**实验 2：可靠数据传输协议-GBN 协议的设计与实现**

|  |  |
| --- | --- |
| **学号：** | 1130310128 |
| **姓名：** | 杨尚斌 |
| **专业：** | 计算机科学与技术 |
| **指导老师：** | 聂兰顺 |

1. **实验目的**

理解滑动窗口协议的基本原理；掌握 GBN 的工作原理；掌握基于 UDP 设计并实现一个 GBN 协议的过程与技术。

1. **实验环境**

* 接入 Internet 的实验主机
* Windows 10, Visual Studio 2013
* 开发语言：C++

1. **实验内容**
   * + 1. 基于 UDP 设计一个简单的 GBN 协议，实现单向可靠数据传输（服 务器到客户的数据传输）。
       2. 模拟引入数据包的丢失，验证所设计协议的有效性。
       3. 改进所设计的 GBN 协议，支持双向数据传输；
       4. 将所设计的 GBN 协议改进为 SR 协议。
2. **实验要点**
3. 基于 UDP 实现的 GBN 协议，可以不进行差错检测，可以利用 UDP 协议差错检测；
4. 自行设计数据帧的格式，应至少包含序列号 Seq 和数据两部分；
5. 自行定义发送端序列号 Seq 比特数 L 以及发送窗口大小 W，应满 足条件 W+1<=2L。
6. 一种简单的服务器端计时器的实现办法：设置套接字为非阻塞方 式，则服务器端在 recvfrom 方法上不会阻塞，若正确接收到 ACK 消息， 则计时器清零，若从客户端接收数据长度为-1（表示没有接收到任何数 据），则计时器+1，对计时器进行判断，若其超过阈值，则判断为超时， 进行超时重传。（当然，如果服务器选择阻塞模式，可以用到 select 或 epoll 的阻塞选择函数，详情见 MSDN）
7. 为了模拟 ACK 丢失，一种简单的实现办法：客户端对接收的数 据帧进行计数，然后对总数进行模 N 运算，若规定求模运算结果为零则 返回 ACK，则每接收 N 个数据帧才返回 1 个 ACK。当 N 取值大于服务 器端的超时阀值时，则会出现服务器端超时现象。
8. 当设置服务器端发送窗口的大小为 1 时，GBN 协议就是停-等协议。
9. **实验结果**
   * + 1. **GBN 协议数据分组格式、确认分组格式、各个域作用**



