Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

DNS Zone Transfer Test Tool

Bachelor's thesis

Daniel Hubáček

Bc programme: Open Informatics
Branch of study: Artificial Intelligence and Computer Science
Supervisor: Doc. Ing. Jiří Novák, Ph.D.

Prague, May 2021

Thesis Supervisor:
Doc. Ing. Jiří Novák, Ph.D.
Department of Measurement
Faculty of Electrical Engineering
Czech Technical University in Prague
Technická 2
160 00 Prague 6
Czech Republic

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, May 2021

.....
Daniel Hubáček

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Hubáček Daniel** Personal ID number: **483726**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

DNS Zone Transfer Test Tool

Bachelor's thesis title in Czech:

Testování transferu doménových zón

Guidelines:

Design a primary DNS server, which generates specific changes in a zone and propagates these changes into a secondary server (server under test). Each zone change should focus on a possibly critical scenario, which could be processed incorrectly on the secondary DNS server side and therefore different zone states could have come into existence on particular servers managing the same zone. Finally, the server should verify the correctness of the zone transfer result and analyze the network load.
Abstract design and modularity are also important aspects of the resulting software. It should be possible to easily create new test cases, extensions and to perform any feasible scenario.

Bibliography / sources:

[1] Mockapetris Paul – RFC 1035: Domain names – Implementation and specification – Information Sciences Institute, 1987
[2] Ohta Masataka – RFC 1995: Incremental Zone Transfer in DNS – Tokyo Institute of Technology, 1996
[3] Vixie Paul – RFC 1996: A Mechanism for Prompt Notification of Zone Changes – Information Sciences Institute, 1987
[4] Lewis Edward – RFC 5936: DNS Zone Transfer Protocol (AXFR) – NeuStar, Inc., 2010

Name and workplace of bachelor's thesis supervisor:

doc. Ing. Jiří Novák, Ph.D., Department of Measurement, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **19.12.2020** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

doc. Ing. Jiří Novák, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Abstract

The DNS represents one of the backbone technologies of the current internet. For latency, stability, and safety issues, it is distributed over multiple servers. With every distributed system, it is important to have a reliable mechanism of synchronization, when some changes happen. In the DNS, it is a mechanism called *zone transfer*. In order to preserve safety and stability, it is important for this mechanism to work correctly and therefore it has to be thoroughly tested. One approach of testing zone transfers of DNS servers is to test the whole functionality. It means propagate some zone changes to the server and then check whether or not have been these changes correctly processed. Because particular implementations of DNS servers may suffer from different bugs, it is more efficient to have just a testing tool or environment, using which it is possible and simple to create and run any test scenario. The development of such a tool is described in the thesis and also one server implementation, *knot*, is tested by this software.

Keywords: Domain Name System, zone transfer, testing, distributivity, KnotDNS

Anotace

DNS je jednou z páteřních technologií dnešního internetu. Pro zaručení stability, bezpečnosti a nízké odezvy jsou tyto systémy distribuovány přes více serverů. U každého distribuovaného systému je ovšem důležité mít spolehlivý proces synchronizace dat v případě jakýchkoliv změn. V DNS se tento proces nazývá *zone transfer* (přenos zóny). Pro zachování bezpečnosti a stability je důležité, aby tento proces přenosu zóny fungoval bezchybně a proto musí být řádně otestován. Jeden z přístupů k testování přenosů zón je testování celé funkcionality jako takové. To znamená propagovat nějaké změny v zóně do testovaného serveru a následně ověřit, že všechny požadované změny byly správně zpracovány. Jelikož různé implementace DNS serverů mohou obsahovat různé chyby, je efektivnější mít celý testovací nástroj či prostředí, pomocí kterého je možné a jednoduché vytvořit a spustit jakýkoliv testovací scénář. Vývoj takového testovacího nástroje je popsán v této práci a následně je tento nástroj také využit k otestování jedné implementace DNS serveru, *knot*.

Klíčová slova: systém doménových jmen, přenos zóny, testování, distributivita, KnotDNS

Acknowledgements

I would like to express my sincere thanks to all the people who helped and supported me during my Bachelor's studies, especially the ones who stayed close to me during writing the final thesis. Firstly, I must thank my friends and family for helping me in deciding important matters and caring about my occasional distraction from the hard work and studying. To Ondřej Švec, who gave me a huge help in the field of technical writing and LaTeX language. To Marko Uusitalo, M.Sc from the Metropolia University of Applied Sciences, who made it possible for me to write the thesis abroad and provided me great technical facility. To Doc. Ing. Jiří Novák, Ph.D. for professional counsels and consultations while writing the thesis. I also sincerely thank Ing. Petr Špaček, who initiated me into the world of DNS and was the perfect guide during the implementation.

List of Figures

List of Acronyms

AXFR DNS Zone Transfer Protocol. 8, 9, 11–15, 17, 20, 21, 25, 28, 31–33, 38, 52, 54, 55

DDoS Distributed Denial of Service. 7

DNS Domain Name System. 1, 3–9, 11, 12, 16–18, 20, 21, 23–26, 28, 29, 31, 36, 38–40, 45, 46, 51–54

DoS Denial of Service. 7

EDNS Extension mechanisms for DNS. 12

GIL Global Interpreter Lock. 38

HTTP Hyper Text Transfer Protocol. 6

IDN Internationalized Domain Names. 34

IP Internet Protocol. 3, 51, 52

IXFR Incremental Zone Change. 8, 13–15, 21, 31, 32, 42, 50, 51, 54

LDAP Lightweight Directory Access Protocol. 1

MX Mail Exchange. 5, 49

NS Name Server. 46, 49

OOP Object Oriented Programming. 23, 24, 40

RFC Request for Comments. 5

RR Resource Record. 12

RRset Resource Record set. 26, 31, 38, 55

SOA Start of Authority. 7, 8, 13–16, 20, 21, 26, 27, 31, 32, 38, 41, 42, 44, 48–50, 53, 54

SQA Software Quality Assurance. 18, 24

TCP Transmission Control Protocol. 11, 21, 28–30, 52

TTL Time To Live. 10, 14, 33, 39, 41, 46

UDP User Datagram Protocol. 11, 21, 52

XFR AXFR and IXFR. 17, 20, 24, 27, 49

Contents

Chapter 1

Introduction

The demands on computing resources are becoming higher and higher, as are the requirements for network infrastructure. Lots of systems are based on mutual communication between more devices (computers, smartphones, watches, sensors, and so on) and lots of systems are also based on cooperation between multiple different services (for example web applications, telephone exchanges, email servers, or Lightweight Directory Access Protocol (LDAP) servers). It is important to keep some sort of order between those stuff and make it more readable and intuitive for society.

The Domain Name System (DNS) represents one of the backbone technologies of the current internet. It makes the communication between individual services and devices much easier for people to manage. For latency issues, DNS servers are usually distributed over more places around the world (or some smaller particular places), and therefore some management issues appear. Every modification must be applied to all the servers to preserve the system stable and faultless. For this purpose, a mechanism called *zone transfer* has been designed.

There are many DNS server implementations (for example *BIND9*, *PowerDNS* or *KnotDNS*) and all of them have to implement this zone transfer mechanism according to the specification. Incorrect processing of a zone transfer might lead to unstable results and therefore it needs to be properly tested.

Every piece of code should be covered by some unit tests, but it is useful to also thoroughly test the mechanism as a whole, as it works in the real operation. Testing such a complex mechanism is not simple, because there are many cases that could go possibly wrong and also because simulating a real operation requires many side utilities. Basically, it is needed to create a *fake*, testing, DNS server, which pretends to be a primary server, which tries to propagate some changes into another, tested, secondary server. These pseudo-changes could be just random or may also aim at a particular weak spot of the software. After the changes are applied, it is also needed to check the resulting zone version of the tested secondary server and evaluate the success.

Individual products (in the meaning of the DNS server implementations) differ in the implementation and therefore different products may suffer from different bugs. Once a bug is found and fixed, it should be also covered by tests. It means that every DNS server implementation

needs its own test scenarios and it would not be really efficient to create one universal testing tool for everything. Much more efficient is to create a tool, or a testing environment, using which it would be quick and easy to create and run any possible scenario.

Chapter 2

Basics of DNS

This thesis is dealing with the distributivity of DNS servers, specifically with a process of zone changes transfer, where changes made on a primary server must be also replicated on secondary servers.

The goal is to create a testing DNS server, which is generating some zone changes and subsequent propagation of these changes to a secondary server. Each change should focus on a possibly critical scenario, which could be processed incorrectly on the secondary DNS server side. In the end, this testing primary server verifies the correctness of the zone transfer result and analyzes the network load.

This chapter describes the basics of DNS - what are domain names, why and how did they come into existence, and why are they actually so useful. Then, more advanced issues about distributivity are described - why is it so important, how is it solved, and which troubles can occur. Details and specifications of zone transfer protocols are described at the end of this chapter.

It should be a sufficient introduction into the world of DNS, which is necessary for understanding the issues described in the thesis.

2.1 Origin of DNS

Computer networks can contain tens, hundreds, or millions of interconnected devices. Nowadays it is not just about computers, but also cameras, televisions, watches, fridges, light bulbs, vacuum cleaners and so on. Almost every piece of electronics can be designed or made to communicate with its surroundings and together with other devices, they create networks of any size. The connection between smartwatches and a mobile phone, internet search engine, both are based on a communication between some devices.

To be possible to send some messages between particular devices, these messages must be addressed to someone. Therefore every device must have an address within a network. In case of Internet Protocol (IP), it can be an IPv4 address, for example *192.168.0.99*. Using an address, every device can determine whether or not does a message belong to it and if the device should send the message somewhere else or just ignore it. These addresses are simple to work with, for

computers. It is hard to even remember for people.

That is the reason why devices started to be named with more human-readable names. One simple file has been created and this file contained pairs of name-address. The computer always went through this mapping file and found an address ("readable" for computer), which corresponded to a name (readable for humans).

Note that this file can be still found on today's devices. It is stored in `/etc/hosts` for UNIX systems or in `%SystemRoot%\system32\drivers\etc\hosts` for Windows. It is used as a static translator of domain names with the highest priority.

In the beginning, when every network was composed of a few devices, it was a sufficient solution. Changes in networks were rather rare, it could be managed by one person or there was a mechanism, which synchronized this file with another, remote, file - a database. But this solution was not sufficient for a long time. There were more and more devices in particular networks and these networks started to be also mutually connected. This *file domain system* was not sufficient anymore, it was unbearable to manage so large networks with this primitive system.

In 1983, an American computer scientist Paul Mockapetris proposed in RFC 882 [1] and RFC 883 [2] a DNS protocol, which was supposed to suit all requirements and solve all the problems, so primarily fast management of resource records - adding and removing. Four years later, in 1987, DNS protocol has been updated in RFC 1034 [3] and RFC 1035 [4] into the form, which is still in effect now - after more than 30 years. After a huge increase of computational resources, number of devices, and also expectations of users, this protocol is still sufficient.

2.2 How does DNS work

The fundamental block of domain names is the hierarchy. The domain *fel.cvut.cz* can be used as an illustration. Dots in this domain name represent individual parts or labels. It is quite intuitive that the first label, *cz*, stays for the Czech republic. Second label, *cvut*, represents the Czech Technical University (**Č**eské **V**ysoké **U**čení **T**echnické), and the last label, *fel*, represents the faculty of electrical engineering (**f**akulta **el**ektrotechnická). There is the hierarchy, where every label somehow makes the previous label more specific. And this hierarchy is present even in the core of the DNS, as it is described in this chapter.

This whole hierarchy of domain names can be understood as a tree, wherein every vertex there is one part of the domain (for example *cz*) and from this vertex then go edges to other, subordinate, vertexes (for example *cvut.cz*) and other subordinate vertexes (*fel.cvut.cz*). So every vertex has its own sub-tree, which contains all the sub-domains. On the very top of this imaginary tree there must be one more vertex, which would roof all the first-labels, so-called *top-level domains*, as is *cz* or *sk*, *com*, *net* and so on. This very first vertex is called *root*. The imaginary hierarchical tree is illustrated in the Figure 2.1.

Note that this root vertex can be explicitly inserted into a domain name by adding one dot at the end - *fel.cvut.cz.*. After the last comma, it is possible to imagine an empty string, which represents exactly this root. Domain names without the ending dot are considered as *relative*

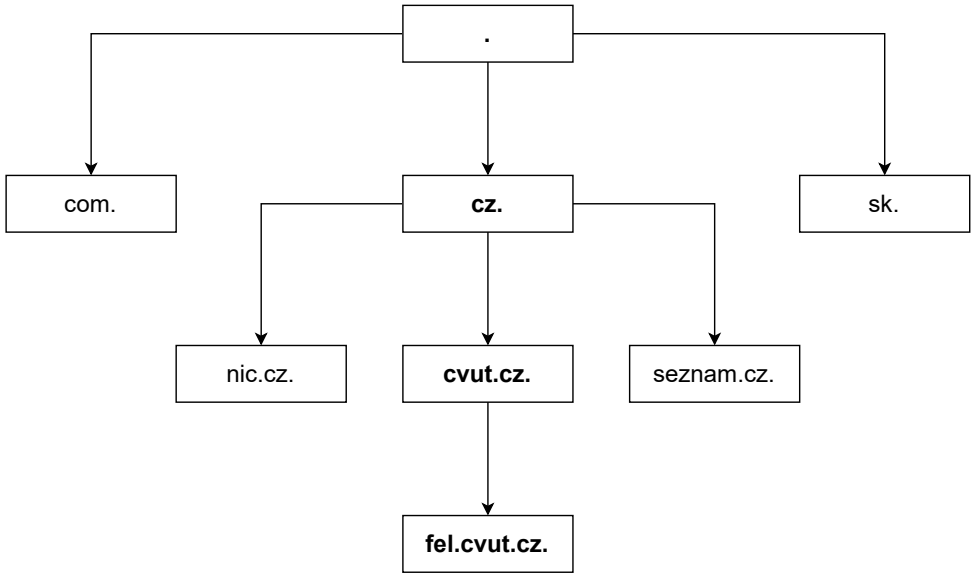

Figure 2.1: DNS hierarchy example with *fel.cvut.cz*

to a context. It is used mostly in configuration files.

Every vertex contains (next to the domain name) also some resource records. One of them is for example the IP address, or information about email services, server applications, contact on the domain administrator, and so on. Actually, it is possible to insert here any information. Every type of information has some sort of identifier, *rdtype* (resource data type). For example, IPv4 addresses are called A records, IPv6 addresses are AAAA records, mail servers have Mail Exchange (MX) records, SRV for service locators, PTR for domain name pointers (useful for reverse lookups - find a domain by an address), NS for authoritative name servers, or TXT for any other text data. There are many other rdtypes defined in different Request for Comments (RFC) documents, according to their usage. In all messages, these rdtypes have 2 bytes reserved, so it is a number from the interval 0..65535, which is large enough for all the types of records.

This distribution of domain names to some vertexes of an imaginary tree is actually a really strong approach, which allows delegating all the management of particular sub-trees to particular, independent, servers. The whole top-level domain *cz* can be managed completely independently from other top-level domains, moreover, it can delegate a part of itself (for example *cvut*) to another server. This delegation mechanism can be recursively repeated over and over again.

Every domain, every vertex, contains some information (in the form of records) about itself and the rest about its subordinates delegates (or to be exact - can delegate) to another server, without any dependencies on itself or its predecessors. Every vertex has a DNS server. This server has all the records of a domain and some records of some sub-domains or can delegate some sub-domains to another server. This imaginary part of the hierarchical tree, which is managed by the DNS server, is called a *zone*. So a zone is a part of a sub-tree, part of the whole DNS space.

The whole DNS space is divided into zones and every server can delegate a part of a zone to another server. And this is exactly the way, how it works during resolving (resolving is a process

when a resolver looks at some particular records of a domain). The resolver asks a server and receives either the answer or a hint, where to ask. When resolving the *fel.cvut.cz* domain, the first query goes to a root. The root does not know the answer, but knows, where to find the *cz* domain. The second query goes to the *cz* domain, but neither this server knows the desired records, so it answers with a hint where is *cvut.cz*. So the third query goes to the *cvut.cz*, which answers with a hint about *fel.cvut.cz* and finally, this server knows the answer, so it answers with the IPv4 address - with the A record. This process is illustrated in the Figure 2.2 (shortened).

```
user@ntb:~$ dig +trace fel.cvut.cz
cz.           172800  IN  NS  c.ns.nic.cz.
;; Received 622 bytes from 198.41.0.4#53(a.root-servers.net)

cvut.cz.      3600    IN  NS  nss.cvut.cz.
;; Received 309 bytes from 194.0.14.1#53(c.ns.nic.cz)

fel.cvut.cz.  6400    IN  NS  lux.fel.cvut.cz.
;; Received 302 bytes from 147.32.1.9#53(nss.cvut.cz

fel.cvut.cz.  36000   IN  A  147.32.192.12
;; Received 114 bytes from 147.32.192.250#53(lux.fel.cvut.cz)
```

Figure 2.2: Resolving process of *fel.cvut.cz*.

This process might seem to be quite long and annoying. Therefore there are DNS resolvers, which do this work and return just the final answer. These resolvers are usually pre-installed in all operation systems. Also, all the records can be cached, so it is not necessary to execute this whole recursive process every time.

