Design of a Reactive DNS Measurement System

Jochem Braakhuis
13-07-2021, University of Twente
j.a.braakhuis@student.utwente.nl

***Abstract*—For speed, robustness and safety of the Internet, it is important that information on name servers in every level of the hierarchical DNS structure is accurate and consistent. OpenINTEL, designed in 2016, measures about half of the global namespace every day, but lacks flexibility in reactive measurement. To provide an easily scalable framework for performing research in DNS inconsistencies, we designed and tested a reactive DNS measurement system compatible with OpenINTEL, supported by Apache Kafka message broker functionality. The software can query multiple levels in the DNS hierarchy and supports 10 query types. Initial high-volume testing of the resolver authoritative level shows the software is able to handle large volumes of queries, but only gives a lower bound on the performance of the software. Testing of more authoritative levels, testing on purpose-built hardware and implementation of extended decoder logic and performance optimisation is required to successfully deploy the software.**

I. Introduction

The Domain Name Service (DNS) is a very important part of the modern Internet, providing a service that can translate human readable domain names into the IP address of their server. DNS uses a hierarchical structure, starting at the Root. This hierarchical structure of DNS can be seen in figure 1.

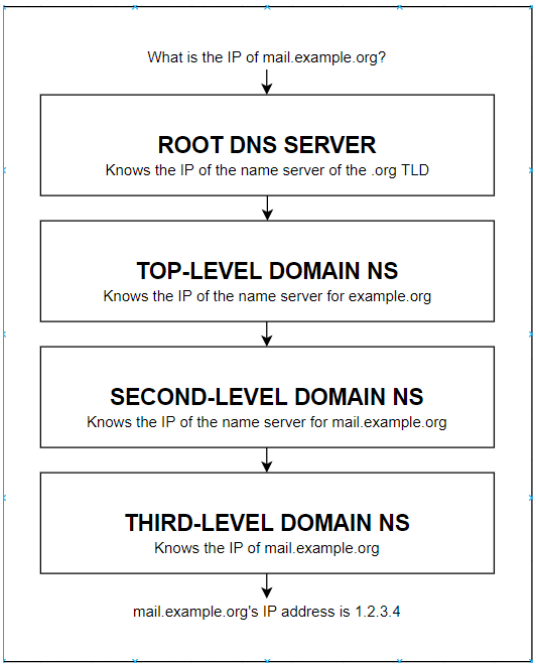

Fig. 1. The Hierarchy of DNS

All DNS queries, provided the resource record for that query is not cached, go to the Root servers first. The Root will then delegate the query to the appropriate top-level domain name server dependent on the entered domain (.com, .org, .nl). Then, the top-level domain name server will in turn delegate the query to the authoritative name server for the entered domain, which finally returns the IP address of the domain's server to the user, or delegate it to authoritative name servers for further sub-domains. Section II further elaborates on this process.

In addition to storing the IP addresses of name servers and domains, name servers also contain other records about domains in their zone. For example, MX records contain the domain names of the servers that handle email traffic for the domain, and their priority. A lot of records are related to security. DNSSEC is a system that was implemented to verify the integrity of DNS traffic and records, and the DNSKEY record for example contains the public key that can be used to verify signatures. [1] In total, there are over 45 different records that can be stored on a name server [2].

Figure 2 shows the response of a query for the IPv4 address of utwente.nl. We provide a more detailed description of the form of a DNS resource record and the DNS response in section II.

```
; <<>> DiG 9.16.16 <<>> a utwente.nl
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 43934
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;utwente.nl.                    IN      A

;; ANSWER SECTION:
utwente.nl.             3600    IN      A       130.89.3.249

;; Query time: 165 msec
;; SERVER: 84.116.46.22#53(84.116.46.22)
;; WHEN: Tue Jul 13 11:10:17 W. Europe Daylight Time 2021
;; MSG SIZE  rcvd: 55
```

Fig. 2. DNS Response for an A Query of utwente.nl

To make sure that the Internet remains fast and secure, it is important to monitor DNS name servers. Since DNS is such an important link between users of the Internet and the domains they want to visit, impact of failure of this system can be huge. If a name server contains inaccurate information, it is possible the IP address of a domain becomes unresolvable, or hackers can take control of a misconfigured name server and lead users to the wrong domain. Research into the effects and prevalence of inconsistency between name servers in different levels of the hierarchy has shown that this is a relevant problem, and while this research was in part performed by retrieving information

from different levels of the DNS hierarchy, and retrieving and comparing information form multiple name servers of the same domain, the measurements were performed in a non-scalable way. To perform these types of research more easily, a scalable and flexible reactive DNS measurement system would be ideal. Reactive measurements in this sense are DNS measurements performed on demand as a reaction to a trigger. Triggers can be for example the registration of a new domain, the issuing of a new certificate of a domain, etc.

Multiple large scale active DNS measurement systems have already been developed. For example in 2016, researchers at Georgia University of Technology developed such a system [3], just like researchers at the University of Vienna in 2015 [4], the latter of which focused on the security of email systems. Another one of these systems, and the system that we will be focusing on most in this paper, is the system developed by researchers at the University of Twente in 2016 [6]. This system is called OpenINTEL, and it measures around 50% of the Internet's Second Level Domain DNS records every day. OpenINTEL provides the ability and scalability to perform this task, however it has certain limitations. Mainly, the system lacks flexibility in being able to perform diverse reactive measurements. OpenINTEL takes a bulk of measurement requests for second level domains every day, and it requests the same set of predefined queries every day. It lacks any possibility of reacting to triggers or performing on-demand measurements.