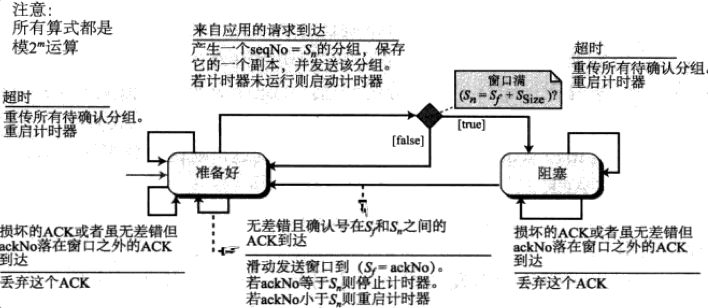
Seq 为 1 个字节，取值为 0~255，（故序列号最多为 256 个）；

Data≤1024 个字节，为传输的数据；

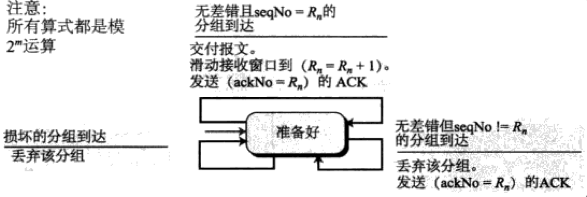
最后一个字节放入 EOF0，表示结尾。

* + - 1. **协议两段程序流程图**

**Clinet**

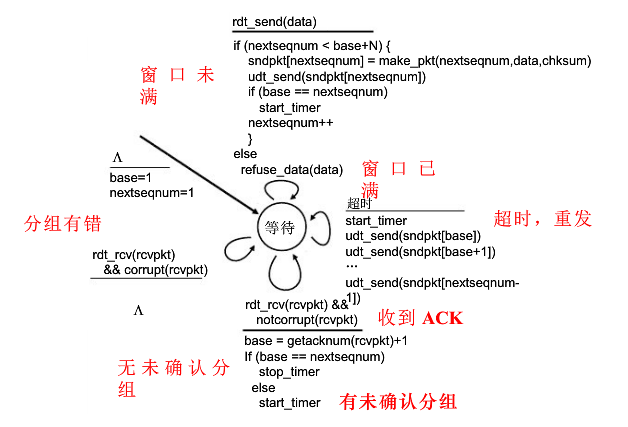


**Server**

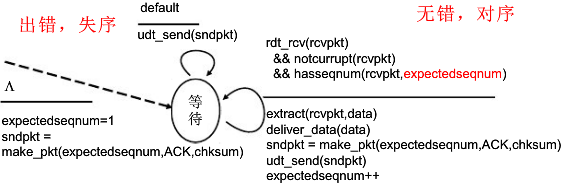


* + - 1. **协议典型交互过程**

**Client**



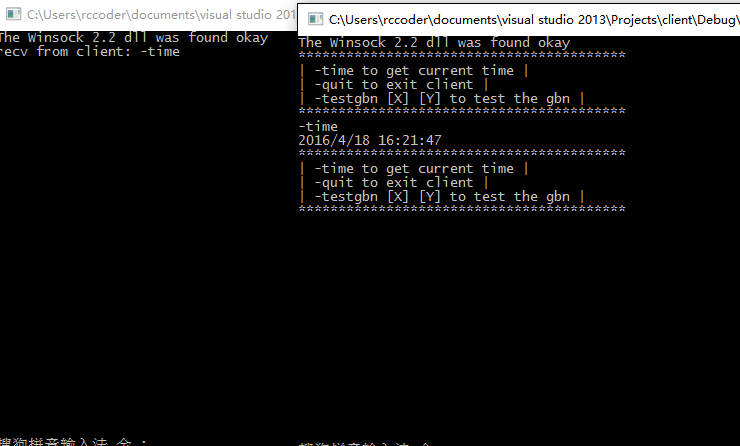
**Server:**

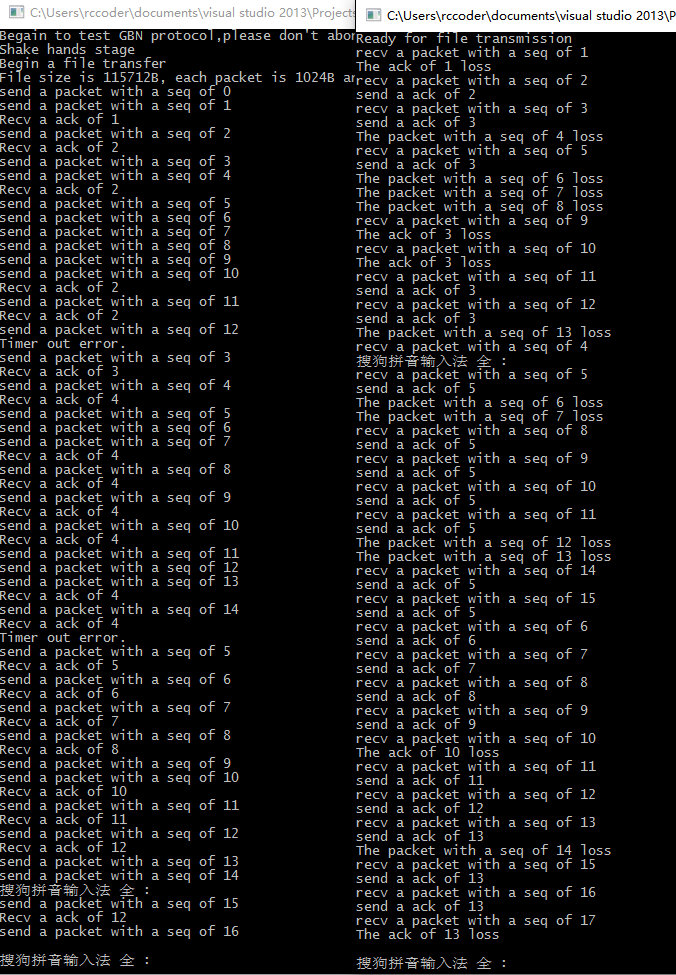


* + - 1. **数据分组丢失验证模拟方法**

客户端对所接收到的数据帧进行计算，然后对总数除 N 去模，如果结果为0则表示需要返回数据，也就是说是 N 个数据帧才会返回一个数据，当N 的取值大于服务器端的阀值的时候就会出现服务器端超时的现象