2.3 Distributivity of DNS

It has been already explained, that DNS was created for the possibility of naming some devices in a readable, easily memorable, and meaningful way. It has been also explained how does resolving work and how does domain (or zone) management work. Or almost. There is still one thing, which is important in computer science and it is speed!

When a computer is located in Prague and tries to resolve the *fel.cvut.cz* domain, it is all right and fast. But when a computer is located on the other side of the world, it takes some more time.

One of the important aspects is the distance from a DNS server. It is obvious, that the best location of a server managing the *cz* domain is the Czech Republic and also that servers of the Czech Technical University should be somewhere in Prague. The overwhelming majority of clients is from the Czech Republic so the average latency is minimized. If all the mentioned servers were only in the Czech Republic, the latency on the other side of the world would be significant and even increased by the number of queries, which has to be actually sent - query for *cz.*, *cvut.cz.*, *fel.cvut.cz.* and also the final Hyper Text Transfer Protocol (HTTP) query of a web browser. All 4 messages would go all over the world and back, therefore the latency would

significantly increase. And this latency might be increasing linearly with every other label, with every other sub-domain.

Latency from a server can be reduced only by a shorter path between the client and the server, or by upgrading the communication technologies. It is not really possible to move the client nor the server, and it is not really possible to upgrade for example a metal wiring to optics. But it is possible to duplicate the server to somewhere else. Somewhere closer to other clients. Then, when a resolver asks a root server for the servers of *cz* domain, the root server could return a list of servers and the resolver might choose the best one of them (the most reliable and fastest one based on some statistical data).

Another important advantage of server duplication is stability. When, for some reason, a server becomes unavailable, there are other servers with all the important information (all the resource records) to provide the service. The latency would be higher, but the service would still work, which is also a great way, how to prevent some types of attacks, such as Denial of Service (DoS) or Distributed Denial of Service (DDoS), when the attacker tries to flood a victim with tons of requests so it is not possible to answer to other, regular, clients.

It can be noticed that in this situation (when a zone is distributed over multiple servers) there are multiple servers, from which the very same behaviour and very same answers are expected. The zone administrator is supposed to manage multiple servers. It is actually the same problem, which also had the simple system with one *hosts* file before - the need to replicate a database across multiple servers. But DNS has a solution for it.

Servers of a zone are either *primary* or *secondary*. A primary server is a server, which provides information about zone changes to other servers. A secondary server is a server, which asks another server about these changes. These definitions are not contradictory, which means that one server can be primary and secondary at the same time:

Let have three servers `A`, `B` and `C`. A zone administrator makes some changes in the server `A`. This server propagates the changes into the server `B`. The server `B` applies the changes and furthermore, it propagates the changes even to the server `C`. So the server `B` was the one, who accepts the changes and at the same time provides changes to another server.

This variant is valid. When the servers are configured properly it is needed to perform the changes only in one server and they are thereafter propagated even to other servers.

2.4 Zone replication between servers

In the first place, there is one record type named Start of Authority (SOA). It is a record, which must be present in every zone exactly once. Its value contains some important information about the whole zone - email address of a responsible administrator; serial number of the zone (kind of a zone version); an amount of time, when a secondary server should ask for zone changes; an amount of time, after which the server should try to contact the primary server again in case the primary server had not answered; an amount of time, after which the server should stop providing the answers, because they may not be in force anymore.

Already the SOA record contains some information, which determines when should a sec-

ondary server ask for changes and how should the secondary server behave. But this *refresh* time can be long and the primary server may not be available at the time when the secondary server is asking for the changes. So the secondary server may not provide current answers quite a long time after a zone has been officially changed. Another situation would be when the DNS servers are logically connected in a chain (as exampled above with the `A`, `B` and `C` servers). It would take some time until the changes would be propagated to the end of this chain, so the answers would differ depending on which server is being queried.

That was the motivation while designing a mechanism named *DNS NOTIFY*, which is described in the RFC 1996 [5]. It is a simple way, how to indicate new zone changes to a secondary server. When a DNS server updates its content (straightly by an administrator or automatically according to a primary server), it sends a Notify message about it to all its secondary servers. It is expected that these secondary servers are going to update their version of the zone as well, so they have actual resource records and the whole zone remains stable all over the servers.

It is important to be aware of the fact that reception of a Notify message does not necessarily mean a need for the zone refresh. Imagine a situation, when a secondary server has two primary servers. Both of them send a Notify message, but one message arrives later for some reason. Meanwhile, the secondary server has received the Notify message from the other primary server and updated its zone. When the delayed Notify message is received, there is no need to update the zone anymore since the zone version is actual.

The naive implementation is to pull the whole zone every time a Notify message is received. It is certain that the zone is always correct. The transfer of the whole zone follows the DNS Zone Transfer Protocol (AXFR), which is described in the RFC 5936 [6]. But as it has been described, the zone might have been already updated and therefore the zone transfer might be completely superfluous.

A small improvement of the naive implementation is to check the zone version (the serial number of the SOA record) first. A simple query for the SOA record is sent to the primary server. If the received SOA record has a greater serial number than the SOA record of the secondary server, a zone transfer should be executed.

Zones might be really large, so a transfer of a whole zone might take some time and also might excessively load down the network. For illustration, 1. 1. 2021 the top-level domain *cz* contained 1 370 749 registered domains (source: `https://stats.nic.cz`). It is highly ineffective to transfer over 1 million records after an addition of one new record, just to replicate the zone. That was the motivation for the Incremental Zone Change (IXFR) protocol, which is specified in the RFC 1995 [7].

Incremental zone transfer consists only of the changes between particular zone versions. If one new domain is added into a zone, the incremental zone transfer contains exactly this (new) record. The rest of the (unchanged) records is not included in the message, which saves some computational time and also reduces the network load. But even this way of transferring zone changes has some disadvantages, which might not be really obvious.

To be able to send only the changes between particular zone versions, it is needed to know

these changes. So the server must have some sort of a *log* (often called *journal*), which contains all the information about the last few changes. A secondary server might not be available at the time (for example because of some blackout, attack, and so on) and it might ask for changes after a longer time, after a few more zone changes. If the primary server does not remember all the incremental changes anymore, it is forced to send the whole zone via AXFR.

And also, the application of the incremental changes may not be processed totally correctly, which would have fatal consequences! There would be two different zone versions, which would be considered the same, and answers on queries might be strongly inconsistent depending on which server is being asked. These errors are hard to find or debug, and moreover - the server is not able to detect them. The only two situations, how to miraculously fix this error, are:

1. Other incremental changes in all affected records, which would the secondary server be able to apply.

2. Pull the whole zone via AXFR. But that is what the secondary server is trying to avoid.

It is a trap, which is not easily detectable and the server is not really able to fix it by itself. Therefore it is important to implement the zone transfers perfectly, especially the incremental changes application, and this implementation must be thoroughly tested.

2.5 Details of the zone transfer protocols

Like every other communication protocol, even DNS has its own format for messages, which has been described in the RFC [4]. The message is divided into 5 sections - header, Question, Answer, Authority and Additional. Not every section has to contain some data. In certain cases, some sections, except the header, might remain empty. It is fully dependent on the type of the message, what information is the message supposed to carry. The zone transfer is essential for this project, therefore this type of message is going to be described more deeply in this section.

Firstly, it is useful to describe the header which is the same for all the messages. It consists of 6 parts, where each part is 16 bits, so in total it is 96 bits long header section.

The first part of the header contains an ID using which it is possible to recognize answers to a particular question. The ID in an answer must be the same as in the question corresponding to the answer. In questions or queries, there can be any number. It is up to the client, which ID will be chosen to distinguish particular answers.

The second part of the header is reserved for flags and codes. Each flag and code is deeply described in the RFC [4]. At this point, most of the header flags are not important and therefore their explanation will be skipped.

The last 4 parts of the header section contain numbers (unsigned 16-bit integer) of entries (resource records) in the corresponding sections (Question, Answer, Authority, Additional) of the message.

The message header format is displayed in Figure 2.3.

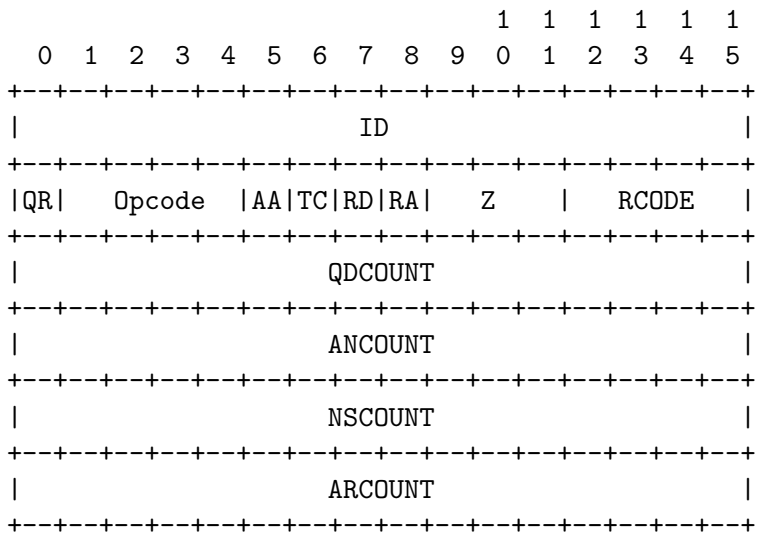

Figure 2.3: DNS message header. [6]

After the header section follow the Question, Answer, Authority, and Additional sections. According to the RFC [4], the Question section is slightly different from the others. It says that every entry (there is usually only one) contains a domain name, a type, and a class. This triplet could be understood as some kind of key to the specific records.

The other three sections extend this triplet with two other attributes. The first one is the Time To Live (TTL). It is a 32-bit unsigned integer that says for how long time (in seconds) can be the records cached. The second attribute is, of course, the data itself. There can be multiple values that are just stacked behind each other and together they create a set of record values.

Formats of these sections are represented in Figure 2.4 (for the Question section) and Figure 2.5 (for Answer, Authority and Additional sections). Notice how do these two formats differ. The beginning is exactly the same, the Figure 2.5 just extends the Figure 2.4 by the TTL and the data (length of the data and the data itself), as described above.

Note that the letter Q in the Figure 2.4 stands for *question*. The values are still equivalent to values in the Figure 2.5.

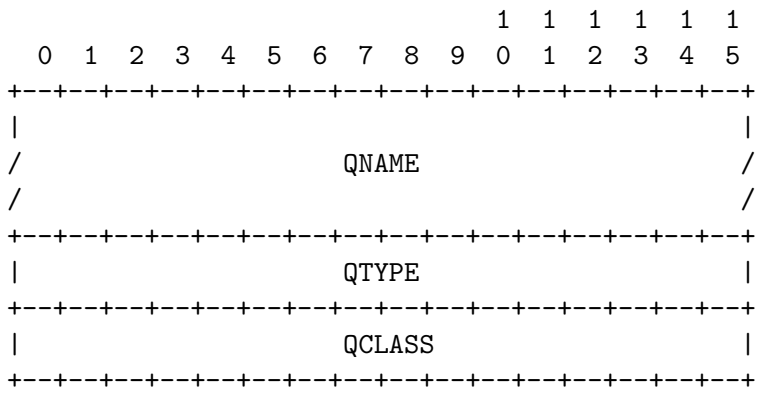

Figure 2.4: Format of the Question section. [6]

At the very end, every message is compressed. This mechanism is supposed to reduce the size of the final message, so the network load is not that high. This mechanism is based on

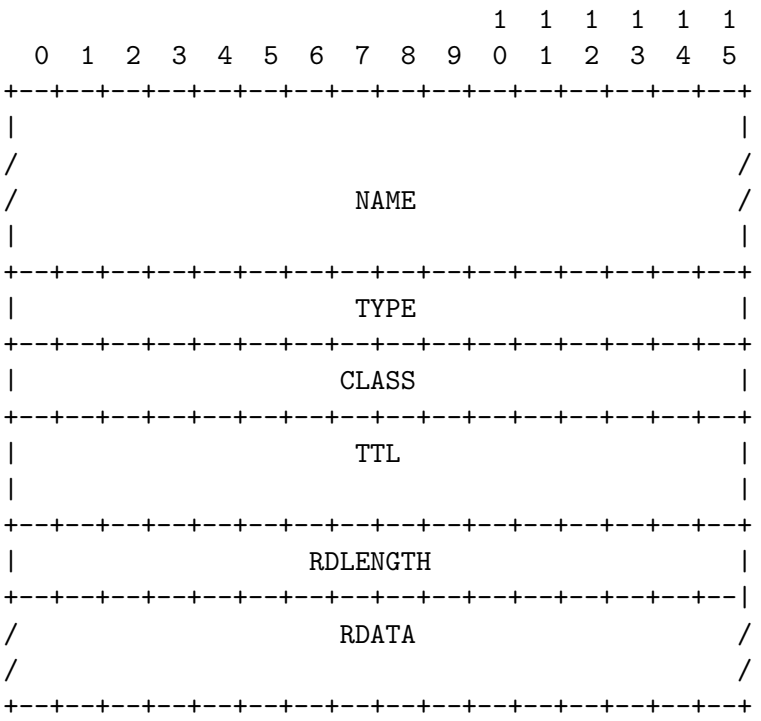

Figure 2.5: Format of the Answer, Authority and Additional sections. [6]

replacing parts of domain names with pointers to a prior occurrence of the same name. So every domain name is written in the message only once and every other use of the same domain name, or its sub-part, is written only as a pointer on the first occurrence with the full wording. Since every domain name can be up to 253 characters long, this mechanism has the potential to save a lot of space.

Message compression is not essential for the purposes of this project, but, as it will be mentioned later, it is important to keep in mind that something like this happens and mainly that it happens at the very end when all the content in all sections is known.

Usually, the DNS messages in general are not so long and therefore the User Datagram Protocol (UDP) is used. For example, when resolving an address, questions contain only a desired domain name and answers contain a few records. On the other side, zone transfers usually consist of more records. Tens, hundreds, thousands or even more. That is the reason why the Transmission Control Protocol (TCP) is mostly used for zone transfers - for its reliability and stability.

2.5.1 AXFR messages

The most essential part of zone transfers is the AXFR described in RFC 5936 [6]. This protocol enables to transfer the whole zone at once. It means all records are transferred in one message, or in one set of messages (in case the content is too long for one message, the size limit is 65535 bytes), so the receiver has the current version of the zone.

As explained previously, all DNS messages share the same format and differ mostly in the content. Also, messages distinguish whether they are queries (questions) or responses (answers). For this purpose, there is the QR flag in the header of the message.

The AXFR query message is sent by a client whenever the zone transfer might be needed and in compliance with the specification the header must contain the following flags:

- **QR** must be set to 0 as an indication that it is a Query.
- **OPCODE** must be set to 0 as an indication that it is a Standard Query.
- **RCODE** must be set to 0 as an indication that there is no error.
- **QDCOUNT** must be set to 1 as an indication that in the Question section there is just 1 entry. This entry describes the question (the triplet of domain name, class, and type).
- **ANCOUNT** must be set to 0 since the Answer section must be empty
- **NSCOUNT** must be set to 0 since the Authority section must be empty
- **ARCOUNT** must be set to the number of records in the Additional section. This section can contain up to 2 records: Extension mechanisms for DNS (EDNS) and DNS transaction security. In case of the testing, this section will be mostly empty and therefore the ARCOUNT will be set to 0.

More important is then the Question section, which must contain exactly 1 entry - the triplet of domain name, class, and type. The domain name and class are obvious - they contain the key values which the secondary server is asking for. The type is not that obvious since the secondary server does not want just one exact type of records, but all of them. Therefore there is a pseudo Resource Record (RR) type for this kind of zone transfer, the AXFR type with the value of 252.

In case of the response, the header again follows the general specification for DNS messages:

- **ID** must be copied from the request.
- **QR** must be set to 1 as an indication that it is a response.
- **OPCODE** must be set to 0 as an indication that it is a Standard Response.
- **AA** must be set to 1 in case there is no error (and therefore RCODE is set to 0). Otherwise, the value must be set according to the rules for that error code.
- **RDCODE** must be set to 0 in case there was no error. Otherwise, this flag should be set to the appropriate value (for example 9 in case the server is not authorized to provide this information or 1 when the query message has been malformed).
- **QDCOUNT** must be set to 0 or 1, further explanation will be provided later.
- **ANCOUNT** must be set to the number of messages sent in the Answer section.
- **NSCOUNT** must be set to 0 since there are no records in the Authority section.
- **ARCOUNT** must be set to the number of records in the Additional section, which has the same rules as for the query message.

In the first place, it is important to keep in mind that one AXFR response might consist of multiple messages. And the rules for the Question section differ depending on whether or not is the message the first one sent. The Question section must be copied from the query in the first message. In subsequent messages, this section may either remain empty or be copied from the query as well.

The Answer section contains the zone content itself, all the resource records. But the form of this section must comply with few rules. The very first and the very last record must be the SOA record of the zone. Thanks to this rule it is possible to identify the last message of the transfer. Not only would the secondary server not know if the connection could be terminated, but there would also be no guarantee that the zone transfer contained all the records and therefore that it was the entire zone.

The resource records between those SOA records represent a set, or unordered collection, of records. Although servers usually attempt to send related records as a continuous group, they are not required to do so. However, these records must not contain any SOA records since the SOA record is already at the beginning and at the end. Violating this rule might result in a zone transfer collapse or, in the worse case, in incorrect processing of the zone transfer and therefore the existence of more different zones versions.

