKET_ERROR){
5         ...// handling exception
6     }
7 }
```

重传至限——报错退出

当重传达到上限，则认为接收端无法接收到，此时报错退出。对应如上代码第46行

状态转换

状态转换基于当前状态和对接收到数据包的解析结果。

服务器

服务器主要状态和动作如下：

1. LISTEN:

监听来自客户端的SYN连接请求，获取其中的约束信息（MSS大小、文件名和大小），同时转换到SYN_RCVD状态。

2. SYN_RCVD:

完成一次完整的数据传输过程：向客户端发送SYN_ACK数据包，对MSS大小进行最后确认；等待来自客户端的ACK（该数据包可同时负载文件数据，具有PSH属性）。若得到确认，则进入连接建立ESTABLISHED状态。

3. ESTABLISHED:

接收来自客户端的ACK_PSH数据包，并进行确认。由于是停等协议，因此每一次接收都可以交付，即将数据写入文件末端。文件下载完毕，若接收到客户端的FIN，则进入LAST_ACK状态，准备关闭连接，释放资源。

4. LAST_ACK:

完成一次完整的数据传输，向客户端发送ACK_FIN，确认其关闭请求，当收到客户端ACK后，释放资源，最终状态为CLOSED。

有限状态机转换精简代码如下：

```
1  void Server::start() {
2      while(true) {
3          switch (state) {
4              /*
5               * Listen state:
6               * Action: Wait for SYN packet from client -> Change state to
SYN_RCVD
7               * Note: The ultimate MSS(max_segment_size) is the minimum of
server's recv buffer size and client's max send buffer size
8               */
9              case LISTEN: {
10                 if (recvfrom(server_socket, recv_buf, RECV_BUF_SIZE, 0,
(SOCKADDR *) &client_addr, &client_addr_len) == SOCKET_ERROR) {
11                     ...// handle exception
12                 }
13                 recv_packet = (UDP_Packet *) recv_buf;
14                 if (CORRUPTED(recv_packet)) {
15                     ...// handle exception
16                     break;
17                 }
18                 if(!SYN(recv_packet)) {
19                     ...// handle exception
20                     break;
21                 }
22                 parse_recv_packet();
23                 state = SYN_RCVD;
24                 break;
25             }
26             /*
27              * SYN_RCVD state:
28              * Action: Send MSS to client -> Wait for ACK packet from client
-> Change state to ESTABLISHED
29              */
30             case SYN_RCVD: {
31                 set_timeout(2, 0); // set timeout to 2 sec
32                 set_retry_count(5);
33                 start = TIME_NOW;
34                 PLAIN_LOG(string("Global timer started"));
35
36                 if(!stop_and_wait_send_and_recv((unsigned char *) &MSS,
sizeof(MSS), PacketManager::SYN_ACK)) {
37                     ...// handle exception
38                 }
39                 parse_recv_packet();
40
41                 state = ESTABLISHED;
42                 STATE_LOG(string("Server state: ESTABLISHED"));
```

```

43         break;
44     }
45
46     /*
47     * ESTABLISHED state:
48     * Action: Return and start transaction
49     */
50     case ESTABLISHED: {
51         while(acc_rcv_size <= rcv_file_size && state != LAST_ACK) {
52             if(!stop_and_wait_send_and_rcv((u8*)&null_data, 1,
PacketManager::ACK)) {
53                 ...// handle exception
54             }
55             parse_rcv_packet();
56         }
57         state = LAST_ACK;
58         break;
59     }
60     /*
61     * LAST_ACK state:
62     * Action: Send FIN packet to client -> Wait for ACK packet from
client -> Change state to CLOSED
63     */
64     case LAST_ACK: {
65         if(!stop_and_wait_send_and_rcv((u8*)&null_data, 1,
FIN_ACK_TYPE)) {
66             ...// handle exception
67         }
68         state = CLOSED;
69         return;
70     }
71 }
72 }
73 }

```

客户端

客户端主要状态和动作为：

1. CLOSED:

主动提起连接请求，选定传输文件后，将自身最大发送缓冲区大小、文件信息发送给服务器，作为连接约定。之后进入SYN_SENT状态。

2. SYN_SENT:

接收服务器的SYN_ACK数据包，确认MSS大小，向服务器发送负载有数据的ACK数据包，进入连接建立状态。

3. ESTABLISHED:

持续完成数据传输。只有接收到服务器确认，才进行下一轮数据传输。文件传输完毕，接收到服务器确认后，进入FIN_WAIT_1状态，准备释放资源。

4. FIN_WAIT_1:

向服务器发送FIN数据包，接收到服务器确认后，进入FIN_WAIT_2状态。

5. FIN_WAIT_2:

向服务器发送断开确认，进入CLOSED状态。

有限状态机转换精简代码如下：

