

YaDNS¹: 支持 DoH 的 DNS 中转服务器

计算机网络课程设计 (2020 年春)

徐逸辰^{1,2}

¹ 北京邮电大学, 计算机学院

²2018213555

linyxus@bupt.edu.cn

目录

第一部分 引言	2
第二部分 系统功能	3
第三部分 系统架构与实现	4
第四部分 开发与测试	21
第五部分 总结与展望	22
第六部分 附录	24

¹YaDNS: Yet another DNS proxy

第一部分 引言

DNS (*Domain Name System*) 也即域名系统,是当今互联网中不可或缺、随处可见、无比重要的一部分. DNS 本质上是一个分布式数据库,能够响应互联网上用户的查找请求,将域名转换为相应的 IP 地址. 域名作为一种方便记忆的标识名称,免去了互联网用户记忆 IP 地址的不便,大大简便了对互联网资源的访问,使互联网蓬勃发展,渗透入个人日常生活的所有角落成为可能. 而 DNS 服务器 (*DNS Server*) 作为 DNS 组成节点,让 DNS 这一纵贯世界、庞大而至关重要的系统运作起来,是互联网体系中不起眼却关键的组成元件.

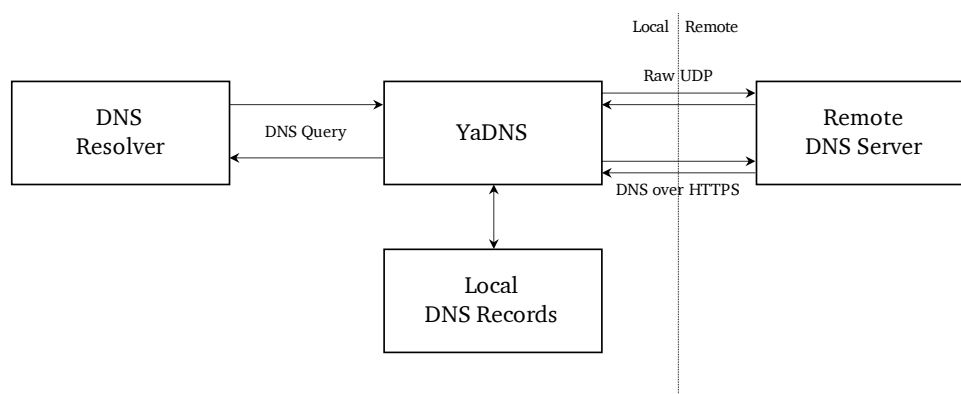


图 1 YaDNS 概览

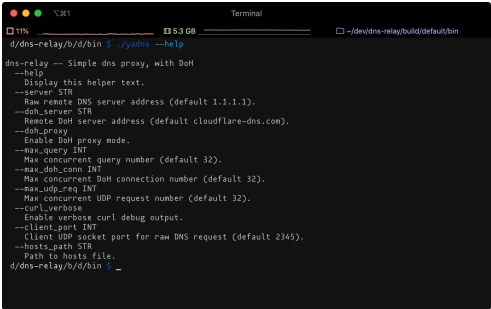
在本次课程设计中,我设计并实现了一个简单、高效的支持 DoH (*DNS over HTTPS*) 的 DNS 中转服务器 YaDNS. YaDNS 能够在 53 端口监听 DNS 查询请求,并转发到远程服务器,实现 DNS 代理功能. 同时,YaDNS 也能够加载域名记录文件,若请求命中了加载的域名记录,则直接返回,借此可以实现对 DNS 污染的反制、去除广告、防御恶意网址等功能.

YaDNS 实现了两种对 DNS 请求的转发方式: 普通 UDP 转发与 DoH 转发. (a) 普通的 UDP 转发将收到的 DNS 请求转发到指定远端服务器的 53 端口上,并将收到的应答返回给用户. YaDNS 使用同一个 UDP 套接字 (*UDP Socket*) 对所有的 UDP 请求进行转发,并通过对 DNS 消息标识符的处理来避免重复与混淆. (b) DoH 转发将 DNS 请求包装在一个 HTTPS 请求中,通过 TLS 连接,发送给远端 DNS 服务器的 443 端口. 通过将 DNS 请求包装在 HTTPS 请求中,DoH 具有更高的安全性. DNS 信息不会被中间人篡改,其他人也无法收集分析用户的域名查询行为. 若开启 DoH 转发模式,则 YaDNS 称为一个安全的 DNS 代理,修改本地 DNS 解析器的默认地址为 YaDNS 地址 (也即本机),则可以将不安全的 UDP 查询封装为安全的 DoH 请求,几乎杜绝了中间人修改、查看的可能.

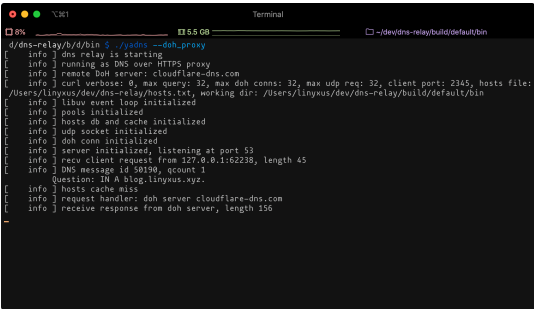
YaDNS 实现了一个基于 **Trie** 树的响应缓存 (*Response Cache*),对于响应中的 **A** 类资源,将会缓存其中的域名与 IP 地址信息. 缓存的生存时间由报文中资源的 TTL 决定.

YaDNS 使用高效的数据结构存储本地加载的域名记录. YaDNS 实现了一个树来存储记录,同时也实现了一个简单而小型的 LRU 缓存 (*LRU Cache*) 与自实现的域名 Hash 算法. 对于经常访问的热点数据,能够大大提高返回速度.

项目中,我使用了成熟高效的构建工具与开发环境提高了开发效率,也利用 **cmocka** 测试库为对 YaDNS 的核心算法逻辑编写单元测试.



(a) 命令行帮助界面



(b) DoH 运行界面

第二部分 系统功能

在本节中,我将对 YaDNS 的主要功能特性进行介绍.

1 命令行用户界面

YaDNS 提供了丰富全面而友好的用户界面. 提供了丰富的命令行选项, 给出 `--help` 标志则会打印出所有选项的帮助信息, 也即

```
./yadns --help
```

YaDNS 在命令行界面中, 提供了修改远程 DNS 服务器地址、修改远端 DoH 服务器接口、修改并行数等功能. 在接下来会分别详细介绍.

2 DNS over HTTPS

DNS over HTTPS 是一种基于 HTTPS 的新型 DNS 查询方式. 传统的 DNS 查询方式很多都基于 UDP 进行明文传输, 不进行加密也没有验证机制. 这导致中间人可以很容易地窃听用户的 DNS 查询行为, 泄露用户隐私. 也能够篡改 DNS 报文内容以进行攻击, 如 DNS 劫持.

HTTPS 是增加了 TLS (*Transport Layer Security*) 的 HTTP 请求. 通过 TLS 规定的密码学方法, 将 HTTP 报文内容加密, 阻止中间人篡改、窃听客户端与服务器之间的 HTTP 流量.

DNS over HTTPS (DoH) 将一次普通的 DNS 查询请求以 HTTPS 请求与响应的方式完成. DoH 客户端将 DNS 报文包裹在 HTTPS 请求中发送给服务器, 而服务器返回的 HTTPS 响应中包含了 DNS 查询结果. 通过利用 HTTPS 的安全性, 保证了原本明文传输的 DNS 请求不被窃听、篡改. 保证了用户的隐私安全与网络安全.

