ECO-DNS:
Expected Consistency Optimization for DNS

Chen Chen
Carnegie Mellon University
chenche1@cmu.edu

Stephanos Matsumoto
Carnegie Mellon University
smatsumoto@cmu.edu

Adrian Perrig
ETH Zurich
adrian.perrig@inf.ethz.ch

***Abstract*—The flexibility of the current Domain Name System (DNS) has been stretched to its limits to accommodate new applications such as content delivery networks and dynamic DNS. In particular, maintaining cache consistency has become a much larger problem, as emerging technologies require increasingly-frequent updates to DNS records. Though Time-To-Live (TTL) is the most widely used method of controlling cache consistency, it does not offer the fine-grained control necessary for handling these frequent changes. In addition, TTLs are too static to handle sudden changes in traffic caused by Internet failures or social media trends, demonstrating their inflexibility in the face of unforeseen events.**

To address these problems, we first propose a metric called Expected Aggregate Inconsistency (EAI), which allows us to consider important factors such as a record's update frequency and popularity when quantitatively measuring inconsistency. We then design ECO-DNS, a lightweight system that leverages the information provided by EAI to optimize a record's TTL. This value can be tuned to individual cache servers' preferences between better consistency and bandwidth overhead. Furthermore, our optimization model's flexibility allows us to easily adapt ECO-DNS to handle various caching hierarchies such as multi-level caching while considering the tradeoff among consistency, overhead, latency, and server load.

I. Introduction

Caching is an important mechanism for reducing server load. In the case of the Domain Name System (DNS), we can reduce server-side load and query latency by introducing caches, but such caching can cause inconsistency between cached copies and reference records. To enable the tradeoff between consistency and overhead, DNS proxy caching servers, or caching servers for short, employ a *Time-To-Live* (TTL) mechanism, which evicts a cached record after a specified amount of time explicitly stated in the record itself.

Unfortunately, the traditional TTL-based consistency control mechanism inherently suffers from two problems. First, TTL only bounds per-query inconsistency, while ignoring the number of queries influenced by the inconsistency. Thus if we consider the notion of *aggregate inconsistency*, which accumulates the entire inconsistency across all queries for a specific record, then in the case of highly popular records the aggregate inconsistency can become unbounded as it increases with the number of DNS queries. For example, when fake (and thus inconsistent) records are returned in a cache poisoning attack [1], a fake record for the much more popular "alwaysvisited.com" would affect many more clients than a fake record for "rarelyvisited.com" even if they have the same TTL. Hence, aggregate inconsistency more accurately reflects the impact of a single inconsistent record on clients.

Second, static owner-defined TTL values are inflexible, because such TTL values are set with limited knowledge of the topology of caching hierarchies and are not dynamically adjusted to adapt to individual caching servers. It is thus unsurprising that most DNS TTL values are chosen from a small set of values. Once set, even a TTL value configured by an ignorant domain administrator will be honored by DNS cache servers all over the Internet.

The lack of adaptive and dynamic DNS consistency control also hampers adoption of a hierarchy of DNS caching servers. In fact, in the current DNS infrastructure we have observed a reluctance to implement a DNS caching structure spanning more than two levels [2]. While a multi-level caching hierarchy could further reduce latency, bandwidth overhead and server load, it inevitably requires a more complex consistency control mechanism [3].

With the emergence of dynamic DNS (DDNS) and content delivery networks (CDNs) such as Akamai [4], DNS records are expected to be updated more frequently to rapidly adapt to changes of mapping between domain name and physical IP. Inconsistent DNS records can result in accessing a released IP address in DDNS or disrupting the load balancing of CDNs [5], hindering availability and performance. Moreover, future Internet architecture proposals such as SCION [6] propose to frequently update name server entries to reflect path changes. Such architectures require better control over name record consistency and need to achieve a low overhead to enable scalability.

However, proposals targeting to provide better DNS consistency control [7], [8] lack a standard metric to quantify consistency. Without such a metric, it is difficult to explore the tradeoff between inconsistency and other network costs such as bandwidth overhead, to compare different schemes, and to optimize caching performance.

Therefore, in this paper, we first define a new consistency metric based on aggregate consistency, which allows us to quantitatively model the tradeoff between consistency and bandwidth overhead in a hierarchial caching system. Then, based on our model, we propose a new consistency control mechanism for a DNS caching subsystem, called ECO-DNS. Our approach is to preserve the "pull-based" nature of DNS and leave the TTL value intact to maintain backwards compatibility, and at the same time automatically tune the TTLs of DNS records to optimize performance to a specified balance between consistency and bandwidth overhead.

In this paper we make the following contributions:

- we define a new metric, *Expected Aggregate Inconsistency* (EAI), which takes into account both real-time popularity and update frequency and enables a quantitative measurement of inconsistency,
- we mathematically model the tradeoff between inconsistency and bandwidth cost in multi-level caches,
- we design ECO-DNS, a lightweight, backwards-compatible DNS cache consistency control mechanism to automate setting TTLs for caching servers, and
- we simulate ECO-DNS with real DNS trace data and Internet topologies to demonstrate the performance advantage of ECO-DNS over current DNS caching schemes.

II. Model

In this section, we mathematically define our inconsistency metric, and establish an explicit relationship between the inconsistency metric and the bandwidth overhead. We then leverage multi-objective optimization to integrate these two metrics into one target cost function, which will allow us to dynamically derive the optimal TTL value. Though in this paper we focus on applying our model to DNS record caching, our model also applies to a more generalized TTL-based caching system.

A. Measuring Inconsistency

Inconsistency in DNS arises from stale records being returned by caching servers. To measure inconsistency in a DNS record, we want to answer two main questions. First, how stale is a record when it is returned? Second, how many stale records are returned overall?

To answer the first question, we want to consider a DNS record's update frequency, since in terms of consistency, receiving a record that is several updates behind is worse than receiving one that is a single update behind. As an example, if we query a cached record for an Akamai server and a cached record for an obscure unpopular site at the same rate, we should expect to receive a stale record more often for the Akamai server because it is updated more frequently. Therefore, we can measure the inconsistency of a single DNS response as the number of updates to a record between the time it was cached and the time it was returned, and expect to see greater inconsistency from records that are more frequently updated. This gives rise to the following definition:

Definition 1: Let q be a DNS query for a record r arriving at a caching server at time t_q, and let t be the time at which the queried record was cached. Then the **inconsistency** $I_r(q)$ of the response to q is

$$I_r(q) = u_r(t, t_q) \tag{1}$$

where $u_r(t, t_q)$ is the number of updates to r between times t and t_q.

To some degree, TTLs already take into account a record's update frequency. For example, Akamai's type A DNS records could have a TTL of 20 seconds, while infrequently updated sites can have TTLs as high as 86400 seconds. Therefore, even in the current system, records can be cached according to how often they are updated. However, server operators set these TTLs by hand, meaning that these TTLs only take into account the *estimated* update frequency rather than the *actual* update frequency.

