Lab3-1

1. 问题描述

在本次实验中，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。

基于给定的实验要求，在实验中，由于实现的是单向的通信，即只从服务器发送给客户端，服务器作为发送端，客户端作为接收端。
因此在建立连接和断开连接的过程中，不适合使用TCP的三次握手和四次挥手，在实验中对建立连接和断开连接的过程进行了简化。
建立连接和断开连接的过程采用了两次握手和两次挥手的方式。
在数据传输的过程中，基于rdt3.0的可靠传输协议，实现了确认重传的功能。

2. 报文设计

在本次实验中，我们的报文仿照TCP数据报的格式进行设计。
其中包含了发送端和接收端的端口号，确认号，序列号，标志位，长度，校验和以及数据段，整体按照16位进行对齐。

```
struct Message {
    unsigned short sourcePort{};
    unsigned short destinationPort{};
    int ack{};
    int seq{};
    unsigned short flagAndLength = 0;
    unsigned short checksum{};
    char data[MSS]{};
};
```

其中对于标志位的设计如下，由于各个数据段要按照16位对齐，因此将标志位和长度合并到同一个16位中，其中高五位为标志位，低12位表示数据段的长度。

```
| 15  | 14  | 13  | 12  | 11  |  10-0  |
| REP | FIN | SYN | ACK | FHD |  LEN   |
```

- 第15位表示当前的数据包是否为重传数据包
- 第14位表示FIN标志位
- 第13位表示SYN标志位
- 第12位表示ACK标志位
- 第11位表示当前的数据包的数据段是否为文件头信息
- 第10-0位表示数据段的长度

对于伪首部，同样参考了TCP的伪首部设计，其中包含了发送端和接收端的IP地址，zero，协议号，以及数据报的长度。

```
struct PseudoHeader {
    unsigned long sourceIP{};
    unsigned long destinationIP{};
    char zero = 0;
    char protocol = 6;
    short int length = sizeof(struct Message);
};
```

可靠数据传输的实现

在本次实验中，基于rdt3.0协议实现了可靠数据传输。只是用两个序列号，完成对于数据包的发送和确认的工作。
为了方便起见，在本次实验中只实现了单向发送的功能，即服务器作为发送端，使用发送端的状态机。
客户端作为接收端，使用接收端的状态机。
接下来分别从接收端和发送端两方面介绍可靠数据传输的实现细节。

校验和

为了保证数据在传输过程中没有发生错误，或者是检查数据在传说过程中是否发生了错误，在实验中仿照UDP的校验和完成了差错检验。
这里的实现原理同课内理论完全一致，就不再赘述，直接上代码。

```
void Message::setChecksum(struct PseudoHeader *pseudoHeader) {
    this->checksum = 0;
    unsigned long long int sum = 0;
    for (int i = 0; i < sizeof(struct PseudoHeader) / 2; i++) {
        sum += ((unsigned short int *) pseudoHeader)[i];
    }
    for (int i = 0; i < sizeof(struct Message) / 2; i++) {
        sum += ((unsigned short int *) this)[i];
    }
    while (sum >> 16) {
        sum = (sum & 0xffff) + (sum >> 16);
    }
    this->checksum = ~sum;
}

bool Message::checksumValid(struct PseudoHeader *pseudoHeader){
    unsigned long long int sum = 0;
    for (int i = 0; i < sizeof(struct PseudoHeader) / 2; i++) {
        sum += ((unsigned short int *) pseudoHeader)[i];
    }
    for (int i = 0; i < sizeof(struct Message) / 2; i++) {
        sum += ((unsigned short int *) this)[i];
    }
    while (sum >> 16) {
        sum = (sum & 0xffff) + (sum >> 16);
    }
    return sum == 0x000ffff;
```

```
}
```

超时重传

当发送端发送了一条报文之后，需要拉起一个超时重传的工作线程，等待RTO的时间，如果在这个时间内没有收到对应的ACK，则需要重传数据包。
如果收到了ACK，则可以结束线程了。而如果提前接收到了ACK，则可以直接终止超时重传的工作线程。
因此这里并没有采用简单的sleep方法，而是采用分段sleep的方法，以便能够提前结束。

