A Global Name Service for a Highly Mobile Internet

Abhigyan Sharma Xiaozheng Tie Hardeep Uppal Arun Venkataramani
David Westbrook Aditya Yadav

UMASS Computer Science Technical Report: UM-CS-2013-023

Abstract

Mobile devices dominate the Internet today, however the Internet continues to operate in a manner similar to its early days with poor infrastructure support for mobility. Our position is that in order to address this problem, a key challenge that must be addressed is the design of a massively scalable global name resolution infrastructure that rapidly resolves identities to network locations under high mobility. Our primary contribution is the design, implementation, and evaluation of Auspice, a next-generation global name resolution service that addresses this challenge. The key insight underlying Auspice is a placement engine that replicates name records to provide low lookup latency, low update cost, and high availablity. We have implemented a prototype of Auspice and evaluated it on Planetlab, a local cluster, as well as through large-scale trace-driven experiments. Our experiments show that Auspice provides 1.0 sec to 24.7 sec lower update latencies than commercial managed DNS services and up to 9× lower lookup latencies than a proposed DHT-based replication alternative to DNS.

1 Introduction

"Mobile" has long arrived, but the Internet remains unmoved. Today, there are almost 6 billion cellphones, over a billion of which are smartphones [37]. The number of smartphones alone now exceeds the number of tethered hosts on the Internet, and recent studies [6] suggests that the total data traffic generated by mobiles now exceeds that of tethered clients. However, the current Internet continues to operate as it did when dominated by tethered hosts, simply ignoring frequent endpoint mobility.

Today, one can not easily initiate communication with a smartphone (even when it has a public IP address) as there is no global infrastructure support for locating it. Applications like Netflix, Dropbox, or smartphone notification systems have to develop application-level support to enable a seamless experience for their users even as they change addresses several times a day, or let connections break (e.g., Skype). The lack of intrinsic support for mobility means that we are paying an unknowable price in terms of stymied application innovation and growth by forcing developers to redundantly develop common-case functionality, and forcing communication initiation to be mostly unidirectional.

Many before us have criticized the Internet architecture's poor support not only for mobility and multihoming [41, 15, 31, 52]. A common criticism is the Internet's so-called conflation of identity and location, i.e., the use of an IP address both to represent the identity of an interface as well as its network location, which is problematic for mobility (same identity, changing locations) and multihoming (single identity, multiple locations). It is commonly accepted wisdom that a cleaner separation of identity and location is instrumental to fixing these problems. However, the Internet does separate identities (domain-names) from network locations (IP addresses) through DNS. Most high-level programming languages also provide syntactic sugar to "connect" to names remaining oblivious to IP addresses; and techniques from a long line of work on connection migration can be employed to seamlessly handle mid-connection mobility.

But a key missing element from this package today is a distributed resolution infrastructure that can scale to orders of magnitude higher update rates than envisioned when DNS was created. To appreciate our envisioned scale, consider 10 billion mobile identifiers moving across a 100 different networks per day (or an aggregate rate of roughly 1M/sec). DNS's heavy reliance on TTL-based caching, a key strength recognized by its creators, researchers, and operators alike, also poses a significant handicap by increasing update propagation delays. It is not uncommon for DNS update propagation to take a day or more, resulting in long outage times when online services have to be moved unexpectedly, prompting cries for help on operator forums [14].

Our position is that seamless support for mobility requires a global name service that rapidly translates identities to locations irrespective of how exactly identities and locations are individually represented. Our primary contribution is the design, implementation, and evaluation of Auspice, a distributed system that addresses this challenge. The key strength of Auspice is an placement engine for replicating name records to achieve low lookup latency, low update cost, and high availability. Auspice achieves low-latency by inferring pockets of high demand for a name so as to create replicas of resolvers for that name close to them. Auspice achieves low latency, low update cost, and high availability using a placement optimization algorithm that (1) controls the number of replicas based on the observed read and write rates, and (2) determines where to place replicas based both on the inferred pockets of demand and the aggregate load at node locations near those pockets.

We have implemented a prototype of Auspice as a geo-distributed key-value store to serve as a flexible name resolution infrastructure for the Internet as well as future Internet architecture proposals. We have deployed and evaluated it on several different platforms including Planetlab, a local cluster, and Amazon EC2. Our evaluation shows the following:

- Auspice provides up to 9$\times$ lower lookup latency than proposed DHT-based alternatives for DNS with comparable update cost, while offering significantly better latency vs. cost . tradeoffs.
- Auspice achieves lookup latencies comparable to leading managed DNS providers with only one-third as many replica locations, and provides a median update latency that is 1.0 s to 24.7 s lower.

2 Global name resolution: Why care?

Given the huge body of prior work specifically on mobility as well as broader efforts on Internet architecture, it is natural to ask: (1) is a global name resolution service the best way to handle high mobility in today's Internet? (2) is a global name resolution service critical or even relevant to handling mobility if we had the luxury of re-architecting Internet naming and routing from a clean slate? In this section, we first present subjective arguments justifying an affirmative position on both questions, and then present objective results showing DNS's shortcomings as a resolution service under high mobility.

2.1 Internet mobility background

Despite a staggering diversity of proposals re-architecting Internet naming and routing, we find that they explicitly or implicitly embed one of three broad approaches to handling mobility–*indirection-based routing, global name-to-address resolution*, or *name-based routing*–based on how they go from the name of an endpoint to the endpoint itself. Table 1 classifies a number of proposed architectural alternatives based on how they handle mobility.

Indirection-based routing schemes are simple as an endpoint remains oblivious to the mobility of other endpoints. No name-to-address[1] lookup is needed at connection initiation time as a human-readable name maps to a *home address* (an IP address in Mobile IP [55] or a flat identifier's consistent-hash location in i3 [60]) that rarely changes by design. Mid-connection mobility, even when both endpoints move concurrently, is seamless to endpoints. However, the data plane pays the price for this simplicity—every data packet must be routed via an indirection agent at the home address, potentially causing significant routing stretch, e.g., two participants at a conference may in each direction need to detour packets halfway across the world despite being in the same room. Furthermore, indirection-based schemes require widespread deployment of indirection agents across different domains, posing a barrier to immediate adoption.

Global name-to-address resolution schemes rely on a distributed service to resolve names to addresses as the first step in connection establishment. The current Internet's DNS as well as a number of designs addressing the Internet's so-called identity-location conflation problem also need such a resolution infrastructure, e.g., to translate a self-certifying host identifier in HIP [41], AIP[19], XIA[38], or MobilityFirst[12]) or an identifier in LISP [15] or HAIR [31] to either an IP address [41], a self-certifying network identifier [19, 38, 12], or a hierarchical locator [31] that encodes routing information. Global name-to-address resolution schemes also subsume DHT-based Internet architectures such as LNA [21] or others [62, 32] as well as resolution systems like CoDoNS [56] that present a DHT-based drop-in replacement for DNS.

Global name-to-address resolution schemes need explicit support at endpoints to handle mid-connection mobility. There is a general consensus [59, 20, 35] that end-to-end connection migration, i.e., bilaterally without relying on an external service, can efficiently migrate connections when endpoints move one at a time; however, an external resolution service is needed to support concurrent mobility. Although the latter is seen as a rare case in most connection migration work, it can be a common event for disconnection-tolerant, mobile application scenarios (e.g., when a user closes her laptop at home and opens it at a coffee shop to continue watching a movie, by which time the cloud-hosted virtual server may have been migrated for load balancing).

Name-based routing schemes in the ideal have a tantalizing intellectual lure—to seamlessly handle mobility by routing directly over names obviating a name-to-address resolution infrastructure—but are

[1]We use the terms *name* and *identifier* interchangeably; likewise for the terms *address* and *location*.

	Indirection-based routing	**Global name-to-address resolution**	**Name-based routing**
Initial overhead	None	Name-to-address resolution (DNS, Nimrod[25], LISP[15], HIP[41], LNA[21], CoDoNS[56], XIA[38], MobilityFirst[12])	None
Data packet routing	Address-based routing through fixed "home address" (Mobile IP[55], GSM[11], i3[60])	Address-based routing, with support for late- or re-binding of names to addresses (e.g., Serval [51], MobilityFirst[12], XIA[38])	Name-based routing directly over flat (ROFL [24]) or structured names (TRIAD[26], CCN[39])
Mid-connection-mobility handling	Seamless in one RTT	Bilateral ([59],[20]) or via name service (under concurrent mobility) in a few RTTs	Outage times $\approx$ routing convergence delays
Routing table size vs. data path stretch	O(#prefixes) with triangle routing stretch	O(#prefixes) or O(#domains) ([38], [12]) for shortest-path routing	Ω(#identifiers) for small stretch over shortest-path routing [44]

Table 1: Classification of many alternative naming and routing architectures (not necessarily designed with mobility in mind) based on how they (might) handle mobility.

marred by several fundamental and practical challenges. First, name-based routing approaches can support seamless mobility only if route update propagation delays across the Internet are on the order of milliseconds, a daunting challenge given that interdomain routing can take several minutes to converge today. Second, theoretical results on compact routing [44] suggest discouraging fundamental trade-offs between the size of forwarding tables at routers and path stretch even without any mobility or multihoming, e.g., routing over N flat identifiers entails a prohibitive $\Omega(N)$ forwarding table size per router in order to ensure a small constant stretch factor ($\approx$3) compared to shortest-path routing. Simulation-based studies of flat-label routing strategies (e.g., ROFL [24]) reaffirm pessimistic conclusions about its scalability. Although it may appear that the scalability limitations of name-based routing can be alleviated by adding a hierarchical structure to names [26, 43, 39] (e.g., CCN/NDN [39] names such as `/umass/john_smith_phone/voip_call3/frame7`), frequent mobility still poses a challenge as routers would have to maintain special forwarding entries for "displaced names" when the name moves from its hierarchically organized namespace for longest-prefix matching to work correctly, i.e., high mobility effectively makes routing directly over structured names as hard as routing over flat names unless indirection or a name resolution infrastructure is used.

