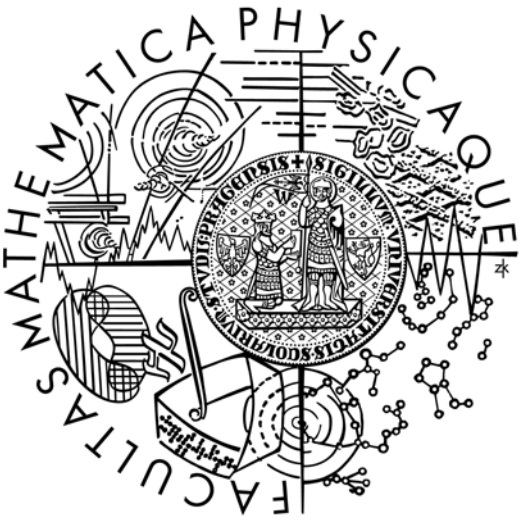
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Ľuboš Slovák

DNS library for a high-performance DNS server

Department of Software Engineering

Supervisor of the master thesis: RNDr. Ing. Jiří Peterka

Study programme: Computer Science

Specialization: Software systems

Prague 2011

I would like to thank my supervisor for his time, suggestions and useful notes. Special thanks go to Ondřej Surý from CZ.NIC Labs for his valuable advices and a lot of support as well as for reading the thesis and commenting on it. I would also like to thank Marek Vavruša and Jan Kadlec from CZ.NIC Labs for cooperation while developing and testing the library and my friend Ján Veselý for reading the thesis and providing useful feedback. Finally I am thankful to my parents, my sister and my close friends for their enormous support and encouragement.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, 5.8.2011 Ľuboš Slovák

Název práce: DNS knihovna pro výkonný DNS server

Autor: Ľuboš Slovák

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Ing. Jiří Peterka, Katedra softwarového inženýrství

Abstrakt: V tejto práci autor navrhol a implementoval vysoko výkonnú knižnicu určenú pre vývoj autoritatívnych DNS serverov. Táto knižnica podporuje všetky základné a niektoré pokročilé funkcie protokolu DNS, ako napríklad EDNS0, DNSSEC či zónové transfery. Má modulárny návrh, je ľahko rozšíriteľná a jednoducho použiteľná. Pre účely testovania a merania výkonu bola zaintegrovaná do existujúcej experimentálnej implementácie DNS serveru. Jej výkon vo väčšine testovaných prípadov prekonával rozšírené implementácie. Práca tiež zahŕňa teoretický úvod do problematiky a podrobnú analýzu úlohy s detailným popisom navrhnutých riešení.

Klíčová slova: DNS, knihovna, autoritativní, server

Title: DNS library for a high-performance DNS server

Author: Ľuboš Slovák

Department / Institute: Department of Software Engineering

Supervisor of the master thesis: RNDr. Ing Jiří Peterka, Department of Software Engineering

Abstract: In this thesis I design and implement a high-performance library for developing authoritative name server software. The library supports all basic as well as several advanced features of the DNS protocol, such as EDNS0, DNSSEC or zone transfers. It is designed to be modular, extensible and easy to use. The library was integrated into an experimental server implementation used for testing and benchmarking. Its performance is evaluated and proved to be superior to prevalent implementations in most cases. The thesis also provides theoretic background and a deep analysis of the task together with detailed description of the implemented solutions.

Keywords: DNS, library, authoritative, server

Contents

1 Preface

While there are several existing implementations of authoritative DNS servers in the world, only a very few of them are used widely. Reasons for this may be various and I will discuss them in this work, while developing my ideas. There are, however, even fewer independent DNS libraries offering functionality needed by authoritative name servers and even those are not well suited for a high-performance server.

In this thesis, I aim at developing an independent library of DNS functions needed by authoritative DNS servers, which will provide clean and convenient API, while maintaining the best possible performance. This library should provide abstraction of all basic parts of the DNS system –domain names, resource records (and the data stored within them), zones and DNS packets. It should also provide API for manipulating these entities. Additionally, it should offer a high-level interface for cornerstone complex functions needed by authoritative servers – such as processing an incoming query and creating a response or processing incoming zone transfer (and updating the corresponding zone).

1.1 Structure of the thesis

The work will start with a brief analysis of requirements of authoritative servers and a comparison of few existing implementations. After stating my exact goals, I will develop main ideas on which the library will be based. Afterwards I advance toward detailed design of particular data structures and functions and outline basic implementation features.

To verify the suitability of the developed library, various tests and benchmarks will be performed. The results will be compared to other existing implementations and evaluated with regard to the stated goals.

2 Introduction

2.1 DNS

Domain Name System is a world-wide distributed database used to map various resources to human-readable domain names. It comprises a set of protocols for storing, retrieving and updating these resources. The most known, and most often used, function of DNS is translating between domain names and IP addresses. When a user's computer wants to retrieve the address stored at a particular domain name, it contacts the nearest *resolver*[1] which asks some *name server* for the information. The request may then be either answered from the server's local information (zone data in case of authoritative server, cache in case of recursive server) or be forwarded to other server.

2.1.1 Authoritative name server

In this work, I will focus on *authoritative name servers*. These serve local zone data (usually loaded from a text zone file) to the clients. The DNS data is hierarchically structured (each label in a domain name corresponds to one level of the hierarchy). It forms a tree and is divided into *zones*. Zone is a administratively separated sub-tree of the domain name tree. An authoritative name server manages one or more zones. One zone may be managed by more servers – these are in such case divided into master and slave servers. Master servers manage the data (which may be changed by administrators or by a mechanism called Dynamic updates[2]) while slave servers fetch the zone data from masters using zone transfers. Via a zone transfer, either the whole zone can be sent (AXFR[3]) or just the changes made to it since the last update (IXFR[4]).

When the authority over some part of the zone is moved to another subject, we talk about zone *delegation*. The point at which the delegated sub-tree starts is called a *delegation point*. Alternatively it may be said that a *zone cut* is made between the apex of the delegated zone and its parent in the domain name hierarchy.

[1] A 'client' in the DNS, sends queries to servers.
[2] Defined in RFC 2136 (23)
[3] RFC 1035 (29) and RFC 5936 (28)
[4] RFC 1995 (22)

2.1.2 Information about DNS

I will not go into more detail about DNS as it would lead me off the main focus of this work. The reader may find many good summaries available on the Web[5]. The main source of knowledge and rules to base the work on are necessarily the Internet Standards[6]. Every name server should be compatible with these standards in order to inter-operate well with other DNS software.

The most fundamental standards related to our work are:

- *RFC 1034* and *RFC 1035* – These two documents define basic concepts of the DNS protocol. While many more recent RFCs update many parts of these documents, the main ideas are still the same and their understanding is essential.
- *RFC 1995* and *RFC 1996* define incremental zone transfers (IXFR) and a mechanism for notification of zone changes (NOTIFY).
- *RFC 2136* defines dynamic updates in DNS.
- *RFC 2671* – Here an extension mechanism (EDNS0) for DNS is defined, which allows the use of larger DNS UDP packets.
- *RFC 4033*, *RFC 4034*, *RFC 4035* – Specify the DNS security extensions (DNSSEC).
- *RFC 5155* - defines a new record for authenticated denial of existence (NSEC3).
- *RFC 5936* obsoletes the previous definition of zone transfer protocol (AXFR) defined in *RFC 1034* and *RFC 1035*.

2.2 Requirements for real-life operation

The main job of an authoritative name server is to serve DNS data for one or more zones by answering queries for domain names lying within these zones. Secondary task is to maintain the zones up-to-date and synchronized between all authoritative servers which are serving them. For these tasks, the server must hold its own copy of

[5] For instance (32)
[6] For list of all DNS-related RFCs, see (30)

the zone data for each zone it is serving, use these data to answer queries and maintain the synchronized state among other servers.

While for the second objective some non-DNS methods may be used (e.g. `rsync` over SSH), the in-band standardized mechanisms (AXFR and IXFR) are preferred for better inter-operability with other DNS software.

Moreover, a server should be able to accomplish these tasks without being restarted, as it is desirable to achieve the best possible accessibility of the service[7]. It must also be able to do these jobs in parallel – to serve as many clients as possible at once, regardless what the requested operation is (a normal query, or a request for zone transfer, etc.).

Besides these fundamental requirements, there are also many other – more or less important – features that an authoritative server may support. Some servers (such as BIND) are designed as very complex software which apart from the authoritative functions may serve also as recursive servers (maybe also with cache) or provide variety of other functions. As my goal is to create a library only for authoritative servers, I will not take these issues into consideration. The authoritative features include:

- Ability to act as a master server (i.e. sending zone transfers and notification about zone changes or receive dynamic updates)
- Ability to act as a slave server (i.e. requesting and receiving zone transfers from master servers)
- Support for DNSSEC
- Support for dynamic updates of zone
- Support for IPv6

From the library point of view, support for IPv6 is of no interest to me, as it is a general server feature, not a DNS-specific one.

[7] Not all current implementations are able to add or remove zones during runtime.

2.2.1 Requirements of TLD[8] servers

Among all domain name servers, the ones serving TLD zones are probably the most specific. A common TLD zone contains hundreds of thousands, or even millions of domain names, most of them being 2nd level domain names, delegated to some other servers. They also receive a very high load of traffic. Due to this, TLD name servers are probably the most demanding with regard to performance. My library should be able to satisfy needs of these name servers by providing high enough throughput of requests.

2.2.2 Withstanding (D)DoS attacks

DoS or DDoS attacks on DNS are quite common and while there are some ways to detect such attacks, there is no perfect way of avoiding them. Therefore, the DNS implementations should be resilient enough to withstand high loads of requests. High performance of the server is the most straightforward way to achieve this, and probably the only way a generic DNS library may be designed to cope with such threats.

2.3 Existing implementations

2.3.1 Authoritative name servers

2.3.1.1 BIND

The Berkeley Internet Name Daemon (BIND) is the most broadly spread name server implementation and also the one supporting the most of (or even all of) DNS-related RFCs. The latter is mostly due the fact that ISC[9] - the main developer of BIND – is also the behind most of these RFCs. BIND is generally used as a test bed for new DNS features. It was also the first name server ever implemented and many parts of the old DNS standards (such as RFC 1034 and RFC 1035) correspond exactly to the way these features were implemented in BIND.

The variety of supported features and functions is probably the strongest point of BIND. On the other hand, it is outweighed by its complexity. It combines the functionality of both authoritative and recursive server provides plenty of features and options to configure. The current version (BIND 9) is thus regarded as a huge

[8] Top Level Domain
[9] Internet Systems Consortium

monolithic and cumbersome piece of software, which is also quite hard to configure and administer. It must be said, however, that the new version being developed (BIND 10) learned from such criticism and its design is highly modular and much cleaner[10].

Its performance (in terms of queries responded per second) is not very resplendent, but this probably never was its authors' goal anyway. There are many faster servers (such as *NSD* or *djbdns*), but none of them can compete with BIND when it comes to the amount of features or functions it provides.

More data about BIND's performance are available in last parts of this work, where I compare benchmark results of my library with results of the most used servers.

2.3.1.2 NSD

NSD (Name Server Daemon) is an implementation from the Dutch research laboratories (NLnet Labs[11]). The main goal of its implementers was to create a very simple piece of software, doing just the necessary work, not supporting such vast variety of features as BIND does and which are of no use to most of the administrators. It was also designed as a very fast – or even high-performance – server, which should definitely surpass BIND in the response rate (more detailed data are also available in the last parts of this thesis).

The first versions of NSD used so-called pre-compiled answers. A zone compiler processed the zone prior to starting the server and prepared all possible answers from the data. While this approach lead to superb performance, it was not suitable when support for DNSSEC or even features like IXFR was added. Latest version, however, retained the zone compilation process that parses all the zones served and creates a database file with all the data optimized for faster loading and responding. One of its drawbacks is that zones cannot be added or removed during runtime mostly because NSD's compiled database contains data from all zones.

2.3.1.3 Other

There are many other authoritative name server implementations I will not take into consideration for various reasons. Some examples are:

[10] For detailed information see the BIND 10 wiki (31)
[11] http://nlnetlabs.nl/

- *djbdns* – Quite popular, but its source code is no longer centrally maintained and lacks any support of DNSSEC.
- *PowerDNS* – uses database back-ends for storing the zone data. Serving data from a database is generally several times slower than using in-memory data structures, so in terms of performance this software would not be a good reference.
- *MaraDNS* – Does not support DNSSEC and no further development is planned.
- *Nominum ANS* – While developed particularly to meet the needs of top-level domains, it is a commercial, closed-source server, so the comparison would not be relevant.

2.3.2 DNS libraries

There are many high-level DNS libraries, such as *Net::DNS*, written in Perl. It is however in fact a complete resolver and lacks any authoritative features at all. *PyDNS* and *PythonDNS* likewise provides resolver functionality for Python, *dnsruby* is a similar library for Ruby and *libbind* for C. I am, however, interested only in those suitable for authoritative servers.

2.3.2.1 ldns

Developed by the NLnet Labs, *ldns* [12] is a comprehensive yet small library of functions, written in C, for developing many different DNS applications. It can be used to create a resolver or to develop a pure authoritative server.

Its main benefits are a very thorough documentation and a straightforward and intuitive design. On the other hand, the implementation itself, though very well-arranged and easy to read, is often cumbersome and could be substantially improved to yield much better performance. It also is quite memory-consuming as the structures are not very optimized in this way either.

2.3.2.2 GoDNS

GoDNS [13] is an experimental library developed by CZ.NIC Labs [14]. The library contains a comprehensive set of functions that can be used by both authoritative and

[12] http://nlnetlabs.nl/projects/ldns/

recursive name servers. Its use is however quite limited to the Go language. Its runtime performance is also questionable due to the used language and strict following of the RFC rules and algorithms. It can be, however, well used as a reference implementation for comparison and testing.

