

计算机网络lab3-1

1 【实验内容】

2 【协议设计与C++编程实现】

2.1 协议概述与UDP_DATAGRAM类的设计

2.2 rdt3.0逻辑实现（程序整体框架设计介绍）

2.2.1 发送端sendandwait()函数

2.2.2 计时器timerLogic()线程

2.2.3 接收端RCACK()

2.3 校验和差错检测

2.4 状态位检测与文件传输协议交互过程

2.3.1 在连接建立过程中

2.3.2 在传输”文件头报文“过程中

2.3.3 在传输文件数据报文过程中

2.3.4 在连接断开过程中

3 【程序测试结果】

3.1 传输文件测试

3.2 停等协议模拟之差错测试

姓名：罗瑞

学号：2210529

专业：密码科学与技术

1 【实验内容】



实验3-1：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。

✓ 本实验我实现的是rdt3.0协议：

`rdt3.0` 是一种面向连接的可靠数据传输协议，采用**停等（Stop-and-Wait）**机制来保证数据的可靠传输。它通过一系列的特性来确保数据从发送端到接收端的正确性和完整性。首先，协议实现了**建立连接**和**断开连接**的功能，确保通信双方能够在开始和结束时建立明确的连接状态。为了防止数据在传输过程中发生错误，`rdt3.0`引入了**差错检测**机制，通过校验和等方法确保数据的完整性。同时，为了应对丢包和网络延迟，它采用**差错重传**和**超时重传**机制，如果发送的数据在规定时间内没有收到确认，发送端会重传该数据包。

2【协议设计与C++编程实现】

2.1协议概述与UDP_DATAGRAM类的设计

✓ 本实验要求基于数据包套接字（UDP）来实现可靠传输协议，因此我在设计UDP数据报的数据结构时，在原UDP类中增加STATE变量，用来模拟实现可靠传输的建立和断开（这里实现的是三次握手、挥手方法），其中注意UDP的校验和计算还是完全按照原有UDP数据类型来进行的。

| | | | | |
|-------|--------------|----------|----------|----------|
| 伪首部 | 153.19.8.104 | | | |
| | 171.3.14.11 | | | |
| | 0 | 17 | 15 | |
| UDP首部 | 1087 | | 13 | |
| | 15 | | 0 | |
| 数据 | 01010100 | 01000101 | 01010011 | 01010100 |
| | 01001001 | 01001110 | 01000111 | 0填充 |

图1-1：UDP报文数据字段示例

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--------------------------------|---|---|---|---|---|---|---|--------------|---|---|---|---|---|---|---|-------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|-----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 源IP地址（Source IP address） | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 目的IP地址（Destination IP address） | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | | | | | | | | 协议（Protocol） | | | | | | | | 长度（Length） | | | | | | | | | | | | | | | |
| 源端口号（Source Port） | | | | | | | | | | | | | | | | 目的端口号（Destination Port） | | | | | | | | | | | | | | | |
| 长度（Length） | | | | | | | | | | | | | | | | 校验和（Checksum） | | | | | | | | | | | | | | | |
| 数据（Data） | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0填充 | |

图1-2：UDP报文格式

`UDP_DATAGRAM` 报文格式如图所示（我的UDP具体协议设置）：

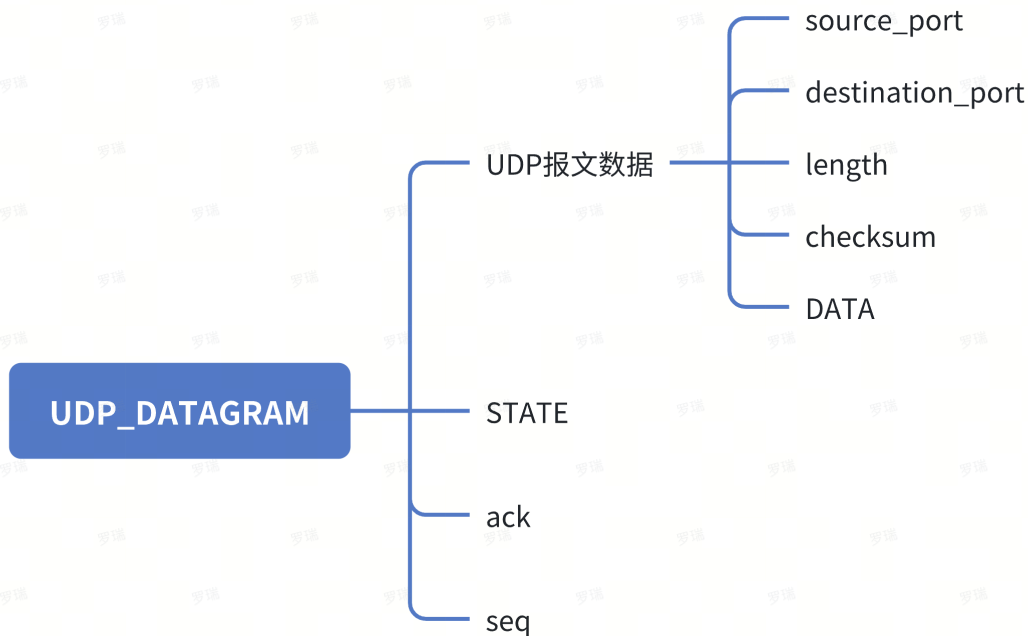


图2: UDP_DATAGRAM 数据组成设计

具体的字段设置和部分关键代码如下：

✓ 1. `unsigned short source_port;` 和 `unsigned short destination_port;`

- 类型: `unsigned short`
- 说明: 这两个成员变量分别表示 UDP 数据报的源端口和目标端口。端口号在 UDP 协议中是必需的，用于标识发送端和接收端的通信端点。由于端口号范围在 0 到 65535 之间，选择 `unsigned short` 类型（16 位无符号整数）是合理的。

2. `unsigned short length;`

- 类型: `unsigned short`
- 说明: 表示 UDP 数据报的总长度（以字节为单位）。该字段包含了 UDP 数据报的头部和数据部分的长度。由于 UDP 数据报长度最大为 65535 字节，使用 `unsigned short` 来表示这一长度是合适的。

3. `unsigned short checksum;`

- 类型: `unsigned short`
- 说明: 这是一个 16 位无符号整数，用于存储数据报的校验和。UDP 校验和用于验证数据在传输过程中是否发生损坏。校验和是 16 位的，因此选择 `unsigned short` 类型来存储。

