PREFETCHing to ease DNSSEC deployment over large Resolving Platforms

Daniel Migault*, Stéphane Sénécal*, Stanislas Francfort*, Emmanuel Herbert† and Maryline Laurent†
* Francetelecom, firstname.lastname@orange.com

† Institut Mines-TELECOM, UMR CNRS 5157 SAMOVAR, firstname.lastname@telecom-sudparis.eu

***Abstract*—Todays' DNS resolving platforms have been designed for DNS with fast and light resolutions over the Internet. With signature checks and larger payload, experimental measurements [10] showed that DNSSEC resolutions require up to 4 times more CPU.**
While DNSSEC requires more CPU than DNS, this paper proposes $PREFETCH_X$, an architecture that optimizes the use of CPU and thus avoids that resolving platforms increase their size for DNSSEC migration. Current resolving platforms IP_{XOR} split the traffic between the nodes according to the IP addresses. Alternatively, $PREFETCH_X$ takes advantage of the FQDN's popularity distribution (Zipf), a layered cache and cache sharing mechanisms between the nodes and requires at least 4 times less nodes. Furthermore $PREFETCH_X$ does not impact the network infrastructure which eases its deployment. Then, defining X the number of prefetched FQDNs makes $PREFETCH_X$ highly scalable and flexible to different type of traffic.

***Index Terms*—DNS, DNSSEC, DHT, Pastry, Cache sharing**

I. Introduction

Internet Service Providers (ISPs) have always hosted Domain Name System (DNS) resolution platform for its end users and Machine-to-Machine communications. However, little attention was paid to optimize DNS resolving platform. Therefore, most of them are currently composed of a load balancer —like [12]—that splits the traffic between the platform's nodes according to the IP addresses of the incoming queries. This architecture is named IP_{XOR} and is used to compare the proposed alternative architectures.

In fact, DNSSEC resolution involves larger payload than DNS and signature checks that require more CPU. Although this CPU overhead depends on the type of traffic, the server's implementation, the resolution policy or the hardware, Migault and al. Experimentally measured in [10] this overhead to be up to 425%. This has been confirmed in real deployment, mainly because DNSSEC generates much more exchanges. In addition, the demand for DNS resolutions keeps on increasing and ISPs can observe roughly a monthly 8% increase rate. With Contend Delivery Network and short Time to Live (TTL) DNS responses, we expect the DNS traffic to keep or increase its growth rate.

As ISPs can hardly afford increasing their resolution platform by 5 and as DNS traffic is still expected to increase, it is now crucial to optimize resolving platform to face the future demands on DNS and DNS(SEC) evolutions. This paper proposes the $PREFETCH_X$ architecture, designed to:

1) Reduce the number of nodes (i.e. CPU or resources).
2) Provide management facilities for Operation, Administration & Management (OAM) such as overcoming nodes fail-over or addition of a node.
3) Deliver limited impact on the core network with a straight forward design to ease its deployment.

$PREFETCH_X$ takes advantage of the Zipf distribution of the Fully Qualified Domain Names (FQDN). This distribution means that a large part of the traffic is constituted by a small amount of FQDNs. In this paper we call $HEAD_X$, the X most popular FQDNs and $TAIL_X$ the remaining FQDNs. $PREFETCH_X$ takes also advantage of a layered cache, by prefetching the $HEAD_X$ FQDNs in a dedicated cache and pre-processing each DNS query before they are sent to the server's software. More specifically, for each incoming DNS query, we check whether the queried FQDN is in $HEAD_X$. Only if the query is not in $HEAD_X$, it is forwarded to the traditional DNSSEC server's software — [1], [18]. We typically expect the dedicated cache and its lookup process to be hosted in a Network Hardware Acceleration Cards (NHAC). The length of the cache X is a trade-off between fast lookup, a large cache hit rate (CHR), and a uniform distribution of the necessary CPU to deal with $TAIL_X$ among the node of the platform. X is the variable for adjustment to find this trade-off with multiple type of traffic, making $PREFETCH_X$ highly flexible.

$PREFETCH_X$ takes advantage of cache sharing mechanisms. This means that a given FQDN should be resolved by a single or a small set of nodes, mainly to avoid duplicated resolutions. In this paper, we considered for $TAIL_X$ different cache sharing mechanisms on top of *Pastry* [15]. Not only *Pastry* avoids duplicated resolutions, but also provides OAM facilities. Note that the advantages provided by *Pastry* mainly apply to DNSSEC and not DNS. In fact, with *Pastry* based architectures, an intra-node communication occurs instead of a resolution. Because DNSSEC resolution requires signature checks, an intra-node communication costs much less than a resolution. This is not so obvious with DNS.

The remaining of this paper is organized as follows. Section II positions the paper towards existing work. Section III details motivations for $PREFETCH_X$ and its

design. As mentioned above, X is defined by the traffic in order to find a trade-off between cache efficiency and a uniform CPU distribution between the nodes. In this paper we considered 10 minute DNS live captures from a large ISP, with more than 35 millions DNS queries. In section IV we show that setting $X = 2000$ FQDNs results in a uniform CPU distribution among the nodes of the platform for the remaining $TAIL_X$ FQDNs. Section V, VI and VII are dedicated to the cache sharing architectures that deals with $TAIL_X$. Section V, defines analytic models of various cache sharing architectures. Section VI computes these models with a live traffic capture, and compares their efficiencies. At last, section VI compares efficiency of the *Pastry* model with those provided by an experimental platform based on FreePastry [5] to validate our analytic models and the thus our considerations for real deployment.

II. Related Work

Many works focus on Web caching architectures. Wang in [19] describes different caching architectures, providing inputs on where and how placing the cache devices according to the Web requirements. Tewari and al. introduce a cost comparison to different distributed caching methods in [17]. Wolman and al. investigate the benefits of cooperative caching in [21], and demonstrates that performance are reached only with a limited number of nodes. The DHT web caching methods are investigated by Iyer and al. with Squirrel [6], which is based on Pastry [15]. This paper compares two architectures, one using a home node dedicated to a bench of web pages and one with a home node that can also delegate the resolution to other nodes during a flash crowd. It happens that the less flexible but simpler architecture have better performance.

Massey and al. in [9] and Risson and al. in [14] analyze how DHT could enhance the robustness of the Naming System. The robustness of both Chord and DNS considers *Data failure rate*, *Path failure rate* and *Path length*. The DNS efficiency was proved to be linked to the popularity of its zone and the number of labels of the domain name, whereas the DHT efficiency is related to the popularity of its RRsets. In fact, DHT main drawback is its heavy routing algorithms. DHT is also more robust to orchestrated attacks and could achieve the same availability of the current DNS with added mechanisms like proactive caching —Beehive [13].

