计算机网络课程设计——“DNS 中继服务器”的实现

张梓良 杨晨
2021212484 2021212171
计算机科学与技术
2023.6

1 概述

1.1 题目概述

设计一个 DNS 服务器程序，读入“IP 地址-域名”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：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

1.2 开发环境

- WiresharkVersion 4.0.6
- Windows_NT x64 10.0.22621
- Visual Studio Code 1.79.2
- cStandard: c17

2 DNS 理论基础

2.1 DNS 简介

域名系统（DNS）是一种用于 TCP/IP 应用程序的分布式数据库，它提供主机名字和 IP 地址之间的转换及有关电子邮件的选路信息。这里提到的分布式是指在 Internet 上的单个站点不能拥有所有的信息。每个站点（如大学中的系、校园、公司或公司中的部门）保留它自己的信息数据库，并运行一个服务器程序供 Internet 上的其他系统（客户程序）查询。DNS 提供了允许服务器和客户程序相互通信的协议。

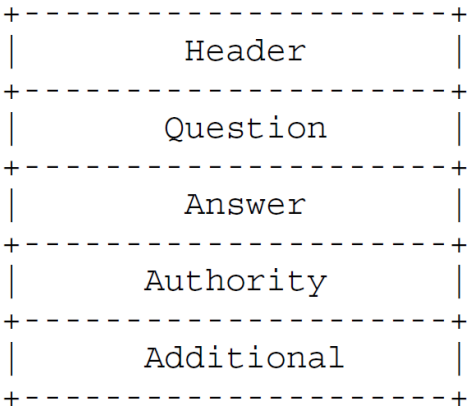
2.2 DNS 名称空间与命名语法

DNS 中使用的所有的名称集合构成了 DNS 名称空间。当前的 DNS 名称空间是一棵域名树，位于顶部的树根未命名。树的最高层是所谓的顶级域名（TLD）。

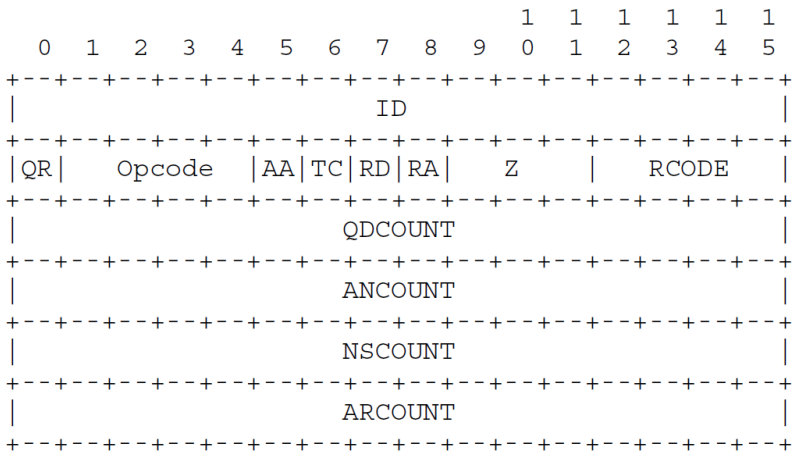
DNS 名称树中 TLD 下面的名称进一步划分成组，称为子域名，例如中国的大部分教育站点使用后缀.edu.cn。一个域名包含一系列的由点分开的标签。名称代表名称层级中的一个位置，句点是层次结构分隔符，并且按名称中自右至左的顺序沿树下降。每个标签最多可到 63 个字符长。

2.3 DNS 报文格式

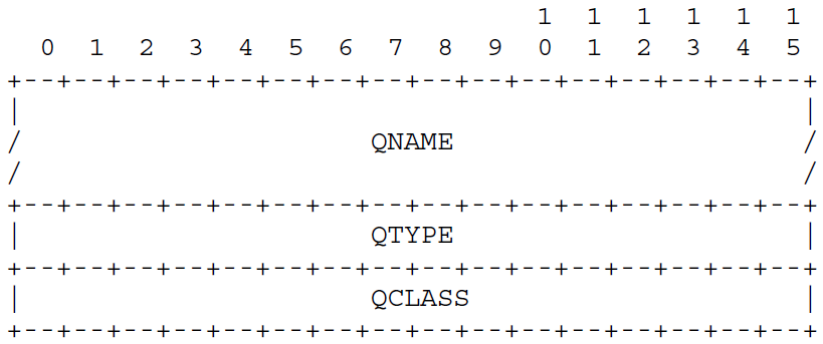
DNS 报文由 12 字节长的首部和 4 个长度可变的字段组成:



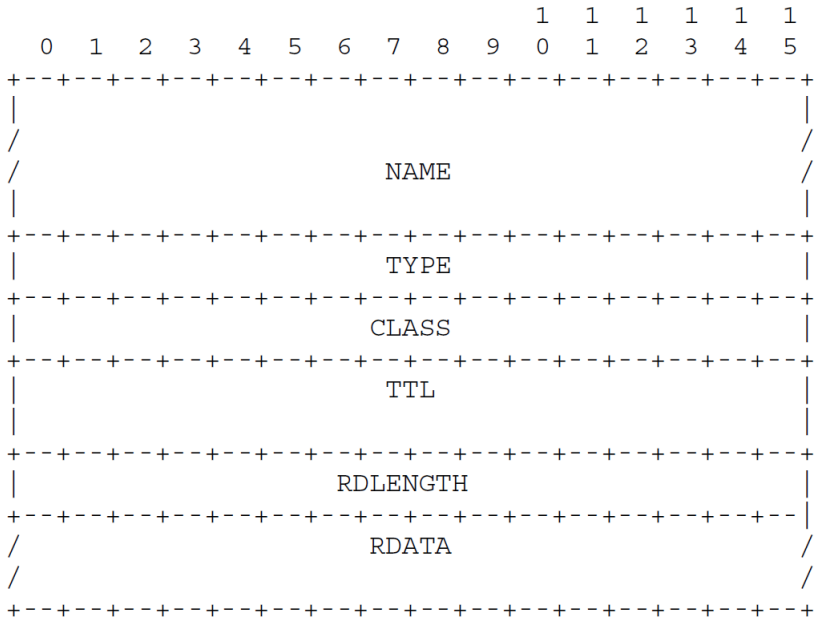
Header 字段:



Question 字段:



后三段格式相同，每段都是由 0-n 个资源记录(Resource Record)构成，资源记录格式如下：

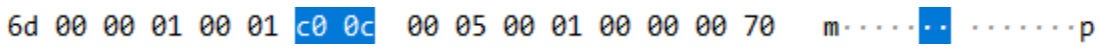


其中 QNAME 和 NAME 均为名称字段，每个名称由一系列的标签组成。标签类型有两种：数据标签和压缩标签。数据标签包含构成一个标签的字符；压缩标签充当指向其他标签的指针。当相同字符串的多个副本在多个标签中出现时，压缩标签有助于节省 DNS 信息的空间。

数据标签：每个数据标签以 1 字节的计数开始，该计数指定了紧随其后的字节数目，名称以值为 0 的字节结束。对于数据标签来说，每个标签的长度字节的值必须在 0 到 63 之间，以确保计数字节的高两位一定为 00。通过在 wireshark 抓包得到域名 config.teams.microsoft.com 的编码如下所示：

```
00 05 00 00 00 00 06 63  6f 6e 66 69 67 05 74 65   ·····c onfig·te
61 6d 73 09 6d 69 63 72  6f 73 6f 66 74 03 63 6f   ams·micr osoft·co
6d 00 00 01 00 01 c0 0c  00 05 00 01 00 00 00 70   m····· ·····p
```

压缩标签：每个压缩标签以 2 个字节的指针指向名称，指针高两位一定为 11，后 14 位为名称偏移量。通过在 wireshark 抓包得到一个压缩标签 c0 0c，计算出偏移量为 12，即名称的首字节在 DNS 报文第 13 字节处。



3 功能设计

3.1 报文解析功能

服务器收到的来自客户端的 DNS 报文为字节流格式，需要从字节流中提取出相关字段的值并存储到自定义的 DNS 报文结构体中完成对 DNS 字节流报文的解析，以便于系统查看相应字段的值，从而判断收到的报文的类型和查询内容，做出相应回应。

3.2 报文构建功能

服务器对客户端的查询做出响应时，需要构建响应报文。通过修改查询报文结构体相应部分的值，并添加上响应部分的内容得到新的响应报文的结构体，再将该自定义的响应报文结构体转换成字节流的格式发送回客户端。

