

LAB7实验报告

专业	学号	姓名	开始/结束日期	Email
计算机科学与技术系	191220156	张桓	6.6~6.8	2659428453@qq.com

LAB7实验报告

实验名称

实验目的

实验内容

Task2: DNS server

Step1: Load DNS Records Table

实现逻辑

核心代码

Step 2: Reply Clients' DNS Request

实现逻辑

核心代码

测试结果

Task3: Caching server

Step1: HTTPRequestHandler

整体逻辑

sendHeaders()函数

do_GET()函数

do_HEAD()函数

Step 2: Caching Server

实现逻辑

核心代码

测试结果

手动测试

测试样例

Optional Step: Stream Forwarding

touchItem()函数

实现逻辑

核心代码

do_GET()和do_HEAD()函数

实现逻辑

核心代码

Task 4: Deployment

测试所有

OpenNetLab

总结感想

实验名称

Content Delivery Network

实验目的

本实验将建立一个内容分发网络CDN，CDN是一个地理分布的代理服务器及其数据中心的网络。其目标是通过相对于最终用户的空间分布服务来提供高可用性和性能。为了使得客户端和需要访问的服务器之间的距离缩小，CDN将其内容的缓存版本存在多个地理位置(PoPs)，每个PoP包含许多缓存服务器负责将其中内容传给附近的客户，客户通过DNS服务器，解析目的域名得到最优的缓存服务器的IP地址来实现访问。

本实验的工作就是实现缓存服务器caching server和用于找到最近缓存服务器的DNS服务器，并在OpenNetLab上部署测试。

实验内容

Task2: DNS server

这个任务要实现DNS服务器的功能，DNS服务器对客户端发出的请求做域名解析，实现这一步需要完成两个任务：

- 加载DNS记录表
- 回应客户端的DNS请求

Step1: Load DNS Records Table

实现逻辑

我们需要扫描dns_table.txt文件，这个文件是一行一行的，每一行的结构都是：域名+类型+IP列表，可以通过 `for line in open("dns_table.txt")` 逐行扫描，按照空格分解 `split()` 每一项即可。

为了方便使用，将该文件存成字典类型，字典每一项中，键为域名，值为该项的类型为A或者CNAME，以及对应的IP列表或者新的域名，类型及地址信息可以组成一个列表，因此字典每一项的结构为：

```
<domain:[type, [ip1, ip2...]]>。
```

为了让域名匹配时更方便，如果域名开头为*或者末尾为.则都*直接省略，CNAME表项给出的新域名也同样省略末尾可能出现的。

核心代码

```
1 def parse_dns_file(self, dns_file):
2     for line in open("dns_table.txt"):
3         item = line.split()
4         domain = item[0]
5         if domain[-1] == '.':
6             domain = domain[:-1]
7         if domain[0] == '*':
8             domain = domain[1:]
```

```

9         self._dns_table[domain] = [item[1], []] #<domain : [type, [ip1,
        ip2,...]]>
10         for i in range(2, len(item)):
11             if item[i][-1] == '.':
12                 self._dns_table[domain][1].append(item[i][: -1])
13             else:
14                 self._dns_table[domain][1].append(item[i])

```

Step 2: Reply Clients' DNS Request

实现逻辑

*DNS*服务器接收客户端的请求，对客户端想要请求的*URL*做出解析，其实就是依托上面实现的*DNS*记录表。遍历记录表字典中的键值对，找到能够匹配的哪一项。

- **匹配域名**的实现是通过 `str.find()` 函数，由于记录表中的键值已经将开头可能出现的*和结尾可能出现的.省略了，因此只需要在被请求的*URL*中看看**能不能找到域名对应的字符串**，如果能找到则表示*URL*属于该域名，否则找不到则返回-1，表示不匹配。
- 如果找到*client_ip*匹配的域名后
 - 若类型为*CNAME*，则**直接返回**其映射的新域名，即令 `response_val=value[1][0]`
 - 若类型为*A*，则按照*IP*列表的大小分两种情况：
 - 只有一个*IP*，**直接返回**其作为目的缓存服务器*IP*即可，即 `response_val=value[1][0]`
 - 列表有多个*IP*，那么遍历可选服务器的*IP*列表，找到距离*client_ip***距离最近**的那个*IP*返回

