# DNS中继服务器实验报告

**小组成员：** 赵文龙 2019211278

魏酉全 2019211274

**一、系统功能设计**

设计一个 DNS 服务器程序，读入 **IP 地址-域名** *(hosts)* 对照表，当客户端查询域名对应的 IP 地址时，用 域名检索该对照表，有三种可能检索结果：

1.检索结果 IP 地址 0.0.0.0 ，则向客户端返回 **域名不存在** 的报错消息（不良网站拦截功能）；

2.检索结果为普通 IP 地址，则向客户端返回该地址（服务器功能） ；

3.表中未检到该域名，则向因特网 DNS 服务器发出查询，并将结果返给客户端（中继功能）考虑多个计算机上的客户端会同时查询，需要进行消息 ID 的转换 。

注：代码附在文件之后。请使用Visual Studio打开工程文件。

**二、模块设计**

我们的DNS中继服务器程序主要分为下列模块：报文转换模块、域名处理模块、监听线程模块、请求池模块、主程序模块五大模块。

**报文转换模块 (MessageConversion)**

1.将从Socket收到的网络字节流解析为自定义的DNS数据包格式

2.将自定义的DNS数据包格式转化为Socket发送的字节流形式

**域名处理模块 （DomainAnalysis）**

1.从文件中获得域名和对应的ip地址

2.根据DNS报文中得到的域名找到ip地址和判定域名的类型

3.根据域名类型发送返回报文

**监听线程模块 （MainHeader）**

1.初始化、建立、关闭Socket

2.接收、发送Socket字节流

3.监听线程、DNS咨询线程的线程函数

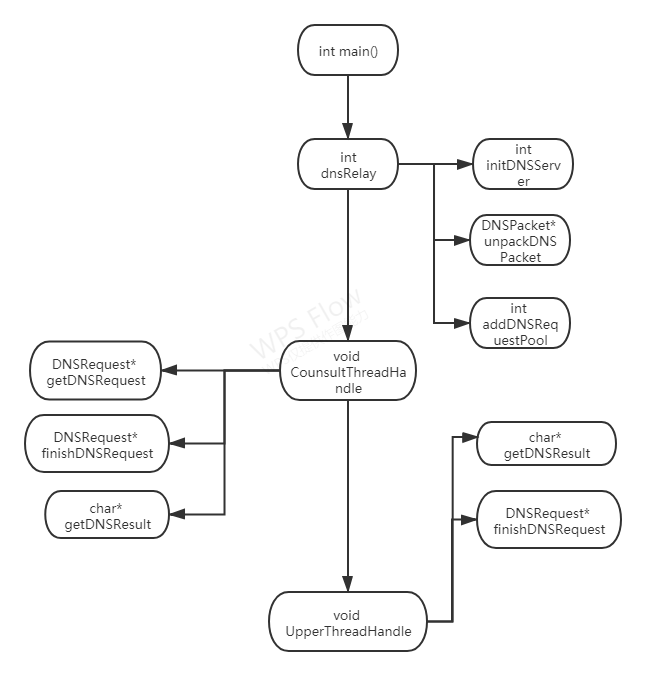
**请求池模块 （RequestPool）**

进行请求池的添加、获取、删除请求。

**主程序模块 （Main）**

综合各个模块，完成所有功能

**三、基本流程图:**

****

**四、运行逻辑：**

本程序由多线程实现，其中包含三个主要线程：客户端监听线程、DNS查询线程和DNS监听线程。

程序结构设计为生产者消费者模式，生产者为客户端监听线程，消费者为DNS查询线程池。所有请求存储在请求池request\_pool中。

**1、客户端监听线程**：

监听线程Listen\_Thread在主函数中持续性地监听本机ip的53端口，当接受到 UDP报文时，通过一定的筛选（如果不进行筛选，有时候会收到模式IP发来的垃圾报文，因此限定了发送ip只能是固定的“127.0.0.1”），将DNS请求报文转化为自定义的DNSRequset结构方便存储，并存储到空闲的请求池的位置中。当DNS请求被存储到请求池中之后，会将该请求报头的ID修改为当前所在请求池中的位置，这种操作保证了请求池中的每一个请求ID的唯一性，不会出现多客户端发来的请求报头ID重复的现象。

重要函数代码部分：

int dnsRelay() {//监听端口报文信息并加入请求池

loadHostsFromTxt();

for (int i = 0; i < MAX\_CACHED\_ITEM; i++){

cached\_list[i] = (cache\_item\*)malloc(sizeof(cache\_item));

cached\_list[i]->webaddr = (char\*)malloc(DEFAULT\_BUFLEN);

cached\_list[i]->occupied = false;

}

int i\_result = 0;

SOCKET Listen\_Socket;

i\_result = initDNSServer(&Listen\_Socket);

if (i\_result == 1) return 255;

request\_pool = (ReqPool\*)malloc(sizeof(ReqPool) \* MAX\_REQ);

if (request\_pool == NULL) {

printf("Fail to initialize request pool.\n");

exit(100);

}

for (int i = 0; i < MAX\_REQ; i++)

request\_pool[i].available = 1;

for (int i = 0; i < MAX\_THREAD; i++)

dns\_consulting\_threads[i] = new thread(CounsultThreadHandle, UPPER\_DNS, Listen\_Socket, i);

printf("Initialize completed.\n");

while (1) {

char\* recvbuf = (char\*)malloc(sizeof(char) \* DEFAULT\_BUFLEN);

struct sockaddr\_in clt\_addr;

int clt\_addr\_len = sizeof(clt\_addr);

if (sizeof(recvbuf) <= 0) {

printf("Fail to allocate buff.\n");

exit(100);

}

memset(recvbuf, 0, sizeof(recvbuf));

ZeroMemory(&clt\_addr, clt\_addr\_len);

//Receive DNS Requests.

if (recvfrom(Listen\_Socket, recvbuf, DEFAULT\_BUFLEN, 0, (struct sockaddr\*)&clt\_addr, &clt\_addr\_len) == SOCKET\_ERROR)

printf("[Listen\_Socket]: recvfrom client error with: %d\n\n", WSAGetLastError());//一直等待数据的到来

else {

if (strcmp("127.0.0.1", inet\_ntoa(clt\_addr.sin\_addr)) != 0){

printf("Receive bad message!\n");

continue;

}

printf("[Listen\_Socket]: %d Bytes received from IP(%s): %d\n",sizeof(recvbuf),inet\_ntoa(clt\_addr.sin\_addr), i\_result);

DNSRequest\* new\_req = (DNSRequest\*)malloc(sizeof(DNSRequest));

new\_req->packet = unpackDNSPacket(recvbuf);

new\_req->processed = 0;

new\_req->client\_addr = clt\_addr;

new\_req->client\_addr\_len = clt\_addr\_len;

if (addDNSRequestPool(new\_req) == -1) {

printf("[Listen\_Socket]:Now on traffic,wait patiently.\n");

Sleep(1000);

}

}

}

**2、DNS查询线程**:

查询线程持续从请求池尝试获取新的请求，当获取到新的请求后，在host文件中查找该请求的域名，如果有，则将根据得到的ip地址产生新的应答报文（注意：此时的应答报文的id要与收到的id一致）；如果无，则说明要向真正的DNS服务器查询。因此将该请求的id修改为新的id（新的id为该请求在请求池中的位置，这样的设计保证了新id的唯一性，不会产生id重复的错误）之后发送给DNS服务器 。

重要函数代码部分：

void CounsultThreadHandle(const char\* upper\_DNS\_addr, SOCKET listen\_socket, int t\_id) {

printf("[Consulting Thread %d]: Created.\n", t\_id);

char\* sendbuf = (char\*)malloc(DEFAULT\_BUFLEN);

char\* dnsbuf = (char\*)malloc(DEFAULT\_BUFLEN);

int i\_result = 0;

