**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** | |
| 2021211312 | 23 | 2021211321 | | 梁晨锐 |  | |
| 2021211313 |  | 2021211313 | | 孙懿 |  | |
| **课**  **程**  **设**  **计**  **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：   1. 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） 2. 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） 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） | | | | | |
| **学生**  **课程设计**  **报告**  **（附页）** |  | | | | | |
| **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。

1. 概览
   1. 系统功能设计

本项目旨在设计和实现一个DNS中继服务器程序，实现以下功能：

* 读取包含“域名-IP地址”对照表的文件，将其存储在服务器内存中。
* 当客户端查询域名对应的IP地址时，先在内存中检索该对照表，得到以下三种检索结果：
* 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）
* 管理员用户一开始时将需要拦截的网站IP设置为“0.0.0.0”，储存在本地的域名-IP表中。在从本地表读出数据，且用户访问的域名对应IP为“0.0.0.0”时，服务器会对用户访问的域名进行拦截，直接丢弃本次访问的DNS包，以超时方式表示域名不存在。
* 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）
* 首先，DNS服务器通过读取cache查看是否在cache中存在此域名记录，若存在，则直接返回此域名对应的IP地址；否则继续在本地的dnsrelay.txt文件中查找是否有对应的记录，如果本地数据有记录是，服务器将直接将目的地IP地址提供给客户端，从而为用户提供有效的域名解析服务。
* 未检索到该域名，则向远程DNS服务器发出查询，并将结果返给客户端。（中继功能）
* 当在cache和本地域名-IP表中无法找到所需查询的域名IP时，需要向外部服务器进行询问。DNS中继服务器向因特网DNS发出查询请求，以获得域名解析信息。默认的外部DNS服务器地址是北邮的DNS服务器IP“10.3.9.4”，可通过命令行参数进行修改。
* 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. 需求分析

1.2.1 基本需求

* 解析和构建DNS报文。
* 监听本地53端口，获取DNS请求；将查询到的DNS回复发送回本地。
* 用相关数据结构维护本地“IP地址-域名”对照表，用于增添、查询DNS记录。
* 将DNS请求发送到远程DNS服务器，并且监听来自远程DNS服务器的回复。
* 屏蔽不良网站拦截功能
* 消息ID的转换

1.2.2 额外需求

除基本需求外，为实现 DNS 中继服务器的高性能和并发查询需求，我们还增加了如下的额外需求并完成实现。

* 多客户端并发：允许多个客户端（可能会位于不同的多个计算机）的并发查询，即：允许第一个查询尚未得到答案前就启动处理另外一个客户端查询请求
* 输出调试信息和DNS报文内容。
* 在转发DNS请求时重新分配序号，以区分不同的查询。
* Non-block（非阻塞） I/O + IO 多路复用 poll 模型：采用异步方式进行网络通信。
* 实现高速缓存，对于经常访问的记录，加快查询效率。
* 命令行参数解析。

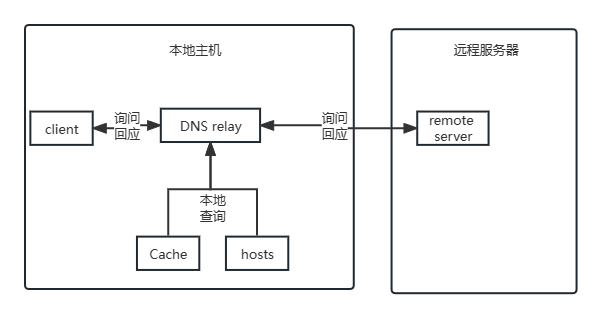
结合以上功能设计和需求分析，可以构建出以下的功能设计图：

Figure 1 ：功能设计

* 1. 开发环境
* Windows 10 x64
* Microsoft Visual Studio Community 2022 (64 位)
* Wireshark Version 4.0.5
  1. 成员分工
* 梁晨锐：顶层设计、网络编程模块设计、日志模块开发、文档编写、进度协调；
* 孙懿：数据结构开发、报文解析与组装模块开发、网络编程模块开发、程序测试、技术迭代

1. 模块划分

本项目可以划分为以下几个模块：

2.1 **DNS 查询报文解析**

2.1.1 DNS 报文格式分析

DNS请求报文以及响应报文具有统一的格式，可分为如下的5部分：

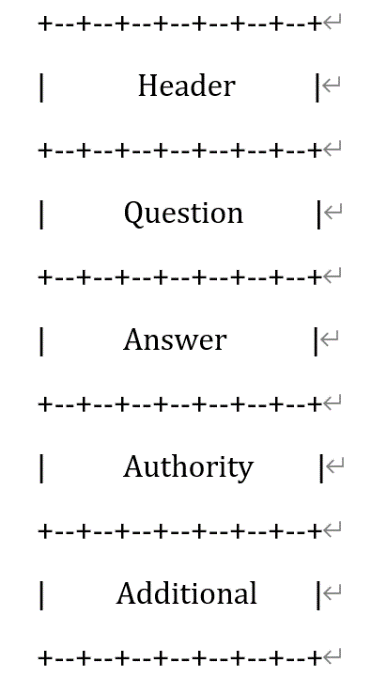


Figure 2 ：DNS 报文格式

* + - Header：报文头
    - Question：向域名服务器的查询

之后的三个部分有相同的格式，即 Resource Record(RR)，并且都可能为空；

* + - Answer 部分是对于查询的回复，
    - Authority 部分的内容指向权威域名服务器，
    - Additional 部分包含一些相关额外信息。

**Header 部分格式** Header 部分格式如下：

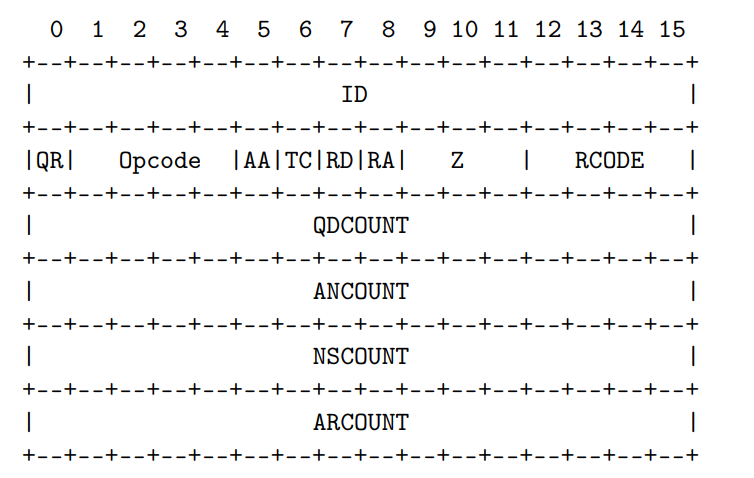


Figure 3 ：Header 部分格式

* ID，用于由产生 DNS 查询的程序分配，用于标识一个请求；一对 DNS 查询和回复的 ID相同。
* QR，查询报文此位为 0，回复报文此位为 1。
* OPCODE，查询的类型：
* 0，标准查询；
* 1，反向查询；
* 2，服务器状态请求。
* AA，在回复报文中有效，如果为 1，表示回复 Question 部分查询的域名服务器是权威服务器。
* TC，如果为 1，说明这条消息由于信道的限制而被截断。
* RD，在查询报文中设置，如果为 1，表示期望域名服务器递归查询这个请求。
* RA，在回复报文中设置，如果为 1，表示递归查询在域名服务器中有效。
* Z，预留字段，全 0。
* RCODE，回复状态编号：
* 0，没有错误；
* 1，查询格式错误；
* 2，由于服务器错误而无法处理查询；
* 3，域名错误，仅在权威服务器的回复中有意义，指查询中请求的域名不存在；
* 4，查询的类型不受支持；
* 5，服务器拒绝处理请求。
* QDCOUNT， Question 部分中查询记录的个数（通常是 1）。
* ANCOUNT， Answer 部分中 RR 的个数。
* NSCOUNT， Authority 部分中 RR 的个数。
* ARCOUNT， Additional 部分中 RR 的个数。

