Manuel Duarte Ribeiro da Cruz

Degree in Computer Science and Engineering

Understanding and evaluating the Behaviour of DNS resolvers

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Co-advisers: José Legatheux, Professor, NOVA University of Lisbon
João Leitão, Professor, NOVA University of Lisbon
Eduardo Duarte, Technical Director, Association DNS.PT

Examination Committee

Chairperson:
Raporteurs:
Members:

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2018

Understanding and evaluating the Behaviour of DNS resolvers

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

To begin with, I would like to thank my three advisers, Professor José Legatheux, Professor João Leitão and Technical Director Eduardo Duarte for all their support, criticism, insight and for always being available to help whenever I required.

I would like to thank FCT-UNL and DNS.pt for the opportunity to work on this project, as well as all the help that was provided, through resources and tools that were offered to be used.

I would also like to thank my classmates and friends, specifically André Catela, Francisco Cardoso, João Santos, Paulo Aires, Ricardo Fernandes and Tiago Castanho, who supported me through this endeavour of five years, always being by my side and motivating me to keep going. Thanks for proof-reading this document and acting as an extra set of eyes.

Finally, I would like to thank my parents, my brother, and Inês Garcez for always being present, supportive and also proof-reading this document. They were an instrumental part in guiding me through this five year journey, and I can safely say that I wouldn't be here without any of them.

Without all of these people, this thesis would not be possible. From the bottom of my heart, thanks to all of you.

Abstract

The Domain Name System is a core service of the Internet, as every computer relies on it to translate names into IP addresses, which are then utilised to communicate with each other. In order to translate the names into IP addresses, computers resort to a special server, called a resolver. A resolver is a special DNS server that knows the DNS structure and is able to navigate the huge number of DNS servers in order to find the final answer to a query. It is important for a resolver to be able to deliver the final answer as quickly as possible, to have the smallest impact on user experienced latency.

Since there is a very large amount of domains and servers, and the system is highly replicated, there has to be some logic as to how a resolver selects which server to query.

This brings us to the problem we will study in this thesis: how do resolvers select which DNS server to contact? If a resolver always selects the best DNS server - the one that will be able to provide the answer to the query the fastest - then resolvers can more quickly answer their clients, and thus speed up the Internet. However, if they contact different, more or less equivalent, servers they could contribute to load balancing.

To understand how exactly the resolvers select the DNS servers to contact, we conducted an experimental study, where we analysed different resolvers and evaluated how they select the servers. We base the structure and parameters of our study in previous research that has been conducted on the topic, which shows that resolvers tend to use the latency of its queries to the servers as a means of selecting which server to contact.

Keywords: Domain Name System , Resolver , Server Selection

Resumo

O Domain Name System é um serviço chave da Internet, uma vez que todos os computadores o utilizam para traduzir nomes (domínios) em endereços de IP, que são, por sua vez, utilizados para comunicar com outros computadores. Para conseguir traduzir os nomes em endereços de IP, os computadores contactam um servidor especial, chamado resolver. Um resolver é um servidor DNS especial que conhece a estrutura do DNS e é capaz de navegar o elevado número de servidores DNS, com o intuito de obter a resposta final a uma pergunta feita ao servidor. É importante que o resolver consiga devolver a resposta final o mais rapidamente possível, de modo a ter o mínimo impacto na latência sentida pelos utilizadores.

Uma vez que existe um número muito elevado de domínios e de servidores, e como o sistema é fortemente replicado, é necessário que exista alguma lógica que dite o processo que o resolver executa para selecionar o servidor para questionar.

A necessidade desta lógica traz-nos ao problema em concreto: como é que os resolvers selecionam qual o servidor DNS que irão contactar? Se um resolver conseguir sempre escolher o melhor servidor (o servidor que consegue fornecer a resposta à pergunta efetuada mais rapidamente) então o resolver consegue responder ao cliente mais rapidamente, e, portanto, aumentar a velocidade da Internet. No caso em que existam vários servidores com latências iguais, os resolvers poderão também ajudar com o load balancing da rede.

Para compreender como é que os resolvers selecionam o servidor DNS a contactar, conduzimos um estudo experimental, onde analisamos vários resolvers e avaliamos como é que estes selecionam os servidores. Baseamos a estrutura e parâmetros do nosso estudo nos resultados obtidos em estudos prévios, que mostram que os resolvers tendem a utilizar a latência das suas queries como uma medida de seleção de servidores.

Palavras-chave: Domain Name System , Resolver , Seleção de Servidor

Contents

List of Figures

List of Tables

Listings

Glossary

authoritative name server	a server that contains a valid and up to date copy of a zone.
caching only server (also called a resolver)	a special server designed to traverse the DNS tree and cache the responses.
ccTLD	Country code top level domain. It is a domain under the root, associated with a country code (such as .pt, .br, .es), as defined by the United Nation's ISO 3166.
gTLD	Generic top level domain. It is a domain under the root, not associated with a country, such as .com , .org, .info.
name server	A server that associates a name of a resource to properties of said resource, as for example the resource's IP address.
Resource Record	An entry in the DNS database. A RR is associated with a domain name and contains a type, a value of that type and its validity time (TTL).
route flapping	When a router advertises a destination via one route and subsequently another, that course of actions is called route flapping.

TTL	Time To Live represents how long (time or iterations of a protocol) a certain piece of information is considered to be up to date.
zone	A zone is a contiguous subsection of the domain space and includes all the information related to one or more of its subdomains.

Chapter 1

Introduction

The Domain Name System (DNS) is crucial to the performance and ease of use of Internet applications. DNS is essentially used to translate domain names into IP addresses, which are then utilized by the computers to communicate with each other. If DNS did not exist, every single user would have to know the physical IP address of every single machine that they wished to connect to, be it a friend's computer or a website's server. The existence of DNS also solves another problem, which consists in the occasional need to relocate or otherwise change the physical IP address of a server, since it introduces an indirection between names and addresses. With DNS, it is as simple as changing the IP field associated with the server name. Without it, it would be necessary to inform every single user of the new IP address, and every single user would then need to memorize the new IP address.

DNS is, fundamentally, a database, that maps domain names with a number of attributes, such as the IP address. Due to the enormous amounts of records within DNS, and the extremely large number of queries per second, it is mandatory to implement it as a heavily distributed database.

DNS is structured as a hierarchy of servers, divided by the domain names that they contain information on. This further decentralizes the database, diminishing the need for extremely powerful servers, as each server only needs to handle a specific subset of data. Since each server also handles less information, it allows them to operate faster. DNS is also heavily replicated in order to provide high availability, as the odds of all the servers regarding a specific subset of data being unreachable is quite low. The fact that it is heavily replicated also helps in providing good geographical coverage, since there are multiple instances of the same server all over the world.

DNS relies heavily on caching, a technique which consists in saving the results of previous lookups, to avoid successive lookups to the same data. With caching, the amount

of queries that have to be responded by consulting (external) servers diminishes considerably, speeding up the overall process of a DNS query. There are special DNS servers that are used solely to cache responses and resolve DNS queries, called resolvers or caching only servers.

A DNS server often can't fully respond to a query. When this is the case, it generally sends the client the necessary information to reach a server that can actually answer the query. This means that, in order to obtain the desired answer to a DNS query, one must often travel the domain name tree. Resolvers also provide that service, along with caching the responses.

Thus, a resolver is a special type of DNS server, that serves to resolve DNS queries. Resolvers cache the responses they obtain from the servers to avoid repeating queries to the external network. These responses can only be cached for so long before they are no longer usable. How long they can be cached for is a parameter in the DNS database entry.

Resolvers are a vital part of DNS, as without them every single client would need to have a private way to traverse the DNS tree. If everyone had a private resolver, caching would not have nearly as much effect as it currently has, since the number of DNS queries would be much larger. Resolvers are generally managed privately, by the Internet Service Provider(ISP) or openly, such as GoogleDNS and OpenDNS.

Since there is a large number of DNS servers that contain the same data, the resolver must choose one of them to contact. In this thesis, we sought to understand how resolvers choose which server to contact. We attempted to understand how this aspect of a resolver works, since it helps DNS operators to better understand how to strategically place their servers to be most effective. Understanding how resolvers select the server they contact also helps to evaluate the performance of resolvers and, potentially, optimizing them to select an even better server. This study can also contribute to improve the methods of deploying DNS replica servers.

We obtained results that confirm our suspicion, which was that the resolvers rely on the Round Trip Time of a query, since, in general, that is a good metric to estimate how quickly a server can provide an answer to a query. This measure, however, does not take into account the load of servers, and thus could, in specific cases, not provide the answer to the query the fastest. We were also able to understand how some resolvers attempt to balance their queries, when presented with a situation in which there are multiple servers available.

Regarding the structure of this document, it will contain five additional chapters.

In chapter 2, we will present the Domain Name System as a whole, delving into its structure, replication mechanism, query communication and its performance.

The following chapter, chapter 3, analyses prior contributions to the problem addressed in the thesis. We start by analysing how much impact DNS has on the overall performance of Internet communications. Following that, we analyse different ways to select which server to query. To conclude this chapter, we analyse two previous studies that focused on understanding how resolvers work.

Chapter 4 depicts the testing environment that we set up in order to test our chosen resolvers. It begins by providing an overview of our system, after which it delves into the specifics of each component and of some tools. The chapter ends by presenting the tests we have created to evaluate the results.

Chapter 5 shows the results we obtained from executing our tests with our chosen resolvers. We group the tests with their resolvers, providing a brief conclusion at the end of each resolver section. We also analyse the performance from a clients' point of view at the end, while also presenting our general conclusions from the tests.

We finalise with chapter 6, which details the conclusions we have gathered from this thesis. We present some aspects of our testing that were not as satisfactory, and thus should require further testing.

Chapter 2

The Domain Name System

2.1 Overview

The Internet Domain Name System (DNS) is a distributed database that provides a global naming service for web, e-mail, and Internet services in general [8] by associating a domain name, such as www.google.com, with a physical IP address, among other properties.

Without DNS, every host that wanted to access a resource on the network would need to know its physical IP address. Given the enormous scale of the Internet, that is simply not feasible. In order to solve this problem, the concept of a name server was introduced. With these specialized servers, a host is only required to know the physical address of a name server, which the host can then contact to obtain the physical address of the desired service, upon giving a name to search for.

Since DNS is a massive system, the information is distributed over thousands of servers. This means that, in order to obtain a specific piece of information, like the physical IP address of a website, one must navigate this web of servers until a server where the necessary information is stored. This is an iterative process that starts at the root of the system.

Many users often query the same domains, which means that it is ideal to have a server, called a caching only server (also called a resolver) to be in charge of resolving queries, as opposed to each user doing so individually, since, this way, we can cache the results obtained and thus reduce query traffic in the DNS infrastructure, since the server will have the most popular information cached. The resolver receives the client's query, performs the necessary steps to retrieve the information and provides the requested information to the client. The resolver caches the responses it obtains from contacting the DNS servers, since those responses can be useful for future queries, whether from the same or different users.

2.2 DNS Structure as a distributed system

The Domain Name System is built as a hierarchy of name servers, structured by the domain names.

At the top of this hierarchy of name servers sits the root, which is then followed by the Top Level Domains (TLD, which are then subdivided in gTLD and ccTLD), the domain-name, and then any number of lower levels. To delimit this structure, we separate each part with a dot (.), including separating the root from the TLD, making it such that the root contains data about the TLDs. DNS's structure is depicted in figure 2.1.

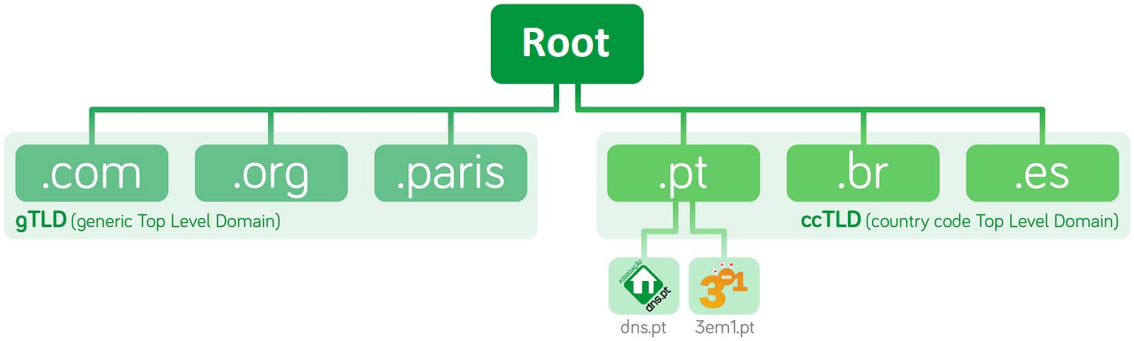

Figure 2.1: The overall structure of DNS, adapted from [2]

There are currently thirteen root-servers world-wide. Each of those servers has several instances, ranging from at least two up to one hundred and eighty eight, so as to offer availability and fault tolerance [16].

2.2.1 Replication

DNS utilizes a Master-Slave replication mechanism for ensuring the divisibility of the information for each domain, where servers replicate masters based on a zone file. Servers that are replicated utilizing this replication mechanism are called authoritative name server over the zone. The slaves can act as masters, providing the records that their master gave them, to another server, which would then be a slave server to the original server in a transitive way.

DNS employs extensive caching, which stores information that has been requested from other servers. However, not all DNS servers perform caching. Instead, the servers that are required to fully answer DNS queries, the resolvers, are the only ones that actually perform caching.

Figure 2.2 shows how a client interacts with a resolver (in the figure it is called by its other name, caching-only server) to retrieve the IP address of the domain "www.wikipedia.org". We see that the client's browser issues a request to its operating system, which in turn contacts a resolver, to obtain the IP address. The resolver then starts by querying the root server, to obtain the IP address of an authoritative server over ".org" (message 2 and its reply, 3). Following that, the resolver then queries the ".org" authoritative server,

requesting the IP address for "wikipedia" (message 4 and its reply, 5). Finally, the resolver obtains the IP address for "www.wikipedia.org" from the server (messages 6 and its reply, 7) and returns it to the client. In this iteration, the resolver's cache did not contain the IP address for "www.wikipedia.org", and thus had to contact external servers. In a case where the resolver's cache contained the IP address for "www.wikipedia.org", we would see only messages 1 and 8, since there would be no need to contact external servers.

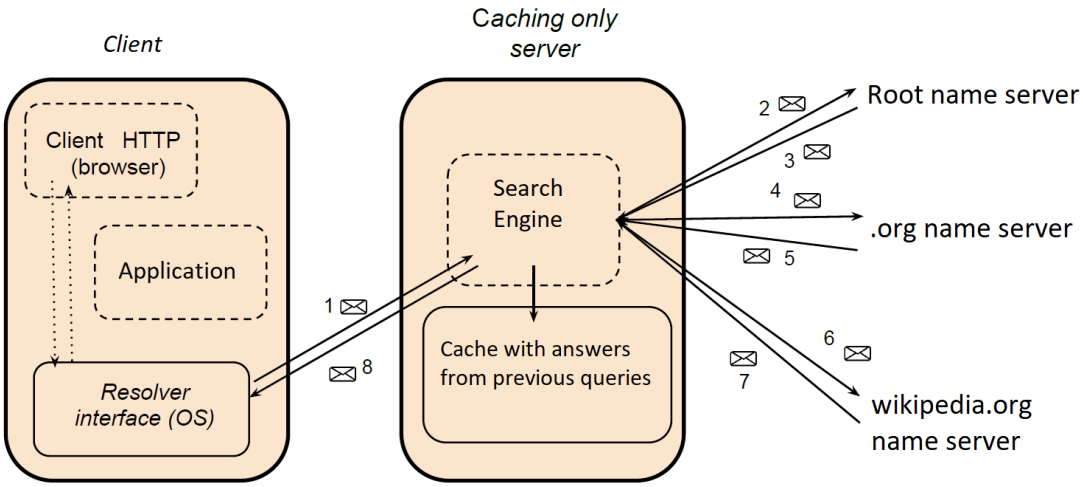

Figure 2.2: A client's browser requesting the IP address of www.wikipedia.org, adapted from [7]

There are a number of private resolvers, which are generally ran by the ISP, as well as public resolvers, such as OpenDNS and GoogleDNS. Both private and public resolvers must cache the answers to previous queries, if they wish to be as efficient and fast as possible, since local answers translates in faster DNS answer retrieval.

Caching in resolvers is heavily influenced by a parameter of each Resource Records (Resource Records), named Time To Live (TTL), that controls how long a specific information can be maintained in the cache of a resolver. A higher TTL value generally provides a better cache hit-rate, since it is not necessary to constantly replace the information. However, if the TTL is too large, the probability of having stale information in the cache is higher. Therefore, a key point of contention for optimization of resolvers is proper management of its cache, by the way of tuning the TTL parameter of RRs.

2.3 Queries

In the context of DNS, a query is a request to a name server to obtain a specific RR, upon giving its name and type. For example, a DNS query to obtain the IP address of "google.com" would be translated to simple terms as "what is the IP of google.com?".

Both DNS queries and responses follow a standard message format, which is outlined below.

Protocol Message Format

The message format contains five sections [9]:

- Header - contains a number of fixed fields, such as the opcode, the ID (which is used to match responses to queries), a one bit field that specifies whether this message is a query(0) or a response(1) and several other flags
- Question - carries the queried name (for example "www.fct.unl.pt") and other query parameters
- Answer - carries RRs which directly answer the query
- Authority - carries RRs which points towards an authoritative name server responsible for the answer
- Additional - carries RRs which relate to the query, but are not strictly answers for the question (e.g. the server may anticipate the next query of the client, and send those RRs)

The opcode is a four bit field which can have values in the range [0,15]. Only the values 0 through 2 are defined, leaving the remainder reserved for future usage.

- 0 corresponds to a standard query
- 1 corresponds to an inverse query (where one provides the IP address and wishes to obtain the domain name)
- 2 corresponds to a server status response

Depending on the Recursion Desired (RD) flag present in the header, a query can be answered in one of two ways. If this flag is not set, the query is treated as a standard or iterative query. If the flag is set, the query is treated as a recursive query.

Iterative Queries

Iterative queries are queries which the DNS server is not required to answer fully, e.g. it is not required to provide a physical IP address that points to the designated host. Instead, the server will reply to the query as best as it can. Typically, it will place the record of a

name server that should know more about the domain being queried in the Additional field of the message.

Iterative queries are very fast, since either a server knows the answer to the query, whether from its cache or from its zone file, and it sends the final answer, or it doesn't, in which case it simply sends a referral answer.

Recursive Queries

Recursive queries on the other hand, have to be fully answered. Due to this requirement, the reply message must contain an Answer section with the requested record, or indicate an error.

If the server's cache contains the answer to the query, it will reply with the data from its cache, assuming the TTL has not expired.

If the server is authoritative over the zone that the requested domain name belongs to, it will respond to this query by getting the necessary data from its zone file.

If none of the above conditions are met, the server will perform a series of iterative queries, with the goal to find a server that is authoritative over the zone that contains the domain present in the recursive query. Once the series of iterative queries has been completed, the server now knows the answer to the original query, and can then reply to the client. Recursive queries can be much slower than iterative queries, since the number of servers that need to be contacted is, generally, much greater.

2.4 Performance

DNS serves a critical role in enabling network traffic. Without it, communication without direct knowledge of the other machine's physical IP address is impossible. Thus, maintaining a high availability and good performance of DNS is extremely important. This is achieved by having several instances of the servers. The speed at which resolvers answer DNS queries is also crucial. Studies have shown that faster responses to DNS queries correlate with increased profits and resolver usage, since faster responses do DNS queries translates in faster overall Internet usage. [18]

In order to maximize the performance, DNS is implemented as a heavily distributed database, with several instances of each server spread around the globe [16] , in order to reduce the number of queries each server has to handle. The fact that the database is also structured hierarchically, with each level of the hierarchy containing only portions of the information, also aids in reducing the amount of data handled by each server instance, which leads to higher performance, but also increases the number of iterations required to find an answer.