```
1 void Client::start() {
2     if(!choose_send_file(res_dir)){
3         ...// handle exception
4     }
5
6     while(true){
7         switch(state) {
8             /*
9              * Closed state:
10             * Action: Send SYN packet to server and wait for ack -> change
state to SYN_SENT
11             */
12             case(CLOSED):{
13                 File_Info* file_info = PacketManager::make_file_info(MSS,
send_file_size, send_file_name);
14                 if(!stop_and_wait_send_and_recv((u8*)file_info,
FILE_INFO_LEN(file_info), SYN_TYPE)){
15                     ...// handle exception
16                 }
17                 parse_rcv_packet();    // parse syn_ack packet
18                 state = SYN_SENT;
19                 break;
20             }
21             /*
22              * SYN_SENT state:
23             * Action: send ACK packet to server and wait for ack -> change
state to ESTABLISHED
24             */
25             case(SYN_SENT):{
26                 read_data_len = READ_DATA(&send_file, data_buf, MSS-
HEADER_SIZE);
27                 if(!stop_and_wait_send_and_recv((unsigned char*)data_buf,
read_data_len, PSH_ACK_TYPE)){
28                     ...// handle exception
29                 }
30                 state = ESTABLISHED;
31                 break;
32             }
33             /*
34              * Established state:
35              * return to continue transaction with server
36              */
37             case ESTABLISHED:{
38                 while(acc_sent_size < send_file_size){
39                     read_data_len = READ_DATA(&send_file, data_buf, MSS-
HEADER_SIZE);
40                     if(!stop_and_wait_send_and_recv((unsigned char*)data_buf,
read_data_len, PSH_ACK_TYPE)){
41                         ...// handle exception
42                     }
43                 }
44                 state = FIN_WAIT_1;
45                 break;
46             }
47             /*
48              * FIN_WAIT_1 state:
```



```

49         * Action: send FIN packet to server and wait for ack -> change
state to FIN_WAIT_2
50         */
51         case FIN_WAIT_1:{
52             if(!stop_and_wait_send_and_recv((u8*)&null_data,
sizeof(null_data), FIN_ACK_TYPE)){
53                 ...// handle exception
54             }
55             parse_rcv_packet();    // parse FIN_ACK packet
56             state = FIN_WAIT_2;
57             break;
58         }
59         /*
60         * FIN_WAIT_2 state:
61         * Action: send ACK packet to server -> close
62         */
63         case FIN_WAIT_2:{
64             send_packet = MAKE_PKT(ACK_TYPE, seq, ack, (u8*)&null_data,
sizeof(null_data));
65             sendto(client_socket, (char*)send_packet,
TOTAL_LEN(send_packet), 0, (SOCKADDR*)&server_addr, server_addr_len);
66             Sleep(1000);    // sendto is non-blocking, sleep for 1s to
ensure packet is sent
67             state = CLOSED;
68             return;
69         }
70     }
71 }
72 }

```

流量控制——停等机制

如确认重传部分所述，将数据的传输过程，理解为一个完整的收发和随后的数据包解析过程，之后再启动下一次的传输。因此同一时刻，只有服务器或客户端在发送数据包，另一方处于等待接收状态。

特别需要注意的是超时时间的设定。详见问题记录部分的超时时间设定部分。最终得到结论：该停等协议只能保证一方的传输是可靠的，且其超时时间与重传次数的乘积，应当小于另一方的单次超时时间。

问题记录

1. Windows下设置 `recvfrom()` 为阻塞-超时模式失效

• 问题现象：

`recvfrom()` 函数为阻塞函数，想要通过如下设置，使其在超时时间内，若没有接收到数据，则返回退出。然而这使其直接返回，成了非阻塞函数。