3.3 不良网站拦截功能

采用分布式防火墙策略，将所有不良网站的 IP 地址重定向到本地，并将其归为无效 IP(如 0.0.0.0)，实现不良网站的拦截。

3.4 服务器功能

作为中继服务器，实现中继转发和缓存功能。接收来自客户端的 DNS 查询请求，首先在本地缓存和字典树中查找，若没有命中则向远程 DNS 服务器查询，并将结果返回给客户端。

3.5 中继功能（ID 转换）

为隐藏内网结构，实现客户端查询 ID 和内网设备 ID 之间的转换。客户端查询时使用客户端 ID，中继服务器内部使用内网 ID 进行查询。

3.6 Cache 缓存功能

维护最近 DNS 查询记录及对应的结果缓存。在接收到新的 DNS 查询时，首先检查缓存。如果命中缓存，将直接返回缓存结果，否则将查询转发到远程 DNS 服务器，并将结果缓存以供未来使用。缓存使用 LRU 置换策略管理缓存条目。

3.7 调试信息输出功能

程序在建立连接的关键步骤输出的调试信息如下：

1. 在初始化 Winsock 和创建 UDP socket 时，如果失败会输出错误日志。
2. 当成功绑定监听 socket 到指定端口时，会输出内容提示服务器已启动。
3. 当接收到客户端 DNS 请求时，会输出客户端 IP 地址、端口和请求包长度信息。
4. 如果在接收客户端请求时发生错误，也会输出相应的错误日志。

程序在中继服务器内查询和转发的关键步骤输出的调试信息如下：

1. 在中继服务器查找成功
    a) 在缓存表查找成功
    b) 在本地字典树查找成功
2. 在本地表和缓存表都未查找到，需要访问远程 DNS 服务器
3. 转发成功时，会输出内容提示中继服务器发送成功

有关报文的调试信息分为两类：

1. 以 16 进制的格式输出纯字节流的报文，便于通过对比查询报文和响应报文相应字段，判断发送的响应报文是否正确。
2. 输出自定义的 DNS 报文结构体的内容，便于观察 DNS 报文的各字段的具体值。

4 模块划分

4.1 dns_msg 模块

该模块定义了 DNS 报文中的常量值和报文结构。

1. 常量的定义：定义了 UDP 数据报的最大长度、QR 字段的值、OPCODE 字段的值、RCODE 字段的值、TYPE 字段的值和 CLASS 字段的值。

```
1.  #define UDP_MAX 512
2.
3.  // QR 字段的值定义
```

```
#define HEADER_QR_QUERY 0
#define HEADER_QR_ANSWER 1

// OPCODE 字段的值定义
#define HEADER_OPCODE_QUERY 0
#define HEADER_OPCODE_IQUERY 1
#define HEADER_OPCODE_STATUS 2

// RCODE 字段的值定义
#define HEADER_RCODE_NO_ERROR 0
#define HEADER_RCODE_NAME_ERROR 3

// TYPE 字段的值定义
#define TYPE_A 1
#define TYPE_NS 2
#define TYPE_CNAME 5
#define TYPE_SOA 6
#define TYPE_PTR 12
#define TYPE_HINFO 13
#define TYPE_MINFO 14
#define TYPE_MX 15
#define TYPE_TXT 16
#define TYPE_AAAA 28

// CLASS 字段的值定义
#define CLASS_IN 1
#define CLASS_NOT 254
#define CLASS_ALL 255
```

---

2. 报文结构的定义： Dns_Header 结构体表示 DNS 消息头，包含了消息头中各个字段的值。Dns_Question 结构体表示 DNS 消息中的问题部分，包含了查询的域名、查询类型和查询类。Dns_RR 结构体表示 DNS 消息中的资源记录部分，包含了资源记录的域名、类型、类、TTL 和数据。Dns_Msg 结构体，它表示整个 DNS 消息。它包含了一个 Dns_Header 结构体、一个 Dns_Question 结构体和一个 Dns_RR 结构体。

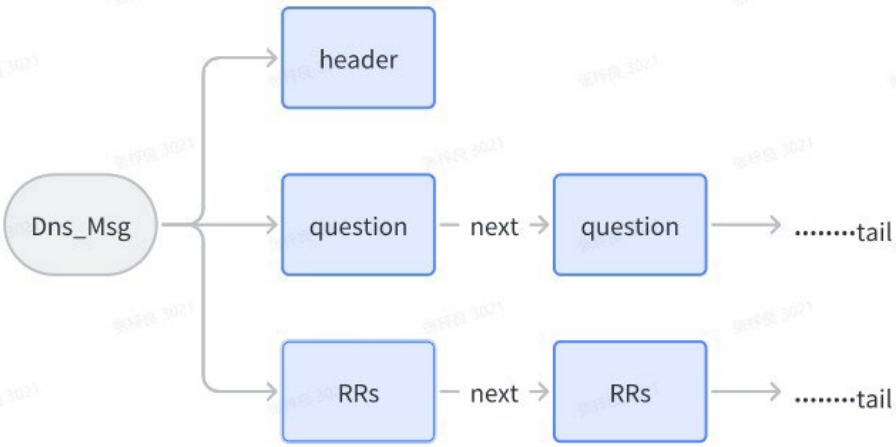
```
// DNS 报文 Header 部分
typedef struct
{
    unsigned short id;
    unsigned char qr : 1;
    unsigned char opcode : 4;
```

```
    unsigned char aa : 1;
    unsigned char tc : 1;
    unsigned char rd : 1;
     unsigned char ra : 1;
     unsigned char z : 3;
     unsigned char rcode : 4;
     unsigned short qdcount;
     unsigned short ancount;
     unsigned short nscount;
     unsigned short arcount;
} Dns_Header;

// DNS 报文 Question 部分
typedef struct Question
{
     unsigned char *qname;
     unsigned short qtype;
     unsigned short qclass;
     struct Question *next;
} Dns_Question;

// Resource Record
typedef struct RR
{
     unsigned char *name;
     unsigned short type;
     unsigned short _class;
     unsigned int ttl;
     unsigned short rdlength;
     unsigned char *rdata;
     struct RR *next;
} Dns_RR;

// DNS 报文
typedef struct
{
     Dns_Header *header;
     Dns_Question *question;
     Dns_RR *RRs;
} Dns_Msg;
```



自定义的 dns 报文结构体

4.2 msg_convert 模块

该模块定义了一些函数，用于将 DNS 消息的字节流表示形式转换为结构体表示形式，以及将结构体表示形式转换为字节流表示形式。这些函数包括：

---

1. `Dns_Msg *bytestream_to_dnsmsg(const unsigned char *bytestream, unsigned short *offset)`

该函数用于将 DNS 消息的字节流表示形式转换为结构体表示形式。

该函数首先动态分配了一个 Dns_Msg 结构体，并将其初始化为 NULL。然后，它调用 getHeader 函数，将字节流中的 DNS 消息头部分转换为 Dns_Header 结构体，并将其填入 Dns_Msg 结构体中。

接下来，该函数使用循环遍历 DNS 消息中的问题部分、资源记录部分，将它们转换为 Dns_Question 结构体和 Dns_RR 结构体，并将它们添加到 Dns_Msg 结构体中。在循环中，它使用 getQuestion 和 getRR 函数从字节流中提取出问题和资源记录的内容，并将它们填入相应的结构体中。

---

2. `unsigned char *dnsmsg_to_bytestream(const Dns_Msg *msg)`

该函数用于将 DNS 消息的结构体表示形式转换为字节流表示形式。

该函数首先动态分配了一个无符号字符数组，大小为 UDP_MAX，即 UDP 数据报的最大长度。接下来，该函数调用 putHeader 函数将 DNS 消息头部分从 Dns_Header 结构体转换为字节流格式，并将其写入无符号字符数组中。

然后，该函数使用循环将 DNS 消息的问题部分从 Dns_Question 结构体转换为字节流格式，并将其写入无符号字符数组中。循环遍历 Dns_Msg 结构体中问题链表中的每个问题，并调用 putQuestion 函数将每个问题转换为字节流格式并写入无符号字符数组中。