**Question 部分格式** Question 部分格式如下：

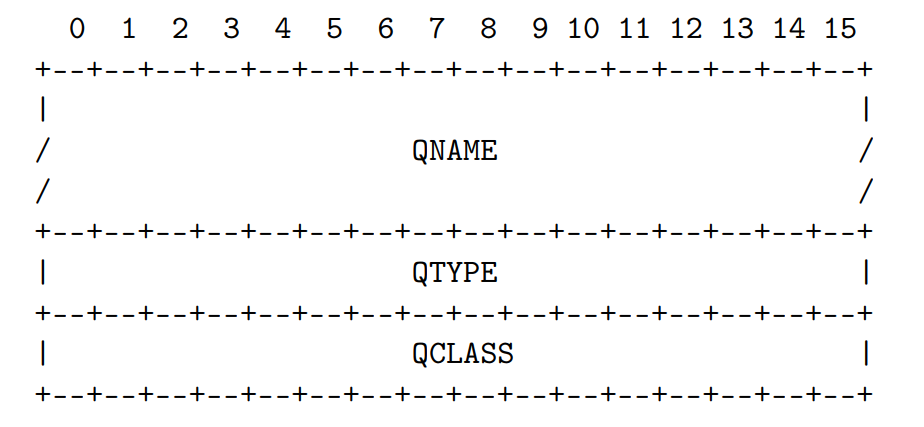


Figure 4 ：Question 部分格式

* QNAME，查询域名。
* QTYPE，查询类型，受支持的类型如下：
* 1，A，主机地址；
* 2， NS，权威域名服务器；
* 5， CNAME，域名引用；
* 6， SOA，授权机构起始；
* 12， PTR，域名指针，用于反向域名查找；
* 13， HINFO，主机信息；
* 14， MINFO，邮箱或邮件列表信息；
* 15， MX，邮件交换；
* 16， TXT，字符串。
* 28， AAAA， IPv6 地址。
* QCLASS，查询类别，仅支持 1， IN，因特网。

**Resource Record 部分格式** Resource Record 部分格式如下：

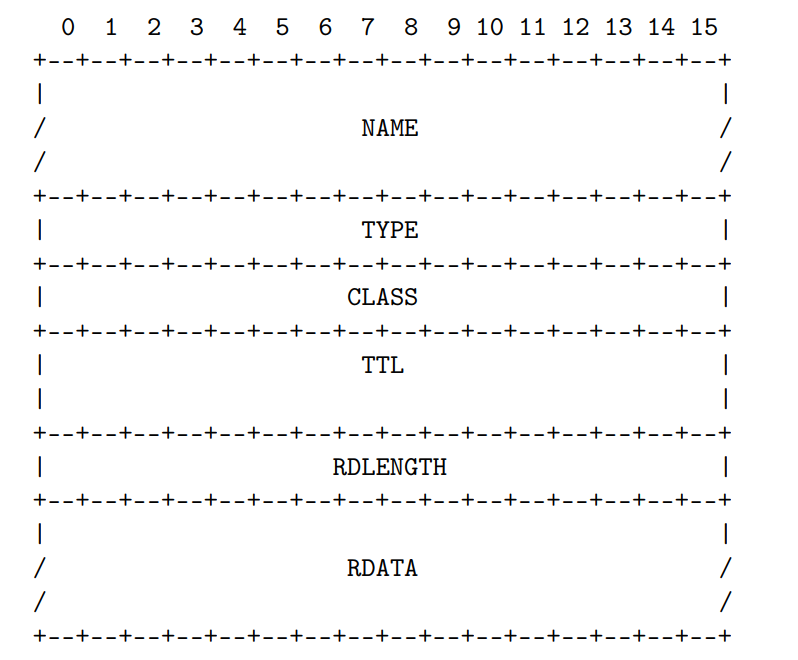


Figure 5 ：Resource Record 部分格式

* NAME，此 RR 从属的域名，格式在3.2.1节说明。
* TYPE， RR 类型，支持的类型与 QTYPE 部分相同。
* CLASS， RR 类别，仅支持 1， IN，因特网。
* TTL，期望此 RR 被缓存的时间。
* RDLENGTH， RDATA 部分的长度。
* RDATA，资源内容，根据 TYPE 的不同，其内容也不同。

2.1.2 DNS报文解析模块

该模块根据RFC 1035标准中规定的 DNS 信息格式，将收到的 DNS 查询报文解析为具体的查询请求，用以后续对于是否命中本地记录的判断。解析的查询信息将被保存在如下的数据结构中：

1. **typedef** **struct** DNS\_message {
2. **struct** DNS\_header\* header;
3. **struct** DNS\_question\* questions;
4. **struct** DNS\_resource\_record\* answers;
5. **struct** DNS\_resource\_record\* authorities;
6. **struct** DNS\_resource\_record\* additionals;
7. } dns\_message;

解析完成的报文中保存了报文头部信息、报文中问题节包含的 DNS 查询问题等。其中，解析的报文头部 header 保存在如下的数据结构中：

1. **typedef** **struct** DNS\_header {
2. uint16\_t id; // 标识符，一对DNS查询和恢复的ID相同
4. /\* 以下变量占用空间大小仅为几个比特，故定义位域，压缩空间以高效利用内存 \*/
5. uint8\_t qr:1;     // 0 查询 1 回复
6. uint8\_t opcode:4; // 0 标准查询 1 反向查询 2 服务器状态请求
7. uint8\_t aa:1;     // 授权回答：如果为1，表示回复的Question部分查询的域名服务器是权威服务器
8. uint8\_t tc:1;     // 截断：如果为1，表示这条消息由于信道的限制而被截断
9. uint8\_t rd:1;     // 期望递归：如果为1，表示期望域名服务器递归查询这个请求
10. uint8\_t ra:1;     // 可用递归：如果为1，表示递归查询在域名服务器总有效
11. uint8\_t z:3;      // 预留字段
12. uint8\_t rcode:4;  // 返回码 0 无差错 1 格式错误 2 服务器错误 3 名字错误 4 无实现 5 拒绝 6 - 15 保留
14. /\* 以下变量大小均为16位 \*/
15. uint16\_t qdCount; // 问题数（通常是1）
16. uint16\_t anCount; // 回答数
17. uint16\_t nsCount; // 授权数
18. uint16\_t arCount; // 附加数
19. } dns\_header;

查询问题 question 保存在下面的数据结构中：

1. **typedef** **struct** DNS\_question {
2. **char**\* q\_name;              // 域名或IP地址
3. uint16\_t q\_type;           // 资源类型
4. uint16\_t q\_class;          // 地址类型，通常为1
5. **struct** DNS\_question\* next;
6. } dns\_question;

