Routing Tables Building Methods for Increasing DNS(SEC) Resolving Platforms Efficiency

Emmanuel Herbert, Daniel Migault,
Stephane Senecal and Stanislas Francfort
France Telecom R&D/Orange Labs
38-40 rue du Général Leclerc
92130 Issy-les-Moulineaux, France
{firstname.lastname}@orange.com

Maryline Laurent
Institut Mines-TELECOM, TELECOM SudParis,
UMR CNRS 5157 SAMOVAR
9 rue Charles Fourier
91000 Evry, France
maryline.laurent@telecom-sudparis.eu

***Abstract*—This paper proposes to use optimization and machine learning methods in order to develop innovative techniques for balancing the DNS(SEC) traffic according to Fully Qualified Domain Names (FQDN), rather than according to the IP addresses. With DNS traffic doubling every year and the deployment of its secure extension DNSSEC, DNS resolving platforms require more and more resources. A way to cope with these increasing resources demands is to balance the DNS traffic among the DNS platform servers based on the queried FQDNs. Several methods are considered to build a FQDN based routing table: mixed integer linear programming (MILP), a K-means clustering algorithm and a heuristic scheme. These load balancing approaches are run and evaluated on real DNS traffic data extracted from the operational IP network of an Internet Service Provider (ISP) and they result in a difference of less than 2% CPU between the servers of a platform.**

Keywords: DNS, DNSSEC, optimization, machine learning, routing, load balancing.

I. Introduction

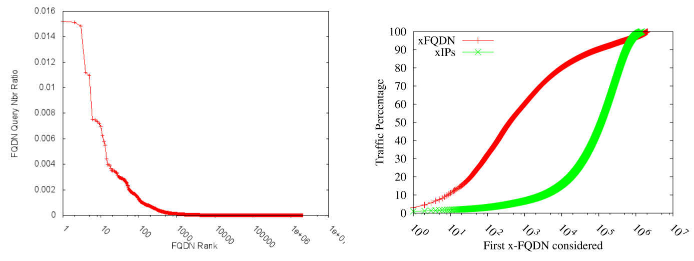
Domain Name System (DNS) is the service which makes possible network communications based on names, by binding an Internet Protocol (IP) address to a Fully Qualified Domain Name (FQDN). As such, end users rely on a DNS resolving platform to determine the IP address of the requested target. The DNS architecture is composed of *Authoritative Servers*, which are DNS servers hosting the IP addresses of the queried web sites. Because *Authoritative Servers* would not be able to process all end users' queries, the DNS architecture introduces *Resolving Servers* which cache the responses during a given number of seconds called Time To Live (TTL). Internet Service Providers (ISPs) manage such servers for their end users. When a *Resolving Server* receives a DNS query and if the response is already cached, then the *Resolving Server* returns the cached response. Caching mechanism provides faster responses to the end users and reduces the traffic load on the DNS *Authoritative Servers*. The whole DNS architecture is described in details in [1].

As DNS traffic doubles every year and as the DNS security extension DNSSEC is deployed, DNS resolving platforms are more and more resource demanding.

Today's load balancers practice xoring operation over the DNS packet's source and destination IP addresses [2] to balance the DNS traffic on the platform. This load balancing technique, referred to as *XOR* in this paper, induces that most of the popular FQDNs are resolved by each DNS *Resolving Server* of the platform.

One way to reduce the load on the servers of the platform is thus to decrease the number of resolutions. That is, as presented in [3] and [4], balancing the DNS traffic according to the queried FQDNs rather than according to the IP addresses. This implies that more DNS requests are resolved by the caching mechanism, thus leading to fewer signature checking operations for DNSSEC, and a 1.32 more efficient architecture. This load balancing algorithm is referred to as *FQDN* in the following.

The FQDN load balancing can be implemented by hashing the queried FQDN (e.g. using SHA1 [5] as a hash function) but this is not sufficient as it results in an unbalanced distribution among the servers. The contribution of the paper lies in the development of methods for building FQDN-based routing tables for load balancing purpose in an ISP operational context.