最后，该函数使用另一个循环将 DNS 消息的回答、授权和附加部分从 Dns_RR 结构体转换为字节流格式，并将其写入无符号字符数组中。循环遍历 Dns_Msg 结构体中资源记录链表中的每个资源记录，并调用 putRR 函数将每个资源记录转换为字节流格式并写入无符号字符数组中。

---

3. `void getHeader(Dns_Header *header, const unsigned char *bytestream)`

该函数首先从字节流中提取出 ID 字段，并使用 ntohs 函数将其从网络字节序转换为主机字节序。然后，它从字节流中提取出 QR（查询/响应）、OPCODE（操作码）、AA（授权回答）、TC（截断）、RD（递归期望）、RA（递归可用）、Z（保留以备将来使用）和 RCODE（响应码）等字段，并将它们存储在 Dns_Header 结构体的相应字段中。

该函数使用位移和掩码操作从字节流中提取字段的各个位。例如，通过该行代码：`header->qr = (bytestream[2] >> 7) & 1;`便能提取出 QR 字段。

---

4. `void getName(unsigned char *qname, const unsigned char *bytestream, unsigned short *offset)`

该函数使用 while 循环遍历字节流并从中提取域名。域名表示为标签序列，每个标签由一个长度字节和相应数量的标签字节组成。域名的结尾由一个长度为零的标签表示。

如果标签的长度字节的两个最高位设置为 1，则表示该标签是指向域名的另一部分的指针。在这种情况下，函数使用指针指定的新偏移量递归调用自身。

```
1. if (((*(bytestream + *offset) >> 6) & 3) == 3) // 压缩标签
2. {
3.     unsigned short new_offset = ntohs(*(unsigned short *)(bytestream + *offset)) & 0x3fff;
4.     getName(qname, bytestream, &new_offset);
5.     (*offset) += 2;
6.     return;
7. }
```

如果标签的长度字节的两个最高位未设置为 1，则表示该标签是数据标签。函数将标签的字节复制到"qname"数组中。

---

5. `void getQuestion(Dns_Question *quesiton, const unsigned char *bytestream, unsigned short *offset)`

该函数首先为 quesiton->qname 动态分配大小为 UDP_MAX 的无符号字符数组的内存，以便存储域名。函数调用"getName"函数从字节流中提取出域名，并将其存储在 Dns_Question 结构体的"qname"字段中。然后，函数从字节流中提取出 QTYPE（问题类型）和 QCLASS（问题类）字段，并将它们存储 Dns_Question 结构体的相应字段中。函数使用 ntohs 函数将字段从网络字节序转换为主机字节序。

---

6. `void getRR(Dns_RR *RR, const unsigned char *bytestream, unsigned short *offset)`

由于 resource record 部分的 name 字段和 question 部分的 qname 字段的格式相同，因此同样可以调用"getName"函数从字节流中提取资源记录的域名，并将其存储在 Dns_RR 结构体的"name"字段中。

该函数还从字节流中提取 TYPE、CLASS、TTL、RDLENGTH 和 RDATA 字段，并使用 ntohs 和 ntohl 函数将字段从网络字节序转换为主机字节序。

---

7. `void putHeader(const Dns_Header *header, unsigned char *bytestream)`

"putHeader"函数将 DNS 消息头中各个字段的值填充到字节流中。它使用位操作将某些字段打包成单个字节，并将其余字段以网络字节序存储。

---

8. `void putQuestion(const Dns_Question *que, unsigned char *bytestream, unsigned short *offset)`

"putQuestion"函数将 DNS 问题中各个字段的值填充到字节流中。它将问题的域名字段复制到字节流中，并填充 QTYPE 和 QCLASS 字段。

---

9. `void putRR(const Dns_RR *rr, unsigned char *bytestream, unsigned short *offset)`

"putRR"函数将 DNS 资源记录中各个字段的值填充到字节流中。它将资源记录的域名复制到字节流中，填充 TYPE、CLASS、TTL 和 RDLENGTH 字段，并复制 RDATA 字段。

---

10. `void transIPv4(unsigned char *original, unsigned char *IPv4)`

该函数使用"sprintf"函数将 IPv4 地址格式化为点分十进制格式的字符串。"sprintf" 函数将格式化的数据写入一个"IPv4" 的字符串中。格式字符串

"%d.%d.%d.%d"指定了输出字符串的格式应该是包含由句点分隔的四个十进制整数。

```
1. sprintf((char *)(IPv4), "%d.%d.%d.%d", original[0], origina
   l[1], original[2], original[3]);
```

四个十进制整数从二进制 IPv4 地址的四个字节中获得。第一个字节对应于第一个十进制整数，第二个字节对应于第二个十进制整数，以此类推

```
11. void transIPv6(unsigned char *original, unsigned char *IPv6)
```

该函数使用"sprintf"函数将 IPv6 地址格式化为冒分十六进制格式的字符串。"sprintf" 函数将格式化的数据写入一个"IPv6" 的字符串中。格式字符串"%x:%x:%x:%x:%x:%x:%x:%x"指定了输出字符串的格式应该是包含由冒号分隔的八个十六进制整数。

```
1. sprintf((char *)(IPv6), "%x:%x:%x:%x:%x:%x:%x:%x", ntohs(*(u
   nsigned short *)(original)), ntohs(*(unsigned short *)(origi
   nal + 2)),
2.         ntohs(*(unsigned short *)(original + 4)), ntohs(*(un
   signed short *)(original + 6)), ntohs(*(unsigned short *)(or
   iginal + 8)),
3.         ntohs(*(unsigned short *)(original + 10)), ntohs(*(u
   nsigned short *)(original + 12)), ntohs(*(unsigned short
   *)( original + 14)));
```

八个十六进制整数从二进制 IPv6 地址的十六个字节中获得。前两个字节对应于第一个十六进制整数，第二个两个字节对应于第二个十六进制整数，以此类推。

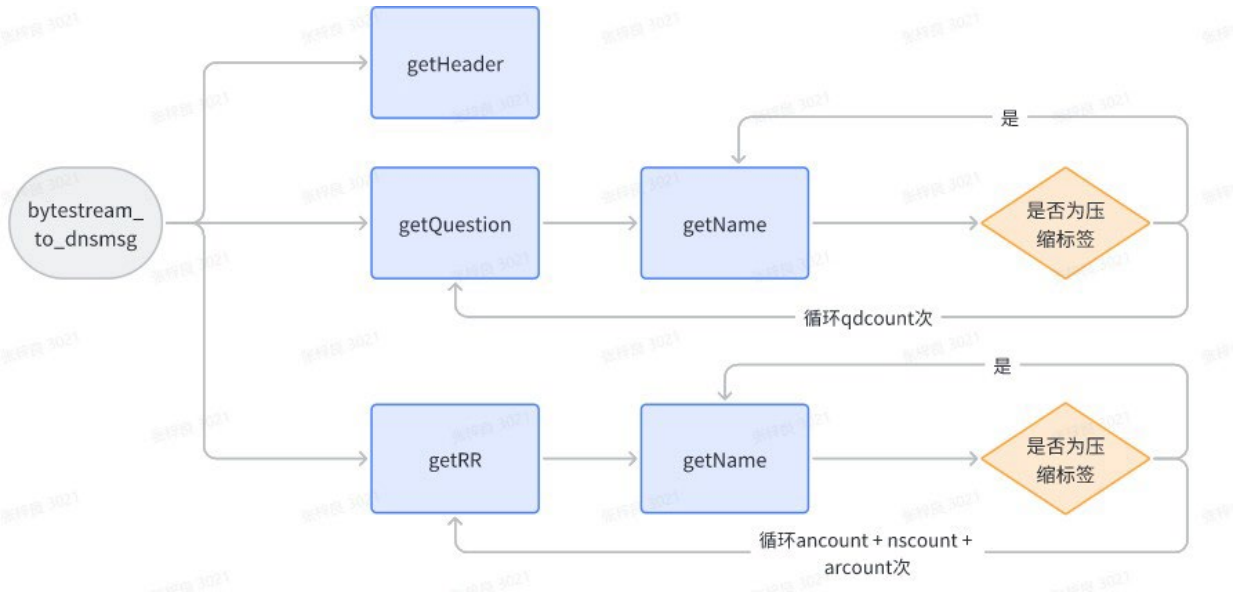
```
12. void transDN(unsigned char *original, unsigned char *DN)
```