资源记录 resource record(RR) 保存在下面的数据结构中：

1. **typedef** **struct** DNS\_resource\_record {
2. **char**\* name;                  // 域名
3. uint16\_t type;               // resource data类型
4. uint16\_t rr\_class;           // 仅支持1，IN，因特网
5. uint32\_t ttl;                // 期望此RR被缓存的时间
6. uint16\_t rd\_length;          // RDATA部分的长度
7. **union** ResourceData rd\_data;  // 资源内容
8. **struct** DNS\_resource\_record\* next;
9. } dns\_rr;

包含了这一模块主要功能函数的头文件位于 include/dns\_struct.h 中。

2.1.3 DNS报文字节流和结构体的转换

从 UDP socket 中获取到的 DNS 报文是字节流的形式，我们需要将它转换成结构体，便于分析与输出，之后需要将结构体转换回字节流进行发送。

1. **size\_t** get\_bits(uint8\_t\*\* buffer, **int** bits); // 读取指定数目的字节
3. **void** set\_bits(uint8\_t\*\* buffer, **int** bits, **int** value); // 以网络字节序写指定数目的字节
5. uint8\_t\* get\_domain(uint8\_t\* buffer, **char**\* name);  // 从字节流提取域名
7. **void** get\_message(dns\_message\* msg, uint8\_t\* buffer);
9. **void** set\_message(dns\_message\* msg, uint8\_t\* buffer);
11. uint8\_t\* get\_header(dns\_message\* msg, uint8\_t\* buffer);
13. **void** set\_header(dns\_message\* msg, uint8\_t\* buffer);
15. uint8\_t\* get\_question(dns\_message\* msg, uint8\_t\* buffer);
17. **void** set\_question(dns\_message\* msg, uint8\_t\* buffer);
19. uint8\_t\* get\_answer(dns\_message\* msg, uint8\_t\* buffer);

将字节流转换成结构体：通过 get\_message() 函数实现

1. **void** get\_message(dns\_message\* msg, uint8\_t\* buffer, uint8\_t\* start) {
2. /\* 开辟空间 \*/
3. msg->header = malloc(**sizeof**(dns\_header));
4. msg->questions = malloc(**sizeof**(dns\_question));
5. msg->answers = malloc(**sizeof**(dns\_rr));
7. /\* 获取报文头 \*/
8. buffer = get\_header(msg, buffer);   // buffer指向读取完报头后的地址
10. /\* 获取询问内容 \*/
11. buffer = get\_question(msg, buffer, start); // buffer指向读取完询问内容后的地址
13. /\* 获取应答内容 \*/
14. buffer = get\_answer(msg, buffer, start);   // buffer指向读取完应答内容后的地址
15. }

该函数首先使用 malloc函数为DNS报文、查询部分、应答部分分配内存，然后分别解析以上部分的信息到DNS结构体内。

将结构体转换成字节流：通过 set\_message() 函数实现

1. **uint8\_t\*** set\_message(dns\_message\* msg, uint8\_t\* buffer, uint8\_t\* ip\_addr) {
2. **uint8\_t\*** start = buffer;
3. /\* 组装报头 \*/
4. buffer = set\_header(msg, buffer, ip\_addr);
5. /\* 组装询问 \*/
6. buffer = set\_question(msg, buffer);
7. /\* 组装回答 \*/
8. buffer = set\_answer(msg, buffer, ip\_addr);
9. return buffer;
10. }

调用函数set\_header组装DNS报头，并更新buffer指针， 以此类推完成其他部分的组装。包含了这一模块主要功能函数的头文件位于 include/dns\_struct.h 中。

2.2 DNS 记录存储

LyDNS 能够加载本地域名记录．当请求的域名信息在本地记录中时， LyDNS 不会进行转发，而是直接将本地记录返回。对于 DNS 记录的存储可以通过一个域名和类型的对应关系，查询到一个 Resource Record 列表作为回复。

2.2.1 记录文件的加载与解析

LyDNS 能够加载文本格式的域名记录文件；每一行为一条记录，每条记录是以空格分隔的域名及其相应的 IP 地址。

记录文件解析模块能够读取并解析记录文件中所有的域名记录信息．包含这一模块主要功能函数的头文件位于 include/system.h 中。读取一个文件中的所有记录通过下面的函数完成．

1. **void** read\_host() {
2. **FILE**\* host\_ptr = fopen(host\_path, "r");
4. **if** (!host\_ptr) {
5. printf("Error! Can not open hosts file!\n");
6. exit(1);
7. }
8. get\_host\_info(host\_ptr);
9. }
11. **void** get\_host\_info(**FILE**\* ptr) {
12. **while** (!feof(ptr)) {
13. uint8\_t this\_ip[4];
15. fscanf(ptr, "%s", IPAddr);
16. fscanf(ptr, "%s", domain);
18. transfer\_IP(this\_ip, IPAddr);
19. add\_node(list\_trie, this\_ip, domain);
20. }
21. }

该部分函数返回读取的记录列表头部指针，并且能够返回读取的记录数量与返回值。

2.2.2 Trie树

为了提高查找效率，在读取本地DNS记录时，将本地DNS 记录存储在一个 Trie树中。Trie树又叫字典树或前缀树，是一种多叉树结构。如下图：

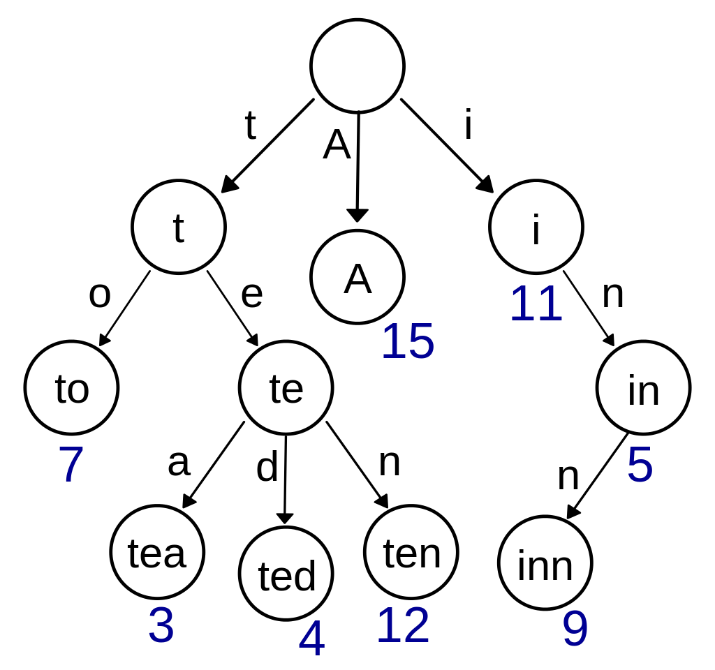


Figure 6：trie 树结构示例

Trie树的基本性质：

* 根节点不包含字符，除根节点外的每一个子节点都包含一个字符。
* 从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。
* 每个节点的所有子节点包含的字符互不相同。

Trie树的关键字一般都是字符串，而且Trie树把每个关键字保存在一条路径上，而不是一个结点中。另外，两个有公共前缀的关键字，在Trie树中前缀部分的路径相同，所以Trie树又叫做前缀树。