(a) Popularity (b) Cumulative Traffic Distribution

Fig. 1: Ranked FQDNs' distribution

II. Related Works

Performing load balancing for the DNS traffic is a crucial task to be implemented on the servers of resolving platforms. General load balancing techniques are introduced in [6]. Previous works concerning load balancing techniques address mainly the problems faced on web servers, like in [7]. Focusing on DNS service, and especially considering problems faced by ISPs in the telecommunications industry, it is possible to highlight two complementary references: [4] and [8]. In [4], a FQDN load balancing technique, based on discrete optimization, is proposed. Reference [8] aims at optimizing the DNS service by addressing its architecture. In this paper, we propose three methods for implementing a FQDN-based load balancing for the DNS traffic, which are derived from mathematical programming, machine learning (clustering), and heurisitic optimization.

III. Building Routing Table

In this section, we focus on the few costly FQDNs of the DNS traffic. There are only few costly FQDNs, as depicted in figure 1a. As shown on this figure, the distribution of FQDNs looks like a Zipf law [9] and [10]. This is why we decide to split FQDNs encountered into 2 parts: $FQDN^+$, the set of the most requested FQDNs for which we can take advantage of caching mechanism and $FQDN^-$, the rest of FQDNs. This section investigates different ways to build a routing table for the FQDNs belonging to $FQDN^+$, namely how to assign each of the considered FQDNs to a unique server. The challenge is to balance the time required to build the routing table with the number of entries of the routing table. FQDNs that are not in the routing table are assigned to the servers according to a hash function. A first method is based on Mixed Integer Linear Programing (*MILP*) and attempts to balance both the queries and the resolutions on every server of the platform. A second approach, called "stacking", uses the global cost associated to the FQDNs. Lastly, we consider to build a routing table based directly on the output of the K-means clustering algorithm applied to some ad hoc variables characterizing the FQDNs. Each of these algorithms are compared by running a simulation with a 10 minutes DNS traffic capture. Comparison is performed via the difference of queries (resp. resolutions) processed by each server of the platform, as well as the corresponding CPU. In this section, we consider a platform formed by a cluster of 10 servers dedicated to DNS resolution.

A. Modelization

Operational teams evaluate the efficiency of different load balancing techniques by comparing the CPU load of each server. The latency is not considered in this work as it is not possible to model it in a simple way. Providing an estimation of CPU load for a server relies on experimental measurements, and [11] mentioned that measured values for the CPU load depend on the hardware, the DNS server implementation and the traffic, among other parameters. Since we do not want to rely on these factors, we evaluate the differences of the CPU load by considering the number of queries and resolutions performed by each server of the platform. Such evaluation requires to define specific notation and formulation introduced in [4]. These notations are used to build a routing table with a Mixed Integer Linear Programing (MILP) method. For each FQDN, the number of resolutions is computed thanks to the queries number and the mean TTL value observed for the FQDN. Since we consider popular FQDNs, we assume that a resolution occurs every TTL seconds.

B. Mixed Integer Linear Programming Method

The Mixed Integer Linear Programming (MILP) method consists in finding a minimum with respect to inequations constraints, cf. [12] for instance. We define a given set I of FQDNs. For a given distribution of these FQDNs on the servers of the set J, we compute the number of queries and resolutions supported by each server $(Q_j, R_j)_{j\in J}$. The distribution we are seeking for is the distribution which minimizes the difference between the different servers of the platform, in term of $(Q_j, R_j)_{j\in J}$: ΔQ and ΔR.
Trying all possible combinations for such a distribution is not feasible, so we formulate our problem as a Mixed Integer Linear Program and use a solver (GLPK in this case, cf. [13]) to find a proper distribution. The challenge of the solver is to find a distribution which is close to the optimal distribution, even though we do not know the optimal solution. Reference [4] shows that a set of 200 FQDNs in $FQDN^+$ is a compromise between the number of FQDNs to process and the resources needed for the computation.
The Integer Linear optimization problem with two objectives consists in minimizing ΔQ and ΔR defined as followed:

$$\Delta Q = \max_{j_1\in J} Q_{j_1} - \min_{j_2\in J} Q_{j_2}$$
$$\Delta R = \max_{j_1\in J} R_{j_1} - \min_{j_2\in J} R_{j_2}$$

To ease the resolution by the solver, we reduce the number of objectives by defining a FQDN cost combining the query rate q_i and the resolution rate r_i for FQDN $i \in I$:

$$c_i = \lambda.q_i + (1-\lambda).r_i \ \forall i \in I \text{ where } \lambda \in [0,1]$$

