Dynamicity Management in Domain Name System Resource Records

Jarno van Leeuwen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.vanleeuwen@student.utwente.nl

ABSTRACT

The Domain Name System (DNS) is a distributed naming system which relates information about entities connected to the Internet or other networks to their domain names using resource records stored in a database. Such a system could be utilised in a network of any domain to access its entities by their names. A vehicular network is an example of such an application domain. In a vehicular network scenario various services and messages may be destined for vehicles located in a specific region of interest. Therefore, the information regarding the location of a network entities could be of significant importance and demand as is in the case of vehicular networks. As specified in IETF RFC 1876 [2], an extended version of DNS system is defined which associates location information of entities to their names and consequently makes it possible to resolve queries containing location information. However, in the case of a vehicular network, as the vehicles are highly mobile, it is a challenge how to effectively keep track of updated location information in the DNS system database. Therefore, in this work a DNS server is extended to handle the dynamic updating of resource records, followed by setting up a simulation environment to perform quantitive measurements on the performance of the implemented solution in different scenarios.

Keywords

DNS, eDNS, Dynamic Zone File, Location Resource Records

1. INTRODUCTION

Communication is an indispensable part of today's world. Vehicular systems are a relevant domain of communication [1] with the potential of providing a wide range of services and applications from safety to infotainment and Internet access. In this regard, Intelligent Transport Systems (ITS) [3] emerged as a standardizing framework for providing intelligence to transport networks. For vehicular systems to function properly, every entity must be accessible in the whole network. To achieve this, most of the current networks rely on the Domain Name System (DNS). DNS operates by translating human friendly domain names to the IP addresses through the format described by the record type. Vehicular systems may also utilise DNS to provide Vehicle to Vehicle communication [4]. The nature of such systems demands for location-dependent message/service provision and it implies that knowledge regarding location information of vehicles is crucial for overall functionality of the system. An extension is defined by introducing location records (e.g., LOC records) to the DNS system [2].

There are various entities in a vehicular network taking part to construct a communication system. Communication in such a system maybe vehicle-to-vehicle and/or vehicle to infrastructure. Infrastructure here refers to servers and RoadSide Units (RSU) in charge of message/service construction and propagation to the vehicles requiring them or affected in the case of safety and emergency. For instance, information regarding an icy road is relevant just for the vehicles around that icy road. Therefore, storing the location information of vehicles is necessary and as already mentioned, location records are defined to add this capability to the DNS system. However, in the scope of vehicular networks, high mobility of vehicles results in challenges in this regard as location information of vehicles turn to highly changing records.

Tiago Fioreze et al. [4] proposed a DNS extension, called eDNS, to provide the ability to issue queries containing location information to an eDNS server for the purpose of directing messages from RSUs to On-Board Units (OBUs) of vehicles in a specified area. This was achieved by extending a DNS server called Name Server Daemon (NSD), which is originally developed in C [6]. The server was extended with functionality to manage the translation of geographic coordinates to IP addresses in range. The database of the eDNS server contains the mapping between location information (e.g., LOC records) and IP addresses. Geographically scoped queries are supported through such mapping.

Vehicular systems have peculiar behaviors and characteristics mainly derived from high mobility of the network entities. Therefore, domain-specific communication systems are required. Continuously moving vehicles means continuously changing location records. Accordingly, eDNS system must be adopted to efficiently manage this dynamicity and keep all entities accessible. A central server could be responsible for monitoring entities movement and provide the most updated mappings between IP addresses and the relevant location information, which could be later used by the eDNS system to keep track of the location changes and update the LOC records in its database.

In this work we developed an algorithm, which is able to retrieve changes from the central server and accordingly update the eDNS server's databased in realtime. A simulation environment is set up to simulate different scenarios. Further, some optimization works have been done to improve the protocol's performance. Observations from simulation results demonstrate reasonable performance of the developed algorithm in terms of resource consumption and time-efficiency.

