**浙江科技学院**

**计算机网络课程设计**

学 院：信息与电子工程学院

专 业： 软件工程

班 级： 161/162

学 号： 1160299260

5160491075

1160299063

姓 名： 李航 苏忠峰 刘宇琪

2018 年 12月31日-2019年 1月6日

1. 课程设计简介

**题目：**调用 UDP 协议完成一个应用层可靠传输协议

**设计目标：**

1. 调用UDP协议完成一个应用层可靠传输协议概要设计；
2. 实现停等协议；
3. 实现滑动窗口协议；
4. 实现一个模拟随机丢失和延迟数据的中间程序 详细设计和编码。

**开发语言：**C

**开发系统：**Mac OS 、Linux

1. 概要设计
2. 整体框架

整个系统是一个类 QQ 的通信系统，允许两方进行全双工通信。因为 TCP 协议每次连接都需要 3 次握手，使用 TCP 来保证所需的即时可靠通信显然是不合理的。因此无需连接的 UDP 协议更适合底层的数据传输，并将可靠传输的任务交给应用层来完成。

为了方便讨论，我们将通信的两方分别设为**发送方**和**接收方**。发送方只有发送数据和接收确认信息两种行为，同样接收方也只有接收数据和发送确认信息两种行为。实际上系统中通信的两方在功能上是完全对称的，任何一方既可以是发送方也可以是接收方。

两方在通信开始前都会各自为发送和接收创建两个子线程。发送方的发送线程对应接收方的接收线程，发送方的接收线程对应接收方的发送线程，如此双方建立了两条信道来满足全双工通信。同时，为了保证子线程能够一致存活，主线程需要进入到一个“循环阻塞”的状态中。



在用户信息输入输出部分，系统依托操作系统自带的 terminal 来显示/输入，不再编写 GUI 控制。

1. 停等协议

“停止等待”指发送方每发送一个**数据包**（基本数据传输单元）就停止发送，等待的接收方的确认。在收到了接收方发来的确认信号后，再发送下一个数据包。

1. 滑动窗口协议

与停等协议有些不同，滑动窗口协议要求发送方设置一个发送窗口，这个窗口能够包含一定数量的数据包，每次（逻辑上，实际是窗口长度次）发送数据就直接将一个窗口内的所有数据包都发送出去，然后等待接收方的确认数据包。本系统要求接收方采用累积确认的方式。发送方通过确认数据包获得 ack 信号，即接收方收到的按顺序到达的最后一个数据包的 seq 信号值加 1。然后，发送方调整窗口信息，从第 ack 信号值个数据包开始发送数据，如此循环。

1. 模拟异常情况

事实上，在同一台计算机内部甚至在同一个小型局域网下，使用 UDP 发送数据是不太可能出现异常情况。为了增加异常，我们可以随机地调用 `**sleep**` 函数来模拟延迟，在代码中添加随机的删除数据包的逻辑来模拟丢包。

1. 详细设计和编码
2. 系统公共部分
   1. 公共部分设计与说明

发送方每次接收用户在 terminal 中输入的**信息**（以用户输入的 enter 信号为一条信息的结尾），并将这条信息写入接收方的缓冲区。

为了能够接收、显示用户的信息，接收方和发送方各自都创建一个固定长度且尺寸相同的缓冲区，在发送方的称为**输入缓冲区**（从 terminal 输入到缓冲区），在接收方的称为**输出缓冲区**（从缓冲区输出到 terminal ）。要求缓冲区能够完整地写入一条信息，而且双方缓冲区的字节单元能够一一对应。

* 1. 主线程逻辑流程图



* 1. 代码实现  
     **主线程代码**

int main(int argc, const char \* argv[]) {

srand((unsigned)time(NULL));

struct sockaddr\_in my,oppo,oppo1;

pthread\_t thread1, thread2;

Params p,p2;

// 建立本地址描述符并绑定。

int my\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

setMySockaddr(&my,NULL,MY);

bind(my\_sock, (struct sockaddr\*)&my, sizeof(my));

// 定时器结构体

typedef struct timeval {

long tv\_sec;

int tv\_usec;

} Timeval;

// 设置超时定时器

Timeval ti;

ti.tv\_sec = 1;

ti.tv\_usec = 0;

int oppo\_sock = socket(AF\_INET, SOCK\_DGRAM, 0);

setsockopt(oppo\_sock,SOL\_SOCKET,SO\_RCVTIMEO,&ti, sizeof(ti));

// ------------发送线程----------------

setOppoSockaddr(&oppo1,NULL,OPPO);

setParams(&p2, oppo\_sock,(struct sockaddr\*)&oppo1);

pthread\_create(&thread2,NULL,ARQ\_send,&p2);