2.4 Goals

As it was already mentioned, the main aim of this thesis is to design and implement a generic, convenient and high-performance library providing all basic functionality required by authoritative name servers. Some of the specific requirements for this library were mentioned in Section 2.2 above. Here is a summary of main features and properties of the library to be designed:

- Abstraction of basic DNS structures and straightforward API for manipulating with them.
- Data structures for holding the zone data, optimized mainly for quick lookup time and response creation.
- Higher-level API for both basic and advanced functionality:
    - Distinguishing query and response types.
    - Full processing of incoming queries.
    - Sending and receiving NOTIFY messages.
    - Sending and receiving full zone transfers (AXFR).
    - Sending and receiving incremental zone transfers (IXFR).
    - Receiving dynamic updates.

High performance of request processing is however the main aim of the library. It should at least match existing implementations, or better, surpass them in this field.

To test the usability of the library as well as to measure its performance capabilities it will be necessary to integrate it into a complete authoritative server. For this purpose I will utilize work done by people of CZ.NIC Labs who are developing an

[13] https://git.nic.cz/redmine/projects/godns/repository
[14] http://labs.nic.cz/

experimental server implementation currently using `ldns` as the main library. This project is not publicly available, but will be well suited for my purpose.

3 Analysis

3.1 Base assumptions

3.1.1 In-memory structures vs. database

In recent time, many name server implementations introduced support for various formats of database back-ends as sources of the zone data. While possibly interesting from the administration point of view, when high performance is the goal, database back-ends are out of question. Even the most sophisticated databases are stored on an external storage (e.g. hard disk), only offering some sort of in-memory cache to speed up lookups. The average performance of lookups in such databases is not remotely comparable with the speed of in-memory data structures. I thus abandon the idea of database back-ends and focus only on in-memory structures in our work.

3.1.2 The cost of operations

The actions performed by an authoritative name server are quite straightforward in nature – look up a name and its associated data in some data structure, put it into the response, look up any other records required in the response and send the response back to the client. DNSSEC does not bring too much hassle neither as it only requires addition of some special records to each answer. It at most complicates the response creation process a bit. Also sending and receiving full zone transfer is rather simple.

Probably the most complex operation is sending incremental zone transfers. The server needs to keep track of changes made to the zone so that information about particular versions of the zone is available for slave servers. Receiving incremental zone transfers and dynamic updates is also a bit tricky as the zone should ideally be updated without affecting the server's ability to response to queries.

However, even though the operations are simple, performing them in the most straightforward way may significantly influence the overall performance. Let's look at a simple example:

Client requests MX record for domain `www.example.com`:

```
dig a.example.com MX
```

A server authoritative for zone `example.com` would respond:

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id:
57052
;; flags: qr aa rd; QUERY: 1, ANSWER: 3, AUTHORITY: 4,
ADDITIONAL: 3
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;example.com.            IN   MX

;; ANSWER SECTION:
example.com.        3600 IN   MX   10 mail.example.com.
example.com.        3600 IN   MX   20 mail2.example.com.
example.com.        3600 IN   MX   50 mail3.example.com.

;; AUTHORITY SECTION:
example.com.        3600 IN   NS   c.a.example.com.
example.com.        3600 IN   NS   ns.example.com.
example.com.        3600 IN   NS   ns.somewhere.com.
example.com.        3600 IN   NS   ns2.example.com.

