**北京邮电大学课程设计报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **课程设计**  **名称** | 计算机网络课程设计 | | **学 院** | 计算机 | **指导教师** | 吴起凡 |
| **班 级** | **班内序号** | **学 号** | | **学生姓名** | **成绩** | |
|  |  |  | |  |  | |
|  |  |  | |  |  | |
| **课**  **程**  **设**  **计**  **内**  **容** | 主要内容：设计一个DNS服务器程序，读入“域名-IP地址”对照表（一个文件），当客户端查询域名对应的IP地址时，用域名检索该对照表，得到三种检索结果：检索结果为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）；检索结果为普通IP地址，则向客户返回这个地址（服务器功能）；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）。  教学目的：使学生了解DNS服务器的工作原理和运作模式，通过实践帮助学生更好地掌握计算机网络的理论知识，包括传输层的协议、域名转换的流程等等  团队分工：A：作为小组组长，负责整体框架的实现，并将整体思路用代码实现，并负责最终报告的撰写；B：负责后期的程序的优化以及调试工作，以及修复了程序中存在的多个问题。 | | | | | |
| **学生**  **课程设计**  **报告**  **（附页）** | **请见计算机网络课程设计报告** | | | | | |
| **课**  **程**  **设**  **计**  **成**  **绩**  **评**  **定** | **评语:**  **成绩:**  **指导教师签名：**  **年 月 日** | | | | | |

注：评语要体现每个学生的工作情况，可以加页。

目录

[DNS中继服务器实验报告 3](#_Toc50465191)

[系统功能设计 3](#_Toc50465192)

[模块设计 3](#_Toc50465193)

[报文转换模块 (MessageConversion.h) 3](#_Toc50465194)

[域名处理模块 （DomainAnalysis.h） 3](#_Toc50465195)

[监听线程模块 （MainHeader.h） 3](#_Toc50465196)

[请求池模块 （RequestPoo.h） 3](#_Toc50465197)

[主程序模块 （MainHeader.h） 3](#_Toc50465198)

[软件流程图 4](#_Toc50465199)

[运行逻辑 5](#_Toc50465200)

[数据结构 6](#_Toc50465201)

[关键全局变量和函数 8](#_Toc50465202)

[特殊问题的应对方案 11](#_Toc50465203)

[测试用例及结果 12](#_Toc50465204)

[遇见的问题 15](#_Toc50465205)

[实验心得 18](#_Toc50465206)

# DNS中继服务器实验报告

## 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：

* 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）
* 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）
* 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

## 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.h)

* 将从Socket收到的网络字节流解析为自定义的DNS数据包格式
* 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.h）