// ------------接收线程----------------

setOppoSockaddr(&oppo,NULL,OPPO);

setParams(&p, my\_sock,(struct sockaddr\*)&oppo);

pthread\_create(&thread1,NULL,ARQ\_receive,&p);

while(1);

return 0;

}

**API 说明**

void setOppoSockaddr(struct sockaddr\_in \* oppo,char \* ipAddr,int port);

设置目标地址

void setMySockaddr(struct sockaddr\_in \* my,char \* ipAddr,int port);

设置本地地址

void setParams(Params \* p,int sock,struct sockaddr\* addr);

结构体转换函数

1. 停等协议的设计与实现
   1. 停等协议的设计

按照定义，每条信息在发送时将被分成多个数据包（每 MAX\_LENGTH 个字节切分一个），以下是每个数据包的数据结构：

udp\_pack {

unsigned int seq;

unsigned int total;

char content[MAX\_LENGTH];

}

seq 标识了当前数据包的序号；

total 标识了整条信息拥有的数据包的数量；

content 记录了数据包中的有效数据，最多 MAX\_LENGTH 个字节长度。

发送方发送完数据包之后需要等待接收方传回一个**确认数据包**（包含接收方确认收到的数据包中的seq 值），并且会对比确认数据包中的 seq 值与真实的 seq 值是否一致，如果不一致或者超时收不到确认数据包，发送方就会使 **outTime 信号**（记录了一次信息发送出现异常的次数）的值加 1 ，同时重新发送刚才的数据包。如果 outTime 信号值大于 20 ，系统将认为当前状态不适合对话，请求用户重新输入。

接收方在**本地**（代码内）设置了一个 seq 信号，对应数据包中的 seq 成员。每当接收到一个数据包，接收方就会对比本地的 seq 信号与数据包中的 seq 成员值，两者一致，接收方才会发送确认数据包给发送方。接收方每成功接收一个数据包，就将本地的 seq 信号值加 1 ，等待下一个数据包。

* 1. 考虑异常情况

**数据包丢失**

如果数据包丢失，必然导致超时，发送方将重新发送数据包直到收到正确的确认数据包。

**确认数据包丢失**

与数据包丢失一样，发送方也会超时重传，不同的是，接收方将会收到两份相同的数据包。如之前的收发双发的缓冲区的存储单元的下标是一一对应的，这样接收方每获得一个数据包，只需要检查数据包中的内容是否已经在缓冲区的对应位置，如果是重复的数据包，就直接丢弃并发送确认数据包。

**数据包延迟**

数据包延迟，会导致接收方收到两份相同的数据包，接收方的操作与确认数据包丢失的情况一致。此情况下，发送方也可能会收到两份相同的确认数据包，发送方将比对确认数据包中的 seq 值与当前已发送的发送的数据包的下标，如果 seq 值小于当前下标，发送方即判定确认数据包过期，直接丢弃。

**确认数据包延迟**

此情况下，收发双方都会分别收到两份相同的数据包或确认数据包，接收方的操作与确认数据包丢失的情况下一致，发送方的操作则与数据包延迟的情况下一致。

**糟糕的通信环境带来的异常**

系统使用一个 outTime 信号来判断当前的通信环境是否适合对话。以一条信息的发送为一次轮回，上诉的所有异常情况出现后，都会导致 outTime 的值增加 1 ，当 outTime 信号的值超过阈值之后，系统就会退出当前信息的发送，告诉用户通信环境比较糟糕，并要求用户重新输入。进入到下一轮的通信前，outTime 又会被置 0 。

* 1. 停等协议流程图



* 1. 停等协议实现代码

**发送方子线程**

void \* stop\_and\_wait\_send(void \* p2){

// 参数转化为可用

Params param = \*(Params \*)p2;

int my\_sock = param.sock;

struct sockaddr \*oppo = param.oppo;

Udp\_pack pack;

char buff[100]; // 发送缓存

while (1) { // 一次完整的发送

memset(buff,0,100);

printf("请输入您要发送的信息:");

if(!scanf("%s", buff)){ break; }

int outTime = 0; // 设定超时次数

//这一次内容的总包数 向下取整

int total = (int) strlen(buff) / MAX\_LENGTH;

for(int i=0;i <= total; i++){

pack2Udp(i, total, buff, &pack);

long err;

do{

// 发送这一个包

sendto(my\_sock, &pack, pack\_len, 0, oppo, len);

// 等待确认

err = recvfrom(my\_sock, &pack, pack\_len,0,oppo,&len);

// 检查是否有异常

checkTime(&outTime,&err,pack,i);