Our paper differs from the papers cited above since the DHT ring is used for hosting authoritative data, whereas our architecture uses the DHT Pastry as a way to define the node responsible for performing the resolutions. The way the data is stored into the DHT also differs from DHash and we use PAST. In DHash, blocks of the files are spread over the DHT nodes, whereas in PAST the whole file is hosted on the node. Our architecture is also expected to consider at maximum around one or two hundred nodes, whereas the DHT described by Cox and al. [3] is a 1000 node experiment which is mentioned as being a restricted number.

III. PREFETCH Architecture: Goals and Design

As mentioned in section I, this paper introduces the design of an architecture that makes DNSSEC deployment feasible for a resolving platform. Current resolving platforms IP_{XOR} are based on a load balancers that split the DNS queries by XORing their IP addresses —like with [11]. Our target architecture must (1) reduce the involved resources, (2) provide OAM and (3) enjoys a straight forward design.

A. Global resource reduction of IP_{XOR} with $FQDN_{SHA1}$

The resource we care in this paper is the CPU which is heavily used for DNS resolution with signature checks and cache lookup over large caches. Thus, our goals are to reduce (a) the number of resolutions, (b) the size of the cache and (c) the number of cache lookups. Then, to prove the architecture is efficient, we must check the CPU is uniformly distributed among the nodes of the platform.

We first consider $FQDN_{SHA1}$, an architecture that assigns each incoming FQDN to a specific node according to the hash of its FQDN. As a result, a given FQDN is resolved and stored by a single node which avoids simultaneous resolutions and partly reduces the cache size. More specifically, the cache size is reduced for FQDNs that are queried more than once. The problem with caches in IP_{XOR} is that they are very large and filled with infrequently queried FQDNs. This increases the necessary CPU for cache lookup without really providing an advantage of the caching mechanism. Finally $FQDN_{SHA1}$ addresses goals a), b) and c).

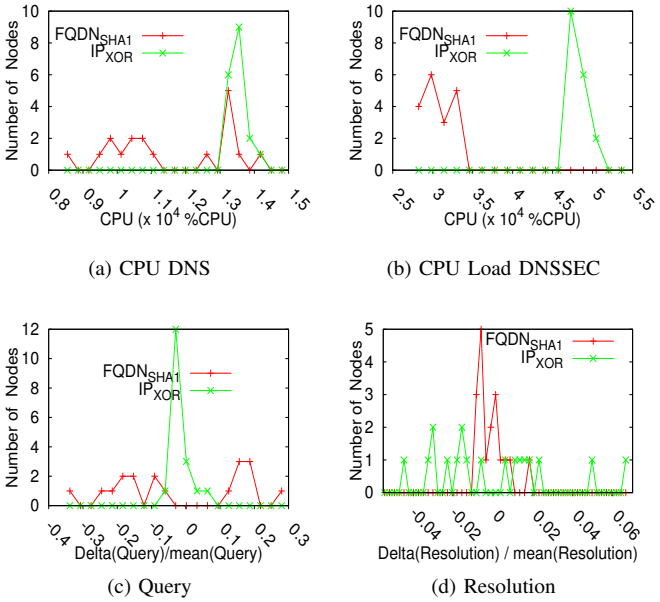
B. Caching to make $FQDN_{SHA1}$ CPU distribution uniform

The problem faced by $FQDN_{SHA1}$, is that resources are non-uniformly distributed among the nodes of the platform. With a 10 minute DNS capture of more than 35 million queries, figure 1 depicts how the CPU cycles, queries and resolutions are distributed among the nodes. Our live traffic is captured on a 18 node platform, so we replay it for a 18 node platform, and plot how many nodes are associated to a given value of CPU cycles, query and resolution numbers. The CPU cycles associated to resolutions are provided by experimental measurements in [10]. Figures 1a and 1b show that $FQDN_{SHA1}$ globally reduces the CPU by 30%, compared to IP_{XOR}. On the other hand CPU are unfairly distributed among the nodes. Similarly, figures 1c and 1d depict the relative dispersion of the number of queries (δ_Q), respectively resolution (δ_R) as defined in equation (2) with Q_i the number of queries on node i and $\bar{Q}$ the mean query number over the platform. From figures 1c and 1d, $\delta_Q^{FQDN} \approx 5\delta_Q^{XOR}$ and $\delta_R^{FQDN} \approx 0.026\delta_R^{XOR}$. Finally, $SHA1$ does not provide any benefit over XOR in term of load balancing, which confirms the use of XOR for load balancing the DNS traffic. $FQDN_{SHA1}$ is adapted for

balancing resolution whereas IP_{XOR} is more adapted for query load balancing.

$$\delta_Q = |MAX(\hat{Q}_i) - MIN(\hat{Q}_i)|, i \in [1..n] \quad (1)$$

$$\hat{Q}_i = \frac{\mid Q_i - \bar{Q} \mid}{\bar{Q}} \quad (2)$$



(a) CPU DNS (b) CPU Load DNSSEC

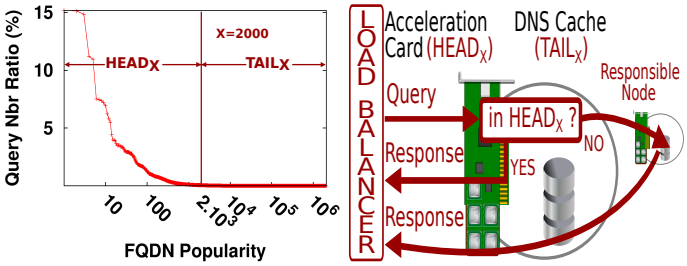
(c) Query (d) Resolution

Fig. 1. Resource Distributions on a 18 node platform

The non-uniform distribution of the resources results from a large inequality between the FQDN's popularity as depicted in figure 2a. Accordingly to the Zipf distribution of FQDN's popularity, to counterbalance the query number of FQDN with popularity i, a large number (K) of FQDNs with popularity $i + k, k \in K$ must be involved. This is especially true for FQDNs in $HEAD_X$, so, to prevent these FQDN to unbalance the resources distribution over the platform, we consider a pre-process that handles FQDNs belonging to $HEAD_X$. Thus, only the remaining FQDN of $TAIL_X$ will be handled by the DNS(SEC) server, and the cache sharing architecture must be designed for $TAIL_X$.

As represented in figure 2b, for each incoming DNS query, the query is pre-processed. If the queried FQDN is in $HEAD_X$, the response is directly sent back, otherwise, the query is forwarded to the node to be resolved.

