

## Lab3-1 基于UDP服务设计可靠传输协议并编程实现

[jassary08/Computer\\_Network \(github.com\)](https://github.com/jassary08/Computer_Network)

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、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。

## 1.协议设计

本次实验我仿照了TCP协议对**UDP 协议**的报文格式进行了设计，以支持可靠传输和连接管理。报文头部包含多个字段：`src_port` 和 `dest_port` 用于标识通信的源端口和目标端口，`seq` 和 `ack` 分别表示序列号和确认号，用于控制数据的发送和接收顺序，确保可靠性。`length` 表示整个数据报的长度，`check` 用于存储校验和，验证数据完整性。`flag` 字段定义了一组标志位，用于连接建立（SYN）、终止（FIN）、确认（ACK）以及自定义功能（CFH）。`reserved` 提供了扩展空间。数据部分最大支持 1024 字节，用于传输实际的应用数据。

字段	大小 (字节)	描述
src_port	4	源端口号
dest_port	4	目标端口号
seq	4	序列号, 用于顺序控制
ack	4	确认号, 用于确认接收数据
length	4	数据包长度
check	2	校验和
flag	2	标志位, 控制协议状态
data	1024	数据部分

0	15   16	31
+-----+		
	src_port (4 bytes)	
+-----+		
	dest_port (4 bytes)	
+-----+		
	seq (4 bytes)	
+-----+		
	ack (4 bytes)	



## 2.消息传输机制

本实验中使用的**ACK** 的值直接**等于**发送数据包的**序列号**，使用这种非累加的确认机制，更适合 UDP 等无连接协议，能够明确地告诉发送端接收端收到了哪些包，提高了可靠性，适合对传输完整性要求较高的实验和场景（如文件传输、实时通信等）。

- **三次握手——建立连接**

改进后的协议建立连接的流程如下：

步骤	发送方	接收方
第一次握手	发送 SYN, Seq = X	
第二次握手		接收并发送 SYN, ACK = X, Seq = Y
第三次握手	接收并发送 ACK = Y	

- **差错检查机制——校验和**

实验设计的**校验和机制**的核心思想是通过累加数据字段的值来形成一个**校验和**，并在传输数据时携带该校验和。当接收端收到数据时，重新计算数据的校验和，并与收到的校验和对比：

- 如果两者一致，说明数据未被破坏。
- 如果不一致，说明数据在传输过程中发生了错误。

- **流量控制机制——停等机制**

停等机制的核心思想是**一次只发送一个数据包**，等待接收方确认（ACK）后再发送下一个数据包。这一过程分为以下几个步骤：

1. **发送数据包：**

- 发送方将当前要发送的数据封装为一个数据包，并附带唯一的序列号（Sequence Number）。
- 数据包被发送到接收方。

2. **等待确认（ACK）：**

- 发送方在发送数据包后，进入等待状态，等待接收方返回的确认消息（ACK）。
- 如果在一定时间内未收到确认消息，发送方会重新发送该数据包（重传机制）。

3. **接收方处理：**

- 接收方在收到数据包后，校验数据的完整性（如校验和验证）。
- 若数据无误，则发送确认消息（ACK）并处理数据。

- 若数据出错（如校验和不一致），丢弃数据包，不发送确认消息。
4. 继续传输：
- 发送方收到确认消息后，继续发送下一个数据包。
- 重传机制——超时重传
- 本实验采用了**动态调整重传时间（RTT-based Timeout Adjustment）**，一种提高传输效率和可靠性的机制。可以通过估算实时的**往返时间（RTT）**，动态调整**超时时间（timeout）**，从而更好适应网络状况的变化。
- 动态调整重传时间的核心思想是**根据往返时间（Round-Trip Time, RTT）的实时测量结果，动态更新超时时间（Timeout Interval）**。超时时间的设置基于以下公式：

$$\text{Timeout Interval} = \text{Estimated RTT} + 4 \times \text{Dev RTT}$$

- 其中：
1. **Estimated RTT** 是当前 RTT 的平滑估值。
- $$\text{Estimated RTT} = (1 - \alpha) \cdot \text{Estimated RTT} + \alpha \cdot \text{Sample RTT}$$
2. **Dev RTT** 是 RTT 的偏差，用于反映网络延迟的波动性。
- $$\text{Dev RTT} = (1 - \beta) \cdot \text{Dev RTT} + \beta \cdot |\text{Sample RTT} - \text{Estimated RTT}|$$

- 四次挥手——断开连接
- 四次挥手的过程与三次握手类似，发送的确认号为对方发来的序列号。

步骤	发送方	接收方
第一次挥手	发送 FIN, Seq = U	
第二次挥手		接收并发送 ACK, Ack = U
第三次挥手		发送 FIN, Seq = V
第四次挥手	接收并发送 ACK, Ack = V	

## 二. 代码实现

### 1. 协议设计及宏定义

我们在头文件 `udp_packet.h` 中定义了UDP数据包的结构体，以及一些**宏定义常量和函数**。在 `udp_packet.cpp` 中给出了宏定义函数的具体实现。

具体各成员变量和功能在上一章节已经阐述，此处不再过多赘述。

```
#ifndef UDP_PACKET_H
#define UDP_PACKET_H

#include <iostream>
#include <bitset>
#include <cstdint>
#include <cstring>
#include <winsock2.h>
#include <fstream>
#include <thread>
```

```

#include <ws2tcpip.h>
#include <chrono>
using namespace std;

#define MAX_DATA_SIZE 1024 // 数据部分的最大大小

#define TIMEOUT 1000 // 超时时间（毫秒）

#define CLIENT_PORT 54321 // 客户端端口
#define ROUTER_PORT 12345 // 目标路由端口
#define CLIENT_IP "127.0.0.1" // 目标路由 IP 地址
#define ROUTER_IP "127.0.0.1" // 目标路由 IP 地址

// UDP 数据报结构
struct UDP_Packet {
    uint32_t src_port; // 源端口
    uint32_t dest_port; // 目标端口
    uint32_t seq; // 序列号
    uint32_t ack; // 确认号
    uint32_t length; // 数据长度（包括头部和数据）
    uint16_t flag; // 标志位
    uint16_t check; // 校验和
    char data[MAX_DATA_SIZE]; // 数据部分

    // 标志位掩码
    static constexpr uint16_t FLAG_FIN = 0x8000; // FIN 位
    static constexpr uint16_t FLAG_CFH = 0x4000; // CFH 位
    static constexpr uint16_t FLAG_ACK = 0x2000; // ACK 位
    static constexpr uint16_t FLAG_SYN = 0x1000; // SYN 位

    UDP_Packet() : src_port(0), dest_port(0), seq(0), ack(0), length(0),
flag(0), check(0) {
        memset(data, 0, MAX_DATA_SIZE); // 将数据部分初始化为 0
    }

    // 设置标志位
    void Set_CFH();
    bool Is_CFH() const;

    void Set_ACK();
    bool Is_ACK() const;

    void Set_SYN();
    bool Is_SYN() const;

    void Set_FIN();
    bool Is_FIN() const;

    // 校验和计算
    uint16_t calculate_Checksum() const;

    // 校验和验证
    bool CheckValid() const;

```

```

    // 打印消息
    void Print_Message() const;
};

#endif

```

在 `udp_packet.cpp` 中为 `CFH`、`ACK`、`SYN` 和 `FIN` 四种标志位提供了设置和检查方法，使用按位或操作 (`|=`) 来设置特定标志位，按位与操作 (`&`) 检查某一位是否被设置。

```

// 以ACK位的设置和检查为例
void UDP_Packet::Set_ACK() {
    flag |= FLAG_ACK;
}

bool UDP_Packet::Is_ACK() const {
    return (flag & FLAG_ACK) != 0;
}

```

`Calculate_Checksum` 方法负责对包头和数据部分计算校验和，用于保证数据完整性：

- 在计算校验和前，验证 `this` 和 `data` 指针，避免空指针导致的错误。
- 将包头字段逐一累加到 `sum`。
- 按双字节读取数据部分，每次读取两个字节形成一个 16 位的 `word`，并累加。
- 将 32 位的累加和中的高 16 位进位加回到低 16 位，直到 `sum` 只剩下 16 位。
- 返回 `sum` 的按位取反值作为校验和。

```

uint16_t UDP_Packet::Calculate_Checksum() const {
    // 验证 this 和 data 的有效性
    if (this == nullptr) {
        cerr << "[错误] this 指针为空，无法计算校验和。" << endl;
        return 0;
    }
    if (data == nullptr) {
        cerr << "[错误] 数据指针无效，无法计算校验和。" << endl;
        return 0;
    }

    uint32_t sum = 0;

    // 累加 UDP 头部
    sum += src_port;
    sum += dest_port;
    sum += (seq >> 16) & 0xFFFF;
    sum += seq & 0xFFFF;
    sum += (ack >> 16) & 0xFFFF;
    sum += ack & 0xFFFF;
    sum += length;

    // 累加数据部分，确保范围合法
    for (size_t i = 0; i < MAX_DATA_SIZE - 1 && i + 1 < length; i += 2) {
        uint16_t word = (data[i] << 8) | (data[i + 1] & 0xFF);
        sum += word;
    }
}

```

```

// 将进位加回低 16 位
while (sum >> 16) {
    sum = (sum & 0xFFFF) + (sum >> 16);
}

return ~sum & 0xFFFF;
}