λ is a weighting parameter which determines the parts of q and r in the definition of the cost. In that sense, the important parameter is not λ itself but the ratio $\frac{\lambda}{1-\lambda}$ (or $\frac{1-\lambda}{\lambda}$). Note that λ is introduced here only for resolving purpose, and has *a priori* no physical meaning. The objective function associated to this cost ensures the minimization of the cost supported by each server, which leads to the minimization of the difference of costs between the different servers. Costs which are not supported by the most loaded server are transfered to other servers, increasing the cost supported by the less loaded server, which reduce difference of cost supported by servers.
We run a solver for different values of λ and for each value we consider ΔQ and ΔR. Figure 2 represents the solutions for specific values of λ. Some values of λ do not provide optimal solution whereas others provide an optimal solution. In the figure 2, these optimal solutions for λ are pointed by the line called *Pareto front*. Among the values on the front, there are no mathematical way to decide whether one is better than the other. Choosing a solution is based on choosing whether we prefer to minimize ΔQ or ΔR. Thus, the decision should consider other aspects such as operational criteria. Also, we should mention that λ values on the *Pareto front* are difficult to characterize. If λ_0 correspond to a point located on the *Pareto front*, then $(\lambda_0+\varepsilon)$ does not necessarily provide a value close to the front. The value can be located anywhere on the $(\Delta Q, \Delta R)$ plane. This reflects the non convex aspect of our problem since we use a discrete space for $(x_{i,j})_{i\in I, j\in J}$. We use the solver *GLPK* [13] running during 1000 seconds to solve this problem. We recall that the routing table is built in an off-line mode, only once. Running *GLPK* longer does not give better results for our 200 FQDNs set. The number of FQDNs chosen is limited by resources used by the solver. We denote this algorithm by *milp-200*, and figure 2 shows that ΔR as well as ΔQ can be very small.

C. Easy Stacking Heuristic

Because of computation resources and time needed by the method described in subsection III-B, we consider another

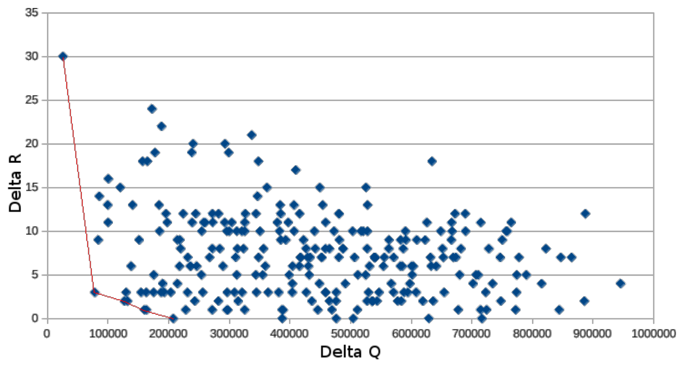


Fig. 2: Bi-criteria MILP results and Pareto front

approach. The goal of this algorithm is similar to *milp-200*: minimizing jointly ΔQ and ΔR. However, the way we build the routing table provides less accurate results as *milp-200*. As a result, we need to consider a much larger set - namely 18 times larger - of FQDNs to build a routing table which balances properly the load among the servers. Even though the routing table is roughly 18 times larger, it takes less than 0.5 second to build it. Compared to 1000 seconds with *milp-200*, this method may present an operational advantage over *milp-200*.
The algorithm starts with I, a set of the most requested FQDNs. From the current set of FQDNs, it takes the costliest FQDN, assigns it to the less charged server (i.e. a server j_{min} verifying $C_{j_{min}} = min_{j \in J} C_j$) and removes it from the FQDNs set. This step is performed until the set of FQDNs is empty.
To compare it with the *milp-200*, we compute ΔQ and ΔR for different values of λ. We first choose a set of 200 FQDNs to be compared with results from subsection III-B. We choose I, the set of FQDNs, such that the last element is associated to a cost which is roughly the difference of costs generated by *stacking-200*. This leads to consider 1580 FQDNs. We denote these algorithms as *stacking-200* and *stacking-1580*. Note that 200 FQDNs represent 16% of the queries number and 1580 FQDNs represent 46% of the queries number.