Trie 树的数据结构和相关操作定义在 data\_struct.h 中：

1. **typedef** **struct** trie\_node {
2. uint16\_t pre;           // 父结点编号
3. uint16\_t val[37];       // 读取域名中的每个字符的编号num，则val[num]存放读入某字符时，该结点的编号
4. uint8\_t IP[4];          // 十进制IP地址
5. uint8\_t isEnd;          // 是否为一个域名的结束
6. } trie;
8. /\* 增加、查询字典树结点 \*/
9. **void** add\_node(trie\* root, uint8\_t\* IP, **char**\* domain);
10. **int** query\_node(trie\* root, **char**\* domain);

使用 Trie树存储域名和IP地址，能够快速的对字符串进行插入、查找，时间复杂度可近似认为O(1)，因其只与字符串长度有关，比其他数据结构具有性能更优的特点（如map查询、插入功能的复杂度都是O(logn)）。

2.2.2 高速缓存 Cache 模块

除了 Trie树之外， LyDNS 也设计并实现了一个高速缓存（Cache ），高速缓存的设计是为了提高DNS服务器的查找效率。根据局部性原理，LyDNS 实现了一个高性能的 LRU 缓存。我们设计了一个链表，里面记录了域名与IP的一对一映射。当DNS中继服务器接收到客户端的DNS请求时，便将从本地找到的域名和IP的对应关系保存到Cache 中，若以后又收到相同的域名请求，就直接将Cache中的记录返回给客户端。在缓存中查找域名所需的时间为 O(n)。

此模块采用了LRU（Least Recently Used）替换算法。因为实际服务器的缓存容量有限，当缓存记录已满又有新的缓存需要添加时，需要替换掉其中一条记录。此时，我们根据LRU算法替换掉最近使用的对应记录，根据研究和测试，我们发现该算法能够较大的增加Cache命中率并提升程序效率。

LRU算法是通过链表实现的，插入的时间复杂度是O(1)，查找和删除的时间复杂度是O(n)，由于Cache的容量设置不大，Cache运行效率是相当高的。对于在这个链表中，最近被访问的记录会被移到链表尾，当链表长度达到上限时，优先移出链表头的节点，即最久未使用的值最先被移出缓存。

LRU算法原理如下图所示：



Figure 7 ：LRU 原理

LRU 的数据结构和相关操作定义在 data\_struct.h 中：

1. **typedef** **struct** node {
2. uint8\_t IP[4];
3. **char** domain[MAX\_SIZE];
4. **struct** node\* next;
5. } lru\_node;
6. /\* 初始化缓存 \*/
7. **void** init\_cache();
9. /\* 从缓存链表中查询 \*/
10. **int** query\_cache(uint8\_t ipAddr[4], **char**\* domain);
12. /\* 更新缓存链表 \*/
13. **void** update\_cache(uint8\_t ipAddr[4], **char**\* domain);
15. /\* 删除最远未使用结点 \*/
16. **void** delete\_cache();

2.3 信息转发

若查询请求并非 A 型请求，或请求的域名不在本地记录中，则会将 DNS 报文转发给远端 DNS 服务器。

2.3.1 RDATA 格式

对于 DNS 中继服务器来说， RDATA 部分的内容是查询请求类型的一方面体现。需要处理的 RDATA 如下。

**A 类型 RDATA 格式**

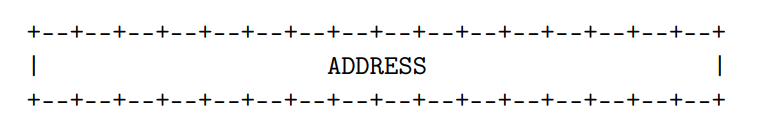


figure 8：A 类型 RDATA 格式

* ADDRESS， 32 位 IPv4 地址。

AAAA 类型 RDATA 格式

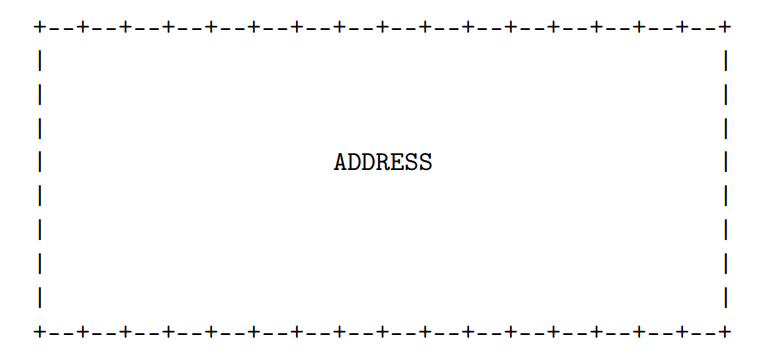


figure 9 ：AAAA 类型 RDATA 格式

* ADDRESS， 128 位 IPv6 地址。

CNAME 类型 RDATA 格式

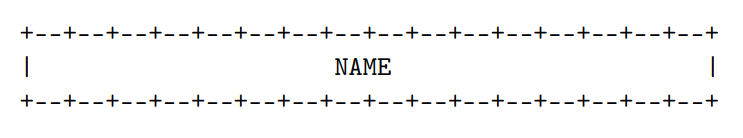


figure 10：CNAME 类型 RDATA 格式

* NAME，一个域名。

若查询请求并非 A 型请求，则会将 DNS 报文转发给远端 DNS 服务器。

2.3.2 本地查询未命中

若本地查询未命中，即一级 Cache 和二级 trie 树中都没有找到客户端发送的待查询ip地址对应的域名，则将会此查询转发给远程服务器（默认为北邮DNS：10.3.9.45，可以自定义）处理，再调用 receive\_server() 函数接受远程DNS服务器发来的DNS应答报文，并将DNS应答报文转发回客户端，同时将此报文写入 Cache 缓存中，为下一次查询节省 DNS 解析时间。receive\_server() 函数代码如下：

1. **void** receive\_server() {
2. uint8\_t buffer[BUFFER\_SIZE]; // 接收的报文
3. dns\_message msg;
4. int msg\_size = -1; // 报文大小
5. /\* 接受远程DNS服务器发来的DNS应答报文 \*/
6. if (is\_listen == 1) {
7. msg\_size = recvfrom(server\_sock, buffer, sizeof(buffer), 0, (struct sockaddr\*)&server\_addr, &addr\_len);
8. get\_message(&msg, buffer, buffer);
9. }
10. /\* 将DNS应答报文转发回客户端 \*/
11. if (msg\_size > 0 && is\_listen == 1) {
12. /\* ID转换 \*/
13. uint16\_t ID = msg.header->id;
14. uint16\_t old\_ID = htons(ID\_list[ID].client\_ID);
15. memcpy(buffer, &old\_ID, sizeof(uint16\_t)); //把待发回客户端的包ID改回原ID
16. struct sockaddr\_in ca = ID\_list[ID].client\_addr;
17. ID\_list[ID].expire\_time = 0;
18. sendto(client\_sock, buffer, msg\_size, 0, (struct sockaddr\*)&client\_addr, addr\_len);
19. is\_listen = 0;
20. write\_log(msg.questions->q\_name, NULL);
21. }
22. }

2.4 多客户端并发

在实现中继功能时，考虑到要实现应答多客户端的请求，我们得进行头部ID的转换。因为每个客户端发来的DNS询问报文都是根据本地数据进行编号的，我们在转发给外部DNS服务器时，得按照DNS中继服务器的规则进行编号，并把旧ID和新ID的转换关系储存到本地的ID转换表中。在接受到外部服务器返回的DNS报文时，通过ID转换表，将ID转换为之前客户端发来的ID，再将报文转发给客户端。

