Final Report

DNS Infrastructure Revamp

Kevin van Nes
Tung Phan
Martin Rogalla

Final Report
DNS Infrastructure Revamp

by

Kevin van Nes
Tung Phan
Martin Rogalla

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,
to be defended publicly on 19 September 2014 at 16:30.

Supervisors:	Dr. Georgios Gousios	TU Delft
	Johan Schuijt	TransIP
Bachelor Project Coordinator:	Dr. ir. Martha Larson	TU Delft

This report remains confidential until 19 September 2014.

Contents

Preface

This report concludes the TI3800 Bachelorproject course, which itself is a compulsory course to gain the Bachelor of Science in Computer Science degree at the Delft University of Technology. This report contains all information about the project that was done over a span of ten weeks at TransIP. The project client was TransIP, a domain- and webhosting company located in Leiden. During these ten weeks the whole DNS back-end of TransIP was restructured from a pull-based structure to a push-based one. Furthermore, this report's goal is to inform the reader about the completed work and the different phases throughout which this was done. The report will also contain recommendations for future work on this project, as the project has gone live at the end of the project and co-employees of TransIP might have to work on the system in the future.

Summary

TransIP, an Internet Service Provider with a focus on web hosting and domain name registration, requested a restructuring of their DNS system. Customers who own a domain at TransIP can modify their records on their TransIP configuration page. These changes are then picked up by several Cron-jobs, which executed scripts at a fixed interval. However, this pull-like fashion of update checking was too slow for TransIP and thus requested a restructuring.

Research has been done on the field of computer networking regarding DNS, the bottlenecks in the current system and the requirements of the project and the infrastructure of the current system. Knowledge of the current system was required for a solid design in order to prevent code regression. The three key words of design were performance, security (or robustness) and maintainability. The main part of the solution for transforming the pull-based system to a push-based system is database replication. Changes by the customer are pushed to a newly created database, TransDNS, and are replicated by the slave databases in the name servers. The structure of the name servers itself did not need any noteworthy modifications. The system was thoroughly tested using PHPUnit and was programmed with testing in mind, using techniques as defensive programming and peer programming.

Acknowledgements

Many people helped us in realizing this project. We would like to thank a few of them in specific. First of all, we would like to thank Johan Schuijt, TransIP's CTO and also our project supervisor at TransIP's side, for guiding us through this project and giving us feedback wherever and whenever it was necessary. Furthermore, we would like to thank our TUDelft supervisor, Georgios Gousios, for helping us in creating and finalizing this thesis and also for giving us feedback throughout the project's duration. We would also like to thank Martha Larson, the Bachelorproject coordinator, who helped us and gave us advice during the time that mr. Gousios was not available. Last, but not least, we would like to thank everyone at TransIP for all of their help and kindness.

LIST OF FIGURES

LIST OF TABLES

1 Introduction

TransIP is an Internet Service Provider (ISP) founded in 2003 and has since then grown to be the largest ISP of the Benelux. Besides providing services like web hosting and registering domain names, the company aims to be easily approachable by their customers, maintaining a high standard in customer service and offering technical support whenever necessary. Speed is key in this business, so in order to compete with other Internet Service Providers, they wrote their own Domain Name Server (DNS) system, because they were not satisfied with the systems that were available at the time and they wanted to have more control over their servers. The current system periodically checks for so-called zone changes and propagates them to the corresponding databases. At the time, this method was acceptable as it was relatively easy to implement. However, its updating speed is now far from desirable, as the speed is heavily affected by the multitude of periodic checks: in the worst case scenario, the updates take several minutes before they are stored in the databases of the name servers. TransIP wants to reduce this time to several seconds, which requires a restructure of parts of the system.

The difference between this project and typical Bachelor projects is the fact that work had to be performed inside a big, already existing system, instead of having to create a system from scratch. This difference brought up a few challenges of its own and these challenges were handled and tackled in different ways. Research was conducted in relation to these challenges, how to solve them and where to be careful during the project. More about this can be read in Section 3.2.

1.1. Structure of this Document

Before work could be started on revamping TransIP's DNS structure, a lot of research had to be done, both inside and outside of the system. In Chapter 2 all matters regarding the initial phase, the Research phase, will be discussed. The research that was conducted, combined with both our own ideas and ideas from TransIP's employees, led to the design and implementation of an improved DNS system for TransIP. This new system and its design and implementation will be expanded upon in Chapter 3 and 4 respectively. After this, Chapter 5 will be dedicated to the ways the new system was tested before it could be deployed to the live server. Next, an evaluation will be done of the requirements, design goals and success criteria that were set during the research phase in Section 6. Finally, the whole project will be reflected on and discussed and recommendations will be given in Chapter 7.

2 Research

2.1. Overview

The first two weeks of this project were fully dedicated to doing research. In the following chapter, many of the aspects and results of this phase will be discussed. The problem that was at hand had to be defined and analyzed in a concrete manner. In Section 2.2 this definition will be given and an analysis will be done. To give an idea of what the old DNS infrastructure looked like, a description and diagram of it will be given in Section 2.3. During the research phase design goals were set, which will be considered whenever a significant decision is made during any of the phases of the project. These design goals (together with the requirements, which will be described shortly) are elaborated upon in Section 2.4. In the section after this, Section 2.5, the requirements for this project will be stated. These requirements give an idea as to what properties the system should have at the least to make for a complete system. Hereafter, the software development methodology used during this project will be discussed in Section 2.6. Finally, all results of the research phase will be discussed in the last four sections. DNS itself will be discussed in Section 2.7. TransIP's DNS system, TransDNS, will be described and compared to other DNS systems in Section 2.8. Next, the results of researching database replication will be elaborated in Section 2.9 and lastly, in Section 2.10, different ways and techniques related to performance testing will be discussed.

2.2. Problem Definition and Analysis

As stated above, the first part of the research phase consisted of defining and analyzing the problem that is addressed by this project. In Section 2.2.1, the problem will be clearly described and defined to give a good view of what exactly needs to be solved during this project. In Section 2.2.2, the problem will be further analyzed. The needs and motivation of the client that led to the birth of this project will be stated and the context of the project will be elaborated upon.

2.2.1. Problem Definition

In 2003, TransIP was founded. At that point, the founders decided that they wanted full control over their DNS system. Because of this, TransDNS (TransIP, 2014) was created. TransDNS is the DNS-server software running on TransIP's nameservers. The system built around and on top of TransDNS encompasses the complete DNS infrastructure of TransIP.

TransDNS worked well in 2003 and the system is still very capable of doing what it is supposed to do. However, the process of propagating changed records is not fast enough anymore, as the amount of competition is getting higher. In order for TransIP to compete with other systems, this process has to become faster.

[redacted]

[redacted]

Figure 2.1: Schematic diagram of TransIP's old DNS infrastructure

[redacted]

2.2.2. Problem Analysis

TransIP wants to hold the dominant position on the market by speeding up the resource record propagation process for its clients, so a solution for the problem that was described above is wanted. During this project a solution will be designed, implemented and deployed. As stated above, TransIP wants to stay ahead of the competition and this can only be done if every part of their system is top-notch. Speeding up the resource record propagation time will have positive effects both on customers and on staying ahead of the competition.

Slow propagation of DNS zone changes throughout the TransIP DNS infrastructure directly affects the time it takes for customers to see their zone changes. This delay is too long, especially if TransIP wants to compete on the market, so a solution to this problem is of vital importance to stay ahead of the competition and to keep customers satisfied.

2.3. Description of Old DNS Infrastructure

In this section, TransIP's old DNS infrastructure is elaborated upon using the diagram found in Figure 2.1. All referrals to 'the figure' in this paragraph refer to this figure. The description of the old system should give a basic understanding of how the system used to work (and what it will work like until successful completion of this project).

[redacted]

2.4. DESIGN GOALS

In this section, the main design goals for this project will be elaborated. These design goals are of high importance and have always been considered while designing, implementing and deploying the new system. All of these design goals will be used to create a collection of requirements. The main design goals set for this project are: performance, security and maintainability. Performance is the design goal that has the highest priority of all three. If the system does not perform well, the system can not be properly completed and will not be able to run on a live server. After performance, security is the most important design goal.

Since this system runs on a live server, malicious attacks on the system might occur and the system should be failure-proof to these kinds of threats. Maintainability is the final design goal priority-wise. However, maintainability is still critically evaluated with every important decision that is made.
The design goals are elaborated on in Section 2.4.1, Section 2.4.2 and Section 2.4.3, respectively.

2.4.1. Performance

Performance is the first main design goal of this project. The new system must produce the same results as the old system did, but in a different and more efficient way. Performance is considered very important as a whole, but there are two important sub-goals that should also be considered individually: data consistency and improved update propagation.

Data Consistency

There are multiple databases and respective backups contained within the system (see Appendix 2) and the data in these databases is not allowed to lose its consistency. If a resource record is in one record database, it should be in all record databases.

Improved Update Propagation

Whenever a user changes his resource records, this change is propagated through the DNS infrastructure. In the old system, a change could take up to five minutes to propagate through the system. One of the main goals of this project is to make this propagation more efficient. Therefore, any implementation decision that is made should be based on this design sub-goal.