YaDNS 实现了 DoH 代理的功能. 在本地运行 YaDNS, 并给出 `--doh_proxy` 标志, 并可以通过 `--doh_server` 选项指定 DoH 服务器地址 (默认为 `cloudflare-dns.com`), 例如

```
./yadns --doh_proxy --doh_server cloudflare-dns.com
```

则可以在本地启动一个 DoH 代理. 将首选 DNS 服务器地址修改为127.0.0.1, 则在访问互联网式产生的 DNS 查询, 都将被 YaDNS 以 DoH 方式转发给远程服务器. 通过这种方式, 不安全的 DNS 流量被转化为了安全的 DoH 流量, 在启用 DoH Proxy 之后, DNS 污染、DNS 劫持等威胁网络安全的行为, 应当都不会发生了.

3 并发连接池

YaDNS 实现了对请求的处理与转发的并发. 当 YaDNS 接口收到了用户发来的请求之后, 若没有命中本地加载的域名信息记录, 需要转发到远端服务器, 则会将请求存入查询池, 并从请求池 (对于 UDP 转发) 或连接池 (对于 DoH 转发) 获取一个请求或连接处理器, 负责向远端服务器发起请求.

请求的接收与发送都是异步的, 利用libuv²实现了基于事件循环的并发. 可以通过命令行选项--max_query, --max_doh_conn, --max_udp_req来规定查询池、请求池、连接池的大小, 也就规定了最大的并发数量.

4 多叉查找树与 LRU 缓存

YaDNS 支持加载本地的域名记录, 当查询的域名在本地记录中时, YaDNS 将直接返回 DNS 响应而不转发到远端服务器.

在查找本地记录时, YaDNS 首先查找记录是否在 LRU 缓存中. LRU 缓存采用 LRU (Least Recently Used) 换出策略, 查找仅需常量时间; 若没有命中缓存, 则在多叉查找树中进行查找. LRU 缓存与多叉查找树的效率在第 8.3.3 节中给出了详细的分析.

5 基于 Trie 树的响应缓存

YaDNS 支持对收到的远端服务器的响应中的 A 类资源进行缓存. 在 YaDNS 收到查询, 发现查询没有命中本地记录时, 将首先在响应缓存中进行查找, 若查找到记录 (且未过期), 则直接将记录返回. 在收到远端服务器返回的请求时, 则会查看回答与权威节, 若其中的记录为 A 类, 则将域名-IP 地址队存入 Trie 树中.

缓存的资源记录的有效时间由报文中的 TTL 决定. 对于过期域名记录的清理是惰性的, 这意味着只有当一条过期的域名被查询到时, 才会将其删除. 在清除过期信息之后, 会对 Trie 树进行清理, 删除无用节点.

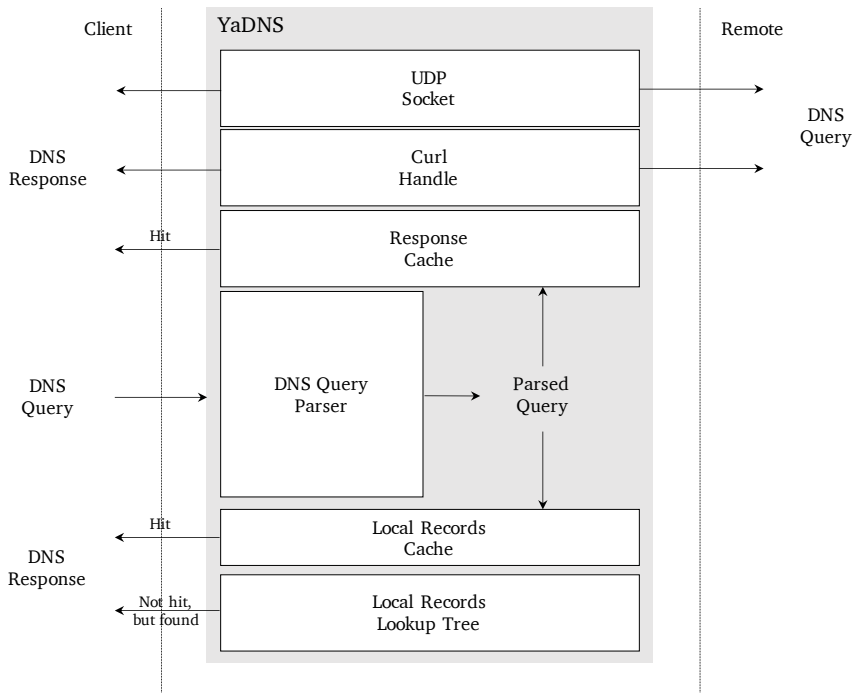


图 2 YaDNS 系统架构

第三部分 系统架构与实现

本节中将介绍 YaDNS 系统的整体架构与各个部分的具体实现. 如图 2 中所示, YaDNS 系统整体可以分为三个模块. 第一个模块为 DNS 查询报文解析, 负责将收到的 DNS 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; 第二个模块为本地记录的命中判断, YaDNS 实现了一个高性能的 LRU 缓存, 与一个多叉查找树. 同时有一个缓存能够对远端服务器发来的资源进行缓存. 若收到的查询请求为 A 型请求, 则会检查请求的域名是否在本地记录中; 第三个模块为 DNS 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. 转发的方式有两种: 基于 UDP 套接字的普通 DNS 查询, 与基于libcurl发起 HTTPS 请求的 DoH 查询. 除此以外, 还有一些工具模块, 如能够配置日志输出级别的日志模块, 以及能够解析命令行参数的参数解析模块. 接下来首先对系统的整体工作方式进行阐述, 随后对各个部分的设计与实现分别进行描述.

6 系统整体运行方式

本节中首先提纲挈领地阐述 YaDNS 运行的主要方式, 在运行过程中使用到的模块. 随后对各个模块进行详细阐述.

程序开始运行后, 首先调用参数解析模块对命令行参数进行解析, 得到系统运行的配置, 并进行相应的设置. 随后初始化 libuv 的事件循环, 对程序使用到的各类资源进行初始

²关于使用libuv而非系统提供的轮询 (poll) 接口来实现并发的原因与考虑, 参见附录 A.

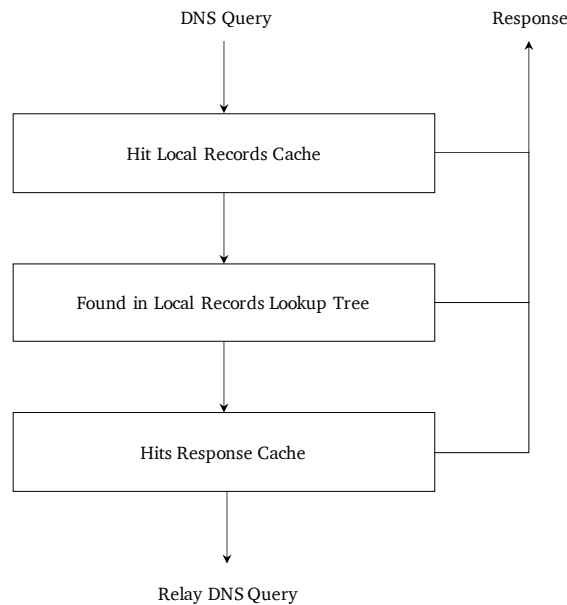


图 3 对收到的查询请求的处理

化,调用本地记录模块加载本地记录信息,系统开始监听本地的 53 端口。

如图 3 中所示,当从 53 端口收到一个 UDP 报文时,首先调用 DNS 查询报文解析模块对报文进行解析,得到查询。若查询的问题是 A 类,则首先调用本地记录模块中的查询函数,查询请求的域名是否在本地记录中。若存在于本地记录中,则直接返回响应报文。若不存在,则再次调用本地记录模块中的查询函数,查询记录是否在响应缓存中,若有也直接返回。若仍然未找到,或查询的问题并非 A 类,则调用 DNS 转发模块对进行进行转发。这涉及到讲查询信息存入查询池中,并获得一个转发的句柄进行转发。