```

1 struct timeval timeout;
2 timeout.tv_sec = 5;
3 timeout.tv_usec = 0;
4 if(setsockopt(server_socket, SOL_SOCKET, SO_RCVTIMEO, (char *)&timeout,
sizeof(timeout)) == SOCKET_ERROR) {
5     exit(EXIT_FAILURE);
6 }

```

• 问题原因：

Window下的时间通过 `DWORD` 表示的毫秒来传递，Linux下使用 `timeval` 结构体传递。这里实际上没有正确设置超时时间。



参考链接：[c - Set timeout for winsock recvfrom - Stack Overflow](#)

- 问题解决：

以如下方式进行设置，`recvfrom` 的阻塞-超时设置成功。但实际实验中，考虑到如果出现发送方传来的包一直出现比特错误导致校验失败的话，每次都会重新启动计时器，这样就会一直阻塞在接收处，超时重传将永远不会到来。实际编程中增加了对发送包时绝对时间的记录，最多经过两个超时间隔（第一个在超时间隔之前到来，但出错，超时时间未到，等待下一个包……最后一个包到来时，如果时间已经超过超时间隔，则请求重传）

```
1  DWORD time_out;
2  time_out = 5 * 1000;
3  if(setsockopt(server_socket, SOL_SOCKET, SO_RCVTIMEO, (char *)&time_out,
4  sizeof(time_out)) == SOCKET_ERROR) {
5      exit(EXIT_FAILURE);
6  }
```

这样的方法，好处是编程简单，可靠；坏处是超时时间会在一个到两个超时间隔内波动。之后考虑使用线程机制模拟中断（信号量）或直接调用中断（似乎可能因编译器，不一定成功，且可能需要编写硬件）

2. 超时重传次数和时间的设定

- 问题现象

在早期的实验过程中，设定服务器和客户端的超时时间均为1s，重传次数为3次，而一旦丢包却出现了很多接收包错误的情况，最终无法接收到完整的数据。

- 问题原因

当出现丢包后，发送方进行重传，此时另一方如果也处于超时重传状态，它们便相互发送数据。此时它们发送的数据包不会相互匹配（对方发送的数据要么领先当前一轮，或落后当前一轮），在校验时便会出现一个或多个校验和错误的情况。

因此必须保证，同一时刻只有一方在发送数据。在下一轮发送前，它必须接收到对方的数据，这也是“停等收发”函数主要解决的问题。

- 问题解决

将一次数据传输的收发视为完整过程，发送方只有接收到确认数据包后才能进行下一轮传输；或者超时重传达到上限，报错退出。因此必须保证，主要数据发送方（此实验中为客户端）的总超时时间（单次超时时间与超时次数乘积）小于主要数据接收方（此实验中为服务器）的单次超时时间，这样就能完全避免同时发送数据的情况。详细实现见协议设计的确认重传部分解决。

3. Router程序设置问题

路由程序作为服务器与客户端的中转，正确的设定是，将Router作为客户端的远程服务器，Router中的服务器设定为服务器的端口号和地址。

4. 性能测量

由于时间问题，没有对程序在不同MSS、时延、丢包率的情况下进行测量，并分析其中关系。这部分工作留作后期继续完成。

程序演示

基本日志

- 初始化