2.4.2. Security

Security is another main design goal that has been worked towards during this project. The new system must be implemented as securely as possible, so it will not be vulnerable to any attacks from the outside. Two sub-goals in particular that will help keeping a secure system are the use of defensive programming and validating input. Both of these sub-goals are discussed below. It must be noted however, that the system should also be secure from the inside. Modular independence will be described below as a sub-goal for being able to create a secure and stable system from the inside.

Defensive Programming

Asides from the fact that defensive programming is intended to increase code quality in general, defensive programming also asks the programmer to think of unexpected input or vulnerabilities of the system. Preventing vulnerabilities and unexpected input is one of the key elements of defensive programming. The TransIP DNS infrastructure needs to be very secure and as such, defensive programming is the programming approach that will be taken during this project.

Input Validation

Something that is closely linked to defensive programming is the validation of user input. Incorrectly entered input from normal users might lead to an error message, but when a person who wants to attack the system knows just the right invalid input to feed to the system, this attacking person might receive information that he should not be receiving. Vulnerabilities as part of input validation are extremely important to prevent. Thus, a lot of time and effort will be put into correctly validating different kinds of user input.

Modular Independence

Modular independence is a design sub-goal that helps both in reaching Security as well as Maintainability. Keeping the system's modules independent will create a more secure system. If this is applied correctly and one part of the system fails, the rest of the system will not fail consecutively.

2.4.3. Maintainability

Maintainability is the last of the main design goals of the project. It is important that other developers can easily modify and understand the newly produced code. Problems that occur should be extensively logged, allowing for fast emergency response, decreasing downtime and thus in turn increasing performance. Some sub-goals have been created that should add to the maintainability of the product. These are code quality, testability, deployment and modular independence, which are all discussed below.

Code Quality
The PHP code should adhere to the PSR-2 code style. JavaScript code should should be validated with Google Closure. MySQL queries should take less than one and a half seconds, because queries that take more than two seconds could cause a system crash.

Testability
Testability fully depends on modular independence, which is discussed in Section 2.4.2. High coupling between modules would decrease testability as it is difficult to test highly dependent subsystems, due to complex and large initialization procedures.

Deployment
Before the product is deployed, it is important to verify that all the code conforms to the requirements set in TransIP's internal wiki. Communication between the project team and the involved TransIP employees is of vital importance; it is important that the involved teams are notified of the changes that are to be made. For safety reasons, the deployment phase will be supervised and assisted by Johan Schuijt more securely than the other phases.

Modular Independence
As stated above, modular independence is a design sub-goal for both Security and Maintainability. Modular independence improves testability of the system and traceability of errors, thus increasing maintainability. Having a modular independent system makes it easier to replace modules whenever a part of the system is broken and it will make it easier to extend the system in the future.

2.5. Requirements Analysis
In order to create clear objectives for the system itself requirements were set at the beginning of the project. These requirements were set making use of the design goals described above. They give a clear and concrete idea of what the system should adhere to at any moment and what it should look like when it is finished. In Section 2.5.1, the priority of the requirements is explained and the requirements themselves are given. The concretization of the requirements also allowed for the creation of a list of success criteria. These criteria are expanded upon in Section 2.5.2.

2.5.1. Requirements
The design goals that were set in Section 2.4 allow for the creation of a concrete list of requirements for the final product. The requirements will be listed in Table 2.1 along with their respective design goal or design sub-goal. These requirements, as well as the design goals, will be taken into account for every significant decision that is made throughout the duration of the project.

Requirement:	Design Goal:
The new system must produce the same results as the old system does. In other words: it should be fully backwards compatible.	Performance - Data Consistency
User's resource record updates must be propagated faster than in the old system.	Performance - Improved Update Propagation
The system should be as consistent as the previous system.	Performance - Data Consistency
system's downtime must stay at a minimum.	Security - Modular Independence
The system should be maintainable for co-employees.	Maintainability
All code in the system must adhere to the Software Engineering principles as given in the Software Engineering Methods course (part of the Bachelor of Science in Computer Science track of the TU Delft).	Maintainability - Code Quality
The system must be scalable.	Maintainability
The system must be tested properly as to minimize unexpected failures.	Maintainability
Deployment must be carefully supervised by TransIP's project supervisor Johan Schuijt.	Maintainability - Deployment

Table 2.1: Requirements and corresponding design goals.

The requirements are prioritized in the same way the design goals are prioritized. The most important requirement is performance. After that security is the most important requirement and finally maintainability will be taken into account.

2.5.2. Success Criteria

Before starting a software project it is important to make an estimation of when the project is done and successful. For this project, the design goals and requirements that were set gave a good idea of what the system should at least consist of and how it should perform to call the project a success. The two main design goals 'Performance' and 'Security', along with the corresponding requirements, are the two design goals with which the system must comply at the least before it could be called a complete and competent system. 'Maintainability' is still an important design goal, but the two other design goals get priority over this one. If at least the requirements corresponding to the 'Performance' and 'Security' design goals are reached, the system will be ready to be deployed to TransIP's live server. This summarizes the success criteria to one specific, concrete criterium: the system, and thus the project, can be called a success as soon as the system has been deployed to TransIP's live server.

2.6. Development Methodology

During this project Scrum was used as the main software development methodology. This was chosen immediately at the start of the project for multiple reasons. One of these reasons is the fact that this project had a relatively short time span and during this time span a lot of different features had to be implemented within a narrow scope. Using Scrum, and the sprints in which different phases of the project are split up, helped planning everything in a general matter, although the planning would still be open for changes at a detailed level as time progressed. Another reason the Scrum methodology was chosen, is the fact that this is also the standard within TransIP. Sprints of two weeks are made, with milestones set at the end of each week. For this project this standard was applied in the exact same manner. More of the company's standards and protocols have been applied and used throughout the project, about which more can be read throughout the rest of this thesis.

Throughout this project, all members of the team will mostly communicate directly, seeing as all members are to be present at the TransIP office on weekdays from 9 a.m. until 5 p.m. This makes sure the division of labor is also well balanced, as each individual will work approximately eight hours per day. Tasks will be divided at the start of each day during a Scrum stand-up meeting and at the end of each day a short evaluation of that day will be done.

2.7. DOMAIN NAME SERVER

2.7.1. OVERVIEW

During the Computer Science course 'Computer Networks', the topic of Domain Name Servers (DNS) was discussed briefly. However, in order to fully understand the current system and to be able to properly redesign the structure, the full concept of DNS has to be understood.

The general structure of DNS will be discussed in Section 2.7.2. In Section 2.7.3, name servers will be treated, along with the different types of name servers that exist. Name servers host so-called zones, which will be elaborated on in Section 2.7.4. Finally, resource records will be explained in Section 2.7.5.

2.7.2. GENERAL STRUCTURE

DNS is a distributed hierarchical naming system, used to find the IP address coupled to a domain name. One of the main advantages of this is the fact that it is easier to remember name-based strings, i.e. a URL, than a sequence of digits, i.e. an IP address. Another advantage is that the end-user will not be affected by changes made to the network's location, as the domain name will stay the same (Mockapetris, 1987a). DNS can be visualised as a tree structure, with the root name server at the root going down to the top-level domain servers and then, beneath these, the (sub)domains (Mockapetris, 1987a).

For example, 'www.example.com' has as top-level domain 'com', 'example' is a subdomain of 'com' and 'www' is a subdomain of 'example.com'. The root name server has no label attached to it, but is optionally represented by a trailing dot. The official string is 'www.example.com.', but the dot is left out most of the time. The complete form with trailing dot is called the absolute rooted fully qualified domain name, or absolute rooted "FQDN" (Gavron, 1993). An example of a portion of the DNS tree is shown in Figure 2.2. The zones indicate the subtrees over which an authoritative server has full control. More on this can be found in Section 2.7.4.

Figure 2.2: Portion of the DNS tree

2.7.3. NAME SERVERS

A name server is a computer responsible for translating domain names to IP addresses. As mentioned before, DNS is distributional, so a single name server can not translate all the domains on its own. In a case where the server is unable to answer the request by itself (e.g. it is not in its database) it will redirect the request to other name servers (Mockapetris, 1987b).

Essentially, there are two kinds of name servers, authoritative name servers and recursive name servers. Authoritative name servers are servers that can answer queries of clients by theirselves, either by looking in their database or by referring to another name server (Mockapetris, 1987b). These name servers are authoritative of a zone (discussed in Section 2.7.4), meaning that the answers of these servers can be trusted to be true. The IP addresses of the authoritative name servers that are responsible for the TLD's are known by the root name servers.

Recursive name servers, on the other hand, operate by answering queries stored in their cache if the queried domain has been requested before. If it is not found in the cache, the name server will recursively ask other servers on behalf of the client, in the worst cases all the way to the root name server, until an answer has been found (DNS Made Easy / Tiggee LLC, 2014). So instead of returning a referral to another name server to the client, the recursive name server keeps on querying other name servers recursively until an answer has been found, which is either the correct address or an error. The records in the cache have a set time to live, meaning they will expire after a while, causing the record to become obsolete (discussed in Section 2.7.5). If that happens, the data is dropped from the cache.

Compared to authoritative name servers, recursive servers can not guarantee that the IP address they find is up-to-date, unless the recursion ends up at an authoritative server. The caches could contain old records which are not updated yet, whereas authoritative servers always have the most recent records. In theory, using only authoritative name servers would suffice, but a combination is more efficient thanks to the caches.