Another factor impacting the performance of DNS is its caching behaviour. The more data that is effectively cached, whether on resolvers or name servers, the less network

traffic is generated, and thus the queries have to travel to fewer servers, which in turn translates in a faster response time.

A key aspect of cache optimization is configuration of the TTL of DNS RRs appropriately. If this value is set too low, it can cause too many unnecessary queries, but if it is set too high, it can produce wrong answers more easily.

There is still the question of how to select the appropriate replica of the name server to send the query to. This problem is, in general, tackled by algorithms run by resolvers that collect statistics of the latency of the different authoritative servers. One other solution is to use the anycast protocol on the network level, which allows the same IP address to be announced at several locations, effectively creating a singular IP address for a set of replicas, which would be the IP address the resolver would query.

2.5 Summary

Over this section we outlined how DNS operates and the importance it has. In Section 2.2 we present the structure of DNS as a whole. Section 2.2.1 outlines the replication mechanism of DNS while also introducing the concept of a resolver. Section 2.3 details how DNS clients communicate with DNS, by analyzing the query template and introducing iterative and recursive queries. Finally, in section 2.4, we start reasoning about the performance implications of several DNS aspects, such as its extensive replication, caching mechanism and server selection.

We start the next chapter with an analysis of the impact of DNS in overall latency experienced by network applications. After that, we discuss previous research on the problem at stake, and explore two different ways to solve it: network level anycasting and authority server selection by the resolvers. In this thesis, we will focus on the latter.

Chapter 3

Related Work

The goal of this thesis is to perform a study that contributes to a better understanding of the behaviour of DNS resolvers, particularly how the resolver selects which instance of the name server it will query. Upon gaining a better understanding of how the resolvers select the instances, it will be possible to more strategically deploy name server instances and fine tune resolvers for optimal, or improved, performance.

We start by analysing the study by Singla, Chandrasekara, Godfrey, and Magg [18], a study detailing how the Internet is far from optimal, from a latency standpoint, which collected data regarding the latency of DNS, allowing us to reason about the importance of a high performing DNS.

Next, we analyse the studies by Yu, Wessels, Larson, and Zhang [23] and Müller, Moura, O. Schmidt, and Heidemann [10]. Both these studies attempt to understand how resolvers select the authoritative server that they query. The first study devises an innovative test bed that allows the authors to evaluate how effective a certain resolver is. The second study builds on the first one, replicating their results in a testing environment and later on also applies the same testing methodology to resolvers outside a testing frame.

Finally, we approach the studies by Sandeep, Pappas, and Terzis [17] and O. Schmidt, Heidemann, and Harm Kuipers [11] which we utilise to assess the impact of the anycast protocol in DNS performance, when it is used as a means to selecting the server with the lowest latency.

3.1 DNS Performance

It is important to provide fast DNS queries to reduce overall Internet latency. There are a number of factors that contribute to an increased latency on the Internet, when compared to perfect speed of light latency. These factors include protocols, the physical infrastructure, and routing [18].

A typical user experience consists of inputing a domain name on a browser and connecting to it. The browser then resorts to DNS to obtain the physical IP address of the server, so that it can establish a communication channel. Once the IP address is obtained, communication can be established by utilizing a transport protocol, such as the Transmission Control Protocol (TCP), to send and receive messages.

DNS queries have been shown to add a 5.4x median delay over a perfect speed of light latency [1]. However, the queries are not the sole contributors to higher latency. There are also other factors in play, such as the TCP protocol. It adds a higher latency factor, 8.7x median delay of TCP transfer and 3.2x median delay of TCP handshake, than DNS queries which indicates that the TCP protocol impacts latency more negatively than DNS queries. It should be noted that these values are inflated by the physical and network layers, and, when accounting for this factor, the median delay is shown to be 1.7x DNS , 1.0x TCP transfer and 2.7x TCP handshake [18].

DNS	1.7
TCP transfer	1.0
TCP handshake	2.7
Routing	1.53

Table 3.1: Median delay of specific Internet components without inflation from lower layers

The observations made in this work also reveal that the underlying physical infrastructure contributes significantly to a higher latency. In fact, they conclude that the physical infrastructure is as significant, if not more, than protocol overheads [18]. Since DNS itself represents a significant portion of the delay in the Internet, we conclude that understanding exactly how resolvers select which authoritative servers to query is important, as that is one of the sources of the delay introduced by DNS.

DNS introduces a 1.7x median delay. The source of the delay consists in the selection of the authoritative server to contact, identifying and transmitting the queried RR, and sending it back to the client. Out of all these three contributors, this thesis aims to focus on the first one. We now analyse two different methods of selecting the appropriate server, detailed in section 3.2 which analyses how resolvers select the appropriate server and section 3.3 which analyses how the usage of the anycast protocol on the network layer can also deal with the selection of the appropriate server.

[1] While referring to median delay in this section, it always relates to the median delay over the perfect speed of light latency, which is, theoretically, the fastest possible way to transmit information

3.2 Authoritative server selection by the resolvers

We now discuss the work that has been conducted to investigate how the resolvers select a specific name server instance. As of 2018-02-07, we are aware of two important studies conducted in this area, by Yu et al. [23] and Müller et al. [10].

In Yu et al.'s (2012) paper, the authors attempt to understand how do current (at the time, 2012) resolvers distribute queries among a set of authority servers. They tested six different resolvers: two versions of BIND (9.7.3 and 9.8.0) [1], [13], Unbound [20], DNSCache [3] and WindowsDNS.

Resolvers usually select an authoritative name server by estimating the Smoothed Round Trip Time (SRTT) for each server, utilizing statistics from past queries. If the server has not answered any previous queries or they have timed out, they set the SRTT value to either the query timeout value or an arbitrarily large value.

Once resolvers have all the SRTT values, they employ one of two selection mechanisms: Least SRTT or Statistical Selection.

3.2.1 Least SRTT

The Least SRTT method presents a challenge due to the way it works since it, like the name suggests, selects the server with the smallest SRTT value. This technique can be suboptimal in cases where a server with a previously high SRTT value (if a server was unreachable but it is now reachable, for example) should now have a smaller SRTT and be chosen. However, since the SRTT value relies on previous queries, and the previously observed value was large, it will not be queried without some other mechanism in place to deal with the SRTT variation.

In order to counteract this effect, Least SRTT implements a decaying SRTT mechanism, whereby the SRTT of unselected servers decreases by a factor β, where $\beta < 1$. How this factor is computed is specific to the resolver. For example, BIND utilizes a constant β value, while PowerDNS utilises an exponential value.

If the β value is constant, it implies that this factor is dependant on query rate as each successive query impacts the decaying factor equally, while if the β value is exponential, it does not possess this dependency. Taking the examples from Yu et al. [23], we have two different cases: in the first case, there are two consecutive queries, t_1 and t_2, while on the second case there are three consecutive queries, t_1, $t^{'}$ and t_2. This leads the first case to have a β value of

$$e^{\frac{t_1 - t_2}{C}}, \tag{3.1}$$

where C is a constant, and the second case shows a β value of

$$e^{\frac{t_1 - t^{'}}{C}} . e^{\frac{t^{'} - t_2}{C}} = e^{\frac{t_1 - t_2}{C}}. \tag{3.2}$$

Thus, we observe that both β values are equal, and are only determined by the constant C and are not dependant on the query rate. By applying the SRTT decaying factor, the

resolvers ensure that, eventually, every server will be queried, and thus, assuming that the most optimal servers maintain the smallest SRTT, the most optimal servers should be selected over a large majority of queries.

3.2.2 Statistical Selection

The Statistical Selection method selects the server based on the SRTT, but instead of selecting the server with the smallest SRTT value, it selects a server based on a probability. The probability of a server being selected is related with the server's SRTT value: a higher SRTT value translates in a lower probability of being selected, while a lower SRTT value translates in a higher probability of being selected. Due to this inclusion of a statistical probability, ever single server will eventually be chosen. Since the likelihood of selecting a server that exhibits a lower SRTT is higher than one with a larger SRTT, this method will tend to select good servers over a majority of queries.

PowerDNS To properly evaluate the selected resolvers, Yu et al. [23] chose three different scenarios:

- **Scenario 1** - RTT of the authority servers range linearly (starting at 50ms, going up to 170ms, in intervals of 10ms)
- **Scenario 2** - RTT of the authority servers range linearly (starting at 50ms, going up to 170ms, in intervals of 10ms), apart from one unresponsive server
- **Scenario 3** - RTT of the authority servers range linearly (starting at 50ms, going up to 170ms, in intervals of 10ms), apart from one unresponsive server, which recovers after five minutes

The authors set up a testing scenario in an isolated environment. They deployed thirteen authoritative servers that serve the tested ".com" domain. They purposefully set the TTL of all DNS records to a large value, as well as the size of the resolver's cache, so as to eliminate differences in results that stem from caching effects. Since the goal was to test server selection, this is a good approach. They also setup a network emulator, to simulate packet delay (adjust the different servers' RTT) and packet loss (to make a certain server unresponsive). As for input, a portion of a resolver log from a large U.S. ISP was utilized. It contained approximately 3.5 million lookups for 408,808 unique domain names. This lead to an average query rate by the resolvers of approximately 250 queries per second.

3.2.3 Measurements

Yu et al. [23] found four types of sub-optimal server selection behaviour. Two of those behaviours were observed in Scenario 1, one in Scenario 2 and the final one in Scenario 3.

In Scenario 1, the authors found three resolvers (DNSCacche, Unbound, and WindowsDNS) that distributed queries evenly among all the authoritative servers. DNSCache

doesn't estimate the RTT of the servers , while Unbound only uses this estimate to rule out under-qualified servers (it randomly selects a server that hast under 400ms SRTT).

Also in Scenario 1, BIND 9.8 was found to send more queries to server with a higher RTT, which is the opposite of the desired behaviour. This was found to be due to the high query rate, since BIND 9.8 was also tested with a slower query rate, which saw some improvement. The experimental results are congruent with the theory, since, as BIND 9.8 utilizes the Least SRTT with a constant decaying factor, once a high RTT server is selected, it will always be selected until a query is responded or times out. In contrast, PowerDNS, which also uses the Least SRTT method, but with an exponential decaying factor displays a large majority of queries being sent to the server with the least latency. It does however suffer from the same problem that the constant decaying factor suffers, which is that, when a high latency server is selected, it will stay as the selected server until a query is responded or times out. This is not as impactful, since the exponential decaying factor means that SRTT decays much slower than when coupled with a constant decaying factor.

Still in Scenario 1, BIND 9.7, which uses the Statistical Selection method coupled with a decaying factor also shows sub optimal server selection. While a larger percentage of queries are indeed performed to the least latent server, it is not a majority, since every other server receives at least 7% of the queries.

In Scenario 2, BIND 9.8 and DNSCache still sent a significant number of queries to the unresponsive server. DNSCache does not keep statistics of previous queries, hence it did not know that a server was unresponsive. BIND 9.8 assumes that a timed out query was responded with a very large RTT. This leads a large number of queries to be sent to an unresponsive server, when one is inevitably selected due do the decaying factor. This problem is also somewhat present in PowerDNS, but since it uses an exponential decaying factor, it takes substantially more time for an unresponsive server to be selected, which means a lower percentage of queries will be sent to the unresponsive server.

Finally, in Scenario 3, some resolvers detected that a server had become responsive slowly. Unbound and PowerDNS took the largest amount of time (15 and 3 minutes, respectively). In the case of Unbound, this is due to the fact that it probes unresponsive servers periodically, once every 15 minutes, and thus, in the worst case scenario, it takes 15 minutes to detect that a server has become responsive once again. PowerDNS on the other hand, since it relies on Least SRTT coupled with exponential decaying, takes a significant amount of time to select the newly responsive server. However, once it is selected, its SRTT value will be updated and it will be selected faster.

Müller et al.'s (2017) paper further develops Yu et al.'s (2012) work and also attempts to update the results. The authors deployed 7 authoritative servers over 7 different datacenters. These servers are then queried by 9,700 vantage points (VPs), scattered throughout the globe as seen in figure 3.1, which are provided by RIPE Atlas [15]. These VPs will query for a DNS TXT resource record, and uses the locally configured resolver. To determine which authoritative server the VP reaches, each server was configured with

a different response for the same DNS TXT RR. Since DNS responses are extensively cached, the authors only query their domain, so as to control the TTL, which they set at 5 seconds. They also ran separate measurements, with an interval of at least four hours between them, which provide the resolvers more than enough time to flush the response from their caches.

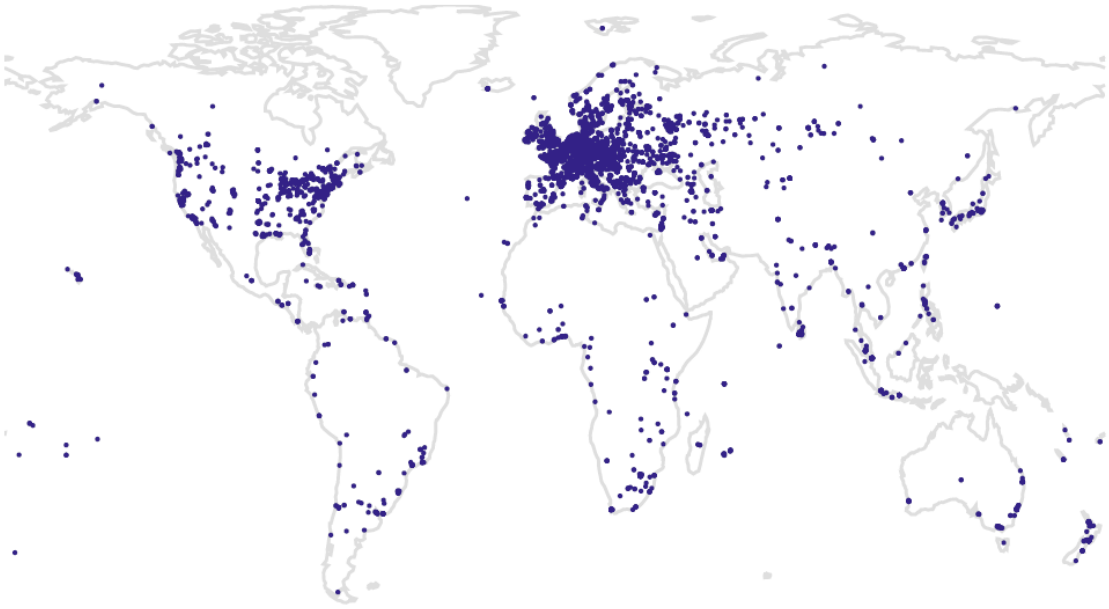

Figure 3.1: Locations of more than 7,900 vantage points from RIPE Atlas, taken from [11]

The deployment of the authoritative servers over several datacenters provides a good geographical coverage, with authoritatives ranging from a close proximity (one authoritative has an instance in Dublin and another in Frankfurt, for example) to large proximity (one authoritative has an instance in São Paulo and another in Tokyo, for example).

Müller et al. [10] consider some measurement challenges:

- The probes might be configured via DHCP to utilize multiple resolver, so they consider a combination of the probe's ID and the resolver's IP address to be a single VP

- Load balancers between VP and resolvers using anycast may send the queries to different instances. This effect cannot be eliminated, but they compared the client and authoritative data to find that it only presents minor effects on the collected data

Müller et al. [10] corroborated Yu et al.'s (2012) results:

- Most resolvers query all instances of an authoritative server in both studies

- In the first study, 3 out of 6 resolvers were classified as being heavily based on RTT, while in the second study most resolvers were based on RTT, as all authoritative servers showed a preference for instances with lower RTT

After performing this measurement setup and corroborating Yu et al.'s (2012) findings, Müller et al. [10] sought to validate their results by comparing them to real-life deployments of the root zone and the ccTLD ".nl".

In the root zone measurements, they found that a significant portion (20%) of resolvers sent queries to only one root letter. A majority of resolvers (60%) queried a majority of root letters (6), but only a very small minority of resolvers actually queried all 10 root letters. These results are explained by the lack of cache control. Since most resolvers have prior queries to root letters, they are more likely to have the necessary RR in cache, making a query to a root letter unnecessary.

As for the ".nl" measurements, a majority of resolvers were found to query all authoritatives, confirming the results from previous studies [23] and their study as well.

In conclusion, both studies [23][10] showed that resolvers are heavily based on RTT, but they still query all server instances. This is a necessary mechanism to attempt to always query the best server, as a better server can suddenly become available. A notable exception is the root zone, where a significant portion of resolvers were shown to only query one authoritative server.

3.3 Anycast

The anycast protocol allows data from a single source client to reach one of several destination nodes. The protocol consists in assigning a specific IP address to multiple servers. The servers then announce that they possess that IP address, and, since multiple servers announce the same address, the routers interpret it as different ways to reach the same server. In reality, the different paths terminate in different servers, that are now all reachable on the same IP address. These paths are not necessarily available to everyone, as the anycast addresses can be announced locally (within the host's routing network) or globally. In the case of DNS, this is then utilized to distribute the queries without needing any application level logic, as the routing is handled by the routers themselves [5].

We will now analyse two studies, Sandeep et al. [17] and O. Schmidt et al. [11]. The first study attempts to measure the performance impact of anycast in DNS, while the second study attempts to determine how many anycast locations are required to provide optimal coverage, making it necessary to also study the performance of anycast. Sandeep et al.'s (2005) study is relatively dated and therefore some of their conclusions on specific aspects of the performance of anycast as a server selection protocol may not apply currently.

Sandeep et al. [17] attempt to measure how much of an impact anycast has on DNS's performance. In order to measure this, they utilize four different zones to represent four

different scenarios:

- A zone multiple servers in a single geographic location, without employing anycast. The B root server is utilized in for this scenario. This scenario is used as a base case, to compare with the anycasted scenarios, and understand how large is the impact of anycast in DNS's performance
- A zone using a single anycast address for all its servers, with multiple servers in multiple different locations. Both F and K root servers are utilized for this scenario. They used two different servers to investigate the effects of the number and location of anycast group members on performance.
- A zone using multiple anycast addresses for its server, with multiple servers in different locations. They used UltraDNS (which is authoritative for the .org and .info TLDs), which provides all their servers with two different anycast addresses (TLD1 and TLD2). Every server is thus accessible via two distinct anycast addresses which should, in theory, increase their resilience to outages[2].
- A zone with multiple servers, distributed geographically, all reachable via their unicast address. To emulate this scenario, the Sandeep et al. [17] configured their clients to send requests to the F root server, through each of its instances unicast address.

The authors compared these four scenarios based on three distinct criteria, of which we highlight two: **Query Latency** which measures the delay of queries and **Availability** which measures outage periods (e.g. a server that experiences no outage periods has optimal availability). They utilized approximately 400 different vantage points, scattered throughout the globe (although a majority of the vantage points were located in North America, as can be seen in table 3.2), provided by PlanetLab [12].

Table 3.2: Distribution of PlanetLab nodes around the world, taken from [17]

Continent	**% of PL Nodes**
South America	0.5
Australia	1.8
Asia	15.8
Europe	16.7
North America	65.2

Sandeep et al.'s (2005) results on the **Query Latency** criteria show that the F root server displayed the lowest latency (75ms mean). This was explained based on the amount of instances that the F root server encompasses, which was, at the time, the highest amongst all the tested servers. Since it had more instances, queries needed to travel a

[2] In this context, an outage is a window of time during which a node is unsuccessful in retrieving a record from the anycast server servicing it

shorter distance to reach a server. Despite this, the latency experienced even in the F root server, which was the lowest amongst the tested anycast deployments, was still higher than the unicast latency, where the F root server showed a mean latency of 45ms, a little bit over half the latency observed in the anycast deployment of the same F root server. The difference in these two measurements was attributed to the fact that only two of the F root servers are global and many clients don't have visibility to the local servers that are closer to them.