}while(err == -1&&(outTime < 20));

memset(&pack, 0, pack\_len);

if (outTime > 20) { // 在跳出一次,跳出本次发送。

break;

}

}

}

return 0;

}

**接收方子线程**

void \* stop\_and\_wait\_receive(void \* p){

// 参数转化为可用

Params param = \*(Params \*)p;

int my\_sock = param.sock;

struct sockaddr \*oppo = param.oppo;

Udp\_pack pack;

char receive[100];

while(1){ // 保证线程存活

memset(receive,0, sizeof(receive));

int seq = -1;

// 保证一次所有信息都收到

while (1) {

memset(&pack, 0, pack\_len);

// 等待接受数据

recvfrom(my\_sock, &pack, pack\_len, 0, oppo, &len);

// 模拟发送丢失 重新监听接收

if (rand()%10 > 8) { continue; }

// 判断是否是这次发送的信息

if (!isThisMsg(seq, pack)){ continue; }

// 接受确认号

int i = add2Receive(pack, receive);

// 模拟确认延迟 确认丢失

if(createError()){ continue; }

// 发送确认信息

sendto(my\_sock, &pack, pack\_len, 0, oppo, len);

printf("确认号%d\n",pack.seq);

if (i == pack.total) {

break;

} else {

seq++;

} // 如果最后一个都收到了就跳出循环,否则序列号+1继续接收

} // 一次完整的接收循环

printf("对面说:%s\n",receive);

}

}

**API 说明**

void pack2Udp(int i, int total, char \* buff, Udp\_pack \* pack);

打包一个udp数据包；

void checkTime(int \* outTime, long \* err, Udp\_pack pack, int i);

处理延迟确认 没有确认的异常；

int add2Receive(Udp\_pack buff, char \* receive);

添加到接收缓存并返回确认号；

int createError(void);

模拟确认延迟 确认丢失；

int isThisMsg(int seq,Udp\_pack pack);

判定是不是这次应该接受的信息

1. 滑动窗口协议的设计与实现
   1. 滑动窗口协议设计

滑动窗口协议相对停等协议要复杂一些，我们仍然以一条信息为单位进行一次完整的滑动窗口收发过程。在本系统中，发送方与接收方在本地各自拥有一个窗口，分别为**发送窗口**和**接收窗口**。接收方接收发送方发来的数据包，被动调整接收窗口信息，最终目的是使输出缓冲区与输入缓冲区的状态同步。以下给出滑动窗口协议中每个数据包的数据结构：

udp\_pack {

unsigned int seq;

unsigned int total;

unsigned int win\_size;

unsigned int ack;

unsigned int winStart;

char content[MAX\_LENGTH];

}

seq 标识了当前数据包相对整个数据包序列的序列值；

total 标识了当前信息中数据包的总数量；

win\_size 标识了窗口大小；

ack 确认号，接收方收到的按顺序到达的最后一个数据包的 seq 信号值加 1 ；

winStart 标识了当前窗口起始的数据包的序号；

content 承载着每个数据包中的有效数据。

发送方每次开始发送窗口数据，都先找到起始数据包应该的下标，即 winStart ，然后会对数据进行打包，每组成一个数据包就发送出去，如此连续发送 win\_size 个数据包。之后等待接收方传来的确认数据包。

接收方接收数据，每一轮只接收 win\_size 个数据包，一轮接收到的数据包可能是失序的也可能是重复，接收方在接收时都不考虑，只是全部照收，“假装”自己收到的和发送方发出的是相同的内容。在接收完毕之后接收方开始计算收到的按顺序到达的最后一个数据包的 seq 信号值加 1，并将其设置为 ack 信号值，写入确认信号包中，发送给发送方。

重新回到发送方。发送方获得确认数据包，就知道接收方当前的本地窗口情况，或使用 Go-Back-N 的算法做异常处理或继续正常发送。与停等协议一样，滑动窗口协议都使用 outTime 信号来记录异常情况（失序、重复等），这里 outTime 的阈值被扩大到了 50 。当 outTime 大于 50 时，系统会认为当前网络环境不适合对话，打印提示信息，并请求用户重新输入。

关于系统中 ack 信号的解释：无论是收发双方传递的数据包中都包含着 ack 信号，但在接收发送确认数据包与发送方接收确认数据包的角度，这个 ack 信号才是有意义的，它同步了收发双方的本地窗口信息。换言之， ack 信号表达了一种“请求”，即接收方希望发送方下一次能发送哪些数据；也表达了一种“确认”，即接收方告诉发送方自己获得了哪些数据。

