南開大學

计算机网络实验报告

Lab3

学院：网络空间安全学院

专业：信息安全、法学

学号：2113203

姓名：付政烨

班级：信安法班

摘要

本实验旨在用户空间内利用 UDP 数据报套接字实现面向连接的可靠数据传输。在这个框架下，开发了一套包含建立连接、差错检测、接收确认和超时重传等功能的传输协议。特别地，实验采用了停等机制进行流量控制，并成功完成了指定测试文件的传输。在协议设计中，数据包的格式由 Packet_Header 结构体定义，其中包括数据大小、控制标志（如 SYN、ACK）、窗口大小、序列号、确认号和校验和等字段，以确保传输的可靠性和顺序性。建立连接的过程模仿了 TCP 的三次握手机制，确保双方在数据传输前建立稳定的连接。断开连接同样参考了 TCP 的四次挥手过程。差错检测机制通过校验和实现，能有效发现数据在传输过程中的任何变化。接收确认则通过 ACK 包实现，确保了每个数据包都被正确接收。这些机制共同保障了数据传输的可靠性。实验的一个关键亮点是超时重传机制的优化。传统的重传策略通常采用固定的超时时间，而本实验采用了指数退避算法和 Jacobson/Karels 算法，这两种方法在动态网络环境中表现更为优越。指数退避算法通过倍增超时时间来应对网络拥塞，有效降低了因重传引起的网络负担。而 Jacobson/Karels 算法则基于往返时间（RTT）的动态测量来调整超时时间，更精确地反映了当前的网络状况，减少了不必要的重传，从而提高了数据传输的效率和稳定性。

关键字：协议设计；　指数退避算法；　Jacobson/Karels 算法

目录

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、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

- 数据报套接字: UDP
- 协议设计: 数据包格式，发送端和接收端交互，详细完整
- 建立连接、断开连接: 类似 TCP 的握手、挥手功能
- 差错检验: 校验和
- 接收确认、超时重传: rdt2.0、rdt2.1、rdt2.2、rtd3.0 等，亦可自行设计协议
- 发送端、接收端单向传输:
- 日志输出: 收到/发送数据包的序号、ACK、校验和等，传输时间与吞吐率测试文件: 必须使用助教发的测试文件 (1.jpg、2.jpg、3.ipg、helloworld.txt)

二、 协议设计

(一) 报文格式

报文头部由 Packet_Header 结构体定义，总长 64 位，包含以下字段:

```
typedef struct Packet_Header
{
    WORD datasize;
    BYTE tag;
    BYTE window;
    BYTE seq;
    BYTE ack;
    WORD checksum;
}
```

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
datasize数据长度															
			END	OVER	FIN	ACK	SYN	window							
seq序列号								ack确认号							
checksum校验和															

图 1

- **datasize**：0-15 位，表示数据包中数据部分的大小。
- **tag**：16-23 位，包含控制标志，如 SYN、ACK 等，用于控制连接的建立、维持和断开。
- **window**：24-31 位，窗口大小，用于流量控制。

- **seq**：32-39 位，序列号，用于数据包排序和丢失恢复。
- **ack**：40-47 位，确认号，指示下一个期望接收的数据包序列号。
- **checksum**：48-63 位，校验和，用于差错检测。

(二) 建立连接（三次握手）

三次握手过程确保双方都准备好进行数据传输。过程如下：

1. 客户端发送一个带有 SYN 标志的数据包到服务器。代码中首先设置了 packet 的 tag 字段为 SYN，然后计算校验和，并通过网络发送。

```
packet.tag = SYN;
packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
sendto(socketClient, buffer.get(), sizeof(packet), 0,
       (sockaddr*)&servAddr, servAddrlen);
```

2. 服务器接收到 SYN 包后，回应一个带有 SYN 和 ACK 标志的数据包。在服务端，收到 SYN 后，会发送一个包含 SYN 和 ACK 标志的响应包。

```
if (packetReceived.tag == SYN) {
    packetToSend.tag = SYN | ACK;
    sendto(socketServer, &packetToSend, sizeof(packetToSend), 0,
           (struct sockaddr *)&clientAddr, sizeof(clientAddr));
}
```

3. 客户端收到服务器的 SYN-ACK 包后，发送一个带有 ACK 标志的包作为响应。这是三次握手的最后一步，确保客户端已经准备好接收数据。

```
if (packetReceived.tag == (SYN | ACK)) {
    packetToSend.tag = ACK;
    sendto(socketClient, &packetToSend, sizeof(packetToSend), 0,
           (sockaddr*)&serverAddr, sizeof(serverAddr));
}
```