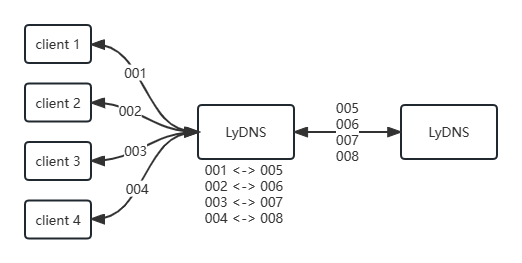


Figure 11 ：多客户端并发查询时的 ID 转换

正常情况下，socket工作在阻塞模式下，在调用accept，connect，read，write等函数时，都是阻塞方式，直到读到数据才会返回。这样的效率会很低，并且无法实现多用户并发查询。但是，如果将socket设置为非阻塞状态，那么这么些函数就会立即返回，不会阻塞当前线程。

因此，LyDNS 首先选用了 non-block(非阻塞) socket 进行网络编程的开发。非阻塞套接字是指执行此套接字的网络调用时，不管是否执行成功，都立即返回。比如调用recv()函数读取网络缓冲区中数据，不管是否读到数据都立即返回，而不会一直挂在此函数调用上。在实际Windows网络通信软件开发中，异步非阻塞套接字是用的最多的。平常所说的C/S（客户端/服务器）结构的软件就是异步非阻塞模式的。

将 socket 设置成非阻塞模式的函数实现如下所示：

1. **int** nonblock() {
2. /\* 设置为非阻塞模式 \*/
3. block\_mode = 1;
5. **int** server\_result = ioctlsocket(server\_sock, FIONBIO, &block\_mode);
6. **int** client\_result = ioctlsocket(client\_sock, FIONBIO, &block\_mode);
8. **if** (server\_result != 0 || client\_result != 0) {
9. // 设置失败
10. printf("ioctlsocket failed with error: %d\n", WSAGetLastError());
11. closesocket(server\_sock);
12. closesocket(client\_sock);
13. **return** 1;
14. }
16. **while** (1) {
17. receive\_client(); // 接收来自客户端的数据
18. receive\_server(); // 接收来自服务器的数据
19. }
20. }

更进一步， LyDNS 在非阻塞模式的设计基础上进行了深入的改进，设计了 IO 多路复用中高性能的 poll 模型，底层通过轮询机制来判断每个socket读写是否就绪，而不是直接被 IO 堵塞。在本报告的 “5. 调试中遇到并解决的问题”中将详细介绍本部分内容。

2.5 超时处理

由于 UDP 提供的是不可靠的传输服务，因而可能会出现数据包丢失的情况．因而为 UDP 请求设置超时时间是必要的．一方面，若发生丢包，没有超时机制的请求将一直被挂起，占用资源．另一方面，超时机制也能够保证 DNS 编号不会发生重复。故而设置超时阈值判断外部请求是否超时。当一个DNS本地服务器向外部DNS服务器发出请求时，设置一个最长等待时间，即超时定时器 expire\_time ，当超过此时间，就将此ID从ID转换表中删除，并向客户端发送响应的回答报文。

为了实现超时处理和并发查询，我们实现了一个查询池，查询池中的内容为 DNS 查询。中继服务器会为每个 DNS 查询分配一个序号，即完成 ID 的转换，同时保存以下信息：

* 请求方的地址，中继服务器向这个地址发送回复。
* 原本的 DNS 查询报文，中继服务器收到远程回复时和原本的查询报文比对。
* 计时器， DNS 查询超时后自动从查询池中移除并销毁。

其实现代码如下：

1. **typedef** **struct**
2. {
3. uint16\_t clientId;
4. **int** expireTime; // time IdConversion expired
5. **struct** sockaddr\_in clientAddr;
6. } IdConversion;
8. IdConversion IdTable[ID\_TABLE\_SIZE]; // ID conversion table

同时，我们使用了 nslookup 的超时机制。当使用nslookup命令进行域名解析时，如果超过了默认的超时时间，nslookup会返回一个超时错误。默认情况下，nslookup的超时时间是2秒。

2.6 命令行参数解析模块

我们为 LyDNS 设置了功能完善的参数，如下所示：

* -d 输出调试信息
* -l 输出日志
* -i 输出系统基本信息
* -m [mode] 选择模式，mode = 0：非阻塞，mode = 1：poll
* -s [server\_address] 设置远程DNS服务器，默认为10.3.9.45（北邮DNS）

2.7 日志输出模块

LyDNS 的工作日志为 log.txt。工作日志的条目包括：

* 查询时间
* 查询的域名
* 应答的IP地址

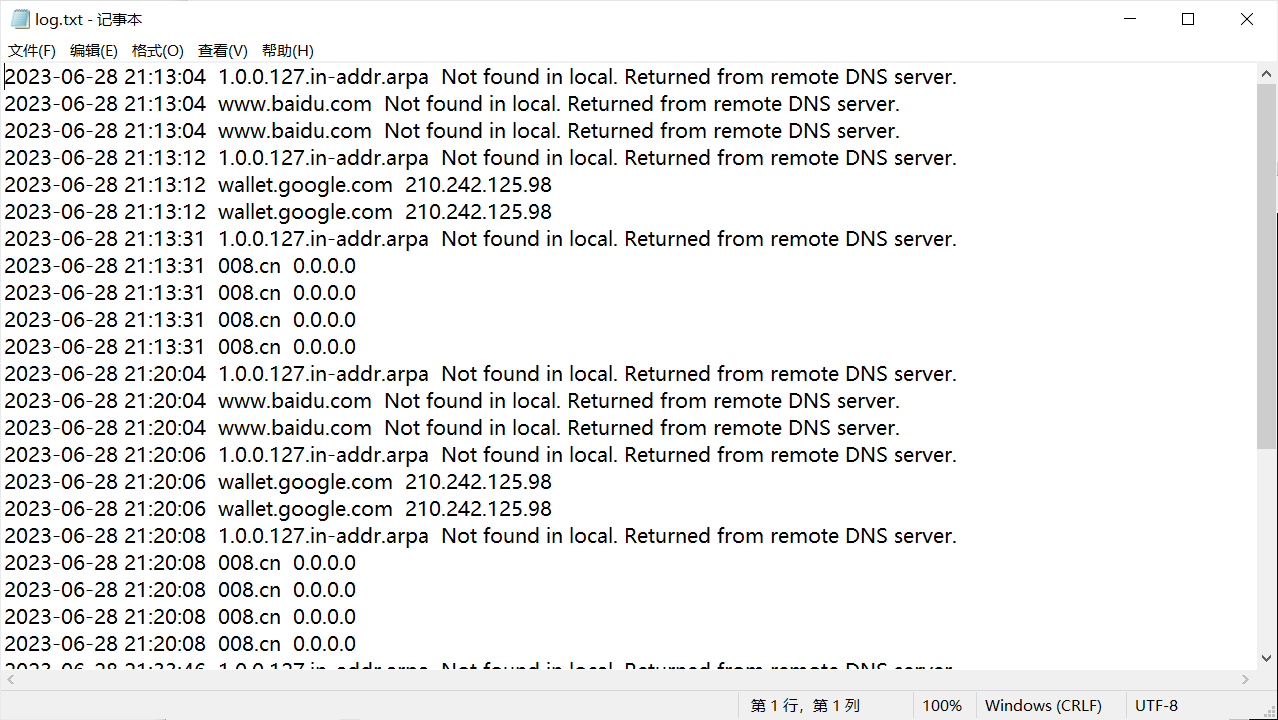
当客户端的查询在本地命中时，可以写入应答的 IP地址（或屏蔽信号0.0.0.0）。若无法命中需要转发到远程服务器查询时，将写入”Not found in local. Returned from remote DNS server.”