Between authoritative servers, there are master and slave servers. Master servers, or primary servers, store their records locally, whereas slave servers, or secondary servers, update their records automatically, keeping their databases identical to the master server (Aitchison & Wilson, 2005). The idea behind this structure is to keep the name servers available, even if one of the servers is not reachable.

2.7.4. ZONES

Zones are essentially subdivisions of the DNS tree. Not to be confused with subdomains, the children of a domain, as zones can contain either a single domain, or several domains complete with their subdomains. Zones can be categorized by two types: primary zones, secondary zones and stub zones (Microsoft TechNet, 2014; Mockapetris, 1987a)

A name server hosting a primary zone has the authority of that zone, meaning that it has the full knowledge over that zone and is able to answer queries concerning that zone by itself (Microsoft TechNet, 2014). In other words, that server is an authoritative name server of that zone. This server has the master copy of the information concerning that zone and is the primary source of information. A name server can only host one primary zone. Whenever changes occur within that zone, the authoritative name server of that zone can write those changes directly in its database. In a sense, the zone is the read/write part of the database.

When a name server hosts a secondary zone, it means that the name server is the secondary source of information about that zone. In order to obtain information about that zone, the name server has to request it from a different name server. This can be an authoritative name server of that zone, but that is not always necessary. The name server could also be the secondary source (Microsoft TechNet, 2014). A single name server can host at most one primary zone, but it can host multiple secondary zones. If the primary zone is the read/write part of the database, the secondary zones are the read-only parts of the database. If changes occur in the secondary zones, the name server can only update its information about that zone by requesting it from other name servers.

Zones can be delegated into smaller zones. In this case authority will be given to a different name server, just like a manager who delegates a portion of the work to employees, who are then responsible for the job they are assigned to.

2.7.5. Resource Records

Resource records are the basic data building blocks of the domains. Records are stored in zone files, which are text files describing a zone. In previous sections, records describing the zones were depicted as if they were immediately stored in a database, whereas they are actually stored in text files first. This is BIND's way of storing records, which became common practice with other systems, but some systems use zone files only as a start up in order to create the database (see Section 2.8.3).

There are different kinds of record types and only the common and relevant ones will be discussed in this section. All resource records follow a standard format (Mockapetris, 1987b):

- The Owner field stands for the domain name that owns the record (in FQDN form).
- The Time-to-Live (TTL) field indicates the time the record stays in a cache before it expires and is discarded from the cache. This field is optional and if it is left blank, it will have the default TTL specified in the SOA record, which is discussed below.
- The Class field indicates the class to which the record belongs. The most common class is IN, which stands for Internet. Other classes are for example the Chaos (CH) and Hesiod (HS) classes.
- The Type field indicates the type of resource record.
- The RData field varies according to type and class, RDLength indicates the length of RData. These two fields are less relevant and will be skipped in this report.

All zones must contain a Start of Authority (SOA) resource record at the beginning of a zone. SOA records contain the following fields:

- Owner, Time-to-Live, Class and Type fields, as discussed above.
- The authoritative name server for the zone.
- The e-mail and name of the administrator of the zone. @ is replaced by a dot.
- The serial number, primarily used to check if one zone is newer than the other. For example, if this zone is a secondary zone, it will have to check with a corresponding primary zone if the secondary zone is outdated. If the serial of the secondary zone is lower than the primary, then a zone transfer will take place, copying the zone of the primary (along with the new serial) and updating the secondary zone.
- The refresh field indicates how often the name server will check with the secondary zone whether the zone is up-to-date or not.
- The retry field indicates how long the secondary server will wait for a response after sending a zone transfer request to the primary server before retrying.
- The expire zone indicates how long the secondary server for the zone stays valid after the last zone transfer before discarding its own zone as invalid. This time refreshes every zone transfer.
- The default TTL for every record, used if the TTL is not specified in a record itself.

Below, an example of a SOA record is given:

```
example.com. IN SOA (
ns0.example.com. ; authoritative server for the zone
admin.example.com. ; zone admin e-mail
5099 ; serial number
3600 ; refresh (1 hour)
600 ; retry (10 mins)
86400 ; expire (1 day)
60 ) ; minimum TTL (1 min)
```

Asides from SOA records, the following three types of resource records are also commonly used:

’A’ (address) resource record types map a domain name to an IP address. For example, the following maps the domain example.com to the IP address 93.184.216.119:

```
example.com. IN A 93.184.216.119
```

’NS’ (name server) resource record types indicate the authoritative name server of the zone. This record has to be present at the delegated zone as well as the parent zone.

’CNAME’ (canonical name) resource record types indicate an alias set for the domain name. This is especially useful if several records refer to a particular domain name. When the aliased domain name is changed, the amount of records changed is minimized.

2.8. TransDNS

2.8.1. Overview

In this chapter, TransDNS will be briefly described. Firstly, TransDNS will be compared to other popular systems in Section 2.8.2. After this, the reasoning behind why TransIP chose to develop their own system will be discussed in Section 2.8.3.

2.8.2. Comparison

TransDNS is a DNS system designed and developed by TransIP. In order to understand the choice of TransIP to develop their own system instead of using existing DNS systems, research on the other systems was required. Thus, three popular DNS systems (BIND, PowerDNS and NSD) were explored, highlighting their strong and weak points.

BIND

Berkeley Internet Name Domain, or BIND, is one of the most widely used DNS systems, especially on UNIX-like systems. The current stable release, BIND 9, is a complete rewrite, as BIND 4 and BIND 8 had serious security vulnerabilities. Despite the massive rewrite, BIND is still ridden with security vulnerabilities according to the CVE statistics (CVE Details, 2014). This fact, however, should be taken with a grain of salt, because of the popularity of the system: the more popular it is, the more security exploits will be discovered. BIND makes use of zone files, which are essentially text files containing the records instead of databases. However, as TransIP operates on a large scale with more than a million domains, zone files alone will be highly inefficient in terms of speed and maintainability. Still, BIND is capable of utilizing a database, but it requires a plugin (DLZ, or Dynamically Loadable Zones) to do so and its performance is bad compared to other DNS setups (South Asian Network Operators Group, 2009). A study by Shen and Schulzrinne (2006) further shows that BIND performs particularly bad at a large scale, because of the high CPU usage and the fact that the whole zone file needs to be reloaded when records are updated, making it unable to answer queries. The upside is that BIND is relatively easy to set up, but TransIP has enough resources available to consider other options. As a result, the cons clearly outweigh the pros.

PowerDNS

PowerDNS is a more modern DNS system designed with better security solutions in mind compared to BIND. One of the differences between BIND and PowerDNS is the fact that PowerDNS has a separate authoritative and recursive server, whereas BIND has these combined. This offers more flexibility and security to PowerDNS, as authoritative servers generally do not use caches, avoiding cache poisoning (Halley, 2008). PowerDNS was also designed with a database backend in mind, increasing its scalability. Even though PowerDNS is an improvement compared to BIND, TransIP still found the performance of PowerDNS lacking and desired a better system.

NSD

NSD is the newest system of the three, designed from scratch as a strict authoritative-only system by design. First developed to introduce more variety for root servers as a form of genetic diversity, NSD is now currently running on several root servers and TLD registrars (NLNetLabs, 2013). Because NSD was designed as an authoritative-only system, it is optimized as such, unlike BIND and PowerDNS, which act more as a jack of all trades. Since TransIP is a registrar, it has no need for recursive servers or caches and prefers authoritative servers, making NSD the perfect candidate for TransIP as DNS system. The reason TransIP did not opt for

NSD could be because it was too new and, as with all other systems, because TransIP wanted full control over their system.

2.8.3. TRANSDNS

Because TransIP is an ISP that sells domain names, it wants a DNS system focused primarily on authoritative servers. With BIND's lackluster performance and the fact that it was not fully RFC-compliant, BIND was dropped as an option.

PowerDNS was designed with a database backend in mind, so PowerDNS was a good candidate. However, even though BIND was not RFC-compliant, it was so widespread, it became the standard, with PowerDNS following in its footsteps. Along with the reasoning that PowerDNS was lacking performance-wise as described above, it decided not to go with PowerDNS.

As NSD is an authoritative-only server, it has focused its resources on those features, ensuring high quality. TransIP has considered switching over to NSD, but due to the fact that responses by both systems differ, compatibility issues will most probably arise.

[redacted]

[redacted]

2.9. DATABASE REPLICATION

2.9.1. OVERVIEW

In this chapter, the results regarding the research about database replication will be expanded upon. Firstly, the current system of TransIP will be discussed in Section 2.9.2. After that, in Section 2.9.3, different techniques for replication will be described. For each of these techniques the advantages and disadvantages will be given. These different techniques will be compared to each other and conclusions will be drawn in Section 2.9.4.

2.9.2. CURRENT SYSTEM

As was stated in Section 2.2.1, the process of propagating new domain entries to the databases of the TransDNS servers is very slow. The process consists of several periodic steps, which in a worst case can cause a delay due to bad queue timing. A new way has to be found in which new domain entries can be saved in a more efficient fashion than the current one. This new approach will most likely involve replacing the TransList text file with a database. For this purpose, research was done on the topic of database replication.

Database replication needed to be understood by the project team, because no matter what the design for the solution of the given problem (i.e. the revamping of TransIP's DNS infrastructure) will look like, a new database will most likely need to be setup in which all domains are saved.