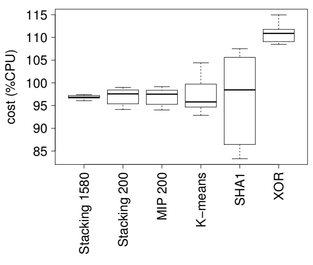
D. K-means Clustering Based Method

This section proposes to use the K-means algorithm as another way to build routing tables. The K-means is a clustering algorithm, widely used in the field of machine learning, cf. [14] and [15] for instance. In our case, it groups similar FQDNs into clusters such that the differences (or dissimilarities) between FQDNs into a cluster are minimal. The K-means is run on all FQDNs and groups the FQDNs with similar costs. One of the cluster is very large and contains the FQDNs with the lowest cost, i.e. the FQDNs requested only a few times during the studied period. These FQDNs are less popular, and thus we do not consider them in routing tables, they will be assigned to a server according to a hash function. For all other clusters, we spread FQDNs within each cluster uniformly among the servers. We expect to balance properly the traffic among the servers.
In this section, the K-means algorithm is applied with one parameter which results in a collection of 5 clusters. The variable used is a weighted sum of parameters reflecting the cost in term of CPU load due to each FQDN. In other words, we cluster FQDNs on the basis of an evaluation of the CPU load computed thanks to the number of query and resolutions. We denote by M the number of servers ($M = card(J)$). We thus construct a routing table based on the cluster outputed by the K-means algorithm. First, we consider the cluster containing the costliest FQDN and denote n_1 its cardinal. FQDNs are distributed by a round-robin among the servers. The last FQDN is associated to the server $n_1 \bmod M$. This ensures that the FQDNs of the cluster containing the costliest FQDNs are spread evenly on all servers. Similarly, the first FQDN of the second cluster is assigned to the server $(n_1 + 1) \bmod M$ and the last FQDN of this cluster to server $(n_1 + n_2) \bmod M$, n_2 being the cardinal of the second cluster. We proceed in this way until the last FQDN of the fourth cluster is assigned to a server. All FQDNs of the four clusters represent around 5% of the DNS traffic, thus this algorithm is called *K-means-5%* in the remaining of the paper. The advantage of the K-means approach over the MILP based solution is that it can deal easily with large amounts of data. However, *stacking* is even faster. K-means does not outperform *stacking* because it does not produce large clusters. In fact, if clusters had a huge amount of data with similar cost, compared to the number of servers, then, the cost would be uniformly distributed among each cluster. In our case, the FQDNs with heavy costs are too specific FQDNs, and are not numerous enough compared to the number of servers. However, this technique avoids the worst case where two very costly FQDNs would be assigned to the same server.

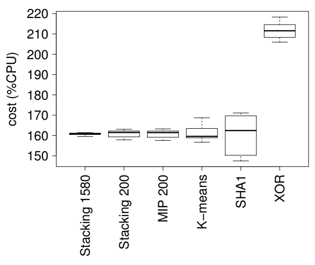
IV. Overall Performance of the Methods

Performance comparison evaluates required resources to generate the routing table, as well as the ΔCPU, which is the difference of CPU load of the servers of the *Resolving Platform*. Measurement of ΔCPU is performed through ΔQ and ΔR. The performance of the algorithms is evaluated in this section by replaying a DNS traffic capture of 10 minutes during the rush hours.
For each query, if the queried FQDN is present in the routing table, then the DNS query is resolved by the server designated by the routing table. If the FQDN is not in the routing table, then a hash of the FQDN is performed. Hash values are then uniformly assigned to the servers. Hash function makes the routing quite short, and does not require the mapping of a FQDN to a server of the platform to be explicit. In our simulation, we choose to use SHA1 [5] as the hash function, because of its good avalanche effect, cf. [16].
The best routing table is the one with the lowest ΔQ and the lowest ΔR. As average queries and responses numbers per server are the same (traffic and servers numbers are identical), the performance indicators considered are ΔQ and ΔR (must be minimal), the Cache Hit Rate (CHR, must be maximal) and the difference between maximal and minimal CHR (ΔCHR, must be minimal).
For each algorithm we represent the routing table generating the best CHR. Statistics presented in figure 3 depict the repartition of the CPU load observed. It highlights the maximum load observed (i.e. the CPU load of the most loaded server in the cluster), the minimum, the quartiles and the median estimate. This representation allows to compare how balanced