The second question motivates the idea of measuring *aggregate inconsistency*. Previous work in DNS consistency protocols [7] has measured consistency simply as whether a cached copy of a DNS record is the same as the copy of the record at the authoritative server. However, we want to extend this and explore how much an instance of inconsistency affects users.

For example, a stale record for a popular site such as Google will result in much more *aggregate* inconsistency than a very stale record for an unpopular site. This is because Google receives far more queries, and thus an inconsistent record will affect many more users in the aggregate sense. Therefore, we want to consider the popularity of a DNS record, which will give us a sense of the aggregate inconsistency a stale record could cause for a given domain name.

Current TTLs also somewhat take into account the popularity of a record. For example, Google's DNS records have a TTL of 300 seconds, which is much lower than those of less popular sites. However, sites with high TTLs may suddenly return a large number of inconsistent records under the "Slashdot effect," in which traffic unexpectedly surges due to a URL appearing in a popular news site, such as Slashdot. This highlights yet another shortcoming of manually set TTLs—they generally reflect the *estimated* popularity of a domain rather than the *real-time* popularity.

With the ever-increasing usage of mobile devices in today's Internet and future Internet proposals such as SCION [6], the inconsistency problems arising from only using TTLs have the potential to get much worse, since update frequencies will increase with the use of short-lived, rapidly-changing paths.

Monitoring this popularity not only allows DNS to reduce the aggregate inconsistency for all types of records, but also allows for a system that can dynamically allocate resources to handle various server loads, traffic patterns, and update frequencies. Therefore, our inconsistency metric is based on a DNS record's popularity as well as its update frequency.

Definition 2: Let $Q_r(T)$ be the set of all queries for a record r received by a DNS caching server in some time interval T. Then the **Expected Aggregate Inconsistency** (EAI) of the caching server is

$$EAI_r(T) = \mathrm{E}\left[\sum_{q \in Q_r(T)} I_r(q)\right] \tag{2}$$

where $I_r(q)$ is as defined in Equation 1.

The intuition behind EAI is that the inconsistency over all the queries reflects the popularity of a DNS record. A more popular DNS record will have more queries and thus affect more users if it is inconsistent, resulting in a higher EAI.

If we assume that T begins at time t when record r is cached, then using Equation 1, we can write Equation 2 as

$$EAI_r(T) = \mathrm{E}\left[\sum_{q \in Q_r(T)} u_r(t, t_q)\right] \tag{3}$$

which shows more concretely that the EAI of a record is simply the expected total number of missed updates over all queries q.

B. Abstracting the DNS Caching Server Topology

So far we have defined EAI for a single DNS caching server. Generally, caching records from other caches or secondary nameservers is discouraged due to increased dependencies and delays in propagating updates [9]. However, much evidence demonstrates that "chained-resolution" widely exist in today's Internet [10]–[12]. Accordingly, we consider these "chains" of caching in order to provide a general model more flexible to changes in DNS caching, which may increase with the advent of future Internet architectures.

We thus consider a *hierarchy* of DNS caches, which we call a *logical cache tree*. This represents a hierarchy of caches, where the root node caches records directly from the authoritative server and the other nodes cache records from their respective parents. This structure is illustrated in Figure 1.

In a logical cache tree, a caching server is defined to be a child of another if it fetches and caches DNS records from that server. For example, the logical cache tree for DNS record "google.com" is made up of Google's authoritative server "ns1.google.com" and all the caching servers querying "ns1.google.com" for the DNS record including proxy caching servers directly answering clients' queries, and DNS forwarders answering queries from other proxy caching servers. Thus the DNS forwarders are parents of proxy caching servers which query domain names from them, and the authoritative servers are parents of both DNS forwarders and other caching servers which directly query those authoritative servers for the domain name. For DNS records served by multiple authoritative servers for load balancing, we treat all of the authoritative servers as a single root node, though these servers may be distributed across the network. In this way, there will be only one logical cache tree for a single DNS record.

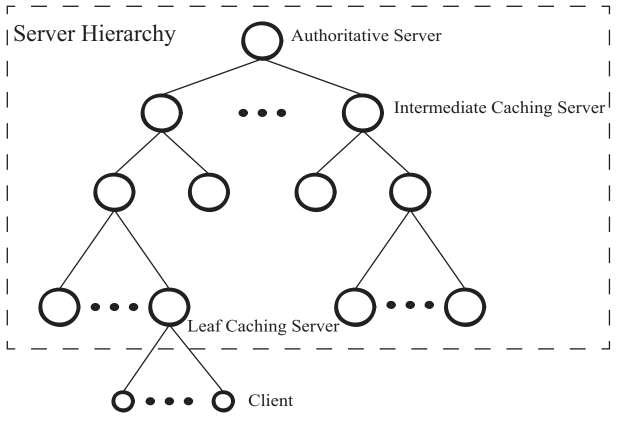

Figure 1. Structure of a logical cache tree.

C. Assumptions

To simplify the analysis of our model, we assume that the queries and updates for a single DNS record received by a caching server can be modeled by a Poisson process. Thus queries occur independently of each other and do not arrive simultaneously at any time. Furthermore, the interval between two successive queries follows a negative exponential distribution. Previous work by Chen et al. [7] has confirmed this pattern, but our model can be analyzed with any underlying distribution.

We further assume that for a given record, the queries received by any two caching servers are independent. Therefore, the arrival patterns of queries at different caching servers can be treated as two entirely independent stochastic processes. While of course certain events such as the broadcast of a highly-anticipated sporting event may cause queries to increase to many caching servers in a similar manner, the increase is caused by an external event rather than by another caching server. In reality there is some correlation since a child cache may use its own query volume to determine the rate at which it queries its parent cache, but we expect that queries from child caches will constitute a negligible fraction of a parent cache's total queries.

Finally, we assume that a caching server prefetches a cached record when its TTL expires. One advantage of this "eager" caching behavior is that after a popular record expires, the extra latency of contacting a parent cache or authoritative server on the following query can be avoided [7].

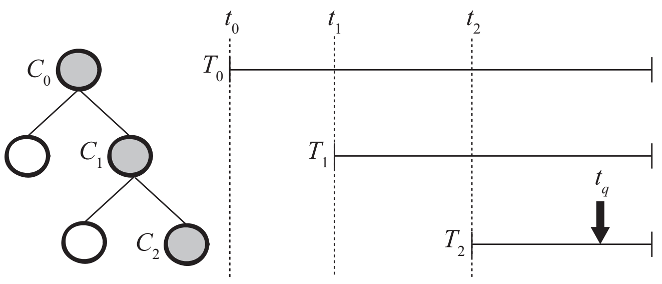

Figure 2. Cascaded inconsistency from caching server C_0 to C_1 and C_2. The record that C_2 caches at time t_2 is the same record cached by C_0 at t_0, and thus $I_r(q)$ should be measured from t_0.