If a new database needs to be setup, this database will have to be backed up by one or more so-called slave databases for security reasons. Different ways of backing up or 'replicating' a database exist and the techniques for doing so are discussed in the next section.

2.9.3. REPLICATION TECHNIQUES

When a database is replicated, the database that is replicated is called the master, while the database that becomes the replica is called a slave. A master can have more than one slave if this is wanted. If replication is wanted, the master database will have to write events into its binary log, from which slaves can read. Based on what these slaves read they will be able to replicate the master.

Currently, there are three main techniques for replicating databases: synchronous replication, asynchronous replication and semisynchronous replication, which are described in the paragraphs below.

SYNCHRONOUS REPLICATION

Synchronous replication, also known as eager replication, is the technique that is used when one wants to be very sure of the consistency of the slave's data. When synchronous replication is used, all slaves will attempt to replicate the master based on its binary log. Only after all of the slaves have confirmed that they are done replicating will the master commit its changes. This technique makes sure that every change on the master's side is always exactly replicated on the slaves' side, which results in high consistency of data. However, because the master only commits after receiving a confirmation from all slaves, the possibility of deadlocking arises quickly (Rantanen, 2010).

Another disadvantage of synchronous replication is that it is slow. Depending on the time-out value of the master it may take several seconds for each change to be committed.

ASYNCHRONOUS REPLICATION

Asynchronous replication, also known as lazy replication, is the opposite of synchronous replication. With asynchronous replication, a change is first committed on the master's side, after which the changes are propagated to the slaves' side. In this case, the master will not have to wait for confirmation from all of its slaves before committing a transaction. This type of replication is much more efficient than synchronous replication, but the drawback is that data on slaves may become inconsistent with that of the master (Rantanen, 2010). In the old DNS system of TransIP, data replication was done using asynchronous replication using a kind of monitoring to ensure data consistency.

SEMISYNCHRONOUS REPLICATION

Semisynchronous replication is a relatively new type of replication that can be seen as a middle-way between synchronous and asynchronous replication. Semisynchronous replication works almost the same as eager replication, but the difference is that the master will only need to receive a confirmation from a single slave before being able to commit a transaction. This is faster compared to synchronous replication and data will be more consistent than it would be when using asynchronous replication (MySQL, n.d.).

2.9.4. RESULTS

From the description above it can be concluded that synchronous replication will not work in an application that has as many users as TransIP's DNS system. Such a system deals with too many queries, possibly at the same time, in which case synchronous replication would have a high risk of being too slow or causing a deadlock.

The decision that needs to be made is whether asynchronous or semisynchronous replication will be chosen for the implementation. On one hand, semisynchronous replication offers higher data consistency than asynchonous replication does, but asynchronous is much faster. There are also extensions to asynchronous replication that ensure higher data consistency.
For the implementation of the new system the type of replication needs to be well-considered, making use of the gathered information and the results that were found.

2.10. PERFORMANCE TESTING

2.10.1. OVERVIEW

TransIP handles over a million domain names, meaning that the DNS system should be able to handle the load, before and after the restructure. In order to identify possible bottlenecks and to proof that the redesigned system is an improvement performance-wise, performance tests will be done. There are different types of performance tests, but the tests most relevant to the project will be discussed in Section 2.10.2.

2.10.2. TESTING TYPES

As stated in the overview, there are different types of performance tests, but the most relevant types will be discussed in the following subsections (TestingGeek, 2011). This does not necessarily mean that all of these testing types will be performed on the final system. Only the types of tests that really serve a purpose for the final system will be performed.

Load Testing

Load testing means testing the system starting at a regular load. The load is then steadily increased until the system nears its optimal performance threshold. There are several purposes to this test. One of these is that this kind of testing is a mandatory sanity check, to see if the system can actually perform under normal conditions. This is used as the baseline to compare the system to. The other purpose is to test the limits of TransDNS and identify the boundaries of it.

Stress Testing

In order to test its ability to gracefully recover in extraordinary cases, stress tests will be performed. The system is subjected to a series of tests where the conditions are suboptimal. An example of such a test is entering an extreme amount of queries way beyond the hardware capabilities of the system. TransIP could be targeted by a Distributed Denial of Service (DDoS) attack, so the behaviour of the system during those attacks must be analyzed. DDoS-attacks can be mimicked with this type of testing. The results of these tests can be used to handle actual attacks.

Soak Testing

Also known as endurance testing, to soak test a system means subjecting it to a load under normal conditions over a longer period of time. This type of testing will check if the behaviour of the system is consistent, so to see if there are any random behaviours or memory leaks as time passes.

Spike Testing

Spike testing is similar to stress testing, with the difference between them the duration of the loads. Spike testing will repeatedly increase the load and drop it back to normal conditions. The goal of this test is to see how the system reacts to sudden spikes and if the system can operate normally after said spike.

Scalability Testing

TransIP is ever growing bigger, meaning the new system must be able to scale with the growing demand. Scalability tests, or capacity tests, involve testing the system's capability scale either up or out. Performing these tests helps in determining when and how many more resources are needed to perform on the same level.

3 DESIGN

3.1. OVERVIEW

Due to the complicated nature of this project, which comes forth from the complexity and size of the existent system, surgical precision is required for each modification. Therefore a lot of energy is put into creating well-thought-out plans to implement new features. In this chapter, all decisions and results of the design and implementation are discussed and after having read this chapter, it should be clear what decisions were made to come to a design that fits the requirements set in Section 2.5.

First, in Section 3.2, an explanation is given on how the required precision gives rise to certain challenges and how these challenges influenced our way of working. In Section 3.3, the initial planning of the restructuring of the system is elaborated upon. Here, an explanation will be given of the order in which the system was restructured and why this was done in such a way. All parts of the system received at least some attention during the design of the restructuring, but one part needed a lot of attention in specific, which was the way in which the "dns backend" and the nameservers would communicate. The decisions that were made around this part of the design are expanded upon in Section 3.4. Finally, in Section 3.5, the final design will be discussed and a complete overview of the designed system is given.

3.2. PROJECT CHALLENGES

As was stated in Chapter 1, this project had some noticeable discrepancies compared to the other bachelor projects. The main discrepancy was the fact that, instead of having to create a system from scratch, there was already a large existing system which required skills like software re-engineering instead of the skills for creating a new system.

This introduced numerous challenges, such as having to implement changes while avoiding code and performance regression, or having to avoid implementing a new feature that would silently break other parts of the system. The project involved changing a small part of the backend, but the main challenge was identifying all dependencies. Careless rewriting of code could have a disastrous effect, propagating errors through the whole system.

Another challenge was that the code that was subject to restructure was written a long time ago. It was untested and not well documented, making it harder to understand the code as for example the uniform coding style was introduced at a later time.

3.3. RESTRUCTURING

Prior to agreeing to the final design, several other designs were considered, which will be described below along with the thought process of the group.

The first challenge a person encounters when working in such a large system is finding out where to start working; it was important to gain a good overview of the full structure of the system before thinking of how to restructure it. Each of the components in the system was split up in so-called "backends" and "frontends".

One of the obvious places to start looking at was the "dns backend". After a meeting with "Johan Schuijt" in which the structure of all parts of the back-end was discussed and also by checking out call hierarchies in the “dns backend” and “domain backend”, a general overview of the structure of the system could be retrieved and put into a diagram. This structure of the (nowadays old) system can be found in Figure 2.1.

After lots of careful scrutiny and discussion it seemed best to start restructuring from the “dns backend”, the main reason for this being that it was easy to extend this component without breaking other components in the system. To this end, it was important to start working on the more abstract changes that had to be made before modifying any components with dependencies.

If restructuring had started from the “customer frontend”, oversight would be easily lost, which would complicate work significantly. Starting the restructuring at the name servers would also slow down progress, because the slow propagation of zone changes was not caused by the name servers, so these did not have to be changed at all. Taking all these arguments into account, it seemed best to slowly start replacing the cron-jobs in the “dns backend” with push structures.

One thing that should be kept in mind is that TransIP’s system is a unique system that is only used inside of the company itself. Therefore it was impossible for TransIP to buy existing technology that would fix their old system for them. This also led to the fact that all ideas about restructuring the system had to be made up by the project team and supervisors at TransIP, while at the same time researching those made up possibilities.

3.4. Design Alternatives for DNS Backend and Nameservers

Most of the restructuring parts were straightforward and had obvious solutions, but the pull structure between the “dns backend” and the name servers deserves some extra attention, because this was a problem by itself. To restructure the superfluous usage of cron-jobs and pull-like structures between the “dns backend” and the name servers, the following two alternatives were found: POST Requests and Database Replication, which will be discussed in Section 3.4.1 and Section 3.4.2, respectively.

3.4.1. POST Requests

The POST Request alternative meant that all the TransList entries would be replaced with arrays or objects, which in turn would be pushed to the nameserver databases using POST requests. The communication for this would be easy to set up. However, this alternative had some serious drawbacks. The POST requests might cause inconsistencies between the nameserver databases if one or more of the requests fail. POST requests could also partially fail, which goes against the idea of atomicity, meaning that a zone is either completely updated or not updated at all. Consistency and atomicity were one of the most important requirements for our design and this alternative would have the risk of creating serious issues with inconsistencies. In Figure 3.1, a small overview is given of the design of the POST requests. This design would have become part of the complete system, which can be seen in Figure 3.4, but for reasons described in Section 3.4.2, this design was not chosen to function as the communication between the “dns backend” and the nameservers.