当 YaDNS 收到一个远端服务器发来的报文时,将会通过 DNS 编号(对于 UDP 转发而言)或连接上下文信息(对于 DoH 转发而言)获取与之对应的查询请求,并将其转发回客户端。除此以外,YaDNS 也会解析收到的报文,对其中能够缓存的 A 类资源记录进行缓存。

7 DNS 查询报文解析

DNS 查询报文解析模块根据RFC 1035标准中规定的 DNS 信息格式对收到的 DNS 查询报文进行解析。解析的查询信息将被保存在如下的数据结构中。

```

1 struct dns_msg {
2     char raw[DNS_MSG_MAX_LEN];
3     uint32_t msg_len;
4     dns_msg_header_t header;
5     dns_msg_q_t question[DNS_MSG_MAX_ENTRY];
6 };
7 typedef struct dns_msg dns_msg_t;
  
```

可见,解析完成的报文中保存了原始信息、原始信息长度、报文头部信息、报文中问题节包含的 DNS 查询问题等. 其中,解析的报文头部保存在如下的数据结构中.

```

1 struct dns_msg_header {
2     uint16_t id;
3     uint8_t qr;
4     uint8_t opcode;
5     uint8_t aa;
6     uint8_t tc;
7     uint8_t rd;
8     uint8_t ra;
9     uint8_t rcode;
10    dns_size_t qd_cnt;
11    dns_size_t an_cnt;
12    dns_size_t ns_cnt;
13    dns_size_t ar_cnt;
14 };
15 typedef struct dns_msg_header dns_msg_header_t;

```

而查询问题保存在下面的数据结构中:

```

1 struct dn_label {
2     dns_size_t len;
3     dns_size_t offset;
4 };
5 typedef struct dn_label dn_label_t;
6
7 struct dn_name {
8     dn_label_t labels[DN_MAX_LABEL];
9     dns_size_t len;
10 };
11 typedef struct dn_name dn_name_t;
12
13 struct dns_msg_q {
14     dn_name_t name;
15     uint16_t type;
16     uint16_t class;
17 };
18 typedef struct dns_msg_q dns_msg_q_t;

```

注意到,由于解析的 DNS 请求的数据结构中包含了原始信息,为了节省空间,记录域名的每个标签时,只需要记录偏移量与长度即可.

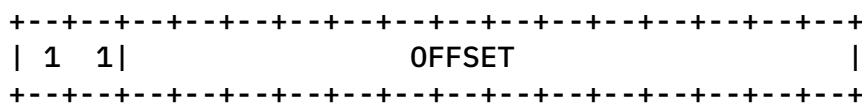


图 4 报文压缩中的域名指针格式

DNS 报文解析模块的实现严格遵守了 RFC 1035 标准中规定的各项要求. 例如在 RFC 1035 标准中的第 4.1.4 节中提出的报文压缩策略也予以考虑实现.

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

8 本地记录

YaDNS 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, YaDNS 不会进行转发, 而是直接将本地记录返回. 本地记录模块分为如下几个部分: 记录文件的解析与加载、DNS 响应报文的生成、LRU 缓存、域名信息查找树.

8.1 记录文件的解析与加载

YaDNS 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. 形式化地,

```
record ::= ip_addr ' ' domain_name
records ::= record | record '\n' records
```

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/db/io.h` 与 `include/db/parse.h` 中. 读取一个文件中的所有记录通过下面的函数完成.

```
1 dn_db_record_t *db_read_all_records(char *path, int *count,
   int *ret_code);
```

这一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.

8.2 生成响应报文

若在本地域名记录中查询到了请求的域名, 则会生成响应报文返回给客户端. 查询到的 IP 地址作为资源记录 (*Resource Record*) 放在报文的回答节中. 生成的报文严格遵守 RFC 1035 中指定的报文格式.

值得一提的是, 在生成响应报文时, 为了缩短报文长度, 使用了 RFC1035 中第 4.1.4 小节提出的报文压缩 (*Message Compression*) 策略. 在返回报文中资源记录的域名部分, 直接使用域名指针指向报文的问题节中查询域名的位置.

域名	IP 地址
sina.cn	66.102.251.24
google.com	46.82.174.69
www.google.com	31.13.71.7
baidu.com	220.101.38.148

表 1 图 5 中多叉树所存储的域名记录

8.3 域名记录的存储结构

YaDNS 使用多叉树来存储读取的域名记录. 除此以外, YaDNS 实现了一个 LRU 缓存, 并且自行设计了适用于域名的哈希函数. 缓存使用哈希函数进行直接映射, 并且采用了两路组相联结构.

8.3.1 多叉树

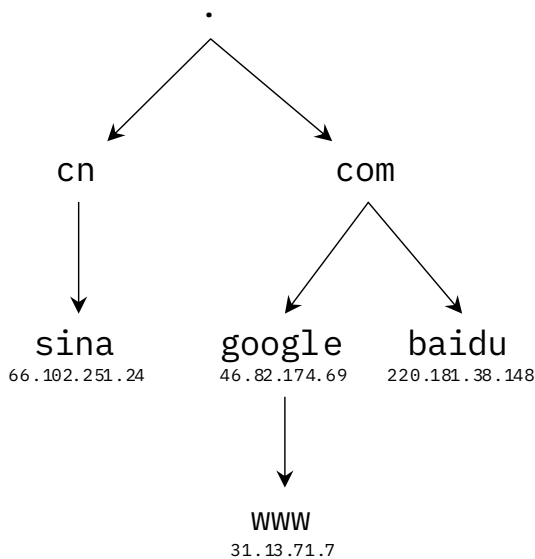


图 5 域名记录多叉树例子

存储域名记录信息的多叉树中, 每个节点都包含域名中的一个标签. 若某个节点存储了一条 IP 地址信息, 则其对应的域名为从这一节点一直到根节点路径上所有标签的拼接. 如图 5 中所示的多叉树存储了表 1 中所示的域名记录信息.

下面形式化地给出域名信息多叉树的具体定义. 为了方便表达, 首先定义列表.

定义 8.1 (列表) 对于某个集合 \mathcal{X} , 记 $\mathbb{L}(\mathcal{X}) = \bigcup_{i \geq 0} \mathcal{X}^i$ 为包含集合 \mathcal{X} 中元素的列表. 令 $|\cdot| : \mathbb{L}(\mathcal{X}) \rightarrow \mathbb{N}$ 为长度函数. 对于 $l \in \mathbb{L}(\mathcal{X})$, 用 $l_k \in \mathcal{X}$ 表示列表中的第 k 个元素. 记 $\mathbb{L}^k(\mathcal{X}) = \{l \in \mathbb{L}(\mathcal{X}) \mid |l| = k\}$ 为长度为 k 的列表.

