**武汉大学计算机学院**

**本科生课程设计报告**

**签到系统的客户端/服务器端的C++实现**

专 业 名 称 ：软件工程

课 程 名 称 ：网络及分布式计算实验

指 导 教 师 ：胡继成 职称：教授

学 生 学 号 ：2016302580320

学 生 姓 名 ：任思远

二○壹玖年陆月

**郑 重 声 明**

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名： 日期：

摘要

实验目的是开发一个签到系统的客户端及服务端程序，使用C++编程语言。

实验内容主要包括：客户端程序开发，服务器端程序开发，windows socket编程。

**关键词：**签到系统，windows socket编程，C++多线程

**目录**

目录

[1. 实验目的和意义 4](#_Toc10806379)

[1.1实验目的 4](#_Toc10806380)

[1.2 实验意义 5](#_Toc10806381)

[2. 实验设计 5](#_Toc10806382)

[2.1 概述 5](#_Toc10806383)

[2.2 实验原理 5](#_Toc10806384)

[2.3 关于实验方案不足之处的讨论 12](#_Toc10806385)

[参考文献 12](#_Toc10806386)

# 实验目的和意义

## 1.1实验目的

本实验为开发一个签到系统的客户端及服务端程序，使用C++编程语言。

主要功能有：

1. 服务器端生成随机token发送给客户端
2. 从客户端提取MAC地址及DHCP并通过token做简单加密
3. 服务器端根据MAC地址及DHCP统计签到信息

## 1.2 实验意义

通过本次实验，将掌握windows下网络编程的基本函数调用，理解windows socket的工作原理，并练习C++编程及windows编程。

# 2. 实验设计

## 2.1 概述

开发模块分为客户端程序和服务器端程序。

客户端程序：接受服务器端token，并获取本机MAC地址和DHCP，简单加密后发送给服务器端。

服务器端程序：向客户端发送token，并监听端口，接受MAC地址和DHCP，并统计签到信息。

## 2.2 实验原理

基础函数：windows下获取MAC地址和DHCP

// 通过WindowsNT/Win2000中内置的NetApi32.DLL的功能来实现的。首先通过发送NCBENUM命令,获取网卡的

// 数目和每张网卡的内部编号,然后对每个网卡标号发送NCBASTAT命令获取其MAC地址。

int getMAC(char\* mac, char\* dhcp); // 用NetAPI来获取网卡MAC地址

获取MAC地址：

NCB ncb; // 定义一个NCB(网络控制块)类型的结构体变量ncb

typedef struct \_ASTAT\_ // 自定义一个结构体\_ASTAT\_

{

ADAPTER\_STATUS adapt;

NAME\_BUFFER NameBuff[30] = { 0 };

}ASTAT, \*PASTAT;

ASTAT Adapter;

typedef struct \_LANA\_ENUM // 自定义一个结构体\_LANA\_ENUM

{

UCHAR length;

UCHAR lana[MAX\_LANA]; //存放网卡MAC地址

}LANA\_ENUM;

LANA\_ENUM lana\_enum;

// 取得网卡信息列表

UCHAR uRetCode;

memset(&ncb, 0, sizeof(ncb)); // 将已开辟内存空间ncb 的值均设为值 0

memset(&lana\_enum, 0, sizeof(lana\_enum)); // 清空一个结构类型的变量lana\_enum，赋值为0

// 对结构体变量ncb赋值

ncb.ncb\_command = NCBENUM; // 统计系统中网卡的数量

ncb.ncb\_buffer = (unsigned char \*)&lana\_enum; // ncb\_buffer成员指向由LANA\_ENUM结构填充的缓冲区

ncb.ncb\_length = sizeof(LANA\_ENUM);

// 向网卡发送NCBENUM命令，以获取当前机器的网卡信息，如有多少个网卡，每个网卡的编号（MAC地址）

uRetCode = Netbios(&ncb); // 调用netbois(ncb)获取网卡序列号

if (uRetCode != NRC\_GOODRET)

return uRetCode;