[redacted]

Figure 3.1: POST Request Structural Diagram

3.4.2. Database Replication

A second alternative that was found, was to completely replace TransList with a new MySQL database, which will henceforth be called TransDNS Master. All DNS changes made would be put into this database, after which the already existing name server databases (TransDNS Slaves) would replicate their data from TransDNS Master. At first this alternative looked like it would be hard to set up, but after doing research into the replication of databases, it was found that setting this up would not be any harder than using POST requests. It was also found that database replication would work better than POST requests in terms of consistency and atomicity. If replication would stop working on one of the TransDNS Slaves' sides, all that would have to be done would be to restart the database of which the replication is broken. The nameserver will then replicate the latest version of the TransDNS Master and everything would be up-to-date again. In Figure 3.2, an overview is given of the database replication design. This design is part of the complete system, which can be seen in Figure 3.4 .

[redacted]

Figure 3.2: Database Replication Structural Diagram

3.5. Final Design

As could be concluded from the described alternatives above, it was decided to use the “Database Replication” alternative to solve the problem of pushing zone changes from the “dns backend” to the nameservers. In this section, the complete design of the system, including the database replication part, will be expanded upon. The design of adding the TransDNS Master and Slaves will be discussed in Section 3.5.1 and Section 3.5.2, respectively. After that, the atomicity of zone changes and consistency of the databases that will be kept safe with this design will be described in Section 3.5.3. An overview diagram of the complete system design has been created in Figure 3.4. For clarity, a flow diagram was also created which shows how a zone change propagates through the system all the way from the “customer frontend” to the nameservers. This diagram can be seen in Figure 3.3.

3.5.1. TransDNS Master

Since TransList does not exist in the design of the new system, individual zone changes will have to be propagated from the “dns backend” to the nameservers instead of a whole batch of changes at once. As was described in Section 3.3, the first thing designed to change the system was a way to do just that: push individual zone changes from the “dns backend” to the nameservers. To do this, a new MySQL database, TransDNS Master, was set up to replace TransList in the “dns backend”. This database fulfills a core role in the restructuring of the system.

As can be seen in Figure 3.4, TransDNS Master's first use is the fact that all zone changes can be propagated from the “customer frontend” through the “domain backend” into the “dns backend”, to which TransDNS Master is attached. These zone changes will be directly stored into TransDNS Master. To avoid the propagation of incorrect changes, the changes themselves are validated in the “customer frontend”.

TransDNS Master's database design is based on that of the TransDNS Slaves, described in Section 3.5.2. TransDNS Master and the TransDNS Slaves had to have identical columns to make sure replication happens correctly.

[redacted]

Figure 3.3: Flow Diagram

[redacted]

Figure 3.4: Final Design Structural Diagram

3.5.2. TRANSDNS SLAVES

Each of TransIP's nameservers has a database attached to it. In the design of the restructure of the system these databases will be called TransDNS Slave databases. These databases will replicate all data that is put into TransDNS Master without question. The replication happens nearly instantaneously whenever a change is made to TransDNS Master. Because of the way replication works (see Appendix 2), it is made sure that all TransDNS Slaves will contain the same data as Master (and thus as each other) at any moment, which guarantees perfect consistency.

3.5.3. ATOMICITY AND CONSISTENCY

Any zone change that is ever made will have two possibilities when it comes to propagation: the whole changed zone is completely propagated, or the changed zone is not propagated at all. This guarantees that no race conditions will interfere and that no data will collide in any way. To make sure that this happens, all zone changes are done in so-called database transactions. For this purpose, a column was also added to the TransDNS Master and Slaves, which is the '*transaction_id*' column. The transaction ID will keep track of the transaction to which a zone change propagation belongs.

4 Implementation

4.1. Overview

After the design of the system was made, it was followed up by the implementation of the system. Several details had to be kept in mind, such as enforcing atomicity during update propagation, keeping the system modular and realizing the database replication.

In Section 4.2 the overall structure of the system will be discussed with the implementation in mind. The section will also describe the components on a more detailed level. It had been discussed that if time permitted it, secondary features would also be implemented inside the customer frontend. These features would increase the usability of the DNS system for customers and would add to the functionality of the DNS system as a whole. It turned out that time would indeed permit the addition of secondary features. These features will be discussed in Section 4.3.

Finally, the secondary features, such as new customer frontend features that were implemented into the system are described in Section 4.3.

4.2. Overall Structure

The implementation of the modifications in the TransIP DNS Infrastructure is split up into the following tasks:

- Improve the propagation velocity of DNS changes in the TransIP DNS Infrastructure.
  - Set up the new database TransDNS and its controls.
  - Replace cron-jobs from the backends to the name servers to database replication.
  - Replace cron-jobs between Domain and DNS backends by a communication class using SOAP.
- Allow easy management and debugging by creating WorkFlows for vital components.

An in-depth description of the implementation process of TransDNS will be discussed in Section 4.2.1. In order to process the queries and handle the connection of the database, a dedicated database class was created and will be elaborated in Section 4.2.2. The replication is done between TransDNS and the databases on the name servers, which will be described in Section 4.2.3. The cron-job between the Domain and DNS backend is replaced by a new class handling the communication between the backends using SOAP, a communication protocol commonly used in networks. More on this class can be found in Section 4.2.4. The company also uses a customly developed process management system called Workflows, which is elaborated upon in Section 4.2.5.

For clarity, the communication and flow of information between the different backends and the name servers is illustrated in Figure 4.1.

[redacted]

Figure 4.1: Sequence Diagram of the TransIP DNS system

4.2.1. CREATING TRANSDNS

In order to replace the cron-jobs going from the backends to the name servers, database replication is used. This is further elaborated in Section 4.2.3. To enable the database replication, the new database, TransDNS Master, had to be created. To keep it simple and to fit it into the existing system, TransDNS Master has the same structure as the databases on the name servers (the TransDNS Slaves). The tables that are relevant for the push-based system are Domain and Record. Domain was left untouched as it is, operating in the same way as before. However, a subtle change has been implemented in the Record table. Besides the usual columns like the name, row number and expiration time of the record, there was also a time stamp, acting as a serial. In the old system, cron-jobs were running to apply the changes per zone by checking these time stamps. That means that either the whole zone gets updated or not, retaining its atomicity. This structure will not work in a push-based system, so in order to keep the atomicity, the time stamp is replaced with a new column, transaction_id. The idea behind this is that every time there is a change in a DNS entry set, a new transaction is created with its associated transaction_id, which is incremented for every transaction. In case of a name server failure, the name server is out of commission for a period of time with the extra chance of losing all data in the database. If this happens, it is still possible to recover to the current state by tracing back the transactions in TransDNS, either from the first transaction or from the last known transaction. In the last case, it also retains its atomicity if the failure was mid transaction, because the last known transaction will be redone either way. The structure of the Record table is shown in Figure 4.2.

Another small change is the way in which changes are handled in the database. In the previous system, every time a record changed, the associated domain was flagged as changed and added to a list. This was also done for deleted domains with a separate flag. This list was pulled by the name servers after which it was processed. For every domain in the list, the whole entry set was requested in case of a changed record to ensure atomicity or to see whether the domain was flagged for deletion in the database. In the new system, whenever a record change occurs in a domain, a copy of the existing set is created and merged with the new records. The transaction_id of the old record set is changed to the current maximum and is flagged for deletion. Doing so will still maintain the atomicity, because either the whole new set is propagated or nothing is propagated at all. It also speeds up the recovery in case of name server failure, because it can skip all the records that are flagged for deletion.

[redacted]

Figure 4.2: TransDNS Schematics

4.2.2. Creating the Database Class

This class is pretty straightforward, as it handles all the database operations and acts as the communication class between PHP and the database.

4.2.3. Setting up the Database Replication

The existing databases on the name servers are set up as the slaves, replicating the master database TransDNS on the backend side. Setting up the replication is relatively easy, as MySQL databases have this capability built in.

In Section 2.9.3, several replication techniques were discussed with their strong and weak points. Synchronous replication was out of the question, the remaining two techniques were asynchronous and semisynchronous replication. Semisynchronous sounded good on paper and was initially chosen as the replication method. However, because the master database has to wait for a confirmation of one of the slaves, it could in turn cause an unnecessary slowdown, especially in cases of limited connectivity between the name servers and the system. Therefore, asynchronous replication was chosen. In combination with transactions, this would still guarantee consistency and atomicity of the data that is sent between the databases. By using the InnoDB database structure, the use of transactions was possible and thus data consistency could be enforced.

4.2.4. Setting up the Communication between Backends

Not only are there cron-jobs between the name server and the backends, there are also cron-jobs between the backends themselves. The company already had a working SOAP setup used for the communication between said backends. SOAP stands for Simple Object Access Protocol and is used for the exchange of information in computer networks. Thanks to this existing setup, setting up the communication was quickly done by creating a class called TransDns and by enabling it with a simple annotation. This class implements the interface ZoneControllerInterface and responsible for the communication between the DNS and Domain backends.