1.1 Problem Statement

Utilisation of an extended version of the DNS system which is able to resolve queries containing location information is of remarkable profit to the networks in the need of location dependent service supply. Vehicular networks are one of the relevant application areas able to realize many services gaining from such a system. In this context high mobility of vehicles is a challenge and must be efficiently managed by means of dynamically changing location records in the eDNS system.

1.2 Research Questions

Derived from the mentioned complexity, the research addresses the following primary research question:

- How to efficiently deal with frequent location changes in a dynamic communication network, which in turn affects the eDNS system management in the need of such information?

which can be subdivided into the following subproblems:

1. How to extend eDNS with functionality to dynamically manage location records at runtime?
2. How to perform quantitive measurements on the extension?
3. How to account for different environment factors, in particular dynamicity?
4. How to interpret efficiency (speed vs. accuracy)?

This paper is organized as follows. In the next section the requirements for the solution will be stated. In Section 3 the underlying architecture will be explained. Then after, a simulation environment simulating real world scenarios is elaborated in Section 4. The results generated by the simulator are presented in Section 5 and interpreted to come up with a conclusion in Section 6. Finally Section 7 provides ideas for future work.

1.3 Related Work

The Domain Name System (DNS) is a cornerstone of the Internet and it has a convenient way of storing and retrieving resource records. Therefore it is an obvious choice to extend DNS and provide it with more logic, since it has already been incorporated in most networks.

Yahya et al. [12] studied and evaluated DNS with respect to a dynamic updating protocol and provided a performance study to determine the ability of DNS to support mobility and real time services. The results have shown that DNS with dynamic update features is suitable for services that require high update rates. One drawback of this research was the use of existing tools to update the DNS server. Not all DNS server provide native support for dynamic updates.

Ben-Othman et al. [13] proposed a distributed naming system based on the peer-to-peer Chord [7] protocol. This distributed naming system is focussed on solving problems concerning latency and scalability and could even replace DNS. However, the simulation results showed the system experiencing large latency to resolve lookups compared to traditional DNS.

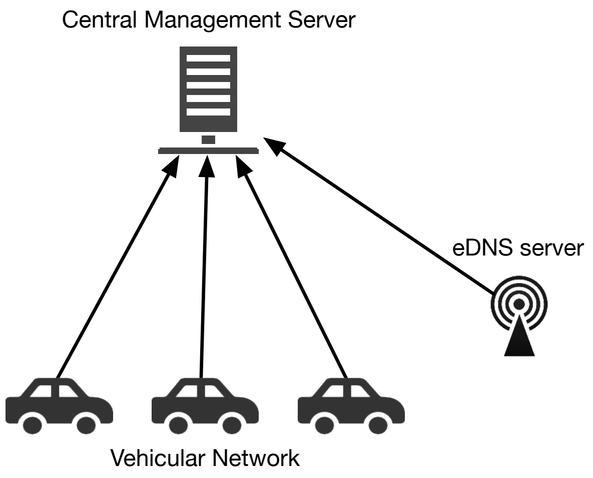

Figure 1. The eDNS server fetches dynamically changing location information from a presumed central location management server.

Contrary to the described work, in this research we want to add dynamicity management in resource records without digressing too much from traditional DNS. This means without relying on peer-to-peer protocols or other existing tools which are not incorporated in all DNS servers.

2. REQUIREMENTS

The amount of changes in the records can vary so the implementation must be scalable. Most DNS servers support a master and slave setup combined with notifications. This enables the server to propagate record changes to its slaves. [9]

In environments with a lot of resources the number of records can considerably grow. The processing of records can become expensive in terms of processing power and time. Therefore an algorithm must be adopted which effectively takes the number of records into account.

The amount of changes may depend on the location and time of the day. During rush hours new entities can pop up at a high rate. To be effective, the solution must support different levels of dynamicity in the environment.

In order to model the above-mentioned scenarios and come up with an algorithm to effectively update eDNS' database, we need to define metrics and from the descriptions above we can already see the following emerging: a refresh timer, the number of records and a dynamicity factor.