列表可以用来表示许多需要的数据结构. 如域名可以表示为标签的列表, 而多叉树的子孙节点也可以表示为一个列表. 在实现中, 列表可以用链表也可以用数组来表示.

定义 8.2 (域名记录多叉树) 记 \mathcal{I} 为所有的 IP 地址, \mathcal{L} 为所有的域名标签. 则 $\mathcal{D} \triangleq \mathbb{L}(\mathcal{L})$ 为所有的域名集合. 定义多叉树的节点 $\mathcal{V} \triangleq \mathcal{L} \times \mathcal{I} \cup \emptyset L(\mathcal{V})$. 也即一个节点包含了一个标签、一个 IP 地址和一个子孙列表. 特别的, 根节点 $R = \emptyset \cup \emptyset \cup \mathcal{V}$. 一棵多叉树可以用根节点来索引.

算法 1: TREEINSERT($v, d \in \mathcal{D}, p \in \mathcal{I}$)

```

1 if  $|d| = 0$  then
2    $v_{ip} \leftarrow p$ ;
3 for  $u \in v_{child}$  do
4   if  $d_0 = u_{label}$  then
5     TREEINSERT( $u, d_{1:}, p$ );
6   return;
7  $u \leftarrow (d_0, \emptyset, \emptyset)$ ;
8 LISTAPPEND( $v_{child}, u$ );
9 TREEINSERT( $u, d_{1:}, p$ );

```

算法 1 为多叉树的插入算法. 算法递归地将域名记录插入到多叉树中. 其中 LISTAPPEND 为列表插入算法.

算法 2: TREEBUILD($r \in \mathbb{L}(\mathcal{DI}) \rightarrow \mathcal{V}$)

```

1  $R \leftarrow (\emptyset, \emptyset, \emptyset)$ ;
2 for  $(d, p)$  in  $r$  do
3   TREEINSERT( $R, d, p$ );
4 return  $R$ ;

```

算法 2 定义了多叉树的构造算法. 算法首先构造一个空的根节点, 随后一条条地将节点插入到多叉树中.

算法 3 定义了对多叉树的查找算法. 算法同样是递归定义的. 若在多叉树中查找到域名, 则返回对应的 IP 地址. 若没有查找到域名, 则会返回空.

8.3.2 LRU 缓存

除了多叉查找树之外, YaDNS 也设计并实现了一个 LRU 缓存. 当需要查找域名时, 首先在缓存中查找, 若没有找到, 再进入多叉树中查找, 并且将多叉树中找到的信息存入缓存中. 在缓存中查找域名所需的时间为 $O(1)$.

算法 3: TREELOOKUP($v \in \mathcal{V}, d \in \mathcal{D}$) $\rightarrow \mathcal{I}$

```

1 if  $|d| = 0$  then
2   return  $v_{ip}$ ;
3 for  $u$  in  $v_{child}$  do
4   if  $u_{label} = d_0$  then
5     return TREELOOKUP( $u, d_{1:}$ );
6 return  $\emptyset$ ;

```

YaDNS 中的 LRU 缓存使用自设计的哈希函数进行直接映射, 并且为多路相联, 使用 LRU 策略进行换出. 假设缓存参数为 S 行, 路数为 W . 则缓存数据结构可以形式化地定义如下.

定义 8.3 (LRU 缓存) 定义 $\mathcal{C} = \mathbb{L}^S(\mathbb{L}^W(\mathcal{D} \times \mathcal{I}) \times \mathcal{B})$. 其中 \mathcal{B} 为 LRU 使用统计信息.

具体地, 缓存可以用如下伪代码来表示.

```

1 typedef CacheLine[S] LruCache;
2 typedef struct {
3     CacheEntry entry[W];
4     Bit lru;
5 } CacheLine;
6 typedef struct {
7     DomainName domain;
8     IP ip;
9 } CacheEntry;
```

注意, 由于 YaDNS 中路数 $W = 2$, 因而每一行的 LRU 统计信息只需要用一比特来表示即可. 这在下面会具体描述.

哈希函数设计 LRU 缓存使用一个哈希函数将每一条域名记录直接映射到缓存中的某一位置. 假设缓存的行数为 S 为一个宽度为 w 的无符号整型变量可以表示的值的个数, 也即 $S = 2^w$. 形式化地, 我们需要一个函数 $\theta : \mathcal{D} \rightarrow \{0, 1, \dots, S-1\}$. 为了获得这样的一个函数, 假设已经有了一个标签的哈希函数 $\gamma : \mathcal{L} \rightarrow \{0, 1, \dots, S-1\}$, 则可以定义 θ 为

$$\theta(d) = \square_{k=0}^{|d|-1} \gamma(d_k), \quad (1)$$

其中 \square 为可选的二元运算符. 在 YaDNS 使用的缓存中为 \oplus 也即异或运算符.

接下来考虑对于标签也即一个字符串的哈希函数. 有许多经典而广发的字符串哈希函数, 在 YaDNS 中使用了

$$\gamma(l) = \sum_{i=0}^{|l|-1} l_i \cdot p^i, \quad (2)$$

其中因子 p 在实现中选择为 31.

YaDNS 使用的哈希函数实现了高效的域名映射. 当 $S = 256$ 也即缓存总大小为 512 行时, 在给出的示例记录文件的 909 条记录中, 缓存的利用率为 91.8%.

LRU 换出策略的实现 LRU (Least Recently Used) 是一种广泛使用的缓存换出策略. 无论是在硬件还是软件缓存中, LRU 都可以被用来决定在换出发生时需要换出的行. LRU 需要统计最近最少使用的路来决定换出选择. 具体地, LRU 策略需要维护一个队列. 假设某一缓

存共有 N 路, 分别记为 $0, 1, \dots, N-1$. 则 LRU 需要保存一个装有所有路的标号的队列, 当第 k 路被访问时, 若当前队列状态为

$$q_1, q_2, \dots, q_{a-1}, k, q_{a+1}, \dots, q_{N-1}, \quad (3)$$

则将队列更新为

$$q_1, q_2, \dots, q_{a-1}, q_{a+1}, \dots, q_{N-1}, k. \quad (4)$$

当换出发生时, 直接从队列首部弹出一个标号即为选择换出的路. 并将这个标号装入队尾.

YaDNS 中的缓存实现为两路, 因而对于每一行, 只需要一个比特位就能够完成对 LRU 统计信息的记录, 确定需要换出哪一行. 具体地, 若将两路分别标号为 0 与 1. 则相应的队列状态仅有两种: 0, 1 与 1, 0.

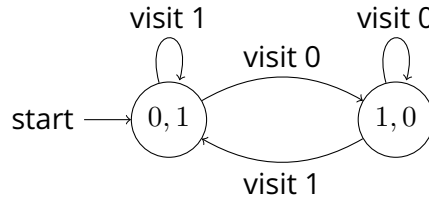


图 6 LRU 队列抽象成的自动机

LRU 队列可以被抽象为图 6 中的有限状态自动机. 由于只有两种状态, 因而使用 1 比特就能够表示. 根据对缓存的访问情况更新状态, 即能够通过 LRU 策略选取换出行.

8.3.3 时间与空间性能分析

接下来, 对多叉树和缓存的效率进行分析.