The main objective of this thesis is to design a system that can be used in addition to OpenINTEL, sharing the same data formats and general setup, but that is more flexible in the types of queries it can perform and is able to query multiple levels in the DNS hierarchy. The system should be dynamically scalable and have a performance speed in the same order as OpenINTEL.

In order for our system to maintain compatibility with OpenINTEL, it will use the same data structure. The reasoning behind maintaining compatibility with this data structure is to allow for the reuse of data analysis code developed for previous works on OpenINTEL. Section II contains more information about the OpenINTEL data format.

To accomplish this design, several research questions can be formulated. The following questions will be discussed in this report:

1) How can the system query different hierarchical levels?
2) How can a DNS response be converted to the desired data format?
3) How can dynamic scalability be achieved?
4) Can the performance in terms of measurements per second be compared to OpenINTEL?

Structure of the Thesis

Section II lists important background concepts for the design of our software, in section III we list existing work related to the objective of this thesis, and start to form design objectives. Section V list the design objectives in a concise manner, so they can be referred back to. In section V we list and motivate design decisions, and in section VI the actual implementation of the design is described. Section VII describes the setup used to perform high-volume testing of the software, the results of which are listed in section VIII. Lastly, section IX contains the general conclusion, and recommendations for future work.

II. Background

DNS Hierarchy Levels

In this thesis, we will frequently discuss the different levels of DNS hierarchy. In this subsection we will elaborate on the distinction between a so-called "parent" name server and a "child" name server. We define the "child" name server as an authoritative name server of a domain. The plural form of this term, i.e. "children", refers to all of the authoritative name servers of a domain. Conversely, "parent" name server refers to the name server of the authoritative level above the domain. For example, the parent of ewi.utwente.nl is utwente.nl, so the parent name server is the authoritative name server of utwente.nl. When the domain is a second-level domain, such as utwente.nl, its parent is the top level domain, and the parent name server is this TLD's name server.

The DNS Resolver

The process by which a domain name is converted to an IP address is called resolution of a domain name, and this function, as well as the lookup of other DNS records, is provided by a recursive resolver. A resolver takes a DNS query and returns the answer, going through the hierarchical levels as shown in figure 1. Web browsers mostly use resolver functionality provided by the ISP, but there are also public DNS resolvers, such as the Google Public Resolver and the Cloudflare DNS Resolver. It is also possible to install a recursive DNS resolver on local hardware, we call this a "local resolver".

DNS Response Format

The format of a DNS resource record is described in RFC1035 [5]. This format is shared between the answer, authority and additional sections. Table I shows the fields in a resource records, with a short description of what the field contains.

TABLE I. The format of a DNS resource record

Field	Description
NAME	The domain name the record belongs to
TYPE	The record type, e.g. A, NS, MX etc.
CLASS	The class of data in the record
TTL	The time that the record may be cached
RDLENGTH	The length in octets of the RDATA field
RDATA	This contains the actual information contained in the record, e.g. the domain's IP address, its name servers, or its mail servers

When a name server is queried, it will return the appropriate resource record. It is important to note here that the data contained within the record is stored in just 1 field, RDATA. This means we have to implement some sort of decoder logic in our software to convert this single field into information that can be stored in the fields of our output data format.

Figure 3 shows the DNS response for an MX query of utwente.nl. This response contains in order the NAME, TTL,

CLASS, TYPE and RDATA fields from the resource record. In this case, the RDATA field contains a mail server priority and domain.

```
; <<>> DiG 9.16.16 <<>> mx utwente.nl
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 30152
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;utwente.nl.                    IN      MX

;; ANSWER SECTION:
utwente.nl.             3600    IN      MX      10 mx1.surfmailfilter.nl.
utwente.nl.             3600    IN      MX      10 mx2.surfmailfilter.nl.

;; Query time: 25 msec
;; SERVER: 84.116.46.22#53(84.116.46.22)
;; WHEN: Fri Jul 09 00:50:25 W. Europe Daylight Time 2021
;; MSG SIZE  rcvd: 94
```

Fig. 3. DNS Response for an MX Query of utwente.nl

DNS responses can be different when the query is directed at different levels in the hierarchy. For example, figure 4 shows the response from the resolver for a query for the name servers of utwente.nl. The domains of the name servers appear in the ANSWER section of the response. However, if we direct the same query directly at the IP address of the name server of the .nl TLD (the parent name server of utwente.nl), we observe a difference in the response. Figure 5 shows the response from the TLD name server. In this case, we find the domain names in the AUTHORITY section instead of the ANSWER section, and their IPv4 and IPv6 addresses appear in the ADDITIONAL section. Our system must be able to include this information obtained from different hierarchical levels into its results, and thus must be set up to retrieve information from the AUTHORITY and ADDITIONAL sections of a DNS response.

```
; <<>> DiG 9.16.16 <<>> ns utwente.nl
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 26210
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;utwente.nl.                    IN      NS

;; ANSWER SECTION:
utwente.nl.             3600    IN      NS      ns3.utwente.nl.
utwente.nl.             3600    IN      NS      ns2.utwente.nl.
utwente.nl.             3600    IN      NS      ns1.utwente.nl.

;; Query time: 25 msec
;; SERVER: 84.116.46.22#53(84.116.46.22)
;; WHEN: Fri Jul 09 00:52:37 W. Europe Daylight Time 2021
;; MSG SIZE  rcvd: 93
```