3. ARCHITECTURE

In this section eDNS is extended to handle the dynamic updating of records. This enables the server to dynamically manage the information of entities in a vehicular network. See Figure 1 for a scenario overview.

The Domain Name System (DNS) is a distributed naming system which mainly translates input (e.g., a domain name) into its corresponding IP address [5]. The mappings are defined in a zone file. DNS has support for LOC records which is a record type for expressing the geographic location of a domain name [2]. A LOC record has the following format:

<owner> <TTL> <class> LOC (d1 [m1 [s1]] "N"|"S" d2 [m2 [s2]] "E"|"W" alt["m"] [siz["m"] [hp["m"] [vp["m"]]]])

Here present d1 the latitude in degrees from 0 to 90, d2 the longitude in degrees from 0 to 180, m1 and m2 the minutes from 0 to 59 and s1 and s2 the seconds from 0 to 59.999. Alt presents the altitude in meters from -100000.00 to 42849672.95 and size, hp and vp present respectively size, horizontal precision and vertical precision in meters from 0 to 90000000.00. Minutes and seconds can be omitted in both the latitude and longitude, defaulting to zero. [2]

Based on the work by T. Fioreze et al. [4], querying by a geographical location area returns all the records falling into the specified location [11]. This extension is called extended DNS (eDNS) and is built on top of Name Server Daemon (NSD) —a high performance, simple and open source name server [6]. In this research this specific extension will be used.

Note that here it is assumed that a central server is available, which is responsible for keeping track of changes in the vehicles location. Management details of this server is beyond the scope of this paper. The entities having messages to be sent to specific target areas of a network may utilise this functionality to query the eDNS server and get back the relevant IP addresses.

NSD has no support for dynamically changing records as defined in RFC 2136 [8] without the need to edit zone files and start the server. Therefore eDNS was extended to retrieve raw records from the central server, rewrite the zone file, rebuild the database and reload the server with the new database. See Algorithm 1 for a naive approach.

```
1 while(true) {
2   retrieve records from central server;
3   rewrite zone file;
4   rebuild database;
5
6   reload server;
7
8   wait(refresh timer);
9 }
```

Algorithm 1. The naive approach.

The above algorithm is always running in the background when the server is active, therefore the algorithm is continuously running as defined by the while loop at line 1. Since it is not the intention to continuously refresh the zone file the execution is delayed for the amount of time defined by the refresh timer (line 8). At line 2 the current state of the records is retrieved from a central server. This data is processed and parsed into the zone file (line 3). Finally the database is rebuilt (line 4) and the server is reloaded (line 6) to adopt the new data.

It should be noted that there is an explicit difference between reloading and restarting the server. Reloading is done while the server keeps running. It keeps running in its old state while reloading, and when it is ready to switch to the new state this is done in one atomic operation. This is desirable behavior because it avoids downtime of the server.

Notice that the above algorithm is running in an infinite loop, immediately starting the update process after a certain amount of time. The I/O operation of rewriting the zone file and the complex operation of rebuilding the database can be very time- and processing power- consuming, especially on larger data sets. See Algorithm 2 for a better approach.

```
1  recordsChanged = 0;
2
3  while(true) {
4    retrieve records from central server;
5    recordsChanged += amount of changed
         records;
6
7    if(recordsChanged > threshold) {
8      rewrite zone file;
9      rebuild database;
10     reload server;
11
12     recordsChanged = 0;
13   }
14
15   wait(refresh timer);
16 }
```

Algorithm 2. The improved approach.

The overall behavior remains unchanged, records are being fetched from a central server and adopted by the DNS server. Except for the introduction of the recordsChanged variable (line 1). This variable keeps track of the amount of changed records (line 5). In this case the server only rewrites the zone file and rebuilds the database when a certain number of records —defined as threshold —are changed (line 7). After carrying out the complex operations of fetching and processing the new data, the new variable is reset to 0 and the same flow is executed again.