In summary, our position is that a global name-to-address resolution service is critical to handling high mobility in any network architecture as it offers the best combination of trade-offs: (1) a constant update overhead per mobility event to the name service, (2) a modest connection establishment overhead and rapid mid-connection mobility, (3) no data path inflation beyond underlying policy routing, and (4) small forwarding table sizes in conjunction with aggregatable addresses (IP prefixes like today or self-certifying network addresses [12, 38]). Perhaps the most compelling argument for the global name-to-address resolution approach is our decades of familiarity with DNS and the current Internet; handling mobility would be a drop-in replacement to DNS provided we can address the challenge of building a distributed system that can scale to billions of devices making tens of updates a day and yet return responses within a few milliseconds. Addressing this challenge forms the focus of the rest of the paper.

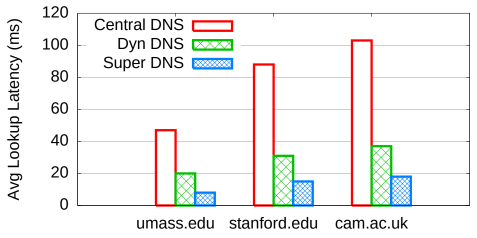


Figure 1: Managed DNS provides lower address lookup latencies than centralized authoritative name servers.

2.2 Why not just adapt DNS?

While architectural positions like above are endlessly debatable, a more pressing question is: what if any are the limitations of DNS as a global name resolution service under high mobility and how can it be retrofitted to address those limitations?

The scalability challenge of high mobility would be felt most strongly at the authoritative name servers for two reasons. First, they would need orders of magnitude higher updates rates from mobile device names a couple orders of magnitude more in number compared to domain names today ($\approx$10B vs 146M [37]). Second, authoritative name servers would be unable to take advantage of TTL-based caching for mobile device names as much as for today's domain names. TTLs of A-type records in DNS are more than 5 minutes for more for 40% of records and more than 1 hour for 10% of records[36], which implies a commensurate service outage time as updates take that much time to fully propagate. In practice, actual update rates to DNS records are orders of magnitude lower than that indicated by TTLs that are set conservatively so as to reduce outage times due to unanticipated updates; yet, ironically, outage times can often last a day or more either because of poorly planned updates or because some resolvers do not respect the TTL limits much to the woe of server administrators [14]. In contrast, mobile devices would require smaller TTLs to ensure reachability and can be expected to make tens of updates per day or more on average (and more frequently in short bursts such as in vehicular WiFi environments [22]). Smaller TTLs imply higher load.

To handle high mobility, we expect most end-users to outsource their authoritative name service to managed DNS providers. Managed DNS providers today are commonly geo-replicated and allow their customers to leverage better performance, availability, economies of scale, and ease of management compared to maintaining a centralized authoritative name server by themselves. However, managed DNS providers today use simplistic strategies such as *replicate-all* or *static-k* that respectively replicate a name record at all available locations or a fixed set of packaged locations (e.g. Dyn DNS offers replicate-at-all package for $180/yr and replicate-at-4 for $30/yr). We argue that these simplistic strategies provide poor cost-vs-performance tradeoffs even for today's (mostly) non-mobile names, a problem that will be further exacerbated by high mobility.

To exemplify our argument for non-mobile names, consider Figure 1 that shows lookup latencies to centralized authoritative name servers of three domain names (CentralDNS), and their projected lookup latencies for two managed DNS services: (1) Dyn DNS, a leading provider with 17 known locations [17] and (2) SuperDNS, a hypothetical managed DNS with 200 (PlanetLab) locations but replicates a name at only 17 out of the 200 locations where the domain name is highly popular. We calculate latencies based on measurements from 100 other PlanetLab locations. We measure lookup latency for Dyn at a location by querying Dyn servers for the address of twitter.com, one of its customers. The latency of SuperDNS at a location is the measured round-trip delay to the nearest replica of the name record. We take the weighted average of lookup latencies across all locations weighted by the popularity of a domain in the geographic area (calculated using the Alexa dataset [1]) for which that location is geographically closest.

Unsurprisingly, managed DNS services do improve performance over a centralized service, e.g., Dyn reduces lookup latency for cam.ac.uk by $2.7\times$. However, its simplistic replication policy leaves significant room for improvement compared to SuperDNS that leverages its massively geo-distributed deployment and intelligent replica placement, to give up to $2.5\times$ better performance than Dyn for the same number of replicas. If SuperDNS were to replicate each name at all 200 locations, its update costs would increase by nearly $12\times$. To limit update costs, if SuperDNS were to follow a static-k policy and choose 17 locations randomly, it would be unable to effectively use its massively geo-distributed deployment to minimize latencies. Highly mobile device names would further exacerbate the cost-benefit tradeoffs of such simplistic replicated strategies as they would not account for the orders of magnitude higher update costs.

3 Auspice design & implementation

3.1 Design goals

In this section, we present the design and implementation of Auspice, a massively scalable distributed global name resolution service designed for high mobility. The design of Auspice is motivated by the following specific goals:

(1) Low lookup latency: The design must ensure low latencies for name lookups. Any endpoint or network entity performing a lookup should get the look-and-feel of being served by a centralized service located tens of milliseconds away. Our implicit goal here is to not only support DNS-like use-cases today but also to accommodate other architectural proposals that involve late-binding of names to addresses, e.g., routers close to the destination re-binding an identifier to a different address in mobile or multi-homed settings [51, 12]; enabling mid-connection mobility with minimal outage times [59, 20], etc.

(2) Low update cost: The design must ensure a low update cost. A naive way to minimize lookup latencies is to replicate every name record at every available location, however high mobility implies high update rates and the cost of pushing each update to every replica can be prohibitive. Worse, load hotspots can actually degrade lookup latencies instead of improving them.

(3) High availability: The design must be resilient to failures of nodes including disasters impacting an entire datacenter or stub network. By consequence, the design must also prevent crippling load hotspots.

(4) Consistency: The system must achieve the above objectives while ensuring flexible consistency semantics for name records as specified by their owners. A naive way to achieve the first two goals is to use aggressive caching with large TTLs, clearly an unusable scheme in mobile scenarios as inconsistency implies unreachability.

(5) Extensibility: The design must be extensible to supporting a rich set of attributes associated with a name and policies for dealing with mobility, multihoming (e.g.,"prefer WiFi to cellular"), etc. The design should be agnostic to how names, addresses, and resolution policies are represented by Internet architectures of the future, while being easily deployable in today's Internet.

(6) Security: The design must be robust to malicious user behavior such as hijacking or corrupting name records. The design must support flexible access control policies to control both read and write access.

To address the above goals, Auspice adopts a massively geo-distributed key-value store design. The geo-distribution is essential to the latency and availability goals while the key-value design enables extensibility. Below, we first describe Auspice's distributed system design so as to achieve the first four goals, and then describe how it achieves the extensibility and security goals. For the former, it suffices to think of a *name* as a domain name, a *name record* as the associated zone file, and a *name server* as analogous to a DNS name server. For ease of exposition, we assume one name server per geo-location.

3.2 Auspice's distributed design

At the core of Auspice is a placement engine that achieves the latency, cost, and availability goals by adapting the number and locations of replicas of each name record in accordance with (1) the lookup and update request rates of the name, (2) the geo-distribution of requests for that name, and (3) the aggregate request load on the system across all names. The key to ensuring high availability and flexible consistency is a two-tier consensus engine for each name; the first "control" tier infrequently recomputes and migrates the current set of replicas of each name record, and the second "data" tier upon each write or read coordinates with other replicas to ensure the necessary consistency semantics.

Figure 2 illustrates the architecture for Auspice. Each name is associated with a fixed number, M (=3 default), of *replica-controllers* and a variable number of *active replicas* of the corresponding name record. The number and locations of the replica-controllers is fixed and is computed using M well-known consistent hash functions each of which maps the name to a name server location. The replica-controllers are responsible only for determining the number and locations of the active replicas, and the actives replicas are responsible for keeping name records updated and responding to client's requests. The replica-controllers implement a replicated state machine using Paxos [46] so as to maintain a consistent view of the current set of active replicas despite failures.

Computing the active replica locations for each name proceeds in epochs as follows. At bootstrap time, the active replicas are chosen to be physically at the same locations as the corresponding replica-controllers. In each epoch, the replica-controllers obtain from each active replica a summarized load report that contains the request rates for that name from different *regions* as seen

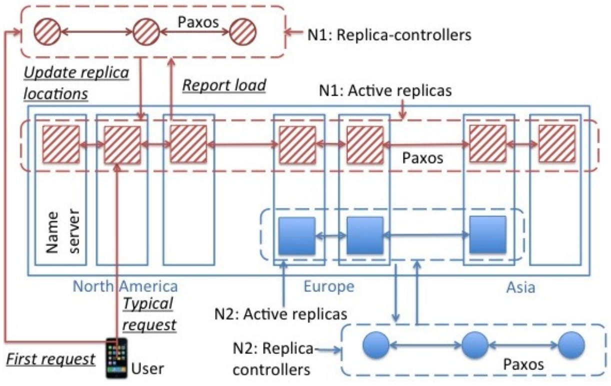


Figure 2: Geo-distributed name servers in Auspice. Replica-controllers (logically separate from active replicas) decide placement of active replicas and active replicas handle requests from end-users. N1 is a globally popular name and is replicated globally; name N2 is popular in select regions and is replicated in those regions.

by that replica. Regions could either be IP prefixes or geographic regions, e.g., cities, that partition users into non-overlapping groups so as to capture locality. Each active replica's load report consists of a spatial vector of request rates as seen by that replica. The replica-controllers aggregate these load reports to obtain a concise spatial distribution of all requests for the name.

3.2.1 Placement algorithm

In each epoch, the replica-controllers use a placement algorithm that takes as input the aggregated load reports and capacity constraints at name servers to determine the number and locations of active replicas for each name so as to minimize client-perceived latency. We have formalized this *global* optimization problem as a mixed-integer program and shown it to be computationally hard. As our focus is on simple and efficient algorithms, we defer the details of the optimization approach to Appendix A, and use it only as a benchmark in small-scale experiments against Auspice's heuristic algorithm.