4. `char message[UDP_DATAGRAM_SIZE];`

- **类型:** `char[UDP_DATAGRAM_SIZE]`
- **说明:** 这是一个字符数组，存储 UDP 数据报的有效载荷部分，即实际的数据内容。
`UDP_DATAGRAM_SIZE` 是一个常量，指定了消息的最大长度。这个数组的大小可以根据实际需求进行调整，通常应该小于或等于最大传输单元（MTU）。由于 `char` 类型占用一个字节，所以它适用于存储字符串或二进制数据。

5. `unsigned short state;`

- **类型:** `unsigned short`
- **说明:** 用于表示数据报的状态。这个字段可以帮助管理数据的发送和接收状态，支持如“已发送”、“已确认”等不同状态的标记。在可靠传输协议中，`state` 字段可以用来跟踪数据报的状态，保证数据的可靠传输。这一部分详细内容解释详见2.4节

6. `unsigned short seq, ack;`

- **类型:** `unsigned short`
- **说明:** 分别用于表示数据报的序列号（`seq`）和确认号（`ack`）。序列号在传输中用于标识数据报的顺序，确认号用于确认接收到的数据报。`unsigned short` 类型足够容纳 16 位的序列号和确认号，符合 UDP 协议的需求。

其中 `UDP_DATAGRAM_SIZE` 的大小为 8×1024 ，而我在程序设计中用 `unsigned short` 类型的数据来存储 UDP 报文的个数，因此我的协议**一次最多支持传输0.5GB**大小的文件。

```
1 class UDP_DATAGRAM {      // UDP首部字段
2     unsigned short source_port;    // 源端口
3     unsigned short destination_port; // 目标端口
4     unsigned short length;        // 数据报总长度
5     unsigned short checksum;      // 校验和
6
7     // UDP数据部分
8     char message[UDP_DATAGRAM_SIZE]; // 消息内容
9
10    // 可靠传输相关字段
11    unsigned short state; // 数据报状态
12    unsigned short seq;   // 序列号
13    unsigned short ack;   // 确认号
14
15    public:
16        .....成员函数方法等
17    };
```

2.2 rdt3.0逻辑实现（程序整体框架设计介绍）

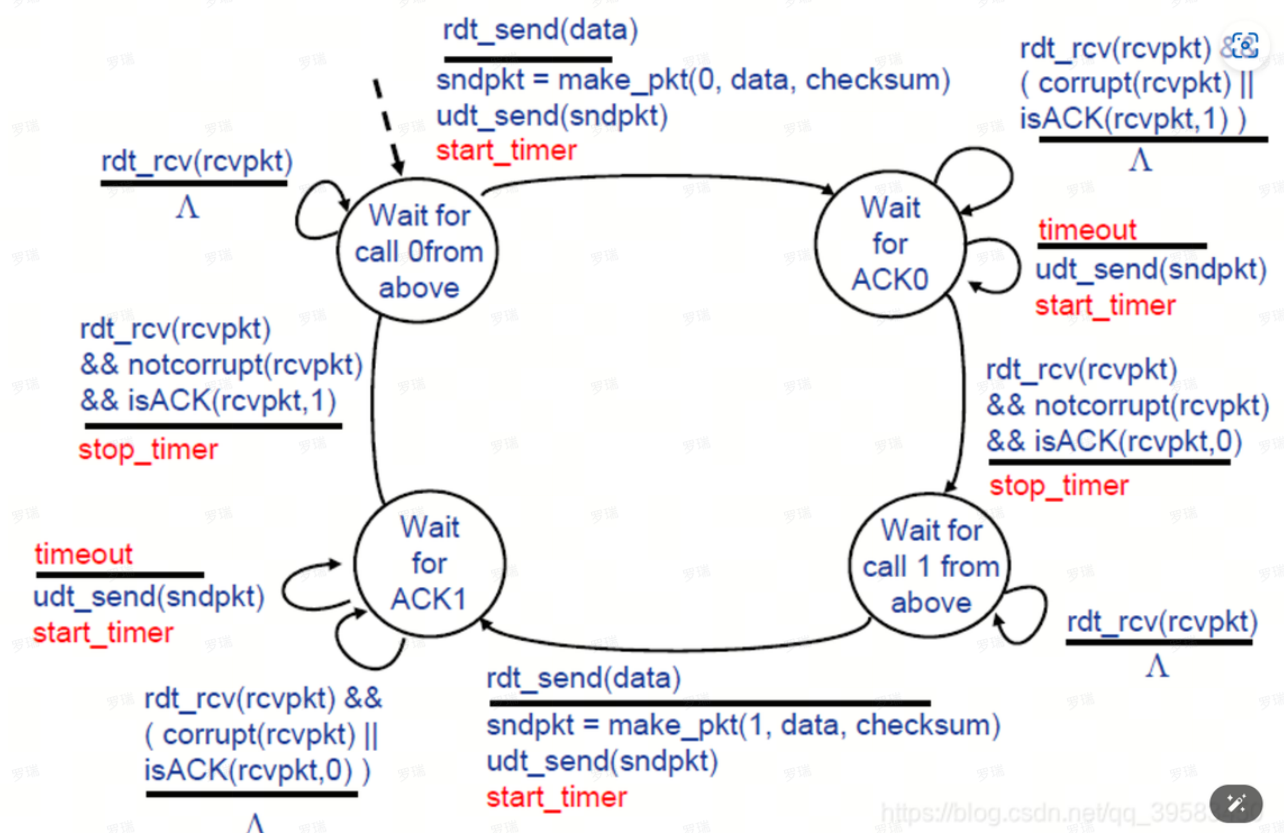


图3：rdt3.0协议



rdt3.0 协议的工作流程大致如下：

1. **数据传输开始**：发送端发送第一个数据包，接收端检查数据包是否有错误。
2. **接收端响应**：
 - 如果接收端收到的数据包没有错误，它会返回一个确认消息（ACK），通知发送端该数据包已成功接收。
 - 如果接收到的数据包有错误，接收端会丢弃该数据包并不会发送ACK，发送端会在超时后重发数据包。
3. **超时检测**：发送端在发送数据包后会启动定时器，等待接收端的ACK。如果在定时器到期前未收到确认，发送端会重新发送该数据包。
4. **数据接收完毕**：当所有数据包都被成功接收并确认后，接收端发送一个特殊的“断开连接”信号，通知发送端数据传输完毕。