Fig. 4. Resolver Response for an NS Query of utwente.nl

```
; <<>> DiG 9.16.16 <<>> utwente.nl ns @194.0.28.53
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2247
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 3, ADDITIONAL: 6
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
;; QUESTION SECTION:
;utwente.nl.                    IN      NS

;; AUTHORITY SECTION:
utwente.nl.             3600    IN      NS      ns1.utwente.nl.
utwente.nl.             3600    IN      NS      ns2.utwente.nl.
utwente.nl.             3600    IN      NS      ns3.utwente.nl.

;; ADDITIONAL SECTION:
ns1.utwente.nl.         3600    IN      A       130.89.1.2
ns1.utwente.nl.         3600    IN      AAAA    2001:67c:2564:a102::3:1
ns2.utwente.nl.         3600    IN      A       130.89.1.3
ns2.utwente.nl.         3600    IN      AAAA    2001:67c:2564:a102::3:2
ns3.utwente.nl.         3600    IN      A       131.155.0.37

;; Query time: 13 msec
;; SERVER: 194.0.28.53#53(194.0.28.53)
;; WHEN: Fri Jul 09 00:52:24 W. Europe Daylight Time 2021
;; MSG SIZE  rcvd: 197
```

Fig. 5. TLD Name Server Response for an NS Query of utwente.nl

Dynamic Scalability Through Message Broker Technology

In general, a message broker system is used to link an information producing source to an information receiver, while performing some type of action on the information. This can range from simply routing data to converting data formats or pre-processing information. Our software needs to be dynamically scalable, i.e. it needs to be able to handle worker machines being added or removed, on its own. For this reason we designed an architecture based on message-broker functionality, with the main interest being the function of a message broker to divide workload over multiple workers. In particular, we choose Apache Kafka. Kafka was developed in 2011 by Kreps et al. [11] and was originally used for log data analysis. With Kafka, producers can publish messages to topics, which are stored on servers called the brokers. Consumer groups can then subscribe to these topics and receive the messages the produces publishes in them. The tracking of which messages were already received is done client-side, so there is no extra load on the brokers and the client can more easily re-request messages it had already received by simply changing the requested offset. Consumer groups consume the messages of a subscribed topic as a group, the messages are only received by one consumer in the group, so no duplicate messages will be in a consumer group. Kafka uses a consensus service called Zookeeper to keep track of changes in the brokers and clients, and is able to re-balance the consumer group if necessary, so messages are evenly distributed among consumers. Consumers can in this way process data provided by the producers in an efficient manner. Zookeeper is also the way in which the dynamic scalability is achieved; whenever consumers are added or removed from a consumer group, Zookeeper handles the change.

Data Formats

The data format that we use for the input of the system is the JSON file format. JSON makes it a straightforward process to assign values to our input parameters, since it works with attribute-value pairs. For example, ”domain”:”example.com”

is a line that sets the domain to measure. Together with lines setting the other relevant parameters required for the requested measurement, it forms a complete description of the measurement, readable by our software. One JSON can contain multiple of these measurement request, which makes it easily scalable.

The data format that we use for the output of the system is the Apache AVRO file format. This is the same format used by OpenINTEL, and we describe it in more detail in section III.

III. Related Work

OpenINTEL

In 2016 van Rijswijk-Deij et al. [6] designed a system for large scale DNS measurements called OpenINTEL. This system was designed to generate large quantities of DNS data that could be used for follow-up measurements. OpenINTEL is able to perform measurements on the largest top-level domain, .com, spanning 123 million domains, but also smaller TLDs. It measures each domain once per day, and stores at least a year's worth of data. The final design of the system uses off-the-shelf DNS measurement software to query the .com, .net and .org TLDs, which when put together contain about half of all second level domains on the Internet. For each name, 14 records are requested, including the IPv4 and if applicable IPv6 address, names of authoritative name server for the domain, and the names of the hosts that handle the domain's e-mail. To ensure speed, a worker cluster is assigned to each TLD to make sure queries that fail or take a long time to complete do not impact the progress of other queries. The data is then analysed using a Hadoop cluster and the Apache Impala engine.

Downsides of OpenINTEL: While this system is capable of reliably measuring huge amounts of data, it has its downsides. Mainly, it lacks flexibility in performing reactive measurements. OpenINTEL performs the same measurements on the same domains every time, and cannot easily be used to perform diverse and specific DNS queries for each domain. Another limitation is that the system is unable to directly query name servers in different authoritative levels. Instead, it uses a local resolver to get the answer to the queries. The last relevant limitation of OpenINTEL is that the system is unable to query multiple name servers of a single domain, it will just use the name server the resolver points it to. Our system will aim to fill these gaps in the OpenINTEL capabilities, while maintaining compatibility with its data formats. It will also use the same general structure and be dynamically scalable, running the software on worker VMs alongside a local resolver.

The OpenINTEL Data Format: The OpenINTEL data structure contains over 100 fields and we will not include it in its entirety in this thesis, but can be seen in the OpenINTEL data dictionary [7]. The output format is in Apache AVRO format, which is a format where data is sorted by rows. Every relevant field has a column, for example there are columns for the domain that was queried, the query type, and for every possible field in a DNS response. For example, there is a field for the MX address, and another for the MX preference. Since a DNS response contains the answer in one line instead of in ordered fields, we will need to implement decoder logic to convert the DNS response into the OpenINTEL format.