"transDN"函数将 DNS 消息中的域名转换为点分隔格式的字符串。它使用 while 循环遍历 DNS 消息中的标签，并将每个标签的内容复制到"DN"数组中。最后，它将最后一个点号替换为空终止符，以便将"DN"数组中的字符串作为 C 语言字符串使用。

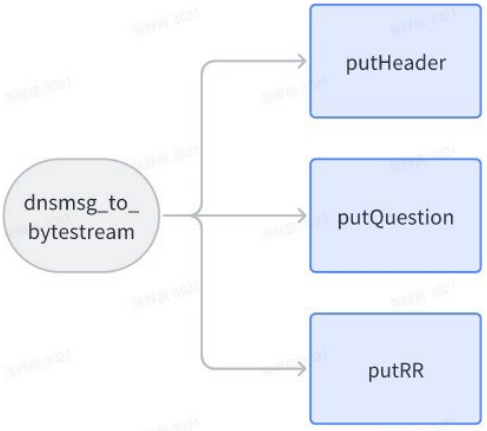
```
13. void releaseMsg(Dns_Msg *msg)
```

"releaseMsg"函数释放为 DNS 消息分配的内存。它检查"msg"指针是否为 NULL，如果不是，则使用"free"函数释放消息的数组分配的内存，并将指针设置为 NULL。这个函数对于防止程序出现内存泄漏非常重要。

下面附上该模块函数调用关系图：



字节流转换为结构体



结构体转换为字节流

4.3 dns_function 模块

该模块里定义了两个与 DNS 消息处理相关的函数："addAnswer"和"getDN_IP"。"addAnswer"函数向 DNS 消息添加一个 answer 记录，而"getDN_IP"函数从接收到的来自外部 DNS 服务器的 DNS 消息中提取域名和 IP 地址。

```
1. void addAnswer(Dns_Msg *msg, const unsigned char *IP, unsigned
    int _ttl, unsigned short _type)
```

"addAnswer"函数首先检查"_type"参数是否等于 TYPE_A 并且 IP 地址是否为 0。如果是，则代表查询的网站是不良网站，应进行拦截，则将 DNS 消息头的"rcode"字段设置为 3，表示域名不存在。然后，函数将 DNS 消息头的"qr"、"rd"和"ra"字段设置为 1，以指示这是一个响应消息，并且递归是期望和可用的。函数增加 DNS 消息头的"ancount"字段，以指示已向消息添加了一个 answer 记录。

接下来，函数向 DNS 消息的"RRs"数组添加一个资源记录。它首先遍历"RRs"数组，以找到数组中的最后一个资源记录。然后，它为一个新的资源记录分配内存，并将其字段设置为适当的值。"name"字段设置为 DNS 消息问题中的域名。"type"字段设置为"_type"参数。"_class"字段设置为 CLASS_IN，表示记录是一个 Internet 地址。"ttl"字段设置为"_ttl"参数。"rdlength"字段对于 IPv4 地址设置为 4，对于 IPv6 地址设置为 16。最后，"rdata"字段设置为 IP 地址。

```
2. void getDN_IP(const unsigned char *bytestream, unsigned char *
   DN, unsigned char *IP, unsigned int *_ttl, unsigned short *_ty
   pe)
```

"getDN_IP"函数首先将二进制 DNS 消息转换为 Dns_Msg 结构，并从 DNS 消息问题中提取域名。然后，它遍历 DNS 消息中的资源记录，以查找类型为 TYPE_A 或 TYPE_AAAA 的第一个记录，并从记录中提取 IP 地址和生存时间。

4.4 Cache 模块

1. 缓存项结构：
用于存储各个缓存项的信息，包括域名、IP 地址、过期时间等。

```
1. struct CacheEntry
2. {
3.     unsigned char domain[512]; // 域名
4.     unsigned char ipAddr[16];  // IPv4 地址
5.     unsigned char ipAddr6[40]; // IPv6 地址
6.     time_t expireTime;         // 过期时间
7.     struct CacheEntry *prev;   // 前驱指针
8.     struct CacheEntry *next;   // 后继指针
9. };
```

---

2. 缓存结构：
用于描述整个缓存，包括哈希表和链表。

```
struct Cache
{
    struct CacheEntry *table[CACHE_SIZE]; // 哈希表
    struct CacheEntry *head;              // 链表头指针
    struct CacheEntry *tail;              // 链表尾指针
};
```

---

3. 初始化函数 `void initCache(struct Cache *cache)`

用于初始化缓存，清空哈希表和链表。函数用于初始化缓存。它将哈希表清零，并将链表头和尾指针设置为 NULL。这样可以确保缓存在开始时是空的。

---

4. 计算哈希函数

`unsigned int hashCode(const unsigned char *domain)`

用于计算缓存项的哈希值。函数用于计算给定域名的哈希值。它使用 MurmurHash 算法计算哈希值，并将其模 CACHE_SIZE ， 以确保哈希值在 0 到 CACHE_SIZE-1 之间。哈希值用于将 DNS 记录存储在哈希表中。

```
#define FMIX32(h) \
    do { \
        (h) ^= (h) >> 16; \
        (h) *= 0x85ebca6b; \
        (h) ^= (h) >> 13; \
        (h) *= 0xc2b2ae35; \
        (h) ^= (h) >> 16; \
    } while (0)

uint32_t MurmurHash(const void *key, size_t len, uint32_t seed)
{
    const uint32_t m = 0x5bd1e995;
    const int r = 24;
    const uint8_t *data = (const uint8_t *)key;
    uint32_t h = seed ^ len;
    while (len >= 4)
    {
        uint32_t k = *(uint32_t *)data;
        k *= m;
        k ^= k >> r;
        k *= m;
```

```
            h *= m;
            h ^= k;
            data += 4;
            len -= 4;
        }
        switch (len)
        {
        case 3:
            h ^= data[2] << 16;
        case 2:
            h ^= data[1] << 8;
        case 1:
            h ^= data[0];
            h *= m;
        }
        FMIX32(h);
        return h;
    }
```

设计了一个 MurmurHash 哈希算法的 C 程序。MurmurHash 是一种快速、非加密的哈希函数，用于将任意长度的数据映射到固定长度的哈希值。

该程序定义了一个名为 MurmurHash 的函数，该函数接受三个参数：key，len 和 seed。key 是要哈希的数据，len 是数据的长度，seed 是哈希函数的种子值。

该函数使用 MurmurHash 算法计算哈希值。算法使用了一些常量，包括 m 和 r，这些常量用于计算哈希值。函数首先将种子值与数据长度异或，然后遍历数据，将每个 4 字节的块哈希到哈希值中。在处理每个块时，函数使用一些位运算和乘法来计算哈希值。最后，函数使用 FMIX32 宏来混合哈希值，以确保哈希值的均匀分布。

该函数的正确实现对于正确计算哈希值至关重要。MurmurHash 算法是一种高效的哈希算法，可以在短时间内计算出高质量的哈希值。因此，该函数可以用于许多不同的应用程序，例如哈希表、数据结构和密码学。

---

5. 查找缓存项

```
int findEntry(struct Cache *cache, const unsigned char *domain,
unsigned char *ipAddr, int ipVersion)
```

该部分定义了一个名为 findEntry 的函数，用于在缓存中查找给定域名的 DNS 记录。该函数使用哈希表和双向链表实现缓存，并使用最近最少使用（LRU）策略来管理缓存。

findEntry 函数接受三个参数：cache，domain 和 ipAddr。cache 是缓存结构体，domain 是要查找的域名，ipAddr 是用于存储 IP 地址的缓冲区。函数首先计算给定域名的哈希值，并遍历哈希表，查找与给定域名匹配的缓存条目。如果找到了条目并且没有过期，它将使用 LRU 策略将其移动到链表头部，并将 IP 地址存储在 ipAddr 缓冲区中。如果找到的条目已过期，则从哈希表和链表中删除。如果

未找到条目，则函数返回 0。

该函数的正确实现对于正确查找 DNS 记录至关重要。使用哈希表和双向链表可以快速查找和管理缓存。LRU 策略可以确保最近使用的缓存条目始终位于链表头部，从而提高缓存的命中率。

---

6. 添加缓存项

```
void addEntry(struct Cache *cache, const unsigned char *domain,
 const unsigned char *ipAddr, int ipVersion, time_t ttl)
```