//上层DNS服务器的地址

struct sockaddr\_in servaddr;

ZeroMemory(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(DNS\_PORT);

inet\_pton(AF\_INET, upper\_DNS\_addr, &servaddr.sin\_addr);

struct sockaddr\_in myaddr;

ZeroMemory(&myaddr, sizeof(myaddr));

myaddr.sin\_family = AF\_INET;

myaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

myaddr.sin\_port = htons(10000 + t\_id);

// 初始化上层DNS线程

SOCKET upper\_dns\_socket = socket(AF\_INET, SOCK\_DGRAM, 0);

::bind(upper\_dns\_socket, (struct sockaddr\*)&myaddr, sizeof(myaddr));

thread return\_thread = thread(UpperThreadHandle, upper\_dns\_socket, listen\_socket, t\_id);

while (1) {

DNSRequest\* req = NULL;

while (req == NULL) {

Sleep(20);

req = getDNSRequest();

time\_t endTime = time(NULL);

if (time\_mutex.try\_lock()) {

for (int i = 0; i < MAX\_REQ; i++) {

if (!request\_pool[i].available){

if (request\_pool[i].req->processed == true) {

if (difftime(endTime, request\_pool[i].startTime) > MAX\_REQ\_TTL) {

DNSRequest\* req = request\_pool[i].req;

DNSPacket\* recv\_packet = req->packet;

finishDNSRequest(req->new\_id);

}

break;

}

}

}

time\_mutex.unlock();

}

}

printf("[Consulting Thread %d]:Got DNSReq %d\n", t\_id, req->new\_id);

DNSPacket\* recv\_packet = req->packet;

u\_int ip\_addr = 0;

ADDR\_TYPE addr\_type = getAddrType(recv\_packet->p\_qpointer[0]->q\_qname, &ip\_addr);

printf("[Consulting Thread %d]:Search type finished, type: %d\n", t\_id, addr\_type);

switch (addr\_type) {

case BLOCKED:

case CACHED:

{

int sendbuflen;

sendbuf = getDNSResult(recv\_packet, req->old\_id, ip\_addr, addr\_type, sendbuflen);

printf("[Consulting Thread %d]:Start sending result to client\n\n", t\_id);

i\_result = sendto(listen\_socket, sendbuf, sendbuflen, 0, (struct sockaddr\*)&(req->client\_addr), req->client\_addr\_len);

if (i\_result == SOCKET\_ERROR)

printf("[Consulting Thread %d]:sendto() failed with error code : %d\n", t\_id, WSAGetLastError());

else

printf("[Consulting Thread %d]:Bytes send back to 127.0.0.1: %d\n", t\_id, i\_result);

finishDNSRequest(req->new\_id);

}

break;

case ADDR\_NOT\_FOUND:

{

int packet\_length;

ushort p\_id = req->new\_id;

recv\_packet->p\_header->h\_id = p\_id;

char\* send\_string = packDNSPacket(recv\_packet, packet\_length);

printf("[Consulting Thread %d]:Start consulting Upper DNS: %s\n", t\_id, upper\_DNS\_addr);

if (sendto(upper\_dns\_socket, send\_string, packet\_length, 0, (struct sockaddr\*)&servaddr, sizeof(servaddr)) == SOCKET\_ERROR)

printf("[Consulting Thread %d]:sendto() failed with error code : %d\n", t\_id, WSAGetLastError());

}

break;

}

}

}

**3、DNS监听线程**：

监听线程只监听由DNS服务器发来的应答报文。收到报文后，根据报文id删除请求池中的请求，并将id修改为客户端发来的id，发回给客户端。此外，DNS监听线程会从DNS发回的应答报文中获取域名及其对应的IP地址保存在 cached\_list 中，达到host信息缓存的功能。下一次查询相同域名时，将直接从缓存中得到对应的IP。

重要函数代码部分：

void UpperThreadHandle(SOCKET upper\_dns\_socket, SOCKET listen\_packet, int t\_id) {

int i\_result = 0, sleeptime = 20;

struct sockaddr\_in servaddr;

int servaddrlen = sizeof(servaddr);

char\* dnsbuf = (char\*)malloc(DEFAULT\_BUFLEN);

while (true) {

i\_result = recvfrom(upper\_dns\_socket, dnsbuf, DEFAULT\_BUFLEN, 0, (struct sockaddr\*)&servaddr, &servaddrlen);

if (i\_result == SOCKET\_ERROR) {

if (WSAGetLastError() == WSAEWOULDBLOCK) {//接受或发送缓存已满

Sleep(sleeptime);

continue;

}

else {

printf("[DNS Thread %d]:! recvfrom\_server() failed with error code : %d\n", t\_id, WSAGetLastError());

break;

}

}

else {

printf("[DNS Thread %d]:Bytes received from DNS\_SERVER: %d\n", t\_id, DEFAULT\_BUFLEN);

int p\_id = ntohs(\*(ushort\*)dnsbuf);

if (request\_pool[p\_id].available == true) continue;

DNSPacket\* return\_pack = unpackDNSPacket(dnsbuf);

//存入缓存

if (return\_pack->p\_rpointer[0]->r\_rdata != NULL && return\_pack->p\_qpointer[0]->q\_qtype == 1){

u\_int\* rdata\_pointer = (u\_int\*)return\_pack->p\_rpointer[0]->r\_rdata;

u\_int ip\_uint = (\*rdata\_pointer);

in\_addr inaddr;

inaddr.S\_un.S\_addr = ip\_uint;

char\* ipaddr = inet\_ntoa(inaddr);

//if (strcmp(ipaddr,"3.119.119.119") == 0) continue;

char\* webaddr = (char\*)malloc(DEFAULT\_BUFLEN);

strcpy(webaddr, return\_pack->p\_qpointer[0]->q\_qname);

// 将不可见的字符转化为.方便对比

for (int i = 0; i < strlen(webaddr); i++){

if (webaddr[i] < 0x20)

webaddr[i] = '.';

else if (webaddr[i] >= 'A' && webaddr[i] <= 'Z')

webaddr[i] -= 'A' - 'a';

}

printf("[DNS Thread %d]:Domain: %s,IP: %s saved in CACHE\n", t\_id, webaddr, ipaddr);

cache\_mutex.lock();

for (int i = 0; i < MAX\_CACHED\_ITEM; i++){

if (cached\_list[i]->occupied) continue;

cached\_list[i]->occupied = true;

inet\_pton(AF\_INET, ipaddr, &cached\_list[i]->ip\_addr);

strcpy(cached\_list[i]->webaddr, webaddr + 1);

break;

}

cache\_mutex.unlock();

}

DNSRequest\* req = finishDNSRequest(p\_id);

\*(ushort\*)dnsbuf = htons(req->old\_id);

i\_result = sendto(listen\_packet, dnsbuf, DEFAULT\_BUFLEN, 0, (struct sockaddr\*)&(req->client\_addr), req->client\_addr\_len);

char\* client\_ipaddr = inet\_ntoa(req->client\_addr.sin\_addr);

if (i\_result == SOCKET\_ERROR)

printf("[DNS Thread %d]:sendto() failed with error code : %d\n", t\_id, WSAGetLastError());

else

printf("[DNS Thread %d]:Bytes send to %s : %d\n", t\_id, client\_ipaddr, i\_result);

}

}

}

**五、关键全局变量、结构体和函数**

**1、全局变量和结构体：**

#define DNS\_PORT 53 //DNS serves on port 53

#define DEFAULT\_BUFLEN 1024

#define DNS\_HEADER\_LEN 12

#define MAX\_HOST\_ITEM 1200

#define MAX\_CACHED\_ITEM 200

#define MAX\_REQ 1000

#define UPPER\_DNS "10.3.9.44"