* 从文件中获得域名和对应的ip地址
* 根据DNS报文中得到的域名找到ip地址和判定域名的类型
* 根据域名类型发送返回报文

### 监听线程模块 （MainHeader.h）

* 初始化、建立、关闭Socket
* 接收、发送Socket字节流
* 监听线程、DNS咨询线程的线程函数

### 请求池模块 （RequestPoo.h）

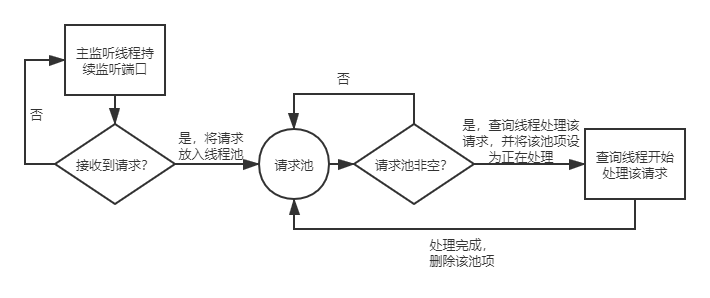
* 请求池的添加、获取、删除请求

### 主程序模块 （MainHeader.h）

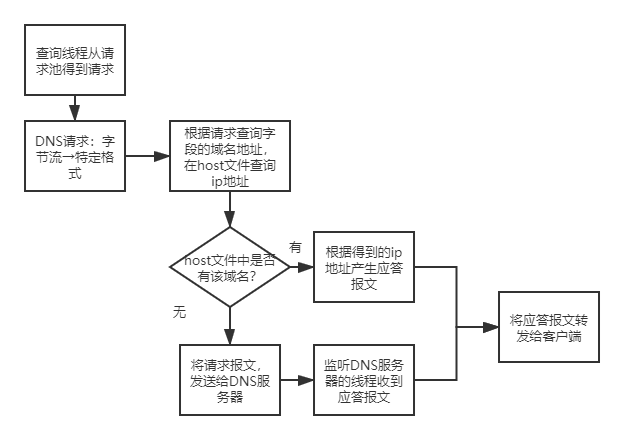
* 综合各个模块，完成所有功能

## 软件流程图

**请求池模块流程图**



**查询线程流程图**



## 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。
* **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，因此我限定了发送ip只能是固定的某几个ip），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。
* int dnsRelay()  
  {  
   // Initialize, load history data  
   // Initialize, create listen socket  
   // Initialize DNSRequest Pool  
   // 初始化部分省略  
   do  
   {  
   char \*recvbuf = (char\*)malloc(DEFAULT\_BUFLEN \* sizeof(char));  
    
   struct sockaddr\_in clt\_addr;  
   int clt\_addr\_len = sizeof(clt\_addr);  
   memset(recvbuf, 0, sizeof(recvbuf));  
   ZeroMemory(&clt\_addr, sizeof(clt\_addr));  
    
   // 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", WSAGetLastError());  
   }  
   else {  
   printf("[Listen\_Socket]: Bytes received from INADDR\_ANY: %d\n", iResult);  
   DNSRequest \*new\_req = (DNSRequest\*)malloc(sizeof(DNSRequest));  
   new\_req->packet = unpackDNSPacket(recvbuf);  
   new\_req->processed = false;  
   new\_req->client\_addr = clt\_addr;  
   new\_req->client\_addr\_len = clt\_addr\_len;  
   if (addDNSRequestPool(new\_req) == -1)  
   {  
   printf("[Listen\_Socket]: Too many requests. Ignore current one.");  
   Sleep(1000);  
   }  
   }  
    
   } while (true);  
  }
* **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
* **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。

## 数据结构

根据DNS的头部的内容设计的struct DNSHeader

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;

DNSQuery、DNSReponse与此类似，不再展示

struct DNSPacket包含了一个头部，50个Query和50个Response部分。还包含了枚举变量p\_qr，用来表示改报文是查询报文还是应答报文

enum Query\_QR { Q\_QUERY = 0, Q\_RESPONSE = 1 };  
  
  
typedef struct DnsPacket  
{  
 Query\_QR p\_qr;  
 DNSHeader \*p\_header;  
 DNSQuery \*p\_qpointer[50];  
 DNSResponse \*p\_rpointer[50];  
}DNSPacket;

struct DNSRequest自定义的DNS请求结构，processed表示当前请求是否正在被查询线程处理，old\_id和new\_id分别表示用户发来的报文id和中继服务器发给DNS服务器的报文id。client\_addr和client\_addr\_len储存客户端的地址信息，以供后续将应答报文发给客户使用。

typedef struct DnsRequest  
{  
 bool processed;  
 int old\_id;  
 int new\_id;  
 DNSPacket \*packet;  
 struct sockaddr\_in client\_addr;  
 int client\_addr\_len;  
}DNSRequest;

ReqPool作为请求池的表项，存储一个请求信息和一个判断该表项是否被使用的变量

typedef struct RequestPOOL  
{  
 bool available;  
 DNSRequest \*req;  
}ReqPool;

host\_item用来存储dnsrelay.txt中的host信息

cache\_item用来存储dnsrelay.txt缺少的host信息（用户查询过的，并且已经收到了DNS服务器的应答报文的host信息）

enum ADDR\_TYPE { BLOCKED = 100, CACHED, ADDR\_NOT\_FOUND };  
  
/\*\*  
 \* host表项  
 \* ip\_addr : ip地址  
 \* webaddr : 域名  
 \* type :  
 \*/  
typedef struct host\_item  
{  
 UINT32 ip\_addr;  
 char\* webaddr;  
 ADDR\_TYPE type;  
}host\_item;  
  
