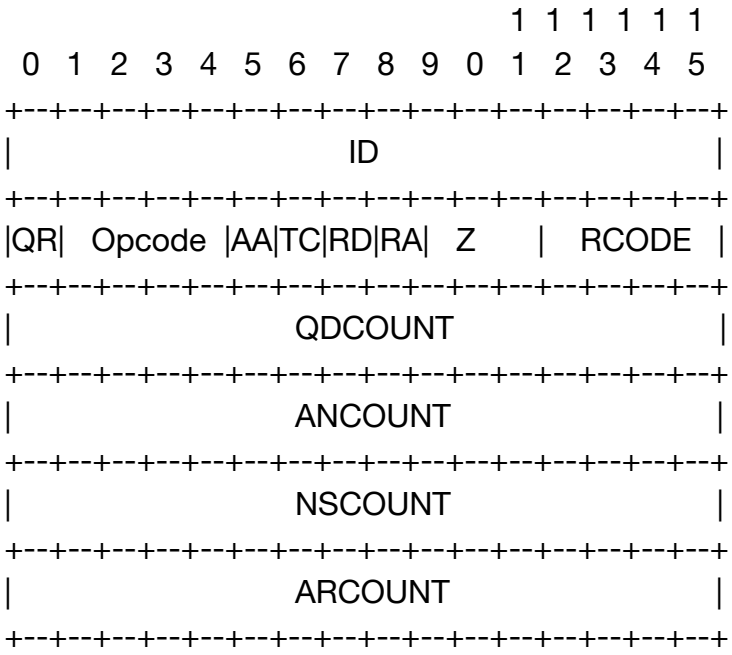
Computer Networks - II

Programming assignment - IV

Saurabh (CS14BTECH11031)

CREATING OWN ‘nslookup’ application

The creation of DNS request involved encoding various values in hexadecimal format and combined in a byte-array. RFC 1034 and RFC 1035 are used to create a DNS query as well as parse a DNS response sent from the server. The following header structure schema defined in RFC 1035 is used :



The above schema defines how a DNS request should be sent to the server. The following code illustrates the addition of each type of field in the DNS request :

```
# adding random header ID
get_request_id = bytearray(encode_hex(1234))[:2]
get_request_id.reverse()
dns_request = get_request_id
# adding flags
flags = bytearray(encode_hex(256))[:2]
flags.reverse()
dns_request += flags
# adding qdcount
var = bytearray(encode_hex(1))[:2]
var.reverse()
dns_request += var
# adding ancount
var = bytearray(encode_hex(0))[:2]
var.reverse()
dns_request += var
# adding nscount
var = bytearray(encode_hex(0))[:2]
var.reverse()
dns_request += var
# adding arcount
var = bytearray(encode_hex(0))[:2]
var.reverse()
dns_request += var
```

Then, for a URL of formal xx.yy.zz, we need to encode 2xx2yy2zz as well as shown :

```
# make the string like 3www5gmail3com if the domain name is www.gmail.com
for i in domain_name.split(".") :
    domain_len_part = bytearray(encode_hex(len(i)))[:1]
    domain_len_part.reverse()
    dns_request += domain_len_part
    dns_request += bytearray(i)
```

And finally, we add QType and QClass :

```
var = bytearray(encode_hex(0))[:1]
var.reverse()
dns_request += var
var = bytearray(encode_hex(1))[:2]
var.reverse()
dns_request += var
var = bytearray(encode_hex(1))[:2]
var.reverse()
dns_request += var
```

Once the request is formed, a UDP socket is used to send this request to the DNS server (192.168.35.52 for IITH).

```
try :
    socket_id.sendall(dns_request)
except Exception as e :
    print e

socket_id.settimeout(timeout)

try :
    get_response = socket_id.recv(512)
except socket.timeout as e :
    print ";; connection timed out; no servers could be reached"
    exit(0)

final = parse_dns_response(get_response)
```

After the request is sent, we wait for the response from the server, and a default timeout of 15 seconds (same as in ‘nslookup’, can change according to command-line argument) is set for the receiving socket. After receiving the response, it’s parsed according to the specifications given in RFC1034. The header comprises of first 12 bytes, followed by domain name, dns type, answer count, non-authoritative as well authoritative answers, etc. The response is parsed and exact output as that of nslookup is printed for a given domain name and the corresponding options (Code snippet not included since it’s too big. Reader can refer to comments in the code).