#define MAX\_THREAD 5

#define MAX\_REQ\_TTL 10

host\_item\* hosts\_list[MAX\_HOST\_ITEM];

cache\_item\* cached\_list[MAX\_CACHED\_ITEM];

ReqPool\* request\_pool;

mutex id\_mutex, pool\_mutex, req\_counter\_mutex, time\_mutex, cache\_mutex;

int req\_counter = 0, host\_counter = 0;

thread\* dns\_consulting\_threads[MAX\_THREAD];

typedef unsigned short ushort;

typedef unsigned int u\_int;

enum Query\_QR {

Q\_QUERY = 0, Q\_RESPONSE

};

typedef struct DnsHeader

{

ushort h\_id;

bool h\_qr;

ushort h\_opcode;

bool h\_aa;

bool h\_tc;

bool h\_rd;

bool h\_ra;

char h\_rcode;

ushort h\_QDCount;

ushort h\_ANCount;

ushort h\_NSCount;

ushort h\_ARCount;

}DNSHeader;

typedef struct DnsQuery

{

char \*q\_qname;

ushort q\_qtype;

ushort q\_qclass;

}DNSQuery;

typedef struct DnsResponse

{

char \*r\_name;

ushort r\_type;

ushort r\_class;

int r\_ttl;

ushort r\_rdlength;

char\* r\_rdata;

}DNSResponse;

typedef struct DnsPacket

{

Query\_QR p\_qr;

DnsHeader \*p\_header;

DnsQuery \*p\_qpointer[50];

DnsResponse \*p\_rpointer[50];

}DNSPacket;

typedef struct DnsRequest

{

bool processed;

int old\_id;//用户发来ID

int new\_id;//发给DNS服务器ID

DNSPacket\* packet;

struct sockaddr\_in client\_addr;//客户端地址信息以应答报文发给客户

int client\_addr\_len;

}DNSRequest;

typedef struct RequestPool//请求池表

{

bool available;

DNSRequest\* req;

time\_t startTime;

}ReqPool;

enum ADDR\_TYPE {

BLOCKED = 100, //IP域名不存在

CACHED, //已在缓存池里

ADDR\_NOT\_FOUND //需向上层服务器查找

};

typedef struct host\_item {

u\_int ip\_addr;

char\* webaddr;

ADDR\_TYPE type;

}host\_item;

typedef struct cache\_item {

u\_int ip\_addr;

char\* webaddr;

int ttl;

int occupied;

}cache\_item;

// 存储dnsrelay.txt中的所有host信息

host\_item \*hosts\_list[MAX\_HOST\_ITEM];

// 请求池

ReqPool \*request\_pool = (ReqPool\*)malloc(sizeof(ReqPool)\*MAX\_REQ);

// 线程锁

mutex id\_mutex, pool\_mutex, req\_counter\_mutex;

// 计数器

int req\_counter = 0, host\_counter = 0;

// 查询线程池

thread \*dns\_consulting\_threads[MAX\_THREAD];

\* 线程从请求池中取出请求，根据域名种类进行不同情况的操作

\* 线程处理函数的参数不能有引用或者指针，指针只能指向常量，如const char\*

\* 处理逻辑：若是BLOCKED或者CACHED则根据得到的ip构造新的应答报文发回给client

处理逻辑：若是NOT\_FOUND则将报文发给DNS\_SERVER（注意要更改id），再将得到的报文发回给

client（由另一线程完成）

\*/

void CounsultThreadHandle(const char\*, SOCKET, int);

/\*\*

\* @brief

\* 监听DNS线程处理函数

\* @param

\* SOCKET upper\_dns\_socket DNS监听线程

\* @param

\* SOCKET listen\_sokcet 客户端监听线程

\* @return

\* int t\_id 线程id

\* @note

\* 持续从DNS监听报文，若收到，则修改id并通过客户端监听线程发给客户端\*/

void UpperThreadHandle(SOCKET, SOCKET, int);

/\*\*

\* @brief

\* 根据域名确定域名类型

\* @param

\* addr 域名

\* @param

\* ip

\* @return

\* ADDR\_TYPE 域名类型

\* @note

\*/

ADDR\_TYPE getAddrType(char \*, UINT32 \*);

/\*\*

\* @brief

\* 根据域名类型确定发送给客户端的字节流

\* @param

\* ori\_packet 从客户端接收到的报文

\* @param

\* old\_id 原来的id

\* @param

\* ip\_addr ip地址

\* @param

\* addr\_type 域名类型

\* @param

\* sendbuflen 发送字长

\* @return

\* char \* 发送给客户端的字节流

\* @note

\*/

char \*getDNSResult(DNSPacket \*ori\_packet, int old\_id, UINT32 ip\_addr, ADDR\_TYPE

addr\_type, int &sendbuflen);

/\*\*

\* @brief

\* 将网络二进制字节流指向的内容转化为DNSPacket

\* @param

\* char\* 源指针

\* @return

\* DNSPacket\*

\* @note

\* 函数调用了fromDNSHeader()、fromDNSQuery()和fromDNSResponse()

\* 根据DNS报文格式进行转换

\* 根据实际需要，不对所有Query和Response进行转换，只分别对第一个进行转换

\*/

DNSPacket \*unpackDNSPacket(char \*);

/\*\*

\* @brief

\* 将DNSPacket\*指向的内容转化为网络二进制字节流

\* @param

\* DNSPacket\* 源指针

\* @param

\* int& 字节流长度

\* @return

**六、特殊问题的应对方案**

1. **多客户端并发**

对于多客户端并发问题，我们采取的解决方案是：**生产者消费者模式**。对于多个客户端同时发DNS查询请求，会由监听线程进行请求的“生产”，并存放到“货架”（请求池）上，由“消费者”（DNS 处理线程）进行“消费”。生产者消费者模式可以使请求的处理更为高效，多线程并发处理请求使得请求不会被阻塞（DNS的处理时间远比收到DNS请求的处理时间要长的多）。此外，请求池作为缓冲区，还可以支持处理速度时快时慢的情况，消费者来不及处理的数据暂时存储在缓冲区中，等到消费者空闲了再进行处理。

2. **超时处理**

对于一些超时的查询报文，如果保存在请求池中而不进行处理，会造成请求池中冗余项过多而使得 能够并发处理的请求数量大幅减少。因此，针对长时间没有响应的报文，必须得通过方法将其删除。因此，我们想出的方法是，以一个时间长度（如5s）作为每一个请求的生命周期，在每一个请求池项中添加一个 time\_t startTime 作为当前请求池中的请求的起始时间。并在每一个DNS处理线程从请求池中获取请求之后添加一段代码块，功能是遍历每个请求池，如果该请求池有请求并且当前的计时器endTime与请求池中的计时器startTime的时间差大于5s，则从请求池中删去该请求。