/\*\*  
 \* cache表项  
 \* ip\_addr : ip地址  
 \* webaddr : 域名  
 \* ttl : 该表项的生命周期  
 \* occupied: 该表现是否被使用  
 \*/  
typedef struct cache\_item  
{  
 UINT32 ip\_addr;  
 char\* webaddr;  
 int ttl;  
 int occupied;  
}cache\_item;

网络字节流则用char\*存储

## 关键全局变量和函数

**全局变量：**

#define DNS\_PORT 53 //DNS serves on port 53  
#define DEFAULT\_BUFLEN 1024  
#define DNS\_HEADER\_LEN 12  
#define MAX\_HOST\_ITEM 1010  
#define MAX\_REQ 1000  
#define UPPER\_DNS "10.3.9.5"  
#define HOST\_FILE\_LOC "dnsrelay.txt"  
#define MAX\_THREAD 5

#define MAX\_REQ\_TTL 10

// 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  
std::mutex id\_mutex, pool\_mutex, req\_counter\_mutex;  
// 计数器  
int req\_counter = 0, host\_counter = 0;  
// 查询线程池  
std::thread \*dns\_consulting\_threads[MAX\_THREAD];

**关键函数：**

/\*\*  
 \* @brief  
 \* 询问线程处理函数  
 \* @param  
 \* const char\* upper\_DNS\_addr DNS服务器地址  
 \* @param  
 \* SOCKET Listen\_Sokcet 监听线程  
 \* @return  
 \* int t\_id 线程id  
 \* @note  
 \* 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  
 \* DNSPacket\*  
 \* @note  
 \* unpackDNSPacket的逆过程  
 \*/  
char \*packDNSPacket(DNSPacket \*, int&);

## 特殊问题的应对方案

1. **多客户端并发**

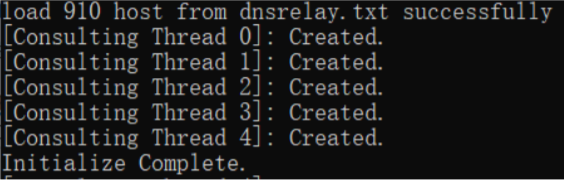
* 对于多客户端并发问题，我们采取的解决方案是：**生产者消费者模式**。对于多个客户端同时发送DNS查询请求，会由监听线程进行请求的“生产”，并存放到“货架”（请求池）上，由“消费者”（DNS处理线程）进行“消费”。生产者消费者模式可以使请求的处理更为高效，多线程并发处理请求使得请求不会被阻塞（DNS的处理时间远比收到DNS请求的处理时间要长的多）。此外，请求池作为缓冲区，还可以支持处理速度时快时慢的情况，消费者来不及处理的数据暂时存储在缓冲区中，等到消费者空闲了再进行处理。