如何找到距离*client_ip*最近的那个缓存服务器？

- 先用 `IP_Utils.getIpLocation(client_ip)` 看看能否得到*client_ip*的经纬度：
 - 若返回*None*表示无法获得客户端所在地址，这时只需在列表中**随机选择一个服务器的IP**返回即可，这里用的是 `random.choice()` 函数
 - 若*client_ip*的经纬度可以得到，同样用 `IP_Utils.getIpLocation()` 函数得到每个服务器*IP*经纬度，计算其到*client_ip*位置的**哈密顿距离**，即 $dis = \sqrt{a^2 + b^2}$ ，其中*a*, *b*分别为两点之间经度和纬度的垂直距离，封装成类函数 `calc_distance()`：

```

1  def calc_distance(self, pointA, pointB):
2      ''' TODO: calculate distance between two points '''
3      locA = IP_Utils.getIpLocation(pointA)
4      locB = IP_Utils.getIpLocation(pointB)
5      return ((locA[0] - locB[0])**2 + (locA[1] - locB[1])**2)** 0.5

```

- 函数 `IP_Utils.getIpLocation(ipstr)` 返回的是 `ip` 位置的经纬度**元组**，所需参数为*IP*的字符串格式。

找到最短的那个 `bestIP` 即可，整体代码如下：

核心代码

```
1  def get_response(self, request_domain_name):
2      response_type, response_val = (None, None)
3      client_ip, _ = self.client_address
4      for key, value in self.table:
5          if request_domain_name.find(key) != -1: #the request domain in table
6              if value[0] == "CNAME":
7                  response_type = "CNAME"
8                  response_val = value[1][0]
9                  break
10             elif value[0] == "A":
11                 response_type = "A"
12                 if len(value[1]) == 1: #only 1 ip in list
13                     response_val = value[1][0]
14                     break
15                 else: #have some ips to get best
16                     if IP_Utils.getIpLocation(client_ip) == None: #the client
17                         ip don't in table, get random
18                         response_val = random.choice(value[1])
19                         break
20                     else: #get best IP in list
21                         bestIP = value[1][0]
22                         bestdis = float('inf')
23                         for ip in value[1]:
24                             curdis = self.calc_distance(client_ip, ip)
25                             if curdis < bestdis:
26                                 bestdis = curdis
27                                 bestIP = ip
28                             response_val = bestIP
29                         break
30             return (response_type, response_val)
```

- 参考文章: [随机选择方法](#)

测试结果

用 `python3 runDNSServer.py` 命令运行编写好的 *DNS* 程序, 再执行测试用例 `python3 test_entry.py dns`, 全部通过:

```
njucs@njucs-VirtualBox:~/networkLab/lab-7-huanhuan6666$ python3 test_entry.py dn
s
2021/06/07-16:50:35| [INFO] DNS server started
test_cname1 (testcases.test_dns.TestDNS) ... ok
test_cname2 (testcases.test_dns.TestDNS) ... ok
test_location1 (testcases.test_dns.TestDNS) ... ok
test_location2 (testcases.test_dns.TestDNS) ... ok
test_non_exist (testcases.test_dns.TestDNS) ... ok

-----
Ran 5 tests in 0.008s

OK
2021/06/07-16:50:36| [INFO] DNS server terminated
```

Task3: Caching server

Caching server 包含一个本地的 *cache table* 表存储远程服务器的数据，它需要处理来自客户端 *client* 的 *http* 请求，收到请求后首先查看本地的 *cache table* 是否存有需要的数据，如果有那就直接向客户端返回，如果没有则需要从远程服务器那里获得数据并存储到缓存表中。主要分为两个部分：

- 对于来自客户端不同 *http* 请求的响应处理
- 本地 *cache table* 的维护及与远程服务器的协作

Step1: HTTPRequestHandler

整体逻辑

这一步实现 *caching server* 对于两种来自客户端的 *http* 请求的回应功能，本实验只涉及两种类型的请求：*GET* 类型和 *HEAD* 类型，对于这两种请求的处理非常类似。再 *caching server* 收到请求后，首先查询请求文件是否在缓存表 *cacheTable* 中，这是通过调用 `self.server.touchItem(self.path)` 来实现的，该函数查询缓存表的文件信息，并且在未查到时从远程服务器请求获取文件信息，定义在下文。