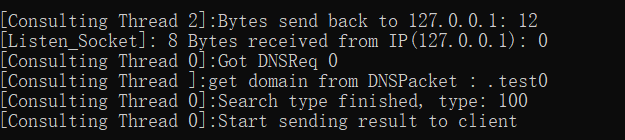
**七、测试用例及结果 ：**

启动DNS中继服务器

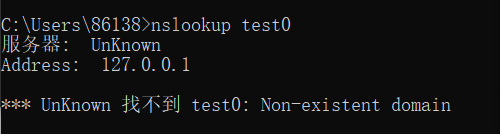
**基本功能测试**

1.不良网站拦截功能 :type==100

server

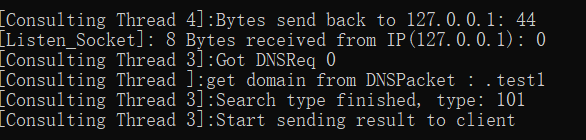


client

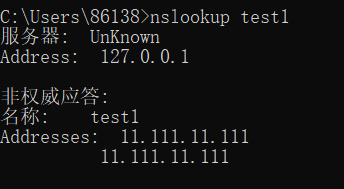


2.服务器功能：命中cache,type== 101

server：



client

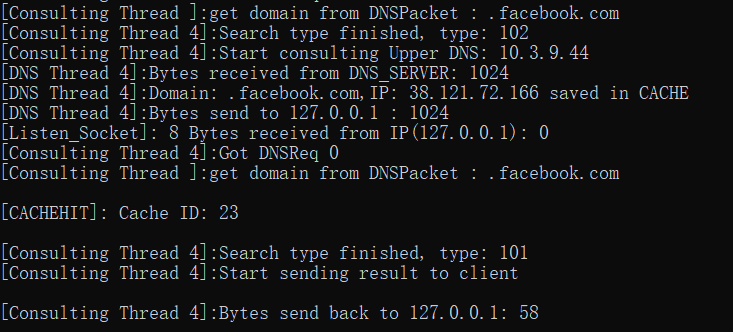


可以看到出现了提示[CACHEHIT]表示命中缓存，并且返回的域名类型 ADDRTYPE 变成了CACHED (101)。

3.中继功能

未命中缓存：type==102

server



client



第一次查询facebook.com，在host中无法找到对应域名，因此返回的域名类型 ADDRTYPE 是 ADDR\_NOT\_FOUND （102）

**八、遇见的问题 ：**

1. **程序在执行过程中经常莫名其妙出现数组越界等问题**

经过调试发现，端口53除了接收用户nslookup发送的DNS报文，还会偶尔接收来自另一个特定IP 的一些字节流，而这些字节流因为没有经过处理，所以会造成程序在进行字节流转换的时候崩溃。因此，在客户端监听线程中添加了一个判定：

// 有时会莫名其妙收到垃圾报文

if (strcmp("127.0.0.1", inet\_ntoa(clt\_addr.sin\_addr)) != 0)

continue;

只有当收到的数据来自127.0.0.1时，才视作DNS请求。

2. **在进行字节流和自定义数据结构的转换不够熟练**

在字节流的转换中，首先要理解字节流内容的形式，还有大端法小端法的差异。对于字节流，从头到尾进行转化。我们需要什么样的数据类型，就使用对于的数据类型的指针，因为对应的数据类型的指针指向的字节数和你所预期的是相同的。因此在读取id的时候，要采用ushort类型的指针（指向2 Bytes），并且不能忘记使用ntohs()进行字节序的转换。难点在于头部的Flags字段的转化。因为Flags中的字段并非对齐的，第一个字节是一个数据qr，后面连续四个字节是opcode，这样则不能用一个现有的指针将其表示出来（如果用指向4 Bytes的指针，指针会取整，无法包含所有的opcode内容）。因此编程时决定采用按位取或的方式来实现。将指针指向的内容置零，在将各个Flags中的字段向左移位到对应的位置，与0或操作，则可得到正确结果。

3. **消息ID的转换**

在向上一层的DNS服务器发送查询请求时，因为下层的DNS请求的消息id可能重复，如果不进行消息id的转换，可能会出现错误的DNS请求结果。因此，我们在向上一层DNS服务器发送DNS请求报文时，必须保证每个ID的唯一性。在设计的过程中，我们发现可以利用到请求池的位置作为新的消息ID。请求池是多个查询线程共用的，因此每个池的位置在全局中具有唯一性，因此以该请求放在请求池中的位置i作为新的消息ID即可以保证唯一性。

1. **从本地文件初始化**

在对“dnsrelay.txt”的内容进行读入，并借此对cache进行初始化时，最初只是把文件中的内容直接读入，在查找时再将字符串转化后，与将转化后的报文再进行比较，比较费时间，浪费cpu。验收时，经老师提醒，改为在一开始就将文件内容直接转化为报文的形式，这样只需要初始化一次就可以了。

**九、实验心得**

虽然我们在计算机网络课程中学习了TCP和UDP的内容，但是当我们真正的着手去做时，我们发现实际编程的难度是远远大于课程上学习理论知识的难度的。UDP是一个不可靠、无差错控制的协议，所以在传输数据包的时候会出现乱序、丢包的问题，基于这个问题，我们选择了使用多线程的方式进行开发，而生产者消费者的模式也颇为符合实际的一些设计模式，保证了通信的可靠性。此外，我们还考虑 到线程之间的资源竞争的问题，因此采用了lock\_guard来加锁。此外，本次课程设计对我最大的启发就是：一个程序的框架设计远比细节实现要重要得多。如果没有框架就开始设计细节，经常会因为考虑不周而陷入死路，极大影响了工作效率。但是框架的设计是一 件非常需要编程经验和全局观的事情，很明显现在我还不具有这样的全局观和经验。希望未来，能通过更多的项目来改善这个问题。

通过这次的实验，我们在网络字节流和讲述的报文格式之间有了更具象的认识。在抓包实验分析包裹信息的基础上，进一步在C语言中实现将接收到的字节流读出，并将要发送的包裹转化为正确的字节流发送。转换过程需要的一些赋值操作，使我们对指针的掌握和类型转换操作有所提升。虽然中继服务器的整体收发思路并不特别复杂，但仍有许多需要考虑的细节，比如实现并发处理中对于多个client的id号一致的id转换，在request中设置好原本的id和new\_id以方便处理；上层包裹可能超时，要设置计时器监视是否出现超时，及时移出请求池，并防止迟到包重复送到client。如果在架构时能够考虑到更多的功能需要，就会减少一些额外补充功能的麻烦，我们应该在这方面多加注意。

具体代码：

**Header.h**

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#include <winsock2.h>

#include <ws2tcpip.h>

#include <windows.h>

#include <stdio.h>

#include <iostream>

#include <thread>

#include <mutex>

#include <time.h>

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

using namespace std;

#define DNS\_PORT 53 //DNS serves on port 53

#define DEFAULT\_BUFLEN 1024

#define DNS\_HEADER\_LEN 12

#define MAX\_HOST\_ITEM 1200

#define MAX\_CACHED\_ITEM 200

#define MAX\_REQ 100

#define UPPER\_DNS "10.3.9.44"

#define MAX\_THREAD 5

#define MAX\_REQ\_TTL 10

typedef unsigned short ushort;

typedef unsigned int u\_int;

enum Query\_QR {

Q\_QUERY = 0, Q\_RESPONSE

};

typedef struct DnsHeader

{

ushort h\_id;

bool h\_qr;

ushort h\_opcode;

bool h\_aa;

bool h\_tc;

bool h\_rd;

bool h\_ra;

char h\_rcode;

ushort h\_QDCount;

ushort h\_ANCount;

ushort h\_NSCount;

ushort h\_ARCount;

}DNSHeader;

typedef struct DnsQuery

{

char \*q\_qname;

ushort q\_qtype;

ushort q\_qclass;

}DNSQuery;

typedef struct DnsResponse

{

char \*r\_name;

ushort r\_type;

ushort r\_class;

int r\_ttl;

ushort r\_rdlength;

char\* r\_rdata;

}DNSResponse;

typedef struct DnsPacket

{

Query\_QR p\_qr;

DnsHeader \*p\_header;

DnsQuery \*p\_qpointer[50];

DnsResponse \*p\_rpointer[50];

}DNSPacket;

typedef struct DnsRequest

{

bool processed;

int old\_id;

int new\_id;

DNSPacket\* packet;

struct sockaddr\_in client\_addr;

int client\_addr\_len;