* + - 1. **实验验证结果**





能够在图中看见累计确认和超时重传都已经正确实现

* + - 1. **源代码**

**Node版本：**

**Counter.js**

'use strict';

const counter = (windowSize, now = 0, base = 0) => {

return {

back: () => {

now -= 1;

if (now < 0) {

now += windowSize;

}

return now;

},

go: () => {

now += 1;

if (now >= windowSize) {

now -= windowSize;

}

return now;

},

next: () => {

let n = now + 1;

if (n >= windowSize) {

n -= windowSize;

}

return n;

},

get\_now: () => {

return now;

},

set\_now: (x) => {

now = x;

},

gt: (x) => {

if (x < base) {

x += windowSize;

}

return (now > x);

},

minus: (x) => {

let n = now;

if (n < base) {

n += windowSize;

}

return (n - x);

},

get\_base: () => {

return base;

},

reset\_base: (x) => {

base = x;

}

}

}

module.exports = counter;

**client.js**

'use strcit';

const dgram = require('dgram');

const socket = dgram.createSocket('udp4');

const counter = require('./counter')

const HOST = '127.0.0.1';

const PORT = '8888';

const windowSize = 5;

const timeout = 500;

socket.bind();

const seg = counter(8);

const last\_ack = counter(8);

const msg = new Buffer("Hello World");

const packages = (() => {

let package\_result = [];

for (let i = 0, l = msg.length; i < l; i++) {

package\_result.push(msg.slice(i, i + 1))

}

return package\_result;

})();

const start = (() => {

let handle = null;

return () => {

if (handle !== null) {

clearTimeout(handle);

}

for (let i = 0, l = last\_ack.minus(last\_ack.get\_base()); i < l; i++) {

packages.shift();

}

if (packages.length === 0) {

console.log('finshed!');

return;

}

last\_ack.reset\_base(last\_ack.get\_now());

seg.set\_now(last\_ack.get\_now());

send\_window();

handle = setTimeout(start, timeout);

return;

}

})();

const send\_one\_package = (buffer, id) => {

const tep = new Buffer(1);

tep.writeInt8(id);

const data = Buffer.concat([tep, buffer]);

socket.send(data, 0, data.length, 8888, HOST);

};

const send\_window = function() {

var i, j, ref, results;

seg.set\_now(last\_ack.get\_now());

console.log("sendding window lastACK:" + (last\_ack.get\_now()));

//for (let i = 0; i <= windowSize; ++i) {

for (i = j = 0, ref = windowSize; j < ref; i = 0 <= ref ? ++j : --j) {

if (packages[i] != null) {

send\_one\_package(packages[i], seg.go());

}

}

};

socket.on('message', (msg) => {

let ACK = msg.readInt8();

if (!last\_ack.gt(ACK)) {

while (last\_ack.get\_now() != ACK) {

last\_ack.go();

}

}

console.log(`receive ACK ${ACK}, last\_ack： ${last\_ack.get\_now()}`);

return;

});

socket.on('error', (err) => {

console.log(err);

});

start();

**server.js:**

'use strict';

const dgram = require('dgram');

const counter = require('./counter');

const socket = dgram.createSocket('udp4');

const PORT = '8888';

socket.bind(PORT);

const seg = counter(8);

socket.on('message', (msg, info) => {

let id = msg.readInt8();

let content = '' + msg.slice(1);

if (seg.next() === id) {

seg.go();

console.log(`${content.length} byte from ${info.address}:${info.port}. Accepted id: id: ${id}, content: ${content}`)

} else {

console.log(`${content.length} byte from ${info.address}:${info.port}. Droped id: id: ${id}, content: ${content}`)

}

if (Math.random() < 0.5) {

console.log(`Not sending the ACK, id: ${seg.get\_now()}`);

return;