Another optimisation can be achieved by making intelligent use of the servers zoning mechanism. In a vehicular system there are various other entities, like RSUs, beside vehicles. As all these entities are not mobile, it is a good approach to split the master zone file in two zones. This way only the file containing resource records of the mobile entities has to be rewritten, reducing I/O operations. Since NSD compares its zone files' hashes before rebuilding the database, waste of processing power is also avoided.

In summary, we have extended the eDNS server to dynamically respond to zone changes in a fairly straight-forward manner.

4. SIMULATION

We are now able to manage dynamic resource records in eDNS. However, it is of great value to know the performance of the solution since vehicular networks can come in many shapes. Therefore, a simulator was built to imitate different real world scenarios. The factors zone size and dynamicity factor were added address multiple environment and to see if the implemented architecture meets the stated requirements. All the factors and definitions that are used in this work are defined in Table 1.

Table 1. The Analysis Notation

Text	Description
Active State	Indicator whether a resource record is enabled or not
Dynamicity Factor	Determines the amount and rate of state changes
Tick	Recurring event, fired at a static interval
Zone Size	Maximum number of records in the zone

The zone size defines the theoretical maximum number of dynamic records that can be active at the same time. It does not necessarily mean that the number of the records in the zone file loaded by the server equals to the defined zone size, in contrary. Every individual dynamic record has a probability, defined by the dynamicity factor, to be flagged as active or inactive at any given time. The active state determines whether the record is included in the zone file or not. The dynamicity factor determines the rate of changes in the environment. It is a real number between zero and one (inclusive). All values are defined as follows:

Definition 1. Every tick there is a DF probability to toggle the active state of $DF \times ZS$ records.

Where DF is defined as the dynamicity factor and ZS as the zone size. A tick is a recurring event, automatically fired by the simulator after a fixed amount of time, e.g. 1 second.

Zero denotes no changes at all, implying a static environment. A value of one denotes toggling the active state of all records every tick. Toggling is obviously defined as changing the state from active to inactive or vice versa. Thus a value of one is actually equal to flipping the state of the full environment every tick.

For example, we define that there are 3000 dynamic units in our environment. We also specify the environment to have a dynamicity factor of 0,300. This implies that per tick there is 30.0% probability for $3000 \times 0.300 = 900$ records to toggle their active state. Note that the factors have no influence on the actual number of records currently loaded by the server. There always is a probability that all records become active, or that there are no records active at all. Consequently, at any point in time there are records ranging from zero to zone size, marked as active and loaded by the server.

See Algorithm 3 for the pseudocode of the actual algorithm of the simulator.

```
 1 ZS = zone size;
 2 DF = dynamicity factor;
 3
 4 intialize ZS dummy records;
 5 set t0;
 6
 7 while(true) {
 8   t1 = time();
 9
10   if(t1-t0 > 1s) {
11     t0 = t1;
12     if(rand(1, 1/DF) == 1) {
13       switch active state of ZS*DF random
           records;
14       rewrite zone file;
15     }
16   }
17
18   dig();
19 }
```

Algorithm 3. The simulator algorithm.

The algorithm starts with initializing ZS records (line 4). Records represent an entity and are featured with random values for their properties (e.g. IP address, latitude and longitude). As described above we also need to introduce the simulation of ticks. This is achieved by introducing two timestamps, t0 and t1. t0 is a fixed timestamp set initially (line 5) and after a tick (line 11). t1 always represents the current timestamp. Calculating the absolute difference of the timestamp enables us to determine the time passed, and firing a tick event accordingly (line 12). The time between two consecutive ticks is defined in our simulation as 1 second. In the end, the time between ticks and the refresh timer of the server is relative and a lower value allows faster simulation.

However, the active state of entities/records is not changed at every tick. As defined by Definition 1, the dynamicity factor determines the probability for changes to occur. To account for this behavior a random integer between 1 and $1/DF$ is generated (line 12). Active states are only changed when the generated value equals 1. E.g., for $DF = 0.1$ a random integer between 1 and 10 (inclusive) is generated. Now there is a 10% probability for this value to equal 1.