Following are the code output snippets :

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python nslookup.py www.google.com 192.168.35.52
Server:         192.168.35.52
Address:        192.168.35.52#53

Non-authoritative answer:
Name:   www.google.com
Address: 74.125.200.99
Name:   www.google.com
Address: 74.125.200.106
Name:   www.google.com
Address: 74.125.200.147
Name:   www.google.com
Address: 74.125.200.105
Name:   www.google.com
Address: 74.125.200.104
Name:   www.google.com
Address: 74.125.200.103
```

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python nslookup.py www.google.com 192.168.35.52 -type=NS
Server:         192.168.35.52
Address:        192.168.35.52#53

Authoritative Name Servers:
origin = ns1.google.com
origin = ns2.google.com
origin = ns4.google.com
origin = ns3.google.com
```

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python nslookup.py www.microsoft.com 192.168.35.52
Server:         192.168.35.52
Address:        192.168.35.52#53

Non-authoritative answer:
www.microsoft.com       canonical name = www.microsoft.com-c-2.edgekey.net.
www.microsoft.com-c-2.edgekey.net       canonical name = www.microsoft.com-c-2.edgekey.net.globalredir.akadns.net.
www.microsoft.com-c-2.edgekey.net.globalredir.akadns.net        canonical name = e1863.dspb.akamaiedge.net.
Name:   e1863.dspb.akamaiedge.net
Address: 104.114.70.95
```

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python nslookup.py random-domain 192.168.35.52
Server:         192.168.35.52
Address:        192.168.35.52#53

** server can't find random-domain: NXDOMAIN
```

DNS PROXY/CACHE

In this part, I implement a DNS proxy server which forwards the DNS requests from various applications such as the nslookup application developed in the first part as well the Linux's nslookup command-line utility. This code simply receives the query, forwards it to the local DNS server (127.0.1.1), receives the response from it, performs caching in a dictionary with the (domain-name + dns_query_type) value as the key and the corresponding response received from the server as the key's value. So, if two DNS queries are made like 'nslookup www.google.com 127.0.0.1 -port=5000', then the first query will cause caching and the second query will simply receive the same cached result as was for the first DNS query. Note that if the type of DNS request is changed like in the command 'nslookup www.google.com -type=NS 127.0.0.1 -port=5000' or 'nslookup google.com 127.0.0.1 -port=5000', then these two queries will need to be cached first since the type is different in the first case (NS, not A) and in the second case, the domain name is different (no canonical name in the output of nslookup for this).

The following is the core of the DNS proxy server running on localhost:5000 :

```
while True :
    recv_query, address = socket_id.recvfrom(512)
    dns_type = get_value(recv_query[12 + len_dom(recv_query, 12):12 + len_dom(recv_query, 12) + 2]
    domain_name = parse_url(recv_query, 12)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    if domain_name + "|" + type_dict[dns_type] in cache :
        print "Domain name %s was cached !" % (domain_name)
        get = recv_query[:2] + cache[domain_name + "|" + type_dict[dns_type]][2:]
        socket_id.sendto(get, address)
    else :
        sock.connect(('192.168.35.52', 53))
        sock.sendall(recv_query)
        query_local_dns = sock.recv(512)
        key = domain_name + "|" + type_dict[dns_type]
        cache[key] = query_local_dns
        socket_id.sendto(cache[key], address)
        print "Query for domain %s processed !" % (domain_name)
```

The server runs infinitely and keeps listening for the DNS queries from various applications like nslookup and the nslookup application developed in the first part. I first create a socket to receive the query from the application requesting for DNS, then, I create another socket to forward this query to the local DNS server or the main DNS server. Then, the response is received from the local DNS server, and cached in the

dictionary, ‘cache’, and returned to the client for parsing purposes. Note that in case the query was cached, we retain the header’s ID from the DNS request sent by the application and combine it with rest of the cached message, and send the combined message back to the client for parsing into canonical names and IP addresses.

The following snippet shows the result for various domain names with the types as well, and shows when the query was cached and when it was not :

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python proxy.py
Query for domain www.google.com, type=A processed !
Query for domain www.google.com, type=NS processed !
Query for domain google.com, type=A processed !
Domain name google.com, type=A was cached !
Domain name www.google.com, type=A was cached !
Domain name www.google.com, type=NS was cached !
```

Note that the response is different for google.com than it was for www.google.com since the output of nslookup is different for both the cases. Hence, even after querying for www.google.com, I query again for google.com. Same thing happens when the type is changed from ‘A’ to ‘NS’. In that case, I again store a new response in the cache dictionary.

Studies performed/inferences

I clearly got to see how DNS resolution works under the covers. A lot happens between just entering the URL in the browser and getting the correct webpage for the given URL. I saw how to make DNS requests, how to encode various fields as hexadecimal in a byte-array, sending request to the DNS server through a socket, getting the response, parsing the response and getting the Authoritative (IP’s and Canonical names) as well Non-Authoritative answers (Nameservers corresponding to the given domain name) as well. I also got to know about DNS request types, DNS class, headers in a DNS request as well as response and the overall encoding scheme of DNS messages. I also got to learn about the canonical names (aliases or nicknames for a canonical host name record in a domain name system (DNS) database). Finally, I came to understand the difference between authoritative and non-autoritative answers. An authoritative answer comes from a nameserver that is considered authoritative for the domain which it's returning a record for. It is basically one of the nameservers in

the list for the domain we do a lookup on. On the other hand, a non-authoritative answer comes from anywhere else (a nameserver not in the list for the domain we did a lookup on). I also got to learn the usefulness of caching in proxy servers, about how fast a DNS lookup can be given caching is being done, and, in case of proxy servers, how certain restrictions can be implemented as well. This assignment thus gave an intricate idea of name resolution as well as proxy servers and caching.

For info on how to run the code, see the Readme.txt supplied with the corresponding codes for both parts.