Regarding the **Availability** criteria, anycast performed remarkably well, as the average percentage of unanswered queries was below 0,9%. Sandeep et al. [17] found that the UltraDNS anycast addresses TLD1 and TLD2 experienced more failed queries than both F and K root servers, which they tentatively attribute to the fact that all UltraDNS clusters are global, which results in clients following more different paths to reach their servers, resulting in a heightened effect of route flapping [21]. Both TLD1 and TLD2 exhibit overall shorter (two to three times) outages when compared to the other zones. However, the combined (TLD1+TLD2) zone experiences a large number of outages, since [17] treat the act of a client switching from TLD1 to TLD2 (and vice-versa) as an outage. The combined zone also revealed short outages, which goes according to the expectations that more than one anycast address provides added resilience to network outages on DNS clients. This data revealed that anycast's recovery time after an outage is governed by the recovery time of the network routing. Thus, the different anycast setups cannot reduce the average time to recover from an outage. They can, however, reduce the severity of the outage (so that it affects less clients) by employing more than one anycast address, as evidenced in the combined (TLD1+TLD2) zone.

Moving on to O. Schmidt et al.'s (2017) study, which consisted in measuring how effective anycast was in selecting the instance of a site with the lowest latency, from different vantage points (7900) scattered throughout the globe, provided by RIPE Atlas [15]. We analysed it to attempt to understand how effective is the use of the anycast protocol in the selection of DNS servers.

All of their testing was performed over four root servers - C, F, K, L. These servers were chosen since they represented the scenarios they wanted to study: a server replicated in only a few sites (C root server, replicated in 8 sites), two servers replicated in a moderate amount of sites(F and K root servers, replicated in 58 and 33 sites, respectively) and in large amount of sites (L root server, replicated in 144 sites). The sites of each server are not all equally available. Both C and L root servers have all their sites available globally, which means anyone can connect to them. However, F and K have significantly more local sites (53/58 and 14/33, respectively) and, by extension, significantly less global sites. This distribution impacts the effectiveness of anycast in selecting the DNS server with the lowest latency, as, due to routing policies, a server may be selected that is not the most optimal. Since during the course of the development of their paper, the K server changed its routing protocol, placing all but one site available globally, the tests were repeated for this new version of the K server.

In order to properly measure the performance of the anycast protocol, the authors devised an interesting strategy: from every vantage point, they measure both the latency to every anycast site, and the latency through the anycast protocol. To do this, they resort to the ping command. After having gathered the closest server latency-wise for every vantage point, they compared that latency with the one obtained from pinging the anycast IP address. They then compute the hit rate, which is how often the anycast protocol resolves the anycasted IP address to the lowest latency server. We believe that we can take advantage of this hit rate measurement and apply it to our goal, utilizing it to identify how accurately a resolver selects the server with the lowest latency.

Sandeep et al. [17] study shows promising results regarding anycast performance on DNS. While it didn't always select the most optimal server, as evidenced by the difference in latency on the unicast and anycast F root server zones, it still chooses a server with relatively low latency. It also revealed that with more than one anycast address, it is more resilient to network outages. However, Sandeep et al. [17] is somewhat dated, and thus we cannot confidently stand behind the results obtained. Despite the study's age, we believe that the overall conclusions withdrawn from it are still sound, as they tend to agree with our analysis of O. Schmidt et al. [11].

O. Schmidt et al. [11] study shows that anycast does not always select the authoritative server with the lowest latency, but it still chooses a server with relatively low latency, thus corroborating the findings from Sandeep et al. [17]. This less than optimal performance can be due to a number of factors such as load-balancing needs or routing policies, which causes clients to prefer nodes in the same network as opposed to the closest node, latency wise.

3.4 Summary

In this chapter we analysed some research that is relevant to understanding the behaviour DNS resolvers. We started by linking the desire for lower latency on the Internet with the subject of this thesis, by showing that DNS introduces a non trivial delay in communications in Section 3.1.

Next, we performed a more thorough analysis of two studies that attempt to determine how resolvers select the authoritative servers in Section 3.2. Both studies present similar results in their internal test frames, showing that some resolvers do indeed select the authoritative server with the lowest latency. However, most resolvers also contact almost every authoritative servers available, as a way to ensure that they can keep choosing the best one in a dynamic setting. The external, live testing performed in the second study also seems to corroborate the results presented in both studies, so we conclude that they are a good starting point to this thesis.

Finally, we sought to understand how effective is the anycast protocol (Section 3.3). From the research conducted, we understand that anycast does not always select the server with the lowest latency. However, it still produces quite good results, since it generally selects a server with a relatively low latency. It also provides additional resilience to network outages, which is further increased when servers are configured to have more than one anycast address.

Both anycast and resolver authority server selection seem to yield very good results, selecting the authoritative server with the lowest latency in a majority of cases. However, since resolvers must check to see if the authoritative server selected is still alive, anycast seems to edge out this approach, as it does not need to periodically check with the other authoritative servers, to see if they are still reachable. However, in case of server failures, anycast was shown to not be as effective in handling them as some of the resolver's implementations.

Chapter 4

Test Bed

In this chapter, we will describe how we set up our test bed and how the components interact with each other. In addition, we also outline the configuration parameters of the individual components, while also describing some tools we used to analyse our data.

4.1 Overview

The testing method consists in the following steps: from a client machine, we issue a series of queries to our chosen domain (www.net.) which we have previously set up. The network traces are then captured using another machine to sniff out the network packets, generating packet tracing (.pcap) files with the contents of the network packets of each test. After obtaining these files, we analyse them resorting to DNSJit, a tool which allows us to parse a .pcap file, loading all the attributes of a DNS message (such as the source IP address, the destination IP address, all the flags, the query itself) into an easily manipulated data structure. Next, we analyse the data structures, retrieving statistics such as the percentage of queries that were responded by each authoritative server and the latencies experienced. Finally, we re-use the latency of the queries to plot charts that associate a query with its latency.

In order to evaluate the resolvers (we chose PowerDNS [13], Unbound [20], Bind [1], Windows-Server 2012 [22]) we deployed a four server wide DNS hierarchy. That is, we have one server which contains an excerpt of the root zone, as found in *IANA* [6]. We edited this zone to only have one Resource Record (RR) and thus a single TLD (net.). Having our fake root zone with a single TLD simplifies our querying process, as we only need to query a domain within this TLD. This is a reasonable simplification to enforce in our tests, since we are only interested in understanding the process through which the resolver selects an authoritative server to contact and thus, we only require one testable

domain.

Having the fake root zone set up, we next set up three authoritative servers for the net. TLD. These authoritative servers all share the same zone file, with a slight difference between them: The zone file itself contains one single entry, for the domain www.net. (for the sake of simplicity), which is associated with a single IP address, which is related to the authoritative server to which the zone belongs to. This leads us to have an easy way to debug and actively monitor the test, as the dig results will contain the IP address obtained, which will vary depending on which authoritative name server was contacted.

4.2 Necessary Machines

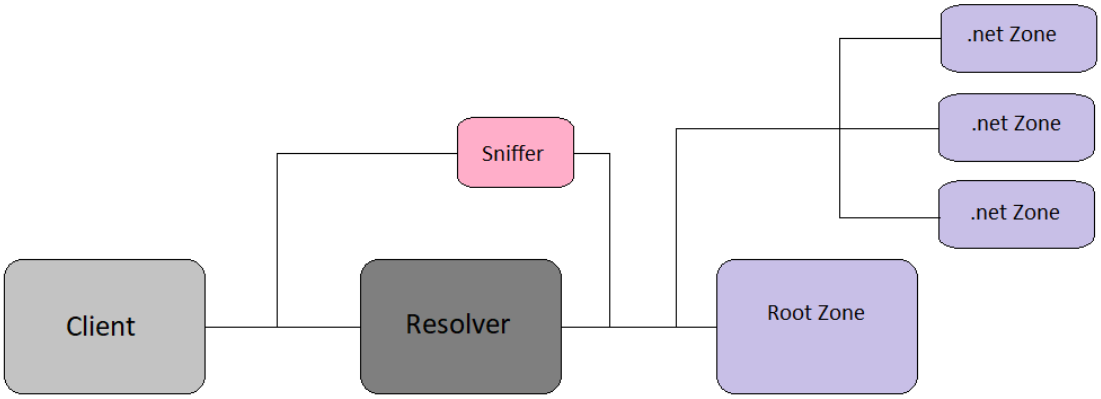

Figure 4.1: Test bed

Our test bed itself consists in seven separate machines, depicted in figure 4.1, which are configured and managed through the usage of virtualisation software. All the machines should have a connection to each other: i.e. any machine should be able to ping any other machine, so they can all communicate with each other. Every machine should have some way of being remotely accessed by the controller of the tests. This means that every machine will require SSH or, in the case of the Windows machine, Windows Remote.

4.2.1 Client

The client machine is running on Linux, and is in charge of overseeing the tests, in addition to retrieving the .pcap files generated by the sniffer machine and analysing them. It is required for SSH and *DNSJit* [4] to be installed in this machine, as we need to access the remaining machines via SSH to control them and transfer files to the client machine. Given that we also need to communicate with Windows machines, we opted to also install python, along with its package pywinrm. Pywinrm (Python Windows Remote) allows us to remotely connect to a Windows machine without requiring us to set up SSH access on the Windows machine (although this can be done), and is then used to access the

Windows resolver. DNSJit is required as it is the tool we have selected to analyse the .pcap files.

In our setup, we were generating the queries outside of the client machine, and then transferring them to the client. If, however, we wanted to keep everything inside the client machine, and thus more automated, we would also need to install Java on this machine, as our query generator is written in Java.

The client machine should also have the IP addresses of the resolvers as well as the authoritative name servers in its hosts file, as the test generator utilises the names to sort the tests, while it retrieving the IP addresses from the hosts file.

4.2.2 Sniffer

The sniffer machine is also running on Linux, and is in charge of collecting the network traffic to then be processed by the client machine. Given that the purpose of this machine is to collect the network traffic , we require a tool that does so and places it in the .pcap format. To this end, we opted to use the widely known tcpdump command, and is thus required in this machine.

4.2.3 Authoritative Servers

There are a total of four authoritative servers and they are all running on a Linux operating system. One server is authoritative over the root zone, while the other three are authoritative over the .net zone, as shown in figure 4.1. All these servers are set to be authoritative only via BIND options, and thus they require BIND to be installed. It should be noted that, despite BIND being installed, they do not act as resolvers. The server responsible for the root zone does not require any additional software, other than BIND. The remaining three servers, however, require a tool that can simulate delays in the network, in order to introduce latency in our tests. For that, we use *tc-netem* [19] to add a qdisc (a qdisc, or queueing discipline, is a traffic control scheduler [14] manageable through the tc linux command) to each server, which we configure to have the necessary parameters for each test.

Moving on to the configuration options of the servers, which are shared between all four of them, except the definition of their zone files. These options are detailed in figure 4.2. Here, the important option, which turns the servers into authoritative only servers, is the option in line 6, signifying that we do not wish for this server to be recursive. All servers must then load their zone files, as shown in figure 4.3. Here, the name of the zone as well as its file path will obviously vary, depending on which zone should be added to which server.

The root zone file is depicted in figure 4.4. It is a stripped down version of the root zone file located at *IANA* [6], to which we added our own .net TLD. We maintained the same SOA record parameters as the ones present in the original file. As can be seen on

```
acl trusted {any;};

options {
    directory "/var/cache/bind";

    recursion no;
    allow-query {trusted;};
    listen-on {localhost;};
    allow-transfer {none;};
    listen-on port 8050 {any;};
    listen-on port 53 {any;};

    dnssec-validation auto;

    auth-nxdomain no;    # conform to RFC1035
    listen-on-v6 { any; };
    querylog yes;
};
```

Figure 4.2: Configuration file of authoritative servers of the root and .net domains

```
zone "." {
type master;
    file "/etc/bind/zones/root.zone";
    allow-transfer{none;};
};
```

Figure 4.3: Configuration file where we load the root zone

lines 6, 9 and 12, this TLD has three name servers, ns1, ns2 and ns3, each associated with their A glue records.

```
.          86400 IN    SOA a.root-servers.net. nstld.verisign-grs.com. 2018061800 1800 900 604800 86400

.          172800  IN  NS  a.root-servers.net.
a.root-servers.net. 172800  IN  A   192.168.5.140

net.            172800  IN  NS  ns1.net.
ns1.net.        172800  IN  A   192.168.5.141

net.            172800  IN  NS  ns2.net.
ns2.net.        172800  IN  A   192.168.5.142

net.            172800  IN  NS  ns3.net.
ns3.net.        172800  IN  A   192.168.5.143
```

Figure 4.4: Root zone file

The authoritative name server zone file is depicted in figure 4.5. It should be noted that this is the zone file for ns1, with the zone files for ns2 and ns3 being slightly different: after SOA, they would have ns2 and ns3, respectively, and the first record would be ns2.net. and ns3.net., respectively. As can be seen, it is quite similar to the root zone file, with the main change being the SOA record, which states that the server with this zone

file is authoritative for the .net zone.

```
net. 86400   IN SOA ns1.net. admin.net. 2018061800 1800 900 604800 86400

; Name servers

    2   IN  NS  ns1.net.

ns1.net.        2   IN  A   192.168.5.141
ns2.net.        2   IN  A   192.168.5.142
ns3.net.        2   IN  A   192.168.5.143



; A records for name servers
www.net.        2   IN  A   192.168.5.141
www.net.        2   IN  A   192.168.5.142
www.net.        2   IN  A   192.168.5.143

```

Figure 4.5: .Net zone file

4.2.4 Resolvers

Finally, we have the resolvers. Out of our four selected resolvers, three of them (PowerDNS, Unbound and Bind) are hosted on machines running a Linux operating system, while the remaining resolver (Windows-Server 2012) is hosted on machines running its respective Windows version, as the resolver itself comes bundled with the whole operating system package.

In order for the resolvers to be able to resolve our domain, we must first point them towards our own root server. This is done by replacing the standard hints file, which points to the original root servers, to ours (as seen in figure 4.6), which replaces the IP address of one root server with the IP address of our own root server. In the Windows resolver, we also change its hints file, to only contain one entry, shown in figure 4.7. As for the remaining configuration options, we stick to the default options. This means that our resolvers may not necessarily be as optimized as they can be, as there may be options which increase their performance.

```
.                       3600000  IN  NS    A.ROOT-SERVERS.NET.
A.ROOT-SERVERS.NET.     3600000      A     192.168.5.140
;
```

Figure 4.6: .Net zone file

4.3 Individual Tools

In this section, we will talk about two tools that we have used, specifically, *DNSJit* [4] and our tool that generates our queries.

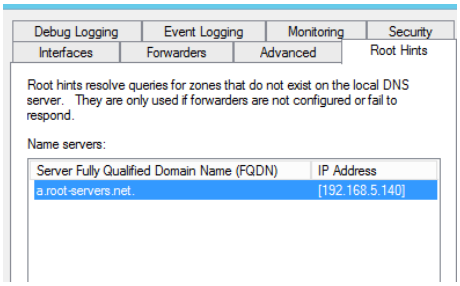

Figure 4.7: Windows root hints

4.3.1 DNSJit

DNSJit is an engine for capturing, parsing and replaying DNS messages. This tool is a combination of parts of other tools (dsc, dnscap, drool) built with a Lua wrapper, which creates a script-based engine. For our purposes, the core functionality of DNSJit consists in its ability to parse and process DNS messages, which we use to process the DNS messages and feed them to a data-structure that we created. Once we have the data-structure with all the information we need, we then process it, extracting statistics about the dataset.

Since DNSJit is a script-based engine, we run the files by executing the command ”dnsjit <our_file> <parameters>”. We have generated two DNSJit scripts, slightly based on the examples available on the tool’s website: one script analyses the .pcap file that we feed to the script, producing statistics, while the other script produces a graph that associates each query with the latency it experienced, in order to facilitate visualisation of the data.

These two scripts require eight parameters, separated by whitespace:

1. <.pcap file> - This parameter provides the .pcap file that will be analysed.
2. <tested_domain> - The domain to which the queries were pointed at. In our tests, this was always ”www.net.”.
3. <test_runtime> - How long the test was ran for.
4. <query_interval> - The interval between each successive query.
5. <filename> - The name of the file where the script will write its output
6. <resolver_used> - The resolver that was used for this test.

The second to last parameter varies between the two scripts. On the statistics script, this parameter is <ttl_used> - the value of the TTL that was used for this test. On the graphing script, this parameter is <resolver_ip_addr> - the address of the IP address used.

The last parameter is the list of servers that were used in this test. This list contains the IP address of the server, the latency that the server was set to during the test, the

packet loss that the server was set to during the test and the shutdown interval, which is how long of an interval there should be between periods where the server is available. This parameter should be set between quotation marks, and multiples of this parameter can bet provided to the script, with each of them detailing the specifications of one server.

The final command to run our scripts looks like this: dnsjit <scriptname> <.pcap file> <tested_domain> <test_runtime> <query_interval> <filename> <resolver_used> <ttl_used OR resolver_used> <"ip_address latency packetloss shutdown_interval">

4.3.2 Query Generator

In order to generate our queries, we wrote a very simple Java-based query generator. This generator outputs a script, which is then ran on the client machine. This script will handle all the set up necessary for every test, the test itself and the collecting of data and its analysis. In order for the generator to support all these functionalities, it requires a few parameters:

1. <tcpdump_opts> - A quotes enclosed list of options for the tcpdump command, all separated by whitespace
2. <test_name> - The name of this test. We named the tests sequentially i.e. Test 1, Test 2, Test 3
3. <resolver> - The IP address resolver to be used for this test.
4. <domain> - The domain to be queried during this test.
5. <resolver_name> - The name of the resolver to be used for this test.
6. <query_filename> - The filename of the output of the query generator.
7. <results_filename> - The filename of the results of the test.
8. <ttl_used> - The TTL used for this test.
9. <next_script> - The name of the next script to be used, in order to chain several tests one after the other.
10. <query_internval> - The interval between each successive query in this test. If set to -1, it will generate a random value between 0 and 1 seconds.
11. <test_runtime> - The amount of time this test should run for

4.4 Tests

In order to evaluate our resolvers, we designed twelve tests. These tests were created to evaluate different aspects of the resolver, like its behaviour in a stable environment, how much does a name server's latency affects its choice and how long does it take to respond to changes in the network topology. Each tests lasts for one hour and queries the same domain, "www.net.", which has a TTL of 2 seconds (slightly less than the interval among queries to avoid resolver caching). Therefore, there should not be any replies from the resolver's cache and the amount of queries analysed should be at a constant value of 1200 throughout testing, barring any form of pre-fetching, through which a resolver would fetch the requested resource record, by predicting future queries based on past behaviour, and thus responding faster to the client..Table 4.1 depicts the tests we created, along with the parameters of the specific tests.