;; ADDITIONAL SECTION:
mail.example.com.        3600 IN   A    10.0.0.3
ns.example.com.          3600 IN   A    10.0.0.2
ns2.example.com.         3600 IN   A    10.0.0.6
```

If the answering algorithm worked in the most straightforward way, it would compose the response more or less like this:

1. Find zone in which to search for `a.example.com`.
2. Look up name `a.example.com` in the zone `example.com`.
3. Name was found and contains several (exactly three) MX records. Put all the records to the Answer section.
4. Put authority records for the `example.com` zone to the Authority section (here we assume that the structures are clever enough to keep the zone apex accessible without further search).
5. For each from the MX records in the Answer section (3 records):

a. Find the domain name of the name server in the zone.

b. Put the A record from the found node to the Additional section.

6. For each from the NS records in the Authority section (4 records):

    a. Find the domain name of the name server in the zone.

    b. Put the A record from the found node to the Additional section. (One of the names does not belong to the zone.)

You can see that for answering a very simple query for one domain name, several (in this case exactly six) searches for domain names must have been performed. In larger zones (e.g. with more NS or MX records) this number might be even higher. In large zone (hundreds of thousands of domain names) search may be a quite expensive operation.

It is quite obvious that a lot of processing time may be saved if the number of domain lookups for one query could be reduced. In section 3.2.1 I will show that the above query may be answered with only one lookup.

3.1.3 Non-stop operation

Until the introduction of incremental zone transfers and dynamic DNS, the data served by authoritative name servers were static and changed only when the whole zone was reloaded. It was thus not very difficult to achieve good responsiveness. These features, however, complicated the operation of authoritative servers which now must deal with a lot of changes to the zone data that may come at any time. My goal is to provide such structures and functions that will allow the server to always answer queries, even when updating zone data.

When designing an algorithm to solve this issue, one must realize several facts:

- Every update will take some time and the server may respond with the old data during this time. It is not required that the server stops serving the old data the moment the update information arrive.
- The updated data may start being served only once they are stored on a permanent storage (i.e. hard disk) so that if the server crashes after already

using the new data the data will be available for reconstruction.[15] If this was not the case it could happen that the server starts serving new data, crashes (or is shut down), is restarted, and once again serves the old data.

- The update (coming in as either an incremental transfer or a dynamic update) may be applied to the zone only as a whole set. Partial updates are not acceptable.

The proposed solutions are presented in section 3.4.

3.1.4 Reads vs. writes

A very important aspect to consider, mostly for choosing the right data structures, is the ratio of reads and writes that occur during operation of an authoritative name server. It is quite obvious that the vast majority of performed operations are reads (lookups invoked by incoming queries). The only writes are zone transfers and dynamic updates. Also, when considering one separate zone, only reads may take advantage of concurrency to achieve better performance. Updating must be done in a serialized manner.

These are two key observations: the internal structures must provide *quick reads* (ideally concurrent and scalable) and *may not support concurrent writes* (in fact, the writes are not even very performance-requiring).

3.1.5 Threads and synchronization

Although the library itself does not have to directly care about the server model (threaded, event-driven or a combination), one must take the possibility of thread usage into account. Threads require some synchronization mechanism on the shared data structures. If a name server used threads, it would probably parallelize the response process, so that several readers access the data simultaneously. As mentioned in the previous section, concurrent writes are not needed in the context of an authoritative server. The possibility of parallel access to different zones may be omitted, as this creates no race conditions. It would be wise to design the library in such a way that concurrent reads are possible.

[15] See (28), chapter 6

This may be ensured in several ways. From one point of view, some sort of read-write locking seems appropriate for these purposes. However, ideally it should be ensured that the writes do not block the reads, so that the responsiveness of the server is not affected. Moreover, locking usually introduces quite a lot of overhead which can be saved if some non-blocking synchronization is in place. In section 3.4 I examine a solution to this problem.

3.2 Domain name lookup – the critical point

Replying to a query as performed by an authoritative name server may be divided into several main steps (I now consider only the case of normal queries as it is the most common case):

1. Parsing the incoming DNS packet.
2. Deciding how to answer it (it may be normal query, request for transfer or a dynamic update).
3. Looking up the requested domain name and its associated data (together with finding the proper zone).
4. Looking up authority data to put into the reply.
5. Looking up Additional data to put into the reply.
6. Assembling the response together and sending.

Lookup of domain names is probably the most frequently performed operation of authoritative name servers. It must be performed for each query at least once and it is definitely the most expensive operation when answering from large zones. It is also the only operation with cost depending on size of the zone (I am not considering structures with constant lookup time for reasons mentioned in section 3.2.2). Also, as shown above (section 3.1.2), answering one query may involve even several searches for various domain names from the zone.

In order to minimize the bottleneck caused by domain name lookup I should achieve two goals in the design of the library:

- Reduce the number of lookups required to answer one query.
- Minimize time spent by looking up one name in the zone.

In the following sections I will analyze how these goals may be approached.

3.2.1 Reducing the number of lookups

In order to reduce the number of lookups needed to answer one particular query, one must analyze the contents of such responses. Besides the data associated directly with the requested name (QNAME, query name), several other types of data may be added to the response:

- CNAME records in case of domain name redirection.
- Authority records, i.e. either SOA or NS records from the zone apex.
- A and AAAA records associated with names present in RDATA of certain RR types, such as MX, NS or SOA.
- RRSIG records for each RRSet present in the response.
- NSEC or NSEC3 records proving the non-existence of a certain record or a domain name.
- DS records denoting a signed delegation.
- DNSKEY records containing public keys used for signing the zone.

These can be divided into few groups according to their relation to the zone or some particular domain name or RRSet:

- Records associated with the zone as a whole and present in the zone apex.
    - Authority records (SOA, NS), DNSKEY
- Records associated with certain domain name.
    - CNAME, A, AAAA records; NSEC ad NSEC3 records proving the non-existence of record of given type in an existing name, DS records
- Records associated with particular RRSet.
    - RRSIG
- Records not directly associated with any particular zone content.
    - NSEC3 records covering non-existent domain names or wildcards.

Each of these groups must be treated differently, but for most of them there is a way of to avoid some lookups.

3.2.1.1 Records associated with zone

These records are always present in the zone apex. Usually the zone must be found before a search for particular name in the zone is made. It is then very easy to just keep a direct reference to the apex in the zone structure. No other lookup is then required for finding these records.

If the zone data are kept together for all zones an obvious solution is to add a reference to zone apex to each node of the particular zone, though this is a bit more space-consuming.

3.2.1.2 Records associated with domain name

Here it is important to note that if certain domain name does not belong to the zone, there is no way for a pure authoritative server to know the data associated with it. The exception is when the server holds both the zone which references the name and the one the name belongs to. But even in this configuration it is useless to search for the name in other zone and add it to the response as most of today's resolvers would drop such data as they are 'out-of-bailiwick' and thus not to be trusted (1).

I may thus consider 'in-bailiwick' names (i.e. names in the same zone as the requested domain name) only. One – the more straightforward – way of getting the data associated with RDATA of some RR is to add a reference to the zone domain name to this RR. However, there is a more elegant (and also memory-saving) solution. One may totally avoid duplication of domain names in one zone, store them all in one place and use only references to these domain names (or some IDs) in all places where a domain name occurs.

This however implies a non-trivial complication. If a zone is updated so that some domain name should be removed, it is not clear whether some other data in the zone are referencing the same domain name (i.e. it should not be destroyed) or not. For this purpose, some sort of reference counting will be necessary. We will analyze the particular solution in the next chapter when dealing with implementation.

3.2.1.3 Records associated with particular RRSet

In this case there is probably no better way than saving a reference to the records directly in structure representing the RRSet. The only records of this type are RRSIGs. This type is a bit special anyway, because RRSIG records are the only ones that do not form RRSets (see (2), chapter 3) as different RRSIGs having the same owner have different TTLs (equal to the TTLs of the signed RRSets). It must be however taken into account and solved in the implementation that in theory an update may remove an RRSet, while not removing the associated RRSIGs.

3.2.2 Minimizing lookup time

Even if I get rid of many lookups, at least one lookup per query is absolutely necessary. This still remains to be the most time-consuming operation and its cost grows with the size of the zone (not considering structures with constant-time lookup – see below). It is thus desirable to minimize also the time required to find one particular domain name. It is not of much importance whether the data for all zones are kept together or each zone is stored separately. In either case we have a large data structure holding thousands or even millions of domain names which must be searched.

It is the common task of finding the structure with best possible properties. As I have already shown in section 3.1.4, one needs:

- fast, ideally concurrent lookups,
- non-concurrent writes with no requirements for speed,
- writes not blocking reads.

In general, data structures with best lookup times are hash tables. Writes are however somewhat more difficult. Conflicts cause the structure's lookup time to degrade because larger number of entries must be searched (e.g. in case of chaining hashing scheme) or may force resizing and rehashing of the whole table (as in case of any open-addressing scheme) which consequently also impacts the average lookup time if this operation blocks reads from the table.

The best possible lookup time can be achieved by using perfect hashing (always guaranteed to be constant). It has however some significant drawbacks which make it

inappropriate for the purpose of a DNS server. These are memory overhead (which may be in fact avoided by using minimal perfect hash function; although it is very difficult to construct) and even more the fact that all keys must be known beforehand and adding a new one is very difficult. Even though I do not have to pay much attention to the writes, I must keep in mind that they occur and that they may even introduce new keys.

In the next section I discuss the choice I have found to be most suitable for my needs – the *cuckoo hashing scheme*.

3.2.3 Looking up non-existing names