The AXFR message is pretty simple, as it can be seen in the example in the Figure 2.6. The zone *nic.cz* has serial number 1 and contains two A records and one MX record.

```
            +-----------------------------------------+
Header      | OPCODE=QUERY, RESPONSE                  |
            +-----------------------------------------+
Question    | QNAME=NIC.CZ., QCLASS=IN, QTYPE=AXFR    |
            +-----------------------------------------+
Answer      | NIC.CZ.            IN SOA serial=1      |
            | NIC.CZ.            IN A   127.0.0.1     |
            | NIC.CZ.            IN A   127.0.0.2     |
            | NIC.CZ.            IN MX  1000 nic.cz.  |
            | NIC.CZ.            IN SOA serial=1      |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 2.6: AXFR message example.

2.5.2 IXFR messages

The IXFR protocol (described in RFC 1995 [7]) is an enhancement of the previous AXFR. In the AXFR messages, plenty of superfluous records are sent since the secondary server might have already had them. In case of one little change (like the addition of one single record), everything has to be transferred again even though the only important part might be just one single record. And that is what this new protocol tries to solve.

The simplest possible description of the IXFR is that the message contains only the changes between two zone versions (version is given by the SOA serial number).

The header of a query message does not differ from the AXFR format. The first difference can be found in the Question section, where the entry does not have the AXFR (= 252) type, but IXFR type with the value of 251. The domain name and class remain unchanged.

Because the incremental changes represent a difference between two states, an old one and a new one, it is a necessity to have the first state (identified with the SOA serial number) from which it is supposed to be started. Therefore in the query message, it is required to have the SOA record, from which the incremental changes will be listed. This record is inserted into the Authority section. Note that because there is a record in this section, this information has to be reflected even in the header and therefore the NSCOUNT number must be set to 1.

The server has actually two options for how to answer the query. For some reason, the incremental zone transfer might not be available (for example the client's version is too old and the primary server does not remember all the changes anymore). In this case, the primary server can send the AXFR response (it means the whole zone) with one small difference - the Question section is still copied from the query, therefore in the Question section entry the IXFR type (= 251) remains unchanged even though it is being responded with the AXFR. The fact that the response is not in the form of incremental changes can be ascertained from the form of the Answer section, as described later.

The second, and most preferred, option is to send the incremental changes. These changes between two zone versions can be described as a set of removed records together with a set of added records. Even a change of one particular record (for example change in the value or the TTL) can be represented as remove and add operation - just remove the old record and add a new, changed, one.

These two sets of records need to be inserted into the Answer section in a form that the secondary server can distinguish which records are supposed to be added and which of them are supposed to be removed.

As described in the RFC [7] and as in the AXFR response, the very first and the very last record must be the actual SOA record of the zone so it is possible to determine the end of the transfer (mostly when it takes more than one message). More important is the content between these SOA records.

Firstly, there is the old SOA record, which is supposed to be removed. Then follows the set of records which is also supposed to be removed. Next comes the new SOA record followed by the set of records to be added.

This one step, this one increment (old SOA, removed records, new SOA, added records) represents the changes between two versions of a zone. And this increment can be repeated more times in one zone transfer message. So a change from version n to version $n+2$ can be expressed as either two increments $n \rightarrow n+1 \rightarrow n+2$ or one direct compressed increment $n \rightarrow n+2$.

An example with two increments, $1 \rightarrow 2 \rightarrow 3$, is showed in the Figure 2.7. During the

changes from 1 to 2, one A record is removed and two are added. During the changes from 2 to 3, one A record is removed (the one which had been added in the previous increment) and another one is added.

```
            +-----------------------------------------+
Header      | OPCODE=SQUERY, RESPONSE                 |
            +-----------------------------------------+
Question    | QNAME=JAIN.AD.JP., QCLASS=IN, QTYPE=IXFR |
            +-----------------------------------------+
Answer      | JAIN.AD.JP.          IN SOA serial=3    |
            | JAIN.AD.JP.          IN SOA serial=1    |
            | NEZU.JAIN.AD.JP.     IN A   133.69.136.5 |
            | JAIN.AD.JP.          IN SOA serial=2    |
            | JAIN-BB.JAIN.AD.JP.  IN A   133.69.136.4 |
            | JAIN-BB.JAIN.AD.JP.  IN A   192.41.197.2 |
            | JAIN.AD.JP.          IN SOA serial=2    |
            | JAIN-BB.JAIN.AD.JP.  IN A   133.69.136.4 |
            | JAIN.AD.JP.          IN SOA serial=3    |
            | JAIN-BB.JAIN.AD.JP.  IN A   133.69.136.3 |
            | JAIN.AD.JP.          IN SOA serial=3    |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 2.7: Example of IXFR message with more increments. [7]

This incremental zone transfer can be compressed to a message with just one increment *1* $\rightarrow$ *3*. The very same situation, as in the previous example, is showed in the Figure 2.8, but this time there is just one increment containing one record to remove and two records to add. The record from the previous example, which was added in the version 2 and removed in the version 3, completely disappeared since it was not present in the original version 1 and neither is in the new version 3. The other records really changed between the compared versions and therefore remained in the message.

Notice two important things. The IXFR response can be either the AXFR or IXFR. Also, in the AXFR Answer section it is strictly forbidden to have any SOA records in the middle (in the meaning of between the border ones) while in the IXFR they have the important role of breakpoints.

2.5.3 DNS NOTIFY mechanism

Secondary server can ask his primary server for zone transfer at any time. It can be right after the start, regularly on daily basis, after the `EXPIRE` countdown from the SOA record (this attribute gives the number of seconds after which secondary servers should stop providing the answers for the zone) and so on. But the secondary server itself does not know when is the right time for asking for changes and possibly for the zone transfer.