is the CPU load in the platform.
Moreover, as in subsection III-D, we decide to evaluate our routing table using only one criterion. We use the cost defined in [11] and consider the repartition of costs on servers for a 10 servers platform. We define different costs representing average CPU load during the simulation for both BIND9 and UNBOUND implementation and for both DNS and DNSSEC. Our goal is to find a routing table balancing the CPU load.
The simulations are run with a number of servers lower than the one used in a real platform and costs are evaluated on a computer which is not representative of production servers. The important parameter is not the CPU load itself but its repartition among the servers and the comparison of its value for different routing tables. It appears that *Stacking 1580* is the best algorithm as it minimizes the differences between resolution servers and results in a maximum of 2% CPU load difference between servers.
All algorithms considered based on the queried FQDNs (all but XOR) are better than the algorithm based on IP addresses (XOR) in minimizing the resources needed, by needing between 1.14 (DNS) and 1.32 (DNSSEC) times less servers.
DNSSEC increases differences between IP addresses based and FQDNs based algorithms.
Routing table length is a parameter influencing resources repartition but the algorithm used to generate the routing table is also important. For example, *K-means-5%* uses a bigger routing table than the one used by *milp-200* but does not perform better in balancing traffic.



(a) DNS



(b) DNSSEC

Fig. 3: repartition of costs (with BIND9)

V. Conclusion

FQDN-based load balancing is attractive for improving the Cache Hit Rate (CHR) and for reducing the resources needed to process the DNS(SEC) traffic on an Internet resolving platform. We can take advantage of the most popular FQDNs distribution to improve this load balancing. The few most requested FQDNs can be easily processed according to a routing table, whose size is rather small compared to the volume of traffic considered, while handling the other FQDNs is performed dynamically. The main contribution of the paper lies in the new application of optimization and machine learning methods (namely mathematical programming and clustering) in order to build these routing tables for load balancing the traffic among the platform. It appears that the heuristic-based algorithm leads to the best routing table as it minimizes the differences between resolution servers and results in a maximum of 2% CPU load difference between servers. Also, all the load balancing methods considered based on the queried FQDNs are better than the one based on IP addresses in minimizing the resources needed. Actually, they need between 1.14 (DNS) and 1.32 (DNSSEC) times less servers. A compromise must be found for the appropriate size of the routing table: the bigger it is, the better is the CHR and the balance between servers, but the slower is its generation.
Further works include the optimized processing of rarely requested FQDNs and the study of the robustness for the proposed load balancing techniques.

References

[1] P. V. Mockapetris, *Domain names - implementation and specification*, RFC 1035 (Standard), IETF, Nov. 1987.

[2] Nortel Networks, *Alteon OS 21.0, Alteon Application Switch*, Part Number: 315394-D.01, Sep. 2003.

[3] D. Migault and M. Laurent, *How DNSSEC resolution platforms benefit from load balancing traffic according to fully qualified domain name*, Proc. of CSNA, 2011.

[4] S. Francfort, D. Migault and S. Senecal, *A bi-objective Mixed Integer Linear Program for load balancing DNS(SEC) requests*, Proc. of DNS EASY, 2011, extended version in International Journal of Critical Infrastructure Protection, Elsevier, 2012.

[5] National Institute of Science and Technology, *Secure Hash Standard*, Federal Informational Processing Standard (FIPS), USA, Apr. 1993.

[6] T. Bourke, *Server Load Balancing*, O'Reilly, 2001.

[7] Y. M. Teo and R. Ayani, *Comparison of Load Balancing Strategies on Cluster-based Web Servers*, Simulation, 77(5-6):185-195, 2001.

[8] X. Jiang, J. Du and A. Bai, *The Design and Research of Smart DNS Applied in ISP*, Recent Advances in Computer Science and Information Engineering, Lecture Notes in Electrical Engineering, Vol. 127, Springer, 2012.

[9] L. Breslau, C. Pei, F. Li, G. Phillips and S. Shenker, *Web caching and Zipf-like distributions: evidence and implications*, Proc. of INFOCOM, 1999.

[10] J. Jung, E. Sit, H. Balakrishnan and R. Morris, *DNS performance and the effectiveness of caching*, Proc. of IMW, 2001.

[11] D. Migault, C. Girard and M. Laurent, *A Performance view on DNSSEC migration*, Proc. of CNSM, 2010.

[12] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley and Sons, 1998.

[13] GNU, *GNU Linear Programming Kit*, http://www.gnu.org/s/glpk.

[14] J. Kogan, *Introduction to Clustering Large and High-Dimensional Data*, Cambridge University Press, 2006.

[15] E. Alpaydin, *Introduction to Machine Learning*, MIT Press, 2010.

[16] H. Feistel, *Cryptography and Computer Privacy*, Scientific American, 228(5):15-23, 1973.