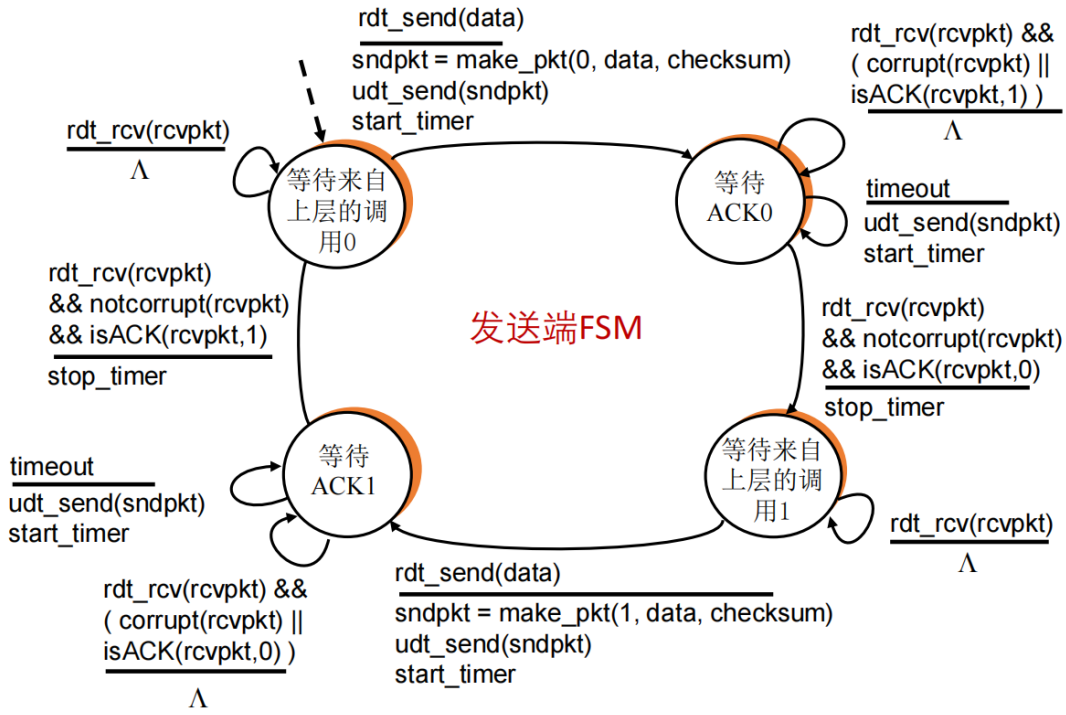
```
int waitTime = 0;
ACK_FLAG = false;
std::thread resendThread([&](){
    // if ACK_FLAG is not set, re-send
    while(!ACK_FLAG){
        while(waitTime < RTO && !ACK_FLAG) {
            Sleep(1);
            waitTime += 1;
        }
        if(!ACK_FLAG) {
            std::cout << "[Timeout] : Re-send Package" << std::endl;
            if(!randomLoss()){
                sendto(serverSocket, (char*) &message, sizeof(struct Message), 0, (struct sockaddr*) &clientAddress, sizeof(SOCKADDR_IN));
            }
            printMessage(message);
        }
    }
});
```

发送端

在发送端使用两个序列号和四个状态实现了可靠数据传输。其具体流程如下

初始状态的时候，发送端的序列号为0，发送端将报文发送出去之后，就进入等待ACK0的状态中，等待接收端回应的ACK。
在这个阶段中，需要开启一个超时重传计时器，根据设定的RTO，超时未接收到ACK时就会重新发送数据包。
在接收到接收端回应的ACK后，会判断ACK中的确认号是否和当前的序列号一致，如果一致则说明之前的数据包发送成功，可以进入下一个发送阶段，此时序列号需要发生反转。
如果接收到的ACK中确认号和当前的序列号不一致，则说明之前的数据包发送失败了，因此发送端需要重新发送数据包。在rdt3.0中并不是立刻发送的，而是等待超时重传。
进入下一状态的序列号之后，整体的流程和之前是一致的，同样是等待当前序列号的确认号，否则就触发超时重传。

■ rdt3.0：发送端状态机



具体代码实现如下

```
/**
 * send a package to client
 * @param message message in package
 */
void sendPackage(struct Message message) {
    // send package
    if(!randomLoss()){
        sendto(serverSocket, (char*) &message, sizeof(struct Message), 0, (struct sockaddr*) &clientAddress, sizeof(SOCKADDR));
    }
    printMessage(message);
    // enter wait for ACK state
    // first start timer for timeout re-send
    int waitTime = 0;
    ACK_FLAG = false;
    std::thread resendThread([&](){
    // if ACK_FLAG is not set, re-send
    while(!ACK_FLAG){
        while(waitTime < RTO && !ACK_FLAG) {
            Sleep(1);
            waitTime += 1;
        }
        if(!ACK_FLAG) {
            std::cout << "[Timeout] : Re-send Package" << std::endl;
            if(!randomLoss()){
                sendto(serverSocket, (char*) &message, sizeof(struct Message), 0, (struct sockaddr*) &clientAddress, sizeof(SOCKADDR_IN));
            }
```

```
            printMessage(message);
        }
    }
    });
    // then wait for ACK
    while(true) {
        struct Message recvBuffer;
        int clientAddressLength = sizeof(SOCKADDR);
        int recvLength = recvfrom(serverSocket, (char *) &recvBuffer, sizeof(struct Message), 0, (struct sockaddr *) &clientAddress, &clientAddressLength);
        // recvLength > 0 means receive success
        if (recvLength > 0) {
            printMessage(recvBuffer);
            // check checksum and ack
            // only if checksum is valid and ack is for the current seq that the package is sent and received correctly
            if (recvBuffer.checksumValid(&recvPseudoHeader) && recvBuffer.ack == CURRENT_SEQ) {
                ACK_FLAG = true;
                resendThread.join();
                // update current seq
                CURRENT_SEQ = (CURRENT_SEQ + 1) % 2;
                // current send task finish
                std::cout << "[ACK] : Package (SEQ:" << recvBuffer.ack << ") sent successfully!" << std::endl;
                break;
            }
            // if ckecksum is not valid or ack is not for current seq, that means the packet sent just now was failed
            // then wait for timeout to re-send the package
            else {
                std::cout << "[RE] : Client received failed. Wait for timeout to re-send" << std::endl;
            }
        }
        else {
            std::cout << "[ERROR] : Package received from socket failed!" << std::endl;
        }
    }
}
```