The disadvantage, however, is that unpopular records will be prefetched without benefiting any client. As a consequence, it would be desirable to consider the capacity of the storage device and select only the most popular records to cache. We will discuss DNS record selection in Section III-C in more detail.

D. Modeling Inconsistency in the Logical Cache Tree

Our assumptions provide several properties that aid the analysis we present in this section. Modeling query arrivals as a Poisson process allows us to concretely estimate the aggregate inconsistency of a given caching server. The independence of caching servers eliminates dependencies which may clutter our analysis. Finally, prefetching allows us to focus on the critical problem of calculating the EAI rather than what happens in the short period between a record's expiration and its subsequent re-caching.

However, up to this point we have calculated inconsistency as if a caching server were directly querying the authoritative nameserver for a record. Calculating inconsistency in a logical cache tree is more complicated, as caching servers pass on their possibly inconsistent records in response to their children's queries.

The resulting effect is a "cascaded" inconsistency. As shown in Figure 2, C_2 will respond to query q with a copy of a record that was originally cached at time t_0. Therefore, calculating the inconsistency of the response to q based on $u_r(t_2,t_q)$ would underestimate the true value. Rather, the true inconsistency should be calculated based on $u_r(t_0,t_q)$.

Notice also from Figure 2 that

$$u_r(t_0,t_q) = u_r(t_0,t_1) + u_r(t_1,t_2) + u_r(t_2,t_q) \tag{4}$$

which follows from the fact that t_1 and t_2 are the times that C_1 and C_2 query their parent server. We can then define the cascaded inconsistency as follows:

Definition 3: Let $A(C_n)$ be the set of ancestors of a caching server C_n excluding the root of its logical cache tree, and $p(i)$ the index of the parent caching server of caching server C_i. Each C_i has cached their copy of the record r at time t_i. Then, the **cascaded inconsistency** of a response to a query q at C_n is defined as

$$I_r(q,C_n) = u_r(t_n,t_q) + \sum_{i \in A(C_n)} u_r(t_{p(i)},t_i) \tag{5}$$

where $I_r(q,C_n)$ is the inconsistency of query q at server C_n.

Following Definition 2, we can define the EAI for caching server C_n in r's logical cache tree during any time period T_n starting at time t_n

$$EAI_r(T_n,C_n) = E\left[\sum_{q \in Q_r(T_n)} I_r(q,C_n)\right] \tag{6}$$

Let t'_n be the time when record r expired on caching server C_n. From now on, we will only consider $T_n = [t_n,t'_n]$. Depending on the way each caching server sets the TTL, we derive two closed form expressions of EAI.

Case 1 (adopted by current DNS): In this case, when responding to a query for record r from a child caching server, the parent caching server appends the outstanding TTL to the replied message, which is computed by subtracting the TTL for r from the parent caching server by the time that has elapsed since r was first cached. Upon receiving the response, the child caching server retrieves the "outstanding TTL" from the response and sets it as the TTL for record r. Following this way, the expiration time t'_is for r on all the descendant caching servers C_i of a caching server C_n will be exactly the same as that on C_n. Interestingly enough, with our assumption that a caching server prefetches a cached record when its TTL expires, the time t_is when r will be cached again on all C_is will also be exactly the same as t_n. As a result, the lifetime T_i of a cached record on each caching server C_i belonging to the same sub-tree of the logical cache tree will be "synchronized".

Applying our assumption of Poisson processes for both record updates and queries, we derive the EAI for a given caching server C_n over time period T_n as

$$EAI_r(T_n,C_n) = \frac{1}{2}\lambda_n \mu \Delta T_n^2 \tag{7}$$

where μ is the update frequency parameter of the Poisson process for record update, λ_n is the query rate parameter of the Poisson process for queries to caching server C_n and $\Delta T_i = t'_i - t_i$ for a given $T_i = [t_i,t'_i]$ is actually the TTL of record r on caching server C_i.

Case 2: In this case, upon receiving a response to a DNS query, each caching server independently decides the TTL to set. Following this method, the lifetime T_is of different caching servers within a sub-tree no longer correlate with each other. Similarly, with the assumptions of Poisson processes for both record updates and queries, we derive the EAI for a given caching server C_n over time period T_n in r's logical cache tree as

$$EAI_r(T_n, C_n) = \frac{1}{2}\lambda_n \mu \Delta T_n \left(\sum_{C_i \in A(C_n)} \Delta T_i \right) \tag{8}$$

E. TTL Optimization

From Equations 7 and 8, we can see that changing the ΔT values (which we henceforth use to represent the TTL of a record at a given caching server) controls the EAI of a record in a logical cache tree. By decreasing the TTL, we can reduce the aggregate inconsistency in a logical cache tree. However, this in turn shifts a greater burden to the bandwidth overhead. Therefore, in order to address the trade-off between inconsistency and bandwidth cost, we formulate a multi-objective optimization problem to calculate a TTL that will strike the optimal balance between inconsistency and bandwidth overhead.

In addition, instead of optimizing the TTL for each server individually, we consider the global inconsistency and bandwidth because it is desirable for a caching server to slightly increase its bandwidth overhead to greatly reduce the inconsistency experienced by its descendant caching servers.

For a DNS record r, we let M be the set of all caching servers in the logical cache tree, ΔT_i the TTL for r on C_i, $EAI_r(T_i, C_i)$ the EAI during the time period T_i, and b_i the bandwidth cost in bytes to cache a record (the size of the record times the number of hops from the parent). For a specific caching server C_i, the EAI per unit time is $EAI_r(T_i, C_i)/\Delta T_i$, and the amortized bandwidth cost per unit time is $b_i/\Delta T_i$. Recall that our goal is to find an appropriate TTL value ΔT_i to achieve a specified balance point between inconsistency and bandwidth overhead. As a result, we can seek to utilize multi-objective optimization by setting the target cost function U as follows:

$$U = \sum_{C_i \in M} \frac{EAI_r(T_i, C_i)}{\Delta T_i} + c \cdot \frac{b_i}{\Delta T_i} \tag{9}$$

where c is a factor describing the "exchange rate," that is, how much bandwidth can be sacrificed for better consistency. We address how to set the exchange rate c in Section V. Our goal then is to compute the TTL values ΔT_i which minimize the target cost function.

For Case 1 in Section II-A, we substitute the expression for EAI from Equation 7 and solve for the optimal TTL value ΔT_i^*s by finding the minimum of the cost function U:

$$\Delta T_i^* = \sqrt{\frac{2c \cdot (\sum_{C_j \in S(C_i)} b_j)}{\mu (\sum_{C_j \in S(C_i)} \lambda_j)}} \tag{10}$$

where λ_j is the rate parameter of queries to C_j and $S(C_i)$ is the set of all the caching servers within the sub-tree containing C_i and rooted at the highest caching server.

Similarly, for Case 2 in Section II-A, we substitute the expression for EAI from Equation 8 and solve for the minimum optimal TTL value ΔT_i^*s:

$$\Delta T_i^* = \sqrt{\frac{2c \cdot b_j}{\mu \left(\sum_{C_j \in D(C_i)} \lambda_j + \lambda_i \right)}} \tag{11}$$

where $D(C_i)$ is the set of all descendant caching servers of server C_i.