该部分定义了一个名为 addEntry 的函数，用于将新的 DNS 记录添加到缓存中。该函数使用哈希表和双向链表实现缓存，并使用最近最少使用（LRU）策略来管理缓存。

addEntry 函数接受五个参数：cache，domain，ipAddr，ipVersion 和 ttl。cache 是缓存结构体，domain 是要添加的域名，ipAddr 是要添加的 IP 地址，ipVersion 是 IP 地址的版本（IPv4 或 IPv6），ttl 是 DNS 记录的生存时间。函数首先计算给定域名的哈希值，并创建一个新的缓存条目。然后，函数将域名、IP 地址和过期时间复制到缓存条目中，并将其添加到链表头和哈希表中。如果缓存已满，则函数从链表尾部删除过期的条目。

---

7. 删除过期缓存项

```
void removeExpiredEntries(struct Cache *cache)
```

该部分定义了一个名为 removeExpiredEntries 的函数，用于从缓存中删除过期的 DNS 记录。该函数使用双向链表实现缓存，并使用过期时间来管理缓存。

removeExpiredEntries 函数接受一个参数：cache，它是缓存结构体。函数首先获取当前时间，并从链表尾部开始遍历缓存条目。对于每个条目，如果它的过期时间早于当前时间，则从链表和哈希表中删除该条目，并释放其内存。如果缓存条目已经从链表中删除，则函数将继续遍历链表，直到找到一个未过期的缓存条目或链表为空。

---

8. 清空缓存 `void clearCache(struct Cache *cache)`

主要功能就是管理缓存，包括查找、添加、删除等。

4.5 id_converter 模块

1. 结构定义：

用于存储 ID 和客户端地址信息的结构。

```
1. typedef struct {
```

```
    unsigned short id;
    struct sockaddr_in clientAddr;
} Key;

typedef struct {
    Key key;
    unsigned short value;
} KeyValue;
```

2. 映射表 `KeyValue map[MAX_CLIENTS];` 用于存储 ID 和客户端地址的映射
全局变量 `unsigned short index = 0;` 用于查找下一个可用的映射位置。

3. 用于添加新的 ID 和客户端地址映射

```
int trans_port_id(unsigned short id, struct sockaddr_in clientAddr);
```

这部分用于将 ID 和客户端地址映射到一个整数值。该程序使用一个结构体数组来存储映射，其中每个元素包含一个键值对。该程序还使用一个布尔数组来跟踪哪些元素已经被使用。

trans_port_id 函数接受两个参数：id 和 clientAddr。id 是要映射的 ID，clientAddr 是客户端的地址。函数使用一个循环来查找未使用的位置，并将映射保存在该位置。如果所有位置都已经被使用，则函数将从头开始查找未使用的位置。一旦找到未使用的位置，函数将保存映射，并将该位置标记为已使用。最后，函数 返回映射的值。

4. 根据映射值查找原始 ID，查找原始客户端地址

```
unsigned short find_id(unsigned Value)
struct sockaddr_in find_clientAddr(unsigned Value)
```

5. 移除特定的映射

```
void remove_id(unsigned Value)
```

4.6 thread_pool 线程池模块

1. 线程参数结构：
用于描述线程需要的参数。

```
1. struct ThreadParam
```

```
2. {
3.     struct Trie *trie;
4.     struct Cache *cache;
5.     int sock;
6.     struct sockaddr_in clientAddr;
7.     int clientAddrLen;
8. };
```

2. 线程池结构：
用于描述整个线程池，包括线程参数数组和参数数量。

```
1. struct ThreadPool
2. {
3.     struct ThreadParam *params[MAX_THREADS]; // 线程池
4.     int count;                                // 线程池中
   空闲线程的数量
5. };
```

3. 初始化线程池和等待队列
`void init_thread_pool(struct ThreadPool *pool)`
这部分使用一个结构体来存储线程池的状态，包括线程数量和临界区。该程序还使用 Windows API 函数来创建信号量，以控制线程池中的线程数量。
init_thread_pool 函数接受一个参数：pool，它是线程池结构体。函数首先将线程数量设置为 0，并使用 InitializeCriticalSection 函数初始化临界区。然后，函数使用 CreateSemaphore 函数创建一个信号量，该信号量的初始值为 MAX_THREADS，最大值为 MAX_THREADS，并且没有名称。这个信号量用于控制线程池中的线程数量，以确保不会创建太多的线程。

4. 销毁线程池和等待队列
`void destroy_thread_pool(struct ThreadPool *pool)`
这部分使用一个结构体来存储线程池的状态，包括线程数量和临界区。该程序还使用 Windows API 函数来删除临界区。
destroy_thread_pool 函数接受一个参数：pool，它是线程池结构体。函数使用一个循环来释放线程池中的参数内存。然后，函数使用 DeleteCriticalSection 函数删除临界区。这个函数用于释放由 InitializeCriticalSection 函数分配的资源

5. 将 DNS 请求添加到线程池或等待队列
`void add_to_pool(struct ThreadPool *pool, struct ThreadParam *param)`
这部分用于将 DNS 请求添加到线程池或等待队列中。该程序使用一个结

构体来存储线程池的状态，包括线程数量和临界区。该程序还使用Windows API 函数来创建和释放信号量，以控制线程池中的线程数量。

add_to_pool 函数接受两个参数：pool 和 param。pool 是线程池结构体，param 是线程参数结构体。函数首先进入临界区，以确保线程池的状态在多线程环境下正确更新。然后，函数检查线程池中是否有空闲线程。如果有，则将参数添加到线程池中，并释放信号量以通知线程池中有新的任务。如果没有空闲线程，则函数将参数添加到等待队列中。最后，函数离开临界区

---

6. 线程函数，用于处理 DNS 请求

```
unsigned __stdcall threadProc(void *pParam)
```

这部分用于处理 DNS 请求的线程入口函数。该程序使用一个结构体来存储线程参数，包括 Trie、缓存、套接字和客户端地址。该程序还使用 Windows API 函数来创建线程。

threadProc 函数是线程的入口点，它接受一个参数：pParam，它是线程参数结构体的指针。函数首先将参数转换为线程参数结构体， 并使用 handle_dns_request 函数处理 DNS 请求。然后，函数释放参数内存，并返回 0。

4.7 trie 字典树模块

1. 节点结构：

存储一个域名对应的节点信息。

```
1. struct Node {
2.     char domain[];
3.     struct Node *next;
4. }
```

---

2. 字典树结构：

存储整棵字典树的信息，包括树状数组、节点前缀、结束标志、节点数量和 IP 地址。

```
1. struct Trie{
2.     int tree[][];
3.     int prefix[];
4.     bool isEnd[];
5.     int size;
6.     unsigned char toIp[][4];
```

```
7. }
```

3. 初始化字典树 `void initTrie(struct Trie *trie)`

这部分用于初始化 Trie 树。Trie 树是一种树形数据结构，用于存储字符串。该程序使用一个结构体来存储 Trie 树的状态，包括树、前缀、结束标志和 IP 地址。

initTrie 函数接受一个参数：trie，它是 Trie 树结构体。函数使用 memset 函数将 Trie 树的所有节点的值清零，将前缀数组清零，将结束标志数组清零，将 IP 地址数组清零，并将树的大小设置为 0。

---

4. 从文件中读取 domain 和 IP，插入到字典树中

```
void loadLocalTable(struct Trie *trie)
```

这部分用于从文件中加载 DNS 转发表并将其插入到 Trie 树中。Trie 树是一种树形数据结构，用于存储字符串。该程序使用一个结构体来存储 Trie 树的状态，包括树、前缀、结束标志和 IP 地址。

loadLocalTable 函数接受一个参数：trie，它是 Trie 树结构体。函数首先打开名为 dnsrelay.txt 的文件。如果文件打开失败，则打印错误信息并返回。然后，函数使用 fgets 函数逐行读取文件中的内容。对于每一行，函数使用 sscanf 函数解析出域名和 4 个字节的 IP 地址。如果解析失败，则打印错误信息并跳过这一行。最后，函数将域名和 IP 地址插入到 Trie 树中。

---

5. 简化域名 `void simplifyDomain(char domain[])`

这部分用于将域名中的大写字母转换为小写字母。该程序使用一个字符数组来存储域名。

simplifyDomain 函数接受一个参数：domain，它是一个字符数组，存储要简化的域名。函数使用 strlen 函数获取域名的长度，并使用一个循环遍历域名的每个字符。对于每个字符，如果它不是.和-，则使用 tolower 函数将其转换为小写字母。最后，函数在域名的末尾添加一个结束符。