// 对每一个网卡，以其网卡编号为输入编号，获取其MAC地址

for (int lana = 0; lana < lana\_enum.length; lana++)

{

ncb.ncb\_command = NCBRESET; // 对网卡发送NCBRESET命令，进行初始化

ncb.ncb\_lana\_num = lana\_enum.lana[lana];

uRetCode = Netbios(&ncb);

if (uRetCode != NRC\_GOODRET)

return uRetCode;

// 准备取得接口卡的状态块取得MAC地址

memset(&ncb, 0, sizeof(ncb));

ncb.ncb\_command = NCBASTAT; // 对网卡发送NCBSTAT命令，获取网卡信息

ncb.ncb\_lana\_num = lana\_enum.lana[0]; // 指定网卡号，这里仅仅指定第一块网卡，通常为有线网卡

strcpy((char\*)ncb.ncb\_callname, "\*"); // 远程系统名赋值为\*

ncb.ncb\_buffer = (unsigned char \*)&Adapter; //指定返回的信息存放的变量

ncb.ncb\_length = sizeof(Adapter);

// 接着发送NCBASTAT命令以获取网卡的信息

uRetCode = Netbios(&ncb);

// 取得网卡的信息，并且如果网卡正常工作的话，返回标准的冒号分隔格式。

if (uRetCode != NRC\_GOODRET)

return uRetCode;

// 把网卡MAC地址格式转化为常用的16进制形式,输出到字符串mac中

sprintf(mac, "%02X-%02X-%02X-%02X-%02X-%02X",

Adapter.adapt.adapter\_address[0],

Adapter.adapt.adapter\_address[1],

Adapter.adapt.adapter\_address[2],

Adapter.adapt.adapter\_address[3],

Adapter.adapt.adapter\_address[4],

Adapter.adapt.adapter\_address[5]

);

}

获取DHCP：

PIP\_ADAPTER\_INFO pIpAdapterInfo = (PIP\_ADAPTER\_INFO)malloc(sizeof(IP\_ADAPTER\_INFO));

unsigned long size = sizeof(IP\_ADAPTER\_INFO);

// 调用GetAdaptersInfo函数,填充pIpAdapterInfo指针变量;其中size参数既是一个输入量也是一个输出量

// GetAdaptersInfo参数传递的内存空间不够,同时其传出size,表示需要的空间大小

unsigned long nRel = GetAdaptersInfo(pIpAdapterInfo, &size);

//如果函数返回的是ERROR\_BUFFER\_OVERFLOW

//则说明GetAdaptersInfo参数传递的内存空间不够,同时其传出stSize,表示需要的空间大小

if (nRel == ERROR\_BUFFER\_OVERFLOW)

{

//释放原有的空间

free(pIpAdapterInfo);

//重新申请内存空间用来存储所有网卡信息

pIpAdapterInfo = (PIP\_ADAPTER\_INFO)malloc(size);

//再次调用GetAdaptersInfo获取网卡信息

nRel = GetAdaptersInfo(pIpAdapterInfo, &size);

}

//如果函数返回的是ERROR\_SUCCESS，则说明获取网卡信息成功

PIP\_ADAPTER\_INFO pHeadIpAdapterInfo = pIpAdapterInfo;

if (nRel == ERROR\_SUCCESS)

{

sprintf(dhcp, pIpAdapterInfo->DhcpServer.IpAddress.String);

}

客户端简单加密函数：

void encode(char(&buffer)[64], const char(&token)[64])

{

for (int i = 0; i < 64; i++)

buffer[i] ^= token[i];

}

服务器端简单加密函数：

void decode(char(&buffer)[64], const char(&token)[64])

{

for (int i = 0; i < 64; i++)

buffer[i] ^= token[i];

}

客户端程序：

void run\_client()

{

std::cout << "run client:" << std::endl;

// 初始化DLL

WSADATA wsaData;