Auspice's placement algorithm is a simple heuristic and can be run *locally* by each replica-controller. The placement algorithm computes the number of replicas using the lookup-to-update ratio of a name in order to limit the update cost to within a small factor of the lookup cost. The number of replicas is always kept more than the minimum number needed to meet the availability objective under failures. The location of these replicas are decided to minimize lookup latency by placing a fraction of replicas close to pockets of high demand for that name while placing the rest randomly so as to balance the potentially conflicting goals of reducing latency and balancing load among name servers.

Specifically, the placement algorithm computes the number of replicas for a name as $(M + \beta R_i/W_i)$, where

R_i and W_i are the lookup and update rates of name i, M is the minimum number of replicas needed to meet the availability goal (§3.1), and β is a replication control parameter. The placement algorithm dynamically adjusts β to control the number of replicas so as to trade off latency benefits against update costs given capacity constraints as follows. In each epoch, the replica-controllers recompute β so that the aggregate load in the system corresponds to a predetermined threshold utilization level μ (= 0.7 default). For simplicity of exposition, suppose read and write operations impose the same load, and the total capacity across all name servers (in reads/sec) is C. Then, β is set so that

$$\mu C = \sum_i R_i + \sum_i (M + \beta \frac{R_i}{W_i}) W_i \qquad (1)$$

where the right hand side represents the total load summed across all names. The first term in the summation above is the total read load and the second is the total write load. Having computed β this equation, Auspice computes the locations of replicas for name i as follows. Out of the $M + \beta R_i/W_i$ total replicas, a fraction, f (=0.5 default), of replicas are chosen according to locality, i.e., the replica-controllers use the spatial vector of load reports to select $K = f \times (M + \beta R_i/W_i)$ name server locations that are respectively the *closest* to the top K regions sorted by demand for name i. The remaining $M + \beta R_i/W_i - K$ are chosen randomly without repetition.

The top-K "closest" replicas above are chosen as the closest with respect to round-trip latency plus load-induced latency measured locally at each name server. An earlier design simply chose the top-K according to round-trip locality alone; however, we found that adding load-induced latencies in this step in addition to choosing the remaining replicas randomly ensures better load balance and results in lower overall user-perceived latency.

3.2.2 Routing client requests

Next, we describe how individual requests from end-hosts are routed to a suitable name server. The list of all name servers is known to each name server and can be obtained by contacting any name server. End-hosts can either directly send requests to a name server or channel them through a local name server like today. When a local name server encounters a request for a name for the first time, it uses the known set of all name servers and hash functions to determine the replica-controllers for that name and sends the request to the closest replica-controller. The replica-controller then returns the set of active replicas for the name and the client resends the request to the closest active replica. In practice, we expect replica-controllers to be contacted infrequently as the set of active replicas can be cached and reused.

To decide the closest active replica and the closest replica-controller, local name servers maintain latency estimates to name servers that reflect both network latency and load-induced latency. As network latency estimates change slowly, local name servers maintain round-trip latency estimates to all name servers using infrequent ping measurements. To track the server load-induced latency, the latency estimate to a name server is passively measured as a moving average over lookups sent to that name server. The local name server also maintains a timeout value based on the moving average and variance of the estimates (similar in spirit to TCP). If a lookup request sent to a name server times out, the local name server infers that either the route to the server is congested or the server is overloaded, and it increases its latency estimate to that name server by a fixed factor (=1.1 default). Overall, this method ensures that if multiple lookups sent to a name server time out, the estimated latency becomes very high, and the local name server stops sending requests to that name server.

3.2.3 Consistency: Why and how

The above proximity-based strategy can route both lookup (or read) and update (or write) requests for a name to any active replica. In order to ensure low lookup latency, it is important for them to be served locally from an active replica. This means that the onus of replica coordination in order to maintain any nontrivial consistency semantics falls on the update procedure. But why should we care about consistency when DNS and network protocols in general prize liveness over consistency[40]?

As a global name-to-address resolution service, Auspice must at a minimum ensure the following property: *if there are no more updates to a name record, all active replicas must eventually return the same value of the name record and, in a single-writer scenario, this value must be the last update made by the (only) client.* Note that not satisfying this property means that a mobile client may be rendered persistently unreachable even though it is no longer changing its network address(es). It would appear that this weak eventual consistency property can be easily satisfied if the client associates a sequence number with each update; an active replica can simply locally record the write, return to the client, and lazily propagate the update to other active replicas.

Unfortunately, the simplistic approach above has some problems. As the set of active replicas for a name can change over time, a write-to-one approach can lose the most recent write if the active replica that received the write crashes and before it recovers, the replica controllers change the set of active replicas. Even if the set of active replicas remains unchanged, the window of inconsistency (or unreachability) is higher with a write-to-one approach if the active replica that received the write fails. Most importantly, we anticipate that expressive name records (see §3.2.5) may be updated by multiple authorized clients via different active replicas, and it is important to ensure that update operations (like appending to or deleting from a list) to a name are applied in the same order by all active replicas. These requirements motivate a state-machine approach to handle updates.

Total write ordering. For each name, Auspice ensures a total ordering of all updates across all active replicas. To this end, active replicas for a name participate in a Paxos instance maintained separately for each name (distinct from Paxos used by replica-controllers to compute active replicas). This of course implies that updates can make progress (Availability) only when a majority of active replicas can communicate with each other (Partition-tolerance) while maintaining safety (Consistency) [4].

Flexible consistency. Totally ordering writes to each name makes it easy for Auspice to support three different consistency models—eventual, sequential, and linearizable. A name owner can choose the desired consistency for their name. The consistency models pertain only to operations to a single name (e.g. reads or writes to different attributes of that name), and there are no consistency guarantees on operations spanning different names. All updates to a name record are executed after being committed by Paxos among active replicas. The execution of lookups determines the consistency semantics for a name. Auspice supports the following three semantics:

(1) Eventual: Paxos guarantees that all replicas will eventually store identical copies of the name record. A lookup request can be simply processed locally at a replica as in §3.2.2.

(2) Sequential: Two properties must be satisfied to provide sequential consistency: (a) read-your-writes: a lookup for a name by a client sees the result of the most recent committed update by that client for the name. (b) monotonic reads: a lookup by a client always reads a more recent or the same state of the name record compared to the previous lookup by that client (irrespective of the replica the client sends the lookup to).

Sequential consistency is implemented by leveraging client support. A lookup request is processed locally at a replica, similar to the eventual consistency model. However, responses to both lookups and updates sent to a client include the sequence number of the most recent committed update by the Paxos instance at that replica. A client accepts responses to the next lookup only if the the sequence number returned with the response is equal or greater than the sequence number received along with the response of the previous request (lookup or update). Otherwise, the response is discarded and lookup is sent to a different active replica.

(3) Linearizable: A lookup must read the result of all committed writes until that time to satisfy the linearizability model. To this end, lookups (in addition to updates) are executed after being committed by Paxos among active replicas. Linearizability is the strongest consistency model, but results in higher lookup latencies compared to other two consistency models as it entails a total ordering of both reads and writes.

3.2.4 Handling active replica group changes

As the group of active replicas can change over time, consistency guarantees depend on safely handling a change in the group of active replicas. The safety property can be stated as follows: *the group of active replicas in epoch $i+1$, before executing any requests, must obtain identical copies of the name record and that copy must include all committed updates to the name record made by the group of active replicas in epoch i.*

The key protocol for group change is Stoppable Paxos [47]. Stoppable Paxos supports a special STOP request that can be committed only once and is always the last request committed by the Paxos instance. The commit of the STOP request ensures that no further changes to the record will be made by the current group of active replicas. The next group of active replicas, before executing any requests, copy the name record from any of the current active replicas that have committed the STOP request. This guarantees that each member in next group of active replicas starts with an identical copy of the name record. Thus, STOP acts as a link between the two otherwise independent Paxos instances per name: maintained by replica-controllers and active replicas respectively.

3.2.5 Extensibility

For extensibility, Auspice is implemented as a general-purpose, geo-distributed key value store. An Auspice *name* acts as the primary key and is an arbitrary bounded-length string while a *name record* is the value represented as an associative array of "super columns". This super-column family representation allows us to store a flexible set of attributes that do not have to be defined a priori. For example, Auspice can store context attributes like geolocation in MobilityFirst [12] or represent evolvable addresses in XIA [38]. Indeed, a name record can be anything that can be represented as a JSON object.

In line with the vision of the MobilityFirst future Internet architecture [12, 61], our position is that a logically centralized global name service should capitalize on its role as the first step in network communication and go beyond simple name-to-address resolution; Venkataramani et al [61] describe how a global name service can significantly enhance network-layer functionality. As part of this broader effort, we have internally used Auspice to develop novel network-layer functions such as simultaneous mid-connection mobility and context-aware communication and used the latter to develop an emergency geocast application (see YouTube demo [12]).

3.2.6 Access control and privacy

We have developed extensive and intrinsic support for self-certifying GUIDs (or globally unique identifiers) as one type of name in Auspice. Each human-readable name in Auspice is first translated to a GUID that is simply a hash of the public key associated with the name. *Every* top-level column in Auspice has access control lists that could either be a whitelist (or blacklist) of GUIDs allowed (disallowed) to perform a read or write operation (in some cases, we have found an append-only ACL distinct from the write ACL to be useful). By default, all columns and network addresses in particular can be read by all but written to only using the private key corresponding to the GUID. Some *keyword* attributes like netAddress and geoLocation have special support (unlike developer-defined attributes), for example, in order to efficiently maintain indexes for attribute-based reverse lookups or for non-default privacy policies like allowing only whitelisted reads for geoLocation.