---

6. 插入域名节点

```
void insertNode(struct Trie *trie, const char domain[], unsigned char ipAddr[4])
```

这部分用于在 Trie 树中插入一个域名和对应的 IP 地址。Trie 树是一种树形数据结构，用于存储字符串。该程序使用一个结构体来存储 Trie 树的状态，包括树、前缀、结束标志和 IP 地址。

insertNode 函数接受三个参数：trie，它是 Trie 树结构体；domain，它是要插入的域名；ipAddr，它是域名对应的 IP 地址。函数首先获取域名的长度，并使用 malloc 函数申请内存保存域名的副本。然后，函数简化域名，将大写字母转换为小写字母。接下来，函数遍历域名的每个字符，并根据字符的值选择相应的子节点。如果不存在对应 id 的子节点，则创建一个新的节点。然后，函数记录前缀，移动到下一个节点，并将当前节点标记为结束节点。最后，函数复制 IP 地

址，释放临时内存。

---

7. 删除域名节点

```
void deleteNode(struct Trie *trie, const unsigned char domain[
])
```

这部分用于在 Trie 树中删除一个域名和对应的 IP 地址。Trie 树是一种树形数据结构，用于存储字符串。该程序使用一个结构体来存储 Trie 树的状态，包括树、前缀、结束标志和 IP 地址。

deleteNode 函数接受两个参数：trie，它是 Trie 树结构体；domain，它是要删除的域名。函数首先检查域名是否为空串。然后，函数查找这个域名是否存在。如果不存在，则直接返回。接下来，函数将这个节点标记为非终止节点。然后，函数复制一份域名，用于后面的操作。如果这个节点不是根节点，则循环遍历域名的每个字符，并根据字符的值选择相应的子节点。如果这个节点的所有子节点都被删除了，就删除这个节点。最后，函数释放临时内存。

---

8. 找域名节点

```
int findNode(struct Trie *trie, const unsigned char domain[])
```

这部分用于在 Trie 树中查找一个域名对应的 IP 地址。Trie 树是一种树形数据结构，用于存储字符串。该程序使用一个结构体来存储 Trie 树的状态，包括树、前缀、结束标志和 IP 地址。

findNode 函数接受两个参数：trie，它是 Trie 树结构体；domain，它是要查找的域名。函数首先获取域名的长度，并使用 malloc 函数申请内存保存域名的副本。然后，函数简化域名，将大写字母转换为小写字母。接下来，函数遍历域名的每个字符，并根据字符的值选择相应的子节点。如果不存在对应 id 的子节点，则返回 0。如果找到的节点不是终止节点，则说明域名不在 Trie 树中，返回 0。最后，函数释放临时内存，返回找到的节点。

4.8 dns_relay_server 模块

1. 处理 DNS 请求

```
void handle_dns_request(struct Trie *trie, struct Cache *cache
, SOCKET sock, struct sockaddr_in clientAddr)
```

这部分用于处理 DNS 请求的函数。该函数接受四个参数：Trie 树、缓存、套接字和客户端地址。该函数使用 Windows API 函数来接收来自用户端的 DNS 请求字节流，并解析 DNS 报文。如果 DNS 请求报文中包含了域名，则该函数会查找域名对应的 IP 地址。如果找到了，则该函数会将 IP 地址添加到 DNS 响应报文中，并发送给用户端。如果没找到，则该函数会转发 DNS 请求报文给远程 DNS 服务

器。如果 DNS 请求报文是来自远程 DNS 服务器的 DNS 响应报文，则该函数会将 DNS 响应报文转发给用户端。如果 DNS 请求报文不是 DNS 请求报文或 DNS 响应报文，则该函数会直接转发 DNS 报文给远程 DNS 服务器。

---

2. 查找域名对应的 IP 地址

```
unsigned char *findIpAddress(struct Trie *trie, struct Cache *cache, unsigned char domain[MAX_DOMAIN_LENGTH])
```

这部分用于在 Trie 树和缓存中查找域名对应的 IP 地址。该函数接受三个参数：Trie 树、缓存和域名。该函数首先在缓存中查找域名对应的 IP 地址。如果找到了，则该函数会返回 IP 地址。如果没有找到，则该函数会在 Trie 树中查找域名对应的 IP 地址。如果找到了，则该函数会将 IP 地址添加到缓存中，并返回 IP 地址。如果在 Trie 树和缓存中都没有找到，则该函数会返回 NULL。

---

3. 向客户端发送 DNS 相应报文

```
void send_dns_response(int sock, Dns_Msg *msg, struct sockaddr_in clientAddr)
```

这部分用于向用户端发送 DNS 响应报文。该函数接受三个参数：套接字、DNS 报文和客户端地址。该函数首先将 DNS 报文转换为字节流，并计算字节流的长度。然后，该函数使用 Windows API 函数 sendto 将字节流发送给用户端。如果发送失败，则该函数会打印错误信息。如果发送成功，则该函数会打印 bytestream 信息和 DNS 报文信息，并提示发送成功。最后，该函数释放临时内存。

---

4. 转发 DNS 请求报文给远程 DNS 服务器

```
void forward_dns_request(int sock, unsigned char *buf, int len)
```

这部分用于将 DNS 请求报文转发给远程 DNS 服务器。该函数接受三个参数：套接字、DNS 请求报文和报文长度。该函数首先创建一个 sockaddr_in 结构体，用于存储远程 DNS 服务器的地址和端口号。然后，该函数使用 Windows API 函数 sendto 将 DNS 请求报文发送给远程 DNS 服务器。如果发送失败，则该函数会打印错误信息。如果发送成功，则该函数会打印 bytestream 信息和提示信息，并释放临时内存。

---

5. 转发 DNS 响应报文给用户端

```
void forward_dns_response(int sock, unsigned char *buf, int le
n, struct sockaddr_in clientAddr, unsigned short id)
```

这部分用于将 DNS 响应报文转发给客户端。该函数接受四个参数：套接字、DNS 响应报文、报文长度、客户端地址和 DNS 响应报文的 ID。该函数首先获取客户端地址的长度，然后使用 Windows API 函数 sendto 将 DNS 响应报文发送给客户端。如果发送失败，则该函数会打印错误信息。如果发送成功，则该函数会打印 bytestream 信息和提示信息。最后，该函数释放临时内存。

4.9 debug_info 模块

该模块定义了几个用于输出程序调试信息函数。这些函数用于输出 DNS 消息、资源记录、二进制字节流和程序执行时间等调试信息。

---

1. `void printTime()`

"printTime"函数用于打印程序的执行时间。它使用"clock"函数获取当前时钟时间，并通过将结束时间减去开始时间并将结果除以每秒钟的时钟滴答数来计算经过的时间。

---

2. `void RRInfo(Dns_RR *rr)`

"RRInfo"函数用于输出资源记录的调试信息，包括其名称、类型、类、生存时间值、数据长度和数据。

如果资源记录的类型为 TYPE_A，则函数将资源记录的 IPv4 地址从二进制格式转换为人类可读的格式，并将其存储在"IPv4"数组中。如果资源记录的类型为 TYPE_AAAA，则函数将资源记录的 IPv6 地址从二进制格式转换为人类可读的格式，并将其存储在"IPv6"数组中。

---

3. `void debug(Dns_Msg *msg)`

"debug"函数用于输出 DNS 消息的调试信息，包括头部字段、询问字段以及资源记录字段。

---

4. `void bytestreamInfo(unsigned char *bytestream)`

"bytestreamInfo" 函数用于输出二进制字节流的调试信息。 它使用"bytestream_to_dnsmsg"函数获取字节流长度并将其以十六进制表示打印到控制台。