* 1. 考虑异常情况

首先需要再次说明，在滑动窗口协议中，我们不要求接收方每轮都一定能接收到一个窗口的数据包，而是每轮都只接收固定数量的数据包。一轮接收结束之后，计算自己到底收到了多少正确数据，并且发送确认数据包给发送方，告诉发送方自己收到了多少数据包，同时请求下一轮的数据包。

在发送窗口中的数据时，如果出现数据包丢失、数据包延迟的情况，使接收方无法收满足够数量的数据包，发送方将无法收到确认数据包，并且导致发送方出现超时情况，发送方将再重新发送窗口内所有的数据包。但是此时，接收方只能再接收窗口内的第一个数据包（假设丢失了一个数据包），并且告诉发送方只收到了第一个数据包。这时候，收发双方将同步调整自己的本地窗口，为下一轮的收发做准备。

在发送完窗口中的数据时，如果出现确认数据包丢失、确认数据包丢失的情况。发送一样会出现超时异常，发送方将直接重发窗口内数据，接收方也会重新接收。

其他的细节操作请查阅停等协议中对异常处理的描述，此处不再赘述。

* 1. 滑动窗口程序流程图



* 1. 滑动窗口程序实现

**发送方子线程**

void \* ARQ\_send(void \* p2){

// 参数转化为可用

Params param = \*(Params \*)p2;

int my\_sock = param.sock;

struct sockaddr \*oppo = param.oppo;

UDP\_pack pack;

char buff[100]; // 发送缓存

while (1) {

memset(buff,0,100);

printf("请输入你要发送的信息:");

if (!scanf("%s", buff)) { break; }

//计算包的长度，向上取整

int total = ceil(strlen(buff) \* 1.0 / MAX\_LENGTH);

int winStart = 0,win\_size = 0,winEnd = 0;

int outTime = 0;

// 确定窗口范围

updateWindow(&win\_size,&winStart,&winEnd,total,NULL);

while(winStart < total){

if(outTime > 50){

printf("发生超时！\n");

break;

}

// 发送，发送窗口内的包

for (int i = winStart; i < winEnd; i++) {

// 组装一个包

pack2Udp(&pack,buff,winStart,i,total,win\_size);

// 发送一个包

sendto(my\_sock, &pack, pack\_len, 0, oppo, len);

}

// 接受确认号

long err = recvfrom(my\_sock, &pack, pack\_len, 0, oppo, &len);

// 处理超时以及接收延迟数据

if(checkTime(&outTime,err,pack,winStart)){ continue; }

// 确定窗口范围

updateWindow(&win\_size,&winStart,&winEnd,total,&pack);

memset(&pack,0,pack\_len);

}

}

return 0;

}

**接收方子线程**

void \* ARQ\_receive(void \* p){

// 参数转化为可用

Params param = \*(Params \*)p;

int my\_sock = param.sock;

struct sockaddr \*oppo = param.oppo;

UDP\_pack buff;

char receive[100];

// 保证线程存活

while(1){

memset(receive,0, sizeof(receive));

int win\_size = WIN\_SIZE, winStart = 0;

int time = 0;

int total = 1000;

do{

// 等待接受数据

recvfrom(my\_sock, &buff, pack\_len, 0, oppo, &len);

// 模拟发送丢失 重新监听接收

if (rand()%10 > 8){

memset(&buff,0,pack\_len);

}else{

// 更新窗口大小

win\_size = buff.win\_size;

total = buff.total;

// 赋值

add2Receive(buff,receive);

// 接收次数加一

time++;

}

// 窗口大小次数接收完毕

if (time >= win\_size){

time = 0;

// 获取确认ack

ensureACK(&buff,receive);

// 窗口移动

winStart = buff.ack;

// 模拟确认延迟 确认丢失

if(createError()){ continue; }

// 发送确认

sendto(my\_sock, &buff, pack\_len, 0, oppo, len);

}

} while (winStart < total);

if (receive[0] != '\0') {

printf("对面说:%s\n",receive);

}

}

return 0;

}

**API 说明**

void pack2Udp(UDP\_pack \* pack, char \* buff, int winStart,

int i, int total, int win\_size);

打包一个 UDP 数据包

void updateWindow(int \* win\_size, int \* winStart, int \* winEnd,

int total, UDP\_pack \* pack);

更新窗口信息

int checkTime(int \* outTime, long err, UDP\_pack pack, int winStart);

处理延迟确认 没有确认的异常

void add2Receive(UDP\_pack buff, char \* receive);

将数据包中的内容添加到输出缓冲区中

int createError(void);

模拟确认延迟 确认丢失

void ensureACK(UDP\_pack \* pack, char \* receive);