多叉树的查找效率 为了对多叉树的效率进行分析, 假设域名的最大长度为 K , 而域名每一级可能的标签集合分别为 $\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_K$. 假设查找的域名 d 在多叉树内, 域名的每一级的标签 $d_i \in \mathcal{L}_i, i = 1, 2, \dots, K$, 且每一级的标签服从两两独立的分布, $d_i \sim p_i$ 且 $\forall j, p(d_i | d_j) = p(d_i)$, 则查找时间的期望

$$\mathbb{E}[T] = \sum_{i=1}^K \sum_{d_i \in \mathcal{L}_i} p_i(d_i) \tau | \mathcal{L}_i |, \quad (5)$$

其中 $\tau | \mathcal{L}_i |$ 为在一个大小为 $| \mathcal{L}_i |$, 随机排列的列表中线性查找一个元素所需的平均时间. 这里的分布 p_i 在大多数时候是一个长尾分布.

因而,

$$\begin{aligned}\mathbb{E}[T] &= \sum_{i=1}^K \tau |\mathcal{L}_i| \sum_{d_i \in \mathcal{L}_i} p_i(d_i) \\ &= \sum_{i=1}^K \tau |\mathcal{L}_i|,\end{aligned}\tag{6}$$

也即多叉树的查找时间约为 $O(\sum_{i=1}^K |\mathcal{L}_i|) \approx O(KN)$, 其中 N 为每一级标签个数的平均值.

作为比较, 若直接在一张线性表中查找域名信息, 则需要的时间期望约为

$$\mathbb{E}[T'] = \tau \prod_{i=1}^K |\mathcal{L}_i|,\tag{7}$$

也即查找时间约为 $O(\prod_{i=1}^K |\mathcal{L}_i|) \approx O(N^K)$.

缓存的查找效率 由于采用了哈希函数直接映射的方式, 缓存的查找效率近似于常数时间. 需要的时间大多用于计算哈希函数值. 而计算哈希函数的时间只与标签长度和域名级数有关, 这两个值都可以被认为是常量.

LRU 状态转移法的推广与效率分析 对于一个有 N 路的缓存, 最朴素的方法是为其显式地维护一个队列, 根据访问情况更新队列, 在需要换出是直接读取队首. 而 YaDNS 中使用比特来表示 LRU 信息, 本质上是一种查表法. 推广来说, 对于一个有 N 路的缓存, 其访问状态共有 $N!$ 种情况. 则需要 $\lceil \log_2 N! \rceil$ 位比特来存储状态. 为其构建一张查找表, 则需要 $N! \lceil \log_2 N! \rceil \in O(N! \log_2 N!)$ 的空间. 由 Stirling 近似, 可知这个值近似于 $O(N! \cdot N \log N)$. 而若简单地维护一个队列, 其空间复杂度仅仅位 $O(N \log N)$. 因而这样的查表法并不适用于 N 很大时, 而在 N 很小时, 查表法才能使用近乎等同的空间实现更快的返回速度 (如当 $N = 2$ 时, 也即 YaDNS 的缓存中使用的路大小).

8.4 响应缓存的设计与实现

YaDNS 实现了一个基于 **Trie** 的响应缓存来缓存从远端服务器收到的响应中的 A 类资源记录. 在下面一段时间内, 若再次查询缓存的域名, 则可以直接返回, 无需从远端服务器再次查询信息, 加快了对频繁热点数据的返回时间, 减少了带宽占用, 提升系统整体效率.

首先, 给出 **Trie** 树的定义. 首先给出 **Trie** 树节点的定义.

定义 8.4 (Trie 树节点) 定义 \mathcal{C} 为所有字符的集合. 则对于任意一个集合 \mathcal{X} , 可以定义 $\mathcal{T} \triangleq \mathcal{C} \times \mathbb{L}(\mathcal{T}) \times \mathcal{X}$. 注意这是一个递归定义, 一个最基本的 **Trie** 树节点可以写为 $r = (\emptyset, \emptyset, \emptyset)$, 这一般可以作为初始化一棵 **Trie** 树时的根节点.

可见, 一个 **Trie** 树的节点包含了这个节点所存储的字符、它的子孙节点的列表以及它存储的值. 在定义了 **Trie** 树的节点之后, 就能够用一个根节点来索引一棵 **Trie** 树, 一般

地,一棵 **Trie** 树可以写为 $t = (\emptyset, \mathbf{L}, \emptyset)$, 其中 \mathbf{L} 为根节点的子孙节点的列表. 注意根节点的字符一般定义为空字符, 且在根节点上不存储数据.

Trie 树可以用伪代码表达如下.

```

1 typedef struct {
2     Char key;
3     X value;
4     List[TrieNode] children;
5 } TrieNode[X];

```

其中 \mathbf{X} 为需要存储的值的类型.

接下来定义 **Trie** 树的插入和查找算法.

算法 4: TRIEINSERT($r, s \in \mathbb{L}(\mathcal{C}), x \in \mathcal{X}$)

```

1 ASSERT( $r$ );
2 if  $s$  is empty then
3      $r.value \leftarrow x$ ;
4     return;
5 for  $c \in r.children$  do
6     if  $c.key = s_0$  then
7         TRIEINSERT( $c, s_1, x$ );
8         return;
9  $u \leftarrow (s_0, \emptyset, \emptyset)$ ;
10 LISTAPPEND( $r.children, u$ );
11 TRIEINSERT( $u, s_1, x$ );

```

算法 5: TRIELOOKUP($r, s \in \mathbb{L}(\mathcal{C})$)

```

1 if  $r$  is empty then
2     return  $\emptyset$ ;
3 if  $s$  is empty then
4     return  $r.value$ ;
5 for  $c \in r.children$  do
6     if  $c.key = s_0$  then
7         return TRIELOOKUP( $c, s_1$ );

```

在获得了算法 4 和算法 5 中定义的 **Trie** 树插入与查找算法之后, 可以很容易地设计出一个能够缓存域名与 IP 地址的缓存. 对于报文中的域名, 直接使用 . 分隔的字符串作为其索引, IP 作为数据, 插入一棵 **Trie** 树中即可.

然而, 还需要考虑记录失效与记录的删除问题. 对记录失效的判断与删除的操作是惰性的 (*lazy*), 这意味着只有在查找到某条记录的时候, 才会判断其是否失效, 若失效, 则删除. 而不是隔一段时间就清理到期的记录.

对记录的清理是很容易的,直接将对应节点存储的数据赋值为空即可. 然而,还有一个问题需要解决:在删除记录之后,Trie 树中将存在许多的无用节点. 他们在原先节点的路径上,却不存储数据. 若缓存的某条记录在此后一直没有再次被访问,这些节点将一直处于无用状态,浪费内存. 因而,需要设计对于Trie树的垃圾回收算法.

算法 6: $\text{TRIECLEANUP}(r) \rightarrow \{0, 1\}$

```

1 if  $r$  is empty then
2   return 1;
3  $n \leftarrow 0$ ;
4  $l \leftarrow \emptyset$ ;
5 for  $c \in r.children$  do
6   if  $\text{TRIECLEANUP}(c) = 0$  then
7      $n \leftarrow n + 1$ ;
8      $\text{LISTAPPEND}(l, c)$ ;
9  $r.children \leftarrow l$ ;
10 if  $n = 0$  then
11    $\text{TRIEDEINIT}(r)$ ;
12   return 1;
13 return 0;

```

其中, TRIEDEINIT 能够释放Trie占用的资源. 上面的算法实现了对 Trie树的垃圾回收. 在得到上面的所有算法之后,我们就能使用Trie树很好地管理缓存的资源记录了.