调试信息输出如下图所示：

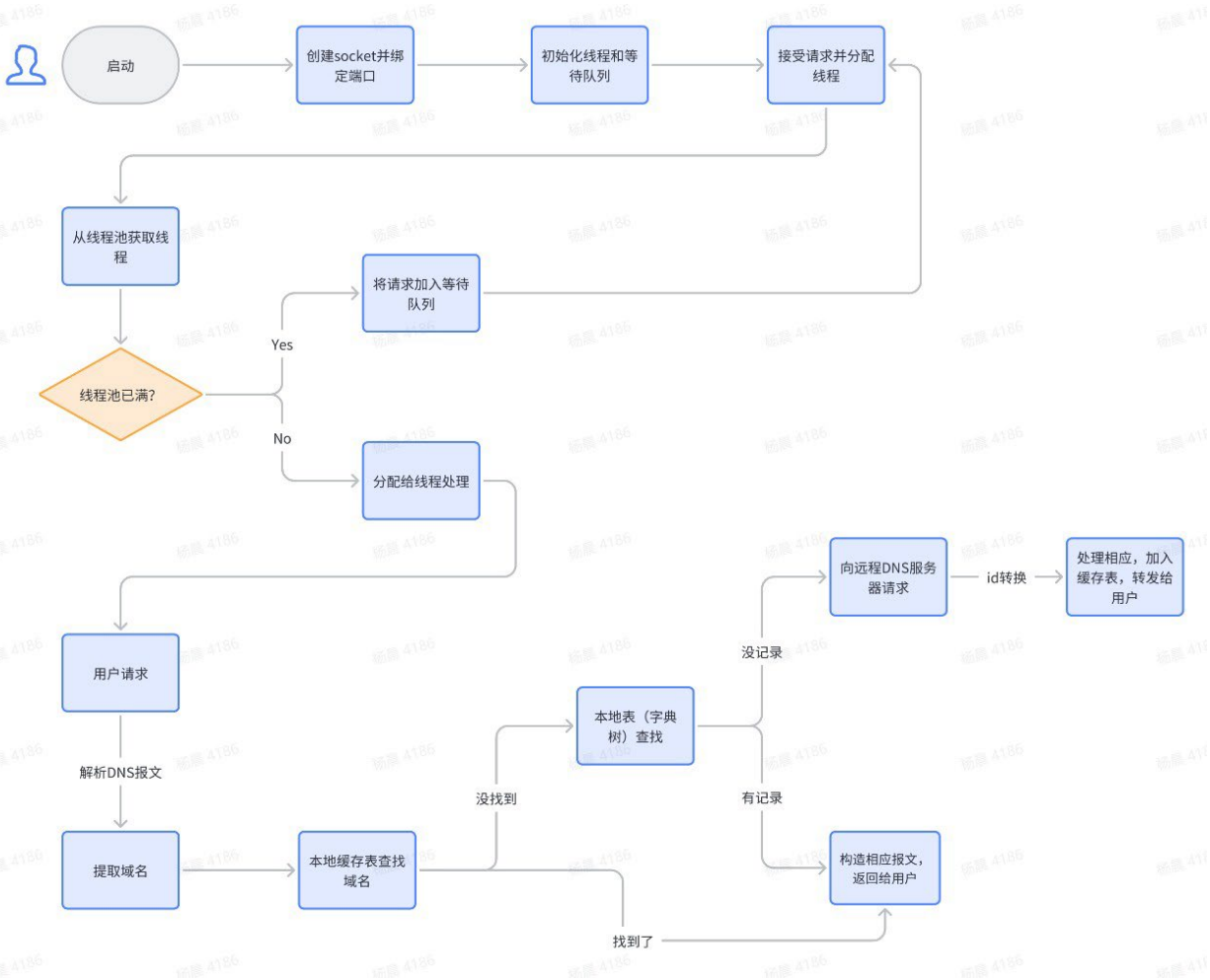
```
0.000:
----------------------HEADER----------------------
ID:62040  QR: 1  Opcode: 0  AA: 0  TC: 0  RD: 1  RA: 1
RCODE: 0  QDCOUNT: 1  ANCOUNT: 5  NSCOUNT: 0  ARCOUNT: 0
----------------------QUESTION----------------------
QUESTION 1
QNAME:config.teams.microsoft.com  QTYPE: 1  QCLASS: 1
----------------------ANSWER----------------------
RR 1
NAME:config.teams.microsoft.com  TYPE: 5  CLASS: 1
TTL:112  RDLENGTH:33
RR 2
NAME:config.teams.trafficmanager.net  TYPE: 5  CLASS: 1
TTL:119  RDLENGTH:28
RR 3
NAME:s-0005-teams.config.skype.com  TYPE: 5  CLASS: 1
TTL:6712  RDLENGTH:31
RR 4
NAME:config-teams.s-0005.s-msedge.net  TYPE: 5  CLASS: 1
TTL:119  RDLENGTH: 2
RR 5
NAME: s-0005.s-msedge.net  TYPE: 1  CLASS: 1
TTL:88  RDLENGTH: 4  RDATA:      52.113.194.132
--------------------------------------------------
```

各字段调试信息

```
0000: f2 58 81 80 00 01 00 05 00 00 00 00 06 63 6f 6e
0010: 66 69 67 05 74 65 61 6d 73 09 6d 69 63 72 6f 73
0020: 6f 66 74 03 63 6f 6d 00 00 01 00 01 c0 0c 00 05
0030: 00 01 00 00 00 70 00 21 06 63 6f 6e 66 69 67 05
0040: 74 65 61 6d 73 0e 74 72 61 66 66 69 63 6d 61 6e
0050: 61 67 65 72 03 6e 65 74 00 c0 38 00 05 00 01 00
0060: 00 00 77 00 1c 0c 73 2d 30 30 30 35 2d 74 65 61
0070: 6d 73 06 63 6f 6e 66 69 67 05 73 6b 79 70 65 c0
0080: 23 c0 65 00 05 00 01 00 00 1a 38 00 1f 0c 63 6f
0090: 6e 66 69 67 2d 74 65 61 6d 73 06 73 2d 30 30 30
00a0: 35 08 73 2d 6d 73 65 64 67 65 c0 54 c0 8d 00 05
00b0: 00 01 00 00 00 77 00 02 c0 9a c0 9a 00 01 00 01
00c0: 00 00 00 58 00 04 34 71 c2 84
```

字节流调试信息

5 程序流程图



6 测试用例及运行结果

```
PS D:\program\dns\relay_server\src> .\main
DNS中继服务器正在监听端口53.
```

6.1 使用 nslookup 测试

6.1.1 查询功能

> nslookup test2 127.0.0.1

```
C:\Users\Administrator>nslookup test2 127.0.0.1
服务器:  UnKnown
Address:  127.0.0.1

非权威应答:
名称:    test2
Addresses:  22.22.222.222
          22.22.222.222
```

```
收到来自用户端的DNS请求,域名为test2
在本地字典树查找成功,域名为test2,IP地址为22.22.222.222
中继服务器查找成功,域名为test2,IP地址为22.22.222.222
向用户端发送DNS响应报文成功
收到来自用户端的DNS请求,域名为test2
在本地字典树查找成功,域名为test2,IP地址为22.22.222.222
中继服务器查找成功,域名为test2,IP地址为22.22.222.222
向用户端发送DNS响应报文成功
```

6.1.2 不良网站拦截功能

>nslookup 008.cn 127.0.0.1

```
C:\Users\Administrator>nslookup 008.cn 127.0.0.1
服务器:  UnKnown
Address:  127.0.0.1

*** UnKnown 找不到 008.cn: No response from server
```

6.1.3 中继功能

>nslookup www.bupt.edu.cn 127.0.0.1

```
C:\Users\Administrator>nslookup www.bupt.edu.cn 127.0.0.1
服务器:  UnKnown
Address:  127.0.0.1

非权威应答:
名称:    vn46.bupt.edu.cn
Addresses:  2001:da8:215:4038::161
          10.3.9.161
Aliases:  www.bupt.edu.cn
```

```
收到来自用户端的DNS请求,域名为www.bupt.edu.cn
本地表和缓存表都未查找到域名www.bupt.edu.cn,需要访问远程DNS服务器
中继服务器查找失败,转发DNS请求报文给远程DNS服务器
向远程DNS服务器发送DNS请求报文成功
收到来自远程DNS服务器的DNS响应报文
向用户端发送DNS响应报文成功
```

6.2 测试网络连接功能

将本机的 DNS 修改为 127.0.0.1，然后输入以下命令清空 DNS 缓存。sudo dscacheutil -flushcache

启动程序，此时电脑仍然可以正常上网并进行域名解析，表示我们的 DNS 工作正常。

现在我们使用 ping 命令来测试网络联通情况。

6.2.1 不良网站拦截功能

>ping ctcv123

我们 ping 一下 ctcv123，提示域名无法解析，表明我们实现了拦截功能

```
C:\Users\Administrator>ping ctcv123
Ping 请求找不到主机 ctcv123。请检查该名称，然后重试。
```