}DNSRequest;

typedef struct RequestPool

{

bool available;

DNSRequest\* req;

time\_t startTime;

}ReqPool;

int dnsRelay();

int initDNSServer(SOCKET \*);

void CounsultThreadHandle(const char\*, SOCKET, int);

void UpperThreadHandle(SOCKET, SOCKET, int);

**DomianAnalysis.h**

#pragma once

#include "Header.h"

enum ADDR\_TYPE {

BLOCKED = 100,

CACHED,

ADDR\_NOT\_FOUND

};

typedef struct host\_item {

u\_int ip\_addr;

char\* webaddr;

ADDR\_TYPE type;

}host\_item;

typedef struct cache\_item {

u\_int ip\_addr;

char\* webaddr;

int ttl;

int occupied;

}cache\_item;

void loadHostsFromTxt();

ADDR\_TYPE getAddrType(char\*, u\_int\*);

char\* getDNSResult(DNSPacket\* ori\_packet, int old\_id, u\_int ip\_addr, ADDR\_TYPE addr\_type, int& sendbuflen);

**MessageConversion.h**

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Header.h"

using namespace std;

//生成

DNSHeader\* fromDNSHeader(char\*, char\*\*);

DNSQuery\* fromDNSQuery(char\*, char\*\*);

DNSResponse\* fromDNSResponse(char\*, char\*, char\*\*);

//解析

char\* toDNSHeader(DNSHeader\*);

char\* toDNSQuery(DNSQuery\*);

char\* toDNSResponse(DNSResponse\*);

char\* packDNSPacket(DNSPacket\*, int&);

DNSPacket\* unpackDNSPacket(char\*);

**RequestPool.h**

#pragma once

#include "Header.h"

using namespace std;

DNSRequest\* getDNSRequest();

int addDNSRequestPool(DNSRequest\*);

DNSRequest\* finishDNSRequest(int);

**RequestPool.cpp**

#include "RequestPool.h"

extern ReqPool\* request\_pool;

extern mutex id\_mutex, pool\_mutex, req\_counter\_mutex;

DNSRequest\* finishDNSRequest(int new\_id){

DNSRequest\* req;

lock\_guard<mutex> pool\_guard(pool\_mutex);

req = request\_pool[new\_id].req;

request\_pool[new\_id].available = true;

return req;

}

DNSRequest\* getDNSRequest(){

DNSRequest\* req = NULL;

if (pool\_mutex.try\_lock()){

for (int i = 0; i < MAX\_REQ; i++){

//该请求池有请求

if (!request\_pool[i].available){

//未被某个线程处理

if (request\_pool[i].req->processed == false){

req = request\_pool[i].req;

request\_pool[i].req->processed = true;

request\_pool[i].startTime = time(NULL);

break;

}

}

}

pool\_mutex.unlock();

}

return req;

}

int addDNSRequestPool(DNSRequest\* req){

lock\_guard<mutex> pool\_guard(pool\_mutex);

for (int i = 0; i < MAX\_REQ; i++){

if (request\_pool[i].available){

request\_pool[i].available = false;

req->old\_id = req->packet->p\_header->h\_id;

req->new\_id = i;

request\_pool[i].req = req;

request\_pool[i].startTime = time(NULL);

return i;

}

}

return -1;

}

**MessageConversion.cpp**

#include "MessageConversion.h"

DNSHeader\* fromDNSHeader(char\* src, char \*\*ret\_ptr) {

DNSHeader\* new\_q = (DNSHeader\*)malloc(sizeof(DNSHeader));

int loc\_pointer = 0;

ushort\* pointer = (ushort\*)src;

ushort cur\_word = htons(\*pointer);

new\_q->h\_id = (int)cur\_word;

cur\_word = htons(\*(++pointer));

new\_q->h\_qr = (bool)((cur\_word & 0x8000) >> 1);

new\_q->h\_opcode = (ushort)((cur\_word & 0x7800) >> 11);

new\_q->h\_aa = (bool)((cur\_word & 0x0400) >> 10);

new\_q->h\_tc = (bool)((cur\_word & 0x0200) >> 9);

new\_q->h\_rd = (bool)((cur\_word & 0x0100) >> 8);

new\_q->h\_ra = (bool)((cur\_word & 0x0080) >> 7);

new\_q->h\_rcode = (ushort)((cur\_word & 0x000F));

cur\_word = htons(\*(++pointer));

new\_q->h\_QDCount = cur\_word;

cur\_word = htons(\*(++pointer));

new\_q->h\_ANCount = cur\_word;

cur\_word = htons(\*(++pointer));

new\_q->h\_NSCount = cur\_word;

cur\_word = htons(\*(++pointer));

new\_q->h\_ARCount = cur\_word;

\*ret\_ptr = (char\*)(++pointer);

return new\_q;

}

DNSQuery\* fromDNSQuery(char\* src, char \*\*ret\_ptr) {

int qname\_length = 0;

DNSQuery\* new\_q = (DNSQuery\*)malloc(sizeof(DNSQuery));

while (\*(src + qname\_length) != '\0')

qname\_length++;

char\* s = (char\*)malloc(qname\_length);

strcpy(s, src);

new\_q->q\_qname = s;

src += (++qname\_length);

ushort\* tmp = (ushort\*)src;

new\_q->q\_qtype = htons(\*(tmp++));

new\_q->q\_qclass = htons(\*tmp);

\*ret\_ptr = (char\*)(++tmp);

return new\_q;

}

DNSResponse\* fromDNSResponse(char\* src, char\* head, char \*\*ret\_ptr) {

DNSResponse\* new\_r = (DNSResponse\*)malloc(sizeof(DNSResponse));

char\* s = (char\*)malloc(256 \* sizeof(char));

int qname\_length = 0;

char\* final\_name\_dst = src;

bool name\_jumped = false;

char\* name\_pointer = src;

//获取域名d

while (1){

if (\*name\_pointer == '\0'){

s[qname\_length] = '\0';

if (name\_jumped == false)

final\_name\_dst = src + qname\_length;

break;

}

if (((\*name\_pointer) & 0xc0) == 0xc0){//偏移量c00c

int new\_dst = ntohs(\*((ushort\*)name\_pointer)) & 0x3f;//取0c

final\_name\_dst = name\_pointer + 2;

new\_dst += (int)head;

name\_jumped = true;

// name\_pointer = head + new\_dst;

name\_pointer = (char\*)new\_dst;

continue;

}

if (\*name\_pointer < 20){

int tmp\_len = \*name\_pointer++;

s[qname\_length++] = tmp\_len;

for (int i = 0; i < tmp\_len; i++)

s[qname\_length++] = \*(name\_pointer++);

}

}

new\_r->r\_name = s;

src = final\_name\_dst;

ushort\* tmp = (ushort\*)src;

new\_r->r\_type = htons(\*(tmp++));

new\_r->r\_class = htons(\*(tmp++));

new\_r->r\_ttl = htonl(\*((int\*)tmp));

tmp += 2;

new\_r->r\_rdlength = htons(\*(tmp++));

src = (char\*)tmp;

s = (char\*)malloc((new\_r->r\_rdlength + 1) \* sizeof(char));

memcpy(s, src, new\_r->r\_rdlength);

s[new\_r->r\_rdlength] = '\0';

new\_r->r\_rdata = s;

\*ret\_ptr = src + new\_r->r\_rdlength;

return new\_r;

}

char\* toDNSHeader(DNSHeader\* ret\_h) {

ushort\* tmp\_s;

char\* ret\_s;

tmp\_s = (ushort\*)malloc(13 \* sizeof(char));

ret\_s = (char\*)tmp\_s;

\*(tmp\_s++) = ntohs((ushort)ret\_h->h\_id);