OpenINTEL Performance: To define a desired performance for our system, it is important to determine the performance of OpenINTEL. From the paper, it can be determined that the .com TLD containing 123M domains was processed in 17 hours and 10 minutes (1030 minutes), by 80 worker VMs. This comes down to about 1500 domains per VM per minute. Since each domain is queried with 14 different query types, we multiply 1500 by 14 to get 21000 queries per minute, or 350 queries per second. This is the number we will be comparing the performance of our software to.

The rest of this section will explain the reasoning behind wanting to expand the capabilities of OpenINTEL, and the design challenges this brings with it.

Same-Level Consistency

Usually domains will use multiple name servers for stability, redundancy, and to ensure a DNS query can be resolved quickly, no matter the location of the user. In 2017, Müller et al. [8] investigated how a recursive DNS resolver selects an authoritative name server for its query. A test domain was set up with 7 authoritative name servers all over the world. A DNS lookup for that test domain was then performed by 9700 Ripe ATLAS probes worldwide to see the behaviour of the local configured recursive resolver of these points. Their results show that recursive resolvers prefer to query authoritative name servers with the lowest round-trip time, also showing that this preference was stronger the closer these name servers were located to the user. Since for a recursive resolver only the round trip time of a query is relevant when selecting an authoritative name server, it is important that the information present on these name servers is consistent and accurate. This shows the need to monitor DNS information from different name servers of the same domain, and while it may not be possible for our system to determine the accuracy of such information, it should be able to determine consistency between name servers of the same authoritative level.

Parent-Child Consistency

In 2020 Sommese et al. [9] investigated consistency of DNS records between parent and child zones of the DNS, in their case the top-level domain and the second-level domain. Using measurement data of 165 million domains gathered by OpenINTEL, which will be discussed later in this section, DNS records of parent and child name servers were compared, and it was found that while just under 80% of the pairs were consistent, 8% or about 13 million domains were not.

To determine the consequences of these findings, measurements were performed using a test domain and two sets of authoritative name servers, one defined in the parent, the other in the child. The name server sets of the parent and the child are either completely disjoint, one is a subset of the other, or they have nonzero overlap but are not subsets. Both have the name servers pointed to by the child in the so-called "authority section". Resolvers may prefer this information over the data provided by the parent.

In all cases, the results showed that when normal responses were given some resolvers preferred the information in the authority section over the information provided by the parent name server, reaching the authoritative name servers pointed

to by the child name server, but when minimal responses were given only the authoritative servers in the parent set were reached. When parent and child servers are not consistent, name server capacity can go unused, other servers may have higher load than necessary, and in worse cases can cause so-called ”lame delegations” discussed later in this section.

This research shows the need for our system to be able to query different levels in the DNS hierarchy, to determine then inter-level consistency of DNS information. Additionally, it shows the advantage of being able to include additional sections of a DNS result in the output of the system, instead of just the answer section. This is another point where our system improves on OpenINTEL.

“Lame Delegation”

A “lame delegation” occurs when a name server is delegated authority for a certain domain but cannot provide authoritative answers to queries about that domain, for example because it does not know the relevant IP address. Research from 2020 by Akiwate et. al. [10] shows that this is a relevant issue. Their research consisted of 2 stages; in the first stage data was used from the zone files from the dns.coffee service. This includes over 499 million domains and nearly 20 million name servers. Static analysis was performed to get a general idea of the prevalence of lame delegations, which found a 4% unresolvability rate. However, this static analysis only gives a lower bound on the prevalence. In the second stage of the research this was rectified by performing active measurements on over 49 million randomly selected domains, which found that over 14% of these domains was at least partly lame, which means at least 1 of the authoritative name servers was unable to provide authoritative info, and that 9,5% of the total domains was fully lame, meaning no authoritative name server was able to provide info and the domain was unresolvable. In the best case, a (partly) lame delegation increases DNS query time, since the DNS resolver will time out when the authoritative name server is unable to provide the requested info, after which it will try a new server. In a slightly worse case, a domain cannot be resolved at all. In an even worse case where the authoritative name server does not exits at all, malicious actors can obtain the domain names of these servers and lead unsuspecting users to their own websites, instead of the ones they were trying to reach.

This research shows the need for DNS measurement data to always be as recent as possible, and to prevent caching of data by both our system and the local resolver it uses. It also shows the importance of all the name servers of a domain to function properly.

Apache Kafka

The Apache Kafka message broker has already been used to process measurement data in a project similar to ours, from 2016 by Orsini et al. [12] They designed a framework for the analysis of measurement data for the Border Gateway Protocol. The actual measurements are performed by a process called BGPCorsaro, which publishes messages to appropriate topics containing the measurement data. These are then consumed by applications that process the data.

While the use of Kafka to handle measurements may not be completely novel, this study shows that Kafka is a good candidate for our goals.

IV. Design Objectives

Now that we have gotten a little more insight into how OpenINTEL functions, what the gaps are in its capabilities and why it is important to fill these gaps, we will now list the design objectives. These design objectives are used as a method to provide solutions to the research questions posed in section I.