}

let ACK = new Buffer(1);

ACK.writeInt8(seg.get\_now());

socket.send(ACK, 0, ACK.length, info.port, info.address);

return;

});

socket.on('error', (err) => {

console.log(err);

})

**Cpp 版本：**

**client.cpp**

// GBN\_client.cpp : 定义控制台应用程序的入口点。

//

#include <stdio.h>

#include <stdlib.h>

#include <WinSock2.h>

#include <time.h>

#pragma comment(lib,"ws2\_32.lib")

#define SERVER\_PORT 12340 //接收数据的端口号

#define SERVER\_IP "127.0.0.1" // 服务器的 IP 地址

const int BUFFER\_LENGTH = 1026;

const int SEQ\_SIZE = 20;//接收端序列号个数，为 1~20

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* -time 从服务器端获取当前时间

-quit 退出客户端

-testgbn [X] 测试 GBN 协议实现可靠数据传输

[X] [0,1] 模拟数据包丢失的概率

[Y] [0,1] 模拟 ACK 丢失的概率

\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void printTips(){

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("| -time to get current time |\n");

printf("| -quit to exit client |\n");

printf("| -testgbn [X] [Y] to test the gbn |\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: lossInLossRatio

// FullName: lossInLossRatio

// Access: public

// Returns: BOOL

// Qualifier: 根据丢失率随机生成一个数字，判断是否丢失,丢失则返回TRUE，否则返回 FALSE

// Parameter: float lossRatio [0,1]

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BOOL lossInLossRatio(float lossRatio){

int lossBound = (int)(lossRatio \* 100);

int r = rand() % 101;

if (r <= lossBound){

return TRUE;

}

return FALSE;

}

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

{

//加载套接字库（必须）

WORD wVersionRequested;

WSADATA wsaData;

//套接字加载时错误提示

int err;

//版本 2.2

wVersionRequested = MAKEWORD(2, 2);

//加载 dll 文件 Scoket 库

err = WSAStartup(wVersionRequested, &wsaData);

if (err != 0){

//找不到 winsock.dll

printf("WSAStartup failed with error: %d\n", err);

return 1;

}

if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2)

{

printf("Could not find a usable version of Winsock.dll\n");

WSACleanup();

}

else{

printf("The Winsock 2.2 dll was found okay\n");

}

SOCKET socketClient = socket(AF\_INET, SOCK\_DGRAM, 0);

SOCKADDR\_IN addrServer;

addrServer.sin\_addr.S\_un.S\_addr = inet\_addr(SERVER\_IP);

addrServer.sin\_family = AF\_INET;

addrServer.sin\_port = htons(SERVER\_PORT);

//接收缓冲区

char buffer[BUFFER\_LENGTH];

ZeroMemory(buffer, sizeof(buffer));

int len = sizeof(SOCKADDR);

//为了测试与服务器的连接，可以使用 -time 命令从服务器端获得当前时间

//使用 -testgbn [X] [Y] 测试 GBN

//其中[X]表示数据包丢失概率

// [Y]表示 ACK 丢包概率

printTips();

int ret;

int interval = 1;//收到数据包之后返回 ack 的间隔，默认为 1 表示每个都返回 ack，0 或者负数均表示所有的都不返回 ack

char cmd[128];

float packetLossRatio = 0.2; //默认包丢失率 0.2

float ackLossRatio = 0.2; //默认 ACK 丢失率 0.2

//用时间作为随机种子，放在循环的最外面

srand((unsigned)time(NULL));

while (true){

gets\_s(buffer);

ret = sscanf(buffer, "%s%f%f", &cmd, &packetLossRatio, &ackLossRatio);

//开始 GBN 测试，使用 GBN 协议实现 UDP 可靠文件传输

if (!strcmp(cmd, "-testgbn")){

printf("%s\n", "Begin to test GBN protocol, please don't abort the process");

printf("The loss ratio of packet is %.2f,the loss ratio of ack is %.2f\n",packetLossRatio,ackLossRatio);

int waitCount = 0;

int stage = 0;