```

`checkValid` 方法负责验证接收包的校验和是否正确,通过比较包内的 `check` 值和 `CalculateChecksum()` 的结果判断包是否完整。

```

bool UDP_Packet::CheckValid() const {
    return (check & 0xFFFF) == CalculateChecksum();
}

```

## 2. 套接字初始化

发送端和接收端的部分我们分别在 `client_send.cpp` 和 `server_receive.cpp` 中实现。我通过 `UDPClient` 类和 `UDPServer` 类进行封装,类内实现了**初始化**, **建立连接**, **传输文件**和**断开连接**等功能。

以**发送端**为例, `SOCKET clientSocket` 用于存储客户端的 UDP 套接字。 `sockaddr_in clientAddr` 保存客户端的地址信息,包括 IP 和端口。保存目标路由的地址信息。 `uint32_t seq` 客户端的当前序列号,用于标记数据包的顺序,在发送每个数据包时递增,确保可靠性。

```

class UDPClient {
private:
    SOCKET clientSocket;           // 客户端套接字
    sockaddr_in clientAddr;        // 客户端地址
    sockaddr_in routerAddr;        // 目标路由地址
    uint32_t seq;                  // 客户端当前序列号

public:
    UDPClient() : clientSocket(INVALID_SOCKET), seq(0) {}
    bool init() {}
    bool connect() {}
    bool Send_Message(const string& file_path) {}
    bool Disconnect() {}
    ~UDPClient() {}
};

```

在**初始化**客户端UDP套接字的过程中,首先初始化**winsock库**,并检查版本是否与指定版本匹配,接着初始化**UDP套接字**,使用IPv4地址族,并将套接字设置为**非阻塞模式**,接着将**客户端地址**绑定在套接字上,最后配置好**目标路由地址**,初始化的过程完成。

```

bool init() {
    // 初始化 winsock
    WSADATA wsaData;
    int result = WSStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0) {
        cerr << "[错误] WSStartup 失败, 错误代码: " << result << endl;
    }
}

```

```

        return false;
    }

    // 检查版本是否匹配
    if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2) {
        cerr << "[错误] 不支持的 winsock 版本。" << endl;
        WSACleanup();
        return false;
    }

    // 创建 UDP 套接字
    clientSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (clientSocket == INVALID_SOCKET) {
        cerr << "[错误] 套接字创建失败, 错误代码: " << WSAGetLastError() << endl;
        WSACleanup();
        return false;
    }

    cout << "[日志] 套接字创建成功。" << endl;

    // 设置非阻塞模式
    u_long mode = 1;
    if (ioctlsocket(clientSocket, FIONBIO, &mode) != 0) {
        cerr << "[错误] 设置非阻塞模式失败, 错误代码: " << WSAGetLastError() <<
endl;

        closesocket(clientSocket);
        WSACleanup();
        return false;
    }

    cout << "[日志] 套接字设置为非阻塞模式。" << endl;

    // 配置客户端地址
    memset(&clientAddr, 0, sizeof(clientAddr));
    clientAddr.sin_family = AF_INET;
    clientAddr.sin_port = htons(CLIENT_PORT);
    inet_pton(AF_INET, CLIENT_IP, &clientAddr.sin_addr);

    // 绑定客户端地址到套接字
    if (bind(clientSocket, (sockaddr*)&clientAddr, sizeof(clientAddr)) ==
SOCKET_ERROR) {
        cerr << "[错误] 套接字绑定失败, 错误代码: " << WSAGetLastError() << endl;
        closesocket(clientSocket);
        WSACleanup();
        return false;
    }

    cout << "[日志] 套接字绑定到本地地址: 端口 " << CLIENT_PORT << endl;

    // 配置目标路由地址
    memset(&routerAddr, 0, sizeof(routerAddr));
    routerAddr.sin_family = AF_INET;
    routerAddr.sin_port = htons(ROUTER_PORT);
    inet_pton(AF_INET, ROUTER_IP, &routerAddr.sin_addr);

```

```
        return true;
    }
```

### 3. 三次握手——建立连接

#### 发送端

`connect()` 方法实现了通过 UDP 协议的三次握手过程，此处仿照 TCP 连接的建立。三次握手流程确保了发送方和接收方的同步，保证双方连接建立成功。同样以发送端为例：

##### 1. 第一次握手

- 初始化一个 `UDP_Packet`，设置以下字段：
  - 源端口和目标端口。
  - 设置 `SYN` 标志位，表示这是一个连接请求包。
  - 设置序列号 `seq`，并计算校验和。
- 发送 `SYN` 数据包。
- 记录发送时间 `msg1_Send_Time`，用于后续超时重传判断。

##### 2. 第二次握手

- 循环等待接收服务端返回的 `SYN + ACK` 包：
  - 验证标志位、校验和和确认号是否正确。
  - 如果验证通过，退出循环，表示第二次握手成功。
- 如果超时未收到正确的响应包：
  - 重新发送第一次握手的 `SYN` 包，并更新发送时间。

##### 3. 第三次握手

- 初始化一个新的 `UDP_Packet`：
  - 设置序列号为 `seq + 1`。
  - 确认号 `ack` 设置为服务端的序列号 `con_msg[1].seq`。
  - 设置 `ACK` 标志位，表示确认服务端的同步。
  - 发送第三次握手的 `ACK` 包。

```
bool connect() {
    UDP_Packet con_msg[3]; // 三次握手消息

    // 第一次握手
    con_msg[0] = {}; // 清空结构体
    con_msg[0].src_port = CLIENT_PORT;
    con_msg[0].dest_port = ROUTER_PORT;
    con_msg[0].Set_SYN(); // 设置 SYN 标志位
    con_msg[0].seq = ++seq; // 设置序列号
    con_msg[0].check = con_msg[0].Calculate_Checksum(); // 计算校验和
    auto msg1_Send_Time = chrono::steady_clock::now(); // 记录发送时间

    cout << "[日志] 第一次握手：发送 SYN..." << endl;
    if (sendto(clientSocket, (char*)&con_msg[0], sizeof(con_msg[0]), 0,
        (sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
        cerr << "[错误] 第一次握手消息发送失败。" << endl;
        return false;
    }
}
```



```

    }

    // 第二次握手
    socklen_t addr_len = sizeof(routerAddr);
    while (true) {
        // 接收 SYN+ACK 消息
        if (recvfrom(clientSocket, (char*)&con_msg[1], sizeof(con_msg[1]),
0,
            (sockaddr*)&routerAddr, &addr_len) > 0) {
            if (con_msg[1].Is_ACK() && con_msg[1].Is_SYN() &&
con_msg[1].IsValid() &&
                con_msg[1].ack == con_msg[0].seq) {
                cout << "[日志] 第二次握手成功: 收到 SYN+ACK。" << endl;
                break;
            }
            else {
                cerr << "[错误] 第二次握手消息验证失败。" << endl;
            }
        }

        // 超时重传第一次握手消息
        auto now = chrono::steady_clock::now();
        if (chrono::duration_cast<chrono::milliseconds>(now -
msg1_Send_Time).count() > TIMEOUT) {
            cout << "[日志] 超时, 重传第一次握手消息。" << endl;
            con_msg[0].check = con_msg[0].Calculate_Checksum(); // 重新计算校验
和
            if (sendto(clientSocket, (char*)&con_msg[0], sizeof(con_msg[0]),
0,
                (sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR)
{
                cerr << "[错误] 重传失败。" << endl;
                return false;
            }
            msg1_Send_Time = now; // 更新发送时间
        }
    }
    seq = con_msg[1].seq;
    // 第三次握手
    con_msg[2] = {}; // 清空结构体
    con_msg[2].src_port = CLIENT_PORT;
    con_msg[2].dest_port = ROUTER_PORT;
    con_msg[2].seq = seq + 1; // 设置序列号
    con_msg[2].ack = con_msg[1].seq; // 设置确认号
    con_msg[2].Set_ACK(); // 设置 ACK 标志位
    con_msg[2].check = con_msg[2].Calculate_Checksum(); // 计算校验和
    cout << "[日志] 第三次握手: 发送 ACK..." << endl;
    if (sendto(clientSocket, (char*)&con_msg[2], sizeof(con_msg[2]), 0,
        (sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
        cerr << "[错误] 第三次握手消息发送失败。" << endl;
        return false;
    }

    cout << "[日志] 三次握手完成, 连接建立成功。" << endl;
    return true;

```

```
}
```

## 服务器端

**服务器端**建立连接的过程与客户端相对应，首先接收来自客户端的 `SYN` 消息，接收后对消息进行校验，发送 `SYN + ACK` 消息，最后等待接收客户端的第三次握手消息。

代码逻辑与客户端相似，此处不再展示。

## 4. 发送数据包

### 客户端

`Send_Message()` 方法实现了通过 UDP 协议发送文件的功能，使用停等协议确保可靠传输，并引入动态 RTT 调整机制以优化超时时间设置。主要的流程如下：

#### 1. 初始化文件信息

- 打开文件，检查路径有效性。
- 提取文件名和计算文件大小。
- 日志输出文件的基本信息。

#### 2. 文件头信息发送

- 构造文件头数据包，包含以下信息：
  - 文件名。
  - 文件大小。
  - 校验和。
- 发送文件头数据包，并在超时未收到 ACK 的情况下进行重传，直到收到有效 ACK。