- The system should be able to retrieve responses for different DNS query types for specified domains. While OpenINTEL is able to retrieve different queries for domains as well, it cannot do specific measurements, rather it does the same measurements every time.
- The system should be able to query different levels in the DNS hierarchy. This is something that OpenINTEL is unable to do, and will be an important improvement.
- The system should be able to include the ADDITIONAL and AUTHORITY sections of the response in its results, rather than just the ANSWER section like OpenINTEL does.
- The system should be used as an extension of OpenINTEL, and use the same data structure to be fully compatible.
- The system, like OpenINTEL, should be able to handle large amounts of measurements, be usable on different platforms and should be able to handle multiple queries in tandem as to not have queries hung up when one of them is taking a longer time than expected.
- The system should be able to run on virtual machines on servers and have performance similar to that of OpenINTEL.

V. Design Decisions

Before designing the implementation of the system, some decisions on the design had to be made first. This is mostly related to the use of certain plugins.

Extension of OpenINTEL Data Format

Since our system is extending the capabilities of OpenINTEL, the output format needs to be slightly extended to accommodate the new functions our system provides. We extended the format with the fields listed in table II.

TABLE II. Extended fields for the OpenINTEL data format

Name	Type	Description
nameserver_used	String	IP address of the name server that answered the query, for the local resolver this will be 127.0.0.1.
section_level	String	Indicates the section level the answer came from, this will be the ANSWER, AUTHORITY or ADDITIONAL section.
error_message	String	This field can be used by the program to indicate an error occuring, such as a timeout.

Since our system is able to query multiple authoritative levels, and able to query multiple name servers for a single domain for comparison, it is important to know what name server answered the request, creating the need for a nameserver_used field. Since OpenINTEL is unable to query different authoritative levels, or different name servers of the same level, it does not need this field.

The section_level field is needed because our system is able to include different sections of the DNS response, so this field is needed to indicate where exactly the response came from. OpenINTEL only processes the ANSWER section, so it does not need this field.

The error_message field is needed to find out if a problem has occured somewhere in the software. OpenINTEL does not use specific error tracking, except for strictly DNS related errors such as NXDOMAIN or SERVFAIL, and thus did not need this field.

For consistency, all of the fields are always present in the output, but only the relevant fields will be filled in. The rest of the fields will simply be Null. This is in line with what OpenINTEL does as well.

Kafka Implementation

We implement the message broker functionality with the Apache Kafka message broker, through a Java plugin called Alpakka [13]. This plugin prevents us from having to write the message broker logic required for Kafka ourselves, and instead allows us to have a much simpler method of producing and consuming messages and topics. With Alpakka, the application can consume messages from the requests topic, and produce messages containing the correctly formatted data in the results topic. Alpakka is based on the reactive processing framework Akka, which allow for asychronous processing. Alternatives for implementation of the message broker functionality, such as Spring or the native Kafka plugin, do not allow for asynchronous processing. Asynchronous processing is desired because it will lower the required number of threads, in turn lowering resource requirements, which increases the number of measurements per second for the same hardware. This solution aims to provide a solution to Research Question 3.

Local Resolver

We will be using a local recursive resolver for all queries not directed at a specific name server. This resolver will be running alongside our software, on the same machine. We will be using the Unbound resolver [14] to provide this functionality, for the simple reason that this is the same way that the worker VMs of OpenINTEL are set up. An important thing to note here is that while recursive resolvers used by regular Internet users will use caching to prevent having to look up recently queried results, we have specifically configured Unbound to not use caching. Since our system needs to be able to detect problems such as misconfiguration or highjacking of a name server as soon as possible, we want to ensure it is always processing the newest possible data. Since caching would prevent this, we have disabled caching in our local resolver.

Kafka Monitoring

In order to monitor the topics and their contents, throughputs, and consumers, we will be using the Lenses Kafka Docker Box [15]. This is a complete container solution providing all the required functionality for testing and development purposes of Kafka based applications. The reason we chose this software to test our system is because it is the simplest way of getting all functionality from a simple source. An alternative would be to manually deploy the entire stack and to write the code for the monitoring functionality by ourselves, but this would require more time and resources.

VI. Internal System Design

In general, we have defined the input and output of the system as AbstractMeasurement and AbstractResult, respectively. These classes are empty by themselves, but can be extended. This way, the program can use the same message-broker architecture to perform different types of measurements. In this thesis, we will be extending AbstractMeasurement with DNSMeasurement, and AbstractResult with DNSRow. These extended classes will contain the parameters and functions to make DNS measurements and format them correctly. The UML in figure 6 shows this extension to AbstractMeasuremnt, as well as a (yet unimplemented) different possible functionality. Figure 7 shows the same for AbstractResult.