1. **超时处理**

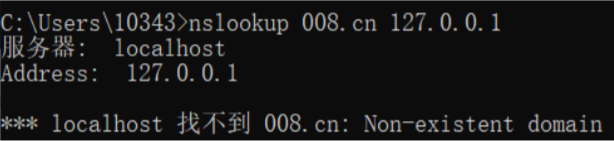
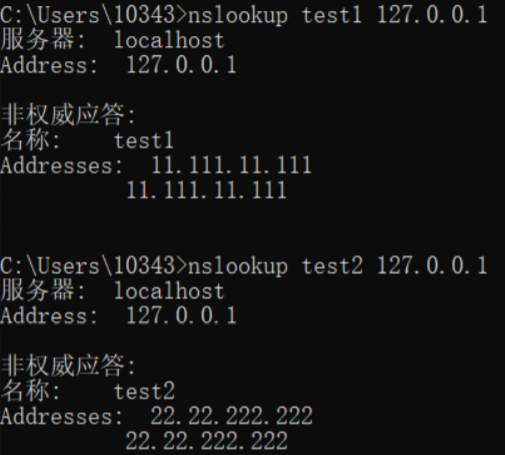
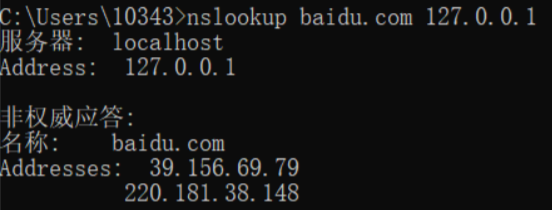
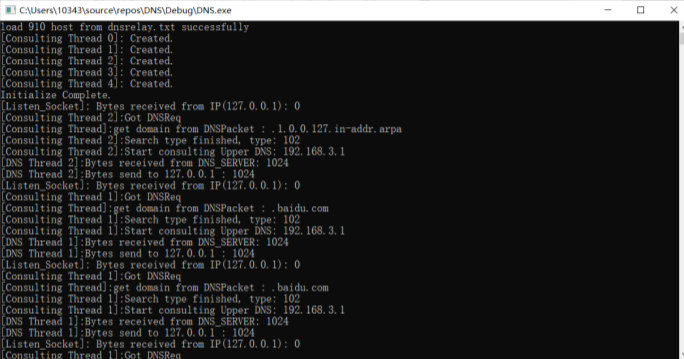
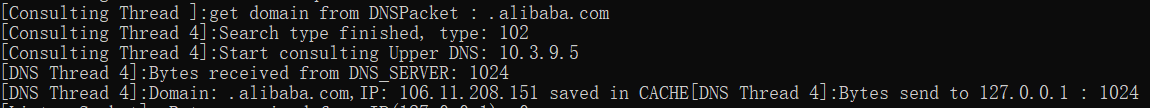
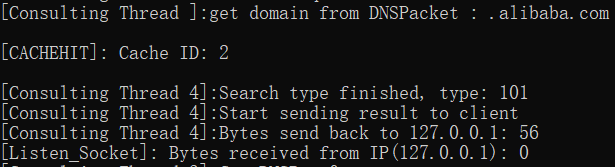
* 对于一些超时的查询报文，如果保存在请求池中而不进行处理，会造成请求池中冗余项过多而使得能够并发处理的请求数量大幅减少。因此，针对长时间没有响应的报文，必须得通过方法将其删除。因此，我们想出的方法是，以一个时间长度（如5s）作为每一个请求的生命周期，在每一个请求池项中添加一个time\_t startTime作为当前请求池中的请求的起始时间。并在每一个DNS处理线程从请求池中获取请求之后添加一段代码块，功能是遍历每个请求池，如果该请求池有请求并且当前的计时器endTime与请求池中的计时器startTime的时间差大于5s，则从请求池中删去该请求。
* while (req == NULL)  
  {  
   Sleep(20);  
   req = getDNSRequest();  
   time\_t endTime = time(NULL);  
    
   if (time\_mutex.try\_lock())  
   {  
   for (int i = 0; i < MAX\_REQ; i++)  
   {  
   //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 == true)  
   {  
   if (difftime(endTime, request\_pool[i].startTime) > 5)  
   {  
   DNSRequest\* req = request\_pool[i].req;  
   DNSPacket\* recv\_packet = req->packet;  
    
   finishDNSRequest(req->new\_id);  
   }  
   break;  
   }  
   }  
   }  
   time\_mutex.unlock();  
   }  
  }

## 测试用例及结果

**启动DNS中继服务器**



1. 基本功能测试

* 不良网站拦截功能
* 
* 服务器功能
* 
* 中继功能
* 
* 此外，根据域名的特性（不区分大小写），针对含有大小写的域名统一进行转化为小写进行分析：
* 
* 服务器情况：
* 
* 未命中缓存
* 
* 第一次查询alibaba.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）
* 命中缓存
* 
* 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)。

## 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，我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。

1. **在进行字节流和自定义数据结构的转换不够熟练**

* 以函数char \*toDNSHeader(DNSHeader \*ret\_h)为例
* 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;  
  }
* 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**）。因此我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。