4.2.5. Workflows

Workflows in TransIP are used for pinpointing where exactly in the system problems occur. Schedulers are launched by the workflows to allow for the detaching of the user and launching separate processes. Workflows are great for monitoring, because the Workflow interface shows the exact state a process is in, visualized as a state machine. It also has the ability to change states on the fly and check its variables, which is useful for debugging. In case of a critical, error, the employees on pager duty will be notified and can instantly see the current status of the error using these workflows. During the implementation of the designed system, it became apparent that workflows had to be incorporated into the system.

[redacted]

The following workflows were created by us:

UpdateZone :*Updates a zone in the TransDNS Master database and verifies if the change has been propagated to the name servers.*

RemoveZone :*Removes a zone from the TransDNS Master database and verifies if the change has been propagated to the name servers.*

DnsMonitor :*Tries for a maximum of three times to update or remove a zone. If it is still not successful, employees on pager duty are notified to inform that there is something going wrong with the zone change propagation of a certain domain.*

CancelDomain :*Completely removes a domain from our name servers, but only after two weeks (cancellation period).*

4.2.6. DNSSEC SIGNING

A side effect was introduced by the new system. In the previous system, changes were first added to a list, only to be processed in batches by cron-jobs. These batches were then signed with DNSSec. Normally, records have an expiration date before they have to be resigned for security reasons, so they are added to a signing queue. However, batches of customer modified zones should get priority over zones that need to be resigned. In the new system, these changes are pushed instantly, so the zones are not signed in batches anymore. Thus, due to this subtle difference, zones changed by customers had to be given a higher priority manually.

4.3. IMPLEMENTED SECONDARY FEATURES

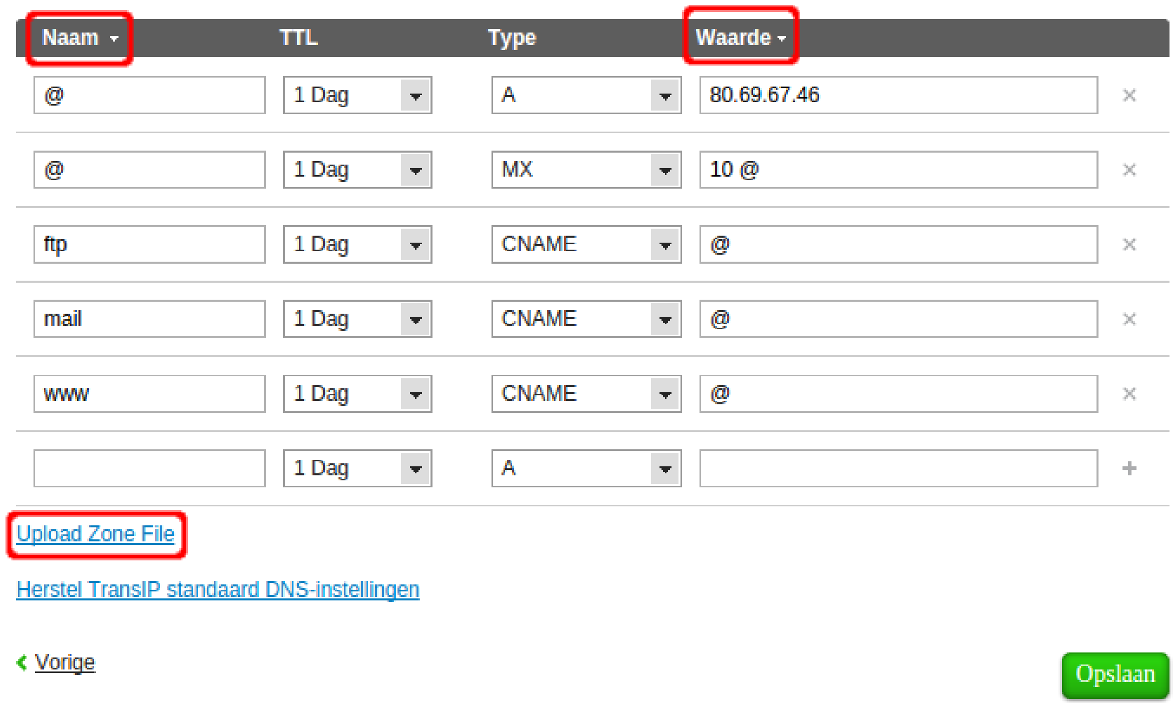
Secondary features are discussed in this section. These features mainly regard enhancement of the customer experience. They were implemented during downtimes and after the main features were implemented. These modifications are performed in the “customer frontend” segment of the TransIP Infrastructure. The following secondary features were requested by TransIP:

- Sort DNS entries by name or by value.
- Allow importing of zone files through the customer control panel.

The request to sort the DNS entries by name or value is a small, but important feature, especially important to customers with domains containing hundreds of records. Designing this feature was relatively simple, but was once again harder to actually realize and implement. The structure of the customer frontend was new and complex, so it took more time to find the right place to incorporate the changes than designing a solution. The additional requirement of this feature was the way the name field had to be sorted: it had to be sorted in a tree-like fashion, prioritizing the Top Level Domain. So ’z.test.com’ comes before ’z.test.nl’ and ’z.test.com’ comes after ’y.test.com’. The order could thus be seen as going from right to left.

Another feature request was to create a feature that gave customers the ability to import a zone file in the control panel, allowing easier management of large domains. Customers now have the ability to upload a zone file, a text file that describes a specific zone and contains the resource records of that zone. This opens up possibilities for customers owning domains with a large amount of resource records, such as the ability to maintain these zone files in a third-party application, instead of in the TransIP web application, or making it easier for them to move to TransIP from other ISPs. During the implementation, security had to be kept in mind, so the files had to be validated prior to processing them on the server. Luckily, a validation library was already written by the company, albeit outdated. The only thing left to do was to patch it up and to tailor it to this feature.

In Figure 4.3 the changes to the control panel are shown.



Naam	TTL	Type	Waarde	
@	1 Dag	A	80.69.67.46	×
@	1 Dag	MX	10 @	×
ftp	1 Dag	CNAME	@	×
mail	1 Dag	CNAME	@	×
www	1 Dag	CNAME	@	×
	1 Dag	A		+

Figure 4.3: Screenshot of Customer Frontend Changes

5 TESTING

5.1. OVERVIEW

From the start of the project, it was known that the final product, the improved DNS structure, would have to be deployed onto the live server. This meant that the system had to be tested extensively, not only for possible bugs, but also for security. Many types of tests were created and run on the system to make sure the system would be as failure-proof as possible. Section 5.2 goes over the performance testing that was done. The different other types of tests that were performed on the rest of the system will be elaborated upon in Section 5.3, along with the results of these tests. Finally, the quality of the final product was assured by different entities. These entities and the way they evaluated the working of the system and its code will be described in Section 5.4.

5.2. PERFORMANCE TESTING

After testing the databases, the performance of the new system was compared to the performance of the old system. It turned out that the performance of the new system became significantly better than that of the old counterpart. Below, graphs are shown of the performance of the old and new system next to each other. These graphs are found in Figures 5.2. The upper graph shows zone change propagations when they happen in serial and the lower graph shows zone change propagations when they happen in parallel.