WSAStartup(MAKEWORD(2, 2), &wsaData);

// 创建套接字

SOCKET sock = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

// 向服务器发起请求

sockaddr\_in sockAddr;

memset(&sockAddr, 0, sizeof(sockAddr));

sockAddr.sin\_family = PF\_INET;

sockAddr.sin\_addr.s\_addr = inet\_addr(ip\_addr);

sockAddr.sin\_port = htons(port);

connect(sock, (SOCKADDR\*)&sockAddr, sizeof(SOCKADDR));

//接收服务器传回的数据

char token[64] = { 0 };

recv(sock, token, 64, NULL);

// 输出接收到的数据

printf("client>>> Token form server: %s\n", token);

// MAC & DHCP

char mac[64] = { 0 };

char dhcp[64] = { 0 };

getMAC(mac, dhcp);

encode(mac, token);

encode(dhcp, token);

send(sock, mac, sizeof(mac), NULL);

send(sock, dhcp, sizeof(dhcp), NULL);

//关闭套接字

closesocket(sock);

//终止使用 DLL

WSACleanup();

}

服务器端程序：

服务器端维护的一些数据结构：

static char tokens[client\_num][64] = { 0 };

static std::vector<SOCKET> clntSocks(client\_num);

static std::vector<sockaddr\_in> sockAddrs(client\_num);

static std::vector<SOCKADDR> addrs(client\_num);

struct info\_t {

std::string mac\_;

std::string dhcp\_;

};

using name\_t = std::string;

std::unordered\_map<name\_t, info\_t> attendee;

tokens 是服务器端随机生成的64Bchar数组，生成方法如下：

void init\_tokens() {

std::random\_device r;

std::default\_random\_engine e(r());

std::uniform\_int\_distribution<int> uniform\_dist(0, 63);

for (int i = 0; i < client\_num; i++) {

for (int j = 0; j < 63; j++) {

tokens[i][j] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789#$"[uniform\_dist(e)];

}

tokens[i][63] = '\0';

}

}

Attendee 用于存储签到信息。

服务器端总程序如下：

void run\_server()

{

init\_tokens();

// 初始化 DLL

WSADATA wsaData;

WSAStartup(MAKEWORD(2, 2), &wsaData);

// 创建套接字

SOCKET socks = socket(PF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

// 创建完成端口句柄

HANDLE completionPort = CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, 0, 0);

// TODO: epoll on windows

for (int i = 0; i < client\_num; i++)

{

// 绑定套接字

memset(&sockAddrs[i], 0, sizeof(sockaddr\_in)); // 每个字节都用0填充

sockAddrs[i].sin\_family = PF\_INET; // 使用IPv4地址

sockAddrs[i].sin\_addr.s\_addr = inet\_addr(client\_address[i]); // 具体的IP地址

sockAddrs[i].sin\_port = htons(port); // 端口

bind(socks, (SOCKADDR\*)&sockAddrs[i], sizeof(SOCKADDR));

// 进入监听状态

listen(socks, 20);

// 接收客户端请求

int nSize = sizeof(SOCKADDR);

clntSocks[i] = accept(socks, (SOCKADDR\*)&addrs[i], &nSize);

// 向客户端发送数据

send(clntSocks[i], tokens[i], sizeof(tokens[i]), NULL);

char buffer[64];

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

recv(clntSocks[i], buffer, sizeof(buffer), NULL);

decode(buffer, tokens[i]);

std::string mac = buffer;

printf("server>>> recv from client \"%s\" MAC: %s\n", client\_name[i], buffer);

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

recv(clntSocks[i], buffer, sizeof(buffer), NULL);

decode(buffer, tokens[i]);

std::string dhcp = buffer;

printf("server>>> recv from client \"%s\" DHCP: %s\n", client\_name[i], buffer);

attendee[client\_name[i]] = { std::move(mac), std::move(dhcp) };

// 关闭套接字

closesocket(clntSocks[i]);

}

closesocket(socks);