The key advantage provided by this pre-processing step is that a large part of the traffic is querying FQDNs in $HEAD_X$, and X is a relatively small number. This makes cache lookup much faster than when handled by the DNS(SEC) server with a large red-black binary tree cache of regular DNS server implementations —BIND [1], UNBOUND [18]. More specifically, we expect the pre-processing part to be computed in Network Hardware Acceleration Card (NHAC) [2], [4]. The key challenge of the $PREFETCH_X$ is to derive X from the DNS traffic so that the resources are uniformly distributed among the nodes of the platform. Section IV proved that that X as little as $X = 2000$ provides a uniform distribution of queries, resolutions, and so CPU.



(a) X, $HEAD_X$, $TAIL_X$ (b) PREFETCH Node Architecture

Fig. 2. PREFETCH Node Parameters

C. Providing OAM via DHT

Because $PREFETCH_X$ prefetches $HEAD_X$, the DNS server software only deals with FQDNs of $TAIL_X$. Each of these FQDN is being assigned to a specific node as with $FQDN_{SHA1}$. Thus, it becomes convenient to place the DNS(SEC) resolving servers on the top of a Distributed Hash Table (DHT) infrastructure. In this paper, we consider Pastry [15] that provides auto-configuration when a node is added or removed from the platform. Note that we only use a subset of the Pastry functionalities. More specifically, we do not use the *Pastry Dynamic Node Discovery* that defines which Pastry node hosts a given content. This mechanism is quite complex because DHT protocols have been designed for billions of dynamic nodes, making impossible for each node to have a global knowledge of the platform. On the other hand, our platform is not expected to have more than a (few) hundred nodes, all administrated by the same ISP. This makes possible for each node to consider the other nodes as its *Neighbors*, and thus does not require *Dynamic Node Discovery*. On the other hand, we take advantage of the *content distribution* among the nodes of the platform and the *auto-configuration* Pastry mechanisms. Figure 2b shows that when the queried FQDN is not in $HEAD_X$, the query is forwarded to the DNS(SEC) server on top of Pastry. The Pastry protocol forwards the query to the responsible node for that FQDN. The Pastry infrastructure for $TAIL_X$ is detailed in section V.

IV. Deriving X, $HEAD_X$, $TAIL_X$ from Live Capture

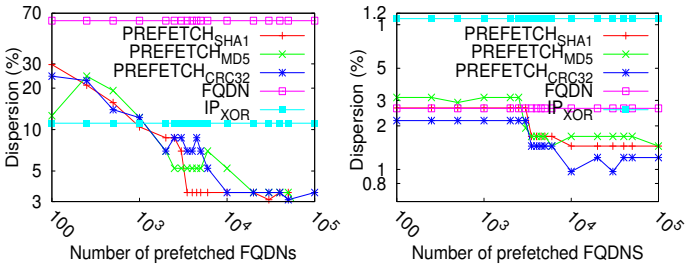
Section III defined $PREFETCH_X$, and showed that caching X most popular FQDN in a NHAC results in an uniform distribution of the number of queries, resolutions and so the CPU. This section derives X, —and so $HEAD_X$ and $TAIL_X$ —from a live DNS traffic capture. We define X so that the necessary resources to handle $TAIL_X$ are uniformly distributed among the nodes. The distribution must remain uniform when different hash functions are used in the DHT, and at any time of the day. Finally, we show that

$PREFETCH_{2000}$ only needs half of the resources required by IP_{XOR}.

A. X for δCPU_X hash function stability

Equation 2 in section III-B defined the relative dispersions δ_{Q_X} and δ_{R_X} and showed that resources for $TAIL_X$ are uniformly distributed when these values are small. $PREFETCH_{X_o}$ is more efficient than IP_{XOR} when X_o makes $\delta_{Q_{X_o}} \leq \delta_{Q_X^{XOR}}$ and $\delta_{R_{X_o}} \leq \delta_{R_X^{XOR}}$ —Q_X^{XOR} (resp. R_X^{XOR}) are the number of queries (resp. responses) with IP_{XOR}.

Figure 3 sums up the hash function stability results of a computation over a 10 minute DNS capture. Figures 3a and 3b depict δ_Q and δ_R for different hash functions (SHA1, MD5, CRC32) and table 3c compares them with IP_{XOR}. It clearly shows that with $X = 2000$, $PREFETCH_{2000}$ overcomes both IP_{XOR} queries and resolution dispersion. On the other hand, the comparison of CPU shows that with $X = 500$ $PREFETCH_X$ has a better CPU distribution than IP_{XOR}. Considering queries and resolutions has more constrains than considering CPU, and we will consider $X = 2000$ in the remaining of the paper.



(a) δ_{Q_X} (b) δ_{R_X}

X	$\frac{\delta_{Q_X}}{\delta_{Q^{XOR}}}$		$\frac{\delta_{R_X}}{\delta_{R^{XOR}}}$		X	$\frac{\delta_{CPU_X}}{\delta_{CPU^{XOR}}}$	
	min	max	min	max		DNS	DNSSEC
1000	1	1	0.19	0.28	100	2.04	0.4
2000	0.63	0.78	0.32	0.47	250	1	0.2
4000	0.31	0.625	0.13	0.15	500	0.75	0.4
					2000	0.25	0.2

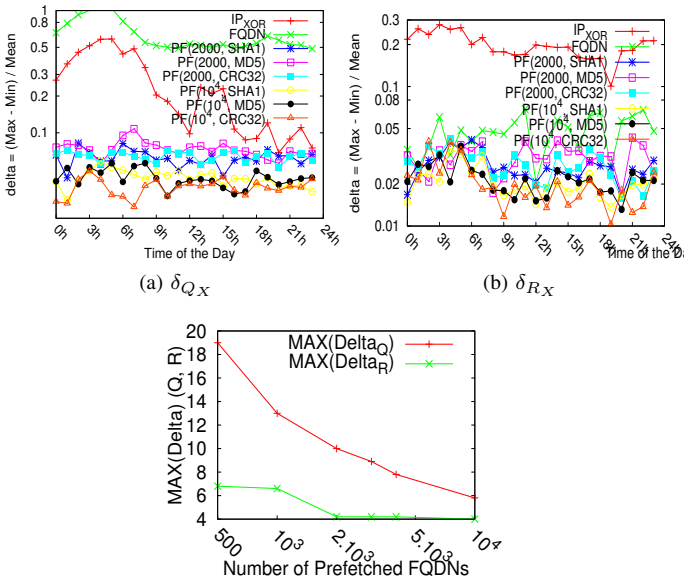
(c) $\frac{PREFETCH_X}{IP_{XOR}}$, for Q, R and CPU

Fig. 3. $PREFETCH_X$ δ_{Q_X}, δ_{R_X} measurements

B. X for δCPU_X time stability

Section IV-A derived $X = 2000$ from hash function stability. In this section, we check if this value is stable over at least a day. Figure 4 plots δ_{Q_X}, δ_{R_X} for various X values during the whole day and depicts in figure 4c for different value of X the maximum of δ_{Q_X} (respectively δ_{R_X}) observed in that day.