The primary server is able to notify the secondary server when any change happens. Every

```
            +-----------------------------------------+
Header      | OPCODE=SQUERY, RESPONSE                 |
            +-----------------------------------------+
Question    | QNAME=JAIN.AD.JP., QCLASS=IN, QTYPE=IXFR  |
            +-----------------------------------------+
Answer      | JAIN.AD.JP.          IN SOA serial=3       |
            | JAIN.AD.JP.          IN SOA serial=1       |
            | NEZU.JAIN.AD.JP.     IN A   133.69.136.5   |
            | JAIN.AD.JP.          IN SOA serial=3       |
            | JAIN-BB.JAIN.AD.JP.  IN A   133.69.136.3   |
            | JAIN-BB.JAIN.AD.JP.  IN A   192.41.197.2   |
            | JAIN.AD.JP.          IN SOA serial=3       |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 2.8: Example of IXFR message with one increment. [7]

change is expressed via increasing the SOA serial number and every time this happens, the primary server can send a special message to the secondary server as a notification. The secondary server is not obliged to take this message into consideration, which means that the server is not obligated to ask for changes and the zone transfer immediately. Nevertheless, the secondary server is still obligated to answer this notification as an indication to the primary server that this server knows about the latest happenings.

This mechanism is called A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY) and is specified in the RFC 1996 [5]. The first message is sent by the primary server which has just made some changes in a zone and therefore increased the SOA serial number. This message is characterized by:

- Header QR flag is set to 0. It can be understood as a request for keeping up-to-date.
- Authoritative Answer (AA) flag should be set to 1 since the primary server is the source of information.
- The OPCODE must be set to 4, which is the number reserved for this purpose.
- The Question section contains one entry with the domain name, class, and the SOA type (= 6).

Moreover, this message can also contain the new SOA record in the Answer section as an insecure hint for the secondary server. The secondary server is free to use this hint in the meaning of skipping the zone transfer, because it may have been already done using another primary server. However, this hint is optional from the side of the primary server.

As mentioned earlier, the secondary server is still obligated to answer this request. The response is the exactly same message with the only one difference and it is that the QR header flag is set to 1 this time.

Chapter 3

Requirements and expected functionality

It is important to be conscious of what are the goals of the software. It is not a goal, and it is not even possible (as mentioned later), to create software that would test just everything, every possible scenario and every possible problem. The goals are:

- to prepare a work-space or environment using which it will be possible to easily and efficaciously implement and run any particular test scenario.
- to prepare some basic set of test scenarios covering well-known vulnerabilities.
- to make it easy for future use or other programmers to extend the software and add new test cases, since some DNS servers might be suffering from product-specific vulnerabilities.

Using the Section 2.5 it is possible to describe the goal of the software more precisely: The final software should be answering on AXFR and IXFR (XFR) requests with some configured or random content and it should be also capable of checking whether or not have all the responses been correctly processed by the tested secondary server. The secondary server zone version can be obtained again via the AXFR protocol. To speed up the testing, the software is also supposed to send DNS NOTIFY messages to indicate new zone changes, which should convince the tested secondary server to begin with the zone transfer every time when new changes in a zone are generated.

The software itself must also meet some requirements, which describe its structure and design:

- The possibility to reproduce every possible scenario which could ever happen. All the correct scenarios, in the meaning, that the messages could really happen in the real-life operation, and the incorrect scenarios as well, in the meaning, that the messages should be rejected by the tested server.
- Make test scenarios implementation as easy as possible. It is not correct if it is needed to write hundreds of lines of code just to create one simple scenario.

- The ability to configure independent modules and create a wider range of possible test scenarios. The DNS protocol is very complex and also, as described before, there are more ways how to perform a zone transfer. But usually, one way does not differ from another that much. The difference can be just the protocol, which has been used, the content of the DNS NOTIFY message, which had been sent before the zone transfer, or an unexpected answer in the form of an AXFR response. All these methods should be somehow configurable so it is easy to combine them in the final scenarios. Said with different words, the components should be as much independent as possible. They should be modular in the way that it should be easy to change or extend them.

- The ability of the final software to be extended. The product will be used by programmers and testers (testers in the meaning of Software Quality Assurance (SQA) engineers) and it will help them to ensure that their product, their DNS server implementation, is working properly. If they find a bug or a weak spot in their particular product, they might want to create their own test scenario for it so next time the problem is covered by tests. It is required to make this interaction for other programmers simple and elegant.

- It is mandatory the software must be easy to run. It is expected that one function takes some configuration parameters and the whole testing process will be executed automatically according to the parameters.

- At the end the software will verify the correct processing of all the zone changes by the secondary server.

- The software will collect basic statistics about the network load, which can be then used to compare different implementations.

It is important to keep in mind, that the software is supposed to **be able to reproduce** any scenario, not really reproduce all the scenarios, which could ever happen. It is not even possible since there is an infinite number of combinations (proof: one record can be always added to the zone and so to a zone transfer message). Moreover, the point is not to do a brute force testing but to test the most common, most important, and meaningful scenarios.

Two significant aspects are not very important for the software - time and memory complexity. The software does not need to be really fast or memory efficient. It will not be used as an ordinary program for clients who are waiting for a response. It will be used only for testing purposes, as a utility to ensure the correctness of processing zone transfers. Therefore it is not needed to focus on time or memory complexity, but rather on the abstract and modular design so that the software satisfies all the requirements which have been mentioned so far.

3.1 Test example

For a better understanding of what is supposed to happen, the simplified process of testing is shown in the Figure 3.1. Except for the last part, where the final check happens, it is the same as a normal operation in everyday life.

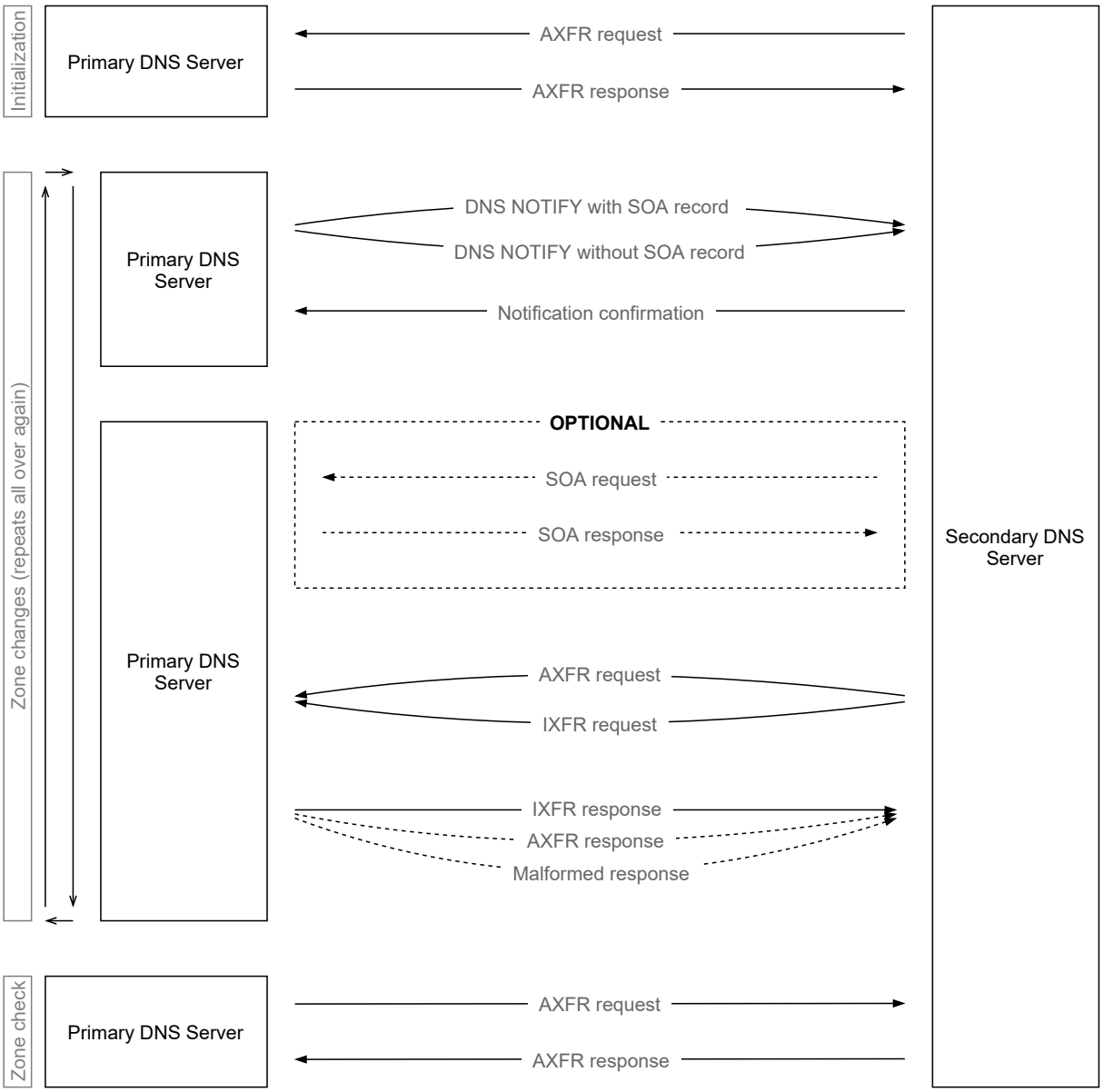

Figure 3.1: Workflow diagram.

Everything starts on the side of a secondary server. This server knows from its configuration, that there is a zone and that there is also a primary server of this zone. But that is all, at the moment only the information that *something exists* is known. No exact records are known yet. Therefore, the secondary server needs to get these records by sending an AXFR request to the primary server. The primary server answers with the zone content and since this moment the secondary server has everything that is needed for serving data. This process is represented in the *Initialization* section in the Figure 3.1.

Note that for security reasons the primary server can refuse to provide the AXFR response. Usually, secondary servers must be listed in the configuration and AXFR requests are refused unless the configured list of secondary servers contains the questioner. The same applies for the DNS NOTIFY messages described later. A server follows these messages only if the message came from an authorized source listed in a configuration.

Everything works perfectly until a zone change is made. When this happens, the primary server sends a DNS NOTIFY message to the secondary server. The secondary server must confirm the receipt of this notification and then it is supposed to start the zone transfer.

Optionally, a SOA request can be sent first to make sure there really are new changes for the particular secondary server. Then, since there is already an old zone version on the side of the secondary server, a request for incremental changes can be sent. Also, for some reason, the AXFR protocol can be chosen for the transfer.

In both cases, the primary server is supposed to answer with the corresponding answer. Also, an error can potentially happen. The secondary server has to deal with that cannot make any zone change.

This is how the workflow of DNS distributivity really works. Every time a change in the zone occurs, this process is executed all over again.

At the end of the testing, it is needed to check the zone content of the secondary server and that is represented in the *Zone check* part in the Figure 3.1. The primary server gets the secondary server's zone via the AXFR protocol and compares it with the expected version of the zone.

As explained earlier, the testing software is supposed to test the processing of the zone transfer messages and therefore the main part is generating the responses on XFR requests. A simple test scenario can be for example the following:

Create a zone with 10 default A records. Then make 9 changes where one record is removed. Propagate these 9 changes into the secondary server and check that at the end the secondary server's zone really contains just the one and only last record. This scenario is showed in the Figure 3.2.

Or a little bit advanced scenario might be to create a zone with 10 default A records, then remove 5 of them and then try to add another 5 new records. But this time answer with a malformed message. It can be just cut in the middle or it can contain incorrect values in the header (for example one bit in the header is reserved and must be 0 every time). After this, check the secondary server's zone version and make sure it contains just the five records. This

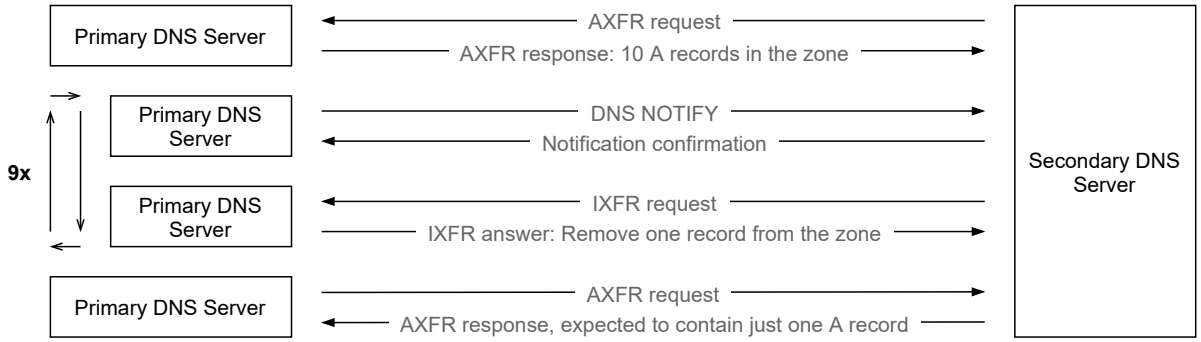

Figure 3.2: Diagram of a test example.

scenario is showed in the Figure 3.3.

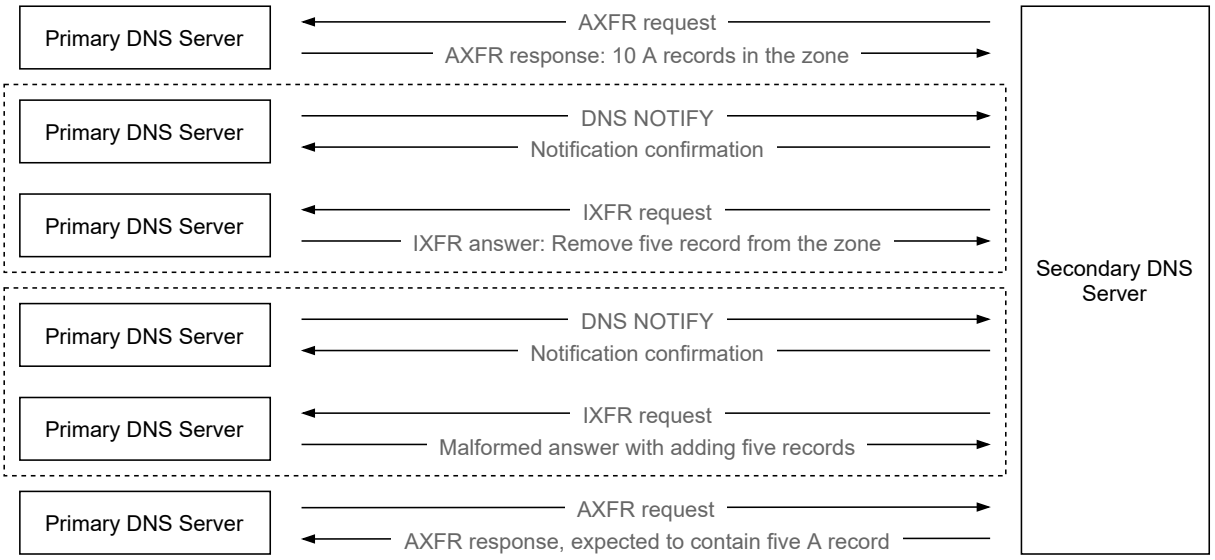

Figure 3.3: Diagram of an advanced test example.

But these scenarios are widely generalized, the exact case can depend even on other details - the DNS NOTIFY messages might (not) contain the SOA record, it might answer with AXFR response on IXFR request, it might (not) answer with some malformed messages, used protocol can be one of TCP or UDP, and so on. The design of the software should make it as simple as possible to adjust all these details and therefore create a much wider variety of test cases.

Chapter 4

Implementation

First of all, the right development tools must be chosen. There are many possibilities, especially in the large selection of programming languages. Moreover, some programming languages have other libraries and packages, which might make some parts of working with DNS much easier.

The basic programming language is `C`. `C` has the big advantages, that it is very fast and memory efficient. Also, there are some open-source projects for working with DNS, which could be possibly used. But the management of `C` projects and files is not really simple and big attention must be paid while compiling. Because `C` is a low-level language, it would be really hard to satisfy the requirement that scenarios implementations should be easy to make. It means some higher-level approach for the DNS messages is needed and this approach would have to be created.

Small improvement brings the `C++`. This language is as fast and memory-efficient as `C`, but gives Object Oriented Programming (OOP) approach, which is very useful while designing, and implementing, modular systems. Because `C++` is compatible with the `C` language, the same open-source extensions can be used and even some new `C++` libraries could be used. But still, the `C++` is also too low-level language, which makes it hard to create higher-level approaches and it also needs to be compiled first.

Another option would be `Java`. It is a nice compromise between low-level and higher-level programming languages. It has all the low-level features for working with bytes, wide variety of structures to use, it is also fast and memory-efficient (but not as much as `C` or `C++`) and it also has some open-source projects for working with DNS. Moreover, it is a class-based and object-oriented programming language, which allows creating modular systems very easily. It has a *garbage collector*, which takes care of memory management. `Java` code also needs to be compiled before running, but there are tools like `Maven`, which make the dependencies management and subsequent build very simple. It sounds really great so far, but `Java` has one big disadvantage and that is the code length. Its programs tend to be long and this does not correspond with the requirement that *it is not correct if it is needed to write hundreds of lines of code just to create a simple scenario.*

The last and the best programming language, which has been taken into consideration and also chosen for development, is `Python` (`https://www.python.org/`). It is not as fast and

memory-efficient as all the previous options, but it is not really important for testing purposes, as mentioned earlier. On the other side, it has a really powerful packaging system offering great libraries for working with the DNS, for example `dnspython` (`https://www.dnspython.org/`), which has been used during the development. The code does not need to be compiled and can be run directly via an interpreter, which is easily accessible for all operating systems. `Python` provides great OOP approach, which makes it easy to design modular systems and enables simple extensions of classes.

Moreover, `Python` is a great language for functional programming. Writing code in that way is not really easy and requires experience in this field, but some aspects of functional programming might be used throughout the project, such as calling a function with some arguments always returns the same result and cannot be affected by any mutable state or other side effects. This aspect is also demanded since the very same test scenarios (the very same messages, the very same sequences of bits) must be reproducible to ensure a bug has been fixed.

During the development, `Python 3.6` and `dnspython 2.0.0` have been used. The software is also suitable for `Python` versions up to `3.9` and `dnspython 2.1.0`.

4.1 Project structure

As explained before, testing requirements and preferences may vary from product to product. Therefore the main goal is to create an environment where it will be simple and fast to adapt all details of a specific scenario with a minimum of work. Also, the software must be really modular so every particular independent module can be exchanged to widen the scale of possible test scenarios.

The project itself is divided into a few independent parts in the way that it abides by the requirements explained in the Section 3 (Requirements and expected functionality) and with a good abstraction over creating the XFR messages described before. Each part is fully modular - every module has just an abstract interface that has to be implemented and which is then used for testing purposes. The internal implementation may vary depending on the objectives of the testing scenario and this is what makes the abstract design of the software so powerful.

The first part cares about creating the DNS NOTIFY messages. Since there are more ways how to construct them, a special module called `NotifyManager` has been dedicated for this purpose.

Then there is a module, `NetworkAnalyst`, which cares about the analysis of network load. Every tester (in the meaning of programmers and SQA engineers) might need different information to check the improvement and difference between two versions of their software and therefore this is also dedicated into a special module, which can be then configured.

The testing is based on a communication between two servers - the testing software in the role of a primary server and the tested software in the role of a secondary server. This communication can be distinguished by the initiator of the communication. Therefore two modules have been designed for these purposes:

1. The Client, which handles all the communication, which is initiated by the primary server. It contains sending the DNS NOTIFY messages or sending AXFR requests for the zones comparison.

2. The Server, which handles all the incoming requests from the secondary server and answers on them. Moreover, this module has to be running in a separated process, as explained later.

The last, and the most important, component is the module that generates all the test scenarios and decides which data will be answered by the `Server`. These answers are constructed using a `ResponseManager`, which basically takes some records and puts them into a byte-message.

All together, these independent components can make the whole testing happen. The workflow is shown in the Figure 4.1. `TestCase` generates some changes in a zone in the form of a `ResponseManager` instance and wants to propagate them into the tested secondary server. By `NotifyManager` a Notify message is constructed and then sent to the secondary server via the `Client` class. The secondary server is supposed to register this notification and send a zone transfer query to the `Server`. In the generated `ResponseManager`, a response message is constructed and via `Server` sent back to the tested secondary server.

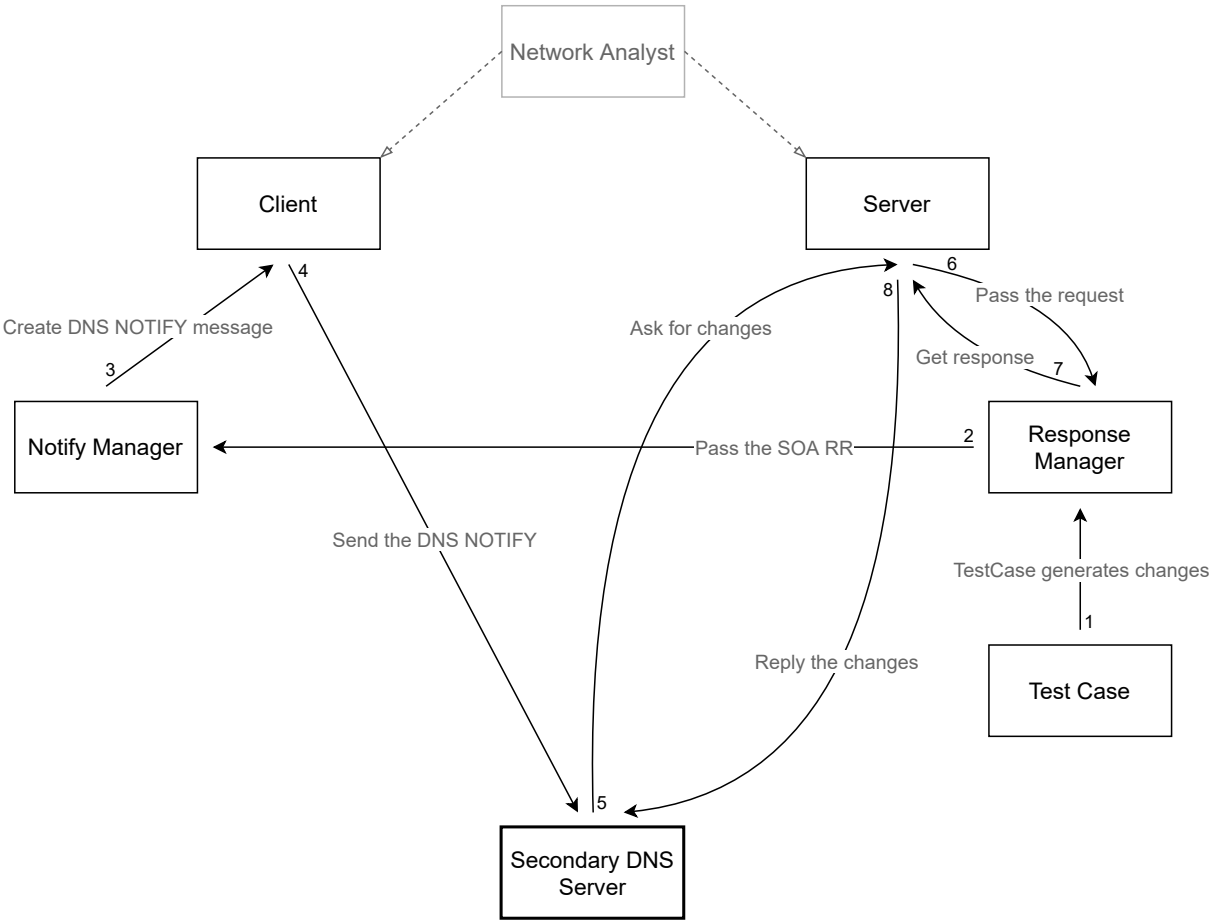

Figure 4.1: Components diagram.

This process might happen all over again until the `TestCase` would be exhausted. In the

end, the `Client` gets the zone content from the secondary server which is then compared with the expected zone content received from the `TestCase`.

The `NetworkAnalyst` does not have any straight impact at any point of this process but is still present in the `Client` and `Server`, so all the statistics are computed during the communication.

4.1.1 Notify manager

The task for every Notify manager is to construct the DNS NOTIFY messages according to the specification [5], which is then sent by the `Client` to the secondary server.

Since there are more possibilities for how to correctly create the Notify message (or Notify request) in the testing software, there is its own module for this purpose. The `NotifyManager` abstract class. In its constructor, it takes the domain name as the only argument. The reason is that the Question section has to contain one entry, where the domain name is required.

This class has also one abstract method and it is `get_notify_message(self, soa_rr: RRset)`. It takes one argument and that is the Resource Record set (RRset) containing the SOA record of the zone, which can be possibly added to the message.

The return value is the message, which is supposed to be sent as the notification. A message (here and even in general throughout this project) is understood as a bytes-form message or an instance of the `dns.message.Message`, which gives a higher-level approach over the messages and also has the `to_wire()` method, which returns the bytes-form message, which is then sent.

The project contains one notify manager class named `DefaultNotifyManager`. It is constructed in the way that it can create both of the most basic Notify messages - either with and without the hint in the Answer section. In the constructor, it takes another two arguments and those are the probability of inserting the hint into the message together with a `Random` instance.

The probability argument is obvious and does not need to be explained. The `Random` instance is provided because of the possibility to generate the exactly same text scenario all over again. This method is widely used throughout the project and is based on controlling the pseudo-random aspects in random decisions via a seed. A seed is a base number that is passed into the `Random` class constructor and the instance then generates the same sequences of numbers every time. Using this trick it is possible to preserve the functional approach over the testing while adding some random aspects.

This default notify manager covers three mostly used types of generating the Notify messages. The one where the hint is always present (when the probability argument equals 1), the one where the hint is never present (when the probability argument equals 0), and a mix of those two, where some messages do contain the hint (with some probability) and some of them do not.

The naive condition for adding the hint into the message would be writing something like `if add_soa_probability > rand.random()`. This can be optimized for the mostly used values, 0 and 1. In those cases, the random number does not have to be computed at all, and therefore it is possible to reduce some computational time by a simple condition which checks whether or not does the `add_soa_probability` equal to 0 or 1.

4.1.2 Network analyst

Network analyst is an independent and tangential component used for the network load analysis. Its instances are passed to the `Server` and `Client`, where it collects all needed data for the statistics.

Every message from the secondary server has to be captured and properly registered. At the end of testing, it is possible to create a summary of the network load and the tested software can be then optimized to minimize the load according to these results.