3.2.7 Deployment path

With modest additional effort, Auspice can be deployed today as a massively scalable managed DNS provider. In order to use Auspice, a domain name owner simply has to set their authoritative name servers to any number of Auspice name servers. Name owners can use the DNSSEC DNSKEY record to derive the GUID and continue to rely on delegation-server based chain of trust model. In architectures like MobilityFirst, XIA, or HIP, we expect Auspice to be deployed in a federated manner where multiple providers may run separate Auspice instances and mobile endpoints can obtain global name service from a provider of their choice. These architectures implicitly assume a name certification service (NCS) that first translates a human-readable name to a self-certifying GUID; this NCS can also supply the name of the provider that provides global name service for that GUID. Currently, we have rolled in a simple NCS into Auspice itself, which through a developer portal (`http://gnrs.name`) binds a user-specified or system-selected GUID to a human-readable name that is simply an email address, i.e., our proof-of-concept NCS is a poor man's CA relying on email-based verification.

3.2.8 Implementation details

The core of Auspice is implemented in Java with 28K lines of code. We have been running an alpha deployment for research use for several months across eight EC2 regions with support for an HTTP interface [10]. We have implemented support in Auspice for two pluggable NoSQL data stores, MongoDB (default) and Cassandra, as persistent local stores at name servers. We do not rely on any distributed deployment features therein as the coordination middleware is our novel contribution.

We implemented the two-tier Paxos engine from scratch. Each Auspice name server manages a very large number of Paxos instances, one for every active replica and replica controller at the node. Our Paxos implementation requires a small constant amount of state[2] for each instance that is currently stored in-memory as a Java object. However, it is feasible to store this state in the data store with some reduction in per-node throughput; this would reduce memory pressure and allow a single server to store a much larger number of name records.

4 Evaluation

Our main goal is to quantify the benefits and costs of the choices in Auspice's distributed design—our main contribution—with respect to the subset of design goals (§3.1) that are quantifiable, namely client-perceived latency benefit and provider-perceived update cost under high mobility. We use the implemented prototype to evaluate (1) Auspice's replication strategies against simple ones used by DNS providers today as well as DHT-based alternatives proposed in research, and (2) a deployment of Auspice against several commercial managed DNS providers in live operation. We use simulations to evaluate (3) the sensitivity of Auspice's benefits with respect to mobile workload parameters, and (4) the scalability of Auspice to regimes beyond those permitted by our testbeds. We do not attempt to evaluate Auspice's functional design goals—resilience to failures, consistency, extensibility, security—except to the extent that all experiments subsume the overhead of these features.

4.1 Experiment setup

Testbeds: We use two testbeds, PlanetLab and a local cluster. The PlanetLab setting consists of 80 nodes for name servers and 80 for local name servers (assumed co-located with end-hosts). The cluster consists of 16 machines (Xeon 5140, 4-core, 8 GB RAM) with 8 each used for name servers and local name servers. Each of these machines runs 10 name server or local name server instances so that each instance can be mapped to a distinct PlanetLab node; the emulated round-trip delay between two nodes is equal to measured round-trip delay between the corresponding PlanetLab nodes. Most of our system experiments were performed on both testbeds, but we report the corresponding cluster results as they are more consistently reproducible.

Workload: A vexing evaluation challenge is that we do not have a workload of clients querying a name service in order to communicate with mobile devices moving across different network addresses. There is no

[2]This in-memory state is distinct from Paxos logs used for recovery after crashes that must be maintained on disk for correctness.

Workload parameter	Value
(Mobile) device names	10,000
(Mostly static) service names	1000
% of device name lookups	33.33%
% of device name updates	33.33%
% of service name lookups	33.33%
% of service name updates	0.01%
Geo-locality	0.75

Table 2: Default experiment parameters (except §4.5).

global name resolution infrastructure for mobile device names today and most mobile devices do not have publicly visible addresses, so no one queries for them.

So we are forced to make up a synthetic workload of requests to mobile *device names* and carefully analyze the sensitivity to workload parameters. The following are default experimental parameters (except for §4.5 on sensitivity analysis). The ratio of the total number of lookups across all devices to the total number of updates is 1:1, i.e., devices are queried for on average as often as they change addresses. The lookup rate of any device name is uniformly distributed between 0.5 to 1.5 times the average lookup rate; the update rate is similarly distributed and drawn independently, so there is no correlation between the lookup and update rate of a name.

How requests are geographically distributed is clearly important for evaluating a replica placement scheme. We define the *geo-locality* of a name as the fraction of requests from top-10% of regions where the name is most popular. This parameter ranges from 0.1 (least locality) to 1 (high locality). For a device name with geo-locality of g, a fraction g of the requests are assumed to originate from N=10% of the local name servers, the first of which is picked randomly and rest N-1 are the local name servers geographically closest to it. How do we pick a reasonable g? With admittedly little basis, we pick $g = 0.75$ for device names, i.e., the top-10% of regions in the world will account for 75% of requests to device names, an assumption not altogether unreasonable given that communication and content access today exhibits a high country-level locality [45, 64].

In addition to device names, *service names* constitute a small fraction (9%) of names in the workload and are intended to capture web services like today with low mobility. Their lookup rate (or popularity) distribution and geo-distribution are used directly from the Alexa dataset[3] [1]. Using this dataset, we calculated the geo-locality exhibited by the top 100,000 websites to be 0.8. Updates for service names are a tiny fraction (0.01%) of lookups, as web services can be expected to be queried much more often than they are moved around. The lookup rate of service names is a third of the total number of requests (same as the lookup or update rates of device names).

[3]Note that we do not rely on Alexa at all for mobile device names.

Table 2 summarizes the default workload parameters.

4.1.1 Schemes compared

Auspice uses the default parameter values as described in §3. We compare against the following other schemes. **Static-3** replicates each name at three random locations, so it has a low update cost and an even distribution of names among name servers. **Replicate-All** replicates all names at all locations, in the hope of optimizing latency but ignoring update cost. All of these three schemes direct a lookup to the closest available replica.

CoDoNS [56] replicates name records using the Pastry DHT. The number of replicas is chosen based on the popularity ranking of a name and the location of replicas is decided by consistent hashing. In our implementation, each request is directly sent to the replica that would have received this request if Pastry routing were followed, i.e., the latency we report would be smaller than the actual latency in CoDoNS. We set the Zipf exponent to be 0.63 calculated based on our workload. The average hop count is set so that CoDoNS creates the same number of replicas as Auspice for a fair comparison.

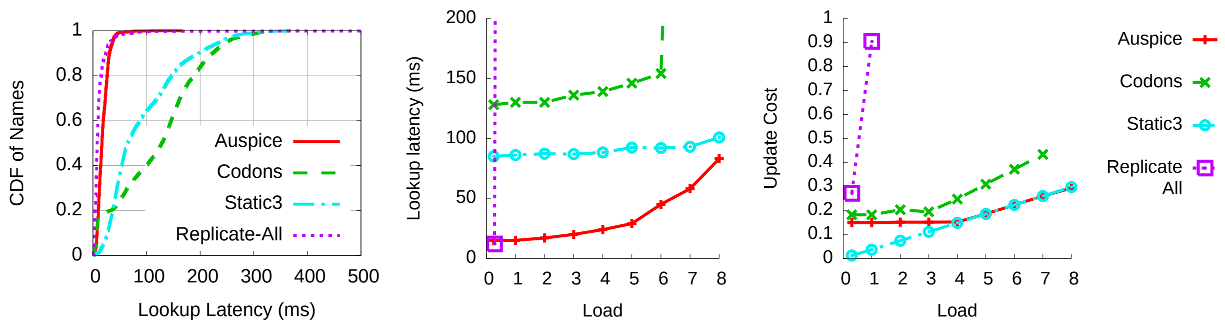
4.2 Comparison of replication schemes

4.2.1 Lookup latency

This experiment compares the lookup latency of schemes across varying load levels. A machine receives 200 lookups/sec and 100 updates/sec at a load = 1. For every scheme, load is increased until 2% of requests (lookups and updates) fail, where a failed request means no response is received within 10 sec. The experiment runs for 10 mins for each scheme and load level. To measure steady-state behavior, both Auspice and CoDoNS pre-compute the placement at the start of the experiment based on historical knowledge of the workload.

Figure 3(a) shows the distribution of median lookup latency across names at the smallest load level (load = 0.3). Figure 3(b) shows load-vs-lookup latency curve for schemes; lookup latency metric is the mean of the distribution of median lookup latencies of names. Figure 3(c) shows the corresponding mean of the distribution of update cost across names at varying loads; the update cost for a name is equal to the number of replicas times the update rate of that name.

Replicate-All gives low lookup latencies at the smallest load level, but generates a very high update cost and can sustain a request load of at most 0.3. This is further supported by Figure 3(c) that shows that the update cost for Replicate-All at load = 0.4 is more than the update cost of Auspice at load = 8. In theory, Auspice can have a capacity advantage of up to N/M over Replicate-All, where N is the total number of name servers and M is



(a) Lookup latency distribution, load = 0.3 (b) Lookup latency-vs-load (c) Update cost-vs-load

Figure 3: [System] Auspice gives up to 5.7× to 9× lower latencies over Static-3 and CoDoNS respectively (Figure 3(b), load = 1). Replicate-All can sustain a request load of at most 0.3; Auspice can sustain a request load of up to 8.

the minimum of replicas Auspice must make for ensuring fault tolerance (reps. 80 and 3 here).

Static-3 can sustain a high request load (Fig. 3(b)) due to its low update costs. Its lookup latencies are high as it only creates a small number of replicas randomly.

Auspice gives up to 5.7× to 9× lower latencies over Static-3 and CoDoNS respectively (Figure 3(b), load = 1). This is because is places a fraction of the replicas close to pockets of high demand, unlike the other two schemes. At low to moderate loads, servers have excess capacity than the minimum needed for fault tolerance, so Auspice creates as many replicas as possible without exceeding the threshold utilization level (refer to Equation 1), thereby achieving very small latencies for loads ≤ 4.

At loads ≥ 4, servers exceed the threshold utilization level even if Auspice creates the minimum number of replicas needed for fault tolerance. This explains why Auspice and Static-3 have equal update costs for loads ≥ 4 (Figure 3(c)). Reducing the number of replicas at higher loads allows Auspice to limit the update cost and sustain a maximum request load that is equal to Static-3.