9 DNS 请求转发

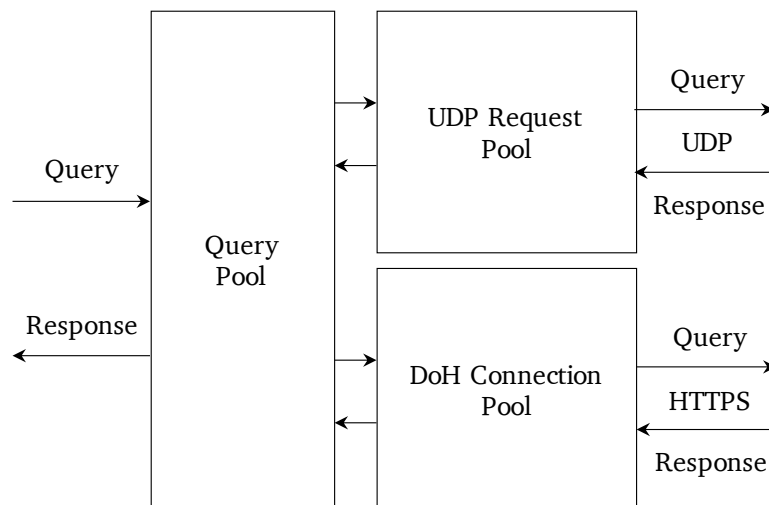


图 7 消息转发系统模块图

当 YaDNS 收到的请求无法在本地记录中找到时,将会转发到远程服务器. YaDNS 支持两种不同的转发方式:简单 UDP 转发与 DoH 转发.

具体地,如图 7 中所示,当有一个需要转发的 DNS 请求报文时,首先将其存入查询池 (*Query Pool*) 中. 然后,根据程序设置,选择从 UDP 池 (*UDP Pool*) 或 DoH 连接池 (*DoH Connection Pool*) 中获取一个传输句柄来与远端服务器进行通信. 下面分别介绍 UDP 与 DoH 转发的具体实现.

9.1 UDP 转发

UDP 转发使用 **libuv** 提供的 UDP 接口来向远端服务器发送 UDP 报文. UDP 转发使用同一个套接字发送所有需要发送的 UDP 报文,并通过对 DNS 编号的处理来分辨不同客户端发来的请求与对应的响应.

具体地,当获得一个需要转发的 DNS 查询时,首先存入查询池,并从 UDP 池获取一个句柄. 句柄保存的上下文数据结构定义如下.

```
1 typedef struct {
2     int query_id;
3     int dns_id;
4     char send_buf[UDP_CONTEXT_BUF_SIZE];
5     size_t send_len;
6     char valid;
7     void *timer;
8 } udp_context_t;
```

可见,句柄保存了与其绑定的查询在查询池中的编号. 且会将 DNS 信息复制一份,以方便对 DNS 报文编号的修改. 它也包含了一个 **valid** 域来指示这个句柄是否处于使用状态,以及一个 **timer** 域包含一个指向 **libuv** 的 **timer** 句柄的指针. 这个 **timer** 会管理请求的超时.

而 UDP 池数据结构的定义如下:

```
1 typedef struct {
2     udp_context_t **pool;
3     size_t pool_size;
4     queue_t *idx_queue;
5     uint16_t next_dns_id;
6 } udp_pool_t;
```

可见,UDP 池保存了 UDP 句柄的列表,存储了池子的大小,也包含一个队列. 队列中存储了池子中空闲的编号,在初始化时,将会将池子中所有的编号按需存入队列中. 当需要分配一个新的 UDP 句柄时,则从这一队列中取出编号,当有一个 UDP 句柄空闲时,则将相应的编号存回队列中.

DNS 编号的分配与处理 除此以外, UDP 池还维护了一个变量来保存为下一个请求分配的 DNS 编号. 当分配新的 UDP 句柄时, 将当前保存的编号分配给这个请求, 并且将 `next_dns_id` 递增. 在初始化时, 这个值被初始化为当前时间.

由于 YaDNS 通过同一个 UDP 套接字转发所有的请求, 因而为每一个转发的请求分配不同的 DNS 编号, 而不是直接使用客户端发来的报文中使用的编号是必须的. 若客户端 A 与 B 几乎同时发来了一个 DNS 查询报文, 其编号刚好都为 x . 若直接将报文转发, 而不对其中的 DNS 编号进行处理, 则在收到远端服务器返回的响应时, 无法分辨两个编号为 x 的响应报文究竟属于哪个客户端. 从而导致错误.

因而在 YaDNS 中, 会将转发的 DNS 报文中的编号修改为分配给相应 UDP 句柄的编号. 由于保证分配给 UDP 句柄的编号在绝大多数时候是独立的, 可以将相应的编号唯一地与一个 UDP 上下文绑定, 也就可以确定响应报文所对应的客户端. 在找到对应的客户端后, 将报文中的编号改为原始编号, 即可以转发给客户端.

UDP 超时 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. 考虑一个 YaDNS 服务器以 p 个每秒的速度收到客户端发来的查询请求, DNS 编号的宽度 $w = 16$, 因而有 2^w 个不同的编号. 在 $\frac{2^w}{p} = \frac{65536}{p}$ 秒后, 分配的 DNS 编号就会发生重复. 需要指出, 在大多数情况下, 因为 UDP 报文的传输速度缓慢而造成编号重复、混淆是不可能发生的. UDP 报文的来回时间一般不会超过 8 秒, 在 8 秒内出现两个相同编号的报文则需要 $p = 8192$. 在这样的较高负载场景下, 基本不可能使用单台机器的单个套接字与远端服务器通信. 尽管如此, 这种情况还是需要在设计时予以考虑.

9.2 DoH 转发

DOH 转发的本质是进行 HTTPS 请求. YaDNS 在实现中使用 `libcurl` 的 `Multi` 接口, 结合 `libuv` 进行非阻塞的并发 HTTPS 请求.

DoH 转发同样有连接池的设计. 连接池的数据结构定义如下.

```

1 typedef struct {
2     CURL *easy_handle;
3     int query_id; // associated query id
4     char read_buf[CONN_CONTEXT_BUF_SIZE];
5     char send_buf[CONN_CONTEXT_BUF_SIZE];
6     size_t send_len;
7     size_t nread;
8     uint16_t dns_id;
9     int conn_id;
10    void *timeout_timer;
11 } conn_context_t;
```

可见,连接池中的连接句柄共保存了下面这些上下文字段:

1. **libcurl**连接句柄;
2. 与连接绑定的查询在查询池中的编号;
3. 从服务器接收的缓冲区;
4. 发送缓冲区;
5. 分配的 DNS 编号;
6. 连接编号;
7. 超时定时器的句柄.

DoH 连接池的设计与 DoH 转发的实现与 UDP 转发是类似的. 一个区别在于 DoH 连接使用了 **Curl** 提供的 HTTPS 请求句柄来发送和接收 DNS 报文. 除此以外,需要指出在 DoH 连接中,重新分配 DNS 编号不是必要的. 原因在于不同的 DNS 查询使用不同的 **Curl**句柄完成,而一个**Curl**句柄完成的 DNS 查询在一次 HTTPS 传输中独立完成,并不存在混淆的情况. 而每个**Curl**句柄也通过连接上下文唯一地与一个客户端请求绑定. 然而,为了保持与 UDP 转发在实现上的一致性,DoH 转发同样实现了对 DNS 编号的分配与处理.