With the introduction of DNSSEC, a completely new requirement on the zone data arose. If a domain name is not found in the zone, either a previous name in canonical order (when using NSEC) or a previous hashed domain name (in case of NSEC3) must be found. For this purpose, either the normal zone domain names, or the hashed domain names (owners of NSEC3 records) must be kept ordered.

The ordering requirement makes hash table unusable and renders a need for some other data structure. Obvious options are binary search trees (BST), particularly self-balancing binary search trees. They achieve probably the best average lookup time while keeping the memory overhead reasonable. There are not actually significant differences between various types of BSTs, such as AVL or red-black trees. According to (1) AVL trees should be, however, slightly more suitable for frequent lookups and seldom updates, so we chose this variant.

Although skip lists are direct competitors of binary search trees, they require randomization to achieve lookup times comparable to that of BSTs. They thus do not provide the logarithmic worst-case guarantee that BSTs do.

3.3 Cuckoo hashing scheme

First presented in 2001 by R. Pagh and F.F.Rodler of the Aarhus University, Denmark (3), cuckoo hashing is a variant of open addressing scheme. As proved by the authors, this scheme “possesses the same theoretical properties as the classic dictionary of Dietzfelbinger et al. (4) , but is much simpler. The scheme has worst case constant lookup time and amortized expected constant time for updates.” (2 p. 2)

This is ideal for my purpose. On top of that, it is fairly simple to implement. I will go through the requirements later in this section, after short description of the method.

3.3.1 How it works

The original cuckoo hashing scheme as introduced by Pagh, et al. uses two hash tables, T_1 and T_2 and two hash functions *h1*, *h2* for hashing keys into indexes of tables T_1 and T_2 respectively. Every key *x* is stored either on index $h_1(x)$ in table T_1 or on index $h_2(x)$ in table T_2. It is thus obvious that the lookup always takes constant time – in this case it is two operations *at worst*.

Deletion is straightforward and constant if we omit possible shrinking of the tables. Insertion works in the following way: the key *x* is first hashed using the h_1 function. If the place $T_1[h_1(x)]$ is free we put the key there. Otherwise, the key *x* is put there nevertheless, 'kicking out' the previous occupant *y*[16], which is then inserted into its alternative location in table $T_2[h_2(x)]$, and so on. It may happen in this sequence that some key is displaced second time. In such case, it is inserted into its former place and all keys displaced before this one are moved back to their original location. This will cause key *x* to be kicked out of its first location, so the alternative location is tried and the process is repeated. This is what makes cuckoo hashing different from other open addressing schemes: that it may move keys back to their previous locations.

This process may, however, create a loop. In this case it is impossible to accommodate all keys, so a rehash using new pair of functions is performed and the last key not placed is inserted again. In practice, there are two ways of preventing a loop – either a record is kept of the used cells or the number of iterations of the 'insert and kick' loop is limited by some value. A rehash is also needed if the tables become full (or nearly full).

The following image shows an example of the hashing tables. Arrows show places where the items may be moved. In picture (a) the item is successfully placed into the first table. Picture (b) shows a situation when the hashing is not successful.

[16] This behavior, similar to some kinds of cuckoos was the reason for the scheme's name

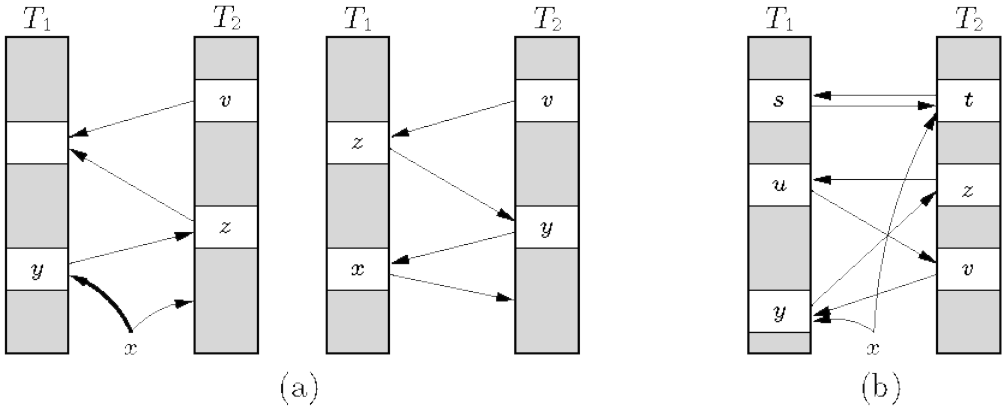


Figure 1 - Cuckoo hashing - successful (a) and unsuccessful (b)[17]

The rehashing has one quite useful property – it does not need any extra space. Keys are taken one-by-one and inserted back using the same insertion algorithm and the new hash functions. In section 2.3 of (3) authors also prove that even with the rehashes, the amortized expected time for insertion is constant. For the rehashing to be useful (i.e. that the new placement of keys will be sufficiently different from the old one, lowering the probability that the insertion of the last key will fail again), the functions must be selected from a universal family of hash functions (see 3.3.4).

3.3.2 Variants

Since the introduction of cuckoo hashing in 2001, many variants of this scheme were developed. We will name and shortly discuss three of them – *blocked cuckoo hashing*, *d-ary cuckoo hashing* and *cuckoo hashing with stash.*

3.3.2.1 D-ary cuckoo hashing

This generalization was designed by D. Fotakis, R. Pagh, P. Sanders and P. Spirakis in 2003 (5). It uses *d* tables and *d* hash functions instead of only two. They come to interesting conclusions that such modification retains all positives of the original cuckoo hashing (constant worst-case lookup time, amortized constant expected insert time) while allowing for much better space utilization. The original scheme with two tables only allowed for a little over 50% space utilization. Fotakis, et al. demonstrate on practical experiments that it achieves 91% utilization for $d = 3$, 97% for $d = 4$ and

[17] The figure was taken from (3)

99% for $d = 5$ (5 p. 14). In practice, choosing the next table to insert a key into is implemented as a random walk.

3.3.2.2 Blocked cuckoo hashing

Another variant of cuckoo hashing is blocked cuckoo hashing (sometimes also called cuckoo hashing with buckets), developed by M. Dietzfelbinger and C. Weidling in 2005 (6). It expands place available for keys in the table, so that at each index can hold more than one key. In contrary to d-ary cuckoo hashing, this variant has an advantage that the buckets are a contiguous blocks of memory so accessing them does not cause cache-miss as can happen with more independent tables.

They also demonstrate the competitiveness of their scheme with the aforementioned d-ary cuckoo hashing by a series of experiments.

3.3.2.3 Cuckoo hashing with stash

Introduced in 2008 by A.Kirsch, M. Mitzenmacher and U.Wieder (7), this modification lowers the probability of a failure causing the table to rehash by adding a small constant-sized *stash*. A key which would cause closed loop in insertion, i.e. normally causing the table to rehash is in this variant inserted into the stash. Only when this stash is full a rehash is initiated.

Authors consider various variants of Cuckoo hashing (actually all I have mentioned above) and the impact the stash is having on them.

3.3.3 The chosen variant

For my purposes I chose to combine two of the three aforementioned variants - d-ary cuckoo hashing and cuckoo hashing with stash. As you will see later on, hash functions from the universal family I decided to use have a range size of power of two. D-ary cuckoo hashing gives better options for choosing size of the hash tables as the total size may be either a power of two (in case of 2 or 4 tables) or between the two consecutive powers of two (in case of 3 tables). Stash, on the other hand, significantly lowers the probability that a rehash would be required.

3.3.4 Universal system of hashing functions

As I have mentioned above, the cuckoo hashing scheme requires an universal family of hashing functions to allow rehashes (either those caused by a loop when inserting

new key or those required for resizing the table). Moreover, they should be given in an analytical form so they can be calculated easily in the program.

A good and simple candidate is class $H_{2\ell,2k}$ of multiplicative hash functions which is defined as:

$$\left\{h_a \middle| h_a: \{0,1,\dots,2^\ell-1\} \to \{0,1,\dots,2^k-1\}\, h_a(x) = \left\lfloor \frac{a *_{2^\ell} x}{2^{\ell-k}} \right\rfloor, a \in \{1,3,5,\dots,2^\ell-1\}\right\},$$

where $\ell, k \in \mathbb{N}^+, \ell \geq k$ and $*_{2^\ell}$ means multiplication modulo 2ℓ. This class was proved to be 2-universal by Dietzfelbinger et al. (8). It has a simple analytical form and actually requires only two arithmetical operations – multiplication modulo 2ℓ and right shift by $\ell - k$ bits. Both of these operations are very cheap on modern processors.

3.3.5 Hashing domain names

Using domain names as keys to be stored in a hash table requires a hash function from the space of domain names to the interval of table indexes. Domain names actually can be treated as simple character strings. I thus need a universal family of functions hashing arbitrary-length strings into integers.

Many functions were created for hashing strings, e.g. the FNV (Fowler/Noll/Vo) hash (9) or the Jenkins' hash (10). A comparison of some of them is available on the internet (11). For our purpose I will combine one of these string-hashing functions and the universal system mentioned in Universal system of hashing functions. The output of the string-hashing function will be used as the input for a function from the $H_{2\ell,2k}$ class.

3.4 Updates

It was already mentioned that besides the obvious lookups, the zone data need to be updated from time to time. There are several ways a zone may be updated. First and the least complicated method is a *full zone transfer* (AXFR). In this case the whole zone contents are sent over TCP to the client. An *incremental zone transfer* (IXFR) is more sophisticated and sends only changes made to the zone since some older version. The last method – a *dynamic update* - is different in that it does not happen between master and slave servers but rather between a client and a master server. This is, however, not important from the library point of view. Dynamic updates are

in nature very similar to IXFR – the packet contains information about changes that are to be made to the zone.

The simplest way of updating a zone would be to create a copy of the zone, update it and then atomically switch the zone in the server. However, this approach creates too much memory overhead, keeping two copies of zone for each update and is also quite slow when it comes to large zones. It is suitable only for full zone transfers when there is no straightforward way to avoid creating a new zone. It is also sufficient for that case.

For updates affecting only part of the zone, a more sophisticated algorithm is required that would not touch the unchanged part of the zone and allow permanent access to the data.

A logical proposition may be to define a unit of atomicity which would always be updated all at once. This unit could be for instance one RR, one RRSet or one domain name (with all its associated data). This will allow updating in similar fashion as mentioned above: copy the unit, apply the required changes and atomically replace the old unit with the new one. It would though not create such memory overhead because only one unit would be copied at a time.

Although seemingly suitable, this algorithm has several fundamental flaws. First – and probably the most important – is that it does not apply the whole update at once. This is not acceptable, because even after updating some part of the zone it may be decided that the update will be discarded. In such case some way of reconstructing the old data would be required. But even if the reconstruction was possible to perform, the server would still use the new data (before it realizes that the update should be cancelled) even though it should not.

One available solution combines two mechanisms: *copy-on-write* (12) and *read-copy-update* (13). Upon an incoming update (or an incremental transfer), a shallow copy of the whole zone is created. The copied zone is then updated accordingly, creating new data and removing references (but not deleting actual data) where necessary. When the update is done, the zone is switched (i.e. the update is performed all at once). Reclamation of the old data is done using the read-copy-update principle. First the zone is switched to the new one. Then the program must

wait while all readers finish reading the old data, and finally the data are destroyed. This on one hand ensures that all readers see consistent data (either the old zone or the updated one) and on the other saves a lot of overhead induced by using locks.

3.4.1 Copy-on-write

As mentioned above, when a zone is to be partially updated, a shallow copy is made. This means that the structures are copied from the top down to the aforementioned atomic unit, e.g. a domain name with its associated data. But the data themselves (the units of atomicity) are not copied. When some unit should be modified, it is first copied, updated and then replaced in the copy of the zone. This approach allows using the old data all the time while the update is being performed on the copy. It also consumes extra memory directly proportional to the amount of updated data.