#### 3. 文件内容发送

- 将文件划分为多个数据段：
  - 每个段最大大小为 `MAX_DATA_SIZE`。
  - 使用最后一个段处理文件尾。
- 逐段发送文件内容：
  - 使用停等协议，等待每段的 ACK，未收到则超时重传。
  - 动态调整 RTT 和超时时间，提升效率。

#### 4. 动态调整 RTT 和超时时间

- 使用公式动态更新超时时间：

```
estimated_rtt = (1 - alpha) * estimated_rtt + alpha * sample_rtt;  
dev_rtt = (1 - beta) * dev_rtt + beta * abs(sample_rtt - estimated_rtt);  
timeout_interval = estimated_rtt + 4 * dev_rtt;
```

- `estimated_rtt` 是当前 RTT 的加权平均估值。
- `dev_rtt` 是 RTT 的偏差，用于反映网络延迟波动性。

## 5. 文件传输统计

- 记录总耗时。
- 计算吞吐率 (KB/s) 。

```
bool Send_Message(const string& file_path) {
    // 打开文件
    ifstream file(file_path, ios::binary);
    if (!file.is_open()) {
        cerr << "[错误] 无法打开文件: " << file_path << endl;
        return false;
    }

    // 获取文件名和文件大小
    size_t pos = file_path.find_last_of("/\\");
    string file_name = (pos != string::npos) ? file_path.substr(pos + 1) :
file_path;
    file.seekg(0, ios::end);
    size_t file_size = file.tellg();
    file.seekg(0, ios::beg);

    cout << "[日志] 准备发送文件: " << file_name << ", 大小: " << file_size << "
字节。" << endl;

    // 构造文件头信息
    UDP_Packet header_packet{};
    header_packet.src_port = CLIENT_PORT;
    header_packet.dest_port = ROUTER_PORT;
    header_packet.Set_CFH(); // 设置 CFH 标志位
    header_packet.seq = ++seq; // 设置序列号
    strncpy_s(header_packet.data, file_name.c_str(), MAX_DATA_SIZE - 1); // 写
入文件名
    header_packet.length = file_size; // 写入文件大小
    header_packet.check = header_packet.Calculate_Checksum(); // 设置校验和
    // 发送文件头消息
    if (sendto(clientSocket, (char*)&header_packet, sizeof(header_packet), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
        cerr << "[错误] 文件头信息发送失败, 错误代码: " << WSAGetLastError() <<
endl;
        return false;
    }
    cout << "[日志] 文件头信息已发送。" << endl;
    // 发送文件头信息, 启用超时重传机制
    auto start_time = chrono::steady_clock::now();
    while (true) {

        // 接收 ACK 确认
        UDP_Packet ack_packet{};
        socklen_t addr_len = sizeof(routerAddr);
        if (recvfrom(clientSocket, (char*)&ack_packet, sizeof(ack_packet), 0,
(sockaddr*)&routerAddr, &addr_len) > 0) {
            if (ack_packet.Is_ACK() && ack_packet.ack == header_packet.seq) {
                cout << "[日志] 收到文件头信息的 ACK 确认。" << endl;
                break; // 成功接收确认, 退出重传循环
            }
        }
    }
}
```

```

        else {
            cerr << "[错误] 收到的 ACK 无效或不匹配。" << endl;
        }
    }

    // 检测超时
    auto now = chrono::steady_clock::now();
    if (chrono::duration_cast<chrono::milliseconds>(now -
start_time).count() > TIMEOUT) {
        cout << "[日志] 超时，重新发送文件头信息。" << endl;
        header_packet.check = header_packet.Calculate_Checksum(); // 重新计
算校验和
        if (sendto(clientSocket, (char*)&header_packet,
sizeof(header_packet), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR)
{
            cerr << "[错误] 重传失败。" << endl;
            return false;
        }
        start_time = now; // 更新发送时间
    }
}

// 文件头信息发送成功
cout << "[日志] 文件头信息发送完成，准备发送文件数据。" << endl;

// 开始发送文件内容
size_t total_segments = file_size / MAX_DATA_SIZE; // 完整数据段数量
size_t last_segment_size = file_size % MAX_DATA_SIZE; // 最后一个数据段的
大小
size_t total_sent = 0; // 已发送字节总数
size_t current_segment = 0; // 当前发送段编号

auto file_start_time = chrono::steady_clock::now(); // 记录文件传输的开始时间

double estimated_rtt = 100.0; // 初始估计的 RTT
double timeout_interval = 200.0; // 初始超时时间
double alpha = 0.125; // 平滑因子
double beta = 0.25; // 平滑因子
double dev_rtt = 0.0; // RTT 偏差

while (total_sent < file_size) {
    UDP_Packet data_packet{};
    data_packet.src_port = CLIENT_PORT;
    data_packet.dest_port = ROUTER_PORT;
    size_t segment_size = (current_segment < total_segments) ?
MAX_DATA_SIZE : last_segment_size;

    file.read(data_packet.data, segment_size);
    data_packet.seq = ++seq;
    data_packet.length = segment_size;
    data_packet.check = data_packet.Calculate_Checksum();

    auto segment_send_time = chrono::steady_clock::now();
    bool ack_received = false;

```

```

        if (sendto(clientSocket, (char*)&data_packet, sizeof(data_packet), 0,
            (sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
            cerr << "[错误] 数据段发送失败, 错误代码: " << WSAGetLastError() <<
endl;

            return false;
        }

        cout << "[日志] 数据段 " << current_segment + 1
            << " 已发送, 大小: " << segment_size << " 字节。" << endl;

        while (!ack_received) {
            UDP_Packet ack_packet{};
            socklen_t addr_len = sizeof(routerAddr);
            if (recvfrom(clientSocket, (char*)&ack_packet,
                sizeof(ack_packet), 0,
                (sockaddr*)&routerAddr, &addr_len) > 0) {
                if (ack_packet.Is_ACK() && ack_packet.ack == data_packet.seq)
                {
                    ack_received = true;

                    // 动态调整 RTT 和超时时间
                    auto now = chrono::steady_clock::now();
                    double sample_rtt = chrono::duration<double, milli>(now -
segment_send_time).count();
                    estimated_rtt = (1 - alpha) * estimated_rtt + alpha *
sample_rtt;
                    dev_rtt = (1 - beta) * dev_rtt + beta * abs(sample_rtt -
estimated_rtt);
                    timeout_interval = estimated_rtt + 4 * dev_rtt;

                    cout << "[日志] 收到 ACK, 确认序列号: " << ack_packet.ack
                        << ", RTT: " << sample_rtt << " ms。" << endl;
                    cout << "[调试] 新的超时时间: " << timeout_interval << " ms,
估计的 RTT: "
                        << estimated_rtt << " ms, RTT 偏差: " << dev_rtt << "
ms。" << endl;

                    break;
                }
            }
            else {
                cerr << "[警告] 收到无效 ACK 或序列号不匹配。" << endl;
            }
        }

        auto now = chrono::steady_clock::now();
        if (chrono::duration_cast<chrono::milliseconds>(now -
segment_send_time).count() > timeout_interval) {
            cout << "[日志] 数据段 " << current_segment + 1
                << " 超时, 重新发送。" << endl;
            data_packet.check = data_packet.Calculate_Checksum();
            if (sendto(clientSocket, (char*)&data_packet,
                sizeof(data_packet), 0,
                (sockaddr*)&routerAddr, sizeof(routerAddr)) ==
SOCKET_ERROR) {

```

```

        cerr << "[错误] 重传失败。" << endl;
        return false;
    }
    segment_send_time = now;
}

total_sent += segment_size;
current_segment++;
}

// 文件传输完成
auto file_end_time = chrono::steady_clock::now();
double total_time = chrono::duration<double>(file_end_time -
file_start_time).count();
double throughput = (total_sent / 1024.0) / total_time; // KB/s
cout << "[日志] 文件传输完成, 总耗时: " << total_time
    << " 秒, 吞吐率: " << throughput << " KB/s。" << endl;

file.close();
return true;
}

```

## 服务器端

服务端接收文件的过程与客户端相对应:

### 1. 接收文件头部信息

- 首先通过 `recvfrom` 接收文件头部数据包。
- 验证文件头信息的合法性, 包括:
  - 检查是否设置了 `CFH` 标志位。
  - 校验数据包的校验和。
- 文件头中包含以下信息:
  - 文件名: 从 `headerPacket.data` 获取。
  - 文件大小: 从 `headerPacket.length` 获取。
- 返回文件头确认 ACK, 确保客户端正确接收 ACK 才开始发送文件内容。

### 2. 打开文件准备写入

- 使用文件名创建输出路径, 将接收的文件数据存储到指定目录。
- 如果文件打开失败, 返回错误信息并退出。

### 3. 接收文件数据

- 循环接收文件内容数据包:
  - 每次接收数据包后, 检查校验和合法性。
  - 如果数据包的序号符合预期, 则写入文件, 并更新接收进度与期望序列号。
  - 返回对应的 ACK, 通知发送端已正确接收。