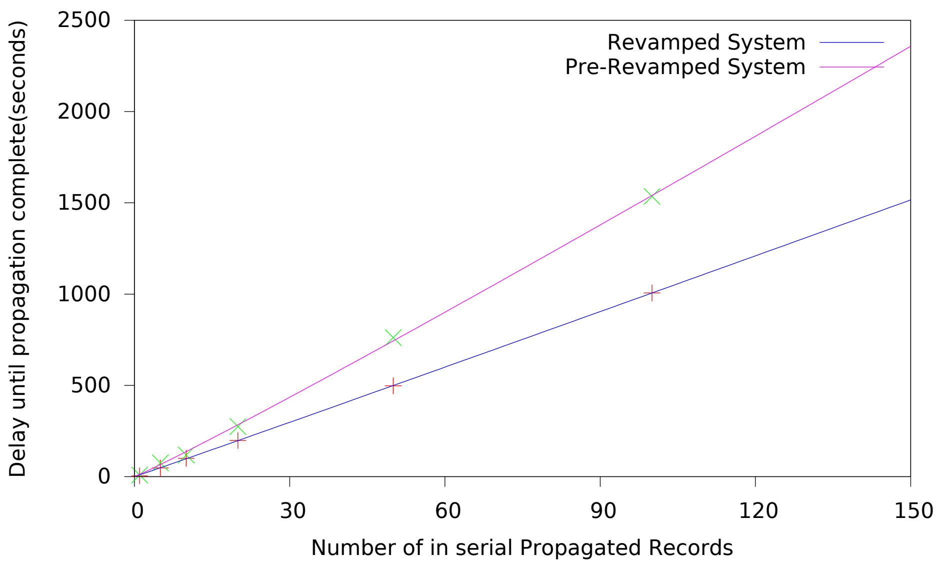


Figure 5.1: Graph of Zone Change Propagation Time for Record Updates in Serial

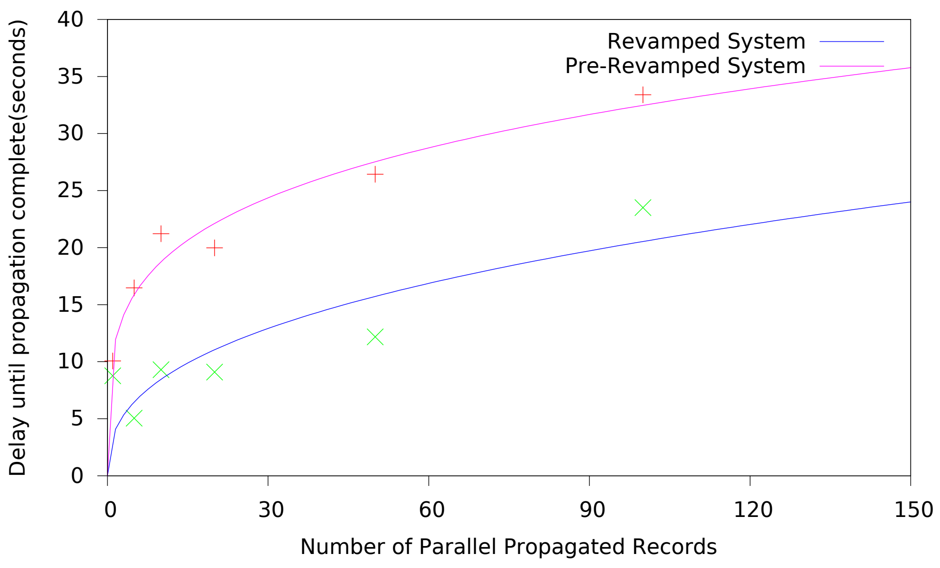


Figure 5.2: Graph of Zone Change Propagation Time for Record Updates in Parallel

5.3. SYSTEM TESTING

The PHP code was mostly tested using the testing framework PHPUnit. Unit tests were created to test the individual units within the classes. To ensure the unit tests covered every branch of the written classes, a coverage report was generated every time the framework ran the tests. As a result, all non-trivial functions written by the group are covered as shown below. The untested classes either contained trivial functions only or were classes unrelated to the project and thus were not taken into account when generating the coverage reports.

[redacted]

[redacted]

[redacted]

5.4. QUALITY ASSURANCE

Several measures were taken to ensure that the code maintains a high degree of quality. The code was periodically peer reviewed internally by employees. After having tested the system, it had to be made sure that the quality of the system was high enough to release it onto the live server of TransIP. To this end, multiple types of reviews have been done, both internal and external. The internal reviews were the reviews that our supervisor, Johan Schuijt, had done for us. His main focus was to assure the quality of the performance and security of the code, which were also two of our main design goals. After looking at those two points, another internal chief coder, Wouter Entius, reviewed our code for maintainability. He checked whether the code was PSR-2 compliant and whether everything was documented. In co-operation with the Delft University of Technology, the code base was sent to an independent code evaluation company, the Software Improvement

Group (SIG). The group would then send back a review of the code base, highlighting its strong and weak points. The code could then be adjusted according to the review and be sent in for a second evaluation to ensure that the review was incorporated correctly. After the first evaluation it turned out that our code was maintainable above average. The reviews (in Dutch) of SIG can be found in Appendix A.

6 Final Product Evaluation

6.1. Overview

In this chapter different kinds of evaluations of the final product will be done. In Section 6.2, an evaluation will be done of the requirements that were set during the research phase. For each requirement it will be decided whether it was met and if it was, in what way exactly. In the section after that, Section 6.3, a more theoretical evaluation will be done regarding the design goals that were also set during the research phase and on which the system's requirements are based. Each design goal will be evaluated in terms of the corresponding requirements. Finally, in Section 6.4, the success criteria, which were set in accordance to the design goals and requirements, will be used to conclude if the project was a success or not.

6.2. Evaluation of Requirements

In this section, all requirements that have been set during the research phase (see Section 2.5) will be evaluated. A table (Table 6.1) will be given, in which each requirement is listed, along with whether the requirement is met and in which way this is reached. In case a requirement is not or not completely met, an explanation for this will be given.

6.3. Evaluation of Design Goals

In this section, the design goals that were set during the research phase (see Section 2.4) will be evaluated. Each design goal will be evaluated separately. The three design goals were (in order of priority): performance, security and maintainability. Each of these design goals also contained a few design sub-goals to which requirements were coupled. Table 6.1 is also used to evaluate the design goals.

1. Performance - Requirements 1, 2 and 3 from Table 6.1 are the requirements that correspond to the design goal 'Performance'. Performance was also the design goal with the highest priority, so it should come as no surprise that all three requirements were met. All data in the different databases is always consistent. Workflows (see Section 4.2.5) are used to monitor all processes and debugging can easily be done by using these. This design goal is evaluated as 'reached'.

2. Security - Requirement 4 from Table 6.1 is the requirement that belongs to the design goal 'Security'. Security was prioritized after Performance and was very important to whether the project would succeed or not. However, it turned out that the parts of the system that would be worked on during this project were already built up very securely. Downtime could still be kept at a non-existent level by securely coding and defensive programming was applied wherever necessary, but no major changes had to be made in order to keep the system as secure as possible. Despite of this, the system is secure and has no expected downtime, so the design goal is evaluated as 'reached'.

3. Maintainability - Requirements 5 through 9 from Table 6.1 are the remaining requirements and they all belong to the design goal 'Maintainability'. Maintainability was the least prioritized design goal, but this did not mean that it was not taken into account. A lot of attention was put into creating a maintainable system. This was very important during this project, because the system would go 'live'

Requirement	**Met?**	**Explanation**
1. The new system must produce the same results as the old system does. In other words: it should be fully backwards compatible.	Yes	The system is able to propagate a zone change from start to end (customer frontend to nameservers) without losing its consistency. The results are thus the same as those of the previous system. See Section **??** for more extensive results.
2. User's resource record updates must be propagated faster than in the old system.	Yes	The system now propagates zone changes faster than it did before the restructure. For more extensive details, see Section 5.2.
3. The system should be as consistent as the previous system.	Yes	Databases contain consistent data (see Section **??**) and all processes are monitored by Workflows (see Section 4.2.5).
4. The system's downtime must stay at a minimum.	Yes	No code was added or changed that would cause downtime of the system. As stated above, Workflows are used to monitor everything. As soon as a workflow goes into an error state, a pager of one of TransIP's employees will start beeping and the fault can and will be solved within minutes.
5. The system should be maintainable for co-employees.	Yes	As stated above, Workflows will make sure that the system is maintainable, because all processes are monitored and Workflows allow for easy debugging (see Section 4.2.5).
6. All code in the system must adhere to the Software Engineering principles as given in the Software Engineering Methods course (part of the Bachelor of Science in Computer Science track of the TU Delft).	Yes	The code was evaluated by the Software Improvement Group twice. All feedback from the first time was taken into account and according to the evaluation, the code is more maintainable than average. See Appendix A.
7. The system must be scalable.	Yes	At any point where it was possible, scalability was kept in mind. The result of this is the ZoneController interface, which will make it possible to implement other DNS systems in the future while still keeping the functionality that was implemented during this project.
8. The system must be tested properly as to minimize unexpected failures.	Yes	The system was tested in numerous ways, about which more can be read in Chapter 5. After having done all these tests, the system has become very robust and unexpected failures are indeed minimized.
9. Deployment must be carefully supervised by TransIP's project supervisor Johan Schuijt.	Yes	Deployment of the system was indeed supervised by our supervisor. The deployment phase is not done yet as of the end of the project. However, the system runs parallel to the old system, but it will slowly be phased out over the coming few weeks, after which the system will be fully deployed.

Table 6.1: Evaluated Requirements and Explanations

and other employees of the company would have to be able to work on the system properly in the future. The evaluation by the Software Improvement Group shows that the system has indeed become highly maintainable and all requirements (5 through 9) were met. This design goal is thus evaluated as 'reached'.

6.4. Evaluation of Success Criteria

At the beginning of the project, a few criteria were set to be able to evaluate the product in its entirety and to define when the product could be called 'done'. These criteria can be found in Section 2.5.2. In this section, a short evaluation will be given on whether the project could indeed be called a success or not.

It was stated that the project, and thus the system, could be called a success if the system was able to go 'live'. To reach this final goal, the system would at least have to meet the requirements that belonged to the design goals 'Performance' and 'Security'. 'Maintainability' would be seen as a design goal of lesser priority. It turned out however, that all three design goals (and the requirements belonging to them) were properly reached. The system was therefore able to go live, which concludes to the fact that the system and project can be called a success.

7 Process Evaluation and Recommendations

In this chapter three topics will be handled in two respective sections. In the first section, Section 7.1, an evaluation will be done in relation to the process of this project. After that, future work that could expand or improve the newly revamped DNS infrastructure and recommendations as to how to make these expansions and improvements will be given in Section 7.2.

7.1. Process Evaluation

Throughout the duration of this project, some parts of the process turned out to work really well or effectively, while other parts did not. In this section a short description will be given of the positive parts of the process, as well as a description of the things that did not turn out too well, along with a way to avoid these things in future projects.
There were quite a few things about this process that worked well or efficiently. The development methodology, Scrum, in combination with TransIP's twist to this (sprints of 2 weeks, milestones per week) worked well for all members of the project group. This made it possible to make concrete deadlines and to know what work had to be done at what moment, while also being able to take time to evaluate or to rework things. If certain pieces of the implementation did not turn out as expected, Scrum allowed for a revision of the plans and to create a new task to fix it. The waterfall development methodology would not allow for this. Trello was used along with Burndown Chart for task management. This gave a good overview and allowed for clear division of tasks and workload. Finally, working as a part of the company (at the office) made it easy to discuss things with the supervisor and other employees and ask for help whenever necessary. This also gave a great view into how work is done inside a company, which was a great experience.