From figures 4a and 4b $PREFETCH_X$ provides, over time, a much more uniform distribution of the resources than IP_{XOR}. $X = 2000$ reduces variations which do not exceed a 10% variation over time, and increasing X to $X = 10000$ does not significantly improve $PREFETCH_X$ stability. This confirms our choice of $X = 2000$.



(a) δ_{Q_X} (b) δ_{R_X}

(c) $MAX(\delta_{Q_X}), MAX(\delta_{R_X})$

Fig. 4. Time Stability

In section III, we exposed our motivations for a new architecture for a DNSSEC resolving platform and designed $PREFETCH_X$. This architecture requires to set X, the number of most popular FQDNs that are cached and pre-processed by NHACs. X must be set according to the DNS traffic and section IV details how to derive X and shows that on our live DNS capture $PREFETCH_{2000}$ provides a uniform distribution of the resources —queries, resolutions and so CPU —. These distributions remain stable to different hash functions and over time. In term of efficiency, $PREFETCH_{2000}$ roughly requires half of the nodes required by IP_{XOR}.

The second part of the paper focuses on the DHT architecture for $TAIL_X$. Section V models various DHT configurations. Theseanalytic models lead to simulation with real traffic in section VI to define which architecture best fits $PREFETCH_X$ goals. At last, we validate these models in section VII with a real implementation based on FreePastry.

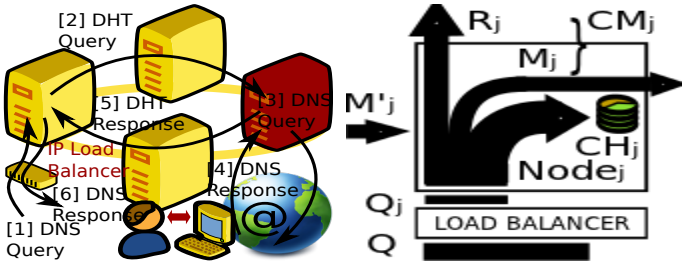
V. DHT Theoretical Analytic Models for $TAIL_X$

In DNS over Pastry [15], as represented in figure 5a, queries are identified according to the requested FQDN. Each node is assigned a set of FQDNs and resolves queries it is a *Responsible Node* (*RN*) for. Other queries are forwarded to their associated RN.

Pastry is widely known by the community, but other DHT protocols may be used - like chord or Tapestry. The way we

use Pastry differs from what it was originally designed for. First, the Pastry nodes constitute the platform and belong to the same administrator, they are located in the same data center, on the same LAN and every node knows all the other nodes - the platform is not expected to be larger than a few hundreds nodes. We do not assert that the node ID is derived from the data by a simple hash function (SHA1), but we may apply a specific mapping known by all nodes. Finally, we do not consider the Pastry routing discovery algorithm. On the other hand, we take advantage of Pastry's auto-configuration mechanisms, robustness to DoS attacks [3], [9], [14] - as such it may balance the sensitivity to DoS attacks introduced by DNSSEC with heavier resolutions. We also take advantage of the cache sharing mechanisms for enhancing Pastry based platforms. Note that how FQDNs are associated to their *Responsible Node* (RN) may require the auto-configuration mechanism to be reconsidered. Furthermore, robustness to DoS requires architectures that cache somehow the responses. This section provides a model description for Pastry based architectures, with different mechanisms that either reduce routing traffic, or enhance the cache of the platform. Section VI computes the models and compares the different pastry based architectures with *FQDN* and IP_{SHA1}.

From figure 5b, the traffic has the following features:



(a) *Pastry* (b) Node J

Fig. 5. DHT Traffic and Node Modelization

- Q: DNS queries sent by all end users with query rate q.
- Q_j: DNS queries sent to n_j with query rate q_j.
- $M_j^{'}$ (resp. M_j): incoming (resp outgoing) exchanges between n_j and the other $n-1$ nodes of the n node platform.

The different entities we consider are:

- $p(IPs, fqdn)$: a DNS query with $IPs = (IP_{source}, IP_{destination})$, and $fqdn$ the queried FQDN as defined below.
- $fqdn(name, ttl, rank)$: a FQDN where $name$ designates the FQDN, ttl its TTL and $rank$ its rank which reflects its popularity. Note that $rank$ is based on the popularity of the FQDN, i.e. the query rate value associated to that FQDN. Each FQDN has a distinct rank, that is, two FQDN with the same request rate, are ordered alphabetically.

The following architecture components are : a Load Balancer that splits the traffic among the different nodes n_j of the platform. We note :

- $\mathcal{F} = \{$the ranked $fqdn_r,\ r \in \mathcal{R}\}$. The list of different FQDN with their associated rank r.
- OT defines the Occupancy Time (OT) that depends on the action performed (cache lookup OT^H, cache insertion OT^R, query forwarding OT^{FWD}...) and the considered protocol (DNS, DNSSEC).

The different probabilities considered for each node are :

- L_j the probability n_j receives a packet p of Q: $q_j = L_j.q$.
- $H_{r,j}$ the probability n_j is RN for $fqdn_r$.
- $\Phi(r)$ the probability $fqdn_r$ of rank r is queried.
- CM (resp. CH) the platform probability for a Cache Miss (resp. Cache Hit) , i.e. the response is not stored in any cache node $\mathcal{C}_j$ of the platform.
- CM_j (resp. CH_j) the probability a packet addressed to n_j triggers a Cache Miss (resp. a Cache Hit).
- R_j the probability p in Q_j triggers a resolution over the Internet on n_j.
- M_j the probability p in Q_j is forwarded by n_j .

Note that we have $CM + CH = 1$, $CM_j + CH_j = 1$ and $CM_j = M_j + R_j$, so an architecture is fully characterized by CM, R_j and M_j.