CoDoNS has higher lookup latencies as it places replicas using consistent hashing without considering the geo-distribution of demand. Further, it answers lookups from a replica selected enroute the DHT route. Typically, the latency to the selected replica is higher than the latency to the closest replica for a name, which results in high latencies. CoDoNS replicates 22.3% most popular names at all locations. Lookups for these names go to the closest replica and achieve low latencies; lookups for remaining 77.7% of names incur high latencies.

CoDoNS incurs higher update costs than Auspice even though both schemes create nearly equal total number of replicas at every load level. This is because CoDoNS decides the number of replicas of a name only based on its popularity, i.e., lookup rates, while Auspice decides the number of replicas based on lookup-to-update ratio of names. Due to its higher update costs, CoDoNS can sustain a lesser request load than Auspice.

4.2.2 Update latency

We measure the update latency as the time difference between when a client sends an update and when it receives confirmation from a replica. Figure 4 shows the distribution of median update latencies from the the experiment in Section 4.2.1 for load = 0.3. The median and 90th percentile update latency for Auspice is 284 ms and is comparable to that of other schemes. A request, after arriving an active replica, takes four one-way network delays to be committed by Paxos, which explains why update latency of all schemes is a few hundred ms.

Paxos' consistency guarantees come at the cost of increased update latencies. We quantify how much update latencies could be reduced with the following update protocol, called *lazy-update*, which gives eventual consistency: send confirmation to client after updating locally, and then propagate the update to other replicas. A client would get a confirmation sooner in case of lazy-update, but the update might take longer to be actually received by all replicas. In another experiment, we measured the time lazy-update takes to propagate the update to all replicas. We measure the median of the distribution of time to update all replicas to be 154 ms for lazy-update. The corresponding median values for Paxos is 292ms. Thus, Paxos increases the latency to propagate updates to all replicas by 1.8× over eventual-consistency.

4.3 Auspice vs. managed DNS providers

Having analyzed a synthetic workload dominated by mobile device names, we next ask how Auspice compares to commercial managed DNS providers in serving their customers' domain names. These state-of-the-art providers such as UltraDNS, DynDNS, and DNS-MadeEasy [16, 8, 7] offer a geo-replicated authoritative DNS service and are widely used by enterprises today.

Lookup latency: We compare Auspice to a leading managed DNS provider for a workload of lookup requests for domain names serviced by the provider. We identify 316 domain names among top-10K Alexa web-

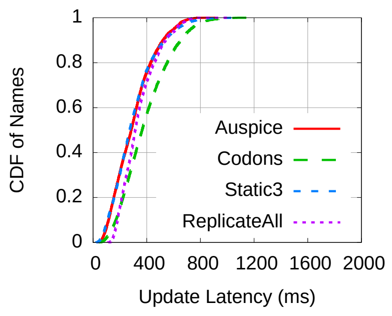


Figure 4: [System] Median update latency of Auspice is 284ms and is comparable to those of other schemes.

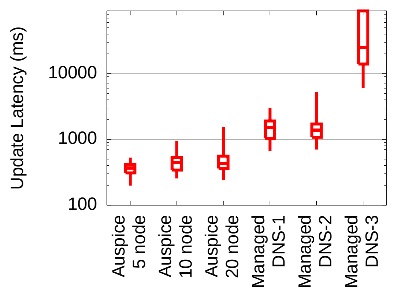


Figure 5: Median update latency of Auspice for updating replicas at 5 locations is 1.0 sec to 24.7 sec lower than three managed DNS providers.

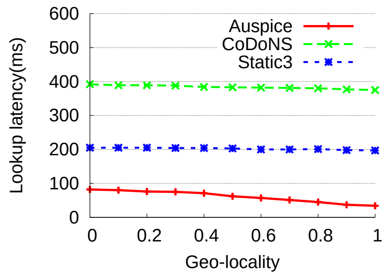


Figure 6: [Simulator] Workload sensitivity. Auspice gives 2× to 5× lower latencies across all locality levels.

sites [1] serviced by this provider. The geo-distribution of lookups for each name is determined from the Alexa dataset [1]. For each name, we measure performance for 1000 lookups across 100 PlanetLab nodes. We ensure that lookups are served from the name servers maintained by the provider by requesting the address for a new random sub-domain name each time, e.g,, xyz.google.com instead of google.com. This name is unlikely to be cached and requires an authoritative name server lookup. Auspice name servers are deployed on 80 PlanetLab locations while the managed DNS has 16 known locations of server deployments [17]. For an even comparison between Auspice and the provider, we limit the maximum number of replicas/name for Auspice to 5, which is less than one-third the number of locations of the provider [17].

	Auspice	**Managed DNS**
Number of replicas	5	16
Median Lookup Latency	45	40
Mean Lookup Latency	74	72

Table 3: Auspice has latencies comparable to a managed DNS provider with less than one-third replica locations.

Table 3 shows median lookup latencies across names for Auspice and for the managed DNS provider. Auspice performs within 11% of the managed DNS provider for all latency metrics. While Auspice creates less than one-third replicas as the managed DNS provider, it places them considering the geo-distribution of demand. Due to its placement strategy, Auspice achieves similar latency with smaller cost of updates. This experiment shows that Auspice provides latencies comparable to the state-of-the-art DNS solutions with much smaller update costs due to a judicious choice of replica locations.

Update latency: To measure update latencies, we purchase DNS service from three providers for separate domain names. All providers replicate a name at 5 locations across US and Europe for the services we purchased. We issue address updates for the domain name serviced by that provider, and then start lookups to the authoritative name servers for our domain name. These authoritative name servers can be queried only via an anycast IP address, i.e., servers at different locations advertise the same externally visible IP address. Therefore, to maximize the number of provider locations queried, we send queries from 50 random PlanetLab nodes. From each location, we periodically send queries until all authoritative name server replicas return the updated address. The update latency at a node is the time difference between when the node starts sending lookup to when it receives the updated address. The latency of an update is the the maximum update latency measured at any of the nodes. We measure latency of 100 updates for each provider.

To measure update latencies for Auspice, we replicate 1000 names at a fixed number of PlanetLab nodes across US and Europe. The number of nodes is chosen to be 5, 10, and 20 across three experiments. A client sends an update to the nearest node and waits for update confirmation messages from all replicas. The latency of an update is the time difference between when the client sent an update and when it received the update confirmation message from all replicas. We show the distribution of measured update latencies for Auspice and for three managed DNS providers in Figure 5.

Auspice provide lower update latencies than all three providers for an equal or greater number of replica locations for names. It is not clear to us why “Managed DNS 3” update latencies are an order of magnitude higher than global propagation delays[4]. This finding is consistent with a recent study [17], which has shown update latencies of up to tens of seconds for multiple providers.

[4]They tend to not be forthcoming with proprietary internal details.

4.4 Simulator validation

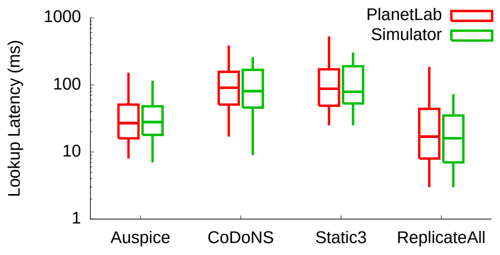


Figure 7: Latency on PlanetLab compared to simulator. 5, 25, 50, 75 and 95 %-ile latencies are shown.

To ensure the accuracy of the simulator, we first validate it based on an experiment on PlanetLab. In simulation, server latency is calculated using a queueing-theoretic model [42] and network latency between two nodes is the measured RTT between corresponding PlanetLab nodes. We introduce a similar packet loss rate in the simulator as seen on PlanetLab. Figure 7 shows a box plot of the distribution of median lookup latencies of names in the PlanetLab experiment and in the simulator. We find that median latencies for all schemes in the simulator are within 8% of that on PlanetLab. The 95% latencies are higher on PlanetLab experiments than in simulator due to unpredictable wide-area latencies and server processing delays.

4.5 [Simulation] Sensitivity analysis

We analyze the sensitivity of Auspice's benefits and costs to the workload parameters used in §4.2. In order to be able to expire a wider range of parameters and scales, we use a custom simulator that simulates round-trip latency, loss, and server load-vs-response time behavior as measured on PlanetLab. Experiments here use 10K name servers, 2K local name servers, 10K service names, and 100K device names.

Geo-locality: Figure 6 compares the latency for workloads with varying levels of geo-locality. Both Static-3 and CoDoNS are choose replica locations randomly, and therefore their latency remains the same irrespective of workload locality. But Auspice can achieve better latencies as the geo-locality in the workload increases. Even in a workload with no locality ($g = 0.1$), Auspice outperforms Static-3 by $2\times$ because it creates more than three replicas for each name, and outperforms CoDoNS by $4\times$ because it redirects requests to the closest replica of a name unlike CoDoNS.

Ratio of device names to service names: This experiment evaluates schemes for workloads with different ratios of device names to service names, called *DS-ratio* for short. We fix the number of service names to be 10K and vary the number of device names between 1000 to

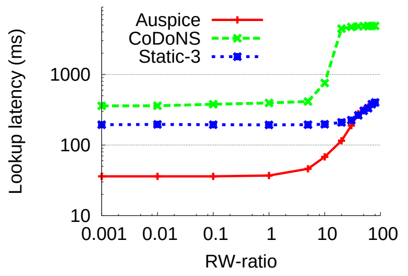


Figure 8: Lookup-to-update ratio for device names. For a lookup-dominated workload (RW-ratio = 10), Auspice has $2.95\times$ lower latency than Static-3. Replicate-All is unsustainable due to high update costs and is not shown.

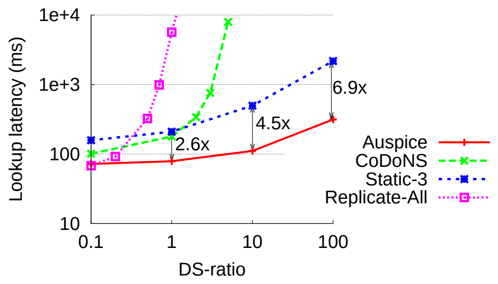


Figure 9: [Simulator] Auspice gives greater latency gains over Static-3 as the number of device names increases in the workload.