	Server 1		Server 2		Server 3	
Test Number	**Latency**	**Packet loss**	**Latency**	**Packet loss**	**Latency**	**Packet loss**
1	10ms	0%	100ms	0%	190ms	0%
2	40ms	0%	100ms	0%	160ms	0%
3	70ms	0%	100ms	0%	130ms	0%
4	100ms	0%	100ms	0%	100ms	0%
5	10ms	0%	100ms	0%	500ms	0%
6	40ms	0%	100ms	0%	500ms	0%
7	70ms	0%	100ms	0%	500ms	0%
8	100ms	0%	100ms	0%	500ms	0%
9	10ms	10%	100ms	0%	190ms	0%
10	10ms	30%	100ms	0%	190ms	0%
11	10ms	Variable	100ms	0%	190ms	0%
12	10ms	0%	100ms	0%	190ms	0%

Table 4.1: Tests designed to evaluate resolvers

There are five distinct categories of tests in our setup:

- Standard Tests (1 through 4)
- Latency Cut-off Tests (5 through 8)
- Packet Loss Tests (9 and 10)
- Network Topology Change Tests (11)
- Random Query Rate Tests (12)

4.4.1 Standard Tests

This group is designed to understand how does the resolver behave in a stable environment, while evaluating how it distributes the queries to the available authoritative name servers. Tests 1, 2 and 3 have a server that is clearly better than the remaining, as it exhibits a lower latency value. Therefore, we will classify resolvers as good if they consistently select the server with the lowest latency, and not optimal, from this point of view, otherwise. Based on previous work [23], we expect our resolvers to stabilize its query distribution over the course of this test group, with Test 1 showing a more prevalent query percentage towards Server 1. This prevalence should then decrease, as its latency increases, while in Test 4 queries should be evenly distributed.

4.4.2 Latency Cut-off Tests

This group is similar to the Standard Tests group, with the caveat of Server 3 always exhibiting a latency value of 500ms. Our objective with this value is to understand if there is a limit to the maximum latency of a server that the resolver is able to choose as a viable one, when other closer servers are available . We chose the value of 500ms as that was reported [23] to be the latency cut-off value of Unbound.

4.4.3 Packet Loss Tests

The third group of tests contains Tests 9 and 10. This group's tests are designed to understand the impact of a "shaky" connection on the choice of a resolver. To that end, we applied a certain degree of packet loss to Server 1, and we monitor how frequently this server is queried.

4.4.4 Network Topology Change Tests

The fourth category of tests is made up of a single test, Test 11. This test is designed to understand how quickly a resolver reacts to a change in the network topology. To that end, Server 1 starts the test with its packet loss value set to 0%. Ten minutes after beginning the test, this value is changed to 100%, which simulates the server being unreachable. After another ten minutes, the packet loss value returns to 0%. This pattern arises periodically up to the end of the test. A resolver will be classified as good, in this test, if, during it, it quickly notices the change in the network topology, and doesn't attempt to keep contacting an unreachable server. In addition, our classification for this test also includes how quickly the resolver starts sending queries to Server 1 after it has become reachable again. Since we perform one query every three seconds, these periods show up on the graphs as intervals every 400 queries. This means that from queries 1-400 Server 1 was reachable, then during queries 401-800 Server 1 was unreachable, repeating until we reach the 1200 queries performed in one test.

4.4.5 Random Query Rate Tests

Finally, the fifth category of tests, which also only contains one test, Test 12. This test exhibits a variable query delay of a random value between 0 and 1 seconds, while the other 11 tests exhibit a query delay of three seconds. This means that, in this test, the query rate will always be higher than one query per second, while the other 11 tests have a query rate of one query every three seconds. This test was designed to understand if there is any impact in having a query rate higher than the resource record's TTL, as well providing a less synthetic context to the previous tests, which had a fixed query rate. It also allows us to understand whether or not a resolver employs pre-fetching. This query rate is set in the client's side, which means that several queries will be answered by the cache of the resolver. Thus, we expect resolvers to show a minimum of 1800 queries (given our TTL of 2 seconds and our test runtime of 3600 seconds). Resolvers that do employ pre-fetching should show a higher amount of queries. With a variable query rate, this last test is also slightly more representative of human behaviour.

Chapter 5

Results

In this chapter, we will discuss the results we obtained during our study of the behaviour of the different resolvers.

It should be noted that the configurations used in each resolver may not necessarily be the most optimal. In fact, it is quite likely that is the case: the only changes we have made were to its root hints file, which we switched to our own version, in order to direct it to our root server. This means that all the resolvers have default configurations.

The tests and their categories are the ones that have been detailed in Table 4.1 and we present the results of each category separately (for each resolver) also in table format.

In addition to that, a graph was created for each test. The graphs that are not shown in this section can be found in the annex. Every graphic contains three servers, listed by their IP addresses in the graphics legend, on the upper right-hand side. Server 1 corresponds to the IP address 192.168.5.141, which is the colour red. Server 2 corresponds to the IP address 192.168.5.142, which is the colour green. Server 3 corresponds to the IP address 192.168.5.143, which is the colour blue. These graphs relate a single query (shown on the X axis) with the latency it experienced (shown on the Y axis) with a dot, coloured according to which server replied to that query.

We start by analysing the Bind resolver, then PowerDNS, followed by Unbound and finally, Windows12. After these resolvers are analysed, we present our conclusions, in addition to a final analysis of the data as a whole, for each resolver, from a client point of view.

5.1 BIND

5.1.1 Standard Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 1	11.07ms	96.5%	104.99ms	2.1%	202.42ms	1.4%
Test 2	41.15ms	90.9%	101.24ms	5.3%	161.11ms	3.8%
Test 3	71.15ms	76.9%	101.08ms	15.3%	131.06ms	7.8%
Test 4	101.15ms	33.9%	101.32ms	33.2%	101.20ms	32.9%

Table 5.1: Results of the Standard Tests category of the Bind resolver

Table 5.1 shows the results obtained in the first category of tests.

The first test showed a heavy preference to the Server 1, which received a dominant percentage of queries (96.5%). Despite this, the two other servers were also queried, albeit not as often as Server 1, which indicates that Bind is contacting every server, in an attempt to select the best one. That attempt is thus successful, as Server 1 is the server queried most often. It should be noted, however, that this first test had a slight anomaly in the network: the first query to Server 2 and the first query to Server 3 both exhibited abnormally high latency (201ms and 382ms, respectively, which seems to be roughly twice the expected latency). Despite this fact, we believe that the test's data still holds, as it was a single query that experienced this unexpected latency and all the remaining queries appear to be within the expected range of latency.

The second test continues the trend of the first, exhibiting a preference for servers with lower latency. We can also see that the percentage of queries received by Server 1 has diminished to 90.9% as its latency has increased, while the other two Servers are receiving a higher percentage of queries. This indicates that the resolver is contacting servers that are not the best more often, as it realises that the difference between the best server and the others is not as large.

The third test continues on the pattern of the two previous tests, exhibiting an even lower query percentage towards Server 1.

Finally, on the fourth test, we see that the query percentage is roughly the same for every server, which is explained by the fact that the latency to each server is also equal, thus there is no "best" server available, and all are chosen equally.

Having concluded the first type of tests, we classify Bind as having a good performance in these tests, as it actively selected the best server in each of them. In addition, it also constantly makes sure it has selected the best server, by sporadically contacting the remaining servers.

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 5	11.25ms	96.8%	101.20ms	2.2%	501.05ms	1%
Test 6	41.33ms	92.1%	101.64ms	6.2%	504.95ms	1.7%
Test 7	71.02ms	81.6%	101.03ms	16.3%	501.60ms	2.1%
Test 8	101.84ms	48.5%	101.95ms	49.2%	501.68ms	2.3%

Table 5.2: Results of the Latency Cut-off Tests category from the Bind resolver

5.1.2 Latency Cut-off Tests

The first test of this category confirms that Bind does not have a cut-off point at or below 500ms, as Server 3 is still selected despite it being quite rare as only 1% of the queries are received by the server. Server 1 and 2, as expected, behave similarly to the first category of tests, which is expected as these two servers share the same latency and packet loss values of those tests. Throughout these tests, it is interesting to note that both Server 1 and Server 2 show slightly higher query percentages when compared to the respective tests of the first category, while Server 3 shows lower query percentages. This further supports our assessment that Bind actively searches for the best server, as when a server's latency increases, its query percentage decreases, while when its latency decreases, its query percentage increases.

Our conclusions from this test are that Bind does not have a cut-off point at or below 500ms and that it consciously selects the best server available and Bind distributes queries among servers, selecting one server with a probability that is the inverse of the previous latency measured, or an equivalent pattern.

5.1.3 Packet Loss Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 9	11.00ms	17.1%	100.96ms	74.8%	191.01ms	8.1%
Test 10	10.95ms	4.8%	101.45ms	86.5%	191.66ms	8.7%

Table 5.3: Results of the Packet Loss Tests category from the Bind resolver

The ninth test (performed with 10% packet loss) shows an average latency to Server 1 (which experiences packet loss) of 11.00ms. It should be noted that we only include successful queries in our latency calculations, therefore queries whose packets were lost do not affect our calculation of the latency. It is evident, however, that BIND treats the failed query differently, since Server 1 only has a query percentage of 17.1%, much less than that shown in the previous tests, without packet loss. Bind seems to be treating this server as having a higher latency, given its decreased query percentage, as a server with a higher latency would be contacted less frequently. That is not the case, since when looking at the graph for this specific test 5.1, it shows that there are periods where Server 1 is contacted regularly, and then periods where it is not contacted at all. This clustered pattern is indicative that Bind is contacting Server 1 until it drops a packet, after which it stops contacting it for a certain amount of time until it tries again.

The tenth test, with an even higher degree of packet loss, supports our findings from Test 9, as Server 1 receives a smaller percentage of queries than the remaining servers, while still exhibiting the lowest average latency according to our calculations. This test experienced an anomaly, as can be seen in Figure 5.2. This anomaly was due to a disconnect of the network card of the machine that was hosting all the virtualised testing machines. It was not necessary to repeat the test since we can see in Figure 5.2 that the testing environment recovered quickly after the anomaly, and the majority of the test was unaffected.

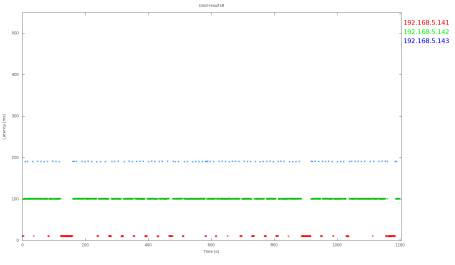

Figure 5.1: BIND Test 9

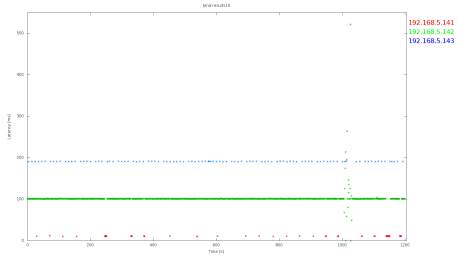

Figure 5.2: BIND Test 10

5.1.4 Network Topology Change Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 11	11.01ms	47.1%	100.98ms	47.4%	191.01ms	5.5%

Table 5.4: Results of the Network Topology Change Tests category from the Bind resolver

Looking only at the query percentages, Bind queries Server 1 47.1% of the time. Given the nature of this test, and that Server 1 is only reachable 50% of the time, this query percentage distribution seems encouraging for the Bind resolver. In fact, upon analysing the graph generated for this test, we believe that Bind performed quite well. After the first downtime period, it took 10 queries until Server 1 was contacted again, which is equivalent to approximately 30 seconds. After the second downtime period, it took 9 queries until Server 1 was contacted again, which is equivalent to approximately 27 seconds. In addition, we note that Server 3 is queried much more frequently when Server 1 is unreachable. This is due to the fact that BIND is actively searching for a better server to select.

Figure s 5.3 and 5.4 represent the same test. However, Figure 5.4 also includes the failed queries in an effort to understand how frequently the resolver queries the unavailable server. We see five queries over a ten minute period that are directed to the unreachable server. We can then conclude that, in the worst case scenario, it would take BIND 2 minutes to contact a previously unreachable server, under our test conditions, as that was the interval at which the unresponsive server was queried.

It would be interesting to research if Bind differentiates a packet loss from an unreachable ping (ICMP).

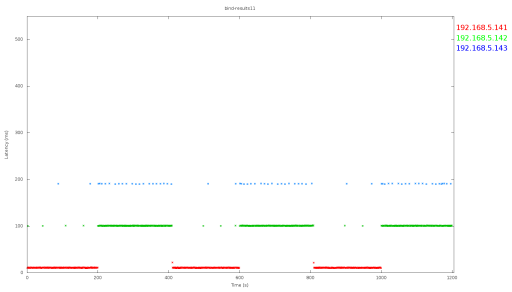

Figure 5.3: Bind Test 11

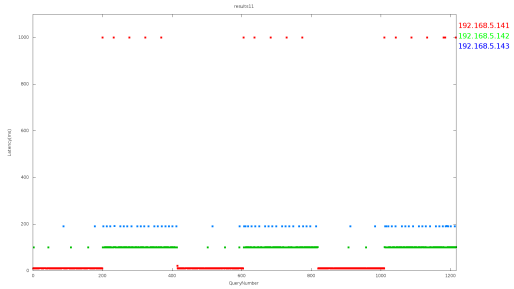

Figure 5.4: Bind Test 11, with failed queries shown

5.1.5 Random Query Rate Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 12	11.04ms	96.7%	100.89ms	1.8%	191.06ms	1.5%

Table 5.5: Results of the Random Query Rate Tests category from the Bind resolver

In regards to the query percentage distribution, this test produces the expected results, with Server 1 showing to be the clear favourite of the resolver for this test. We did, however, notice that the total amount of queries was not the amount expected. Given our TTL of two seconds, we expected at least 1800 queries, since the test ran for 3600 seconds. However, we see that we have just a little bit over 1200 queries, at 1226. We sought to understand the reasoning behind this, and it turns out that BIND was sending the client responses with its TTL set to 0, which may pose a red flag.

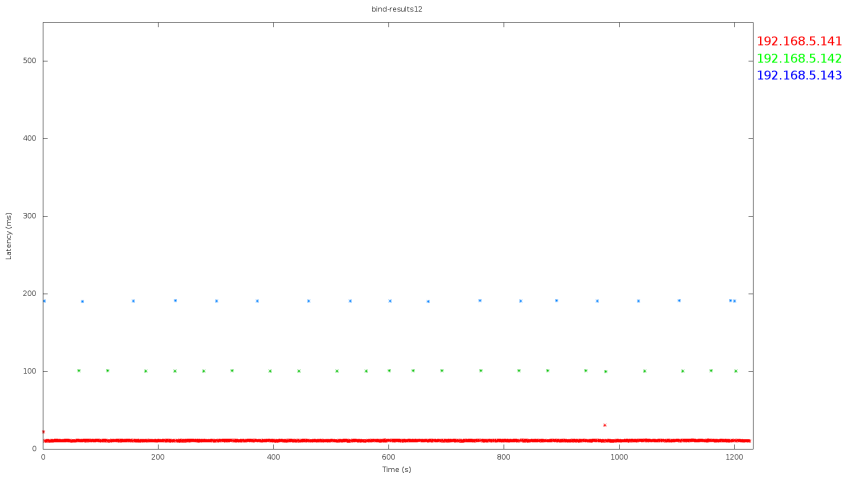

Figure 5.5: BIND Test 12

Our final classification of the Bind resolver is a favourable one. It demonstrated that it chooses the best server in the standard test scenario, while still validating that it was indeed choosing the best server. It was able to deal with packet loss, by quickly querying another server. It was also fairly quick to adapt to changes in the network topology. The biggest drawback that this resolver showed was the abnormal query number on the last test, which indicated that it was responding to the client with expired resource records. However, these resource records were very recently expired, in the order of the second.

5.2 PowerDNS

5.2.1 Standard Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 1	11.03ms	92.2%	100.99ms	4.3%	191.12ms	3.5%
Test 2	41.08ms	83.7%	101.10ms	9.7%	161.11ms	6.6%
Test 3	71.02ms	67.4%	101.10ms	19.8%	131.00ms	12.8%
Test 4	101.03ms	33.3%	101.02ms	33.4%	101.03ms	33.3%

Table 5.6: Results of the Standard Tests category of the PowerDNS resolver

The first test reveals that PowerDNS starts off well, choosing the least latent server, Server 1, for 92.2% of the queries. Despite this clear preference, it also contacts the other two available servers, although it contacts Server 2 more frequently than Server 3 (4.3% vs 3.5%). Throughout the next tests, we see the query percentage move towards the 33.3% mark, which is reached by Test 4. During all the tests where the query percentage had not yet reached this value, Server 1 was chosen a majority of the time, always exhibiting a higher query percentage than the remaining two. Of the remaining two servers, Server 2 was always chosen more frequently than Server 3. This indicates that, like the previous resolver, Bind, PowerDNS actively seeks to confirm that it is indeed choosing the best server available and, from our observations, does so successfully. This leads us to classify, in regards to this test, PowerDNS as a good resolver.

5.2.2 Latency Cut-off Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 5	11.03ms	92.9%	101.02ms	4.4%	501.00ms	2.7%
Test 6	41.01ms	86.4%	101.01ms	9.8%	501.07ms	3.8%
Test 7	71.12ms	74.6%	100.86ms	20.4%	501.17ms	5.0%
Test 8	101.03ms	47.2%	101.02ms	47.2%	501.02ms	5.6%

Table 5.7: Results of the Latency Cut-off Tests category from the PowerDNS resolver

The fifth test proves that PowerDNS does not have a cut-off point at or below 500ms, since Server 3, which was set to have a latency value of 500ms, was still contacted. This test group follows the same pattern exhibited in the first test group of standard tests, with the exception of the query percentage of Server 3, which is due to the fact that Server 3 had a different latency. The query percentages again show that they tend to converge towards a value, in this case being 47.2% for both Server 1 and Server 2. Server 3 finishes this group with the remaining 5.6% of queries. This is further evidence that PowerDNS is

actively evaluating if the server it has selected is the best, as the query percentages shift according to the latency experienced towards each server.

5.2.3 Packet Loss Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 9	11.13ms	24.4%	101.06ms	64.8%	191.16ms	10.8%
Test 10	16.85ms	8.7%	102.51ms	81.3%	194.12ms	10.0%

Table 5.8: Results of the Packet Loss Tests category from the PowerDNS resolver

Test number nine was performed with a packet loss value of 10% set on Server 1. Despite this value, Server 1 still saw a large portion of queries, which may indicate that PowerDNS is good at handling network failures. In fact, Server 1 still had a query distribution percentage of 24.4%, which is still lower than Server 2's 64.8% but higher than Server 3's 10.8%. This indicates that, on a network with 10% packet loss towards a normal 10ms latency server, it is still better than a no packet loss 190ms latency server, but worse than a 100ms latency server without packet loss.

The tenth test is performed with 30% packet loss. In this test, we finally see Server 1 being chosen the least amount of times, sitting at 8.7% of all queries, whereas Server 3 now has 10% of all queries, with the remaining 81.3% queries being answered by Server 2. Interestingly, PowerDNS Server 3's query percentage falls off 0.8%, indicating that PowerDNS is now preferring Server 2 more strongly than in the previous test case.

In both these tests, we saw the same burst pattern (shown in Figure s 5.6 and 5.7) as was revealed in the corresponding Bind tests, which indicates that these two resolvers likely use a similar method of authoritative name server selection.