A. Single Node Model: τ_r

Let us consider the case where a DNS traffic Q is addressed to a single node. In our model, we consider as in [8] a constant TTL for the FQDNs and a constant query rate q measured in $queries.s^{-1}$. Let CM_o be the experimental Cache Miss measured on our live capture. We define τ_r the time (s) so that measured $CM_o = \sum_{r\in\mathcal{R}}(1-\Phi(r))^{q.\tau_r}.\Phi(r)$. For each $fqdn$ with associated rank r, a Cache Miss (CM) occurs if the $fqdn$ is queried at t and has not been queried during the last TTL seconds. $\Phi(r)$ represents the probability the FQDN is queried, and $(1-\Phi(r))$ the probability it is not being requested. If $q\tau_r$ represents the number of queries received during $fqdn.ttl$, then $(1-\Phi(r))^{q.\tau_r}$. represents the probability the FQDN has not been queried during $fqdn.ttl$. As such, τ_r is the mean time a FQDN is cached in the live capture. The empirical definition of τ_r can also be expressed in a theoretical way as a function of T the duration of the capture, and the various $fqdn.ttl$s of the FQDNs. We clearly have $R_o = CM_o$ and $M_o = 0$.

B. $\mathbf{IP_{SHA1}}$ Architecture

If we assume that fqdn and IP addresses are independent, then $CM = \sum_{i=0}^{n-1} L_i.CM_i$. Because nodes are equal, $CM_j = CM$, and $CM_j = \sum_{r\in\mathcal{R}} \Phi(r).(1-\Phi(r))^{L_j.q\tau_r}$, $R_j = CM_j$, $M_j = 0$ as node do not forward incoming traffic. Finally:

$$OT(q_j) \quad = \quad q_j.OT^H + q_j.CM_j.OT^R \qquad (3)$$

C. FQDN Architecture

In *FQDN* we assume that load balancers redirect each $fqdn_r$ to its RN. We also considered that our FQDN mapping results in a uniform distribution of queries, resolutions and

homed FQDNs. Then $CM = CM_o$, $CM_j = CM_o$, $R_j = CM_j$ and $M_j = 0$. Finally:

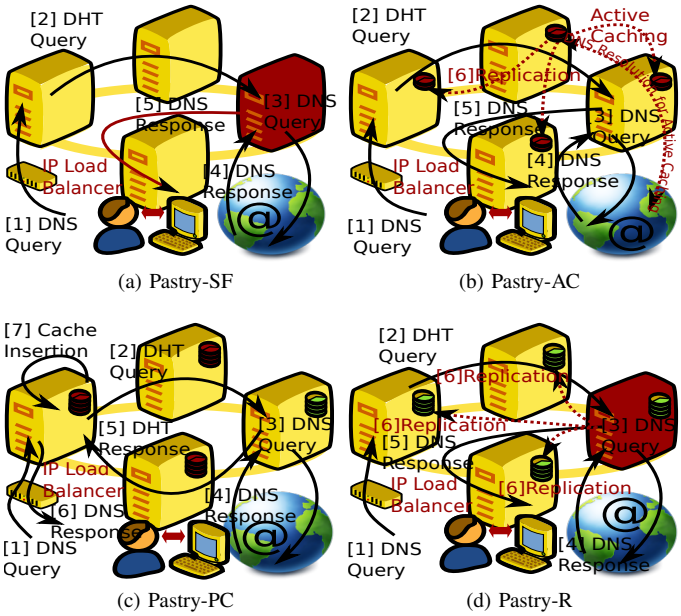
$$OT(q_j) = q_j.OT^H + q_j.CM.OT^R \quad (4)$$

D. Pastry *Architecture (no cache, no replication)*

Pastry architectures are of interest since Pastry [15] is quite robust to DoS attacks [3], [9], [14], nodes are self organized, and can be deployed as a flat architecture without modifying the core network infrastructure. In Pastry, nodes route queries to their RN node and resolves the queries they are RN for. The challenge is to find out how managing the traffic balances the number of avoided DNS(SEC) resolutions. There are various ways to manage the incoming traffic. In this section we consider the following mechanisms :

- No cache no Replication (*Pastry*): When a node receives a query it is not *RN* for, a query response exchange is performed with the *RN* node.
- Stateless Forwarding (*Pastry-SF*): It works like *Pastry* except that the *RN* node sends directly the response to the end user, rather than to the *Pastry* node.
- Passive Caching (*Pastry-PC*): It works like *Pastry* except that nodes cache the responses.
- Replication (*Pastry-R*): It works like *Pastry*, but when RN node performs a resolution, it provides a copy of the response to k neighbours that cache them.
- Active Caching (*Pastry-AC*): It works like *Pastry-SF* but takes advantage of the FQDN's Zipf distribution. RN provides the response for their γ most popular FQDN to all other nodes of the platform.

Pastry, *Pastry-SF*, *Pastry-PC*, *Pastry-R* and *Pastry-AC* are illustrated in figures 6.



(a) Pastry-SF (b) Pastry-AC

(c) Pastry-PC (d) Pastry-R

Fig. 6. DHT Architecture Principles

Thus for *Pastry*, $CM = CM_o$, $CM_j = \sum_{r\in\mathcal{R}} \Phi(r)(1 - H_{r,j}) + H_{r,j}.\Phi(r).(1-\Phi(r))^{q\tau_r}$, $R_j = H_{r,j} \sum_{r\in\mathcal{R}} \Phi(r).(1 - \Phi(r))^{q\tau_r}$, $M_j = (1 - H_{r,j})\Phi(r)$. With OT^{FWD} the OT required to forward a packet we have :

$$\begin{aligned} OT(q_j) = {} & q_j.(CH_j + M_j.CH).OT^H + \\ & q_j.M_j.OT^{FWD} + \\ & q_j.(R_j + M_j.CM).OT^R \end{aligned} \quad (5)$$

$OT^{FWD} \approx OT^H$ as we consider a routing table lookup on a small table, forwarding the query to the RN node and then forwarding the response to the end user. OT^H considers reading the DNS query, but this might not be necessary, and redirection may be based on reading and hashing a fixed number of bits at a defined position, as routers do with IP addresses.

E. Pastry – Stateless Forwarding (Pastry – SF): *(no cache, no replication)*

SF-Pastry n_j sends the response directly to the end user rather than to the pastry node that has forwarded the query. Similarly to *Pastry* in section V-D:

$$\begin{aligned} OT(q_j) = {} & q_j.(CH_j + M_j.CH).OT^H + \\ & q_j.M_j.OT^{FWD'} + \\ & q_j.(R_j + M_j.CM).OT^R \end{aligned} \quad (6)$$

$OT^{FWD'} \approx \frac{OT^H}{2}$, since to the difference with OT^{FWD} there is no response - which is much larger than query.