注意,通过连接池的实现,YaDNS 实现了对**Curl**句柄的复用. 这一方面更好地利用了资源,免去了每次转发都需要初始化再销毁**Curl**句柄的繁琐,另一方面,复用**Curl**也意味着复用了 TCP 连接. 在短时间频繁转发请求的情况下,复用 HTTPS 连接 (*Keep Alive*) 能够省去 TCP、TLS 握手的环节,大幅提升响应速度.

10 工具模块

本节将介绍 YaDNS 的工具模块:日志输出与命令行参数解析.

```
d/dns-relay/b/d/bin $ ./yadns
[ debug ] this is debug
[ warning ] this is warning
[ error ] this is error
[ info ] this is info
```

图 8 日志输出模块示例

10.1 日志输出

YaDNS 实现了简单的日志分级输出功能,共分为 4 级:调试信息、警告信息、错误信息、用户信息. 对应了四个日志函数: `logd`, `logw`, `loge`, `logi`. 通过模块提供的 `logging_init` 函数,能够对输出等级进行设置.

```
1 void logging_init(char level);
```

其中 `level` 为等级掩码,有且仅有低 4 位比特有效. 4 位比特从高到低分别对应调试信息、警告信息、错误信息、用户信息,当对应比特为 1 时,输出信息,为 0 时,不输出信息. 如图 8 中给出了日志输出模块的运行示例.

10.2 命令行参数解析

YaDNS 实现了一个简单的命令行参数解析模块,能够解析命令行参数进行系统配置. 参数解析模块的接口设计参考了 **Python** 的 `argparse` 模块. 参数解析模块支持很多不同类型的参数. 如下给出的函数调用添加了一个参数,其标示为 `server`,类型为字符串. 并且给出了描述和默认值.

```
1 ap_add_argument(ap, "server", AP_STR, "Raw remote DNS server  
address (default 114.114.114.114).", "114.114.114.114");
```

在这样设置之后,就可以这样运行程序:

```
./yadns --server your.udp.dns.server
```

在 **Python** 的 `argparse` 模块中,相应的调用为

```
1 parser.add_argument('--server', type=str, help='Raw remote  
DNS server address (default 114.114.114.114).', default='  
114.114.114.114')
```

下面的例子设置了一个类型为开关的参数:

```
1 ap_add_argument(ap, "doh_proxy", AP_STORE_TRUE, "Enable DoH  
proxy mode.", 0);
```

添加的参数标示为 `doh_proxy`,给出了描述,没有默认值. 对于开关类参数而言,默认值即为 0. 可以如下面这样运行程序:

```
./yadns --doh_proxy
```

这样将会设置这个开关.

同时,命令行参数解析模块也能根据给出的描述信息,自动输出帮助信息.当系统运行时只提供了开关 `--help` 时,将打印出帮助信息.下面给出了运行时输出的例子.

```
$ ./yadns --help
dns-relay -- Simple dns proxy, with DoH
--help
    Display this helper text.
--server STR
    Raw remote DNS server address (default 114.114.114.114).
--doh_server STR
    Remote DoH server address (default cloudflare-dns.com).
--doh_proxy
    Enable DoH proxy mode.
--max_query INT
    Max concurrent query number (default 32).
--max_doh_conn INT
    Max concurrent DoH connection number (default 32).
--max_udp_req INT
    Max concurrent UDP request number (default 32).
--curl_verbose
    Enable verbose curl debug output.
--client_port INT
    Client UDP socket port for raw DNS request (default 2345).
--hosts_path STR
    Path to hosts file.
--logging INT
    Logging mask for controlling output verbosity (default 15).
```

第四部分 开发与测试

在开发中,我采用了成熟的开发工具,包括编译工具、编辑工具来加快开发速度,保证代码质量;我也借助`cmocka`库对 YaDNS 的主要算法与数据结构的实现进行了单元测试,保证代码质量,保证算法逻辑的正确性.

11 开发环境与构建环境

软件开发使用的工具与环境如下所示:

1. macOS Catalina 10.15.6
2. fish (friendly interactive shell), version 3.1.2
3. iTerm 2, 3.3.12
4. CMake 3.17.2
5. Ninja 1.10.0
6. Conan 1.28.1
7. CLion 2020.2.1
8. Visual Studio Code 1.48.2

具体地,在开发中我使用的主力集成开发环境为 CLion,CLion 提供了成熟的代码高亮、语法提示、纠错与警告等功能,并提供了许多方便的代码补全行为,也具有方便的快捷键与将为好用的 Vim 模式,为提高开发效率提供了莫大的帮助.

而在构建方面,我使用 CMake 作为构建系统,一方面因为他的成熟与好用,另一方面因为它是与 CLion 最为契合的构建方式.而使用 Ninja 作为构建器,执行 CMake 生成的构建规则.使用 Conan 来进行依赖管理,保证了构建的可移植与可复现.

12 单元测试

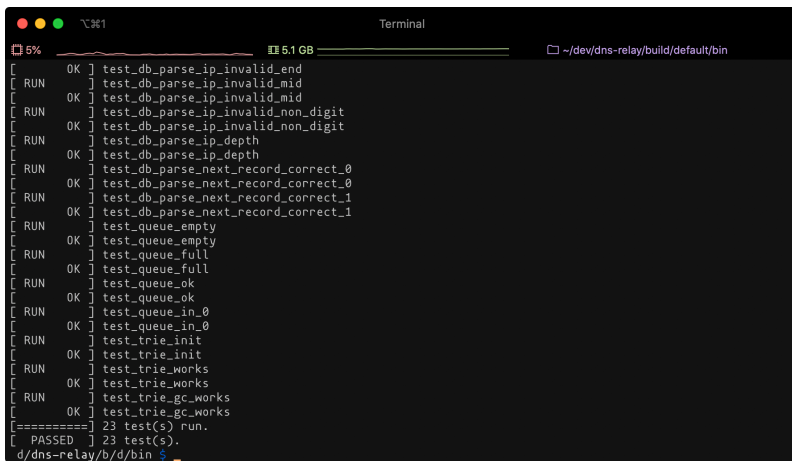
对于 YaDNS 中使用到的主要数据结构以及算法,我借助 `cmocka` 进行了单元测试,来保证其正确性.测试的部分主要包括:对本地记录文件的解析与读取、队列、Trie 树.

本地记录文件的解析 这一部分主要测试了对记录文件中域名、IP 地址的读取与解析.保证了主要函数功能的正确性.除此以外,也对存储记录的数据结构的创建与销毁进行了测试.

队列 在 YaDNS 的查询池、请求池、连接池中都使用到了队列的数据结构来进行池中资源的维护与管理. 单元测试覆盖了队列的入队、出队等主要操作, 以及对队列空、满情况的判断, 对元素是否在队列中的判断等等.

Trie 树 DNS 响应缓存中使用了 Trie 树来将域名映射到对应的资源记录. 单元测试对 Trie 树的插入、查找、销毁以及垃圾回收等功能进行了测试.

YaDNS 对于这三个部分共添加了 23 个单元测试. 在开发过程中, 通过测试可以保证代码质量, 减少显然的错误, 降低调试成本. 大大提升开发效率. 图 9 中是 YaDNS 通过 23 个单元测试的运行截图.