6.2.2 服务器功能

>ping www.bupt.edu.cn

我们 ping 一下 www.bupt.edu.cn，网络连通良好

```
C:\Users\Administrator>ping www.bupt.edu.cn

正在 Ping vn46.bupt.edu.cn [10.3.9.161] 具有 32 字节的数据:
来自 10.3.9.161 的回复: 字节=32 时间=2ms TTL=59
来自 10.3.9.161 的回复: 字节=32 时间=1ms TTL=59
来自 10.3.9.161 的回复: 字节=32 时间=3ms TTL=59
来自 10.3.9.161 的回复: 字节=32 时间=2ms TTL=59

10.3.9.161 的 Ping 统计信息:
    数据包: 已发送 = 4，已接收 = 4，丢失 = 0 (0% 丢失)，
往返行程的估计时间(以毫秒为单位):
    最短 = 1ms，最长 = 3ms，平均 = 2ms
```

7　调试中所遇问题

7.1 内存泄露

在程序调试过程中，遇到了程序运行一段时间就崩溃的现象，编译器报出“段错误”的错误信息，其原因在于对于一个已经分配内存的指针再次分配内存或者对于一个空指针进行释放。但是出错的地方的指针是初次定义，按照理论逻辑来说不应该存在已经分配内存的情况，这个问题困扰了我们很久。我们通过单步调试，反复运行程序，查看程序崩溃前后相关变量以及指针的值，最终将问题锁定为内存泄露，因为内存泄露才导致一个初次定义的指针无法为其分配内存。

我们对程序代码进行了仔细的核查，最终将内存泄露的问题定位到了这行代码：

```
1. memcpy(msg->RRs->name, msg->question->qname, strlen((char *)
   (msg->question->qname)) + 1);
```

起初这行代码的 memcpy 函数的第三个参数没有+1，从而导致拷贝的字符串在末尾处不含‘\0’，从而导致在其他函数中访问 msg->RRs->name 会造成内存泄露。

7.2 初始化问题

在响应客户端的询问请求时，发送给客户端响应报文一直有错误，导致客户端无法正确解析报文。我们通过打印发送的响应报文的字节流，发现报文的第四个字节出错。经过排查，我们将错误定位到了构造响应报文的函数中，在该函数中，对字节流的第四个字节构造时未初始化，从而导致了第四个字节的实际值与理论值不符，造成错误。

```
1. bytestream[3] = 0; // 开始时缺少这行初始化的代码
2. bytestream[3] |= header->ra << 7;
3. bytestream[3] |= header->z << 4;
4. bytestream[3] |= header->rcode;
```

7.3 链表连接错误

报文自定义结构体的资源记录部分由单链表实现，在对单链表节点进行增加和删除的过程中，出现了链表断裂，节点丢失，或者顺序错误等问题。其原因是对于指针的应用掌握不够熟练，出现了指针乱指，空指等问题，修改代码后，问题得以解决。

7.4 多线程问题

多线程方案上，一开始采用的异步 I/O 技术，虽然能避免线程切换开销，但开启的 IO 线程过多影响效率。后使用了线程池技术，充分利用 CPU 资源，降低线程创建和销毁的开销。同时使用了等待队列来缓存任务，临界区和信号量来协调共享数据的访问。最终构建了一个高效的多线程模型。

7.5 Cache 存储问题

写缓存表时，由于没有清理完全已释放的节点，引起内存泄漏。通过单步调试，定位到头指针和尾指针丢失的 nodes 节点，进而导致链表断裂。修改代码后，在释放节点时完全清理，并将头尾指针指向下一个可用节点，成功解决内存泄漏问题。

7.6 ID 转换问题

一开始没有设置 ID 转换模块，使得多个客户端并发时，发送的报文 ID 可能相同。这就导致了转发的报文无法正确地路由到对应的客户端。为了解决这个问题，我设计了一个转换模块。其工作原理是:客户端发送报文给中继服务器时，会附带一个唯一的 socket 地址。中继服务器收到报文后，会将这个地址和 ID 绑定，同时生成一个内部 ID 给这个客户端。以后所有的报文转发的时候，都使用内部 ID 来 routing。这样就不管客户端发送的 ID 如何，都可以正确地将报文转发给对应的客户端。

7.7 命令行参数问题

最初命令行启动中继服务器时，使用了-d 参数，但没有指定端口号。系统因此会自动分配一个未用端口，通常是 0。由于 0 号端口无法被绑定，导致中继服务器无法正常监听。为了解决这个问题，我修改了启动命令，增加了-p 参数，用于指定绑定端口。这样中继服务器在启动的时候就可以正确地绑定一个有效的端口，从而实现监听。

7.8 Hash 表构建问题

在哈希表构建时，我最初使用的哈希函数存在较高的冲突率，并且哈希值的计算消耗比较高。为了解决这些问题，我改用了 murmurhash 算法计算哈希值。它具有较高的随机性，可以有效降低哈希冲突。此外 murmurhash 算法的计算效率也远高于简单取模的哈希函数。改用 murmurhash 后，哈希表的构建速度和空间利用率有了显著提高。

8 心得体会

通过本次课程设计，我们实践实现了一个 DNS 中继服务器。服务器能够以一个较为良好的性能实现拦截、服务、中继等功能。

在实验的开始前，我们通过仔细阅读参考 ppt、RFC 1035 文档以及通过运行样例程序进行观察，为实验的开始打下了理论基础。同时我们通过 wireshark 抓包软件进行抓包观察 DNS 报文的具体格式，清晰了 DNS 报文各个字段的具体用处。

在实验中我们从报文解析入手，结合所学习到的 DNS 报文结构的理论知识进行编程设计，自定义了 DNS 报文的结构体，实现了字节流和 DNS 报文结构体的相互转换，这个过程让我们对 DNS 报文的每个字段在报文中的位置、长度、以及每个字段的功能又有了一个更加深入的掌握。

同时，在实验的调试过程中，我们对多线程和内存管理有了更深的理解。我们发现，在多线程设计上，异步 I/O 虽然可以减少线程切换开销，但开启过多的 IO 线程会影响效率。后来我们采用了线程池技术，充分利用 CPU 资源，减少线程创建和销毁的开销。同时使用等待队列缓存任务，采用锁和信号量来协调共享数据的访问，构建了一个高效的多线程模型。

在内存管理方面，我们由于没有及时清理已释放的节点，导致内存泄漏。通过调试，我们定位到链表中丢失的节点，修改代码后彻底清理已释放节点，并更新头尾指针，成功解决了这个问题。这使我们认识到内存管理的重要性，节点或资源释放后要及时清理，更新相关指针，否则会造成“垃圾”节点长期占用空间。除此之外，我们还设计了 ID 转换模块，用于将不同客户端的报文 ID 转换为统一标识，保证可以正确识别和转发;修改了启动命令，增加了指定端口的参数，使服务可以监听多个客户端;优化了哈希表，采用 murmurhash 算法和链地址法解决哈希冲突，提高了性能。

总体而言，通过本次课程设计，让我们对多线程、内存管理、网络通信等有了更深刻的理解，也认识到软件调试的重要性。软件开发是一个不断学习和提高的过程，我们需要不断总结经验，遇到问题理清思路，在解决问题的同时也补充相关知识，不断进步。

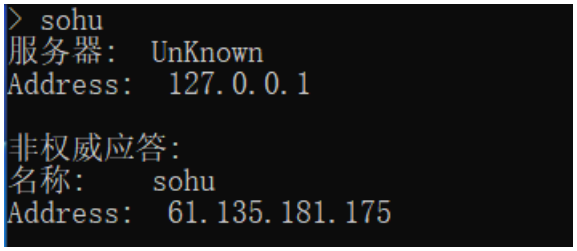
附录

在验收后，我们针对验收时老师指出的一些问题进行了修复，对程序的功能加以完善

问题一：返回两个相同的ipv4地址

通过对代码逻辑加以完善，现在的代码，可以同时返回ipv4和ipv6的地址（此处1.1.1.1是由于txt文档设置为了1.1.1.1）

对于没有真正ipv6地址的测试域名，只返回一个txt文档中的ip地址



问题二：由于我们用了c语言来设计ID转换，所以会有序号绕回后，发生堵塞的情况

现在，我们已经将序号绕回的逻辑更改为，一旦绕回，则丢弃早期的序号，因为这个数据包可能已经丢失