(三) 差错检测

差错检测通过校验和实现的，旨在检测数据在传输过程中是否发生了变化。下面是详细介绍：

计算校验和

校验和的计算是通过 `compute_sum` 函数实现的。该函数执行以下步骤：

1. 准备数据：分配一个临时缓冲区，将数据包内容复制到其中。为保证数据长度为偶数，可能需要添加额外的零字节。
2. 累加：遍历缓冲区中的每个 16 位块，将它们累加。如有溢出，则将溢出部分加回总和中。
3. 取反：返回总和的补码作为校验和。

```
WORD compute_sum(WORD* message, int size) {
    int count = (size + 1) / 2;
    WORD* buf = (WORD*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, message, size);
    u_long sum = 0;
    while (count--) {
        sum += *buf++;
        if (sum & 0xffff0000) {
            sum &= 0xffff;
            sum++;
        }
    }
    return ~(sum & 0xffff);
}
```

校验数据包

当接收方收到数据包时，会重新计算校验和，并与数据包中的校验和进行比较。如果不匹配，则表明数据包可能已损坏。

```
memcpy(&packet, buffer.get(), sizeof(packet));
if (!(packet.tag == ACK && compute_sum((WORD*)&packet,
    sizeof(packet)) == 0)) {
    throw runtime_error("无法接收服务端回传ACK, 或校验和错误");
}
```

(四) 接收确认

接收确认机制确保了数据包正确地被接收方接收。这一过程是通过发送带有 ACK（Acknowledgement）标志的数据包来实现的。当接收方收到数据包后，它会检查序列号和校验和。如果数据包未损坏（即校验和正确），接收方会发送一个 ACK 包作为响应，其确认号设置为接收到的数据包序列号加一。以下是详细介绍：

```
Packet_Header packet;
memcpy(&packet, buffer.get(), sizeof(packet));
if (compute_sum((WORD*)&packet, sizeof(packet)) == 0) {
    packet.tag = ACK;
    packet.ack = packet.seq + 1;
    sendto(socketServer, &packet, sizeof(packet), 0,
           (sockaddr*)&clieAddr, clieAddrlen);
}
```

在这段代码中，首先使用 `memcpy` 函数将接收到的数据复制到 `packet` 结构中。然后，通过 `compute_sum` 函数计算校验和，以验证数据包的完整性。如果校验和正确，程序将设置 `packet` 的 `tag` 字段为 `ACK`，并将 `ack` 字段设置为 `seq` 字段的值加一。最后，使用 `sendto` 函数发送 ACK 包。

（五） 超时重传

在网络通信中，等待确认响应是确保数据可靠传输的重要机制。如果发送方在预定时间内没有收到来自接收方的确认，它将会重传数据包。这是通过在发送数据包后检测是否收到 ACK 响应来实现的。如果在设定的超时时间内没有收到 ACK，发送方将会重传该数据包。以下是实现等待确认响应的相关代码实现：

```
// 等待ACK响应
while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr
    , &servAddrlen) <= 0) {
    clock_t currentTime = clock();
    if ((currentTime - start) / CLOCKS_PER_SEC > timeoutDuration) {
        if (retryCount >= MAX_RETRY_COUNT) {
            throw runtime_error("重传次数超过限制，错误码: " + to_string(
                WSAGetLastError()));
        }
        // 重传数据包
        sendto(socketClient, buffer, sizeof(packet) + data_len, 0, (sockaddr
            *)&servAddr, servAddrlen);
        lastSendTime = clock();  // 更新上一次发送时间
        timeoutDuration *= 2;  // 指数退避，超时时间加倍
        retryCount++;  // 增加重传次数
    }
}
```

在这段代码中，‘recvfrom‘ 函数用于接收来自服务端的 ACK。如果在 ‘timeoutDuration‘内没有收到 ACK，就会执行重传逻辑。此外，使用了指数退避算法来增加超时时间，并限制了最大重传次数。

