**1**

**实验 1：HTTP 代理服务器的设计与实现**

|  |  |
| --- | --- |
| **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**

熟悉并掌握 Socket 网络编程的过程与技术；深入理解 HTTP 协议， 掌握 HTTP 代理服务器的基本工作原理；掌握 HTTP 代理服务器设计与编程实现的基本技能。

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. 设计并实现一个基本 HTTP 代理服务器。要求在指定端口（例如 8080）接收来自客户的 HTTP 请求并且根据其中的 URL 地址访问该地址 所指向的 HTTP 服务器（原服务器），接收 HTTP 服务器的响应报文，并 将响应报文转发给对应的客户进行浏览
       2. 扩展 HTTP 代理服务器，支持如下功能：

网站过滤：允许/不允许访问某些网站；

用户过滤：支持/不支持某些用户访问外部网站；

1. **实验总结**
   * + 1. **Socket 编程的客户端和服务器端主要步骤**

服务器端：

* + - * 1. 使用 socket 函数创建一个 socket 描述符，来唯一标识一个 socket。函数原型为： int socket(int domain, int type, int protocol)，函数返回一个
        2. 使用 bind 函数绑定 IP 地址，端口信息等。函数原型为：int bind(int sockfd, const struct sockaddr \* addr, socklen\_t addrlen)
        3. 使用 listen 函数进行监听创建的 socket。函数原型为 int listen(int sockfd, int backlog)
        4. 使用 accept 函数接收请求，此时 socket 连接也就建立好了。函数原型为 int accept(int sockfd, struct sockaddr \* addr, socklen\_t \* addrlen)
        5. 使用 read(), write() 等函数调用网络I/O 进行读写操作，来实现网络中不同进程之间的通信
        6. 使用 close 函数关闭网络连接，即关系相应的 socket 描述字。函数原型为 int close(int fd)，注：close 操作知识是相应的 socket 描述字的引用计数-1，只有当引用计数为0的时候，才会触发客户端想服务器发送终止连接请求

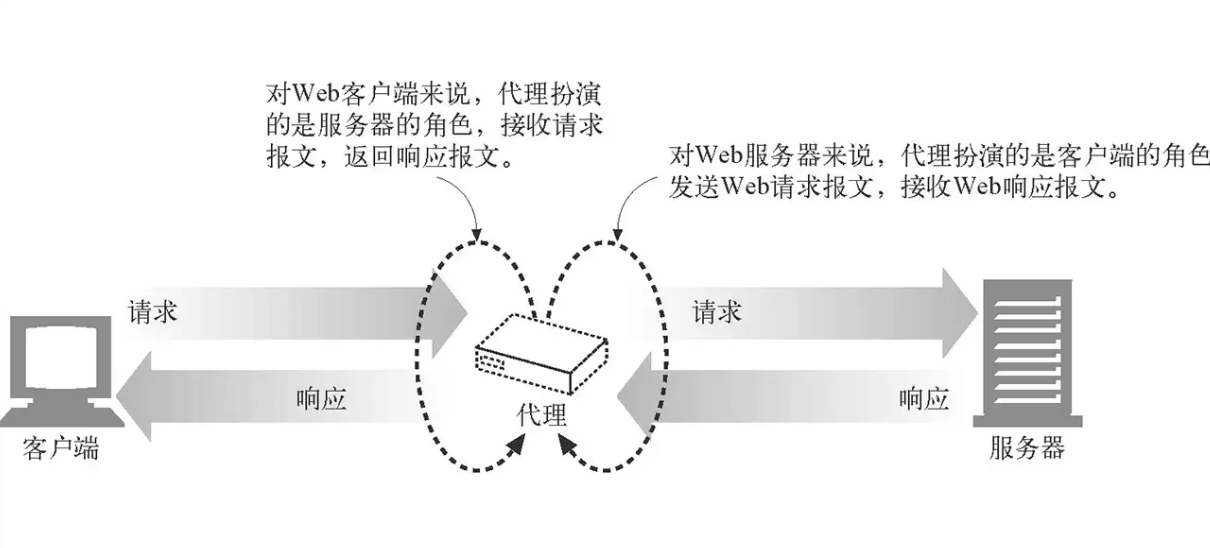
客户端：

1. 创建 socket
2. 绑定 IP 地址，端口信息
3. 设置要连接对方的 IP 地址和端口属性
4. 使用connect 函数连接服务器。 函数原型为int connect(int sockfd, const struct sockaddr \* addr, socklen\_t addrlen)
5. 使用 read(), write()等函数进行网络 I/O 的读写
6. 关闭网络连接
   * + 1. **HTTP 代理服务器的基本原理**