The two forms of optimal TTLs in Equations 10 and 11 have different numbers of required parameters. For Case 1, to compute the optimal TTL for any caching server, λ_i and b_i from each caching server C_i in $S(C_i)$ are required. However, for Case 2, in order to calculate the optimal TTL, only the λ_i from each descendant caching server C_i is required. For a logical DNS cache tree where each node has many children and the tree size is on the order of the number of DNS cache servers in the whole world, the number of estimated parameters required in Case 2 for each cache server is significantly smaller than that required in Case 1. In the following sections, we will focus on the model for Case 2 to reduce the number of required parameters, which in turn equals the number of caching servers uploading the parameters, and therefore improve the usability of the model.

With the form of ΔT_i^* in Equation 11, each caching server can collect the parameters c, b_i, μ and $(\sum_{C_j \in D(i)} \lambda_j) + \lambda_i$, which is the sum of λs from all its descendant cache servers, and compute the optimal TTL to minimize the global cost function U. The minimum of the cost function U is:

$$U^* = \sum_{C_i \in M} \sqrt{2c\mu b_i \left(\left(\sum_{C_j \in D(C_i)} \lambda_j \right) + \lambda_i \right)} \tag{12}$$

III. System Design

We now address the following real-world aspects of ECO-DNS: a) parameter monitoring and aggregation, b) how to set the TTL value, c) DNS record selection, and d) prefetching DNS records. We will then discuss deployment challenges for ECO-DNS.

A. Parameter monitoring and aggregation

To follow our model in Section II, ECO-DNS requires caching servers to estimate the update frequency μ and the frequency of queries λ. While all caching servers share the same update frequency of a given record and could possibly retrieve it as a field of the DNS record, each caching server has to independently estimate its own local query frequency. In addition, for multi-level logical cache trees, λs originated from various nodes must be aggregated and made available to other nodes. More specifically, each node, in order to calculate its optimal TTL value, must have access to the λs of all of its descendant nodes.

In ECO-DNS, for each DNS record, the nodes in the logical cache tree are divided into three categories: the authoritative server serving the updated record, intermediate caching servers caching the DNS entry, serving queries originating from descendant caching server nodes, and leaf cache servers directly answering the DNS queries from clients. As shown in Table I, nodes belonging to different categories will take different responsibilities. First, the root node estimates the update frequency μ and incorporates it into the DNS record. Second, the intermediate caching server nodes estimate the λ of the queries, aggregate parameter λs received from descendants, and propagates the aggregated value upwards. Finally, the leaf caching servers estimate the local λ's and append the local λ in each query sent to the parent server.

Node	Estimated Param.	Aggregated Param.
Authoritative Server	μ	
Intermediate Server	λ	λ
Leaf Server	local λ	append λ in queries

Table I
ROLES AND TASKS OF DIFFERENT NODES IN THE LOGICAL CACHE TREE.

To estimate the update frequency parameter μ, the root node preserves a history of record updates and estimate the parameter accordingly. To estimate the query frequency parameter λ, each node utilizes a sliding window method to estimate the query frequency periodically.

To aggregate the Poisson Process rate parameters λ, we present two algorithms exploring the trade-off between accuracy and the amount of state kept on parent caching servers. Each caching server can arbitrarily select either of these two methods based on the trade-off between the amount of state it can maintain and the required accuracy.

In the first design, when a record stored in a cache server expires, the caching server appends the current aggregated λ value to the query. The parent caching server keeps an updated λ value for each of its children. This design requires per-child state, and is sensitive to topology changes in the logical cache tree, but provides accurate parameter aggregation.

In the second design, when a record stored in a caching server expires, the caching server appends the product of the parameter λ and the current TTL value ΔT. On the parent caching server side, instead of keeping state for each of its children, the parent caching server will aggregate the product $\lambda_i \Delta T_i$ in a sampling session with time duration $[t, t']$. After the sampling session, the parent server estimates the aggregated value as:

$$\frac{\sum_{i \in \mathscr{Q}} \lambda_i \Delta T_i}{t' - t}$$

where $\mathscr{Q}$ is the set of sampled DNS queries. This design does not require per-child state and is robust to topology churn in the logical cache tree. However, it is possible for the sampling method to fail in covering parameters from all child servers, which renders the method less accurate.

B. Setting TTL

The final TTL ΔT for a cached DNS record depends on two TTL values: the pre-defined TTL value ΔT^d specified in the DNS record, and the locally-calculated optimal TTL value ΔT^* based on observed parameters. For the latter TTL, as long as a caching server is provisioned with λ parameters aggregated from all its descendant caching servers and with the μ parameter it can use Equation 11 to compute ΔT^*. The resulting TTL value ΔT is

$$\Delta T = \min(\Delta T^*, \Delta T). \tag{13}$$

The decision of setting the TTL as the minimum between an automatically calculated value and a pre-determined value precludes two extreme cases. First, some records may be so "unpopular" that their optimal TTL is set to be exceedingly long. This method allows the owner of the DNS record to set an upper bounds over the TTL value. Second, during DNS cache poisoning attacks [13], the pre-determined TTL value of the fake DNS record could possibly be set to a huge number. In this case, the final TTL would be completely determined by a local calculated TTL. As a consequence, hijacking a popular DNS record becomes more challenging, as the fake DNS record will soon be dissipated with the timeout.

Each time a DNS record is first cached or refreshed, the caching server sets the TTL value based on Equation 13. During the lifetime of the cached record, this TTL value is fixed even though the underlying parameters may change. Compared to resetting the TTL value upon detecting parameter changes, this methodology reduces the computation cost of re-calculating optimal TTL values and avoids fluctuation of TTL within short time. Following this method, the more popular a DNS record is, the smaller the TTL is set, and the more "up-to-date" its corresponding TTL will be.

C. DNS record selection

ECO-DNS allows individual caching servers to select DNS records whose TTLs are managed and optimized. Generally, it is desirable to evict less popular records to accommodate more popular ones. Furthermore, caching and managing unpopular records is not economical, because resources are spent without benefiting any clients.

ECO-DNS relies on a traditional cache replacement algorithm to decide which DNS records to manage. Particularly, ECO-DNS utilizes the Adaptive Replacement Cache (ARC) algorithm [14] to account for heavy-tail DNS access patterns [15]. ARC is a self-tuning, low-overhead cache replacement algorithm which provides higher hit ratio under one-time accesses and loop accesses given the same amount of resources. The algorithm divides cached objects into two subsets: T-set, in which the whole object is cached, and B-set, in which only the metadata of the object is cached. For the DNS records in the T-set, ECO-DNS maintains and updates the parameters, and computes the optimal TTL for them. For the DNS records in the B-set, ECO-DNS will only keep the last estimated λ's for them as the initial values in case these records are added back to the T-set. The administrator is simply responsible for setting the number of DNS records for ECO-DNS to manage, and ECO-DNS can then self-tune based on local queries.