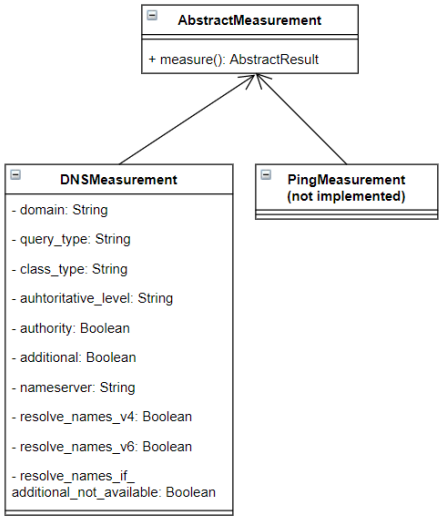

Fig. 6. UML of AbstractMeasurement

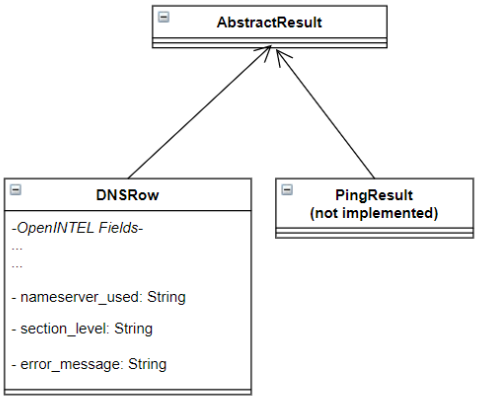

Fig. 7. UML of AbstractResult

The DNSMeasurement class defines the parameters for the measurement, which we describe in more detail in table III.

TABLE III. THE FIELDS IN THE DNSMEASUREMENT CLASS

Parameter	Type	Description
domain	String	The domain to measure
query_type	String	The query type to request
class_type	String	The DNS class to be requested, this will be left to IN (Internet) for our purposes
authoritative_level	String	The level in the DNS hierarchy the request is directed to
authority	Boolean	Includes the authority section records in the results
additional	Boolean	Includes the additional section records in the results
nameserver	String	Uses a specific name server for the request, can be left blank
resolve_names_v4	Boolean	Resolves the IPv4 address of the name server or mailserver, if the query type is NS or MX
resolve_names_v6	Boolean	Resolves the IPv6 address of the name server or mailserver, if the query type is NS or MX
resolve_names_if_additional_not_available	Boolean	Resolves the IP address of the name server or mailserver if the name does not appear in the additional section

Besides these parameters, DNSMeasurement contains all of the functions required to select a name server to direct a query to -or use the local recursive resolver-, get the results from that server, and create the DNSRow containing the output of the measurement. From this point on, the software operates asynchronously, using Java's Futures-like mechanism to provide the results.

A. Authoritative Level

The system supports 5 possible authoritative levels to direct the query at: TLD, Parent, Child, Children, and Resolver. The paper by Sommese et al. [9] previously referenced in section III shows the necessity of being able to query different levels. It is important to be able to verify consistency not only between name servers on the same hierarchical level, but also between levels. By implementing the querying of different levels, we answer Research Question 1 in this section.

TLD: This option sends the DNS query to the name server of the top level domain. We use Mozilla's Pubic Suffix List [16] to determine the domain's TLD, resolve the name servers for this TLD and direct the query to these name servers. Since the TLD name servers only know the domains and IP addresses of the name servers one level below them, this option can only handle NS queries. It is important to note that these domains do not appear in the ANSWER section of the response, but in the AUTHORITY section. This is why when the TLD option is used, the authority field in DNSMeasurement must be set to true for the results to be returned. The ADDITIONAL section in the response usually contains the IP addresses of these name servers, which can be included in the results if needed.

Parent: This option sends the DNS query to the name server of the authoritative level above the domain. First, the domain is trimmed, all characters in front of the first period are removed: mail.example.com becomes example.com. The name server for this new domain is resolved, and the query is sent to this name server. If the parent is the top level domain, e.g. example.com is trimmed and the new domain becomes .com, this option is functionally identical as the TLD option. Like the TLD option, this option can only handle NS queries and has to have the AUTHORITY section enabled.

Child: This option sends the DNS query to one of the name servers of the domain itself. First, the name servers of the domain are resolved, then one of them is randomly selected for the query to be directed at. The name server of the domain should be able to provide all authoritative information of the domain, which means that this option can handle all DNS queries. Since the recursive resolver will also end up at this authoritative level, through the root and top-level domain name servers, it is likely that using this option will give the same results as using the resolver.

Children: This option is comparable to the Child option in that the name servers of the domain itself are first resolved, but instead of randomly selecting one of the name servers of the domain, the query is directed to all of them. This makes this option useful for comparing information in different name servers of the same domain, making it an effective tool to verify consistency of records.

Resolver: This option is the only option not to direct a DNS query directly to one authoritative level, instead using the local recursive resolver to answer the query. Like the Child and Children options, this option supports all the DNS query types. This is also the authoritative level used to resolve the name servers for the other levels, as well as the authoritative level used to perform the followup resolution of IP addresses for the resolve_names option in DNSMeasurement.

B. Getting and Parsing Results

In order for the system to use the same data format as OpenINTEL, the data received from the name server in response to the query must be parsed into the OpenINTEL format. We do this through the DNSRow class, the extended format described in section V. The DNSRow class contains fields for all DNS query types, as well as functions for all supported DNS queries. These will first check whether the answer is the correct length, then splits the answer and inserts the parts of the answer into the appropriate DNSRow fields. If

the answer is not of the expected length or the query type is unsupported, the error will be reported with the error_message field. The supported DNS query types can be seen in table IV, with a short description of the corresponding record.