- 如果最终返回 *None*，则表示缓存表和远程服务器都没有该文件信息，返回 *404 Not Found*，通过调用 `send_error(HTTPStatus.NOT_FOUND, "File not found")` 函数实现。
- 如果能够找到该文件的信息，那么将得到返回的文件首部信息 *headers* 以及文件体本身 *body*，*caching server* 就是要根据返回的 *headers* 和 *body* 做出后续处理，对不同请求的回应略有不同：
 - 对于 *GET* 请求需要回应首部信息 *headers* 和文件体 *body*
 - 对于 *HEAD* 请求只需要回应首部信息 *headers* 即可

sendHeaders()函数

经过上面的分析可以看到无论是处理 *GET* 还是 *HEAD* 请求，都需要回应首部信息，利用 `self.touchItem(self.path)` 返回的 *headers*，将其单独抽象成一个 `sendHeaders()` 函数。

- 由于返回的 *headers* 很原始的首部信息，包含很多我们不需要关心的字段，比如 *server, data, connection*，因此先调用 `server.filterHeaders(headers)` 来过滤那些不关心的字段。
- 接下来就是根据过滤后的 *headers* 构建首部信息并发出，这需要通过调用一系列的 *API*：
 - 首先用 `send_response()` 函数将响应首部添加到首部缓冲区，并定好 *code* 类型为 `HTTPStatus.OK`。
 - 接下来用 `send_header()` 函数设置首部中各个字段的信息，由于 *headers* 是 `list [pair(head, value)]`，即(字段, 值)元组的列表，因此遍历列表用 `send_header()` 函数为首部的每个字段填好对应值。
 - 接下来调用 `end_headers()` 函数将上文在首部缓冲区内构建的首部信息发出即可，它其实调用了 `flush_headers()` 函数。

完整的代码如下：

```

1  def sendHeaders(self, headers):
2      ''' Send HTTP headers to client'''
3      # TODO: implement the logic of sending headers
4      headers = self.server._filterHeaders(headers)
5      print("do_GET and get headers:", headers)
6      self.send_response(HTTPStatus.OK)
7      for header, value in headers: #headers is the list of (head, value)
8          self.send_header(header, value)
9      self.end_headers()

```

do_GET()函数

对于`GET`请求的回应为首部信息+文件体本身`body`，因此在调用 `touchItem(self.path)` 得到原始的首部信息和文件体后，先调用 `sendHeaders(headers)` 函数**构造首部并且发出**，再调用 `sendBody(body)` 发出文件体即可。

完整代码如下：

```

1  def do_GET(self):
2      headers, body = self.server.touchItem(self.path)
3      if headers != None:
4          self.sendHeaders(headers) #construct and send the headers
5          self.sendBody(body) #send the body
6      else:
7          self.send_error(HTTPStatus.NOT_FOUND, "File not found")

```

do_HEAD()函数

对于`HEAD`请求的处理与`GET`请求的唯一区别就是**不发送文件体**，只发生首部信息。

完整代码如下：

```

1  def do_HEAD(self):
2      headers, _ = self.server.touchItem(self.path)
3      if headers != None:
4          self.sendHeaders(headers) #send headers
5      else:
6          self.send_error(HTTPStatus.NOT_FOUND, "File not found")

```

Step 2: Caching Server

实现逻辑

`caching server`的主要工作就是向`HttpRequestHandler`提供一个查询缓存表的API, 即 `touchItem(path)` 函数。当需要查询的文件在缓存表时**直接返回**文件信息, 不在时, 需要向远程服务器请求得到此文件信息存放在表中。由于缓存表是个类**字典结构**, 需要查询的文件路径为 `self.path`, 通过`in`即可判断是否在缓存表中:

- 如果在表中的话还需要查看表项是否过期, 通过 `cacheTable.expired(path)` 函数判断即可:
 - 如果过期则需要**请求远程服务器**, 调用 `self.requestMainServer(path)` 得到远程服务器对该文件的回应 `response`。如果 `response` 为`None`, 那么说明根本就没有该文件的信息, 返回`None, None`; 否则需要将该文件的**首部信息和文件体**存放到缓存表中: 通过 `response` 类的 `getheaders()` 函数和 `read()` 函数得到首部信息和文件体, 再用 `cacheTable` 的 `setHeaders()` 函数和 `appendBody()` 函数添加表项。再**返回**首部信息和文件体, 即 `headers, body`。
 - 如果没有过期则**直接返回**首部信息和文件体`headers, body`, 以供上一步的`do_GET()`和`do_HEAD()`函数使用。
- 如果不在表中的话则和表项过期类似, 请求远程服务器获取文件信息并添加表项, 最后返回 `headers, body`信息。

核心代码

```
1 def touchItem(self, path: str):
2     if path in self.cacheTable:
3         self.log_info(f"the file of {path} is in cacheTbale!!!")
4         if self.cacheTable.expired(path): #target expired, ask mainserver
5             response = self.requestMainServer(path)
6             if response == None:
7                 return None, None
8             print(f"update the {path}in cachetable")
9             headers = response.getheaders()
10            rawfile = response.read()
11            self.cacheTable.setHeaders(path, headers)
12            self.cacheTable.appendBody(path, rawfile)
13            return headers, rawfile
14        else:
15            return self.cacheTable.getHeaders(path),
16            self.cacheTable.getBody(path)
17    else:#ask mainserver
18        print(f"the file of {path} isn't in cacheTbale!!!")
19        response = self.requestMainServer(path)
20        if response == None:
21            return None, None
22        print(f"add the {path} in cachetable")
23        headers = response.getheaders()
24        rawfile = response.read()
25        self.cacheTable.setHeaders(path, headers)
26        self.cacheTable.appendBody(path, rawfile)
27        return headers, rawfile
```

测试结果

手动测试

- 开启两个终端，分别键入命令，开启主(远程)服务器和*caching server*

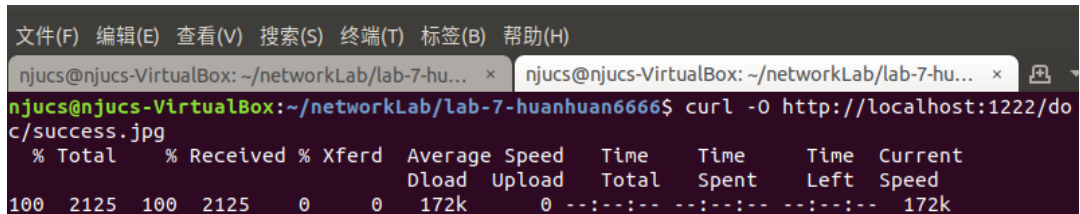
```
1 python3 mainServer/mainServer.py -d mainServer/  
2 python3 runCachingServer.py localhost:8000
```

- 再开启一个终端作为客户端，键入命令表示访问*caching server*下载图片

```
1 curl -O http://localhost:1222/doc/success.jpg
```