## 4. 结束文件接收

- 文件接收完成后，关闭文件。
- 输出日志，显示接收文件的总字节数与保存路径。

接收文件代码部分与客户端高度重合，此处不做展示。

## 5. 四次挥手——断开连接

### 客户端

`disconnect()` 方法实现了基于四次挥手机制的连接断开流程。通过确保双方的 FIN 和 ACK 消息正确收发，达到可靠断开连接的目的。

#### 1. 第一次挥手：发送 FIN 消息

- 通过 `Set_FIN` 设置 FIN 标志位，表示开始断开连接。
- 设置源端口和目标端口，计算校验和，并通过 `sendto` 发送消息。
- 如果超时未收到 ACK，重传 FIN 消息。

#### 2. 第二次挥手：接收 ACK 消息

- 通过 `recvfrom` 接收 ACK 消息。
- 验证消息合法性：
  - 是否设置了 ACK 标志位。
  - ACK 是否对应第一次挥手的序列号。
  - 校验和是否正确。
- 超时重传第一次挥手的 FIN 消息。

#### 3. 第三次挥手：接收 FIN 消息

- 通过 `recvfrom` 接收服务端的 FIN 消息。
- 验证消息合法性：
  - 是否设置了 FIN 标志位。
  - 校验和是否正确。
- 如果超时未收到 FIN，断开连接失败。

#### 4. 第四次挥手：发送 ACK 消息

- 设置 ACK 标志位，确认服务端的 FIN 消息。
- 通过 `sendto` 发送 ACK，标志连接已完全断开。

```
bool Disconnect() {
    UDP_Packet wavehand_packets[4]; // 定义四次挥手消息数组
    socklen_t addr_len = sizeof(routerAddr);
    auto start_time = chrono::steady_clock::now();

    // 初始化挥手消息数组
    memset(wavehand_packets, 0, sizeof(wavehand_packets)); // 清零消息结构体数组

    // 第一次挥手：发送 FIN 消息
    wavehand_packets[0].src_port = CLIENT_PORT;
    wavehand_packets[0].dest_port = ROUTER_PORT;
    wavehand_packets[0].Set_FIN();
    wavehand_packets[0].seq = ++seq;
```

```

        wavehand_packets[0].check = wavehand_packets[0].Calculate_Checksum();
        cout << "[日志] 第一次挥手: 发送 FIN 消息, 序列号: " <<
wavehand_packets[0].seq << endl;
        if (sendto(clientSocket, (char*)&wavehand_packets[0],
sizeof(wavehand_packets[0]), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
            cerr << "[错误] FIN 消息发送失败, 错误代码: " << WSAGetLastError() <<
endl;
            return false;
        }
        while (true) {
            // 第二次挥手: 等待 ACK 消息
            if (recvfrom(clientSocket, (char*)&wavehand_packets[1],
sizeof(wavehand_packets[1]), 0,
(sockaddr*)&routerAddr, &addr_len) > 0) {
                if (wavehand_packets[1].Is_ACK() &&
                    wavehand_packets[1].ack == wavehand_packets[0].seq &&
                    wavehand_packets[1].IsValid()) {
                    cout << "[日志] 收到第二次挥手消息 (ACK), 确认序列号: " <<
wavehand_packets[1].ack << endl;
                    break;
                }
                else {
                    cerr << "[警告] 收到无效的 ACK 消息, 丢弃。" << endl;
                }
            }

            // 超时重传第一次挥手消息
            auto now = chrono::steady_clock::now();
            if (chrono::duration_cast<chrono::milliseconds>(now -
start_time).count() > TIMEOUT) {
                cout << "[日志] FIN 消息超时, 重新发送。" << endl;
                wavehand_packets[0].check =
wavehand_packets[0].Calculate_Checksum(); // 重算校验和
                if (sendto(clientSocket, (char*)&wavehand_packets[0],
sizeof(wavehand_packets[0]), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR)
{
                    cerr << "[错误] 重传失败。" << endl;
                    return false;
                }
                start_time = now; // 更新计时
            }
        }

        // 第三次挥手: 接收 FIN 消息
        start_time = chrono::steady_clock::now();
        while (true) {
            if (recvfrom(clientSocket, (char*)&wavehand_packets[2],
sizeof(wavehand_packets[2]), 0,
(sockaddr*)&routerAddr, &addr_len) > 0) {
                cout << wavehand_packets[2].Is_FIN() <<
wavehand_packets[2].IsValid();
                if (wavehand_packets[2].Is_FIN() &&
                    wavehand_packets[2].IsValid()) {

```



```

        cout << "[日志] 收到第三次挥手消息 (FIN)，序列号: " <<
wavehand_packets[2].seq << endl;
        break;
    }
    else {
        wavehand_packets[2].Print_Message();
        cerr << "[警告] 收到无效的 FIN 消息，丢弃。" << endl;
    }
}

// 超时处理
auto now = chrono::steady_clock::now();
if (chrono::duration_cast<chrono::milliseconds>(now -
start_time).count() > TIMEOUT) {
    cerr << "[日志] 等待 FIN 超时，断开连接失败。" << endl;
    return false;
}
}
seq = wavehand_packets[2].seq;
// 第四次挥手：发送 ACK 消息
wavehand_packets[3].src_port = CLIENT_PORT;
wavehand_packets[3].dest_port = ROUTER_PORT;
wavehand_packets[3].Set_ACK();
wavehand_packets[3].ack = wavehand_packets[2].seq;
wavehand_packets[3].seq = ++seq;
wavehand_packets[3].check = wavehand_packets[3].Calculate_Checksum();
if (sendto(clientSocket, (char*)&wavehand_packets[3],
sizeof(wavehand_packets[3]), 0,
(sockaddr*)&routerAddr, sizeof(routerAddr)) == SOCKET_ERROR) {
    cerr << "[错误] 第四次挥手消息发送失败，错误代码: " << WSAGetLastError() <<
endl;
    return false;
}
cout << "[日志] 第四次挥手：发送 ACK 消息，确认序列号: " <<
wavehand_packets[3].ack << endl;

// 等待 2 * TIMEOUT 时间以确保消息完成
cout << "[日志] 等待 2 * TIMEOUT 确保连接断开..." << endl;
this_thread::sleep_for(chrono::milliseconds(2 * TIMEOUT));
return true;
}

```

## 服务器端

服务器端实现断开连接的操作与客户端相对应：

### 1. 第一次挥手：接收 FIN 消息

- 调用 `recvfrom` 函数接收客户端的 **FIN** 消息。
- 使用 `IS_FIN` 和 `CheckValid` 验证消息的合法性，确保收到的是有效的 **FIN** 消息。

## 2. 第二次挥手：发送 ACK 消息

- 构造 **ACK** 消息，通过 `Set_ACK` 设置 **ACK** 标志位。
- 使用 `Calculate_Checksum` 计算校验和，确保消息完整性。
- 调用 `sendto` 函数发送 **ACK** 消息给客户端。

## 3. 第三次挥手：发送 FIN 消息

- 构造 **FIN** 消息，使用 `Set_FIN` 和 `Set_ACK` 同时设置 **FIN** 和 **ACK** 标志位。
- 再次通过 `Calculate_Checksum` 计算校验和，调用 `sendto` 函数发送消息。

## 4. 第四次挥手：接收 ACK 消息

- 调用 `recvfrom` 接收客户端发送的 **ACK** 消息。
- 验证消息合法性：
  - 是否设置了 **ACK** 标志位。
  - 确认号是否匹配服务端发送的 **FIN** 消息序列号。
- 如果在超时时间内未收到 ACK，重传 **FIN** 消息。
- 重新计算校验和，确保重传消息完整。

断开连接代码部分与客户端高度重合，此处不做展示。

## 程序效果

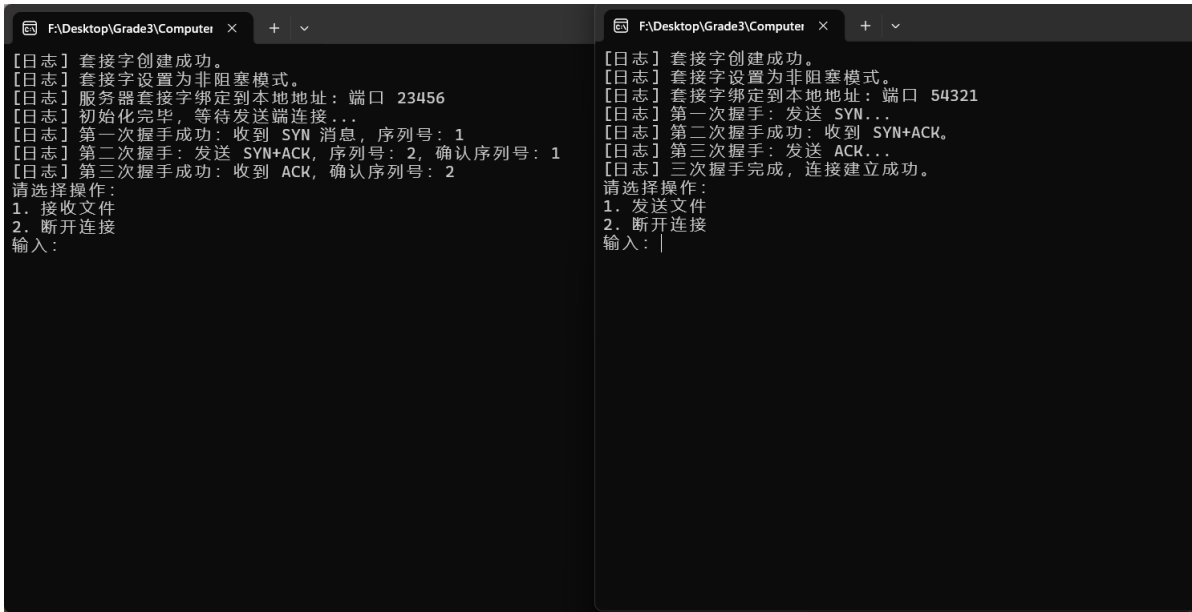
为了方便对程序的运行效果进行测试，客户端与服务器端之间的交互要通过一个路由转发，基本原理如图所示：