\*tmp\_s = 0;

ushort tags = 0;

tags |= (ret\_h->h\_qr << 15);

tags |= (ret\_h->h\_opcode << 11);

tags |= (ret\_h->h\_aa << 10);

tags |= (ret\_h->h\_tc << 9);

tags |= (ret\_h->h\_rd << 8);

tags |= (ret\_h->h\_ra << 7);

tags |= (ret\_h->h\_rcode);

\*(tmp\_s++) = ntohs(tags);

\*(tmp\_s++) = ntohs(ret\_h->h\_QDCount);

\*(tmp\_s++) = ntohs(ret\_h->h\_ANCount);

\*(tmp\_s++) = ntohs(ret\_h->h\_NSCount);

\*(tmp\_s++) = ntohs(ret\_h->h\_ARCount);

\*(char\*)tmp\_s = '\0';

return ret\_s;

}

char\* toDNSQuery(DNSQuery\* ret\_q){

char\* ret\_s, \* tmp\_c;

ushort\* tmp\_u;

int tot\_length;

tot\_length = strlen(ret\_q->q\_qname) + 6;

ret\_s = (char\*)malloc(tot\_length \* sizeof(char));

tmp\_c = ret\_s;

//Copy qname to reply message

strcpy(tmp\_c, ret\_q->q\_qname);

tmp\_c += strlen(ret\_q->q\_qname);

\*tmp\_c = '\0';

tmp\_c++;

tmp\_u = (ushort\*)tmp\_c;

\*(tmp\_u++) = ntohs(ret\_q->q\_qtype);

\*(tmp\_u++) = ntohs(ret\_q->q\_qclass);

tmp\_c = (char\*)tmp\_u;

\*tmp\_c = '\0';

return ret\_s;

}

char\* toDNSResponse(DNSResponse\* ret\_r){

char\* ret\_s, \* tmp\_c;

ushort\* tmp\_u;

int tot\_length;

tot\_length = strlen(ret\_r->r\_name) + 11 + ret\_r->r\_rdlength + 1;

//rname

ret\_s = (char\*)malloc(tot\_length \* sizeof(char));

tmp\_c = ret\_s;

strcpy(tmp\_c, ret\_r->r\_name);

tmp\_c += strlen(ret\_r->r\_name);

\*tmp\_c = '\0';

tmp\_c++;

tmp\_u = (ushort\*)tmp\_c;

//其它ushort和int

\*tmp\_u++ = ntohs(ret\_r->r\_type);

\*tmp\_u++ = ntohs(ret\_r->r\_class);

\*(int\*)tmp\_u = ntohl(ret\_r->r\_ttl);

tmp\_u += 2;

\*tmp\_u++ = ntohs(ret\_r->r\_rdlength);

tmp\_c = (char\*)tmp\_u;

memcpy(tmp\_c, ret\_r->r\_rdata, ret\_r->r\_rdlength);

return ret\_s;

}

DNSPacket\* unpackDNSPacket(char\* buf){

char\* cur\_ptr = buf, \* ret\_ptr;

DNSPacket\* dns\_packet = (DNSPacket\*)malloc(sizeof(DNSPacket));

// Read DNS Header

dns\_packet->p\_header = fromDNSHeader(cur\_ptr, &ret\_ptr);

cur\_ptr = ret\_ptr;

// Read DNS Query

for (int i = 0; i < dns\_packet->p\_header->h\_QDCount; i++){

dns\_packet->p\_qpointer[i] = fromDNSQuery(cur\_ptr, &ret\_ptr);

cur\_ptr = ret\_ptr;

}

// Read DNS Response

if (dns\_packet->p\_header->h\_ANCount > 0){

dns\_packet->p\_rpointer[0] = fromDNSResponse(cur\_ptr, buf, &ret\_ptr);

cur\_ptr = ret\_ptr;

dns\_packet->p\_header->h\_ANCount = 1;

}

else{

dns\_packet->p\_rpointer[0] = (DNSResponse\*)malloc(sizeof(DNSResponse));

dns\_packet->p\_rpointer[0]->r\_rdata = NULL;

}

dns\_packet->p\_qr = dns\_packet->p\_header->h\_qr ? Q\_RESPONSE : Q\_QUERY;

return dns\_packet;

}

char\* packDNSPacket(DNSPacket\* packet, int& len){

char\* new\_header = toDNSHeader(packet->p\_header);

char\* ret\_string = (char\*)malloc(DEFAULT\_BUFLEN);

memcpy(ret\_string, new\_header, DNS\_HEADER\_LEN);

len = DNS\_HEADER\_LEN;

if (packet->p\_header->h\_QDCount == 1){

char\* new\_query = toDNSQuery(packet->p\_qpointer[0]);

memcpy(ret\_string + len, new\_query, strlen(packet->p\_qpointer[0]->q\_qname) + 5);

len += strlen(packet->p\_qpointer[0]->q\_qname) + 5;

}

if (packet->p\_qr == Q\_RESPONSE && packet->p\_header->h\_ANCount > 0){

char\* new\_response = toDNSResponse(packet->p\_rpointer[0]);

memcpy(ret\_string + len, new\_response, strlen(packet->p\_rpointer[0]->r\_name) + 11 + packet->p\_rpointer[0]->r\_rdlength);

len += strlen(packet->p\_rpointer[0]->r\_name) + 11 + packet->p\_rpointer[0]->r\_rdlength;

}

return ret\_string;

}

**MessageConversion.cpp**

#include "DomainAnalysis.h"

#include "MessageConversion.h"

using namespace std;

extern host\_item\* hosts\_list[];

extern cache\_item\* cached\_list[];

extern int host\_counter;

void loadHostsFromTxt() {

FILE\* fp = fopen("dnsrelay.txt","r");

if (fp == NULL) {

printf("The file can't open...\n");

exit(1);

}

char ipaddr[DEFAULT\_BUFLEN];

char domain[DEFAULT\_BUFLEN];

int cnt = 0;

while (!feof(fp)) {

fgets(ipaddr, DEFAULT\_BUFLEN, fp);

for (int i = 0; i < DEFAULT\_BUFLEN; i++){

if (ipaddr[i] == ' '){

ipaddr[i] = '\0';

strcpy(domain, ipaddr + i + 1);

if (domain[strlen(domain) - 1] == '\n')

domain[strlen(domain) - 1] = '\0';

else

domain[strlen(domain)] = '\0';

break;

}

}

hosts\_list[cnt] = (host\_item\*)malloc(sizeof(host\_item));

hosts\_list[cnt]->webaddr = (char\*)malloc(DEFAULT\_BUFLEN);

inet\_pton(AF\_INET, ipaddr, &hosts\_list[cnt]->ip\_addr);//IP地址转成字符串

strcpy(hosts\_list[cnt]->webaddr, domain);

if (hosts\_list[cnt]->ip\_addr == 0)

hosts\_list[cnt]->type = BLOCKED;

else

hosts\_list[cnt]->type = CACHED;

cnt++;

}

host\_counter = cnt - 1;

printf("load %d host from dnsrelay.txt successfully\n", cnt);

fclose(fp);

}

ADDR\_TYPE getAddrType(char\* addr, u\_int\* ip) {

\*ip = 0x0;

char\* tmp\_addr = (char\*)malloc(DEFAULT\_BUFLEN);

strcpy(tmp\_addr, addr);

for (int i = 0; i < strlen(addr); i++) {

if (tmp\_addr[i] < 0x20)

tmp\_addr[i] = '.';

else if (tmp\_addr[i] >= 'A' && tmp\_addr[i] <= 'Z')

tmp\_addr[i] += 'a' - 'A';