```
18:16:22:138 [PLAIN] Server params set
18:16:22:139 [PLAIN] Timeout set to 200 ms
18:16:22:140 [PLAIN] Retry count set to 5
18:16:22:140 [PLAIN] Reset
```

初始化日志

```
1 1.jpg
2 2.jpg
3 3.jpg
4 helloworld.txt
Choose a file to send:2
File chosen: 2.jpg
File size: 5898505
```

资源文件选择
打印选中文件信息

```
18:16:50:15 [STATE] Client state: CLOSED
18:16:50:16 [PLAIN] SND_PKT PKT: seq_num: 0, ack_num: 0, data_len: 13, checksum: 14214, flags: SYN,
18:16:50:69 [PLAIN] RCV_ACK PKT: seq_num: 0, ack_num: 13, data_len: 2, checksum: 55277, flags: SYN,ACK,
18:16:50:70 [PLAIN] Parsing SYN packet
18:16:50:71 [PLAIN] MSS set to 10240
```

状态转换日志

数据包信息日志

动作日志

- 连接建立

```
File size: 5898505
18:16:50:15 [STATE] Client state: CLOSED
18:16:50:16 [PLAIN] SND_PKT PKT: seq_num: 0, ack_num: 0, data_len: 13, checksum: 14214, flags: SYN,
18:16:50:69 [PLAIN] RCV_ACK PKT: seq_num: 0, ack_num: 13, data_len: 2, checksum: 55277, flags: SYN,ACK,
18:16:50:70 [PLAIN] Parsing SYN packet
18:16:50:71 [PLAIN] MSS set to 10240
18:16:50:71 [STATE] Client state: SYN_SENT
18:16:50:73 [PLAIN] SND_PKT PKT: seq_num: 13, ack_num: 2, data_len: 10226, checksum: 64821, flags: ACK,PSH,
18:16:50:117 [PLAIN] RCV_ACK PKT: seq_num: 2, ack_num: 10239, data_len: 1, checksum: 55290, flags: ACK,
18:16:50:119 [PLAIN] Sent 10226 bytes
18:16:50:119 [STATE] Client state: ESTABLISHED
```


- 连接断开

以客户端为例，展示