（六） 流量控制（停等机制）

停等机制是一种用于控制网络流量的基本方法，它确保发送方在接收到前一个数据包的确认之前不会发送下一个数据包，从而避免了网络拥塞。以下是停等机制的实现细节：在 `SendMessage` 函数中，发送方在发送一个数据包后，会等待接收方的确认响应。如果在设定的超时时间内没有收到确认，它将重传该数据包。

```
sendto(socketClient, buffer, sizeof(packet) + data_len, 0,
       (sockaddr*)&servAddr, servAddrlen);
while (recvfrom(socketClient, buffer, sizeof(packet), 0,
                (sockaddr*)&servAddr, &servAddrlen) <= 0) {
    ...
}
```

在这段代码中，`sendto` 函数用于发送数据包，而 `recvfrom` 函数用于接收确认响应。如果没有及时收到确认，将进入超时重传逻辑，这强制发送方停止发送下一个数据包，直到收到当前数据包的确认。

（七） 断开连接（四次挥手）

TCP 协议中的四次挥手过程用于终止连接，以下是过程的关键步骤：

1. 客户端发送一个带有 FIN_ACK 标志的数据包，开始断开连接的过程。

```
packet.tag = FIN_ACK;
memcpy(buffer.get(), &packet, sizeof(packet));
if (sendto(socketClient, buffer.get(), sizeof(packet), 0, (sockaddr*)&
    servAddr, servAddrlen) == -1) {
    throw runtime_error("sendto函数发送数据出错，错误码：" + to_string(
        WSAGetLastError()));
}
```

2. 服务器接收到这个 FIN 包后，回应一个带有 ACK 标志的数据包，确认收到断开连接的请求。

```
packet.tag = ACK;
memcpy(buffer.get(), &packet, sizeof(packet));
if (sendto(socketServer, buffer.get(), sizeof(packet), 0, (sockaddr*)&
    clieAddr, clieAddrlen) == -1) {
    throw runtime_error("发送ACK失败，错误码：" + to_string(WSAGetLastError
        ()));
}
```

3. 服务器随后发送一个带有 FIN_ACK 标志的数据包，表示准备好断开连接。

```
packet.tag = FIN_ACK;
memcpy(buffer.get(), &packet, sizeof(packet));
if (sendto(socketServer, buffer.get(), sizeof(packet), 0, (sockaddr*)&
    clieAddr, clieAddrlen) == -1) {
    throw runtime_error("发送FIN_ACK失败，错误码：" + to_string(
        WSAGetLastError()));
}
```

4. 客户端收到这个 FIN 包后，回应一个带有 ACK 标志的数据包，完成断开连接的过程。

```
packet.tag = ACK;
memcpy(buffer.get(), &packet, sizeof(packet));
if (sendto(socketClient, buffer.get(), sizeof(packet), 0, (sockaddr*)&
    servAddr, servAddrlen) == -1) {
    throw runtime_error("发送第四次挥手数据失败，错误码：" + to_string(
        WSAGetLastError()));
}
```

在这个过程中，每个步骤都伴随着适当的数据包发送和接收，确保了连接的可靠和有序地关闭。

(八) 优化部分

相比于普通的固定超时重传算法，指数退避和 Jacobson/Karels 算法提供了更为动态和高效的超时时间调整机制。它们能够根据网络环境的实际情况调整重传策略，从而优化网络性能，减少不必要的网络负担，并更好地应对网络延迟和拥塞。这在动态变化的网络环境中尤为重要。

1. 指数退避算法

指数退避算法在重传机制中起着关键作用，当重传发生时，超时时间根据以下公式进行调整：

$$TimeoutDuration = TimeoutDuration \times 2$$

指数退避算法通过减少重传频率来降低网络上的数据包数量。在下述代码中，这一算法体现在重传逻辑中。如果数据包在预定时间内未收到确认，超时时间将翻倍，随后进行数据包的重传。下面是实现这一算法的代码片段：

```
clock_t start = clock();
int retryCount = 0;
int timeoutDuration = 1;

while (recvfrom(socketServer, buffer.get(), sizeof(packet), 0, (sockaddr*)&
    clieAddr, &clieAddrlen) <= 0) {
    if ((clock() - start) / CLOCKS_PER_SEC > timeoutDuration) {
        retryCount++;
        if (retryCount > MAX_RETRY_COUNT) {
            throw runtime_error("重传次数超过限制，错误码: " + to_string(
                WSAGetLastError()));
        }
        // 重传逻辑
        if (sendto(socketServer, buffer.get(), sizeof(packet), 0, (sockaddr*)
            &clieAddr, clieAddrlen) == -1) {
            throw runtime_error("重传FIN_ACK失败，错误码: " + to_string(
                WSAGetLastError()));
        }
        cout << "重传第三次挥手信息，重传次数:␣" << retryCount << endl;
        start = clock();
        timeoutDuration *= 2;
    }
}
```