BOOL b;

unsigned char u\_code;//状态码

unsigned short seq;//包的序列号

unsigned short recvSeq;//接收窗口大小为 1，已确认的序列号

unsigned short waitSeq;//等待的序列号

sendto(socketClient, "-testgbn", strlen("-testgbn") + 1, 0, (SOCKADDR\*)&addrServer, sizeof(SOCKADDR));

while (true)

{

//等待 server 回复设置 UDP 为阻塞模式

recvfrom(socketClient, buffer, BUFFER\_LENGTH, 0, (SOCKADDR\*)&addrServer, &len);

switch (stage){

case 0://等待握手阶段

u\_code = (unsigned char)buffer[0];

if ((unsigned char)buffer[0] == 205)

{

printf("Ready for file transmission\n");

buffer[0] = 200;

buffer[1] = '\0';

sendto(socketClient, buffer, 2, 0,

(SOCKADDR\*)&addrServer, sizeof(SOCKADDR));

stage = 1;

recvSeq = 0;

waitSeq = 1;

}

break;

case 1://等待接收数据阶段

seq = (unsigned short)buffer[0];

//随机法模拟包是否丢失

b = lossInLossRatio(packetLossRatio);

if (b){

printf("The packet with a seq of %d loss\n", seq);

continue;

}

printf("recv a packet with a seq of %d\n", seq);

//如果是期待的包，正确接收，正常确认即可

if (!(waitSeq - seq)){

++waitSeq;

if (waitSeq == 21){

waitSeq = 1;

}

//输出数据

//printf("%s\n",&buffer[1]);

buffer[0] = seq;

recvSeq = seq;

buffer[1] = '\0';

}

else{

//如果当前一个包都没有收到，则等待 Seq 为 1 的数据包，不是则不返回 ACK（因为并没有上一个正确的 ACK）

if (!recvSeq){

continue;

}

buffer[0] = recvSeq;

buffer[1] = '\0';

}

b = lossInLossRatio(ackLossRatio);

if (b){

printf("The ack of %d loss\n", (unsigned

char)buffer[0]);

continue;

}

sendto(socketClient, buffer, 2, 0,

(SOCKADDR\*)&addrServer, sizeof(SOCKADDR));

printf("send a ack of %d\n", (unsigned char)buffer[0]);

break;

}

Sleep(500);

}

}

sendto(socketClient, buffer, strlen(buffer) + 1, 0,

(SOCKADDR\*)&addrServer, sizeof(SOCKADDR));

ret =

recvfrom(socketClient, buffer, BUFFER\_LENGTH, 0, (SOCKADDR\*)&addrServer,

&len);

printf("%s\n", buffer);

if (!strcmp(buffer, "Good bye!")){

break;

}

printTips();

}

//关闭套接字

closesocket(socketClient);

WSACleanup();

return 0;

}

**Server.cpp**

#include <stdlib.h>

#include <time.h>

#include <WinSock2.h>

#include <fstream>

#pragma comment(lib,"ws2\_32.lib")

#define SERVER\_PORT 12340 //端口号

#define SERVER\_IP "0.0.0.0" //IP 地址

const int BUFFER\_LENGTH = 1026; //缓冲区大小，（以太网中 UDP 的数据帧中包长度应小于 1480 字节）

const int SEND\_WIND\_SIZE = 10;//发送窗口大小为 10，GBN 中应满足 W + 1 <=N（W 为发送窗口大小，N 为序列号个数）

//本例取序列号 0...19 共 20 个

//如果将窗口大小设为 1，则为停-等协议

const int SEQ\_SIZE = 20; //序列号的个数，从 0~19 共计 20 个

//由于发送数据第一个字节如果值为 0，则数据会发送失败

//因此接收端序列号为 1~20，与发送端一一对应

BOOL ack[SEQ\_SIZE];//收到 ack 情况，对应 0~19 的 ack

int curSeq;//当前数据包的 seq

int curAck;//当前等待确认的 ack

int totalSeq;//收到的包的总数

int totalPacket;//需要发送的包总数

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: getCurTime

// FullName: getCurTime

// Access: public

// Returns: void

// Qualifier: 获取当前系统时间，结果存入 ptime 中

// Parameter: char \* ptime

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void getCurTime(char \*ptime){

char buffer[128];

memset(buffer, 0, sizeof(buffer));

time\_t c\_time;

struct tm \*p;

time(&c\_time);

p = localtime(&c\_time);

sprintf\_s(buffer, "%d/%d/%d %d:%d:%d",

p->tm\_year + 1900,

p->tm\_mon,

p->tm\_mday,

p->tm\_hour,

p->tm\_min,

p->tm\_sec);

strcpy\_s(ptime, sizeof(buffer), buffer);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: seqIsAvailable