```
18:17:29:927 [PLAIN] Sent 8329 bytes
18:17:29:928 [PLAIN] File 2.jpg sent
18:17:29:928 [STATE] Client state: FIN_WAIT_1
18:17:29:928 [PLAIN] SND_PKT PKT: seq_num: 5898518, ack_num: 579, data_len: 1, checksum: 64580, flags: ACK,FIN,
18:17:29:994 [PLAIN] RCV_ACK PKT: seq_num: 579, ack_num: 5898519, data_len: 1, checksum: 64579, flags: ACK,FIN,
18:17:29:994 [PLAIN] Parsing FIN packet
18:17:29:995 [STATE] Client state: FIN_WAIT_2
18:17:31:5 [PLAIN] SND_PKT PKT: seq_num: 5898519, ack_num: 580, data_len: 1, checksum: 64582, flags: ACK,
18:17:31:8 [STATE] Client state: CLOSED
```

- 传输验证

设定MSS为10240，在如下的丢包率和时延下进行测试。

 Router

路由器IP: 127 . 0 . 0 . 200

服务器IP: 127 . 0 . 0 . 100

端口: 9999

服务器端口: 8888

丢包率: 1 %

延时: 10 ms

确定

修改

日志

count:78.
Delay 10 ms.
count:79.
Delay 10 ms.
count:80.
Delay 10 ms.
count:81.
Delay 10 ms.
count:82.

客户端最终日志为:

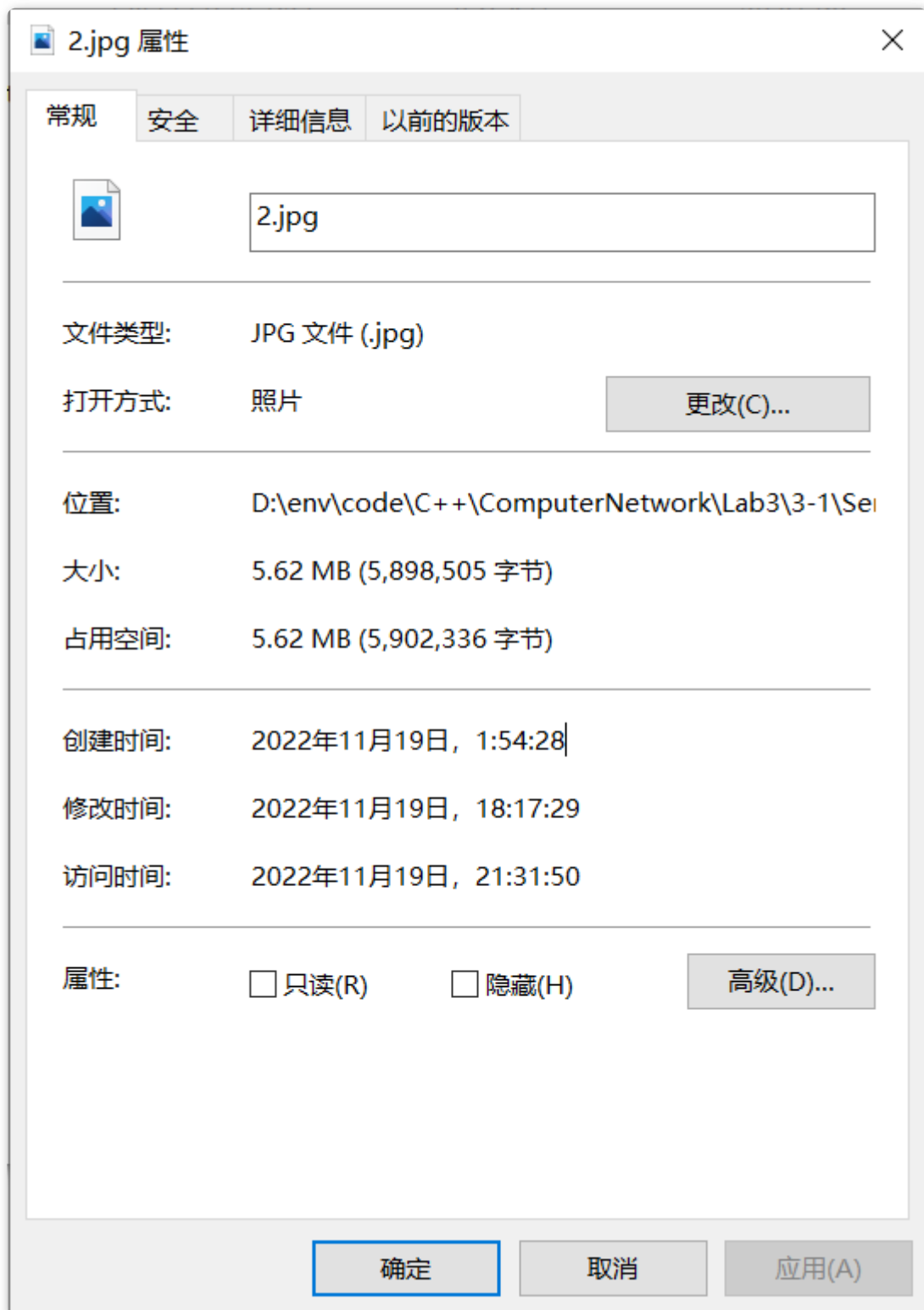
```
=====
File 2.jpg sent
Total bytes sent: 5898505
Duration: 40.993 s
Bandwidth: 140.518 KB/s
Total packets corrupted made: 5
Total Packets sent: 590
Total Packets resent: 11
=====
```

服务器最终日志为:

```
=====
File 2.jpg downloaded
Total bytes received: 5898505
Duration: 39.9198 s
Bandwidth: 144.296 KB/s
Total packets corrupted: 5
=====
```

传输结果：

显然客户端发送数据和服务器接收到的数据相同，出错包一致，重传数大致满足丢包条件。最终文件大小与原始文件一致。



可靠传输验证——校验&超时重传

对出错包的处理

设定出错率，达到累计的传包次数时，对包的校验和字段进行随机替换（这样确认时不会得到16位全1结果）；否则在重传时，还原数据包的校验和字段。主要代码如下：

```
while(retry_cnt < retry_count) {
#ifdef DEBUG_CORRUPT
    if (corrupt_count == CORRUPT_RATE - 1) {
        send_packet = (UDP_Packet*)send_buf;
        send_packet->header.checksum = rand() % 256;
        acc_corrupt++;
        PLAIN_LOG(string(s: "Made corrupted packet"));
    }
    if(corrupt_count == 0) {
        send_packet->header.checksum = checksum_backup;
    }
    corrupt_count = ++corrupt_count % CORRUPT_RATE;
#endif
    acc_sent++;
    if (sendto(s: client_socket, buf: send_buf, len: SEND_BUF_SIZE, flags: 0, to: (SOCKADDR *) &server_addr, tolen: server_addr_len) == SOCKET_ERROR) {
        EXCPT_LOG(string(s: "sendto() failed with error code : ") + to_string(val: WSAGetLastError()));
        exit( Code: EXIT_FAILURE);
    }
}
```

打印得到的日志：

- 客户端（发送方）

下图为制作错包，服务器不作回应，引起客户端超时重传的日志

```
18:16:57:36 [PLAIN] SND_PKT PKT: seq_num: 961257, ack_num: 96, data_len: 10226, checksum: 10137
18:16:57:115 [PLAIN] RCV_ACK PKT: seq_num: 96, ack_num: 971483, data_len: 1, checksum: 11442, f
18:16:57:116 [PLAIN] Sent 10226 bytes
18:16:57:118 [PLAIN] SND_PKT PKT: seq_num: 971483, ack_num: 97, data_len: 10226, checksum: 3631
18:16:57:203 [PLAIN] RCV_ACK PKT: seq_num: 97, ack_num: 981709, data_len: 1, checksum: 1215, fl
18:16:57:205 [PLAIN] Sent 10226 bytes
18:16:57:207 [PLAIN] SND_PKT PKT: seq_num: 981709, ack_num: 98, data_len: 10226, checksum: 7619
18:16:57:323 [PLAIN] RCV_ACK PKT: seq_num: 98, ack_num: 991935, data_len: 1, checksum: 56523, f
18:16:57:324 [PLAIN] Sent 10226 bytes
18:16:57:325 [PLAIN] SND_PKT PKT: seq_num: 991935, ack_num: 99, data_len: 10226, checksum: 5814
18:16:57:325 [PLAIN] Made corrupted packet
18:16:57:542 [EXCPT] recvfrom() timeout: resend packet
18:16:57:633 [PLAIN] RCV_ACK PKT: seq_num: 99, ack_num: 1002161, data_len: 1, checksum: 46296,
18:16:57:634 [PLAIN] Sent 10226 bytes
18:16:57:635 [PLAIN] SND_PKT PKT: seq_num: 1002161, ack_num: 100, data_len: 10226, checksum: 44
```

制作错包，服务器不回应
超时重传

客户端制作错包统计

```

18:17:29:774 [PLAIN] Parsing FIN packet
18:17:29:995 [STATE] Client state: FIN_WAIT_2
18:17:31:5 [PLAIN] SND_PKT: seq_num: 5898519, ack
18:17:31:8 [STATE] Client state: CLOSED
=====
File 2.jpg sent
Total bytes sent: 5898505
Duration: 40.993 s
Bandwidth: 140.518 KB/s
Total packets corrupted made: 5
Total Packets sent: 590
Total Packets resent: 11
=====

```

累计制作了5个错包

- 服务器（接收方）

下图为服务器接收到错包且不回应的日志