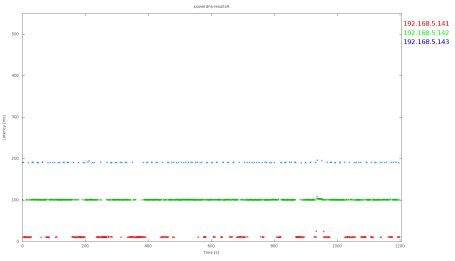

Figure 5.6: PowerDNS Test 9

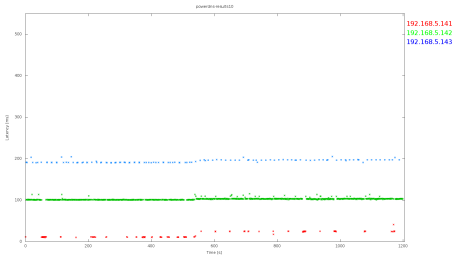

Figure 5.7: PowerDNS Test 10

5.2.4 Network Topology Change Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 11	24.82ms	43.5%	103.60ms	50.5%	197.13ms	6.0%

Table 5.9: Results of the Network Topology Change Tests category from the PowerDNS resolver

As with Bind, we can see that in this test, Server 1 still has a large percentage of the queries directed to it, despite not a majority. When we analyse the graph Figure 5.8 of this test, we can add context to this data. The graphs show a burst pattern, coinciding with the periods where Server 1 is reachable. From the graphs, we can conclude that in this test, PowerDNS adapted relatively quickly. It took approximately 60 seconds to contact Server 1 after it was brought back up at the 400 query mark, and again 60 seconds once the server was brought back up again at the 800 query mark, as shown in Figure 5.8. Figure 5.9 shows the failed queries that PowerDNS sent to Server 1 while it was unreachable. From this graph, we can see that it sent a query to Server 1 approximately once every 85 seconds. Given this value, we believe that the largest amount of time that Server 1 would not be contacted while it was available would be 85 seconds, since that is the frequency at which PowerDNS contacted it to check if it was back up.

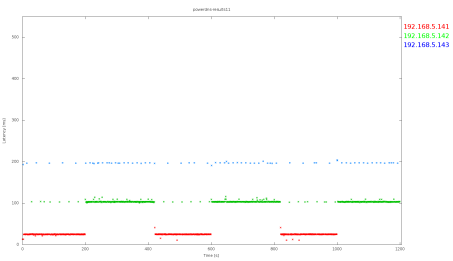

Figure 5.8: PowerDNS Test 11

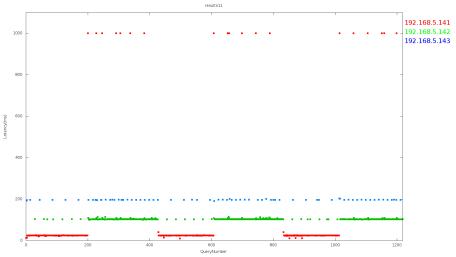

Figure 5.9: PowerDNS Test 11, with failed queries shown

5.2.5 Random Query Rate and Interval Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 12	24.60ms	95.3%	103.24ms	2.8%	196.76ms	1.9%

Table 5.10: Results of the Random Query Rate Tests category from the PowerDNS resolver

This final test adheres to the patterns observer in Test 1, with Server 1 showing being heavily prefered by PowerDNS.

The main conclusion of this test, however, is that PowerDNS must employ some sort of pre-fetching mechanism with our default configurations. We believe this to be the case because, as we can see in Figure 5.10, there are over 3000 queries performed. Given our TTL of 2 seconds, without any pre-fetching mechanism, we would expect 1800 queries. Since we had over a thousand more queries, we can confidently say that PowerDNS has a pre-fetching mechanism.

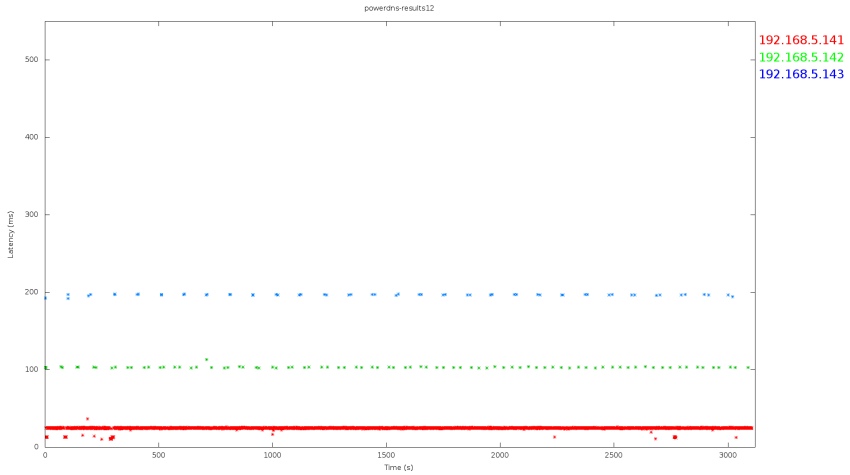

Figure 5.10: PowerDNS Test 12

With these tests, we classify PowerDNS as a very good resolver. We saw that it heavily preferred the best server in the first two groups of tests, and it doesn't seem to have a latency cut-off point below 500ms. In order to deal with packet loss, it simply chose another server for a certain amount of queries, always returning to the server with packet loss eventually, but never sending too many queries to it. It reacted very quickly to changes in the network topology, with a worst case scenario reaction time of 85 seconds, which is quite good. We also discovered that it employed some method of pre-fetching.

5.3 Unbound

5.3.1 Standard Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 1	11.23ms	44.5%	101.24ms	47.2%	191.05ms	8.7%
Test 2	41.13ms	35.4%	101.11ms	32.0%	161.10ms	32.6%
Test 3	71.52ms	34.9%	101.32ms	29.8%	131.30ms	35.3%
Test 4	101.87ms	30.8%	101.69ms	35.9%	101.65ms	33.3%

Table 5.11: Results of the Standard Tests category of the Unbound resolver

The first test reveals that Unbound does not choose the server with the lowest latency a majority of the time. In fact, Server 1 only has a query percentage of 44.5%, which is lower than that of Server 2, sitting at 47.2%. Server 3 though shows a low percentage of queries (8.3%), which is a good sign, as a server with higher latency should respond later to the resolver and thus be undesirable. In Figure 5.11 we see something that we do not understand and can not, at this moment, explain the reasoning behind it: throughout the test, Server 3 is barely contacted, except for the final third of the test, which is equivalent to the last 20 minutes, where it is contacted very frequently for a brief period of time. In fact, it seems to be contacted roughly with the same frequency as the remaining two servers during that time period. We do not possess a plausible reason for this behaviour at the time. By the second test, we see in Figure 5.12 that the query frequency has almost

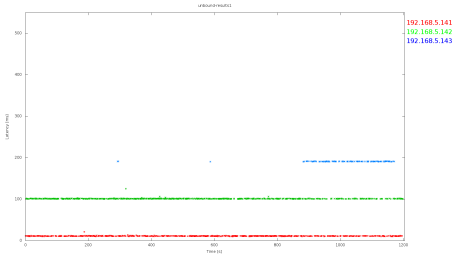

Figure 5.11: Unbound Test 1

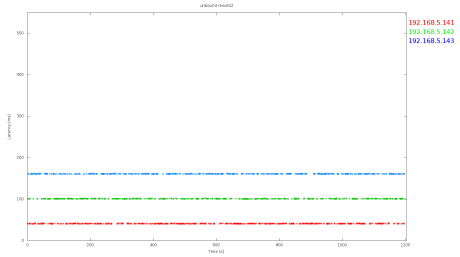

Figure 5.12: Unbound Test 2

balanced out among the resolvers. The test still shows a slight preference for Server 1, but since Test 1 did not reveal such a preference, we believe this falls within the acceptable statistical deviation.

With the third and fourth tests, we reach a balance in the query percentage among all three servers. As stated before, having a balance at Test 4 is advisable, which is present in the resolver. It does, however, reach that balance earlier than anticipated.

Given the results obtained from these tests, it seems that the probability of a server being selected for a certain query is less relative to the measured latency of previous queries and more close to a random distribution. Despite this, it is possible that Unbound has some sort of range, within which it treats all servers as equal. Since it prefers Servers 1

and 2 on the first test, we believe that this range does favour lower latency servers. Given our classification criteria for this group of tests, we classify Unbound as a not optimal resolver, since it does not choose the best server a majority of the time.

5.3.2 Latency Cut-off Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 5	11.24ms	48.9%	101.28ms	50.9%	500.72ms	0.2%
Test 6	41.37ms	48.3%	101.36ms	51.3%	501.35ms	0.4%
Test 7	71.33ms	50.1%	101.33ms	49.7%	501.30ms	0.2%
Test 8	101.79ms	48.5%	101.47ms	51.2%	501.09ms	0.3%

Table 5.12: Results of the Latency Cut-off Tests category from the Unbound resolver

Tests 5,6,7 and 8 seem to indicate that Unbound does not retain its cut-off point, which was shown to be 500ms in Yu, Wessels, Larson, and Zhang [23]. This is the case since in all the tests, Server 3 was chosen for some queries (albeit a very small number of queries) as the server to be contacted by Unbound. In addition to this, these tests further reinforce our claim that Unbound does not always select the server with the lowest latency but does seem to have a range of values within which it chooses server from, in addition to querying every server periodically, since Server 3 is still contacted in these tests.

5.3.3 Packet Loss Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 9	17.45ms	49.9%	102.10ms	49.7%	191.98ms	0.4%
Test 10	25.55ms	44.6%	103.18ms	53.3%	196.55ms	1.9%

Table 5.13: Results of the Packet Loss Tests category from the Unbound resolver

Test number nine was performed with a packet loss value of 10% set on Server 1, while Test number ten had Server 1 with a packet loss of 30%. In Test 9, Server 3 received less than 1% of queries, while Server 1 and 2 received around half of all queries each. At first glance it seems that packet loss had no effect in this test, since the query percentages are quite similar to the tests that do not experience packet loss. However, when comparing the two generated graphs seen in figures 5.13 and 5.14 (which also displays the packets that were lost) we see that Unbound seems to be quickly acknowledging that packets have been lost. Once a packet has been deemed as lost, Unbound sends it again, to the same server. Test 10 displayed the same patterns of quickly realising that a packet had not reached its destination and is thus sent once again. This strategy of handling packet loss differs from the previous two analysed resolvers, in that they treat a failed query as the server being unreachable, instead of instantly double checking if the server is still reachable and merely experienced a network hiccup.

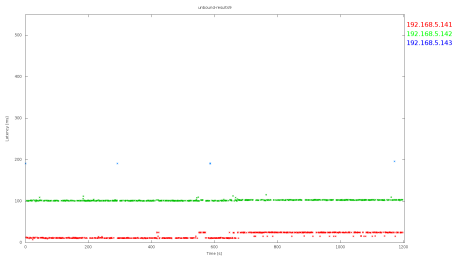

Figure 5.13: Unbound Test 9

Figure 5.14: Unbound Test 9 with dropped packets

5.3.4 Network Topology Change Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 11	24.37ms	11.5%	102.97ms	67.1%	196.60ms	21.4%

Table 5.14: Results of the Network Topology Change Tests category from the Unbound resolver

In this test, Unbound adapted to the new network topology very slowly, as is hinted at by the low amount of queries directed towards Server 1, with only 11.5%. To further analyse its response time, we looked at figures 5.15 and 5.16. By resorting to these graphs, we were able to verify that when Unbound lost contact with Server 1, it sent four queries to Server 1 in very quick succession (it is likely that the resolver is retrying the same original failed query), which, since the server is down, are unanswered. After that, Unbound tries again to send two queries to the server that is down, roughly 90 seconds after the first four queries were sent.

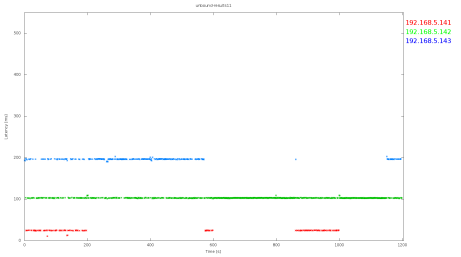

Figure 5.15: Unbound Test 11

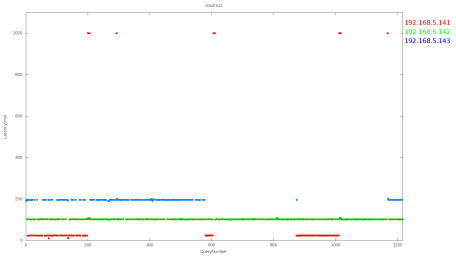

Figure 5.16: Unbound Test 11, with failed queries shown

Once those queries fail, since the server is still down at this point, Unbound then takes approximately 14 minutes to contact the server that has been up for over five minutes, thus showing a very poor score in this particular test. It does, however, recover more quickly from the second downtime period, but it still remains the fact that it took over five minutes to contact a newly reachable server.

Our findings in this test support the findings from Yu, Wessels, Larson, and Zhang [23], which state that Unbound periodically queries unresponsive servers every 15 minutes. Since we found that Unbound also employs a mechanism to quickly check if the server is available and the failed queries were simply a freak occurrence, Unbound seems to have been changed to handle very short periods of unavailability, while still retaining it's periodic poll rate to unavailable servers. This mechanism thus has a maximum response time of approximately 14 minutes, as was shown on our test, which classifies this resolver as being relatively slow to adapt to successive network topology changes. However, in a real world environment, outages should not be in the order of magnitude experienced in this test, which turns these results into a very favourable characteristic.

5.3.5 Random Query Rate and Interval Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 12	24.26ms	48.6%	102.84ms	51.1%	196.64ms	0.3%

Table 5.15: Results of the Random Query Rate Tests category from the Unbound resolver

Test number twelve shows the same alternating behaviour of choosing different servers that was seen on Test 1. It also revealed that Unbound suffers from a similar problem to Bind, in that it sometimes replies to the user with a record from its cache, while the value of that record's TTL is 0. This was discovered upon a more thorough analysis which was prompted by the discrepancy in the expected query number of at least 1800 queries in the test, to the obtained 1255 queries.

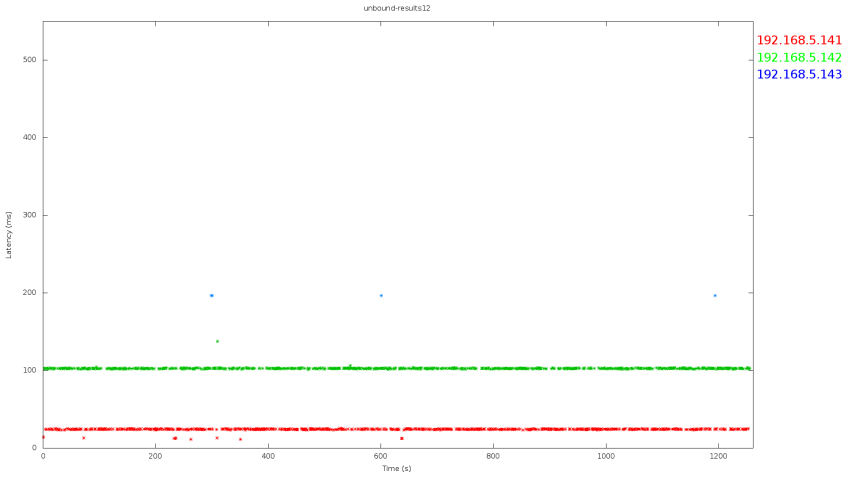

Figure 5.17: Unbound Test 12

Upon completing all the tests for Unbound, we can now say a few things about it. It seems that it no longer possesses its previous 500ms latency cut-off point. It does not choose the best server a majority of the time, however it does alternate between the best two servers. Unbound also took a very large amount of time to react to successive network topology changes. Despite this slow reaction time, it did check to make sure that it was indeed a big outage and not a small network anomaly, which is a good plus.

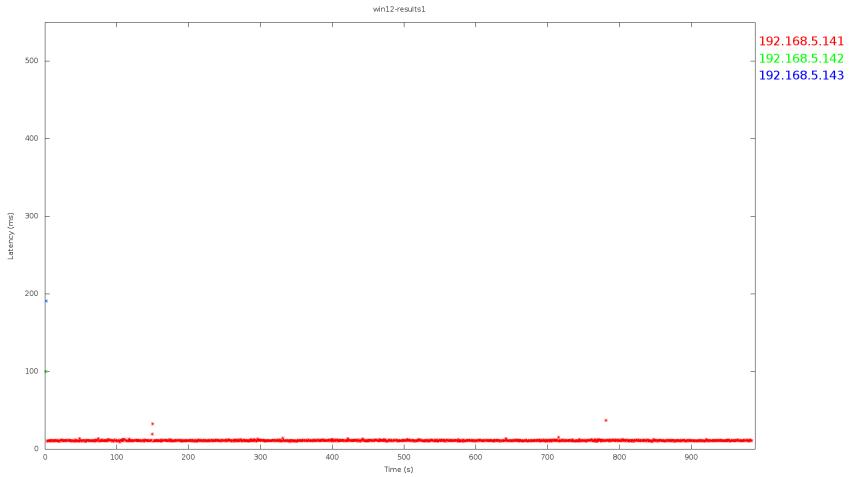

Figure 5.18: Windows12 Test 1

5.4 Windows12

All the tests performed on Windows12 share an anomaly: instead of the expected 1200 queries, we find that the number of queries actually performed by Windows12 is not even constant. In fact, it hovers around 1000 queries. This is very strange behaviour, since we perform one query every three seconds, and the resource record we query has a TTL of two seconds, we should always have 1200 queries. The fact that we have less is indicative of Windows12 not respecting the TTL set, as it is the only way to have a lower amount of queries, which was proven to be true in previous resolvers' results.

5.4.1 Standard Tests

	Server 1		**Server 2**		**Server 3**	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 1	11.18ms	99.8%	100.31ms	0.1%	190.75ms	0.1%
Test 2	41.36ms	99.9%	100.70ms	0.1%	n/a	0%
Test 3	71.06ms	51.3%	101.19ms	48.7%	n/a	0%
Test 4	n/a	0%	101.02ms	0.1%	101.13ms	99.9%

Table 5.16: Results of the Standard Tests category of the Windows12 resolver

The first test reveals that Windows12 chooses the server with the lowest latency most of the time. However, it also shows that Windows12 does not take in consideration the possibility of, eventually, a different server could be faster. Therefore, it starts and it contacts Server 2 and Server 3 only once, as seen in Figure 5.18. We assume it stores the latency it experienced, and then contacts Server 1. As Server 1 seems to have a much lower latency, it is then the server of choice for the duration of the test. This may prove to be a problematic point, depending on the next tests' results.

In the second test, we see a repeat of the pattern observed in the first test. It initially ends up querying Server 1 and stores the latency it experienced. Two more queries are sent to Server 1 before it decides to check on the availability of remaining servers. Since it finds Server 2, which has a higher latency than Server 1, it does not choose this server any further. Server 3 is never selected in Test 2 and therefore we do not measure a latency to it.

In the third test, Windows12 now attempts to find a better server, as shown by the query percentages of Server 1 and Server 2. It seems that it reaches a point where it is not satisfied with the latency experienced and thus attempts to find a better server. Server 3 is, again, not queried at all during this test.

In the final test, Server 3 is the most queried server, with Server 2 having only a single query, while Server 1 is not queried in this test at all.

With these tests, we believe that Windows12 is indeed choosing the best server. However, it apparently does not employ a mechanism through which it checks if it is still contacting the best server, as there are tests in which a server is not contacted at all. Test 4 also shows that Windows12 does not attempt to spread the query load in some situations.

5.4.2 Latency Cut-off Tests

	Server 1		**Server 2**		**Server 3**	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 5	11.09ms	99.7%	100.65ms	0.2%	500.75ms	0.1%
Test 6	41.21ms	38.0%	101.18ms	61.8%	500.91ms	0.2%
Test 7	71.14ms	99.9%	100.63ms	0.1%	n/a	0%
Test 8	102.33ms	50.0%	102.05ms	49.9%	500.67ms	0.1%

Table 5.17: Results of the Latency Cut-off Tests category from the Windows12 resolver

In Test 5, Windows12 contacts Server 3 only once, and thus we are inclined to think it does have a cut-off point. However, given its past behaviour, we can't yet assume that to be the case. With Test 6, however, we discover that it does not have a cut-off point, as Server 3 is contacted twice. Still in this test, there is some weird behaviour. Despite there not being any packet loss, which we verified, Windows12 still switches before 20 minutes of testing has passed, as seen in Figure 5.19. This is strange behaviour, as previously Windows12 generally chose the same server, especially in this mirror test on the first set (Test 2).