D. *DNS record prefetching*

ECO-DNS prefetches new records upon expiration of cached records. In the traditional way to handle record expiration, the new DNS record will not be fetched until the next query for that record arrives. Consequently, the next query usually suffers from latency which is an order of magnitude larger than for the query for a cached record [15]. However, without knowledge of the popularity of a DNS record, prefetching a record upon expiration could create a high overhead: unpopular records would be prefetched without benefiting any queries simultaneously.

With knowledge about records' popularity as a cornerstone, ECO-DNS breaks this dilemma by only prefetching the records that are expected to be queried soon, that is, records with relatively large λ values. Therefore, the latency caused by expired cached records is eliminated for popular records.

E. *Ease of deployment*

ECO-DNS is a lightweight system that is easy to deploy. First, for network traffic, ECO-DNS adds only one extra field in each DNS query and answer message, without requiring new message exchanges or protocol changes. Second, for extra state, ECO-DNS requires only $O(1)$ state for each DNS record. Third, for the software model, ECO-DNS requires no asynchronous events to be processed. As a result, it can be easily implemented and deployed as a module of current DNS software [16].

As for incremental deployment, ECO-DNS can be deployed alongside current legacy servers. For a single-level cache hierarchy, where each caching server directly contacts the authoritative server, the administrator can individually deploy ECO-DNS on each caching server to improve the caching server performance. For a multi-level cache hierarchy, ECO-DNS can be deployed from lower-level caching servers to higher-level ones. As long as the caching servers within a sub-tree implement ECO-DNS, ECO-DNS will function perfectly independently from caching servers in other sub-trees.

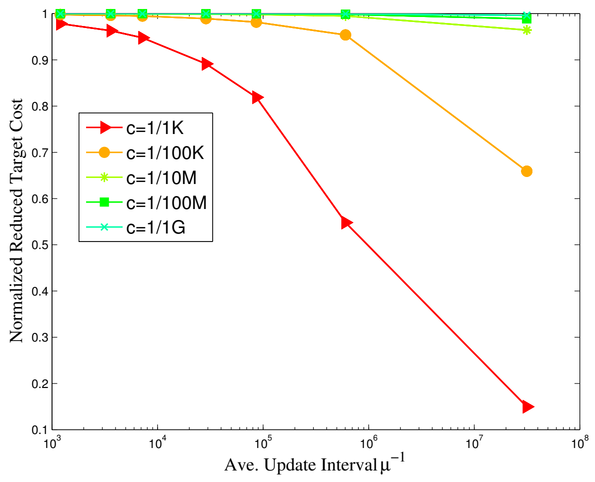

Figure 3. Normalized reduced target value for the single-level caching hierarchy.

IV. Evaluation

We now evaluate ECO-DNS by carrying out simulations based on real DNS traces and AS topology data. In our evaluation, we seek to answer three main questions:

- For a given cost in bandwidth, does ECO-DNS achieve better consistency than manually setting TTL in today's Internet?
- How do the inconsistency and bandwidth overhead of a multi-level caching scheme with ECO-DNS compare to those of a single-level DNS caching scheme?
- How well does the optimization mechanism of ECO-DNS adapt to real-time changes in the rate parameters of DNS queries?

A. *Dataset*

KDDI, the second largest ISP in Japan, provided us with DNS trace data containing 10 minutes of traffic to their DNS caching server every four hours on Feb. 28th, 2013 and Mar. 3rd, 2013. These data include DNS query arrival times, response packet sizes and response record types. The data are divided by different domains, which are in turn categorized based on their popularity: the top 100 most popular domains, and the domains which in each trace are queried at most 100K, 10K, 1K and 100 times, respectively. These data enable us to test our model under a range of domain popularities and network conditions.

B. *Single-Level Caching*

To explore the consistency benefits of ECO-DNS over manually set TTL used by current DNS caches with respect to various combination of parameters, we design a simulation for a simple topology, containing a single caching

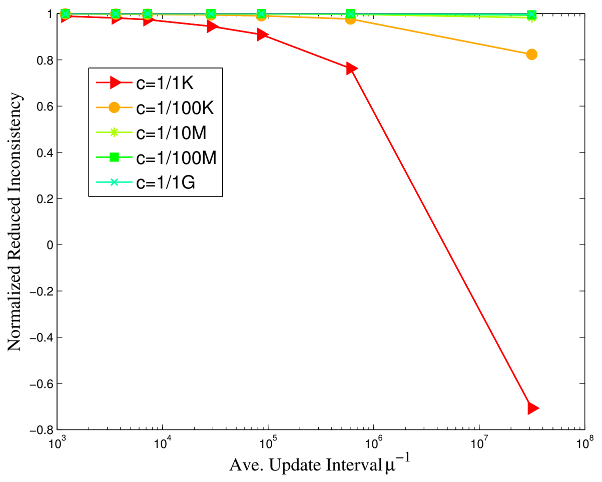

Figure 4. Normalized reduced inconsistency for the single-level caching hierarchy.

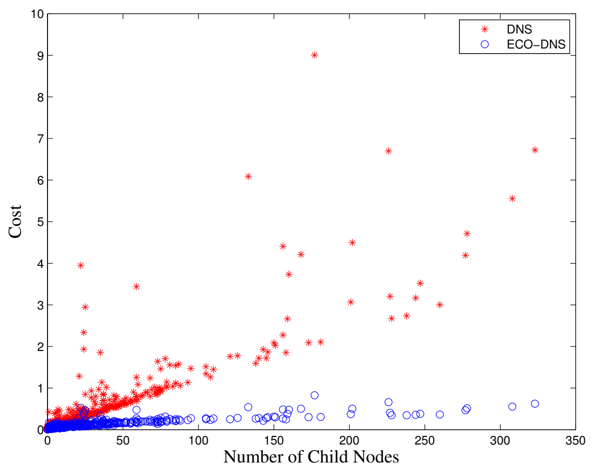

Figure 5. Cost for each node in the CAIDA cache trees versus the number of children of the node.

server and an authoritative server. We set the number of hops between the caching server and the authoritative server as 8 hops. We simulate a period of time over 1000 DNS record updates in the authoritative server. Because this time period (weeks or even years) is longer than the time period of our DNS trace from KDDI, we repeat the DNS trace itself to create one of sufficient length. In the simulation, we simulate both ECO-DNS and manually set TTL-based method, with the manually set TTL as 300s, which is common for popular domains. The values of the target function and the number of inconsistent DNS answers are recorded and compared when varying the average update intervals and the weight c that quantifies the tradeoff between clients' missed updates and bandwidth cost.