2. Jacobson/Karels 算法

该算法通过动态调整基于往返时间 (RTT) 的超时时间来优化重传。算法如下：

- 估计 RTT：$EstimatedRTT = (1 - \alpha) \times EstimatedRTT + \alpha \times SampleRTT$。
- 计算偏差：$DevRTT = (1 - \beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT|$。
- 计算超时时间：$TimeoutInterval = EstimatedRTT + 4 \times DevRTT$。

其中，α 和 β 是权重因子。在下述代码中，这个算法主要体现在客户端与服务端建立连接的过程。关键代码如下：

```
double estimatedRTT = 1.0;
double devRTT = 0.0;
const double alpha = 0.125;
const double beta = 0.25;
```

```
5
6  double sampleRTT = double(clock() - start) / CLOCKS_PER_SEC;
7  estimatedRTT = (1 - alpha) * estimatedRTT + alpha * sampleRTT;
8  devRTT = (1 - beta) * devRTT + beta * abs(sampleRTT - estimatedRTT);
9
10 double timeoutDuration = estimatedRTT + 4 * devRTT;
```

三、 程序设计

(一) client

本节详细介绍了在网络编程中 client 端的实现流程，因为这部分代码是实验中的核心，包括网络连接的初始化、数据发送与接收的处理，以及连接的终止。

1. 网络初始化

首先，程序进行网络环境的初始化，包括加载 Winsock 库和创建 UDP 套接字。

```
1 WSADATA wsaData;
2 int err = WSAStartup(MAKEWORD(2, 2), &wsaData);
3 SOCKET client = socket(AF_INET, SOCK_DGRAM, 0);
```

2. 服务器地址设置

程序设置了服务器的地址信息，包括协议族、端口号和 IP 地址。

```
1 struct sockaddr_in serveraddr;
2 serveraddr.sin_family = AF_INET;
3 serveraddr.sin_port = htons(PORT);
4 inet_pton(AF_INET, IP, &serveraddr.sin_addr.s_addr);
```

3. 连接建立与数据传输

客户端程序根据用户输入的标签（label）决定是发送数据还是接收数据。如果输入为 0，则程序进入发送数据模式；若非 0，则进入接收数据模式。

发送数据模式 在发送数据模式下，程序首先提示用户输入要发送的文件名。如果用户输入'q'，则发送一个全局结束标志给服务器，结束程序。否则，程序会尝试以二进制模式打开指定的文件。

```
1 if (label == 0) {
2     while (true) {
3         ...
4         ifstream file(InFileName, ifstream::binary);
5         if (!file) {
6             cout << "文件打开失败! " << endl;
7             continue;
8         }
9         ...
```

```
        file.read(FileBuffer.get(), F_length);
        ...
    }
}
```

在二进制模式下，文件以原始的字节形式读取，保证了数据的完整性。程序首先通过 'seekg' 和 'tellg' 函数计算文件的大小，然后使用 'read' 函数读取整个文件到一个字符数组中。接着，程序调用 'SendMessage' 函数发送文件名和文件内容，计算传输所需的时间和吞吐率。

接收数据模式 在接收数据模式下，程序持续等待接收文件名和文件内容。使用 'RecvMessage' 函数接收数据，然后将接收到的文件内容写入到新文件中。

```
else {
    while (true) {
        ...
        ofstream file(fileName, ofstream::binary);
        if (!file) {
            cout << "文件打开失败! " << endl;
            continue;
        }
        file.write(Message.get(), file_len);
        ...
    }
}
```

程序以二进制模式创建新文件，并使用 'write' 函数将接收到的数据写入文件中。这样可以确保原始数据的完整性和准确性，无论接收到的是文本还是二进制数据。

4. 连接断开

无论是发送还是接收操作完成后，程序通过 'Client_Server_Disconnect' 函数终止与服务器的连接。

```
Client_Server_Disconnect(client, serveraddr, length);
```

5. 清理资源

最后，程序清理分配的资源，包括关闭套接字和卸载 Winsock 库。

```
closesocket(client);
WSACleanup();
```

本程序展示了如何使用 C++ 和 Winsock API 在 UDP 协议上实现类似 TCP 的可靠数据传输机制。通过精心设计的程序流程，我们能够在不可靠的 UDP 上实现可靠的数据传输。