}

printf("[Consulting Thread ]:get domain from DNSPacket : %s\n", tmp\_addr);

for (int i = 0; i <= host\_counter; i++) {

if (strstr(tmp\_addr, hosts\_list[i]->webaddr)) {

\*ip = htonl(hosts\_list[i]->ip\_addr);

if (\*ip != 0)

return CACHED;

else

return BLOCKED;

}

}

for (int i = 0; i < MAX\_CACHED\_ITEM; i++) {

if (!cached\_list[i]->occupied) continue;

if (strstr(tmp\_addr, cached\_list[i]->webaddr)) {

printf("\n[CACHEHIT]: Cache ID: %d\n\n", i);

\*ip = htonl(cached\_list[i]->ip\_addr);

cached\_list[i]->ttl = 50;

if (\*ip != 0)

return CACHED;

else

return BLOCKED;

}

else {

(cached\_list[i]->ttl)--;

if (cached\_list[i]->ttl == 0)

cached\_list[i]->occupied = false;

}

}

return ADDR\_NOT\_FOUND;

}

char\* getDNSResult(DNSPacket\* ori\_packet, int old\_id, u\_int ip\_addr, ADDR\_TYPE addr\_type, int& sendbuflen) {

DNSPacket\* ret\_packet = (DNSPacket\*)malloc(sizeof(DNSPacket));

DNSHeader\* ret\_header = (DNSHeader\*)malloc(sizeof(DNSHeader));

DNSQuery\* ret\_query = ori\_packet->p\_qpointer[0];

DNSResponse\* ret\_response = (DNSResponse\*)malloc(sizeof(DNSResponse));

ushort ret\_id;

if (addr\_type == BLOCKED){//创建新的DNS头

ret\_header->h\_id = ori\_packet->p\_header->h\_id;

ret\_header->h\_qr = 1;

ret\_header->h\_opcode = ori\_packet->p\_header->h\_opcode;

ret\_header->h\_aa = 0;

ret\_header->h\_tc = 0;

ret\_header->h\_rd = 1;

ret\_header->h\_ra = 1;

ret\_header->h\_rcode = 3;

ret\_header->h\_QDCount = 0;

ret\_header->h\_ANCount = 0;

ret\_header->h\_NSCount = 0;

ret\_header->h\_ARCount = 0;

ret\_packet->p\_header = ret\_header;

ret\_packet->p\_qpointer[0] = NULL;

ret\_packet->p\_rpointer[0] = NULL;

ret\_packet->p\_qr = Q\_RESPONSE;

}

else{

ret\_response->r\_name = ori\_packet->p\_qpointer[0]->q\_qname;

ret\_response->r\_type = 1;

ret\_response->r\_class = ori\_packet->p\_qpointer[0]->q\_qclass;

ret\_response->r\_ttl = 0x100;

ret\_response->r\_rdlength = 4;

ret\_response->r\_rdata = (char\*)malloc(sizeof(u\_int) + 1);

\*(u\_int\*)(ret\_response->r\_rdata) = htonl(ip\_addr);

ret\_header->h\_id = ori\_packet->p\_header->h\_id;

ret\_header->h\_qr = 1;

ret\_header->h\_opcode = ori\_packet->p\_header->h\_opcode;

ret\_header->h\_aa = 0;

ret\_header->h\_tc = 0;

ret\_header->h\_rd = 1;

ret\_header->h\_ra = 1;

ret\_header->h\_rcode = 0;

ret\_header->h\_QDCount = 1;

ret\_header->h\_ANCount = 1;

ret\_header->h\_NSCount = 0;

ret\_header->h\_ARCount = 0

ret\_packet->p\_header = ret\_header;

ret\_packet->p\_qpointer[0] = ret\_query;

ret\_packet->p\_rpointer[0] = ret\_response;

ret\_packet->p\_qr = Q\_RESPONSE;

}

ret\_packet->p\_header->h\_id = old\_id;

char\* sendbuf = (char\*)malloc(DEFAULT\_BUFLEN);

sendbuf = packDNSPacket(ret\_packet, sendbuflen);

return sendbuf;

}

**Main.cpp**

#include "DomainAnalysis.h"

#include "Header.h"

#include "MessageConversion.h"

#include "RequestPool.h"

host\_item\* hosts\_list[MAX\_HOST\_ITEM];

cache\_item\* cached\_list[MAX\_CACHED\_ITEM];

ReqPool\* request\_pool;

mutex id\_mutex, pool\_mutex, req\_counter\_mutex, time\_mutex, cache\_mutex;

int req\_counter = 0, host\_counter = 0;

thread\* dns\_consulting\_threads[MAX\_THREAD];

int main() {

dnsRelay();

return 0;

}

int dnsRelay() {//监听端口报文信息并加入请求池

loadHostsFromTxt();

for (int i = 0; i < MAX\_CACHED\_ITEM; i++){

cached\_list[i] = (cache\_item\*)malloc(sizeof(cache\_item));

cached\_list[i]->webaddr = (char\*)malloc(DEFAULT\_BUFLEN);

cached\_list[i]->occupied = false;

}

int i\_result = 0;

SOCKET Listen\_Socket;

i\_result = initDNSServer(&Listen\_Socket);

if (i\_result == 1) return 255;

request\_pool = (ReqPool\*)malloc(sizeof(ReqPool) \* MAX\_REQ);

if (request\_pool == NULL) {

printf("Fail to initialize request pool.\n");

exit(100);

}

for (int i = 0; i < MAX\_REQ; i++)

request\_pool[i].available = 1;

for (int i = 0; i < MAX\_THREAD; i++)

dns\_consulting\_threads[i] = new thread(CounsultThreadHandle, UPPER\_DNS, Listen\_Socket, i);

printf("Initialize completed.\n");

while (1) {

char\* recvbuf = (char\*)malloc(sizeof(char) \* DEFAULT\_BUFLEN);

struct sockaddr\_in clt\_addr;

int clt\_addr\_len = sizeof(clt\_addr);

if (sizeof(recvbuf) <= 0) {

printf("Fail to allocate buff.\n");

exit(100);

}

memset(recvbuf, 0, sizeof(recvbuf));

ZeroMemory(&clt\_addr, clt\_addr\_len);

//Receive DNS Requests.

if (recvfrom(Listen\_Socket, recvbuf, DEFAULT\_BUFLEN, 0, (struct sockaddr\*)&clt\_addr, &clt\_addr\_len) == SOCKET\_ERROR)

printf("[Listen\_Socket]: recvfrom client error with: %d\n\n", WSAGetLastError());//一直等待数据的到来

else {

if (strcmp("127.0.0.1", inet\_ntoa(clt\_addr.sin\_addr)) != 0){

printf("Receive bad message!\n");

continue;

}

printf("[Listen\_Socket]: %d Bytes received from IP(%s): %d\n",sizeof(recvbuf),inet\_ntoa(clt\_addr.sin\_addr), i\_result);

DNSRequest\* new\_req = (DNSRequest\*)malloc(sizeof(DNSRequest));

new\_req->packet = unpackDNSPacket(recvbuf);

new\_req->processed = 0;

new\_req->client\_addr = clt\_addr;

new\_req->client\_addr\_len = clt\_addr\_len;

if (addDNSRequestPool(new\_req) == -1) {

printf("[Listen\_Socket]:Now on traffic,wait patiently.\n");

Sleep(1000);

}

}

}

}

int initDNSServer(SOCKET\* ret\_socket) {//初始化socket

int i\_result = 0;

WSADATA wsaData;

i\_result = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (i\_result != 0)

return 255;

i\_result = 0;

SOCKET ListenSocket = INVALID\_SOCKET;

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

if (ListenSocket == INVALID\_SOCKET) {

WSACleanup();

return 1;