我的代码中，实现rdt3.0的关键函数为发送端的sendandwait()以及接收端的RCACK()函数和设置计时器线程来完成差错重传和超时重传。

2.2.1 发送端sendandwait()函数

此函数是在一次UDP数据包发送函数中调用的，即每一次发送都进行停等控制。在外层函数：sendfiledata中，我开启计时器线程，在具体的一次UDP数据报发送中，我重置计时器参数，开始计时。当超时接收端未回复时，则触发重传逻辑（详见2.2.2）。

```
1  /* 停等协议 */
2  bool sendAndwait(UDP_DATAGRAM& datagram, UDP_DATAGRAM& recv_datagram, SOCKET
   sock, SOCKADDR_IN ServerAddr, SOCKADDR_IN ClientAddr) {
3      // 发送数据报
4      sendDatagram(datagram, sock, ServerAddr);
5      datagram.showseq();
6
7      // 启动计时器
8      LARGE_INTEGER clockStart, clockEnd;
9      QueryPerformanceCounter(&clockStart);
10     timeStart = clockStart;
11     isTimerActive = true;
12
13     // 保存全局状态变量以供超时处理
14     datagram0 = datagram;
15     sock0 = sock;
16     ServerAddr0 = ServerAddr;
17     times0 = 0;
18
19     std::cout << "停等协议开始..." << std::endl;
20
21     while (true) {
22         // 接收数据报
23         recvDatagram(recv_datagram, sock, ServerAddr);
24
25         // 处理差错重传
26         if (recv_datagram.getstate() & REQ) {
27             std::cout << "[Info] 差错检测，准备重传数据报..." << std::endl;
28
29             // 重传数据报
30             sendDatagram(datagram, sock, ServerAddr);
31             datagram.showseq();
32
33             // 重置计时器
34             QueryPerformanceCounter(&timeStart);
35             continue; // 继续等待响应
36         }
37
38         // 检查是否收到正确的ACK
39         if (recv_datagram.getack() == datagram.getseq()) {
40             std::cout << "[Success] 此UDP报文发送且接收成功!" << std::endl;
41         }
```

```

42         // 停止计时器
43         isTimerActive = false;
44         return true; // 传输成功
45     }
46
47     // 处理其他异常情况 (可扩展)
48     std::cout << "[Warning] 未知响应状态, 等待中..." << std::endl;
49 }
50
51 std::cerr << "[Error] 发送失败" << std::endl;
52 return false;
53 }
54

```

在此函数中，始终等待来自接收端服务器的答复，**当接收端服务器检测到差错（checksum校验和计算）时，会返回给sendandwait()一个标志位REQ的报文，被sendandwait()检测到时就会触发差错重传。**

2.2.2 计时器timerLogic()线程

```

1  // 初始化定时器
2  void initializeTimer() {
3      QueryPerformanceFrequency(&frequency);
4      QueryPerformanceCounter(&timeStart);
5  }
6
7  // 打印时间信息
8  void printElapsedTime(double elapsedTime) {
9      if (elapsedTime > elapsedTicks) {
10         std::cout << "Elapsed time: " << elapsedTicks << "s\n";
11         elapsedTicks++;
12     }
13 }
14
15 static bool isTimeout = false; // 超时标志
16 static bool isTimerActive = true; // 计时器是否激活
17 static int elapsedTicks = 1; // 用于记录并打印经过的时间
18 static int resendCount = 0; // 记录重传次数
19 static const int TIMEOUT_LIMIT = 5; // 超时重传限制次数
20
21 // 定时器控制标志
22 static bool isTimerRunning = true;
23
24 // 模拟的外部资源
25 UDP_DATAGRAM datagram0;

```



```

26 SOCKET sock0;
27 SOCKADDR_IN ServerAddr0;
28
29 // 重置计时器
30 void resetTimer() {
31     QueryPerformanceCounter(&timeStart);
32     elapsedTicks = 1;
33 }
34
35 // 计时器逻辑
36 void timerLogic() {
37     while (isTimerRunning) {
38         if (!isTimerActive) continue; // 如果计时器未激活, 跳过逻辑
39
40         // 获取当前时间
41         QueryPerformanceCounter(&timeEnd);
42         double elapsedTime = static_cast<double>(timeEnd.QuadPart -
timeStart.QuadPart) / frequency.QuadPart;
43
44         // 打印时间信息
45         printElapsedTime(elapsedTime);
46
47         // 检查是否超时
48         if (elapsedTime >= TIMELIMIT) {
49             isTimeout = true;
50             std::cout << "-----超时-----" << std::endl;
51
52             // 调用超时处理函数
53             handleTimeout(datagram0, sock0, ServerAddr0, timeStart,
resendCount);
54
55             // 重置计时器
56             resetTimer();
57
58             // 检查是否达到重传限制
59             if (++resendCount >= TIMEOUT_LIMIT) {
60                 std::cout << "-----重传超限-----" <<
std::endl;
61                 exit(EXIT_FAILURE); // 程序退出
62             }
63         }
64     }
65 }

```

有了计时器, 当我检测到在一次UDP发送: 停等过程超时, 会调用handleTimeout(datagram0, sock0, ServerAddr0, timeStart, resendCount);函数, 进行重新发送。这个函数就是执行

sendDatagram(datagram, sock, ServerAddr)重新发送报文。

2.2.3 接收端RCACK()

每当接收端接收到来自发送客户端的报文时，会先进行差错检测，若校验和正确，则返回标志位为ACK，否则返回的报文标志位为REQ。

```
1  /* 停等协议 RECV */
2  bool RCACK(UDP_DATAGRAM& datagram, SOCKET& sock, SOCKADDR_IN& ClientAddr) {
3      UDP_DATAGRAM ack;
4      memset(&ack, 0, sizeof(UDP_DATAGRAM));
5
6      while (true) {
7          // 接收数据报文
8          recvDatagram(datagram, sock, ClientAddr);
9          std::cout << "Received Seq: " << datagram.getseq() << " | Expected
10         Seq: " << sendseq << std::endl;
11
12         // 校验和检测
13         if (!datagram.check_checksum()) {
14             std::cout << "[Error] 校验和错误\n";
15
16             // 构造请求重传的ACK报文
17             ack.setstate(REQ);
18             ack.setseq(datagram.getseq());
19             sendDatagram(ack, sock, ClientAddr);
20
21             std::cout << "发送请求重传的ACK报文，序号： ";
22             ack.showseq();
23             std::cout << "数据报长度： " << datagram.getlength() << std::endl;
24
25             // 清空报文数据
26             memset(&ack, 0, sizeof(UDP_DATAGRAM));
27             memset(&datagram, 0, sizeof(UDP_DATAGRAM));
28             continue; // 继续等待接收
29         }
30
31         // 检查序列号是否匹配
32         if (datagram.getseq() == sendseq) {
33             std::cout << "[Info] 序列号匹配\n";
34
35             // 构造确认ACK报文
36             ack.setack(datagram.getseq());
37             ack.setstate(ACK);
38             ack.setseq(sendseq);
```

```

38         sendDatagram(ack, sock, ClientAddr);
39
40         // 更新发送序列号并返回成功
41         sendseq++;
42         std::cout << "发送ACK确认报文, 序号: ";
43         ack.showseq();
44         return true;
45     }
46
47     // 序列号不同步的情况
48     std::cout << "[Error] 序列号不同步, 当前报文序号: " << datagram.getseq()
49         << ", 期望序号: " << sendseq << std::endl;
50     break; // 退出循环
51 }
52
53 return false; // 接收失败
54 }