For this purpose, there is a component named `NetworkAnalyst`. This class takes two important arguments in its constructor. The first one is just a title of the instance and is used to distinguish the results of multiple instances in the final summary. The second argument is a set of analyst modules. These modules are the most important in the whole analysis. The `NetworkAnalyst` just groups them together and offers a simple interface for working with all of them at once - it somehow broadcasts method calls into method calls of all the modules.

`AnalystModule` is a simple unit for measuring one single quantity. Every message, which is received from the tested secondary server, is registered via `register_message(self, message: Message, time_received: float)` method and it depends on the particular module, which information does it take from the message. Moreover, there is also a `get_results(self)` method, which is supposed to return the computed statistics at the end of the testing in the form of a string so it can be printed on the standard output.

In most of the implementations, every module collects data only about one single unit, and this unit is then returned in the `get_results` method in the form of a string. Therefore the abstract `AnalystModule` itself creates a variable `self.stat_unit` already in the constructor and returns it in the `get_results` method. For most of the sub-classes, it is needed only to implement the `register_message` method. This method is usually pretty simple since it just collects some data. Most of them can be written just in one line. This makes it really simple to add other module implementations.

There are 5 module implementations. Two of them measure the time of the first and last message, one of them counts how many messages have been received, another one counts how many bytes have been received and the last one counts the message types. As the message type, it is meant the QTYPE of the first entry in the Question section. In the cases during testing, there are supposed to be only SOA and XFR queries and therefore this module is sufficient enough. Even if an unexpected query was received, this method would give sufficient enough results since most of the queries contain one entry in the Question section as well.

As it has been mentioned before, the `NetworkAnalyst` is just a tool for grouping these module instances. It also has the `register_message` method, which broadcasts the arguments to the modules, and it also has the `print_results(self)` method, which is supposed to print all the statistics to the standard output. The results are taken from the `get_results` module method.

This implementation has been chosen, because most of the statistical information is mutually independent and not every statistics information might be important for both `Client` and

`Server`. This way it is easily configurable and any other statistics aspect can be easily added.

4.1.3 Client

All the communication between this pseudo-primary server and a tested secondary server can be divided by the one who initiates the communication. There is a component, which is listening to all incoming connections and answers the queries, named `Server` (will be described in the Section 4.1.4). Also, there is a component, which initiates a connection with the secondary server and sends him a message. This component is named `Client`.

Reasons for the primary server in this project to initiate a connection are two:

1. Send a DNS NOTIFY message.
2. Send an AXFR request to check the zone.

The `Client` class takes two arguments in the constructor. The first one is the IP address of the secondary server in the form of two elements tuple with the address and the port. The second argument is an instance of `NetworkAnalyst`, which has been described in the Section 4.1.2. The client is supposed to register every received message via the `NetworkAnalyst.register_message` method. In the case of the AXFR response, it is a set of messages containing the whole zone, and in the case of the notification, it is the confirmation about receiving the message.

The first abstract method (abstract means that it has to be implemented in a subclass) is `send(self, message: BytesMessage)`. The purpose of this message is very simple - send a message passed as an argument, wait for the confirmation in the form of an answer and return the answer. This method is being used for sending the Notify messages generated by the `NotifyManager`.

Next method is `receive_axfr(self, request: BytesMessage)`. The only argument of this method is a message containing the AXFR request message. The client is supposed to send this message and after that receive all messages containing the zone. This list of messages is also returned. The way how to recognize whether or not all messages have been sent during the transfer has been explained in the Section 2.5.1 and it is the reason why a special method has been designed for this purpose.

The last method of the `Client` is the `stop(self)` method, which is being called at the end of testing. This method is not abstract and therefore does not need to be implemented in every subclass, but should be overridden or extended when a client implementation needs to do some cleaning, for example closing sockets. By default, it contains only the results printing of the network analyst instance.

The project contains one client named `TcpClient`. As the name indicates, this implementation uses the TCP for communication. Moreover, in the `dns.query` module there are two useful functions - `send_tcp` and `receive_tcp`. The main advantage of using those utilities is that they are working with instances of the `dns.message.Message` class, which provide a higher level approach over the message and therefore easier manipulation with details.

The TCP has been chosen for implementation primarily for the zone transfers reasons. With larger zones (or even smaller zones with large changes) the zone transfers consist of multiple messages, therefore TCP (as the safer and more stable protocol) should be preferred.

4.1.4 Server

The second component intended for the communication is the `Server` and it is supposed to handle all the incoming requests and answer them. The answers are taken from a `ResponseManager`, which is generated by a particular `TestCase`.

This module is basically very simple, but other circumstances make it seem like a complicated one - the server has to be running all the time so the tested secondary DNS server can connect to him at any time. This will be described later in the Section 4.2 about the project core, but for now, it is important to keep in mind that the software is supposed to be doing two things at once - act as a server and generate changes in a zone - and therefore it has to run in a separated process or thread.

The constructor of the `Server` takes 4 arguments:

1. A two elements tuple of an IP address and a port. The server is supposed to run on this address.

2. A state, which can be understood as a zone state. It is a variable, or a structure, which is shared between the two processes and using which all the generated changes are propagated to the server so it is able to provide actual data in its answers. By calling simple `state.get_responses(request)` the server receives all messages which are supposed to be sent as an answer.

3. An event, which is a multiprocessing synchronization feature. It could be compared to semaphores in the C language but works a little bit differently. It has just two values - whether or not is an imaginary flag set. By using three simple methods (`set`, `wait` and `clear`) is is simple to signalize some events to the other process and get perfect process synchronization.

4. The well-known `NetworkAnalyst` instance.

Also, in the constructor, the handler for `SIGTERM` is set. The `SIGTERM` means *signal termination* and signalizes that the server should be stopped, terminated. This signal is sent from the main process at the end of testing when there is no need to have a server running anymore. This handler just raises `SigtermException`, which is supposed to be caught (or excepted, in the language of Python) by the particular server implementation so the server can be stopped properly - close all sockets and print the analysis results.

The abstract `Server` class has 3 methods to be implemented - `start`, `run` and `stop`. During the run-time of the program, they are called in this very order. Neither of them accepts any arguments nor returns any value. In the `start` method, the server is supposed to prepare needed sockets bound to the address, which had been passed to the constructor. The `run` method should

contain the (infinite) loop for managing the queries received from the tested secondary server until the `SigtermException` is raised. And finally, in the `stop` method, the server is supposed to finish its job and properly stop itself.

As it was mentioned, in the constructor there is the `state` argument with its `get_responses` method. This method returns a tuple with two items - `responses` and `force_close`. The `responses` part is a list of messages to be sent as an answer. The `force_close` part (or flag) is a True/False value indicating whether or not should be the connection closed right after sending the last message from the `responses` list. Using this it is possible to pretend a blackout, server error, or any other unexpected circumstances. The tested secondary server must survive it (in the meaning of not terminating) and also the zone must remain correct.

As well as the `Client`, the project contains one server, which uses the TCP for communication - `TcpServer`. The reason is the same - zone transfers consisting usually from multiple messages and therefore TCP gives the certitude of successful transfer.

One more thing, which the server has to do after serving the responses, is setting the `event` flag. It works as a notification for the other process, that the zone transfer should have been done and therefore generation of another zone changes may begin.

4.1.5 Test scenarios

The last fully modular component is the `TestCase`. It configures and manages the whole zone, cares about its changes, determines which records should be present on the side of the tested secondary server, and so on. The simplest explanation would be that a test case just prepares some initial data (data in the meaning of default resource records of a zone), then generates some changes and at the end determines the expected set of resource records, which is then compared with the final resource records of the secondary server. But the design of this component is much more abstract, so the test cases can be way more complex.

The `TestCase` works mostly with the Response managers, which will be also described in this section. These Response managers are then shared with the `Server`, which uses them for receiving answers on all the queries accepted from the tested secondary server.

The management of test scenarios is the most important part of the project, so it is no wonder that this component is the largest. For better understanding, the description is split into two parts - the scenario generator itself and the *ResponseManager*.

Response manager

The `ResponseManager` is an abstract interface, which figures as an intermediary segment between test scenarios and the `Server`. It determines concrete versions of messages according to the query received from the secondary server and the data received from the `TestCase`.

Firstly, it might be a good idea to describe the interface itself and later on explain the interest of the `TestCase`, because the idea of this component is basically pretty simple. It is just a class, which implements two methods: `get_soa_rr(self)` and `get_responses(self, request: Message)`.

The `get_soa_rr` is supposed to return the SOA RRset. Since there can be only one SOA record in a zone, this RRset contains exactly this one record. It is used as argument for the method `NotifyManager.get_notify_message`.

The other method, `get_responses`, has been actually already mentioned in the Section 4.1.4 (Server). This method takes as an argument a message, which was received from the secondary server, and is supposed to return a tuple with a list of messages to be sent together with the flag, whether or not should be the connection immediately terminated. The design is really abstract and therefore all the implementations might do whatever they want - answer regular responses, answer just half of the response, do not answer at all, answer only a non-zero error code, answer just random bytes with no DNS meaning, and so on. Of course, the most used answer will be the first type, where the server answers a regular response filled with valid data received from the `TestCase`, but it is important to keep in mind, that there are actually no limitations and therefore it **is possible** to reproduce **any scenario** with this design and mechanism.

The most common (and most generalized) usage is, of course, answering regular responses. In those responses, the most variable part is the answer section, which contains just a list of records (which are generated by a `TestCase`). For this purpose, there are already 3 implementations.

First and the most abstract implementation is named `DummyResponseManager`. In the constructor it takes the SOA RRset (which is used for answering on SOA queries), the AXFR content (list of resource records sets used for answering on AXFR queries) and IXFR content (again list of resource records sets used for answering on IXFR queries). In the `get_responses` method it just maps the question type to the corresponding content.

Second implementation extends the `DummyResponseManager`. It overrides the IXFR content with the AXFR content and therefore it is named `AxfrResponseManager`. Because no IXFR content argument is needed, the constructor is left only with the two arguments for SOA and AXFR. This response manager is not used only on special occasions to test a server if it is able to process the AXFR response on IXFR query, but it is also used for the very first query, where the tested secondary server just downloads the initial data, the initial zone content.

The third implementation, named `BasicResponseManager`, also extends the first one, even with two things. Firstly, the constructor takes much more arguments. Old SOA record, new SOA record, all records of the zone, removed records during the zone change, and added records during the zone change. The content for AXFR and IXFR responses is then constructed simply from these arguments, which saves lots of effort and repetitive work during the scenario generation. The other thing, which is improved, is the zone serial number check during the IXFR. The query contains a SOA record in its Authority section. This record is then compared with the old SOA record of the zone and if there is a mistake, AXFR response is replied instead.

During message construction, it is important to keep in mind the maximum limit of the message size. The `dns.message.Message` class works with data and at the end, during the conversion into the bytes form, it executes the final compression. It can easily happen that the message is too large and therefore cannot be sent. The `DummyResponseManager` solves it in the way that it tries to convert the message into the bytes form and if it fails, it randomly splits

the content of the message (in the meaning of the Answer section, which is the largest one) and makes multiple shorter messages from the original large one. This mechanism continues recursively until all messages are short enough to be sent.

Because the software is not supposed to be fully deterministic (by randomization, larger scale of cases is covered), this splitting of a message is randomized. Also, the response manager uses the randomization trick described before - it takes a `Random` instance in its constructor as an optional argument to be able to produce the very same splitting sequence all over again, if needed.

So the response manager is supposed to generate concrete messages, but not changes in a zone nor any other data. For these purposes there is the `TestCase` component.

Another implementation of the response manager, which would boost the randomization and the scale of possible scenarios, is to return an AXFR response on IXFR query once in a while. In case there is an IXFR query, the response manager would (with some probability) respond with the AXFR message and the tested secondary server would have to adapt to it.

TestCase

`TestCase` implementations generate various test scenarios and also define the expected state of a zone on the side of a tested secondary server. They have 3 abstract methods, which are essential for the testing, and one other method, which has a huge impact on the test and its default implementation can be overridden.

The first abstract method, which is used at the beginning of testing and must be implemented, is named `get_init_data(self)`. This method returns a `ResponseManager`, which contains the initial records of a zone. The test scenario may want to have the zone completely clean, but every zone must contain at least a SOA record and therefore the zone is not literally clean and its initial form has to be propagated to the secondary server as well.

Another abstract method is the `__next__(self)` method. All the changes are generated here and this method also returns a `ResponseManager`. Instances are supposed to remember their inner state and adjust those changes to it. Once all changes have been generated, this method should raise a `StopIteration` exception, which is a signal to stop the testing.

Last abstract method is `get_zone(self)` and, how the name indicates, returns the actual zone in the form of a `dns.zone.Zone` instance. The content of the zone must be adapted to the inner state of the `TestCase` instance. If the testing software decides to check the zone of the secondary server, this zone is used as the expected state. If these zones differ, an exception is raised and the test is designated as failed.

There is one more method, which can be overridden and it is `require_zone(self)`. The return value is either True or False and it means whether or not should the testing software run a zone check - check the zone of the secondary server. Thanks to this feature it is possible to enforce a zone check any time in the middle of testing, for example after a dangerous zone update.

The design of the test cases is very simple and together with the response manager compo-

nent, they make up a very strong testing utility. Moreover, the design of the test cases is even abstract enough for creating so-called *wrappers*. A wrapper is a `TestCase` subclass, which takes another class as an argument and just slightly changes its behaviour.

An example is the `XfrBlackoutTestCase`. This wrapper takes another test case and in every iteration, it firstly returns a modified response manager and after that the normal, original, response manager. The modified one is just another wrapper over the normal response manager, which cuts all the messages in half and sends just the first half. After that, it forces the server to immediately close the connection and pretend a blackout.

Utilities for generating random values

During generating the test data, there are a few functions, which might be useful. Mostly they help during randomization when it is needed to come up with some values.

A simple, but often used function is a function for a random TTL. The maximum value for a TTL is $2^{31} - 1$ seconds, which equals to more than 68 years.

Another utility is the `get_zone_from_records` function. It takes a list of resource records sets as an argument and returns a `dns.zone.Zone` instance. This may come in handy since there is a need to put all resource records to the response manager for the AXFR responses and also to get the `Zone` instance in the `get_zone` method.

A next useful function is for generating random domain names. This may seem to be really simple, but it is actually not really true. Domain names have few rules and limitations, which have to be observed. The maximal length of a domain name is 253 characters. Every domain name consists of labels, which are separated by dots and whose maximal length is 63 characters.

Because of the labels limitation, it is a good idea to generate a new label, add this label to the beginning of a domain name, and repeat this procedure while the whole domain name is not long enough.

The problem is that at the end it might miss just one character until the name has the desired length. But by adding one more label, one dot is also added and it makes two more characters, which overfills the domain name. A naive solution is not to add a whole new label (when just one character is missing), but just add one single character to the lastly added one. And yet, it is not a sufficient solution. It might happen, that the last label is exactly 63 characters long, and adding the one character breaks the rule about labels.

Although the task sounds pretty simple, the solution is not. There are lots of rigours which must be covered. An example of an implementation is in the Figure 4.2.

Another completely different solution might be generating random strings and then checking whether or not is the string a valid domain name with a regular expression, as shown in the Figure 4.3. Surprisingly, this solution is only 3.5 times slower than the previous one (measured 10 times with 1000 function calls) while being much shorter and the code is more readable. But it brings lots of complications. For example, there is still a (very) small chance that the function might run for an extremely long time or the necessity of coming up with a regular expression, which might not be simple for more complex domain names.

Note that these functions do not generate all possible domain names, but only a subset, which is the most common. Although there is RFC 3696 [8], which introduces some sort of "preferred form" of a domain name, the RFC 2181 [9] permits any combination of bits. This allows even non-ASCII labels, as described in the RFC 3490 [10]. The .cz domain is ready for the implementation of Internationalized Domain Names (IDN), but it has not been implemented yet due to low interest from the community (sourced from `https://háčkyčárky.cz/`).