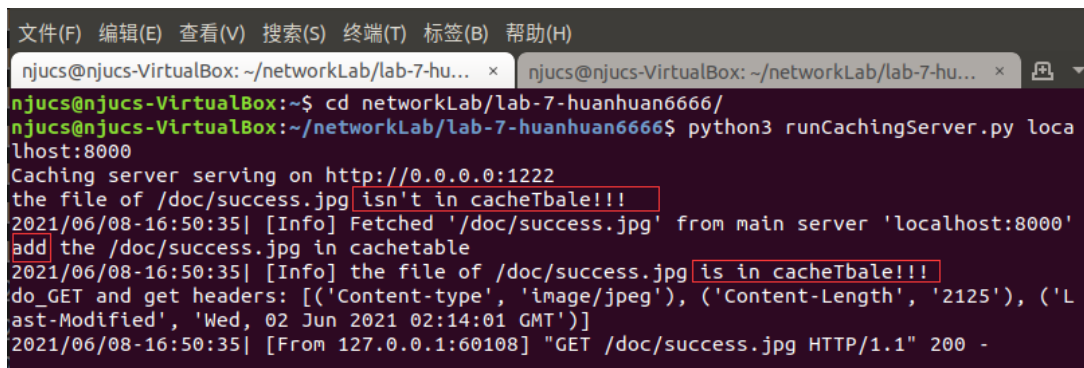
得到结果如下：

- 客户端输出：



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 标签(B) 帮助(H)  
njucs@njucs-VirtualBox: ~/networkLab/lab-7-hu... x njucs@njucs-VirtualBox: ~/networkLab/lab-7-hu... x  
njucs@njucs-VirtualBox:~/networkLab/lab-7-huanhuan6666$ curl -O http://localhost:1222/doc/success.jpg  
 % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current  
                               Dload  Upload  Total   Spent    Left   Speed  
100  2125  100  2125    0     0  172k      0 --:--:-- --:--:-- --:--:-- 172k
```

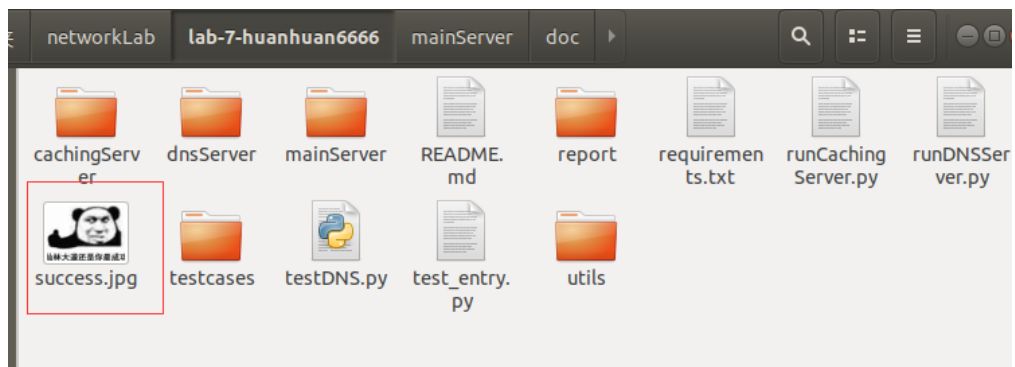
- *caching server*输出



```
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 标签(B) 帮助(H)  
njucs@njucs-VirtualBox: ~/networkLab/lab-7-hu... x njucs@njucs-VirtualBox: ~/networkLab/lab-7-hu... x  
njucs@njucs-VirtualBox:~$ cd networkLab/lab-7-huanhuan6666/  
njucs@njucs-VirtualBox:~/networkLab/lab-7-huanhuan6666$ python3 runCachingServer.py localhost:8000  
Caching server serving on http://0.0.0.0:1222  
the file of /doc/success.jpg isn't in cacheTable!!!  
2021/06/08-16:50:35| [Info] Fetched '/doc/success.jpg' from main server 'localhost:8000'  
add the /doc/success.jpg in cachetable  
2021/06/08-16:50:35| [Info] the file of /doc/success.jpg is in cacheTable!!!  
do_GET and get headers: [('Content-type', 'image/jpeg'), ('Content-Length', '2125'), ('Last-Modified', 'Wed, 02 Jun 2021 02:14:01 GMT')]  
2021/06/08-16:50:35| [From 127.0.0.1:60108] "GET /doc/success.jpg HTTP/1.1" 200 -
```

通过输出信息可以看到，一开始图片信息不在*cacheTable*中，是通过访问远程服务器获取的文件信息。

- 此时图片已经下载到客户端所在文件夹了



- 通过浏览器访问图片文件



测试样例

- 用命令 `python3 test_entry.py cache` 执行写好的`cache`测试用例, 全部通过