Test 7 shows the absence of the anomaly recorded in Test 6, and returns to the values displayed in Test 5. It doesn't, however, contact Server 3 a single time.

Test 8 then shows the resolver switching between selected servers, as evidenced by the fact that both Server 1 and Server 2 have almost equal query percentages, while still contacting Server 3 once.

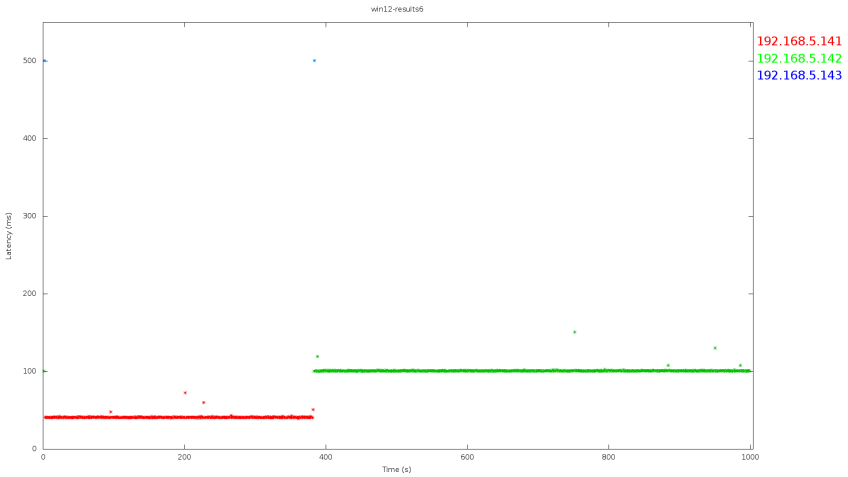

Figure 5.19: Windows12 Test 6

With the results of these tests reviewed, we can't quite comprehend the exact behaviour of the resolver. However, we do verify that it almost always selects the server with the lowest latency, as was observed in the first group of tests. We can also say that Windows12 does not appear to have the cut-off point that these tests check for.

5.4.3 Packet Loss Tests

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 9	n/a	0%	101.14ms	99.9%	190.74ms	0.1%
Test 10	n/a	0%	101.19ms	99.9%	190.44ms	0.1%

Table 5.18: Results of the Packet Loss Tests category from the Windows12 resolver

Test 9 and Test 10 have identical query percentages, with Server 2 receiving 99.9% of queries in both tests. while Server 3 receives the remaining 0.1% and Server 1 is never selected. Since Server 1 is never selected, in an exact setup where it once was, with the exception of the packet loss, we have to conclude that Windows12 must have some sort of mechanism that does not rely on DNS queries to classify the servers. We believe this is the case as Server 1 should have been selected in either of these tests and only after it had failed to reply to a query would then change server. It is possible that in both tests, both initial queries to Server 1 had failed, with a chance of 0.3% of that situation occurring. We can, however, verify that is not the case, as can be seen in Figure 5.20 which shows the queries sent by the resolver, including those which had their packets lost. This hypothesis may explain the previously seen behaviour of this resolver. However, we could not perform a deeper investigation of it.

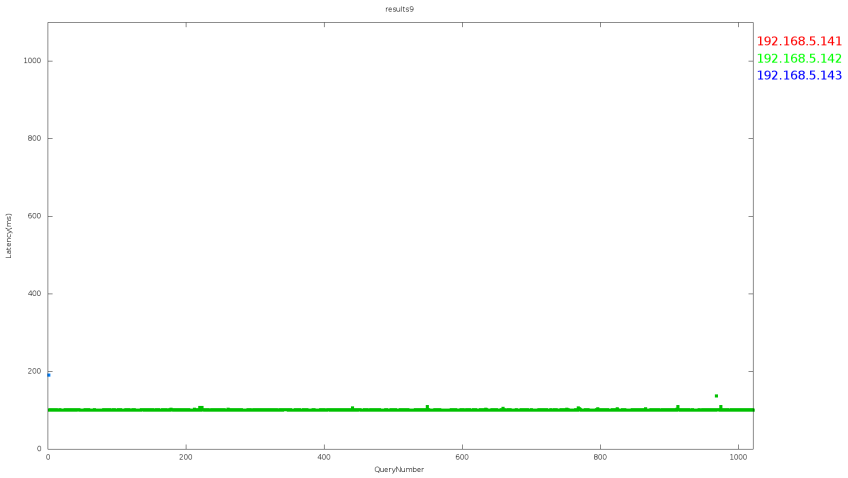

Figure 5.20: Windows12 Test 9, with failed queries shown

5.4.4 Network Topology Change Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 11	11.43ms	16.1%	101.28ms	83.9%	n/a	0%

Table 5.19: Results of the Network Topology Change Tests category from the Windows12 resolver

Looking only at the query percentages, this test seems to demonstrate that Windows12 is not contacting Server 1 after it becomes reachable again. In fact, by looking at the graph of this test (Figure 5.21) we can indeed see that this is the case. It should be noted that there is a slight discrepancy in this graph, when compared with the other resolvers. According to our testing methodology, the server should go down at query number 200. The graph shows that Server 1 goes down before that. This is explained by the fact that Windows12 does not seem to be respecting the TTL of its cache, as evidenced by the number of queries analysed, which is always less than the expected 1200. Thus, since our testing assumes a steady query to time ratio, this graph has this slight discrepancy.

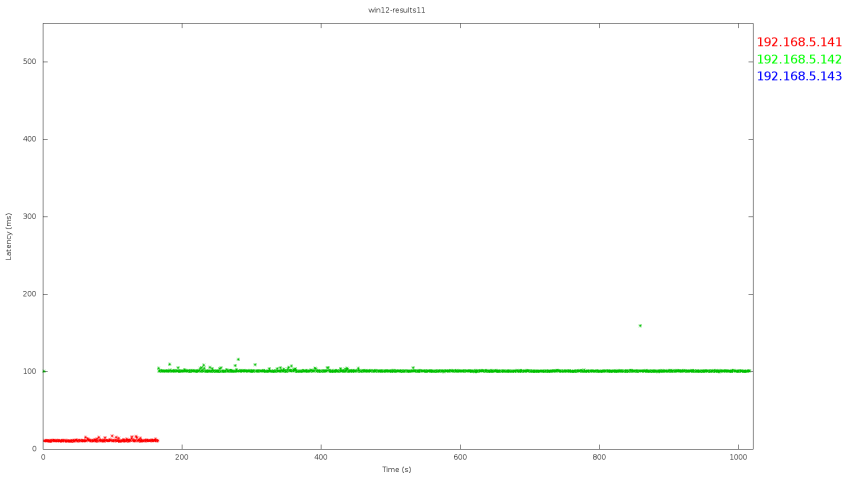

Figure 5.21: Windows12 Test 11

5.4.5 Random Query Rate and Interval Test

	Server 1		Server 2		Server 3	
	Average Latency	Query Percentage	Average Latency	Query Percentage	Average Latency	Query Percentage
Test 12	11.14ms	50.3%	101.16ms	49.7%	n/a	0%

Table 5.20: Results of the Random Query Rate Tests category from the Windows12 resolver

Test twelve also reveals strange behaviour. Windows12 alternates between choosing Server 1 and Server 2, and also performs more than 1200 queries but less than the expected 1800. This confirms our theory that Windows12 is not respecting the TTL. Besides that, we are not certain why, in this specific test, it is alternating between both Server 1 and Server 2. In fact, it does not even contact Server 3 once. We can not explain the fact that both Server 1 and Server 2 are selected 50% of the time on this test but not on the first test, for instance.

Given all the results from Windows 12, we classify it as an average resolver, when compared to the other three. It does indeed choose the best server in terms of latency in almost all cases. It does not have the cut-off point, making it the fourth resolver to not exhibit that feature. It does not react positively to successive network topology changes, as Test 11 revealed that after a server comes back up, it does not get contacted again.

In fact, to provide a fair classification, more tests are needed to this resolver, as it is necessary to understand the rationale behind the behaviour exhibited in our tests. We suggest looking at other potential message exchanges between the Windows12 resolver and the servers, as it seems very likely that it performs some measurements without resorting to the DNS messages.

5.5 Client POV and Conclusion

One flaw of all our tests is that they do not measure the latency felt by the client. This was done by design, as we were interested in how the resolvers chose which authoritative name server to contact. However, in order to be able to properly rank them, we feel that we need to address this final concern. To that effect, we took the .pcap files of all the tests and fed them through a script that computed the average query latency for all the tests, except Test 12, as that test is influenced by the caches of the resolvers. The values we obtained are shown in 5.21.

	Average Latency
Bind	76.219 ms
PowerDNS	89.483 ms
Unbound	88.567 ms
Windows12	63.692 ms

Table 5.21: Average latency experienced from the client point of view

From this final analysis of the data collected, we, surprisingly, see that Windows12 provided the best user experience, boasting the lowest latency of all three resolvers. Upon some consideration, we came to the conclusion that this is due to the fact that Windows12 does not respect the cached TTL, and thus serves the clients more replies from its cache than the other resolvers, which don't serve any responses from their caches in Tests 1 through 11. For reference, assuming we had a perfect resolver that always chose the server with the lowest latency, that hypothetical resolver would exhibit an average latency value of 50.09ms.

Table 5.22 puts in perspective some of the conclusions we gathered with our tests.

	Latency Cut-off	**Cache Management**	**Load Balancing**
Bind	No	Does not respect TTL	Yes
PowerDNS	No	Likely pre-fetching	Yes
Unbound	No	Does not respect TTL	Yes
Windows	No	Does not respect TTL	No

Table 5.22: Comparison between features of all the resolvers

To conclude, we found that:

- Bind, PowerDNS and Windows12 chose the server with the lowest latency the majority of the time.

- None of the resolvers had a latency cut-off at 500ms, despite one resolver previously having it.
- Bind, PowerDNS and Windows12 assume that if a server does not respond to a query, it is unreachable, while Unbound first tests to see if it was a simple network anomaly.
- PowerDNS and Bind were the fastest to respond to successive network topology changes. Unbound took several minutes, while Windows12 did not respond at all.
- Only PowerDNS employs pre-fetching. The other resolvers do not always respect the cached TTL.
- Finally, all resolvers except Windows12 try to balance the interest of their clients (less latency) with the interests of the network (load balancing)

Chapter 6

Conclusions

This thesis had the goal of understanding the behaviour of some DNS resolvers, more specifically how they choose which server to contact. To that end, we first began by studying previous work that had been done in this topic. Through our research, we discovered a previous study detailing the exact behaviour of some resolvers, which was discovered by analysing the resolvers' code. This differs from our approach, which requires us to strictly analyse the data produced by our tests, as some resolvers do not have an open source nature. It does, however, provide us with a starting point by analysing their methodology and results, which confirmed that our initial idea of observing the resolvers in an enclosed network and measure their performance through the latency experienced by each query was a good starting point. We also were able to determine an alternative way that resolvers use to choose a server, involving the anycast protocol, which places the burden of the decision upon the network itself. Since this removes the part that the resolvers have in the decision of which server to contact, we do not focus our thesis in this protocol, but offer it as a possible alternative.

Having confirmed our initial idea of basing our testing environment on the latency experienced by the resolver, we designed several tests to measure their behaviour. Our tests measured how a resolver behaved in a static environment, attempted to understand if there is a certain latency after which a server is no longer contacted, how do they balance the query load, how a resolver handles packet loss and network topology changes and, how it responds to a slightly higher load and whether or not it implies pre-fecthing. We came up with twelve tests to serve our purpose. With the data from all the tests, we were also able to determine how good the resolvers were, from a client's point of view.

For our testing environment, we settled on using three authoritative name servers. This implies that we need, at the very minimum, a total of five machines: one for each authoritative name server, one for the resolver and one for the client. We ended up

implementing a structure with seven machines, in which the two additional machines were our own version of the root authoritative name server and a packet sniffer, to collect the data from our tests.

We evaluated four resolvers (Bind, PowerDNS, Unbound and Windows-Server 2012) in our testing, and found that the two most well-rounded resolvers were Bind and PowerDNS, as they performed the best in our tests, distributing the load in situations where it was possible, very frequently choosing the best server and reacting to packet loss and network topology changes quite quickly. PowerDNS does seem to be better, though, as it has pre-fetching, while Bind does not.

From a client's point of view, the resolvers all seemed relatively equal (once we accounted for the disrespect of the TTL by Windows12), with Bind showing to be the best one.

Our testing had some limitations, in that it was not able to tell how Windows12 knew which server to choose (we suspect it has some other mechanism for this purpose, since it does not use the DNS messages for it). Given this limitation, we think that it would be interesting to study Windows12 more closely, to understand how exactly it evaluates the servers. It would also be interesting to understand how the resolvers differentiate between packet loss and an unreachable ping.

We believe that our testing environment can be applied to any resolver, provided the very small configuration adjustments are performed, and thus is our main contribution with this thesis, along with the results we obtained.

Bibliography

[1] *BIND 9.11.3*. URL: http://www.isc.org/downloads/bind/.

[2] *DNS Structure*. URL: https://www.dns.pt/pt/dominios-2/o-sistema-dns/ (visited on 01/29/2018).

[3] *DNSCache*. URL: http://cr.yp.to/djbdns/dnscache.html.

[4] *DNSJit*. URL: https://github.com/DNS-OARC/dnsjit.

[5] T. Hardie. *Distributing Authoritative Name Servers via Shared Unicast Addresses*. RFC 3258. http://www.rfc-editor.org/rfc/rfc3258.txt. RFC Editor, 2002. URL: http://www.rfc-editor.org/rfc/rfc3258.txt.

[6] *IANA*. URL: https://www.iana.org/domains/root/files.

[7] J. Legatheux. In: *Fundamentos de Redes de Computadores*. 2017. Chap. 1, p. 28.

[8] P. Mockapetris. *Domain names - concepts and facilities*. STD 13. http://www.rfc-editor.org/rfc/rfc1034.txt. RFC Editor, 1987. URL: http://www.rfc-editor.org/rfc/rfc1034.txt.

[9] P. Mockapetris. *Domain names - implementation and specification*. STD 13. http://www.rfc-editor.org/rfc/rfc1035.txt. RFC Editor, 1987. URL: http://www.rfc-editor.org/rfc/rfc1035.txt.

[10] M. Müller, G. C. M. Moura, R. de O. Schmidt, and J. Heidemann. *Recursives in the Wild: Engineering Authoritative DNS Servers*. Tech. rep. ISI-TR-720. Available: https://www.isi.edu/~johnh/PAPERS/Mueller17a.pdf. johnh: pafile: USC/Information Sciences Institute, June 2017. URL: http://www.isi.edu/%7ejohnh/PAPERS/Mueller17a.html.

[11] R. de O. Schmidt, J. Heidemann, and J. Harm Kuipers. "Anycast Latency: How Many Sites Are Enough?" In: *Proceedings of the Passive and Active Measurement Workshop*. 2017.

[12] *PlanetLab*. URL: https://www.planet-lab.org/ (visited on 02/18/2018).

[13] *PowerDNS 4.1.2*. URL: http://www.powerdns.com/.

[14] *qdisc*. URL: http://tldp.org/HOWTO/Traffic-Control-HOWTO/components.html.

[15] *RIPE Atlas*. URL: https://atlas.ripe.net/ (visited on 02/18/2018).

[16] *root-servers.org*. URL: http://www.root-servers.org/ (visited on 01/29/2018).

[17] S. Sandeep, V. Pappas, and A. Terzis. "On the Use of Anycast in DNS." In: *ACM SIGMETRICS*. 2005.

[18] A. Singla, B. Chandrasekara, P. B. Godfrey, and B. Magg. "The Internet at the Speed of Light." In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. 2014.

[19] *tc-netem*. URL: http://man7.org/linux/man-pages/man8/tc-netem.8.html.

[20] *Unbound 1.7.0*. URL: http://www.unbound.net/.

[21] C. Villamizar, R. Chandra, and R. Govindan. *BGP Route Flap Damping*. RFC 2439. RFC Editor, 1998.

[22] *Windows Server 2012 R2 Essentials*. URL: https://www.microsoft.com/en-us/cloud-platform/windows-server.

[23] Y. Yu, D. Wessels, M. Larson, and L. Zhang. "Authority Server Selection of DNS Caching Resolvers." In: *SIGCOMM Computer Communication Review*. 2012.