计算 ack 信号值

1. 模拟随机丢失和延迟数据的中间程序

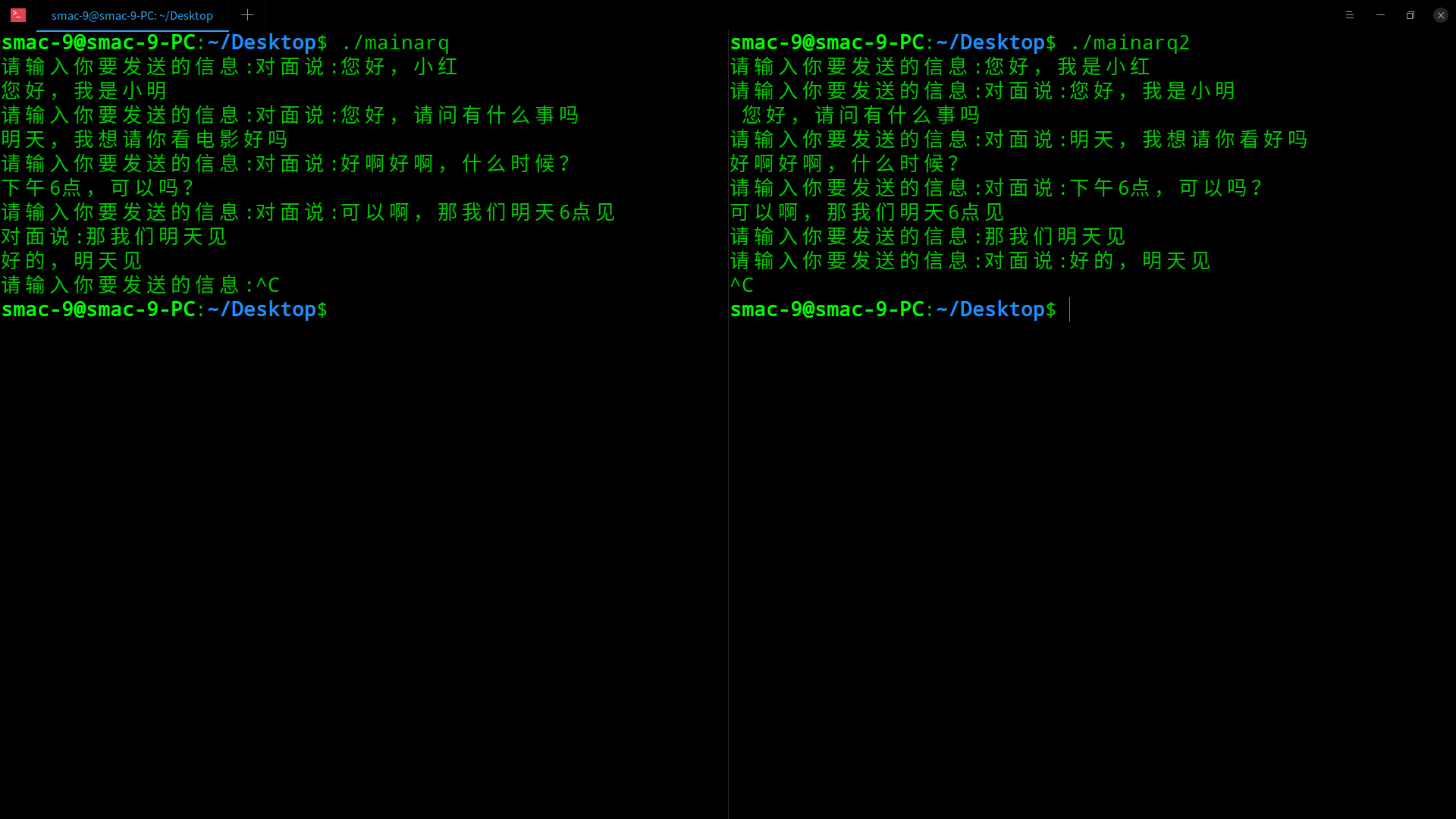
如概要设计中描述，编写随机函数，控制线程进入睡眠状态或是删除数据包/确认数据包即可。此处不再赘述。

1. 测试数据及程序运行情况
2. 停等协议



注：停等协议的测试在 MacOS 平台下进行，单机多端口通信。

1. 滑动窗口协议



注：滑动窗口协议的此次测试在 Linux 平台下进行，单机多端口通信。

1. 实验体会

通过本次课程设计，分别基于停等协议、连续ARQ协议，实现了对UDP的可靠传输。通过设计报文结构以及调用 SocketAPI ，体会到了传输协议的概念。加深了对多线程的理解。