正常情况下去访问 HTTP 协议的网站，一般是浏览器向服务器发送一个请求，之后服务器会产生响应，从而客户端得到相应的数据。

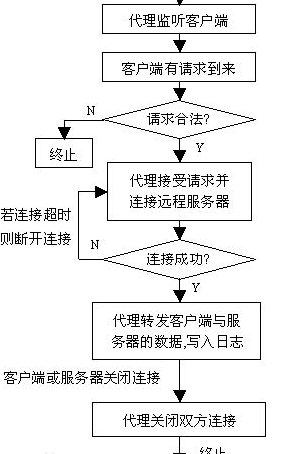
如果我们在请求与响应的中间加入一层，即 浏览器向代理服务器发送请求，代理服务器去解析这个请求，然后向真正的服务器去发送请求，然后代理服务器得到真正服务器所得到的响应，然后再发送给客户端。

也就是说，加了代理服务器之后，相对于客户端来说，代理服务器就是服务器；相对于真正的服务器来说，代理服务器就是客户端。代理服务器在整个过程中充当的是一个媒介的作用，用图来表示就是这样的：



除此之外，我们还可以利用代理服务器设置缓存，即再报文解析后代理服务器在本地查找缓存，如果有缓存并且没有过期的情况下，直接使用缓存的内容，这样能真正的减少服务器端的负荷，提高客户端的访问速度。

* + - 1. **HTTP 代理服务器的程序流程图**



* + - 1. **实现 HTTP 代理服务器的关键技术及解决方案**

1. **Socket 编程**

HTTP 的请求都是遵循相应的标准格式，因此我们针对标准的格式要进行正确的处理，来得到正确结果。

1. **代理流程的正确组织**

整个代理服务器中，相关的流程不能出现差错。即必须遵循这样的一个流程： 先建立相关的 socket 连接，然后真正的客户端首先去请求代理服务器，代理服务器去解析请求的内容，然后由他像真正的服务器发出 connect 的 socket 连接，真正的服务器去响应代理服务器的这个请求，然后做出响应发送给代理服务器，然后代理服务器再发送给真正的服务器，这样就完成了一次 HTTP 的请求与响应。

1. **HTTP 请求头的正确解析**

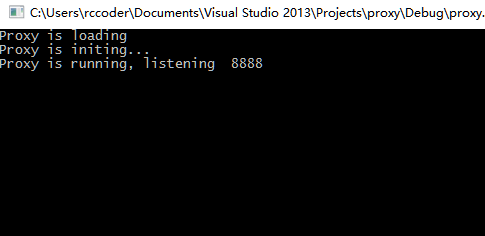
HTTP的请求头的格式是一定的，我们要针对 HTTP 的请求头去正确的分析出服务器的 host，端口号，cookies等其他信息。

* + - 1. **HTTP 代理服务器实验验证过程以及实验结果**

1. **设置浏览器代理端口**

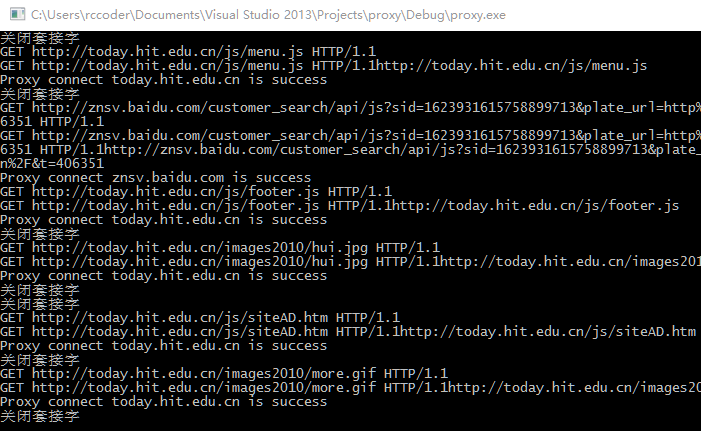


1. **运行代理**



1. **访问要访问的网站，查看代理运行**





* + - 1. **HTTP 代理服务器源代码（带有详细注释）**

#include <stdio.h>

#include <Windows.h>

#include <process.h>

#include <string.h>

#include <tchar.h>

#pragma comment(lib,"Ws2\_32.lib")

#define MAXSIZE 65507 //发送数据报文的最大长度

#define HTTP\_PORT 80 //http 服务器端口

#pragma warning(disable : 4996)