// FullName: seqIsAvailable

// Access: public

// Returns: bool

// Qualifier: 当前序列号 curSeq 是否可用

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bool seqIsAvailable(){

int step;

step = curSeq - curAck;

step = step >= 0 ? step : step + SEQ\_SIZE;

//序列号是否在当前发送窗口之内

if (step >= SEND\_WIND\_SIZE){

return false;

}

if (ack[curSeq]){

return true;

}

return false;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: timeoutHandler

// FullName: timeoutHandler

// Access: public

// Returns: void

// Qualifier: 超时重传处理函数，滑动窗口内的数据帧都要重传

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void timeoutHandler(){

printf("Timer out error.\n");

int index;

for (int i = 0; i< SEND\_WIND\_SIZE; ++i){

index = (i + curAck) % SEQ\_SIZE;

ack[index] = TRUE;

}

totalSeq -= SEND\_WIND\_SIZE;

curSeq = curAck;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: ackHandler

// FullName: ackHandler

// Access: public

// Returns: void

// Qualifier: 收到 ack，累积确认，取数据帧的第一个字节

//由于发送数据时，第一个字节（序列号）为 0（ASCII）时发送失败，因此加一了，此处需要减一还原

// Parameter: char c

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void ackHandler(char c){

unsigned char index = (unsigned char)c - 1; //序列号减一

printf("Recv a ack of %d\n", index);

if (curAck <= index){

for (int i = curAck; i <= index; ++i){

ack[i] = TRUE;

}

curAck = (index + 1) % SEQ\_SIZE;

}

else{

//ack 超过了最大值，回到了 curAck 的左边

for (int i = curAck; i< SEQ\_SIZE; ++i){

ack[i] = TRUE;

}

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

ack[i] = TRUE;

}

curAck = index + 1;

}

}

//主函数

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

{

//加载套接字库（必须）

WORD wVersionRequested;

WSADATA wsaData;

//套接字加载时错误提示

int err;

//版本 2.2

wVersionRequested = MAKEWORD(2, 2);

//加载 dll 文件 Scoket 库

err = WSAStartup(wVersionRequested, &wsaData);

if (err != 0){

//找不到 winsock.dll

printf("WSAStartup failed with error: %d\n", err);

return -1;

}

if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) != 2)

{

printf("Could not find a usable version of Winsock.dll\n");

WSACleanup();

}

else{

printf("The Winsock 2.2 dll was found okay\n");

}

SOCKET sockServer = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP);

//设置套接字为非阻塞模式

int iMode = 1; //1：非阻塞，0：阻塞

ioctlsocket(sockServer, FIONBIO, (u\_long FAR\*) &iMode);//非阻塞设置

SOCKADDR\_IN addrServer; //服务器地址

//addrServer.sin\_addr.S\_un.S\_addr = inet\_addr(SERVER\_IP);

addrServer.sin\_addr.S\_un.S\_addr = htonl(INADDR\_ANY);//两者均可

addrServer.sin\_family = AF\_INET;

addrServer.sin\_port = htons(SERVER\_PORT);

err = bind(sockServer, (SOCKADDR\*)&addrServer, sizeof(SOCKADDR));

if (err){

err = GetLastError();

printf("Could not bind the port %d for socket.Error code is %d\n",SERVER\_PORT,err);

WSACleanup();

return -1;

}

SOCKADDR\_IN addrClient; //客户端地址

int length = sizeof(SOCKADDR);

char buffer[BUFFER\_LENGTH]; //数据发送接收缓冲区

ZeroMemory(buffer, sizeof(buffer));

//将测试数据读入内存

std::ifstream icin;

icin.open("../test.txt");

char data[1024 \* 113];