F. Pastry – Active Caching (Pastry – AC): *(no replication)*

Active Caching [13] takes advantage of the Zipf distribution of the FQDNs. Each node informs all other nodes of the γ most popular RNd FQDNs. In our model, nodes responsible for the γ FQDNs are proactive, which means that no cache miss occurs for those FQDNs. If we consider that all FQDN have the same TTL (as in [8]), during TTL, n_j sends γ responses to all $n-1$ nodes and receives the γ most popular FQDNs from all $n-1$ nodes. As a consequence, a DNS query with rank greater than $n.\gamma$ follows the *Stateless Forwarding* Pastry resolution procedure. As such, with $CM = \sum_{r\in\mathcal{R}}^{r>n.\gamma} \Phi(r)(1 - \Phi(r))^{q.\tau_r}$,
$R_j = \sum_{r>n.\gamma} H_{r,j}\Phi(r)(1-\Phi(r))^{q.\tau_r}$ and $M_j = \sum_{r>n.\gamma}(1 - H_{r,j})\Phi(r)$. Finally:

$$\begin{aligned} OT(q_j) = {} & q_j.(CM_j + M_j.CH)OT^H + \\ & q_j.(R_j + M_j.CM).OT^R + \\ & q_j.M_j.OT^{FWD'} + \frac{\gamma}{\tau_r}.OT^{R_{AC}} + \\ & OT^{FWD_{AC}}\left(2.\frac{\gamma.(n-1)}{\tau_r}\right) \end{aligned} \quad (7)$$

$OT_j^{FWD'} \approx \frac{OT^H}{2}$ as in equation 6. For resolution of Active Cached FQDNs, $OT_j^{R_{AC}} \approx OT^H$. $OT_j^{FWD_{AC}} \approx n.OT_{DNS}^R$ since cache updates are sent in one block, and the cache to be updated is quite small. Optimization may also consider different levels of caches, so to improve cache lookup and interactions between $\mathcal{C}_j$ and the *Active Cache*.

G. $\mathbf{Pastry-PassiveCaching(Pastry-PC)}$: *(cache, no replication)*

With Passive Caching node proceeds as in the regular Pastry, but keeps the response in its cache. Similarly to *Pastry* $CM = CM_o$, $R_j = \sum_{r\in\mathcal{R}} \Phi(r).H_{r,j}.(1-\Phi(r))^{q.\tau_r}$ and $M_j = \sum_{r\in\mathcal{R}}(1-H_{r,j}).\Phi(r).(1-\Phi(r))^{q.L_j.\tau_r}$. Finally:

$$\begin{aligned} OT(q) &= q_j.(R_j + M_j.CM).OT^R + \\ & \quad q_j.(CH_j + M_j.CH).OT^H + \\ & \quad q_j.M_j.OT^{FWD''} \end{aligned} \tag{8}$$

$OT^{FWD''} \approx OT^R_{DNS}$ as insertion is performed on a large cache.

H. $\mathbf{Pastry-Replication(Pastry-R)}$: *(no cache, replication)*

Replication with no cache works like the regular Pastry architecture with no cache except that when a resolution is performed the response is replicated on k neighbors. This mechanism is close to the active caching mechanism to the extent that all FQDNs will be replicated. On the contrary, in active caching one only cares about the most popular FQDNs, then replication occurs only to k nodes, whereas Active Caching replicates the responses on all nodes. When node n_j does not have the response in its cache, it proceeds as in Pastry (cf. section V-D). *Pastry-R* provides $CM = CM_o$, $R_j = \sum_{r\in\mathcal{R}} \Phi(r).H_{r,j}.(1-\Phi(r))^{q.\tau_r}$, and $M_j = \sum_{r\in\mathcal{R}} \Phi(r).k.H_{r,j}.(1-\Phi(r))^{q.\tau_r} + \Phi(r).(1-(k+1).H_{r,j})$. Then, we estimate the probability of replicating or receiving a replication from a peer $PC^{FWD} = 2kR_j$. Finally:

$$\begin{aligned} OT(q_j) &= q_j.(CH_j + M_j.CH).OT^H + \\ & \quad q_j.(R_j + M_j.CM).OT^R + \\ & \quad q_j.M_j.OT^{FWD'} + q_j.2.k.R_j.OT^{FWD_{PC}} \end{aligned} \tag{9}$$

$OT^{FWD'}_j \approx \frac{OT^H}{2}$ (cf. equation 6). $OT^{FWD_{PC}}_j \approx \frac{1}{2}OT^R_{DNS}$ since cache update is performed, but no response is sent nor cache lookup performed.

VI. Executing DHT models with $TAIL_X$

This section computes the distribution $TAIL_{2000}$ over various DHT architectures modeled in section V. The efficiency of a DHT architecture depends on the relative cost of caching versus requesting another node. Caching may result in large expensive caches whereas requesting may result in heavy network operations. Because cache length's impact on performance is hard to estimate, in our evaluation we assume that all cache operations have the same cost over the various DHT architectures. This is true as long as the cache remains small or of the same size.

Figure 7 exhibits how traffic variations impacts the CPU ratio of the DHT architectures to IP_{XOR}. Architectures based on routing provide better performances, and are more stable to TTL or query rate variations —with constant number of FQDNs. Thus *Pastry-SF* is recommended. *Pastry-PC* and *Pastry-R*, especially for DNSSEC, take advantage of large TTL values and large query Rate because the number of FQDNs remains constant in our computations. Hence, increasing TTL or query rates results in increasing the CHR.

Figure 7c shows that $R_{CPU} = \frac{CPU_{Resolution}}{CPU_{Cache}} \geq 20$, DHT provides a clear advantage over IP_{XOR}, which remains stable for greater values. [10] measured on UNBOUND $R^{DNS}_{CPU} = 3.74$ and $R^{DNSSEC}_{CPU} = 38.69$, which shows *Pastry-SF* only requires 55% of IP_{XOR} resources for DNS and 19.2% for DNSSEC.

Figures 7a shows that with $TAIL_{2000}$, TTL above 100 have small impact on the platform. Furthermore, routing packet is lighter than caching operations, which makes *Pastry-SF* and *Pastry* the recommended architectures for $TAIL_X$. Since our simulations run with a constant number of FQDNs, it reduces the Cache Hit Rate (CHR), and increasing the query rates in figure 7b increases the Cache Hit Rate. This makes *Pastry-PC* and *Pastry-R* take much more advantage to other architecture, especially with DNSSEC. Architectures provide a balance between caching mechanisms and routing and the efficiency of an architecture really depends on the traffic characteristics.

