基于UDP服务设计可靠传输协议

一、实验要求

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。

（1）实现单向传输。

（2）对于每个任务要求给出详细的协议设计。

（3）给出实现的拥塞控制算法的原理说明。

（4）完成给定测试文件的传输，显示传输时间和平均吞吐率。

（5）性能测试指标：吞吐率、文件传输时延，给出图形结果并进行分析。

（6）完成详细的实验报告。

（7）编写的程序应该结构清晰，具有较好的可读性。

（8）现场演示。

（9）提交程序源码、可执行文件和实验报告。

二、总体结构

服务端：监听端口，负责接收文件，并根据校验结果确认数据包的正确性，若发现错误则请求重传。
客户端：读取文件后，负责将文件分块后打包发送，监听确认响应，并在超时或失败时执行重传。

并实现了以下功能：

- 模拟三次握手和四次挥手，实现可靠连接。
- 采用包校验（CRC32）确保数据完整性。
- 支持超时重传，解决丢包问题。
- 支持连续文件传输，输出传输统计信息。

三、 各模块功能设计及实现

1. 传输协议模块

- **数据包结构：** 自定义数据包格式，包含包类型、序列号、数据长度、数据内容和校验和。
- **包类型定义：**

```
struct message {
    PacketType type;  // 包类型
    u_long seq;       // 序列号
    u_long ack;       // 确认号
    u_short len;      // 数据长度
    char data[BUF_SIZE]; // 数据
    uint32_t checksum;  // 校验和
};
```

2. 三次握手与四次挥手模块

1. **三次握手（连接建立）：**
   - 客户端发送 `SYN`，服务端回复 `SYN_ACK`。
   - 客户端确认并发送 `ACK`。
2. **四次挥手（连接断开）：**
   - 一方 a 发送 `FIN` 请求断开。
   - 另一方 b 回复 `FIN_ACK`，再次发送 `FIN`。
   - a接受后发送 `FIN_ACK`。
   - 双方断开连接。
   - 在我的代码中客户端发送完所有数据后主动断开连接。

代码实现

服务器与客户端逻辑相同，这里只展示服务器的代码实现

```
// ---------- 三次握手 ----------
// 接收 SYN 包
recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr *)&cliaddr, &cliaddr_len);
if (recvMsg.type == SYN) {
    cout << "[Handshake] Received SYN, seq=" << recvMsg.seq << ", ack=" << recvMsg.ack << er

    // 发送 SYN-ACK 包
    sendMsg.type = SYN_ACK;
    sendMsg.seq = 0;
    sendMsg.ack = recvMsg.seq + 1;
    sendMsg.checksum = calculateChecksum(sendMsg);
    sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (const struct sockaddr *)&cliaddr, cliaddr_

    cout << "[Handshake] Sent SYN-ACK, seq=" << sendMsg.seq << ", ack=" << sendMsg.ack << er

    // 接收 ACK 包
    recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr *)&cliaddr, &cliaddr_len
    if (recvMsg.type == ACK && recvMsg.ack == sendMsg.seq + 1) {
        cout << "[Handshake] Received ACK, seq=" << recvMsg.seq << ", ack=" << recvMsg.ack <
        cout << "连接成功!" << endl;
    } else {
        handleError("Failed to establish connection.");
    }
}
```

```
// ---------- 四次挥手 ----------
    // 接收 FIN 包
    recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr *)&cliaddr, &cliaddr_len);
    if (recvMsg.type == FIN) {
        cout << "[Teardown] Received FIN, seq=" << recvMsg.seq << ", ack=" << recvMsg.ack << enc

        // 发送 FIN-ACK 包
        sendMsg.type = FIN_ACK;
        sendMsg.seq = recvMsg.ack;
        sendMsg.ack = recvMsg.seq + 1;
        sendMsg.checksum = calculateChecksum(sendMsg);
        sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (const struct sockaddr *)&cliaddr, cliaddr_

        cout << "[Teardown] Sent FIN-ACK, seq=" << sendMsg.seq << ", ack=" << sendMsg.ack << enc

        // 发送 FIN 包
        cout << "[Teardown] Sent FIN, seq=" << sendMsg.seq << ", ack=" << sendMsg.ack << endl;

        // 接收最后的 ACK
        recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr *)&cliaddr, &cliaddr_len
        if (recvMsg.type == ACK) {
            cout << "[Teardown] Received FIN-ACK, seq=" << recvMsg.seq <<", ack=" << recvMsg.ack
            cout << "客户端断开连接..." << endl;
        } else {
            cerr << "[Teardown] Failed to terminate connection!" << endl;
        }
    }
```