```

这里对于停等协议的差错检测是关键。在停等协议中，`RCACK` 函数通过校验和检测 (`check_checksum`) 实现差错检测机制。**如果接收到的数据报文校验和正确（不正确），函数会构造一个请求重传的 ACK 报文 (REQ 状态) 并发送给发送方，通知其重新传输相应的数据报。**这种机制确保了传输的可靠性，即使数据报在传输过程中由于网络噪声或其他问题发生损坏，协议仍然能够通过重传机制恢复正确的数据。

2.3 校验和差错检测



UDP校验和检测步骤

构造伪首部：

- 伪首部包含源IP地址、目的IP地址、协议号（固定为17，表示UDP），以及UDP长度字段。
- 伪首部的作用是将IP地址信息加入校验，确保数据包传输过程中IP地址不会被错误修改。

累加伪首部：

- 将伪首部中的字段按照16位对齐进行逐字段累加。

累加UDP报文首部和数据：

- 将UDP报文的源端口、目的端口、长度、数据，以及校验和字段（初始为0）按16位对齐进行累加。注意数据的字节数必须是偶数。

溢出处理：

- 如果累加结果超过16位，将高位溢出部分加回到低位，直到结果小于或等于16位。

取反：

- 将累加结果按位取反，得到最终的校验和值。

校验步骤：

- 接收到UDP数据包后，重复以上步骤计算校验和。
- 校验和计算的结果如果为全1（0xFFFF），表示数据无差错；否则，表示数据存在差错。

这里实现则对应是具体的关键类中函数：**calculate_checksum()**：设置UDP校验位与**check_checksum()**：验证校验位。由于我设置传输的最大UDP报文数不会超过unsigned short的大小，因此最终直接进行循环进位的逻辑不会出现超限等问题。可以直接在最后这样循环进位：

`while (sum >> 16) {sum = (sum & 0xFFFF) + (sum >> 16);}`。

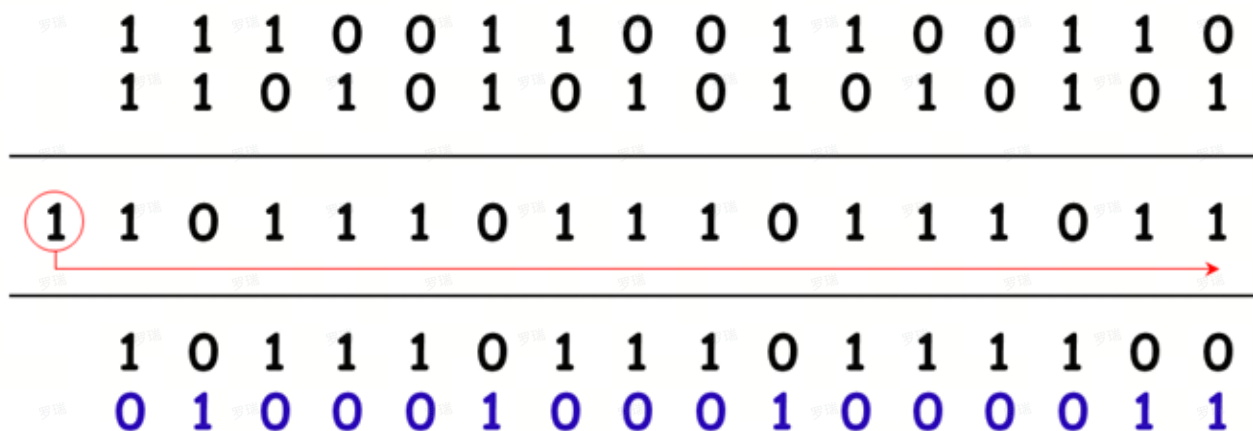


图4：校验和检验示意：2字节数据对齐

其中，发送端在发送报文前执行 `calculate_checksum` 函数来置校验位，接收端接收报文后执行 `check_checksum` 函数来验证UDP报文的校验和是否正确。



. `calculate_checksum` 函数：

- 目的：计算UDP数据报的校验和，并存储在 `checksum` 字段中。
- 步骤：
 - a. 初始化伪首部：伪首部包含源IP、目的IP、协议类型和数据长度等字段，这些字段对于计算校验和是必需的。
 - b. 累加伪首部：逐字段（按16位）将伪首部的各个字段累加到一个 `sum` 变量中。
 - c. 累加UDP报文头和数据：将UDP报文头部和数据（按16位）逐字段累加到 `sum` 中。
 - d. 溢出处理：由于UDP校验和的计算可能会导致溢出，使用16位的“进位”处理，确保最终的 `sum` 值在16位范围内。

e. **取反**：对累加后的 `sum` 取反，得到校验和。

2. `check_checksum` 函数：

- **目的**：验证UDP报文的校验和是否正确。
- **步骤**：
 - a. **初始化伪首部**：与 `calculate_checksum` 相同，构建伪首部并累加各字段。
 - b. **累加UDP报文头和数据**：与计算校验和时一样，按16位累加UDP报文的头和数据。
 - c. **溢出处理**：同样进行16位溢出处理。
 - d. **校验**：检查最终的 `sum` 是否等于 `0xFFFF`（表示校验正确）。如果是，返回 `true`，否则返回 `false`，并输出校验错误信息。

总结：

- `calculate_checksum` 用于计算和设置UDP报文的校验和。
- `check_checksum` 用于验证接收到的UDP报文的校验和是否正确。
- 两者通过伪首部和UDP报文的字段按16位逐一累加，计算出校验和，确保数据在传输过程中未被损坏。

具体实现关键部分代码如下：

UDP伪首部(4+4+ 1+1(合并) +2)

```
1 struct pseudoheader {
2 public:
3     unsigned int source_ip;
4     unsigned int destination_ip;
5     unsigned short con;
6     unsigned short length;
7 };
```