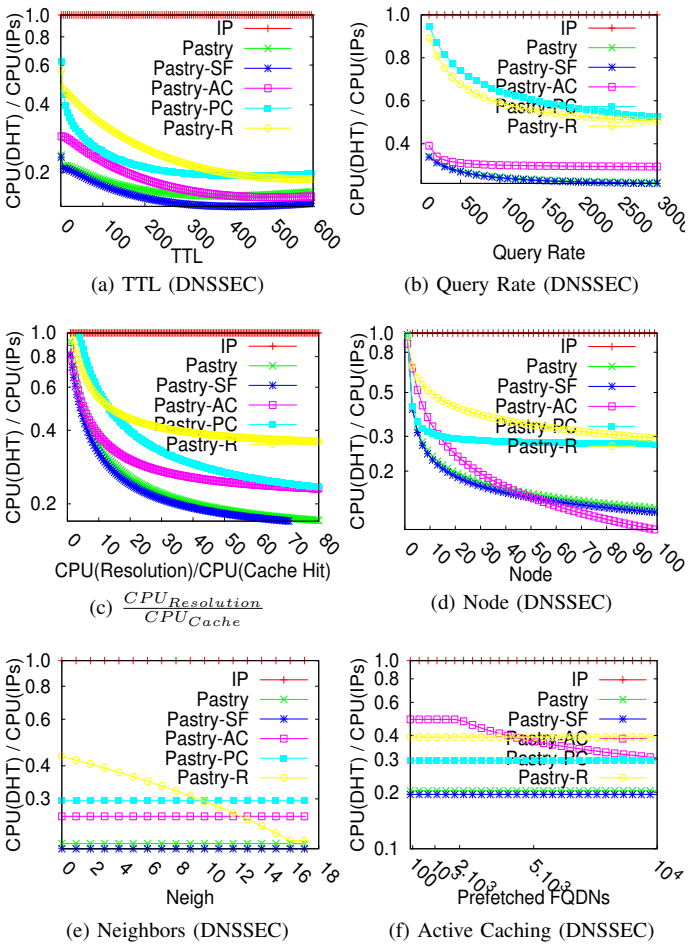
Figures 7d confirms that DHT architectures reduce by up to 60% the CPU consumption over IP_{XOR}. Then, *Pastry-AC* is clearly the most scalable architecture, and is able to lower the load by expanding the number of prefetched FQDNs. This is confirmed by figure 7f. However, *Pastry-AC* is redundant with $PREFETCH_X$ which is expected to take advantage of NHAC, and *Pastry-SF* is the most appropriated architecture in term of scalability.

Figure 7d shows that response replication proves to be beneficial with DNSSEC due to the relative costs of caching toward forwarding. However replication on multiple nodes increases the size of the cache which is not taken into account in our models. *Pastry-R* is useful in case of failover and future work should measure how starting a node with empty cache may impact the platform. If it is of importance, then the marginal cost provided by figure 7d may require to activate replication.

In section IV, we concluded that prefetching $X = 2000$ results in dividing the number of nodes of $PREFETCH$ by 2 over IP_{XOR}. In this section we show that, for the remaining $TAIL_X$ traffic, which represents around 32% of the traffic, DHT can reasonably decrease the necessary resources by 55% and 80%. This means that overall DHT reduces the necessary resources by roughly between 20% and 35% over IP_{XOR}. This confirms the 30% resource reduction provided by the $FQDN_{SHA1}$. As a result, $PREFETCH_X$ requires at least 4 times fewer nodes than IP_{XOR}. The purpose of section VII is to validate our models with experimental measurements.

VII. Free Pastry Experimentation

This section validates our models by implementing and testing a DNS platform based on FreePastry [5], results are compared to those provided by the theoretical model of *Pastry* which models Pastry without any cache and IP_{XOR}. In this section, a uniform FQDN popularity distribution is



(a) TTL (DNSSEC) (b) Query Rate (DNSSEC)

(c) $\frac{CPU_{Resolution}}{CPU_{Cache}}$ (d) Node (DNSSEC)

(e) Neighbors (DNSSEC) (f) Active Caching (DNSSEC)

Fig. 7. $TAIL_{2000}$ Models evaluation with live DNS capture

used for the experimentation and the simulation.

First, we measure the performance on a single node, and then extend the platform to 10 nodes. Our 10 node Pastry experimental platform is based on the Java FreePastry library (version 2.1alpha3) [5]. FreePastry implements Pastry [15] as the routing protocol between DHT nodes, PAST [16] implements the DHT, and GCPAST on the top of PAST implements a garbage collector to remove expired contents. DNS responses are stored in the DHT and indexed by the hash of the queried FQDN and type, the assigned GCPAST TTL is the one associated to the DNS response. DNS resolutions and interaction with GCPAST are performed with DNSJava [20]. The format of the DHT contents must be chosen carefully, and we choose to store a complete answer involving multiple RRsets, rather than each RRsets individually. The hash key is built using the name, class and type of the query field. The main drawback is that it generates a high redundancy between the RRsets contained in different DHT contents and the size of the DHT caches on the nodes. For the TTL, we assign to GCPAST the TTL provided for the ANSWER field in the DNS response.

Our experimental platform is composed of 10 Pentium III and Pentium II servers with Debian *Lenny*. Although different configurations were used, the CPU frequency varies from $500\,MHz$ to $1\,GHz$, and RAM varies from $128\,M\,bytes$ to $384\,M\,bytes$. The 10 node platform is loaded with traffic with a $T = 3$ minute TTL. We fix $CHR_o = 0.7$, so caches are pre-populated with FQDNs from a list, and 7 out of 10 queries are from this list. Tests last $T_{test} = 1$ minute so with a maximum of $Q = 1100\,q.s^{-1}$ with *Pastry*.

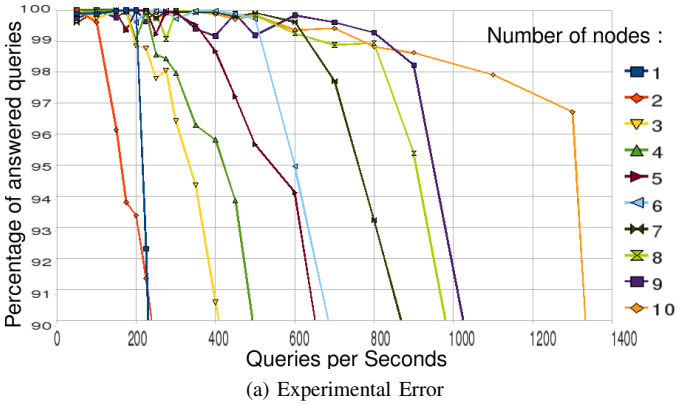
Figure 8a shows for different numbers of nodes n the percentage of answered queries. We estimated the Maximum Load (ML) reached when there are more than 2% errors. Figure 8b plots experimental measured ML, and plots the Stand Alone value (for $n = 1$, $ML(1) = 200$) as well as the linear approximation for the experimental value with $n \geq 2$: $ML^{Pastry}(n) = 114.7n - 96.9$. In a Stand Alone mode, the whole FreePastry network is supported by the same hardware, and no routing operations are required. Although in our theoretical model, we neglect here the routing table lookup, because we have reliable nodes. Our routing discovery protocol is much lighter than the one originally designed in Pastry, and all CPU estimations have been modeled by measuring a C implementation of a performance driven server: UNBOUND. In FreePastry and the DNS module we add results from academic development of a Java software, and routing table is definitely not negligible. As such, the Stand Alone mode does not provide a good estimation for the experimental IP_{XOR} measurement. In order to provide a experimental measurement for IP_{XOR}, we need to estimate the costs of routing table on a FreePastry node. $ML(n)$ estimates that routing table lookup cost to $\approx 96.9 queries.s^{-1}$, which leads to the experimental IP_{XOR} experimental measurement $ML^{IP}(n) = 200n - 96.9\,queries.s^{-1}$.