figure 12：日志示例

1. 软件流程图

如下图所示：

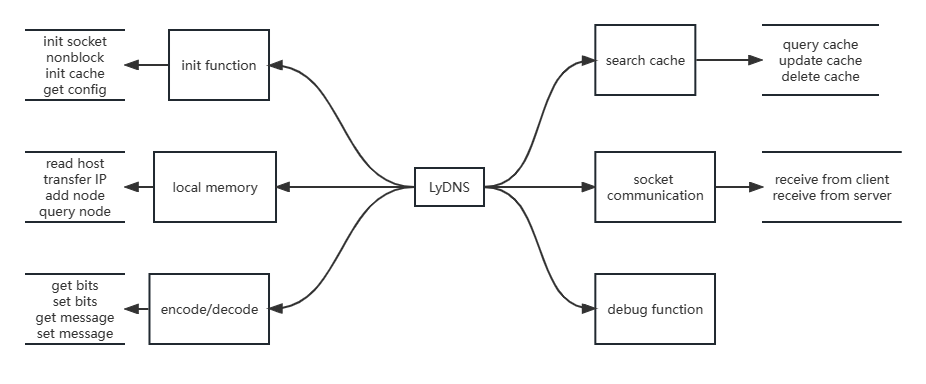


Figure 13 ：软件流程图1

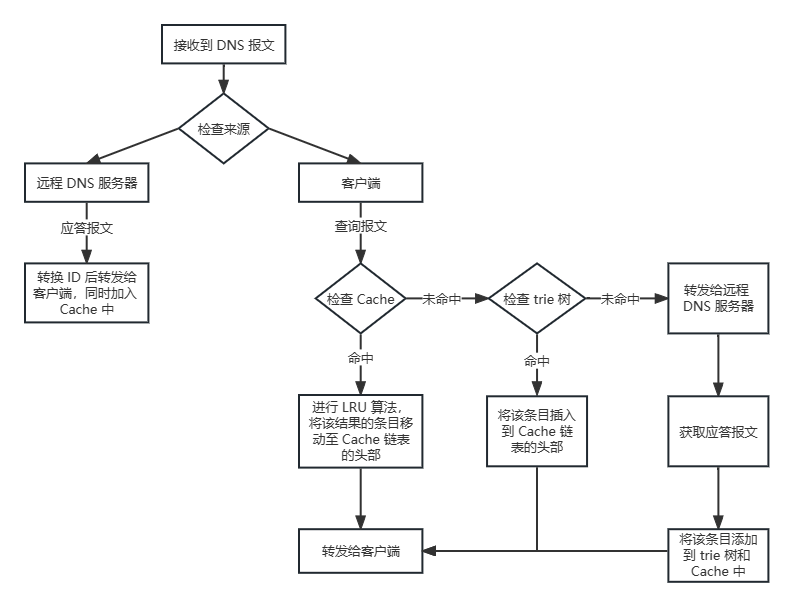


Figure 14：软件流程图 2

1. 测试用例及运行结果

4.1 本地测试



Figure 15：本地测试命令行窗口

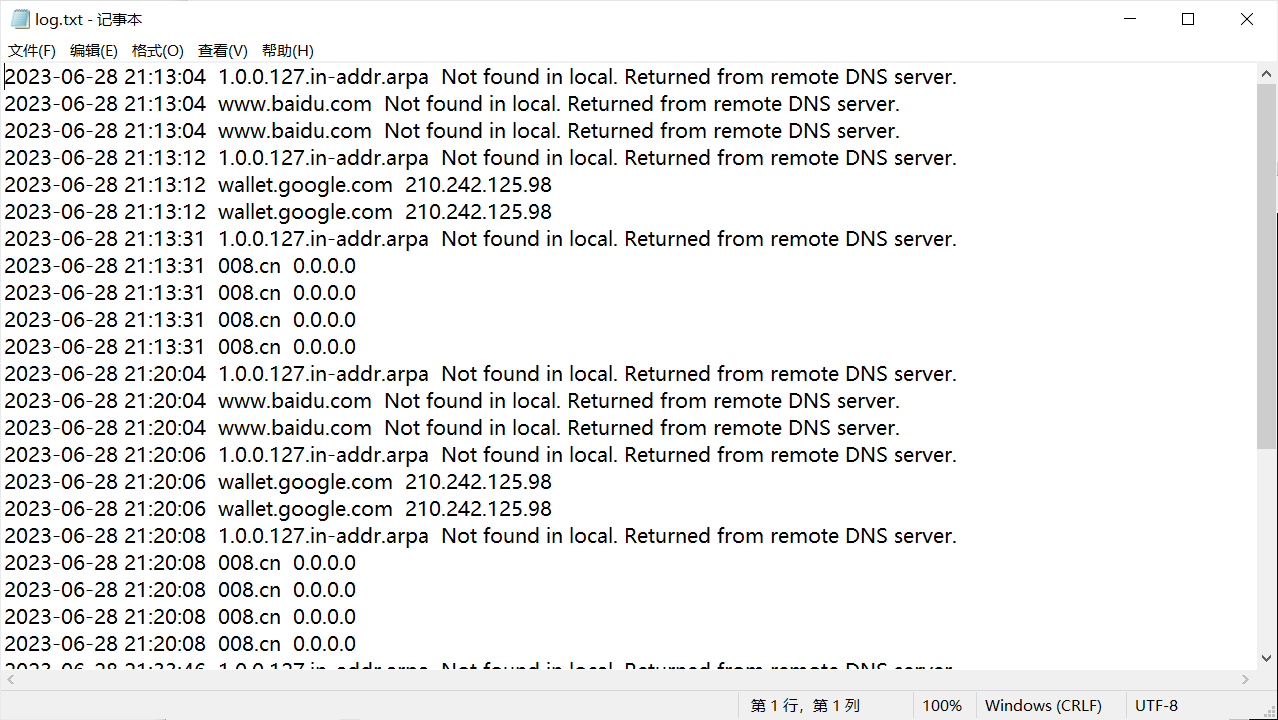


figure 16：本地测试日志文件

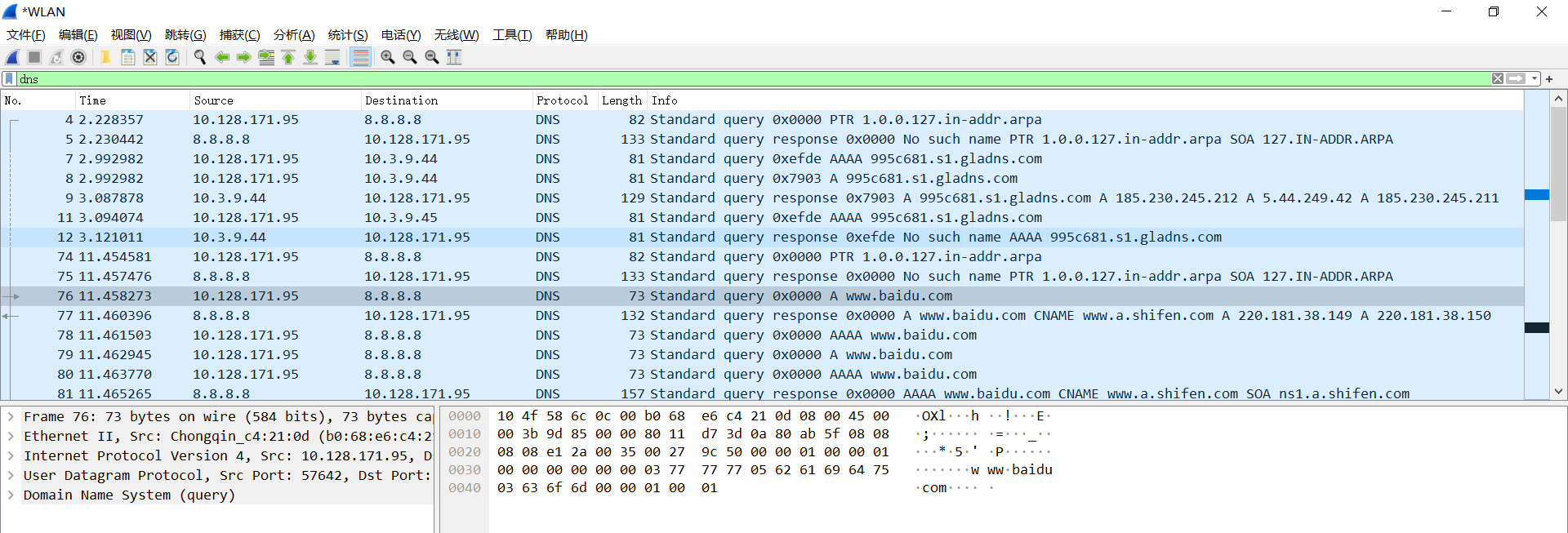


figure 17：Wireshark 抓取的 DNS 包

1. 调试中遇到并解决的问题

5.1 对 IO 模型的思考

在解决多用户并发查询的问题时，我们选用了非阻塞（non-block）以及阻塞（poll）两种方式解决，但我们并不太了解这两种方式各自的优缺点，以及是否有其他更高效的模型方法。

基于此，我们通过查阅《TCP/IP 网络编程》（【韩】尹圣雨）和《Linux 高性能服务器编程》（游双）及其他网络资料，进一步学习了Unix 网络编程以及IO多路复用的三种机制：select，poll，epoll。

根据文献的描述，Unix网络编程中具有五种IO模型，分别是：

* **Blocking IO - 阻塞IO**
* **NoneBlocking IO - 非阻塞IO**
* **IO multiplexing - IO多路复用**
* **signal driven IO - 信号驱动IO**
* **asynchronous IO - 异步IO**

在 LyDNS 中，如果不做任何处理，在实现多用户并发查询时默认的 IO 模型就是 Blocking IO。当用户进程调用了recvfrom这个系统调用，在 IO 的两个阶段（准备数据、拷贝数据）时，用户进程都会进入 block 的状态，整个进程将会被阻塞。一个典型的读操作流程大概如下图：

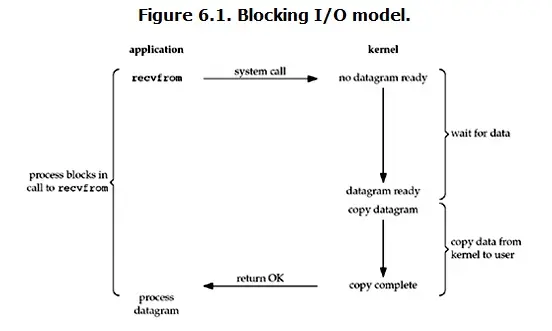


figure 19：Blocking IO

而我们已经通过 nonblock() 函数实现了 NonBlocking IO。当用户进程发出recvfrom这个系统调用后，如果数据还没有被准备好，用户进程并不会等待，它可以每隔一段时间再次发送recvfrom操作。一旦kernel中的数据准备好了，并且又再次收到了用户进程的system call，那么它马上就将数据拷贝到了用户内存，然后返回。所以，用户进程其实是需要不断的主动询问kernel数据好了没有。典型流程如下：