1,000,000. Figure 9 presents our results. Replicate-All saturates server capacity for a workload with DS-ratio = 1 due to high update costs. Auspice supports workloads with DS-ratio up to 100 as it minimizes the update cost for device names. Due to its locality-aware design, Auspice has $2.6\times$, $4.5\times$ and $6.9\times$ lower latency than Static-3 when DS-ratios are 1, 10 and 100 respectively.

Lookup-to-update ratio: We vary the ratio of lookups to updates, termed as *RW-ratio*, by increasing the number of lookups in the workload, but keeping the number of updates fixed. Figure 8 shows that Auspice provides lower latencies for both write-dominated workload (RW-ratio < 1) as well as read-dominated workloads (RW-ratios > 1). As RW-ratios increase beyond 1, Auspice handles the increase in number of lookups in the workload by decreasing the replication parameter β (refer to Equation 1). Lower β values reduce number of replicas and hence the update costs for Auspice, which helps Auspice accommodate workloads with RW-ratio $>$ 1. Reduced number of replicas increases lookup latency of Auspice, but still Auspice has $2.95\times$ lower latency than Static-3 for RW-ratio = 10.

4.6 [Simulation] Comparison to Optimal

We have compared Auspice to Optimal based on an optimization formulation of the placement problem (Appendix A). Optimal takes as input the set of names, their request geo-distribution, the capacity of name servers, network latency between local name servers and name servers, and a load-vs-response-time curve at each name server, and computes replica placement so as to minimize the sum of network and server latency. For a similar workload as in Section 4.2.1, we find that the latency for Auspice is between 1.1×-2.1× of the Optimal across all load levels. Optimal performs better as it can globally optimize server resource allocation across all names, but Auspice uses a decentralized placement algorithm to independently decide replica placement for each name. In ongoing work, we are evaluating Optimal using the testbed; this is nontrivial partly because Optimal must know the exact load-vs-response time behavior, which is not always stationary or easy to measure, so we conjecture that the clean simulator environment overestimates the benefits of Optimal.

4.7 TTL-value selection

Auspice internally uses active replication, but does allow TTL-based caching at local name servers and clients. What TTL should names use? TTL-based caching means that `connect(name,port)` calls from end-hosts can occasionally time out because the destination `name` has moved. In this case, end-hosts must send a refresh query to Auspice and attempt to reconnect. The overall time-to-connect to a name depends on the name's update rate, lookup rate, and the TTL. We have developed a simple analytical model to calculate the optimal TTL based on these three values (Appendix B), to serve as a recommendation to name owners. Using simulations of a TTL-based cache, we show that the optimal TTL values predicted by our model are effective in minimizing connection-setup delay to names with a wide range of update to lookup ratios.

The time to setup a connection to a name including the time to lookup the name record from the cache depends on the current state of the cache. The connection time is shortest in case of a cache hit because name record lookup from the cache adds near zero overhead to the connection establishment time. The connection time is slightly longer if TTL has expired because the cache looks up the name record from an active replica in Auspice. The connection time is longest if TTL has not expired but the cached address is no longer correct. The user first attempts to connect to the incorrect address of the name and after a timeout, forces the cache to get the correct name record from Auspice and finally establishes the connection. The factor by which connection setup time increases due to a name lookup is taken to be $l_1 = 1$, $l_2 = 1.5$, and $l_3 = 4$ respectively for the above three cases. For simplicity, we assume that cache evictions due to addition of other name records to the cache have a negligible probability.

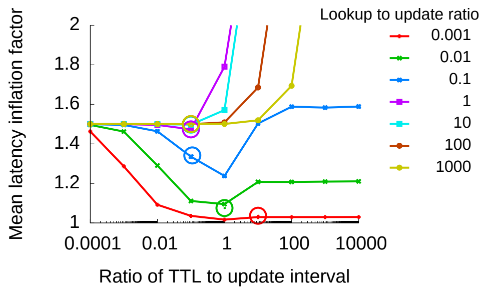


Figure 10: [Simulator] The circled dots show the optimal TTL values based on our model.

Figure 10 shows the mean latency inflation factor (y-axis) for several values of ratios of TTL to update interval (x-axis) and update to lookup ratios (each line). For each value of update to lookup ratio, we also show the optimal TTL based on our model, called *Model-Opt-TTL*, and the corresponding latency inflation factor. First, the minimum value of the latency inflation factor reduces from 1.5 (= l_2) to 1.0 (= l_1) as the update to lookup ratio reduces. When updates are much more frequent than lookups, nearly all lookups require the cache to contact an active replica; hence, the minimum latency inflation is close to 1.5. Conversely, when lookups are much more frequent than updates, suitably choosing a TTL value results in a high fraction of TTL-cache hits and hence minimum latency inflation is close to 1.0. Second, Model-Opt-TTL gives a latency inflation factor which is close the minimum latency inflation factor for each curve. Hence, our model helps select TTLs that are effective is effective in reducing the latency inflation factor.

4.8 Limitations and future work

Mid-connection mobility: In this paper, we have focused on reducing the time-to-connect through the global name service. However, a complete solution needs to also support mid-connection mobility, especially simultaneous mobility of both endpoints (as the mobility of just one endpoint can be handled bilaterally without relying on the global name service). We have developed a user-level socket library, `msocket`, that enables application developers to seamlessly develop network applications using Auspice at connection initiation time, using bilateral connection migration for individual mobility, and falling back on Auspice to handle simultaneous mobility. Under simultaneous mobility, we have shown that endpoints can re-establish a connection within ≈2RTTs of the latter endpoint coming back online. However, a detailed description of msocket is nontrivial and is described in a separate report [3].

Scope of evaluation: Our comparison of Auspice to managed DNS providers pits a pre-release prototype against opaque production services in live operation, an admittedly narrow comparison. Furthermore, we stand on loose footing with prototype-driven experiments involving 10^4 names and simulation experiments involving 10^6 names against DNS that serves 10^8 names and commands the luxury of decades of deployment experience, so our contribution is but academic at this point.

Estimating network distances: Auspice's placement engine relies on efficiently estimating network distances based on the network addresses of clients or local name servers. Our current implementation involves infrequently measuring ping latencies between name servers and local name servers; as part of ongoing work, we are evaluating other alternatives including using IP-to-geo distances or iPlane [49] in conjunction with EDNS [9] options to efficiently estimate network distances to both local name servers and originating end-hosts.

5 Related Work

Our work on Auspice draws on lessons learned from an enormous body of prior work on distributed systems for name resolution as well as more general services. Compared to this work, the novel contribution in Auspice is an engine to automate geo-distributed replica placement to achieve low-latency, low-cost, and high availability.

DNS: Until the early 80s, the Internet relied on a system called HOSTS.TXT for name resolution, which was simply a centrally maintained text file distributed to all hosts. The current Internet's distributed DNS [50] arose in response to the rapidly increasing size of the file and the cost of distributing it. Mockapetris and Dunlap point to TTL caching to reduce load and response times as a key strength, noting that "the XEROX system (Grapevine [58]) was then ... the most sophisticated name service in existence, but it was not clear that its heavy use of replication, light use of caching ... were appropriate". We have since come a full circle, turning to full replication in Auspice in order to address the challenges of mobility, a concern that wasn't particularly pressing in the 80s.

Since then, many have studied issues related to performance, scalability, load balancing, or denial-of-service vulnerabilities in DNS's resolution infrastructure [54, 56, 23, 29]. Several DHT-based alternatives have been put forward [56, 28, 57, 53] and we compare against a representative proposal, Codons [56]. In general, DHT-based designs are ideal for balancing load across servers, but are less well-suited to scenarios with a large number of service replicas that have to coordinate upon updates, and are at odds with scenarios requiring placement of replicas close to pocket of demand. In comparison, Auspice uses planned placement approach.

Server selection: A number of prior systems have addressed the server selection problem where data or services are replicated across a wide-area network. OASIS [34] maps users based on IP addresses to the best server based on latency and server load. DONAR [63] enables an expressive API for content providers to specify performance or cost optimization objectives under load balance constraints. These systems as well as CDNs and cloud hosting providers [2] share our goals of proximate server selection and load balance given a fixed placement of server replicas. Auspice differs in that it additionally considers replica placement itself as a degree of freedom in achieving latency or load balancing objectives.

Placement: Volley [18] optimizes the placement of dynamic data objects based on the geographic distribution of accesses to the object and is similar in spirit to Auspice in that respect. However, Volley implicitly assumes a single replica for each object and therefore does not have to worry about high update rates or coordination overhead for replica consistency. Auspice is also similar in spirit to edge services offered by commercial CDNs. However, the geo-distributed locations of edge services as well as cloud-hosted services today are chosen manually and infrequently updated. In comparison, Auspice automates geo-distributed placement of replicas, but the "service", a key-value store, is much simpler compared to black-box cloud-hosted services.

Data stores: Auspice is also related to many distributed key-value stores [13, 30, 48, 5], but most of these are optimized for distribution within, not across data centers. Some like Cassandra [5] support a geo-distributed deployment (but are rarely used in this mode) using a fixed number of replica sites. Spanner [27] is a geo-distributed data store that synchronously replicates data across datacenters and provides database like abstractions e.g., distributed transactions, semi-relational data model. Auspice does not provide any guarantees on operations spanning multiple records. In comparison to these systems, Auspice automatically determines the number and placement of replicas so as to reduce lookup latency and update cost.

6 Conclusions

In this paper, we presented the design, implementation, and evaluation of Auspice, a massively scalable, geo-distributed, global name service for an Internet where high mobility is the norm. The name service can resolve flexible identifiers (human-readable names, self-certifying identifiers, or arbitrary strings) to network locations or other attributes that can also be defined in a flexible manner. At the core of Auspice is a placement engine for replicating name records to achieve low lookup latency, low update cost, and high availability. Our evaluation shows that Auspice's placement algorithms significantly outperform both commercial managed DNS services employing simplistic replication strategies as well as previously proposed DHT-based replication alternatives. A pre-release version of Auspice on EC2 can be accessed through the developer portal at `http://gnrs.name`.