Figure 8c compares our theoretical and experimental results for $\frac{ML^{Pastry}}{ML^{IP}}(n)$. Theoretical and experimental measurements confirm that the ratio between *Pastry* and IP_{XOR} remains stable and independent of n. However, theoretical model expects *Pastry* to be equivalent as IP_{XOR}, whereas Experimental measurement shows that *Pastry* costs around twice as much as IP_{XOR}. The difference between the two results can be explained by the FreePastry implementation and the testing conditions: 1) FreePastry is not optimized for performance. Profiling the code with JRat [7] revealed that 64% of the time would be spent on insertion and DHT look up, if we were using FreePastry in synchronous mode. Using asynchronous mode leverages this bottleneck, but does not mean the code is now optimal. In addition, 2) Routing operations are much heavier than those we require in *Pastry*. When we compare the time necessary to perform a resolution in a Stand Alone configuration or in a FreePastry configuration we measure that when the DNS response is in the cache, it takes $5\,ms$ to $65\,ms$ (resp. $4\,ms$ to $300\,ms$) in a Stand Alone mode (resp. in a DHT mode). When the responses require a resolution it takes $8\,ms$ to $175\,ms$ (resp. $59\,ms$ to $342\,ms$) in a Stand Alone (resp. DHT) mode. In both cases time variation reveals that

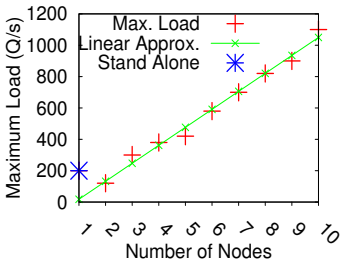
optimization may be improved in event thread synchronization of the FreePastry implementation. On the other hand, it also shows that the routing function is clearly not negligible in this implementation. At last, the lake of FreePastry and hardware performance only makes experimental measurements under quite small load which makes overhead more visible. In our case, routing overhead consists in a bit less that half of the performance.

Another way to measure the dependence between the experimental and theoretical values is to derive the sample correlation coefficient as an estimator of the Pearson correlation. For both IP_{XOR} and *Pastry*, we consider the set of experimental measurements for Maximum Load: ML_{exp}, and the set of computed values for Maximum Load: ML_{model}. The various values are those measured and computed with various values of n the number of nodes. We derive $r^{IP}_{ML_{exp}ML_{model}} = 0.9991$ and $r^{Pastry}_{ML_{exp}ML_{model}} = 0.9827$. Correlation coefficients are very close to 1 which shows a perfect positive linear relationship between the measured values and those computed from our models.

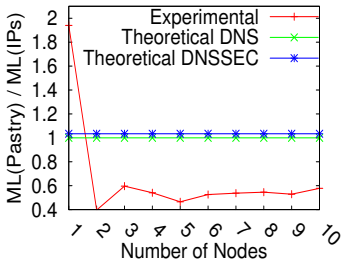
As a result, experimental measurements shows our theoretical model does not present major bias, and confirms the stable ratio of $\frac{ML^{Pastry}}{ML^{IP}}(n)$ in the testing conditions. On the other hand it also shows that a FreePastry implementation may not fill the requirements of our models, and that further investigation would need specific developments, especially to simplify the routing code and algorithms.



(a) Experimental Error



(b) Experimental Maximum Load



(c) Experimental vs Theoretical

Fig. 8. Maximum Load

VIII. Conclusion

This paper proposes the $PREFETCH_X$ architecture which prefetches X FQDNs ($HEAD_X$) and handles the remaining FQDNs ($TAIL_X$) with a Distributed Hash Table (DHT) structure. Traffic analysis sets $X = 2000$ to uniformly distribute the resource among the nodes. Prefetching $HEAD_X$ reduces the number of nodes by 2, and DHT reduces nodes for $TAIL_X$ between 55% and 80%, making $PREFETCH_X$ 4 times more efficient than the current IP_{XOR} architecture. $PREFETCH_X$'s efficiency can be enhanced by increasing X, providing an adapted light *Pastry*-like implementation. In this paper, we keep X small to limit the exchanges between the nodes to fill their cache. Large values for X require protocols and architectures optimized for cache updates. Similarly, a light *Pastry* implementation is also expected to enhance the architecture.All these aspects are let for future work.

References

[1] BIND. http://www.isc.org/bind10.

[2] cavium. http://www.cavium.com/table.html.

[3] R. Cox, A. Muthitacharoen, and R. Morris. Serving dns using a peer-to-peer lookup service. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 155–165, London, UK, 2002. Springer-Verlag.

[4] endace. http://www.endace.com.

[5] FreePastry. http://www.freepastry.org/.

[6] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *12th ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[7] JRat. The java runtime analysis toolkit http://jrat.sourceforge.net/.

[8] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. Dns performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 153–167, New York, NY, USA, 2001. ACM.

[9] D. Massey. A comparative study of the dns design with dht-based alternatives. In *In the Proceedings of IEEE INFOCOM'06*, 2006.

[10] D. Migault, C. Girard, and M. Laurent. A performance view on dnssec migration. In *CNSM 2010*, oct 2010.

[11] N. Nortel. Alteon OS 21.0, Alteon Application Switch, Sept. 2003.

[12] Radaware. Alteon Application Switch 5412 Case study, 2010.

[13] V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[14] J. Risson, A. Harwood, and T. Moors. Topology dissemination for reliable one-hop distributed hash tables. *IEEE Transactions on Parallel and Distributed Systems*, 20:680–694, 2009.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329, 2001.

[16] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, Oct. 2001.

[17] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the Internet. Technical Report TR98-0, 1998.

[18] UNBOUND. http://unbound.net/.

[19] J. Wang. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review*, 25(9):36–46, 1999.

[20] B. Wellington. dnsjava http://www.dnsjava.org/.

[21] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles*, pages 16–31, 1999.