首先我们对路由程序进行设置，路由IP地址为127.0.0.1，端口号为12345，服务器端端口号为23456，客户端端口号为54321：



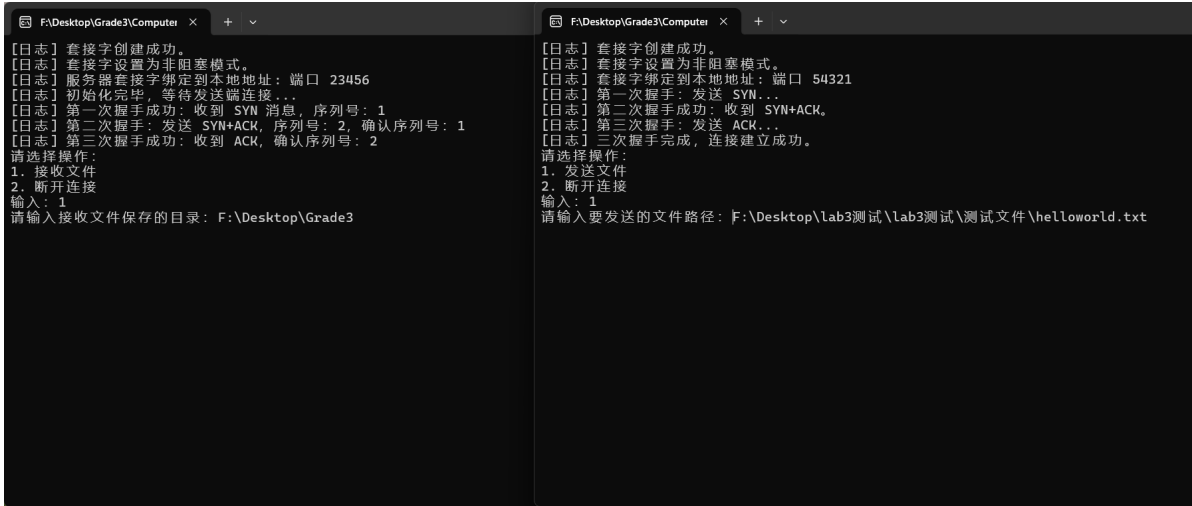
我们分别打开客户端和服务器端的程序，双方通过路由转发完成三次握手，同时给用户进行下一步操作的选择，输入1为发送或接收文件，输入2为断开连接：



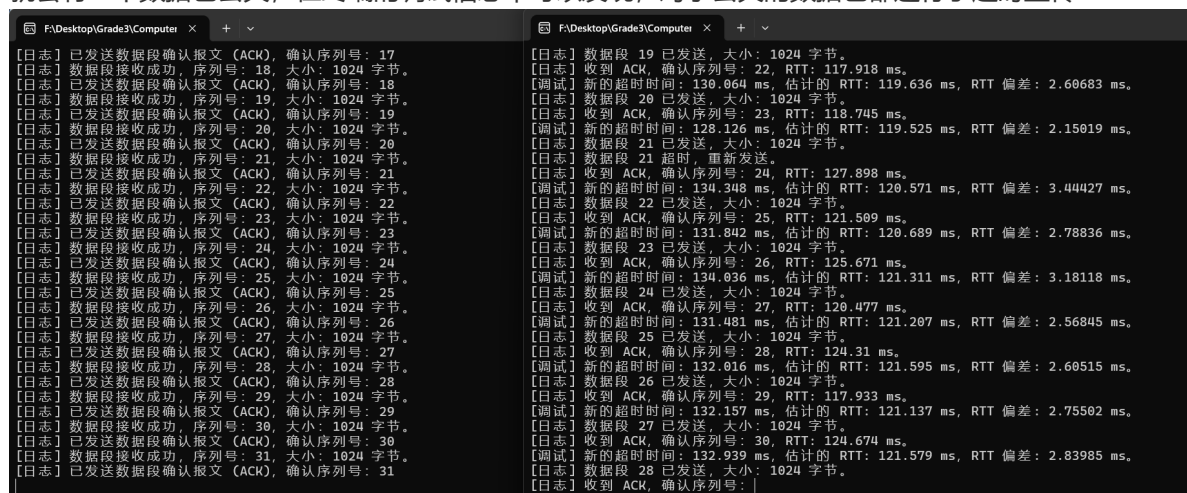
在wireshark中我们捕获到了本地回环中的udp数据包，由于中间通过了一次路由转发，所以三次握手的过程一共收到了捕获到了六个数据包：

正在捕获 Adapter for loopback traffic capture							
文件(E) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)							
udp							
No.	Time	Source	Destination	Protocol	Length	Info	
133	20.924102	127.0.0.1	127.0.0.1	UDP	1080	54321 → 12345	Len=1048
136	21.046014	127.0.0.1	127.0.0.1	UDP	1080	12345 → 23456	Len=1048
137	21.047299	127.0.0.1	127.0.0.1	UDP	1080	23456 → 12345	Len=1048
138	21.062252	127.0.0.1	127.0.0.1	UDP	1080	12345 → 54321	Len=1048
139	21.062592	127.0.0.1	127.0.0.1	UDP	1080	54321 → 12345	Len=1048
140	21.182840	127.0.0.1	127.0.0.1	UDP	1080	12345 → 23456	Len=1048