(二) server

由于客户端和服务端之间有大量功能类似的代码，为了节省篇幅，本节将专注于介绍服务端程序中与客户端不同的部分。

1. 绑定套接字

与客户端不同，服务端在创建套接字后需要将其绑定到一个特定的端口上，以便监听来自客户端的连接请求。

```
SOCKADDR_IN addr;
bind(server, (SOCKADDR*)&addr, sizeof(addr));
```

2. 接收数据

在服务端，主要任务是接收来自客户端的数据。服务端在一个循环中等待客户端的数据并处理接收。

```
while (true) {
    int name_len = RecvMessage(server, addr, length, ...);
    // 接收数据逻辑
}
```

3. 保存接收的数据

服务端在接收数据后，需要将数据保存到文件系统中，这是服务端独有的步骤。

```
ofstream file_stream(filename, ios::binary);
file_stream.write(Message.get(), file_len);
```

服务端程序的关键在于设置监听端口、处理接收的数据并保存到文件系统。与客户端相比，服务端更专注于管理来自多个客户端的连接和数据流。

四、 丢包与延时设计

在本段代码中，通过设置丢包率和延时参数，模拟了网络环境中的不确定性。这有助于测试协议在真实世界条件下的健壮性。

(一) 丢包函数

丢包函数通过生成一个随机数来决定是否丢弃某个数据包。如果生成的随机数小于设定的丢包率，则丢弃该包。

```
bool shouldDropPacket() {
    random_device rd;
    mt19937 gen(rd());
    uniform_real_distribution<> dis(0, 1);
    return dis(gen) < PACKET_LOSS_RATE;
}
```

(二) 延迟函数

延迟函数通过使线程暂停特定的时间来模拟网络延迟。

```
void delayPacket() {
    this_thread::sleep_for(chrono::milliseconds(PACKET_DELAY_MS));
}
```

(三) SendMessage 函数中的实现

'SendMessage' 函数中集成了以上的丢包和延迟处理，以及指数退避算法的测试。

1. 数据发送与丢包模拟

在发送数据前，函数先执行延迟处理。然后，根据丢包函数的结果决定是否发送数据包。

```
delayPacket();
bool packetDropped = shouldDropPacket();
if (!packetDropped) {
    sendto(socketClient, buffer, sizeof(packet) + data_len, 0, (sockaddr*)&
        servAddr, servAddrlen);
}
```

2. 指数退避算法测试

指数退避算法是一种关键的网络通信机制，用于处理超时重传的情况。在本代码中，这一算法的测试对于验证网络协议在面对不稳定网络环境时的鲁棒性至关重要。当发送的数据包在预定的超时时间内未收到确认响应时，发送方需要进行数据包的重传。这种情况可能是由于网络延迟或数据包丢失造成的。

```
while (recvfrom(socketClient, buffer, sizeof(packet), 0, (sockaddr*)&servAddr
    , &servAddrlen) <= 0) {
    if ((currentTime - start) / CLOCKS_PER_SEC > timeoutDuration) {
        // 重传逻辑
        ...
    }
}
```

指数退避算法通过将超时时间翻倍来应对网络拥塞，避免在网络状况不佳时频繁的数据包重传。这减少了网络拥塞的可能性，并提高了数据传输的效率。

```
int retryCount = 0;
int timeoutDuration = 1; // 初始超时时间为1秒

while (...) {
    if ((currentTime - start) / CLOCKS_PER_SEC > timeoutDuration) {
        // 重传数据包
        sendto(socketClient, buffer, sizeof(packet) + data_len, 0, (sockaddr
            *)&servAddr, servAddrlen);
        timeoutDuration *= 2; // 指数退避
        retryCount++;
    }
}
```

在代码中，特别设置了一个 ‘TestFlag‘ 用于测试指数退避算法。通过限制重传次数，可以观察指数退避算法在实际应用中的表现。

```
int TestFlag = 0;

while (...) {
    if (TestFlag < 4) {
        // 测试指数退避算法
        TestFlag++;
        timeoutDuration *= 2; // 指数退避
        cout << "测试指数退避算法，当前超时时间:␣" << timeoutDuration << "␣秒
            " << endl;
    } else {
        // 正常重传逻辑
        ...
    }
}
```