}

struct sockaddr\_in hints;

hints.sin\_family = AF\_INET;

hints.sin\_addr.s\_addr = INADDR\_ANY;

hints.sin\_port = htons(DNS\_PORT);

i\_result = ::bind(ListenSocket, (struct sockaddr\*)&hints, sizeof(hints));//绑定socket和端口

if (i\_result == SOCKET\_ERROR) {

WSACleanup();

return 1;

}

\*ret\_socket = ListenSocket;

return 0;

}

void CounsultThreadHandle(const char\* upper\_DNS\_addr, SOCKET listen\_socket, int t\_id) {

printf("[Consulting Thread %d]: Created.\n", t\_id);

char\* sendbuf = (char\*)malloc(DEFAULT\_BUFLEN);

char\* dnsbuf = (char\*)malloc(DEFAULT\_BUFLEN);

int iResult = 0;

//上层DNS服务器的地址

struct sockaddr\_in servaddr;

ZeroMemory(&servaddr, sizeof(servaddr));

servaddr.sin\_family = AF\_INET;

servaddr.sin\_port = htons(DNS\_PORT);

inet\_pton(AF\_INET, upper\_DNS\_addr, &servaddr.sin\_addr);

struct sockaddr\_in myaddr;

ZeroMemory(&myaddr, sizeof(myaddr));

myaddr.sin\_family = AF\_INET;

myaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

myaddr.sin\_port = htons(10000 + t\_id);

// 初始化上层DNS线程

SOCKET upper\_dns\_socket = socket(AF\_INET, SOCK\_DGRAM, 0);

::bind(upper\_dns\_socket, (struct sockaddr\*)&myaddr, sizeof(myaddr));

thread return\_thread = thread(UpperThreadHandle, upper\_dns\_socket, listen\_socket, t\_id);

while (1) {

DNSRequest\* req = NULL;

while (req == NULL) {

Sleep(10);

req = getDNSRequest();

time\_t endTime = time(NULL);

if (time\_mutex.try\_lock()) {

for (int i = 0; i < MAX\_REQ; i++) {

if (!request\_pool[i].available && request\_pool[i].req->processed){

if (difftime(endTime, request\_pool[i].startTime) > MAX\_REQ\_TTL) {

DNSRequest\* req = request\_pool[i].req;

DNSPacket\* recv\_packet = req->packet;

finishDNSRequest(req->new\_id);

}

break;

}

}

time\_mutex.unlock();

}

}

printf("[Consulting Thread %d]:Got DNSReq %d\n", t\_id, req->new\_id);

DNSPacket\* recv\_packet = req->packet;

u\_int ip\_addr = 0;

ADDR\_TYPE addr\_type = getAddrType(recv\_packet->p\_qpointer[0]->q\_qname, &ip\_addr);

printf("[Consulting Thread %d]:Search type finished, type: %d\n", t\_id, addr\_type);

switch (addr\_type) {

case BLOCKED:

case CACHED:

{

int sendbuflen;

sendbuf = getDNSResult(recv\_packet, req->old\_id, ip\_addr, addr\_type, sendbuflen);

printf("[Consulting Thread %d]:Start sending result to client\n\n", t\_id);

iResult = sendto(listen\_socket, sendbuf, sendbuflen, 0, (struct sockaddr\*)&(req->client\_addr), req->client\_addr\_len);

if (iResult == SOCKET\_ERROR)

printf("[Consulting Thread %d]:sendto() failed with error code : %d\n", t\_id, WSAGetLastError());

else

printf("[Consulting Thread %d]:Bytes send back to 127.0.0.1: %d\n", t\_id, iResult);

finishDNSRequest(req->new\_id);

}

break;

case ADDR\_NOT\_FOUND:

{

int packet\_length;

ushort p\_id = req->new\_id;

recv\_packet->p\_header->h\_id = p\_id;

char\* send\_string = packDNSPacket(recv\_packet, packet\_length);

printf("[Consulting Thread %d]:Start consulting Upper DNS: %s\n", t\_id, upper\_DNS\_addr);

if (sendto(upper\_dns\_socket, send\_string, packet\_length, 0, (struct sockaddr\*)&servaddr, sizeof(servaddr)) == SOCKET\_ERROR)

printf("[Consulting Thread %d]:sendto() failed with error code : %d\n", t\_id, WSAGetLastError());

}

break;

}

}

}

void UpperThreadHandle(SOCKET upper\_dns\_socket, SOCKET listen\_packet, int t\_id) {

int i\_result = 0, sleeptime = 20;

struct sockaddr\_in servaddr;

int servaddrlen = sizeof(servaddr);

char\* dnsbuf = (char\*)malloc(DEFAULT\_BUFLEN);

while (true) {

i\_result = recvfrom(upper\_dns\_socket, dnsbuf, DEFAULT\_BUFLEN, 0, (struct sockaddr\*)&servaddr, &servaddrlen);

if (i\_result == SOCKET\_ERROR) {

if (WSAGetLastError() == WSAEWOULDBLOCK) {//接受或发送缓存已满

Sleep(sleeptime);

continue;

}

else {

printf("[DNS Thread %d]:! recvfrom\_server() failed with error code : %d\n", t\_id, WSAGetLastError());

break;

}

}

else {

printf("[DNS Thread %d]:Bytes received from DNS\_SERVER: %d\n", t\_id, DEFAULT\_BUFLEN);

int p\_id = ntohs(\*(ushort\*)dnsbuf);

if (request\_pool[p\_id].available == true) continue;

DNSPacket\* return\_pack = unpackDNSPacket(dnsbuf);

//存入缓存

if (return\_pack->p\_rpointer[0]->r\_rdata != NULL && return\_pack->p\_qpointer[0]->q\_qtype == 1){

UINT32\* rdata\_pointer = (UINT32\*)return\_pack->p\_rpointer[0]->r\_rdata;

UINT32 ip\_uint = (\*rdata\_pointer);

in\_addr inaddr;

inaddr.S\_un.S\_addr = ip\_uint;

char\* ipaddr = inet\_ntoa(inaddr);

char\* webaddr = (char\*)malloc(DEFAULT\_BUFLEN);

strcpy(webaddr, return\_pack->p\_qpointer[0]->q\_qname);

// 将不可见的字符转化为.方便对比

for (int i = 0; i < strlen(webaddr); i++){

if (webaddr[i] < 0x20)

webaddr[i] = '.';

else if (webaddr[i] >= 'A' && webaddr[i] <= 'Z')

webaddr[i] -= 'A' - 'a';

}

printf("[DNS Thread %d]:Domain: %s,IP: %s saved in CACHE\n", t\_id, webaddr, ipaddr);

cache\_mutex.lock();

for (int i = 0; i < MAX\_CACHED\_ITEM; i++){

if (cached\_list[i]->occupied) continue;

cached\_list[i]->occupied = true;

inet\_pton(AF\_INET, ipaddr, &cached\_list[i]->ip\_addr);

strcpy(cached\_list[i]->webaddr, webaddr + 1);

break;

}

cache\_mutex.unlock();

}

DNSRequest\* req = finishDNSRequest(p\_id);

\*(ushort\*)dnsbuf = htons(req->old\_id);

i\_result = sendto(listen\_packet, dnsbuf, DEFAULT\_BUFLEN, 0, (struct sockaddr\*)&(req->client\_addr), req->client\_addr\_len);

char\* client\_ipaddr = inet\_ntoa(req->client\_addr.sin\_addr);

if (i\_result == SOCKET\_ERROR)

printf("[DNS Thread %d]:sendto() failed with error code : %d\n", t\_id, WSAGetLastError());

else

printf("[DNS Thread %d]:Bytes send to %s : %d\n", t\_id, client\_ipaddr, i\_result);

}

}

}