```
while curr_len < max_length:
    label_len = min(
        [max_length - curr_len - 1, rand.randint(1, 63)])

    # It's a trap, not enough space for the last label!
    if max_length - curr_len - label_len - 1 == 1:
        # We cannot enlarge the label, just generate another label length
        if label_len == 63:
            continue
        # Or we can enlarge the label length by the missing one byte
        else:
            label_len += 1

    labels.append(rand_label(label_len))
    curr_len += label_len + 1
```

Figure 4.2: Example implementation of generating a random domain name.

```
characters = string.ascii_lowercase + string.digits + '-.'
while True:
    s = ''.join(random.choices(characters, k=252))
    if re.match(r'^([\w\d-]{1,63}\.)+[\w\d-]{1,63}$', s):
        return s
```

Figure 4.3: Example implementation of generating a random domain name using random strings.

4.1.6 Quick overview

It is not easy to keep track of all the components described above and think about how do they cooperate together. The diagram in the Figure 4.4 should help to illustrate how are the components and their implementations segmented and which components are used together.

Red lines represent inheritance between particular classes (the outer classes inherit from the inner ones). For example in response managers, `OneByOneAxfrResponseManager` inherits from `AxfrResponseManager`, which inherits from `DummyResponseManager`, which finally inherits from the original abstract `ResponseManager` class.

Rectangles represent components. Most of them have been already described above, so now it should be clear that each component is a class with abstract methods, which are then implemented in particular sub-classes. (The only exception is the `NetworkAnalyst`, which is not abstract, because it would not have any benefits, but it does not matter. It is just an implementation detail.)

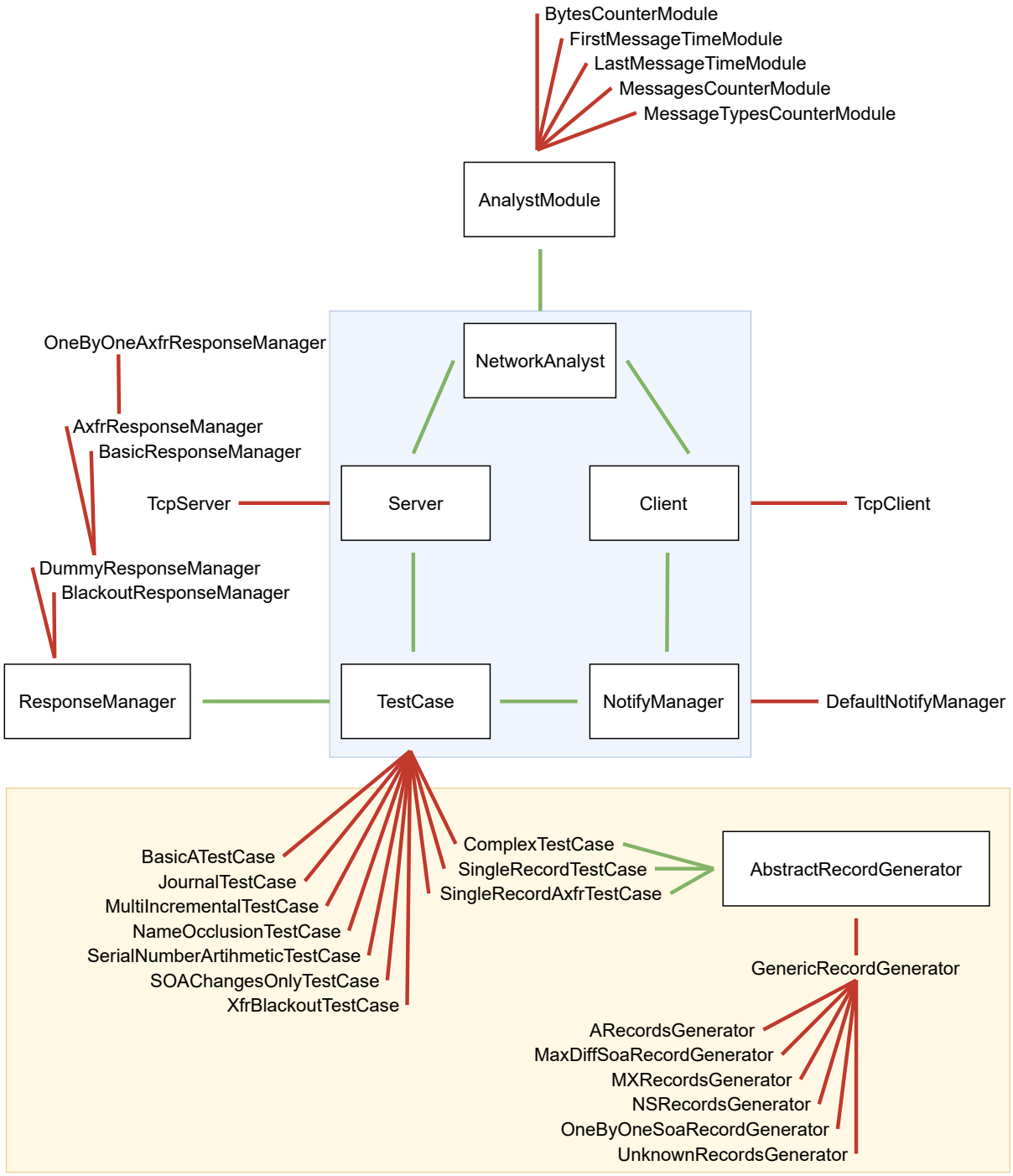

Figure 4.4: Hierarchy of used classes and components.

Green lines represent cooperation between individual classes, as described below (goes from top to bottom):

- `NetworkAnalyst` contains a set of `AnalystModule` that gather some information and create statistical data.
- `Client` uses `NetworkAnalyst` to collect information about the network load.
- `Server` uses `NetworkAnalyst` to collect information about the network load as well.
- `Client` uses `NotifyManager` for creating all DNS NOTIFY messages.
- `Server` uses `TestCase` for answering queries from the tested secondary server.
- `TestCase` uses `ResponseManager` for providing bytes as answers to queries.
- `NotifyManager` uses `TestCase` for providing the `SOA` record used in DNS NOTIFY messages.
- *And finally, some test cases use* `AbstractRecordGenerator` *for generating some records, but this part has not been described nor mentioned yet.*

The blue highlighted part of the diagram (blue background) has been just described in this section (Section 4.1). So the blue part together with stuff associated with those components should sound familiar at the moment. In the next section (Section 4.2), it will be described how these components interact with each other in more details.

The yellow highlighted part of the diagram (yellow background) refers to test scenario implementations and have not been described yet, it will be described in the Section 5. If the hierarchy of described classes will not be clear, this diagram should make it better.

4.2 Project core

All modules with their abstract interface have been introduced. The last thing is to put them together to create a solid testing utility. And that is what the last component, `Core`, is for. It tries to implement (and moreover, extend with other features) the simple and ideal workflow showed already in the Figure 3.1.

The task for the core is to receive a configuration via its arguments and make the whole testing happen. Use all the designed modules and components in the way they are supposed to be used and also compare the results of the testing.

Core implementation

The `Core` class takes lots of arguments in the constructor. Some of them are just passed into constructors of other components, some of them are used for the configuration of the testing, which makes the scope of possible test scenarios again even larger. The arguments are:

- **domain** containing the tested zone domain name.
- **client** with the `Client` instance.
- **test_case** with the `TestCase` instance. This test scenario is being used for the testing.
- **notify_manager** is the instance of `NotifyManager`.
- **server_cls** containing the `Server` class. The whole class, not an instance! The class is then started in its own process.
- **test_server_ip** is a tuple with an IP address and a port, where is the server supposed to run. Of course, it has to be identical to the primary server, which is set in the configuration of the tested secondary server.
- **server_network_analyst** with an instance of the `NetworkAnalyst`, which is then used in the server.
- **iter_wait** is a value determining whether or not should the first process (process, which is generating zone changes) wait for the other process (the server process, until the last generated zone transfer happens). It can be either a True/False value or a function returning a True/False value. Using the function approach, it is possible to randomize it or affect it in a deterministic way. Together with specific test cases, it may create interesting scenarios. Note that this function is not supposed to accept any arguments.
- **iter_sleep** describes whether or not should the process give some time to the secondary server to process the transfer before generating new changes. Basically, it is a sleep at the end of every iteration. The value can be either `None` (do not give any extra time, do not sleep the process), a float (static sleep time at the end of every iteration), or a function returning a float. Again, using this approach it is possible to make the scenario scale larger. Note that even this function is not supposed to accept any arguments.
- **compare_zone_attempts**. When there is a zone check and a zone comparison fails, the reason might be that the secondary server has not managed to apply the changes yet and just needs some more time. Therefore, it is possible to give it some time and repeat the zone check a few more times. This number gives how many tries should be performed.
- The last argument is **compare_zone_sleep_time**, which sets the sleep time between the zone checks attempts.

These arguments represent a configuration for running the test. As described below, the class consists of three main parts and functions, which has to be called in this very order: `start`, `run` and `stop`. As well as in `Server` or `Client` components, neither these methods do not accept any arguments.

The `start` method sets up the shared resources (in the meaning of shared between the two processes) and then starts the server in its own process. The reason is that the server

must be available the whole time, so the tested secondary server can connect to it and ask for data. Meanwhile, it is needed to generate some changes in the test case and this might be computationally intensive, so some queries might run out of time before they would be answered if everything was done just in one single process and one single thread. Unfortunately, Python has a feature called Global Interpreter Lock (GIL), which allows only one thread to hold control over the interpreter. In other words, in Python, two different threads are not running at the same time. Therefore threads would have not solved this issue. Neither would have using the *asyncio* library (Asynchronous Input and Output). The only possibility is to split it into two processes and communicate via some shared objects.

The second method to be described is the `run` method, but since it contains more stuff to explain, a special section will be dedicated to this topic.

Last method is the `stop` method. It just stops other dependent components so the program can terminate successfully. Firstly it sends the SIGTERM to the Server. As described in the Section 4.1.4 about servers, it handles the signal, closes all sockets, and peacefully shuts down the server. Then the main process joins the server process, so no collisions happen (the collision might be for example mixing the printings of analysis between the `Client` and the `Server`). After the server process is terminated, the shared structures are destroyed and after all the *Client.stop* method is called. By this, the whole testing ends and the program terminates with zero (means successful) code.

The run method

This method takes care of the whole testing process, mechanism. After the server starts, in the shared `ResponseManager` there is the return value of the `TestCase.get_init_data` method, which contains the basic and initial data about the zone.

When the tested secondary server downloads these data, a signal is sent to the `run` method and all the test iterations might begin. At this point, it is known that the secondary server is properly configured and is running, ready for testing.

The first set of changes is generated. The new response manager is set to the shared structure so this change can appear in the server as well. The SOA RRset is passed to the notify manager and a DNS NOTIFY message is sent from the client to the secondary server. At this point, it is expected that the secondary server is going to ask for changes and process the zone update.

According to the `iter_wait` configuration, the process either waits for the zone transfer or immediately continues. Next, according to the `iter_sleep` configuration, it either sleeps for a while or immediately continues again.

At the end of every iteration, now, the test case can determine whether or not should be executed a zone check. As described before, it is conditioned by the return value of the `TestCase.require_zone` method. If the zone check is required, it is also executed.

The zone check starts in the client, which sends an AXFR query to the secondary server. The answers are then passed back to the `Core`, where the zones (received and expected one) are compared. Unfortunately, in the equality implementation (`__eq__` method) of the `dns.zone.Zone`

class, the `TTL` is not taken into consideration. Therefore, own zone comparison, which checks even TTL values, had to be implemented.

If the zones are not equal, another zone check attempts are executed according to the configuration. If none of those attempts are successful, the testing immediately ends.

If the zones are equal, testing continues until there are no more zone updates to propagate to the secondary server. When the test case is out of zone updates, one final zone check is executed. After this last zone comparison, the method ends and calling the `stop` method is expected.

4.3 Running a test

Running a test is just a matter of configuration. Lots of details can be adjusted through the `Core` constructor. Set up a domain, a client, a server, a notify manager and primarily the test case, create a `Core` instance and run the trio `start`, `run` and `stop`.

At this point, calling the 3 main methods (plus constructor) is needed. However, it does not satisfy the requirement, which has been set down in the Section 3, that *It is expected that one function takes some configuration parameters and the whole testing process will be executed automatically according to the parameters.* Therefore the shortcut has been created and it is possible to trim the code to just calling `Core.test(`*`args`*`)`. The arguments are passed to the constructor and the three methods are called in the correct order.

Remember, that the tested secondary server must be also properly configured. Lots of implementations restrict zone transfers due to security reasons. For every zone (corresponds to the `domain` argument) a set of primary servers must be provided (has to contain the server address) as well as a set of servers, from which it can receive the DNS NOTIFY messages (again, has to contain the client address). And last but not least, the secondary server IP address and port must be set accordingly to the address given to the `Client`.

All the configuration is prepared in the `__main__` file. It is only needed to edit the configuration according to the particular needs and then run simply `python xfr_tester`. When the software is running, it is possible to invoke the secondary server to download the initial zone and then the testing process begins automatically.

4.4 Requirements fulfilment

All the structure and components have been deeply described and explained. It is important to have a look back and check whether or not are all the predefined requirements fulfilled, whether or not is the design correct.

- *"The possibility to reproduce every possible scenario which could ever happen."*
  The `ResponseManager` can return any sequence of bytes, which is then sent. Therefore it is possible to send anything, it is possible to reproduce any scenario.

- *"Make test scenarios implementation as easy as possible."*
  There are prepared basic implementations of the `ResponseManager`, which simplify the

conversion of data (records) to a message. Also, the `TestCase` abstract design does not have many methods to be implemented and after all, there are some helpful utilities, which can be used during testing. The software is able to handle the raw byte data, but also it is compatible with the `dnspython` structures bringing a higher-level approach over the DNS messages.

- *"The ability to configure independent modules and create a wider range of possible test scenarios."*
  The abstract design fulfils this requirement in general. All independent parts of the mechanism (both the testing mechanism and the zone transfer mechanism) were detached and the final configuration can mix different combinations of implementations.

- *"The ability of the final software to be extended."*
  One example of possible extensions are for example the wrappers of test cases or the implemented Response managers. But all the modules are, thanks to the OOP approach, fully extendable.

- *"It is mandatory the software must be easy to run."*
  The project itself needs just a Python interpreter and installed package `dnspython`. The testing can be run using only one command, `python xfr_tester`, which then runs the predefined configuration. In the code, the testing is started by calling one single method, which then starts the whole workflow - `Core.test(args)`.

- *"At the end, the software will verify the correct processing of all the zone changes by the secondary server."*
  It is being done and moreover, the zone check can be executed any time in the middle of the testing. It is suggestible in the `TestCase`.

- *"The software will collect basic statistics about the network load, which can be then used to compare different implementations."*
  This is done by the `NetworkAnalyst`. The concrete set of modules is easily configurable and implementation of new modules can be also very simple - even one-liners, as described before.

Chapter 5

Test cases implementations

At this point, all the testing environment is designed and all the components have at least one implementation, which can be used for testing. It is the right time for creating the basic set of scenarios to show, how is this testing tool supposed to be used and which can be then used for testing a real server implementation.

5.1 Default set of implemented test cases

The very first, basic and the simplest test scenario is `BasicARecordsTestCase`. It starts with the serial number 1 and 10 A records in the zone. Then, in every iteration, all records are removed and replaced by another 10 records. The serial number is incremented by 1. The values of the A records have the form `127.0.X.Y`, where `X` represents the iteration number (starting with 1 for initial records) and `Y` is an index of the particular record in each iteration, so values from 1 to 10. Thank to this indexing rule, it is possible to build the whole zone in every situation, the only needed information is an index - the iteration number. All the other data (TTL of records and SOA refresh, retry, expire, and minimum attributes) are static, so they are not changed during the zone changes. An example of one message with an incremental change between versions with serials 1 and 2 is showed in the Figure 5.1.

This scenario is very simple, therefore, unlikely to be inaccurately processed. However, it is also important to test the most obvious scenarios and is advised to have at least one simple scenario. By using this simple test case, it is possible to check that everything is configured correctly and failures of other, more advanced, test cases are not caused because of various setting incongruities.

Next test case, which has been implemented, is the `SingleRecordTestCase`. In this scenario, a set of records is removed and immediately added back to the zone. So every increment is semantically empty but syntactically nonempty. Moreover, if there are not any other records, the zone is completely empty after applying the *remove* part of the increment. This situation might be forbidden in the configuration of some server implementations, for example BIND9 has *empty-zones-enable* option. But after applying the other part of the increment, the *add* part, the zone becomes non-empty (and therefore valid) again. The secondary server must not fail in

```
            +-----------------------------------------+
Header      | OPCODE=SQUERY, RESPONSE                 |
            +-----------------------------------------+
Question    | QNAME=zone., QCLASS=IN, QTYPE=IXFR      |
            +-----------------------------------------+
Answer      | zone.                IN SOA serial=2    |
            | zone.                IN SOA serial=1    |
            | zone.                IN A   127.0.1.1   |
            | zone.                IN A      ...      |
            | zone.                IN A   127.0.1.10  |
            | zone.                IN SOA serial=2    |
            | zone.                IN A   127.0.2.1   |
            | zone.                IN A      ...      |
            | zone.                IN A   127.0.2.10  |
            | zone.                IN SOA serial=2    |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 5.1: Example of BasicARecordsTestCase increment change from version 1 to 2.

the middle of the transfer and must apply the whole increment. Since the zone content looks exactly the same every time (only SOA record changes), it is simple to build the whole zone for the final check.

An example of this test case is in the Figure 5.2. Increment from the serial 4 to the serial 5 consists of deletion and addition of two same records, A records with values `127.0.0.1` and `127.0.0.2`. Note that it is not the same, as the first example. Here the very same records are immediately added back, while in the first case the records have been changed, so the increment was not semantically empty.

Another test case is `MultiIncrementalTestCase`. It tests the capability of processing one zone transfer message consisting of multiple increments, as described in the Section 2.5.2 about IXFR messages. This situation is not really common, because usually, the primary servers try to propagate all changes to secondary servers immediately, before any other increment appears, but yet must be covered accordingly to the specification.

Every zone transfer consists of `N` increments, in every increment one A record and one MX record are removed and one A record and one MX record are also added. Again, the whole zone can be built just by knowing the actual iteration index, so the final zone check can be done easily.

An example is showed in the Figure 5.3. The message contains serial changes $1 \rightarrow 2 \rightarrow 3$ in one message (so the `N` parameter equals to 2), every time A and MX records from the previous version are removed and new records are added. The A records are different in the last part of the IP address, the MX records differ in the sub-domain of the exchange attribute.

Next implemented test case is `SOAChangesOnlyTestCase`. As the name indicates, the changes in this scenario are only in the SOA records. All other records remain untouched. The IXFR

```
            +-----------------------------------------+
Header      | OPCODE=SQUERY, RESPONSE                 |
            +-----------------------------------------+
Question    | QNAME=zone., QCLASS=IN, QTYPE=IXFR      |
            +-----------------------------------------+
Answer      | zone.                IN SOA serial=5    |
            | zone.                IN SOA serial=4    |
            | zone.                IN A   127.0.0.1   |
            | zone.                IN A   127.0.0.2   |
            | zone.                IN SOA serial=5    |
            | zone.                IN A   127.0.0.1   |
            | zone.                IN A   127.0.0.2   |
            | zone.                IN SOA serial=5    |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 5.2: Example of SingleRecordTestCase increment change from version 4 to 5.

```
            +-----------------------------------------+
Header      | OPCODE=SQUERY, RESPONSE                 |
            +-----------------------------------------+
Question    | QNAME=zone., QCLASS=IN, QTYPE=IXFR      |
            +-----------------------------------------+
Answer      | zone.          IN SOA serial=3          |
            | zone.          IN SOA serial=1          |
            | zone.          IN A   127.0.0.1         |
            | zone.          IN MX  1000 m1.zone.     |
            | zone.          IN SOA serial=2          |
            | zone.          IN A   127.0.0.2         |
            | zone.          IN MX  1000 m2.zone.     |
            | zone.          IN SOA serial=2          |
            | zone.          IN A   127.0.0.2         |
            | zone.          IN MX  1000 m2.zone.     |
            | zone.          IN SOA serial=3          |
            | zone.          IN A   127.0.0.3         |
            | zone.          IN MX  1000 m3.zone.     |
            | zone.          IN SOA serial=3          |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 5.3: Example of MultiIncrementalTestCase incremental changes.

message contains only 4 SOA records in the Answer section, which might seem a little bit strange, but it is a completely valid scenario.

The initial zone contains 3 A records and 2 MX records by default. Then, only SOA record is changed, nothing is removed or added, so the final zone consists only from the default records and the actual SOA record, which is given by the iteration number (how many iterations have been executed).