Annex I

Annex 1

I.1 Bind Graphs

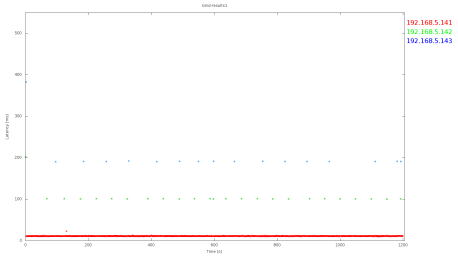

Figure I.1: BIND Test 1

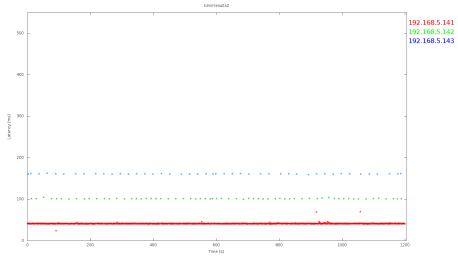

Figure I.2: BIND Test 2

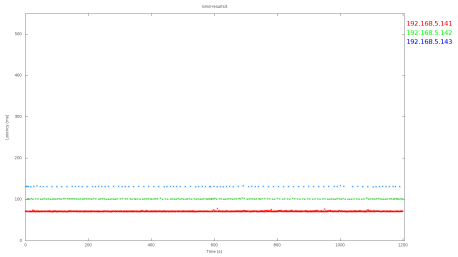

Figure I.3: BIND Test 3

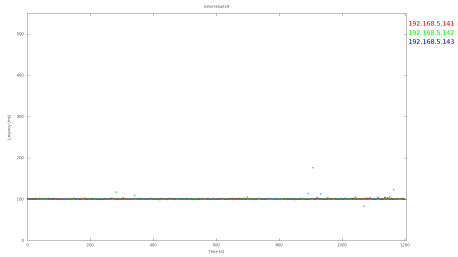

Figure I.4: BIND Test 4

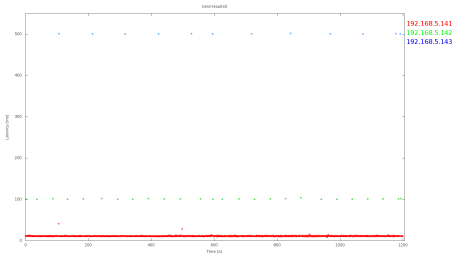

Figure I.5: BIND Test 5

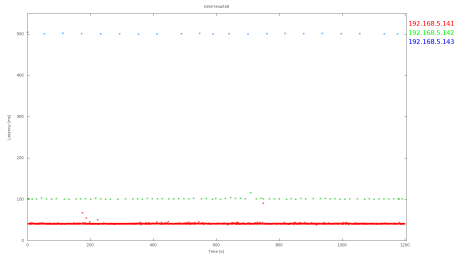

Figure I.6: BIND Test 6

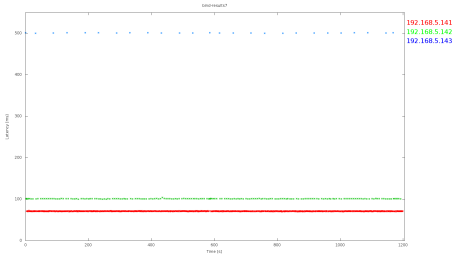

Figure I.7: BIND Test 7

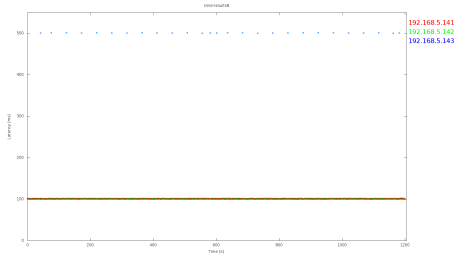

Figure I.8: BIND Test 8

I.2 PowerDNS Graphs

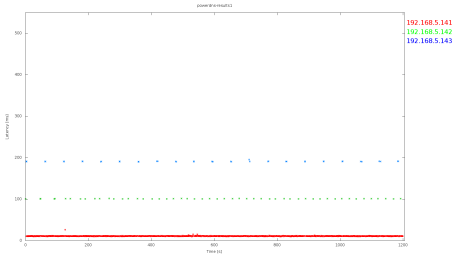

Figure I.9: PowerDNS Test 1

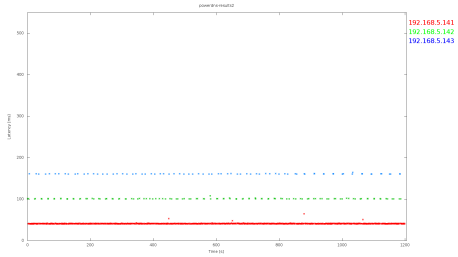

Figure I.10: PowerDNS Test 2

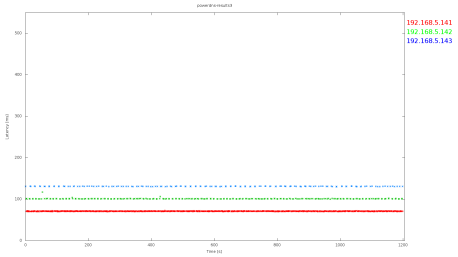

Figure I.11: PowerDNS Test 3

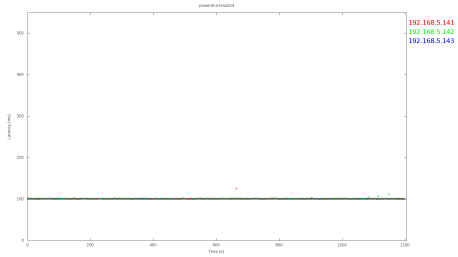

Figure I.12: PowerDNS Test 4

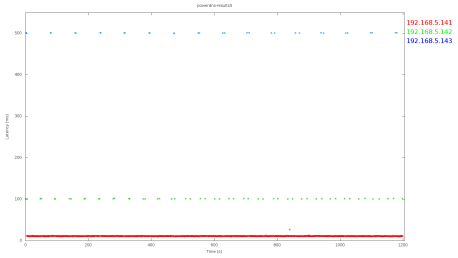

Figure I.13: PowerDNS Test 5

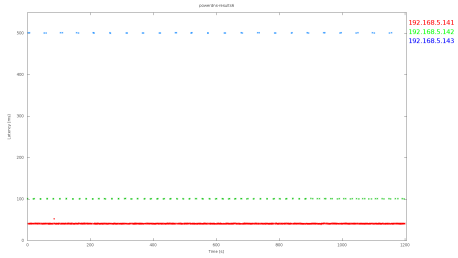

Figure I.14: PowerDNS Test 6

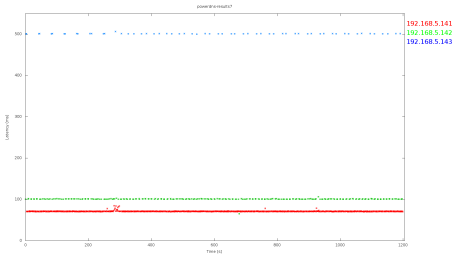

Figure I.15: PowerDNS Test 7

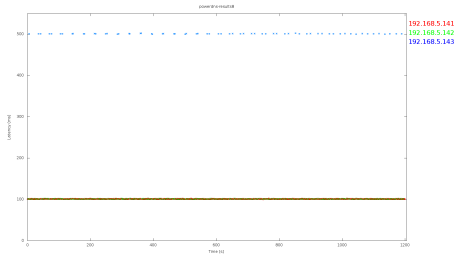

Figure I.16: PowerDNS Test 8

I.3 Unbound Graphs

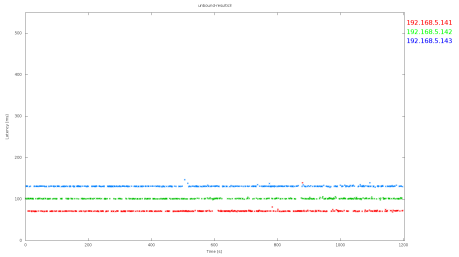

Figure I.17: Unbound Test 3

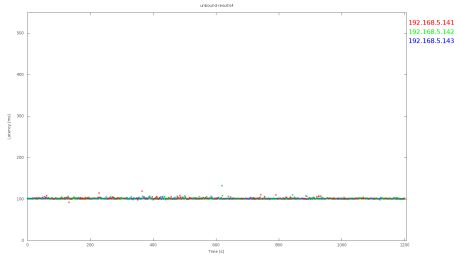

Figure I.18: Unbound Test 4

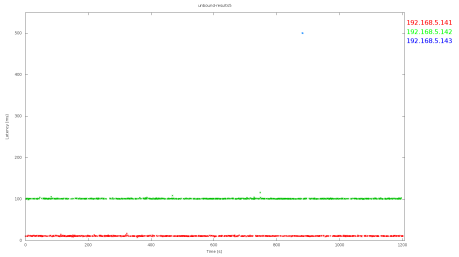

Figure I.19: Unbound Test 5

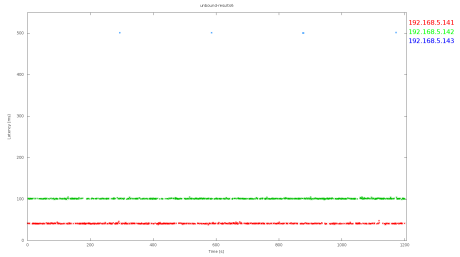

Figure I.20: Unbound Test 6

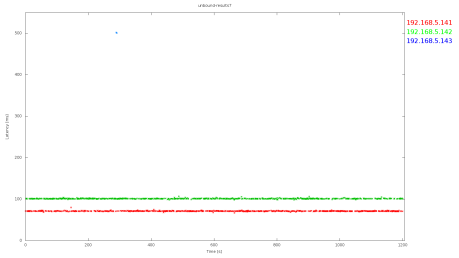

Figure I.21: Unbound Test 7

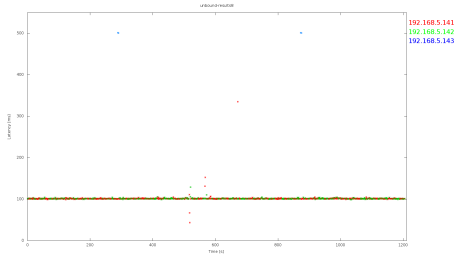

Figure I.22: Unbound Test 8

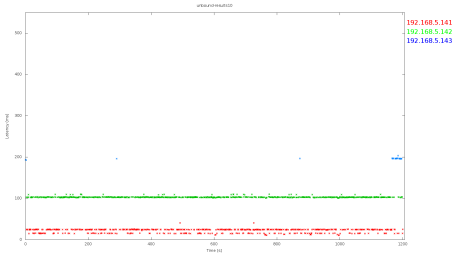

Figure I.23: Unbound Test 10

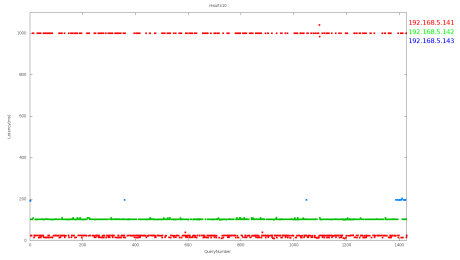

Figure I.24: Unbound Test 10 with dropped packets

I.4 Windows12 Graphs

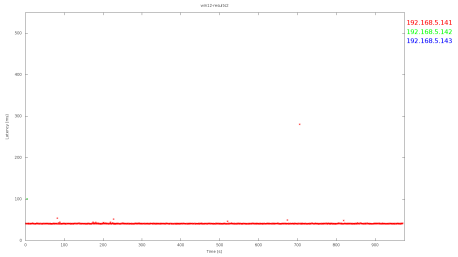

Figure I.25: Windows12 Test 2

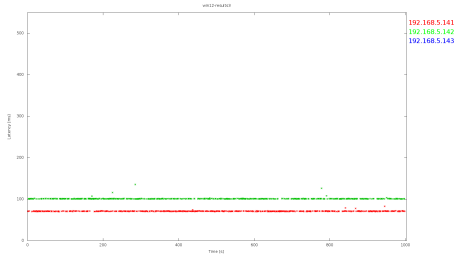

Figure I.26: Windows12 Test 3

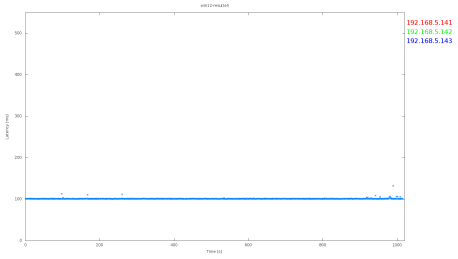

Figure I.27: Windows12 Test 4

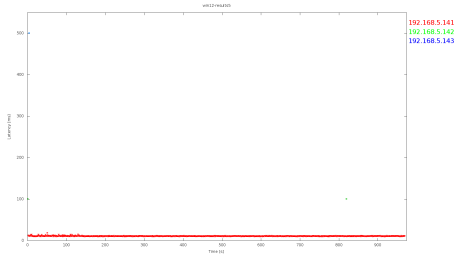

Figure I.28: Windows12 Test 5

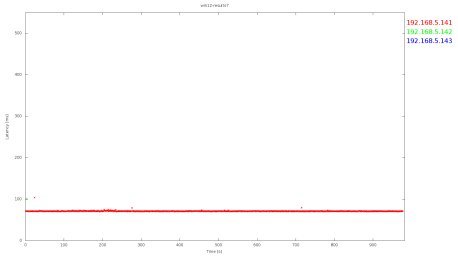

Figure I.29: Windows12 Test 7

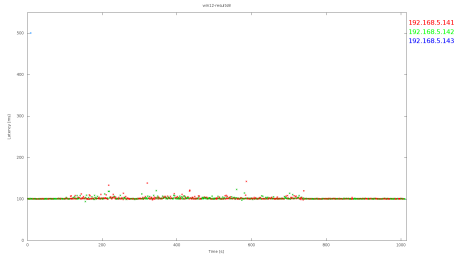

Figure I.30: Windows12 Test 8

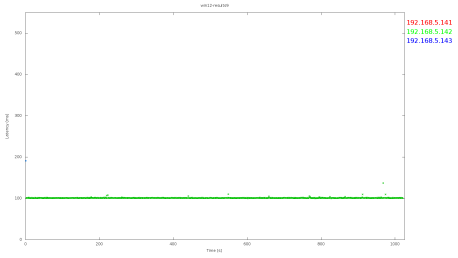

Figure I.31: Windows12 Test 9

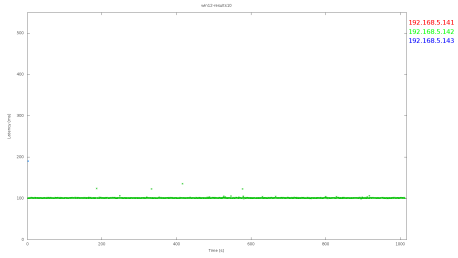

Figure I.32: Windows12 Test 10

ANNEX II

ANNEX 2

II.1 Analyse Results

```
#! / usr / bin / env dnsjit
local pcap = arg[2]
local tested_domain = arg[3]
local test_runtime = arg[4]
local query_interval = arg[5]
local filename = arg[6]
local resolver_used = arg[7]
local ttl_used = arg[8]
local servers = {}

local index = 9
while arg[index]  and arg[index + 1] and arg[index + 2] and arg[index+3] do
  table.insert(servers,{
    address = arg[index],
    latency = arg[index+1],
    packetloss = arg[index+2],
    shutdownInterval = arg[index+3]
  })
  index = index + 4
end


if pcap == nil then
  print("usage: "..arg[1].." <pcap>")
  return
end

require("dnsjit.core.object")
require("dnsjit.core.object.packet")
```

```
30
31 output = require("dnsjit.filter.coro").new()
32
33 local queries_by_id = {}
34
35 local query_info = {}
36
37 local lowest_time = 9999999999999999
38 local total_latency = 0
39 local total_count = 0
40
41
42 output:func(function(filter, obj)
43   --print("entered method")
44   local pkt = obj:cast()
45   local dns
46   if pkt:type() == "packet" then
47     dns = require("dnsjit.core.object.dns").new(obj)
48     if dns:parse() ~= 0 then
49       return
50     end
51   elseif pkt:type() == "dns" then
52     dns = pkt
53     if dns:parse() ~= 0 then
54       return
55     end
56     pkt = dns:prev()
57     while pkt ~= nil do
58       if pkt:type() == "packet" then
59         pkt = pkt:cast()
60         break
61       end
62       pkt = pkt:prev()
63     end
64     if pkt == nil then
65       return
66     end
67   else
68     return
69   end
70   if dns then
71     if dns.qr == 0 then
72       local n = dns.questions
73
74       while n > 0 and dns:rr_next() == 0 do
75         if dns:rr_ok() == 1 then
76           label = dns:rr_label()
77           --print("label")
78
79
```

```
80          --print("query_id ", dns.id)
81
82          init_time = tonumber(pkt.ts.sec) * 1000000000 + tonumber(pkt.ts.nsec
                ↪ )
83          queries_by_id[dns.id] = init_time
84
85          --dns:print()
86
87        end
88        n = n - 1
89      end
90      --print("\n")
91    end
92  end
93  if dns.qr == 1 then
94    if dns.questions > 0 and dns:rr_next() == 0 and dns:rr_ok() then
95      if string.match(dns:rr_label(), tested_domain) then
96        if dns.aa == 1 then
97          --print("response_id ", dns.id)
98          if queries_by_id[dns.id] then
99            latency = tonumber(pkt.ts.sec) * 1000000000 + tonumber(pkt.ts.nsec
                  ↪ ) - queries_by_id[dns.id]
100           if latency < lowest_time then
101             lowest_time = latency
102           end
103           --print(latency / 1000000000)
104           total_latency = total_latency + latency
105           total_count = total_count + 1
106
107           queries_by_id[dns.id] = nil
108           table.insert(query_info, {
109             latency = latency,
110             resolver = pkt:dst(),
111             resolver_port = pkt.dport,
112             authoritative = pkt:src(),
113             authoritative_port = pkt.sport,
114             query = dns:rr_label()
115           })
116         end
117       end
118     end
119   end
120   --dns:print()
121 end
122 end
123 )
124 input = require("dnsjit.input.pcapthread").new()
125
126 print("analyzing␣file␣" .. pcap)
127 input:open_offline(pcap)
```

```
input:receiver(output)
input:run()
```

```
authoritative_table = {}
avg_latency_to_auth = {}
query_count = 0
for _, q in pairs(query_info) do
query_count = query_count + 1
if authoritative_table[q.authoritative] then
  authoritative_table[q.authoritative] = authoritative_table[q.authoritative] + 1
  table.insert(avg_latency_to_auth[q.authoritative], q.latency)

else
  authoritative_table[q.authoritative] = 1
  avg_latency_to_auth[q.authoritative] = {}
  table.insert(avg_latency_to_auth[q.authoritative], q.latency)
end
end

print("results/".. resolver_used .. "/"..filename ..  ".txt")
local file = io.open("results/".. resolver_used .. "/"..filename ..  ".txt", "w")
io.output(file)

io.write("Analysis␣of␣traffic␣from␣".. resolver_used .."␣to␣.net␣authoritative␣name␣servers␣(one␣hour␣period)\n␣while␣answering␣to␣queries:␣" .. tested_domain)
io.write("\nTest␣runtime␣(s):␣", test_runtime, "\n")
io.write("Query␣interval␣(s):␣", query_interval, "␣\n")
io.write("TTL␣used␣in␣test␣(s):␣", ttl_used, "\n")

for _, v in pairs(servers) do
io.write("Server:␣", v.address, "␣␣Latency␣(ms):␣", v.latency, "␣Packetloss␣(%):␣", v.packetloss , "␣Shutdown␣Interval␣(s):␣", v.shutdownInterval, "\n")
end


io.write("\n")

for k, v in pairs(authoritative_table) do
table = avg_latency_to_auth[k]
count = 0
latency = 0
```

```
min_latency = 99999999999999999999999999999 -- a large starting number so it gets properly updated - this is equivalent to roughly one sextillion seconds
max_latency = 0
for _, av in pairs(table) do
  count = count + 1
  latency = latency + av
  min_latency = math.min(min_latency,av)
  max_latency = math.max(max_latency,av)
end
--print(latency, count)
io.write("Authoritative: ", k, "  QueriesToAuth: ", v, "  QueryPercentage: ", v / query_count, "  MinLatency(ms): ", (min_latency ) / 1000000,
"  MaxLatency(ms): ", (max_latency ) / 1000000,"  AvgLatency(ms): ", (latency / count) / 1000000, "\n")
end


io.close(file)

print("Wrote to file " .. filename .. ".txt")
print("cat results/"..filename.. ".txt")
```

II.2 Draw Graph

```
#! / usr / bin / env dnsjit
local pcap = arg[2]
local tested_domain = arg[3]
local test_runtime = arg[4]
local query_interval = arg[5]
local filename = arg[6]
local resolver_used = arg[7]
local resolver_ip_addr = arg[8]
local servers = {}

local index = 9
while arg[index]  and arg[index + 1] and arg[index + 2] and arg[index+3] do
  print(arg[index])
  table.insert(servers,{
    address = arg[index],
    latency = arg[index+1],
    packetloss = arg[index+2],
    shutdownInterval = arg[index+3]
  })
  index = index + 4
end