Figure 3 shows the reduced cost normalized by the total cost using manually-set TTL, with respect to different values of update interval and weight c. The update intervals vary from 2 hours to 1 year and the weight c would change from 1KB per inconsistent answer (a high consistency requirement) to 1GB per inconsistent answer (a low consistency requirement). In general, ECO-DNS could reduce the cost by 90% when the average update interval is within a week. When the average update interval increases as the update frequency drops, the reduced target function value will fall to 10%, as the manually set TTL becomes closer to the optimal TTL.

Figure 4 shows the reduced inconsistency normalized by the total inconsistency using manually-set TTL, with respect to different values of update intervals and weight c. The overall curves are similar to the curves for reduced target function values. Nevertheless, Figure 4 demonstrates the effect of changing the weight c. When c is as small as 1KB per inconsistent answer, ECO-DNS automatically adjusts the TTL value to decrease the query updating rates to alleviate the bandwidth burden caused by DNS records updating. As the weight c grows larger, indicating a preference for consistency over bandwidth, ECO-DNS re-adjusts the TTL value to update more frequently to reduce inconsistency.

C. Multi-Level Caching

To measure the consistency and overhead benefits that the deployment of ECO-DNS would have in a logical cache tree, we simulated our caching model and a model of the current DNS caching mechanism on various network topologies. We collected real-world topologies from the Inferred AS Relationships data set from CAIDA [17], and generated similar topologies using Tomasik and Weisser's aSHIIP [18], a hierarchical random topology generator. From these topologies we constructed logical cache trees of various sizes and depths on which to test our models.

We obtained a total of 270 logical cache trees from the CAIDA dataset and generated 469 logical cache trees using aSHIIP. We used a general linear preference (GLP) model to generate our random topologies, with parameters $m_0 = 10$ (number of starting nodes), $m = 1$ (number of edges added), $p = 0.548$ (probability of adding an edge versus a node) and $\beta = 0.80$ (preference to connect a new node to more popular nodes) [19]. These parameters yield topologies with core sizes and peering link ratios similar to those in the CAIDA dataset [20]. The edges in the GLP topology were then classified as provider-to-customer or peer-to-peer based on aSHIIP's inference algorithm.

From these topologies we were able to form logical cache trees by assigning each customer node an unique provider. If the node had multiple providers in the original topology, we randomly chose a provider, weighting the probabilities of each provider by relative total degree. We constructed a total of 558 logical cache trees ranging in size from 2 to 11057 nodes and spanning up to six levels. (We did not consider single-node trees, as that would represent an authoritative server with no caching servers.)

For a given cache tree, we conducted 1000 runs of our simulation. In each run we randomly chose the λ parameters for each leaf node and the size of the response for the DNS records, modeling the distribution of these values after those in the KDDI data.

Note that in today's DNS, pre-defined TTLs are often poorly set, so rather than using TTLs similar to those in the wild for our model of the current DNS, we chose to compare the performance of ECO-DNS to that of DNS *assuming that the TTL is optimally chosen*. Therefore, our simulations demonstrate a *lower bound* for the performance benefits of ECO-DNS over the current system.

In order to determine the optimal value of the target function in today's DNS, we simply set the TTL for each node to be the same and find the TTL that minimizes the cost function U. This TTL is

$$\Delta T^* = \sqrt{\frac{2c\sum_{i\in M} b_i}{\mu \sum_{i\in M}\left(\lambda_i + \sum_{C_j\in D(C_i)}\lambda_j\right)}} \tag{14}$$

To account for the fact that ASes near the root of our topologies are often larger than at the bottom, we define b_i as the size of the response times 4 hops for ASes at depth 1, 7 hops for those at depth 2, 9 for those at depth 3, and an additional hop over that of the previous depth for all other nodes. However, since in ECO-DNS the nodes pull the record from their parents rather than the authoritative server, in this model we define b_i to be the size of the response times 4 hops for ASes at depth 1, 3 hops for those at depth 2, 2 hops for those at depth 3, and 1 hop for those at greater depth.

In addition to calculating the optimal value of the target function, we also calculated the sum of the inconsistency and overhead for each node. Figs. 5 and 6 show the value of the cost function U under today's DNS and ECO-DNS required by each node versus the number of children the node has. Note that parents with more children bear a greater cost because they must update more frequently to minimize the inconsistency of the records their children receive.

Figs. 7 and 8 present the average cost for a node in each level of a CAIDA and aSHIIP topology, respectively. The high variability in the first level is due to the fact that both small and large cache trees have nodes in level 1, resulting in nodes with a wide range of connectivity. In deeper levels, there are fewer cache trees with nodes at that depth and therefore a smaller range in the number of a children a node may have.

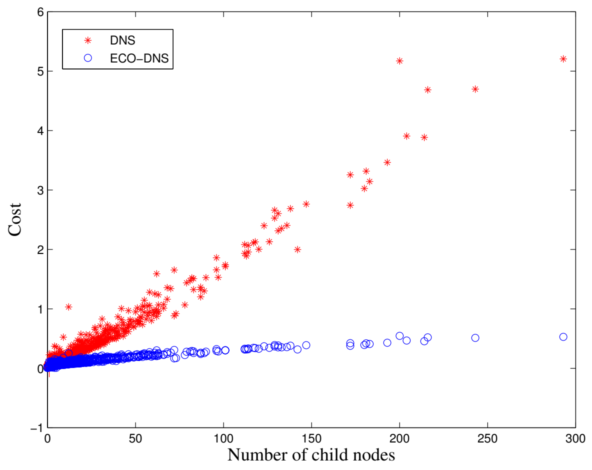

Figure 6. Cost for each node in the aSHIIP-generated cache trees versus the number of children of the node.

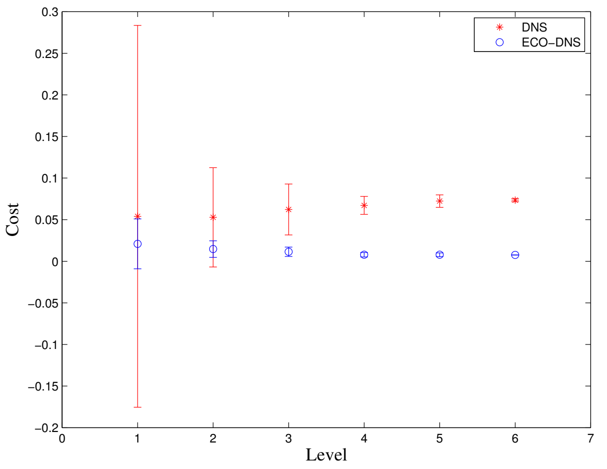

Figure 7. Average cost for a node in each level of a CAIDA cache tree. Error bars represent the standard error of the mean.