// 终止 DLL 的使用

WSACleanup();

std::cout << "atendee info:" << endl;

for (auto const&[name, info] : attendee) {

std::cout << name << "\t";

std::cout << "MAC: " << info.mac\_ << "\t";

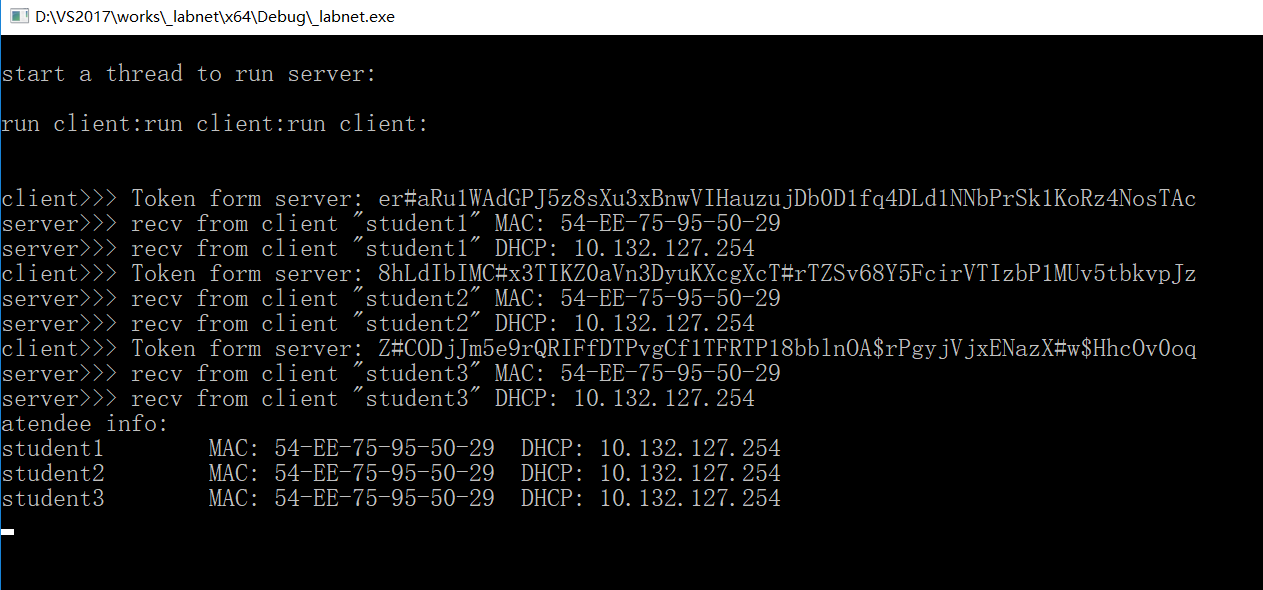
std::cout << "DHCP: " << info.dhcp\_ << "\t";

std::cout << endl;

}

}

运行截图：



## 2.3 关于实验方案不足之处的讨论

从上面的服务器端程序实现可以看出，服务器端与客户端交互时单线程按顺序执行的：

// TODO: epoll on windows

for (int i = 0; i < client\_num; i++)

{

// …

}

这样，如果某一个客户端出现异常而卡住，就会引发服务器端程序阻塞，那么就不能与之后的客户端交互。而如果为每一个客户端都创建一个线程，则会导致开销过大，服务端负载高。一个比较合理的解决方案是使用epoll，即所谓的IO多路复用，我们将socket注册为一个event，并附上回调函数，将其变为一个事件驱动模型。

# 参考文献

[1] C++之获取网卡物理地址（MAC）

<https://blog.csdn.net/pbl18392021230/article/details/72673417>

[2] c++ windows 获取mac地址

<https://blog.csdn.net/lxj434368832/article/details/80207460>

[3] C/C++ socket编程教程之三：Windows下的socket程序

<https://blog.csdn.net/Ctrl_qun/article/details/52454232>