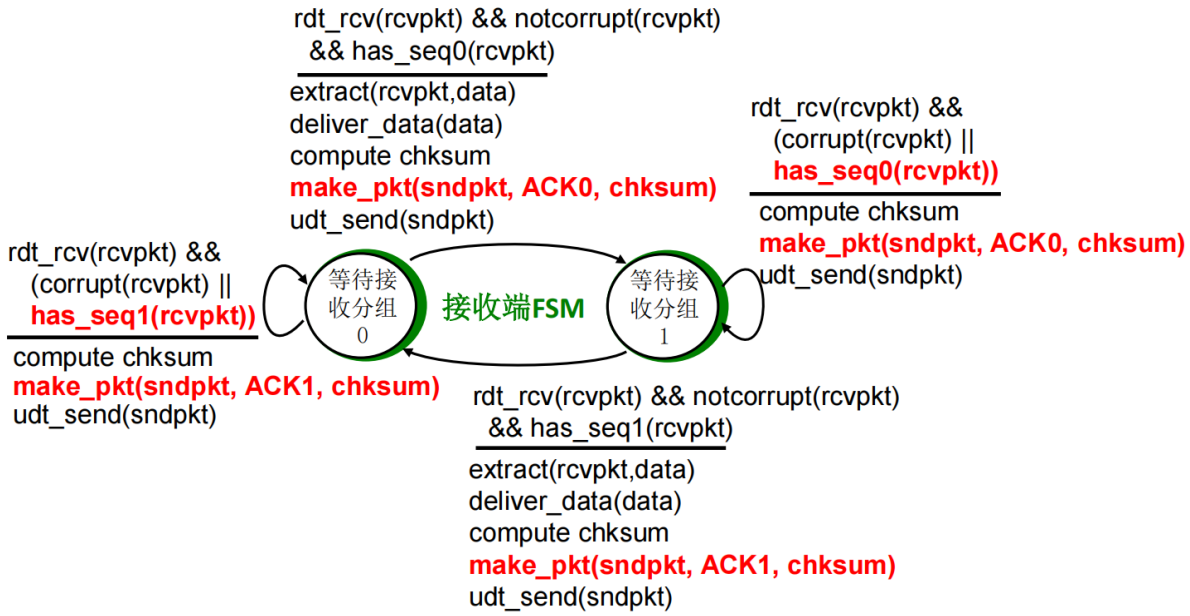
接收端

在接收端使用两个序列号和两个状态实现了可靠数据传输，具体流程如下

接收端初始状态的序列号为0，等待接收0号数据包。在接收到数据包之后，首先要计算数据包的校验和，确保数据包在传输的过程中没有发生损坏。
如果数据包的校验和不为0xFFFF，则说明在传输的过程中出现了错误，需要对当前的数据包进行重传，此时接收端回复的ACK中确认号为上一个序列号。

而如果成功通过了校验和，要判断当前接收到的数据包中的序列号是否和自己想要确认的序列号一致，如果一致的话则说明这条报文的发送正确，确认对应的序列号，并进入下一个状态。
如果数据包的序列号和想要确认的序列号不一致，则同样回复上一个序列号，表示上一个这个数据包确认失败了。

接收端不需要考虑超时重传的问题，因为如果接收端回复的ACK失败了，会触发发送端的超时重传，此时接收到的序列号和期望确认的序列号不符，而回复的确认号正好为接收到数据包的序列号。
因此发送端在接收到这个ACK之后就会进入下一个序列号的发送状态，整个流程就恢复正常了。