```

```
24 if pcap == nil then
25   print("usage: "..arg[1].." <pcap>")
26   return
27 end
28
29 require("dnsjit.core.object")
30 require("dnsjit.core.object.packet")
31
32 output = require("dnsjit.filter.coro").new()
33
34 local queries_by_id = {}
35 local query_info = {}
36 local lowest_time = 9999999999999999
37 local total_latency = 0
38 local total_count = 0
39 -- ********************************************************
40
41
42 local queries_to_domain = {}
43 local max_latency_allowed = 1000000000 / 2
44 local queryNumber = 0
45 local srv_addresses = {}
46 for _,s in pairs(servers) do
47   srv_addresses[s.address] = 1
48 end
49
50
51 output:func(function(filter, obj)
52   --print("entered method")
53   local pkt = obj:cast()
54   local dns
55   if pkt:type() == "packet" then
56     dns = require("dnsjit.core.object.dns").new(obj)
57     if dns:parse() ~= 0 then
58       return
59     end
60   elseif pkt:type() == "dns" then
61     dns = pkt
62     if dns:parse() ~= 0 then
63       return
64     end
65     pkt = dns:prev()
66     while pkt ~= nil do
67       if pkt:type() == "packet" then
68         pkt = pkt:cast()
69         break
70       end
71       pkt = pkt:prev()
72     end
73     if pkt == nil then
```

```
74        return
75      end
76    else
77      return
78    end
79    if dns then
80      if dns.qr == 0 then
81        local n = dns.questions
82        while n > 0 and dns:rr_next() == 0 do
83          if dns:rr_ok() == 1 then
84            label = dns:rr_label()
85            --print(label .. " " .. tested_domain .. " " .. resolver_ip_addr)
86            if label == nil then
87              break
88            end
89            if string.match(label, tested_domain) then
90
91              if pkt:src() == resolver_ip_addr and srv_addresses[pkt:dst()] then
92                init_time = tonumber(pkt.ts.sec) * 1000000000 + tonumber(
                      ↪ pkt.ts.nsec)
93                queries_by_id[dns.id] = init_time
94
95
96
97              end
98            end
99
100         end
101         n = n - 1
102       end
103       --print("\n")
104     end
105   end
106   if dns.qr == 1 then
107     if dns.questions > 0 and dns:rr_next() == 0 and dns:rr_ok() then
108       if string.match(dns:rr_label(), tested_domain) then
109         if dns.aa == 1 then
110           --print("response_id ", dns.id)
111           if queries_by_id[dns.id] then
112             queryNumber = queryNumber + 1
113             latency = tonumber(pkt.ts.sec) * 1000000000 + tonumber(pkt.ts.nsec
                    ↪ ) - queries_by_id[dns.id]
114                                 --print(latency / 1000000000)
115             queries_by_id[dns.id] = nil
116             table.insert(query_info, {
117               latency = latency,
118               authoritative = pkt:src(),
119               authoritative_port = pkt.dport,
120               number = queryNumber
121             })
```

```
          end
        end
      end
    end
    --dns:print()
  end
end
)
input = require("dnsjit.input.pcapthread").new()

print("analyzing file " .. pcap)
input:open_offline(pcap)
input:receiver(output)
input:run()


local starting_query_time = 0


print("results/".. resolver_used .. "/"..filename .. ".gnuplot")
local gnuplotFile = io.open("results/".. resolver_used .. "/"..filename .. "
    ↪ .gnuplot", "w")
io.output(gnuplotFile)

io.write("set terminal png size 1920,1080")
io.write("\nset samples ",queryNumber)
io.write("\nset size 0.9,1")
io.write("\nset key off")
io.write("\nset output \"".. filename .. ".png\"")
io.write("\nset title \"".. filename .. "\"")
io.write("\nset ylabel \"Latency (ms)\"")
io.write("\nset xlabel \"Time (s)\"")
io.write("\nset yrange [0:",(max_latency_allowed*1.1)/1000000,"]")
io.write("\nset xrange [0:", queryNumber*1.005, "]")
io.write("\nset label 1 '192.168.5.141' at graph 1.005,0.95 font \"Arial,20\" "
    ↪ tc rgb \"red\" ")
io.write("\nset label 2 '192.168.5.142' at graph 1.005,0.91 font \"Arial,20\" "
    ↪ tc rgb \"green\"")
io.write("\nset label 3 '192.168.5.143' at graph 1.005,0.87 font \"Arial,20\" "
    ↪ tc rgb \"blue\"")
io.write("\nplot '" .. filename .. ".dat' using 1:2:3 title \"" .. filename ..
    ↪ "\" with points lc variable pointtype 3")
io.close(gnuplotFile)



local gnuplot_table = {}

for _ , q in pairs(query_info) do
  latency = q.latency / 1000000
```

```
167   gnuplot_table[q.number] = latency
168 end
169
170 local gnuplot_keys = {}
171 for k in pairs(gnuplot_table) do
172   table.insert(gnuplot_keys,k)
173 end
174
175 table.sort(gnuplot_keys)
176
177 local datFile = io.open("results/".. resolver_used .. "/"..filename .. ".dat",
    ↪  "w")
178 io.output(datFile)
179
180 query_start_time_secs = -1
181 for _, k in ipairs(gnuplot_keys) do
182   latency = gnuplot_table[k]
183   for _, q in ipairs(query_info) do
184     if k == q.number then
185       serverReached = string.sub(q.authoritative,13)
186       io.write(k, "␣" , latency, "␣", serverReached, "\n")
187     end
188   end
189 end
190 io.close(datFile)
191
192
193 print(queryNumber)
```

II.3 Client POV

```
1  #! / usr / bin / env dnsjit
2  local client_ip = arg[2]
3  local tested_domain = arg[3]
4
5  local pcaps = {}
6  local index = 4
7  while arg[index] do
8    table.insert(pcaps,arg[index])
9    index = index + 1
10 end
11
12
13
14 if pcaps == nil then
15   print("usage:␣"..arg[1].."␣<pcap>")
16   return
17 end
```

```
18
19 require("dnsjit.core.object")
20 require("dnsjit.core.object.packet")
21
22 output = require("dnsjit.filter.coro").new()
23
24 local queries_by_id = {}
25
26 local query_info = {}
27
28 local lowest_time = 9999999999999999
29 local total_latency = 0
30 local total_count = 0
31
32
33 output:func(function(filter, obj)
34   --print("entered method")
35   local pkt = obj:cast()
36   local dns
37   if pkt:type() == "packet" then
38     dns = require("dnsjit.core.object.dns").new(obj)
39     if dns:parse() ~= 0 then
40       return
41     end
42   elseif pkt:type() == "dns" then
43     dns = pkt
44     if dns:parse() ~= 0 then
45       return
46     end
47     pkt = dns:prev()
48     while pkt ~= nil do
49       if pkt:type() == "packet" then
50         pkt = pkt:cast()
51         break
52       end
53       pkt = pkt:prev()
54     end
55     if pkt == nil then
56       return
57     end
58   else
59     return
60   end
61   if dns then
62     if dns.qr == 0 then
63       local n = dns.questions
64
65       while n > 0 and dns:rr_next() == 0 do
66         if dns:rr_ok() == 1 then
67           label = dns:rr_label()
```

```
68            --print("label")
69
70
71            --print("query_id ", dns.id)
72            if pkt:src() == client_ip then
73              init_time = tonumber(pkt.ts.sec) * 1000000000 + tonumber(
                    ↪ pkt.ts.nsec)
74              queries_by_id[dns.id] = init_time
75            end
76            --dns:print()
77
78          end
79          n = n - 1
80        end
81        --print("\n")
82      end
83    end
84    if dns.qr == 1 then
85      if dns.questions > 0 and dns:rr_next() == 0 and dns:rr_ok() then
86        if string.match(dns:rr_label(), tested_domain) then
87
88          if queries_by_id[dns.id]  and pkt:dst() == client_ip then
89            latency = tonumber(pkt.ts.sec) * 1000000000 + tonumber(pkt.ts.nsec) -
                  ↪  queries_by_id[dns.id]
90            if latency < lowest_time then
91              lowest_time = latency
92            end
93            --print(latency / 1000000000)
94            total_latency = total_latency + latency
95            total_count = total_count + 1
96            queries_by_id[dns.id] = nil
97            table.insert(query_info, {
98              latency = latency,
99              query = dns:rr_label()
100           })
101         end
102       end
103     end
104     --dns:print()
105   end
106 end
107 )
108 input = require("dnsjit.input.pcapthread").new()
109
110 for _,pcap in pairs(pcaps) do
111   input:open_offline(pcap)
112   input:receiver(output)
113   input:run()
114 end
115
```

```
print(total_count)
print("Average Latency: ", (total_latency/ 1000000) / total_count)
```

II.4 Generate Tests

```
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Random;

public class Main {

  static class Auth {

    private String execAddress;
    private String controlAddress;
    private int delayMS;
    private float packetLoss;
    private int intervalDown;

    public Auth(String exec, String control, int delay, float packetLoss, int
        ↪ intervalDown) {
      this.execAddress = exec;
      this.controlAddress = control;
      this.delayMS = delay;
      this.packetLoss = packetLoss;
      this.intervalDown = intervalDown;
    }

    public String getExec() {
      return this.execAddress;
    }

    public String getControl() {
      return this.controlAddress;
    }

    public int getDelay() {
      return this.delayMS;
    }

    public float getPacketLoss() {
```

```
43      return this.packetLoss;
44    }
45
46    public int getIntervalDown() {
47      return this.intervalDown;
48    }
49  }
50
51  private static final int SETUPSLEEPSECONDS = 1;
52  private static final String DNSJIT = "./dnsjit/src/dnsjit"; // location of
        ↪ dnsjit
53  private static final String JITSCRIPT = "jitScripts/analyse_match.lua"; //
        ↪ location of dnsjit script to analyse results
54  private static final String JITGRAPH = "jitScripts/draw_graph.lua"; //
        ↪ location of dnsjit script to draw the graph
55  private static final String PCAPFOLDER = "pcaps/"; // folder with the pcaps
56  private static final String TEST_FOLDER = "Test_folder/"; // folder where
        ↪ the tests should be located
57
58  public static void main(String[] args) throws IOException {
59    // List<String> remainingArgs = new ArrayList<String>();
60
61    String queryTrace = "#!/bin/bash\n";
62    String tcpdump_opts = "";
63    String tcpdump_filename = "";
64    String resolver = "";
65    String domain = "";
66    String resolverType = "";
67    String query_filename = "";
68    String results_filename = "";
69    String ttl_used = "";
70    String nextScript = "";
71    String serviceRestart = "";
72    String sniffer = "sniffer";
73    float queryInterval = 0;
74    int testRuntime = 0;
75    List<Auth> servers = new ArrayList<Auth>();
76
77    //
78    String usage = "******␣ssh␣needs␣to␣be␣previously␣established␣(using␣ssh-
        ↪ copy-id␣-i␣root@server)␣******\n"
79        + "java␣Main\n" + "␣\"tcpdump_options\"␣.........#1\n" + "␣␣
            ↪ test_number␣..............#2\n"
80        + "␣resolver␣.................#3\n" + "␣␣domain␣
            ↪ ..................#4\n"
81        + "␣resolverType␣.............#5\n" + "␣␣query_filename␣
            ↪ ............#6\n"
82        + "␣results_filename␣.........#7\n" + "␣␣ttl_used␣
            ↪ ................#8\n"
```

```
83          + "␣nextScript␣.................#9\n" + "␣␣queryInterval␣
                ↪ .............#10\n"
84          + "␣testRuntime(seconds)␣.......#11\n"
85          + "␣\"server_exec_addr␣server_control_addr␣server_delay␣
                ↪ server_packetloss␣server_interval_down\"␣.....#12\n"
86          + "can␣have␣multiple␣servers,␣just␣have␣multiple␣entries␣of␣#12";
87      if (args.length < 12) {
88        System.out.println("Usage:\n" + usage);
89        return;
90      }
91
92      tcpdump_opts = args[0];
93      tcpdump_filename += args[1].contains(".pcap") ? args[1] : args[1] + ".pcap
            ↪ ";
94      resolver = args[2];
95      domain = args[3];
96      resolverType = args[4];
97      query_filename = args[5] + ".sh";
98      results_filename = args[6];
99      ttl_used = args[7];
100     nextScript = args[8] + ".sh";
101     queryInterval = Float.parseFloat(args[9]);
102     testRuntime = Integer.parseInt(args[10]);
103
104     for (int i = 11; i < args.length; i++) {
105       String[] server = args[i].split("␣");
106       // exec_ip, control_ip, delay, packetloss, timeDownMins
107       Auth a = new Auth(server[0], server[1], Integer.parseInt(server[2]),
              ↪ Float.parseFloat(server[3]),
108           Integer.parseInt(server[4]) * 60);
109       servers.add(a);
110     }
111
112     switch (resolverType) {
113     case "BIND":
114       serviceRestart = "bind9";
115       break;
116     case "POWERDNS":
117       serviceRestart = "pdns-recursor";
118       break;
119     case "UNBOUND":
120       serviceRestart = "unbound";
121       break;
122     }
123     switch(resolver){
124     case "WINDOWS12": queryTrace += "python␣win12_clear_cache.py\n"; break;
125     //case "WINDOWS16": queryTrace += "python win16_clear_cache.py\n"; break;
126     default: queryTrace += "ssh␣root@" + resolver + "␣'service␣" +
              ↪ serviceRestart + "␣restart'\n"; break;
127     }
```

```
128
129     // add delays
130     for (Auth a : servers) {
131       int delayMS = a.getDelay();
132       String controlIP = a.getControl();
133       float packetLoss = a.getPacketLoss();
134       queryTrace += "\nssh␣root@" + controlIP + "␣'tc␣qdisc␣del␣dev␣eth0␣root␣
              ↪ handle␣1:1␣netem'␣\n"
135           + "echo␣\"Deleted␣previous␣qdisc␣on␣" + controlIP + "\"␣\n";
136       queryTrace += "sleep␣" + SETUPSLEEPSECONDS + "\n";
137       queryTrace += "ssh␣root@" + controlIP + "␣'tc␣qdisc␣add␣dev␣eth0␣root␣
              ↪ handle␣1:1␣netem␣delay␣" + delayMS
138           + "000␣0␣loss␣" + packetLoss + "%'␣" + "\necho␣\"Added␣new␣qdisc␣on␣
                  ↪ " + controlIP + "␣with␣"
139           + delayMS + "ms␣delay␣and␣" + packetLoss + "%␣packetloss␣\"\n";
140       // queryTrace += "ssh root@" + controlIP + " 'tc qdisc change dev
141       // eth0 root netem loss " + packetLoss + "%'\n";
142     }
143
144     // open tcpdump
145     queryTrace += "\nssh␣root@" + sniffer + "␣'nohup␣tcpdump␣" + tcpdump_opts
            ↪ + "␣-w␣" + tcpdump_filename
146           + "␣>/dev/null␣2>&1␣&'\n";
147
148     // run queries
149     queryTrace += "\n" + createQueries(domain, queryInterval, testRuntime,
            ↪ servers, resolver) + "\n";
            ↪
150
151     // queries executed, close tcpdump
152     queryTrace += "ssh␣root@" + sniffer + "␣'pkill␣tcpdump'\n";
153
154     // make directory to store results
155     queryTrace += "\nmkdir␣-p␣results/" + resolver;
156
157     // make directory to store pcaps
158     queryTrace += "\nmkdir␣-p␣pcaps/" + resolver;
159
160     // copy file
161     queryTrace += "\nscp␣root@" + sniffer + ":/root/" + tcpdump_filename + "␣
            ↪ pcaps/"+resolver+"/" + tcpdump_filename + "\n";
162
163     // analyse queries
164     for (Auth a : servers) {
165       String exec = a.getExec();
166       queryTrace += exec + "=$(getent␣hosts␣" + exec + "␣|␣awk␣'{print␣$1}')\n
              ↪ " + "echo␣$" + exec + "\n";
167     }
168     queryTrace += DNSJIT + "␣" + JITSCRIPT + "␣" + PCAPFOLDER + resolver + "/"
            ↪ + tcpdump_filename + "␣" + domain + "␣" + testRuntime
```

```
169         + "␣" + queryInterval + "␣" + results_filename + "␣" + resolverType +
                ↪ "␣" + "␣" + ttl_used + "␣";
170
171     for (Auth a : servers) {
172
173       queryTrace += "$" + a.getExec() + "␣" + a.getDelay() + "␣" + a.
              ↪ getPacketLoss() + "␣" + a.getIntervalDown()
174           + "␣";
175     }
176
177     // if(results_filename.equals("results11")){
178     queryTrace += "\n" + resolver + "=$(getent␣hosts␣" + resolver + "␣|␣awk␣'{
            ↪ print␣$1}')\n" + "echo␣$" + resolver
179         + "\n";
180     queryTrace += DNSJIT + "␣" +JITGRAPH + "␣" + PCAPFOLDER + resolver + "/"+
            ↪ tcpdump_filename + "␣" + domain + "␣"
181         + testRuntime + "␣" + queryInterval + "␣" + results_filename + "␣" +
                ↪ resolverType + "␣" + "␣$"
182         + resolver + "␣";
183     for (Auth a : servers) {
184
185       queryTrace += "$" + a.getExec() + "␣" + a.getDelay() + "␣" + a.
              ↪ getPacketLoss() + "␣" + a.getIntervalDown()
186           + "␣";
187     }
188
189     queryTrace += "\ncd␣results/"+resolver;
190     queryTrace += "\ngnuplot␣" + results_filename + ".gnuplot";
191     /*for (Auth a : servers) {
192       queryTrace += "\ngnuplot $" + a.getExec() + ".gnuplot";
193     }*/
194     queryTrace += "\ncd";
195
196     // }
197
198     if (!nextScript.equals(query_filename)) {
199       System.out.println("Different!");
200       queryTrace += "\nbash␣" + TEST_FOLDER  + nextScript;
201
202     }
203
204     System.out.println(queryTrace);
205
206     try (Writer writer = new BufferedWriter(
207         new OutputStreamWriter(new FileOutputStream(query_filename), "utf-8"))
                ↪ ) {
208       writer.write(queryTrace);
209     } catch (IOException e) {
210       e.printStackTrace();
211     }
```

```
212   }
213
214   private static String createQueries(String domain, float queryInterval, int
          ↪ testRuntime, List<Auth> servers, String resolver) {
215     String result = "echo␣\"Starting␣queries\"␣\n{\n"
216         + resolver + "IP=$(getent␣hosts␣" + resolver + "␣|␣awk␣'{print␣$1}')\n
                ↪ " + "echo␣$" + resolver + "IP\n";
217
218     String stdDig = "\ndig␣@$"+resolver+"IP␣␣+cdflag␣" + domain + "␣&&␣sleep␣"
            ↪  + queryInterval + "\n";
219     Map<Auth, Integer> execDowntimeCount = new HashMap<Auth, Integer>();
220     for (Auth a : servers) {
221       execDowntimeCount.put(a, a.getIntervalDown());
222     }
223     float i = 0;
224     while (i < testRuntime) {
225       // System.out.println("adding query");
226       if (queryInterval < 0) {
227         double interval = myRandom(0, 1);
228         result += "dig␣+cdflag␣" + domain + "␣&&␣sleep␣" + interval + "\n";
229         i += interval;
230       } else {
231         result += stdDig;
232         i += queryInterval;
233       }
234
235       for (Auth a : execDowntimeCount.keySet()) {
236         if (a.getIntervalDown() != 0) {
237           if (i >= execDowntimeCount.get(a)) {
238             int timesExecuted = execDowntimeCount.get(a) / a.getIntervalDown()
                    ↪ ;
239             result += "}␣&>␣/dev/null␣\n";
240             if (timesExecuted % 2 == 0) {
241               result += "echo␣\"Turning␣server␣" + a.getExec() + "␣back␣up\"\
                      ↪ nssh␣root@" + a.getControl()
242                   + "␣'tc␣qdisc␣change␣dev␣eth0␣root␣netem" + "␣delay␣" + a.
                          ↪ getDelay()
243                   + "000␣0␣loss␣0%'\n";
244             } else {
245               result += "echo␣\"Shutting␣server␣" + a.getExec() + "␣down\"\
                      ↪ nssh␣root@" + a.getControl()
246                   + "␣'tc␣qdisc␣change␣dev␣eth0␣root␣netem␣" + "␣delay␣" + a.
                          ↪ getDelay()
247                   + "000␣0␣loss␣100%'\n";
248             }
249             result += "{\n";
250             execDowntimeCount.put(a, execDowntimeCount.get(a) + a.
                    ↪ getIntervalDown());
251           }
252         }
```

```
253       }
254     }
255     result += "\necho␣\"this␣needs␣to␣be␣here␣in␣case␣this␣final␣part␣has␣no␣
            ↪ commands\"\n}␣&>␣/dev/null␣\n";
256     for (Auth a : servers) {
257       result += "echo␣\"Resetting␣server␣" + a.getExec() + "␣packetloss␣to␣
              ↪ zero\"\n";
258     }
259     return result;
260   }
261
262   private static double myRandom(double min, double max) {
263     Random r = new Random();
264     return (r.nextInt((int) ((max - min) * 10 + 1)) + min * 10) / 10.0;
265   }
266
267 }
```