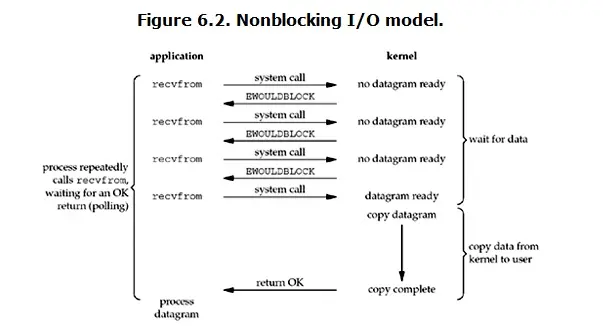


figure 20：NonBlocking IO

NonBlocking IO 的核心代码如下，添加该部分代码即可实现从阻塞态模式到非阻塞态模式的切换：

1. ioctlsocket(clientSock,FIONBIO, (u\_longFAR\*)&nonBlock);
3. ioctlsocket(serverSock,FIONBIO, (u\_longFAR\*)&nonBlock);

接下来将介绍 IO multiplexing 的三种方式。I/O多路复用（multiplexing）的本质是通过一种机制（系统内核缓冲I/O数据），让单个进程可以监视多个文件描述符，一旦某个描述符就绪（一般是读就绪或写就绪），能够通知程序进行相应的读写操作。select、poll 和 epoll 都是 Linux API 提供的 IO 复用方式。而由于本次我们是基于 Window 环境进行开发，所以能使用的 IO 复用方式只有 select 和 poll 两种，epoll 是 Linux 独有的 API。

以 select模型为例：

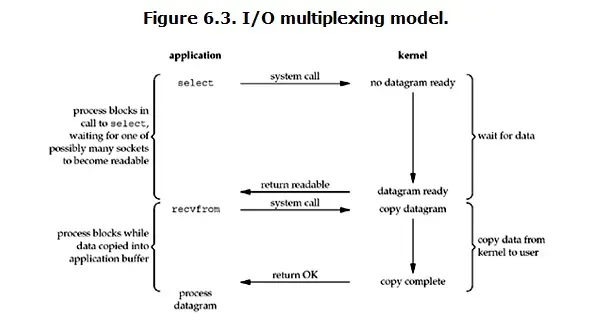


figure 21 ：multiplexing IO（select）

select 函数：

1. **int** select(**int** maxfdp1,fd\_set \*readset,fd\_set \*writeset,fd\_set \*exceptset,**const** **struct** timeval \*timeout);

它与blocking I/O很相似，两个阶段都阻塞。但它与blocking I/O的一个重要区别就是它可以等待多个数据报就绪（datagram ready），即可以处理多个连接。这里的select相当于一个“代理”，调用select以后进程会被select阻塞，这时候在内核空间内select会监听指定的多个datagram (如socket连接)，如果其中任意一个数据就绪了就返回。此时程序再进行数据读取操作，将数据拷贝至当前进程内。由于select可以监听多个socket，我们可以用它来处理多个连接。

在select模型中每个socket一般都设置成non-blocking，虽然等待数据阶段仍然是阻塞状态，但是它是被select调用阻塞的，而不是直接被I/O阻塞的。select底层通过轮询机制来判断每个socket读写是否就绪。

