Networks and Distributed Systems
DNS Resolver

Nick Stracke, Johannes-Lucas Löwe

April 2018

1 Introduction

Our DNS server is using the provided Framework and the split-up of tasks suggested in that. As can be seen in 1 our resolver is supposed to use an iterative method of solving the DNS-query if recursion is desired.

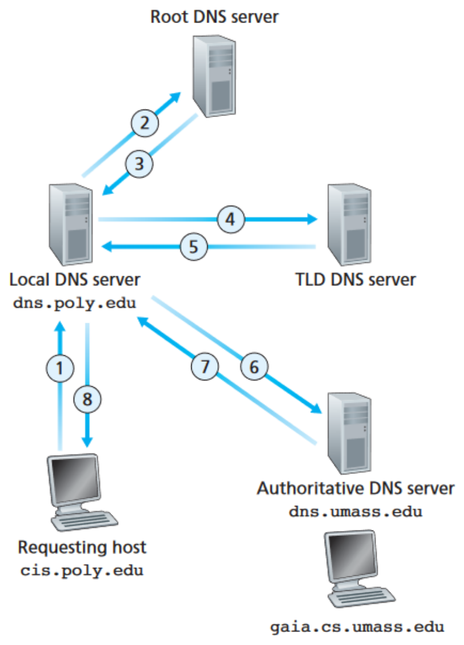

Figure 1: DNS Server Query

2 Design Choices

As the Algorithms are specifications in the RFCs RFC1034 [1], RFC1035 [2] , but certain simplifications were allowed in the class, we made the following design choices:

1. The DNS Resolver will not return answers that go back into its own zone when using the resolver
2. The Cache will be used at the beginning of gethostname - thereby being reused in recursive calls
3. Our zone file will use the virtual [non-existent] domain ourdomain.com
4. We do not support stupid users or domain managers, e.g. who put a CName and an A-Record for the same domain name. Thus in that case always the CName will be resolved.
5. Cache and Zone-Catalogue will passed on through the classes so issues with multiple instances will be avoided
6. the cache will be written at the end of each Request-Handler call, so in case of an breakdown before proper shutdown not all cache is lost

3 Resolver

Our resolver follows the pattern described in the flow-diagram below:

For the ease of use, we have decided, that if no NS is left and we get a record without glue-records, we search for the record starting form the root-domain server. [2]

The scheme of operation is as follows:

We changed the return types for the Resolver from Strings to Resource Records. This was useful since the information that is given by the resource-record format is needed to make a proper response in the server anyway. By doing so a lot of trouble with extensive tuple return-types could be saved. Furthermore we named the variables according to the algorithm specifications. e.g. safetybelt is a list of rootservers, and SNAME the name that we are looking for.

4 Cache

The cache implementation is made out of the Cache class. Saving uses a json encoded version of the record to dict to save the records. Loading loads the records form dict and filters out all old records. This is chosen to allow old records to disappear, since subsequent writing to that cache will overwrite the old file. Thereby we keep only the records needed. Also since the cache may be active for a long time, we check of a record is valid in every lookup, and sort out the invalid ones.

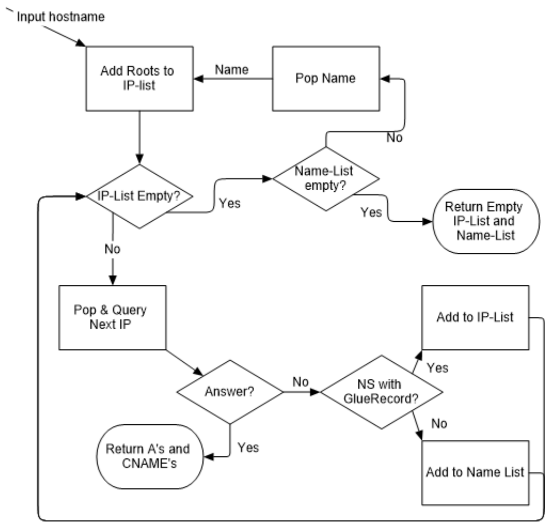

Figure 2: DNS-Resolver in Iterative form as flow-diagram

5 Server

The Server runs an UDP socket to, listen on port 53 and spawns a new Handler for each request received. The Handler then subsequently handles the communication to one specific query. [1] Basically the Server management structure can be seen in the 2 Flow-diagrams below.