3. 停等协议

停等协议是一种简单的可靠传输协议。其核心思想是在发送方发送一个数据包后，等待接收方的确认（ACK），在确认到达之前不会发送下一个数据包。通过这一机制可以确保：

- 每次传输的数据包都能被正确接收。
- 丢包或出错时，数据包会被重新发送。

代码实现

在客户端中，每个数据包发送后，客户端必须等待服务端的 `ACK`，只有收到正确的 `ACK`（确认号等于数据包序列号 + 1）后才能继续发送下一包数据。

```
while (true) {
    input_file.read(sendMsg.data, BUF_SIZE);
    sendMsg.len = input_file.gcount();
    sendMsg.seq = seq;  // 设置序列号
    sendMsg.checksum = calculateChecksum(sendMsg); // 计算校验和

    bool ackReceived = false;  // 确认是否收到ACK
    int retries = 0;           // 重传计数

    // 发送数据包后等待ACK
    while (!ackReceived && retries < MAX_RETRIES) {
        sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (struct sockaddr *)&serveraddr, serveraddr

        // 设置超时等待
        struct timeval timeout;
        timeout.tv_sec = TIMEOUT_SEC;
        timeout.tv_usec = TIMEOUT_USEC;
        setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));

        if (recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr *)&serveraddr, &serv
            // 判断是否是期望的ACK
            if (recvMsg.type == ACK && recvMsg.ack == sendMsg.seq + 1) {
                ackReceived = true;
                seq++; // 更新序列号
            }
        } else {
            retries++; // 超时，重传
        }
    }

    if (!ackReceived) {
        cerr << "Failed to receive ACK after " << MAX_RETRIES << " retries. Terminating transmis
        break;
    }

    // 传输完成判断
    if (input_file.eof()) {
        break;
    }
}
```

4. 超时重传功能

设计逻辑

超时重传功能是可靠传输协议的重要部分，用于应对丢包或确认包未到达的情况。在每次发送数据后，发送方会启动定时器。如果在设定时间内未收到ACK，发送方会认为数据包丢失，重新发送该包。

- **定时器：** 利用 `setsockopt` 设置接收超时。
- **重传机制：** 在超时时，重新发送上一次的数据包，直到收到正确的ACK或达到最大重传次数。

代码实现

在客户端的发送逻辑中实现超时重传：

1. 每次发送数据后，设置接收ACK的超时时间。
2. 若超时未收到ACK，重新发送数据包。
3. 若达到最大重试次数，退出传输。

```
while (!ackReceived && retries < MAX_RETRIES) {
    sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (struct sockaddr *)&serveraddr, serveraddr_len

    // 设置超时
    struct timeval timeout;
    timeout.tv_sec = TIMEOUT_SEC;
    timeout.tv_usec = TIMEOUT_USEC;
    setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));

    if (recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr *)&serveraddr, &serverad
        if (recvMsg.type == ACK && recvMsg.ack == sendMsg.seq + 1) {
            ackReceived = true;
        }
    } else {
        retries++; // 超时，重传
        cout << "Timeout! Resending packet with sequence number " << sendMsg.seq << endl;
    }
}
```

在服务端，处理接收到的包并及时发送ACK：

```
recvfrom(sockfd, &recvMsg, sizeof(recvMsg), 0, (struct sockaddr *)&cliaddr, &cliaddr_len);
if (calculateChecksum(recvMsg) == recvMsg.checksum) {
    // 校验成功，发送ACK
    sendMsg.type = ACK;
    sendMsg.ack = recvMsg.seq + 1;
    sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (struct sockaddr *)&cliaddr, cliaddr_len);
    // 保存数据
    output_file.write(recvMsg.data, recvMsg.len);
} else {
    // 校验失败，忽略数据包或请求重传
    cerr << "Checksum error. Ignoring packet with sequence number " << recvMsg.seq << endl;
}
```