```
njucs@njucs-VirtualBox:~/networkLab/lab-7-huanhuan6666$ python3 test_entry.py c
ache
2021/06/08-17:58:33| [INFO] Main server started
2021/06/08-17:58:33| [INFO] RPC server started
2021/06/08-17:58:33| [INFO] Caching server started
test_01_cache_missed_1 (testcases.test_cache.TestCache) ...
[Request time] 89.30 ms
ok
test_02_cache_hit_1 (testcases.test_cache.TestCache) ...
[Request time] 94.31 ms
ok
test_03_cache_missed_2 (testcases.test_cache.TestCache) ...
[Request time] 89.43 ms
ok
test_04_cache_hit_2 (testcases.test_cache.TestCache) ...
[Request time] 2.21 ms
ok
test_05_HEAD (testcases.test_cache.TestCache) ...
[Request time] 50.34 ms
ok
test_06_not_found (testcases.test_cache.TestCache) ...
[Request time] 52.31 ms
ok
-----
Ran 6 tests in 4.699s

OK
2021/06/08-17:58:38| [INFO] Caching server terminated
2021/06/08-17:58:38| [INFO] PRC server terminated
2021/06/08-17:58:38| [INFO] Main server terminated
njucs@njucs-VirtualBox:~/networkLab/lab-7-huanhuan6666$
```

Optional Step: Stream Forwarding

选做部分要完成的东西是在客户端向`caching server`请求一个文件时, 如果`cacheTable`里面没找到, 那么需要向远程服务器请求, 按照上面的实现的话, 我们需要先从远程服务器发送到缓存表, 再从缓存表发送到客户端, 时间反而是原来的2倍。我们想要让这两个过程**同时进行**, 形成一个**数据流**。

`touchItem()`函数

实现逻辑

主要工作在于修改`touchItem()`函数上, 在请求远程服务器时, 不再直接用`read()`函数将全部文件体都读出, 而是用`readinto()`函数读出一部分内容放到缓冲区后, 就立刻将缓冲区的内容发送给客户端。显然用`yield`生成器实现, 每填满一个缓冲区`buffer`就将其`yield`。分为两种情况:

- 当远程服务器也没有该文件时, `yield`一个`None`
- 其余情况均为, 先`yield`文件的首部信息, 然后不断循环`readinto(buffer)`读取文件的部分内容并将该部分内容`yield`, 直到读完整个文件, 即`readinto(buffer)`返回值为0。

核心代码

```
1 def touchItem(self, path: str):
2     # TODO: implement the logic described in doc-string
3     if path in self.cacheTable:
4         self.log_info(f"the file of {path} is in cacheTbale!!!")
5         if self.cacheTable.expired(path): #target expired, ask mainserver
6             response = self.requestMainServer(path)
7             if response is None:
8                 yield None
9                 return
10            print(f"update the {path}in cachetable")
11            the_headers = response.getheaders()
12            self.cacheTable.setHeaders(path, the_headers)
13            yield the_headers
14            buffer = bytearray(BUFFER_SIZE)
15            while True:
16                size = response.readinto(buffer)
17                if size == 0:
18                    break
19                self.cacheTable.appendBody(path, buffer[:size])
20                yield buffer[:size]
21            else: #normal condition
22                yield self.cacheTable.getHeaders(path)
23                yield self.cacheTable.getBody(path)
24            else: #ask mainserver
25                ...#同上述代码6~20行的内容
```

- 参考文章: [yield全面总结](#)

do_GET()和do_HEAD()函数

实现逻辑

这两个函数都是根据`touchItem()`函数的返回值来处理的, 由于返回的是一个迭代器, 令其为`item`, 那么当迭代器的第一个内容`first = next(item)`为`None`时, 就代表远程服务器中也没有文件, 输出404 *Not Found*; 否则说明服务器中有该文件, 那么这时的`first`一定是首部信息, 调用`sendHeaders()`发出即可。`do_HEAD()`函数就此结束。

`do_GET()`函数还需要发送文件体, 只需要不断调用`next(item)`即可将`touchItem()`函数生成的一个个`buffer`大小的部分信息逐个发出, 直到遇到`StopIteration`读完整个迭代器, 说明文件体全部发完了。