At the end of the update process, the zone copy contains some references to the old data (that were not changed) and some references to new ones created during the process. After the zone is switched, the old data that are of no use should be removed. It is thus necessary to keep track of these data. For removal, the read-copy-update mechanism is used (see the next section) to ensure that no data are destroyed while still referenced by some reader.

3.4.2 Read-copy-update

Although using hash table with cuckoo hashing scheme allows very fast lookups of domain names and their associated data, another aspect I need to take care of is data updates. You have seen that the hash table allows insertion and deletion of keys. However, if I have to lock the whole table or even if I have to lock only the one element containing the data, it will significantly impact the overall performance. One thing is the inevitable overhead of locking and unlocking and another one is blocking reads while the update is in progress. Ideally, the data structure should allow updating without affecting reads at all. For this purpose a very simple but strong mechanism is available – RCU (read-copy-update). This is proved to achieve superior performance for workloads with small portion of updates (which is the case of an authoritative server as mentioned in 3.1.4) (14).

RCU provides wait-free reads with very low overhead. It uses so-called *read-side critical sections*: an RCU-protected data structure may be only referenced in these

sections of the code. Another term used by RCU is *grace period* which is any time period during which each thread of the program at least once enters a state when it is not inside any read-side critical section. A grace period is not a time interval of given length. It starts in a specified moment (usually when the writer call some specific function) and lasts until last thread that entered a read-side critical section before the start of the grace period exits the critical section. This definition implies that any read-side critical section which began before the start of some grace period must complete within this period. This is how an update is performed when using RCU:

1. Remove all pointers to the old data (or replace them by pointers to the new – updated – data). By doing so it will be ensured that no readers can gain reference to the old data.
2. Wait until the grace period ends, i.e. until all readers finish their read-side critical sections that started before the period.
3. Destroy (reclaim) the old data. It may be done because at this moment no thread can hold a reference to it. It may be done by the writer or the reference may be handled to some special ‘garbage collecting’ thread.

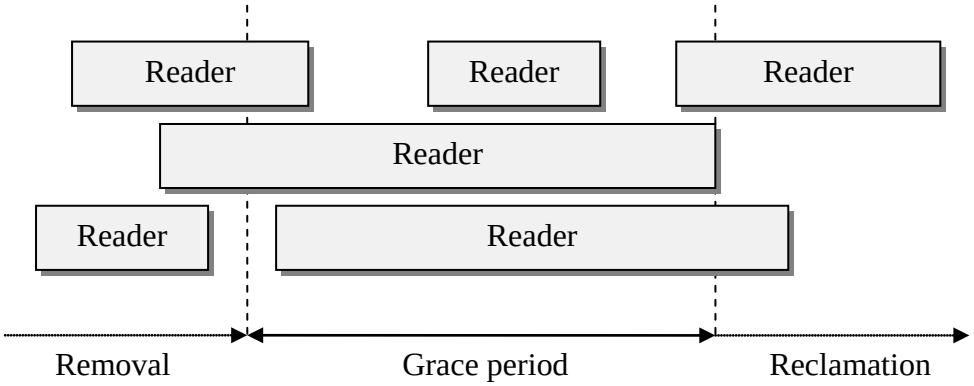


Figure 2 - RCU phases

4 Design and implementation

4.1 Modules

The whole library is divided into several modules. This separation is both logical and functional. Logically, structures and functions corresponding to one DNS structure or feature are grouped into one module. Functionally, each module provides a clearly defined API used in other modules or in any application using the library. Modules are organized into files in a one-to-one fashion - one module in one file and one file contains only one module (more precisely, each module is represented by two files – the header and the source file). A naming convention of using the module name as prefix is used to distinguish structures and functions from particular module.

The main modules of the library are:

- `dname` – domain name,
- `rdata` – RDATA part of resource records,
- `rrset` – RRSet – a set of resource records,
- `node` – one node in the zone tree,
- `zone` – entire zone,
- `zone-tree` – wrapper around AVL tree to provide some special features,
- `zonedb` – all zones operated by the server,
- `packet` – DNS packet,
- `query` – query manipulation,
- `response` – response manipulation,
- `nsec3` – special handling required by NSEC3,
- `edns` – ENDS0 functionality
- `xfr` – functions related to AXFR and IXFR
- `nameserver` - high-level query processing

There are several other parts of the library that are not actual modules but use the same conventions for naming, logical and functional separation. These include resource record type descriptors, set of functions for manipulating wire format of the packet, debug facilities, error codes and routines, etc.

I will not discuss the modules in more detail here. Most important data structures (often directly corresponding to some module) will be examined in the following section. For more detailed information refer to the developer and/or reference documentation[18].

4.2 Data structures

In this section I will present an overall look on the data structures representing the zone, its data and DNS messages. Zone structures will be presented from top to bottom as some design choices on lower levels make sense only within wider context of the higher level.

4.2.1 Zone structures

4.2.1.1 Zone database

Zone database is a very simple set of zones operated by the server. It is implemented as an AVL tree. Although it may seem favorable to use some even faster data structure, like hash table, when searching for the right zone to find the requested name in I do not know the exact zone name. On the contrary, I must find a zone with name matching the most possible labels. This can be also done as a search for 'highest' domain name 'less' than the requested one, in canonical order that is easy in an AVL tree.

4.2.1.2 Zone

Zone is a rather simple structure that just keeps reference to its contents (see below) and an arbitrary data that can be used by the application (a common usage would be to store some Access Control List or some other configuration data associated in the zone). The zone contents are however separated, so that updates to the zone may be done more easily as just the contents' pointers is switched.

[18] See Appendix B

4.2.1.3 Zone contents and zone tree

Zone-contents is a quite complex structure and provides high-level API for operations on it. It holds zone data in several main structures – a hash table and three AVL trees. Data in the zone are organized into *nodes* (see 4.2.2), which are inserted in one or two of the main structures. So-called empty non-terminals are also present in the zone structures, represented as normal nodes with no data in them. This is mainly because they have to be treated in a special way (a query for domain name being an empty non-terminal in a zone results in a 'NODATA' response, which is different from 'NXDOMAIN' sent when the domain name does not exist).

Normal zone nodes are inserted into the hash table and to the main *zone tree* (which is actually implemented as AVL tree with some extended functionality). Nodes containing NSEC3 RRSets are not kept in the hash table, or in the main zone tree. Instead, a separate zone tree is used to keep them. This has several reasons. There may be many NSEC3 records in the zone (de facto roughly as many as there are normal nodes), but these are looked up only in some situations. Inserting them to the hash table and main zone tree would cause all searches to be slower. There is also another reason, even more important – when searching for NSEC3 records it is often the case that the program needs to find previous NSEC3 RRSet in canonical order. This is only possible if they are kept separately and not mixed up with other nodes.

The last tree in the zone structure is actually called *domain table*. It is basically a table (implemented as an AVL tree) holding all domain names present in the zone. It saves space that would be required if the domain names were duplicated. It also allows easy management of these domain names when creating, modifying and destroying zone.

As already mentioned, the `zone` module provides a high-level API for all operations on the zone:

- Creating and destroying zone.
- Adding nodes to the zone. The structure distinguishes between normal nodes and so-called NSEC3 nodes (see above).
- Adding both normal and NSEC3 RRSets to the zone. (There are special functions for this because it must be ensured that all the domain names from

the inserted RRSets are properly saved in the domain table or replaced by the already existing ones.)

- Adding RRSIGs to the zone.
- Searching for particular name in the zone. Also separate functions for normal and NSEC3 names. There are several functions for this purpose which differ in the used structure (tree or hash table) and on the results given (only the one node if found, or also other nodes such as previous node in canonical order).
- Functions for walking through the zone in various ways (forwards or backwards, in-order or post-order) and applying a given function to each of its nodes.
- Creating a shallow copy of the zone. (This is useful when updating zone from an IXFR or a dynamic update.)
- Other functions for getting some zone characteristics.

The application using the library should use only this API for zone manipulation, not the lower-level API of modules representing smaller parts of the zone data (such as `node`, `rrset`, or other), so that the zone integrity is ensured at all times. Moreover, after a zone is built, it is necessary to do some overall adjusting by calling `knot_zone_contents_adjust()`

4.2.1.4 Zone adjusting

This is one of the most important features of the library. When a zone is created (either from reading a zone file or from a zone transfer), it is not optimized for faster lookup in any way. The zone adjusting walks through the zone and sets various cross-references, removes redundant domain names and marks zone nodes with various flags. All optimizations mentioned in 3.2.1 are performed. For detailed information consult the developer's documentation or reference documentation[19].

4.2.2 Node

From various points of view several types of nodes may be distinguished:

- Normal nodes vs. NSEC3 nodes

[19] See Appendix B

    - Normal nodes belong to the regular zone tree, i.e. their owners are regular domain names. These nodes may or may not contain data (see below).
    - NSEC3 nodes contain NSEC3 (and RRSIG) records, their owners are hashed domain names from the regular zone tree. They are kept in a separate tree (see Zone database, zone, zone tree).
- Regular nodes vs. empty non-terminals
    - Regular nodes are normal nodes containing some data (i.e. some RRSets).
    - Empty non-terminals contain no RRSets. They are not marked by any flag. The lack of RRSets is enough to distinguish them.
- Authoritative nodes vs. delegation points vs. non-authoritative nodes
    - Authoritative nodes are simply those nodes whose data are authoritative in the zone. It means all nodes in the domain name hierarchy from zone apex down to zone cuts (delegation points).
    - Delegation point is a node within a zone which is not the zone apex, but contains NS records. Such nodes denote zone cuts. They delegate authority over the child zone to other name servers. In this implementation they are marked using a flag (authoritative nodes are actually recognized by lack of any flag).
    - Non-authoritative node is any node lying below a zone cut. Although this node does not belong to the parent zone by definition, it usually carries Glue records required for referral responses and thus must be stored within the zone, so that the data may be retrieved. Also marked by a flag.

All these types share the same `node` structure and API, but are distinguished among either by a flag or by some characteristic (see above).

4.2.3 RRSet

An RRSet is one or more resource records (RR) that all share the same owner, type, class and TTL. The concept was inherent with DNS from the start, but only in 1997 the term RRSet was defined in RFC 2181 (15). Zone data are almost always manipulated on the RRSet level. For instance, when putting RRs into the response, the server may choose to include either whole RRSet or nothing. Moreover, this behavior is enforced when DNSSEC is used.

In practice, all RRs of certain type present at one owner domain name always share the same TTL and thus form an RRSet. However, as mentioned before (3.2.1.3), RRSIG records comprise an exception to this rule. This is due to the fact that an RRSIG record signing some RRSet must have the same TTL as this RRSet. This may result in various RRSIG records at one owner having different TTLs and thus not forming an RRSet. (For more information see (2)) On the other hand, RRSIGs signing the same RRSet will always have the same TTL and thus de facto do form an RRSet.