在测试过程中，‘TestFlag‘ 的增加将导致超时时间的指数级增长，从而模拟在实际网络环境中可能遇到的拥塞情况。

指数退避算法的测试展示了在面对不稳定的网络条件时，如何有效地处理数据包重传问题。通过这种方法，我们可以确保即使在网络状况不佳的情况下，协议也能够有效地运行，减少数据包的丢失，并优化整体的网络性能。通过集成丢包和延迟处理，以及指数退避算法的测试，此代码模拟了真实网络环境的不稳定性。这样的设计不仅提高了协议的健壮性，也验证了其在不同网络条件下的有效性。

五、 实验结果

1. 指数退避重传测试

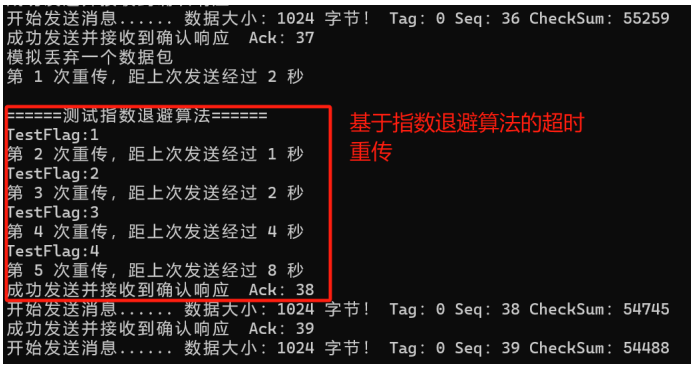

图 2

2. 三次握手 + 发送/接收文件 + 动态重传测试

图 3

3. 丢包模拟

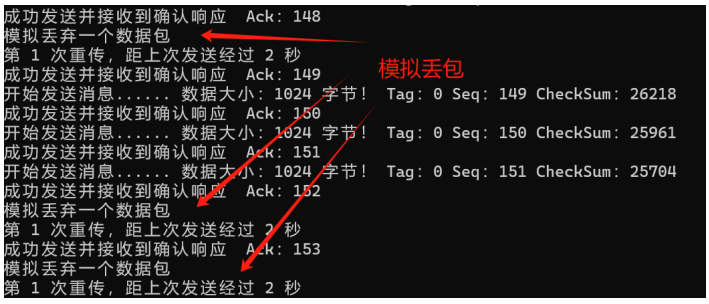

图 4

4. 四次挥手

图 5

5. 文件传输成功

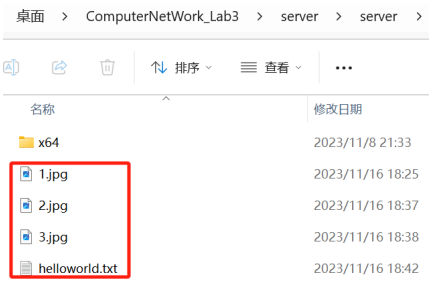

图 6

六、 实验心得与体会

通过本次实验，我深刻体会到了在用户空间使用 UDP 实现面向连接的可靠数据传输协议的挑战与价值。此次实验不仅增强了我对计算机网络基础概念的理解，还让我领会到了实际应用中的复杂性和实现细节的重要性。首先，在设计和实现过程中，我对 UDP 和 TCP 的基本工作原理有了更深入的认识。尽管 UDP 本身是无连接且不可靠的，但通过在应用层实现类似 TCP 的机制，如三次握手建立连接、四次挥手断开连接，以及 ACK 确认和重传机制，我成功地在 UDP

之上构建了一个可靠的传输协议。这个过程让我意识到，即使是基于简单协议的构建，也可以实现复杂且强大的功能。差错检测的实现特别让我印象深刻。通过在数据包中引入校验和，我能够有效地检测并应对传输过程中的错误。这不仅增强了数据传输的可靠性，还让我对数据完整性的维护有了更实际的理解。此外，超时重传机制的优化是我在本实验中最为重视的部分。通过比较传统的固定超时重传方法和更先进的指数退避算法以及基于 RTT 的 Jacobson/Karels 算法，我深刻体会到了动态调整策略在应对网络变化时的重要性。尤其是 Jacobson/Karels 算法，它根据网络的实际状况动态调整超时时间，有效地减少了不必要的重传，并提高了传输效率。最后，本次实验还提升了我的编程能力和问题解决技巧。面对复杂的需求和潜在的问题，我学会了如何逐步分解问题、设计算法，并实现有效的解决方案。整个过程不仅是对我的技术知识的考验，也锻炼了我的逻辑思维和创新能力。

参考文献