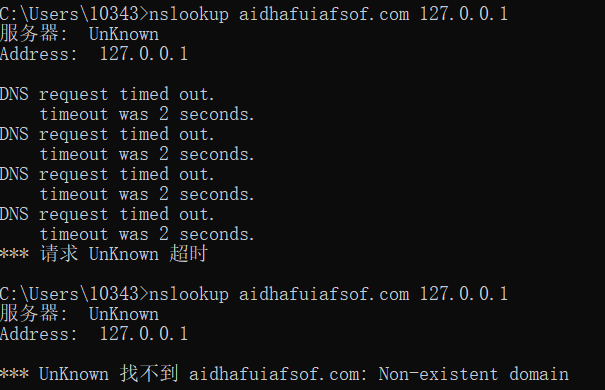
1. **项目头文件的重复**

* 这本来不是一个计网方面的问题，但是也对我们的项目进度产生了一定的阻碍，因此我在这里总结一下这个问题。
* 我在MessageConversion.h中包含了MainHeader.h，而在MainHeader.h中又包含了MessageConversion.h的内容，因此产生报错：xxx未定义。
* 实际上这种问题很好解决，一般通过前置声明即可解决。但是前置声明一般用于class的声明，而struct不能通过前置声明来解决。而产生未定义的主要的来源基本上都是一些struct在另一些头文件的函数里面作为了参数而导致的。因此针对这次课设，我把所有定义的结构体都放在了MainHeader.h中，这样就避免了重复定义。

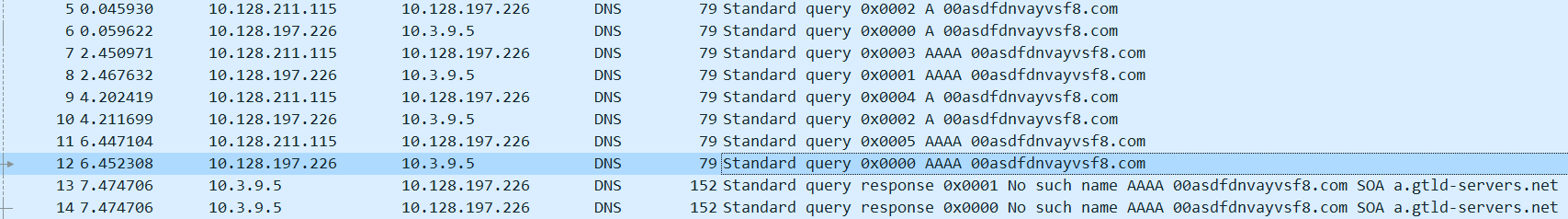
1. **消息ID的转换**

* 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. **对缓存的不理解**

* 在我们的测试过程中，我们发现假如连续两次输入一个相同的不存在的域名，会得到不一样的结果。
* 
* 起初我们以为是程序中的问题，并尝试寻找程序中的错误，但是一直没有找到。之后我们测试了老师给的样例程序，发现结果与我们的结果一致。通过对DNS服务器的原理的复习，我们发现，第一次请求一个陌生不存在的域名，因为查不到而超时；而第二次查询同一个域名时，因为第一次查询已经将该域名缓存到了本地DNS服务器的缓存中，因此直接会返回Non-existent domain。此外只要该缓存仍在生命周期之内，则会一直返回相同的结果。

1. **ID的重复和最大等待响应时间的设置**

* 在测试的过程中，我们遇到了这样的问题，当第一次发送一个不存在的域名（不是屏蔽域名），客户端连续发送了四次查询报文给服务器，并由服务器转发。以下是抓包发现问题的截图：
* 
* 由截图中可见，连续发送四次的查询报文的ID分别是0x2，0x3，0x4，0x5，而由服务器转发给上层DNS服务器的查询报文的ID分别是0x0，0x1，0x2，0x0。可见第一个报文和最后一个报文的ID发生了重复，而再发送最后一个报文之后，收到了原本0x0（0x2转化的）的应答报文，可是此时0x0对应的是原先的0x5报文，因此服务器将应答报文转发给了0x5，因此产生了BUG。
* 一开始我是怀疑自己的请求池的函数出现了问题，但是经过检查之后发现逻辑上并没有错误。于是我开始关注发送的时间，并注意到第一个报文的发送时间是0.04s，而第四个报文的发送时间是6.44s，而中间并没有收到第一个报文对应的应答报文。因此，根据程序的最大等待响应时间的设定(5s)，第一个请求报文应该已经被在请求池中被删除了。因此我修改了最大等待响应时间为10s，并再进行测试，类似情况没有再出现了。

## 实验心得

**史嘉程**：

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。我希望未来我能通过更多的项目来改善这个问题。

**李浩天：**

通过这次的实验，我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，我应该在这方面多加注意。