`calculate_checksum` 函数和 `check_checksum` 函数：

```
1 //设置UDP校验位
2 void calculate_checksum() {
3     //进行溢出操作，因此sum变量要大于两字节
4     unsigned int sum = 0;
5     //在此函数中完成port字段与伪首部字段初始化
6     this->source_port = CLIENTPORT;
7     this->destination_port = SERVERPORT;
```

```

8      struct pseudoheader* udp_header = new pseudoheader();
9      udp_header->source_ip = stringtoint(IPClient);
10     udp_header->destination_ip = stringtoint(IPServer);
11     udp_header->con = 17;
12     udp_header->length = this->getlength();
13     //求和(两字节对齐相加)
14
15     // 逐字段累加伪首部(两字节对齐相加)
16     sum += (udp_header->source_ip >> 16) & 0xFFFF;
17     sum += udp_header->source_ip & 0xFFFF;
18
19     sum += (udp_header->destination_ip >> 16) & 0xFFFF;
20     sum += udp_header->destination_ip & 0xFFFF;
21
22     sum += udp_header->con;
23     sum += udp_header->length;
24
25     //逐字段累加 UDP 报文首部以及数据(两字节对齐相加)
26     unsigned short* point = (unsigned short*)this;
27     int udp_header_length = sizeof(*this) / 2;
28     for (int i = 0; i < udp_header_length; i++) {
29         sum += *point++;
30     }
31
32     // 溢出处理
33     while (sum >> 16) {
34         sum = (sum & 0xFFFF) + (sum >> 16);
35     }
36     // 取反
37     this->checksum = ~((unsigned short)sum);
38
39 };
40 //校验UDP校验位
41 bool check_checksum() {
42     //进行溢出操作, 因此sum变量要大于两字节
43     unsigned int sum = 0;
44     //在此函数中依然要设置伪首部字段
45     struct pseudoheader* udp_header = new pseudoheader();
46     udp_header->source_ip = stringtoint(IPClient);
47     udp_header->destination_ip = stringtoint(IPServer);
48     udp_header->con = 17;
49     udp_header->length = this->getlength();
50     //求和(两字节对齐相加)
51
52     // 逐字段累加伪首部(两字节对齐相加)
53     sum += (udp_header->source_ip >> 16) & 0xFFFF;
54     sum += udp_header->source_ip & 0xFFFF;

```

```

55
56     sum += (udp_header->destination_ip >> 16) & 0xFFFF;
57     sum += udp_header->destination_ip & 0xFFFF;
58
59     sum += udp_header->con;
60     sum += udp_header->length;
61
62     //逐字段累加 UDP 报文首部以及数据(两字节对齐相加)
63     unsigned short* point = (unsigned short*)this;
64     int udp_header_length = sizeof(*this) / 2;
65     for (int i = 0; i < udp_header_length; i++) {
66         sum += *point++;
67     }
68
69     // 溢出处理
70     while (sum >> 16) {
71         sum = (sum & 0xFFFF) + (sum >> 16);
72     }
73
74     // 检查是否全1
75     if (sum == 0xFFFF)
76     {
77         printf("%s", "校验正确\n");
78         return true;
79     }
80     printf("%s", "校验出错\n");
81     cout << sum;
82     return false;
83 }

```

另外需要注意的是，UDP校验和规则要求数据字节数量为偶数，这样才便于2字节对齐产生校验和。因此，在发送文件的最后一个尾部UDP数据报时，我加入字节奇偶性检验，并进行相应的调整。

```

1 // 处理非整数据
2 UDP_DATAGRAM last;
3 memset(&last, 0, sizeof(UDP_DATAGRAM));
4 last.setseq(++nowseq);
5 // 检查rest是否为奇数
6 if (rest % 2 != 0)
7 {
8     // rest 是奇数，需要增加一个全零字节
9     memcpy(last.getmessage(), fileContent + (num_udp * UDP_DATAGRAM_SIZE),
10 rest);
11     last.getmessage()[rest] = 0; // 添加一个全零字节
12     last.setlength(rest + 8 + 1); // 更新length字段，包括新增的字节

```

```

12 }
13 else
14 {
15     // 如果rest是偶数，直接复制数据
16     memcpy(last.getmessage(), fileContent + (num_udp * UDP_DATAGRAM_SIZE),
17            rest);
18     last.setlength(rest + 8); // 设置正常的length
19 }
20 sendAndWait(last, respond, sock, ServerAddr, ClientAddr)
21 alive = 0;
22 t1.join();
23 return 1;
24

```

2.4 状态位检测与文件传输协议交互过程

为了描述报文的状态（或称为标志），我采用高效的位逻辑来标识。具体而言是我在代码中引入宏定义来清晰地标识逻辑与实际比特位之间的关系。这里实际上是四个bit位，当需要设置某个状态位的时候，只需将对应位置置1即可，反之亦然。

```

1 #define SYN 0x01
2 #define ACK 0x02
3 #define FIN 0x04
4 #define REQ 0x08