当然select也有一些缺点，比如底层轮询机制会增加开销、支持的文件描述符数量过少等。

而poll的机制与select类似，与select在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是poll没有最大文件描述符数量的限制。

LyDNS 实现 poll 模型的核心代码如下：

1. **typedef** **struct** pollfd {
2. **int** fd;                         // 需要被检测或选择的文件描述符
3. **short** events;                   // 对文件描述符fd上感兴趣的事件
4. **short** revents;                  // 文件描述符fd上当前实际发生的事件
5. } pollfd\_t;
7. **void** poll() {
8. **struct** pollfd fds[2];
10. **while** (1)
11. {
12. fds[0].fd = client\_sock;
13. fds[0].events = POLLIN;
14. fds[1].fd = server\_sock;
15. fds[1].events = POLLIN;
17. **int** ret = WSAPoll(fds, 2, 1);
18. **if** (ret == SOCKET\_ERROR)
19. {
20. printf("ERROR WSAPoll: %d.\n", WSAGetLastError());
21. }
22. **else** **if** (ret > 0)
23. {
24. **if** (fds[0].revents & POLLIN)
25. {
26. receive\_client();
27. }
28. **if** (fds[1].revents & POLLIN)
29. {
30. receive\_server();
31. }
32. }
33. }
34. }

而考虑到在 LyDNS 的实际应用场景中，并不会发生很多的多用户并发查询事件，因此只实现 NonBlocking IO 和 multiplexing IO 中的 poll 模型，从业务处理和资源占用的角度分析是适当且合理的。

5.2 操作系统对 DNS 的缓存

在对 LyDNS 进行调试的过程中，我们发现在 Windows 下短时间内多次 ping 同一个域名时，程序只会收到一次 DNS 查询报文， 据 此 推 测 ，Windows 自 身 也 会 保 留 DNS 缓 存 。

通过网络查证，我们了解到Windows 是带有系统级别的 DNS 缓存的。它由计算机维护，其中包含所有最近访问的网站及其 IP 地址的记录。它充当数据库，保存 DNS 查找的副本，本地存储在浏览器或操作系统上。每当尝试加载网站时，计算机都可以快速引用它。DNS 缓存就像一个电话簿，存储所有公共网站及其 IP 地址的索引。它的主要目的是通过在请求发送到大量公共 DNS 服务器之前处理最近访问的地址的名称解析来加快加载网站的请求。由于信息在本地可用，因此该过程要快得多。

可以在命令行窗口输入 ipconfig /displaydns 以查看计算机系统的 DNS缓存：

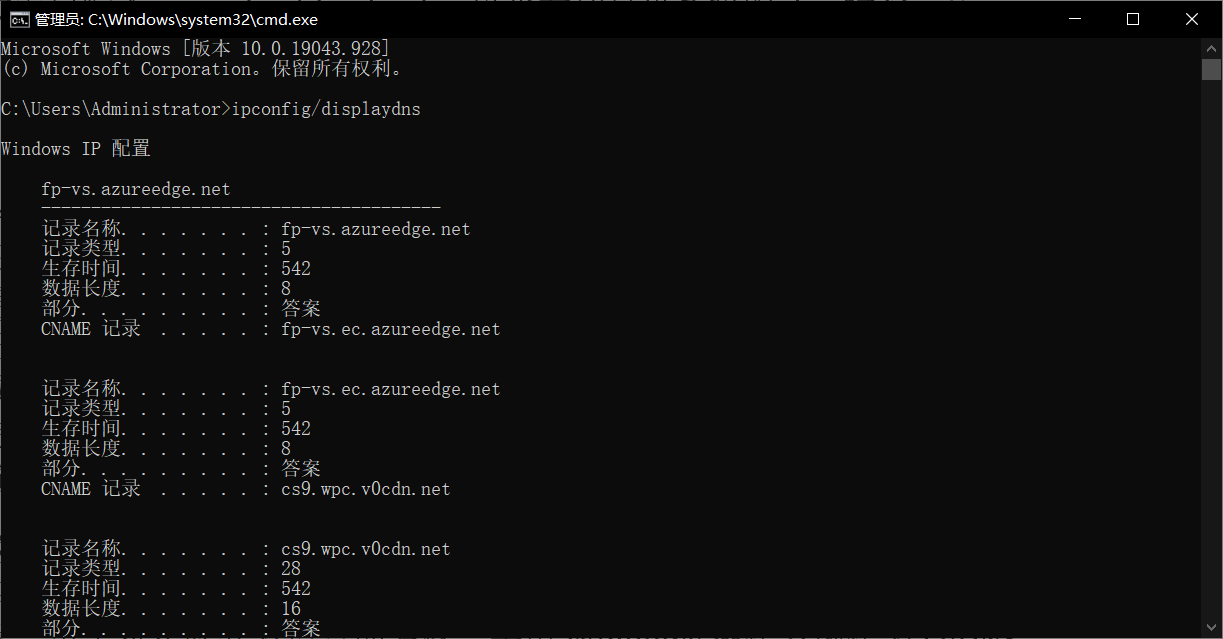


figure 22：系统 DNS 缓存

1. 心得体会

这是一个很有挑战性的课程设计任务，实现这样的一个中继器程序,既要了解 DNS 的工作方式与相关细节,又要依赖 socket 编程进行实现。我们从无到有地完成了一个高性能的 DNS 服务器的设计和实现，从底层原理的角度解答了我们从前的疑惑，同时也打开了 DNS和 Socket 网络编程新世界的大门。

在课程设计的前期阶段，我们通过大量的查阅相关资料学习 DNS 和 Socket 网络编程的基础知识，在 Windows 和 Linux 两个操作系统中都完成了相关的代码编写和简单功能实践。出于对实际开发情景的考察，采用 Windows 操作系统作为 LyDNS 的开发环境。

在实际开发阶段，我们充分发挥双人团队的作用，明确分工，互相激励，以极高的开发效率完成了课程设计。在调试和测试阶段，我们通过使用 Wireshark 抓包工具，加深了对理论课数据报格式的理解，同时也加深了对 Wireshark 这个计算机网络常有的工具的使用熟练度。我们曾尝试过在进行在线游戏、播放在线视频时对网络情况进行抓包分析以试图了解在这过程中的网络质量。这也许是本门课程带给我们的另一个新的收获和思考。

关于 DNS 安全的思考。通过课程设计我们认识到了 DNS 安全是一件非常重要的事情。在查阅了 RFC 文档后，我们了解到 DNS 安全不仅指 DNS 中的数据（即资源记录，RR）安全，还包含在同步或更新 DNS 服务器内容时的传输安全。鉴于 DNS 在 Internet 运行中的重要作用，现代的网络运营商针对其部署了充分的安全机制，称为域名系统安全扩展（DNSSEC）。这些安全扩展不仅提供了端到端的 DNS 安全（即中间解析器不需要是可信的），还减少了中间服务器的计算负担。因此在日常使用 DNS 时并不需要刻意担心 DNS 的安全问题。

综上所述，计算机网络课程设计带给我们的收获远远不止一个 LyDNS，更多的是对未知领域的探索、对团队成员的理解和包容、对网络运行原理和安全的思考，以及对工业代码设计的尝试和改进。课堂带给我们的也远远不止课堂本身，真正的学问和技术永远藏在一次次跃动着敲击键盘的指尖之下。