接下来我们在客户端和服务端选择发送和接收文件功能，选择指定文件路径：

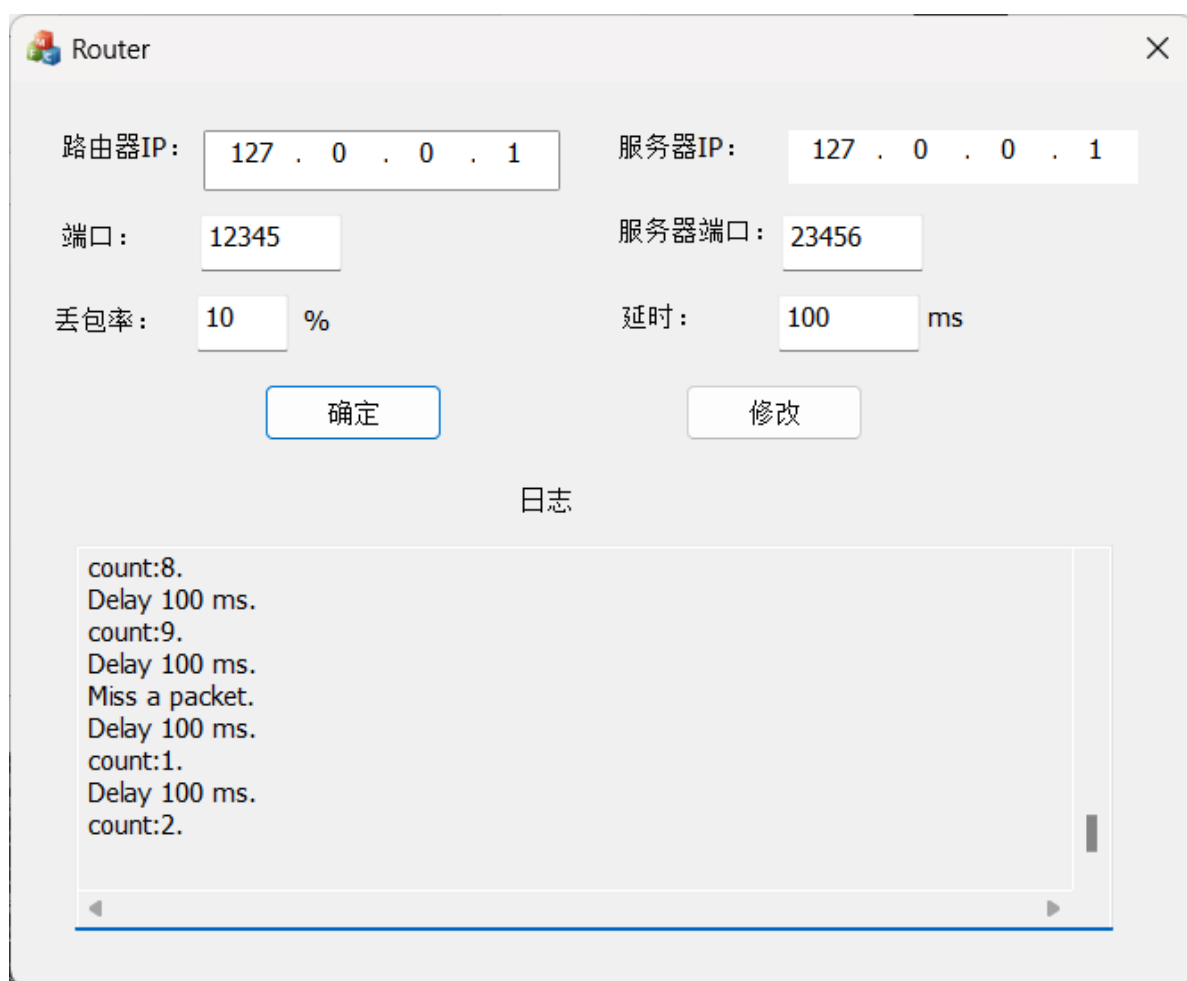


接下来开始传输文件，可以发现客户端每次会对超时时间进行重新估计，这样的做法可以让传输的效率大大提高，在路由程序中，我们将丢包率设置为了10%，可以在路由程序中可以看到，每发送9个数据包就会有1个数据包丢失，在终端的调试信息中可以发现，对于丢失的数据包都进行了超时重传：



```
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 17
[日志] 数据段接收成功, 序列号: 18, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 18
[日志] 数据段接收成功, 序列号: 19, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 19
[日志] 数据段接收成功, 序列号: 20, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 20
[日志] 数据段接收成功, 序列号: 21, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 21
[日志] 数据段接收成功, 序列号: 22, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 22
[日志] 数据段接收成功, 序列号: 23, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 23
[日志] 数据段接收成功, 序列号: 24, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 24
[日志] 数据段接收成功, 序列号: 25, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 25
[日志] 数据段接收成功, 序列号: 26, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 26
[日志] 数据段接收成功, 序列号: 27, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 27
[日志] 数据段接收成功, 序列号: 28, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 28
[日志] 数据段接收成功, 序列号: 29, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 29
[日志] 数据段接收成功, 序列号: 30, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 30
[日志] 数据段接收成功, 序列号: 31, 大小: 1024 字节。
[日志] 已发送数据段确认报文 (ACK), 确认序列号: 31

[日志] 数据段 19 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 22, RTT: 117.918 ms。
[调试] 新的超时时间: 130.064 ms, 估计的 RTT: 119.636 ms, RTT 偏差: 2.60683 ms。
[日志] 数据段 20 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 23, RTT: 118.795 ms。
[调试] 新的超时时间: 128.126 ms, 估计的 RTT: 119.525 ms, RTT 偏差: 2.15019 ms。
[日志] 数据段 21 已发送, 大小: 1024 字节。
[日志] 数据段 21 超时, 重新发送。
[日志] 收到 ACK, 确认序列号: 24, RTT: 127.898 ms。
[调试] 新的超时时间: 134.348 ms, 估计的 RTT: 120.571 ms, RTT 偏差: 3.44427 ms。
[日志] 数据段 22 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 25, RTT: 121.509 ms。
[调试] 新的超时时间: 131.842 ms, 估计的 RTT: 120.689 ms, RTT 偏差: 2.78836 ms。
[日志] 数据段 23 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 26, RTT: 125.671 ms。
[调试] 新的超时时间: 134.036 ms, 估计的 RTT: 121.311 ms, RTT 偏差: 3.18118 ms。
[日志] 数据段 24 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 27, RTT: 120.477 ms。
[调试] 新的超时时间: 131.481 ms, 估计的 RTT: 121.207 ms, RTT 偏差: 2.56845 ms。
[日志] 数据段 25 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 28, RTT: 124.31 ms。
[调试] 新的超时时间: 132.016 ms, 估计的 RTT: 121.595 ms, RTT 偏差: 2.69515 ms。
[日志] 数据段 26 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 29, RTT: 117.933 ms。
[调试] 新的超时时间: 132.157 ms, 估计的 RTT: 121.137 ms, RTT 偏差: 2.75502 ms。
[日志] 数据段 27 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 30, RTT: 124.674 ms。
[调试] 新的超时时间: 132.939 ms, 估计的 RTT: 121.579 ms, RTT 偏差: 2.83985 ms。
[日志] 数据段 28 已发送, 大小: 1024 字节。
[日志] 收到 ACK, 确认序列号: 31
```



Router

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

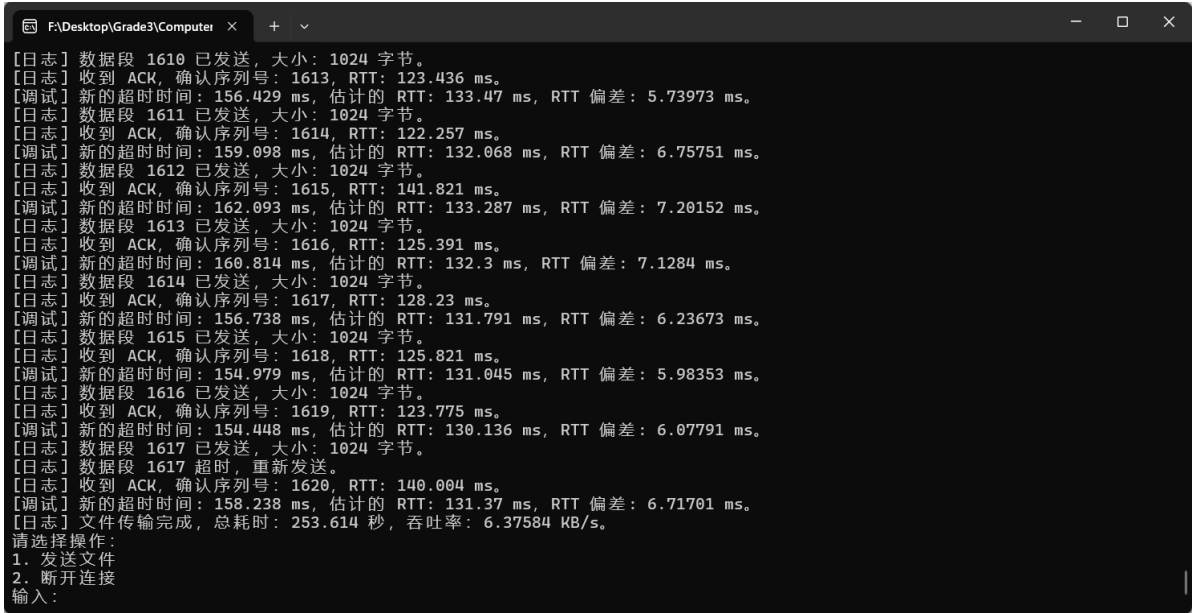
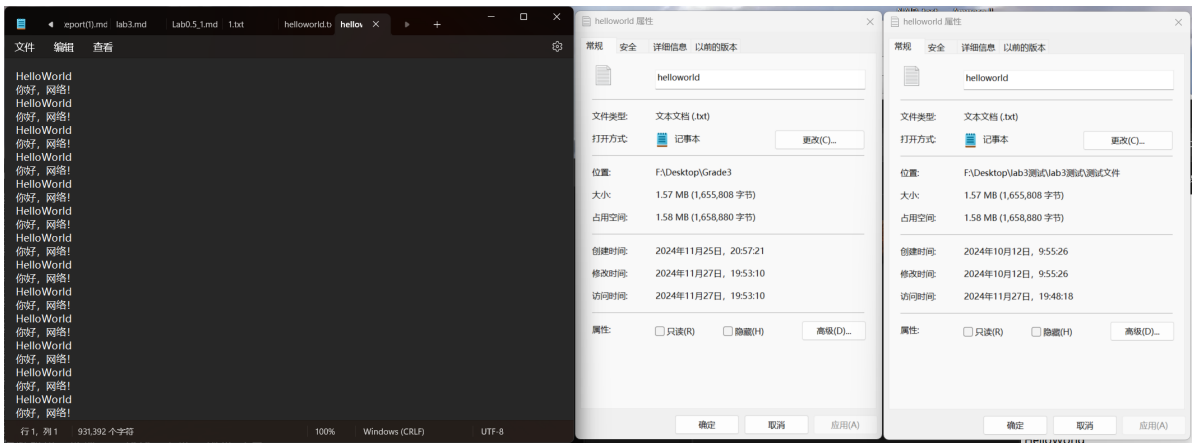
端口: 12345      服务器端口: 23456