```

1  def do_GET(self):
2      item = self.server.touchItem(self.path)
3      first = next(item)
4      if first is None: #a empty
5          self.send_error(HTTPStatus.NOT_FOUND, "File not found")
6          return
7      else:
8          self.sendHeaders(first)#first item is headers
9          while True: #else is body
10             try:
11                 body = next(item)
12                 self.sendBody(body)
13             except StopIteration:
14                 break

```

- `do_HEAD()`函数就是把上述代码的9 – 14行去掉即可

注意：原来的代码在文件中我注释掉了，只留下了选做部分的代码，如果需要检查去掉注释即可。

Task 4: Deployment

测试所有

- 用命令 `python3 test_entry.py all` 同时测试DNS服务器和caching server的用例，全部通过

```

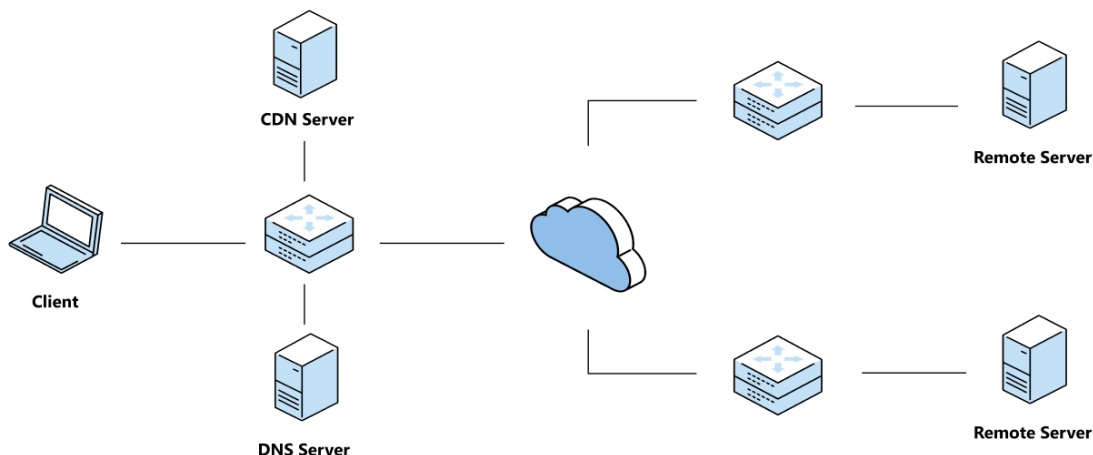
njucs@njucs-VirtualBox:~/networkLab/lab-7-huanhuan6666$ python3 test_entry.py all
2021/06/08-18:04:16| [INFO] DNS server started
2021/06/08-18:04:16| [INFO] Main server started
2021/06/08-18:04:16| [INFO] RPC server started
2021/06/08-18:04:16| [INFO] Caching server started
test_01_cache_missed_1 (testcases.test_all.TestAll) ...
[Request time] 94.44 ms
ok
test_02_cache_hit_1 (testcases.test_all.TestAll) ...
[Request time] 90.25 ms
ok
test_03_not_found (testcases.test_all.TestAll) ...
[Request time] 46.02 ms
ok
-----
Ran 3 tests in 2.297s

OK
2021/06/08-18:04:20| [INFO] DNS server terminated
2021/06/08-18:04:20| [INFO] Caching server terminated
2021/06/08-18:04:20| [INFO] PRC server terminated
2021/06/08-18:04:20| [INFO] Main server terminated
njucs@njucs-VirtualBox:~/networkLab/lab-7-huanhuan6666$

```

OpenNetLab

- 在OpenNetLab上部署测试，网络拓扑如下：



- 上传写好的`dns_server.py`文件和`cachingServer.py`文件后得到`client`，`DNS`服务器和`caching_server`的日志信息。下图为`client`的日志，可以看到第一次`cacheTable`里没有所求信息时，请求完成的时间为`718.80ms`，时间之所以这么长是因为`caching_server`需要请求远程服务器获取文件信息，并将信息存到`cacheTable`中以备后续使用。使得第二次`client`请求同样的文件时可以直接从`cacheTable`中获取信息，时间只需要`2.65ms`。

client视角

```
191220156_client.log x 191220156_c
test_01_cache_missed_1 (testcases.test_all.TestAll) ... ok
test_02_cache_hit_1 (testcases.test_all.TestAll) ... ok
test_03_not_found (testcases.test_all.TestAll) ... ok