These facts led to creation of a structure representing an RRSet and also to the lack of any structure representing one particular RR. An RRSet has a ‘header’ (owner, type, TTL, class) and a set of RDATAs. RRs within RRSet differ only in the RDATA section, so one RDATA together with the ‘header’ from the RRSet may be considered one RR. In case you need to manipulate only one RR it is very simple and straightforward to create an RRSet structure containing only one RDATA.

An RRSet structure also contains reference to the set of RRSIG records signing this RRSet. These records are not saved directly in the nodes because they are tightly associated with the RRSet they sign.

4.2.4 Domain name

Domain name is represented by the `dname` structure (and module). It is a rather simple structure just holding the name in wire format and some information about it – length and positions of labels. The positions of labels within the domain name brings some parsing overhead both on zone parsing and on packet parsing but enormously improves the domain name comparison process. Domain names must be compared label-by-label, not just byte-by-byte. As domain name comparison is the most

frequent operation done by the server (each domain name lookup involves several domain name comparisons) improvement in its speed impacts the overall performance greatly.

I already mentioned that the domain names are kept in wire format. This is useful for several reasons. Firstly, domain names must be compared in wire format. It is also significantly faster to put the name into the response packet because no conversion is required. On the other hand it brings overhead when parsing zone file but that is acceptable as it may be done independently on the server operation and does not impact the server's performance.

It was also mentioned above (4.2.1.3), all domain names from the zone are kept in a domain table stored within the `zone` structure. This prevents domain name duplication in the zone and allows for easy cross-referencing data between various nodes and their data as designed in section 3.2.1. The `dname` structure also holds a pointer to its associated node from the zone (if there is such a node). This allows to find a node (and thus the associated records) owned by domain name referenced in RDATA of some MX record. However, to use this functionality, the domain name must be inserted into the zone and thus to the domain table (see 4.2.1.3).

As references to one domain name may be scattered among the zone, it is necessary to keep track of them so that on one hand the domain name is not destroyed while some other structure still refers to it, and on the other that it may be destroyed when no one is referencing it anymore. For this purpose, a reference-counting mechanism was implemented, utilizing the GCC atomic built-ins. The reference counting is already used by many of the library functions, but it can be ignored in an application, if needed.

4.2.5 RDATA and RDATA items

RDATA (actually RDATA items) are the smallest units of DNS data that are represented in the library. An RDATA contains the data stored within one resource record (not counting the 'header'). This data may be divided into smaller parts which are named *RDATA items*. Although it is not completely necessary to parse the RDATA and keep them in parsed form (i.e. divided into the RDATA items), it is often useful during the response process. Furthermore, without parsed RDATA items

it would not be possible to replace domain names present in RDATA with the domain names from the zone and thus I would not be able to get rid of some name lookups (see section 3.2.1 and 4.2.4).

RDATA items are, however, kept also in wire format (as the domain names). They are just separated for easier access.

4.2.6 DNS packet

A DNS packet is represented by the `packet` structure and the `packet`, `query` and `response` modules. The functionality is divided logically but reasons for this division are mostly organizational.

The `packet` structure is rather complex. Besides containing basic parts of the DNS packet – the header, Question section and list of RRs belonging to the Answer, Authority and Additional sections, it contains many other useful data or references. Pointer to wire format of the packet is kept there (which may be either network data or a created and converted packet) as well as size of the packet in wire format. Moreover, it contains some information useful for compressing domain names in the packet, a list of temporary RRSets that may have been created or a pointer to associated query (if this packet is a response).

Working with the `packet` structure is easy thanks to the convenient API available. Its internals are, however, quite complex. It is possible to create a packet structure with pre-allocated space so that number of allocations while processing it is minimized. There are some predefined types of pre-allocation. The structure then manages the pre-allocated space and obtains more memory only when needed.

Functions for parsing packet from wire format are available. Either the whole packet may be parsed at once or only header and Question section are parsed and the rest may be parsed iteratively, one RRSet at a time.

4.2.6.1 DNS query

Though there is only one universal structure (`packet`) used to represent both queries and responses, there are separate modules for queries and responses available with specialized API. The `query` module contains several functions for getting some specifics of the request (such as whether DNSSEC was requested or not). Besides

that it provides functions for constructing a query as an authoritative server may also need that in some cases, such as when asking for transfer from its master server.

4.2.6.2 DNS response

API for creating responses contains similar functions for composing a reply. This is somewhat more complicated than creating queries so the API is more comprehensive as well. It provides functions for initializing response from an incoming query, for adding RRSets to all parts of the response, setting various flags, RCODE, etc.

4.3 Query processing

Query processing is the most high-level functionality provided by the library. In fact, it provides functions for complete handling of incoming queries and responses (in case of inbound transfer). This was the ultimate goal for the library – to provide such interface that it would be simple and straightforward to integrate the library into a server.

The design of the module somehow implies a way the server may process incoming packets, but it does not require it. The complete decision graph is available in Appendix B – Contents of the CD. In the following diagram we present the implicit way of packet handling which takes most advantage of the prepared API.

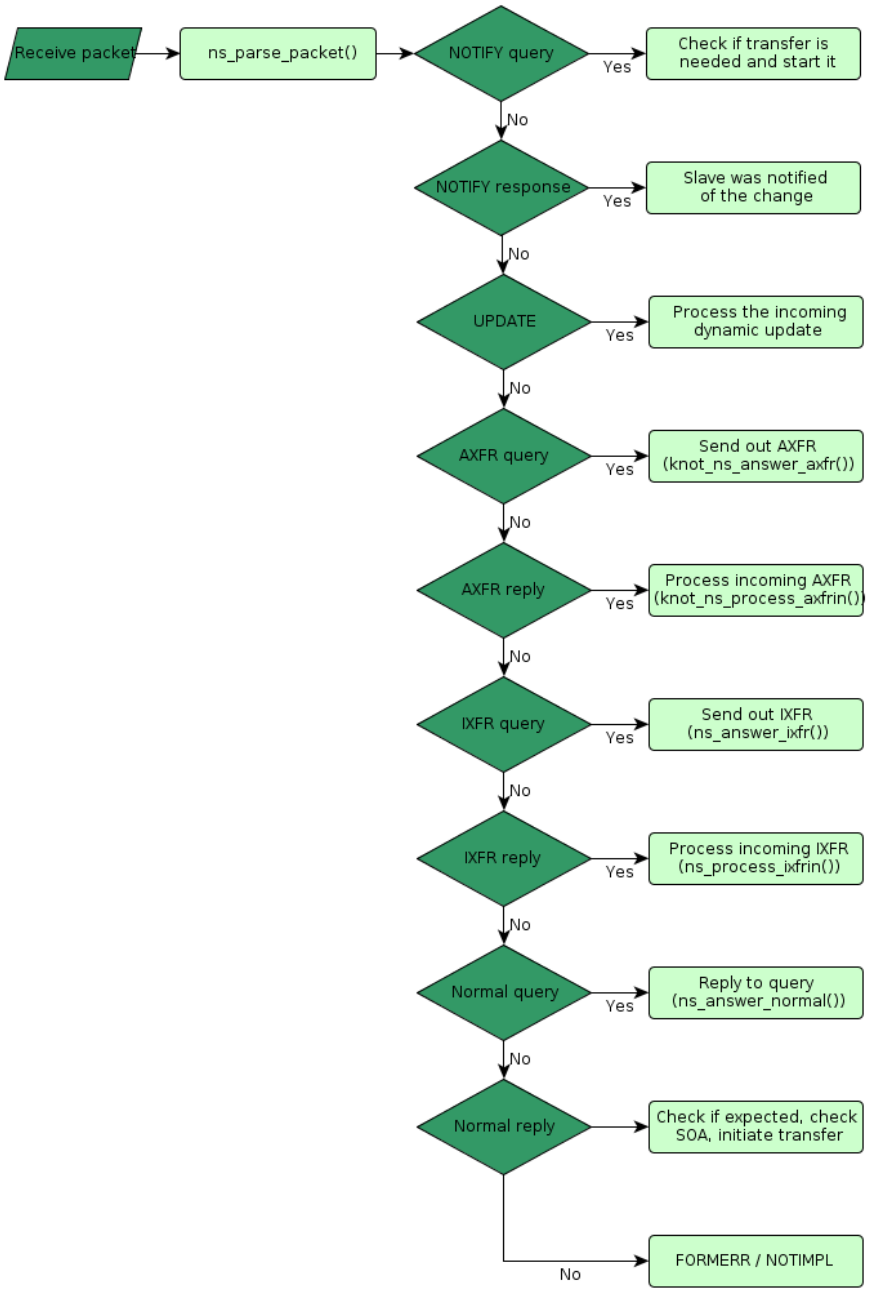


Figure 3 - Packet types and associated actions

As you can see in the diagram, there is a function for handling each type of packet. Let's now take a closer look at the internals of these functions. I will not go into much implementation details but rather outline the basic workflow of the functions.

4.3.1 Normal queries

Normal queries (i.e. not the one mentioned in the following sections) are the most common packets a DNS server receives. They ask for some data at a particular

domain name. Their processing is probably the most complicated as the responses are quite complex containing not only the direct answer to the query but also some data about the authoritative name servers for the zone and additional data the client might find useful. It also differs depending on the actual answer – whether the name was found or not, whether it contained any data, etc. The diagram in Appendix A shows the replying process with the hash table as the main structure.

4.3.2 Zone transfers

Zone transfers require special handling for various reasons. Firstly, inbound transfers are often divided into many DNS packets and the data from them must be collected and processed together. This requires a mechanism of keeping the transfer information between subsequent calls to some processing function. For this purpose, the `knot_ns_xfr_t` structure is available. It stores opaque data that may be used by the server to keep the state of the transfer. But more importantly, it stores data required by the transfer-processing functions. These are also obscured from the view of the server.

Moreover, the IXFR transfers require information about changes made to the zone. To give this information a common format, the `knot_changeset_t` and `knot_changesets_t` structures were created. It is utilized by the IXFR-processing functions which either fill in these structures or use the data in them to construct a reply.

4.3.2.1 AXFR request

Processing an AXFR request is rather straightforward – find the zone the client asks for and send out all RRs contained within the zone. This is what the `knot_ns_answer_axfr()` function does – it walks the whole zone (it uses the zone trees containing normal and NSEC3 nodes for this purpose) and puts all data into the reply. Although the client must be able to receive the AXFR reply no matter the order of RRs in it, my implementation has a nice property that the data are sent out in canonical order of owner domain names, RRSets are always grouped together and RRSIGs signing a RRSet immediately follow that RRSet. If the client knew that the server is using this format (e.g. via some identification mechanism, such as NSID – see (16)), it could take advantage of it and simplify the zone building process.

The only tricky part in responding to an AXFR query is dividing the reply into more DNS packets if this is required. AXFR is possible only via TCP but the limit for one DNS packet is 64 kB which is clearly not enough for zone sizes in megabytes or even gigabytes. For this purpose it uses the `knot_ns_xfr_t` structure, which holds information about the TCP session and a function for sending out the packet. These are obscure to the AXFR-processing function. It does not need to understand the data. It just calls the function when a packet is ready to be sent out. Everything else should be handled by the server itself.

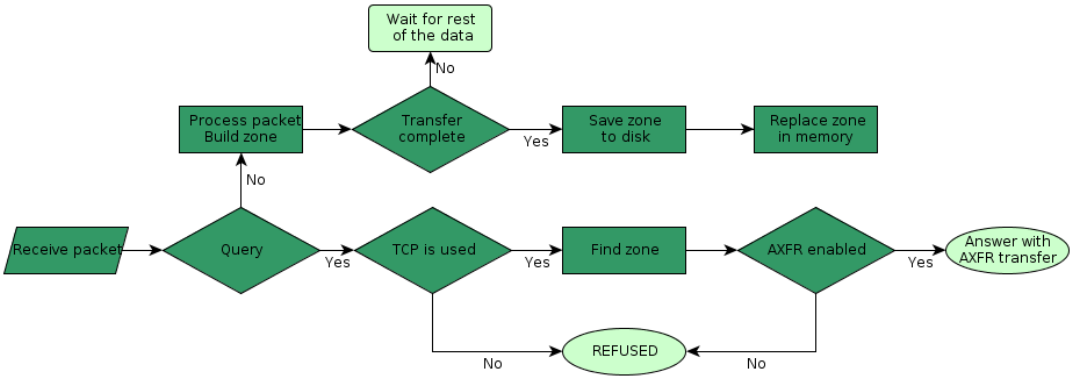


Figure 4 - AXFR processing

4.3.2.2 AXFR reply

An AXFR reply should only be processed if the server did request it. This is however a responsibility of the server, not the library itself. The library provides function for processing the reply. This function also takes advantage of the `knot_ns_xfr_t` structure where it stores the partially-built zone. The `knot_ns_process_axfrin()` function in fact takes only one DNS packet with part of the AXFR reply and depending on the data stored in the `knot_ns_xfr_t` structure it either starts building a new zone (if there is none present) or continues to build the one present there.

While building a zone, each RR from the reply is processed separately (i.e. an RRSet structure with only one RDATA is created) and as such it is inserted into the zone. It utilizes the zone functions for inserting RRSets which also take care about merging the RRs into RRSets.

After creating the zone, it must be saved to the disk. This is however the server's responsibility. After doing it, the `knot_ns_xfr_t` structure containing the built zone should be handed over to the `knot_ns_switch_zone()` function which takes care about updating the zone. This is rather simple, because just the zone contents in the `knot_zone_t` structure are changed. This is, of course, synchronized using RCU (for more details see section 4.4).

4.3.2.3 IXFR request

An IXFR request processing has two main steps. First (besides finding the right zone) it is necessary to find the range of zone versions the client needs to receive. The difference information should be kept in some form of journal which is server-dependent. The server should also take care of loading the data from journal to the library's `knot_changesets_t` structure.

After the changesets are ready, the `knot_ns_answer_ixfr()` function creates a reply for the request. This is rather simple and consists of simply putting one changeset after another into the Answer section of the packet. It uses the same mechanism to send out packets as AXFR request processing (see 4.3.2.1).

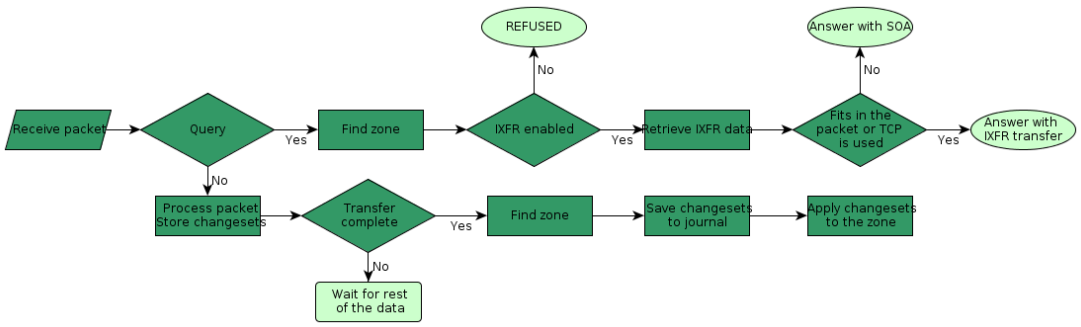


Figure 5 - IXFR processing

4.3.2.4 IXFR reply

As in the case of AXFR, the inbound incremental transfer is quite complicated to handle. First of all it must deal with the reply divided into several DNS packets. This is done in similar fashion as in AXFR reply processing (see 4.3.2.2), but instead of storing the partially built zone, it uses the `knot_ns_xfr_t` structure to keep the changesets created from the incoming data between successive calls to the `knot_ns_process_ixfrin()`.

Another tricky part is processing of changesets. These must be firstly saved into the permanent storage. The server must however take care of it. Afterwards they should be applied to the zone using the `xfrin_apply_changesets_to_zone()` function. The zone update process utilizes the copy-on-write principle and makes extensive use of RCU. For more details see section 4.4.2.