Example of this increment is in the Figure 5.4, where the only change is in the SOA serial number $5 \rightarrow 6$.

```
            +-----------------------------------------+
Header      | OPCODE=SQUERY, RESPONSE                 |
            +-----------------------------------------+
Question    | QNAME=zone., QCLASS=IN, QTYPE=IXFR      |
            +-----------------------------------------+
Answer      | zone.                IN SOA serial=6    |
            | zone.                IN SOA serial=5    |
            | zone.                IN SOA serial=6    |
            | zone.                IN SOA serial=6    |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 5.4: Example of SOAChangesOnlyTestCase incremental changes.

Following test case, `SerialNumberArithmeticTestCase`, is not so simple and it is needed to explain *serial number arithmetic* first. This topic is described in RFC 1982 [11] in details. Basically, the SOA SERIAL number is an unsigned 32-bit integer, which as a data type gives enough space, but not infinite space, so after some finite time, the defined space for serial numbers might overflow.

The solution is to merge the beginning of the scale (`0x0` in HEX) and the end of the scale (`0xffffffff`), so `0xffffffff + 1 = 0x0`. In the mathematical terms, it can be expressed as `c = (a + b) mod` 2^{32}. In this way, the scale becomes infinite (or to be more precise - the increments can be done infinitely), but the comparison mechanism must also be adjusted. Reason is obvious - `0x0` is less than `0xffffffff`, even though it actually comes right after.

Comparison of two numbers is more complicated. Two numbers are equal if and only if they are the same (if and only if they have the same bites representation). Then, according to the specification [11], `a` is greater than `b` if and only if `a` is not equal to `b` and:

$$\begin{aligned} &(\mathrm{a} < \mathrm{b} \text{ and } \mathrm{b} - \mathrm{a} > 2^{31}) \quad \text{or} \\ &(\mathrm{a} > \mathrm{b} \text{ and } \mathrm{a} - \mathrm{b} < 2^{31}) \end{aligned} \tag{5.1}$$

That is the exact definition, but it is not really clear how it works. The comparison can be understood in the way, that `a` is greater than `b`, if and only if the count of increments from `b` to `a` is lower than the count of increments from `a` to `b`. It can be visualized via a circle, as shown in

the Figure 5.5. Numbers are being incremented clockwise, and the length from `b` to `a` is smaller, than the length from `a` to `b`. Therefore, serial number `a` is greater than serial number `b`.

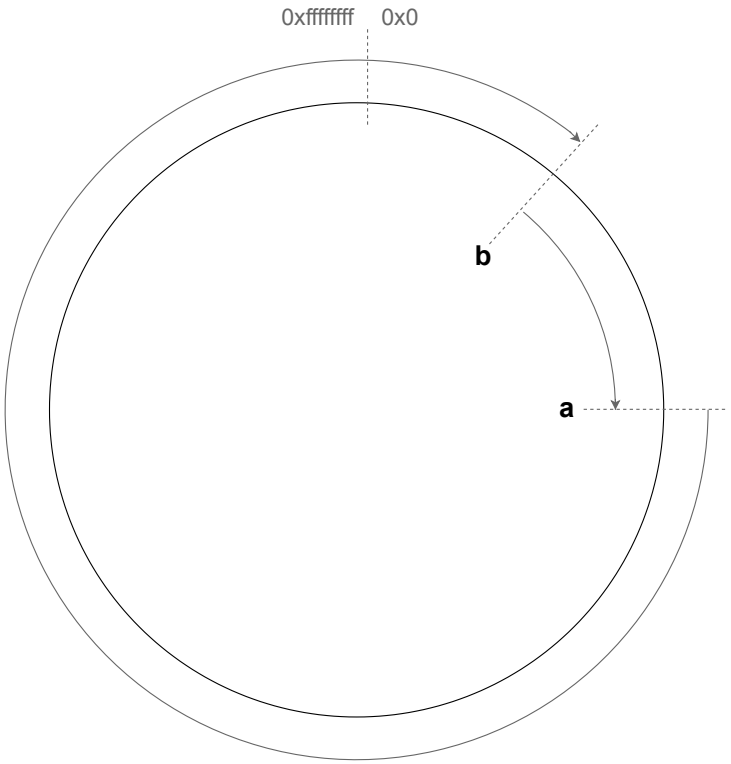

Figure 5.5: Serial number arithmetic visualized with a circle.

A problem appears when the count of increments from `a` to `b` is the same as the count of increments from `b` to `a`. In this situation, an attempt to use this ordering operator produces an undefined result.

In the real-life operation it means that serial numbers with their maximum difference of $(2^{32}/2) - 1 = 2^{31} - 1 = 2\ 147\ 483\ 647$ can be compared while the result still corresponds to reality. If a secondary server is off for a long time, a zone could make up to 2^{31} increments and the mechanism would still work perfectly. For illustration, 2^{31} seconds is over 68 years. But the serial number is not required to increment by 1. The increment between zone versions can be higher. One approach for numbering the zone versions is described in the RFC 1912 [12]. The recommended syntax is `YYYYMMDDnn`, where `YYYYMMDD` is the date of the zone change and `nn` is the revision number of the day (this revision number increments by one and resets to `01` every day). This format is enough for updating the zone every 15 minutes (what is enough for practical usage) and will not overflow until the year 4294.

`SerialNumberArithmeticTestCase` is testing exactly this ability to compare serial numbers and refuse zone transfers with serial number, which is not higher than the actual one. Some naive implementations of DNS servers may ignore this restriction because it is not very ordinary situation. During the testing, one regular zone transfer is generated (for example $1 \rightarrow 2\ 147\ 483\ 648$, difference $2^{31} - 1$) and after that one false zone transfer is generated ($2\ 147\ 483\ 648 \rightarrow 0$, difference 2^{31}). The tested secondary server is supposed to refuse the second transfer.

Another implemented test scenario is `NameOcclusionTestCase`. Name occlusion is a situation, which appears in a combination of Name Server (NS) records with other types of records. It is described in the RFC 2136 [13]. NS records delegate a part of a zone (it could be said *subdomain*) to another server, but there might still be some records in the part of the zone, which has been delegated somewhere else. Those records are fully-fledged, they still belong to the zone, but they cannot be reached during a resolving. For illustration, think of the following example.

A `com.` server has the following records:

example.com.	600	IN	NS	ns.example.com.
www.example.com.	600	IN	A	127.0.0.1

A resolver is looking for the A record of `www.example.com`. It asks a root server. The root server answers with an NS record of `com.` domain. The resolver asks this server with the same question. Now, the `com.` server knows a desired A record of `www.example.com.`, but it answers with the `NS` record, which points to `ns.example.com.`, so the resolver asks this name server and uses its A record (which can be possibly different). The A record from the `com.` server with the value `127.0.0.1` has been somehow ignored, but only during the resolving! This record still belongs to the zone and must be taken into account while zone transfers.

One of the main purposes of DNS servers is to provide fast and correct results. Therefore some server implementations might use some kind of efficient structures or algorithms to provide these answers more quickly. But it is still important for zone transfers to keep all the records and work with them (the NS record might get deleted and then the "ignored" A record stops being ignored). And that is exactly what is this test case for.

In the beginning, the zone consists of some NS records, then some static A records (static in the meaning that they are not changed during zone increments), and some dynamic A records (dynamic in the meaning that they are changed during zone increments). One zone change removes the dynamic A records together with the NS records and replaces them with new A and NS records. One change is showed in the Figure 5.6.

The last simple test case is named `JournalTestCase`. Changes of a zone are saved into so-called *journals*. It is useful to have a facility to take a journal and re-run the scenario. And that is exactly what this implementation does. It takes a journal, parses it, and tries to apply all the changes in a specified order. An example of a journal (created by KnotDNS) is illustrated in the Figure 5.7. The first section contains the initial zone version with serial number 1, 1 NS record, and 3 A records. Then it contains one incremental change, which is divided into two parts - *removed* and *added*. It is actually a journal of the `NameOcclusionTestCase` described before.

The most important part of this journal test is reading and parsing a journal file. Every row contains either a comment (starts with two semicolons) or one record. Every record consists of 4 parts separated by a tabulator - domain name, TTL, record type, and record value. A class of the records is not included in the journal file, so the `IN` class (= 1) is used by default. Every line containing a resource record is properly parsed and converted to a `dns.rrset.RRset` instance.

```
            +-----------------------------------------+
Header      | OPCODE=SQUERY, RESPONSE                 |
            +-----------------------------------------+
Question    | QNAME=zone., QCLASS=IN, QTYPE=IXFR      |
            +-----------------------------------------+
Answer      | zone.              IN SOA serial=4      |
            | zone.              IN SOA serial=3      |
            | zone.              IN NS  ns.g3.i1.zone.|
            | dyn.i3.zone.       IN A   127.0.3.1     |
            | zone.              IN SOA serial=4      |
            | zone.              IN NS  ns.g4.i1.zone.|
            | dyn.i4.zone.       IN A   127.0.4.1     |
            | zone.              IN SOA serial=4      |
            +-----------------------------------------+
Authority   | <empty>                                 |
            +-----------------------------------------+
Additional  | <empty>                                 |
            +-----------------------------------------+
```

Figure 5.6: Example of NameOcclusionTestCase incremental changes.

```
;; Zone-in-journal, serial: 1, changeset: 1
;; Added
zone.          600   SOA   zone. admin.zone. 1 600 600 600 600
zone.          600   NS    ns.g1.i1.
dyn.i1.zone.   600   A     127.0.1.1
static.zone.   600   A     127.0.0.1
static.zone.   600   A     127.0.0.2
;; Changes between zone versions: 1 -> 2, changeset: 2
;; Removed
zone.          600   SOA   zone. admin.zone. 1 600 600 600 600
zone.          600   NS    ns.g1.i1.
dyn.i1.zone.   600   A     127.0.1.1
;; Added
zone.          600   SOA   zone. admin.zone. 2 600 600 600 600
zone.          600   NS    ns.g2.i1.
dyn.i2.zone.   600   A     127.0.2.1
```

Figure 5.7: Example of journal for JournalTestCase.

The journal is composed of some sections - the initial zone version and particular zone increments. Each zone increment section consists of two more sub-sections - removed and added records. When reading a journal, it is possible to orient by the comment rows to distinguish between different sections. But this approach is not very stable since the comment formats might differ between products or even versions of one product. A more general approach is to observe the SOA records because every single section begins with a SOA record and the sections repeat regularly.

The last important piece of information to mention is the size of a journal. A journal can be a few gigabytes big, so it is a bad and dangerous idea to parse the whole journal and then start the testing. A better approach is to read the journal progressively, section by section, and every time when a new section is needed.

The implementation of `JournalTestCase` takes a reader generator as an argument. The generator is supposed to read a journal and yield every record. The test case then calls the generator every time, when new records are needed and groups the records into desired sections. This design allows to have only one shared test case implementation and provide multiple different journal readers for multiple different journal formats.

Also remind the `XfrBlackoutTestCase` from the Section 4.1.5. Using this wrapper, it is possible to extend every described test case and create other unique scenarios.

5.2 Complex test scenario

All these simple test cases have one thing in common - they just fill a message with some data. It would be possible to randomly generate some records and insert them into some messages. Possibly, after some time, the randomization might cause that all the previous scenarios would be tested - it might generate a message without changes, just an increment of the SOA serial number (alias `SOAChangesOnlyTestCase`). Or it could generate a message with multiple increments (alias `MultipleIncrementTestCase`). It could also generate the same combinations of NS records with other types of records (alias `NameOcclusionTestCase`). Or it could remove a record and immediately add it back into the zone (alias `SingleRecordTestCase`).

This test case would just use some random records, insert them into some messages and all these cases might be covered, if the test runs for a long enough amount of time. Simple as that. The question is how to solve the randomization and which records should be used.

It should depend on the configuration. If completely everything would be random (random domain names, random record types, random record values, and so on), the space of all possible combinations would be so huge that none of the previously described situations could be possibly generated (actually could, but the probability would be approaching zero). But also, it might be intended to have this randomized test case, so it should be also possible to randomize completely everything.

Test case like this was also designed and implemented. It is named `ComplexTestCase` and it is based on *records generators*. A record generator is an abstract interface, whose implementations take care of generating the records, which are then inserted into messages, so it is possible to

control the randomization aspect, as it was required.

5.2.1 Records generators

A record generator takes care of records, which are then used during the testing. It is a very simple component, which has an abstract interface with two methods - `next_add()` and `next_remove()`.

The first method, `next_add()`, is supposed to generate a record, which is then added into a zone. It returns a `dns.rrset.RRset` instance, which contains the generated resource record, and this `RRset` is also inserted into the XFR message. Which particular resource record is generated, it depends on the purpose of every implementation. One can focus on generating some records with the same type, another one can focus on generating records of a domain or some implementations can generate just completely random records.

The second method, `next_remove()`, takes care of removing the generated items from the zone. It returns either a record (again in the form of a `RRset` instance) or `None`. The returned record must have been generated by the `next_add()` method first!

Most common implementations would just generate some records in the `next_add()` method, save them into a list, and then it would be removing these records from this list one by one in the `next_remove()` method. Also, some of the implementations would just randomize the domain name. This general implementation has been already implemented in the `GenericRecordGenerator` class.

`GenericRecordGenerator` takes domain and `randomize_names` (`True`/`False`) flag as arguments in the constructor. Furthermore, it has two class attributes - `RD_CLASS` (`IN` (= 1) by default) and `RD_TYPE`. Then, it has a method `get_record()`, which has to be implemented in its sub-classes and return a `dns.rdata.Rdata` instance (which is basically just a single record). Using these, it is possible to take care of the `next_add()` and `next_remove()` methods, as it is going to be described now.

In the adding part, it just creates an `RRset` according to the `RD_CLASS`, `RD_TYPE`, domain and `randomize_names` configuration. Then, it inserts one resource record (from the `get_record()` method) into this `RRset` and saves it to a list, which is then used for removing records. Every instance has also a `self.i` attribute, which means *index*, and this index attribute is incremented every time in the `next_add()` method. This index is supposed to be used in the `get_record()` method, so it can generate different values every time.

As mentioned, the `next_remove()` method just pops first item from the list and returns it. If the list is empty, then `None` is returned, so no record is removed (because there is actually no record to be removed).

This `GenericRecordGenerator` makes the implementation of basic record generators very simple. For example a generator of `A` records just needs to specify the `RD_TYPE` attribute and the value of the record in the `get_record()` method. It takes just a few lines of code, as it is showed in the Figure 5.8.

Other implemented generators are for SOA, MX and NS records. Also, there is one im-

```
class ARecordsGenerator(GenericRecordGenerator):
    """Generator of A records."""

    RD_TYPE = RdataType.A

    def get_record(self) -> Rdata:
        """Get next A record."""
        return A(RdataClass.IN, RdataType.A, f'127.0.0.{self.i}')
```

Figure 5.8: Implementation of the ARecordsGenerator using GenericRecordGenerator.

plementation, which generates random records with unknown types. It means it uses those resource record type values, which are not reserved for any purposes. The value structures of these records are not specified (it cannot be converted into a text format, compressed nor handled in a type-specific way), so the server must treat it as unstructured binary data. Those records must be stored and transmitted without any change [14].

5.2.2 Complex test case

The `ComplexTestCase` basically takes some record generators and uses them to add/remove resource records to/from a zone. The constructor takes a domain, the desired number of iterations, instance of a record generator with SOA records, and then a list of generators with normal (in the meaning of not SOA) records. Also, it has some optional arguments, which control the count of changes in every iteration, but they are not important for understanding this test case, therefore will not be described.

The initial zone is empty. Every zone is represented as a list of all records. Then, in every iteration, a number of increments is generated and for every increment, there are generated some changes in the zone, which are also saved as the content for IXFR messages. These changes are created in the following way:

- Generate a random number of attempts to remove a record.
- In every attempt, select randomly a record generator and via the `next_remove` method get a resource record to remove.
- If the return value of the `next_remove` method is not `None` (in most cases it means *nothing to remove*, but of course it can be also a purpose), remove this record from the list of all records and also add this record into the content of the IXFR message.
- When all attempts are executed, generate a new SOA record via the SOA records generator (taken in the constructor of the test case) and also add it into the IXFR message content.
- Generate a random number of attempts to add a record.
- For every attempt, select randomly a record generator and via the `next_add` method get a resource record to add.

- Add this resource record into the list of all records and also add it to the IXFR message content.

After this loop, the list of all records is updated accordingly to the generated changes and also there is the IXFR content, which contains the whole zone update. Everything can be passed to the `ResponseManager` (described in the Section 4.1.5) and propagated to a tested secondary server.

Every `TestCase` must also implement the `get_zone` method. The implementation is very simple - the complex test case has the list of all records and furthermore, there is the utility `get_zone_from_records` (described also in the Section 4.1.5), which can convert this list into the `dns.zone.Zone` instance.

Shortly - the `ComplexTestCase` gets a list of record generators and then makes some random changes in the zone. With the right record generators, it is possible to simulate almost all scenarios from the default set and moreover, to create other scenarios very easily.

5.3 Testing of KnotDNS

Software chosen for testing is the *KnotDNS* (`https://knot-dns.cz`). It is a high-performance authoritative DNS server, which is being developed by a Czech association CZ.NIC and which is deployed (for example) on 3 root servers.

Note that testing of any other DNS server implementation (*BIND9*, *PowerDNS*, and others) would be very similar. The main point is that the testing software simulates an authoritative server and acts as an authoritative server. Setup the tested DNS server implementation so that it considers the testing software as its primary server of a specified zone and the testing software takes care of the rest.

Testing any software is actually mostly about correct configuration. The tested secondary server and the testing tool (or software) must share their IP addresses, ports, and also the zone, which will be used for testing purposes.

For the illustration, the following configuration was used:

- IP address of the tested secondary (KnotDNS) server is `127.0.0.1`, port `5301`.
- IP address of the primary server is `127.0.0.1`, port `5302`.
- Used zone is `dh.`

The configuration of the tested KnotDNS server is shown in the Figure 5.9. The first part, `server`, defines global settings, in this case, the IP address, where the server will be running. The `remote` section defines all the primary servers. Here it is the address, where will be running the testing software. The third section, `acl`, contains the setting for accepting Notify messages and zone transfers. It also has to be explicitly configured. And finally, in the `zone` part, the `dh.` zone is created and the primary server is assigned to it together with the `acl`.

In the configuration of the testing tool, the `Core` (see the Section 4.2) takes 3 arguments, which must correspond to the previous settings:

```
server:
    listen: 127.0.0.1@5301

remote:
  - id: master
    address: 127.0.0.1@5302

acl:
  - id: notify_from_master
    address: 127.0.0.1
    action: notify
  - id: transfer_from_client
    address: 127.0.0.1
    action: transfer

zone:
  - domain: dh
    storage: /tmp/knot/zones/
    file: dh.zone
    master: master
    acl: [notify_from_master, transfer_from_client]
```

Figure 5.9: Example configuration of the KnotDNS server.

1. The zone must be set to `dh.`, as it is in the configuration of the tested KnotDNS server.
2. The `Client` must receive the same IP address, which has been set in the `server` section of the tested KnotDNS server - `127.0.0.1:5301`
3. The `test_server_ip` argument must be set to the `127.0.0.1:5302`, according to the `remote` section in the KnotDNS configuration file.

The default port of DNS servers is **53** for both UDP and TCP [4], but it is not a requirement. As any other software, even DNS servers can listen to any ports they need (according to security options of the machine, where it runs). Therefore, ports and addresses used in the configuration are not set in stone, they can be changed according to any needs. It would be also possible to run the primary server inside one virtual machine and the secondary server in another virtual machine and use the assigned IP addresses of those virtual machines together with the ports 53.

When everything is set properly, the testing may begin. The testing tool is run via `python xfr_tester`. It takes the scenario configured in the `__main__.py` file. The KnotDNS server is run with the command `knotd -c knot.conf` (where the `knot.conf` is the configuration file, default is `/etc/knot/knot.conf`). It is also recommended to use the `--verbose` (`-v` shortly) flag, which enables debug output. It is very useful during debugging.

When the KnotDNS server is started, it loads the configuration and finds out that there should be a `dh.` zone. But it does not have any data in the memory, so it automatically starts the zone transfer via AXFR (because it has no zone to increment from). This query starts the testing workflow and everything continues automatically.

Every time a zone transfer is executed, Knot sets a timer, which disables initial zone transfers

for a period of time. It means that when the server is started repeatedly, the server stops executing the initial zone transfer. It is needed to force the server to perform this initial zone transfer. It can be done using the Knot DNS control utility - *knotc*. This utility has the option to refresh all zones via `knotc -c knot.conf zone-refresh`. This *zone-refresh* starts the initial zone transfer, which then starts the testing workflow again.

At the end of the testing, the `xfr_tester` outputs the results and terminates itself. The KnotDNS server still runs. It is possible to run some custom queries to check the functionality or the server can be also terminated. The server must be restarted before running another test and the zone files must be removed. Otherwise, the server would load the old zone file from the previous test and refuse some zone transfers, because it would evaluate the zone transfers as outdated (its SOA record may have a greater serial number).

Note that when running the server with a custom configuration, the server must have permissions to the configuration file. Also, when specifying zone files and journal databases, the server needs the permissions. It is recommended to run the server as the `knot` user. It can be done using the `sudo -u knot` command.

KnotDNS has been tested with all described test scenarios, every test case was also extended with the `XfrBlackoutTestCase` wrapper. All tests were successful, which affirms the reliability of this software.

5.3.1 Further testing of KnotDNS

According to the testing results, which have been already presented, it seems like the KnotDNS is faultless and perfect. But there is also a possibility, that the testing software (`xfr_tester`) is not working properly. It is a great approach to intentionally configure the server incorrectly to see if the tester evaluates the behaviour as incorrect.

Another approach, how to prove (or rather show) that the `xfr_tester` is working properly, is to find a real bug and cover it with a test scenario. In this section it is going to be explained, how is the software supposed to be used and it will turn out, how powerful the design really is!

Open-source projects usually maintain something called a *changelog* or *release notes*. It is a very simple text file that contains all the important changes between versions. The KnotDNS also has these release notes (`https://gitlab.nic.cz/knot/knot-dns/-/blob/master/NEWS`) and it is possible to find there that one bug was fixed in version 2.7.3: *"Improper processing of an AXFR-style-IXFR response consisting of one-record messages"*! The intention now is clear - install the old, broken version (2.7.2) and try to write a test case, which would detect this bug.

It is going to be useful to be able to easily change KnotDNS versions and try to run the test case with different implementations. Therefore, the *git* repository with all the source code is cloned.

Then, it is intended to install a particular version. Every version is specified via a git *tag* and the installation process is described in the `README.md` file.

1. `git checkout v2.7.2`

2. `autoreconf -it`
3. `./configure`
4. `make`
5. `sudo make install`
6. `sudo ldconfig`

The correct installation can be verified by printing the version - `knotd --version`.

Now, it is needed to write a test case for the situation, which was shortly described in the release notes. It says that it is about AXFR answers on IXFR queries and that those responses are *"consisting of one-record messages"*.

There are two possible ways how to understand this sentence:

1. The content of the AXFR message contains only one single resource record, so the whole zone consists only of one record.
2. One zone transfer message is split into `N` DNS messages, where every message is sent separately and every message contains only one single record in the Answer section.

To test the first case, a new `TestCase` has to be written. This test case will need a way how to generate the SOA records. For this purpose, a `AbstractRecordGenerator` (which generates SOA resource records) from the complex test cases can be used. It saves a lot of work and code! Again - great modular design and reusable components are paying off. Just pass it through the constructor as an argument.

The initial zone is not important and can be empty (so the method has only one line of code). The most important part is generating the changes. Only AXFR responses are intended to be used, so it will return an `AxfrResponseManager`. And this response manager will contain only the SOA record and one other record. It can be written on 4 lines of code - update the SOA, create a *RRset*, add a record into this `RRset` and finally return the response manager.

Lastly, the `get_zone` method must be implemented. Since the zone really contains just two records, it is also simple. The dns.zone.Zone instance can be built directly or using the `get_zone_from_records` utility. It takes again only 3 lines of code.

After all, the `require_zone` method can be overridden to always return `True`, so the zone check is required in every iteration, and therefore it is possible to watch the server more closely.

The test case is finished and the important parts are only on less than 10 lines of code! Perfectly according to the requirements.

The test case is prepared, KnotDNS is installed with the specified version and the configuration is ready from the previous testing. It is possible to run the scenario. Unfortunately, the test results are successful.

It seems like it was not the bug. But there is still the second possible explanation of the release notes and that is: *"One zone transfer message is split into `N` DNS messages, where every*

message is sent separately and every message contains only one single record in the answer section."

It has nothing to do with the changes generation. It is about transforming some data into messages. For this purpose, there is the response manager component! Firstly, clarify the goal:

Every zone change should be propagated into the secondary server via AXFR and it should consist of multiple messages where every single message contains only one single resource record in its Answer section.

There already is one implementation of the abstract `ResponseManager`, which answers only using the AXFR - `AxfrResponseManager`. It is possible to extend it and just add the second feature - the content splitting. In the parent class, the message splitting is performed in the `_get_messages(request, answer)` method, so it is possible just to override this method and adjust the behaviour to the new purpose (the purpose is to create a special message for every resource record). Arguments of this method are the request message and a list of RRset instances to be answered.

Every message is supposed to contain only one resource record. It is possible to simplify it - insert only one `RRset` (not record) into every message and then use a test case, which generates only one record per `RRset`. For example, the new, previously implemented, test case. The splitting of content is just one simple *for*-loop.

The implementation of the `ResponseManager` is showed in the Figure 5.10, it is just a few lines of code again. Then, this response manager can be used in the test case and the testing may continue.

This time the test successfully fails! The server output is illustrated in the Figure 5.11. It can be seen that the server tries to process the zone transfer, but it fails to free some memory and is immediately aborted. The connection between the secondary server and the `xfr_tester` is unexpectedly closed, which causes an exception, which is then printed on the standard output.

```
class OneByOneAxfrResponseManager(AxfrResponseManager):
    def _get_messages(
        self, request: Message, answer: List[RRset]
    ) -> List[BytesMessage]:
        ret = []
        for rrset in answer:
            response = make_response(request)
            response.answer.append(rrset)
            ret.append(response)
        return ret
```

Figure 5.10: Implementation of a ResponseManager for special AXFR-style-IXFR answers.

Now, upgrade the server to version 2.7.3, which is (according to the release notes) supposed to fix this bug. The newer version can be installed in the very same way, which was described before. The only change is that the checkout must lead to the tag `v2.7.3`.

After re-installation, run the very same test with the very same configuration. The test finishes successfully, as it can be seen in the log, which is illustrated in the Figure 5.12.

```
user@ntb:~$ knotd -v -c knot.conf
...
info: Knot DNS 2.7.2 starting
....
info: notify incoming, 127.0.0.1@44265: received, serial none
info: refresh outgoing, 127.0.0.1@5302: remote serial 2, zone is outdated
info: IXFR incoming, 127.0.0.1@5302: receiving AXFR-style IXFR
info: AXFR incoming, 127.0.0.1@5302: starting
free(): double free detected in tcache 2
Aborted (SIGABRT)
```

Figure 5.11: The KnotDNS server output, failure.

```
user@ntb:~$ knotd -v -c knot.conf
...
info: Knot DNS 2.7.3 starting
...
info: AXFR incoming, 127.0.0.1@5302: starting
info: AXFR incoming, 127.0.0.1@5302: finished, 0.00 seconds,
                                     1 messages, 269 bytes
info: refresh, outgoing, 127.0.0.1@5302: zone updated, serial none -> 1
...
info: notify incoming, 127.0.0.1@39987: received, serial none
info: refresh outgoing, 127.0.0.1@5302: remote serial 2,
                                        zone is outdated
info: IXFR incoming, 127.0.0.1@5302: receiving AXFR-style IXFR
info: AXFR incoming, 127.0.0.1@5302: starting
info: AXFR incoming, 127.0.0.1@5302: finished, 0.00 seconds,
                                     2 messages, 120 bytes
info: refresh outgoing, 127.0.0.1@5302: zone updated, serial 1 -> 2
....
info: notify incoming, 127.0.0.1@36089: received, serial 10
...
info: refresh outgoing, 127.0.0.1@5302: zone updated, serial 9 -> 10
...
```

Figure 5.12: The KnotDNS server output, success.

The bug has been corroborated and moreover, a test case has been written. From now on, there is a prepared scenario, which ensures that this bug will not happen again.

This was exactly the way, how is this `xfr_tester` supposed to be used. Find a bug or a suspicion and write a test case for it. Creating a new scenario is really simple, as it could be seen, and moreover, thanks to the modular design it is possible to combine the modules and get a larger scale of scenarios.

Note that this bug could have been detected even with the `ComplexTestCase` with a non-zero probability. According to analysis via Wireshark, the KnotDNS server terminates after receiving the second message. Now imagine the following situation: `ComplexTestCase` generates some changes, which are too big for one message. Therefore the message must be cut in two. This cutting mechanism is random (described in the Section 4.1.5 about response managers), so with a small (but non-zero) probability, it can be cut right after the first `RRset`, which is exactly the problematic case.

Chapter 6

Conclusion

The final version of the testing tool meets all the requirements specified in the Section 3 (Requirements and expected functionality), as listed at the end of the Section 4.4 (Requirements fulfilment). It is easy to use and easy to extend, so it can be used for testing any scenarios.

The abstract design was described in detail as well as the basic set of test scenarios and then the KnotDNS was tested with successful results. The newest version of KnotDNS (currently 3.0.0) has correctly processed all the test scenarios from the basic set of test cases and also has correctly processed the more complex test cases which were randomly generated.

Also, one bug in an old version was discovered and verified. A new test scenario for this bug has been written, so for future versions, this problem is covered and should never happen again. It was also checked that this bug has been fixed in version 2.7.3.

New test cases can be added according to specific needs of particular server implementations (as illustrated in the Section 5.3.1), but in general, it would be useful to have some more tests, for example, to check the ability to refuse an invalid transfer or the ability to work with the digital signatures of resource records sets (the `RRSIG` (= 46) type), which is important for security reasons.

There are also many ideas for other possible enhancements, not related to testing scenarios. First of all, it would be really useful to be able to run a whole list of test scenarios, not only one case. It could be easily done by wrapping the core into a loop. Another nice-to-have improvement is to create a command-line interface, using which it would be possible to configure the testing process already from the command line, so it would not be needed to write the configuration in the python code. It could be implemented with the `click` library, which shows up to be the best in comparison with `argparse`, `docopt` or `typer`. Also, more sophisticated error handling might be used. At this version, not all exceptions are caught - for example when the tested server dies (as it happened with the old version of KnotDNS), the `xfr_tester` just raises an exception, but does not terminate itself. The testing process must be then terminated manually. This issue could be easily solved by adding one simple `try..except` construct, or also more of these constructs on multiple places to have more detailed information about all these exceptions.

Note that the final implementation of the testing software can be found in a public repository at `https://gitlab.nic.cz/knot/xfr-tester`.

Bibliography

[1] P. Mockapetris, *Domain names: Concepts and facilities*, RFC 882, Obsoleted by RFCs 1034, 1035, updated by RFC 973, Internet Engineering Task Force, Nov. 1983. [Online]. Available: `http://www.ietf.org/rfc/rfc882.txt`.

[2] ——, *Domain names: Implementation specification*, RFC 883, Obsoleted by RFCs 1034, 1035, updated by RFC 973, Internet Engineering Task Force, Nov. 1983. [Online]. Available: `http://www.ietf.org/rfc/rfc883.txt`.

[3] ——, *Domain names - concepts and facilities*, RFC 1034 (Standard), Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936, Internet Engineering Task Force, Nov. 1987. [Online]. Available: `http://www.ietf.org/rfc/rfc1034.txt`.

[4] ——, *Domain names - implementation and specification*, RFC 1035 (Standard), Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, Internet Engineering Task Force, Nov. 1987. [Online]. Available: `http://www.ietf.org/rfc/rfc1035.txt`.

[5] P. Vixie, *A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)*, RFC 1996 (Proposed Standard), Internet Engineering Task Force, Aug. 1996. [Online]. Available: `http://www.ietf.org/rfc/rfc1996.txt`.

[6] E. Lewis and A. Hoenes, *DNS Zone Transfer Protocol (AXFR)*, RFC 5936 (Proposed Standard), Internet Engineering Task Force, Jun. 2010. [Online]. Available: `http://www.ietf.org/rfc/rfc5936.txt`.

[7] M. Ohta, *Incremental Zone Transfer in DNS*, RFC 1995 (Proposed Standard), Internet Engineering Task Force, Aug. 1996. [Online]. Available: `http://www.ietf.org/rfc/rfc1995.txt`.

[8] J. Klensin, *Application Techniques for Checking and Transformation of Names*, RFC 3696 (Informational), Internet Engineering Task Force, Feb. 2004. [Online]. Available: `http://www.ietf.org/rfc/rfc3696.txt`.

[9] R. Elz and R. Bush, *Clarifications to the DNS Specification*, RFC 2181 (Proposed Standard), Updated by RFCs 4035, 2535, 4343, 4033, 4034, 5452, Internet Engineering Task Force, Jul. 1997. [Online]. Available: `http://www.ietf.org/rfc/rfc2181.txt`.

[10] P. Faltstrom, P. Hoffman, and A. Costello, *Internationalizing Domain Names in Applications (IDNA)*, RFC 3490 (Proposed Standard), Obsoleted by RFCs 5890, 5891, Internet Engineering Task Force, Mar. 2003. [Online]. Available: `http://www.ietf.org/rfc/rfc3490.txt`.

[11] R. Elz and R. Bush, *Serial Number Arithmetic*, RFC 1982 (Proposed Standard), Internet Engineering Task Force, Aug. 1996. [Online]. Available: `http://www.ietf.org/rfc/rfc1982.txt`.

[12] D. Barr, *Common DNS Operational and Configuration Errors*, RFC 1912 (Informational), Internet Engineering Task Force, Feb. 1996. [Online]. Available: `http://www.ietf.org/rfc/rfc1912.txt`.

[13] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound, *Dynamic Updates in the Domain Name System (DNS UPDATE)*, RFC 2136 (Proposed Standard), Updated by RFCs 3007, 4035, 4033, 4034, Internet Engineering Task Force, Apr. 1997. [Online]. Available: `http://www.ietf.org/rfc/rfc2136.txt`.

[14] A. Gustafsson, *Handling of Unknown DNS Resource Record (RR) Types*, RFC 3597 (Proposed Standard), Updated by RFCs 4033, 4034, 4035, 5395, 6195, Internet Engineering Task Force, Sep. 2003. [Online]. Available: `http://www.ietf.org/rfc/rfc3597.txt`.