D. *Convergence upon parameter changes*

To evaluate how ECO-DNS reacts to the changes of λ parameter for DNS queries and investigate the performance impact of the deviation between estimated λ and real λ, we design a simulation on the same topology as that in Section IV-B, i.e., a single caching server and an authoritative server. The simulation is based on one of the DNS query trace data from KDDI, which records the queries for a domain name received by a DNS caching server during the time period of a day. The trace data is divided into six pieces, each of which is a 10-min trace and is sampled every 4 hours. The λs computed from each piece of the trace data are [301.85, 462.62, 982.68, 1041.42, 993.39, 1067.34]. Based on the extracted λs, we simulate the Poisson Process

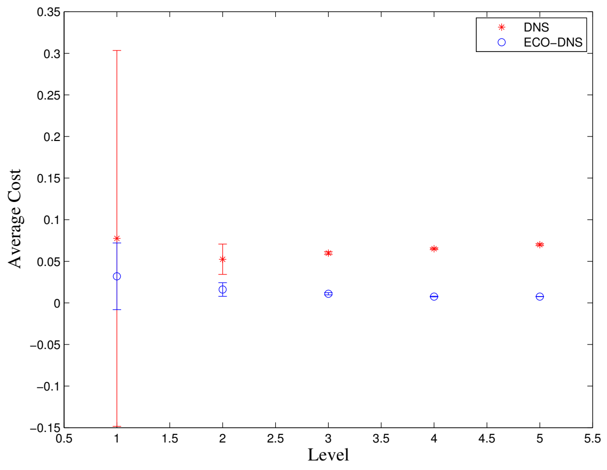

Figure 8. Average cost for a node in each level of a aSHIIP cache tree. Error bars represent the standard error of the mean.

for DNS queries of 24 hours, during which each λ stays the same for 4 hours. We intentionally set the initial λ as the mean value of the λs from the trace data to simulate the influence caused by the initial λ value. We compare two designs for parameter estimation: a) counting the number of queries within a fixed-length time window, and b) calculating the duration given a fixed number of queries. We simulate a) with time interval 100s and 1s, while we simulate b) with number of queries 5000 and 50.

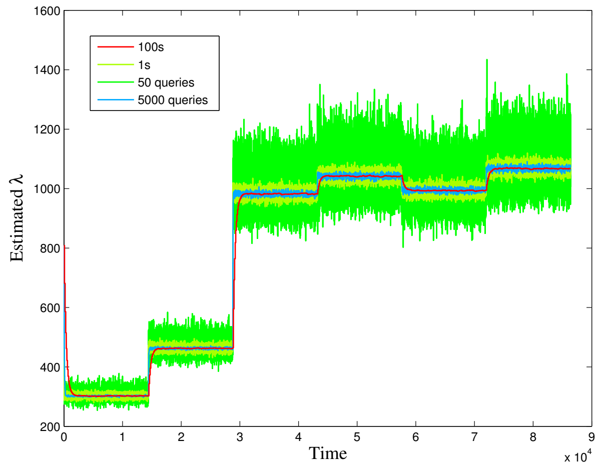

Figure 9. Dynamics of the estimated λ on parameter changes.

Figure 9 shows the dynamics of the estimated λ upon parameter changes. According to the figure, there are two conflicting factors that affect the performance of parameter estimation. The first one is the convergence speed, which depicts how fast the parameter estimation method can update the parameter to the new value. The second one is stability, which indicates how large the amplitude of the vibration of the estimated parameter is when the parameter stays unchanged. In Figure 9, while method b) with 50 queries converges within seconds, we observe a vibration amplitude larger than 10% of the real λ. Instead, a) with 100s will take 10 minutes before the estimated parameter converges, but the amplitude of the vibration is within 0.1% of the true value. The dynamics of a) with 1s and b) with 5000 queries sit in the middle, possessing faster convergence speed than a) with 100 and smaller deviation than b) with 50 queries.

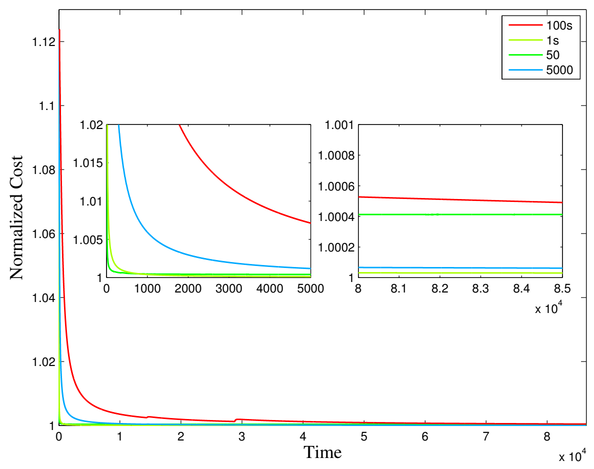

Figure 10. Extra cost incurred upon parameter changes.

Figure 10 demonstrates the extra cost caused by the deviation of the estimated λ from the real λ under different parameter estimation methods. Here, normalized cost is calculated by dividing cumulative cost with estimated λ by cumulative cost with real λ. We could observe from the figure that the two factors discussed above influence the extra cost in different ways. While the low convergence speed will incur a one-time extra cost, the lack of stability will incur cumulative extra cost growing as time goes on. For instance, in the left zoom-in sub-figure, method a) with 100s and method b) with 5000 incur a relatively large extra cost because of the slow convergence from initial λ to real λ. On the other hand, in the right zoom-in sub-figure, method b) with queries number 50 incur an extra cost linear in time, which is reflected by the constant normalized extra cost. Therefore, if the frequency of the parameter changes is very high, we should choose the parameter estimation method with fast convergence speed. Otherwise, in most cases, we should choose the parameter estimation method with better stability. In general, after 10 minutes from starting ECO-DNS, the extra cost incurred by parameter estimation is within 0.1% of the total cost, which is usually negligible compared to the improvement of ECO-DNS over current DNS caching.

V. Discussion

In Equation 9, we utilize the weight c to quantify the tradeoff between the number of the clients' missed updates and the bandwidth cost of querying a DNS record. The administrator of the local cache server could tune this c parameter to balance the consistency, as part of quality of its caching service, and bandwidth cost. In addition, the administrator could intentionally set the weight c as the globally agreed value, which could reflect the global tradeoff between inconsistency and bandwidth cost. Interestingly, the establishment of the DNS cache hierarchy itself actually provides hints about the range of the weight c that is well accepted. While one-level cache is constructed to reduce the bandwidth cost through sacrificing consistency, the fact that no two-level cache is widely deployed reflects the reluctance against further sacrificing the consistency for reducing the bandwidth cost. Therefore, the global weight c could possibly be set as a value corresponding to the underlying trade-off.

We also observe in Equation 9 the parameter b which characterizes the bandwidth cost of updating the local DNS record. One form of b could be calculated as the product of the size of a DNS record and the number of hops. This form well characterizes the number of bits transmitted in the whole network to update the local record. Another form of b could be calculated as the latency of transmitting the DNS record. While the latency could only roughly indicate the real bandwidth cost, it could cover the server load and the network status. Finally, b could also directly reflect the real-world expense by considering the bandwidth cost between customer and provide ISPs. By setting the parameter b in different forms, the administrator controls over different forms of cost he/she would like to limit.