```

下面我将具体介绍程序中发送保温过程中的报文状态设置逻辑：

2.3.1 在连接建立过程中

本协议采用类三次握手（前两次握手基于简单地停等协议，ack直接等于seq）：

TCP建立连接过程

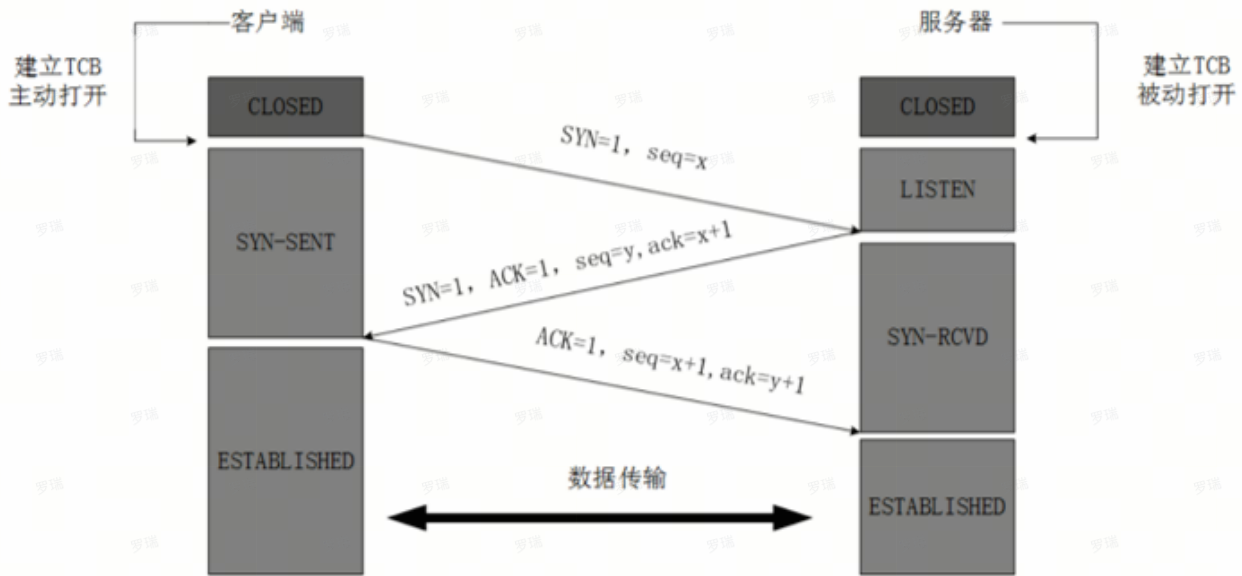


图5：三次握手逻辑



SYN位为1表示连接建立请求报文

SYN位为1且ACK位为1表示服务器建连回复确认

ACK位为1表示客户端建立连接确认

在客户端（发送者）建立连接函数中：

要设置发送报文SYN状态位时：`request.setstate(SYN);`

要设置发送报文ACK状态位时：`request.setstate(ACK);` 这样设置，其大大方便了代码可读性。

发送端连接建立的具体代码实现为：

```
1 // 发送并确认数据报函数封装
2 bool sendAndReceiveAck(UDP_DATAGRAM& request, UDP_DATAGRAM& response, SOCKET&
  sock, SOCKADDR_IN& ServerAddr, SOCKADDR_IN& ClientAddr)
3 {
4     if (!sendAndAck(request, response, sock, ServerAddr, ClientAddr)) {
5         return false; // 发送失败或确认失败
6     }
7     return true; // 成功发送并收到确认
8 }
9
10 // 第一次握手：发送SYN报文并等待ACK
```

```

11 bool firstHandshake(SOCKET& sock, SOCKADDR_IN& ServerAddr, SOCKADDR_IN&
    ClientAddr)
12 {
13     UDP_DATAGRAM request, response;
14     memset(&request, 0, sizeof(UDP_DATAGRAM));
15     memset(&response, 0, sizeof(UDP_DATAGRAM));
16
17     // 设置SYN标志, 发送请求
18     request.setstate(SYN);
19     ++nowseq; // 增加序列号
20     request.setseq(nowseq);
21     printf("%s", "第一次握手:\n");
22
23     // 发送SYN并等待ACK
24     return sendAndReceiveAck(request, response, sock, ServerAddr, ClientAddr);
25 }
26
27 // 第三次握手: 接收服务器的响应并发送ACK确认
28 bool secondHandshake(SOCKET& sock, SOCKADDR_IN& ServerAddr, UDP_DATAGRAM&
    respond)
29 {
30     UDP_DATAGRAM request;
31     memset(&request, 0, sizeof(UDP_DATAGRAM));
32
33     int t = respond.getseq(); // 获取服务器返回的序列号
34     request.setstate(ACK); // 设置ACK标志
35     request.setack(t); // 设置ACK为服务器的seq
36     ++nowseq; // 增加序列号
37     request.setseq(nowseq); // 设置本地seq
38     printf("%s", "第三次握手:\n");
39
40     // 发送ACK确认报文
41     sendDatagram(request, sock, ServerAddr);
42     request.showseq(); // 显示当前序列号
43
44     return true;
45 }
46
47 // 建立连接: 执行三次握手过程
48 bool initiateConnection(SOCKET& sock, SOCKADDR_IN& ServerAddr, SOCKADDR_IN&
    ClientAddr)
49 {
50     UDP_DATAGRAM respond;
51
52     // 第一次握手
53     if (!firstHandshake(sock, ServerAddr, ClientAddr)) {
54         return false; // 第一次握手失败

```

```

55     }
56
57     // 第二次握手：接收服务器响应并发送确认
58     memset(&respond, 0, sizeof(UDP_DATAGRAM));
59     int len = sizeof(SOCKADDR);
60     recvfrom(sock, (char*)&respond, sizeof(UDP_DATAGRAM), 0, (struct
sockaddr*)&ServerAddr, &len);
61
62     return secondHandshake(sock, ServerAddr, respond);
63 }
64

```

同理，接收端进行发送第二次握手报文时需要设置SYN和ACK状态位，则设置：

handshakeResponse.setstate(SYN | ACK)即可

服务端（接收者）具体建立连接函数为：

```

1 //全局变量序列号操作函数
2 /* 增加发送序列号 */
3 void incrementnowseq(){++nowseq;}
4 /* 获取当前发送序列号 */
5 unsigned short getnowseq(){return nowseq;}
6
7 /* 连接建立 */
8 int establishConnection(UDP_DATAGRAM& request, SOCKET& socket, SOCKADDR_IN&
serverAddress)
9 {
10     UDP_DATAGRAM handshakeResponse;
11     UDP_DATAGRAM clientResponse;
12     memset(&handshakeResponse, 0, sizeof(UDP_DATAGRAM));
13
14     // 增加发送序列号
15     incrementSendSeq();
16     handshakeResponse.setseq(getnowseq()); // 设置发送的序列号
17     handshakeResponse.setack(request.getseq()); // 设置确认号为收到的请求包的序列号
18
19     // 设置握手状态
20     handshakeResponse.setstate(SYN | ACK);
21
22     // 发送连接确认数据报文
23     sendDatagram(datagram, socket, serverAddress);
24     handshakeResponse.showseq();
25
26     // 准备接收第三次握手的响应
27     int addressLength = sizeof(SOCKADDR);


```

```

28     int bytesReceived = recvfrom(socket, (char*)&clientResponse,
    sizeof(UDP_DATAGRAM), 0, (struct sockaddr*)&serverAddress, &addressLength);
29
30     // 判断是否成功收到第三次握手确认
31     if (bytesReceived > 0 && (clientResponse.getState() & ACK))
32     {
33         return 1; // 成功建立连接
34     }
35
36     return 0; // 连接建立失败
37 }
38
39
40

```

2.3.2 在传输”文件头报文“过程中

 SYN位为1且REQ位为1，说明接收到文件头报文

为了实现发送文件的报文数量和文件名称等间接信息的传输，我引入了文件传输的“头报文”机制。就是一旦确认开始传输文件时，程序会先进入 `sendfile` 主逻辑函数，这个函数要做三个事情：