4.3.3 NOTIFY request and reply

The whole NOTIFY protocol is tightly bound to the server's internals. It must deal with the REFRESH timer (used to schedule a check whether the zone should be updated) as well as match NOTIFY replies to queries (in case of master server). For this reason, support for this protocol was not included in the library.

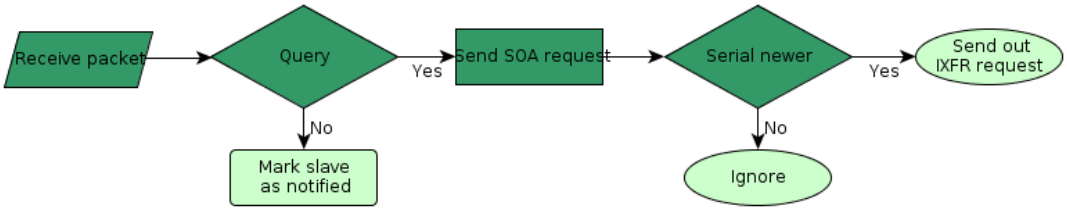


Figure 6 - NOTIFY processing

4.3.4 Dynamic update

Dynamic updates are very similar to incremental transfers in their nature. They can also remove and add RRs to the zone, though they additionally state prerequisites that must be met in order to apply the update. From the viewpoint of zone changes they are, however, nearly identical.

We can use this fact to design the dynamic update processing mechanism. The dynamic update can be converted into the same format of changesets as in case of IXFR. For the rest of the processing the same functions can be used.

Support for dynamic updates was not implemented in the library after all, both for time reasons and because there was no way of testing it as the server used for testing (see 2.4) does not support this protocol yet.

4.4 Synchronization

As mentioned before (section 3.4), the library uses *copy-on-write* and *read-copy-update* mechanisms to ensure non-blocking synchronization of zone data. Data

between zones do not have to be synchronized, as replying from one zone is independent on the others. The same goes for updating.

4.4.1 Updating whole zone (AXFR)

When updating whole zone, a new zone is created from the incoming transfer. There is no straightforward way to avoid this. The library could search for the data in the old zone and copy references, but it would be very time-consuming and will severely impact the speed of transfer processing. After the zone is created, it is atomically replaced in the zone database structure. This comprises the first step in RCU-aided synchronization (see 3.4.2). After this moment, any new searches for domain names belonging to this zone use the new zone, but all previously started searches still use the old data. Next, the library function waits for the end of the grace period (by calling `synchronize_rcu()`). Afterwards it is possible to destroy the old data (last step of RCU synchronization), so the whole old zone is destroyed.

4.4.2 Updating part of the zone (IXFR, dynamic updates)

In case of incremental transfer or a dynamic update, only a part of the zone is updated. This is done precisely as described in section 3.4.1. A new `zone` structure is created with new hash table, domain table, normal zone tree and NSEC3 zone tree. These structures are then filled with the same data as the original zone, i.e. the `node`, `rrset` (with `rdata`) and `dname` structures are not copied. The update information, saved in form of changesets is then processed and the copy of the zone is updated accordingly.

When an RRSet should be deleted, only reference to it is removed from the new zone. The same goes for the whole node if all RRSets were removed from it. However, a node may be removed only when it is a leaf node in the zone hierarchy, i.e. its owner has no descendants. This is because if the node had any descendants, it would then be considered an empty non-terminal which has to be included in the zone structures as mentioned before (4.2.1.3).

After the update is complete, the new copy of the zone references some old data and some new ones. It is then replaced in the zone database structure similarly as in case of AXFR (see above). The rest of the process is the same – wait until the end of the grace period and reclaim the old data. The data to be deleted after the update are kept

in a dedicated private structure `xfrin_changes_t`. This structure also keeps references to all the data that should be removed in case the update fails.

4.5 Language, platform, system requirements

The whole library is implemented in the C language. This language allows writing quite low-level and high-performance code. The library was developed primarily for Linux system and the x86 and x86-64 platforms. However, it may be used on any POSIX-compatible system. It was tested on the FreeBSD and Mac OS X operating systems.

Only the implementation of RCU (the `urcu` library, see 4.6) may impose complications when porting the library to other systems. It also requires GCC to build because of some atomic built-in macros required to implement the reference-counting of domain names.

4.6 External sources

Several parts of the library are taken from various sources. Mainly the user-space read-copy-update implementation (the `urcu` library[20]) came in very handy. It is probably the only one existing implementation, but is referenced from various places and actively developed. So far there were no problems using it.

Another adopted part is the implementation of an AVL tree[21]. This is one of the notoriously known data structures and thousands of implementations exist. I chose one which, although written in C, provides a generic interface. It is very lightweight and serves well too. It was, however, partially modified to better suite my needs (e.g. a function searching for largest key less or equal that some given key).

Another feature not implemented by hand is the string hashing function. I have incorporated two different hashing functions that can be easily switched in the code. They are downloaded from various sources, more precisely:

- FNV hash function from
  http://freebsd.active-venture.com/FreeBSD-srctree/newsrc/sys/fnv_hash.h.html

[20] http://lttng.org/urcu
[21] Taken from http://piumarta.com/software/tree/

- Jenkins hash function from
  http://burtleburtle.net/bob/hash/doobs.html

There is some additional code that was implemented in the server I used for testing (see next section) and that was utilized in the library as well. These source files, together with some generic structures and functions of my own, are grouped in the `common` module and include: slab allocator (not required, but currently utilized by the `dname` module), Base32 encoding with extended hex alphabet and a generic structure representing a socket address.

4.7 Integration into server

As already mentioned when stating goals of this work (section Goals), the library itself could not be properly tested or benchmarked. Integration into DNS server software is required to accomplish this. I thus cooperated with developers from CZ.NIC Labs who were implementing an experimental server based on `ldns` library. Thanks to the good and comprehensive design of my library and ease of use of its API, it was not too much work to meld it together. The server modules manage network and threading, and call the most high-level API of my library (mostly from the `zone`, `nameserver` and `xfr` modules) to process the incoming data.

The result was used to on one hand test functionality of the library – i.e. whether it properly processes the requests, and on the other hand to measure the performance and compare it with other servers, namely BIND and NSD.

5 Benchmarks and tests

5.1 Setup

5.1.1 Server

For benchmarking the library, the server mentioned before, was used (see 4.7). The server was implemented and set-up in such way that allows utilizing as much of the library performance as possible. However, such goal can never be fully achieved. The network and threading code always presents some overhead (even if it is very lightweight) and in certain cases even the limitations and oddities of the operating system may influence the overall performance.

5.1.1.1 Socket handling

When testing the server with the integrated library, two different solutions of socket-handling were tried and they showed a significantly different performance. If the sockets were managed with a `select()` and subsequently `recvfrom()` was used to read the data, the performance scaled linearly but the peak was not as high as when using the second approach. The other solution used only a blocking `recvfrom()` called from more threads at once. It does not scale as well when incrementing thread count, but it yielded a much better top performance (probably due to the lower syscall overhead),.

For our benchmarks we chose the second variant as it allowed us to achieve the top performance and thus was not limiting the library.

5.1.2 Hardware setup

For my purposes, a small dedicated test lab has been provided by CZ.NIC. This includes three identical servers with a dedicated local network connecting them together and a separate ‘control’ interface for connecting to the servers from outside (to ensure that the control traffic does not interfere with the DNS traffic). All three machines have the same hardware configurations – 4-core Intel Xeon X3430 2.4 GHz processors, 2 GB of memory and Ubuntu Linux 10.04 operating system. The testing network had 1 Gb bandwidth and the machines were connected by a professional-grade Cisco switch.

5.1.3 Reference implementations

I have chosen just two reference implementations for the benchmarks – the BIND server and NSD. They are the most often used authoritative name servers, they fully implement DNSSEC with NSEC3 and they are both open-source. Other available implementations are either proprietary (Nominum ANS), do not use in-memory structures at all so the speed is not comparable (PowerDNS) or are no longer maintained (MaraDNS, djbdns). The chosen implementations are however a good reference, though it must be noted that both of them are production-grade name servers developed many years by dedicated groups of developers.

5.1.4 Zone setup

I used a copy of the `.cz` zone file (also provided by CZ.NIC) signed with 2048-bit key and using NSEC3. Queries were generated from the zone contents and contained names from the zone, names outside the zone, names under delegation points and RR types present or not present at the names. This provided a wide variety of queries and tested probably every type of response that could have been generated from the data.

5.1.5 Testing tools

In the benchmarks I aimed for measuring the maximum reply rate achieved by the servers. For this purpose a convenient tool `dnsperf` is provided by Nominum. This tool is designed to stress the server with a high load of queries while the server manages to answer them within certain interval. When it reaches the highest reply rate of the server it automatically limits the speed. This way it effectively measures the maximum possible performance of the server.

The tool allows limiting the maximum number of outstanding queries (i.e. the queries that were sent and not answered yet). The default value is 20 which I used for one set of runs. For the other set I set the value to 100 to allow for slightly longer reply times and thus allowing stressing the server even more. This value was chosen after few tries with different values (50, 100, 200, 500, 1000) because it yielded the best performance of the servers.

Beside this I used only some basic shell commands to run the tests and collect results.

5.1.6 Scenarios

One of the servers hosted the currently tested authoritative server implementation. Beside the server daemon the number of running programs was reduced to minimum. The lab setup allowed a maximum two-client setup. I thus tested both situations – a single client sending queries to the server and two clients sending queries simultaneously. In both cases I measured the total reply rate of the server.

Another varying parameter was DNSSEC. In one test set, the queries did not request DNSSEC records which naturally resulted in better reply rates. When DNSSEC was requested, the response sizes raised significantly (from approx. 140 B without DNSSEC to approx. 500 B with DNSSEC), influencing the performance as well.

To more precisely test the capabilities of each server, I varied the number of threads (or processes in case of NSD) used by the server. The test started at one thread / process and increased until it seemed that the reply rate may increase. I found out that with this setup (4-core processors and only 2 clients) the performance started to drop at only 5 threads / processes. It is although likely that with more clients or a more-CPU server the optimum count would be different. My goal was, however, not to extensively test the capabilities of each implementation but rather to provide some basic reference of my library's performance.

5.2 Results

5.2.1 How to read the results

In each chart, the X axis shows the number of threads the server used. The Y axis represents reply rate achieved by the tested server. In each chart all three servers' results are presented for better comparison. Moreover each server is represented by two lines – one for each setup of the maximum number of outstanding queries (see 5.1.6), named 'frame' in the charts.