The simulator finally changes the active state of records when we are in the situation where a tick event occurred and the probability equation (line 12) equals to true. As defined by Definition 1, the dynamicity has also influence on the amount of records to be changed. Therefore the active state is only changed (flipped from active to inactive or vice versa) of $ZS * DF$ records.

Lastly the zone file is rewritten to reflect the new active states. eDNS then fetches and processes the new active states. We could also imitate a central server and make eDNS fetch the data through a web socket. Fetching delay is also being simulated, since exchanging data via web sockets and writing to a file both add overhead.

Also remarkable is the (optional) dig method inside the loop. Dig (domain information groper) is a UNIX command for querying DNS servers [10]. It replies with an overview of the query and server answer, including information about the latency. Using this tool we are able to determine the current active state of the dynamic records on the server side. We can achieve this by, for example, carefully picking the IP address or location when initializing the records. If all records share the same location we could send a dig request to the eDNS server matching all records. The data returned enables us to infer all active states, namely active when a record is present in the answer section and otherwise inactive.

With all these characteristics implemented, the simulator is capable of comparing the generated state, defining the actual environment, to the current state on the server. This enables us to measure the time it takes for record changes to propagate to the server.

More importantly, we are now capable of artificially simulating different levels of dynamicity in a vehicular network. The next step is to quantify the performance of the dynamic management of resource records. Different measurements are elaborated in the next section.

5. RESULTS

To determine the efficiency of dynamic zone handling by the server the simulator was executed using different settings. Zone size affects the rebuild time, so it is important to see how the server behaves in terms of rebuild time for different values of the zone file. Therefore, the rebuild time is determined for different values of the zone size. Finally CPU utilisation is taken in account.

5.1 Zone size vs. Rebuild time

The rebuild time is the time the server spends loading the zone file and rebuilding the database. It is measured for different values of the zone size. It should be noted that the server is still available during its rebuild phase, but

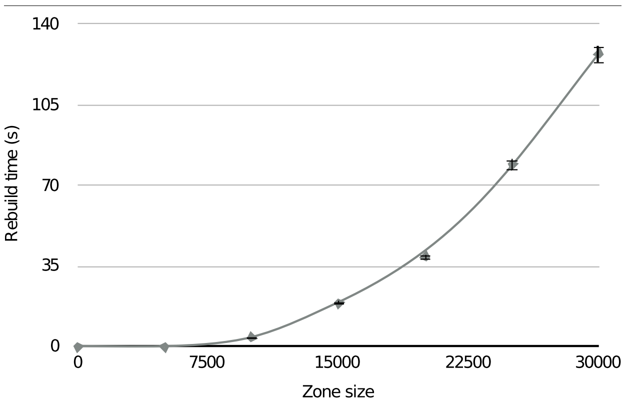

Figure 2. Zone size versus rebuild time.

still using the old zone file.

The rebuild time was measured for different values of the zone size and plotted in Figure 2. The x-axis represents the zone size and the y-axis represents the rebuild time in seconds. The rebuild time is monotonic increasing for higher values of the zone size. The rebuild time remains close to zero till a zone size of about 7500 entities/records. From there the rebuild time greatly increases. The growth is bigger than linear growth, but not much. This can be explained due to the fact that every record adds the same amount of processing delay plus I/O writing operations on a growing zone file.

5.2 Zone size vs. Average CPU utilisation vs. Server reload timer

The percentual average CPU usage over the last minute was measured for different values of the zone size. The performance was measured for two values of the server reload timer: 0 and 60 seconds. The data is plotted respectively in Figure 3 and Figure 4. The x-axis represents the zone size and the y-axis represents the CPU utilisation from 0% to 100%. The diagram shows the CPU utilisation monotonic increasing for bigger values of the zone size. For a server reload timer of 0 the CPU utilisation starts growing exponentially till it reaches a limit of 100%. Increasing the server reload timer to 60 shows a significantly different diagram, the CPU utilisation remains close to zero before growing linearly.