TABLE IV. SUPPORTED QUERY TYPES

Query Type	Description
A	IPv4 address of the domain
AAAA	IPv6 address of the domain
CDS	Child-side copy of DS record
DS	Identifies the DNSKEY for its zone
MX	Email exchange servers for the domain
NS	Name servers of the domain
NSEC	Identifies next domain in the zone
RRSIG	Holds DNSSEC signature for a record set
SOA	Contains administrative information about the zone
TLSA	Holds the public key for the domain's SSL certificate
TXT	Contains text information

The system is also able to handle CNAME, DNAME and PTR query response types. The system can only handle these responses because they all have a set number of fields which contain the parts of the response. For example, an MX record contains a priority and a domain, so the response can always be split in two, and the mx_priority and mx_domain fields can be filled. There are other query types, such as NSEC3 and DNSKEY, that have a different amount of fields in their answer depending on their algorithm. Future works will focus on implementing decoding logic for these types of records. This provides a partial solution to Research Question 2, as only response types with a consistent number of fields can be processed this way.

VII. PERFORMANCE TESTING

The setup for the performance testing of the system consists of 2 virtual machines running on a host machine connected by an NAT network. This means the machines can communicate with each other and with the Internet. The specifications of the two virtual machines are listed in table V.

TABLE V. SPECIFICATIONS OF THE VIRTUAL MACHINES

\	VM1	VM2
Processor	1 core	2 cores
RAM	1 GB	8 GB
Storage	10 GB	20 GB
OS	Debian 10 (no GUI)	Ubuntu 21.04 (GUI)
Software	Unbound Our application	Lenses Kafka Binaries

VM1 runs the application, as well as the Unbound local recursive resolver. Its specifications make sure that the test performance is representative for when the software gets deployed on comparable server machines. VM2 is the broker and monitoring machine, running Lenses and the Kafka message broker. It is on this machine that the measurement requests are uploaded to the requests topic. The measurement payloads consist of JSON files containing the Alexa top 1 million domains, together with varying parameters. The measurements will be performed in random order from this list, meaning the first domain measured will not always be google.com, despite this domain being number 1 on the Alexa top 1 million. In table VI, the measurements are listed. For each measurement, we will determine the average speed at which messages are produced to the results topic, and compare them.

TABLE VI. THE MEASUREMENTS FOR PERFORMANCE TESTING

Authoritative Level	Query Type	Other Parameters
Resolver	A	
Resolver	NS	
Resolver	NS	resolve_names_v4 = true resolve_names_if_additional_not_available = true
Resolver	MX	
Resolver	MX	resolve_names_v4 = true resolve_names_if_additional_not_available = true
Resolver	SOA	

The reason these measurements are chosen is that the software will most likely not perform all of the possible measurements in the same timeframe. For the simplest request, an A or NS record, the software just has to query the resolver, wait for an answer and put it in the ip4_address or ns_adddress field. For MX and SOA, the software has to additionally split the answer, check if it's the right length, and put the results in the right places. NS and MX queries also have the options of resolving the names of the domains, which takes more time still. For this thesis, we did not perform high-volume measurements on the other authoritative levels.

The performance testing was performed by creating a new requests topic and a new results topic in Lenses. We then put the JSON with the payload in the request topic. For a half hour, the message speed was monitored. After 30 minutes, we stopped the program and deleted the topics, to set up for the next measurement.

VIII. RESULTS AND DISCUSSION

Table VII contains an overview of the results from the performance testing, Figure 8 shows in a more graphical way the average number of messages per second for the different query types and their standard deviation. The + behind NS and MX means that the resolve names_v4 field was set to true.

As evidenced by the standard deviation, the message production speed fluctuates. The system does not reach an equilibrium speed. This is most likely due to the different in query response time from different name servers. The payload contains a million domains, the larger of which have better DNS infrastructure, the smaller will be significantly slower. Since query time is the main factor in message speed, this would explain the fluctuation in speed.

TABLE VII. MESSAGES PER SECOND FOR THE RESOLVER QUERY TYPES

Query Type	Messages per Second			Standard Deviation
	Lowest	Average	Highest	
A	38	76	100	11
NS	50	78	103	11
NS+	30	42	48	4
MX	63	81	97	9
MX+	34	41	51	5
SOA	50	83	107	12

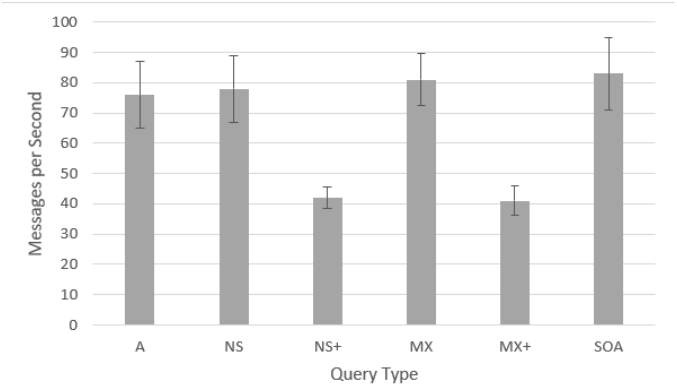

Fig. 8. Average Number of Messages per Second with Standard Deviation

Since an MX record contains 2 fields, and an A record and NS record only 1, the fact that MX queries produce a higher average number of messages per second means processing speed of the decoder logic does not have a noticeable impact on the message speed.

The results do show a possible correlation between message speed and the number of lines in a response. SOA will always only have 1 line [17] but A, NS and MX records may contain multiple lines with multiple IP addresses, name servers or mail servers, respectively. Each line needs a new DNSRow, and processing speed may be influenced by this. Note that these measurements are not sufficiently detailed to conclusively determine this, since the measurement requests were not selected by the number of lines in their records.