ZeroMemory(data, sizeof(data));

icin.read(data, 1024 \* 113);

icin.close();

totalPacket = sizeof(data) / 1024;

int recvSize;

for (int i = 0; i < SEQ\_SIZE; ++i){

ack[i] = TRUE;

}

while (true){

//非阻塞接收，若没有收到数据，返回值为-1

recvSize =

recvfrom(sockServer, buffer, BUFFER\_LENGTH, 0, ((SOCKADDR\*)&addrClient), &length);

if (recvSize < 0){

Sleep(200);

continue;

}

printf("recv from client: %s\n", buffer);

if (strcmp(buffer, "-time") == 0){

getCurTime(buffer);

}

else if (strcmp(buffer, "-quit") == 0){

strcpy\_s(buffer, strlen("Good bye!") + 1, "Good bye!");

}

else if (strcmp(buffer, "-testgbn") == 0){

//进入 gbn 测试阶段

//首先 server（server 处于 0 状态）向 client 发送 205 状态码（server进入 1 状态）

//server 等待 client 回复 200 状态码，如果收到（server 进入 2 状态），则开始传输文件，否则延时等待直至超时\

//在文件传输阶段，server 发送窗口大小设为

ZeroMemory(buffer, sizeof(buffer));

int recvSize;

int waitCount = 0;

printf("Begain to test GBN protocol,please don't abort the process\n");

//加入了一个握手阶段

//首先服务器向客户端发送一个 205 大小的状态码（我自己定义的）表示服务器准备好了，可以发送数据

//客户端收到 205 之后回复一个 200 大小的状态码，表示客户端准备好了，可以接收数据了

//服务器收到 200 状态码之后，就开始使用 GBN 发送数据了

printf("Shake hands stage\n");

int stage = 0;

bool runFlag = true;

while (runFlag){

switch (stage){

case 0://发送 205 阶段

buffer[0] = 205;

sendto(sockServer, buffer, strlen(buffer) + 1, 0,

(SOCKADDR\*)&addrClient, sizeof(SOCKADDR));

Sleep(100);

stage = 1;

break;

case 1://等待接收 200 阶段，没有收到则计数器+1，超时则放弃此次“连接”，等待从第一步开始

recvSize =

recvfrom(sockServer, buffer, BUFFER\_LENGTH, 0, ((SOCKADDR\*)&addrClient), &length);

if (recvSize < 0){

++waitCount;

if (waitCount > 20){

runFlag = false;

printf("Timeout error\n");

break;

}

Sleep(500);

continue;

}

else{

if ((unsigned char)buffer[0] == 200){

printf("Begin a file transfer\n");

printf("File size is %dB, each packet is 1024B and packet total num is %d\n",sizeof(data),totalPacket);

curSeq = 0;

curAck = 0;

totalSeq = 0;

waitCount = 0;

stage = 2;

}

}

break;

case 2://数据传输阶段

if (seqIsAvailable()){

//发送给客户端的序列号从 1 开始

buffer[0] = curSeq + 1;

ack[curSeq] = FALSE;

//数据发送的过程中应该判断是否传输完成

//为简化过程此处并未实现

memcpy(&buffer[1], data + 1024 \* totalSeq, 1024);

printf("send a packet with a seq of %d\n", curSeq);

sendto(sockServer, buffer, BUFFER\_LENGTH, 0,

(SOCKADDR\*)&addrClient, sizeof(SOCKADDR));

++curSeq;

curSeq %= SEQ\_SIZE;

++totalSeq;

Sleep(500);

}

//等待 Ack，若没有收到，则返回值为-1，计数器+1

recvSize =

recvfrom(sockServer, buffer, BUFFER\_LENGTH, 0, ((SOCKADDR\*)&addrClient), &length);

if (recvSize < 0){

waitCount++;

//20 次等待 ack 则超时重传

if (waitCount > 20)

{

timeoutHandler();

waitCount = 0;

}

}

else{

//收到 ack

ackHandler(buffer[0]);

waitCount = 0;

}

Sleep(500);

break;

}

}

}

sendto(sockServer, buffer, strlen(buffer) + 1, 0, (SOCKADDR\*)&addrClient,

sizeof(SOCKADDR));

Sleep(500);

}

//关闭套接字，卸载库

closesocket(sockServer);

WSACleanup();

return 0;

}