具体代码实现如下，这里的代码十分冗长，是因为将这个接受逻辑都封装在这一个函数内部了。
因此接收端接收到数据包之后，需要根据不同的标志位分别进行处理。

- 当接收到的是服务端发送的SYN的时候，表示当前正处于建立连接的阶段，客户端应当回应ACK SYN报文
- 当接收到的是服务端发送的FIN的时候，表示当前正处于断开连接的阶段，客户端应当回应ACK FIN报文，并且进入等待退出的阶段。
  拉起一个计时器，等待两倍的RTO，如果两倍的RTO内没有接到服务端重发的FIN则说明能够正常断开，否则的话需要进行重传并重置计时器状态。
- 当接收到的额是服务端发送的数据段的时候，需要判断当前传输的数据段包含的内容是文件头还是文件内容
  - 如果当前的数据段包含的是文件头，则根据数据段的描述信息创建文件，然后回复相应的ACK序列号
  - 如果当前的数据段包含的文件内容，则进行写文件的操作，然后回复指定相应的ACK序列号

```
[[noreturn]] void beginRecv() {
    // receive message from server
    struct Message recvBuffer{};
    // write to file
    std::fstream file;
```

```
    unsigned int fileSize;
    unsigned int currentSize = 0;
    std::string filename;
    while(true) {
        int serverAddressLength = sizeof(SOCKADDR);
        int recvLength = recvfrom(clientSocket, (char*) &recvBuffer,
sizeof(struct Message), 0, (struct sockaddr *) &serverAddress,
&serverAddressLength);
        if(recvLength > 0) {
            printMessage(recvBuffer);
            if(recvBuffer.checksumValid(&recvPseudoHeader) &&
recvBuffer.seq == CURRENT_SEQ) {
                // if message is SYN
                if(recvBuffer.isSYN()) {
                    if(!randomLoss()) {
                        sendACKSYN(recvBuffer.seq);
                    }
                }
                // if message is FIN
                else if(recvBuffer.isFIN()) {
                    if(!randomLoss()) {
                        sendACKFIN(recvBuffer.seq);
                    }
                    exitTime = 0;
                    waitExit();
                }
                // if message contains data
                else {
                    // if message contains file header
                    if(recvBuffer.isFHD()) {
                        struct FileDescriptor fileDescriptor{};
                        memcpy(&fileDescriptor, recvBuffer.data,
sizeof(struct FileDescriptor));
                        std::cout << "Receive file header: [Name:" <<
fileDescriptor.fileName << "] [Size:"
                                  << fileDescriptor.fileSize << "]" <<
std::endl;
                        fileSize = fileDescriptor.fileSize;
                        filename = fileDir + "/" +
fileDescriptor.fileName;
                        currentSize = 0;
                        // create file
                        file.open(filename, std::ios::out |
std::ios::binary);
                    }
                    else {
                        // write to file
                        file.write(recvBuffer.data,
recvBuffer.getLen());
                        currentSize += recvBuffer.getLen();
                        if(currentSize >= fileSize){
```

```
                            std::cout << "File receive success!" << filename << std::endl;
                            file.close();
                        }
                    }
                    // send ACK to server
                    if(!randomLoss()){
                        sendACK(recvBuffer.seq);
                    }
                }
                // update current seq
                CURRENT_SEQ = (CURRENT_SEQ + 1) % 2;
            }
            // if package is not valid or receive seq is not equal to current seq
            // need to send last ACK again
            else {
                if(!randomLoss()) {
                    if(recvBuffer.isSYN()) {
                        sendACKSYN((CURRENT_SEQ + 1) % 2);
                    }
                    else {
                        sendACK((CURRENT_SEQ + 1) % 2);
                    }
                }
            }
        }
    }
}
```

功能测试

由于在一开始代码设计的阶段并没有考虑用于测试的路由程序的影响，导致我在设计报文的时候使用了发送端和接收端的IP和端口号信息。
如果接入路由程序，发送端和接收端是分别连到路由程序的两个端口上的，也就是说其实双端之间是无感的。
这就导致我在计算校验和的时候，会因为发送端和接收端的端口号不一致导致问题。

因此在本次实验中并没有使用提供的路由程序，而是自己手动按照丢包率进行模拟丢包。模拟丢包的代码如下

```
bool randomLoss() {
    return rand() % 100 < LOSS_RATE;
}
```

接下来分别展示数据传输的效果和超时重传的效果。可以看到在下面的实验运行结果中清晰的看到如下现象。

- 当发送了序列号为seq的报文之后，会等待接收到ack==seq的ACK应答报文，并且展示接收成功的信息。

- 当发送了序列号为seq的保温之后，如果没有等到对应的ACK应答报文，则会触发超时重传，之后会再次收到接收端应答的ACK。
- 接收端这边可以看到相邻接收到的报文的序列号总是0/1，证明能够在两个接收状态之间切换。