`prepareFileContent`：负责读取文件内容并返回。

`initiateFileTransfer`：负责发起头文件传输的操作。

`sendFileDataChunks`：负责发送文件数据。

为了使得接收方可以得知UDP报文传输总数，要首先告诉接收方一共有多少UDP报文以及传输文件名称、尾部报文字节个数等信息，这些变量在 `prepareFileContent` 函数中完成，会返回一个文件指针便于后续文件发送使用，这部分不是rdt3.0逻辑的关键部分，因此只做简单讲解。

```

1 // 读取文件内容并返回内容和一些元数据
2 char* prepareFileContent(char* fileName, unsigned short& num_udp, unsigned
    short& rest, unsigned int& size) {
3     return readFile(fileName, num_udp, rest, size);
4 }
5
6 // 初始化文件传输请求
7 bool initiateFileTransfer(char* fileName, SOCKET& sock, SOCKADDR_IN&
    ServerAddr, SOCKADDR_IN& ClientAddr, unsigned short num_udp, unsigned short
    rest, unsigned int size) {
8     if (!headfile(fileName, sock, ServerAddr, ClientAddr, num_udp, rest,
        size)) {
9         cout << "[error]文件传输请求失败\n";

```

```
10     return false;
11 }
12 return true;
13 }
14
15 // 发送文件数据
16 bool sendFileDataChunks(unsigned short rest, unsigned short num_udp, SOCKET&
    sock, SOCKADDR_IN& ServerAddr, SOCKADDR_IN& ClientAddr, char* fileContent) {
17     if (!sendFileData(rest, num_udp, sock, ServerAddr, ClientAddr, respond,
        fileContent)) {
18         cout << "[error]文件传输失败\n";
19         return false;
20     }
21     return true;
22 }
23
24 // 主发送文件函数
25 int sendFile(char* fileName, SOCKET& sock, SOCKADDR_IN& ServerAddr,
    SOCKADDR_IN& ClientAddr) {
26     unsigned short num_udp, rest;
27     unsigned int size;
28
29     // 读取文件内容并获取相关元数据
30     char* fileContent = prepareFileContent(fileName, num_udp, rest, size);
31     if (fileContent == NULL) {
32         return 0; // 读取文件失败
33     }
34
35     // 发起 文件头 传输请求
36     if (!initiateFileTransfer(fileName, sock, ServerAddr, ClientAddr, num_udp,
        rest, size)) {
37         return 0; // 请求发送失败
38     }
39
40     // 发送文件拆分好的UDP数据
41     if (!sendFileDataChunks(rest, num_udp, sock, ServerAddr, ClientAddr,
        fileContent)) {
42         return 0; // 发送文件数据失败
43     }
44
45     return 1; // 文件传输成功
46 }
```

在函数 `initiateFileTransfer` 中,发送头报文, 包含UDP报文总数、文件名称, 余字节等信息, 让接收方得知。

这里就是设置了**头报文的状态为SYN且REQ**, 当接收方检测到此两个状态同时出现的报文时, 说明有文件要来传输了, 接收端进行确认: **接收方返回ACK报文**, 发送方的`sendAndWait()`: 停等逻辑函数便得到正确执行。

下表是头报文数据报的具体信息: (其中字段1、2、3直接储存在UDP报文的`message[]`中)

目 表格

| □ | 🔒 字段 | 说明 | 示例值 |
|---|---------------------|------------------|-------------|
| 1 | total | 总报文数, 计算方法: t... | 10 |
| 2 | 文件名 | 文件的名称, 从路径中... | example.txt |
| 3 | rest | 文件UDP传输报文剩余... | 256 |
| 4 | send_DataGram.state | 报文的状态, 设置为SYN... | SYN, REQ |
| 5 | 发送方式 | 使用的发送函数 | sendAndWait |

5 条记录

头报文实现具体代码:

```
1 bool headfile(char* fileName, SOCKET& sock, SOCKADDR_IN& ServerAddr,
2   SOCKADDR_IN& ClientAddr, unsigned short& num_udp, unsigned short& rest,
3   unsigned int& size)
4 {
5     unsigned int t = strlen(fileName);
6     // 拆分文件名
7     while (fileName[t] != '/' && t > 0)
8         t--;
9     // 拼接total和文件名
10    char fileInfo[1024]; // 用于存储 total + 文件名 信息
11    if (rest % 2 != 0)
12        rest += 1;
13    // 计算报文总数
14    int total = (rest == 0 ? num_udp : num_udp + 1);
15    cout << "总报文数: " << total << endl;
16    sprintf(fileInfo, "%d %s %d", total, fileName + t + 1, rest); // 格式化为
17    "total:<total> <filename>"
```

```

16 // 将拼接的 total 和文件名信息、rest信息存入报文
17 strcpy(send_DataGram.getMessage(), fileInfo);
18
19 //报文准备发送
20 memset(&send_DataGram, 0, sizeof(UDP_DATAGRAM));
21 memset(&respond, 0, sizeof(UDP_DATAGRAM));
22 send_DataGram.setseq(++nowseq);
23 //seq=4
24 send_DataGram.setstate(SYN);
25 send_DataGram.setstate(REQ);
26 // 请求传输的初始报文
27 sendAndWait(send_DataGram, respond, sock, ServerAddr, ClientAddr);
28 }

```

2.3.3 在传输文件数据报文过程中



接收端返回报文状态位ACK为1，说明收到正确校验的报文

接收端返回报文状态位REQ为1，说明收到错误校验的报文，需要重传

对于正常的数据报文，接收端收到正确校验的报文时需要设置状态位ACK表示报文确认状态，
(ack.setstate(ACK););收到校验和出错的报文时需要设置状态位为REQ表示差错传输，从而让发送端实现差错重传，这样就完善了停等逻辑。正常的文件数据发送函数 `sendFileData` 的执行过程，
 会对于每一个UDP实现停等协议，调用**sendAndWait ()** 函数来实现传输和停等控制，此部分逻辑具体在2.2节讲述。

接收端确认报文：

```

1 // 构造确认ACK报文
2 ack.setack(datagram.getseq());
3 ack.setstate(ACK);
4 ack.setseq(sendseq);
5 sendDatagram(ack, sock, ClientAddr);

```

发送端检测来自接收端的确认：

```

1 // 处理差错重传
2 if (recv_datagram.getstate() & REQ) {

```



```

3         std::cout << "[Info] 差错检测,准备重传数据报..." << std::endl;
4
5         // 重传数据报
6         sendDataagram(datagram, sock, ServerAddr);
7         datagram.showseq();
8
9         // 重置计时器
10        QueryPerformanceCounter(&timeStart);
11        continue; // 继续等待响应
12    }
13
14    // 检查是否收到正确的ACK
15    if (recv_datagram.getack() == datagram.getseq()) {
16        std::cout << "[Success] 此UDP报文发送且接收成功!" << std::endl;
17
18        // 停止计时器
19        isTimerActive = false;
20        return true; // 传输成功
21    }