5.2.2 Scenario 1 – one client, no DNSSEC

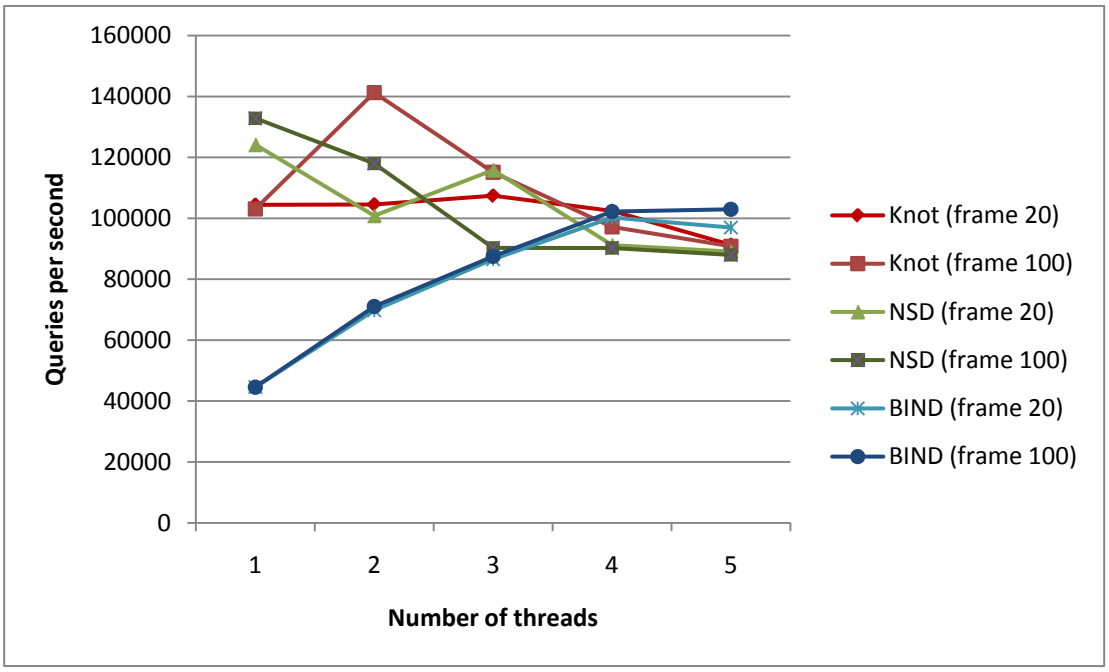


Figure 7 - Reply rate vs. number of threads - 1 client, no DNSSEC

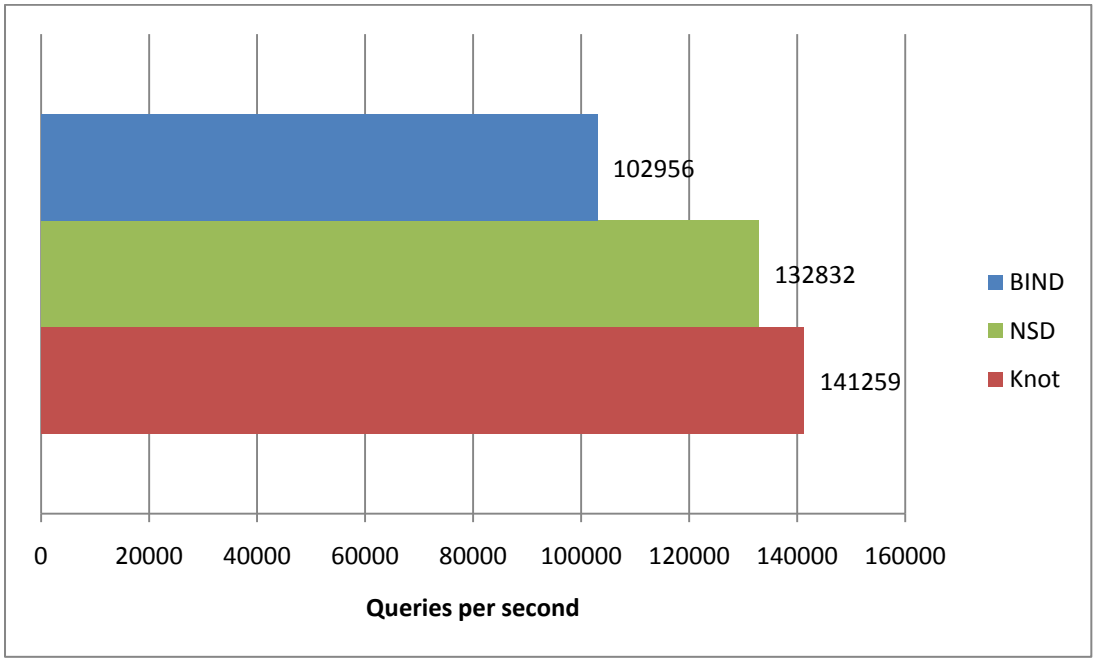


Figure 8 - Maximum reply rate - 1 client, no DNSSEC

5.2.3 Scenario 2 – one client, DNSSEC

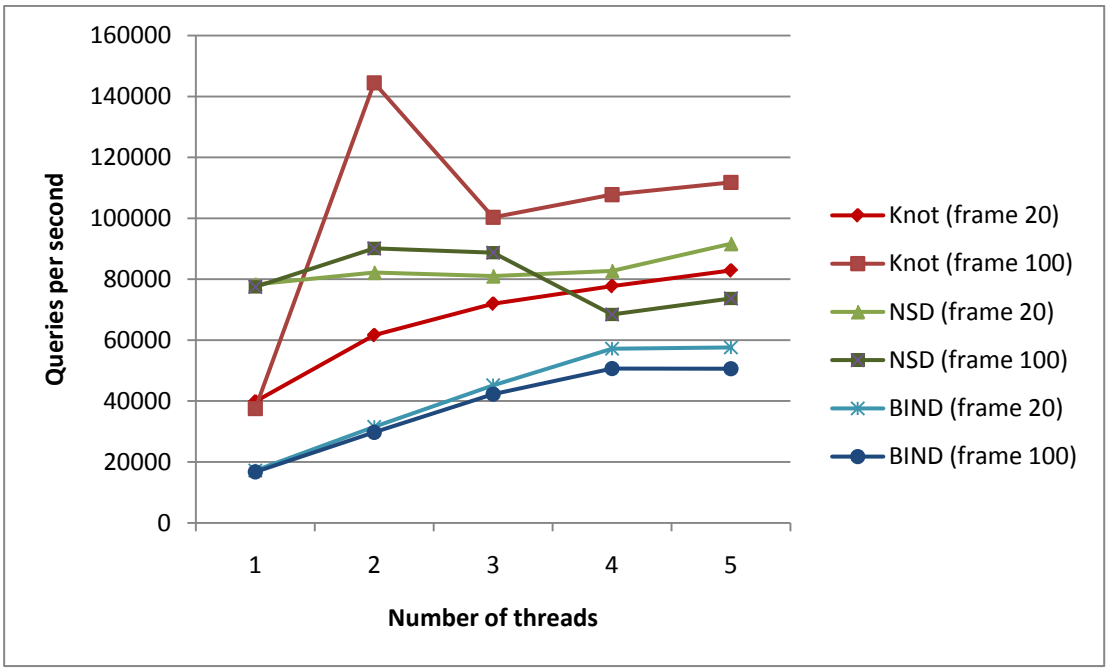


Figure 9 - Reply rate vs. number of threads - 1 client, DNSSEC

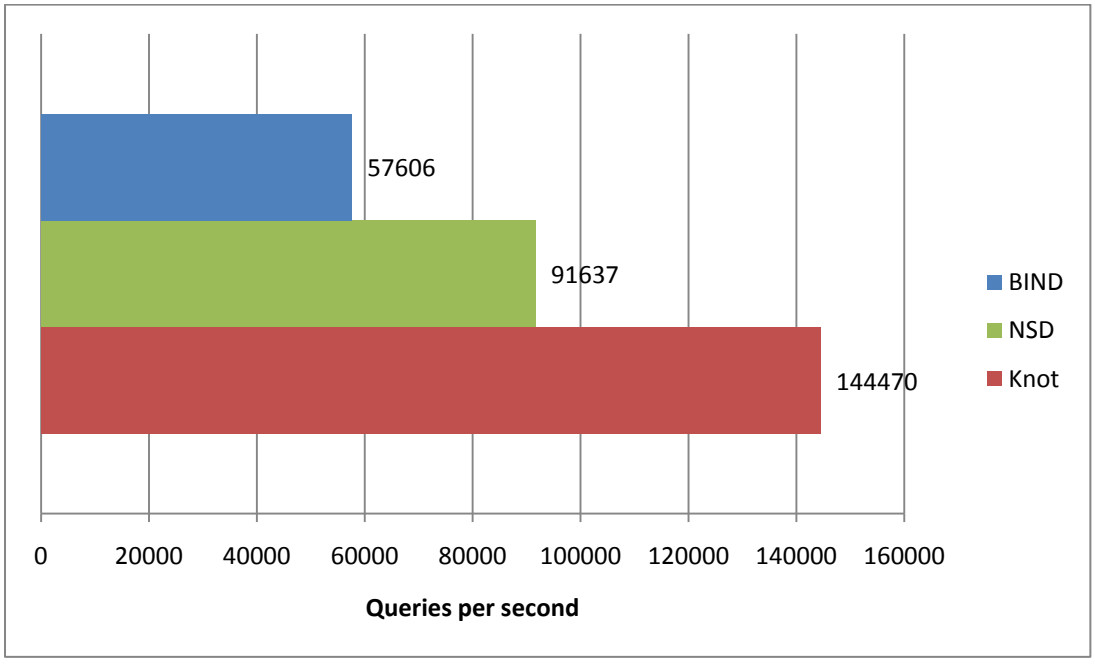


Figure 10 - Maximum reply rate - 1 client, DNSSEC

5.2.4 Scenario 3 – two clients, no DNSSEC

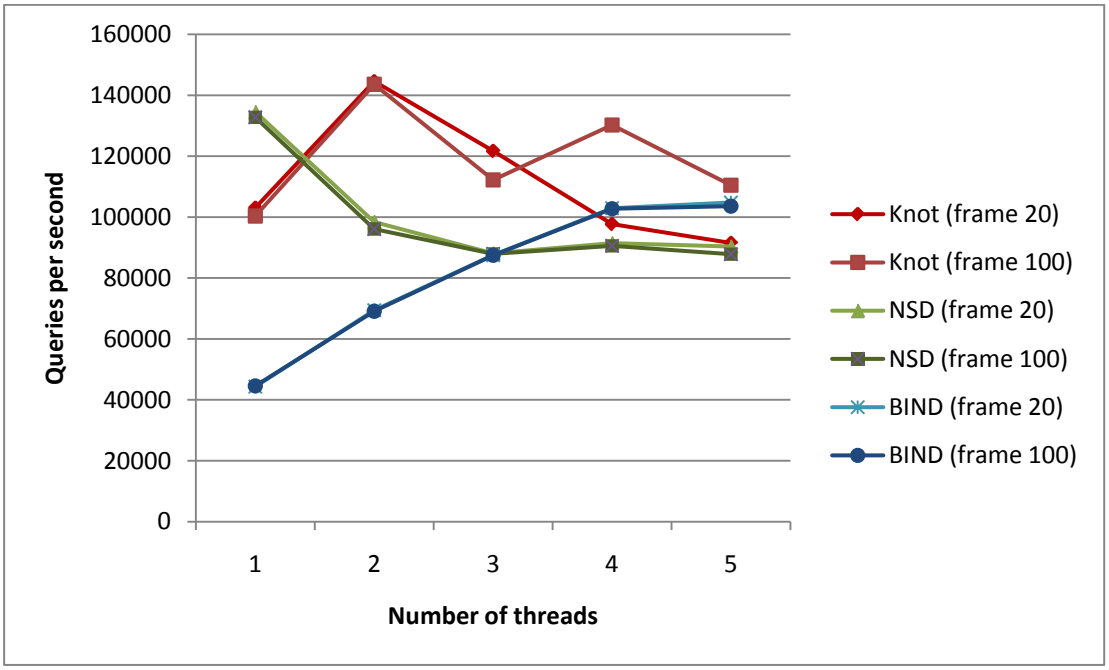


Figure 11 - Reply rate vs. number of threads - 2 clients, no DNSSEC

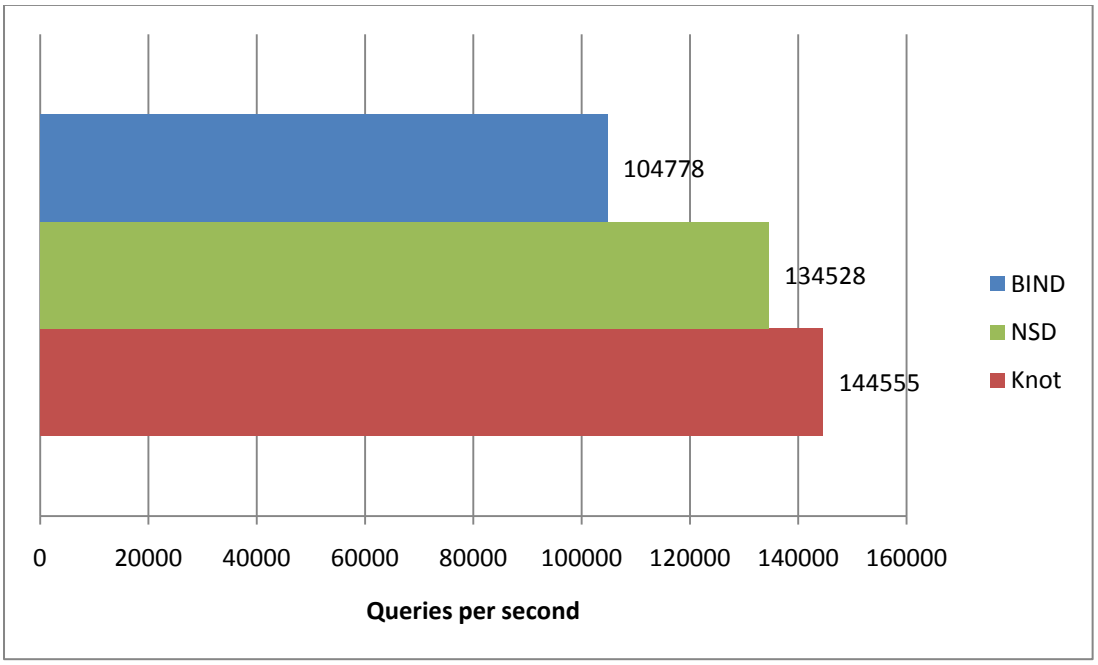


Figure 12 - Maximum reply rate - 2 clients, no DNSSEC

5.2.5 Scenario 4 – two clients, DNSSEC

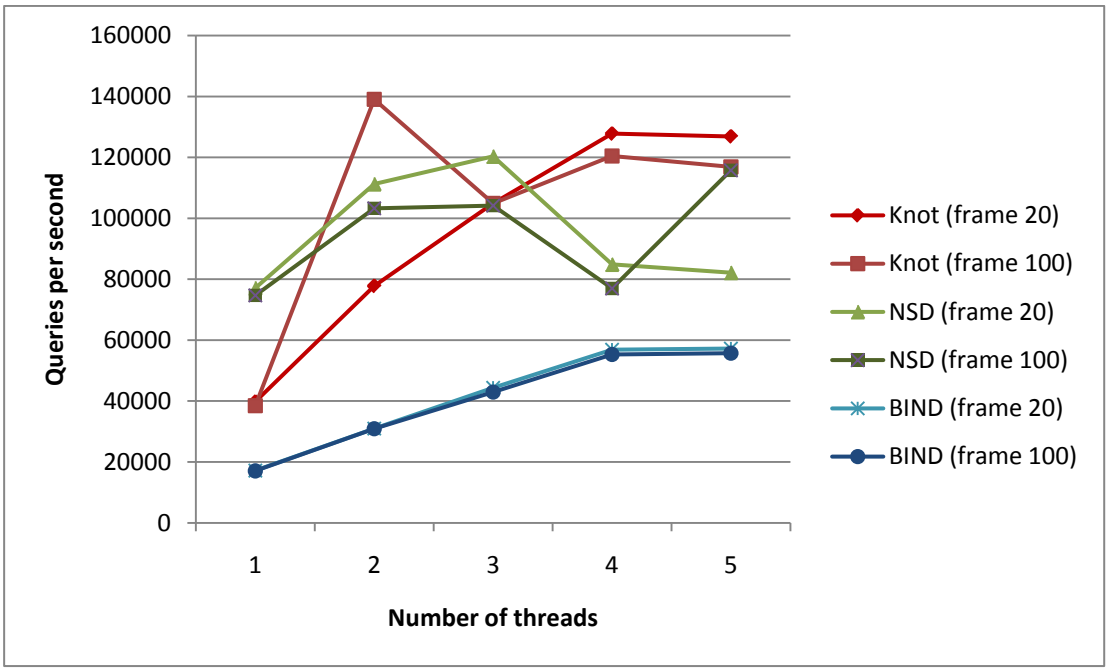


Figure 13 - Reply rate vs. number of threads - 2 clients, DNSSEC

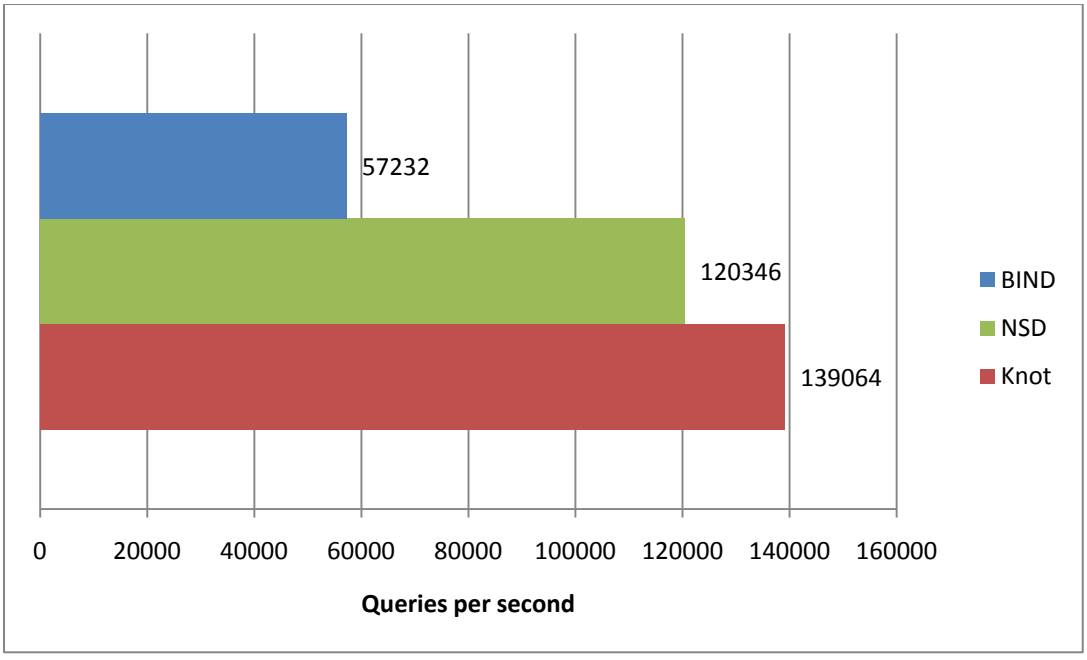


Figure 14 - Maximum reply rate - 2 clients, DNSSEC

5.2.6 Interpreting the results

Let's first look at the more straightforward data – maximum reply rate. In all performed test the Knot server featuring our DNS library achieved the best results. In the case of one client and replies containing DNSSEC it outperformed NSD by 57% and BIND by whole 250%. However, when two clients were issuing queries, NSD was catching up quite well. As in real situations servers receive traffic from many clients, this trend suggests that the differences between NSD and our implementation would not be very significant in real life. BIND's performance still remained very low in this case, but without DNSSEC it was able to achieve quite satisfying results (around 100 000 qps).

Comparison of reply rate against number of threads used by the server yields some interesting results. You can clearly see from the charts that although BIND does not achieve the peak performances of NSD and our implementation, its reply rate increases linearly until the maximum is reached (in all cases it is with 4 threads). This could show a better and more balanced handling of traffic. It might be said that it is quite stable when it comes to the reply rate.

This is related with another data collected from the benchmarks, which are not shown in the charts – the percentage of unanswered queries. This became visible when the 'frame' for outstanding queries was set to 100 and the queries required DNSSEC. In these cases BIND achieved highest numbers, though in general the ratio was not very high. In the worst case 0.05 % of queries were marked as 'lost' by `dnsperf`. This may point to a conservative strategy of BIND's traffic handling causing it to rather drop queries instead of trying to answer them at all costs.

When comparing NSD's and my results, in both cases the reply rate fluctuated more than in the case of BIND. However, most of the time NSD's performance is best when less processes are used and decreases with more. This is a rather unwanted behavior and would suggest not using NSD's ability to fork more processes for normal answering. It may be however still useful for processing zone transfers or dynamic updates. More or more specialized tests may be useful as in other situations NSD may make better use of more processes.

Another interesting observation is that NSD achieved almost twice the performance of my implementation when it used just one process. One possible reason is that my library uses quite a lot of structures for each zone and the search is repeated when a domain name is not found in the hash table. It would be worth analyzing and finding way how to improve the performance of the library in this case.

The performance of my implementation showed probably the least regular behavior. This may be due to the chosen network-handling model (see section 5.1.2 above) which allows achieving peak performance for the cost of less predictability. Nevertheless, as already noted, the performance was superior in all cases. A quite interesting observation is that in several cases (and mostly when using higher number of outstanding queries) the best reply rate was achieved with only two threads, though the hardware setup (4-core CPU) would point to best results at four or five threads. This again seems to be more a property of the server rather than the library itself.

6 Conclusion

In this thesis I designed and implemented a DNS library intended for implementation of authoritative name servers. The goal was to provide a set of high-level functions for all important DNS-related functions of such servers, with support for all important DNS protocols and features, such as DNSSEC with NSEC3 or zone transfers. But most of all, the library should have been designed to achieve superb reply rates, matching or even surpassing the most used existing implementations.

I first analyzed the task, possible solutions and different ways of improving the performance. Afterwards I designed the main parts of the library, dividing it into logically and functionally separate modules and proposing architectural and design concepts chosen to achieve the goals. The implementation brought several more particular problems that had to be solved and the solutions were presented in the next part. Finally I tested and evaluated the library by integrating it into an existing experimental name server implementation. The performed benchmarks prove the achievement of the main goal – high performance of the library.

The design of the library is very clean and convenient. Where applicable, I separated ‘public’ and ‘private’ functionality so that each module as an API used by the other modules so that the implemented structures are nearly never used directly. The most high-level API provides convenient functions for processing different types of requests that the name server may encounter. Integration into server software was trouble-free and very straightforward.

On the other hand, the support for some features could be better or more tested. The version of the server available for my testing was not yet supporting incremental zone transfers (IXFR), so this feature, though implemented in the library, could not be properly tested and debugged. On the other hand, dynamic updates are not supported by the library at all, although it was designed and should present no significant task to implement. A support in the server would be needed for testing this feature as well.

Another drawback of the library is its high memory usage. Runtime zone structures in memory are approximately five times larger than the original size of the zone (size of the text zone file). While some implementation (such as NSD) do also have high

memory requirements, ours are even worse. This may be a problem mainly when dealing with very large zones and using full zone transfers. In case of AXFR there is no way to avoid duplicating the data in memory but in our case the overhead is very large.

7 Future work

Although the library achieves superb performance and its design is quite advanced, there are still ways in which it may be improved further. The most important would probably be to finish and test support for IXFR and dynamic updates or even to add new features (such as NSID). If the library was to be used in a production-grade name server, a more extensive testing would be definitely required, including a testing operation so that it may be evaluated with real data.

Another task may be to improve the memory requirements of the library. There are probably ways of reducing the size of the structures. The implementation is in many ways redundant (e.g. zone data are kept in several parallel structures). However, it must be noted that better memory requirements are often redeemed by worse performance so this is a purely design choice and some servers may take advantage of the superior performance even with the high memory consumption.

A less significant improvement, but still worth reviewing, are different options for the main data structure. One possibility may be to replace the Cuckoo hashing scheme with a Hopscotch scheme which was claimed to achieve superior performance in some cases due to better cache-locality of the stored data (17).

The performance of the library is another issue worth pursuing. Although the results are quite satisfying, there may be ways to improve it even more, particularly in case when only one thread is used.

8 Bibliography

1. Chapter 8 - SOA Record. *ZYTRAX*. [Online] [Cited: August 4, 2011.] http://www.zytrax.com/books/dns/ch8/soa.html.

2. **Arends, Roy, et al.** *RFC 4034 - Resource Records for the DNS Security Extensions.* 2005.

3. **Pagh, Rasmus and Roder, Flemming Friche.** CiteSeerX - Cuckoo Hashing. *CiteSeerX*. [Online] 2001. [Cited: July 10, 2011.] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.4189.

4. *Dynamic perfect hashing: Upper and lower bounds.* **Dietzfelbinger, Martin, et al.** 1994.

5. **Fotakis, Dimitris, et al.** CiteSeerX - Space Efficient Hash Tables with Worst Case Constant Access Time. *CiteSeerX*. [Online] 2003. [Cited: July 10, 2011.] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.5337.

6. **Dietzfelbinger, Martin and Weidling, Christoph.** *Computer Science Department at Princeton University.* [Online] 2007. [Cited: 7 10, 2011.] http://www.cs.princeton.edu/courses/archive/fall09/cos521/Handouts/BalancedAllocation.pdf.

7. **Kirsch, Adam, Mitzenmacher, Michael and Wieder, Udi.** CiteSeerX - More Robust Hashing: Cuckoo Hashing with a Stash. *CiteSeerX*. [Online] 2008. [Cited: July 10, 2011.] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.152.685.

8. **Dietzfelbinger, Martin, et al.** A Reliable Randomized Algorithm for the Closest-Pair Problem. *Journal of Algorithms.* 1997, 25.

9. FNV Hash. *Landon Curt Noll.* [Online] [Cited: August 4, 2011.] http://www.isthe.com/chongo/tech/comp/fnv/index.html.

10. A Hash Function for Hash Table Lookup. *Bob Jenkin's Web Site.* [Online] [Cited: August 4, 2011.] http://www.burtleburtle.net/bob/hash/doobs.html.

11. Pluto Scarab - Hash Functions. [Online] [Cited: August 4, 2011.] http://home.comcast.net/~bretm/hash/.

12. Copy-on-write. *Wikipedia, the free encyclopedia.* [Online] [Cited: August 4, 2011.] http://en.wikipedia.org/wiki/Copy-on-write.

13. Read-copy-update. *Wikipedia, the free encyclopedia.* [Online] [Cited: August 4, 2011.] http://en.wikipedia.org/wiki/Read_copy_update.

14. **McKenney, Paul E.** RCU vs. Locking Performance on Different CPUs. [Online] [Cited: August 4, 2011.] http://www.skytel.co.cr/linux/research/acrobat/040116.pdf.

15. **Elz, Robert and Bush, Randy.** *RFC 2181 - Clarifications to the DNS Specification.* 1997.

16. **Austein, Rob.** *RFC 5001 - DNS Name Server Identifier (NSID) Option.* 2007.

17. Hopscotch hashing. *Wikipedia, the free encyclopedia.* [Online] [Cited: August 3, 2011.] http://en.wikipedia.org/wiki/Hopscotch_hashing.

18. **Pfaff, Ben.** Performance Analysis of BSTs in System Software. *Stanford University.* [Online] 2004. [Cited: July 17, 2011.] http://www.stanford.edu/~blp/papers/libavl.pdf.

19. **Surý, Ondřej.** February 23, 2011.

20. **Mockapetris, Paul.** *RFC 1034 - Domain Names - Concepts and Facilities.* 1987.

21. **Vixie, Paul.** *RFC 1996 - A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY).* 1996.

22. **Ohta, Masataka.** *RFC 1995 - Incremental Zone Transfer in DNS.* 1996.

23. **Vixie, Paul, et al.** *RFC 2136 - Dynamic Updates in the Domain Name System (DNS UPDATE).* 1997.

24. **Vixie, Paul.** *RFC 2671 - Extenson Mechanisms for NDS (EDNS0).* 1999.

25. **Arends, Roy, et al.** *DNS Security Introduction and Requirements.* 2005.

26. **Arends, Roy, et al.** *RFC 4035 - Protocol Modifications for the DNS Security Extensions.* 2005.

27. **Laurie, Ben, et al.** *RFC 5155 - DNS Security (DNSSEC) Hashed Authenticated Denial of Existence.* 2008.

28. **Lewis, Edward and Hoenes, Alfred.** *RFC 5936 - DNS Zone Transfer Protocol (AXFR).* 2010.

29. **Mockapetris, Paul.** *RFC 1035 - Domain Names - Implementation and Specification.* 1987.

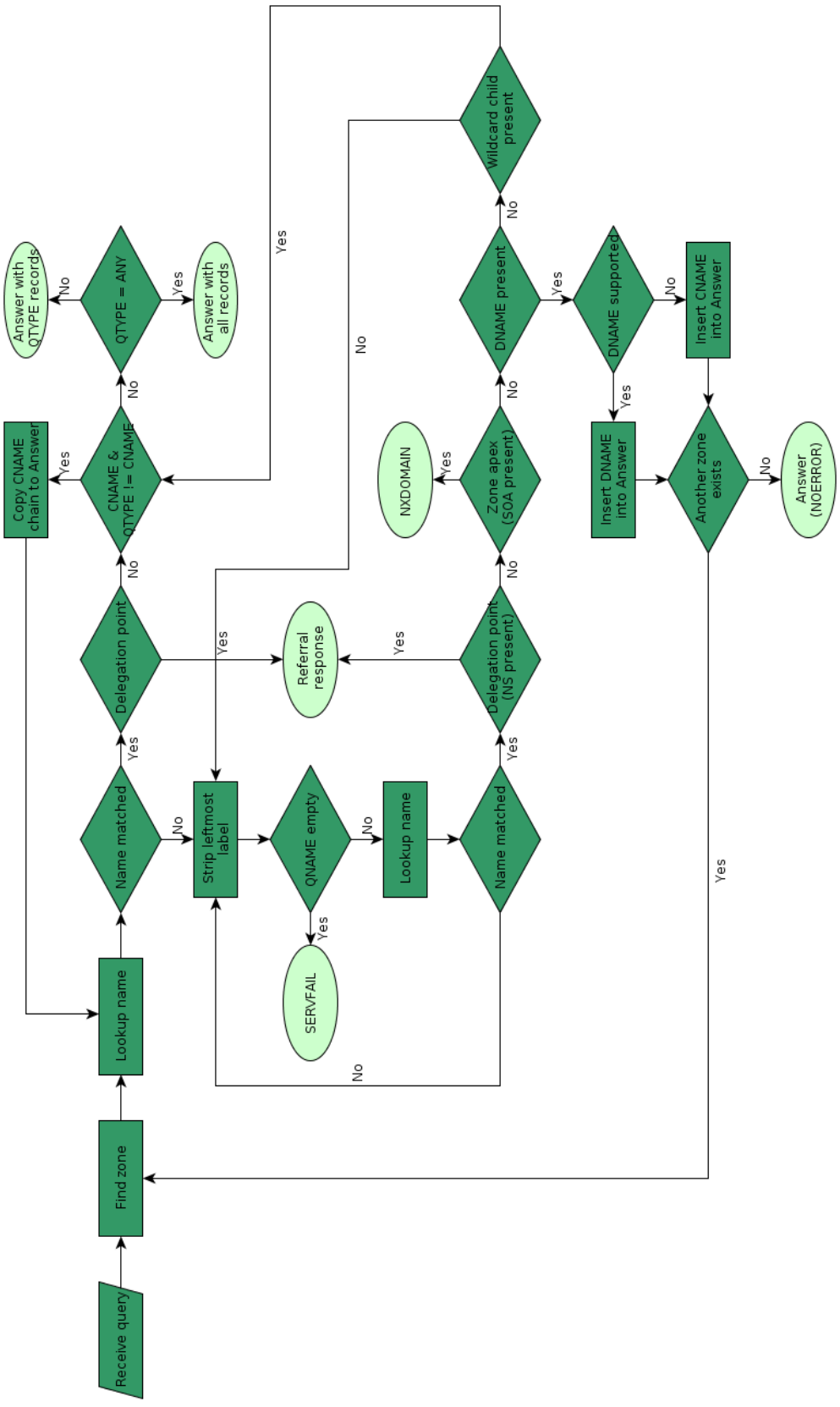
30. DNS related RFCs. *DNS, BIND Nameserver, DHCP, LDAP and Directory Services.* [Online] [Cited: August 4, 2011.] http://www.bind9.net/rfc .

31. BIND 10 Development. [Online] [Cited: August 4, 2011.] http://bind10.isc.org/.

32. DNS for Rocket Scientists. *ZYTRAX*. [Online] [Cited: August 3, 2011.] http://www.zytrax.com/books/dns.

Table of figures

Appendix A – Query processing decision graph



Appendix B – Contents of the CD

- `sources/`

  Source files of the library.

- `thesis.pdf`

  This document in electronic form.

- `developer-documentation.pdf`

  Developer documentation

- `reference/`

  Reference documentation