Something that can be determined from these results is the clear difference in speed of the measurements that also resolved the names of the NS and MX servers. Since these have to perform follow-up measurements which take just as much time as the original query, the speed is significantly lower. In both cases the average is about half of the original speed of the non-resolving queries.

Because the performance testing was performed on a personal laptop using virtual machines, these results only give a lower bound for the performance of the software. Our software is also not yet optimized for performance, but for function. To answer research question 4, so far our software reaches about 21% to 23% of the performance speed of OpenINTEL. To make this performance difference more insightful, table VIII lists the time it would take for each query type in our system and OpenINTEL to complete all the measurements for the entire Alexa top 1 million domains. These numbers are estimated from the number of measurements per second for both systems, and may not be entirely accurate.

TABLE VIII. ESTIMATED TOTAL TIME TO MEASURE 1 MILLION DOMAINS

Type	Total Time
OpenINTEL	48m
A	3h39m
NS	3h34m
NS+	6h37m
MX	3h26m
MX+	6h47m
SOA	3h21m

IX. CONCLUSION AND RECOMMENDATIONS

We designed a reactive system for DNS measurements based on the message broker Kafka, as an extension for and compatible with the OpenINTEL system. The system is able to query a local recursive resolver, the name server of the parent of a domain, the name server of the TLD of the domain, and either one or all of the name servers of the domain itself. The system supports 10 query types, others would require additional development to implement and were outside the scope of this thesis. The software was tested and was able to produce single results for all supported query types and authoritative levels. Additionally, we tested the performance of the resolver authoritative level of the system with a high volume of queries on hardware comparable to the systems it will be deployed on. When deployed, the system will provide an easily scalable tool for research into consistency of DNS records, such as the research discussed in section III. The system can be expanded to provide the same dynamic scalability functionality to other types of research. These are the main contributions to the scientific community.

The next step for the design of the software is to test the high-volume performance of the other authoritative levels, the same as what we did for the resolver level. Only then will the performance of the software be sufficiently known to make a decision on the merits of deploying the system in a server environment.

After this comes testing the deployment on a larger scale. The performance testing done on a single machine shows promise in terms of scalability, but only testing on a larger scale on appropriate hardware will show the true relevant performance of the software. We performed the tests in this thesis on a personal laptop, which seems to have a huge impact on the processing speed, seeing as initial testing on server hardware yielded results close to 600 messages per second, a huge improvement.

The first possibility for future development of the software is expanding the decoder logic to be compatible with the query types that have a varying amount of fields in their response, such as NSEC3 and DNSKEY.

Another possibility would be to expand the system to be able to do more than just DNS measurements, for example ping or traceroute measurements. The AbstractResult class which DNSMeasurement expands upon makes it possible to do more types of measurements with the same message broker infrastructure.

REFERENCES

[1] ”Definition - DNSSEC” (n.d.) retrieved 25-6-2021 from https://simpledns.plus/help/definition-dnssec

[2] ”List of DNS record types” (2021) retrieved 19-6-2021 from https://en.wikipedia.org/wiki/List_of_DNS_record_types

[3] Kontouras, A., Kintis, P., Lever, C., Chen, Y., Nadji, Y., Dagon, D., Antonakakis, M., Joffe, R., (2015) *Enabling Network Security Through Active DNS Datasets*

[4] Gojmerac, I., Zwickl, P., Kovacs, G., Steindl, C. (2015) *Large-scale active measurements of DNS entries related to e-mail system security*

[5] Mockapetris, P. (1987) *RFC 1035: Domain Names - Implementation and Specification*

[6] van Rijswijk-Deij, R., Jonker, M., Sperotto, A., Pras, A. (2016) *A High-Performance, Scalable Infrastructure for Large-Scale Active DNS Measurements*

[7] "Data Dictionary" (2021), retrieved 16-6-2021 from https://openintel.nl/background/dictionary/

[8] Müller, M. Moura, G.C.M., Schmidt, R.O., Heidemann, J. (2017) *Recursives in the Wild: Engineering Authoritative DNS Servers*

[9] Sommese, R., Moura, G.C.M., Jonker, M., van Rijswijk-Deij, R., Dainotti, A., Claffy, K.C., Sperotto, A. (2020) *When parents and children disagree: Diving into DNS delegation inconsistency*

[10] Akiwate, G., Jonker, M., Sommese, R., Foster, I., Voelker, G.M., Savage, S., Claffy, K.C. (2020) *Unresolved Issues: Prevalence, Persistence, and Perils of Lame Delegations*

[11] Kreps, J., Narkhede, N., Rao, J., (2011) *Kafka: a Distributed Messaging System for Log Processing*

[12] Orsini, C., King, A., Giordano, D., Giotsas, V., Dainotti, A. (2016) *BGPStream: A Software Framework for Live and Historical BGP Data Analysis*

[13] "Alpakka Kafka Documentation" (2021), retrieved 28-6-2021 from https://doc.akka.io/docs/alpakka-kafka/current/

[14] "Unbound" (n.d.) retrieved 22-6-2021 from https://www.nlnetlabs.nl/projects/unbound/about/

[15] "Apache Kafka Docker" (2020) retrieved 10-6-2021 from https://lenses.io/apache-kafka-docker/

[16] "Public Suffix List" (n.d.) retrieved 30-6-2021 from https://publicsuffix.org

[17] "SOA Records" (n.d.) retrieved 28-6-2021 from https://support.dnsimple.com/articles/soa-record/