丢包率: 10 %      延时: 100 ms

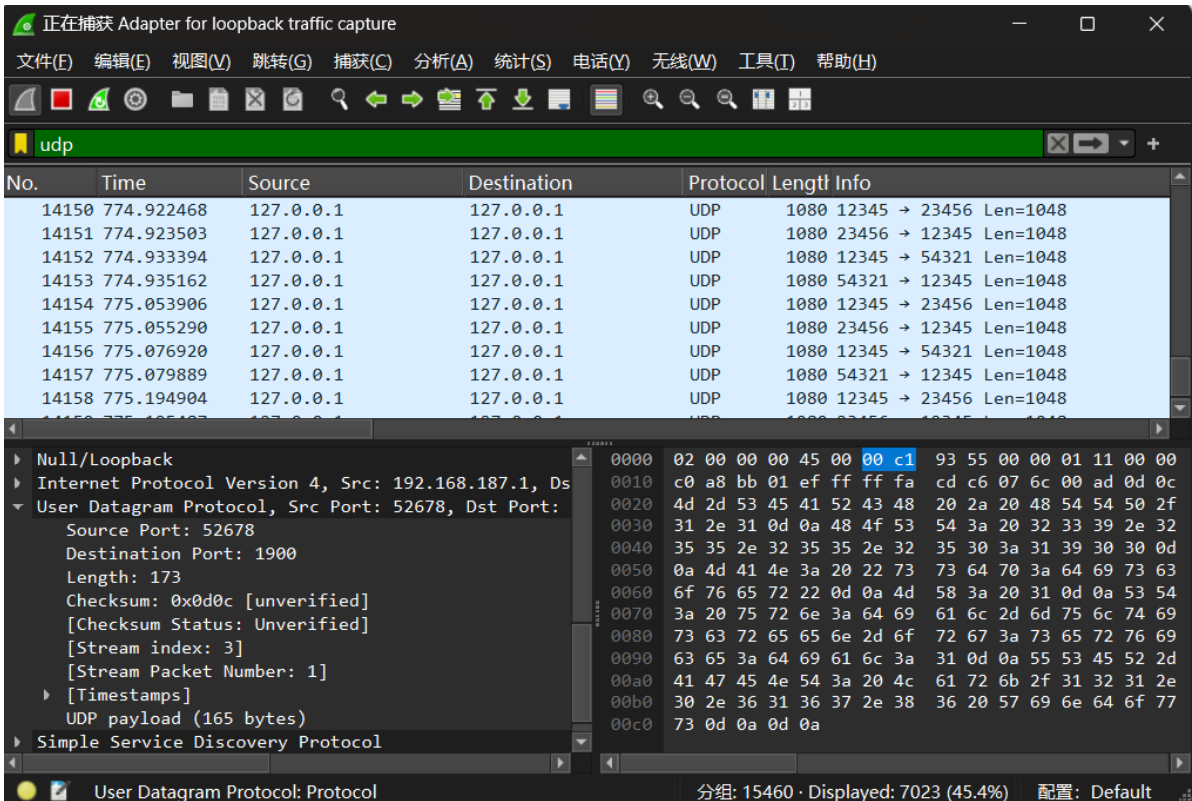
日志

```
count:8.
Delay 100 ms.
count:9.
Delay 100 ms.
Miss a packet.
Delay 100 ms.
count:1.
Delay 100 ms.
count:2.
```

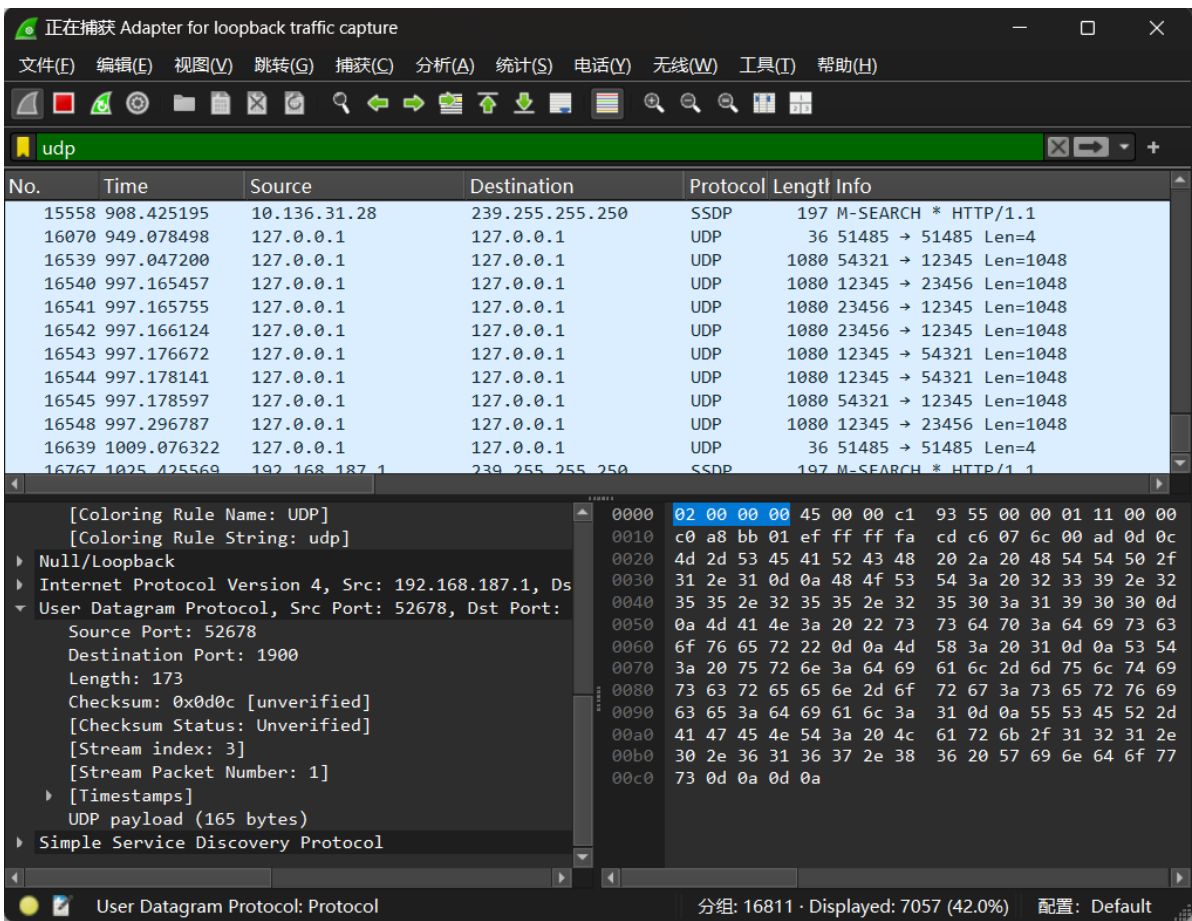
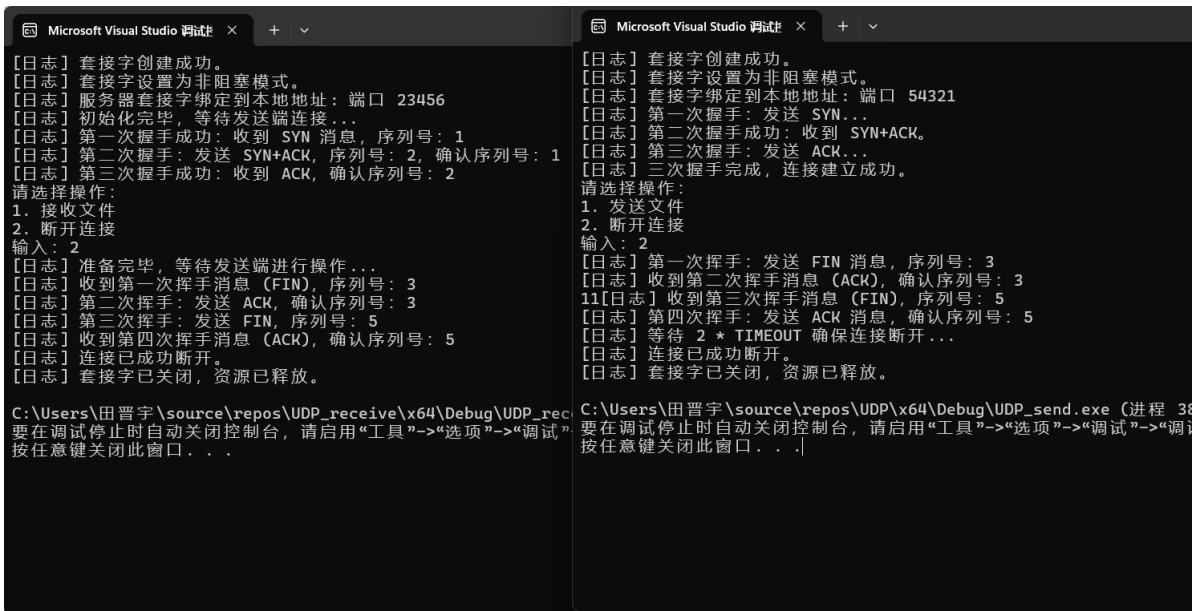
传输完毕后可以发现测试文件前后大小没有发生改变，且可以正常打开，同时在终端中展示了传输率，吞吐率等信息：



wireshark中捕获到的文件传输数据包：



接下来分别在两个窗口中选择断开连接功能，并在wireshark中捕获到了四次挥手的8个数据包：



## 总结与反思

### 为何使用超时重传机制？

我一开始未使用路由转发，直接在本地进行测试时，文件可以正常传输，后来加入了路由转发，模拟数据包在真实网络环境下的传输场景，设置了丢包率，在传输过程中有可能出现数据包丢失的现象，此时文件传输中途就会停下来，我对这个线下很疑惑，以为是某个数据段的内存分配有问题，后来仔细阅读了实验要求以及相关资料，认识到超时重传是一个重要的部分，如果没有超时重传，那么传输过程将在路由丢包的时候暂停，无法重新发送数据包，如果在有网络波动的场景下根本无法完成数据的传输。