```

2.3.4 在连接断开过程中

由于基于程序实现逻辑，**正常断连时一定完成了文件所有数据的传输**，故此处相比TCP的四次挥手**减少了一次挥手**。我直接采用三次挥手机制。具体与2.3.1小节描述的类三次握手机制完全相同，这里不再过多重复叙述。

✓ 需要指出的不同是：在第一次挥手，置 **state为FIN**，第二次挥手接收端发送**state为FIN且ACK**的报文，第三次挥手则发送**state为ACK**的报文即完成了断连确认。

3 【程序测试结果】

3.1传输文件测试

这里尝试传输helloworld.txt和1.jpg，日志正常输出，显示每一个传输报文的序列号：

```
D:\28301\Desktop\CSNet作业\编程3\3-1\Sender\x64\Debug\Sender.exe
停等协议.....
RECVING
此UDP报文发送接收成功!!!*SENDING

传输序列seq为203
停等协议.....
RECVING
此UDP报文发送接收成功!!!*SENDING

传输序列seq为204
停等协议.....
RECVING
此UDP报文发送接收成功!!!*SENDING

传输序列seq为205
停等协议.....
RECVING
此UDP报文发送接收成功!!!*SENDING

传输序列seq为206
停等协议.....
RECVING
此UDP报文发送接收成功!!!*SENDING

传输序列seq为207
停等协议.....
RECVING
此UDP报文发送接收成功!!!文件传输成功输入单字母T传输文件;
输入单字母Q退出

D:\28301\Desktop\CSNet作业\编程3\3-1\receiver\x64\Debug\receiver.exe
接收报文文件列198
203 203校验正确
bSENDING

传输序列seq为203
接收报文文件列199
204 204校验正确
bSENDING

传输序列seq为204
接收报文文件列200
205 205校验正确
bSENDING

传输序列seq为205
接收报文文件列201
206 206校验正确
bSENDING

传输序列seq为206
rest 大小为 1024
207 207校验正确
bSENDING

传输序列seq为207

传输时间:1.9547s
吞吐率:847090char/s
```

图6：传输日志输出

传输时间：1.95s，吞吐率847090char/s。

且最终文件传输成功，大小完全一致，能正常打开且自动完成命名。

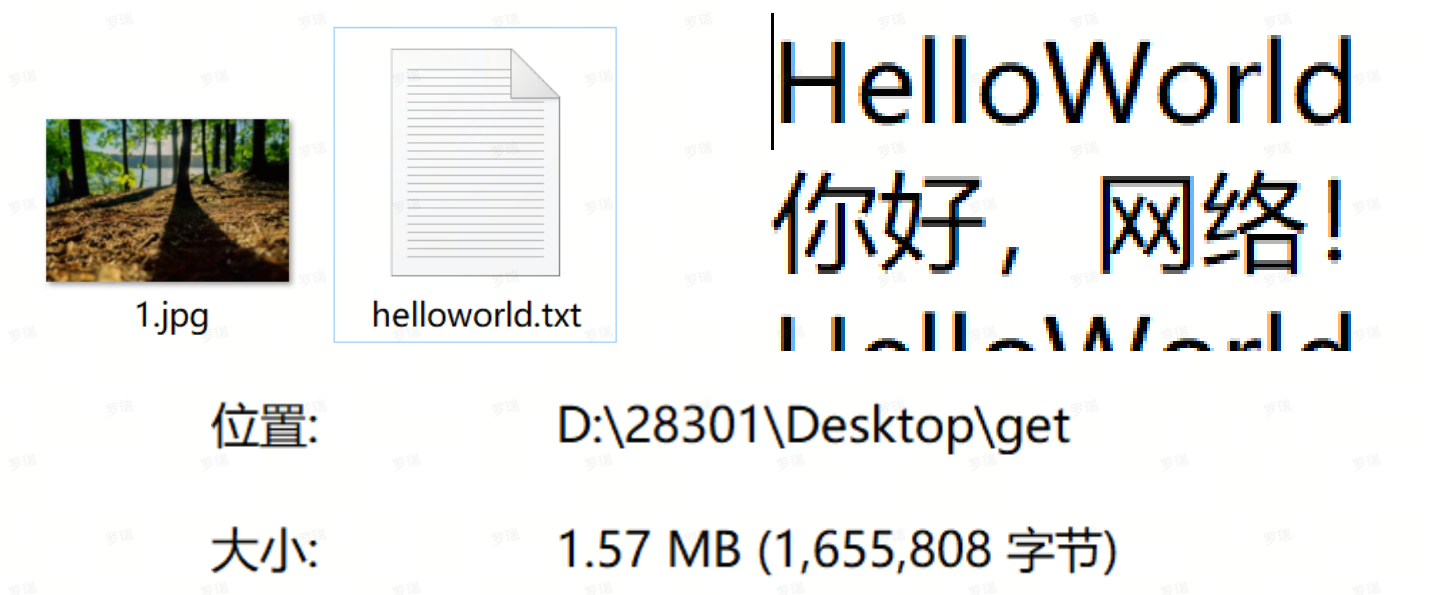


图7：文件正确传输

3.2停等协议模拟之差错测试

通过对接收端recvDatagram函数的设置,实现随机模拟差错传输，激发差错重传。

```
1 int recvDatagram(UDP_DATAGRAM& datagram, SOCKET& sock, SOCKADDR_IN& addr)
2 {
3     int res = 0;
4     int len = sizeof(SOCKADDR);
5     lost++;
```

```

6     res = recvfrom(sock, (char*)&datagram, sizeof(UDP_DATAGRAM), 0, (struct
    sockaddr*)&addr, &len);
7     //覆盖原理设置差错
8     if (Step % lost == 0)
9         memset(&datagram, 0, sizeof(UDP_DATAGRAM));
10
11     return res;
12 }

```

发送端：

传输序列seq为196
 停等协议.....
 RECVING
 传输出错，准备重传
 SENDING

传输序列seq为196
 RECVING
 传输出错，准备重传
 SENDING

传输序列seq为196
 RECVING
 此UDP报文发送接收成功!!!*SENDING

图8-1 差错重传 发送端日志输出

接收端：

```
0 196校验出错
65043ASENDING
接下来打印出错报文的发送端seq
传输序列seq为0
0
196 196校验出错
59937ASENDING
接下来打印出错报文的发送端seq
传输序列seq为196
8200
196 196校验正确
bSENDING
传输序列seq为196
接收报文文件列192
```

图8-2 差错重传 发送端日志输出

最终重传报文正确通过验证，正确传输文件。