VI. Related Work

The assumption that the stochastic process of DNS queries' arrival time could be modeled as a Poisson Process has been adopted and verified by Cohen et al. [21] and Chen et al. [7]. Jung et al. [15] seek to extend the TCP arrival time model to model DNS traffic and propose to use a Pareto or Weibull distribution to model the distribution of the arrival time of DNS queries. Jung [22] further weakens the assumption to model the random process of DNS queries' arrival time by a renewal process, without assuming any distribution of the arrival intervals between DNS queries. In this work, we rely on no particular assumption about distribution of the arrival time to derive our inconsistency metric and move one step further to use the Poisson process assumption to optimize the performance of ECO-DNS.

Differing from our efforts to preserve the "pull-based" nature of the current DNS cache system and enhance its consistency by optimizing its behavior, several works have been proposed to revolutionize the way current DNS works to improve various performance metrics including bandwidth cost, latency, consistency and security against Denial of Service (DoS) attacks. Handley et al. [8] and Chen et al. [7] have proposed that the authoritative servers push DNS records to caching servers to improve robustness and consistency, respectively. The key observation under these two works is that the number of active DNS records is actually limited and manageable. However, in order to push the DNS records directly from an authoritative server to n subordinate caching servers, it still requires $O(n)$ state stored on both the authoritative server and the cache servers, which is difficult to manage across the whole networks and consume local storage.

As another direction, Ramasubramanian [23] proposes CoDoNS, which leverages peer-to-peer overlay networks to improve load balance and resilience against denial-of-service attacks. However, improved availability comes at the price of weakened consistency. In fact, the distributed nature of DNS records render it even more challenging to maintain consistent cached DNS records. As a consistency control mechanism, ECO-DNS could be combined with CoDoNS to further reduce the latency of DNS resolution and retain high consistency while preserving high availability.

Finally, Sharma et al. [24] propose Auspice, a new global name service with low-latency, high availability, and high consistency by replicating records across geo-distributed servers in a demand-aware manner. While Auspice targets bounded consistency for all mobile clients' records, ECO-DNS solves an tunable optimization problem between consistency and update cost, resulting high consistency for popular and dynamic records and low cost for unpopular and static records. Furthermore, instead of requiring new name service infrastructures, ECO-DNS bases itself on today's DNS architectures which greatly reduces its deployment difficulty.

Acknowledgment

We gratefully thank Akira Yamada, Takashi Matsunaka, and Ayumu Kubota from KDDI for providing us the DNS data, Vyas Sekar, Soo Bum Lee, Kevin Fall for insightful feedbacks of the initial version of the paper, and the anonymous reviewers for their valuable comments.

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 617605. We also gratefully acknowledge support by ETH Zurich, and NSF under award number CNS-1040801.

VII. Conclusion

Current static TTL-based cache invalidation policies are sub-optimal and difficult to configure in a flexible fashion. Furthermore, the lack of comprehensive consistency control prevents us from leveraging a multi-level caching hierarchy to further reduce network overhead. Without formal definition of a consistency metric, it is very difficult to explore the tradeoff between consistency and network cost, and to optimize the performance of cache servers.

In this paper, we first define a consistency metric, and formally model the tradeoff between inconsistency and bandwidth cost in the DNS cache hierarchy. Based on this model, we propose ECO-DNS, an effective consistency control system that automatically adjusts the TTL according to local popularity and the update frequency of the DNS record. Simulations show that ECO-DNS can reduce the inconsistency by 90% in the single-level cache hierarchy and also enables multi-level cache hierarchies. The Internet could greatly benefit from the improved consistency that ECO-DNS provides, even while reducing the total bandwidth overhead of DNS queries.

References

[1] D. cache poison attack incidents in Brazil, "http://threatpost.com/major-dns-cache-poisoning-attack-hits-brazilian-isps-110711/."

[2] "Understanding DNS forwarders," http://technet.microsoft.com/en-us/library/cc782142(v=ws.10).aspx.

[3] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache," in *Proceedings of USENIX ATC*, 1996.

[4] Akamai, "www.akamai.com."

[5] J. Pang, A. Akella, A. Shaikh, B. Krishnamurthy, and S. Seshan, "On the responsiveness of dns-based network control," in *Proceedings of ACM IMC*, 2004.

[6] X. Zhang, H.-C. Hsiao, G. Hasker, H. Chan, A. Perrig, and D. G. Andersen, "SCION: Scalability, control, and isolation on next-generation networks," in *Proceedings of IEEE Oakland*, 2011.

[7] X. Chen, H. Wang, and S. Ren, "DNScup: Strong cache consistency protocol for DNS," in *Proceedings of IEEE ICDCS*, 2006.

[8] M. Handley and A. Greenhalgh, "The case for pushing DNS," in *Proceedings of ACM HotNets*, 2005.

[9] D. Barr, "Common DNS operational and configuration errors," https://tools.ietf.org/html/rfc1912, 1996.

[10] A. Herzberg and H. Shulman, "Vulnerable delegation of dns resolution," in *Proceedings of ESORICS*, 2013.

[11] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: illuminating the edge network," in *Proceedings of ACM IMC*, 2010.

[12] K. Schomp, T. Callahan, M. Rabinovich, and M. Allman, "On measuring the client-side DNS infrastructure," in *Proceedings of ACM IMC*, 2013.

[13] J. Trostle, B. Van Besien, and A. Pujari, "Protecting against DNS cache poisoning attacks," in *Proceedings of IEEE NPSec*, 2010.

[14] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proceedings of USENIX FAST*, 2003.

[15] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "DNS performance and the effectiveness of caching," *IEEE/ACM Transaction on Networking*, 2002.

[16] Bind, "https://www.isc.org/software/bind."

[17] "CAIDA AS Relationships," http://www.caida.org/data/active/as-relationships/index.xml.

[18] J. Tomasik and M. A. Weisser, "aSHIIP: autonomous generator of random internet-like topologies with inter-domain hierarchy," in *Proceedings of IEEE MASCOTS*, 2010.

[19] T. Bu and D. Towsley, "On distinguishing between internet power law topology generators," in *Proceedings of IEEE INFOCOM*, 2002.

[20] J. Tomasik and M. A. Weisser, "The inter-domain hierarchy in measured and randomly generated AS-level topologies," in *Proceedings of IEEE ICC*, 2012.

[21] E. Cohen and H. Kaplan, "Proactive caching of DNS records: Addressing a performance bottleneck," in *Proceedings of IEEE/IPSJ SAINT*, 2001.

[22] J. Jung, A. W. Berger, and H. Balakrishnan, "Modeling TTL-based Internet Caches," in *Proceedings of IEEE INFOCOM*, 2003.

[23] V. Ramasubramanian and E. G. Sirer, "The design and implementation of a next generation name service for the Internet," in *Proceedings of ACM SIGCOMM*, 2004.

[24] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, and A. Yadav, "A global name service for a highly mobile internetwork," in *Proceedings of ACM SIGCOMM*, 2014.