//Http 重要头部数据

struct HttpHeader{

char method[4]; // POST 或者 GET，注意有些为 CONNECT，本实验暂不考虑

char url[1024]; // 请求的 url

char host[1024]; // 目标主机

char cookie[1024 \* 10]; //cookie

HttpHeader(){

ZeroMemory(this, sizeof(HttpHeader));

}

};

BOOL InitSocket();

void ParseHttpHead(char \*buffer, HttpHeader \* httpHeader);

BOOL ConnectToServer(SOCKET \*serverSocket, char \*host);

unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter);

//代理相关参数

SOCKET ProxyServer;

/\*

sockaddr\_in 数据结构如下，协议不同相应也不同(Ipv4)

struct sockaddr\_in {

sa\_family\_t sin\_family; // 协议族，如AF\_INET

in\_port\_t sin\_port; // 端口

struct in\_addr sin\_addr; // 地址

};

struct in\_addr {

uint32\_t s\_addr;

};

\*/

sockaddr\_in ProxyServerAddr;

// 绑定的端口号

const int ProxyPort = 8888;

//由于新的连接都使用新线程进行处理，对线程的频繁的创建和销毁特别浪费资源

//可以使用线程池技术提高服务器效率

//const int ProxyThreadMaxNum = 20;

//HANDLE ProxyThreadHandle[ProxyThreadMaxNum] = {0};

//DWORD ProxyThreadDW[ProxyThreadMaxNum] = {0};

struct ProxyParam{

SOCKET clientSocket;

SOCKET serverSocket;

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

printf("Proxy is loading\n");

printf("Proxy is initing...\n");

if (!InitSocket()){

printf("socket init error!\n");

return -1;

}

printf("Proxy is running, listening %d\n", ProxyPort);

SOCKET acceptSocket = INVALID\_SOCKET;

ProxyParam \*lpProxyParam;

HANDLE hThread;

DWORD dwThreadID;

//代理服务器不断监听

while (true){

// init 过程中已经完成 socket 与 bind，描述符给 ProxyServer

// int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

acceptSocket = accept(ProxyServer, NULL, NULL);

lpProxyParam = new ProxyParam;

if (lpProxyParam == NULL){

continue;

}

lpProxyParam->clientSocket = acceptSocket;

// 调用线程执行函数

// unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter)

hThread = (HANDLE)\_beginthreadex(NULL, 0, &ProxyThread, (LPVOID)lpProxyParam, 0, 0);

CloseHandle(hThread);

Sleep(200);

}

closesocket(ProxyServer);

WSACleanup();

return 0;

}

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

// Method: InitSocket

// FullName: InitSocket

// Access: public

// Returns: BOOL

// Qualifier: 初始化套接字

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

BOOL InitSocket(){

// 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("加载 winsock 失败，错误代码为: %d\n", WSAGetLastError());

return FALSE;

}

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

{

printf("不能找到正确的 winsock 版本\n");

WSACleanup();

return FALSE;

}

/\*

创建和服务器端连接的socket

int socket(int domain, int type, int protocol)

AF\_INET决定了要用ipv4地址（32位的）与端口号（16位的）的组合

protocol 为 0 的时候会自动根据 type 选择正确的协议

函数返回的是一个 Socket 描述符，

但是返回的socket描述字它存在于协议族（address family，AF\_XXX）空间中，但没有一个具体的地址

需要调用 bind 函数来指定地址

\*/

ProxyServer = socket(AF\_INET, SOCK\_STREAM, 0);

if (INVALID\_SOCKET == ProxyServer){

printf("Create socket error: %d\n", WSAGetLastError());

return FALSE;

}

/\*

绑定 Socket 具体地址

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

struct sockaddr 数据类型如 ProxyServerAddr 定义时的注释所示

\*/

ProxyServerAddr.sin\_family = AF\_INET; //协议族

ProxyServerAddr.sin\_port = htons(ProxyPort); //端口

ProxyServerAddr.sin\_addr.S\_un.S\_addr = INADDR\_ANY; //协议地址

if (bind(ProxyServer, (SOCKADDR\*)&ProxyServerAddr, sizeof(SOCKADDR)) == SOCKET\_ERROR){

printf("Bind socket error\n");

return FALSE;

}

/\*

监听

int listen(int sockfd, int backlog);

backlog 可排队的最大连接数

\*/

if (listen(ProxyServer, SOMAXCONN) == SOCKET\_ERROR){

printf("listen %d port error\n", ProxyPort);

return FALSE;

}

return TRUE;

}

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

// Method: ProxyThread

// FullName: ProxyThread

// Access: public

// Returns: unsigned int \_\_stdcall

// Qualifier: 线程执行函数

// Parameter: LPVOID lpParameter

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

unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter){

char Buffer[MAXSIZE];

char \*CacheBuffer;