References

[1] Alexa web information service. `http://www.alexa.com`.

[2] Amazon elastic load balancing. `http://aws.amazon.com/elasticloadbalancing/`.

[3] Application Level End-to-End Seamless Migration Support for Mobile Hosts. `http://bit.ly/1bD7WKx`.

[4] CAP theorem. `http://en.wikipedia.org/wiki/CAP_theorem`.

[5] Cassandra. `http://cassandra.apache.org`.

[6] Cisco visual networking index: Global mobile data traffic forecast update, 2012-2017.

[7] Dns made easy. `http://www.dnsmadeeasy.com`.

[8] Dyn dns. `http://dyn.com/`.

[9] Extension Mechanisms for DNS. `http://tools.ietf.org/html/rfc6891`.

[10] Gnrs portal. `http://gnrs.name`.

[11] GSM Technical Specifications. GSM UMTS 3GPPNumbering Cross Reference. ETSI. Retrieved 30 December 2009.

[12] MobilityFirst Future Internet Architecture Project. `http://mobilityfirst.cs.umass.edu/`.

[13] mongodb. `http://www.mongodb.org/`.

[14] Server fault: DNS - Any way to force a name server to update the record of a domain? `http://serverfault.com/questions/41018/dns-any-way-to-force-a-nameserver-to-update-the-record-of-a-domain`,.

[15] The Locator/ID Separation Protocol (LISP). RFC 6830.

[16] Ultra dns. `http://www.neustar.biz/`.

[17] Comparison and analysis of managed dns providers, Aug 2012. Cloud Harmony Inc.

[18] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: automated data placement for geo-distributed cloud services. NSDI'10.

[19] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable internet protocol (aip). SIGCOMM, 2008.

[20] M. Arye, E. Nordstrom, R. Kiefer, J. Rexford, and M. J. Freedman. A formally-verified migration protocol for mobile, multi-homed hosts. In *ICNP*, 2012.

[21] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *SIGCOMM*, 2004.

[22] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *MobiSys*.

[23] N. Brownlee, K. Claffy, and E. Nemeth. Dns measurements at a root server. In *GLOBECOM '01. IEEE*, 2001.

[24] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. ROFL: routing on flat labels. *SIGCOMM*.

[25] I. Castineyra, N. Chiappa, and M. Steenstrup. Nimrod routing architecture. RFC 1992.

[26] D. R. Cheriton and M. Gritter. Triad: a scalable deployable nat-based internet architecture. Technical report, January 2000.

[27] J. C. Corbett, J. Dean, M. Epstein, and et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 2013.

[28] R. Cox, A. Muthitacharoen, and R. Morris. Serving dns using a peer-to-peer lookup service. In *IPTPS*, 2002.

[29] DNSSEC. DNS Threats & Weaknesses of the Domain Name System, 2012. `http://www.dnssec.net/dns-threats.php`.

[30] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[31] A. Feldmann, L. Cittadini, W. Mühlbauer, R. Bush, and O. Maennel. Hair: hierarchical architecture for internet routing. ReArch '09, 2009.

[32] B. Ford. Unmanaged internet protocol: Taming the edge network management crisis. In *HotNets-II*, 2003.

[33] B. Fortz and M. Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM*, 2000.

[34] M. J. Freedman, K. Lakshminarayanan, and D. Mazires. Oasis: Anycast for any service, 2006.

[35] D. Funato, K. Yasuda, and H. Tokuda. Tcp-r: Tcp mobility support for continuous operation. ICNP '97.

[36] H. Gao, V. Yegneswaran, Y. Chen, P. Porras, S. Ghosh, J. Jiang, and H. Duan. An empirical re-examination of global dns behavior. In *SIGCOMM*, pages 267–278, New York, NY, USA, 2013. ACM.

[37] Gartner. Mobile Connections Will Reach 5.6 Billion in 2011, 2011. `http://www.gartner.com/it/page.jsp?id=1759714`.

[38] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. Xia: efficient support for evolvable internetworking. NSDI'12.

[39] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. CoNEXT '09.

[40] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus routing: the internet as a distributed system. NSDI'08.

[41] P. Jokela, P. Nikander, J. Melen, J. Ylitalo, and J. Wall. Host identity protocol, extended abstract. In *Wireless World Research Forum*, 2004.

[42] L. Kleinrock. In *Queueing Systems Vol 1: Theory. p. 77*, 1975.

[43] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. SIGCOMM '07.

[44] D. Krioukov, k. c. claffy, K. Fall, and A. Brady. On compact routing for the internet. *SIGCOMM CCR'07.*

[45] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, 2010.

[46] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[47] L. Lamport, D. Malkhi, and L. Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

[48] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[49] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iplane: an information plane for distributed services. In *OSDI*, pages 367–380, Berkeley, CA, USA, 2006. USENIX Association.

[50] P. Mockapetris. Domain names - concepts and facilities. *The Internet Society*.

[51] E. Nordstrom, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Y. Ko, J. Rexford, and M. J. Freedman. Serval: An end-host stack for service-centric networking. In *NSDI*, 2012.

[52] NSF. MobilityFirst Future Internet Architecture Project, 2012. `http://mobilityfirst.winlab.rutgers.edu//`.

[53] V. Pappas, D. Massey, A. Terzis, and L. Zhang. A comparative study of the dns design with dht-based alternatives. In *INFOCOM*, 2006.

[54] V. Pappas, Z. Xu, S. Lu, D. Massey, A. Terzis, and L. Zhang. Impact of configuration errors on dns robustness. SIGCOMM, 2004.

[55] C. E. Perkins. Mobile IP. *IEEE Comm. Magazine*, May 1997.

[56] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. SIGCOMM '04.

[57] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI*, 2004.

[58] M. D. Schroeder, A. D. Birrell, and R. M. Needham. Experience with grapevine: the growth of a distributed system. *ACM Trans. Comput. Syst.*

[59] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MobiCom*, 2000.

[60] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, 2002.

[61] A. Venkataramani, X. Tie, A. Sharma, D. Westbrook, H. Uppal, J. Kurose, and D. Raychaudhuri. Design guidelines for a global name service for a mobility-centric, trustworthy internetwork. *COMSNETS*, 2013.

[62] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.

[63] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford. Donar: decentralized server selection for cloud services. SIGCOMM, 2010.

[64] M. P. Wittie, V. Pejovic, L. Deek, K. C. Almeroth, and B. Y. Zhao. Exploiting locality of interest in online social networks. CoNEXT, 2010.

A Optimization Formulation

All variables used in this formulation are described in Table 4. Let D be the set of locations of name servers, and C_d the total capacity of the name server at location $d \in D$. Users requesting the name records are partitioned into distinct, geographically distributed network regions or user-groups U. The user-groups are assumed to be fine-grained enough so that the latency from any member of a user-group $u \in U$ to a name server d is close to the average latency L_{ud} from users in u to d.

The system decides the placement of resolvers and also decides to which resolver to redirect each user. A user's request is assumed to be serviceable from any of the resolvers. If a resolver of name record $s \in S$ is placed at location $d \in D$, the corresponding (binary) decision variable x_{sd} takes the value 1, otherwise x_{sd} equals zero. The volume of requests from user-group $u \in U$ to the replica (if any) at location $d \in D$ of a service $s \in S$ is denoted by y_{uds}, a decision variable that takes values between 0 and r_{us}.

Minimizing the average latency can be formulated as a mixed integer program. The following objective minimizes the aggregate latency across all users' requests. M_d is the total server latency at name server $d \in D$. The first term and the second term denote the aggregate network and server latency respectively.

$$\text{minimize: } \sum_{s\in S}\sum_{d\in D}\sum_{u\in U} L_{ud}\, x_{ds}\, y_{uds} + \sum_{d\in D} M_d \tag{2}$$

The optimization must satisfy the constraints of the problem specified from Equation (5) to Equation (14).

All users' requests must be satisfied.

$$\sum_{d\in D} y_{uds} = R_{us} \quad \forall u \in U, s \in S \tag{3}$$

The capacity at each name server must be greater than the total request rate of users' and the update rate of name records placed at that location. The intermediate variable t_d is the total request rate at name server $d \in D$.

$$\sum_{s\in S}\sum_{u\in U} y_{uds} + \sum_{s\in S} W_s\, x_{ds} = t_d \le C_d \quad \forall d \in D \tag{4}$$

Server utilization at $d \in D$ is t_d/C_d. Server latency per request is defined as a function of server utilization. The function f is a piecewise convex linear function defined as $f(0) = 0$ and its derivatives.

$$f'(t_d/C_d) = \begin{cases} r_1 & \text{if } 0 \le t_d/C_d \le u_1, \\ r_2 & \text{if } u_1 < t_d/C_d \le u_2, \\ \dots & \\ r_j & \text{if } u_{j-1} < t_d/C_d \le 1 \end{cases} \tag{5}$$

Essentially, the above equations transform a load vs. response time curve to a piecewise-linear, convex function, a technique that has also been in used in other domains [33] to make the optimization linear. Let M_d be the total server latency at location $d \in D$. M_d is defined by the following set of equations.

$$v_0 = 0, u_0 = 0, u_j = 1 \tag{6}$$

$$M_d \ge v_{i-1} + r_i\,(t_d - u_{i-1}\, C_d) \quad \forall i \in \{1,2,...,j\} \tag{7}$$

$$v_i = v_{i-1} + r_i\,(u_i - u_{i-1})\, C_d \quad \forall i \in \{1,2,...,j\} \tag{8}$$

To ensure availability, each name record should be replicated at B locations or more.

$$\sum_{d\in D} x_{ds} \ge B \quad \forall s \in S \tag{9}$$

A request can be served from a name server only if a resolved is placed at that name server.

$$y_{uds} \le x_{ds} R_{us} \quad \forall u \in U, d \in D, s \in S \tag{10}$$

The next two equations constrain the values of each variable.

$$x_{ds} \in \{0,1\} \quad \forall d \in D, s \in S \tag{11}$$

$$0 \le y_{uds} \le R_{us} \quad \forall u \in U, d \in D, s \in S \tag{12}$$

B TTL Cache Model

In this section we propose a simple Markov model to determine the TTL value of a DNS record so that the expected delay of a read request is minimized. We make several simplifying assumptions which enable us to determine a closed form expression for the TTL. The purpose of this model based approach is to provide ballpark values which can be utilized by network operators to set TTL values for the DNS records.