The shown behavior can be explained as follows. One would expect the server to be utilising all its CPU power when the server reload timer is set to zero, since it is continually rewriting the zone file and rebuilding database. However, the server also spends time on reloading, which is a less expensive operation. For a small zone size the rebuild time is very low as measured in Section 5.1. Therefore the server spends a relatively long time reloading, compared to the time spent on refreshing the zone data. Once the zone size reaches proportions where the rebuild time becomes significant larger than the time needed for a server reload, the CPU usage jumps to 100%.

The same behaviour is visible in Figure 4. However, the server spends less time on rebuilding the database in this case. Again the CPU utilisation depends on the ratio between the amount of time spent on refreshing zone data and the contrary.

Parsing the zone file and rebuilding the database is a CPU expensive operation. Shorter server reload timers result in a more real-time representation of the environment on the server, but also causes the server to utilise all its processor

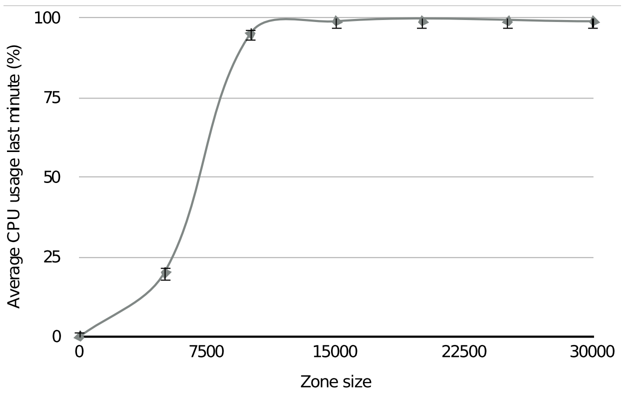

Figure 3. Zone size versus Average CPU usage for Server reload timer = 0s.

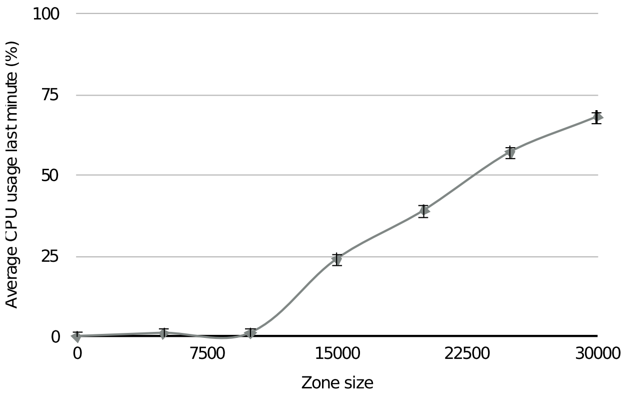

Figure 4. Zone size versus Average CPU usage for Server reload timer = 60s.

power on keeping the records up to date. This processing power is therefore unavailable to handle incoming queries.

In this simulation a static value is assigned to the server reload timer. Specifying reload times could be dynamically managed based on the circumstances to have the best performance possible, but this surpasses the scope of this research.

5.3 Zone size vs. Average CPU usage vs. Collect threshold and Dynamicity factor

As mentioned before the algorithm running on the server can be optimised by specifying a refresh threshold. The server then only goes into its rebuild phase when a certain number of state changes occurred. One would suggest that the performance should be tested for different threshold values. But if we take a look at the timing mechanism of the server and the simulator, we observe that the server reload timer directly depends on the refresh threshold and dynamicity factor:

$$SRT = \frac{RT}{DF^2}$$

Where SRT stands for the server reload timer, RT for the refresh threshold and DF for the dynamicity factor. Here RT is a percentage expressed as a real value between 0 and 1. As an example, if we want to refresh the server only when 50% of the records are changed in the environment with a dynamicity factor of 0.1, we obtain:

$$SRT = \frac{0.5}{0.1^2} = 50s$$

This equation is derived as follows. Every time the environment changes, 10% of the records are affected. The server only needs to update its database when a total of 50% of the zone size changes occurred. In this case, this is reached after 5 environment changes. Because of this dynamicity factor, in every time step, there is a 10% probability for an environment change to occur. This is a geometric distribution since the probability of success remains unchanged, namely 0.1. The corresponds with an expectation value of 10 seconds. Therefore it takes approximately $5 \times 10 = 50$ seconds to reach the refresh threshold of 50%.