ZeroMemory(Buffer, MAXSIZE);

SOCKADDR\_IN clientAddr;

int length = sizeof(SOCKADDR\_IN);

int recvSize;

int ret;

// recv IO操作

recvSize = recv(((ProxyParam\*)lpParameter)->clientSocket, Buffer, MAXSIZE, 0);

if (recvSize <= 0){

goto error;

}

HttpHeader\* httpHeader = new HttpHeader();

CacheBuffer = new char[recvSize + 1];

ZeroMemory(CacheBuffer, recvSize + 1);

memcpy(CacheBuffer, Buffer, recvSize);

// 处理 HTTP header部分信息

ParseHttpHead(CacheBuffer, httpHeader);

delete CacheBuffer;

// 连接服务器

if (!ConnectToServer(&((ProxyParam\*)lpParameter)->serverSocket, httpHeader->host)) {

goto error;

}

printf("Proxy connect %s is success\n", httpHeader->host);

// 将客户端发送的 HTTP 数据报文直接转发给目标服务器

// send IO操作

ret = send(((ProxyParam \*)lpParameter)->serverSocket, Buffer, strlen(Buffer) + 1, 0);

//等待目标服务器返回数据

recvSize = recv(((ProxyParam\*)lpParameter)->serverSocket, Buffer, MAXSIZE, 0);

if (recvSize <= 0){

goto error;

}

//将目标服务器返回的数据直接转发给客户端

ret = send(((ProxyParam\*)lpParameter)->clientSocket, Buffer, sizeof(Buffer), 0);

//错误处理

error:

printf("关闭套接字\n");

Sleep(200);

closesocket(((ProxyParam\*)lpParameter)->clientSocket);

closesocket(((ProxyParam\*)lpParameter)->serverSocket);

delete lpParameter;

\_endthreadex(0);

return 0;

}

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

// Method: ParseHttpHead

// FullName: ParseHttpHead

// Access: public

// Returns: void

// Qualifier: 解析 TCP 报文中的 HTTP 头部

// Parameter: char \* buffer

// Parameter: HttpHeader \* httpHeader

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

void ParseHttpHead(char \*buffer, HttpHeader \* httpHeader){

char \*p;

char \*ptr;

const char \* delim = "\r\n";

p = strtok\_s(buffer, delim, &ptr);//提取第一行

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

if (p[11] == 'p') {

char \* ppp = "GET http://today.hit.edu.cn/ HTTP/1.1";

strcpy(p, ppp);

}

printf("%s", p);

if (p[0] == 'G'){//GET 方式

memcpy(httpHeader->method, "GET", 3);

memcpy(httpHeader->url, &p[4], strlen(p) - 13);

}

else if (p[0] == 'P'){//POST 方式

memcpy(httpHeader->method, "POST", 4);

memcpy(httpHeader->url, &p[5], strlen(p) - 14);

}

printf("%s\n", httpHeader->url);

p = strtok\_s(NULL, delim, &ptr);

while (p){

switch (p[0]){

case 'H'://Host

memcpy(httpHeader->host, &p[6], strlen(p) - 6);

break;

case 'C'://Cookie

if (strlen(p) > 8){

char header[8];

ZeroMemory(header, sizeof(header));

memcpy(header, p, 6);

if (!strcmp(header, "Cookie")){

memcpy(httpHeader->cookie, &p[8], strlen(p) - 8);

}

}

break;

default:

break;

}

p = strtok\_s(NULL, delim, &ptr);

}

}

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

// Method: ConnectToServer

// FullName: ConnectToServer

// Access: public

// Returns: BOOL

// Qualifier: 根据主机创建目标服务器套接字，并连接

// Parameter: SOCKET \* serverSocket

// Parameter: char \* host

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

BOOL ConnectToServer(SOCKET \*serverSocket, char \*host){

sockaddr\_in serverAddr;

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(HTTP\_PORT);

HOSTENT \*hostent = gethostbyname(host);

if (!hostent){

return FALSE;

}

in\_addr Inaddr = \*((in\_addr\*)\*hostent->h\_addr\_list);

serverAddr.sin\_addr.s\_addr = inet\_addr(inet\_ntoa(Inaddr));

\*serverSocket = socket(AF\_INET, SOCK\_STREAM, 0);

if (\*serverSocket == INVALID\_SOCKET){

return FALSE;

}

if (connect(\*serverSocket, (SOCKADDR \*)&serverAddr, sizeof(serverAddr)) == SOCKET\_ERROR){

closesocket(\*serverSocket);

return FALSE;

}

return TRUE;

}