Of course there were also some points at which the process could have gone better. For example, good use was made of Scrum to plan everything as well as possible, but we had underestimated the work that was required to deploy the system. Failures had to be prevented in various ways, such as a complete code review by our TransIP supervisor Johan Schuijt. In a future project, work on the deployment phase could be better calculated into the planning. Despite of everything, the system still went live, but right now it runs parallel to the old system. The old system will slowly be phased out of the system in the coming weeks, after which the new system that we created will run by itself.
Another thing was that sometimes we should have decided faster to go to our supervisor when stuck on a problem, instead of spending too much time on trying to fix said problem on our own. To conclude, the process of the project went predominantly well, but there are some points of improvement for future projects.

7.2. Future Work and Recommendations

Because of the fact that the created system has gone live, the system had to be maintainable and expandable by TransIP co-employees who might start working on the system in the future. Some recommendations will be given to this end and suggestions will be given on where the system could be improved and/or expanded. As stated in Section 7.1, the new system runs parallel to the old one. The old system will slowly be phased out in the upcoming few weeks. The first thing that could give a performance boost to the system would be the optimization of Workflows (discussed in Section 4.2.5). Right now, the Workflows are not fully optimized, so time could be saved after these are indeed optimized.

Another point that could be optimized and that would decrease propagation delay is the removal of the delay with which TransDNS nameservers copy their data to hashmaps. This is a part of the system that is slightly out of the scope of the project, but the TransDNS nameservers lie close to the part of the system that was worked on during this project. The removal of this delay would decrease the propagation delay significantly. A third point that could be improved on is the removal of the use of XSLT in the customer frontend. This is something that was worked with during the implementation of the secondary features (see Section 4.3). This part of the system will be replaced in the near future by TransIP itself, making additions or changes to the customer frontend a lot easier.

The fourth recommendation would be to implement more customer frontend features that improve the experience of users when they are using the DNS system from their control panel. A last recommendation is to create proper documentation in wiki form. The PHP and Javscript code are already documented within the code itself, but online documentation would give a much better overview.

8 Conclusion

At the start of this project, TransIP's wish was to have a restructured DNS system, that would not only become better in terms of performance, but that would also be able to stay ahead of the competition. Therefore, the pull-like system that existed would have to be changed into a push-based system. No existing software could be used to this end, because TransIP's DNS system is custom-made and thus unique in such a way that, outside of TransIP, no software is created for this system.

Before picking up this project, it was clear that this project would become different from regular Bachelorproject. Not only was it very educational in terms of development and project skills, but the fact that the project was done within TransIP itself gave the opportunity to get experience within the business itself, which was a unique experience.

After nine weeks the new system was fully designed, implemented and tested, after which it went into the deployment phase. At the moment of writing, the new and improved system runs in parallel to the old system, which in turn will slowly be phased out. The new system has significantly improved in terms of performance (see Section 5.2) and is also implemented in such a way that it is as failure-proof as could be predicted in advance. All of the design goals have been reached, including the design sub-goals (see Section 2.4) and all requirements have been met (see Section 2.5).

The system will indeed be able to compete with those of competitors. The fact that all design goals and requirements have been met, together with the fact that the system is competent enough to go live and compete with other systems on the market leads to the conclusion that this project can be seen as a complete success.

A
Software Improvement Group Evaluation

A.1. First Submission

[Aanbevelingen]
De code van het systeem scoort bijna 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de opgestuurde code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size en Unit Complexity.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de 'onEnter'-methode binnen 'State_UpdateDns', zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld '// Get current serial from our nameservers.' en '// Construct ZoneInfo for UpdateZone call.' zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Met name binnen de XSLT zou het goed zijn om de templates kleiner de maken. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld complex is. Ook hier geldt dat het opsplitsen van dit soort methodes in kleinere stukken ervoor zorgt dat elk onderdeel makkelijker te begrijpen, makkelijker te testen en daardoor eenvoudiger te onderhouden wordt. In dit geval komen de meest complexe methoden ook naar voren als de langste methoden, waardoor het oplossen van het eerste probleem ook dit probleem zal verhelpen.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

A.2. Second Submission

[Hermeting] In de tweede upload zien we dat de omvang van het totale systeem is gedaald, en dat daarbij de score voor onderhoudbaarheid is gestegen. Deze stijging is met name toe te schrijven aan de verbetering van de Unit Size en Unit Complexity. We kunnen zien dat binnen de PHP de eerder genoemde methoden zijn gerefactored en dat de units over het algemeen kleiner zijn geworden. Ook het verwijderen van de XSLT code zorgt ervoor dat de onderhoudbaarheid is gestegen. Wat wel opvalt is dat er een stijging is van het aantal regels PHP code voor productie, maar dat de hoeveelheid testcode is gedaald.

Uit deze observaties kunnen we concluderen dat een deel van de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject. De verhouding tussen de hoeveelheid test-code en de hoeveelheid code voor productie valt op dit moment nog wel vrij laag uit, het is aan te raden nog eens kritisch te bekijken of alle belangrijke delen van de functionaliteit getest worden.

B
PROJECT DESCRIPTION

TransIP is an Internet Service Provider (ISP) founded in 2003 and has since then grown to be the largest ISP of the Benelux. Besides providing services like web hosting and registering domain names, the company aims to be easily approachable by their customers, maintaining a high standard in customer service and offering technical support whenever necessary. Speed is key in this business, so in order to compete with other Internet Service Providers, they wrote their own Domain Name Server (DNS) system, because they were not satisfied with the systems that were available at the time and they wanted to have more control over their servers. The current system periodically checks for so-called zone changes and propagates them to the corresponding databases. At the time, this method was acceptable as it was relatively easy to implement. However, its updating speed is now far from desirable, as the speed is heavily affected by the multitude of periodic checks: in the worst case scenario, the updates take several minutes before they are stored in the databases of the name servers. TransIP wants to reduce this time to several seconds, which requires a restructure of parts of the system.

C
Confidential

Due to the fact that this project consisted of improving on an existing system, some constraints were set as to what kind of information could be disclosed. An agreement about this was made between TransIP and the developers. After this report has been evaluated by the Thesis committee, some parts will have to be redacted before the report is sent to the TU Delft repository.

Bibliography

Aitchison, R., & Wilson, B. (2005). *Pro DNS and BIND*. Springer.

CVE Details. (2014). *CVE details BIND.* Retrieved July 17, 2014, from http://www.cvedetails.com/product/144/ISC-Bind.html?vendor_id=64

DNS Made Easy / Tiggee LLC. (2014). *Mechanics behind the internet: Authoritative vs. recursive dns servers: What's the difference?* Retrieved July 17, 2014, from http://www.dnsmadeeasy.com/blog/authoritative-vs-recursive-dns-servers-whats-the-difference/

Gavron, E. (1993). *A security problem and proposed correction with widely deployed dns software* (No. 1535). RFC 1535 (Informational). IETF. Retrieved July 17, 2014, from http://www.ietf.org/rfc/rfc1535.txt

Halley, B. (2008). *How DNS cache poisoning works.* Retrieved July 17, 2014, from http://www.networkworld.com/article/2277316/tech-primers/how-dns-cache-poisoning-works.html

Microsoft TechNet. (2014). *Understanding zone types.* Retrieved July 17, 2014, from http://technet.microsoft.com/en-us/library/cc771898.aspx

Mockapetris, P. (1987a, November). *Domain names - concepts and facilities* (No. 1034). RFC 1034 (INTERNET STANDARD). IETF. Retrieved July 17, 2014, from http://www.ietf.org/rfc/rfc1034.txt (Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936)

Mockapetris, P. (1987b, November). *Domain names - implementation and specification* (No. 1035). RFC 1035 (INTERNET STANDARD). IETF. Retrieved July 17, 2014, from http://www.ietf.org/rfc/rfc1035.txt (Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604)

MySQL. (n.d.). *Semisynchronous replication.* Retrieved July 14, 2014, from http://dev.mysql.com/doc/refman/5.5/en/replication-semisync.html

NLNetLabs. (2013). *Name Server Daemon (NSD).* Retrieved July 18, 2014, from http://www.nlnetlabs.nl/projects/nsd/support.html

Rantanen, P. (2010). *Database replication - an overview of replication techniques in common database systems.* Retrieved July 14, 2014, from https://www.theseus.fi/bitstream/handle/10024/20433/Rantanen_Patrik.pdf?sequence=1

Shen, C., & Schulzrinne, H. (2006). Evaluation and comparison of BIND, PDNS and Navitas as ENUM server. *Columbia University Computer Science Technical Report*.

South Asian Network Operators Group. (2009). *Scaling DNS.* Retrieved July 17, 2014, from http://www.sanog.org/resources/sanog14/sanog14-devdas-dns-scalability.pdf

TestingGeek. (2011). *Software testing - performance testing.* Retrieved July 16, 2014, from http://www.testinggeek.com/software-testing-performance-testing-types

TransIP. (2014). *TransDNS.* Retrieved July 9, 2014, from https://www.transip.eu/domain-name/transdns/