6. CONCLUSION

DNS is a widely used naming system in communication networks to locate the entities of a network. Location dependency could be of integral necessity in some domains, such as vehicular networking for certain services. This resulted in the definition of LOC records mapping to IP addresses for DNS as in RFC 1876 [2].

High mobility as the most dominant characteristic of a vehicular system, brings out the challenge of location records which are valid for a very short time and it was our motivation to manage this challenge and develop a relevant efficient solution.

To dynamically update location records at runtime the eDNS server was extended to rewrite its zone file and rebuild the database. The rate of this process can be configured and has influence on the utilisation of the server. Therefore must be determined what the best configuration is in different scenarios.

Different factors are distinguished to identify an environment: its zone size and dynamicity factor. The size of the zone file has the most impact on the time needed for the server to update its zone file. The update time grows exponentially with the size of the zone. Two algorithms were presented: the first updates the server at a given interval, and the seconds updates the server after a given amount of records are changed.

To perform quantitive measurements on the implementation a simulator was built with configuration options for the given factors. Different scenarios were simulated and compared. The best solution depends on the interpretation of efficiency.

When the interpretation of efficiency is to have very accurate data, processing power will have to be sacrificed. In such cases it is efficient to configure a low reload timer. However, a too low reload timer can result in high cpu load, rendering the DNS server useless for incoming connections.

Dynamically changing location records can be maintained by a DNS server. However, the efficiency can drop very quickly when the zone becomes larger (> 5000) and more dynamic. To efficiently use dynamically changing location records, an approximation of the zone size and dynamicity must be known to be able to configure the server.

7. FUTURE WORK

This work relies on eDNS which is built on top of NSD version 3. Currently, NSD version 4 is available which does support dynamically updating zone files with the UNIX command nsupdate. A different approach using this functionality could be implemented and compared with the solution described in this paper. Further research can also be carried out on utilizing multiple zone files. In this research we only distinguished static and dynamic records. By further classifying the dynamic records, for example by refresh rate, a hierarchical network of smaller zone files can be set up. This most likely results in lower rebuild times and less I/O operations.

8. REFERENCES

[1] H. Conceicao et al. Large-scale simulation of v2v environments. *SAC '08 Proceedings of the 2008 ACM symposium on Applied computing*, pages 28–33, 2008.

[2] C. David et al. A means for expressing location information in the domain name system. *RFC 1876*, 1996.

[3] ETSI. ETSI: intelligent transport systems. `http://www.etsi.org/technologies-clusters/technologies/intelligent-transport`. Accessed: 2014-04-02.

[4] T. Fioreze and G. Heijenk. Extending the domain name system (dns) to provide geographical addressing towards vehicular ad-hoc networks (vanets). November 2011.

[5] P. Mockapetris. Domain names - implementation and specification. *RFC 1035*, 1987.

[6] NLnet Labs. NSD: name server daemon description. `http://www.nlnetlabs.nl/projects/nsd/`. Accessed: 2014-03-28.

[7] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, 2001.

[8] S. Thomson et al. Dynamic updates in the domain name system. *RFC 2136*, April 1997.

[9] P. Vixie. A mechanism for prompt notification of zone changes. *RFC 1996*, 1996.

[10] P. Vixie. Extension mechanisms for dns. *RFC 2671*, August 1999.

[11] M. Westra. Extending the domain name system with geographically scoped queries. August 2013.

[12] B. Yahya and J. Ben-Othman. Achieving host mobility using dns dynamic updating protocol. *33rd IEEE Conference on Local Computer Networks, 2008.*, pages 634–638, October 2008.

[13] B. Yahya and J. Ben-Othman. A peer-to-peer based naming system for mobile ad hoc networks. *35th IEEE Conference on Local Computer Networks, 2008.*, pages 821–826, October 2010.