停等协议确保了每次发送的一个数据包都被正确确认，避免了乱序与重复问题。超时重传功能则解决了丢包或确认丢失的情况。这两者结合实现了UDP可靠文件传输的核心机制。

- 正确的ACK到达后才继续传输，保证数据完整性。
- 超时未收到ACK时进行重试，提升可靠性。
- 最大重传次数限制避免死循环。

5. 校验和模块

采用CRC32一种循环冗余校验算法计算校验和，保障数据完整性。校验和覆盖数据包除校验和字段外的所有字节。

代码实现

(1) 生成 CRC-32 查找表

- 每个字节（0~255）被作为初始值，反复应用 CRC-32 多项式更新 8 次。
- 结果存储到数组 `table[i]` 中，供后续查表时使用。

```
constexpr std::array<uint32_t, 256> generateCRC32Table() {
    constexpr uint32_t polynomial = 0xEDB88320; // 标准 CRC-32 多项式
    std::array<uint32_t, 256> table{};

    for (uint32_t i = 0; i < 256; ++i) {
        uint32_t crc = i;
        for (uint32_t j = 0; j < 8; ++j) {
            if (crc & 1) {
                crc = (crc >> 1) ^ polynomial;
            } else {
                crc >>= 1;
            }
        }
        table[i] = crc;
    }
    return table;
}
```

(2) 计算 CRC 校验值

- 利用预生成的查找表对数据进行逐字节计算。
- 对数据的每个字节，通过表查找快速更新 CRC 值。
- 最终对结果按位取反得到 CRC-32 校验值。

```
uint32_t calculateChecksum(const message &msg) {
    uint32_t crc = 0xFFFFFFFF;  // 初始值为全 1
    const uint8_t *data = reinterpret_cast<const uint8_t *>(&msg);
    size_t length = sizeof(msg) - sizeof(msg.checksum);  // 排除 checksum 字段

    for (size_t i = 0; i < length; ++i) {
        crc = (crc >> 8) ^ crc32_table[(crc ^ data[i]) & 0xFF];  // 更新 CRC 值
    }

    return ~crc;  // 按位取反返回
}
```