We consider a local name server with only a single DNS record. Hence all read requests at this local name server are for this particular DNS record. Every read request arriving into the system can result in a cache hit or a cache miss. But unlike regular caching systems, misses here can be of two types - one due to TTL expiration of a record and the other due to stale state information. Note that stale state information at a local

Parameters	
U	Set of geographic regions spanning all users
D	Set of name servers
S	Set of all name records
C_d	Capacity of name server $d \in D$
L_{ud}	Average latency between users in region $u \in U$ and name server location $d \in D$
R_{us}	Lookup query rate of name record $s \in S$ from users in region $u \in U$
W_s	Update query rate of name record $s \in S$
B	Minimum number of resolvers of each name
α	Replication parameter for all name records
Variables	
x_{ds}	Binary variable indicating whether name record $s \in S$ is replicated at $d \in D$.
y_{uds}	At location $d \in D$, lookup rate of users $u \in U$ for name record $s \in S$

Table 4: Optimal parameters and variables

name server results from the corresponding entry at the global name server being modified.

If the read request arrives before TTL expiration and the state of the record at the local and global name servers are the same, then the read request results in a Hit. If the read request arrives after TTL expiration, we refer to it as a Type I Miss. However, if the read request arrives before TTL expiration and the state of the record at the local and global name servers are different, then the read request results in a Type II Miss. The delays experienced in these three situations are different. Let δ_H, δ_{M_1} and δ_{M_2} be the delays incurred due to a Hit, Type I Miss and Type II Miss respectively.

We denote the state of the system by a result of the last arrival i.e., Hit, Type I Miss or Type II Miss. Let H, M_1 and M_2 denote the state of the system i.e, the last arrival was a Hit, Type I Miss or Type II Miss respectively. We assume that the time between two successive updates (U) to the DNS record at the global name server is exponentially distributed with rate λ_U. Similarly we assume that the TTL (denoted by T) is also exponentially distributed with rate λ_T. The memoryless property of the exponential distribution enables us to model the system by a Markov chain.

Let X denote the time between two successive read requests. Let $f_X(x)$ denote the distribution of X. Because U and T are modeled as an exponential distribution, once we observe the system, i.e., a read request arrives, the time for the next update and the time left for the TTL to expire are still exponential with the same parameters as before. Irrespective of the current state of the system, the probability that the next read request will result in the system transitioning to state M_1 (P_{M_1}) is given by $P[X > T]$. Similarly, probability of transitioning to state H (P_H) and M_2 (P_{M_2}) are $P[X < T \; \& \; X < U]$ and $P[X < T \; \& \; X > U]$ respectively. Note that the transition probabilities can be calculated (at least numerically) for different distributions of X. As our goal here is to provide ballpark values for the TTL we assume X is also exponentially distributed with rate λ_R. P_H, $P_{M_1}$and P_{M_2} are also the steady state probabilities of being in state H, M_1 and M_2 respectively. It can be easily shown that with the above assumptions,

$$P[H] = \frac{\lambda_R}{\lambda_T + \lambda_R + \lambda_U} \tag{13}$$

$$P[M_1] = \frac{\lambda_T}{\lambda_T + \lambda_R} \tag{14}$$

$$P[M_2] = \frac{\lambda_R \lambda_U}{(\lambda_T + \lambda_R)(\lambda_T + \lambda_R + \lambda_U)} \tag{15}$$

Let D denote the delay experience by a read request. Then we can show that,

$$E[D] = \delta_H \frac{\lambda_R}{\lambda_T + \lambda_R + \lambda_U} + \delta_{M_1} \frac{\lambda_T}{\lambda_T + \lambda_R} + \delta_{M_2} \frac{\lambda_R \lambda_U}{(\lambda_T + \lambda_R)(\lambda_T + \lambda_R + \lambda_U)} \tag{16}$$

To determine the value of λ_T which minimizes the $E[D]$ we differentiate (16) w.r.t. λ_T and equate it to 0. In this paper we assume that $\frac{\delta_{M_1}}{\delta_H} = 1.5$ and $\frac{\delta_{M_1}}{\delta_H} = 4$. For the above values, the possible value of λ_T which minimizes (16) is given by (17) (the second differential is greater than 0 in most cases of interest).

$$\lambda_T = -\lambda_R + 10.47\lambda_U \tag{17}$$

λ_T in reality has to be greater than 0. If value of λ_T given by (17) is less than zero, then the minimum expected delay occurs for $\lambda_T = 0$ or $\lambda_T = \infty$. Note that if the read request interval distribution for every DNS record at every local name server is known, it is possible to set different TTL values for records based on both popularity and locality thereby providing additional flexibility to the network operator. In the current system TTL values are set only based on the popularity and not on the locality.

One of the major differences between the real system and the above model is the exponentially distributed TTL assumption. In the real system each name has TTL set of a fixed value. But we will observe from our evaluation that the exponential assumption does not hurt us badly and the value of λ_T determined by our model is successful in providing useful ballpark values.

Lemma 1 $P[M_1] = P[X > T] = \frac{\lambda_T}{\lambda_T + \lambda_R}$

$$\begin{aligned} P[X > T] &= \int_0^\infty P[X > T | T = t] \lambda_T e^{-\lambda_T t} dt \\ &= \int_0^\infty P[X > t] \lambda_T e^{-\lambda_T t} dt \\ &= \int_0^\infty e^{-\lambda_R t} \lambda_T e^{-\lambda_T t} dt \\ &= \lambda_T \int_0^\infty e^{-(\lambda_R + \lambda_T)t} dt \\ &= \frac{\lambda_T}{\lambda_R + \lambda_T} \end{aligned} \tag{18}$$

Lemma 2 $P[H] = P[X < T \ \& \ X < U] = \frac{\lambda_R}{\lambda_T+\lambda_R+\lambda_U}$

$$\begin{aligned}
P[X < T \ \& \ X < U] &= \int_0^\infty P[X > T \ \& \ X < U | X = x] \lambda_R e^{-\lambda_R x} dx \\
&= \int_0^\infty P[T > x \ \& \ U > x] \lambda_R e^{-\lambda_R x} dx \\
&= \int_0^\infty e^{-\lambda_T x} e^{-\lambda_U x} \lambda_R e^{-\lambda_R x} dx \\
&= \lambda_R \int_0^\infty e^{-(\lambda_R+\lambda_T+\lambda_U)x} dx \\
&= \frac{\lambda_R}{\lambda_R+\lambda_T+\lambda_U} \qquad (19)
\end{aligned}$$

Lemma 3 $P[M_2] = P[X < T \ \& \ X > U] = \frac{\lambda_R \lambda_U}{(\lambda_T+\lambda_R)(\lambda_T+\lambda_R+\lambda_U)}$

$$\begin{aligned}
P[M_2] &= 1 - P[H] - P[M_1] \\
&= 1 - \frac{\lambda_T}{\lambda_T+\lambda_R} - \frac{\lambda_R}{\lambda_R+\lambda_T+\lambda_U} \\
&= \frac{(\lambda_T+\lambda_R)(\lambda_R+\lambda_T+\lambda_U) - \lambda_T(\lambda_R+\lambda_T+\lambda_U) - \lambda_R(\lambda_T+\lambda_R)}{(\lambda_T+\lambda_R)(\lambda_R+\lambda_T+\lambda_U)} \\
&= \frac{\lambda_R \lambda_U}{(\lambda_T+\lambda_R)(\lambda_R+\lambda_T+\lambda_U)} \qquad (20)
\end{aligned}$$

Lemma 4 $\lambda_T = -\lambda_R + 10.47\lambda_U$

$$\frac{E[D]}{d\lambda_T} = \lambda_R \left(\frac{\delta_{M_2} - \delta_H}{(\lambda_R+\lambda_T+\lambda_U)^2} - \frac{\delta_{M_2} - \delta_{M_1}}{(\lambda_R+\lambda_T)^2} \right) \qquad (21)$$

By equating (21)=0 we have,

$$(\delta_{M_1} - \delta_H)(\lambda_T+\lambda_R)^2 - 2(\delta_{M_2} - \delta_{M_1})\lambda_U(\lambda_T+\lambda_R) - \lambda_U^2(\delta_{M_2} - \delta_{M_1})^2 = 0 \qquad (22)$$

Therefore we have,

$$\lambda_T = -\lambda_R + \lambda_U \frac{(\delta_{M_2} - \delta_{M_1}) \pm \sqrt{(\delta_{M_2} - \delta_{M_1})(\delta_{M_2} - \delta_H)}}{(\delta_{M_1} - \delta_H)} \qquad (23)$$

Substituting the values of δ_H, δ_{M_1}, δ_{M_2} we have, the only possible candidate for λ_T as

$$\lambda_T = -\lambda_R + 10.47\lambda_U \qquad (24)$$

The second differential is given by

$$\frac{d^2E[D]}{d\lambda_T} = 2\lambda_R \left(\frac{\delta_{M_2} - \delta_{M_1}}{(\lambda_R+\lambda_T)^3} - \frac{\delta_{M_2} - \delta_H}{(\lambda_R+\lambda_T+\lambda_U)^3} \right) \qquad (25)$$

For λ_T given by (23) to be a minima, we have to show that

$$(\delta_{M_2} - \delta_{M_1})(\lambda_R+\lambda_T+\lambda_U)^3 - (\delta_{M_2} - \delta_H)(\lambda_R+\lambda_T)^3 > 0 \qquad (26)$$