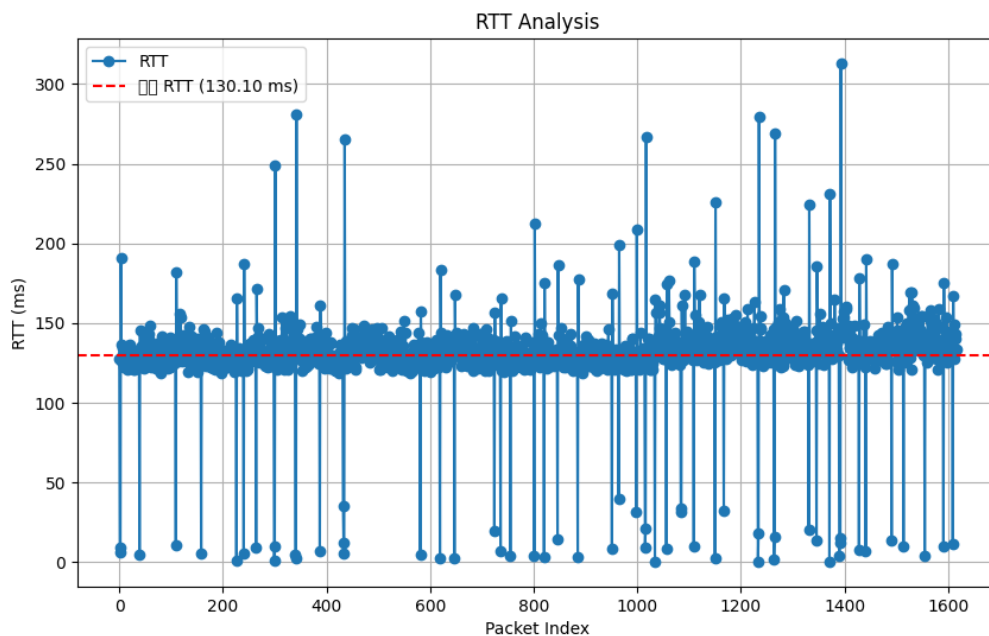
## 协议数据段的大小分配问题？

在对四次挥手断开连接部分进行调试的过程中，我发现一个很奇怪的bug，如果我设置了源端口和目的端口，那么FIN位将无法设置成功，会被客户端拒收，仔细查看数据包的内存分配，发现seq字段设置为16位，此时如果seq字段过大就会导致结构体内部内存溢出，使得FIN位被覆盖，无论如何都无法被设置。我调整了seq字段的内存大小，定义为32位，成功的解决了这个问题。

## 动态调整重传时间的实际作用？

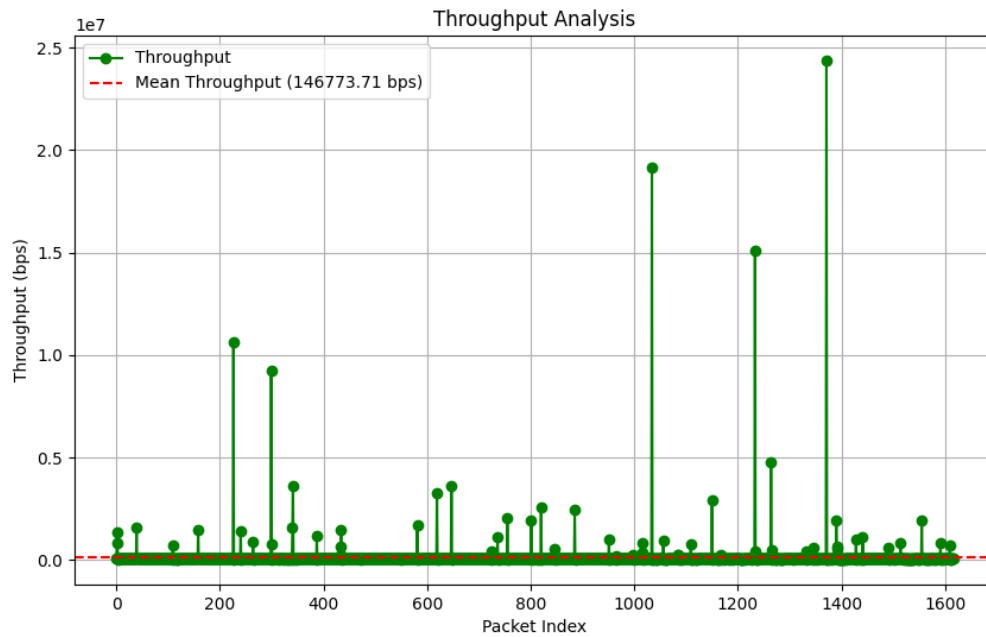
我将使用了动态调整重传时间机制的每个数据包的传送时间进行了记录，使用python进行数据分析，得到了如下的结果：

- 平均 RTT: 130.10 ms
- 最大 RTT: 312.98 ms
- 最小 RTT: 0.34 ms
- RTT 标准差: 26.72 ms



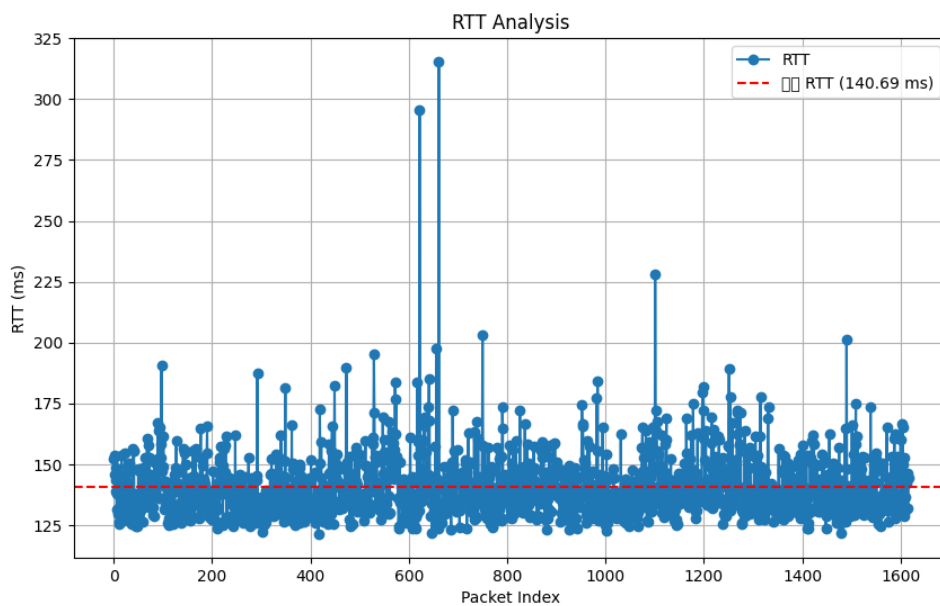
- 平均吞吐率: 146773.71 bps
- 最大吞吐率: 24373698.30 bps
- 最小吞吐率: 26174.11 bps
- 吞吐率标准差: 958977.19 bps





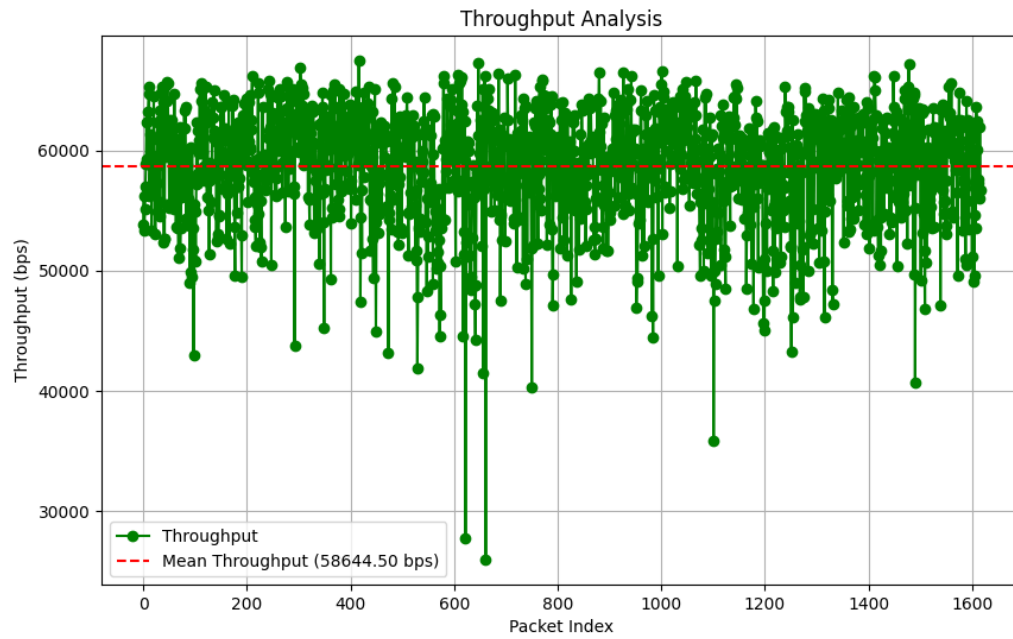
同时我也对未使用动态调整超时重传机制的传输延时进行了数据分析：

- 平均 RTT: 140.69 ms
- 最大 RTT: 315.40 ms
- 最小 RTT: 121.42 ms
- RTT 标准差: 13.10 ms



- 平均吞吐率: 58644.50 bps
- 最大吞吐率: 67467.74 bps
- 最小吞吐率: 25973.37 bps
- 吞吐率标准差: 4607.71 bps





动态调整超时重传机制有效应对了网络波动，维持较低的平均 RTT。但是静态机制下超时时间是固定的，因此对 RTT 的波动性不敏感。当网络波动较小时，超时时间固定会导致 RTT 数据看起来更平稳，因为超时不会过于频繁触发。而动态调整机制会灵敏响应 RTT 的变化，当 RTT 偶尔剧烈波动时会调整超时时间，导致 RTT 的动态波动更显著，反而增加了 RTT 的标准差。