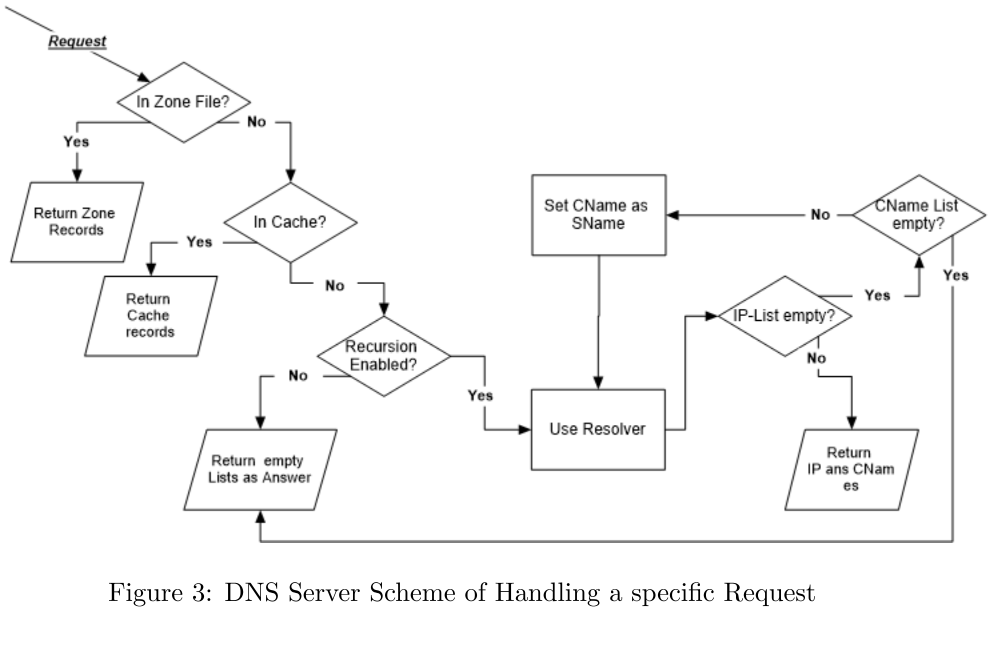

Figure 3: DNS Server Scheme of Handling a specific Request

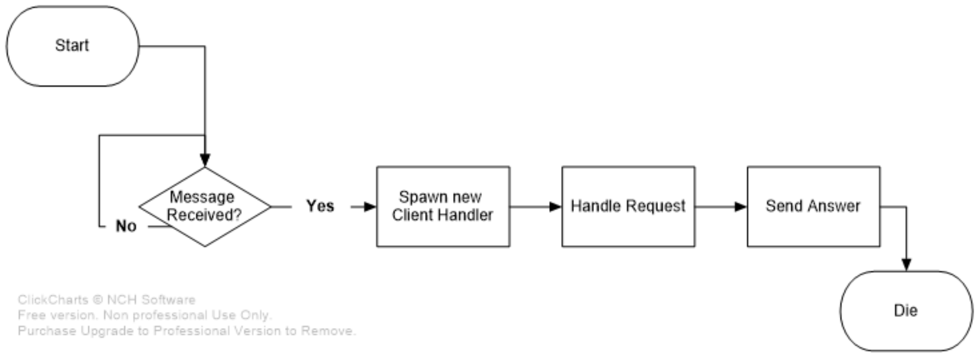
Start
Message Received?
No
Yes
Spawn new Client Handler
Handle Request
Send Answer
Die
ClickCharts © NCH Software
Free version. Non professional Use Only.
Purchase Upgrade to Professional Version to Remove.

Figure 4: DNS Server Scheme of Handling the Clients

6 UDP-Handling

UPD packages must be send and received over port 53. This is a challenge since the sending and especially receiving must be synchronized between the threads in some way. This can be done by wrapping the socket in a central server.

7 Concurrency

For allowing multiple queries to be handed simultaneously, it is needed to be reusing the port 53. Since it is not possible to bind more then one subject at the same port at any given point in time, we wrote a socket wrapper called socketWrapper, which enables this. This is done by keeping track of the conversations in a dictionary which matches the communication ID to the messages received. by asking the method msgThere(id), passing the id, it is possible to get the messages available for the certain id. They will then be removed from the dictionary. By passing the id -1, all question will be returned. Thus allowing the Server to get the questions that have been received. To send the method send adds the data to be send to a Queue, which will then be written whenever the socket is not receiving. This implementation allows for a simple replacement of the socket implementation with the socketWrapper.

8 Testing

In order to automatically proof that our server works, we implemented as required the unit tests. This is done in *dns_tests.py.* in the file our main area of focus lies on the single parts of the system, but also the server as a whole, which is tested in test_server.py.

8.1 Cache

The cache is tested for 3 things:

1. general working: does the cache cache?
2. type security: does it only return the right type of records
3. TTL: does the ttl work

8.2 Zone

The Zone resolving is tested for the following:

1. general working: do we get the right answer?
2. do we get CNames?
3. do we get gluerecords to the NS Resourcerecords?

8.3 Resolver without Cache

The resolver is tested without caching to ensure all queries work according to plan, and host-names can be resolved.

8.4 Resolver with Cache

The Resolver has already been tested, so here we simply try whether it is caching. for that matter we give it a blank cache file, make a request. This should then have been cached. Therefore the same request should be faster the next time.

8.5 Server

In order confirm that all the parts work together nicely and form a working system we test the server as such. In order to do so we test different queries in order to test for all possibilities. In order to do so we run one or two sockets on different ports to the server and let them then connect to the server and pretend to send queries. The answers are then evaluated and compared with the socket-integrated resolution of host-names.

9 Conclusion

After some effort the DNS server now complies with the requirements, as such that it:

1. resloves Names through the resolver
2. Uses caching to keep already resolved records

3. Concurrently allows to handle multiple queries
4. Has interfaces to be started stopped and configured and interacted with

Also test have been written to test:

1. The resolution of names
2. Caching on the server
3. Concurrent queering
4. Zone resolutions

Overall and in short, it can be expressed in the concise sentence: *It works!*

References

[1] P. Mockapetris. *RFC 1034*. 1987. URL: `https://tools.ietf.org/html/rfc1034`.

[2] P. Mockapetris. *RFC 1035*. 1987. URL: `https://tools.ietf.org/html/rfc1035`.