-----
Ran 3 tests in 3.171s

OK

[Request time] 718.80 ms 未命中
[Request time] 2.65 ms 命中
[Request time] 707.37 ms
```

caching server视角

```
191220156_client.log x 191220156_cache.log x 191220156_dns.log
2021/06/08-10:10:25 [INFO] Caching server started
Caching server serving on http://0.0.0.0:8176
the cachetable is: {}
the file of /doc/success.jpg isn't in cacheTbale!!!
2021/06/08-10:10:28 [Info] Fetched '/doc/success.jpg' from main server '20.188.122.123:8888'
add the /doc/success.jpg in cachetable
do_GET and get headers: [('Content-type', 'image/jpeg'), ('Content-Length', '2125'), ('Last-Modified', 'Tue, 01 Jun 2021
11:14:49 GMT')]
2021/06/08-10:10:28 [From 10.0.0.24:45172] "GET /doc/success.jpg HTTP/1.1" 200 -
the cachetable is: {'/doc/success.jpg': <cachingServer.cacheTable.HTTPCacheItem object at 0x7f69b4e2bc18>}
2021/06/08-10:10:29 [Info] the file of /doc/success.jpg is in cacheTbale!!!
do_GET and get headers: [('Content-type', 'image/jpeg'), ('Content-Length', '2125'), ('Last-Modified', 'Tue, 01 Jun 2021
11:14:49 GMT')]
2021/06/08-10:10:29 [From 10.0.0.24:45174] "GET /doc/success.jpg HTTP/1.1" 200 -
the cachetable is: {'/doc/success.jpg': <cachingServer.cacheTable.HTTPCacheItem object at 0x7f69b4e2bc18>}
the file of /noneexist isn't in cacheTbale!!!
2021/06/08-10:10:30 [Error] File not found on main server '20.188.122.123:8888'
2021/06/08-10:10:30 [From 10.0.0.24:45176] code 404, message 'File not found'
2021/06/08-10:10:30 [From 10.0.0.24:45176] "GET /noneexist HTTP/1.1" 404 -
```

从`718.80ms`到`2.65ms`时间大概节约了270倍，显然在`CDN`中由于`caching server`的存在，客户端的访问响应变得非常高效。

总结感想

这次实验总体来说不是很难，*CDN*网络中各个部分，比如*DNS*服务器，*cache*的功能还是很好理解的，主要是如何用代码实现。例如*DNS*服务器中如何匹配域名，以及*cache*如何对客户端的请求做出回应。特别是很多*API*的理解，比如这次回应*http*请求时，构造的*headers*是用一系列的函数 `send_response()`，`send_header()`，`end_headers()` 来实现的。一开始我并不理解这三个函数怎么能实现构造和发包的功能，后来查阅手册和资料才知道它们是在一个缓冲区*headers buffer*里添加修改内容达到构造包的效果，`end_headers()` 函数清空缓冲区实现发包的功能。时间大多花在了阅读手册上面，还是很有收获的。

再有这个实验本身是对应用层的一个模拟，之前对于链路层网络层的感受可能不是那么直接，这个实验的*CDN*网络可以说和生活息息相关了。我们作为客户端通过**域名**访问服务器，首先需要通过*DNS*服务器进行**域名解析**，得到离我们**最近的**服务器的*IP*地址。然后用该地址与该*caching server*建立连接，如果*cacheTable*上有我们需要访问的数据就能够直接返回，否则需要向更远程的服务器请求获取信息，最终返回给我们。为了让响应更高效，我们还在选做部分中实现了类似数据流的东西，用一个缓冲区实现从服务器发送到*caching server*和从*caching server*到客户端的并行。

这是从最顶层的视角观察网络，也是最贴近我们的视角。底层支撑的传输层，网络层，链路层的实现细节都屏蔽掉了，也是我们之前所有实验实现过的。用了一学期的时间对整个网络的构成和实现有了一个整体性的了解，无论是各种各样的算法，各种各样的协议，都是为了让网络**更快捷，更高效，更经济地运作**，收获还是非常大的。

计网实验，完结撒花!!! 🌸🌸🌸

最后感谢zjgg们一学期的耐心答疑，辛苦了！有缘再会👋👋