A terminal window titled "Terminal" showing the execution of 23 unit tests. The tests are listed in a column, each preceded by a status indicator (OK or RUN) and a bracket. The tests include: test_db_parse_ip_invalid_end, test_db_parse_ip_invalid_mid, test_db_parse_ip_invalid_mid, test_db_parse_ip_invalid_non_digit, test_db_parse_ip_invalid_non_digit, test_db_parse_ip_depth, test_db_parse_ip_depth, test_db_parse_next_record_correct_0, test_db_parse_next_record_correct_0, test_db_parse_next_record_correct_1, test_db_parse_next_record_correct_1, test_queue_empty, test_queue_empty, test_queue_full, test_queue_full, test_queue_ok, test_queue_ok, test_queue_in_0, test_queue_in_0, test_trie_init, test_trie_init, test_trie_works, test_trie_works, test_trie_gc_works, test_trie_gc_works. At the bottom, it says "==== 23 test(s) run. PASSED 23 test(s)." and the prompt is "d/dns-relay/b/d/bin".

```
5% 5.1 GB ~/dev/dns-relay/build/default/bin
[ OK ] test_db_parse_ip_invalid_end
[ RUN ] test_db_parse_ip_invalid_mid
[ OK ] test_db_parse_ip_invalid_mid
[ RUN ] test_db_parse_ip_invalid_non_digit
[ OK ] test_db_parse_ip_invalid_non_digit
[ RUN ] test_db_parse_ip_depth
[ OK ] test_db_parse_ip_depth
[ RUN ] test_db_parse_next_record_correct_0
[ OK ] test_db_parse_next_record_correct_0
[ RUN ] test_db_parse_next_record_correct_1
[ OK ] test_db_parse_next_record_correct_1
[ RUN ] test_queue_empty
[ OK ] test_queue_empty
[ RUN ] test_queue_full
[ OK ] test_queue_full
[ RUN ] test_queue_ok
[ OK ] test_queue_ok
[ RUN ] test_queue_in_0
[ OK ] test_queue_in_0
[ RUN ] test_trie_init
[ OK ] test_trie_init
[ RUN ] test_trie_works
[ OK ] test_trie_works
[ RUN ] test_trie_gc_works
[ OK ] test_trie_gc_works
[====] 23 test(s) run.
[ PASSED ] 23 test(s).
d/dns-relay/b/d/bin
```

图 9 单元测试运行结果

第五部分 总结与展望

YaDNS 是一个简单的 DNS 中继服务器, 能够转发客户端请求, 实现对远端服务器返回资源的缓存, 也能够加载本地的域名记录. 除此以外, YaDNS 实现了 DoH, 能够作为一个本地的 DoH 代理, 安全化 DNS 流量. YaDNS 采用了成熟的开发框架, 使用了高效的数据结构, 并且经过了测试. 在最后, 我从收获、问题、展望三个角度, 来对本次课程设计进行总结.

收获 通过对 YaDNS 的设计, 我获益颇丰. 一方面, 通过对 DNS 中继服务器的设计与实现, 我更加透彻地了解了 DNS 协议的约定、机制与原理. 在进行开发实现的过程中, 我对其的认识与掌握也越发深刻. 除此以外, 通过对一个应用层网络协议的研究与实现, 我对网络编程的基本方法与注意事项有了更为切实的体会与理解. 通过在实践中真正地、透彻地了解一个网络协议, 我对于计算机网络的认知与掌握更为扎实了. 最后, YaDNS 让我接触到了较为底层的编程, 对 C 语言这样贴近机器底层的语言, 与相应的开发模式与方式都有了真切、全面的体会. 对一个 DNS 服务器的实现, 让我对底层编程的技能掌握地更加透彻与熟稔.

问题 DNS 协议算得上应用层中较为简单的协议。DNS 报文的格式非常直观简洁,而 RFC 1035 标准将 DNS 协议讲解的非常简明透彻,通过对它的阅读,事实上在协议的实现过程中,并没有遇到很大的困难。为我带来最大困难的是 **libuv** 较为复杂的逻辑与对 C 这样底层语言编程经验的不足。一方面,由于 **libuv** 实现的并发模型较为复杂,库函数底层的实现也繁多而难以阅读,导致在出现与其相关的错误时,调试难度较大,常常没有头绪,需要花较多时间。另一方面,由于对 C 语言编程经验的不足,导致时常容易出现低级错误。尤其是内存的分配、释放、管理上,稍不留神,就可能出现内存的反复释放或内存泄漏问题。但通过对于上面两个问题的解决,我也在这些方面积累了许多经验,在成功实现了 YaDNS 的全部功能,攻克这些困难之后,我感到受益匪浅。

展望 在对 YaDNS 的设计与实现中,我攻克了很多难关,也发现了一些不足之处。一方面,我对于底层编程的经验太少,虽然 YaDNS 让我更好地掌握了 C 语言的使用,加深了对网络协议的理解,但在以后的时间中,我也要看重对自己这方面编程能力的培养。另一方面, YaDNS 作为一个简单的中继服务器,也有许多需要改进的地方。例如,由于本地记录往往是固定的,可以对存储本地记录的查找树的每一个节点的子孙节点列表进行排序,在查找时,对子孙使用二分法查找,能够进一步提升效率。除此以外,DoT 也是实现安全的 DNS 流量非常主流的方式,可以增加 DoT 转发方式来进一步使 YaDNS 的功能更为健全。

综上所述,在 YaDNS 课程设计的设计、开发、调试过程中,我遇到了许多困难,也通过解决问题收益匪浅,巩固了知识,加强了技能。回顾过去,展望未来,在接下来的学习中,我应当继续努力,巩固能力,弥补不足,不断前行。

第六部分 附录

附录 A 为什么要使用 libuv ?

YaDNS 直接使用 libuv 实现的事件循环模型而没有自己通过系统调用来实现并发, 主要有下面的一些原因.

开发效率 libuv 提供了一整套基于事件循环的并发接口. 可以很大地提升开发效率. 除此以外, DoH 实现所依赖的 libcurl 也提供的 Multi 接口也已很容易地和 libuv提供的接口相结合, 只需要比较少的新增代码, 就能够将 Curl 句柄结合入 libuv 提供的并发模型中.

可移植性 虽然系统提供的系统调用同样可以实现基于循环的并发. 但不同系统提供的相似功能的调用是不同的. 下表给出了不同系统提供的调用.

操作系统	系统调用
Windows	select
Linux	epoll
macOS (BSD)	kqueue

表 2 不同系统提供的系统调用

因而, 若自己调用这些系统调用来实现并发, 可移植性将大打折扣, 需要将并发循环相关的代码使用不同系统的 API 进行重写, 较为繁荣, 且不具备较大意义. libuv已经为我们做好了对于不同底层 API 的封装, 提供了统一、成熟、较为易用的接口. 因而使用它事实上更加合理.

附录 B 编译与构建

YaDNS 构建依赖于如下工具.

- CMake
- Ninja (Optional)
- Conan

在构建时, 首先创建构建输出文件夹, 并使用 Conan 安装依赖库:

```
mkdir -p build/default
cd build/default
conan install ../..
```

然后, 使用 CMake 生成 Ninja 构建文件:


```
cmake -GNinja ../..
```

最后,使用 Ninja 进行编译:

```
ninja
```

在编译成功后,可以运行单元测试:

```
./bin/ctest
```

而编译好的 YaDNS 可执行文件也在 **./build/default/bin/** 文件夹中.

如果不想使用 Ninja,也可以使用 CMake 默认的 Makefile 系统进行编译:

```
cmake ../..
```

```
make
```