四、功能展示

1. 握手连接

三次握手连接过程，包括每次传输的seq和ack值，连接后可以请求进行传输

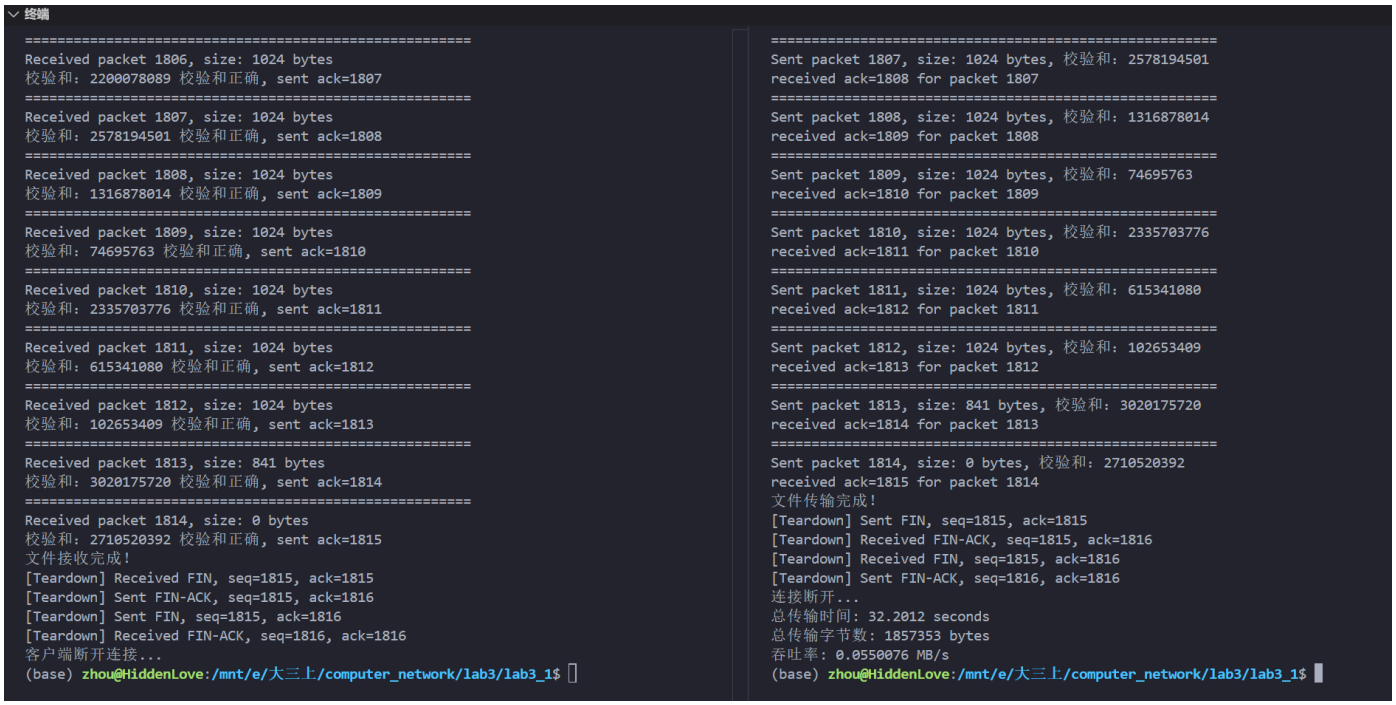
2. 文件传输

选择1.jpg进行传输

```
if(a == 'y'){
        // Transfer("send/helloworld.txt");
        Transfer("send/1.jpg");
        // Transfer("send/2.jpg");
        // Transfer("send/3.jpg");

        end = chrono::high_resolution_clock::now();
    }
```

- 显示客户端输出的每个数据包的序列号
- 计算两端同时计算校验和，服务器进行比较
- 显示服务器对每个数据包发送的确认ack
- 接受完成后客户端主动向服务器发送断开请求，完成四次挥手过程（与三次挥手逻辑相似）
- 在最后输出总传输时间、总传输字节数、吞吐率，显示性能测试指标