```

corrupted
18:17:24:225 [PLAIN] Downloaded 10226 bytes
18:17:24:225 [PLAIN] SND_PKT: PKT: seq_num: 488, ack_num: 4980075, data_len: 1, checksum: 93, flags: ACK,
18:17:24:286 [PLAIN] RCV_PKT: PKT: seq_num: 4980075, ack_num: 489, data_len: 10226, checksum: 29938, flags: ACK,PS
18:17:24:288 [PLAIN] Parsing PSH packet
18:17:24:288 [PLAIN] Downloaded 10226 bytes
18:17:24:289 [PLAIN] SND_PKT: PKT: seq_num: 489, ack_num: 4990301, data_len: 1, checksum: 55401, flags: ACK,
18:17:24:356 [PLAIN] RCV_PKT: PKT: seq_num: 4990301, ack_num: 490, data_len: 10226, checksum: 42490, flags: ACK,PS
18:17:24:359 [PLAIN] Parsing PSH packet
18:17:24:359 [PLAIN] Downloaded 10226 bytes
18:17:24:360 [PLAIN] SND_PKT: PKT: seq_num: 490, ack_num: 5000527, data_len: 1, checksum: 45174, flags: ACK,
18:17:24:432 [EXCPT] Corrupted packet received 接收到错包
18:17:24:635 [PLAIN] RCV_PKT: PKT: seq_num: 5000527, ack_num: 491, data_len: 10226, checksum: 28368, flags: ACK,PS
18:17:24:639 [PLAIN] Parsing PSH packet
18:17:24:640 [PLAIN] Downloaded 10226 bytes

```

服务器最终统计的错包数量与客户端制作错包数量一致

```

=====
File 2.jpg downloaded
Total bytes received: 5898505
Duration: 39.9198 s
Bandwidth: 144.296 KB/s
Total packets corrupted: 5
=====

```

弱网条件丢包下的超时重传

下图为客户端在路由程序执行丢包操作后，打印的日志

```
Client x
Q timeout 2/12
18:16:50:260 [PLAIN] SND_PKT PKT: seq_num: 40917, ack_num: 6, data_len: 10226, checksum: 23892, flags: ACK,PSH,
18:16:50:311 [PLAIN] RCV_ACK PKT: seq_num: 6, ack_num: 51143, data_len: 1, checksum: 14382, flags: ACK,
18:16:50:312 [PLAIN] Sent 10226 bytes
18:16:50:314 [PLAIN] SND_PKT PKT: seq_num: 51143, ack_num: 7, data_len: 10226, checksum: 40622, flags: ACK,PSH,
18:16:50:377 [PLAIN] RCV_ACK PKT: seq_num: 7, ack_num: 61369, data_len: 1, checksum: 4155, flags: ACK,
18:16:50:377 [PLAIN] Sent 10226 bytes
18:16:50:378 [PLAIN] SND_PKT PKT: seq_num: 61369, ack_num: 8, data_len: 10226, checksum: 36660, flags: ACK,PSH,
18:16:50:426 [PLAIN] RCV_ACK PKT: seq_num: 8, ack_num: 71595, data_len: 1, checksum: 59463, flags: ACK,
18:16:50:426 [PLAIN] Sent 10226 bytes
18:16:50:427 [PLAIN] SND_PKT PKT: seq_num: 71595, ack_num: 9, data_len: 10226, checksum: 5883, flags: ACK,PSH,
18:16:50:636 [EXCP] recvfrom() timeout: resend packet
18:16:50:687 [PLAIN] RCV_ACK PKT: seq_num: 9, ack_num: 81821, data_len: 1, checksum: 49236, flags: ACK,
```

接收确认

发送数据包

未接收到回应，超时重传

实验环境

CLion2022 , C++17, MinGW, Windows10