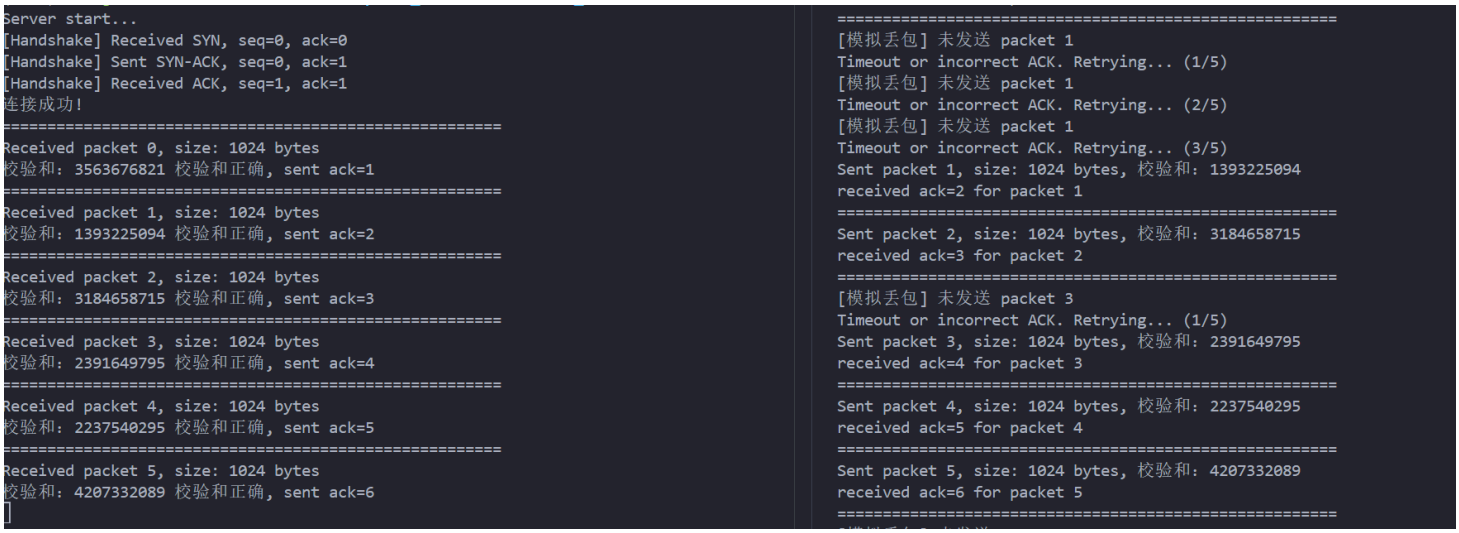


3. 验证超时重传

- 自行在程序中模拟丢包以及延迟情况
- 通过生成随机数，模拟随机丢包过程
- 可以设置丢包率大小
- 丢包时显示发送失败

```
#define PACKET_LOSS_RATE 0.5 // 丢包率
// 随机数生成器初始化
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> dis(0.0, 1.0);
```

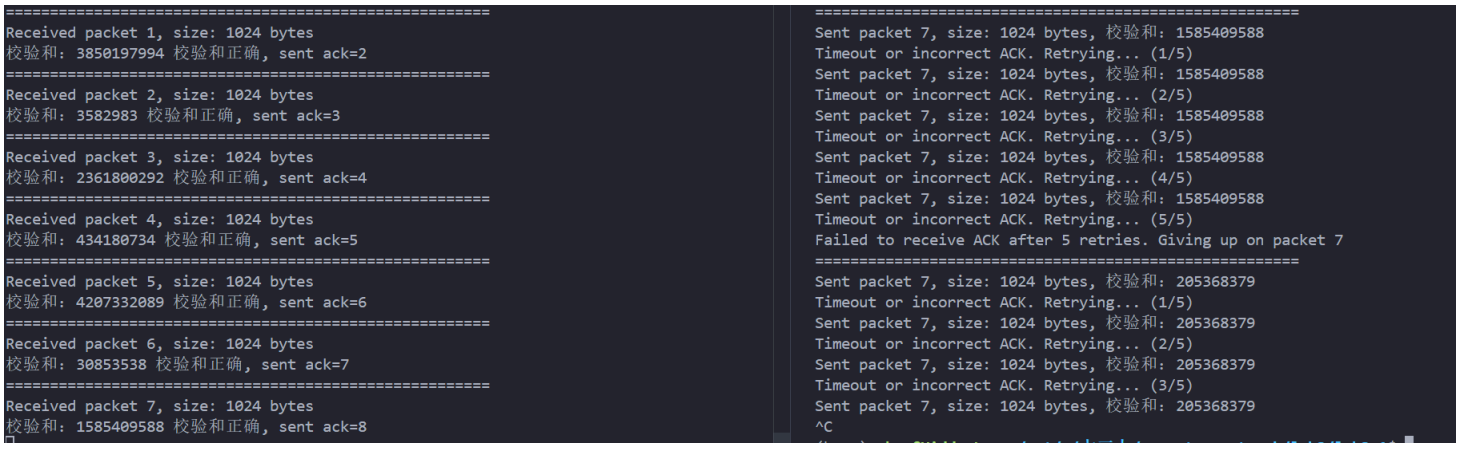
```
if (dis(gen) < PACKET_LOSS_RATE) {
    cout << "[模拟丢包] 未发送 packet " << sendMsg.seq << endl;
    } else {
        sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (const strucsockaddr *)&clientaddr, clienta
        cout << "Sent packet " << seq << ", size: " << sendMsg.len << bytes, 校验和：" << sendMs
    }
```



- 发生丢包后进行了客户端为接受到服务器的ack确认，进行重传
- 重传后直到服务器进行确认（有最大重传次数），发送下一个数据包

也可以为服务器设置延迟大小,在发送ack设置一定时间延迟

```
//模拟固定延迟
std::this_thread::sleep_for(std::chrono::milliseconds(1000)
// 发送 ACK
sendMsg.type = ACK;
sendMsg.ack = recvMsg.seq + 1;
sendto(sockfd, &sendMsg, sizeof(sendMsg), 0, (const struct sockad*)&cliaddr, cliaddr_len);
expectedSeq++;
```



- 客户端没有在规定时间接受到服务器发来的确认，进行重传
- 超出最大重传次数后略过当期数据包